Project Overview

In this project, I trained a Convolutional Neural Network (CNN) that predicted what category an image belonged to. To do so, I used the provided *CIFAR-10* image dataset. This report outlines my process and findings.

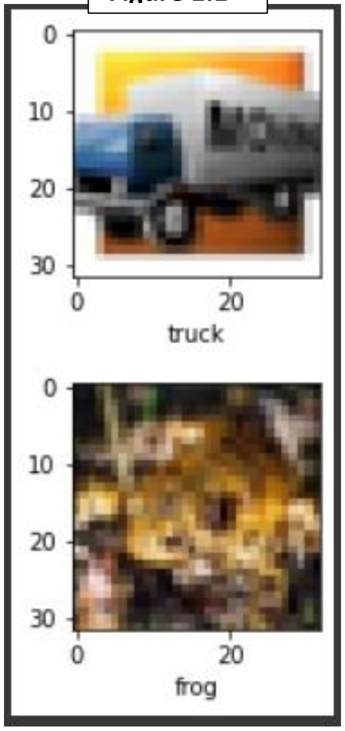
Framing the Problem

The objective of this project was to predict what category an image belonged to using a neural network. The dataset was the *CIFAR-10* set from Kaggle, which was a collection of 60,000 images partitioned evenly across ten different categories.

Since the data was labelled, the problem was a supervised learning problem. It was a classification problem, specifically a discrete multicategory classification problem since it involved classifying data into multiple separate categories. There was no continuous stream of data coming into the project, and no need to adapt to future data. This meant that batch learning was adequate for training my models. I decided to use accuracy scores to evaluate my model, since I cared most about the number of correctly classified images.

Data Visualization and Cleaning

Figure 1.1

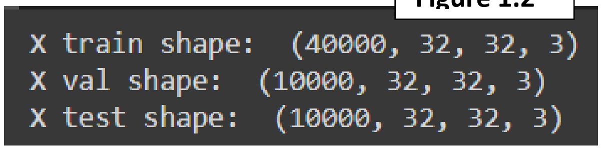


Before loading the dataset, I read its description on Kaggle. As mentioned above, the dataset contained 60,000 images spread across ten categories, meaning there was 6,000 images per category. The categories were airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each image was 32 x 32 pixels, and each pixel had a rgb color value between 0 and 255.

I loaded the data directly from Kaggle into my Colab notebook, since this was easiest. I then plotted several images, as seen in Figure 1.1. Figure 1.1 showed the images were primitive and sometimes hard to identify to the human eye. This was not surprising, considering they were only 32 x 32 pixels.

The data was pre-partitioned into a train and test set, with 50,000 images in the train set and 10,000 in the test set. I created a validation set of 10,000 images out of the train set, since I would later need a validation set for training my neural network.

Figure 1.2



```
X train shape:  (40000, 32, 32, 3)
X val shape:  (10000, 32, 32, 3)
X test shape:  (10000, 32, 32, 3)
```

I also confirmed that there were 40,000 images in my train set, 10,000 in my validation set, and 10,000 in my test set, as seen in Figure 1.2. Figure 1.2 also showed that the images had three dimensions: width, height, and color.

Now that I had a train, test, and validation set, the last step was to normalize the data. Since the pixels of each image held a value between 0 and 255, I divided by 255 to normalize all values from a range of 0 to 1.

Training a Convolutional Neural Network

CNN Overview

For my neural network I chose a Convolutional Neural Network (CNN). A CNN is a deep learning neural network that processes structured arrays of data. It is a feed-forward neural network that uses one or more hidden layers to detect key features from input. It then feeds those features through one or more fully connected layers to make a prediction.

The core building block of a CNN is the convolutional layer. This layer takes in input and filters it using a kernel to determine activation. A kernel is a small matrix of adjustable size which is moved across the entire input matrix. At each step, the dot product of the kernel and the input is saved to a new matrix, called a feature map. The feature map is then passed through a nonlinear activation function, which decides which neurons will fire (i.e. are activated). The resulting feature map is the output of the convolutional layer.

Usually, a pooling layer comes after each convolutional layer. This layer is used to downsize the feature map produced by the convolutional layer. It uses a kernel to detect and preserve the most important parts of the feature map, and then discards the rest. This helps reduce overfitting and keeps calculations simpler, making the network more efficient and capable of handling large amounts of input.

