UNIVERSITÄT BONN

Introduction to Neural Networks

Moritz Wolter

September 25, 2023

High-Performance Computing and Analytics Lab

Overview

Neural networks

Classification with neural networks

Network coding

Neural networks

The wonders of the human visual system

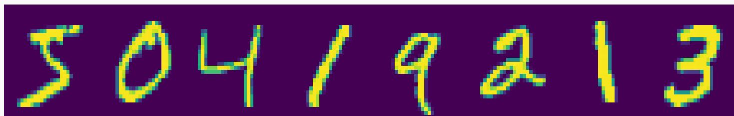


Figure: Most humans effortlessly recognize the digits 5 0 4 1 9 2 1 3.

Biological motivation

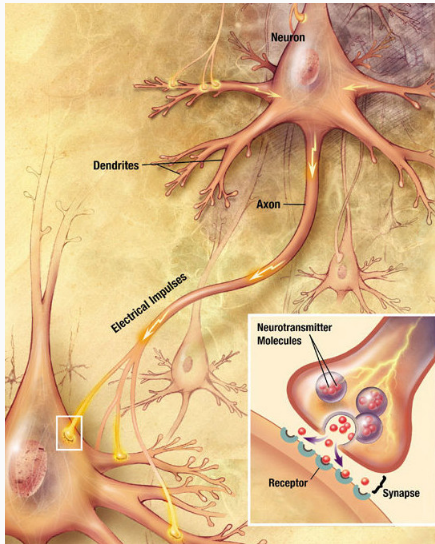
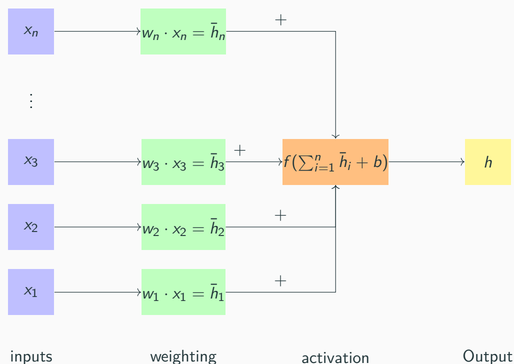


Image source: en.wikipedia.org

The perceptron

Can computers recognize digits? Mimic biological neurons,



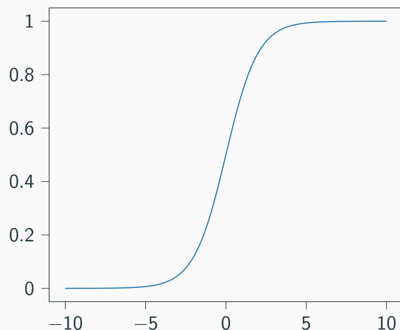
Formally a single perceptron is defined as

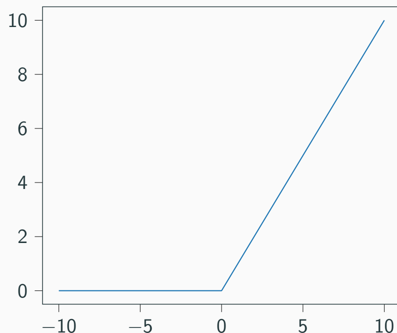
$$f(\mathbf{w}^T \mathbf{x} + b) = h \tag{1}$$

with $\mathbf{w} \in \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^n$ and $h, b \in \mathbb{R}$.

The activation function f

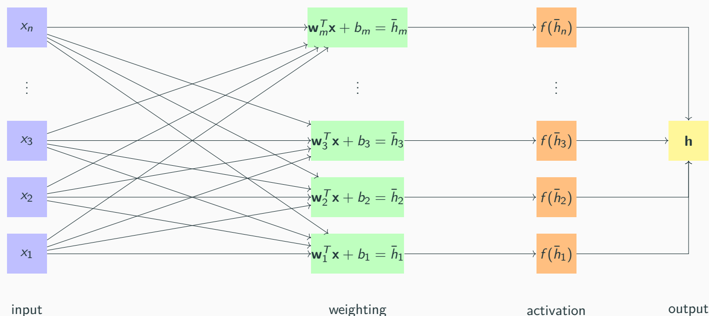
Two popular choices for the activation function f.





