In solving the questions in this assignment, I worked together with my classmate Addison Weatherhead 1005691128. I confirm that I have written the solutions/code/report in my own words.

A2

Part I

Lisa Yu
1005786366
Yu Lisa

Q1) want to show that in a fully connected neural network with linear activation functions, the number of layers has no impact on network.

It is sufficient to show that such a network composed of any number of layers can be expressed as a single layer network.

At each layer i, we can express the output vector $\vec{y}_i$ as a function of the previous layer's outputs $\vec{y}_{i-1}$ and a weight matrix W_i of dimension $|\vec{y}_i| \times |\vec{y}_{i-1}|$ and a bias vector $\vec{b}_i$.

More specifically, $\vec{y}_i = W_i \vec{y}_{i-1} + \vec{b}_i$

So we have $\vec{y}_1 = W_1 \vec{y}_0 + \vec{b}_1$ where $\vec{y}_0$ is the input vector

$\vec{y}_2 = W_2 \vec{y}_1 + \vec{b}_2$

$\vdots$

$\vec{y}_i = W_i \vec{y}_{i-1} + \vec{b}_i$

Suppose we have a network with n number of layers, $n \in \mathbb{Z} \geq 1$, and weight matrices $W_1, W_2, \ldots W_n$ at each layer, and $\vec{x}$ as input,

we can write

$$\vec{y}_n = W_n \vec{y}_{n-1} + \vec{b}_n$$

$$= W_n (W_{n-1} \vec{y}_{n-2} + \vec{b}_{n-1}) + \vec{b}_n$$

$$= W_n W_{n-1} \vec{y}_{n-2} + W_n \vec{b}_{n-1} + \vec{b}_n \quad \text{by the distributive property of matrix multiplication}$$

$$= W_n W_{n-1} (W_{n-2} \vec{y}_{n-3} + \vec{b}_{n-2}) + W_n \vec{b}_{n-1} + \vec{b}_n$$

$$= W_n W_{n-1} W_{n-2} \vec{y}_{n-3} + W_n W_{n-1} \vec{b}_{n-2} + W_n \vec{b}_{n-1} + \vec{b}_n$$

$\vdots$

$$= \underbrace{W_n W_{n-1} \cdots W_1}_{W'} \vec{x} + \underbrace{W_n W_{n-1} \cdots W_2 \vec{b}_1 + W_n W_{n-1} \cdots W_3 \vec{b}_2 + \ldots + W_n \vec{b}_{n-1} + \vec{b}_n}_{\vec{b}'}$$

