Q1.1

Softmax(xi+c) = $e^{xi+c} / \Sigma_j e^{xi+c}$ = e^{xi} e^{c} / Σ_j e^{xi} e^{c} = e^{xi} e^{c} / e^{c} Σ_j e^{xi} = e^{xi}/ Σ_j e^{xi}

=Softmax(xi)

So softmax is invariant to translation.

When c=0, the value range is 0 to infinity, when c=-max xi, the value range is (0,1].

So we always set c=-max xi to simplify calculation and avoid overflow.

Q1.2

- The range of each element is (0, 1], the sum over all elements is 1.
- And turns it into a probability distribution between 0 and 1.
- The first step is to get the exponential result of xi, so they can have the same base.

  The second step is to get the sum of all si.

  The third step is to compute the probability of each si so that they can be distributed corresponding to their magnitude between 0 and 1.

Q1.3

If the multi-layer neural network has no activation function, the output of each neuron would be:

$y_i = b + \Sigma w x_i$

This is a linear function and the final output of the whole network would also be computed by all these linear functions.

Thus, this is equivalent to linear regression.

Q1.4

Gradient of $\sigma(x) = e^{-x}(1+e^{-x})^{-2} = e^{-x}/(1+e^{-x}) * 1/(1+e^{-x})$

$= (1-1/(1+e^{-x})) * 1/(1+e^{-x}) = (1-\sigma(x)) * \sigma(x)$

Q1.5

$y = W x + b$

So $\partial y/\partial W = x$, $\partial y/\partial x = W$, $\partial y/\partial b = 1$

Then $\partial J/\partial W = \Sigma\, \partial J/\partial y * \partial y/\partial W = x\delta^T \in R^{dxk}$

$\partial J/\partial x = \Sigma\, \partial J/\partial y * \partial y/\partial x = W\delta \in R^{dx1}$

$\partial J/\partial b = \Sigma\, \partial J/\partial y * \partial y/\partial b = \delta \in R^{kx1}$

Q1.6

1. The final $dy/dx = \sigma'(h(x))h'(x)$

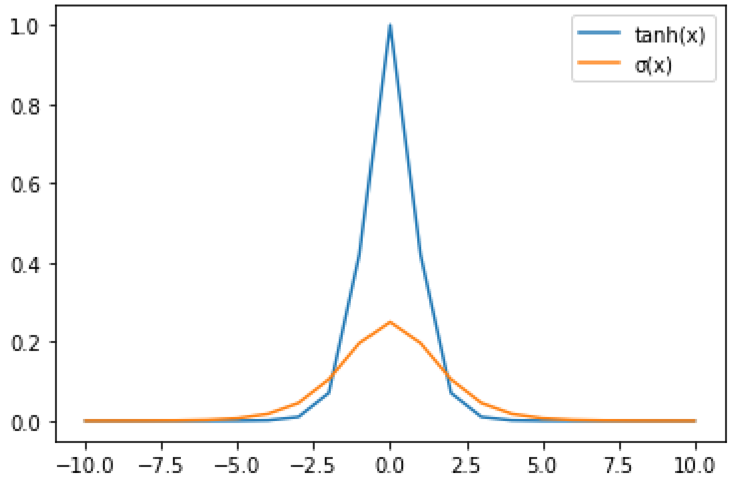
   And $\sigma'(x) = (1-\sigma(x)) * \sigma(x)$, the final gradient will be close to zero if we multiply $\sigma'(x)$ which is between 0 and 1 for many times.

   This phenomenon is called vanishing gradients.

2. The output range of tanh(x) is (-1,1), output range of sigmoid(x) is (0,1). So tanh(x) has symmetric output and this can make the average value around zero, which will increase the speed to converge.

3. As we can see, $\tanh'(x) = 1 - \tanh(x)^2$ and $\sigma'(x) = (1-\sigma(x)) * \sigma(x)$

   For the gradient value around 0, tanh'(x) is larger which means it will decrease more slowly. So tanh(x) has less of a vanishing gradient problem.



4. $\sigma(x) = 1/(1+e^{-x})$

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

   $\tanh(x) = 2\sigma(2x)-1$

Q2.1.1

If we initialize the network with all zeros, there would be no difference for output between different input. Thus, the output would be the same after forward propagation and the errors would be the same in backpropagation. This means the same output would produce the same gradient and the corresponding weights would be updated in the same way. However, we want to update weights differently.

Q2.1.2

```
def initialize_weights(in_size,out_size,params,name=''):
    W, b = None, None
    upper = (6/(in_size+out_size))**0.5
    lower = -upper
    W = np.random.uniform(lower, upper, (in_size, out_size))
    b = np.zeros(out_size)

    params['W' + name] = W
    params['b' + name] = b
```

Q2.1.3

Initializing with random numbers can improve the variance of parameters and increase the process to find optimal parameters with less loss value. Thus it can make convergence faster.

By scaling the initialization depending on layer size, the variance of weights gradient will become the same through every layer. And this can make gradient change within a limited range during training.

Q2.2.1

```
def sigmoid(x):
    res = None
    res = 1/(1+np.exp(-x))

    return res

############################## Q 2.2.1 ##############################
def forward(X,params,name='',activation=sigmoid):
    """
    Do a forward pass

    Keyword arguments:
    X -- input vector [Examples x D]
    params -- a dictionary containing parameters
    name -- name of the layer
    activation -- the activation function (default is sigmoid)
    """
    pre_act, post_act = None, None
    # get the layer parameters
    W = params['W' + name]
    b = params['b' + name]

    pre_act = W @ X+b
    post_act = activation(pre_act)

    # store the pre-activation and post-activation values
    # these will be important in backprop
    params['cache_' + name] = (X, pre_act, post_act)

    return post_act
```

Q2.2.2

```
def softmax(x):
    res = None

    c = -np.max(x, axis=1)
    c.reshape(-1, 1)
    tmp = np.exp(x+c)
    res = tmp / np.sum(tmp, axis=1).reshape(-1, 1)

    return res
```

Q2.2.3

```
def compute_loss_and_acc(y, probs):
    loss, acc = None, None

    loss = -np.sum(y * np.log(probs))
    y_index = np.argmax(y, axis=1)      # get the class as label=1
    probs_index = np.argmax(probs, axis=1)      # get the most
possible class
    acc = np.sum(np.equal(y_index, probs_index)) / y.shape[0]

    return loss, acc
```

Q2.3.1

```python
def backwards(delta,params,name='',activation_deriv=sigmoid_deriv):
    """
    Do a backwards pass

    Keyword arguments:
    delta -- errors to backprop
    params -- a dictionary containing parameters
    name -- name of the layer
    activation_deriv -- the derivative of the activation_func
    """
    grad_X, grad_W, grad_b = None, None, None
    # everything you may need for this layer
    W = params['W' + name]
    b = params['b' + name]
    X, pre_act, post_act = params['cache_' + name]

    # do the derivative through activation first
    # (don't forget activation_deriv is a function of post_act)
    # then compute the derivative W, b, and X
    delta = delta * activation_deriv(post_act)
    grad_W = X.T @ delta
    grad_X = delta @ W.T
    grad_b = np.sum(delta, axis=0)

    # store the gradients
    params['grad_W' + name] = grad_W
    params['grad_b' + name] = grad_b
    return grad_X
```

Q2.4

```
def get_random_batches(x, y, batch_size):
    batches = []
    indexes = range(x.shape[0])
    while len(indexes) > 0:
        rand_i = np.random.choice(len(indexes), batch_size)
        select = [indexes[i] for i in rand_i]
        batch_x = [x[i] for i in select]
        batch_y = [y[i] for i in select]
        batches.append((np.array(batch_x), np.array(batch_y)))
        indexes = list(set(indexes) - set(select))
    return batches
```

```
max_iters = 500
learning_rate = 1e-3
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        # forward
        h1 = forward(xb, params, 'layer1')
        probs = forward(h1, params, name='output', activation=sigmoid)
        # loss
        # be sure to add loss and accuracy to epoch totals
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc
        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, name='output',
activation_deriv=linear_deriv)
        backwards(delta2, params, name='layer1',
activation_deriv=sigmoid_deriv)
        # apply gradient
        # gradients should be summed over batch samples
        params['Wlayer1'] -= learning_rate * params['grad_W'+'layer1']
        params['blayer1'] -= learning_rate * params['grad_b'+'layer1']
        params['Woutput'] -= learning_rate * params['grad_W'+'output']
        params['boutput'] -= learning_rate * params['grad_b'+'output']
    avg_acc /= batch_num
    if itr % 100 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc :
{:.2f}".format(itr,total_loss,avg_acc))
```

