University of Salford MANCHESTER

Name: Jack Travis

ID: ---

Username: ---

Module Name: Deep Learning

Assessment Title: Developing a CNN for image classification

CRN: 56630

Overview of the assignment

The task of this assignment was to create a neural network that can identify digits expressed in American Sign Language, using the provided dataset. One must start by developing a neural network consisting of single layers of Conv2D, MaxPooling2D, Flatten and Dense layers, before improving upon it to create a more accurate solution, utilizing a range of other tools.

Provided were 1,929 images for training, 548 images for validation and 287 images for testing. I opted to use a batch size of 20 throughout the development of my model. Since the batch sizes are 20, other variables that rely on this data are also constants, such as:

- Steps_per_epoch=97
- Validation_steps=28
- (for evaluation) steps=15

This ensures all data is utilized.

Base Model

```
from keras import models, layers, optimizers
network = models.Sequential()
network.add(layers.Conv2D(256, (3,3), activation= 'relu', input_shape=(150,150,3)))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Flatten())
network.add(layers.Dense(10,activation='sigmoid'))
```

Image 1 – The network for the base model

```
from keras import optimizers
network.compile(loss='categorical_crossentropy', optimizer= optimizers.RMSprop(learning_rate=1e-4), metrics=['acc'] )
```

Image 2 – The compiler for the base network

For the base model, I opted to use the network shown above. As shown, and stated as a requirement in the assignment brief, only one of each mentioned layer was used to create the network. After some experimentation, I deemed that 256 filters for the Conv2D yielded the best results, as shown below:

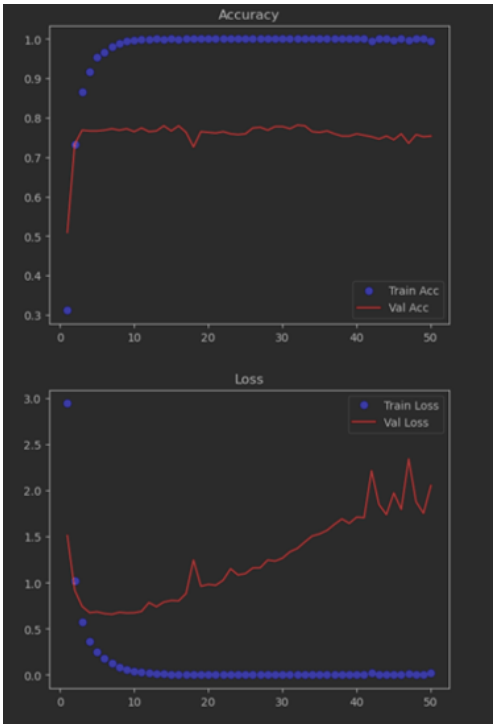

Image 3 – The Accuracy and Loss of the base model over 50 epochs

As the graphs shows, the network began to best suit the training and validation data in the epochs range of 6-10. Past this, the validation accuracy remains mostly unaffected. However, the Validation loss began to grow, indicating possible overfitting.

Using these graphs, the potentially best epoch can be found be seeing where the validation data accuracy is at/near its peek and where validation data loss is at/near its trough.

Based on analysing these, I opted to use 10 epochs for the base model, yielding the following results:

```
n 76  1  res = network.evaluate(test_generator, steps=14, verbose=1)
      2  print('Accuracy on test set: %.3f' % res[1])

         14/14 [==============================] - 0s 14ms/step - loss: 0.6894 - acc: 0.8000
         Accuracy on test set: 0.800
```

Image 4 – The accuracy of the 10th epoch of base model on the test data

As shown, the accuracy of the base model on the test set was 80%. This is a poor result, which was to be expected from such a simple network. Each feature map is also very large, requiring the subsequent Dense layer to require millions of parameters. This is unideal, and can lead to poor performance, as shown above. Therefore, the aim moving forward is to create a network that yields a result that is better than 80%. The first step I took was the addition of more layers.

