CS109A Introduction to Data Science:

Homework 9: ANNs

Harvard University
Fall 2018
Instructors: Pavlos Protopapas, Kevin Rader

In [1]:
```
# RUN THIS CELL FOR FORMAT
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS
HTML(styles)
```

Out[1]:

Import libraries:

In [2]:
```python
import random
random.seed(112358)

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import cross_val_score
from sklearn.utils import resample
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import r2_score

import keras
from keras.models import Sequential
from keras.layers import Dense

%matplotlib inline

import seaborn as sns
pd.set_option('display.width', 1500)
pd.set_option('display.max_columns', 100)

from keras import regularizers

from sklearn.utils import shuffle
```

```
C:\Applications\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning:
Conversion of the second argument of issubdtype from `float` to `np.floating` i
s deprecated. In future, it will be treated as `np.float64 == np.dtype(float).t
ype`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Neural Networks

Neural networks are, of course, a large and complex topic that cannot be covered in a single homework. Here we'll focus on the key idea of NNs: they are able to learn a mapping from example input data (of fixed size) to example output data (of fixed size). We'll also partially explore what patterns the neural network learns and how well they generalize.

In this question we'll see if Neural Networks can learn a (limited) version of the Fourier Transform. (The Fourier Transform takes in values from some function and returns a set of sine and cosine functions which, when added together, approximate the original function.)

In symbols: $\mathcal{F}(s) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i x s}dx$. In words, the value of the transformed function at some point, s, is the value of an integral which measures, in some sense, how much the original f(x) looks like a wave with period s. As an example, with $f(x) = 4cos(x) + sin(2x)$, $\mathcal{F}(s)$ is 0 everywhere except at -2, -1, 1, and 2, mapping to the waves of period 1 and 1/2. The values at these points are linked to the magnitude of the waves, and their phases (roughly: sin waves versus cosine waves).

The only thing about the Fourier transform that matters for this pset is this: function goes in, re-write in terms of sine and cosine comes out.

In our specific problem, we'll train a network to map from 1000 sample values from a function (equally spaced along 0 to 2π) to the four features of the sine and cosine waves that make up that function. Thus, the network is attempting to learn a mapping from a 1000-entry vector down to a 4-entry vector. Our X_train dataset is thus N by 1000 and our y_train is N by 4.

Questions 1.1 and 1.2 will get you used to the format of the data.

We'll use 6 data files in this question:

- `sinewaves_X_train.npy` and `sinewaves_y_train.npy` : a (10,000 by 1,000) and (10,000 by 4) training dataset. Examples were generated by randomly selecting a,b,c,d in the interval [0,1] and building the curve $a\sin(b\,x) + c\cos(d\,x)$
- `sinewaves_X_test.npy` and `sinewaves_y_test.npy` : a (2,000 by 1,000) and (2,000 by 4) test dataset, generated in the same way as the training data
- `sinewaves_X_extended_test` and `sinewaves_y_extended_test` : a (9 by 1,000) and (9 by 4) test dataset, testing whether the network can generalize beyond the training data (e.g. to negative values of a)

These datasets are read in to their respective variables for you.

Question 1 [50pts]

1.1 Plot the first row of the `X_train` training data and visually verify that it is a sinusoidal curve.

1.2 The first row of the `y_train` data is $[0.024, 0.533, 0.018, 0.558]$. Visually or numerically verify that the first row of X_train is 1000 equally-spaced samples in $[0, 10\pi]$ from the function $f(x) = 0.024\sin(0.533\,x) + 0.018\cos(0.558\,x)$. This pattern (y_train is the true parameters of the curve in X_train) will always hold.

1.3 Use `Sequential` and `Dense` from Keras to build a fully-connected neural network. You can choose any number of layers and any number of nodes in each layer.

1.4 Compile your model via the line `model.compile(loss='mean_absolute_error', optimizer='adam')` and display the `.summary()` . Explain why the first layer in your network has the indicated number of parameters.

1.5 Fit your model to the data for 50 epochs using a batch size of 32 and a validation split of 0.2. You can train for longer if you wish- the fit tends to improve over time.

1.6 Use the `plot_predictions` function to plot the model's predictions on `X_test` to the true values in `y_test` (by default, it will only plot the first few rows). Report the model's overall loss on the test set. Comment on how well the model performs on this unseen data. Do you think it has accurately learned how to map from sample data to the coefficients that generated the data?

1.7 Examine the model's performance on the 9 train/test pairs in the `extended_test` variables. Which examples does the model do well on, and which examples does it struggle with?

1.8 Is there something that stands out about the difficult examples, especially with respect to the data the model was trained on? Did the model learn the mapping we had in mind? Would you say the model is overfit, underfit, or neither?

Hint:

- Keras's documentation and examples of a Sequential model are a good place to start.
- A strong model can achieve validation error of around 0.03 on this data and 0.02 is very good.

In [3]:
```
def plot_predictions(model, test_x, test_y, count=None):
    # Model - a Keras or SKlearn model that takes in (n,1000) training data a
    # test_x - a (n,1000) input dataset
    # test_y - a (n,4) output dataset
    # This function will plot the sine curves in the training data and those
    # It will also print the predicted and actual output values.

    #helper function that takes the n by 4 output and reverse-engineers
    #the sine curves that output would create
    def y2x(y_data):
        #extract parameters
        a=y_data[:,0].reshape(-1,1)
        b=y_data[:,1].reshape(-1,1)
        c=y_data[:,2].reshape(-1,1)
        d=y_data[:,3].reshape(-1,1)

        #build the matching training data
        x_points = np.linspace(0,10*np.pi,1000)
        x_data = a*np.sin(np.outer(b,x_points)) + c*np.cos(np.outer(d,x_point
        return x_data

    #if <20 examples, plot all. If more, just plot 5
    if count==None:
        if test_x.shape[0]>20:
            count=5
        else:
            count=test_x.shape[0]