Q2.5

```python
# Q 2.5 should be implemented in this file
# you can do this before or after training the network.

# compute gradients using forward and backward
h1 = forward(x,params,'layer1')
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(y, probs)
delta1 = probs - y
delta2 = backwards(delta1,params,'output',linear_deriv)
backwards(delta2,params,'layer1',sigmoid_deriv)

# save the old params
import copy
params_orig = copy.deepcopy(params)

# compute gradients using finite difference
eps = 1e-6
for k,v in params.items():
    if '_' in k:
        continue
    # we have a real parameter!
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   subtract 2*epsilon
    #   run the network
    #   get the loss
    #   restore the original parameter value
    #   compute derivative with central diffs
    cur_grad = params['grad_' + k]
    if len(v.shape) > 1:
        for i in range(v.shape[0]):
            for j in range(v.shape[1]):
                v_cur = v[i, j]
                v[i, j] = v_cur - eps
                h1 = forward(x, params, 'layer1')
                probs = forward(h1, params, 'output', softmax)
                loss1, acc1 = compute_loss_and_acc(y, probs)

                v_cur = v[i, j]
                v[i, j] = v_cur + eps
                h1 = forward(x, params, 'layer1')
```

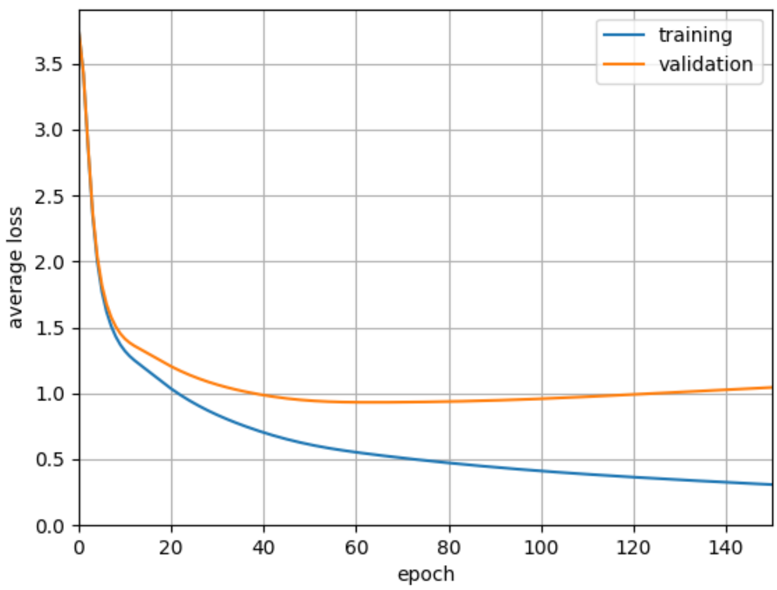
```
            probs = forward(h1, params, 'output', softmax)
            loss2, acc2 = compute_loss_and_acc(y, probs)

            cur_grad[i, j] = (loss2 - loss1) / (2 * eps)

    else:
        for i in range(v.shape[0]):
            v_cur = v[i]
            v[i] = v_cur - eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss1, acc1 = compute_loss_and_acc(y, probs)

            v_cur = v[i]
            v[i] = v_cur + eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss2, acc2 = compute_loss_and_acc(y, probs)

            cur_grad[i] = (loss2 - loss1) / (2 * eps)


total_error = 0
for k in params.keys():
    if 'grad_' in k:
        # relative error
        err = np.abs(params[k] -
params_orig[k])/np.maximum(np.abs(params[k]),np.abs(params_orig[k]))
        err = err.sum()
        print('{} {:.2e}'.format(k, err))
        total_error += err
# should be less than 1e-4
print('total {:.2e}'.format(total_error))
```

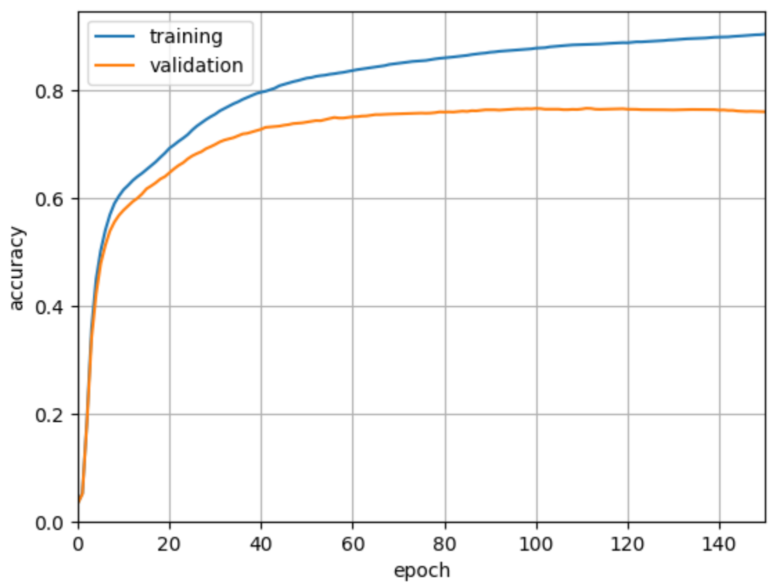
Q3.1

```
max_iters = 150
# pick a batch size, learning rate
batch_size = 15
learning_rate = 0.003
hidden_size = 64
```

Validation accuracy: 0.76

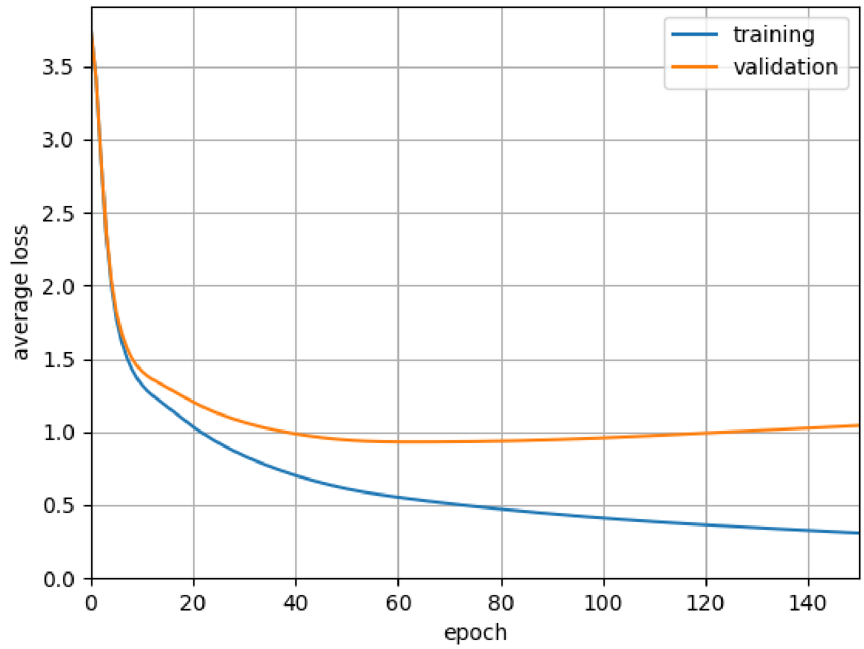
Test accuracy: 0.7772222222222223

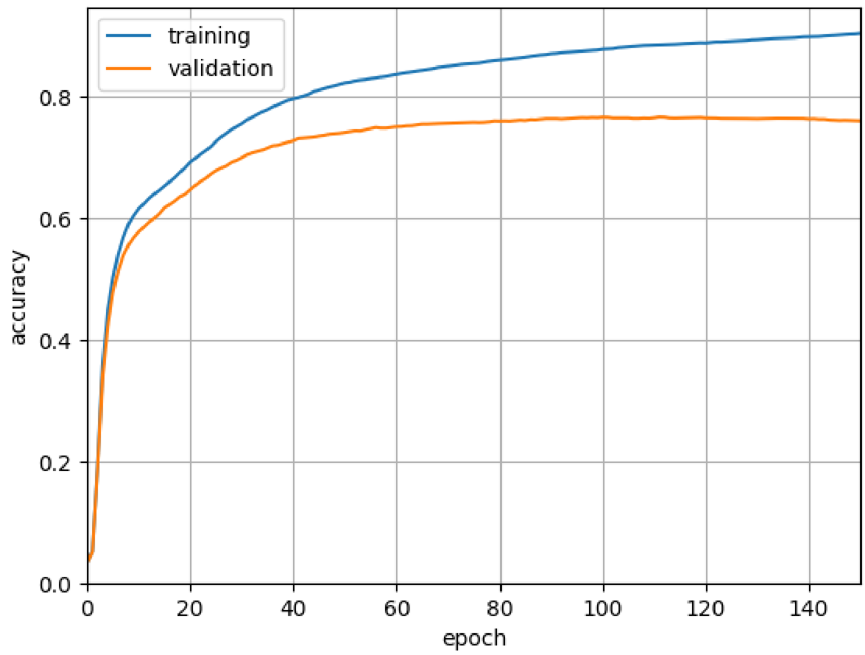




Q3.2

Learning rate with tuned learning rate=0.003:

