ESTR 4998 FYP LYU2002 2020/21 Term 1 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 is proposed 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. The latest experiment result shows that there is no redundancy in the input token embedding.

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

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.

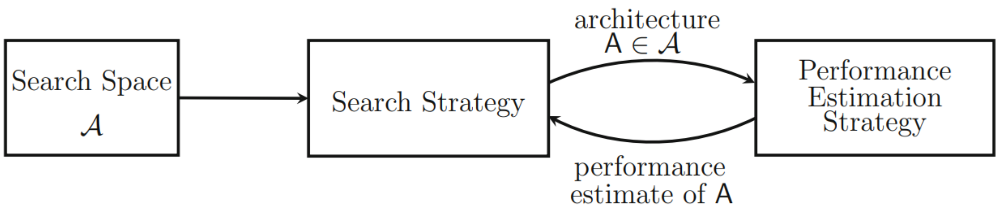


Fig. from [1, Chapter 3]. An illustration of NAS.

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.

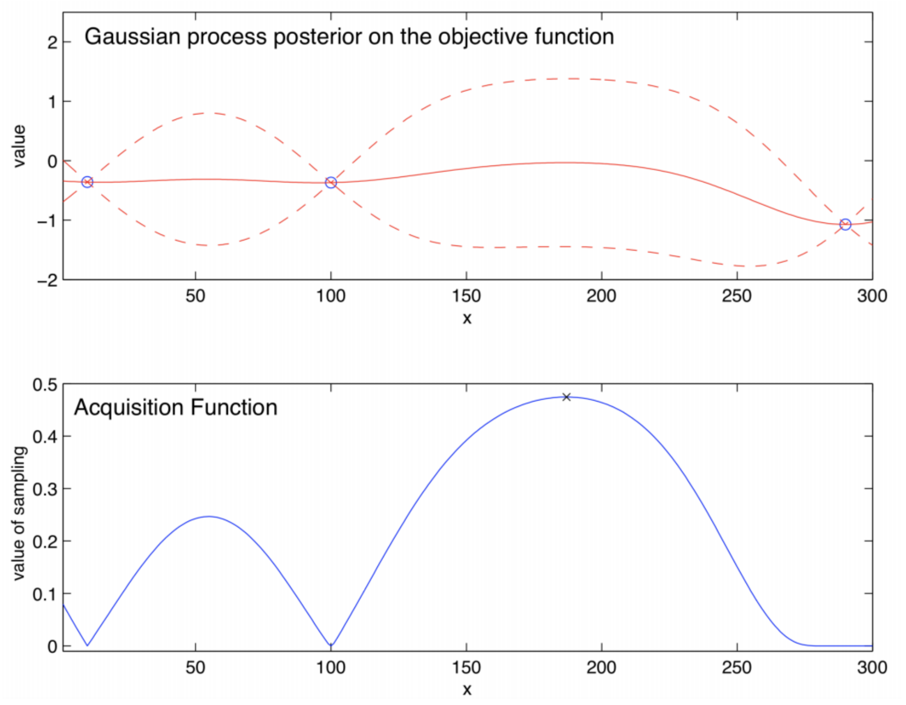


Fig. from [6]. Top: 3 blue points represent observed data point, which can be referred as the evaluated architectures in NAS. Dashed red lines represent the confidence intervals of the estimation of unobserved input. Bottom: Acquisition function suggests the next relevant points of sampling to get the 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 is perhaps the most famous approach to network compression, where the less informative and less impactful connections are removed 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.

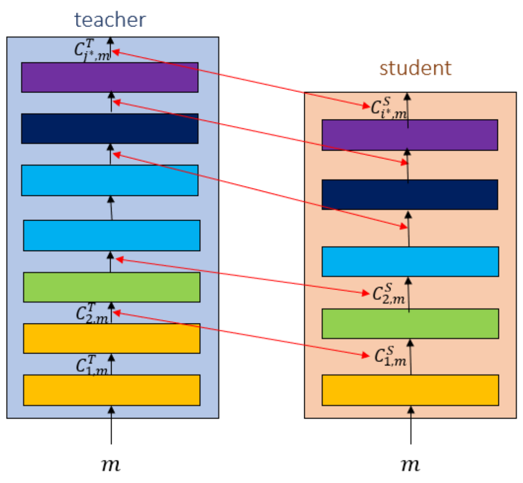


Fig. Example illustrating intermediate layer distillation, red arrows represent that the student is training w.r.t the output of intermediate layers of the teacher network, the same color of the layer blocks represent that they have a similar function for processing the input.

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.

2.3 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.

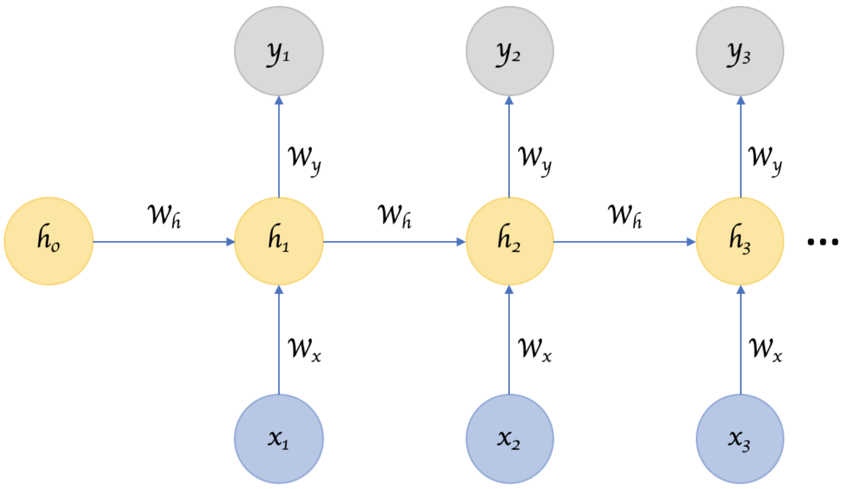


Fig. from [14]. An illustration of the computational graph of RNN. h_i *represents the internal output (hidden states) of the RNN at time* i *after manipulating the input* x_{i-1} *and* h_{i-1}. y_i *are the outputs of the RNN at time i.*

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.

