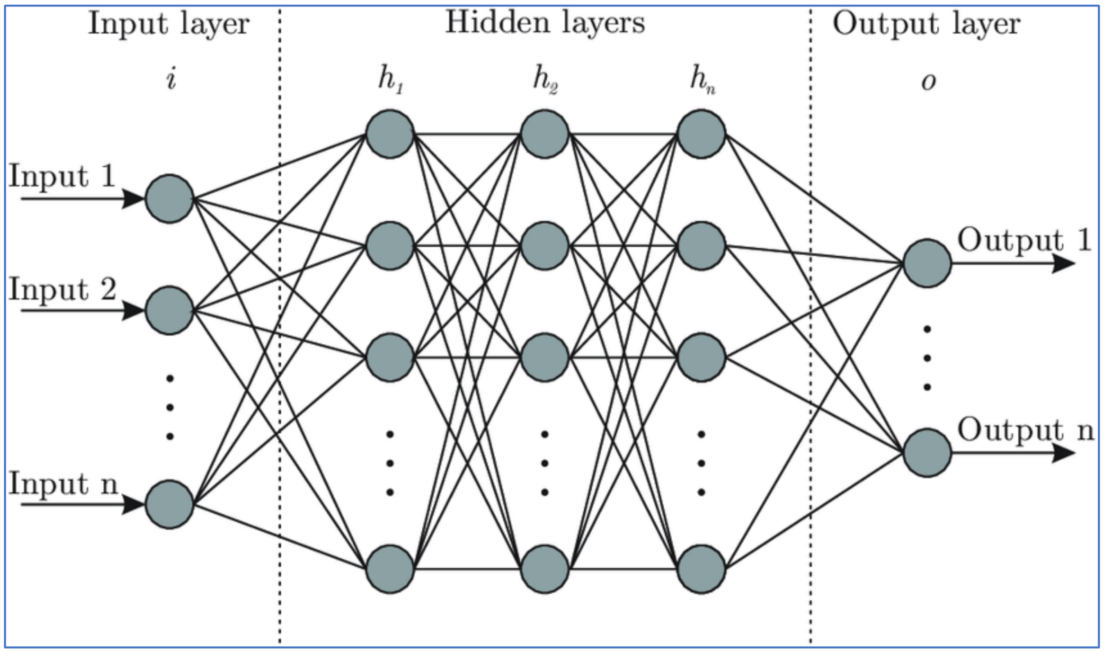


Experiments for Artificial Neural Network Forecasting

DATA SCIENCE IN POWER ENGINEERING

ALEJANDRO VILLARREAL CABALLERO

DR. MICHAŁ GUZEK

WARSAW UNIVERSITY OF TECHNOLOGY

Table of Contents

Introduction 3
Results original architecture 3
Experiment 1, GPU vs CPU 6
Results GPU calculations 6
Speed of CPU vs GPU 9
Experiment 2, SGD vs Adam 11
Results Adam optimizer 11
Experiment 3, Additional hidden layer 14
Results additional hidden layer 14
Conclusion 17

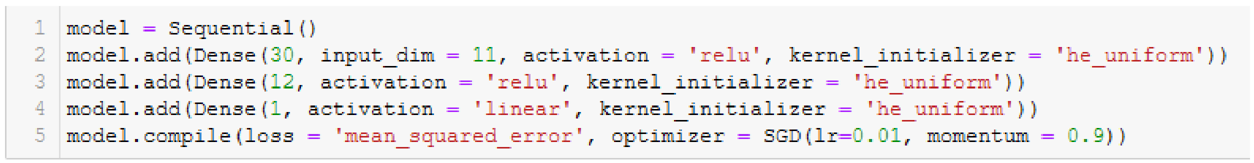
List of Figures

Figure 1 Original ANN Architecture 3
Figure 2 Original ANN model training 3
Figure 3 Original architecture, Prediction vs Output 4
Figure 4 Original architecture, Prediction vs Output, 200 rows 5
Figure 5 Original architecture, total execution time and final values 6
Figure 6 Original architecture, losses 6
Figure 7 GPU, Prediction vs Output 7
Figure 8 GPU, Prediction vs Output, 200 rows 8
Figure 9 GPU, total execution time and final values 9
Figure 10 GPU, losses 9
Figure 11 Neuron-modified architecture 9
Figure 12 Modified epochs 10
Figure 13 CPU, total execution time 2000 neurons 10
Figure 14 GPU, total execution time 2000 neurons 10
Figure 15 Optimizer-modified architecture 11
Figure 16 Adam, Prediction vs Output 12
Figure 17 Adam, Prediction vs Output, 200 rows 13
Figure 18 Adam, total execution time and final values 14
Figure 19 Adam, losses 14
Figure 20 Modified architecture 14
Figure 21 Additional hidden layer, Prediction vs Output 15
Figure 22 Additional hidden layer, Prediction vs Output, 200 rows 16
Figure 23 Additional hidden layer, total execution time and final values 17
Figure 24 Additional hidden layer, losses 17

Introduction

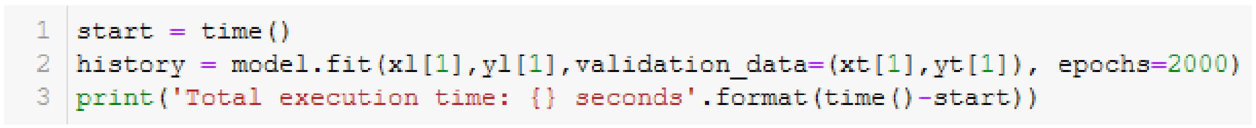
This activity will consist in modifying the properties of a control ANN architecture and then compare its predictions with the modified architectures. Three different experiments will be made to try and increase the overall performance of the ANN. This will be done by trying to increase the accuracy of the predictions and by keeping total execution time at a minimum.

Figure 1 Original ANN Architecture



```
model = Sequential()
model.add(Dense(30, input_dim = 11, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(12, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(1, activation = 'linear', kernel_initializer = 'he_uniform'))
model.compile(loss = 'mean_squared_error', optimizer = SGD(lr=0.01, momentum = 0.9))
```

Figure 2 Original ANN model training



```
start = time()
history = model.fit(xl[1],yl[1],validation_data=(xt[1],yt[1]), epochs=2000)
print('Total execution time: {} seconds'.format(time()-start))
```

Results original architecture

Figure 3 is the series of plots comparing the predicted output after training the ANN using data set Xl1 and output Yl1 while using Xt1 and Yt1 as validation data. It can be observed that the prediction for Yl1 appears to be the most accurate out of all the data sets. Figure 4 provides a clearer view of the predictions by just plotting the first 200 rows. Figure 5 indicates the total execution time for 2000 epochs, using the CPU takes 531.76 seconds, which is almost 9 minutes. Figure 6 shows the losses and validation losses as a function of the epochs, as expected the losses reduce as the epochs increase. Notice, however, how the validation losses start to increase after approximately 500 epochs which indicates the model is slightly overfitted.

