Deep Learning for image classification

Author: B924007

I. Introduction

Convolutional Neural Networks (CNNs) are a type of Artificial Neural Network (ANN), primarily used in deep learning to analyze and classify visual imagery. A CNN network is able to take an image input and assign importance (through weighting and bias) to particular features of the image in order to identify and class different objects within the image. Some of the most relevant real-life applications of this technology include: Facial recognition, self-driving vehicles, text prediction, cancer detection, and 3D medical image segmentation. With such a vast range of applications, CNNs can be applied to a range of industries to optimize the performance of tasks using machine learning **[1]**.

The idea of CNNs was inspired from the human brain connectivity pattern of neurons as it replicates the structure of these neurons **[2]**. Neocognitron (1980) was the first architecture of this kind, making use of feature extraction, pooling layers, and using the architecture for recognition and classification. CNNs further developed with the introduction of the LeNet-5 (1989) architecture which was developed for handwritten digit recognition. LeNet-5 make use of several convolutional and pooling layers to train a model in order to recognize handwriting. Beyond this, major strides forward in the development of CNN took place in 2010 with the formation of ImageNet which is a large-scale dataset consisting of millions of images which is used to train CNN models. Making use of the weights from ImageNet allowed CNNs performance rates to improve greatly, with the presence of pre-trained models such as AlexNet (2012) and VGGNet (2014). GoogLeNet (2014) further developed CNNs by making use of inception modules and stacking layers, after which ResNet (2015) popularized skipping connections layers and creating deeper models with more layers.

The task at hand involves making use of an image dataset split into training and testing data to create a CNN algorithm, training the model, and then testing the performance of the model. Based on the results, the CNN model can be optimized in order to maximize testing accuracy by using many different techniques such as data augmentation, selecting the best optimizer, changing activation type, batch normalization, and changing the models learning rate to name a few. Finally, the model can be compared to an existing pre-trained CNN architecture to see how well it performs in comparison.

A. Overview of Results

Using a simple single layer CNN resulted in a very low performing model with a testing accuracy of only 30%. Once the model is improved by stacking convolutional layers and optimizing optimizers and activation functions, the model performs significantly better with accuracies of up to 65%. Despite this, there were signs of overfitting in the data which resulted in the implementation of a 40% dropout rate to reduce the effect of overfitting. Finally, the 'optimal' model created is compared to VGG16 (an existing pre-trained model) which had a test accuracy of 88.6% with no signs of overfitting over 5 epochs.

II. Data & preliminary analysis

```
batch_size = 32 #batch for each iteration
image_size = (128,128) #size of images

# Read image data from file directory
train_dir = 'imageset/imageset/train'
test_dir = 'imageset/imageset/val'

# Use 'image_dataset_from_directory' to read images
train_dataset = image_dataset_from_directory(train_dir, shuffle = True, batch_size = batch_size, image_size = image_size)
test_dataset = image_dataset_from_directory(test_dir, shuffle = True, batch_size = batch_size, image_size = image_size)
```

```
Found 9469 files belonging to 10 classes.
Found 3925 files belonging to 10 classes.
```

Figure 1 - Code to read in image data.

The image set being used for this task is a collection of 13394 images which are split into training (9469 images [70%]) and testing (3925 images [30%]) data. These images are further separated into 10 classes based on the distinguishing feature within each image. For example, all chainsaw images are grouped into one class. Other classes include things like: Fish, trombone, dogs, trucks, cathedrals etc. Below is a visualization of some random images and their labels that exist in the training dataset:

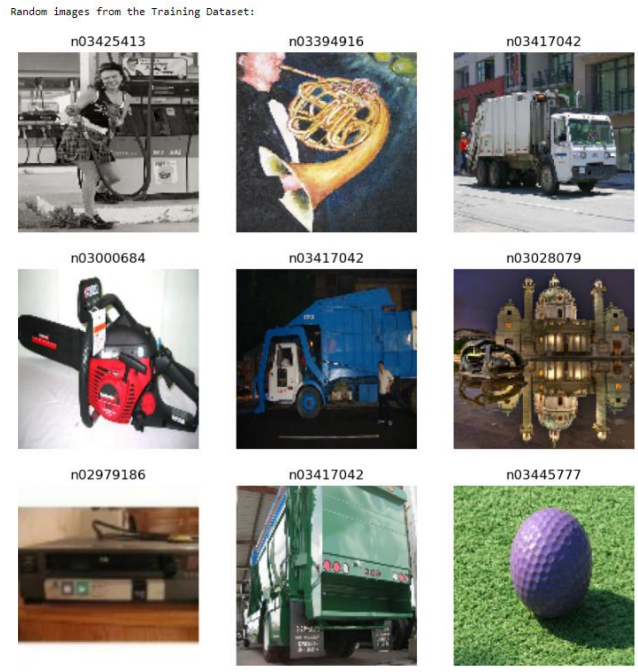

Figure 2 - Training dataset images.

