SOFTWARE DEVELOPMENT FOR DATA SCIENCE

Created by Spyridon Kalogeropoulos - S1632672

This note book was created as part of a coursework for Glasgow Caledonian University.

Introduction

This coursework will provide a detail implementation of AlexNet and VGG16 for the CIFAR-10 dataset. The architecture of the models will be based on the original papers provided by the authors.

Some minor changes have been implemented for each model but without alterring the initial architecture of either model. The models will be assesed based on the simplicity and easiness to implement as well as for they accuracy and time required to train.

By the end of this report, the reader will have a basic idea of what happens inside a CNN and how to replicate one. Furthermore, we will discuss problems encountered along the way and possible solutions to solve these problems.

Finally, the report shows how each algorithm performed and what can be further improved in future implementations.

Dataset selection

Discussion upon the choice of dataset along with other considerations.

CIFAR-10

The dataset the algorithms will be making use is that of CIFAR-10.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

The dataset was chosen among others for its image complexity and easy implementation. To properly test the chosen algorithms it was essential to choose a dataset that would have some variety but also was not extremelly heavy.

In comparison to other datasets such as mnist or fashion mnist, CIFAR 10 is smaller in size, however contains RGB images that make the classification harder and more complex, traits essential for proper algorithmic testing.

CIFAR 10 is lighter in size than ImageNet which has a size of 150GB or visual QA which weights 25 GB, infact cifar 10 is sized at 170MB. Which makes this dataset both as small as MNIST or FASHION MNIST but also as complex as those heavier datasets.

As seen in the 'Training and Testing' section, CIFAR 10 was very easy to implement, with the minor setback that the images have a size of 32x32 and the chosen algorithms require larger size images. Meaning that we would need to resize the images in order to implement them.

However, TensorFlow has an inbuild method that allows us to resize images with ease.

```
# Resize images from 32x32 to 224x224
  image = tf.image.resize(image, (224,224))
```

After setting class names for each image type in the dataset, we create a validation and a test dataset by spliting the initial dataset.

Finally, we set the image pre-processing variables so that they can be called by the model at a later stage.

```
# Perform operations on each dataset.

train_dataset = (train_dataset
                 .map(process_images,num_parallel_calls=2)
                 .shuffle(buffer_size = 10000)
                 .batch(batch_size=64, drop_remainder=True)
                 .prefetch(tf.data.experimental.AUTOTUNE)
                 )
...
```

CIFAR 10 is a good starting dataset as it is more 'difficult' than simple greyscale datasets with high image number, MNIST, FASHION MNIST, etc. Which when impleneted with LeNet or other models achieve high accuracy without much effort.

Algorithm choice

Covers the choice of algorithms to be implemented as well as other potential choices.

As the dataset has now been chosen based on variety and difficuty, we assessed the available algorithms in the same manner.

Certainly, this coursework could have implemented solutions like LeNet and AlexNet on simple datasets, but these datasets and models are 'solved'. Meaning that there are so many solutions available online that our

solution would offer little insight.

Therefore, I decided to skip LeNet which was and still is a fantastic model and dwelve deeper with AlexNet and GoogLeNet. The initial choice was based on the parameters each model involved.

AlexNet, when properly implemented with an accurate image size (227x227), has over 60 million paramenters and achieved a top 5 error rate of 26.3% on ILSVRC 2012 challenge.

On the other hand, GoogLeNet managed to score an astonishing top 5 error rate of 6.7% with only 4 million parameters (ILSVRC 2014 challenge). Due to implementation difficulties, GoogLeNet was not implemented, but instead we chose VGG-16.

VGG-16 has double the parameters of AlexNet (140 million) and scored second behing GoogLeNet in the same challenge, with only a small error rate higher (6.8%). VGG-16 offers a very similar approach same as that of AlexNet which makes it ideal for a starting project.

VGG 16, has more less popular variations, such as VGG-11, VGG-13 and VGG-19. However this is the most well known version due to its simplicity and accuracy.

The most common impementation issue was that of Image size. Given the chosen dataset (CIFAR 10) and the size of the images contained (32x32), neither of the models were compatible with the default architecture. As the coursework focuses mostly on the models rather the dataset, a simple solution but not very smart was to resize the images using TensorFlow. This solved the image size problem but it did affect the dataset accuracy.

Another solution to this problem would have been to change the model's architecture (filters, kernel_size, strides, input_size) so that it can read smaller images. I believe this is the preffered solution for a problem of this nature when we care about accuracy rather than architecual integrity. As this was not the case, we implemented the first approach.

Another issue was dataset overfit. Both the datasets, when trained long enough would 'memorise' the dataset and reach an accuracy of 99%. This was solved by implementing data augmentation.

```
# Augmentate images by fliping and rotating
  data_augmentation = tf.keras.Sequential([
    keras.layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
    keras.layers.experimental.preprocessing.RandomRotation(0.2),
])
```

This allowed us to train the dataset for a longer period of time without overfitting the dataset.

Detailed analysis of the implementation of each algorithm can be seen in the section bellow, where each step is thoroughly analysed.

Training and testing

Showcase the implementation process for each algorithm.

Pre-processing the dataset

Covers in detail the necessary actions taken to pre-process the CIFAR-10 dataset before training.

```
#Import libraries
import tensorflow as tf
import numpy as np
from tensorflow import keras
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau
import matplotlib.pyplot as plt
import os
import time
%load_ext tensorboard
```

```
#Check if GPU is available and make use
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
else:
  tf.device('/device:GPU:0')
  print('Found GPU at: {}'.format(device_name))
```

```
    Found GPU at: /device:GPU:0
```

```
#Download the training set and split into train & test
(train_images, train_labels), (test_images, test_labels) = keras.datasets.cifar10.load_data()
```

```
#Set the class names for the CIFAR-10 dataset
CLASS_NAMES= ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
```

```
#Create a validation set to assess the performance of the dataset at various iterations.
validation_images, validation_labels = train_images[:5000], train_labels[:5000]
train_images, train_labels = train_images[5000:], train_labels[5000:]
```

```
#Create TensorFlow Dataset representation of Train, Test, Validation.

train_ds = tf.data.Dataset.from_tensor_slices((train_images, train_labels))

test_ds = tf.data.Dataset.from_tensor_slices((test_images, test_labels))

validation_ds = tf.data.Dataset.from_tensor_slices((validation_images, validation_labels))
```

```
#Create a plot of images with coresponding labels that will be used for training purposes with
plt.figure(figsize=(20,20))
for i, (image, label) in enumerate(train_ds.take(6)):
    ax = plt.subplot(6,6,i+1)
    plt.imshow(image)
    plt.title(CLASS_NAMES[label.numpy()[0]])
```

```
    plt.axis('off')
```


```
# Normalise images
def process_images(image, label):
    # Normalize images to have a mean of 0 and standard deviation of 1
    image = tf.image.per_image_standardization(image)
    # Resize images from 32x32 to 224x224
    image = tf.image.resize(image, (224,224))
    return image, label
```

```
# Print the size of each dataset

print("Training data size:", len(train_ds))
print("Test data size:", len(test_ds))
print("Validation data size:", len(validation_ds))
```

```
Training data size: 45000
Test data size: 10000
Validation data size: 5000
```

To improve accuracy with the cost of performance the program takes each dataset and applies the following.

