Project 3: Softmax Regression and NN

December 14, 2022

0.1 Part I Softmax Regression [45%]

Part - 1 : Show that the probabilities sum to 1

K = number of classes

$P[y=i] = \frac{e^{\vec{w}_i \cdot \vec{x}}}{\sum_{j=0}^{K-1}\left(e^{\vec{w}_j \cdot \vec{x}}\right)}$

$\sum_{i=0}^{K-1}\left(P[y=i]\right) = \sum_{i=0}^{K-1}\left(\frac{e^{\vec{w}_i \cdot \vec{x}}}{\sum_{j=0}^{K-1} e^{\vec{w}_j \cdot \vec{x}}}\right)$

$= \frac{\sum_{i=0}^{K-1} e^{\vec{w}_i \cdot \vec{x}}}{\sum_{j=0}^{K-1} e^{\vec{w}_j \cdot \vec{x}}}$

i and j span the same range so

$\sum_{i=0}^{K-1}\left(P[y=i]\right) = \frac{\sum_{i=0}^{K-1} e^{\vec{w}_i \cdot \vec{x}}}{\sum_{j=0}^{K-1} e^{\vec{w}_j \cdot \vec{x}}} = 1$

Part - 2 : What are the dimensions of W? X? WX?

Symbols

- F : Features per example

- N : Number of examples

- K : Number of classes

$dimW = [K \times F]$

$dimX = [F \times N]$

$dim\,[W \times X] = [K \times N]$

0.1.1 Qsr 2 - See softmax.py

0.1.2 Qsr 3

Part - 1 : Show that this does not affect the predicted probabilities.

K = number of classes

$P[y = i] = \frac{e^{\vec{w}_i \cdot \vec{x}}}{\sum_{j=0}^{K-1}\left(e^{\vec{w}_j \cdot \vec{x}}\right)}$

with the subtraction of max(W_X) the above becomes

$= \frac{e^{\vec{w}_i \cdot \vec{x} - max(\vec{w}_i \cdot \vec{x})}}{\sum_{j=0}^{K-1}\left(e^{\vec{w}_j \cdot \vec{x} - max(\vec{w}_i \cdot \vec{x})}\right)}$

spliting the max out from the e

$= \frac{e^{\vec{w}_i \cdot \vec{x}} \times e^{-max(\vec{w}_i \cdot \vec{x})}}{\sum_{j=0}^{K-1}\left(e^{\vec{w}_j \cdot \vec{x}} \times e^{-max(\vec{w}_i \cdot \vec{x})}\right)}$

pull out the $e^{-max(\vec{w}_i \cdot \vec{x})}$ on the bottom summation

$= \frac{e^{\vec{w}_i \cdot \vec{x}} \times e^{-max(\vec{w}_i \cdot \vec{x})}}{e^{-max(\vec{w}_i \cdot \vec{x})} \times \sum_{j=0}^{K-1}\left(e^{\vec{w}_j \cdot \vec{x}}\right)}$

now the $e^{-max(\vec{w}_i \cdot \vec{x})}$ cancel out

$= \frac{e^{\vec{w}_i \cdot \vec{x}}}{\sum_{j=0}^{K-1}\left(e^{\vec{w}_j \cdot \vec{x}}\right)}$

Thus, subtracting the max(W_X) doesn't affect $P[y = i]$

Part - 2: Why might this be an optimization over using W_X? Justify your answer.

This makes all values of W_X <= 0 (i.e. negative). When taking the exponential (np.exp()), the result will always be in the range (0,1]. This is an optimization for floating point arithmetic. When dividing floating point numbers, dividing small numbers by large numbers will cause computational errors. This is a significant risk because we are using an exponential function. To avoid this, scaling the softmax input helps improve computational accuracy.

