VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

INFORMATION TECHNOLOGY FACULTY

FINAL REPORT

MACHINE LEARNING

Instructor:	Prof. Lê Anh Cường
Student 1:	Đỗ Phạm Quang Hưng - 520K0127
Student 2:	Lê Phước Thịnh - 520K0343

HO CHI MINH CITY, 2023

TOC

Table of Contents

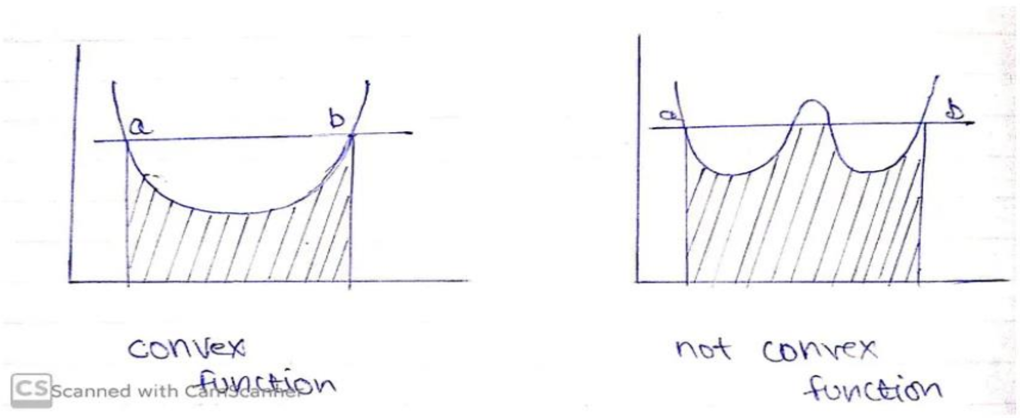
Task 1

1. Gradient Descent:

Gradient Descent is an **optimization algorithm** which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the **cost function** within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is **close to or equal to zero**, the model will continue to adjust its parameters to yield the smallest possible error. Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.

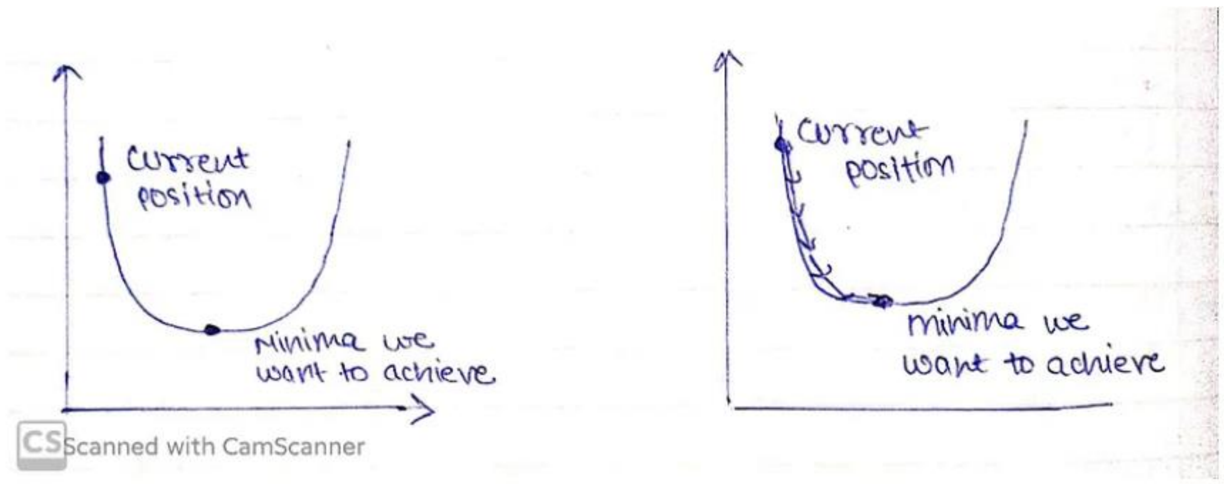
In machine/deep learning terminology, it's the task of minimizing the cost/loss function J(w) parameterized by the model's parameters w ∈ R^d. Optimization algorithms (in the case of minimization) have one of the following goals:

1. Find the global minimum of the objective function. This is feasible if the objective function is convex, i.e. any local minimum is a global minimum.
2. Find the lowest possible value of the objective function within its neighborhood. That's usually the case if the objective function is not convex as the case in most deep learning problems.



