Lab 2 - Computer Vision

Diego Torres

November 2023

1 Introduction

In this lab exercise we will implement several end-to-end methods to perform image classification. In early ages of Computer Vision, handcrafted features based on gradients were the norm. When more computation power was available, neural networks were a suitable option that proved to be better than previous approaches. The chosen models are the Linear Classifier and the Multi Layer Perceptron (MLP). I will discuss the theory behind these models, explain details about my particular implementation, and explore the performance in benchmark datasets. As this is a computer vision course, the MNIST dataset will be used to check the performance of the proposed methods.

2 Linear Classifier

The linear classifier tries to learn a decision boundary between two classes that is linear. This means that the choice to classify one sample as the positive or negative class will be determined by the evaluation of a linear function given by:

$$f(x) = \beta^T x + b > 0, \tag{1}$$

for a weight vector β and bias parameter b.
The way to train this model is by means of gradient descent on a loss that is suitable to the classification problem. One such loss is the binary cross entropy. In this framework, we pass the output of the linear function through a sigmoid function, which converts the raw value to a probability that can be interpreted as the parameter of a Bernoulli distribution. In that case, the binary cross entropy is equivalent to the negative log-likelihood of the dataset.

$$-\mathcal{L} = \text{BinaryCrossEntropy} = -\sum_{(x,y)\in D} y \log \sigma(f(x)) + (1-y)\log(1-\sigma(f(x))). \tag{2}$$

What remains then is to compute the gradient of this loss with respect to the model parameters β and b. For sake of simplicity, we will ignore b and set it to 0, which will be handled in practice by adding an additional feature of 1 to every sample. With this modification, and using $\sigma(x) = \exp(x)/(1+\exp(x))$, we can compute a simpler expression for the loss of an (x, y) sample pair:

$$-\text{BCE}(x, y) = yf(x) - \log(1 + \exp(f(x))). \tag{3}$$

Taking the derivative with respect to $f(x)$ is straightforward, yielding the result $-y + \sigma(f(x))$. Then, to get the derivative with respect to β we use the chain rule and the fact that $\nabla_\beta f(x) = x$, which gives:

$$\nabla_\beta \text{BCE}(x, y) = x\,(\sigma(f(x)) - y). \tag{4}$$

With this in mind, the algorithm for training the linear classifier reads as follows:

1. Initialize β with random values
2. Compute the gradient

```
class LinearClassifier:

    def __init__(self, input_size) -> None:
        self.weights = np.random.randn(input_size, 1)*0.1

    def forward(self, x):
        return np.dot(x, self.weights).reshape(-1, 1)

    def predict(self, x):
        return np.where(self.forward(x) > 0, 1, 0)

    def train(self, x, y, n_iter=1000, lr=0.01):
        for _ in range(n_iter):
            self.train_step(x, y, lr)

        return self

    def train_step(self, x, y, lr):
        # size of x: (N, D)
        # size of y: (N, 1)
        # size of w: (D, 1)

        diff = -y.reshape(-1, 1) + self.sigmoid(self.forward(x)) # (N, 1)
        grad = x.T@ diff

        self.weights -= lr*grad

    def score(self, x, y):
        preds = self.predict(x).flatten()
        return np.mean(preds==y)

    @staticmethod
    def sigmoid(x):
        return 1/(1+np.exp(-x))
```

