16-820 Advanced Computer Vision
Homework 4: Neural Networks for Recognition

Abhinandan Vellanki

1 Theory

1.1

The softmax function over a vector x is given by:

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

Adding a constant $c \in R$ to the vector x:

$$softmax(x+c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} \implies \frac{e^{x_i}e^{c}}{\sum_j e^{x_j}e^{c}} \implies \frac{e^{c}e^{x_i}}{e^{c}\sum_j e^{x_j}} \implies \frac{e^{x_i}}{\sum_j e^{x_j}} = softmax(x)$$

Hence, adding a constant to the vector does not change the output of applying a softmax function on it.

In practice, the constant added is: $-max(x)$. This is done for numerical stability so that the remaining values are only relative to each other and the function does not have to deal with very large numbers.

1.2

In softmax function, when $x \in R^d$, the range of each element is between 0 and 1. The sum over all elements is 1. Hence, one could say that "softmax takes an arbitrary real valued vector x and turns it into a probability distribution". Each step in the softmax function has a specific task:

- $s_i = e^{x_i}$: This step ensures all values are positive and emphasizes larger values
- softmax $= \frac{s_i}{\sum s_i}$: This step normalizes all values within 0 and 1.

Hence, The output probabilities maintain the relative scale of the input scores. Higher input scores result in higher probabilities.

1.3

In multi-Layer neural networks with a non-linear activation function, each layer performs linear combinations of its inputs:

$$Layer1 : Input = x; Output = y = Ax + b$$
$$Layer2 : Input = y; Output = z = Cy + d$$

This can be collapsed into a single layer:

$$Input = x; Output = (CA)x + (Cb + d)$$

Which is a linear combination of the inputs.

Hence, in such networks, all the layers can be collapsed into a single layer which performs a linear combination on the input of the form $y = Mx + c$ which is a linear regression equation.

1.4

Given sigmoid function:

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

Its gradient:

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\frac{1}{(1+e^{-x})} \\
\implies \frac{d}{dx}\frac{1}{(1+e^{-x})} &* \frac{d}{dx}(1+e^{-x}) \\
\implies \frac{e^{-x}}{(1+e^{-x})^2} &\implies \frac{e^{-x}+1-1}{(1+e^{-x})^2} \\
\implies \frac{1}{(1+e^{-x})}&(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}) \\
&\implies \sigma(x)(1-\sigma(x))
\end{aligned}$$

Hence, the derivative can be computed without having access to x

1.5

Given

$$\begin{aligned} y &= Wx + b \\ y_i &= \sum_{j=1}^{d} x_j W_{ij} + b_i \\ \frac{\partial J}{\partial y} &= \delta \in R^{k\times 1} \\ W &\in R^{k\times d} \\ x &\in R^{d\times 1} \\ b &\in R^{k\times 1} \end{aligned}$$

Solutions in scalar form:

$$\begin{aligned} \frac{\partial J}{\partial W} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} = \delta x \\ \frac{\partial J}{\partial x} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = \delta W \\ \frac{\partial J}{\partial b} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta \end{aligned}$$

Writing in matrix form:
Gradient with respect to W:

$$\begin{aligned} \frac{\partial J}{\partial W} &= \begin{bmatrix} \frac{\partial J}{\partial W_1} \\ . \\ . \\ . \\ \frac{\partial J}{\partial W_k} \end{bmatrix} \\ \frac{\partial J}{\partial W_{ij}} &= \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial W_{ij}} = \delta_i X_j \\ \implies \frac{\partial J}{\partial W_i} &= \delta_i X^T \\ \frac{\partial J}{\partial W} &= \begin{bmatrix} \delta_1 X^T \\ . \\ . \\ . \\ \delta_k X^T \end{bmatrix} \\ \implies \frac{\partial J}{\partial W} &= \delta X^T \end{aligned}$$

Gradient with respect to X:

$$\frac{\partial J}{\partial X} = \begin{bmatrix} \frac{\partial J}{\partial X_1} \\ . \\ . \\ . \\ \frac{\partial J}{\partial X_d} \end{bmatrix}$$

$$\implies \frac{\partial J}{\partial X_j} = \delta_1 W_{ij} + ... + \delta_k W_{kj}$$

$$\frac{\partial J}{\partial X} = \begin{bmatrix} \delta_1 & . & . & . & \delta_k \end{bmatrix} \begin{bmatrix} W_1 \\ . \\ . \\ . \\ W_k \end{bmatrix}$$

$$\implies \frac{\partial J}{\partial X} = (\delta^T W)^T$$

$$\implies \frac{\partial J}{\partial X} = W^T \delta$$

Gradient with respect to B:

$$\frac{\partial J}{\partial B} = \begin{bmatrix} \frac{\partial J}{\partial B_1} \\ . \\ . \\ . \\ \frac{\partial J}{\partial B_k} \end{bmatrix}$$

$$\frac{\partial J}{\partial B_i} = \frac{\partial J}{\partial y_i}.1 = \delta_i$$

$$\implies \frac{\partial J}{\partial B} = \begin{bmatrix} \delta_1 \\ . \\ . \\ \delta_k \end{bmatrix}$$

1.6

1. Sigmoid activationn function and vanishing gradient problem:
For inputs that are far away from zero, the sigmoid activation function has more or less constant values, i.e., very small or "vanishing" gradient. Hence, this derails the outputs of the hidden layers, rendering sigmoid activation function a poor choice.

2. Tanh activation function:
The output range of tanh function is (-1, 1), while for sigmoid function it is (0,1). Tanh's output is centered around zero, helping in faster convergence. making it a better choice over sigmoid.

3. Tanh has less of a vanishing gradient problem because it's distribution is more peaked, leading to larger gradients in more areas when compared to sigmoid function.

4. Tanh in terms of sigmoid:

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

2 Implement a Fully Connected Network

2.1 Network Initialization

2.1.1 Theory

It is a bad idea to initialize a network with all zeros because:

- This causes symmetry among neurons in each layer, so they effect the loss function in the same manner and hence, they are updated identically, making more than one neuron redundant.
- Due to the symmetry, the network does not learn diverse features and instead, learns the same features if symmetry is never broken due to the use of standard gradient descent methods.
- Neurons using ReLU activation functions might become "dead", i.e., always outputting zero, because their gradient is always zero.

