Advanced Computer Vision HW4

Ricky Yuan
rickyy@andrew.cmu.edu

November 7, 2024

1 Theory

1.1

Adding a constant c to the vector $\mathbf{x}$ gives us:

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

for all x_i in $\mathbf{x}$ holds, hence proved.

Since x_i could be large, making e^x exponentially large, resulting in a huge number divided by a huge number, it can lead to numerical instability. By setting $c = -\max x_i$, the largest term becomes 0, and the range of the exponential function becomes manageable.

1.2

The output of the softmax function lies within the range $(0, 1)$, and the sum over all the elements is 1. Therefore, we can view the output of the softmax function as a **probability distribution**. The exponential operation makes the input value positive and amplifies larger values than smaller values. The summation normalizes the values of the next step, and ensures that the final softmax output sums to 1. The last operation then calculates the corresponding probabilities.

1.3

Consider a multi-layer neural network with input vector $\mathbf{x}$ and parameters W_i, b_i, where $i \in (1, 2, ..., N)$, and N is the number of layer. The output y is

$$y = W_N W_{N-1} ... W_1 \mathbf{x} + b_N + W_N b_{N-1} + W_N W_{N-1} b_{N-1} + ...$$

We can combine and express the neural network as $W\mathbf{x} + b$, where

$$\begin{aligned} W &= W_N W_{N-1} ... W_1 \\ b &= b_N + W_N b_{N-1} + W_N W_{N-1} b_{N-1} + ... \end{aligned}$$

Therefore it is equivalent to linear regression.

1.4

$$\begin{aligned} \frac{\partial \sigma(x)}{\partial x} &= \frac{\partial}{\partial x} \frac{1}{1+e^{-x}} = -(1+e^{-x})^{-2} \cdot (-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} \cdot (1 - \frac{1}{1+e^{-x}}) = \sigma(x) \cdot (1 - \sigma(x)) \end{aligned}$$

1.5

$$y \in \mathbb{R}^{k \times 1} \quad W \in \mathbb{R}^{k \times d} \quad x \in \mathbb{R}^{d \times 1} \quad b \in \mathbb{R}^{k \times 1}$$

The loss J is a function of y, so we can write:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial W}$$

Since $y_i = \sum_{j=1}^{d} W_{ij} x_j + b_i$, then the partial derivative of y_i w.r.t W_{ij} is:

$$\frac{\partial y_i}{\partial W_{ij}} = x_j$$

We then have the matrix form $\frac{\partial y}{\partial W} = x^T$. Thus the answer is:

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

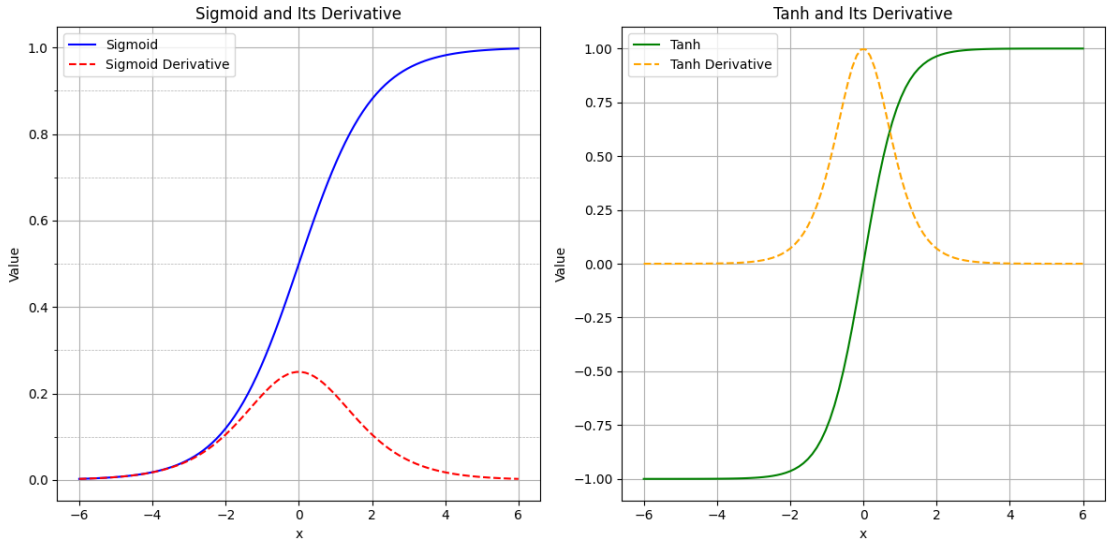
where $\delta \in \mathbb{R}^{k \times 1}$ and $x^T \in \mathbb{R}^{1 \times d}$. The result is a $k \times d$ matrix.

We use a similar technique to derive other partial derivatives.

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial x} = W^T \delta \qquad \frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial b} = \delta$$

where $W^T \in \mathbb{R}^{d \times k}$ and $\delta \in \mathbb{R}^{k \times 1}$. The result of $\frac{\partial J}{\partial x}$ is a $d \times 1$ vector.

1.6



1. The value of the derivative of sigmoid ranges from 0 to 0.25. When we do backpropagation in a very deep network, multiple values that range from 0 to 0.25 propagate back to the input, causing the gradient to vanish.