Arrays of perceptrons

Let's extend the definition to cover an array of perceptrons:



Every input is connected to every neuron. In matrix language, this turns into

$$\bar{\mathbf{h}} = \mathbf{W}\mathbf{x} + \mathbf{b}, \qquad \mathbf{h} = f(\bar{\mathbf{h}}). \tag{2}$$

With $\mathbf{W} \in \mathbb{R}^{m,n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $\mathbf{h}, \bar{\mathbf{h}} \in \mathbb{R}^m$.

The loss function

To choose weights for the network, we require a quality measure. We already saw the squared error cost function,

$$C_{\text{se}} = \frac{1}{2}\sum_{k=1}^{n}(\mathbf{y}_k - \mathbf{h}_k)^2 = \frac{1}{2}(\mathbf{y} - \mathbf{h})^T(\mathbf{y} - \mathbf{h}) \tag{3}$$

This function measures the squared distance from each desired output. $\mathbf{y}$ denotes the desired labels, and $\mathbf{h}$ represents network output.

The gradient of the mse-cost-function

Both the squared error loss function and our dense layer are differentiable.

$$\frac{\partial C_{\text{se}}}{\partial \mathbf{h}} = \mathbf{h} - \mathbf{y} = \triangle_{\text{se}} \tag{4}$$

The $\triangle$ symbol will re-appear. It always indicates incoming gradient information from above. If the labels are a vector of shape $\mathbb{R}^m$, $\triangle$ and the network output $\mathbf{h}$ must share this dimension.

The gradient of a dense layer

The chain rule tells us the gradients for the dense layer [Nie15]

$$\delta\mathbf{W} = [f'(\bar{\mathbf{h}}) \odot \triangle]\mathbf{x}^T, \qquad \delta\mathbf{b} = f'(\bar{\mathbf{h}}) \odot \triangle, \tag{5}$$

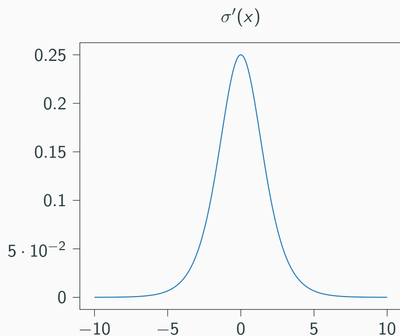
$$\delta\mathbf{x} = \mathbf{W}^T[f'(\bar{\mathbf{h}}) \odot \triangle], \tag{6}$$

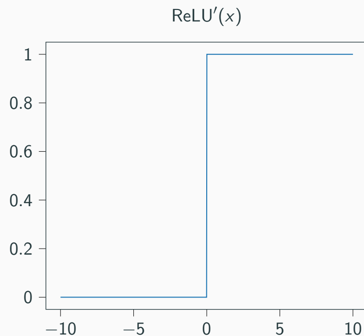
where $\odot$ is the element-wise product. δ denotes the cost function gradient for the value following it [Gre+16].
With $\delta\mathbf{W} \in \mathbb{R}^{m,n}$, $\delta\mathbf{x} \in \mathbb{R}^n$ and $\delta\mathbf{b} \in \mathbb{R}^m$. **Modern libraries will take care of these computations for you!**

Derivatives of our activation functions

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{7}$$

$$\mathrm{ReLU}'(x) = H(x) \tag{8}$$





Perceptrons for functions

The network components described this far already allow function learning. Given a noisy input signal $x \in \mathbb{R}^m$ and a ground through output $\mathbf{y} \in \mathbb{R}^m$, define,

$$\mathbf{h}_1 = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{9}$$

$$\mathbf{o} = \mathbf{W}_{\text{proj}}\mathbf{h}_1 \tag{10}$$

With $\mathbf{W} \in \mathbb{R}^{m,n}$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$. m and n denote the number of neurons and the input signal length. For signal denoising, input and output have the same length. Therefore $\mathbf{W}_{\text{proj}} \in \mathbb{R}^{n,m}$. $\mathbf{o} \in \mathbb{R}^n$ denotes the network output.

