Data Analysis II: Neural Networks

Tino Cestonaro

Recap and what this session is about

- You learnt how to read data, preprocess data, describe data using visualizations and summary statistics, and apply models like Decision Trees
- In recent years, **artifical neural networks (ANNs)** have become particularly popular for supervised tasks, e.g., image recognition (e.g. facial recognition or X ray detection) and time series forecasting (weather, traffic or stock prices).
- **Advantages:** ANN learns complex, non-linear functions in a flexible manner, does not impose any restrictions on input variables
- **Disadvantages:** large amounts of data needed to perform well, computationally intense, limited insights

Neural networks - motivation

- ANNs are not a new concept: Foundations were laid by **McCulloch and Pitts (1943)**
- Today: technology companies such as Facebook, Google or Netflix use ANNs to, automatically tag videos or news, provide (movie) recommendations, etc.
- ANNs are inspired by animal/human brains that have **billions of cells called neurons that process information** via electric signals
- A **neuron receives an electric signal and can accept or reject it**, depending on the strength of the signal
- In case of acceptance, it process the signal in the neuron call body and **pass it to the next neuron** that accepts or rejects the signal

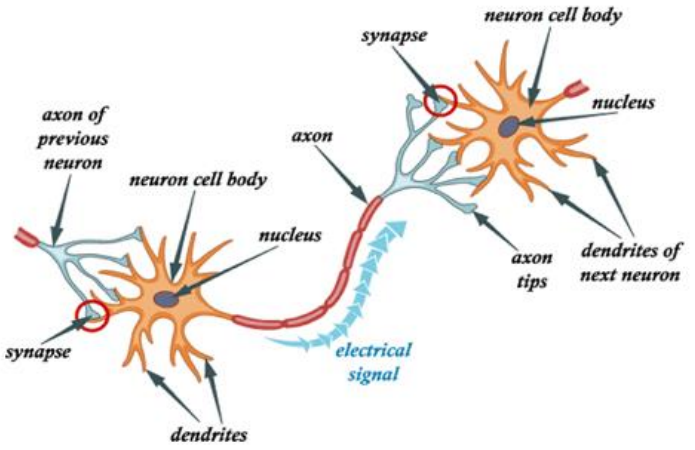


Fig 1: Biological Neurons
Source: https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207

Feed-forward neural network

- One of the simplest types of ANNs is the **feed-forward neural network,** where the information moves only forward, from input nodes through hidden nodes and ends in the output nodes
- Note that there are **many other types of ANNs** than feed-forward neural networks such as recurrent neural networks
- In general, there are two major types of feed-forward neural networks:
  - **Single layer perceptron** – does not contain any hidden layer and can only learn linear functions
  - **Multi layer perceptron** – contains one or more hidden layers and can learn non-linear functions

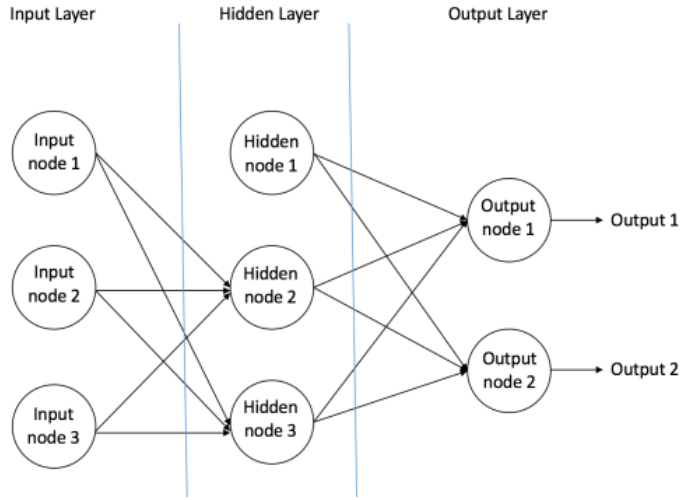
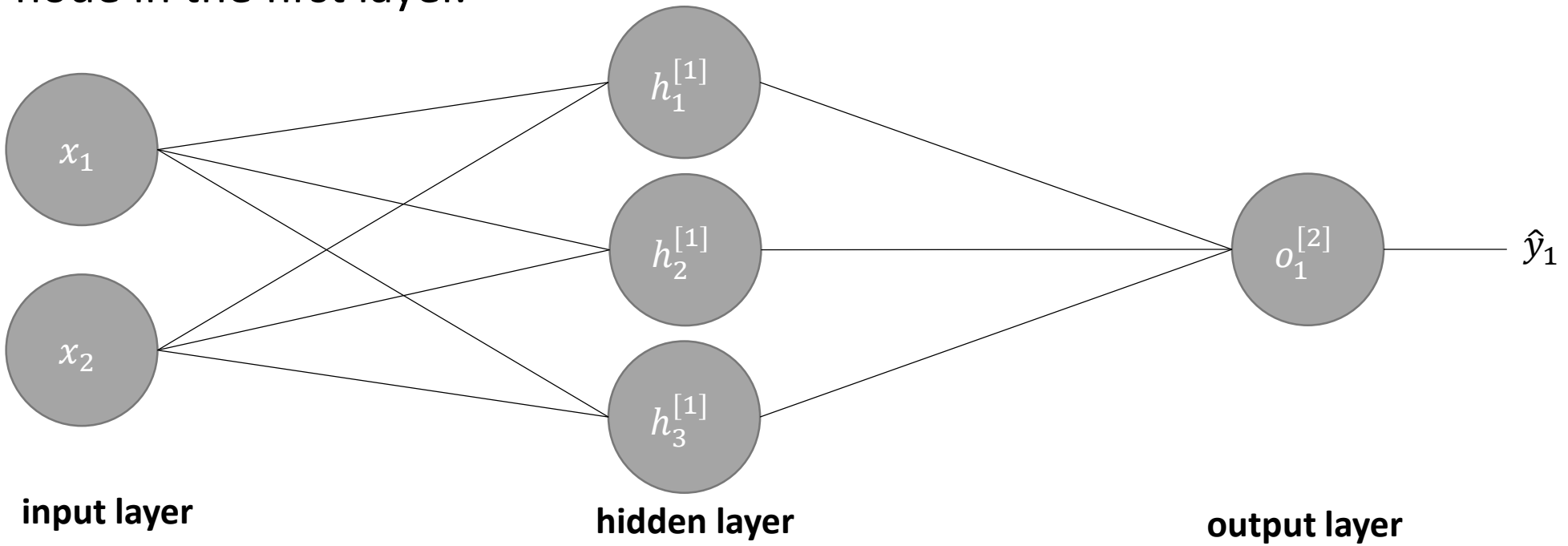


