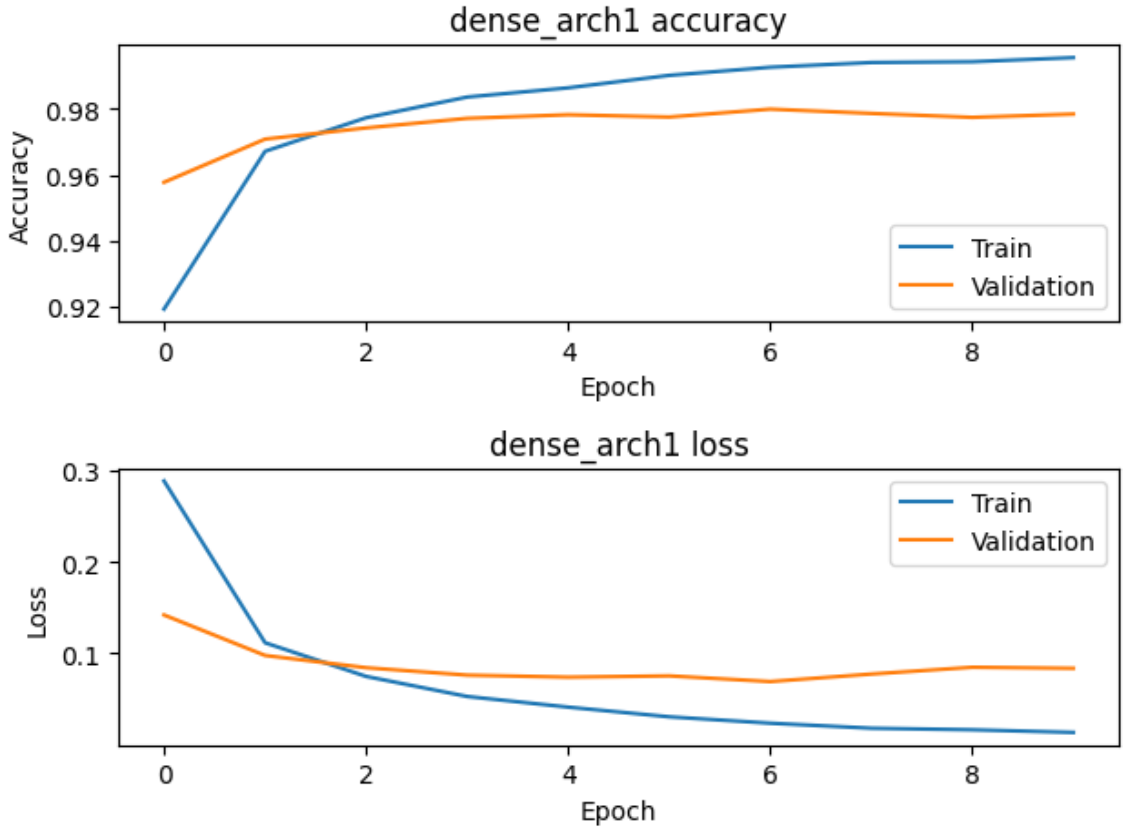
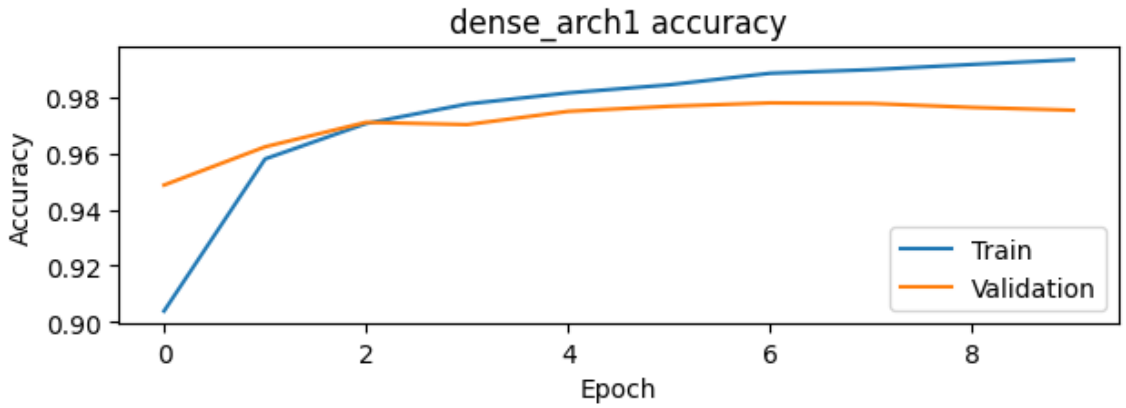
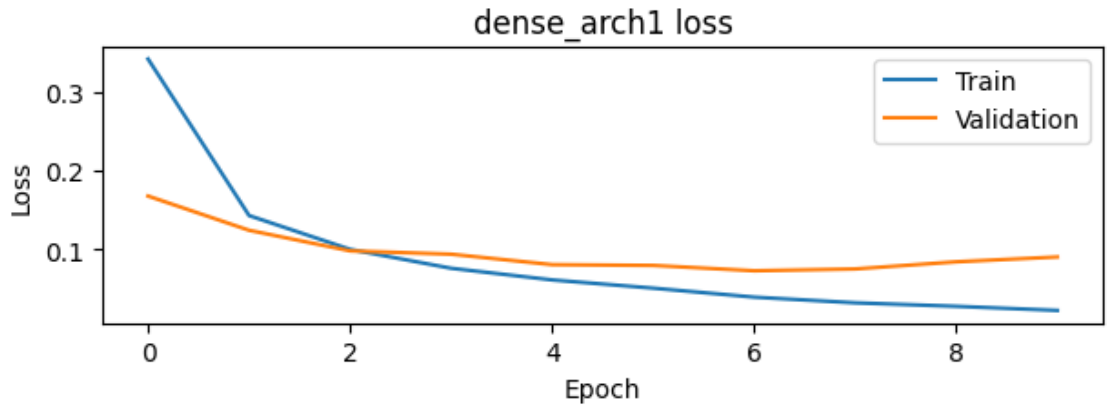
1.3 - Number of Parameters

In [ ]:
```
hidden_layers = (200, 100)
```

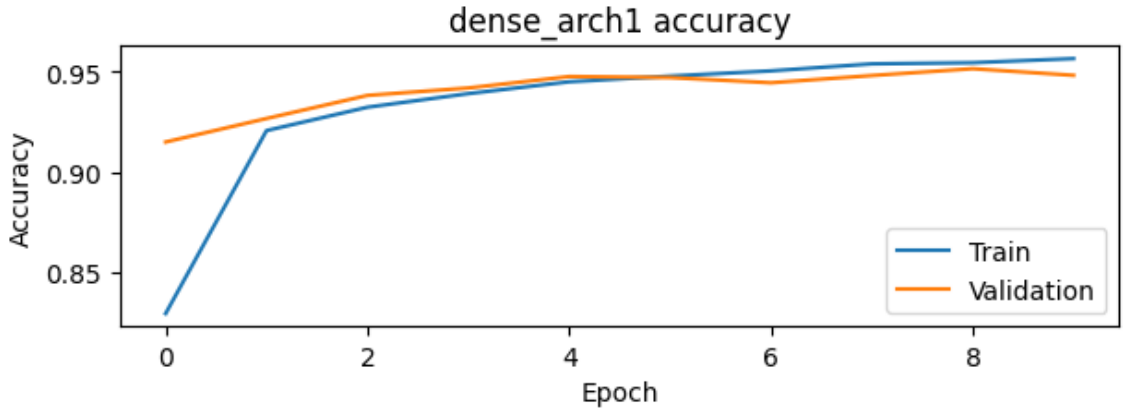


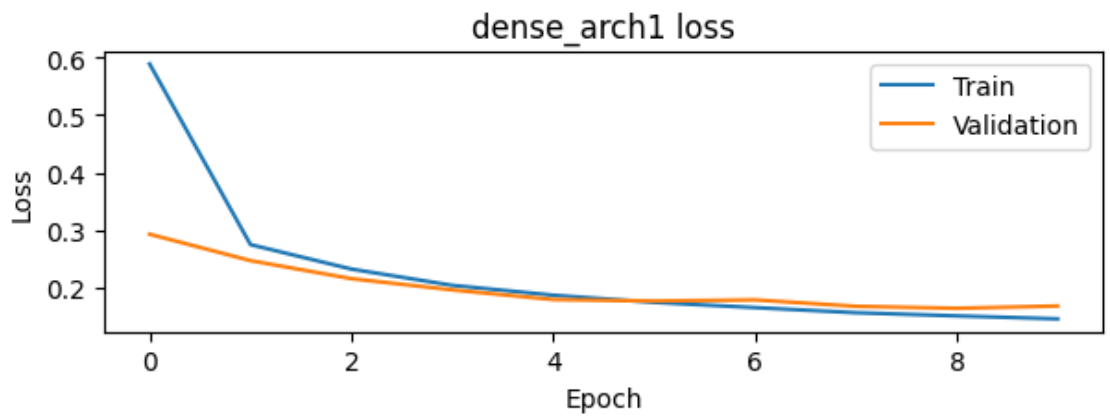
In [ ]:
```
hidden_layers = (100, 100)
```