When calling the images into python, a batch size of 32 is selected. This is a typical batch size as it refers to the number of images present within each batch that will be tested using a CNN. The CNN will work through each

batch, completing one iteration at a time, until it has covered all images in the dataset to complete one full epoch. The images in the dataset have also been shuffled so that the model is not constantly getting the images in the exact same order which could cause data overfitting and a much lower model performance when tested on the testing dataset consisting of a completely new set of images to the training dataset.

The image size is set as (128,128), leading to a slight reduction in quality of images. This is done purposely as it will reduce the computational demands significantly. Having a larger image size with greater resolution leads to models that take longer to compute, and require greater computational power, when the images will still reduce in resolution due to the several pooling layers in the CNN architecture. On the other hand, having a resolution which is too low will significantly reduce CNN performance since the model will not be able to find any distinguishing features to classify images **[3]**.

```
train_dataset, test_dataset = train_dataset / 255.0, test_dataset / 255.0
```

Figure 3 - Normalize pixel size of images.

Images have also been normalized before being used in a CNN. This is done because the values of pixels intensify images and can range up to 255 pixels. Normalizing between 0 and 1 pixels makes the model easier to work with. Normalizing the pixel values also makes the training process more stable and efficient, as updated weights become more stable leading to faster convergence of data.

III. Methods

In order to create a high-functioning CNN model which has a high testing accuracy, several steps are taken. Firstly, a very simple CNN model consisting of just a single convolutional layer and max pooling layer is trained using the dataset. This model can be used to understand all the different stages of a standard basic CNN model structure.

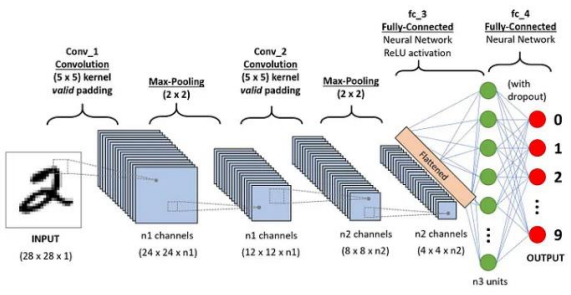

*Figure 4 - Standard CNN Model structure **[4]***

```
# Define functions for a single layer basic CNN Model

class Basic_Model(Model):
    def __init__(self):
        super(Basic_Model, self).__init__()
        # Define the conv layer (32 filter, 3 kernel size)
        self.conv1 = Conv2D(32,3, activation = 'relu')
        # Define max pool layer
        self.pool1 = MaxPool2D()
        # Convert 2D image to 1D vector
        self.flatten = Flatten()
        # Define fully connected layer
        self.d1 = Dense(128, activation = 'relu')
        # Dense further to have output size 10
```

Figure 5 - Simple CNN Model architecture

A convolutional layer is the key building block in the CNN architecture that is used to extract useful features from the input image. A CNN consists of an input layer, hidden layers, and an output layer. The hidden layers are the several convolution layers that create a feature map. It does this using a feed forward mechanism where filters and kernels slide across the image which is seen as a matrix of pixels. As the kernel strides over the matrix and performs elementwise multiplication with the input pixels within its receptive field, the model gives weights to parts of the image so that it can extract dominant features. Once the kernel has passed through the entire image, it can produce a set of output feature maps. After the convolution, an activation function (ReLU used above) is applied elementwise to the output feature maps to introduce non-linearity into the model, allowing the network to model more complex relationships between the input and output data. The model can be optimized by selecting different filter values and kernel sizes. The filter size is commonly between 16 and 512, where the CNN model learns multiple features in parallel for a given input **[4]**.