Convex function v/s Not Convex function

Let's take a look at what we want to achieve.


Current position
Minima we want to achieve
CS Scanned with CamScanner
Current position
minima we want to achieve

Intuition behind Gradient Descent

For ease, let’s take a simple linear model.

Error = Y(Predicted)-Y(Actual)

A machine learning model always wants low error with maximum accuracy, in order to decrease error, we will intuit our algorithm that you’re doing something wrong that is needed to be rectified, that would be done through Gradient Descent.

We need to minimize our error, in order to get pointer to minima we need to walk some steps that are known as alpha (learning rate).

Steps to implement Gradient Descent

1. Randomly initialize values.

2. Update values.

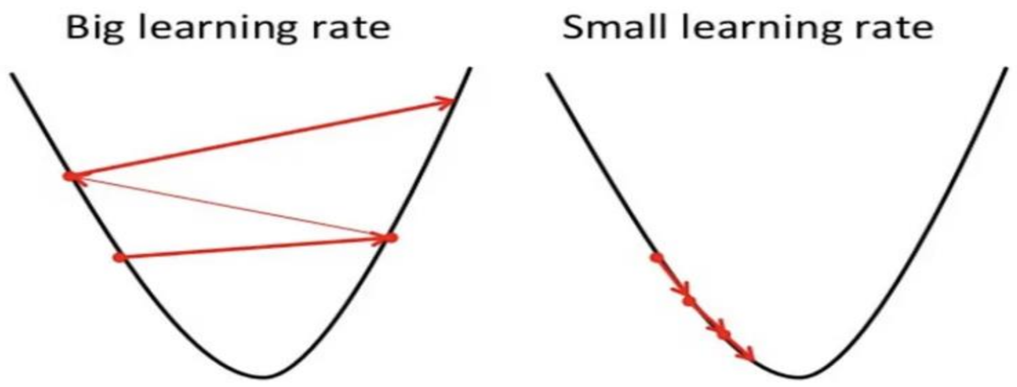
$$weight^{(new)} = weight^{(old)} - constant \frac{\partial J(\Theta)}{\partial weight}$$

3. Repeat until slope =0

Learning rate (λ) is one such **hyper-parameter** that defines the **adjustment in the weights of our network with respect to the loss gradient descent**. It determines how fast or slow we will move towards the optimal weights

The Gradient Descent Algorithm estimates the weights of the model in many iterations by minimizing a cost function at every step.

Learning rate must be chosen wisely as:
1. if it is too small, then the model will take some time to learn.
2. if it is too large, model will converge as our pointer will shoot and we'll not be able to get to minima.

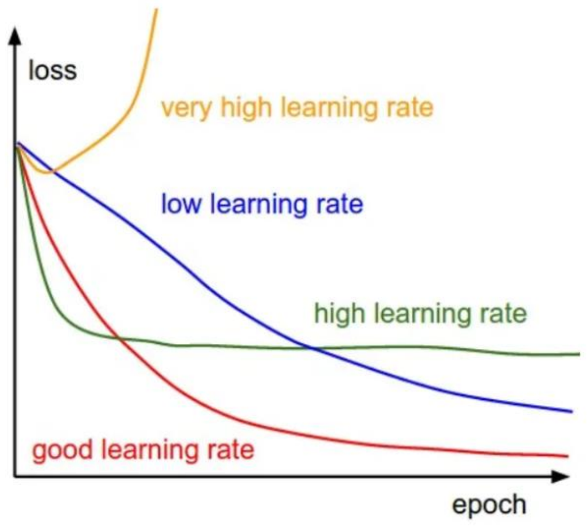


Big Learning rate v/s Small Learning rate, Source

A good way to make sure the gradient descent algorithm runs properly is by plotting the cost function as the optimization runs. Put the number of iterations on the x-axis and the value of the cost function on the y-axis. This helps you see the value of your cost function after each iteration of gradient descent, and provides a way to easily spot how appropriate your learning rate is.

If the gradient descent algorithm is working properly, the cost function should decrease after every iteration.

When gradient descent can't decrease the cost-function anymore and remains more or less on the same level, it has converged. The number of iterations gradient descent needs to converge can sometimes vary a lot. It can take 50 iterations, 60,000 or maybe even 3 million.



Gradient Descent with different learning rates

There are three main types of Gradient Descent:

1. Batch Gradient Descent
2. Stochastic Gradient Descent
3. Mini-batch Gradient Descent

2. Stochastic Gradient Descent

There are a few downsides of the gradient descent algorithm. We need to take a closer look at the amount of computation we make for each iteration of the algorithm.

