PROJECT 3 REPORT

SAM LEVINE

1. TASK 3A: K-Means (30pt)

1.1. Euclidean Distance Function.

The K-Means algorithm segments data into groups (clusters) based on their similarities, and the first step towards achieving this is to accurately calculate the distance between each point in the dataset and the current centroids (initial guesses of cluster centers).

The euclidean_dist function is implemented to fulfill this requirement. It takes two inputs: a single data point (`cur_point`) and a dataset (`dataset`), and outputs the Euclidean distances between `cur_point` and every point in the dataset. This distance measure forms the basis for assigning points to the nearest centroid, a critical step in the K-Means algorithm.

The Euclidean distance between two points in Euclidean space is the length of a line segment between the two points. It can be calculated from the Cartesian coordinates of the points using the Pythagorean theorem. The formula for calculating the Euclidean distance between two points p and q, each having m dimensions, is given by:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_m - q_m)^2}$$

The implementation involves subtracting `cur_point` from every point in dataset, squaring the result, summing these squared differences across each dimension (`axis=1`), and finally taking the square root of these sums to get the Euclidean distances.

```
def euclidean_dist(cur_point, dataset):
    '''
    cur_point has dimensions (m,), dataset has dimensions (n, m), and output
will be of size (n,).
    '''
    dists = None
    # Your Code Starts Here
    diff = dataset - cur_point
    squared_dists = np.sum(np.square(diff), axis=1)
    dists = np.sqrt(squared_dists)
    # End of Your Code

    return dists
```

1.2. K-Means Implementation.

1.2.1. *The `fit` function.*

The `fit` method encompasses the entire workflow of the K-Means clustering algorithm, divided into two main steps: initialization of centroids using the k-means++ method and iterative refinement of these centroids.

1. Centroid Initialization (K-Means++ Method): The first centroid is randomly selected from the dataset. Subsequent centroids are chosen with a probability proportional to the square of their distance from the nearest existing centroid. This method aims to space out the initial centroids, potentially leading to more effective clustering.
2. Iterative Optimization: The function enters a loop that repeatedly assigns data points to the nearest centroid and then updates each centroid to the mean position of the points assigned to it. This process continues until the centroids no longer significantly change between iterations or a maximum number of iterations is reached. This iterative refinement is central to the K-Means algorithm, ensuring that clusters are as coherent and distinct as possible.

The effectiveness of the fit function in finding meaningful clusters heavily depends on the initialization step and the iterative approach to refine centroids based on the current clustering.

```
def fit(self, X_train):
    """
    # Step 1:
    Initialize the centroids, using the "k-means++" method, where a random
datapoint is selected as the first,
     then the rest are initialized w/ probabilities proportional to their
distances to the first
    Pick a random point from train data for first centroid
    """
    # Your Code Starts Here
    n_samples, n_features = X_train.shape
    self.centroids = np.zeros((self.n_clusters, n_features))
    first_centroid_idx = np.random.choice(n_samples)
    self.centroids[0] = X_train[first_centroid_idx]

    for k in range(1, self.n_clusters):
                dist_sq  =  np.array([min([np.inner(c-x,  c-x)  for  c  in
self.centroids[:k]]) for x in X_train])
        probs = dist_sq / dist_sq.sum()
        cum_probs = probs.cumsum()
        # r = np.random.rand()
        r = uniform(0, 1)
        for j, p in enumerate(cum_probs):
            if r < p:
                self.centroids[k] = X_train[j]
                break
    # End of Your Code

    """
    Step 2:
   Iterate, adjusting centroids until converged (new centroids are the same
as previous centroids)
    or until passed max_iter
    """
    iteration = 0
    prev_centroids = None
```

```
    # Your Code Starts Here
    for iteration in range(self.max_iter):
        clusters = [[] for _ in range(self.n_clusters)]
        for x in X_train:
            distances = np.linalg.norm(x - self.centroids, axis=1)
            closest_centroid = np.argmin(distances)
            clusters[closest_centroid].append(x)

        prev_centroids = self.centroids.copy()
        for i in range(self.n_clusters):
            self.centroids[i] = np.mean(clusters[i], axis=0) if clusters[i]
else prev_centroids[i]

        if np.all(prev_centroids == self.centroids):
            break
    # End of Your Code
```

1.2.2. *The `predict` function.*

Once the model is trained, the predict function assigns data points to the nearest centroid, effectively classifying them into the learned clusters. This function calculates the Euclidean distance between each point in a new dataset and all centroids, assigning each point to the centroid with the shortest distance. The result is a set of cluster indices indicating the cluster membership for each data point.