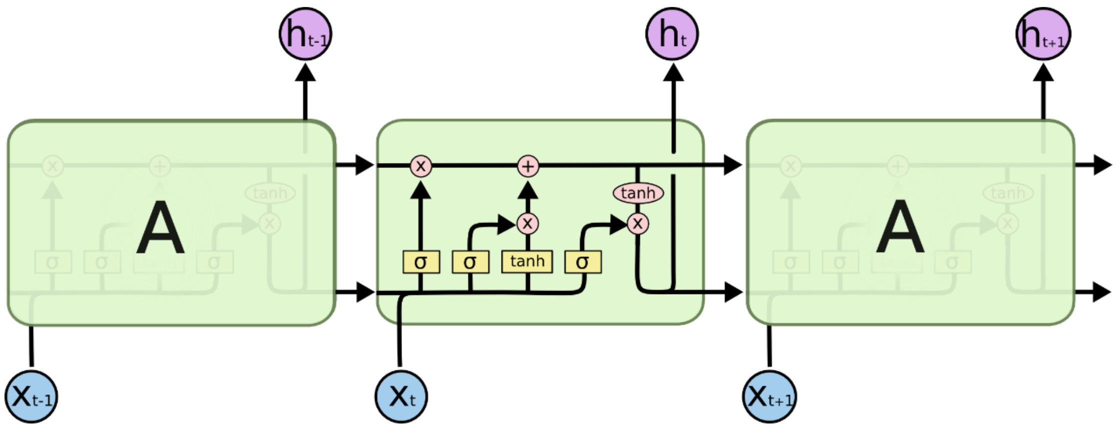


Fig. from [16]. An overview of LSTM. The non-linearities in the connection of gates expanded the learning capacity of the memory mechanism.

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.

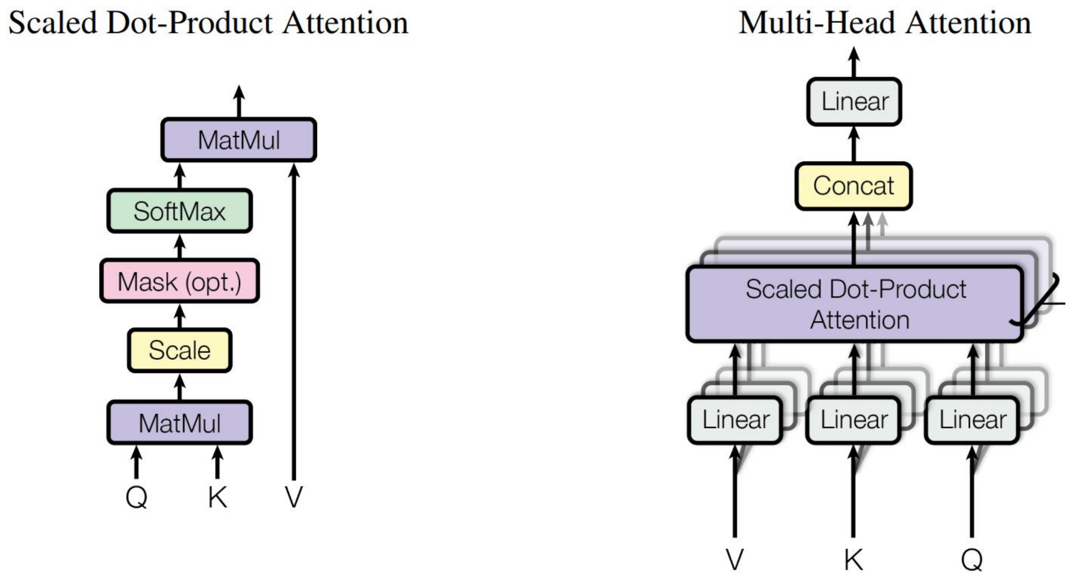


Fig. from [17]. On the right side is the illustration of a self-attention layer. On the left side is the illustration of the attention mechanism.

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.