Such a network would lose sensitivity to varying inputs and output constant results after training, no matter how much the input is changed. This is because only the bias terms are learned, essentially making it a baseline predictor. In classification tasks, such networks would only output the most common class encountered in training data, no matter what the input is.

2.1.2 Code

Using Xavier initialization:

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

    # get variance
    limit = np.sqrt(6) / np.sqrt(in_size + out_size)

    # initialize W with numbers with mean 0 and variance of variance from uniform distribution
    W = np.random.uniform(-limit, limit, (in_size, out_size))

    # initialize b with zeros
    b = np.zeros((1, out_size))

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

2.1.3 Theory

The weights are commonly initialized to non-zero, small, random numbers in order to make each layer as unsymmetrical as possible. When the weights are randomly chosen, their effect on the loss function and hence, their updates are all different, making each neuron unique and enabling the network to learn as many diverse features in the input. Xavier initialization also chooses the variance of the weights such that the variance of the input and variance of the output are the same, which ensures that all weights carry importance and contribution to the prediction and hence, the loss function.

The initialization is scaled by the size of the layer in order to avoid the vanishing gradient and exploding gradient problems. By scaling, the variance of the weights is maintained, ensuring that all the weights contribute sufficiently to the final output.

2.2 Forward Propagation

2.2.1 Code

Sigmoid function:

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

    # compute the sigmoid activation of x
    res = 1 / (1 + np.exp(-x))

    return res
```

Forward pass:

```
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]

    # compute the pre-activation values
    pre_act = np.dot(X, W) + b

    # compute the post-activation values
    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 function:

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

    exp_x = np.exp(x - np.max(x))  # Subtract max for numerical stability
    res =  exp_x / exp_x.sum(axis=0) # apply softmax

    return res
```

2.2.3 Code

Loss and Accuracy calculation:

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

    # compute cross entropy loss
    loss = -np.sum(y * np.log(probs))

    # compute accuracy
    acc = np.mean(np.argmax(y, axis=1) == np.argmax(probs, axis=1))

    return loss, acc
```

2.3 Backwards Propagation

2.3.1 Code

Back-propagation in a layer

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

    Keyword arguments:
    delta -- errors to backprop
    params -- a dictionary containing parameters
    name -- name of the layer
    activation_deriv -- the derivative of the activation_func
    """
    grad_X, grad_W, grad_b = None, None, None

    # print("Backwards for layer", name)

    # 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

    # convert to numpy arrays
    delta = np.array(delta)
    W = np.array(W)
    X = np.array(X)
    pre_act = np.array(pre_act)
    post_act = np.array(post_act)

    loss = delta * activation_deriv(post_act) # chain rule, back propagate the error through the activation function
    grad_W = X.T @ loss # gradient of the loss with respect to weights W
    grad_b = np.sum(loss, axis=0) # gradient of the loss with respect to bias b
    grad_X = loss @ W.T # gradient of the loss with respect to input X

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

2.4 Training Loop

Get Batches

```python
def get_random_batches(x, y, batch_size):
    batches = []

    # get the number of examples
    num_examples = x.shape[0]

    (variable) indices: Any
    indices = np.arange(num_examples)
    shuffled_indices = np.random.permutation(indices)

    # split the data into batches
    for i in range(0, num_examples, batch_size):
        batch_indices = shuffled_indices[i:i + batch_size]
        batch_x = x[batch_indices] # get the batch examples
        batch_y = y[batch_indices] # get the batch labels
        batches.append((batch_x, batch_y)) # append the batch to the list of batches

    return batches
```

Training Loop

```python
batch_size = 5
batches = get_random_batches(x, y, batch_size)
# print batch sizes
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)

# WRITE A TRAINING LOOP HERE
max_iters = 501
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:
        # forward
        h1 = forward(xb, params, "layer1", sigmoid) # forward pass through the first layer
        probs = forward(h1, params, "output", softmax) # forward pass through the output layer

        # loss
        # be sure to add loss and accuracy to epoch totals
        loss, acc = compute_loss_and_acc(yb, probs) # compute loss and accuracy
        total_loss += loss
        total_acc += acc

        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, "output", linear_deriv) # back propagate the error through the output layer
        grad_xb = backwards(delta2, params, "layer1", sigmoid_deriv) # back propagate the error through the first layer

        # 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 Numerical Gradient Checker

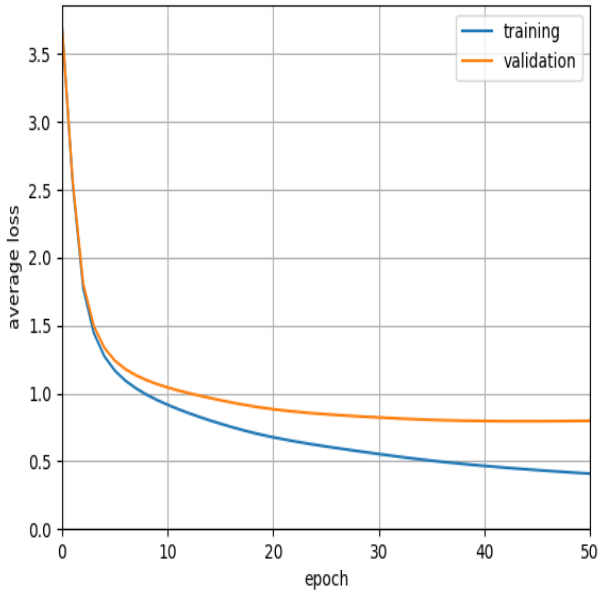
Compute gradients using finite difference of 0.00005