0.1.3 Qsr 4

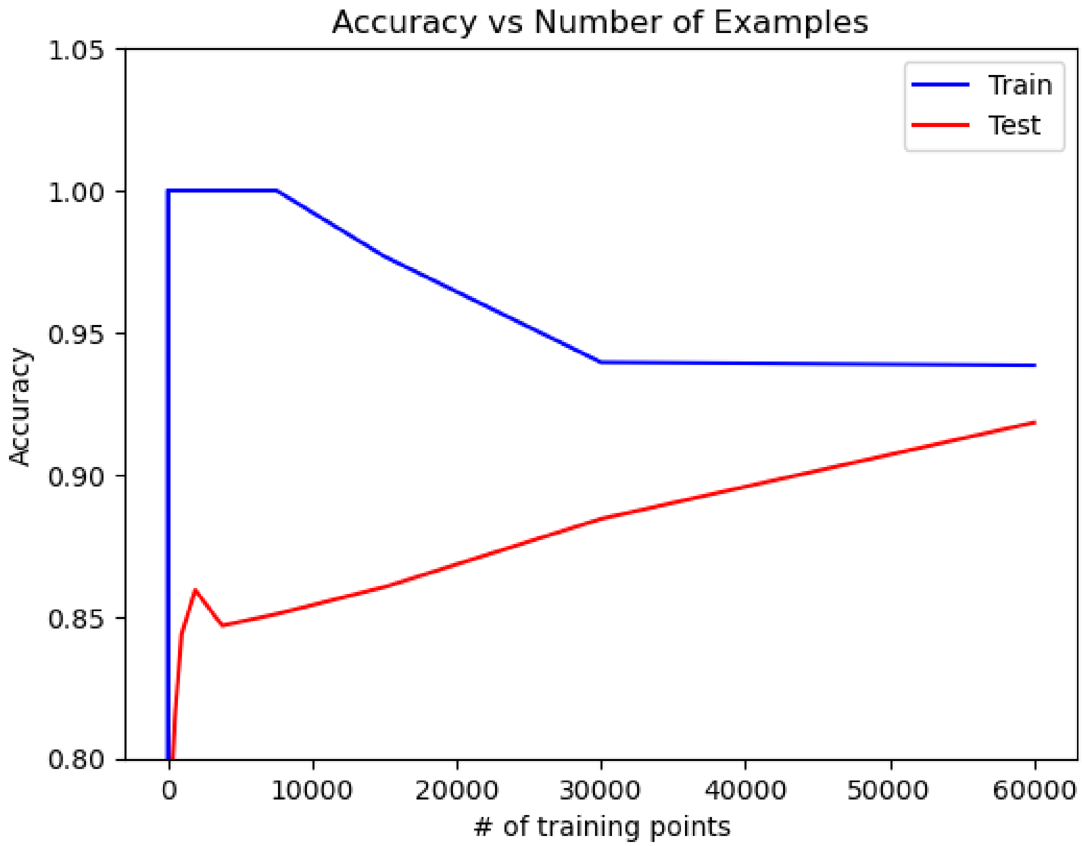
Do you observe any overfitting or underfitting? Discuss and expain what you observe.

There's a slight bit of overfitting because the training accuracy is still higher than the test accuracy, but we do see that as the number of examples increases, there less overfitting because more training examples help the model's ability to generalize.

There appears to be significant overfitting below 5,000 training examples, but training accuracy plateaus after 30,000 examples. We could speculate that with 100,000 we may see test accuracy reach ~94% because test accuracy is increasing linearly with the number of training examples.

There doesn't seem to be underfitting because our test accuracy is fairly high and inline with this type of classifier. Human performance on MNIST is about 98% with a majority of errors related to mislabeling or illegible examples.

[26]:
```
import matplotlib.pyplot as plt
import runClassifier
import softmax
from utils import *

exSize = 28*28
# 10 digits
numClasses = 10
# Regularizer coefficient
reg = 0.0001

X, Y = loadMNIST('data/train-images.idx3-ubyte', 'data/train-labels.idx1-ubyte')
testX, testY = loadMNIST('data/t10k-images.idx3-ubyte', 'data/t10k-labels.
 ↪idx1-ubyte')

size, trna, tsta = runClassifier.learningCurve(
                            softmax.SoftmaxRegression(numClasses, exSize),
                            numClasses,exSize,
                            X,Y,testX, testY)

plt.plot(size, trna, 'b-',
         size, tsta, 'r-')
plt.legend( ('Train', 'Test') )
plt.xlabel('# of training points')
plt.ylabel('Accuracy')
plt.title("Accuracy vs Number of Examples")
plt.ylim([.8, 1.05])
plt.show()
```


