Data-X Spring 2018: Homework 03

Regularization and Neural Networks

In this homework, you will get some practice working with regularisation and hyperparameter tuning in prediction models and comparing the construction and training of a simple logistic regression model with a Dense Neural Network model.

Assignment link: https://goo.gl/PW5roz (https://goo.gl/PW5roz)
Course Github: https://github.com/ikhlaqsidhu/data-x (https://github.com/ikhlaqsidhu/data-x)

Part 1: Regularization

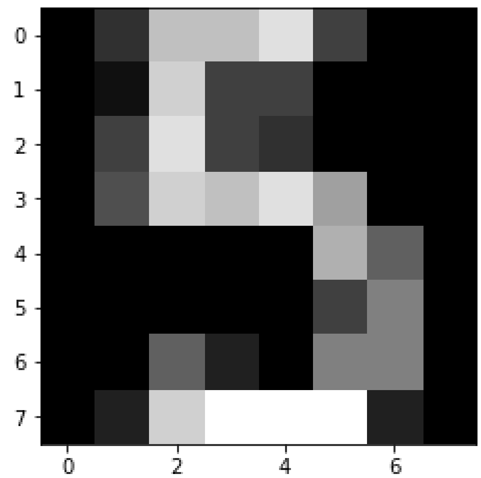
In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, train_test_split, c
ross_val_score
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_digits

# Hide warnings
import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
# data:
digits= load_digits()
x_train, x_test, y_train, y_test = train_test_split(digits.data, digits.
target, test_size=0.20, random_state=0)
```

In [3]:
```
# view the images using this code snippet:

plt.imshow(np.reshape(x_train[1], (8,8)), cmap=plt.cm.gray)
plt.show()
```



Goal: Use this data to train a Logistic Regression classifier that predicts what number the image of the digit is (categories 0-9). You will train two different Regularized models, and tune hyper parameters using K-fold Cross validation (use scikit learn inbuilt method of Stratified K-Fold cross-validation: [http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html)](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html)

Overview

Ridge and Lasso Regression are regularization techniques generally used for creating parsimonious models (Occum's Razor) in presence of a large number of features. The main goal is to combat tendency of models to overfit and balance the statistical-computational tradeoffs that are ubiquitous in the high-dimensional statistics era.

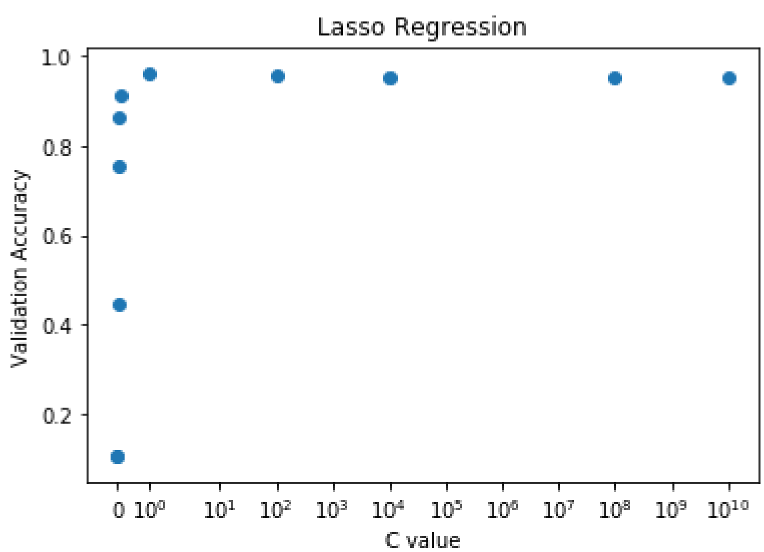
- Ridge Regression
  - L2-regularization
  - Obj = Loss Function + $\alpha||\beta||_2$
- Lasso Regression
  - L1-regularization
  - Obj = Loss Function + $\alpha||\beta||_1$

In [4]:
```
# first we need to load our data and normalize our inputs

digits= load_digits()
x_train, x_test, y_train, y_test = train_test_split \
                           (digits.data, digits.target, test_size=0.20, ran
dom_state=0)

