CS-GY 6953 / ECE-GY 7123
Giorgi Merabishvili
Homework 1

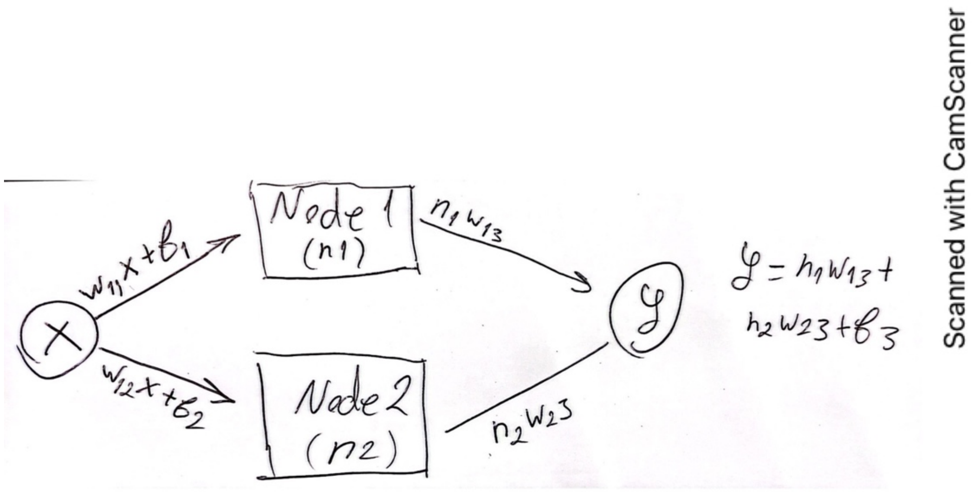
Question 1

1) Expressivity of neural networks. Recall that the functional form for a single neuron is given by y = σ(hw, xi + b, 0), where x is the input and y is the output. In this exercise, assume that x and y are 1-dimensional (i.e., they are both just real-valued scalars) and σ is the unit step activation. We will use multiple layers of such neurons to approximate pretty much any function f. There is no learning/training required for this problem; you should be able to guess/derive the weights and biases of the networks by hand.

a. (1pt) A box function with height h and width δ is the function f(x) = h for 0 < x < δ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activations can realize this function. Draw this network and identify all the weights and biases. (Assume that the output neuron only sums up inputs and does not have a nonlinearity.)

Using a neural network with step activations and two hidden neurons, represent the box function with height h and width δ. The function f(x) is defined as h for 0 < x < δ, and 0 otherwise.

The neural network:



CONNECTION	WEIGHT (w)	BIAS(b)	CORRESPONDING RELATION
Input and Node 1	$w_{11} = 1$	$b_1 = - \delta$	$n_1=w_{11} \cdot x + b_1$
Input and node 2	$w_{12} = 1$	$b_2 = 0$	$n_2=w_{12} \cdot x + b_2$
Node 1 and Output	w13 = -h	$b_3 = 0$	Can be as $w_{13} \cdot x + b_3$
Node 2 and Output	$w_{23} = h$	$b_3 = 0$	Can be as $w_{23} \cdot x + b_3$

Condition	n_1	n_2	Output (y)
X < 0	0	0	y = w13 · n1 + w23 · n2 + b3 = 0
X ≥ δ	1	1	y=w13·n1+w23·n2+b3 =−h+h+0=0
0≤x<δ	1	1	y=w13·n1+w23·n2+b3 =0+h+0=h

To mimic a box function with width δ, the weight w in front of the input x should be set so that the step function activates when x falls within the range (0, δ). Setting w1 and w2 to 1/δ ensures that x is within the range (0, δ), and the step function produces output close to 1, representing the height h of the box. By setting biases to −1/2, the step function activates when the weighted sum of input is greater than zero. This can be changed depending on the desired behavior of the neural network.

The input x is multiplied by the weight 1/δ with the corresponding bias, followed by the step function σ. The same procedure is followed for the second hidden neuron. The output neuron sums the outputs of the two hidden neurons without using any nonlinearity.

b. Now suppose that f is any arbitrary, smooth, bounded function defined over an interval [−B, B]. (You can ignore what happens to the function outside this interval, or just assume it is zero). Use part a to show that this function can be closely approximated by a neural network with a hidden layer of neurons. You don't need a rigorous mathematical proof here; a handwavy argument or even a figure is okay here, as long as you convey the right intuition.

The neural network from part a can be used to closely approximate an arbitrary, smooth, bounded function f defined over the interval [−B, B].

We can place multiple such neural networks along the interval to capture the function's complexity, and the step function allows the network to focus on different parts of the function at different points along the interval.

This is equivalent to having localized approximations using step functions and combining them to mimic the behavior of a smooth function. It works because step functions generate steps that approximate changes in the smooth function.

c. Do you think the argument in part b can be extended to the case of d-dimensional inputs? (i.e., where the input x is a vector – think of it as an image, or text query, etc). If yes, comment on potential practical issues involved in defining such networks. If not, explain why not.