Assuming we have 10,000 data points and 10 features. The sum of squared residuals consists of as many terms as there are data points, so 10000 terms in our case. We need to compute the derivative of this function with respect to each of the features, so in effect we will be doing 10000 * 10 = 100,000 computations per iteration. It is common to take 1000 iterations, in effect we have 100,000 * 1000 = 100000000 computations to complete the algorithm. That is pretty much an overhead and hence gradient descent is slow on huge data.

This is where we consider using Stochastic Gradient Descent. The word '*stochastic*' means a system or process linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration.

Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm used for optimizing machine learning models. In this variant, only one random training example is used to calculate the gradient and update the parameters at each iteration. Here are some of the advantages and disadvantages of using SGD:

Advantages:

- **Speed**: SGD is faster than other variants of Gradient Descent such as Batch Gradient Descent and Mini-Batch Gradient Descent since it uses only one example to update the parameters.
- **Memory Efficiency**: Since SGD updates the parameters for each training example one at a time, it is memory-efficient and can handle large datasets that cannot fit into memory.
- **Avoidance of Local Minima:** Due to the noisy updates in SGD, it has the ability to escape from local minima and converge to a global minimum.

Disadvantages:

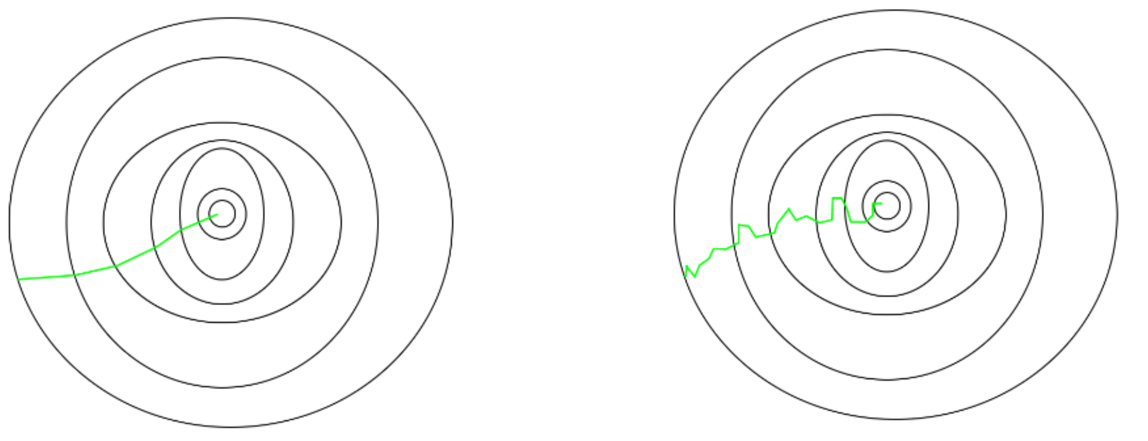
- **Noisy updates:** The updates in SGD are noisy and have a high variance, which can make the optimization process less stable and lead to oscillations around the minimum.
- **Slow Convergence**: SGD may require more iterations to converge to the minimum since it updates the parameters for each training example one at a time.

- **Sensitivity to Learning Rate:** The choice of learning rate can be critical in SGD since using a high learning rate can cause the algorithm to overshoot the minimum, while a low learning rate can make the algorithm converge slowly.
- **Less Accurate**: Due to the noisy updates, SGD may not converge to the exact global minimum and can result in a suboptimal solution. This can be mitigated by using techniques such as learning rate scheduling and momentum-based updates

In SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.

Since only one sample from the dataset is chosen at random for each iteration, the *path taken by the algorithm to reach the minima is usually noisier* than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with a significantly shorter training time.

Comparision of path taken between 2 algorithms from a top-down view



path taken by normal Gradient Descent *path taken by Stochastic Gradient Descent*

As you can see, both algorithms reached the minima but with different shapes of path. For all we care is the duration for it to reach the destination, not the path it took.

One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent.

3. Adaptive Learning Rate

Before the adaptive learning rate methods were introduced, the gradient descent algorithms including **Batch Gradient Descent** (BGD), **Stochastic Gradient Descent** (SGD) and **mini-Batch Gradient Descent** (mini-BGD, the mixture of BGD and SGD) were state-of-the-art.

The performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response.

This is called an adaptive learning rate.