Fig 2: Structure of a multi layer perceptron with one hidden layer
Source: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

Multi layer perceptron - Structure

- Here is a basic example of a **multi layer perceptron** with 2 input features, 1 hidden layer with 3 nodes, and 1 output node
- Notation: The subscript i indicates the sequence of nodes within a single layer, while the superscript l indicates the number of the layer. E.g., $h_2^{[1]}$ is the second node in the first layer.



Multi layer perceptron – Forward pass (1)

- Given a vector of input features $\vec{x} = \{x_1, x_2\}$, the information is passed through the network as follows:

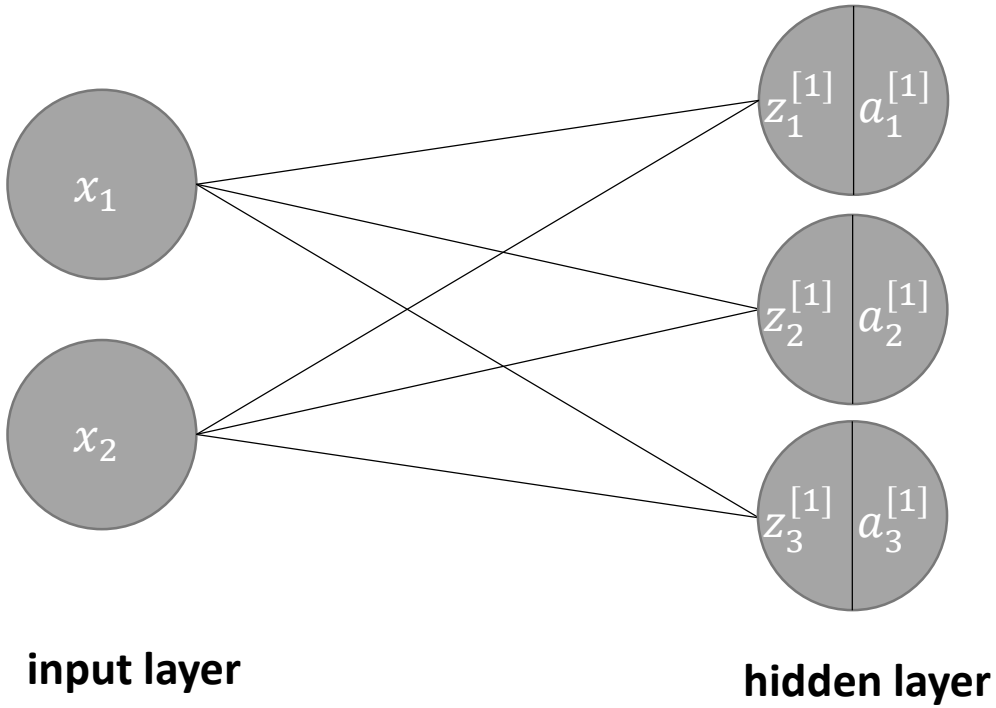
- Each node i of the first hidden layer obtains the weighted sum of the outputs of the previous nodes as input:

$$\boldsymbol{z_i^{[1]} = b_i^{[1]} + w_{i1}^{[1]} \cdot x_1 + w_{i2}^{[1]} \cdot x_2}$$

- The weighted sum is used as input for the node's activation function σ :

$$\boldsymbol{a_i^{[1]} = \sigma\left(z_i^{[1]}\right)}$$

- Output $\boldsymbol{a_i^{[1]}}$ is the input for subsequent nodes.