Yes, the argument presented in Part B can be extended to include d-dimensional inputs in the context of neural network modeling. By treating each dimension as an independent input, the approach outlined in Part A can be applied to each dimension separately. However, this scaling poses a number of practical challenges. Notably, the number of weights and biases grows exponentially with the number of dimensions, resulting in significant computational challenges and an increased risk of overfitting. Because of this complexity, more sophisticated strategies for determining the best network architecture must be used. Furthermore, to address these challenges, regularization techniques are required, which aid in moderating the model's complexity and improving its generalization capabilities. Furthermore, dimensionality reduction methods are critical in effectively managing these issues, ensuring that the network remains efficient and practical for high-dimensional data analysis.

Question 2

Let $\hat{y}$ be the output of softmax function, and $\hat{y}_i$ be the i-th element of $\hat{y}$. softmax function is defined :

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$

Considering log-softmax function:

$$\log(\hat{y}_i) = \log\left(\frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}\right)$$

Using properties of logarithms:

$$\log(\hat{y}_i) = z_i - \log\left(\sum_{j=1}^n e^{z_j}\right)$$

Differentiating the log-softmax function with respect to z_j:

$$\frac{\partial \log(\hat{y}_i)}{\partial z_j} = \frac{\partial(z_i - \log(\sum_{k=1}^n e^{z_k}))}{\partial z_j}$$

Here, the term $\frac{\partial z_i}{\partial z_j}$ will be δ_{ij} (the Dirac delta).

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

The derivative of the second term involving the sum depends on whether $j = i$ or not.

If $j = i$, the derivative of $\log(\sum_{k=1}^n e^{z_k})$ with respect to z_i:

$$\frac{e^{z_i}}{\sum_{k=1}^n e^{z_k}} = \hat{y}_i$$

If $j \neq i$, the derivative:

$$\frac{-e^{z_j}}{\sum_{k=1}^n e^{z_k}} = -\hat{y}_j$$

Now, we get:

$$\frac{\partial \log(\hat{y}_i)}{\partial z_j} = \delta_{ij} - \hat{y}_j$$

Now, Jacobian J for the softmax function is obtained by taking the exponential of the log-softmax derivatives:

$$J_{ij} = e^{\frac{\partial \log(\hat{y}_i)}{\partial z_j}} = e^{\delta_{ij} - \hat{y}_j}$$

Simplifying using properties of exponentials:

$$J_{ij} = \frac{e^{z_i}}{\sum_{k=1}^n e^{z_k}}\left(\delta_{ij} - \frac{e^{z_j}}{\sum_{k=1}^n e^{z_k}}\right) = \hat{y}_i(\delta_{ij} - \hat{y}_j)$$

This is the Jacobian of the softmax function with respect to the input vector z.

article amsmath

To begin, the softmax function is defined as:

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Taking the logarithm of both sides:

$$\log(y_i) = \log\left(\frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}\right)$$

$$\log y_i = z_i - \log\sum_{j=1}^{n} e^{z_j}$$

Taking the derivative with respect to z_j:

$$\frac{\partial \log y_i}{\partial z_j} = \frac{\partial z_i}{\partial z_j} - \frac{\partial}{\partial z_j}\log\sum_{k=1}^{n} e^{z_k} \quad (1)$$

The derivative on the first term δ_{ij} (dirac delta) is given by,

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

For the derivative of the second term, we can use the chain rule to obtain:

$$\frac{\partial}{\partial z_j}\log\sum_{k=1}^{n} e^{z_k} = \frac{1}{\sum_{j=1}^{n} e^{z_j}} \cdot e^{z_j}$$

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

$$= y_j$$

Substituting these expressions back into the equation (1):

$$\frac{\partial \log y_i}{\partial z_j} = \delta_{ij} - y_j$$

The Jacobian of y with respect to z is given by:

$$J_{i,j} = \frac{\partial y_i}{\partial z_j} = y_i(\delta_{ij} - y_j)$$

Question 3

untitled43

February 24, 2024

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
```

```
2024-02-23 18:37:29.939768: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
```

[2]:
```
# Load dataset
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
 ↪load_data()

# Normalize the images
train_images = train_images / 255.0
test_images = test_images / 255.0

# Convert labels to one-hot encoding
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

[3]:
```
from tensorflow.keras.layers import Dropout

# Define the model
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(256, activation='relu'),
    Dropout(0.2),
    Dense(128, activation='relu'),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])
```

[4]:
```
# Compile the model
from tensorflow.keras.optimizers import Adam

# Compile the optimized model
optimizer = Adam(learning_rate=0.001)  # Adjust the learning rate as needed
model.compile(optimizer=optimizer,
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])
```