Figure 3 Original architecture, Prediction vs Output

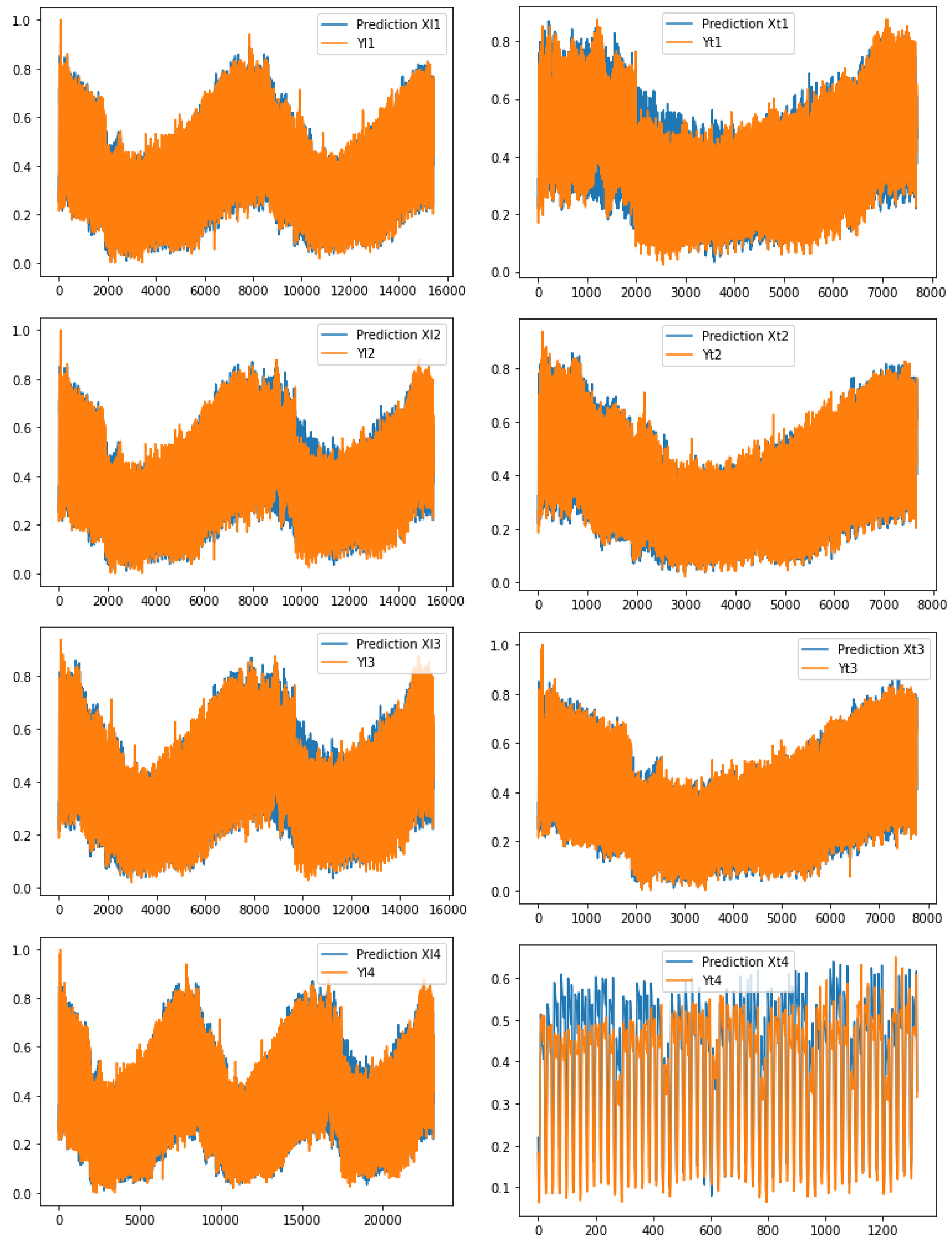


Figure 4 Original architecture, Prediction vs Output, 200 rows

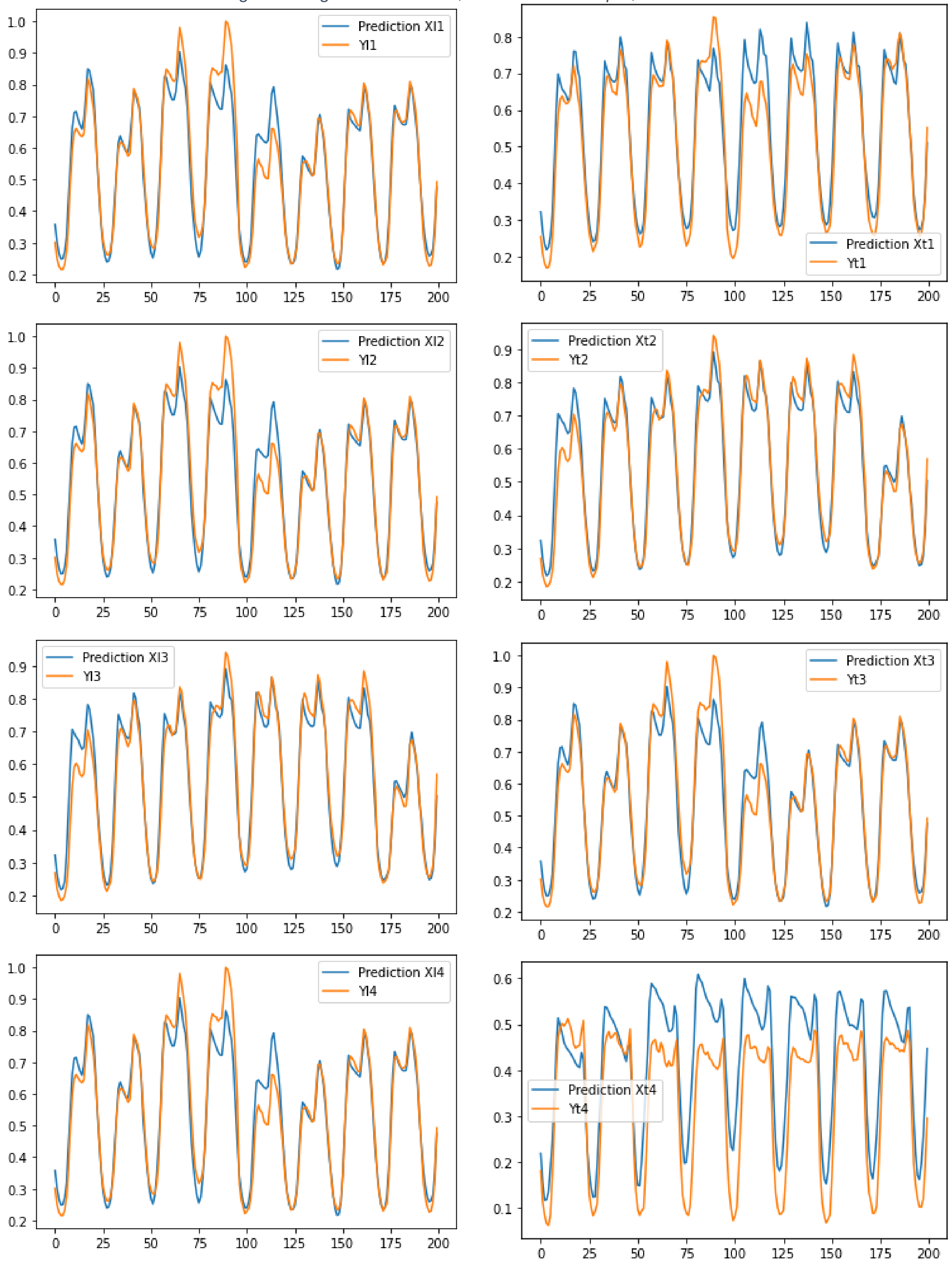
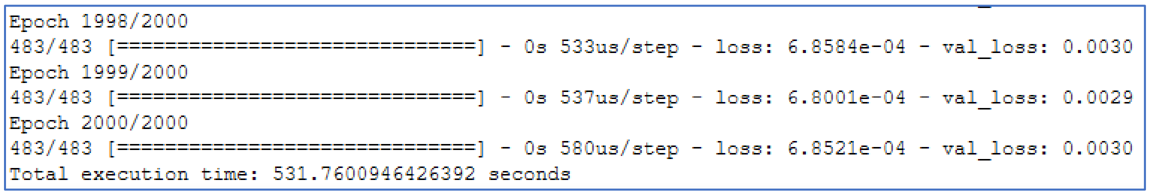
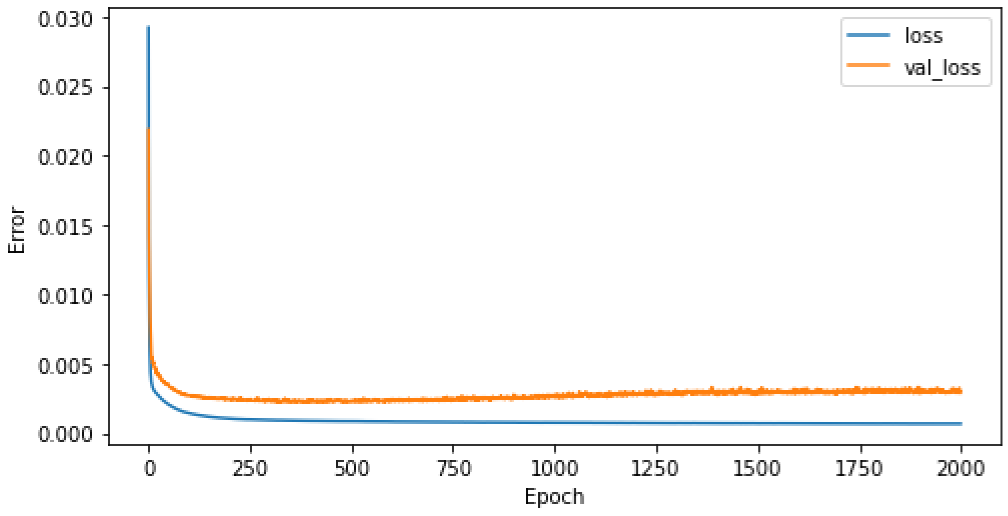