This function allows the model to be applied to new data, extending the utility of the K-Means algorithm beyond the initial dataset on which it was trained.

```
def predict(self,X):
    """
    Assign each data point to the nearest centroid.

    return:
        centroids: a list of the final centroids.
       centroid_idxs: a list of the indices of the centroids that each data
point is assigned to.
    """
    centroids = []
    centroid_idxs = []

    # Your Code Starts Here
    distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
    centroid_idxs = np.argmin(distances, axis=1)
    # End of Your Code

    return self.centroids, centroid_idxs
```

1.3. **Results.**

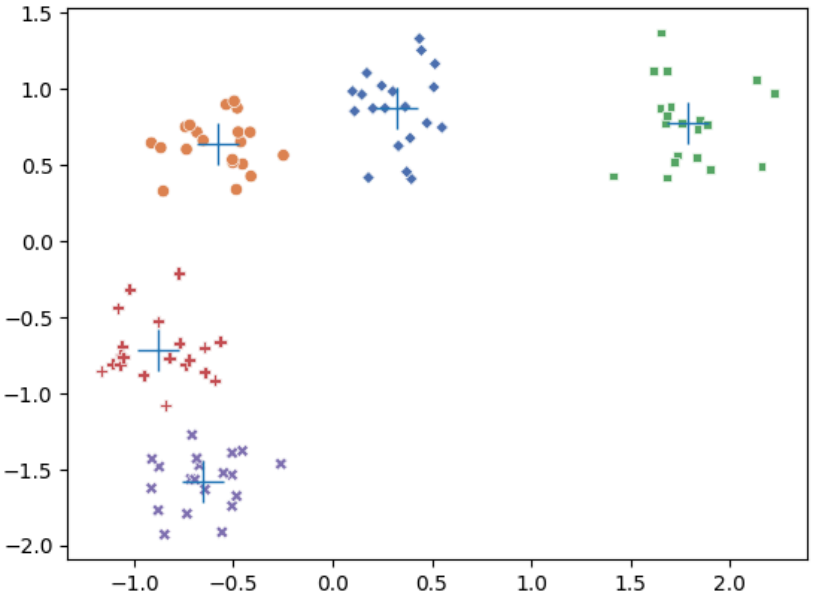

FIGURE 1. Testing and visualizing our k-means algorithm.

2. TASK 3B: PRINCIPAL COMPONENT ANALYSIS (30PT)

2.1. **Linear Classifier (Provided).**

Before implementing and applying PCA, a baseline performance metric was established using a simple linear classifier. This classifier serves as a benchmark to assess the effectiveness of PCA in improving model performance on the MNIST dataset. The training process involved 200 iterations, with the classifier's weights and biases updated to minimize the cross-entropy loss. The model's performance was evaluated based on classification accuracy, both on the training set and a held-out test set.

The simple linear classifier achieved a training accuracy of approximately 57.6% and a test accuracy of 57.6%, using the subsampled 27 features. These results establish a baseline against which the effectiveness of PCA for dimensionality reduction and potential improvement in classification performance can be compared.

2.2. **PCA Implementation.**

```
class PCA:
    def __init__(self, n_components):
        # number of principal components to keep when we apply PCA
        assert n_components <= 784
        self.n_components = n_components

        self.feature_means = None
        self.U, self.S, self.Vt = None, None, None
        self.U_subset, self.S_subset, self.Vt_subset = None, None, None
```

2.2.1. *The `fit` function.*

The fit function is responsible for learning the model parameters from the dataset. It executes the following steps:

1. Data Centering: The first step involves centering the data by subtracting the mean of each feature from the dataset. This ensures that the PCA operates on a mean-centered dataset, which is crucial for capturing the variance in the data accurately.
2. Singular Value Decomposition (SVD): After centering the data, the Singular Value Decomposition (SVD) of the centered data is computed. SVD decomposes the dataset into three matrices (U, S, and V^T), where U and V^T are orthogonal matrices representing the left and right singular vectors, and S is a diagonal matrix containing the singular values.
3. Truncation of SVD: To achieve dimensionality reduction, only the first `n_components` singular values (and corresponding singular vectors) are kept. This truncation captures the most significant modes of variation in the data while reducing its dimensionality.

```
def fit(self, X):
    """
    Steps:
    - Center the data (compute, store, and subtract the mean of each feature)
    - Take the SVD of the centered data
     - Truncate the SVD by keeping only the first n_components (singular
values)
    """
    n_samples, n_features = X.shape

    ## Your Code Starts Here ##
    # center the data per feature
    self.feature_means = X.mean(axis=0)
    X_centered = X - self.feature_means

    # take SVD
    self.U, self.S, self.Vt = np.linalg.svd(X_centered, full_matrices=False)

    # store truncated SVD
    self.U_subset = self.U[:, :self.n_components]
    self.S_subset = self.S[:self.n_components]
    self.Vt_subset = self.Vt[:self.n_components, :]

    ## End of Your Code ##
```

2.2.2. *The `predict` function.*

Once the PCA model is trained, the predict function applies the learned transformation to new or existing data to reduce its dimensionality:

1. Centering: The data to be transformed is first centered using the feature means stored during the `fit` process.
2. Projection: The centered data is then projected onto the retained principal components (the right singular vectors corresponding to the largest singular values kept during truncation). This projection results in a new dataset of reduced dimensionality, where each dimension corresponds to a principal component.

