Member-only story

Classifying Handwritten Digits Using A Multilayer Perceptron Classifier (MLP)

What is a Multilayer Perceptron? What are the pros and cons of MLP? Can we classify handwritten digits accurately using a MLP classifier? How do learnt weights look like?

Serafeim Loukas, PhD · Follow

Published in Towards Data Science
7 min read · Nov 27, 2021

Listen Share More

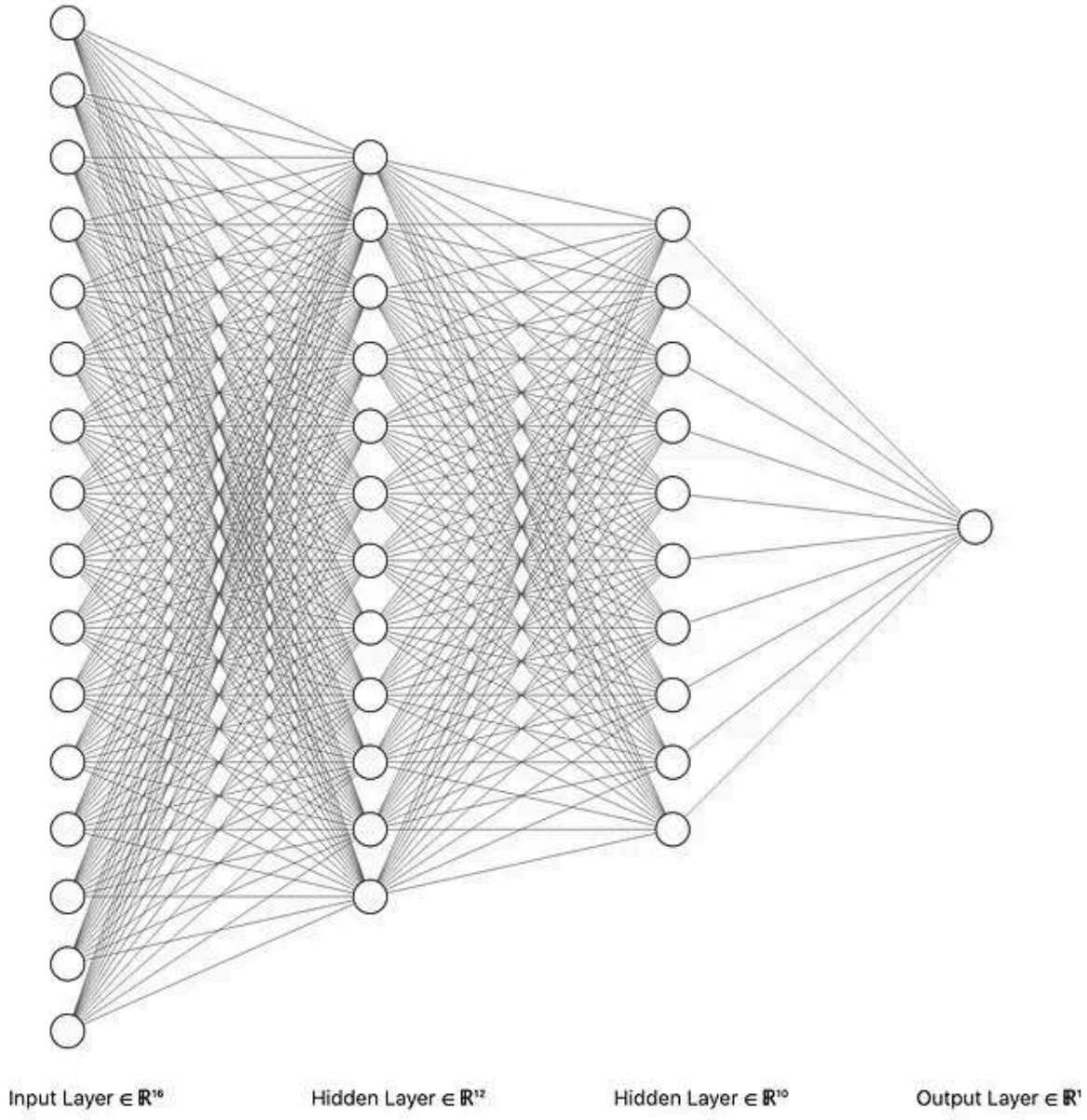

Figure 1: A Multilayer Perceptron Network (source).