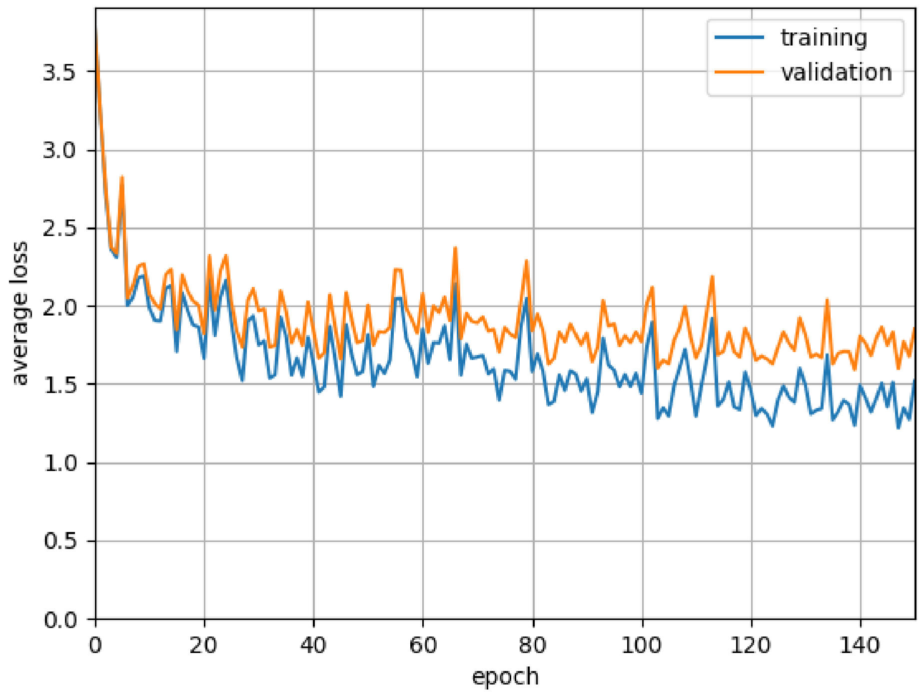


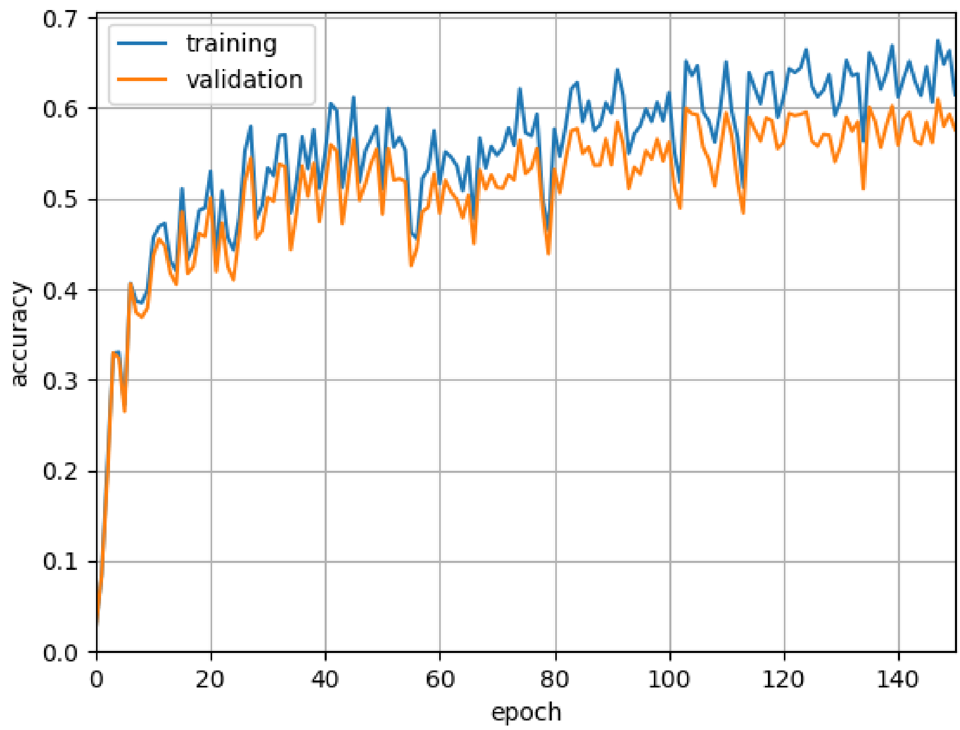


Validation accuracy: 0.76

Test accuracy: 0.7772222222222223

Learning rate with 10 times that learning rate=0.03:

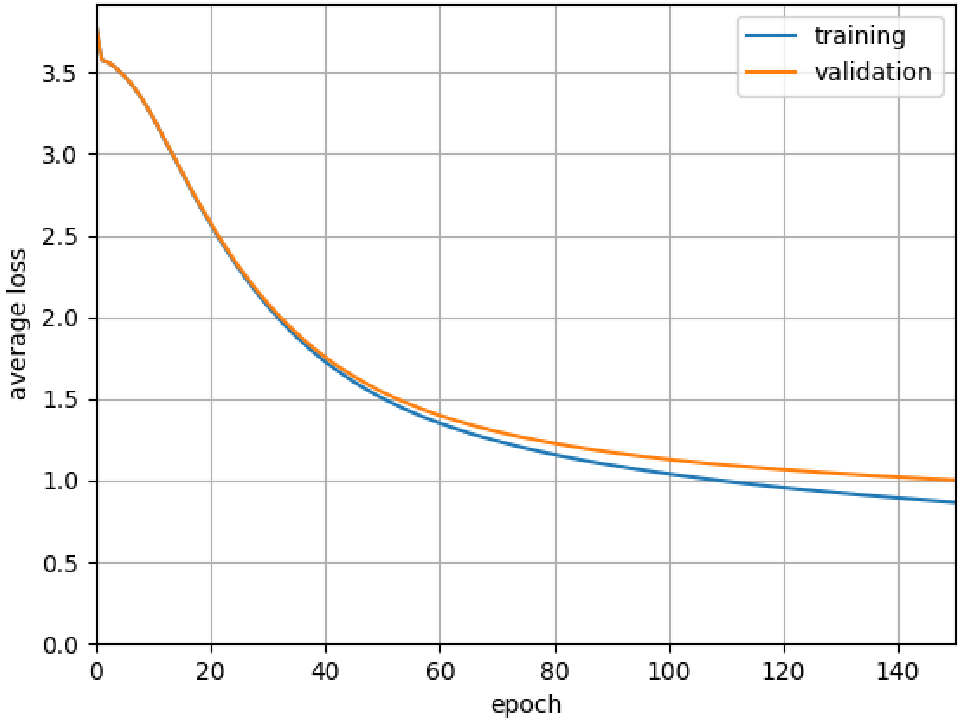


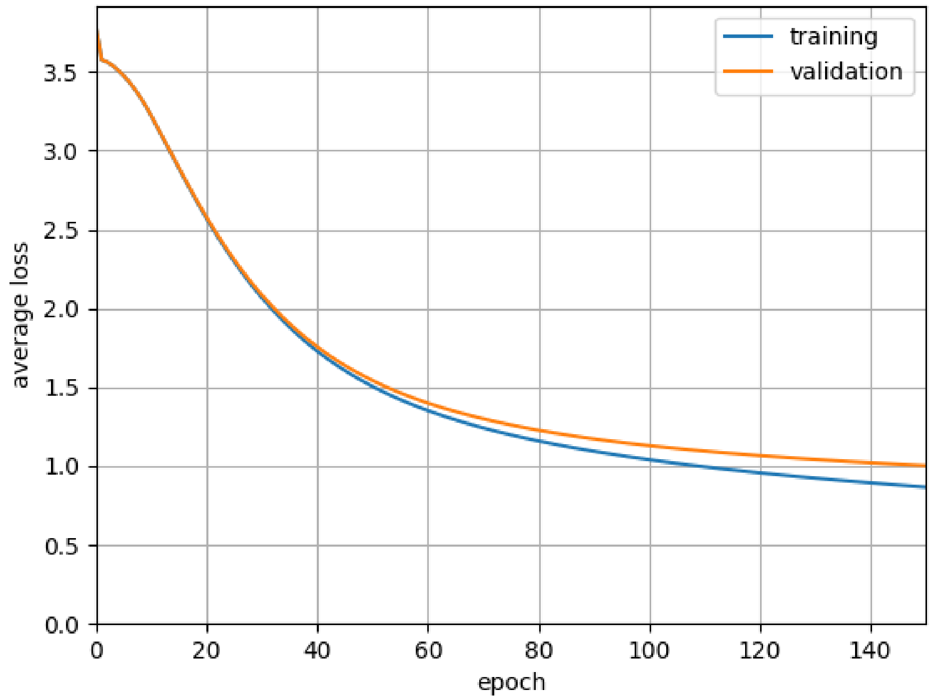


Validation accuracy: 0.5761111111111111

Test accuracy: 0.57

Learning rate with one tenth that learning rate=0.0003:





