Machine Learning in Finance

Presession

Dr. Sebastian Becker

RiskLab, ETH Zurich

Toolset

We will use the following tools:

- Google Colab (colab.research.google.com)
- Numpy
- Keras
- Tensorflow
- Tensorboard
- Matplotlib

A toy example

Goal: "Learn"/Approximate a function with a fully connected/feedforward neural network

How?

1. Create a dataset
2. Build a model
3. Train the model

Why does this work?

- Feedforward neural networks are universal approximators
- Universal approximation theorems (e.g., George Cybenko in 1989 for sigmoid activation functions)

A toy example

What happens if we

- pick another optimizer?
- change the learning rate?
- change the number of hidden layers?
- change the size of the hidden layers?
- pick another activation function?
- evaluate our model outside the trained domain?
- shift our labels?
- shift the domain?
- disturb our labels?

Data and preprocessing

Data driven problems:

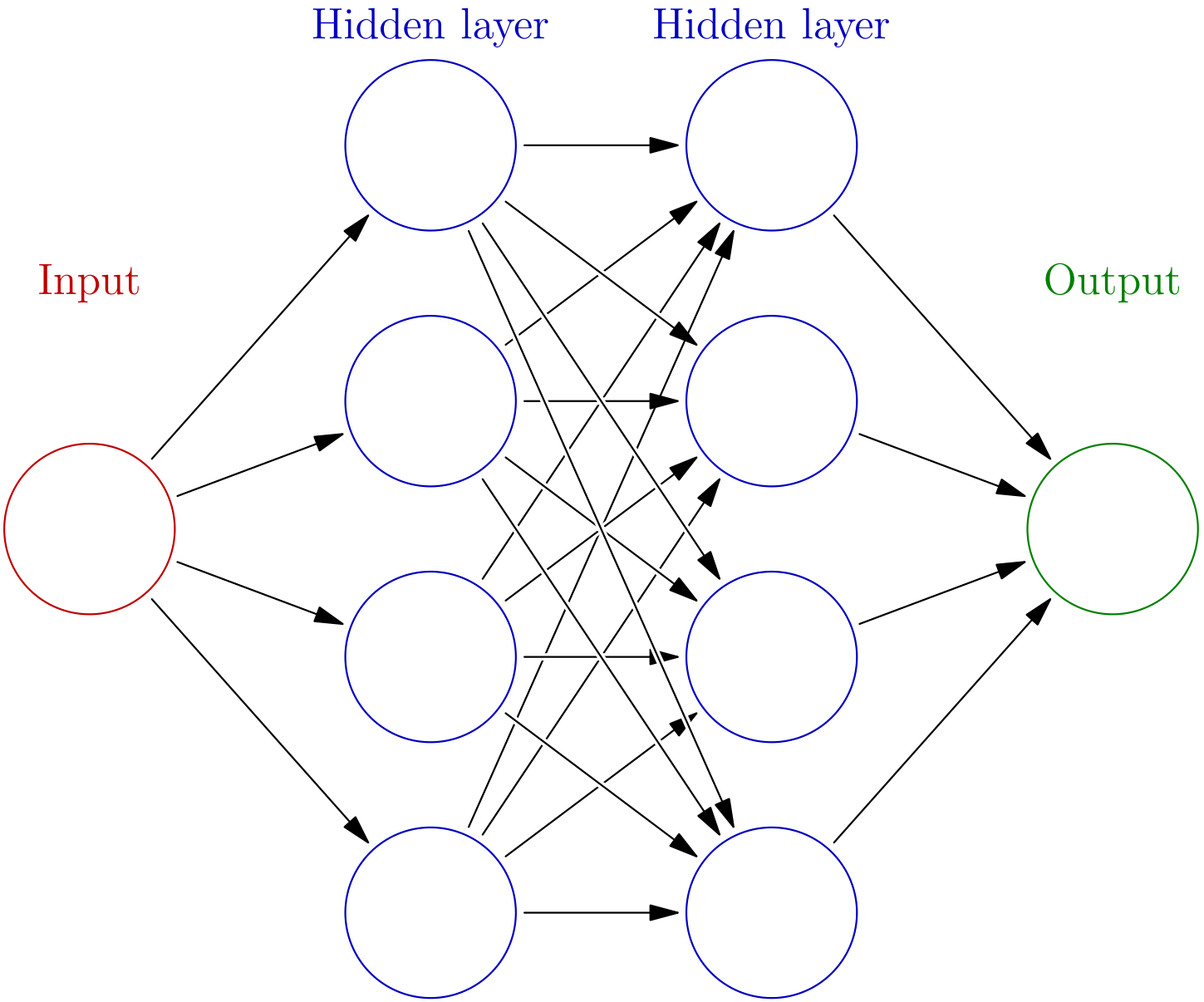
- Only a finite number of samples available
- Split the dataset into train- and testset
- Only use information from the trainingset (!)
- We may use a sample more than one time (epochs)
- Shuffle the data in each epoch

Model driven problems:

- An infinite number of samples available (?!)
- The concept of splitting is not necessary
- The concept of epochs is not necessary