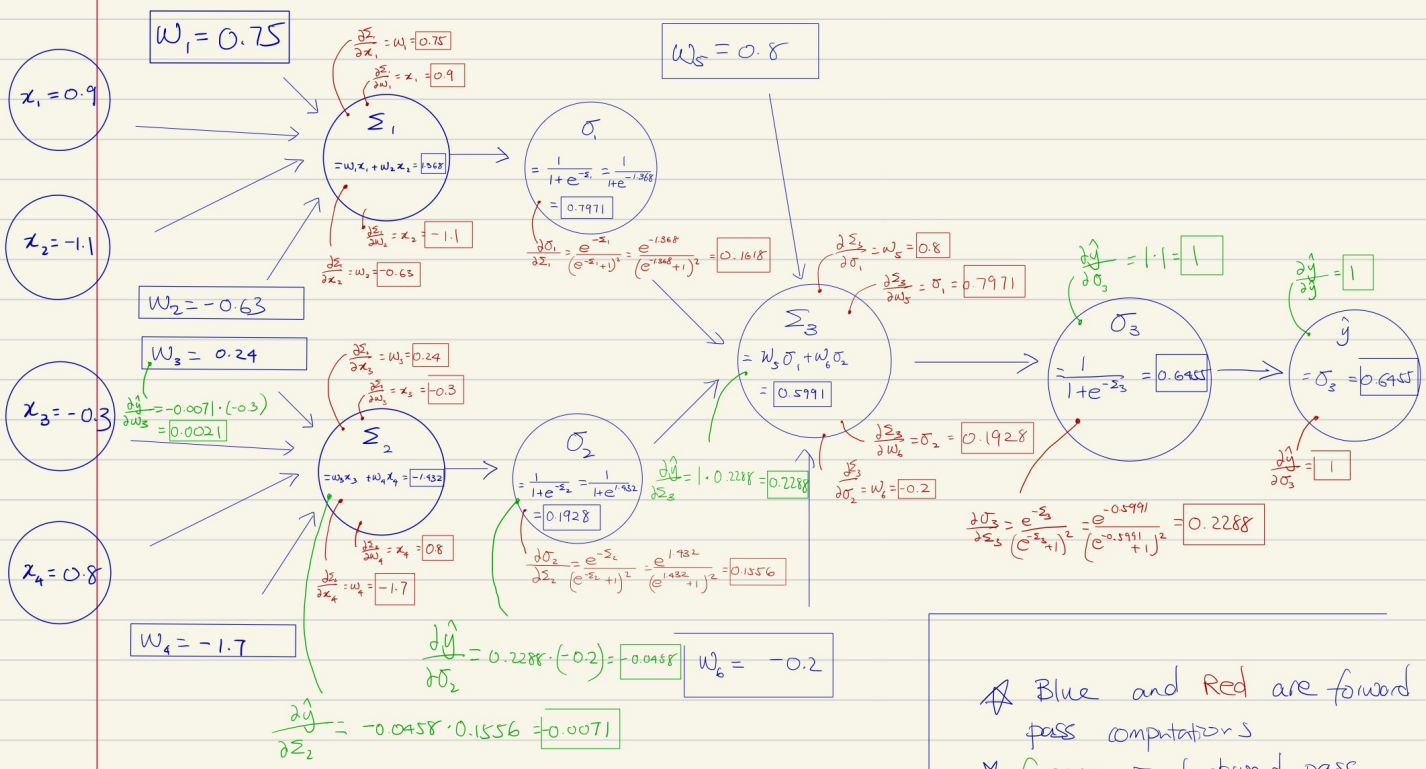
if we let $W_n W_{n-1} \cdots W_1 = W'$

and let $W_n W_{n-1} \cdots W_2 \vec{b}_1 + W_n W_{n-1} \cdots W_3 \vec{b}_2 + \ldots + W_n \vec{b}_{n-1} + \vec{b}_n = \vec{b}'$

$$= W' \vec{x} + \vec{b}'$$

meaning that we can express the output $\vec{y}_n$ as one-layer neural network with some new weight matrix W' and bias vector $\vec{b}'$, as wanted ∎.

(Q2)



* Blue and Red are forward pass computations
* Green is backward pass computations

$$L(y, \hat{y}) = \|y - \hat{y}\|_2^2 = (y - \hat{y})^2 = y^2 - 2y\hat{y} + \hat{y}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_3}$$

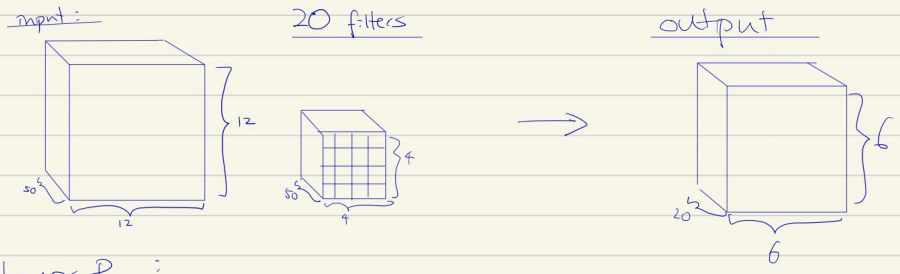
$$= -2y + 2\hat{y} \cdot \frac{\partial \hat{y}}{\partial w_3} = \left(-2y + 2\hat{y}\right)(0.0021)$$

$$= \left[-2(0.5) + 2(0.6455)\right](0.0021)$$

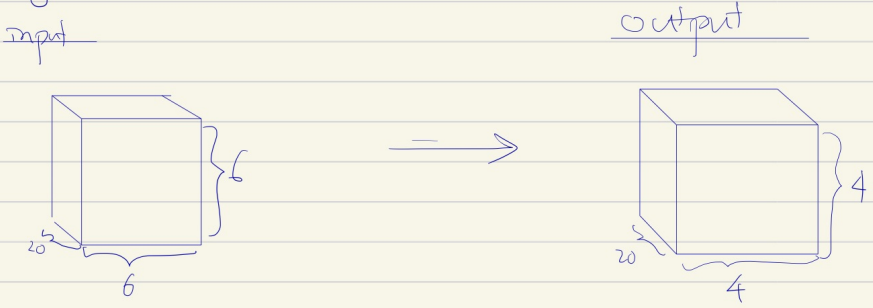
$$= 0.0006$$

(Q3)