Adaptive learning rate methods are an optimization of gradient descent methods with the goal of minimizing the objective function of a network by using the gradient of the function and the parameters of the network.

Adaptive learning rates can accelerate training and alleviate some of the pressure of choosing a learning rate and learning rate schedule.

An adaptive learning rate computed at every step is designed to accomplish several goals:

- Remove the hyperparameter, thus relieving us from the need to optimize it
- Speed up the learning process when the loss function hits a plateau
- Solve the problem of exploding gradients

As an improvement to traditional gradient descent algorithms, the adaptive gradient descent optimization algorithms or adaptive learning rate methods can be utilized.

One of the most popular adaptive learning rate algorithms is called Adam, which stands for Adaptive Moment Estimation.

In Adam, the learning rate is adjusted for each parameter based on the first and second moments of the gradients. Specifically, the first moment (mean) and second moment (variance) of the gradients are estimated using exponentially decaying moving averages. The estimates of the first and second moments are then used to compute an effective learning rate for each parameter.

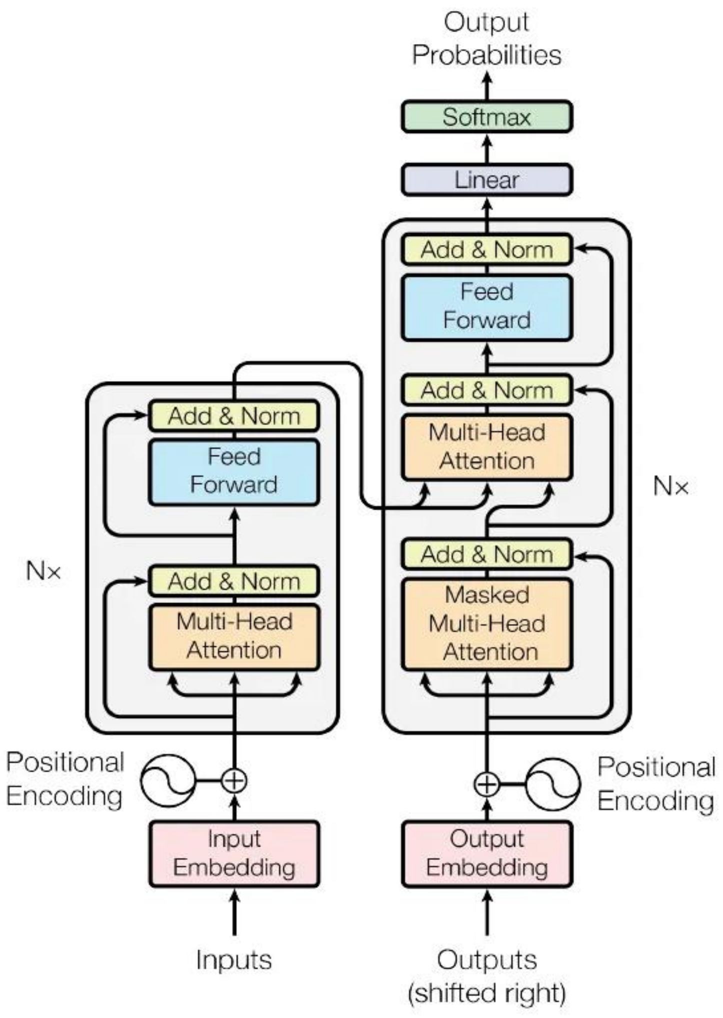
The effective learning rate for a parameter is computed by dividing the learning rate by the square root of the second moment estimate for that parameter, which is a measure of the variance of the gradients. The resulting effective learning rate is then multiplied by the first moment estimate for that parameter, which is a measure of the mean of the gradients. This results in a weight update that is scaled based on both the magnitude and direction of the gradients.

By adapting the learning rate in this way, Adam is able to converge faster and more reliably than standard gradient descent, especially for problems with sparse gradients or noisy or non-stationary objective functions.

Task 2

"**Attention Is All You Need" by Vaswani et al., 2017**" was a landmark paper that proposed a completely new type of model — the Transformer. The model introduced a new mechanism that caused a revolution in the field of Natural Language Processing – The Attention Mechanism. Nowadays, the Transformer model plays an essential role in the realms of machine learning, but its algorithm is quite complex and hard to chew on.

