ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Models, Algorithms and Data (MAD): Introduction to Computing

Spring semester 2019

Prof. Dr. Jens Honore Walther
Dr. Julija Zavadlav
ETH Zentrum, CLT
CH-8092 Zürich

Set 7

Issued: 05.04.2019; Due: 14.04.2019

In this exercise, you will learn about the workhorse behind neural networks training, i.e. the Backpropagation algorithm, and get hands-on experience with `TensorFlow` by completing a Regression task. In order to complete the programming task, you should follow the **template source code** provided for you and proceed incrementally, always remembering to test and debug your program after every modification. We recommend using `Python3` and loading the `TensorFlow` library that enables easy implementation of neural network architectures.

In this exercise, we deal with the **multilayer perceptron** (also called feed-forward neural network). We assume that our neural network has L layers, each with $\{h_1, \ldots, h_L\}$ nodes (selected by the user), while for simplicity h_0 is the dimension of the input (problem specific), and h_{L+1} the output dimension (problem specific). This notation may differ in the literature. The input equations are:

$$z_j^1 = \sum_{i=1}^{h_0} w_{ij}^1 x_i + b_j^1, \quad \forall j \in \{1, \ldots, h_1\}, \quad \text{and} \quad o_j^1 = \sigma(z_j^1) \tag{1}$$

where $\mathbf{x}$ is the input and $x_i = (\mathbf{x})_i$ is the i^{th} input component. j is a neuron in the first layer, h_0 the input dimension, h_1 the size of the first layer (number of hidden nodes), σ the activation function (non-linearity), and o_j^1 the output of the neuron j in the first layer. In a similar fashion, the equations between two consecutive layers are

$$z_j^k = \sum_{i=1}^{h_{k-1}} w_{ij}^k o_i^{k-1} + b_j^k, \quad \forall j \in \{1, \ldots, h_k\}, \quad \text{and} \quad o_j^k = \sigma(z_j^k), \tag{2}$$

the argument behind the notation w_{ij}, is that the weight connects the node i of layer $k-1$ to the node j of layer $k+1$. b_j^k denotes the bias of node j in layer k. Finally, in the output layer, we have

$$z_j^{L+1} = \sum_{i=1}^{h_L} w_{i,j}^{L+1} o_i^L + b_j^L, \quad \forall j \in \{1, \ldots, h_{L+1}\}, \quad \text{and} \quad o_j^{L+1} = \sigma(z_j^{L+1}). \tag{3}$$

Stacking all neural network outputs in one vector we get the total output $f^w(\mathbf{x}) = \mathbf{o}^{L+1}$. The most common technique to train this network is **supervised learning** based on **Backpropagation**, where examples (**training data**) are used to learn a mapping from the input to the output. In the following, we are going to explain how Backpropagation works and derive the mathematical expressions.

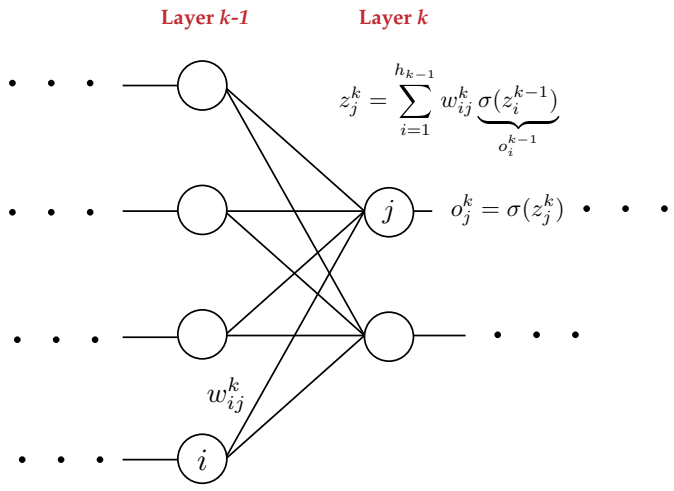

Figure 1: For node j at layer k, we denote with z_j^k the weighted sum of the outputs of layer $k-1$. By applying the activation function $\sigma(\cdot)$ to this weighted sum we end up with the final output of node j of layer k, i.e. o_j^k.

Question 1: Shedding light in Backpropagation

In this exercise, we are going to explain how the Backpropagation algorithm works. Backpropagation is the workhorse of state-of-the-art deep neural network architectures. The name is a shortcut for **Backpropagation of errors**. In this exercise, we assume that the output is one dimensional and the loss function is the mean squared error (MSE), defined as

$$E = \frac{1}{N} \sum_{s=1}^{N} ||f^w(x_s) - y_s||^2, \tag{4}$$

where s is the sample index and (x_s, y_s) for $s \in \{1, \ldots, N\}$ are the **training data**. Backpropagation is a method that allows us to compute the **gradient** of each **weight** of the neural network with respect to the loss function. This gradient is expressed mathematically as $\partial E / \partial w_{ij}^k$, where w_{ij}^k is the weight from node i to node j at layer k and E is the loss. For simplicity we will derive the Backpropagation rules based on the loss of a single training sample $E_s = ||f^w(x_s) - y_s||^2$. The respective gradient we need to compute is $\partial E_s / \partial w_{ij}^k$. The overall gradient is then averaged among training samples. Using these gradients we will formulate an optimization update for each weight, moving towards a descent direction in order to minimize the loss.