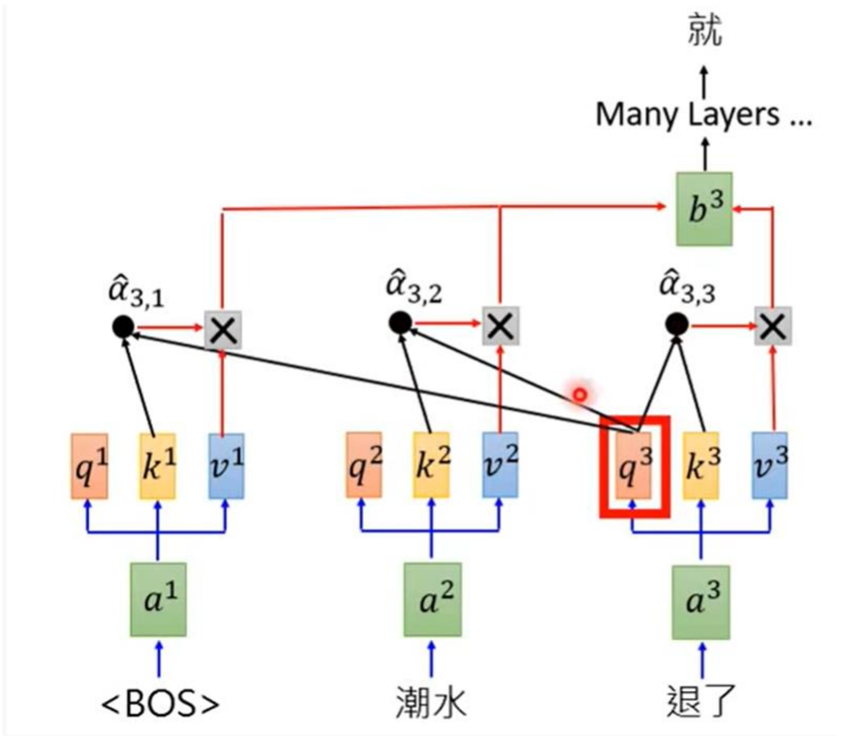


Fig. from [18]. An example of the attention mechanism used in the decoder of Transformer. The query will match with all the related keys in the input sequence. In this case, attention is masked and is limited to the word tokens at earlier locations.

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.

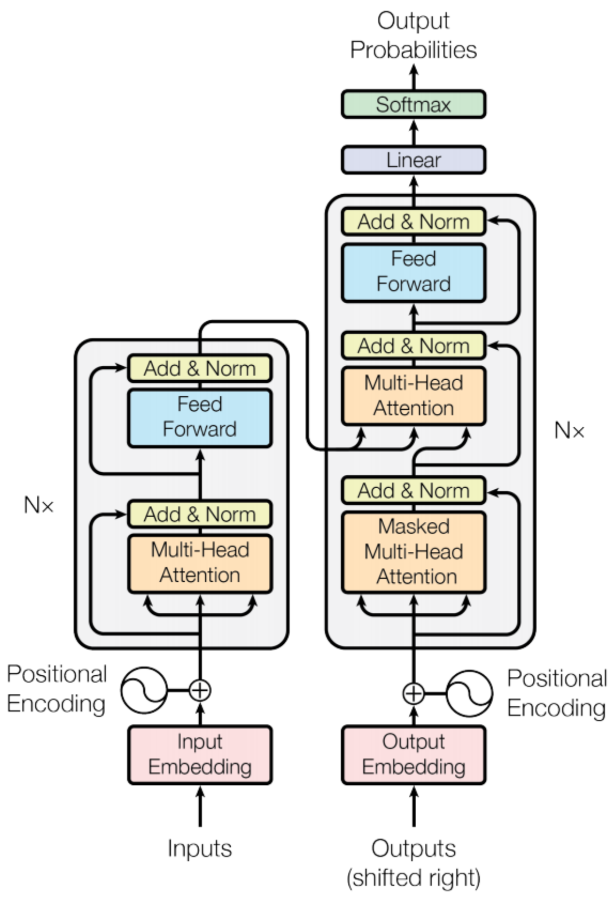


Fig. from [17]. The architecture of Transformer. On the left is the encoder stack, on the right is the decoder stack.

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, $t_1, \ldots, t_{k-1}$ is the history of the forward LSTM and $t_{k+1}, \ldots, t_N$ 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.

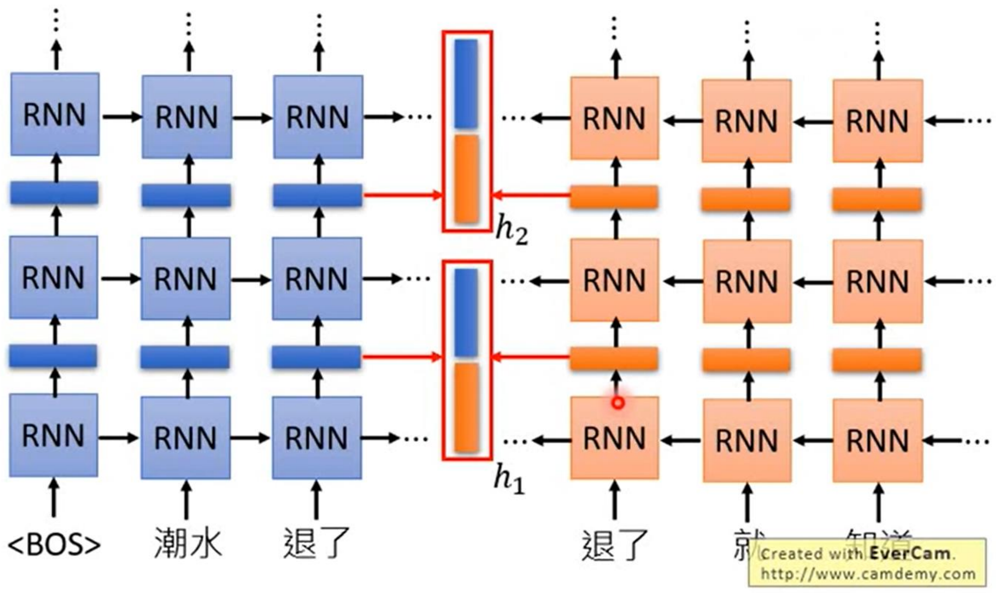


Fig. from [18]. An illustration of bidirectional LSTM used in ELMo. h_1, h_2 *in the middle represents the intermediate layer representations of the stack of LSTMs.*

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.