x_train = x_train / 16
x_test = x_test / 16
```

In [5]:
```
# Select a wide range of possible values for our hyperparamter C
# We know C to be equal to 1 / alpha
sample_alpha = np.array([1e-10, 1e-8, 1e-4,1e-2, 1, 10, 20, 50, 100, 1e3
, 1e5])
sample_c = 1/sample_alpha
```

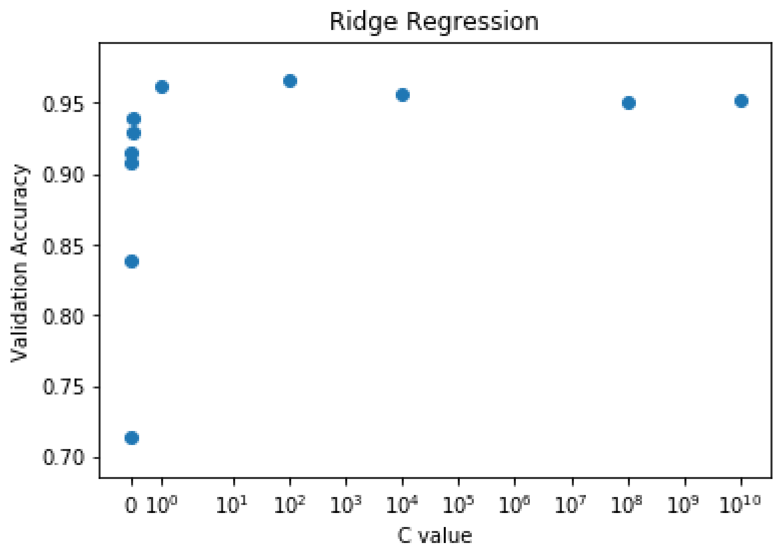
In [6]:
```
def c_optimizer(p, c, title, k=2):
    val_scores = []
    skf = StratifiedKFold(n_splits=7)
    for i in c:
        lgr = LogisticRegression(penalty=p, C=i, random_state=333)
        score = (cross_val_score(lgr, x_train, y_train, cv=skf)).mean()
        val_scores.append(score)

    plt.scatter(c, val_scores)
    plt.ylabel("Validation Accuracy")
    plt.xlabel("C value")
    plt.title(title)
    plt.xscale('symlog')
    plt.show()

    print("Optimal c value: ", c[val_scores.index(max(val_scores))])
    print("Validation Accuracy: ", max(val_scores))
    return max(val_scores)
```

In [7]:
```
# find optimal c for Lasso Regression (L1 Regularization)
lasso_c = c_optimizer('l1', c=sample_c, title="Lasso Regression")
```




```
Optimal c value:  1.0
Validation Accuracy:  0.9637921449761316
```

In [8]:
```
# find optimal c for Ridge Regression (L2 Regularization)
ridge_c = c_optimizer('l2', c=sample_c, title="Ridge Regression")
```




```
Optimal c value:  100.0
Validation Accuracy:  0.9659331917923104
```

In [9]:
```
def fit_model(regularizer, optimal_c):
    model = LogisticRegression(penalty=regularizer, C=optimal_c)
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)
    score = accuracy_score(y_test, y_pred)
    if regularizer == 'l1':
        print ("Optimized Lasso Accuracy: ", score)
    else:
        print ("Optimized Ridge Accuracy: ", score)
```

In [10]:
```
# fit and validate models using optimal c
fit_model('l1', lasso_c)
fit_model('l2', ridge_c)
```

```
Optimized Lasso Accuracy:  0.9527777777777777
Optimized Ridge Accuracy:  0.9611111111111111
```

In [11]:
```
# compared to non-normalized, non-regularized, non-cross validated mode
l:
model2 = LogisticRegression()
model2.fit(x_train*16, y_train)
y_pred = model2.predict(x_test*16)
score = accuracy_score(y_test, y_pred)
print ("Non-optimized LRG Accuracy: ", score)
```