    #build predictions
    predicted = model.predict(test_x)
    implied_x = y2x(predicted)
    for i in range(count):
        plt.plot(test_x[i,:],label='true')
        plt.plot(implied_x[i,:],label='predicted')
        plt.legend()
        plt.ylim(-2.1,2.1)
        plt.xlabel("x value")
        plt.xlabel("y value")
        plt.title("Curves using the Neural Network's Approximate Fourier Tran
        plt.show()
        print("true:", test_y[i,:])
        print("predicted:", predicted[i,:])
```

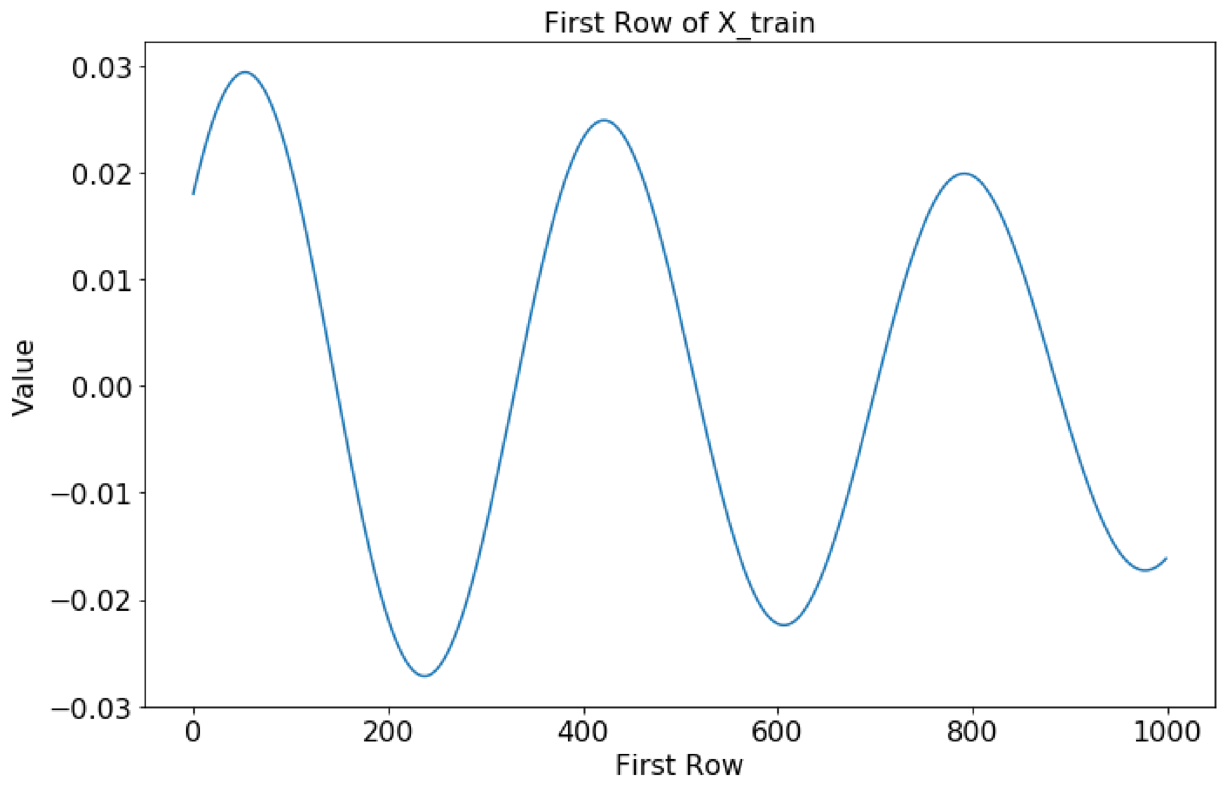
In [4]:
```
X_train = np.load('data/sinewaves_X_train.npy')
y_train = np.load('data/sinewaves_y_train.npy')

X_test = np.load('data/sinewaves_X_test.npy')
y_test = np.load('data/sinewaves_y_test.npy')

X_extended_test = np.load('data/sinewaves_X_extended_test.npy')
y_extended_test = np.load('data/sinewaves_y_extended_test.npy')
```

Answers:

1.1 Plot the first row of the `X_train` training data and visually verify that it is a sinusoidal curve

In [5]:
```
# your code here
print("Shape of X_train: " + str(X_train.shape))
print("Length of first row of X_train: " + str(len(X_train[0, :])))

fig, ax = plt.subplots(1, 1, figsize=(11, 7))
ax.plot(X_train[0, :])
ax.tick_params(labelsize=16)
ax.set_title('First Row of X_train', fontsize=16)
ax.set_xlabel("First Row", fontsize=16)
ax.set_ylabel("Value", fontsize=16)
plt.show()
```

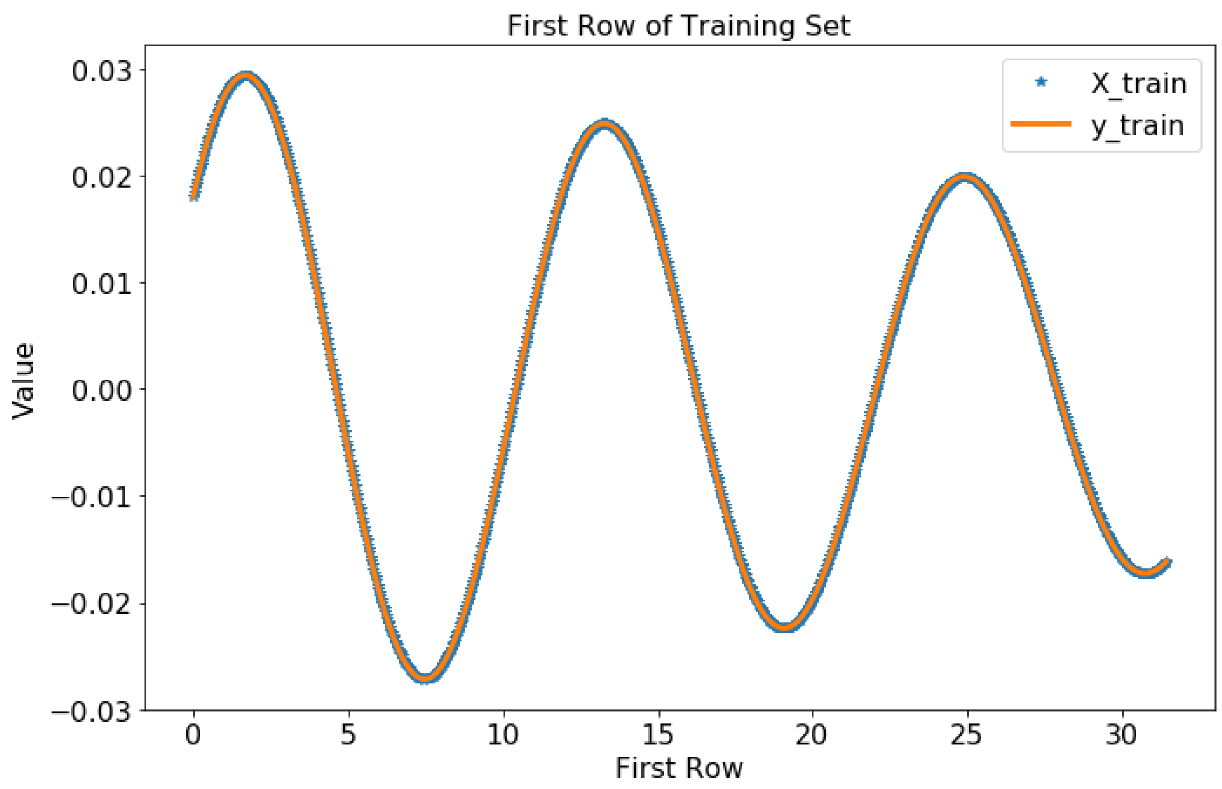
```
Shape of X_train: (10000, 1000)
Length of first row of X_train: 1000
```



The above chart visually shows a sinusoidal curve.

1.2 The first row of the `y_train` data is $[0.024, 0.533, 0.018, 0.558]$. Visually or numerically verify that the first row of X_train is 1000 equally-spaced samples in $[0, 10\pi]$ from the function $f(x) = 0.024\sin(0.533\,x) + 0.018\cos(0.558\,x)$. This pattern (y_train is the true parameters of the curve in X_train) will always hold.

In [6]:
```python
# your code here
x = np.linspace(0, 10 * np.pi, 1000)
f = 0.024 * np.sin(0.533 * x) + 0.018 * np.cos(0.558 * x)

fig, ax = plt.subplots(1, 1, figsize=(11, 7))
ax.plot(x, X_train[0, :], '*', label='X_train')
ax.plot(x, f, lw=3, label='y_train')
ax.tick_params(labelsize=16)
ax.set_title('First Row of Training Set', fontsize=16)
ax.set_xlabel("First Row", fontsize=16)
ax.set_ylabel("Value", fontsize=16)
ax.legend(fontsize=16)
plt.show()
```



In [7]:
```python
import scipy
scipy.stats.describe(f - X_train[0, :])
```

Out[7]:
```
DescribeResult(nobs=1000, minmax=(0.0, 0.0), mean=0.0, variance=0.0, skewness=
0.0, kurtosis=-3.0)
```

Verification:

- Visually from the chart above, the first row of X_train matches with the function constructed given the parameters in the first row of y_train.

- Numerically from the above statistics of the difference beween the two arrays, the two arrays are exactly the same.

1.3 Use `Sequential` and `Dense` from Keras to build a fully-connected neural network. You can choose any number of layers and any number of nodes in each layer.

In [8]:
```python
# your code here
n_hidden = 250
n_ouput = 4

