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-CS109A/m
        aster/content/styles/cs109.css").text
        HTML(styles)
Out[1]:
```

Import libraries:

In [2]:
```
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

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

from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from keras.layers import Dropout
from keras.constraints import maxnorm
from keras.callbacks import EarlyStopping
from keras.preprocessing.image import ImageDataGenerator
```

```
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: $\mathcal{F}(s) = \int_{-\infty}^{\infty} f(x)e^{-2\pi ixs}dx$. 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.**

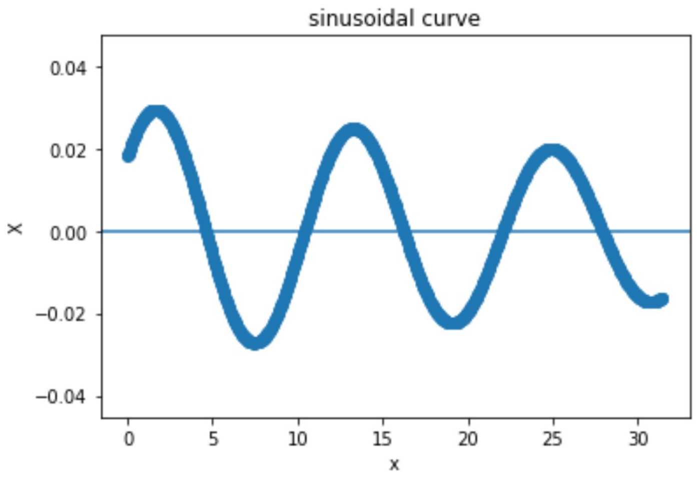
In [3]:
```
def plot_predictions(model, test_x, test_y, count=None):
    # Model - a Keras or SKlearn model that takes in (n,1000) training data and pr
edicts (n,4) output data
    # test_x - a (n,1000) input dataset
    # test_y - a (n,4) output dataset
    # This function will plot the sine curves in the training data and those impli
ed by the model's predictions.
    # 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_points))
        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 Transform
")
        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]:
```
def plot_X(X, i):
    x = np.linspace(0, 10*np.pi,X.shape[1])
    plt.scatter(x, X[i])
    plt.axhline(y=0)
    plt.xlabel('x')
    plt.ylabel('X')
    plt.title('sinusoidal curve')

plot_X(X_train, 0)
```



The plot shows a sinusoidal cuve of amplitude around 0.02, wavelength around 12.

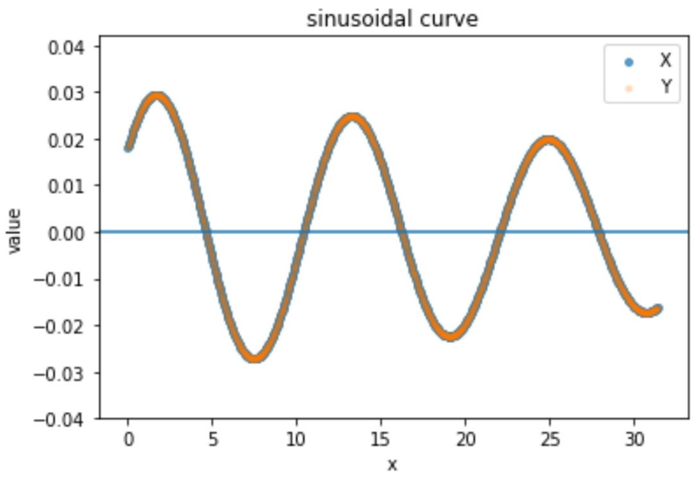
1.2 The first row of the `y_train` data is $[0.024, 0.533, 0.018, 0.558]$. Visually or numerically verify that the first row of X_train is 1000 equally-spaced points in $[0, 10\pi]$ from the function $f(x) = 0.024\sin(0.533\,x) + 0.018\cos(0.558\,x)$...

In [6]:
```
def f(x, y):
    a = y[0]
    b = y[1]
    c = y[2]
    d = y[3]
    return a*np.sin(b*x) + c*np.cos(d*x)
```

In [7]:
```
x = np.linspace(0, 10*np.pi,1000)
y = f(x, y_train[0])
y.shape
```

Out[7]:
```
(1000,)
```

In [8]:
```
def plot_X(X, y, i):
    x = np.linspace(0, 10*np.pi,X.shape[1])
    y_est = f(x, y[0])
    plt.scatter(x, X[i], label='X', s=15, alpha=0.7)
    plt.scatter(x, y_est, label='Y', s=10, alpha=0.2)
    plt.axhline(y=0)
    plt.xlabel('x')
    plt.ylabel('value')
    plt.title('sinusoidal curve')
    plt.legend()

plot_X(X_train, y_train, 0)
```



The plot shows that the first row X of X_train are points sampled from the function Y=f(X). Bpth X and f(X) match perfectly.

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 [9]:
```
# function that create a FCN using the provided parameters
def create_nn(input_dim, output_dim, nb_nodes=1, nb_hidden_layer=0, hidden_activat
ion='relu', output_activation='linear'):
    model = Sequential()
    # input layer
    model.add(
        Dense(nb_nodes, input_dim=input_dim, activation=hidden_activation)
    )
    # hidden layers
    for i in range(nb_hidden_layer):
        model.add(
            Dense(nb_nodes, activation=hidden_activation)
        )
    # output layer
    model.add(
        Dense(output_dim, activation=output_activation)
    )
    return model
```

In [10]:
```
# we create 10 hidden layers, each with 100 nodes
model = create_nn(X_train.shape[1], y_train.shape[1], 100, 10, hidden_activation='
relu', output_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 [11]:
```
def compile_nn(model, loss='mean_absolute_error', optimizer='adam'):
    model.compile(loss=loss, optimizer=optimizer)
    model.summary()
```

In [12]:
```
compile_nn(model)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 100)               100100    
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_4 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_5 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_6 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_7 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_8 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_9 (Dense)              (None, 100)               10100     
_________________________________________________________________
dense_10 (Dense)             (None, 100)               10100     
_________________________________________________________________
dense_11 (Dense)             (None, 100)               10100     
_________________________________________________________________
dense_12 (Dense)             (None, 4)                 404       
=================================================================
Total params: 201,504
Trainable params: 201,504
Non-trainable params: 0
_________________________________________________________________
```

In [13]:
```
X_train.shape, y_train.shape
```

Out[13]:
```
((10000, 1000), (10000, 4))
```

The output shape of the input and hidden layers is (None, 100). None is a placeholder for the number of samples presented to the layer. 100 refers to the number of nodes in the layer.

The input vector has 1000 features which are fed into the 100 nodes of the input layer, this makes 1000 x 100 weights plus 100 bias (1 per node), what explains the number of parameters 100100 for the input layer.

Each hidden layer receives an input from each of the 100 nodes in the previous layer, this makes 100 x 100 weights plus 100 bias; this makes in total 10100 parameters.

The output layer has 4 nodes, each receiving 100 inputs from the last hidden layer, adding 4 bias, we have 404 parameters.

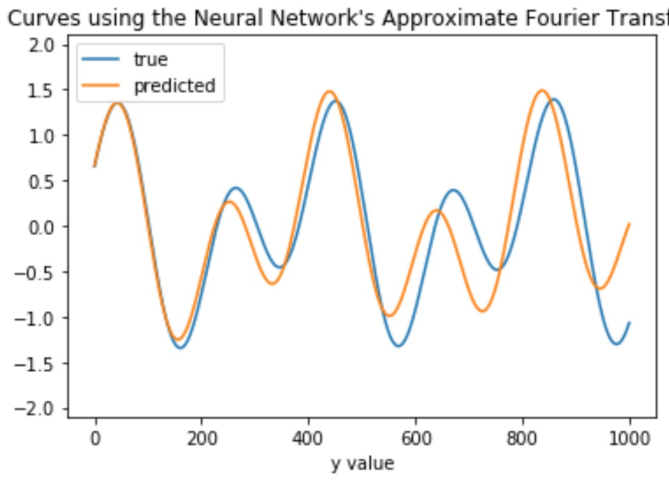
1.5 Fit your model to the data for 50 epochs using a batch size of 32 and a validation split of .2. You can train for longer if you wish- the fit tends to improve over time.

```
In [14]: def fit_nn(model, X, y, batch_size=32, epochs=50, validation_split=.2):
             return model.fit(X, y, batch_size=batch_size, epochs=epochs, validation_split=
         validation_split, verbose=False)
```

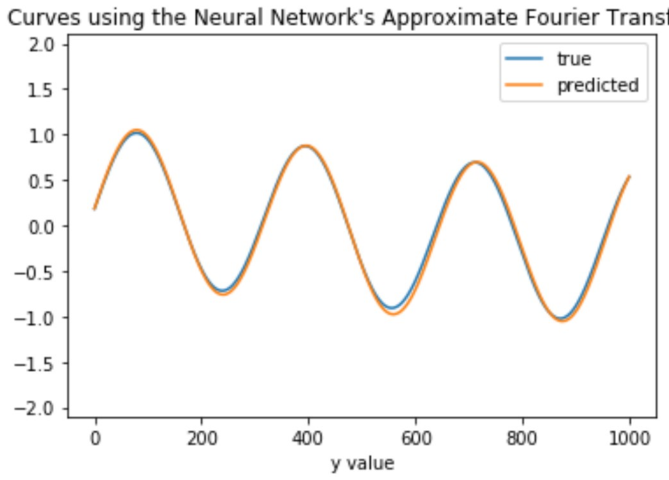
```
In [15]: model_history = fit_nn(model, X_train, y_train)
```

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?

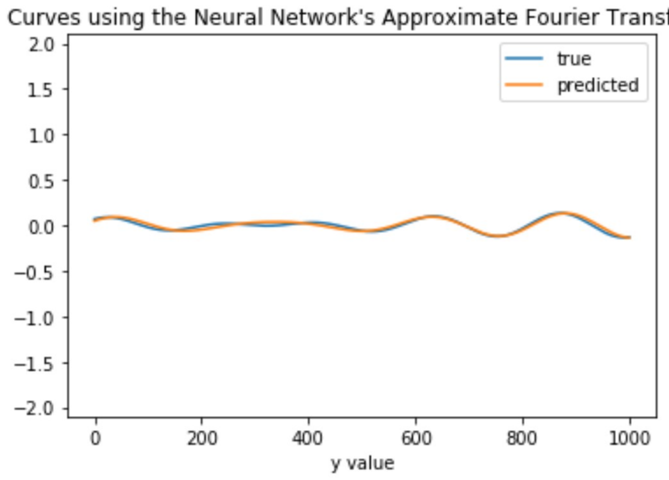
In [16]:
```
plot_predictions(model, X_test, y_test)
```



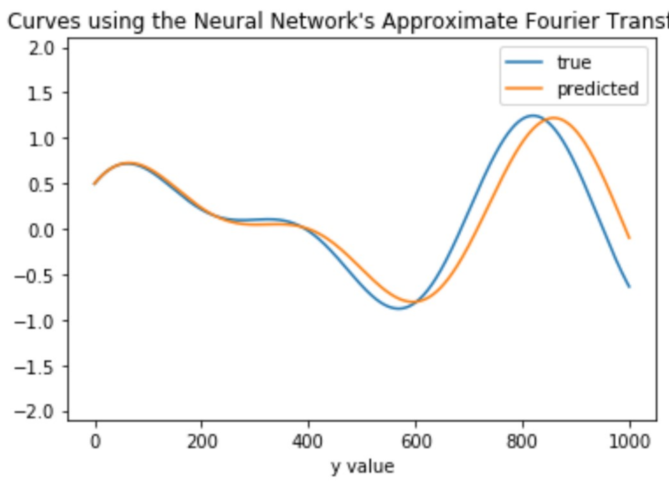
```
true: [0.86199664 0.98175913 0.65523998 0.4870337 ]
predicted: [0.8348559  1.0174758  0.6652667  0.47162244]
```



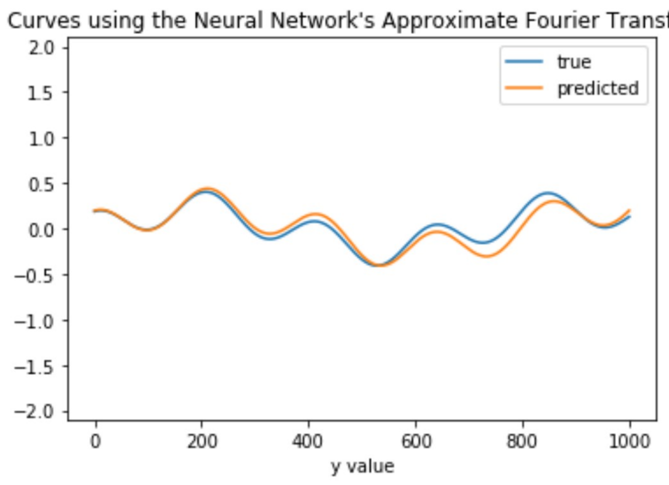
```
true: [0.8406355  0.63159555 0.18328701 0.11174618]
predicted: [0.872184   0.6281548  0.1866716  0.12358321]
```



```
true: [0.06591224 0.75183886 0.06986143 0.91352303]
predicted: [0.08727003 0.74575603 0.04975602 0.89700496]
```



```
true: [0.75610725 0.30861152 0.49522059 0.48394499]
predicted: [0.7484355  0.29891375 0.49923593 0.45798746]
```



```
true: [0.2229353  0.27885697 0.18696198 0.94846283]
predicted: [0.24443647 0.25049517 0.19568534 0.94208986]
```

