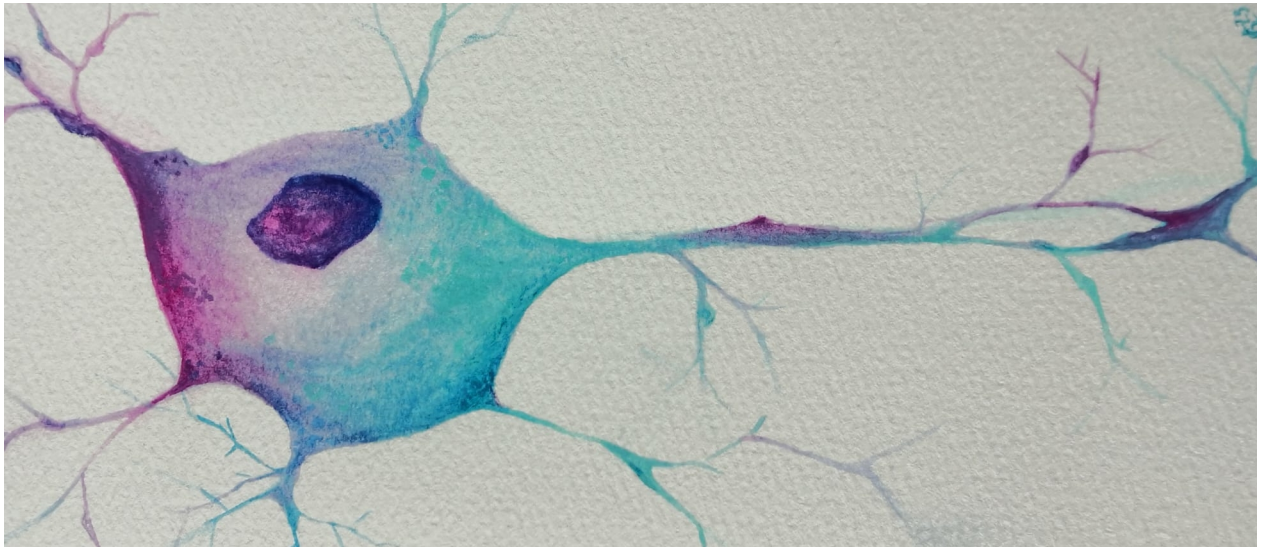
# Introduction to Data Analysis and Machine Learning in Physics: 3. Machine Learning Basics, Multivariate Analysis

Jörg Marks

Studierendentage, 7-11 April 2025



# Books / Material

- Books on Machine Learning
  - Ian Goodfellow and Yoshua Bengio and Aaron Courville, MIT Press Book Deep Learning
    free online http://www.deeplearningbook.org/
  - Aurelien Geron, Hands-On Machine Learning with Scikit-Learn and TensorFlow
  - Francois Chollet, Deep Learning with Python
  - Deep Learning for Physics Research, Martin Erdmann, Jonas Glombitza, Gregor Kasieczka, Uwe Klemradt (available via Kindle)

- Papers
  - A high-bias, low-variance introduction to Machine Learning for physicists
    https://arxiv.org/abs/1803.08823
  - Machine learning and the physical sciences
    https://arxiv.org/abs/1903.10563

- Credits: The talk contains material of a tutorial at the 3rd IML workshop 2019 at CERN by Yannik Rath, Train-the-Trainer Workshop "Deep Learning Basics" in February 2023 and “Introduction to Data Analysis and Machine Learning in Physics” by Jörg Marks and Klaus Reygers in April 2023

# Introduction

- ChatGPTs opinion on the relation of Artificial Intelligence (AI) Machine Learning (ML) and Deep Learning

AI (Artificial Intelligence) is a broad field that encompasses a range of techniques and approaches for developing intelligent systems. Machine Learning (ML) is a subset of AI that focuses on teaching machines to learn from data and make predictions or decisions without being explicitly programmed. Deep Learning (DL) is a subset of ML that uses artificial neural networks with many layers to learn increasingly abstract features from data, and has achieved significant breakthroughs in areas such as computer vision, natural language processing, and robotics. Therefore, DL is a specific type of ML, and ML is a specific type of AI.

Question: Where are you in this hirarchy?

As an AI language model, I am built using various ML techniques, such as deep learning and natural language processing. Therefore, I am a subset of AI, specifically falling under the category of ML-based AI.

- Stable Diffusion
  is a latent text-to-image diffusion model
  capable of generating photo-realistic
  Images given any text input