# Initialize the model
model = Sequential()

# Hidden Layer
model.add(Dense(n_hidden, activation='sigmoid', input_shape=(X_train.shape[1]

# More hidden layers
for _ in range(4):
    model.add(Dense(n_hidden, activation = 'sigmoid'))

# Output layer
model.add(Dense(n_ouput, activation = 'linear'))
```

1.4 Compile your model via the line `model.compile(loss='mean_absolute_error', optimizer='adam')` and display the `.summary()`. Explain why the first layer in your network has the indicated number of parameters.

In [9]:
```python
# your code here
# Compile it
model.compile(loss='mean_absolute_error', optimizer='adam')

# Summary
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 250)               250250    
_________________________________________________________________
dense_2 (Dense)              (None, 250)               62750     
_________________________________________________________________
dense_3 (Dense)              (None, 250)               62750     
_________________________________________________________________
dense_4 (Dense)              (None, 250)               62750     
_________________________________________________________________
dense_5 (Dense)              (None, 250)               62750     
_________________________________________________________________
dense_6 (Dense)              (None, 4)                 1004      
=================================================================
Total params: 502,254
Trainable params: 502,254
Non-trainable params: 0
_________________________________________________________________
```

Your answer here

There are 250,250 number of parameters in the first layer (hidden layer) with 250 nodes. There are 1,000 features in the X_train, times 250 nodes in the first layer, which gives 250,000 parameters, plus 250 bias terms. Thus, in total, we have 250,250 parameters to train in the first layer.

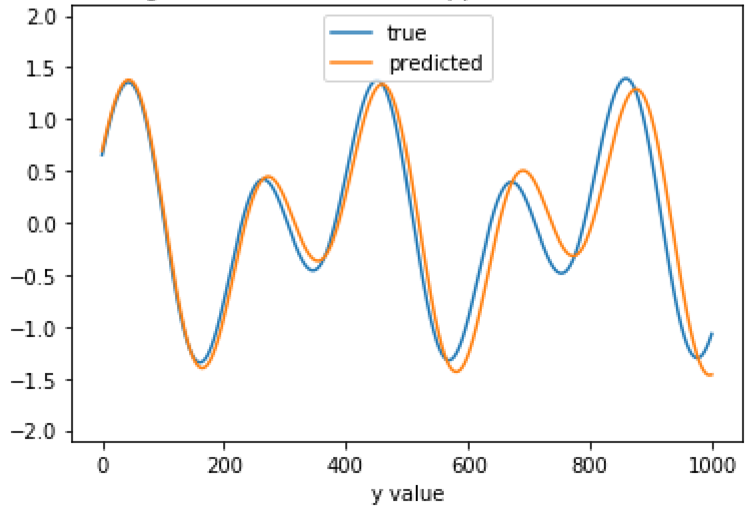
1.5 Fit your model to the data for 50 epochs using a batch size of 32 and a validation split of .2. You can train for longer if you wish- the fit tends to improve over time. A good network can achieve a loss of .03, and .02 is excellent.

In [10]:
```python
# your code here
epochs = 50
batch_size = 32
validation_split = 0.2

model_hist = model.fit(X_train, y_train,
                       batch_size=batch_size, epochs=epochs, verbose=False,
                       validation_split=validation_split)
```

1.6 Use the `plot_predictions` function to plot the model's predictions on `X_test` to the true values in `y_test` (by default, it will only plot the first few rows). Report the model's overall loss on the test set. Comment on how well the model performs on this unseen data. Do you think it has accurately learned how to map from sample data to the coefecients that generated the data?

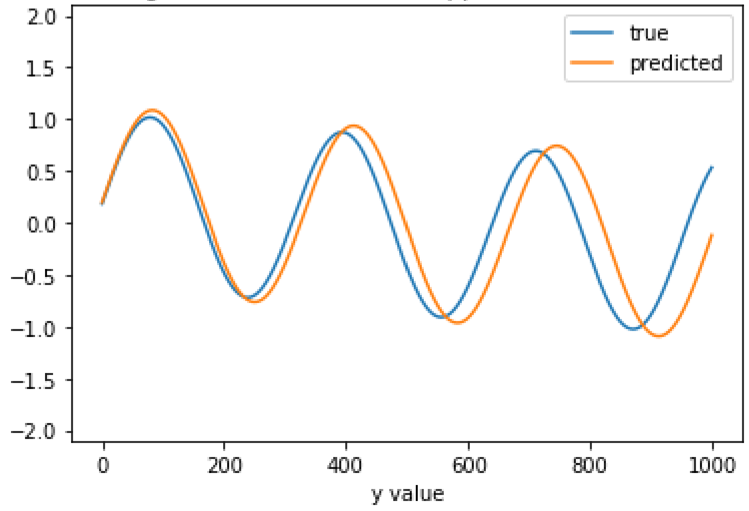
In [11]:
```
# your code here
plot_predictions(model, X_test, y_test)
```

Curves using the Neural Network's Approximate Fourier Transform

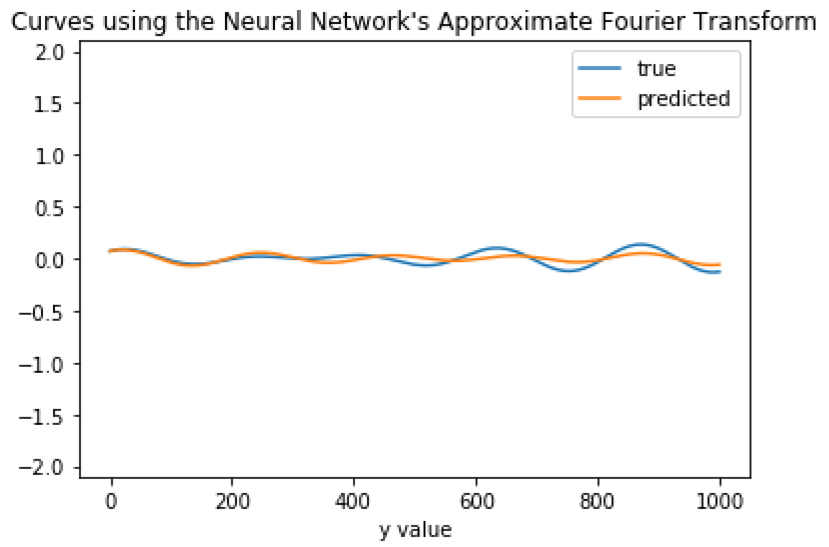


