Stockholms universitet

Enhancing Image Classification with a Hybrid CNN-Transformer Model: A Comparative Study of ResNet-18 and a Modified Architecture

Chinmaya Mathur

Masteruppsats i matematisk statistik
Master Thesis in Mathematical Statistics

Masteruppsats 2025:1
Matematisk statistik
Februari 2025

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Stockholm University

Mathematical Statistics
Stockholm University
Master Thesis **2025:1**
http://www.math.su.se

Enhancing Image Classification with a Hybrid CNN-Transformer Model: A Comparative Study of ResNet-18 and a Modified Architecture

Chinmaya Mathur*

February 2025

Abstract

In this thesis, we propose a Hybrid model that integrates the strengths of Convolutional Neural Networks (CNNs) and transformer encoders to enhance image classification. We specifically modify the ResNet-18 by replacing its 4th block with a transformer encoder which includes a multi-head self-attention layer and a position-wise feedforward network. This modification aims to leverage the transformer's ability to capture long-range dependencies and improve the feature extraction capability of the model.

On evaluating the performance of both the models on the CIFAR-10 dataset, we see that the Hybrid model performs slightly better than ResNet-18. The classwise accuracy analysis shows that the Hybrid model performs better for several classes like "airplane", and "dog" but shows a decrease in accuracy for classes like "cat". To understand the impact of architectural modification, we compare the weights of the first 3 blocks using a quantile-quantile (QQ) plot. The analysis shows that the weights remain largely similar in distribution but the magnitude changes with the Hybrid model having bigger weights.

We further analyze the significance of the changes in classwise accuracies using the Wilcoxon signed rank test that confirms the observed changes in accuracy are significant across all classes but the magnitude of change in medians of the difference in the accuracy of the two models is not big in all classes. Our findings support the integration of the transformer encoder into CNN architecture but we see that the performance of the model can still be increased by introducing regularization terms in the training. We can also explore different configurations using a transformer encoder and experiment with different datasets to generalize our results and further improve model accuracy.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: mathurchinmaya@gmail.com. Supervisor: Chun-Biu Li.

Acknowledgements

I would like to thank my supervisor Chun-Biu Li for his help in this thesis. He helped me shape this thesis with his ideas, critique, and feedback. I am thankful for all the help he provided while writing this thesis.

I have used AI tools to help with spell checks and grammar.

Table of Contents

Abstract **1**

Acknowledgements **2**

List of Figures **5**

1 Introduction **6**

2 Methodology **8**

2.1 Neural Network 8

2.2 Optimization 10

2.2.1 Cross-Entropy Loss function 10

2.2.2 Gradient Descent 10

2.2.3 Stochastic Gradient Descent 11

2.2.4 Adam 12

2.3 Convolutional Neural Network (CNN) 13

2.3.1 Convolution operation 13

2.3.2 Pooling 15

2.3.3 Batch Normalization 17

2.4 ResNet-18 20

2.4.1 Residual connection 20

2.4.2 Architecture 21

2.5 Transformers 23

2.5.1 Attention and Self Attention 24

2.5.2 Layer Normalization 27

2.5.3 Architecture 28

3 Data **33**

4 Results **36**

4.1 Training of ResNet-18 and Hybrid model 36

4.2 Computational Efficiency 39

4.3 Comparing Weights of the blocks in the models 39

4.4 Comparing Class-wise accuracy 40

5 Conclusion **53**

References **54**

List of Figures

2.1 Feedforward Network. Orange dots represent the input neurons. Green dots represent the hidden layer and blue dot represent the output layer. 9
2.2 Example of a convolution with a 3×3 input and a 2×2 kernel with a stride of 1 and no padding. 15
2.3 An example of average pooling with an input of size 6x6, a kernel of size 2×2, and a stride of 2. 16
2.4 Nested function and non-nested functions. For non-nested functions increasing the function does not lead it closer to the true function (f^*). 19
2.5 This figure illustrates how a residual connection works. The residual connection is shown by the line diverging from the input. The input x is processed through two weight layers and an activation function and the output f(x) is then added to the original input x, forming the final output f(x) + x. This helps the network learn identity mapping and helps with the vanishing gradient problem. 19
2.6 This figure illustrates a ResNet-18 architecture. On the left, it shows the overall structure of ResNet-18 which includes the convolution layer, batch normalization, and ReLU activation function. It is followed by 4 residual blocks and an average pooling and fully connected layer leading to the output. On the right, it shows the detailed structure of a residual block. The 1×1 convolution in the residual connection is not used in block 1, since there is no change in the number of channels. 22
2.7 (left) Scaled Dot-Product Attention. (right) Multi-head attention consists of several attention layers running in parallel. Taken from [1]. 26
2.8 Transformer architecture taken from [2]. It consists of 2 parts: encoder and decoder. In encoder, there are 2 layers: Multihead attention and Positionwise FFN. The 'Add & norm' means that it first add the residual connection and then performs a Layer normalization. The 'n' in the figure refers to the number of encoder and decoder blocks in the transformer. 31

2.9 Architecture of the Hybrid model. The overall structure as shown on the left is similar to ResNet-18 (Figure 2.6) but we replace the 4^{th} block with a transformer encoder. The structure of residual blocks is the same as in ResNet-18. On the right, we can see a detailed structure of a transformer encoder. The blue lines coming from the side and connecting with the '+' sign represent the residual connection. Integrating a transformer encoder into the ResNet-18 structure enhances the model's ability to capture long-range dependencies in the input (image or words) which leads to improved feature extraction using its self-attention feature as explained in the above section of Transformers. 32

3.1 Classes in the CIFAR-10 dataset and 10 random images from each class 34

4.1 ResNet-18 training and validation curves 37
4.2 ResNet-18 validation accuracy 37
4.3 Hybrid model training and accuracy curves 38
4.4 Number of parameters in the 2 models (ResNet-18 and Hybrid) . . 39
4.5 weight comparison (original vs modified) block 1 conv. 1 40
4.6 weight comparison (original vs modified) block 1 conv. 2 41
4.7 weight comparison (original vs modified) block 3 conv. 3 41
4.8 weight comparison (original vs modified) block 3 conv. 4 42
4.9 Confusion matrix of model 1 (original ResNet-18) 42
4.10 Confusion matrix of model 2 (Hybrid model) 43
4.11 95% Confidence Interval for accuracy for all classes for both original ResNet-18 and Hybrid model. The points represent the accuracy of the model. Model 1 (blue) represents the Original ResNet-18 model and Model 2 (orange) represents the Hybrid model. Only automobiles, birds, cats, and dogs have non-overlapping intervals showing significance. 46
4.12 Distribution of accuracies for ship class in both the models. They overlap but the medians are different. Model 1 is the ResNet-18 and Model 2 is the Hybrid model. 47
4.13 Test statistics of all classes using Wilcoxon signed rank test. 49
4.14 p-values of all classes using Wilcoxon signed rank test. All classes are significant. 50
4.15 The median of the differences between the ResNet-18 and the Hybrid model calculated from the Bootstrap samples. 51
4.16 Number of ties in each class of the dataset from the Wilcoxon Signed Rank Test 52

Chapter 1

Introduction

Deep learning has revolutionized the field of computer vision, leading to significant advancements in tasks such as image classification and object detection. The ability to categorize and identify objects within images has revolutionized how we interact with and interpret visual data. This has led to advancements in medical imaging diagnostics [3], real-time video analysis, and personalized content delivery. Despite the significant progress made with Convolutional Neural Networks (CNNs), the quest for improvement in image classification continues. CNNs have demonstrated remarkable success by efficiently capturing local spatial hierarchies in images. However, their inherent limitations in modeling long-range dependencies and global context led to the introduction of transformer-based architectures.

Convolutional Neural Networks (CNNs) have been influential, with models like ResNet-18 achieving state-of-the-art performance on numerous benchmarks. Recently, transformers, originally developed for natural language processing (NLP) tasks, have been adapted for image processing, showing promising results. The Vision Transformer (ViT) [4] exemplifies this potential, showing that transformers can outperform traditional CNNs in image classification tasks when provided with sufficient data and computational resources. Carion et al. (2020) also introduced the Detection Transformer (DETR) model, which integrates transformers for end-to-end object detection, highlighting the versatility and effectiveness of transformers in vision tasks [5]. Parmar et al. (2018) [6] illustrates, how models based on the architecture of self-attention can significantly improve image modeling of complex images in the ImageNet dataset.

In this thesis, we propose a Hybrid model that integrates the strengths of both CNNs and transformers. We modify the ResNet-18 architecture by replacing its 4^{th} block with a transformer encoder containing a multi-head self-attention layer and a Position-wise Feedforward network. The motivation behind this modification is to leverage the transformer's ability to capture long-range dependencies and contextual information, potentially enhancing the feature extraction capabilities of the model. The architecture of ResNet-18 is taken from [7]. The transformer encoder architecture is similar to the one described in [1]. We used the ResNet-18 model and compared it with the Hybrid model to determine what changes occur when a transformer encoder is introduced in ResNet-18. Additionally, we analyzed how these processes impact the model's

performance for image classification and how the weights of the first 3 blocks change between the models after training. Using Bootstrapping, we compare the classwise accuracy in the two models by visualizing the 95% Confidence Interval (CI) of the accuracies. Furthermore, we conducted a Wilcoxon Signed Ranked test to see the significance of the changes in the accuracy.

The overall accuracy of both models remains almost the same, with the Hybrid model doing slightly better by 2% than ResNet-18. Regarding classwise accuracy, the Hybrid model provides a better classification for most of the classes but for certain classes like "cat", it does not work as well as ResNet-18. The weights of the first 3 blocks in both models are pretty much the same in terms of distribution but differ in terms of magnitude with the Hybrid model having higher weights which can be seen in the quantile-quantile (QQ) plot as it is slightly bent upwards (more steep) compared to the diagonal line. From the Wilcoxon Signed Rank test, we find that all the classes have significant change in the accuracy between the models but only certain classes have practical significance. We propose some regularization in the training for better results and further testing to dig deeper into the effects of introducing a transformer encoder in ResNet-18.

The structuring of this thesis is as follows: Chapter 2 discusses the methodology used in the thesis including the methods used and the models' architecture. Chapter 3 discusses the dataset used. Chapter 4 examines the thesis results and Chapter 5 concludes with reflections and final thoughts. The code used to make the models and do the analysis can be found here: https://github.com/ChinmayaMathur/Thesis-for-masters-Transformers-.

Chapter 2

Methodology

2.1 Neural Network

A neural network works similarly to how neurons work in a human brain. It is a connection of neurons that pass on information to do a specific task such as classification and pattern recognition. The most common straightforward type of neural network is a Feedforward Neural Network (FFNN) also known as Multilayer perceptions (MLP). Feed-Forward Neural Network is a single-layer perceptron. A sequence of inputs enters the layer and is multiplied by the models' weights. The weighted input values sum together to form a total. If the sum of the values exceeds a predetermined threshold (typically zero), the output value is usually 1. If the sum is less than the threshold, the output value is usually -1. The single-layer perceptron is a popular Feedforward Neural Network model frequently used for classification. They are named this since information is only propagated forward through the network [8].

All neural networks have three main layers: Input, hidden, and output. Figure 2.1 shows an example where the input layer has 4 neurons, the hidden layer has 3 neurons, and 1 neuron in the output layer. In this case, the model's depth is 3 as it contains three layers, and its width, defined by the dimensionality of these layers, is 4. In this example, each neuron is connected to every neuron in the subsequent layer, meaning that each layer is fully connected.

In a feedforward network, for some vector input $x = (x_1, x_2, ..., x_n)$ where n is the total number of inputs, the layer calculates outputs as $\phi(\omega^T x + b)$ where ω is the weight matrix and b is the bias and both of these are learnable parameters. ϕ here represents a non-linear activation function acting element-wise on its input.

The most commonly used activation function in neural networks is the Rectified Linear Unit (ReLU). The formula for ReLU is:

$$f(x) = max(0, x) \tag{2.1}$$

where x is the input to the neuron. The main advantage of using ReLU is its sparse activation, indicating that not all neurons are activated when using it; only non-zero outputs are used. This helps reduce the computational cost making the network more efficient to compute. It also helps reduce the risk of

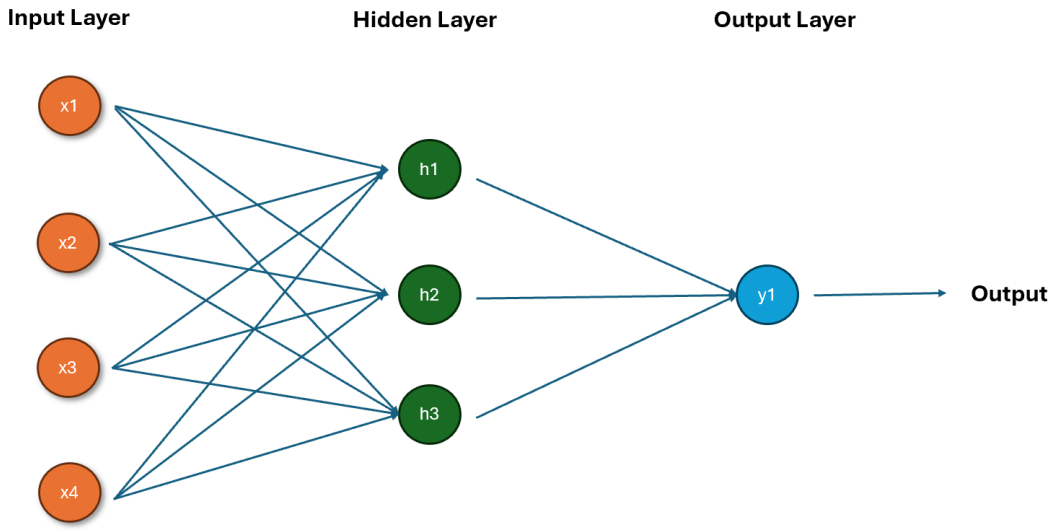

Figure 2.1: Feedforward Network. Orange dots represent the input neurons. Green dots represent the hidden layer and blue dot represent the output layer.

overfitting the data since fewer neurons are active at a time and it also helps in better feature selection. ReLU has better gradient propagation than other activation functions like sigmoid or tanh, leading to fewer vanishing gradient problems. It is computationally efficient and also scale-invariant.

For classifying multiple classes, the Softmax function works the best since it converts the raw output scores of a model into probabilities, making each output interpretable as the likelihood of belonging to a particular class. It also emphasizes the highest score, allowing the model to identify the most probable class. The formula for Softmax is:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.2}$$

Here, z represents the values from the neurons of the output layer - the exponential acts as the non-linear function. The Softmax activation function normalizes the input values into a probability distribution, ensuring that the sum of all output values is 1. It is suitable for classification problems where the output needs to represent probabilities over multiple classes. By exponentiating the inputs, the Softmax function in machine learning amplifies the differences between the input values, making the largest value more pronounced in the output probabilities.

There are other activation functions like sigmoid and tanh but in this thesis since we are dealing with modern architectures like Resnet 18 and Transformers, ReLU is preferred as it is usually used in hidden layers due to its ability to avoid vanishing gradient problem and its also computationally efficient. We also use Softmax in the output layer instead of any other activation function since it works well with multi-class classification tasks as it converts raw logits into

normalized probabilities, ensuring that the model outputs are interpretable as probabilities.

2.2 Optimization

2.2.1 Cross-Entropy Loss function

We want to optimize the parameters θ (i.e., the set of all weights and biases) to minimize the loss function, also known as the cost function while training our model. The loss function quantifies the difference between the model's predicted and target values, where a higher loss indicates poorer predictions. In our thesis, we will be using the cross-entropy loss function. It measures the difference between the predicted probability distribution and the true labels. The benefit of using this loss function in our thesis is that it penalizes incorrect predictions more strongly when the model is confident but wrong, encouraging accurate probability estimates which makes it perfect for classification tasks. Another advantage of using it is that it can be used in combination with the Softmax function as it ensures that the output probabilities sum to 1.

Consider a scenario where we have q number of classes; we represent the correct labels with a vector y of length q, where each element is either 0 or 1. The model's predictions are represented by the vector $\hat{y}$, also of length q, which contains the predicted probabilities for each class. The cross-entropy loss for a single observation is defined as:

$$l(y, \hat{y}) = -\sum_{i=1}^{q} y_i \log \hat{y}_i \tag{2.3}$$

here, y_i is the actual label (1 if the class is correct, 0 otherwise), and $\hat{y}_i$ is the predicted probability for the i^{th} class. The loss function is bounded below by 0 and increases as the predicted probability of the true class decreases. We compute the average loss across all observations to evaluate the model's performance over the entire dataset. This aggregate loss provides a measure of the model's overall performance and is used to guide the optimization process.

