Q1.1

$softmax\ (x_i) = \frac{e^{x_i}}{\Sigma_j e^{x_j}}$

$softmax\ (x_i\ +\ c) = \frac{e^{x_i + c}}{\Sigma_j e^{x_j + c}}$

$softmax\ (x_i\ +\ c) = \frac{e^{x_i}\ e^c}{\Sigma_j e^{x_j}\ e^c}$

$softmax\ (x_i\ +\ c) = \frac{e^{x_i}}{\Sigma_j e^{x_j}}\ = softmax\ (x_i)$

Using $c\ = -\ max\ x_i$ prevents numerical instability when dealing with large input values. It ensures that the exponential function does not overflow.

Q1.2

- Range: [0,1], Sum: 1
- Probability distribution
- Role:
    - First step: Widens the gap between input values.
    - Second step: Normalizes them.
    - Third step: Creates probability distribution through probability calculation of normalized values.

Q1.3

$y_n\ =\ w_n x_n +\ b_n$ (1)

$y_{n-1}\ =\ w_{n-1} x_{n-1} +\ b_{n-1}$ (2)

Since there is no non-linear activation function,

$x_n\ =\ y_{n-1}\ =\ w_{n-1} x_{n-1} +\ b_{n-1}$ (3)

Subs (3) in (1),

$y_n\ =\ w_n (w_{n-1} x_{n-1} +\ b_{n-1}) +\ b_n$

$y_n\ =\ w_n w_{n-1} x_{n-1} +\ w_n b_{n-1} +\ b_n$

$y_n\ =\ Wx +\ b$ (4)

Eq (4) resembles linear regression

Q1.4

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

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

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

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

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

$$= \frac{1}{(1+e^{-x})}1 - \frac{1}{(1+e^{-x})} = \sigma(x)[1\text{-}\sigma(x)]$$

Q1.5

Given:

$$y = wx + b \qquad (5)$$

$$\frac{\partial J}{\partial y} = \delta$$

Gradient of J with respect w, J with respect x, and J with respect b from (5) is given below:

$$\frac{\partial y}{\partial w} = x\,, \frac{\partial y}{\partial x} = w, \frac{\partial y}{\partial b} = 1$$

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial w} = \delta x^T$$

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = w^T\delta$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta$$

Q1.6

1. Derivative of sigmoid function has range [0, 0.25]. When used for many layers, the gradient of the loss function with respect to weights become very small. Thus, vanishing.
2. Sigmoid range: [0, 1]   Tanh range: [-1, 1]
    a. Tanh is preferred because it is centered at 0 and includes negative values.
3. Derivative of tanh function has range [0, 1]. It is larger than sigmoid. It approaches zero slowly when moved away from origin. Hence, has less of a vanishing gradient problem.
4. $\sigma(x) = \frac{1}{(1+e^{-x})}$

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

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

Q2.1.1

If a network is initialized with all zeroes, the layers calculate the same output. No meaningful learning is done, making the network useless. It outputs a sub-optimal solution.

Q2.1.2

```
def initialize_weights(in_size, out_size, params, name=""):
    W, b = None, None
    r = np.sqrt(6.0 / (in_size + out_size))
    W = np.random.uniform(-r, r, (in_size, out_size))
    b = np.zeros(out_size)

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

Q2.1.3

If a network is initialized with random numbers, the layers calculate different output. Scaling the initialization based on layer size improves training stability. Ensures activations are within a suitable range, thus preventing vanishing gradients.

Q2.2.1

```
def sigmoid(x):
    res = None
    ##########################
    ##### your code here #####
    ##########################
    res = 1.0 / (1.0 + np.exp(-x))
    return res
```

```
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]

    ##########################
    ##### your code here #####
    ##########################
    pre_act = X.dot(W) + 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

    ##########################
    ##### your code here #####
    ##########################
    x -= np.max(x, axis=1, keepdims=True)
    res = np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True)

    return res
```

Q2.2.3

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

    ##########################
    ##### your code here #####
    ##########################
    loss = -np.sum(y * np.log(probs))
    acc = np.mean(np.argmax(probs, axis=1) == np.argmax(y, axis=1))

    return loss, acc
```

Q2.3

```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
    ##########################
    ##### your code here #####
    ##########################
    grad_X = (delta * activation_deriv(post_act)).dot(W.T)
    grad_W = (X.T).dot(delta * activation_deriv(post_act))
    grad_b = np.sum(delta * activation_deriv(post_act), 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 = []
    ##########################
    ##### your code here #####
    ##########################
    tBatches = x.shape[0] // batch_size
    for _ in range(tBatches):
        indices = np.random.randint(0, x.shape[0], batch_size)
        xB = x[indices]
        yB = y[indices]
        batches.append((xB, yB))

    return batches
```

```
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb, yb in batches:
        # pass
        # forward
        h1 = forward(xb, params, "layer1")
        probs = forward(h1, params, "output", softmax)
        # 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, "output", linear_deriv)
        backwards(delta2, params, "layer1", sigmoid_deriv)
        # apply gradient
        # gradients should be summed over batch samples
        params["Wlayer1"] -= learning_rate * params["grad_Wlayer1"]
        params["blayer1"] -= learning_rate * params["grad_blayer1"]
        params["Woutput"] -= learning_rate * params["grad_Woutput"]
        params["boutput"] -= learning_rate * params["grad_boutput"]

    total_loss /= batch_num
    avg_acc /= batch_num

    if itr % 100 == 0:
        print("Iteration: {:02d} \t Loss: {:.2f} \t Accuracy :
{:.2f}".format(itr, total_loss, avg_acc))
```