- map each image and resize to be faithfull to AlexNet architecture, also make 2 parallel calls for faster processing.
- shuffle the images to get a wider variety and achieve better accuracy.
- seperate the dataset into smaller baches to preprocess
- prefetch the dataset into the GPU to utilize the GPU at all times

Resizing images is not great practice but given the size of the images in CIFAR-10 (32x32), we do so to remain faithfull to AlexNet and VGG architecture.

```
# Perform operations on each dataset.

processed_train_ds = (train_ds
                  .map(process_images,num_parallel_calls=2)
                  .shuffle(buffer_size = 10000)
                  .batch(batch_size=128, drop_remainder=True)
                  .prefetch(tf.data.experimental.AUTOTUNE)
                  )

processed_test_ds = (test_ds
```

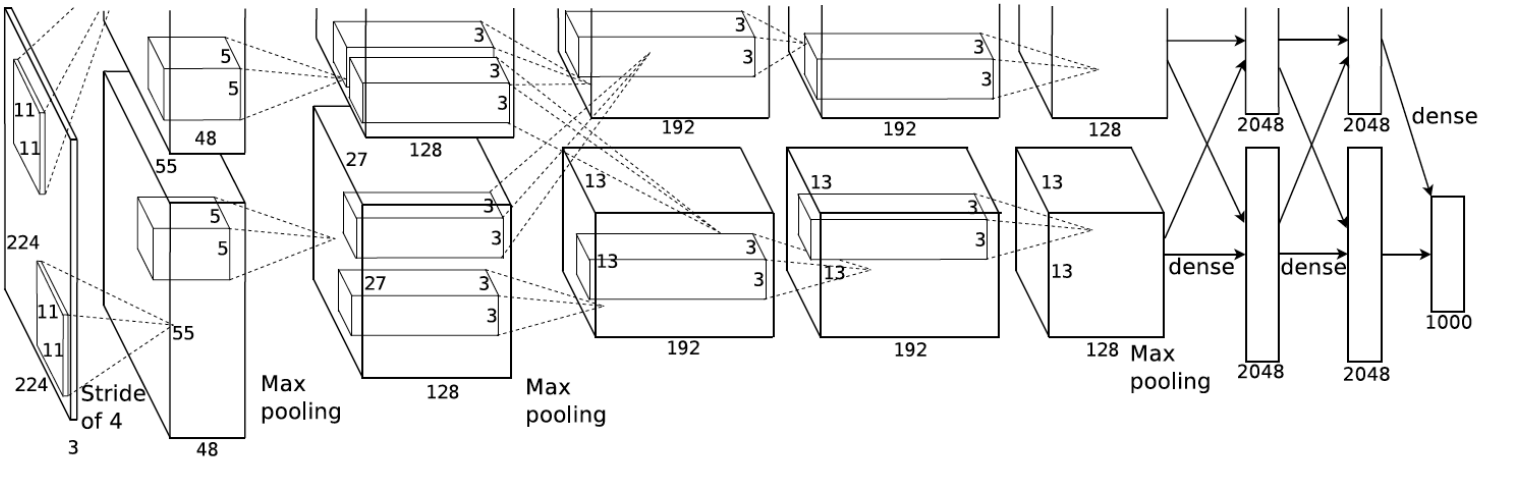
```
                    .map(process_images,num_parallel_calls=2)
                    .shuffle(buffer_size = 10000)
                    .batch(batch_size=128, drop_remainder=True)
                    .prefetch(tf.data.experimental.AUTOTUNE)
                    )

processed_validation_ds = (validation_ds
                    .map(process_images, num_parallel_calls=2)
                    .shuffle(buffer_size = 5000)
                    .batch(batch_size=128, drop_remainder=True)
                    .prefetch(tf.data.experimental.AUTOTUNE))
```

Experimenting with the parameters above has lead to the following for both models.

1. Smaller batch sizes lead to more accurate results faster, but increase time per epoch. Some balance is essential.
2. Higher number of shuffles gives better results but requires a lot of ram. When buffering > 30.000 images, the system crashes.
3. prefetching as well as parallel calls reduce the time per epoch by ~5 seconds (Valuable when performing large numbers of epochs)

AlexNet [2012]

The neural network developed by Krizhevsky, Sutskever, and Hinton in 2012 was the coming out party for CNNs in the computer vision community. AlexNet managed to out-perform LeNet and achieved a Top-5 error rate of 15.3%. The next best result was far behind (26.2%). AlexNet is quite heavy with around 60 milion parameters that make each epoch take approximately 50 seconds.



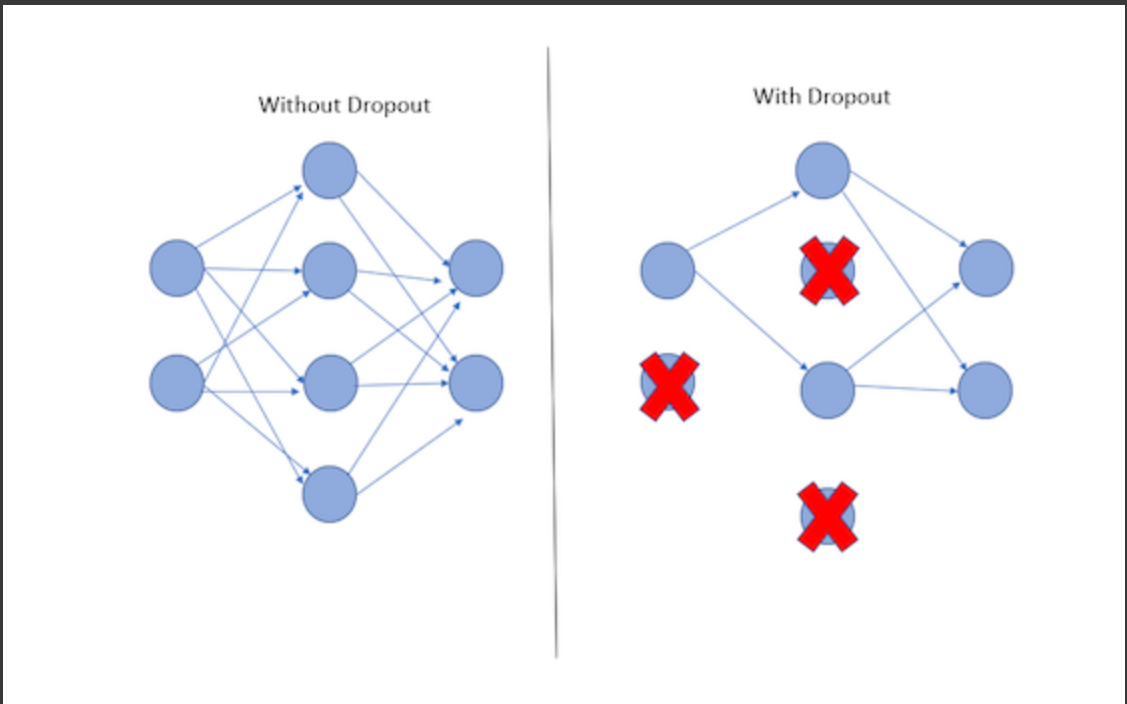
AlexNet was specificaly chosen due to its simplicity as a baseline CNN, but also for its ability to implement novel techniques such as,

- Data Augmentation
- GPU Training

- Dropouts

Some of these techniques will be used in our implementation.

Due to resource limitations, we will be implementing data augmentation as part of the model, this way we can utilise the GPU to the fullest extend.