The transformation is mathematically represented as:

$$X_{\text{projected}} = (X - \mu)V_{\text{subset}}^T$$

where X is the centered data, μ is the mean of each feature (`self.feature_means`), and V_{subset}^T represents the matrix containing the right singular vectors corresponding to the largest singular values.

```
def predict(self, X):
    """
    Steps:
    - Center the data (subtract the mean of each feature, stored in the fit)
    - Project the centered data onto the principal components

    Returns:
    - Projected data of shape (X.shape[0], n_components)
    """
    if self.feature_means is None:
        raise ValueError('fit the PCA model first')

    ## Your Code Starts Here ##
    # center X
    X_centered = X - self.feature_means

    # project the centered data set onto our kept principal components
    X_projected = X_centered @ self.Vt_subset.T

    return X_projected
    ## End of Your Code ##
```

2.3. **Results.**

These were our results.

```
# fit PCA
pca = PCA(27)
pca.fit(X_train)

# plot the explained variance ratio
S = pca.S
plt.plot(S.cumsum() / S.sum())

# report the truncated feature ratio
explained_var_ratio = (S.cumsum() / S.sum())[pca.n_components]
print(f'Explained variance ratio with {pca.n_components} components:
{explained_var_ratio:.4f}')
```

```
Explained variance ratio with 27 components: 0.3143
```

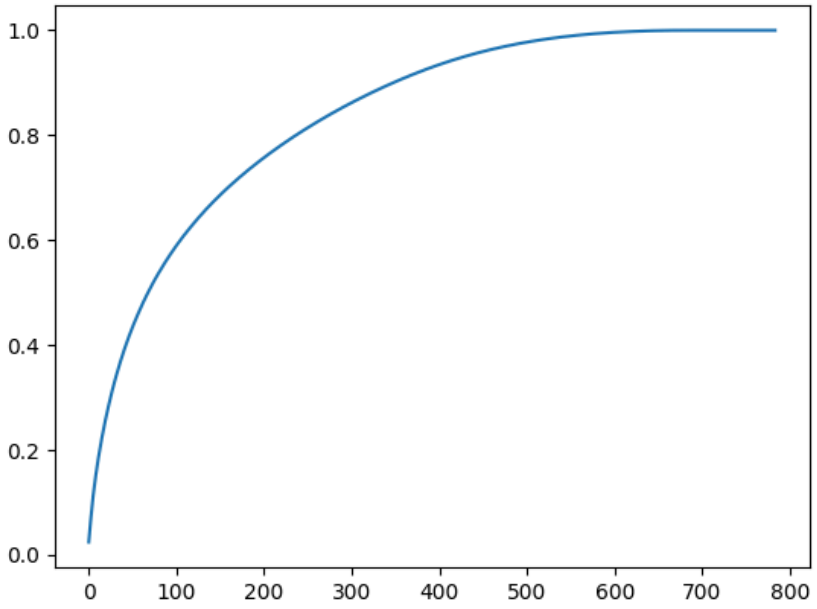

FIGURE 2.

```
X_train_subset = pca.predict(X_train)
X_test_subset = pca.predict(X_test)

classifier = LinearClassifier(n_features=27)
classifier.fit(X_train_subset, y_train)
```

```
iteration 0: loss 23.0139, accuracy 0.1239
iteration 50: loss 0.6564, accuracy 0.8636
iteration 100: loss 0.4564, accuracy 0.8783
iteration 150: loss 0.4186, accuracy 0.8792
Final iteration 199: loss 0.4140, accuracy 0.8790
```

Evaluating our test accuracy with PCA showed an accuracy of 87.90%.

```
preds = classifier.predict(X_test_subset)
test_acc = (preds == y_test).mean()
print(f'Test accuracy: {test_acc:.4f}')
```

```
Test accuracy: 0.8790
```

And this is our visualization.