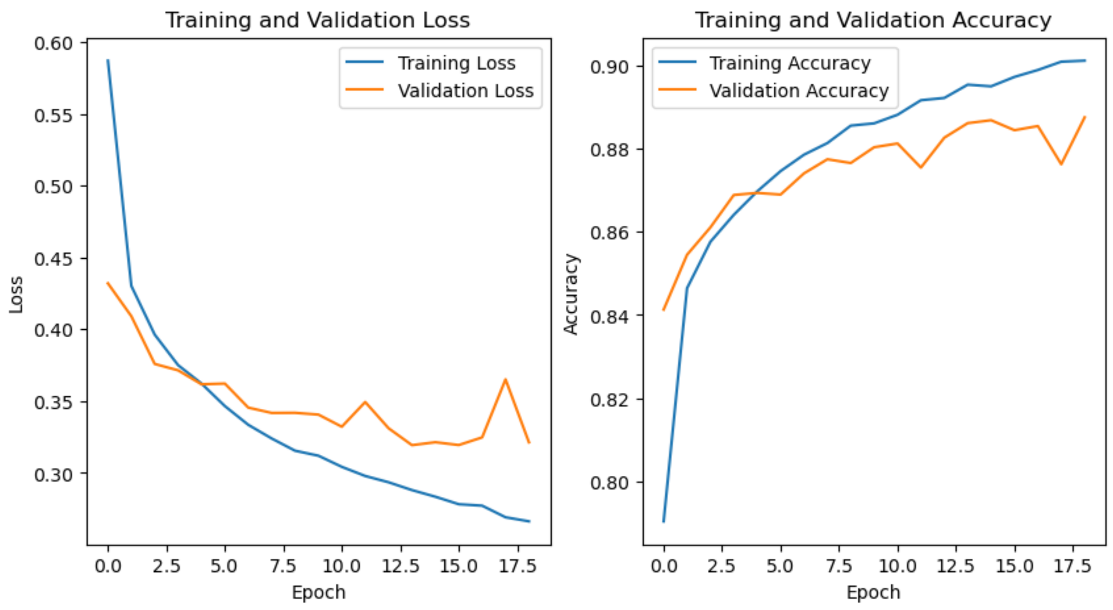
[5]:
```
# Train the model
from tensorflow.keras.callbacks import EarlyStopping

# Early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)

# Train the optimized model with early stopping
history = model.fit(train_images, train_labels, epochs=50,  # Increase epochs
                                        validation_data=(test_images,␣
 ↪test_labels),
                                        callbacks=[early_stopping])
```

```
Epoch 1/50
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5871 -
accuracy: 0.7903 - val_loss: 0.4320 - val_accuracy: 0.8413
Epoch 2/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4303 -
accuracy: 0.8464 - val_loss: 0.4092 - val_accuracy: 0.8545
Epoch 3/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3963 -
accuracy: 0.8576 - val_loss: 0.3759 - val_accuracy: 0.8611
Epoch 4/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3751 -
accuracy: 0.8641 - val_loss: 0.3715 - val_accuracy: 0.8689
Epoch 5/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3623 -
accuracy: 0.8698 - val_loss: 0.3618 - val_accuracy: 0.8694
Epoch 6/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3466 -
accuracy: 0.8747 - val_loss: 0.3622 - val_accuracy: 0.8690
Epoch 7/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3335 -
accuracy: 0.8786 - val_loss: 0.3455 - val_accuracy: 0.8741
Epoch 8/50
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3240 -
accuracy: 0.8814 - val_loss: 0.3418 - val_accuracy: 0.8775
Epoch 9/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3155 -
accuracy: 0.8856 - val_loss: 0.3419 - val_accuracy: 0.8766
Epoch 10/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3120 -
accuracy: 0.8862 - val_loss: 0.3406 - val_accuracy: 0.8804
Epoch 11/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3043 -
accuracy: 0.8882 - val_loss: 0.3322 - val_accuracy: 0.8813
Epoch 12/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2979 -
accuracy: 0.8917 - val_loss: 0.3494 - val_accuracy: 0.8755
Epoch 13/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2935 -
accuracy: 0.8923 - val_loss: 0.3311 - val_accuracy: 0.8827
Epoch 14/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2881 -
accuracy: 0.8955 - val_loss: 0.3194 - val_accuracy: 0.8862
Epoch 15/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2835 -
accuracy: 0.8951 - val_loss: 0.3214 - val_accuracy: 0.8869
Epoch 16/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2783 -
accuracy: 0.8973 - val_loss: 0.3195 - val_accuracy: 0.8845
Epoch 17/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2772 -
accuracy: 0.8990 - val_loss: 0.3248 - val_accuracy: 0.8855
Epoch 18/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2692 -
accuracy: 0.9010 - val_loss: 0.3652 - val_accuracy: 0.8763
Epoch 19/50
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2663 -
accuracy: 0.9013 - val_loss: 0.3214 - val_accuracy: 0.8876
```

