16-720 Computer Vision: Homework 5 (Fall 2022) Neural Networks for Recognition

Submitted by: Sushanth Jayanth
Andrew ID: sushantj

December 10, 2022

1 Theory

Q1.1 Prove that softmax is invariant to translation
Ans. $\text{softmax}(x+c)_i = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i}e^c}{\sum_j e^{x_j}e^c} = \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}} = \frac{e^{x_i}}{\sum_j e^{x_j}} = \text{softmax}(x)$

We often choose $c = -\max x_i$ since we want the images be normalized such that mean is centered at origin.

Q1.2 Properties of Softmax
Ans.

- The range of all elements in the output of *softmax(x)* is **0 to 1**. The sum of all elements of *softmax(x)* is 1
- One could say that "softmax takes an arbitrary real valued vector x and turns it into a **class probability score between 0 and 1**".
- The first softmax step converts the arbitrary vector into an exponential form. The second steps sums over total scores. The final steps normalizes the score of one vector from the entire set

Q1.3 Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression

Ans. For a multi-layer neural network with no activtion function to introduce the non-linearity, the output can be defined as: $y_i = w * y_{i-1} + b_i$

For a linear regression model the output is usally given as: $y = wx + b$

Therefore, one can see that both would give the same results

Q1.4 Derive the gradient of the sigmoid activation function

Ans. $\frac{\partial \sigma(x)}{\partial x} = \frac{-1}{(1+e^{-x})^2} \times e^{-x} \times (-1) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \times \frac{1+e^{-x}-1}{1+e^{-x}} = \sigma(x)(1-\sigma(x))$

Q1.5 Derive graident of Loss 'J' w.r.t 'W' and 'x'

Ans. We know the output is defined as $y = wx + b$

Let's define gradient of loss w.r.t y as delta:
$\frac{\mathrm{dJ}}{\mathrm{dy}} =$ delta

To find the derivative of the Loss w.r.t 'W' we do:

$$\begin{aligned} \frac{\mathrm{dJ}}{\mathrm{dW}} &= \left(\frac{\mathrm{dJ}}{\mathrm{dy}}\right) * \left(\frac{\mathrm{dy}}{\mathrm{dw}}\right) \\ &= \text{delta} * \frac{d(w * x + b)}{\mathrm{dw}} \\ &= \text{delta} * x \end{aligned} \tag{1}$$

Similarly, to find the derivative of the loss w.r.t 'x':

$$\begin{aligned} \frac{\mathrm{dJ}}{\mathrm{dW}} &= \left(\frac{\mathrm{dJ}}{\mathrm{dy}}\right) * \left(\frac{\mathrm{dy}}{\mathrm{dx}}\right) \\ &= \text{delta} * \frac{d(w * x + b)}{\mathrm{dx}} \\ &= \text{delta} * w \end{aligned} \tag{2}$$

Q1.6

Ans.

- 1. Sigmoid Activation function can lead to vanishing gradients because the sigmoid curve has flat regions at the top and bottom which sets gradients to zero if the values are too high or too low (only a small middle band will yield good amount of gradients)
- 2. Output range of sigmoid is between [0,1] and that of tanh is [-1,1]. TanH is preferred since it can produce gradients for highly positive or negative activations
- 3. tanh has lesser flat regions in the curve leading to lesser zero gradients
- 4.

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

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

$$= 1 - 2 * \mathrm{sigma}(2x)$$

2 Implement a Fully Connected Network

2.1 Network Initialization

2.1.1 Why is it not a good idea to initialize a network with all zeros? If you imagine that every layer has weights and biases, what can a zero-initialized network output after training?

Ans.

- Firstly, setting all the weights to zero will result in all weights receiving similar gradients and learning similar features. Therefore they'll struggle to break symmetry and may not reduce loss for a long time.
- Secondly, initializing all with zeros may result in the neural network reaching only a local minima. The global minima will almost always have weights different from each other (due to non-linearity of optimization). Therefore, it's better to not set them all to zero

2.1.2 Code: Xavier Initialization

Ans.

```
def initialize_weights(in_size,out_size,params,name=''):
"""
Initialize weights according to xavier initialization

Args:
    in_size (_type_):
    out_size (_type_): _description_
    params (_type_): _description_
    name (str, optional): _description_. Defaults to ''.
"""
b = np.zeros(out_size)
W = np.random.randn(in_size, out_size) # since we're doing Wx multiplication
# we need to ensure the number of columns = number of inputs (i.e. size of x)
# however, they will be XW instead of Wx so here the order is switched
W = W * np.sqrt(6/(in_size+out_size))

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

2.1.3 Why do we initialize with random numbers? Why do we scale the initialization depending on layer size?

Ans. Initializing with random numbers allows for the weights to not all change in the same direction and **helps in breaking symmetry**. This also prevents the network from saturating at a local minima at the beginning of train time.
We scale all the weights based on the layer size to ensure that we maintain activation variances and back-propagated gradients variance as one moves up or down the network. The effect of doing so is seen in the figure below:

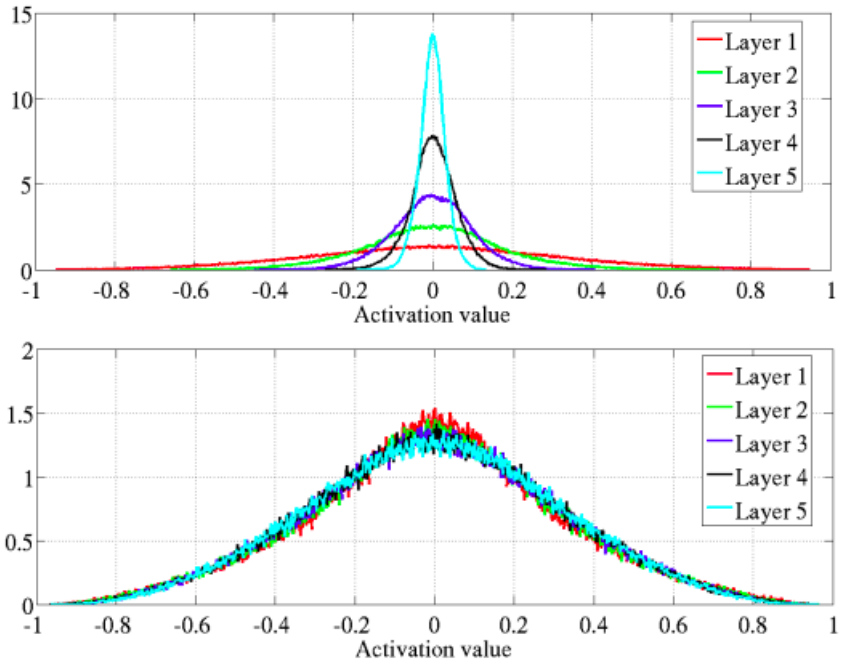


Figure 1: Non-scaled weight initialization (top) vs scaled weight initialization (bottom)

2.2 Forward Propagation

2.2.1 Code: Implement Sigmoid with forward propagation

```
############################### Q 2.2.1 ###############################
# x is a matrix
# res is sigmoid activation function
def sigmoid(x):
    """
    Sigmoid Activation Function

    Args:
        x (_type_): _description_

    Returns:
        _type_: _description_
    """
    res = (1/ (1 + np.exp(-(x))))
    return res

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

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

    pre_act = 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 Code: Softmax with numerical stability

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

    # shift the data in x
    shifted_x = x - np.expand_dims(np.max(x,axis=1),1)

    # create a column vector of the sum along each row
    exp_sum = np.expand_dims(np.sum(np.exp(shifted_x),axis=1), axis=1)

    # use this sum of each row to normalize the class prediction scores
    res = (1/exp_sum) * np.exp(shifted_x)

    return res
```

2.2.3 Code: Compute Loss and Accuracy

```
    ############################## 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

    # probs is the output of our softmax function
    # we'll define our loss as the -1*(unnormalized log of these probs)

    assert y.shape == probs.shape
    log_probs = y * np.log(probs)
    loss = -(np.sum(log_probs))

    # calculate accuracy over all training examples
    true_positives = (np.where(np.argmax(y, axis=1) == np.argmax(probs, axis=1)))[0].shape[0]
    acc = true_positives / probs.shape[0]

    return loss, acc
```

2.3 Backward Propagation

2.3.1 Code: Backprop

```
    ############################## 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
    activation_derivative = activation_deriv(post_act) * delta

    # we do W*x + b => activation_deriv
    # Therefore, the sequence of gates are:
    # multiplication gate -> addition gate -> sigmoid_gate (or softmax)
    # the addition gate flows equal gradient to grad_b and mutliplication gate
    grad_b = np.sum(activation_derivative, axis=0)/activation_derivative.shape[0]

    grad_W = X.T @ activation_derivative
    grad_X = activation_derivative @ W.T

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

2.4 Training Loop

2.4.1 Code: Training Loop

```
    ############################## Q 2.4 ##############################
# split x and y into random batches
# return a list of [(batch1_x,batch1_y)...]

# remember: x = (examples, dimensions)
#           y = (examples, classes)
# therefore, we should split data along the rows

def get_random_batches(x,y,batch_size):
    batches = []
    print("x and y shape is", x.shape, y.shape)

    assert x.shape[0] == y.shape[0]
    p = np.random.permutation(x.shape[0])
    shuffled_x, shuffled_y = x[p,:], y[p,:]

    num_batches = (x.shape[0]/batch_size)
    x_batches = np.split(shuffled_x, num_batches)
    y_batches = np.split(shuffled_y, num_batches)

    for xb, yb in zip(x_batches, y_batches):
        batches.append((xb,yb))

    return batches
```

```
import numpy as np
# you should write your functions in nn.py
from NN import *
from util import *

def main():
    # 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 = {}

    print("STARTING X shape is", x.shape)
    print("STARTING Y shape is", y.shape)

    # 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('should be zero and one\t',test.min(),test.max())

# Forward function does the following in order:
# 1. Finds the value of XW + b (idk why it's XW, this was TA's choice)
#
h1 = forward(x,params,'layer1', sigmoid)
print("forward shape is",h1.shape)

# Q 2.2.2
# implement softmax
probs = forward(h1,params,'output',softmax)
# make sure you understand these values!
# 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] or higher, and 0.25
# if it is not, check softmax!
print("{}, {:.2f} loss and accuracy".format(loss,acc))

#? TA comments
# here we cheat for you
# the derivative of cross-entropy(softmax(x)) is probs - 1[correct actions]

#? My comments
# derivative in front of (upstream of softmax)
delta1 = probs - y

#? TA comments
# 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

#? Reason behind using linear_deriv (np.ones) as softmax derivative #
# In pytorch the cross entropy loss includes the softmax on final layer + loss function
# Therefore in pytorch's case, if we calculate derivative of cross-entropy loss,
#  we automatically have calculated the derivate through softmax as well

#? My comments
# basically the derivative w.r.t the softmax function should only return an array of np.ones
# of the same shape as the post_activation shape of softmax( 40,4) -> this is done by
    linear_deriv

# the backwards function does the following in order
# 1. Finds derivative through activation function (here linear_deriv)
# 2. Find derivative to bias
# 3. Finds derivative to W and X
# Finally, it returns derivative on X as delta
delta2 = backwards(delta1, params, 'output', linear_deriv)
print("delta2 shape is", delta2.shape)

delta3 = backwards(delta2,params,'layer1',sigmoid_deriv)
print("delta3 shape is", delta3.shape)
```

```
# W and b should match their gradients sizes
print("name | grad_shape | weight shape")
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([_[0].shape[0] for _ in batches])

######### WRITE A TRAINING LOOP HERE #################
train_loop(batches)


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

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

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

# compute gradients using finite difference

eps = 1e-6 # epsilon value

# for k,v in params.items():
#     if '_' in k:
#         continue
    # we have a real parameter!
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   subtract 2*epsilon
    #   run the network
    #   get the loss
    #   restore the original parameter value
    #   compute derivative with central diffs

forward_loss, forward_params = grad_check_loop(params, eps, x, y)
backward_loss, backward_params = grad_check_loop(forward_params, -2*eps, x, y)

numerical_grad = (forward_loss-backward_loss)/(2*eps)
print("numerical grad is", numerical_grad)
params = backward_params

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))

def train_loop(batches):
    # init weights
    # parameters in a dictionary
    params = {}

    # initialize a layer
    initialize_weights(2,25,params,'layer1')
    initialize_weights(25,4,params,'output')

    max_iters = 500
    learning_rate = 0.5e-3
    # with default settings, you should get loss < 35 and accuracy > 75%
    for itr in range(max_iters):
        total_loss = 0
        avg_acc = 0
        for xb,yb in batches:

            # implement forward
            h1 = forward(xb,params,'layer1', sigmoid)

            # implement softmax
            probs = forward(h1,params,'output',softmax)
            # print(probs.min(),min(probs.sum(1)),max(probs.sum(1)),probs.shape)

            # loss
            loss, acc = compute_loss_and_acc(yb, probs)
            total_loss += loss
            avg_acc += acc
            # print("{}, {:.2f}".format(loss,acc))

            # backward
            delta1 = probs - yb

            delta2 = backwards(delta1, params, 'output', linear_deriv)

            delta3 = 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] -= learning_rate * v

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


if __name__ == '__main__':
    main()
```

2.5 Numerical Gradient Checker

Main function

```
def main():
    ######### WRITE A TRAINING LOOP HERE #################
    train_loop(batches)


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

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

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

    # compute gradients using finite difference

    eps = 1e-6 # epsilon value

    # for k,v in params.items():
    #     if '_' in k:
    #         continue

    forward_loss, forward_params = grad_check_loop(params, eps, x, y)
    backward_loss, backward_params = grad_check_loop(forward_params, -2*eps, x, y)

    numerical_grad = (forward_loss-backward_loss)/(2*eps)
    print("numerical grad is", numerical_grad)
    params = backward_params

    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))
```

