

Vision Transformers from Scratch Project Report

Akash Kawle
Prithvi Tambewagh
Sneha Singh

August 2024

Acknowledgements

We are extremely grateful to our mentors - Aryan Nanda and Abhinav Ananthu for their guidance and support throughout the course of this project.

We also express gratitude towards SRA-VJTI for their support as well as organization of 'Eklavya - 2024' and providing us with the opportunity to work on this project.

Akash Kawle
Prithvi Tambewagh
Sneha Singh

Overview

Transformers are State Of The Art Architectures, especially in the field of Natural Language Processing. The speciality of transformers is that it uses 'attention' mechanism. This mechanism helps the model to pay the optimum amount of attention to each of the words in the whole sentence. This helps to understand the sentence with the proper context.

Our project explores how we can use the transformer architecture for the task of image captioning, by using a special variant of transformers, which is - Vision Transformers. Vision transformers take image as input and provide its caption as output.

Chapter 1

Linear Algebra

Linear algebra is a mathematical subject that studies vectors, vector spaces (also called linear spaces), linear transformations and systems of-linear equations. Linear Algebra forms the basis of the operations, from the very basic to very advanced level, in Deep Learning.

Some of the useful concepts in Linear Algebra include :

1. **Vectors** : Quantities with direction and magnitude. A vector is a point in space, and you can add vectors together or multiply them by scalars (numbers).
2. **Matrices** : Matrices are units that can transform a vector from one vector space to another dimensional space by means of matrix multiplication.

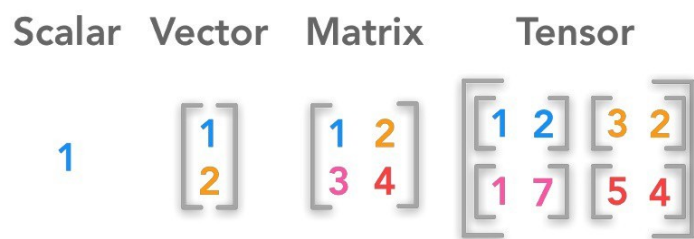


Figure 1.1: Different types of fundamental data structures

Arrays are the form in which computers interpret matrices. In NumPy, matrices are thus represented by arrays. In terms of machine learning, the arrays are termed as 'Tensors'. Tensors provide built-in support for complex operations, automatic differentiation, and are designed for efficient computation on specialized hardware.

Chapter 2

Neural Networks and Deep Learning

2.1 Neural Networks :

Neural Networks are one of the fundamental architectures in the field of Machine Learning. They can be considered equivalent to the architecture of neurons in the human brain, hence its name. They consist of densely interconnected nodes, termed as 'Neurons', which get activated according to the neurons sending them data and the nature of their interconnections and finally, provide a means to make an inference on the basis of these activations.

2.2 Neuron :

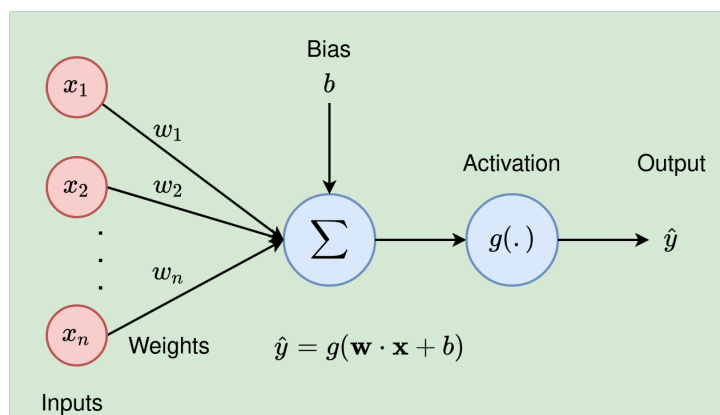


Figure 2.1: Neuron

A 'Neuron' is a single node of a neural network, which performs the computation in the neural network. Basically it performs 3 computations : a matrix multiplication i.e. computation of a linear function, using 'weights' and 'biases' acting on the input 'x' which produces a quantity 'z' and next, computation of a non-linear function on 'z'.

2.3 Weights and Biases :

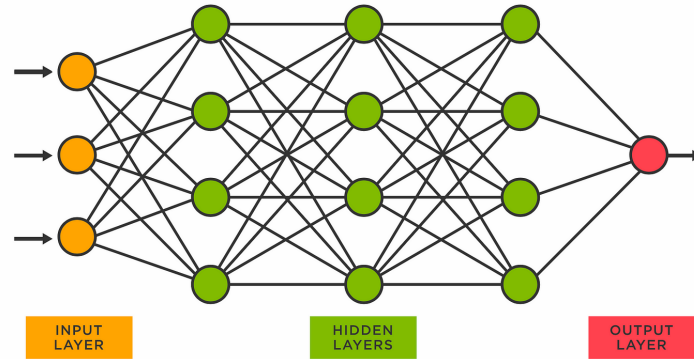


Figure 2.2: Neural Network

Every neuron in a neural network is connected to every other neuron in the preceding and succeeding layers, resulting in a very densely connected network. When data passes along any of these interconnections, it gets multiplied by a specific scalar termed as ‘weight’ and then another scalar termed as ‘bias’ is added to this product. Then the non-linear function operates on this weighted sum to produce the ‘activation’ of the neuron from which the data originated.

To ease these operations, which are performed by the computer in large quantity and thus speed up the process, the weights, biases and the output from each neuron of a layer are condensed to form matrices. thus these operations reduce to operations of linear algebra like matrix multiplication and element-wise operation (for non-linear function).

2.4 Operations performed in every Neuron :

The operations performed in every neuron are :

$$z = Wx + b$$

$$A = k(z)$$

2.5 Activation Functions :

Activation functions are an integral part of a neural network. In essence, each layer of the neural network just operates using a linear function over its inputs. This can result in the output layer just computing a linear function over the inputs of its previous layer's activations, which boils down to the computation of a linear function on the inputs of the input layer. This makes the hidden layers of the neural network redundant. Hence, introducing non-linearity in the neurons helps the neural network learn the correlation between inputs and outputs effectively. Different activation functions can be employed in different layers of the neural network.