```
true: [0.86199664 0.98175913 0.65523998 0.4870337 ]
predicted: [0.8601745  0.95918846 0.69671506 0.48683256]
```

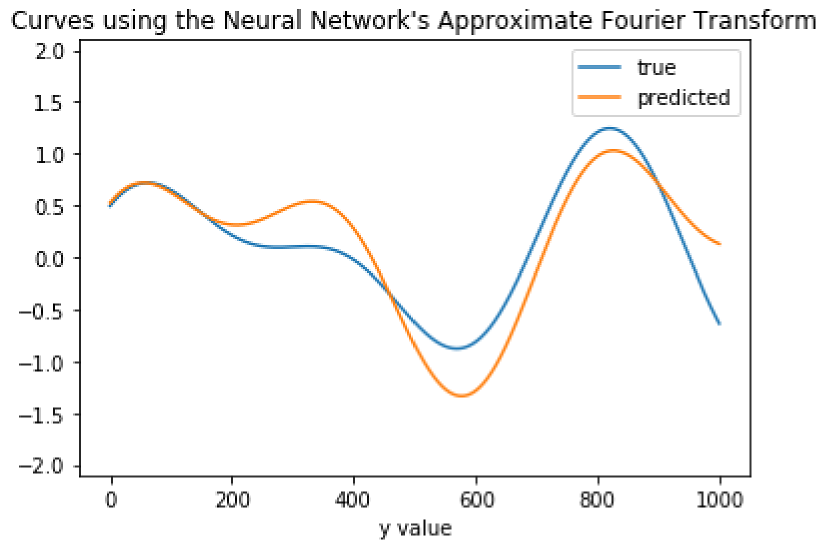
Curves using the Neural Network's Approximate Fourier Transform



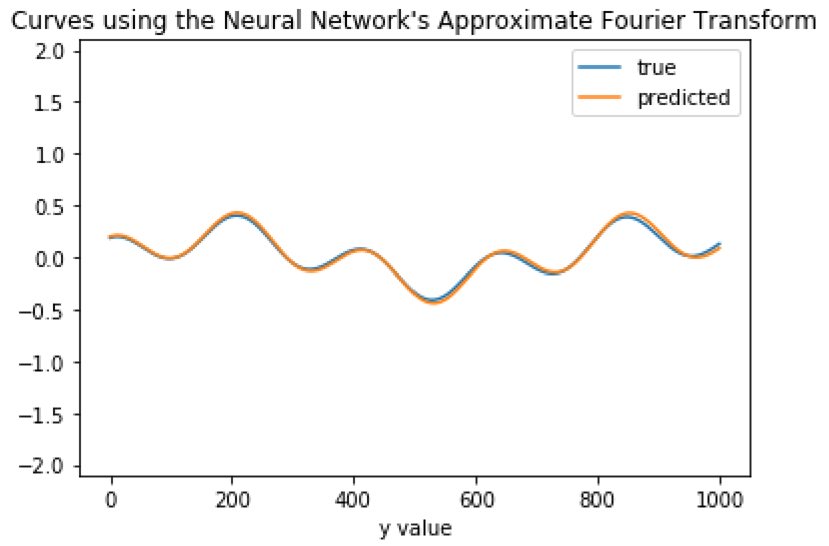
```
true: [0.8406355  0.63159555 0.18328701 0.11174618]
predicted: [0.8957183  0.60264045 0.19750291 0.10488825]
```



```
true: [0.06591224 0.75183886 0.06986143 0.91352303]
predicted: [0.04800424 0.8052628  0.07062822 0.8908605 ]
```



```
true: [0.75610725 0.30861152 0.49522059 0.48394499]
predicted: [0.84598464 0.2730945  0.524877   0.5089449 ]
```

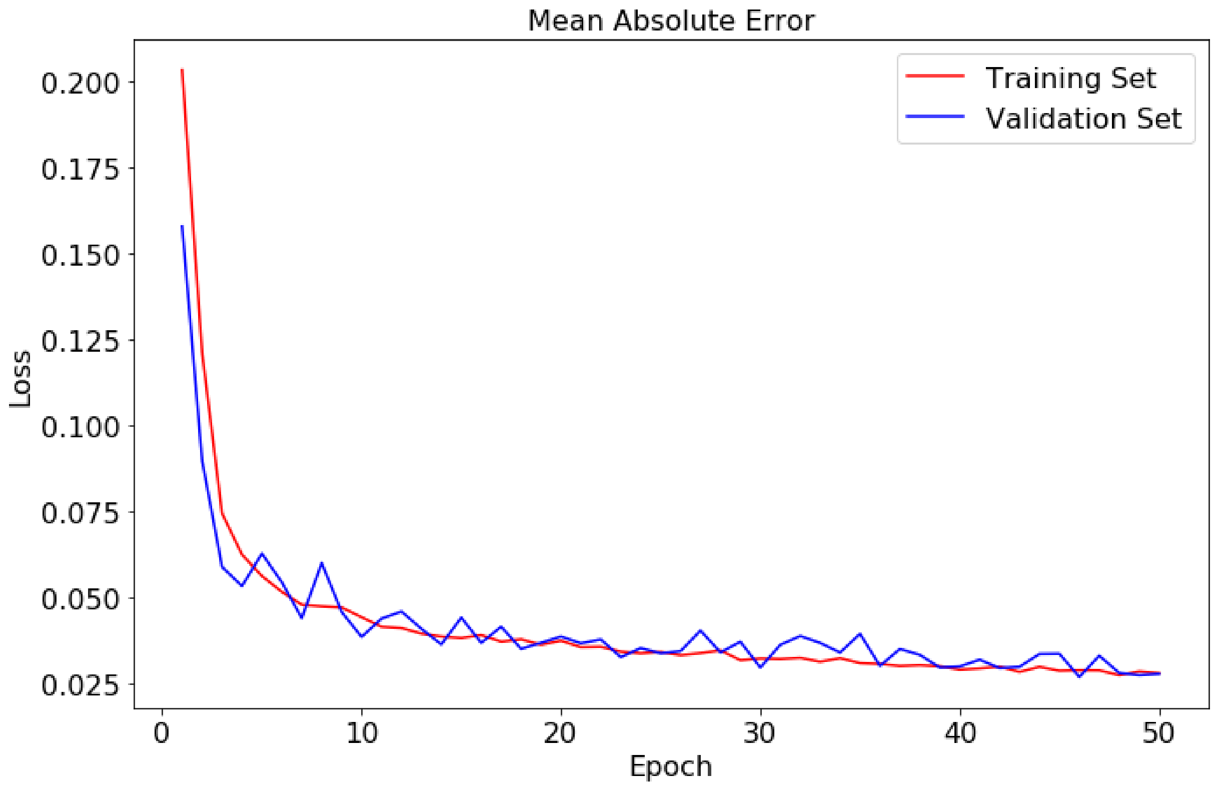


```
true: [0.2229353  0.27885697 0.18696198 0.94846283]
```

```
predicted: [0.24110106 0.28141394 0.19838232 0.9424417 ]
```

In [12]:

```python
fig, ax = plt.subplots(1, 1, figsize=(11,7))
ax.plot(range(1, epochs + 1), model_hist.history['loss'], 'r', label='Trainin
ax.plot(range(1, epochs + 1), model_hist.history['val_loss'], 'b', label='Val
ax.set_title('Mean Absolute Error', fontsize=16)
ax.set_xlabel('Epoch', fontsize=16)
ax.set_ylabel('Loss', fontsize=16)
ax.legend(fontsize=16)
ax.tick_params(labelsize=16)
```



In [13]:

```python
train_score = model.evaluate(X_train, y_train, verbose=False)
print('Train loss = {:.3f}'.format(train_score))
print('Train R2 = {:.2%}'.format(r2_score(y_train, model.predict(X_train))))

test_score = model.evaluate(X_test, y_test, verbose=False)
print('Test loss = {:.3f}'.format(test_score))
print('Test R2 = {:.2%}'.format(r2_score(y_test, model.predict(X_test))))
```

