ESTR 4999 FYP LYU2002 2020/21 Term 2 Thesis Report

Yau Chung Yiu, Oscar 1155109029

Study Neural Architecture Search

[Neural Architecture Search on BERT for Network Compression]

Abstract

Due to the limitation of manually designing neural network architecture, Neural Architecture Search arises to algorithmically learn the suitable network architecture for machine learning tasks. This report will emphasize on two elements of this project, i.e. Neural Architecture Search and its application on BERT, an attention-based neural network for natural language understanding. After experiments, we realized prediction distillation is the most effective objective for sub-architecture searching over the multi-heads and the feed-forward layer connection. The latest experiment result shows that the result of our architecture searching algorithm can surpass the performance of the existing BERT models of similar architecture computational complexity.

Abbreviation

AutoML – Automatic Machine Learning
BERT – Bidirectional Encoder Representations from Transformers
FLOPS – Floating Point Operations Per Second
LSTM – Long Short-Term Memory
NAS – Neural Architecture Search
NLP – Natural Language Processing
RNN – Recurrent Neural Network
TinyBERT 4L – A variant of BERT with 4 hidden layers from [28]

Index

1 Introduction

To understand NAS, we are trying to experiment with the possibility of NAS on deep neural network. Existing research results mainly focus on the implementation of NAS on state-of-the-art neural network modules such as convolution, residual connection, which shows the best performance on image cognition problems. Thus, we decide to work on the less explored architectures of neural network.

We have seen the rapid development and success of deep neural network on natural language processing problems. A new emerging architecture named Transformer caught all the attention in the natural language processing community. Considering the constraints of our resources, we decide to focus on BERT and its application on sentence pairs classification problems using GLUE dataset. The transformer mechanism utilities the correlation of pairs of words within the sentences to infer information about the contextual meaning of the sentences.

We propose to apply NAS on BERT architecture and perform network compression on the architecture. We foresee that at the end we should be able to remove redundancy in the architecture and reduce the number of parameters in the network. We might also hope that the network would improve in accuracy, as network compression can be thought of as an action of regularization.

2 Background Study

2.1 Neural Architecture Search, and AutoML

A family of methodologies that allows computers to automatically learn the better computational model to solve a specific task is called Automated Machine Learning. Intuitively it can perform architecture development just like a machine learning developer will do, but better at being data-driven.

It is common to describe the problem of AutoML as a Combined Algorithm Selection and Hyperparameter optimization problem, dubbed as CASH [7]. In a CASH problem, we are trying to minimize the evaluation loss of the model trained on the training dataset, where the model is parametrized by the hyperparameters and the choice of algorithms. This equivalently captures the idea of finding the best solution to solve the existing problems, using machine learning.

Under AutoML we have three popular areas of study, namely hyperparameter optimization, meta-learning and neural architecture search.

Hyperparameter optimization [1, Chapter 1], as its name suggests, focuses on searching for the best hyperparameters of a machine learning model to attain the best performance. Common hyperparameters of a model are learning rate, batch size, number of training epoch etc. While it is not the focus of our project, it is worth to mention that hyperparameter optimization overlaps a lot with NAS. We can think of the architecture of a network as one of the hyperparameters of the network.

Meta-learning suggests using meta-data to lead the learning of our model. Meta-data is the data we get from learning other models on different datasets. Across datasets and across machine learning models we can observe and calculate statistically what is the factors behind that leads to the success of some models and the failure of some other models. For example, if we know certain models will not perform well on some tasks, we can predict that they will not perform well on similar tasks. Meta-learning utilizes this idea and allows the computer to learn how to learn [1, Chapter 2]. For example, Auto-PyTorch Tabular do both NAS and hyperparameter optimization on tabular datasets and set up a benchmark called LCBench for learning curve prediction [2].

NAS covers all the methods that use automatic algorithms to design the architecture of a neural network. NAS algorithms can be categorized according to its search space, search strategy and performance estimation strategies [1, Chapter 3]. Inside the search space are all the candidate architectures for the task. At each iteration of the searching, we sample one architecture from the search space for evaluation of its performance, using the performance estimation strategy. The most intuitive way to estimate the performance of the architecture is to use a training dataset for training until convergence and perform evaluation on the unseen dataset as the estimated performance of the architecture.

Most of the time NAS procedures are computationally expensive due to the cost of performance estimation. Training cost of a deep neural network can be as expensive as up to a GPU day. The more architecture that we have evaluated on, the more information about the search space we have and the higher chance that we can evaluate on a suitable architecture that performs well on the given tasks. This situation makes NAS different from other machine learning algorithms, where in general we can monitor the learning progress of a neural network by looking at its validation results and infer that whether the model is converging or overfitting. But in NAS the search will not overfit on sampling too many architectures. So we cannot adopt strategies such as early stopping in our searching process. This motivates the choices of better search strategies and better performance estimation strategies that do prediction on the performance of the architecture without training the architecture to convergence, to overcome the large cost deep neural network training. Searching strategies that could achieve efficiency by this idea includes Bayesian optimization and gradient-based algorithms.

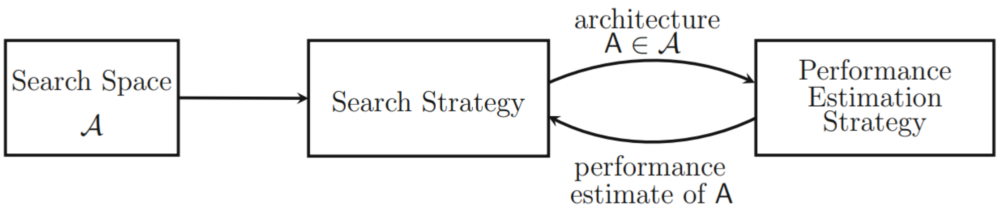


Fig. from [1, Chapter 3]. An illustration of NAS.

2.1.1 Search Strategy

Here we will describe the details of different searching strategies. The major classes of NAS algorithms according to the search strategies are searching by Bayesian optimization, reinforcement learning, genetic algorithms and gradient-based algorithms.

Bayesian optimization is the method that allows us to predict a certain function without a full evaluation of the function [6]. This is very helpful to NAS, when especially the function of interest is the function of network performance, given the hyperparameters and architecture as the input of the function. Using Bayesian optimization is like exploring an unknown function, and in each iteration, Bayesian optimization will tell us what is the best point to evaluate on to get a more accurate estimation of the high points of the function. With the objective of finding the best architecture, Bayesian optimization will save us a lot of time from preventing evaluation on weak architectures, or architectures that will not bring us new information about the best architecture.

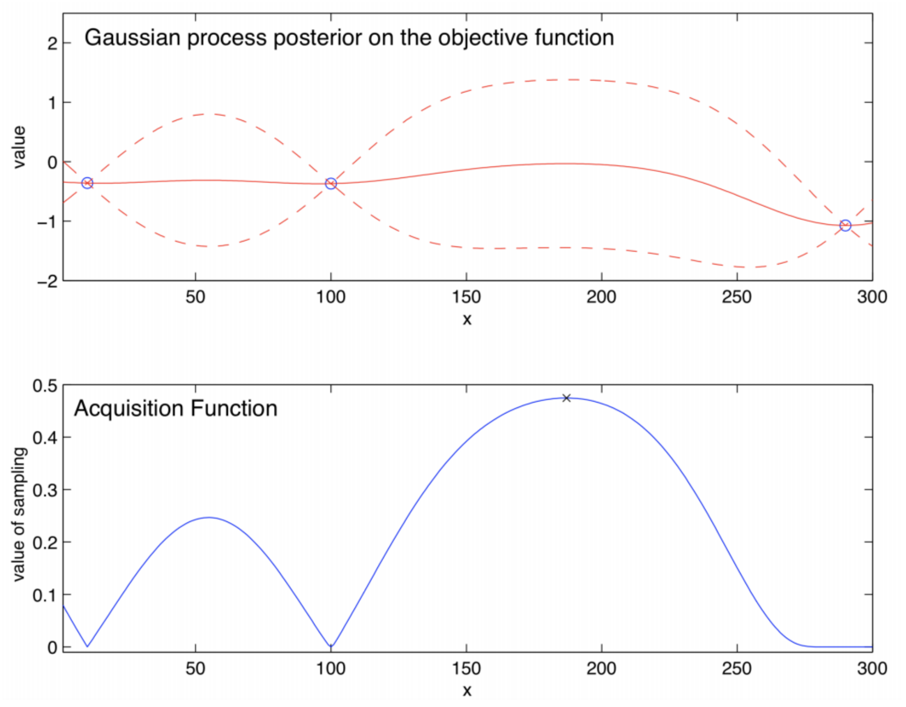


Fig. from [6]. Top: 3 blue points represent observed data point, which can be referred as the evaluated architectures in NAS. Dashed red lines represent the confidence intervals of the estimation of unobserved input. Bottom: Acquisition function suggests the next relevant points of sampling to get the most unknown information about the distribution.

When we consider architecture searching as a task of reinforcement learning, the agent's action will choose how to build the next architecture for evaluation. The evaluation results of the architecture become the reward of the agent. So, the reinforcement agent will use achieving the best network performance as its target to generate the best architecture according to the given data and tasks.

Using genetic algorithms to produce the best architecture for a certain task is also popular as an intuitive method to perform NAS. In a population each individual is a candidate architecture, and through mutation process new architectures will be generated, for example, by picking the fittest parent and generate its offspring by applying mutation on its architecture [1, Chapter 3]. Each candidate is evaluated on the unseen dataset to get their own scores as a fitness function, so suitable genes of architecture that perform well on the tasks will survive in the population.

Gradient-based searching stands out to be one of the most efficient searching strategies. We can see from the above strategies that evaluation of the architecture is done for each iteration of searching, for example genetic algorithms have to train each of the candidate architectures to convergence in order to calculate its fitness score. For larger architecture or larger dataset, the cost of training an architecture is huge, thus becomes the reason of inefficiency. In contrast, gradient-based methods enable us to learn the architecture during the training of the network. Extra learnable weights are attached to the network, which serves as probabilities of the possible architectures. If we are training to minimize the training loss and architecture loss at the same time, we could have the gradient of the architecture weights with respect to the losses and have the architecture weights trained together with the original weights of the network. For example, [3] has shown a differentiable searching algorithm to search for the width and depth of a ResNet [4], a convolutional neural network with residual connection.

In our project, we decide to use a gradient-based searching strategy to perform NAS by the reason that it is more feasible to execute within a reasonable period of GPU hours.

2.1.2 AutoML System

To get a better understanding of the state-of-the-art AutoML methods, here we introduce a few of those that become successful in the community of AutoML. Essentially each of the following is a product of “easy to use” machine learning library that allows normal users with no expert knowledge about machine learning or about the data to perform machine learning tasks.

Auto-Keras is the realization of NAS that uses Bayesian optimization to guide the network morphism [5]. For efficiency of the searching procedure, Auto-Keras perform Bayesian optimization that searches for architectures on CPU while in parallel it performs model training on GPU so that evaluation results are passed back to the Bayesian optimization searcher to update its estimation of the performance graph of the model. By the constraint of Bayesian optimization methods that the search space of parameters needs to be continuous, it is not applicable to NAS since network architectures are discrete. To tackle this problem Auto-Keras decided to use the edit distance of two architectures to lay the discrete architecture onto continuous dimensions. It means that two architectures are close to each other when it only takes a few numbers of morphing to transform one architecture into the other one. Finally, it is reasonable to see that Auto-Keras focuses on searching the architecture of a deep neural network since we expect that a deeper network creates flexibility for the architecture to be changed, and thus being more probable that we will see a good performance of deep neural network after architecture searching comparing to shallow machine learning models.

In contrast to searching for the best deep neural network architecture for a certain task, Auto-sklearn takes the traditional machine learning approach and put together an ensemble of weak learners as the resulting model for the given task [7]. The system performs meta-learning to initialize the prediction of its Bayesian optimization searcher as a warm-starting operation. While in each searching iteration the Bayesian optimizer will search for the best hyperparameter for the base learner, after evaluation the trained weak model is also saved and serve as a candidate to participate in the final ensemble of the model. In the end, the system will perform ensemble selection using a held-out set of data to produce the ensemble of models.

2.2 Network Compression

State-of-the-art deep neural networks have a large size of trainable parameters, and certainly leads to more computational operations and longer inference time is required. For systems that want to achieve real-time performance, this is the bottleneck that limits the representation power of the neural network. Under the study of network compression, there are methods that will decrease the number of weights in the model while maintaining the model performance. In the following, we will try to introduce some of the existing compression methods.

2.2.1 Weights Quantization

In modern use of neural networks, weights of the network are often represented as 32-bits float point values for precision. Using parameter quantization we can reduce memory consumption by limiting the precision of weights down to 16-bits or 8-bits, while correspondingly providing up to 2x and 4x more memory space on the GPU. This is advantageous in the case that we need to train very deep neural networks, as one of the bottlenecks for increasing model complexity is the memory size of a GPU. At the extreme people have tried using binary numbers as the weights of the network, however in practice this often suffers from the significantly lowered accuracy of the resulting model [8].

2.2.2 Network Pruning

Network pruning achieve network compression by removing less informative and less impactful connections from the network to save computation. Early approaches such as using the second-order gradient information of the loss function to deduce the contribution of neurons have shown superior performance over the others in the early times [8]. Network pruning is always applicable to connections of feed-forward layers, which are common types of connections among all the neural networks. Similarly approaches such as imposing L1-norm regularization can also be thought of as a kind of network pruning, since L1-norm regularization induces sparsity in the connection of the network. For the same reason we can say that network pruning has the same effect of regularization, which will limit the learning capacity of the network and avoid overfitting.

2.2.3 Low-rank Approximation

Aside from feed-forward layer, another common type of layer is convolution. Low-rank approximation allows us to split a big convolution filter into two smaller convolution filter, which reduces the number of learnable weights and the number of computational operations. Two famous examples of such kind are called spatial separable convolutions and depthwise separable convolutions [9]. Spatial separable convolution deals with the case where we want to factor a 2D convolution filter into two 1D convolution filter, thus the number of computations for a $m \times n$ matrix reduces from mn to $m + n$. On the other hand, depthwise separable convolution is more power as it also deals with convolution across channels. It is common to see a network that adopts the use of 3D convolution filters. Depthwise separable convolution makes it possible to reduce computation by separating convolution into two parts, depthwise convolution and pointwise convolution. In depthwise convolution, we will be using k filters to learn the spatial characteristic within the same channel, for k input channels. Using the output of depthwise convolution we perform the second step called pointwise convolution, which uses h $1 \times 1 \times k$ filters to learn the cross-channel information and generate an output of h channels. We see that separable convolution is applicable to all shapes of convolution layers while being able to show a significant reduction in computational complexity.

2.2.4 Knowledge Distillation

If we look at a very deep neural network, we might ask ourselves how much each layer is contributing to the prediction results. When it is the case that some of the layers are redundant or even be no-op that does not learn any details of the data we will consider doing knowledge distillation. In the setup of knowledge distillation, we will transfer the knowledge of a trained teacher network to a smaller student network. It can be further separated into two types of distillation, the prediction layer distillation and intermediate layer distillation. In the prediction layer distillation, the final output of the smaller student network is trained with respect to the output of the teacher network.

This is based on the belief that the knowledge of the teacher is more informative than the data labels, which allows the student network to generalize well [10]. We can illustrate this by an example of MNIST, the recognition of handwritten digits. A typical MNIST classifier will have the final layer having softmax output, representing the probability of each class that the digit will belong to. To see how informative a teacher network output can be we can assume that upon taking a '2' as the input, the teacher network will assign $P(x \text{ is } 2) = 0.95, P(x \text{ is } 3) = 0.03, P(x \text{ is } 7) = 0.02$ and all the other classes as 0. These softmax outputs are referred as the "soft target" of the student network. From this distribution we can see that the teacher correctly predicts the input to its corresponding class, which provides the same information that the data label can give, while in extra is telling us that the digit '2' is more similar to the digits '3' and '7' than the other digits. If the student network can also generate similar distribute of probability at its output layer, it is very likely that the student network has learnt the useful features from the input data that allows the model to make correct classification decision. If we extend the idea of knowledge distillation further into the output of the intermediate layers of the teacher network we have intermediate layer distillation. Each layer of the student network will be trained to match the intermediate layers' output of the teacher network. Intuitively this is similar as we are trying to let one layer of the student network to mimic the operation of several layers of the teacher network. For example, in [11] this idea is referred as "Task-useful Knowledge Probe", where the authors are performing task-specific fine-tuning of the language model BERT together with knowledge distillation to achieve network compression.

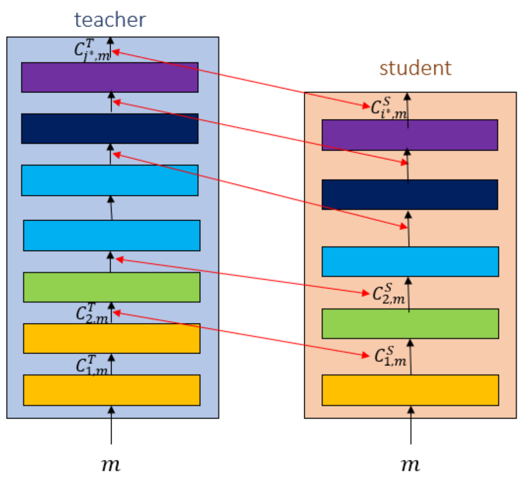


Fig. Example illustrating intermediate layer distillation, red arrows represent that the student is training w.r.t the output of intermediate layers of the teacher network, the same color of the layer blocks represent that they have a similar function for processing the input.