Code 1: My implementation of the Linear Classifier

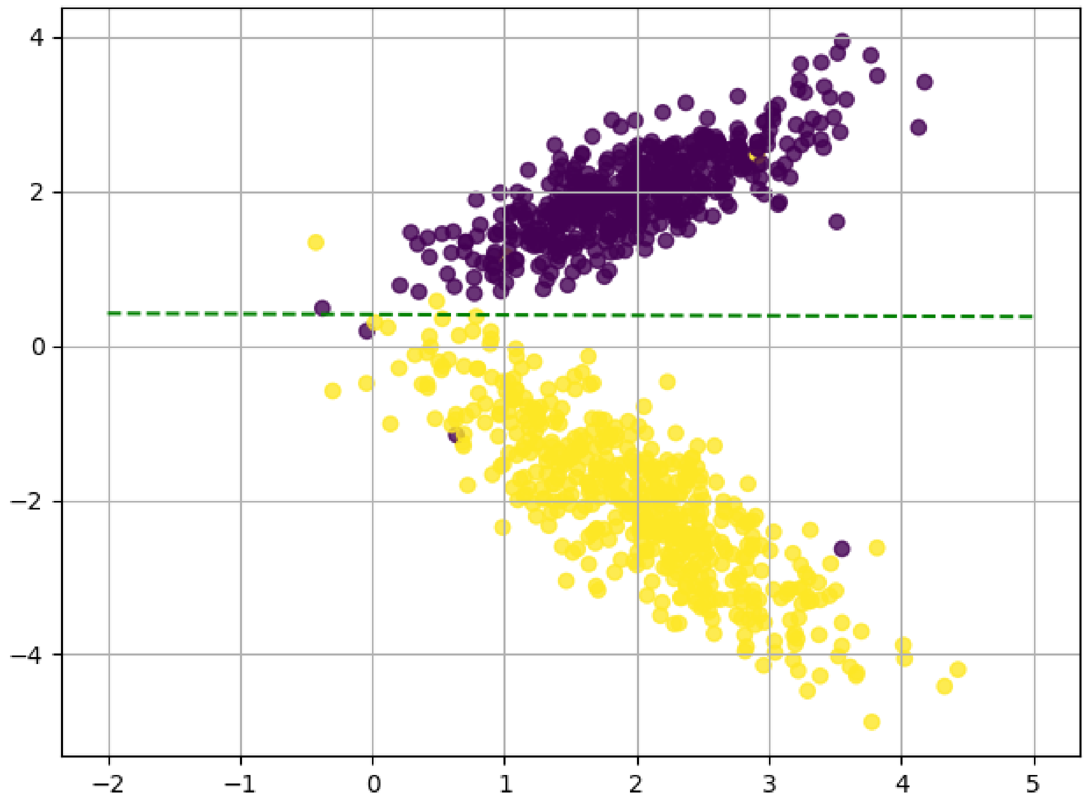

Figure 1: Toy dataset created with sklearn, the dotted line represents the decision boundary learned by the model.

3. Update $\beta \leftarrow \beta - \eta \nabla_{\beta} \text{BCE}$
4. Do until convergence or maximum steps are reached

The code with my implementation is given in In order to test the code I use the **make_classification** function of Scikit-learn to create a toy dataset that is separable and with only two features, which is shown in figure 1. As it can be seen, the model is able to learn an effective decision boundary between the two classes, which gives an accuracy of 99.5% and 98.5% on the training and validation datasets, respectively.

3 Multilayer Perceptron (MLP)

In order to build the MLP, I will implement the autograd algorithm for 3 different types of operations: linear layer, activation layer, cross entropy loss. For each one of them, I will need to define the forward method (which computes output given the input), and backward method (which computes the gradient of the input given the gradient of the output).

3.1 Linear Layer

Given an input X (either the features or a previous layer), the linear layer computes the output by taking the matrix multiplication with the weight matrix W:

$$Y = WX. \tag{5}$$

Here we use the convention where each sample is a column in X, so that their shapes are (D, N) and (P, D), and the shape of the output (P, N). The next ingredient that we need is the way to compute the gradients

```python
class LinearLayer:
    def __init__(self, input_size, output_size, eta=1e-3, random_state=None):
        np.random.seed(random_state)
        self.weights = np.random.normal(scale=0.1, size=(output_size, input_size+1))
        self.gradient = np.zeros_like(self.weights)
        self.eta = eta

    def __call__(self, input_layer):
        self.input_layer = input_layer
        self.D = input_layer.values.shape[1]
        self.values = self.weights@ np.vstack([np.ones((1, self.D)), input_layer.values])
        return self

    def backward(self, input_gradient):
        self.gradient = input_gradient@ np.vstack([np.ones((1, self.D)), self.input_layer.values]).T
        gradient_back = (self.weights[:, 1:]).T@ input_gradient
        self.input_layer.backward(gradient_back)

    def step(self):
        self.weights = self.weights - self.eta*self.gradient
        self.gradient = np.zeros_like(self.weights)
        self.input_layer.step()
```

Code 2: LinearLayer class

with respect to W and X, given that with respect to Y. The first one will be used to update the parameters, whereas the other one will be used by the previous layers to update their parameters. Following the chain rule and taking care of the dimensions we find:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial Y} X^T, \quad \frac{\partial \mathcal{L}}{\partial X} = W^T \frac{\partial \mathcal{L}}{\partial Y} \tag{6}$$

The Python code for this layer is given in code 2. Here the code also takes into account the bias term by adding a row of 1's to the input array. The class' backward method takes charge of calling the backward of its input layer as well, providing it with the appropriate gradient. Additionally, the **step** method updates the parameters and resets the gradient.