After the convolution layer, a max pooling layer is present. The pooling operation is applied to the output feature maps to reduce their spatial dimensions in order to extract the most important features. This is also done to decrease the computational power required to process the data. Max pooling returns the maximum value from the portion of the image covered by the kernel, whereas average pooling returns the average of all values from the kernel over the image. The reason max pooling is better is because it acts as a noise suppressant, ridding the effects of surrounding distractions that may deviate the CNN model away from the more important features **[4]**.

The flatten stage of the CNN converts the output of the hidden layers into a 1-D vector which can then be fed into the fully connected layer. The flatten layer itself does not have any learnable parameters, but just simply rearranges the data. The vector passes to the fully connected layer where a matrix multiplication is carried out with the learned weights, biases, and applied activation function in order to provide a resulting output. The fully connected layer connects every neuron in one layer to every neuron in another layer (following a Multilayer Perceptron network format).

The model is improved following an original algorithm designed similarly to a LeNet structure consisting of 3 convolution layers and 3 max pooling layers. The improved model also contains padding and a dropout layer.

Padding is used to maintain the spatial size of the input data after each convolutional later. This helps in preventing information loss and allows the model to better detect features that may be present around the edges of an image. However, it is worth considering that this would cause an increase in model size, and computational cost so may not be ideal if computational efficiency is a limiting factor.

Dropout is a regularization technique used to help prevent overfitting by randomly removing a percentage of the

neuron outputs in the layer during training. By doing this, the network does not rely too heavily on any particular neuron, hence, making a more robust model.

This model is used as a base case with which optimizations are carried out in order to select the best optimizer and the best activation function. The model is firstly run over 4 optimizers (Adam, rmsprop, adadelta, sgd) after which the best optimizer is used in a model which is run over 4 activation functions (ReLU, linear, sigmoid, tanh). Each test runs the machine learning model over a range of optimizers and activation functions so that the best combination can be selected in order to maximize model performance.

This results in the final optimum model consisting of many convolutional layers (8 layers), with dropout and the best performing optimizer and activation functions. Finally, a pre-trained existing model is tested on the dataset to see its performance in comparison to the CNN network manually created.

IV. Experiments

A. *Basic CNN Model*

Firstly, a simple CNN model with a single convolutional layer is trained and tested. The model makes use of the ReLU activation function, Adam optimizer, and the categorical crossentropy loss function:

```
class Basic_Model(Model):
    def __init__(self):
        super(Basic_Model, self).__init__()
        # Define the conv layer (32 filter, 3 kernel size)
        self.conv1 = Conv2D(32,3, activation = 'relu')
        # Define max pool layer
        self.pool1 = MaxPool2D()
        # Convert 2D image to 1D vector
        self.flatten = Flatten()
        # Define fully connected layer
        self.d1 = Dense(128, activation = 'relu')
        # Dense further to have output size 10
        self.d2 = Dense(10)

    def call(self, x):
        # call conv layer 1
        x = self.conv1(x)
        # call max pool 1
        x = self.pool1(x)
        # call flatten
        x = self.flatten(x)
        # call dense 128
        x = self.d1(x)
        # call dense 10
        return self.d2(x)

# Create model instance
BasicModel = Basic_Model()
# Define loss function
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True)
# Define optimizer
optimizer = tf.keras.optimizers.Adam()
```

Figure 6 - Basic CNN Model

```
Epoch1,Train Loss: 69.5379409790039,Train Accuracy: 18.396873474121094,Test Loss: 2.2112133502960205,Test Accuracy: 19.898090325488828
Epoch2,Train Loss: 1.8418005704879761,Train Accuracy: 36.60365295410156,Test Loss: 2.145111560821533,Test Accuracy: 27.261146544410156
Epoch3,Train Loss: 1.3511350154876709,Train Accuracy: 54.926605224609375,Test Loss: 2.1988325119018555,Test Accuracy: 29.65604928393555
Epoch4,Train Loss: 1.0119801759719849,Train Accuracy: 67.3038330078125,Test Loss: 2.2678234577178955,Test Accuracy: 29.783439623047
Epoch5,Train Loss: 0.7844353914260864,Train Accuracy: 76.12207794189453,Test Loss: 2.4779889583587646,Test Accuracy: 30.242038602680664
```

Figure 7 - Loss and Accuracy using Basic CNN

There is no basis currently for the choice of 'Adam' and 'ReLU' other than the fact that they are the most commonly used in existing algorithms and are considered the best. This model is run using 5 epochs and 32 batch size. The purpose of this model is to get a working CNN model and understand what may be lacking and can be improved from the analysis. The results show that the accuracy increases every epoch, with a maximum testing accuracy of 30.02% after the fifth epoch. The model shows signs of overfitting as the test accuracy was too low considering the training accuracy was 76%. Additionally, the test loss began to increase after the fourth epoch, suggesting that the model was overfitted to the training data.