Gradient Check function

```
    def grad_check_loop(params, eps, x, y):
    for k,v in params.items():
        if '_' in k:
            continue
        else:
            params[k] += eps

    h1 = forward(x,params,'layer1')
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(y, probs)
    return loss, params
```

3 Training Models

3.1 Code: Train the above network for 50 epochs

The following parameters were used:

- learning rate = 1e-3
- batch size = 64
- number of epochs = 100

```
    import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
from NN import *

def main():
    visualize = True

    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']

    # the train_x has a shape of (10800,1024) viz 10800 images of 32x32 = 1024 size each
    # the train_y has a shape of (10800, 36) viz 10800 images and 36 possible labels for each
    # Note. out of the 36 possible labels for each image
    # only 1 element will be = 1, remainaing = 0

    if False: # view the data
        np.random.shuffle(train_x)
        for crop in train_x:
            plt.imshow(crop.reshape(32,32).T, cmap="Greys")
            plt.show()

    max_iters = 100
    # pick a batch size, learning rate
    batch_size = 64
    learning_rate = 1e-3
    hidden_size = 64

    batches = get_random_batches(train_x,train_y,batch_size)
    batch_num = len(batches)

    params = {}

    # initialize layers

    # LAYER 1
    # the train_x shape of (10800,1024) will cause
    # the weights to have shape (1024 x hidden_layer_size)
    # (hidden layer size is arbitrary and here = 64)
    initialize_weights(train_x.shape[1], hidden_size, params, "layer1")

    # LAYER 2
    # here too the weights will have a size which maps the hidden layer
    # to the output layer. The weights shape will be (hidden_size, 36)
    # where 36 = number of total possible labels
    initialize_weights(hidden_size, train_y.shape[1], params, "output")
    layer1_W_initial = np.copy(params["Wlayer1"]) # copy for Q3.3
```

```
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []

# iterate over epochs (max_iters = epochs)
for itr in range(max_iters):
    learning_rate = learning_rate*0.9995
    # record training and validation loss and accuracy for plotting
    h1 = forward(train_x,params,'layer1')
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(train_y, probs)
    train_loss.append(loss/train_x.shape[0])
    train_acc.append(acc)

    h1 = forward(valid_x,params,'layer1')
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(valid_y, probs)
    valid_loss.append(loss/valid_x.shape[0])
    valid_acc.append(acc)

    total_loss = 0
    avg_acc = 0

    # iterate over batches
    for xb,yb in batches:
        # training loop can be exactly the same as q2!
        total_loss, avg_acc = train_loop(xb, yb, params, learning_rate, total_loss, avg_acc)

    avg_acc = avg_acc/len(batches)

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


# record final training and validation accuracy and loss
h1 = forward(train_x,params,'layer1')
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(train_y, probs)
train_loss.append(loss/train_x.shape[0])
train_acc.append(acc)


h1 = forward(valid_x,params,'layer1')
val_probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(valid_y, val_probs)
valid_loss.append(loss/valid_x.shape[0])
valid_acc.append(acc)

# report validation accuracy; aim for 75%
print('Validation accuracy: ', valid_acc[-1])

# compute and report test accuracy
h1 = forward(test_x,params,'layer1')
test_probs = forward(h1,params,'output',softmax)
_, test_acc = compute_loss_and_acc(test_y, test_probs)
print('Test accuracy: ', test_acc)

# save the final network
import pickle
saved_params = {k:v for k,v in params.items() if '_' not in k}
with open('q3_weights.pickle', 'wb') as handle:
       pickle.dump(saved_params, handle, protocol=pickle.HIGHEST_PROTOCOL)

if visualize is True:
```

```
# plot loss curves
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(valid_loss)), valid_loss, label="validation")
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(train_loss)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()

# plot accuracy curves
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(valid_acc)), valid_acc, label="validation")
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.xlim(0, len(train_acc)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()


# Q3.3

# visualize weights
fig = plt.figure(figsize=(8,8))
plt.title("Layer 1 weights after initialization")
plt.axis("off")
grid = ImageGrid(fig, 111, nrows_ncols=(8, 8), axes_pad=0.05)
for i, ax in enumerate(grid):
    ax.imshow(layer1_W_initial[:,i].reshape((32, 32)).T, cmap="Greys")
    ax.set_axis_off()
plt.show()

v = np.max(np.abs(params['Wlayer1']))
fig = plt.figure(figsize=(8,8))
plt.title("Layer 1 weights after training")
plt.axis("off")
grid = ImageGrid(fig, 111, nrows_ncols=(8, 8), axes_pad=0.05)
for i, ax in enumerate(grid):
    ax.imshow(params['Wlayer1'][:,i].reshape((32, 32)).T, cmap="Greys", vmin=-v, vmax=v)
    ax.set_axis_off()
plt.show()

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

# compute confusion matrix
for i in range(0, train_y.shape[0]):
    truth = np.argmax(probs[i, :])
    pred = np.argmax(train_y[i, :])

    confusion_matrix[pred, truth] += 1

import string
plt.imshow(confusion_matrix,interpolation='nearest')
plt.grid()
plt.xticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in
    range(10)]))
plt.yticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in
    range(10)]))
plt.xlabel("predicted label")
plt.ylabel("true label")
plt.colorbar()
```

```
        plt.show()


def train_loop(xb, yb, params, learning_rate, total_loss, avg_acc):

        # implement forward
        h1 = forward(xb,params,'layer1', sigmoid)

        # implement softmax
        probs = forward(h1,params,'output',softmax)
        # print(probs.min(),min(probs.sum(1)),max(probs.sum(1)),probs.shape)

        # loss
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc
        # print("{}, {:.2f}".format(loss,acc))

        # backward
        delta1 = probs - yb

        delta2 = backwards(delta1, params, 'output', linear_deriv)

        delta3 = 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] -= learning_rate * v

        return total_loss, avg_acc

if __name__ == '__main__':
    main()
```

The plots for the above code was visualized below:

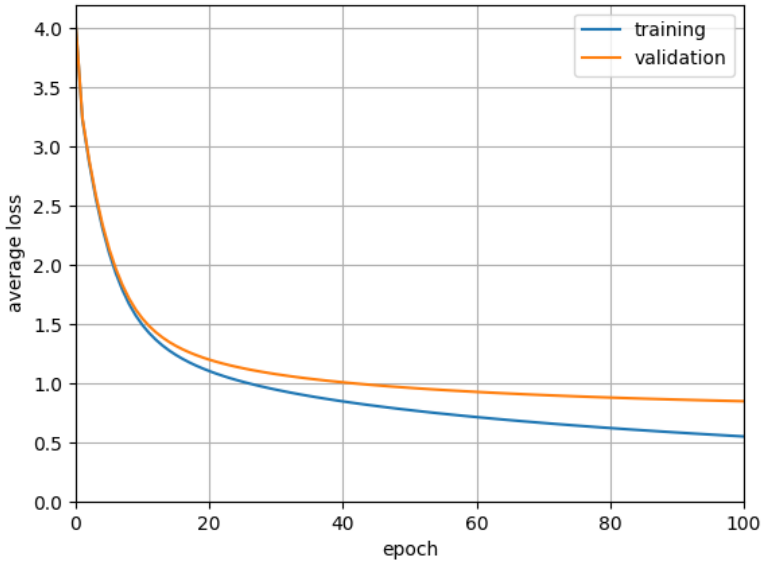


Figure 2: Training loss and Validation Loss vs epochs

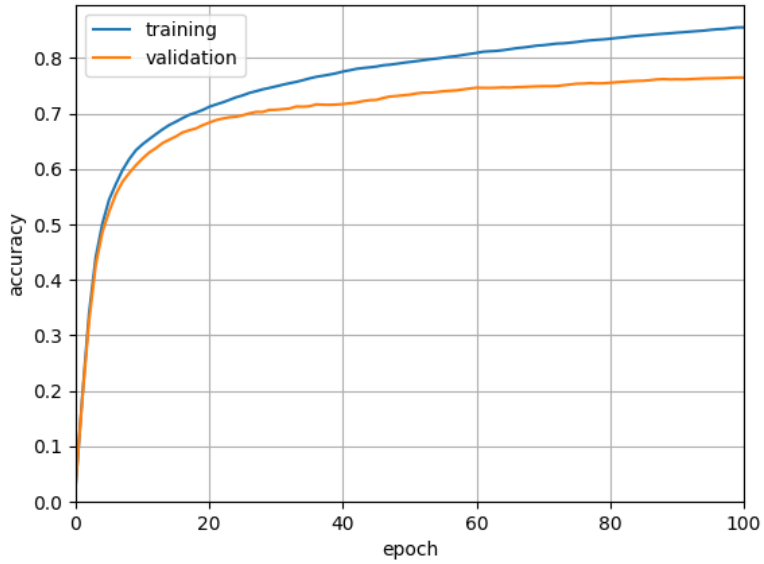


Figure 3: Training accuracy and Validation accuracy vs Epochs

3.2 Code: Training Network with different Hyper-parameters

Original Learning Rate

- train_accuracy = 86%
- val_accuracy = 76.4%
- test_accuracy = 75.5%

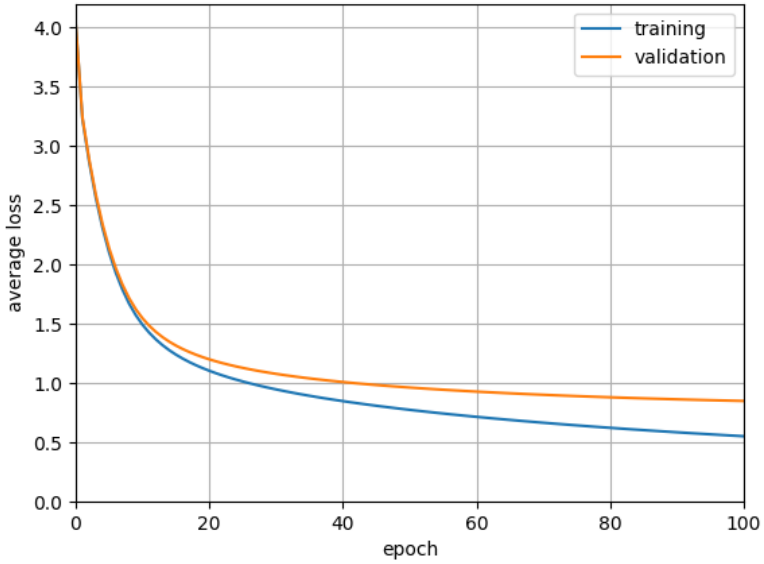


Figure 4: Training loss and Validation Loss vs epochs

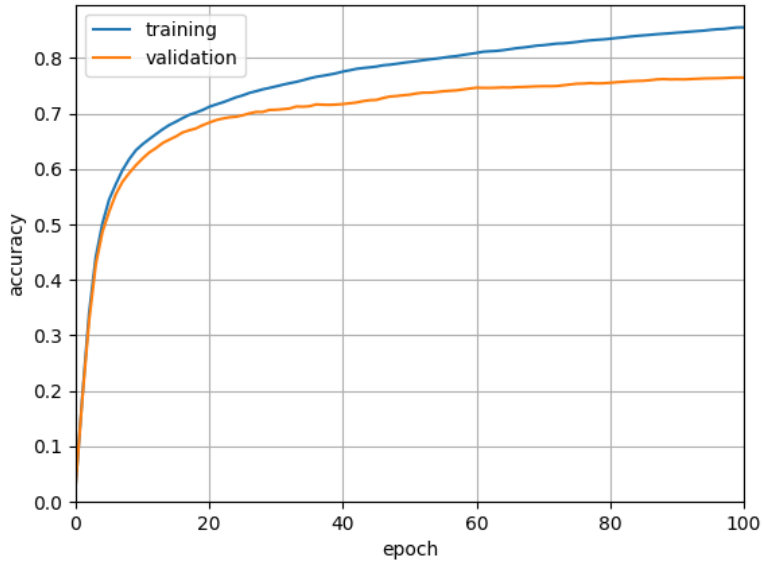


Figure 5: Training accuracy and Validation accuracy vs Epochs

10 * Learning Rate

- train_accuracy = 94%
- val_accuracy = 73%
- test_accuracy = 74%

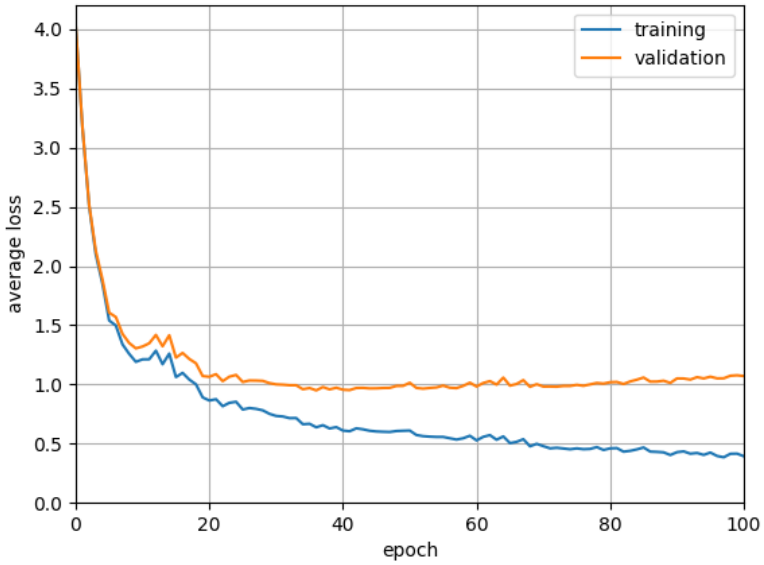


Figure 6: Training loss and Validation Loss vs epochs

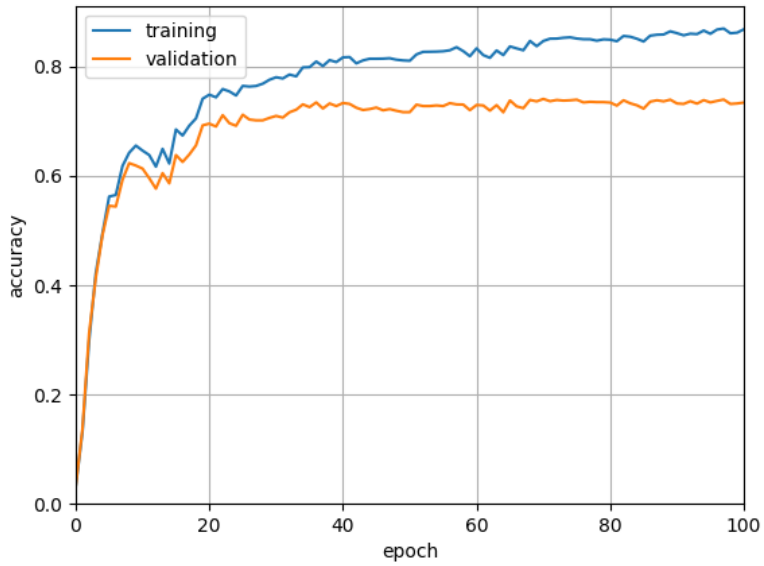


Figure 7: Training accuracy and Validation accuracy vs Epochs

0.1 * Learning Rate

- train_accuracy = 65%
- val_accuracy = 63%
- test_accuracy = 63%

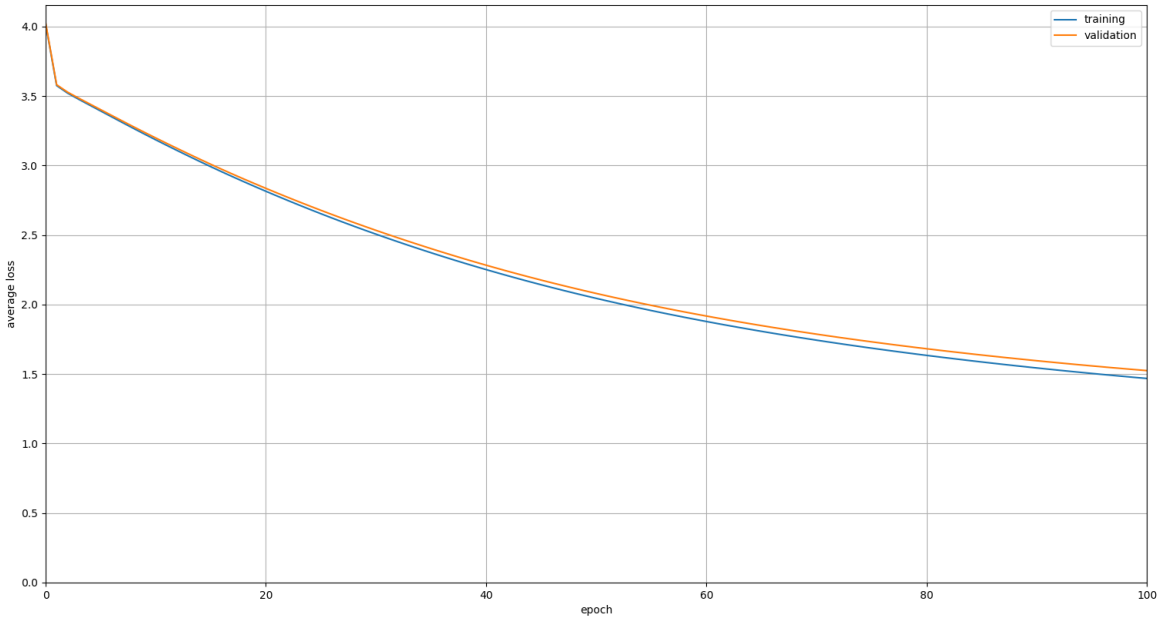


Figure 8: Training loss and Validation Loss vs epochs

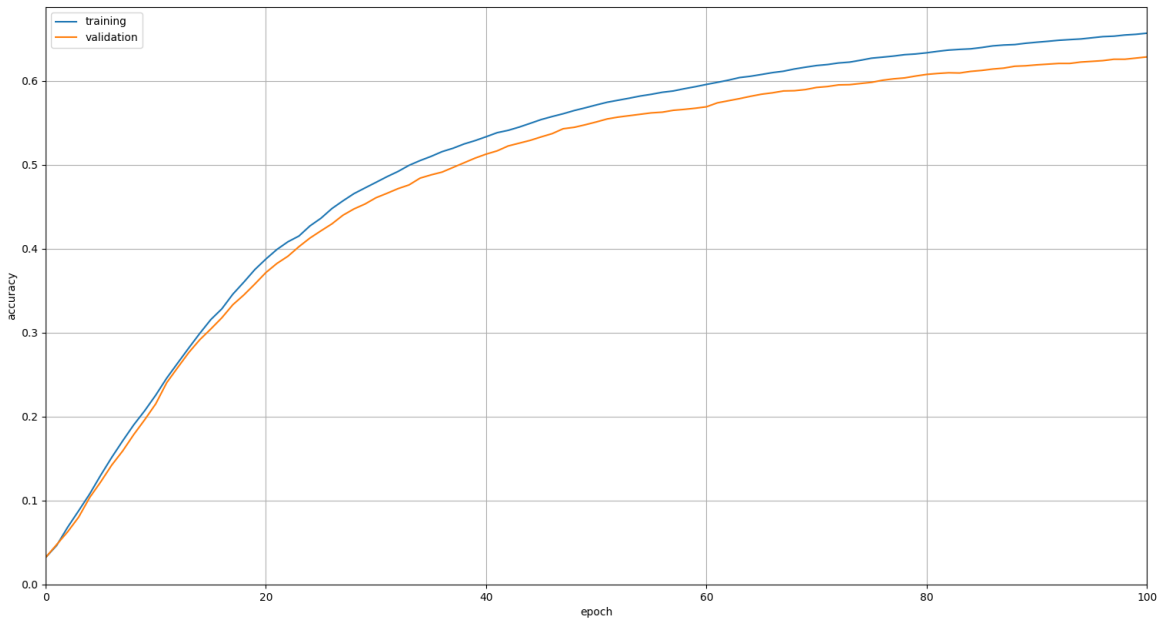
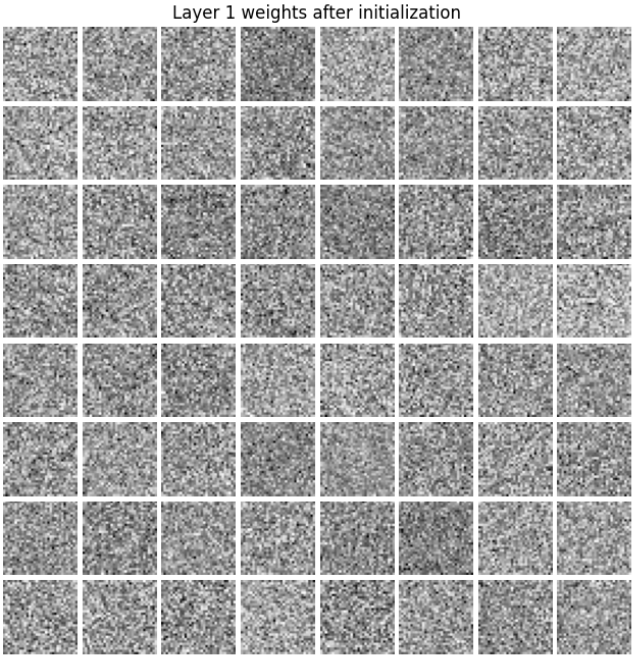
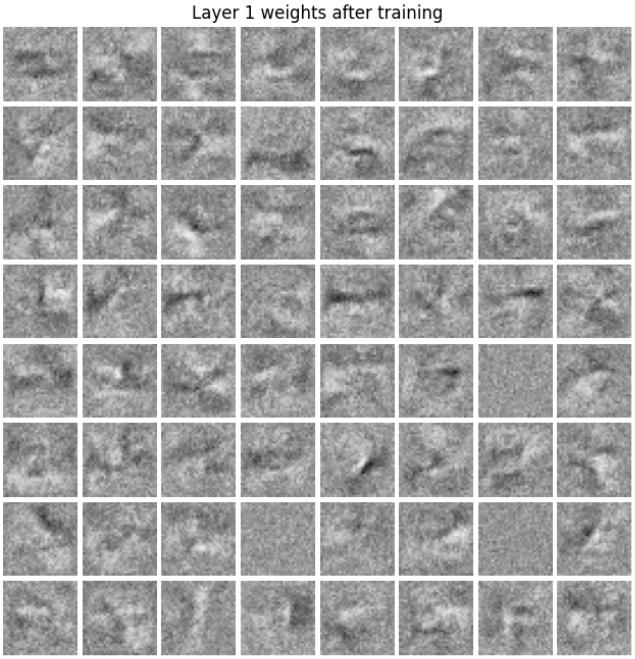


Figure 9: Training accuracy and Validation accuracy vs Epochs

3.3 Visualize Weights





3.4 Visualize Confusion Matrix

Original Learning Rate was the best model

- train_accuracy = 86%
- val_accuracy = 76.4%
- test_accuracy = 75.5%

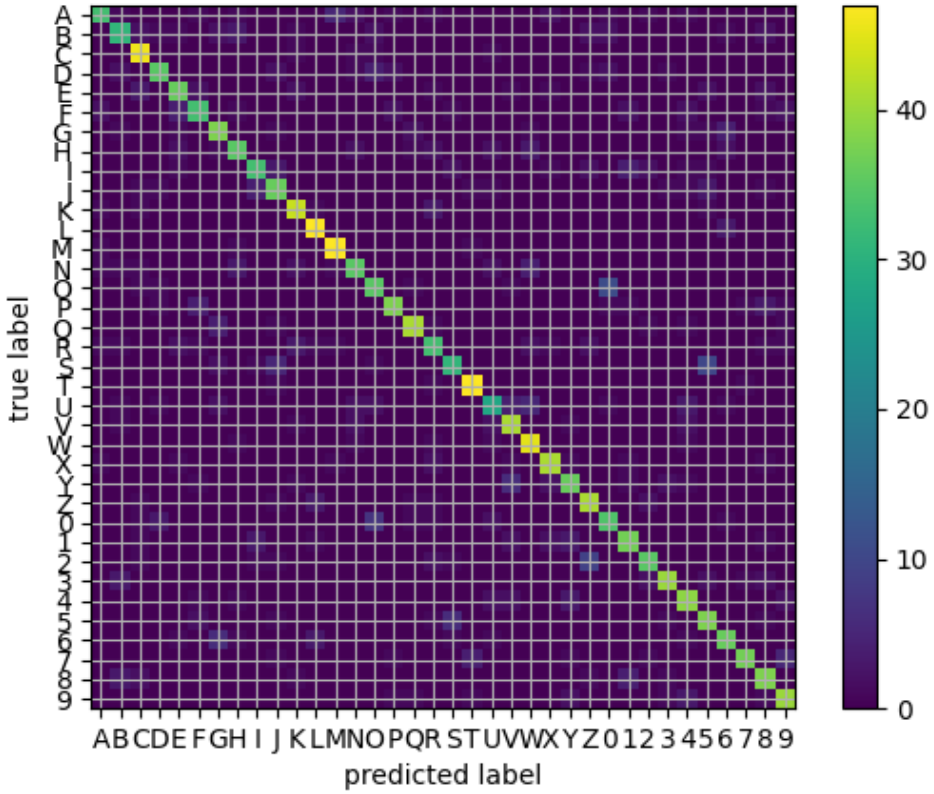


Figure 10: Confusion Matrix

4 Extract Text from Images

4.1 Theory: Assumptions and Failure points in sample based model

The two big assumptions are:

- The image was filtered using a binary mask and then passed to the network for detection. If a non-binary image was passed (say background was not white), then the network may not have good detections
- If the letters overlap, then our network which was trained on individual letters will fail to distinguish between letters

Examples of possible failure points are shown below:

Figure 11: Skewed letters difficult to classify and non-white background

Figure 12: Overlapping letters would be difficult to classify

4.2 Code: Implement find_letters function

```
def findLetters(image):
bboxes = []
bw = None

# apply threshold
image = skimage.color.rgb2gray(image)
thresh = threshold_otsu(image)
bw = closing(image < thresh, square(3))
black_white = deepcopy(bw)

# remove artifacts connected to image border
cleared = clear_border(bw)

# label image regions
label_image = label(cleared)

bbox_list = []
for region in regionprops(label_image):
    # take regions with large enough areas
    if region.area >= 200:
        bbox_list.append(region.bbox)

# print(bbox_list)

return bbox_list, bw
```