There are also cases where the student network is deeper than the teacher network but thinner [12]. This will allow the student network to perform a richer feature extraction process, while still being better than the teacher model by having shorter inference time and smaller model size.

While there are a variety of techniques to do network compression, a rule of thumb is that we should not sacrifice too much of the performance of the network for efficiency, unless under the situation where memory efficiency or computation efficiency is more important than the accuracy of the model. Network compression can not only reduce the resource required to build and use a deep neural network, it also enables us to train a deeper neural network and gain more learning capacity.

2.2.5 Lottery Ticket Hypothesis

Inspired by all kinds of pruning techniques, [32] proposed the famous Lottery Ticket Hypothesis which states that a randomly-initialized dense neural network contains a subnetwork which is able to be trained to match the test accuracy of the original network under the same initialization of the original network. An analogy to lottery ticket is mentioned where different subnetwork of the initialized neural network is similar to a lottery ticket and a larger model have more combination of subnetwork, thus it has a larger chance of winning the lottery, i.e. converge to the parameters that obtains high test accuracy.

In [32] the author also proposed an iterative pruning method such that it can identify the winning ticket (i.e. the best subnetwork) after n rounds of iteration:

1. Randomly initialize a neural network $f(x;\theta_0)$ (where $\theta_0 \sim \mathcal{D}_\theta$).
2. Train the network for j iterations, arriving at parameters θ_j.
3. Prune $p\%$ of the parameters in θ_j, creating a mask m.
4. Reset the remaining parameters to their values in θ_0, creating the winning ticket $f(x;m\odot\theta_0)$.

By repeating the above algorithm for n iterations, each time with $p = (p^*)^{\frac{1}{n}}$ where p^* is the final ratio of weights we want to prune, this is referred as Iterative pruning in [32]. In step 3 we are pruning the first $p\%$ smallest magnitude parameters. Intuitively the parameters with small magnitude should be the least influential in the network.

In the discussion of [32], it is hypothesized that the initialization of weights is the core behind determining the subnetwork to be well trained.

Beyond the idea of Lottery Ticket Hypothesis [34] found out the phenomenon that within a large randomly weighted network there exist subnetwork that performs well, even without training the network weights. These results strongly suggest the importance of an architecture over weight optimization, and reveal more underlying unknown about state-of-the-art deep neural networks.

2.3 Natural Language Processing

Natural language processing concerns with all the machine learning algorithms that allow computers to understand and extract useful information from sentences of the same language. This area is getting most of the attention of the machine learning community, with its advancement in architecture that it brings to the study of machine learning. In the old days, the common approach is to use a recurrent network to capture contextual information from a sequence of words. Notice that in NLP we need models that could handle sequence to sequence operation. By design, popular layer choices in other problems such as convolution will not work for NLP problems since information of a sentence is not positional invariant property. The order of word appearance will affect the meaning of the word. For example, for sentiment analysis problem such as customer review analysis, "Although I like the product, but the delivery is too late." and "Although the delivery is too late, but I like the product." will have different scores of positiveness, where the first one is more negative than the second one. By this we see that using convolution operation will not help much in problems of NLP. Instead, there are other kinds of neural networks that are more suitable for NLP.

2.3.1 Recurrent Neural Network, LSTM

Recurrent neural network has first been used to tackle NLP problems. Due to its cyclic connection between current states and previous states, RNN can model sequential information flow and have been successfully used for sequence labelling and sequence prediction tasks [13]. RNN is a sequence to sequence model, that it inputs in a sequence of data input and outputs a sequence of hidden representation. At the segment of input data, RNN will take the combination of the internal output of the previous layer and the data to generate its output. The internal output acts as an internal memory of RNN, which allows the network to remember useful information that is encountered while taking in the previous segment of the input data. Sometimes the internal outputs will be referred as the hidden states of the RNN.

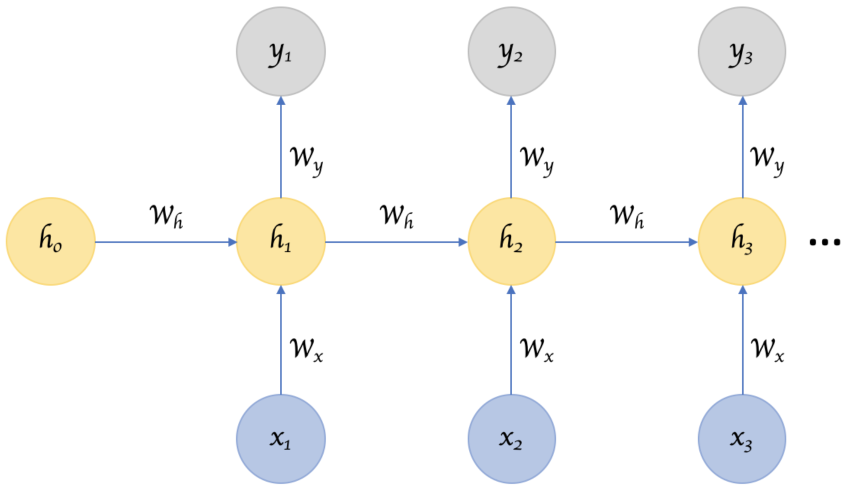


Fig. from [14]. An illustration of the computational graph of RNN. h_i represents the internal output (hidden states) of the RNN at time i after manipulating the input x_{i-1} and h_{i-1}. y_i are the outputs of the RNN at time i.

One limitation of the vanilla RNN is that it can only consider sequential information flow in one direction. This leads to an improved version of RNN called Bidirectional RNN, where essentially it is two vanilla RNN that one takes the input sequence from head to tail while the other one takes the input sequence from tail to head. The i-th output of the two RNN is the combination of the i-th output of each of the RNN.

Another problem is that it is difficult for vanilla RNN to represent the long-term dependencies within the sequence in practice. To further expand the representation power of RNN, the popular and practical solution of RNN type network is the Long Short-term Memory network, introduced back in 1997 [15]. In LSTM, it has expanded the mechanism of the internal states of RNN. There are several gates around the hidden states to allow LSTM to decide what to remember and what to forget in its internal memory. The input gate unit is to protect the memory content and allows the network to decide what information is allowed to manipulate the hidden states. The output gate unit is to control what content to flow out of the hidden states, and irrelevant memory contents will not be received by the neurons outside. Finally, the forget gate decides whether to accept the hidden state from the previous layer or not, depending on the current input and current internal state. Such a mechanism allows LSTM to model long-range dependencies and learn useful information out of the sequential data input.

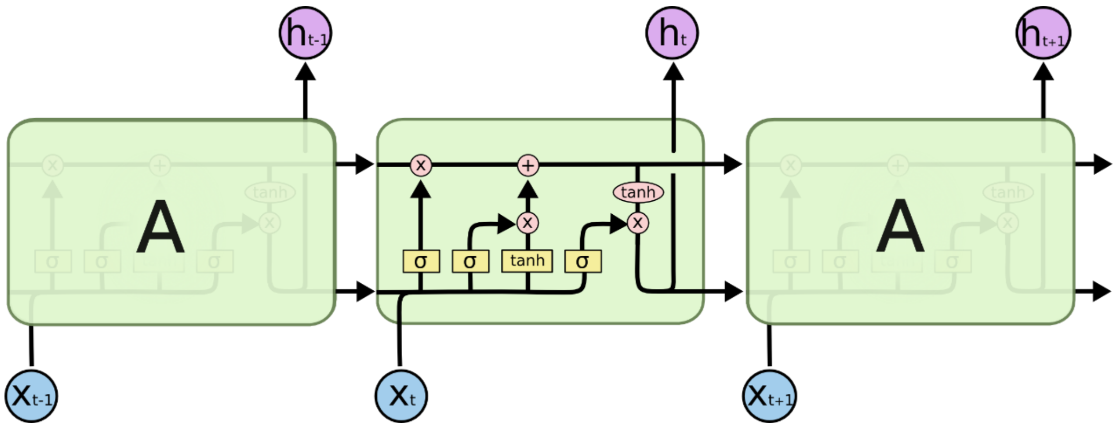


Fig. from [16]. An overview of LSTM. The non-linearities in the connection of gates expanded the learning capacity of the memory mechanism.

2.3.2 Transformer

Right now, when we talk about the state-of-the-art model for modelling sequence many people would refer to a better design of architecture called self-attention mechanism. Transformer is the sequence transduction model that only uses self-attention mechanism, dispensing with recurrence and convolutions entirely [17]. To understand self-attention, we can assume that within a sequence each word is probably related to each other word. Self-attention tries to capture all pairs of relationship between the word tokens within the sequence. We say that a token *i* is attended to the other token *j* when the attention score of the *i,j* position is large. For a better understanding, we look at the following figure and explain how we do computation from the input of the attention layer to the output of the same layer.

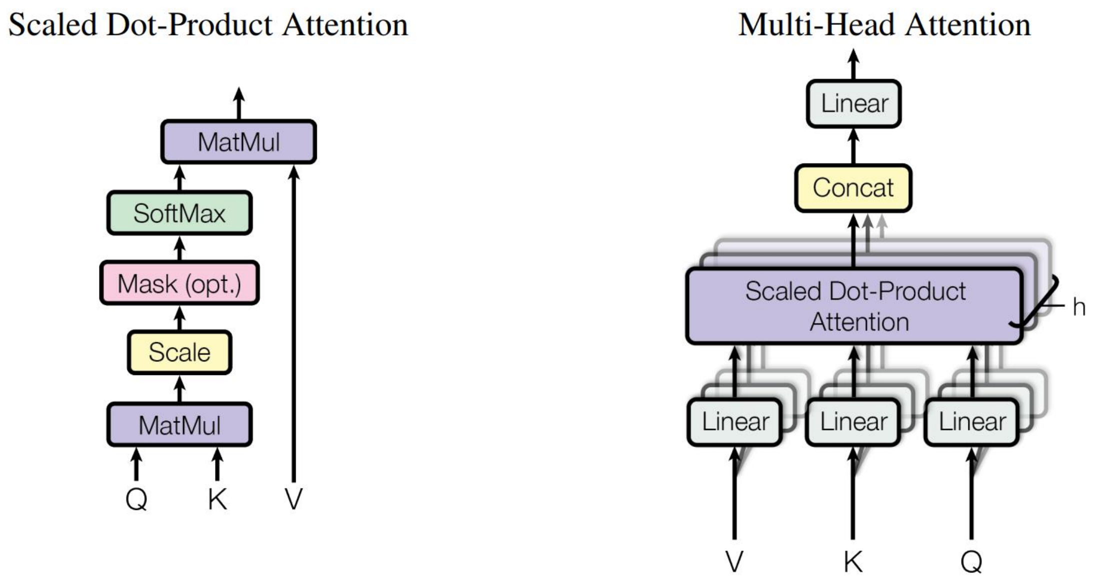
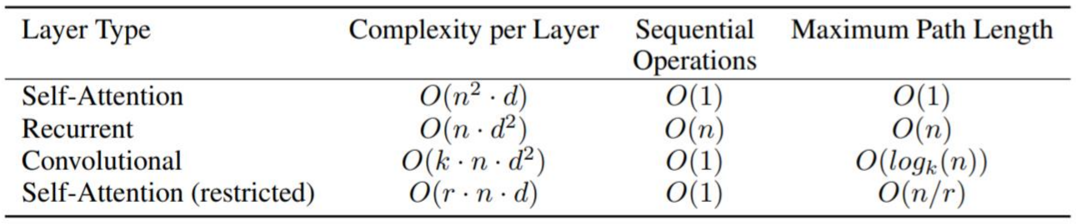


Fig. from [17]. On the right side is the illustration of a self-attention layer. On the left *side is the illustration of the attention mechanism.*

Generally, the self-attention layer takes in three sequences of input, namely the query sequence (Q), the key sequence (K) and the value sequence (V), where the key and value sequences come in pair and describe a certain value of a key. Each of the input is linearly transformed into sequences of the head size, where the head size is the size of a hidden representation within the self-attention mechanism. Within the scaled dot-product attention computation, each query position will be attended to each key position. A high value will be yield in this step if this query has related semantic information with the specific key, which is further passed through a softmax to generate probability weight. With this attention weight, it is used to weight the values of the corresponding keys and summed up as the output for this query. In practice, the attention computation of all tokens in the query is done by matrix multiplication for computational efficiency. So, in forward through the self-attention mechanism, only two matrix multiplication is required, as shown on the left of the figure above. This is significantly important to sequence transduction process since we want to allow long-range dependencies within the sequence. The operation required between temporal sequence unit is bounded by the operation stated on the left of the figure, which shows that we ensure a constant path length for the temporal information flow [17]. Unlike the situation in RNN where the path length of temporal information flow can grow in $O(n)$, depending on how far away the locations of the two information is within the sequence.



Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table from [17]. Comparison of self-attention with recurrent layer and convolutional layer. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r is the size of neighbourhood in restricted self-attention (masked self-attention). Showing that self-attention is more parallelizable and has a constant path length of long-range dependencies.

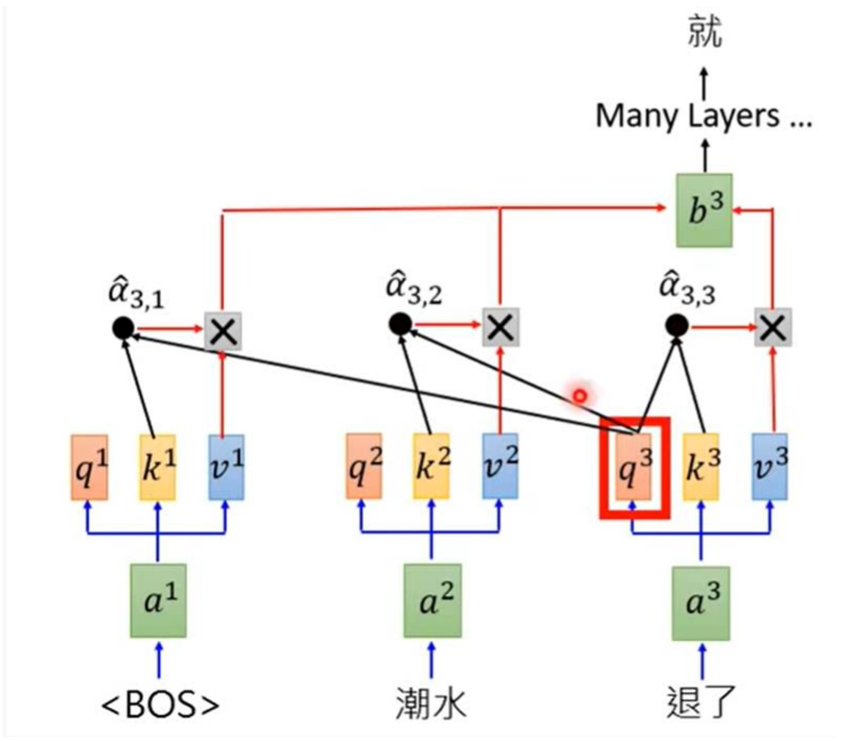


Fig. from [18]. An example of the attention mechanism used in the decoder of Transformer. The query will match with all the related keys in the input sequence. In this case, attention is masked and is limited to the word tokens at earlier locations.

In practice, we can assign different kinds of input to query, key and value for different desired behaviour of the attention. In the Transformer architecture, it adopts the encoder-decoder structure and uses self-attention differently in the encoder and the decoder. The following figure will illustrate the architecture of Transformer.

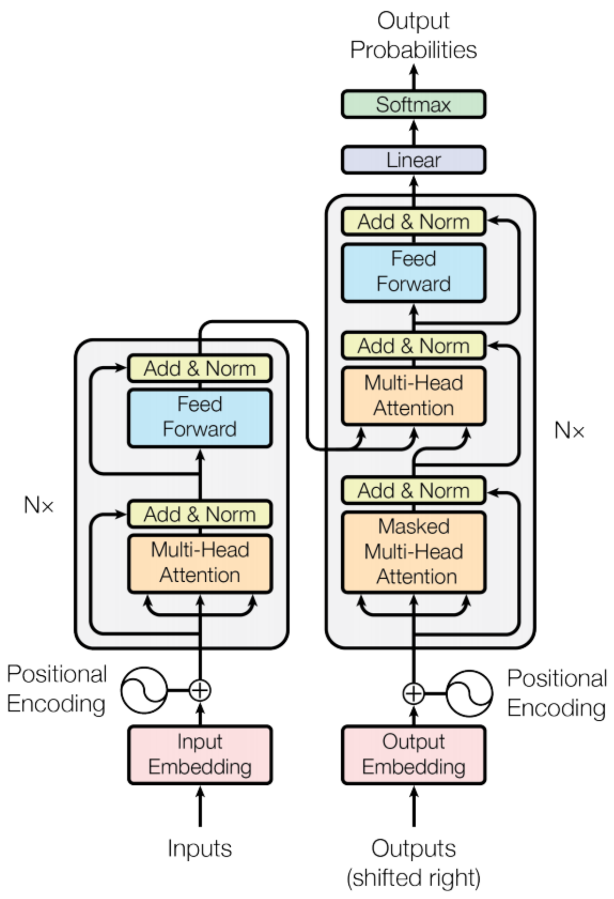


Fig. from [17]. The architecture of Transformer. On the left is the encoder stack, on the right is the decoder stack.

The encoder-decoder structure generally can be understood as the encoder takes the original input and learn a representation of the input sequence, and the decoder generates a sequence of output according to the learned representation of the data from encoder and the previously outputted sequence tokens by itself.