```python
# compute gradients using finite difference
eps = 1e-5
for k, v in params.items():
    if "_" in k:
        continue
    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
```

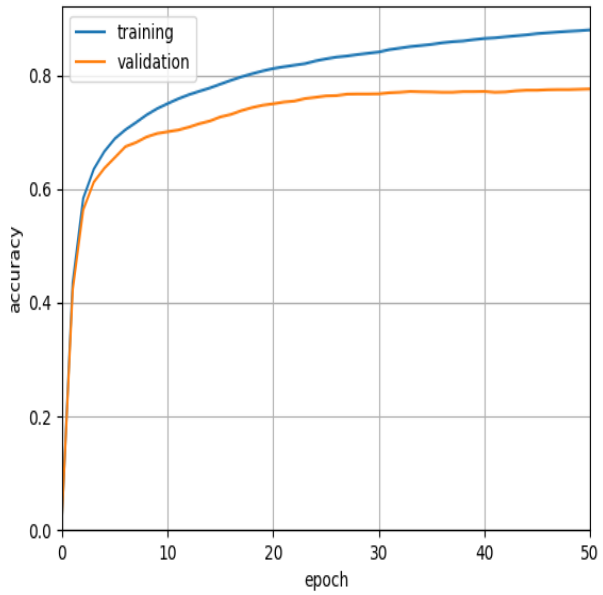
3 Training Models

3.1 Code

The chosen batch size = 32
The chosen learning rate = 5e-3
Average Cross-Entropy Loss:



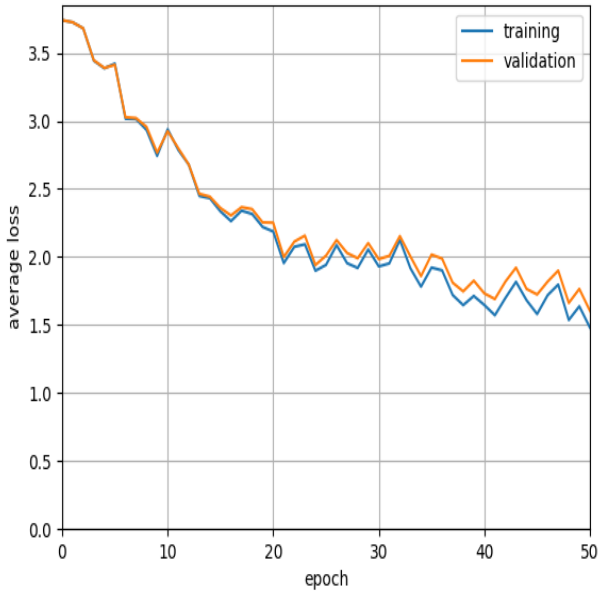
Accuracy:



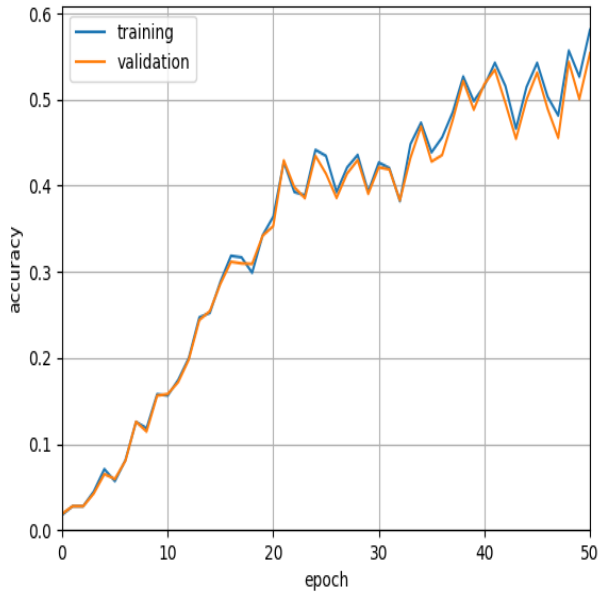
The final validation accuracy : 77.5%

3.2 Writeup

With learning rate: 5e-2:
Average Cross-Entropy Loss:

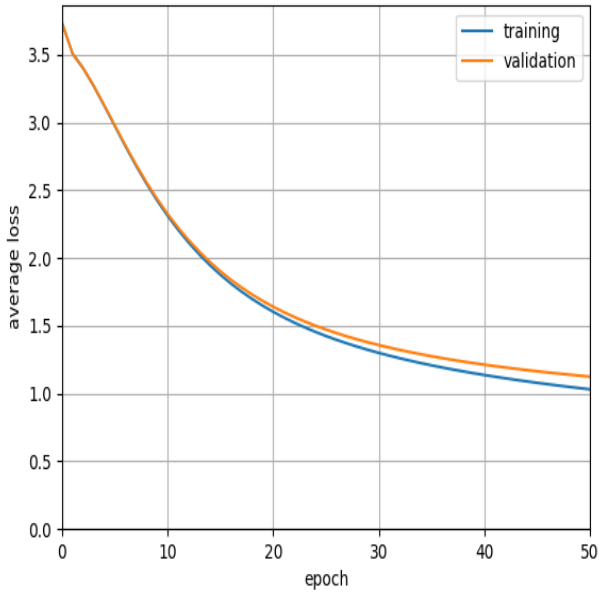


Accuracy:

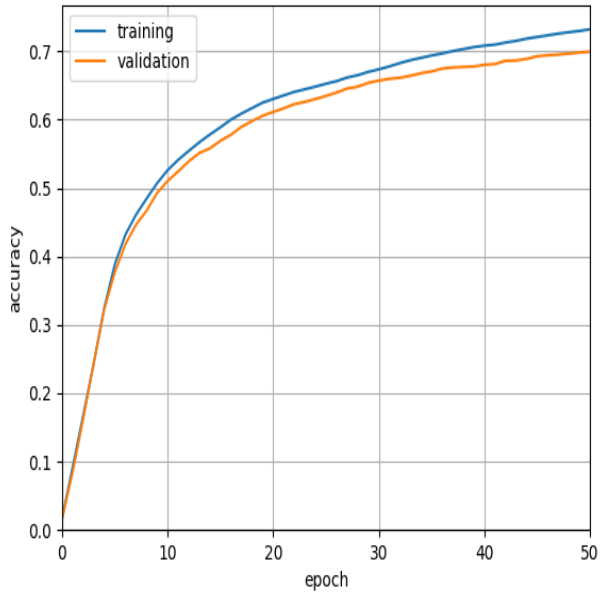


It can be observed that upon increasing the learning rate, the curve changes from a smooth to a jagged curve, indicating the large