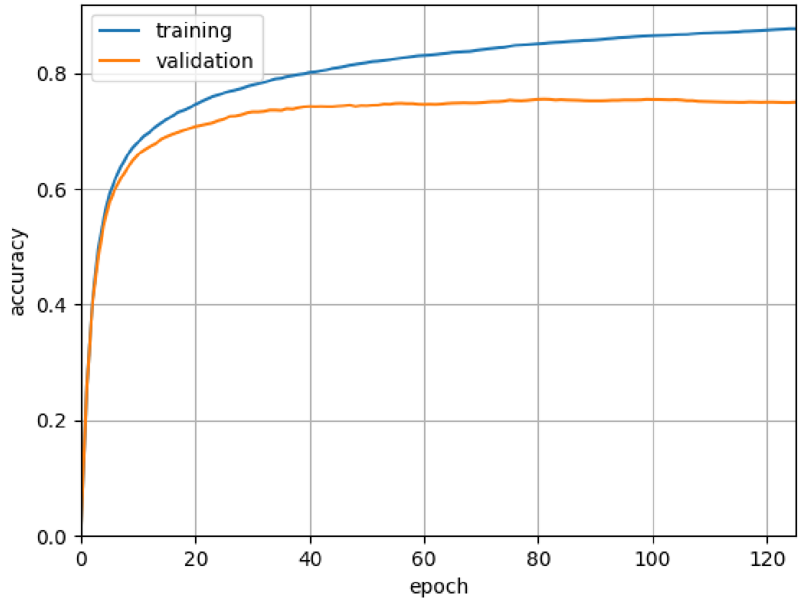
Q2.5

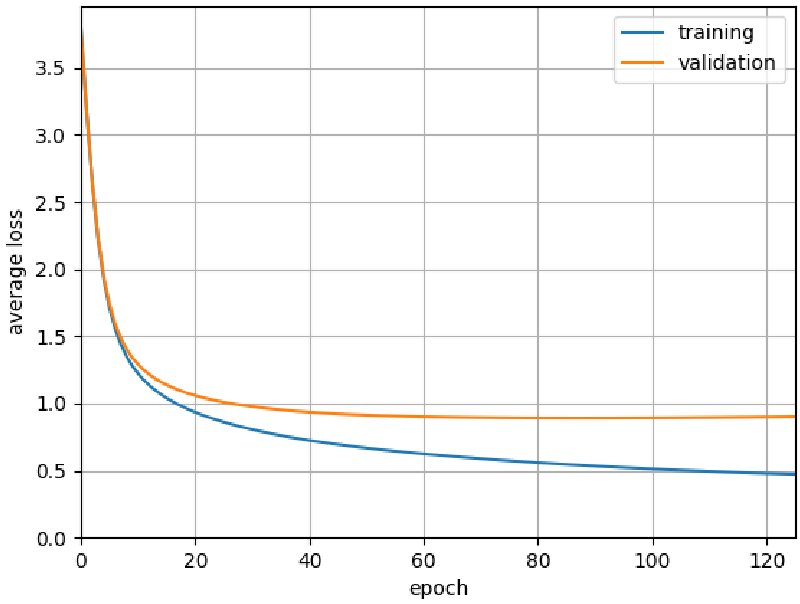
```python
for k, v in params.items():
    if "_" in k:
        continue
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   subtract 2*epsilon
    #   run the network
    #   get the loss
    #   restore the original parameter value
    #   compute derivative with central diffs
    ##########################
    ##### your code here #####
    ##########################
    if 'W' in k:
        for i in range(v.shape[0]):
            for j in range(v.shape[1]):
                v[i, j] += eps
                h1 = forward(x, params, "layer1")
                probs = forward(h1, params, "output", softmax)
                positive_loss = compute_loss_and_acc(y, probs)[0]

                v[i, j] -= 2 * eps
                h1 = forward(x, params, "layer1")
                probs = forward(h1, params, "output", softmax)
                negative_loss = compute_loss_and_acc(y, probs)[0]

                v[i, j] += eps
                grad_loss = (positive_loss - negative_loss) / (2 * eps)
                params["grad_" + k][i, j] = grad_loss

    elif 'b' in k:
        for i in range(v.shape[0]):
            v[i] += eps
            h1 = forward(x, params, "layer1")
            probs = forward(h1, params, "output", softmax)
            positive_loss = compute_loss_and_acc(y, probs)[0]

            v[i] -= 2 * eps
            h1 = forward(x, params, "layer1")
            probs = forward(h1, params, "output", softmax)
            negative_loss = compute_loss_and_acc(y, probs)[0]

            v[i] += eps
            grad_loss = (positive_loss - negative_loss) / (2 * eps)
            params["grad_" + k][i] = grad_loss
```