The architecture of AlexNet is quite simple in comarison to modern CNNs yet more complex than its predecesor LeNet. AlexNet consists of five convolutional layers and three fully connected layers. However, this is not what makes AlexNet special;

- Multiple GPUs: AlexNet was the first CNN to allow multiple GPUs
- ReLU Nonlinearity: Advanced technique that allows AlexNet to reach a 25% error on CIFAR-10 dataset almost 6 times faster than tanh, its predecessor.
- Overlapping Pooling: was first introduced by the authors of AlexNet who saw a reduction in error about 0.5% and also noticed that the model was harder to overfit. Overlapping is used to reduce the height and width of tensors while maintaining the depth.

```
# Augmentate images by fliping and rotating
data_augmentation = tf.keras.Sequential([
  keras.layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
  keras.layers.experimental.preprocessing.RandomRotation(0.2),
])
```

```
#Declare the model and run each layer in sequential order
alexnet = keras.models.Sequential([
    data_augmentation,
    #1st Iteration with convolution and max pooling
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', inpu
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    #2nd Iteration with convolution and max pooling
```

```
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', paddi
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    #3rd Iteration with convolution
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', paddi
    keras.layers.BatchNormalization(),
    #4th Iteration with convolution
    keras.layers.Conv2D(filters=384, kernel_size=(1,1), strides=(1,1), activation='relu', paddi
    keras.layers.BatchNormalization(),
    #5th Iteration with convolution and max pooling
    keras.layers.Conv2D(filters=256, kernel_size=(1,1), strides=(1,1), activation='relu', paddi
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    #1st Hidden layer
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    #2nd Hidden layer
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    #3rd Hidden layer
    keras.layers.Dense(1000, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    # Model output
    keras.layers.Dense(10, activation='softmax')
])
```

It is important to mention that looking at the diagram above the implemented architecture looks different, the reason behind this is that AlexNet was first implemented using two GPUs while we are only using one.

Furthermore, another minor addition was the implementation of Batch normalisation, which is a technique that mitigates the effect of unstable gradients within a neural network through the introduction of an additional layer that performs operations on the inputs from the previous layer.
This leads to more accurate results is shorter time span.

To optimise the model to the fullest extent, the system makes use of tensorboard. Tensorboard is usefull for seeing graphs and charts about the performance of the model. However, it also provides feedback on how input-bound the system is through TensorFlow Profiler. Input bound means that the system is waiting for the data to load prior to training them. Meaning that the GPU is not processing images.
Before implementing Profiler, the system was HIGHLY input bound (>20%). Through various suggestions such as pre-processing or cache data, the system now is NOT input-bound (<5%)

```
#create directory to store profiler data for each run
root_logdir = os.path.join(os.curdir, "logs//alexnet")

def get_run_logdir():
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
```

```
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir, profile_batch = '500,520')
```

The model was tested making use of different optimisation values.

1. **Adam** (20 epochs, data augmentation, 128 batch size, lr=0.001)

    - 56s per epoch
    - Accuracy ~ 0.51, validation accuracy ~ 0.53

2. **SGD** (20 epochs, data augmentation, 128 batch size, lr=0.001)

    - 50s per epoch
    - Accuracy ~ 0.47, validation accuracy ~ 0.46

Adam performes significantly faster on the AlexNet model, therefore we will further test the system by making use of this.
Further testing showed that Data Augmentation (DA) has no effect on the accuracy nor performance of neither models at such low epochs. However, DA is imperative to reduce model overfiting.

3. Adam (20 epochs, data augmentation, **256 batch size**, lr=0.001)

    - 52s per epoch.
    - Accuracy ~ 0.35, validation accuracy ~ 0.42

4. Adam (20 epochs, data augmentation, **64 batch size**, lr=0.001)

    - 50s per epoch
    - Accuracy ~ 0.49, validation accuracy ~ 0.50

5. Adam (20 epochs, data augmentation, 128 batch size, **lr=0.002**)

    - 53s per epoch
    - Accuracy ~ 0.46, validation accuracy ~ 0.50

6. Adam (20 epochs, data augmentation, 128 batch size, **lr=0.0008**)

    - 51s per epoch
    - Accuracy ~ 0.44, validation accuracy ~ 0.45

From the above test we deduce that the optimal settings for the AlexNet model are those described on the 1st run. To conclude we will make some high epoch measurements to understand the model's efficiency.

7. Adam (50 epochs, data augmentation, 64 batch size, lr=0.001)

    - 72s per epoch
    - Accuracy ~ 0.64, validation accuracy ~ 0.65

8. Adam (50 epochs, NO Data Augmentation, 64 batch size lr=0.001)

    - 72s per epoch
    - Accuracy ~ 0.75, validation accuracy ~ 0.70

The times shown are different from those recorded yesterday due to the fact that Google Collab has a threshold to the resources available to each user.
What's interesting is the accuracy between the two models. This shows the importance of data augmentation, as '8' is overfitting the dataset, meaning that it memorises rather than learning.

9. Adam (100 epochs, data augmentation, 64 batch size, lr=0.001)

    - 72s per epoch
    - Accuracy ~ 0.65, validation accuracy ~ 0.67

10. Adam (100 epochs, NO Data Augmentation, 64 batch size lr=0.001)

    - 72s per epoch
    - Accuracy ~ 0.76, validation accuracy ~ 0.71

11. Adam (50 epochs, data augmentation, 128 batch size, lr=0.001)

    - 72s per epoch
    - Accuracy ~ 0.74, validation accuracy ~ 0.73

On the last recorder run (11) the model scores a satisfactory accuracy reaching a 75% accuracy and 72% validation accuracy. This indicates that the model has some overfitting issues despite having implemented data augmentation.
Some attempts were made to fix the overfitting issue, such as implement dropouts after some convolutional layers, but the score accuracy was low.
Overall, AlexNet is a fast and lightweight model, as each epoch requires around 60s and can reach a validation accuracy of 0.73 at 50 epochs. Meaning that on the CIFAR-10 dataset can achieve satisfactory validation in almost an hour.

```
#Learning Rate Annealer
lrr= ReduceLROnPlateau(   monitor='val_accuracy',   factor=.01,   patience=3,  min_lr=1e-5)
```

```
# Compile the model and implement optimiser
alexnet.compile(loss='sparse_categorical_crossentropy', optimizer=tf.optimizers.Adam(), metrics
```

```
#Start training the model
alexnet_history = alexnet.fit(
          processed_train_ds,
          epochs=10,
          validation_data=processed_validation_ds,
          validation_freq=1,
          callbacks=[lrr,tensorboard_cb]
```

```
)
```

```
Epoch 1/10
351/351 [==============================] - 58s 152ms/step - loss: 0.7298 - accuracy: 0.74
Epoch 2/10
351/351 [==============================] - 60s 156ms/step - loss: 0.7310 - accuracy: 0.74
Epoch 3/10
351/351 [==============================] - 58s 153ms/step - loss: 0.7271 - accuracy: 0.74
Epoch 4/10
351/351 [==============================] - 59s 153ms/step - loss: 0.7242 - accuracy: 0.74
Epoch 5/10
351/351 [==============================] - 59s 154ms/step - loss: 0.7259 - accuracy: 0.74
Epoch 6/10
351/351 [==============================] - 59s 153ms/step - loss: 0.7155 - accuracy: 0.74
Epoch 7/10
351/351 [==============================] - 59s 154ms/step - loss: 0.7269 - accuracy: 0.74
Epoch 8/10
351/351 [==============================] - 59s 153ms/step - loss: 0.7150 - accuracy: 0.74
Epoch 9/10
351/351 [==============================] - 59s 154ms/step - loss: 0.7154 - accuracy: 0.75
Epoch 10/10
351/351 [==============================] - 59s 154ms/step - loss: 0.7145 - accuracy: 0.75
```

```
#Evaluate the model on the test dataset
alexnet.evaluate(processed_test_ds)
```

```
78/78 [==============================] - 9s 55ms/step - loss: 0.8063 - accuracy: 0.7247
[0.8062922358512878, 0.7246594429016113]
```