A CNN has at least one convolutional/pooling layer, although it often has more than one. After these layers it has at least one fully connected layer, which produces a probability between 0 and 1. The last fully connected layer acts as an output layer.

CNN Hyperparameters

I chose TensorFlow, specifically the Keras Deep Learning library, to build my Convolutional Neural Network. There are many hyperparameters for a CNN and covering them all is outside the scope of this report. However, the essential ones are explained below.

The following hyperparameters are essential to the convolutional layer in Keras: *filter*, *kernel_size*, *strides*, *padding*, and *activation*. *Filter* is an integer that specifies the dimensionality of the output space (i.e. the number of output filters in the convolution). *Kernel_size* is a tuple of two integers and specifies the width and height of the kernel. *Strides* is a tuple of two integers that specifies the strides of the kernel along the height and width of the input. By default, it is set to (1,1). *Padding* is the padding added to the input when it is being processed. When *padding* is set to ‘same’ and *strides* to (1,1), the output has the same size as the input. This is the combination I used for my CNN. Finally, *activation* is the function that decides which neurons are activated. I used the ‘relu’ function for my activation function.

The max pooling layer in Keras has one important hyperparameter, *pool_size*. This is a tuple that specifies the window size which the maximum value is taken from. *Pool_size* is usually set to (2,2) which takes the maximum value over a two by two window. This reduces the input size by a factor of two, and is the value I used for my pooling layers.

The fully connected layer has two important hyperparameters: an integer that specifies the number of nodes in the layer, and *activation,* which again is the function that determines which neurons are activated. This function is usually set to ‘*softmax,*’ which is the value I used for my CNN.

When compiling a CNN in Keras, the most important hyperparameters are *optimizer, loss,* and *metrics*. *Optimizer* is the stochastic gradient descent optimization algorithm used in training the neural network. I used the ‘Adam’ optimizer, which is commonly used for training CNNs. *Loss* is the function used to estimate the loss of the model during training, so that the weights can be updated on the next evaluation. I used ‘categorical_crossentropy,’ which is the default loss function for multi-class classification. Finally, *metrics* is the metric used to evaluate the model during training and testing. I used ‘accuracy,’ since that was my pre-determined metric for evaluating my networks.

Finding the Optimal Model

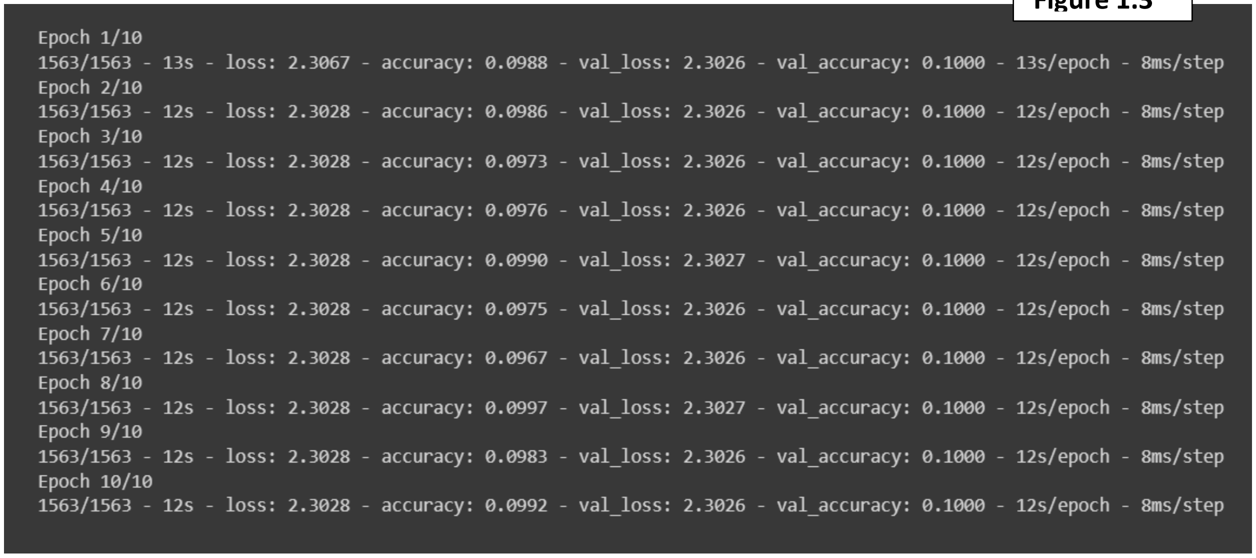
Since there were so many hyperparameters, I decided to use Keras' tuner library to find the optimal hyperparameters for my dataset. I first set up a basic CNN with one convolutional layer, one pooling layer, and a feed forward (dense) layer that fed into the dense output layer of 10 neurons.

