Deep Learning 03

Prof. Dr. David Spieler – `david.spieler@hm.edu`

University of Applied Sciences Munich

October 31, 2018

Multilayer Perceptron

Introduction

In order to be able to create more complex and expressive models we can

- stack together neurons in multiple layers,
- use different activation functions to model non-linear relationships, and
- adapt the goal of learning using different cost functions.

All these building blocks form a modular toolkit for deep learning which we will discover in this chapter.

Multilayer Perceptron

Introduction

Multilayer Perceptron

A multilayer perceptron (MLP) is a network of interconnected perceptrons with a unique input layer, several (or no) hidden layers, and a unique output layer.

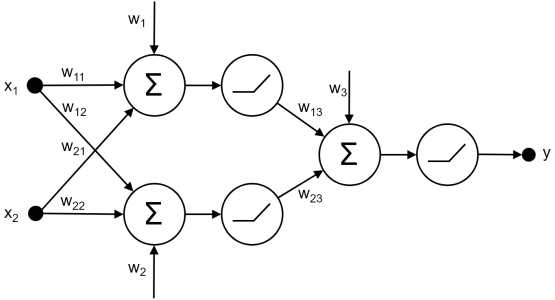


Figure 4: A MLP for learning xor with input layer **x**, hidden layer (neurons 1, 2), and output layer (neuron 3).

Multilayer Perceptron

Introduction

ReLU Activation Function

The rectified linear activation function (ReLU) is defined as

$$\alpha(x) = \max\{0, x\}.$$

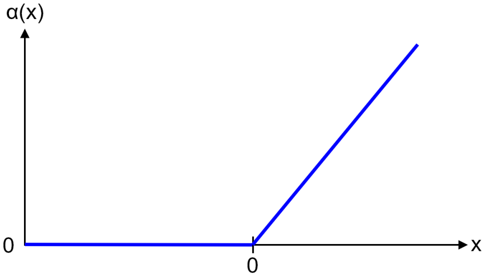


Figure 5: The ReLU activation function.

Multilayer Perceptron

Introduction

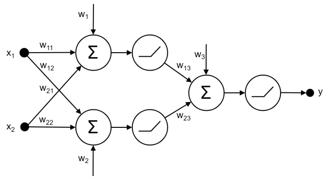
Example

Parameters for the XOR MLP:

$$\mathbf{w}_1 = 0, \mathbf{w}_{11} = 1, \mathbf{w}_{12} = 1,$$

$$\mathbf{w}_2 = -1, \mathbf{w}_{21} = 1, \mathbf{w}_{22} = 1,$$

$$\mathbf{w}_3 = 0, \mathbf{w}_{13} = 1, \mathbf{w}_{23} = -2$$



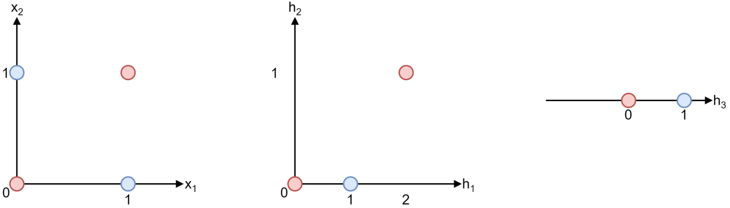


Figure 6: Mapping of inputs to outputs in the XOR MLP.

Multilayer Perceptron

Introduction

Cost Function

A cost function for a MLP is a function $C : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$ that quantifies the error $C(\mathbf{y}, \hat{\mathbf{y}})$ a MLP makes when outputting $\mathbf{y}$ on a (training or test) sample $\mathbf{x}$ when it should output $\hat{\mathbf{y}}$.

Multilayer Perceptron

Introduction

Least Squares Cost Function

The least squares cost function is defined as

$$C(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2} \sum_i (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2.$$

Note

The least squares cost function is a general purpose cost function that can be used in classification and regression settings.

Multilayer Perceptron

Backpropagation Learning

Goal

We want to develop a learning algorithm for general MLP but want to simplify presentation as far as possible. Thus, we assume we are given a MLP with activation function α, weights $\mathbf{w}$, and an error metric e for a single training data point $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$. Since we want to use gradient descent, we are interested in computing

$$\nabla_{\mathbf{w}} e = \left[\frac{\partial e}{\partial \mathbf{w}_{ij}} \right].$$

Generalization

Note that the method can be extended to varying activation functions and multiple training data points accordingly.

