Olamide OLADEJI.
Project 2 Report: 6.036/6.862 MIT Spring 2017

Problem 1.

1.1. Code Implemented

1.2. Code Implemented

1.3 Code Implemented

1.4 Final test error = 0.1005

1.5 How temperature affects probability of sample x(i) being assigned a label that has a large theta.

The lower the value of tempParameter, the higher the probability of a sample x(i) being assigned a relatively large theta, as the softmax exponential accentuates the contribution of the large theta fraction to the overall sum of exponentials such that the probability tends to 1.

The lower the value of tempParameter the lower the probability of a sample x(i) being assigned to a relatively small theta.

1.6

i. With tempParemeter = 1
Error Rate = 0.1005

ii. With tempParameter= 0.5,
Error Rate = 0.084

iii. With tempParameter = 2
Error Rate = 0.1261

We can see that as you increase the tempParameter, your error rate increases. The lower tempParameter gave lower test errors (i.e. the probability matched), implying that the probability distribution of the dataset has a small variance.

1.7

New ErrorRate with new labels. Is 0.0768
Compared to old error rate of 0.1005, we see that this is much less.
The difference arises because the mod3 factor streamlines the maximum difference between estimated label and correct label to be 2 instead of 9 as in original. Also, many points which

would have originally had losses with previous labelling are now with losses of 0 because their mod3 values are the same for estimated and corrected.