Some of the prominent activation functions utilized in neural networks are :

1. **Sigmoid activation function :-** The sigmoid activation function takes any real number as its input and provides a number in the range $[0, 1]$ as its output. Sigmoid function is particularly useful in problems like binary classification tasks (classification of image into 2 categories : cat or dog, prediction of whether it will rain tomorrow or not, etc.) If the value of activation is greater than 0.5, the activation can be considered to represent class with label 1; else, it can be considered to represent the other class, with label 0.

$$A = \sigma(z)$$

$$z \in \mathbb{R}, A \in [0, 1]$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

2. **tanh activation function :** 'tanh' i.e. hyperbolic tangent function, is another popular activation function. It takes any real number as its input and provides a number in the range $[-1, 1]$ as its output. The tanh function is zero-centred in nature; hence, it can result in efficient training.

$$A = \tanh(z)$$

$$z \in \mathbb{R}, A \in [-1, 1]$$

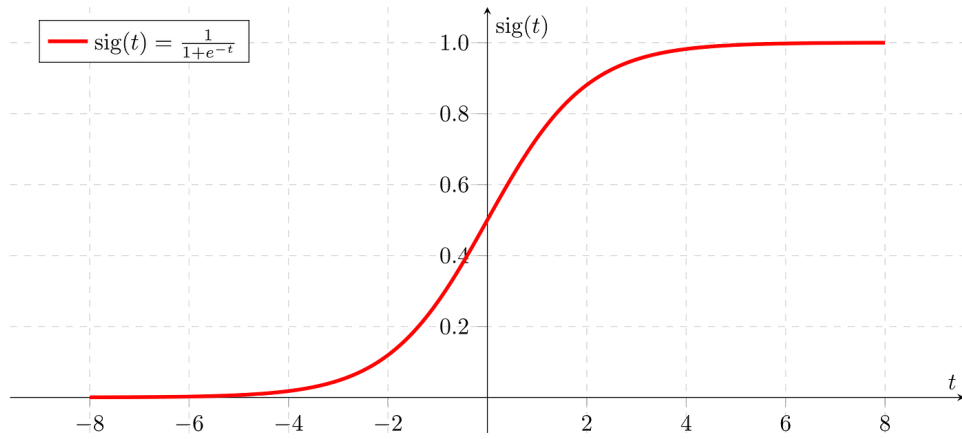


Figure 2.3: Sigmoid Activation Function

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

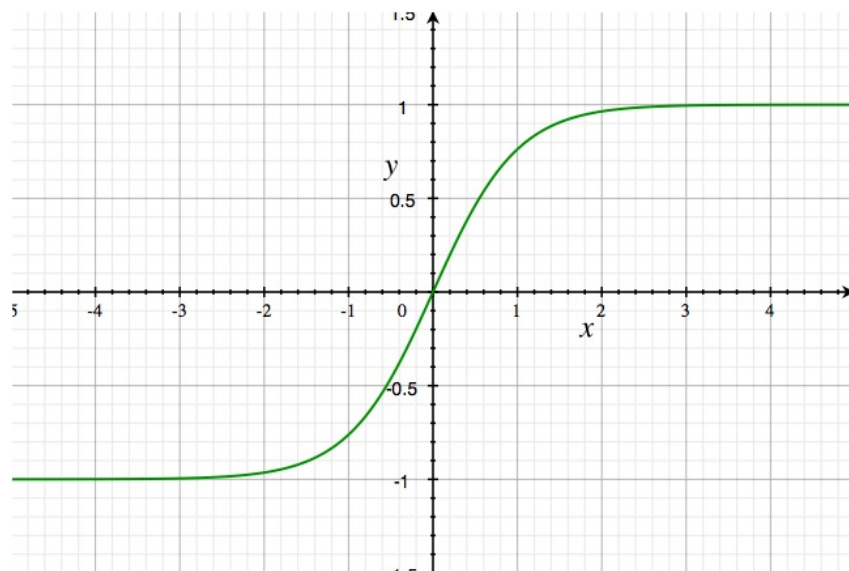


Figure 2.4: tanh Activation Function

3. ReLU activation function : The sigmoid and tanh activation functions have a major flaw in context of Deep Neural Networks : for very large values of z , value of A diminishes slowly during gradient descent (Chp. 2 Sec. 2.7) which slows down the learning significantly. So, another function, known as ‘Rectified Linear Unit’ (i.e. ReLU) is used instead.

$$A = \text{ReLU}(z)$$

$$z \in \mathbb{R}, A \in [0, z]$$

$$ReLU(z) = \max(0, z) = \begin{cases} 0 & z \leq 0 \\ z & z > 0 \end{cases}$$

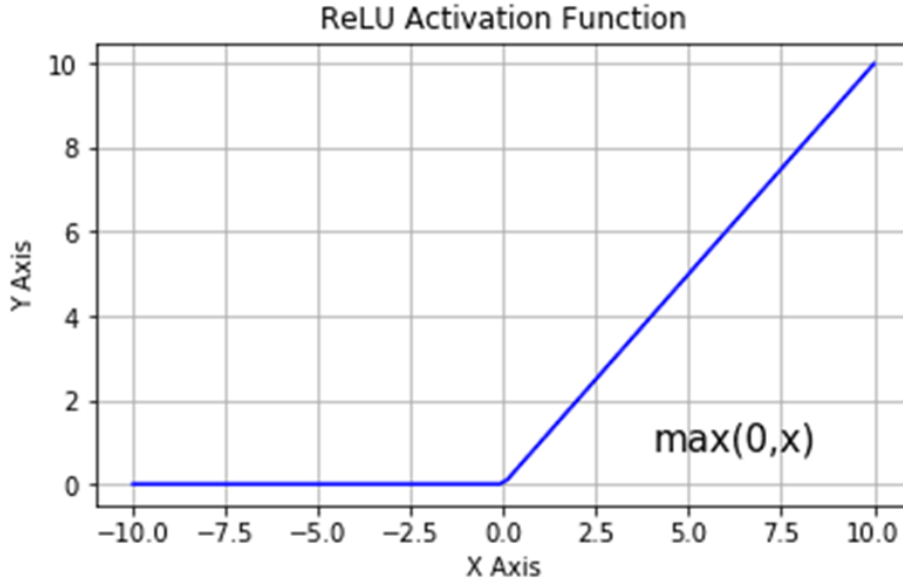


Figure 2.5: ReLU Activation Function

Leaky ReLU activation function : Another version of ReLU function is the Leaky ReLU function, which also gives small negative numbers as the output for input being negative numbers. Hence, the problem of gradient descent. Neurons can ‘die’ and stop learning if they output zero for all inputs (due to negative values in the data). This occurs because the gradient is zero for negative inputs. Leaky reLU solves this issue.

$$A = ReLU(z)$$

$$z \in \mathbb{R}, A \in [0, z]$$

$$ReLU(z) = \max(kz, z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

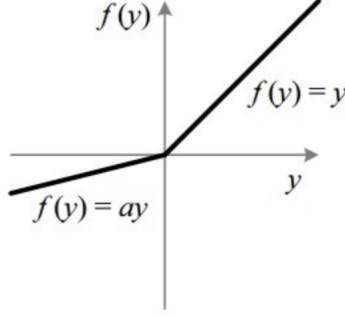


Figure 2.6: Leaky ReLU Activation Function

Softmax Activation Function : Softmax is an activation function which converts a set of numbers to another set, in terms of probabilities. Thus, now the data elements all sum to 1.

It is represented by the equation :

$$\text{softmax}(z) = \frac{e^z}{\sum_{k=1}^m e^k}$$

2.6 Loss function and Cost function:

In supervised learning, the model is given both the inputs as well as expected outputs as input. It then learns the correlation between the corresponding inputs and expected outputs and finally learns to predict output when an unknown input (of same type as used in training) is given to it. We compare how well the model performs on the provided inputs, where it produces certain output which are termed as ‘predictions’ (\hat{y}) and compare them with the expected outputs (y) by providing \hat{y} and y as input to a function, termed as ‘Loss’ function and denoted by ‘ L ’. For instance, logistic regression has the loss function defined as :

$$L(y^{(i)}, \hat{y}^{(i)}) = -y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

The loss function is calculated for every batch/example on which the model trains. The cost function, denoted by ‘ J ’ is the average of the loss function.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

Here, the parameters ‘ m ’, ‘ w ’ and ‘ b ’ represent the total number of examples, weights and biases in the network. We aim to reduce the cost

function by changing the values of weights and biases via gradient descent, so as to tune them to optimal values. As a neural network has several weights and biases, so the cost function is a higher dimensional function.