The encoder of Transformer will take the input of data word sequence, such as sentence or pairs of sentences, and use learned embedding to convert the input tokens into embedding token sequence. Notice that positional encoding is added to the embedding sequence because from the design of attention mechanism we know that there is no ordering information during the calculation of attention. When positional encoding information is added to the embedding tokens ordering information is kept within the embedding token, so the network has the chance to refer to the ordering information during attention. For self-attention in the encoder, the input embedding sequence is used as query, key and value inputs equally, to let the network to learn the contextual knowledge of the given sequence and generate the encoded representation of the input sequence. After self-attention we have a feed-forward layer, to generate the output of one encoder layer. For self-attention in the decoder, they serve different purposes as those in the encoder. At each step of decoding, the decoder is only allowed to look at the full encoded representation and the previously decoded output tokens. That is why a masked self-attention layer is used in the lower half of the decoder, to ensure that there is no leftward information flow during decoding and preserve auto-regressive nature of the decoding process. There is one more type of self-attention in the decoder, referred as the "encoder-decoder attention", that it takes the encoded representation of the corresponding encoder as the memory keys and values. The query is given by the output of the previous masked attention, which is essentially the output of the previous layer of the decoder stack. This serves the purpose that we want the decoded sequence to consider all the information we have from the whole sequence, by taking in the representation learned by the encoder. Notice that across each layer of operation there is a residual connection, which allows gradient to propagate further and allows Transformer to stack more encoder-decoder structure while still being able to train well.

2.3.3 Language Modelling: ELMo, GPT, BERT

To handle NLP tasks, we want our model to understand the syntax and semantics of the language in order to learn useful information from sentence input. State-of-the-art projects use large text corpus to perform unsupervised pre-training of the model. Generally, language modelling requires the model to be capable of predicting future tokens or missing tokens of the sentence, which is kind of a behaviour of understanding the language. The following paragraphs will introduce three well-known methods of language modelling.

First, we have a language model that uses bidirectional LSTM to generate its contextualized word representations, famously known as ELMo [19]. The representations learnt by ELMo are contextualized in the sense that the same word appearing in different locations could have different learnt representation. Bidirectional language model essentially uses a forward LSTM and a backward LSTM to generate the representation and jointly maximize the log likelihood of both two of the outputs given the history of the sequence, where for a sentence t, $t_1, \dots, t_{k-1}$ is the history of the forward LSTM and $t_{k+1}, \dots, t_N$ is the history of the backward LSTM when computing the probability of t_k. We can train a deep bidirectional LSTM by taking the output of each LSTM as the input of the upper layer LSTM, which allows the model to learn a deeper representation of the input sequence. ELMo builds on top of deep bidirectional language models by learning the task-specific weighting of the intermediate layer representations in a stack of bidirectional language models. ELMo puts together all the weighted representation of the token and generates its own representation, and these richer representations can be used as extra inputs to other supervised models solving downstream NLP tasks, where this is referred as a feature-based approach.

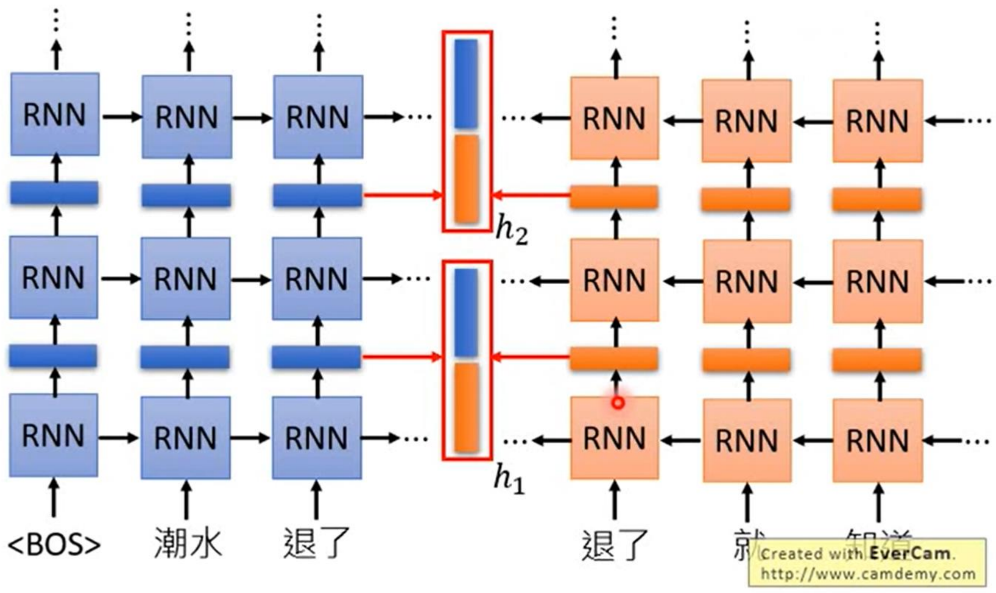


Fig. from [18]. An illustration of bidirectional LSTM used in ELMo. h_1, h_2 *in the middle represents the intermediate layer representations of the stack of LSTMs.*

Second, we have an attention mechanism based approach that uses the decoder of Transformer to perform language modelling, which is famously known as GPT [20]. GPT performs its unsupervised generative pre-training on large text corpus by a forward language modelling objective, by using a stack of decoders of the Transformer and masking out the future attentions to simulate forward predictions. At the last layer of the decoders, a softmax layer is used to model the probability distribution of the target tokens we want to predict. After pre-training, according to the downstream task we want to solve we perform supervised fine-tuning of the parameters with respect to the target, and depending on the structure of the downstream tasks we will need to use the output of the decoder differently, where usually the final layer would be a linear connection. During the fine-tuning process, GPT continues unsupervised language modelling as an auxiliary objective, which can improve the generalization of the supervised model and accelerate convergence [20].

At last, we would like to introduce BERT [21], which uses a stack of encoders of the Transformer to build a deep bidirectional language model. Bidirectional understand of the sentence is done by the attention mechanism. During unsupervised pre-training, two objectives are used to perform language modelling, i.e. the masked language modelling (Masked LM) and next sentence prediction (NSP). At each iteration of pre-training, a percentage of input tokens are chosen at random to either be masked by 80% probability, or be replaced with a random token by 10% probability or be unchanged by 10% probability, and BERT is required to perform predictions of the original token on these chosen locations as to perform language modelling. At the same time, the given masked sentences come in pairs and BERT is required to predict the relationship of the two sentences, whether the given sentence B is the next sentence of sentence A as a binary classification task. The combined pre-training methods of BERT allows the model to utilize the language information given by surrounding tokens, and understand language structure across sentences, which is useful for some downstream tasks such as Question-Answering. After pre-training, we perform downstream tasks fine-tuning on the BERT model using the same pre-trained model parameters for initialization. It is reported that the fine-tuning procedure is way less expensive than the pre-training procedure [21], and in fact it is obvious by the difference of sizes of the dataset used.

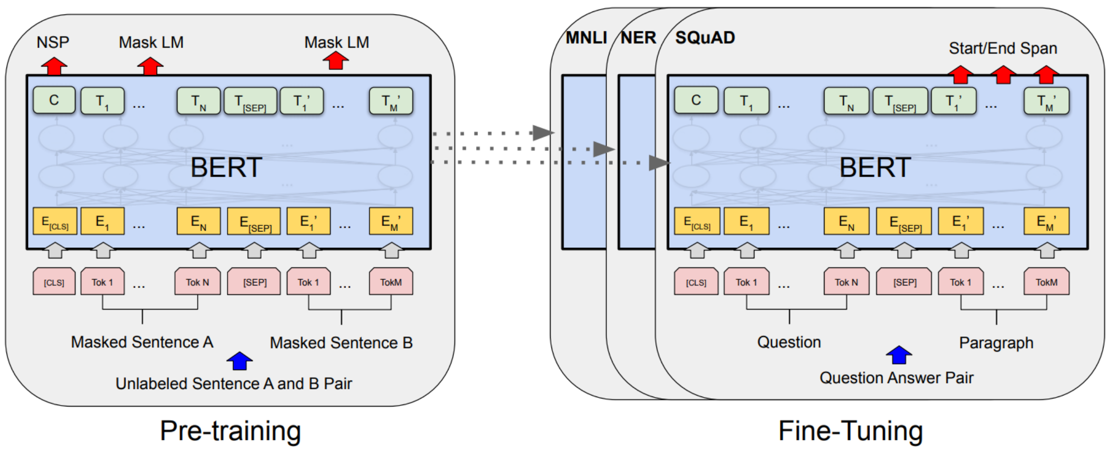


Fig. from [21]. Overall pre-training and fine-tuning procedures for BERT. Unsupervised pre-training is shown on the left, where two tasks are simultaneously used to train the model. Supervised fine-tuning is shown on the right, depending on the task specification we use the output of the BERT encoder differently.

As we know from our previous discussion about Transformer, the ordering information of tokens will be lost during attention. BERT uses a combination of token embedding (word representation), segment embedding (whether the token belongs to sentence A or B) and position embedding (index of the token position) as the input to the BERT model.

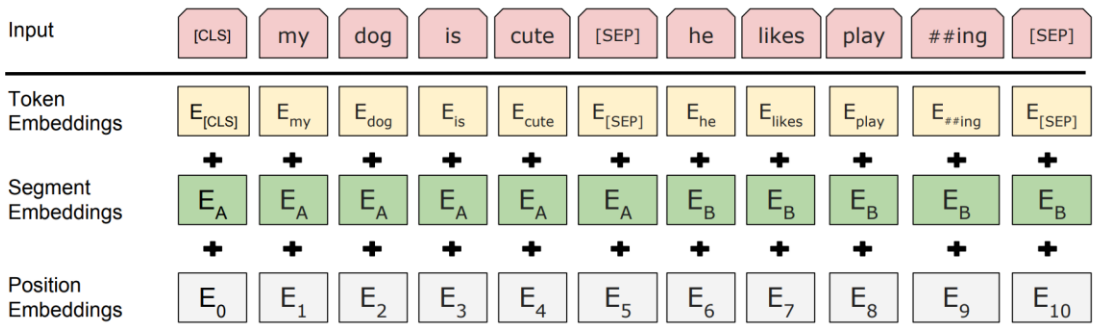


Fig. from [21]. An illustration of how the input embedding is produced.

We can summarize their differences in the following table.

	ELMo	**GPT**	**BERT**
Language modelling mechanism	LSTM	Attention, decoder of Transformer	Attention, encoder of Transformer
Language modelling direction	Bidirectional	Unidirectional	Bidirectional
Downstream task approach	Feature-based, combining with other models	Fine-tuning	Fine-tuning

Table 1. Comparison between ELMo, GPT, BERT.

2.3.4 Tokenization

Tokenization of words is an important procedure to preprocess the input before feeding it into any language model. Since a word can appear in a different form while carrying similar meaning, for example in English the word “walk”, “walks”, “walking” and “walked” all refer to the action of moving around but with different time scenario. In this case, we want the model to understand these words as having a similar meaning, where then we need similar encoding of these words as tokens.

WordPiece [22] is one of the tokenization algorithms, where it initializes its vocabulary starting with every character present in the corpus. By merging two of its vocabulary according to the likelihood of subwords of the corpus, it progressively generates new word units until a predefined limit of word units is reached. This allows the vocabulary to capture all the frequently occurring sub word tokens in the corpus.

2.3.5 GLUE

To allow experiment with a language model, we need a benchmark to compare the performance of different architectures. General Language Understanding Evaluation benchmark [23] (GLUE) is the set 9 of tasks and dataset that combines a diverse range of existing language understanding tasks. At the following paragraphs, we will describe three of the tasks that are of small, medium and large size of corpus respectively.

CoLA is a relatively small dataset consisting of English sentences, designed for the acceptability judgment of the grammatical correctness of the sentence. Matthews correlation coefficient (mcc) is used as the evaluation metric, which evaluates binary classification performance on an unbalanced dataset.

SST-2 is the medium-sized dataset consisting of movie reviews. The corresponding task is to predict the sentiment of the sentence, whether it is positive or negative.

RTE is the large-sized dataset consisting of textual entailment. Binary classification is done on RTE to distinguish the sentence pairs as entailment or not entailment.

The remaining tasks also cover a broad range benchmark on language understanding ability.

MRPC is a corpus of sentence pairs from online news sources and annotated for whether the sentences in the pair are semantically equivalent.

QQP is a collection of question sentence pairs from Quora, a community question-answering website, and the corresponding task is to determine whether the pair is semantically equivalent.

STS-B is a collection of sentence pairs form news headlines and annotated by human with a similarity score from 1 to 5, the corresponding task requires to predict the similarity scores.

MNLI is a crowd-sourced collection of sentence pairs, annotated according to textual entailment of the pairs. Labels are either entailment, contradiction or neutral.

QNLI is a dataset for question-answering training. The dataset consists of question-answer pairs, where the corresponding task is to predict whether the given answer sentence is the correct answer to the question sentence.

WNLI consists of sentence pairs where the first sentence contains a pronoun and the second sentence has the pronoun substituted with other possible referents. The corresponding task is to predict whether the sentence with the pronoun substituted is entailed by the original sentence with the correct substitution.

3 Problem Statement

In our project, we propose that there exist redundancies in the pre-trained BERT model. During fine-tuning, these redundancies are learnt to be omitted and useful connections are learnt to perform the downstream tasks well. To efficiently use the pre-trained model to solve downstream tasks, we would like to use Neural Architecture Search methods to search for the best sub-network architecture of the pre-trained model during the fine-tuning stage of BERT.

We will be using BERT for Natural Language Understanding tasks. These downstream tasks are taken out from the GLUE dataset. The objective of our problem is to find a minimal set of connections in the fine-tuned BERT model that has the minimal performance drop comparing to the original fine-tuned model.

4 Related Works

In the works of this project, we have referenced to several of the following existing methods that work with NAS and network compression of BERT.

4.1 DARTS

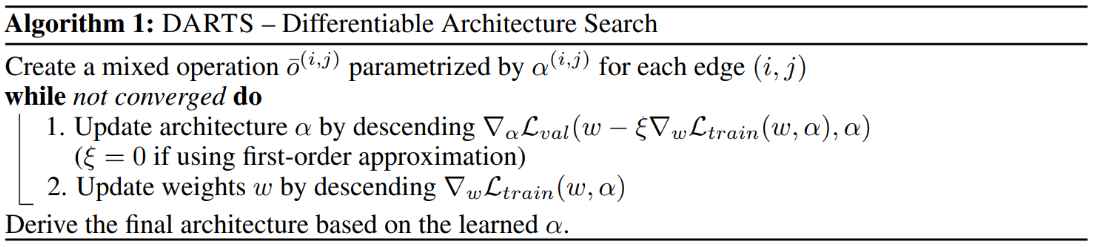
Differentiable architecture search (DARTS) [24] perform its architecture search by formulating a continuous relaxation of the architecture representation. DARTS is a cell-based approach to architecture searching, meaning that it targets to find the best cell architecture and the final network architecture is a stack of the searched cell. For example, when DARTS searches for convolutional cells on CIFAR-10, two types of cells are searched including the normal cell that maintains the same size for the input and output dimensions and the reduction cell that is used to reduce the output dimension to be smaller than the input dimension.

DARTS formulate its searching as a bilevel optimization problem, which uses α as an upper-level architecture variable and w as the lower-level model parameters variable: (by [24, equations (3), (4)])

$$\min_{\alpha} \quad \mathcal{L}_{val}(w^*(\alpha), \alpha) \tag{1}$$

$$\text{s.t.} \quad w^*(\alpha) = \operatorname{argmin}_w \mathcal{L}_{train}(w, \alpha) \tag{2}$$

To solve the above optimization, we need a solution to (2) for solving (1). However, in practice solving (2) requires large amount of computation and it becomes expensive to solve (1). DARTS is the algorithm that approximate the gradient of the target function in (1) without solving (2).



```
Algorithm 1: DARTS – Differentiable Architecture Search
Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge $(i,j)$
while not converged do
    1. Update architecture $\alpha$ by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$
       ($\xi = 0$ if using first-order approximation)
    2. Update weights $w$ by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$
Derive the final architecture based on the learned $\alpha$.
```

Fig. from [24]. Algorithm of DARTS.

From the above definition we see that DARTS iterate between optimizing (1) and (2) alternatively using gradient descent. The overall idea is to use

$w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha)$ as an approximation to $w^*(\alpha)$, the solution of (2). This term is obtained by performing one step of training of the network parameters w under the current α. By this approach, we have an efficient searching algorithm that can approximately solve the bilevel optimization problem stated above.

To model the continuous search space using α, within a cell we learn a set of α that models the probability of each candidate operations on each edge using softmax. The set of candidate operations defines the discrete search space of the architecture, for example convolution, max pooling and zero operation of different dimensions forms a set of candidate operations.