```
Non-optimized LRG Accuracy:  0.95
```

Part 2. Neural Networks

In [13]:
```
import tensorflow as tf
tf.reset_default_graph()
tf.logging.set_verbosity(tf.logging.ERROR)
```

2.1:

Train a multiclass logistic regression model (softmax regression is a good choice) on the DIGITS dataset in tensorflow. You will create input and output placeholders, define the model initializing weight and bias variables, then define a loss function for the model. Once you create a blueprint of the model, compile or train the model to minimize the loss function. Use Batch gradient descent to train the model, and use the hyperparameters: batch_size=100, epochs=200, learning rate=.001. Report accuracy on 20% validation data. Plot the training accuracy vs epoch and plot validation accuracy vs epoc. Show your network graph as seen on tensorboard.

Before we do anything else, we'll first make sure we have Tensorboard properly set up.

In [33]:
```
# TensorBoard Graph visualizer in notebook
import numpy as np
from IPython.display import clear_output, Image, display, HTML

def strip_consts(graph_def, max_const_size=32):
    """Strip large constant values from graph_def."""
    strip_def = tf.GraphDef()
    for n0 in graph_def.node:
        n = strip_def.node.add()
        n.MergeFrom(n0)
        if n.op == 'Const':
            tensor = n.attr['value'].tensor
            size = len(tensor.tensor_content)
            if size > max_const_size:
                tensor.tensor_content = "<stripped %d bytes>"%size
    return strip_def

def show_graph(graph_def, max_const_size=32):
    """Visualize TensorFlow graph."""
    if hasattr(graph_def, 'as_graph_def'):
        graph_def = graph_def.as_graph_def()
    strip_def = strip_consts(graph_def, max_const_size=max_const_size)
    code = """
        <script src="//cdnjs.cloudflare.com/ajax/libs/polymer/0.3.3/plat
form.js"></script>
        <script>
          function load() {{
            document.getElementById("{id}").pbtxt = {data};
          }}
        </script>
        <link rel="import" href="https://tensorboard.appspot.com/tf-grap
h-basic.build.html" onload=load()>
        <div style="height:600px">
          <tf-graph-basic id="{id}"></tf-graph-basic>
        </div>
    """.format(data=repr(str(strip_def)), id='graph'+str(np.random.rand
()))

    iframe = """
        <iframe seamless style="width:1200px;height:620px;border:0" srcd
oc="{}"></iframe>
    """.format(code.replace('"', '&quot;'))
    display(HTML(iframe))
```

Then, we'll do some quick EDA to determine the shape of our TF Placeholders:

In [14]:
```
print("Number of images: ",x_train.shape[0]+x_test.shape[0])
print("Number of features: ", x_train.shape[1])
```

```
Number of images:  1797
Number of features:  64
```

Next, we can create the basic framework for our model, composed of the following pieces:

- x: TensorFlow placeholder, will eventually contain our x_test data
- y: TensorFlow placeholder, will eventually contain our actual y values
- w: weight matrix
- b: bias matrix
- y_pred: predicted y values, calculated using Softmax Regresssion
- cross_entropy: our chosen loss function

In [26]:
```
# convert labels from integers to 1x10 arrays, with a 1 in the row that
 corresponds to the label value
def dense_to_one_hot(labels_dense, num_classes):
    num_labels = labels_dense.shape[0]
    index_offset = np.arange(num_labels) * num_classes
    labels_one_hot = np.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
    return labels_one_hot

labels_train = dense_to_one_hot(y_train, 10)
labels_train = labels_train.astype(np.uint8)
```

In [16]:
```
# create input & output placeholders, defined in a new graph
x = tf.placeholder(tf.float32, [None, 64])
y = tf.placeholder(tf.float32,[None,10])