Denoising a cosine

Training works by iteratively descending along the gradients. For $\mathbf{W}$ the weights at the next time step τ are given by,

$$\mathbf{W}_{\tau+1} = \mathbf{W}_{\tau} - \epsilon \cdot \delta \mathbf{W}_{\tau}. \tag{11}$$

The step size is given by $\epsilon \in \mathbb{R}$. At $\tau = 0$, matrix entries are random. $\mathcal{U}[-0.1, 0.1]$ is a reasonable choice here. The process is the same for all other network components.

Denoising a cosine

Optimization for 500 steps with 10 neurons, leads to the output below:

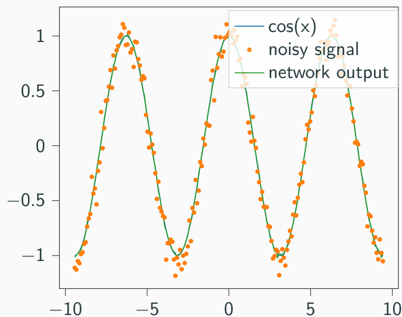


Figure: The cosine function is shown in blue, a noisy network input in orange, and a denoised network output in green.

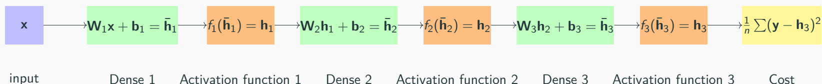
Summary

- Artificial neural networks are biologically motivated.
- Gradients make it possible to optimize arrays of neurons.
- A single array of layers of neurons can solve tasks like denoising a sine.

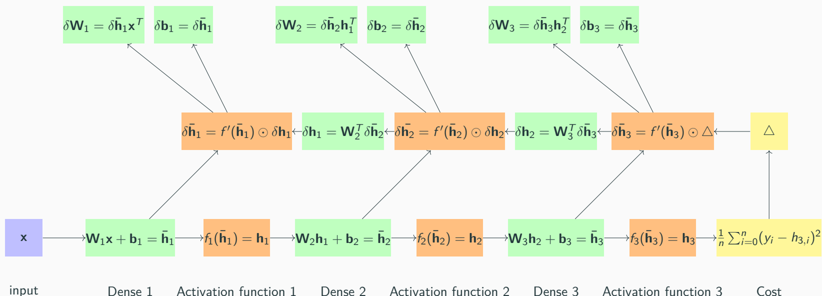
Classification with neural networks

Deep multi-layer networks

Stack dense layers and activations to create deep networks.



Backpropagation



The cross-entropy loss

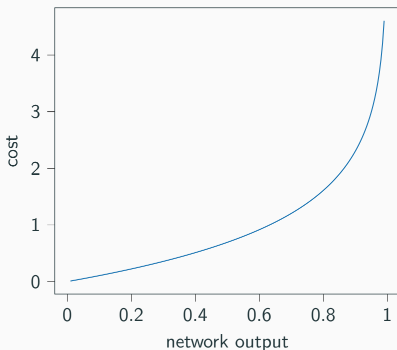
The cross-entropy loss function is defined as [Nie15; Bis06]

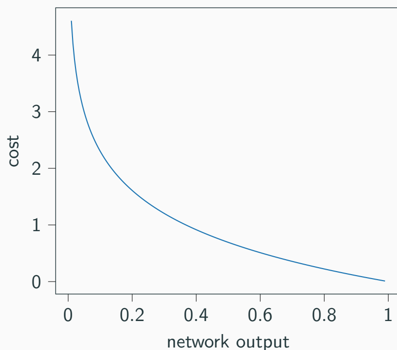
$$C_{\text{ce}}(\mathbf{y}, \mathbf{o}) = -\sum_{k}^{n_o} [(\mathbf{y}_k \ln \mathbf{o}_k) + (\mathbf{1} - \mathbf{y}_k) \ln(\mathbf{1} - \mathbf{o}_k)]. \qquad (12)$$

With n_o the number of output neurons. $\mathbf{y} \in \mathbb{R}^{n_o}$ the desired output and $\mathbf{o} \in \mathbb{R}^{n_o}$ the network output.

Understanding how cross-entropy works

To understand cross entropy let's consider the boundary cases $y = 0$ and $y = 1$.