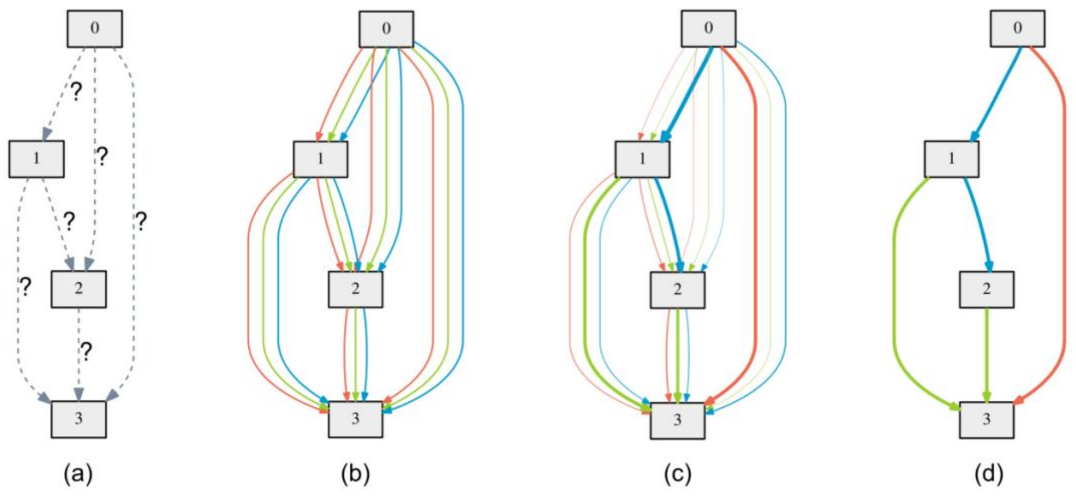


Fig. from [24]. An overview of DARTS. (a) shows that the operation on each edge is initially unknown. (b) shows that a continuous relaxation of the discrete search space is done by allowing a mixture of the operations happening on each edge. (c) by learning the set of α *we can tell which operation is the most important on each edge. (d) the final architecture is determined by the operation of maximal probability.*

4.2 TAS

Transformable architecture search [2] (TAS) is another differentiable architecture searching algorithm, which searches for the best width and depth of the network efficiently. TAS achieve differentiable searching by modelling the probability of choices of architecture as $softmax(\alpha)$. TAS also adopts the idea of sampling the options of architecture at each training step, to avoid traversing all the paths of possible architectures for a more efficient searching [26]. In order for sampling to be differentiable, [26] uses the Gumbel-softmax trick to turn categorical sampling into a differentiable procedure of sampling. The sampled *k*-dimensional vector y is given by the equation ([27, equation (2)])

$$y_i = \frac{\exp\left((\log(\pi_i) + g_i)/\tau\right)}{\sum_{j=1}^{k} \exp\left((\log(\pi_i) + g_j)/\tau\right)}, \qquad \text{for } i = 1, \dots, k$$

, where π is the class probabilities and $g \sim \text{Gumbel}(0, 1)$. This Gumbel-softmax will behave like one-hot sampling when τ approaches 0, and similarly it will behave like uniform sampling when τ approaches ∞. The class probability π is $softmax(\alpha)$ in TAS.

For example, when searching for the width of a convolutional neural network, TAS sampled two architectures of different width in one layer and weight their output according to the class probabilities. To deal with the difference in dimension of the sampled width choices, TAS perform a channel-wise interpolation which transformer the smaller width output to the same size as the larger width output so as to do weight sum of their output. The implementation of channel-wise interpolation can be considered as expanding the smaller dimensional output with the mean values of neighbour dimensions output.

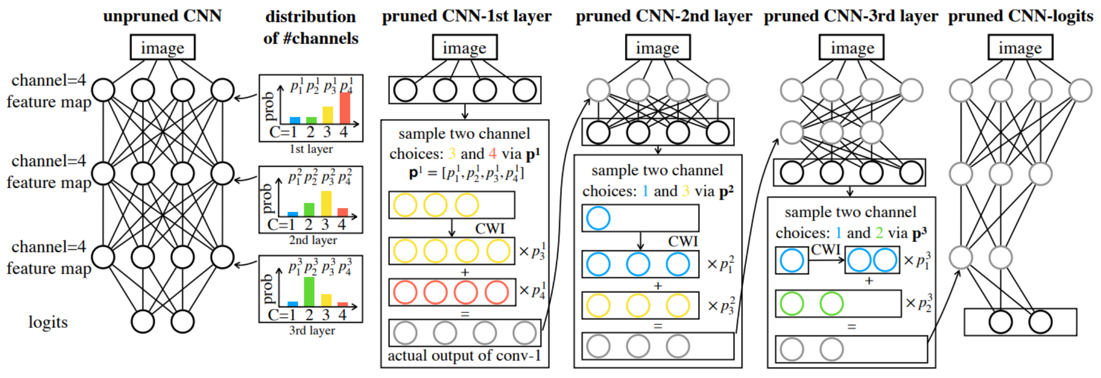


Fig. from [3]. An illustration of the procedure of searching for the width of a convolutional neural network using TAS. At each layer, 2 choices of the number of channels are sampled and the architecture for one forward step is determined after all the sampling.

In TAS, the training set of data is used to train the pruned network's weights and the validation set of data is used to train the architecture parameters α. TAS has combined two searching objectives for α, i.e. the cross-entropy classification loss of the network and the penalty for computation cost. The loss of validation that is used to update α is given by the following equations [25, equations (7),(8)]:

$$\mathcal{L}_{val} = \underbrace{-\log\left(\frac{\exp(z_y)}{\sum_{j=1}^{|z|}\exp(z_j)}\right)}_{\textit{cross-entropy classification loss}} + \underbrace{\lambda_{cost}\mathcal{L}_{cost}}_{\textit{computational cost loss}}$$

$$\mathcal{L}_{cost} = \begin{cases} \log(\mathbb{E}_{cost}(\mathbb{A})) & \text{when } F_{cost}(\mathbb{A}) > (1+t)\times R \\ 0 & \text{when } (1-t)\times R < F_{cost}(\mathbb{A}) < (1+t)\times R \\ -\log(\mathbb{E}_{cost}(\mathbb{A})) & \text{when } F_{cost}(\mathbb{A}) < (1+t)\times R \end{cases}$$

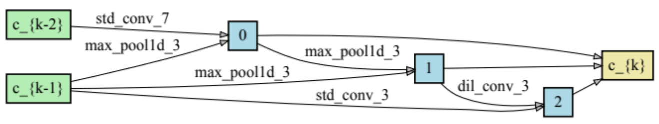
, where $\mathbb{A}$ represents the architecture parameters modelled by α, $\mathbb{E}_{cost}(\mathbb{A})$ is the expected computation cost of the possible architectures and $F_{cost}(\mathbb{A})$ is the actual cost of the sampled architecture. Here we are using the target *R* as a parameter to control the network to converge at having *R* computation cost, and we use $t \in [0,1]$ to model the tolerance of efficiency of the model.

When the optimal architecture is found, TAS performs knowledge distillation from the unpruned network to the searched architecture.

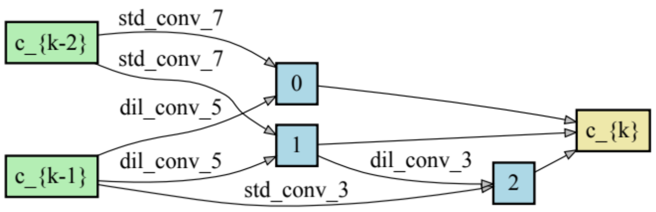
4.3 AdaBERT

AdaBERT [11] inherit the work of DARTS [24] and implements neural architecture search to find a convolutional-based architecture cell that performs similar to a fine-tuned BERT by knowledge distillation. In the search space of AdaBERT operations like convolution, pooling, skip connection and zero operation are possible. As similar to DARTS, each operation is allowed to take two inputs within the cell and provide one output. To achieve knowledge distillation, the searching objective of the architecture is to learn generating the intermediate layer output of the teacher BERT model with the searched architecture. The resulting architecture would be a stack of the searched cells, composed of only the operations in the search space, without attention operation of BERT. AdaBERT uses downstream tasks from GLUE for knowledge distillation and architecture searching.

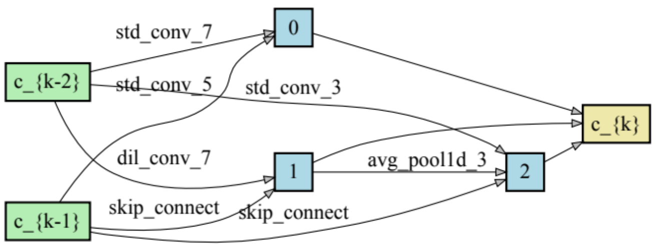
The results of AdaBERT is promising, showing the advantage of inference speedup of the searched architecture and significant compression ratio of the network. This implies the computational efficiency of convolutional operations over attention mechanism.



(a) Sentiment Classification, SST-2



(b) Sentiment Equivalence Classification, MRPC



(c) Entailment Recognition, QNLI

Fig. from [11]. The searched cells for different downstream tasks from GLUE.

4.4 TinyBERT

TinyBERT [28] provides a solid demonstration of network compression by knowledge distillation on the BERT model. BERT as the encoder of the Transformer has several intermediate operations before the output layer. TinyBERT looks into the details of BERT operations and performs knowledge distillation of transformer layers by comparing the attention matrices and the hidden states, while also distillate the embedding layer and the prediction layer of BERT. Attention matrices distillation is motivated by the fact that self-attention of BERT can capture rich linguistic knowledge [29], and that would be important for natural language understanding.

Knowledge distillation is performed first on the pre-trained model, using the large text corpus that is used to train the original general model and this is referred as General Distillation. Knowledge distillation is also performed using the downstream task augmented training set, which is referred as Task-specific Distillation. So, we need to prepare two teacher models for a downstream task, one being the pre-trained general BERT model and one being the fine-tuned BERT model.

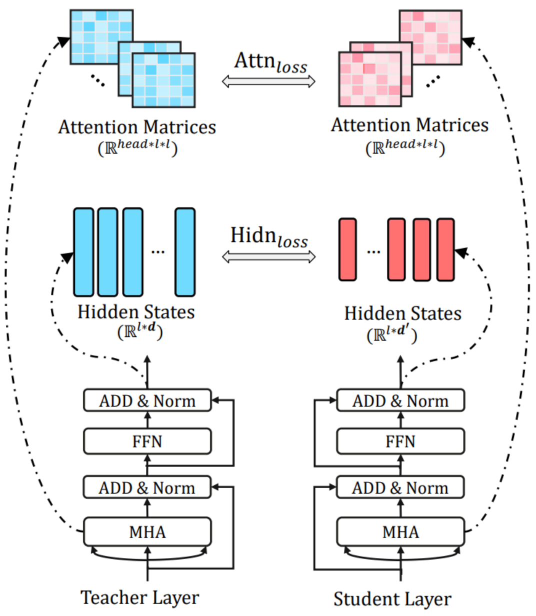


Fig. from [28]. An illustration of how the attention matrices and hidden states are used to perform knowledge distillation.

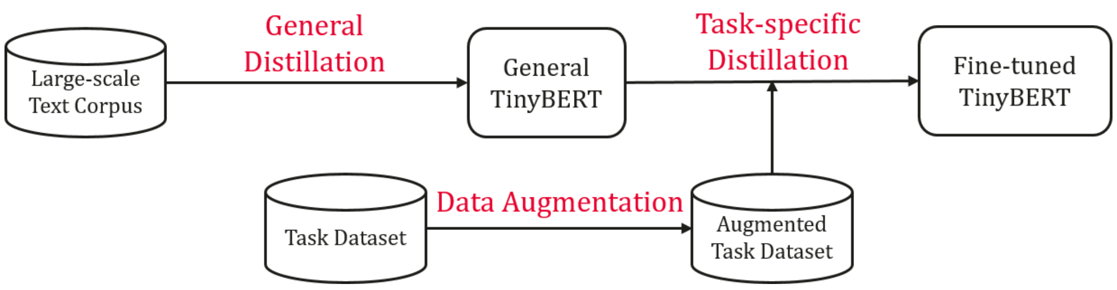


Fig. from [28]. An overview of TinyBERT learning.

Notice that data augmentation in TinyBERT uses the language model BERT and a pre-trained embedding GloVe to generate new training data. Given a sequence of words, we can generate a similar sentence as follows. First, we choose which and how much of the words to replace. If the chosen word is a single-piece word, BERT is used by taking the sentence and mask out the target word to feed it into BERT, where the predictions of BERT is used as candidate words to replace the chosen word. If the chosen word is a multi-piece word, GloVe embedding is used to retrieve the most similar words for replacement.

GloVe is the embedding representation of words where training is performed on aggregated global word-word co-occurrence statistics from a corpus.

4.5 Overview on Pruning NLP Model

By the summary of [36], most of the pruning methods regarding Transformer-like models can be overviewed by the following diagram.

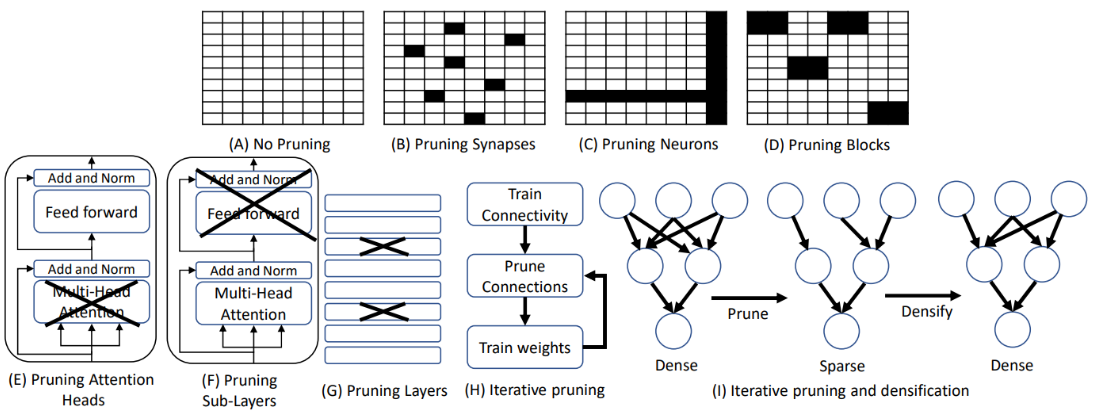


Fig. 2. Different Types of Pruning: (A) represents no pruning. Filled cells represent pruned entries. (B), (H) and (I) are unstructured pruning methods discussed in Section 2.1. Remaining are structured pruning methods: (C) is discussed in Section 2.2, (D) in Section 2.3, (E), (F) and (G) in Section 2.4.

Fig. Illustration of all types of pruning over Transformer network, from [36].

Besides the difference in search space of each pruning method, the rule of determining which weights to be removed also varies. Hessian based method uses a measure of saliency of each weight to determine the importance of a weight. Hessian based methods are usually expensive since second derivatives are computationally expensive to calculate. A more efficient and computationally feasible method is to prune the weight according to the magnitude. Weights with smaller magnitude are considered to be less influential to the prediction output. Another variant of magnitude weight pruning is iterative magnitude pruning where pruning is done gradually during training.

For computationally efficiency on GPU, some methods decide to group the weights into blocks and prune the blocks according to the maximum magnitude within the same block. For a bigger group of weights, we can consider pruning the attention heads. A few possible ways to consider the important of the heads are by gradient information to the attention score of the heads, or computing the average maximum attention weights over tokens in a set of sentences, or by layer-wise relevance propagation.

5 Methodology

5.1 Differentiable Search Method

To facilitate differentiable architecture searching similar to [24] and [3], we need to design how to generate the architecture from the architecture variable α. The following paragraphs will describe how to model α to perform searching on the representation dimensions.

In this method, we use the architecture variable α to generate the mask by sigmoid(α). Since we want to simulate the mask with sigmoid(α), we want α to be outside of the range $[-5, 5]$ so that the mask would contain $\{0, 1\}$ values. We do some trick to make sure that the gradient arriving at α would make a large enough step to jump across -5 and 5, by rescaling the gradient arriving at α and adding limitation to the update of α according to the magnitude of the gradient. Details are referred to Section 7.4.

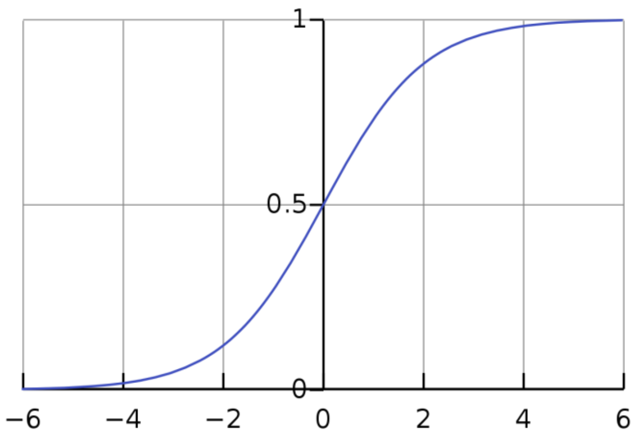


Fig. from Wikipedia. Sigmoid function.

Algorithm: Search Method for Representation Dimension

```
Initialize $\alpha$ to a constant value 5, since $\text{sigmoid}(5) \approx 1$.
Initialize encoder weights $w$ from pre-trained model.
For each forward pass:
    1. Generate mask by sigmoid($\alpha$).
    2. Mask the corresponding representation during forwarding of the encoder.
    3. Backpropagate the cross-entropy loss w.r.t labels and FLOPS loss to learn $\alpha$.
Choose the dimensions according to the activated α, i.e. sigmoid($\alpha$) > 0.99
```

5.2 Search Objective

The objective of our NAS algorithm is to maximize network efficiency. In particular, to model the network efficiency, we propose to calculate the floating point operations per second (FLOPS) to represent the network efficiency. At the further development of the project, we can change the objective to other measures of the network efficiency, such as the network parameter size or the inference time.