2.7 Gradient Descent :

Gradient Descent is the fundamental process in Machine Learning. It is based on calculus. The “Gradient” of a function at a point gives the direction of increasing slope on the function, at that point. To minimize the function, we intend to approach the point where the slope is zero. hence, we can calculate the gradient of the cost function at various points and traverse in the direction opposite to it, so as to reach the minima of the cost function. This minima will have the optimal values along the dimensions of various weights and biases.

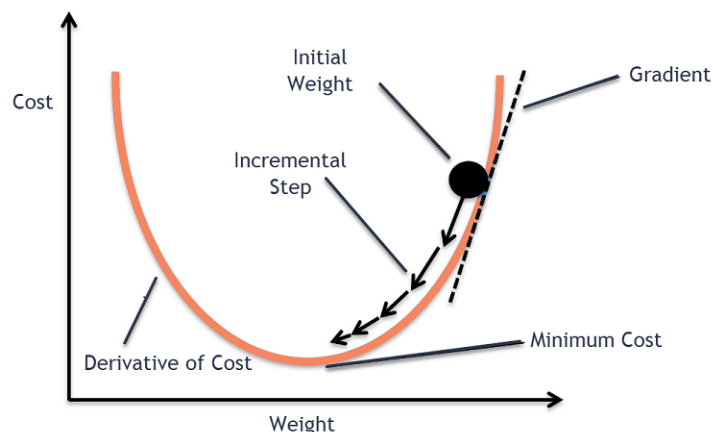


Figure 2.7: Schematic diagram : Gradient Descent

Initially we choose random values of the weights and biases; they get tuned as gradient descent progresses. This is the secret behind the process of learning of neural networks. For a particular weight matrix ‘W’ and bias ‘b’ of a particular layer, the parameters are updated after gradient descent as :

$$W = W - \alpha \frac{\partial J}{\partial W}$$

$$b = b - \alpha \frac{\partial J}{\partial b}$$

Here, ‘ α ’ i.e. Learning Rate, is a hyperparameter (Chp. 3 Sec. 3.1).

2.8 Vectorization :

Vectorization are features (in-built) available in languages like Python. They help us to increase computation time significantly while providing the same result. This is because using vectorization, like the 'dot' function in NumPy library, we are able to take advantage of SIMD (Single Instruction Multiple Data) so that processes run parallelly and computation speed increases, as opposed to working of for loops, which kind of run sequentially, for each iteration, thus not taking complete advantage of SIMD and parallel computing.

For instance, we initialize 2 random arrays - a and b, of 1 million dimensions, and store their dot product in c, using both methods - vectorization and for loops. It is observed that vectorization is clearly 50x (or even more) faster than using for loops.

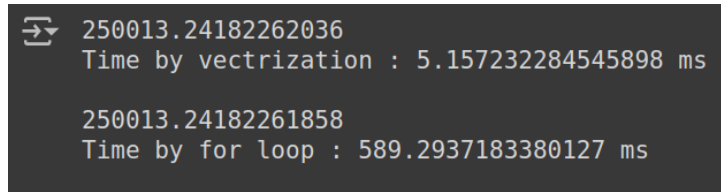
```
import numpy as np
import time as t

a=np.random.rand(1000000)
b=np.random.rand(1000000)
tic=t.time()
c=np.dot(a,b)
tac=t.time()
print(c)
print("Time by vectrization : " + str(1000*(tac-tic)) + " ms\n")

c=0;
tic=t.time()
for i in range(1000000):
    c+=a[i]*b[i]
tac=t.time()
print(c)
print("Time by for loop : " + str(1000*(tac-tic)) + " ms\n")
```

2.9 Deep Neural Networks :

Neural Networks that contain a very large number of layers, are termed as 'Deep Neural Networks'. Such neural networks are used for high-end



```
↔ 250013.24182262036  
Time by vectrization : 5.157232284545898 ms  
  
250013.24182261858  
Time by for loop : 589.2937183380127 ms
```

Figure 2.8: Vectorized v/s non-vectorized approach

applications, like scientific research, autonomous vehicles, natural language processing, etc. They employ all the same principles as those of shallow neural networks (neural networks with nominal number of layers). Their architecture is also very same.

Chapter 3

Finer aspects of Deep Neural Networks

3.1 Bias and Variance :

When a model has a nominal train set error and a dev set error of around the same about as train set error, we can classify it as a case of ‘high bias’. In this case, the model has not properly learnt the relationship between the inputs and expected outputs provided. High bias can be reduced by training the data on a larger network or by improving the overall architecture of the neural network.

When a model has very low train set error but a relatively considerable dev set error, in that case the model is said to have high variance. It indicates that the model has almost perfectly been trained on the train dataset but is not able to generalize on the dev dataset (or for instance an input it has not been trained with. High variance can be reduced by techniques like regularization, early stopping, learning rate decay, etc.

We aim to have low bias and low variance for our model.

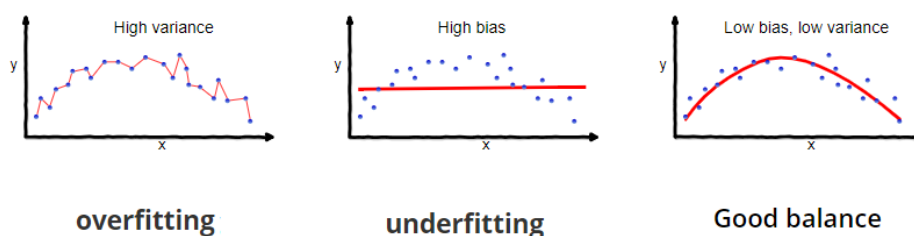


Figure 3.1: Bias and Variance

3.2 Regularisation

Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function, discouraging the model from assigning too much importance to individual features or coefficients. There are various methods of regularization :

1. **Dropout Regularization :** In Dropout Regularization, some of the neurons of a layer are randomly shut off, i.e. their activations aren't considered for passing on to the next layer. This avoids the model to rely on a specific feature too much and thus reduces normalization.

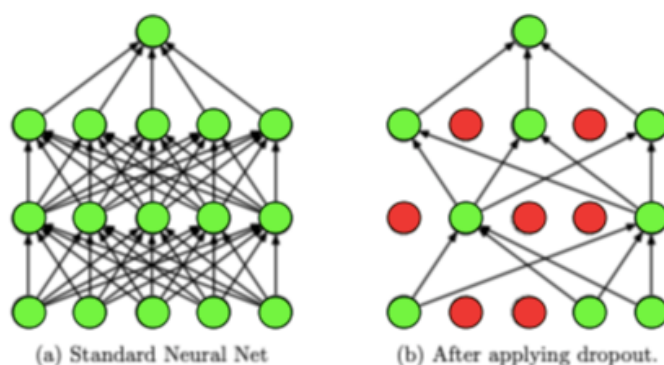


Figure 3.2: Dropout regularization

2. **L1 and L2 regularization :** L1 and L2 regularization are techniques used in machine learning to prevent overfitting by adding a penalty to the loss function. They help to constrain the model's complexity, ensuring it generalizes better to new data.
3. **Data Augmentation :** In data augmentation, the images in a dataset are operated by various operations, like rotation, mirroring, etc. so as to produce new images for the model to train on and generalize the model.
4. **Data Normalization :** In data normalization, the parameters on different scales are converted to those on similar scales, so as to speed up gradient descent.