Multilayer Perceptron
Backpropagation Learning

Let us start with the last layer of the MLP:

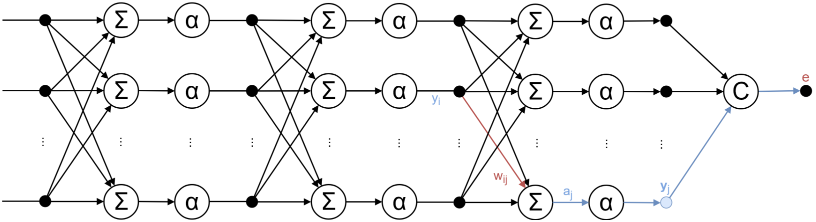


Figure 7: Backpropagation starts in last layer.

Using the chain rule, we arrive at

$$\frac{\partial e}{\partial \mathbf{w}_{ij}} = \frac{\partial e}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial \mathbf{w}_{ij}}$$

Multilayer Perceptron

Backpropagation Learning

For variables y_j in the last layer, we can directly compute the derivative

$$\frac{\partial e}{\partial y_j}$$

since it only depends on the definition of the cost function C.

Example

For the least squares cost function $e = C(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}\sum_i (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$ the above derivative is

$$-(\hat{\mathbf{y}}_j - \mathbf{y}_j).$$

Note that $y_j = \mathbf{y}_j$ since we evaluate the vector valued training data in the last layer.

Multilayer Perceptron

Backpropagation Learning

The derivative $\frac{\partial y_j}{\partial a_j}$ depends on the definition of the activation function α since

$$y_j = \alpha(a_j)$$

and thus

$$\frac{\partial y_j}{\partial a_j} = \alpha'(a_j)$$

Example

For the ReLU activation function $\alpha(x) = \max\{0, x\}$ (which is not differentiable at 0) one usually uses

$$\alpha'(x) \stackrel{!}{=} \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise.} \end{cases}$$

Multilayer Perceptron

Backpropagation Learning

For the derivative

$$\frac{\partial a_j}{\partial \mathbf{w}_{ij}}$$

we recall the computation

$$a_j = \sum_{k \in \mathrm{Pred}(j)} \mathbf{w}_{kj} y_k$$

in whose derivative only the term for $k = i$ is non-zero and thus

$$\frac{\partial a_j}{\partial \mathbf{w}_{ij}} = y_i .$$

Multilayer Perceptron

Backpropagation Learning

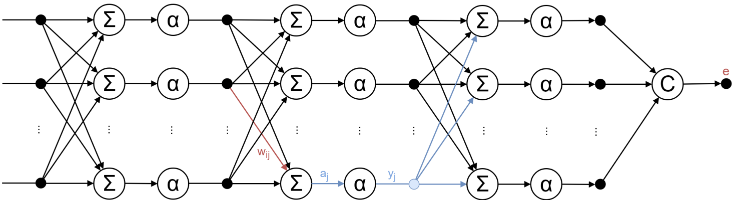


Figure 8: Backpropagation propagates the error from the last to the first layer.

In previous layers, i.e., if variable $\mathbf{y}_j$ is not in the last layer, we need to take into account all possible path through the network:

$$\frac{\partial e}{\partial y_j} = \sum_{i \in \mathrm{Succ}(j)} \underbrace{\frac{\partial e}{\partial y_i}}_{\text{recursive}} \underbrace{\frac{\partial y_i}{\partial a_i}}_{\alpha'(a_i)} \frac{\partial a_i}{\partial y_j}.$$

Multilayer Perceptron

Backpropagation Learning

For the factor

$$\frac{\partial a_i}{\partial y_j}$$

we insert the definition

$$a_i = \sum_{k \in \mathrm{Pred}(i)} \mathbf{w}_{ki} y_k$$

and get

$$\frac{\partial a_i}{\partial y_j} = \frac{\partial \sum_{k \in \mathrm{Pred}(i)} \mathbf{w}_{ki} y_k}{\partial y_j} = \mathbf{w}_{ji}.$$

Multilayer Perceptron

Backpropagation Learning

Summary: Backpropagation Formulas

The final formulas for the backpropagation algorithm are:

$$\frac{\partial e}{\partial \mathbf{w}_{ij}} = y_i \cdot \alpha'(a_j) \cdot \frac{\partial e}{\partial y_j}$$

$$\frac{\partial e}{\partial y_j} = \begin{cases} \sum_{i \in \mathsf{Succ}(j)} \mathbf{w}_{ji} \cdot \alpha'(a_i) \cdot \frac{\partial e}{\partial y_i} & \text{if } j \notin \text{ last layer} \\ \frac{\partial C}{\partial \mathbf{y}_j}(\mathbf{y}, \hat{\mathbf{y}}) & \text{otherwise} \end{cases}$$

Computation

Note that first the values a_i and y_i are computed using a forward pass followed by a backpropagation pass using the above formulas.