Layer C:



Layer P:



No bias:

Layer C: each layer C's output pixel is the result of a convolution.
→ there are 20×6×6 output pixels
each convolution is the result of the summation of products between filter pixels and the input
→ there are 50×4×4 pixels per filter, each convolution requires 50×4×4−1 summations and 50×4×4 multiplications

⇒ total FLOPs for layer C = $20\times6\times6\times(50\times4\times4-1+50\times4\times4) = 1,151,280$

Layer P: each layer P's output pixel is the result of a max pooling operation.
$\max(x_1, x_2, \ldots x_9)$
→ there are 20×4×4 output pixels and each $\max(x_1,\ldots x_9)$ operation accounts for 8 FLOPs

⇒ total FLOPs for layer P = $20\times4\times4\times8 = 2560$

⇒ Total FLOPs without bias = $1,151,280 + 2,560 = \boxed{1,153,840 \text{ FLOPs}}$

With bias:

one additional FLOP is added to each convolution operation. There are 20×6×6 convolutions. Nothing else changes.

$20\times6\times6 = 720$

⇒ Total FLOPs with bias = $1,153,840 + 720 = \boxed{1,154,560 \text{ FLOPs}}$

Q4)

Layer C1: 6 filters of size 5x5x1
(assuming stride of 1)
6 bias, one for each filter
total trainable parameters: $6 \times 6 \times 5 \times 1 + 6 = \boxed{156}$

Layer S2: assuming fixed subsampling (according to Piazza instructor instructions)
total trainable parameters: $\boxed{0}$

Layer C3: 16 filters of size 5x5x6
(assume stride = 1)
16 bias, one for each filter
total trainable parameters: $16 \times 5 \times 5 \times 6 + 16 = \boxed{2416}$

Layer S4: assuming fixed subsampling (according to Piazza instructor instructions)
total trainable parameters: $\boxed{0}$

Layer C5: 120 filters (or can be thought of as sets of weights)
each of size 5x5x16
120 bias, one for each filter
total trainable parameters: $120 \times 5 \times 5 \times 16 \times 120 = \boxed{48120}$

Layer F6: 84 sets of weights, each weight vector of dimension 120 x 1
84 bias
Total trainable parameters: $84 \times 120 + 84 = \boxed{10,164}$

Layer Output: 10 sets of weights, each weight vector of dim 84 x 1
10 bias
total trainable parameters = $10 \times 84 + 10 = \boxed{850}$

Total trainable parameters for network = $156 + 2416 + 48120 + 10164 + 850$

$= \boxed{61,706}$

Q5) Given any neural network, and any neuron i in the neural network, we have the output of the neuron being

$y_i = f(\vec{w}_i, \vec{x}_i, b_i) = \frac{1}{1+e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)}}$, where $\vec{x}_i$ is the vector representation of the neuron's inputs. $\vec{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ \vdots \\ x_{ik} \end{bmatrix}$

$\vec{w}_i$ is the vector representation of the neuron's weights $\vec{w}_i = \begin{bmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{ik} \end{bmatrix}$

and b_i is a scalar bias.

for all $k \in \{1, 2, \dots k\}$, $\frac{\partial y_i}{\partial w_{ik}} = \frac{(x_{ik})e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)}}{(e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)} + 1)^2} = (x_{ik})(y_i)^2\left(\frac{1}{y_i} - 1\right)$ since $y_i = \frac{1}{1+e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)}} \Rightarrow e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)} = \frac{1}{y_i} - 1$, and we can do this since $y_i \neq 0$.

$= (x_{ik})(y_i - y_i^2)$

and since x_{ik} is the output of some other neuron j (a parent of i in the graph representation of the network)
i.e. $x_{ik} = y_j = f(\vec{w}_j, \vec{x}_j, b_j)$ for some $\vec{w}_j, \vec{x}_j, b_j$.