Accuracy vs Number of Examples
Train
Test
Accuracy
1.05
1.00
0.95
0.90
0.85
0.80
0
10000
20000
30000
40000
50000
60000
# of training points

0.2 # Part 2 : Neural Networks

0.2.1 Qnn 1

Part - 2

[27]:
```
from nn import *

x_train, label_train = loadMNIST('data/train-images.idx3-ubyte', 'data/
 ↪train-labels.idx1-ubyte')
x_test, label_test = loadMNIST('data/t10k-images.idx3-ubyte', 'data/t10k-labels.
 ↪idx1-ubyte')
y_train = onehot(label_train)
y_test = onehot(label_test)

model = NN(Relu(), SquaredLoss(), hidden_layers=[128, 128], input_d=784,␣
 ↪output_d=10)
model.print_model()
training_data, dev_data = {"X":x_train, "Y":y_train}, {"X":x_test, "Y":y_test}
from run_nn import train_1pass
model, plot_dict = train_1pass(model, training_data, dev_data,␣
 ↪learning_rate=1e-2, batch_size=64)
```

```
activation:Relu
loss function:SquaredLoss
Layer 1 w:(128, 784)    b:(128, 1)
Layer 2 w:(128, 128)    b:(128, 1)
Layer 3 w:(10, 128)     b:(10, 1)
#Samples  6400  loss:0.48946    dev_acc:0.57050
#Samples 12800  loss:0.33665    dev_acc:0.69190
#Samples 19200  loss:0.29153    dev_acc:0.75070
#Samples 25600  loss:0.26063    dev_acc:0.78970
#Samples 32000  loss:0.24242    dev_acc:0.81270
#Samples 38400  loss:0.22898    dev_acc:0.82960
#Samples 44800  loss:0.21810    dev_acc:0.84340
#Samples 51200  loss:0.20687    dev_acc:0.85420
#Samples 57600  loss:0.19843    dev_acc:0.86110
```

Yes the loss goes down.

Part - 3

Running **python3 run_nn.py** gave the following output with a dev_accuracy of **0.94720**.

```
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 784)    b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.21414    dev_acc:0.83510
Epoch   2/20    loss:0.19313    dev_acc:0.88050
Epoch   3/20    loss:0.15603    dev_acc:0.89550
Epoch   4/20    loss:0.15659    dev_acc:0.90790
Epoch   5/20    loss:0.13327    dev_acc:0.91570
Epoch   6/20    loss:0.12474    dev_acc:0.92130
Epoch   7/20    loss:0.12152    dev_acc:0.92550
Epoch   8/20    loss:0.10968    dev_acc:0.92920
Epoch   9/20    loss:0.10461    dev_acc:0.93190
Epoch  10/20    loss:0.12148    dev_acc:0.93480
Epoch  11/20    loss:0.10291    dev_acc:0.93610
Epoch  12/20    loss:0.09557    dev_acc:0.93920
Epoch  13/20    loss:0.09403    dev_acc:0.94060
Epoch  14/20    loss:0.10219    dev_acc:0.94090
Epoch  15/20    loss:0.08904    dev_acc:0.94220
Epoch  16/20    loss:0.08797    dev_acc:0.94350
Epoch  17/20    loss:0.08685    dev_acc:0.94530
Epoch  18/20    loss:0.09967    dev_acc:0.94590
Epoch  19/20    loss:0.07163    dev_acc:0.94700
Epoch  20/20    loss:0.08838    dev_acc:0.94720
```

Part - 4

When initializing the weight matrix, in some cases it may be appropriate to initialize the entries as small random numbers rather than all zeros. Give one reason why this may be a good idea.

If initialized with all zeros, the weights may move the same way such that only one optimization minimum is found. With random initialization with floating points between -1 and 1, we could find more minima and more effectively use floating point processing.

