Representational Redundancy Reduction Strategies for Efficient Neural Network Architectures for Visual and Language Tasks

by

Rene Bidart

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Systems Design Engineering

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Stefan C. Kremer Professor, Computer Science, University of Guelph
Supervisor:	Alexander Wong Professor, Systems Design Engineering, University of Waterloo
Internal Member:	John Zelek Associate Professor, Systems Design Engineering, University of Waterloo
Internal Member:	Bryan Tripp Associate Professor, Systems Design Engineering, University of Waterloo
Internal-External Member:	Jeff Orchard Associate Professor, Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Chapter 3 contains material from two papers which are authored by myself and my supervisor, Alexander Wong. For both papers I was the lead author, and was responsible for designing the model, designing experiments, the implementation and training of models, and drafting and submitting the manuscripts. My supervisor Prof. Wong provided guidance throughout the process as well as assisting in editing the papers. The two papers are *Affine Variational Autoencoders: An Efficient Approach for Improving Generalization and Robustness to Distribution Shift*[15] and *Disentangling Shape and Orientation with Affine Variational Autoencoders*[16]

Abstract

Deep neural networks have transformed a wide variety of domains including natural language processing, image and video processing, and robotics. However, the computational cost of training and inference with these models is high, and the rise of unsupervised pretraining has allowed ever larger networks to be used to further improve performance. Running these large neural networks in compute constrained environments such as on edge devices is infeasible, and the alternative of doing inference using cloud compute can be exceedingly expensive, with the largest language models needing to be distributed across multiple GPUs.

Because of these constraints, size reduction and improving inference speed has been a main focus in neural network research. A wide variety of techniques have been proposed to improve the efficiency of existing neural networks including pruning, quantization, and knowledge distillation. In addition there is extensive effort on creating more efficient networks through hand design or an automated process called neural architecture search. However, there remain key domains where where there is significant room for improvement, which we demonstrate in this thesis.

In this thesis we aim to improve the efficiency of deep neural networks in terms of inference latency, model size and latent representation size. We take an alternative approach to previous research and instead investigate redundant representations in neural networks. Across three domains of text classification, image classification and generative models we hypothesize that current neural networks contain representational redundancy and show that through the removal of this redundancy we can improve their efficiency.

For image classification we hypothesize that convolution kernels contain redundancy in terms of unnecessary channel wise flexibility, and test this by introducing additional weight sharing into the network, preserving or even increasing classification performance while requiring fewer parameters. We show the benefits of this approach on convolution layers on the CIFAR and Imagenet datasets, on both standard models and models explicitly designed to be parameter efficient.

For generative models we show it is possible to reduce the size of the latent representation of the model while preserving the quality of the generated images through the unsupervised disentanglement of shape and orientation. To do this we introduce the affine variational autoencoder, a novel training procedure, and demonstrate its effectiveness on the problem of generating 2 dimensional images, as well as 3 dimensional voxel representations of objects.

Finally, looking at the transformer model, we note that there is a mismatch between the tasks used for pretraining and the downstream tasks models are fine tuned on, such as text classification.We hypothesize that this results in a redundancy in terms of unnecessary spatial information, and remove it through the introduction of learned sequence length bottlenecks. We aim to create task specific networks given a dataset and performance requirements through the use of a neural architecture search method and learned downsampling. We show that these task specific networks achieve superior performance in terms of inference latency and accuracy tradeoff to standard models without requiring additional pretraining.

Acknowledgements

I would like to thank my supervisor Alexander Wong for his guidance and support, especially during the more frustrating parts of this PhD. Also thanks to everyone in the VIP lab for keeping things interesting with research, and the many interesting discussions. I am always grateful to my parents because they've done everything for me, and my grandparents, without their work in the coal mines and steel plants I wouldn't have the chance to work on AI today.

Table of Contents

List of Figures

Chapter 1

Introduction

Deep neural networks have achieved impressive performance across a wide variety of domains including natural language, images and video, but their large size makes inference costly, limiting their use for practical applications such as mobile phones or edge devices. This issue has grown in recent years as models are becoming progressively larger to achieve improved performance on more difficult problems. In this work we explore methods for reducing representational redundancy in deep neural networks with the downstream goal of improving inference latency and reducing size.

1.1 Problem Definition

Since a convolutional neural network[81] won the Imagenet[39] competition in 2012 neural networks have progressively taken over more domains as the dominant approach. For problem after problem the best approach changed from using hand designed features and statistical learning to an end to end approach using neural networks with learned features.

Instead of designing feature extraction manually, the dominant approach became to design a particular neural network for each task which automatically learns features using gradient descent. This approach worked across a wide variety of domains. For example, fully convolutional neural networks were used for semantic segmentation[100], a casual convolution based wavenet for audio generation[113], generative adversarial networks[52] and variational autoencoders[78] for image generation, and a combination of recurrent neural networks and attention for machine translation[104]. While these approaches consistently improved performance over the older methods, the downside was that they were much more

computationally expensive, making deployment to time sensitive or low power applications such as edge devices difficult. As research progressed these issues became worse, as more performance could be gained using ever larger models at a rate that surpassed performance improvements from better hardware.

A new paradigm emerged with the introduction of unsupervised pretraining[67, 124, 120, 117], and the transformer[163]. Unsupervised pretraining allowed unlabelled datasets to be used to improve performance of models which could later be fine tuned on smaller labelled datasets. While this approach was not new in machine learning, having been used for n-grams[17], word embeddings[106, 116], image classification[10] and even text classification[67], the true potential of pretraining was not discovered until it was used in conjunction with a flexible model like the transformer[163, 42], where it achieved dramatically better performance on a variety of Natural Language Processing (NLP) tasks. These developments were not limited to NLP as in the original transformer model, and it was found that variants of the transformer were also shown to be superior on a wide variety of tasks, including image classification[186], image segmentation[167], language modelling[18], audio compression[38], speech recognition[192, 119] and a variety of text-image tasks[167].

These flexible models were uniquely able to leverage massive datasets to generate performance improvements, leading to a paradigm shift where model size, data and computation could be scaled to directly improve performance[73]. Progressively more flexible architectures would be trained using more data to enable less human design at the cost of more computation, data and model complexity[156]. Larger models perform better, but are also more computationally expensive, causing a variety of issues related to inference latency in time sensitive applications, power consumption in low power edge devices and high energy consumption, resulting in high costs and negative effects on climate[83]. The largest and best performing language models such as GPT3[18] take scaling to the extreme, containing 175 billion parameters, and require inference to be distributed across 8 Nvidia A100 GPUs[41], each costing about $10,000USD. This is because the model uses half precision, so to store the model weights alone takes 350GB. The overhead of the intermediate representations should add another 20% to this, so we would require over 400GB of memory in total.

The issue of model complexity has been a focus of research efforts for many years, and even before the rise of deep learning there were investigations into improving the efficiency of more shallow neural networks[85]. In fact, the end of the AI winter came through the utilization of graphics processing unit (GPU) acceleration to overcome compute bottlenecks in neural network training[27, 81]. GPUs were designed to do massive parallel computation of simple functions for graphics, which is similar to the large matrix multiplications used in deep neural networks. Since then, the available compute has increased massively[138],

and benchmarks that took weeks to train can now be done in minutes[31]. Yet we have scaled model size even faster so the same problems with computational complexity remain, and efficiency remains a focus in deep learning research.

There are a variety of ways in which neural networks are limited by compute constraints and can be made more efficient. For example, during the training process better optimizers[76, 145, 101] can be used to speed up convergence, data augmentation[190, 43, 34, 197] or active learning[165, 47, 183] can reduce the number of labelled examples needed. Alternatively the hardware itself can be optimized[107], as well as the development of methods for improving distributed training across multiple GPUs[166, 122]. In this work we instead focus on reducing redundancy in the neural network's representations, with the end goal of reducing the model size and increasing the speed of inference.

The two goals of small model size and fast inference are closely linked, as smaller models tend to be faster, but this is not a consistent relationship[63] as models with the same number of parameters can require wildly different numbers of floating point operations (FLOPS), can be bottlenecked by memory or not have optimized software for execution on a particular hardware. For this reason in this work we will be looking at both these objectives to measure the downstream effects of our approach of reducing representational redundancy.

1.2 Contributions

How can we construct deep neural networks that are more efficient in terms of inference speed and model size while preserving their performance? Is there representational redundancy in these networks that can be reduced to achieve these goals? In this work we will approach these questions in three different domains, image classification with CNNs, generative models with variational autoencoders, and text classification with transformer models. While there has been extensive work on this problem, we see opportunities for extension by reducing representational redundancy through the use of additional weight sharing in CNNs, shape and orientation disentanglement in variational autoencoders, and learned downsampling in transformer models, shown in Figure 1.1.

- *Additional Weight Sharing in Convolutional Neural Networks* - Our work on CNNs is motivated by the fact that many of the efficient hand designed architectures use special convolution layers with additional sparsity and weight sharing. In particular, depth wise separable[148][25], dilated[185] and lightweight convolutions[175] all can

be viewed as the introduction of sparsity or weight sharing on a standard convolution layer. Weight sharing not only reduces the network size in terms of parameters, but also allows the model to be trained normally, with the benefit of fewer independent parameters to be learned. While previous research has been effective in removing redundant parameters through pruning[85][89], or designing more efficient architectures more generally[159], we hypothesize that there remains additional representational redundancy within the convolution layer, and take advantage of this through the introduction of additional weight sharing into convolutions. We show this can decrease model size while preserving or even increasing performance on large scale image classification tasks such as Imagenet[39].

- *Efficient Variational Autoencoders through Disentanglement of Shape and Orientation* - We approach an unsupervised learning problem with the aim of creating more compact representations through the disentanglement of shape and orientation. While there has been extensive previous work on disentangling representations, research has focused on the more straightforward task where labels for the factors of variation are available[77, 130, 147, 82, 40], so it is not truly unsupervised disentanglement. We instead focus on the task of disentanglement where no labels are available, in this case referring to the orientation of objects. We extend the variational autoencoder[78] to perform unsupervised disentanglement of shape and orientation through the addition of two affine transform layers into the model, along with an optimization procedure enabling this disentanglement. We show this both results in a more compact representation of the input, but also learns a more interpretable latent space. This is demonstrated both for 2D images using 2D affine transforms, as well as for 3D objects using 3D affine transforms.

- *Fast Transformer Inference by Task Specific Learned Downsampling* - Finally, we approach the most recent and computationally intensive deep neural network architecture, the transformer[163]. While there has been extensive work in efficient transformer research[149, 22, 180, 97, 194, 136, 70, 155, 36, 153, 111, 53, 69], our contribution distinguishes itself by addressing the redundancy arising from the mismatch between pretraining tasks and the specific downstream tasks like text classification these networks are commonly used for. Unlike previous work such as evolution-based neural architecture search [149] and manual design approaches like DistilBERT [136], MobileBERT [155], and Funnel Transformer [36], our approach introduces a unique concept of learned sequence length bottlenecks adapted to a specific dataset, performance needs, and hardware. This is accomplished through a neural architecture search based approach, where we use Bayesian optimization to maximize inference

speed subject to a constraint on accuracy. By focusing on text classification tasks, our work sets a new benchmark, demonstrating a superior tradeoff between inference speed and accuracy, thereby advancing the efficiency frontier in transformer research [163].

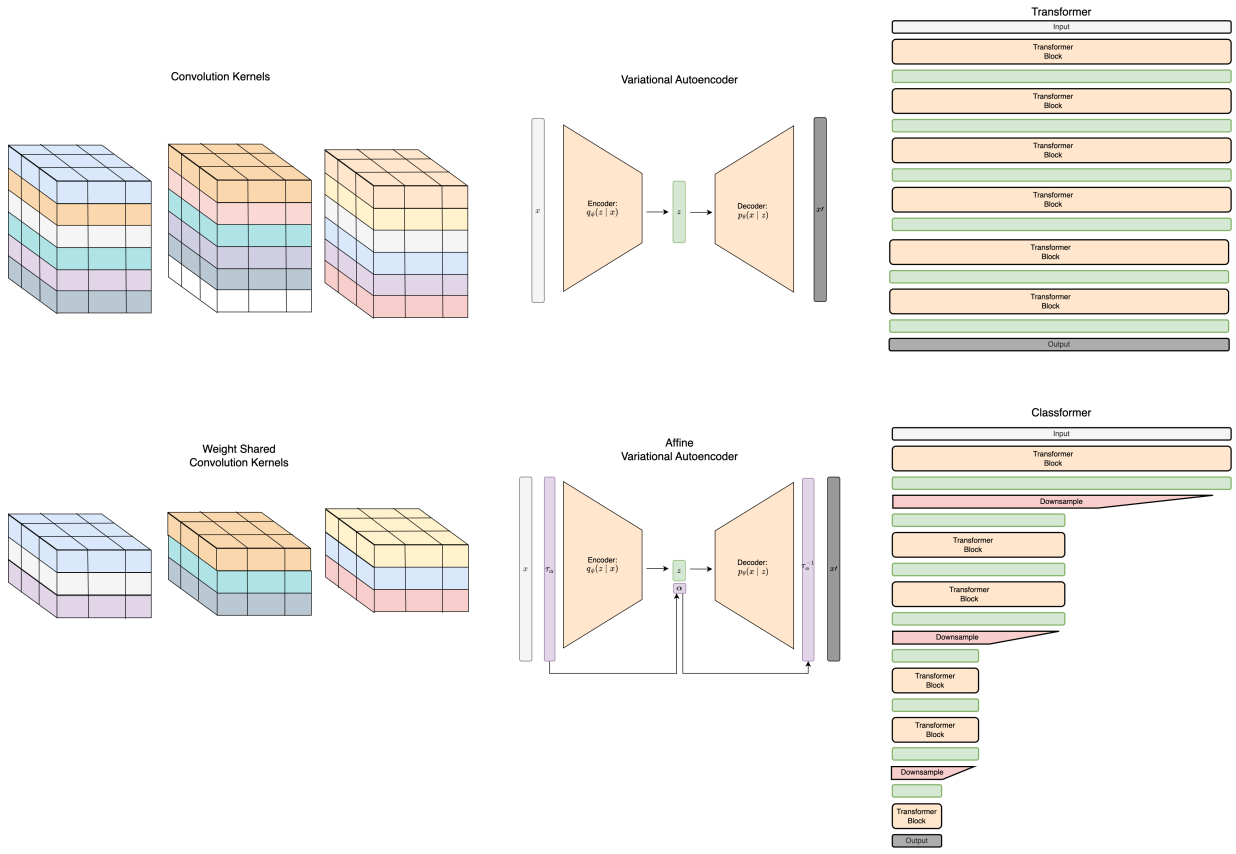


Figure 1.1: Visualizing the three proposed methods for redundancy reduction within neural networks. The first method, on the left, is the use of additional weight sharing in convolutional kernels. In the middle we see the Affine Variational Autoencoder, designed to disentangle orientation and shape to create a compressed latent representation. On the right is the Classformer, a method to learn task specific architectures for classification tasks through learned sequence length bottlenecks.

1.3 Thesis Outline

We first introduce background information and previous research relevant to the subsequent chapters in Chapter 2. This includes a review of architecture search in Section 2.1 and pruning and quantization in Section 2.2, as well as a review of the connections between these. Background relevant for generative models, specifically variational autoencoders along with work on disentangled representations is reviewed in Section 2.3. In Section 2.4, we review the transformer architecture and previous work on efficient transformers.

In Chapter 3 we investigate how the use of additional weight sharing in convolutional kernels can remove representational redundancy. We show this method can improve performance on smaller datasets, and can also scale to large datasets.

In Chapter 4 we demonstrate a method to create more compact latent representations in variational autoencoders through the disentanglement of orientation and shape. We show this can be used to create more compact and interpretable representations, creating more efficient generative models.

In Chapter 5 we extend transformer based models for text classification through the use of learned downsampling, removing redundant sequence length information through the use of task specific architectures. We show this can be used to improve inference speed, and can also be used to tailor the architecture to a particular dataset, hardware and performance requirements.

Chapter 2

Background

In this chapter, we review relevant background for our goal of removing representational redundancy in neural networks for applications in image classification, generative modelling, and text classification. We will review methods for creating more parameter efficient convolutional neural networks (CNN) through architecture search and neural network compression through sparsity and quantization of the weights. For generative models we begin with an introduction to Variational Autoencoders (VAE) as well as the most relevant research for improving representations, particularly the research focused on disentangling representations. In addition, we will review transformers and methods to increase efficiency both through computationally efficient layers as well as in the architecture at a macro level.

2.1 Neural Architecture Search

The success of deep learning proved the superiority of learned features compared to hand design, as was the norm with classical machine learning methods. A natural next step was to learn the architecture of the network, called Neural Architecture Search (NAS), instead of only learning the model's weights. In this section we provide a general overview of approaches to architecture search, provide detailed review of key research in this area, as well as investigating the limitations of current NAS methods.

2.1.1 Overview of Neural Architecture Search

Neural Architecture Search methods can be decomposed across three dimensions[46], (i) **Search Space**, (ii) **Search Strategy**, (iii) **Performance Estimation Strategy**. While these tasks are not totally independent (e.g. constraining the search space may result in alternative performance estimation strategies), this is a practically useful way to break down the problem.

Search Space

The search space is the set of all networks discoverable by the NAS method. Performance of NAS methods are heavily dependent on good search space design to make the problem tractable because the space of all possible neural networks is massive. In practice, even limiting the search space to particular type of neural network, such as convolutional neural networks, is not enough to make this efficiently searchable by known methods. Current NAS methods use stronger restrictions on the search space, such as rearranging a number of building blocks and their associated hyperparameters that are known to perform well, including dilated convolutions[185], separable convolutions[148][25], as well as pooling and identity layers. Other work has investigated more general architectures in the form of fully connected networks[7][151], but the focus of this work will be on NAS methods that attain strong performance on standard computer vision tasks, all of which require this constrained search space. Common search spaces used in NAS are explained below:

1. **Flexible Chain Structured Networks**[198], where the network can be written as a sequence of layers, possibly incorporating skip connections between layers. In this case there is no fixed high level structure of the network, allowing for the entire network to be constructed based on pre-defined building blocks, but with flexible ordering and connections.

2. **Cell Based Networks**[199], where instead of searching over the entire architecture, the higher level architecture is fixed, and the search takes place over cells, which are then stacked together in a pre-defined way to form the full network. Cells refer to small neural networks which are combined into a single, larger network. The higher level architecture is hand defined, and the search happens within the cells. The benefit of this is both the reduced search space, as well as the ability to scale up a learned architecture to a different size by using a different higher level structure. This method has a key limitation that the same blocks are repeated throughout the

network, even though it may be optimal to use a different structure in different parts of the network.

3. **Other Search Spaces** have also been investigated, including hierarchical search spaces, where the network decides both the lower level cell, as well as some control over the higher level structure of the network[92]. Other work has investigated relaxing the structure even further by searching over more flexible connections between convolution layers without a fixed high level architecture or the restriction of a chain structured network[179].

Search Strategy

A NAS method needs some way to explore the search space and find architectures with high performance. Some common approaches are evolution, reinforcement learning, or random search. While initial approaches framed this problem in terms of reinforcement learning[198][199], there was a resurgence of evolution based methods when it was discovered they outperform reinforcement learning based approaches in terms of training time. Interestingly, both of these approaches only outperform random search by a small margin of about 0.5% on CIFAR10[127].

Performance Estimation Strategy

The performance of the networks created by the search strategy must be evaluated, but the standard method for evaluation of a discovered architecture is training the architecture to full accuracy, which is extremely expensive. Alternative forms of performance evaluation have been proposed to overcome this bottleneck. For example, some methods share weights across models[96][118], using another model to estimate performance on an untrained model[93], while other methods estimate performance after the network is only partially trained[127].

2.1.2 Key Approaches to Neural Architecture Search

Reinforcement Learning Based NAS

Architecture search can be framed as a reinforcement learning problem where we construct an agent that aims to generate an architecture to maximize some reward, generally the

classification accuracy. This formulation allows reinforcement learning algorithms to be used even though architecture search is different than most reinforcement learning applications because NAS lacks a changing state. This means NAS is more similar to a stateless multi-arm bandit problem[46]. Nevertheless, these approaches have shown some usefulness for this problem, which we describe below.

Neural Architecture Search was first popularized with a search strategy based on reinforcement learning[198]. They used a recurrent neural network, called the controller to define the network. At each time step the architecture is updated by the controller which outputs parameters defining a convolutional layer(number of filters, filter height and width, stride height and width), with its input being the parameters at the previous time step. Because of the recurrent architecture, it is possible to generate convolutional architectures with a variable number of layers. In addition, the controller can generate skip connections in the network through an attention mechanism over the previous $N-1$ layers, where the attention is shown as:

$$P(Layer\ j\ is\ an\ input\ to\ layer\ i) = logistic(v^T \tanh(W_{prev} * h_j + W_{curr} * h_i)) \tag{2.1}$$

Here h represents the hidden states, while W and v are trainable parameters. These additional inputs from the skip connections are concatenated in the feature dimension, and if they have different spatial sizes the input is 0 padded.

The goal of this work is to optimize the parameters, θ_c of the controller to maximize the expected validation accuracy of the generated CNN. In reinforcement learning terms, the controller produces a set of actions, $a_{1:T}$, which are the layers of the model, and is given a reward, R the validation accuracy of the created model. Then we can write the expected reward of a given set of parameters as:

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R] \tag{2.2}$$

This is optimized using a policy gradient method, REINFORCE[172].

$$\nabla J(\theta_c) = \sum_{t=1}^{T} E_{P(a_{1:T};\theta_c)}[\nabla \theta_c log P(a_t | a_{(t-1):1}; \theta_c) R] \tag{2.3}$$

In practice, this is optimized with an empirical approximation of this expectation:

$$\nabla J(\theta_c) \approx \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} E_{P(a_{1:T};\theta_c)}[\nabla \theta_c log P(a_t | a_{(t-1):1}; \theta_c)(R_k - b)] \tag{2.4}$$

In addition, they subtract a baseline, b from the reward, taken as an exponential moving average of previous rewards. This is used to reduce the variance in gradient estimates and can be shown to be equivalent as long as the baseline doesn't depend on the current action.

This work achieves comparable accuracy to the best human designed networks on CIFAR10, and also was able to achieve SOTA for some NLP tasks. The limitation of this method was the huge amount of compute required, because it was necessary to train a total of 12800 architectures before the controller achieves their best results.

This method was improved on with the introduction of **NASNet**[199], where the search space was redesigned to better allow the architectures to transfer to other datasets. Instead of directly outputting the entire architecture in terms of convolution layers, they search over cells, which are then stacked together in a pre-defined way to form the full network. The higher level architecture is hand defined, with search only happening over the cell structure.

They define two types of cells, normal and reduction, where reduction cells have a spatial downsampling of a factor of 2. Each cell takes as input the outputs from the previous two cells in the network, and the cell is composed of 5 blocks that the RNN generates sequentially. For each block, the RNN first selects two inputs, either from the previous two layers, or the outputs of previous blocks within this cell. Next the RNN outputs two operations to apply to these inputs from a set of common operations in convolution networks, including pooling, convolutions with various filter sizes and stride as well as depthwise separable convolution. Next the outputs of these operations are combined using either element wise addition or concatenation, again chosen by the RNN. The outputs of these 5 blocks are then concatenated to get the output of the cell. Similarly to the original NAS method, an RNN controller is used to generate the cell, and it is trained using another policy gradient algorithm, Proximal Policy Optimization(PPO)[140].

This work achieved 2.4% top-1 accuracy on CIFAR10, which was state of the art at the time, and 82.7% top-1 accuracy ImageNet even though the model was not directly trained on ImageNet. They found that the best performing blocks mostly used depthwise separable convolutions, as well as element wise addition instead of concatenation. The downside was the extreme computational requirements. While the experiments are significantly faster than the original NAS paper because of the smaller search space, they still take 1800 GPU days. In addition, this is inefficient, because the results achieved with this process are only marginally better than random search. The key take away from this work is the importance of search space design, because with an appropriate search space even random search is effective.

2.1.3 Evolution Based Architecture Search

Evolutionary strategies is a black box optimization method inspired by the process of biological evolution. The optimization process proceeds in terms of generations, where at each generation a population of possible solutions is evaluated based on some fitness function, and based on the fitness, the population is mutated and recombined to generate a new population for the next generation.

Evolutionary methods for architecture search have been investigated since the 1990s, with early work showing the effectiveness of evolutionary algorithms for constructing both the weights and structure of recurrent networks[7]. They created a method called GNARL (GeNeralized Acquisition of Recurrent Links), where there are two types of mutations, corresponding to mutating a parameter by adding Gaussian noise, and mutating the structure of the network by adding or removing edges or nodes. These mutations are both controlled by temperature parameters, allowing the severity of mutations to be annealed over later generations.

Another method for simultaneously evolving weights and architectures is NeuroEvolution of Augmenting Topologies (NEAT)[151], which grows a simple network through three mutations: modifying a weight, adding a connection between existing unconnected nodes, or by adding a node in a given weight, splitting it into two parts. They directly encode the entire structure of the network along with history markers allowing for recombination, as well as using speciation, meaning that niches of individuals will compete with each other, not the entire population.

More recent work has also used evolution based approaches for architecture search for more modern convolutional networks. Tournament selection[128] has been used to evolve networks, which is based on mutations, not recombination of different individuals as is common in other evolutionary algorithms. At each step two models are randomly selected from the population, and their performance is compared, with the worse model being killed(removed from the population), and the better performing model being selected to be a parent, meaning that it will undergo some mutation to produce a child model. This child model is then added to the population, and the process is repeated. The population is initialized with poorly performing architectures consisting of no convolution layers to ensure that it truly is the evolution process discovering the architecture, not the result of seeding it with a good human designed architecture. The DNA is represented as a directed acyclic graph (DAG), where nodes represent nonlinearities, and the edges represent convolution layers. For these mutations, there are a wide variety of commonly implemented mutations like altering filter size, adding a convolution layer, changing stride or adding skip connections. These mutations act on entire layers, and evolution is used on the architecture

only; weights are trained normally given a fixed architecture using SGD.

Other work[127] extended the tournament selection based approach by adding regularization in the form of removing older architectures, creating the first non-human designed architecture to achieve SOTA on ImageNet, **AmoebaNet**. They use an architecture with fixed higher level structure and search over two types of cells. Mutations are random operations within the these cells, as in other work[199]. The regularization method adds the constraint that the oldest individual in the population is killed, in addition to the standard tournament selection process. They also noted this tends to search faster than RL[199], and getting at least as good or better performance.

2.1.4 Efficiency Improvements in Neural Architecture Search

The extreme computational requirements of both reinforcement learning and evolution based approaches meant that NAS was not practically useful for many problems. Because of this, many approaches to speed up the process of NAS through search space redesign have been proposed

Efficient Neural Architecture Search via Parameter Sharing (**ENAS**) aimed to speed up architecture search by forcing models to share weights[118]. Because of weight sharing between models, not every architecture needs to be trained fully from scratch, so compute can be reduced by over 1000x. The key part of this work was to observe that every architecture searchable by NAS is a subgraph in some larger graph, and the controller should search for the optimal subgraph. This means that the same parameters can be used for each model instead of being retrained each time. They investigate two types of search spaces here, one designing the full architecture as is the original NAS paper[198], where the controller RNN deciding the type of convolution and skip connections as well as a cell-based search[199], which gave better performance.

Progressively increasing the search space for cell-based NAS was also investigated to improve efficiency[93]. They first searched over limited cells, and by keeping only the best performing architectures they extended the space to include more complex structures. In addition, they train a surrogate model to estimate performance of a model based on the architecture to avoid the expensive training process.

Other work[96] formulated NAS as a differentiable optimization problem so efficient methods like SGD can be used for the optimization with no controller required. To make this differentiable they use a learned softmax over all operations and alternatively train the network parameters with SGD on training loss and the softmax parameters using SGD on validation loss. To create the final architecture, they retain the top-k softmax activations,

using $k = 2$, and retrain this network. This took only 4 gpu days of compute using a small cell based search space.

Hierarchical representations of architectures[95] have been used to improve the search space for tournament selection based NAS using a cell based search space. They use primitives including various forms of pooling and convolution as the first level of the hierarchy, and as search progresses these are combined to form higher level structures, which are used as elements in higher levels of the hierarchy. This allows for more efficient search because mutations can be applied at any level of the hierarchy, rather than only on lower level operations. In addition, hierarchical search spaces have been used for search for semantic segmentation[92], allowing the search over both the cell and network level. At the network level, the search space is the amount of downsampling, and both the cells and network are searched using a continuous relaxation of the problem[96].

Automatic search methods have also been applied across a variety of other problems. A combination of hand designed scaling methods with base networks learned though architecture search[159] has been used to design efficient networks. Reinforcement learning based search was used to discover activation functions, resulting in the Swish activation function, $f(x) = x * sigmoid(\beta x)$[125]. Data Augmentation strategies have also been learned using a similar reinforcement learning based strategy[33]. In addition, other work has included objective functions in the form of inference latency to optimize the network for mobile applications[158].

2.1.5 Limitations of Current NAS Methods

Though a comprehensive evaluation of the entire search space of discoverable architectures using a cell based approach, it was shown that reinforcement learning based methods converge more slowly than simple regularized evolution approaches[182]. In addition, random search performs quite well, with state of the art semantic segmentation results found using only random search[23]. Other methods such as ENAS under performed random search[3], and other reinforcement learning or evolution based approaches provided only a small improvement over random search[127]. Many of the best human designed architectures lie near the Pareto frontier of accuracy vs. network size[182], meaning that the architecture search methods provide minimal improvement over the best human-designed models. Architectures also exhibit locality, meaning that similar architectures attain similar accuracy so there are many near optimally performing architectures within the search space.

Current NAS methods have limited ability to create genuinely new architectures. Developments like ResNets[59], attention layers[163], dilated convolution[185] and depth wise

separable convolution[148] could not have been discovered using current architecture search methods. Important hand-designed innovations need to be directly added to the search space for these NAS methods to use them. For this reason, existing NAS methods can be best seen as rearranging existing blocks in a network to some optimal structure, not generally searching in the space of architectures.

The key limitation of NAS is the difficulty of hand designing a search space small enough for the inefficient search methods to work while also being large enough to include novel components. In current NAS methods the search spaces need to be extremely limited, mostly limited to rearranging existing hand-designed components. Because NAS search methods remain relatively inefficient it is not possible to increase the search space enough to be able discover novel lower-level components. To discover architectures with significant performance improvements, NAS methods will need to use an expanded search space along with more efficient exploration methods.

2.2 Network Compression

In this section we will focus on pruning and quantization methods for reducing network size, instead of using hand design or NAS for efficiency. These approaches are closely related, but instead of searching in some larger space for an optimal network like in NAS, pruning and quantization methods start with an existing network and use methods to reduce the size of this while preserving a certain level of accuracy. This problem turns out to have much less computationally expensive approaches compared to the general NAS problem, which we will discuss in detail in this section.

2.2.1 Sparsity

Sparsity refers to a model having a subset of parameters being exactly 0[48] and can be divided into two categories, structured and unstructured. Unstructured sparsity simply has the goal of reducing the number of parameters in the network without any concern for how this is achieved. While unstructured pruning decreases the size of the network, this does not necessarily correspond to a network that is more computationally efficient given current hardware. Structured pruning overcomes this issue by forcing a particular form of sparsity into the network that is practically useful, for example by removing entire kernels in a convolutional network

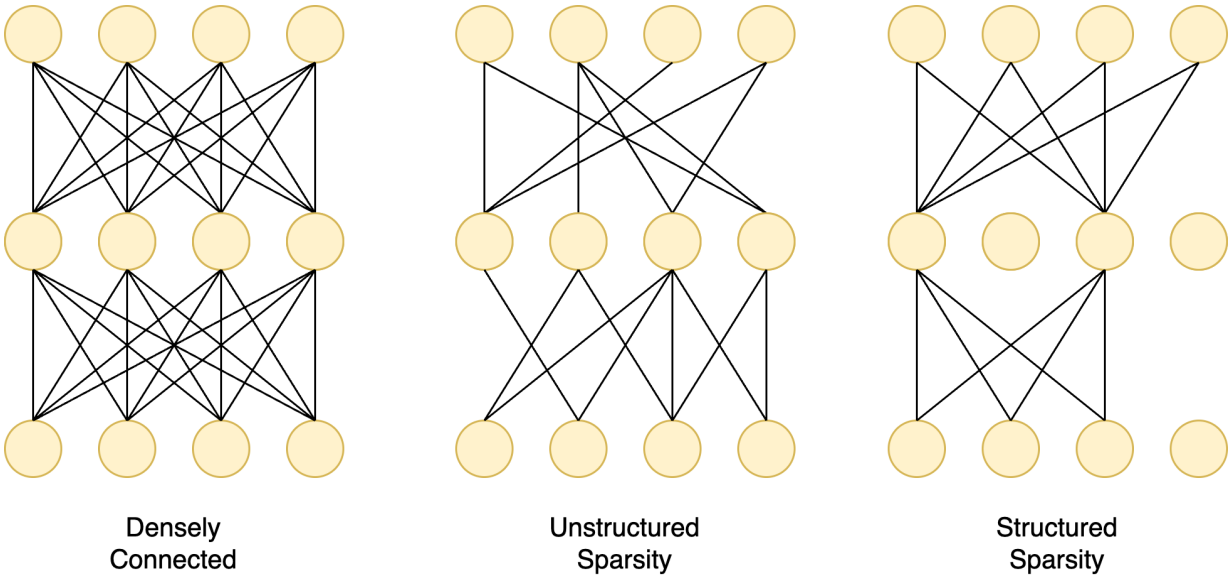


Figure 2.1: Comparing structured and unstructured sparsity. On the left we see the densely connected neural network, with each node connected to every other node. In the middle we see unstructured sparsity, where the weights are randomly removed. On the right we see structured sparsity, where the weights are removed in a structured way, in this case removing an entire neuron by removing all weights connected to it.