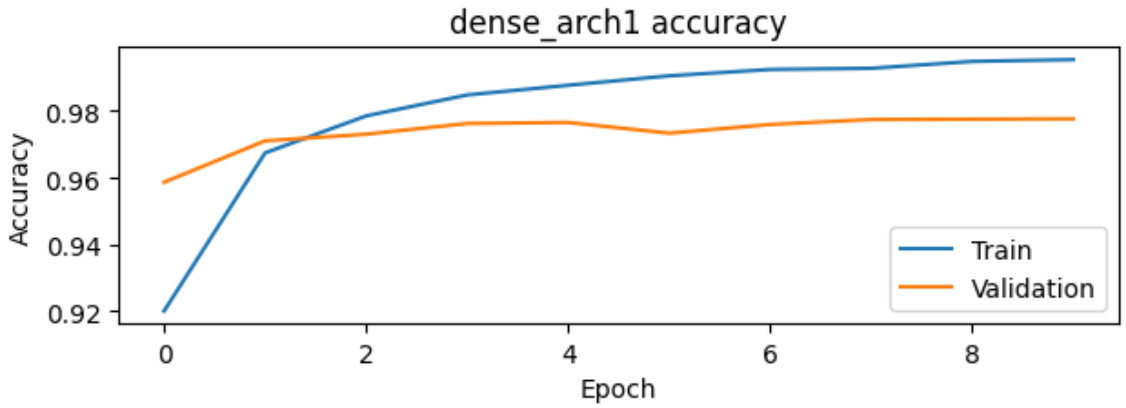


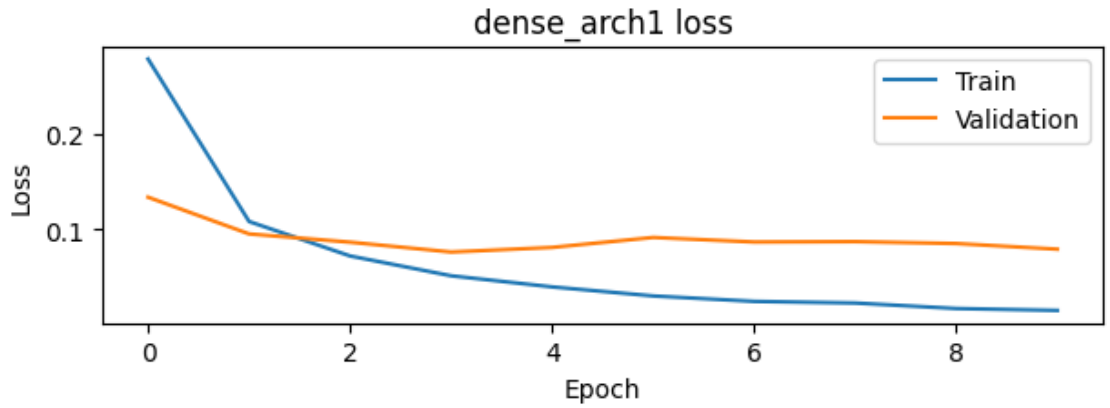
In [ ]:
```
hidden_layers = (10, 100)
```



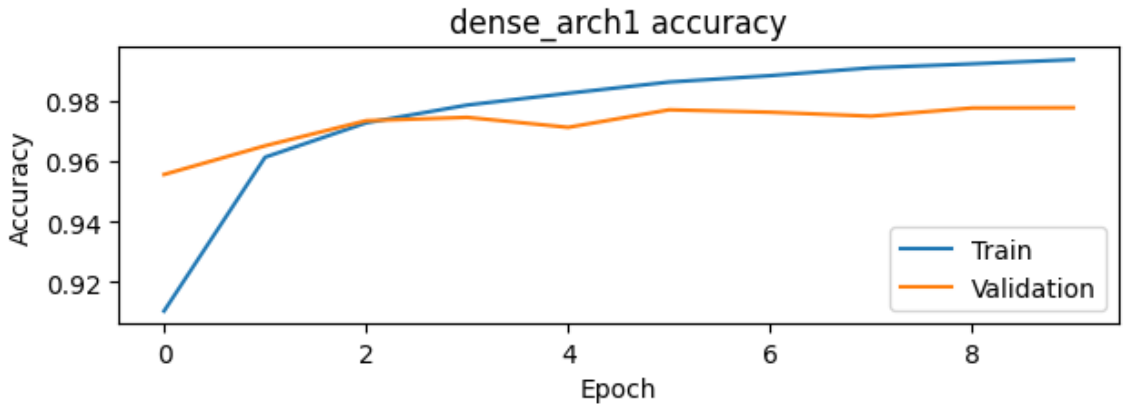


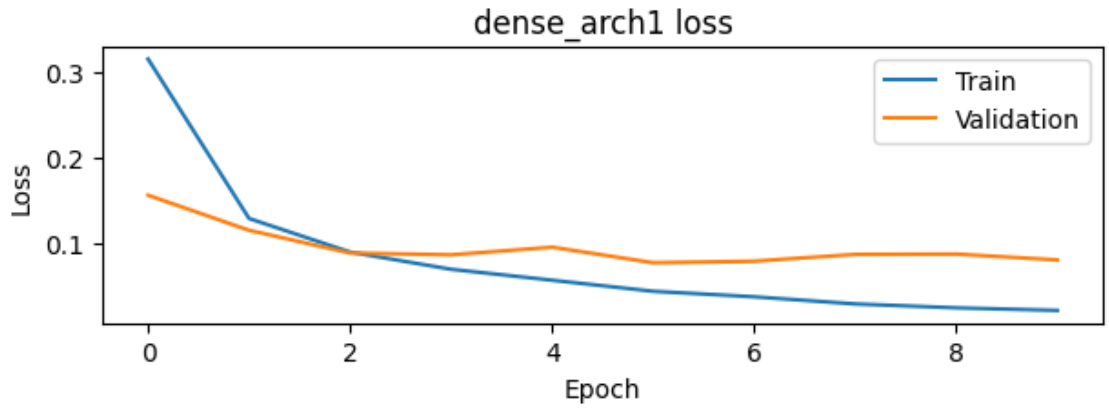
In [ ]:
```
hidden_layers = (200, 200)
```



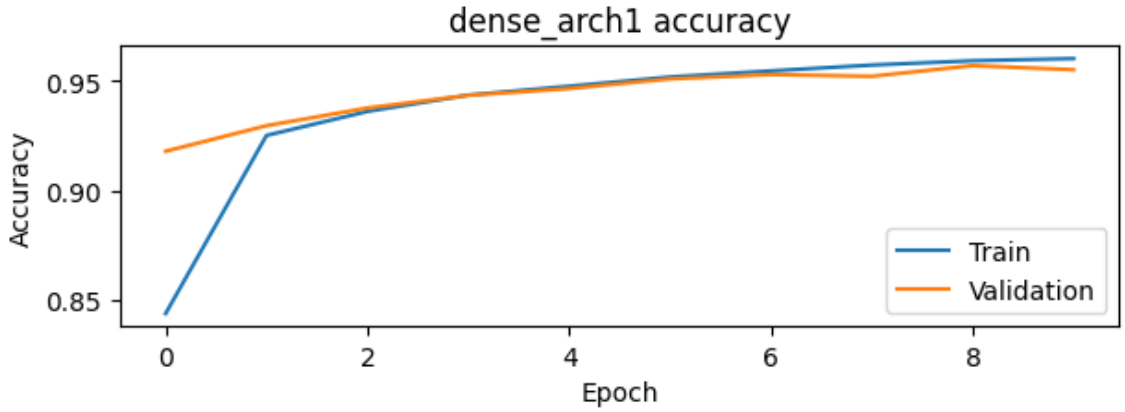


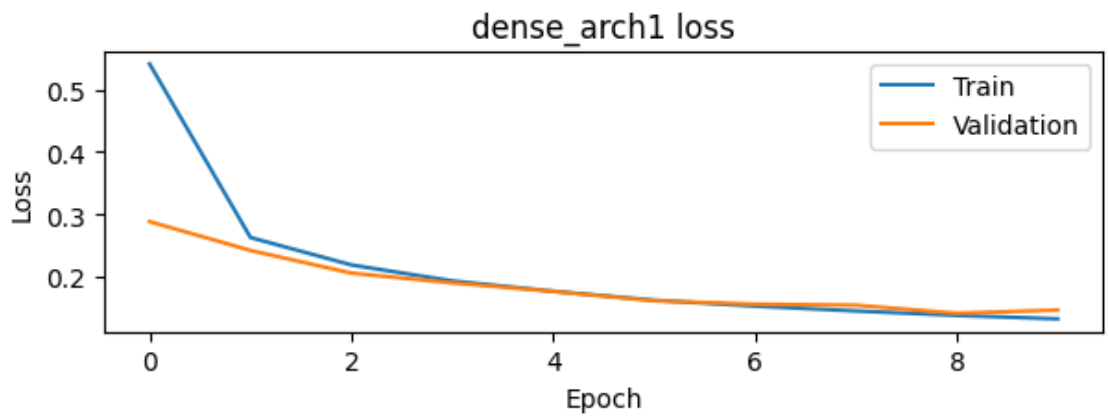
In [ ]:
```
hidden_layers = (100, 200)
```