In both cases we may pre-process the data

Fully connected neural networks



Fully connected neural networks

A fully connected neural network is a function $F : \mathbb{R}^d \to \mathbb{R}^{\hat{d}}$ of the form

$$F = a_I \circ \varphi_{q_{I-1}} \circ a_{I-1} \circ \cdots \circ \varphi_{q_1} \circ a_1$$

where

- $d, \hat{d}, I, q_1, q_2, \ldots, q_I$ are positive integers,
- $a_1 : \mathbb{R}^d \to \mathbb{R}^{q_1}, \ldots, a_{I-1} : \mathbb{R}^{q_{I-2}} \to \mathbb{R}^{q_{I-1}}, a_I : \mathbb{R}^{q_{I-1}} \to \mathbb{R}^{\hat{d}}$ are affine functions given by matrices $W_1 \in \mathbb{R}^{q_1 \times d}$, ..., $W_I \in R^{\hat{d} \times q_{I-1}}$ and vectors $b_1 \in \mathbb{R}^{q_1}$, ..., $b_I \in \mathbb{R}^{\hat{d}}$ such that

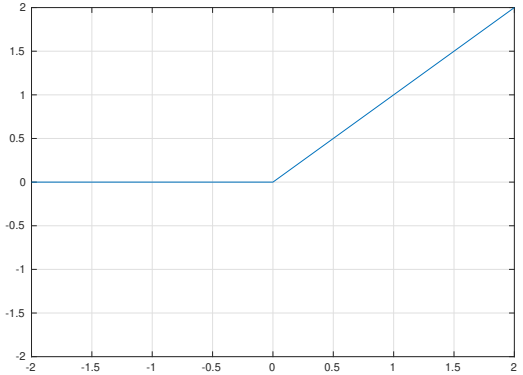
$$a_i(x) = W_i x + b_i, \qquad i = 1, \ldots I,$$

- for every $j \in \mathbb{N}$, $\varphi_j : \mathbb{R}^j \to \mathbb{R}^j$ is a component-wise activation function.

Activation functions

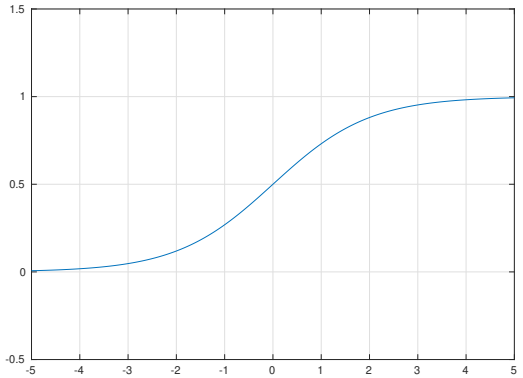
RELU:

- $f(x) = \max(0, x) \in [0, \infty)$
- $f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \\ \text{undefined} & x = 0 \end{cases}$



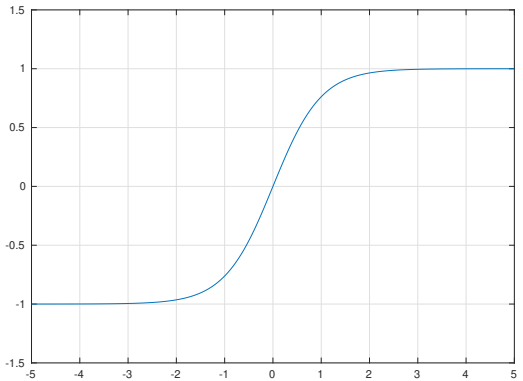
Logistic function/"sigmoid"/soft step:

- $f(x) = \frac{1}{1+e^{-x}} \in (0, 1)$
- $f'(x) = f(x)(1 - f(x))$



tanh:

- $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in (-1, 1)$
- $f'(x) = 1 - f(x)^2$



Initialization

Our weights/trainable variables need to be initialized!

Matrices:

- random normal
- random uniform
- Xavier/Glorot normal with variance $\sigma = \sqrt{\frac{2}{d_{in}+d_{out}}}$
- Xavier/Glorot uniform in $[-r, r]$ with $r = \sqrt{\frac{6}{d_{in}+d_{out}}}$

Idea behind Glorot/Xavier initialization is to avoid vanishing or exploding gradients! Variance of the output of each layer should equal the variance of the inputs.

Bias: mostly zero

Training

Stochastic gradient descent with mini-batches:

1. Initialize the weights
2. Pick a batch of training samples
3. Calculate the gradients of the loss function with respect to the weights (forward pass)
4. Update the weights as $w_{new} = w_{old} - \text{learning rate} * \text{gradient}$ (backward pass)
5. repeat from step 2
6. (evaluate some metric on the testset)

Thank you for your attention!

Appendix

Batch normalization (Ioffe & Szegedy 2015):

- Avoid unstable gradients
- Reduces the effect of bad initialization
- Allow for higher learning rates and hence faster training

Batch normalization is applied after the affine transormation but before activation.

How:

- Calculate mean and variance of a mini-batch
- Normalize by $\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$
- Scale and shift by $y_i = \gamma \hat{x}_i + \beta$