In [17]:
```
# helper function for plotting train vs validation of a fitted FCN
def plot_nn_accuracy(model_history, loss='Mean Absolute Error', metric='loss', fig
size=(10,5)):
    fig, ax = plt.subplots(1, 1, figsize=figsize)
    plt.plot(model_history.history[metric], '-o', label = 'train')
    plt.plot(model_history.history['val_'+metric], '-*', label = 'val')
    plt.xlabel('epochs')
    plt.ylabel(loss)
    plt.legend();

    print("{}: train={:.1f}, val={:.1f}".format(loss,
        model_history.history[metric][-1],
        model_history.history['val_'+metric][-1]))
```

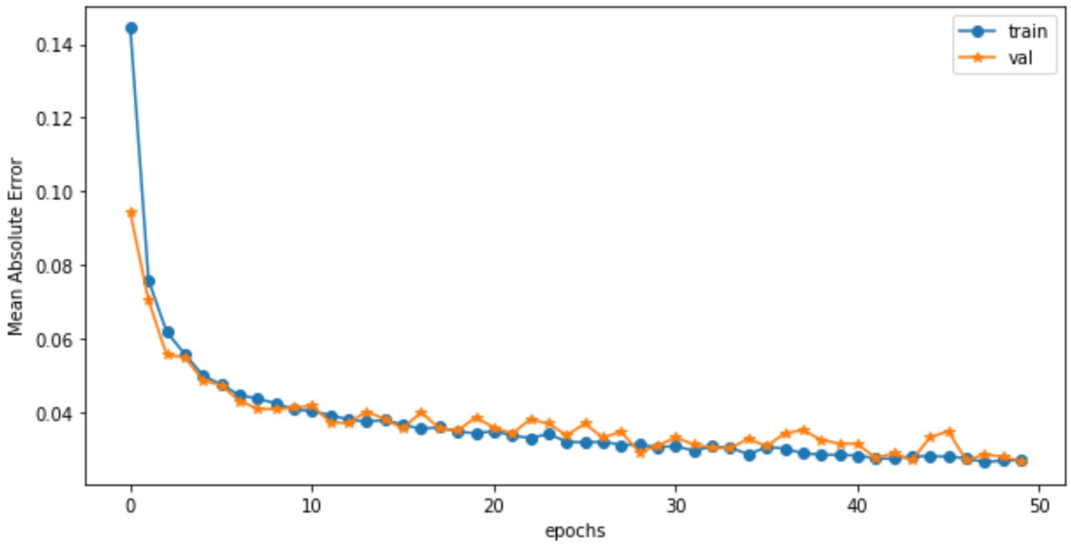
In [18]:
```
print("Overall loss on test set")
model.evaluate(X_test, y_test)
```

```
Overall loss on test set
2000/2000 [==============================] - 0s 64us/step
```

Out[18]:
```
0.02587362512946129
```

```
In [19]: plot_nn_accuracy(model_history)
```

```
Mean Absolute Error: train=0.0, val=0.0
```



We observer that the loss on test is ca. 25% which is not so good. The validation and train loss show a similar trend on the plot above, however with higher variability for validation. Around epoch 12, the model appears not to learn and overfitting starts.

We cannot assume that the network has accurately learned from the training set. The features were not standardized between zero and one; their distribution might be skewed. The network might be overfitting to the training set and therefore is not generalizing very well to the unseen test data. Regularization might be required.

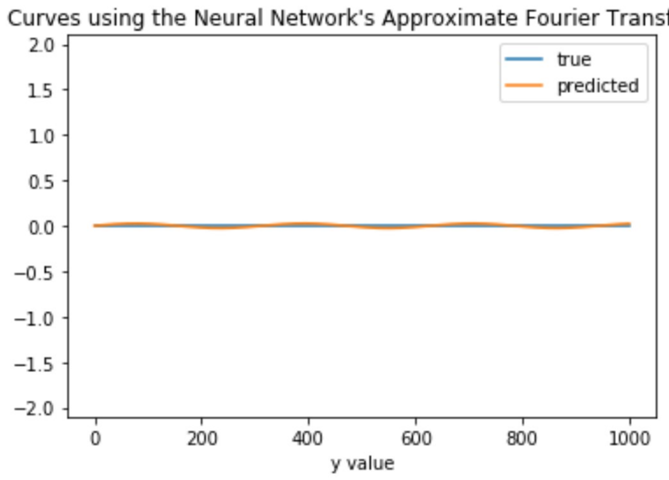
1.7 Examine the model's performance on the 9 train/test pairs in the `extended_test` variables. Which examples does the model do well on, and which examples does it struggle with?

```
In [20]: print("Overall loss on extended test set")
         model.evaluate(X_extended_test, y_extended_test)
```

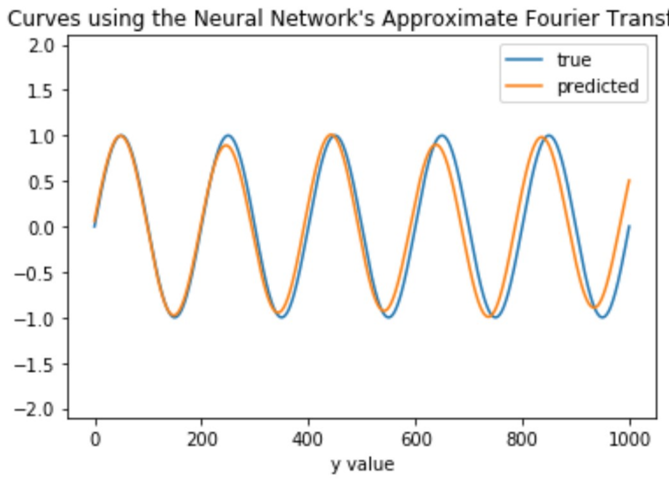
```
Overall loss on extended test set
9/9 [==============================] - 0s 332us/step
```

```
Out[20]: 0.4807286858558655
```

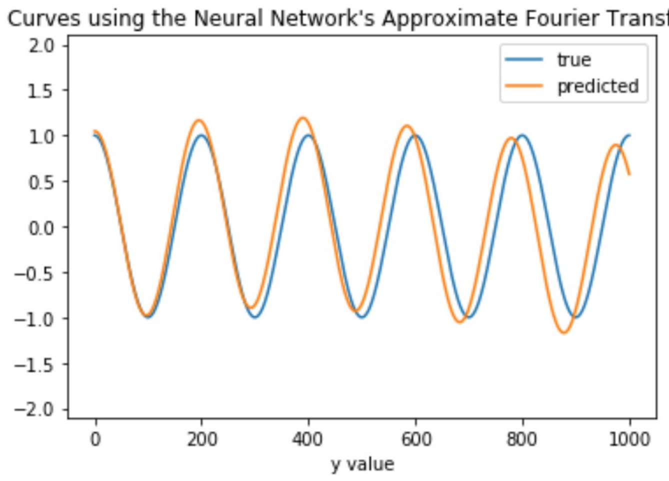
In [21]:
```
plot_predictions(model, X_extended_test, y_extended_test, 9)
```



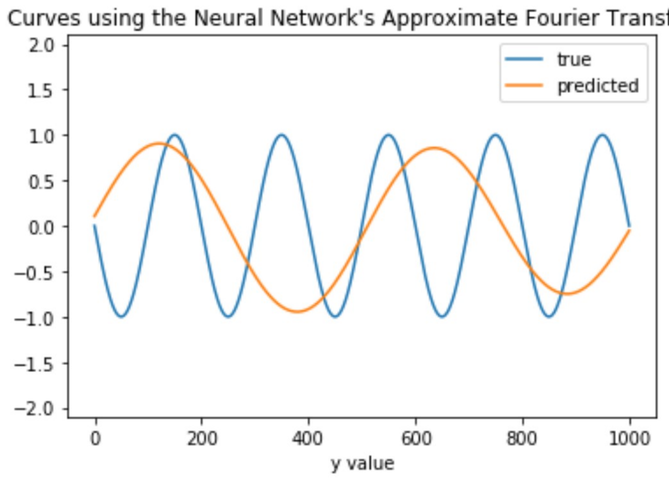
```
true: [0. 0. 0. 0.]
predicted: [0.02574775 0.63762355 0.00063868 0.5355149 ]
```



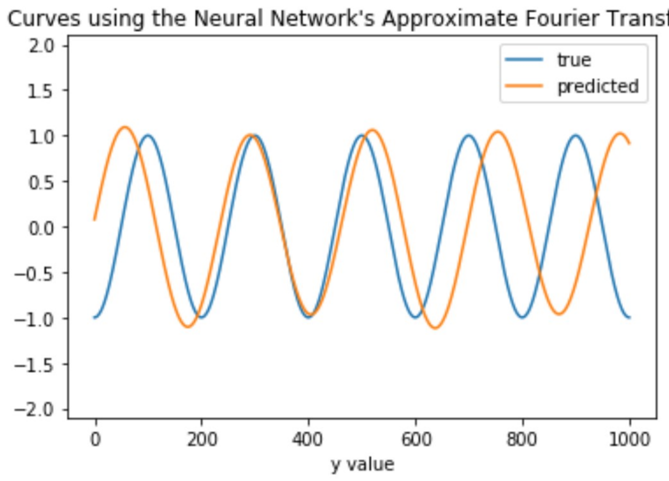
```
true: [1. 1. 0. 0.]
predicted: [0.949822  1.0171212 0.060489  0.440173 ]
```



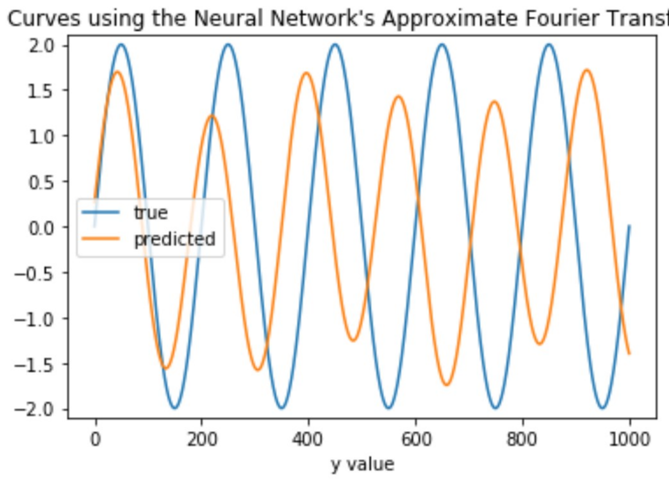
```
true: [0. 0. 1. 1.]
predicted: [0.15202825 0.14895427 1.0456022  1.0255867 ]
```



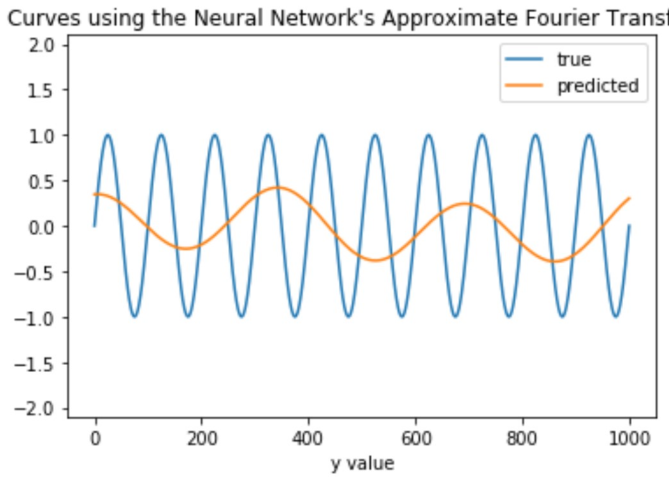
```
true: [-1.  1.  0.  0.]
predicted: [0.8435298  0.3971761  0.10585264 0.24347547]
```



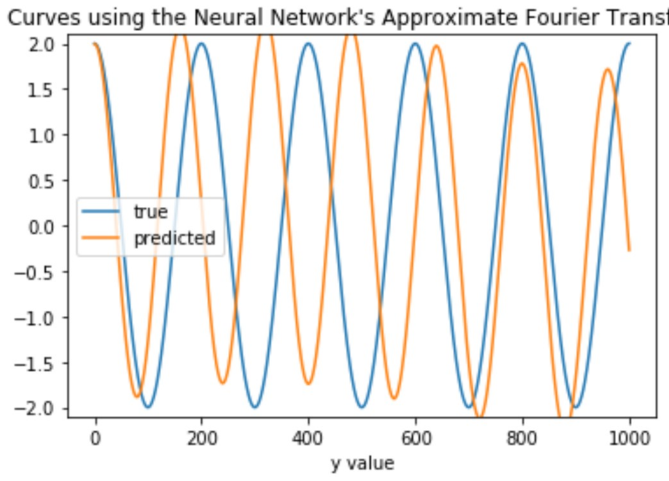
```
true: [ 0.  0. -1.  1.]
predicted: [1.0406052  0.8633865  0.07733119 0.46491125]
```



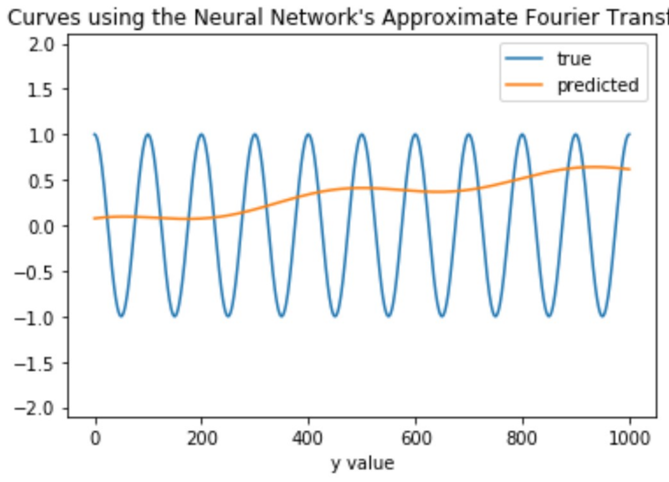
```
true: [2. 1. 0. 0.]
predicted: [1.4817227  1.1391217  0.26508698 0.4502153 ]
```



```
true: [1. 2. 0. 0.]
predicted: [0.10170594 0.21368644 0.34564567 0.5770926 ]
```



```
true: [0. 0. 2. 1.]
predicted: [0.2783688  0.15979642 1.9937022  1.250087  ]
```



```
true: [0. 0. 1. 2.]
predicted: [0.66066456 0.03544251 0.07648376 0.43973935]
```

The overall loss on the extended test dataset is much worse, ca. 50%. The model is strugglying with samples where either the cosine or the sine component is missing. This happens when a=b=0 and/or c=d=0. These are pure sine and cosine functions.