4.3 Letters Detected in Images

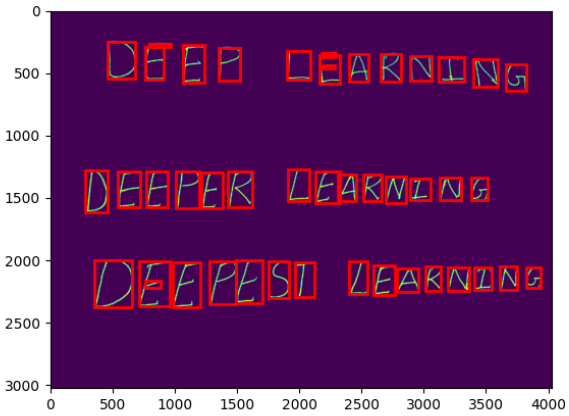


Figure 13

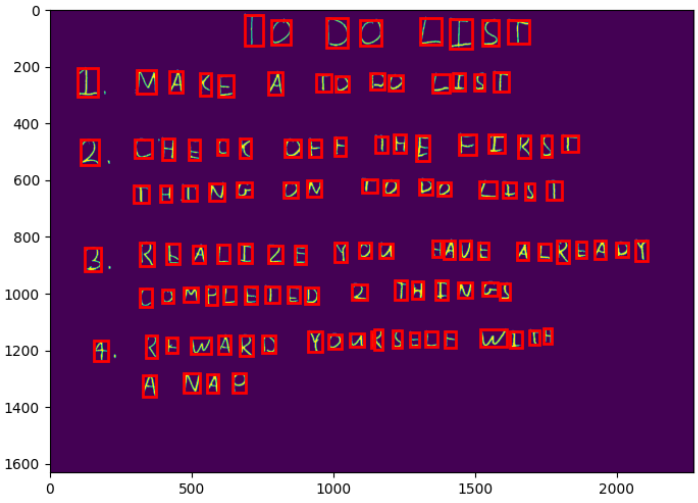


Figure 14

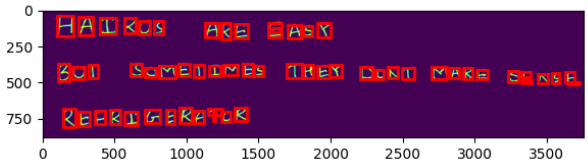


Figure 15

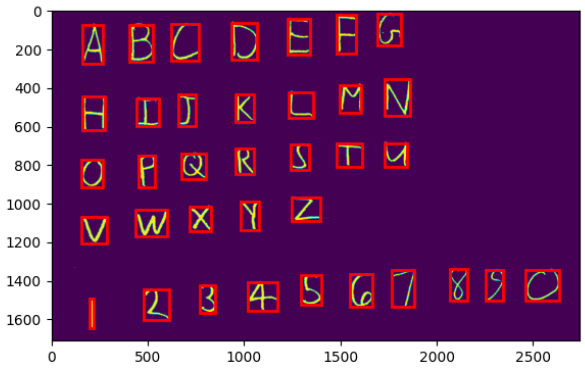


Figure 16

4.4 Classification of Letters through the Neural Network

The rows of data were split. The inference throught the network could not be done in time.

```
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 NN import *
from q4 import *
# do not include any more libraries here!
# no opencv, no sklearn, etc!
import warnings
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 #####
    ############################
    rows = []
    realtime_arr = [bboxes[0]]

    for i in range(1,len(bboxes)):
        curr_bbox = bboxes[i]
        box_center_x, box_center_y = (curr_bbox[3] + curr_bbox[1])/2, (curr_bbox[2] +
            curr_bbox[0])/2

        ref_bbox = realtime_arr[-1]

        if box_center_y < ref_bbox[0] or box_center_y > ref_bbox[2]:
            rows.append(realtime_arr)
            realtime_arr = []
            realtime_arr.append(curr_bbox)

        else:
            realtime_arr.append(curr_bbox)

    rows.append(realtime_arr)

    print(" This image had this many rows", len(rows))
```

```
# 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 #####
##########################

# 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 #####
##########################
```

---

5 Not Attempted

6 PyTorch

6.1 Train a neural network in PyTorch

6.1.1 Fully Connected Network on given NIST36

Main Code

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import sys

import torch
import torchvision
import torch.optim as optim
import torch.nn as nn

from mpl_toolkits.axes_grid1 import ImageGrid
from NN import *
from torch.utils.data import TensorDataset, DataLoader

from torch.utils.tensorboard import SummaryWriter

# add models folder to path
sys.path.insert(0,'/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/python/network_models')

from q6_model import SushNet

def main():

    # default `log_dir` is "runs" - we'll be more specific here
    writer = SummaryWriter('runs/mnist')

    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']

    """
    previously if we had 100 examples and 4 possible classes, we got 100x4 matrix of labels
    where each row would be [0,1,0,0] meaning that the correct class label = 2
    (as only 2nd index is equal to 1)

    Now, pytorch does not like this format. If there are 100 examples, pytorch wants a 100x1
    vector where the element in each row is the correct class directly

    To do this, we can just take the argmax of each row of our train_y, valid_y and test_y matrices

    the train_x has a shape of (10800,1024) viz 10800 images of 32x32 = 1024 size each
    the train_y has a shape of (10800, 36) viz 10800 images and 36 possible labels for each
    Note. out of the 36 possible labels for each image
    only 1 element will be = 1, remainaing = 0
    """

    # convert the labels to a Nx1 format (see comments above)
    # train_y = np.expand_dims(np.argmax(train_y, axis=1), axis=1)
    # valid_y = np.expand_dims(np.argmax(valid_y, axis=1), axis=1)
    # test_y = np.expand_dims(np.argmax(test_y, axis=1), axis=1)
```

```
train_y_arg = np.argmax(train_y, axis=1)
valid_y_arg = np.argmax(valid_y, axis=1)
test_y_arg = np.argmax(test_y, axis=1)

# convert every image and label to a torch tensor
train_xt = torch.from_numpy(train_x)
train_yt = torch.from_numpy(train_y_arg)
val_xt = torch.from_numpy(valid_x)
val_yt = torch.from_numpy(valid_y_arg)
test_xt = torch.from_numpy(test_x)
test_yt = torch.from_numpy(test_y_arg)

# check for GPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)

max_iters = 50
# pick a batch size, learning rate
batch_size = 128
learning_rate = 1e-3

# initialize your custom dataset to be used with the dataloader
train_dataset = TensorDataset(train_xt, train_yt)
val_dataset = TensorDataset(val_xt, val_yt)
test_dataset = TensorDataset(test_xt, test_yt)

# create dataloader objects for each of the above datasets
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True, num_workers=4)

net = SushNet()
net.to(device)

# criterion = nn.CrossEntropyLoss()
criterion = nn.NLLLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)

# iterate over epochs (max_iters = epochs)
for epoch in range(max_iters): # loop over the dataset multiple times

    running_loss = 0.0
    total = 0
    correct = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)
        inputs = inputs.to(torch.float32)
        # labels = labels.to(torch.float32)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 20 == 0:  # print every 200 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 20:.3f}')

            # ...log the running loss
```

```
            writer.add_scalar('training loss',
                              running_loss / 20,
                              epoch * len(train_loader) + i)
            running_loss = 0.0

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            print(f"traning accuracy for epoch {epoch} and batch {i} is {(100 * correct //
                total)}")\

            # ...log the running loss
            writer.add_scalar('training accuracy',
                              (100 * correct // total),
                              epoch * len(train_loader) + i)
            running_loss = 0.0

    #============================ Run Validation Accuracy Pass ==========================
    correct = 0
    total = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        inputs, labels = val_xt.to(device), val_yt.to(device)
        inputs = inputs.to(torch.float32)
        # labels = labels.to(torch.float32)

        # calculate correct label predictions
        outputs = net(inputs)
        val_loss = criterion(outputs, labels)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    # ...log the running loss
    writer.add_scalar('validation loss',
                      val_loss / 20,
                      epoch)
    running_loss = 0.0

    print("Validation accuracy for this epoch is", (100 * correct // total))
    # ...log the running loss
    writer.add_scalar('validation accuracy',
                          (100 * correct // total),
                      epoch)
    running_loss = 0.0



print('Finished Training')

# save the trained model to disk
PATH = './sush_net_fc.pth'
torch.save(net.state_dict(), PATH)



# reload the network to measure test accuracy
net = SushNet()
net.load_state_dict(torch.load(PATH))
net.to(device)

"""
# USING NUMPY
# Run Validation Accuracy pass
```

```
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        inputs, labels = val_xt.to(device), val_yt.to(device)
        inputs = inputs.to(torch.float32)
        labels = labels.to(torch.float32)

        # calculate outputs by running images through the network
        outputs = net(inputs)
        outputs = outputs.cpu()
        # the class with the highest energy is what we choose as prediction

        nump_outputs = outputs.numpy()
        print("outputs shape is", nump_outputs.shape)

        labels = labels.cpu()
        nump_labels = labels.numpy()
        print("labels size is", labels.shape)

        loss, acc = compute_loss_and_acc(valid_y, nump_outputs)
        print("Validation Loss is", loss)
        print("Validation Accuracy is", acc)
    """


    #============================== Run Test Accuracy Pass ==========================
    correct = 0
    total = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        inputs, labels = test_xt.to(device), test_yt.to(device)
        inputs = inputs.to(torch.float32)
        # labels = labels.to(torch.float32)

        # calculate outputs by running images through the network
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print("Test accuracy is", (100 * correct // total))


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

    # probs is the output of our softmax function
    # we'll define our loss as the -1*(unnormalized log of these probs)

    assert y.shape == probs.shape
    log_probs = y * (probs)
    loss = -(np.sum(log_probs))

    # calculate accuracy over all training examples
    true_positives = (np.where(np.argmax(y, axis=1) == np.argmax(probs, axis=1)))[0].shape[0]
    acc = true_positives / probs.shape[0]

    return loss, acc

if __name__ == '__main__':
    main()
```

Network Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SushNet(nn.Module):

    def __init__(self):
        # inherit the necessary superclass
        super().__init__()

        # initialize layers

        # LAYER 1
        # the train_x shape of (10800,1024) will cause
        # the weights to have shape (1024 x hidden_layer_size)
        # (hidden layer size is arbitrary and here = 64)

        # LAYER 2
        # here too the weights will have a size which maps the hidden layer
        # to the output layer. The weights shape will be (hidden_size, 36)
        # where 36 = number of total possible labels
        self.layer1_shape = (1024, 64)
        self.layer2_shape = (64, 36)

        # Define Operations
        # an affine operation: y = XW + b
        self.fc1 = nn.Linear(self.layer1_shape[0], self.layer1_shape[1])
        self.fc2 = nn.Linear(self.layer2_shape[0], self.layer2_shape[1])

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = F.log_softmax(self.fc2(x), dim=1)
        return x

    # Backward function doesn't need explicit functions, Autograd will pick up
    # required gradients for each layer on its own
```

The Training and Validation accuracy for the above is shown below. Test accuracy was 77%