1.8 Implemented.

1.9 Test Error rate with training on new labels is 0.1872

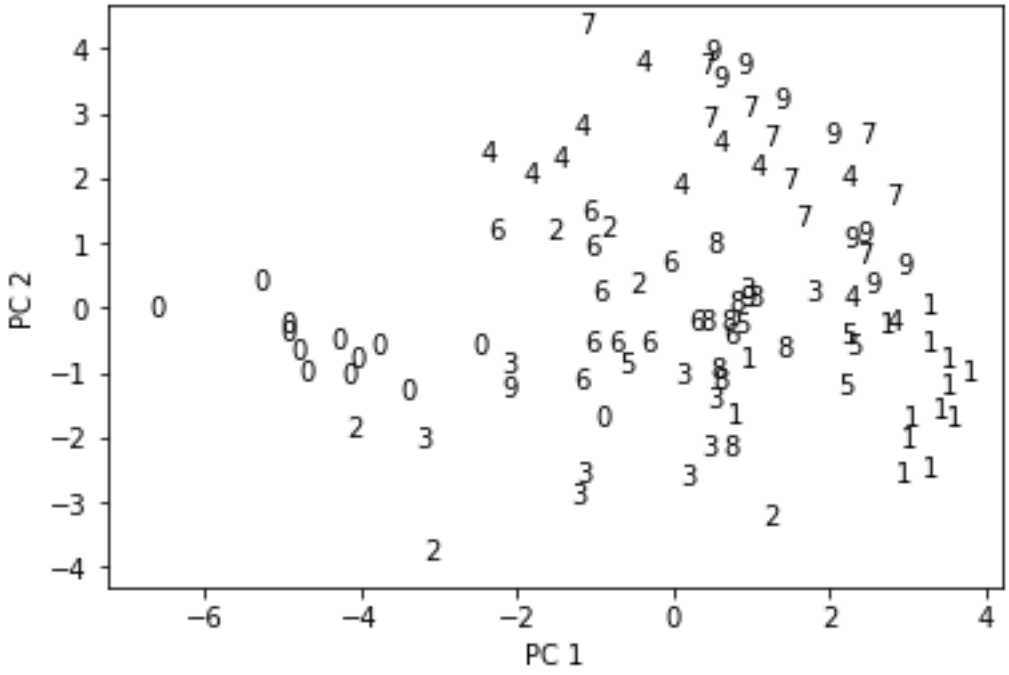
Problem 2

2.1 Code Implemented

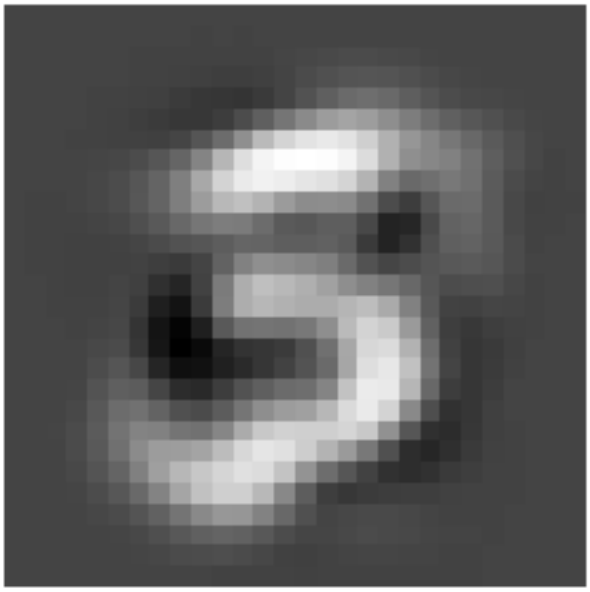
2.2 Code Implemented

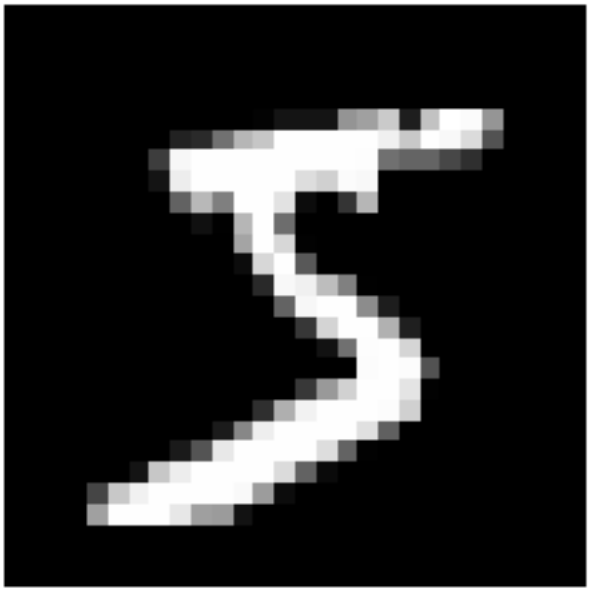
2.3 Error on Test Set using PCA = 0.1483

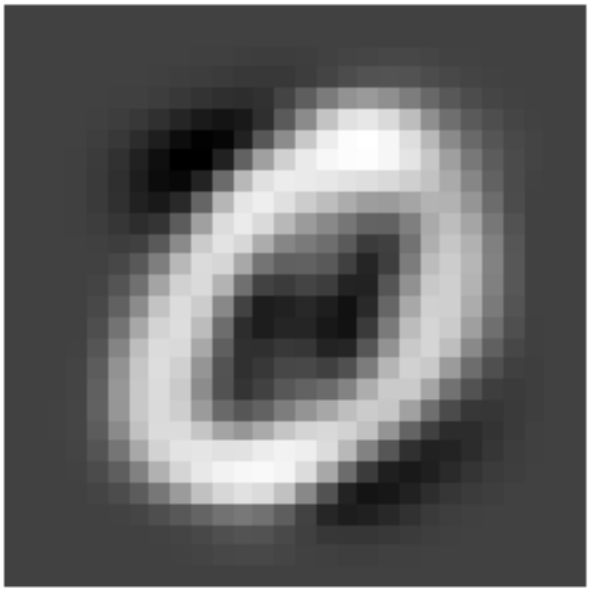
2.4 First 100 Images.

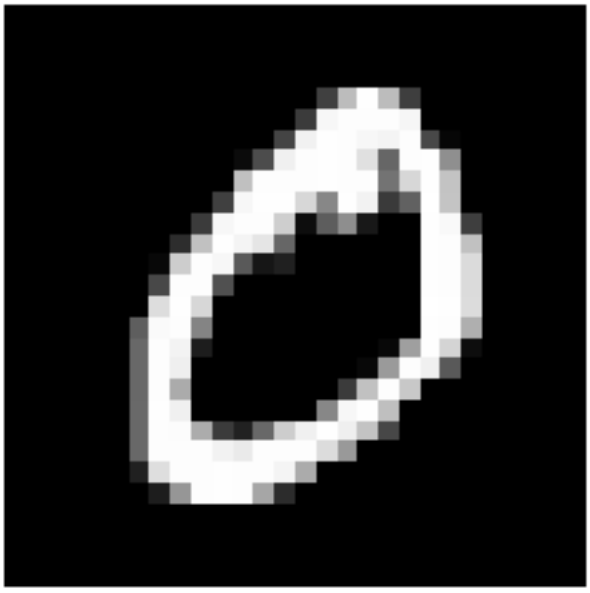


2.5 Reconstruction of first two MNIST images and their originals.









2.6: CUBIC Features.