$= (y_j)(y_i - y_i^2)$

$\frac{\partial y_i}{\partial x_{ik}} = \frac{w_{ik} e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)}}{(e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)} + 1)^2} = (w_{ik})(y_i - y_i^2)$

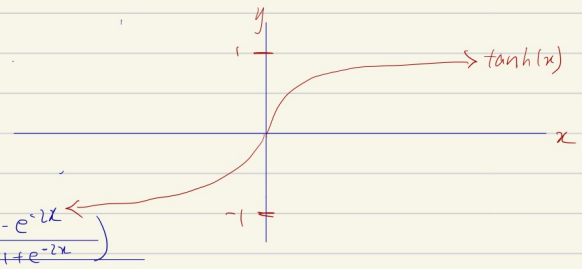
$\frac{\partial y_i}{\partial b_i} = \frac{e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)}}{(e^{-(\vec{w}_i \cdot \vec{x}_i + b_i)} + 1)^2} = y_i - y_i^2$

we note that non of these requires input $\vec{x}_i$.

When we are calculating the gradient of this neural network with respect to the individual w_{mn}'s (with weight in the mth row in the nth layer of the network), we can use back propagation and multiply the individual $\frac{\partial y_i}{\partial x_{ik}}$ along the graph going backwards until we reach the neuron m. Then we can multiply by $\frac{\partial y_m}{\partial w_{mn}}$ to get the partial of the model output w.r.t. w_{mn}. We can do the same for partial w.r.t a bias term b_m by multiplying by $\frac{\partial y_m}{\partial b_m}$ at the last step. And none of the products of the back propagation would require any inputs to the neurons as we wanted to show.

Q6)

a) the output range for this function is between -1 and 1 exclusive.



b) $\frac{d\tanh(x)}{dx} = \frac{d\left(\frac{1-e^{-2x}}{1+e^{-2x}}\right)}{dx}$

$$= \frac{2e^{-2x}}{e^{-2x}+1} + \frac{2(1-e^{-2x})e^{-2x}}{(e^{-2x}+1)^2}$$

$$= \frac{4e^{2x}}{(e^{2x}+1)^2}$$

$$= 4\left[\frac{e^{2x}}{(e^{2x}+1)^2}\right]$$

let σ be the sigmoid function $\sigma(x) = \frac{1}{e^{-x}+1}$

$\Rightarrow \sigma(2x) = \frac{1}{e^{-2x}+1}$

$\Rightarrow \frac{d\sigma(2x)}{dx} = \frac{2e^{-2x}}{(e^{-2x}+1)^2} = \frac{2e^{2x}}{(e^{2x}+1)^2}$

$$= 2\frac{d\sigma(2x)}{dx}$$

OR

$$= 4\left(\sigma(2x) - \sigma^2(2x)\right)$$

c) The sigmoid activation function σ has range $(0,1)$ and the tanh activation function has range $(-1,1)$. So we would want to use sigmoid when we want to output a number that represents a probability.

However, we see that the tanh function has a stronger gradient. I.e. for all $x \in \mathbb{R}$, $\tanh(x) > \sigma(x)$ proof $\frac{d\tanh(x)}{dx} = 2\frac{d\sigma(2x)}{dx} > \frac{d\sigma(2x)}{dx}$ since $\sigma > 0 \ \forall x$.

$> \frac{d\sigma(x)}{dx}$ since $\sigma(2x)$ = squeezing the function $\sigma(x)$ horizontally and $\frac{d\sigma(x)}{dx} > 0$.

So we would want to use $\tanh(x)$ as the activation function when we want faster convergence (speed up the learning process).

Part I

Task 1

* note that we have deleted the two corrupted files before we loaded in the data.

```
[88] DATA_PATH = "/content/drive/MyDrive/Colab Notebooks/csc420/assignments/A2/notMNIST_small.tar/notMNIST_small"

     SPLIT_DATA_PATH = "/content/drive/MyDrive/Colab Notebooks/csc420/assignments/A2/notMNIST_small.tar/notMNIST_small_split"

     splitfolders.ratio(DATA_PATH, output=SPLIT_DATA_PATH, seed=1337, ratio=(15000/18720, 1000/18720, (18720-15000-1000)/18720))

     Copying files: 18724 files [05:21, 58.31 files/s]
```

```
[89] transform = transforms.Compose([
                                transforms.Grayscale(num_output_channels=1),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                               ])
     TRAIN_DATA_PATH = "/content/drive/MyDrive/Colab Notebooks/csc420/assignments/A2/notMNIST_small.tar/notMNIST_small_split/train"
     VAL_DATA_PATH = "/content/drive/MyDrive/Colab Notebooks/csc420/assignments/A2/notMNIST_small.tar/notMNIST_small_split/val"
     TEST_DATA_PATH = "/content/drive/MyDrive/Colab Notebooks/csc420/assignments/A2/notMNIST_small.tar/notMNIST_small_split/test"

     trainset = datasets.ImageFolder(root=TRAIN_DATA_PATH, transform=transform)
     trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

     valset = datasets.ImageFolder(root=VAL_DATA_PATH, transform=transform)
     validloader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True)

     testset = datasets.ImageFolder(root=TEST_DATA_PATH, transform=transform)
     testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)
```