MLP Building Blocks

MLP Building Blocks

MLP Building Blocks

Activation Functions

Linear Activation Function

The linear activation function is defined as

$$\alpha(x) = x.$$

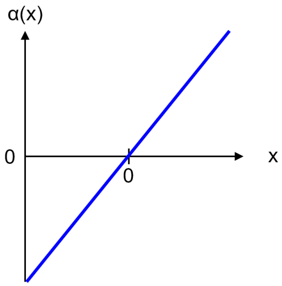


Figure 10: The linear activation function.

MLP Building Blocks

Activation Functions

Logistic/Sigmoid Activation Function

The logistic/sigmoid activation function is defined as

$$\alpha(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}.$$

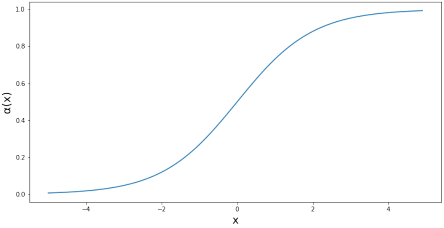


Figure 11: The logistic activation function.

MLP Building Blocks

Activation Functions

Softmax Activation Function

The softmax activation function is defined as

$$\alpha(\mathbf{x}) = \begin{bmatrix} e^{\mathbf{x}_1} \\ e^{\mathbf{x}_2} \\ \vdots \\ e^{\mathbf{x}_n} \end{bmatrix} \cdot \frac{1}{\sum_{i=1}^{n} e^{\mathbf{x}_i}}.$$

Note

The softmax activation function maps a vector to a vector and thus is represented graphically by a layer block. It is used to scale incoming real values of arbitrary range to values from $[0, 1]$ which add up to 1 (like probabilities) usually in the output layer.

MLP Building Blocks
Activation Functions

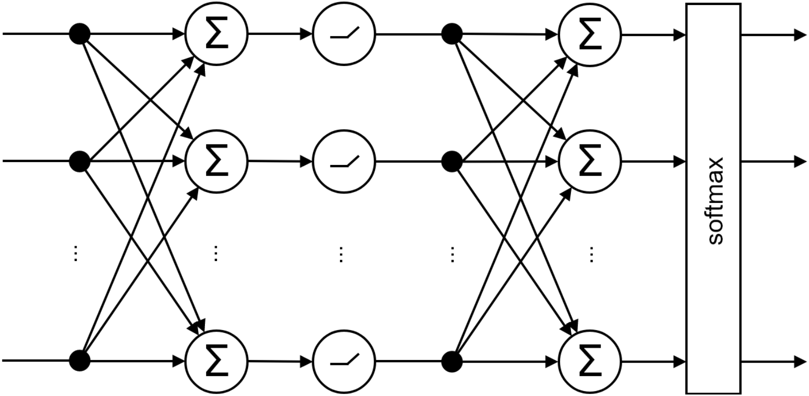


Figure 12: A MLP with an input layer, a ReLU hidden layer, and a softmax output layer.

MLP Building Blocks
Activation Functions

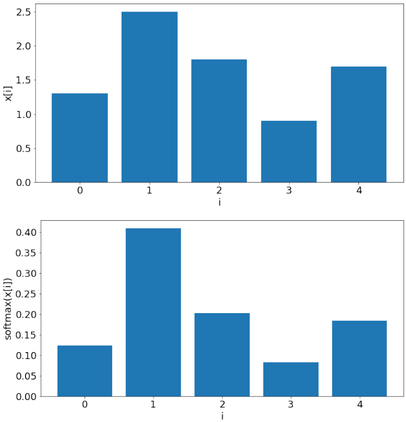


Figure 13: Inputs to a softmax layer and its outputs.

MLP Building Blocks

Cost Functions

Binary Cross-Entropy Cost Function

The binary cross-entroy cost function is defined as

$$C(y, \hat{y}) = -\hat{y} \log y - (1 - \hat{y}) \log(1 - y).$$

Note

The binary cross-entropy cost function is used for binary classification which explains the scalar instead of vector notation. The neural network output y shall be interpreted as the probability that an input belongs to a certain class. The true label $\hat{y}$ can either be a class probability as well or an class indicator (0 or 1). The binary cross-entropy cost function is also often called log loss function in the literature.