Q3.1

Validation accuracy: 0.7502777777777778

Test accuracy: 0.7566666666666667





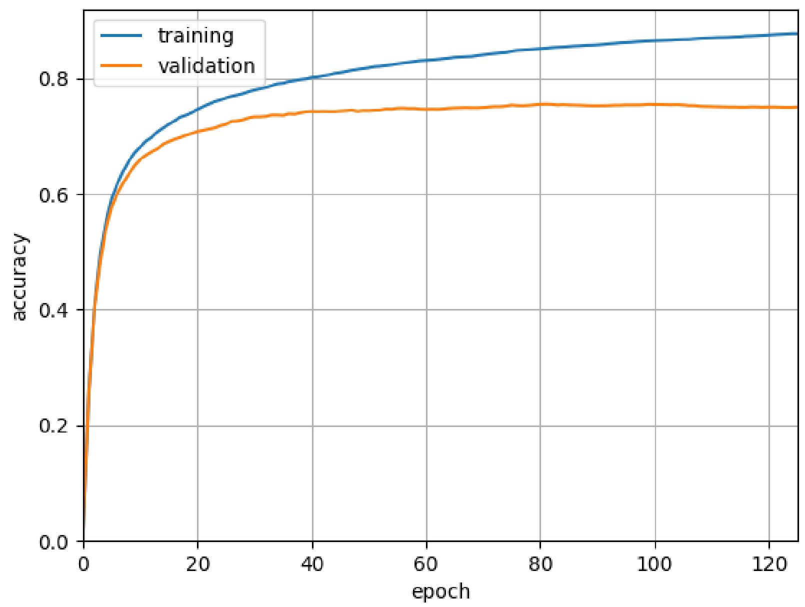
Q3.2

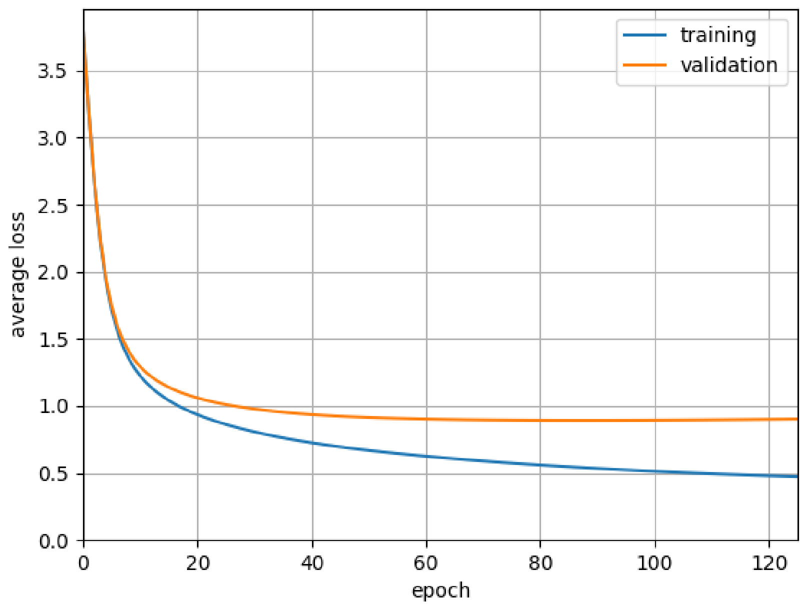
1X

learning_rate = 0.002

Validation accuracy: 0.7502777777777778

Test accuracy: 0.7566666666666667





10X

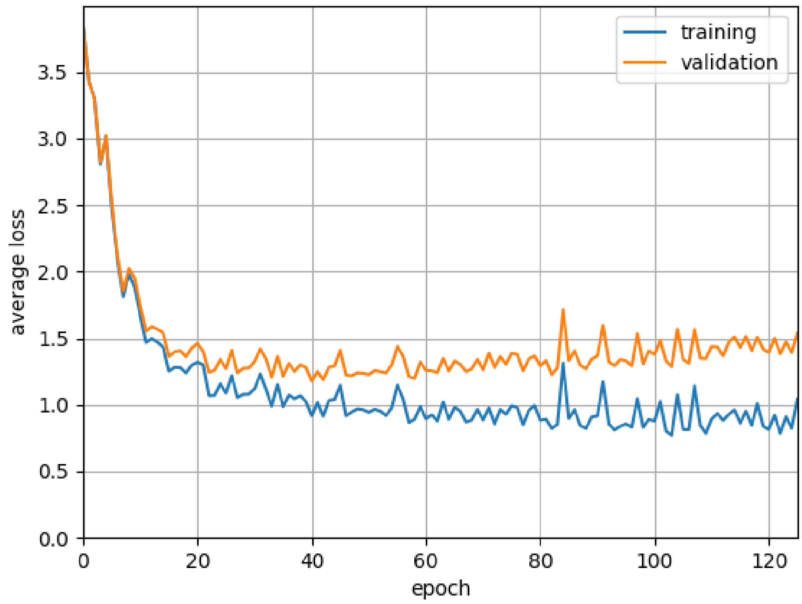
learning_rate = 0.02

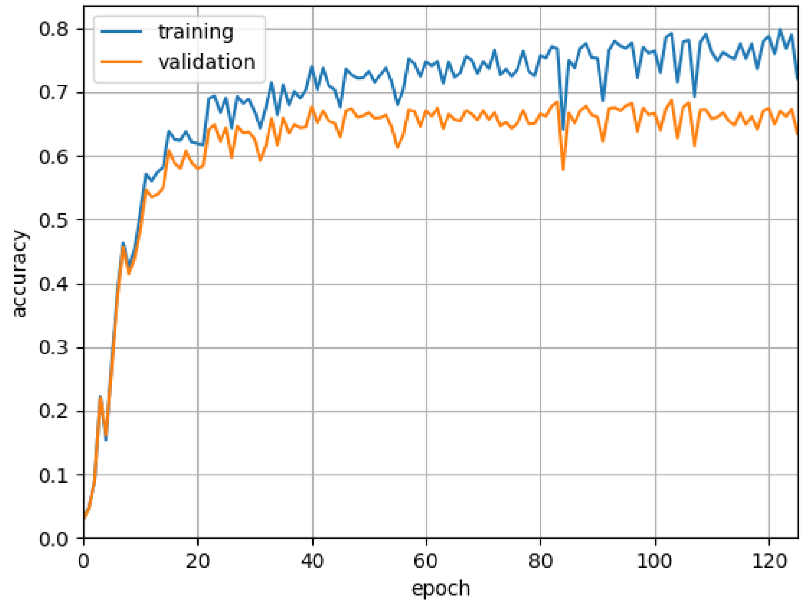
Validation accuracy: 0.6347222222222222

Test accuracy: 0.6438888888888888

Increasing the learning rate caused oscillations.

The losses are also high and the accuracy is low compared to 1X.