Adding more layers

Currently, the feature extraction section of my network is very basic, consisting of a single Conv2D and MaxPooling2D, creating 256 feature maps of size 49x49. These feature maps are very big, causing the later Dense layer, and therefore the model, to require a significant number of parameters (314,716,682 total parameters). In order to extract smaller, more precise feature maps, more layers are required. Therefore, I created more Conv2D and MaxPooling2D layers, of varying filters, producing the network shown below:

```
network = models.Sequential()
network.add(layers.Conv2D(32, (3,3), activation= 'relu', input_shape=(150,150,3)))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(64,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(128,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(128,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Flatten())
network.add(layers.Dense(512, activation='relu'))
network.add(layers.Dense(10,activation='sigmoid'))
```

Image 5 – The model after altering feature extraction

This alteration results in 128 feature maps of size 4x4 being produced before the Flatten layer, resulting in the total parameters reducing to 1,295,050. The combination of an improved feature extraction, along with a 99.6% reduction of the total parameters yielded the following results:

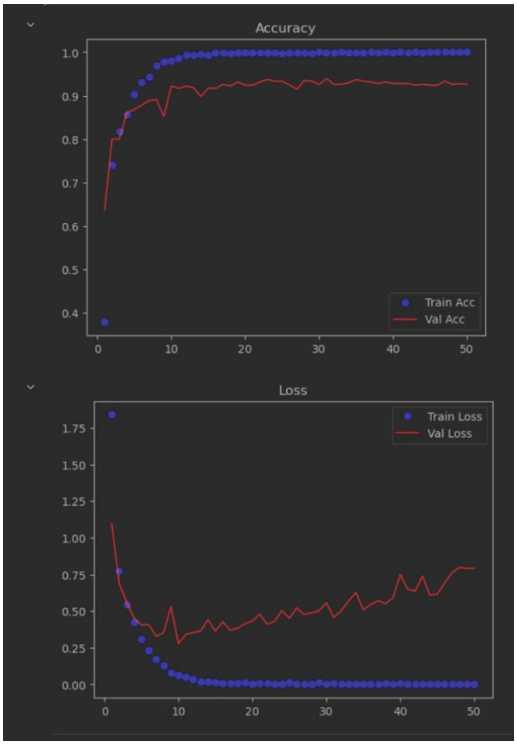

Image 6 – The results after altering feature extraction

As image 6 portrays, the validation accuracy of this network appears to be more than 10% greater than the previous model, suggesting it is an improved model. As with the base model, from epoch 10, the validation loss beings to climb, indicating possible overtraining. Analysis of the graph seems to suggest epoch 10 was the most successful, and hence is the one I chose to use with the testing data.

```
14/14 [==============================] - 0s 12ms/step - loss: 0.2257 - acc: 0.9286
Accuracy on test set: 0.929
```

Image 7- The results of the 10th epoch after altering feature extraction

As shown in image 7, this network successfully achieved 92.9% accuracy with the testing data: A 12.9% improvement over the base model.

As of the current model, the filters chosen for each layer in the feature extraction were randomly picked. Moving forward, I opted to find what quantity of filters worked the best on each layer, as well as experimenting with the filters of and total amount of Dense layers.

Finding more optimal filter values

Further experimentation was done via a trial-and-error method, altering the values of filters, and adding/removing/altering the Dense layers, making note of the current best performing network. The created network is as follows:

```
from keras import models, layers

network = models.Sequential()
network.add(layers.Conv2D(64, (3,3), activation= 'relu', input_shape=(100,100,3)))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(256,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(512,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(64,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))

network.add(layers.Flatten())
network.add(layers.Dense(512, activation='relu'))
network.add(layers.Dense(10,activation='sigmoid'))

network.summary()
```

Image 8 – The network after altering the filters

As seen in image 8, I found that increasing the total filters at the begging if the feature extraction, the decreasing them towards the end yielded the best result. I also found that altering the filters of or the number of Dense layers was detrimental to the model's success. Therefore, the Dense layers remain untouched.