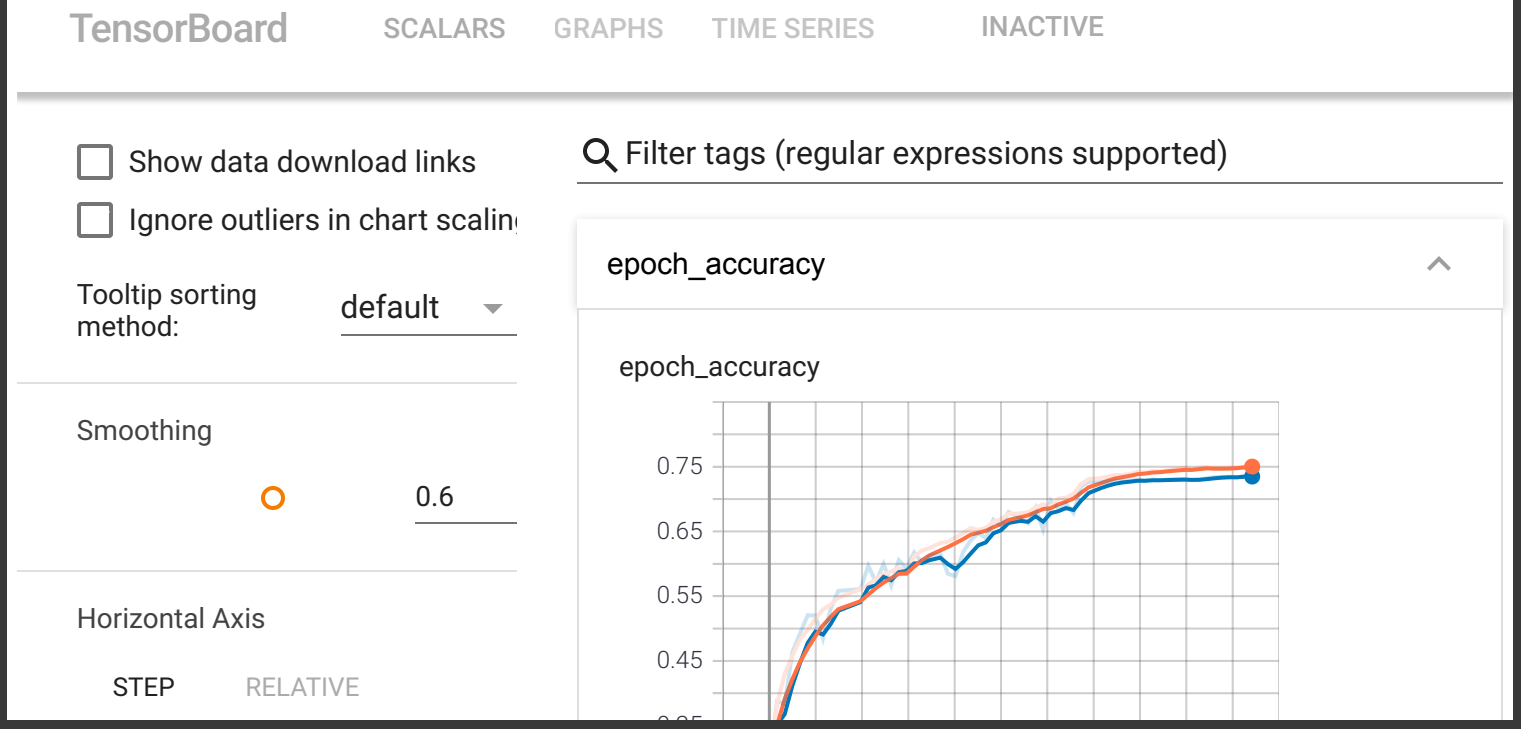
```
#Show a summary of the models layers
alexnet.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential (Sequential)      (128, 224, 224, 3)        0
_________________________________________________________________
conv2d (Conv2D)              (128, 54, 54, 96)         34944
_________________________________________________________________
batch_normalization (BatchNo (128, 54, 54, 96)         384
_________________________________________________________________
max_pooling2d (MaxPooling2D) (128, 26, 26, 96)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (128, 26, 26, 256)        614656
_________________________________________________________________
batch_normalization_1 (Batch (128, 26, 26, 256)        1024
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (128, 12, 12, 256)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (128, 12, 12, 384)        885120
_________________________________________________________________
batch_normalization_2 (Batch (128, 12, 12, 384)        1536
_________________________________________________________________
conv2d_3 (Conv2D)            (128, 12, 12, 384)        147840
_________________________________________________________________
batch_normalization_3 (Batch (128, 12, 12, 384)        1536
_________________________________________________________________
conv2d_4 (Conv2D)            (128, 12, 12, 256)        98560
```

```
_________________________________________________________________
batch_normalization_4 (Batch (128, 12, 12, 256)        1024
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (128, 5, 5, 256)          0
_________________________________________________________________
flatten (Flatten)            (128, 6400)               0
_________________________________________________________________
dense (Dense)                (128, 4096)               26218496
_________________________________________________________________
batch_normalization_5 (Batch (128, 4096)               16384
_________________________________________________________________
dropout (Dropout)            (128, 4096)               0
_________________________________________________________________
dense_1 (Dense)              (128, 4096)               16781312
_________________________________________________________________
batch_normalization_6 (Batch (128, 4096)               16384
_________________________________________________________________
dropout_1 (Dropout)          (128, 4096)               0
_________________________________________________________________
dense_2 (Dense)              (128, 1000)               4097000
_________________________________________________________________
batch_normalization_7 (Batch (128, 1000)               4000
_________________________________________________________________
dropout_2 (Dropout)          (128, 1000)               0
_________________________________________________________________
dense_3 (Dense)              (128, 10)                 10010
=================================================================
Total params: 48,930,210
Trainable params: 48,909,074
Non-trainable params: 21,136
_________________________________________________________________
```

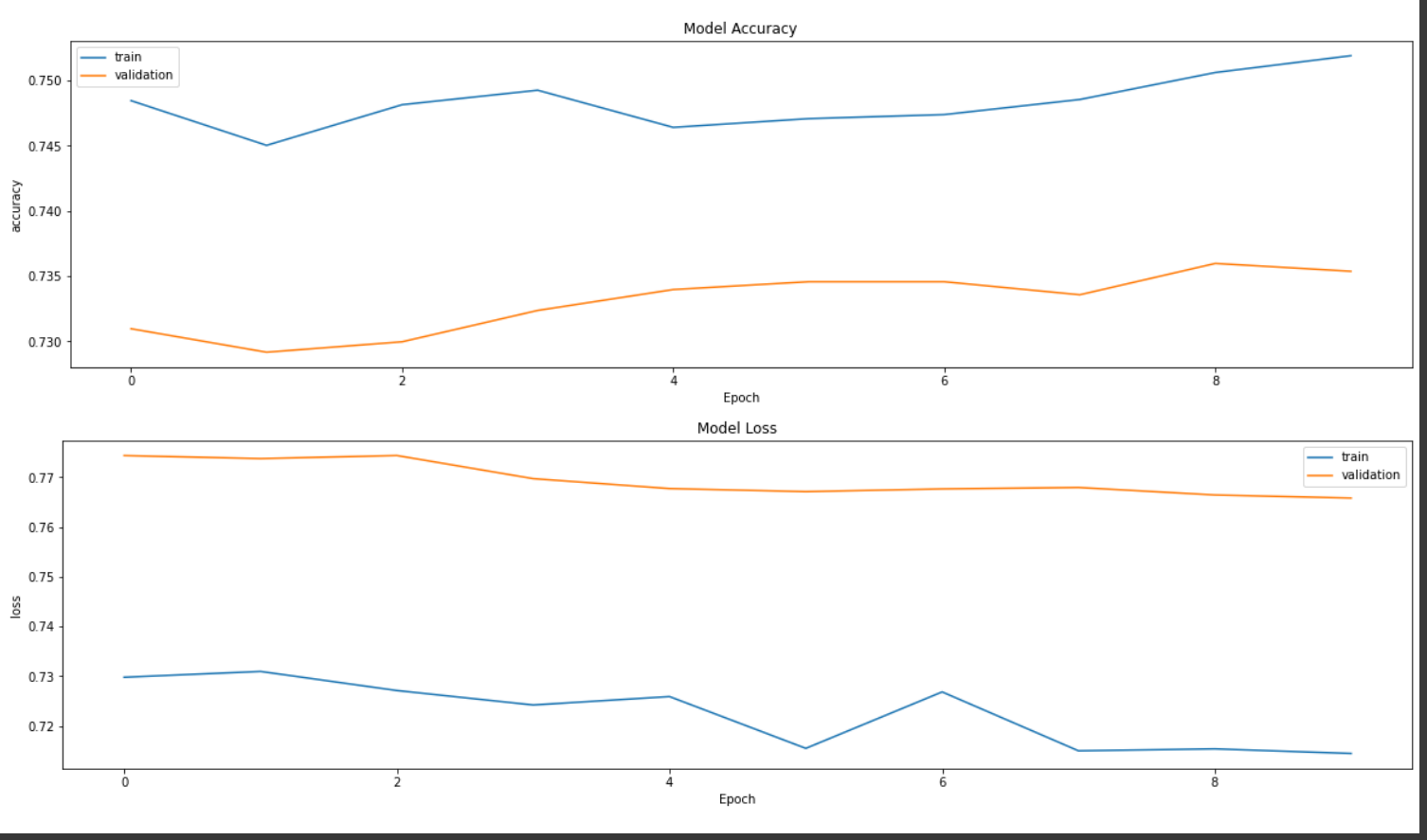
```
# load tendorboard and show the AlexNet logs
%tensorboard --logdir  ./logs/alexnet
```

```
Reusing TensorBoard on port 6006 (pid 264), started 0:53:50 ago. (Use '!kill 264' to kill
```



