CV HW4

Lifan Yu lifany

October 2023

Question 1

1.1

Softmax is invariant to translation

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

$$Softmax(x_i + c) = \frac{e^{x_i+c}}{\sum_{j=1}^{n} e^{x_j+c}} = \frac{e^{x_i}e^{c}}{\sum_{j=1}^{n} e^{x_j}e^{c}} = \frac{e^{c}e^{x_i}}{e^{c}\sum_{j=1}^{n} e^{x_j}}$$

When $c = 0$, the above becomes

$$\frac{1e^{x_i}}{1\sum_{j=1}^{n} e^{x_j}} = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

When $c \neq 0$, because $e^c > 0, \forall c \in R$, we cancel out e^c in both the denominator and numerator, and get the original expression as below. This case is satisfied when $c = -max(x_i)$ as well.

$$\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Therefore, before and after translation, the output of the softmax function is always $\boxed{\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}}$. Therefore the softmax function is translation invariant.

Why $\mathbf{c = -max(x_i)}$ is a good idea

By subtracting each element with the maximum, each $x_i \leq 0$ (Zero for the maximum itself). This can help the exponent not explode when x_i's are huge, as each e^{x_i} for $x_i <= 0$ is less than or equal to 1. $e^{x_i} = 1$ is achieved at the maximum x_i, e^{x_i} is between (0, 1) for all other x_i's smaller than the maximum. In this way we handle smaller exponent values and the computation is more efficient. This can allow **better numerical stability**.

1.2

We have $s_i = e^{x_i}, S = \sum s_i, softmax(x_i) = \frac{1}{S} s_i$

- Range of each element $softmax(x_i)$ is $\boxed{(0,\ 1)}$, because S is a sum of all s_i's, and each $s_i > 0$

- Softmax takes and arbitrary real valued vector $x \in R^d$ and turns it into $\boxed{\text{a probability value in } (0,\ 1)}$ for each category

- The first step converts all x_i into an exponent value, the second sums over all such exponent values, the third normalizes each e^{x_i} with the sum of all e^{x_i}'s, so the final output is always within (0, 1).

1.3

For the neural network, the input x is passed through a layer with weights w and bias term b and produces an output, let's denote it as x'. We have $x' = w * x + b$ for each layer. The final output from the multi-layer neural network (without a non-linear activation function) will be

$$y = w_k(.....w_2 * (w_1(w_0 * x + b_0) + b_1) + b_2.....) + b_k$$

That can be rearranged into

$$y = Wx + B$$

Which is essentially a linear regression

1.4

Sigmoid activation function is $\sigma(x) = \frac{1}{1+e^{-x}}$.

Its gradient, by chain rule, is

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

Because $\frac{1}{\sigma(x)} = 1 + e^{-x}$, the numerator $e^{-x} = \frac{1}{\sigma(x)} - 1$

The expression now becomes

$$\sigma'(x) = \sigma^2(x)(\frac{1}{\sigma(x)} - 1) = \sigma(x) - \sigma^2(x) = \sigma(x)(1 - \sigma(x))$$

We do not need to access x directly.

1.5

Given

$$\frac{\partial J}{\partial y} = \delta \in R^{k\times 1}, W \in R^{k\times d}, x \in R^{d\times 1}, b \in R^{k\times 1}$$

(1)

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} = \delta x^T =\in R^{k\times d}$$

(2)

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = W^T\delta \in R^{d\times 1}$$

(3)

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta \in R^{k\times 1}$$

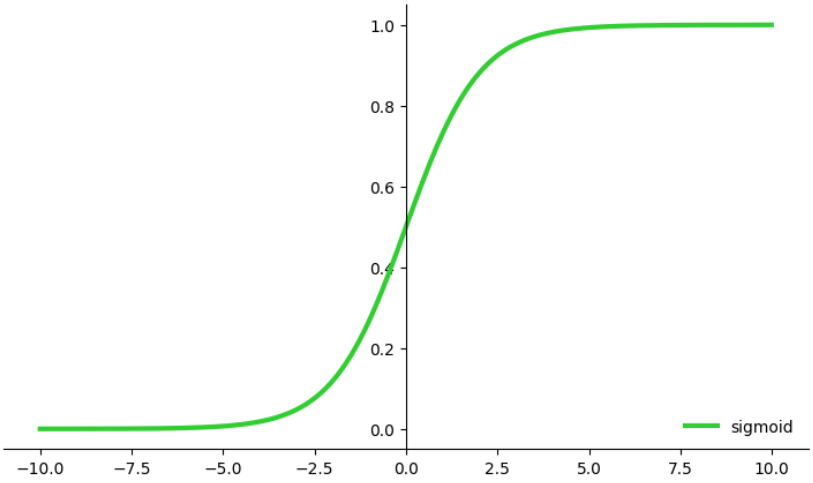
Each partial derivative's dimension is same as the variable on which the partial derivative is taken.

Math reference for matrix chain rule: https://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf
https://medium.com/@pdquant/all-the-backpropagation-derivatives-d5275f727f60

1.6

1

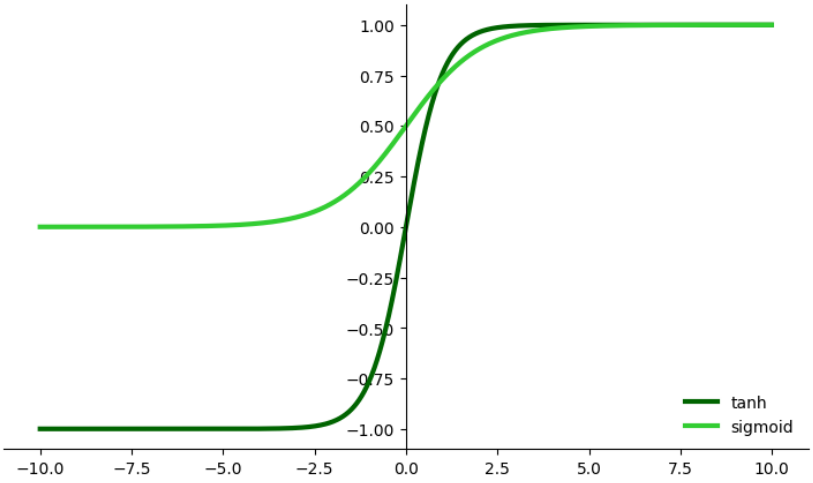
plotting 1.4 Sigmoid function



We see that when x is far from 0, the Sigmoid function has gradient close to zero. Only when the value is around zero do they correspond to a significant gradient. When values get passed through many layers, there may be increasingly bigger and smaller values produced. These values correspond to gradients very close to zero. That is why we have vanishing gradients.

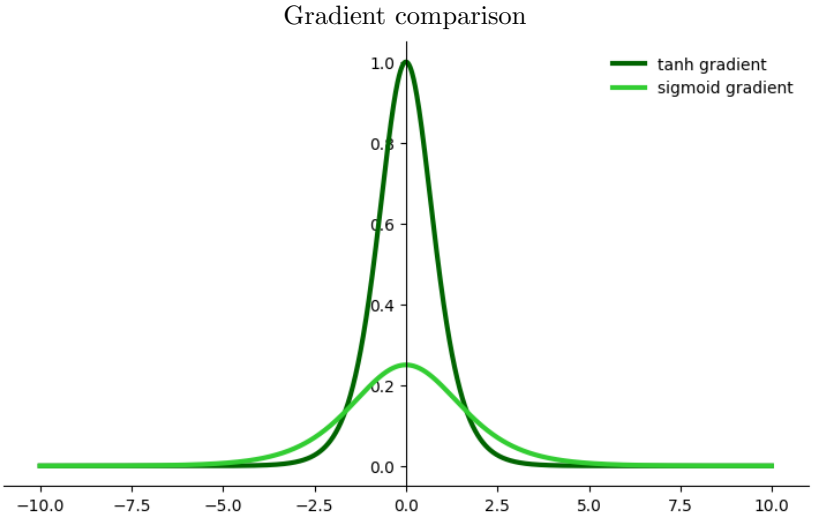
2

tanh(x) and Sigmoid(x) graph



tanh has output range (-1, 1), while Sigmoid has output range (0, 1). tanh's output value is centered around zero, which is helpful for faster convergence.

3



The gradiend of tanh is significantly higher than that of Sigmoid around x = 0. While we mostly have normalised input centered around zero, bigger gradient will result in bigger learning steps, and we have bigger updates on the network's weights. In such regions around x = 0, tanh(x) is slightly better than Sigmoid(x) for avoiding vanishing gradient problem because it has a greater gradient.

4

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

Question 2

2.1

2.1.1

Why it's not good to initialize a network with all zeros

In this case all the neurons will output the same zero result, and in the back propagation process, they will have the same partial derivatives. In each iteration, all the neurons in the same layer will have the same weights, this will lead to them learning the same parameter, preventing each neuron from learning different things. This approach will make the model lose all non-linearity, and the final result is no different than a linear model. Unless the data actually has a linear characteristic, the model is not robust enough.

2.1.2

```
def initialize_weights(in_size, out_size, params, name=""):
    W, b = None, None

    ##########################
    ##### your code here #####
    ##########################
    wmin, wmax = -np.sqrt(6)/(np.sqrt(in_size + out_size)), np.sqrt(6)/(np.sqrt(in_size + out_size))
    W = np.random.uniform(low=wmin, high=wmax, size=(in_size, out_size))
    b = np.zeros(out_size)
    print("W", W)

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

2.1.3

Why we initialize with random numbers

The initial parameters need to "break symmetry" between different units. If we initialize the neural network with the same same initial weights, in the back progagation process, all hidden units of the neural network will have the same gradient, then the model will constantly update them with the same value. All neurons in the same layer will evolve in the same way, and they will not learn different results. Additionally, this can also cause the model to get stuck at a local minima.

Why we scale the initialization depending on layer size

In order to avoid the gradient exploding and vanishing problem, we want the variance of the output to stay the same across different layers.

$$Var(out_L) = Var(out_{L-1})$$

Variance of the output in layer L can be written with the variance of its weights as

$$Var(out_L) = N_{L-1}Var(W_L)Var(out_{L-1})$$

Therefore, we need the variance of weights in layer L to be

$$Var(W_L) = \frac{1}{N_{L-1}}$$

where N_{L-1} is the number of neurons in layer L-1.

In back propagation, we need

$$Var(W_L) = \frac{1}{N_L}$$

Combining the two, we need

$$Var(W_L) = \frac{2}{N_{L-1} + N_L}$$

This is why we need to scale the randomly initialized weights depending on layer size to keep the variance of the weights as the value above.

2.2

2.2.1

```
############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    res = None

    ###########################
    ##### your code here #####
    ###########################
    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]

    ###########################
    ##### your code here #####
    ###########################
    pre_act = np.matmul(X, 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
```

2.2.2

```
############################## Q 2.2.2  ##############################
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):
    res = None

    ##########################
    ##### your code here #####
    ##########################
    # syntax reference: https://stackoverflow.com/questions/43290138/softmax-function-of-a-numpy-
                                                        array-by-row
    x_c = x - np.max(x, axis=1)[:, np.newaxis]
    res = np.exp(x_c) / np.sum(np.exp(x_c), axis=1)[:, np.newaxis]

    return res
```

2.2.3

```
############################## Q 2.2.3 ##############################
# compute total loss and accuracy
# y is size [examples,classes]
# probs is size [examples,classes]
def compute_loss_and_acc(y, probs):
    loss, acc = None, None

    ##########################
    ##### your code here #####
    ##########################
    logprobs = np.log(probs)
    loss_elements = y * logprobs
    loss = -np.sum(loss_elements)

    preds = (probs == probs.max(axis=1)[:,None]).astype(int)
    diffs = preds - y
    # count number of zero difference rows (where y == pred)
    correct_count = np.sum(~diffs.any(1))
    acc = correct_count / preds.shape[0]

    return loss, acc
```

2.3

nn.py

```
############################## Q 2.3 ##############################
# we give this to you
# because you proved it
# it's a function of post_act
def sigmoid_deriv(post_act):
    res = post_act * (1.0 - post_act)
    return res


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 #####
    ############################
    dJdy = activation_deriv(post_act)
    loss_dJdy = delta * dJdy
    grad_W = np.matmul(X.T, loss_dJdy)
    # -- take the per-class average of bias
    grad_b = np.sum(loss_dJdy, axis=0)
    grad_X = np.matmul(loss_dJdy, W.T)