At this point, it was realized that the source images were 100px by 100px, which the model was upscaling to 150px by 150px. Upscaling the image meant that more processing was occurring than necessary, and the potential loss resolution could be detrimental to the success of the model. As such, I change the input shape to 100px by 100px, to bring it inline with the source images.

The results of this network are as follows:

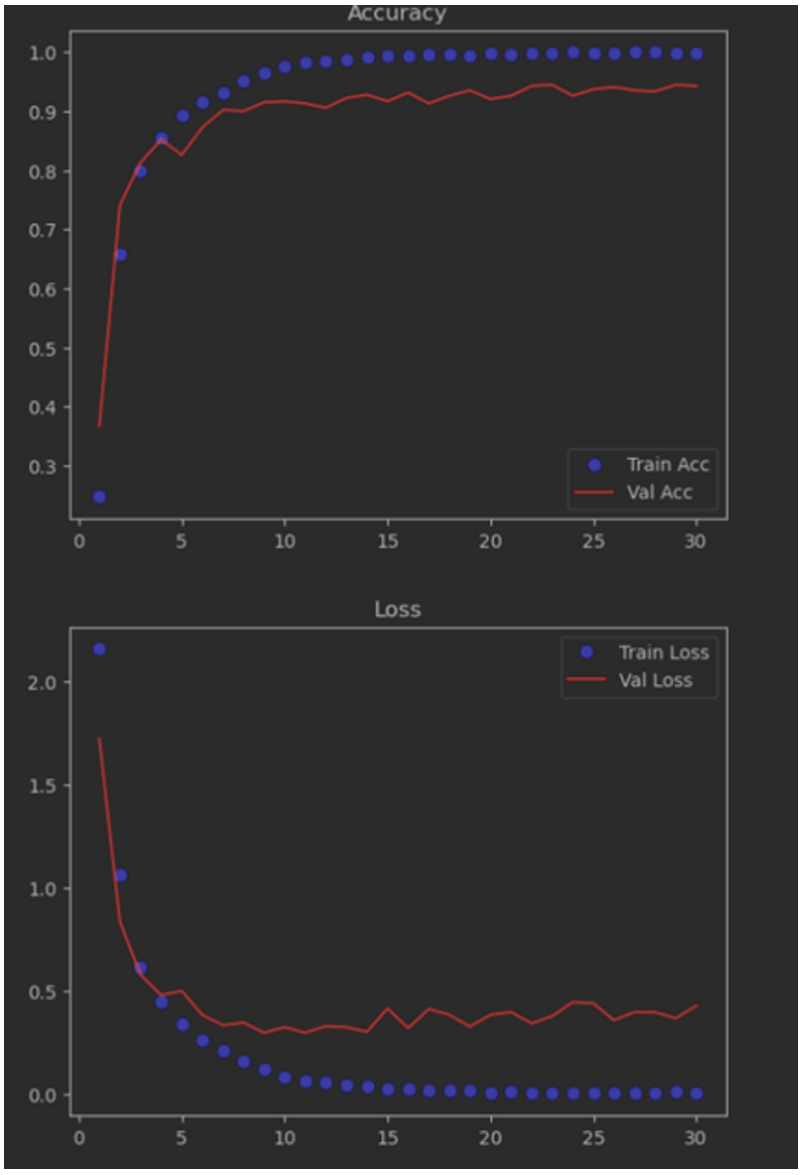

Image 9 – The results after altering the filters

As shown, the network appears to deal with overfitting significantly better with this model, with later epochs no longer leading to an increasing valuation loss. Epoch 22 was picked from this model, achieving the result shown below:

```
res = network.evaluate(test_generator, steps=14, verbose=1)
print('Accuracy on test set: %.3f' % res[1])
```

```
14/14 [==============================] - 0s 8ms/step - loss: 0.1718 - acc: 0.9643
Accuracy on test set: 0.964
```

Image 10 – The results of the 22nd epoch after altering the filters

An accuracy of 96.4% was reached, which is significantly better than the original base model, and a further increase on the previous solution. Moving forward, I opted to experiment with the addition of Dropout layers.

