ECE 520.438 & 520.638: Deep Learning
Homework 2
Spring 2024

Tanay Nayak

March 4, 2024

1 Part 1 (20 points)

1.1 Answer the following questions.

1. What kind of decision boundary is learned by perceptrons?

   Perceptrons learn linear decision boundaries. This means they can only classify data that is linearly separable.

2. How can we use perceptrons to do C-class classification?

   For C-class classification, a common approach is to use the ”one-vs-all” (OvA) strategy, where C perceptrons are trained, each to recognize a separate class.

3. Briefly explain stochastic gradient descent.

   Stochastic gradient descent (SGD) is an optimization method that updates the model’s parameters using only a single sample, or a few samples, which is thought to represent the entire dataset at each iteration.

4. Briefly explain mini-batch gradient descent.

   Mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent. It updates parameters using a small subset of the training data, called a ”mini-batch”, at each iteration.

5. Let $\sigma(a) = \frac{1}{1+e^{-a}}$. Show that $\sigma'(a) = \sigma(a)[1 - \sigma(a)]$.

To find the derivative $\sigma'(a)$, we start by differentiating $\sigma(a)$ with respect to a:

$$\begin{aligned}
\sigma(a) &= \frac{1}{1+e^{-a}} \\
\implies \sigma'(a) &= \frac{d}{da}\left(\frac{1}{1+e^{-a}}\right) \\
&= \frac{d}{da}\left(1+e^{-a}\right)^{-1} \\
&= -(1+e^{-a})^{-2} \cdot \frac{d}{da}\left(e^{-a}\right) \\
&= -(1+e^{-a})^{-2} \cdot (-e^{-a}) \\
&= \frac{e^{-a}}{(1+e^{-a})^2} \\
&= \frac{1}{1+e^{-a}} \cdot \frac{e^{-a}}{1+e^{-a}} \\
&= \sigma(a) \cdot \frac{e^{-a}}{1+e^{-a}} \\
&= \sigma(a) \cdot \frac{1+e^{-a}-1}{1+e^{-a}} \\
&= \sigma(a)\left(1 - \frac{1}{1+e^{-a}}\right) \\
&= \sigma(a)(1-\sigma(a)).
\end{aligned}$$

Thus, we have shown that $\sigma'(a) = \sigma(a)(1-\sigma(a))$.

6. What is an Epoch?

An epoch is a single pass through the entire training dataset by the learning algorithm.

7. If the dataset size is 2000 samples and the batch size is 50, then an epoch will consist of how many iterations?

An epoch will consist of $\frac{2000}{50} = 40$ iterations.

8. Briefly describe Dropout.

Dropout is a regularization technique where randomly selected neurons are ignored during training to prevent overfitting by providing a form of ensemble learning.

9. What is early stopping?

Early stopping is a form of regularization where training is stopped as soon as the validation error begins to increase, to prevent overfitting.

10. Why is regularization used during training a neural network?

    Regularization is used to prevent overfitting, ensuring the model generalizes well to unseen data.

11. What are different ways in which we can reduce over-fitting in neural networks?

    Methods to reduce overfitting include using more data, data augmentation, regularization techniques (like L1, L2, and dropout), and early stopping.

12. Why do we use batch normalization?

    Batch normalization is used to improve the speed, performance, and stability of neural networks by normalizing the inputs of each layer.

Part 2: Perceptron (40 points)

In this problem, you will classify handwritten digits using the MNIST Database. Please download the four files found there, these will be used for this homework. To reduce computation time, please use only the first 20,000 training images/labels and only the first 2,000 testing images/labels.

Read in the data from files. Each image is 28 x 28 pixels. An image can be viewed as a 784-dimensional vector of pixel intensities. For each image, append a `1` to the beginning of each vector; this will act as an intercept term (for bias). Use the gradient descent algorithm for Perceptron derived in class, to classify given `x ∈ R^785` whether a digit's label is `k` or if it is some "other" digit, i.e. `k ≠ {0, ..., 9}`. For instance, if you are classifying "2", you would designate `y = 2` as the positive class, and all other digits as the negative class. For each image, do this two-way classification for all 10 digits. You will train 10 perceptrons that will collectively learn to classify the handwritten digits in the MNIST dataset. Each perceptron will have 785 inputs and one output.