1. Short Introduction

1.1 What is a Multilayer Perceptron (MLP)?

An **MLP** is a **supervised** machine learning (ML) algorithm that belongs in the class of feedforward artificial neural networks [1]. The algorithm essentially is trained on the data in order to learn a function. Given a set of features and a target variable (e.g. labels) it learns a non-linear function for either classification or regression. In this article, we will only focus on the classification case.

1.2 Is there any similarity between the MLP and the Logistic Regression?

There is! The logistic regression has only two layers i.e. the **input** and **output** however, in the case of a MLP model, the only difference is that we can have additional intermediate **nonlinear** layers. These are known as **hidden layers**. Except for the input nodes (nodes that belong to the input layer), each node is a neuron that

uses a **nonlinear activation function** [1]. Due to this non-linearity nature, the MLP can learn complex nonlinear functions and thus, distinguish data that are not linearly separable! See **Figure 2** below for a visual representation of an MLP classifier with one hidden layer.

> *NEW: After a great deal of hard work and staying behind the scenes for quite a while, we're excited to now offer our expertise through a platform, the "**Data Science Hub**" on Patreon (https://www.patreon.com/TheDataScienceHub). This hub is our way of providing you with **bespoke consulting services** and comprehensive **responses to all your inquiries**, ranging from Machine Learning to strategic data analytics planning.*

1.3 How is an MLP trained?

MLP uses **backpropagation** for training [1]. You can have a look at this website here for the formal mathematical formulation.

1.4 Main Advantages and Disadvantages of MLP

Pros

- Can learn **nonlinear functions** and thus **separate not linearly separable data** [2].

Cons

- The loss function for the hidden layer leads to a **non-convex** optimization problem and thus, local minimum exist.
- Due to the above issue, different weight **initializations** may lead to different outputs/weights/results.
- The MLP has some **hyperparameters** e.g. the number of hidden neurons, layers that need to be tuned (time & power consuming) [2].
- MLP can be **sensitive** to feature **scaling** [2].