Multi layer perceptron – Forward pass (1)

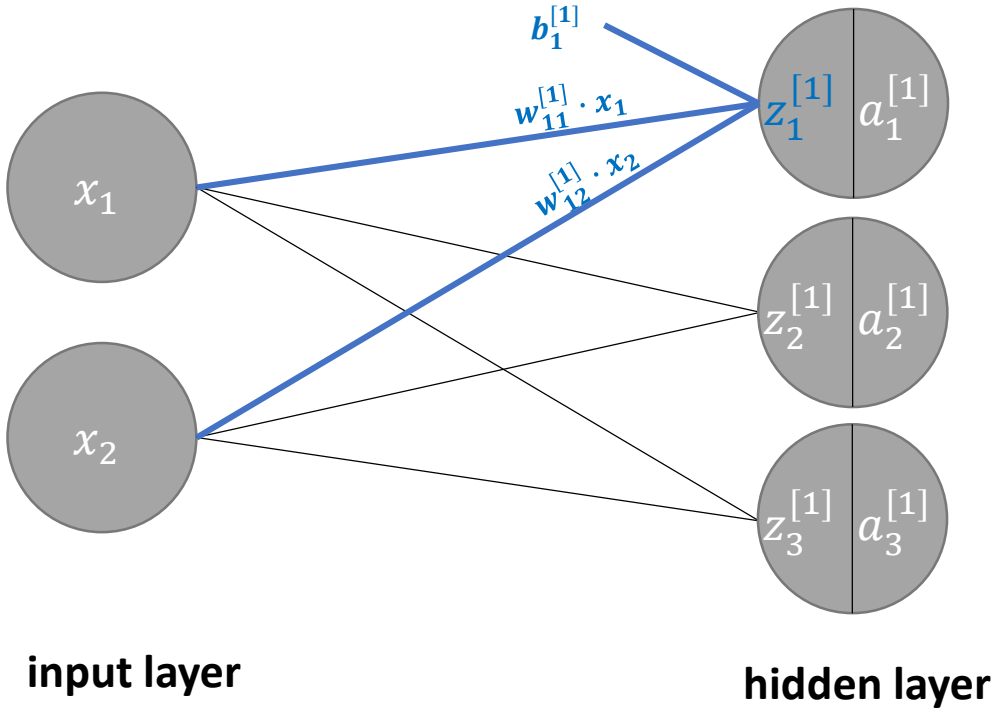
- Given a vector of input features $\vec{x} = \{x_1, x_2\}$, the information is passed through the network as follows:
- Each node i of the first hidden layer obtains the weighted sum of the outputs of the previous nodes as input:

$$\boldsymbol{z_i^{[1]} = b_i^{[1]} + w_{i1}^{[1]} \cdot x_1 + w_{i2}^{[1]} \cdot x_2}$$

- The weighted sum is used as input for the node's activation function σ :

$$\boldsymbol{a_i^{[1]} = \sigma\left(z_i^{[1]}\right)}$$

- Output $\boldsymbol{a_i^{[1]}}$ is the input for subsequent nodes.



Multi layer perceptron – Forward pass (2)

- **Activation function**: is a differentiable, non-linear function that connects the input layer (hidden layer) with another succeeding hidden layer to determine whether a **node is active or not**. Two examples are:

- **Sigmoid**

$$a(z) = \frac{1}{1 + e^{-z}}$$

- Yields a non-linear output (from a linear input)
- Easy to differentiate for gradient computation

- **ReLU** (rectified linear unit)

$$a(z) = max(0, z)$$

- Yields a non-linear output (from a linear input), but close to linear due to its piece-wise definition
- Easy to differentiate for gradient computation

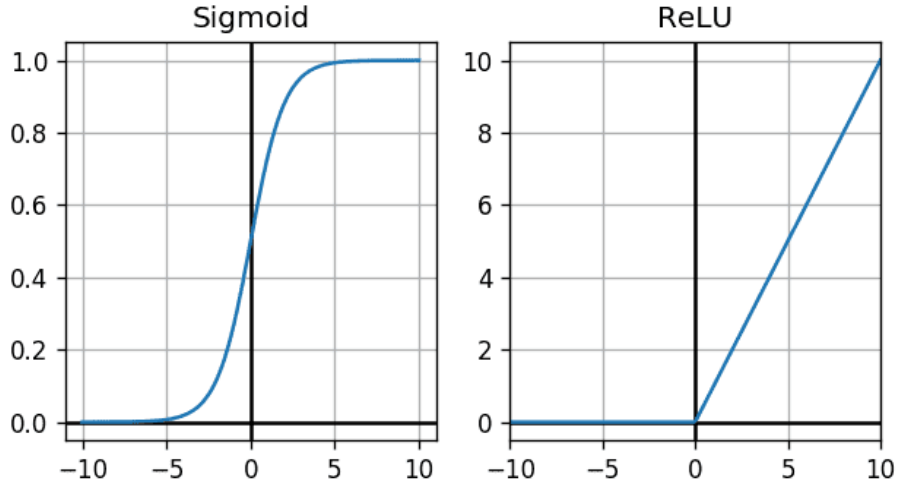


Fig 3: Sigmoid and ReLU function
Source: https://i.stack.imgur.com/gLrAJ.png

Multi layer perceptron – Forward pass (3)