Figure 5 Original architecture, total execution time and final values



```
Epoch 1998/2000
483/483 [==============================] - 0s 533us/step - loss: 6.8584e-04 - val_loss: 0.0030
Epoch 1999/2000
483/483 [==============================] - 0s 537us/step - loss: 6.8001e-04 - val_loss: 0.0029
Epoch 2000/2000
483/483 [==============================] - 0s 580us/step - loss: 6.8521e-04 - val_loss: 0.0030
Total execution time: 531.7600946426392 seconds
```

Figure 6 Original architecture, losses



Experiment 1, GPU vs CPU

The first experiment consists in comparing the results and speed of using a CPU and GPU for the original architecture. To use the GPU, a new environment was created where “tensorflow-gpu” is downloaded instead of “tensorflow”.

Results GPU calculations

Figure 7 is the series of plots comparing the predicted output after training the ANN using a GPU. Similarly, Figure 8 provides a clearer view of the predictions by just plotting the first 200 rows. Figure 9 indicates the total execution time for 2000 epochs, using the GPU takes 1209.43 seconds, which is around 20 minutes and at least twice more than the CPU. Figure 10 shows validation losses remain constant instead of increasing after approximately 500 iterations.

Figure 7 GPU, Prediction vs Output

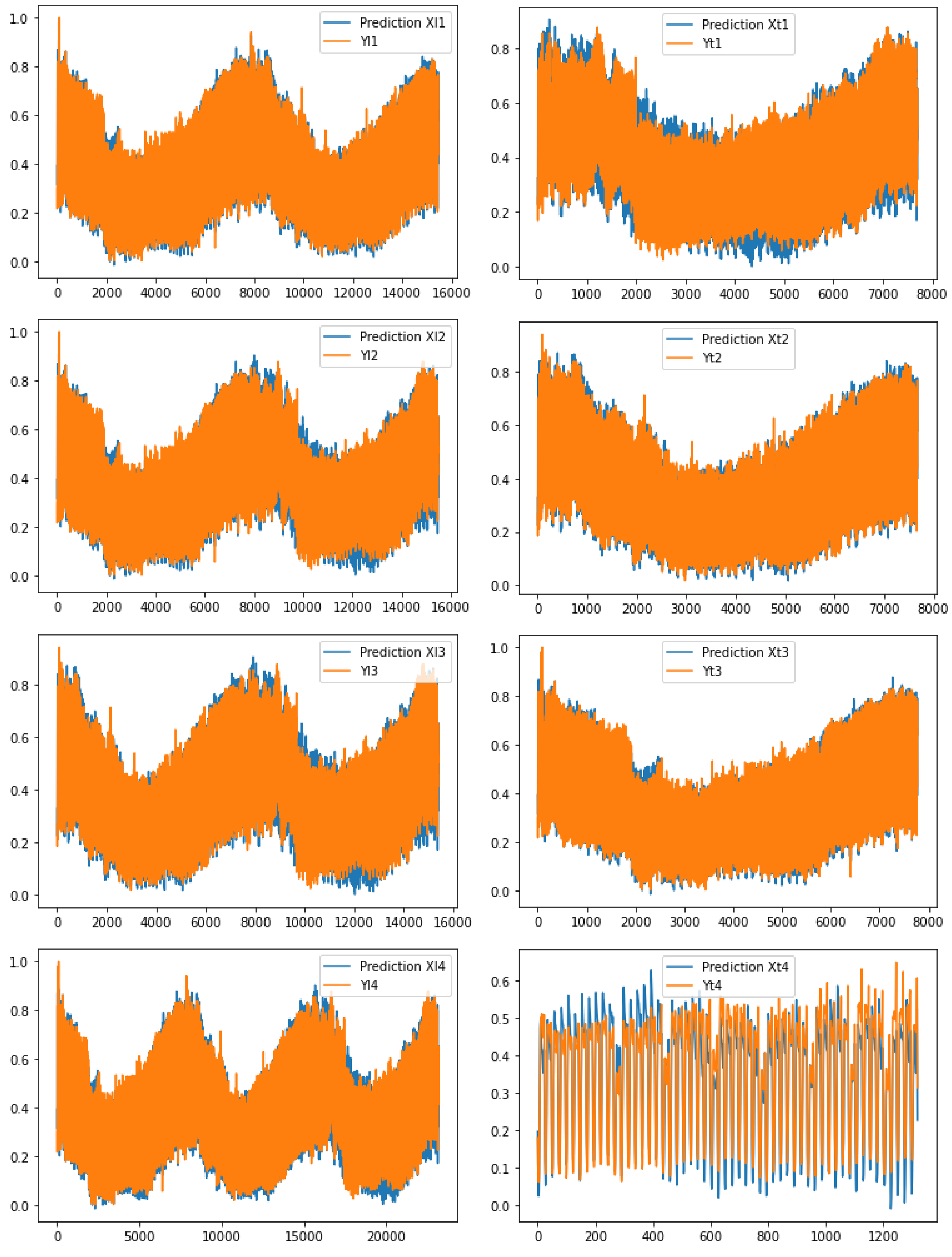


Figure 8 GPU, Prediction vs Output, 200 rows

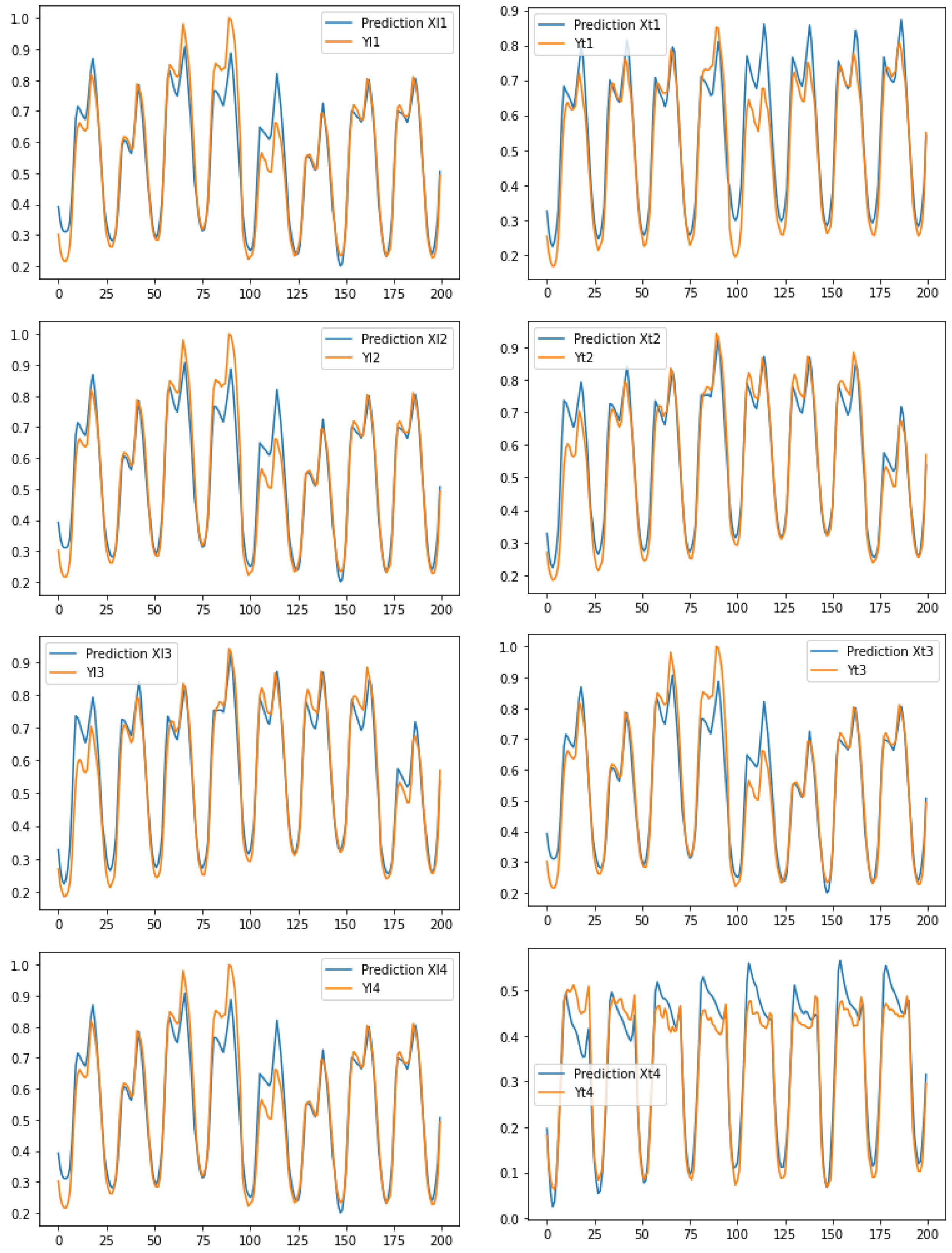