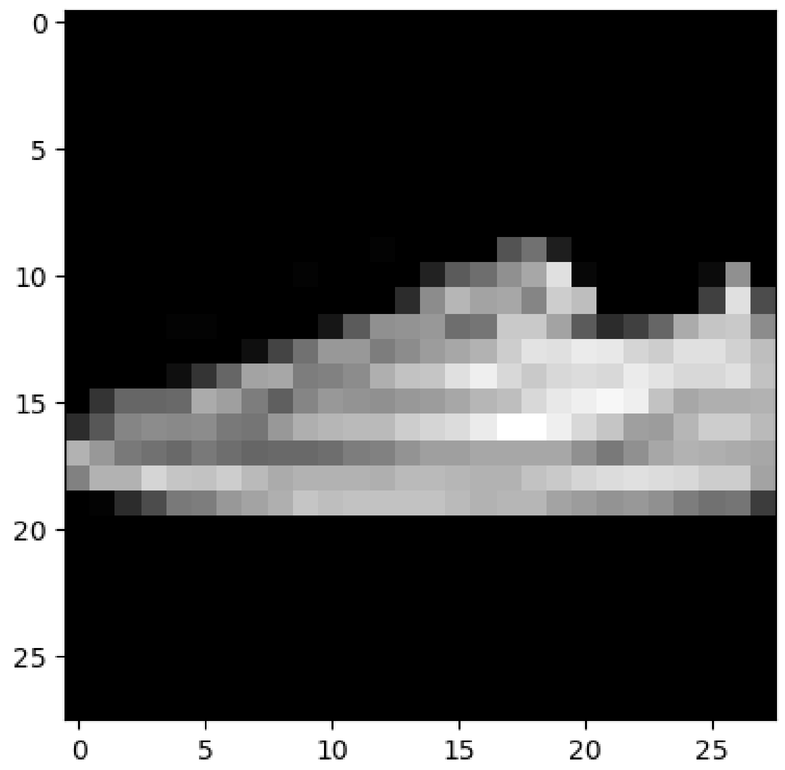
[6]:
```
# Plot training and validation loss
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plot training and validation accuracy
```

```
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

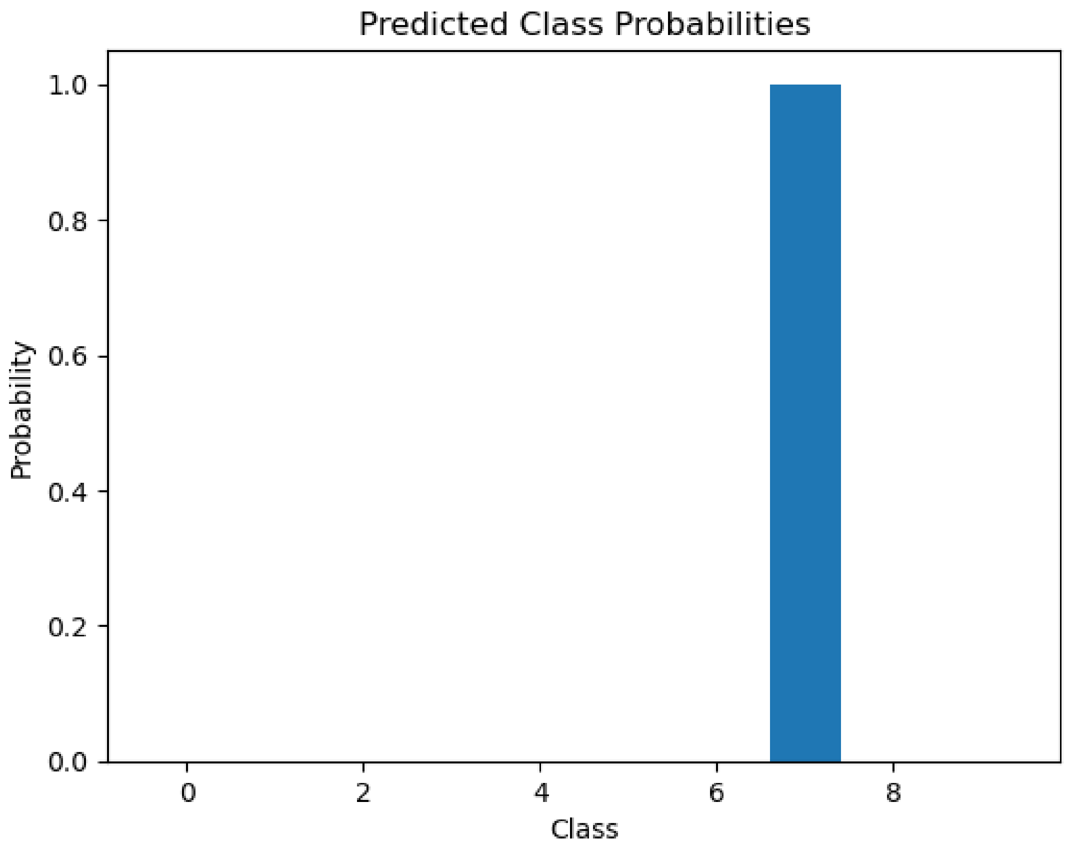


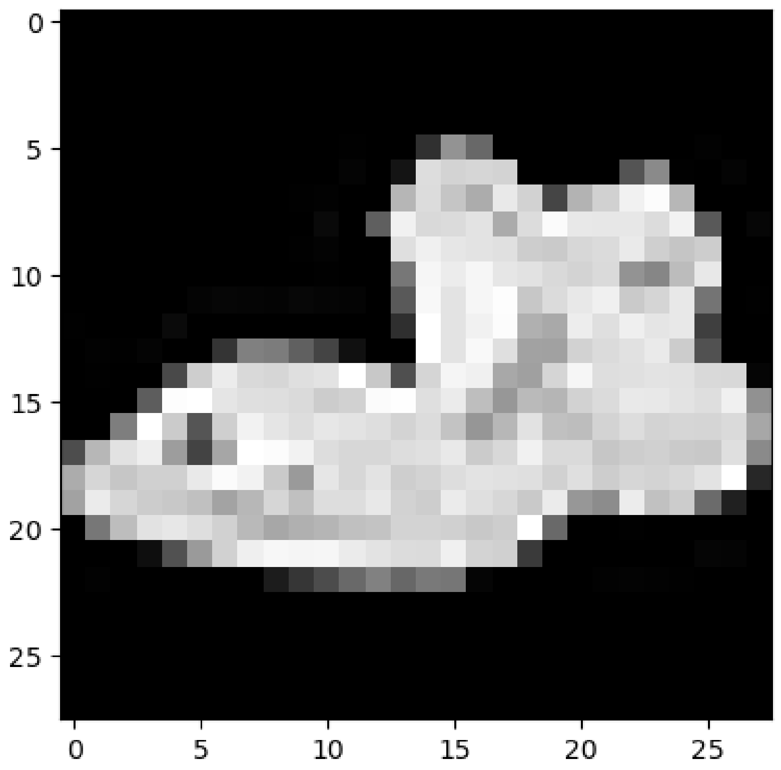
[7]:
```
# Evaluate the model on test set
test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=2)
print(f'Test Accuracy: {test_accuracy*100:.2f}%')