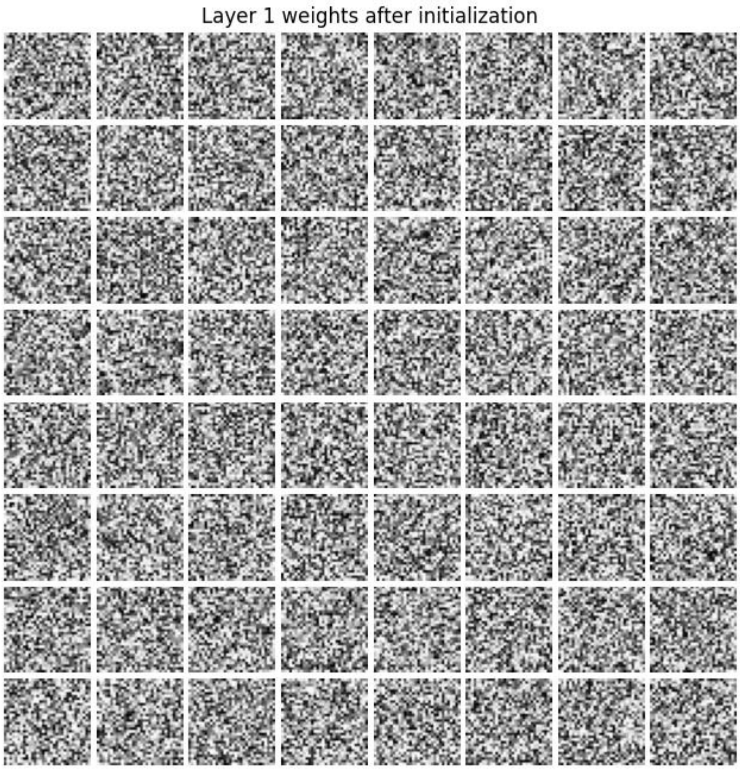
Validation accuracy: 0.7241666666666666

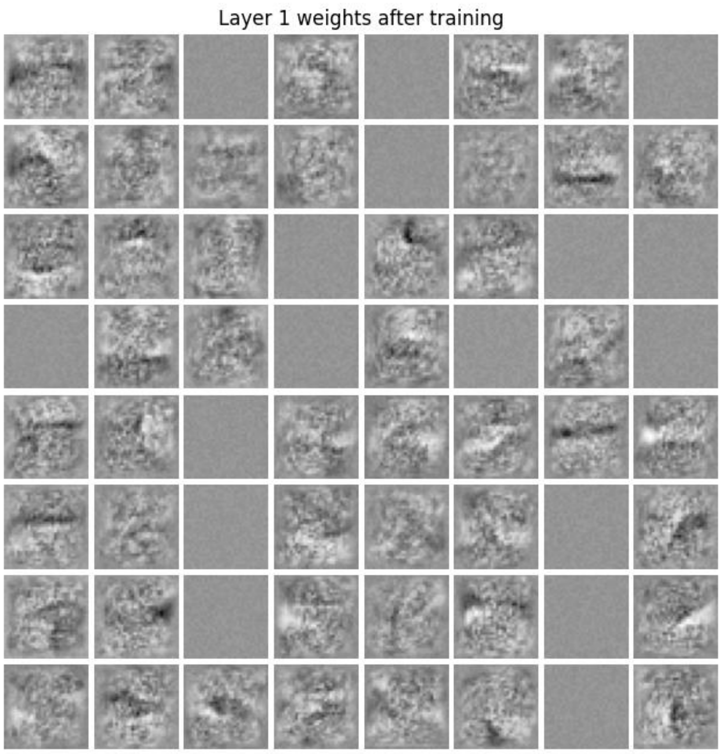
Test accuracy: 0.7233333333333334

The value learning rate affects the training by the spped to change weights. Too big learning rate will make the convergence process unstable and fluctuated, which leads to unstable loss and accuracy. On the contrary, too small learning rate will make the convergence too slow and need more epoches to make the model converge.

The final accuracy of the best network on the test set is the network with learning rate=0.003, and test accuracy is 0.7772222222222223.

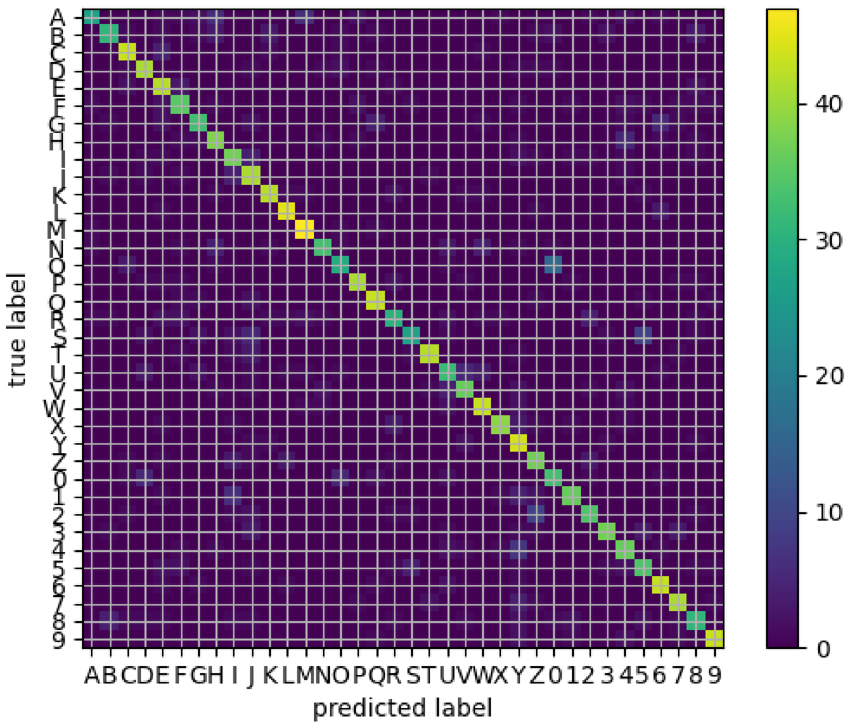
Q3.3





After first intialization, the weights turn out to be random patterns. After training, weights are updated by propagation and seem more ordered according to different inputs.

Q3.4



The top few pairs of classes that are most commonly confused include <0, O>, <5, S> and <Z, 2>. One possible reason why these pairs are confused may be they all seem similar, especially 0 and O. So it's also hard for model to identify these similar patterns.

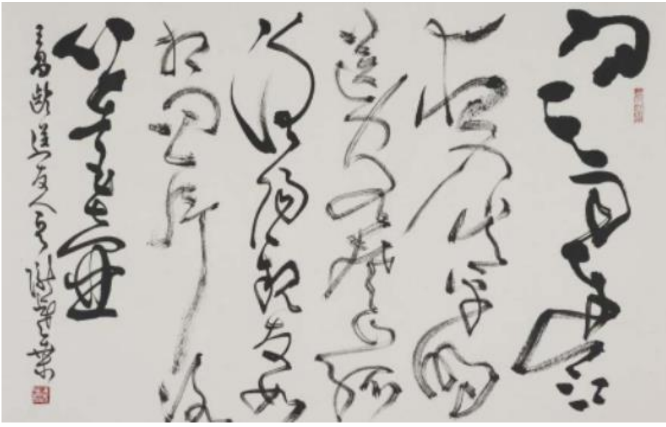
Q4.1

(1) The whole image only contains letters and numbers, no other non-letters appear.

(2) The size and gap between different letters are appropriate.