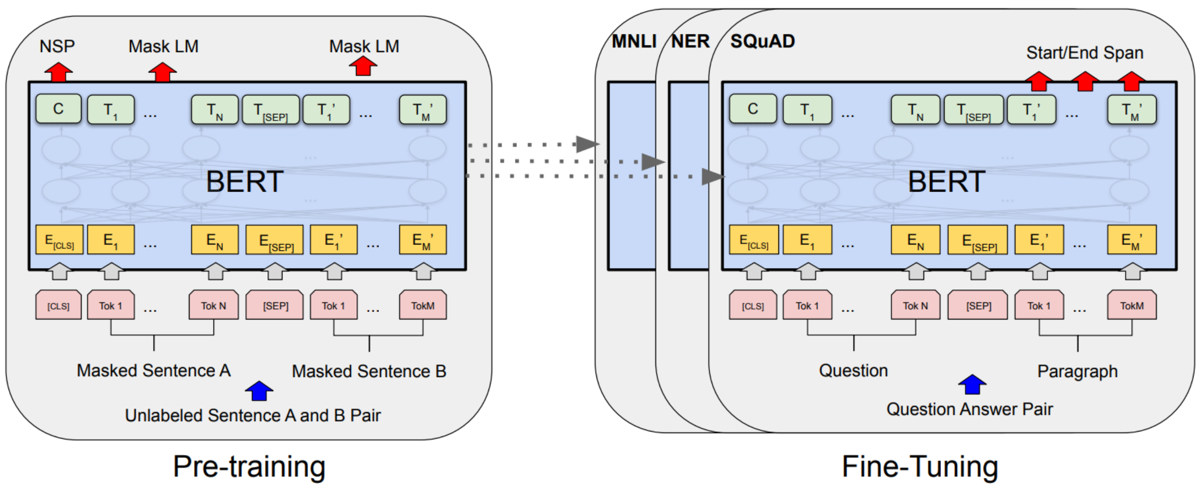


Fig. from [21]. Overall pre-training and fine-tuning procedures for BERT. Unsupervised pre-training is shown on the left, where two tasks are simultaneously used to train the model. Supervised fine-tuning is shown on the right, depending on the task specification we use the output of the BERT encoder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

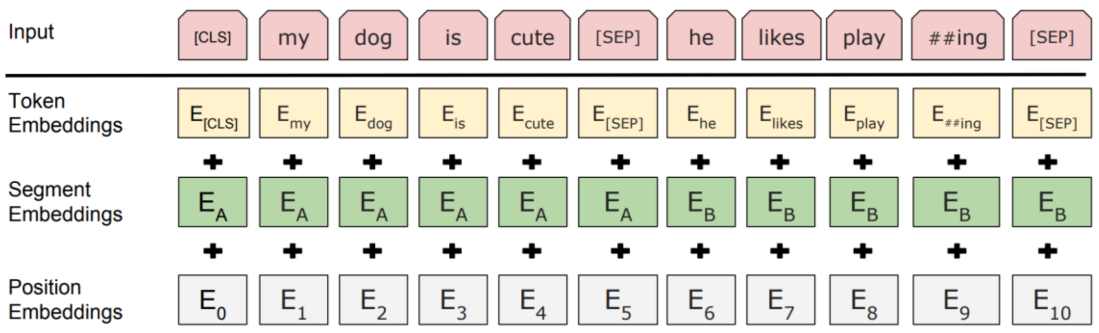

Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

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. 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)])

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

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

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.

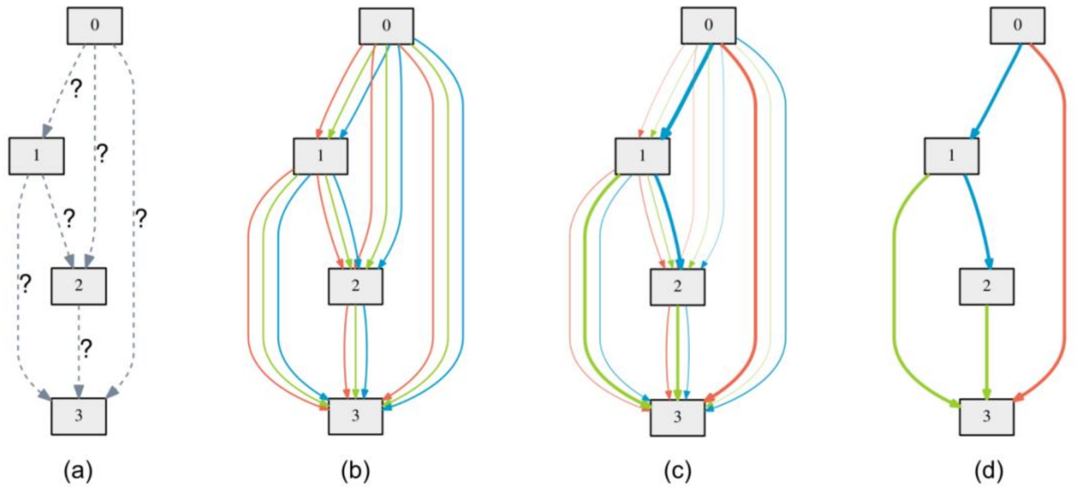


Fig. from [24]. An overview of DARTS. (a) shows that the operation on each edge is initially unknown. (b) shows that a continuous relaxation of the discrete search space is done by allowing a mixture of the operations happening on each edge. (c) by learning the set of α we can tell which operation is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