# set up weight and bias matrices
W = tf.Variable(tf.zeros([64,10]))
b = tf.Variable(tf.zeros([10]))

# define softmax model
y_pred = tf.nn.softmax(tf.matmul(x, W) + b)

# use cross_entropy error as our loss function
#cross_entropy = tf.reduce_mean(-tf.reduce_sum(y * tf.log(y_pred), \
 #                                                 reduction_indices=
[1]))
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y* tf.log(y_pred),axis=1))
```

In [17]:
```
# minimize our loss function using Gradient Descent
train_step = tf.train.GradientDescentOptimizer(0.001)\
                                .minimize(cross_entropy)
```

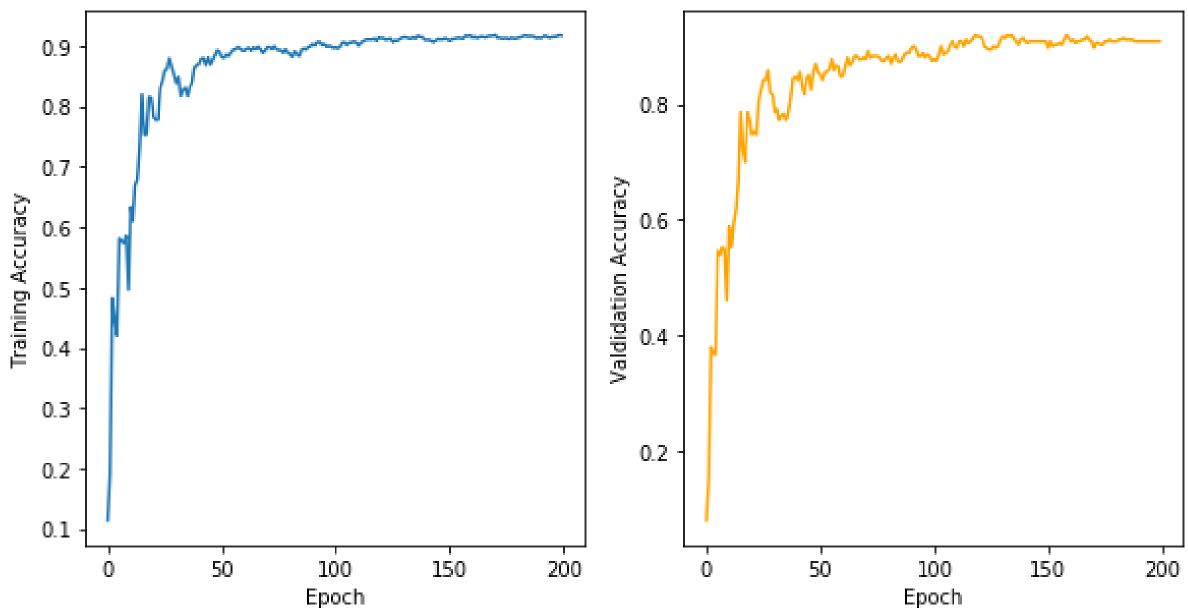
In [18]:
```
# start a session
sess = tf.Session()
tf.global_variables_initializer().run(session=sess)
```

In [19]:
```
def score(x_data, y_data):
    return accuracy_score(y_data, prediction.eval(feed_dict={x: x_data},
 session=sess))
```

In [20]:
```
# 200 epochs
steps = range(200)
batch_size = 100
prediction = tf.argmax(y_pred,1)
train_scores = []
val_scores = []