* note that I have used the external library split_folders to split the data proportionally.

processing pipeline:

→ first, I turned the images to grayscale images since input images are grayscale. (otherwise would have 3 channels of repeated values).

→ I also normalized the data after transforming them into tensors.
this ensures that the pixels would have similar data distribution and speed up convergence when training

→ I did not rescale the images since they are all the same input size.

→ I did not augment the images since there are approx. the same number of dataset images for each class (the letters A, B, C ... J.). No inherent bias, good representation.

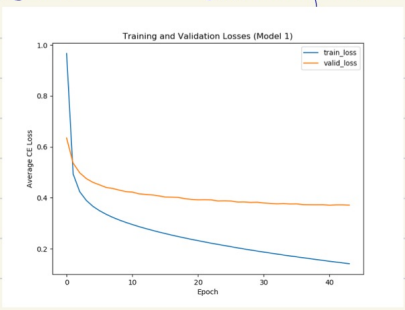
Task 2

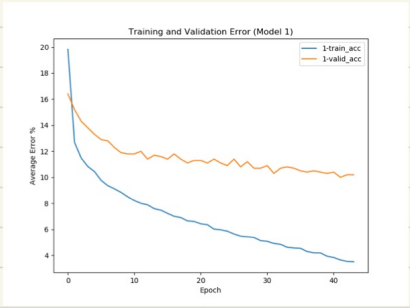
Note that I did not implement a softmax layer at the last layer of my model because I am using the CrossEntropyLoss() from PyTorch for my loss function, and the implementation of this function already accounted for the softmax function (equivalent to applying PyTorch's NLLLoss then LogSoftMax). Pytorch probably did this for numeric stability. But since the question asked to implement with softmax (instead of log softmax), I decided to just use PyTorch's CrossEntropyLossfunction for whenever I evaluate loss, and when I need to get the predictions of the model (for accuracy or error purposes), I just passed the output through another softmax function before taking the maximum prediction. This is equivalent to adding a layer of softmax to my model and using the actural cross entropy lass.

$$L\left(\frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}, t\right) = -\sum_{k=1}^{K} t_k \log \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} = \text{CrossEntropyLoss as defined by PyTorch}$$

* all models used learning rate decay schedule (optim.lr_scheduler.StepLR(SGD, step_size=5, gamma=0.1)
and early stopping with max number of non-increasing epoch validation loss = 3

Model 1 : Learning rate = 0.01, batch size = 64

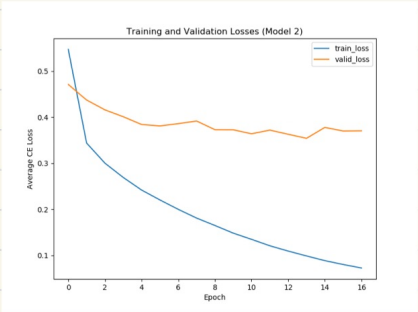


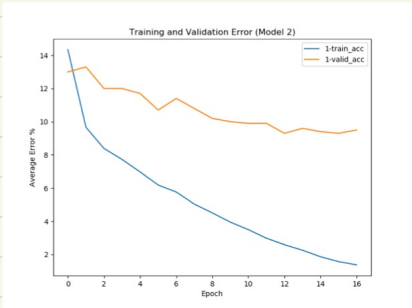


val accuracy = 89.80%
val loss = 0.37
val error = 10.20%

Best Model

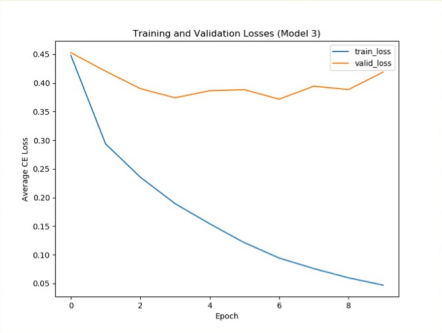
Model 2 : Learning rate = 0.05 batch size = 64