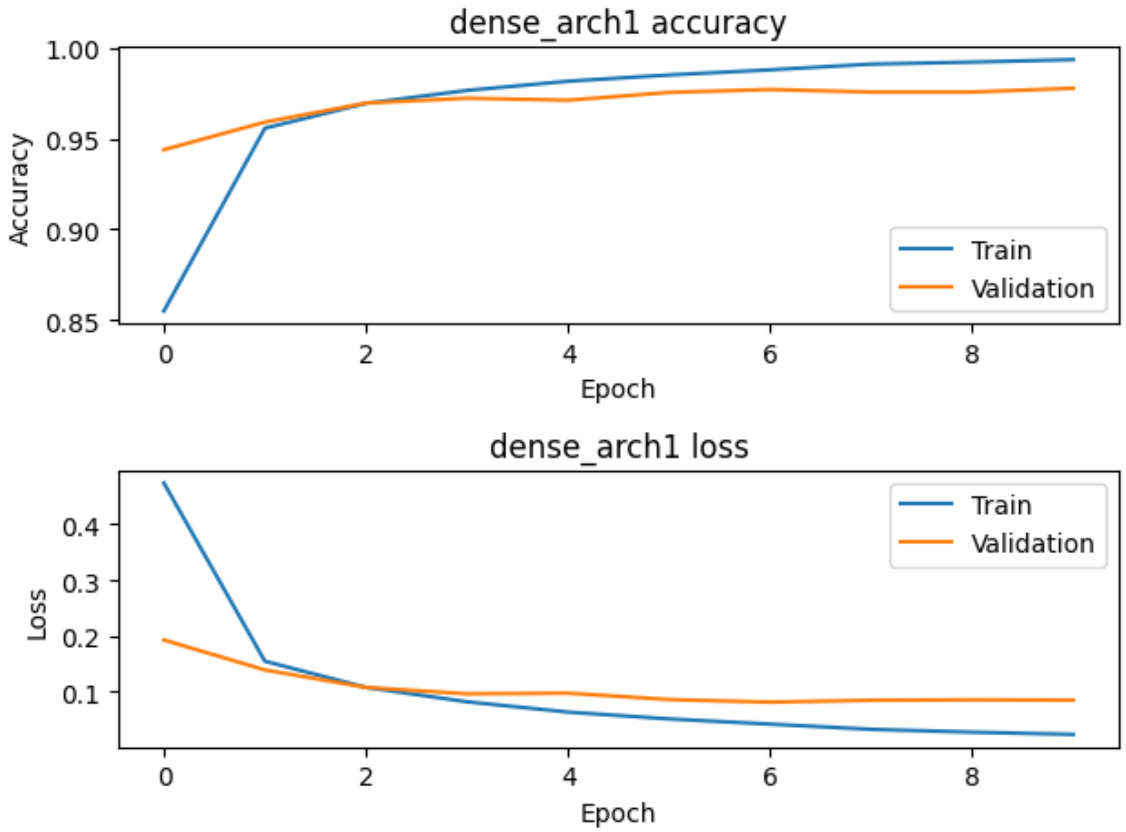


In [ ]:
```
hidden_layers = (10, 200)
```

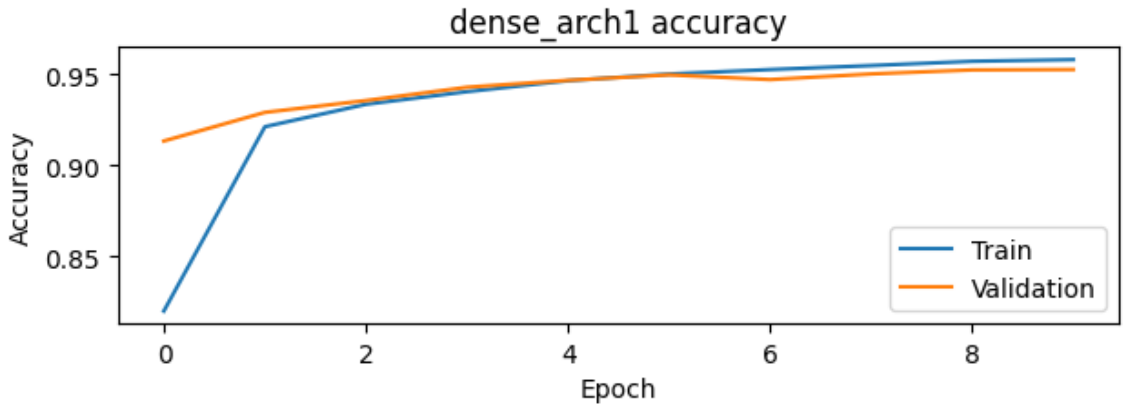


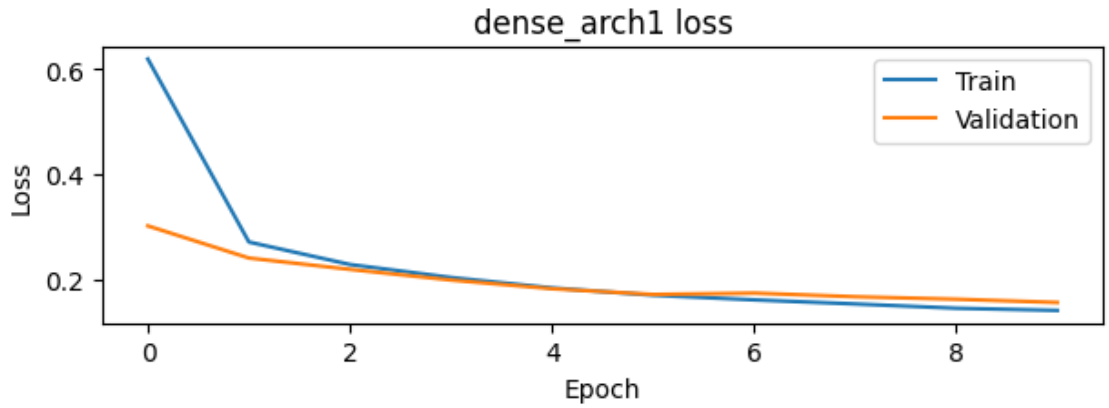


In [ ]:
```
hidden_layers = (200, 10)
```

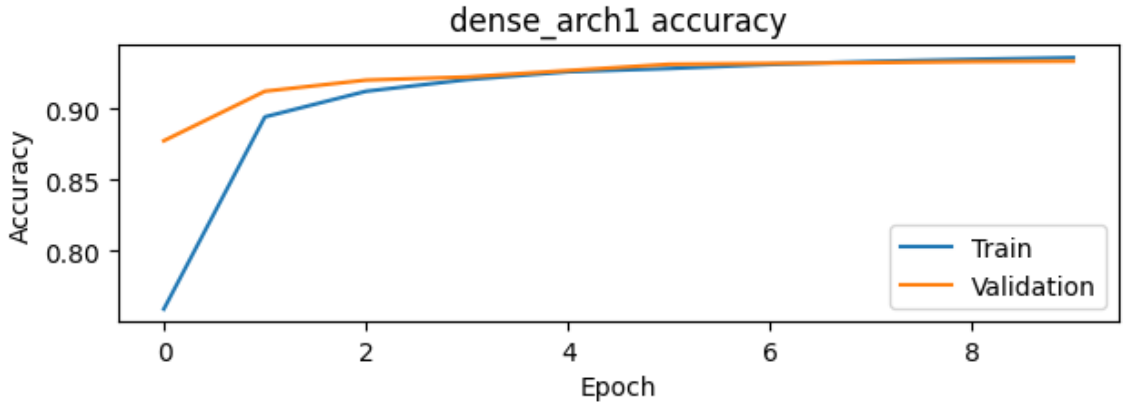


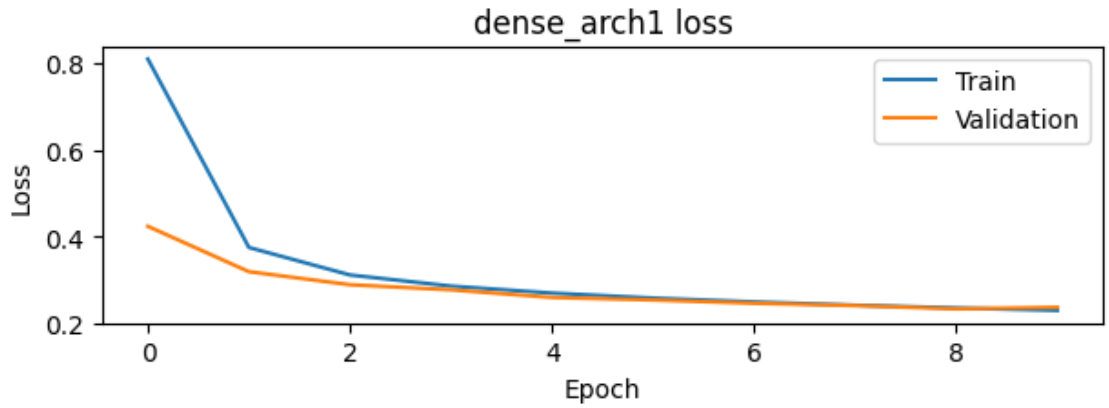
In [ ]:
```
hidden_layers = (100, 10)
```



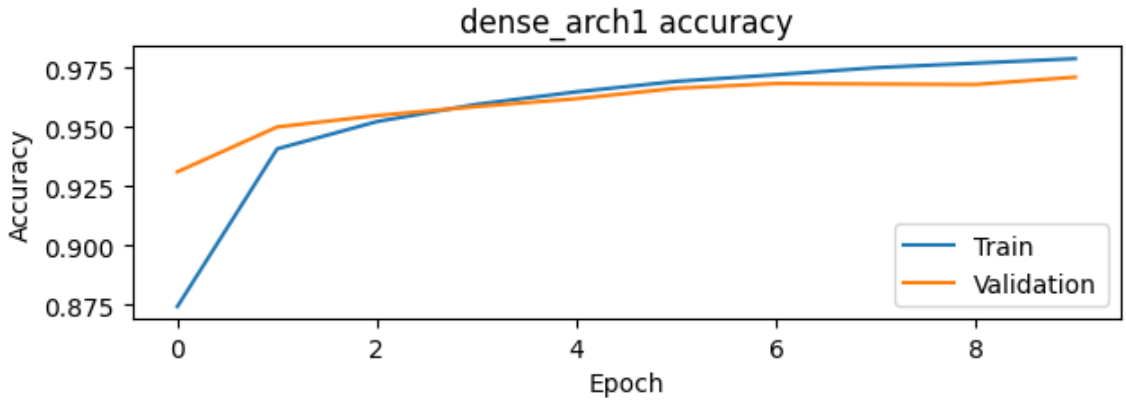


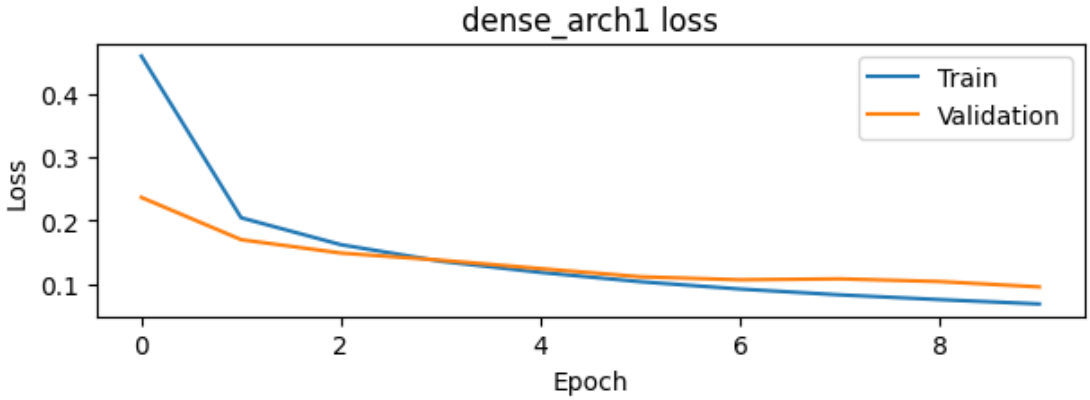
In [ ]:
```
hidden_layers = (10, 10)
```





In [ ]:
```
hidden_layers = (30, 100)
```





The general trends that are observed is that the training and validation accuracy goes down as the sum total of the hidden units goes down, which makes intuitive sense. Much more interestingly, the number of units in the first hidden layer has a much greater impact on the final training and validation accuracy than the number of units in the second hidden layers. This makes sense as the first hidden layers contributes much more to the number of model parameters than the second hidden layer. Within each experiment as well, regardless of the number of hidden units, the training accuracy and validation loss curve were greater than their counterparts, and though the starting point of each curve wildly varied based on the number of hidden units, after only 1 epoch, both curves converge and continue to do so as the number of epochs increases.