2.2.2 Gradient Descent

Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent in machine learning finds the values of a loss function's parameters (coefficients) i.e. θ that minimize a cost function. A gradient measures the change in all the weights with regard to the change in error (i.e. difference between the true and predicted values). You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient of a function is a partial derivative with respect to its arguments.

For a loss function $l(y, \hat{y}; \theta)$ which depends on some parameters θ, the gradient descent updates the parameter at each step as follows:

$$\theta_{j+1} \leftarrow \theta_j - \eta . \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i; \theta_j)}{\partial \theta_j} \tag{2.4}$$

where η is the learning rate which is a positive number, N is the total number of observations in our dataset, j represents the current step, and i labels the observation.

In practice, the learning rate decreases after each step. If the learning rate is kept constant and is too small, it can cause the parameters to update very slowly, which makes it harder to converge efficiently leading to a prolonged training process. On the other hand, if it is too large then it can lead it to overshoot the optimal minima and just oscillate around it. Decreasing the rate after each step helps reduce this risk and helps the loss function to reach a minimum quickly because this way it makes large parameter updates in the initial stages for faster exploration and then slowly smaller updates in later stages for precise convergence. This helps the model make parameter updates more efficiently.

To calculate the gradient of the loss function we do this using backpropagation [2]. While training the network, we use Forward propagation (or forward pass) to calculate and store intermediate variables from the input layer including all layers in the middle till we reach the output layer. This is used to make a prediction and then compared with the actual target value to compute the error (loss) using the loss function. Backpropagation evaluates the gradient of the loss function with respect to each parameter (weight and bias) in the network using the chain rule. The gradient indicates how much the loss would change if the parameter were slightly adjusted. Using the chain rule of calculus, we compute the gradient of the loss with respect to each weight and bias, layer by layer, moving backward from the output layer to the input layer. Once the gradients are computed, the weights and biases are updated in the opposite direction of the gradient (hence "back" propagation). Once all the gradients are computed we then use an optimization algorithm to update them. The forward pass and the backward pass are repeated many times which reduces the error gradually, improving the models' performance by fine-tuning the network parameters.

2.2.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variant of gradient descent in which instead of computing the gradient of the function over the entire dataset, we compute the gradient for a mini-batch of observations at each iteration. A mini-batch is a random subset of observations of the whole dataset. We use SGD as it is more computationally efficient than running it over millions of observations.

Let there be b samples in a mini-batch, then at each step, the stochastic gradient descent updates the parameters using the following rule:

$$\theta_{j+1} \leftarrow \theta_j - \eta . \frac{1}{b} \sum_{i=1}^{b} \frac{\partial l(y^i, \hat{y}^i; \theta_j)}{\partial \theta_j} \tag{2.5}$$

Selecting a smaller mini-batch size helps with training larger models as the small number of observations in a mini-batch uses less memory while training. It can also lead to faster convergence as the parameters are updated more often allowing for quicker adjustments based on the latest gradient information. One iteration over the whole training set is known as an epoch.

2.2.4 Adam

The Adam (Adaptive Moment Estimation) optimizer is a popular optimization algorithm used for training deep learning models. It was introduced by Diederik P. Kingma and Jimmy Ba in their 2015 paper titled "Adam: A Method for Stochastic Optimization" [9]. Adam optimizer maintains two moving averages for each parameter: the gradients' first moment (mean) which tracks the direction of the gradient and the second moment (uncentered variance) which measures the magnitude of the gradient. Using these, Adam can adaptively adjust the learning rate for each parameter based on the gradient. This leads to stable and effective learning. Adam also includes bias correction in both moments estimates which ensures that they are not underestimated due to their bias to 0 during initilization and this leads to appropriately scaled updates which leads to faster convergence of the training process.

The hyperparameter used in Adam is the Learning rate set to a default value of 0.001, Decay rates (or momentum) with default values are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. These control the decay rates of the moving averages and Epsilon (ϵ) a small constant (e.g., 10^{-8}) added to the denominator to improve numerical stability and prevent division by zero.

Adam uses Momentum, a technique used to accelerate gradient descent by considering past gradients to smooth out the updates. It helps navigate the parameter space more effectively. The first-moment estimate '$\hat{m}_t$' acts as a form of momentum. It accumulates the gradients with an exponential decay rate β_1, similar to how momentum accumulates gradients in the traditional momentum-based gradient descent. Adam enjoys the benefits of momentum like smoother updates by reducing the impact of noisy gradients due to the exponential decay of past gradients. Exponential decay also leads to faster convergence in directions with consistent gradients, and it also improves stability during training by avoiding local minima due to accumulated past gradients history stored in m_t. A pseudocode of the algorithm of how Adam works is shown below in Algorithm 1.

Adam uses adaptive learning rates for each parameter as shown in step 12 of the algorithm 1, which makes it adaptive. The term $\frac{\alpha}{\sqrt{\hat{v}_t}+\epsilon}$ represents the adaptive learning rate as based on the value of $\hat{v}_t$, the learning rate changes. The term α is the base rate which is fixed, so if $\hat{v}_t$ is large meaning high variance in the gradients, then the learning rate will be small and vice versa. The ϵ term makes sure that the denominator is not zero when $\hat{v}_t$ is zero or really small. Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments (Seen in Steps 8 and 9). This is because at the time of initilaztion, moving averages m_t and v_t are

Algorithm 1 The Adam Algorithm

Require: Step size α (Suggested default: 0.001)
Require: Exponential decay rates β_1 and β_2 in $[0, 1)$ (defaults: 0.9 and 0.999)
Require: Small constant ϵ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters θ_0
Initialize 1st moment vector $m_0 = 0$
Initialize 2nd moment vector $v_0 = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
    $t \leftarrow t + 1$
    Sample a minibatch of m examples from the training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
    Compute gradient: $g_t \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(f(x^{(i)}; \theta_{t-1}), y^{(i)})$
    Update biased first moment estimate: $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    Update biased second moment estimate: $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t$
    Compute bias-corrected first moment estimate: $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
    Compute bias-corrected second moment estimate: $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
    Compute update: $\Delta\theta_t \leftarrow -\alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$
    Apply update: $\theta_t \leftarrow \theta_{t-1} + \Delta\theta_t$
end while

initialized to zero and are biased in the early stages. Bias correction ensures that these averages are unbiased estimators of the true first-order and second-order moments. Adam generally performs well with default hyperparameters(learning rate, decay rate (β_1 and β_2) and epsilon), reducing the need for extensive hyperparameter tuning.[9] The default values for these hyperparameters are mentioned above.

2.3 Convolutional Neural Network (CNN)

2.3.1 Convolution operation

Generally, convolution operation is applied on two functions of a real-valued argument. When processing images in a network, convolution operation plays an important part. The image data is in the form of tensors or multi-dimensional arrays of real values. The x and y axes in these arrays represent the spatial dimensions or pixel values, while the third axis represents color channels. The number of color channels depends on the image type: grayscale images have one channel, while colored images usually have three (red, green, and blue).

As mentioned in the book Dive into Deep Learning [2], if we are classifying images then, connecting each pixel of an image of size for example 1 megapixel to the nodes of a 1000-node fully connected layer would require $10^6 \cdot 10^3 = 10^9$ weights to be trained. This makes doing this infeasible as it would require a lot of computational power. This means that flattening an image and using it as a

vector to a feedforward neural network is not the correct way of inputting an image.

Convolutional operations use filters also known as kernels. Each filter is a small matrix that convolves around the input image, performing element-wise multiplications and summing the results to produce a single value in the output feature map. This process allows the network to detect local patterns and features such as edges, textures, and shapes in the initial layers, and more complex patterns like objects and faces in the deeper layers. They are usually smaller in height and width than that of the image. The usual kernel choices are odd-sized like 3×3 and 5×5 [8]. This is because odd-sized kernels have a well-defined center which allows the kernel to symmetrically convolve around the current pixel in the input image. It also allows for symmetric padding on both sides of the input image.

The smaller size of the kernel (dimensions of the kernel) than the input image leads to sparse interactions because each output value depends only on a localized region of the input. Each filter in the convolutional layer extracts specific features from the input, such as edges, textures, etc. For example, a convolutional layer with 32 filters of size 3×3 applied to an input with three color channels produces 32 feature maps, each representing the activations for one filter. This setup only requires $32 \times (3 \times 3 \times 3 + 1) = 896$ parameters, a drastic reduction from the billions required by a fully connected approach as it limits the number of connections between inputs and outputs, decreasing the computational costs and training time. Convolutional operations also utilize parameter sharing which means that they use the same filter (kernel) across the entire input image because of this the weights of the kernel are shared for all spatial positions which makes the model detect the same features (e.g., edges or textures) regardless of their location in the image.

The convolution output can be written as:

$$S(i,j) = (I * K)_{(i,j)} = \sum_m \sum_n I_{(i+m;j+n)} K_{(m,n)} \tag{2.6}$$

Here, the kernel K is a multidimensional array that contains trainable parameters, and I is an input with two axes. i, j represents the elements of the convolution operation. Figure 2.2 shows an example of a convolution operation. In equation 2.6, we did not consider the case of multiple channels.

In convolution operation changing an object's location in a picture does not change the prediction. This is due to the property of convolution operation called equivariance to translation. It is a property of the convolution operation where a shift in the input results in a corresponding shift in the output, without altering the structure of the features detected. To explain this in mathematical terms, let us consider: $I(x)$ be the input signal (e.g. an image) and $k(x)$ be the convolutional kernel (filter). Then, The convolution operation is defined as:

$$(I * k)(x) = \int I(y) k(x - y)\, dy \tag{2.7}$$

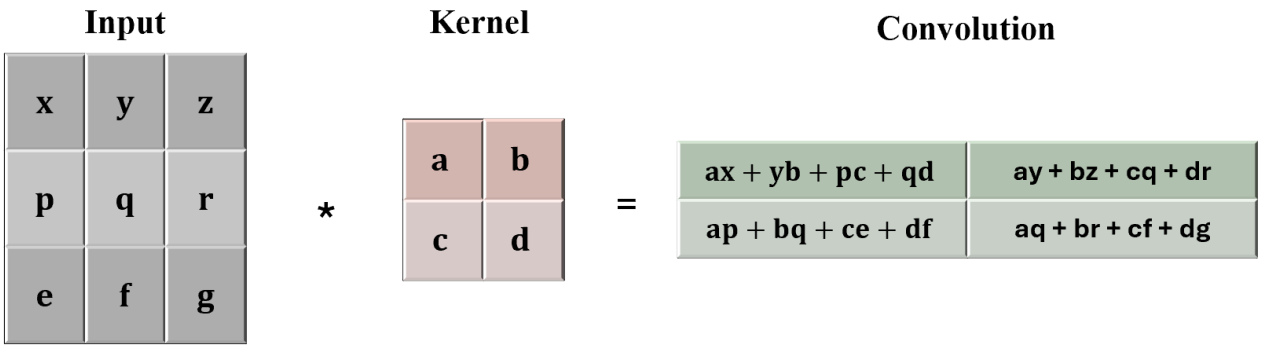

Figure 2.2: Example of a convolution with a 3×3 input and a 2×2 kernel with a stride of 1 and no padding.

or in discrete terms:

$$(I * k)(x) = \sum_{y} I(y)k(x - y) \tag{2.8}$$

Now let the input $I(x)$ be shifted by a certain amount z, the shifted input becomes: I'(x) = I(x - z)

When the convolution is applied to this shifted input:

$$(I' * k)(x) = \sum_{y} I(y - z)k(x - y) \tag{2.9}$$

Now due the equivariance to translation property of convolution operation, the output also shifts by an amount z, Hence:

$$(I' * k)(x) = (I * k)(x - z) \tag{2.10}$$

The size of the convolution can be changed using zero padding and stride. Adding a padding of "x" in the convolution adds "x" zero-valued rows and columns on each side of the input. The stride helps change the kernel's step size as it goes over the input. A bigger stride lets the kernel skip over some elements of the input and leads to a smaller convolution that does not extract the features of the image as well compared to that with a lower stride.

2.3.2 Pooling

As mentioned in the book by Goodfellow [8], a typical layer of a convolutional network consists of three stages. The first stage, the layer performs multiple convolutions simultaneously to give a set of linear activations and in the second stage, each linear activation runs through a non-linear activation function like ReLU. The third stage is where the pooling function is used. A pooling function replaces the output of the layer with a summary statistic of the nearby outputs. It helps to make the representation approximately invariant to small translations of the input which means that the output after pooling remains the same even if

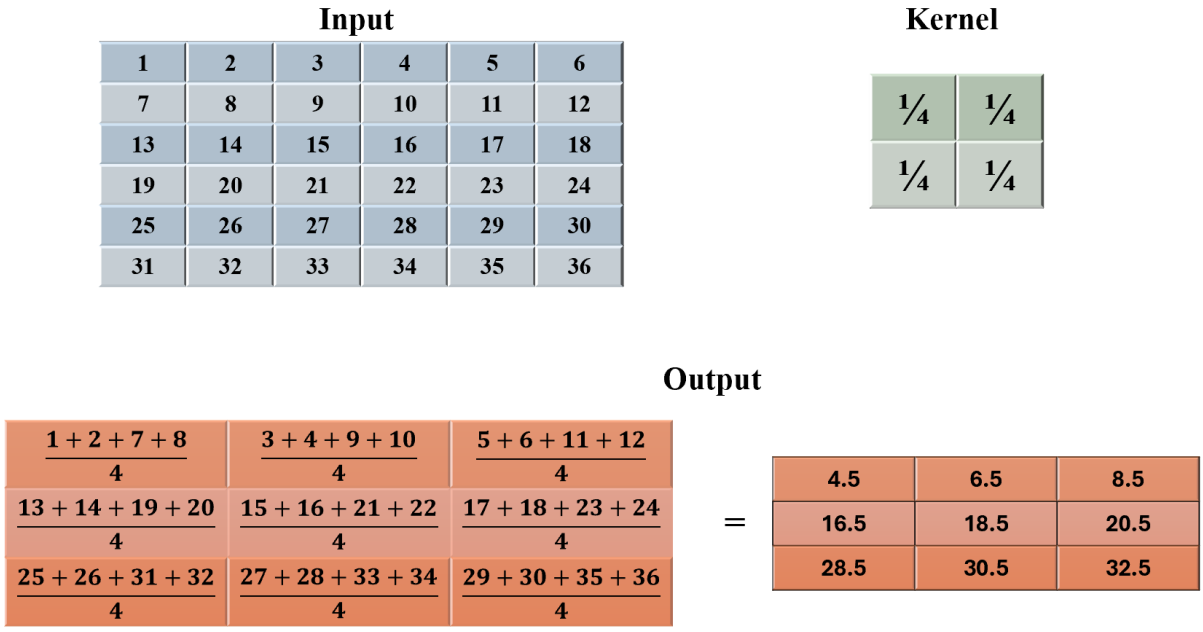

Figure 2.3: An example of average pooling with an input of size 6x6, a kernel of size 2×2, and a stride of 2.

the input is shifted even by a small amount. Invariance to local translation can be a useful property if we care more about whether some feature is present in the input than exactly where it is located[8]. This property helps us in the case of image classification tasks.

There are many types of pooling operations, we use average pooling in this thesis. Consider an input x of size $C \times H \times H$, where C is the number of channels, and $H \times H$ is the Height and Width respectively. Let K be a kernel of size $k \times k$ and s be the stride. Performing average pooling with these arguments yields an output of size $C \times H_{out} \times H_{out}$, where

$$H_{out} = \frac{H - k}{s} + 1 \tag{2.11}$$

here, H_{out} is the height and width of the output feature map. From [10], we find that the element (c, h, w) of the output of such an operation is defined as:

$$AP(x)_{c,h,w} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} x_{c,s \cdot h+m,s \cdot w+n} \tag{2.12}$$

This formula computes the average of the elements in the $k \times k$ neighborhood of the input tensor x, reducing the output size by the stride factor. In average pooling, the kernel used does not have any trainable parameters, it simply computes the mean of the input values within the receptive field. Figure 2.3 shows an example of an average pooling operation where an input of size 6×6 is passed through a kernel of size 2×2 with 1 channel.

