Dustin Kut Moy Cheung
996881147

CSC 2515H
Assignment 2

1. $$p(x|\pi,\mu) = \sum_{k=1}^{K} \pi_k \, p(x|\mu_k)$$

$$p(x|\mu_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i}$$

$$L(X|\pi,\mu) = \log\left[\prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \prod_{i=1}^{D} \mu_{ki}^{x_i}\right]$$

$$\text{Log likelihood} = \sum_{n=1}^{N} \log\left[\sum_{k=1}^{K} \pi_k \prod_{i=1}^{D} \mu_{ki}^{x_i}\right]$$

$$\frac{dL}{d\pi_k} = \sum_{n=1}^{N} \frac{\prod_{i=1}^{D} \mu_{ki}^{x_i}}{\sum_{k=1}^{K} \pi_k \prod_{i=1}^{D} \mu_{ki}^{x_i}}$$

$\pi_1(\mu_{11}^{x_1} * \mu_{12}^{x_2} * \dots)$
$+ \pi_2$

$ax^c = ac\,x^{c-1}$

$$\frac{dL}{d\mu_{ki}} = \sum_{n=1}^{N} \frac{\left[\pi_k x_i \prod_{i=1}^{D} \mu_{ki}^{x_i}\right] / \left[\mu_{ki}\right]}{\sum_{k=1}^{K} \pi_k \prod_{i=1}^{D} \mu_{ki}^{x_i}}$$

1 Pre-Processing Inputs

The provided data can be processed by normalizing the data provided so that the dimensions of the data are all within the same scale.

[Batch-Size: 10, 100 units in hidden layer]

Epoch	% Classification error (no pre-processing)	% Classification error (with pre-processing)
1	91.6	90.6
2	85.0	82.7
3	82.5	77.5

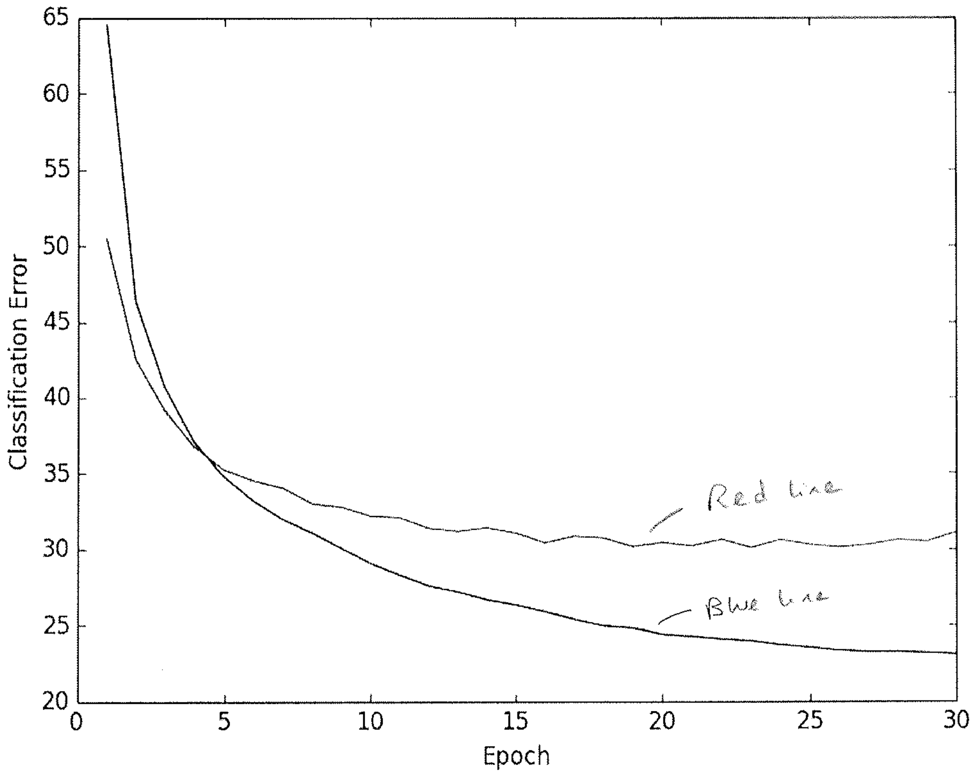
Playing with learning rate and momentum

[Batch-Size: 10, 100 units in hidden layer]

Learning Rate	Momentum	% Classification Error after 3 Epochs
0.3	0.9	45.1
0.3	0.2	40.7
0.03	0.2	70.5
0.03	0.9	77.5

After experimenting with the learning rate and momentum for our Neural Network, the learning rate of 0.3 and momentum of 0.2 is chosen for the rest of the questions.