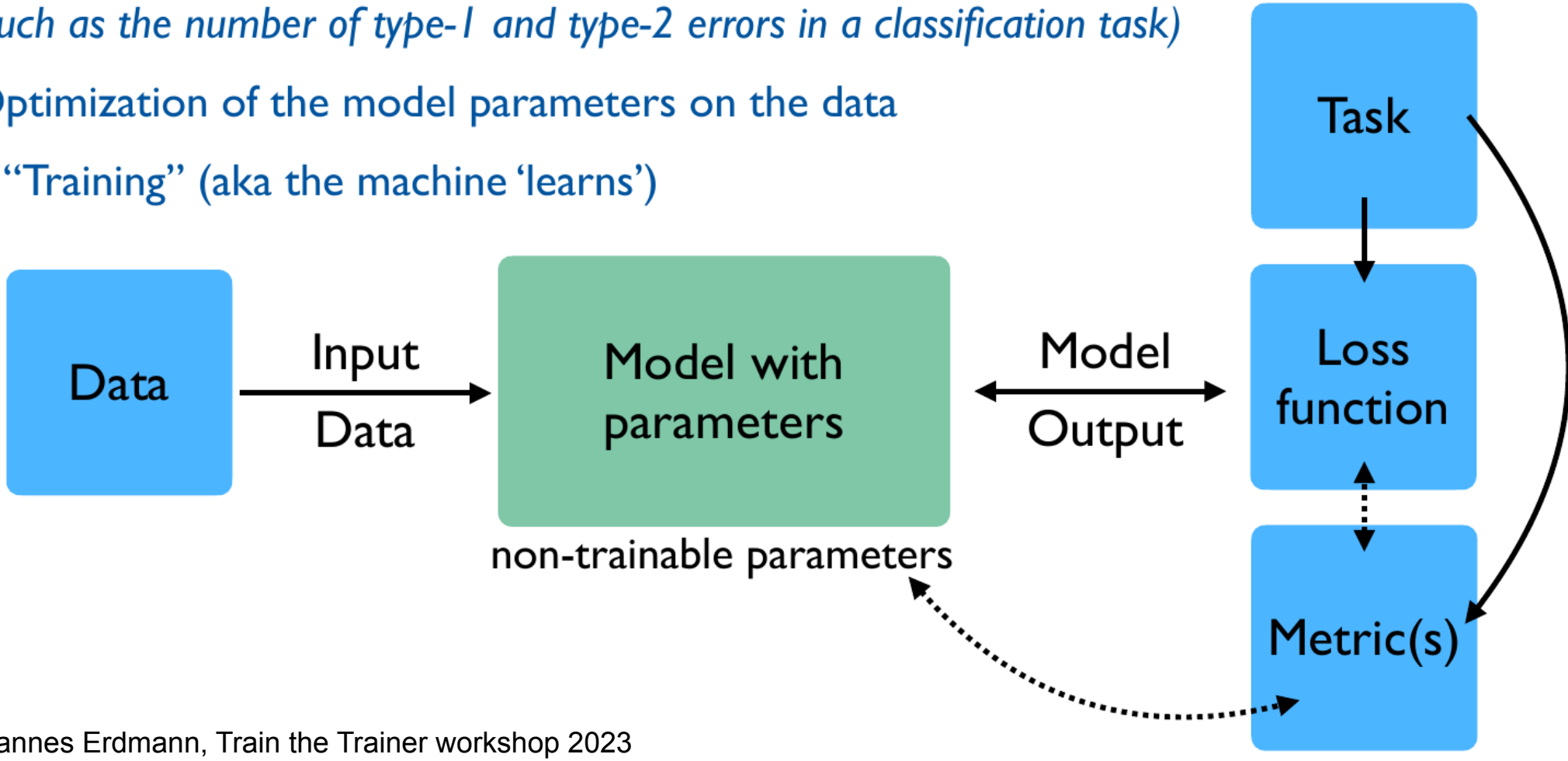
# Introduction Machine Learning

- Data + Task + "a mathematical formulation of what means 'better' " *

  *(such as the number of type-1 and type-2 errors in a classification task)*

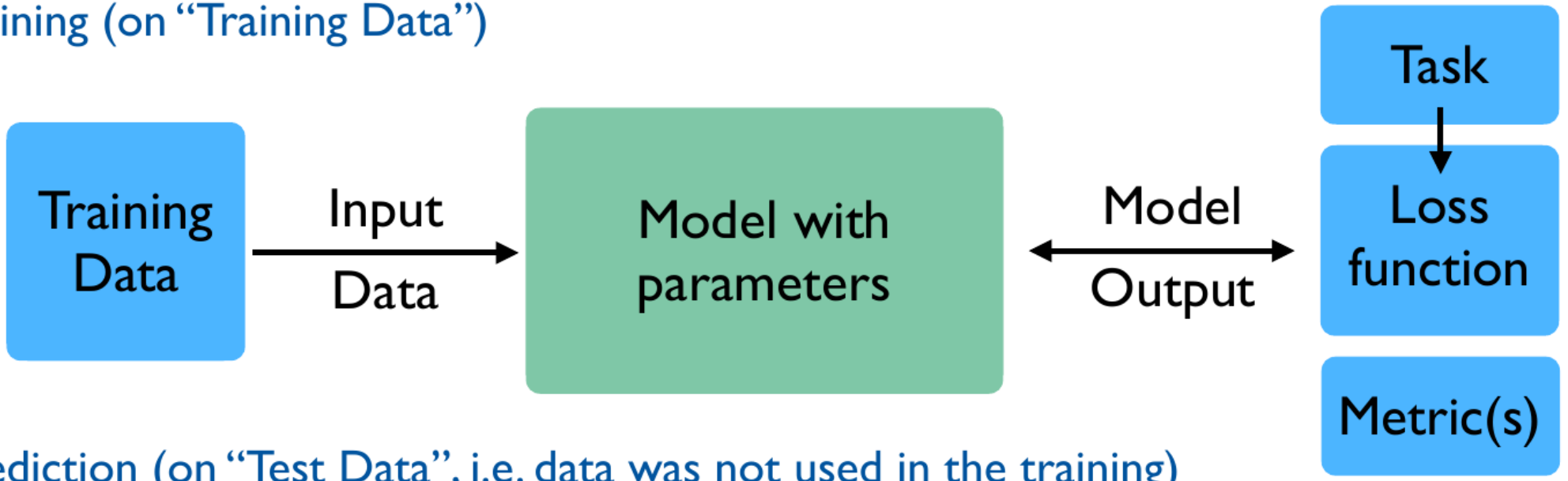
- Optimization of the model parameters on the data

  = "Training" (aka the machine 'learns')

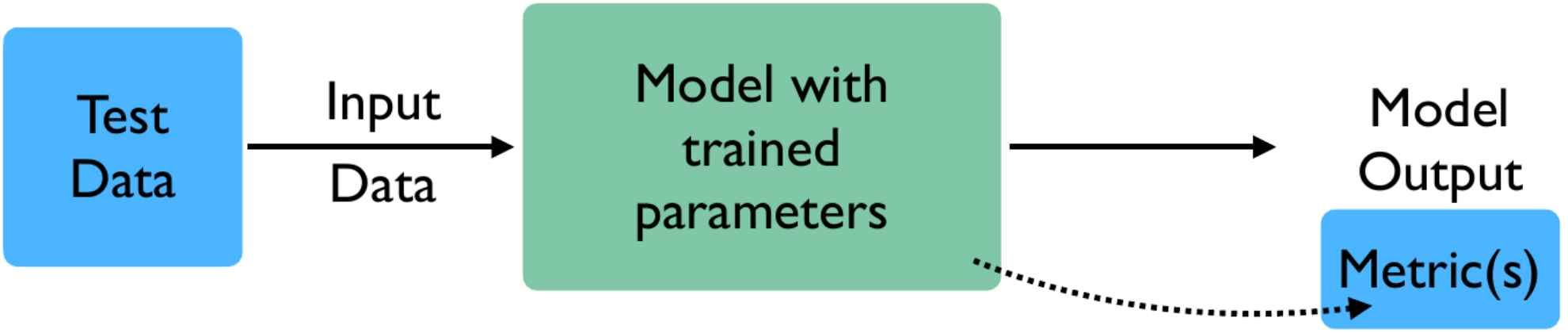


# Introduction Machine Learning

- Training (on "Training Data")