There were two hyperparameters that I wanted to tune in the convolutional layer: *filter* and *kernel_size*. For *filter* I set up a search space of 32 - 128, with a step size of 16. For *kernel_size*, I set up a search space with values between 2 and 5. The remaining parameters were set to the values mentioned above, in the hypertuning section.

Next, I added a pooling layer with a *pool_size* of (2,2). I then added a flattening layer, which was necessary to reduce the pooling layer's output into a format the dense layer could interpret. For the number of neurons in the dense layer, I set up 32 – 128 search space with steps of 16. The output of this dense layer fed into in the final output layer, which classified the input into one of 10 classes.

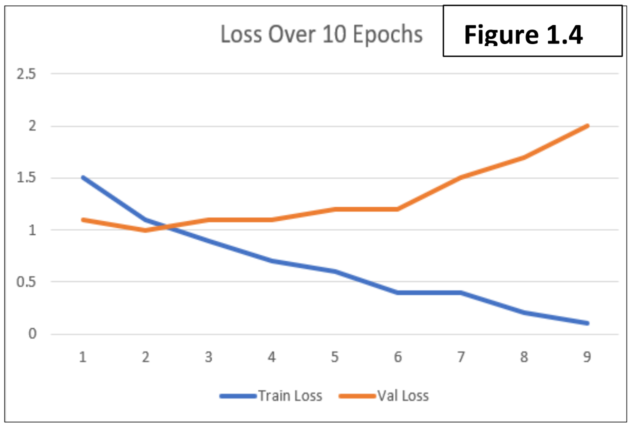
I then compiled the CNN using the hyperparameters mentioned in the hypertuning section, using my train and validation set. I used a batch size of 32 and trained for 10 epochs. The results were abysmal, as shown in Figure 1.3.

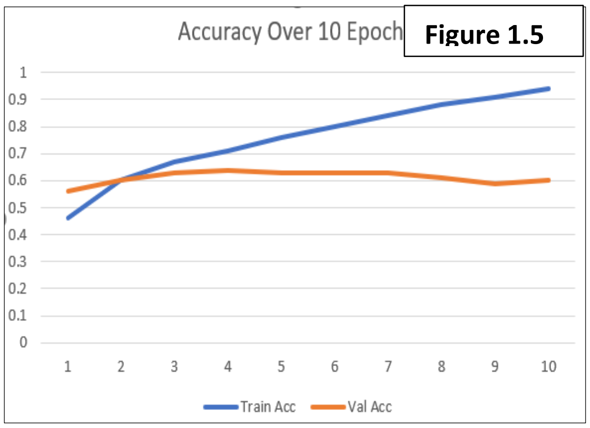
Figure 1.3



```
Epoch 1/10
1563/1563 - 13s - loss: 2.3067 - accuracy: 0.0988 - val_loss: 2.3026 - val_accuracy: 0.1000 - 13s/epoch - 8ms/step
Epoch 2/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0986 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 3/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0973 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 4/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0976 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 5/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0990 - val_loss: 2.3027 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 6/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0975 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 7/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0967 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 8/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0997 - val_loss: 2.3027 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 9/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0983 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
Epoch 10/10
1563/1563 - 12s - loss: 2.3028 - accuracy: 0.0992 - val_loss: 2.3026 - val_accuracy: 0.1000 - 12s/epoch - 8ms/step
```

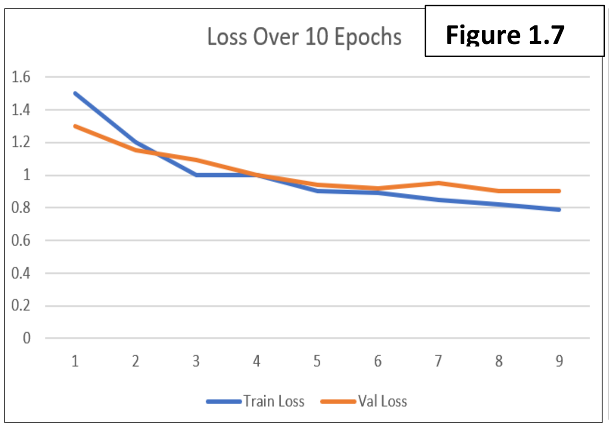
I did not bother to plot the accuracy and loss since the model was not improving at all. Clearly the architecture was too simple. To solve this, I added a second convolutional and pooling layer with the same search spaces/values as the first and compiled and fit the new model.