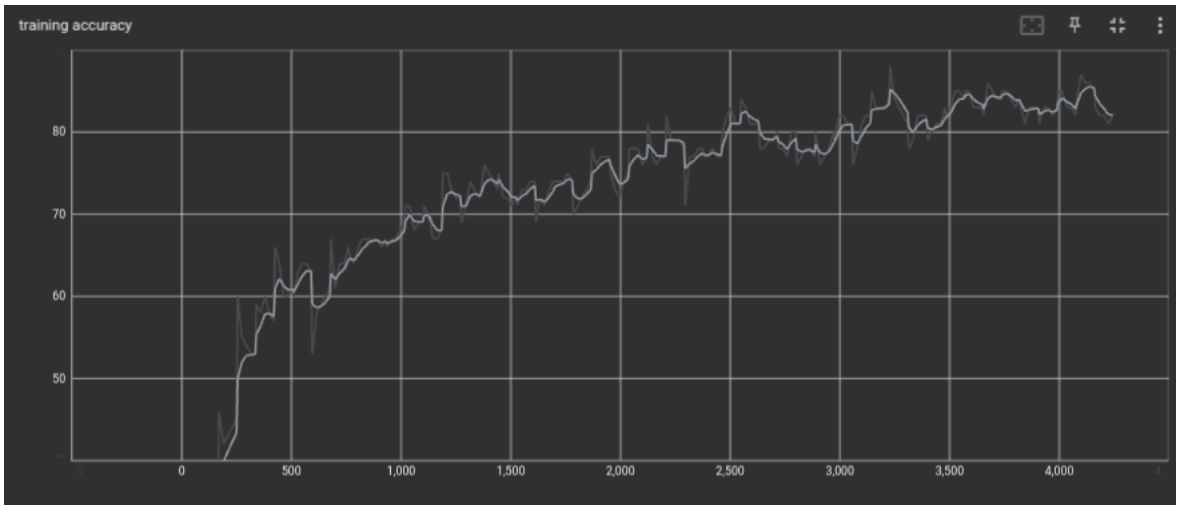


Figure 17: Training Accuracy

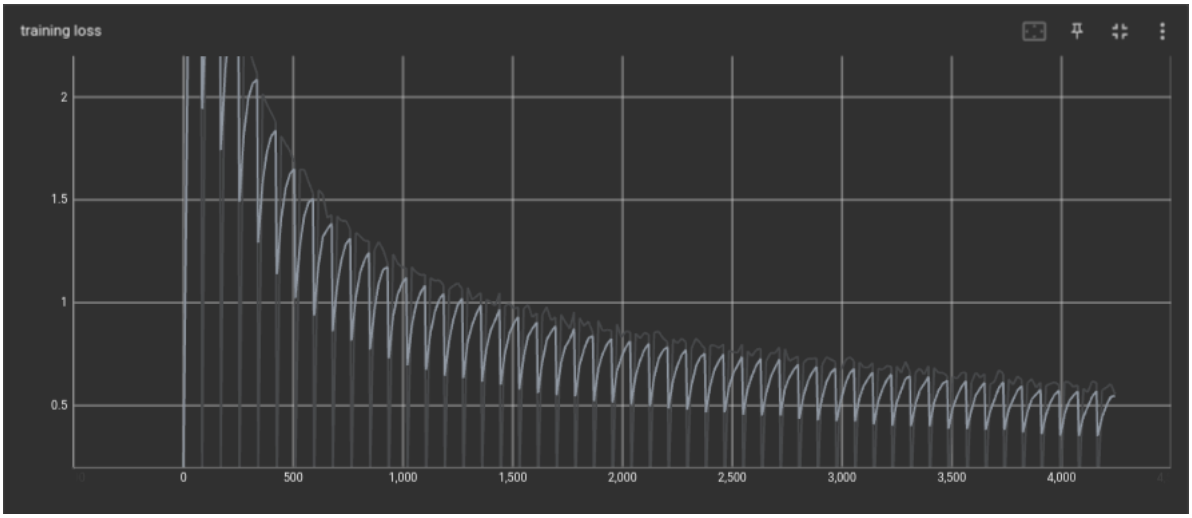


Figure 18: Training Loss

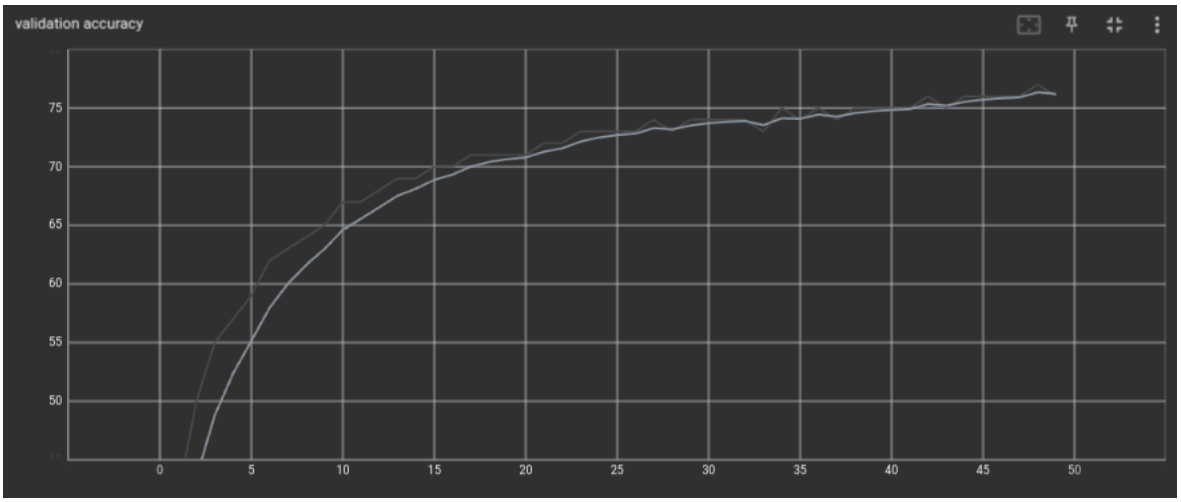


Figure 19: Validation Accuracy

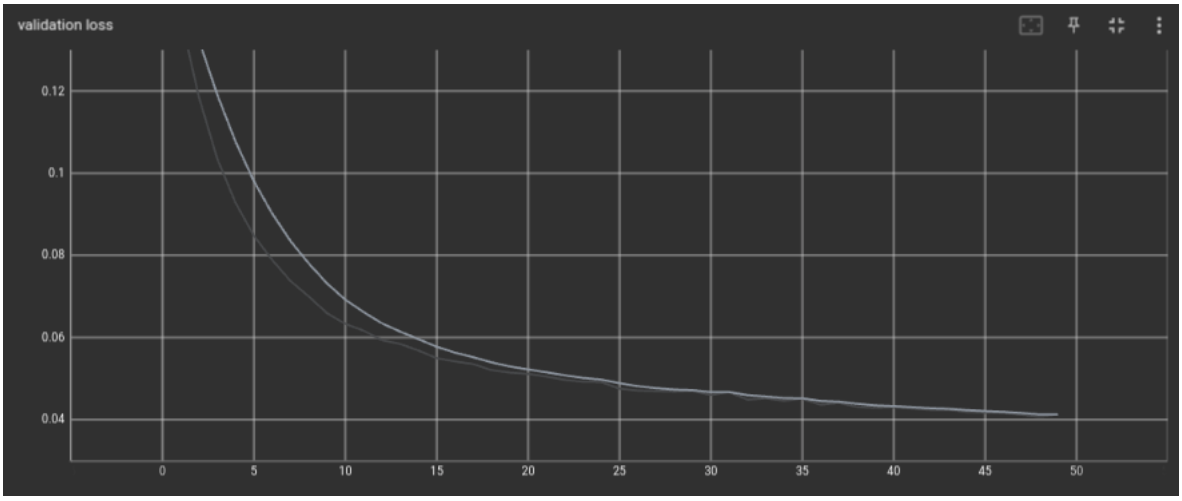


Figure 20: Validation Loss

6.1.2 Convolution Network on given NIST36

The main part of the code remains the same, just the network was changed as shown below:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SushConvNet(nn.Module):

    def __init__(self):
        # inherit the necessary superclass
        super().__init__()

        # initialize layers

        # LAYER 1
        # the train_x shape of (10800,1024) will cause
        # the weights to have shape (1024 x hidden_layer_size)
        # (hidden layer size is arbitrary and here = 64)

        # LAYER 2
        # here too the weights will have a size which maps the hidden layer
        # to the output layer. The weights shape will be (hidden_size, 36)
        # where 36 = number of total possible labels
        self.layer1_shape = (400, 64)
        self.layer2_shape = (64, 36)

        # Define Operations
        # conv operations

        # conv2d (in_channels, out_channels, kernel size)
        self.conv1 = nn.Conv2d(1, 6, 5)
        # maxpool2d (kernel_size, stride), automatically zero pads
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # an affine operation: y = XW + b
        self.fc1 = nn.Linear(self.layer1_shape[0], self.layer1_shape[1])
        self.fc2 = nn.Linear(self.layer2_shape[0], self.layer2_shape[1])

    def forward(self, x):
        # reshape to N,C,H,W
        x = x.view(-1, 1, 32, 32)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = torch.sigmoid(self.fc1(x))
        x = F.log_softmax(self.fc2(x), dim=1)
        return x

    # Backward function doesn't need explicit functions, Autograd will pick up
    # required gradients for each layer on its own
```

The Training and Validation accuracy for the above is shown below. Test accuracy was 89%

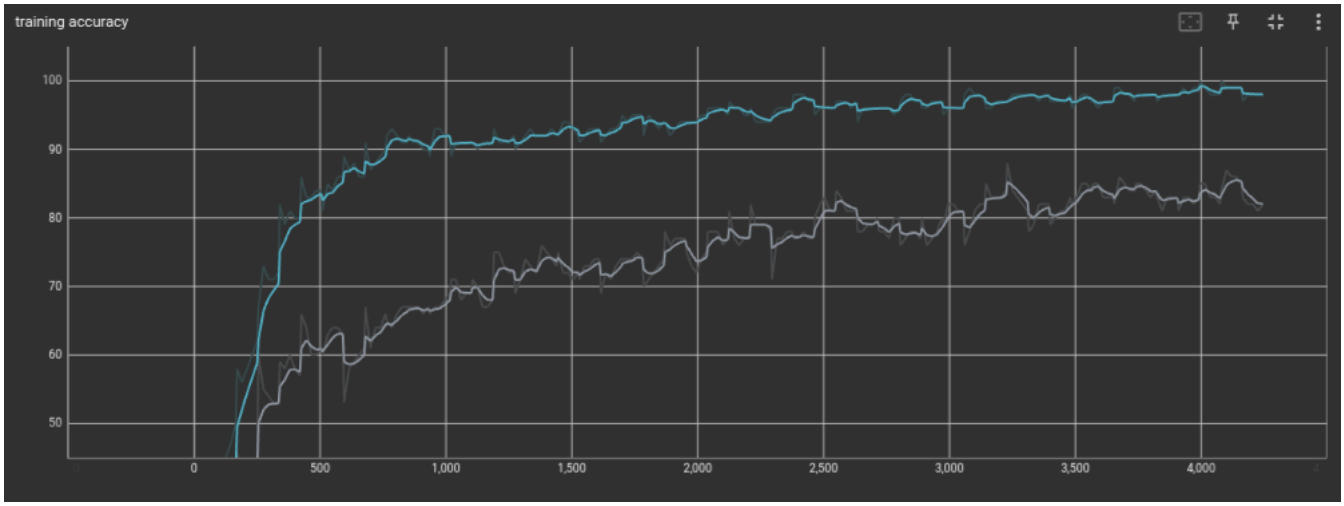


Figure 21: Training Accuracy

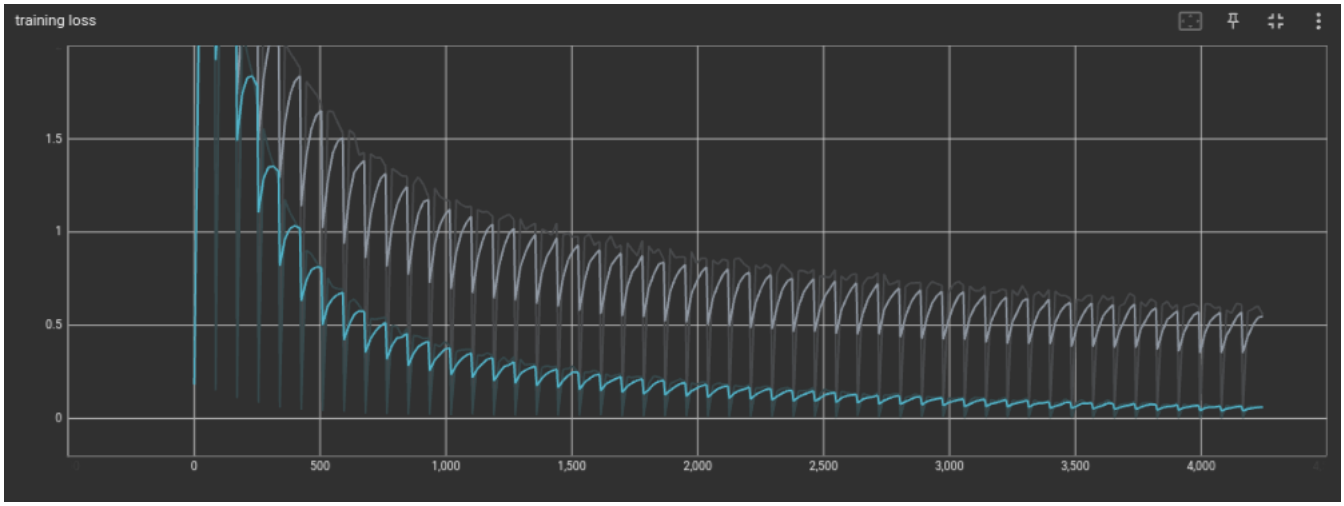


Figure 22: Training Loss

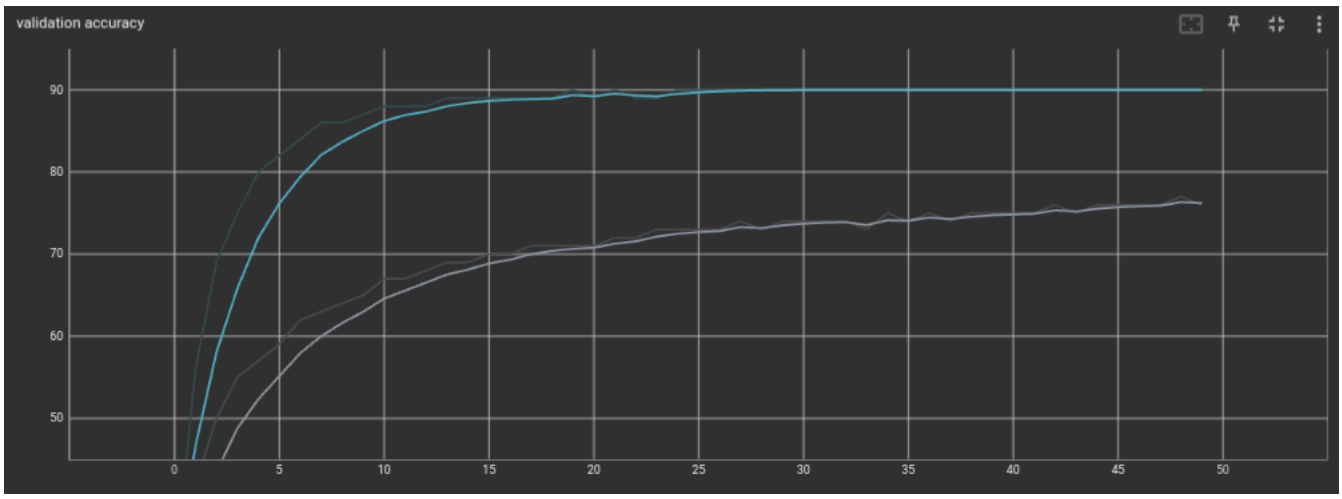


Figure 23: Validation Accuracy

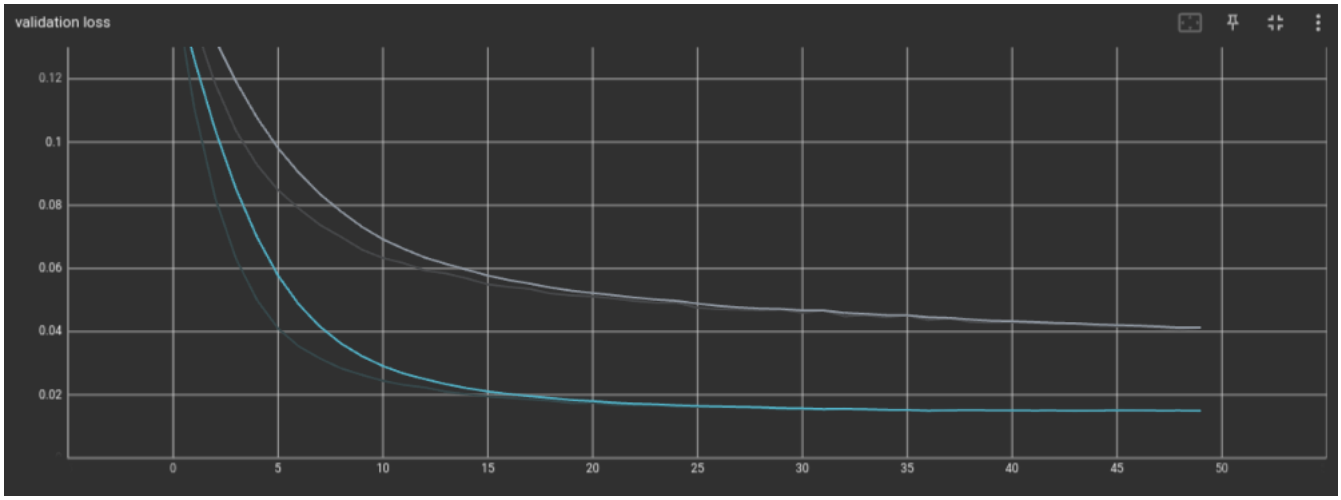


Figure 24: Validation Loss

6.1.3 CIFAR10 Dataset

Main Code