- Prediction (on "Test Data", i.e. data was not used in the training)



# Categories of Machine Learning

- Machine Learning types are closely related to data available for the task

  - Supervised Learning

    Here the model is trained on a labeled dataset, each data has an associated target variable or label. The goal of the model is to learn a mapping between the input features and the output label → make accurate predictions on new, unseen data.

  - Unsupervised Learning

    The model is trained on unlabeled data.The goal of the model is to find patterns and structure in the data, such as clusters, associations, or dependencies.

  - Reinforcement Learning

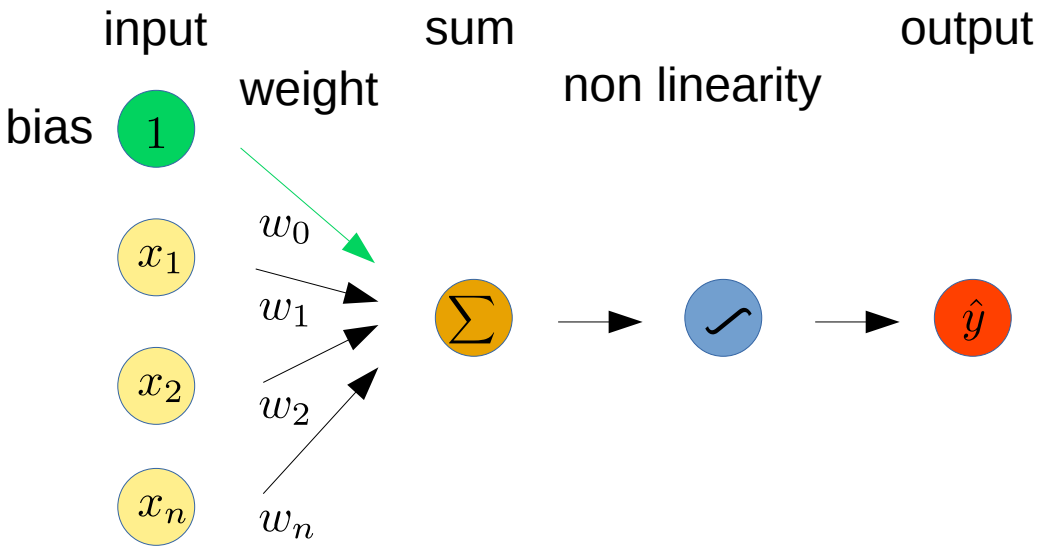
    The model learns to make decisions based on feedback it receives. The goal is to maximize a reward signal (a score or a profit)

  - Transfer Learning

    A technique in machine learning where a pre-trained model is used as a starting point for a new task, rather than training a new model from scratch. Typically a pre-trained model has already learned useful features from a large dataset, that is then used for a new task with only a small amount of additional training data.

# Multilayer Perceptron

- Forward propagating perceptron




Activation function

$$\hat{y} = \theta \cdot (w_0 + \vec{x}^T \vec{W})$$

The bias term shifts the activation independent of the input.

Idea: minimize a cost function to determine the weights and bias of the perceptron by comparing the predicted output and the true output.

→ obtain a relation between the multidimensional input parameter space and some output

The input of the perceptron is passed through a non linear activation function, which determines the output of the perceptron. There is only a contribution to the perceptron if a threshold value is reached.

# Multilayer Perceptron

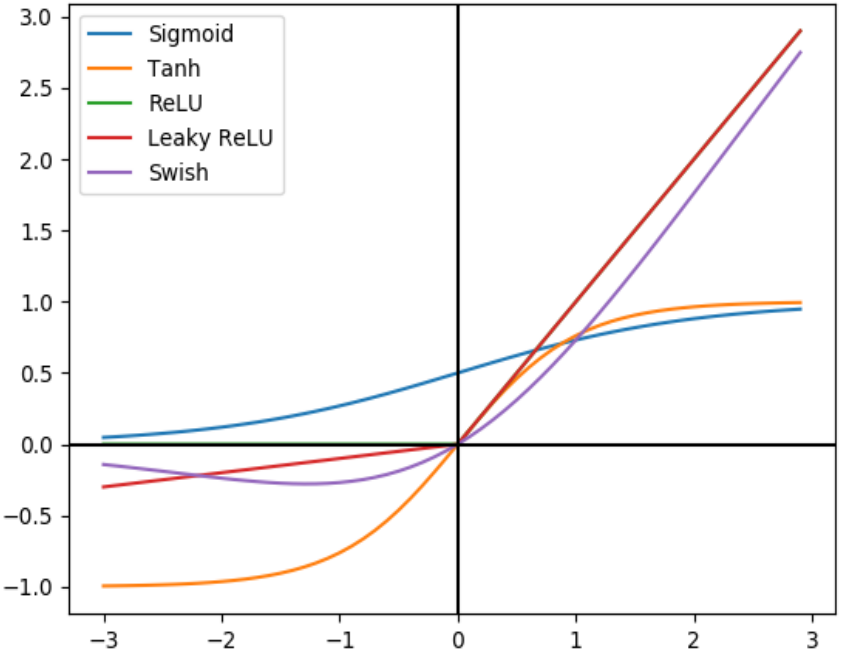
- Activation function

  - Activation functions are applied to each node of a neural network

  - Introduce non linearities into the network → allows to approximate complex shapes