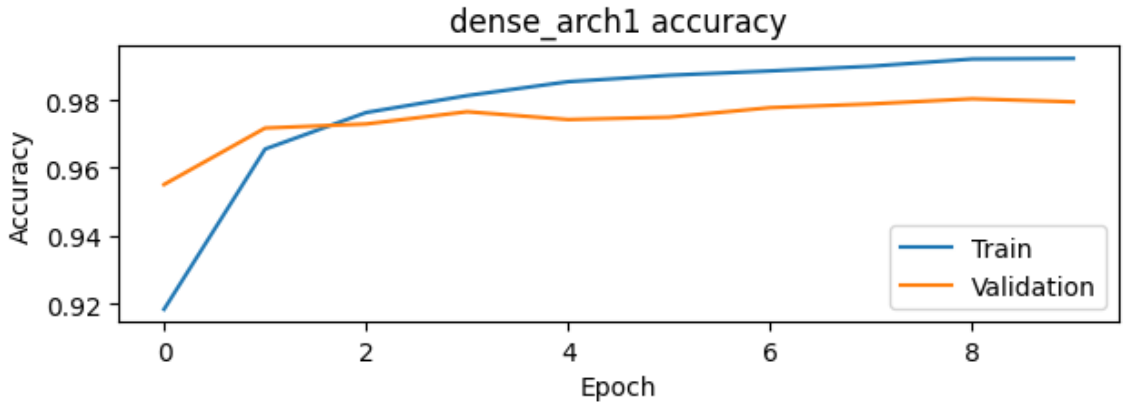
The last experiement shows that we need at minimum ~27000-28000 model parameters to achieve over a 97% validation accuracy.

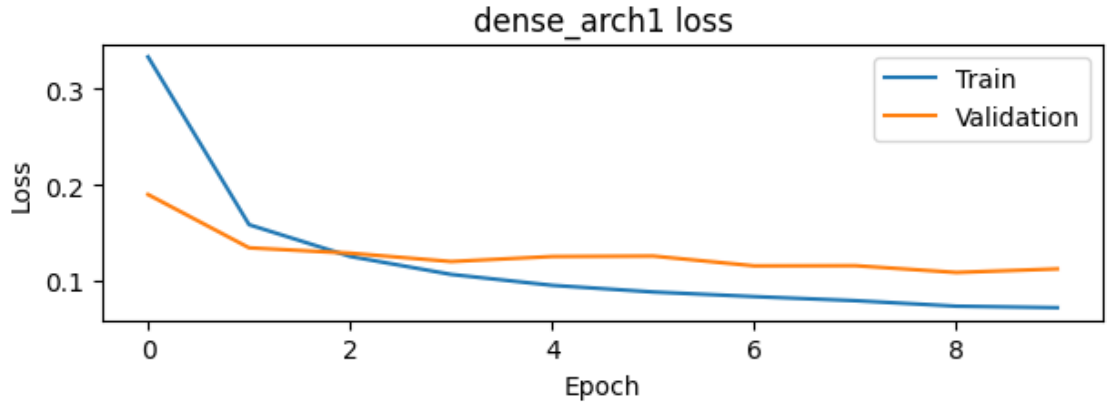
```
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_48 (Dense)            (None, 30)                23550

 activation_48 (Activation)  (None, 30)                0

 dense_49 (Dense)            (None, 100)               3100

 activation_49 (Activation)  (None, 100)               0

 dense_50 (Dense)            (None, 10)                1010

 activation_50 (Activation)  (None, 10)                0

=================================================================
Total params: 27,660
Trainable params: 27,660
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
469/469 [==============================] - 1s 3ms/step - loss: 0.4652 - accuracy: 0.8714 - val_loss: 0.2421 - val_accuracy: 0.9273
Epoch 2/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2170 - accuracy: 0.9385 - val_loss: 0.1846 - val_accuracy: 0.9444
...
Epoch 9/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0787 - accuracy: 0.9760 - val_loss: 0.1095 - val_accuracy: 0.9677
Epoch 10/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0715 - accuracy: 0.9779 - val_loss: 0.1006 - val_accuracy: 0.9701
```