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

2.4

nn.py

```
############################## Q 2.4 ##############################
# split x and y into random batches
# return a list of [(batch1_x,batch1_y)...]
def get_random_batches(x, y, batch_size):
    batches = []
    ##########################
    ##### your code here #####
    ##########################
    assert x.shape[0] == y.shape[0]
    np.random.seed(3)
    random_idx = np.arange(x.shape[0])
    np.random.shuffle(random_idx)
    x, y = x[random_idx, :], y[random_idx, :]
    num_batches = x.shape[0] / batch_size
    x_split = np.split(x, num_batches)
    y_split = np.split(y, num_batches)
    batches = list(zip(x_split, y_split))
    return batches
```

run_q2.py

```
# Q 2.4
batches = get_random_batches(x, y, 5)
# print batch sizes
print("\nQ2.4 print batch sizes")
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)
# WRITE A TRAINING LOOP HERE
print("\ntraining loop with default settings, you should get loss < 35 and accuracy > 75%")
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    total_acc = 0
    for xb, yb in batches:
        ##########################
        ##### your code here #####
        ##########################
        # forward
        h1 = forward(xb, params, "layer1", sigmoid)
        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
        total_acc += acc

        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, "output", linear_deriv)
        grad_xb = backwards(delta2, params, "layer1", sigmoid_deriv)

        # apply gradient
        # gradients should be summed over batch samples
        for k, v in sorted(list(params.items())):
            if "grad" in k:
                name = k.split("_")[1]
                #print(name, v.shape, params[name].shape)
                params[name] = params[name] - learning_rate*v
    avg_acc = total_acc / len(batches)
    if itr % 100 == 0:
        print(
            "itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(
                itr, total_loss, avg_acc
            )
        )
```

2.5

total relative error 9.80e-06 which is lower than 1e-4

run_q2.py

```
# Q 2.5 should be implemented in this file
print("\nQ2.5")
# 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

    # 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 #####
    ############################
    # code reference: https://towardsdatascience.com/coding-neural-network-gradient-checking-
                                                    5222544ccc64    print("k dim", k, params[k].
                                                    shape[1])

    this_shape = params[k].shape
    if len(this_shape) == 2:
        for i in range(this_shape[0]):
            for j in range(this_shape[1]):
                params[k][i][j] = params[k][i][j] + eps
                h1 = forward(x, params, "layer1")
                probs1 = forward(h1, params, "output", softmax)
                loss1, acc1 = compute_loss_and_acc(y, probs1)

                params[k][i][j] = params[k][i][j] - 2*eps
                h1 = forward(x, params, "layer1")
                probs2 = forward(h1, params, "output", softmax)
                loss2, acc2 = compute_loss_and_acc(y, probs2)

                params[k][i][j] = params[k][i][j] + eps

                central_diff = (loss1 - loss2) / (2*eps)
                params["grad_" + k][i][j] = central_diff
    else:
        for i in range(this_shape[0]):
            params[k][i] = params[k][i] + eps
            h1 = forward(x, params, "layer1")
            probs1 = forward(h1, params, "output", softmax)
            loss1, acc1 = compute_loss_and_acc(y, probs1)

            params[k][i] = params[k][i] - 2*eps
            h1 = forward(x, params, "layer1")
            probs2 = forward(h1, params, "output", softmax)
```

```
            loss2, acc2 = compute_loss_and_acc(y, probs2)

            params[k][i] = params[k][i] + eps

            central_diff = (loss1 - loss2) / (2*eps)
            params["grad_" + k][i] = central_diff


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))
```

run_q2.py complete code

```
import numpy as np

# you should write your functions in nn.py
from nn import *
from util import *


# fake data
# feel free to plot it in 2D
# what do you think these 4 classes are?
g0 = np.random.multivariate_normal([3.6, 40], [[0.05, 0], [0, 10]], 10)
g1 = np.random.multivariate_normal([3.9, 10], [[0.01, 0], [0, 5]], 10)
g2 = np.random.multivariate_normal([3.4, 30], [[0.25, 0], [0, 5]], 10)
g3 = np.random.multivariate_normal([2.0, 10], [[0.5, 0], [0, 10]], 10)
x = np.vstack([g0, g1, g2, g3])
# we will do XW + B
# that implies that the data is N x D

# create labels
y_idx = np.array(
    [0 for _ in range(10)]
    + [1 for _ in range(10)]
    + [2 for _ in range(10)]
    + [3 for _ in range(10)]
)
# turn to one_hot
y = np.zeros((y_idx.shape[0], y_idx.max() + 1))
y[np.arange(y_idx.shape[0]), y_idx] = 1

# parameters in a dictionary
params = {}

# Q 2.1
# initialize a layer
initialize_weights(2, 25, params, "layer1")
initialize_weights(25, 4, params, "output")
assert params["Wlayer1"].shape == (2, 25)
assert params["blayer1"].shape == (25,)

# expect 0, [0.05 to 0.12]
print("{}, {:.2f}".format(params["blayer1"].mean(), params["Wlayer1"].std() ** 2))
print("{}, {:.2f}".format(params["boutput"].mean(), params["Woutput"].std() ** 2))

# Q 2.2.1
# implement sigmoid
test = sigmoid(np.array([-1000, 1000]))
print("\nQ2.2.1 should be zero and one\t", test.min(), test.max())
# implement forward
h1 = forward(x, params, "layer1")

# Q 2.2.2
# implement softmax
probs = forward(h1, params, "output", softmax)
# make sure you understand these values!
# positive, ~1, ~1, (40,4)
print("\nQ2.2.2 positive, ~1, ~1, (40,4)")
print(probs.min(), min(probs.sum(1)), max(probs.sum(1)), probs.shape)

# Q 2.2.3
# implement compute_loss_and_acc
loss, acc = compute_loss_and_acc(y, probs)
# should be around -np.log(0.25)*40 [~55], and 0.25
# if it is not, check softmax!
print("\nQ2.2.3 should be around -np.log(0.25)*40 [~55], and 0.25")
print("{}, {:.2f}".format(loss, acc))

# here we cheat for you
# the derivative of cross-entropy(softmax(x)) is probs - 1[correct actions]
delta1 = probs - y

# we already did derivative through softmax
# so we pass in a linear_deriv, which is just a vector of ones
```

```
# to make this a no-op
delta2 = backwards(delta1, params, "output", linear_deriv)
# Implement backwards!
backwards(delta2, params, "layer1", sigmoid_deriv)

# W and b should match their gradients sizes
print("\nW and b should match their gradients sizes")
for k, v in sorted(list(params.items())):
    if "grad" in k:
        name = k.split("_")[1]
        print(name, v.shape, params[name].shape)

# Q 2.4
batches = get_random_batches(x, y, 5)
# print batch sizes
print("\nQ2.4 print batch sizes")
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)


# WRITE A TRAINING LOOP HERE
print("\ntraining loop with default settings, you should get loss < 35 and accuracy > 75%")
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    total_acc = 0
    for xb, yb in batches:
        ##########################
        ##### your code here #####
        ##########################

        # forward
        h1 = forward(xb, params, "layer1", sigmoid)
        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
        total_acc += acc

        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, "output", linear_deriv)
        grad_xb = backwards(delta2, params, "layer1", sigmoid_deriv)

        # apply gradient
        # gradients should be summed over batch samples
        for k, v in sorted(list(params.items())):
            if "grad" in k:
                name = k.split("_")[1]
                #print(name, v.shape, params[name].shape)
                params[name] = params[name] - learning_rate*v
    avg_acc = total_acc / len(batches)

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


# Q 2.5 should be implemented in this file
print("\nQ2.5")
# 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

    # 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 #####
    ############################
    # code reference: https://towardsdatascience.com/coding-neural-network-gradient-checking-
                                                    5222544ccc64    print("k dim", k, params[k].
                                                    shape[1])

    this_shape = params[k].shape
    if len(this_shape) == 2:
        for i in range(this_shape[0]):
            for j in range(this_shape[1]):
                params[k][i][j] = params[k][i][j] + eps
                h1 = forward(x, params, "layer1")
                probs1 = forward(h1, params, "output", softmax)
                loss1, acc1 = compute_loss_and_acc(y, probs1)

                params[k][i][j] = params[k][i][j] - 2*eps
                h1 = forward(x, params, "layer1")
                probs2 = forward(h1, params, "output", softmax)
                loss2, acc2 = compute_loss_and_acc(y, probs2)

                params[k][i][j] = params[k][i][j] + eps

                central_diff = (loss1 - loss2) / (2*eps)
                params["grad_" + k][i][j] = central_diff
    else:
        for i in range(this_shape[0]):
            params[k][i] = params[k][i] + eps
            h1 = forward(x, params, "layer1")
            probs1 = forward(h1, params, "output", softmax)
            loss1, acc1 = compute_loss_and_acc(y, probs1)

            params[k][i] = params[k][i] - 2*eps
            h1 = forward(x, params, "layer1")
            probs2 = forward(h1, params, "output", softmax)
            loss2, acc2 = compute_loss_and_acc(y, probs2)

            params[k][i] = params[k][i] + eps

            central_diff = (loss1 - loss2) / (2*eps)
            params["grad_" + k][i] = central_diff


total_error = 0
for k in params.keys():
    if "grad_" in k:
        # relative error
```

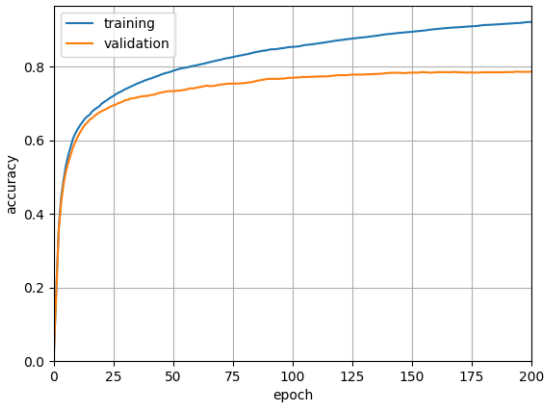
```
        err = np.abs(params[k] - params_orig[k]) / np.maximum(
            np.abs(params[k]), np.abs(params_orig[k])
        )
        err = err.sum()
        print("{} {:.2e}".format(k, err))
        total_error += err
# should be less than 1e-4
print("total {:.2e}".format(total_error))
```

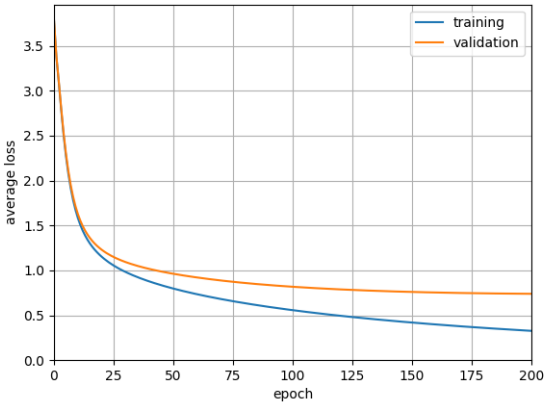
Question 3

3.1

```
max_iters = 200
# pick a batch size, learning rate
batch_size = 64
learning_rate = 1e-3
hidden_size = 64
```

```
Validation accuracy:  0.7861111111111111
Test accuracy:  0.79
```