2. The output ranges for sigmoid and tanh are $(0, 1), (-1, 1)$, respectively. The mean of tanh is 0 so that the model would not learn bias compared to softmax.

3. The derivative of tanh ranges from $(0, 1)$, which has a larger domain compared to the derivative of softmax, making the model less likely to cause gradient vanishing.

4. The definition of tanh can be written as:

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

2 Implement a Fully Connected Network

2.1 Network Initialization

2.1.1

Initializing a network with all zeros would make the output of each neuron to be 0, causing the gradient to be all zeros. The network would then learn nothing since the gradient is zero.

2.1.2

```python
def initialize_weights(in_size, out_size, params, name=""):
    init_range = np.sqrt(6 / (in_size + out_size))
    W = np.random.uniform(-init_range, init_range, (in_size, out_size))
    b = np.zeros(out_size)

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

2.1.3

Initialization with random numbers helps different neurons better capture good and different features. If we initializing all the weights to be the same, effectively all the neurons would make no difference at all. Scaling the initialization properly makes the variance of the output reasonable, and it prevents the gradient vanishing or gradient exploding.

2.2 Forward Propagation

2.2.1

```python
def sigmoid(x):
    res = 1 / (1 + np.exp(-x))
    return res

def forward(X, params, name="", activation=sigmoid):
    # 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

```python
def softmax(x):
    _x = x - np.max(x, axis=1, keepdims=True)
    res = np.exp(_x) / np.sum(np.exp(_x), axis=1, keepdims=True)
    return np.array(res)
```

2.2.3

```python
def compute_loss_and_acc(y, probs):
    loss = -np.sum(y * np.log(probs)) # prob = f(x), y is one-hot
    acc = np.mean(np.argmax(y, axis=1) == np.argmax(probs, axis=1))
    return loss, acc
```

2.3 Backwards Propagation

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

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

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

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

2.4 Training Loop

```python
def get_random_batches(x, y, batch_size):
    batches = []
    num_samples = x.shape[0]
    indices = np.random.permutation(num_samples)

    for i in range(0, num_samples, batch_size):
        batch_indices = indices[i : i + batch_size]
        batches.append((x[batch_indices], y[batch_indices]))

    return batches
```

```python
batches = get_random_batches(x, y, 5)
batch_num = len(batches)

# WRITE A TRAINING LOOP HERE
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb, yb in batches:
        # forward
        h1 = forward(xb, params, "layer1")
        probs = forward(h1, params, "output", softmax)

        # loss
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc

        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, "output", linear_deriv)
        backwards(delta2, params, "layer1", sigmoid_deriv) # No need to
            store the final grad_X

        # apply gradient
        # gradients should be summed over batch samples
        for k in params.keys():
            if "grad" in k:
                name = k.split("_")[1] # layer name
                params[name] -= learning_rate * params[k]

    avg_acc /= batch_num
    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
eps = 1e-6
for k, v in params.items():
    if "_" in k:
        continue
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   subtract 2*epsilon
    #   run the network
    #   get the loss
    #   restore the original parameter value
    #   compute derivative with central diffs

    # Create an iterator to go through each element of the parameter
    it = np.nditer(v, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index # Current index, 2D for W, 1D for b

        # Compute f(x + eps)
        v[idx] += eps
        h1 = forward(x, params, "layer1")
        probs = forward(h1, params, "output", softmax)
        loss1, _ = compute_loss_and_acc(y, probs)

        # Compute f(x - eps)
        v[idx] -= 2 * eps
        h1 = forward(x, params, "layer1")
        probs = forward(h1, params, "output", softmax)
        loss2, _ = compute_loss_and_acc(y, probs)

        v[idx] += eps  # Restore original value

        # Compute the numerical gradient
        grad = (loss1 - loss2) / (2 * eps)
        params["grad_" + k][idx] = grad

        it.iternext()  # Move to the next element
```

3 Training Models

3.1

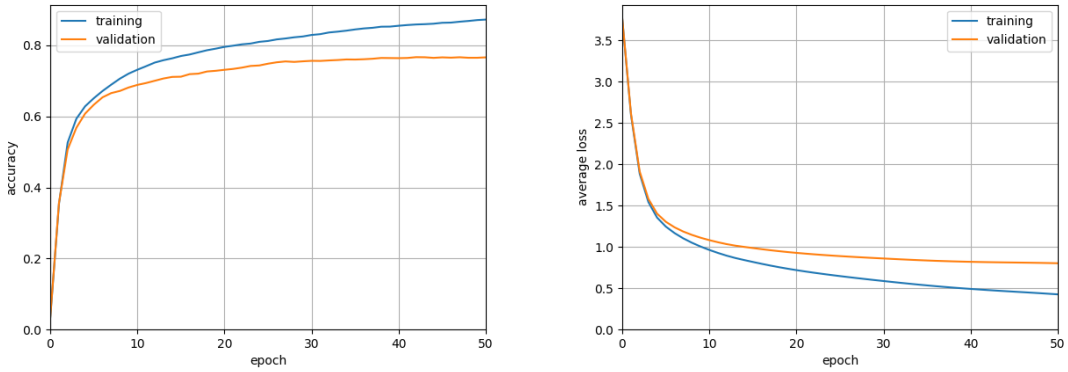


Figure 1: Accuracy & Loss with Batch = 64, Learning Rate = 0.005, Random Seed = 1487