==========

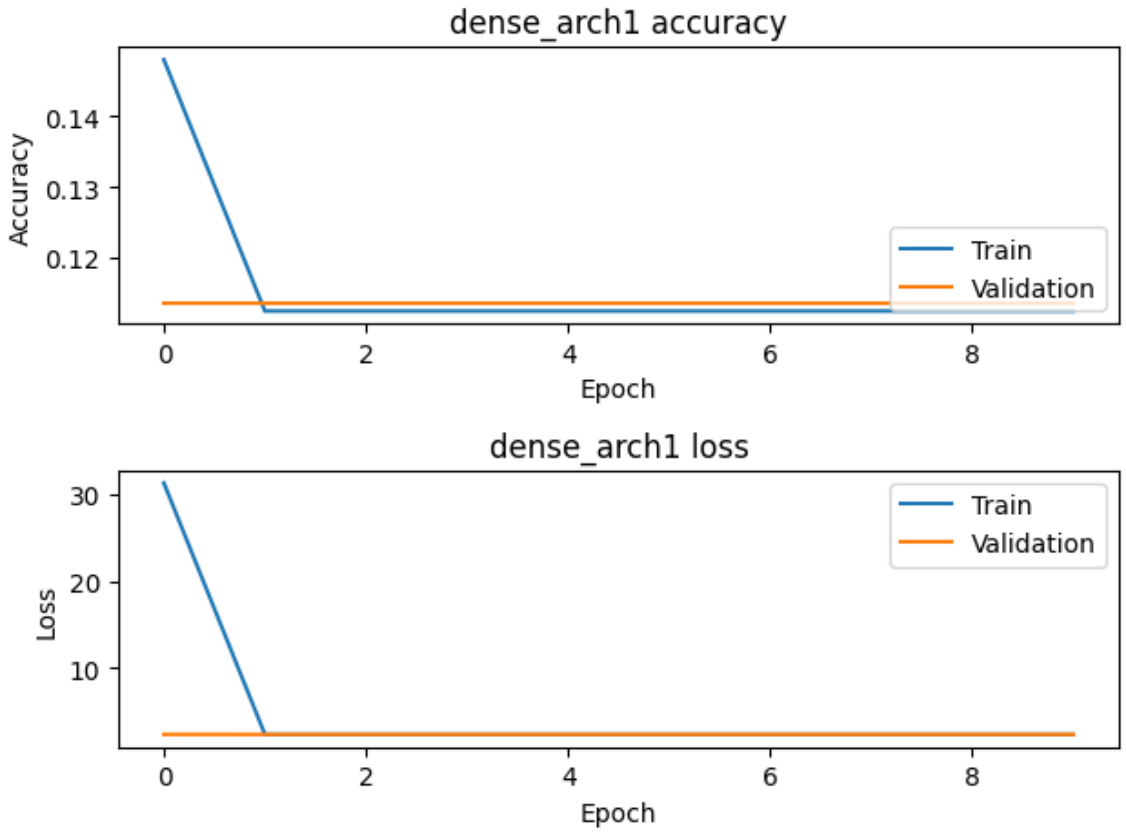
1.4

In [ ]:
```
lambda = 1e-4
```

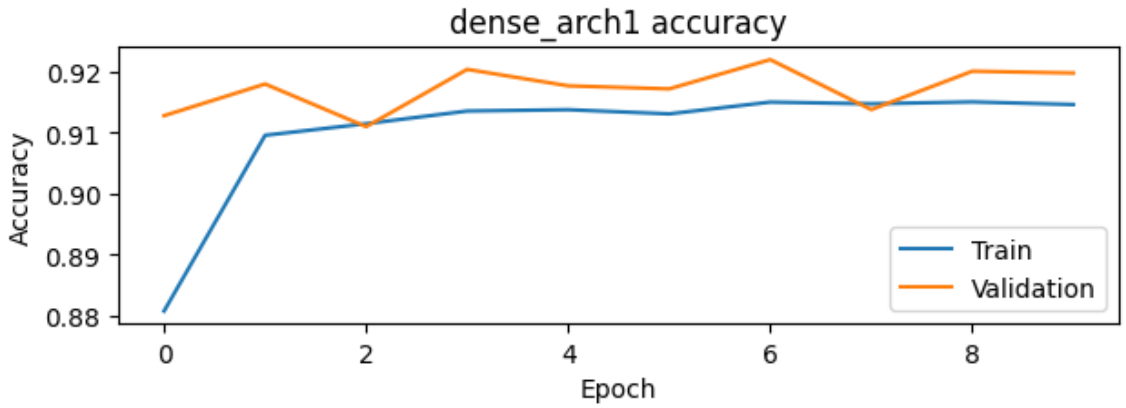


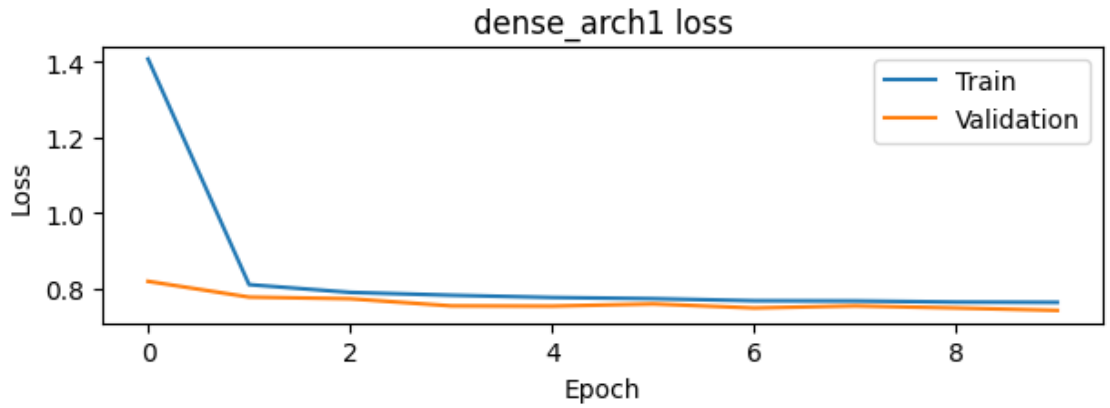


In [ ]:
```
lambda = 1
```

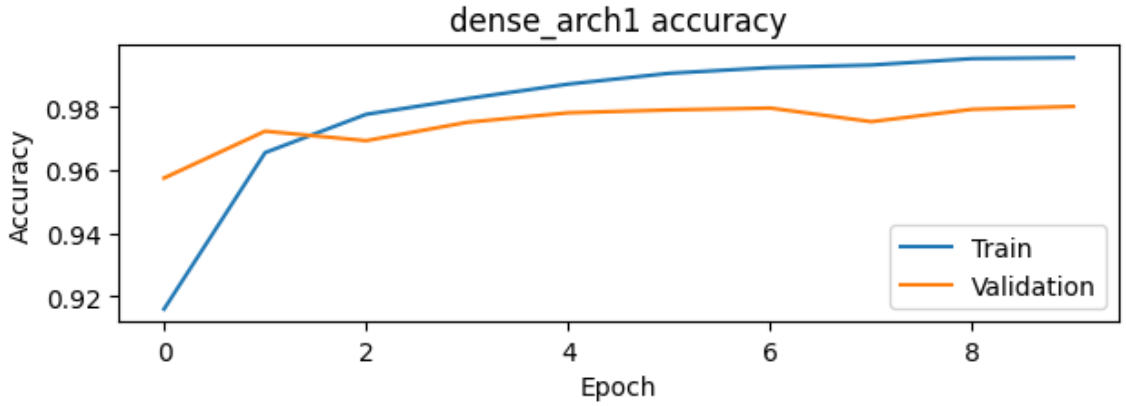


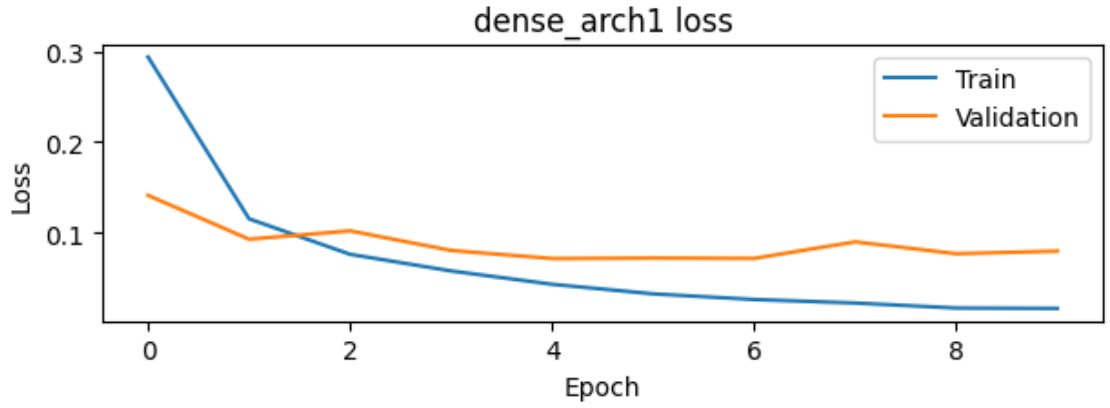
In [ ]:
```
lambda = 1e-2
```



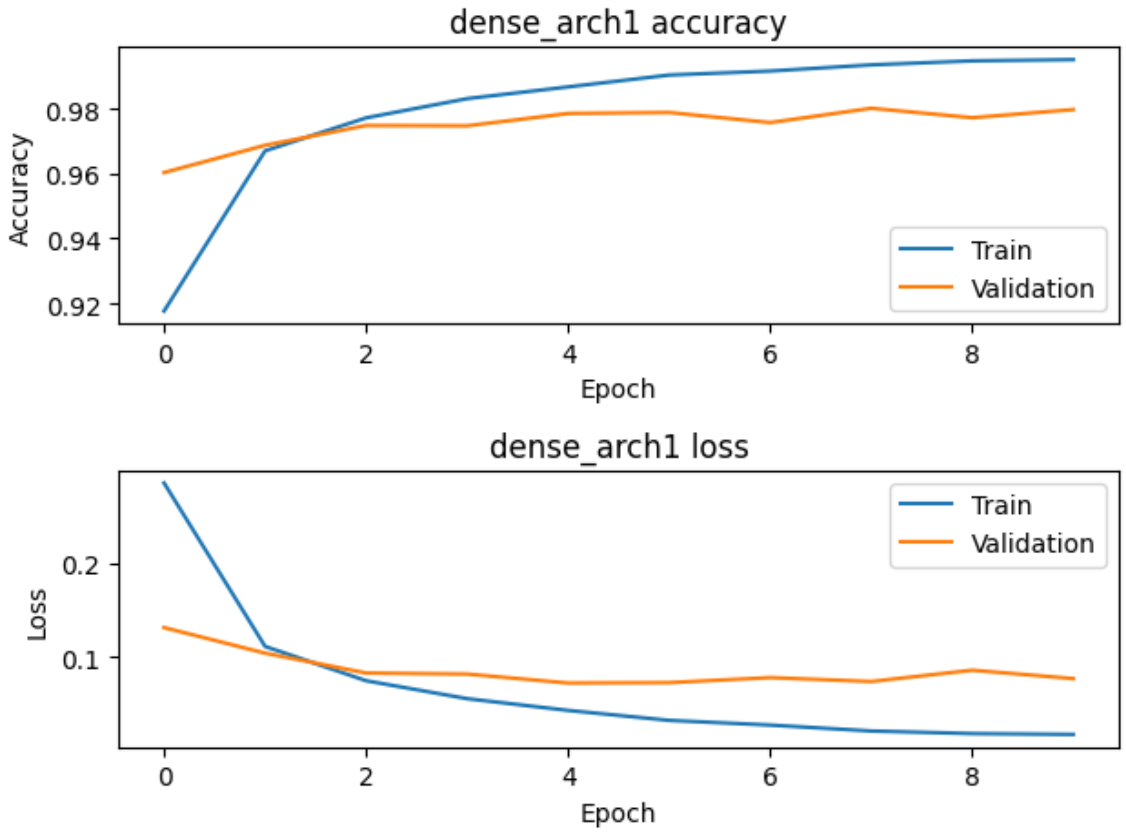


In [ ]:
```
lambda = 1e-6
```





In [ ]:
```
lambda = 1e-8
```

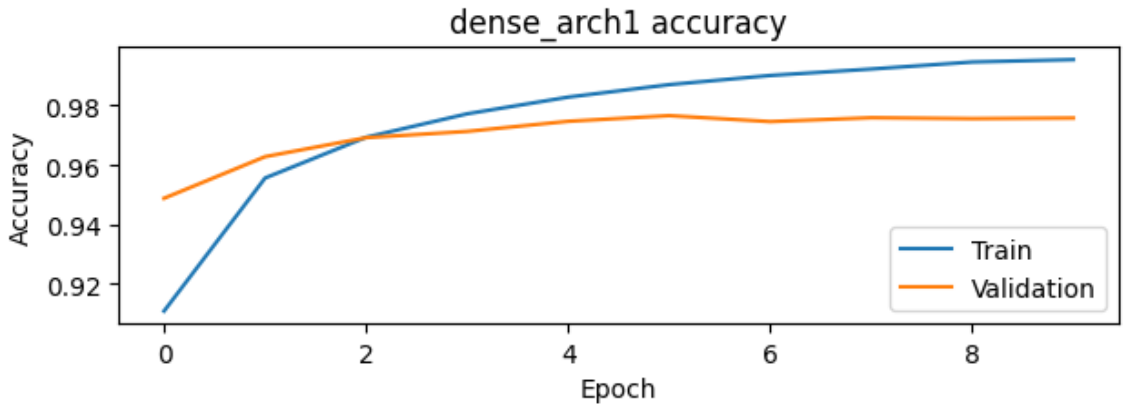


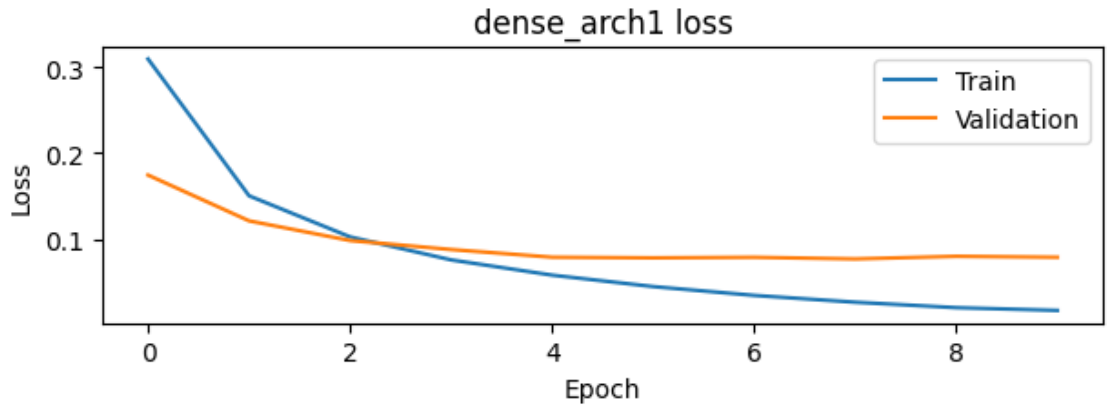
A lambda value of 1e-2 creates a learning curve that's similar to a smaller neural network (10 units in each hidden layer). This makes sense as a lambda value that's very small will lead to an overfit of the data leading to poor performance on the validation set and generalization to out of sample.