Create simple plot to show the accuracy and efficiency of the model

```
plt.figure(figsize=(20,5), facecolor='w',)
plt.title('Model Accuracy')
plt.plot(alexnet_history.history['accuracy'])
plt.plot(alexnet_history.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('accuracy')
plt.legend(labels=['train','validation'])
plt.show()

plt.figure(figsize=(20,5), facecolor='w')
plt.title('Model Loss')
plt.plot(alexnet_history.history['loss'])
plt.plot(alexnet_history.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.legend(labels=['train','validation'])
plt.show()
```

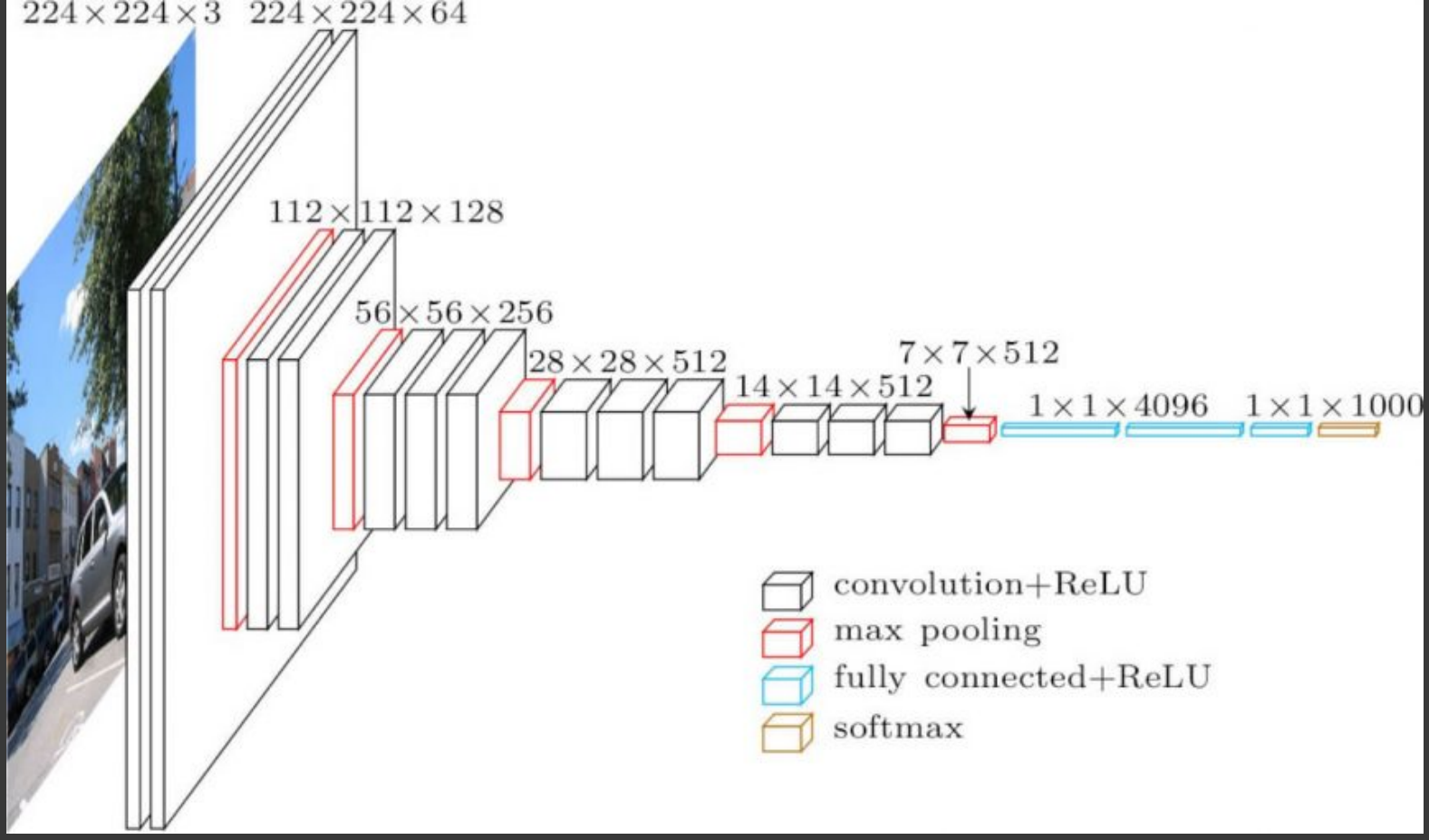


VGG 16 [2014]

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”.

Initially the chosen algorithms for this coursework were AlexNet and GoogLeNet. It was chosen so to compare the two models based on the huge parameter difference (AlexNet 60 million / GoogLeNet 4 million). Due to the difficulty of implementing GoogLeNet and the training times, it was decided to implement VGG16, which has a more simplistic architecture similar to that of AlexNet.

Therefore, we will compare the accuracy of the two models and instead of choosing a lighter architecture we will implement VGG16 which has almost 140 million parameters, more than double the amount of these in AlexNet.



VGG 16 managed to win second place behind GoogLeNet in ILSVRC 2014 image classification chalenge with a top 5 error rate of 7.6%. In comparison, AlexNet in the same challenge in 2012 achieved 15.3%. That was the first year ever that a Convolutional Neural Network achieved an error rate bellow 10%.

Although quite accurate, VGG has some major drawbacks,

- It is painfully slow to train
- The network architecture weights themselves are quite large, which means it requires more memory bandwidth.

However in comparison to AlexNet, VGG16 is a simpler model, as it is not using much hyper parameters.

Some of the key features of VGG 16 are;

- Small kernel sizes in each convolution(3 x 3), it combines smaller kernels to emulate larger receptive fields.
- 16 layers, max pooling between some layers(stride: 2, size: 2)
- Similar learning process to AlexNet.
- Very slow to train.