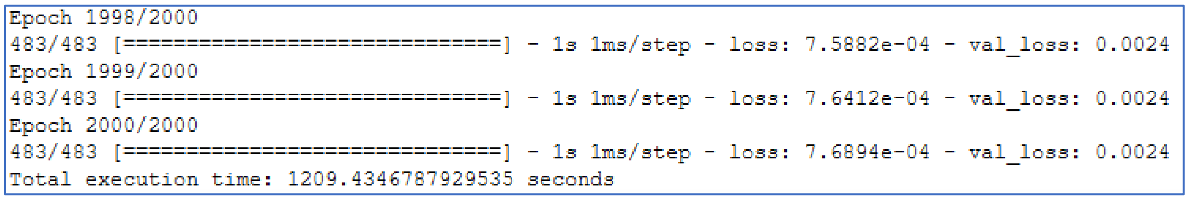
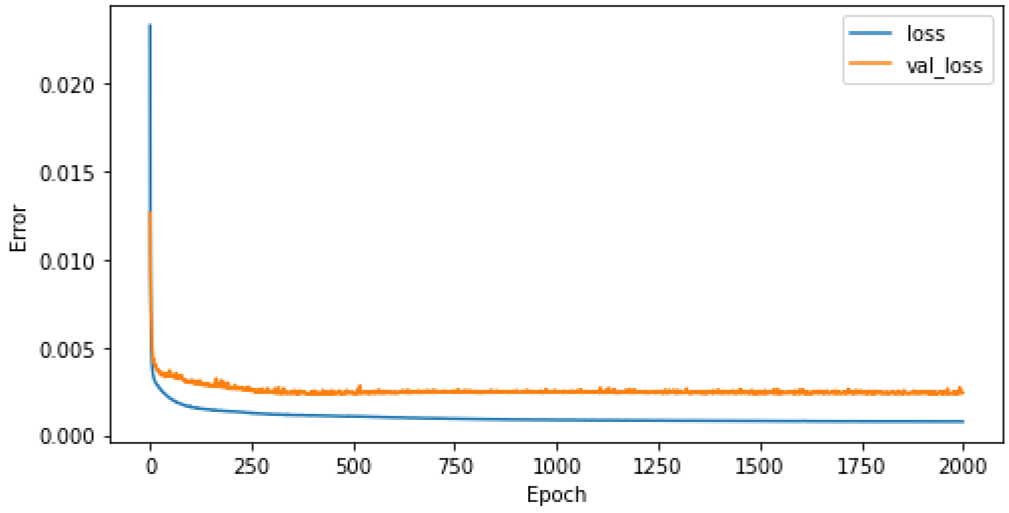


Figure 9 GPU, total execution time and final values



```
Epoch 1998/2000
483/483 [==============================] - 1s 1ms/step - loss: 7.5882e-04 - val_loss: 0.0024
Epoch 1999/2000
483/483 [==============================] - 1s 1ms/step - loss: 7.6412e-04 - val_loss: 0.0024
Epoch 2000/2000
483/483 [==============================] - 1s 1ms/step - loss: 7.6894e-04 - val_loss: 0.0024
Total execution time: 1209.4346787929535 seconds
```

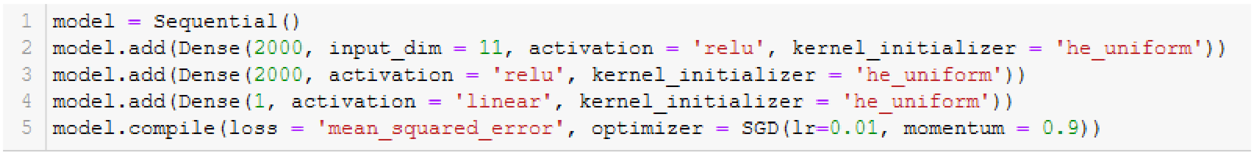
Figure 10 GPU, losses



Speed of CPU vs GPU

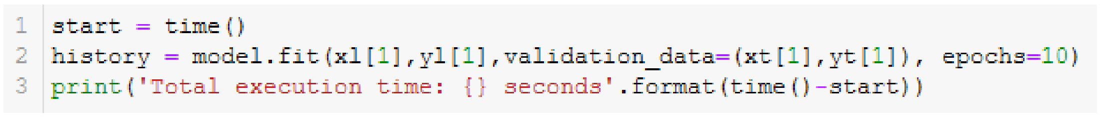
In theory, GPU calculations are supposed to be faster. Nonetheless, there is a threshold of complexity the model must satisfy for this to be true. For simple models, such as the architecture of this exercise of just 30 neurons and 1 hidden layer, the execution time is shorter when the information does not need to be exchanged between GPU and CPU. For more complex architectures, where multiple parallel computations take place, the tradeoff of the information being exchanged is surpassed and GPUs offer a better performance. In order to prove this, the original architecture was modified to have 2000 neurons and the number of epochs was reduced to 10 as shown in Figure 11 and Figure 12. The number of epochs was dramatically reduced to avoid long execution times for the CPU.