==========

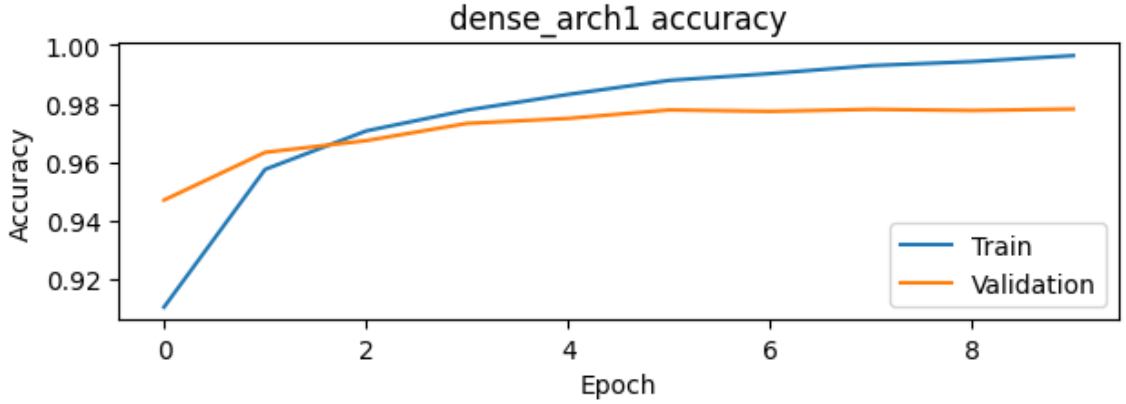
1.5

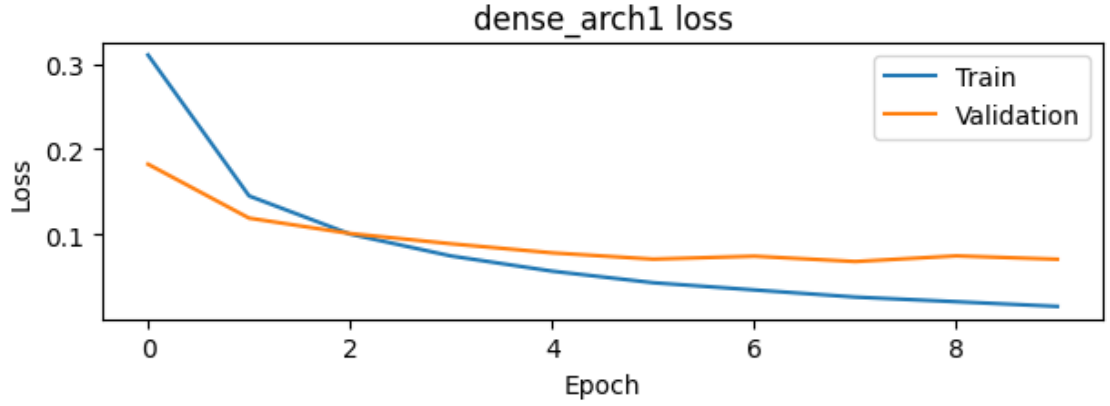
In [ ]:
```
hidden_layers = (200, 100)
lambda = 0
```