Another type of pooling operation is max pooling, where instead of taking the average value within the receptive field, it takes the maximum value. However, this type of pooling is not covered in this thesis.

2.3.3 Batch Normalization

It was introduced in the paper "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" [11]. Batch normalization is used to improve the training of deep neural networks by addressing the problem of internal covariate shift. In a neural network, each layer has inputs (activations) with their own distribution. These distributions can change during training as the parameters of the previous layers are updated. While the randomness of parameter initialization contributes to the initial variability, the continuous changes in activations are primarily caused by the updates to the parameters during gradient-based optimization. This phenomenon is often referred to as internal covariate shift, where the input distribution to a layer changes due to parameter updates in earlier layers. This can cause instability in the network as shifting input forces layers to continuously adapt, making optimization less efficient. This leads to layers having to relearn optimal weights for changing inputs which leads to more epochs to reach good performance and makes the training process slow.

Batch normalization works by normalizing the input to each layer so that the mean and variance become 0 and 1 respectively. It performs this normalization over each mini-batch during training to make sure that the input distribution for each layer remains constant. This operation can be divided into two main steps: normalization and rescaling.

For each mini-batch B containing b examples, batch normalization first computes the mean $\hat{\mu}_B$ and variance $\hat{\sigma}_B^2$ of the mini-batch:

$$\hat{\mu}_B = \frac{1}{b}\sum_{i=1}^{b} x_i \tag{2.13}$$

$$\hat{\sigma}_B^2 = \frac{1}{b}\sum_{i=1}^{b}(x_i - \hat{\mu}_B)^2 \tag{2.14}$$

Using these statistics, the input x is normalized to produce $\hat{x}$:

$$\hat{x} = \frac{x - \hat{\mu}_B}{\sqrt{\hat{\sigma}_B^2 + \epsilon}} \tag{2.15}$$

Here, ϵ is a small constant added to the variance to prevent division by zero and ensure numerical stability.

After normalization, the normalized values are rescaled and shifted using two learnable parameters, γ (scale) and β (shift):

$$BN(x) = \gamma\hat{x} + \beta \tag{2.16}$$

Here, the parameters γ and β restore the degrees of freedom lost by normalization, ensuring that the operation can represent the identity transform.

Batch normalization is primarily used to reduce internal covariate shift, however, as highlighted in [12], the effectiveness of batch normalization extends beyond just reducing internal covariate shift. It also contributes to smoothing the loss function's surface with respect to the model's parameters (weights and biases). This loss function's surface is known as the optimization landscape. It contains many local minima, saddle points, and flat regions where optimization becomes unstable or can get stuck. Smoothing this landscape reduces sharp peaks or valleys, making it easier for optimization algorithms like gradient descent to converge to a good solution. Batch normalization does this by normalizing the inputs at each layer which ensures that inputs to subsequent layers have a consistent mean and variance. This reduces large fluctuations in the loss caused by varying input scales or distributions. This also leads to more consistent gradients during backpropagation which reduces the likelihood of vanishing or exploding gradients, making it easier for the optimizer to find a low-loss region.

Batch normalization normalizes the inputs of each layer in a mini-batch during training using the mean and variance computed from the mini-batch. Since each mini-batch is a random subset of the dataset, the computed mean and variance are only an estimate of the overall mean and variance of the whole dataset. This variability in mean and variance due to randomness of the mini-batch introduces noise into the normalized outputs, which in turn affects the gradients during training. This noise acts as a form of regularization and can reduce overfitting as it slightly changes the activation for each mini-batch preventing it from replying to a certain pattern in the training data which improves the generalization of the network to unseen data. As a result, batch normalization can sometimes reduce the need for other regularization techniques such as dropout. Batch normalization helps maintain healthy gradient magnitudes throughout the network preventing the gradients from becoming too large or too small (vanishing or exploding gradients) and ensures stable and efficient backpropagation.

During training, batch normalization uses the statistics (mean and variance) of the current mini-batch. However, during testing, the network needs to be consistent and use the estimated population statistics (mean and variance) computed over the entire training set. This ensures that the network behaves consistently and does not depend on the specific mini-batch used during evaluation. In CNNs, batch normalization is applied before the activation function to ensure more stable and effective activation, improving the learning process. It is also applied after the convolution operation to normalize the outputs of the convolution layer. Given a mini-batch of size b and a convolutional layer output of width w, height h, and number of channels c, batch normalization is applied independently to each channel. The mean and variance are computed over all $b \times h \times w$ elements in each channel, and the normalization is performed per channel. The rescaling and shifting parameters γ and β are also learned per channel.

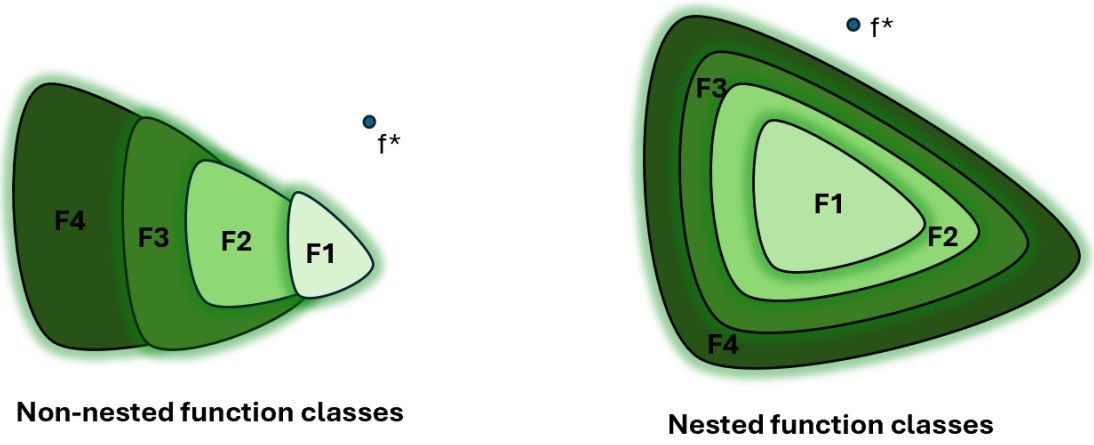

Figure 2.4: Nested function and non-nested functions. For non-nested functions increasing the function does not lead it closer to the true function (f^*).

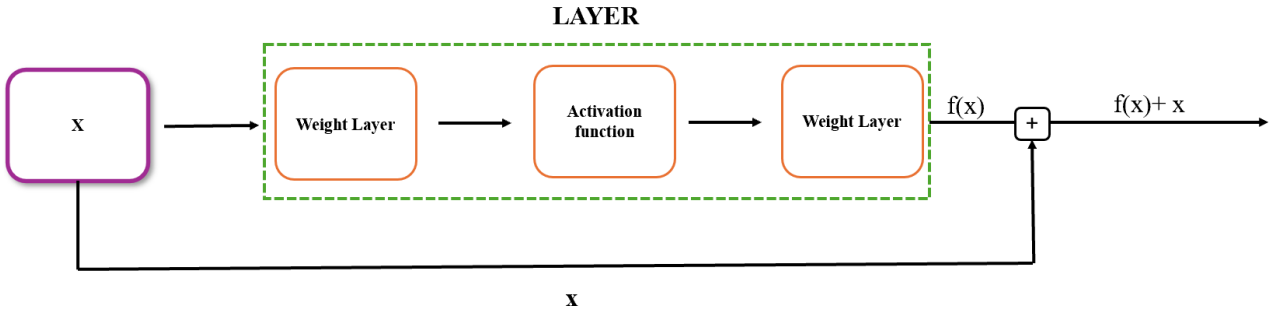

Figure 2.5: This figure illustrates how a residual connection works. The residual connection is shown by the line diverging from the input. The input x is processed through two weight layers and an activation function and the output f(x) is then added to the original input x, forming the final output f(x) + x. This helps the network learn identity mapping and helps with the vanishing gradient problem.

2.4 ResNet-18

2.4.1 Residual connection

Residual connections also known as skip connections, were first introduced in the paper "Deep residual learning for image recognition" by He et al. [7]. It talked about the degradation problem in deep neural networks, where the performance of deeper networks was worse compared to the performance of swallow networks. This happened because deeper networks struggled to learn identity mapping, which is essential for preserving the learned features from previous layers.

The book "Dive into Deep Learning" [2], talks about how for non-nested functions, increasing the functions does not lead us to be closer to the actual value of the function. From Figure 2.4, we can see that for some target function f^* that we want to approximate, let $F1$ be the class of functions that a network can model using different parameter configurations. Now, if f^* lies within $F1$, then we can approximate f^* but if it lies outside then for our network to be able to approximate this function, we might think of adding layers to it and increase the complexity of the model as shown by F2, F3 and F4 in the figure. This does not always lead to getting the correct answer. Adding another layer without knowing if that layer would contain the previous layer/function, we might end up going farther from our target function f^*.

To solve this issue, we use residual connections. Suppose that we are trying to approximate a function $g(x)$ for some input x. The layer learns a function $f(x)$ such that $g(x) = f(x) + x$. This way, if $f(x)$ has a small change or is close to 0, each layer can learn identity mapping where $g(x) \approx x$, and it makes sure that the new layer always contains the previous layer. This makes sure that adding more layers with residual connections increases the network's ability to approximate the target function f^*. Figure 2.5 shows an example of a residual connection.

One can think of residual connections as a mechanism to prevent the loss of information. They act as a "shortcut" that helps preserve the original information about the input throughout the layers. This mechanism ensures a consistent flow of information throughout the network. During backpropagation, residual connections play a critical role in mitigating the vanishing gradient problem. In Figure 2.5, the output is computed as: $g(x) = f(x) + x$, where $f(x)$ is the output of the layers, and x is the input to the layer. When calculating gradients during backpropagation, the derivative of this function becomes:

$$\frac{\partial g(x)}{\partial x} = \frac{\partial f(x)}{\partial x} + 1$$

Even if $\frac{\partial f(x)}{\partial x}$ becomes very small or zero, the gradient of x with respect to itself is always 1. This ensures that the gradient never vanishes completely and maintains a consistent flow of gradient information which enables the network to learn efficiently and effectively mitigate the issue of vanishing gradients.

2.4.2 Architecture

For our thesis, we use the ResNet-18 model. It is mostly used for image classification tasks due to its robust performance and high accuracy. The architecture has become a benchmark in the field, significantly influencing subsequent neural network designs. ResNet-18 is a CNN network that contains block structures known as residual blocks. The architecture begins with an initial convolutional layer using a 3×3 kernel with a stride of 1 and padding 1, followed by batch normalization and the ReLU activation function as shown in Figure 2.6.

The network then consists of four residual blocks. Each residual block in ResNet-18 contains four convolutional layers each using a 3×3 kernel, each followed by batch normalization and ReLU activation. Importantly, these blocks incorporate residual connections that bypass the convolutional layers, allowing the network to learn identity mappings. This helps mitigate the degradation problem in deep networks, ensuring that deeper models do not perform worse than their shallower counterparts.

Some residual connections, especially when the number of channels changes between blocks, utilize a convolution operation with a 1×1 kernel followed by batch normalization to match the dimensions. This can be seen in the right image of Figure 2.6 where there is a line connecting input 'y' to the output after 2^{nd} Batch normalization (g(y)) right before applying ReLU function. This adjustment ensures that the residual connection aligns with the output of the convolutional layers. Specifically, this applies to the first residual connection in the second, third, and fourth blocks, where the number of channels increases. In the first block there is no 1×1 kernel as there is no increase in channels.

The convolutions in the network utilize padding of 1 and a stride of 1 other than the first convolutions in blocks 2, 3, and 4, which use a stride of 2. We use a padding of 1 with a 3×3 convolution to ensure that the output feature map retains the same spatial dimensions (height and width) as the input. Increasing the stride to 2 effectively halves the spatial dimensions (height and width) of the feature maps which makes the convolutional filter cover a larger portion of the original image, effectively increasing the receptive field. This helps the network capture more global features, which are essential for understanding complex patterns and contexts within the image. The 1×1 convolutions in the model use a stride of 1 and no padding as padding is unnecessary because the filter perfectly aligns with each spatial location in the input feature map.

Following the residual blocks, an average pooling layer is used. In the average pooling, we use an adaptive average pooling function to make sure that our output is 1×1. So to make sure our output is 1×1, we choose the kernel size in average pooling to be 4×4. The final fully connected (dense) layer outputs a 10-dimensional vector, corresponding to the 10 classes in the CIFAR-10 dataset. Each element of this output vector represents the unnormalized log probability of each class. After this, we need to apply a softmax function to get a proper output that represents a certain class from our dataset.

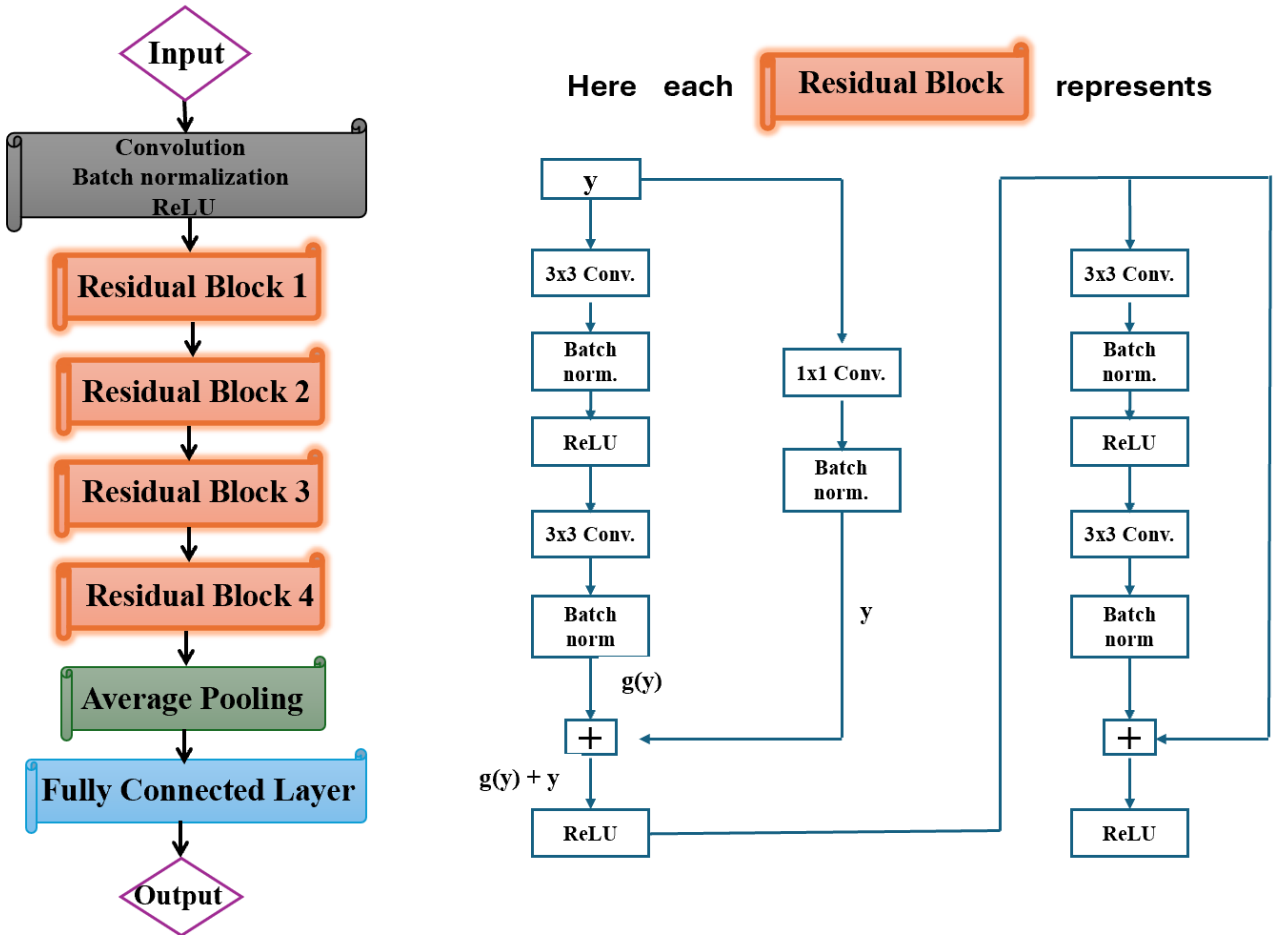

Figure 2.6: This figure illustrates a ResNet-18 architecture. On the left, it shows the overall structure of ResNet-18 which includes the convolution layer, batch normalization, and ReLU activation function. It is followed by 4 residual blocks and an average pooling and fully connected layer leading to the output. On the right, it shows the detailed structure of a residual block. The 1×1 convolution in the residual connection is not used in block 1, since there is no change in the number of channels.