E.g.

The above picture only contains one car, no other letters.

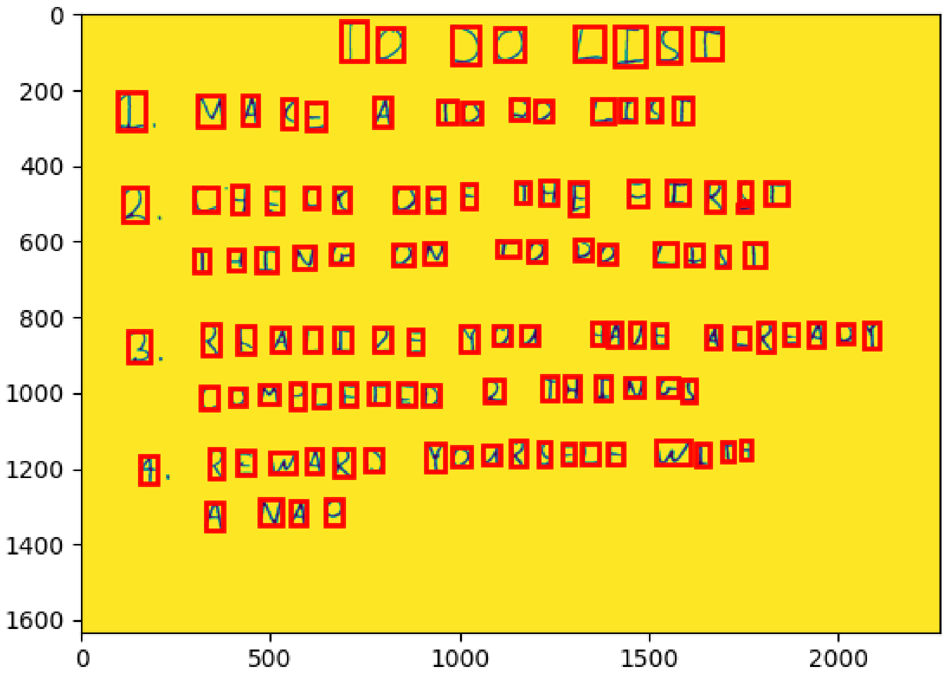


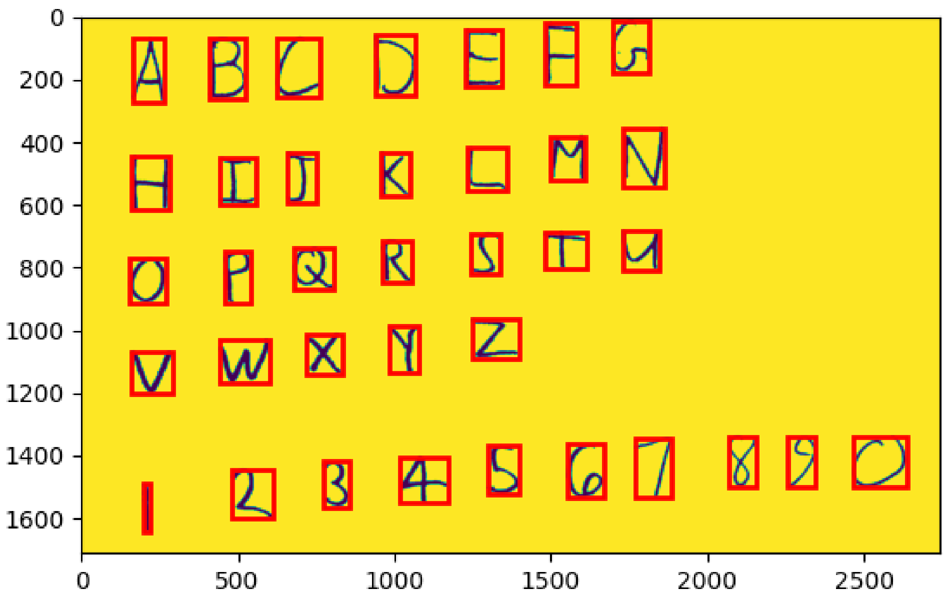
The above hand writing work doesn't include English letters and number, and the gap between characters are very casual.

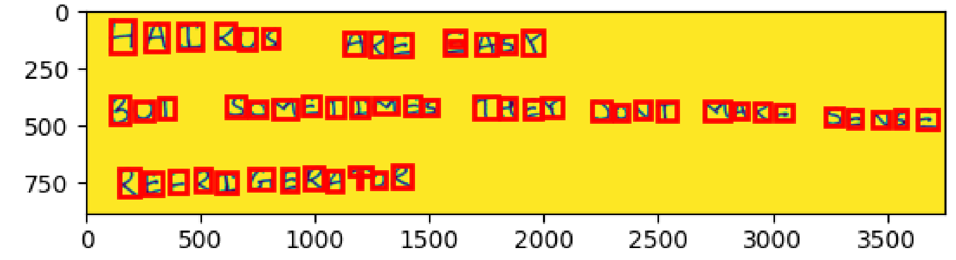
Q4.2

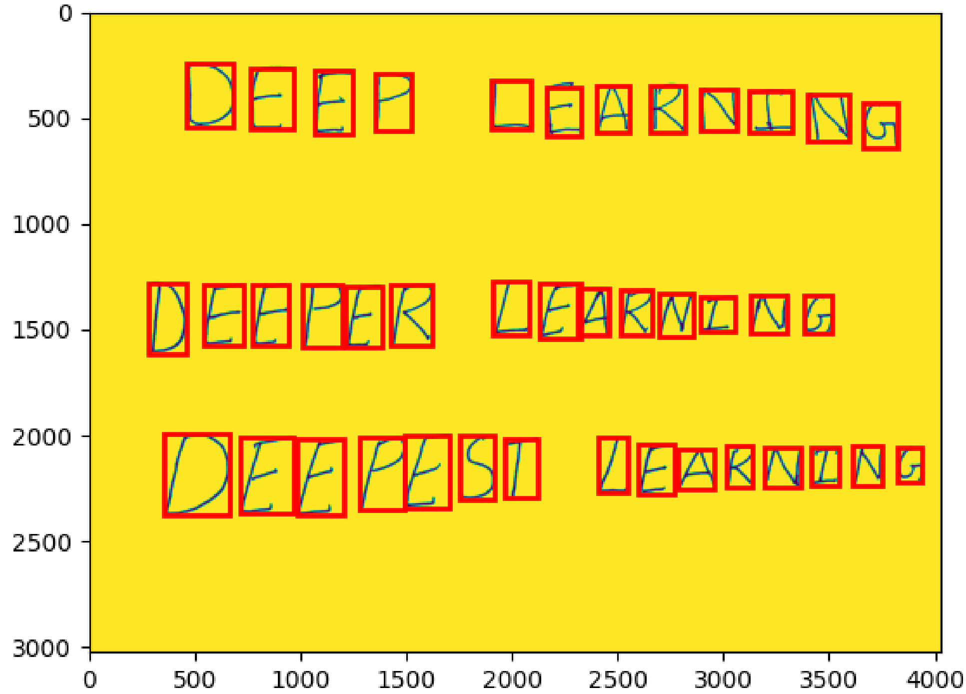
```
    def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold ->
morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

    #denoise
    img = skimage.restoration.denoise_bilateral(image,
multichannel=True)
    # greyscale
    img_grey = skimage.color.rgb2gray(img)

    # threshold, morphology
    thresh = skimage.filters.threshold_otsu(img_grey)
    bw = skimage.morphology.closing(img_grey < thresh,
skimage.morphology.square(5))

    # label
    label_img = skimage.morphology.label(bw, connectivity=2)
    properties = skimage.measure.regionprops(label_img)

    # skip small boxes
    mean_area = sum([x.area for x in properties]) / len(properties)
    for x in properties:
        if x.area > mean_area/2:
            bboxes.append(x.bbox)

    bw = (~bw).astype(float)

    return bboxes, bw
```

Q4.3


0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000


0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500


0
250
500
750
0
500
1000
1500
2000
2500
3000
3500


DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING
0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000
3500
4000

Q4.4

01_list:

TQ DQ LIIT
I HDX6 A TQ BQ LIIT
2 LH6CK JFF 3HE FIRFW
THING QW TQ DQ LI8T
3 RFALIZE YQU NAUE RLR6ADT
LQMPLFTCD J THING3
4 RFWDRD YQWRSFLF WITB
A NAP

02_letters:

2 B L D F F G
H I J K L M N
Q P Q R I T W
V W X T Z
1 Z 3 G B G 7 8 9 D

03_haiku:

HAIWUK ARH HMAGY
BWT SDMETIMBS TREY DQWT MAKG BHMGE
RBGRIGBRA1MQR

04_deep:

DEFP LLARHING
DEDPER LEARHIMG
DEEPE3T LEARNIMG

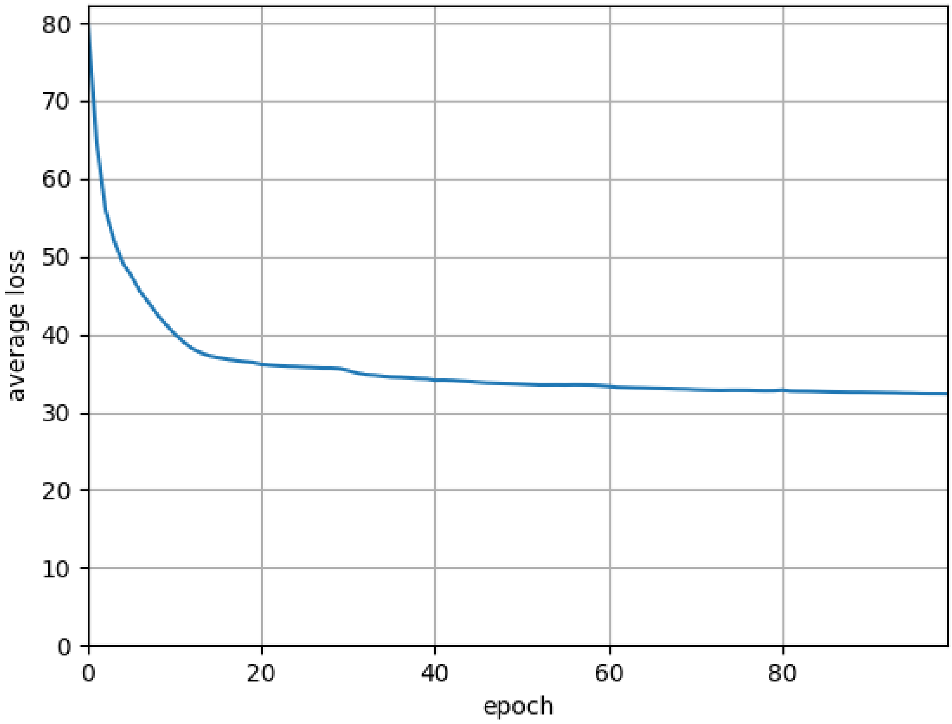
Q5.1.1

```
initialize_weights(train_x.shape[1], hidden_size, params, 'layer1')
initialize_weights(hidden_size, hidden_size, params, 'hidden1')
initialize_weights(hidden_size, hidden_size, params, 'hidden2')
initialize_weights(hidden_size, train_x.shape[1], params, 'output')

losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:

        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'hidden1', relu)
        h3 = forward(h2, params, 'hidden2', relu)
        probs = forward(h3, params, 'output', sigmoid)

        # your loss is now squared error
        loss = np.sum(np.square(probs - xb))
        total_loss += loss

        # delta is the d/dx of (x-y)^2
        delta1 = 2.0 * (probs - xb)
        delta2 = backwards(delta1, params, 'output', sigmoid_deriv)
        delta3 = backwards(delta2, params, 'hidden2', relu_deriv)
        delta4 = backwards(delta3, params, 'hidden1', relu_deriv)
        backwards(delta4, params, 'layer1', relu_deriv)

        for key in params.keys():

            if '_' in key:
                continue
            params[key] -= learning_rate * params['grad_' + key]

    losses.append(total_loss/train_x.shape[0])
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f}".format(itr,total_loss))
    if itr % lr_rate == lr_rate-1:
        learning_rate *= 0.9
```

Q5.1.2

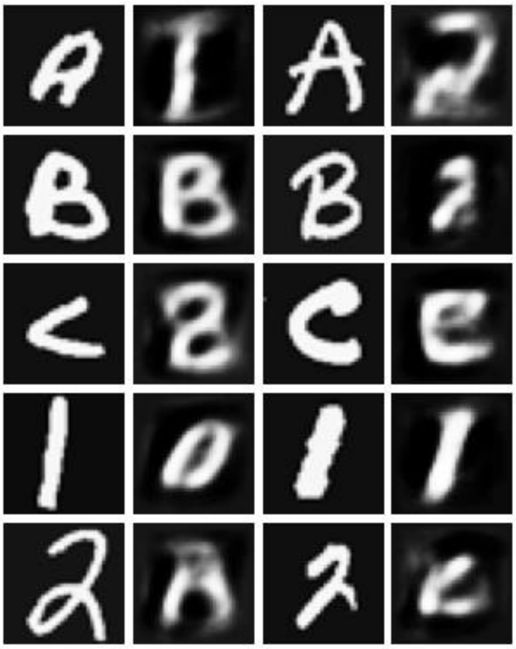
```
initialize_weights(train_x.shape[1], hidden_size , params,'layer1')
initialize_weights(hidden_size , hidden_size ,params,'hidden1')
initialize_weights(hidden_size , hidden_size ,params,'hidden2')
initialize_weights(hidden_size , train_x.shape[1] ,params,'output')

keys = [key for key in params.keys()]
for key in keys:
    params['m_' + key] = np.zeros(params[key].shape)

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:

        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'hidden1', relu)
        h3 = forward(h2, params, 'hidden2', relu)
        probs = forward(h3, params, 'output', sigmoid)

        # your loss is now squared error
        loss = np.sum(np.square(probs - xb))
        total_loss += loss

        # delta is the d/dx of (x-y)^2
        delta1 = 2.0 * (probs - xb)
        delta2 = backwards(delta1, params, 'output', sigmoid_deriv)
        delta3 = backwards(delta2, params, 'hidden2', relu_deriv)
        delta4 = backwards(delta3, params, 'hidden1', relu_deriv)
        backwards(delta4, params, 'layer1', relu_deriv)

        for key in params.keys():
            if '_' in key:
                continue
            params['m_' + key] = 0.9 * params['m_' + key] -
learning_rate * params['grad_' + key]
            params[key] += params['m_' + key]

    losses.append(total_loss/train_x.shape[0])
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f}".format(itr,total_loss))
    if itr % lr_rate == lr_rate-1:
        learning_rate *= 0.9
```

Q5.2



The training loss decrease fast in the first 20 epoches, then it starts decreasing more slowly and reaches a stable state after about 80 epoch.

Q5.3.1

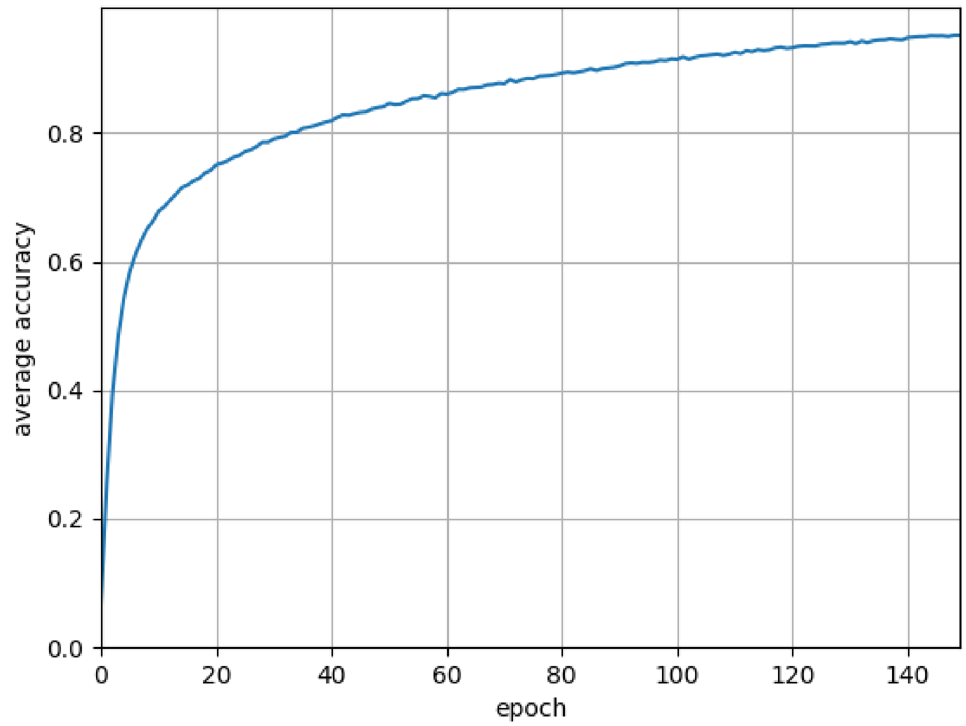


The reconstructed validation images seem more vague and blurred than the original images. And the shape of some reconstructed images are distorted or twisted.