In general, the Transformer model is based on an encoder-decoder architecture. The encoder is the grey rectangle on the left-hand side, the decoder the one on the right-hand side. Both the encoder and decoder consist of two and three sub-layers, respectively: **multi-head self-attention**, a **fully-connected feed forward network** and — in the case of the **decoder** — **encoder self-attention**.



s

What cannot be seen as clearly in the picture is that the Transformer actually stacks multiple encoders and decoders (which is denoted by Nx in the image, i.e., encoders and decoders are stacked n times). This means that the output of one encoder is used as the input for the next encoder — and the output of one decoder as the input for the next decoder.

1. Attention Mechanism in the Encoder:

The attention mechanism in the encoder of a Transformer model is a key component that allows the model to efficiently process input sequences and capture relevant information from them.

One of the problems of recurrent models is that long-range dependencies (within a sequence or across several sequences) are often lost. That is, if a word at the beginning of a sequence carries importance for a word at the end of a sequence, the model might have forgotten the first word once it reaches the last word. And to overcome this problem, self-attention is invented for the model to memorize the sentence better!

The young <u>boy</u> always carries his teddy bear with <u>him</u>.

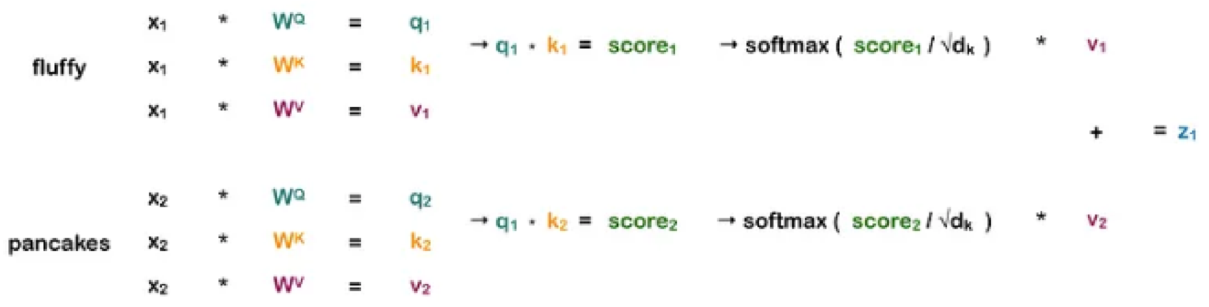
For instance, when an NLP model process the above sentence, it may not remember what is the proper pronounce to use at the end of the sentence, it could be ‘him’, ‘her’ or ‘it’? It does not know!

In the self-attention layer, an input x (represented as a vector) is turned into a vector z via three representational vectors of the input: q (Queries), k (Keys) and v (Values). These are used to calculate a score that shows how much attention that particular input should pay to other elements in the given sequence.

This is the formula to calculate the Attention:

$$\text{Attention}\ (Q, K, V) = \text{softmax}\ (((Q * K^T) / \sqrt{d_k})) * V$$

That might still be hard to get the idea, we can express it this way:


fluffy
$x_1 * W^Q = q_1$
$x_1 * W^K = k_1$
$x_1 * W^V = v_1$
$\rightarrow q_1 \cdot k_1 = \text{score}_1$
$\rightarrow \text{softmax}(\text{score}_1 / \sqrt{d_k}) * v_1$
$+$
$= z_1$
pancakes
$x_2 * W^Q = q_2$
$x_2 * W^K = k_2$
$x_2 * W^V = v_2$
$\rightarrow q_1 \cdot k_2 = \text{score}_2$
$\rightarrow \text{softmax}(\text{score}_2 / \sqrt{d_k}) * v_2$

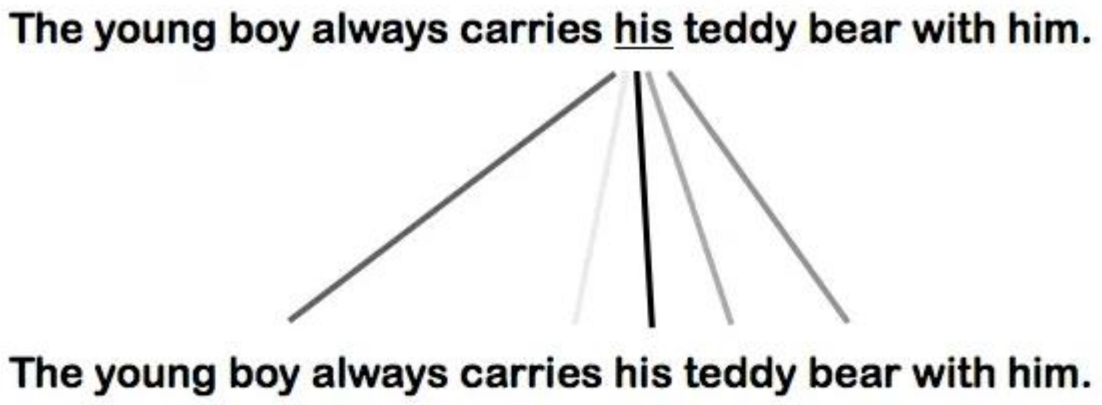
Say we want to calculate **self-attention for the word “fluffy”** in the sequence “fluffy pancakes”. First, we take the input vector *x1* (representing the word “fluffy”) and multiply it with three different weight matrices Wq, Wk and Wv (which are continually updated during training) in order to get three different vectors: q1, k1 and v1. The exact same is done for the input vector *x2* (representing the word “pancakes”). We now have a query, key and value vector for both words.