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?

The model could not learn the special cases when the function to learn is made just of a sine or just of a cosine component. Such special cases where not covered by the orginal X_train data.

The model overfitted to the X_train data where both sine and cosine components were always provided. All weights and bias were estimated to fit that data. Without regularizing those parameters, we end up with a model, which could not be generalize to simple Fourier transformations with pure components.

Regulrizing 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.

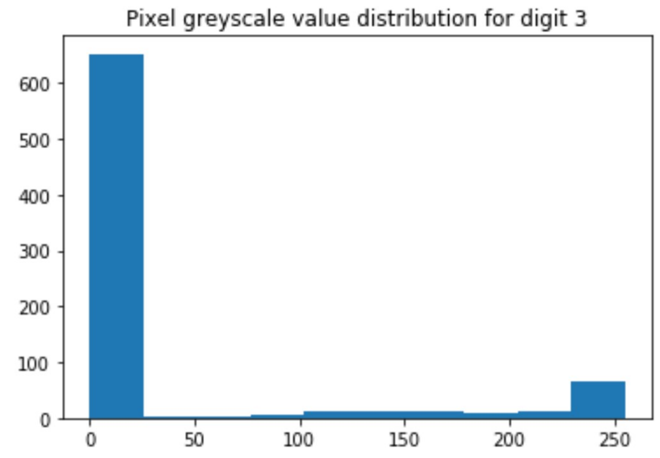
Please refer to the code below to process the data.

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 [22]:
```
random_state = 1
```

In [23]:
```
## 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 [24]:
```
# histogramm of the pixels
plt.hist(x_train[0].reshape(28*28))
plt.title('Pixel greyscale value distribution for digit {}'.format(y_train[0]));
```



As shown above the greyscales range from 0 to 255; most pixel have values close to zero because of the background. The distribution is right skweed and would cause issue when training the network, which could get stucked in local minima. Therefore we need to standardize the features between 0 and 1. We further need to reshape the features from (28,28) into one input vector (784,) for training our network.

Since we will consider only the digits 4 and 9, the output of our network should be a categorical vector (2,).

Helper function for preparing the data

In [25]:
```
# returns a random subset of nb_samples NMIST samples corresponding to the given d
igits
def get_data_subset(X, y, digits=[4, 9], nb_samples=800):
    mask = np.isin(y, digits)
    X_subset = X[mask]
    y_subset = y[mask]
    if nb_samples == -1:
        return X_subset, y_subset
    idx = np.random.choice(y_subset.shape[0], nb_samples, replace=False)
    return X_subset[idx], y_subset[idx]

# normalize scale features X to a 1D vector
# turn the response y into a binary categorical representation (2,)
def get_scaled_data(X, y):
    X = X.astype('float32')
    X_scaled  = keras.utils.normalize(X)
    #X_scaled  /= 255
    X_scaled_reshaped = X_scaled.reshape(-1, 28*28)
    y_scaled = keras.utils.np_utils.to_categorical(y)
    y_scaled = y_scaled.T[~np.all(y_scaled == 0, axis=0)].T
    return X_scaled_reshaped, y_scaled
```

In [26]:
```
# prepare a random subset of 800 training samples labelled 4 or 9
x_train_subset, y_train_subset = get_data_subset(x_train, y_train, digits=[4, 9],
nb_samples=800)

# prepare a random subset of 1991 test samples labelled 4 or 9
x_test_subset, y_test_subset = get_data_subset(x_test, y_test, digits=[4, 9], nb_s
amples=-1)
```

In [27]:
```
# verify the shape of the train data subset
x_train_subset.shape, y_train_subset.shape
```

Out[27]:
```
((800, 28, 28), (800,))
```

In [28]:
```
# verify the shape of the test data subset
x_test_subset.shape, y_test_subset.shape
```

Out[28]:
```
((1991, 28, 28), (1991,))
```

In [29]:
```
# scale the training features to a 1D vector with [0..1] values range, and respons
e to category 0 or 1
x_train_subset_scaled,  y_train_subset_scaled = get_scaled_data(x_train_subset, y_
train_subset)