3.2

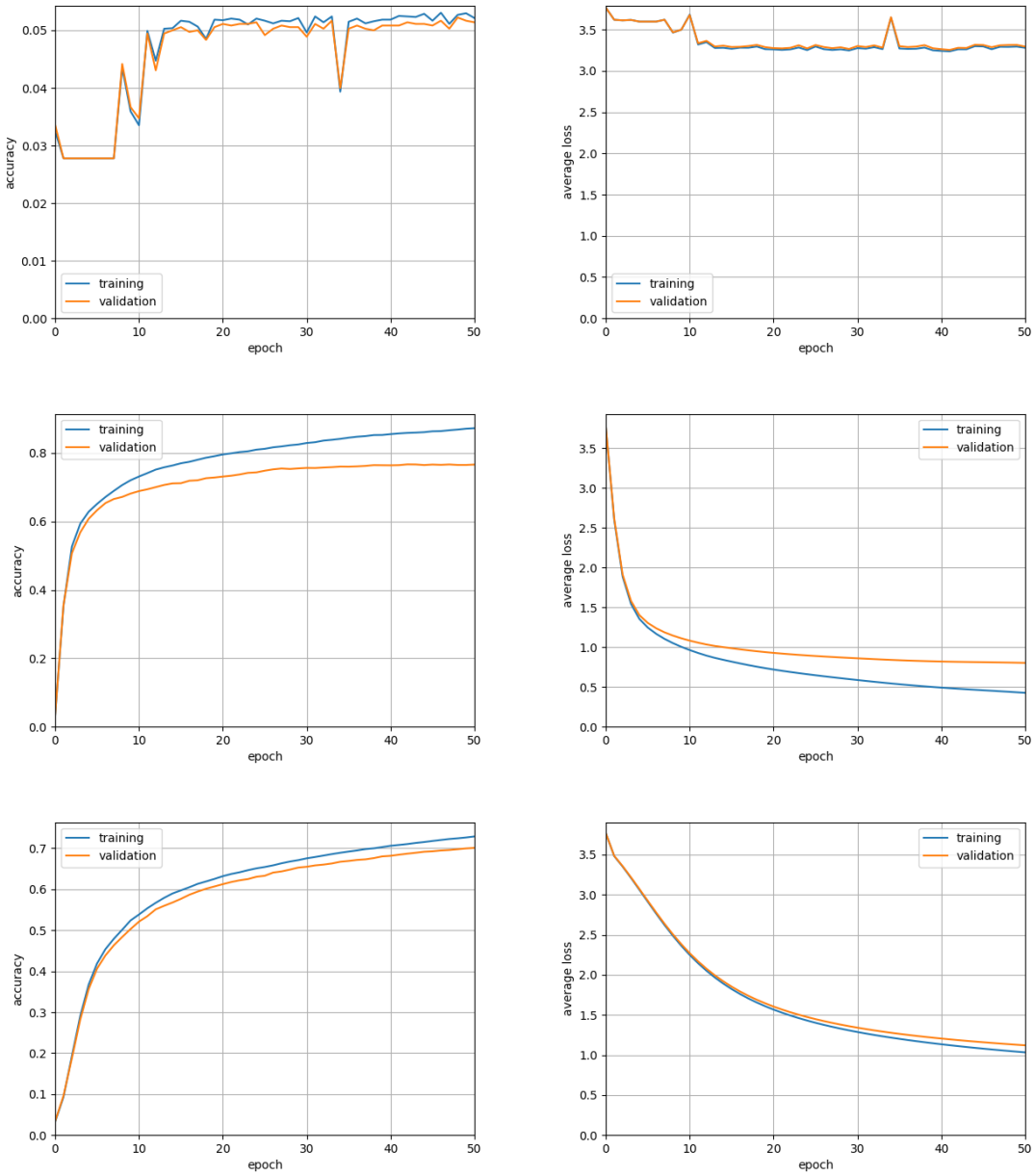


Figure 2: Accuracy & Loss with Learning Rates = $[0.05, 0.005, 0.0005]$, respectively

From the figures above, we see that the result for learning rate = 0.05 is bad because the update steps are too large, causing the loss oscillating and not able to converge to a minimum. On the other hand, the result for learning rate = 0.0005 have the validating accuracy around 0.71, which is still worse than the original experiment settings because the learning rate is too small. It needs more training iterations until fully convergence. The best network's accuracy reaches 0.766 for validation data and 0.773 for test data.