And what are these Query, Key and value mean? Here’s what they do:

The **query** is the representation for the word we want to calculate self-attention for. So since we want to get the self-attention for “fluffy”, we only consider its query, not the one of “pancakes”. As soon as we are finished calculating the self-attention for “fluffy”, we can also discard its query vector.

The **key** is a representation of each word in the sequence and is used to match against the query of the word for which we currently want to calculate self-attention.

The **value** is the actual representation of each word in a sequence, the representation we really care about. Multiplying the query and key gives us a score that tells us how much weight each value (and thus, its corresponding word) obtains in the self-attention vector. Note that the the value is not directly multiplied with the score, but first the scores are divided by the square root of the dk, the dimension of the key vector, and softmax is applied.
The result of these calculations is one vector for each word. As a final step, these two vectors are summed up, and voilà, we have the self-attention for the word "fluffy".


The young boy always carries his teddy bear with him.
The young boy always carries his teddy bear with him.

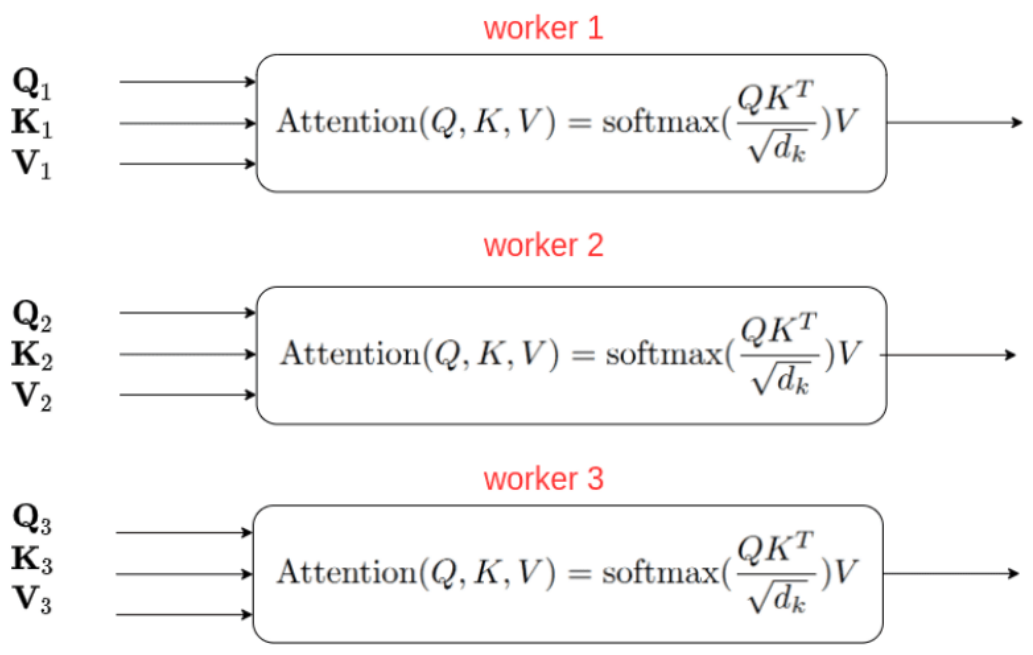
The picture above illustrates how much Attention the word 'his' pays to any other word in the sentence.