Adding Dropout layers

Dropout layers attempt to reduce overfitting by stopping a specified percentage of filters each cycle. Although overfitting is significantly less of a problem that it was previously for this network, there is a slight gradual increase in validation loss as the quantity of epochs increases. Thus, this model may benefit from the inclusion of Dropout layers. After experimentation, the most optimal solution found was the following:

```
from keras import models, layers

network = models.Sequential()
network.add(layers.Conv2D(64, (3,3), activation= 'relu', input_shape=(100,100,3)))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(256,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(512,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Conv2D(64,(3,3),activation= 'relu'))
network.add(layers.MaxPooling2D(2,2))
network.add(layers.Flatten())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(512, activation='relu'))
network.add(layers.Dense(10,activation='sigmoid'))



network.summary()
```

Image 10 – The model after the addition of a dropout layer

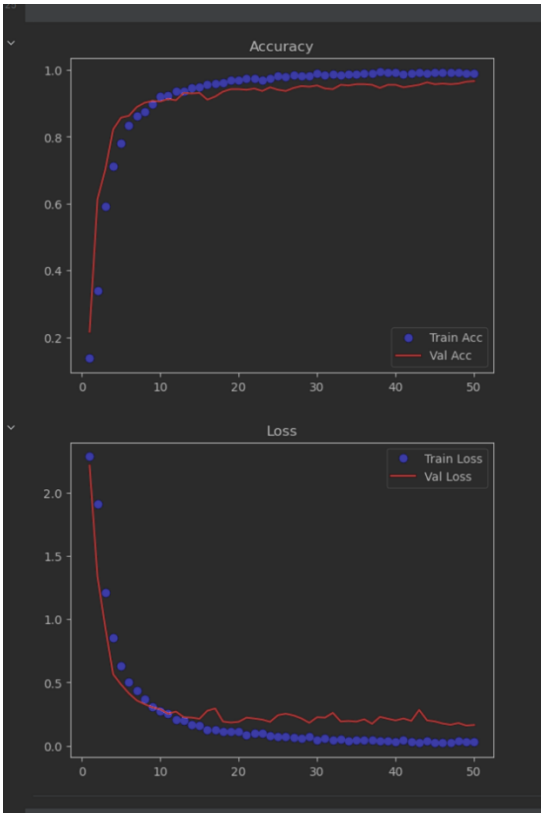

Image 11 – The results after the addition of a dropout layer

The addition of this dropout layer appeared to help level out the validation loss as more epochs were ran, as expected. This also means that later epochs, which may yield better results, may no be viable, due to their reduced validation loss. However, in this case, it was decided that epoch 22 appeared to still be the best, yielding the following results:

```
14/14 [==============================] - 0s 9ms/step - loss: 0.1059 - acc: 0.9750
Accuracy on test set: 0.975
```

Image 12 – The results of epoch 22 after the addition of a dropout layer

As shown, the addition of a Dropout layer led to a slight increase in the accuracy of the model. Moving forward, it was decided test the success of the model when using a pretrained feature extractor.