0.2.2 Qnn 2 - PCA

Do dimension reduction with PCA. Try with different dimensions. Can you observe the trade-off in time and acc? Plot training time v.s. dimension, testing time v.s dimension and acc v.s. dimension. Visualize the principal components.

Note: Plots were based on **new_run_nn.py** run by Wei. Leo's computer was not able to run PCA for dimensions greater than 500. Data was saved by Wei into a pickle. Code can be seen at the bottom of **new_run_nn.py**. Two sets of dimensions are available. Wei's dimensions and Leo's dimensions.

Explain what you did and what you found. Comment the code so that it is easy to follow. Support your results with plots and numbers. Provide the implementation so we can replicate your results.

Dimensionality reduction with PCA was selected as the extra credit option. The file **new_run_nn.py** was created for the purposes of demonstrating PCA. The PCA reduction was performed in **new_run_nn.py** and relies on the model provided in **nn.py**. To observe results we used 14 dimensions, and ran a training/test sequence similar to **run_nn.py**. Based on the results, we can see that fewer dimensions significantly reduce training time. Testing/forward passes do not benefit as much because it's very fast to start with, but there is a slight uptrend as more dimensions were run. Accuracy goes up but appears to have significant diminishing returns after 8 PCs.

PCs were visualized based on the fact that the raw data is 784 dimensional due to a 28x28 image getting flattened. If we unflatten the PCs, we should get something analogous to the images we saw in the raw data. Looking at the visualization of PCs, we can see something that supports the diminishing accuracy returns with greater than 8 PCs. The first 10 PCs vaguely resemble number like symbols. After the 10th PC, the PCs start to look noisier and noisier with less information resembling a number. After the 18th PC, the PCs begin to look like noise. This is also where we see accuracy plateau at around 94% at the point between 16 and 20 PCs. This can be seen in the pickle file "time_stuff_LEO.pickle". PCs were visualized with the highest eigenvalue vector at the top left and the lowest at the bottom right.

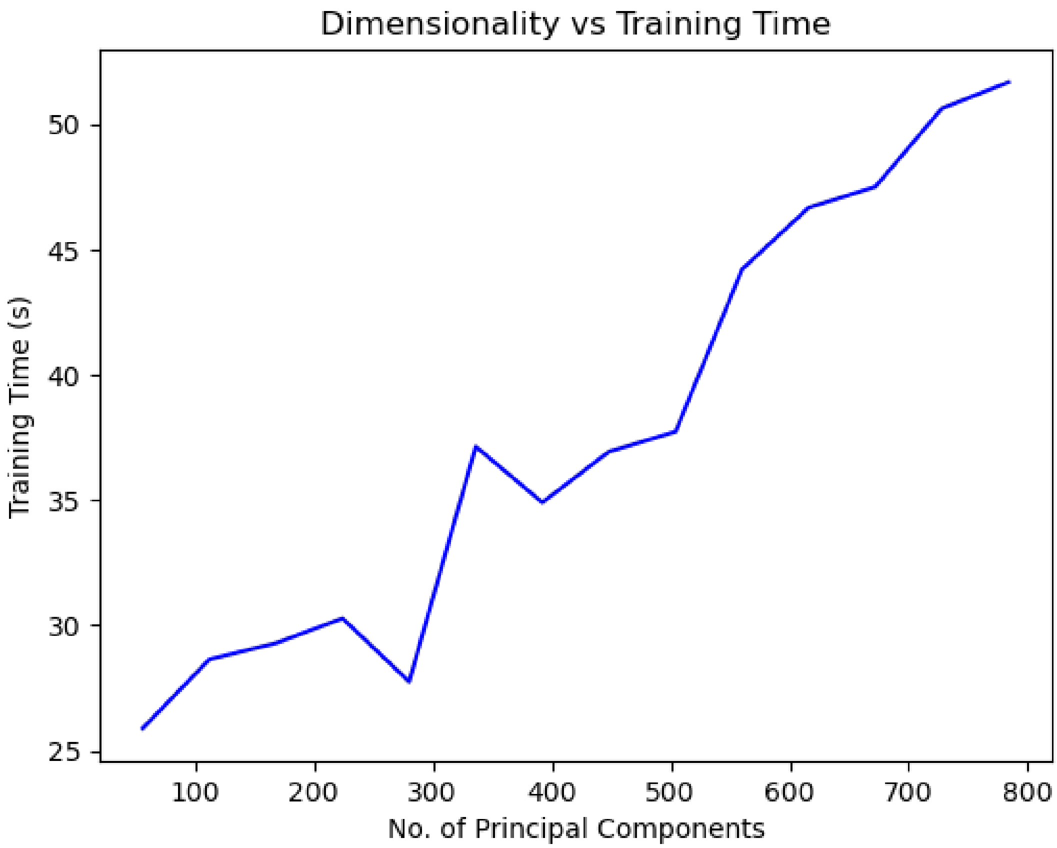
To run the pickles, just remove the ".txt" at the end.