Unstructured Pruning

Magnitude based pruning[58] is a method for creating sparsity in a network where the weights with the lowest absolute value are removed, and the network is retrained. This method demonstrated that neural networks can achieve strong performance even with a high level of sparsity. They also showed the effectiveness of l_1 and l_2 regularization during pruning, finding that l_1 regularization is most effective when pruning a network without retraining, but l_2 regularization is more effective when retraining is used. The best performing method was an iterative process of training, pruning and retraining using l_2 regularization. Earlier work investigated a second-order approximation of the model's loss for pruning[85], but recent work has found comparable compression is possible without this[109]. In addition, this iterative training and retraining has been improved by allowing gradient updates to the pruned weights, letting incorrectly pruned weights to be added back to the model[54][195].

Bayesian deep learning is another approach to creating sparse neural networks, where a distribution, rather than a single value, is learned for each parameter. The reparameterization trick[78] allows these networks to be efficiently trained using standard methods. After learning a distribution over the weights in the network, there is a principled way to introduce sparsity by noticing that weights corresponding to distributions with high variance are likely to be uninformative and only add noise to the model prediction, so can be removed[108][102].

The problem of unstructured sparsity can also be framed as l_0 regularization, where the network is penalized based on the total number of non-zero weights. In contrast to l_1 or l_2 regularization, this is a non-differentiable objective so it is difficult to optimize directly. Recent work[103] has reformulated this problem using stochastic gates instead of exact l_0 norm, so the expected l_0 norm can be used to allow differentiability while still forcing the weights to be exactly 0.

While variational dropout and l_0 regularization can achieve state of the art sparsity on smaller models, a recent review paper[48] showed this does not extend to larger models such as ResNet50 on ImageNet, where modified magnitude based pruning can outperform these methods. It was found that variational dropout can be slightly better than standard magnitude based pruning, but l_0 is significantly worse. This was because variational dropout is better able to distribute parameters across layers, retaining more in the most important initial and final layers. With small hard-coded changes to preserve these parameters, magnitude based pruning is comparable to variational dropout. Interestingly this was not the case on an NLP task, where l_0 regularization performed well.

Structured Pruning

Many unstructured sparsity methods can be transformed into structured sparsity methods through small modifications. Magnitude based pruning[58] can be extended to structured pruning by removing groups of weights based on any relevant summary statistic like l_p norm. The approximate l_0 regularization approach can be extended to structured sparsity by sharing the same gate across multiple parameters[103], as well as the Bayesian approach[102] by sharing the variance parameters between groups of weights.

In addition, group level sparsity has been introduced into simple fully connected neural networks[137] through group l_1 regularization[187], also known as group lasso. Regularization using a standard l_p norm shrinks the weight values independently, so the optimal weight values can be shown as:

$$w^\star = \arg\min_w \left\{ \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i, w)) + \lambda \|w\|_p \right\} \tag{2.5}$$

Here $w^\star$ indicates the optimal weight values, L indicates the loss function for the network's performance, and

$$||w||_p = \left(\sum_{i=1}^{K} |w_i|^p \right)^{\frac{1}{p}} \tag{2.6}$$

is the l_p norm of the network's weights. This can be extended to encourage group sparsity as:

$$w^\star = \arg\min_x \left\{ \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i)) + \lambda \frac{1}{J} \sum_{j=1}^{J} ||w_j||_2 \right\} \tag{2.7}$$

Now the loss is replaced with this, which is the sum of the l_2 norms of the J groups of parameters. Because this is positive, it is equivalent to l_1 norm of the losses of the groups. Because within each group the l_2 norm is used, this does not encourage sparsity within groups, only between them.

This approach has been extended to encourage sparsity in convolutional networks[169], where they used 2D slices of each convolution filter as groups, or by directly pruning filters with small l_1 norm[89]. Other group sparsity methods have taken the entire filter as the group and pruned based on the expected loss change after removing it, approximated with

the first order taylor expansion. They apply this method to iteratively remove the least important filters and retrain the network, finding that this outperforms simple methods such as mean activation or minimum weight. For a given feature map z with M elements, and a sample loss, C on a minibatch of examples, this is shown as:

$$approximate\ cost = \left| \frac{1}{M} \sum_{i=1}^{M} \frac{\delta C}{\delta z_i} z_i \right| \tag{2.8}$$

Other work created networks with individual and group level sparsity by formulating the problem as the synthesis of a new network given some existing network[143]. They model the creation of a network probabilistically:

$$P(S_g|W_{g-1}) = \prod_{c \in C} \left[P(\bar{s}_{g,c}|w_{g-1}) \cdot \prod_{i \in c} P(s_{g,i}|W_{g-1,i}) \right] \tag{2.9}$$

Here w refers to the weights in the network, and s to the connections. Here the right term shows the probability of a given synapse/connection existing in the network given the weights in the previous generation, and the left term represents the probability of a given cluster of synapses given the network in the previous generation. Then environmental constraints can be added to increase sparsity in the weights or clusters over generations.

2.2.2 Quantization

In addition to reducing the model size by enforcing sparsity, model compression is also possible by quantizing weights, which means storing weights in a more compact format than standard 32 bit float[55]. Quantization can be applied either during or after training, with the simpler approach being after training, but this can come at the cost of reduced performance. After training, quantization can be done through rounding or a more complex method like k-means. In this section we will focus on learned quantization methods during training because they have been shown to preserve performance best while reducing model size.

A more extreme approach to quantization has shown weights can be quantized to binary values during training by taking the sign of the weight[32]. They use this quantization during the forward and backward pass, but the full precision is used during the update step.

$$w_{quantized} = \begin{cases} +1, & \text{if } w \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{2.10}$$

Alternatively, they can be quantized probabilistically using the sigmoid of the weight:

$$w_{quantized} = \begin{cases} +1, & \textit{with probability } p = \sigma(w) \\ -1, & \textit{with probability } 1 - p \end{cases} \tag{2.11}$$

This was extended[126] to use binary filters in addition to scaling factors for each kernel, as well as using the proximal Newton algorithm[65] to minimize binarization loss. In addition, this was extended to ternarization, where the weight is quantized to a set of three values -1, 0, +1[91] and loss aware ternarization[64].

Other work has taken a different approach to quantization, using prior distribution over weights as a mixture of k Gaussians fit using the Empirical Bayes method[161]. In this case the main objective was model compression, because this allows the weights to be compressed by encoding only the k cluster means and an assignment parameter, instead of all parameters in the network. An additional component with mean fixed at 0 can be used to learn sparsity, where its weight in the mixture will correspond to the amount of sparsity in the network. In the final step each weight is assigned to the most likely component, and the weight is quantized to the mean of this component. Other work[6] has also used the end to end training approach, where parameters are quantized by being annealed gradually from soft to hard assignment to a quantization level during training.

2.2.3 Sparsity and Weight Sharing in Convolutions

Many key advancements in convolutional neural networks can be framed in terms of introducing sparsity and or weight sharing into an existing convolution. Here we will briefly review a few of these instances.

Dilated convolutions [185] can be seen as a form of sparsity, with a dilation of a factor of α being equivalent to a convolution with a kernel size α times as large, but with many weights zeroed out. A more extreme form of sparsity is depthwise separable convolutions[148, 25]. While normal convolutions take the entire feature map as input, depth wise separable only take single 2d slice of the feature map as input, corresponding to sparsity where weights in a convolution filter are zero except for a single channel.

Group equivariant convolution layers[30] can be seen as a from of weight sharing, where the same weight-shared convolution is applied at four 90 degree rotations. Also, lightweight convolutions[175] can be seen as a case of both sparsity and weight sharing, where additional weight sharing is added to depthwise separable convolutions, forcing multiple filters in a layer to share weights.

2.3 Generative Models

We review Variational Autoencoders [75] (VAE) to motivate our research on reducing the size of latent representations in generative models. We also review some background on disentangled representations which is closely related to our research question.

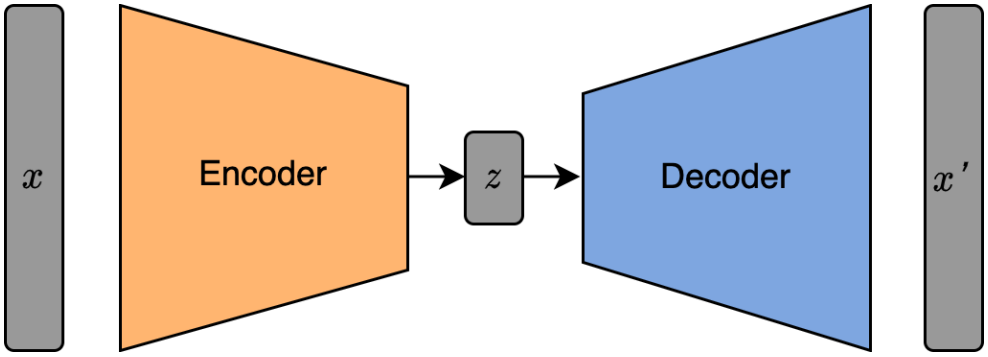
2.3.1 Variational Autoencoders

Variational Autoencoders are generative models where it is assumed that the data, $X = \{x\}_{i=1}^{n}$ are generated from latent variables, z, with a prior distribution, $p_\theta(z)$, as a centered isotropic multivariate Gaussian. The likelihood, $p_\theta(x|z)$ is assumed to be a multivariate Gaussian, and the posterior, $q_\phi(z|x)$ is assumed to be Gaussian with diagonal covariance. The parameters of the likelihood and posterior are represented with neural networks, as shown in Figure2.2(b).

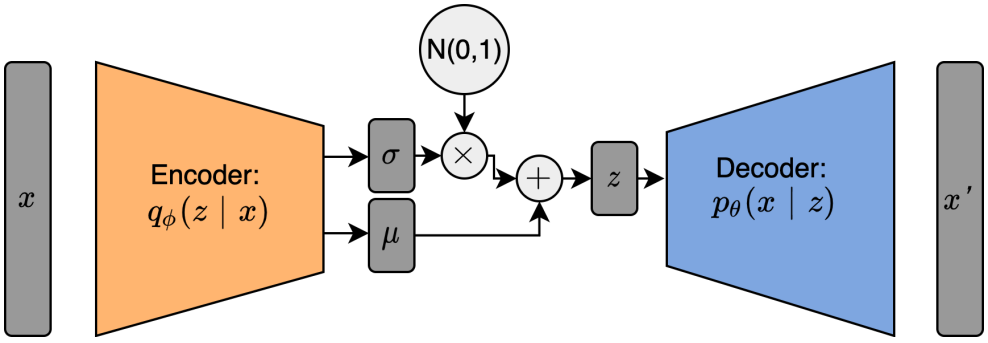
The VAE is trained to maximize likelihood of the data by maximizing the evidence lower bound (ELBO) shown in equation 2.12. This entire network is differentiable, so can be trained using stochastic gradient descent with this loss.

$$-L_{VAE} = E_{z\sim q_\phi}[logp_\rho(x|z)] - KL(q_\phi(z|x)||p_\rho(z))] \tag{2.12}$$

An alternative motivation for VAEs can come from the autoencoder, which is a model composed of two neural networks, an encoder and decoder that encode data x to a compressed representation, z before decoding it as x', as seen in Figure2.2(a). This model is also trained using gradient descent to minimize the reconstruction error, the difference between x and x', commonly measured using mean squared error. We can see the VAE as forcing a distribution on the latent space of the autoencoder, making it possible to sample from the latent space and pass this through the decoder to generate samples from the distribution.



(a) Autoencoder



(b) Variational Autoencoder

Figure 2.2: The Autoencoder is composed of two neural networks, an encoder to generate a compressed representation of the data, z, and a decoder reconstruct the data from the latent representation. Variational Autoencoders also generate a compressed representation of the data but add additional regularization to the latent space to force it to be normally distributed, and are motivated by Bayesian inference.

2.3.2 Disentangled Representations

The goal of learning disentangled representations is to create a human interpretable representation where variables correspond to an understandable category, such as color, shape or size. Even though in general it is impossible to learn disentangled representations without making some assumptions about the data [99], there has been extensive work on this problem using relatively weak assumptions. One approach learns disentangled representations where semantically relevant variables are explicit in the latent space [130]. These are not limited to affine transforms, and include variations such as lighting, color, or physical attributes like shape. Another approach is based on semi-supervised learning, where images are generated based on both a latent variable and some relevant factor of variation, which are assumed to be independent [77]. For face generation, disentangling shape and appearance was tackled through the synthesis of appearance on a template followed by a deformation [147]. Other work divides the latent space into explicit and implicit factors of variation, and a training process where one factor is varied while fixing the others is used to enforce the disentangled latent space [82]. These methods all require supervised inputs, where they are labeled based on some factor of variation. It is also possible to to use invertible transformations in VAEs [40] to disentangle appearance and perspective using transformation parameters inferred from the object.

In addition to encouraging disentanglement, other work has explicitly encouraged neural networks to generalize better to a specific factor of variation, which is also relevant to creating better representations. In particular, Spatial Transformer Networks (STN) [72] aim to transform images to some canonical orientation by applying an affine transform to the input image using a differentiable three stage process:

1. **Localization Network**: This is a neural network taking an image as input, and outputting the affine transformation parameters, $\alpha \in \mathbb{R}^6$ to be applied to this image.
2. **Sampling Grid**: Given a transform, the grid of coordinates in the input image associated with each pixel in the output image
3. **Image sampling**: Given the grid, use bilinear sampling to apply it to the input image.

This is differentiable, so it can be trained using stochastic gradient descent along with the rest of the network. The STN generates a single prediction of the best transform parameters and has no way to update them, so there is no guarantee of disentanglement.

It is also possible to create networks that are equivariant to one specific factor of variation to improve representations, such as CNNs that are equivariant to rotation and reflection [30]. While this is an effective method, it is limited to 90° rotations, and adding more factors of variation increases the complexity dramatically. Other work has focused on rotations specifically using polar coordinates [157], but this is also limited to rotations only. Another approach to ensure deep convolutional neural networks are invariant to arbitrary affine transforms is to add a layer that applies a random affine transform to the feature map, forcing the model to output the same classification regardless of the orientation of the image [146], but additional methods are needed to get a disentangled representation based on this approach.

2.4 Text Classification

In this section we will review the dominant approach to text classification, the transformer architecture, as well as various methods to increase efficiency through shared parameters, efficient self-attention layers, and modified architectures.

2.4.1 Transformers

The introduction of the Transformer[163] led to a paradigm shift in natural language processing (NLP), where a variety of problems including language generation, question answering, and text classification could all be solved with a single architecture. In addition, its flexible architecture with minimal inductive bias allowed it to be used in a variety of other domains including vision and time series. This is all made possible by the transformer's use of unsupervised pretraining, allowing it to leverage large amounts of unlabelled data to increase performance on tasks with limited amounts of labelled data.

Transformer Architecture

The architecture of most transformer style models[163, 98, 42] are composed of a stack of identical blocks (excluding input and output layers), where each block consists of fully connected layers, multihead attention[9], layer norm[8] and residual connections[60], shown in 2.3.

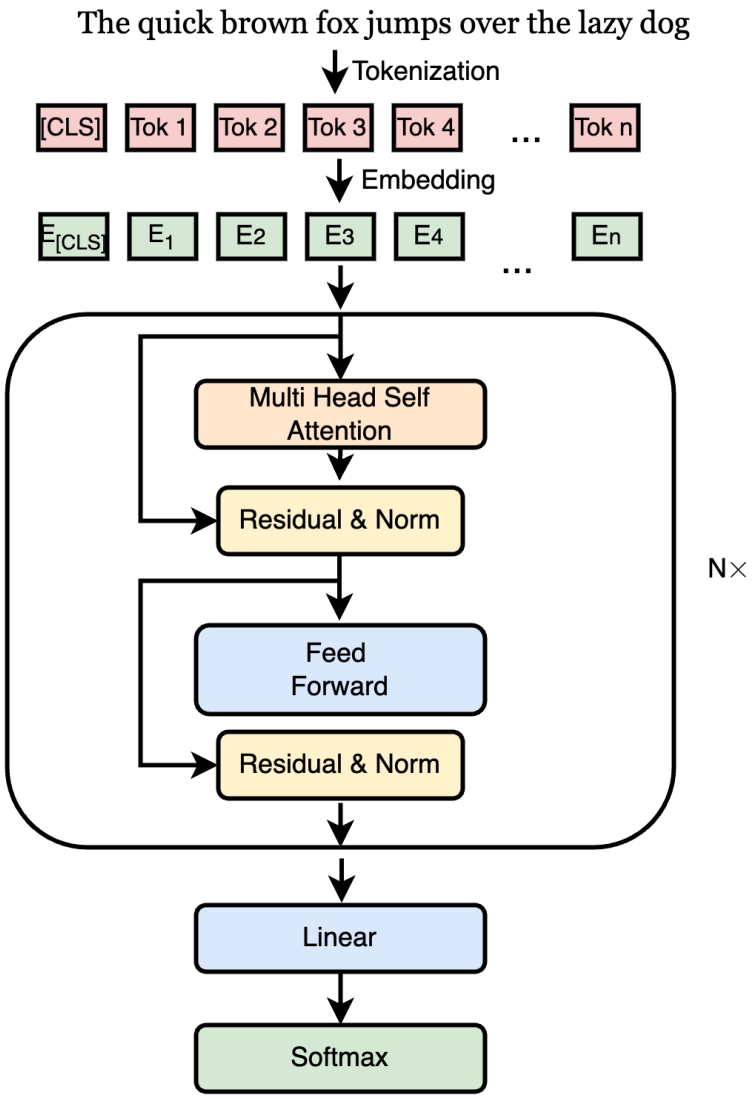


Figure 2.3: The high level architecture of the transformer model, specifically a BERT style encoder-only model. The input is a sequence of tokens which are first passed through an embedding layer, then through a stack of identical transformer blocks. The output of the transformer blocks is passed through a linear layer, and then through a softmax layer to generate a probability distribution over the vocabulary.

The transformer block can be shown as:

$$X_A = \text{LayerNorm}(\text{MultiheadSelfAttention}(X)) + X$$
$$X_B = \text{LayerNorm}(\text{PositionFFN}(X_A)) + X_A$$

Multihead Self Attention

We first describe single head self attention, a special case of multihead self attention. We assume an input x of size $\mathbb{R}^L \times \mathbb{R}^{d_k}$, which represents the entire sequence along with an embedding dimension. Self-attention transforms the input sequence with three distinct linear projections: W^k, W^q, and W^v, which correspond respectively to the key, query, and value in the attention mechanism. Self-attention computes a weighted sum of the values ($\mathbf{x}\mathbf{W}^v$), where the weighting is determined by the similarity of the query ($\mathbf{x}\mathbf{W}^q$) with the keys ($\mathbf{x}\mathbf{W}^k$). Similarity is assessed by taking the dot product of the query with all keys and dividing by $\sqrt{d_k}$ to scale down the magnitude. Finally a softmax is used to normalize attention scores to between 0 and 1. These attention scores specify the extent to which each value in the sequence should be attended to when processing the current part of the sequence. Mathematically it is defined as:

$$\text{SelfAttention}(\mathbf{x}) = softmax\Big(\frac{(\mathbf{x}\mathbf{W}^q)(\mathbf{x}\mathbf{W}^k)^T}{\sqrt{d_k}}\Big)(\mathbf{x}\mathbf{W}^v) \tag{2.13}$$

This is extended to multihead attention by dividing the input, x, into N_h vectors of size $\mathbb{R}^L \times \mathbb{R}^{\frac{d_k}{N_h}}$, and performing the self attention independently for each of the N_h vectors, and finally concatenating them together again.

Position Wise Feed Forward

The feedforward part of the transformer block operates on the output of the self attention layer. Given two linear functions F_1 and F_2 of the form $Wx + b$, and a non-linear activation act, the feed-forward layer acts independently on each position in the sequence and is defined as:

$$F_2(act(F_1(X_A)))$$

Tokenization

Before text is inputted to the network it must be converted from text to numbers, called tokenization. A naive approach is to map each word or character to an integer which is

used as a lookup for a d_{emb} dimensional embedding vector. These approaches are limited because character level tokenization results in extremely long sequences and word level tokenization results in large embedding sizes and limited flexibility in dealing with either similar or rare words. An improved approach is to instead map commonly occurring parts of words to individual tokens.

For the popular BERT[42] architecture wordpiece tokenization[176] with 30k tokens is used. It improves on word level tokenization by dividing the input into a limited set of subwords. Given some desired number of tokens and a training corpus, the tokenization mapping is created by choosing a mapping to minimize the length of the training corpus after tokenization.

Another popular approach for tokenization is Byte Pair Encoding (BPE)[142], used for other transformer models such as Roberta[98], where it is used with a vocabulary size of 50k tokens. It is also a compromise between word and character level tokenization and is based on subwords. It uses the byte pair encoding algorithm which works by starting off with all individual characters as the vocabulary and iteratively merging the most frequent instances in a sequence until the desired vocabulary size is reached.

Position Embeddings

As described above, the pure transformer architecture is translation invariant in the tokens because position information is not introduced in the tokens or anywhere else in the network. This would limit performance to that of a 'bag of words' style model, which in this case would be 'bag of tokens'. To overcome this, position information is added to tokens through position embeddings.

One approach for position embeddings is to directly add position information to the token embeddings at the start of the network. These can be in the form of sinusoidal embeddings[163], where alternating sin and cos functions are used to generate an embedding based on the position, pos and the dimension, i:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

Another effective method for position embeddings is to make them fully learnable[163], which can give similar performance in practice to the sinusoidal embeddings. In this case there is a vector of size $\mathbb{R}^L \times \mathbb{R}^{d_k}$ that is directly added to the input embeddings, and gradient updates are also performed on this vector, creating a learned position embedding.

Alternatively to absolute position embeddings discussed above, relative position embeddings [144] can explicitly represent positions between elements. This is implemented by

adding position information into the key matrices at various layers of the network instead of through direct addition only at the first layer of the network.

In self attention, the unnormalized (before softmax) attention weights are computed by taking the dot product of the query and key. The query and key are computed by taking the input vector, x_i times the query projection W^Q, and the key by multiplying the key vector x_j with the key projection W^K:

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d}}$$

For relative position embeddings an additional vector a_{ij}^K is added to each of the projected keys. These are only added to a fixed number of offsets, and they find it is only useful to add to their neighbors at some distance, k, from the query value:

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K + a_{ij}^K)^T}{\sqrt{d}}$$

The standard attention weight as shown in equation 2.13 can be rewritten in the following form by making the absolute position embeddings explicit, where p_i corresponds to the absolute position embedding for input x_i[170]:

$$e_{ij} = \frac{x_i W^Q W^{KT} x_j^T + x_i W^Q W^{KT} p_j^T + p_i W_q W^{KT} x_j^T + p_i W^Q W^{kT} p_j^T}{\sqrt{d}}$$

Other work such as Transformer XL[37] and XL Net[181] are designed to overcome the fixed context length in standard transformers. They overcome this with a recurrent style generalization of the transformer where hidden states between segments are reused, but to do this a modified version of relative position embeddings is necessary to ensure the same position embedding is not reused for multiple positions.

They replace the fixed position embedding p_i with a relative position embedding r_{i-j}. $p_i W^Q$ is separated into two learnable parameters $u \in \mathbb{R}^d$ for content and $v \in \mathbb{R}^d$ for location based embeddings. Similarly the key weight matrix W^K is split into location and content components, W_R^K and W_E^K:

$$e_{ij}^{\text{rel}} = \frac{x_i W^q {W_E^k}^T x_j^T + x_i W^q {W_R^k}^T r_{i-j}^T + u {W_E^k}^T x_j^T + v {W_R^k}^T r_{i-j}^T}{\sqrt{d}}$$

The most promising form of position embeddings are called rotary embeddings[152], which improve on the performance of existing position embeddings without the extra parameters required for relative position embeddings. The goal is to create a position embedding where the inner product between a key and query value depends only on their relative positions and their values, not their absolute positions. This is done by representing the vectors as complex numbers, where the absolute value represents their content, and their orientation represents position. This ensures when taking the inner product between vectors the position embedding of the output will only depend on their relative locations. This can be applied to for standard self attention layers or most other efficient versions of self attention.

Transformer Pretraining

The main advantage of the transformer is its ability to leverage large unlabelled datasets through pretraining. The most common pretraining approaches are casual language modelling (CLM), where the model is trained to predict the next words in a sentence, and masked language modelling (MLM) where the model must fill in missing words in a sentence[42, 120].

The masked language modelling objective for pretraining was introduced in BERT[42], which showed state of the art performance on a variety of downstream tasks. In MLM, 15% of tokens in a sentence are chosen for masking. Of this 15%, 80% are directly masked with a special mask token [MASK]. 10% of the time it is replaced with a random token, and 10% with the original token. It is evaluated using cross entropy loss to predict the original value of these masked tokens. In addition, there is a next sentence prediction objective, where two sentences are sampled from either the same document or different documents. These are concatenated as input to the model, and in addition to the MLM objective the model must classify if the sentences are from the same document.

Roberta[98], is an improvement over the BERT model mostly through modified pretraining. It uses more extensive pre-training with the MLM objective and entirely drops the next sentence prediction objective, as they found it to be useless.

Other research has focused on more complex forms of pretraining objectives to improve performance. Electra[28] is inspired by generative adversarial networks (GAN)[51], where a discriminator and generator are jointly trained with opposing objectives, where the discriminator's goal is to classify if a data point is from the generator or the true distribution, and the generator is trained to fool this discriminator. This results in a generator that learns to generate realistic samples from the dataset.

Electra is based on a similar principle, except it uses a fixed generator that is not optimized jointly with the discriminator, and the focus here is on the discriminator. In contrast to standard MLM training, where the true value of the [MASK] token is predicted, during Electra pretraining the model is tasked with predicting if the token is from the original data or from a smaller generator network. This means that the loss is computed over the entire sequence length, instead of only the masked tokens like in standard MLM. It was shown that this increases performance on downstream tasks while having faster convergence during pretraining.

Overall, the transformer's pretraining step dramatically improves performance on supervised tasks with smaller datasets [163, 42, 120]. But these flexible architectures come at a cost, and this is increased compute requirements and slow inference times, limiting their use in low power applications such as mobile phones or edge devices.

2.4.2 Efficient Transformers

The high computational complexity of the transformer model[163] causes difficulty with deployment in real world applications and has motivated a large research interest in improving efficiency, especially in the expensive self attention layer. This has been achieved through a variety of methods, including modified architectures, sparse attention, or less computationally complex approximations to attention. In this section we will review some of the most prominent and effective approaches to efficiency in transformers.

Efficient Self Attention Layers

One way to improve the computational efficiency of transformers is to focus on the attention layer directly, because its quadratic complexity in sequence length is expensive, especially for longer sequences. This has motivated a variety of approaches to efficient self attention layers, including sparsity and lighter approximations to attention.

Transformers normally use multi-head attention, but recent work[177] has shown that many attention heads are heavily focused on local context, meaning that attention over the entire sequence length is unnecessarily computationally expensive. To take advantage of this, the light transformer architecture was developed which uses a two branch design where attention is used for global context and convolution for local context. This enables both local and global aggregation of information without the computational complexity of using full attention. Another approach is to replace self attention entirely with dynamic convolution[175], which is a form of convolution with larger kernel sizes, and kernels that

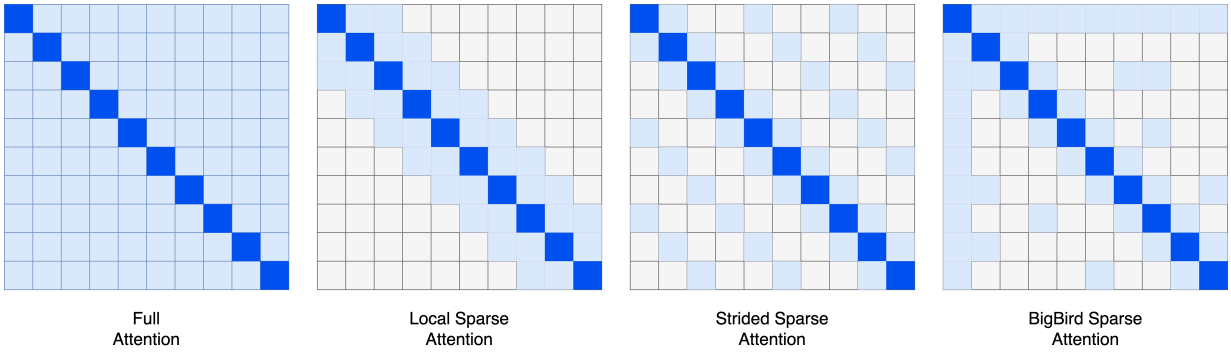


Figure 2.4: Comparing different versions of sparse self attention in the transformer. Each block represents a token, and the dark blue represent the token attention is computed on, the query. On the far left we see full attention, implemented in the original transformer, where every token attends to every other token. In the center we see two versions of sparse attention, local attention, which only attends to a fixed number of tokens on either side of the current token, and strided attention, which attends to the n^{th} token across the entire sequence. On the far right we see a combination of local, fixed and random attention used in the BigBird model.

are dynamically generated based on the input context. Additionally, weight sharing is added in the dynamic convolution, so weights are repeated across multiple features.

Another more direct approach for efficient self attention is sparse attention, which is applying attention to only a subset of the input tokens. There are different variants of sparse attention, including attending a fixed sparse pattern of tokens[24], shown in Figure 2.4 or through dynamic sparsity with an adaptive span[154] or content based with k-means clustering[133].

The combiner[129] is motivated by the idea of treating attention as an expectation over the sequence, and instead of computing attention directly in the combiner, attention is factorized to achieve sub quadratic complexity. Practically this is implemented as a two stage attention, where local chunks of the sequence are aggregated, and then these are further aggregated. This results in a complexity of $O(\mathbf{L}\sqrt{\mathbf{L}})$ or $O(\mathbf{L}log\mathbf{L}$ depending on the level of sparsity. Other work[94] showed that for vision and NLP problems self attention can be replaced with gated MLP layers, called gMLP. More recent work leveraged the advances of the combiner[129] and the gMLP[94] to enable a linear complexity model competitive with full attention. They introduced a model called FLASH (Fast Linear Attention with a Single Head), based on a linear approximation of the gated attention unit, a weaker single

head version of attention. It matches the perplexity of transformers over long (8k) and short (512) contexts for language modelling with nearly 5x speedup in training time.

Full attention can also be approximated through locally sensitive hashing where only the most relevant tokens are attended to, giving complexity of $O(\mathbf{L}log\mathbf{L})$[79]. Linear complexity approximations have also been achieved, either with more complex forms of direct sparsity, as in BigBird[188], or approximating the attention using orthogonal random projections[26]. These linear complexity transformer models have high fixed costs so they only outperform the standard transformer over longer length inputs, greater than 512[26]. This means that the speed of the standard transformer is still a strong baseline for many text classification problems.

Efficient Transformer Architectures

Another approach is to modify the full architecture instead of just the self attention layer itself. There are a wide variety of approaches to this, from various forms of architecture search, to hand designed architectures, to special training procedures such as distillation.

Evolution based neural architecture search was able to improve machine translation while reducing model size[149], and able to construct efficient task specific transformer architectures [22]. Architectures can also be adapted dynamically during inference using more computation for more difficult inputs. In practice, this means generating predictions at an intermediate point in the transformer and not using the later layers. Auxiliary classifiers can be added to each layer and early prediction can be decided by the entropy of the predictions[180, 97], or by only predicting after the prediction hasn't been updated for a set number of layers[194].

Manual design of efficient architectures has also been effective in improving transformers, such as in DistilBERT[136] a smaller version of BERT with 6 layers instead of 12. It retains most of BERT's accuracy while increasing inference speed and reducing the model size by pretraining with knowledge distillation, where its' objective is to match the output of the larger model. SqueezeBERT[70] found that with lower sequence lengths of 128 the feed forward part of the BERT model are most computationally expensive, accounting for nearly 90% of the latency on CPU. They replace this with faster grouped convolutions, increasing inference speed to 4.3x faster than BERT. MobileBERT[155] uses much smaller layers with greater depth, changing from 12 to 24 blocks. In addition, they change the form of the transformer block, changing layer norm to a linear layer, GeLU[61] to ReLU[4], and use an inverted bottleneck layer to reduce dimension. The model is trained to copy the outputs and layer activations of a larger teacher model which, resulting in MobileBERT

having 25% of the parameters and up to 5.5x faster than BERT with only slightly worse accuracy.