3.3 Hyperparameters :

Hyperparameters are the parameters that ultimately control the parameters i.e. $W[l]$ and $b[l]$ i.e. weights and biases of a layer 'l'. Hyperparameters

Data augmentation

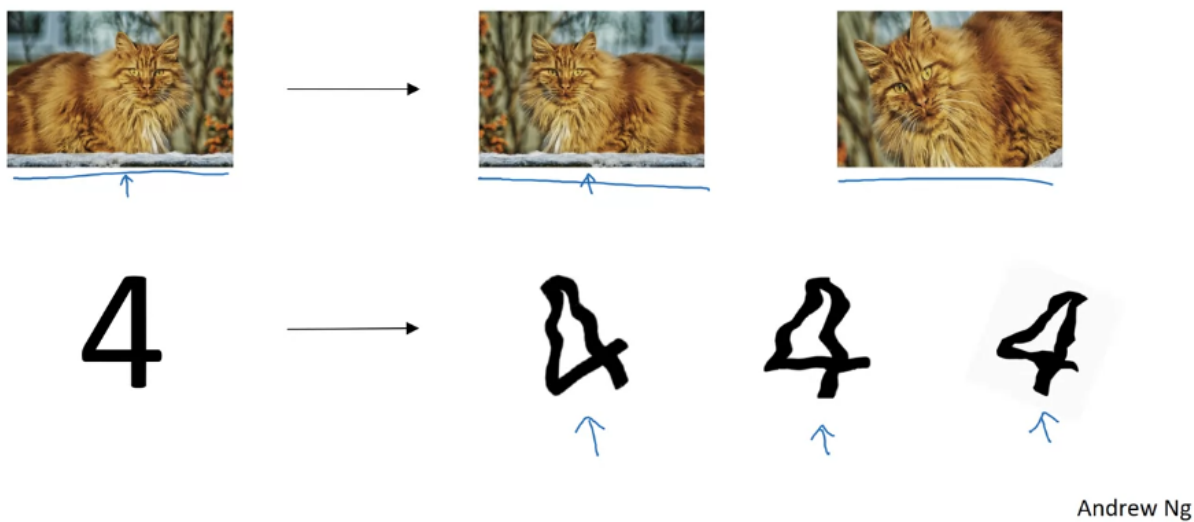


Figure 3.3: Data Augmentation

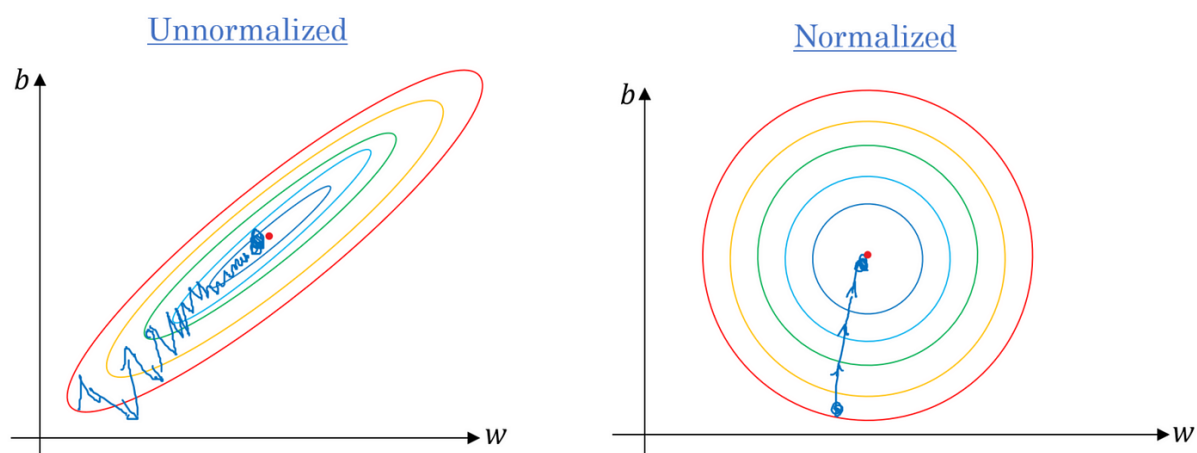


Figure 3.4: Data Normalization

include learning rate (which could, in turn, determine the number of iterations in gradient descent), number of hidden layers, number of hidden units, etc. The best value of hyperparameters can change with time.

Some of the commonly tuned hyperparameters include :

1. Learning Rate (α)
2. Number of iterations
3. Number of hidden layers
4. Number of hidden units (neurons) in various layers
5. choice of activation function

3.4 Hyperparameter Tuning :

There are several hyperparameters to tune at a time. They are organised according to the priority of the extent of tuning they require and then tuning is carried out.

One way to find the best values, for instance out of 2 hyperparameters, is to plot some random points in a 2-D space, where each axis denotes each of those 2 hyperparameters. Choosing uniform points leads to inaccuracy due to redundancy of any of the hyperparameters. Hence we choose random points. Same idea can be extended for multiple hyperparameters. Also, after we find a best point in this grid, optionally, we can choose to repeat this process for a small region around it. This is known as going from coarse to fine.

Some of the hyperparameters like the learning rate or momentum have very small values, less than 1. In that case we can shift to logarithmic scale for selecting points with better distribution in small intervals.

Chapter 4

Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a class of deep learning models used almost exclusively for the processing structured grid data (such as images). CNNs - sometimes referred to as ‘convnets’ - use principles from linear algebra, particularly convolution operations, to extract features and identify patterns within images. Convolution neural network (CNNs) are a particular CNNs are composed of building blocks that learn spatial hierarchies features using backpropagation, automatically and adaptively.

4.1 Structure of a CNN :

A CNN consists of a few layers in each of which the operations of convolution and pooling are performed.

1. **Convolution** : In convolution, a small sized kernel is used. An element-wise multiplication and summation is carried between the kernel and elements of the image of the same size as kernel to produce a feature map . This operation is termed as ‘convolution’.

Convolution results in a new image (assumed to be a square image formed from a square kernel and a square input image) the dimensions can be stated as :

$$d = \frac{n + 2p - f}{s} + 1$$

2. **Padding** : The convolution operation on an image results in reduction of the size of the image as it traverses through the various convolution layers. To prevent this, ‘padding’ is done along the edges of the image. This increase in dimensions of the image reduces or even nullifies the unwanted reduction of the size of the image. Padding is done by surrounding the image by dark pixels, i.e, pixels with value ‘0’.