The process above is carried out multiple times with different weight matrices, which means we end up with multiple vectors (called heads in the formulae below). These heads are then concatenated and multiplied with a weight matrix Wo. This means that each head learns different information about a given sequence and that this knowledge is combined at the end.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) * W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

But the most important trait of Transformer is that all of the calculations above can be parallelized, and that is what makes the Transformer model even more beautiful!


worker 1
$\mathbf{Q}_1$
$\mathbf{K}_1$
$\mathbf{V}_1$
$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
worker 2
$\mathbf{Q}_2$
$\mathbf{K}_2$
$\mathbf{V}_2$
$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
worker 3
$\mathbf{Q}_3$
$\mathbf{K}_3$
$\mathbf{V}_3$
$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$

Let's look at RNNs first. They need to process sequential data in order, each word of a sequence is passed to the model one by one, one after the other. Transformer models, however, can process all inputs at once. And this makes these models incredibly fast, allowing them to be trained with huge amounts of data.

As for the Transformer model, it can process all words in a sequence in parallel. However, this means that some important information is lost: the **word position in the sequence**. To retain this information, the position and order of the words must be made explicit to the model. This is done via positional encodings. These positional encodings are vectors with the same dimension as the input vector and are calculated using a sine and cosine function. To combine the information of the input vector and the positional encoding, they are simply summed up.

2. Attention Mechanism in the Encoder-Decoder:

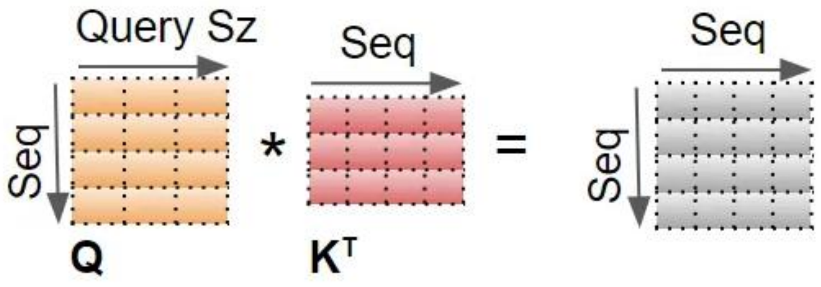
The encoder-decoder attention mechanism is used to weight the encoder outputs based on their relevance to the current position in the decoder.

Unlike the self-attention mechanism used within each layer of the Transformer encoder and decoder, which computes attention scores between all pairs of positions within a single sequence, the encoder-decoder attention mechanism computes attention scores between each position in the decoder and all positions in the encoder.

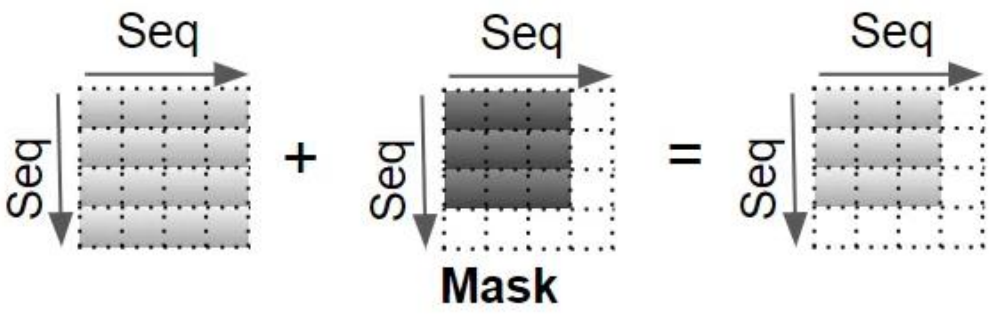
Specifically, at each position in the decoder, the encoder-decoder attention mechanism computes a set of attention scores between that position and all positions in the encoder. These attention scores are then used to weight the encoder outputs, allowing the decoder to attend to different parts of the input sequence as needed.

To compute the attention scores, the Transformer model uses three learned parameter matrices: W_q, W_k, and W_v. These matrices are used to transform the decoder input, encoder outputs, and encoder outputs, respectively, into query, key, and value representations that are used to compute the attention scores.

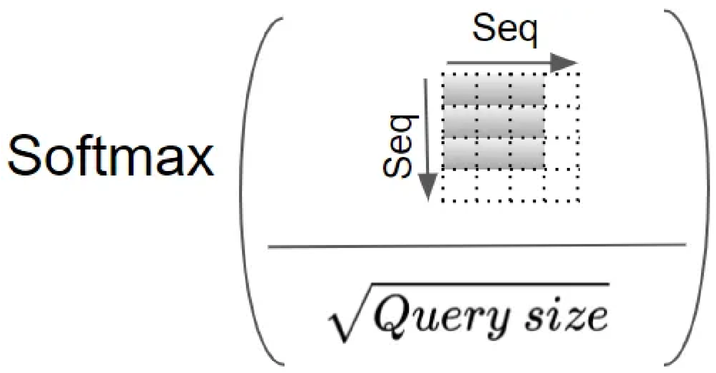
The first step is to do a matrix multiplication between Q and K.