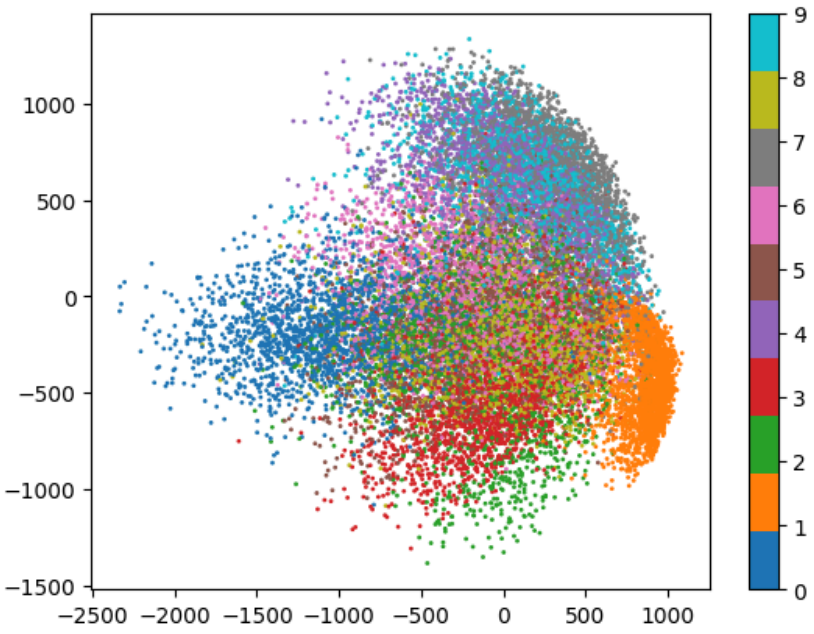

FIGURE 3.

Notice how similar digits have similar features. For example, notice how the clusters for 4 and 9 have a lot of overlap. The same is true for 3 and 8. On the flipside, 1 is quite unique!

3. TASK 3C: NEURAL NETWORK (40PT + 10 BONUS PT)

3.1. **Task 3C-I: Forward and Backward Propogation (20pt).**

3.1.1. *Foward Propogation:* `forward`.

The `forward` function is responsible for computing the output of a neural network layer for a given input. It takes x_l as input and outputs x_{l+1} which is given by $x_{l+1} = w^T x_l + b_l$. These are the operations performed:

1. Linear Transformation: Each input x_l is transformed linearly using the weight matrix w and bias vector b, resulting in an intermediate score for each neuron in the layer.
2. Non-linear Activation: The linear scores are then passed through an activation function to introduce non-linearity into the model. This step allows the neural network to capture complex patterns in the data. The choice of activation function can vary (e.g., ReLU, sigmoid, tanh) and significantly affects the network's learning capabilities.
3. Caching for Backpropagation: The function caches the inputs and activations, which will later be used in the backward pass for computing gradients.

```
def forward(self, inputs):
      """Forward propagate the activation through the layer.
```

```
        Given the inputs (activation of previous layers),
        compute and save the activation of current layer,
        then return it as output.
        """

      ###################################Description begins#################
        # Forward pass

        # Use the linear and non-linear transformation to
        # compute the activation and cache it in a the field, self.a.

        # Functions you may use:
        # np.dot: numpy function to compute dot product of two matrix.
        # self.activate: the activation function of this layer,
       #              it takes in a matrix of scores (linear transformation)
      #             and compute the activations (non-linear transformation).
        # (plus the common arithmetic functions).

        # For all the numpy functions, use google and numpy manual for
        # more details and examples.

        # Object fields you will use:
        # self.w:
        #     weight matrix, a matrix with shape (H_-1, H).
        #     H_-1 is the number of hidden units in previous layer
        #     H is the number of hidden units in this layer
        # self.b: bias, a matrix/vector with shape (1, H).
        # self.activate: the activation function of this layer.

        # Input:
        # inputs:
        #    a matrix with shape (N, H_-1),
        #    N is the number of data points.
        #    H_-1 is the number of hidden units in previous layer

                              ###################################Description
ends####################
        # Modify the right hand side of the following code.

        # The linear transformation.
        # scores:
        #     weighted sum of inputs plus bias, a matrix of shape (N, H).
        #     N is the number of data points.
        #     H is the number of hidden units in this layer.

        scores = np.dot(inputs, self.w) + self.b
        # End of your code


        # The non-linear transformation.
        # outputs:
        #     activations of this layer, a matrix of shape (N, H).
        #     N is the number of data points.
```

```
#     H is the number of hidden units in this layer.

activations = self.activate(scores)
# End of your code

# End of the code to modify
##########################################################

# Cache the inputs and the activations (to be used by backprop).

self.inputs = inputs
self.a = activations
outputs = activations
return outputs
```

3.1.2. *Backward Propogation:* `backward`.

The `backward` function computes the gradients of the loss function with respect to the layer's parameters (weights and biases) and the inputs. This function performs the following steps:

1. Gradient of Activation Function: It first calculates the gradient of the activation function using the cached activations from the forward pass.
2. Gradients w.r.t. Weights and Biases: Using the chain rule, the function computes the gradient of the loss with respect to the weights and biases. The mathematical expressions for these gradients are:

$$\frac{\partial \ell(x,y)}{\partial w} = \text{inputs}^T \cdot \text{d\_scores} \qquad \frac{\partial \ell(x,y)}{\partial b} = \sum \text{d\_scores}$$

3. Gradient w.r.t. Inputs: It also computes the gradient of the loss with respect to the layer's inputs, which is passed backward to the previous layer. This gradient is given by:

$$\frac{\partial \ell(x,y)}{\partial x_l} = \text{d\_scores} \cdot w^T$$

4. Gradient Averaging: The gradients with respect to the weights and biases are averaged over all data points to minimize the average loss, preparing them for the gradient descent update step.