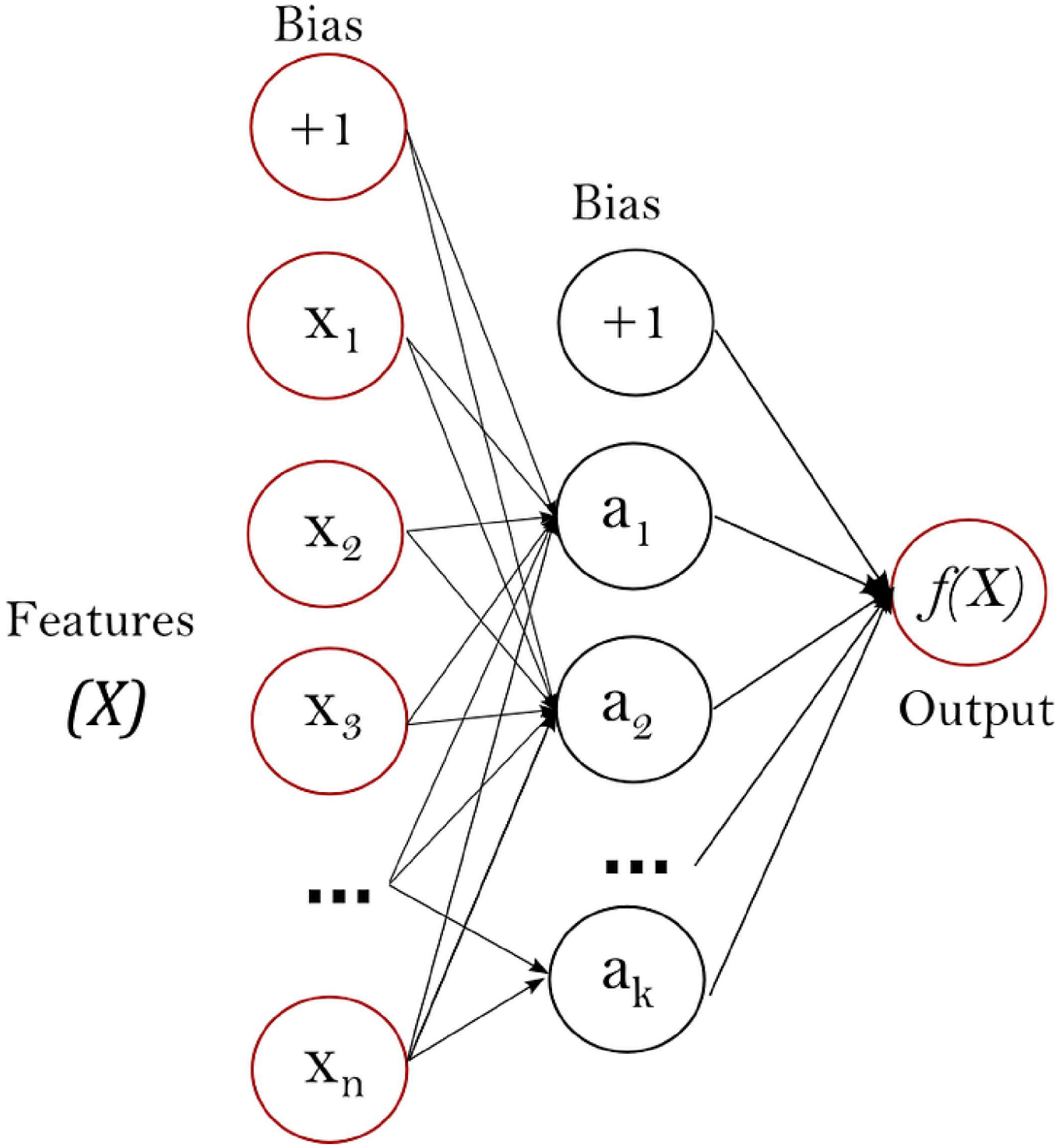

Figure 2: A MLP with one hidden layer and with a scalar output. Image adapted from scikit-learn python documentation.

2. Python hands-on example using scikit-learn

2.1 The dataset

For this hands-on example we will use the MNIST dataset. The MNIST database is famous database of handwritten digits that is used for training several ML models [5]. There are handwritten images of 10 different digits so the **number of classes** is `10` (see **Figure 3**).

Note: Since we deal with **images**, these are represented by **2D arrays** and the initial dimension of the data is `28 by 28` for each image (`28x28 pixels`). The 2D images are then **flattened** and thus, represented by vectors at the end. **Each 2D image** is transformed into a 1D **vector** with dimension `[1, 28x28] = [1, 784]` . Finally, our dataset has ***784 features/variables/columns.***