Other work has focused on reducing the sequence length in intermediate layers of the network. The Funnel Transformer[36] increases efficiency by reducing sequence length in later layers of the model with strided mean pooling on the query values. They find this allows a deeper model while still reducing the number of flops and preserving accuracy. To allow for standard pre-training methods, they include an optional decoder that up-samples the sequence to the full input length. Similarly there has been research in efficiency for language modelling by pooling tokens across multiple scales[153], and by constructing an explicit hierarchical structure by downsampling and upsampling the sequence length in the hourglass transformer[111], similarly to the UNet in computer vision[132]. Other approaches use the average attention score on each token as a measure of importance for pruning, as in PoWER-BERT[53] or aim to reduce the most redundant tokens based on core set selection[69].

Other Methods for Efficient Transformers

Efficient neural networks have been an active area of research in a wide variety of applications and many of the innovations described previously can be applied to transformers. Examples include pruning to remove redundant weights[85][89], quantizing weights[57], various forms of architecture search including genetically inspired methods, reinforcement learning and continuous relaxations to enable differentiation[143, 158, 96]. Efficient layer design has also been popular with convolutional neural networks, including depth wise separable[148][25] and dilated[185] convolutions.

2.4.3 Multi Task Training

It is standard for transformer based NLP models to be pre-trained on an unsupervised objective, such as MLM or CLM before being fine-tuned on a supervised task such as text classification. An intuitive approach is to include a third step, called multitask learning or pre-finetuning[5], where the pre-trained model is further trained on a set of tasks with the aim of increasing performance on the downstream task.

Additional pre-training has been shown to be useful in a variety of settings. A domain specific dataset can improve performance within this domain, such as in BioBERT[87], where additional pre-training on biomedical data was used on the BERT model. This improved performance when the model was finetuned on biomedical related tasks. In

ToDBert[174] additional pretraining with a modified objective on task oriented dialogue datasets improved performance on downstream tasks.

Exploring transfer between tasks has been an important research area in machine learning for a long time[19], but there has been renewed interest in transfer and multitask learning in the NLP domain since the introduction of the transformer. One approach that found benefit from using multitask training step was UNIFIEDQA[74]. This focused exclusively on question answering (QA) tasks, and involved converting all data into text-to-text format which enabled positive transfer between tasks. One approach is to convert a variety of problems into a text-to-text format, enabling a single model to be used for many tasks including QA, summarization and classification[121]. But this work also showed that naive approaches to multitask learning in NLP does not work and can hurt performance on a downstream task.

The Muppet[5] model trains on multiple tasks including classification, summarization, machine reading comprehension and commonsense reasoning were all trained using the same backbone with different heads for each type of task. They found that there are multiple keys to effective multitask training. The number of tasks used is especially important, and 10-25 tasks are needed to improve performance depending on the downstream tasks.

While the datasets have extremely different sizes, they found adjusting the sampling method performed worse than just sampling according to the natural distribution of the datasets. In addition, during training it is essential to use heterogeneous batches, where each gradient update is computed from a batch containing samples from multiple datasets. This can be implemented efficiently through gradient accumulation to avoid the difficulty of merging multiple tasks and objectives into a single batch.

Loss scaling is also necessary because of the different scales of the losses on each task and dataset. This is of the form:

$$\mathcal{L}_i^{scaled}(x_i, y_i; \theta) = \frac{\mathcal{L}_i(x_i, y_i; \theta)}{log\ n(i)} \tag{2.14}$$

where $\mathcal{L}_i(x_i, y_i; \theta)$ indicates the loss for data i on a model with parameters θ. This loss function $\mathcal{L}$ depends on the particular task being done, with cross entropy used for classification, label smoothed cross entropy for summarization, span prediction for MRC and sentence ranking loss for commonsense reasoning. $n : \mathbb{N} \rightarrow \mathbb{N}$ indicates the number of predictions each loss operates over, for example in classification this would be the number of classes.

2.5 Discussion

Across these disparate fields of image processing, generative models and natural language processing, a wide variety of approaches have been used to increase the efficiency of neural networks. Some approaches more directly frame the problem in terms of reducing redundancy, such as quantization reducing the number of bits required to represent a weight. Others, such as neural architecture search, are indirect, but still closely related to the goal of reducing representational redundancy. Motivated by all these previous developments across various applications of efficient neural networks, in the next chapters we investigate approaches to improve efficiency through the direct approach of reducing representational redundancy in the form of weight sharing, disentangled representations, and the removal of redundant sequence information. In Chapter 3 we begin with the problem of image classification, and find that additional weight sharing is a way to reduce representational redundancy within convolutional neural networks.

Chapter 3

Efficient Convolutional Neural Networks through Weight Sharing

As reviewed in Chapter 3, the convolutional neural network (CNN) is a powerful tool for image classification, however, their large size can make training and deployment difficult. A wide variety of approaches to solving this problem have been explored, but in this chapter we propose an alternative approach centered around removing redundancy in the convolution layer. We hypothesize that there is representational redundancy in the form of unnecessary diversity between channel weights in a convolution filter, and take advantage of this to develop a novel type of convolution using additional weight sharing. To demonstrate this we show that the weights in a convolution filter can be shared across channels while preserving or even improving classification performance, and that this method is compatible with some of the most efficient CNNs, including those designed through neural architecture search.

3.1 Representational Redundancy in CNNs

Previous research on improving efficiency in CNNs has focused on reducing the number of parameters in the network, either through structured or unstructured pruning, or reducing the size of each parameter with quantization. Architecture search is another approach but is less flexible, being limited to rearranging predefined layers to make the problem computationally tractable, as discussed in the background, Chapter 3. All these approaches retain the original convolution layers, without investigating if there is redundancy in the

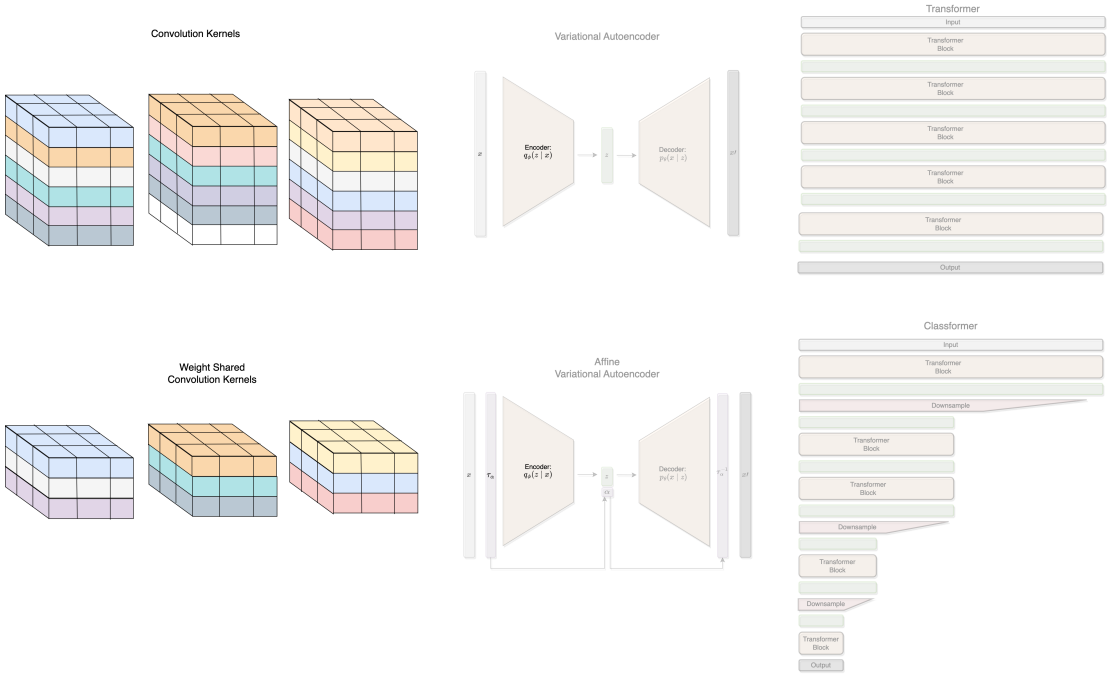


Figure 3.1: On the left we see a visualization of convolution kernels with (bottom) and without (top) additional weight sharing. Weight sharing reduces the number independent channels and parameters in each convolution kernel. To take the same size feature map as input, the weight shared convolution is expanded through duplication in the channel dimension.

convolution itself. In contrast, we take a novel approach and investigate redundancy in the convolution kernel's weights, and redesign the convolution through a constrained optimization process.

We hypothesize that there exists redundancy within the convolution kernel in terms of unnecessary flexibility between channels. By reducing this channel-wise flexibility, we hope to reduce the overall number of parameters that need to be stored.

Our approach is to constrain the convolution kernel by forcing weights to take the same values during the optimization process, known as weight sharing. In particular, we will be focused on constraining the channel wise flexibility of convolution kernels, so we will be removing the independence of parameters between channels within a convolution kernel.

3.2 Constrained Optimization and Weight Sharing in CNNs

We introduce additional weight sharing into convolution layers in the CNN. More specifically, we apply a constraint within a layer that requires certain sets of parameters to have identical values. This strategy particularly applies to weights in the channel dimension, making them interdependent within a single convolution kernel. This approach can be easily implemented using PyTorch[114], by creating copies of a convolution kernel across the channel dimension. This ensures that during backpropagation, PyTorch automatically enforces the weight sharing rule via gradient accumulation. This method of weight sharing is confined within each individual layer and does not extend across different layers. We describe the principle of weight sharing in more detail below.

Weight sharing can be formulated as a constrained optimization problem, where sets of parameters are required to have the same value. For example, if the weights, $\mathbf{w}$, were divided into sets: $\{\mathbf{w}_0^{basis}\}, \{\mathbf{w}_1^{basis}\}, .., \{\mathbf{w}_{N^{basis}}^{basis}\}$, the weights within these sets are enforced to take the same value. If we assume a neural network f, with inputs x_i and labels y_i, this constrained optimization problem can be expressed mathematically as:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{N}\sum_{i=1}^{N} L(f(x_i, \mathbf{w}), y_i) \\ \text{s.t.} \quad & \forall\{\mathbf{w}_j\} \in \{\{\mathbf{w}_0\}, \{\mathbf{w}_1\}, ..., \{\mathbf{w}_{N^{basis}}\}\}, \forall w_{j,k} \in \{\mathbf{w}_j\}, w_{j,0} = w_{j,1} = ... = w_{j,k} \end{aligned}$$

Weight sharing is enforced during training through the use of gradient accumulation. At each iteration, we first perform a forward pass, then compute the gradients of the loss

function with respect to the model's parameters. Following this, we apply the weight sharing constraints by averaging the gradients corresponding to the weights within the same set. This ensures that the weights in the constrained sets are updated by the same amount, effectively maintaining the shared values throughout training. Finally, we perform a gradient update step using an optimizer, in this case Adam[76], to adjust the model's parameters based on the constrained gradients.

3.3 Details of Weight Sharing Implementation

While the reduction of representational redundancy through additional weight sharing is the focus of this work, we do not have to start from the classic convolution kernel, and we can instead leverage previous innovations as a stronger starting point. Specifically, we can leverage the depthwise separable convolution, which has been shown to be a more efficient alternative to standard convolutions[25][66].

Given a feature map of size $(n \times n \times d)$, where n represents the height and width of the image and d denotes the number of feature maps, a standard convolution layer aiming to output a feature map of identical dimensions would require $(d \times d \times h \times w)$ independent parameters. The dimensions (h, w) correspond to the kernel width and height, and the number of independent parameters arises from the fact that d convolutions of size $(d \times h \times w)$ are required for each channel.

To alleviate the issue of excessive parameters, depthwise separable convolutions were developed, reducing the number of parameters in the convolution layer by performing convolution independently over each input channel. This approach leads to the depthwise separable convolution taking only a single slice of the feature map as input, as opposed to the entire feature map used in standard convolutions. Consequently, convolutions are only sized $(1 \times h \times w)$, resulting in a total parameter count of $(d \times 1 \times h \times w)$ per convolution layer. The significant parameter savings provided by depthwise separable convolutions have made them a popular choice for efficient neural networks, particularly in mobile applications[25][66]. This can be visualized in Figure 3.2, where the top panel illustrates a standard convolution, and the middle panel demonstrates the parameter reduction achieved through depthwise separable convolution.

Furthermore, the number of parameters in depthwise separable convolutions can be reduced even more by incorporating weight sharing along the channel dimension, as we have discussed above. This technique involves multiple filters in a single layer sharing identical weights. By incorporating additional weight sharing at a factor of s, the number

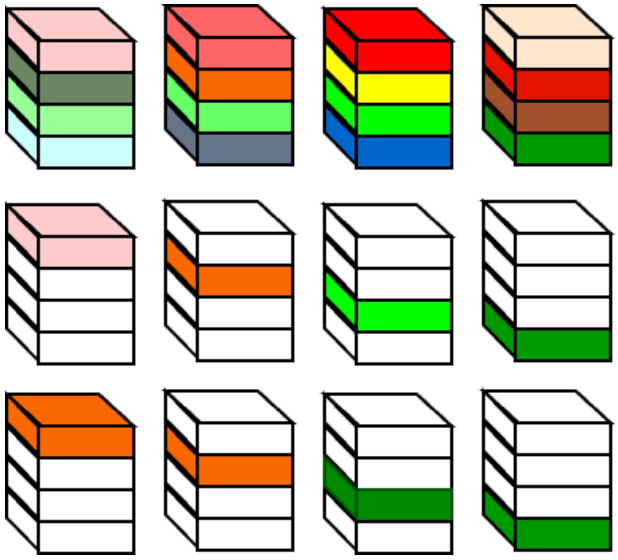


Figure 3.2: Comparing standard convolution (top), depthwise separable convolution (middle) and weight shared convolution (bottom) filters for 4 channel inputs. Each row shows 4 convolution filters which each will output a feature map. White portions correspond to areas with no parameters. Standard convolution uses independent parameters for each filter, while depthwise separable uses only a single feature map as input to each filter. Weight shared depthwise separable further simplifies the convolution by forcing multiple filters to share the same parameters, as indicated by the same colors in different filters.

of parameters can be decreased to $(d/s \times 1 \times h \times w)$, achieved by reusing the same filter s times. It should be noted that weight shared convolutions with $s = 1$ are equivalent to depthwise separable convolutions. A summary of the details is provided in table 3.3.

Convolution Type	**Parameters ($h \times w$ kernel; d feature maps)**
Regular Convolution	$d^2 \times h \times w$
Depthwise Separable Convolution	$d \times h \times w$
Weight Shared Convolution	$(d/s) \times h \times w$, where s is the amount of weight sharing

Why Weight Sharing within Convolutions? We may question why the weight sharing is applied within the convolution layer, as opposed to between layers. In theory we could implement weight sharing between layers in the CNN, such as was done in

ALBERT[84] for transformer models. But this is more difficult for CNNs, as in a transformer model the same sequence length and latent dimension is preserved throughout the entire network, making parameter sharing straightforward. In CNNs, the number of feature maps change throughout the network, generally increasing at later layers, meaning this straightforward implementation is not possible. In addition, it has been shown that features extracted at earlier layers of the CNN are different than those extracted at later layers, with those from earlier layers generally corresponding to low level features like edges, and those at deeper layers corresponding to higher level concepts like objects[189]. While there could be benefit from sharing features between layers, it more difficult than sharing within a layer, so we leave this for future work.

3.4 Generalizing Weight Sharing

We can also look at weight sharing more generally, and instead allow the network to learn which parameters to share instead of it being hardcoded. Given a network with a set of weights $W = \{w_1, w_2, ..., w_N\}$, assume there are only N^{basis} independent weights in the network, with all other weights being equal to one of these: $(w_0^{basis}, w_1^{basis}, .., w_{N^{basis}}^{basis})$.

Naively searching over weight sharing in a network is intractable, because the total number of configurations of weight sharing in a network is over 2^N, where N is the number of weights in the network. To simplify this problem we will limit it to searching for weight sharing within convolutions, which can reduce the search space to only one layer with many fewer parameters. In addition, we will not aim to search over all possible weight-sharing arrangements, the search will be limited to a fixed number of shared weights.

This reduces the problem significantly, so for a given layer l, we will aim to learn the set of N_l^{basis} basis parameters for this layer. This will be a vector of weights: $w_l^{basis} = w_{l,0}^{basis}, w_{l,1}^{basis}, .., w_{l,N_l^{basis}}^{basis}$. In addition, each weight in the network will have to take on the value of one of these basis weights. So we will also have to learn an assignment that maps each weight in a given layer N_l to one of the weights in N_l^{basis}. This is still computationally intractable to brute force search, but in this form the problem is constrained enough to approach using other methods.

3.5 Weight Sharing within Convolutions

With the problem of searching over weight sharing limited to within convolution kernels, we take inspiration from work where seemingly intractable problems were solved directly.

When creating sparse neural networks, there are a massive number of combinations of weights which could possibly be removed, but a simple heuristic of removing the weights with the smallest value is surprisingly effective. While removing weights directly is not our goal, we can develop methods to help weights converge toward the basis weights during training, and then quantize them directly to these basis weights. This contrasts with pruning, which can be thought of as quantizing the weights to 0.

3.5.1 Bayesian Priors - Gaussian Mixture Model

A standard method for regularization in regression is using the l_2 norm, which is equivalent to a Gaussian prior distribution over the parameters, with the mean set to 0. In addition, a Gaussian prior centered at any mean has the effect of encouraging the parameters to be close to this mean. Because of this, a prior in the form of a Gaussian Mixture Model (GMM) has the effect of encouraging parameters to become close to a fixed set of values, the means of the mixture components. In addition, we can allow these means to be trained, so that the network can learn both set of basis weights w^{basis}, which are the means of the components, as well as encouraging weights in the network to be close to these basis weights during training.

This strategy has been investigated in previous work[112][161], where it was used for quantization. In addition, this method can also be used to introduce sparsity into the network, just as with a standard Gaussian prior. We can fix one component as $\mu_j = 0$, and then all parameters that are quantized to this component can be set to 0. The level of sparsity can be adjusted based on the probability of that component, π_j.

For a mixture with J components, the probability of a set of weights $\boldsymbol{w}$ is shown as:

$$p(\boldsymbol{w}) = \sum_{j=1}^{J} \pi_j \mathcal{N}(\boldsymbol{w}|\mu_j, \sigma_j^2) \tag{3.1}$$

There is flexibility in how this model can be trained. It is possible to train both the model's weights w, as well as the GMM's parameters π, μ, σ jointly by maximum likelihood, which is an Empirical Bayes approach. In addition, the component probabilities, π can be trained with a hyperprior, or they can be fixed, for example fixing π_j corresponding to $\mu_j = 0$ to enforce some level of sparsity. In practice, to train this model we can add the negative log likelihood of the model parameters under this prior to the loss function along with some factor τ to train the model:

$$Loss = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i)) - \tau \log\ p(\boldsymbol{w}, |\mu, \sigma, \pi) \tag{3.2}$$

After training, the weights of the model should become closer to the basis weights, so they can be quantized to these values.

3.5.2 Bayesian Priors - Laplace Mixture Model

A Gaussian Prior corresponds to l_2 regularization, which encourages values to be a small distance from the mean, but not exactly equal to it. Using l_1 regularization instead of l_2 regularization was first suggested for the purpose of variable selection and increasing interpretability in standard regression models by selecting only a subset of variables to be used for the final regression model [160]. Because l_1 regularization encourages the absolute value of the parameters to be small, it performs variable selection, compared to the more standard l_2 regularization that only encourages small values. Because of this, a more appropriate prior to encourage weights to be exactly equal to the mean of the distribution is the Laplace distribution, corresponding to l_1 regularization:

$$p(w) = \frac{1}{2b} e^{-\frac{|w-\mu|}{b}} \tag{3.3}$$

We can also leverage the Laplace distribution to encourage better quantization of weights. Instead of using a single component and setting the mean $\mu = 0$ as in standard l_1 regularization, we can use a mixture of Laplace distributions, and allow the means to the learned. So instead of encouraging parameters to be exactly equal to 0, it encourages them to be exactly equal to the means of the components. This is implemented similarly to the Gaussian Mixture Model, but with the Laplace distribution for each of the components, as shown below:

$$p(w) = \sum_{j=1}^{J} \pi_j \frac{1}{2b_j} e^{-\frac{|w-\mu_j|}{b_j}} \tag{3.4}$$

This is beneficial because at the quantization step more weights will already be identical to the mean of the distribution, which is the basis weight these weights will be quantized to. Because the weights will have to change less during the quantization step, we expect the model's performance will not be harmed as much through quantization.

3.5.3 Loss functions

Taking inspiration form the approach of Gaussian and Laplace priors, we propose an alternative approach, where we design specific loss functions to encourage the weights to cluster in a sensible way which is useful for weight sharing. Sometimes loss functions can be equivalent to using prior distributions, as the l_2 loss is equivalent to a Gaussian prior, and the l_1 loss is equivalent to Laplace. We use loss functions in a slightly different form, where the loss will only be a function of the distance from a weight to its nearest basis weight.

We will use loss functions of the form:

$$loss = \sum_{j=1}^{N} \|min\{|w_j - w_i^{basis}|; \quad \forall i \in w^{basis}\}\|_p \tag{3.5}$$

Where N is the number of weights in the network, and the loss is interpreted as the minimum distance from a given weight to the nearest basis weight.

We can also enforce similarity between kernels by using this loss over entire kernels instead of individual weights. For example, we can use a fixed set of basis kernels, wk^{basis}, corresponding to the vector of weights representing a single kernel. The l_2 norm is used to indicate similarity between kernels. This is similar to group lasso[187]. Assuming N^k is the number of kernels in the layer, we can write this as:

$$loss = \sum_{j=1}^{N^k} \|min\{\|wk_j - wk_i^{basis}\|_2; \quad \forall i \in wk^{basis}\}\|_p \tag{3.6}$$

3.6 Experiments

We investigate if additional weight sharing within convolution layers is a useful way to reduce redundant parameters for image classification problems by evaluating on the CIFAR 10 and 100 datasets[80] as well as the Imagenet dataset[39]. We use standard and efficient architectures with varying amounts of weight sharing within the depthwise separable convolution layers to understand the effects of additional weight sharing.

3.6.1 CIFAR 10 and 100

We investigated adding additional weight sharing into Xception[25], a model that popularized the use of more parameter efficient depthwise separable convolutions. We tested this model on the CIFAR 10 and 100 datasets, both consisting of 60000 32×32 pixel images with 10 and 100 classes, respectively. We train our model for 300 epochs using SGD with momentum. A batch size of 32 is used and the learning rate is stepped down by a factor of 5 every 50 epochs. The initial learning rate is 0.1 with a weight decay of 0.0001 and momentum of 0.9. The model is trained on a single NVIDIA GTX Titan GPU and is implemented using PyTorch[114].

Looking at Figure 3.3, we see how the performance of the Xception model changes as additional weight sharing is added to the depthwise separable convolutions. This indicates that classification accuracy increases for both the CIFAR10 and CIFAR100 datasets with only half the number of independent parameters in the convolution layers. In CIFAR100, this is true even with 75% of the parameters in these layers being shared. Both these models were trained from scratch, using the standard Xception model, except changing the last layer to correspond to the correct number of classes, as well as changing the standard depthwise separable convolution to include weight sharing.

3.6.2 Visualization of the Effects of Weight Sharing on CIFAR

We visualize the difference between the standard Xception model and the model with additional weight sharing. Figure 3.4 shows a comparison between the kernels in the first layer at 4 levels of weight sharing, 100% (standard model), 50%, 25%, and 12.5%. Because we are visualizing the kernels in the first layer, taking the image as input, we can visualize the 3 channel input using color, although this would not be possible for any other layers in the network. Another special aspect of this layer is that this is the only convolutional layer in the network where weight sharing is not used. Standard dense convolutions are used for the input layer, compared to separable convolutions used in later layers of our version of the Xception architecture. The kernels are not presented in the order they are found in the layer, as this ordering is meaningless. Instead we have sorted the kernels based on cosine similarity to the baseline network (100% weight sharing). This allows an easier comparison to see differences between kernels learned under different amounts of weight sharing. Based on the kernels in figure 3.4, we see that the weight sharing in later layers of the network has minimal effect on the kernels learned in the first layer of the network.

In figure 3.5 we visualize the kernels in the second layer of the network. Weight sharing

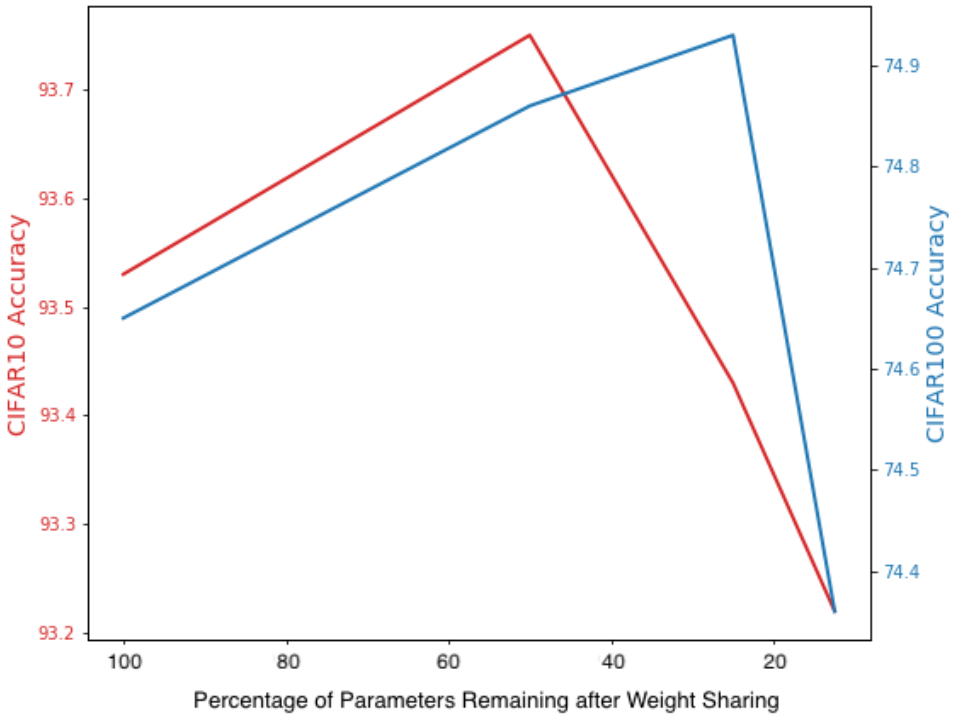


Figure 3.3: Test set classification accuracy of the Xception model with additional weight sharing on the CIFAR10 & CIFAR100 datasets. Performance increases with additional weight sharing, and reaches a maximum at 50% of original parameters for CIFAR10, and 25% for CIFAR100.

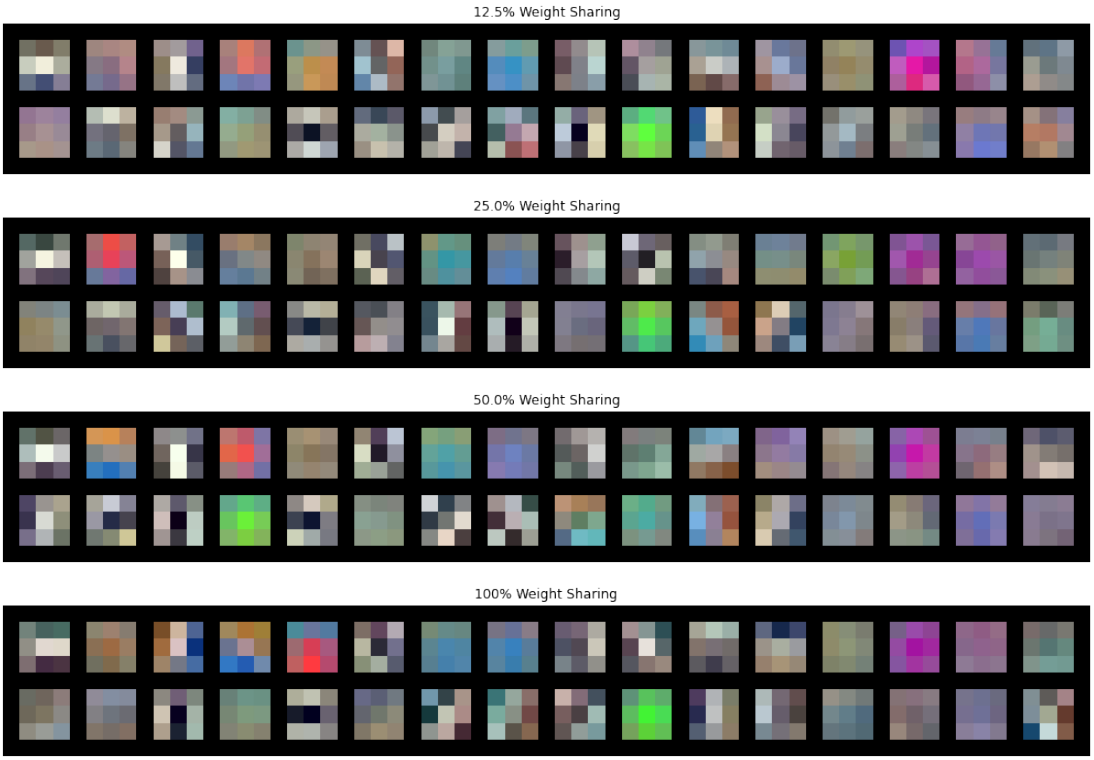


Figure 3.4: Visualization of the first layer kernels with varying amounts of weight sharing. Because this layer takes RGB images as input, we can visualize the full 3 channel kernel using color, and see that the amount of weight sharing has a minimal effect on the kernels learned.

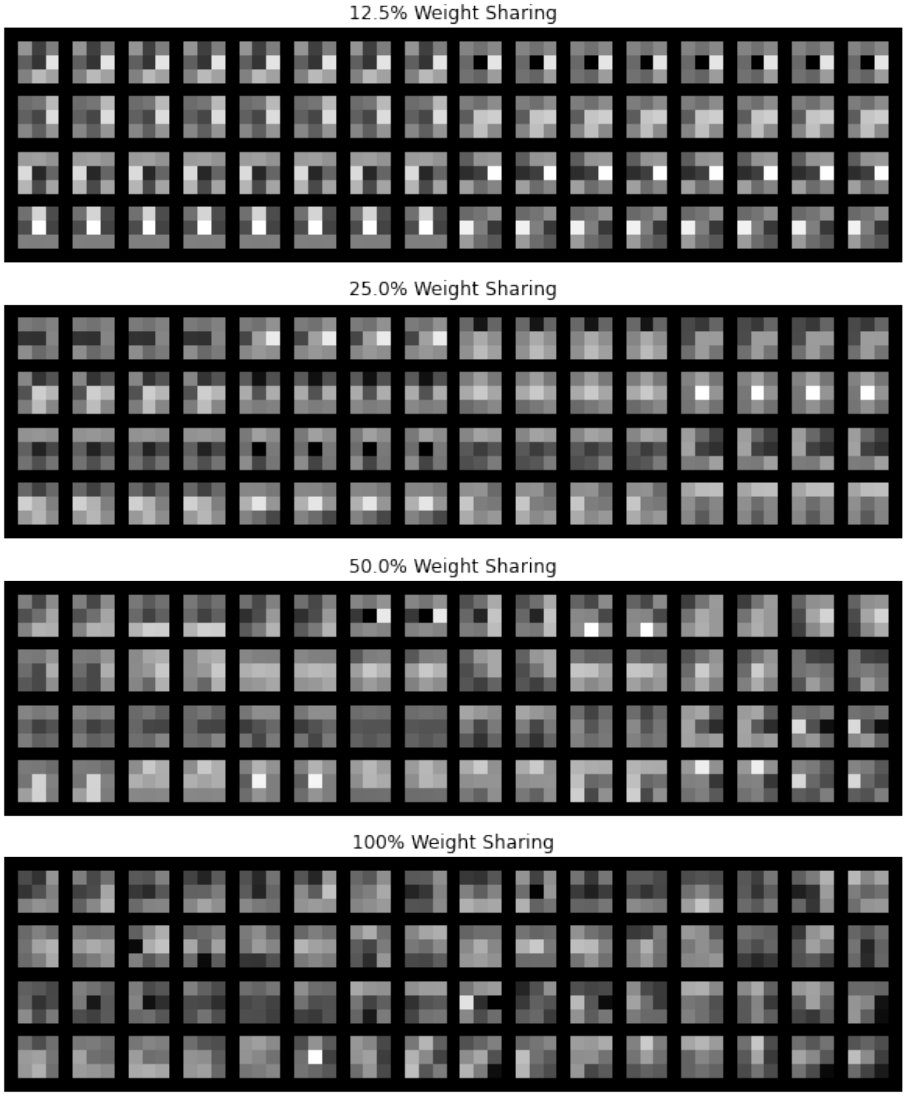


Figure 3.5: Visualization of the second layer kernels with varying amounts of weight sharing. The duplicated weights are clearly visible in the network, but there is no consistent change in the types of learned kernels through the addition of weight sharing.

is used at this layer so we see fewer unique kernels, but there is no consistent change in the types of kernels we see.