The backward function is vital for learning the parameters of the network. It backpropagates the error from the output layer towards the input layer, allowing the network to adjust its parameters to minimize the loss function.

After completing this entire task successfully we were able to define the MLP class that was provided.

3.2. **Task 3C-II: Training the Neural Network (10pt).**

The objective of this task was to train a neural network using a Multilayer Perceptron (MLP) for the classification of handwritten digits from the MNIST dataset. The network architecture consisted of an input layer, one hidden layer with 16 neurons, and an output layer, with the sigmoid function as the activation for the hidden layer and a linear output layer followed by a softmax and cross-entropy loss function.

```
model = MLP(784, 10, [16], [sigmoid])
```

The MLP was trained over 25 epochs with a batch size of 100 and an initial learning rate of 0.01 using full-batch gradient descent optimization. The training process was monitored by tracking the loss and accuracy at each epoch:

- Training Loss: The loss decreased significantly from the initial epoch, indicating that the model was effectively learning from the data. The decreasing trend in the training loss plot confirms that the optimizer was successfully minimizing the cross-entropy loss over time.
- Training Accuracy: Correspondingly, the accuracy increased, demonstrating the model's improved performance on the training data as it learned to classify digits more accurately. The training accuracy plot shows a saturating curve, which is typical as the model begins to converge to a solution.

After training, the MLP was evaluated on a held-out test set to assess its generalization performance:

- Test Loss: The model achieved a test loss of approximately 0.816, which is higher than the final training loss, as expected due to the model being more tailored to the training data.
- Test Accuracy: The test accuracy reached around 81.2%, which is a substantial improvement over the baseline accuracy of approximately 57% achieved by the simple linear classifier. This improvement validates the efficacy of using a more complex neural network architecture for this classification task.

```
Start training
====================Epoch 0====================
Train loss 2.283327048384687 accuracy 0.25423809523809526
====================Epoch 1====================
Train loss 2.188262769006617 accuracy 0.4472857142857143
====================Epoch 2====================
Train loss 2.108019813330819 accuracy 0.5419047619047619
====================Epoch 3====================
Train loss 2.0230473512232843 accuracy 0.5803333333333334
====================Epoch 4====================
Train loss 1.932098147256449 accuracy 0.6095238095238096
====================Epoch 5====================
Train loss 1.8377437229333573 accuracy 0.6349523809523809
====================Epoch 6====================
Train loss 1.7432347052917012 accuracy 0.6559047619047619
====================Epoch 7====================
Train loss 1.651359626960199 accuracy 0.6715714285714286
====================Epoch 8====================
Train loss 1.5640679185794755 accuracy 0.6851904761904762
====================Epoch 9====================
```

```
Train loss 1.4824822449979043 accuracy 0.6966190476190476
====================Epoch 10====================
Train loss 1.4070769251878594 accuracy 0.7066666666666667
====================Epoch 11====================
Train loss 1.33788518411369 accuracy 0.7148095238095238
...
====================Training finished====================

Test loss 0.8161564260207178 accuracy 0.8121836734693878
```

