Neuronal Deep Learning for Autonomous Systems

Summer Term 2017

Neuroscientific System Theory
Michael Lutter, Prof. Jörg Conradt

Worksheet 2: Neural Network Basics

This worksheet covers the basics of Neural Networks ranging from forward inference, parameter initialization, backpropagation, gradient descent, momentum and learning rate scaling. In class, we are going to discuss the example solution of some problems. The problems, which we will not explicitly cover during the exercise are meant for self-study. Subproblems, which are marked as optional, are not relevant for the exam. **We will provide an example solution for every task.**

For all individual tasks, we prepared **Python scripts** that should give you an intuition about one possible class architecture. However, you are free to adapt the class architecture or build your own functions. The provided solution will conform with the provided class architecture. The Python scripts are tested on 2.7.12.

Task A: Neural Network

Within this task, you should write a Neural Network class that can simulate any feedforward Multi-Layer Perceptron with different neuron types and cost functions. The class should be able to simulate MLP's with specified input, output and hidden width and depth. Therefore, the neural network class should consist of the following methods:

class NeuralNetwork:

def __init__(w_i, w_h, n_h, w_o, hidden_neuron_type, output_neuron_type):
The constructor receives the layer width of the input- (w_i), hidden- (w_h) and output layer (w_o), the number of hidden layers (n_h) as well as the neuron types of the hidden and output layers. From this information, the constructor builds the networks and initializes the weights.

def eval(self, x):
The eval method computes the network output from the input x.

def weight_initialization(self):
The weight initialization method initializes / resets all network parameters to the parameter distribution specified by the internal variable weight_pdf.

def j(self, x, y):
The j method computes the cost function from the given input (x) and desired values (y).

def grad_j(self, x, y):
The grad_j method returns the gradient of the cost function with respect to **all network parameters**. The gradient is computed via backpropagation from the mini-batch described by (x, y).

def train(self, x, y, reset=False, plot=True):
The train method uses the optimizer implemented in Task B to train the neural network. Optionally, this method plots the network output before and after training as well as the error plot. An example is shown in Figure 1.

The class definition can be found in the NeuralNetwork.py script.

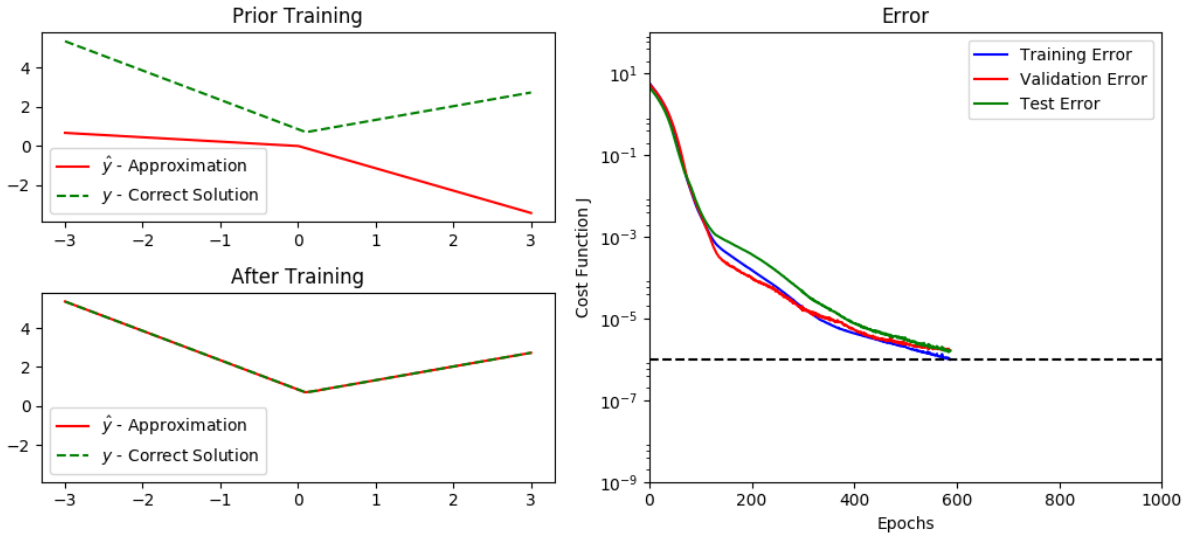

Figure 1 *Network Performance before and after training. The dotted green lines show the ideal piecewise linear function. The red line shows the neural network approximation. The right plot shows the error curves of the training-, validation and test-error.*