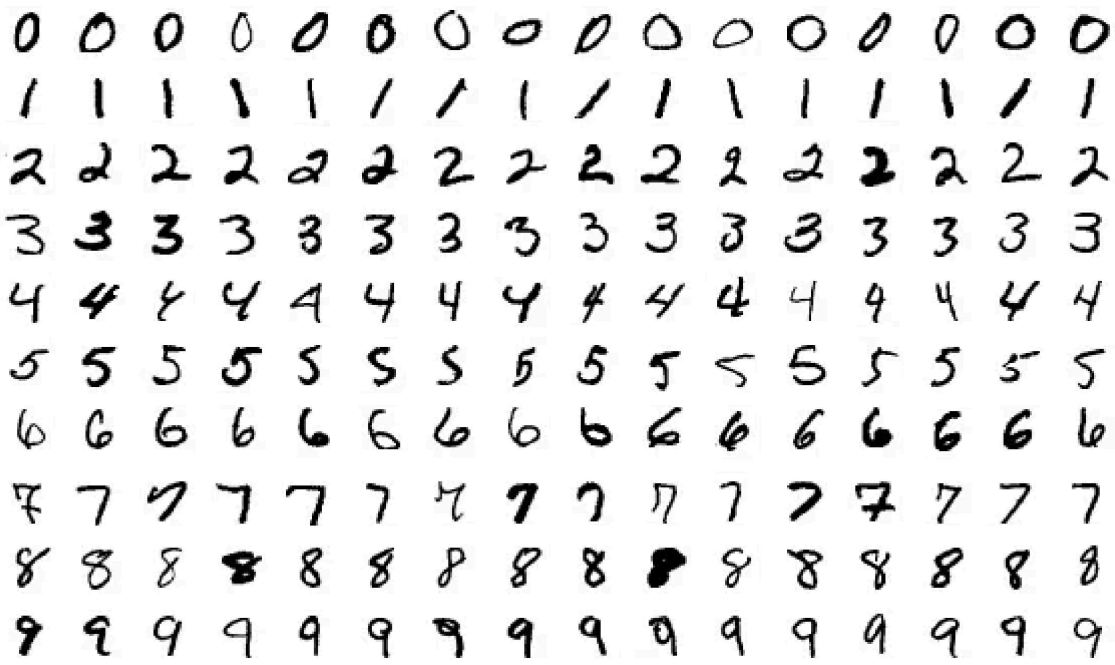

Figure 3: Some samples from the dataset (Source).

2.2 Data import and preparation

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier

# Load data
X, y = fetch_openml("mnist_784", version=1, return_X_y=True)

# Normalize intensity of images to make it in the range [0,1] since
255 is the max (white).
X = X / 255.0
```

Remember that **each 2D image** is now transformed into a **1D vector** with dimension `[1, 28x28] = [1, 784]` . Let's verify this now.

```
print(X.shape)
```

This returns: `(70000, 784)`. We have **70k flattend images** (samples), each containing 784 pixels (28*28=784) (variables/features). Thus, the **input layer weight matrix** will have shape `784 x #neurons_in_1st_hidden_layer.` The **output layer weight matrix will have shape** `#neurons_in_3rd_hidden_layer x #number_of_classes.`

2.3 Model training

Now let's build the model, train it and perform the classification. We will use `3` hidden layers with `50,20 and 10` neurons each, respectively. Also, we will set the max iterations to `100` and the learning rate to `0.1`. These are the **hyperparameters** that I mentioned in the Introduction. We will not fine-tune them here.

```
# Split the data into train/test sets
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

classifier = MLPClassifier(
    hidden_layer_sizes=(50,20,10),
    max_iter=100,
    alpha=1e-4,
    solver="sgd",
    verbose=10,
    random_state=1,
    learning_rate_init=0.1,
)

# fit the model on the training data
classifier.fit(X_train, y_train)
```

2.4 Model evaluation

Now, let's evaluate the model. We will estimate the mean accuracy on the training and test data and labels.

```
print("Training set score: %f" % classifier.score(X_train, y_train))
print("Test set score: %f" % classifier.score(X_test, y_test))
```

> *Training set score:* `0.998633`
>
> *Test set score:* `0.970300`

Great results!

2.5 Visualizing the Cost Function evolution

How fast was the loss decreased during training? Let's make a nice plot!

```
fig, axes = plt.subplots(1, 1)