The chosen loss function (Sparse categorical crossentropy) is selected as it is used when classes are mutually exclusive. This loss function calculates the difference between the predicted probability distribution and the true probability distribution for the classes.

B. *Optimizer optimization*

To further develop the model, a simple 3 convolutional layer model is created where the filter size doubles every layer. The model is tested on a range of different optimizers to find the best performing algorithm as follows:

```
opt_perf = {}
for optim in ['adam', 'rmsprop', 'adadelta', 'sgd']:
    test_model = models.Sequential()
    test_model.add(layers.Conv2D(32,(3,3), activation = 'relu', padding = 'same'))
    test_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
    test_model.add(layers.Conv2D(64,(3,3), activation = 'relu', padding = 'same'))
    test_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
    test_model.add(layers.Conv2D(128,(3,3), activation = 'relu', padding = 'same'))
    test_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
    test_model.add(layers.Flatten())
    test_model.add(layers.Dense(128, activation='relu'))
    test_model.add(layers.Dropout(0.4))
    test_model.add(layers.Dense(10))

    test_model.compile(optimizer = optim,
                       loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                       metrics=['accuracy'])
    opt_perf[optim] = test_model.fit(train_dataset, epochs=10, validation_data=(test_dataset))
```

Figure 8 - Code used to select best optimizer ***[5]***

```
for optim in opt_perf:
    plt.plot(opt_perf[optim].history['val_accuracy'], label = f'{optim} val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```

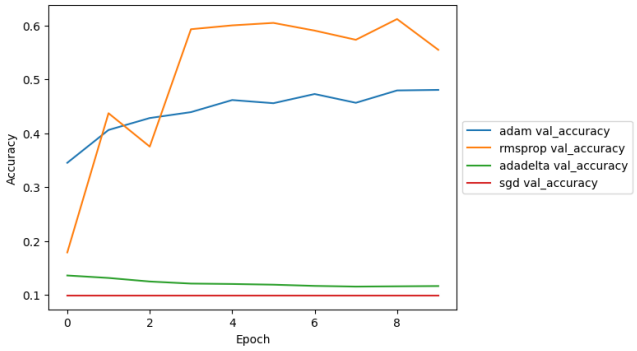

Figure 9 - Graph showing the test accuracy for different optimizers.

Each optimizer was run over 10 epochs to see which performed the best. While Adam is typically the best optimizer for CNN models, it seems that 'rmsprop' performed significantly better with a peak test accuracy of 61.17% in the 9th epoch, after which the model slightly overfits, resulting in a drop of accuracy. For this reason, it will be the default optimizer moving forward in this model. The basic theory behind 'rmsprop' is that it adapts the learning rate for each weight in the network based on the average of the squared gradients for that weight over the previous steps. This is done with 'decay rate' and 'moving average' hyperparameters in order to update the weight. This process helps to speed up rate of convergence during training by optimizing the learning rate based on the gradients for that weight.

C. *Activation function optimization*

The second optimization carried out was in selecting the optimal 'activation function' as follows:

```
activ_fctn = {}
for act_fnc in ['relu', 'linear', 'sigmoid', 'tanh']:
    activation_model = models.Sequential()
    activation_model.add(layers.Conv2D(32,(3,3), activation = act_fnc, padding = 'same'))
    activation_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
    activation_model.add(layers.Conv2D(64,(3,3), activation = act_fnc, padding = 'same'))
    activation_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
    activation_model.add(layers.Conv2D(128,(3,3), activation = act_fnc, padding = 'same'))
    activation_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
    activation_model.add(layers.Flatten())
    activation_model.add(layers.Dense(128, activation= act_fnc))
    activation_model.add(layers.Dropout(0.4))
    activation_model.add(layers.Dense(10))

    activation_model.compile(optimizer = 'rmsprop',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
    activ_fctn[act_fnc] = activation_model.fit(train_dataset, epochs=10, validation_data=(test_dataset))
```

Figure 10 - Code used to select best activation function.