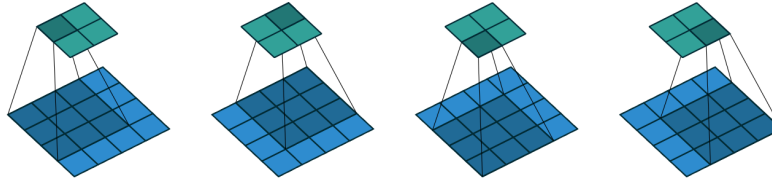


Figure 4.1: Convolution

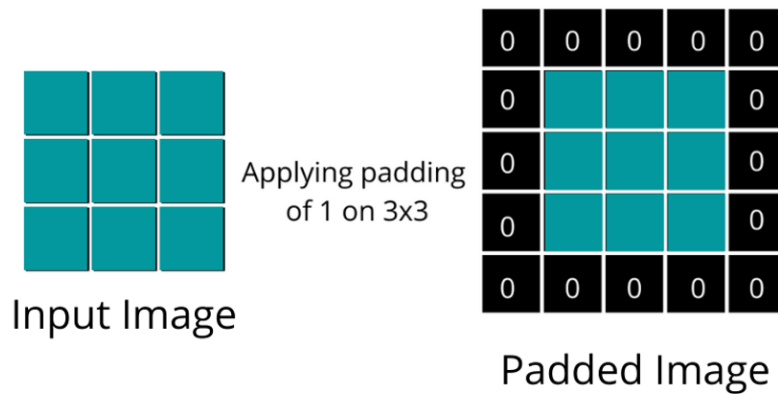


Figure 4.2: Padding

3. **Pooling** : After convolution, so as to reduce the computations, the feature map so produced is operated by ‘Pooling’ operation, which further reduces the size of the feature map. There are various types of pooling :
 - (a) **Max Pooling** : Selecting the maximum value from each region of a specified size of the feature map.
 - (b) **Average Pooling** : Selecting the average value of each region of a specified size of the feature map.
4. **Fully Connected Layers** : The features extracted by the convolution and pooling layers are passed to fully connected layers i.e. dense layers, as in neural networks to make accurate predictions.

CNNs offer several advantages, like :

1. **Parameter Sharing** : In CNNs, the parameters used are small sized kernels. The same kernel is shared by all the pixels of the input image. Hence, this results in a significant decrease in the number of parameters and operations.
2. **Dimensionality Reduction** : The Pooling operations result in lower dimensional feature maps, due to which final computations become easier and more time as well as resource efficient.

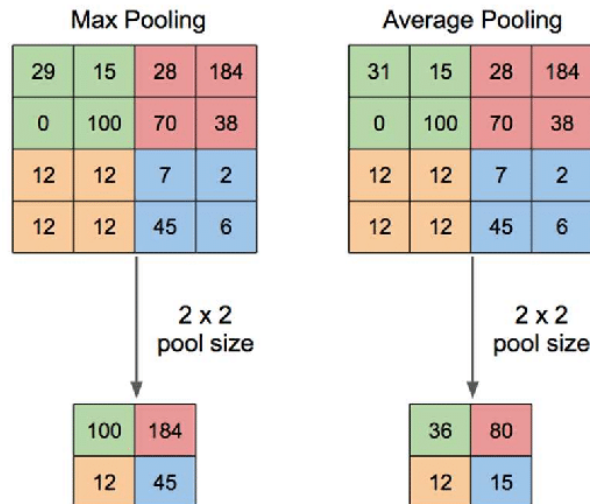


Figure 4.3: Pooling

3. **Scalability** : We can make very deep CNNs using Residual Networks which makes it smoother to form CNNs which can work effectively.
4. **Spatial Invariance** : Convolution and pooling operations make CNNs translationally and distortion invariant, i.e. they can provide same predictions for augmented images.

Some of the prominent CNN architectures include :

1. **LeNet-5**
2. **AlexNet**
3. **VGG-16**
4. **ResNet**
5. **Inception Network**
6. **MobileNet**
7. **EfficientNet**

Chapter 5

Sequence Models

Sequence models are models that process sequences to yield meaningful results. They are particularly useful for applications like natural language processing, language translation, forecasting based on times series like stock prices prediction, weather forecasting, etc.

There are 3 main types of sequence models :

1. **Recurrent Neural Networks (RNNs)** : Recurrent Neural Networks (RNNs) are another type of neural networks made for handling sequential data. Unlike the traditional feed forward neural networks that hold no memory, RNN have connections back on themselves which gives them some form of memory in their function about earlier input. RNNs keep a hidden state which is updated at each time step when handling the sequences. The hidden state is a way of remembering some information from previous time steps.

However, when the gradients backpropagated through many time steps can become small (vanish) during training. If, however new input occurs after only just one transformation step (towards the right), it may change how processing performs during training. Marginal increase exaggerates gradients preserving information to flash over many time steps. This makes the network difficult learning long-time dependencies of them all. In turn, gradients may also get very big (explode) which can lead to training instability.

RNN works according to the following equations :

$$a_t = k(W_a a_{t-1} + W_x x_t + b_a)$$

$$y_t = W_y a_t + b_y$$

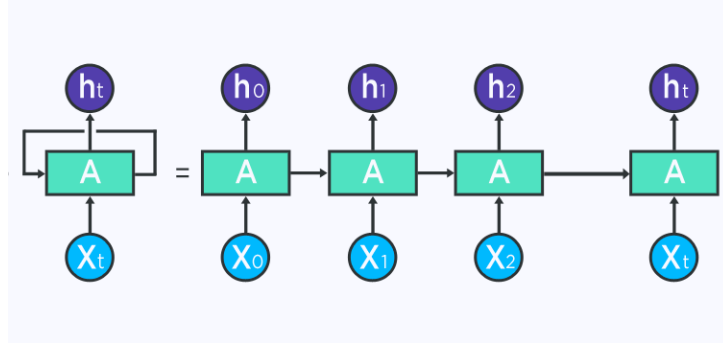


Figure 5.1: Recurrent Neural Network

2. **Gated Recurrent Unit (GRU)** : Gated Recurrent Unit (GRU) is a type of RNN that can process sequential data, sequences in general (time series or text) more effectively. It is tasked with retaining useful information across long sequences.

GRUs are characterized by gates, which make it different and efficient from RNNs. It has 2 types of gates :

- (a) **Update Gate** : Determine what information is to be kept, and which one should not (previous memory) in the past information, so how much to update with new input.

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$

- (b) **Reset Gate** : It determines how much of the past knowledge should be reset or thrown away when calculating new state.

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$

It also has 2 types of hidden states :

- (a) **Candidate State** : New information the network could add to its memory.

$$\tilde{h}_t = \tanh(W_h[r_t * h_{t-1}, x_t] + b_h)$$

- (b) **Final Hidden State** : It is This merges the old memory and new candidate information to generate an updated memory.

$$h_t = (1 - z_t) * h_{t-1} * z_t \tilde{h}_t$$

At each step in a sequence we make decision about what to keep from the past and new information, this is implemented using gates, which decide when something enters your system. This makes the network pay attention to important parts and ignore less information.