Figure 11 Neuron-modified architecture



```python
model = Sequential()
model.add(Dense(2000, input_dim = 11, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(2000, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(1, activation = 'linear', kernel_initializer = 'he_uniform'))
model.compile(loss = 'mean_squared_error', optimizer = SGD(lr=0.01, momentum = 0.9))
```

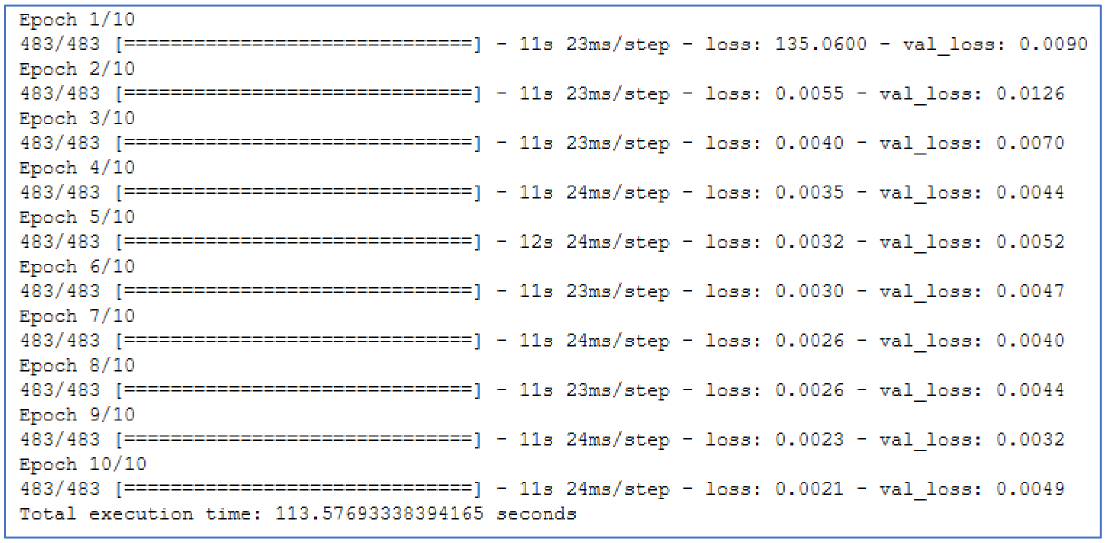
Figure 12 Modified epochs



```
start = time()
history = model.fit(xl[1],yl[1],validation_data=(xt[1],yt[1]), epochs=10)
print('Total execution time: {} seconds'.format(time()-start))
```

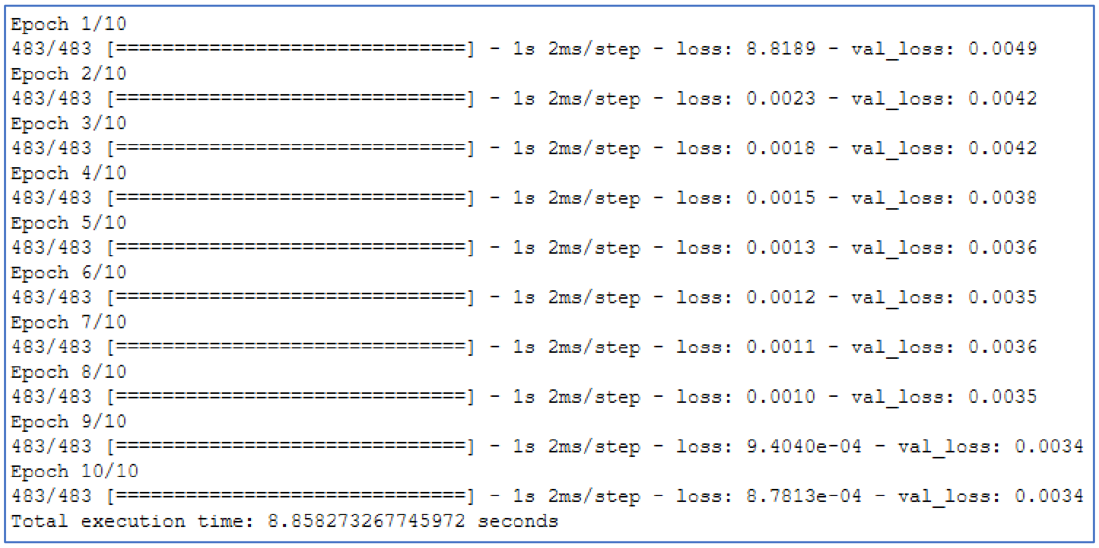
Figure 13 and Figure 14 confirm that using a GPU for multiple parallel operations is the faster option. For this architecture, it can be observed that the execution time for the CPU is at least ten times slower than using the GPU. Projecting the time spent for 10 epochs to the original 2000 epochs, the CPU would take around 22,000 seconds which is more than 6 hours. On the contrary, the GPU would only need around 30 minutes to complete these operations.

Figure 13 CPU, total execution time 2000 neurons



```
Epoch 1/10
483/483 [==============================] - 11s 23ms/step - loss: 135.0600 - val_loss: 0.0090
Epoch 2/10
483/483 [==============================] - 11s 23ms/step - loss: 0.0055 - val_loss: 0.0126
Epoch 3/10
483/483 [==============================] - 11s 23ms/step - loss: 0.0040 - val_loss: 0.0070
Epoch 4/10
483/483 [==============================] - 11s 24ms/step - loss: 0.0035 - val_loss: 0.0044
Epoch 5/10
483/483 [==============================] - 12s 24ms/step - loss: 0.0032 - val_loss: 0.0052
Epoch 6/10
483/483 [==============================] - 11s 23ms/step - loss: 0.0030 - val_loss: 0.0047
Epoch 7/10
483/483 [==============================] - 11s 24ms/step - loss: 0.0026 - val_loss: 0.0040
Epoch 8/10
483/483 [==============================] - 11s 23ms/step - loss: 0.0026 - val_loss: 0.0044
Epoch 9/10
483/483 [==============================] - 11s 24ms/step - loss: 0.0023 - val_loss: 0.0032
Epoch 10/10
483/483 [==============================] - 11s 24ms/step - loss: 0.0021 - val_loss: 0.0049
Total execution time: 113.57693338394165 seconds
```

Figure 14 GPU, total execution time 2000 neurons

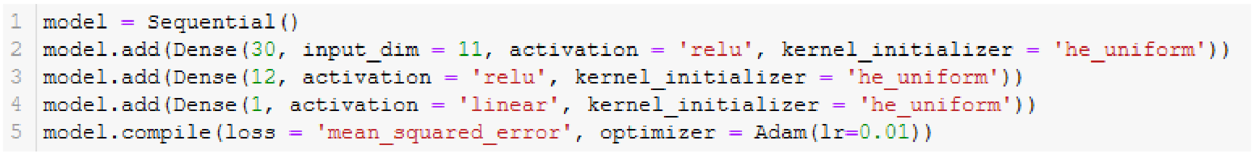


```
Epoch 1/10
483/483 [==============================] - 1s 2ms/step - loss: 8.8189 - val_loss: 0.0049
Epoch 2/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0023 - val_loss: 0.0042
Epoch 3/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0018 - val_loss: 0.0042
Epoch 4/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0015 - val_loss: 0.0038
Epoch 5/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0013 - val_loss: 0.0036
Epoch 6/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0012 - val_loss: 0.0035
Epoch 7/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0011 - val_loss: 0.0036
Epoch 8/10
483/483 [==============================] - 1s 2ms/step - loss: 0.0010 - val_loss: 0.0035
Epoch 9/10
483/483 [==============================] - 1s 2ms/step - loss: 9.4040e-04 - val_loss: 0.0034
Epoch 10/10
483/483 [==============================] - 1s 2ms/step - loss: 8.7813e-04 - val_loss: 0.0034
Total execution time: 8.858273267745972 seconds
```

Experiment 2, SGD vs Adam

The second experiment consists in comparing the results of changing the optimizer from SGD to Adam. The architecture was modified as shown in Figure 15. As the network remains relatively simple, calculations were made using the CPU.