To train VGG we will use the same dataset as above and similar settings, as it can be seen bellow. Again, we will make use of tensorflow profile to ensure the model is not input bound

```
#create directory to store profiler data for each run
root_logdir = os.path.join(os.curdir, "logs//vgg16")
```

```
def get_run_logdir():
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir, profile_batch = '500,520')
```

```
# Construct VGG
vgg = keras.models.Sequential([
        # 1-2
        keras.layers.Conv2D(64, 3,  input_shape=(224,224,3), activation='relu', padding='same'
        keras.layers.Conv2D(64, 3, activation='relu', padding='same'),
        keras.layers.MaxPooling2D(2, 2),


        # 3-4
        keras.layers.Conv2D(128, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(128, 3, activation='relu', padding='same'),
        keras.layers.MaxPooling2D(2, 2),


        # 5-7
        keras.layers.Conv2D(256, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(256, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(256, 3, activation='relu', padding='same'),
        keras.layers.MaxPooling2D(2, 2),

        # 8-10
        keras.layers.Conv2D(512, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(512, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(512, 3, activation='relu', padding='same'),
        keras.layers.MaxPooling2D(2, 2),

        # 11-13
        keras.layers.Conv2D(512, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(512, 3, activation='relu', padding='same'),
        keras.layers.Conv2D(512, 3, activation='relu', padding='same'),
        keras.layers.MaxPooling2D(2, 2),

        # 14
        keras.layers.GlobalAveragePooling2D(),
        keras.layers.Dense(4096, activation='relu'),
        keras.layers.Dropout(0.5),

        # 15
        keras.layers.Dense(4096, activation='relu'),
        keras.layers.Dropout(0.5),

        # 16
        keras.layers.Dense(10, activation='softmax')
    ])
```

```
#Learning Rate Annealer
```

```
lrr= ReduceLROnPlateau(   monitor='val_accuracy',   factor=.01,   patience=3,  min_lr=1e-5)
```

```
# Compile the model and implement optimiser
vgg.compile(loss='sparse_categorical_crossentropy', optimizer=tf.optimizers.Adam(lr=0.001), me
```

```
#Start training the model
vgg_history = vgg.fit(
          processed_train_ds,
          epochs=20,
          validation_data=processed_validation_ds,
          validation_freq=1,
          callbacks=[lrr,tensorboard_cb]
          )
```

```
Epoch 1/20
351/351 [==============================] - 638s 2s/step - loss: 3.9957 - accuracy: 0.1721
Epoch 2/20
351/351 [==============================] - 617s 2s/step - loss: 1.7420 - accuracy: 0.3434
Epoch 3/20
351/351 [==============================] - 616s 2s/step - loss: 1.5353 - accuracy: 0.4337
Epoch 4/20
351/351 [==============================] - 616s 2s/step - loss: 1.3530 - accuracy: 0.5075
Epoch 5/20
351/351 [==============================] - 615s 2s/step - loss: 1.2275 - accuracy: 0.5573
Epoch 6/20
351/351 [==============================] - 615s 2s/step - loss: 1.0905 - accuracy: 0.6098
Epoch 7/20
351/351 [==============================] - 614s 2s/step - loss: 0.9739 - accuracy: 0.6574
Epoch 8/20
351/351 [==============================] - 614s 2s/step - loss: 0.8900 - accuracy: 0.6876
Epoch 9/20
351/351 [==============================] - 613s 2s/step - loss: 0.8010 - accuracy: 0.7191
Epoch 10/20
351/351 [==============================] - 614s 2s/step - loss: 0.7250 - accuracy: 0.7457
Epoch 11/20
351/351 [==============================] - 614s 2s/step - loss: 0.6509 - accuracy: 0.7738
Epoch 12/20
351/351 [==============================] - 613s 2s/step - loss: 0.5881 - accuracy: 0.7917
Epoch 13/20
351/351 [==============================] - 613s 2s/step - loss: 0.5164 - accuracy: 0.8189
Epoch 14/20
351/351 [==============================] - 612s 2s/step - loss: 0.4542 - accuracy: 0.8407
Epoch 15/20
351/351 [==============================] - 612s 2s/step - loss: 0.4169 - accuracy: 0.8554
Epoch 16/20
351/351 [==============================] - 612s 2s/step - loss: 0.3779 - accuracy: 0.8693
Epoch 17/20
351/351 [==============================] - 611s 2s/step - loss: 0.3237 - accuracy: 0.8873
Epoch 18/20
351/351 [==============================] - 611s 2s/step - loss: 0.2904 - accuracy: 0.8991
Epoch 19/20
351/351 [==============================] - 611s 2s/step - loss: 0.2699 - accuracy: 0.9060
Epoch 20/20
351/351 [==============================] - 610s 2s/step - loss: 0.1910 - accuracy: 0.9332
```

Training the model for 20 epochs (approximatelly 2 hours) achieved an accuracy of 0.90 and a validation accuracy of 0.70. This indicates that the model ovefits the data after several epochs. When attempted to

implement data augmentation as an attempt to reduce this, the model wouldnt score above 0.5 validation accuracy. More attempts were made to fix the overfit problem but with no avail.

To improve validation performance we attempted some techniques used in AlexNet earlier,

1. Added batch normalisation after each convolutional layer, this lead to increasing the epoch time by 2 but also increasing the accuracy per epoch. As the model is quite slow already, it was decided best to not implement it at all.
2. Implemented a dropout of (0.2 to 0.5) after each convolution.
3. Added kernel initializer = 'he_uniform'.
4. Implemented data augmentation.
5. Tried SGD with no success, as the highest accuracy achieved was 0.63 and validation accuracy 0.65. Adam fitted better
6. Tested opyimizer momentum with SGD above and bellow 0.
7. Increased / Decreased batch sizes.

After numerous tests, the model didnt show significant improvement, some of the implementations made the model so heavy, that the system crashed when attempting to run it.

The final version of the model can be seen below.

```
#Show a summary of the model's layers
vgg.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 64)      1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 224, 224, 64)      36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 64)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 128)     73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 128)     147584
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 56, 56, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 56, 56, 256)       590080
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 56, 56, 256)       590080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 512)       1180160
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 28, 28, 512)       2359808
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 512)       2359808
_________________________________________________________________
```

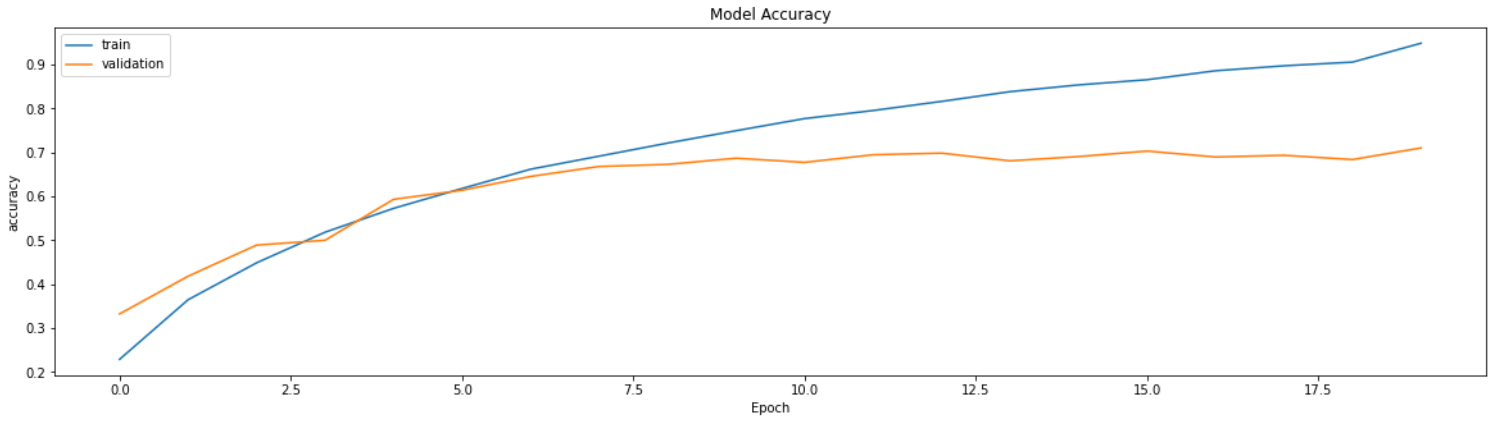
```
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 512)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 512)         0
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 4096)              2101248
_________________________________________________________________
dropout (Dropout)            (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                40970
=================================================================
Total params: 33,638,218
Trainable params: 33,638,218
Non-trainable params: 0
_________________________________________________________________
```