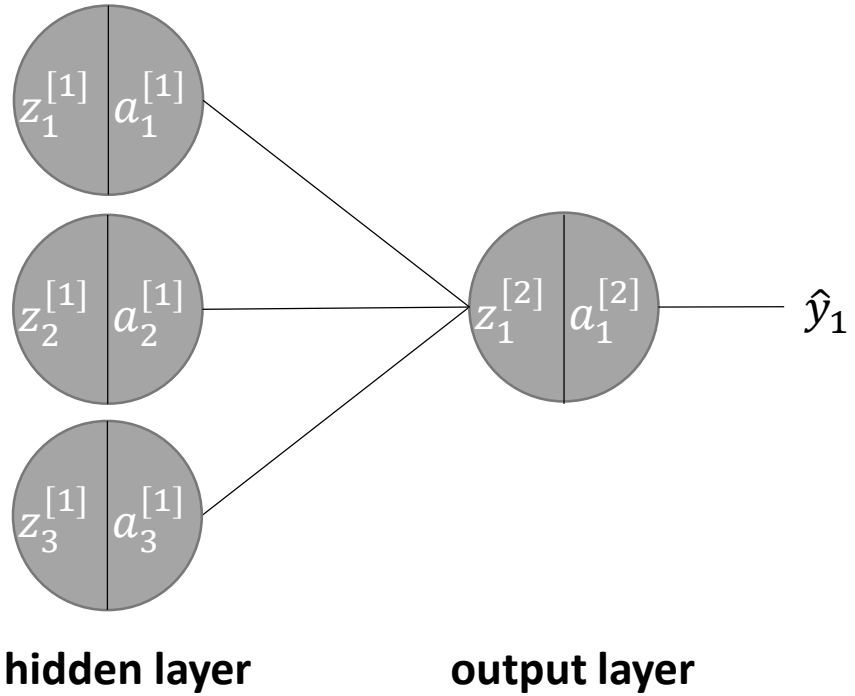
- Each node i of the output layer obtains the weighted sum of the outputs of the previous nodes as input:

$$\boldsymbol{z_i^{[2]} = b_i^{[2]} + w_{i1}^{[2]} \cdot a_1^{[1]} + w_{i2}^{[2]} \cdot a_2^{[1]} + w_{i3}^{[2]} \cdot a_3^{[1]}}$$

- Use the sum as input for output node's activation function σ :

$$\boldsymbol{a_i^{[2]} = \sigma\left(z_i^{[2]}\right)}$$

- Output $\boldsymbol{a_1^{[2]}}$ is the final output in our net with one hidden layer, i.e. here: $\boldsymbol{a_1^{[2]} = \hat{y}_1}$



Multi layer perceptron – Forward pass (3)

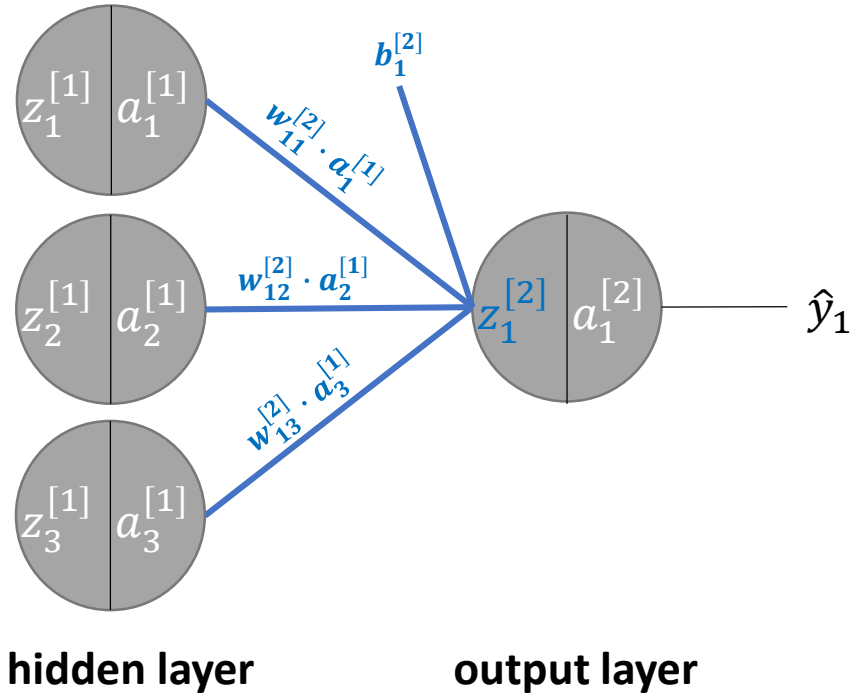
- Each node i of the output layer obtains the weighted sum of the outputs of the previous nodes as input:

$$\boldsymbol{z_i^{[2]} = b_i^{[2]} + w_{i1}^{[2]} \cdot a_1^{[1]} + w_{i2}^{[2]} \cdot a_2^{[1]} + w_{i3}^{[2]} \cdot a_3^{[1]}}$$

- Use the sum as input for output node's activation function σ :

$$\boldsymbol{a_i^{[2]} = \sigma\left(z_i^{[2]}\right)}$$

- Output $\boldsymbol{a_1^{[2]}}$ is the final output in our net with one hidden layer, i.e. here: $\boldsymbol{a_1^{[2]} = \hat{y}_1}$



Activation function for output

- The functional form of the activation function σ in the output layer units depends on the actual problem that the data scientist tackles. Here are some common problems:

Problem	Activation function	Output
Binary classification (e.g. Yes/No)	**Sigmoid** $$\boldsymbol{a_1(z_1) = \frac{1}{1 + e^{-z_1}}}$$	Output layer has one node with sigmoid activation function. $\boldsymbol{a_1(z_1)}$ can be interpreted as probability that $\mathbf{y = 1}$.
Multi-class classification (e.g. buy/hold/sell)	**Softmax** $$\boldsymbol{a_i(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}}$$	Output layer has i nodes, where $i =$ number of possible classes K. Since $\sum_{i=1}^{K} \boldsymbol{a_i} = 1$, $\boldsymbol{a_i(z_i)}$ can be interpreted as probability that $\boldsymbol{\hat{y}_i = y}$.
Regression (e.g. predicting stock price)	**Relu** (for non-negative values) $$\boldsymbol{a_i(z_i) = max(0, z_i)}$$	Output layer has i nodes, where $i =$ number of predicted values. $\boldsymbol{a_i(z_i)}$ refers to the predicted numerical value of $\boldsymbol{y_i}$.

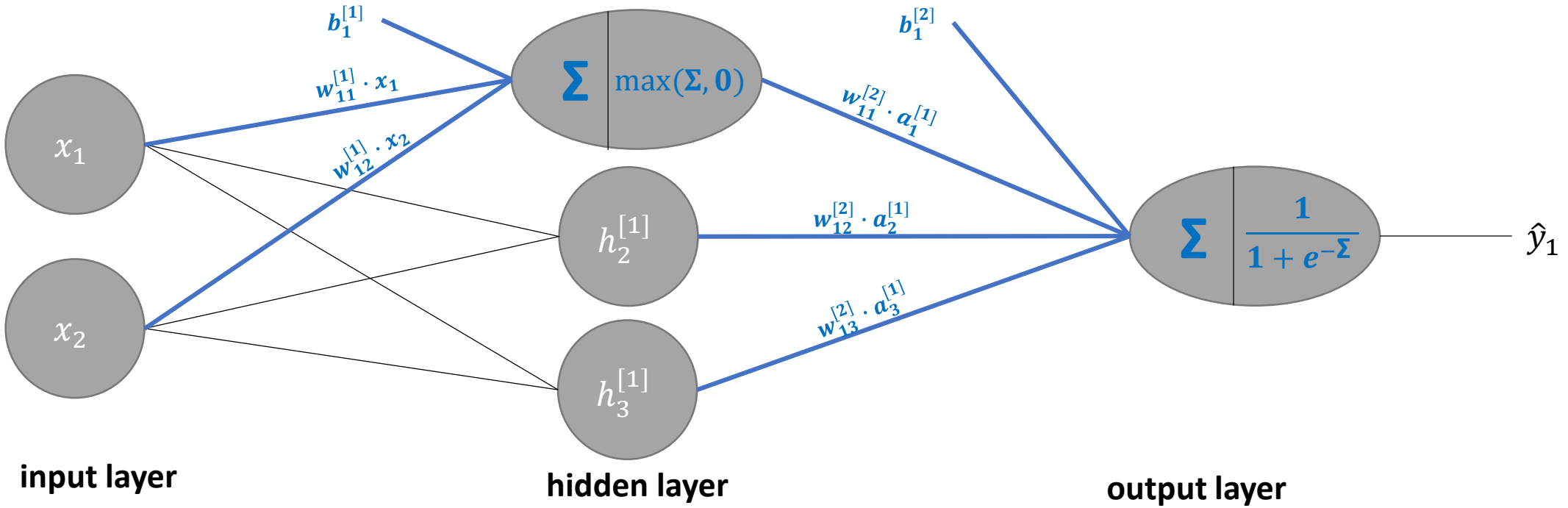
Multi layer perceptron – Backward propagation (1)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