To facilitate the minimization of architecture FLOPS, we adopt a similar approach like [3] and formulate the objective loss function as follows:

$$\mathcal{L}_{arch} = \underbrace{\mathrm{MSE}(o_{student}, o_{teacher})}_{\textit{prediction distillation loss}} + \underbrace{\lambda_{cost}\mathcal{L}_{cost}}_{\textit{computation cost loss}} \tag{3}$$

$$\mathcal{L}_{cost} = \begin{cases} \log(\mathbb{E}_{cost}(\mathbb{A})) & \text{when } F_{cost}(\mathbb{A}) > (1+t) \times R \\ 0 & \text{when } (1-t) \times R < F_{cost}(\mathbb{A}) < (1+t) \times R \\ -\log(\mathbb{E}_{cost}(\mathbb{A})) & \text{when } F_{cost}(\mathbb{A}) < (1+t) \times R \end{cases} \tag{4}$$

$\mathcal{L}_{arch}$ represents the loss function value of the architecture variables and is used to train the architecture variables only. $\mathcal{L}_{arch}$ consists of two parts, the first part is the cross-entropy classification loss with respect to the data labels, which trains the architecture to prune away the connections that do not contribute to the task performance. The second part is the weighted $\mathcal{L}_{cost}$, where $\mathcal{L}_{cost}$ models the network efficiency and train the architecture variables to approach a target R within a range of tolerance. The target R is usually determined by a portion of the maximum FLOPS of the architecture, which is the unpruned architecture FLOPS. According to different searching methods we would have a different formulation of $\mathbb{E}_{cost}(\mathbb{A})$, the expected FLOPS of the architecture.

5.3 Search Space

To fully extend the ability to compress the network into any targeted computational cost, i.e. any sub-architecture, we model the searching algorithm in the way that it covers all the computations involving matrix multiplication. While keeping the connection between hidden layers and the choice of activation functions unchanged, our searching algorithm will decide on the dimension reduction of hidden representations. The overall effective search space can be divided into two parts, the multi-heads attention and the feed-forward intermediate dimension.

In BERT model we use multiple attention heads to learn different aspects of similarity among the hidden representation. We expect that after fine-tuning towards the downstream task, not all heads trained are responsible for the prediction task, by evident shown from [31], where attention heads are removed without much of an impact towards the prediction performance of the final model. In our searching algorithm each head is independently assigned with one parameter variable α.

In BERT model we have a large intermediate size in the feed-forward block before each hidden layer output. We can prune the intermediate dimensions of each feed-forward block as long as the output dimension of the first linear layer matches with the input dimension of the second linear layer. In our searching algorithm each dimension in the intermediate representation is independently assigned with one parameter variable α.

Within the BERT architecture, there are several parts that we consider as the candidates to be pruned.

First, we consider reducing the hidden representation size. In the original BERT architecture, a fixed hidden size is used throughout all the layers so that within one layer of the encoder of the Transformer each token is represented with the same hidden size. We investigate bottom-up to see which operations allow reduction of hidden representation size. The following illustrations are adapted from [17].

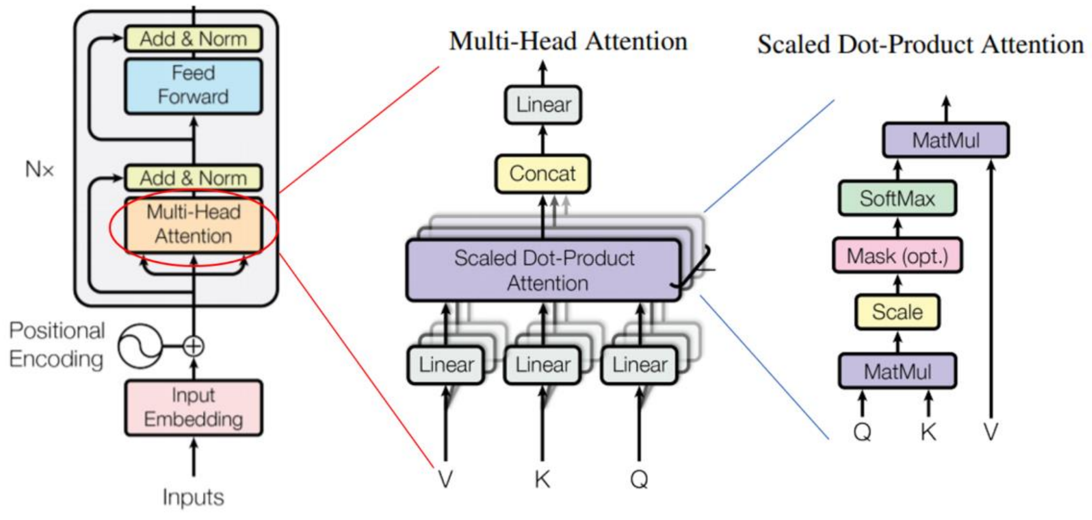


Fig. An overview of a hidden layer of BERT, read from left to right. Red and blue lines represent what are the inner operations of the concerned layer.

On the leftmost, we have a hidden layer of BERT. We suggest that we can search for the hidden representation size of the multi-head attention layer since [31] have shown that there are redundancies among the multiple heads in each layer of BERT. To understand where the hidden size flexibility is, we take a look at the multi-head attention layer and scaled dot-product attention individually.

5.3.1 Input Embedding

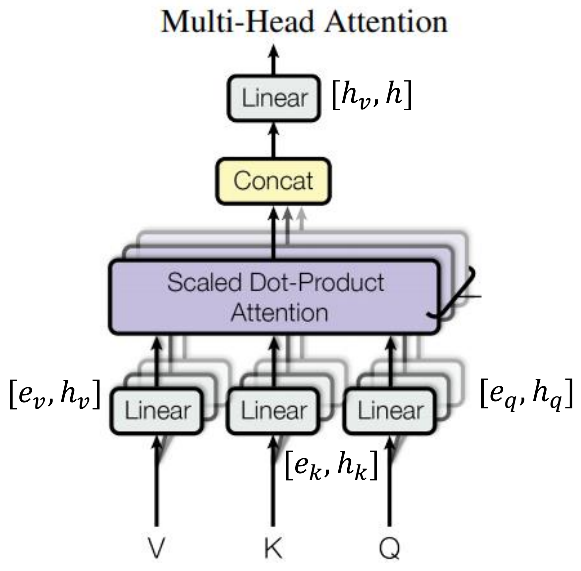


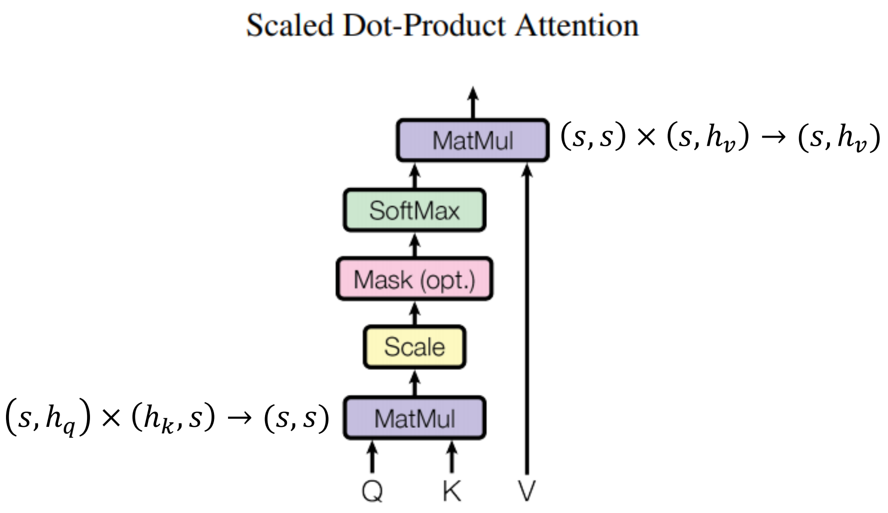
Fig. $[x, y]$ *represents a linear transformation of a vector from x dimensions to y dimensions.*

(e_v, e_k, e_q) In the original setup, the linear transformation before scaled dot-product attention takes the sentence token embedding as inputs. In this case $e_v = e_k = e_q =$ the representation size of a token embedding, which is often referred as the hidden representation size or the hidden size of a BERT model. We can search on how much of the hidden representation is required as input to this linear transformation to gain enough information for specific downstream tasks, and we refer to this as searching on the dimensions of the input embedding.

5.3.2 QKV Hidden Representation

(h_v, h_k, h_q) Constrained by the nature of the matrix multiplication operation within the attention mechanism, we require $h_q = h_k$. By the definition of BERT attention, the output of the attention matrix is precisely ([17], equation (1))

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Scaled Dot-Product Attention
MatMul
$(s,s)\times(s,h_v)\rightarrow(s,h_v)$
SoftMax
Mask (opt.)
Scale
MatMul
$(s,h_q)\times(h_k,s)\rightarrow(s,s)$
Q
K
V

Fig. $(x_1,x_2)\times(y_1,y_2)\rightarrow(o_1,o_2)$ *represents a matrix multiplication* $O = XY$, *where* $X \in \mathbb{R}^{x_1\times x_2}$ *and* $Y \in \mathbb{R}^{y_1\times y_2}$. s *represents the maximum sentence length.*

We can search on the dimensions of h_v, h_k, h_q constrained by $h_q = h_k$. We will refer to this as searching on the qkv hidden representation.

At the top layer of multi-head attention, we have a linear transformation $[h_v, h]$. The output size of this transformation is fixed to be the hidden size of the network because its output will be added with the residual connection from before the attention layer.

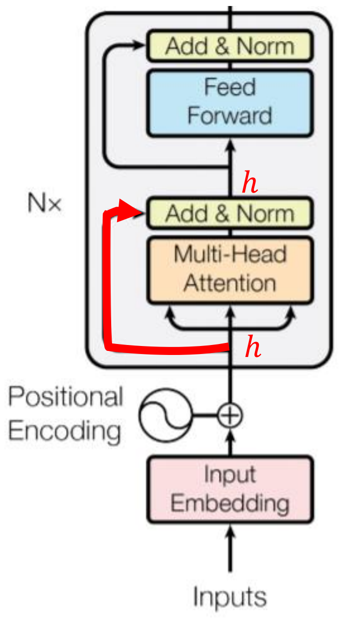
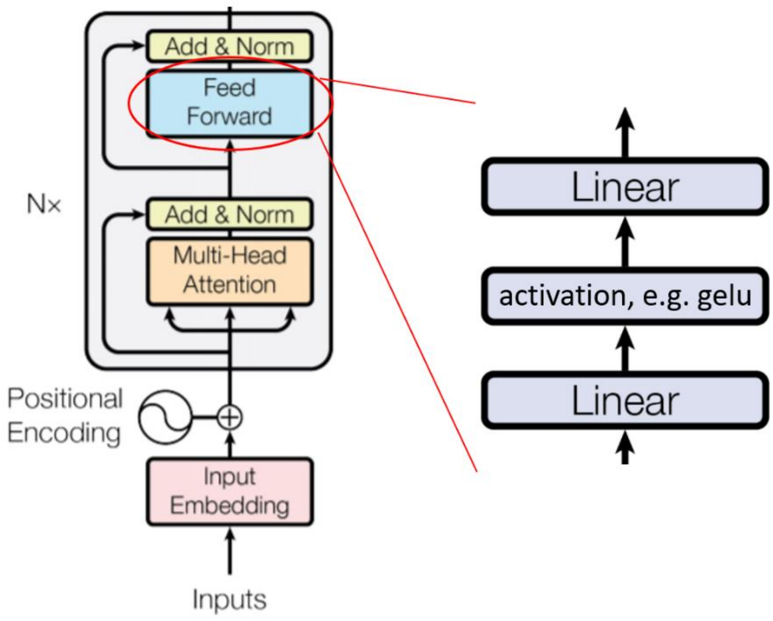


Fig. Red arrow shows the residual connection in concern. The same hidden size h *must be maintained at the output of the multi-head attention layer.*



5.3.3 Feed Forward Intermediate Representation

Fig. An illustration of the feed-forward layer of the encoder of the Transformer.

At the top layer of the encoder, we have a feed-forward layer consists of two linear transformations. We can search for the intermediate representation.

5.3.4 Multi-heads Pruning

Motivated by [31], we would like to reproduce the result of [31] using our differentiable NAS method. It has been shown that across the multiple heads, only several of them at each layer are responsible for the performance on the downstream task. In [31] the experiment covers the setup of manually choosing one head to be removed from the model and manually choosing only one head to remain in each layer. While the combination of important heads is yet to be observed by the experiment. We suggest investigating whether our searching methods could find the suitable combination of heads in each layer.

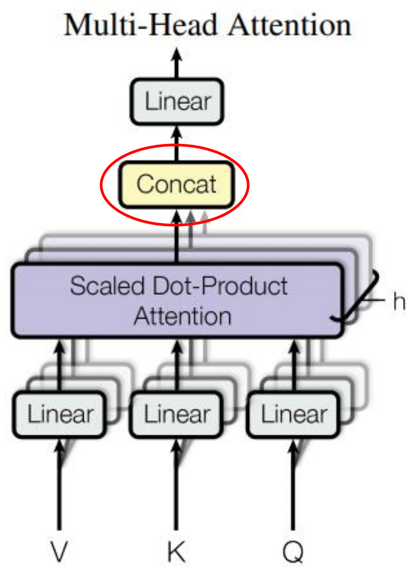


Fig. Observe that the output of each attention head is combined in the above circled layer. Pruning the multi-heads will prune away part of the linear transformation at the top as well.

5.4 Knowledge Distillation - BERT-Base to TinyBERT

5.4.1 Procedure

We follow the procedure below throughout the experiment to obtain the results. Notice that during distillation we perform architecture search at the same time, similarly as the DARTS algorithm (refer to **4.1**). For distillation, we always use fine-tuned bert-base-uncased model of the specific task as the teacher model. We would record the intermediate models only during prediction layer distillation, and only when the model performs better than the previous best model on the evaluation set. Each of the following steps is trained on 10 epochs of the training data.

1. Use 2nd_General_TinyBERT_4L_312D as student model, perform intermediate layer distillation and architecture search.
2. Inherit the resulting model of 1. as the student model, perform prediction layer distillation and architecture search.
3. Inherit the resulting model of 2. as the searched architecture, extract the architecture to initialize 2nd_General_TinyBERT_4L_312D as student model, and perform intermediate layer distillation.
4. Inherit the resulting model of 3. as student model and perform prediction layer distillation.

5. The final resulting model is obtained by the output of step 4.

5.5 Self-Knowledge Distillation

5.5.1 Procedure

Most of the training, including architecture searching and fine-tuning, uses 10 epochs unless specified, especially for larger dataset we will use less epochs. In our experiment the teacher model is always the fine-tuned TinyBERT_4L_312D on the specific downstream task. The searching and fine-tuning procedure is as follows:

1. Use TinyBERT_4L_312D as student model, perform prediction layer distillation and architecture search.
2. Inherit the resulting model of 1. as the searched architecture, extract the architecture to initialize a subnet of TinyBERT_4L_312D as student model, and perform intermediate layer distillation.
3. Inherit the resulting model of 3. as student model and perform prediction layer distillation.
4. The final resulting model is obtained by the output of step 3.

5.6 Search Scheduling and Parameter Control

In our approach to architecture searching, we use a moving architecture target size (R in equation (4)) in the search objective so that the architecture converges to the target architecture size slowly. The scheduling of the architecture target size is determined by the target architecture size R^* and the number of training step s, where $s = \left\lceil \frac{\#\, training\ samples}{batch\ size} \times \#epoch \right\rceil$ and

$$R^{(i)} = \frac{1 - R^*}{s} \times i \tag{5}$$

Using equation (5) we can determine the architecture target size at the i-th training step. The intuition behind is that the scheduling is done linearly so that $R^{(i)}$ decreases linearly towards R^*.

To avoid early convergence of the architecture, we limit the update of gradient to the architecture parameters in the way that at the backpropagation step, only a certain ratio of gradients can be propagated to the architecture parameter, while the rest of the updates are not allowed in the same backpropagation step. The best ratio depends on the architecture size. We evenly distribute the allowed updates over each search region so that the resulting architecture is not biased on pruning any of the search region because of this parameter control.

5.7 Conclusion

As a wrap up of all the methodology, our final approach towards architecture searching relies on searching the subnetwork by self-prediction distillation. For prediction performance concern it is best that we do not change the connection of the input embedding dimensions and the QKV hidden dimensions alone, but instead searching for the pruning by grouping them as individual heads and prune the connection of heads. Aside from the pruning of heads we also prune the intermediate dimension of the feed-forward layer in each BERT layer for the sake of large computation reduction.

6 Experiment

By implementing our method on PyTorch we managed to experiment over the searching algorithm in hopes of understanding the relationship between the search space, search objective, data augmentation and prediction performance.

By configuring different search space and different composition of architecture parameter it provides different level of freedom for the searching algorithm. We investigate the relationship between constrained search and less constrained search. Using different search objectives like distillation and prediction loss by cross entropy will lead to different architecture. We would like to understand which objectives is most suitable in the context of architecture search.

Motivated by the result from [33] where experiments show that retraining the network after pruning would help upholding prediction performance, after our searching algorithm we fine-tune the searched architecture before evaluating the final model.

Limited by the resources we have, and the fact that BERT model and other NLP deep learning models are expensive to train [35], we would mainly inherit the result from TinyBERT [28] where it provides a BERT model of smaller size for experimentation.

The major contribution of this project is that we suggest pruning only the multi-head attention and the intermediate dimensions of the feed-forward layer, while keeping the hidden embedding unpruned for performance consideration. Also, we proposed an iterative learning procedure to learn the subnetwork by distillation.

6.1 Knowledge Distillation - BERT-Base to TinyBERT

6.1.1 Reproducing TinyBERT & Setup

In our experiment we inherit the setup and results of TinyBERT and extend the project by applying NAS for network pruning during the fine-tuning stage of TinyBERT. We will expand the code for task-specific distillation in TinyBERT and add new functionality to perform pruning on the representation of input embedding, qkv hidden representation, feed-forward intermediate representation and the multi-head attentions of each layer.