2 Controlling Overfitting



[Blue line: training data, Red line: dev set data]
Strategy: Stop the training when the difference in classification error between 2 consecutive epochs of the dev set is negligible (~ 0.01).

```
best_dev_accuracy = 0
for i in range(arguments.max_epochs):
       nn_train.train_for_one_epoch(train_src, eps, momentum, l2, batch_size)
       accuracy, lg_p = nn_train.test(validation_src)
       # CODE FOR CONTROLLING OVERFITTING...GOES SOMEWHERE AROUND HERE..
       if best_dev_accuracy + 0.01 > accuracy:
       break

       best_dev_accuracy = accuracy
```

Classification Error of dev set: 31.4%

3 # Of Units

Units	% Classification error (train set)	% Classification error (dev set)	Epochs used
100	26.7	31.4	14
300	21.9	28.9	13
500	38.1	39.6	5

The development set classification error is lowest for 300 hidden units. The number of epochs used decreases as the number of units increases.

4 Assessing Impact of Depth

With 2 layers, 300 hidden units,

Epoch	% Classification Error
1	85.9
2	71.5
3	56.4
4	49.1
5	44.3

With 2 layers, the classification error is worse than with 1 layer, and the training takes significantly more time. This might be the case because of the huge entropy in a 2 layer system, which requires significantly more data points to decrease the entropy in the system.

5 Decoding Results

The models.mat was trained for 30 epochs with 1 layer, 100 hidden units, and a momentum of 0.2 and learning rate of 0.3.

31.58% PER

Code Snippets

train_nnet.py

```
def fwd_prop(self, data):
"""
NEED TO IMPLEMENT.
Return list of outputs per layer.

Do a pass over all layers, and return a list of activations for each
layer. You may want to call layer.fwd_prop for each layer
"""
lst_layer_outputs = []
# data is (345, 1)

previous_activation = data
lst_layer_outputs.append(previous_activation)
for layer in self._lst_layers:
      previous_activation = layer.fwd_prop(previous_activation)
      lst_layer_outputs.append(previous_activation)

return lst_layer_outputs


def back_prop(self, lst_layer_outputs, data, targets):
"""
NEED TO IMPLEMENT

Perform a backpropagation, return 'self' with updated gradient of
weights and biases for all layers. You may want to call layer.back_prop
for each layer.

lst_layer_outputs is from fwd_prop output
"""

input_grad = 0

for index, layer in enumerate(reversed(self._lst_layers), start=1):
      last_layer_output = lst_layer_outputs[-index]
      last_layer_input = lst_layer_outputs[-(index + 1)]
      # consider act_grad as dE_dxj
      if isinstance(layer, softmax_layer):    # only for last layer
      act_grad = layer.compute_act_gradients_from_targets(targets,
last_layer_output)
      else:   # for everything else
      act_grad = layer.compute_act_grad_from_output_grad(last_layer_output,
input_grad)
      input_grad = layer.back_prop(act_grad, last_layer_input)
```

```
def apply_gradients(self, eps, momentum, l2=0):
"""
NEED TO IMPLEMENT

Perform stochastic gradient descent step. You may want to call
layer.apply_gradients for each layer.
"""
for layer in self._lst_layers:
      layer.apply_gradients(momentum, eps, l2)
```

nnet_layers.py

```
def apply_gradients(self, momentum, eps, l2=.0001):
""" NEED TO IMPLEMENT

update wts_inc(b_inc) and use wts_inc(b_inc) to update the weight
(bias). You may want the gradient wts_grad(b_grad) as well as momentum
and learning rate.

TODO: apply linear regularizer later
"""
# ==== IMPLEMENTED ====================================================
self._b_inc = eps * (self._b_grad) - momentum * (self._b_inc)
self._wts_inc = eps * (self._wts_grad) - momentum * (self._wts_inc)
# now that we have found the wts_grad, let's update the bias and weight
# itself
self._b = self._b - self._b_inc
self._wts = self._wts - self._wts_inc
# ====================================================================

# ==
# act_grad   :: gradient wrt activation function of this layer
# input_grad :: gradients wrt the input of this layer
# ==
def back_prop(self, act_grad, data):
'''
NEED TO IMPLEMENT.
Feel free to add member variables.
back prop activation grad, and compute gradients.

Back propagate activation gradients and compute gradients for one layer.
The output is a struct consisting of 3 parts, wts_grad, b_grad,
input_grad

NEED TO FIND WTS_GRAD HERE and update the self object.

data is the layer input

input grad is dE_dyi
act_grad is dE_dxj
```

```
        '''
        batch_size = data.shape[1]

        dE_dxj = sum(act_grad, axis=1)
        dE_dxj.shape = (dE_dxj.shape[0], 1)
        dE_wij = data.dot(act_grad.T)

        self._b_grad = dE_dxj / batch_size

        self._wts_grad = dE_wij / batch_size

        input_grad = self._wts.dot(act_grad)

        return input_grad


# ======
# self.wts :: weights for each layer
# b          :: bias for each layer
# wts_grad :: gradient for weights you calculated from back_prop for each layer
# wts_inc  :: actual update you will do for wts in a SGD step for each layer
# b_grad   :: gradient for bias you calculated from back_prop for each layer
# b_inc      :: actual update you will do for b in a SGD for each layer
# ======
class sigmoid_layer(layer):
        pass

        def fwd_prop(self, data):
        """ NEED TO IMPLEMENT

        Perform a forward pass
        """
        # data is (345, 1)
        # _wts is (345, 300)
        # (300, 345) x (345, 1)
        z = self._wts.T.dot(data) + self._b

        # sigmoid :: use logistic regression
        sigmoid = 1 / (1 + exp(-z))
        # we want to return a column vector, not a row vector
        return sigmoid

        def compute_act_grad_from_output_grad(self, output, output_grad):
        """ NEED TO IMPLEMENT

        Compute the gradients wrt activations of sigmoid layer, the input are
        the current activations of this layer and the another gradients wrt outputs of
        the sigmoid.
        """
        yj = output
```

```
        dE_dyj = output_grad

        act_grad = yj * (1 - yj) * dE_dyj
        return act_grad


class softmax_layer(layer):
        pass

        def fwd_prop(self, data):
        """ NEED TO IMPLEMENT

        Perform a forward pass

        weight is (300, 44)
        data is (300, 1)
        """
        # z is (44, 5)
        z = self._wts.T.dot(data) + self._b
        top_part = exp(z)
        bottom_part = sum(top_part, axis=0)
        result = top_part / bottom_part
        return result

        def compute_act_gradients_from_targets(self, targets, output):
        """ NEED TO IMPLEMENT

        Compute the gradients wrt activations of the softmax layer, given the
        targets and the outputs of the softmax, the inputs are the current
        activations of this layer and the target.
        """
        act_grad = output * (1 - output) * (output - targets)
        return act_grad
```