```
Train loss = 0.025
Train R2 = 96.62%
Test loss = 0.026
Test R2 = 96.49%
```

Your answer here

The model's overall loss on the test set is 0.026.

The model performs very well on the unseen data with overall loss of 0.026 and R-squared of 96.49%. The loss is very low and R-squared is very high in the test set.

The above plots show the true and predicted curves for the 5 samples, and the true and predicted curves are very close. Based on the charts and the strong statistics in the test set, I think the model has reasonably accurately learned how to map from sample data to the coefficients that generated the data.

Additionally, based on the loss vs epoch chart above, the validation error keeps trending down without deviating from the training error, so the model keeps to be improved with more epochs and there is no sign of overfitting.

1.7 Examine the model's performance on the 9 train/test pairs in the `extended_test` variables. Which examples does the model do well on, and which examples does it struggle with?

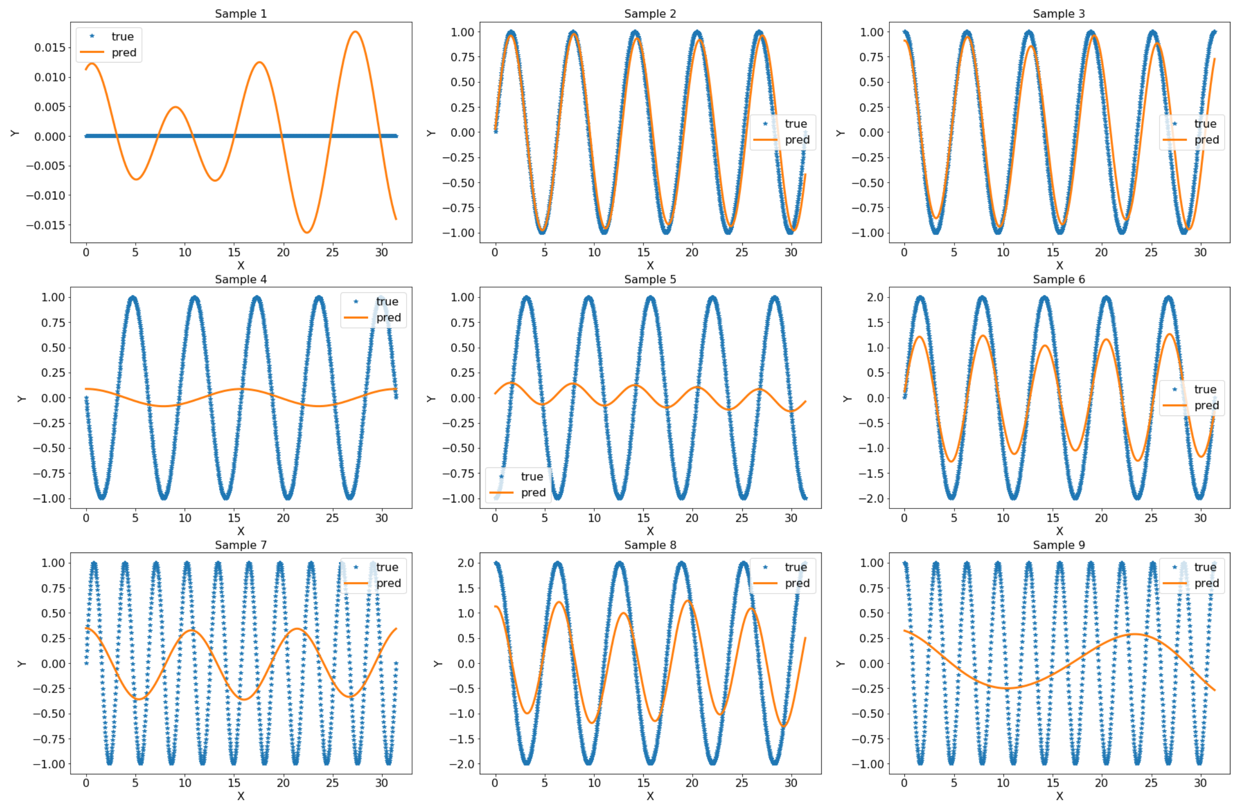
In [14]:

```python
# your code here
test_extended_score = model.evaluate(X_extended_test, y_extended_test, verbos
print('Extended Test loss = {:.3f}'.format(test_extended_score))

test_extended_pred = model.predict(X_extended_test)
print('Extended Test R2 = {:.2%}'.format(r2_score(y_extended_test, test_exten
```

```
Extended Test loss = 0.518
Extended Test R2 = -5.29%
```

In [15]:

```python
# Visual Comparison chart
f, ax = plt.subplots(3, 3, figsize = (30, 20))

for i in range(len(ax)):
    for j in range(len(ax[0])):
        index = i * len(ax[0]) + j

        a_true, b_true, c_true, d_true = y_extended_test[index, :]
        a_pred, b_pred, c_pred, d_pred = test_extended_pred[index, :]

        x = np.linspace(0, 10 * np.pi, 1000)
        f_true = a_true * np.sin(b_true * x) + c_true * np.cos(d_true * x)
        f_pred = a_pred * np.sin(b_pred * x) + c_pred * np.cos(d_pred * x)

        ax[i][j].plot(x, f_true, '*', label='true')
        ax[i][j].plot(x, f_pred, lw=3, label='pred')
        ax[i][j].tick_params(labelsize=16)
        ax[i][j].set_title('Sample {}'.format(index + 1), fontsize=16)
        ax[i][j].set_xlabel("X", fontsize=16)
        ax[i][j].set_ylabel("Y", fontsize=16)
        ax[i][j].legend(loc='best', fontsize=16)

plt.show()
```



Your answer here

On the extended test set, the loss is 0.518 and R2 is -5.29%. So the model is performing very badly on the extended test set.

Examples the model does well:

- As shown in above charts and tables, the model does well on sample 2 and 3. These two samples have the lowest mean absolute error, and the true and predicted curves are very close.

Examples the model struggles with:

- Except for sample 2 and 3, the model stuggles with all other samples. The model has large mean absolute error on these samples, and the true and predicted curves don't align.

1.8 Is there something that stands out about the difficult observations, especially with respect to the data the model was trained on? Did the model learn the mapping we had in mind? Would you say the model is overfit, underfit, or neither?

In [16]:

```
# True and predicted value comparison table
df_test_extended_pred = pd.DataFrame(test_extended_pred, columns = ['a_pred',
df_y_extended_test = pd.DataFrame(y_extended_test, columns = ['a_true', 'b_tr
df_comp = pd.merge(df_test_extended_pred, df_y_extended_test,left_index=True,