# scale the test features to a 1D vector with [0..1] values range, and response to
category 0 or 1
x_test_subset_scaled,  y_test_subset_scaled = get_scaled_data(x_test_subset, y_tes
t_subset)
```

In [30]:
```
# verify the shape of the scaled train data subset
x_train_subset_scaled.shape, y_train_subset_scaled.shape
```

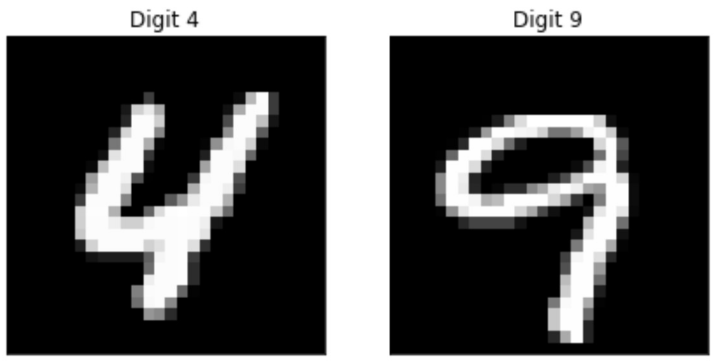
Out[30]:
```
((800, 784), (800, 2))
```

In [31]:
```
# verify the shape of the scaled test data subset
x_test_subset_scaled.shape, y_test_subset_scaled.shape
```

Out[31]:
```
((1991, 784), (1991, 2))
```

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

In [32]:
```
fig = plt.figure()
plt.subplot(1,2,1)
plt.tight_layout()
plt.imshow(x_train_subset[y_train_subset==4][0], cmap='gray', interpolation='none'
)
plt.title('Digit 4')
plt.xticks([])
plt.yticks([])
plt.subplot(1,2,2)
plt.tight_layout()
plt.imshow(x_train_subset[y_train_subset==9][0], cmap='gray', interpolation='none'
)
plt.title('Digit 9')
plt.xticks([])
plt.yticks([]);
```



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 1,000**
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.

Helper functions

In [33]:
```
# Build and compile a FCN without regularization
def build_model_overfit(optimizer='sdg'):
        model = Sequential([
            Dense(100, input_shape=(784,), activation='relu'),
            Dense(100, activation='relu'),
            Dense(100, activation='relu'),
            Dense(2, activation='sigmoid')
        ])
        model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['a
ccuracy'])
        return model
```

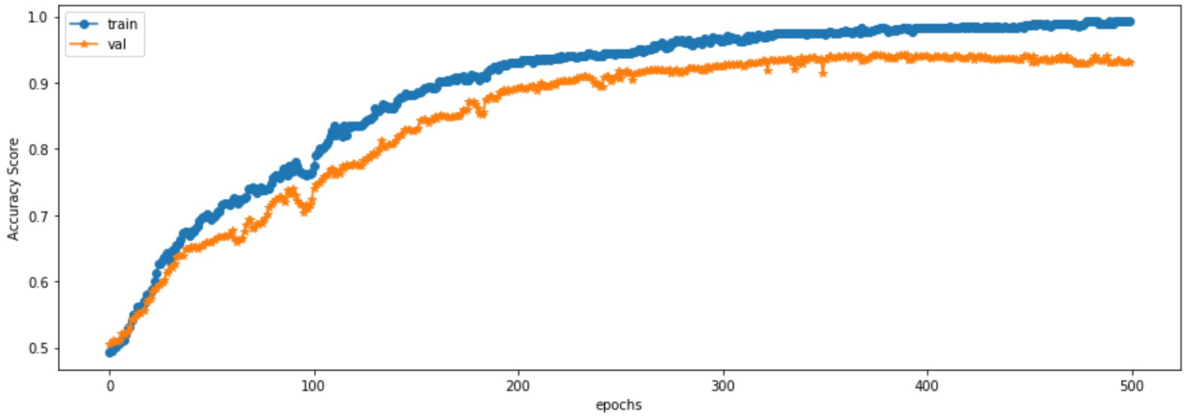
In [34]:
```
# generic function that fits a FCN with given parameters
# evaluates the fitted FCN on test set
# reports training, validation and test accuracy scores
def build_fit_evaluate_nn(build_model, Xtrain, ytrain, Xtest, ytest,
                          epochs=500, batch_size=128,
                          callbacks = None,
                          optimizer='sgd',
                          verbose=False):
    np.random.seed(5)
    model = build_model(optimizer=optimizer)
    model_history = model.fit(Xtrain, ytrain,
                                        epochs=epochs, batch_size=batch_size,
                                        validation_split=.5,
                                        callbacks = callbacks,
                                        verbose=verbose)
    plot_nn_accuracy(model_history, 'Accuracy Score', metric='acc', figsize=(15,5)
)
    accuracy_score_train = model_history.history['acc'][-1]
    accuracy_score_val = model_history.history['val_acc'][-1]
    accuracy_score_test = model.evaluate(Xtest, ytest)[1];
    print("Accuracy Score - Train: {0:.2%}".format(accuracy_score_train))
    print("Accuracy Score - Val: {0:.2%}".format(accuracy_score_val))
    print("Accuracy Score - Test: {0:.2%}".format(accuracy_score_test))
    return {'train': [accuracy_score_train], 'val': [accuracy_score_val], 'test':
[accuracy_score_test]}
```

In [35]:
```
# function for reporting scores
def report_scores(df, label, scores):
    scores['regularization'] = label
    scores['improvment'] = str(round((scores['test'][0]/df[df['regularization']=='
none']['test'][0] - 1)*100,2))+'%'
    df = df[df.regularization != label]
    df = df.append(pd.DataFrame(scores, columns=['regularization', 'train', 'val',
'test','improvment']))
    df = df.drop_duplicates()
    return df
```

Overfitted FCN

In [36]:
```
accuracy_score_overfit = build_fit_evaluate_nn(build_model_overfit,
                                               x_train_subset_scaled, y_train_subs
et_scaled,
                                               x_test_subset_scaled, y_test_subset
_scaled);
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 48us/step
Accuracy Score - Train: 99.25%
Accuracy Score - Val: 93.12%
Accuracy Score - Test: 92.54%
```



In [37]:
```
accuracy_score_overfit['regularization'] = 'none'
df_scores = pd.DataFrame(accuracy_score_overfit, columns=['regularization', 'train
', 'val', 'test', 'improvment'])
df_scores
```

Out[37]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN

We observe that, unlike the training accuracy that increases monotonically, the validation accuracy increases to a maximum, and then decreases for a while. As we move towards the right (increasing epochs) the accuracy on training keeps improving but the accuracy on validation does not.

Based on this observation we can make an interpretation that probably during the first phase of the training the model was improving its overall fit, but then as it continues training the training loss decreases, but the validation accuracy starts decresing. This is a sign where we are probably starting to overfit to our training set.

Ideally if we are going to do early stopping, we probably want to stop around epoch where the trend changes.

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.

Motivation

Our network from question 2.3 is very complex (with 10 of thousands of weights). This makes is very prone to overfitting. Regularization is a technique to make modification in the learning algorithm of our network so that the network will generalize better. We do so by penalizing the weights of the network. For example if the penalty is so high that some of the weights are nearly equal to zero, than this would result in a model closer to a simple linear model. Our goal is to find the optimal value of regularization.

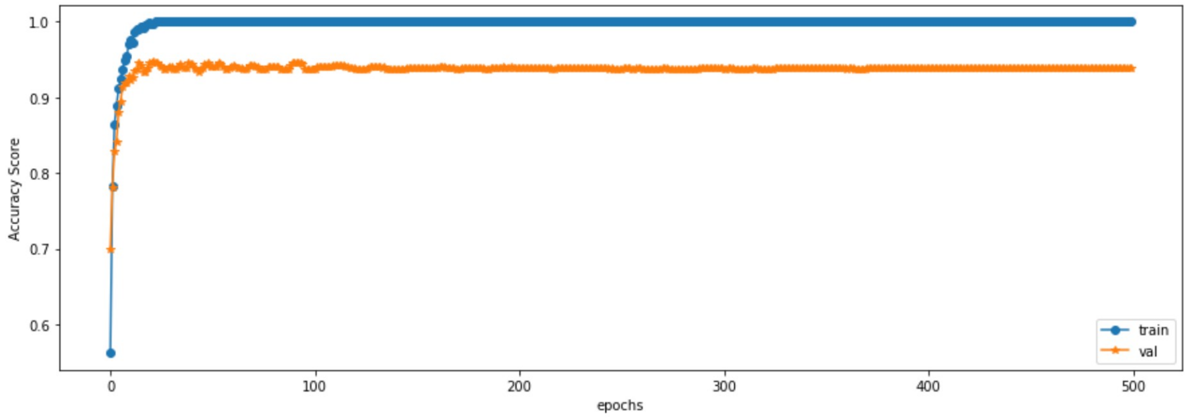
We tried to find the optimal value of some hyperparameters using GridSearchCV (cross-validation). We will not present the results here because it took hours to run and most searches did not end. Our approach here is based on intuition and try/fail.

Optimizer

Normally we would choose an optimization approach and tune the hyper-parameters to fit that approach. Here we investigate whether stochastic gradient descent or adam would be the best apriori approach for our problem.

In [38]:
```
accuracy_score_adam = build_fit_evaluate_nn(build_model_overfit,
                                              x_train_subset_scaled, y_train_subs
et_scaled,
                                              x_test_subset_scaled, y_test_subset
_scaled,
                                              optimizer='adam');
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 46us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 93.88%
Accuracy Score - Test: 94.78%
```



In [39]:
```
df_scores = report_scores(df_scores, 'optimizer Adam', accuracy_score_adam)
df_scores
```

Out[39]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%

We observe that Adam optimizer provides a slightly improve in test accuracy score, compare to stochastic gradient descent. We will continue investigating further regularization while using Adam in the next steps.

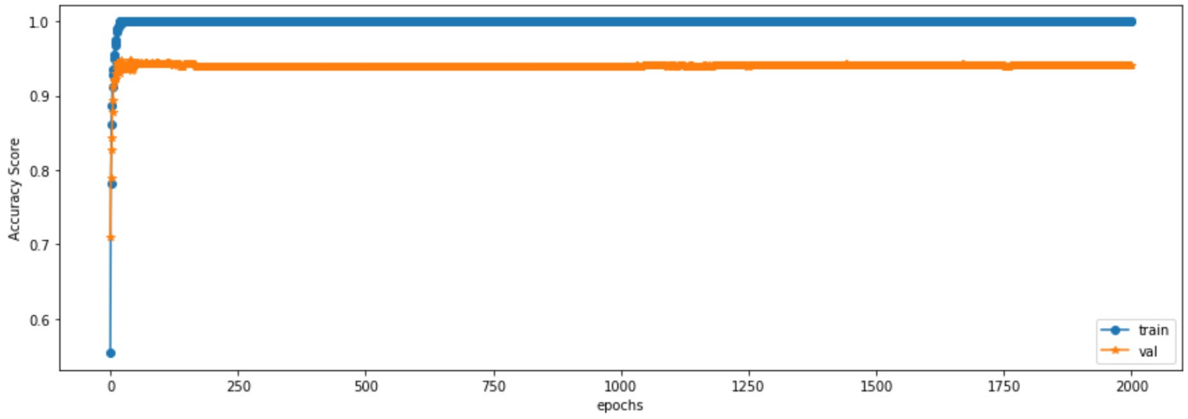
Number of Epochs

The number of epochs is the number of times the entire training set is used during training. One epoch is when an entire dataset is pass forward through the neural network and weights are updated backwards once.

Below we investigate the appropriate epochs between 2000, 1000, 125 (compared to the default 500 used in the previous sections) to use in the iterative gradient descent approach, that will not underfit the data. In the next section we will find the best approach to divide the data into batches.

In [40]:
```
accuracy_score_2000_epochs = build_fit_evaluate_nn(build_model_overfit,
                                                   x_train_subset_scaled, y_train_subs
et_scaled,
                                                   x_test_subset_scaled, y_test_subset
_scaled,
                                                   optimizer='adam',
                                                   epochs=2000
                                                   );
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 58us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 94.12%
Accuracy Score - Test: 94.85%
```



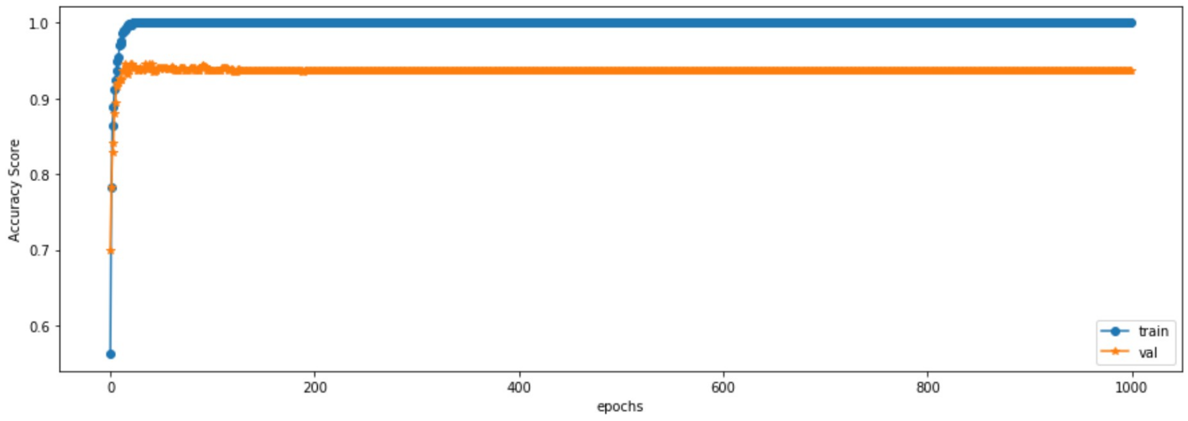
In [41]:
```
df_scores = report_scores(df_scores, '2000 epochs', accuracy_score_2000_epochs)
df_scores
```

Out[41]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%

In [42]:
```
accuracy_score_1000_epochs = build_fit_evaluate_nn(build_model_overfit,
                                                    x_train_subset_scaled, y_train_subs
et_scaled,
                                                    x_test_subset_scaled, y_test_subset
_scaled,
                                                    optimizer='adam',
                                                    epochs=1000
                                                    );
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 66us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 93.62%
Accuracy Score - Test: 94.88%
```



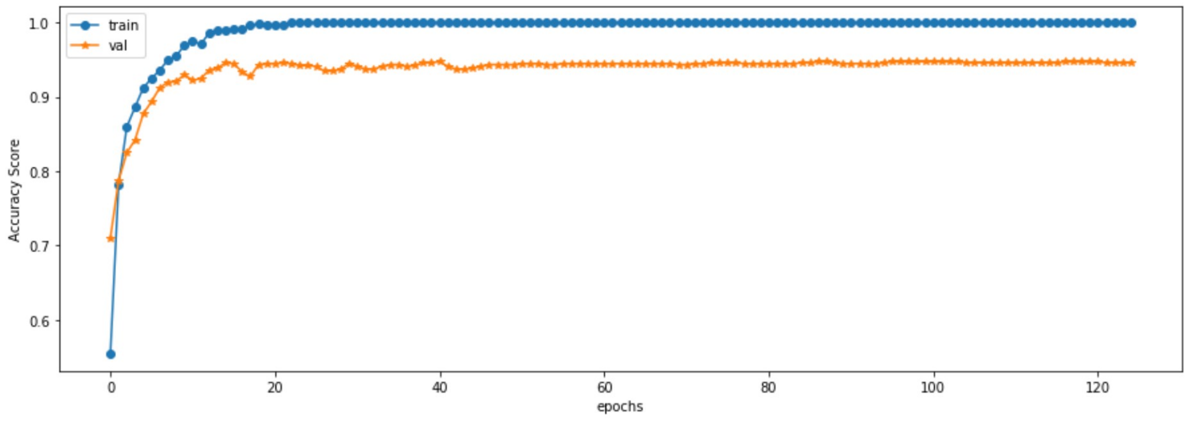
In [43]:
```
df_scores = report_scores(df_scores, '1000 epochs', accuracy_score_1000_epochs)
df_scores
```

Out[43]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%

In [44]:
```
accuracy_score_125_epochs = build_fit_evaluate_nn(build_model_overfit,
                                                  x_train_subset_scaled, y_train_subs
et_scaled,
                                                  x_test_subset_scaled, y_test_subset
_scaled,
                                                  optimizer='adam',
                                                  epochs=125
                                                  );
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 60us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 94.62%
Accuracy Score - Test: 94.73%
```



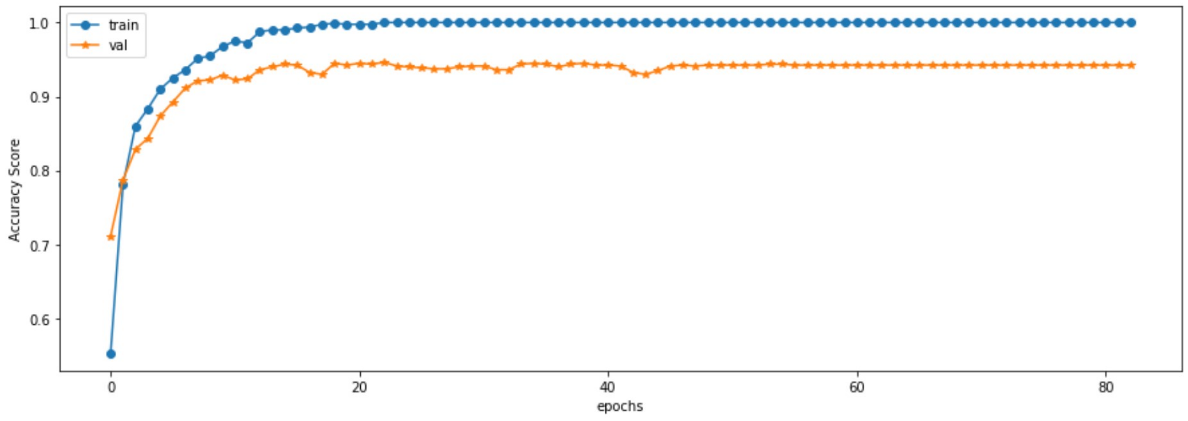
In [45]:
```
df_scores = report_scores(df_scores, '125 epochs', accuracy_score_125_epochs)
df_scores
```

Out[45]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%

In [46]:
```
accuracy_score_83_epochs = build_fit_evaluate_nn(build_model_overfit,
                                                 x_train_subset_scaled, y_train_subs
et_scaled,
                                                 x_test_subset_scaled, y_test_subset
_scaled,
                                                 optimizer='adam',
                                                 epochs=83
                                                 );
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 59us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 94.25%
Accuracy Score - Test: 94.58%
```



In [47]:
```
df_scores = report_scores(df_scores, '83 epochs', accuracy_score_83_epochs)
df_scores
```

Out[47]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%
0	83 epochs	1.0000	0.94250	0.945756	2.2%

Increasing the number of epochs from 500 to 1000 or 2000 improves the test accuracy just a little bit.

Reducing training to 125 epochs, and further to 83 epochs seems to achive less overfit.

We will continue with 83 epochs and thus gain more rooms for data augmentation. In fact, we are limited to use 200000 training samples. By choosing 83 epochs we can augment our training/validation dataset to 4800 samples (2400 for train, 2400 for validation).

Batch Size

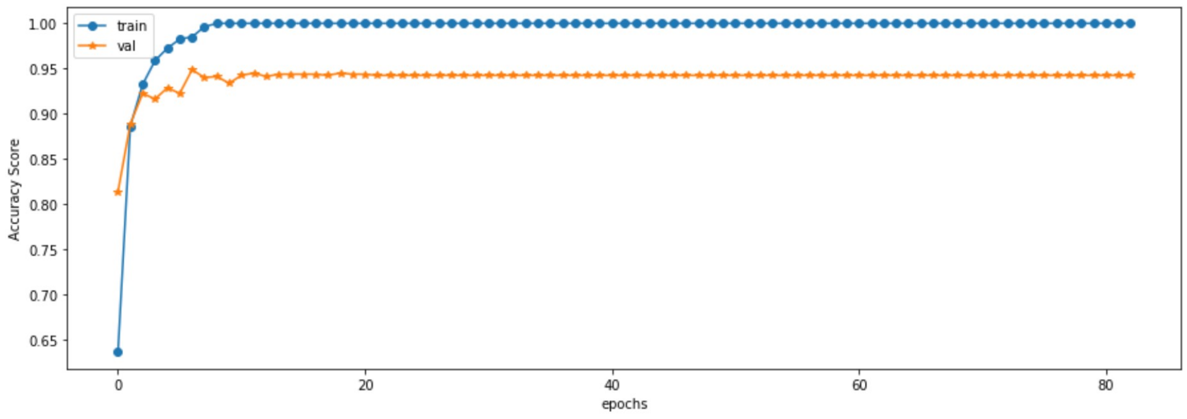
The batch size is the number of samples shown to the network before the weights are updated.

It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize. Reference: https://arxiv.org/abs/1609.04836 (https://arxiv.org/abs/1609.04836)

We tend to use a smaller batch size also because of lower computational power. We will try batch size of 32 and 64, compared to 128 which we used so far.

In [48]:
```
accuracy_score_batchsize_32 = build_fit_evaluate_nn(build_model_overfit,
                                                    x_train_subset_scaled, y_train_subs
et_scaled,
                                                    x_test_subset_scaled, y_test_subset
_scaled,
                                                    optimizer='adam',
                                                    epochs = 83,
                                                    batch_size=32
                                                    );
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 60us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 94.25%
Accuracy Score - Test: 94.68%
```



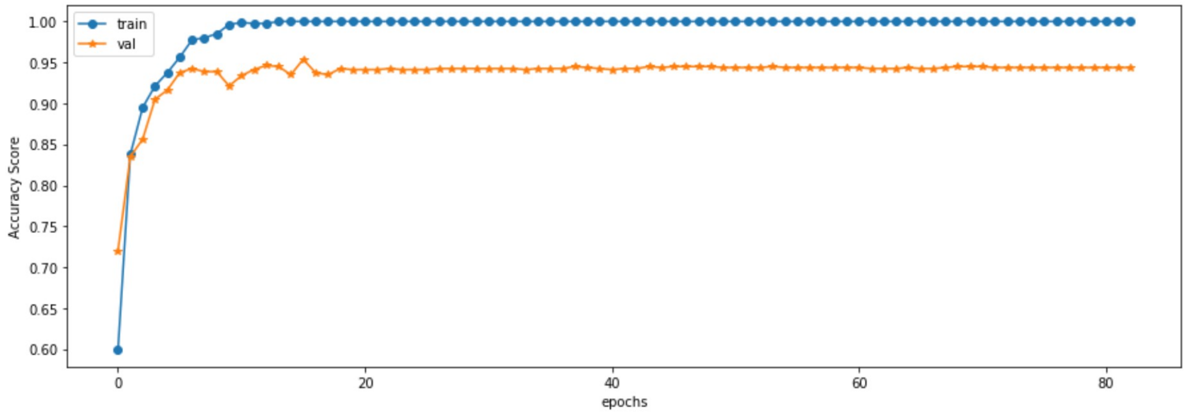
In [49]:
```
df_scores = report_scores(df_scores, 'batch size 32', accuracy_score_batchsize_32)
df_scores
```

Out[49]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%
0	83 epochs	1.0000	0.94250	0.945756	2.2%
0	batch size 32	1.0000	0.94250	0.946760	2.31%

In [50]:
```
accuracy_score_batchsize_64 = build_fit_evaluate_nn(build_model_overfit,
                                                    x_train_subset_scaled, y_train_subs
et_scaled,
                                                    x_test_subset_scaled, y_test_subset
_scaled,
                                                    optimizer='adam',
                                                    epochs = 83,
                                                    batch_size=64
                                                    );
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 63us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 94.38%
Accuracy Score - Test: 94.68%
```



In [51]:
```
df_scores = report_scores(df_scores, 'batch size 64', accuracy_score_batchsize_64)
df_scores
```

Out[51]:

	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%
0	83 epochs	1.0000	0.94250	0.945756	2.2%
0	batch size 32	1.0000	0.94250	0.946760	2.31%
0	batch size 64	1.0000	0.94375	0.946760	2.31%

Lowering the batch size improves test accuracy. It also leads to a smaller number of iterations of a training algorithm.

For example, we divide 400 training samples into batches of 32, which take 12 iterations to complete an epoch.

We will continue with batch size 32.

Data Augmentation

We try a simple way to reduce overfitting by increasing the training set size by rotating, flipping, scaling or shifting the original images.

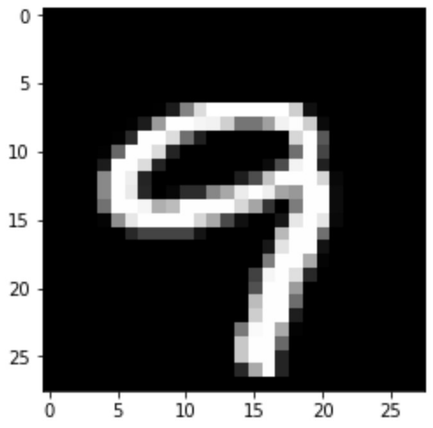
Partly ideas and code adapted from https://machinelearningmastery.com/image-augmentation-deep-learning-keras/ (https://machinelearningmastery.com/image-augmentation-deep-learning-keras/)

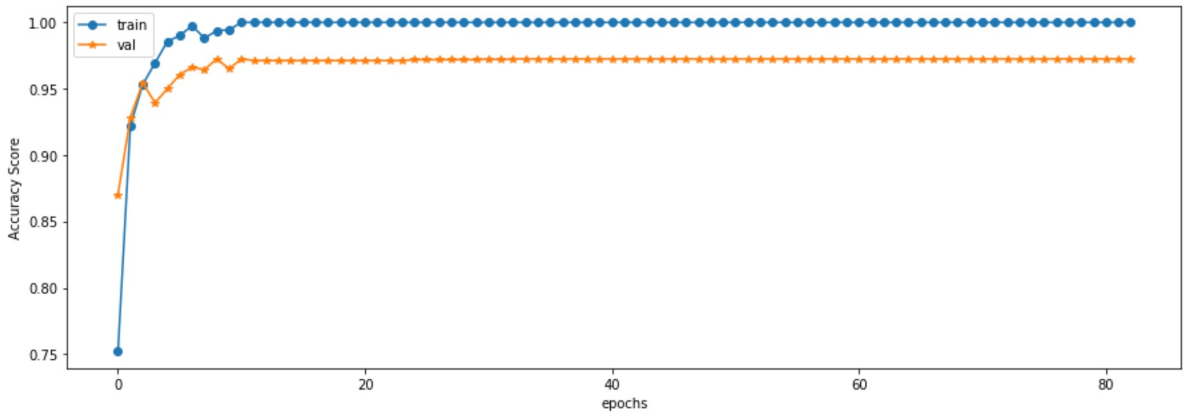
In [59]:
```
def augment_data(Xtrain, ytrain, nb_new_samples, datagen, append=True, display=Tru
e):
    # reshape to be (samples, pixels, width, height
    Xtrain_4D = Xtrain.reshape(Xtrain.shape[0], 28, 28, 1)
    # convert from int to float
    Xtrain_4D = Xtrain_4D.astype('float32')
    # fit parameters from data
    datagen.fit(Xtrain_4D)
    # retrieve one batch of images
    (Xbatch, ybatch) = datagen.flow(Xtrain_4D, ytrain, batch_size=nb_new_samples,
shuffle=False).next()
    Xbatch = Xbatch.reshape(Xbatch.shape[0], 28, 28)
    if display:
        #display an example
        plt.imshow(Xbatch[0].reshape(28, 28), cmap=plt.get_cmap('gray'))
    if append:
        # append
        Xtrain_augmented = np.append(Xtrain, Xbatch, axis=0)
        ytrain_augmented = np.append(ytrain, ybatch, axis=0)
    else:
        Xtrain_augmented = Xbatch
        ytrain_augmented = ybatch
    #shuffle the data
    Xtrain_augmented, ytrain_augmented = shuffle(Xtrain_augmented, ytrain_augmente
d, random_state=1)
    # scale the data
    Xtrain_augmented_scaled,  ytrain_augmented_scaled = get_scaled_data(Xtrain_aug
mented, ytrain_augmented)
    return Xtrain_augmented_scaled, ytrain_augmented_scaled
```

Darkening, Lightening

In [60]:
```
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalizatio
n=True)
x_train_subset_augmented1, y_train_subset_augmented1 = augment_data(x_train_subset
, y_train_subset,
                                                                              nb_new_sam
ples=800, datagen=datagen)
accuracy_score_augmented1 = build_fit_evaluate_nn(build_model_overfit,
                                               x_train_subset_augmented1, y_train_
subset_augmented1,
                                               x_test_subset_scaled, y_test_subset
_scaled,
                                               optimizer='adam',
                                               epochs = 83,
                                               batch_size=32);
```