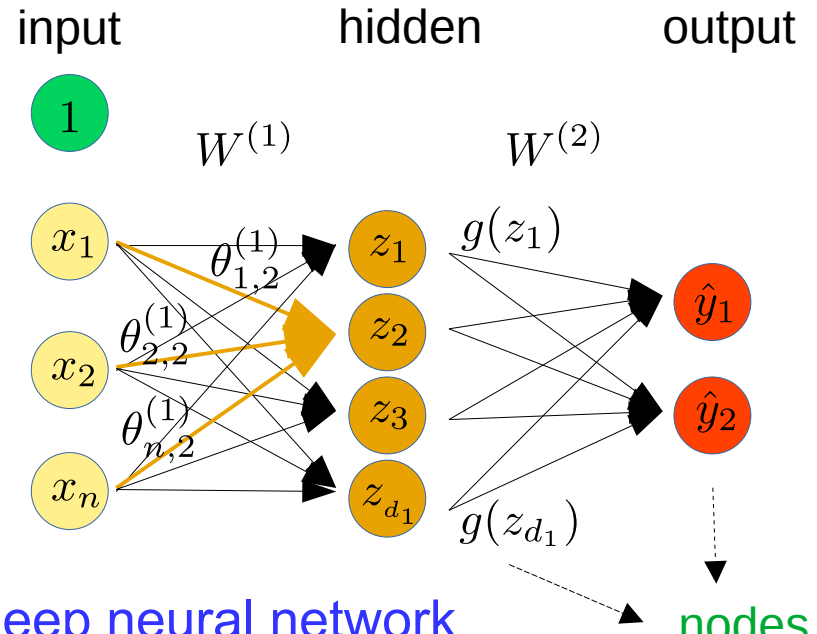
03_ml_basics_activation.ipynb

| ACTIVATION FUNCTION | EQUATION | RANGE |
|---|---|---|
| Linear Function | $f(x) = x$ | $(-\infty, \infty)$ |
| Step Function | $f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$ | $\{0, 1\}$ |
| Sigmoid Function | $f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$ | $(0, 1)$ |
| Hyperbolic Tanjant Function | $f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$ | $(-1, 1)$ |
| ReLU | $f(x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$ | $[0, \infty)$ |
| Leaky ReLU | $f(x) = \begin{cases} 0.01 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$ | $(-\infty, \infty)$ |
| Swish Function | $f(x) = 2x\sigma(\beta x) = \begin{cases} \beta = 0 \; for \; f(x) = x \\ \beta \to \infty \; for \; f(x) = 2\max(0, x) \end{cases}$ | $(-\infty, \infty)$ |



# Deep Neural Networks (DNN)

- Single layer neural network



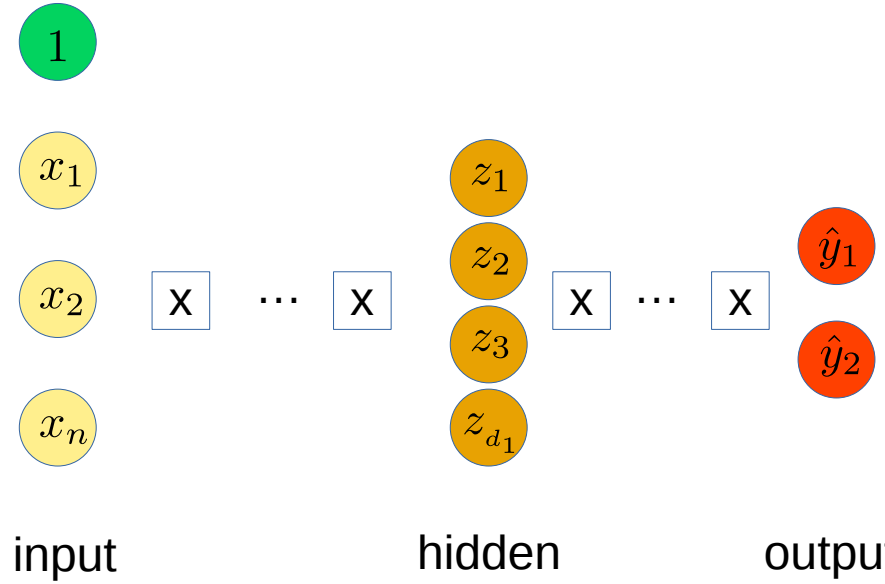

2nd element hidden layer 1 :

$$z_2 = w^{(1)}_{0,2} + \sum_{j=1}^{n} x_j w^{(1)}_{j,2}$$

ith output :

$$\hat{y}_i = g(w^{(2)}_{0,2} + \sum_{j=1}^{d1} z_j w^{(2)}_{i,j})$$

- Deep neural network

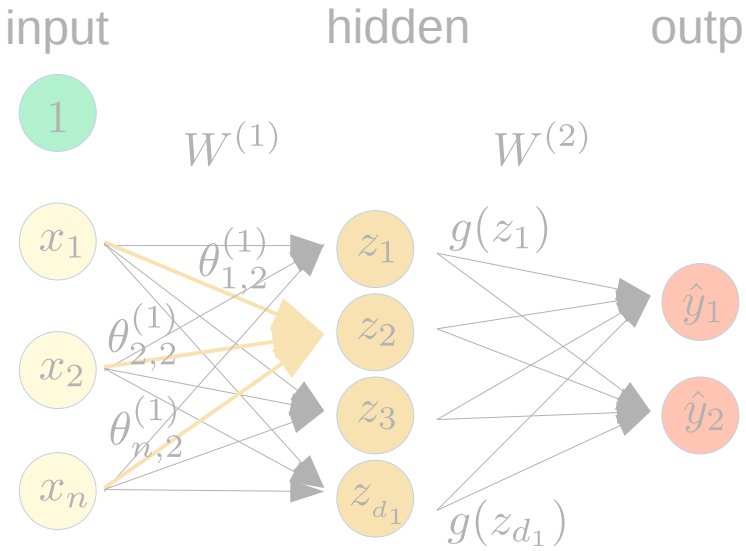



ith element hidden layer k :

$$z_{k,i} = w^{(k)}_{0,i} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j}) w^{(k)}_{i,j}$$