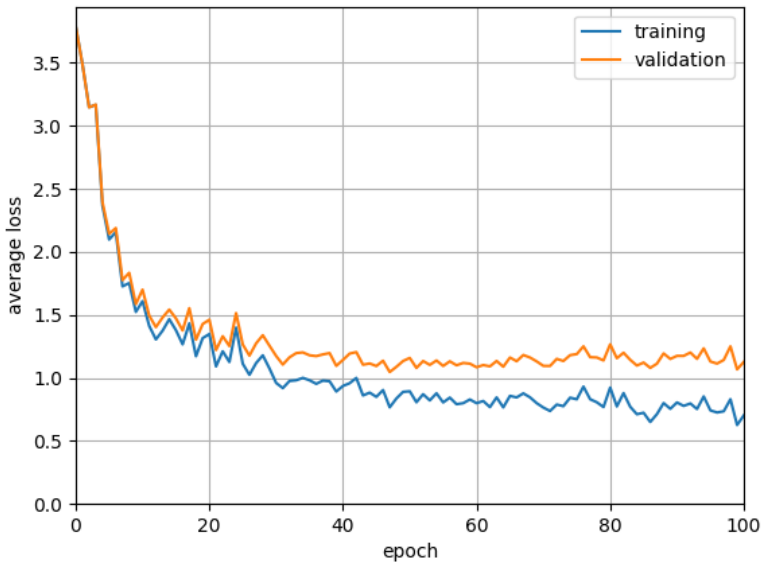


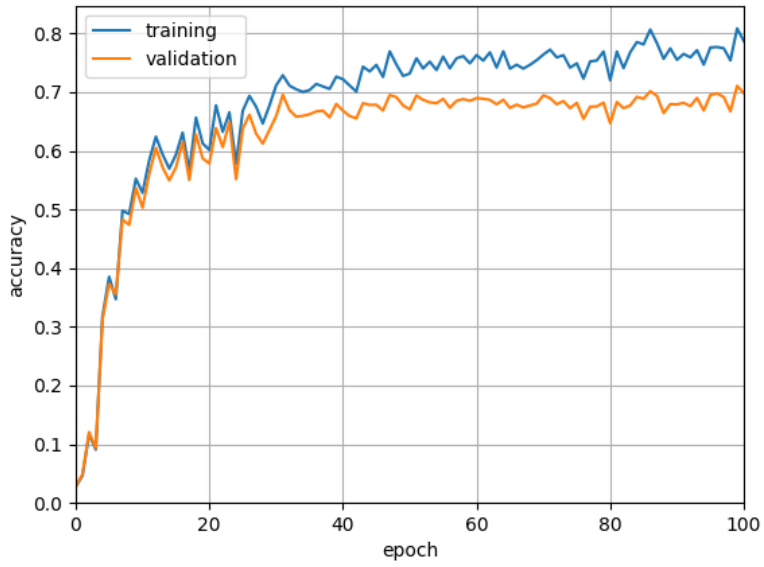


3.2

10 times best learning rate 1e-2

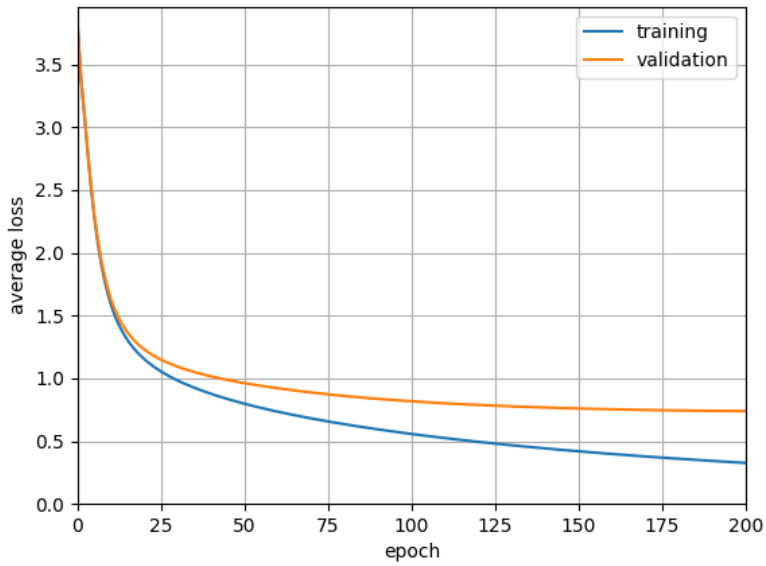
```
Validation accuracy:  0.6983333333333334
Test accuracy:  0.7094444444444444
```

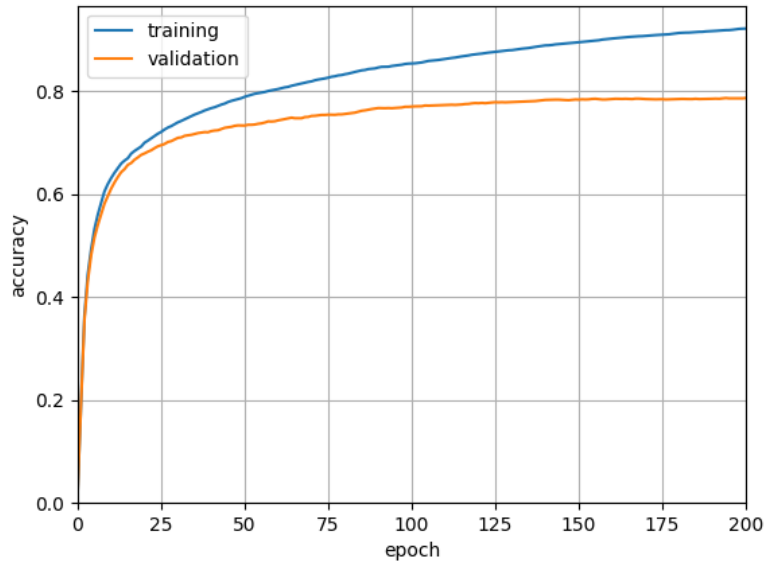




Using best learning rate 1e-3

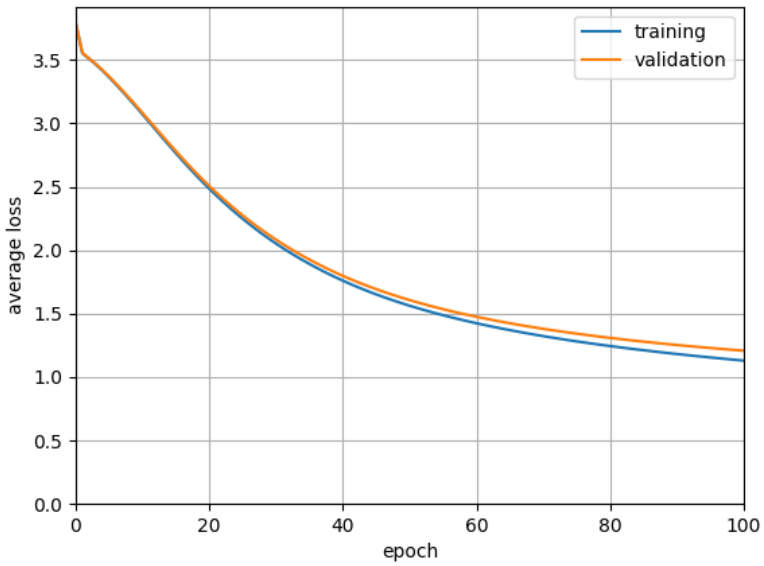
```
Validation accuracy:  0.7861111111111111
Test accuracy:  0.79
```

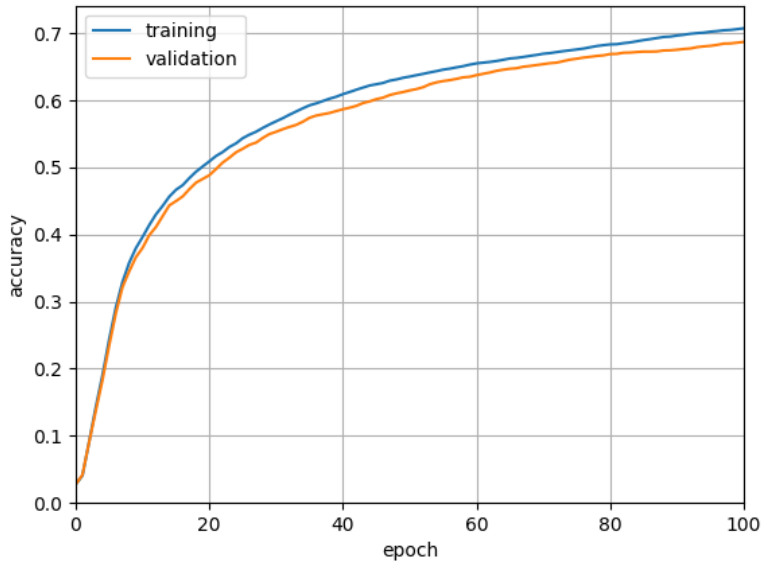




One tenth best learning rate 1e-4

```
Validation accuracy:  0.6869444444444445
Test accuracy:  0.6866666666666666
```





How learning rate affect the training

If the learning rate is too small, the parameter updates are small, and it can take a long time to train until convergence when the parameters have the optimal values. Observe the plots of 0.1*best learning rate. The loss is still on its way of decreasing when the maximum epoch 100 was reached.

If the learning rate is too large, the parameter updates are very large, and during back prapagation we might step over the minima of the gradient, and never arrive at a minima. Such big changes in the parameter values will result in unstable model performance, which is shown in the fluctuations in the loss and accuracy plots.

Final accuracy of best network on the test set

Test accuracy: 0.79

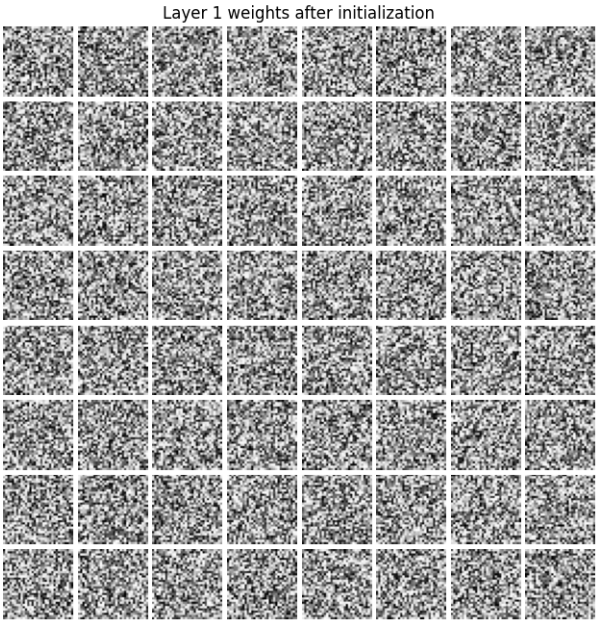
Using the following parameters

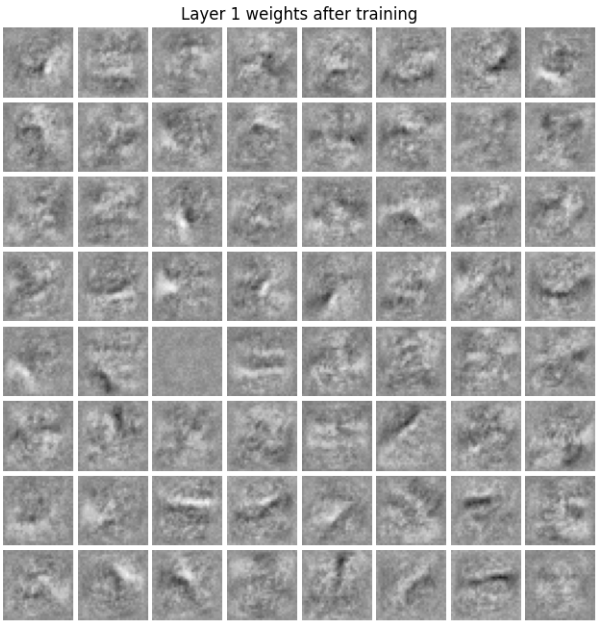
```
max_iters = 100
batch_size = 64
learning_rate = 2e-4
hidden_size = 64
```

3.3

The initial visualization shows randomly initialized weight images. There are no significannt patterns.

After training, the weight images in each unit shows patterns of certain parts of the image to put high weights on. They started responding differently to images representing different letters and numbers.

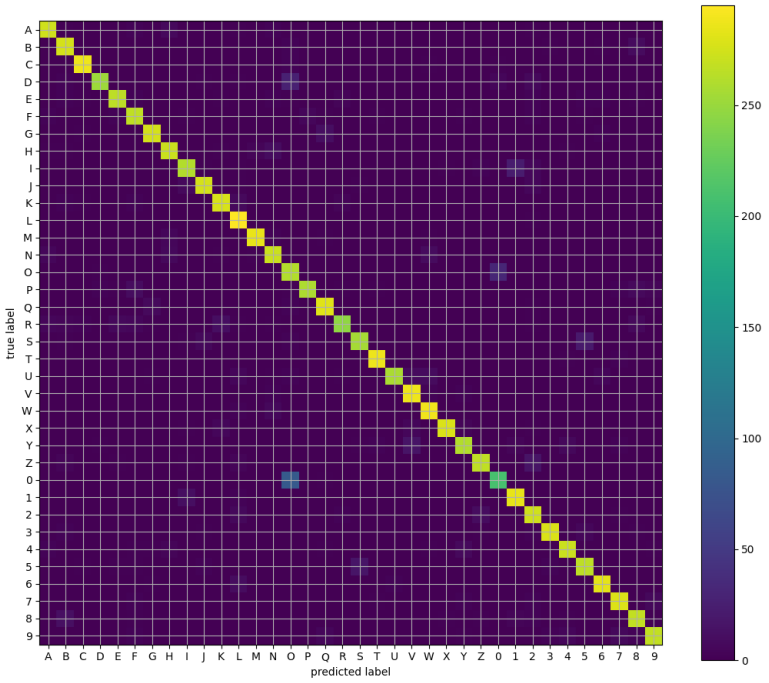




3.4

The confusion matrix showed that the network mostly correctly classifies the letters, but most commonly confused letter / number pairs include:

letter O, number 0
lettern D, letter O
letter S, number 5
letter I, number 1
letter Z, number 2
letter R, letter K




```
# Q3.4
confusion_matrix = np.zeros((train_y.shape[1], train_y.shape[1]))

# compute confusion matrix
############################
##### your code here #####
############################
def confusion_mat(true, pred):
  k = true.shape[1]
  result = np.zeros((k, k))
  for i in range(true.shape[0]):
    itrue = true[i].argmax()
    ipred = pred[i].argmax()
    result[itrue][ipred] += 1
  return result

train_preds = (train_probs == train_probs.max(axis=1)[:,None]).astype(int)
confusion_matrix = confusion_mat(train_y, train_preds)
```

Question 4

4.1

Two of the assumptions that the method makes:

(1) Assuming that the words are written line by line. If words are written in a circle or letters are oriented in different angles, this method might fail to correctly predict and sort the letters.