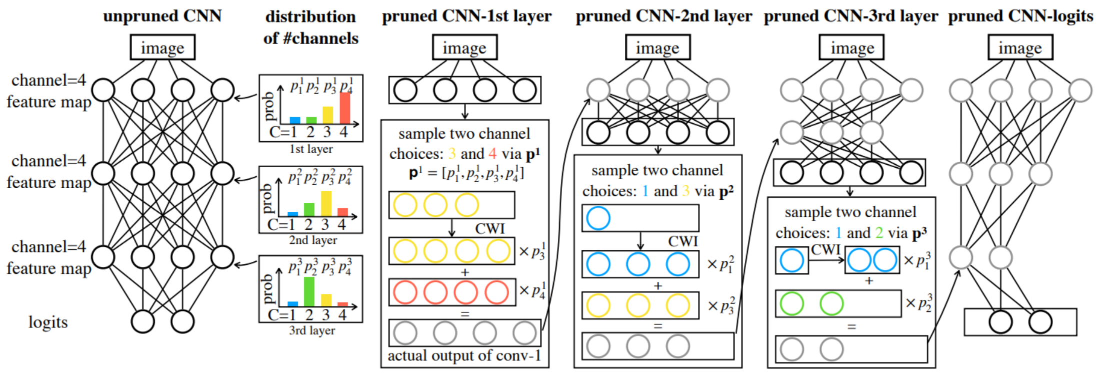


Fig. from [25]. An illustration of the procedure of searching for the width of a convolutional neural network using TAS. At each layer, 2 choices of the number of channels are sampled and the architecture for one forward step is determined after all the sampling.

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)]:

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

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

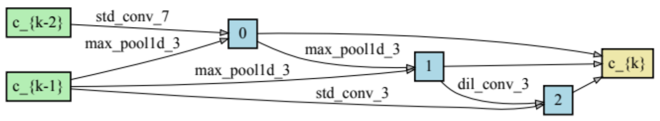
, where $\mathbb{A}$ represents the architecture parameters modelled by α, $\mathbb{E}_{cost}(\mathbb{A})$ is the expected computation cost of the possible architectures and $F_{cost}(\mathbb{A})$ is the actual cost of the sampled architecture. Here we are using the target R as a parameter to control 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.

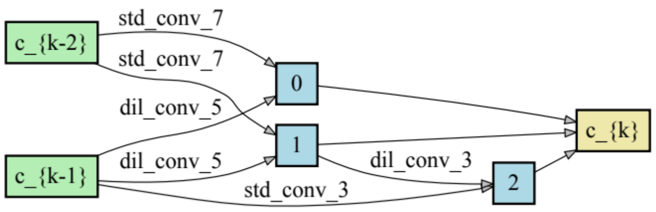
4.3 AdaBERT

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.

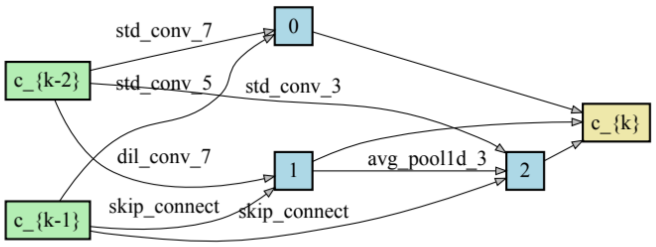
The results of AdaBERT is promising, showing the advantage of inference speedup of the searched architecture and significant compression ratio of the network. This implies the computational efficiency of convolutional operations over attention mechanism.



(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.

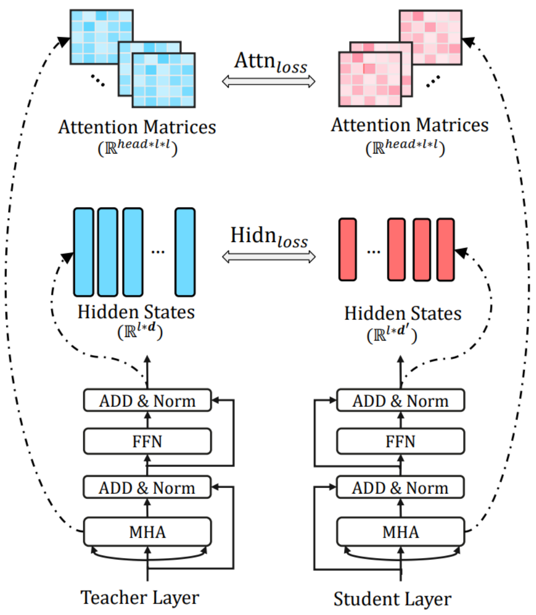

Attn$_{loss}$
Attention Matrices ($\mathbb{R}^{head*l*l}$)
Attention Matrices ($\mathbb{R}^{head*l*l}$)
Hidn$_{loss}$
Hidden States ($\mathbb{R}^{l*d}$)
Hidden States ($\mathbb{R}^{l*d'}$)
ADD & Norm
FFN
ADD & Norm
MHA
Teacher Layer
Student Layer

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

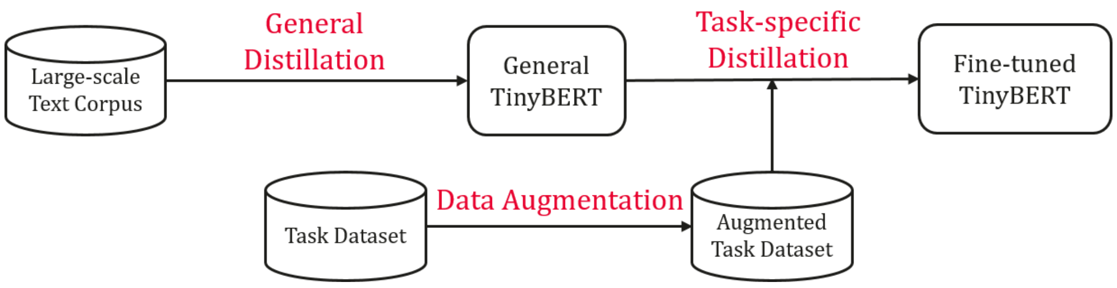


Fig. from [28]. An overview of TinyBERT learning.