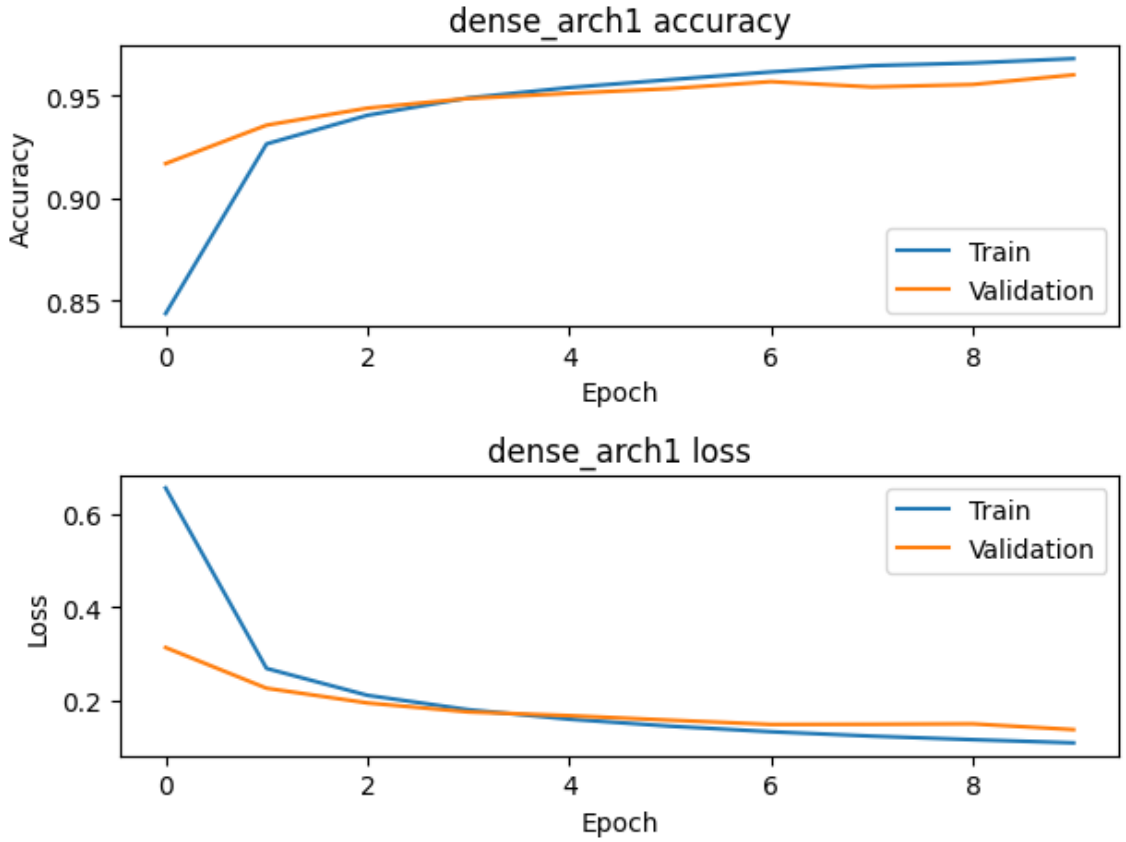


In [ ]:
```
hidden_layers = (200, 100)
lambda = 1e-8
```

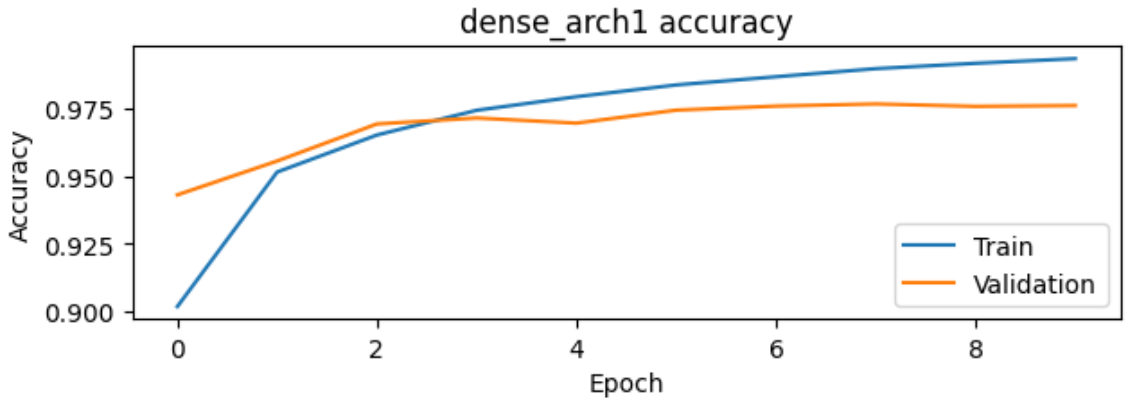


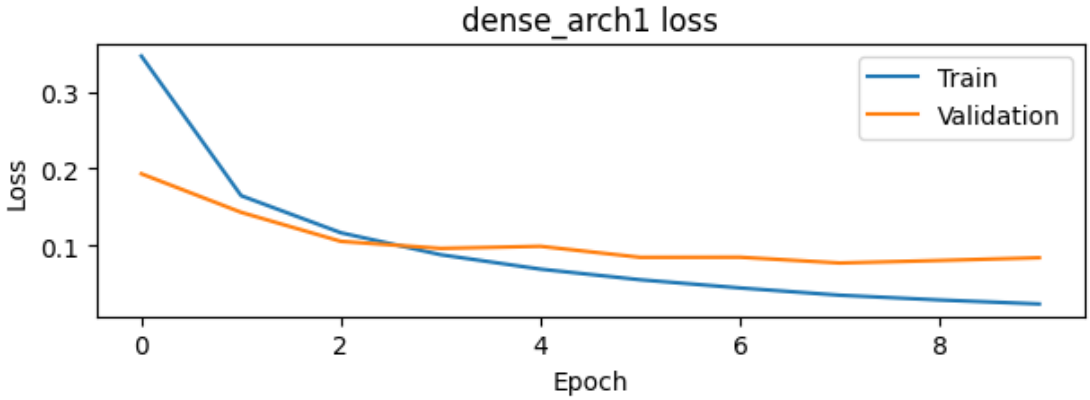


In [ ]:
```
hidden_layers = (100, 100)
lambda = 0
```



In [ ]:
```
hidden_layers = (100, 100)
lambda = 1e-8
```

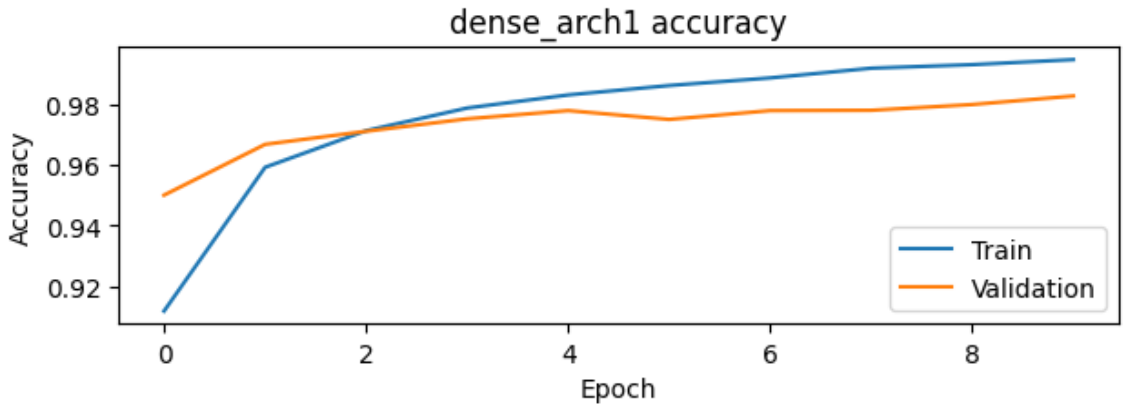


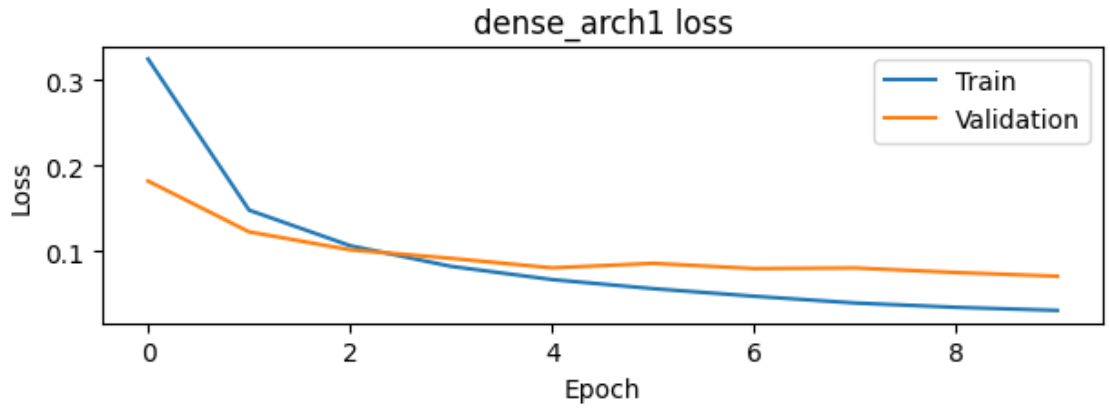


From the above experiments, we see that even if we vary the number of parameters and the regularization, it does take more epochs for convergence to occur. It used to occur around the first epoch, but tanh activation converges around the 3rd epoch. This does make sense because as s goes to infinity, tanh approaches 1, but ReLU approaches infinity. When the signal is very large, the output of tanh is limited so it will take a few more epochs for the model to learn around this constraint. With ReLU however, a positive signal can be as large as it needs to which allows more freedom for each unit in the hidden layers leading to a faster convergence. The linearity of ReLU and the constant derivative of it also accelerates the convergence of the gradient descent.

==========

1.6 - Different Architectures



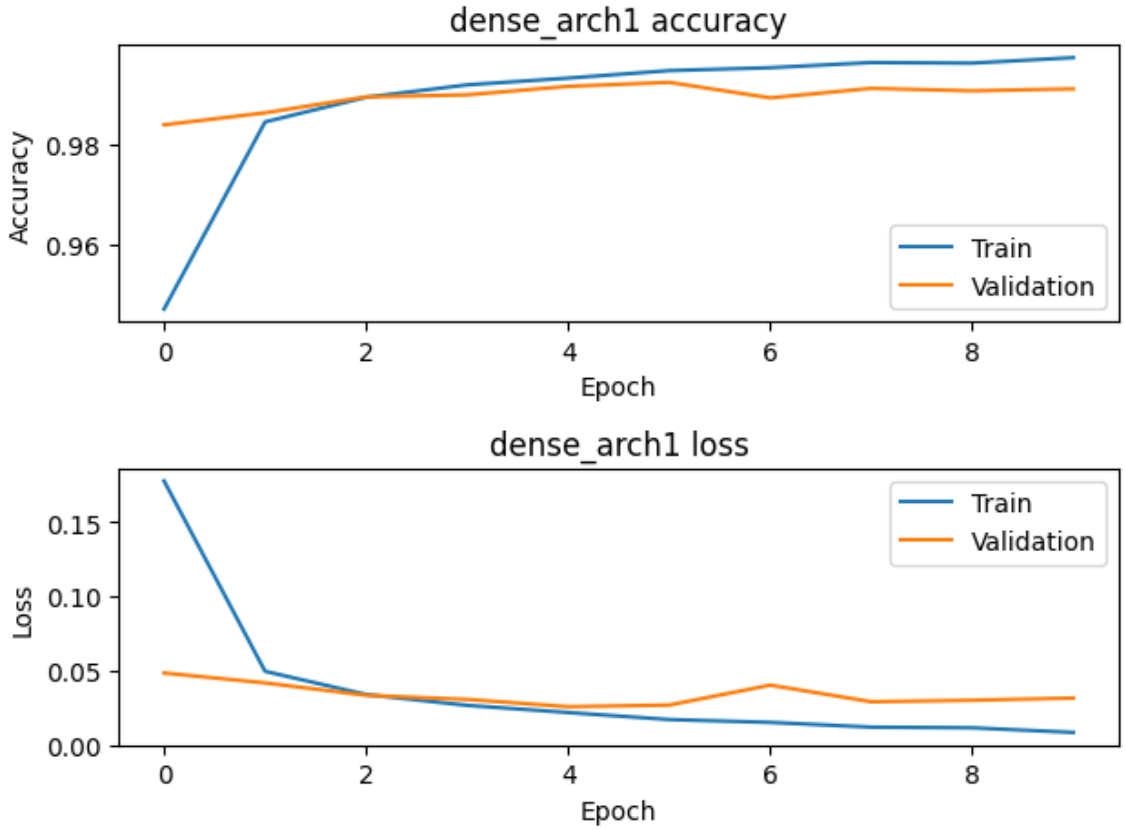



```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_36 (Dense)            (None, 200)               157000

 activation_36 (Activation)  (None, 200)               0

 dense_37 (Dense)            (None, 10)                2010

 activation_37 (Activation)  (None, 10)                0

=================================================================
Total params: 159,010
Trainable params: 159,010
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
469/469 [==============================] - 2s 3ms/step - loss: 0.3248 - accuracy: 0.9119 - val_loss: 0.1818 - val_accuracy: 0.9500
Epoch 2/10
469/469 [==============================] - 1s 3ms/step - loss: 0.1477 - accuracy: 0.9592 - val_loss: 0.1222 - val_accuracy: 0.9667
Epoch 3/10
469/469 [==============================] - 1s 3ms/step - loss: 0.1060 - accuracy: 0.9712 - val_loss: 0.1012 - val_accuracy: 0.9710
Epoch 4/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0817 - accuracy: 0.9786 - val_loss: 0.0912 - val_accuracy: 0.9751
...
Epoch 9/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0338 - accuracy: 0.9929 - val_loss: 0.0746 - val_accuracy: 0.9798
Epoch 10/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0302 - accuracy: 0.9946 - val_loss: 0.0702 - val_accuracy: 0.9826
```

I was able to get a validation accuracy of 0.9826 with one hidden layer with 200 units, l1 regularization, ReLU activation, adam optimizer, and a regularization strenght of 1e-6.

==========

1.7 - Convolutional Neural Networks


dense_arch1 accuracy
Accuracy
0.98
0.96
Train
Validation
0
2
4
6
8
Epoch
dense_arch1 loss
Loss
0.15
0.10
0.05
0.00
Train
Validation
0
2
4
6
8
Epoch

