Machine Learning - Assignment 1

Brian Pulfer

October 2019

This document represents the personal submission of the assignment no. 1 of the Machine Learning course taking place at USI (*Università della Svizzera Italiana*) during the fall semester 2019 of the Master in Artificial Intelligence.

1 Exercise 1: The Perceptron

1.1

The dimension of the output $\overline{y}$, considering a batch of input data in form of $\underline{x} \in \mathbb{R}^{d \times k}$, is $\in \mathbb{R}^d$.

1.2

$$\overline{y} = \underline{x} * w + \overline{1}_d * b$$

1.3

$$E = \frac{1}{d}(\frac{1}{2}(\hat{y} - \overline{y})^T(\hat{y} - \overline{y}))$$

1.4

$$\frac{dE}{dw} = \frac{dE}{d\overline{y}} * \frac{d\overline{y}}{dw} = x^T * \frac{1}{d}(\overline{y} - \hat{y})$$

1.5

$$w^{new} = w^{old} - \eta \frac{dE}{dw^{old}}$$

1.6

New values of the weights after applying one step of gradient descent:

$$w_1 = 0.17$$

$$w_2 = -0.02$$

$$w_3 = 0.02$$

Note that these values are computed evaluating the whole set at once, and averaging the 'desired changes to the weights' from every observation $x_1, x_2, ..., x_{10}$.

1.7

The gradient descent allows the network to learn how the weights must be changed to minimize the loss function. The backpropagation allows to use already calculated gradients to calculate further gradients going back in the network (from output layer to input layer).

2 Exercise 2: Simple Machine Learning Framework

2.1

The *tanh* is defined as

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

and it's derivative is

$$tanh'(x) = \frac{4}{(e^{-x} + e^x)^2}$$

While the sigmoid function is defined as

$$\sigma = \frac{1}{1 + e^{-x}}$$

and it's first derivative is

$$\sigma' = \frac{e^x}{(e^x + 1)^2}$$

2.2

Forward pass for linear layer is implemented as

$$x * W + b$$

while the backward pass is implemented as

$$dW = x_{old}^T * error$$

$$db = \frac{1}{n} * \sum_{i=0}^{n} error_i$$

$$d\_input = error * w_{old}^T$$

2.3

The forward pass for the MSE is implemented as

$$MSE = \frac{1}{n} * (\frac{1}{2}(T - Y)^T * (T - Y))$$

while the backward pass is implemented as

$$\frac{1}{n} * (Y - T)$$

2.4

For what concerns the sequential layer, the forward step is a call to every *forward()* method of every module. The input for the first module is the given input, then for the next modules the input is the output of the previous.

For the backward step the loss function's methods *forward()* and *backward()* must first be called in order to calculate the first error to propagate back to the network (in reversed order w.r.t to the forward pass).
The error is propagated back to the last module of the network (method *backward()*), which result is passed to the second-last module and so on.

2.5

The network is built as

```
Sequential([Linear(2, 50),
            Tanh(),
            Linear(50, 30),
            Tanh(),
            Linear(30, 1),
            Sigmoid()])
```

while the one_step_training method is implemented as such

```
def train_one_step(model, loss, learning_rate, inputs, targets):
    predictions = model.forward(inputs)
    error = loss.forward(prediction=predictions, target=targets)

    gradient = loss.backward()
    model.backward(gradient * learning_rate)

    return error
```

3 Exercise 3: Handwritten Digit Recognition

3.1

The variable *n_train* is given. It represents the index at the 70% of the total training set. A 70-30 split (70% still for training, 30% for validation) can be achieved with the following code:

```
train_indices = indices[:n_train]
validation_indices = indices[n_train:]
```

and then using these indices on the total training set.

3.2

The main training loop was implemented in this way:

```
best_validation_accuracy = 0
best_epoch = -1

for epoch in range(1000):
    ## Implement
    for i in range(0, len(train_indices), batch_size):
        train_x, train_y = train_validation_set.images[train_indices[i: i + b
                           train_validation_set.labels[train_indices[i: i + b
        error = lib.train_one_step(model,
                                   loss,
                                   learning_rate,
                                   train_x,
                                   train_y)
    validation_accuracy = validate()
    ## End
```

3.3

Testing the network is done in the following manner:

```
def verify(images, targets):
    ## Implement
    num_ok, total_num = 0, 0
    for i in range(len(images)):
        predictions = model.forward(images[i])

        prediction = np.where(predictions == np.amax(predictions))
        target = targets[i]

        if prediction == target:
            num_ok = num_ok + 1
```

```
            total_num = total_num + 1
        ## End
        return num_ok, total_num

def validate():
    accu = 0.0
    count = 0

    ## Implement. Use the verify() function to verify your data.
    for i in range(0, len(validation_indices), batch_size):
        images = train_validation_set.images[i:i + batch_size]
        labels = train_validation_set.labels[i:i + batch_size]

        partial_ok, partial_sum = verify(images, labels)
        accu, count = accu + partial_ok, count + partial_sum
    ## End

    return accu/count * 100.0
```

3.4

The early stopping is implemented in the following manner (inside the epochs loop):

```
## Implement
    ## Hint: you should check if current accuracy is better than the best so
    ## iterations ago the best one came, and terminate if it is more than 10.
    ## if needed.
    if validation_accuracy > best_validation_accuracy:
        best_validation_accuracy = validation_accuracy
        best_epoch = epoch

    if epoch - best_epoch > 10:
        break
    # end
```