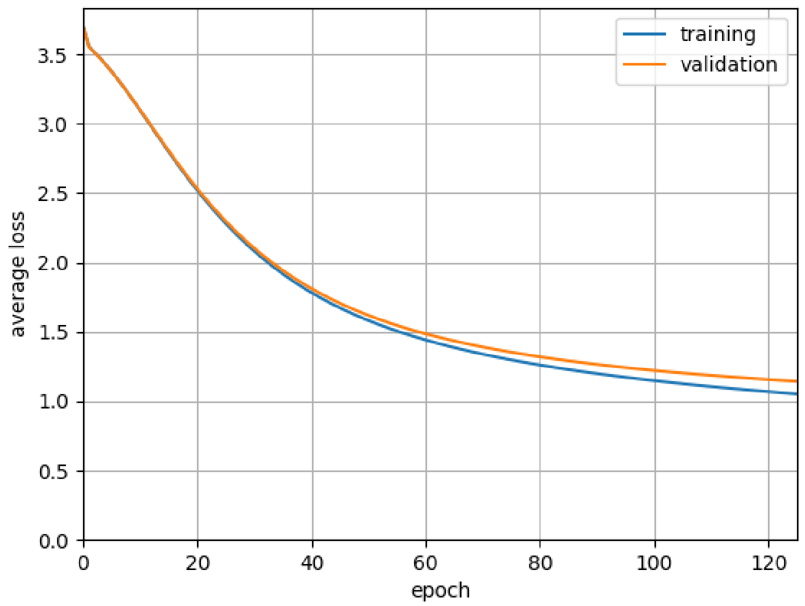
0.1X

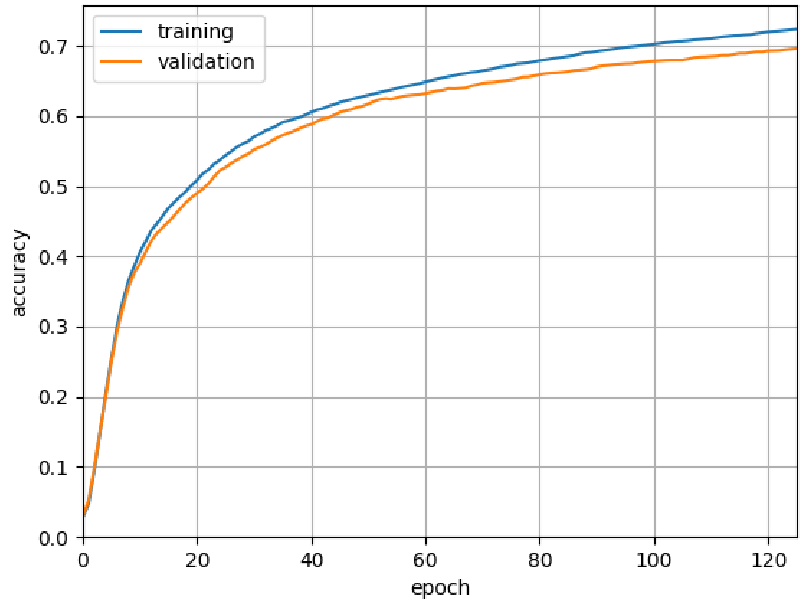
learning_rate = 0.0002

Validation accuracy: 0.6961111111111111

Test accuracy: 0.6972222222222222

The model is yet to converge because of the low learning rate. For same epochs, accuracy is low.

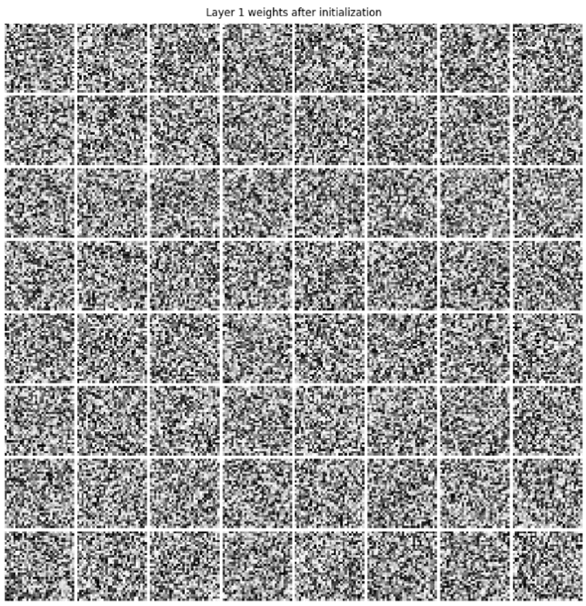




Best accuracy: 0.7566666666666667

Best network: Learning rate 1X

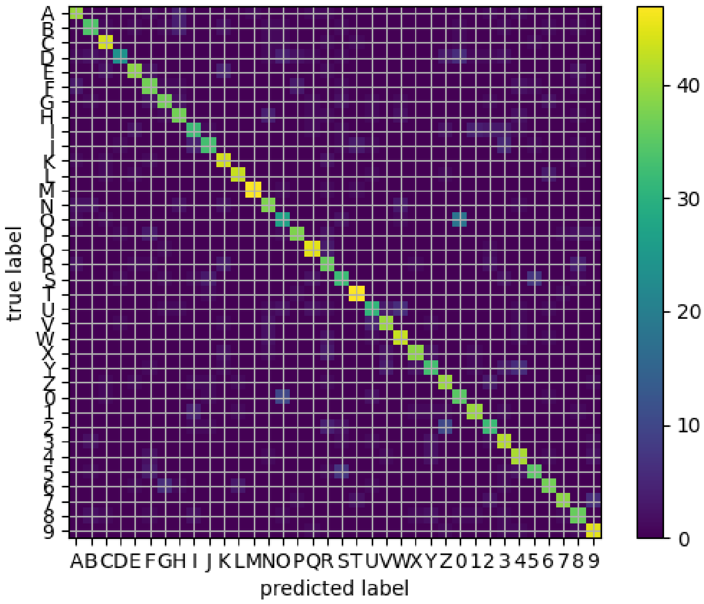
Q3.3



Does not have any pattern. Is random noise.

Has some pattern. Is not random noise.

Q3.4



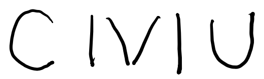
Those which have similar shape are most confused. For example, ‘0’ and ‘O’; ‘2’ and ‘Z’

Q4.1

The two assumptions are: 1. Characters does not overlap. 2. Each character is continuous.