Figure 15 Optimizer-modified architecture

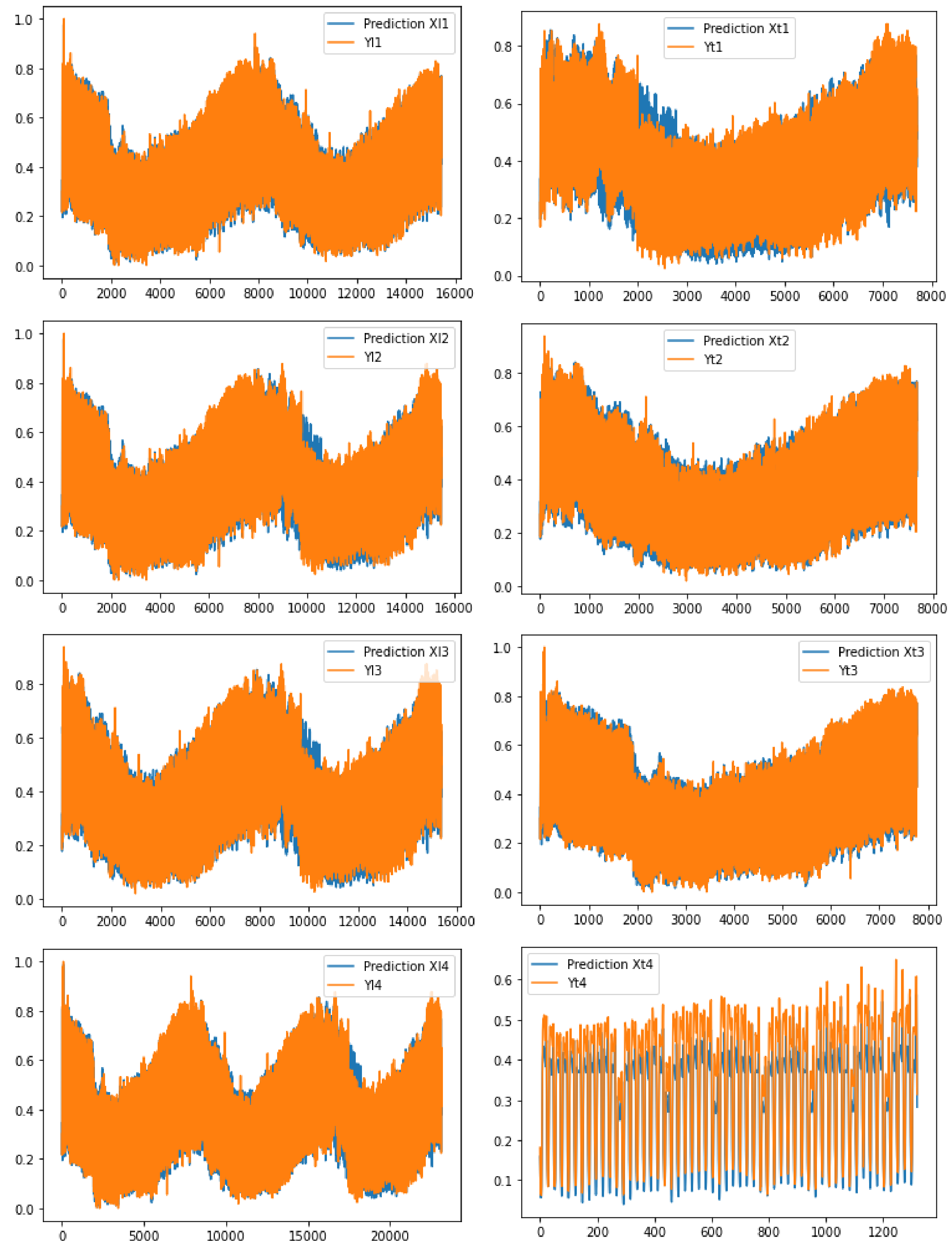


```
model = Sequential()
model.add(Dense(30, input_dim = 11, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(12, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(1, activation = 'linear', kernel_initializer = 'he_uniform'))
model.compile(loss = 'mean_squared_error', optimizer = Adam(lr=0.01))
```

Results Adam optimizer

Figure 16 and Figure 17 help visualize that the predictions made under the Adam optimizer seem to be more accurate than using SGD. Figure 18 further proves this as the final error values are smaller than the original results, while the total execution time is less than 20 seconds slower. Figure 19 shows validation losses do not seem to increase, even though overfitting is eliminated, validation losses fluctuate a lot more in Adam than in SGD.

Figure 16 Adam, Prediction vs Output



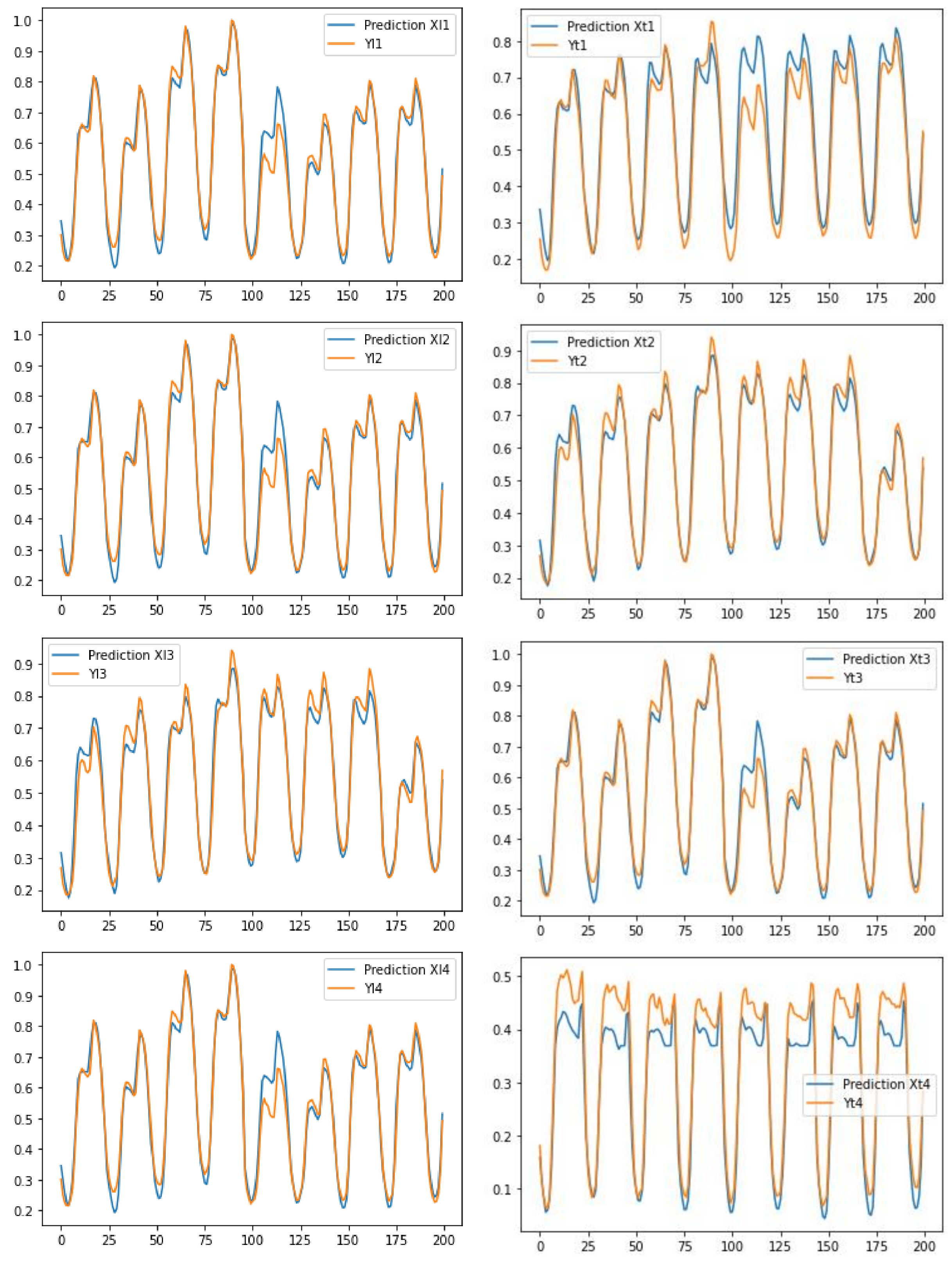
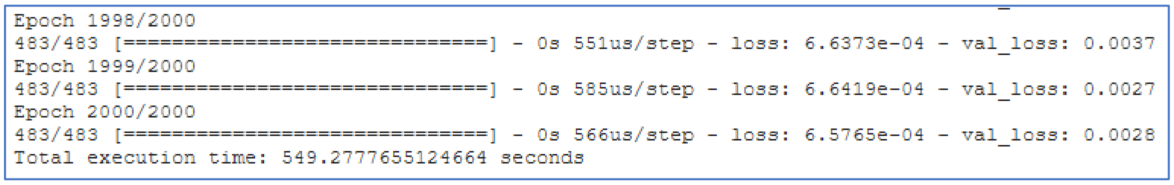