steps the loss and therefore, the output take at every iteration. The final accuracy is 55.3% which is lower than the previously obtained value.

With learning rate: 5e-4:
Average Cross-Entropy Loss:



Accuracy:

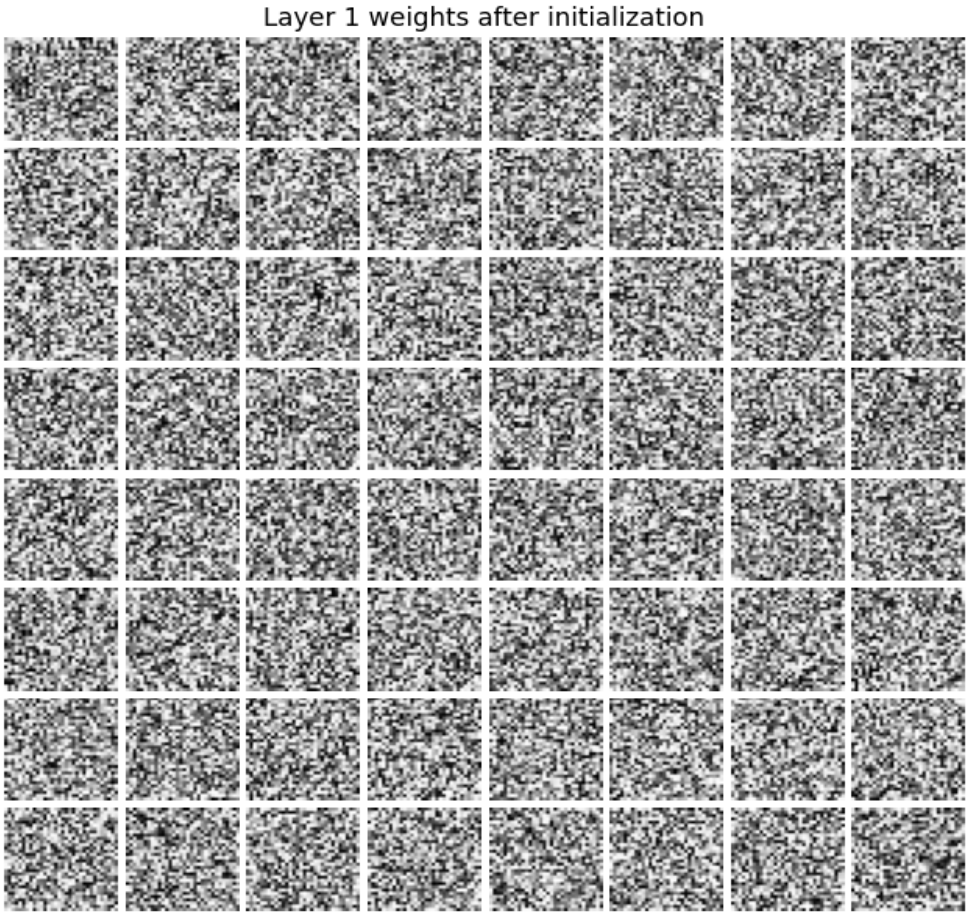


It can be observed that the loss and hence, the output vary smoothly, consequent of the lower learning rate. However, the final accuracy is 69.9%, which is lower than the accuracy obtained using learning rate of 5e-3. This might improve if the network was trained for more epochs.

Hence, the final learning rate chosen is 5e-3 and accuracy obtained is 77.5%.

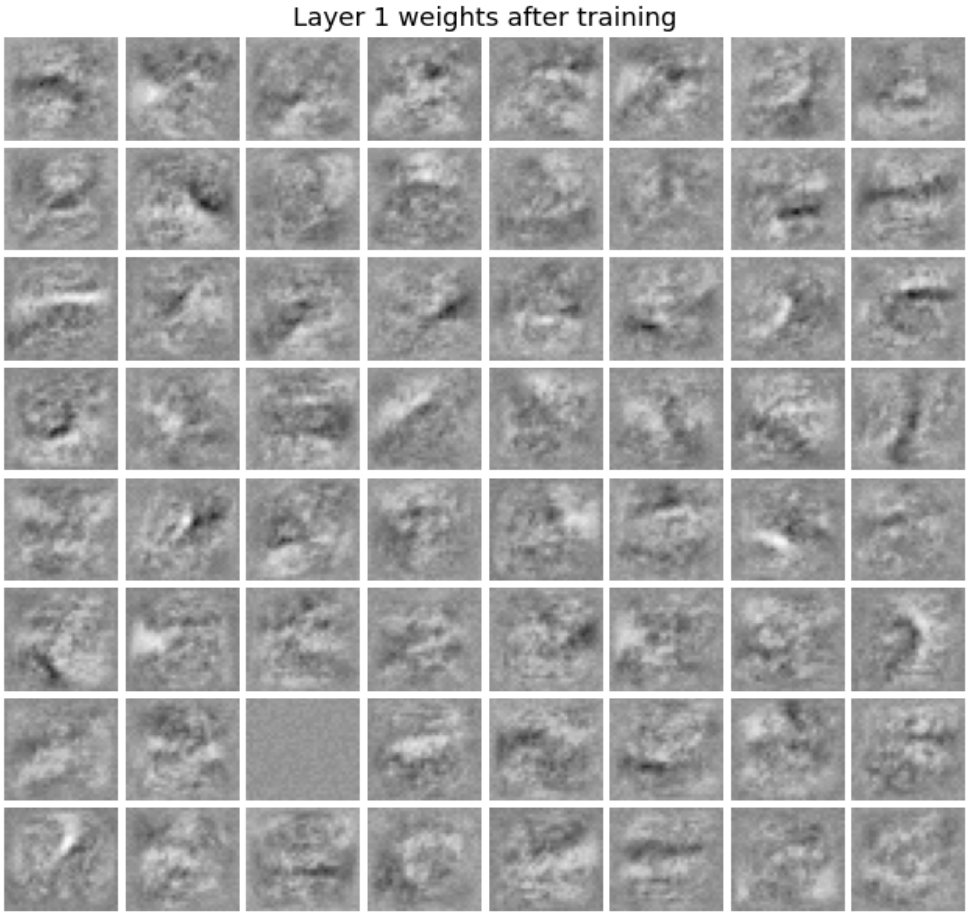
3.3 Writeup

First layer weights after initialization:



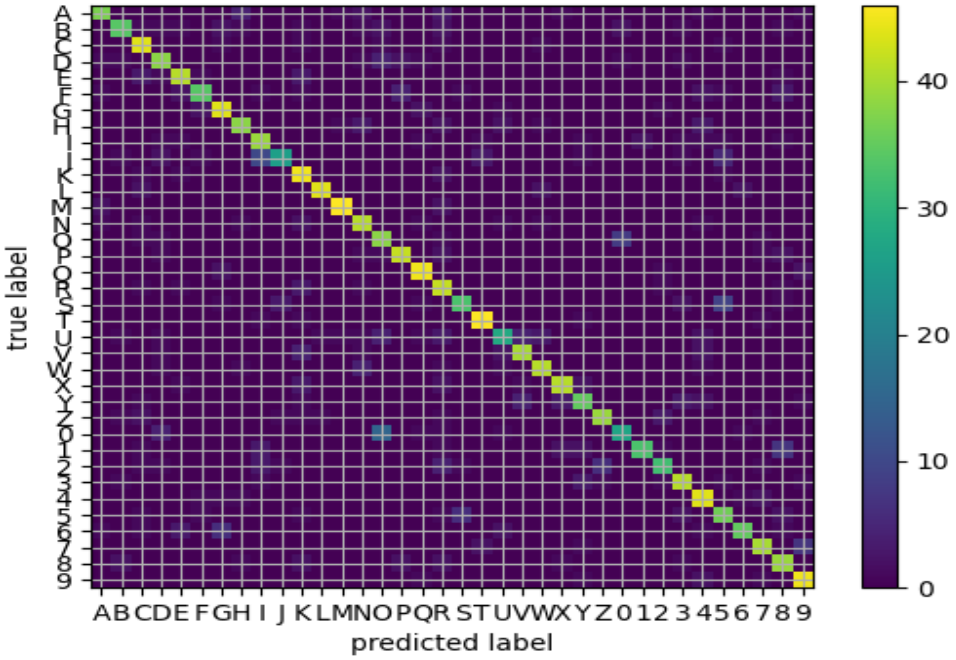