Feature Extractor

This step involved replacing the currently, custom-created feature extractor in favour of a pretrained feature extractor. For this step, both VGG16 and VGG19 were tested to see if they provided favourable results. After testing, the following model was established:

```
from keras import models, layers


from keras.applications import VGG16
vgg_base = VGG16(weights='imagenet', include_top=False,input_shape=(100,100,3))
network = models.Sequential()
network.add(vgg_base)
network.add(layers.Flatten())

network.add(layers.Dropout(0.5))

network.add(layers.Dense(512, activation='relu'))
network.add(layers.Dense(10,activation='sigmoid'))

network.summary()
```

Image 13 – The model after the addition of a trained feature extractor

After testing, the decision to change to the Adam optimizer and to reduce the learning rate was made. The change in optimizer was due to RMSProp occasionally becoming stuck on an unideal local maximum, providing inconsistent results. Changing the optimizer fixed this issue, whilst retaining a similar best result.

```
network.compile(loss='categorical_crossentropy', optimizer= optimizers.Adam(learning_rate=1e-5), metrics=['acc'] )


hist = network.fit(train_generator, steps_per_epoch=97, epochs=30,
                   validation_data=val_generator, validation_steps=28)
```

Image 14 – The updated network.compile()

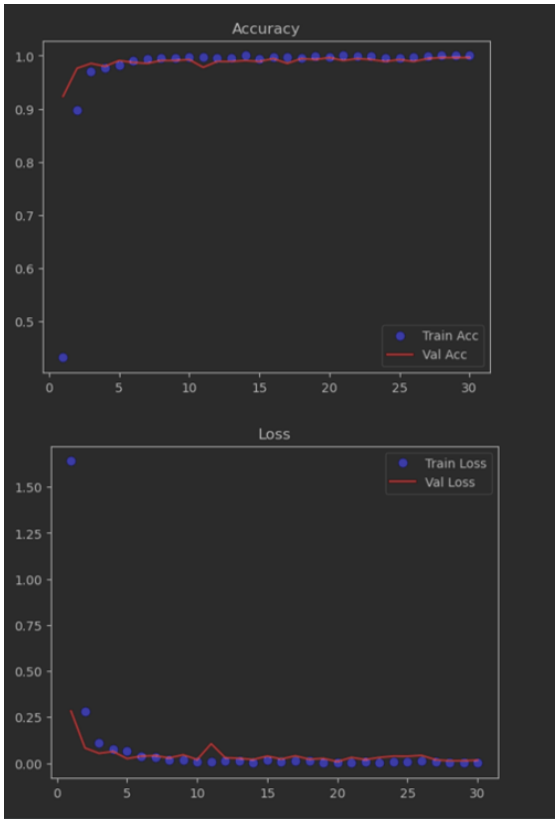

Image 15 – The results after the addition of a trained feature extractor

As seen in image 15, changing to the VGG16 base and altering the optimizer and learning rate was very successful. The model consistently yields high validation accuracy and low validation loss. After further testing, the 30th epoch appeared to be the most reliable in achieving a consistent result:

```
res = network.evaluate(test_generator, steps=15, verbose=1)
print('Accuracy on test set: %.3f' % res[1])

 15/15 [==============================] - 1s 90ms/step - loss: 6.2203e-04 - acc: 1.0000
 Accuracy on test set: 1.000
```

Image 15 – The results of the 30th epoch after the addition of a trained feature extractor

As the above image shows, the network was able to successfully predict the category every test image belonged to. At this point, the model cannot be improved further with the current dataset. It can successfully classify every ASL digit provided in the testing set.

To further prove the success rate, or to further train this network to ensure it continues working over a more diverse dataset would require more data, which I do not have access to.

Despite this fact, data augmentation was also tested to see what effect it would have on the model.

Data Augmentation

Data Augmentation is the process of editing the test images to create a more diverse dataset that may be more reflective of all images that may be encountered for which it is being trained to detect. This can help the model to focus on more important, defining features of the images that it might miss otherwise.