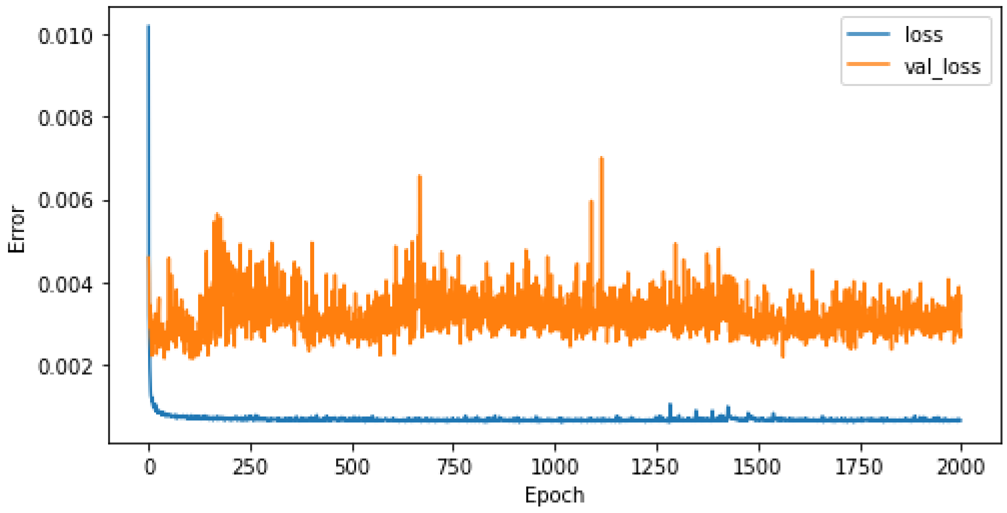
Figure 17 Adam, Prediction vs Output, 200 rows

Figure 18 Adam, total execution time and final values



```
Epoch 1998/2000
483/483 [==============================] - 0s 551us/step - loss: 6.6373e-04 - val_loss: 0.0037
Epoch 1999/2000
483/483 [==============================] - 0s 585us/step - loss: 6.6419e-04 - val_loss: 0.0027
Epoch 2000/2000
483/483 [==============================] - 0s 566us/step - loss: 6.5765e-04 - val_loss: 0.0028
Total execution time: 549.2777655124664 seconds
```

Figure 19 Adam, losses



Experiment 3, Additional hidden layer

The second experiment showed that Adam led to more accurate results, consequently the final experiment is done by using Adam and adding an additional hidden layer to the architecture as shown in Figure 20.

Figure 20 Modified architecture

```
model = Sequential()
model.add(Dense(30, input_dim = 11, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(12, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(12, activation = 'relu', kernel_initializer = 'he_uniform'))
model.add(Dense(1, activation = 'linear', kernel_initializer = 'he_uniform'))
model.compile(loss = 'mean_squared_error', optimizer = Adam(lr=0.01))
```

Results additional hidden layer

As expected, Figure 21 and Figure 22 show how the plots are the most accurate out of all the architectures. The new architecture requires around 30 additional seconds to execute all epochs compared to the original results. Likewise, Figure 23 demonstrates that the final losses are the lowest for this experiment. Validation losses have the same trend as in the previous experiment but with less fluctuations as seen in Figure 24.