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.

5 Experiment

5.1 Experiment 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. 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. 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 behaviour. For our project, we will only use the reproduced models for development and testing, as well as establishing results of our pruning method.

5.2 Experiment 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 4., evaluated on the evaluation set.

5.3 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 inference time or the network parameter size.

To facilitate the minimization of architecture FLOPS, we adopt a similar approach like [25] and formulate the objective loss function as follows:

$$\mathcal{L}_{arch} = \underbrace{-\log\left(\frac{\exp(z_y)}{\sum_{j=1}^{|z|}\exp(z_j)}\right)}_{\textit{cross-entropy classification loss}} + \underbrace{\lambda_{cost}\mathcal{L}_{cost}}_{\textit{computation cost loss}} \tag{3}$$

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

$\mathcal{L}_{arch}$ 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.

5.4 Search Space

In our NAS problem, we want to search for the best pruning of the BERT model. The search space is all the possible sub-network of the original BERT. Since we are extending the work of TinyBERT, our super net would at most be in the shape of the distilled model. We currently are using 4layer-312dim for the experiment since the smaller version requires less training time. The results of the experiment will be repeated on 6layer-768dim for comparison.

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].

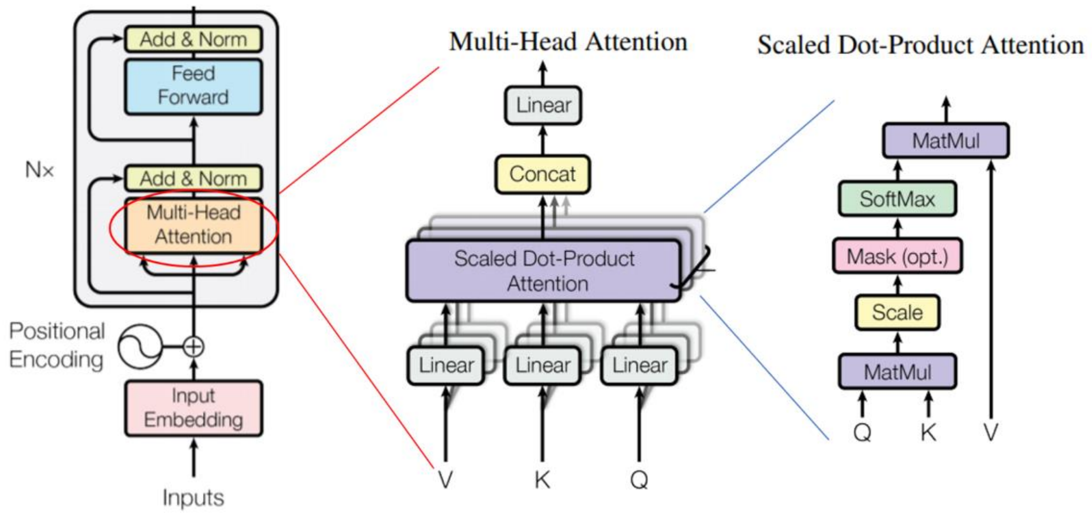


Fig. An overview of a hidden layer of BERT, read from left to right. Red and blue lines represent what are the inner operations of the concerned layer.

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.

5.4.1 Input Embedding

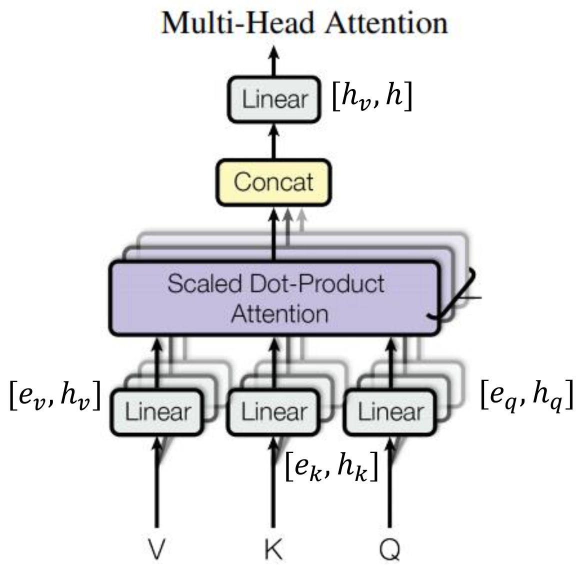


Fig. $[x, y]$ *represents a linear transformation of a vector from x dimensions to y dimensions.*

(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.