```
for act_fnc in activ_fctn:
    plt.plot(activ_fctn[act_fnc].history['val_accuracy'], label = f'{act_fnc} val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```

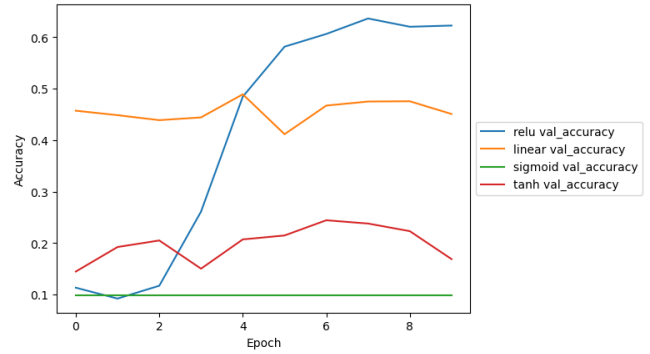

Figure 11 - Graph showing the test accuracy for a range of activation functions.

It is evident that the best performing activation function is 'ReLU' as it has the highest accuracy from 5 epochs and above with its peak accuracy being 63.62% on the 8th epoch after which the model began to overfit slightly. Sigmoid and tanh performed very poorly whereas the 'linear' model provided a decent stable accuracy throughout every epoch.

D. *My Best Model*

A final CNN model can be created based on all the previously collected data on best optimizers (rmsprop), activation functions (ReLU), filter sizes, dropout rates etc.

```
rmsprop_model = models.Sequential()
rmsprop_model.add(layers.Conv2D(16,(3,3), activation = 'linear', padding = 'same', input_shape = (128,128,3)))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(16,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(32,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(32,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.Dropout(0.2))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(64,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(64,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(128,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Conv2D(128,(3,3), activation = 'relu', padding = 'same'))
rmsprop_model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
rmsprop_model.add(layers.Flatten())
rmsprop_model.add(layers.Dense(128, activation='relu'))
rmsprop_model.add(layers.Dropout(0.2))
rmsprop_model.add(layers.Dense(10))

rmsprop_model.compile(optimizer='rmsprop',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history_rmsprop = rmsprop_model.fit(train_dataset, epochs=10,
                    validation_data=(test_dataset))
```

Figure 12 - CNN Code for my best model

This model makes use of 8 convolutional layers where every 2 layers are identical. For example, a filter size of 16 is used twice, then 32 twice, and so on up until 128 filters. The model also has padding, 'ReLU' activation and two dropout layers.

Repeating a filter size in a multi-convolutional layer model can provide several advantages to a CNN model. Firstly, it increases non-linearity as each layer applied a non-linear transformation to the data. Secondly, each layer extracts different features from the input data. By using the same filter in multiple layers, similar features can be extracted at different scales, allowing the model to capture more complex patterns within the data. Thirdly, the model has a reduced parameter space due to more layers, leading to faster training times and better generalization performance. Fourthly, it introduces a higher degree of spatial invariance which allows the CNN to recognize unique features regardless of its location.

Note that the first convoluted layer also has a linear activation function, whereas the remaining are ReLU. This is because a linear function for the first layer allows the model to learn a range of filter values to make the model more stable during training. After the first layer, ReLU is used in the middle layers since it introduces non-linearity and allows the network to learn complex patterns. Finally, a SoftMax would typically be used in the final dense layer to convert the outputs of the model into probability values in order to predict class. However, since the loss function has 'from_logits = True' means that this is not required. When logits=True, the model outputs raw, unnormalized predictions for each class. Both methods are suitable, and this does not greatly affect the model.

The results of running this 'optimum' model can be seen in the following figures:

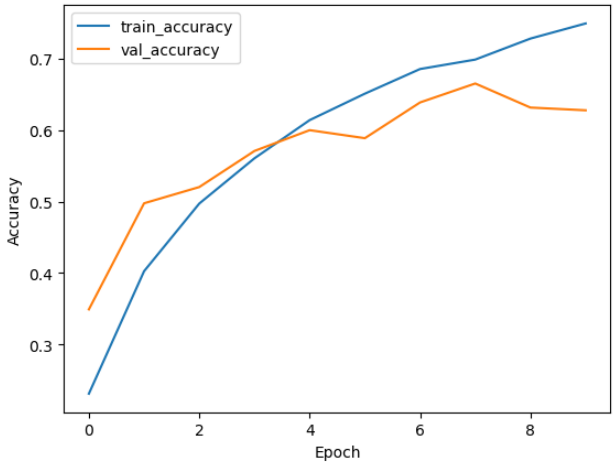

Figure 13 - Graph showing training and testing accuracy.