# Predict and visualize for 3 random test images
np.random.seed(0) # For reproducibility
indices = np.random.choice(range(len(test_images)), 3, replace=False)
for i in indices:
    img = test_images[i]
    plt.imshow(img, cmap='gray')
    plt.show()
    prediction = model.predict(np.array([img]))
    plt.bar(range(10), prediction[0])
    plt.ylabel('Probability')
    plt.xlabel('Class')
    plt.title('Predicted Class Probabilities')
    plt.show()
```

```
313/313 - 0s - loss: 0.3194 - accuracy: 0.8862 - 300ms/epoch - 960us/step
Test Accuracy: 88.62%
```

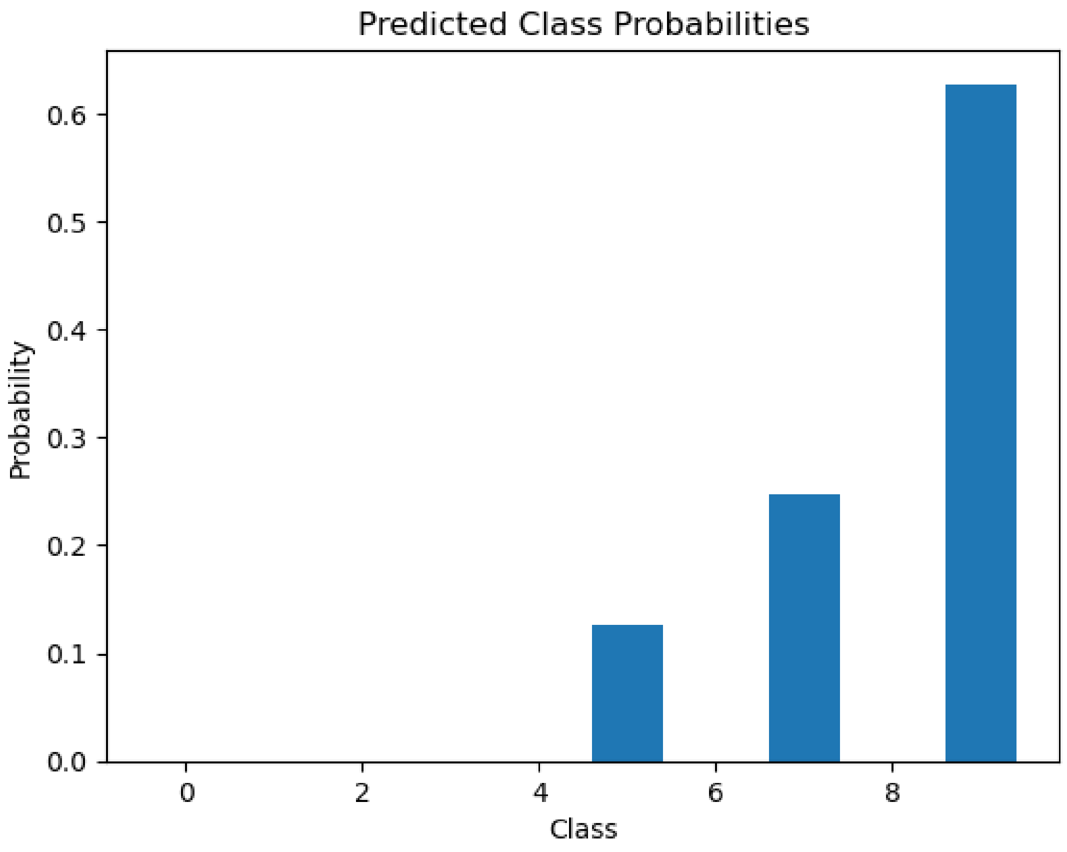


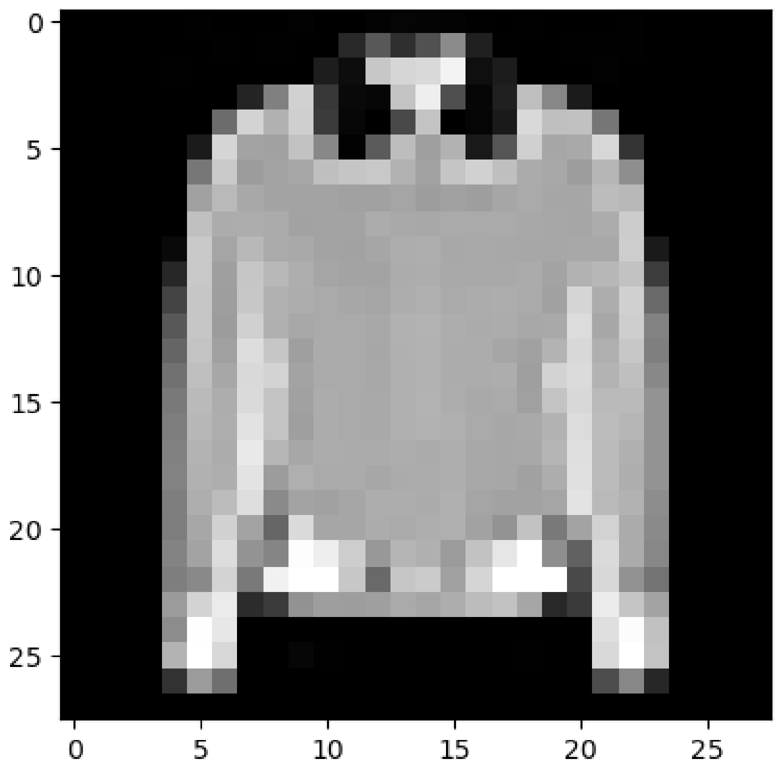
```
1/1 [==============================] - 0s 65ms/step
```


Predicted Class Probabilities
Probability
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
Class

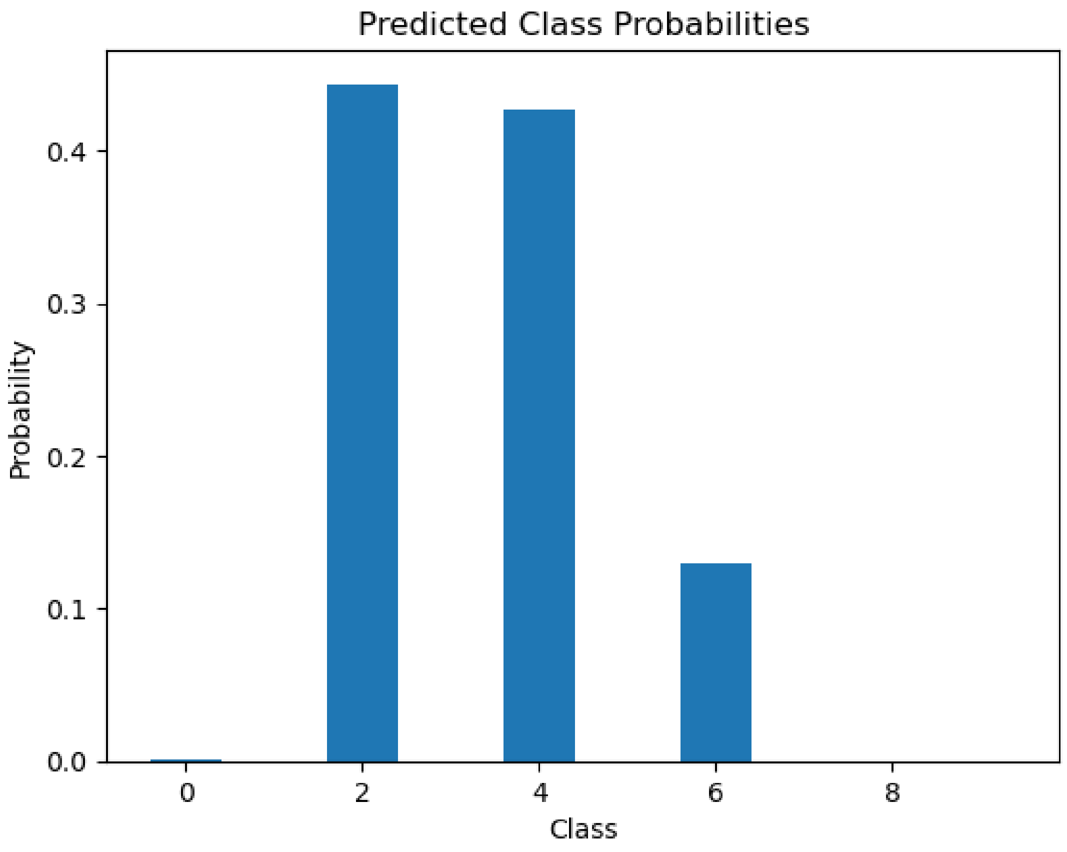