Example 1: Character overlap

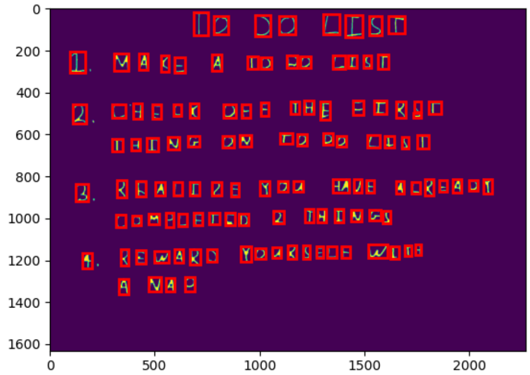
Example 2: Discontinuity in character

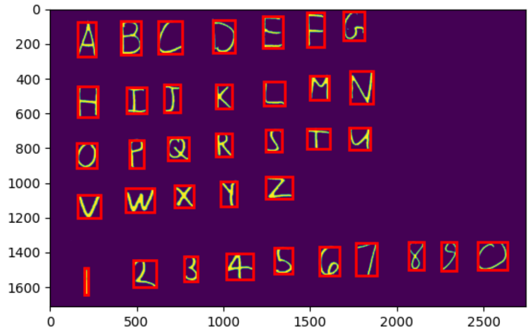


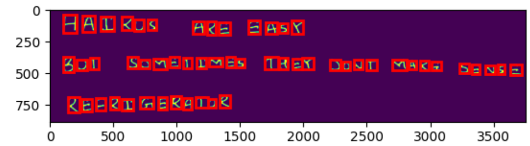
Q4.2

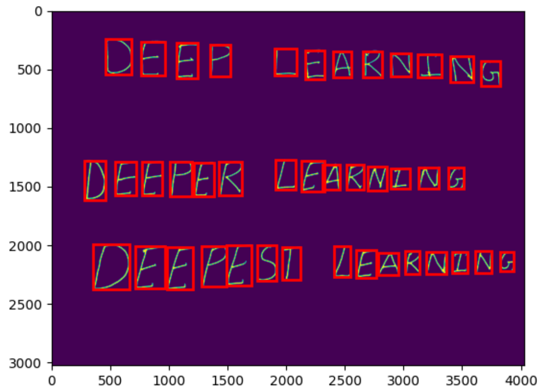
```
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold ->
morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions
    ##########################
    ##### your code here #####
    ##########################
    # Estimate noise
    noise = skimage.restoration.estimate_sigma(image, average_sigmas=True)
    win_size = max(5, 2*np.ceil(3*noise)+1)
    # Denoise
    denoisedImage = skimage.restoration.denoise_bilateral(image,
win_size=win_size, channel_axis=-1)
    # Greyscale
    greyscaleImage = skimage.color.rgb2gray(denoisedImage)
    # Threshold
    threshold = skimage.filters.threshold_otsu(greyscaleImage)
    # Morphology
    bw = skimage.morphology.closing(greyscaleImage<threshold,
skimage.morphology.square(10))
    # Label
    label = skimage.measure.label(bw)
    # Skip small boxes
    regions = skimage.measure.regionprops(label)
    for region in regions:
        if region.area > 100:
            bboxes.append(region.bbox)

    return bboxes, bw
```

Q4.3









Q4.4

01_list

T0 D0 LI5T

I MAKE A T0 00 LIST

2 CHVCK DFE THE FIRHTT

THING 0N T0 O0 LXST

3 R8ALIZEY0U MVE ALREA0Y

C0MPLET2D 2 THINGS

4 REWARD Y0URSEEF WITH

A NAP

02_letters

A B C D G F G

H I I K L M N

Q P Q R S T U

V W X Y Z

Z 3 4 B G 7 8 7 Q 8

03_haiku

HAZRUS ARR BAGX

BUT SOMRTZMBA TAR9 DONT MAKR BRNAR

RRGRXGBRAT0R

04_deep

CFFM LKAKMINW

DHBTEK LEAKNING

CEBPEBP EEARNING

Classifies >50% of the letters in each of the sample images.

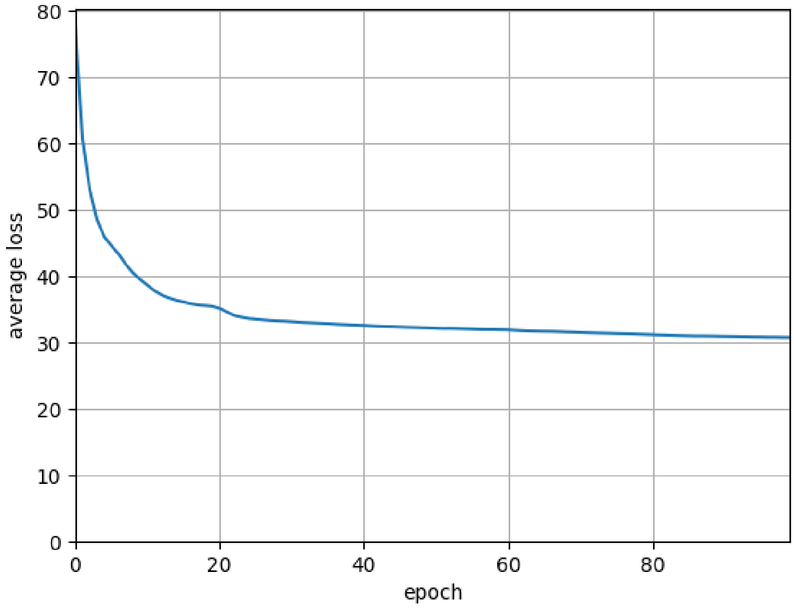
Q5.1.1

```
initialize_weights(train_x.shape[1], hidden_size, params, 'input')
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')

layers =
['Winput','Whidden1','Whidden2','Woutput','binput','bhidden1','bhidden2','bout
put']
for layer in layers:
    params['m_'+layer] = np.zeros_like(params[layer])
```

Q5.1.2

```
# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    loss = 0
    for xb,_ in batches:
        # training loop can be exactly the same as q2!
        # your loss is now squared error
        # delta is the d/dx of (x-y)^2
        # to implement momentum
        #   use 'm_'+name variables in initialize_weights from nn.py
        #   to keep a saved value
        #   params is a Counter(), which returns a 0 if an element is missing
        #   so you should be able to write your loop without any special
conditions
        # forward pass
        h1 = forward(xb, params, 'input', relu)
        h2 = forward(h1, params, 'hidden1', relu)
        h3 = forward(h2, params, 'hidden2', relu)
        probs = forward(h3, params, 'output', sigmoid)
        # loss
        loss = np.sum((probs - xb)**2)
        total_loss += loss
        # backward
        delta1 = 2*(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, 'input', relu_deriv)

        # apply gradient, remember to update momentum as well
        for layer in layers:
            params['m_'+layer] = 0.9*params['m_'+layer] - learning_rate *
params['grad_'+layer]
            params[layer] += params['m_'+layer]
```

Q5.2



Loss reduced drastically for 20 epochs after which not much change.