3.2 Activation Layer

This is one of the simplest layers, given a function of one variable h, it applies it to every element in the input matrix:

$$Y = f(X), \tag{7}$$

with $Y_{ij} = f(X_{ij})$. Notice that for this layer we don't have any parameters, so that the only remaining function we need is backpropagation. Fortunately this is very simple:

$$\frac{\partial \mathcal{L}}{\partial X} = f'(X) * \frac{\partial \mathcal{L}}{\partial Y}, \tag{8}$$

where $*$ denotes the element-wise multiplication and $f'(X)$ denotes the derivative of f as a univariate function applied to every element in X. The code for this layer is provided in code 3

3.3 Loss Layer

This layer takes charge of computing the cross entropy loss given the ground-truth and the model predictions. It also computes the gradient with respect to the predictions, which is crucial for training the network. In a very similar way to what I did with the linear classifier, we can simplify the expression for the loss if we consider the logits directly (instead of the probabilities coming from the softmax). After some manipulation, it's possible to find that

$$\text{CrossEntropy} = -y^T w + \log \sum_i \exp(w_i), \tag{9}$$

```python
class ActivationLayer:
    def __init__(self, activation_function, derivative):
        self.function = activation_function
        self.derivative = derivative

    def __call__(self, input_layer):
        self.input_layer = input_layer
        self.values = self.function(input_layer.values)
        return self

    def backward(self, input_gradient):
        self.gradient = self.derivative(self.input_layer.values)*input_gradient
        self.input_layer.backward(self.gradient)

    def step(self):
        del self.gradient
        self.input_layer.step()
```

Code 3: ActivationLayer class

```python
class LossLayer:
    def __init__(self):
        self.pred_layer = None
        self.gt = None

    def __call__(self, gt, pred_layer):
        probs = softmax(pred_layer.values)
        self.gradient = probs.copy()
        self.gradient[gt, np.arange(gt.shape[0])] -= 1
        self.pred_layer = pred_layer
        return -np.log(probs[gt, np.arange(gt.shape[0])]).mean()

    def backward(self):
        self.pred_layer.backward(self.gradient)

    def step(self):
        self.pred_layer.step()
```

Code 4: LossLayer class

with w representing the logits and y the one-hot vector with the 1 at the true label index. In that case, the gradient also simplifies to a very nice expression:

$$\nabla_w \text{CrossEntropy} = -y + \text{softmax}(w) \tag{10}$$

The code is provided in 4, where we assume that the ground-truth is given by an array of indices, so that we can handle the gradient computation more efficiently.

3.4 Combining Everything

Finally, we combine these layers into a higher level class **MLP**, shown in code 5.

3.5 Iris Dataset

I tried the MLP on the Iris dataset, which is a classic of machine learning. I used 4 layers with 10 neurons each, and trained it for 10k iterations using the **train** method of the class. The algorithm got an accuracy of 96.6% in both the training and validation datasets. The confusion matrices are shown in figure 2.

```
class MLP:
    def __init__(self, input_size, output_size, *, hidden_size, n_layers):
        self.layers = []
        self.layers.append(LinearLayer(input_size, hidden_size))
        self.layers.append(ActivationLayer(ReLU, ReLU_derivative))

        for _ in range(n_layers-1):
            self.layers.append(LinearLayer(hidden_size, hidden_size))
            self.layers.append(ActivationLayer(ReLU, ReLU_derivative))

        self.layers.append(LinearLayer(hidden_size, output_size))
        self.loss_layer = LossLayer()

    def __call__(self, input_values):
        input_x = InputLayer(input_values)
        for layer in self.layers:
            input_x = layer(input_x)
        return input_x

    def train_step(self, x, y):
        logits = self(x)
        loss = self.loss_layer(y, logits)
        self.loss_layer.backward()
        self.loss_layer.step()

        return loss

    def train(self, x, y, *, n_iter=1000, print_every=100):
        for _ in range(n_iter):
            loss = self.train_step(x, y)
            if _ % print_every == 0:
                print(loss)
        return self

    def predict(self, x):
        return self(x).values.argmax(axis=0)

    def score(self, x, y):
        preds = self.predict(x).flatten()
        return np.mean(preds==y)
```