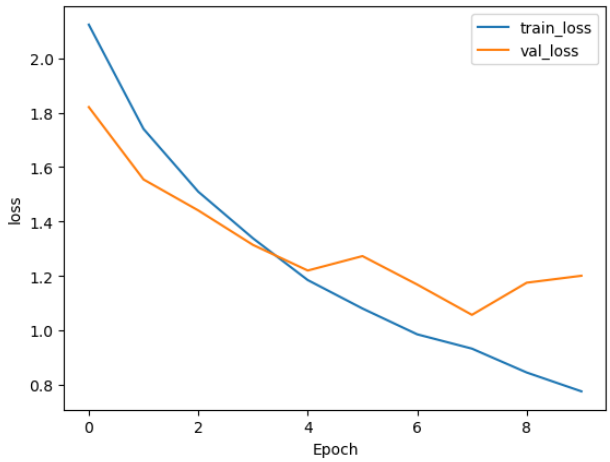

Figure 14 - Graph showing testing and training loss.

The testing accuracy shows that this is the best performing model so far with a maximum accuracy of 64.05% in the 7th epoch. The model shows signs of overfitting however, as the val_loss staggers and increases despite the training loss continuously going down. To combat this, perhaps a greater dropout percentage could be used, or further regularization could be implemented.

It is also worth mentioning that despite 'Adam' being a worse optimizer according to Figure 9, the final CNN model was also run using the 'Adam' optimizer to see its performance in comparison to 'rmsprop':

```
Epoch 1/10
296/296 [==============================] - 43s 141ms/step - loss: 2.0436 - accuracy: 0.2766 - val_loss: 1.8164 - val_accuracy:
0.3654
Epoch 2/10
296/296 [==============================] - 41s 138ms/step - loss: 1.7118 - accuracy: 0.4176 - val_loss: 1.6634 - val_accuracy:
0.4410
Epoch 3/10
296/296 [==============================] - 41s 139ms/step - loss: 1.4827 - accuracy: 0.4948 - val_loss: 1.4290 - val_accuracy:
0.5225
Epoch 4/10
296/296 [==============================] - 42s 141ms/step - loss: 1.3116 - accuracy: 0.5573 - val_loss: 1.3022 - val_accuracy:
0.5710
Epoch 5/10
296/296 [==============================] - 44s 147ms/step - loss: 1.2016 - accuracy: 0.6024 - val_loss: 1.1956 - val_accuracy:
0.6127
Epoch 6/10
296/296 [==============================] - 44s 148ms/step - loss: 1.0907 - accuracy: 0.6460 - val_loss: 1.2305 - val_accuracy:
0.6099
Epoch 7/10
296/296 [==============================] - 45s 151ms/step - loss: 0.9763 - accuracy: 0.6890 - val_loss: 1.1649 - val_accuracy:
0.6283
Epoch 8/10
296/296 [==============================] - 45s 152ms/step - loss: 0.9144 - accuracy: 0.7039 - val_loss: 1.1624 - val_accuracy:
0.6362
Epoch 9/10
296/296 [==============================] - 44s 147ms/step - loss: 0.8288 - accuracy: 0.7412 - val_loss: 1.1091 - val_accuracy:
0.6420
Epoch 10/10
296/296 [==============================] - 44s 147ms/step - loss: 0.7936 - accuracy: 0.7415 - val_loss: 1.0596 - val_accuracy:
0.6780
```

Figure 15 - Loss and Accuracy using Adam optimizer.