	Dimensionality
Input	**32 x 32 x 3**
After Block 1	**32 x 32 x 64**
After Block 2	**16 x 16 x 128**
After Block 3	**8 x 8 x 256**
After Block 4	**4 x 4 x 512**
After Average Pooling	**1 x 1 x 512**
After Fully Connected Layer	**1 x 1 x 10**

Table 2.1: Dimenionality of input as it passes through the network. It is represented in the form Height x Width x Channels.

The dimensionality of the data changes as it passes through the network as we can see from Table 2.1. The dimension of the input image is $32 \times 32 \times 3$ (Height x Width x Channel), after passing through each block it changes to $32 \times 32 \times 64$, $16 \times 16 \times 128$, $8 \times 8 \times 256$, and $4 \times 4 \times 512$ respectively. After it passes through the average pooling layer, the dimensionality changes to $1 \times 1 \times 512$, and the final output we get after the fully connected layer is of dimension $1 \times 1 \times 10$. There is an initial increase in the number of channels after passing through the first block since it helps increase the image's dimensions which helps with finding the decision boundary. We then decrease the spatial dimensions by half and increase the channels by 2 after each block. The spatial dimensions are reduced gradually so that the network still retains enough spatial information while decreasing the resolution which decreases the computational cost. The channels are increased after each block so that the networks learns more features from the deeper layers as shallow layers focus on simple features like edges which require less channels but deeper layers focus on complex features which require higher number of channels. The reason for increasing it by 2 is to balance out the decrease in the spatial information by half and preserve the total information capacity of the feature map. The architecture of ResNet-18 which gets its name due to 18 convolution layers in the structure, is used for various computer vision tasks, including image classification, object detection, and segmentation.

2.5 Transformers

In the paper by Vaswani et al. (2017), "Attention Is All You Need"[1], Transformers were first introduced and now they are used for mostly all natural language processing (NLP) tasks and has also been used in many other fields as well like computer vision. The main reasoning behind using Transformers is its ability to process input sequences in parallel, unlike recurrent neural networks (RNNs) or long-short-term memory (LSTM) networks, which process data sequentially [13]. This makes Transformers highly efficient and scalable specially when using large datasets. The Transformers were originally built for sequence transduction tasks such as language modeling and machine translation. It consists of encoder and decoder stacks, each with multiple identical layers that utilize self-attention mechanisms. The model uses attention mechanisms such as Scaled Dot-Product Attention and Multi-Head Attention, Layer normalization,

Position-wise feedforward networks, and positional encoding. [1]. The Transformers have better computational efficiency than RNN's or CNN's and can also handle long sequences. In my thesis, we are going to use Transformers for image classification task.

2.5.1 Attention and Self Attention

The attention mechanism allows the Transformer model to dynamically focus on relevant parts of the input data. It assigns different importance (weights) to various elements in the input sequence or image. The intuition behind using attention is that rather than compressing the whole input as we do in RNNs (Recurrent Neural Network), it might be better to revisit the input sequence at every step. It also helps in seeing different representations of the input by selectively focusing on certain aspects of the input instead of always seeing only one representation. [2] Transformers use mainly self-attention which is a specific type of attention, where each element in the input sequence either words in a sentence or image patches, attends to all other elements in the same sequence. This allows the model to capture both local and global dependencies. For example, in a sentence, self-attention helps determine how much each word contributes to the meaning of another word by comparing them together. The key difference between normal attention and self-attention is that self-attention works within a single sequence to model relationships among its elements, while normal attention operates across two sequences to align their elements.

The attention is computed using the scaled dot-product attention, defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.17}$$

Here:

- Q (Query): Represents what the model is focusing on (e.g., a specific word or image patch).
- K (Key): Represents the "labels" or identity of all input elements.
- V (Value): Contains the actual information of the input elements.
- d_k: Dimensionality of the keys used to scale the dot product.
- Attention(Q, K, V): The attention score.

In this equation, the dot product (QK^T) measures the similarity between queries and keys. A higher dot product value means a stronger similarity between the query and a specific key. The values V are weighted by the attention scores, producing a context-aware representation of the query. We also normalize the dimensionality of the keys because for large values of d_k the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. We scale the dot products by $1/\sqrt{d_k}$ to counteract this effect. [1] We use the Softmax function which converts the similarity scores into probabilities (attention weights) that sum to 1. These probabilities

represent how much attention the model pays to each key for a given query. These probabilities are used to weight the value vectors (V) when constructing the final output (context vector). By summing the weighted values, the model produces a context-aware representation of the query that reflects the relative importance of all keys. For example, let's consider a case where our input is a sentence containing 3 words. In this, Q is the query vector for the first word, V is the value vector for all the words in the sentence and K is the key vector for all the words in the input sentence. After applying the Softmax function, we get that P(1) = 0.5, P(2) = 0.2, and P(3) = 0.3, this means that 50% attention is paid to the first word, 20% attention is paid to the second word and 30% attention is paid to the third word for this query.

Let us consider another example, where the input is an image, we first convert the image into smaller patches and then flatten them into 1-D and project them into a higher-dimensional embedding space using a linear transformation as shown in Figure 2.7 (on the right-hand side in the light blue box with Linear written in it). For each patch, the 3 vectors: Q, K, V are calculated using the learned weight matrix for each of them respectively. Attention scores are then calculated using the formula in Eq. 2.17, in which the dot product of Q and K compares how much one patch is similar to another. For example, if the input image is a dog, it will give a high score to a patch containing the dog's ear with the patch containing the dog's face. These scores are then scaled by dividing them by the square root of d_k to stabilize them and then passed through a softmax function to turn them into probabilities between 0 and 1. These probabilities are then multiplied by the weighted sum of the values. This produces a new representation for each patch, enriched with information from other patches.

In Transformers, multi-head Self-attention is used which allows the model to focus on different aspects of relationships within the input sequence (words or image patches) by running multiple self-attention mechanisms in parallel. Each attention head operates in a different subspace learned from the input. This enables the model to focus on multiple types of dependency simultaneously. For example, in a sentence, one head might focus on the relationships between subject and verb alignment, while another might focus on word meanings. Similarly in images, one head might detect edges, while another identifies textures. By combining the outputs of these multiple attention heads, the model creates a more comprehensive representation of the input. This aggregation of different perspectives allows the Transformer to learn more complex features, improving its performance. Multi-head attention also reduces the risk of overfitting by splitting the input into multiple small subspaces. This reduces the reliance on a single attention head overfitting to specific patterns. Each head captures a different perspective of the sequence or image, which makes the model generalize better as various parts of the input contribute differently to the overall prediction. However, using too many heads can also increase the risk of overfitting due to over-parameterizing the model as each head has its own set of learnable parameters. Therefore, the number of heads should be chosen carefully to balance the embedding size (d_{model}) and model capacity based on the task and dataset size. It also helps Transformer capture a wider range of dependencies for

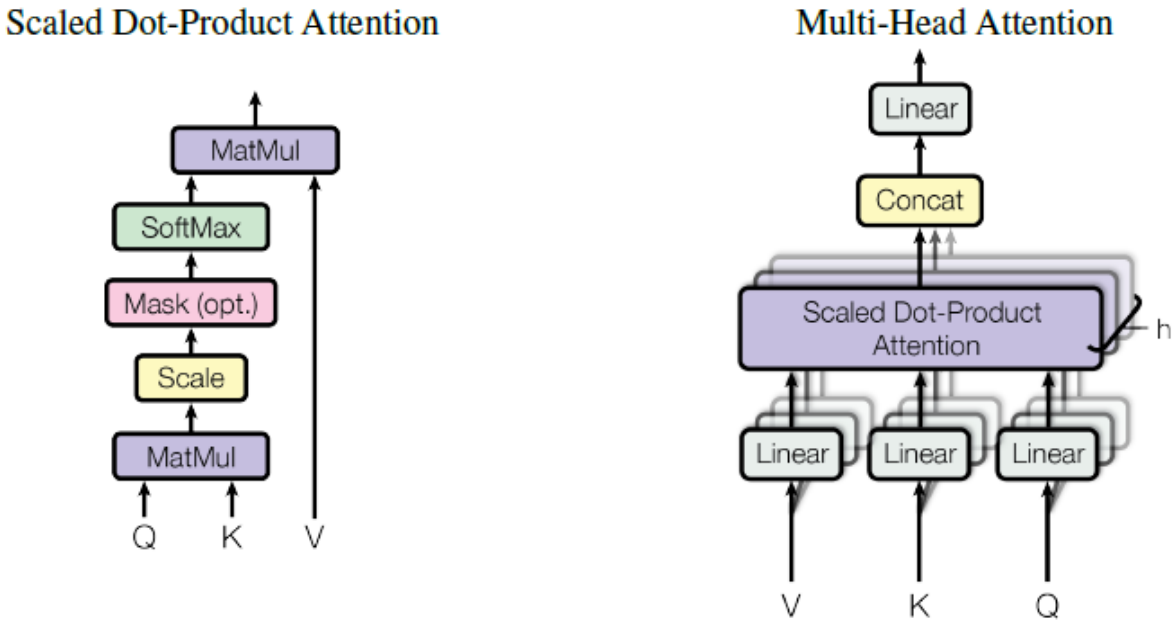

Figure 2.7: (left) Scaled Dot-Product Attention. (right) Multi-head attention consists of several attention layers running in parallel. Taken from [1].

example in language tasks, one head might focus on short-range dependencies like consecutive words, while another tracks long-range dependencies across sentences and in images, one head might attend to nearby patches, while another looks at distant patches.

The final output from each of these multiple attention heads is then concatenated and linearly transformed using dot product multiplication to obtain the final output as shown in Figure 2.7 (right side image in the yellow box with 'Concat' written in it and the grey box with 'Linear' written in it). The multi-head attention mechanism can be represented by the following equations:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2.18}$$

where

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.19}$$

Here,

- The Q, K, V are query, key and, value same as in the formula of attention score.
- $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ are weight matrices.
- Here d_{model}, d_k and d_v means the dimensionality of the input and output embeddings in the model, dimensionality of the keys and query, and dimensionality of the value respectively.

- Here h represents the number of heads in the multi-head attention model
- The $Attention(QW_i^Q, KW_i^K, VW_i^V)$ here is the attention score calculation formula as mentioned in Eq. 2.17. Here dot product multiplication is used to find attention scores.

The 'Concat' operation in this formula means stacking the outputs of all attention heads along the feature dimension. For example, if each attention head outputs a matrix of size (n, d_v), where n is the sequence length and d_v is the dimension per head, concatenation combines these matrices into a single matrix of size $(n, h \cdot d_v)$, where h is the number of heads. Finally, the concatenated matrix is multiplied by a learnable weight matrix W^O to project the output back to the original feature dimension d_{model} and this done using dot product multiplication. The advantage of this design is that multi-head attention allows the model to capture diverse relationships in the input data by attending to different parts of the input simultaneously. Also since we operate on a lower dimension subspace due to splitting into multiple heads for each head the dimension of the model is decreased by the number of heads, which makes computational complexity easier to manage which helps in the case of larger datasets or complex models.

In our thesis, we employ h (number of heads) $= 4$, and we use $d_k = d_v = d_{model}/h = 64$. The total computational cost in this case is approximately the same as that of the single-head attention model as the decreased dimensionality of the keys and values balances out the increase in the number of heads. This is beneficial because using multiple heads allows the model to compute attention for smaller subspaces of the input embedding in parallel while keeping the computational cost almost the same.

2.5.2 Layer Normalization

After passing the input through the attention mechanism in the Transformer, it is necessary to normalize the inputs for improved training stability and convergence. To achieve this, layer normalization is applied at two key points in the Transformer: after the attention mechanism and after the position-wise feedforward layer, as shown in Figure 2.8. In the multi-head attention mechanism, the outputs from different attention heads are concatenated which results in a dynamic range of outputs which can vary significantly. Layer normalization ensures that these combined outputs are well-scaled before being passed to subsequent layers.

Layer Normalization was first introduced in the paper by Ba et al.[14] and it differs from batch normalization by normalizing across the feature dimension instead of the batch dimension. This design choice provides several benefits that make layer normalization particularly suited for Transformer architectures. Unlike batch normalization, which relies on statistics computed per batch, layer normalization computes the mean and variance independently for each training sample. This makes it batch-size independent, allowing consistent performance even for small or variable batch sizes, which is usually seen in natural language processing (NLP) and sequence modeling tasks.

For an n-dimensional vector x, layer norms are given by:

$$x \to LN(X) = \frac{x - \hat{\mu}}{\hat{\sigma}} \tag{2.20}$$

where scaling and offset are applied coefficient-wise and given by,

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.21}$$

and

$$\hat{\sigma^2} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 + \epsilon \tag{2.22}$$

As before we add a small offset $\epsilon > 0$ to prevent division by zero. One of the major benefits of using layer normalization is that it prevents divergence. Divergence refers to the instability that occurs when the model's parameters or gradients grow uncontrollably, leading to a failure in convergence. Transformers have a large number of parameters and nonlinear activations, which can result in gradients that either explode (grow too large) or vanish (become too small). Layer normalization addresses this by normalizing the outputs of each layer to have a mean of zero and a standard deviation of one across the feature dimension. This normalization ensures that the inputs to subsequent layers are consistently scaled, stabilizing gradient updates during backpropagation and improving the likelihood of successful convergence. It is also independent of whether we are doing training or testing. This is due to the fact that layer normalization computes statistics per sample during both training and inference. This per-sample normalization ensures consistent behavior, leading to more reliable and stable model performance during deployment. In contrast, batch normalization behaves differently during these two phases: during training, it uses statistics (mean and variance) from the mini-batches, while during testing, it uses the estimated population statistics (mean and variance) computed over the entire training set. This discrepancy can sometimes lead to performance issues if the running averages fail to capture the true data distribution. This is another reason why Layer normalization is used here instead of Batch normalization.

2.5.3 Architecture

In the original paper "Attention is all you need" [1], Transformers have an encoder-decoder structure. In the encoder, the input sequence $x = (x_1, \ldots, x_n)$, where n is the number of tokens in the input sequence, is mapped to a sequence of continuous representations $z = (z_1, \ldots, z_n)$ through layers of multi-head self-attention and position-wise feed-forward networks as shown in Figure 2.8 on the left-hand side in the encoder block. Here, x represents the input tokens or embeddings (e.g., words or patches), and z represents the intermediate continuous representations capturing contextual information about the input sequence. The decoder takes z as input, along with its own input sequence $y = (y_1, \ldots, y_{m-1})$, to generate the output sequence $y = (y_1, \ldots, y_m)$, where m is the number of tokens in the output sequence. The decoder generates y one token

at a time in an auto-regressive manner, using masked multi-head self-attention (In Figure 2.8 in decoder block) to prevent attending to future tokens in the sequence. Additionally, the decoder employs attention over the encoder's output z to condition the generation of y on the input sequence x.

At each step, the decoder consumes previously generated symbols as additional input to produce the next. This architecture is implemented using stacked self-attention layers and point-wise feedforward networks for both the encoder and decoder. The encoder comprises 6 identical stacked layers, each with two sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feedforward network. Residual connections surround each sub-layer, followed by layer normalization which can be seen in Figure 2.8 (left side in the encoder block). The output of each sub-layer can be expressed as: $LayerNorm(x + Sublayer(x))$, where $Sublayer(x)$ is the function implemented by the sub-layer itself. The decoder is also composed of a stack of 6 identical layers as mentioned in [1]. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. The decoder output is then processed through a fully connected layer (FC) (In Figure 2.8 at the end of the decoder block) to produce the final probabilities for each token in y.

Transformers architecture can be adapted so that they can be used for image data and they are known as Vision Transformers (ViTs). ViTs, introduced by Dosovitskiy et al. [4] in 2020, demonstrated that the Transformers which are known for their effectiveness in Natural Language Processing (NLP), can also be used to handle image data and they outperform CNN's in this task. ViTs only use the encoder component, converting images into embeddings via patch embedding and positional encoding. An additional classification (CLS) token is prepended to the sequence of patch embeddings to aggregate global image information for final classification tasks. Due to this property of ViT's, it makes them flexible with respect to image size unlike CNNs, where input resizing affects feature extraction. They also perform well on larger datasets like ImageNet due to their ability to model global relationships using self-attention. This helps them capture complex dependencies across the entire image especially when the dataset is large.