```
import sys
import numpy as np
import matplotlib.pyplot as plt

import torch
import torchvision
import torch.optim as optim
import torch.nn as nn

from NN import *
from torch.utils.data import TensorDataset, DataLoader
import torchvision.transforms as transforms

from torch.utils.tensorboard import SummaryWriter

# add models folder to path
sys.path.insert(0,'/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/python/network_models')
from q6_cifar_conv_model import SushConvNet_CIFAR


def main():

    # check for GPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    print(device)

    max_iters = 5
    # pick a batch size, learning rate
    batch_size = 100
    learning_rate = 1e-3

    # The output of torchvision datasets are PILImage images of range [0, 1]
    # We transform them to Tensors of normalized range [-1, 1]
    transform = transforms.Compose(
                                [transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                ])

    # default `log_dir` is "runs" - we'll be more specific here
    writer = SummaryWriter('runs/cifar')

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

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

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

    net = SushConvNet_CIFAR()
    net.to(device)

    # criterion = nn.CrossEntropyLoss()
    criterion = nn.NLLLoss()
    optimizer = optim.Adam(net.parameters(), lr=learning_rate)
```

```
# iterate over epochs (max_iters = epochs)
for epoch in range(max_iters): # loop over the dataset multiple times

    running_loss = 0.0
    total = 0
    correct = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 20 == 0:  # print every 200 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 20:.3f}')

            # ...log the running loss
            writer.add_scalar('training loss',
                              running_loss / 20,
                              epoch * len(train_loader) + i)
            running_loss = 0.0

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            print(f"traning accuracy for epoch {epoch} and batch {i} is {(100 * correct //
                total)}")\

            # ...log the running loss
            writer.add_scalar('training accuracy',
                               (100 * correct // total),
                              epoch * len(train_loader) + i)
            running_loss = 0.0

    correct_acc = 0
    total_acc = 0
    # again no gradients needed
    with torch.no_grad():
        i = 0
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            loss += criterion(outputs, labels)

            _, predicted = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predicted == labels).sum().item()
            i += 1

        # ...log the running loss
        writer.add_scalar('validation loss',
                          loss / i,
                          epoch)


        # ...log the running loss
```

```
            writer.add_scalar('validation accuracy',
                              (100 * correct_acc // total_acc),
                              epoch)
            print("validation accuracy is", (100 * correct // total))



    # print overall accuracy
    print(f'Accuracy of the network on test images: {100 * correct // total} %')

    print('Finished Training')

    # save the trained model to disk
    PATH = './cifar_net.pth'
    torch.save(net.state_dict(), PATH)


    # reload the network to measure test accuracy
    net = SushConvNet_CIFAR()
    net.load_state_dict(torch.load(PATH))
    net.to(device)

    #================================== Run Test Accuracy Pass ==============================
    # prepare to count predictions for each class
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    correct_acc = 0
    total_acc = 0
    # again no gradients needed
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            _, predicted = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predicted == labels).sum().item()
            # collect the correct predictions for each class
            for label, prediction in zip(labels, predicted):
                if label == prediction:
                    correct_pred[classes[label]] += 1
                total_pred[classes[label]] += 1

    # print overall accuracy
    print(f'Accuracy of the network on test images: {100 * correct_acc // total_acc} %')

    # print accuracy for each class
    for classname, correct_count in correct_pred.items():
        accuracy = 100 * float(correct_count) / total_pred[classname]
        print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')

    # ========================================================================================

if __name__ == '__main__':
    main()
```

Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SushConvNet_CIFAR(nn.Module):
```

```
def __init__(self):
    # inherit the necessary superclass
    super().__init__()

    # initialize layers

    # LAYER 1
    # the train_x shape of (10800,1024) will cause
    # the weights to have shape (1024 x hidden_layer_size)
    # (hidden layer size is arbitrary and here = 64)

    # LAYER 2
    # here too the weights will have a size which maps the hidden layer
    # to the output layer. The weights shape will be (hidden_size, 36)
    # where 36 = number of total possible labels
    self.layer1_shape = (400, 64)
    self.layer2_shape = (64, 10)

    # Define Operations
    # conv operations

    # conv2d (in_channels, out_channels, kernel size)
    self.conv1 = nn.Conv2d(3, 6, 5)
    # maxpool2d (kernel_size, stride), automatically zero pads
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)

    # an affine operation: y = XW + b
    self.fc1 = nn.Linear(self.layer1_shape[0], self.layer1_shape[1])
    self.fc2 = nn.Linear(self.layer2_shape[0], self.layer2_shape[1])

def forward(self, x):
    # reshape to N,C,H,W
    x = x.view(-1, 3, 32, 32)
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = torch.flatten(x, 1) # flatten all dimensions except batch
    x = torch.sigmoid(self.fc1(x))
    x = F.log_softmax(self.fc2(x), dim=1)
    return x

# Backward function doesn't need explicit functions, Autograd will pick up
# required gradients for each layer on its own
```

The Training and Validation accuracy for the above is shown below. Test accuracy was 55% after **only 5 epochs**

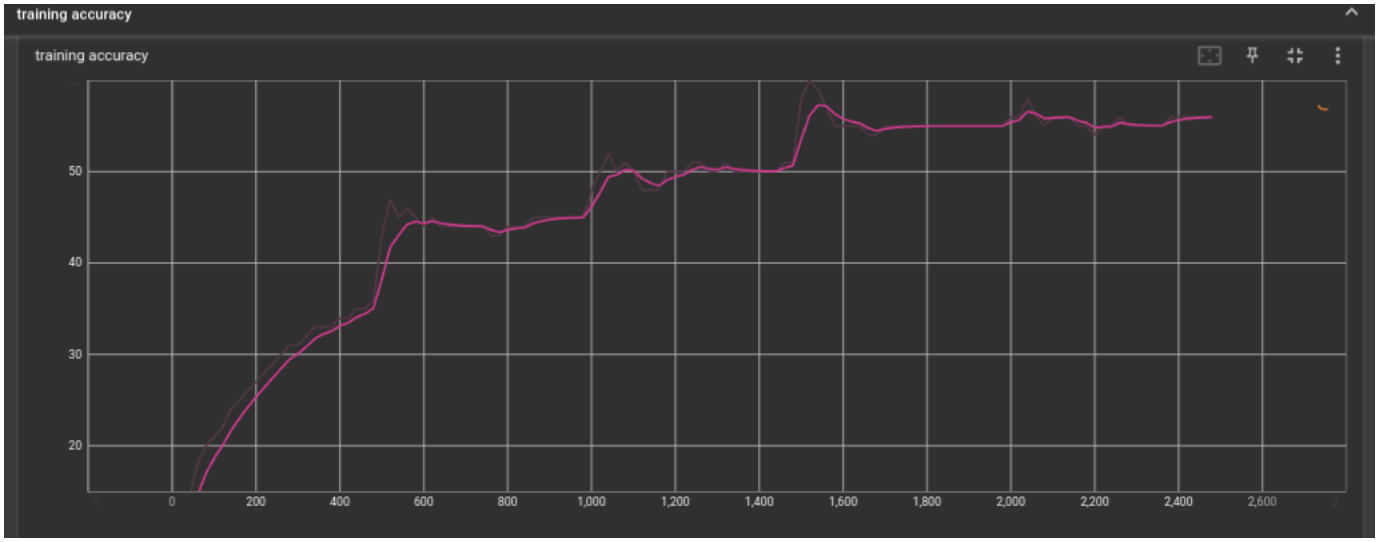


Figure 25: Training Accuracy

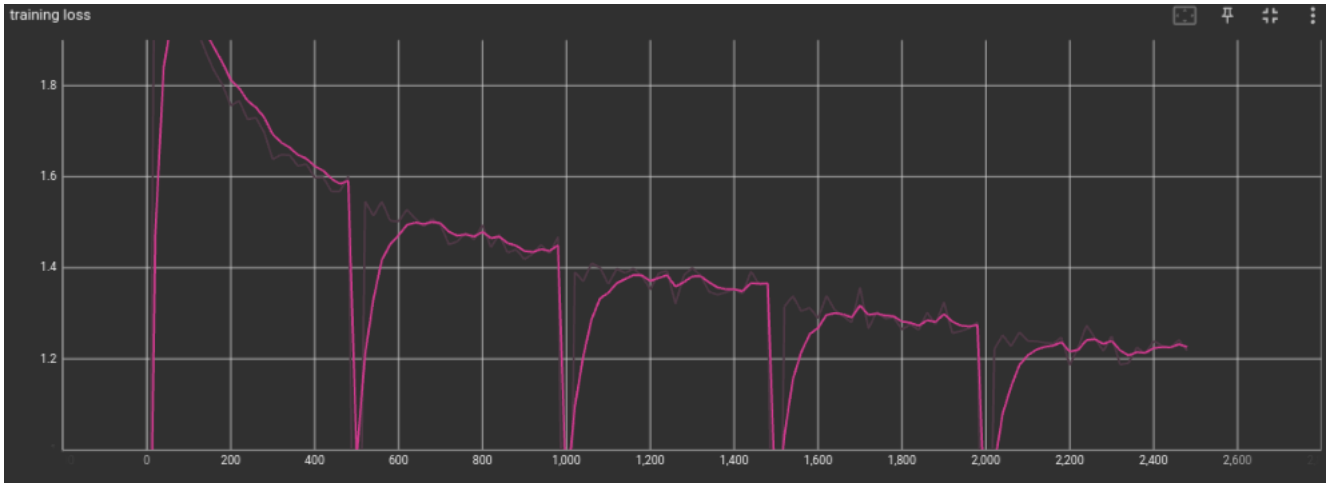


Figure 26: Training Loss

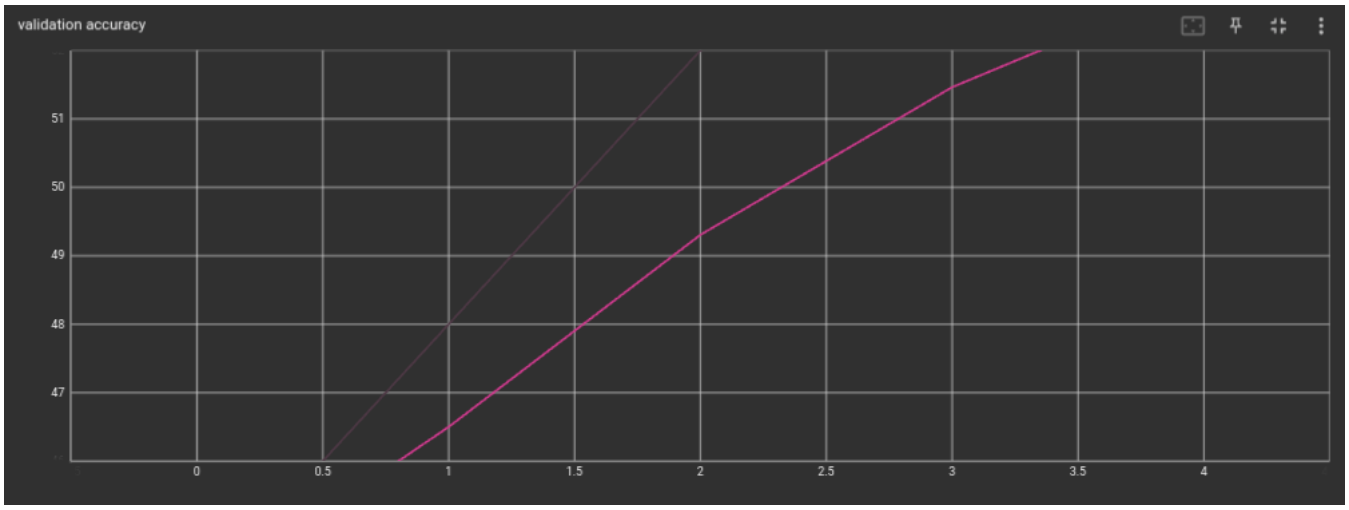


Figure 27: Validation Accuracy

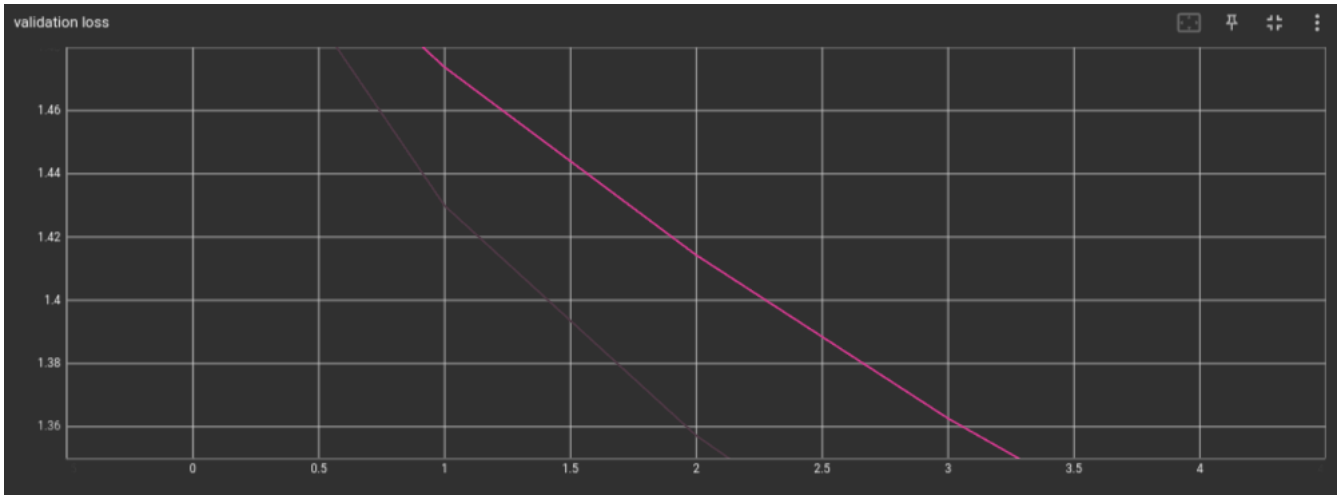


Figure 28: Validation Loss

6.1.4 SUN Dataset

Main Code

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import sys

import torch
import torchvision
import torch.optim as optim
import torch.nn as nn