[40]:

```
import pickle

with open("time_stuff_WEI.pickle", "rb") as ofile:
    data_time = pickle.load(ofile)
```

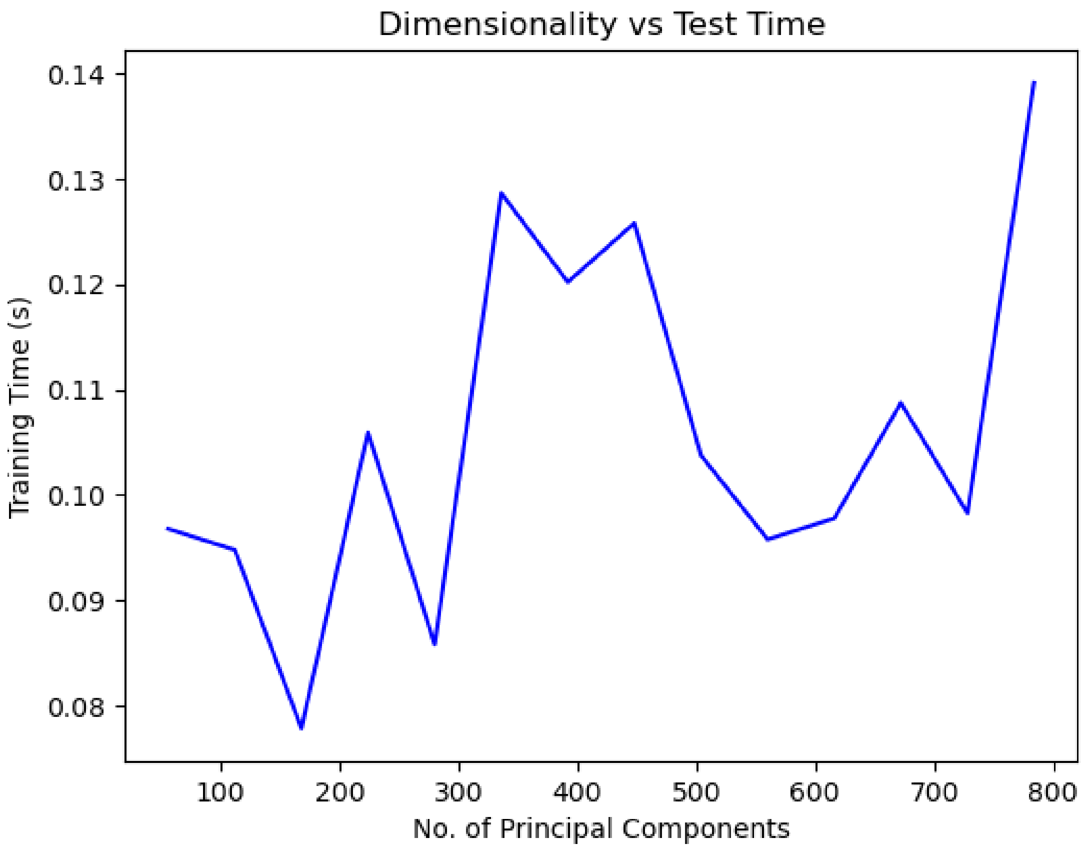
[46]:

```
plt.plot(data_time["dimensions"], data_time["train_time"], 'b-')
plt.xlabel('No. of Principal Components')
plt.ylabel('Training Time (s)')
plt.title("Dimensionality vs Training Time")
plt.show()
```