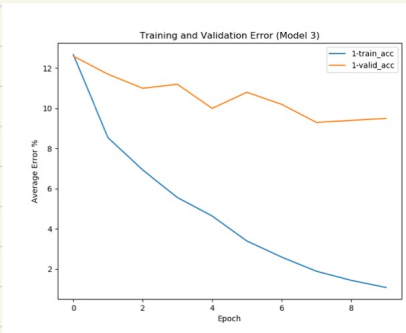




val accuracy = 90.50%
val loss = 0.35
val error = 9.50%

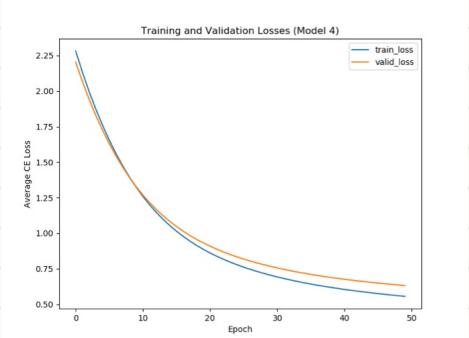
Model 3 : Learning rate = 0.07 batch size = 32

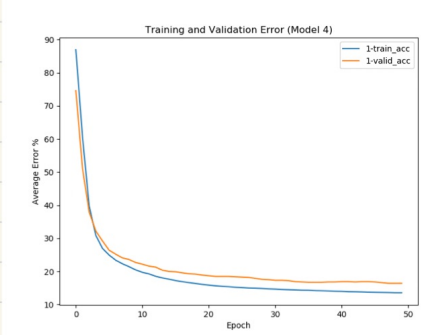




val accuracy = 90.50%
val loss = 0.37
Val error = 9.50%

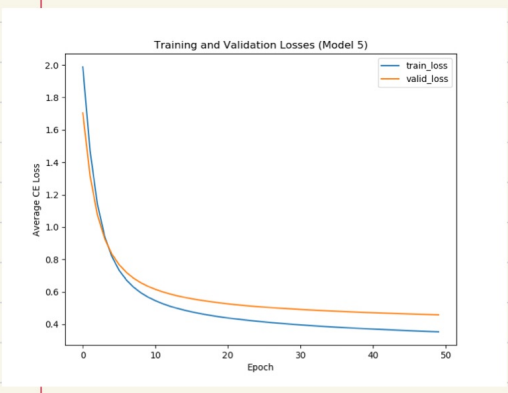
Model 4 : learning rate = 0.0001 batch size = 32

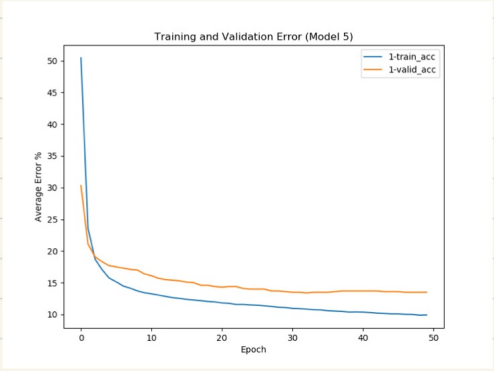




val accuracy = 83.60%
val loss = 0.63
Val error = 16.40%

Model 5 learning rate = 0.001 batch size = 64



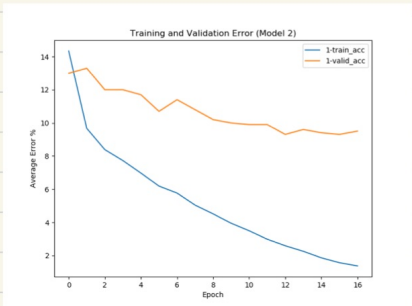


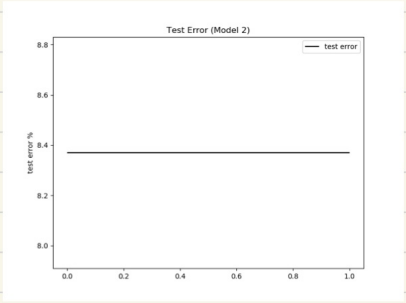
val accuracy = 86.50%
val loss = 0.46
val error = 13.50%