```
1/1 [==============================] - 0s 14ms/step
```


Predicted Class Probabilities
Probability
Class
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
2
4
6
8



```
1/1 [==============================] - 0s 14ms/step
```


Predicted Class Probabilities
Probability
Class
0.0
0.1
0.2
0.3
0.4
0
2
4
6
8

```
[ ]:
```

Question 4

hw1-s24-p4-1-5

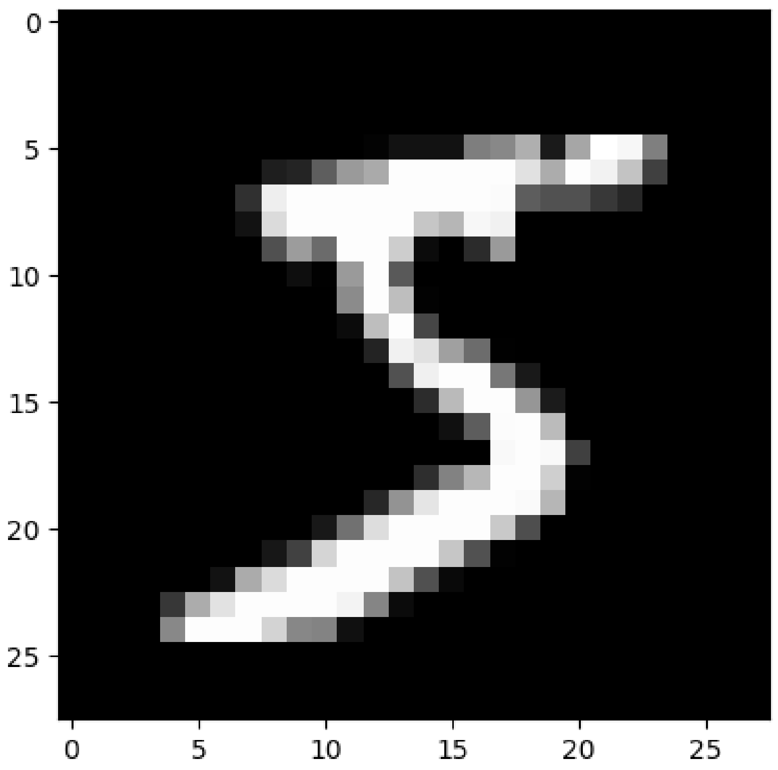
February 24, 2024

In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

[75]:
```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
 ↪load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

[76]:
```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  # Compute the sigmoid function in a numerically stable way.
  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(x >= 0, 1 / (1 + np.exp(-x)), np.exp(x) / (1 + np.exp(x)))

# Compute the derivative of the sigmoid function.
def dsigmoid(x):
    sig = sigmoid(x)
    return sig * (1 - sig)

# Compute the softmax function in a numerically stable way.
```

```
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)

# Compute the cross-entropy loss.
def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))
# Convert an integer to a one-hot encoded vector.
def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/\max(n_{in}, n_{out})$.

[77]:
```
import math

# Import default random number generator from numpy for consistent and␣
↪reproducible initialization.
from numpy.random import default_rng

# Create a random number generator with fixed seed for reproducibility.
rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...

# Define the sizes of the input layer, two hidden layers, and the output layer.
input_size = 784
hidden_size1 = 32
hidden_size2 = 32
output_size = 10

# Initialize weights for each layer using a normal distribution.
# The standard deviation is chosen as 1/sqrt(n), where n is the max number of␣
↪inputs or neurons in the layer pairs,
# to help maintain the scale of the activations across the network.
weights = [
    rng.normal(0, 1/np.sqrt(max(input_size, hidden_size1)), (input_size,␣
  ↪hidden_size1)),
    rng.normal(0, 1/np.sqrt(max(hidden_size1, hidden_size2)), (hidden_size1,␣
  ↪hidden_size2)),
    rng.normal(0, 1/np.sqrt(max(hidden_size2, output_size)), (hidden_size2,␣
  ↪output_size))
]

# Initialize biases for each layer.
# Biases are initialized to zero vectors, which is a common practice. This␣
  ↪initialization does not break the symmetry
# because the weights are initialized randomly.
biases = [
    np.zeros(hidden_size1),
    np.zeros(hidden_size2),
    np.zeros(output_size)
]
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