In this case, the addition of data augmentation had no effect, or was detrimental to the system, and as such was removed from the system. Below shows what augmentation was done, and the results of testing.

```
train_datagen = ImageDataGenerator(rotation_range=40, rescale=1./255,width_shift_range=0.2,
height_shift_range=0.2, shear_range=0.2, horizontal_flip=False, fill_mode='nearest')
```

Image 16 – The data augmentation

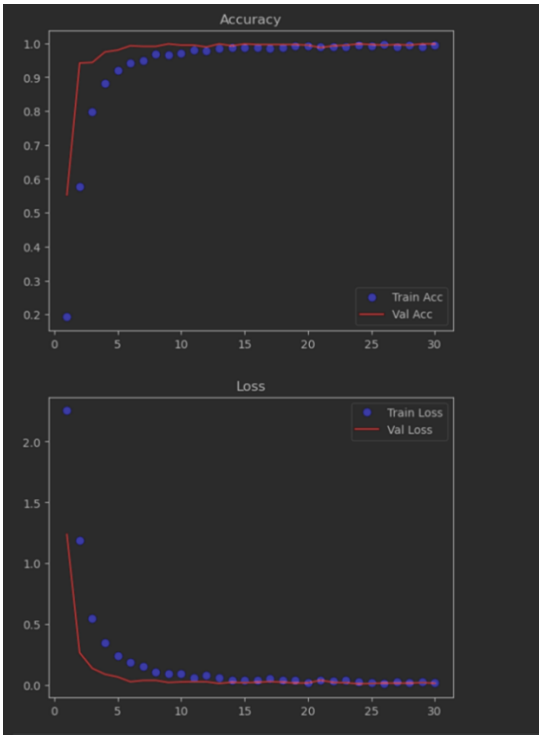

Image 17 – The results of using Data Augmentation

```
res = network.evaluate(test_generator, steps=15, verbose=1)
print('Accuracy on test set: %.3f' % res[1])
```

```
15/15 [==============================] - 0s 23ms/step - loss: 0.0089 - acc: 0.9930
Accuracy on test set: 0.993
```

Image 18 – The results of the 30th epoch using Data Augmentation

As image 18 shows, some of the results when using data augmentation were worse than without it. Add this feature could be beneficial for the network to predict ASL digits more reliably in a more diverse dataset. However, since maximum accuracy has already been achieved, it is much harder to prove that adding different data augmentations would help such a situation. Data that is incorrectly identified by the network first needs to be found before progress with adding this feature can be made.

Comparison of models to other research

As show in the previous sections, a successful attempt at creating a better model was made. The base model only achieved an accuracy of 80% when passed the testing data, whereas the improved model achieved 100% accuracy. This implies the network is near perfect for the data that has been provided. However, its realistic effectiveness in a real word deployment could differ greatly. This will be explained more in the subsequent section.

(Mavi , *A New Dataset and Proposed Convolutional Neural Network Architecture for Classification of American Sign Language Digits*) stated that their attempt at recognising digits of ASL was met with a success rate of 95-97%. Therefore statistically, the neural network that I have created was more successful at accomplishing its task. However, to say mine is better would be to disregard the different variables surrounding each network.

For example, both models are using a different dataset. Although both datasets are of the same image types, slight differences in each individual image could cause one model to perform better than another. To properly compare both models, they should both use the same dataset to attempt to classify the images.

Another factor to consider is the success of each system when deployed in a real-life situation, or a much more diverse dataset for testing. The success of each model could drastically vary given this. This would require further testing.

As the above points show, it would be unfair to solely base which model is better on the current information.

Limiting factors of the dataset

As mentioned previously, the provided data was as follows: 1,929 images for training, 548 images for validation and 287 images for testing. The amount of data is fairly limited and may not be truly reflective of the whole population. For example, there are 10 possible digits, meaning there are less than 20 images per category. This quantity of images is not enough to truly reflect all the examples the network may come across if deployed. Using this network on other examples could yield significantly different results. More data needs to be required to confirm the success rate of the network, or to help improve the solution further.