Best Model:

Model 2: learning rate = 0.05 batch size = 64

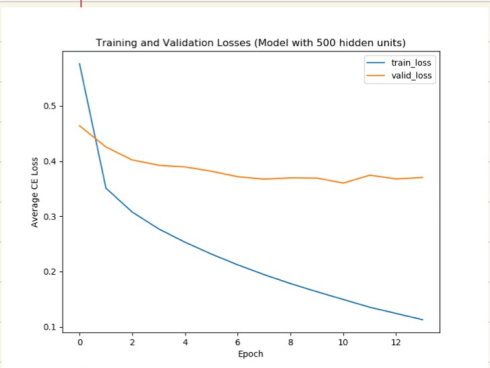
training error = 2%
val error = 9.50%
test error = 8.37 %

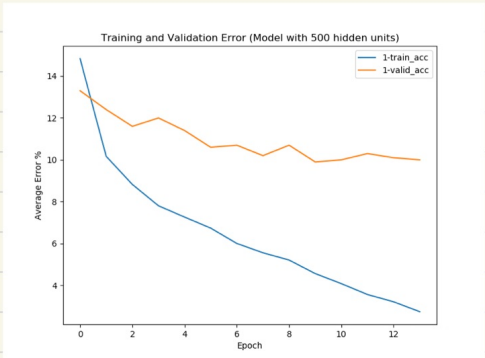




Task 3

All models used learning rate = 0.01, batch size = 64 (optimal parameters from task 2)
500 hidden units

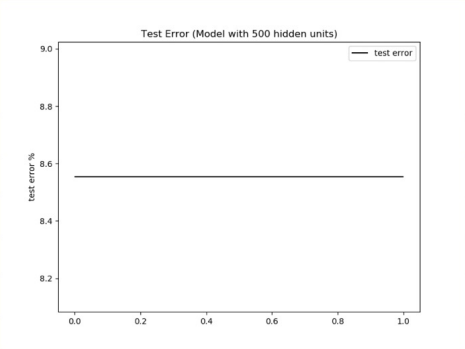




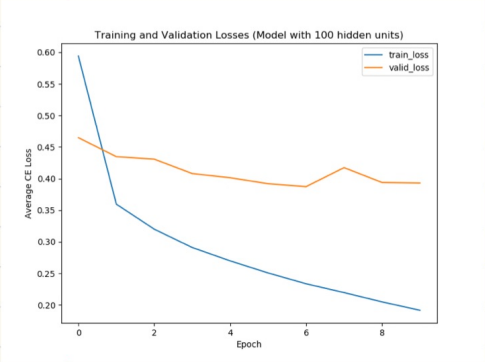
val accuracy = 90.00%
val loss = 0.36
val error = 10.00%

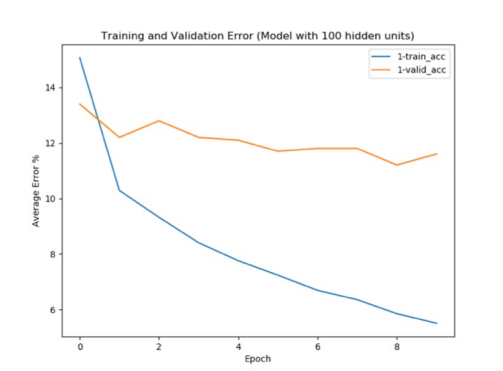
training error = 4.09%

test error = 8.55%

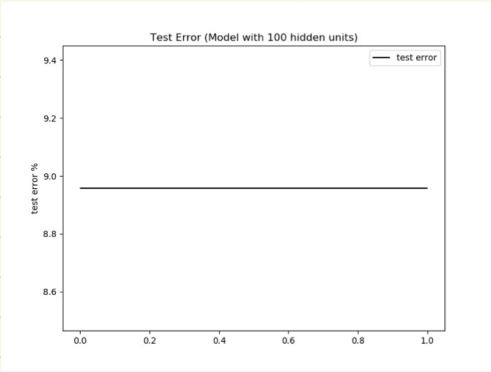


100 hidden units





val accuracy = 88.40%
val loss = 0.39
Val error = 11.60%
training error = 6.69%
test error = 8.96%