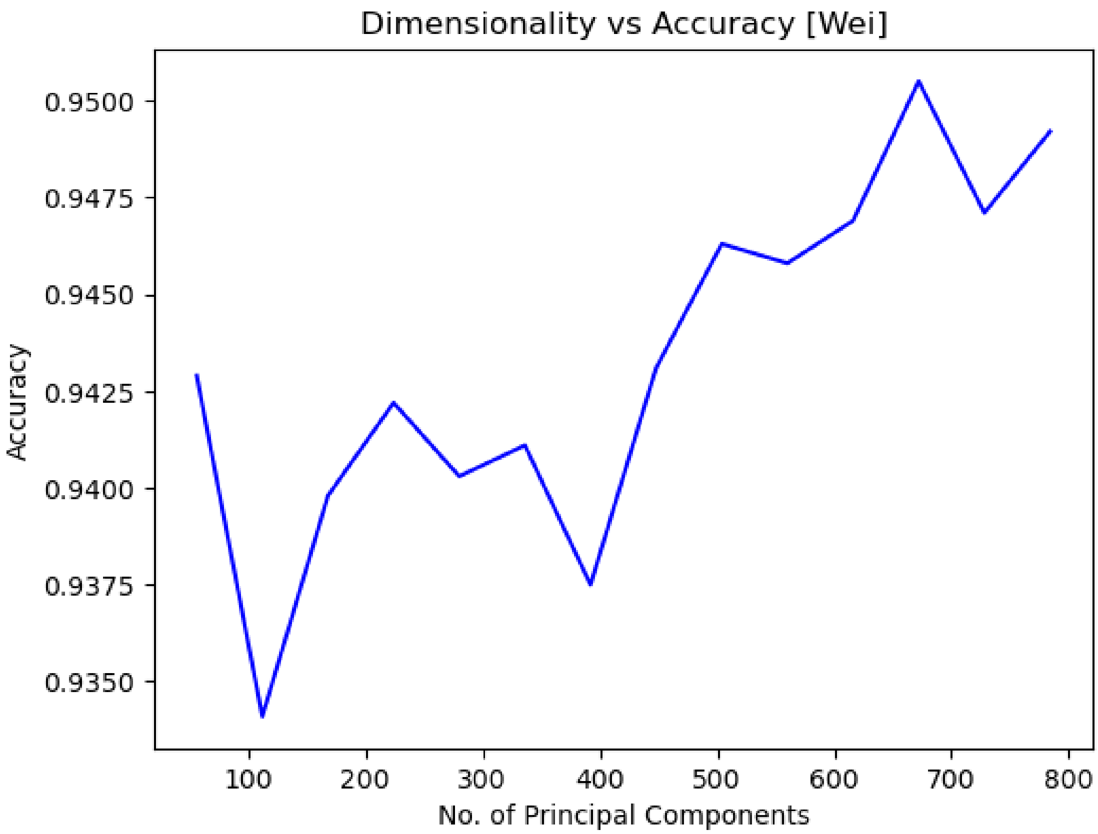
[35]:

```
plt.plot(data_time["dimensions"], data_time["test_time"], 'b-')
plt.xlabel('No. of Principal Components')
plt.ylabel('Training Time (s)')
plt.title("Dimensionality vs Test Time")
plt.show()
```