# Deep Neural Networks (DNN)

- Single layer neural network




- Have to apply activation functions on nodes in each hidden layer and the final output layer

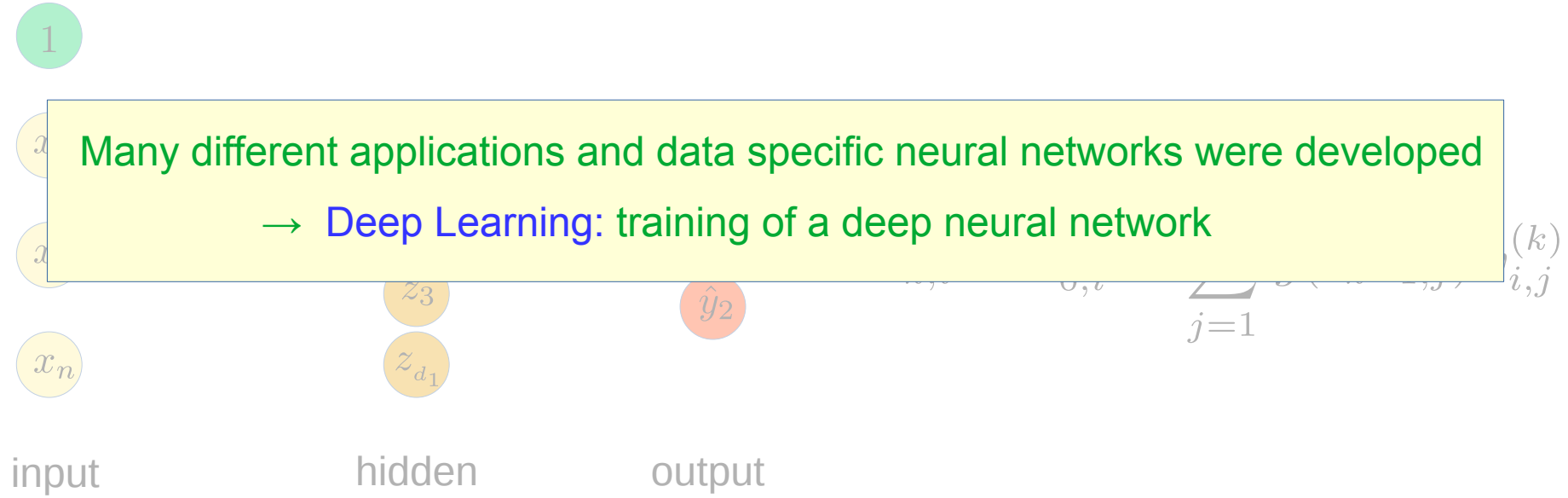
2nd element hidden layer 1 :

$$z_2 = w^{(1)}_{0,2} + \sum_{j=1}^{n} x_j w^{(1)}_{j,2}$$

$$z_i \rightarrow \theta(z_i)$$

Non-linear activation functions are really the key

- Deep neural network

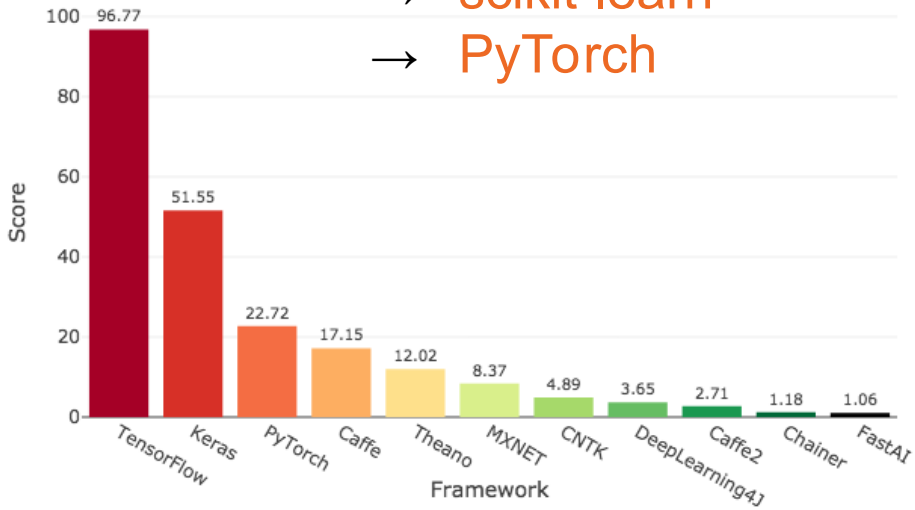


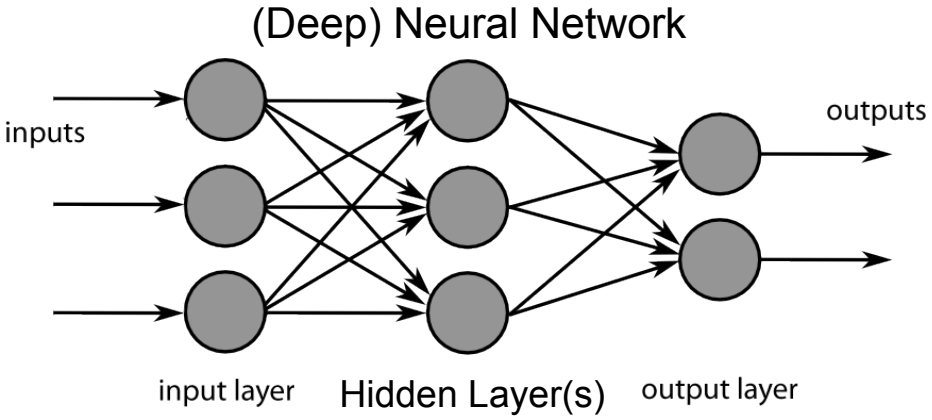

Many different applications and data specific neural networks were developed

→ Deep Learning: training of a deep neural network

# Deep Neural Networks (DNN)

- Deep learning
  - Part of a broader family of machine learning methods based on artificial neural networks that use multiple layers to progressively extract higher level features from raw input
- Deep neural network
  - Network with an input layer, at least a hidden layer and an output layer
  - Each layer performs specific types of sorting and ordering in a process that some refer to as “feature hierarchy”
  - Deal with unlabeled or unstructured data
  - Algorithms are called deep if the input data is passed through a series of hidden layers with nonlinearities (nonlinear transformations) before it becomes output.
- Most Deep Learning frameworks (user interface) are based on Python
  - → TensorFlow and Keras are one of the most popular frameworks
  - → scikit-learn
  - → PyTorch