```
Accuracy Score: train=1.0, val=1.0
1991/1991 [==============================] - 0s 68us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 97.25%
Accuracy Score - Test: 96.69%
```





In [61]:
```
df_scores = report_scores(df_scores, 'data augmentation - darkening, lightening',
accuracy_score_augmented1)
df_scores
```

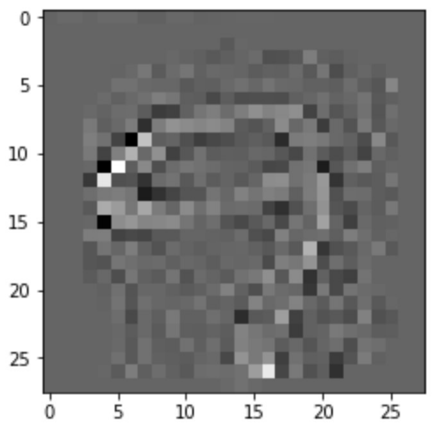
Out[61]:

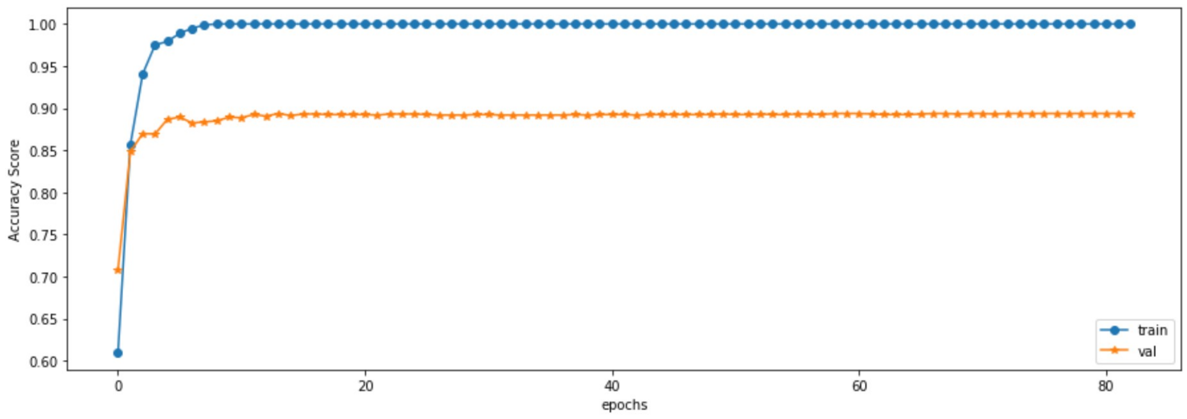
	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%
0	83 epochs	1.0000	0.94250	0.945756	2.2%
0	batch size 32	1.0000	0.94250	0.946760	2.31%
0	batch size 64	1.0000	0.94375	0.946760	2.31%
0	data augmentation - ZCA whitening	1.0000	0.75000	0.713963	-22.85%
0	data augmentation - darkening, lightening	1.0000	0.97250	0.966851	4.48%

ZCA Whitening

In [62]:
```
datagen = ImageDataGenerator(zca_whitening=True)
x_train_subset_augmented2, y_train_subset_augmented2 = augment_data(x_train_subset
, y_train_subset,
                                                                        nb_new_samples
=800, datagen=datagen)
accuracy_score_augmented2 = build_fit_evaluate_nn(build_model_overfit,
                                               x_train_subset_augmented2, y_train_
subset_augmented2,
                                               x_test_subset_scaled, y_test_subset
_scaled,
                                               optimizer='adam',
                                               epochs = 83,
                                               batch_size=32
                                                 );
```

```
C:\ProgramData\Anaconda3\lib\site-packages\keras_preprocessing\image.py:836: Use
rWarning: This ImageDataGenerator specifies `zca_whitening`, which overrides set
ting of `featurewise_center`.
  warnings.warn('This ImageDataGenerator specifies '

Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 70us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 89.38%
Accuracy Score - Test: 94.75%
```





In [63]:
```
df_scores = report_scores(df_scores, 'data augmentation - ZCA whitening', accuracy
_score_augmented2)
df_scores
```

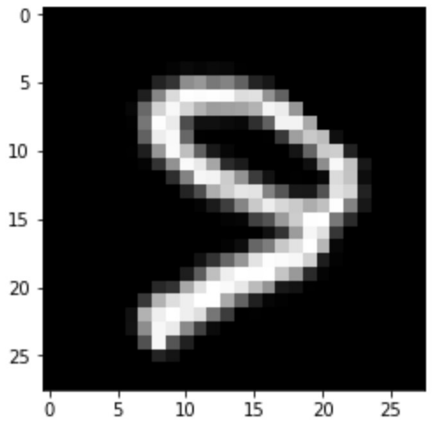
Out[63]:

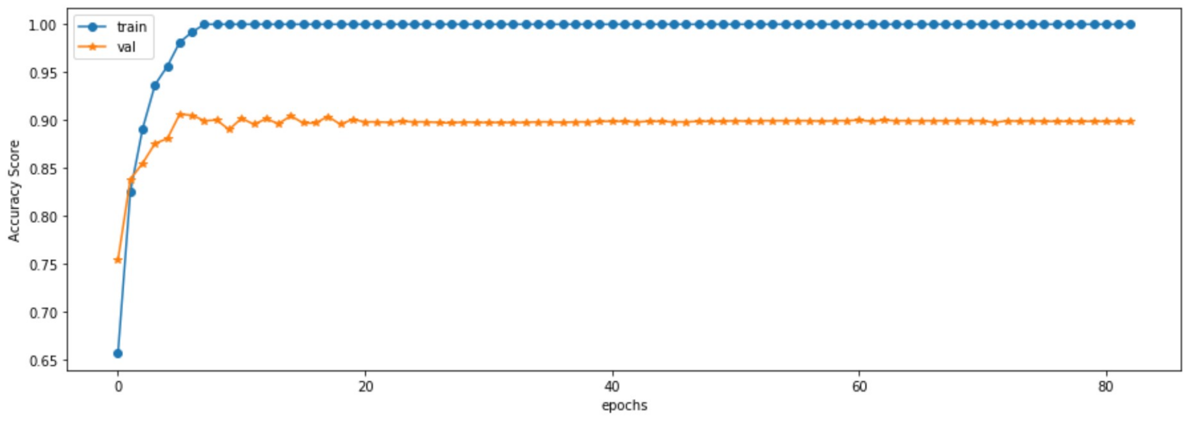
	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%
0	83 epochs	1.0000	0.94250	0.945756	2.2%
0	batch size 32	1.0000	0.94250	0.946760	2.31%
0	batch size 64	1.0000	0.94375	0.946760	2.31%
0	data augmentation - darkening, lightening	1.0000	0.97250	0.966851	4.48%
0	data augmentation - ZCA whitening	1.0000	0.89375	0.947514	2.39%

Random Rotation

In [64]:
```
datagen = ImageDataGenerator(rotation_range=90)
x_train_subset_augmented3, y_train_subset_augmented3 = augment_data(x_train_subset
, y_train_subset,
                                                                      nb_new_samples
=800, datagen=datagen)
accuracy_score_augmented3 = build_fit_evaluate_nn(build_model_overfit,
                                               x_train_subset_augmented3, y_train_
subset_augmented3,
                                               x_test_subset_scaled, y_test_subset
_scaled,
                                               optimizer='adam',
                                               epochs = 83,
                                               batch_size=32);
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 70us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 89.88%
Accuracy Score - Test: 95.48%
```





In [65]:
```
df_scores = report_scores(df_scores, 'data augmentation - random rotation', accura
cy_score_augmented3)
df_scores
```

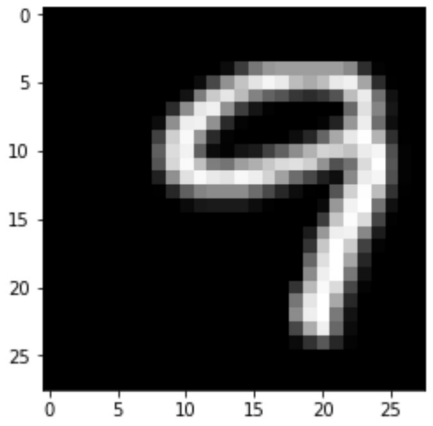
Out[65]:

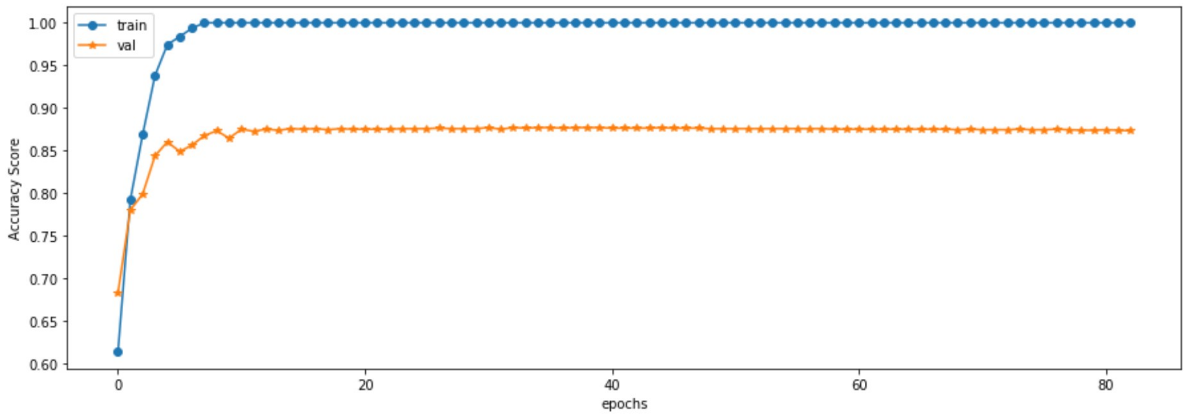
	regularization	train	val	test	improvment
0	none	0.9925	0.93125	0.925414	NaN
0	optimizer Adam	1.0000	0.93875	0.947765	2.42%
0	2000 epochs	1.0000	0.94125	0.948518	2.5%
0	1000 epochs	1.0000	0.93625	0.948769	2.52%
0	125 epochs	1.0000	0.94625	0.947263	2.36%
0	83 epochs	1.0000	0.94250	0.945756	2.2%
0	batch size 32	1.0000	0.94250	0.946760	2.31%
0	batch size 64	1.0000	0.94375	0.946760	2.31%
0	data augmentation - darkening, lightening	1.0000	0.97250	0.966851	4.48%
0	data augmentation - ZCA whitening	1.0000	0.89375	0.947514	2.39%
0	data augmentation - random rotation	1.0000	0.89875	0.954797	3.18%

Random Shifts

```
In [66]: datagen = ImageDataGenerator(width_shift_range=0.2, height_shift_range=0.2)
         x_train_subset_augmented4, y_train_subset_augmented4 = augment_data(x_train_subset
         , y_train_subset,
                                                                                  nb_new_samples
         =800, datagen=datagen)
         accuracy_score_augmented4 = build_fit_evaluate_nn(build_model_overfit,
                                                        x_train_subset_augmented4, y_train_
         subset_augmented4,
                                                        x_test_subset_scaled, y_test_subset
         _scaled,
                                                        optimizer='adam',
                                                        epochs = 83,
                                                        batch_size=32);
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 73us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 87.31%
Accuracy Score - Test: 95.48%
```





```
In [67]: df_scores = report_scores(df_scores, 'data augmentation - random shift', accuracy_
         score_augmented4)
         df_scores
```

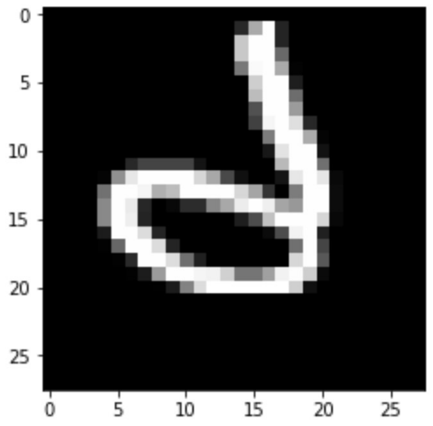
Out[67]:

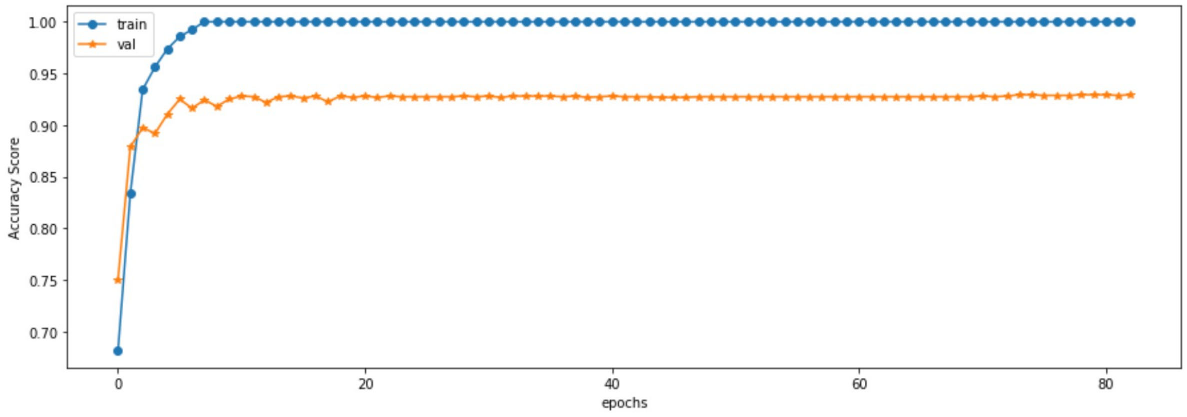
	regularization	train	val	test	improvment
0	none	0.9925	0.931250	0.925414	NaN
0	optimizer Adam	1.0000	0.938750	0.947765	2.42%
0	2000 epochs	1.0000	0.941250	0.948518	2.5%
0	1000 epochs	1.0000	0.936250	0.948769	2.52%
0	125 epochs	1.0000	0.946250	0.947263	2.36%
0	83 epochs	1.0000	0.942500	0.945756	2.2%
0	batch size 32	1.0000	0.942500	0.946760	2.31%
0	batch size 64	1.0000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.0000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.0000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.0000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.0000	0.873125	0.954797	3.18%

Random Flips

In [68]:
```
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
x_train_subset_augmented5, y_train_subset_augmented5 = augment_data(x_train_subset
, y_train_subset,
                                                                     nb_new_samples
=800, datagen=datagen)
accuracy_score_augmented5 = build_fit_evaluate_nn(build_model_overfit,
                                               x_train_subset_augmented5, y_train_
subset_augmented5,
                                               x_test_subset_scaled, y_test_subset
_scaled,
                                               optimizer='adam',
                                               epochs = 83,
                                               batch_size=32);
```

```
Accuracy Score: train=1.0, val=0.9
1991/1991 [==============================] - 0s 76us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 92.94%
Accuracy Score - Test: 94.20%
```





```
In [69]: df_scores = report_scores(df_scores, 'data augmentation - random flips', accuracy_
         score_augmented5)
         df_scores
```

Out[69]:

	regularization	train	val	test	improvment
0	none	0.9925	0.931250	0.925414	NaN
0	optimizer Adam	1.0000	0.938750	0.947765	2.42%
0	2000 epochs	1.0000	0.941250	0.948518	2.5%
0	1000 epochs	1.0000	0.936250	0.948769	2.52%
0	125 epochs	1.0000	0.946250	0.947263	2.36%
0	83 epochs	1.0000	0.942500	0.945756	2.2%
0	batch size 32	1.0000	0.942500	0.946760	2.31%
0	batch size 64	1.0000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.0000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.0000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.0000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.0000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.0000	0.929375	0.941989	1.79%

Multiple Augmentations

Here we use all 5 image transformations above in order to build a single augmented dataset.

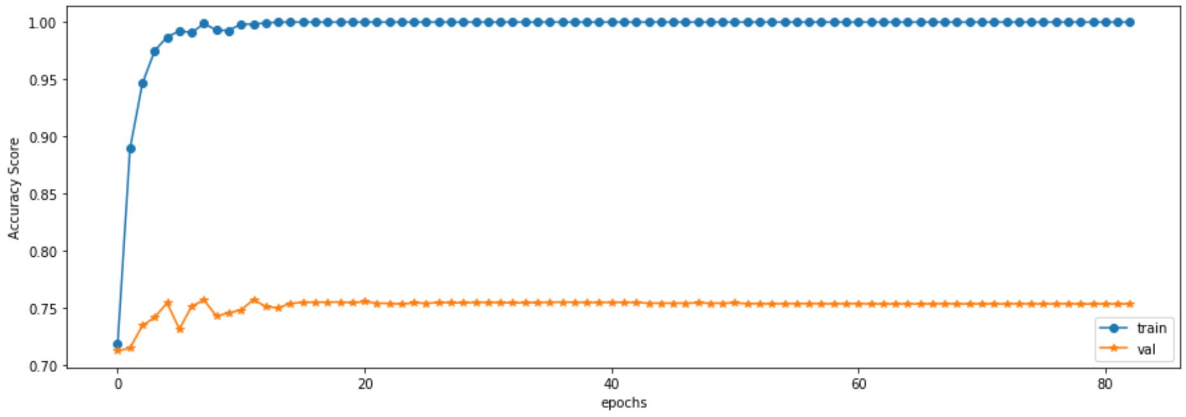
In [71]:
```
# function that performs 5 types of image transformation on a 800x28x28 set
# it adds 480*5 new samples, scales and returns a 1D input vector
def augment_data_full(x_train_subset, y_train_subset, nb_samples_to_augment=480):
    datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normaliz
ation=True)
    x_train_subset_augmented1, y_train_subset_augmented1 = augment_data(x_train_su
bset, y_train_subset,
                                                                      nb_new_sam
ples=nb_samples_to_augment,
                                                                      datagen=da
tagen, append=False, display=False)
    datagen = ImageDataGenerator(zca_whitening=True)
    x_train_subset_augmented2, y_train_subset_augmented2 = augment_data(x_train_su
bset, y_train_subset,
                                                                      nb_new_sam
ples=nb_samples_to_augment,
                                                                      datagen=da
tagen, append=False, display=False)
    datagen = ImageDataGenerator(rotation_range=90)
    x_train_subset_augmented3, y_train_subset_augmented3 = augment_data(x_train_su
bset, y_train_subset,
                                                                      nb_new_sam
ples=nb_samples_to_augment,
                                                                      datagen=da
tagen, append=False, display=False)
    datagen = ImageDataGenerator(width_shift_range=0.2, height_shift_range=0.2)
    x_train_subset_augmented4, y_train_subset_augmented4 = augment_data(x_train_su
bset, y_train_subset,
                                                                      nb_new_sam
ples=nb_samples_to_augment,
                                                                      datagen=da
tagen, append=False, display=False)
    datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
    x_train_subset_augmented5, y_train_subset_augmented5 = augment_data(x_train_su
bset, y_train_subset,
                                                                      nb_new_sam
ples=nb_samples_to_augment,
                                                                      datagen=da
tagen, append=False, display=False)
    x_train_subset_scaled,  y_train_subset_scaled = get_scaled_data(x_train_subset
, y_train_subset)
    x_test_subset_scaled,  y_test_subset_scaled = get_scaled_data(x_test_subset, y
_test_subset)
    # append all augmented datasets
    x_train_subset_augmented = np.append(np.append(np.append(np.append(np.append(
                               x_train_subset_augmented1,
                               x_train_subset_augmented2, axis=0),
                               x_train_subset_augmented3, axis=0),
                               x_train_subset_augmented4, axis=0),
                               x_train_subset_augmented5, axis=0),
                               x_train_subset_scaled, axis=0)
    y_train_subset_augmented = np.append(np.append(np.append(np.append(np.append(
                               y_train_subset_augmented1,
                               y_train_subset_augmented2, axis=0),
                               y_train_subset_augmented3, axis=0),
                               y_train_subset_augmented4, axis=0),
                               y_train_subset_augmented5, axis=0),
                               y_train_subset_scaled, axis=0)
    return x_train_subset_augmented, y_train_subset_augmented
```

In [72]:
```
x_train_subset_augmented, y_train_subset_augmented = augment_data_full(x_train_sub
set, y_train_subset, nb_samples_to_augment=800)
print("Fully augmented training set features shape", x_train_subset_augmented.shap
e)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\keras_preprocessing\image.py:836: Use
rWarning: This ImageDataGenerator specifies `zca_whitening`, which overrides set
ting of `featurewise_center`.
  warnings.warn('This ImageDataGenerator specifies '