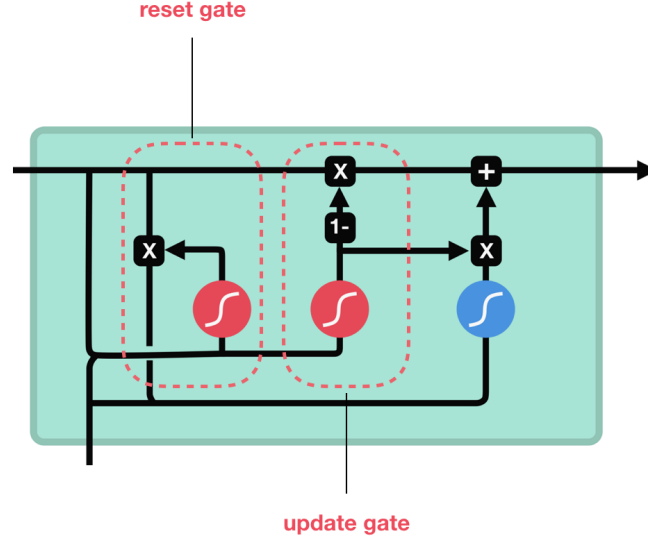


Figure 5.2: Gated Recurrent Unit

The former is used to calculate the gates, while updating memory and passing it on. This updated memory will then be used to process the next part of the sequence.

5.1 Long Short-Term Memory (LSTM) :

Long Short-Term Memory is a particular kind of neural network capable of processing entire sequences — whether sentence or word. LSTMs differ from basic RNNs in their ability to learn long term dependencies, which are a necessity for time series predictions when learning the inputs based on ordering and context.

LSTMs includes something called a ‘memory cell’ which controls what information is stored in the long term, and when it should be forgotten. Meanwhile, some cells are capable of remembering information and then determining whether to remember it further or not.

LSTMs use gates to regulate the flow of information, Gates The gates are mainly 3 types:

- (a) **Forget Gate** : Decides which information you will no longer need anymore and removes it from the memory cell.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f]$$

- (b) **Input Gate** : Controls which new information should be added to the cell state.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i]$$

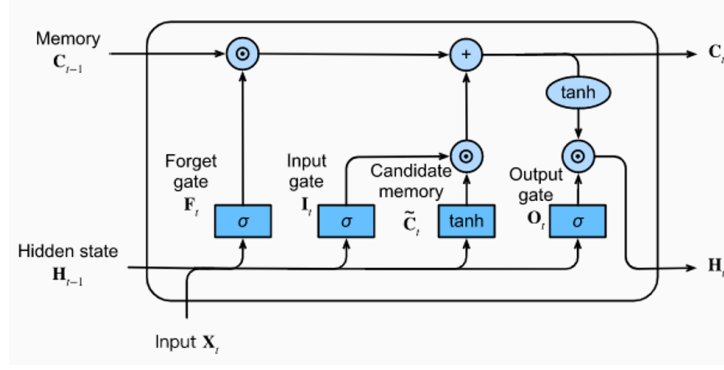


Figure 5.3: Long Short-Term Memory

(c) **Output Gate** : Determines what the next hidden state should be, based on the cell state.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o]$$

Cell State: This is the inner memory of LSTM which changes in each step (here at every piece of data) that the processing unit process. The best way to think of memory cells is as something that you want LSTM to remember for the long time, or a continual Memory.

Candidate Cell state : The candidate cell state provides potential new information to be added to the cell state.

$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c]$$

Hidden State : The hidden state is the final output of the LSTM cell for the current time step and is used for subsequent computations and predictions.

$$h_t = o_t * \tanh(\tilde{C}_t)$$

In general, LSTMs are best suited for language modeling problems speech recognition and more generally wherever understanding context and time-aware is of importance.

Chapter 6

Transformers and Vision Transformers

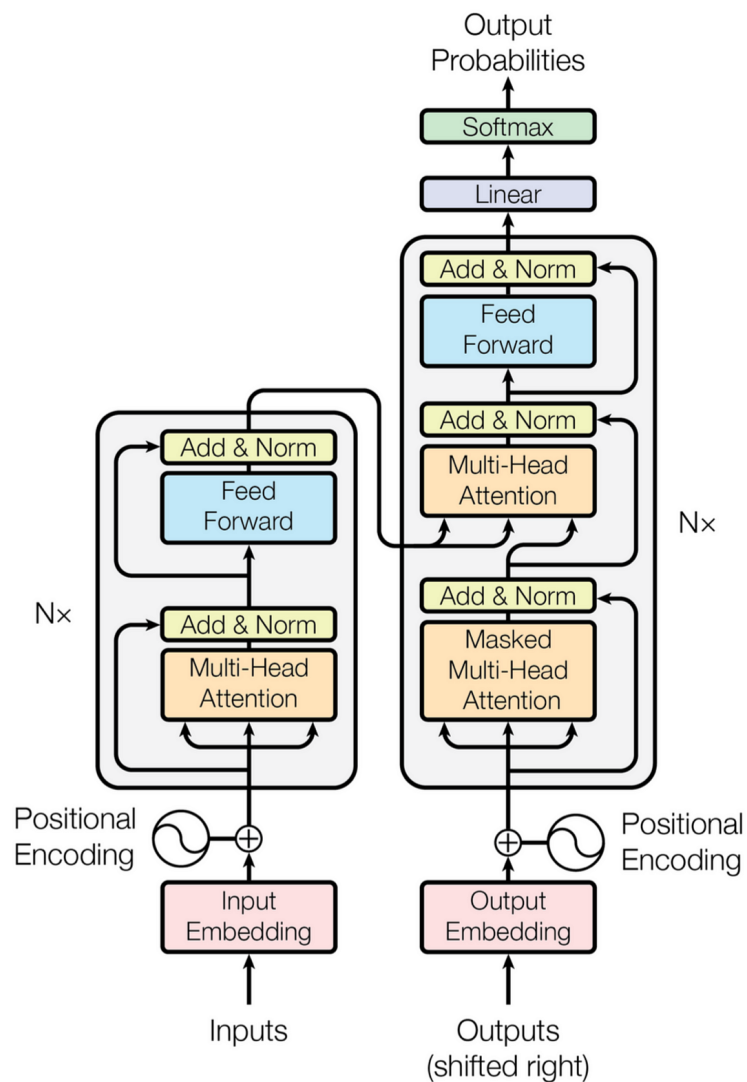


Figure 1: The Transformer - model architecture.

6.1 Transfromer :

Transformers are a type of neural network architecture designed to handle sequences of data, especially in natural language processing. Transformers are unique and different from other architectures, as they employ a completely different approach, in which ‘attention’ is paid to each word in the sentence, in accordance with its actual expected attention.

1. **Self-Attention Mechanism** : It allows each word in a sequence to focus on different words to understand their context better. Each word calculates how much attention it should give to every other word in the sequence. This helps in understanding the relationship between words, regardless of their distance in the sequence.
2. **Queries, Keys and Values** : They facilitate the self-attention mechanism. By comparing queries with keys, the model decides how much attention to pay to each value from other words. Here,
 - (a) **Query** : Represents the current word’s focus or question.
 - (b) **Key** : Represents potential information from other words.
 - (c) **Value** : Contains the actual data or content of other words.
3. **Multi-Head Attention** : It enhances the model’s ability to focus on different aspects of the sequence simultaneously. The attention mechanism is split into multiple “heads,” each focusing on different parts or aspects of the sequence. The outputs of these heads are then combined to form a comprehensive understanding.
4. **Positional Encoding** : It provides information about the order of words in the sequence. Since transformers process words in parallel rather than sequentially, positional encodings are added to the word embeddings to retain the sequence information.
5. **Feed-Forward Network** : It refines the information after attention processing. After the self-attention step, each word’s representation is passed through a feed-forward neural network to further process and refine it.
6. **Layer Normalization and Residual Connections** : It stabilizes and improve the training of the model. Residual connections (short-cuts) add the input of a layer to its output, and layer normalization helps to stabilize and speed up training by normalizing the input of

each layer. Residual connections are important for preventing problems of vanishing or exploding gradients.