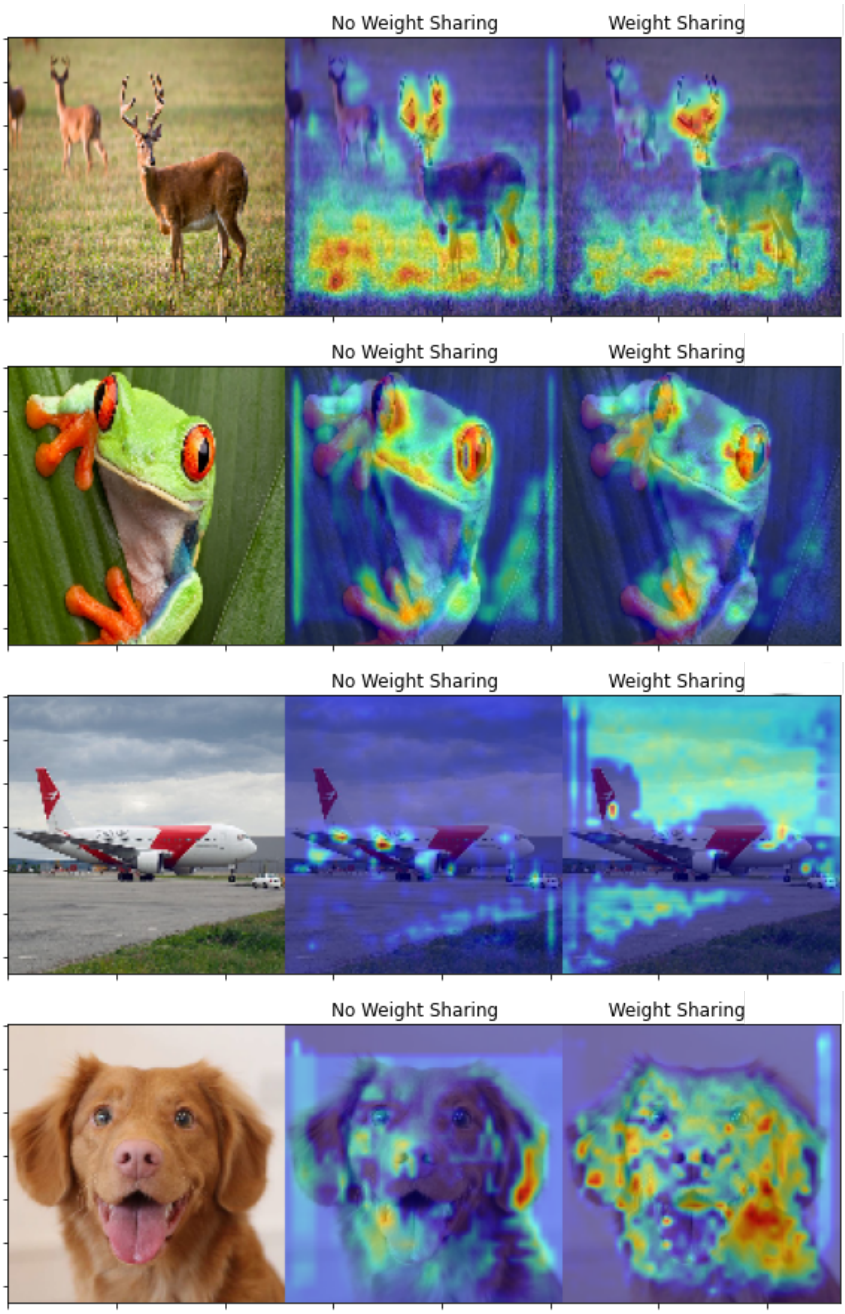


Figure 3.6: Visualization of class activation maps of a convolutional neural network using weight sharing with 25% convolution parameters compared to one using standard convolution layers with no additional weight sharing.

We also visualize the most relevant areas of the image used by the model for classification with and without weight sharing using GradCam[141]. This is an extension of Class Activation Maps[193], which generates heatmaps by multiplying the global average pooling layer's activations by the weights of the last linear layer that correspond to the class we are interested in. This allows us to see which parts of the image are most relevant to the classification. GradCam extends this method by looking at the gradients of the class of interest, and so is more flexible and can be used to create visualizations for different layers and different structured CNNs. Figure 3.6 shows visualizations created using GradCam for the CIFAR10 dataset, looking at the third convolutional layer and the activations corresponding to the correct class. We see that the class activation maps may be more diffuse for the model with additional weight sharing, but further investigation is needed to determine if this is a consistent effect.

3.6.3 Imagenet

We also investigated this method on a more difficult benchmark, Imagenet[39], containing 1.2 million high resolution images divided into 1000 classes. For this we use a much stronger baseline architecture called EfficientNet[159]. This is a model explicitly designed to be parameter efficient by building on a variety of previous developments. These include depthwise separable convolutions[148, 25], inverted bottleneck residual blocks[135], squeeze-and-excitation blocks[68] all optimized using a multi objective neural architecture method designed to optimize accuracy and model size jointly[158].

There are a variety of EfficientNet architectures trading off model size and accuracy, but we use the smallest version of this model, B0, containing 5.3 million parameters. It is trained from scratch using RMSprop optimizer[62], a learning rate of 0.072 and a linear learning rate warmup. A batch size of 120 is used and it is trained for 550 epochs using distributed training on 8 NVIDIA V100 GPUs. The model is implemented using PyTorch[114].

% of Weights Remaining in Separable Convolution	**Accuracy (top1)**	**Accuracy (top5)**
100% (Original implementation)	76.3%	93.2%
100% (PyTorch implementation)	77.2%	93.4%
50%	77.1%	93.4%
25%	76.8%	93.3%

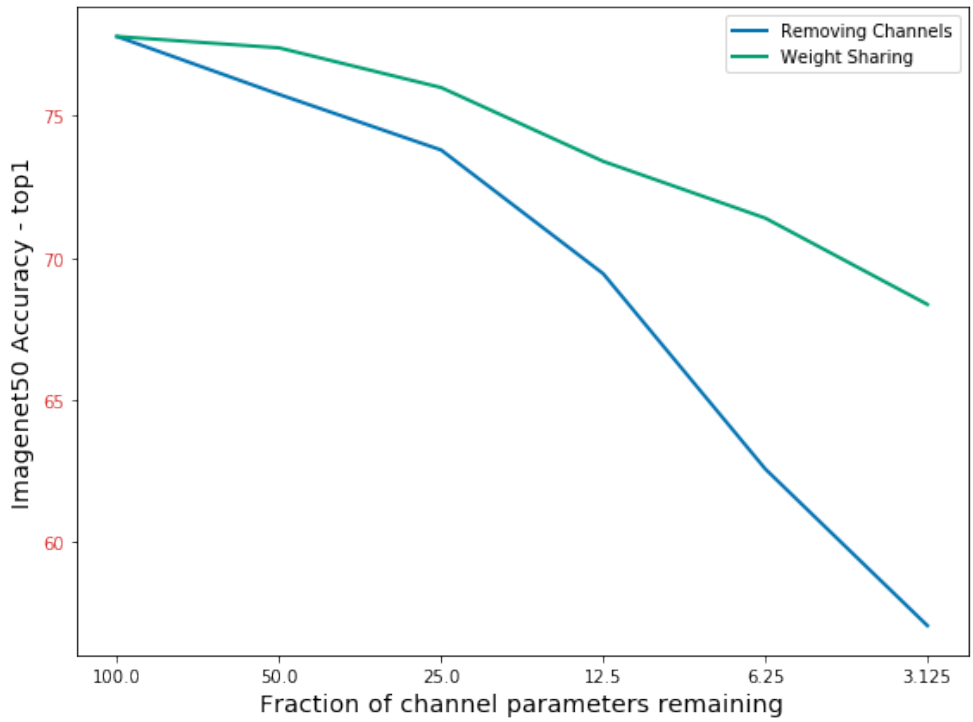


Figure 3.7: Using a subset of Imagenet and the Resnet50 model, we compare the performance when removing channels from the model vs. using weight sharing. We see that weight sharing is more effective than channel removal across all levels of weight sharing.

Table 3.6.3 shows the results of these tests, indicating that 50% of the convolution parameters can be removed through weight sharing with no significant loss in accuracy, and at 25% of the original convolution parameters the model still performs competitively. We note that accuracy here is slightly higher than in the original implementation because we modified the training schedule to train for more epochs (550 vs. 300).

3.6.4 Weight Sharing vs. Channel Removal

We also compare weight sharing to removing channels entirely from the network. This is to ensure that the improvement is not simply due to the fact that it is possible to use a much smaller model and achieve the same results, but rather that weight sharing is a useful technique in its own right.

In figure 3.7 we see that weight sharing is more effective than channel removal across all levels of weight sharing, although weight sharing is less effective on the Resnet50 model compared to other efficient models such as Xception[25] and EfficientNet[159]. This is demonstrated on a subset of the Imagenet dataset with 50 classes compared to the standard 1000, and using the Resnet50[60] model.

3.6.5 Generalized Weight Sharing

Experiments in generalizing weight sharing were less successful than fixed weight sharing, and suffered from an unacceptable level of accuracy loss. We include results of some experiments using loss functions to encourage a different distribution of weights during the process of training the model.

In Figure 3.8 we see that the approach of using loss functions introduced in section 3.5.3 to encourage a certain distribution of weights works well, as the weights learn to cluster around these basis weights during the training process. The top half of the figure shows the initial distribution of weights before training, and the bottom half shows the final distribution after training 75 epochs on CIFAR10. In this case there was no component fixed at 0, but this could be added to encourage sparsity. Also, there was no constraint placed on the number of weights mapped to a given basis weight, but this is another constraint that can be added. For these more complex constraints, viewing the problem in terms of using a prior distribution introduced in section 3.5.1 is another option.

In addition, we have done some preliminary work on using loss functions to encourage redundancy between kernels. As shown in Figure 3.9, the addition of the group lasso based loss function (3.6) using l_2 for both the within and between group norms shows a clear advantage over not using this loss before quantization. Unfortunately this harmed performance too much to be practically useful, but it could be possible to overcome these difficulties using alternative techniques.

3.6.6 Discussion

This chapter proposed that convolution kernels used for image classification tasks contained representational redundancy in terms of unnecessary flexibility between channels. We tested this hypothesis by adding additional weight sharing within convolution kernels, and showed this can preserve or even improve performance.

Using the Xception architecture, which was designed to be more parameter efficient compared to regular convolution networks, we further reduced the number of parameters

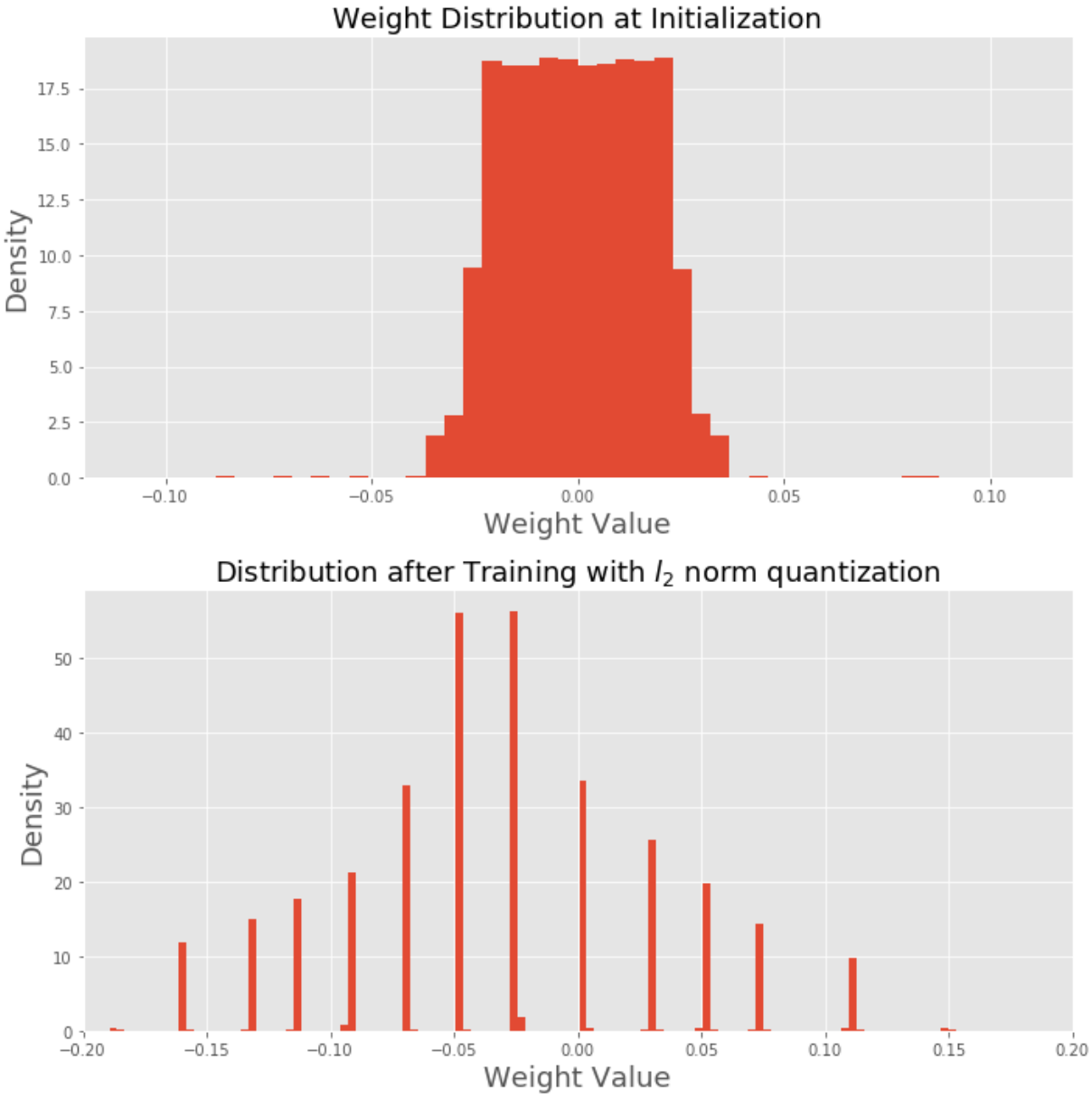


Figure 3.8: Distribution of weights before and after training using the loss function with 16 basis weights with the l_2 loss. This clearly encourages the values to cluster near the basis weights, forcing a weight distribution that is useful for weight sharing.

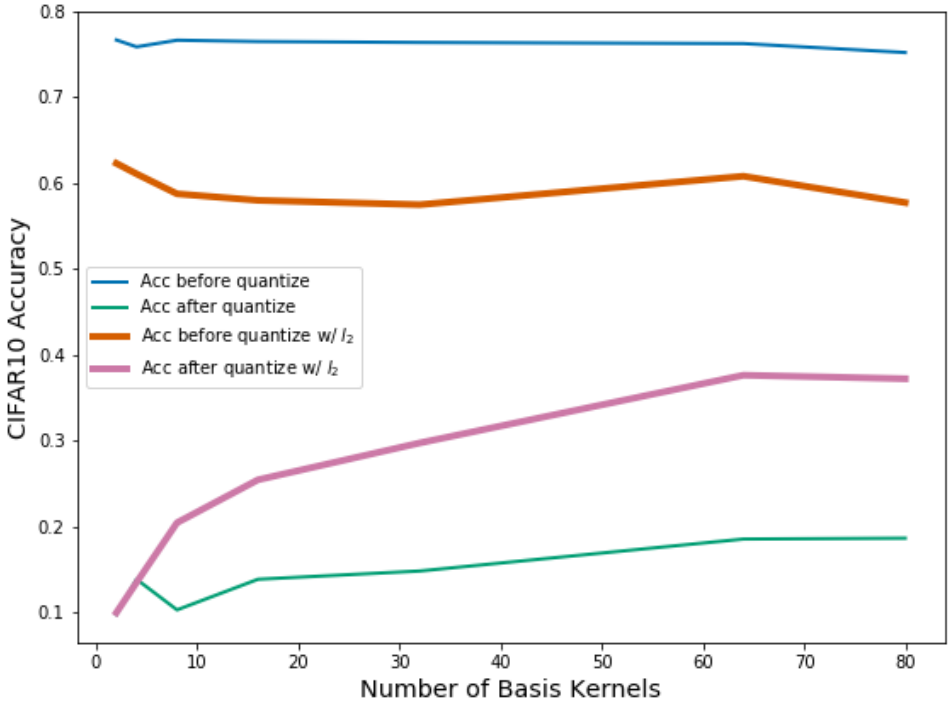


Figure 3.9: Comparing the effect of l_2 regularization and quantization CIFAR10 performance. The addition of the l_2 group norm reduces loss during quantization, but still performs quite poorly overall. Comparing the blue and green lines, we see that quantization after training the model normally destroys the model's performance. In contrast, comparing the orange and purple lines we see that the additional loss harms performance, but also results in less performance drop after quantization.

needed through the use of additional weight sharing and we found accuracy can be increased on the CIFAR10 and CIFAR100 datasets with a reduction in the number of convolution parameters of 50% and 75%, respectively. On the Imagenet dataset we see that even with a more complex and already heavily optimized model, EfficientNet, additional weight sharing within convolutions can also be used to reduce the number of parameters in convolution layers by 50% with almost no decrease in accuracy.

In addition, we investigated extending this approach to fully learning the weight sharing assignments, and showed our proposed loss functions can encourage weight clustering during training and reduce loss during a quantization step. While we find value in these explorations, the fact remains that in this case, as in many others, simple approaches can outperform more complex ones. Similarly to how simple magnitude based pruning tends to outperform more complex algorithms[48], we find the simple hard coded approach to weight sharing outperforms more complex approaches.

Having found a general form of representational redundancy inside convolutional kernels in the form of weight sharing, we now turn to the question of removing representational redundancy in the latent space of generative models in Chapter 4. We aim to demonstrate a specific form of representational redundancy through the disentanglement of shape and orientation parameters, leading to a more efficient latent representation of data.

Chapter 4

Compressed Representations with Affine Variational Autoencoders

In this chapter we extend our work on the reduction of representational redundancy to the domain of generative models. Here we focus on the latent space of generative models, in this case Variational Autoencoders, instead of the parameters of the convolution as in the last chapter. We aim to create a more compressed representation of the data itself, and our approach will not only lead to more efficient representations, but also to a more human interpretable and disentangled representation as well.

4.1 Affine Variational Autoencoder

In this section we describe the Affine Variational Autoencoder (AVAE) [14], as well as the novel training procedure used to encourage disentanglement of shape and orientation.

4.1.1 Affine Transforms

The key innovation in the proposed AVAE architecture is the introduction of two affine layers on top of the VAE architecture. Unlike the spatial transformer network, the proposed AVAE architecture does not output the parameters of the transform through a localization network. As a result, each affine layer in the AVAE consists of two parts: (i) a sampling grid, where the grid of coordinates in the input associated with each pixel in the output

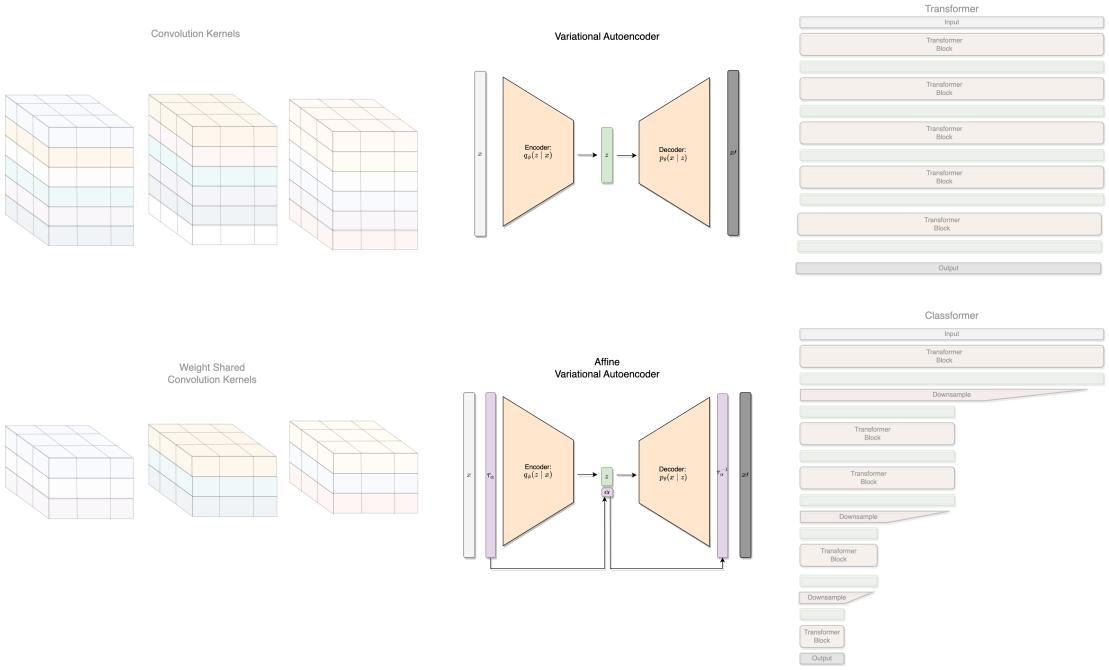


Figure 4.1: Affine Variational Autoencoder (below) compared to the original Variational Autoencoder (above). The AVAE is trained create a more compressed latent representation by disentangling shape and orientation through the use of affine transform layers and a novel training procedure.

is determined for a given transform, and (ii) image sampling, where bilinear sampling is applied to the input given the derived grid.

More specifically, the affine layers in the AVAE architecture both take an affine transform as input, parameterized by an 3×3 affine transform matrix, α.

$$\alpha = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \alpha_4 & \alpha_5 & \alpha_6 \\ \alpha_7 & \alpha_8 & \alpha_9 \end{bmatrix} \tag{4.1}$$

For example, in the case of rotations, the resulting transform can be represented as:

$$\alpha = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

The key difference between the two affine layers in the AVAE is that the first affine layer performs an affine transform on the input directly based on α, while the second affine layer performs an affine transform on the input based on the inverse of α.

4.1.2 Affine Variational Autoencoder Architecture

The AVAE extends the conventional VAE architecture with the introduction of two affine layers. More specifically, the input data is fed into the first affine layer, which performs an affine transform on the input before passing it into an encoder to create a latent space representation. The output of the decoder in the AVAE architecture is fed into a second affine layer, which performs the inverse affine transform to the output of the decoder, producing the final output. The parameters of the affine layers are learned such that the resulting AVAE can effectively encode input images at canonical orientations, which results in a more compressed representation of the latent variables and a disentangled latent space. Similar to the VAE, we assume the data $\boldsymbol{X} = \{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$ are generated by an unobserved latent variable z. In this case we assume the prior distribution of $\boldsymbol{z} = [z_1, z_2, ..., z_k]$ can be written as:

$$p(\boldsymbol{z}) = p(z_1, z_2, ..., z_p, z_{p+1}, ..., z_l) = p(z_1, z_2, ..., z_p)p(z_{p+1})...p(z_k) \tag{4.3}$$

We then relabel $p(\boldsymbol{z})$ as $p(z_1, z_2, ..., z_p)p(\alpha_1)...p(\alpha_k)$. We consider $\boldsymbol{z}$ to be latent variables representing shape, $\boldsymbol{\alpha}$ to be those representing orientation, and assume these are independent.

With a fixed orientation, $\boldsymbol{y}$, this would further simplify the problem because we could model the conditional distribution $p(z_1, z_2, ..., z_p)p(\alpha_1 = y_1)...p(\alpha_k = y_k) = p(z_1, z_2, ..., z_p)$. But this is limited to a single orientation, and we would like a model that generalizes. Assuming we have a standard VAE to model $p(x, \alpha = y)$ for the distribution with fixed orientation, how could we make it generalize to more orientations?

We would like a transform, τ_α that can transform an object x to the correct orientation for our VAE, $\boldsymbol{y}$, along with a corresponding inverse transform τ_α^{-1} to transform the object back to its original orientation. This is an affine transform, so both of these are straightforward to implement, and can be added to the standard VAE, resulting in the AVAE shown in figure 4.2.

Assuming we have a trained AVAE for a single orientation, how can we apply this to a randomly oriented input and find the right affine transform parameters? We can take advantage of the fact that the VAE indirectly learns a distribution over the data, $p(x)$.

The VAE is trained to maximize a lower bound on the log-likelihood of the data, so for a image x, we can use the loss to approximate $p(x)$. Assuming the VAE was trained on a distribution with fixed orientation $\boldsymbol{y}$, samples at any other orientation should have low $p(x)$. To find the correct orientation, we should optimize the transform parameters, α to minimize the VAE's loss. Given a VAE with encoder q_ϕ, and decoder p_ρ, and the invertible transformation τ_α, we optimize α to maximize the likelihood of x under the model:

$$\alpha^\star = \underset{\alpha}{\operatorname{argmin}} \left\{ L_{VAE}[\tau_\alpha^{-1}(p_\rho(q_\phi(\tau_\alpha(x))))] \right\} \tag{4.4}$$

Affine transforms can be made differentiable [72], so this can be optimized using gradient descent. In practice, this optimization is difficult and likely to be caught in a local optima, so random restarts are required.

4.1.3 Disentangling Orientation and Shape

Given a dataset $\boldsymbol{X} = \{\boldsymbol{x}^{(i)}\}_{i=1}^N$ distributed randomly over affine transforms, we use the observation that the conditional distribution of $\boldsymbol{z}$ given a fixed orientation, $p(z_1, z_2, ..., z_p|\alpha_1 = y_1, ..., \alpha_k = y_k)$, is less complex than the full distribution. This means that given a limited capacity model, the most efficient solution is to encode the objects at a fixed orientation because this simplifies the distribution to be modelled. But how can we do this when we have a dataset with objects at random orientations?

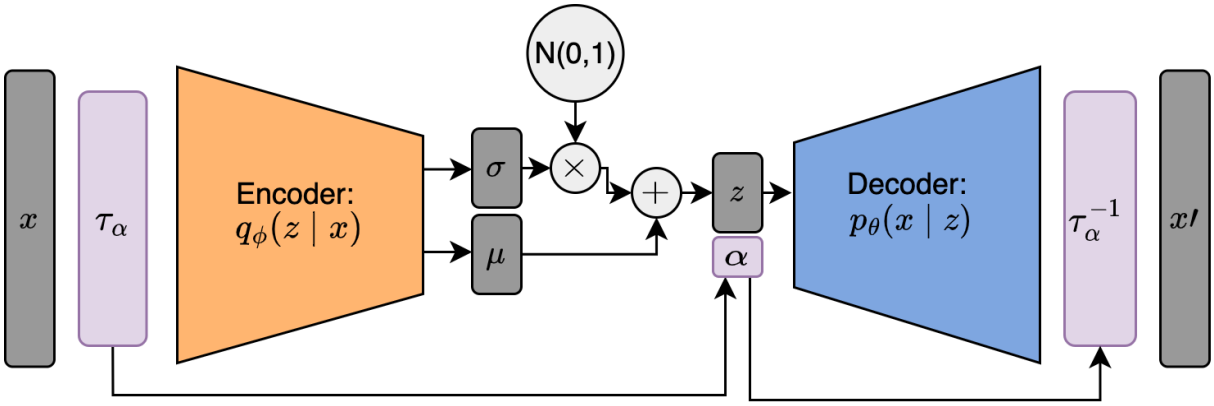


Figure 4.2: Affine Variational Autoencoder (AVAE) extends the conventional VAE by introducing two affine layers, the first performing an affine transform to the input, parameterized by α. This is encoded and decoded by the VAE, and finally the second affine layer does the inverse transform, producing the final output, x'.

We train the AVAE on this dataset, but before each step of SGD, the affine parameters are optimized using the above procedure. The goal is that as training progresses, the representation will be progressively more disentangled as the model learns to transform the data to a fixed orientation.

4.1.4 2d and 3d AVAE

We use a version of the AVAE for both 2 and 3 dimensional inputs. This extension to 3d is straightforward by substituting the 2d convolutions with 3d convolutions and extending the affine transform layer to a 3d affine transform. In addition, we have to change the form of the likelihood to accommodate the different formats for 2d and 3d datasets.

For the 2d MNIST dataset, we take the likelihood, $p_\theta(x|z)$, to be an isotropic Gaussian distribution with the variance fixed as 1. For the 3d ModelNet dataset, because we take the voxels as representing the volume of an object, we assume each voxel should take values in $\{0, 1\}$. For this reason we use a Bernoulli distribution for the likelihood.

4.1.5 Design Choices

For optimization of the affine transform, we use 32 random restarts. We select the 8 best parameters and perform 20 steps of gradient descent on them using the Adam optimizer [76], finally selecting the one with lowest loss.

We use a VAE architecture based on a previously successful implementation[139], which uses and encoder composed of four convolutional layers of sizes [32, 32, 64, 16], and a decoder composed of transposed convolutions of sizes [32, 16, 16, 1]. For the first three layers of both the encoder and decoder, Exponential Linear Unit activation functions (ELU)[29] are used as well as batch normalization[71].

4.2 Experimental Results

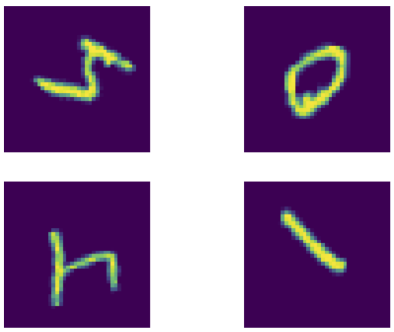
In this section we perform experiments to verify that affine transformed data is more complex than data at a fixed orientation, and that standard VAEs do not generalize well to affine transformed data unless they are explicitly trained on it using data augmentation. We show a standard VAE requires a higher capacity in terms of a greater latent dimension to encode affine transformed data. We also show how the AVAE is comparatively more efficient at encoding these distributions, giving a more compressed latent representation with better reconstruction error. In addition, we show the AVAE learns disentangled representations of shapes and orientations. Experiments are done on both a 2 dimensional image and 3 dimensional objects dataset, and all experiments were implemented using PyTorch [114].

4.2.1 Datasets

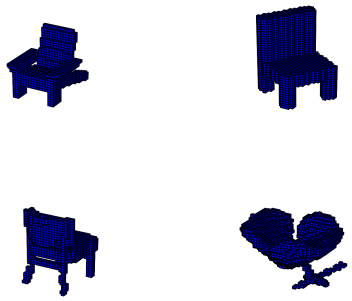
We include datasets from both 2d and 3d domains. For the 2d domain we use the MNIST dataset, and for the 3d domain we use the ModelNet dataset.

MNIST

The MNIST dataset is a set of grayscale numbers between 0 and 9, sized 28×28. We do standard preprocessing by normalizing pixel values and also pad the image with zeroes to a size of 40×40 to ensure there is no distortion under rotation or translation, shown in figure 4.3(a).



(a) MNIST Dataset



(b) ModelNet Dataset, chair class

Figure 4.3: Examples from the MNIST and ModelNet object datasets, both shown with random orientations. MNIST is represented as a 2 dimensional grayscale image, while ModelNet is a 3d voxel object.

ModelNet

To show the performance of the AVAE on 3d objects we use the ModelNet dataset [178]. It is composed of 3d voxel images of common objects, shown in figure 4.3(b). For this work we use the 10 class version of the dataset, and focus in particular on the sofa and chair classes. The images are padded to $48 \times 48 \times 48$ to allow for rotation and translations without distortion.

4.2.2 Complexity of Affine Transformed Data

VAEs can only encode data into a latent representation when they were trained on that type of data. While a VAE can easily encode and generate examples of digits, the model is unable to generalize to rotated digits unless it was explicitly trained on it. Here we explore the generalization of VAEs under rotations, and will later show that the standard solution to this problem of using rotation augmentation requires a higher latent capacity.