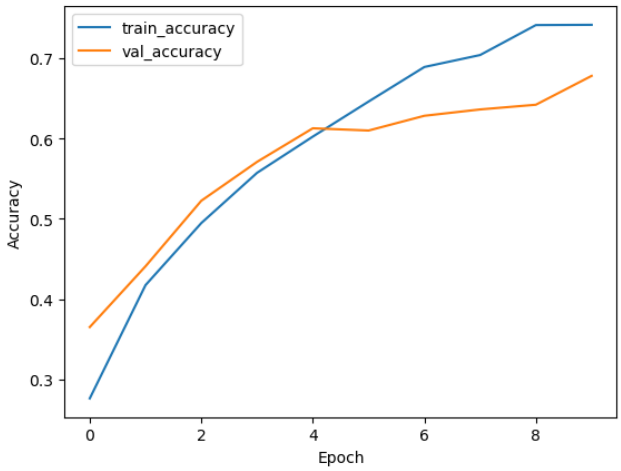

Figure 16 - Graph showing accuracy using Adam Optimizer

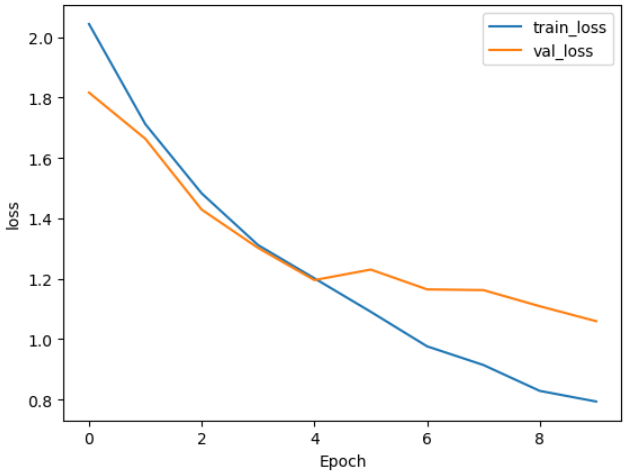

Figure 17 - Graph showing loss using Adam Optimizer

As seen in Figures 15,16,17 the highest test accuracy (67.8% in Epoch 10) is present when using Adam as the optimizer, despite Figure 9 suggesting that 'rmsprop' is better. The model has no overfitting over a 10-epoch range and consistently improves, making it the best model yet.

E. VGG16 – Pretrained model

VGG16 is a pre-trained CNN that has been trained on an extremely large dataset (ImageNet) consisting of millions of images. It is a very deep network with 16 layers, allowing it to learn and recognize highly complex features and classifications from images. In comparison, the manual CNN model I created has been trained on a small image dataset of approximately 9000 images, explaining why the performance of my model is much lower. VGG16 can be applied to the image dataset as follows:

```
# Set the number of epochs and batch size for training
epochs = 5
batch_size = 32

# Preprocess our input images by rescaling pixel values
train_datagen = ImageDataGenerator(rescale=1./255)

# Load the VGG16 model, pre-trained on ImageNet dataset
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(img_width, img_height, 3))

# Freeze the weights of the pre-trained layers in the base model
for layer in base_model.layers:
    layer.trainable = False

# Create a new model by adding our own layers on top of the pre-trained layers
model = Sequential()
model.add(base_model)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile the model with categorical cross-entropy loss and Adam optimizer
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Generate training and validation batches from the preprocessed images
train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode='categorical')

test_generator = train_datagen.flow_from_directory(
        test_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode='categorical')

# Train the model on the training data and validate on the testing data
model.fit_generator(
        train_generator,
        steps_per_epoch=nb_train_samples // batch_size,
        epochs=epochs,
        validation_data=test_generator,
        validation_steps=nb_test_samples // batch_size)
```

Figure 18 - Code used to apply the VG166 architecture to the given image set.

Running this model for only 5 epochs gives the following results:

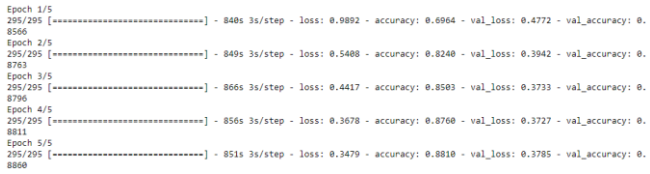

Figure 19 - Training/Testing Loss/Accuracy for VGG16

It is clear from the first epoch that VGG16 is a completely superior CNN algorithm in every way as it immediately gives an accuracy of 85.66%. There are also no signs of overfitting as the model continues to increase in accuracy (even if only slightly) as it reaches 88.60% accuracy after the 5th epoch. The one possible setback of using VGG16 for a task such as this is that since the model is so deep and diverse, it takes a long time to process if the computational efficiency is a limiting factor, which it likely is if you are using a standard computer or laptop. Ultimately, VGG16 and other pre-trained models like AlexNet or GoogLeNet will outperform most models due to the depth of their algorithms and due to the sheer mass of data that they have been trained on.
[6][7][8]