Multi layer perceptron – Backward propagation (1)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

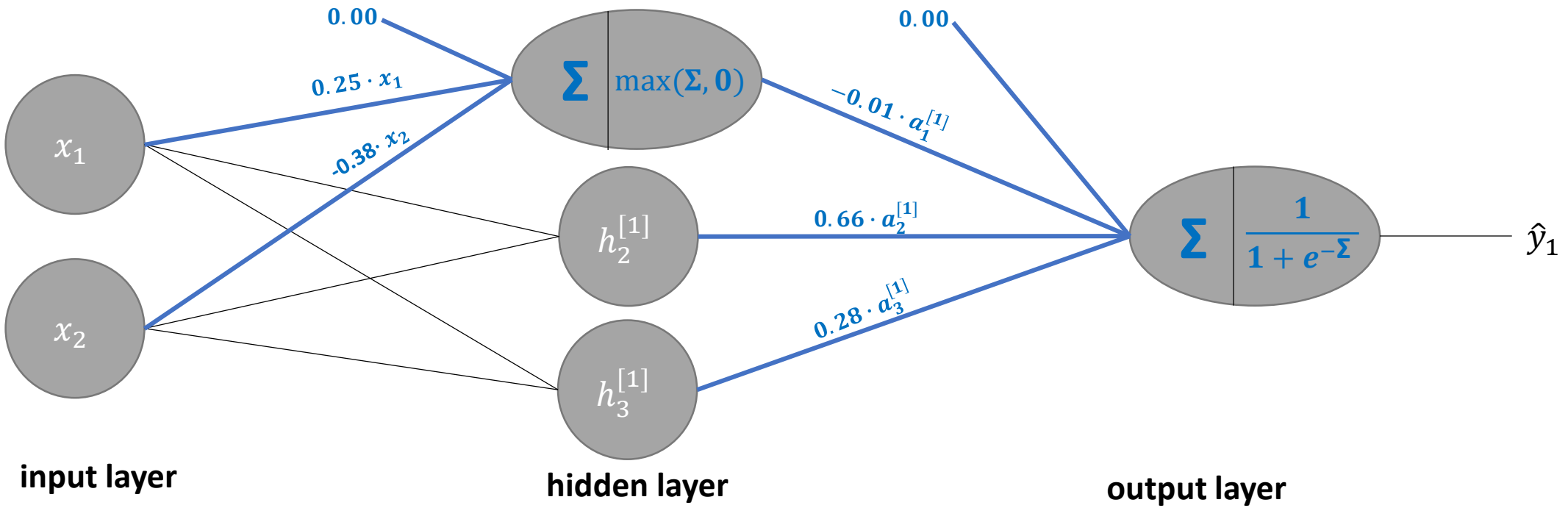
1. We **initialize all weights and biases** in our network randomly



Multi layer perceptron – Backward propagation (1)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

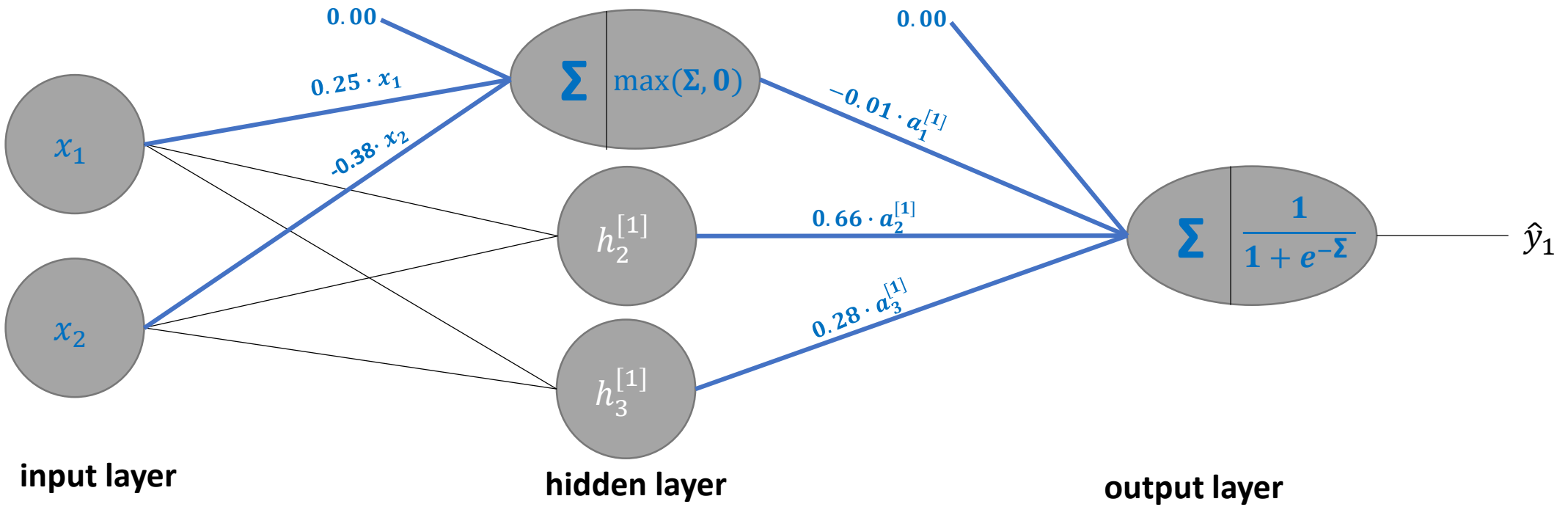
1. We **initialize all weights and biases** in our network randomly