5.4.2 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))

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

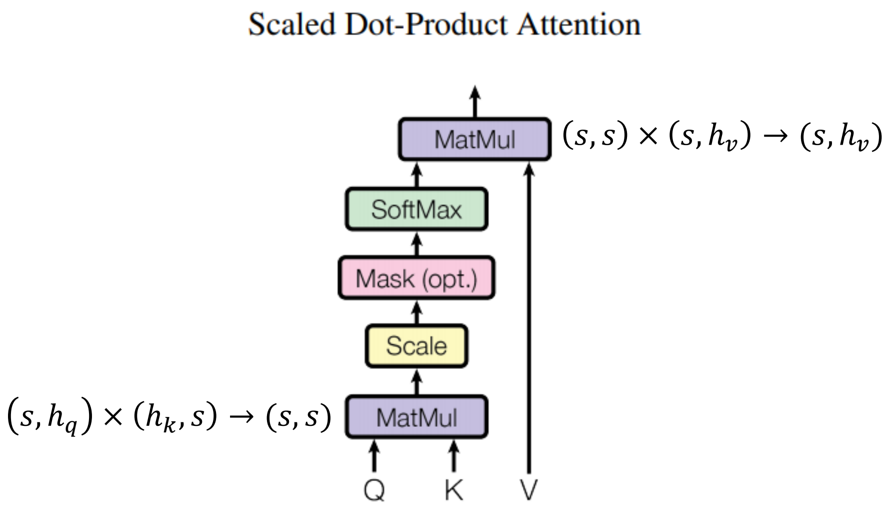

Scaled Dot-Product Attention
MatMul
$(s,s) \times (s,h_v) \rightarrow (s,h_v)$
SoftMax
Mask (opt.)
Scale
$(s,h_q) \times (h_k,s) \rightarrow (s,s)$
MatMul
Q
K
V

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.

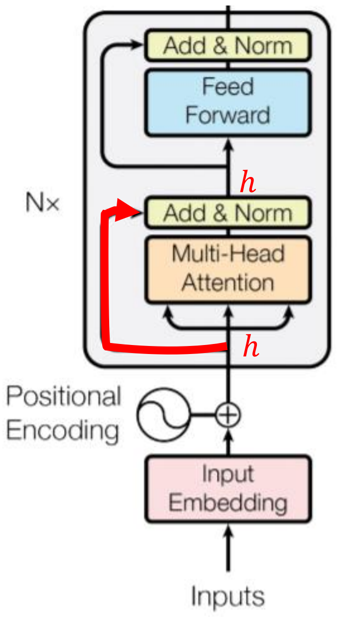


Fig. Red arrow shows the residual connection in concern. The same hidden size h must be maintained at the output of the multi-head attention layer.

5.4.3 Feed Forward Intermediate Representation

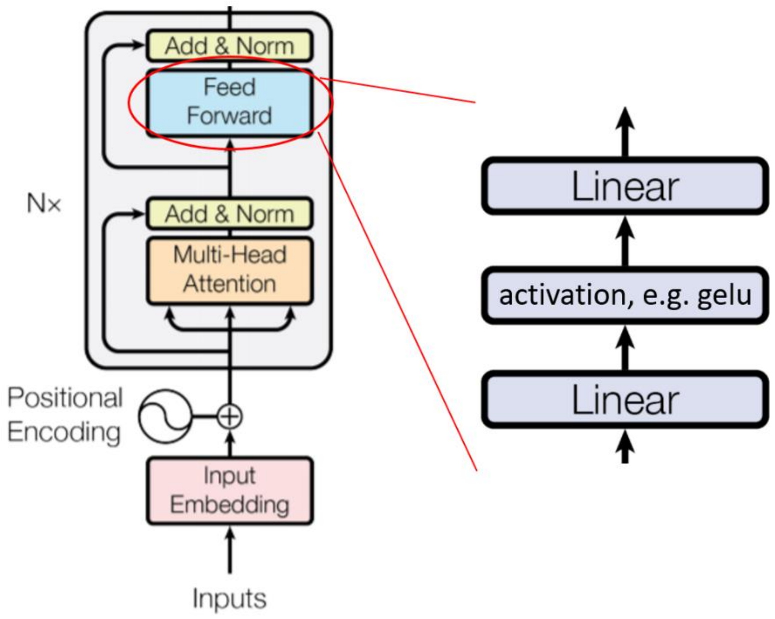


Fig. An illustration of the feed-forward layer of the encoder of the Transformer.

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.

5.4.4 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.

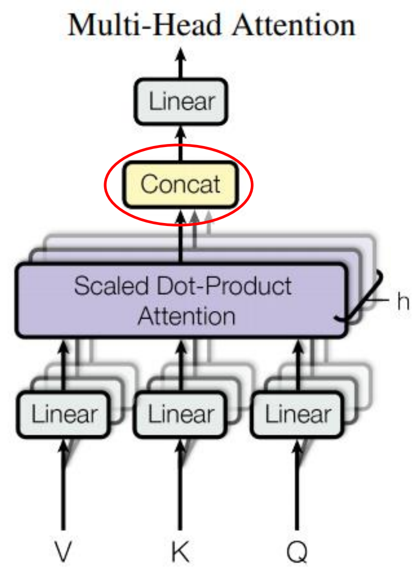


Fig. Observe that the output of each attention head is combined in the above circled layer. Pruning the multi-heads will prune away part of the linear transformation at the top as well.

5.5 Search Method

To facilitate differentiable architecture searching similar to [24] and [25], we need to design how to generate the architecture from the architecture variable α. The following method will describe how to model α to perform searching on the representation dimensions (**5.4.1, 5.4.2, 5.4.3**). An alternative method is proposed to search for multi-heads pruning (**5.4.4**).

5.5.1 Search Method for Representation Dimension

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.