As shown in Fig. 4.4, the performance of a conventional VAE architecture decreases as they are forced to encode images that deviate from the training set, in this case in the form of a rotation. The loss reaches a maximum around 100 degrees, with it decreasing after this. This is because many digits look similar when vertically flipped, like 1 or 8, so

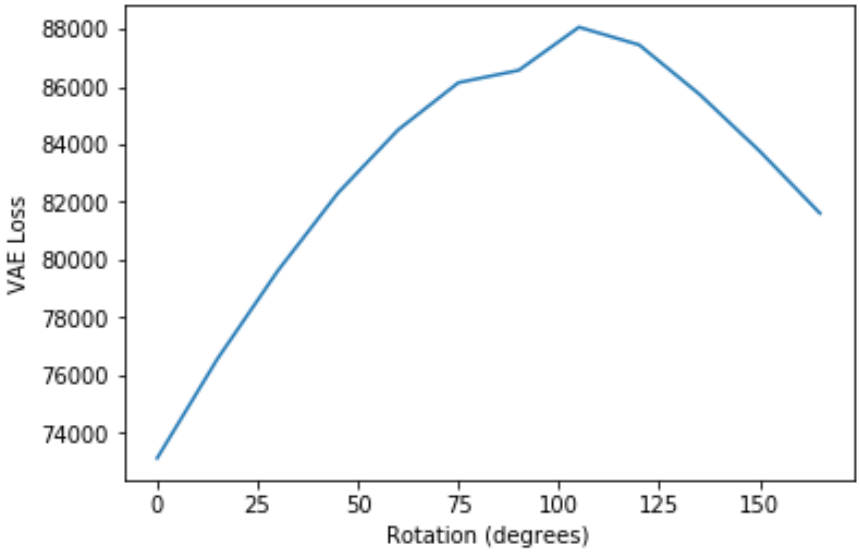


Figure 4.4: Average loss of a VAE over the MNIST validation set while varying rotation. Model was trained without data augmentation, so does not generalize well to novel orientations as the input image deviates from the training set.

models can effectively reconstruct those digits with a 180 degree rotation. Fig. 4.5 visually shows the inability of the VAE to encode and decode images after they have been rotated.

Now we directly evaluate the complexity of encoding the full distribution of rotation augmented data with a VAE. We compare the standard VAE on rotation augmented data to one on data of a single orientation and show for a given latent dimension the loss is higher with rotation augmented data compared to data at a fixed orientation. This is to verify our hypothesis that the rotation augmented data is a more complex distribution so will require a higher capacity model.

For the MNIST dataset, in figure 4.7, we vary the latent capacity for the VAE while comparing rotation augmented data to the single orientation. We see that reconstruction error in terms of mean squared error is greater for the rotation augmented data for any given latent size. At smaller latent sizes this difference is most pronounced, but as latent dimension increases performance converges. This also fits with our assumption, as we would expect that in theory a very high capacity model could achieve the same performance on randomly rotated and single orientation data, as latent capacity is not a limiting factor.

We demonstrate a similar result when also adding shear transformations. As shown in Fig. 4.6, it can be observed that there is a clear improvement in the performance of the

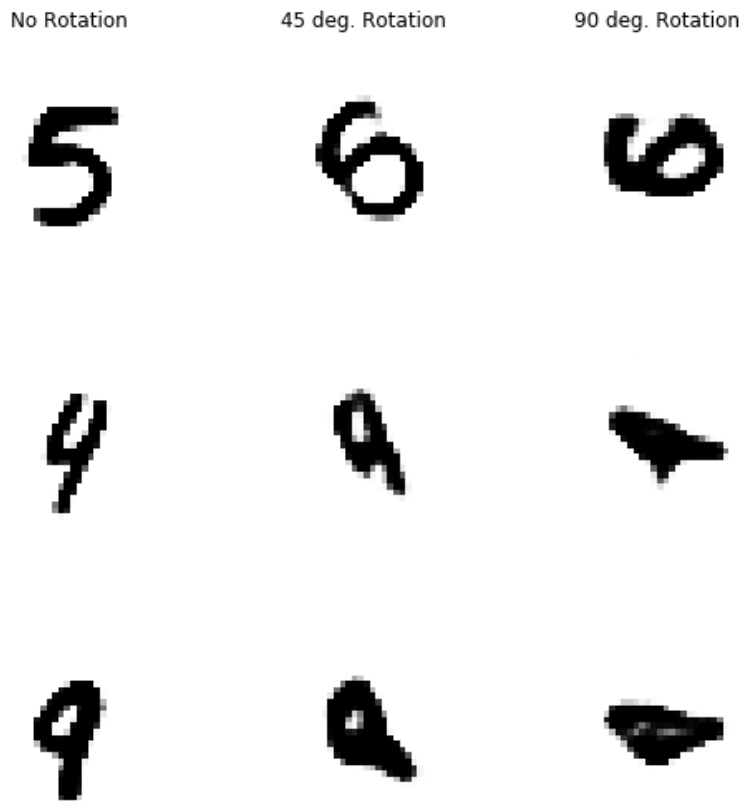


Figure 4.5: Examples of reconstructed images from a VAE trained on MNIST with no rotation, 45 deg. rotation, and 90 deg. rotation. It can be clearly observed that the quality of the reconstructed images degrade significantly under rotation.

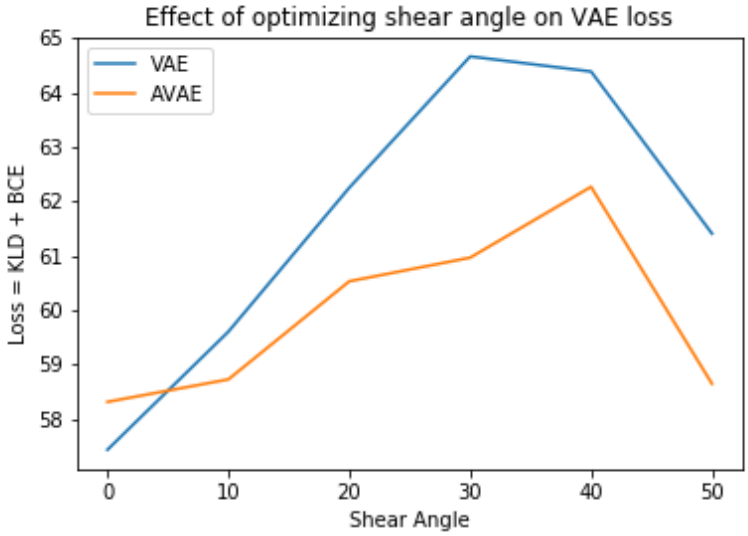


Figure 4.6: Comparison of the performance of the VAE and AVE under shear perturbations of various angles. AVAE shows overall improved performance compared to the VAE.

proposed AVAE architecture compared to the conventional VAE architecture in terms of loss for a given latent size across most shear perturbations. The improvement of AVAE over VAE is not as significant as in the case of rotational perturbations due to the fact that the conventional VAE architecture tends to generalize better to shear perturbations than rotational perturbations, but in general the AVAE works well when applied to a variety of affine transforms.

For the ModelNet dataset, in figure 4.8 we vary the dimension of the latent space used to encode the objects. We consistently see the model for all rotations has higher mean squared error (MSE) compared to the model for a single orientation, confirming the hypothesis that the distribution of rotation augmented data is more complex. This is true for all latent vector sizes, but becomes less large as the latent size increases and model capacity is less of a limitation.

We note that for both classes the difference between the rotation augmented and the single orientation data is larger than what was seen that on the MNIST dataset. The loss is twice as high when using the latent size of 4 on the ModelNet dataset compared to only 15% higher on MNIST. This is because rotations added greater complexity to ModelNet than MNIST, which also indicates there is more room for improving performance with the

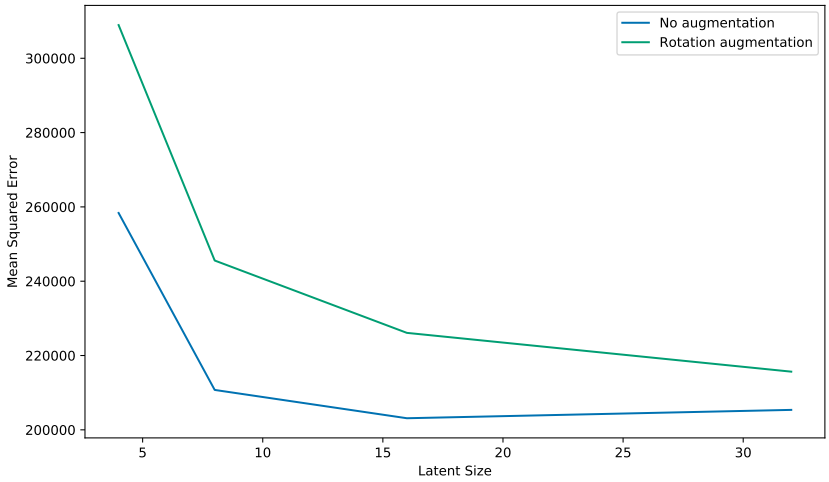


Figure 4.7: Average reconstruction error (MSE) of the VAE with and without rotation augmentation on MNIST. MSE is greater for the rotation augmented data with any latent size, and the difference is most significant with limited latent capacity

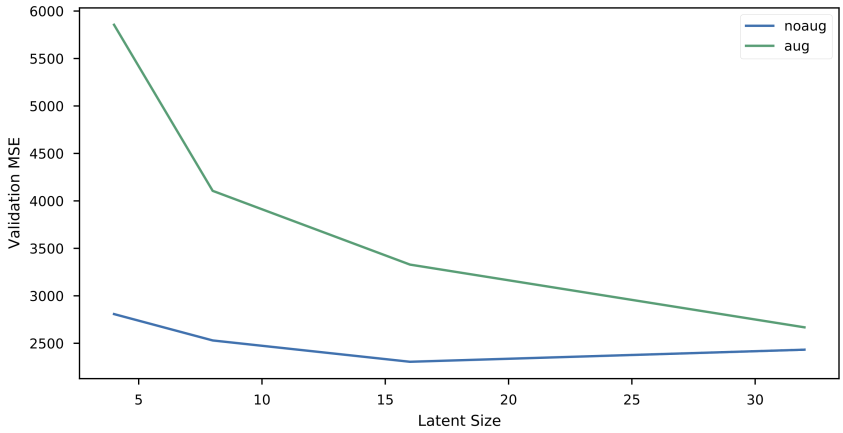


Figure 4.8: Average MSE of a VAE over the ModelNet validation set (sofa class) for different latent sizes, compared to the same dataset with rotation augmentation. Performance is lower on rotation augmented data, indicating rotation augmented data has a more complex distribution.

introduction of the AVAE.

These experiments clearly show that a larger latent size is needed to encode the more complex affine transformed data compared to data at a single orientation, both for 2d and 3d objects. We next show that the AVAE can leverage this observation to reduce redundancy in the representation by disentangling the affine transform parameters from the shape parameters.

4.2.3 Compressed Representations with AVAE

In this section, we evaluate the performance of the AVAE in generalizing to affine transformed data. We aim to show for a given latent size, the AVAE outperforms the standard VAE in terms of reconstruction error, indicating a more compressed latent representation. We intentionally use models with small latent dimension to make the differences between the models more clear.

In the AVAE the latent space is explicitly separated into the affine transform parameters for affine transform layers and the shape parameters from the VAE part of the model. Because of this we have to add additional dimensions to the latent space of the standard VAE model to make the comparison equivalent. For rotation augmented data we use a

latent size of 7 which is composed of a 6 dimensional VAE and the single rotation parameter, $[r]$. For rotation & translation augmentation we use a size of 9, which is composed of the 6 dimensional latent space, the rotation parameter and translation parameters, $[r, t_x, t_y]$.

In table 4.1, we see that the AVAE outperforms the standard VAE for both rotation and rotation & translation augmentation by decreasing MSE by 12% and 10% respectively. It is more efficient to explicitly use a parameter for the rotation angle which is learned by the AVAE rather than to leave it to the model to learn its own mapping.

Table 4.1: Improvement of AVAE over VAE on MNIST (MSE)

Augmentation	Validation set (all classes)
Rotation	12%
Rotation & Translation	10%

Table 4.2: Improvement of AVAE over VAE on ModelNet (MSE)

Augmentation	Sofa class	Chair class
Rotation	30%	48%
Rotation & Translation	38%	18%

We also compare the AVAE to the standard VAE on a rotation augmented version of the ModelNet dataset. Because the AVAE uses 3 additional rotation $[r_x, r_y, r_z]$ parameters, we compare the AVAE with 3 orientation parameters and 16 shape parameters to a standard VAE with 19 dimensional latent space.

We also test this procedure using rotations and translations. Here there is an additional 6 parameters, 3 for rotation and three for translation, $[r_x, r_y, r_z, t_x, t_y, t_z]$, so we compare the 16 dimensional AVAE to a VAE with latent size 22. As shown in Table 4.2 for both the sofa and chair classes the AVAE shows significant improvement over the standard VAE. This is also true for augmented with both rotation & translation, and here we see a significantly larger improvement compared to the MNIST dataset.

Fig. 4.9 shows specific examples of the poor reconstruction performance of the conventional VAE architecture when generalizing to rotation perturbations, while the AVAE performs well under such perturbations. Overall this indicates that by leveraging the proposed AVAE architecture, one can construct generative models that can generalize well to images under rotations.

Figure 4.9: Examples of reconstructed images using conventional VAE architecture (top row) and the proposed AVAE architecture (bottom row) under rotational perturbations.

These experiments have clearly shown the superiority of the AVAE over the VAE in terms of a more compressed latent space through better generalization to affine transformed data. Next we further investigate why this is the case, and show it is the result of the AVAE's ability to disentangle orientation and shape during training.

Training Dynamics of the AVAE

We investigate the training dynamics of the AVAE by comparing its loss to that of the standard VAE for a given number of optimization steps for the MNIST dataset, as shown in figure 4.10. We see the standard VAE has a normal loss curve, where the loss quickly goes downward for the first few epochs, and decreases more slowly after. In contrast, the AVAE's loss remains high for the first few epochs, before it eventually decreases quickly to a lower value than the standard VAE. Looking to the ModelNet dataset, we see this same training behavior, with the VAE first outperforming the AVAE before converging to a lower loss, as shown in figure 4.11

This is because the AVAE is initially limited by having its latent space divided between shape and orientation parameters. Initially the rotation optimization process is useless because the model has not learned to encode any orientation well, so the AVAE is forced to encode all orientations with its more limited latent size. As the training progresses the model learns to encode only a single orientation so the loss drops quickly. By looking

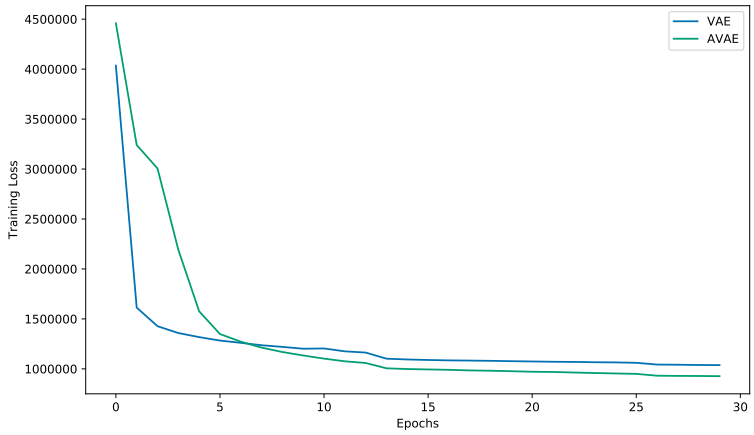


Figure 4.10: Training of VAE vs. AVAE on the MNIST dataset with rotation augmentation. AVAE takes longer to converge to a low loss because it takes a few epochs for the model to learn encode digits at a particular orientation.

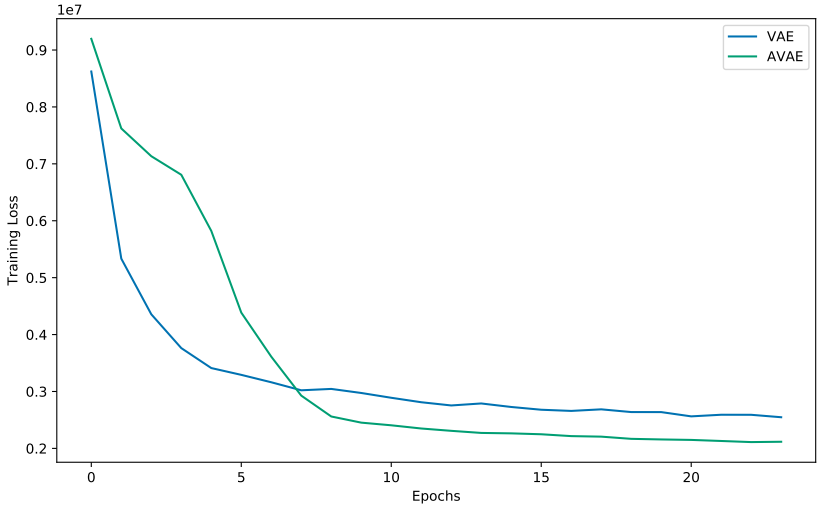


Figure 4.11: Training of VAE vs. AVAE on the ModelNet dataset with the sofa class, similarly to MNIST the AVAE first has higher loss before it learns to encode data at a particular orientation.

further into the distribution of orientations during training for a single class on the MNIST dataset we can see this disentanglement more clearly.

Disentanglement During Training

The AVAE's improvement over the standard VAE in terms of training loss that we discovered above corresponds to the point where the AVAE learns to disentangle orientation and shape. In practice this means the AVAE learns to use its affine transform layer to transform objects to a fixed orientation, so the encoder is able to tackle the easier task of encoding shape alone, which is demonstrated by looking at the rotations digits are encoded at during training of the AVAE.

When training a VAE normally with rotation augmentation, we should expect the model see images distributed uniformly over $[0°, 360°]$. But the AVAE optimizes rotation before encoding, so this is no longer the case. Optimizing the affine transform during the training process allows the AVAE to learn a more efficient representation by changing the distribution of rotations that digits are encoded at.

We look at how digits are rotated by the AVAE during the training process, in this case looking at the '1' digit. At the first epoch of training the model encodes each digit at a relatively uniform distribution over rotations, as seen in the top graph of figure 4.12. The model hasn't learned to encode any rotation better than another, so the optimization of rotation during training is useless, returning a random distribution over rotations. This is what we would expect to see when training a standard VAE.

As training progresses, shown in the lower graphs in figure 4.12, the model becomes biased towards encoding digits at particular orientations. It learns that it is better to encode only a subset of the true distribution to better utilize the limited latent capacity of the model. This is consistent with our earlier observation that it takes a lower capacity model to encode a single rotation compared to all possible orientations of images.

In figure 4.13, we are comparing the digits '6' and '9'. Because these digits have no rotational symmetries, they are encoded at a single orientation, leading to the unimodal distribution seen for both digits. But these digits are quite similar, nearly being 180° rotations of one another. Because we train the AVAE to jointly encode all classes of MNIST, we see it learned a more efficient representation between digits too, encoding the "6" and "9" digits at 180° rotations of one another to simplify the data distribution.

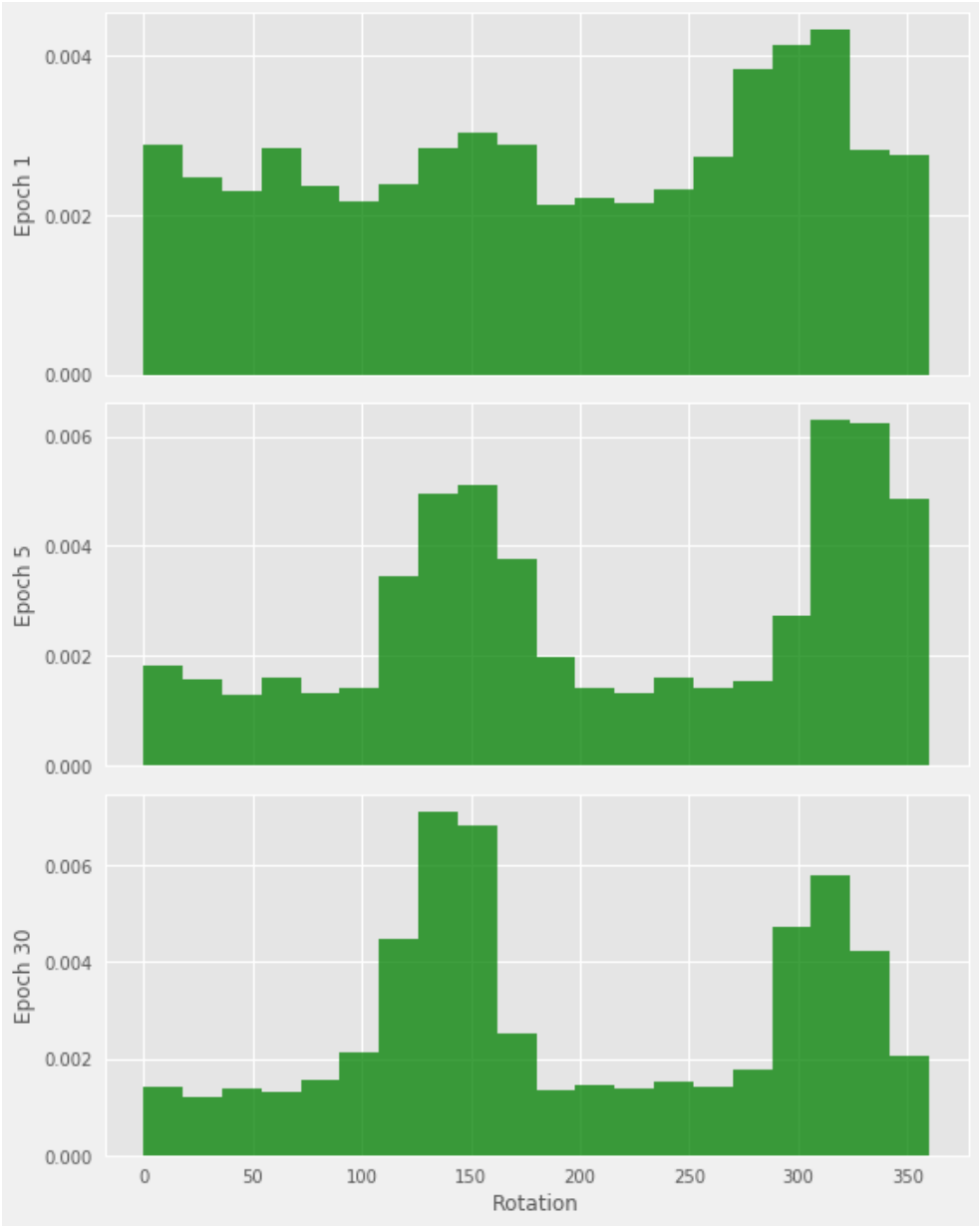


Figure 4.12: These histograms show the orientation that digits are rotated to by the affine transform layer in the AVAE. We show the '1' digit during training of the AVAE at epochs 1, 5 and 30. Initially the AVAE rotates them randomly, but as training progresses it learns to encode most digits at two orientations 180° apart.

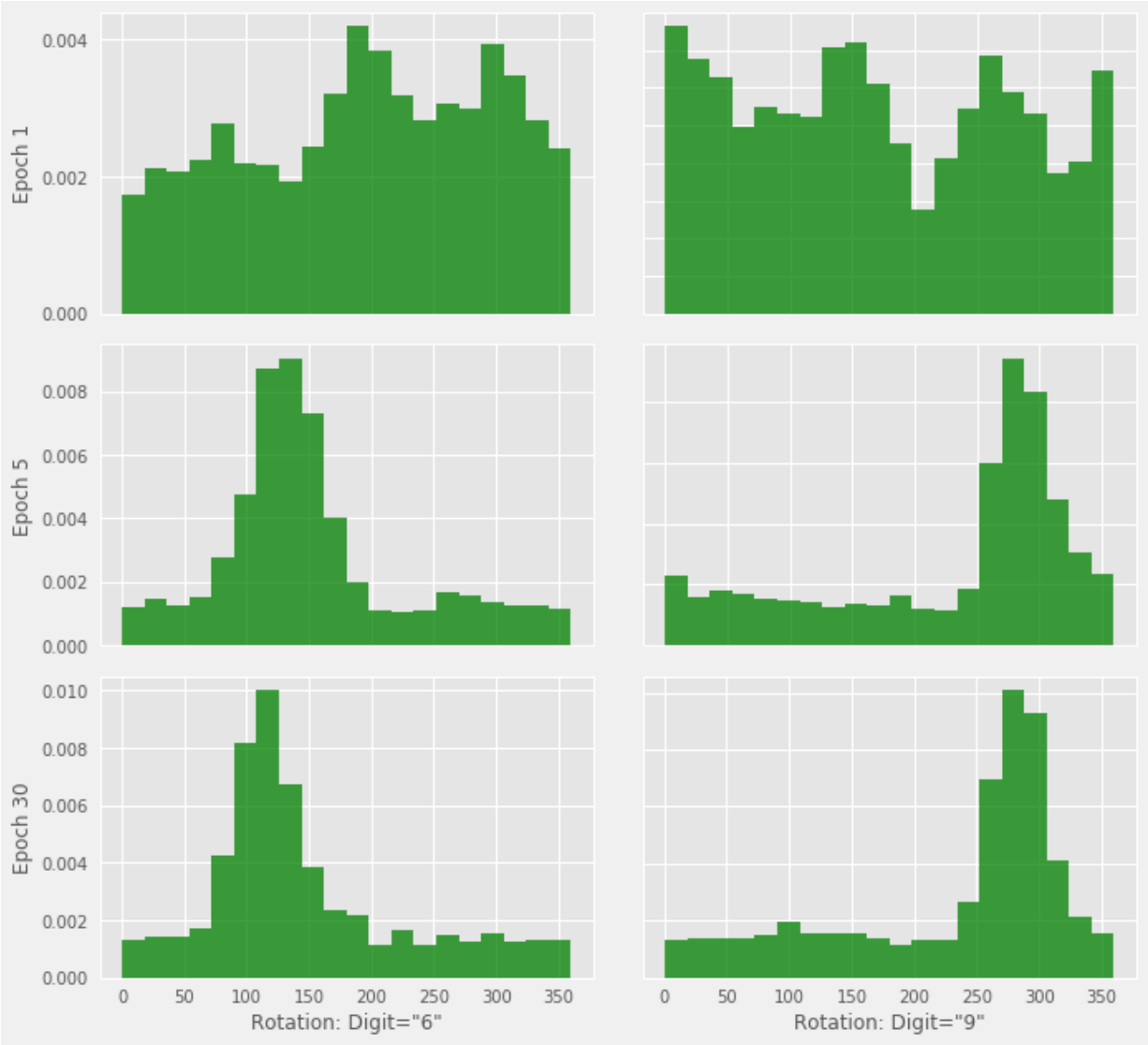


Figure 4.13: Distribution of rotations of the ”6” and ”9” digits during training of the AVAE at epochs 1, 5 and 30. As training progresses the AVAE learns to encode most digits at the same orientation, but additionally these numbers are encoded as 180° rotations of one another.

4.3 Discussion

In this chapter we proposed a method to reduce the representational redundancy present in the latent space of Variational Autoencoders through the disentanglement of orientation and shape parameters. We introduced the Affine Variational Autoencoder, a novel extension of the Variational Autoencoder including affine layers which transform an object to a canonical orientation before it is encoded, and inverting this transform after decoding. We demonstrated it can perform the unsupervised disentanglement of shape and orientation through an optimization process which finds the best orientation to encode objects by minimizing the AVAE's loss. This can be seen as approximately maximizing the likelihood of the object under our AVAE. We demonstrated this disentanglement of orientation and shape learns more compressed representations compared to a standard VAE on the 2d MNIST and 3d ModelNet datasets.

In Chapter 5 we will again look at a different form of representational redundancy, in this case the redundancy arising from a mismatch between the form of data models are pretrained on and the downstream task they are fine tuned on. We look at the domain of text classification, and find this mismatch and associated representational redundancy appears in the form of redundant sequence information being preserved throughout the network for a task where this is not required.

Chapter 5

Classformer: Efficient Transformer Architectures for Text Classification with Optimized Sequence-Length Bottlenecks via Neural Architecture Search

In this chapter we aim to reduce representational redundancy within transformer models applied to the problem of text classification. Standard approaches for text classification problems involve using a large pretrained model which is later fine tuned on a specific task. We hypothesize that this results in representational redundancy because of a mismatch between the pretraining objective and the fine tuning objective, where the pretraining objective requires a full sequence length output and the fine tuning output requires no sequence information.

We aim to reduce this inefficiency by generating an architecture based on the specific attributes of each task, the hardware used for inference, and the performance requirements. We introduce an architecture search method to design a domain specific transformer where sequence length bottlenecks are introduced, and this is implemented in a fully learnable way. An overview of this approach is shown in Figure 5.2. We also investigate a novel pretraining objective to further increase performance. We focus exclusively on tasks that can be framed as a classification problem, although this approach could be more widely applicable. We show this process produces models with a superior inference speed and

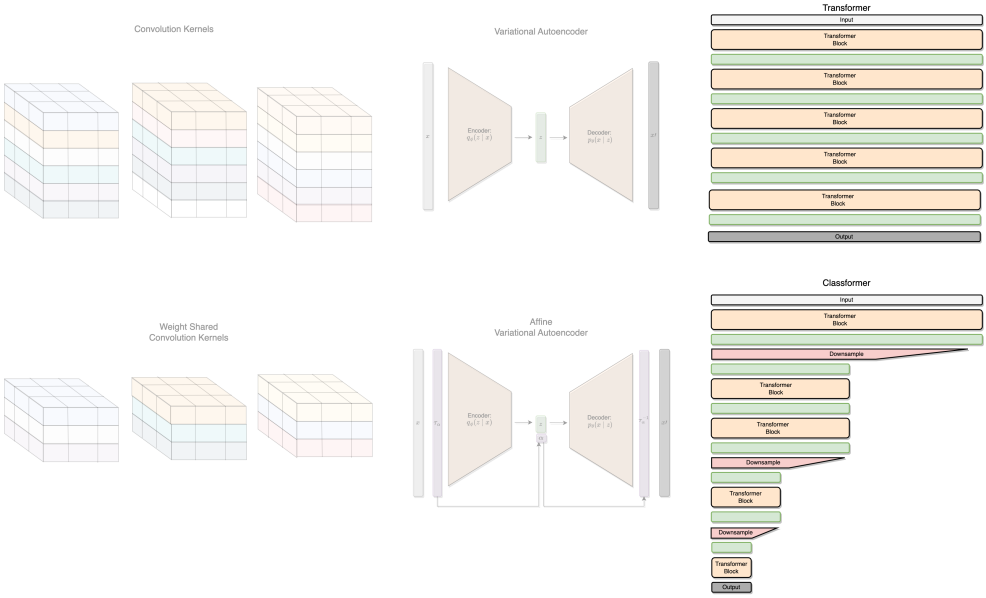


Figure 5.1: The Classformer (bottom) compared to a standard Transformer architecture (top). In the Classformer the latent size is reduced through a task specific learned downsampling, reducing the amount of computation and memory required.

accuracy tradeoff on the GLUE benchmark compared to existing efficient architectures.

5.1 Classformer Architecture

Transformers are a stack of alternating self attention and fully connected blocks which preserve the full input sequence length throughout the model. Preserving the full sequence length is necessary because transformers are designed to be used with a pretraining objective, such as masked language modelling, that requires the full sequence be preserved at the final layer of the model. But transformers are commonly fine tuned on tasks where no sequence information is required, so is preserving the full length of the sequence necessary? For example, in text classification sequence information is not needed at the final layer, so there could be redundancy in preserving this full length representation throughout the model. Previous research has shown that tokens at later layers of the network generally have higher correlation with one another[53], reinforcing this view that there may be redundancy in the later layers.

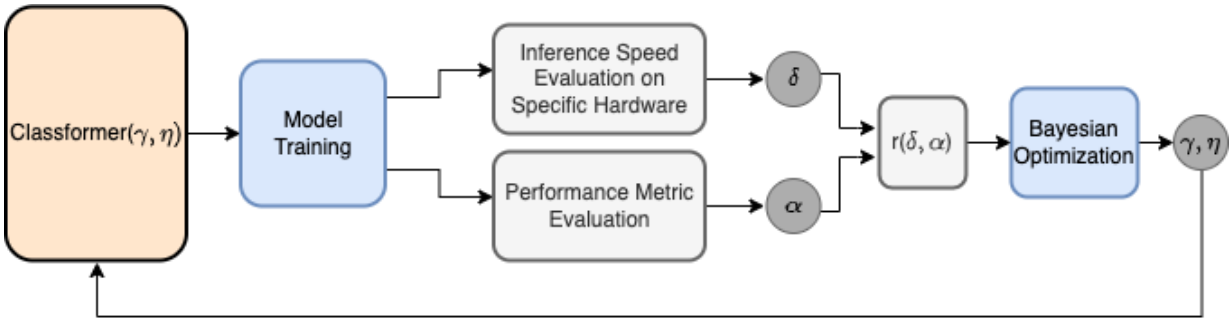


Figure 5.2: Illustration of the Classformer's neural architecture search procedure. Given performance requirements, hardware, and a dataset the procedure iteratively trains and evaluates the model, and based on its performance generates another Classformer configuration.

In this work we reduce this redundancy in transformers with sequence length bottlenecks that are optimized using neural architecture search. Specifically, we generate transformers to maximize inference speed conditional on the performance reaching some threshold, given some dataset and hardware. This is done by optimizing fully learnable sequence length bottlenecks within the transformer. To accomplish this we use a Bayesian optimization based approach to search for the best architecture through varying these bottlenecks, and show the Classformer has a favorable performance speed tradeoff compared to other models on the GLUE benchmark.

This is in contrast to most research on efficient transformer models which don't take into account performance requirements, the hardware used for inference or the characteristics of the classification task itself. Creating an efficient architecture or layer independent of the problem in question can improve performance, but will always have a limitation compared to creating a problem specific architecture, which is the approach we take in this work.