In this thesis for our hybrid model, we integrate a Transformer block into a ResNet-18 architecture, replacing the 4th residual block as seen in Figure 2.9 (left side image). The transformer block includes only 1 encoder stack since we want to see how the encoder changes the performance of the original Resnet-18 model. We do not include the decoder part in this block since it's not a Natural Language translation task and we do not need decoding. In this model, we are not using positional encoding in the transformer block as the positional encoding is used in the transformer to make sure that the information about the position of the input image is preserved since the transformer processes inputs as sequences that lose the positional context, but since in our hybrid model there are convolutional layers in the previous 3 blocks of the model due to ResNet-18 structure, it already preserve the spatial structure of the image. As a result, the transformer encoder processes spatially-aware feature representations directly,

making positional encoding redundant. The encoder consists of a single stack with a multi-head self-attention layer and a position-wise feed-forward network. The dimensions of the output by the multi-head attention is 256, the same as the input taken from the output of the 3rd residual block of our model as shown in figure 2.9 (left side diagram, f(x) is the output of the multi-head attention). In this model, after the multi-head attention layer, there is a residual connection that is added to the output and then it is passed through the Layer Normalization as seen in the left side of Figure 2.9. After this, the output 'y' of the Layer Normalization is passed through a position-wise fully connected feedforward network (FFN). This is applied to the feature vector of each position in the input sequence independently and identically. This ensures that the transformations at each position are consistent and do not interfere with the positional relationships learned by the attention mechanism. This consists of two linear transformations with a ReLU activation in between. The output from FFN (In Figure 2.9 right side in the Position-wise Feedforward Network block) can be as follows:

$$FFN(x) = max(0, x^T W_1 + b_1)W_2 + b_2 \tag{2.23}$$

Here,

- Here x represents the input to the positionwise feedforward network. It is the output of the multi-head attention layer.
- W_1 and W_2 are the weights of the 2 linear transformation layer and b_1, b_2 represent the bias in respective layer.

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is 256 and 512 respectively to facilitate the residual connections, and the inner layer has a dimensionality of 2048. The output then passes through a Layer Normalization layer again before which residual connection is again added to it to make sure there is no loss of information. This is where the Transformer encoder ends and the output of this goes into the average pooling layer and then to a Fully connected layer to get the final output similar to how it is done in ResNet-18 architecture. By comparing this architecture against standard ResNet-18, we aim to explore how the inclusion of a Transformer encoder influences image classification accuracy and performance.

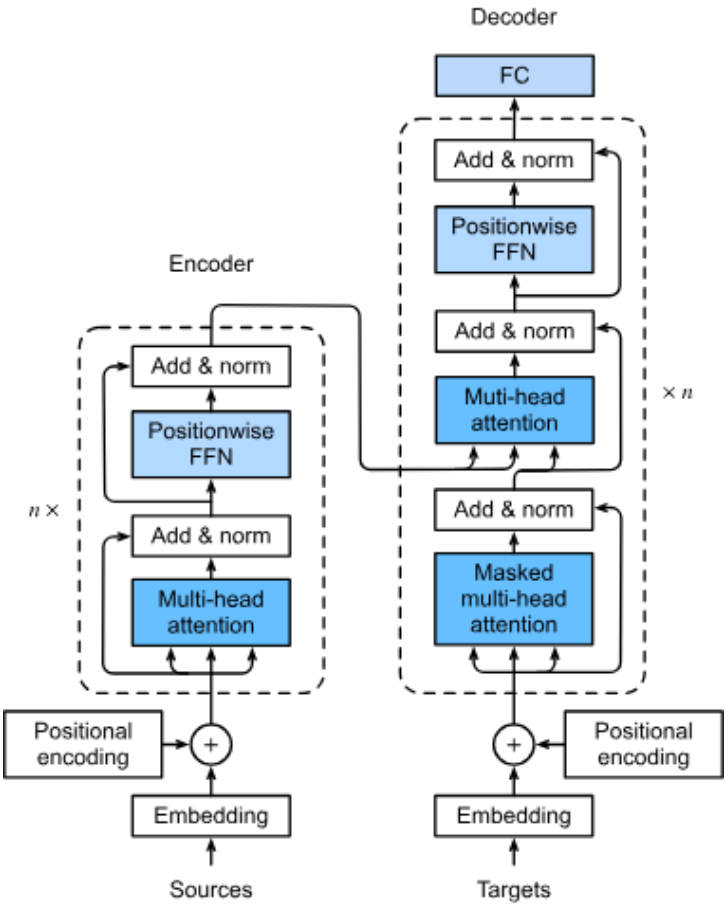

Figure 2.8: Transformer architecture taken from [2]. It consists of 2 parts: encoder and decoder. In encoder, there are 2 layers: Multihead attention and Positionwise FFN. The 'Add & norm' means that it first add the residual connection and then performs a Layer normalization. The 'n' in the figure refers to the number of encoder and decoder blocks in the transformer.

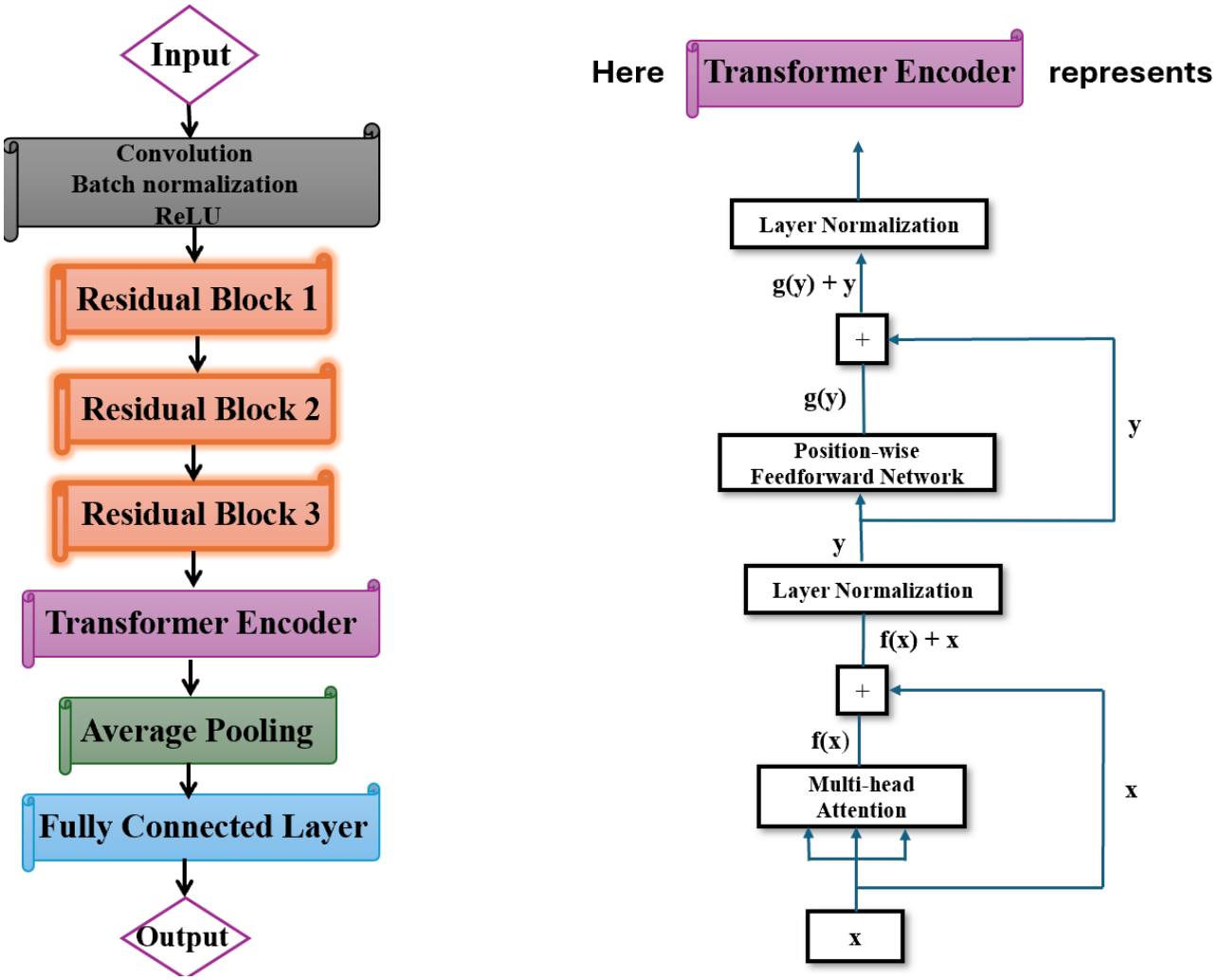

Figure 2.9: Architecture of the Hybrid model. The overall structure as shown on the left is similar to ResNet-18 (Figure 2.6) but we replace the 4^{th} block with a transformer encoder. The structure of residual blocks is the same as in ResNet-18. On the right, we can see a detailed structure of a transformer encoder. The blue lines coming from the side and connecting with the '+' sign represent the residual connection. Integrating a transformer encoder into the ResNet-18 structure enhances the model's ability to capture long-range dependencies in the input (image or words) which leads to improved feature extraction using its self-attention feature as explained in the above section of Transformers.

Chapter 3

Data

We use CIFAR-10 as our dataset [15]. It is commonly used for classification tasks, it consists of 60000 images and all the images are from one of the 10 classes present in our dataset. Each class has 6000 images each of resolution 32×32. Since our dataset has 3 color channels, they are represented as a $3 \times 32 \times 32$ tensor. The dataset is split into a training set and validation or testing set with 50000 and 10000 observations each. Each set contains an equal number of elements from each class. In Figure 3.1, we see a few examples of images from each class.

Before training, each image is normalized by subtracting the mean of each channel and dividing by the standard deviation, based on the training set. This ensures that all the images in the training set have mean 0 and standard deviation of 1 across all channels which benefits the overall training process as all the inputs are on similar scale and one channel can't dominate the other. We also randomly flip some images horizontally in the training set with a probability of 0.5. This augmentation is done during the data loading process, such that, for each training epoch an image has an equal chance of appearing in its original form or as a horizontally flipped version. This does not increase the size of the dataset so it is also efficient. The purpose of this is to prevent overfitting and improve the generalization capability of the model by exposing the model to flipped versions of the input images, it learns more robust and invariant feature representations that are less sensitive to the spatial orientation of objects. This is particularly useful in datasets like CIFAR-10, where the horizontal orientation of objects (e.g., animals, vehicles, etc.) does not affect their class label. These normalized and data augmented images are input to our ResNet-18 and the Hybrid model, which is trained using the Adam optimizer and the cross-entropy loss function.

For weight initialization in the model, we use He initialization [16]. This method helps to keep the variance of the activations roughly the same across every layer. It is especially useful for layers with ReLU (Rectified Linear Unit) activation functions. The He initialization sets the weights to be sampled from a normal distribution with a mean of 0 and a variance of $\frac{2}{l}$, where l is the number of input units in the weight tensor.

$$W \sim \mathcal{N}(0, \frac{2}{l}) \tag{3.1}$$

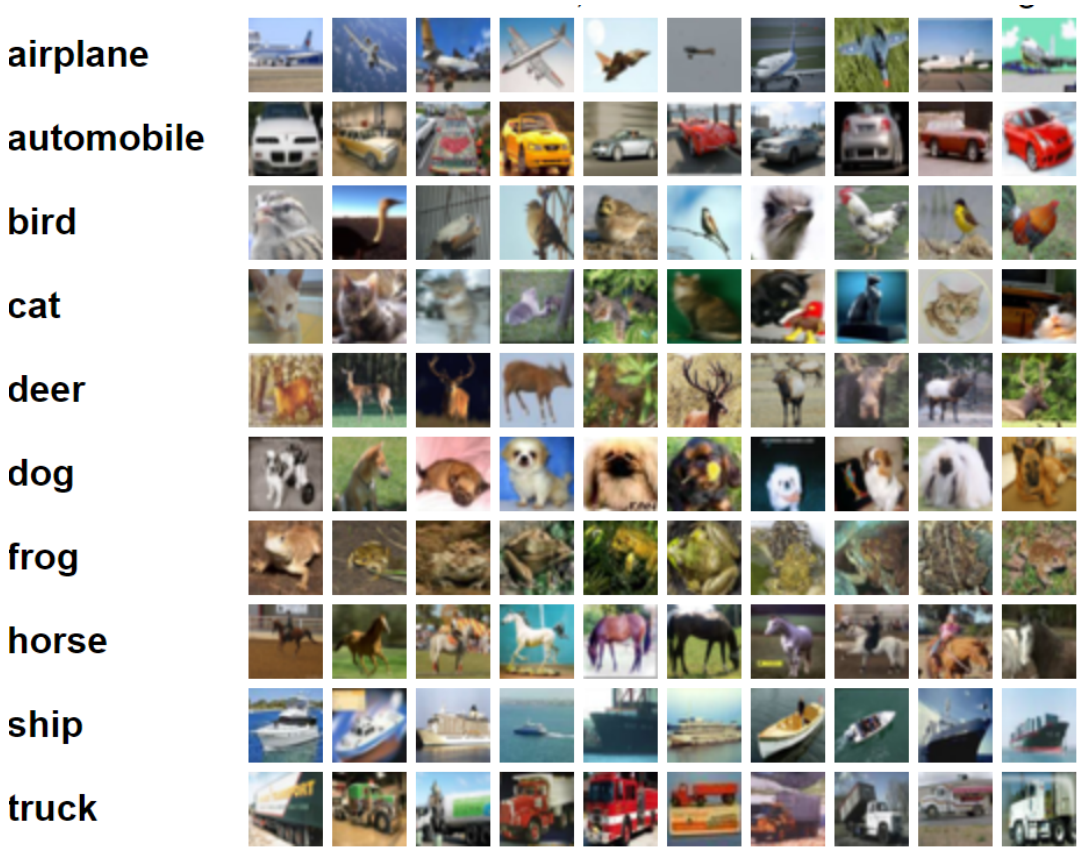

Figure 3.1: Classes in the CIFAR-10 dataset and 10 random images from each class

After training the ResNet-18, we use the weights of the first 3 blocks and load them in our Hybrid model, and train them again on the CIFAR-10 training set. We do this since we want to compare the two models (ResNet-18 and Hybrid) and make sure that any results or changes in performance that we see are due to architectural changes in the Hybrid model and not due to any initialization difference. This also helps in converging the Hybrid model faster as the weights for the first 3 blocks are already close to optimal.

Chapter 4

Results

4.1 Training of ResNet-18 and Hybrid model

We trained our models on the CIFAR-10 dataset. We made the training and validation loss curves along with a graph for validation accuracy for both the models as seen from Figures 4.1, 4.2, and 4.3. Validation accuracy is calculated by testing the trained model on a validation dataset (a subset of the data that is not used for training) and determining the proportion of correct predictions. This metric is used to evaluate how well the model generalizes to unseen data. For ResNet-18, the validation accuracy shows a steep increase from around 67.5% to over 80% within the first 10 epochs, indicating efficient initial learning. After 15 epochs, we see that the validation accuracy increases more slowly and stabilizes around 87.5% between epochs 20 and 40, indicating that the model has captured most of the significant patterns in the dataset. The training loss decreases rapidly, approaching near-zero values, showing the model's effective learning on the training data. The training loss plateaus after around 15 epochs, indicating the model has started to overfit the training data. The validation loss decreases initially but stabilizes with minor fluctuations indicating that it is performing well. We use the weights that gave the lowest validation loss for our model to prevent overfitting which is done around epoch 7 as seen in Figure 4.1. The accuracy we get around this epoch is 86%.