(2) Assuming that the image contains only letters. if there are non-letter patterns that shape like letters, the detection can fail.

For example CMU's logo:

(3) Assuming that the letters are separate from each other, meaning, that they do not overlap, connect, or appear messy. In these cases the detection might also fail.

For example the image made by myself:

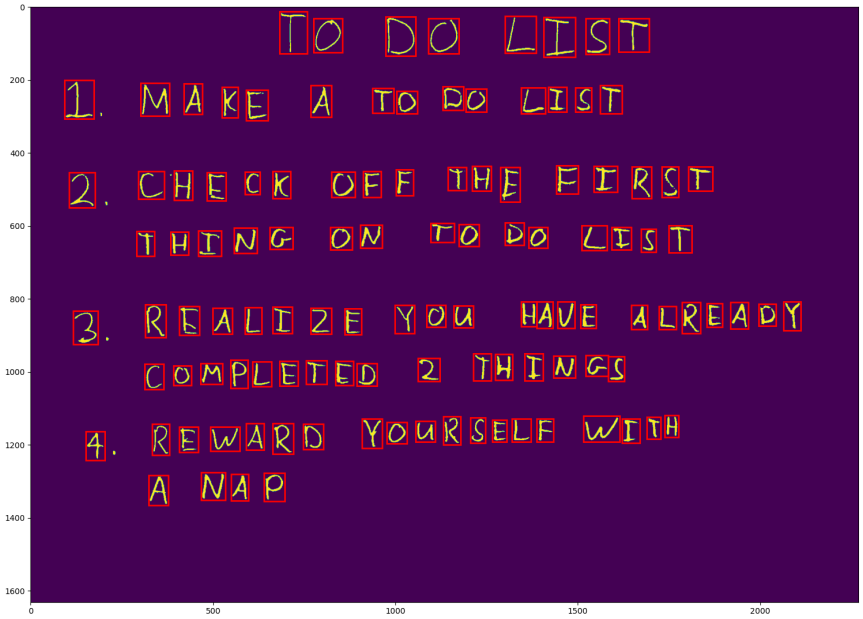
4.2

Clustering and sorting code is included in q4.4. This code just finds bounding boxes.

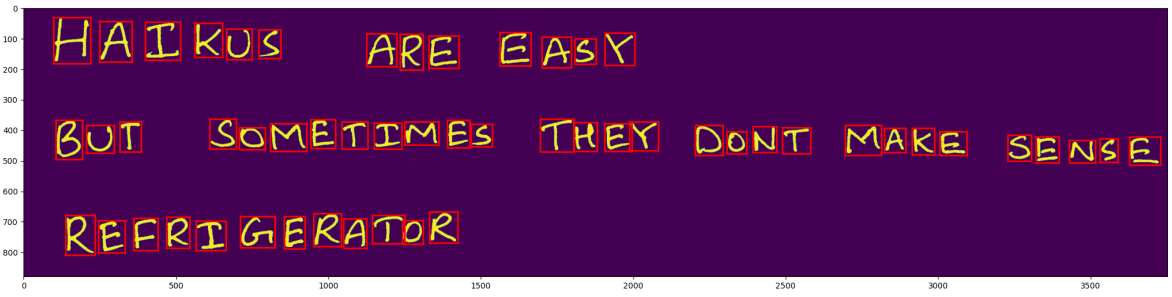
q4.py

```
import numpy as np

import skimage
import skimage.measure
import skimage.color
import skimage.restoration
import skimage.filters
import skimage.morphology
import skimage.segmentation

from skimage.io import imsave
import matplotlib.pyplot as plt
from skimage import data
from skimage.filters import threshold_otsu
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops
from skimage.morphology import closing, square, area_opening, binary_closing, dilation, erosion
from skimage.color import label2rgb


# takes a color image
# returns a list of bounding boxes and black_and_white image
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 #####
    ############################
    # code reference: https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_label.html
    sigma_est = skimage.restoration.estimate_sigma(image, channel_axis=-1, average_sigmas=True)
    print("image noise", sigma_est)
    denoised_image = skimage.restoration.denoise_bilateral(image, channel_axis=-1)
    grey_image = skimage.color.rgb2gray(denoised_image)

    thresh = threshold_otsu(grey_image)
    bw = binary_closing(grey_image < thresh, square(3))
    # dilation to prevent strokes marked as a letter
    d = np.ones((9,9)) # -row -column
    dilated_bw = dilation(bw, d)
    # remove artifacts connected to image border
    cleared = clear_border(dilated_bw)
    # label image regions
    label_image = label(cleared)

    for region in skimage.measure.regionprops(label_image):
        # take regions with large enough areas
        if region.area >=350:
            # draw rectangle
            minr, minc, maxr, maxc = region.bbox
            bboxes.append([minr, minc, maxr, maxc])

    return bboxes, grey_image
```

4.3


TO DO LIST
1. MAKE A TO DO LIST
2. CHECK OFF THE FIRST THING ON TO DO LIST
3. REALIZE YOU HAVE ALREADY COMPLETED 2 THINGS
4. REWARD YOURSELF WITH A NAP


A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 2 3 4 5 6 7 8 9 0


HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REFRIGERATOR


DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING

4.4

01_list.jpg

Accuracy: 0.861

TO DO LIST
L HAKE A TO DO LZST
2 2HECK OFF THE FIRST
THING 0N T0 DO LIIT
3 REALI2B YOU HQVE ALR6ADY
COAPLETZD 2 THINGS
4 RRWARD YOHRSELF W8TH
A NAP

s

02_letters.jpg

Accuracy: 0.806

A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
3 Z 3 G S G 7 8 D Q

03_haiku.jpg

Aaccuracy: 0.833

HAIKUS ARE EASX
BUT SQMETIMBS TREX DONT MAK2 SZNSE
REFRIGBRATDR

04_deep.jpg

Accuracy: 0.927

DEEP LEARMING
DFBPER LEARNING
DEEPEST LEARNING

run_q4.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches

import skimage
import skimage.measure
import skimage.color
import skimage.restoration
import skimage.io
import skimage.filters
import skimage.morphology
import skimage.segmentation
from skimage.transform import resize

from nn import *
from q4 import *

# do not include any more libraries here!
# no opencv, no sklearn, etc!
import warnings

# --
hidden_size = 64
new_max, new_min = 1, 0 # for enhancing image contrast
t1 = '
                                                    TODOLIST1MAKEATODOLIST2CHECKOFFTHEFIRSTTHINGONTODOLIST3RE
                                                    '
t2 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890'
t3 = 'HAIKUSAREEASYBUTSOMETIMESTHEYDONTMAKESENSEREFRIGERATOR'
t4 = 'DEEPLEARNINGDEEPERLEARNINGDEEPESTLEARNING'
# --

warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=UserWarning)

for img in os.listdir("../images"):
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join("../images", img)))
    bboxes, bw = findLetters(im1)

    plt.imshow(bw)
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle(
            (minc, minr),
            maxc - minc,
            maxr - minr,
            fill=False,
            edgecolor="red",
            linewidth=2,
        )
        plt.gca().add_patch(rect)
    plt.show()
    # find the rows using..RANSAC, counting, clustering, etc.
    ############################
    ##### your code here #####
    ############################
    # list of center points corresponding to the returned bboxes
    points = np.array([(bbox[0] + bbox[2])/2 for bbox in bboxes])
    points_h = [(i, (bboxes[i][1] + bboxes[i][3])/2) for i in range(len(bboxes))]
    N = len(points)
    # clustering parameter
    eps = 50
    curr_point = points[0]
    curr_cluster = [points_h[0]]
    lines = []
    for i in range(1, N):
        point = points[i]
        if point <= curr_point + eps:
            curr_cluster.append(points_h[i])
        else:
            lines.append(curr_cluster)
```

```
            curr_cluster = [points_h[i]]
        curr_point = point
    lines.append(curr_cluster)
    sorted_lines = []
    for i in range(len(lines)):
        line = lines[i]
        line.sort(key = lambda x: x[1])
        sorted_lines.append([item[0] for item in line])

    # crop the bounding boxes
    # note.. before you flatten, transpose the image (that's how the dataset is!)
    # consider doing a square crop, and even using np.pad() to get your images looking more like the
                                                        dataset
    ############################
    ##### your code here #####
    ############################
    # preprocess the mini images and populate the X matrix for model input
    myX = np.zeros((N, 1024))
    xidx = 0
    for sl in sorted_lines:
        for idx in sl:
            # retrieve the mini images using bounding boxes and resize
            bbox = bboxes[idx]
            y1, x1, y2, x2 = bbox[0]-5, bbox[1]-5, bbox[2]+5, bbox[3]+5
            im = bw[y1:y2, x1:x2]
            im = np.pad(im, (int(0.14*im.shape[0]), int(0.145*im.shape[1])), 'maximum')
            d = np.ones((int(11*im.shape[0]/im.shape[1]), int(10*im.shape[1]/im.shape[0]))) # for
                                                            letter dilation
            dilated_im = erosion(im, d)
            resized_im = resize(dilated_im, (32, 32))
            resized_im = resized_im.T.reshape(1, 1024)
            # increase each mini image's contrast
            minimum, maximum = np.min(resized_im), np.max(resized_im)
            m = (new_max - new_min) / (maximum - minimum)
            b = new_min - m * minimum
            resized_im = m * resized_im + b
            # populate the X matrix
            myX[xidx][:] = resized_im
            xidx += 1

    # load the weights
    # run the crops through your neural network and print them out
    import pickle
    import string

    letters = np.array(
        [_ for _ in string.ascii_uppercase[:26]] + [str(_) for _ in range(10)]
    )
    params = pickle.load(open("q3_weights.pickle", "rb"))
    ############################
    ##### your code here #####
    ############################
    # Create y groundtruth data
    indices = [i for i in range(36)]
    D = dict(zip(letters, indices))
    D1 = dict(zip(indices, letters))
    if "01" in img:
        myt = t1
    elif "02" in img:
        myt = t2
    elif "03" in img:
        myt = t3
    elif "04" in img:
        myt = t4
    myY = np.zeros((len(myt), 36))
    for i in range(len(myt)):
        myY[i][D[myt[i]]] = 1

    # Model predictions
    h1 = forward(myX, params, "layer1")
    probs = forward(h1, params, "output", softmax)
    loss, acc = compute_loss_and_acc(myY, probs)
```

```
    # The raw extracted text
    txt = "".join([D1[probs[idx].argmax()] for idx in range(myY.shape[0])])
    print("img name: ", img, " loss: ", loss, " accuracy: ", acc)

    # Print the extracted text in rows
    txt_rows = ''
    a, b = 0, 0
    for line in sorted_lines:
        b += len(line)
        txt_rows += txt[a:b] + "\n"
        a = b
    print(txt_rows)
```

Question 5

5.1

5.1.1

```
# Q5.1 & Q5.2
# initialize layers here
##########################
##### your code here #####
##########################
initialize_weights(train_x.shape[1], hidden_size, params, "layer1")
initialize_weights(hidden_size, hidden_size, params, "layer2")
initialize_weights(hidden_size, hidden_size, params, "layer3")
initialize_weights(hidden_size, train_x.shape[1], params, "output")

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_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