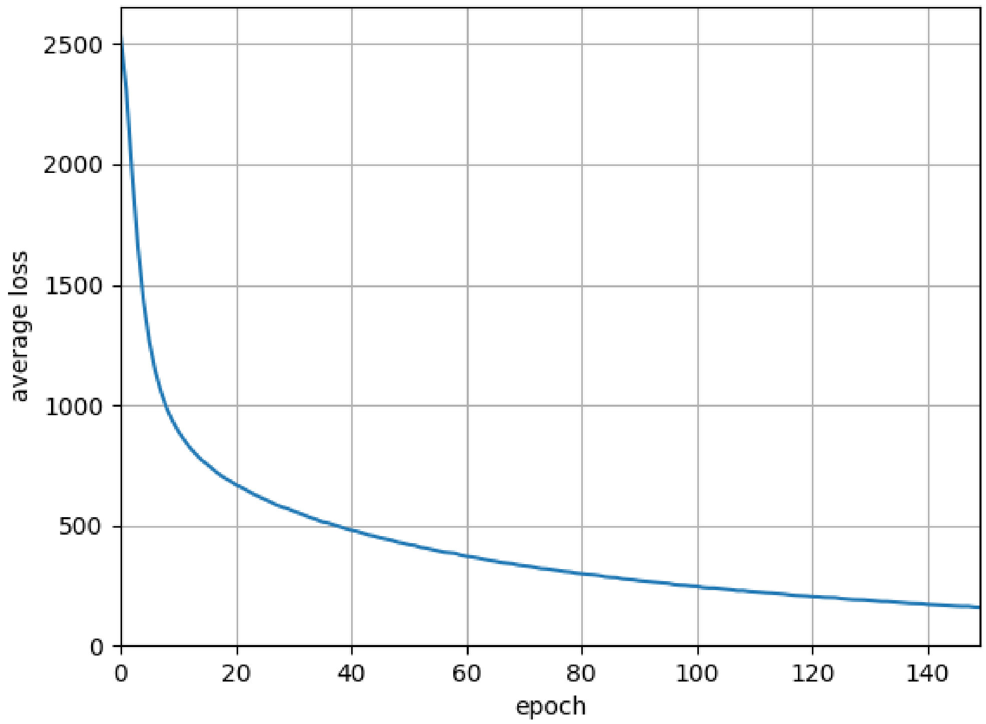
Q5.3.2

PSNR=15.81791007430785

Q6.1.1



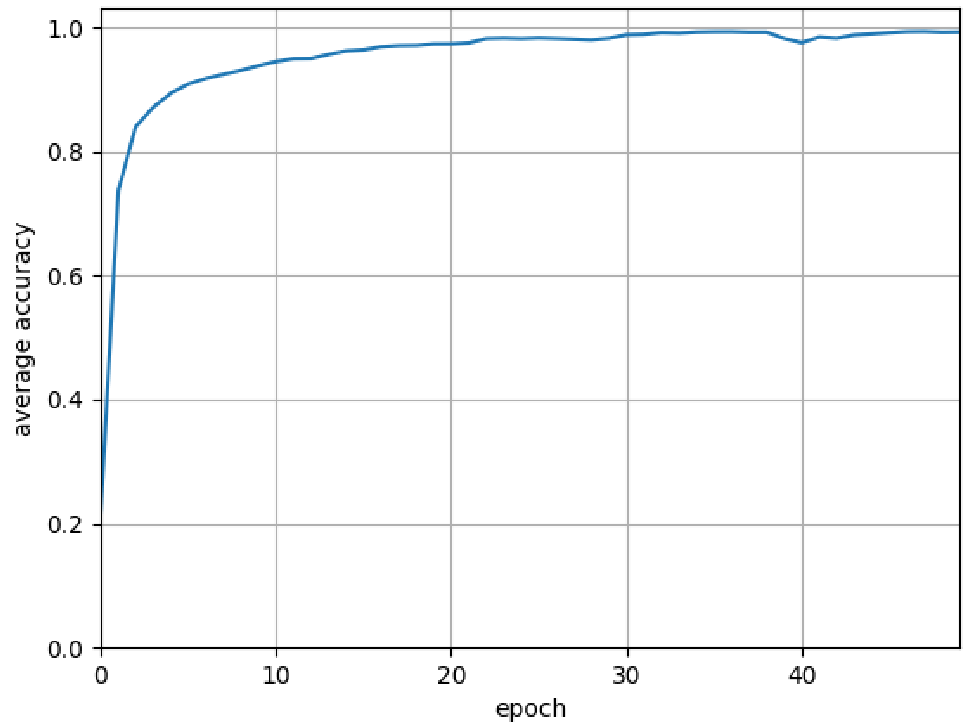
average accuracy
0.0
0.2
0.4
0.6
0.8
0
20
40
60
80
100
120
140
epoch

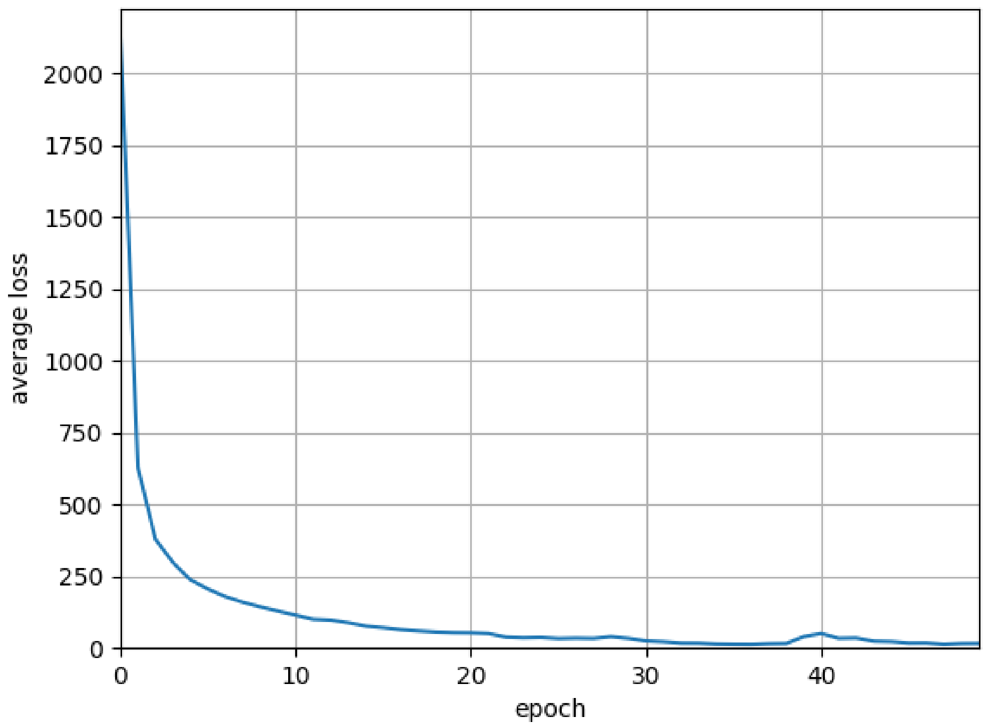


average loss
0
500
1000
1500
2000
2500
0
20
40
60
80
100
120
140
epoch

```
# 6.1.1
class orig_Net(nn.Module):
    def __init__(self, input_dim, hid_size, output_dim):
        super(orig_Net, self).__init__()
        self.fc1 = nn.Linear(input_dim, hid_size)
        self.fc2 = nn.Linear(hid_size, output_dim)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = self.fc2(x)
        return x
simple_net = orig_Net(train_x.shape[1], hidden_size,train_y.shape[1])
optimizer = torch.optim.SGD(simple_net.parameters(),
lr=learning_rate, momentum=0.9)

train_loss = []
train_acc = []
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for data in train_loader:
        inputs = torch.autograd.Variable(data[0])
        labels = torch.autograd.Variable(data[1])
        targets = torch.max(labels, 1)[1]

        pred = simple_net(inputs)
        loss = nn.functional.cross_entropy(pred, targets)
        total_loss += loss.item()
        preds = torch.max(pred, 1)[1]
        avg_acc += preds.eq(targets.data).cpu().sum().item()

        # backward
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    avg_acc = avg_acc / train_y.shape[0]
    train_loss.append(total_loss)
    train_acc.append(avg_acc)

    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc :
{:.2f}".format(itr,total_loss,avg_acc))
```