Initially, the weights look like random noise, which corresponds to their initialization as small, random numbers.

First layer weights after training:



After training, the weights resemble some patterns, which correspond to which parts of the image the features are expected to lie in. Hence, the network learns to increase weights for these parts to extract uniqeness from input images.

3.4 Confusion Matrix



The confusion matrix shows that the characters are all identified with high accuracy. It also shows that the network can confuse the following pairs:

- 0 and o
- I and J
- 5 and S

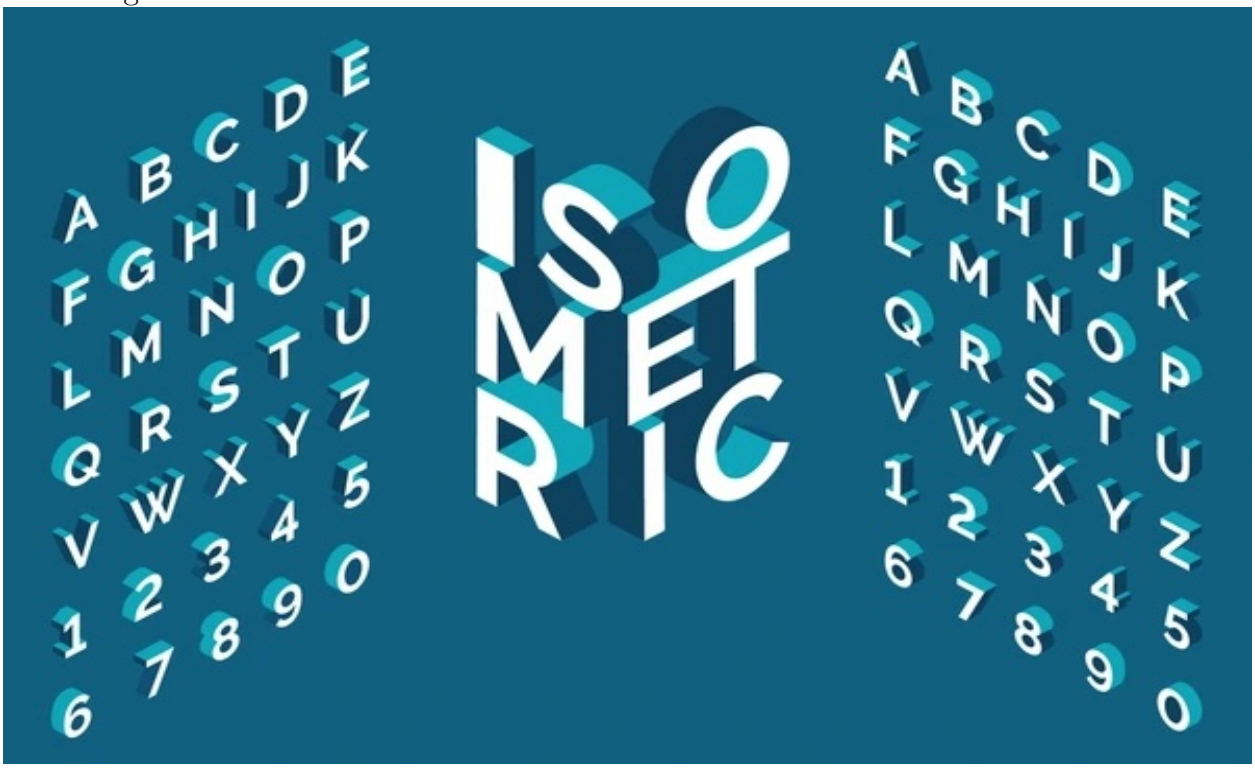
4 Extract Text From Images

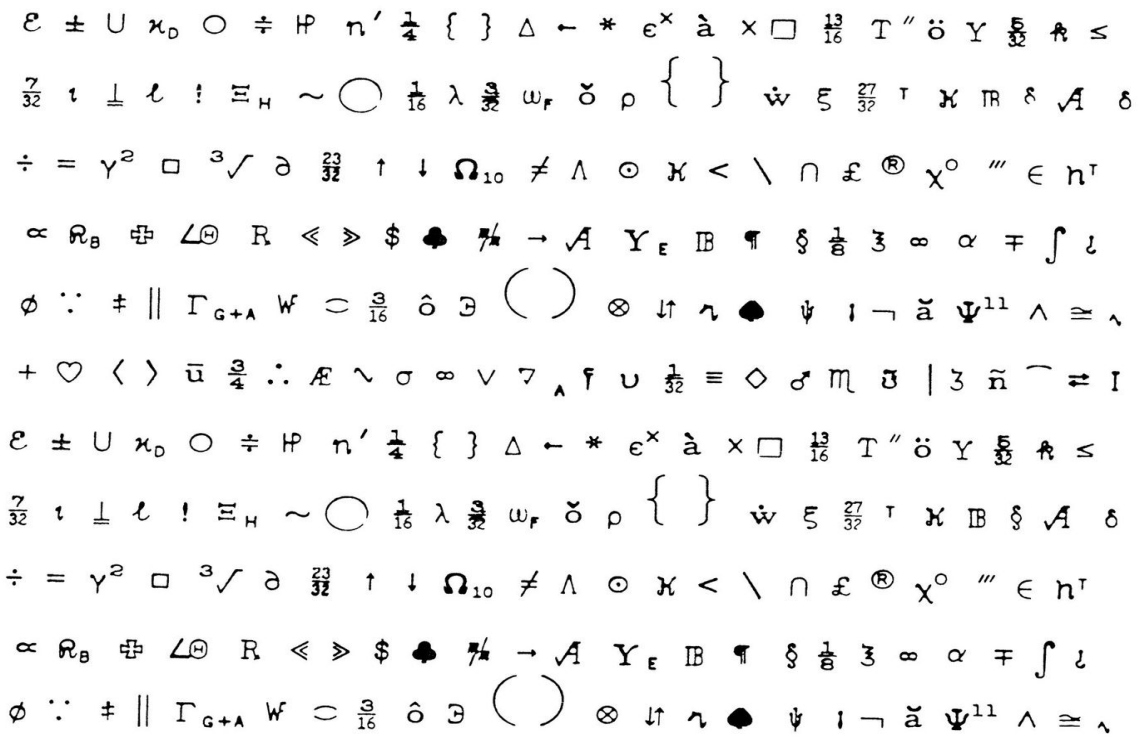
4.1 Theory

The two main assumptions made my this method are:

- The text is written in horizontal lines with spaces between each other. This method will not be able to recognize text when it is written in different orientations in the image.
- The image only contains characters on which the classifier is trained on. In this case, the classifier is trained only on text and numbers, but the method will identify all characters, which are then passed into the classfier. If there are non-text or non-number characters in the image, which are passed into the classifier, it can cause false detections.

Two images for which this method won't work:

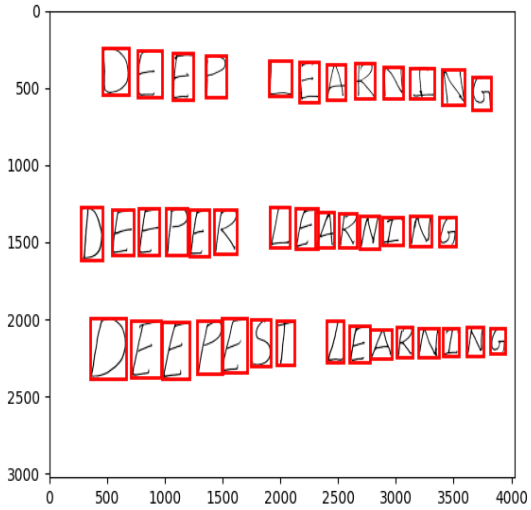


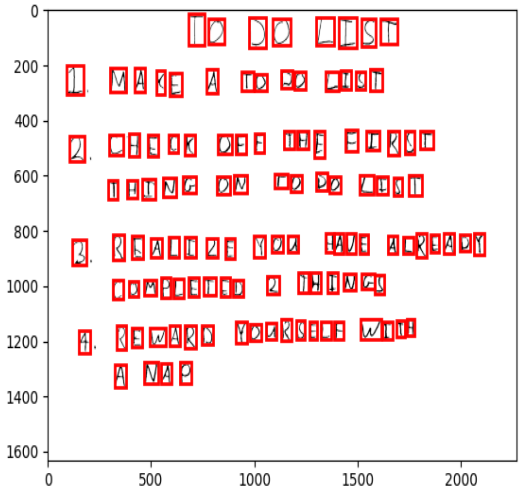


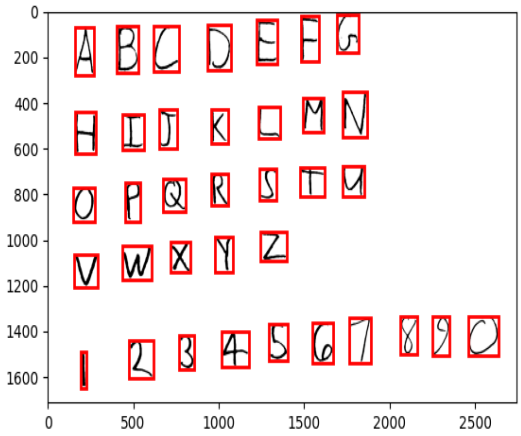
4.2 Code

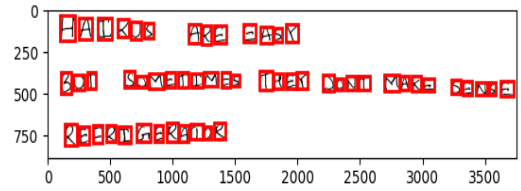
```
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