[78]:
```
def feed_forward_sample(sample, y):

  # Q2. Fill code here.
  # ...
  activations = []

  # Flatten the input sample if not already flat and treat as input activation
  a = sample.flatten()
  activations.append(a)

  # Hidden layer 1: compute weighted sum (z1); apply sigmoid activation function
  z1 = np.dot(a, weights[0]) + biases[0]
  a1 = sigmoid(z1)
  activations.append(a1)

  # Hidden layer 2: similar process as in hidden layer 1
  z2 = np.dot(a1, weights[1]) + biases[1]
  a2 = sigmoid(z2)
  activations.append(a2)

  # Output layer: compute output activations using softmax
  z_out = np.dot(a2, weights[2]) + biases[2]
  a_out = softmax(z_out)
  activations.append(a_out) # Store final output activation

  # Convert the highest probability from softmax output to one-hot encoded guess
  one_hot_guess = integer_to_one_hot(np.argmax(a_out), 10)

  # Calculate the cross-entropy loss between the predicted output and true label
  loss = cross_entropy_loss(integer_to_one_hot(y, 10), a_out)

  return loss, one_hot_guess


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0]) # Array to hold loss for each sample
  one_hot_guesses = np.empty((x.shape[0], 10)) # Array to hold one-hot guesses␣
 ↪for each sample




  # Iterate over dataset, feeding each sample forward and recording the loss␣
 ↪and guess
  for i in range(x.shape[0]):
    loss, one_hot_guess = feed_forward_sample(x[i], y[i])
    losses[i] = loss
    one_hot_guesses[i] = one_hot_guess
  # Calculate the number of correct guesses
  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")

  # Print the average loss and accuracy for the dataset
  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0],␣
 ↪"(", correct_guess_percent, "%)")

#Feed all training data through the neural network.
def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

#Feed all test data through the neural network.
def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

# Execute the function to feed forward all test data and print results
feed_forward_test_data()
```

```
Feeding forward all test data…

Average loss: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

```
[79]: def train_one_sample(sample, y, learning_rate=0.003):
          a = sample.flatten()

          # Store each layer's activations to calculate gradient
          activations = [] # List to store activations of each layer

          # Compute activation for the first hidden layer
          z1 = np.dot(a, weights[0]) + biases[0]
          a1 = sigmoid(z1)
          activations.append(a1)

          # Compute activation for the second hidden layer
          z2 = np.dot(a1, weights[1]) + biases[1]
          a2 = sigmoid(z2)
          activations.append(a2)

          # Compute output layer activation
          z_out = np.dot(a2, weights[2]) + biases[2]
          a_out = softmax(z_out)
          activations.append(a_out)

          # Convert output probabilities to one-hot encoded vector representing␣
         ↪predicted class
          one_hot_guess = integer_to_one_hot(np.argmax(a_out), 10)
          loss = cross_entropy_loss(integer_to_one_hot(y, 10), a_out)

          # Backward pass
          weight_gradients = [0, 0, 0]
          bias_gradients = [0, 0, 0]
```

```
  # Output layer
  delta_out = a_out - integer_to_one_hot(y, 10)
  weight_gradients[2] = np.outer(a2, delta_out)
  bias_gradients[2] = delta_out

  # Hidden layer 2
  delta2 = np.dot(delta_out, weights[2].T) * dsigmoid(z2)
  weight_gradients[1] = np.outer(a1, delta2)
  bias_gradients[1] = delta2

  # Hidden layer 1
  delta1 = np.dot(delta2, weights[1].T) * dsigmoid(z1)
  weight_gradients[0] = np.outer(a, delta1)
  bias_gradients[0] = delta1

  # Update weights & biases based on the calculated gradient
  for i in range(3):
    weights[i] -= weight_gradients[i] * learning_rate
    biases[i] -= bias_gradients[i] * learning_rate
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

[80]:
```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  # ...

  # Iterate over each sample in the training dataset
  for i in range(x_train.shape[0]):
    # Train the model on a single sample and update its parameters
    train_one_sample(x_train[i], y_train[i], learning_rate)

  print("Finished training.\n")

# After defining the training function, evaluate model on test dataset
feed_forward_test_data()

def test_and_train():
  # Train the model for one epoch on the training data
  train_one_epoch()
  # Test the model on the test dataset and print the evaluation metrics
  feed_forward_test_data()

# Train and test the model for a fixed number of epochs, in this case, 3 epochs
```

```python
for i in range(3):
  test_and_train()
```

```
Feeding forward all test data…

Average loss: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)

Training for one epoch over the training dataset…
Finished training.

Feeding forward all test data…

Average loss: 0.96
Accuracy (# of correct guesses): 6653.0 / 10000 ( 66.53 %)

Training for one epoch over the training dataset…
Finished training.

Feeding forward all test data…

Average loss: 0.96
Accuracy (# of correct guesses): 6596.0 / 10000 ( 65.96 %)

Training for one epoch over the training dataset…
Finished training.

Feeding forward all test data…

Average loss: 0.84
Accuracy (# of correct guesses): 7257.0 / 10000 ( 72.57 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.