V. Reflection

In conclusion, CNNs are a vital piece of technology as it is one of the best methods of training a model to classify images into different classes. Making use of an image set of 9000 files in the training data, a simple CNN model was created in an attempt to class testing data into their respective classes. This basic model performed quite poorly with only a 30% accuracy over 5 epochs. Adding more convolutional layers and trying out a range of filter sizes allowed the accuracy of the model to increase. Additionally, it was determined that 'rmsprop' was the best performing optimizer and 'ReLU' was the best performing activation function. Therefore, both of these were used in the final 'optimal' model with a test accuracy of approximately 65%. Despite being a relatively good model, it cannot perform as highly as pre-existing models such as VGG16 which has trained its 16-layer network on millions of images.

Despite a fairly optimized CNN model being created, several problems were encountered preventing the execution of certain techniques. Therefore, there is still room for several optimization techniques to be carried out, as well as discussing their limitations/complications:

Firstly, since the image set used to train the data was relatively small (9000 images), data augmentation could be carried out to randomly rotate images, change contrast, and randomly zoom or flip images etc. This can create an augmented dataset, making a training sample size well over 9000 images in order to generalize the model and reduce any overfitting as it is trained on a larger training set. While this is good, operating on a larger dataset will increase the time taken to carry out the test.

Secondly, batch normalization and regularization are two additional methods that can help to reduce overfitting. While both methods are effective, it was found that applying them on top of the dropout layer already present resulted in a lower testing accuracy. Therefore, while attempting to prevent overfitting, all these precautionary measures combined can actually cause underfitting of data.

Thirdly, automated hyperparameter tuning using methods like gridsearch are a great way to optimize number of epochs, batch size, and learning rate. However, due to the computational limitations, carrying out such tests were not possible. Ideally, optimizing and testing a range of learning rates and batch size would have been optimal in creating a more accurate CNN.

Additionally, performance could be increased further by making use of transfer learning, making use of a pre-trained model, and fine-tuning it to the dataset present. This method helps in improving accuracy and saving time.

Making use of all these improvements, on top of all the optimizations already having been carried out could result in a much more optimized CNN model with a greater accuracy and lower overfitting.

VI. References

[1] *Real-world applications of Convolutional Neural Networks*, *Data Analytics*. Available at: https://vitalflux.com/real-world-applications-of-convolutional-neural-networks/ (Accessed: March 16, 2023).

[2] Kumar, B. (2021) *Convolutional Neural Networks: A brief history of their evolution*, *Medium*. AppyHigh Blog. Available at: https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597#:~:text=The%20name%20convolutional%20neural%20networks,the%20handwritten%20digit%20recognition%20task.&text=The%20overall%20architecture%20was%20%5BCONV,with%20a%20strike%20of%201. (Accessed: March 16, 2023).

[3] Ramalingam, A. (2021) *How to pick the optimal image size for training convolution neural network?*, *Medium*. Analytics Vidhya. Available at: https://medium.com/analytics-vidhya/how-to-pick-the-optimal-image-size-for-training-convolution-neural-network-65702b880f05#:~:text=Downscaling%3A%20Bigger%20images%20will%20be,be%20present%20is%20significantly%20reduced. (Accessed: March 16, 2023).

[4] Saha, S. (2022) *A comprehensive guide to Convolutional Neural Networks-the eli5 way*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (Accessed: March 16, 2023).

[5] *Convolutional Neural Network (CNN) : Tensorflow Core* (no date) *TensorFlow*. Available at: https://www.tensorflow.org/tutorials/images/cnn (Accessed: March 16, 2023).

[6] Thakur, R. (2020) *Step by step VGG16 implementation in Keras for beginners*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c (Accessed: March 16, 2023).

[7] Brownlee, J. (2019) *How to use the pre-trained VGG model to classify objects in photographs*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/use-pre-trained-vgg-model-classify-objects-photographs/ (Accessed: March 16, 2023).

[8] Author: James McDermott Data Science Consultant and James McDermott Data Science ConsultantJames is a data science consultant and technical writer. He has spent four years working on data-driven projects and delivering machine learning solutions in the research industry. (no date) *Hands-on transfer learning with keras and the VGG16 model*, *Learn Data Science - Tutorials, Books, Courses, and More*. Available at: https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/ (Accessed: March 16, 2023