Code 5: MLP class

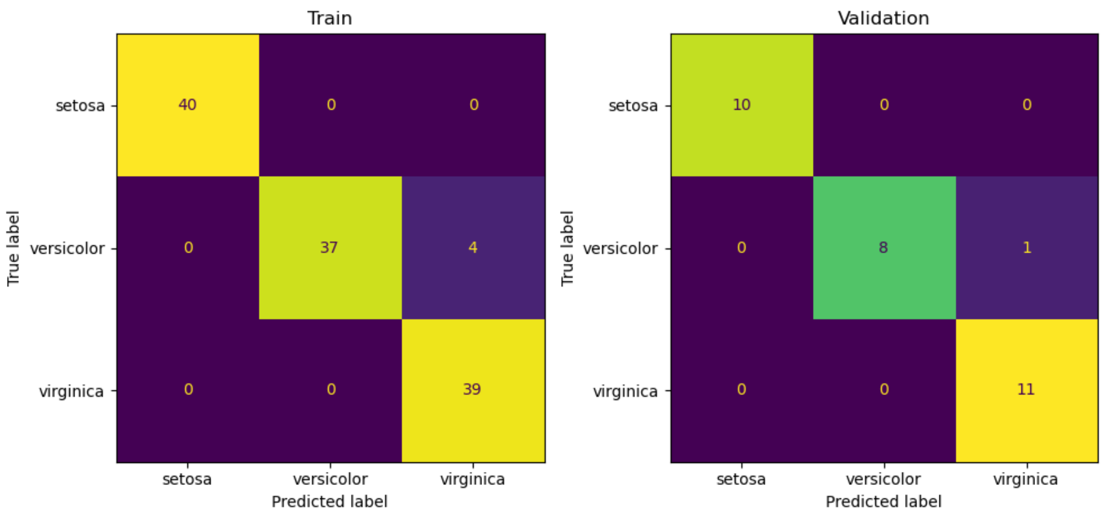

Figure 2: Confusion matrices of the MLP for the Iris dataset

```python
losses = []
train_accs = []
val_accs = []
for epoch in range(10):
    print("Epoch: ", epoch+1)

    # Training
    tot_loss = 0
    for i in range(0, X_train.shape[0], 100):
        loss = model.train_step(X_train[i:i+100].T, y_train[i:i+100])
        tot_loss += loss*len(y_train[i:i+100])/len(y_train)

    # Evaluation
    train_acc = model.score(X_train.T, y_train)
    val_acc = model.score(X_val.T, y_val)

    print("Training loss: ", tot_loss)
    print("Training accuracy: ", train_acc)
    print("Validation accuracy: ", val_acc)

    losses.append(tot_loss)
    train_accs.append(train_acc)
    val_accs.append(val_acc)
```

Code 6: Training loop for MNIST dataset

4 MNIST Dataset

Probably the most widely used image classification dataset, it consists of 70k of 28x28 images of handwritten digits. I used an MLP with 4 layers and 50 neurons per layer. As this dataset is much bigger than the Iris one, I used a training loop, updating the parameters in batches of 100 samples, the code is shown in code 6.

In figures 3 and 4 are shown the confusion matrices and evolution of the loss and accuracy. We can see that this model effectively learns to solve the dataset.