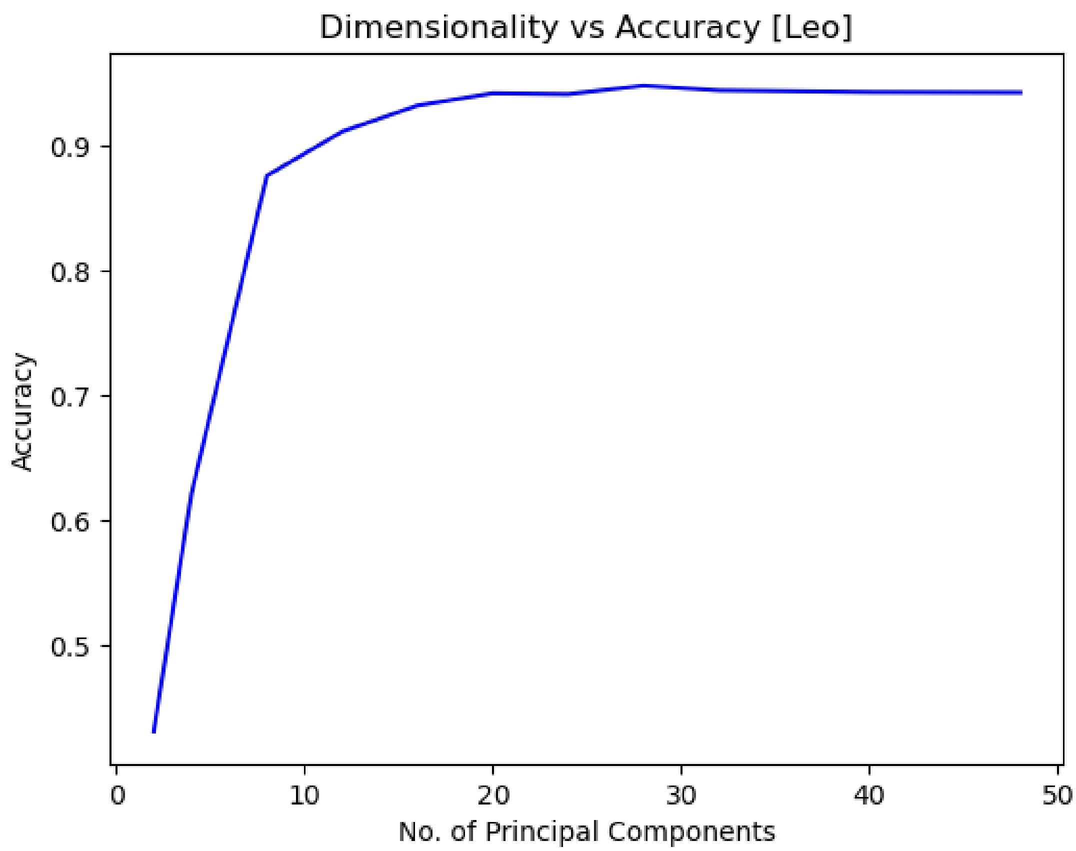


[47]:

```
plt.plot(data_time["dimensions"], data_time["test_acc"], 'b-')
plt.xlabel('No. of Principal Components')
plt.ylabel('Accuracy')
plt.title("Dimensionality vs Accuracy [Wei]")
plt.show()
```