For task-specific distillation, we need a teacher model that is fine-tuned on specific downstream task. In our setup, we use bert-base-uncased pre-trained model from HuggingFace's implementation [30] and perform fine-tuning on all the tasks on GLUE to obtain the teacher models for task-specific distillations. Bert-base-uncased is the implementation of BERT that has 12 hidden layers, 768 hidden representation size, performing 12 heads attention, 3072 feedforward size and has 110 million parameters. Bert-base-uncased is pre-trained on all lower-case English corpus. We will be using the teacher models as baseline performance of any fine-tuned models. In our experiment we will focus on three tasks from GLUE, each represents a small, medium and large-sized dataset respectively. To obtain a fine-tuned BERT model, we trained the pre-trained model for the following three tasks for 10 epochs, 32 batch size, 5e-5 initial learning rate for Adam.

	CoLA (mcc)	**RTE (accuracy)**	**SST-2 (accuracy)**
reproduced performance (10 epochs)	0.572	0.708	0.921
reported performance [21] (3 epochs)	**0.521**	**0.664**	**0.935**

Table 2. Showing evaluation results of fine-tuned bert-base-uncased on GLUE tasks.

In TinyBERT two versions of the final distilled model are available, 4layer-312dim represents the smaller version that has 4 hidden layers, 312 hidden size, 1200 feedforward size and 12 attention heads in a layer. 6layer-768dim represents the larger version that has 6 hidden layers, 768 hidden size, 3072 feedforward size and 12 attention heads.

We follow the data augmentation procedure given by TinyBERT using GloVe embedding and the pre-trained bert-base-uncased language model to generate augmented data. Fine-tuning of TinyBERT model is done on the augmented dataset. To obtain a fine-tuned TinyBERT model, we perform knowledge distillation from a fine-tuned bert-base-uncased model to both general pre-trained 4layer-312dim and 6layer-768dim.

	CoLA (mcc)	**RTE (accuracy)**	**SST-2 (accuracy)**
reproduced 4layer-312dim TinyBERT performance (10, 10)	0.426	0.667	0.917
reported 4layer-312dim TinyBERT performance [28] (20, 3)	**0.441**	**0.666**	**0.926**
reproduced 6layer-768dim TinyBERT (10, 10)	0.556	0.714	0.926
reported 6layer-768dim TinyBERT performance [28] (20, 3)	**0.511**	**0.700**	**0.931**

Table 3. Showing evaluation results of distilled TinyBERT on GLUE tasks. Brackets at the end of first column (x,y) represent the model spent x epochs of training for intermediate layer distillation and spent y epochs of training for prediction layer distillation.

From the above tables, it shows that our reproduced models perform similar to the reported behavior.

6.2 Self-Knowledge Distillation

By experiment we concludes that self-knowledge distillation is better than knowledge distillation from large model to smaller model, for the reason described in Section 7.2.5. In this section we focus on describing the second experiment setup, which at the end becomes our main result of this project.

6.2.1 Experiment Setup

In this experiment setup we require a lot less than the previous setup. We only need a fine-tuned model in any architecture size and apply self-distillation throughout the searching process. In this project we focus on using TinyBERT-4L as the architecture we interested in. A fine-tuned model of TinyBERT-4L on the GLUE dataset are open to public so it is easy to get both the student and the teacher model ready, and no fine-tuning to downstream task from general pre-trained model is required in this setup.

To facilitate the searching algorithm with stability and high performance, we design specific learning tricks to help with architecture searching. We control the amount of architecture parameters updated at each training step. We also control the target architecture size so that the algorithm converges slowly and avoid early convergence. Throughout most of the experiment result we update 10% of the architecture parameter and set the final target architecture size as one of {10%, 30%, 50%, 70%} of the original FLOPS.

6.3 Experiment Results

For all the results we report in this section, each task measures the prediction performance using the metric as specified here. Matthew's Correlation Coefficient is used for CoLA, F1 score is used for MRPC and QQP, both Pearson-Spearman correlations are used for STS-B, and the remaining tasks use accuracy as metric.

6.3.1 Development Evaluation

In order to balance between computational efficiency and prediction performance, we analysis the tradeoff between them and find the best configuration which satisfies both of two requirements.

We study the tradeoff in the setup where we do not use data augmentation during fine-tune due to the limited GPU resources for this project. We observe that the best architecture size in consideration of prediction performance is at around 30% of the original architecture FLOPS, since beyond 30% the model would suffer from a larger drop of prediction performance.

Tasks \ FLOPS	10%	30%	50%	70%	100%
CoLA	36.2 (-27.2%)	42.7 (-14.1%)	43.6 (-12.3%)	46.6 (-6.2%)	49.7
MRPC	86.6 (-4.0%)	89.2 (-1.1%)	88.5 (-1.9%)	89.3 (-1.0%)	90.2
STS-B	82.1/82.1 (-5.1%/-4.9%)	84.1/83.9 (-2.8%/-2.8%)	85.2/85.1 (-1.5%/-1.4%)	86.0/85.8 (-0.6%/-0.6%)	86.5/86.3

Table 4. GLUE dev set result. Comparison between different size of searched architecture. Percentage change in performance is calculated with respect to the original architecture performance, i.e. 100% column in table 4.

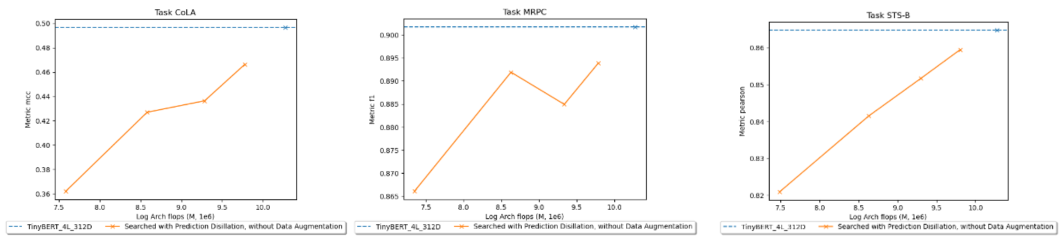


Fig. Visualization of GLUE dev set result. Blue dashed horizontal line represents the performance of the original architecture.

We also created an architecture search baseline by randomly initializing an architecture of the specific sizes of FLOPS. The comparison will show an advantage of our architecture searching algorithm above the random heuristic approach.

FLOPS / Tasks	10%		30%		50%		70%	
	NAS	Random	NAS	Random	NAS	Random	NAS	Random
CoLA	36.2	21.5	42.7	32.4	43.6	41.5	46.6	46.9
MRPC	86.6	78.9	89.2	78.4	88.5	81.8	89.3	90.0
STS-B	82.1/82.1	17.3/17.5	84.1/83.9	42.9/41.2	85.2/85.1	70.6/69.9	86.0/85.8	70.9/70.3

Table 5. GLUE dev set result by our searching method and random architecture.

6.3.2 Test Evaluation

Naturally we would compare the searched architecture with the original architecture in term of prediction performance. With only 30% of the original computational cost we achieved a fast inference model with minor performance accuracy drop in many of the tasks in GLUE. FLOPS in the searched architecture rows are presented in a range, representing a set of searched architecture with similar FLOPS within this range.

To obtain the best result but limited by GPU resources we have, we fine-tuned architecture for CoLA, STS-B, MRPC and RTE on augmented data while fine-tuned the rest without data augmentation.

Models	FLOPS (B)	Speedup	MNLI (-m/-mm)	QQP	QNLI	SST-2	CoLA	STS-B (Pear/Spea)	MRPC	RTE
TinyBERT-4L [28]	1.239	1.0x	82.5/81.8	71.3	87.7	92.6	44.1	-/80.4	86.4	66.6
30% TinyBERT-4L (Our main result)	[0.375, 0.403]	[3.0x, 3.3x]	80.8/80.4	70.9	84.4	91.8	40.7	79.9/78.6	85.4	61.4
Percentage Change in Accuracy	/	/	-2.06%/-1.71%	-0.56%	-3.76%	-0.86%	-7.71%	-2.24%	-1.16%	-7.81%

Table 6. GLUE test set result scored by GLUE evaluation server. Comparison between searched 4-layer model and the original model.

Models	FLOPS (B)	Speedup	MNLI (-m/-mm)	QQP	QNLI	SST-2	CoLA	STS-B (Pear/Spea)	MRPC	RTE
TinyBERT-6L [28]	11.100	1.0x	84.6/83.2	71.6	90.4	93.1	51.1	-/83.7	87.3	70.0
30% TinyBERT-6L (Our main result)	[3.348, 3.590]	[3.1x, 3.3x]	83.7/83.0	71.8	89.5	93.0	46.1	84.2/83.3	86.9	63.6
Percentage Change in Accuracy	/	/	-1.06%/-0.24%	+0.28%	-1.00%	-0.11%	-9.78%	-0.48%	-0.46%	-9.14%

Table 7. GLUE test set result scored by GLUE evaluation server. Comparison between searched 6-layer model and the original model.

Models	FLOPS (B)	Speedup	MNLI (-m/-mm)	QQP	QNLI	SST-2	CoLA	STS-B (Pear/Spea)	MRPC	RTE
Reproduced Bertbase-12L [28]	22.199	1.0x	84.0/83.2	70.7	91.1	92.7	55.3	84.2/82.5	86.6	65.5
30% Bertbase-12L (Our main result)	[6.697, 9.471]	[2.3x, 3.3x]	83.6/83.1	71.6	91.1	91.9	49.9	82.9/81.5	86.5	65.2
Percentage Change in Accuracy	/	/	-0.48%/-0.12%	1.27%	0.00%	-0.86%	-9.76%	-1.54%/-1.21%	-0.12%	-0.46%

Table 8. GLUE test set result scored by GLUE evaluation server. Comparison between searched 6-layer model and the original model.

6.3.3 Baseline Comparison

Besides comparing with the original large model, we also compare with existing small model of BERT architecture and we achieved overall better performance than the existing BERT model produced by [25].

Models	FLOPS (B)	MNLI (-m/-mm)	QQP	QNLI	SST-2	CoLA	STS-B (Pear/Spea)	MRPC	RTE
BERT-Mini [25]	0.873	74.8/74.3	66.4	84.1	85.9	0.0	75.4/73.3	81.1	57.9
30% TinyBERT-4L (Our main result)	[0.375, 0.403]	80.8/80.4	70.9	84.4	91.8	40.7	79.9/78.6	85.4	61.4
Percentage Change in Accuracy	/	+8.02%/+8.21%	+6.78%	+0.36%	+6.87%	+inf	+5.97%	+5.30%	+6.04%

Table 9. GLUE test set result scored by GLUE evaluation server. Comparison between searched model and existing baseline models.

Another baseline result provided by [37], which prunes BERT architecture via iterative magnitude pruning. We show that we achieved overall better performance than unstructured pruning.

Dataset	MNLI (-m)	QQP	QNLI	SST-2	CoLA	STS-B (Pearson)	MRPC	RTE	Avg
Ratio	28.3%	28.3%	39.9%	28.6%	28.9%	31.5%	29.2%	28.2%	
30% Bertbase-12L	84.2	90.3	91.7	92.3	58.3	88.8	85.3	69.7	82.6
Ratio	30%	10%	30%	40%	50%	50%	50%	40%	
IMP [37]	82.6	90.0	88.9	91.9	53.8	88.2	84.9	66.0	80.8
Difference	+1.6	+0.3	+2.8	+0.4	+4.5	+0.6	+0.4	+3.7	+1.8

Table 10. GLUE dev set result comparison between our result and Iterative Magnitude Pruning (IMP) [37]. Difference highlighted in green are out-performing architecture with smaller size.

7 Analysis & Discussion

7.1 Verification of Methodology on MNIST

To analysis our architecture searching method (Section 5.1) and verify that the searching method can correctly choose dimensions of hidden representation that is useful for prediction, we have setup a simpler learning problem on MNIST dataset. The neural network we use is a simple feed-forward classifier composed with 3 linear layers and ReLU activation in between. We design the architecture parameter α at the input dimension of the first layer such that it allows the architecture parameter to choose which dimension of the input image is kept. In order to show the significance of the effectiveness of our searching method, aside from the images of MNIST we also feed noise data from gaussian distribution as input to the first layer.

Here is the detailed description of the model we interested in.

```
Sequential(
    (0): Linear(in_features=1568, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=10, bias=True)
    (5): LogSoftmax(dim=1)
)
```

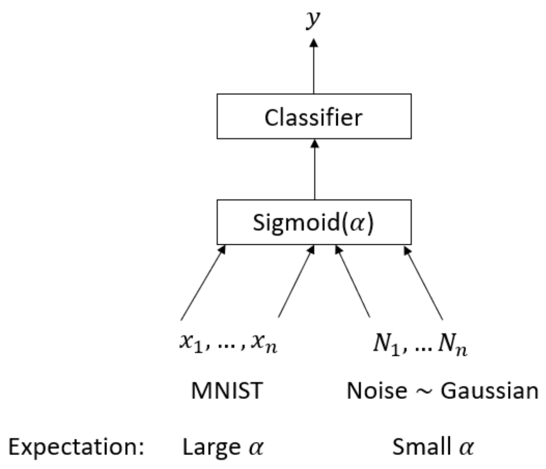


Fig. Design diagram of the learning problem. x_i represents the image data from MNIST while N_i represents noise. We expect the architecture parameter to prune all the noise input and only keep dimensions of the MNIST images.

We randomly initialize the weight of the linear layers. Training of architecture parameters and linear layers' parameters is done simultaneously. 15 epochs of the training data are used.

Observe that this model takes in 1568-dimensional input, where the first 784 of them refers to the MNIST images and the later 784 of them refers to the noise.

Accuracy	**Target Input Size Ratio**	**Resulting Input Size Ratio**	**Result Split**
0.758	0.01	0.012	[20, 0]
0.937	0.04	0.040	[63, 0]
0.952	0.05	0.050	[78, 0]
0.970	0.10	0.100	[154, 3]
0.976	0.30	0.265	[336, 79]
0.977	0.50	0.452	[453, 255]
0.975	0.75	0.703	[588, 514]
0.976	1.0	0.951	[726, 765]

Table 8. Searching results on validation set of the 3-layer model. Highlighted row represents the breakpoint of accuracy drop. Result split column shows the number of activated dimensions at the first half and the second half of the input.

By observing the results at the column of result split, we see that our searching algorithm successfully removed the dimensions containing noisy data input, as the target input size reduces eventually the searching algorithm converges to architecture where the majority of input is the image data while pruning away the noise.

At the same time, we would also like to investigate the performance of our searching algorithm on deeper neural network. We decided to run our searching algorithm on a similar neural network, but configurated with more hidden layers. Here are the details of the 11-layer version.

```
Sequential(
    (0): Linear(in_features=1568, out_features=119, bias=True)
    (1): ReLU()
    (2): Linear(in_features=119, out_features=95, bias=True)
    (3): ReLU()
    (4): Linear(in_features=95, out_features=76, bias=True)
    (5): ReLU()
    (6): Linear(in_features=76, out_features=61, bias=True)
    (7): ReLU()
    (8): Linear(in_features=61, out_features=48, bias=True)
    (9): ReLU()
    (10): Linear(in_features=48, out_features=39, bias=True)
    (11): ReLU()
    (12): Linear(in_features=39, out_features=31, bias=True)
    (13): ReLU()
    (14): Linear(in_features=31, out_features=25, bias=True)
    (15): ReLU()
    (16): Linear(in_features=25, out_features=20, bias=True)
    (17): ReLU()
    (18): Linear(in_features=20, out_features=16, bias=True)
    (19): ReLU()
    (20): Linear(in_features=16, out_features=10, bias=True)
    (21): LogSoftmax(dim=1)
)
```

The same experiment is ran on choosing the dimensions of the input. 20 epochs of the training samples are used for training.

Accuracy	Target Input Size Ratio	Resulting Input Size Ratio	Result Split
0.113	0.01	0.010	[16, 0]
0.794	0.04	0.036	[51, 7]
0.834	0.05	0.046	[69, 4]
0.916	0.10	0.100	[125, 32]
0.946	0.30	0.262	[288, 124]
0.960	0.50	0.456	[429, 287]
0.964	0.75	0.701	[539, 561]
0.940	1.0	0.963	[732, 778]

Table 10. Searching results on validation set of the 11-layer model.

We observe that our searching algorithm performs worse on deeper model, by comparing the highlighted row between the two tables we see that on deeper model more noise dimensions are kept which is harmful to the prediction performance. At the same time, we also observe that a deeper model does not perform better on the validation set, suggesting that the model overfits with the training data.

As a conclusion we know that our searching algorithm works well on models with small numbers of hidden layer, while not as good when dealing with deeper neural network.