\# data = 2200

2200
752
1568

3.1

k	% misclassified	
1	66.7	732
3	65.3	763
5	62.5	825

3.2

k	% misclassified	
1	69.4	673
3	67.8	709
5	63.9	795

3.3 The kNN algorithm is affected negatively by the addition of label noise (% misclassified increases). The label noise affects the boundary conditions, which reduces the accuracy of our training.

Choosing a high value of k seems to improve the accuracy. It is also noted that with a higher value of k, the kNN algorithm seems to be less affected by label noise.

3.4 kNN ⇒ ① Find distance between test point and all of training points

② Assign a weight to the classification of each training point based on the distance in ①

③ For each classification target, calculate its total weight from ②. The target selected will be the one with the highest weight.

① $d_i = \|x - r_i\|^2 \; \forall i$, x : data needing classification

r_i : training set ;

② W_d : weight of target d

I : # training set

$$W_d = \sum_{i=1}^{I} z\left[\frac{1}{d_i}\right]$$, t_i : target of training ;

where $z = \begin{cases} 1 & \text{if } t_i = d \\ 0 & \text{if } t_i \neq d \end{cases}$

(3) Classification $= \arg\max_{d} \{W_d\}$

Do well $\Rightarrow$ When the training set have roughly the same # of training points for each target

However it will perform poorly if the partitioning of the training point is severly skewed to one target (one target has many more training points than a neighbouring target)

$$J = \frac{1}{N} \sum_{i=1}^{N} \|x_i - \theta_i\|^2 \Rightarrow \text{Find } \theta_i \text{ to minimize this.}$$

4.1 Probabilistic Interpretation of PCA // Loss Function
Explain how maximum likelihood in this model corresponds to minimizing squared error.

① z: latent variable corresponding to the principal-component subspace

$$p(z) = N(z \mid 0, I) \leftarrow \text{Zero-mean unit-covariance Gaussian.}$$

$$p(x \mid z) = N(x \mid Wz + \mu, \sigma^2 I)$$

Observed variable $\Rightarrow x = Wz + \mu + \varepsilon$

Mean of $x \Rightarrow$ Linear function of z governed by $D \times M$ matrix W.

Assume: Latent variable has a Gaussian Distribution
Linear Relationship between latent and observed variables.

$$L(\theta; x) = \log(X \mid \theta) = -\frac{N}{2} \log |C| - \frac{1}{2} \sum_{n} (x_n - \mu) C^{-1} (x_n - \mu)^T$$

where $\theta = W, \mu, \sigma$
C = Covariance of (x)

We can view PPCA as an extreme of minimizing squared error when the covariance of x, C is an identity matrix, i.e when $\lim \sigma^2 \to 0$ or when there is no shared information between the dimensions.

$C = C^{-1} = I \Rightarrow$ All dimensions are statistically independent
Variance of data along each dimensions is equal to one.

4.2 The optimal weights of PCA corresponds to a projection of data onto a ~~linear~~ lower dimensional linear space, such that the variance of the projected data is maximized.

One can estimate the global optimum by finding the average and covariance matrix of the data set, and then finding the M eigenvectors of S corresponding to the M largest eigenvalues.

4.3 $\boxed{K=1}$

	Classification error(%)	
PCA - 5	70.9	601
PCA - 10	68.6	690
PCA - 20	66.7	732

4.4 ① PCA is helping by reducing the dimensions of our dataset. That results in a speedup in our kNN algorithm

② However we do not see any classification improvement when using PCA + kNN compared to kNN only. In fact, the error increases with PCA - 5 + kNN. This is because we lose too much information as we reduce the dimension of our dataset.

I would expect it to help when we have a lot of dimensions in our data, but only a few of them influences the classification. The application of PCA will reduce the noise in our dataset and help the classification in kNN.