Task B: Optimization

Within this task, you should implement a gradient descent optimization class. The class performs stochastic gradient descent (SGD) for a given optimization function. This SGD can be optionally extended by momentum and adaptive learning rate scaling. The class should have the following methods:

class GradientDescent:

def __init__(j, grad_j, x, y, theta, parameter):
The constructor receives functions to compute the cost function *(j)* and the gradient of the cost function *(grad_j),* the training set *(x,y),* a pointer to the trainable parameters *(θ)* and a parameter dictionary, which contains the optional hyperparameter *(parameter)* of SGD. From these input variables, the iterative gradient descent procedure is started and the trainable parameters are optimized. The gradient descent algorithm should stop, if the cost function is lower than j_{min} or the maximum number of iteration is reached. Additionally, the gradient descent should incorporate early stopping to prevent overfitting. To achieve this functionality, one should use the other class methods and passing functions as arguments.

def slice_dataset(self, x):
This method slices the dataset into a training-, validation and test set.

def fixed_alpha(self, grad_j)
This method returns the fixed learning rate α.

def rmsprop(self, grad_j)
This method returns the learning rate α computed via RMSprop.

def adam(self, grad_j)
This method returns the learning rate α computed via ADAM.

def standard_momentum(self, alpha, grad_j)
This method computes the standard momentum and returns the parameter update $\Delta\theta$. The parameter update can be used to update the parameters described by

$$\theta_{t+1} = \theta_t + \Delta\theta$$

def adam_momentum(self, alpha, grad_j)
This method computes the ADAM momentum and returns the parameter update $\Delta\theta$. The parameter update can be used to update the parameters described by

$$\theta_{t+1} = \theta_t + \Delta\theta$$

The class definition can be found in the NeuralNetwork.py script.

Task C: Network Performance

Once you implemented the Neural Network and Gradient Descent class and verified the functionality, you should evaluate the performance on a regression problem.

1. Piecewise Linear Regression.
   For this example, you approximate a piecewise linear function with a neural network. The piecewise linear function is described by

$$f(x) = \begin{cases} +0.7\ (x - x_0) + 0.7 & x > x_0 \\ -1.5\ (x - x_0) + 0.7 & x \leq x_0 \end{cases}$$

   where x_0 corresponds to a horizontal shift. To approximate this function you are going to use a network with two hidden rectified linear neurons and a linear output neuron. Such a network has sufficient capacity to model such a piecewise linear function.

   a) Manually derive at least 5 different model parameters that approximate this function perfectly.

   b) Plot the cost function w.r.t. to every model parameter. Can you identify plateaus or saddle points that could prevent learning this function?

   c) Train the neural network with gradient descent and observe the success rate of the network converging to a correct solution. Why does the model not always converge to the correct solution even though the model capacity is sufficient? How can the parameters be initialized that the gradient descent optimization cannot find the correct solution?

   d) With the x_0 parameter one can horizontally shift the function $f(x)$. Try out x_0 close to the boundaries of $x \in [-3, 3]$ and observe the training behavior. What is the success rate for learning this horizontally shifted function?

2. Polynomial Regression
   For this example, you approximate a cubic function with a neural network. The cubic function is described by

$$f(x) = 0.1\ (x - x_0)^3 + 0.0\ (x - x_0)^2 - 0.695\ (x - x_0) + 0.2$$

   . This polynomial should be approximated by a neural network with width *n* and depth *m*. The hidden neurons are rectified linear neurons and the output neurons are linear neurons.

   a) Change the width and depth of the neural network and observe the number of linear regions within the trained output. How does the number of linear regions scale with width and depth? How does this compare to the theoretical number of regions given by

$$O\left(\binom{n}{d}^{d\,(m-1)} n^d\right)$$

where d is the input dimensionality, m the number of hidden layers and n the width of the hidden layers. How can the difference be explained?

b) Change the mini-batch size, which are used to compute the gradient and observe the error plot during training. How does the error curves differ when changing the mini-batch size?

c) Vary the momentum parameter from 0 to 0.95 and observe the differences in the error plot. How does the error plot differ for different momentum parameters?