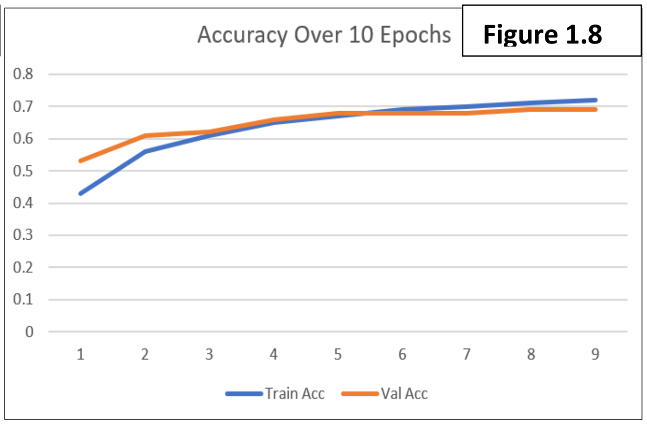




The new model had a significant increase in accuracy but the overfit very badly, as shown in Figures 1.4 and 1.5 (above). In Figure 1.5 we can see the accuracy of the validation set stabilized around 0.6 percent, but the accuracy of the training set continues to climb, indicating extreme overfitting. In Figure 1.5 the loss drops steadily for the test set which seems ideal but is actually a result of overfitting. Conversely, the loss for the validation set grows over time, indicating the model is not improving over each epoch.

To reduce overfitting, I added a dropout layer after each pooling layer. I then recompiled the model with the following dropout values: [0.1, 0.2, 0.5, 0.7]. The model with a dropout rate of 0.2 performed the best. Its training performance is shown in Figures 1.7 and 1.8.





This model accurately classified 72 percent of the train set and 69 percent of the validation set. The loss for the training set was .77, and the loss of the validation set was .89. Overall, this model was promising, especially since it showed signs of improving with more epochs. (Note: I also trained a model the same hyperparameters as above, but with a third convolutional and pooling layer. However, this model did not outperform the model with two convolution/pooling layers, so I discarded it).

There was one final hyperparameter to tune, *batch_size*. I evaluated the following *batch_size* values: [64, 128, 256]. The model trained on batches of 128 performed the best. It's training data is shown in Figures 1.8 and 1.9.