df_comp['a_diff'] = abs(df_comp['a_pred'] - df_comp['a_true'])
df_comp['b_diff'] = abs(df_comp['b_pred'] - df_comp['b_true'])
df_comp['c_diff'] = abs(df_comp['c_pred'] - df_comp['c_true'])
df_comp['d_diff'] = abs(df_comp['d_pred'] - df_comp['d_true'])
df_comp['sum_diff'] = df_comp['a_diff'] + df_comp['b_diff'] + df_comp['c_diff
df_comp.sort_values(by='sum_diff', inplace=True)
df_comp
```

Out[16]:

	a_pred	b_pred	c_pred	d_pred	a_true	b_true	c_true	d_true	a_diff	b_diff	
2	0.056777	0.385805	0.910261	0.980933	0.0	0.0	1.0	1.0	0.056777	0.385805	0.
1	0.946192	0.985844	0.032513	0.735799	1.0	1.0	0.0	0.0	0.053808	0.014156	0.
0	0.006384	0.516561	0.011286	0.691244	0.0	0.0	0.0	0.0	0.006384	0.516561	0
7	0.133395	0.374868	1.126633	0.967617	0.0	0.0	2.0	1.0	0.133395	0.374868	0.
5	1.151271	0.995421	0.122845	0.687227	2.0	1.0	0.0	0.0	0.848729	0.004579	0.
3	-0.000885	-0.326392	0.086244	0.399802	-1.0	1.0	0.0	0.0	0.999115	1.326392	0.
8	0.078450	-0.395047	0.321860	0.283776	0.0	0.0	1.0	2.0	0.078450	0.395047	0.
4	0.106289	0.998371	0.041580	-0.079772	0.0	0.0	-1.0	1.0	0.106289	0.998371	1.
6	0.021301	-0.135660	0.346724	0.587989	1.0	2.0	0.0	0.0	0.978699	2.135660	0.

In [17]:
```python
# Distribution of value in y_train
df_y_train = pd.DataFrame(y_train, columns = ['a_train', 'b_train', 'c_train'
df_y_train.describe()
```

Out[17]:

	a_train	b_train	c_train	d_train
count	10000.000000	10000.000000	10000.000000	10000.000000
mean	0.501988	0.497651	0.500799	0.494677
std	0.289179	0.291927	0.287402	0.287070
min	0.000128	0.000139	0.000047	0.000051
25%	0.251623	0.239095	0.254517	0.247989
50%	0.506676	0.504151	0.499648	0.493205
75%	0.752637	0.749297	0.750759	0.738573
max	0.999560	0.999944	0.999989	0.999707

Your answer here

As shown in the table of distribution of value in y_train above, all a, b, c and d in the training set seem to be uniformly ditributed between 0 and 1. So essentially the model has been trained to predict a, b, c and d to have a value between 0 and 1. However, the samples 4 to 9 all have one value (among a, b, c and d) outside the range, for example b of 2 in sample 7, c of -1 in sample 5, etc.

I think the model has the learned the mapping we have in mind, with very high training R-squared of 96.44% and test R-squared of 96.24%.

I would say the model is overfited on the training set which only have postive response values between 0 and 1. When the model saw the samples outside the range, it is not able to generalize the behavior and capture the pattern correctly.

Regularizing Neural Networks

In this problem set we have already explored how ANN are able to learn a mapping from example input data (of fixed size) to example output data (of fixed size), and how well the neural network can generalize. In this problem we focus on issues of overfitting and regularization in Neural Networks.

As we have explained in class, ANNs can be prone to overfitting, where they learn specific patterns present in the training data, but the patterns don't generalize to fresh data.

There are several methods used to improve ANN generalization. One approach is to use an achitecutre just barely wide/deep enough to fit the data. The idea here is that smaller networks are less expressive and thus less able to overfit the data.

However, it is difficult to know a priori the correct size of the ANN, and computationally costly to hunt for a correct size. Given this, other methodologies are used to fight overfitting and improve the ANN generalization. These, like other techniques to combat overfitting, fall under the umbrella of Regularization.

In this problem you are asked to regularize a network given to you below. The train dataset can be generated using the code also given below.

Question 2 [50 pts]

2.1 Data Download and Exploration: For this problem, we will be working with the MNIST dataset (Modified National Institute of Standards and Technology database) which is a large database of handwritten digits and commonly used for training various image processing systems. We will be working directly with the download from `keras MNIST dataset` of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

Please refer to the code below to process the data.

For pedagogical simplicity, we will only use the digits labeled `4` and `9`, and we want to use a total of 800 samples for training.

2.2 Data Exploration and Inspection: Use `imshow` to display a handwritten 4 and a handwritten 9.

2.3 Overfit an ANN: Build a fully connected network (FCN) using `keras`:

1. Nodes per Layer: 100,100,100,2 (<-the two-class 'output' layer)
2. Activation function: reLU
3. Loss function: binary_crossentropy
4. Output unit: Sigmoid
5. Optimizer: sgd (use the defaults; no other tuning)
6. Epochs: no more than 2,000
7. Batch size: 128
8. Validation size: .5

This NN trained on the dataset you built in 2.1 will overfit to the training set. Plot the training accuracy and validation accuracy as a function of epochs and explain how you can tell it is overfitting.

2.4 Explore Regularization: Your task is to regularize this FCN. You are free to explore any method or combination of methods. If you are using anything besides the methods we have covered in class, give a citation and a short explanation. You should always have an understanding of the methods you are using.

Save the model using `model.save(filename)` and submit in canvas along with your notebook.

We will evaluate your model on a test set we've kept secret.

1. Don't try to use extra data from NMIST. We will re-train your model on training set under the settings above.
2. Keep the architecture above as is. In other words keep the number of layers, number of nodes, activation function, and loss fucntion the same. You can change the number of epochs (max 2000), batch size, optimizer and of course add elements that can help to regularize (e.g. drop out, L2 norm etc). You can also do data augmentation.
3. You *may* import new modules, following the citation rule above.

Grading: Your score will be based on how much you can improve on the test score via regularization:

1. (0-1] percent will result into 10 pts
2. (1-2] percent will result into 20 pts
3. (2-3] percent will result into 30 pts
4. Above 3 percent will result in 35 pts
5. Top 15 groups or single students will be awarded an additional 10 pts
6. The overall winner(s) will be awarded an additional 5 pts

2.1 Data Download and Exploration: For this problem, we will be working with the MNIST dataset (Modified National Institute of Standards and Technology database) which is a large database of handwritten digits and commonly used for training various image processing systems. We will be working directly with the download from `keras MNIST dataset` of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

Please refer to the code below to process the data.

For pedagogical simplicity, we will only use the digits labeled `4` and `9`, and we want to use a total of 800 samples for training.

In [18]:
```python
## Read and Setup train and test splits in one
from keras.datasets import mnist
from random import randint

(x_train, y_train), (x_test, y_test) = mnist.load_data()

#shuffle the data before we do anything
x_train, y_train = shuffle(x_train, y_train, random_state=1)
```

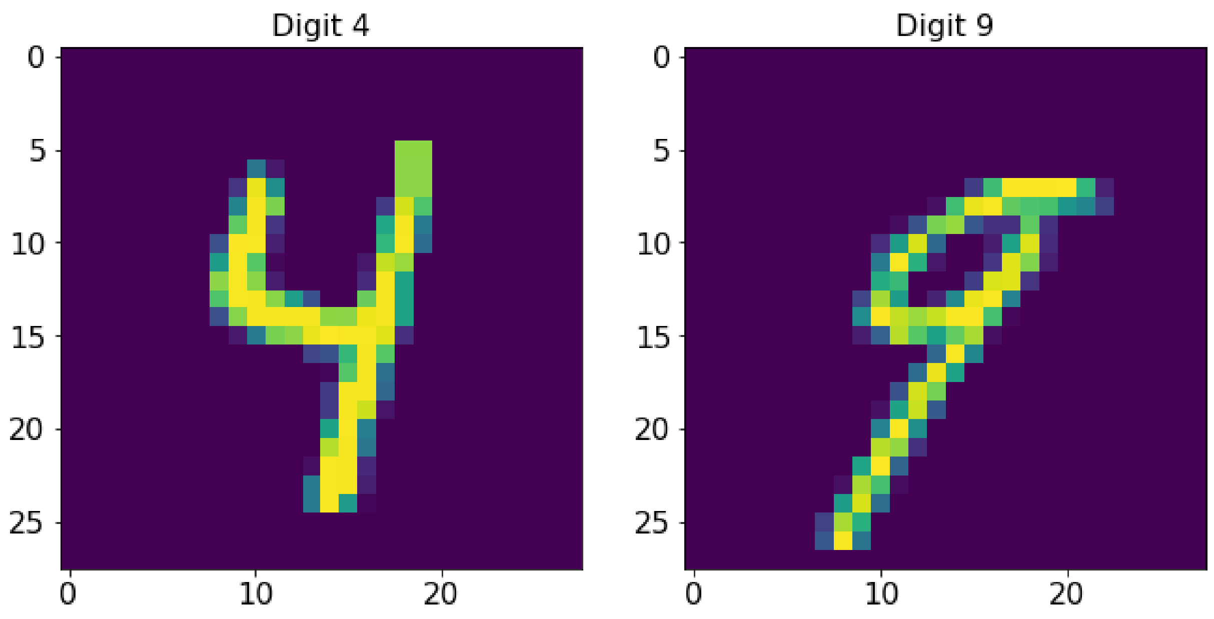
In [19]:
```python
## separating 4s and 9s select 800 points
# your code here
train_index = np.isin(y_train, [4,9])
test_index = np.isin(y_test, [4,9])