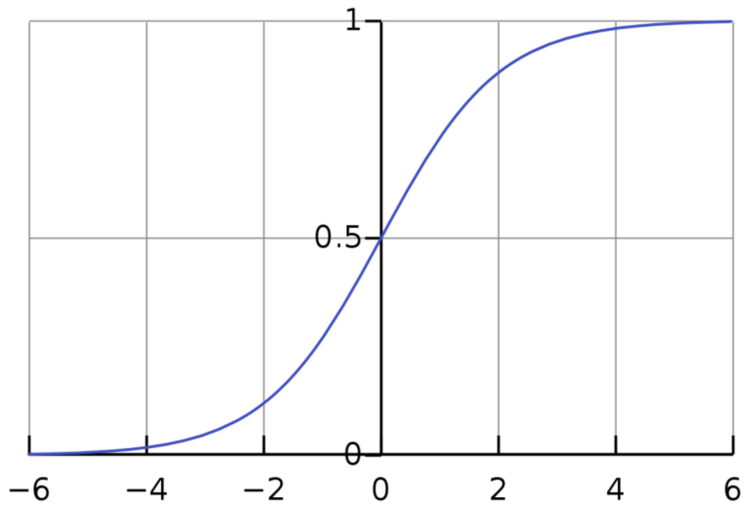


Fig. from Wikipedia. Sigmoid function.

```
Algorithm: Search Method for Representation Dimension
Initialize $\alpha$ to a constant value 5, since $\mathrm{sigmoid}(5) \approx 1$.
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($\alpha$) > 0.01
```

5.5.2 Multi-head Pruning Method

We need to model the importance of each head using α. We propose a similar approach like DARTS and weight the output of each head by α using softmax:

$$\bar{o}(x) = \sum_{h \in \mathcal{H}} \frac{\exp(\alpha_h)}{\sum_{h' \in \mathcal{H}} \exp(\alpha_{h'})} o_h(x) \tag{5}$$

, where $\mathcal{H}$ is the set of all heads within the same layer, α_h is the architecture variable of head h, $o_h(x)$ is the output of head h and $\bar{o}(x)$ is the weighted output of all the heads.

Algorithm: Search Method for Multi-head Pruning

```
Initialize $\alpha$ from Normal(0, 0.01).
Initialize encoder weights $w$ from pre-trained model.
for each forward pass:
    1. Generate attention heads weighted output according to equation (5).
    2. Backpropagate the cross-entropy loss w.r.t labels and FLOPS loss to learn $\alpha$.
Choose the heads according to the top-$n$ $\alpha$.
```

5.6 Search Results

5.6.1 Input Embedding Pruning

By the results of the experiment, we realize that it is difficult to tune the hyperparameter of our search to get the desired searching result. The search often results in using all the dimensions of the input embedding or using the minimum size of the input embedding. The following scenarios are the possible reason behind the results:

1. FLOPS weight of searching is too small and $\mathcal{L}_{arch}$ is dominated by the cross-entropy classification loss, so the model would want full information from the input embedding.
2. FLOPS weight of searching is too large and $\mathcal{L}_{arch}$ is dominated by the FLOPS loss, so the model would like to use the minimal number of dimensions to reduce network FLOPS.

At the same time through experiments, we realize even when the FLOPS weight is 0, i.e. network efficiency is not the training target of the searcher, the searching result would not use all the input embedding dimensions.

The experiment results are listed in the tables below.

TinyBERT distilled model for comparison

	CoLA (mcc)	**RTE (accuracy)**	**SST-2 (accuracy)**
reproduced 4layer-312dim TinyBERT performance (10, 10)	0.426	0.667	0.917

Searching without FLOPS loss

Task	**Evaluation**	**Result at Global Step**	**Search Result Size Ratio**
CoLA	0.236 mcc (decreased by 0.19)	699	0.406
RTE	0.621 acc (-6.89%)	199	0.456
SST-2	0.894 acc (-2.50%)	17499	0.455

Searching with FLOPS loss

Task	**Evaluation**	**Result at Global Step**	**Search Target Size Ratio**	**Search Result Size Ratio**
CoLA	0.267 mcc (decreased by 0.159)	2349	0.5	0.654
CoLA	0.289 mcc (decreased by 0.137)	99	0.75	0.828
CoLA	0.355 mcc (decreased by 0.071)	1249	1.0	0.974
RTE	0.646 acc (-3.14%)	49	0.5	0.663
RTE	0.646 acc (-3.14%)	149	0.75	0.825
RTE	0.653 acc (-2.09%)	499	1.0	0.975
SST-2	0.905 acc (-1.30%)	17499	0.5	0.662
SST-2	0.906 acc (-1.19%)	15549	0.75	0.852
SST-2	0.909 acc (-0.872%)	749	1.0	0.974

Search target size ratio is representing r in $rM = R$, where R is the target in equation (4) (Section **5.3**) and M is the maximum embedding size limit of the architecture. In this experiment $M = 312$. Search result size ratio represents the size of the resulting architecture, because by equation (4) the FLOPS loss would only train the architecture to approach the target size within a tolerance range. Change in performance is quoted in brackets, comparing to the distilled model without pruning. The results are incomplete since the training of SST-2 takes a longer time, and experiment have not completed yet for SST-2.

From the results above we see that the performance drop is significant even when the search result size ratio approaches to 1.0, i.e. no pruning. It suggests that all the dimensions of the input embedding are informative in performing the downstream task. There is little redundancy in the input token embedding. However, we see that the performance drop on SST-2 is minimal, which suggests that pruning the dimension of input embedding for downstream tasks of a large dataset will do less damage to the performance as compared with pruning on smaller datasets.

6 Future Direction

At the time of writing the report, we have not yet experimented on pruning of the qkv hidden representation (Section **5.4.2**) and the feedforward intermediate layer representation (Section **5.4.3**). But we have a higher expectation on the pruning of attention multiple heads (Section **5.4.4**), based on the finding of [31].

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