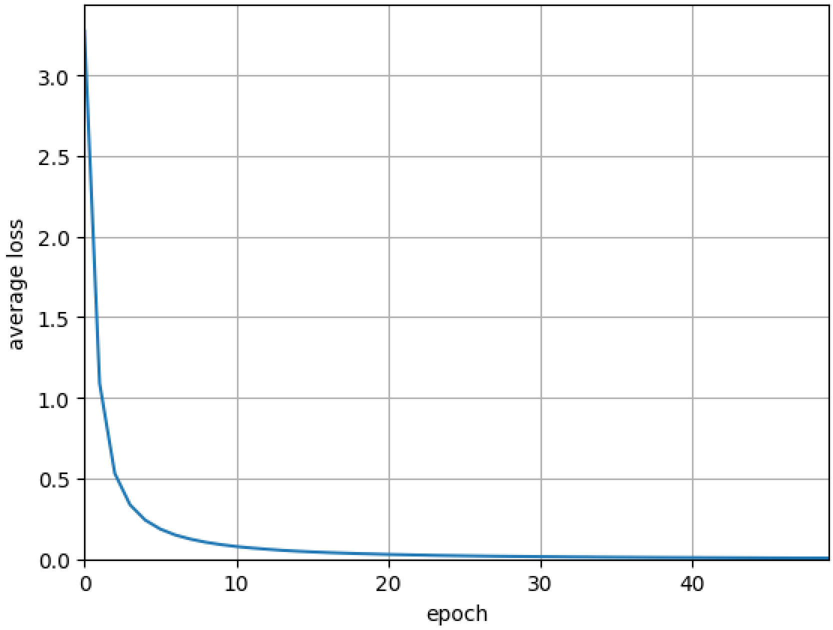
Q5.3.1

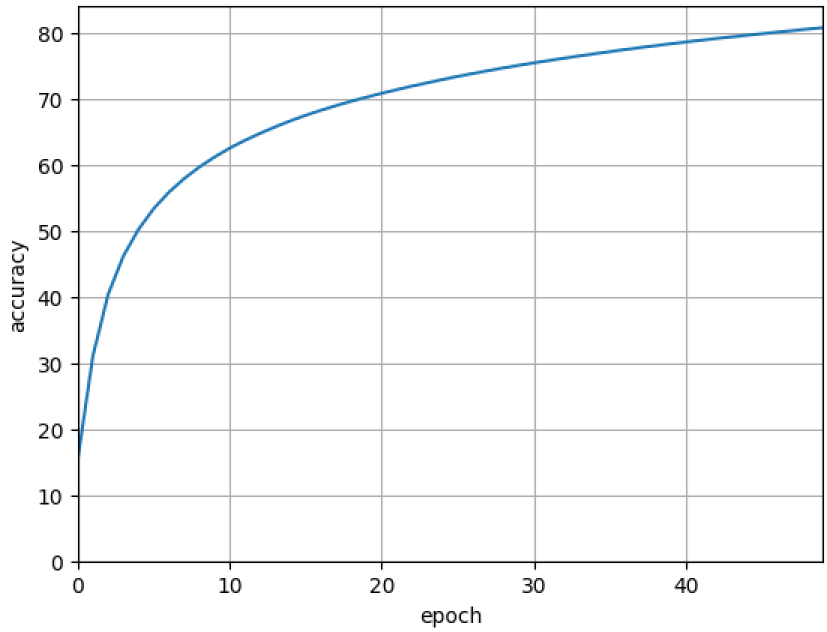
Images reconstructed are blurry but are close to original.