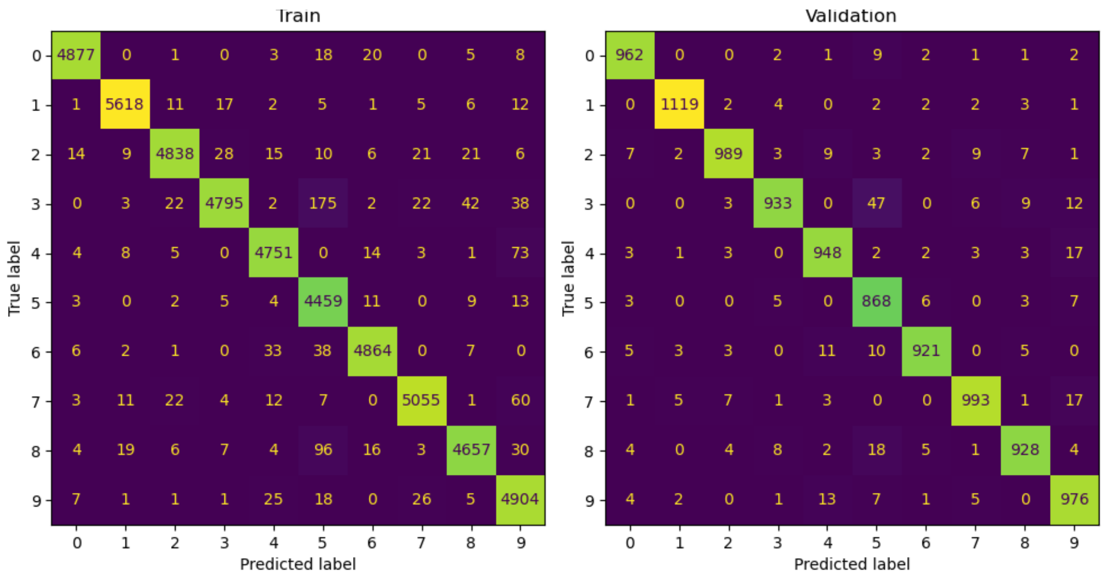

Figure 3: Confusion matrices on the MNIST dataset

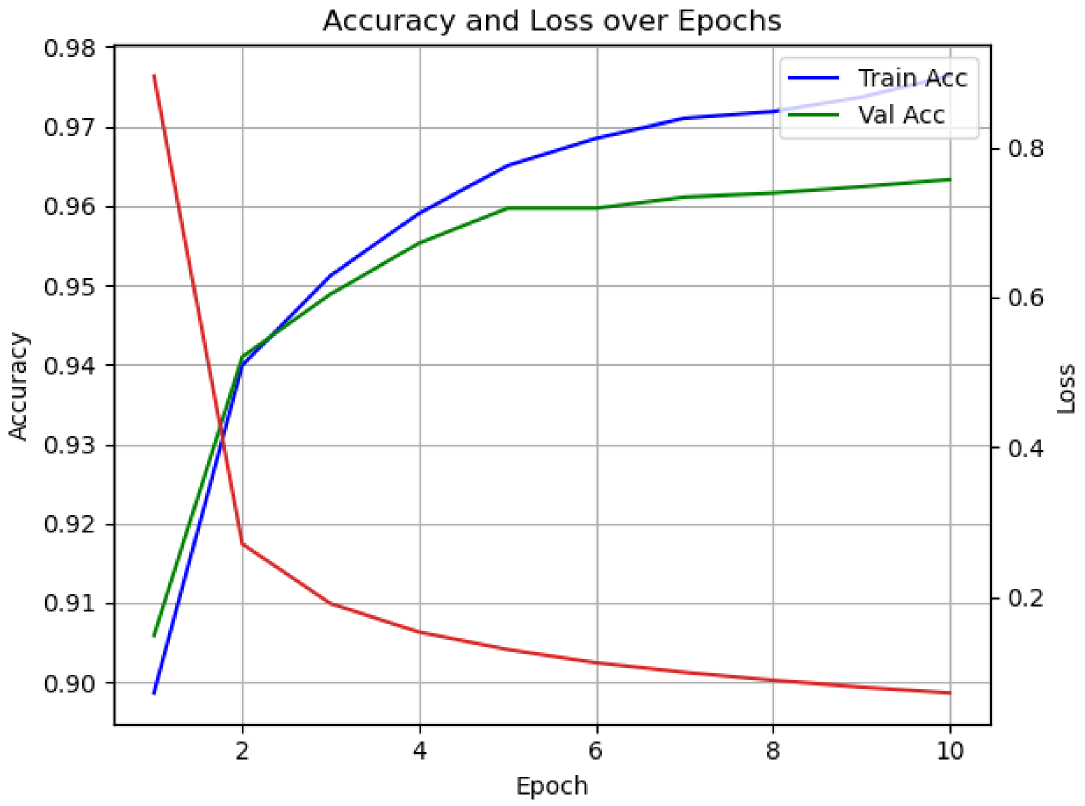

Figure 4: Evolution of the loss and accuracy on the training and validation datasets during