for step in steps:
    # laels_train[0] - ..
    offset = np.random.randint(0, x_train.shape[0] - batch_size - 1)
    batch_xs = x_train[offset:(offset + batch_size), :]
    batch_ys = labels_train[offset:(offset + batch_size), :]
    for batch_x, batch_y in zip(batch_xs, batch_ys):
        sess.run(train_step, feed_dict={x: batch_xs, y: batch_ys})
    train_scores.append(score(x_train, y_train))
    val_scores.append(score(x_test, y_test))
```

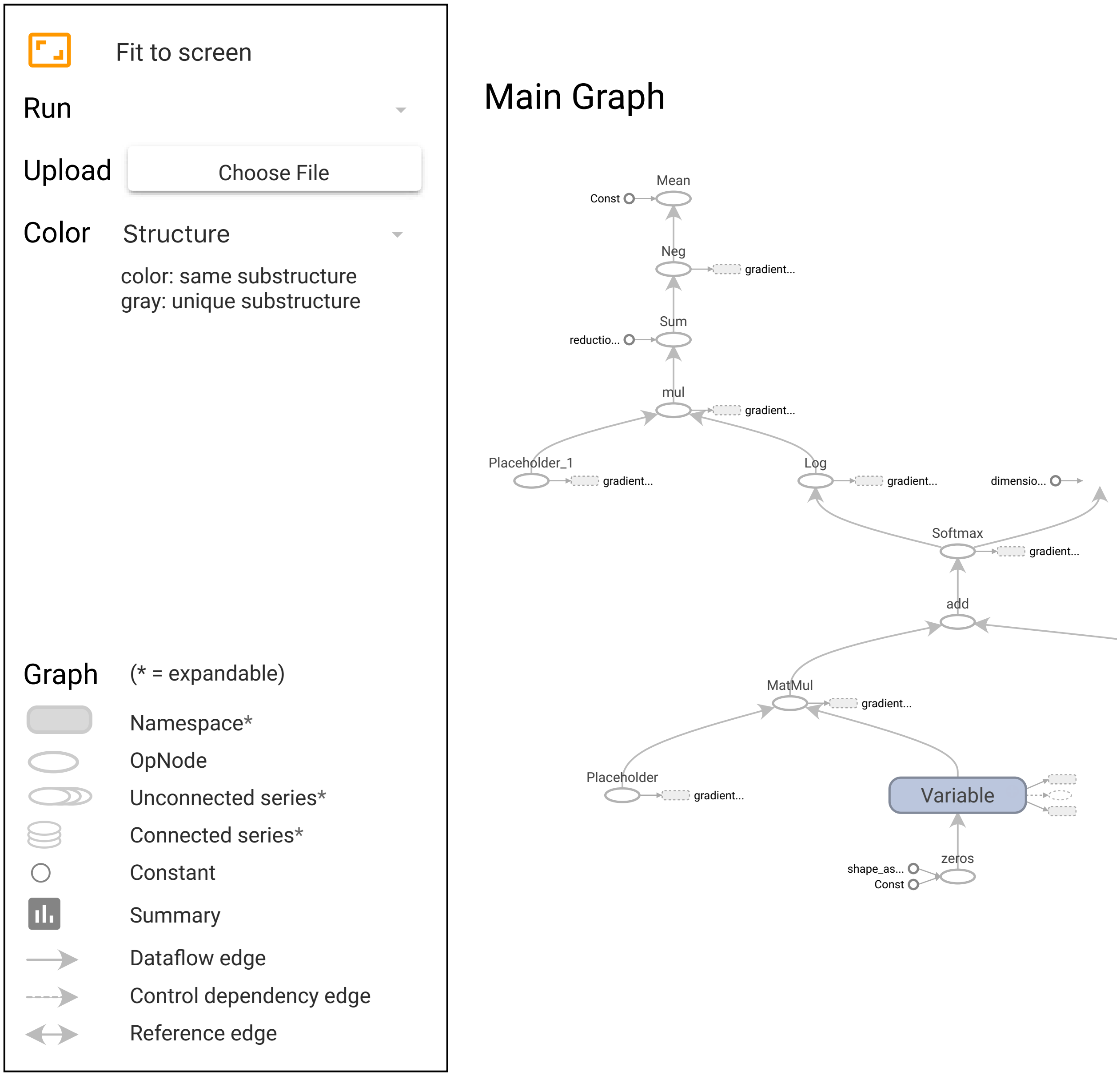
Data Visualization

In [22]:
```
plt.figure(figsize=(10,5))

plt.subplot(121)
plt.plot(steps, train_scores)
plt.xlabel("Epoch")
plt.ylabel("Training Accuracy")

plt.subplot(122)
plt.plot(steps, val_scores, color='orange')
plt.xlabel("Epoch")
plt.ylabel("Valdidation Accuracy")
plt.show()

print ("Training accuracy:", train_scores[-1])
print ("Validation accuracy:", val_scores[-1])
```




```
Training accuracy: 0.9171885873347251
Validation accuracy: 0.9083333333333333
```