5.1.1 Sequence-Length Bottlenecks

The architecture of the transformer model is a stack of alternating fully connected and self attention layers. The self attention layer has quadratic complexity in the input length, so reducing this sequence length will be the focus of our work. Given three linear projections W^k, W^q, W^v for key, query and values, and tokens of dimension d_k, attention is defined

as:

$$Attend(\mathbf{x}) = softmax\Big(\frac{(\mathbf{x}\mathbf{W}^q)(\mathbf{x}\mathbf{W}^k)^\top}{\sqrt{d_k}}\Big)(\mathbf{x}\mathbf{W}^v) \tag{5.1}$$

There are many approaches to reduce sequence length in transformers, including hand designed rules for dropping tokens[69], dropping tokens based on attention score[53], or downsampling between layers[36]. In contrast, we will use a fully learned approach by modifying the attention mechanism, reducing the sequence length without introducing any extra operations.

We will use a modified version of attention designed to downsample the sequence length. Assuming we would like to downsample the sequence by a factor of γ, we will reduce the query by selecting every γ^{th} element. Given an input X and a connectivity pattern $S = \{S_1, \ldots, S_L\}$ where these indicate the query positions that will be preserved, we can define downsampled attention as:

$$Attend(\mathbf{X}, S) = softmax\Big(\frac{(\mathbf{x_j}\mathbf{W^q})_{\mathbf{j}\in\mathbf{S}}(\mathbf{x}\mathbf{W}^k)^\top}{\sqrt{d_k}}\Big)(\mathbf{x}\mathbf{W}^v) \tag{5.2}$$

This uses the original full length key and value matrices, so while the output length is reduced it still computes attention over the full sequence for each query. This is similar to the implementation in the Funnel Transformer[36], but instead of using mean pooling to reduce the sequence we select every γ^{th} token. We do not alter any other parameters of the base model, preserving the same number of layers, hidden dimension, linear projection dimension and embedding dimension.

For all Classformer models we use a common base architecture, Roberta[98]. By using a common architecture for the base model it allows us to load pretrained weights from the original Roberta model into the Classformer. While the reduced sequence length reduces computation and memory required in the forward pass, it involves the same number of parameters as a model without sequence length reduction.

In addition to the above mentioned method for sequence reduction, we also introduce another method where the first η tokens are kept throughout the entire depth of the network, (denoted $\eta = 8$ in the case of preserving 8 tokens) with no downsampling performed on them. In figure 5.3 we compare this method to the default approach of reducing the entire sequence. This results in a longer sequence because the reduction is not performed on the first 8 tokens, but also improves performance on some of the GLUE tasks.

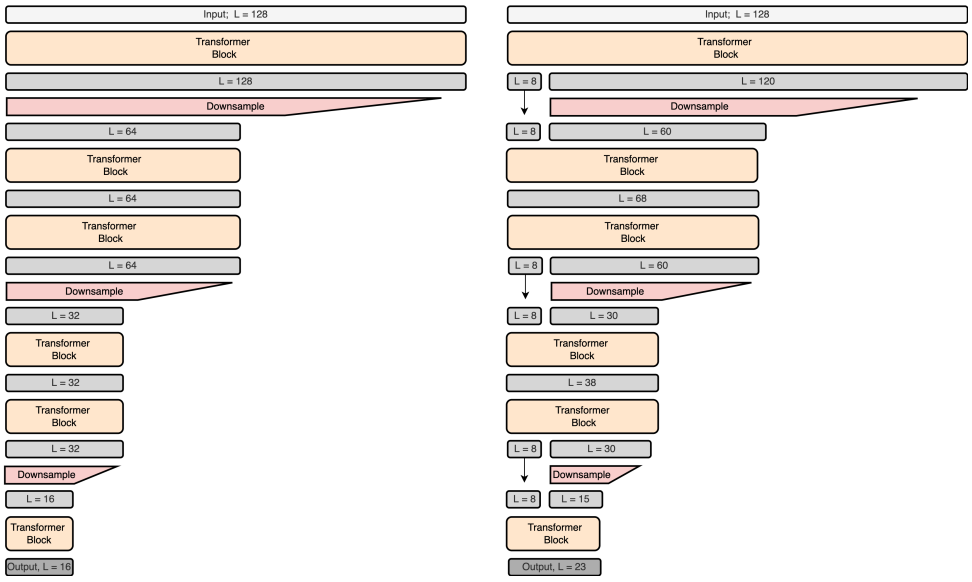


Figure 5.3: Comparison between the standard sequence reduction in Classformer (left), and Classformer with $\eta = 8$ (right). The $\eta = 8$ model preserves the first η tokens throughout the network, so during downsampling these tokens are ignored. We find this improves performance significantly, but with the downside of additional sequence length. This is a simplification of a standard transformer, with only 6 transformer blocks, 3 downsampling layers and an input length of 128

5.1.2 Architecture Search

Architecture Search Objective

Our objective is to create an optimal transformer for a given classification task, where we trade off the inference speed δ with our performance requirement, α^{req}. Given a transformer, $\mathcal{T}_{\gamma_1,\gamma_2,...,\gamma_{\mathbf{n}},\eta;\theta}$, we characterize our search space to be a set of reductions along each layer of the network, $\gamma = \gamma_1, \gamma_2, ..., \gamma_{\mathbf{n}}$ where $\mathbf{n}$ is the number of layers in the network and a number of tokens to be preserved throughout the network, η. We denote the validation dataset $\mathcal{D}_j$, and $\mathcal{L}_j(x_i, y_i; \theta, \gamma, \eta)$ a dataset specific loss on data point i from dataset j. Given required performance for this dataset, α_j^{req}, we can write this formally as:

$$\begin{aligned} \min_{\gamma=\gamma_1,\gamma_2,...,\gamma_{\mathbf{n}},\eta} \quad & \frac{1}{N}\sum_{i=1}^{N} time(\mathcal{T}_{\gamma,\eta;\theta}(x_i)) \\ \text{s.t.} \quad & \alpha_j \geq \alpha_j^{req} \\ \text{where} \quad & \alpha_j = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_j(x_i, y_i; \theta, \gamma, \eta) \end{aligned} \tag{5.3}$$

In practice our approach is to use a softer constraint during the optimization process to reflect the preference that higher performance is better when the desired performance is not reached. We choose the reward r for our optimizer as:

$$r = \begin{cases} \alpha_j & \alpha_j < \alpha_j^{req} \\ \alpha_j^{req} + \frac{1}{\delta}, & \text{otherwise} \end{cases} \tag{5.4}$$

Bayesian Optimization

We need to choose an architecture to maximize this reward by selecting hyperparameters γ, η. Based on the characteristics of our, problem, we use Gaussian Process based Bayesian Optimization. It performs well in scenarios where function evaluations are expensive and there are few hyperparameters. This is because it uses an expensive surrogate function to model the loss, and only evaluates at the most promising parts of the parameter space. This is appropriate for our problem because function evaluations are equivalent to fine tuning the model, which is extremely expensive and the total hyperparameters is quite low at 13. We use the implementation from Scikit-Opt[115]. To speed up convergence of

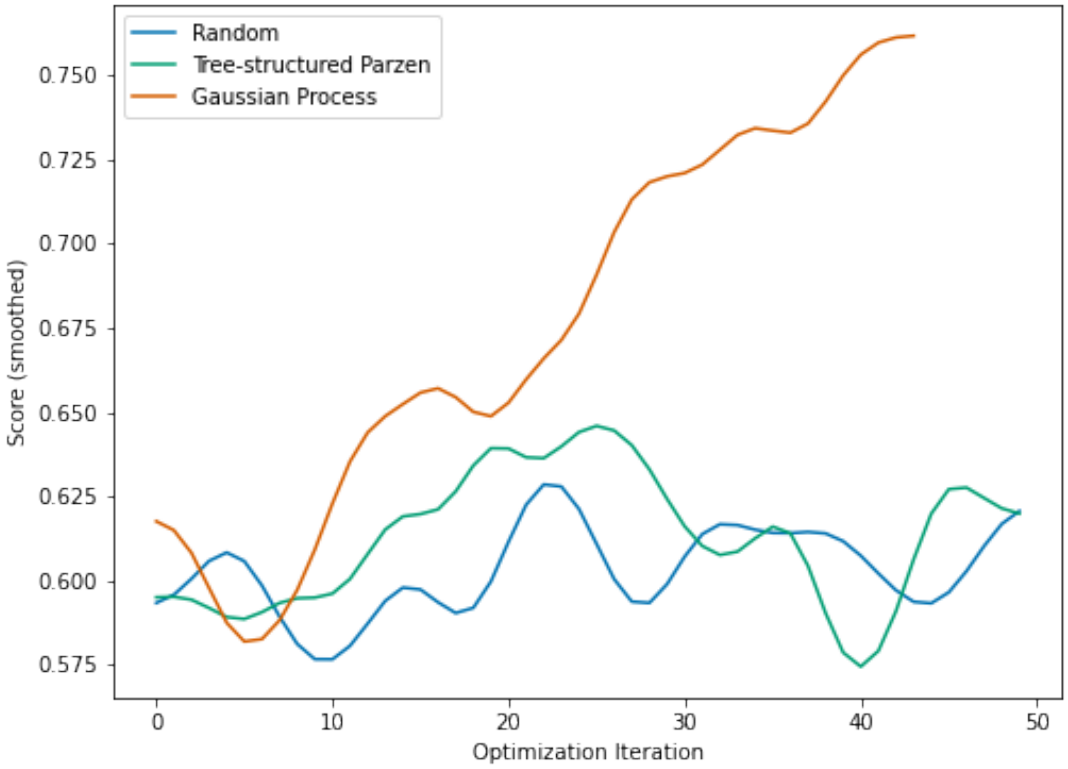


Figure 5.4: Comparison between different methods for optimizing the Classformer's accuracy on the RTE dataset. On the x axis we see the number of function evaluations, in this case the number of architectures fine tuned. The y axis shows the score, which is the reward defined in equation 5.4. We clearly see the superiority of Gaussian Process based optimization over the Tree-structured Parzen estimator and Random search optimizers as it is much more efficient in improving performance for a given number of function evaluations.

the algorithm we can select an initial population to be evaluated before the optimization is ran. We use a set of 8 random configurations, but also include 10 hard coded ones. We select these based on our intuition of what architectures are likely to perform well, in addition to selecting a variety to better span the search space. In this set we include the default Roberta architecture, and 9 others representing a range of architectures with sequence reduction at progressively earlier layers.

In figure 5.4, we compare this Gaussian Process based optimization to random search and the Tree-structured Parzen estimator[12] implemented in Hyperopt. For this experiment we remove the hand designed initial population used to speed up converge. Over 50 iterations we see it clearly outperforms both alternatives, so we use this method for all architecture search experiments.

5.1.3 Fine Tuning

Performance on text classification tasks is heavily dependent on the fine tuning procedure, especially when fine tuning on datasets with a small number of examples[86]. Performance is also dependent on the weight initialization of the classifier layer, as well as the order the data is fed to the model during training[44]. Multiple related papers have attempted to overcome these difficulties[191] [110], and to reduce the variance in results during fine tuning. The most useful changes to the standard fine tuning procedure were to train the model for a longer number of epochs as well as to use the debiased version of the Adam optimizer.

There is an inherent conflict between the pre-training and fine-tuning tasks. Previous work has shown that earlier features in earlier layers of the network are the most general, while those in later layers are specific to a particular task[184]. Recent work has shown this is also the case when pretraining and fine tuning BERT[191]. They found that re-initialization of the last 1-6 layers in BERT large before fine tuning improves performance. On the other hand this is not the case with the smaller base size models we use for our work, so we do not consider this finding to be relevant.

For our fine tuning procedure we follow the best practices for reducing variance and use the debiased Adam version along with training for 10 epochs with early stopping. We also preserve the same data order and weight initialization by setting the seed in PyTorch, although some randomness is still injected into the training process through non determinism in some CUDA functions[196].

A learning rate warmup of 10% of the total steps is used, along with a decay to 0 over the 10 epochs of training. We use these hyperparameters for all tasks because they have

been shown to give good performance in previous work and because fine tuning tasks can be sensitive to changes in hyperparameters[86, 44, 191, 110]. There are other more complex methods to improve fine tuning performance such as mixout[86], where the parameters of two models are stochastically mixed, but in our work we focus on the more standard and less computationally expensive form of fine tuning. FP16 is used for all models, with the exception of Squeezebert which was faster using FP32. All results were obtained using Nvidia V100 and A6000 GPUs.

We verify the effectiveness of this fine tuning procedure in table 5.1. In the first row we see we can reproduce the results of the Roberta paper well, so our fine tuning procedure is generally effective. In the last line we demonstrate the importance of pre-training, showing there is a massive drop in performance when Roberta is fine tuned using randomly initialized weights instead of the weights from the pre-training objective. For all models we ensure they are initialized with pretrained weights based on a standard implementation found on HuggingFace[173].

Model	MNLI	QNLI	QQP	RTE	SST	MRPC	CoLA	STS	AVG
Roberta paper	87.6	92.8	91.9	78.7	94.8	90.2	63.6	91.2	86.4
Roberta pretrained	87.0	92.9	88.8	83.4	94.8	91.8	63.0	90.9	86.6
Roberta; random	64.5	60.4	72.0	52.7	79.8	83.4	60.4	45.8	64.9

Table 5.1: Comparison of the performance of different Roberta model configurations on various GLUE tasks. Performance is evaluated using a number of benchmarks, including MNLI, QNLI, QQP, RTE, SST, MRPC, CoLA, STS, with an average score (AVG) calculated for overall performance assessment.

5.2 Experiments

We compare our search method for finding the optimal architecture to a variety of alternative transformer models on the GLUE dataset. We explicitly include efficient transformer models in this comparison to ensure we are comparing to other models designed to have fast inference speed. In addition, we individually investigate our hypothesis of generating an architecture conditional on dataset, hardware and accuracy requirements and also look at a synthetic dataset to evaluate inference speed on different sequence lengths.

5.2.1 GLUE Dataset

We aim to evaluate how the Classformer performs in terms of inference speed and accuracy compared to a variety of efficient transformer models. To do this we use a variety of tasks in the General Language Understanding Evaluation (GLUE) benchmark. GLUE is a set of 9 tasks designed to evaluate natural language understanding, including tasks in sentiment analysis, English grammar, question answering and sentence similarity[164]. We used this benchmark because the wide variety of tasks will better inform how the Classformer will perform in real world applications. Also, because it is widely used we can compare performance to other popular models. Following previous work[42], we exclude the WNLI dataset from our analysis because of its adversarial validation set construction. More information on the individual datasets is found in the table 5.2.

1. **CoLA** - Corpus of Linguistic Acceptability[168] is a classification task to categorize sentences as grammatically correct or incorrect.

2. **SST-2** - The Stanford Sentiment Treebank[150] is a classification task consisting of sentences from movie reviews where the objective is to predict sentiment.

3. **MRPC** - The Microsoft Research Paraphrase Corpus [45] is composed of sentence pairs with the binary objective of classifying if the pair are semantically equivalent.

4. **QQP** - Quora Question Pairs[2] is also a binary task of classifying if sentences are semantically equivalent, in this case questions from Quora.

5. **STS-B** - The Semantic Textual Similarity Benchmark[20] is another sentence similarity benchmark but with a 5 category classification objective representing varying degrees of similarity.

6. **MNLI** - The Multi-Genre Natural Language Inference Corpus[171] is a textual entailment prediction task where given a premise and hypothesis sentence, the task is to predict whether the premise entails the hypothesis.

7. **QNLI** - The Stanford Question Answering Dataset[123] is a question answering dataset framed as a binary classification problem where there are two sentences and it must be determined if one contains the answer to the other.

8. **RTE** - Recognizing Textual Entailment is a binary classification textual entailment prediction task consisting of a combination of the data from RTE1[35], RTE2[56], RTE3[49], and RTE5[11].

9. **WNLI** - The objective of the Winograd Schema Challenge[88] is to determine the referent of a pronoun but it is framed in terms of a binary classification problem of textual entailment. The ambiguous pronoun is replaced with a correct or incorrect pronoun, and the task is to predict if the original sentence entails this modified sentence. There is an adversarial split between train and validation split because if two examples contain the same sentence they will usually have opposite labels, but because the train and validation set share examples it is difficult to prevent overfitting to the sentences themselves. This means that classifying as the majority class tends to be most successful[1], so we exclude it from our experiments.

Dataset	**Train Examples**	**Validation Examples**
COLA	8,551	1,043
SST2	67,349	872
MRPC	3,668	408
QQP	363,846	40,430
STSB	5,749	1,500
MNLI	392,702	9,815
QNLI	104,743	5,463
RTE	2,490	277

Table 5.2: The number of training and validation examples available for each dataset in the GLUE benchmark.

5.2.2 Accuracy Dependent Architecture Optimization

Our search method optimizes an architecture to have the lowest inference speed for a given desired accuracy. In this section we confirm our hypothesis that this optimization is possible through the addition of sequence reductions inside the model, and investigate how the architecture discovered by our search method changes as the desired accuracy changes.

To show that lower accuracy requirements can enable a greater level of downsampling and faster inference speed, we experiment with varying the desired accuracy in the Classformer's optimization process. On the RTE dataset we vary the desired accuracy between 60% and 85% of the full model's accuracy and investigate how this affects the inference speed and architecture of the Classformer. Looking at figure 5.5 we see that changing the desired accuracy changes the optimal architecture discovered, with lower desired accuracy

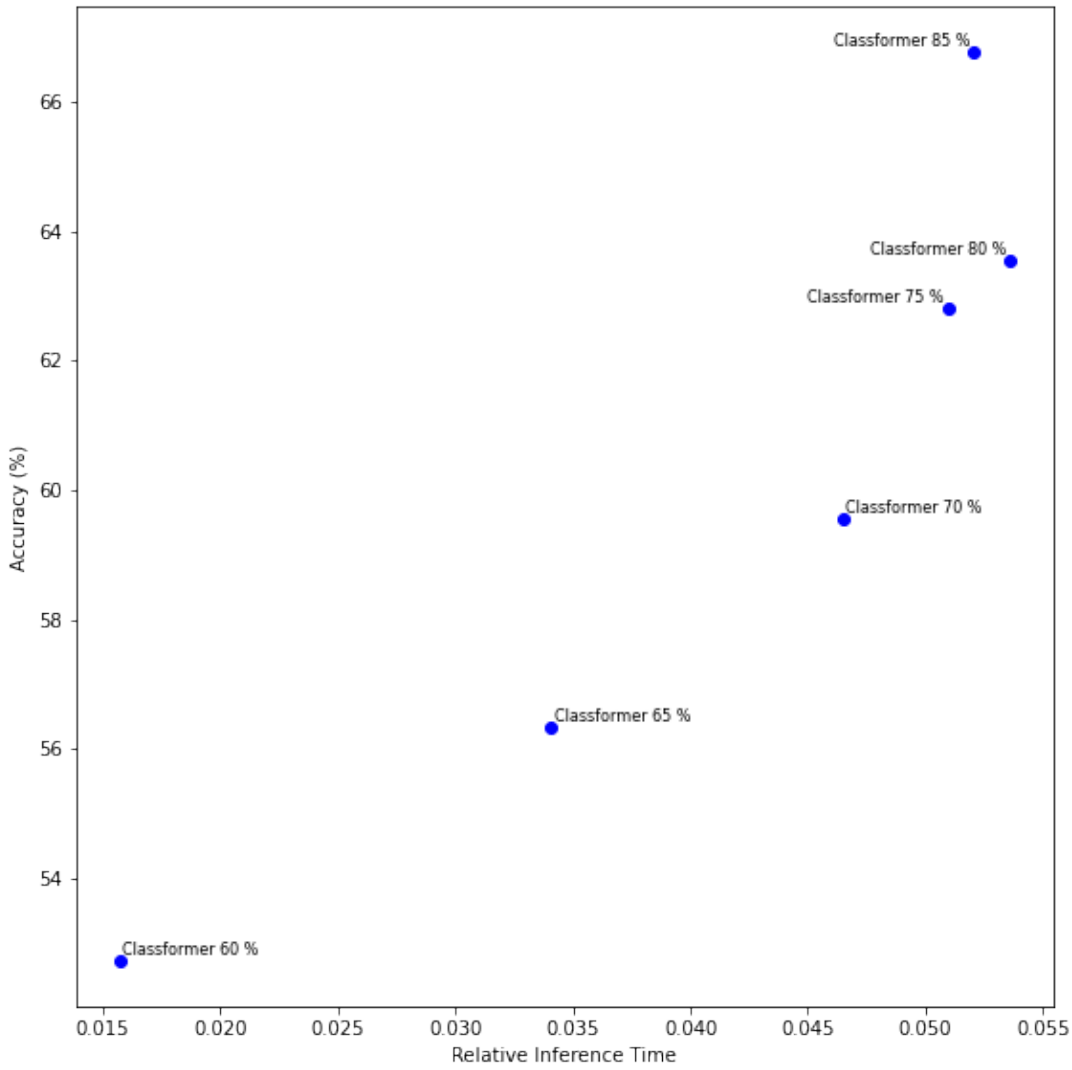


Figure 5.5: Demonstrating the relationship between required accuracy and inference speed. With a lower required accuracy simpler architectures can be used, allowing for faster inference. For the labels, Classformer $x\%$ refers to a Classformer with a required accuracy of $x\%$ of the full model.

enabling greater inference speedup. This indicates conditioning the creation of the Classformer on the desired accuracy is useful to enable the maximum inference speedup. To more clearly visualize how the architectures differ as accuracy requirements change, Table 5.3 compares the sequence length reductions across all layers using an example sequence of length 512.

Layer	60 %	65 %	70 %	75 %	80 %
0	512	512	512	512	512
1	128	256	512	512	512
2	32	256	512	512	512
3	8	256	512	512	512
4	4	128	512	512	512
5	2	128	256	512	512
6	1	64	256	512	512
7	1	64	256	512	512
8	1	64	256	512	512
9	1	64	256	512	512
10	1	64	256	512	512
11	1	64	256	256	512

Table 5.3: Layer sizes in intermediate layers of the Classformer after being optimized for a given accuracy on the RTE dataset. Lower performance requirements enable greater downsampling throughout the network.

5.2.3 Dataset Dependent Architecture Optimization

Our proposed method assumes performance and optimal architecture are dataset dependent, and in this section we verify this assumption. In contrast, if all datasets exhibited the same performance properties this optimization would be a waste of resources and the optimal architecture could be computed independently of the dataset, optimizing only based on performance requirements and hardware.

We compare how downsampling affects performance across datasets in the GLUE benchmark for a variety of levels of downsampling. For consistency, here we remove the optimization procedure and instead use a fixed downsampling schedule so we can compare performance across datasets using the exact same architecture. We use a fixed downsampling schedule where the sequence length of the last n layers of the network are reduced

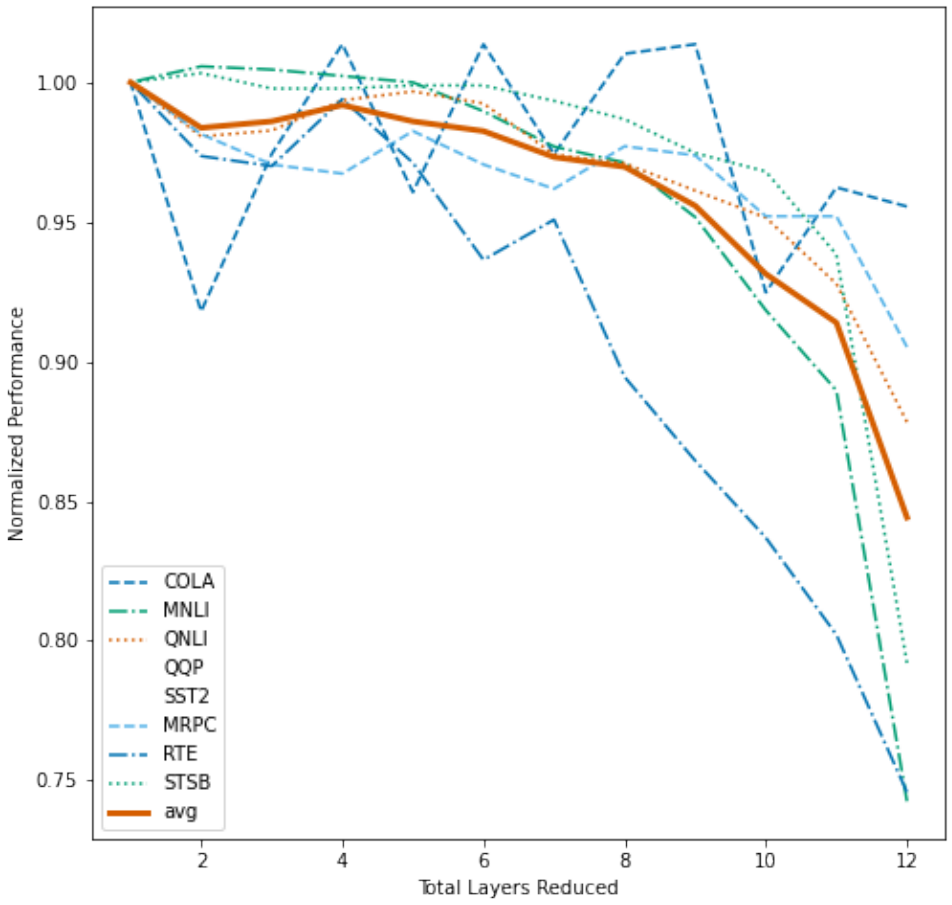


Figure 5.6: Effect of sequence length reduction on GLUE performance. The x-axis indicates the number of layers removed from the original model. The y-axis indicates the percentage of the original model's performance. The reduction is done in a systematic way, where the sequence length in the last n layers of the network are reduced by a factor of 4x. Across all datasets the performance drops with more sequence length reduction, but for some such as RTE this drop off happens quickly, while for others such as QNLI it happens more gradually.

by a factor of 4x. Downsampling more layers results in a model with faster inference, but also with more degraded performance.

Looking at figure 5.6, we see how the performance of the GLUE datasets is affected by this increased downsampling, with the total number of downsampled layers on the x axis, and the normalized performance on the y axis. On the far left we see the model with no downsampling, and we normalize the performance of all other models to this. While all models exhibit the trend of decreasing performance with increased downsampling, performance drop varies widely across datasets for a given amount of downsampling, confirming our hypothesis.

We can focus directly on the performance drop across datasets for a given amount of downsampling, to see a more clear view of the difference between datasets, as shown in figure 5.7. Here we see more clearly how downsampling harms the performance of RTE and MNLI, while QNLI and MRPC perform well even under high downsampling. This indicates the degradation of performance differs across the GLUE datasets, confirming the usefulness of a dataset dependent architecture search.

5.2.4 Hardware Dependent Architecture Optimization

We investigate how the optimal architecture changes with different hardware. As with conditioning the optimal architecture on the dataset, we should verify that different hardware requirements result in different optimal architectures being discovered, otherwise hardware would not be a relevant variable, and the Classformer architecture could be computed independently of any particular hardware used for inference. For this we will compare optimal architecture for CPU and GPU, comparing an AMD Ryzen Threadripper 3960X 24-Core CPU (limited to 8 cores) with 256GB of memory to an NVIDIA RTX A6000 GPU with 48GB of memory. Although we compute inference speed using the CPU, the training is still ran on the GPU, as training speed is not a concern for this experiment.

In figure 5.8, we run this optimization for the CPU and GPU on the RTE dataset, and see that different architectures are discovered based on the hardware used for inference. This process finds the model with greatest inference speed conditional on the performance requirement being met (in this case 70%), with the left model corresponding to GPU and the right to CPU. The architecture designed for CPU seems to optimize for lowering overall FLOPS more than the one for GPU, possibly because the GPU is able to run more computations in parallel, but further investigation is needed to know exactly what makes a model perform well on different hardware.

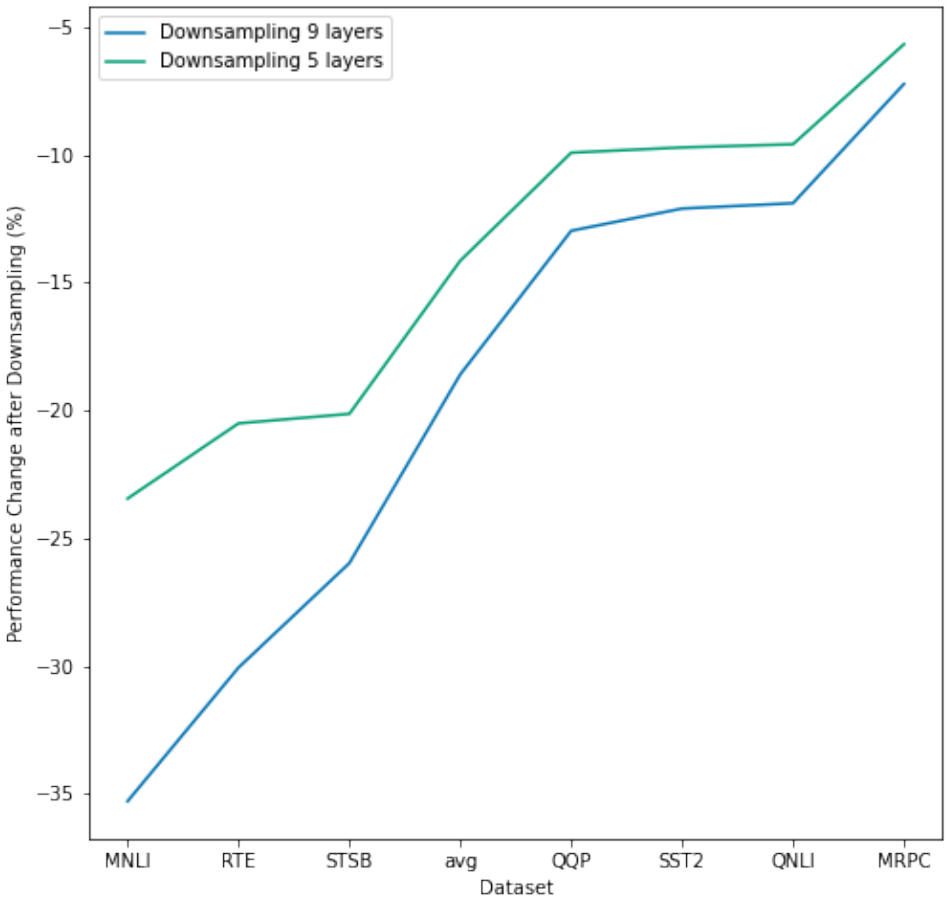


Figure 5.7: Directly comparing how performance varies between datasets depending on the amount of sequence reduction used in the Classformer. Datasets on the left, such as MNLI or RTE face a large performance drop under sequence reduction, while for datasets such as QNLI or MRPC the drop is much smaller. This is consistent across different levels of sequence reduction, shown in the graph with lines corresponding to 5 and 9 layers with a reduction of 4x in both cases.

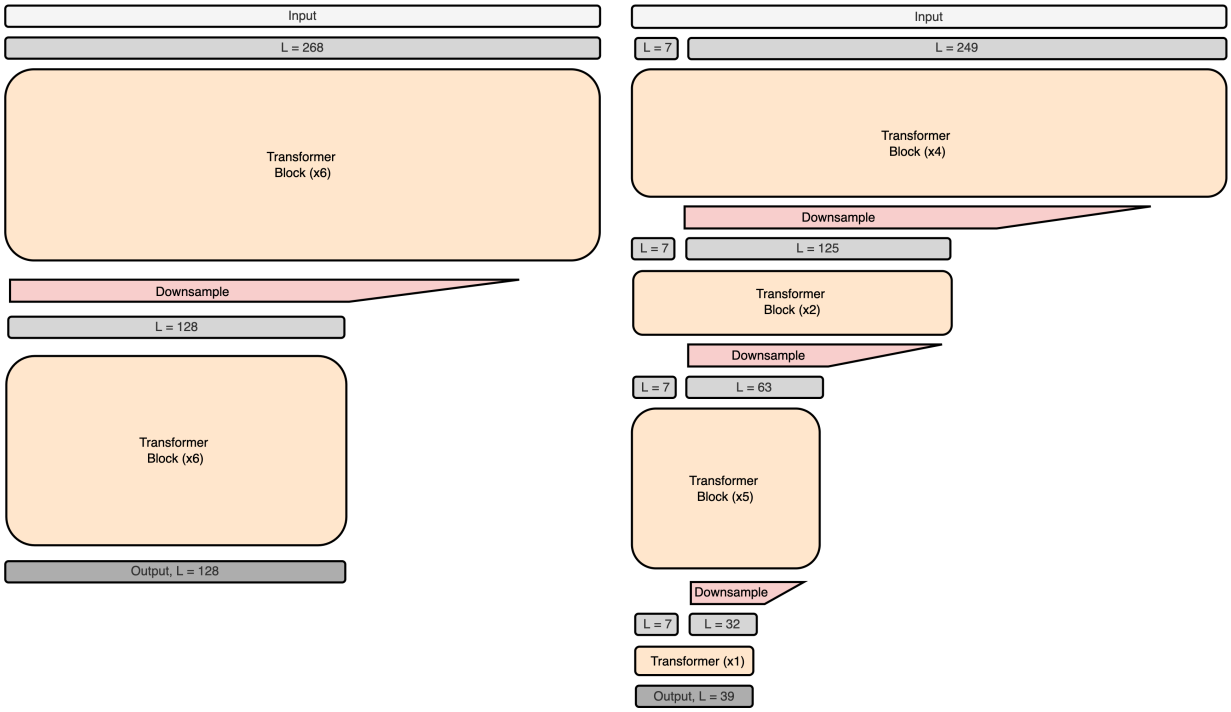


Figure 5.8: Comparing best architectures found on RTE dataset when running inference on GPU(left) and CPU(right). The Classformer's optimization process is done for specific hardware, so the architecture is created based on the hardware's specific constraints.