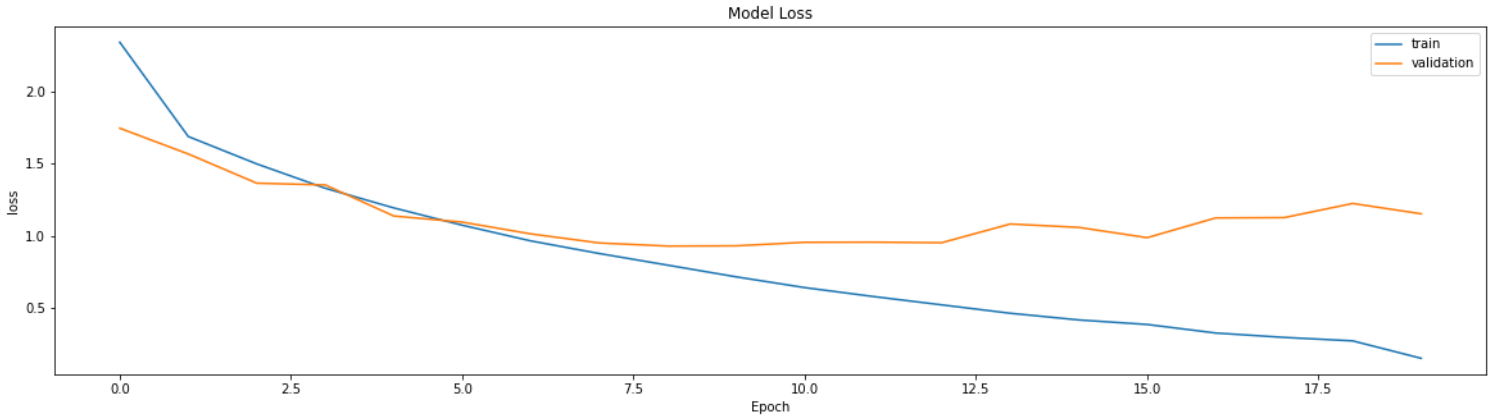
```
#Evaluate the model on the test dataset
vgg.evaluate(processed_test_ds)
```

```
78/78 [==============================] - 42s 472ms/step - loss: 1.2027 - accuracy: 0.7045
[1.2026511430740356, 0.7045272588729858]
```

```
%tensorboard --logdir  ./logs/vgg16
```

```
# Create a plot to mesure accuracy and loss
plt.figure(figsize=(20,5), facecolor='1',)
plt.title('Model Accuracy')
plt.plot(vgg_history.history['accuracy'])
plt.plot(vgg_history.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('accuracy')
plt.legend(labels=['train','validation'])
plt.show()

plt.figure(figsize=(20,5), facecolor='1')
plt.title('Model Loss')
plt.plot(vgg_history.history['loss'])
plt.plot(vgg_history.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.legend(labels=['train','validation'])
plt.show()
```





As can be seen from the diagram above, the model starts to overfit the data from the 5th epoch. Despite the efforts to reduce this, I was unable to maintain the original VGG16 architecture and yet achieve good accuracy score. The problem with this model is that it is much heavier than AlexNet, and each epoch requires at least 10 minutes, meaning that changes require big amount of time to be visible, thus making it hard to train this model with the available resources.

The same model with architecture changes to work without image resizing, manages to achieve better results as above with each epoch taking 6s and with much less data overfit.

However, with so many changes done to the structure of the model, could it still be called VGG16?

```
# Test VGG without image resizing
vgg_small = keras.models.Sequential([
  keras.layers.Conv2D(32,(3,3), activation = 'relu', kernel_initializer = 'he_uniform', paddin
  keras.layers.BatchNormalization(),
  keras.layers.Conv2D(32,(3,3), activation = 'relu', kernel_initializer = 'he_uniform', paddin
  keras.layers.BatchNormalization(),
```

```
  keras.layers.MaxPooling2D((2,2)),
  keras.layers.Dropout(0.2),
  keras.layers.Conv2D(64,(3,3), activation = 'relu', kernel_initializer = 'he_uniform', paddin
  keras.layers.BatchNormalization(),
  keras.layers.Conv2D(64,(3,3), activation = 'relu', kernel_initializer = 'he_uniform', paddin
  keras.layers.BatchNormalization(),
  keras.layers.MaxPooling2D((2,2)),
  keras.layers.Dropout(0.3),
  keras.layers.Conv2D(128,(3,3), activation = 'relu', kernel_initializer = 'he_uniform', paddi
  keras.layers.BatchNormalization(),
  keras.layers.Conv2D(128,(3,3), activation = 'relu', kernel_initializer = 'he_uniform', paddi
  keras.layers.BatchNormalization(),
  keras.layers.MaxPooling2D((2,2)),
  keras.layers.Dropout(0.4),
  keras.layers.Flatten(),
  keras.layers.Dense(128, activation = 'relu', kernel_initializer = 'he_uniform'),
  keras.layers.BatchNormalization(),
  keras.layers.Dropout(0.5),
  keras.layers.Dense(10, activation = 'softmax'),
])
```

```
# Compile the model and implement optimiser
vgg_small.compile(loss='sparse_categorical_crossentropy', optimizer=tf.optimizers.Adam(lr=0.00
```

```
#Start training the model
vgg_small_history = vgg_small.fit(
          processed_train_ds,
          epochs=20,
          validation_data=processed_validation_ds,
          validation_freq=1,
          callbacks=[lrr,tensorboard_cb]
          )
```

```
Epoch 1/20
351/351 [==============================] - 7s 16ms/step - loss: 2.3386 - accuracy: 0.2757
Epoch 2/20
351/351 [==============================] - 7s 17ms/step - loss: 1.5445 - accuracy: 0.4459
Epoch 3/20
351/351 [==============================] - 6s 15ms/step - loss: 1.3240 - accuracy: 0.5235
Epoch 4/20
351/351 [==============================] - 6s 15ms/step - loss: 1.1719 - accuracy: 0.5851
Epoch 5/20
351/351 [==============================] - 6s 15ms/step - loss: 1.0887 - accuracy: 0.6172
Epoch 6/20
351/351 [==============================] - 6s 15ms/step - loss: 1.0184 - accuracy: 0.6441
Epoch 7/20
351/351 [==============================] - 6s 15ms/step - loss: 0.9540 - accuracy: 0.6603
Epoch 8/20
351/351 [==============================] - 6s 15ms/step - loss: 0.9166 - accuracy: 0.6763
Epoch 9/20
351/351 [==============================] - 6s 15ms/step - loss: 0.8732 - accuracy: 0.6944
Epoch 10/20
351/351 [==============================] - 6s 15ms/step - loss: 0.8318 - accuracy: 0.7106
Epoch 11/20
351/351 [==============================] - 6s 15ms/step - loss: 0.7915 - accuracy: 0.7257
Epoch 12/20
```

```
351/351 [==============================] - 6s 15ms/step - loss: 0.7712 - accuracy: 0.7301
Epoch 13/20
351/351 [==============================] - 6s 15ms/step - loss: 0.7326 - accuracy: 0.7449
Epoch 14/20
351/351 [==============================] - 6s 15ms/step - loss: 0.7116 - accuracy: 0.7540
Epoch 15/20
351/351 [==============================] - 6s 15ms/step - loss: 0.6779 - accuracy: 0.7639
Epoch 16/20
351/351 [==============================] - 6s 15ms/step - loss: 0.6675 - accuracy: 0.7684
Epoch 17/20
351/351 [==============================] - 6s 15ms/step - loss: 0.6406 - accuracy: 0.7775
Epoch 18/20
351/351 [==============================] - 6s 15ms/step - loss: 0.6249 - accuracy: 0.7801
Epoch 19/20
351/351 [==============================] - 6s 15ms/step - loss: 0.6041 - accuracy: 0.7889
Epoch 20/20
351/351 [==============================] - 6s 15ms/step - loss: 0.5866 - accuracy: 0.7929
```

```
#Show a summary of the models layers
vgg_small.summary()
```