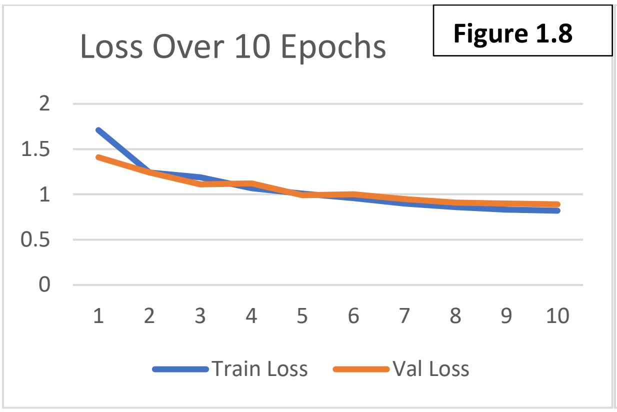


Figure 1.8

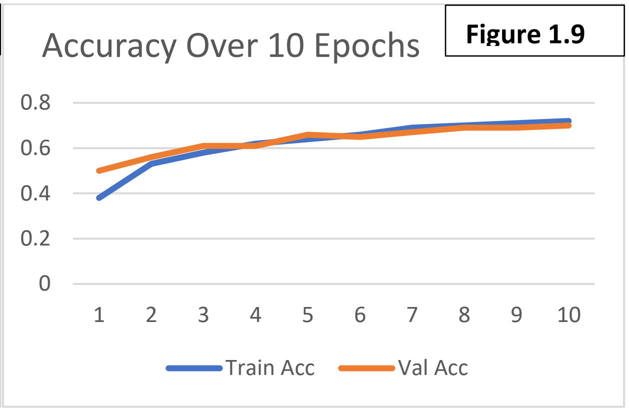
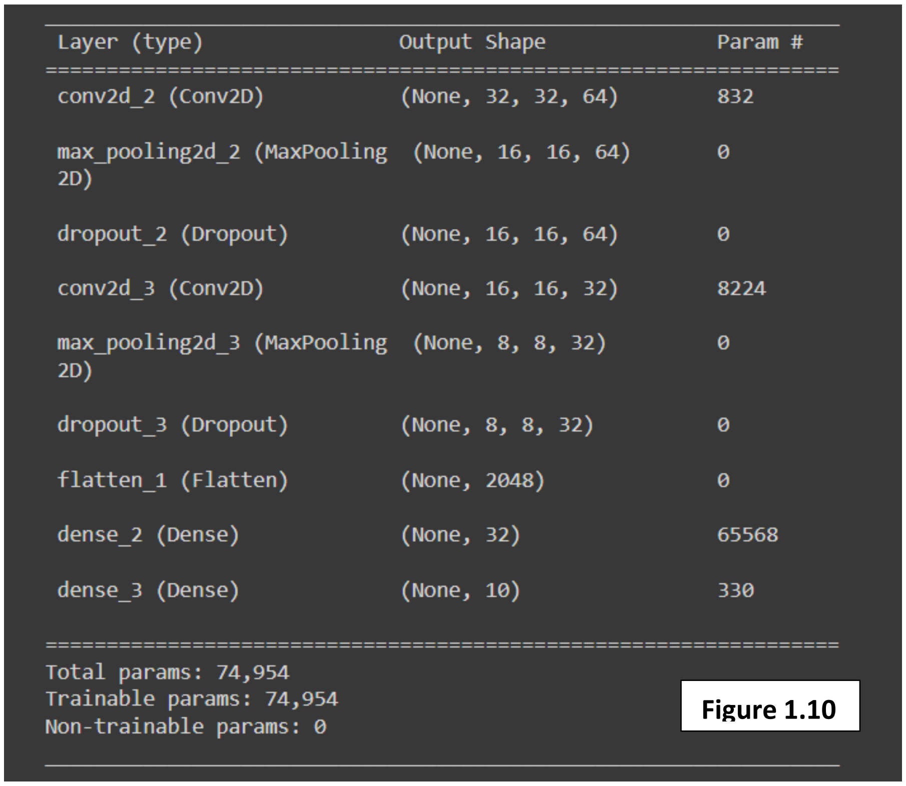


Figure 1.9


```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 32, 32, 64)        832

 max_pooling2d_2 (MaxPooling  (None, 16, 16, 64)       0
 2D)

 dropout_2 (Dropout)         (None, 16, 16, 64)        0

 conv2d_3 (Conv2D)           (None, 16, 16, 32)        8224

 max_pooling2d_3 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 dropout_3 (Dropout)         (None, 8, 8, 32)          0

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 32)                65568

 dense_3 (Dense)             (None, 10)                330

=================================================================
Total params: 74,954
Trainable params: 74,954
Non-trainable params: 0
```
Figure 1.10

This model accurately classified 72 percent of the training set and 70 percent of the validation set. It had a train loss of .82 and validation loss of .89.

This model performed better all others, with the highest accuracy scores and smallest overfitting gap. Its architecture is pictured to the left, in Figure 1.10.

As we can see from Figure 1.10, the first convolutional layer had a *filter* value of 64, the second convolutional layer had a *filter* value of 32, and the first dense layer had 32 nodes that fed into the final dense output layer of 10. Overall, the model was surprisingly small and efficient.

Now that I had determined the optimal hyperparameters for my dataset, I was ready to experiment with optimal epochs. I declared a new model with the optimal hyperparameters and trained it on the following epoch values: [10, 20, 25, 30]. I found that 25 epochs were optimal, and beyond this the validation loss began to increase and the validation accuracy to decrease. The model trained on 25 epochs accurately classified 70 percent of the train, validation, and test sets. Its training performance is pictured below in Figures 1.11 and 1.12.