The hybrid model's training curve shows a rapid decrease and stabilizes at a lower value, indicating effective learning. Since we already loaded the trained weights of the ResNet-18 model in the Hybrid model the loss is already low as seen in Figure 4.3. This is because the weights are already optimized in the training of ResNet-18 but we do see an initial decrease in the training loss due to the structural change in the architecture of the model. There is an initial decrease in the validation loss but it starts rising again after epoch 5 which shows that the model has started overfitting. The accuracy curves for the hybrid model show that the training accuracy increases rapidly, reaching close to 98% within 50 epochs. However, the validation accuracy fluctuates around 88%, indicating stability. In this model, we choose the weights for the optimal model where the validation loss is minimum which is around epoch 5. The accuracy we get at epoch 5 is around 88%

Both models achieve similar validation accuracies but the Hybrid model has

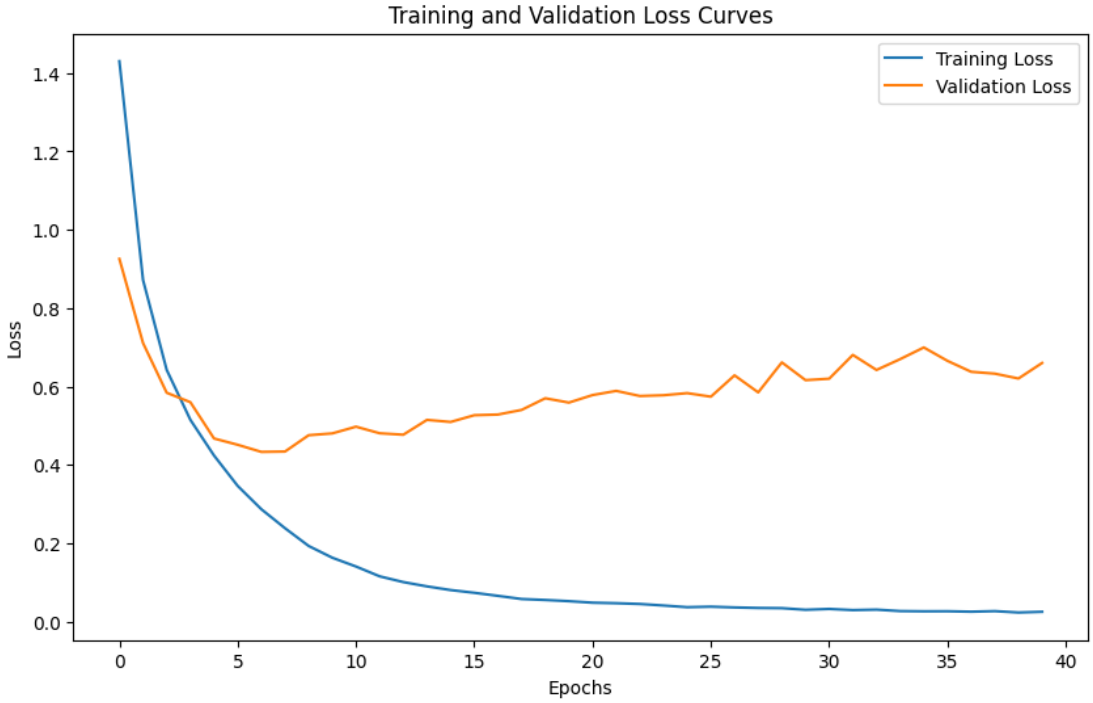

Figure 4.1: ResNet-18 training and validation curves

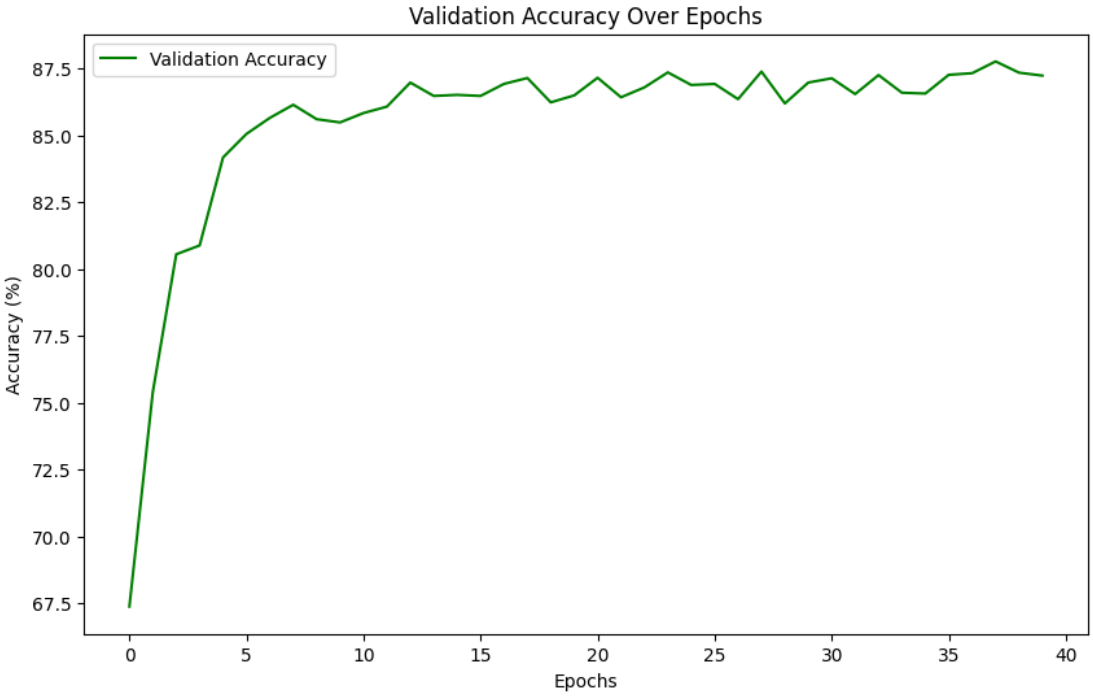

Figure 4.2: ResNet-18 validation accuracy

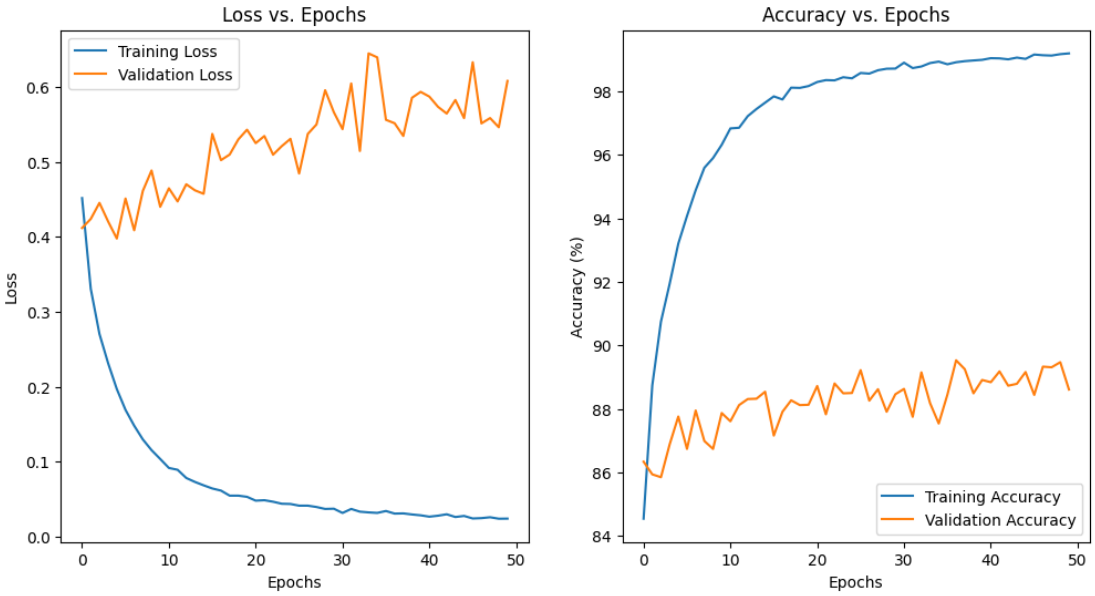

Figure 4.3: Hybrid model training and accuracy curves

slightly better accuracy indicating that it performs slightly better than the ResNet-18 model. The ResNet-18 model, with its well-established architecture and residual connections, effectively mitigates the vanishing gradient problem as already explained in the previous section of ResnNet-18. On the other hand, the hybrid model, which incorporates a combination of convolutional and transformer-based components, shows high training accuracy but struggles with generalization as seen in Figure 4.3 on the right side graph by the gap between training accuracy and validation accuracy. The high variability in validation performance of the hybrid model as seen in Figure 4.3, suggests that the models ability to generalize on unseen data is not good as it does not show stable or increasing pattern in the accuracy curve. This can also be linked to overfitting as the validation loss increases and fluctuations in validation accuracy imply that the model's predictions on unseen data are unstable. Overfitting occurs because the model has focused too much on specific features of the training data, leading to a situation where it performs well on data it has seen but poorly or inconsistently on new data. This highlights the importance of regularization techniques, such as dropout and data augmentation to enhance model robustness.

Comparisons between the models based on which one is faster to train can not be done based on these graphs as although we are training the ResNet-18 from scratch but for the Hybrid model, we are using pre-trained weights. This will not give us a full picture of which model is easier to train. In our thesis, we are not doing any regularization techniques as we just want to see how the model architecture of ResNet-18 and transformers differ from each other in terms of training on CIFAR-10.

Block	Number of trainable parameters in Resnet 18	Number of trainable parameters in the Hybrid model
1	147968	147968
2	525568	525568
3	2099712	2099712
4	8393728	1971712
Total	**11166976**	**4744960**

Figure 4.4: Number of parameters in the 2 models (ResNet-18 and Hybrid)

4.2 Computational Efficiency

To determine which model would be faster and cheaper to train meaning which model is more computationally efficient, we can see how many trainable parameters each model has. This provides a good indication of the computational resources and time required by the models for training.

The number of parameters is greater in ResNet-18 compared to the hybrid model. The number of parameters in the first 3 blocks of both the models is the same since the architecture of both of them is similar as seen in Figure 4.4. The difference comes in the 4^{th} block where in ResNet-18 there are 8,393,728 trainable parameters and in the hybrid model there are 1,971,712 parameters. This means that it will be more computationally efficient to train the hybrid model as it has fewer parameters in total.

4.3 Comparing Weights of the blocks in the models

We check the distribution of the weights for the first 3 blocks of both models and compare them to see if there are any differences in them. We used the weights of the trained ResNet-18 model, loaded them on the Hybrid model, and trained it again. We want to see if introducing a transformer encoder changes the weights of the first 3 blocks.

We plot the quantile-quantile (QQ) plot as well as a Scatter plot to see how the distribution of the weights is for each layer in the first 3 blocks. We can see this from Figures 4.5 and 4.6, which show the QQ-plot and Scatter plot for the 1^{st} and 2^{nd} convolution layer in block 1 respectively. From this, we can see that in the QQ-plot for 1^{st} convolution layer, the blue line is slightly tilted upwards (more steep) than the diagonal red line. In a perfect QQ-plot the blue line should perfectly align with the red line which would mean that the weights of both the models in these 2 convolution layer in block 1 are similar. But in our case, this upward tilted blue line means that the distribution of weights in the hybrid model is scaled differently compared to the original ResNet-18 but the overall distribution of weights between the two models remains similar and the weights in the hybrid model are larger in magnitude compared to those in the original ResNet-18. These changes in the weights are likely due to the hybrid

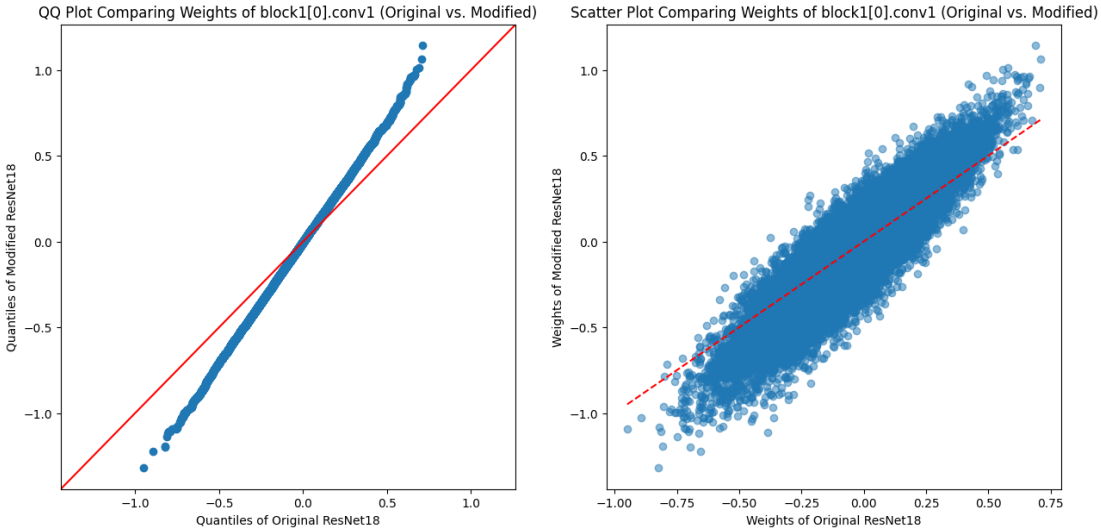

Figure 4.5: weight comparison (original vs modified) block 1 conv. 1

model's adaptation to the introduction of the transformer encoder block, which excels at capturing long-range dependencies and global relationships in the data. To support this processing, the earlier layers of the hybrid model (inherited from ResNet-18) adjust their weights to extract features that better align with the transformer's strengths. In Figure 4.6, the one for the 2^{nd} convolution layer of block 1, we see similar behavior, but in the scatter plot, we see more scatter of the points compared to the previous convolution layer, suggesting that it underwent greater adaptation during training, possibly because this layer deals with higher-level feature extraction compared to the previous convolution layer. We also see slight deviations at the lower and upper quantiles (the tails) that suggests that extreme weight values in the hybrid model differ from those in the original ResNet-18.

We see similar results for the convolution layers 3 and 4 of the 3^{rd} blocks as shown in Figures 4.7 and 4.8. In the scatter plot of both the convolution layers we see that the scatter of points is more compared to what we saw in the earlier convolution layers of the first block. This indicates that the closer the convolution layers in the earlier blocks are to the Transformer encoder block, the more their weights are influenced by the architectural change. In the QQ-plot, we see similar results to the ones we observed before which indicates in these layers the weights distribution is also almost similar but the scale is different and hybrid model has higher magnitude of weights compared to Original ResNet-18 model especially at the tails. In conclusion, we see that as layers get closer to the transformer block, the weights show increasing adaptation, reflecting the hybrid model's adjustment to the architectural change.

4.4 Comparing Class-wise accuracy

To further analyze the change that introducing a transformer encoder in ResNet-18 architecture gets, we compare the classwise accuracy for both models.