Gradients and cross-entropy

If a sigmoidal activation function produced **o** the gradients can be computed using [Nie15; Bis06]

$$\frac{\partial C_{ce}}{\partial \mathbf{h}} = \sigma(\mathbf{o}) - \mathbf{y} = \triangle_{\text{ce}} \tag{13}$$

The MNIST-Dataset

Figure: The MNIST-dataset contains 70k images of handwritten digits.

Validation and Test data splits

- To ensure the correct operation of the systems we devise, it is paramount to hold back part of the data for validation and testing.
- Before starting to train, split off validation and test data.
- The 70k MNIST samples could, for example, be partitioned into 59k training images. 1k validation images and 10k test images.

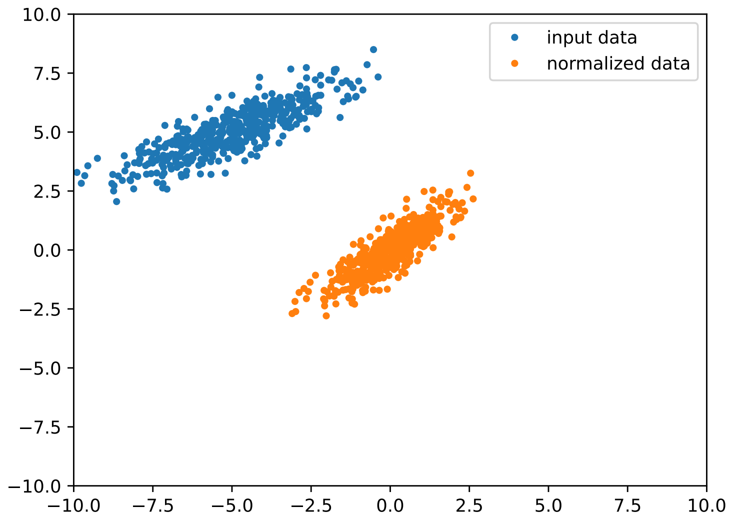
Input-preprocessing

Standard initializations and learning algorithms assume an approximately standard normal distribution of the network inputs. Consequently, we must rescale the data using,

$$x_{ij} = \frac{x_{ij} - \mu}{\sigma} \tag{14}$$

With μ and σ the training set mean and standard deviation. For all pixels, i, j up the height and width of every image. b denotes the number of data points, and n is the data dimension.

The effect of normalization



Whitening the inputs [23]

Instead of dividing by the standard deviation, rescale the centered data with the singular values of the covariance matrix.

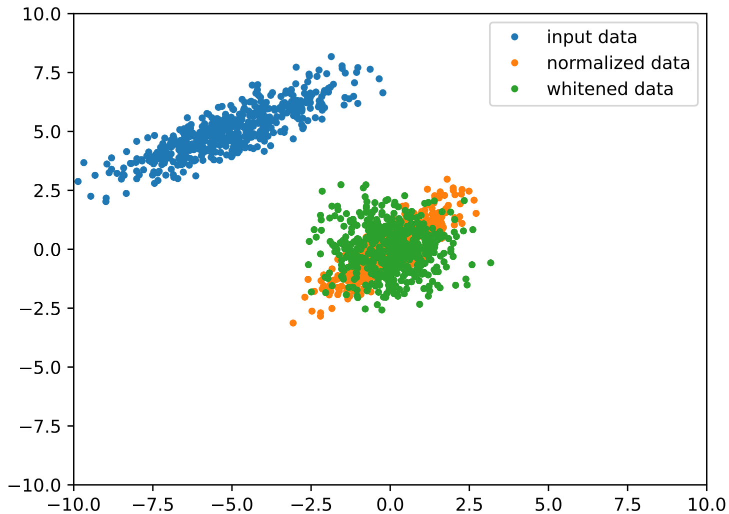
$$\mathbf{C} = \frac{1}{n}(\mathbf{x} - \mu)^T(\mathbf{x} - \mu) \tag{15}$$

With n as the total number of data points. Next we find $\mathbf{U}\mathbf{\Sigma}\mathbf{V} = \mathbf{C}$. After projecting the data via $\mathbf{p} = \mathbf{x}\mathbf{U}$. Whitening now uses the singular values of $\mathbf{C}$ to rescale the data,

$$p_{ij} = \frac{p_{ij}}{\sqrt{\sigma_j} + \epsilon} \tag{16}$$

With ϵ i.e. equal to $1e^{-8}$ for numerical stability. The operation is repeated for all pixel locations i, j in the input image.