Multi layer perceptron – Backward propagation (2)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

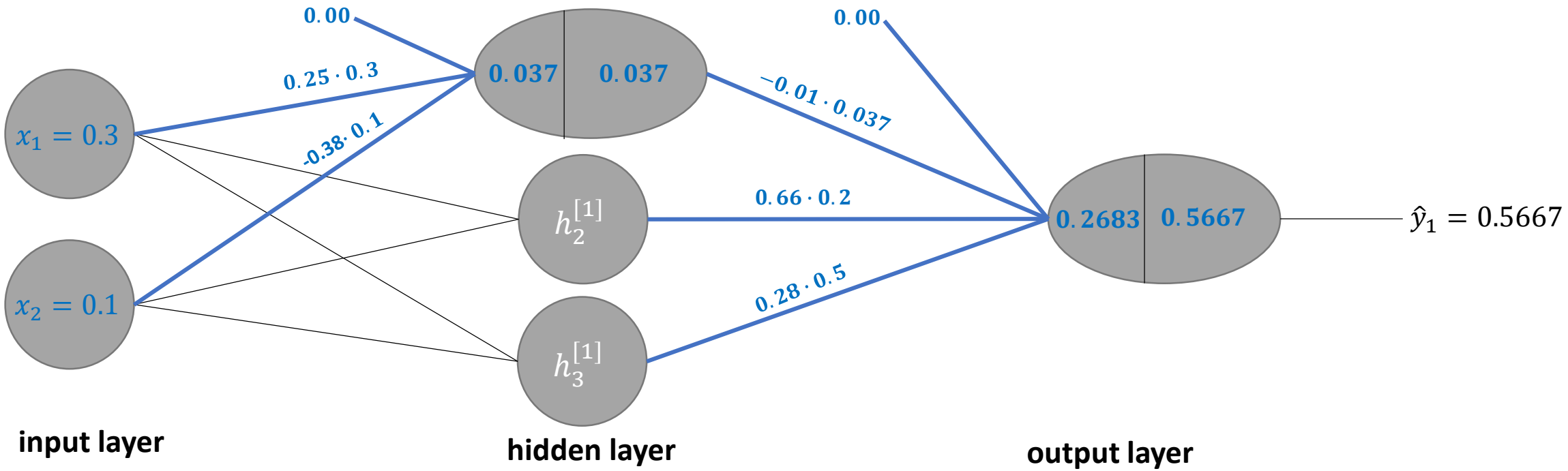
2. We **propagate** the first observation $\left\{x_1^{(1)}, x_2^{(1)}\right\}$ **through our network** to get $\hat{y}_1^{(1)}$



Multi layer perceptron – Backward propagation (2)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

2. We **propagate** the first observation $\left\{x_1^{(1)}, x_2^{(1)}\right\}$ **through our network** to get $\hat{y}_1^{(1)}$



Multi layer perceptron – Backward propagation (3)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

3. We **compare our prediction $\boldsymbol{\hat{y}_1^{(1)}}$ with true value $\boldsymbol{y_1^{(1)}}$** to evaluate the quality of our forecast. Therefore, we use a loss function. Again, the **loss function depends on the** tackled **problem**, here **binary classification**.
   - **Binary cross entropy** loss function for the $\boldsymbol{m}$-th observation:
   - $$L\left(y_1^{(m)}, \hat{y}_1^{(m)}\right) = -\left[y_1^{(m)} \cdot log\left(\hat{y}_1^{(m)}\right) + \left(1 - y_1^{(m)}\right) \cdot log\left(1 - \hat{y}_1^{(m)}\right)\right]$$

Multi layer perceptron – Backward propagation (3)

- But how do we obtain the weights $\boldsymbol{w_{ij}^{[l]}}$ and biases $\boldsymbol{b_i^{[l]}}$?