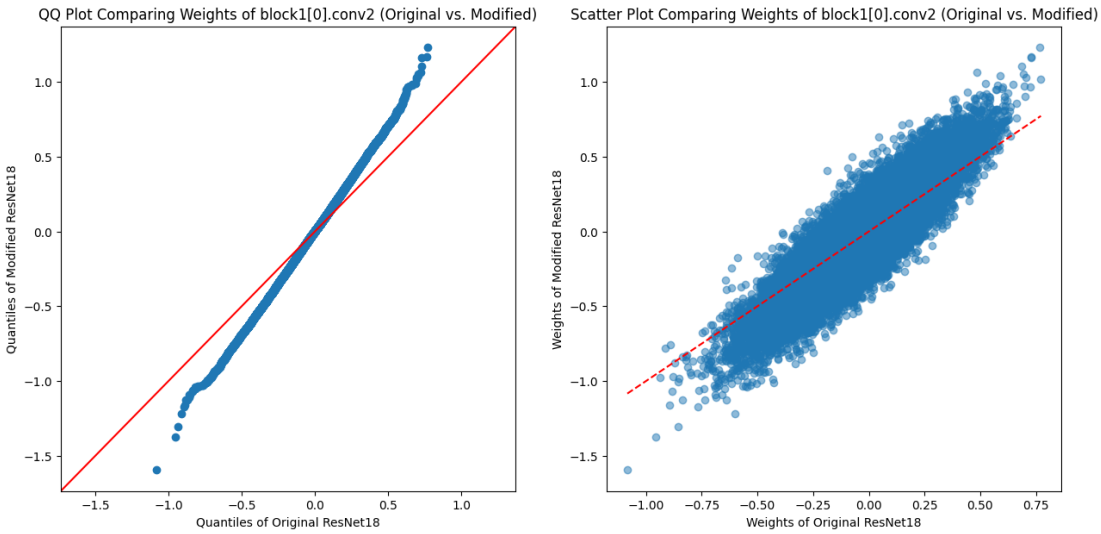

Figure 4.6: weight comparison (original vs modified) block 1 conv. 2

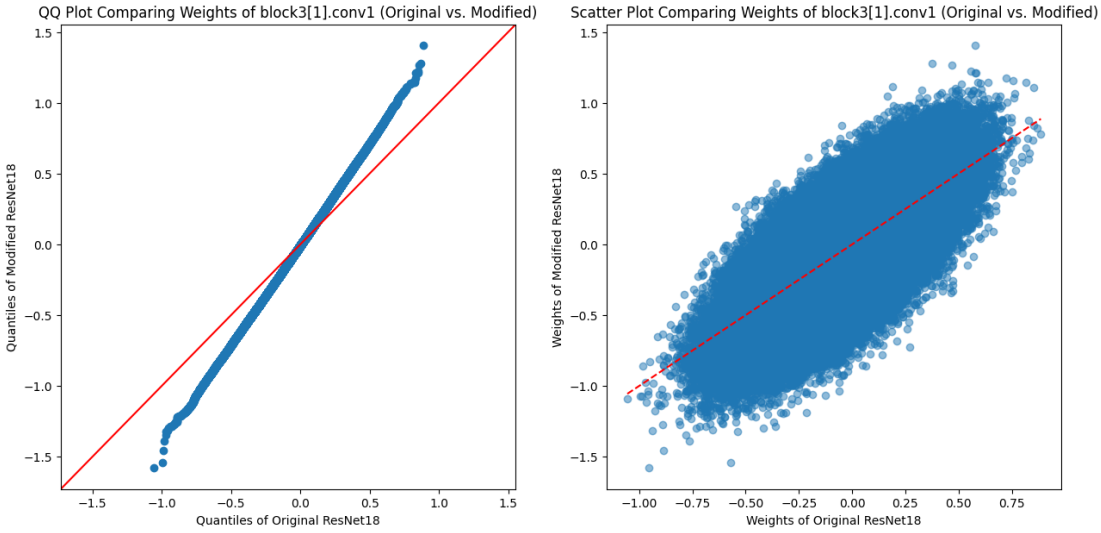

Figure 4.7: weight comparison (original vs modified) block 3 conv. 3

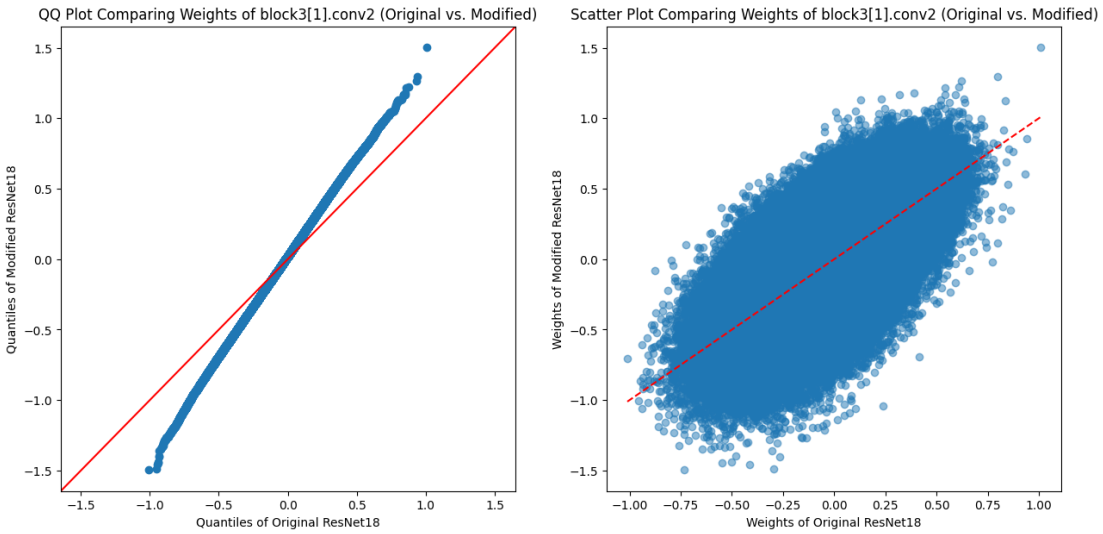

Figure 4.8: weight comparison (original vs modified) block 3 conv. 4

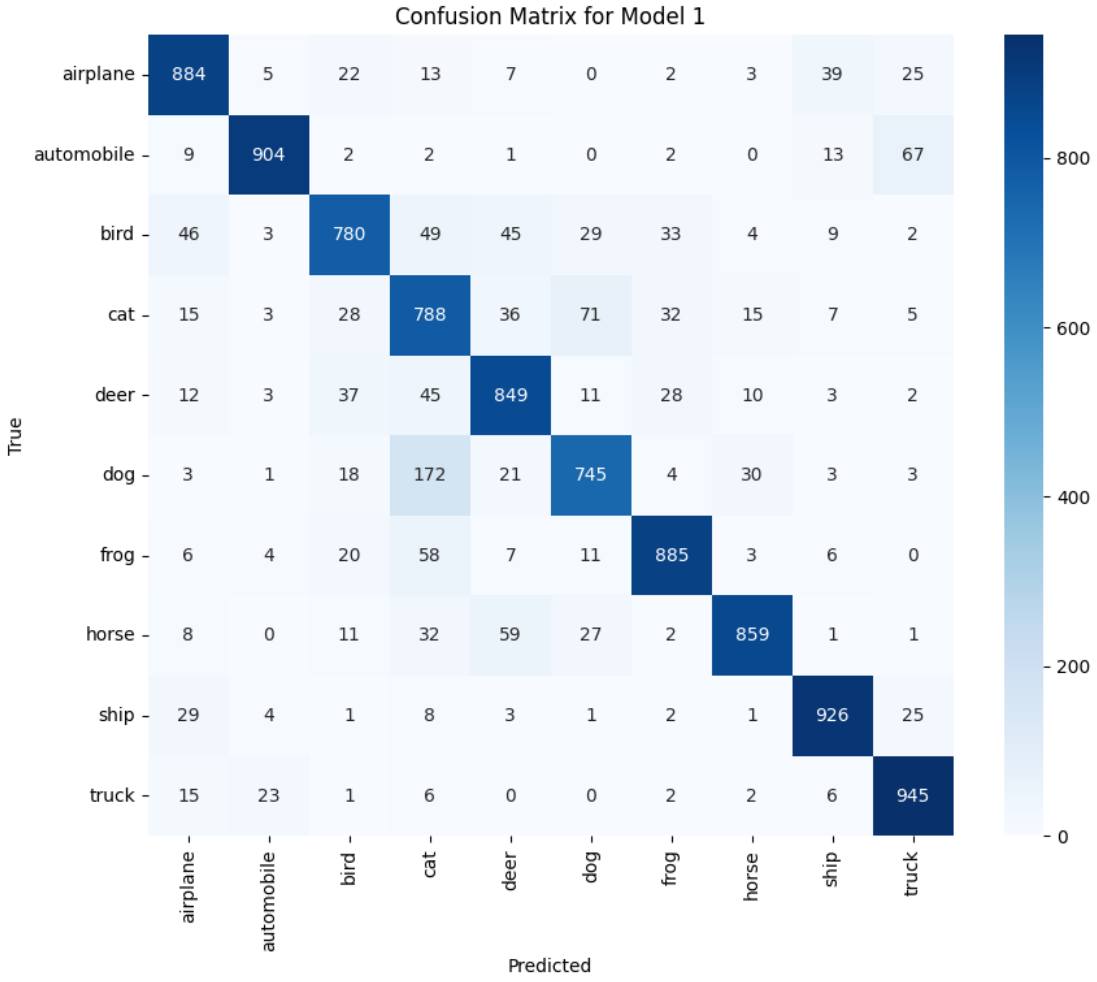

Figure 4.9: Confusion matrix of model 1 (original ResNet-18)

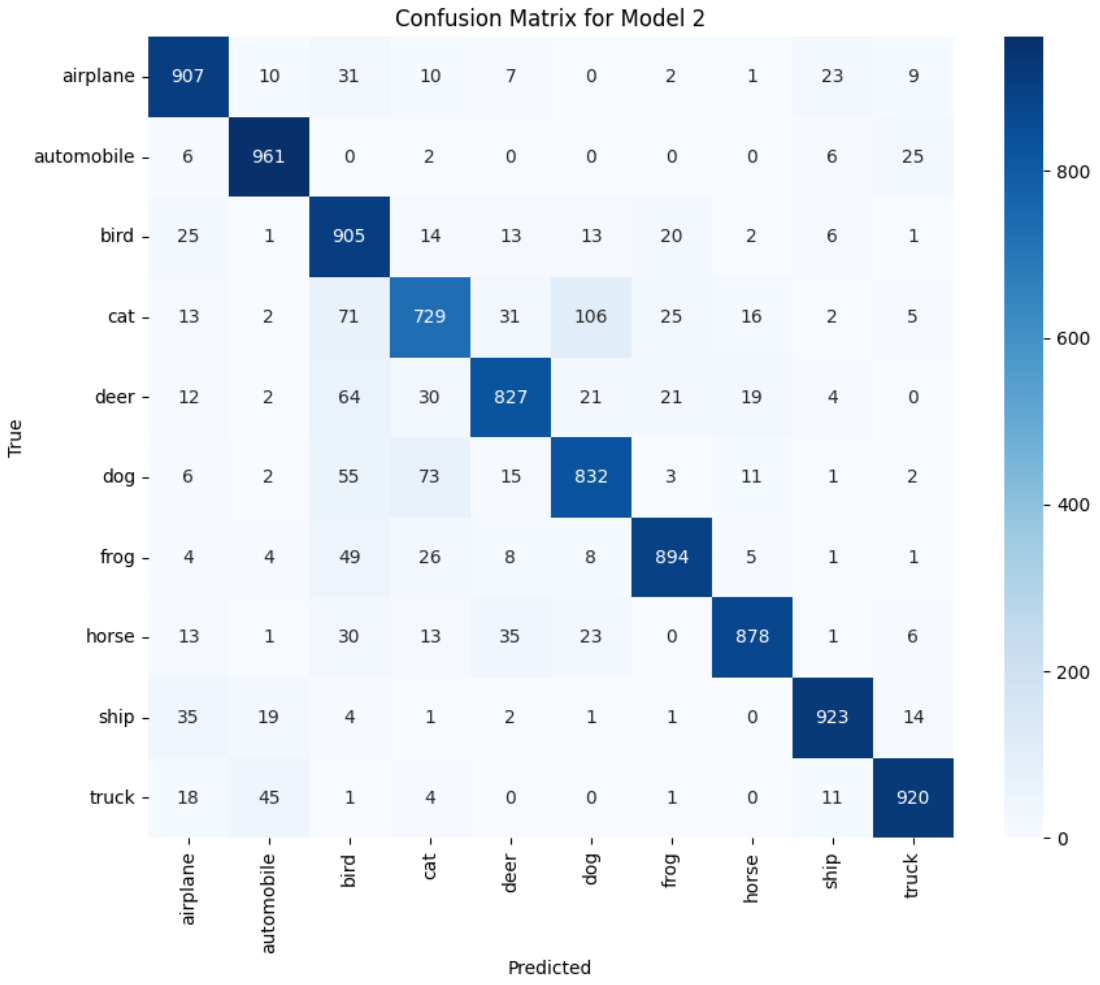

Figure 4.10: Confusion matrix of model 2 (Hybrid model)

For this, we use the confusion matrix which gives us classwise data on the number of images correctly classified for each class and also the misclassified ones. This can be seen in Figures 4.9 and 4.10 which represent the confusion matrix for ResNet-18 and Hybrid model respectively. From these confusion matrices, we can also find classwise accuracies for both models by summing over the correctly classified images for a class over total observations. From Figure 4.11, we can see classwise accuracy for both the models (the points represent the accuracies of the model).

We can also look at the Type 1 and Type 2 errors to know how well our models are performing. Type 1 errors (False Positives) occur when an instance from a different class is incorrectly classified as the target class. Table 4.1, shows the Type 1 error for both the models, and from this we can see that the ResNet-18 model has higher Type 1 errors than the Hybrid model for most of the cases. For example in the table see can see that for the class Airplane the Type 1 error is 143 and 132 in the ResNet-18 and Hybrid model respectively. This reduction suggests that the hybrid model has better generalization abilities and is less prone to incorrectly classifying instances from different classes.

Class	**ResNet-18 Model Type 1 Error**	**Hybrid Model Type 1 Error**
Airplane	143	132
Automobile	46	86
Bird	140	305
Cat	385	173
Deer	179	111
Dog	150	172
Frog	107	72
Horse	68	55
Ship	87	55
Truck	130	63

Table 4.1: Type 1 Errors Comparison for ResNet-18 and Hybrid model

Type 2 errors (False Negatives) occur when an instance of the target class is incorrectly classified as another class. Table 4.2 shows the Type 2 error in both the models. In this, we see that for most of the cases the Hybrid model performs better than the ResNet-18 model. For example, in the class Airplane, the Type 2 error in the Hybrid model is 93 compared to 116 in the ResNet-18 model. This suggests that the transformer encoder helps in capturing features that are critical for accurately identifying certain classes, thereby reducing the likelihood of false negatives. However for classes like "cats" and "deer", this trend is not observed. ResNet-18 performs better in these classes. The reason behind this could be the superiority of the ResNet-18 model in capturing local features and for classes like "cats" and "deer", local features might be important to classify them. In conclusion, we can see that the Hybrid model performs better than the ResNet-18 model both in terms of generalization as well as correctly classifying classes. However in some cases, the Hybrid model has slightly higher Type 1 or

Type 2 error which shows that the improvements in the Hybrid model in metric may come with a minor trade-off in the another.

Class	**ResNet-18 Model Type 2 Error**	**Hybrid Model Type 2 Error**
Airplane	116	93
Automobile	96	39
Bird	220	95
Cat	212	271
Deer	151	173
Dog	255	168
Frog	115	106
Horse	141	122
Ship	74	77
Truck	55	80

Table 4.2: Type 2 Errors Comparison for ResNet-18 and Hybrid model

From Figure 4.11, we can see that for almost all the classes the Hybrid model performs better than the ResNet-18 in terms of accuracy. It could be because the Hybrid model might be better at capturing certain features from the images than the ResNet-18 model due to the introduction of a transformer encoder. But we do see certain classes like "cat" where the accuracy of the Hybrid model (72.9%) is less than that of the ResNet-18 model (78.8%). This suggests that the ResNet-18 model might be better at capturing features related to cats than the Hybrid model.

We find that there is a change in classwise accuracy between the models but to know which class has a significant change in accuracy, we make a 95% confidence interval for each class in both the models using the Bootstrapping technique by taking 1000 random samples from the testing set and finding accuracy for each sample using both models. From the Confidence intervals for each class, as shown in Figure 4.11, we see that there are only 4 classes (Automobile, bird, cat, and dog) for which the Confidence intervals don't overlap indicating that the change in accuracy for these classes between the models might be significant. But from Figure 4.12, can see that for the class ship the median of the two models is different but the distributions of the two models overlap where model 1 is ResNet-18 and model 2 is the Hybrid model. This means that by just seeing if the intervals overlap or not we can't comment on the significance of the change in accuracies and we need further testing to clarify this.