https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

# Scikit learn


- Scikit-learn is an open-source machine learning library for Python https://scikit-learn.org/stable/
  - Current version 1.6.1
  - Includes simple tools for data mining and analysis
  - Software is based on NumPy SciPy and matplotlib

- Various classification, regression and clustering algorithms
  Many algorithms are included: k-nearest neighbors, multi-layer, perceptrons, support vector machines, random forests, gradient boosting, k-means

- Dimensionality reduction with PCA, feature selection, non-negative matrix factorization

- Model selection with comparing, validating and choosing parameters and models.

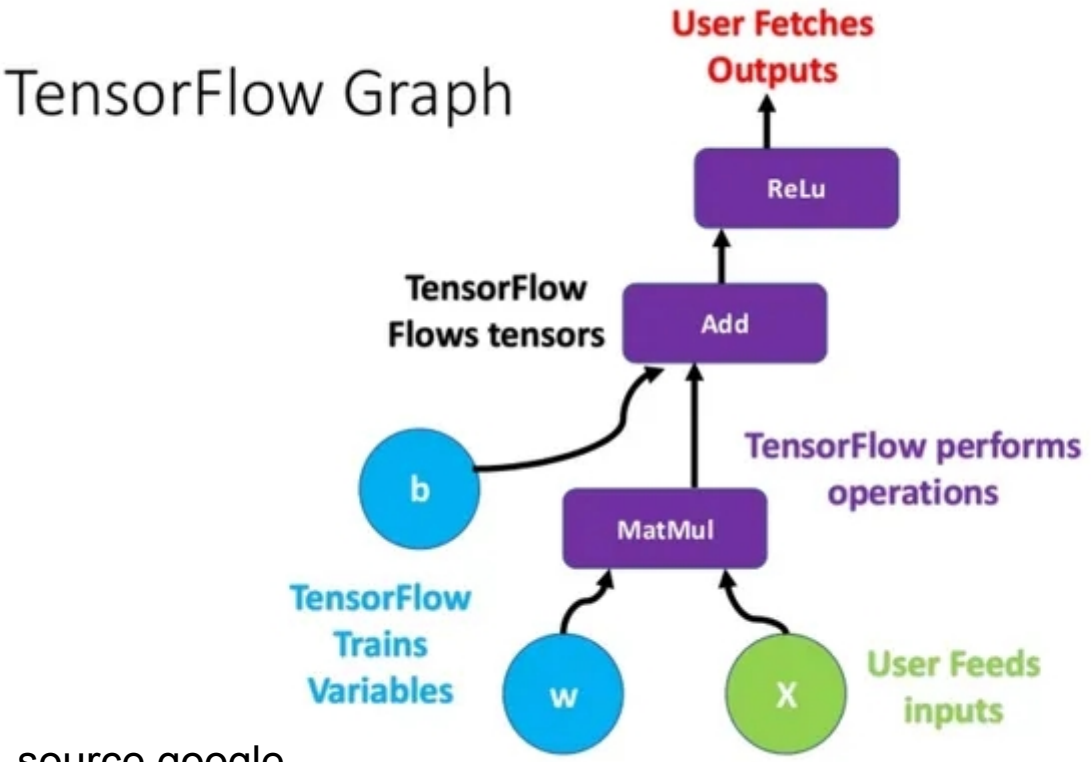
- Preprocessing: feature extraction and normalization

- Examples from various areas are available via the web site

# Introduction to Tensorflow

TensorFlow (TF), developed by the Google Brain team, is an open source software library for numerical computation using data flow graphs

- Tensors are multidimensional data, e.g a rank-0 tensor is a scalar, a rank-1 tensor is a vector, a rank-2 tensor is a matrix, ….
- Tensors represent in deep learning algorithms  input data, model parameters and intermediate activations of the model during training and inference. They can be created and manipulated using TensorFlow operations, and are passed between operations
- TF implements standard mathematical operations on tensors, as well as many operations specialized for machine learning
- TF runs on CPU, GPU and TPU
- TF handles datasets, splitting data in test and training samples and compiles models
- TF implements automatic differentiation (autodiff) to compute loss functions
- Processes training loops and keeps track of weights

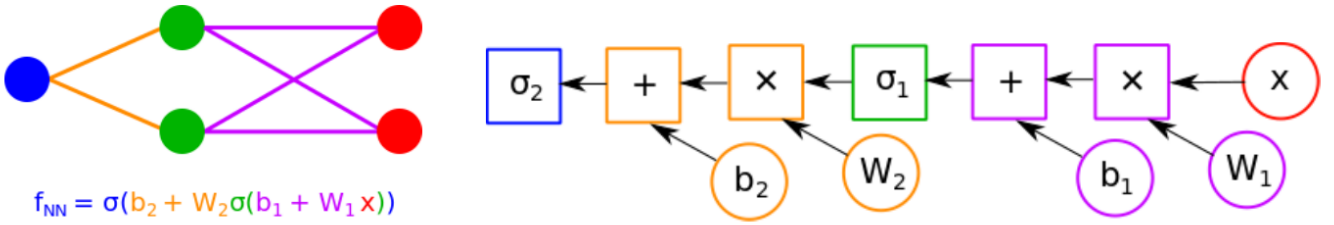



source google

# Computational Graphs

- Computational graphs are the basic concept of TensorFlow (TF)