Q5.3.2

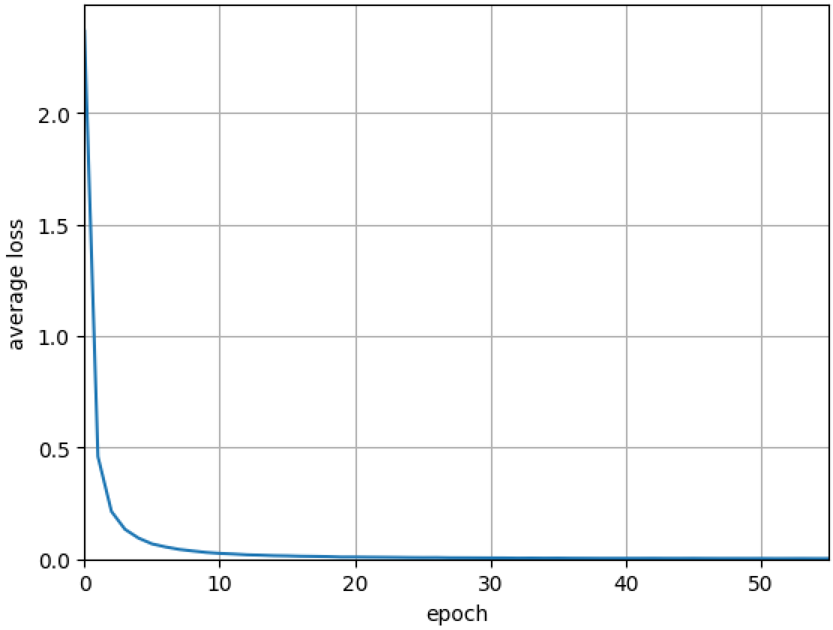
PSNR: 14.284133657921881

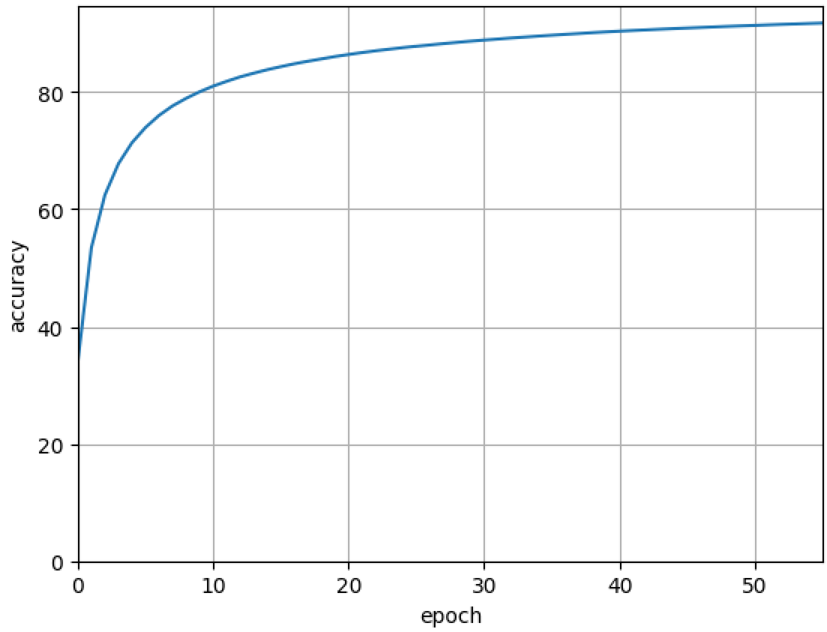
Q6.1.1





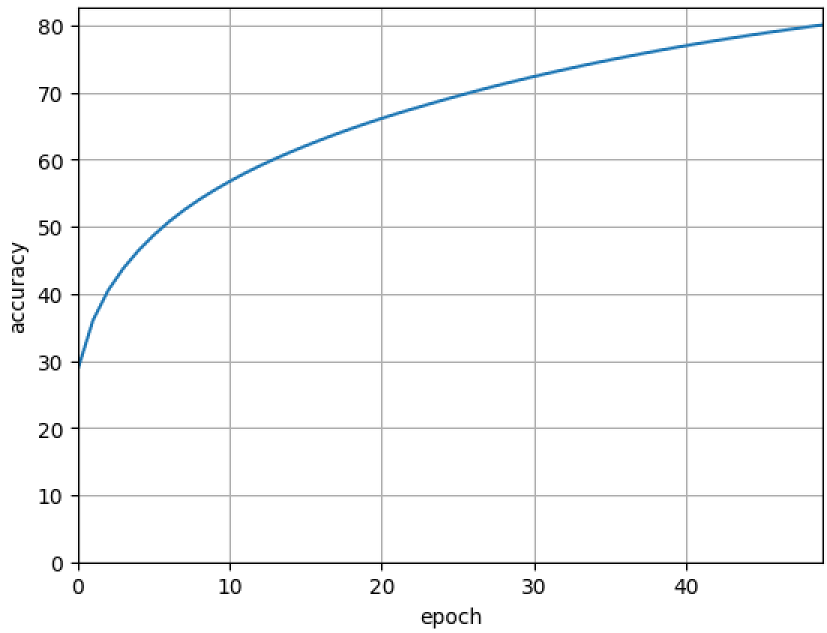
Q6.1.2



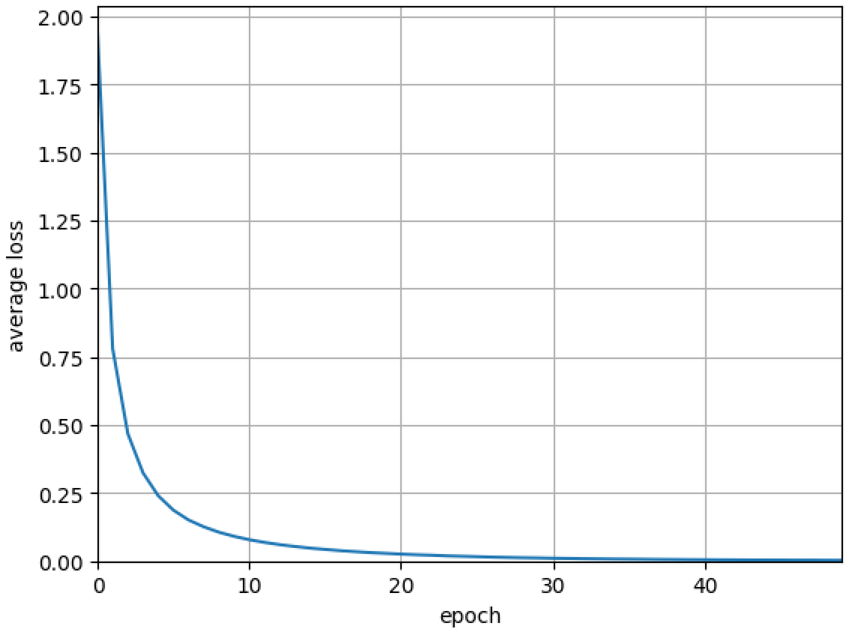


CNN converged faster and has higher accuracy than fully-connected network

Q6.1.3



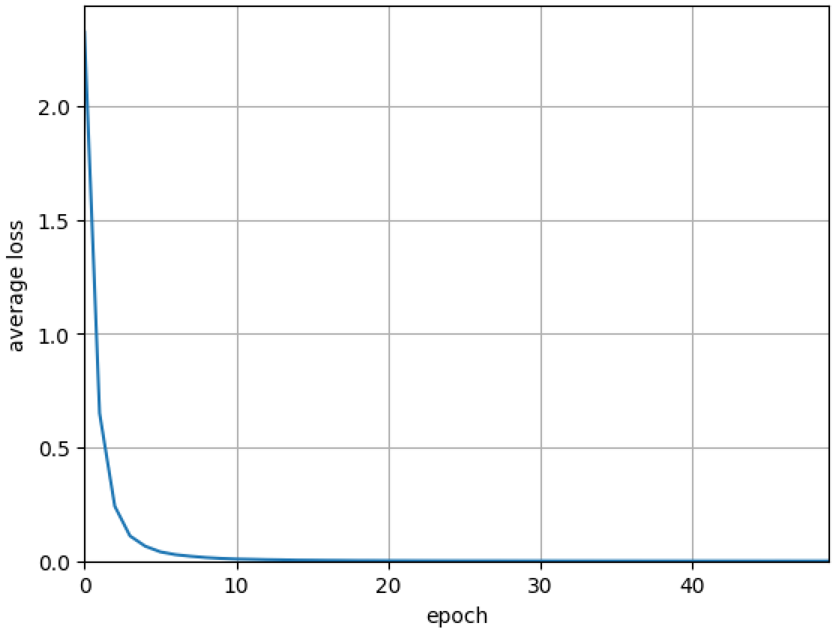
80
70
60
50
40
30
20
10
0
accuracy
0
10
20
30
40
epoch