Q6.1.2



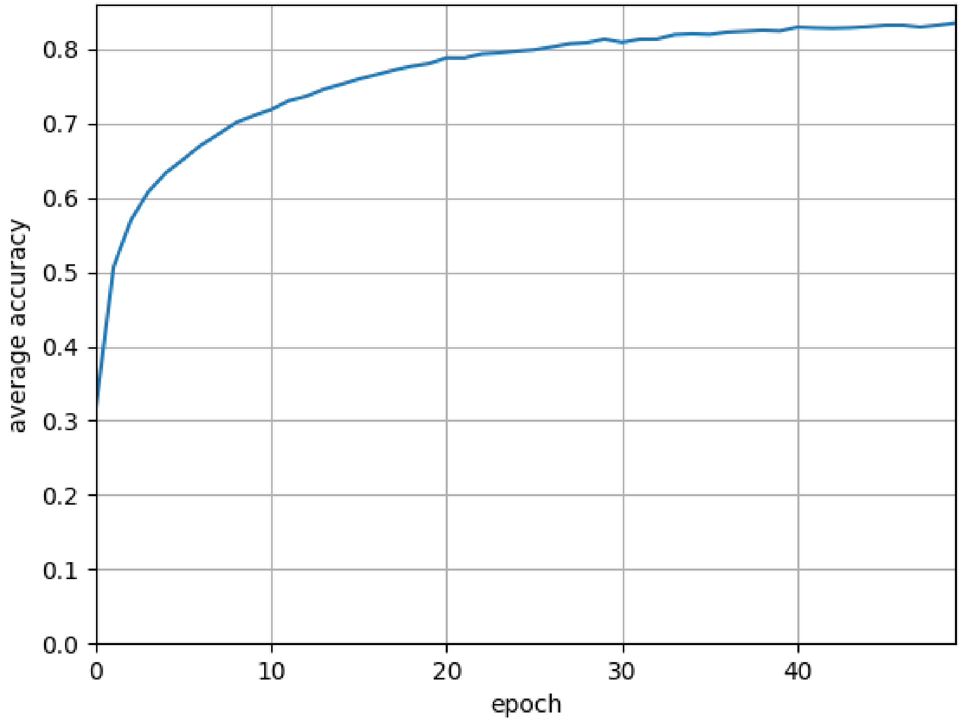


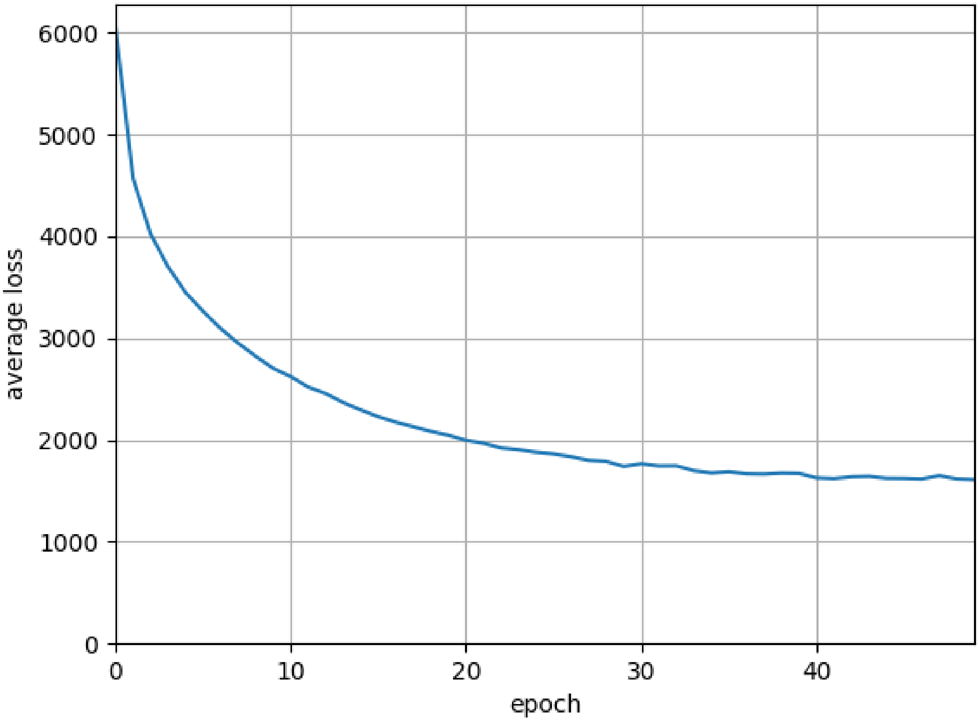
We can see this convolution model has much higher accuracy and lower loss within less epoches compared to the fully-connected network.

```
# 6.1.2
max_iters = 50
# pick a batch size, learning rate
batch_size = 15
learning_rate = 0.003

train_x = np.array([train_x[i, :].reshape((32, 32)) for i in
range(train_x.shape[0])])
test_x = np.array([test_x[i, :].reshape((32, 32)) for i in
range(test_x.shape[0])])

trainset_x, trainset_y =
torch.from_numpy(train_x).type(torch.float32).unsqueeze(1),
torch.from_numpy(train_y).type(torch.long)
test_x, test_y =
torch.from_numpy(test_x).type(torch.float32).unsqueeze(1),
torch.from_numpy(test_y).type(torch.long)

train_loader = DataLoader(TensorDataset(trainset_x, trainset_y),
batch_size=batch_size, shuffle=True, drop_last=True)
test_loader = DataLoader(TensorDataset(test_x, test_y),
shuffle=False)


class ConvNet1(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ConvNet1, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2))
        self.conv2 = nn.Sequential(
            nn.Conv2d(8, 16, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2))
        self.fc1 = nn.Linear(5 * 5 * 16, 50)
        self.fc2 = nn.Linear(50, output_dim)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(-1, 5*5*16)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
```

```
        return x

net = ConvNet1(train_x.shape[1], train_y.shape[1])
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate,
momentum=0.9)

train_loss = []
train_acc = []
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0

    for data in train_loader:

        inputs = torch.autograd.Variable(data[0])
        labels = torch.autograd.Variable(data[1])
        targets = torch.max(labels, 1)[1]

        pred = net(inputs)
        loss = F.cross_entropy(pred, targets)
        total_loss += loss.item()
        preds = torch.max(pred, 1)[1]
        avg_acc += preds.eq(targets.data).cpu().sum().item() /
labels.shape[0]

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    avg_acc = avg_acc / len(train_loader)
    train_loss.append(total_loss)
    train_acc.append(avg_acc)

    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc :
{:.2f}".format(itr,total_loss,avg_acc))
```

Q6.1.3


average accuracy
epoch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40


average loss
epoch
0
1000
2000
3000
4000
5000
6000
0
10
20
30
40

```
    # 6.1.3
class ConvNet2(nn.Module):
    def __init__(self):
        super(ConvNet2, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(5 * 5 * 16, 120)
        self.fc2 = nn.Linear(120, 50)
        self.fc3 = nn.Linear(50, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 5 * 5 * 16)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


if __name__ == '__main__':      # only for 6.1.3, because of
multiprocess calculation
    max_iters = 50
    # pick a batch size, learning rate
    batch_size = 15
    learning_rate = 0.003

    transformer_train = torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])
    transformer_test = transformer_train

    train_loader = DataLoader(
        torchvision.datasets.CIFAR10('../data/', train=True,
download=True, transform=transformer_train),
        batch_size=batch_size, shuffle=True, num_workers=4)

    test_loader = DataLoader(
        torchvision.datasets.CIFAR10('../data/', train=False,
download=True, transform=transformer_test),
        shuffle=False, num_workers=4)
```

```
    net = ConvNet2()
    optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate,
momentum=0.9)

    train_loss = []
    train_acc = []
    for itr in range(max_iters):
        total_loss = 0
        avg_acc = 0

        for data in train_loader:

            inputs = torch.autograd.Variable(data[0])
            labels = torch.autograd.Variable(data[1])

            pred = net(inputs)
            loss = F.cross_entropy(pred, labels)
            total_loss += loss.item()
            preds = torch.max(pred, 1)[1]
            avg_acc += preds.eq(labels.data).cpu().sum().item() /
labels.shape[0]

            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        avg_acc = avg_acc / len(train_loader)
        train_loss.append(total_loss)
        train_acc.append(avg_acc)

        if itr % 2 == 0:
            print("itr: {:02d} \t loss: {:.2f} \t acc :
{:.2f}".format(itr,total_loss,avg_acc))
```