5.2.5 Classformer Performance on GLUE

We aim to compare the performance of the Classformer to standard models on the GLUE benchmark in terms of inference speed and accuracy to determine if the Classformer is a viable alternative to standard models. For this work we limit our exploration to a single hardware type, CPU, although this could be applied to other hardware types as well.

For all experiments we follow the evaluation method outlined in the BERT paper[42]. We report accuracy for all tasks with the exception of Spearman correlation for STSB, Matthew's correlation for COLA and F1 scores for QQP and MRPC. The matched MNLI dataset is used and we evaluate on development set. All models are fine tuned using pretrained models, and the initialization for the Classformer is from the pre-trained Roberta model. All are fine tuned using the method described in section 5.1.3 in order to reduce variance. In addition, for smaller datasets like COLA, MRPC, STSB and RTE we do fine tuning 3 times and report the average performance.

We compare model inference using CPU and batch size of 1. This is representative of many real world applications because higher batch sizes are only possible in online applications where there is high enough demand that multiple requests can be batched together in real time. CPU inference is also relevant in practice because while GPU inference is faster than CPU, it is also much more expensive, with the greatest benefits at higher batch sizes which are not relevant for many real time applications.

Model	**CoLA**	**MNLI**	**QNLI**	**QQP**	**SST2**	**MRPC**	**RTE**	**STSB**	**AVG**
BERT	52.1	71.2	88.9	85.8	66.4	93.5	84.6	90.5	79.1
Mobilebert	51.1	70.5	88.8	84.8	70.4	92.6	84.3	91.6	79.3
Distilbert	51.3	88.5	87.5	86.9	59.9	91.3	82.2	89.2	79.6
Classformer 95%	57.3	84.7	90.4	89.7	73.7	93.0	85.5	88.7	82.9
Squeezebert	53.7	90.9	92.0	90.3	80.9	92.2	82.5	89.2	84.0
Classformer 99%	58.1	87.3	91.0	89.8	73.7	94.4	85.4	92.2	84.0
Funnel Small	62.8	91.3	89.2	89.2	74.3	93.6	86.0	91.6	84.8
Funnel Medium	63.9	91.4	90.2	91.0	77.6	94.2	87.0	92.2	85.9
Roberta	63.6	91.9	90.2	91.2	78.7	94.8	87.6	92.8	86.3

Table 5.4: Performance of Classformer and other models on all GLUE datasets. Values are based on our experimental results for the Classformer model, and the performance of the other models is from the original papers.

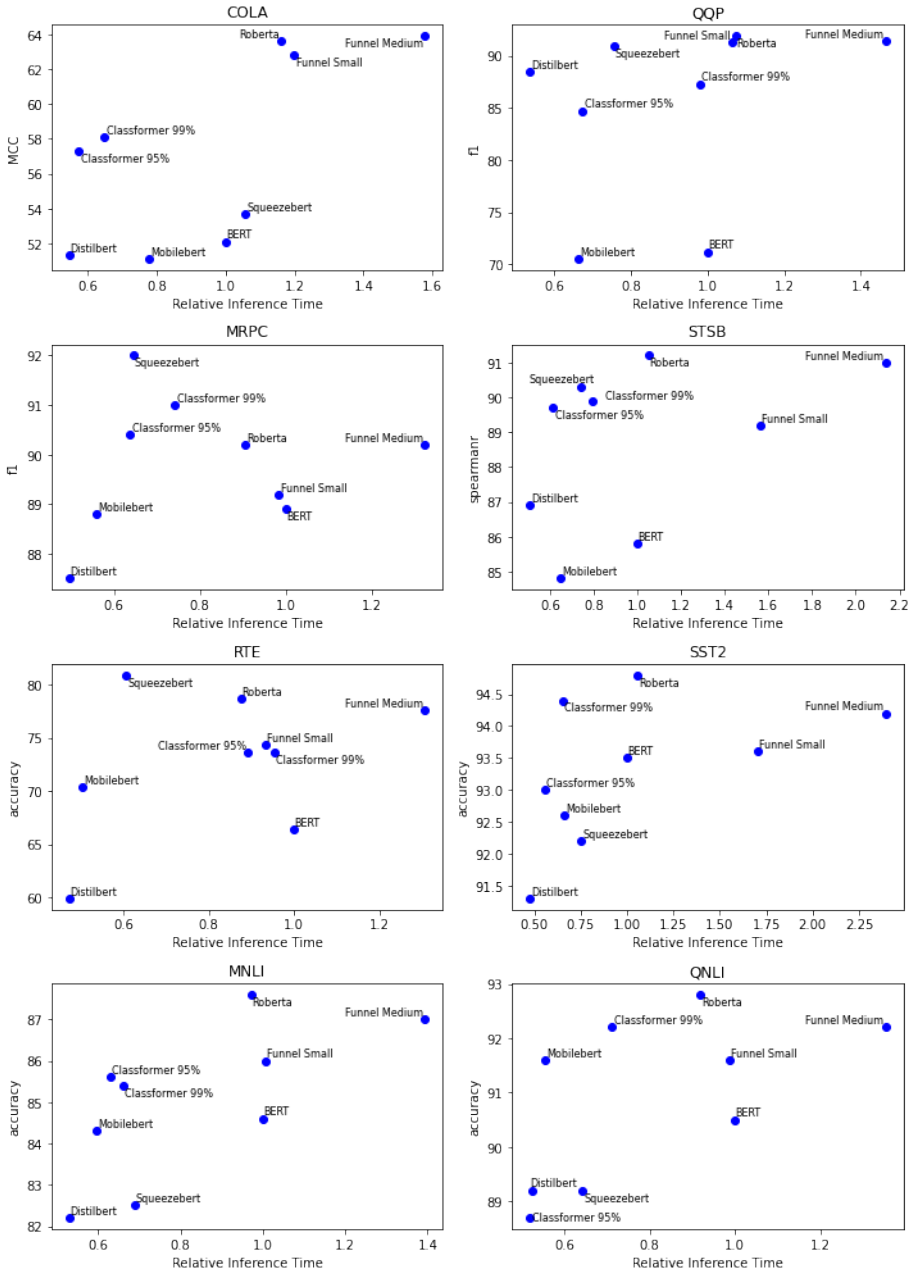


Figure 5.9: Comparing inference speed and accuracy across all GLUE datasets for our architecture search based Classformer to standard models. Our models are denoted Classformer α, where α represents the desired minimum performance compared to a baseline model, in this case Roberta. We compare stated results in the papers to our empirical results.

Comparing the Classformer to Other Efficient Models

The results of designing the Classformer conditional on different GLUE datasets are visualized on figure 5.9, where we investigate two variants of the Classformer, using an objective of 99% and 95% of the full model's performance. For this set of experiments we are comparing the Classformer's performance to the performance stated in the other papers, instead of rerunning the experiments ourselves.

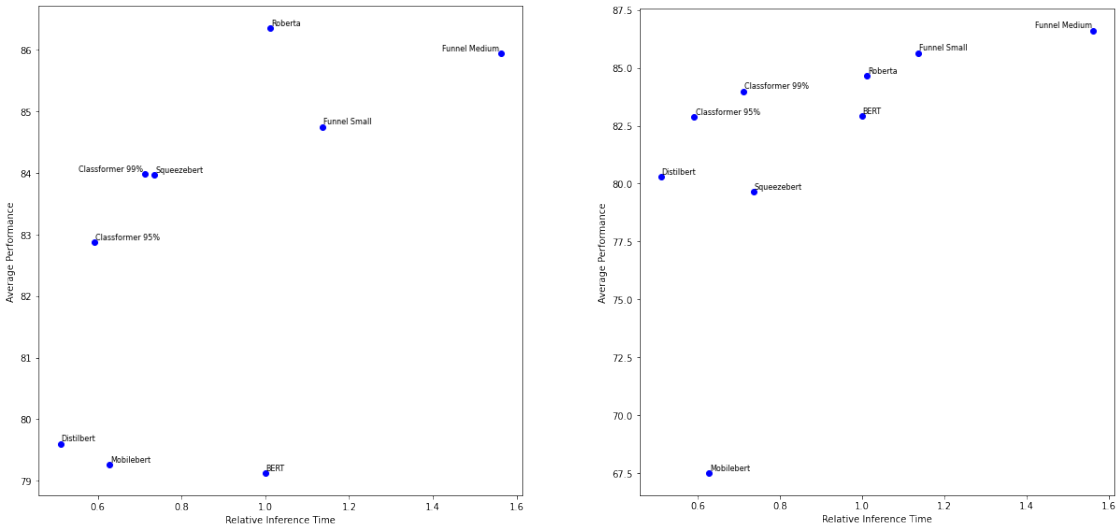
We see that for most datasets the Classformer lies at a point on the Pareto frontier, meaning it represents some optimal trade off between accuracy and inference speed. The performance differs quite significantly across datasets, for example on COLA, STSB, SST2, and MNLI our method clearly lies at a point with an optimal inference time / performance trade off. On MRPC the Classformer is slightly inside the Pareto frontier, improved on by Squeezebert, similarly to QNLI where is is improved on by MobileBERT. On QQP and RTE the Classformer is beaten by both Distilbert and Squeezebert. Overall performance of the Classformer is superior, it being on the Pareto frontier for 4/8 datasets, and the next best models, Squeezebert and Distilbert, only succeeding on 2/8.

To better visualize the performance, we also look at the average performance and inference speed of the Classformer across all GLUE datasets, shown in figure 5.10(a). We see that the Classformer is able to achieve a good average performance, with both the 99% and 95% models lying on a point on the Pareto frontier for inference speed performance tradeoff. Compared to the Classformer, Bert, MobileBert and Distilbert are harmed by their overall low performance. While they have good performance relative to the number of parameters in the model, Funnel style models tend to have excessively slow inference speed. Detailed results for all models can be found in table 5.4.

Empirical Performance of Other Efficient Models

We also investigate training these other models instead of simply citing the performance from the papers. We compare the Classformer to the empirical performance we find after finetuning each of the efficient transformer models individually. We do this because pretrained models that were fine tuned on the GLUE datasets were not available, and there was possible inconsistency in the exact evaluation metric used for some datasets, such as whether Mathew's correlation or accuracy is used on the COLA dataset.

As expected based on the above results comparing the Classformer to the stated performance from papers, the Classformer also lies on a point on the Pareto frontier of inference speed and accuracy tradeoff here, which can be seen in Figure 5.11. We found we could



(a) Comparing Inference Speed and Accuracy on all GLUE Datasets

(b) Comparing Empirical Inference Speed and Accuracy on all GLUE Datasets

Figure 5.10: Comparing the average inference speed and accuracy across all GLUE datasets for the Classformer to standard models. We compare the results from papers (left) to the performance we find empirically (right). In both cases the Classformer lies at a point on the Pareto frontier of inference speed accuracy tradeoff.

generally reproduce the results quite well, although this was not the case with one model in particular, MobileBERT. We do not doubt the results of these models, but in practice efficient models with unusual architectures can be difficult to train properly. Detailed results are shown in Table 5.5.

Model	**CoLA**	**MNLI**	**QNLI**	**QQP**	**SST2**	**MRPC**	**RTE**	**STSB**	**AVG**
BERT	54.8	84.3	90.8	87.7	91.7	88.4	75.6	90.0	82.9
Mobilebert	16.8	81.4	90.8	85.7	90.3	29.5	57.8	87.7	67.50
Distilbert	47.7	81.3	88.0	86.2	90.0	89.9	70.6	88.7	80.30
Squeezebert	39.3	80.8	89.3	86.5	89.6	89.9	71.8	90.0	79.65
Roberta	52.7	87.6	91.4	88.2	94.2	89.8	82.4	90.9	84.7
Funnel Small	62.6	87.7	91.2	88.0	94.3	90.7	80.1	90.6	85.65
Funnel Medium	64.8	88.1	93.3	89.0	94.6	90.1	81.7	91.3	86.61

Table 5.5: Empirical performance of models on the GLUE dataset

Constrained Architecture Search

Initial experiments used a more constrained search space where instead of using flexible amounts of downsampling across all layers, we used a fixed downsampling schedule with a single parameter to choose. Downsampling layers were spaced evenly throughout the network, enabling a simple optimization procedure where we only had to choose the first layer where 2x downsampling was performed. Two other downsampling layers of 2x were evenly spaced between this layer and the output layer. This meant that the search space could be described by a single number, and because increasing this single parameter would have a monotonic effect on performance and inference speed, binary search could be used. The amount of downsampling could be either reduced or or increased at each step depending on if performance requirements α are satisfied until the minimal amount of downsampling that still satisfied α is found.

The results of the architecture search method are visualized in figure 5.12 where we see that for most datasets the constrained architecture search method shows good performance or even lies at a point on the Pareto frontier. Yet for many datasets the performance of Distilbert is superior. This is because the architectures were excessively constrained, and not flexible enough to adapt to requirements of different datasets. This limitation was overcome by widening the search space and the addition of our Bayesian optimization based approach to search over it.

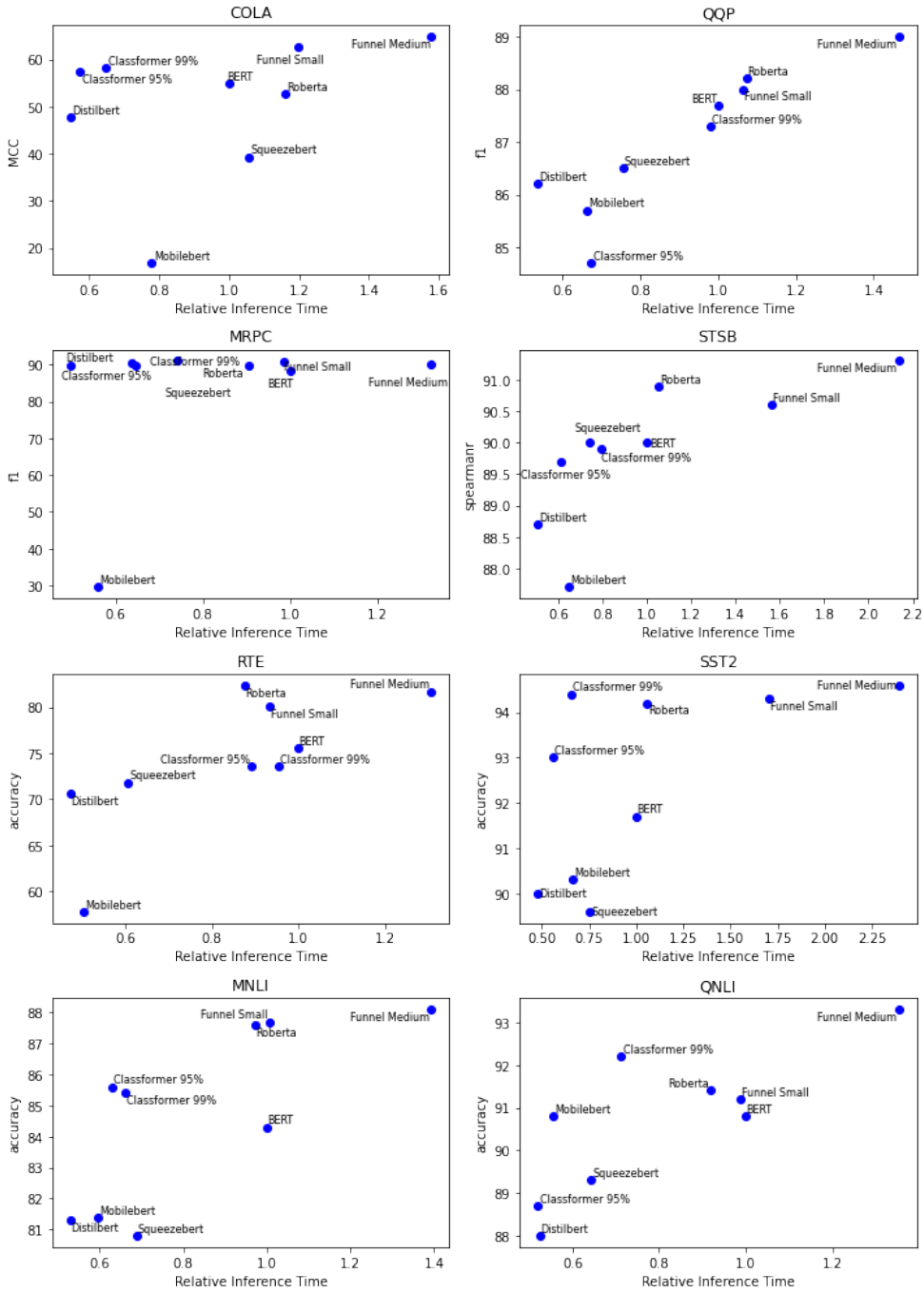


Figure 5.11: Comparing the performance of standard models achieved empirically through fine tuning with the Classformer on the GLUE datasets. We were unable to fine tune Mobilebert successfully, but it is included for completeness.

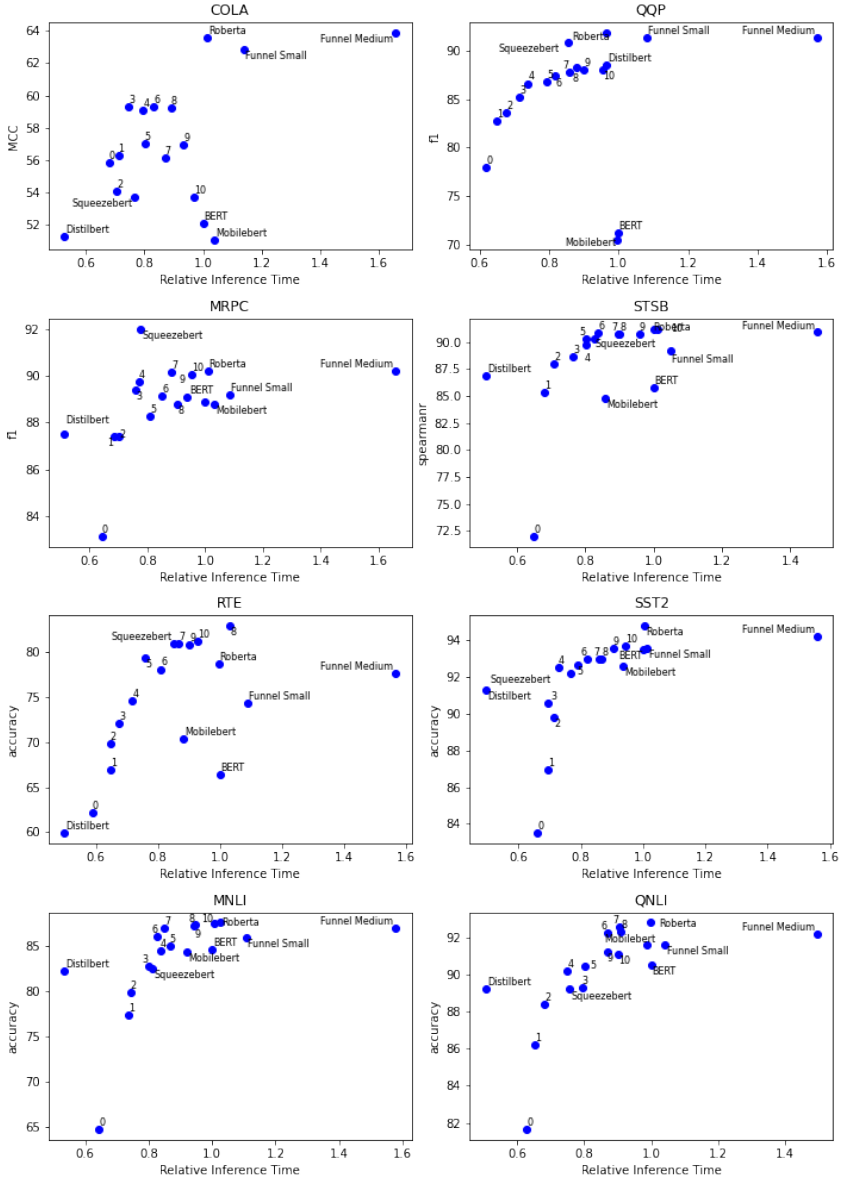


Figure 5.12: Full details of inference speed vs. accuracy tradeoff for all baseline models and all architectures with fixed sequence reduction schedule. Classformer architectures are labelled with an integer representing the first layer that is downsampled. This experiment was ran on different hardware than the experiments with full architecture search so inference speeds are not consistent.

Inference Speed vs. Sequence Length

To better understand how sequence length affects inference speed we compare the Classformer to other models when varying inputs length. In figure 5.13 we see how efficient models improve on the full size BERT and Roberta models, with the height indicating latency on CPU, and and the x axis representing the length of sequence in terms of tokens. We note that there will be slight differences in lengths for the same text due to differences in tokenizers between models. The slope of the line can be interpreted as how the model scales to longer sequence lengths. We include a comparision to the Classformer with reductions of 2x at layers 3 and 8, and retaining 8 tokens throughout the network ($\gamma_3 = 2, \gamma_8 = 2, \eta = 8$). We see that the main benefit of the Classformer is that it scales more favourably to longer sequences compared to other models. We can roughly divide these models into three categories:

1. **Standard Models** - BERT[42] and Roberta[98] both are relatively slow and scale poorly with increasing sequence lengths because of the full attention mechanism and the preservation of the full sequence length throughout the network.

2. **Smaller Standard Models** - SqueezeBERT[70] and DistilBERT[136] both have overall faster inference time than the standard BERT models. Their performance is most impressive at shorter sequence lengths, while scaling to longer sequences is less impressive because they retain the same sequence lengths throughout the network.

3. **Sequence Reduction Models** - MobileBERT[155] and the Classformer both reduce the sequence length throughout the network, which allows them to scale more favourably to longer sequences. This means that these models tend to outperform their larger counterparts at longer sequence lengths. The funnel models[36] also belong to this category but because of space constraints and their slow inference speed we do not include them in this comparison.

Predicting Optimal Model For Architecture Search

While our approach to architecture optimization is effective, it is still computationally expensive because of the requirement to retrain the model multiple times. We investigate whether we can predict the optimal model for a given dataset without performing the architecture search. We use the number of training examples in the dataset and average

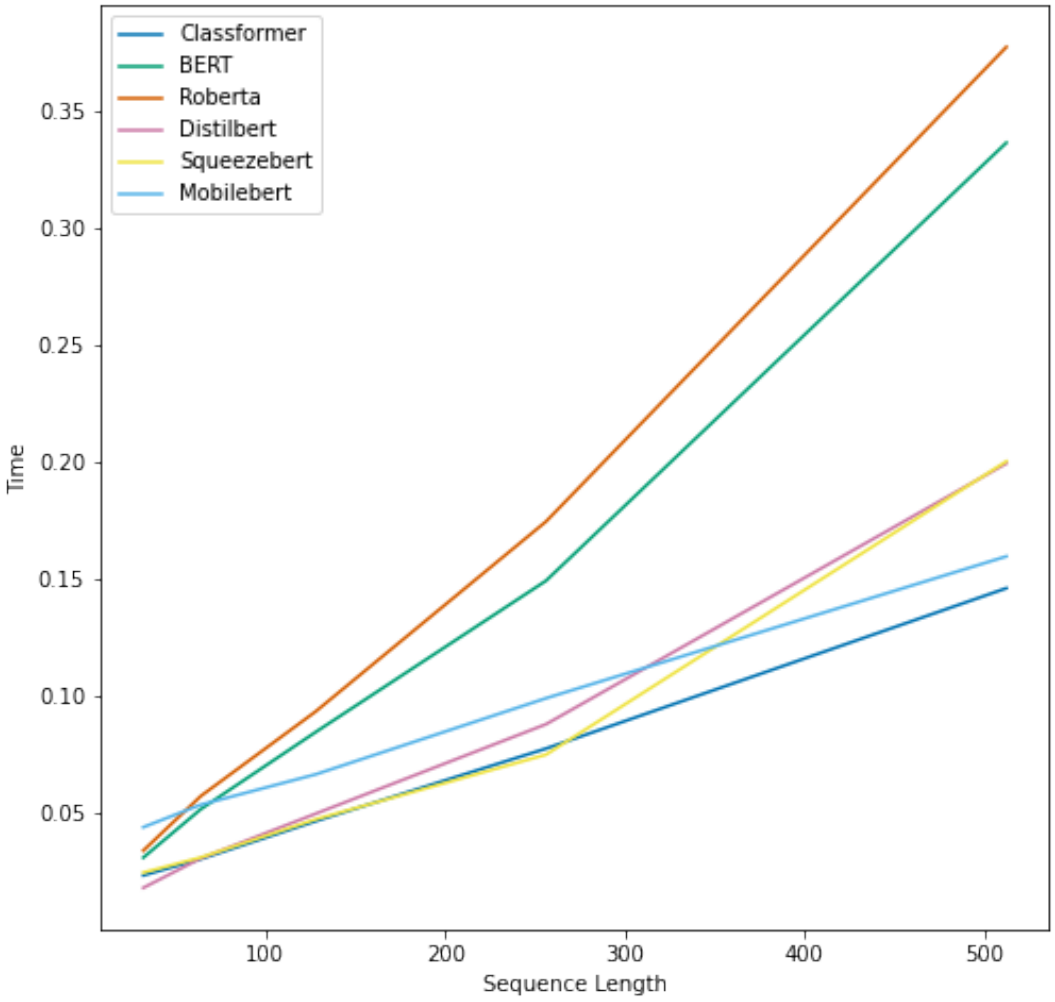


Figure 5.13: Sequence length vs. Inference Speed for a variety of models. We see that the Classformer has favorable scaling properties with increasing sequence length, but other models such as Distilbert and Squeezebert perform well with very short sequences.

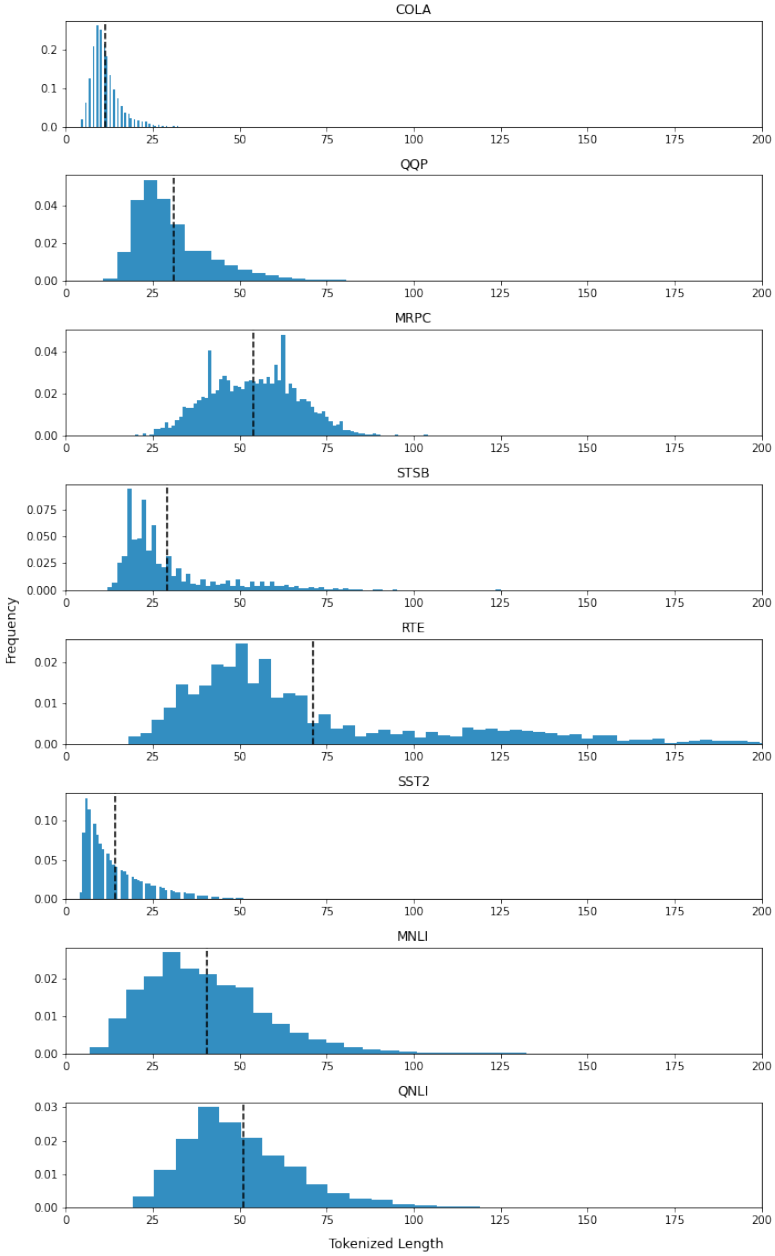


Figure 5.14: **GLUE Tokenized Sequence Length Distribution**: Distribution of lengths of training examples in the GLUE dataset after tokenization with the Roberta tokenizer.

sequence length to train a linear regression model to predict the optimal model. Unfortunately these results were inconclusive, and we did not find a straightforward way to predict optimal architecture from the dataset size or average sequence length.

We hypothesized that datasets with more samples may face less performance degradation with additional downsampling but empirically we see this is not the case, and there is a correlation coefficient of -0.25 but a p-value of .59 between dataset size and performance. We also look at the average length of the samples, hypothesizing that shorter sentences may face less performance drop under downsampling, but again we find this is not the case, with a correlation coefficient of -0.17 and a p-value of 0.71. More details on the length distributions in the GLUE dataset can be seen in figure 5.14.

5.2.6 Comparison of Pretraining Objectives

Our downsampled architectures are initialized with pretrained weights, but this results in a mismatch when using these weights with a different, downsampled architecture. An intuitive approach would be to use additional pretraining, but the sequence length reduction used in the Classformer means this isn't possible with the standard MLM objective because it requires input and output sequences of the same length. In figure 5.16 we see the standard transformer outputs a sequence length that is the same length as the input sequence, while the Classformer uses a downsampled sequence length. This necessitates an alternative form of pretraining, either using an upsampling layer as in the Funnel transformer, or through a modified MLM objective on the downsampled sequence length as in the Classformer. We investigated a variety of MLM objectives on downsampled sequences

1. **First k** - Masked language modelling objective but only evaluated on the first $L_{reduced}$ tokens. Evaluation is not performed on remaining tokens.
2. **Last k** - Similar to First k, but instead evaluation is performed on the last $L_{reduced}$ tokens.
3. γ^{th} **token** - We select tokens evenly along the sequence to evaluate on.
4. **Funnel** - Similarly to the funnel transformer, we add an upsampling layer so that the standard MLM objective can be used. Also has a residual connection to the first layer to add fine grained position information.
5. **Dense** - Predict all the masked tokens, or the first $L_{reduced}$ if total dowsampling performed is greater than a factor of 4. Here locations of tokens are not preserved in the output, so each output does not directly correspond to an input token.

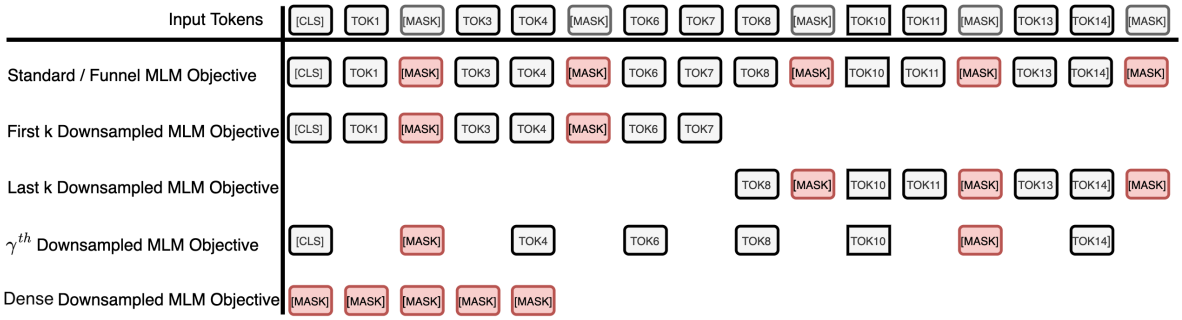


Figure 5.15: Comparison of different methods for masked language modelling. Red tokens represent the tokens the loss is calculated on, while missing tokens mean the token is dropped. For the standard loss the entire sequence is preserved, but with 'first k', 'last k', and γ^{th} a fixed set are dropped based on position, while with *dense* objective all non-masked tokens are dropped.