7. **Encoder and Decoder Structure :** It handles different aspects of sequence processing and generation. The Encoder processes the input sequence and creates an internal representation. The Decoder uses this representation to generate the output sequence, such as translating text or generating a response.

In summary, transformers excel at managing and understanding complex sequences by using self-attention to dynamically weigh the importance of different parts of the input data. They rely on parallel processing and multiple layers of attention and feed-forward networks to build rich, contextual representations of the data.

6.2 Vision Transformers (ViT) :

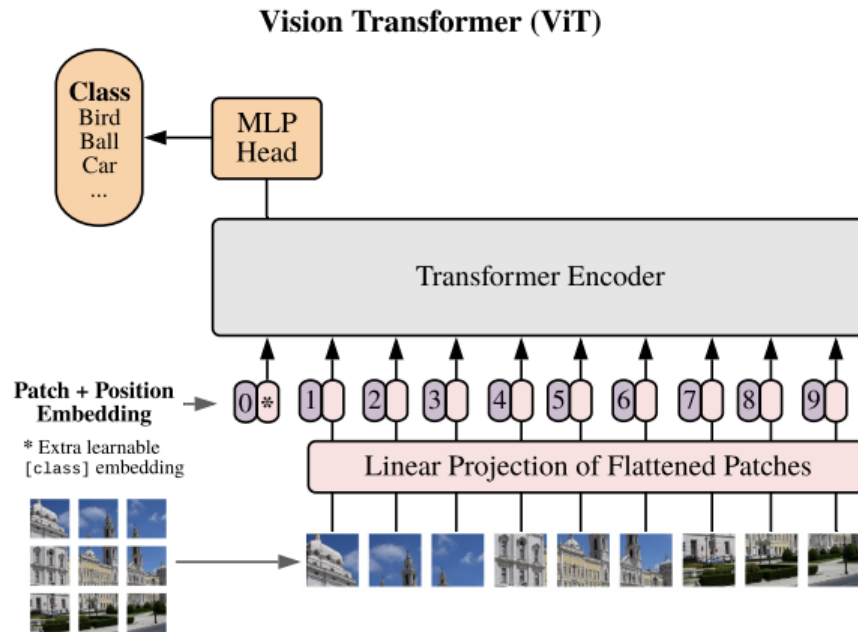


Figure 6.1: Vision Transformer

Vision transformer is a special type of transformer architecture, which performs the task of image classification. Here, instead of providing the transformer input in the form of words, the input is provided in the form of patches of images. These inputs are operated on by the encoder to produce a set of activations, equal to the number of image classes. Using

softmax activation function, the prediction of the image class is made.

Here, a token termed as ‘CLS’ token plays a very important role. It is prepended to the sequence of embeddings of the image patches. Its state at the output of the transformer encoder serves as the image representation. After passing it through feed forward layers, we can get image class.

6.3 Our Project - Vision Transformers from Scratch

Our project aims to implement vision transformers from scratch. We implemented it using Pytorch.

We take images of varying size and use convolution and a kernel of size 16, to convert it into a 14*14 image. This is converted to an embedding of 768 dimensions (thrice the size required by the embedding in ViT. A ‘CLS’ token is also prepended to the embeddings.

This embedding is passed into the multi-head attention block. Here, it is acted upon by the query, key and value matrices together. Then, the vector is passed through a fully connected layer to reduce it to 256 dimensional size. Then we carry out the scaled dot product attention and calculate attention scores. Hence, we get the attention output. Then we pass the embedding through the Add and Norm layer and Feed Forward layer according to the model structure. This way the encoder block is created.

The decoder block is also created in a similar manner using the classes of attention and other classes as per the structure.

The encoder block and decoder block are repeated ‘n’ times.

Chapter 7

Models

We implemented several models during the course of this project. The results of these models are summarized here

1. **Handwritten Digit Recognizer from MNIST Database from scratch using NumPy** : Accuracy : 93.95