    # denoise
    image_denoised = skimage.restoration.denoise_bilateral(image, channel_axis=-1)

    # greyscale
    image_gray = skimage.color.rgb2gray(image_denoised)

    # show image
    # skimage.io.imshow(image_gray)
    # skimage.io.show()

    # threshold
    b_w_threshold = skimage.filters.threshold_otsu(image_gray)

    # morphology
    bw = image_gray < b_w_threshold
    bw = skimage.morphology.binary_closing(bw, skimage.morphology.square(3))
    dilated_bw = skimage.morphology.dilation(bw, np.ones((9, 9))) # dilate to connect strokes into letters
    # show image
    # skimage.io.imshow(dilated_bw)
    # skimage.io.show()

    # label
    label_image = skimage.measure.label(skimage.segmentation.clear_border(dilated_bw))
    regions = skimage.measure.regionprops(label_image)

    # small region threshold
    small_region_threshold = 300

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

    return bboxes, bw
```

4.3 Writeup


DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING




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



4.4 Code/Writeup

Cluster:

```
# Sort boxes by top edge (y1)
sorted_boxes = sorted(bboxes, key=lambda box: box[0])

lines = []
current_line = [sorted_boxes[0]]
line_height = sorted_boxes[0][2] - sorted_boxes[0][0]  # Height of the first box

for box in sorted_boxes[1:]:
    # Check if the box belongs to the current line
    if (
        abs(box[0] - current_line[-1][0]) < line_height * 0.5
    ):  # Adjust threshold as needed
        current_line.append(box)
    else:
        # Start a new line
        lines.append(current_line)
        current_line = [box]

    # Update line height (average of boxes in the line)
    line_height = np.mean([b[2] - b[0] for b in current_line])

# Add the last line
if current_line:
    lines.append(current_line)

# Sort boxes within each line by left edge (x1)
sorted_lines = [sorted(line, key=lambda box: box[1]) for line in lines]
```

Crop and Resize:

```
crops = []
for line in sorted_lines:
    for box in line:
        y1, x1, y2, x2 = box  # current letter bounding box
        crop = bw[y1:y2, x1:x2]  # crop the letter (only the part inside the bounding box)

        # calculate padding to make it square
        height, width = crop.shape
        size_diff = abs(height - width)
        pad_top = pad_bottom = pad_left = pad_right = 0

        if height > width:  # taller than wide
            pad_left = size_diff // 2
            pad_right = size_diff - pad_left
        elif width > height:  # wider than tall
            pad_top = size_diff // 2
            pad_bottom = size_diff - pad_top

        # invert the crop
        crop = 1 - crop

        # # show the crop
        # plt.imshow(crop, cmap="Greys")
        # plt.show()

        # pad the crop to make it square
        crop = np.pad(crop, ((pad_top, pad_bottom), (pad_left, pad_right)), mode="constant", constant_values=1)

        # dialate the crop to make lines thicker
        d = np.ones((int(11*crop.shape[0]/crop.shape[1]), int(10*crop.shape[1]/crop.shape[0]))) # for letter dilation
        crop = skimage.morphology.erosion(crop, d)
        # crop = skimage.morphology.erosion(crop, np.ones((3, 3)))
        # crop = skimage.morphology.dilation(crop, np.ones((3, 3)))

        # resize the crop to 32x32
        crop = skimage.transform.resize(
            crop, (32, 32), anti_aliasing=True, preserve_range=True
        )

        # show the resized crop
        # plt.imshow(crop, cmap="Greys")
        # plt.show()

        # traspose the crop
        crop = crop.T

        # flatten the crop
        crops.append(crop.flatten())

inputs = np.array(crops)
```

Classify:

```
# forward pass
h1 = forward(inputs, params, "layer1")
probs = forward(h1, params, "output", softmax)

# get the most likely letter for each crop
most_likely = np.argmax(probs, axis=1)
most_likely_letters = "".join(letters[most_likely])

# separate the letters by line
start = 0
for line in sorted_lines:
    end = start + len(line)
    print(most_likely_letters[start:end])
    start = end
print("\n")
```

Extracted Text:

```
Found 41 letters
C5FYLBAKMING
DFFY1KLEAK4ING
JF1P15ILFARNING


Found 115 letters
IQDQLI8T
IMBE6ATQQQLIST
ZCH6CKQFFTHEFIR8T
THINGQNTQDQLIIT
ZR8BLIZEYQUHA66BLR6AQ8
CQMPL8TLBZTHINGI
988WBRDYQWRSBLFWIIB
ANAP