Figure 1.11

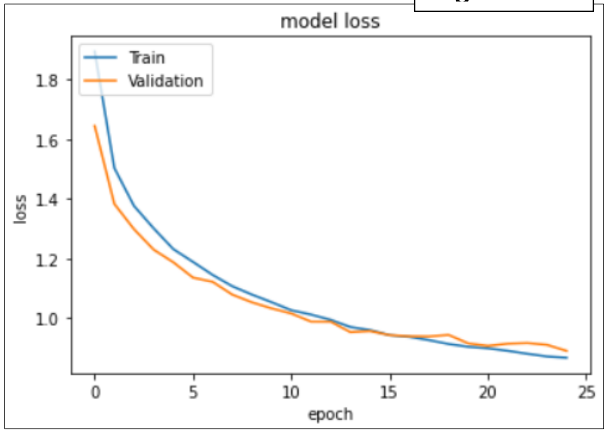
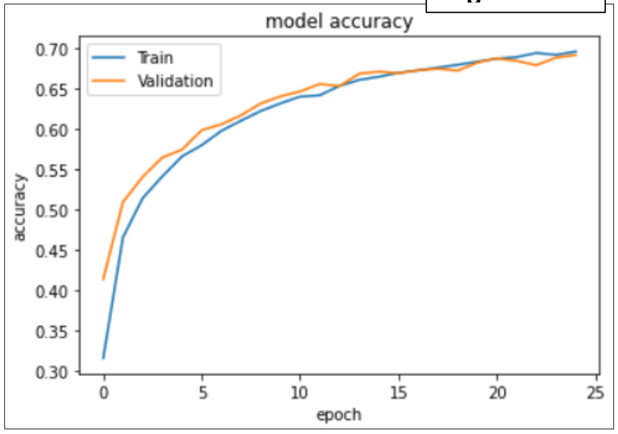


Figure 1.12



Optimal Model Summary

After extensive hypertuning using Kera's tuner feature, as well as experimenting with search spaces for dropout rates, batch sizes, and epochs, I found an optimal model that accurately classified 70 percent of the images in the train set and test set. This model had a convolutional layer with filter size of 64, followed by a pooling layer and dropout layer with a rate of 0.2. It then had a second convolutional layer with a filter size of 32 and a similar pooling and dropout layer. Next, it had a dense layer with 32 neurons, and finally an output layer with 10 neurons.

The hypermeters for this model are pictured below in Figure 1.13.

Figure 1.13

Architecture	**Hyperparameter**

Convolutional Layer 1	filter = 64
	kernel_size = (2,2)
	strides = (1,1)
	padding = 'same'
	activation = 'relu'

Pooling Layer 1	pool_size = (2,2)
Dropout Layer 1	rate = 0.2

Convolutional Layer 2	filter = 32
	kernel_size = (2,2)
	strides = (1,1)
	padding = ‘same’
	activation = ‘relu’

Pooling Layer 2	pool_size = (2,2)
Dropout Layer 2	rate = 0.2
Flatten Layer	N/A
Dense Layer 1	size = 32
Dense Layer 2 (Output)	size = 10

Compile (compiling the model)	optimizer = ‘Adam’
	loss = ‘categorical_crossentropy’
	metrics = ‘accuracy’

Fit (fitting the model)	batch_size = 128
	epochs = 25

Conclusion

In this assignment I trained a Convolutional Neural Network to predict images from *CIFAR-10* dataset, which was a dataset of 60,000 simple images across 10 categories. I used Keras’ tuner library to find the optimal hyperparameters and architecture for the dataset and built a CNN model using what I found. The model accurately classified 70 percent of the train and test set.

My primary takeaway from this assignment was that training a neural network is a heavy endeavor, and a tuning library and plenty of statistical analysis is crucial to success. I spent significantly more time finding tuning and training my CNN model than any other machine learning model this semester, since there were so many hyperparameters. Without the Keras tuner library and regularly analysis of my training accuracy and loss, it is highly unlikely I would have improved my model as much as I did.

Even with extnesive tuning, my model was only moderately successful with a train and test classification accuracy of 70 percent. It is possible that additional techniques such as weight regularization or dataset augmentation might improve the model. It is also possible that a more advanced recurrent neural network model such as a LSTM or GRU neural network might perform better than a CNN model.