(xT x’ + 1) ^3 = (x1x1’ + x2x2’ + 1)^3

= (x1x1’ + x2x2’ + 1) (x1x1’ + x2x2’ + 1)^2

Expanding we have;

(x1x1')^3 + 3(x1x1')^2 (x2x2') + 3(x1x1')^2 + 3(x1x1')(x2x2')^2 + 6(x1x1'x2x2') + 3(x1x1') + (x2x2')^3 + 3(x2x2')^2 + 3(x2x2') + 1

*We can thus write phi(x) = [x1^3, √3*x1^2*x2, √3*(x1^2), √3*x1*x2^2, √6*x1*x2, √3*x1, (x2^3), √3*(x2^2), √3*x2, 1]*
Which is a 10-dimensional vector as required.

2.7 Error with CubicFeatures
Obtained Error = 0.0865

Question 3

3.1 Implemented (code attached)

3.2 Implemented (code attached). All errors under 0.15 as required.

3.3 Implemented (code attached). Prediction tests passed for all.

3.4. *Improving the learning rate to improve network efficiency in terms of training and accuracy*: One way to improve the learning rate for network efficiency is to decay it over time.
There are many ways to decay or anneal the learning rate, one of which is a (1/t) decay in which the learning rate $\alpha=\alpha_0/(1+kt)$ where alpha_0 and k are hyper-parameters and t is training iteration count.

3.5. *Danger of having too many hidden units in network:* The danger is overfitting to training data and poor generalization on test data.

3.6. *What would happen in terms of training and test accuracy if code is run for more epochs?* If I run for more epochs, the training accuracy would have increased as empirical risk/ error becomes better optimized. The testing accuracy may increase then reduce after a point due to overfitting of the neural network on training data.

3.7 *How to optimize amount of epochs*: One method is to run the training data of several epochs and to record statistics (the mean squared error) for each n=1,2,… epochs. The epoch number with least MSE is chosen

Question 4.
4.1.a.
Accuracy on test set: 0.9171

4.1.b

What did not work.
Initially Increasing learning rate, lr past 0.005.
Modifying the neurons within dense hidden layers to 500 reduced accuracy.
Modifying the neurons within dense hidden layers to 600 from 700 did not increase accuracy.
Increasing the momentum to above 0.91 wasn't helping too much with accuracy.
Adding extra dense layers after above did not provide any significant gains in test accuracy.

What worked

- Initially increasing lr to 0.005 worked well
- Adding two more hidden dense relu activated layers of 128 neurons worked – 0.9633
- Modifying the neurons within dense hidden layers to about 700 and then 800 got accuracy up to ~0.97
- Increased the momentum to 0.9 now seemed to help increase it 0.978.
- Reducing the number of hidden layer to just one and then adjusting the neurons in this dense layer to 1270.

Finally settled on following architecture:

With Test Accuracy 0.9802

```
model.add(Dense(output_dim=1270, input_dim=784))
model.add(Activation("relu"))
model.add(Dense(output_dim=10))
model.add(Activation("softmax"))

model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.0048, momentum=0.91),
metrics=["accuracy"])
```

i.e a two layer network with one fully connected layer of 1270 neurons and relu activation function. And one output layer of 10 neurons and softmax output activation.

4.2a
Implemented (Code attached).

```
0.2190 - acc: 0.9307 - val_loss: 0.0532 - val_acc: 0.9848
 9984/10000 [============================>.] - ETA: 0sLoss on test set:0.0594630632247 Accuracy on test
set: 0.9817
```

For one Epoch as seen;
Training Accuracy = 0.9307
Validation Accuracy = 0.9848
Test Accuracy = 0.9817

Question 5.

5.0
64 Images inside the training and testing data.
Size of each image: 42x28 pixels

27 model.compile(loss='categorical_crossentropy',optimizer='sgd', metrics=['accuracy'], loss_weights=[0.5, 0.5])

Model.compile function configures the learning process.
The input argument; 'loss' shows that we are using "categorical_crossentropy" loss in this case.
Optimizer= 'sgd' means that we are using stochastic gradient descent as our optimization algorithm.
Metrics=['accuracy'] means that we measure want to evaluate the accuracy on training (as well as validation data) since this is a classification problem.
Loss_weights as [0.5, 0.5] means we want the loss weight to be same ratio for main and auxiliary inputs.