Found 36 letters
2BLDFFG
HIIKLMN
QPQR5TW
VWX7Z
1Z3GS67X9S


Found 54 letters
HAIKUQAR6HABT
BWTSQMETIMBGTHETDQWTMAKLBHNGE
RBFRIGBRRTQR
```

The outputs written as text with spaces that "make sense":

C5FY LBAKMING
DFFY1K LEAK4ING
JF1P15I LFARNING

IQ DQ LI8T
I MBE6 A TQ QQ LIST
Z CH6CK QFF THE FIR8T
THING QN TQ DQ LIIT
Z R8BLIZE YQU HA66 BLR6AQ8
CQMPL8TLB Z THINGI
9 88WBRD YQWRSBLF WIIB
ANAP

2BLDFFG
HIIKLMN
QPQR5TW
VWX7Z

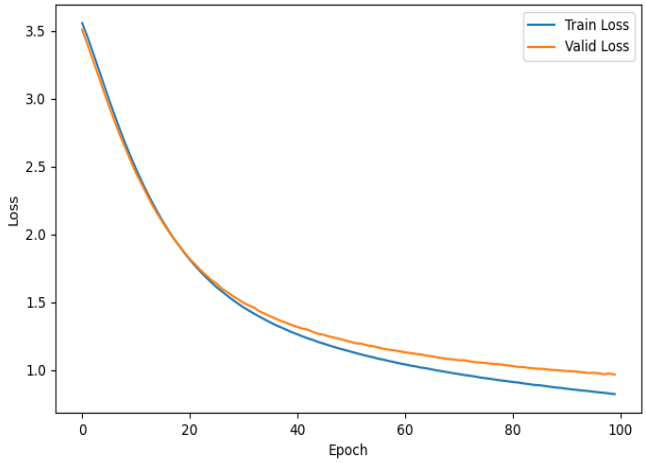
HAIKUQ AR6 HABT
BWT SQMETIMBG THET DQWT MAKL BHNGE
RBFRIGBRRTQR

6 PyTorch

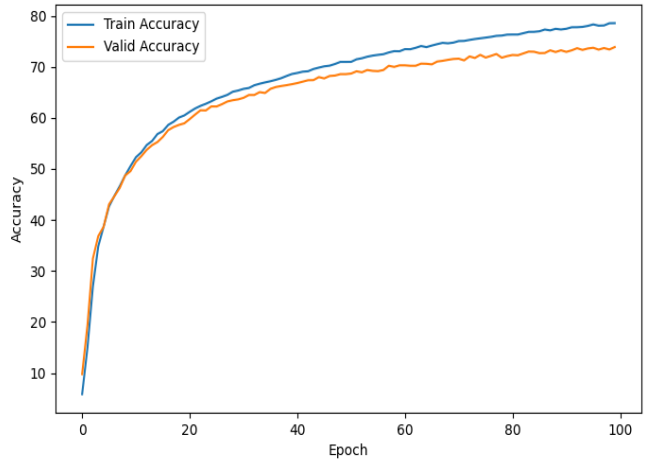
6.1 Train a Neural Network in PyTorch

6.1.1 Fully-Connected Network on NIST36

Loss:



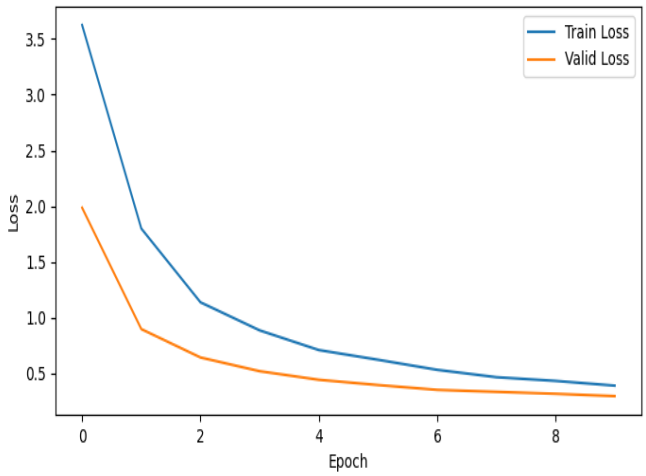
Final Validation Loss: 0.9654
Accuracy:



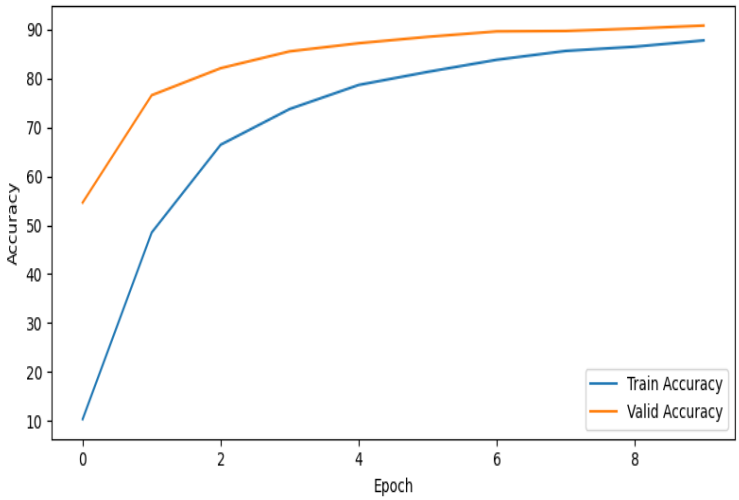
Final Validation Accuracy: 73.86%

6.1.2 Convolutional Neural Network on NIST36

Loss:



Final Validation Loss: 0.2933
Accuracy:

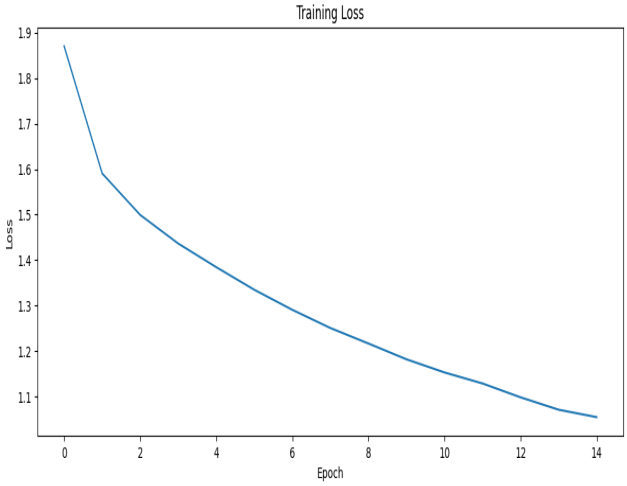


Final Validation Accuracy: 90.83%

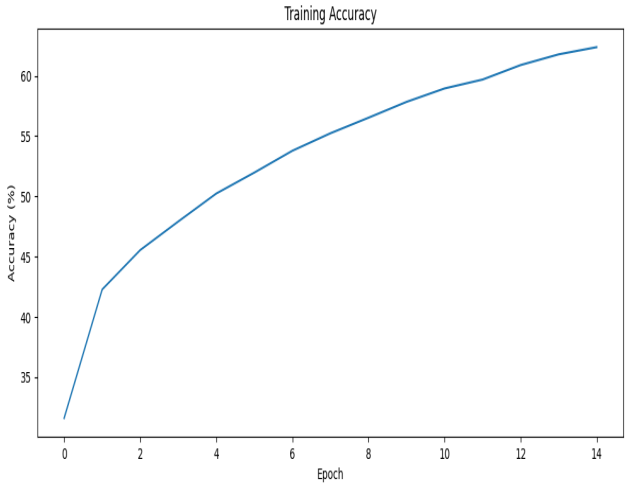
In fewer epochs, the convolutional neural network outperforms the fully connected network. The validation loss and accuracy are much better in 1/10th the number of epochs. This is because convolutional neural networks are much better suited to image data than fully connected netoworks because of CNNs can automatically and adaptively learn spatial hierarchies of features. That is, lower layers learn simple features (like edges), while deeper layers learn more complex features (like shapes or objects). This hierarchical feature learning is particularly well-suited for image data.

6.1.3 Convolutional Neural Network on CIFAR-10

Loss:



Final Validation Loss: 1.0552
Accuracy:



Final Validation Accuracy: 62.38%