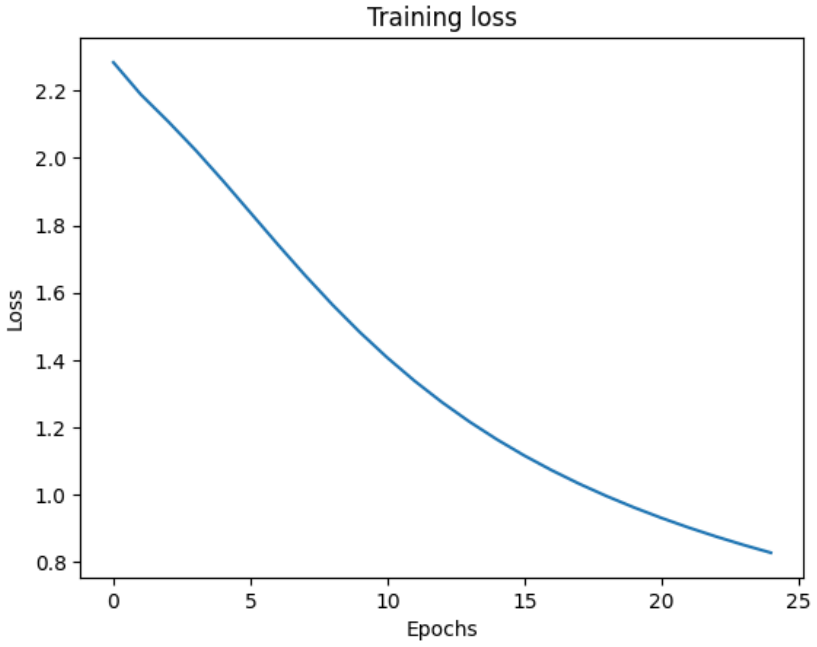

FIGURE 4. Training Loss

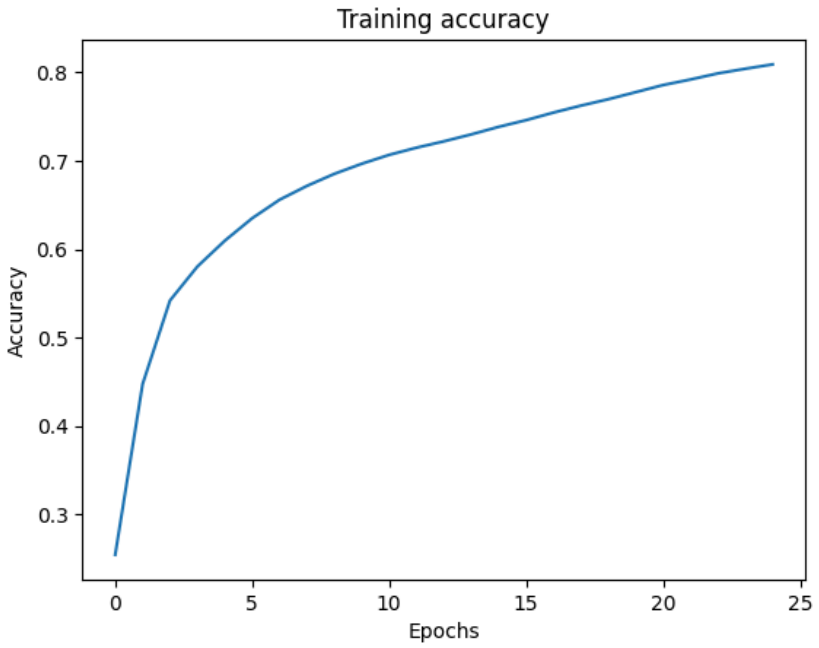

FIGURE 5. Training Accuracy

3.3. **Task 3C-III: Visualizing Predictions (5pt).**

This is a plot of the test samples as per the code supplied in task 3C-III.

Here's how to interpret the information in the plot:

1. Ground Truth Labels: The black numbers on the top left of each image are the true labels, representing the actual digit depicted in the image.
2. Predictions Before Training (Initial Predictions): The red numbers on the bottom right are the predictions made by the MLP model before it was trained. These predictions are likely to be inaccurate, as the model's weights are initially set randomly and have not yet been adjusted to fit the data.
3. Predictions After Training: The blue numbers on the bottom left are the predictions made by the model after it has been trained. The training process adjusts the model's weights through forward and backward propagation, improving its ability to correctly classify the images.

FIGURE 6.

3.4. **Task 3C-IV: Visualizing the Weights (5pt).**

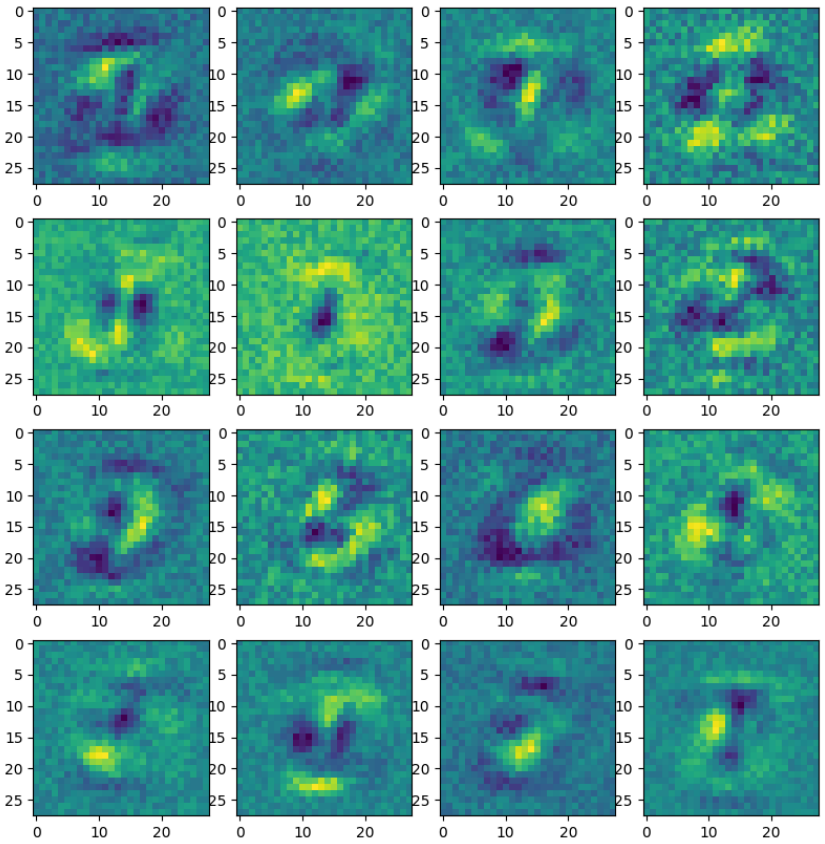

FIGURE 7.