28 model.fit(X_train, [y_train[0], y_train[1]], nb_epoch=nb_epoch, batch_size=batch_size, verbose=1)

Model.fit function actually trains the model. X_train is the training input data, y_train are the labels, nb_epoch refers to number of epochs, while setting the batch_size means every batch of input would have a batch shape of (batch_size, 42,28)
verbose: 0 for no logging to stdout, 1 for progress bar logging, 2 for one log line per epoch.

y_train[0] is training data of labels of first digit while y_train[1] is training data of labels of second digit of two-digit images.

5.1
After code was completed and model built as required:
Mlp.py
Initial Model Architecture: Functional.

- digit_input = Input(shape=(1, img_rows, img_cols))
- Flattening
- 64 neuron dense layer with relu activation

- Two outputs
    - 1st is a 10-neuron dense layer with softmax activation
    - 2nd is also 10-neuron dense layer with softmax activation.

First Run after Model was built.

```
Evaluation on test set: {'main_output_loss': 0.33931382107734681, 'main_output_acc':
0.90475000000000005, 'loss': 0.34908425879478455, 'auxiliary_output_loss': 0.35885469555854799,
'auxiliary_output_acc': 0.89349999999999996}
```

Test Accuracy on first number = 0.90475
Test Accuracy on second number = 0.8935
Total Training time for 30 Epochs = 62s

5.2

After code was completed and model built as required:

Conv.py

Initial Model Architecture: Functional

- digit_input = Input(shape=(1, img_rows, img_cols))
- A 2D Conv layer (3 x 3) with 8 filters and relu activation
- A maxpooling layer with (2,2) size filter and (2,2) stride
- A 2D conv layer (3 x 3) with 16 filters and relu activation
- A maxpooling layer with (2,2) size filter and default/ (1,1), stride
- Flattening
- then a 64 neuron Dense layer, relue activation
- Dropout with rate 0.5
- Two outputs:
    - 1st is a dense layer (10 neurons and softmax activation)
    - 2nd is also 10-neuron dense layer and softmax activation.

First Run after Model was built.

```
3968/4000 [============================>.] - ETA: 0sEvaluation on test set: {'auxiliary_output_acc':
0.86524999999999996, 'main_output_acc': 0.90249999999999997, 'main_output_loss': 0.4189267945289612,
'auxiliary_output_loss': 0.49473580932617189, 'loss': 0.45683129954338075}
```

Test Accuracy on first number = 0.9025
Test Accuracy on second number = 0.86525
Training time 1st Epoch = 65s
Training time 2nd Epoch = 64s
Training time 3rd Epoch = 64s

5.3: Changing Parameters and Settings for models above.

a. **For Conv.py.**

i. Edited Model Architecture 1:

- Same Architecture as in conv.py in 5.2 but with ‘adam’ optimizer instead of ‘sgd’
- Nb_epoch = 3

Results

```
Evaluation on test set: {'loss': 0.15709160351753235, 'main_output_loss': 0.13490725541114806,
'auxiliary_output_loss': 0.17927595084905623, 'main_output_acc': 0.9625000000000002,
'auxiliary_output_acc': 0.9505000000000001}
```

Test Accuracy on First number = 0.9625
Test Accuracy on Second number = 0.9505
Training time 1st Epoch = 71s
Training time 2nd Epoch = 67s
Training time 3rd Epoch = 70s

ii. Edited Model Architecture 2**: (saved as extra_model1.py)**
Added an extra dense 64 neuron hidden layer with relu activation. Maintained 'adam' optimizer, nb_epoch=3
Model Architecture: Functional as follows

- digit_input = Input(shape=(1, img_rows, img_cols))
- A 2D Conv layer (3 x 3) with 8 filters and relu activation
- A maxpooling layer with (2,2) size filter and (2,2) stride
- A 2D conv layer (3 x 3) with 16 filters and relu activation
- A maxpooling layer with (2,2) size filter and default/ (1,1), stride
- Flattening
- then a 64 neuron Dense layer, relu activation
- then another 64 neuron Dense layer, relu activation
- Dropout with rate 0.5
- Two outputs:
    - 1st is a dense layer (10 neurons and softmax activation)