Graph: S. Wunsch, IML workshop, 2019




Neural Network → Computational Graph

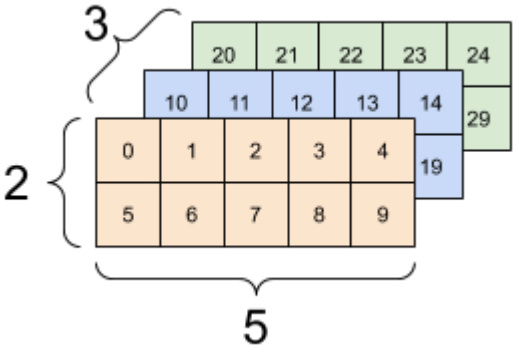
- Nodes in the graph represent mathematical operations
- Edges are represented by multidimensional data arrays (tensors) which communicate between the nodes
- The nodes take 1 or more input tensors, does the computation and produces 1 or more output tensor which are then passed on to the next nodes
- TensorFlow can optimize the graph by reordering and merging operations, eliminating redundant computations, and parallelizing operations
- Computational graph can be modified dynamically during runtime, allowing to build models that can adapt to different inputs and perform different computations depending on the input
- Computational graph can be visualized using tools like TensorBoard, to understand the structure of your model

- Eager execution → Calculations are performed on demand (as other python code)

# Introduction to Tensorflow

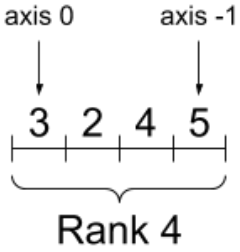
- Tensors for details, see https://www.tensorflow.org/guide/tensor

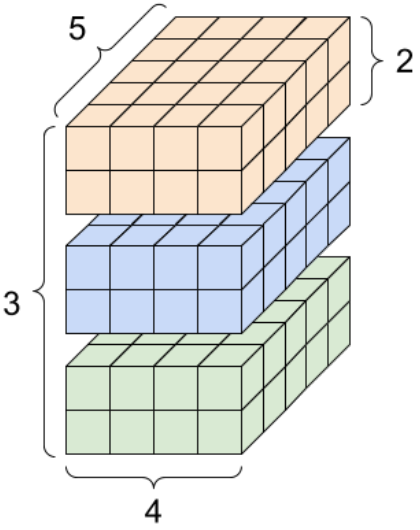
```
rank_0_tensor = tf.constant(4)                          # int32 tensor
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])            # float32 tensor
rank_3_tensor = tf.constant([                           # int32 tensor
  [[0, 1, 2, 3, 4],                                     # shape(3,2,5)
   [5, 6, 7, 8, 9]],
  [[10, 11, 12, 13, 14],
   [15, 16, 17, 18, 19]],
  [[20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29]],])
```




- can contain also `complex` and `string`
- can be created with arbitrary dimensions

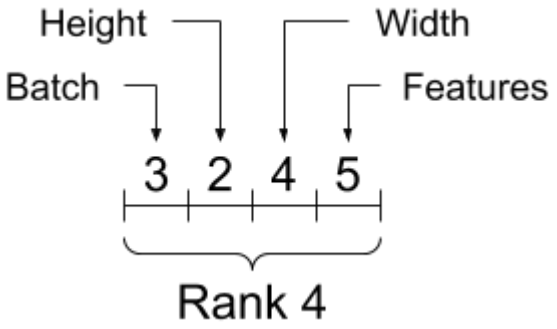
```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```




- axis are refered to by their indices

important:
keep in mind the meaning



# Introduction to Tensorflow

- Tensor operations
  - Export and import tensors to numpy arrays

```
tensor = tf.constant([[1,2,3], [4,5,6], [7,8,9]]) # rank 3 tensor
arr = tensor.numpy()                               # numpy array

arr = np.array([[1,2,3], [4,5,6], [7,8,9]])        # numpy array
tensor = tf.constant(arr)                          # imported tensor
```

  - Accessing parts of tensors via numpy

```
rank_2_tensor[1, 1].numpy()          # access single tensor elements
rank_2_tensor[-1, :].numpy()         # last row
rank_2_tensor[1:, :].numpy()         # skip first row
rank_2_tensor[:, 1].numpy()          # 2nd column
```

  - Reshaping of tensors, a new tensor is created pointing to the same data

```
rank_3_tensor = tf.reshape(rank_3_tensor, [-1])
```

`-1` means, shape it to whatever fits → in this case a 1 D tensor is created

```
tf.Tensor([ 0  1  2  3  4……. 26 27 28 29], shape=(30,), dtype=int32)
tf.reshape(rank_3_tensor, [3*2, 5]) # reshaping to (3x2)x5
tf.reshape(rank_3_tensor, [3, -1])  # reshaping to 3x(2x5)
```

Axis are swaped using `tf.transpose`

  - Tensors are rectangular structures, but there is `tf.RaggedTensor` (a non rectangular structure) and `SparseTensor` (only few elements are nonzero)

# Introduction to Tensorflow

- Tensor operations

  - Basic math on tensors

```
tf.add(a,b) and tf.multiply(a,b)    # elementwise operation
tf.matmult(a,b)                     # matrix multiplication
tf.reduce_sum(a)                    # Sum all elements in the tensor
tf.square(a)                        # Square all elements
tf.divide(b, a)                     # Element-wise division
tf.negative(a)                      # Element-wise negation
tf.abs(tf.constant([-1, 2, -3, 4], dtype=tf.float32))
                                    # Element-wise absolute value

c = tf.constant([[4.0, 5.0], [10.0, 1.0]])
tf.reduce_max(c)                  # Find the largest value
tf.math.argmax(c)                 # Find the index of the largest value
tf.nn.softmax(c)                  # Compute softmax from tf.nn module