axes.plot(classifier.loss_curve_, 'o-')
axes.set_xlabel("number of iteration")
axes.set_ylabel("loss")
plt.show()
```

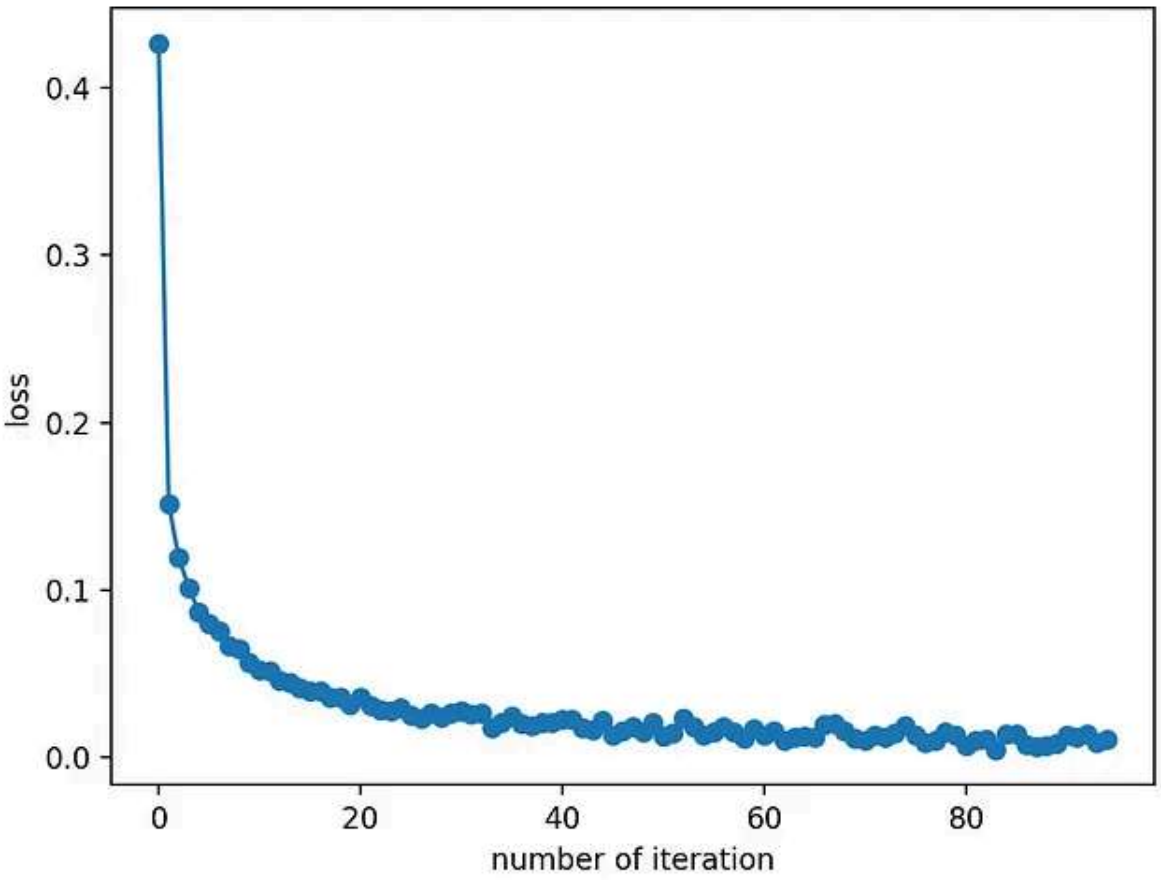

Figure 4: The evolution of the loss across the training iterations. Figure made by the author.

Here we see that the loss decreases really fast during training and it saturates after the `40th` iteration (remember that we defined max 100 iterations as **hyperparameter**).

2.6 Visualizing the learnt weights

Here we need first to understand how the weights (learnt model parameters of each layer) are stored.

Based on the documentation, the attribute `classifier.coefs_` is a list of shape `(n_layers-1, )` of weight arrays, where weight matrix at index i represents the weights between layer `i` and layer `i+1`. In this example, we defined 3 **hidden layers**

and we also have the **input** and **output layers.** Thus, we expect to have 4 weight arrays for the between-layers weights (`in-L1`, `L1-L2`, `L2-L3` and `L2-out` in **Figure 5**).

Similarly, `classifier.intercepts_` is a list of **bias** vectors, where the vector at index `i` represents the bias values added to layer `i+1` .

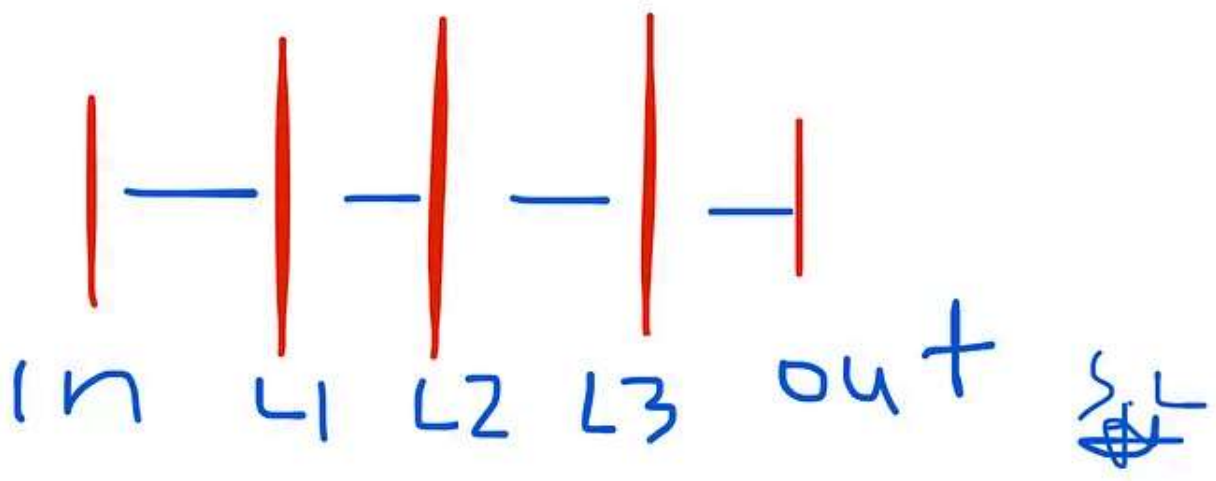

Figure 5: Handmade figure by the author on Notes (iOS).

Let's verify this:

```
len(classifier.intercepts_) == len(classifier.coefs_) == 4
```

correctly returns `True` .

Reminder: The **input layer weight matrix** will have shape `784 x #neurons_in_1st_hidden_layer`. The **output layer weight matrix will have shape** `#neurons_in_3rd_hidden_layer x #number_of_classes`.

Visualizing the learnt weights of the input layer

```
target_layer = 0 #0 is input, 1 is 1st hidden etc

fig, axes = plt.subplots(1, 1, figsize=(15,6))
axes.imshow(np.transpose(classifier.coefs_[target_layer]),
cmap=plt.get_cmap("gray"), aspect="auto")

axes.set_xlabel(f"number of neurons in {target_layer}")
axes.set_ylabel("neurons in output layer")
```

```
plt.show()
```

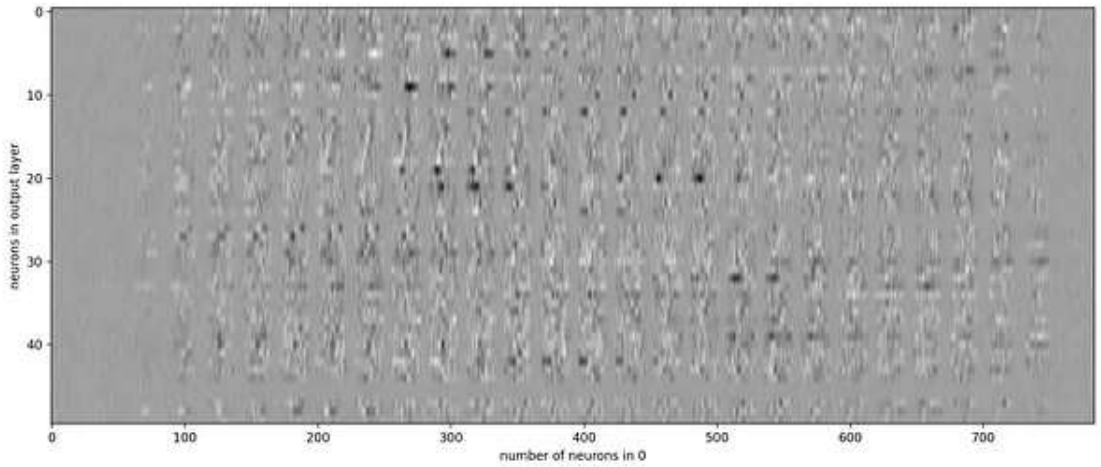

Figure 6: Visualization of the learnt weights of the neurons between the input and the 1st hidden layer. Figure made by the author.

Or reshape and plot them as 2D images again.

```
# choose layer to plot
target_layer = 0 #0 is input, 1 is 1st hidden etc

fig, axes = plt.subplots(4, 4)
vmin, vmax = classifier.coefs_[0].min(),
classifier.coefs_[target_layer].max()

for coef, ax in zip(classifier.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=0.5 *
vmin, vmax=0.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())
plt.show()
```

3. Summary

MLP classifier is a very powerful neural network model that enables the learning of non-linear functions for complex data. The method uses forward propagation to build the weights and then it computes the loss. Next, back propagation is used to update the weights so that the loss is reduced. This is done in an iterative way and the **number of iterations** is an input **hyperparameter** as I explained in the Introduction. Other important **hyperparameters** are the **number of neurons** in each hidden layer and the **number of hidden layers** in total. These need to be fine-tuned.