3.3

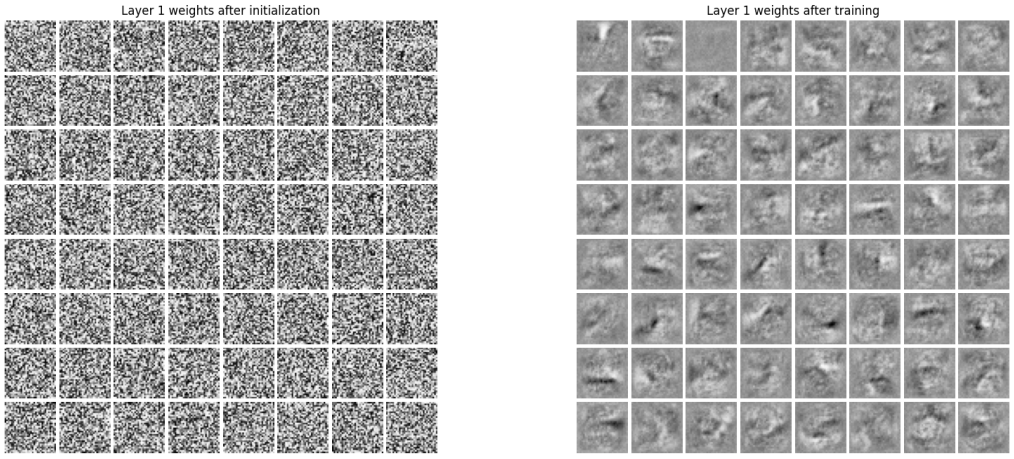


Figure 3: Visualization of the first layer before and after training

The first layer before training looks noisy and random, whereas the one after training looks there are patterns like strokes or lines.

3.4

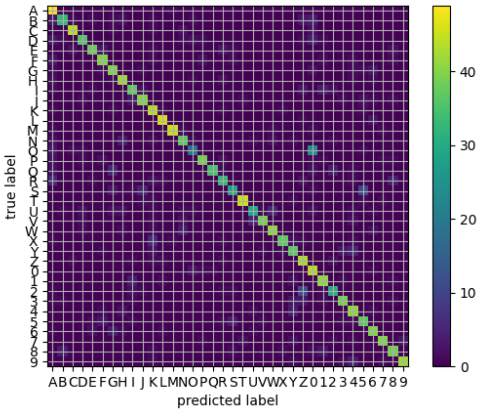


Figure 4: Confusion Matrix of the test data

From the figure, we can see that the character "O" and number "0" are the most easily to be confused by the model. Other pairs like "Z" and "2", "S" and "5" are easy to be confused as well.

4 Extract Text from Images

4.1

The sample method relies on several assumptions. One key assumption is that characters are not connected, allowing us to draw a bounding box around each character and feed them into our neural network one at a time. However, as shown in the example below, when characters are connected, it becomes difficult to separate them during the bounding box stage. This can lead to poor classification results.

Another important assumption is that the testing data is sampled from the same domain as the training data. The model is trained solely on English characters, and as a result, it cannot recognize characters from other languages. For example, the image below shows two Traditional Chinese characters meaning "hello." Since these characters are not part of the model's output classes, the model fails to classify them correctly.

4.2

```python
def findLetters(image):
    # 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 the image using a denoising filter
    # Ref: https://scikit-image.org/docs/stable/auto_examples/filters/
        plot_denoise.html
    # https://medium.com/@betulmesci/image-processing-tutorial-using-
        scikit-image-noise-20e181c541a1
    denoised = skimage.restoration.denoise_bilateral(image, channel_axis
        =-1)
    # denoised = skimage.filters.gaussian(image, sigma=1)

    # 2. Convert to grayscale
```

```
    gray = skimage.color.rgb2gray(denoised)

    # 3. Apply a threshold to get a binary (black-and-white) image
    threshold = skimage.filters.threshold_otsu(gray)
    bw = gray < threshold  # Invert the binary image if needed

    # 4. Apply morphological operations to clean up small noise
    bw = skimage.morphology.remove_small_objects(bw, min_size=100)
    bw = skimage.morphology.closing(bw, skimage.morphology.square(6))

    # 5. Label connected components, 8-connectivity
    labeled = skimage.measure.label(bw, connectivity=2)

    # 6. Find regions and filter small components (bounding boxes)
    regions = skimage.measure.regionprops(labeled)

    # 7. Filter regions based on area
    areas = [region.area for region in regions]
    mean, std = np.mean(areas), np.std(areas)
    # Drop all regions with area less than mean - 2.5 * std
    bboxes = [region.bbox for region in regions if region.area > mean -
        2.5 * std]

    # Make the stroke of the letters thicker
    bw = skimage.morphology.dilation(bw, footprint=skimage.morphology.
        square(5))
    return bboxes, bw.astype(np.float64)
```