1. Report the test accuracy for each of the 10 two-way classifications on the test set.
2. For each image in the test set, report the overall test accuracy. An example will be considered labeled correctly if the perceptron classification of the true label has the highest probability. So for instance, if the true label was `{2}` for an image, you would count it as correctly classified if the perceptron test of `{2}` vs. `{0,1,3,4,5,6,7,8,9}` had the highest probability of all the 10 2-way classifications.

In [ ]:

```
import os
import struct
import numpy as np

# Paths to the dataset files
files = {
    'train_images': 'data/train-images.idx3-ubyte',
    'train_labels': 'data/train-labels.idx1-ubyte',
    'test_images': 'data/t10k-images.idx3-ubyte',
    'test_labels': 'data/t10k-labels.idx1-ubyte'
}

def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims)
        return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)

# Preparing the dataset
```

```
def prepare_dataset(files, limit_train=20000, limit_test=2000):
    train_images = read_idx(files['train_images'])
    train_labels = read_idx(files['train_labels'])
    test_images = read_idx(files['test_images'])
    test_labels = read_idx(files['test_labels'])

    train_images = train_images[:limit_train]
    train_labels = train_labels[:limit_train]
    test_images = test_images[:limit_test]
    test_labels = test_labels[:limit_test]

    train_images = train_images.reshape(limit_train, 784)
    test_images = test_images.reshape(limit_test, 784)

    train_images = np.insert(train_images, 0, 1, axis=1)
    test_images = np.insert(test_images, 0, 1, axis=1)

    return train_images, train_labels, test_images, test_labels

train_images, train_labels, test_images, test_labels = prepare_dataset(files

(train_images.shape, train_labels.shape, test_images.shape, test_labels.shap
```

Out[ ]:

```
((20000, 785), (20000,), (2000, 785), (2000,))
```

In [ ]:

```
from tqdm.notebook import tqdm

def train_perceptron(digit, X, y, learning_rate=1.0, epochs=1000):
    """
    Trains a perceptron model using the given features and target labels.
    Args:
    X (numpy.ndarray): The feature vectors.
    y (numpy.ndarray): The target labels.
    learning_rate (float): The learning rate for weight updates.
    epochs (int): The number of times to run through the training data.
    Returns:
    numpy.ndarray: The weights of the trained perceptron model.
    """
    weights = np.zeros(X.shape[1])

    for epoch in tqdm(range(epochs), desc="Training Perceptron for digit " +
        for i in range(X.shape[0]):
            prediction = np.dot(X[i], weights) >= 0
            if (prediction != y[i]):
                weights += learning_rate * (y[i] - prediction) * X[i]
    return weights

def train_10_perceptrons(X_train, y_train):
    perceptron_weights = []
    for digit in tqdm(range(10), desc="Training perceptrons for each digit")
        y_binary = (y_train == digit).astype(int)
        weights = train_perceptron(digit, X_train, y_binary)
        perceptron_weights.append(weights)
    return perceptron_weights

perceptron_weights = train_10_perceptrons(train_images, train_labels)
```

```
print(f'Number of perceptrons trained: {len(perceptron_weights)}')
print(f'Shape of each perceptron weight vector: {perceptron_weights[0].shape
```

```
Training perceptrons for each digit:   0%|          | 0/10 [00:00<?, ?it/s]
Training Perceptron for digit 0:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 1:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 2:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 3:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 4:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 5:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 6:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 7:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 8:   0%|          | 0/1000 [00:00<?, ?it/s]
Training Perceptron for digit 9:   0%|          | 0/1000 [00:00<?, ?it/s]
Number of perceptrons trained: 10
Shape of each perceptron weight vector: (785,)
```