The plot provided visualizes the weights of the first layer in the Multilayer Perceptron (MLP) that was trained on the MNIST dataset. Each subplot corresponds to one neuron's weights in the hidden layer, reshaped and visualized as a 28x28 image, which is the same shape as the input images.

```
fig, axes = plt.subplots(4, 4, figsize=(10, 10))
fig.subplots_adjust(wspace=0)
idx = 0
for a in axes.flatten():
    a.imshow(model.layers[0].w[:, idx].reshape((28, 28)))
    idx +=1
plt.show()
```

(Wasn't sure which detailed explanation was being asked for. Here is the heatmap explanation. The MLP explanation is below.) The code block is designed to iterate over the weights associated with each neuron in the first hidden layer of the MLP and display them in a grid format:

- `fig, axes = plt.subplots(4, 4, figsize=(10, 10))` creates a figure and a 4x4 grid of subplots, as there are 16 neurons in the hidden layer.

- `fig.subplots_adjust(wspace=0)` adjusts the space between the subplots to remove any whitespace for a cleaner display.
- The `for` loop iterates over the flattened axes and the corresponding weights of each neuron in the first hidden layer.
- `a.imshow(model.layers[0].w[:, idx].reshape((28, 28)))` takes the weights of the `idx`-th neuron, reshapes them into a 28x28 array to match the input image dimensions, and displays them using `imshow`. This reshaping is logical since each weight corresponds to a pixel in the input images.
- `idx +=1` increments the index to move to the next neuron's weights for each iteration of the loop.

In the visualizations, each image represents the weights as learned features from the input data that are used to activate the respective neuron in the hidden layer. The patterns observed in these visualizations can often be interpreted as the features that the network 'sees' as important for making decisions. For example:

- Bright Spots: Bright areas in the weight visualizations indicate higher weight values, suggesting that these pixels play a more significant role in the neuron's activation when those pixels are present in the input image.
- Dark Spots: Conversely, darker areas correspond to lower weights and suggest that these pixels have less influence on the neuron's activation.
- Patterns Resembling Digits: Some weight visualizations vaguely resemble parts or strokes of digits!
- Contrasting Regions: Areas of contrasting colors (e.g., bright spots next to dark spots) can indicate that the neuron is sensitive to edges or changes in pixel intensity, which are important for distinguishing between different shapes and thus different digits.

Detailed Explanation of MLP Code

(Wasn't sure how thorough to be. Sorry if this is too much!)

`MLP Class`

- `__init__(self, input_dim, output_dim, sizes, activ_funcs)`: This method initializes the MLP with the specified architecture. It takes the dimensions of the input and output layers, a list of the number of neurons in each hidden layer (`sizes`), and a list of activation functions for each layer (`activ_funcs`). It then constructs a list of `Layer` objects, each corresponding to a layer in the network, and an additional output layer with a linear activation function.
- `forwardprop(self, data, labels=None)`: This method performs forward propagation through the network. It processes the input data through each layer, applying the corresponding activation function, and computes the output probabilities using the softmax function. If labels are provided, it also computes the loss using the `self.loss` method.
- `backprop(self, labels)`: This method performs backward propagation, computing gradients of the loss with respect to the network parameters. It uses these gradients to update the weights and biases of each layer in the network.

- `loss(self, outputs, labels)`: This function computes the cross-entropy loss between the predicted outputs and true labels.
- `d_loss(self, outputs, labels)`: This function computes the gradient of the cross-entropy loss with respect to the outputs of the network.
- `predict(self, data)`: This method predicts labels for the input data by performing forward propagation and returning the class with the highest probability.

`GradientDescentOptimizer Class`

- `__init__(self, learning_rate, decay_steps=1000, decay_rate=1.0)`: Initializes the optimizer with a given learning rate and parameters for learning rate decay.
- `update(self, model)`: Applies gradient descent to update the model parameters based on the computed gradients. It also handles learning rate decay after a certain number of steps.