X_train = x_train[train_index][0:800,]
Y_train = y_train[train_index][0:800,]
X_test = x_test[test_index]
Y_test = y_test[test_index]
```

In [20]:
```
# Preprocess data using keras.utils.to_categorical
# your code here
from keras.utils import to_categorical

# Digit 4 has label 0, and digit 9 has label 9
Y_train[Y_train == 4] = 0
Y_train[Y_train == 9] = 1
Y_train = to_categorical(Y_train)

Y_test[Y_test == 4] = 0
Y_test[Y_test == 9] = 1
Y_test = to_categorical(Y_test)
```

In [21]:
```
# scale the data otherwise reLU can become unstable
# your code here
X_train = X_train.astype('float32')
X_train /= 255

X_test = X_test.astype('float32')
X_test /= 255
```

2.2 Data Exploration and Inspection: Use `imshow` to display a handwritten 4 and a handwritten 9.

In [22]:
```
# your code here
fig, ax = plt.subplots(1, 2, figsize=(11, 7))
ax[0].imshow(X_train[2])
ax[0].tick_params(labelsize=16)
ax[0].set_title('Digit 4', fontsize=16)

ax[1].imshow(X_train[0][:,:])
ax[1].tick_params(labelsize=16)
ax[1].set_title('Digit 9', fontsize=16)

plt.show()
```



2.3 Overfit an ANN: Build a fully connected network (FCN) using `keras` :

1. Nodes per Layer: 100,100,100,2 (<-the two-class 'output' layer)
2. Activation function: reLU
3. Loss function: binary_crossentropy
4. Output unit: Sigmoid
5. Optimizer: sgd (use the defaults; no other tuning)
6. Epochs: no more than 2,000
7. Batch size: 128
8. Validation size: .5

This NN trained on the dataset you built in 2.1 will overfit to the training set. Plot the training accuracy and validation accuracy as a function of epochs and explain how you can tell it is overfitting.

In [23]:

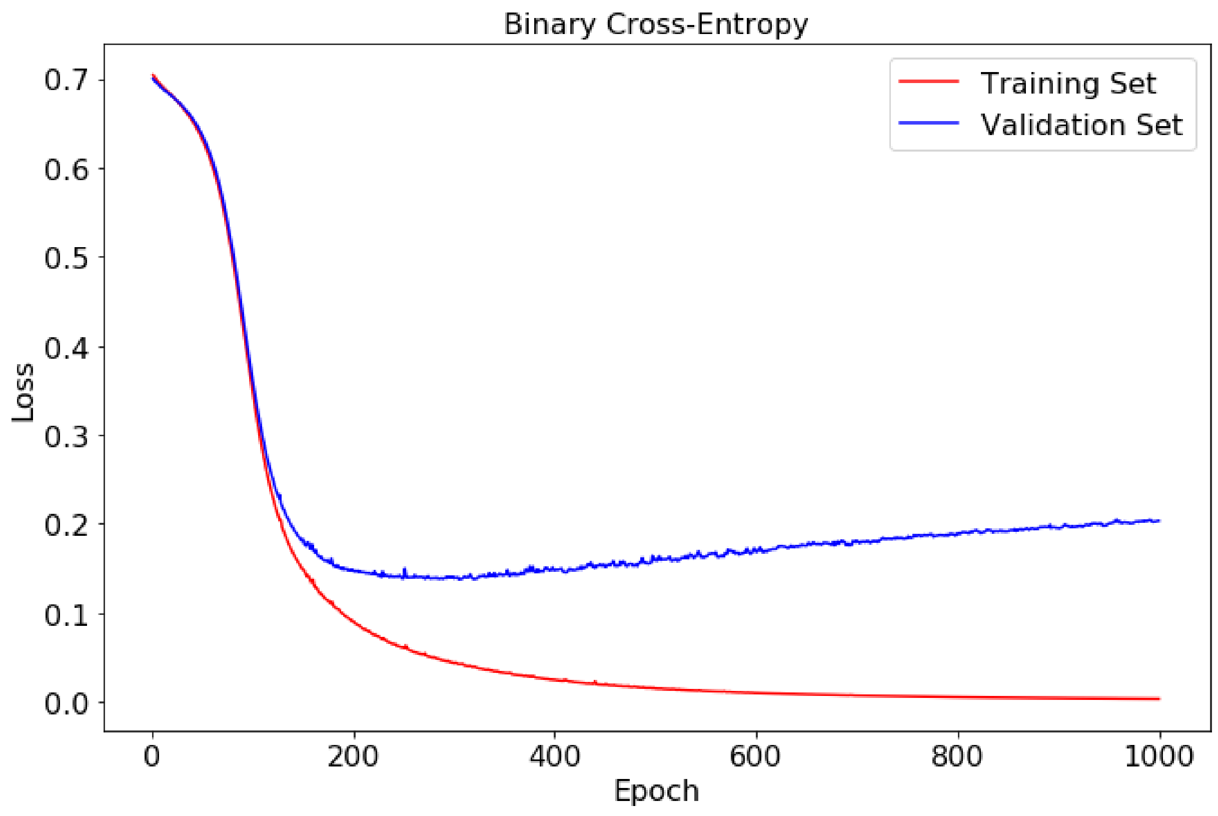
```
# your code here
X_train = X_train.reshape(800, 28*28)
H = 100 # number of nodes in the layer
input_dim = 28*28 # input dimension
output_dim = 2 # output dimension

model = Sequential() # create sequential multi-layer perceptron

# layer 0, our hidden layer
model.add(Dense(H, input_dim=input_dim, activation='relu'))

# layer 1, our hidden layer
model.add(Dense(H, activation='relu'))

# layer 2, our hidden layer
model.add(Dense(H, activation='relu'))

# layer 3
model.add(Dense(output_dim, activation='sigmoid'))

# compile the model
model.compile(loss='binary_crossentropy', optimizer='sgd')
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_7 (Dense)              (None, 100)               78500     
_________________________________________________________________
dense_8 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_9 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_10 (Dense)             (None, 2)                 202       
=================================================================
Total params: 98,902
Trainable params: 98,902
Non-trainable params: 0
_________________________________________________________________
```

In [24]:

```
# fit the model
epochs = 1000
batch_size = 128
validation_split = 0.5

model_history = model.fit(X_train, Y_train,
                          batch_size=batch_size,
                          epochs=epochs, verbose=False,
                          shuffle = True, validation_split=validation_split)
```

In [25]:
```
fig, ax = plt.subplots(1, 1, figsize=(11,7))
ax.plot(range(1, epochs + 1), model_history.history['loss'], 'r', label='Trai
ax.plot(range(1, epochs + 1), model_history.history['val_loss'], 'b', label='
ax.set_title('Binary Cross-Entropy', fontsize=16)
ax.set_xlabel('Epoch', fontsize=16)
ax.set_ylabel('Loss', fontsize=16)
ax.legend(fontsize=16)
ax.tick_params(labelsize=16)
```



In [26]:
```
# your code here
train_score = model.evaluate(X_train, Y_train, verbose=False)
print('Train loss = {:.3f}'.format(train_score))
print('Train R2 = {:.2%}'.format(r2_score(Y_train, model.predict(X_train))))

X_test = X_test.reshape(-1, 28*28)
test_score = model.evaluate(X_test, Y_test, verbose=False)
print('Train loss = {:.3f}'.format(test_score))
print('Train R2 = {:.2%}'.format(r2_score(Y_test, model.predict(X_test))))
```