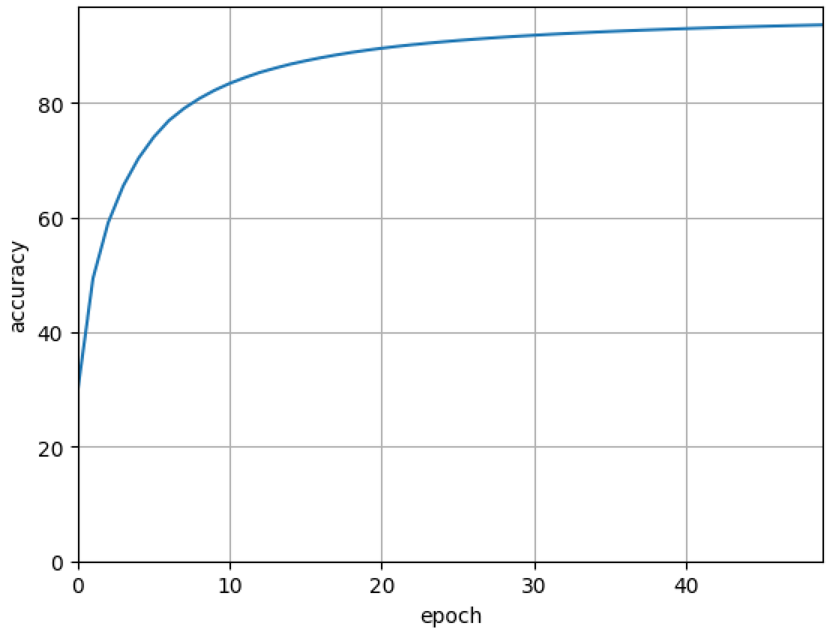


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
average loss
0
10
20
30
40
epoch

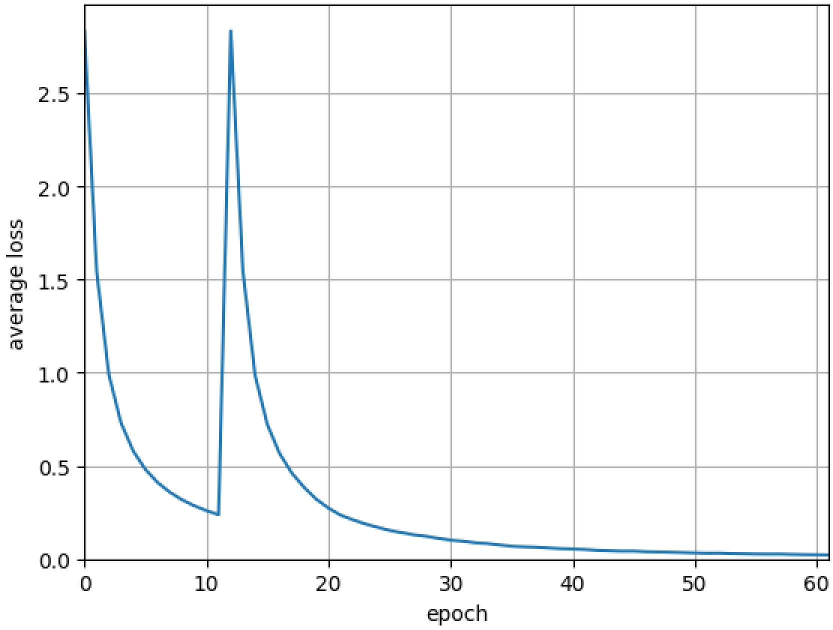
Q6.2

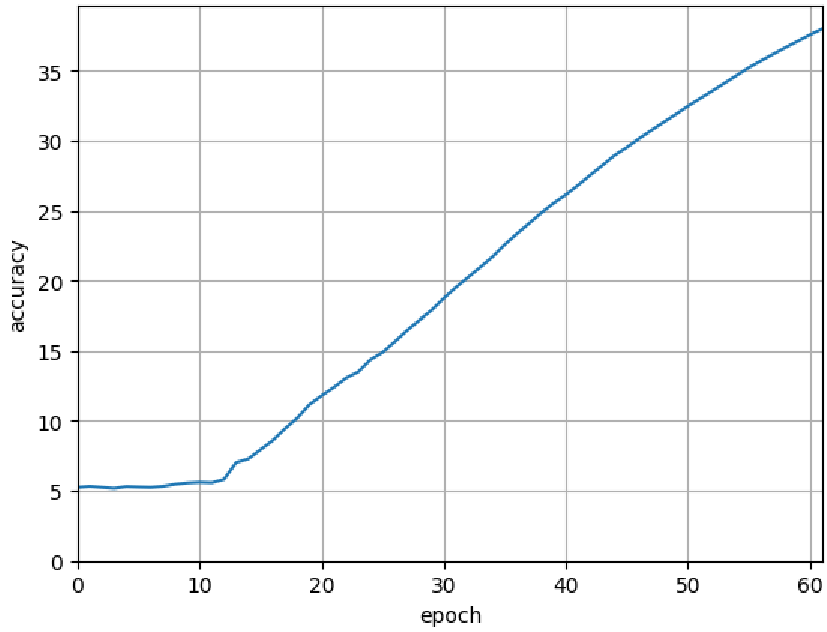
Finetuned squeezenet1_1





Own architecture





Comparison:

CNN converged less than 10 epochs. Has very high accuracy compared to own architecture. Upon more epochs, own architecture could have better accuracy.

Q6.3

Data extracted from video had higher accuracy than validation set. The reason being that the images in the validation folder are diverse compared to the carton in the video. If the carton was not recognised, the video would have lower accuracy.

But normally, when a model is tasked with real world scenario, it generally performs poorer because of lighting changes, background clutter, etc.

Some ways to make model robust:

1. Object tracking.
2. Usage of data augmentation techniques.
3. Using ensemble of architectures.

Collaborated with ddhrafan