Report of the First exercise: Classifying MNIST with MLPs

Joshua Heipel

matriculation number: 3706603

The first exercise was about implementing a Neural Network in Python by completing the given code on https://github.com/aisrobots/dl-lab-2018 (https://github.com/aisrobots/dl-lab-2018). The object oriented approach containes classes for the layers (different behaviour for the first, the hidden and the last layers), the activation function and the neural network. Each of the layer classes and the activation function provide methods for forward- and backwardpropagation (fprop() and bprop()) through the network.

Forward Pass

During a forward pass, the output of a given layer is calculated by **a = W * x + b** and **h = t(a)**, where **t** is the activation function. **a** and **t(a)** are stored for the backward pass. To calculate the output for different inputs in parallel, the numpy dot product can be used. The overall output of the network **Y_pred** for a given input **X** is then computed with the predict() method of the neural network class, which just iterates over all layers. The main functions are given below:

In [ ]:
```
class Activation(object):

    def fprop(self, input):
        self.last_input = input
        return self.act(input)

class FullyConnectedLayer(Layer, Parameterized):

    def fprop(self, input):
        self.last_input = input
        a = np.dot(input, self.W) + self.b
        t = self.activation_fun
        h = t.fprop(a) if t else a
        return h

class NeuralNetwork:

    def predict(self, X):
        output = X
        for layer in self.layers:
            output = layer.fprop(output)
        Y_pred = output
        return Y_pred
```

Backward Pass

For the backward propagation the gradient of the last layer (loss function) with respect to the input activation needs to be computed first. This is done with the input_gradient() method of the SoftmaxOutput or LinearOutput class. Then for each hidden layer the "error_term" (gradient with respect to its input activation) is calculated by multiplying the "output_gradient" of the following layer (calculated one step before) with the gradient of the Activation function. Afterwards the gradient of the parameters for the hidden layer and the new "input_gradient" to the previous layer (used in the next step) can be computed. The backpropagation through the whole network is done with the backpropagation() method of the neural network class, while iterating over all layers in reverse order. Again numpy dot product is used to parallelize the computation. The main functions look like the following:

In [ ]:
```python
class Activation(object):

    def bprop(self, output_grad):
        return output_grad * self.act_d(self.last_input)

class FullyConnectedLayer(Layer, Parameterized):

    def bprop(self, output_grad):
        n = output_grad.shape[0]
        t = self.activation_fun
        # calculate the gradient with respect to the input activation
        error_term = t.bprop(output_grad) if t else output_grad
        # calculate the gradient of the parameters
        self.dW = np.dot(self.last_input.transpose(), error_term) / n
        self.db = np.mean(error_term, axis=0)
        # calculate the input gradient to the previous laye
        grad_input = np.dot(error_term, self.W.transpose())
        return grad_input

class LinearOutput(Layer, Loss):

    def input_grad(self, Y, Y_pred):
        return Y_pred - Y

class SoftmaxOutput(Layer, Loss):

    def input_grad(self, Y, Y_pred):
        return Y_pred - Y

class NeuralNetwork:

    def backpropagate(self, Y, Y_pred, upto=0):
        next_grad = self.layers[-1].input_grad(Y, Y_pred)
        output = next_grad
        for layer in reversed(self.layers[:-1]):
            output = layer.bprop(output)
        grad = output
        return grad
```

Gradient descent

After all to train the network for given dataset (X,Y) the (stochastic) gradient descent methods of the neural network class are used. During each cycle, both, the stochastic version (sgd_epoch() method) and the standard gradient descent (gd_epoch() method), partitionate the given dataset into mini-batches first. Then for each mini-batch a forward and backward pass is computed. While the stochastic version updates the parameters of the layers directly during each forward-backward pass, the gradients of all parameters in the standard version are accumulated over all mini-batches before updating. Training of the network is done by a given number of update-cycles (epoches). Both gradient descent methods are found in the Jupyter Notebook document.

Optimization of meta parameters

Finally the implementation should be used to classify handwritten digits of the MNIST dataset. It was found, that the given parameter setting already leeds to a classification error on the validation dataset of below 3%. Changing parameters (number of units per layer, different activation functions, parameter initialization, learning rate, batch size and number of training epoches) as well as adding more layers could improve the resulting validation error just a little. So the final setting doesn't differ from the original setting much. The illustration shows, that the loss on the training data as well as the classification error on the training and test data is still improving towards the end of the update-cycles, while the loss on the validation data has already saturated very early.