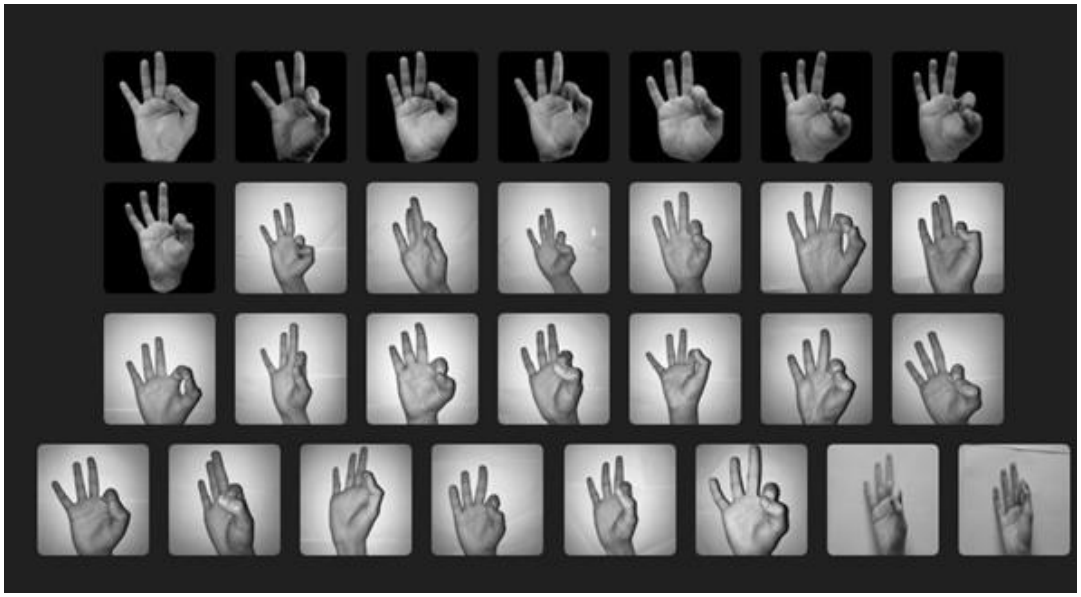

Image 19 – All ASL digit 9 used as testing data

Another limiting factor is the diversity of the provided images. As image 19 shows, all hands are in front of a relative plain background, centred to the image and are facing the camera at very similar angles. This makes every image of the same category to be very similar to each other. If a less 'perfect' image was inserted into the testing batch, it is quite possible it will get an incorrect classification, due to expecting the image to look very similar to those shown above.

The failure of data augmentation supports this point, due to moving and rotating the training images by small amounts led to a network that was slightly worse at identifying the ASL digits in the testing batch. This would have to be investigated to confirm/deny this potential floor in the dataset.

Another notable feature is the lack of noise in the images. There is no complexed scenery in the background that might be present when using the system in a real-world scenario. Since the model has never learnt how to deal with this, it may begin to focus on wrong features and start incorrectly classifying the ASL digits.

Conclusions

The model created was very successful at identifying ASL digits in the provided data set. However, due to a lack of diverse images, and a lack of total images, the performance of the neural network may significantly differ. Further investigation and potential improvements are required moving forward when more data is acquired.

Lessons learnt

I learnt the following lessons:

- A greater understanding of how to use Python.
- Learning how to use a new IDE: DataSpell, which uses a Jupyter Server to execute the written code.
- How to install and use Anaconda, tensorflow and tensorflow-gpu.
- How neural networks work.
- How to create and test the effectiveness of a neural network.
- How to create a Sequential Model, consisting of Cov2D, MaxPooling2D, Flatten, Dense and Dropout layers.
- How to utilize pretrained feature extractors, such as VGG16 and VGG19
- What different optimizer options there are (such as RMSProp and Adam) and the advantages and disadvantages of them.

References

Mavi , A. (no date) *A New Dataset and Proposed Convolutional Neural Network Architecture for Classification of American Sign Language Digits*. Available at: https://doi.org/10.48550/arXiv.2011.08927 (Accessed: January 11, 2023).