```
In [23]: show_graph(tf.get_default_graph())
```



2.2: Vanilla Dense Neural Network

Now using the same technique as in the above multiclass logistic regression model, train a vanilla Dense Neural Network using the DIGITS dataset, with the characteristics listed below. Observe that the complexity of a Neural Network depends on the additional layers called 'Hidden Layers', which can extract relevant features and latent information in the data. Compare the number of weights and bias terms (model parameters) in the DNN with the parameters in the simple multiclass logistic regression model that you trained in the above question.

- Input layer of size 8*8=64
- Three hidden layers of size 300,200,100
- Output layer of size 10
- TANH activation function in hidden layers
- The Dropout ratio should be set to 10%
- Use the cross entropy loss (Ref: tensorflow) and minibatch gradient descent as the optimization function.
- Hyperparameters:
  - Batch size: 100
  - Epochs: 1000
  - Learning rate =.001

In [14]:
```
# define features
feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(x_t
rain)

# dense neural network classifier
# two layers 300 and 100
# 10 clases
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,200, 100], n_
classes=10,
                                         feature_columns=feature_cols)

# if TensorFlow >= 1.1, make compatible with sklearn
dnn_clf = tf.contrib.learn.SKCompat(dnn_clf)
```

In [15]:
```
# fit the model, 4000 iterations
dnn_clf.fit(x_train, y_train, batch_size=100, steps=1000)
```

Out[15]: SKCompat()

In [16]:
```
# Calculate accuracies
from sklearn.metrics import accuracy_score

y_pred = dnn_clf.predict(x_test)
print('Accuracy',accuracy_score(y_test, y_pred['classes']))
```

```
Accuracy 0.975
```

In [17]:
```
# Define hyperparameters and input size
n_inputs = 8*8  # MNIST
n_hidden1 = 300
n_hidden2 = 200
n_hidden3 = 100
n_outputs = 10
```

In [18]:
```
# Reset graph
tf.reset_default_graph()
```

In [19]:
```
# Placeholders for data (inputs and targets)
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
keep_prob = tf.placeholder(tf.float32, name="keep_prob")
```

In [20]:
```
# Define neuron layers (ReLU in hidden layers)
# We'll take care of Softmax for output with loss function

def neuron_layer(X, n_neurons, name, activation=None):
    # X input to neuron
    # number of neurons for the layer
    # name of layer
    # pass in eventual activation function

    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])

        # initialize weights to prevent vanishing / exploding gradients
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)

        # Initialize weights for the layer
        W = tf.Variable(init, name="weights")
        # biases
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")

        # Output from every neuron
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

In [21]:
```
# Define the hidden layers
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",
                           activation=tf.nn.tanh)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                           activation=tf.nn.tanh)
    hidden3 = neuron_layer(hidden2, n_hidden3, name="hidden3",
                          activation=tf.nn.tanh)
    dropout = tf.layers.dropout(hidden3, keep_prob)
    logits = neuron_layer(dropout, n_outputs, name="outputs")
```

In [22]:
```
# Define loss function (that also optimizes Softmax for output):
with tf.name_scope("loss"):
    # logits are from the last output of the dnn
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                               logits=log
its)
    loss = tf.reduce_mean(xentropy, name="loss")
```