        ##########################
        ##### your code here #####
        ##########################
        # forward pass
        h1 = forward(xb, params, "layer1", relu)
        h2 = forward(h1, params, "layer2", relu)
        h3 = forward(h2, params, "layer3", relu)
        output_img = forward(h3, params, "output", sigmoid)

        # loss
        loss = np.sum((output_img - xb)**2)
        total_loss += loss

        # backward
        delta1 = 2*output_img - 2*xb
        delta2 = backwards(delta1, params, "output", sigmoid_deriv)
        delta3 = backwards(delta2, params, "layer3", relu_deriv)
        delta4 = backwards(delta3, params, "layer2", relu_deriv)
        grad_xb = backwards(delta4, params, "layer1", relu_deriv)

        # apply gradient, remember to update momentum as well
        for k, v in sorted(list(params.items())):
            if "grad" in k:
                name = k.split("_")[1]
                if "W" in name:
                    Mw = params["m_" + name]
                    Mw = 0.9 * Mw - learning_rate * v
                    params[name] = params[name] + Mw
                else:
                    params[name] = params[name] - learning_rate * v

    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
```

5.1.2

Modifications to nn.py (last line)

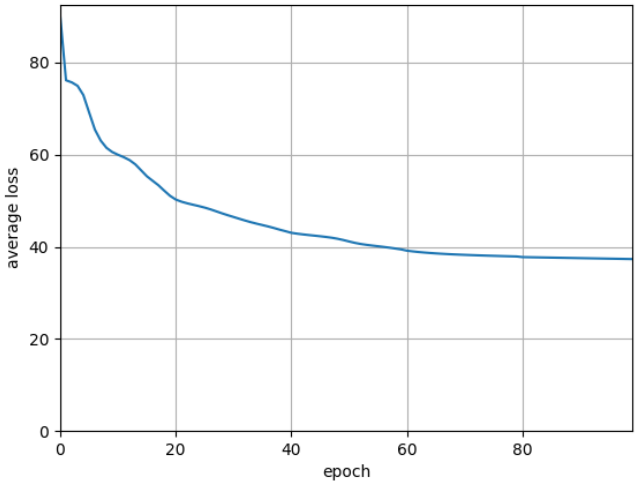
```
def initialize_weights(in_size, out_size, params, name=""):
    W, b = None, None

    ##########################
    ##### your code here #####
    ##########################
    wmin, wmax = -np.sqrt(6)/(np.sqrt(in_size + out_size)), np.sqrt(6)/(np.sqrt(in_size + out_size))
    W = np.random.uniform(low=wmin, high=wmax, size=(in_size, out_size))
    b = np.zeros(out_size)

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

    # -- for question 5
    params["m_" + name] = np.zeros((in_size, out_size))
```

run_q5.py question 5.1 complete code

```
import numpy as np
import scipy.io
from nn import *
from collections import Counter
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import string
import pickle

train_data = scipy.io.loadmat('../data/nist36_train.mat')
valid_data = scipy.io.loadmat('../data/nist36_valid.mat')

# we don't need labels now!
train_x = train_data['train_data']
valid_x = valid_data['valid_data']

max_iters = 100
# pick a batch size, learning rate
# batch_size = 128
#learning_rate =  1e-2
batch_size = 36
learning_rate =  3e-5
hidden_size = 32
lr_rate = 20
batches = get_random_batches(train_x,np.ones((train_x.shape[0],1)),batch_size)
batch_num = len(batches)

params = Counter()

# Q5.1 & Q5.2
# initialize layers here
##########################
##### your code here #####
##########################
initialize_weights(train_x.shape[1], hidden_size, params, "layer1")
initialize_weights(hidden_size, hidden_size, params, "layer2")
initialize_weights(hidden_size, hidden_size, params, "layer3")
initialize_weights(hidden_size, train_x.shape[1], params, "output")

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_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

        ##########################
        ##### your code here #####
        ##########################
        # forward pass
        h1 = forward(xb, params, "layer1", relu)
        h2 = forward(h1, params, "layer2", relu)
        h3 = forward(h2, params, "layer3", relu)
        output_img = forward(h3, params, "output", sigmoid)

        # loss
        loss = np.sum((output_img - xb)**2)
        total_loss += loss

        # backward
        delta1 = 2*output_img - 2*xb
        delta2 = backwards(delta1, params, "output", sigmoid_deriv)
        delta3 = backwards(delta2, params, "layer3", relu_deriv)
        delta4 = backwards(delta3, params, "layer2", relu_deriv)
        grad_xb = backwards(delta4, params, "layer1", relu_deriv)

        # apply gradient, remember to update momentum as well
        for k, v in sorted(list(params.items())):
            if "grad" in k:
                name = k.split("_")[1]
                if "W" in name:
                    Mw = params["m_" + name]
                    Mw = 0.9 * Mw - learning_rate * v
                    params[name] = params[name] + Mw
                else:
                    params[name] = params[name] - learning_rate * v

    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
# --
# save the final network
saved_params = {k: v for k, v in params.items() if "_" not in k}
with open("q5_weights.pickle", "wb") as handle:
    pickle.dump(saved_params, handle, protocol=pickle.HIGHEST_PROTOCOL)
# --

# plot loss curve
plt.plot(range(len(losses)), losses)
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(losses)-1)
plt.ylim(0, None)
plt.grid()
plt.show()
```

5.2



The loss went down during training and converged around 7.5e6, but it is still rather high.

This means the network learned to copy the input approximately, and it has learned to represent data with a limited number of hidden nodes. However, this compressed representation of the original image still has a significant difference from the original.

```
# plot loss curve
plt.plot(range(len(losses)), losses)
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(losses)-1)
plt.ylim(0, None)
plt.grid()
plt.show()
```

5.3

5.3.1

Difference:

(1) The reconstructed validation images are more **blurry** than the original ones. The edges of the letters are less sharp.

(2) The shapes of the reconstructed images also **slightly distorted** comparing to their original ones. The shape also tend to be more dilated.

(3) The original images have rather high contrast, but the reconstructed ones have **less contrast** in certain parts of the letters.

```
# Q5.3.1
# choose 5 labels (change if you want)
visualize_labels = ["A", "B", "C", "1", "2"]

# get 2 validation images from each label to visualize
visualize_x = np.zeros((2*len(visualize_labels), valid_x.shape[1]))
for i, label in enumerate(visualize_labels):
    idx = 26+int(label) if label.isnumeric() else string.ascii_lowercase.index(label.lower())
    choices = np.random.choice(np.arange(100*idx, 100*(idx+1)), 2, replace=False)
    visualize_x[2*i:2*i+2] = valid_x[choices]

# run visualize_x through your network
# name the output reconstructed_x
############################
##### your code here #####
############################
params = pickle.load(open("q5_weights.pickle", "rb"))
h1 = forward(visualize_x, params, "layer1", relu)
h2 = forward(h1, params, "layer2", relu)
h3 = forward(h2, params, "layer3", relu)
reconstructed_x = forward(h3, params, "output", sigmoid)

# visualize
fig = plt.figure()
plt.axis("off")
grid = ImageGrid(fig, 111, nrows_ncols=(len(visualize_labels), 4), axes_pad=0.05)
for i, ax in enumerate(grid):
    if i % 2 == 0:
        ax.imshow(visualize_x[i//2].reshape((32, 32)).T, cmap="Greys")
    else:
        ax.imshow(reconstructed_x[i//2].reshape((32, 32)).T, cmap="Greys")
    ax.set_axis_off()
plt.show()
```

5.3.2

Output:

```
PSNR:  14.922
```

Code:

```
# Q5.3.2
from skimage.metrics import peak_signal_noise_ratio
# evaluate PSNR
###########################
##### your code here #####
###########################
res = 0

params = pickle.load(open("q5_weights.pickle", "rb"))
h1 = forward(valid_x, params, "layer1", relu)
h2 = forward(h1, params, "layer2", relu)
h3 = forward(h2, params, "layer3", relu)
reconstructed_valid_x = forward(h3, params, "output", sigmoid)

n = valid_x.shape[0]
for i in range(n):
    im_true = valid_x[i]
    im_test = reconstructed_valid_x[i]
    res += peak_signal_noise_ratio(im_true, im_test)
res = res / n
print("PSNR: ", res)
```

Question 6

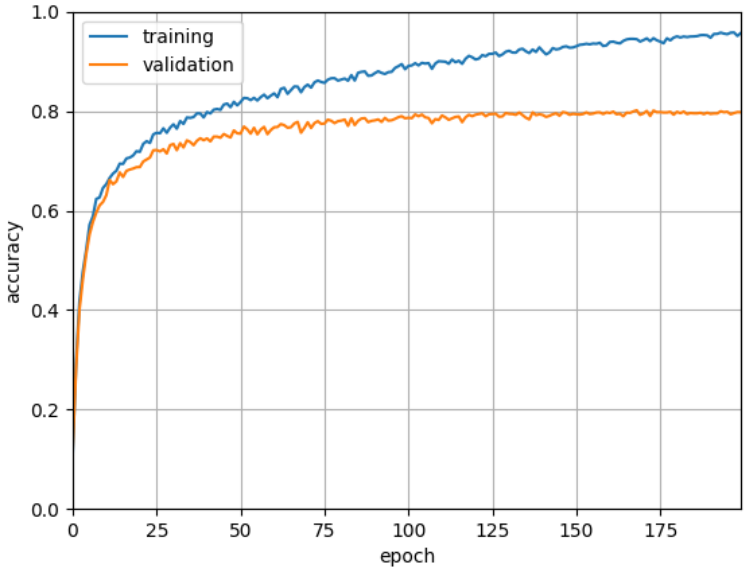
6.1

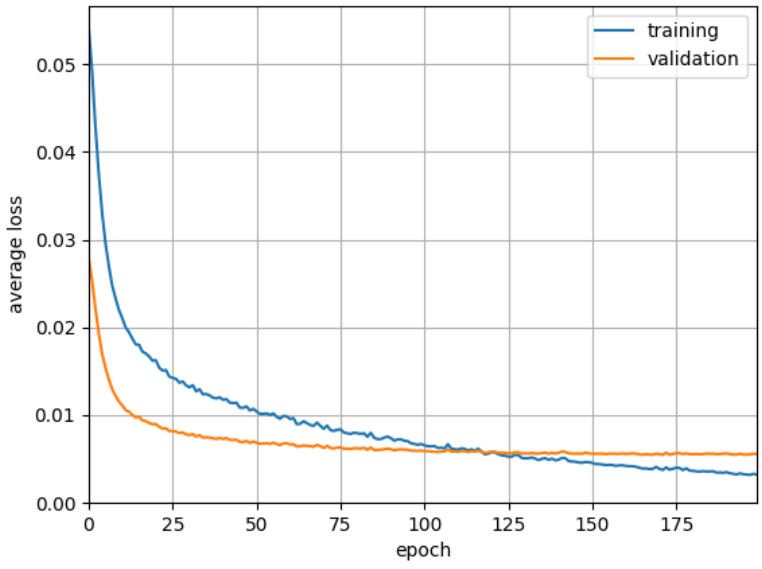
6.1.1

TO RUN:

```
python run_q6_mnist_fullyconnected.py
```

Reimplementing q2 fully connected network with PyTorch





Test accuracy: 0.8022

nn_q6.py (This files contains all NN objects and helper functions for question 6)

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

##################################################################################################
##################################### NEURAL NETWORKS ############################################
##################################################################################################

# for Q6.1.1
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.main = nn.Sequential(
            nn.Linear(1024, 64),
            nn.Sigmoid(),
            nn.Linear(64, 36)
        )

    def forward(self, X):
        return self.main(X)

# for Q6.1.2
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU(),
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(500, 64),
            nn.Sigmoid(),
            nn.Linear(64, 36),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = torch.flatten(x, 1)
        x = self.fc_layers(x)
        return x

# for Q6.1.3
class CNNcifar(nn.Module):
    def __init__(self):
        super(CNNcifar, self).__init__()

        self.conv_layers = nn.Sequential(
            # nn.Conv2d(3, 32, kernel_size=5),
            # nn.MaxPool2d(2, 2),
            # nn.ReLU(),
            # nn.Conv2d(32, 64, kernel_size=5),
            # nn.Dropout(0.2),
            # nn.MaxPool2d(2, 2),
            # nn.ReLU(),
            # nn.BatchNorm2d(64)

            # sees 32x32x3 image tensor
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
```

```
            # sees 16x16x16 tensor
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            # sees 8x8x32 tensor
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)

            # nn.Conv2d(3, 32, 5, padding=1),
            # nn.MaxPool2d(2, 2),
            # nn.Conv2d(32, 64, 5, padding=1),
            # nn.MaxPool2d(2, 2)
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(1024, 500),
            nn.ReLU(),
            nn.Linear(500, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = torch.flatten(x, 1)
        x = self.fc_layers(x)
        return x

# for Q6.2
class CNN2(nn.Module):
    def __init__(self):
        super(CNN2, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU(),
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(56180, 128),
            nn.Sigmoid(),
            nn.Linear(128, 17)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = torch.flatten(x, 1)
        x = self.fc_layers(x)
        return x

##############################################################################################
##################################### HELPER FUNCTIONS #######################################
##############################################################################################

# convert numpy array to dataloader
def np2loader(X, y, batchsize=128, shuffling=True):
    y = y.argmax(axis=1)
    X = torch.from_numpy(np.float32(X))
    y = torch.from_numpy(y)
    data = torch.utils.data.TensorDataset(X, y)
    if batchsize != None:
        loader = torch.utils.data.DataLoader(data, batch_size=batchsize, shuffle=shuffling)
    else:
        loader = torch.utils.data.DataLoader(data, shuffle=shuffling)
    return loader

def training_loop(myNet, trainLoader, validLoader, device, max_iters, learning_rate, lossf,
                                                    optimizer, fname, flatten=True):
```

```
    # Training loop
    # Initialize the network
    myNet = myNet.to(device)
    print(myNet)
    train_loss_list, train_acc_list, val_loss_list, val_acc_list = [], [], [], []
    for itr in range(max_iters):
        myNet.train()

        total_loss = 0.0
        total_correct = 0.0
        total_instances = 0

        for times, data in enumerate(trainLoader):
            inputs, labels = data[0].to(device), data[1].to(device)
            if flatten:
                inputs = inputs.view(inputs.shape[0], -1)

            # Zero the parameter gradients
            optimizer.zero_grad()
            # Foward, backward, optimize
            outputs = myNet(inputs)
            loss = lossf(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss

        train_accuracy, train_loss = evaluate_model(myNet, trainLoader, lossf, device, flatten)
        train_acc_list.append(train_accuracy)
        train_loss_list.append(train_loss)

        val_accuracy, val_loss = evaluate_model(myNet, validLoader, lossf, device, flatten)
        val_acc_list.append(val_accuracy)
        val_loss_list.append(val_loss)

        if itr % 10 == 0:
            print(
                "itr: {:02d} \t loss: {:.2f} \t acc : {:.2f} \t eval_acc : {:.2f}".format(
                    itr, total_loss, train_accuracy, val_accuracy
                )
            )

    # save the weights
    torch.save(myNet.state_dict(), fname)

    # visualize
    plot_train_valid(train_acc_list, val_acc_list, "accuracy")
    plot_train_valid(train_loss_list, val_loss_list, "average loss")
    plot_train(train_acc_list, "accuracy")
    plot_train(train_loss_list, "average loss")

def evaluate_model(myNet, dataLoader, lossf, device, flatten=True, my_class=False):
    myNet.eval()
    total_loss = 0.0
    total_correct = 0.0
    total_instances = 0
    for times, data in enumerate(dataLoader):
        inputs, labels = data[0].to(device), data[1].to(device)
        if my_class != False:
            for i in range(len(labels)):
                labels[i] = my_class
        if flatten:
            inputs = inputs.view(inputs.shape[0], -1)
        outputs = myNet(inputs)
        loss = lossf(outputs, labels)
        # Total loss
        total_loss += loss.item()
        with torch.no_grad():
            # average accuracy
            classifications = torch.argmax(outputs, dim=1)
            correct_predictions = sum(classifications==labels).item()
            total_correct+=correct_predictions
            total_instances+=len(inputs)
    accuracy = round(total_correct/total_instances, 4)
```

```
    return accuracy, total_loss/total_instances

# Plot train and valid loss / accuracies
def plot_train_valid(train_data, valid_data, datatype):
    plt.plot(range(len(train_data)), train_data, label="training")
    plt.plot(range(len(valid_data)), valid_data, label="validation")
    plt.xlabel("epoch")
    plt.ylabel(datatype)
    plt.xlim(0, len(train_data) - 1)
    plt.ylim(0, None)
    plt.legend()
    plt.grid()
    plt.show()

# Plot train and loss / accuracies
def plot_train(train_data, datatype):
    plt.plot(range(len(train_data)), train_data, label="training")
    plt.xlabel("epoch")
    plt.ylabel(datatype)
    plt.xlim(0, len(train_data) - 1)
    plt.ylim(0, None)
    plt.legend()
    plt.grid()
    plt.show()
```

run_q6_mnist_fullyconnected.py

```
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import numpy as np
import scipy
from nnq6 import *

# GPU or CPU
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print('GPU State:', device)

# Load original data
train_data = scipy.io.loadmat("../data/nist36_train.mat")
valid_data = scipy.io.loadmat("../data/nist36_valid.mat")
test_data = scipy.io.loadmat("../data/nist36_test.mat")

train_x, train_y = train_data["train_data"], train_data["train_labels"]
valid_x, valid_y = valid_data["valid_data"], valid_data["valid_labels"]
test_x, test_y = test_data["test_data"], test_data["test_labels"]

trainLoader = np2loader(train_x, train_y, batchsize=64)
validLoader = np2loader(valid_x, valid_y, shuffling=False)
testLoader = np2loader(test_x, test_y, shuffling=False)

trainLoaderCNN = np2loader(train_x.reshape((len(train_x), 1, 32, 32)), train_y, batchsize=64)
validLoaderCNN = np2loader(valid_x.reshape((len(valid_x), 1, 32, 32)), valid_y)

################################## Q6.1.1 ######################################
# Call the network
myNet = Net()

# Parameters
max_iters = 200
learning_rate = 1e-1
lossf = nn.CrossEntropyLoss()
optimizer = optim.SGD(myNet.parameters(), lr=learning_rate)
fname = 'q6_fully_connected.pth'

# Training loop
training_loop(myNet, trainLoader, validLoader, device, max_iters, learning_rate, lossf, optimizer,
                                                    fname)

# Test
myNet.load_state_dict(torch.load(fname))
```

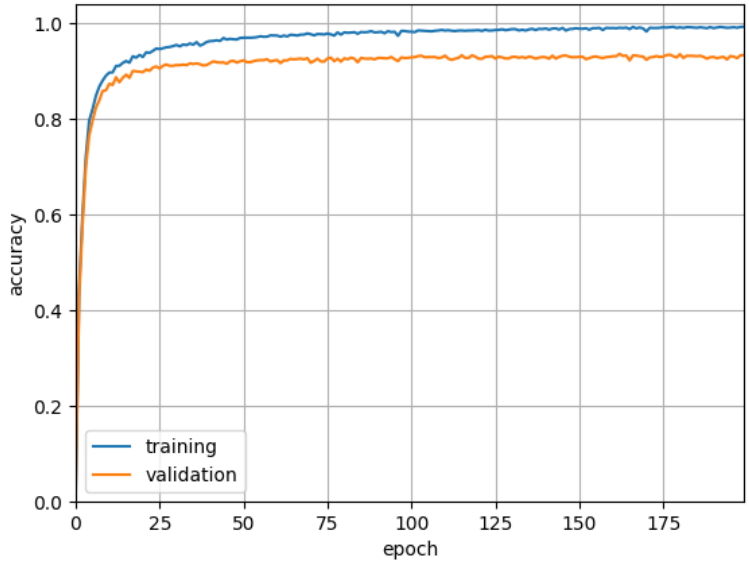
```
test_acc, _ = evaluate_model(myNet, testLoader, lossf, device)
print("Test accuracy: ", test_acc)
```

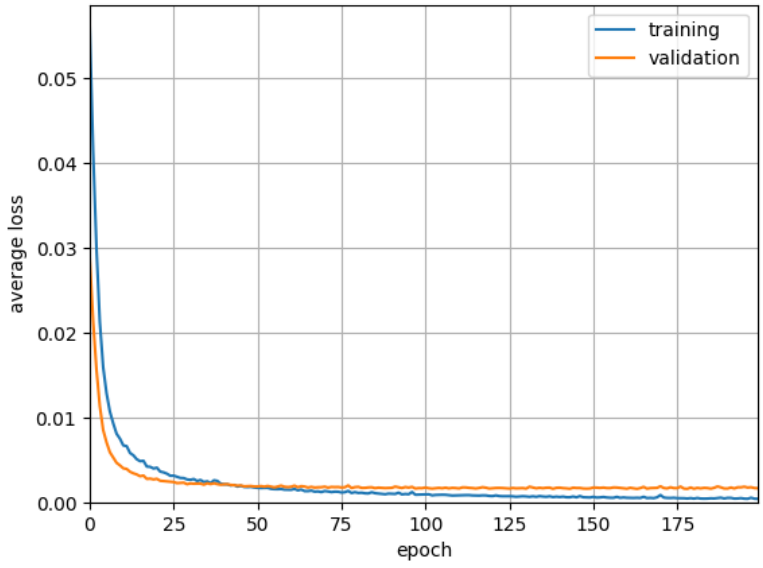
6.1.2

TO RUN:

```
python run_q6_mnist_cnn.py
```

Reimplementing q2 CNN network with PyTorch





Test accuracy: 0.9344

Performance Compared to Fully Connected Network

Comparing with the previous fully connected network, the performance of CNN is much better. The convergence is much faster as well. Their training accuracy both approached 100, but their validation and test accuracies differ.

The validation accuracy of the CNN model converged to around 95%, while the validation accuracy of the fully connected network was only able to reach around 80%.

The test accuracy of the CNN model is 0.9344, while it was 0.8022 for the fully connected network.

nn_q6.py

```
# for Q6.1.2
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU()
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(500, 64),
            nn.Sigmoid(),
            nn.Linear(64, 36)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = torch.flatten(x, 1)
        x = self.fc_layers(x)
        return x
```

run_q6_mnist_cnn.py

```
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import numpy as np
import scipy
from nnq6 import *

# GPU or CPU
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print('GPU State:', device)

# Load original data
train_data = scipy.io.loadmat("../data/nist36_train.mat")
valid_data = scipy.io.loadmat("../data/nist36_valid.mat")
test_data = scipy.io.loadmat("../data/nist36_test.mat")

train_x, train_y = train_data["train_data"], train_data["train_labels"]
valid_x, valid_y = valid_data["valid_data"], valid_data["valid_labels"]
test_x, test_y = test_data["test_data"], test_data["test_labels"]

trainLoader = np2loader(train_x, train_y, batchsize=64)
validLoader = np2loader(valid_x, valid_y, shuffling=False)
testLoader = np2loader(test_x, test_y, shuffling=False)

trainLoaderCNN = np2loader(train_x.reshape((len(train_x), 1, 32, 32)), train_y, batchsize=64)
validLoaderCNN = np2loader(valid_x.reshape((len(valid_x), 1, 32, 32)), valid_y)
testLoaderCNN = np2loader(test_x.reshape((len(test_x), 1, 32, 32)), test_y)

################################## Q6.1.2 ######################################
# Call the network
myCNN = CNN()

# Parameters
max_iters = 200
learning_rate = 1e-1
lossf = nn.CrossEntropyLoss()
optimizer = optim.SGD(myCNN.parameters(), lr=learning_rate)
fname = 'q6_mnist_cnn.pth'

# Training loop, comment this line when only doing testing
training_loop(myCNN, trainLoaderCNN, validLoaderCNN, device, max_iters, learning_rate, lossf,
                                                    optimizer, fname, False)
```

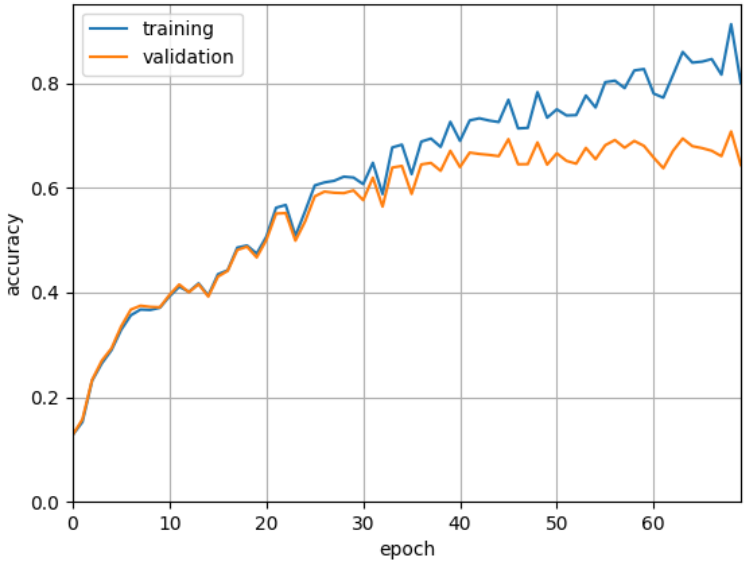
```
# Test
myCNN.load_state_dict(torch.load(fname))
test_acc, _ = evaluate_model(myCNN, testLoaderCNN, lossf, device, False)
print("Test accuracy: ", test_acc)
```