In [ ]:

```
def two_way_classification_accuracy(weights, X, y, positive_class):
    """
    Computes the accuracy of a two-way classification for a given positive c

    Args:
    weights (numpy.ndarray): The weights of the perceptron.
    X (numpy.ndarray): The feature vectors of the test set.
    y (numpy.ndarray): The true labels of the test set.
    positive_class (int): The digit considered as the positive class.

    Returns:
    float: The accuracy of the two-way classification.
    """
    predictions = np.dot(X, weights) >= 0
    true_positive_class = y == positive_class
    accuracy = np.mean(predictions == true_positive_class)
    return accuracy

two_way_accuracies = []
for digit in range(10):
    weights = perceptron_weights[digit]
    accuracy = two_way_classification_accuracy(weights, test_images, test_la
    two_way_accuracies.append(accuracy)
    print(f'Accuracy for digit {digit}: {accuracy:.2f}')
```

```
Accuracy for digit 0: 0.99
Accuracy for digit 1: 0.99
Accuracy for digit 2: 0.96
Accuracy for digit 3: 0.96
Accuracy for digit 4: 0.97
Accuracy for digit 5: 0.97
Accuracy for digit 6: 0.97
Accuracy for digit 7: 0.96
Accuracy for digit 8: 0.90
Accuracy for digit 9: 0.94
```

In [ ]:

```
def predict_perceptron(weights, X):
    """Make predictions using learned weights"""
    return np.dot(X, weights)
```

```
def calculate_overall_accuracy(perceptron_weights, X_test, y_test):
    """
    Calculate the overall accuracy on the test set where an image is conside
    correctly labeled if the perceptron for the true label outputs the highe

    Args:
    perceptron_weights (list of numpy.ndarray): The list of weights for each
    X_test (numpy.ndarray): The test set images.
    y_test (numpy.ndarray): The true labels for the test set.

    Returns:
    float: The overall test accuracy.
    """
    perceptron_outputs = np.array([predict_perceptron(weights, X_test) for w

    predicted_classes = np.argmax(perceptron_outputs, axis=0)

    overall_accuracy = np.mean(predicted_classes == y_test)

    return overall_accuracy

overall_test_accuracy = calculate_overall_accuracy(perceptron_weights, test_

print(f'The overall test accuracy is: {overall_test_accuracy:.2f}')
```

```
The overall test accuracy is: 0.84
```

Part 3: Logistic Regression (40 points)

Logistic regression is a binary classification method which can be modeled as using a single neuron reading in an input vector $x \in \mathbb{R}^d$ and parameterized by weight vector $w \in \mathbb{R}^d$, where the neuron outputs the probability of the class being $y = 1$ given x

$$P(y=1|x) = g_w(x) = \frac{1}{1+\exp(-w^T x)} = \sigma(w^T x)$$

$$P(y=0|x) = 1 - P(y=1|x) = 1 - g_w(x).$$

Given $(x^{(i)}, y^{(i)})_{i=1}^N$, the Cross Entropy Loss function is defined as follows

$$J(w) = -\sum_{i=1}^{N} \left(y^{(i)} \log(g_w(x^{(i)})) + (1-y^{(i)}) \log(1-g_w(x^{(i)})) \right),$$

where N denotes the total number of training samples. We will optimize this cost function via gradient descent.

1. Show that the gradient of the cost function with respect to the parameter w is:

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{N} x_j^{(i)} (g_w(x^{(i)}) - y^{(i)}).$$

Show your work.

2. Using the gradient derived for Logistic Regression cross entropy loss, use gradient descent to classify given $x \in \mathbb{R}^{785}$ whether a digit's label is k or if it is some "other" digit, i.e. $k \neq \{0, \ldots, 9\}$. Report the test accuracy for each of the 10 two-way classifications on the test set.

3. For each image in the test set, report the overall test accuracy. An example will be considered labeled correctly if the perceptron classification of the true label has the highest probability. So for instance, if the true label was $\{2\}$ for an image, you would count it as correctly classified if the perceptron test of $\{2\}$ vs. $\{0, 1, 3, 4, 5, 6, 7, 8, 9\}$ had the highest probability of all the 10 2-way classifications.

To derive the gradient of the cost function $J(w)$ with respect to the parameter w_j, let's start with a cost function defined as the sum of the errors between the predictions $g_w(x^{(i)})$ and the true values $y^{(i)}$, squared and summed over all N samples. Assuming a mean squared error (MSE) cost function for simplicity:

$$J(w) = \frac{1}{2} \sum_{i=1}^{N} (g_w(x^{(i)}) - y^{(i)})^2$$

The gradient of $J(w)$ with respect to w_j is:

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial}{\partial w_j} \frac{1}{2} (g_w(x^{(i)}) - y^{(i)})^2$$

Applying the chain rule gives:

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{N} (g_w(x^{(i)}) - y^{(i)}) \frac{\partial}{\partial w_j} (g_w(x^{(i)}) - y^{(i)})$$

Since $y^{(i)}$ does not depend on w_j, its derivative with respect to w_j is zero. The derivative of $g_w(x^{(i)})$ with respect to w_j depends on the form of g_w. If we assume $g_w(x)$ is a linear function of w, then $g_w(x^{(i)}) = w^T x^{(i)}$, and the derivative is just the corresponding feature value $x_j^{(i)}$. Therefore, we have:

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{N} (g_w(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

This gives us the gradient of the cost function with respect to the parameter w_j.

In [ ]:

```
import numpy as np
from tqdm.notebook import tqdm
```

```
def sigmoid(z):
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))

def compute_cost(X, y, w, epsilon=1e-5):
    m = X.shape[0]
    predictions = sigmoid(np.dot(X, w))
    cost = -(1/m) * np.sum(y * np.log(predictions + epsilon) + (1 - y) * np.
    return cost

def gradient_descent(X, y, w, learning_rate, iterations):
    m = X.shape[0]
    cost_history = []

    for _ in tqdm(range(iterations), desc='Gradient Descent Iterations'):
        predictions = sigmoid(np.dot(X, w))
        error = predictions - y
        gradient = np.dot(X.T, error) / m
        w -= learning_rate * gradient
        cost_history.append(compute_cost(X, y, w))

    return w, cost_history

def predict(X, w):
    predictions = sigmoid(np.dot(X, w))
    return [1 if p >= 0.5 else 0 for p in predictions]

learning_rate = 0.1
iterations = 1000

trained_models = []
test_accuracies = []

for digit in tqdm(range(10), desc="Training logistic regression models for e
    print(f"Training model for digit {digit} vs all")
    y_train_binary = (train_labels == digit).astype(int)
    y_test_binary = (test_labels == digit).astype(int)

    initial_weights = np.zeros(train_images.shape[1])

    final_weights, cost_history = gradient_descent(train_images, y_train_bir

    trained_models.append(final_weights)

    y_pred_train = predict(train_images, final_weights)
    train_accuracy = np.mean(y_train_binary == y_pred_train)

    y_pred_test = predict(test_images, final_weights)
    test_accuracy = np.mean(y_test_binary == y_pred_test)

    test_accuracies.append(test_accuracy)

for digit, accuracy in enumerate(test_accuracies):
    print(f"Test accuracy for digit {digit}: {accuracy:.4f}")
```

```
Training logistic regression models for each digit:   0%|          | 0/10 [0
0:00<?, ?it/s]
Training model for digit 0 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 1 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 2 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 3 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 4 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 5 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 6 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 7 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 8 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Training model for digit 9 vs all
Gradient Descent Iterations:   0%|          | 0/1000 [00:00<?, ?it/s]
Test accuracy for digit 0: 0.9895
Test accuracy for digit 1: 0.9915
Test accuracy for digit 2: 0.9735
Test accuracy for digit 3: 0.9655
Test accuracy for digit 4: 0.9745
Test accuracy for digit 5: 0.9230
Test accuracy for digit 6: 0.9730
Test accuracy for digit 7: 0.9660
Test accuracy for digit 8: 0.9270
Test accuracy for digit 9: 0.8965
```

In [ ]:

```python
def calculate_overall_test_accuracy(models, X_test, y_test):
    """
    Calculate the overall test accuracy. An image is considered correctly cl
    if the model corresponding to the true label gives the highest probabili

    Args:
    models (list of numpy.ndarray): List of weight vectors for each logistic
    X_test (numpy.ndarray): Test set images.
    y_test (numpy.ndarray): True labels for the test set.

    Returns:
    float: The overall test accuracy.
    """
    probabilities = np.array([sigmoid(np.dot(X_test, w)) for w in models])

    predicted_classes = np.argmax(probabilities, axis=0)

    overall_accuracy = np.mean(predicted_classes == y_test)

    return overall_accuracy


overall_test_accuracy = calculate_overall_test_accuracy(trained_models, test
print(f"The overall test accuracy is: {overall_test_accuracy:.4f}")
```

```
The overall test accuracy is: 0.8105
```

In [ ]:

```python

```