Figure 21 Additional hidden layer, Prediction vs Output

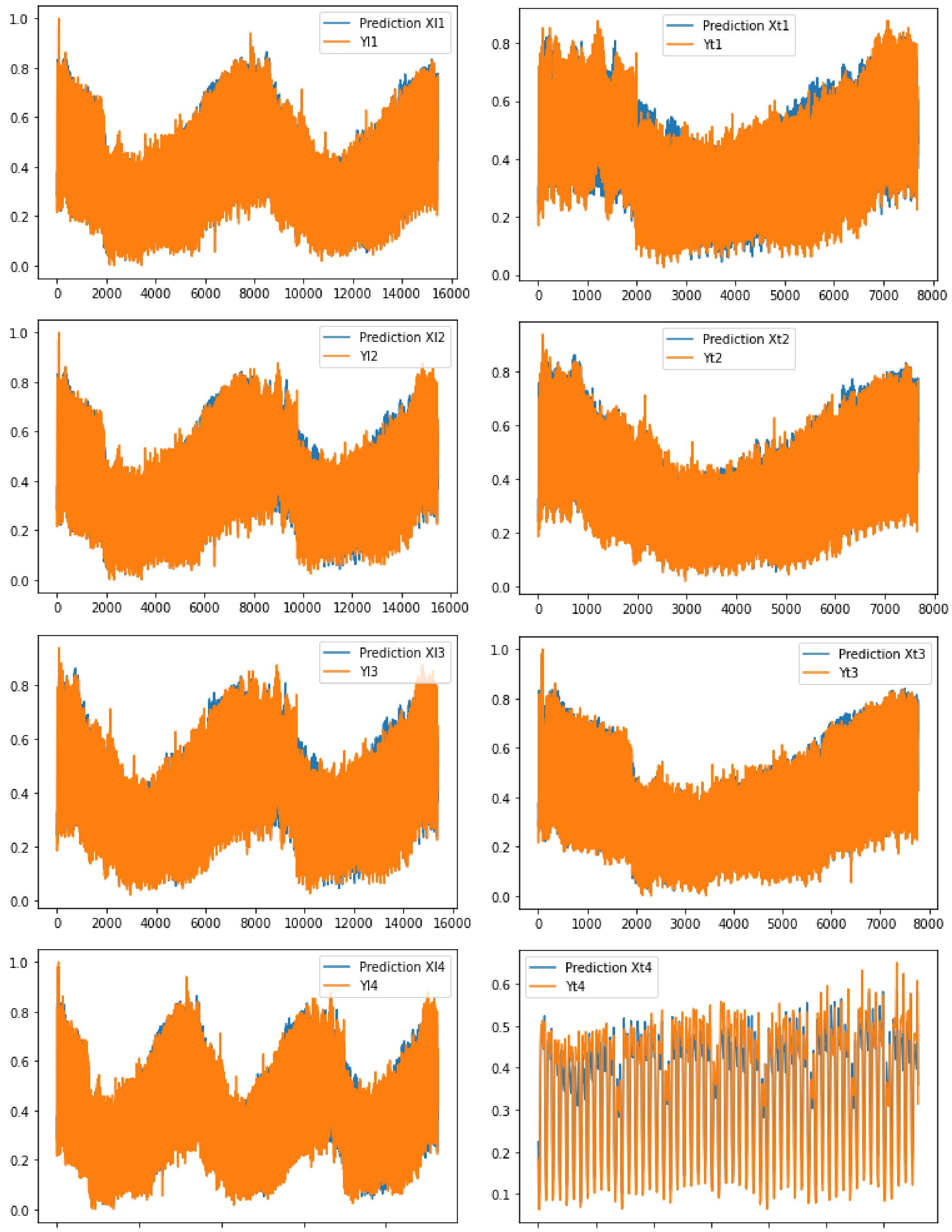


Figure 22 Additional hidden layer, Prediction vs Output, 200 rows

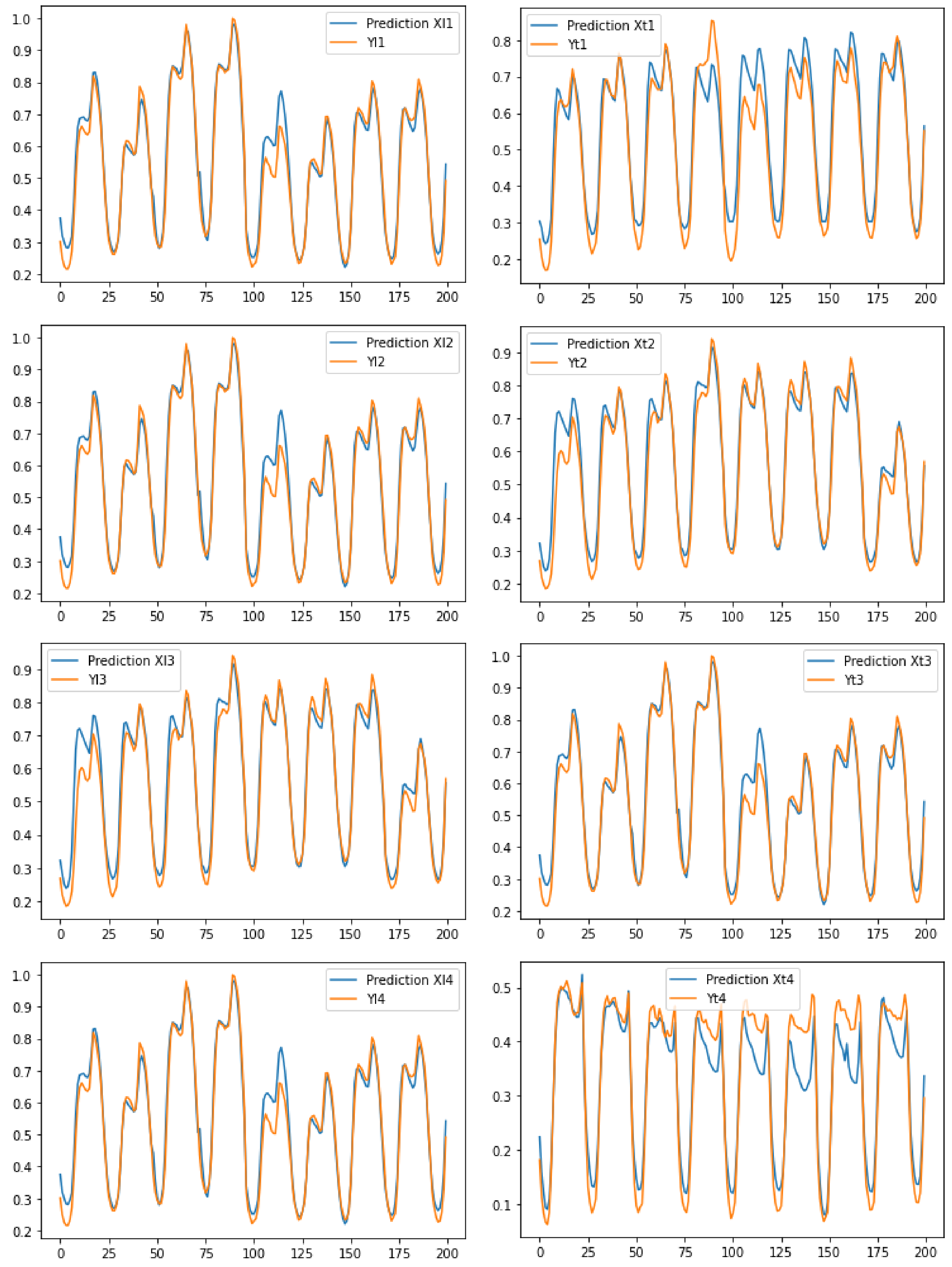
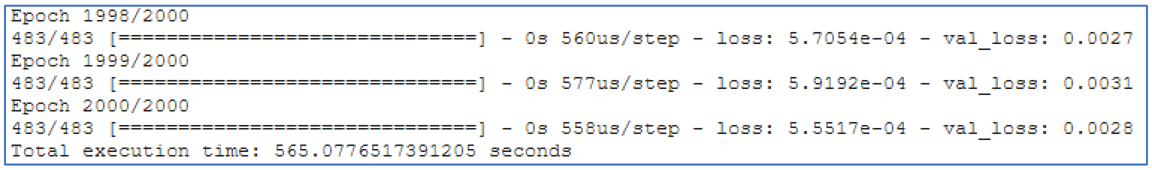
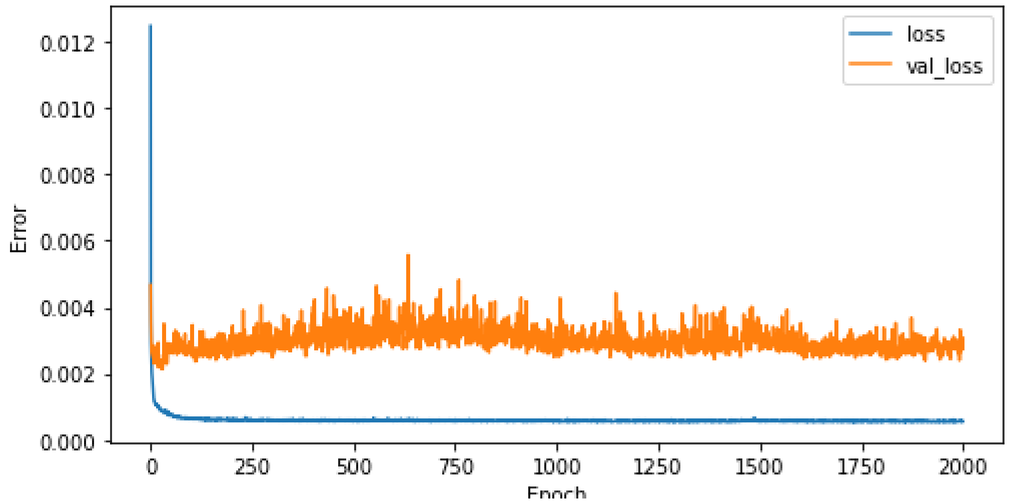


Figure 23 Additional hidden layer, total execution time and final values



```
Epoch 1998/2000
483/483 [==============================] - 0s 560us/step - loss: 5.7054e-04 - val_loss: 0.0027
Epoch 1999/2000
483/483 [==============================] - 0s 577us/step - loss: 5.9192e-04 - val_loss: 0.0031
Epoch 2000/2000
483/483 [==============================] - 0s 558us/step - loss: 5.5517e-04 - val_loss: 0.0028
Total execution time: 565.0776517391205 seconds
```

Figure 24 Additional hidden layer, losses



Conclusion

Experiment 1 revealed that for the rather simplistic architecture there is no need to use a GPU and it is in fact detrimental for this scenario. Execution times were almost three times slower for the GPU with the original architecture. Increasing the number of neurons made the option of a GPU-based solver a more attracting solution, as using the CPU would take more than 6 hours to complete all epochs. Therefore, all consequent experiments were done using a CPU. Experiment 2 confirmed that using the Adam optimizer made the ANN deliver more accurate results while slightly increasing execution time by only 10 seconds. Finally, as anticipated, experiment 3 provides the best results as it combines Adam optimizer with an additional hidden layer, while only adding 30 seconds of additional total execution time.