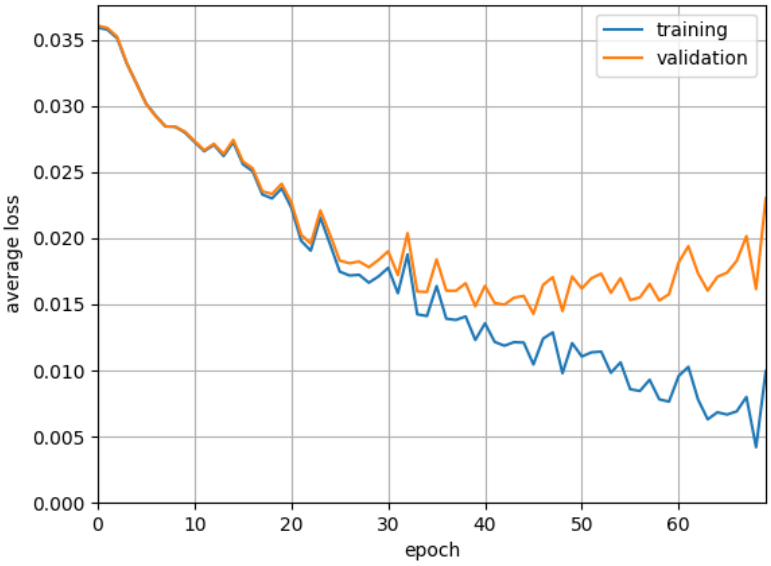
6.1.3

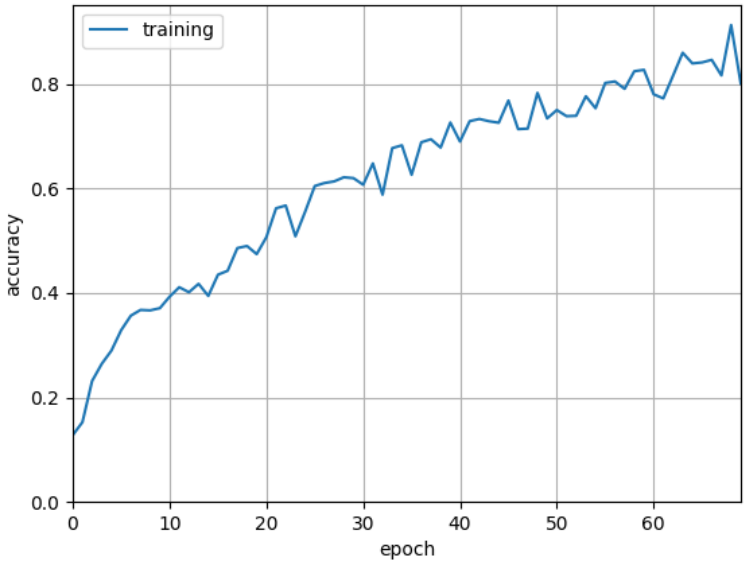
TO RUN:

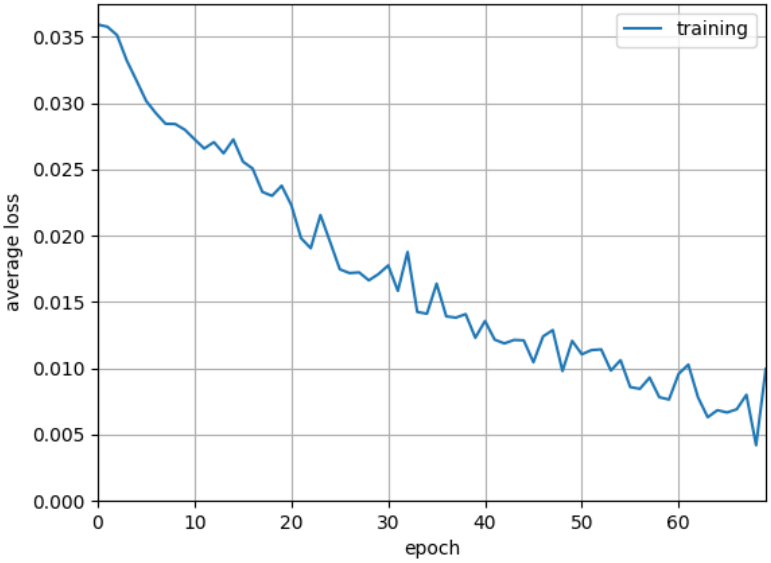
```
python run_q6_cifar10.py
```

Test accuracy: 0.7066






training
accuracy
epoch
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40
50
60


training
average loss
epoch
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0
10
20
30
40
50
60

nn_q6.py

```
# for Q6.1.3
class CNNcifar(nn.Module):
    def __init__(self):
        super(CNNcifar, self).__init__()

        self.conv_layers = nn.Sequential(
            # sees 32x32x3 image tensor
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            # sees 16x16x16 tensor
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            # sees 8x8x32 tensor
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(1024, 500),
            nn.ReLU(),
            nn.Linear(500, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = torch.flatten(x, 1)
        x = self.fc_layers(x)
        return x
```

run_q6_cifar.py

```
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import numpy as np
import scipy
from nnq6 import *

# GPU or CPU
device = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print('GPU State:', device)

################################## Q6.1.3 #######################################
# Call the network
myCNN = CNNcifar()

# Parameters
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Parameters
max_iters = 70
learning_rate = 5e-3
lossf = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(myCNN.parameters(), lr=learning_rate)
fname = 'q6_cifar_cnn.pth'
batch_size = 64

# Dataloaders
trainset = torchvision.datasets.CIFAR10(root='../data', train=True,
                                        download=True, transform=transform)
trainLoaderCNN = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                             shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='../data', train=False,
                                       download=True, transform=transform)
```

```
testLoaderCNN = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                            shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# Training loop, comment this line when only doing testing
training_loop(myCNN, trainLoaderCNN, testLoaderCNN, device, max_iters, learning_rate, lossf,
                                                    optimizer, fname, False)

# Test
myCNN.load_state_dict(torch.load(fname))
test_acc, _ = evaluate_model(myCNN, testLoaderCNN, lossf, device, False)
print("Test accuracy: ", test_acc)
```

6.2

TO RUN

```
python run_q6-2.py
```

Using flowers 17 dataset

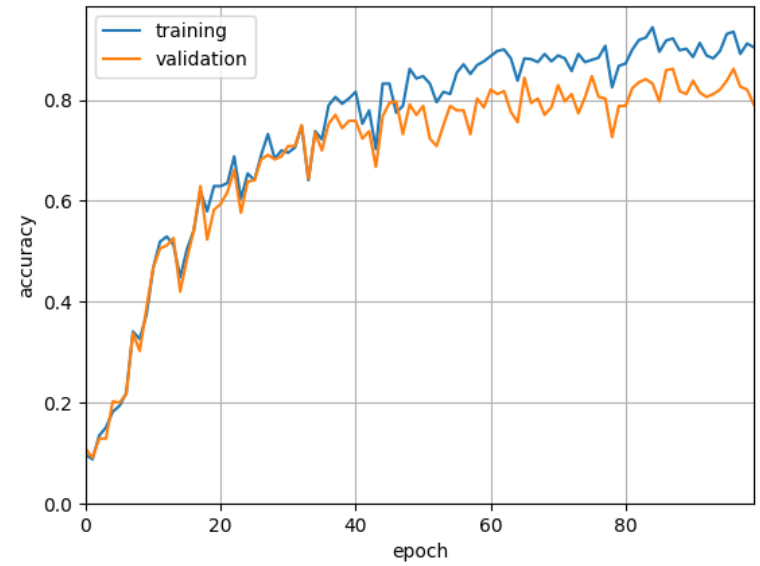
Squeezenet1_1 pretrained fine tuned: Test accuracy: 0.8471

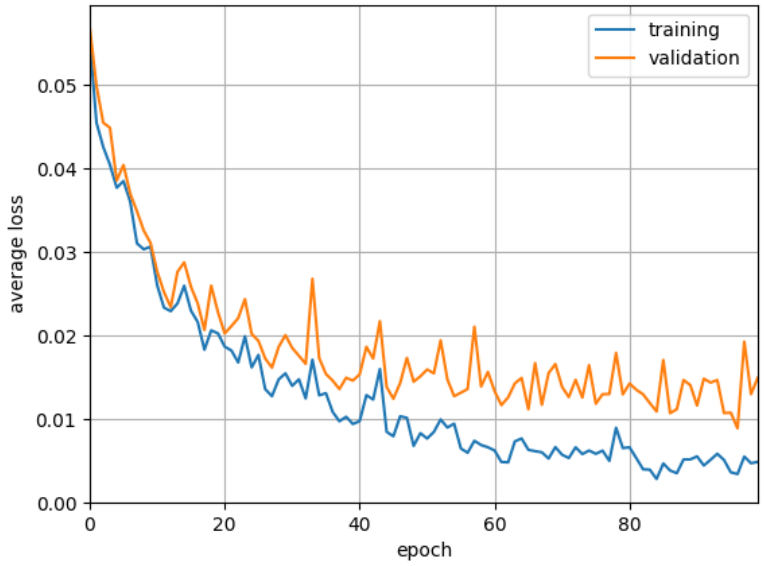
My own network from scratch for comparison: Test accuracy: 0.6235

Comparison: - The pretrained squeezenet has much higher test accuracy than my own network.

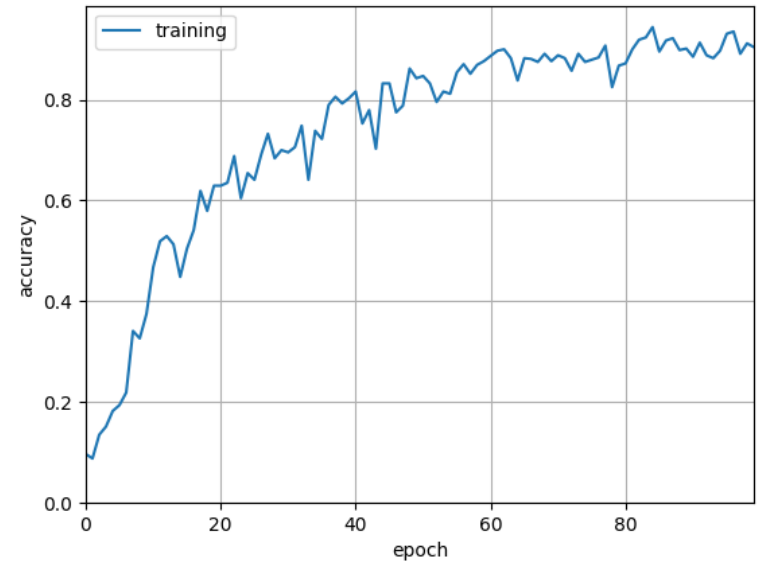
- The squeezenet1_1 I used has pretrained weights, so the model has a good starting point and fine tuning is easier.
- My own model has randomly initiated weights, which is not helpful for high accuracy within small ranges of changes in the parameters. Additionally, it has only simple convolutional and linear layers, thus does not have a good enough model structure to learn the image features.