Fully augmented training set features shape (4800, 784)
```

In [73]:
```
accuracy_score_augmented = build_fit_evaluate_nn(build_model_overfit,
                                               x_train_subset_augmented, y_train_s
ubset_augmented,
                                               x_test_subset_scaled, y_test_subset
_scaled,
                                               optimizer='adam',
                                               epochs = 83,
                                               batch_size=32);
```

```
Accuracy Score: train=1.0, val=0.8
1991/1991 [==============================] - 0s 86us/step
Accuracy Score - Train: 100.00%
Accuracy Score - Val: 75.33%
Accuracy Score - Test: 95.50%
```



In [74]:
```
df_scores = report_scores(df_scores, 'data augmentation - multiple', accuracy_scor
e_augmented)
df_scores
```

Out[74]:

	regularization	train	val	test	improvment
0	none	0.9925	0.931250	0.925414	NaN
0	optimizer Adam	1.0000	0.938750	0.947765	2.42%
0	2000 epochs	1.0000	0.941250	0.948518	2.5%
0	1000 epochs	1.0000	0.936250	0.948769	2.52%
0	125 epochs	1.0000	0.946250	0.947263	2.36%
0	83 epochs	1.0000	0.942500	0.945756	2.2%
0	batch size 32	1.0000	0.942500	0.946760	2.31%
0	batch size 64	1.0000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.0000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.0000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.0000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.0000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.0000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.0000	0.753333	0.955048	3.2%

When we extend the training/validation set to 4800 samples using a transformation of the images, we can observe a good improve on test accuracy.

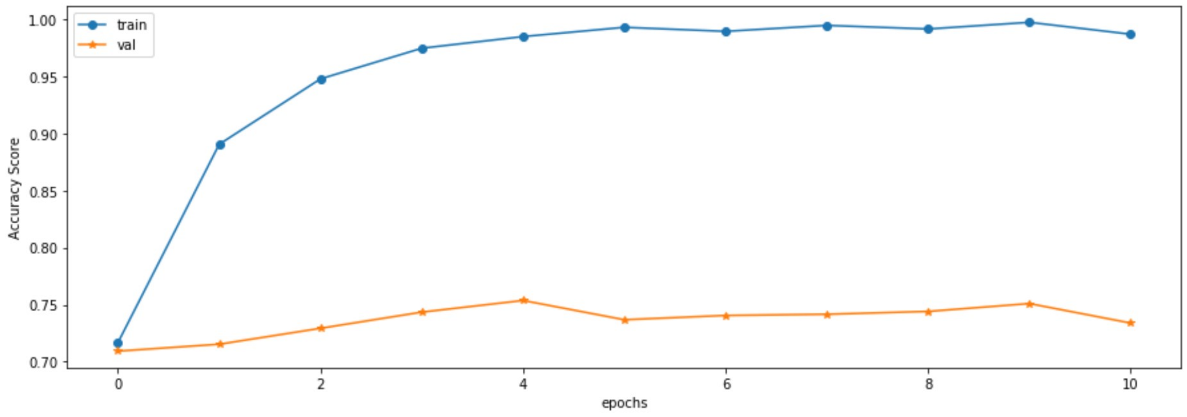
We will investigate further regularization using this augmented training set.

Early Stopping

With early stopping we abort training shortly after the validation accuracy was not improving.

In [75]:
```
callbacks = [EarlyStopping(monitor='val_acc', patience=6)]
accuracy_score_early_stop = build_fit_evaluate_nn(build_model_overfit,
                                                  x_train_subset_augmented, y_tra
in_subset_augmented,
                                                  x_test_subset_scaled, y_test_su
bset_scaled,
                                                  optimizer='adam',
                                                  epochs = 83,
                                                  batch_size=32,
                                               callbacks=callbacks);
```

```
Accuracy Score: train=1.0, val=0.7
1991/1991 [==============================] - 0s 81us/step
Accuracy Score - Train: 98.71%
Accuracy Score - Val: 73.40%
Accuracy Score - Test: 94.42%
```



In [76]:
```
df_scores = report_scores(df_scores, 'early stopping', accuracy_score_early_stop)
df_scores
```

Out[76]:

	regularization	train	val	test	improvment
0	none	0.992500	0.931250	0.925414	NaN
0	optimizer Adam	1.000000	0.938750	0.947765	2.42%
0	2000 epochs	1.000000	0.941250	0.948518	2.5%
0	1000 epochs	1.000000	0.936250	0.948769	2.52%
0	125 epochs	1.000000	0.946250	0.947263	2.36%
0	83 epochs	1.000000	0.942500	0.945756	2.2%
0	batch size 32	1.000000	0.942500	0.946760	2.31%
0	batch size 64	1.000000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.000000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.000000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.000000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.000000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.000000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.000000	0.753333	0.955048	3.2%
0	early stopping	0.987083	0.733958	0.944249	2.04%

Early stopping after a low number of epochs with patience 6 achieves good accuracy on test, with less overfitting to the training set.

Dropout

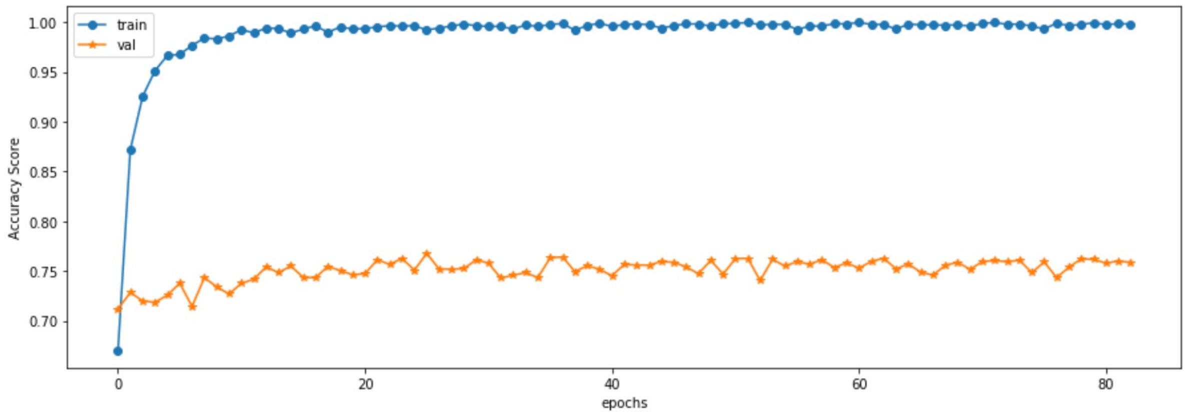
We randomly drop some weights during training in order to learn independent internal representations. We add a new 'Dropout' layer and impose a constraint on the weights on input and hidden layers. We further constrain the size of network weights.

The following reference was used: https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/ (https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/)

In [77]:
```
def build_model_dropout(dropout_rate=0.17, optimizer='adam'):
        model = Sequential([
            Dense(100, input_shape=(784,), activation='relu', kernel_initializer='
normal', kernel_constraint=maxnorm(3)),
            Dropout(dropout_rate),
            Dense(100, activation='relu', kernel_initializer='normal', kernel_cons
traint=maxnorm(3)),
            Dropout(dropout_rate),
            Dense(100, activation='relu', kernel_initializer='normal', kernel_cons
traint=maxnorm(3)),
            Dropout(dropout_rate),
            Dense(2, activation='sigmoid', kernel_initializer='normal')
        ])
        model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['a
ccuracy'])
        return model
```

In [79]:
```
#callbacks = [EarlyStopping(monitor='val_acc', patience=1)]
accuracy_score_dropout = build_fit_evaluate_nn(build_model_dropout,
                                                   x_train_subset_augmented, y_tra
in_subset_augmented,
                                                   x_test_subset_scaled, y_test_su
bset_scaled,
                                                   optimizer='adam',
                                                   epochs = 83,
                                                   batch_size=32
                                                #,callbacks=callbacks
                                              );
```

```
Accuracy Score: train=1.0, val=0.8
1991/1991 [==============================] - 0s 88us/step
Accuracy Score - Train: 99.83%
Accuracy Score - Val: 75.88%
Accuracy Score - Test: 96.03%
```



In [80]:
```
df_scores = report_scores(df_scores, 'dropout', accuracy_score_dropout)
df_scores
```

Out[80]:

	regularization	train	val	test	improvment
0	none	0.992500	0.931250	0.925414	NaN
0	optimizer Adam	1.000000	0.938750	0.947765	2.42%
0	2000 epochs	1.000000	0.941250	0.948518	2.5%
0	1000 epochs	1.000000	0.936250	0.948769	2.52%
0	125 epochs	1.000000	0.946250	0.947263	2.36%
0	83 epochs	1.000000	0.942500	0.945756	2.2%
0	batch size 32	1.000000	0.942500	0.946760	2.31%
0	batch size 64	1.000000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.000000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.000000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.000000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.000000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.000000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.000000	0.753333	0.955048	3.2%
0	early stopping	0.987083	0.733958	0.944249	2.04%
0	dropout	0.998333	0.758750	0.960321	3.77%

Fitting the model on augmented dataset with dropout rate 0.17 improves the the baseline overfitted model by over 4%.

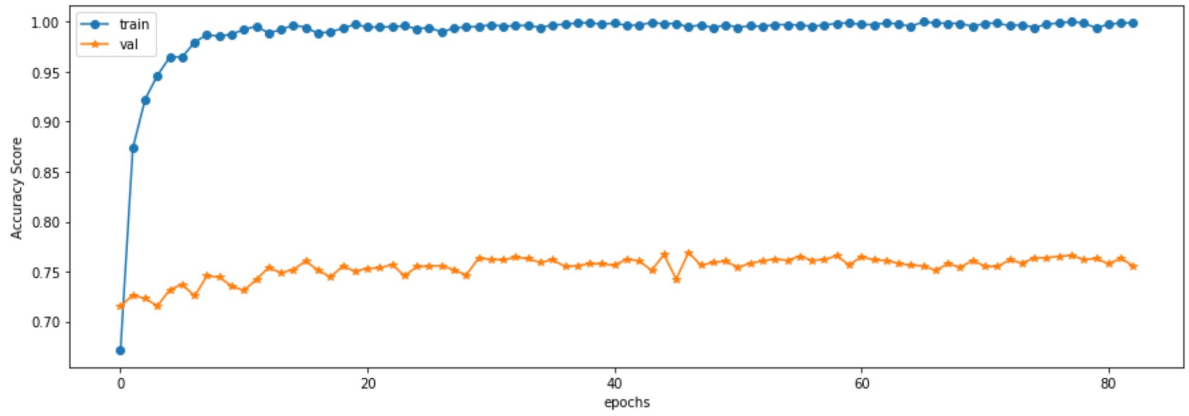
L2 Regularization

We use L2 to force the weights to decay towards zero.

In [81]:
```
def build_model_l2(dropout_rate=0.17, optimizer='adam', regularization=0.0001):
        model = Sequential([
            Dense(100, input_shape=(784,), activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l2(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l2(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l2(regularization)),
            Dropout(dropout_rate),
            Dense(2, activation='sigmoid', kernel_initializer='normal')
        ])
        model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['a
ccuracy'])
        return model
```

In [82]:
```
accuracy_score_l2 = build_fit_evaluate_nn(build_model_l2,
                                                    x_train_subset_augmented, y_tra
in_subset_augmented,
                                                    x_test_subset_scaled, y_test_su
bset_scaled,
                                                    optimizer='adam',
                                                    epochs = 83,
                                                    batch_size=32
                                          );
```