In [23]:
```
# Training step with Gradient Descent
learning_rate = 0.001

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

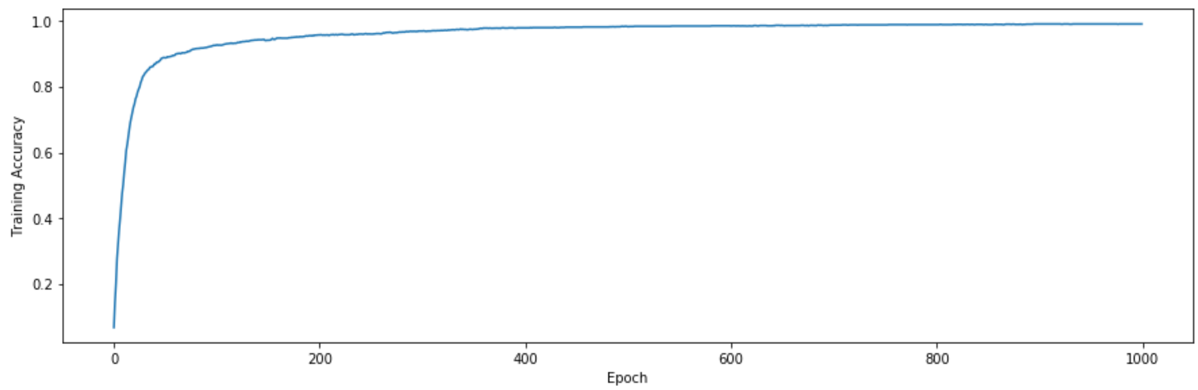
In [24]:
```
# Evaluation to see accuracy

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

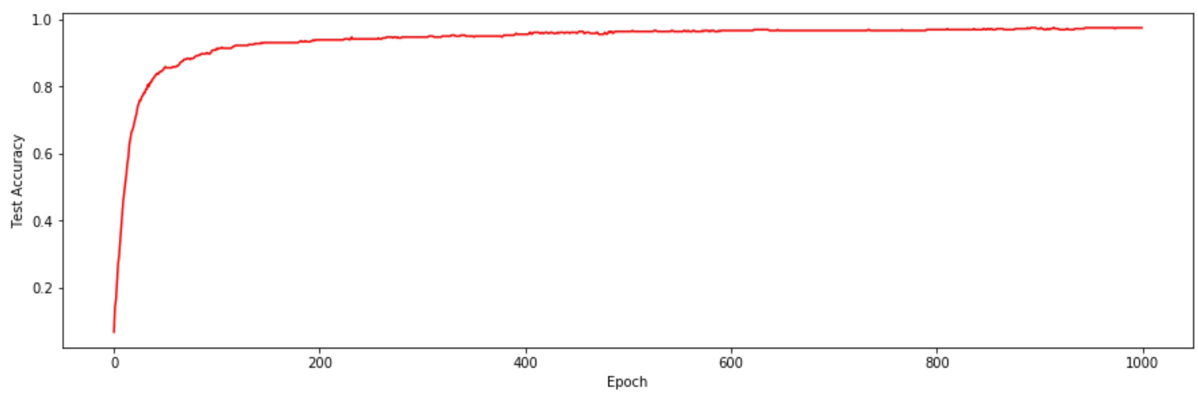
In [27]:
```
init = tf.global_variables_initializer()
saver = tf.train.Saver()

n_epochs = 1000
batch_size = 100

train_scoresdnn = []
test_scoresdnn = []

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(x_train.shape[0] // batch_size):
            offset = np.random.randint(0, labels_train.shape[0] - batch_
size - 1)
            batch_xs = x_train[offset:(offset + batch_size), :]
            batch_ys = y_train[offset:(offset + batch_size)]
            sess.run(training_op, feed_dict={X: batch_xs, y: batch_ys, k
eep_prob:0.9})
        train_scoresdnn.append(accuracy.eval(feed_dict={X: x_train, y: y
_train, keep_prob:0.9}))
        test_scoresdnn.append(accuracy.eval(feed_dict={X: x_test, y: y_t
est, keep_prob:0.9}))

    save_path = saver.save(sess, "./my_model_final.ckpt") # save model
```

In [28]:
```
print("Test Accuracy: ", test_scoresdnn[-1])
```

```
Test Accuracy:  0.975
```

In [29]:
```
f = plt.figure(figsize=(15,10))

ax = f.add_subplot(211)
ax.plot(range(n_epochs), train_scoresdnn)
ax.set_xlabel("Epoch")
ax.set_ylabel("Training Accuracy")

ax3 = f.add_subplot(212)
ax3.plot(range(n_epochs), test_scoresdnn, color='red')
ax3.set_xlabel("Epoch")
ax3.set_ylabel("Test Accuracy")

plt.show()
```





In [34]:
```
show_graph(tf.get_default_graph())
```