```
Train loss = 0.103
Train R2 = 91.15%
Train loss = 0.158
Train R2 = 84.51%
```

Your answer here

The loss on the validation set first goes down, and then starts to go up at around 200 epochs, which is a sign that the model starts to overfit.

2.4 Explore Regularization: Your task is to regularize this FCN. You are free to explore any method or combination of methods. If you are using anything besides the methods we have covered in class, give a citation and a short explanation. You should always have an understanding of the methods you are using.

Save the model using `model.save(filename)` and submit in canvas along with your notebook.

We will evaluate your model on a test set we've kept secret.

1. Don't try to use extra data from NMIST. We will re-train your model on training set under the settings above.
2. Keep the architecture above as is. In other words keep the number of layers, number of nodes, activation function, and loss fucntion the same. You can change the number of epochs (max 2000), batch size, optimizer and of course add elements that can help to regularize (e.g. drop out, L2 norm etc). You can also do data augmentation.
3. You *may* import new modules, following the citation rule above.

Grading: Your score will be based on how much you can improve on the test score via regularization:

1. (0-1] percent will result into 10 pts
2. (1-2] percent will result into 20 pts
3. (2-3] percent will result into 30 pts
4. Above 3 percent will result in 35 pts
5. Top 15 groups or single students will be awarded an additional 10 pts
6. The overall winner(s) will be awarded an additional 5 pts

Reshape image in 3 dimensions

In [27]:
```
X_train = X_train.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

print("Shape of X_train: {}".format(X_train.shape))
print("Shape of Y_train: {}".format(Y_train.shape))
print("Shape of X_test: {}".format(X_test.shape))
print("Shape of Y_test: {}".format(Y_test.shape))
```

```
Shape of X_train: (800, 28, 28, 1)
Shape of Y_train: (800, 2)
Shape of X_test: (1991, 28, 28, 1)
Shape of Y_test: (1991, 2)
```

Define the Model

- The first layer is a convolutional (Conv2D) layer (citation: https://keras.io/layers/convolutional/) (https://keras.io/layers/convolutional/)). I choose to set 32 filters for the first conv2D layer, and filters can be seen as a transformation of the image. The CNN can isolate features that are useful everywhere from these transformed images.
- MaxPool2D (citation: https://keras.io/layers/pooling/ (https://keras.io/layers/pooling/)) acts as a downsampling filter, which looks at the two neighboring pixels and picks the maximal value.

Pooling is used to reduce computational cost and reduce overfitting.
- Dropout (citation: [https://keras.io/layers/core/#dropout (https://keras.io/layers/core/#dropout)](https://keras.io/layers/core/#dropout)) is a regularization method, where a proportion of nodes in the layer are randomly ignored (setting their wieghts to zero) for each training sample. This drops randomly a propotion of the network and forces the network to learn features in a distributed way. This technique also improves generalization and reduces the overfitting.
- The second layer is also a convolutional (Conv2D) layer, with 64 filters.
- The third layer is a fully-connected (Dense) layer.

In [28]:
```
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D

model = Sequential()

# Layer 1
model.add(Conv2D(filters = 32, kernel_size = (5,5), padding = 'Same',
                 activation ='relu', input_shape = (28,28,1)))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

# Layer 2
model.add(Conv2D(filters = 64, kernel_size = (3,3), padding = 'Same',
                 activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.25))

# Layer 3
model.add(Flatten())
model.add(Dense(100, activation = "relu"))
model.add(Dropout(0.5))

# Output layer
model.add(Dense(2, activation = "sigmoid"))
```

Set the optimizer and annealer

- I choose RMSprop (with default values) as the optimizer (Citation: [https://keras.io/optimizers/) (https://keras.io/optimizers/)](https://keras.io/optimizers/)), becaues it is usually a good choice for recurrent neural networks.
- I choose ReduceLROnPlateau as the annealer (Citation: [https://keras.io/callbacks/) (https://keras.io/callbacks/)](https://keras.io/callbacks/)), because it reduces learning rate when a metric has stopped improving.

In [29]:
```
# Define the optimizer
from keras.optimizers import RMSprop
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
```

In [30]:
```
# Compile the model
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=["accu
```

In [31]:
```
# Set a learning rate annealer
from keras.callbacks import ReduceLROnPlateau
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
```

In [32]:
```
epochs = 200
batch_size = 128
```

Data augmentation

In order to avoid overfitting problem, we need to expand artificially our handwritten digit dataset. The idea is to alter the training data with small transformations to reproduce the variations occuring when someone is writing a digit. For the data augmentation, I choose to :

- Randomly rotate some training images by 10 degrees
- Randomly zoom by 10% some training images
- Randomly shift images horizontally by 10% of the width
- Randomly shift images vertically by 10% of the height
- Randomly flip images horizontally
- Randomly flip images vertically

In [33]:
```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the d
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in the range (degrees, 0
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fractio
        height_shift_range=0.1,  # randomly shift images vertically (fraction
        horizontal_flip=True,  # randomly flip images
        vertical_flip=True)  # randomly flip images
```

In [34]:
```
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)
```

```
(800, 28, 28, 1)
(800, 2)
(1991, 28, 28, 1)
(1991, 2)
```

In [35]:
```
# Fit the model
history = model.fit_generator(datagen.flow(X_train,Y_train, batch_size=batch_
                              epochs = epochs, validation_data = (X_test, Y_t
                              verbose = 2, steps_per_epoch=X_train.shape[0] /
                              , callbacks=[learning_rate_reduction])
```

```
 - 1s - loss: 0.1435 - acc: 0.9494 - val_loss: 0.0519 - val_acc: 0.9834

Epoch 187/200
 - 2s - loss: 0.1648 - acc: 0.9422 - val_loss: 0.0517 - val_acc: 0.9829
Epoch 188/200
 - 1s - loss: 0.1569 - acc: 0.9348 - val_loss: 0.0517 - val_acc: 0.9829
Epoch 189/200
 - 2s - loss: 0.1738 - acc: 0.9351 - val_loss: 0.0517 - val_acc: 0.9832
Epoch 190/200
 - 2s - loss: 0.1440 - acc: 0.9557 - val_loss: 0.0517 - val_acc: 0.9829
Epoch 191/200
 - 2s - loss: 0.1587 - acc: 0.9448 - val_loss: 0.0518 - val_acc: 0.9829
Epoch 192/200
 - 1s - loss: 0.1987 - acc: 0.9332 - val_loss: 0.0518 - val_acc: 0.9829
Epoch 193/200
 - 2s - loss: 0.1461 - acc: 0.9494 - val_loss: 0.0518 - val_acc: 0.9829
Epoch 194/200
 - 2s - loss: 0.1596 - acc: 0.9432 - val_loss: 0.0518 - val_acc: 0.9832
Epoch 195/200
 - 2s - loss: 0.1652 - acc: 0.9382 - val_loss: 0.0517 - val_acc: 0.9832
```

In [38]:
```
# your code here
train_metrics = model.evaluate(X_train, Y_train, verbose=False)
print('Train loss = {:.3f}'.format(train_metrics[0]))
print('Train R2 = {:.2%}'.format(train_metrics[1]))

test_metrics = model.evaluate(X_test, Y_test, verbose=False)
print('Test loss = {:.3f}'.format(test_metrics[0]))
print('Test R2 = {:.2%}'.format(test_metrics[1]))
```

```
Train loss = 0.070
Train R2 = 98.06%
Test loss = 0.052
Test R2 = 98.32%
```

In [37]:
```
# Save the model
model.save('model')
```

In [ ]:
```

```