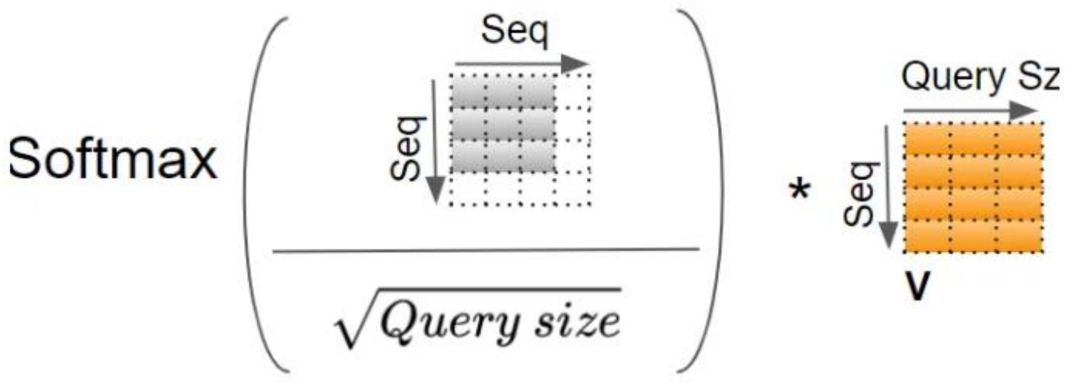
A Mask value is now added to the result. In the Encoder Self-attention, the mask is used to mask out the Padding values so that they don't participate in the Attention Score.



The result is now scaled by dividing by the square root of the Query size, and then a Softmax is applied to it.



Another matrix multiplication is performed between the output of the Softmax and the V matrix.



The resulting attention output is then concatenated with the decoder input at the current position and passed through a feedforward network to produce the final output for that position.

The encoder-decoder attention mechanism is a key component of many natural language processing tasks, including machine translation, where it allows the model to align the source and target sentences and generate accurate translations.

3. Attention Mechanism in the Decoder:

All of the basics of the Attention mechanism in the Encoder will still be used in the Decoder Attention mechanism.

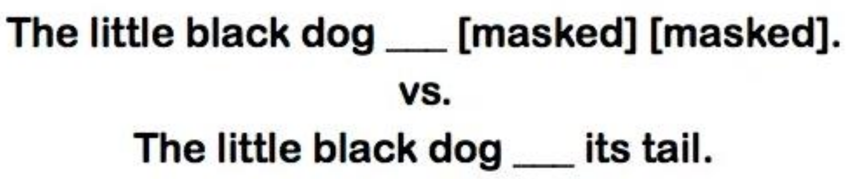
The decoder, however, uses what is called ***masked multi-head self-attention***. This means that some positions in the decoder input are masked and thus ignored by the self-attention layer.

The reason some positions in the decoder attention mechanism are masked is to prevent the model from "cheating" by attending to future positions in the output sequence.

In the decoder of a Transformer model, the output sequence is generated one position at a time, with each position being generated based on the previous positions. When computing the attention scores between a given position in the decoder and the encoder outputs, it's important to ensure that the decoder can only attend to information that was available at the time of generation, and not to any information that comes after the current position.

To achieve this, the decoder attention mechanism typically includes a masking step that sets the attention scores for any future positions to a very large negative value. This effectively excludes those positions from consideration, as the resulting softmax over the attention scores will assign a near-zero weight to any future positions.

By masking future positions in the decoder attention mechanism, the model is forced to rely only on information that was available at the time of generation, which encourages it to generate outputs that are more coherent and realistic.



The little black dog ___ [masked] [masked].

vs.

The little black dog ___ its tail.

In the first sentence (masked), the next word is far more difficult to predict than in the second sentence (unmasked). The words “its tail” make it clear the word to predict is probably “wiggled”.

Task 3

References

1. https://builtin.com/data-science/gradient-descent

2. https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/

3.
https://wiki.tum.de/display/lfdv/Adaptive+Learning+Rate+Method#AdaptiveLearningRateMethod-AdaptiveLearningRateMethod