The effect of Whitening



Label-encoding

It has proven useful to have individual output neurons produce probabilities for each class. Given integer labels $1, 2, 3, 4, \dots \in \mathbb{Z}$. One-hot encoded label vectors have a one at the label's position and zeros elsewhere. I.e.

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \vdots \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ \vdots \end{pmatrix}, \dots \tag{17}$$

for the integer label sequence above.

Batching the data

Working with individual data points is not efficient in practice. Instead, we would like to process multiple (i.e. 64) samples in parallel. Add a leading batch dimension and rely on broadcasting.

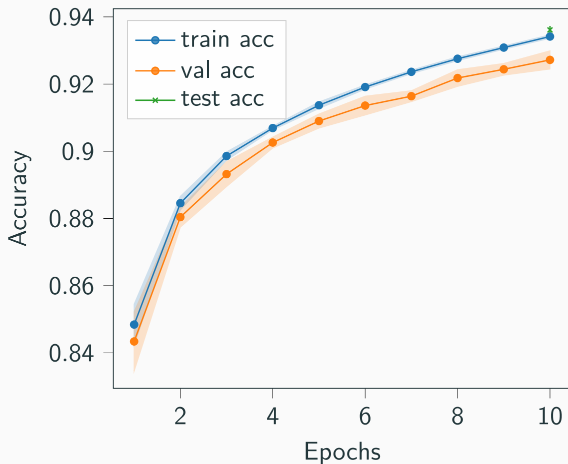
An additional mean converts the cost of a batch into a scalar. In the cross-entropy case:

$$C_{\text{ce}}(\mathbf{y}, \mathbf{o}) = -\frac{1}{n_b} \sum_{i=1}^{n_b} \sum_{k=1}^{n_o} [(\mathbf{y}_{i,k} \ln \mathbf{o}_{i,k}) + (\mathbf{1} - \mathbf{y}_{i,k}) \ln(\mathbf{1} - \mathbf{o}_{i,k})] \tag{18}$$

With n_o the number of output neurons and n_b the batch size.

MNIST-Classification

Training a three-layer dense network on mnist for five runs leads to:



Conclusion

- Preprocessing followed by forward passes, backward passes, and testing from the classic training pipeline.
- Using the pipeline, artificial neural networks enable computers to make sense of images.
- The optimization result depends on the initialization.
- The initialization depends on the seed of the pseudorandom number generator.
- *Seed-values must be recorded*, to allow reproduction.
- Share the results of multiple re-initialized runs, if possible.

References

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[23] *Data-preprocessing*. https://cs231n.github.io/neural-networks-2/. Accessed: 2023-03-26. 2023.

Literature ii

[Gre+16] Klaus Greff, Rupesh K Srivastava, Jan Koutnik, Bas R Steunebrink, and Jürgen Schmidhuber. "LSTM: A search space odyssey." In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

Network coding

Networks in Flax

Code examples for added implementation fun!! A minimal net:

```
import jax.numpy as jnp
from flax import linen as nn

class Net(nn.Module):
@nn.compact
def __call__(self, x):
    x = jnp.reshape(x,
        [x.shape[0], -1])
    x = nn.Dense(10)(x)
    x = nn.sigmoid(x)
    return x
```

Add more layers in our own experiments!!

Network initialization

Initialization requires a key for the pseudorandom number generator.

```
key = jax.random.PRNGKey(key)
net = Net()
variables = net.init(
    key, jnp.ones([batch_size]
    + list(img_data_train.shape[1:])
    + [1])
)
```

A forward pass

A forward pass through the net requires the weights and an input array.

```
output = net.apply(
  variables,
  jnp.expand_dims(img_batch, -1)
)
```

Gradient steps on weight trees

```
weights = jax.tree_util.tree_map(
    update_fun,
    weights, grads
)
```

It's useful to use a lambda function here. The function should have two arguments `weights` and `grads` and return `weights - learning_rate * grads`.
More information on lambda functions:
https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions .
More information on the tree maps:
https://jax.readthedocs.io/en/latest/_autosummary/jax.tree_util.tree_map.html