`Utility Functions`

- `Activation functions (sigmoid, relu, tanh, linear)`: These functions apply the corresponding activation to the input data.
- `Derivative functions (d_sigmoid, d_relu, d_tanh, d_linear)`: These functions compute the derivative of the activation functions, which are used in backpropagation.
- `softmax`: This function applies the softmax transformation to the logits to obtain class probabilities.
- `mean_cross_entropy`: Computes the cross-entropy loss given the outputs and labels.
- `mean_cross_entropy_softmax`: Combines the softmax transformation and the computation of the cross-entropy loss into a single step.
- `d_mean_cross_entropy_softmax`: Computes the gradient of the loss obtained from mean_cross_entropy_softmax.

`GRAD_DICT`

A dictionary mapping each activation function to its corresponding derivative function. This is used within the `Layer` class to apply the correct derivative during backpropagation.

3.5. **(Bonus 10pt) Task 3C-V: Hyperparameter tuning.**

To improve our test accuracy, we used the following model:

```
model = MLP(784, 10, [128, 64], [relu, relu])
```

and achieved these results:

```
Initializing neural network
Start training
====================Epoch 0====================
Train loss 2.1410277898460346 accuracy 0.6403333333333333
====================Epoch 1====================
Train loss 1.5210985167712776 accuracy 0.7783333333333333
```

```
====================Epoch 2====================
Train loss 0.8986019743127578 accuracy 0.8391904761904762
====================Epoch 3====================
Train loss 0.6347175882744549 accuracy 0.8606666666666667
====================Epoch 4====================
Train loss 0.5207361098703692 accuracy 0.8741904761904762
====================Epoch 5====================
Train loss 0.459036872294058 accuracy 0.8836666666666667
====================Epoch 6====================
Train loss 0.42018951850072034 accuracy 0.8901428571428571
====================Epoch 7====================
Train loss 0.393047995907568 accuracy 0.8949047619047619
====================Epoch 8====================
Train loss 0.3726454069837369 accuracy 0.9003333333333333
====================Epoch 9====================
Train loss 0.3564837495995607 accuracy 0.9036190476190477
====================Epoch 10====================
Train loss 0.3431751743168801 accuracy 0.9064285714285715
====================Epoch 11====================
...
====================Training finished====================

Test loss 0.27274906453001063 accuracy 0.9225918367346939
```

Here is a breakdown of what we changed and some other considerations:

1. **Activation Function: From Sigmoid to ReLU**

Switching from the `sigmoid` function to the `ReLU` (Rectified Linear Unit) activation function is a crucial change. The `sigmoid` function $f(x) = \frac{1}{1+e^{-x}}$ suffers from the vanishing gradient problem, where for very high or very low values of xx, the gradient of the function becomes very small, slowing down the learning process during backpropagation. This problem is more pronounced in deep networks with many layers.

On the other hand, the `ReLU` function, defined as $f(x) = \max(0, x)$, has a derivative of 1 for all positive inputs and 0 otherwise. This characteristic helps mitigate the vanishing gradient problem, allowing for faster learning and convergence, especially in deep networks. The choice of `ReLU` is significant because it enables the network to learn complex patterns more efficiently compared to `sigmoid`.

2. **Hidden Layers and Units**

The architecture chosen, with two hidden layers of 128 and 64 units, respectively, is designed to increase the model's capacity to learn from the high-dimensional MNIST dataset. The first layer with 128 units can capture a broad range of features from the input data, while the second layer with 64 units focuses on refining these features into more abstract representations that can be used for classification. The number of units in these layers is a trade-off between the model's ability to learn (more units can capture more complex patterns) and the computational cost and risk of overfitting (more units can lead to a model that is too tailored to the training data and performs poorly on unseen data).

3. **Overfitting Considerations**

Overfitting is a critical concern when increasing the complexity of a model. It occurs when a model learns the noise in the training data to the extent that it negatively impacts the model's performance on new data. Monitoring the test loss and accuracy is vital to identify overfitting. In this case, the test accuracy improved to approximately 92.3%, indicating that the model generalizes well to unseen data. However, it's essential to continue evaluating the model on a separate validation set during training and potentially employ regularization techniques if signs of overfitting appear.

4. **Computational Cost**

The training time increased by more than double due to the new more complex architecture. More hidden units and layers mean more parameters to update during backpropagation, which increases the computational cost. This trade-off is often worthwhile when it leads to better model performance, as seen in the improved test accuracy. Nonetheless, it's essential to balance model complexity with computational resources and training time, especially for larger datasets or more complex tasks. And it was still done in just a few minutes!

In summary: The changes to the neural network architecture and activation functions were carefully chosen to improve the model's ability to learn from the MNIST dataset effectively. The use of ReLU activation functions and the increase in hidden units and layers were aimed at enhancing the model's capacity while managing the risk of overfitting. The improvement in test accuracy confirms the effectiveness of these changes, though the increase in computational cost is a necessary trade-off. The results indicate that the model's complexity and training duration were acceptable for achieving higher accuracy, demonstrating a successful application of hyperparameter tuning in neural network design.