from mpl_toolkits.axes_grid1 import ImageGrid
from NN import *
from torch.utils.data import DataLoader
import torchvision.transforms as transforms

from torch.utils.tensorboard import SummaryWriter

# add models folder to path
sys.path.insert(0,'/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/python/network_models')
from q6_SUN_model import SushConvNet_SUN

def main():

    # check for GPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    print(device)

    max_iters = 20
    # pick a batch size, learning rate
    batch_size = 100
    learning_rate = 1e-3

    # The output of torchvision datasets are PILImage images of range [0, 1]
    # We transform them to Tensors of normalized range [-1, 1]
    transform = transforms.Compose(
                                [transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                                 transforms.Resize((256,256))
                                ])

    # default `log_dir` is "runs" - we'll be more specific here
    writer = SummaryWriter('runs/SUN')

    #! Change from datasets.SUN397 to ImageFolder
    trainset = torchvision.datasets.ImageFolder(
                                            root='/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_1/data/pytorch_s
                                            transform=transform
                                           )
    train_loader = DataLoader(trainset, batch_size=batch_size,
                              shuffle=True, num_workers=4, pin_memory=True)

    testset = torchvision.datasets.ImageFolder(
                                           root='/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_1/data/pytorch_st
                                           transform=transform
                                          )
    test_loader = DataLoader(testset, batch_size=batch_size,
                              shuffle=True, num_workers=4, pin_memory=True)


    classes = ('acquarium', 'desert', 'highway', 'kitchen', 'laundromat',
               'park', 'waterfall', 'windmill')
```

```
net = SushConvNet_SUN()
net.to(device)

# criterion = nn.CrossEntropyLoss()
criterion = nn.NLLLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)

# iterate over epochs (max_iters = epochs)
for epoch in range(max_iters): # loop over the dataset multiple times

    running_loss = 0.0
    total = 0
    correct = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 20 == 0:  # print every 200 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 20:.3f}')

            # ...log the running loss
            writer.add_scalar('training loss',
                              running_loss / 20,
                              epoch * len(train_loader) + i)
            running_loss = 0.0

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            print(f"traning accuracy for epoch {epoch} and batch {i} is {(100 * correct //
                total)}")\

            # ...log the running accuracy
            writer.add_scalar('training accuracy',
                               (100 * correct // total),
                              epoch * len(train_loader) + i)
            running_loss = 0.0

    correct_acc = 0
    total_acc = 0
    # again no gradients needed
    i = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            _, predictions = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predictions == labels).sum().item()
            i += 1
```

```
            writer.add_scalar('training loss',
                              loss / i,
                              epoch)

            # ...log the running accuracy
            writer.add_scalar('training accuracy',
                              (100 * correct_acc // total_acc),
                          epoch)



        # print overall accuracy
        print(f'Accuracy of the network on test images: {100 * correct // total} %')

    print('Finished Training')

    # save the trained model to disk
    PATH = './sun_net.pth'
    torch.save(net.state_dict(), PATH)


    # reload the network to measure test accuracy
    net = SushConvNet_SUN()
    net.load_state_dict(torch.load(PATH))
    net.to(device)

    #================================== Run Test Accuracy Pass ==============================
    # prepare to count predictions for each class
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    correct_acc = 0
    total_acc = 0
    # again no gradients needed
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            _, predictions = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predictions == labels).sum().item()
            # collect the correct predictions for each class
            for label, prediction in zip(labels, predictions):
                if label == prediction:
                    correct_pred[classes[label]] += 1
                total_pred[classes[label]] += 1

    # print overall accuracy
    print(f'Accuracy of the network on test images: {100 * correct_acc // total_acc} %')

    # print accuracy for each class
    for classname, correct_count in correct_pred.items():
        accuracy = 100 * float(correct_count) / total_pred[classname]
        print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')

    # ========================================================================================

if __name__ == '__main__':
    main()
```

Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SushConvNet_SUN(nn.Module):

    def __init__(self):
        # inherit the necessary superclass
        super().__init__()

        # initialize layers

        # LAYER 1
        # the train_x shape of (10800,1024) will cause
        # the weights to have shape (1024 x hidden_layer_size)
        # (hidden layer size is arbitrary and here = 64)

        # LAYER 2
        # here too the weights will have a size which maps the hidden layer
        # to the output layer. The weights shape will be (hidden_size, 36)
        # where 36 = number of total possible labels
        self.layer1_shape = (59536, 1200)
        self.layer2_shape = (1200, 8)

        # Define Operations
        # conv operations

        # conv2d (in_channels, out_channels, kernel size)
        self.conv1 = nn.Conv2d(3, 6, 5)
        # maxpool2d (kernel_size, stride), automatically zero pads
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # an affine operation: y = XW + b
        self.fc1 = nn.Linear(self.layer1_shape[0], self.layer1_shape[1])
        self.fc2 = nn.Linear(self.layer2_shape[0], self.layer2_shape[1])

    def forward(self, x):
        # reshape to N,C,H,W
        x = x.view(-1, 3, 256, 256)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = torch.sigmoid(self.fc1(x))
        x = F.log_softmax(self.fc2(x), dim=1)
        return x

    # Backward function doesn't need explicit functions, Autograd will pick up
    # required gradients for each layer on its own
```

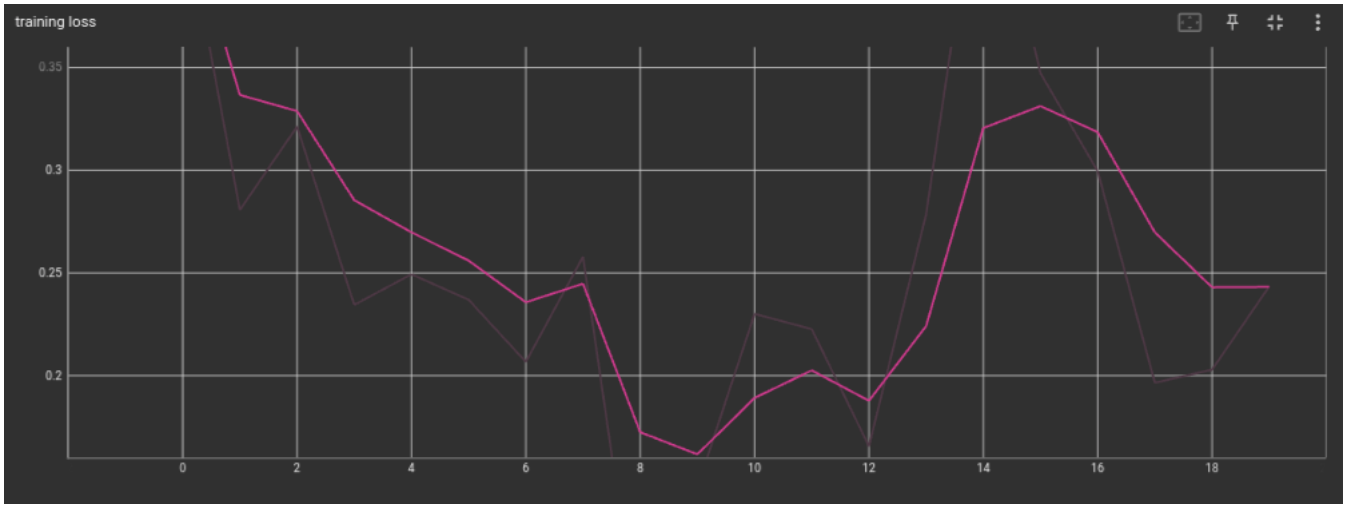


Figure 29: Validation Loss

6.2 Fine Tuning

Using SqueezeNet1_1 on Flower_102 Dataset

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

import torch
import torchvision
import torch.optim as optim
import torch.nn as nn

from NN import *
from torch.utils.data import DataLoader

from torch.utils.tensorboard import SummaryWriter
from torchvision.models import squeezenet1_1, SqueezeNet1_1_Weights

model = squeezenet1_1(pretrained=True)

def main():

    # check for GPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    print(device)

    max_iters = 50
    # pick a batch size, learning rate
    batch_size = 128
    learning_rate = 1e-4

    # The output of torchvision datasets are PILImage images of range [0, 1]
    # We transform them to Tensors of normalized range [-1, 1]
    transform = SqueezeNet1_1_Weights.IMAGENET1K_V1.transforms()

    # default `log_dir` is "runs" - we'll be more specific here
    writer = SummaryWriter('runs/flower_squeezenet')
    data_path = '/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/data/oxford-flowers102/'

    trainset = torchvision.datasets.ImageFolder(
                                          root=os.path.join(data_path, 'train'),
                                          transform=transform
                                         )
    train_loader = DataLoader(trainset, batch_size=batch_size,
                              shuffle=True, num_workers=4, pin_memory=True)

    valset = torchvision.datasets.ImageFolder(
                                         root=os.path.join(data_path, 'val'),
                                         transform=transform
                                        )
    val_loader = DataLoader(valset, batch_size=batch_size, \
                              shuffle=True, num_workers=4, pin_memory=True)

    testset = torchvision.datasets.ImageFolder(
                                         root=os.path.join(data_path, 'test'),
                                         transform=transform
                                        )
    test_loader = DataLoader(testset, batch_size=batch_size,
                              shuffle=True, num_workers=4, pin_memory=True)

    # set all internal weights to fixed
    for param in model.parameters():
        param.requires_grad = False
```

```
final_conv = nn.Conv2d(512, 102, kernel_size=1)
model.classifier = nn.Sequential(
       nn.Dropout(p=0.5), final_conv, nn.ReLU(inplace=True), nn.AdaptiveAvgPool2d((1, 1))
    )

# set classifier weights to true
for param in model.parameters():
    param.requires_grad = True

net = model
net.to(device)

criterion = nn.CrossEntropyLoss()
# criterion = nn.NLLLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate) #, weight_decay=0.002)

# iterate over epochs (max_iters = epochs)
for epoch in range(max_iters): # loop over the dataset multiple times

    running_loss = 0.0
    total = 0
    correct = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 20 == 0:  # print every 200 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 20:.3f}')

            # ...log the running loss
            writer.add_scalar('training loss',
                          running_loss / 20,
                          epoch * len(train_loader) + i)
            running_loss = 0.0

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)

            correct += (predicted == labels).sum().item()

            print(f"traning accuracy for epoch {epoch} \
                   and batch {i} is {(100 * correct // total)}")\

            # ...log the running accuracy
            writer.add_scalar('training accuracy',
                           (100 * correct // total),
                          epoch * len(train_loader) + i)
            running_loss = 0.0

    #================================= Run Val accuracy pass ============================#
    # correct_acc = 0
    # total_acc = 0
```

```
        # # again no gradients needed
        # with torch.no_grad():
        #     for data in val_loader:
        #         inputs, labels = data[0].to(device), data[1].to(device)
        #         outputs = net(inputs)
        #         _, predictions = torch.max(outputs, 1)
        #         total_acc += labels.size(0)
        #         correct_acc += (predictions == labels).sum().item()

        # # print overall accuracy
        # print(f'Accuracy of the network on val images for epoch{epoch}: {100 * correct // total}
            %')

    print('Finished Training')

    # save the trained model to disk
    PATH = './flower_squeeze_net.pth'
    torch.save(net.state_dict(), PATH)


    # reload the network to measure test accuracy
    net = model.eval()
    net.load_state_dict(torch.load(PATH))
    net.to(device)

    #================================== Run Test accuracy pass ================================#
    correct_acc = 0
    total_acc = 0
    # again no gradients needed
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            _, predictions = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predictions == labels).sum().item()

    # print overall accuracy
    print(f'Accuracy of the network on val images: {100 * correct / total} %')



if __name__ == '__main__':
    main()
```

Using Conventional Architechture

```
# add models folder to path
sys.path.insert(0,'/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/python/network_models')

from q6_flower_model import SushConvNet_Flower

def main():