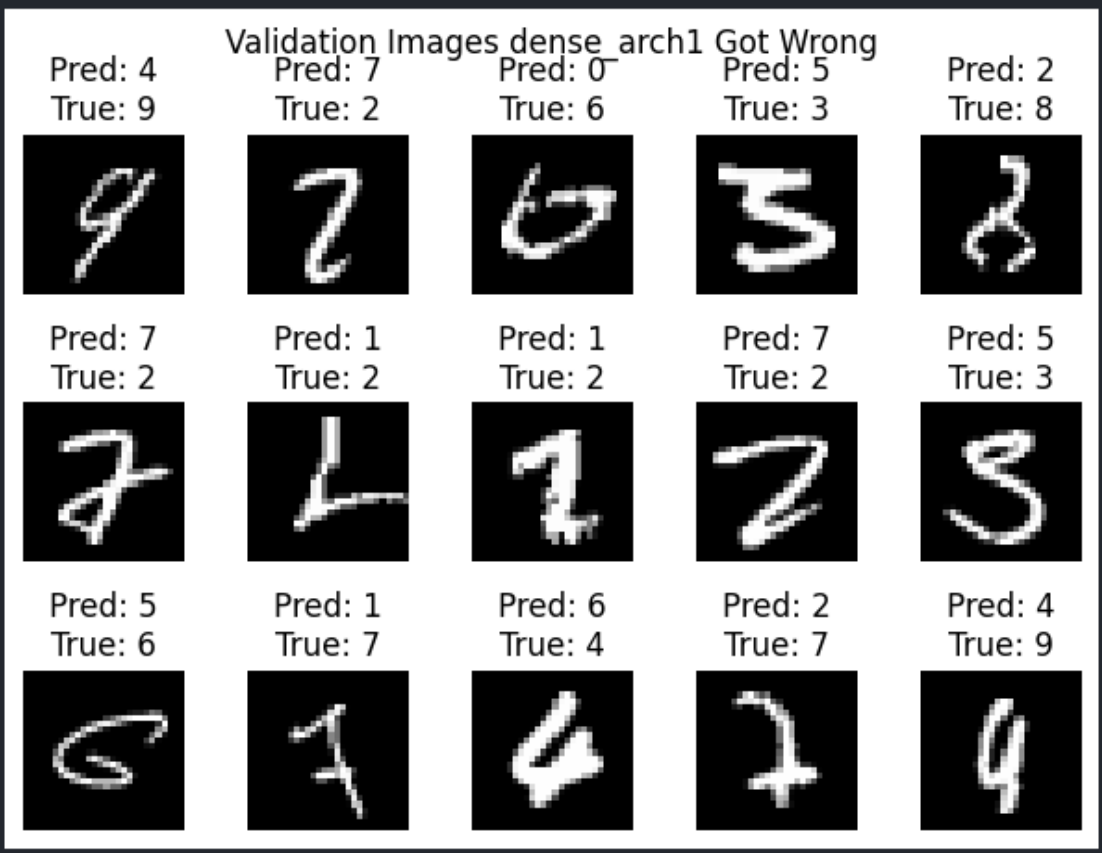
```
Model: "sequential_25"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 200)       5200

 activation_64 (Activation)  (None, 28, 28, 200)       0

 max_pooling2d (MaxPooling2D  (None, 14, 14, 200)      0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 100)       180100

 activation_65 (Activation)  (None, 14, 14, 100)       0

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 100)        0
 2D)

 flatten (Flatten)           (None, 4900)              0

 dense_64 (Dense)            (None, 10)                49010

 activation_66 (Activation)  (None, 10)                0

=================================================================
Total params: 234,310
Trainable params: 234,310
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
469/469 [==============================] - 105s 224ms/step - loss: 0.1773 - accuracy: 0.9470 - val_loss: 0.0483 - val_accuracy: 0.9839
Epoch 2/10
469/469 [==============================] - 109s 231ms/step - loss: 0.0494 - accuracy: 0.9845 - val_loss: 0.0416 - val_accuracy: 0.9863
Epoch 3/10
469/469 [==============================] - 123s 262ms/step - loss: 0.0338 - accuracy: 0.9894 - val_loss: 0.0333 - val_accuracy: 0.9895
Epoch 4/10
469/469 [==============================] - 128s 272ms/step - loss: 0.0265 - accuracy: 0.9919 - val_loss: 0.0305 - val_accuracy: 0.9899
Epoch 5/10
469/469 [==============================] - 127s 271ms/step - loss: 0.0217 - accuracy: 0.9933 - val_loss: 0.0256 - val_accuracy: 0.9916
Epoch 6/10
469/469 [==============================] - 132s 281ms/step - loss: 0.0170 - accuracy: 0.9948 - val_loss: 0.0267 - val_accuracy: 0.9924
Epoch 7/10
469/469 [==============================] - 124s 265ms/step - loss: 0.0151 - accuracy: 0.9953 - val_loss: 0.0401 - val_accuracy: 0.9893
Epoch 8/10
469/469 [==============================] - 74s 158ms/step - loss: 0.0119 - accuracy: 0.9964 - val_loss: 0.0290 - val_accuracy: 0.9912
Epoch 9/10
469/469 [==============================] - 74s 158ms/step - loss: 0.0115 - accuracy: 0.9962 - val_loss: 0.0300 - val_accuracy: 0.9907
Epoch 10/10
469/469 [==============================] - 75s 160ms/step - loss: 0.0084 - accuracy: 0.9973 - val_loss: 0.0314 - val_accuracy: 0.9911
```

The number of parameters increased greatly from ~178,000 model parameteres in the dense net to ~234,000 parameters in the convolutional net. The learning curves partially reflect this change as well as the validation accuracy is much higher than that of a dense network ane the training accuracy is almost 1. We also see from the evaluation script that the images it misclassified are numbers that qualitatively we as humans wouldn't be able to place very well either e.g. first image in the second row. The in-sample error is less than 1% and the

validation error is only 3%. This means the model will generalize very well to out of sample points.

```
Training loss: 0.004533630795776844
Training accuracy: 0.9986333250999451

Validation loss: 0.03143271058797836
Validation accuracy: 0.991100013256073
313/313 [==============================] - 5s 15ms/step
```



==========

2.2 - How does a GAN work?

Math is on separate PDF

==========

2.4 - Reflecting on GAN behavior

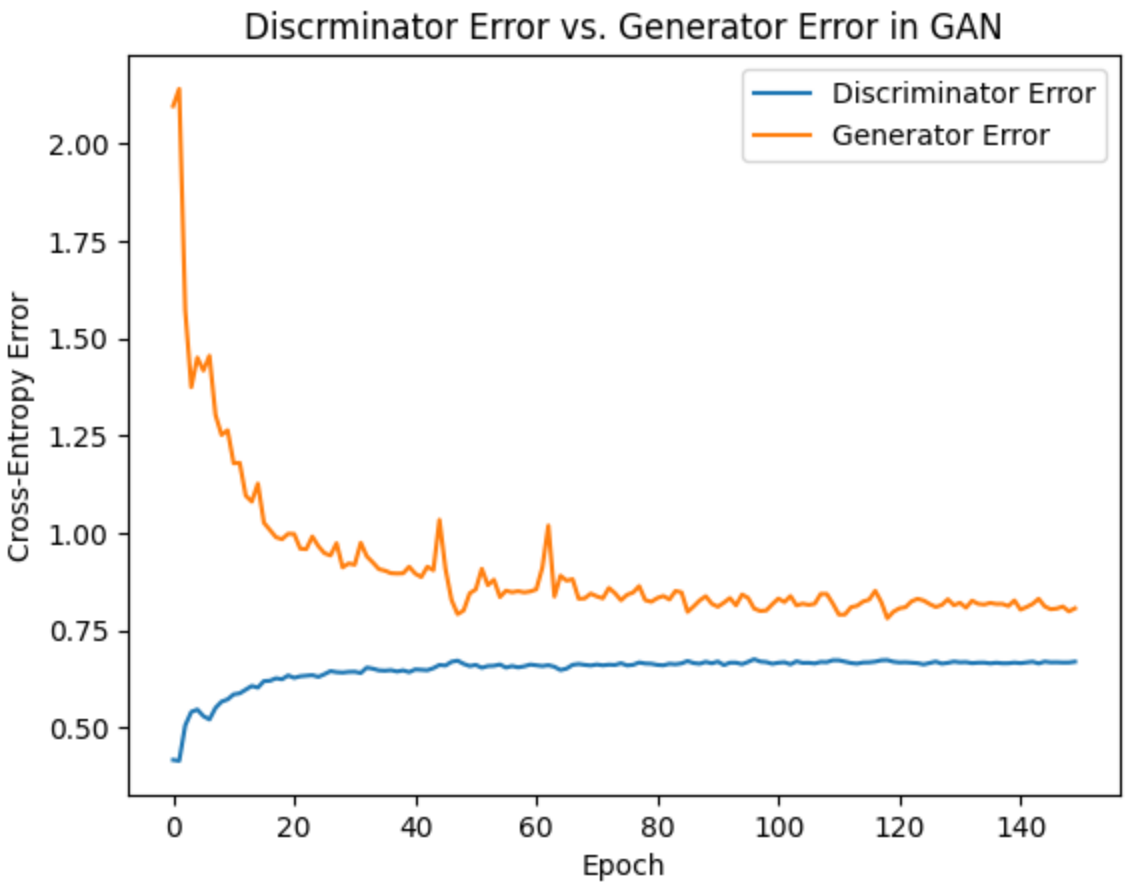
In [ ]:
```
import matplotlib.pyplot as plt
```

In [ ]:
```
discriminator_error = []
generator_error = []
```

In [ ]:
```
with open("output.txt", "r", encoding='utf-8') as f:
    for line in f:
        if line.startswith("Discriminator"):
            error = float(line.split()[2])
            discriminator_error.append(error)

        elif line.startswith("Generator"):
            error = float(line.split()[2])
            generator_error.append(error)
```

In [ ]:
```
fig, ax = plt.subplots()

ax.plot(discriminator_error, label='Discriminator Error')
ax.plot(generator_error, label='Generator Error')
ax.set_title("Discrminator Error vs. Generator Error in GAN")
ax.set_xlabel("Epoch")
ax.set_ylabel("Cross-Entropy Error")
ax.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x1af839a4f40>
```



By looking through the images, we see pieces of numbers slowly start to form as the number of epochs increases. The generator model is learning how to generate these numbers by learning how to generate different parts of them and then putting them together. It figures

this out by gathering 'feedback' from the discriminator and backpropagating the error to improve performance. From the plot above, we see the generator loss starts quite large as the generator can't generate very many images to fool the discriminator which leads to a low discriminator loss as well. As the epochs progress though, we see the generator model improve dramatically at first and then make incrementally smaller changes. Consequently, we see the discriminator error rise as well. Both curves seem that they will converge at an error of 0.75 if trained for more epochs. This would mean the generator is able to generate images that the discriminator is no longer able to distinguish from the real data.

Mode collapse occurs when a GAN produced a limited diversity of samples and doesn't capture the full variation of the training data. This is problematic because mode collapose greatly decreases the quality of the generated data. In the context of our GAN being trained with MNIST, it may only generate realistic images for a few digits like 2, 4, and/or 7 and struggle with generating realistic images for the other digits. Through skimming the code, note that there is no regularization applied on the kernel in the dense layers as with the previous part. Regularization helps prevent mode collapse by preventing overfitting and the GAN from memorizing specific patterns. Since there is no regularization applies, the generator's implementation did not avoid mode collapse.

==========

3 - Feedback

I throughly enjoyed every aspect of this project. Beinga able to explore the current tools being used in machine learning and being able to apply the theory we've been learning in class has been a very fulfilling experience. The only minor change I would propose to this project is to provide a more reliable and less "hacky" default way to train the GAN. Google Colab is a fine service, but it can be quite finnicky with the free tier, so investigating more reliable alternatives would be worthwhile. No parts of the assignment were repetive, and I ended up using the IPython notebook format to utilize the ability to run different cells of code. It allowed me to run more experiments and explore different architectures, so I throughly appreciated the IPython notebook files being provided.