The first step of the Backpropagation algorithm is the **forward pass**. The input x_s is fixed. In the forward pass we compute the node values z_j^k and o_j^k for each node j in each layer k of the neural network, as well as the final output $f^w(x_s)$, according to the equations (1-3). As the name implies, we just propagate forward the input values, computing all intermediate variables in the neural network. Next, we need to derive the formulas for the **backward pass**. **To simplify notation biases are ignored.** Incorporating them is just a matter of notation and left as an optional exercise to the reader.

a) As a first step, we rewrite the error $\partial E_s/\partial w_{ij}^k$ using the chain rule as

$$\frac{\partial E_s}{\partial w_{ij}^k} = \frac{\partial E_s}{\partial z_j^k}\frac{\partial z_j^k}{\partial w_{ij}^k}, \tag{5}$$

where z_j^k is the output of node j at layer k **before** applying the activation function. We refer to the first term as error gradient $\delta_j^k = \partial E_s/\partial z_j^k$. Find a formula for the second term $\partial z_j^k/\partial w_{ij}^k$ using formula (2). Assume that h_{k-1} is the number of nodes (hidden dimension) at layer $k-1$ and consult Figure 1.

b) Next, we need an expression for the error gradient δ_j^k. Again, using the chain rule we can write

$$\delta_j^k = \frac{\partial E_s}{\partial z_j^k} = \sum_{i=1}^{h_{k+1}} \frac{\partial E_s}{\partial z_i^{k+1}}\frac{\partial z_i^{k+1}}{\partial z_j^k} = \sum_{i=1}^{h_{k+1}} \delta_i^{k+1}\frac{\partial z_i^{k+1}}{\partial z_j^k}, \tag{6}$$

where we decomposed δ_j^k to contributions by the error gradients of the next layer δ_i^{k+1}. Find an expression for the term $\partial z_i^{k+1}/\partial z_j^k$. (Hint: use the fact that $o_j^k = \sigma(z_j^k)$ and z_i^{k+1} is a weighted sum of $o_{j'}^k$ for $j' \in \{1, \ldots, h_k\}$. The final outcome includes the derivative of the activation function σ')

c) Plug the formula you computed in the previous step to equation (6) to compute an expression for δ_j^k, that depends on δ_i^{k+1}, w_{ji}^{k+1}, and σ'.

d) Now plug the expressions you found for δ_j^k and $\partial z_j^k/\partial w_{ij}^k$ in (5) to formulate the Backpropagation weight update rule $\partial E_s/\partial w_{ij}^k$.

e) Assume you are using a gradient descent optimizer with step-size η to update the weights of the neural network. Formulate the update equation in terms of the weight update Δw_{ij}^k. Take into account the **total** error E and decompose it into the contributions of E_s.

f) [optional] There are many activation functions commonly used in practice (ReLU, tanh, sigmoid, etc.). Can you find one that satisfies the following expression?

$$f'(x) = f(x)\Big(1 - f(x)\Big) \tag{7}$$

g) [optional] You build a deep neural network with **tanh** activation functions and you observe that training converges slowly. After debugging you find out that most **tanh** activation functions **saturate**. Can you justify this behavior based on the Backpropagation algorithm and the form of the nonlinearity? Which activation function would you propose to solve the problem?

Question 2: Regression with `TensorFlow`

In this exercise, you are asked to build a neural network that models the sinusoidal function $\sin(x)$ in the domain $x \in [0, 2\pi]$. We train the neural network on noisy data. You are instructed to follow the template provided to you and fill in the missing code blocks.

a) We first generate 200 noisy data samples for training the neural network. In order to avoid overfitting, we shuffle and split the data into a training set and a validation set (see note in the template source code). Build a neural network with one intermediate layer with $h = 10$ hidden units and **tanh** activation function, to solve the task. Do not use an activation in the output (identity). Bear in mind that the batch size is an additional dimension in the tensorflow model.

b) First, set the noise level to zero (no noise). Use the mean squared error loss (MSE) and an optimizer of your choice (gradient descent, Adam, etc.). Set the batch size to 10 and run 10000 epochs. Can you recover approximately the sinusoidal signal? Plot the result and compare it qualitatively. Try different learning rates.

c) Increase the noise in the training data. Can you still recover the sinusoidal signal?

d) Another application of neural networks is the prediction of time series signals, where the available data is the evolution of a signal in time. Time series data might be for example data about the weather in an area, the evolution of stock prices, etc. How would you use a neural network on time series data? How could you improve your method if you know that long-term correlations exist? (You do not have to program, just give a short explanation)