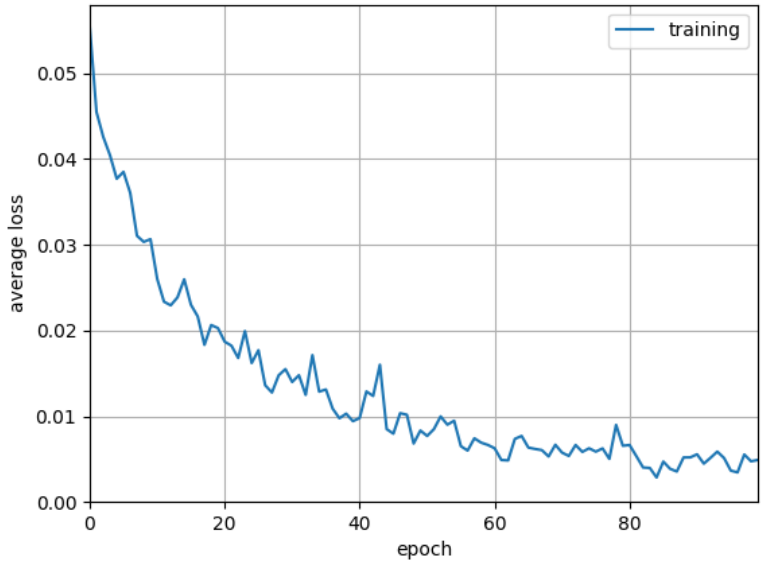
Squeezenet1_1 train/val results



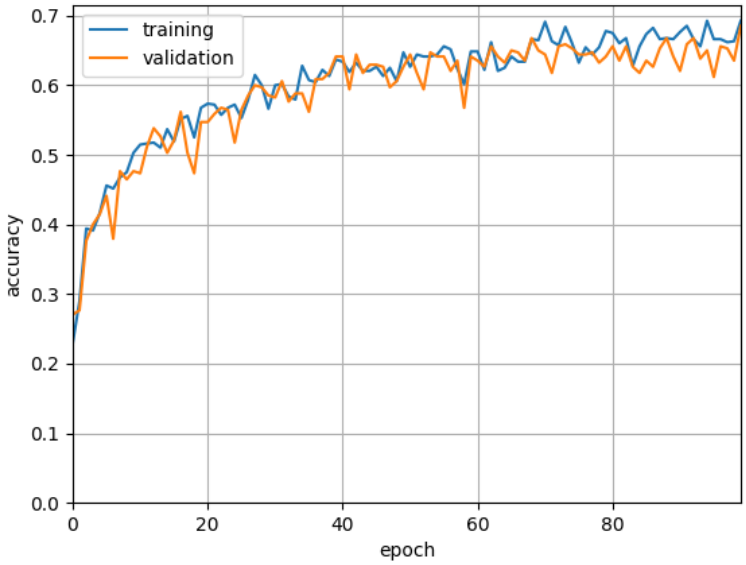


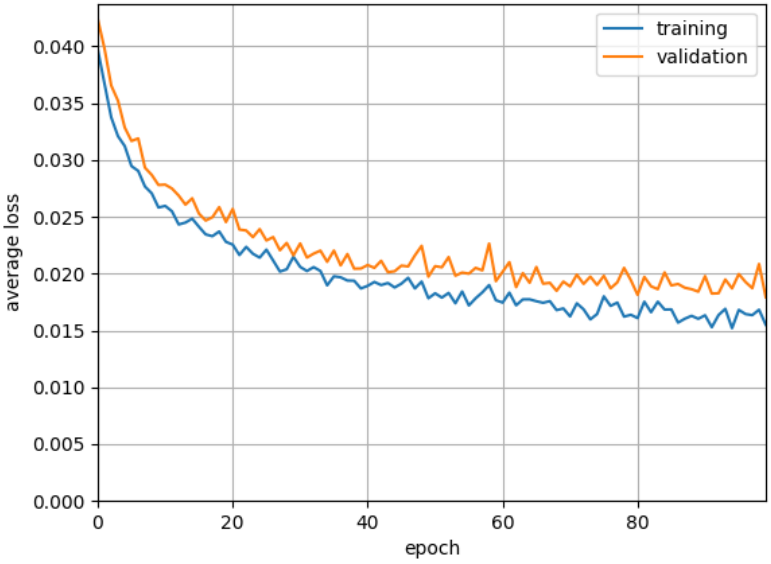
Squeezenet1_1 train results



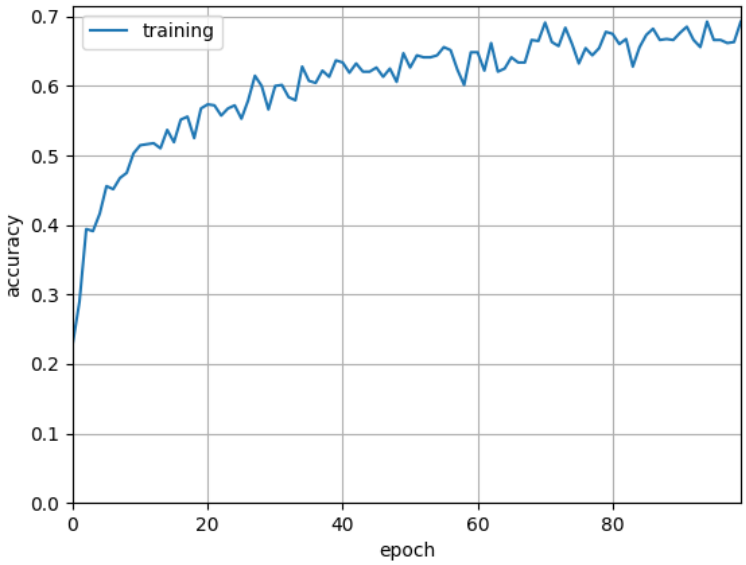


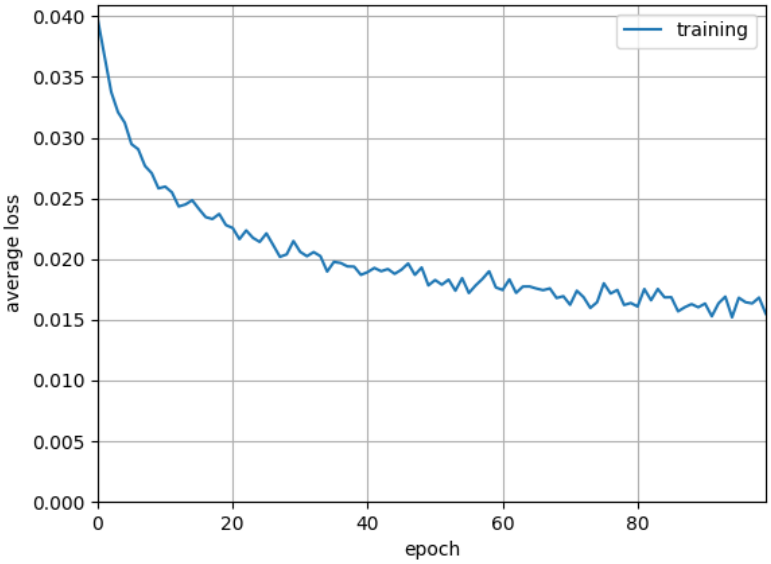
My own network train/val results





My own network train results





nn_q6.py
This is my own network

```
# for Q6.2
class CNN2(nn.Module):
    def __init__(self):
        super(CNN2, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10, 20, kernel_size=5),
            nn.Dropout(),
            nn.MaxPool2d(2),
            nn.ReLU(),
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(56180, 128),
            nn.Sigmoid(),
            nn.Linear(128, 17),
            #nn.Softmax()
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = torch.flatten(x, 1)
        x = self.fc_layers(x)
        return x
```

run_6-2.py

```
import torch
import torchvision
from torchvision.models import squeezenet1_1, SqueezeNet1_1_Weights
import numpy as np
import scipy
from nnq6 import *
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
from torch.utils.data import DataLoader
import torchvision.transforms as T
from torchvision.datasets import ImageFolder

if torch.cuda.is_available():
    device = 'cuda:0'
else:
    device = 'cpu'
print('GPU State:', device)

################################ Q 6.2 ############################
# Load the pretrained network squeezenet1_1
myNet = squeezenet1_1(SqueezeNet1_1_Weights)
print(myNet)

# Parameters
data_dir = "../data/oxford-flowers17/"
max_iters = 100
learning_rate = 1e-3
batch_size = 64
numworkers = 2
lossf = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(myNet.parameters())
fname = "q6_flowers.pth"

IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224, 0.225]

train_transform = T.Compose([
    T.Resize(224),
```

```
    T.RandomResizedCrop(224),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    T.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD),
  ])

val_transform = T.Compose([
    T.Resize(224),
    T.CenterCrop(224),
    T.ToTensor(),
    T.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD),
  ])

# Dataloaders
train_dset = ImageFolder(data_dir + "train", transform=train_transform)
train_loader = DataLoader(train_dset,
                    batch_size=batch_size,
                    num_workers=numworkers,
                    shuffle=True)

val_dset = ImageFolder(data_dir + "val", transform=val_transform)
val_loader = DataLoader(val_dset,
                    batch_size=batch_size,
                    shuffle=False,
                num_workers=numworkers)

test_dset = ImageFolder(data_dir + "test", transform=val_transform)
test_loader = DataLoader(test_dset,
                    batch_size=batch_size,
                    shuffle=False,
                num_workers=numworkers)

# Training loop
training_loop(myNet, train_loader, val_loader, device, max_iters, learning_rate, lossf, optimizer,
                                                    fname, False)

# Test
myNet.load_state_dict(torch.load(fname))
test_acc, _ = evaluate_model(myNet, test_loader, lossf, device, False)
print("Test accuracy: ", test_acc)
```

6.3

TO RUN

```
run_q6-3.py
```

Create 2 empty folders inside "custom_dataq6" folder, one named **images**, the other named **video**. Then download into video/ folder my goose video from https://drive.google.com/file/d/1GS8fJBwooZP9ImphqEkHNaAzn3yNmsic/view?usp=

My choice of data

- For this experiment, I chose to run pretrained RetNet50 with ImageNet class 99, goose.
- ImageNet subset data and my own video are all stored in a folder named "custom_dataq6" under "python" folder.
- My custom video is a short video of my aunt's goose eating vegetables in her garden.

Test accuracies on ImageNet validation dataset "goose" class:

Test accuracy on validation dataset: 0.3

Test accuracy on custom video dataset: 0.266

Comparison:

The accuracy on the frames from my video is slightly lower than that on the ImageNet dataset. The ImageNet validation dataset is challenging, as each goose picture looks different and are in different colors and orientations.

My own pictures also gave rather low results probably due to the geese in my video having different color and being in a different pose from most of the geese in ImageNet.

There are ways to make the model more rubust:

(1) Include more diverse representation of the same object in training data, or train on larger datasets.

(2) Utilize models with pretrained weights, fine-tune

(3) Utilize new models emerged from other machine learning domains, say, using vision transformers inspired by transformers in NLP. Transformers can have a global understanding of the image and take into account dependencies between features and avoid convolutional inductive bias.

ImageNet data (above) and my video (below):

run_q6-3.py

```
import torch
import torchvision
from torchvision.models import resnet50
import numpy as np
import scipy
from nnq6 import *
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset,DataLoader
import torchvision.transforms as T
import pickle
from torchvision.datasets import ImageFolder
import cv2
from PIL import Image

if torch.cuda.is_available():
    device = 'cuda:0'
else:
    device = 'cpu'
print('GPU State:', device)

################################## Q 6.3 ##############################
# Load the model
myNet = resnet50(pretrained=True)

# Parameters
data_dir = "../data/tiny-imagenet-200/val/goose"
batch_size = 16
numworkers = 2
lossf = nn.CrossEntropyLoss()
my_class = 99 # the chosen class to evaluate on

IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224, 0.225]

val_transform = T.Compose([
    T.Resize(224),
    T.CenterCrop(224),
```

```
    T.ToTensor(),
    T.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD),
  ])

# Dataloader
val_dset = ImageFolder(data_dir, transform=val_transform)
val_loader = DataLoader(val_dset,
                    batch_size=batch_size,
                    shuffle=False,
                num_workers=numworkers)

# Test with the validation dataset
test_acc, _ = evaluate_model(myNet, val_loader, lossf, device, False, 99)
print("Test accuracy on validation dataset: ", test_acc)

# ########################## custom video ###################3######
# Parameters
custom_data_dir = "custom_dataq6"

# Process video
video_dir = custom_data_dir + '/video/'
video_output_dir = custom_data_dir + '/images/'

# Open a video file
video_capture = cv2.VideoCapture(video_dir+f"goose.mp4")

frame_count = 0
while video_capture.isOpened():
    ret, frame = video_capture.read()
    if not ret:
        break
    cv2.imwrite(video_output_dir+f"frame_{frame_count}.jpg", frame)
    frame_count += 1

video_capture.release()
cv2.destroyAllWindows()

# Dataloader
custom_dset = ImageFolder(video_output_dir, transform=val_transform)
custom_loader = DataLoader(custom_dset,
                    batch_size=batch_size,
                    shuffle=False,
                num_workers=numworkers)

# Test with the custom video dataset
test_acc, _ = evaluate_model(myNet, custom_loader, lossf, device, False, 99)
print("Test accuracy on custom video dataset: ", test_acc)
```