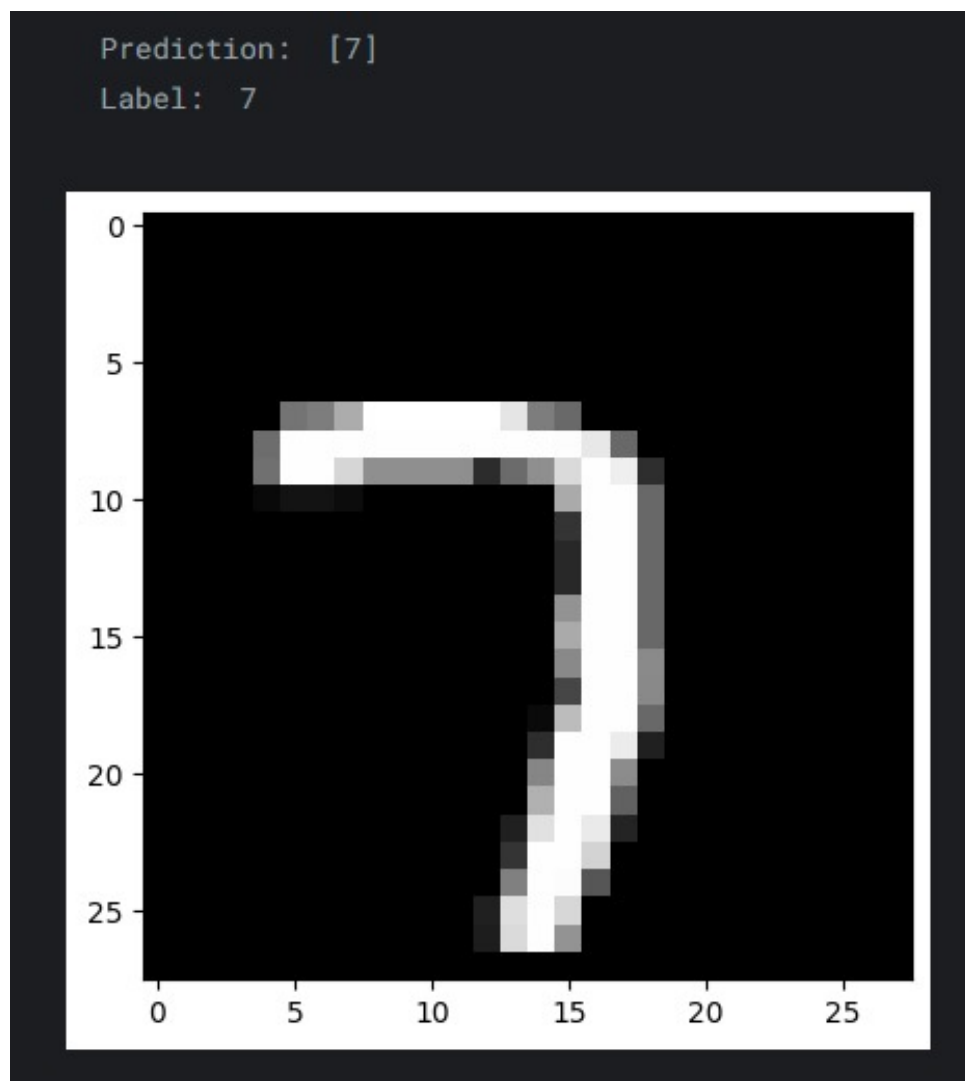


Figure 7.1: Recognized Digit by model

2. Hand Gesture Recognition model from scratch using NumPy

: Accuracy : 95.39%

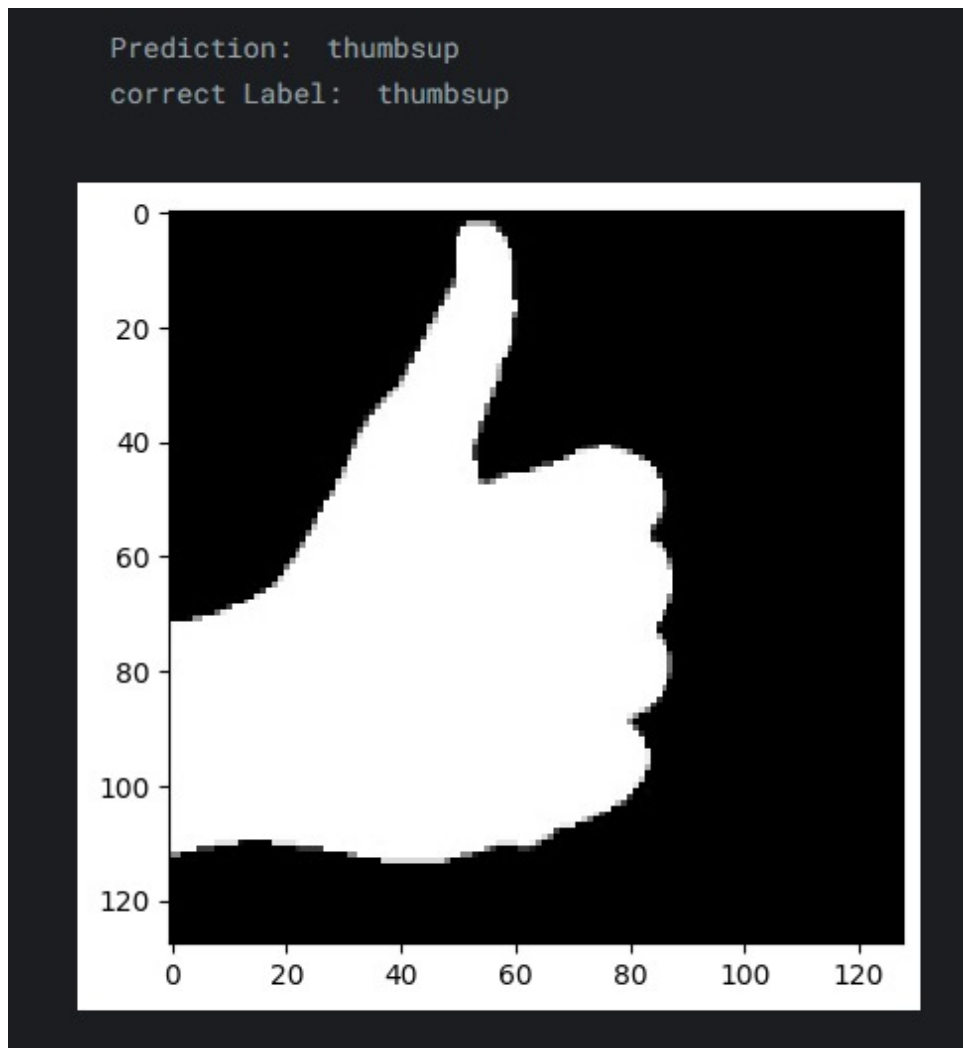


Figure 7.2: Hand Gesture Recognized by model

3. Stock Prices Prediction model based on Yahoo yfinance dataset using Tensorflow : Mean Absolute Error Percentage : 9.38%

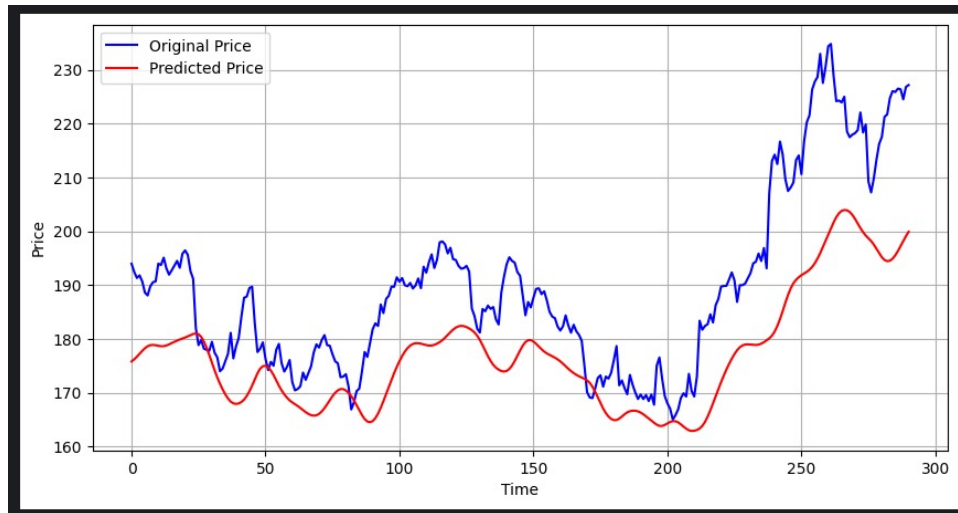


Figure 7.3: Stock Prices Prediction

4. CNN+LSTM Model for Imge captioning using TensorFlow : The CNN + LSTM model achieved a BLEU-1 score of 0.553085 and BLEU-2 score of 0.333717

```

-----Actual-----
startseq tennis player retrieving ball from the courts surface endseq
startseq boy reaching for tennis ball on court endseq
startseq man bending down to pick up tennis ball on blue tennis court endseq
startseq man with racquet and ball bending over endseq
startseq the young player is picking tennis ball up off the court endseq
-----Predicted-----
startseq tennis player playing tennis on tennis court endseq

```



Figure 7.4: Result : CNN+LSTM Model

References

1. **Neural Networks and Transformers course by 3Blue1Brown (YouTube)** : https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
2. **Short Tutorial for implementation of neural network from scratch (YouTube)** : <https://www.youtube.com/watch?v=w8yWXqWQYmU>
3. **Short Tutorial for Pandas library (YouTube)** : <https://www.youtube.com/watch?v=G2iVj7WKDFk>
4. **Research Paper on Residual Network** : <https://arxiv.org/abs/1512.03385>
5. **Research Paper on Transformer Architecture** : <https://arxiv.org/abs/1706.03762>
6. **Research Paper on Vision Transformer** : <https://arxiv.org/abs/2010.11929>