4.3

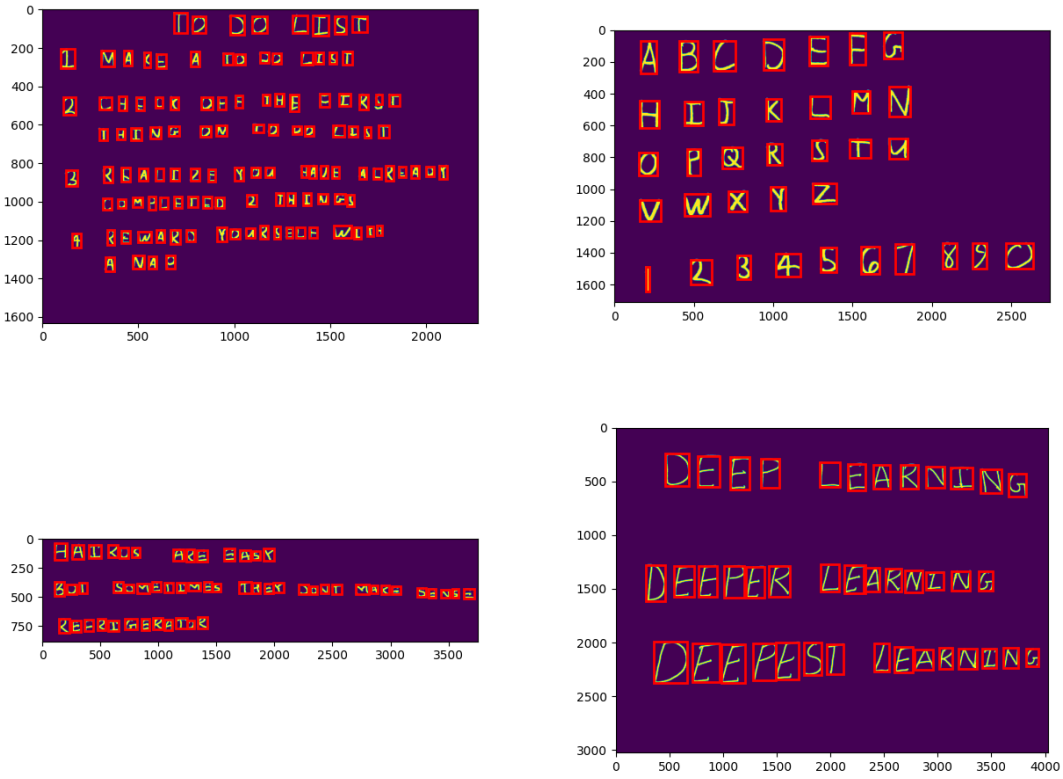


Figure 5: Result of `findLetters()`

4.4

list
TODOLI5T
IMAXEATDDULIST
2LHF5KDFF7HEFIRFT
7HINGQNTQWQLI5T
3RIALIZEYQUHAVEALR8ADT
CDMPLFTIDZTHINGS
AREWARDYOURSELFWITH
ANAP

letters
ABCDEFG
HIIKLMN
QPQR5TU
VWXYZ
1Z345G789Q

haiku
HAIKUSAREHASY
BUTSQMETIMESTHEYDDNTMAK2SGNGE
RBFRIGERATQR

deep
DEEFLEARMIHG
DE8PERLEARAING
D55PE5TLEARNIHG

The total accuracy is 193/246 (78.46%).

5 (Extra Credit) Image Compression with Autoencoders

5.1 Building the Autoencoder

5.1.1

```
# Initialize the layers for the autoencoder
input_size = train_x.shape[1]  # 1024 for the NIST36 dataset
initialize_weights(input_size, hidden_size, params, "layer1")
initialize_weights(hidden_size, hidden_size, params, "layer2")
initialize_weights(hidden_size, hidden_size, params, "layer3")
initialize_weights(hidden_size, input_size, params, "output")
# Initialize the momentum terms
for k in ["layer1", "layer2", "layer3", "output"]:
    params["m_W" + k] = np.zeros_like(params["W" + k])
    params["m_b" + k] = np.zeros_like(params["b" + k])

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # forward pass
        h1 = forward(xb, params, "layer1", relu)
        h2 = forward(h1, params, "layer2", relu)
        h3 = forward(h2, params, "layer3", relu)
        reconstructed_x = forward(h3, params, "output", sigmoid)

        # loss
        loss = np.sum((xb - reconstructed_x) ** 2) # total squared error
        total_loss += loss

        # backward
        delta = 2 * (reconstructed_x - xb)
        delta = backwards(delta, params, "output", sigmoid_deriv)
        delta = backwards(delta, params, "layer3", relu_deriv)
        delta = backwards(delta, params, "layer2", relu_deriv)
        backwards(delta, params, "layer1", relu_deriv)

        # apply gradient, remember to update momentum as well
        for k in params.keys():
            if "grad" in k:
                layer_name = k.split('_')[1]  # Layer name, e.g. Wlayer1
                m_key = "m_" + layer_name  # Momentum key, e.g. m_Wlayer1
                params[m_key] = 0.9 * params[m_key] - learning_rate *
                    params[k]
                params[layer_name] += params[m_key]

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

5.1.2

See the code above.

5.2 Training the Autoencoder

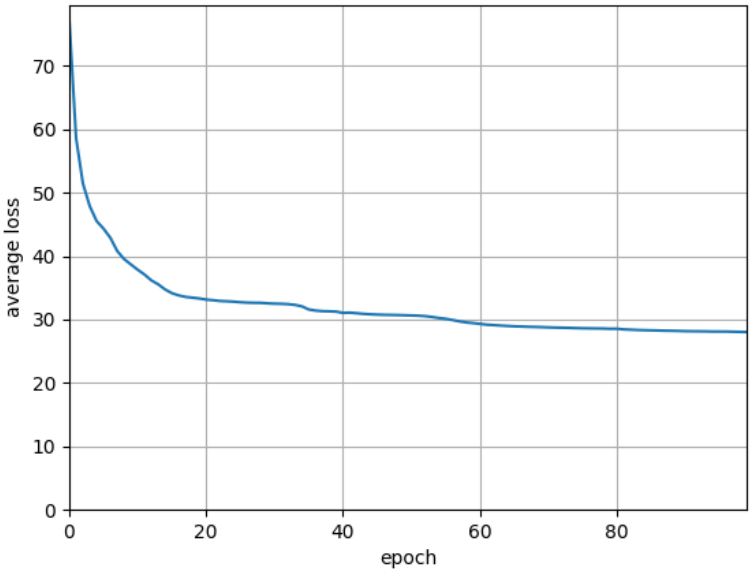


Figure 6: Training loss curve for the autoencoder

From the figure above, we observe that the loss decreases rapidly during early training. After that, the loss converges to value around 30.

5.3 Evaulating the Autoencoder

5.3.1

Figure 7: Reconstruction result for character R, I, C, K, Y

From the figure above, we observe that the reconstructed characters retains general shape but are more blurry and noisy. Some characters, such as "R" or "K" suffer from noticeable distortion and lack of details. This suggests that the autoencoder captures basic features but struggles with finer details due to the limited latent space.

5.3.2

With default hyper-parameters, the average PSNR in my experiment is 16.00.

6 PyTorch

6.1 Train a neural network in PyTorch

6.1.1

Hyper-parameters:

```
batch_size = 64
learning_rate = 0.005
num_epochs = 100
hidden_size = 64
momentum = 0.9
```

Network Structure:

```
class FullyConnectedNetwork(nn.Module):
    def __init__(self):
        super(FullyConnectedNetwork, self).__init__()
        self.hidden_size = hidden_size
        self.model = nn.Sequential(
            nn.Linear(1024, self.hidden_size),
            nn.Sigmoid(),
            nn.Linear(self.hidden_size, 36),
        )

    def forward(self, x):
        return self.model(x)
```

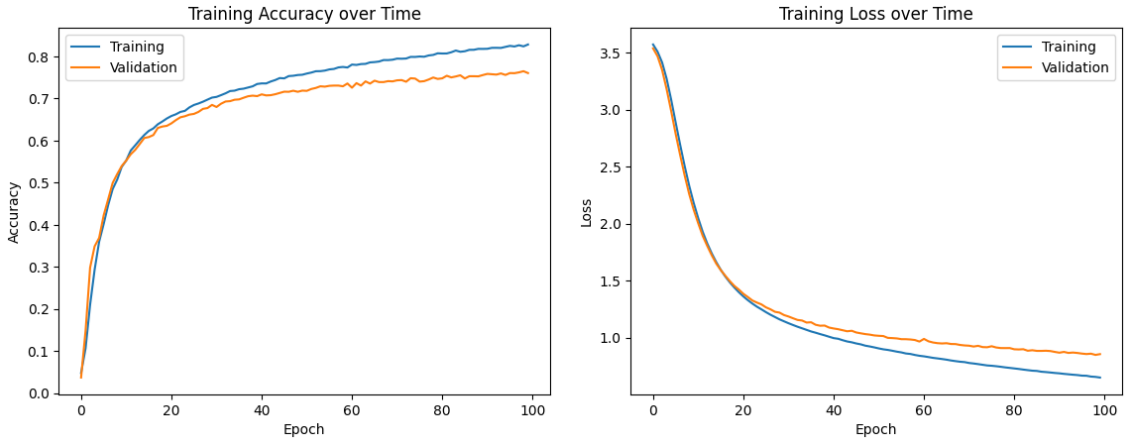


Figure 8: Accuracy and Loss of Training and Validation for fully-connected network

The network's accuracy reaches 0.7642 for validation data and 0.7656 for test data.

6.1.2

The hyper-parameters are the same as the previous setting, except here I used `num_epoch=30` since it converges much faster.

Network Structure:

```
class ConvNetwork(nn.Module):
    def __init__(self):
        super(ConvNetwork, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.Conv2d(4, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
            nn.Flatten(),
            nn.Linear(16 * 8 * 8, 256),
            nn.ReLU(),
            nn.Linear(256, 36)
        )