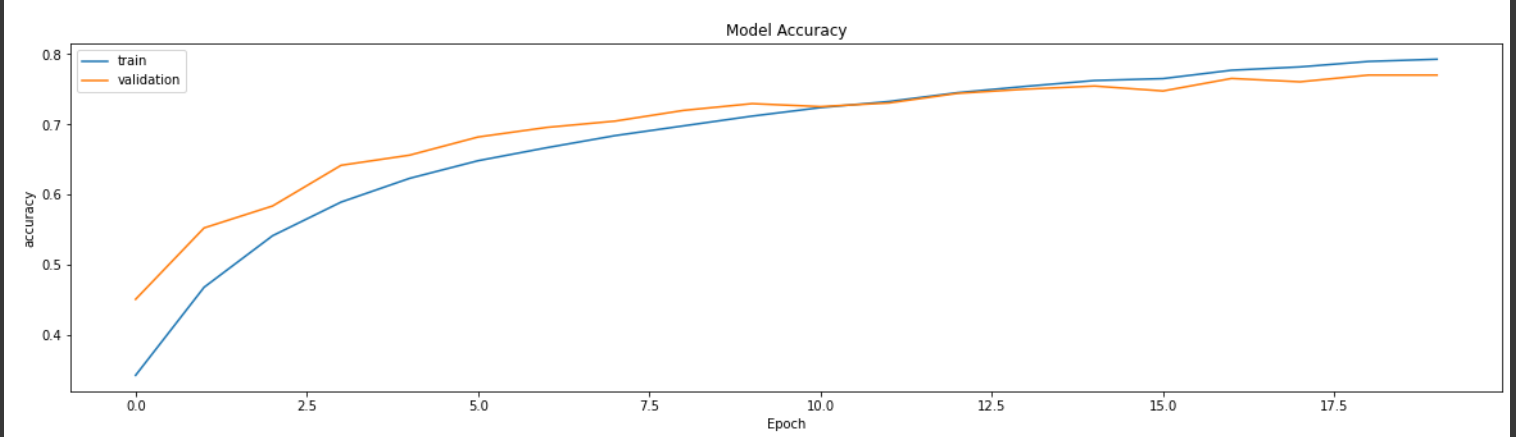
```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_11 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization_15 (Batc (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_16 (Batc (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_17 (Batc (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_18 (Batc (None, 16, 16, 64)        256
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_8 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_19 (Batc (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
batch_normalization_20 (Batc (None, 8, 8, 128)         512
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
dropout_9 (Dropout)          (None, 4, 4, 128)         0
```

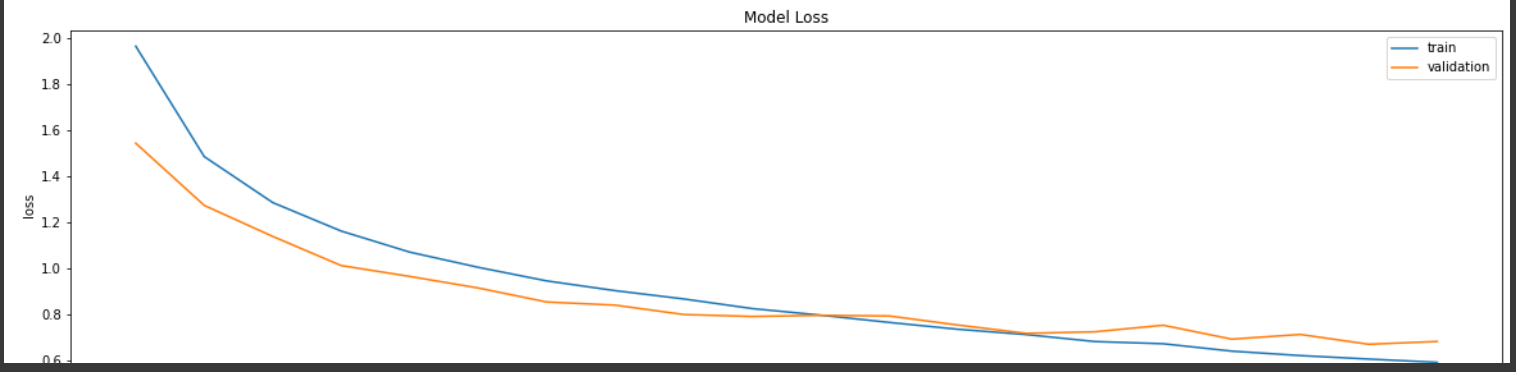
```
flatten_2 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               262272
_________________________________________________________________
batch_normalization_21 (Batc (None, 128)               512
_________________________________________________________________
dropout_10 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 552,874
Trainable params: 551,722
Non-trainable params: 1,152
_________________________________________________________________
```

```
#Evaluate the model on the test dataset
vgg_small.evaluate(processed_test_ds)
```

```
78/78 [==============================] - 1s 5ms/step - loss: 0.7165 - accuracy: 0.7595
[0.7164577841758728, 0.7595152258872986]
```

```
# Create a plot to mesure accuracy and loss
plt.figure(figsize=(20,5), facecolor='1',)
plt.title('Model Accuracy')
plt.plot(vgg_small_history.history['accuracy'])
plt.plot(vgg_small_history.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('accuracy')
plt.legend(labels=['train','validation'])
plt.show()

plt.figure(figsize=(20,5), facecolor='1')
plt.title('Model Loss')
plt.plot(vgg_small_history.history['loss'])
plt.plot(vgg_small_history.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.legend(labels=['train','validation'])
plt.show()
```





Algorithm performance and comparison

Discuss previous findings and evaluate the algorithms

Based on the findings from training and testing, the two algorithms are quite different. We can comfortably admit that AlexNet performed better on the CIFAR-10 dataset than VGG16, with similar training time required but better accuracy results. It is important to highlight that despite VGG taking 10 times the amount of time per epoch than AlexNet, both models reach an accuracy of 0.73 at the same time, approximatelly 1 hour.

However, our implementation from VGG16 suffers greatly from overfit data. Despite the numerous attempts to solve this issue, we always reached the same result, that the model performed terribly with data augmentation implemented and without it would overfit the data.

The second implementation of VGG16 that is fitted for images of the size 32x32, would score much better and a lot faster. However, that model was no longer VGG16, rather it was a model inspired from VGG and customised to fit our needs.

One of the most helpfull technologies used in this coursework was tensorflow profiler. Essentially reduced the time per epoch for each model significantly by showing recomendations on how to make each model less input-bound, meaning that we could utilise the GPU to its full potential.

Both models, performed greatly and showed an accuracy of above 70% on the CIFAR-10 dataset and despite the implementation issues we managed to produce some satisfactory results.

Conclusion

As the form of this coursework was free to explore, the models implemented were chosen for their simplicity and for the difference in the parameters. Each model has both pros and cons, but for the CIFAR-10 dataset due to its relatively small size, Alexnet performed better than VGG16.

It is important to mention that if the pixel size of the images in the dataset were bigger and we didnt have to resize them (32x32 to 224x224), both models would have achieved better results.

This can be seen by the customised implementation of VGG16 for the CIFAR-10 dataset without image resizing. This model however is far from the original model, and as this coursework aimed to explore the original models and not a custom implementation, we chose to resize the images.

Both models were evaluated based on the accuracy, the amount of time required to train, as well as other issues, such as data overfit.

In conclusion, AlexNet achieved better, more accurate results for the CIFAR-10 dataset. I believe this is due to VGG16 being designed for larger heavier datasets, the size of which we are unable to explore given the capabilities of Google Colab.