7.2 Search Space

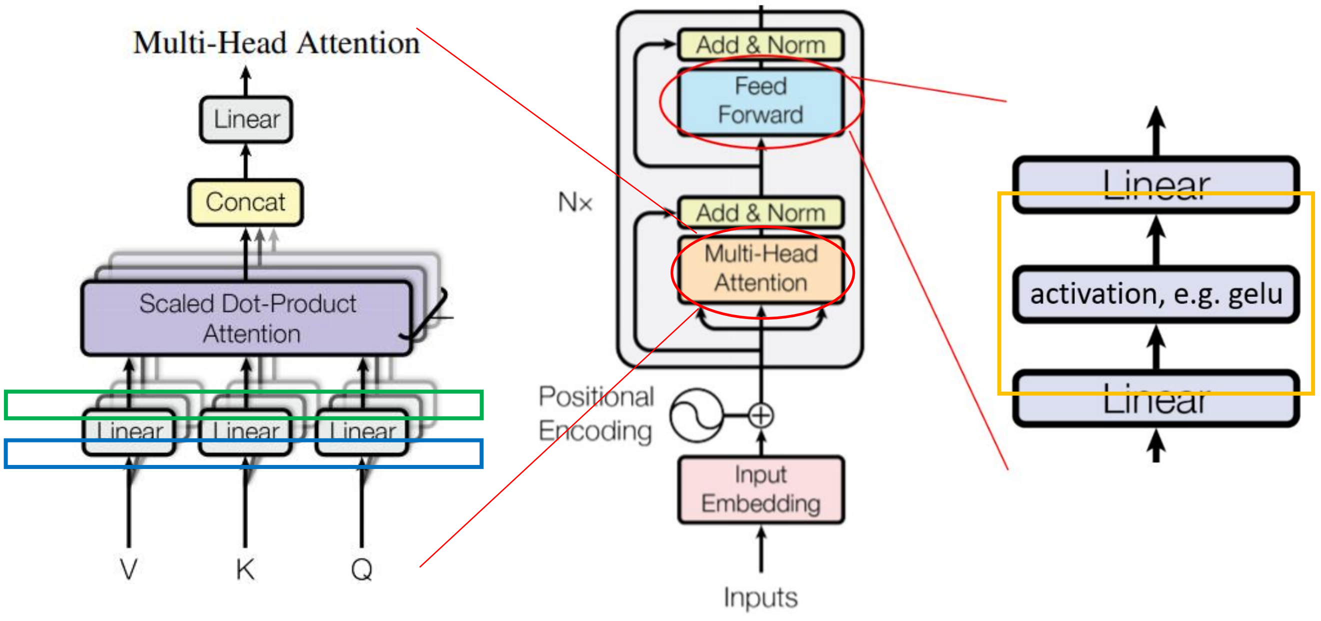


Fig. Annotated parts of the search space.

Throughout this project we give names to the specific part of the architecture for discussion. The blue part, which is the input dimension of each BERT layer, will be referred as the embedding input (embedding). The green part, which is the output dimension of the linear layer that learns representation of QKV vectors, will be referred as the QKV hidden dimension (QKV). The orange part, which is the intermediate dimension of the feed-forward layer, will be referred as the feed-forward intermediate dimension (FF).

To understand the search space, we first analysis the distribution of FLOPS over the BERT architecture. We use TinyBERT 4L as the architecture we discuss over.

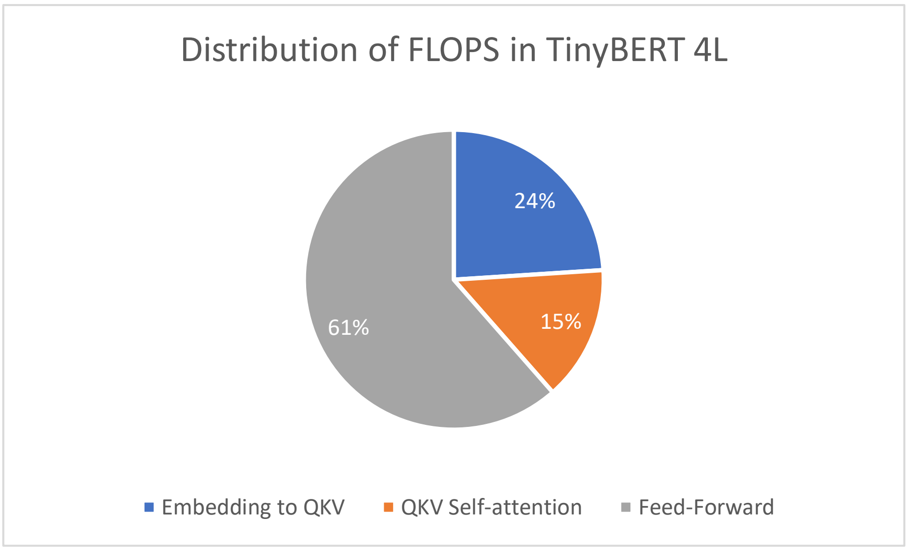


Fig. Distribution of FLOPS in TinyBERT 4L.

Notice that Feed-Forward layers contributes to most of the FLOPS in the network, due to that fact that BERT model uses two linear layers with large intermediate dimension (1024 in TinyBERT 4L) as the feed-forward layer. This suggests that network compression is most efficient if we try to prune the intermediate connection within the feed-forward layer.

To cover all computations within the architecture, we propose two methods for the NAS algorithm to manipulate with the search region. First we can give freedom to the NAS algorithm on deciding the size of each dimension of the searchable embedding, QKV and FF. On the other hand, we can restrict the freedom of the NAS algorithm and consider the region covered by QKV as multiple heads, and each head is pruned as a whole, instead of being reduced in dimensionality.

7.2.1 Input Embedding Pruning

By experiment we have shown evidence that the input embedding dimension is crucial towards the model performance. Intuitively the input embedding serves as the input signal to the layer. Partial information given to each layer would be disadvantageous compared to a fully informed layer.

Experiments shows that keeping all the input embedding not pruned will allow the NAS algorithm to arrive at a better architecture.

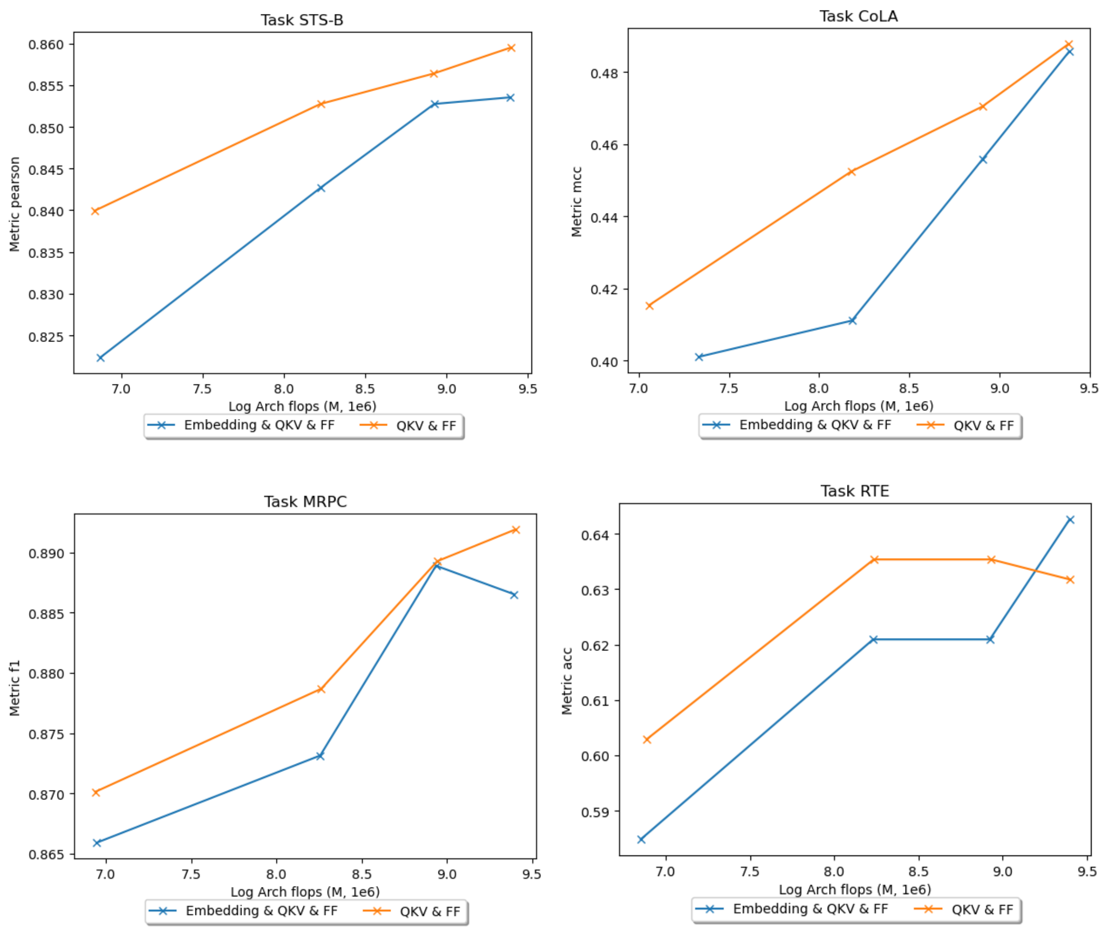


Fig. Experiment results on removing embedding dimension search.

From the above figures, we can see that searching only on QKV and FF outperforms searching that includes embedding. It suggests that all the dimensions of the input embedding are informative in performing the downstream task. There is little redundancy in the input embedding.

7.2.2 QKV Dimension Pruning / Multi-Head Pruning

Essentially QKV dimension pruning and multi-head pruning covers the same ground of FLOPS contribution in the model. They only differ in the configuration of the architecture variables alpha. In QKV dimension pruning each dimension in the query, key and value vectors is assigned with one variable. In multi-head pruning each head is assigned with one variable. We can see that QKV dimension pruning has more flexibility and freedom when compared with multi-head pruning. When we study the difference between these two approaches, we are also understanding whether this architecture searching algorithm flavours a more constrained or less constrained architecture variables. In multi-head pruning you can think of the constraints as requiring the QKV dimension alphas to share the same value if they are within the same head.

We can inspect the effect of computation efficiency gain after pruning the QKV hidden dimension. We shall see that pruning part of the dimensions of each query, key, value vectors would decrease the number of computations required during self-attention. However, when we compare QKV dimension pruning with multi-head pruning, we can realize that the result of multi-head pruning is more efficient than QKV pruning. As long as certain QKV dimension remains active in the searched architecture, we might end up doing matrix multiplication in lower dimension and the number of matrix multiplication carried out by the self-attention layer remains unchanged. On the other hand, architectures found by multi-head pruning directly reduce the number of matrix multiplication, since each head represents at least one matrix multiplication when calculating attentions. Even on the same value of FLOPS in the searched architecture, the final model that is produced through multi-head pruning will be more efficient on GPU than the one that is produced through QKV dimension pruning.

On the aspect of empirical error, through experiment we have shown that the two pruning methods show similar strengths and performance varies over different dataset. We could not conclude which one is better. Further studies can be done on the comparison between

For these two reasons we have decided to adopt multi-head pruning as our final strategy.

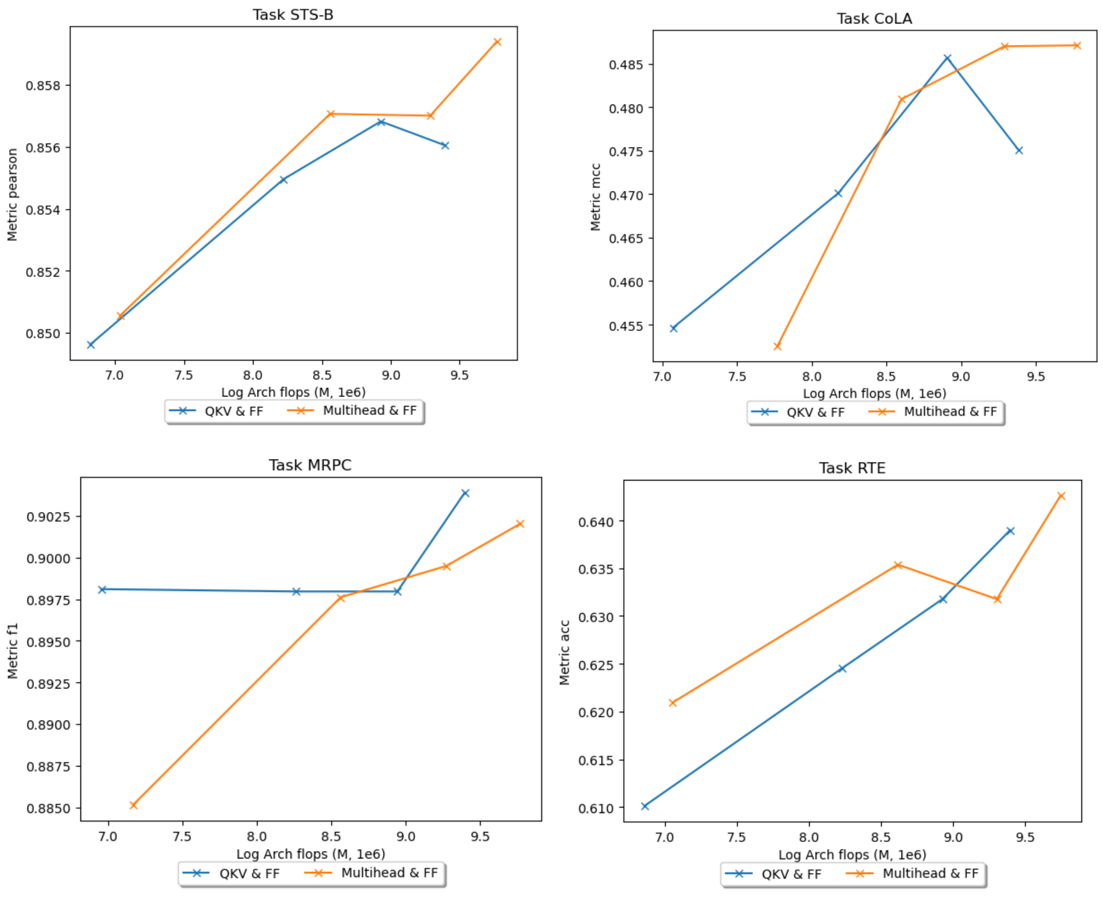


Fig. Experiment results on comparing QKV pruning and multi-head pruning.

7.2.3 Feed-Forward Intermediate Dimension Pruning

This search space contributes most of the computation because in BERT model we use a large intermediate hidden size between the two linear layers that form the feed-forward layer. For example in TinyBERT 4L, the intermediate hidden size of the two linear layers is almost four times the hidden size of the model. Reducing on the intermediate size can greatly affect the computational cost since we would end up doing matrix multiplication that is smaller in dimension, and especially the intermediate size affect both two linear layers.

We do not consider removing this region from our search region because it contributes to 61% of the total FLOPS in the original architecture, referring to figure in Section 6.1.

7.3 Search Objective

In our searching algorithm we use gradient descent learning to search for the architecture parameter that minimizes the search objective. We try to mix different loss functions to see which helps the searching algorithm to find an architecture that performs well on the downstream task. In the rest of this section we introduce a several kinds of possible searching objective and compare the results.

7.3.1 Empirical Error Loss via Cross Entropy

Since predicting the classification label is our final goal of the model generated by the searching algorithm, we want to incorporate classification learning during architecture searching as well. By adding cross entropy loss according to the model classification layer and the data label we allow the searching algorithm to look for an architecture that have the ability to classify the data.

7.3.2 Two Stage Distillation

In the setup of TinyBERT, [28] uses two stage distillation in their compression algorithm, i.e. knowledge distillation is separated into two parts. First we transfer the knowledge of the intermediate output of the layers of the teacher model to the specific layer of the student model. Next we transfer the knowledge prediction output logits of the teacher model to the prediction layer of the student model.

To apply two stage distillation in our architecture search algorithm, we use distillation loss as the search objective of the architecture parameters. This allows the algorithm to learn an architecture that performs similar to the original model. The original fine-tuned model, i.e. a model that is already trained on certain downstream task, serves as a teacher model. At the same time we use the same model as the student model. During the searching procedure, the architecture of the student model keeps shrinking, and thus to keep the performance of the student model we use distillation loss to learn a sub-architecture that minimizes the damage done to the student model. The network parameters are also fine-tuned in the process of searching, towards the same distillation loss.

7.3.3 Intermediate Distillation Loss

The objective loss is calculated according to two parts, the mean squared error between the student's outputted attention matrix and the teacher's outputted attention matrix, and the mean squared error between the student's outputted layer-wise hidden representation and the teacher hidden representation. The learning target of the student layer is chosen in the way that we are evenly picking up the hidden representation of the teacher layers, because the teacher model has more hidden layers than the student model.

By experiment we have shown that using only intermediate distillation loss as the searching target cannot learn an architecture that performs well.

7.3.4 Prediction Distillation

Prediction distillation allows the student model to focus on outputting the prediction of the teacher model. This has a similar effect with empirical error loss by cross entropy but differ in the way that the teacher model prediction output contains the prediction distribution which prevails extra information that the teacher model had learnt.

By experiment we know that using only prediction distillation loss as the searching target is comparable with two stage distillation and empirical error loss, which suggest that the major contribution in two stage distillation is by prediction distillation, i.e. the second stage.

7.3.5 Why Self-Distillation?

Before comparing the effectiveness between knowledge distillation from larger model to smaller model and self-distillation for searching, we state the difficulty of evaluating their effectiveness in the following sense. Since we are only able to access the fine-tuned models of TinyBERT, we do not have the access to the teacher model that they used for knowledge distillation. If we continue architecture searching using another teacher model that we reproduce (e.g. Bert-base-uncased by Hugging), we will end up doing incorrect knowledge distillation because the student model will experience inconsistent teacher knowledge from its previous fine-tuning stages during searching.

On the other hand, self-distillation avoids this issue because we can easily inference the hidden representation of the original student model as the teacher knowledge. This also makes our method more applicable in many scenarios, as long as we have a fine-tuned model ready to be pruned.

By experiments we know that architecture found by self-distillation is better than architecture found by knowledge distillation with inconsistent teacher model.

7.3.6 Conclusion of Search Objective

At this point we can acknowledge the importance of prediction distillation, i.e. to transfer the ability of prediction to student sub-architecture. This is more effective compared to cross entropy loss by the advantage that knowledge distillation can bring. However, we cannot conclude whether self-distillation is better than traditional knowledge distillation or not and this is a potential future direction of study.