Looking at figure 5.15, these MLM methods are visualized. Key differences are that the 'first k' and 'last k', and γ^{th} objectives work by preserving relative input locations but reducing the number of tokens that are evaluated on. The *dense* objective differs because each token in the input does not correspond to a specific location in the output. This means the model must both learn which token to mask, but also must remember the relative location. The benefit of this is that many more tokens are predicted, but because of the variable locations the overall loss is about 3x higher compared to standard MLM.

In addition to these novel methods, following the approach of the Funnel Transformer[36] we investigate using an upsampling layer to convert the Classformer output to the size of the input layer, allowing the standard MLM objective to be used. Here we repeat each element of the tensor lengthwise until the full length is reached. For example, if the Classformer reduces the sequence length by a factor of 4, after the output each token will be duplicated 4 times to create an output equal to the input size. To add more fine grained positional information to this, the full length hidden states of the 1^{st} layer are added to the upsampled sequence, similarly to a residual connection[60]. Following this there are two standard attention layers before the final MLM layer.

We investigate the effectiveness of these modified pretraining objectives for overcoming the weight mismatch introduced by downsampling in the Classformer architecture. We use the performance of the model on downstream classification tasks as the objective, since we do not consider the MLM loss to be a relevant objective. For all tests we first initialize the Classformer with the weights from a pretrained Roberta model of comparable size, and also include this model with no additional pretraining as a baseline, which is the

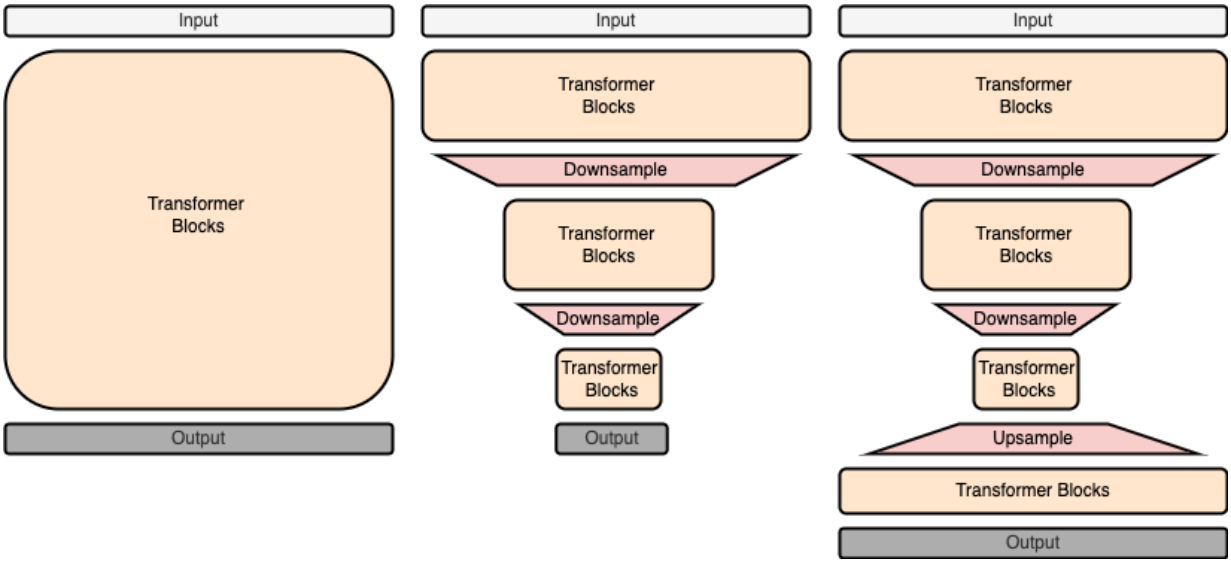


Figure 5.16: Comparison between the transformer (left), Classformer (middle), and Funnel transformer (right) architectures. The original transformer preserves the full sequence length through the entire model, while the Classformer and Funnel transformer both use downsampling to reduce the sequence at intermediate layers, with the key difference being the type of downsampling used and that the Funnel transformer also has an upsampling layer to allow for standard pretraining to be used.

	CoLA	MNLI	QNLI	QQP	SST2	MRPC	RTE	STSB	AVG
None	6.0	74.3	80.0	78.9	85.6	85.5	65.7	78.2	69.3
first k	7.1	75.8	83.9	81.1	85.9	86.1	65.6	83.9	71.2
last k	3.1	76.1	83.4	81.2	87.4	86.7	64.3	84.7	70.9
γ^{th}	0.0	35.4	50.5	0.0	50.9	81.2	52.7	-5.5	33.2
dense	13.2	77.4	85.5	82.0	88.0	87.7	69.3	83.6	73.3
funnel	23.1	79.0	86.4	81.8	88.6	88.1	73.4	83.6	75.5

Table 5.6: Comparison of various pretraining objectives for the models on the GLUE dataset. Models are initialized with weights from Roberta-base, and additional pretraining is done on the WikiText103 dataset. Consistent with BERT, we use the matched MNLI dataset, evaluate on development set, and report accuracy for all tasks with the exception of Spearman correlation for STSB, Matthew's correlation for COLA and F1 scores for QQP and MRPC. The model finetuned for MNLI is used for RTE, STS and MRPC.

first row of table 5.6. This Classformer uses sequence reduction of 50% at layers 1 and 3 ($\gamma_1 = 2, \gamma_3 = 2$).

We evaluate our pretraining methods on the GLUE dataset[164] after pretraining with WikiText103[105], a set of articles from Wikipedia with over 100 million tokens. We followed the pretraining hyperparameters used in the Roberta paper, and ran it for 1.5 days using 4 32GB V100 GPUs. This is relatively small for pretraining, but we did this to reduce compute time so we could run multiple tests to evaluate the best pretraining objective. It also takes much less time to converge compared to standard pretraining because we initialize using pretrained weights from the standard Roberta model.

Looking at table 5.6, we see *first k* and *last k* are quite bad, which could be expected because they bias the model to focusing on only one part of the text, while the entire sequence is important for the classification problem. The γ^{th} method was entirely useless, being much worse than no additional pretraining. The *dense* method shows promising results, improving over the baseline of using Roberta's pretrained weights with no additional pre-training. We also investigated the objective used in the funnel transformer, and found this to be most effective, but also has an additional cost during training compared to other methods because of the additional upsampling layers.

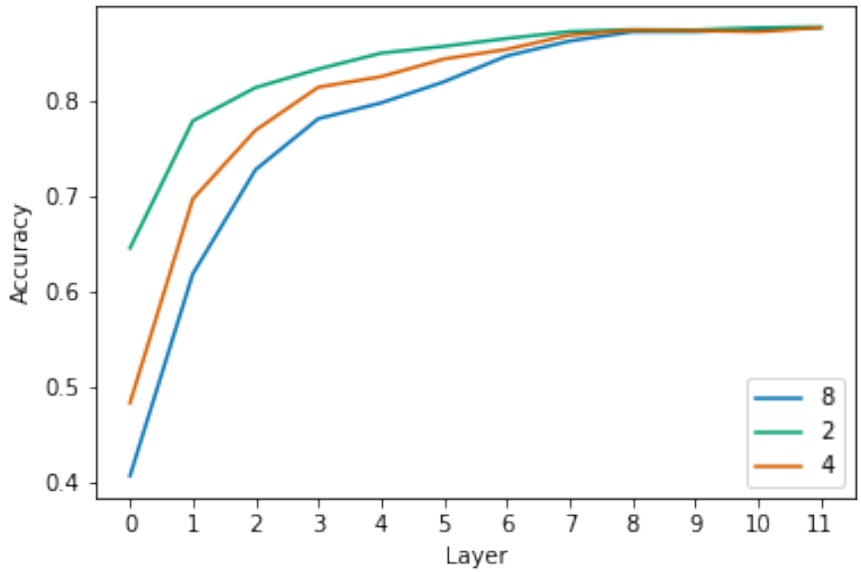


Figure 5.17: Classformer accuracy on MNLI while varying sequence reductions over layers. This shows using sequence reduction of 2x, 4x and 8x on various layers

5.2.7 Investigating the Dense Objective

Given pretraining with the *Dense* objective was able to improve performance on downstream tasks, we further investigate this on the GLUE dataset while varying the layer and amount of reduction. We compare reductions of 2x, 4x, and 8x with models initialized with Roberta pretrained weights, and those with additional pretraining using the dense objective, as seen in figure 5.18.

Here we focus exclusively on sequence reduction at lower layers because this has a greater effect on performance and inference speed than later layers. We demonstrated this with a more extensive study on the effect of downsampling on performance without additional pretraining on MNLI, shown in Figure 5.17. We compared reductions of 2x, 4x, and 8x across layers. The most dramatic effects are at the earlier layers, with the accuracy loss being largest with larger downsampling of 4x and 8x. After the 8^{th} layer even a downsample of 8x has minimal harm to the performance, indicating that for the last few layers high downsampling can be used.

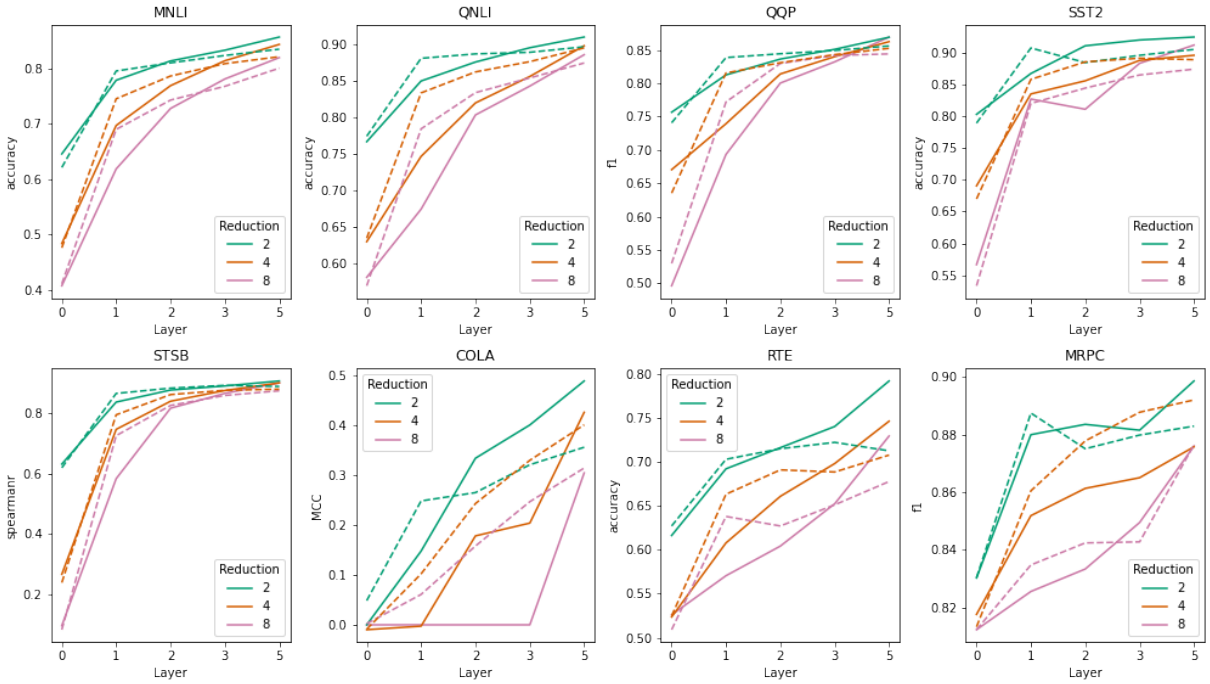


Figure 5.18: Performance on GLUE datasets with models using additional dense MLM pretraining compared to using only pretrained Roberta weights. The solid lines refer to models with no additional pretraining, and the dotted lines refer to models with additional pretraining. The x-axis refers to the layer at which sequence reduction is applied, and the y-axis refers to the performance on the GLUE dataset.

Results of using additional pretraining on the GLUE dataset are shown in figure 5.18. The dotted lines represent performance with the modified dense pre-training, and we can see it gives a consistent performance improvement across all reductions when used on the first layer of the network. It is also generally positive on the second layer, but has minimal benefits for later layers. In general reduction at later layers does not harm performance nearly as much as at earlier layers, so there is less room for improvement from this additional pretraining. Reduction at the input layer completely ruins the performance of the model both with and without pretraining.

We note that performance is highly variable the COLA, RTE, STSB and MRPC datasets because of their relatively small size. Because of this we train each of these datasets three times using a different random initialization each time for the linear layer and take the mean. For this we also follow the Roberta paper's approach to pretraining, using the same hyperparameters and using the model trained on MNLI as initialization for the RTE, STSB and MRPC datasets. We omit running experiments on later layers of the network because it is computationally expensive, and downsampling on higher layers has minimal performance impact.

5.3 Discussion

In this chapter, we approached the problem of reducing representational redundancy in transformer models applied to downstream tasks that have different requirements in terms of sequence information than the tasks the networks were pretrained on.

We introduced the Classformer, a novel transformer architecture using sequence-length bottlenecks for efficient text classification, along with a neural architecture search based approach for adapting it to a specific dataset, hardware and performance requirement. The architecture is created by optimizing sequence-length bottlenecks within the model using Bayesian optimization, maximizing inference speed while satisfying performance requirements.

We individually demonstrated that generating dataset, hardware and performance requirement specific models results in different optimal architectures, providing increases in inference speed. We also demonstrated the Classformer's impressive performance, outperforming transformer models explicitly designed for efficiency on the GLUE benchmark in terms of the inference speed - performance tradeoff. In addition, we introduced MLM based pretraining objectives for the Classformer and demonstrated their effectiveness in instances of large sequence length downsampling.

Chapter 6

Conclusion

This thesis presented three methods for improving the efficiency of deep neural networks across the domains of image classification, generative models, and text classification through the reduction of representational redundancy. In this chapter we summarize the contributions of this thesis, and discuss future work that can be done to further improve deep neural networks along these research directions.

6.1 Summary of Contributions

6.1.1 Weight Sharing

In Chapter 3 we hypothesized there was representational redundancy in convolutional neural networks used for image classification in terms of unnecessary flexibility within the convolution kernels. We created an approach to test this hypothesis through the introduction of additional weight sharing, which can be seen as modifying the optimization process, formulating it as a constrained optimization problem where multiple channels within the same convolution filter are forced to take the same values.

Experiments on both small scale and large scale image classification datasets showed that our method can improve classification performance while reducing the total number of parameters in the model. We show this can be combined with existing methods for creating parameter efficient neural networks without harming performance on Imagenet, and can even improve performance on the CIFAR10 and CIFAR100 datasets

6.1.2 Affine Variational Autoencoder

In Chapter 4 we approached the problem of reducing representational redundancy in generative models, specifically in the latent space of variational autoencoders. We aim to generate a disentangled representation of shape and orientation within the latent space, which can be used to reduce the size of the latent space and improve the quality of the generated images. To do this we introduced the affine variational autoencoder, a variational autoencoder extended through the introduction of two affine layers along with a novel training procedure that enables the unsupervised disentanglement of shape and orientation. More specifically, an initial affine layer performs an affine transform on the input before passing it into an encoder to create a latent space representation which is then decoded normally. This output is fed into a second affine layer, inverting the initial affine transform. The affine transform parameters are learned such that the resulting AVAE can effectively encode input images at canonical orientations, which results in a more compressed representation of the latent variables and a disentangled latent space.

We demonstrate this on the MNIST dataset, showing that this procedure results in a latent space where shape and orientation are disentangled, as well as overall reducing the size of the latent space. We also extend this to 3d affine transforms and show this procedure works for objects in the Shapenet dataset.

6.1.3 Classformer

In Chapter 5 we aim to reduce redundancy in terms of unnecessary sequence information in transformer models. Because transformers are optimized to perform well on a specific task for pretraining, there is a mismatch between the standard transformer architecture and the optimal one for a downstream task, in this case text classification. To overcome this we introduced the Classformer, a novel transformer architecture using learned sequence-length bottlenecks designed to improve inference speed for text classification tasks, along with a neural architecture search based approach for adapting it to specific dataset, hardware and performance requirements.

We individually showed that generating dataset, hardware and performance requirement specific models through this optimization process result in different architectures which are tailored to the specific problem. We also demonstrated the Classformer's impressive performance in terms of inference speed - accuracy tradeoff compared to transformer models explicitly designed for efficiency on the GLUE benchmark. In addition, we introduced a MLM based pretraining objectives for the Classformer and demonstrated their effectiveness in instances of large sequence length downsampling.

6.2 Future Work

In this section, we discuss future work in reducing representational redundancy that can be done to further improve the efficiency of deep neural networks.

6.2.1 Weight Sharing

In this work, we demonstrated the effectiveness of using additional weight sharing in convolutional neural networks, but this can be extended in multiple ways. A straightforward extension is to share weights across multiple layers, similarly to the ALBERT[84] model. This approach was straightforward in ALBERT because transformer models use a set of identical layers, so the same weights could be replicated across layers directly. For convolutional neural networks later layers tend to use larger numbers of filters, so this approach would require partial weight sharing, where only a subset of weights are shared across layers. For example, the first layers weights would be duplicated across all layers, and whenever the filter size was increased additional non-shared weights would be added.

6.2.2 Affine Variational Autoencoder

While the Affine Variational Autoencoder was effective for the MNIST and Shapenet datasets, in its current form it cannot generalize to other datasets where there is more than one object in the image. The AVAE learns a single affine transform for the entire image, so if there are multiple objects in the image, the affine transform will not be able to account for the different orientations of each object. In the case of a 2d image, a complex background would also render the AVAE ineffective. To generalize to this case, it would be necessary to extend the AVAE to learn a separate affine transform for each object in the image, or even for each rigid component of each object.

A possible solution to this would be to apply the AVAE across the image similarly to a convolution, where the affine transform is learned for each patch of the image but the VAE's parameters are shared across all parts. This would perform similarly to the CapsuleNet[134] architecture, but instead of predicting the orientation directly and using routing by agreement, the AVAE would learn the affine transform parameters at the bottom layer based on this disentanglement of shape and orientation. For higher layers, the AVAE would have to act on the latent parameters of the lower level VAE, which would then be mapped back to the original image space to compute a loss. This would be closely related to work on hierarchical VAEs[131, 13, 162] extended to do disentanglement through the

procedure proposed by the AVAE. While this could handle multiple objects in the image, it would not be able to handle a complex background, and would still not be able to fully disentangle real world images, because they are a 2 dimensional projection of a 3 dimensional object. To approach this it would be necessary to map a real world image to a 3d representation[90, 50, 21], before finally using the 3 dimensional AVAE in this hierarchical manner.

6.2.3 Classformer

While the Classformer has achieved impressive performance on a wide variety of text classification tasks, it could be possible to further improve its efficiency through combining it with current methods for efficient transformers. For example, there are some models with sub quadratic or even linear complexity in the sequence length such as the Bigbird[188] or Performer[26] model. We note that many language tasks such as GLUE involve relatively short sequences of length less than 512. These linear complexity transformer models have high fixed costs so they only outperform the standard transformer over longer length inputs, greater than 512[26]. This means that for proper evaluation of these methods we would have to use tasks with much longer sequences.

Another possibility would be to use a more intelligent method to decide which tokens should be dropped from the sequence, instead of dropping every n^{th} token as was done in our work. This could take the form of a better but still fixed pattern of tokens that are dropped, or an approach where tokens are dynamically dropped according to some characteristics during the inference process, as in the PowBERT[53] model where tokens are dropped based on attention scores.

To further improve the performance of the Classformer it may be possible to use more extensive additional pretraining. For example, we could do multitask training on a variety of objectives formulated as language modelling, which was shown to improve performance in T5[121]. Alternatively, we could train on multiple tasks that are very close to the downstream task, which has been shown to improve performance on standard transformer architectures[5].

References

[1] Glue frequently asked questions. https://gluebenchmark.com/faq. Accessed: 2022-06-27.

[2] Quora question pairs. https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs. Accessed: 2022-06-27.

[3] George Adam and Jonathan Lorraine. Understanding neural architecture search techniques. *arXiv preprint arXiv:1904.00438*, 2019.

[4] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[5] Armen Aghajanyan, Anchit Gupta, Akshat Shrivastava, Xilun Chen, Luke Zettlemoyer, and Sonal Gupta. Muppet: Massive multi-task representations with pre-finetuning. *CoRR*, abs/2101.11038, 2021.

[6] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc V Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations. In *Advances in Neural Information Processing Systems*, pages 1141–1151, 2017.

[7] Peter J Angeline, Gregory M Saunders, and Jordan B Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65, 1994.

[8] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.

[11] Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. The fifth pascal recognizing textual entailment challenge. In *TAC*, 2009.

[12] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[13] Rene Bidart. Hierarchical variational autoencoders. https://github.com/renebidart/hvae, 2018.

[14] Rene Bidart and Alexander Wong. Affine variational autoencoders. In Fakhri Karray, Aurélio Campilho, and Alfred Yu, editors, *Image Analysis and Recognition*, pages 461–472, Cham, 2019. Springer International Publishing.

[15] Rene Bidart and Alexander Wong. Affine variational autoencoders: An efficient approach for improving generalization and robustness to distribution shift. *CoRR*, abs/1905.05300, 2019.

[16] Rene Bidart and Alexander Wong. Disentangling shape and orientation with affine variational autoencoders. *Journal of Computational Vision and Imaging Systems*, 7(1):1–3, Apr. 2022.

[17] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–480, 1992.

[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[19] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[20] Daniel M. Cer, Mona T. Diab, Eneko Agirre, Inigo Lopez Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity - multilingual and cross-lingual focused evaluation. *CoRR*, abs/1708.00055, 2017.

[21] Chao Chen, Zhizhong Han, Yu-Shen Liu, and Matthias Zwicker. Unsupervised learning of fine structure generation for 3d point clouds by 2d projections matching. In *Proceedings of the ieee/cvf international conference on computer vision*, pages 12466–12477, 2021.

[22] Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. AdaBERT: Task-adaptive BERT compression with differentiable neural architecture search. *arXiv preprint arXiv:2001.04246*, 2020.

[23] Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems*, pages 8699–8710, 2018.

[24] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[25] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[26] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[27] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE, 2012.

[28] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[29] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015.

[30] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

[31] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[32] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[33] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

[34] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 113–123, 2019.

[35] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine learning challenges workshop*, pages 177–190. Springer, 2005.

[36] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *arXiv preprint arXiv:2006.03236*, 2020.

[37] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[38] Alexandre Défossez, Jade Copet, Gabriel Synnaeve, and Yossi Adi. High fidelity neural audio compression. *arXiv preprint arXiv:2210.13438*, 2022.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[40] Nicki Skafte Detlefsen and Søren Hauberg. Explicit disentanglement of appearance and perspective in generative models. *arXiv preprint arXiv:1906.11881*, 2019.

[41] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[42] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[43] Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[44] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint arXiv:2002.06305*, 2020.

[45] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[46] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[47] Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[48] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *CoRR*, abs/1902.09574, 2019.

[49] Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and William B Dolan. The third pascal recognizing textual entailment challenge. In *Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing*, pages 1–9, 2007.

[50] Georgia Gkioxari, Jitendra Malik, and Justin Johnson. Mesh r-cnn. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[51] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[52] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[53] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. Power-bert: Accelerating BERT inference via progressive word-vector elimination. In *International Conference on Machine Learning*, pages 3690–3699. PMLR, 2020.

[54] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.

[55] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[56] R Bar Haim, Ido Dagan, Bill Dolan, Lisa Ferro, Danilo Giampiccolo, Bernardo Magnini, and Idan Szpektor. The second pascal recognising textual entailment challenge. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, volume 7, 2006.

[57] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[58] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[60] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[61] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[62] Geoffrey Hinton. Neural networks for machine learning: Overview of mini-batch gradient descent, 2012.

[63] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(241):1–124, 2021.

[64] Lu Hou and James T Kwok. Loss-aware weight quantization of deep networks. *arXiv preprint arXiv:1802.08635*, 2018.

[65] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*, 2016.

[66] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[67] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[68] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[69] Xin Huang, Ashish Khetan, Rene Bidart, and Zohar Karnin. Pyramid-BERT: Reducing complexity via successive core-set based token selection. *arXiv preprint arXiv:2203.14380*, 2022.

[70] Forrest N Iandola, Albert E Shaw, Ravi Krishna, and Kurt W Keutzer. SqueezeBERT: What can computer vision teach NLP about efficient neural networks? *arXiv preprint arXiv:2006.11316*, 2020.

[71] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[72] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *CoRR*, abs/1506.02025, 2015.

[73] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[74] Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. Unifiedqa: Crossing format boundaries with a single QA system. *arXiv preprint arXiv:2005.00700*, 2020.

[75] D. P Kingma and M. Welling. Auto-Encoding Variational Bayes. *ArXiv e-prints*, December 2013.

[76] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[77] Durk P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in neural information processing systems*, pages 3581–3589, 2014.

[78] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

[79] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[80] Alex Krizhevsky et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[82] Tejas D Kulkarni, William F Whitney, Pushmeet Kohli, and Josh Tenenbaum. Deep convolutional inverse graphics network. In *Advances in neural information processing systems*, pages 2539–2547, 2015.

[83] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[84] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[85] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[86] Cheolhyoung Lee, Kyunghyun Cho, and Wanmo Kang. Mixout: Effective regularization to finetune large-scale pretrained language models. *arXiv preprint arXiv:1909.11299*, 2019.

[87] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020.

[88] Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth international conference on the principles of knowledge representation and reasoning*, 2012.

[89] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[90] Chen-Hsuan Lin, Chen Kong, and Simon Lucey. Learning efficient point cloud generation for dense 3d object reconstruction. In *proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[91] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.

[92] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *arXiv preprint arXiv:1901.02985*, 2019.

[93] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[94] Hanxiao Liu, Zihang Dai, David So, and Quoc Le. Pay attention to MLPs. *Advances in Neural Information Processing Systems*, 34, 2021.

[95] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[96] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[97] Weijie Liu, Peng Zhou, Zhe Zhao, Zhiruo Wang, Haotang Deng, and Qi Ju. FastBERT: a self-distilling BERT with adaptive inference time. *arXiv preprint arXiv:2004.02178*, 2020.

[98] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[99] Francesco Locatello, Stefan Bauer, Mario Lucic, Sylvain Gelly, Bernhard Schölkopf, and Olivier Bachem. Challenging common assumptions in the unsupervised learning of disentangled representations. *CoRR*, abs/1811.12359, 2018.

[100] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[101] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[102] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017.

[103] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[104] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[105] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[106] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[107] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[108] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.

[109] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[110] Marius Mosbach, Maksym Andriushchenko, and Dietrich Klakow. On the stability of fine-tuning BERT: Misconceptions, explanations, and strong baselines. *arXiv preprint arXiv:2006.04884*, 2020.

[111] Piotr Nawrot, Szymon Tworkowski, Michał Tyrolski, Łukasz Kaiser, Yuhuai Wu, Christian Szegedy, and Henryk Michalewski. Hierarchical transformers are more efficient language models. *arXiv preprint arXiv:2110.13711*, 2021.

[112] Steven J Nowlan and Geoffrey E Hinton. Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493, 1992.

[113] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[114] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[115] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[116] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[117] Matthew E Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. *arXiv preprint arXiv:1705.00108*, 2017.

[118] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[119] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. Technical report, Tech. Rep., Technical report, OpenAI, 2022.

[120] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[121] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[122] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[123] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[124] Prajit Ramachandran, Peter J Liu, and Quoc V Le. Unsupervised pretraining for sequence to sequence learning. *arXiv preprint arXiv:1611.02683*, 2016.

[125] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[126] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[127] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

[128] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[129] Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. Combiner: Full attention transformer with sparse computation cost. *Advances in Neural Information Processing Systems*, 34, 2021.

[130] Karl Ridgeway. A survey of inductive biases for factorial representation-learning. *arXiv preprint arXiv:1612.05299*, 2016.

[131] Adam Roberts, Jesse Engel, and Douglas Eck. Hierarchical variational autoencoders for music. In *NIPS Workshop on Machine Learning for Creativity and Design*, volume 3, 2017.

[132] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[133] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[134] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. *Advances in neural information processing systems*, 30, 2017.

[135] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[136] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[137] Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017.

[138] J Schmidhuber. 10-year anniversary of supervised deep learning breakthrough (2010). no unsupervised pre-training. *By 2010, when compute was 100 times more expensive than today, both our feedforward NNs [MLP1] and our earlier recurrent NNs were able to beat all competing algorithms on important problems of that time. This deep learning revolution quickly spread from Europe to North America and Asia. The rest is history.*

[139] Lukas Schott, Jonas Rauber, Matthias Bethge, and Wieland Brendel. Towards the first adversarially robust neural network model on MNIST. *arXiv preprint arXiv:1805.09190*, 2018.

[140] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[141] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[142] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[143] Mohammad Javad Shafiee and Alexander Wong. Evolutionary synthesis of deep neural networks via synaptic cluster-driven genetic encoding. *CoRR*, abs/1609.01360, 2016.

[144] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

[145] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[146] Xu Shen, Xinmei Tian, Anfeng He, Shaoyan Sun, and Dacheng Tao. Transform-invariant convolutional neural networks for image classification and search. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1345–1354. ACM, 2016.

[147] Zhixin Shu, Mihir Sahasrabudhe, Riza Alp Guler, Dimitris Samaras, Nikos Paragios, and Iasonas Kokkinos. Deforming autoencoders: Unsupervised disentangling of shape and appearance. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 650–665, 2018.

[148] Laurent Sifre and Stéphane Mallat. Rigid-motion scattering for texture classification. *arXiv preprint arXiv:1403.1687*, 2014.

[149] David So, Quoc Le, and Chen Liang. The evolved transformer. In *International Conference on Machine Learning*, pages 5877–5886. PMLR, 2019.

[150] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[151] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[152] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[153] Sandeep Subramanian, Ronan Collobert, Marc'Aurelio Ranzato, and Y-Lan Boureau. Multi-scale transformer language models. *arXiv preprint arXiv:2005.00581*, 2020.

[154] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019.

[155] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. MobileBERT: a compact task-agnostic BERT for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.

[156] Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13:12, 2019.

[157] Kai Sheng Tai, Peter Bailis, and Gregory Valiant. Equivariant transformer networks. *CoRR*, abs/1901.11399, 2019.

[158] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[159] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[160] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[161] Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*, 2017.

[162] Arash Vahdat and Jan Kautz. Nvae: A deep hierarchical variational autoencoder. *Advances in Neural Information Processing Systems*, 33:19667–19679, 2020.

[163] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[164] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[165] Keze Wang, Dongyu Zhang, Ya Li, Ruimao Zhang, and Liang Lin. Cost-effective active learning for deep image classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(12):2591–2600, 2016.

[166] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[167] Wenhui Wang, Hangbo Bao, Li Dong, Johan Bjorck, Zhiliang Peng, Qiang Liu, Kriti Aggarwal, Owais Khan Mohammed, Saksham Singhal, Subhojit Som, et al. Image as a foreign language: Beit pretraining for all vision and vision-language tasks. *arXiv preprint arXiv:2208.10442*, 2022.

[168] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

[169] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.

[170] Lilian Weng. The transformer family. *lilianweng.github.io/lil-log*, 2020.

[171] Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*, 2017.

[172] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[173] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[174] Chien-Sheng Wu, Steven Hoi, Richard Socher, and Caiming Xiong. TOD-BERT: Pre-trained natural language understanding for task-oriented dialogue. *arXiv preprint arXiv:2004.06871*, 2020.

[175] Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430*, 2019.

[176] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[177] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. *arXiv preprint arXiv:2004.11886*, 2020.

[178] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[179] Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *CoRR*, abs/1904.01569, 2019.

[180] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating BERT inference. *arXiv preprint arXiv:2004.12993*, 2020.

[181] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[182] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[183] Donggeun Yoo and In So Kweon. Learning loss for active learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 93–102, 2019.

[184] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*, 2014.

[185] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[186] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917*, 2022.

[187] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[188] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

[189] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.

[190] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[191] Tianyi Zhang, Felix Wu, Arzoo Katiyar, Kilian Q Weinberger, and Yoav Artzi. Revisiting few-sample BERT fine-tuning. *arXiv preprint arXiv:2006.05987*, 2020.

[192] Yu Zhang, James Qin, Daniel S Park, Wei Han, Chung-Cheng Chiu, Ruoming Pang, Quoc V Le, and Yonghui Wu. Pushing the limits of semi-supervised learning for automatic speech recognition. *arXiv preprint arXiv:2010.10504*, 2020.

[193] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[194] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. BERT loses patience: Fast and robust inference with early exit. *arXiv preprint arXiv:2006.04152*, 2020.

[195] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[196] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems*, 4:316–336, 2022.

[197] Barret Zoph, Ekin D Cubuk, Golnaz Ghiasi, Tsung-Yi Lin, Jonathon Shlens, and Quoc V Le. Learning data augmentation strategies for object detection. In *European conference on computer vision*, pages 566–583. Springer, 2020.

[198] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[199] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.