3. We **compare our prediction $\hat{\boldsymbol{y}}_{\boldsymbol{1}}^{(\boldsymbol{1})}$ with true value $\boldsymbol{y}_{\boldsymbol{1}}^{(\boldsymbol{1})}$** to evaluate the quality of our forecast. Therefore, we use a loss function. Again, the **loss function depends on the** tackled **problem**, here **binary classification**.
   - **Binary cross entropy** loss function for the $\boldsymbol{m}$-th observation:
   - $L\left(y_1^{(m)}, \hat{y}_1^{(m)}\right) = -\left[y_1^{(m)} \cdot log\left(\hat{y}_1^{(m)}\right) + \left(1 - y_1^{(m)}\right) \cdot log\left(1 - \hat{y}_1^{(m)}\right)\right]$
   - In our example: $\hat{y}_1^{(m)} = 0.5667$, we assume $y_1^{(m)} = 0$, therefore:

$$L(0, 0.5667) = 0.8363$$

   - Idea: find a set of **weights and biases such that the loss function is minimized**, i.e., we find a set of parameters that optimally fit the observed data (potentially subject to constraints → future course)

Multi layer perceptron – Backward propagation (4)

- Backward propagation is an algorithm that **calculates the gradient** of a loss function with respect to its (trainable) input parameters
- I.e. calculate $\frac{dL}{dw_{ij}^{[l]}}$ and $\frac{dL}{db_{i}^{[l]}}$ for all weights & biases in our network
- Afterwards, use the partial derivatives to update the individual weight according to an optimization algorithm:

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha \cdot \frac{dL}{dw_{ij}^{[l]}} \qquad b_{i}^{[l]} = b_{i}^{[l]} - \alpha \cdot \frac{dL}{db_{i}^{[l]}}$$

- α is the learning rate and determines how quickly the parameters adapt

Working on a full data set

- All explanations and formulas before assumed that we propagate **one observation** $\boldsymbol{m}$ forward and backward through our network
- Usually, we propagate a set of observations through our network. The number of observations we propagate simultaneously through the network is called **mini batch size**. Mini batch size can be between 1 and $\boldsymbol{M}$, where $\boldsymbol{M}$ is the number of all observations in our data set.
- **Number of epochs**: number of times the full dataset (all observations) are propagated through the network and used for estimating the model parameters $\boldsymbol{w}_{\boldsymbol{ij}}^{[\boldsymbol{l}]}$ and $\boldsymbol{b}_{\boldsymbol{i}}^{[\boldsymbol{l}]}$

Hyperparameters (not complete) – actually the hardest part in applied Deep Learning

- Number of layers
- Number of nodes in each layer
- Activation functions
- Optimizer (the way of adapting parameters to minimize loss function)
- Mini batch size
- Weight initialization
- Techniques to prevent overfitting (not covered today):
    - Regularization
    - Dropout layers
    - …

Our today's goal with neural networks

- Given data on a direct marketing campaign by a bank we want to predict whether the client subscribes a term deposit.
- I.e. we have **a binary classification task,** where

$$\boldsymbol{y = 1}\ \textit{if client subscribes},\qquad \boldsymbol{y = 0}\ \textit{otherwise}$$

- Our **input features $\boldsymbol{X}$** are different **numerical and categorical variables** such as age, number of contacts, phone call duration, educational status, etc.
- The goal is to train a neural network on our data that approximates $\boldsymbol{y = f(X)}$.
- To solve our task, we rely on the deep learning library **keras**.

Common preprocessing steps in Deep Learning

- In each data science project, preprocessing steps are required. Our data set requires two common actions:
- **Scale input features**
  - Recommended in each deep learning project!
  - Normalize the absolute values across all inputs to ensure that they have similar effects on loss function → stabilize training process
- **Transform categorical to numerical variables**
  - We use OneHotEncoding for transformation, i.e., creating $K-1$ dummy variables where K refers to the number of unique values
  - Example in our data set: $martial$: $\{married, single, divorced, unkown\}$

before	after transformation		
martial	**martial. married**	**martial. single**	**martial. divorced**
married	1	0	0
single	0	1	0
divorced	0	0	1
unknown	0	0	0

Tasks (together)

- Create a feed forward neural network with
  - 1 input layer with #nodes is the number of features (let's see!)
  - 1 hidden layer with 5 nodes
  - 1 output layer with 1 node and sigmoid activation to solve binary classification task
- Train network with
  - Batch size 32
  - Five epochs

Tasks (individually)

- Hyperparameter optimization
  - Create a loop to run the model for different number of number of nodes [5, 50, 500) and mini batch sizes [1, 32, 128] and report the respective accuracies for a test set.
  - If you can, try to test all hyperparameter combinations, i.e. [(5, 1), (5, 32), (5, 128), (50, 1), (50, 32), … , (500, 128)].

- Evaluation
  - A higher number of nodes means that the ANN can mimic more complex functions, but might also be prone to overfit. Does the model fit using the test data improve with an increasing number of nodes? If yes, is the effect large?
  - The effect of changing mini batch size depends on the underlying data. What do you observe for different specifications? Do you have explanations for your findings?

References

- Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press, available at https://www.deeplearningbook.org/
- Keras: The Python Deep Learning library, https://keras.io/
- McCulloch, W., & Pitts, W. (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133.