7.4 Target Search Size Scheduling

We observe that the learning curve of alpha architecture parameters has an influence on the performance of the resulting architecture. A smooth learning curve towards a smaller architecture is better than a non-smooth learning curve which directly arrives at the desired architecture size. The insight comes from the observation that without scheduling over the target size, our searching algorithm can easily and quickly converge to the target size, without considering all of the data samples. This is a sign of the algorithm converging to a local minimum of the best architecture by just considering one batch of samples, while the deactivated alphas most likely would not be reactivated after the first drop of architecture size due to the convergence of architecture flop loss.

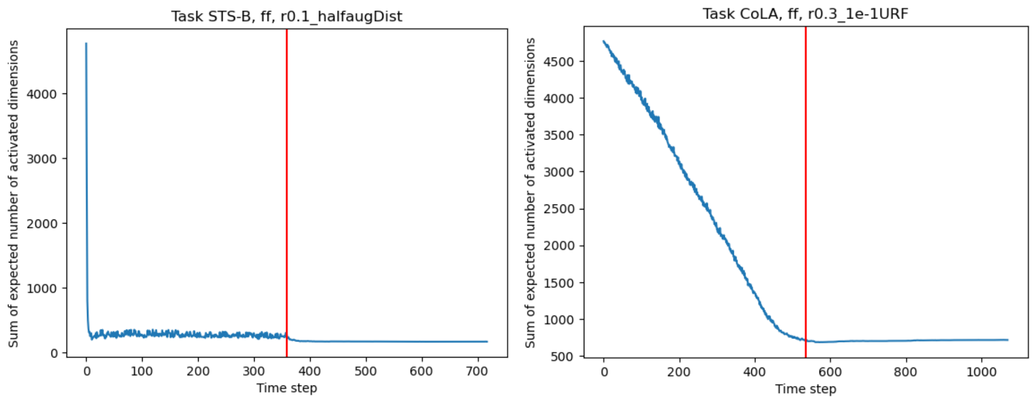


Fig. Left: A learning curve example of early converged harmful small architecture, at the start of two stage distillation search. Right: A learning curve example of scheduled searching by addictive scheduling.

We further experimented with two kinds of scheduling scheme which search for the target size ratio R in s global steps of training: the addictive scheduling where the update rule is given by $r_{t+1} = r_t + c$ and $c = \frac{1-R}{s}$, and the multiplicative scheduling is given by $r_{t+1} = mr_t$ for some choices of m, e.g. $m = 10^{\frac{\log(R)}{s}}$. For both cases we require $r_1 = 1$ and $r_s = R$.

Multiplicative	Average Performance (5 runs)	Standard Deviation
CoLA	0.422	0.0211
SST-2	0.916	0.0040

Additive	Average Performance (5 runs)	Standard Deviation
CoLA	0.414	0.0117
SST-2	0.918	0.0033

Table 11. Experiment results of multiplicative and additive scheduling.

For our project we adopted the additive scheduling, which has less standard deviation and is a more stable choice for architecture searching.

7.5 Architecture Parameter Control

With scheduled target architecture size, we need to further control the learning of alpha parameters. We propose this method of controlling the update of alpha parameters by only backpropagating the gradient of large magnitude, which ideally represents the gradient that carries the most information and has the most influence towards the final goal of searching. For TinyBERT 4L, we control the backpropagation algorithm so that gradients from the top 10% in terms of magnitude can be used to update the corresponding alpha parameter value, while the rest of the gradients are removed and no updates would be done to their corresponding alpha value.

This amount of allowed gradient update is a heuristic and the best value depends on the size of the original architecture. As long as the algorithm converges to the desired architecture size then the chosen ratio of gradient update is suitable for the algorithm combined with the architecture. By experiment we expect that models that have larger architecture size would require a larger amount of allowed gradient update.

By experiment we have shown that these controlling tricks would lead us to a better architecture when comparing the setup without controlling tricks. Removing control over the update of the gradient backpropagation will lead to instability of the searching algorithm.

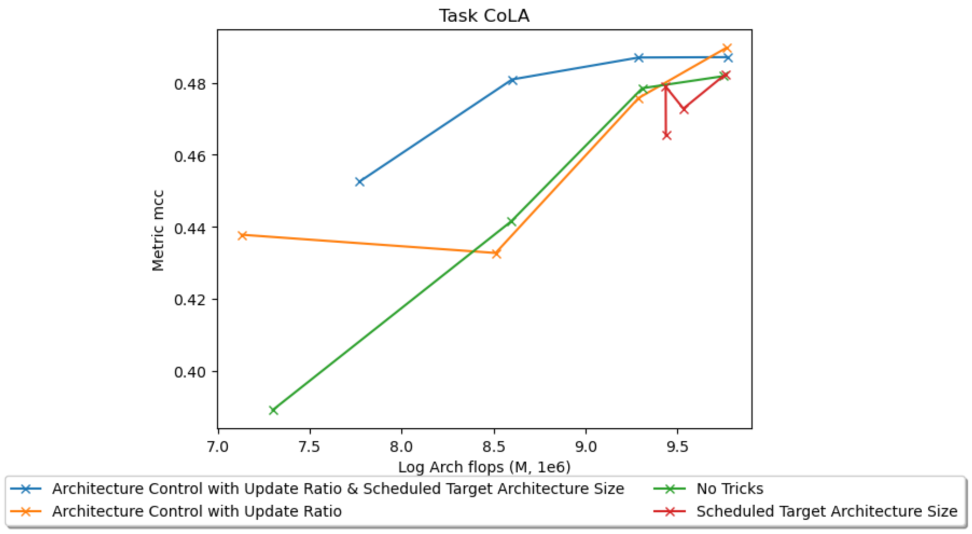


Fig. Performance comparison between different control setup.

To understand the behavior of this altered gradient descent algorithm, where only the top-10% magnitude partial derivatives are backpropagated, we review the theory of gradient descent:

Proposition 1 *Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable at $\bar{x} \in \mathbb{R}^n$. If there exists a $d \in \mathbb{R}^n$ such that $\nabla f(\bar{x})^T d < 0$, then there exists an $\alpha_0 > 0$ such that $f(\bar{x} + \alpha d) < f(\bar{x})$ for all $\alpha \in (0, \alpha_0)$. In other words, d is a* **descent direction** *of f at $\bar{x}$.*

Fig. Descent direction for unconstrainted optimization problem, from [38].

We further verify that masking some dimensions of the gradient $\nabla f(\bar{x}) \in \mathbb{R}^n$, i.e. $-\nabla f(\bar{x})' = Masking\big(-\nabla f(\bar{x})\big)$ is still a descent direction of f at $\bar{x}$. Suppose the index set S contains the index where $\nabla f(\bar{x})'_i = 0 \; \forall i \in S$, then

$$\nabla f(\bar{x})^T(-\nabla f(\bar{x})') = -\sum_{i \notin S}^{n} \nabla f(\bar{x})_i^2 \le 0$$

, and equality holds when $\nabla f(\bar{x}) = 0$ the zero vector, i.e. when the gradient descent algorithm converges. Thus, we have shown that $-\nabla f(\bar{x})'$ is a descent direction of the loss function.

7.6 Data Augmentation in Architecture Search

As stated in Section 5 we use augmented data to fine-tune the searched architecture, but augmented data is not used during architecture search since it might lead to instability of the searching algorithm. We have observed that our architecture searching algorithm is unstable when the number of training step is too large, e.g. when searching on augmented dataset. At some point our searching algorithm will early converge to an architecture that is larger than the expected architecture size. In consideration of implementation we would like to avoid searching for the architecture on augmented dataset. Future work is necessary on investigating why this happens only when searching on larger datasets, and also whether searching on smallest enough number of samples would be enough for our searching algorithm.

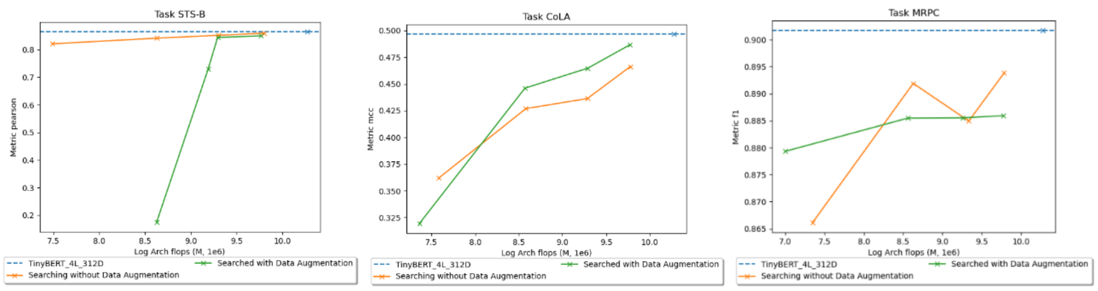


Fig. Dev set result by searching on augmented data in 1 epoch. For some task it fails to converge to the target architecture size. In difficult task such as CoLA data augmentation helps with searching for a better architecture.

7.7 Performance-FLOPS Plot Analysis

Observe from the figures in 6.4.1 that our major resulting models in different sizes forms a straight line on x-axis log plot, which suggests that prediction performance and computational FLOPS are in the log relationship. This means that if we want to have a slight increase in the prediction performance most likely we need to double the computational FLOPS, i.e. double the architecture size. Meanwhile we can compare two methods by looking at the slope in the log plot. In the context of network compression, a method with a smaller slope in the log plot suggests that it has a smaller performance drop than those with a higher slope.

7.8 Keep Weights from Searching

During architecture search both the architecture parameters and the BERT model parameters are trained at the same time, for the original model to adapt to the new architecture. Intuitively we would like to see if keeping the trained BERT model parameters for fine-tuning is helpful or not.

By experiment we know that keeping weights from searching procedure will hurt the performance of fine-tuned model. The main difference between keeping and not keeping the weight from searching is that the sub-architecture is fine-tuned on different initialization of network parameters. To explain why not keeping the weights is better, we need to focus on the fact that the objective of our architecture searching is to get as close to the original model as possible. During the searching procedure, the network weights of the BERT model might walk far away from the original values to fit with the varying architecture. Initializing the BERT weights from original model can be a good starting point because the original model is already fine-tuned, and it is closest to the searching objective, i.e. minimizing distillation loss. At this point of writing we also reflect whether fixing the BERT weights during searching would be beneficial or not. This can be future direction of investigation.

7.9 FLOPS Weight Sensitivity Analysis

Recall equation (3) in section 5.2:

$$\mathcal{L}_{arch} = \mathrm{MSE}(o_{student}, o_{teacher}) + \lambda_{cost}\mathcal{L}_{cost} \tag{3}$$

, where we can recognize λ_{cost} as the FLOPS weight. We further experiment with the different values of the FLOPS weight.

CoLA: FLOPS Weight	0.01	0.1	1	10	100
AVG (3 runs)	0.4063	0.4221	0.4338	0.4126	0.4060
STD. DEV	0.0074	0.0223	0.0134	0.0225	0.0145

SST-2: FLOPS Weight	0.01	0.1	1	10	100
AVG (3 runs)	0.916284	0.916284	0.916284	0.916667	0.916667
STD. DEV	0.00716	0.00459	0.00229	0.00132	0.00331

Table 12. Experiment results of different FLOPS weight averaged over 3 runs.

By the experiment results we see that $\lambda_{cost} = 1$ performs the best on CoLA, while other values for SST-2 performs comparably. For the universal choice of λ_{cost} for all tasks, we choose $\lambda_{cost} = 1$.

At the same time, we observe that the searching algorithm early converges to larger architecture size than the target size, when searching on large tasks (e.g. SST-2) with small FLOPS weight (e.g. 0.01). This can be explained by the fact that the gradient propagated back to the alpha variables is not strong enough when FLOPS weight is too small.

8 Conclusion

At the time of writing this report, we do not have the results on QQP and MNLI yet due to the large size of these dataset. An updated version of this report with these results will be available soon.

Through the result of this project we can see that the redundancy is high in the TinyBERT 4L architecture when fine-tuned for tasks including MRPC, STS-B and SST-2. While for the remaining tasks searching for a sub-architecture from TinyBERT 4L seems to make severe damage to the prediction performance. This also suggests that the ability of compression not only depends on the original architecture but also on the nature and difficulty of the downstream task. By observation we suggest that model fine-tuned on larger datasets like MNLI and QQP are more tolerant to network compression, thus more data samples can eventually lead to more efficiently computable neural network model.

A potential future direction of this project is to understand the tolerance of compression on different size of the original architecture. In this report we mainly focus on pruning TinyBERT 4L (1.239 B FLOPS), while other variants of BERT such as TinyBERT 6L (11.10 B FLOPS) and Bert-base (22.20 B FLOPS) are up to 20x larger and we would like to understand whether larger architecture is more tolerant to pruning or not. Using similar idea from Section 7.6 we can understand the difference between these architectures of different sizes, even though they do not end up having the same size on the compressed model.

We hope that the discovery of this report is insight for the continuing study on network compression over neural network architectures, especially state-of-the-art self-attention based deep neural networks where computational complexity is growing faster than ever.

Here I would also like to mention that this work is impossible without the help from Haoli BAI, Prof. Michael Lyu and Edward Yau. Thank you everyone for the advices to this project.

Reference

[1] F. Hutter, L. Kotthoff, J. Vanschoren, H. J. Escalante, I. Guyon, S. Escalera: *Automated Machine Learning Methods, Systems, Challenges (2019)*

[2] L. Zimmer, M. Lindauer, F. Hutter: *Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL (2020)*

[3] X. Dong, Y. Yang: *Network Pruning via Transformable Architecture Search (2019)*

[4] K. He, X. Zhang, S. Ren, J. Sun*: Deep Residual Learning for Image Recognition (2015)*

[5] H. Jin, Q. Song, X. Hu: *Auto-Keras: An Efficient Neural Architecture Search System (2019)*

[6] P. I. Frazier: *A Tutorial on Bayesian Optimization (2018)*

[7] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, F. Hutter: *Efficient and Robust Automated Machine Learning (2015)*

[8] Y. Cheng, D. Wang, P. Zhou, T. Zhang: *A Survey of Model Compression and Acceleration for Deep Neural Networks (2020)*

[9] C. F. Wang*: A Basic Introduction to Separable Convolutions (2018)*

[10] G. Hinton, O. Vinyals, J. Dean: *Distilling the Knowledge in a Neural Network (2015)*

[11] D. Chen, Y. Li, M. Qiu, Z. Wang, B. Li, B. Ding, H. Deng, J. Huang, W. Lin, J. Zhou: *AdaBERT: Task-Adaptive BERT Compression with Differentiable Neural Architecture Search (2020)*

[12] A. Romero, N. Ballas, S. Ebrahimi Kahou, A. Chassang, C. Gatta, Y. Bengio: *FitNets: Hints for Thin Deep Nets (2015)*

[13] K. M. Tarwani, S. Edem: *Survey on Recurrent Neural Network in Natural Language Processing (2017)*

[14] M. Venkatachalam*: Recurrent Neural Networks (2019)*

[15] S. Hochreiter, J. Schmidhuber: *Long Short-Term Memory (1997)*

[16] C. Olah*: Understanding LSTM Networks (2015)*

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin: *Attention Is All You Need (2017)*

[18] H. Y. Lee: *ELMO, BERT, GPT (2019)*

[19] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, L. Zettlemoyer*: Deep contextualized word representations (2018)*

[20] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever: *Improving Language Understanding by Generative Pre-Training (2018)*

[21] J. Devlin, M. W. Chang, K. Lee, K. Toutanova*: BERT: Pre-training of Deep*

Bidirectional Transformers for Language Understanding (2019)

[22] M. Schuster, K. Nakajima*: Japanese and Korean Voice Search (2012)*

[23] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, S. R. Bowman: *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding (2019)*

[24] H. Liu, K. Simonyan, Y. Yang: *Darts: Differentiable architecture search* (2019)

[25] I. Turc, M. W. Chang, K. Lee, K. Toutanova: *Well-Read Students Learn Better: On the Importance of Pre-training Compact Models (2019)*

[26] X. Dong, Y. Yang: *Searching for A Robust Neural Architecture in Four GPU Hours (2019)*

[27] E. Jang, S. Gu, B. Poole: *Categorical Reparameterization with Gumbel-Softmax (2017)*

[28] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, Q. Liu: *TinyBERT: Distilling BERT for Natural Language Understanding (2020)*

[29] K. Clark, U. Khandelwal, O. Levy, C. D. Manning: *What Does BERT Look At? An Analysis of BERT's Attention (2019)*

[30] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. v. Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush: *HuggingFace's Transformers: State-of-the-art Natural Language Processing (2020)*

[31] P. Michel, O. Levy, G. Neubig*: Are Sixteen Heads Really Better than One? (2019)*

[32] J. Frankle, M.Carbin: *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks (2018)*

[33] S. Han, J. Pool, J. Tran, W. J. Dally: *Learning both Weights and Connections for Efficient Neural Networks (2015)*

[34] V. Ramanujan, M. Wortsman, A. Kembhavi, A. Farhadi, M. Rastegari: *What's Hidden in a Randomly Weighted Neural Network? (2019)*

[35] E. Strubell, A. Ganesh, A. McCallum: *Energy and Policy Considerations for Deep Learning in NLP (2019)*

[36] M. Gupta, P. Agrawal: *Compression of Deep Learning Models for Text: A Survey (2020)*

[37] T. Chen, J. Frankle, S. Chang, S. Liu, Y. Zhang, Z. Wang, M. Carbin*: The Lottery Ticket Hypothesis for Pre-trained BERT Networks* (2020)

[38] Anthony M. C. So: *Handout 7: Optimality Conditions and Lagrangian Duality in ENGG5501 (CUHK, 2020-2021)*