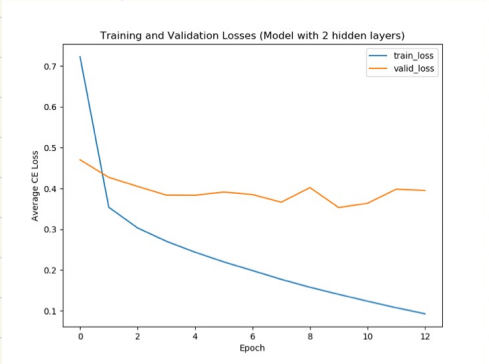


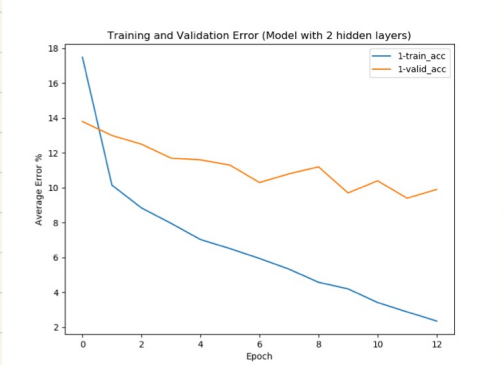
1000 hidden units (already trained and tested. see task 2)

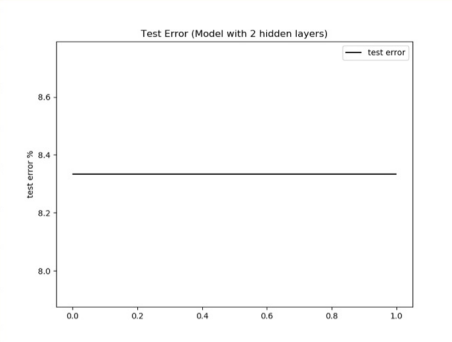
It appears that the errors (especially test) do not differ by much, so the # of hidden units has little impact on the performance of the model; the 1000 hidden unit still performed better by a little bit.

Task 4

Again, all models used learning rate = 0.01, batch size = 64 (optimal parameters from task 2).





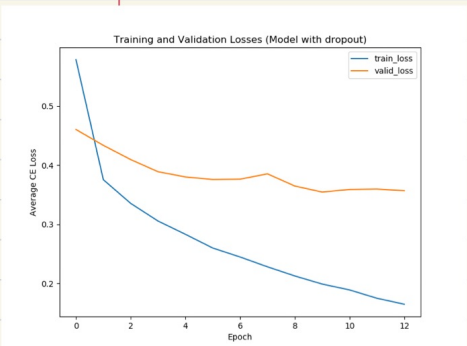


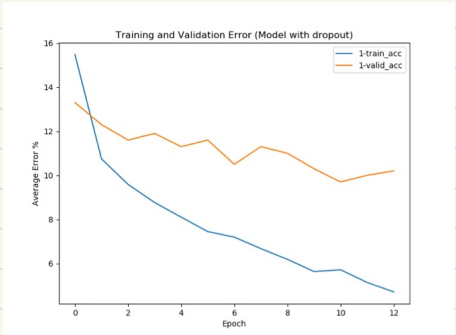
val accuracy = 90.10%
val loss = 0.35
Val error = 9.90%
training error = 4.19%
test error = 8.33%

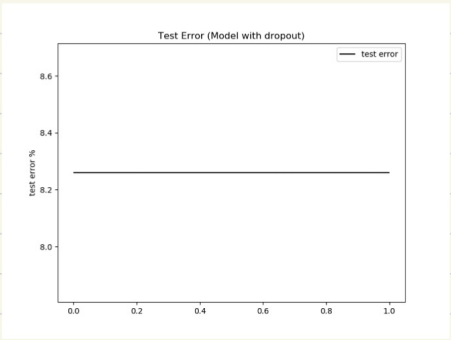
Again, it seems that there is little difference from the previous tasks ones, so number of layers also has small impact on the model performance.

Task 5

Again, all models used learning rate = 0.01, batch size = 64 (optimal parameters from task 2).







val accuracy = 89.80%
val loss = 0.35
Val error = 10.20%
training error = 5.63%
test error = 8.26%

The validation error and training error here is a little higher than without drop out, but test error is a little lower. Dropout ensures the model not overfitting and generalize better. However, the overall performance is still not too different.