To confirm our results from the visual inspection of the graph, we conduct a statistical test to find if the significance shown in the graph is true or not. Since the distribution of the accuracies of the classes is unknown and not necessarily normally distributed, we will use a non-parametric test to find the significance. We will use the Wilcoxon Signed-Rank test, which is a non-parametric test for

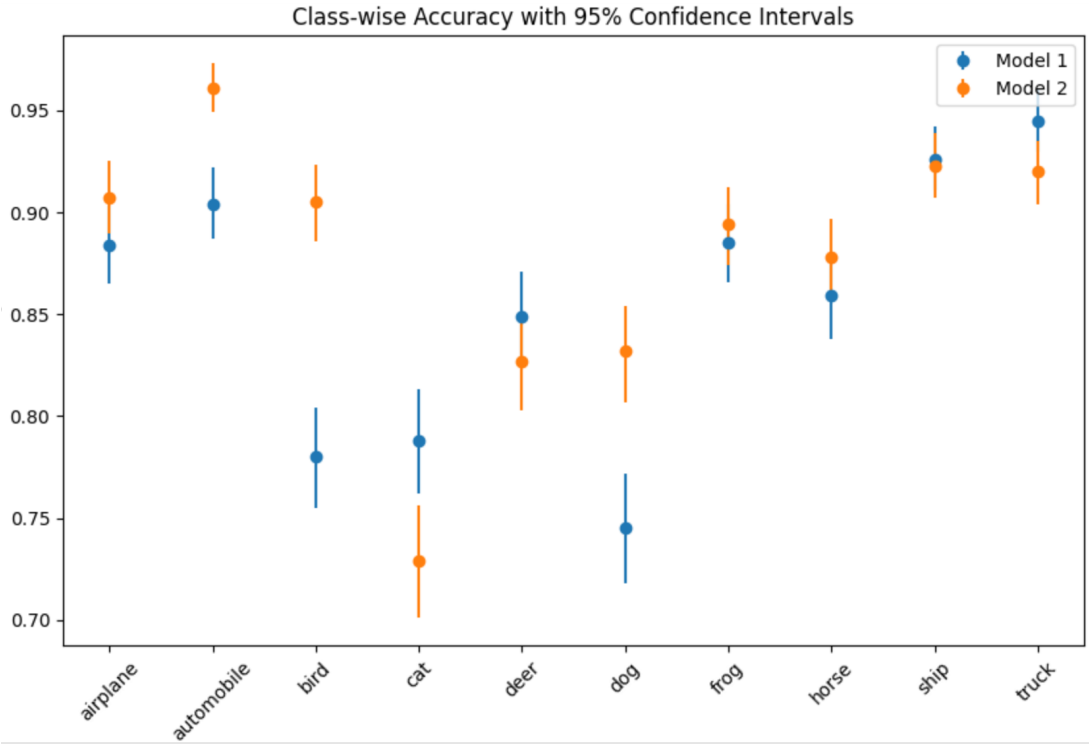

Figure 4.11: 95% Confidence Interval for accuracy for all classes for both original ResNet-18 and Hybrid model. The points represent the accuracy of the model. Model 1 (blue) represents the Original ResNet-18 model and Model 2 (orange) represents the Hybrid model. Only automobiles, birds, cats, and dogs have non-overlapping intervals showing significance.

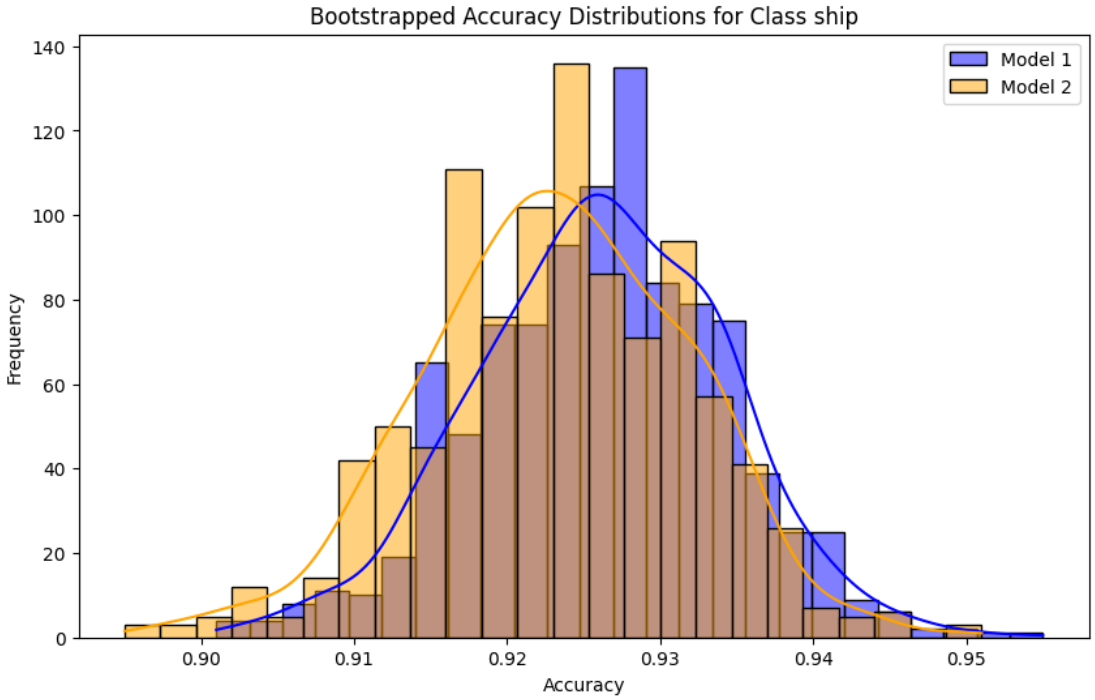

Figure 4.12: Distribution of accuracies for ship class in both the models. They overlap but the medians are different. Model 1 is the ResNet-18 and Model 2 is the Hybrid model.

paired samples. In this case, the paired samples are the accuracies of the two models. Since both models are evaluated on the same dataset, this test is a suitable choice for comparing their performance. The Wilcoxon Signed-Rank test will help us determine whether the observed differences in accuracies between the models are statistically significant.

The objective of this test is to find if the median of the differences between the accuracy of the 2 models for each class is zero or not. It means that if the median of the differences is zero then it means that there is no significant difference in accuracy between the 2 models. Wilcoxon test is more sensitive to the outliers so the median is a good measure to check if there is any significance.

The null hypothesis (H_0): The median of the differences of the accuracy for each class is zero.
The alternative hypothesis (H_A): The median of the differences of the accuracy for each class is not zero.

For the Wilcoxon test, we first calculate the difference in accuracy between the 2 models for each class. We use the samples generated by the Bootstrapping. We then take the absolute differences and rank the differences in ascending order and then put the positive and negative signs back into the ranks as they were initially. For the test statistics, we sum all the positive ranks and negative ranks and then choose the one which is the smallest of the two, and then find the p-value based on that test statistics. If the p-value is less than 0.05 then we can

reject the null hypothesis meaning the differences in the accuracies of the 2 models are significant.

The Wilcoxon Signed-Rank test is well-suited for comparing the median of paired differences as it takes into account both the direction and magnitude of the differences. The test determines whether these absolute differences are positive, negative, or centered around zero. This makes it less sensitive to outliers and makes it robust, unlike tests such as the paired t-test, which rely on means and can be influenced by extreme values. As a result, the Wilcoxon test is a good method for assessing whether the median difference between paired samples is significantly different from zero, especially when the distribution of the differences is unknown or non-normal.

By doing this test, we find that the p-value for all the classes is small and less than 0.05 as shown in Figure 4.14. This means we reject the null hypothesis and that all the classes have a significant change in accuracy between the two models. Although we see significant results from the Wilcoxon test, we also need to see if these changes in accuracy are practically significant or not. To see this, we find the median of the differences in the accuracies of the two models for each class from the Bootstrap samples to see if the median difference is big enough to be significant or not. The negative difference would mean that the Hybrid model outperformed the ResNet-18 model and the positive difference would mean the ResNet-18 model performed better. This is shown in Figure 4.15 and from this we can that only 4 classes (automobile, bird, cat, and dog) have differences higher than 0.05 or 5%. This suggests that although all classes show significant changes in accuracy only these four classes mentioned above actually have any practical significance which also aligns with our results from Figure 4.11 where only these classes had nonoverlapping intervals. The other classes have a median difference of less than 5% which could be due to the large sample size and consistent direction of differences, amplifying statistical significance even for small changes.

Class	**Positive Ranks Sum**	**Negative Ranks Sum**
Airplane	5.5	500494.5
Automobile	0.0	500500.0
Bird	0.0	500500.0
Cat	500500.0	0.0
Deer	192716.0	282109.0
Dog	0.0	500500.0
Frog	12490.5	478054.5
Horse	7921.0	490580.0
Ship	321428.5	151449.5
Truck	500485.5	14.5

Table 4.3: Positive and Negative Ranks for Each Class

From the Wilcoxon test, we also see some test statistics are 0 (for automobile, bird, cat, and dog) (Figure 4.13), meaning that all differences for that class are

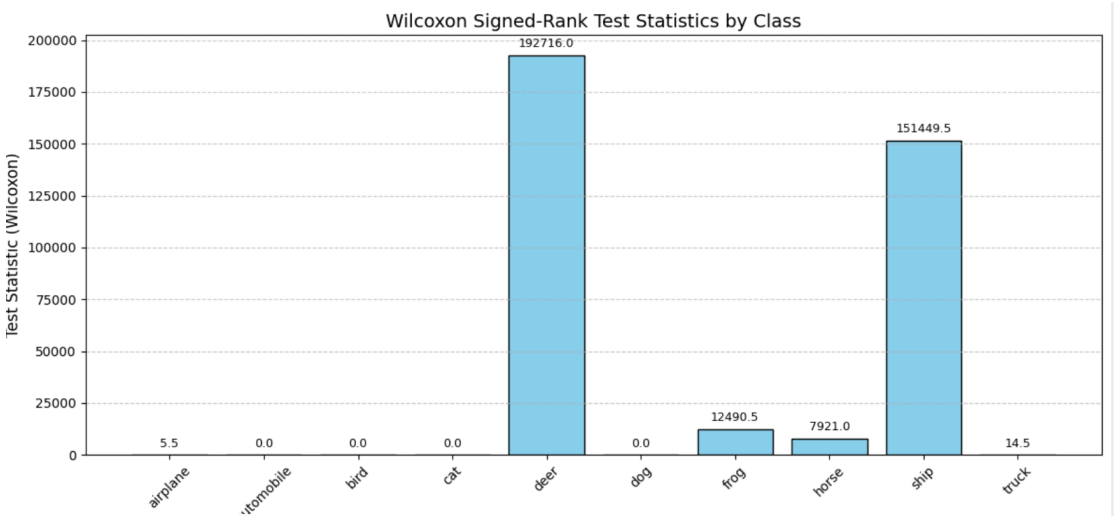

Figure 4.13: Test statistics of all classes using Wilcoxon signed rank test.

either positive or negative, making the rank sum zero. This suggests a consistent direction of difference between the two models for these classes. This is further verified by the results from Table 4.3, which shows the positive and negative rank sum for all the classes. From this, we see that for classes like automobile, bird, and dog, the sum of ranks are all negative and for cat the rank sum is all positive, hence the test statistics is 0 since we take the minimum of the two sums. This just means that the Hybrid model out performed the ResNet-18 model in these 3 classes (automobile, bird, and dog) but for cat class ResNet-18 outperformed Hybrid model which we also saw from Figure 4.11.

From Figure 4.13, we also we that for some classes like airplanes, the test statistics are in fractions. This is due to the presence of ties in the ranked sum which arises due to the same magnitude (ignoring the sign) of some paired sample differences. The Wilcoxon test assigns average ranks to these tied differences, which can result in fractional ranks. This kind of behavior is normal in the presence of ties in the sample and does not make the test invalid. We also confirm this from Figure 4.16, which shows how many ties are present in each class which are calculated from the Wilcoxon Test. This shows that due to this, some test statistics are in fractions.

```
Class airplane:
  p value (Wilcoxon): 3.203497385464335e-165
Class automobile:
  p value (Wilcoxon): 3.0935422650140584e-165
Class bird:
  p value (Wilcoxon): 3.199285369479734e-165
Class cat:
  p value (Wilcoxon): 3.247562413859079e-165
Class deer:
  p value (Wilcoxon): 3.5668159804608184e-07
Class dog:
  p value (Wilcoxon): 3.211113342174579e-165
Class frog:
  p value (Wilcoxon): 1.450742543698392e-147
Class horse:
  p value (Wilcoxon): 1.0324693062369808e-154
Class ship:
  p value (Wilcoxon): 2.711745956424567e-22
Class truck:
  p value (Wilcoxon): 3.2925997984297416e-165
```

Figure 4.14: p-values of all classes using Wilcoxon signed rank test. All classes are significant.

```
Median of Differences Between Models:
Class airplane: Median Difference = -0.0230
Class automobile: Median Difference = -0.0570
Class bird: Median Difference = -0.1250
Class cat: Median Difference = 0.0590
Class deer: Median Difference = 0.0220
Class dog: Median Difference = -0.0860
Class frog: Median Difference = -0.0090
Class horse: Median Difference = -0.0190
Class ship: Median Difference = 0.0030
Class truck: Median Difference = 0.0250
```

Figure 4.15: The median of the differences between the ResNet-18 and the Hybrid model calculated from the Bootstrap samples.

```
Number of ties for each class:
Class airplane: 974 ties
Class automobile: 996 ties
Class bird: 980 ties
Class cat: 955 ties
Class deer: 998 ties
Class dog: 991 ties
Class frog: 996 ties
Class horse: 992 ties
Class ship: 988 ties
Class truck: 987 ties
```

Figure 4.16: Number of ties in each class of the dataset from the Wilcoxon Signed Rank Test

Chapter 5

Conclusion

In this thesis, we compared the performance of two models: ResNet-18 and a Hybrid model where the 4^{th} of ResNet-18 was replaced with a transformer encoder. The primary objective was to understand how the substitution affects model performance on the CIFAR-10 dataset and to investigate the changes in weights and classification accuracy.

We compared the weights of the original ResNet-18 and the Hybrid model using a quantile-quantile (QQ) plot. This analysis revealed that the weights of the first three blocks for both the models remained the same in distribution but in terms of magnitude, the Hybrid model had higher weights. This meant that due to the architectural change in the Hybrid model, the weights were adapted in the earlier layers, and as the layers come closer to the transformer encoder in the 4th block, these adaptations increase which can also be seen from the scatterplot of these two models' weights as the points had bigger scatter.

Both models were evaluated on the CIFAR-10 dataset. The ResNet-18 model achieved a validation accuracy of approximately 86%, while the hybrid model attained a slightly higher accuracy of around 88%. This suggests that the transformer encoder in the 4^{th} block contributed positively to the overall performance. Classwise accuracy was also analyzed using confusion matrices. The hybrid model showed improvements in most of the classes however, a slight decrease in accuracy was noted for the "cat" class, indicating that certain features might be better captured by traditional convolutional layers. By making the 95% Confidence interval for both models, we found that only four classes (automobile, cat, bird, and dog) show significance in accuracy change as their intervals did not overlap. To statistically validate the observed changes in accuracy, a Wilcoxon signed-rank test was conducted. The results indicated that the changes in accuracy between the two models were significant for all classes even though their confidence intervals overlapped. To further investigate this we checked the median of the differences in accuracy of the two models to see if the median difference is even big enough to be practical. We found that for most of the classes, this difference was less than 5%, and only the 4 classes (Automobile, bird, cat, and dog) for which the intervals didn't overlap have this difference of more than 5% and this was in the favour of the Hybrid model meaning it performed better than ResNet-18 model for all these 4 classes other than cat.

Our findings align with prior research that has demonstrated the effectiveness of transformer models in various computer vision tasks. For example, Dosovitskiy et al. (2020) introduced the Vision Transformer (ViT), which showed that transformers could achieve competitive performance with convolutional networks on image classification tasks [4]. Similarly, recent studies by Parmar et al. (2018) and Carion et al. (2020) have highlighted the benefits of integrating transformer architectures with CNNs for enhanced feature extraction and image classification [5, 6].

In the future, we can explore more configurations of transformer encoders within CNN architectures. Instead of replacing just the 4^{th} block of ResNet-18, we can replace other blocks or a combination of blocks to see how it impacts the performance of the model. We can also include the positional encoding in the Transformer block to see if adding this has any effect on the effectiveness of the model. Additionally, experimenting with different datasets could provide further insights into the generalizability and versatility of hybrid models. We can also investigate the impact of regularization techniques like the L2 norm, which might also yield improvements in model performance. Another future work could be to do a test to find out after which block the distance becomes apparent between the images of different classes. For this, we can find Frobenius distance between the images and take into account the different dimensions of the blocks as well as the separation measure and compactness of the images to make sure that the distance is comparable.

In conclusion, we demonstrate that incorporating a transformer encoder into a standard CNN architecture like ResNet-18 can lead to notable improvements in classification accuracy. The hybrid model leverages the strengths of both CNNs and transformers, offering a balanced approach to feature extraction and representation. These findings support the idea of integrating transformers with CNNs for computer vision tasks, paving the way for future innovations in model architecture design.

Bibliography

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

[2] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. `https://D2L.ai`.

[3] Huanhuan Zhang. Applying deep learning to medical imaging: A review. *Applied Sciences*, 13(18):10521, 2023.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2021.

[5] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. *arXiv preprint arXiv:2005.12872*, 2020.

[6] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. *arXiv preprint arXiv:1802.05751*, 2018.

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

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2015.

[10] PyTorch Documentation. *torch.nn.AvgPool2d*, 2023. `https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html`.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, July 2015. PMLR.

[12] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In Samy Bengio, Hanna Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 2483–2493. Curran Associates, Inc., 2018.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[14] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. Available at `https://arxiv.org/abs/1607.06450`.

[15] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, May 2012.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.