    def forward(self, x):
        x = x.reshape(-1, 1, 32, 32)
        return self.model(x)
```

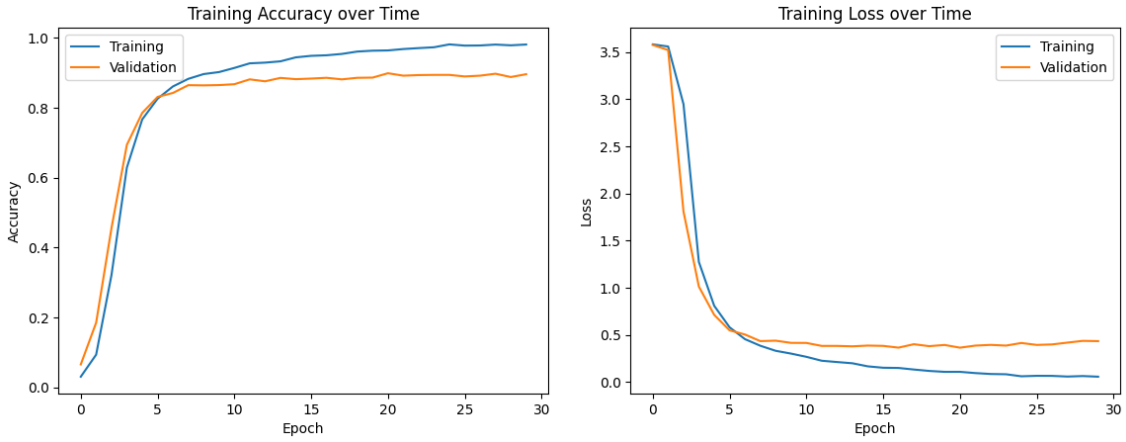


Figure 9: Accuracy and Loss of Training and Validation for convolution neural network

The network's accuracy reaches 0.8864 for validation data and 0.8917 for test data. Compared to the fully connected network, CNN converges much faster and has better performance.

6.1.3

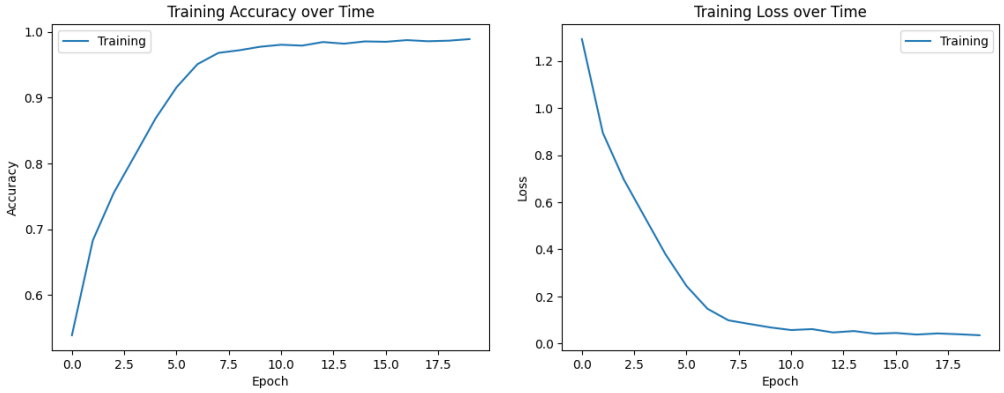


Figure 10: Accuracy and Loss of Training for CIFAR10 dataset

The network reaches around 0.72 accuracy over the test data.

6.2 Fine Tuning

I chose to finetune on the flowers 17 dataset, and trained only 10 epochs. Below are my results.

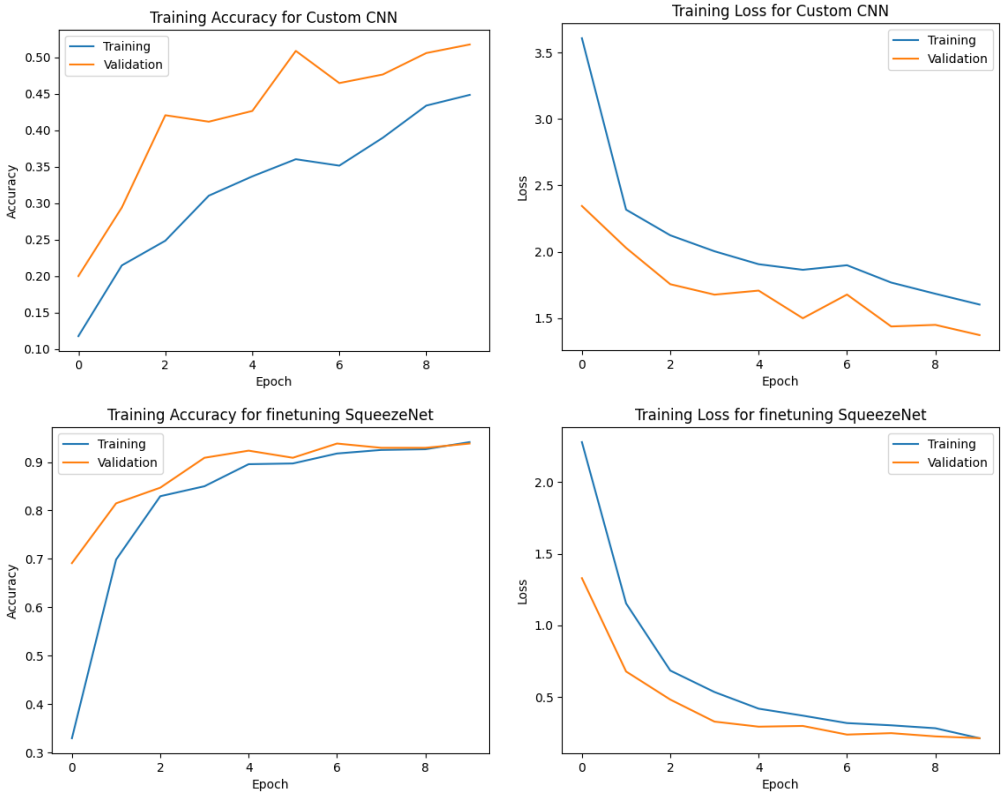


Figure 11: Accuracy and Loss of finetuning and training the whole model

As we can see, finetuning converges much faster and has a better performance. The performance of finetuning SqueezeNet for 10 epoch reaches 0.94 accuracy on validation set whereas training the whole model only reaches 0.52 accuracy. Below are my code and model structures:

```
model_squeezenet = models.squeezenet1_1(weights=models.
    SqueezeNet1_1_Weights.DEFAULT)

num_classes = len(class_names)
# Replace the classifier part of the model
model_squeezenet.classifier[1] = nn.Conv2d(512, num_classes, kernel_size
    =(1, 1))
model_squeezenet.num_classes = num_classes

# Freeze all layers in the feature extractor
for param in model_squeezenet.features.parameters():
    param.requires_grad = False
```

And the custom CNN model structure:

```
class CustomCNN(nn.Module):
    def __init__(self, num_classes):
        super(CustomCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(256 * 28 * 28, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten the features
        x = self.classifier(x)
        return x
```

Training logics:

```
def train_model(model, criterion, optimizer, num_epochs=10):
    train_losses, valid_losses = [], []
    train_accs, valid_accs = [], []
    for epoch in range(num_epochs):
        model.train()
        train_acc, train_loss = 0, 0
        for inputs, labels in dataloaders['train']:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            train_acc += (predicted == labels).sum().item()

        train_acc /= dataset_sizes['train']
        train_loss /= len(dataloaders['train'])
        train_losses.append(train_loss)
        train_accs.append(train_acc)

        # Validation loop
        model.eval()
        valid_acc, valid_loss = 0, 0
        with torch.no_grad():
            for inputs, labels in dataloaders['val']:
                outputs = model(inputs)
                _, predicted = torch.max(outputs, 1)
                valid_acc += (predicted == labels).sum().item()
                valid_loss += criterion(outputs, labels).item()

        valid_acc /= dataset_sizes['val']
        valid_loss /= len(dataloaders['val'])
        valid_losses.append(valid_loss)
        valid_accs.append(valid_acc)


        print(f"Epoch {epoch+1}, Train Loss: {train_loss:.4f}, Train Acc:
            {train_acc:.4f}, Valid Loss: {valid_loss:.4f}, Valid Acc: {
            valid_acc:.4f}")

    return model, train_losses, valid_losses, train_accs, valid_accs
```

6.3 Neural Networks in the Real World

I chose class index 291 (Lion) for my experiment and used a video from National Geographic on YouTube (link). On the validation set, the pretrained model achieves perfect accuracy (1.00), but for the video, the accuracy drops to 0.9595. This makes sense since the video comes from a slightly different domain. To improve performance, we could apply techniques like domain adaptation or data augmentation. Below is my code for downloading and processing the video sequence. Note that I trimmed the video to a few seconds before feeding to my model.

```
import torch
import torchvision.models as models
from torchvision.models.resnet import ResNet50_Weights
import torchvision.transforms as transforms
from PIL import Image
import numpy as np
from tqdm import tqdm
from matplotlib import pyplot as plt

model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)
model.eval()
imagenet_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
       0.225]),
])
from torchvision.datasets import ImageNet

val_dir = '../data/'
imagenet_data = ImageNet(val_dir, split='val', transform=
   imagenet_transform)
val_loader = torch.utils.data.DataLoader(imagenet_data, batch_size=32,
   shuffle=False)
class_idx = 291
specific_class_data = [(img, label) for img, label in imagenet_data if
   label == class_idx]
# Evaluate model on specific class
correct = 0
for inputs, labels in tqdm(specific_class_data[:]):
    inputs = inputs.unsqueeze(0)  # Add batch dimension
    outputs = model(inputs)
    _, preds = torch.max(outputs, 1)
    correct += (preds == labels).sum().item()

print(f"Validation accuracy for class {class_idx}: {correct / 50:.4f}")
```

```
from pytubefix import YouTube

yt = YouTube('https://www.youtube.com/watch?v=OMkEVX23BdM')
stream = yt.streams.get_highest_resolution()
stream.download(output_path='../data/', filename='lion.mp4')
```

```
print('Download complete!')

import cv2
import os

video_path = '../data/lion_trimed.mp4'
output_dir = '../data/lion_frames'
os.makedirs(output_dir, exist_ok=True)

# Extract frames from the video
cap = cv2.VideoCapture(video_path)
frame_count = 0

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break
    frame_path = os.path.join(output_dir, f"frame_{frame_count:04d}.jpg")
    cv2.imwrite(frame_path, frame)
    frame_count += 1

cap.release()
print(f"Extracted {frame_count} frames.")
frame_paths = sorted([os.path.join(output_dir, fname) for fname in os.
   listdir(output_dir)])
frame_preds = []

for frame_path in frame_paths:
    frame = Image.open(frame_path).convert('RGB')
    input_tensor = imagenet_transform(frame).unsqueeze(0)  # Add batch
       dimension
    outputs = model(input_tensor)
    _, preds = torch.max(outputs, 1)
    frame_preds.append(preds.item())

correct_frames = sum([1 for pred in frame_preds if pred == class_idx])
print(f"Accuracy on video frames: {correct_frames / len(frame_preds):.4f}"
   )
```

7 Collaboration

For this homework, I discussed with Daniel Yang (danielya).