MLP Building Blocks

Cost Functions

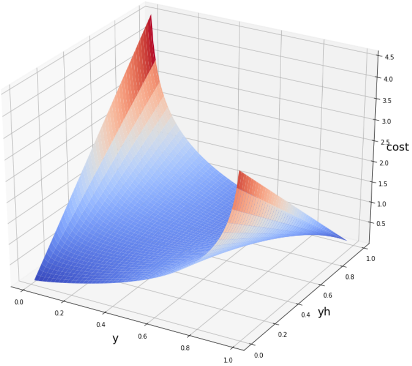


Figure 14: The cross-entropy cost function favours (nearly) equal class probabilities and penalizes differences.

MLP Building Blocks

Cost Functions

Cross-Entropy Cost Function

The (multi-class) cross-entroy cost function is defined as

$$C(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \hat{\mathbf{y}}_i \log \mathbf{y}_i$$

where $\mathbf{y}_i$ are class probability (network outputs) and $\hat{\mathbf{y}}_i$ are the true class probabilities or class indicators.

Loss Function

In the literature, sometimes the term loss function is used for cost function.

MLP Building Blocks
Gradient Descent

In the following, we will introduce some variants of gradient descent algorithms. For this, we will introduce the notation

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \mid 1 \leq i \leq n\}$$

for a set of n training data points.

MLP Building Blocks

Gradient Descent

Algorithm 2 batch_gd($\mathcal{D}$, η)

initialize $\mathbf{w}$
while stopping criteria not met **do**
    for $i = 1, \ldots, n$ **do**
        $\Delta\mathbf{w} = \Delta\mathbf{w} + \frac{1}{n}\nabla_{\mathbf{w}} e^{(i)}$
    end for
    $\mathbf{w} = \mathbf{w} - \eta\Delta\mathbf{w}$
end while

The (batch) gradient descent (GD) optimization algorithm uses all training data points in each iteration for approximating the gradient and to update the weights $\mathbf{w}$.

MLP Building Blocks

Gradient Descent

Algorithm 3 mini_batch_gd($\mathcal{D}$, η, b)

```
initialize w
while stopping criteria not met do
    B = random.choice([1, ..., n], b)
    for i ∈ B do
        Δw = Δw + (1/b) ∇_w e^(i)
    end for
    w = w − ηΔw
end while
```

Mini-batch gradient descent (MBGD) uses only a subset (batch) of the training data points in each iteration for approximating the gradient and to update the weights $\mathbf{w}$.

MLP Building Blocks

Gradient Descent

Algorithm 4 stochastic_gd($\mathcal{D}$, η)

1: initialize $\mathbf{w}$
2: **while** stopping criteria not met **do**
3: $\quad i = \text{random.uniform}(1, n)$
4: $\quad \Delta\mathbf{w} = \nabla_{\mathbf{w}} e^{(i)}$
5: $\quad \mathbf{w} = \mathbf{w} - \eta\Delta\mathbf{w}$
6: **end while**

In its extreme, the stochastic gradient descent (SGD) optimization algorithm uses only a single training data point in each iteration to update the weights $\mathbf{w}$ but in the literature, MBGD is also often regarded as part of SGD.

MLP Building Blocks
Gradient Descent

The following aspects influence the choice of the optimization algorithm:

- More data points improve the precision of the gradient estimate (but not linearly).
- Multicore architectures and GPUs need bigger batch sizes to be fully utilized.
- Bigger batch sizes in parallel need more memory.
- SGD is fast but the gradient approximation is noisy and thus works well if there are many (local) minima.

MLP Building Blocks

MLP Layer Representation

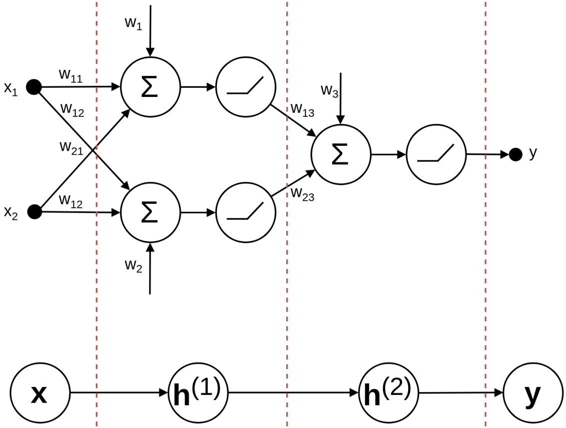


Figure 15: If we are only interested in a neural network's architecture (and not in the details like weights or activation functions) we will use a layer representation.