    # check for GPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    print(device)

    max_iters = 5
    # pick a batch size, learning rate
    batch_size = 128
    learning_rate = 1e-4

    # The output of torchvision datasets are PILImage images of range [0, 1]
```

```
# We transform them to Tensors of normalized range [-1, 1]
transform = torchvision.transforms.Compose(
                            [torchvision.transforms.ToTensor(),
                             torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                                 0.5)),
                             torchvision.transforms.Resize((256,256))
                            ])

# default `log_dir` is "runs" - we'll be more specific here
writer = SummaryWriter('runs/flower')
data_path = '/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/data/oxford-flowers102/'

trainset = torchvision.datasets.ImageFolder(
                                      root=os.path.join(data_path, 'train'),
                                      transform=transform
                                     )
train_loader = DataLoader(trainset, batch_size=batch_size,
                          shuffle=True, num_workers=4, pin_memory=True)

valset = torchvision.datasets.ImageFolder(
                                     root=os.path.join(data_path, 'val'),
                                     transform=transform
                                    )
val_loader = DataLoader(valset, batch_size=batch_size, \
                          shuffle=True, num_workers=4, pin_memory=True)

testset = torchvision.datasets.ImageFolder(
                                     root=os.path.join(data_path, 'test'),
                                     transform=transform
                                    )
test_loader = DataLoader(testset, batch_size=batch_size,
                          shuffle=True, num_workers=4, pin_memory=True)

net = SushConvNet_Flower()
net.to(device)

# criterion = nn.CrossEntropyLoss()
criterion = nn.NLLLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)

# iterate over epochs (max_iters = epochs)
for epoch in range(max_iters): # loop over the dataset multiple times

    running_loss = 0.0
    total = 0
    correct = 0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 20 == 0:  # print every 200 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 20:.3f}')

            # ...log the running loss
```

```
                writer.add_scalar('training loss',
                                  running_loss / 20,
                                  epoch * len(train_loader) + i)
                running_loss = 0.0

                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

                print(f"traning accuracy for epoch {epoch} \
                       and batch {i} is {(100 * correct // total)}")\

                # ...log the running accuracy
                writer.add_scalar('training accuracy',
                                  (100 * correct // total),
                                  epoch * len(train_loader) + i)
                running_loss = 0.0

        #================================== Run Val accuracy pass ==============================#
        correct_acc = 0
        total_acc = 0
        # again no gradients needed
        with torch.no_grad():
            for data in val_loader:
                inputs, labels = data[0].to(device), data[1].to(device)
                outputs = net(inputs)
                _, predictions = torch.max(outputs, 1)
                total_acc += labels.size(0)
                correct_acc += (predictions == labels).sum().item()

        # print overall accuracy
        print(f'Accuracy of the network on val images for epoch{epoch}: {100 * correct // total}
            %')

    print('Finished Training')

    # save the trained model to disk
    PATH = './flower_net.pth'
    torch.save(net.state_dict(), PATH)


    # reload the network to measure test accuracy
    net = SushConvNet_Flower()
    net.load_state_dict(torch.load(PATH))
    net.to(device)

    #================================== Run Test accuracy pass ==============================#
    correct_acc = 0
    total_acc = 0
    # again no gradients needed
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = net(inputs)
            _, predictions = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predictions == labels).sum().item()

    # print overall accuracy
    print(f'Accuracy of the network on test images: {100 * correct // total} %')



if __name__ == '__main__':
    main()
```

Conventional Network used for Flower Classification

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SushConvNet_Flower(nn.Module):

    def __init__(self):
        # inherit the necessary superclass
        super().__init__()

        # initialize layers

        # LAYER 1
        # the train_x shape of (10800,1024) will cause
        # the weights to have shape (1024 x hidden_layer_size)
        # (hidden layer size is arbitrary and here = 64)

        # LAYER 2
        # here too the weights will have a size which maps the hidden layer
        # to the output layer. The weights shape will be (hidden_size, 36)
        # where 36 = number of total possible labels
        self.layer1_shape = (59536, 1200)
        self.layer2_shape = (1200, 102)

        # Define Operations
        # conv operations

        # conv2d (in_channels, out_channels, kernel size)
        self.conv1 = nn.Conv2d(3, 6, 5)
        # maxpool2d (kernel_size, stride), automatically zero pads
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # an affine operation: y = XW + b
        self.fc1 = nn.Linear(self.layer1_shape[0], self.layer1_shape[1])
        self.fc2 = nn.Linear(self.layer2_shape[0], self.layer2_shape[1])

    def forward(self, x):
        # reshape to N,C,H,W
        x = x.view(-1, 3, 256, 256)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        # print("x shape after conv is", x.shape)
        x = torch.sigmoid(self.fc1(x))
        x = F.log_softmax(self.fc2(x), dim=1)
        return x

    # Backward function doesn't need explicit functions, Autograd will pick up
    # required gradients for each layer on its own
```

6.2.1 Comparison between SqueezeNet and Conventional Network

Both Networks were run for **15 epochs**
The following was observed on the Conventional Architechture:

- Training Accuracy 28%
- Test Accuracy 19%

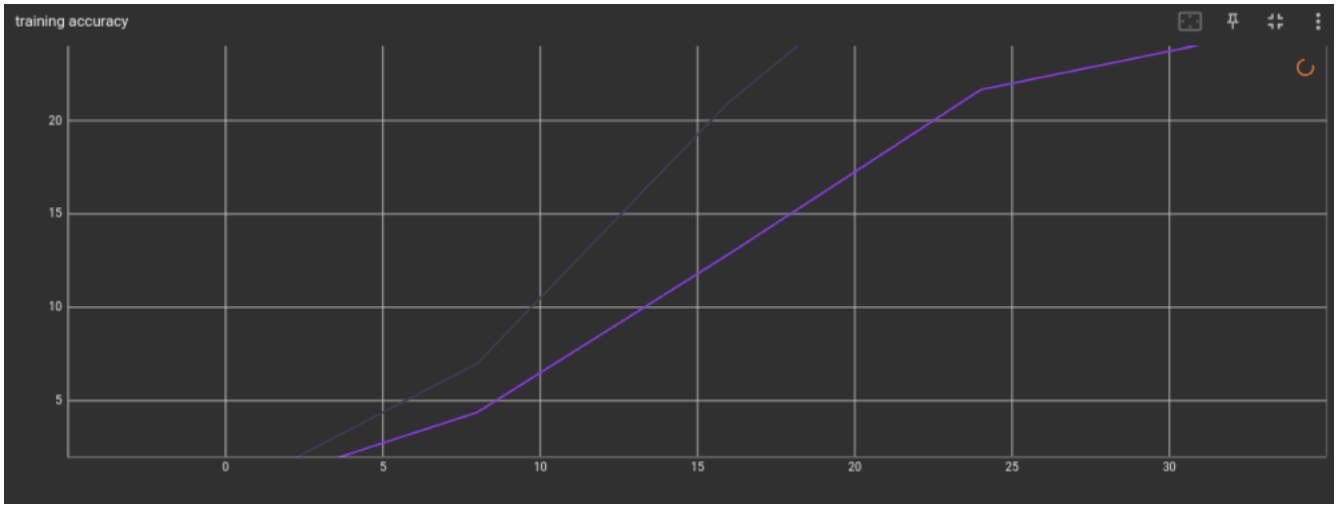


Figure 30: Training Accuracy

The following was observed with SqueezeNet:

- Training Accuracy 75%
- Test Accuracy 49%

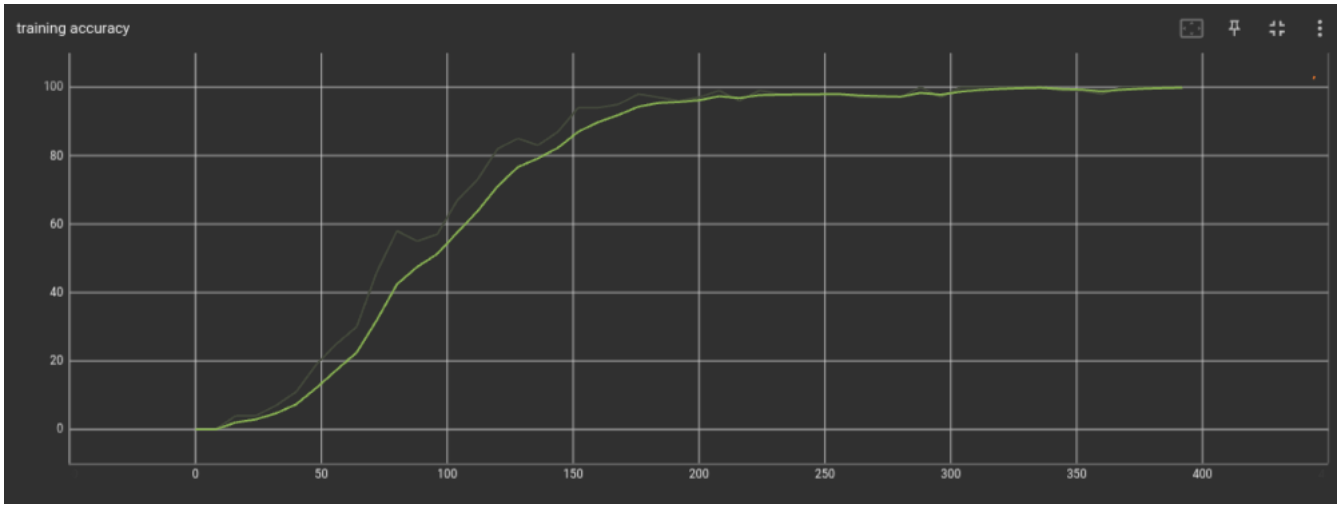


Figure 31: Training Accuracy

6.3 Extra Credit: Neural Networks in the Real World

- A video of a desk was chosen to as the desired label
- the 64x64 mini batch of ImageNet was downloaded

Figure 32: Example Frame from Video used

The accuracy on the ImageNet validation set was 15% while that on the test set did not go above 1%

```
import numpy as np
import matplotlib.pyplot as plt
import os
import sys
import scipy.io

import torch
import torchvision
import torch.optim as optim
import torch.nn as nn
import cv2

from NN import *
from torch.utils.data import DataLoader, TensorDataset

from torch.utils.tensorboard import SummaryWriter
from torchvision.models import efficientnet_b2, EfficientNet_B2_Weights

model = efficientnet_b2(parameters=EfficientNet_B2_Weights.IMAGENET1K_V1)

def main():

    # check for GPU
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    print(device)

    max_iters = 90
    # pick a batch size, learning rate
    batch_size = 64

    transform = EfficientNet_B2_Weights.IMAGENET1K_V1.transforms()

    # default `log_dir` is "runs" - we'll be more specific here
    writer = SummaryWriter('runs/desk_efficientnetb2')
    val_path = '/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/data/Imagenet64_val/val_data'
```

```
valid_data = np.load(val_path, allow_pickle=True)
valid_x, valid_y = valid_data['data'], valid_data['labels']
valid_y = np.array(valid_y)

# convert every image and label to a torch tensor
val_xt = torch.from_numpy(valid_x)
val_yt = torch.from_numpy(valid_y)

# check for GPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)

# initialize your custom dataset to be used with the dataloader
val_dataset = TensorDataset(val_xt, val_yt)

# create dataloader objects for each of the above datasets
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True, num_workers=4)

# set all internal weights to fixed
for param in model.parameters():
    param.requires_grad = False
net = model
net.to(device)


#================================== Run Val accuracy pass ================================#
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for i, data in enumerate(val_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)
        inputs = inputs.to(torch.float32)

        # reshape inputs to match the val_set shape of 64x64 images
        inputs = inputs.view(-1, 3, 64, 64)
        inputs = transform(inputs)

        # calculate correct label predictions
        outputs = net(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        print("Running Validation Accuracy", (100 * correct / total))

print("Validation accuracy is", (100 * correct // total))

#================================== Run Val accuracy pass ================================#
# """
# Here we will be testing images of a desk manually captured on a phone
# """
test_path = '/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/data/phone_test_images'
test_video_path = '/home/sush/CMU/Assignment_Sem_1/CV_A/Assignment_5/data/phone_test_video.mp4'

extract_frames_2(test_video_path, os.path.join(test_path, 'test'))

testset = torchvision.datasets.ImageFolder(
                                        root=test_path,
                                        transform=transform
                                       )
test_loader = DataLoader(testset, batch_size=16,
                         shuffle=True, num_workers=4, pin_memory=True)

correct_acc = 0
```

```
    total_acc = 0
    # again no gradients needed
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data[0].to(device), data[1].to(device)

            outputs = net(inputs)
            _, predictions = torch.max(outputs, 1)
            total_acc += labels.size(0)
            correct_acc += (predictions == labels).sum().item()

    # print overall accuracy
    print(f'Accuracy on custom phone camera images: {100 * correct_acc // total_acc} %')

def extract_frames_2(path, dump):
    currentframe = 0

    # Read the video from specified path
    cam = cv2.VideoCapture(path)

    while(True):

        # reading from frame
        success,frame = cam.read()

        currentframe += 1
        if success and currentframe%1 == 0 :
            # if video is still left continue creating images
            frame = frame[:,150:502,:]
            image = cv2.rotate(frame, cv2.ROTATE_180)
            cv2.imwrite(os.path.join(dump, (str(currentframe) + ".jpg")),image)

        if currentframe == 1000:
            break

    # Release all space and windows once done
    cam.release()


if __name__ == '__main__':
    main()
```

7 Collaborations

I thank Atharv Pulapaka for helping me do the theory questions