2nd is also 10-neuron dense layer and softmax activation

Results

```
3968/4000 [============================>.] - ETA: 0sEvaluation on test set: {'auxiliary_output_loss':
0.16436434197425842, 'main_output_acc': 0.9675000000000003, 'auxiliary_output_acc': 0.9415,
'main_output_loss': 0.11057035151124001, 'loss': 0.13746734631061555}
```

Test Accuracy on First number = 0.9675
Test Accuracy on Second number = 0.9415
Training time 1st Epoch = 79s
Training time 2nd Epoch = 77s
Training time 3rd Epoch = 78s

iii. Edited Model Parameter/Setting 3.
Observing the effect of increasing training time.
Modified epoch number, nb_epoch from '3' to '10'.
Used following architecture as in 5.3a but with more epochs:

- digit_input = Input(shape=(1, img_rows, img_cols))
- A 2D Conv layer (3 x 3) with 8 filters and relu activation
- A maxpooling layer with (2,2) size filter and (2,2) stride
- A 2D conv layer (3 x 3) with 16 filters and relu activation
- A maxpooling layer with (2,2) size filter and default/ (1,1), stride
- Flattening

- then a 64 neuron Dense layer, relue activation
- Dropout with rate 0.5
- Two outputs:
    - 1st is a dense layer (10 neurons and softmax activation)
    - 2nd is also 10-neuron dense layer and softmax activation.

Results:

```
ETA: 0sEvaluation on test set: {'loss': 0.097850237339735024, 'main_output_loss': 0.0863900191411376,
'auxiliary_output_acc': 0.96550000000000002, 'auxiliary_output_loss': 0.10931045722961426,
'main_output_acc': 0.97575000000000001}
```

Test Accuracy on First number = 0.97575
Test Accuracy on Second number = 0.9655

b. **For Mlp.py**

i. Edited Model Architecture as in mlp.py in 5.1 but with adam optimizer instead of 'sgd'.
Nb_epoch=30

Results

```
3584/4000 [=========================>....] - ETA: 0sEvaluation on test set: {'auxiliary_output_loss':
0.30344367212057116, 'main_output_acc': 0.93374999999999997, 'loss': 0.30356375133991242,
'main_output_loss': 0.30368382900953295, 'auxiliary_output_acc': 0.91600000000000004}
```

Test Accuracy on First number = 0.9334
Test Accuracy on Second Number = 0.916
Total Training time = 91s

ii. Edited Model Architecture as in mlp.py in 5.1 but with adam optimizer instead of 'sgd'. Also introduced additional 64-neuron dense layer with relu activation
Nb_epoch=30 **(saved as extra_model2.py)**

Functional Model is as follows.
- digit_input = Input(shape=(1, img_rows, img_cols))
- Flattening
- 64 neuron dense layer with relu activation
- 64 neuron dense layer with relu activation.
- Two outputs
    - 1st is a 10-neuron dense layer with softmax activation
    - 2nd is also 10-neuron dense layer with softmax activation.

Observation: Training Accuracy increased significantly (compared to without addition of extra layer) but increase in test accuracy was relatively small. Training time was also increased (97s vs 91s)

Results

```
3904/4000 [============================>.] - ETA: 0sEvaluation on test set: {'main_output_acc': 0.9395,
'auxiliary_output_acc': 0.91925000000000001, 'auxiliary_output_loss': 0.36523338156938551,
'main_output_loss': 0.34454929667711259, 'loss': 0.35489133787155153}
```

Test Accuracy on First number = 0.9395
Test Accuracy on Second number = 0.91925
Total Training time: 97s

iii. Edited Model Architecture as in mlp.py in 5.1 but with adam optimizer instead of 'sgd. Increased number of neurons of hidden dense layer from 64 to 128**. (saved as extra_model3.py)**

Functional Model is as follows.

- digit_input = Input(shape=(1, img_rows, img_cols))
- Flattening
- 128 neuron dense layer with relu activation
- Two outputs
    - 1st is a 10-neuron dense layer with softmax activation
    - 2nd is also 10-neuron dense layer with softmax activation.

Results

```
3648/4000 [==========================>...] - ETA: 0sEvaluation on test set: {'main_output_loss':
0.30189536148309709, 'loss': 0.3138675104379654, 'auxiliary_output_acc': 0.92974999999999997,
'main_output_acc': 0.94425000000000003, 'auxiliary_output_loss': 0.32583966147899629}
```

Test Accuracy on First number = 0.94425
Test Accuracy on second number = 0.92974
Total training time: 99s