```
Accuracy Score: train=1.0, val=0.8
1991/1991 [==============================] - 0s 95us/step
Accuracy Score - Train: 99.88%
Accuracy Score - Val: 75.58%
Accuracy Score - Test: 96.28%
```



In [83]:
```
df_scores = report_scores(df_scores, 'L2 penalty', accuracy_score_l2)
df_scores
```

Out[83]:

	regularization	train	val	test	improvment
0	none	0.992500	0.931250	0.925414	NaN
0	optimizer Adam	1.000000	0.938750	0.947765	2.42%
0	2000 epochs	1.000000	0.941250	0.948518	2.5%
0	1000 epochs	1.000000	0.936250	0.948769	2.52%
0	125 epochs	1.000000	0.946250	0.947263	2.36%
0	83 epochs	1.000000	0.942500	0.945756	2.2%
0	batch size 32	1.000000	0.942500	0.946760	2.31%
0	batch size 64	1.000000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.000000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.000000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.000000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.000000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.000000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.000000	0.753333	0.955048	3.2%
0	early stopping	0.987083	0.733958	0.944249	2.04%
0	dropout	0.998333	0.758750	0.960321	3.77%
0	L2 penalty	0.998750	0.755833	0.962833	4.04%

Fitting the model with augmented training data together with L2 regularization 0.001 with dropout rate 0.17 improved the test accuracy by over 4%.

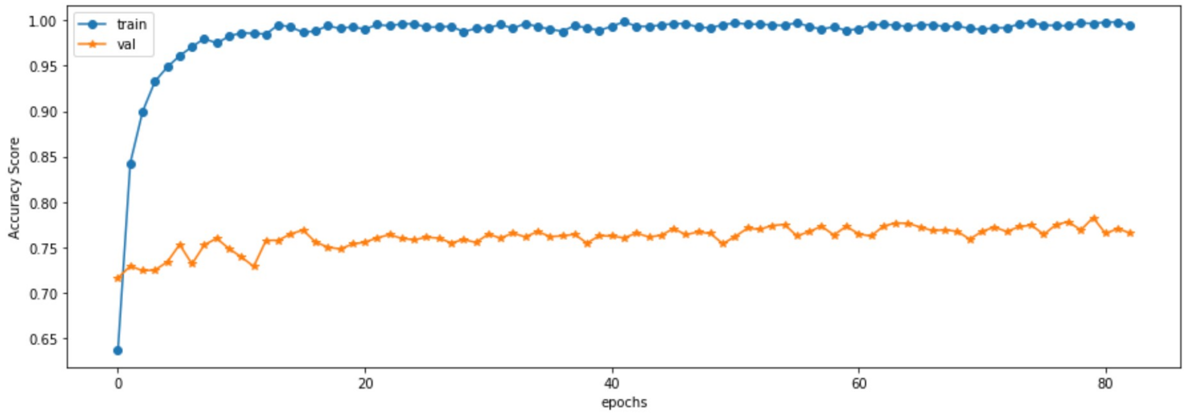
L1 Regularization

L1 regularization forces some weights to zero.

In [84]:
```
def build_model_l1(dropout_rate=0.17, optimizer='adam', regularization=0.0001):
        model = Sequential([
            Dense(100, input_shape=(784,), activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(2, activation='sigmoid', kernel_initializer='normal')
        ])
        model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['a
ccuracy'])
        return model
```

In [85]:
```
accuracy_score_l1 = build_fit_evaluate_nn(build_model_l1,
                                                   x_train_subset_augmented, y_tra
in_subset_augmented,
                                                   x_test_subset_scaled, y_test_su
bset_scaled,
                                                   optimizer='adam',
                                                   epochs = 83,
                                                   batch_size=32
                                         );
```

```
Accuracy Score: train=1.0, val=0.8
1991/1991 [==============================] - 0s 106us/step
Accuracy Score - Train: 99.42%
Accuracy Score - Val: 76.58%
Accuracy Score - Test: 96.33%
```



In [86]:
```
df_scores = report_scores(df_scores, 'L1 penalty', accuracy_score_l1)
df_scores
```

Out[86]:

	regularization	train	val	test	improvment
0	none	0.992500	0.931250	0.925414	NaN
0	optimizer Adam	1.000000	0.938750	0.947765	2.42%
0	2000 epochs	1.000000	0.941250	0.948518	2.5%
0	1000 epochs	1.000000	0.936250	0.948769	2.52%
0	125 epochs	1.000000	0.946250	0.947263	2.36%
0	83 epochs	1.000000	0.942500	0.945756	2.2%
0	batch size 32	1.000000	0.942500	0.946760	2.31%
0	batch size 64	1.000000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.000000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.000000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.000000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.000000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.000000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.000000	0.753333	0.955048	3.2%
0	early stopping	0.987083	0.733958	0.944249	2.04%
0	dropout	0.998333	0.758750	0.960321	3.77%
0	L2 penalty	0.998750	0.755833	0.962833	4.04%
0	L1 penalty	0.994167	0.765833	0.963335	4.1%

L1 regularization 0.0001 with dropout rate 0.17 also achieves one of the best test accuracy, similarly to L2.

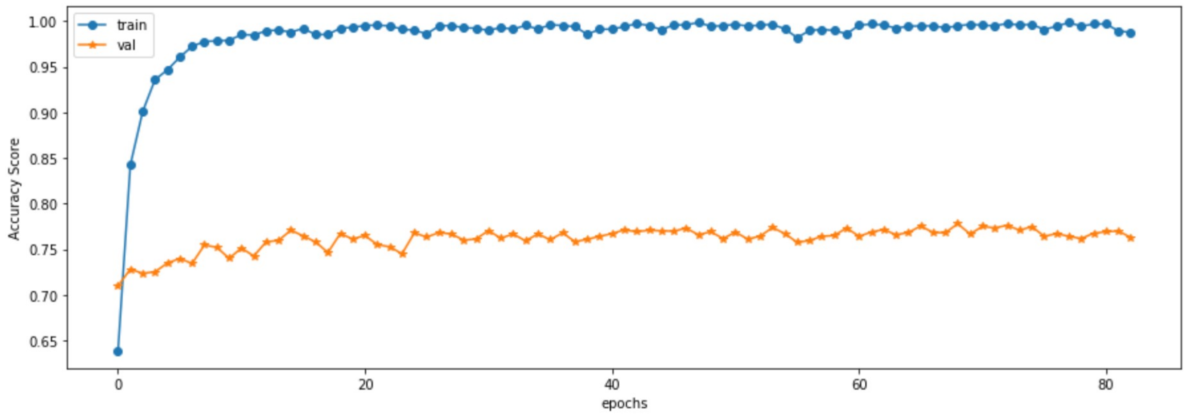
Learning Rate

Here we are interested in finding how much to update the weight at the end of each batch (learning rate), and how much to let the previous update influence the current weight update (momentum)

In [87]:
```
def build_model_lr(dropout_rate=0.17, optimizer='adam', regularization=0.0001, lea
rn_rate=0.001):
        model = Sequential([
            Dense(100, input_shape=(784,), activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(2, activation='sigmoid', kernel_initializer='normal')
        ])
        model.compile(loss='binary_crossentropy', optimizer=keras.optimizers.Adam(
lr=learn_rate), metrics=['accuracy'])
        return model
```

In [88]:
```
#callbacks = [EarlyStopping(monitor='val_acc', patience=7)]
accuracy_score_lr = build_fit_evaluate_nn(build_model_lr,
                                                    x_train_subset_augmented, y_tra
in_subset_augmented,
                                                    x_test_subset_scaled, y_test_su
bset_scaled,
                                                    optimizer='adam',
                                                    epochs = 83,
                                                    batch_size=32
                                                    #,callbacks=callbacks
                                          );
```

```
Accuracy Score: train=1.0, val=0.8
1991/1991 [==============================] - 0s 96us/step
Accuracy Score - Train: 98.77%
Accuracy Score - Val: 76.29%
Accuracy Score - Test: 95.66%
```



In [89]:
```
df_scores = report_scores(df_scores, 'learning rate', accuracy_score_lr)
df_scores
```

Out[89]:

	regularization	train	val	test	improvment
0	none	0.992500	0.931250	0.925414	NaN
0	optimizer Adam	1.000000	0.938750	0.947765	2.42%
0	2000 epochs	1.000000	0.941250	0.948518	2.5%
0	1000 epochs	1.000000	0.936250	0.948769	2.52%
0	125 epochs	1.000000	0.946250	0.947263	2.36%
0	83 epochs	1.000000	0.942500	0.945756	2.2%
0	batch size 32	1.000000	0.942500	0.946760	2.31%
0	batch size 64	1.000000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.000000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.000000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.000000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.000000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.000000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.000000	0.753333	0.955048	3.2%
0	early stopping	0.987083	0.733958	0.944249	2.04%
0	dropout	0.998333	0.758750	0.960321	3.77%
0	L2 penalty	0.998750	0.755833	0.962833	4.04%
0	L1 penalty	0.994167	0.765833	0.963335	4.1%
0	learning rate	0.987708	0.762917	0.956554	3.36%

Tuning the learning rate to 0.001 helped us regularizing the model further up to 4.95%.

Final Model

Based on the observations above, we regularize our FCN as follows:

- **adam optimizer**
- **32 batch size**
- **125 epochs**
- **data augmentation to 4800 samples in the training/validation dataset**
- **L1 regularization with penalty 0.0001**
- **dropout with rate 0.17**
- **learning rate 0.001**

1. Augment training/validation dataset

In [90]:
```
x_train_subset_augmented, y_train_subset_augmented = augment_data_full(x_train_sub
set, y_train_subset, nb_samples_to_augment=800)
print("Original training set features shape", x_train_subset.shape)
print("Fully augmented training set features shape", x_train_subset_augmented.shap
e)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\keras_preprocessing\image.py:836: Use
rWarning: This ImageDataGenerator specifies `zca_whitening`, which overrides set
ting of `featurewise_center`.
  warnings.warn('This ImageDataGenerator specifies '
```

```
Original training set features shape (800, 28, 28)
Fully augmented training set features shape (4800, 784)
```

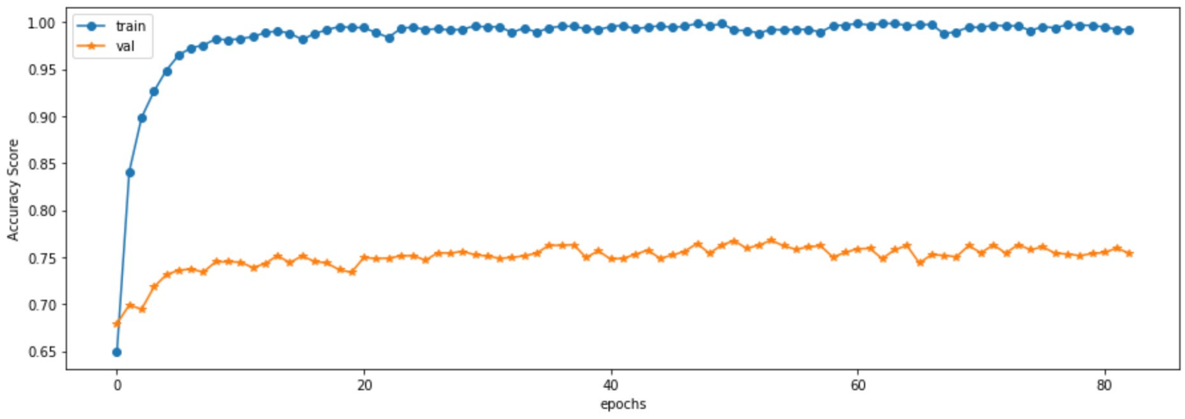
2. Create a FCN with tuned hyper-parameters

In [91]:
```
def build_model_optimal(dropout_rate=0.17, optimizer='adam', regularization=0.0001
, learn_rate=0.001):
        model = Sequential([
            Dense(100, input_shape=(784,), activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(100, activation='relu',
                  kernel_initializer='normal', kernel_constraint=maxnorm(3),
                  kernel_regularizer=regularizers.l1(regularization)),
            Dropout(dropout_rate),
            Dense(2, activation='sigmoid', kernel_initializer='normal')
        ])
        model.compile(loss='binary_crossentropy', optimizer=keras.optimizers.Adam(
lr=learn_rate), metrics=['accuracy'])
        return model
```

3. Fit and Evaluate the FCN with further tuned hyper-parameters

In [92]:
```
accuracy_score_optimal = build_fit_evaluate_nn(build_model_optimal,
                                                x_train_subset_augmented, y_tra
in_subset_augmented,
                                                x_test_subset_scaled, y_test_su
bset_scaled,
                                                optimizer='adam',
                                                epochs = 83,
                                                batch_size=32
                                          );
```

```
Accuracy Score: train=1.0, val=0.8
1991/1991 [==============================] - 0s 236us/step
Accuracy Score - Train: 99.25%
Accuracy Score - Val: 75.42%
Accuracy Score - Test: 95.53%
```



4. Report the accuracy score, compared to other regularization approaches

In [93]:
```
df_scores = report_scores(df_scores, 'optimal', accuracy_score_optimal)
df_scores
```

Out[93]:

	regularization	train	val	test	improvment
0	none	0.992500	0.931250	0.925414	NaN
0	optimizer Adam	1.000000	0.938750	0.947765	2.42%
0	2000 epochs	1.000000	0.941250	0.948518	2.5%
0	1000 epochs	1.000000	0.936250	0.948769	2.52%
0	125 epochs	1.000000	0.946250	0.947263	2.36%
0	83 epochs	1.000000	0.942500	0.945756	2.2%
0	batch size 32	1.000000	0.942500	0.946760	2.31%
0	batch size 64	1.000000	0.943750	0.946760	2.31%
0	data augmentation - darkening, lightening	1.000000	0.972500	0.966851	4.48%
0	data augmentation - ZCA whitening	1.000000	0.893750	0.947514	2.39%
0	data augmentation - random rotation	1.000000	0.898750	0.954797	3.18%
0	data augmentation - random shift	1.000000	0.873125	0.954797	3.18%
0	data augmentation - random flips	1.000000	0.929375	0.941989	1.79%
0	data augmentation - multiple	1.000000	0.753333	0.955048	3.2%
0	early stopping	0.987083	0.733958	0.944249	2.04%
0	dropout	0.998333	0.758750	0.960321	3.77%
0	L2 penalty	0.998750	0.755833	0.962833	4.04%
0	L1 penalty	0.994167	0.765833	0.963335	4.1%
0	learning rate	0.987708	0.762917	0.956554	3.36%
0	optimal	0.992500	0.754167	0.955299	3.23%

5. Save the fitted model

In [94]:
```
model = build_model_optimal()
model_history = model.fit(x_train_subset_augmented, y_train_subset_augmented,
                                    epochs=83, batch_size=32,
                                    validation_split=.5,
                                    verbose=False)
model.save('cs109a_hw9_submit.model')
```

The best accuracy score we have obtained on test set was 4.95%. Each time the network is fitted, the score changes.