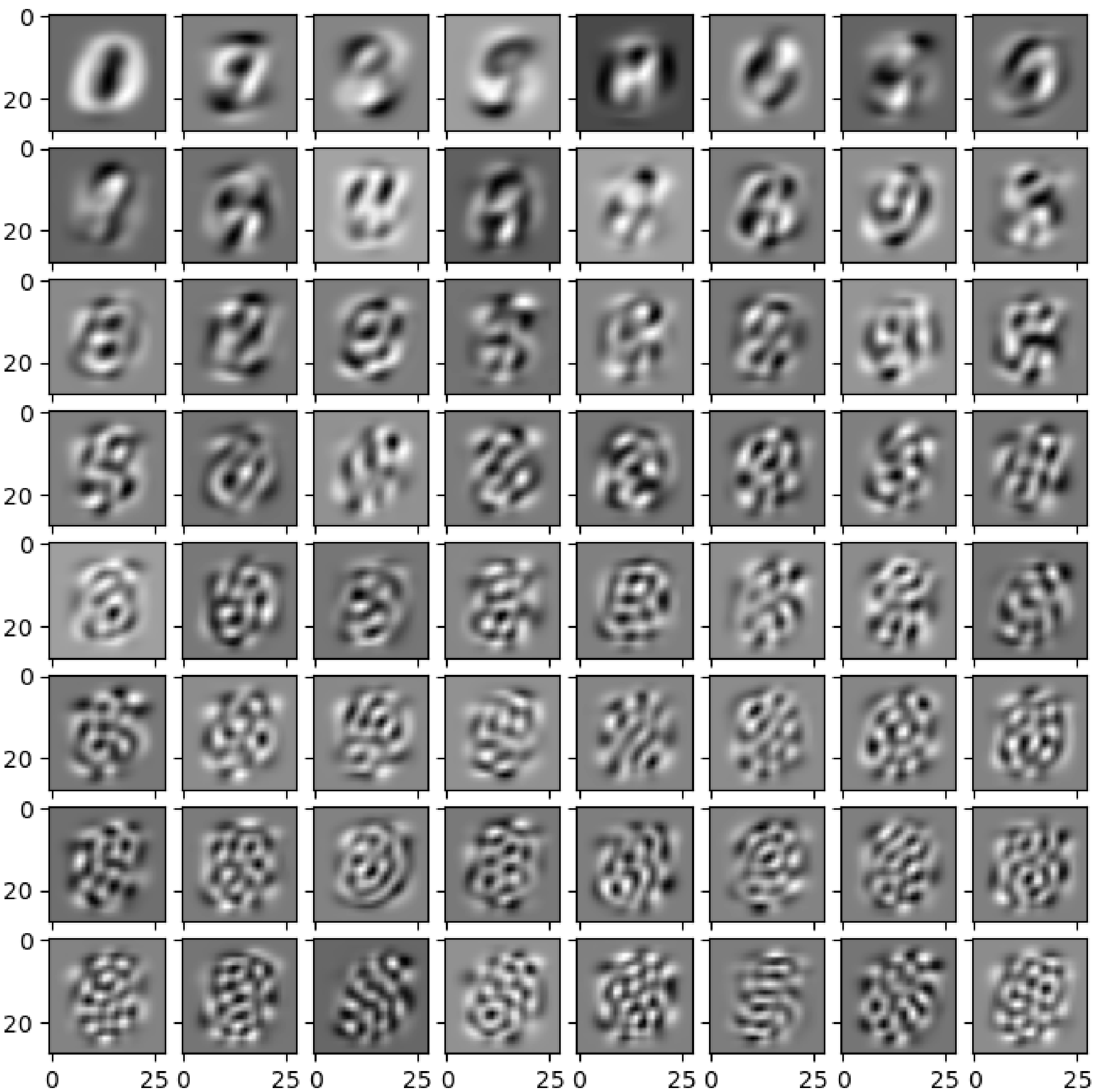


[48]:
```
with open("time_stuff_LEO.pickle", "rb") as ofile:
    data_time = pickle.load(ofile)

plt.plot(data_time["dimensions"], data_time["test_acc"], 'b-')
plt.xlabel('No. of Principal Components')
plt.ylabel('Accuracy')
plt.title("Dimensionality vs Accuracy [Leo]")
plt.show()
```



[37]:

```python
from matplotlib import pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
from sklearn.decomposition import PCA
import utils

# load data
x_train, label_train = utils.loadMNIST('data/train-images.idx3-ubyte', 'data/
↪train-labels.idx1-ubyte')
x_test, label_test = utils.loadMNIST('data/t10k-images.idx3-ubyte', 'data/
↪t10k-labels.idx1-ubyte')

# setup PCA
feat_size = 64

pca10 = PCA(n_components=feat_size)
pca10.fit(x_train.T) # requires shape [examples,features]

x_train_pca10 = pca10.transform(x_train.T)
x_test_pca10 = pca10.transform(x_test.T)

# setup displaying images
img_dim = (28,28)
img_display = (8,8)
img_cnt = 64

fig = plt.figure(figsize=img_display)
grid = ImageGrid(fig, 111, nrows_ncols=(8, 8), axes_pad=0.1)

for ax, im in zip(grid, pca10.components_[:img_cnt]):
    # Iterating over the grid returns the Axes.
    ax.imshow(np.reshape(im,img_dim), cmap=plt.get_cmap('gray'))

plt.title("Visualization of Principal Components")
plt.show()
```


0
20
0
25