rank_3_tensor.dtype               # type of the elements  dtype:'int32'
rank_3_tensor.ndim                # number of axes        3
rank_3_tensor.shape               # shape of tensor       (3, 2, 5)
rank_3_tensor.shape[0]            # elements along axis 0 of tensor
rank_3_tensor.shape[-1]           # elem along the last axis of tensor
tf.size(rank_3_tensor).numpy()    # total number of elements
```

# Introduction to Tensorflow

- Tensor operations

- Broadcasting is a feature in TensorFlow that allows operations to be performed on tensors of different shapes, so that there is no need to manually reshape tensors to match each other.
  Broadcasting works by replicating tensor elements along one or more dimensions to match the shape of the other tensor(s) in the operation.
  Broadcasting rule:
  - If the two tensors have the same rank, they are compatible if their shapes are equal in each dimension.
  - if the two tensors have different ranks, the smaller tensor is broadcast to match the shape of the larger tensor. The smaller tensor is pre-padded with 1s on the left until it has the same rank as the larger tensor.
  - If the two tensors have different shapes and cannot be broadcasted according to rules 1 and 2, an error is raised.

- Example of broadcasting in TensorFlow: 03_ml_basics_tf_broadcasting.ipynb

```
import tensorflow as tf
a=tf.constant([[1,2,3], [4,5,6]])      # Define 2 tensors with different shapes
b=tf.constant([10, 20, 30])
c = a * b                              # Perform element-wise multiplication using broadcasting
print(c)                                [[ 10  40  90]
                                         [ 40 100 180]],shape=(2, 3),dtype=int32)
```

# Introduction to Tensorflow

- Variables in TF

- Variables are created and tracked via the tf.Variable class, they can be modified after intialization

```
tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
variable = tf.Variable(tensor)      # same type as the initialization
variable.numpy()                    # convert to numpy
tf.convert_to_tensor(variable)      # convert to a tensor
tf.math.argmax(variable))           # index of highest value

b = tf.Variable([1, 2, 3])          # creates a variable tensor
b.assign_add([1, 1, 1])             # and then modify its values
a = tf.Variable([2.0, 3.0])         # create variable a
b = tf.Variable(a)                  # Create b based on value of a
a.assign([5, 6])                    # asign a new value to a
a.assign_add([2,3]).numpy()         # add element wise to a
a.assign_sub([2,3]).numpy()         # subtract element wise from a

a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.Variable([[1.0, 2.0, 3.0]])
k = a*b                             # element-wise multiply

a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)                 # new tensor
```

# Introduction to Tensorflow

- Gradients and differentiation in TF

To differentiate automatically, TF needs to remember what operations happen in which order during the forward pass. In backward pass the list of operations is done in reverse order. `GradientTape.gradient(target, sources)` is used to calculate the gradient of some target (often a loss) relative to some source.

- Example with a scalar

```
x = tf.Variable(3.0)                    # variable  = 3
with tf.GradientTape() as tape:
   y = x**2
dy_dx = tape.gradient(y, x)             # dy = 2x * dx
dy_dx.numpy()                           # convert to a number     = 6.0
```

03_ml_basics_tf_differentiate.ipynb

- Example with tensors

```
w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]
with tf.GradientTape(persistent=True) as tape:
   y = x @ w + b    # tape can be used multiple times to calc grad
   loss = tf.reduce_mean(y**2) # get the mean value
[dl_dw, dl_db] = tape.gradient(loss, [w, b]) # gradients to both vars
```

# Machine Learning Datasets

- Machine Learning datasets in TF

- TensorFlow provides several built-in datasets for machine learning tasks from various areas and different topics, e.g. 3d - 2d image detection, classification and generation, categorical solutions, abstractive text summarization, anomaly detection and language modeling
  detailed infos in https://www.tensorflow.org/datasets/catalog/overview

- cached locally in ~/tensorflow_datasets in one directory per dataset

```
import tensorflow as tf
(x_train,y_train),(x_test,y_test)=tf.keras.datasets.mnist.load_data()
```

- Example: import `horses_or_humans` dataset, select horses and preprocessing

```
import tensorflow_datasets as tfds
dataset, info = tfds.load('horses_or_humans', with_info=True)
horse_ds = dataset['train'].filter(lambda x: x['label'] == 0)        # horses
horse_examples = horse_ds.take(5)                          # 5 horses from the dataset

train_dataset, valid_dataset = tfds.load('horses_or_humans',
                          split=['train','test'], as_supervised=True)

def preprocess(image, label):
  image = tf.cast(image, tf.float32)
  image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))#resize to a fixed size
  image = image / 255.0       #rescale the pixel values to be between 0 and 1
  label = tf.one_hot(label, NUM_CLASSES)                          #asign labels
  return image, label
```

03_ml_basics_display_HorseOrHuman.ipynb

# Machine Learning Datasets

- Input data usually has to be pre processed

  - Needed for numerical stability

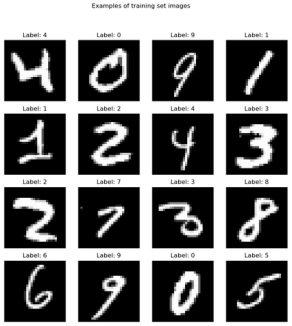
  - Want to have the mean around zero and the variance order of one

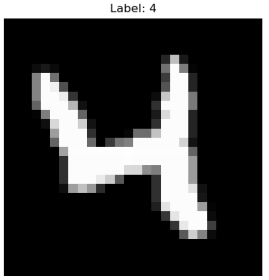
    - gaussian like distributed data → subtract mean and divide by standard deviation
    - data peaking at zero → apply logarithm and then subtract mean and divide by standard deviation

- Example datasets in the MNIST database (Modified National Institute of Standards and Technology database) used for machine learning and practicing techniques
  The data can be loaded directly in Tensorflow using e.g. tf.keras.datasets.mnist.load_data()

  - MNIST handwriting
    60000 images

    03_ml_basics_display_Ha




28 x 28 array with greyscale values, 0 – 255
→ preprocessing: divide by 255

use pixel array and label

  - Fashion-MNIST dataset
    60000 images

    03_ml_basics_display_Clothing.ipynb

28 x 28 array with greyscale values, 0 – 255 and labels

- Input data needs often reshaping

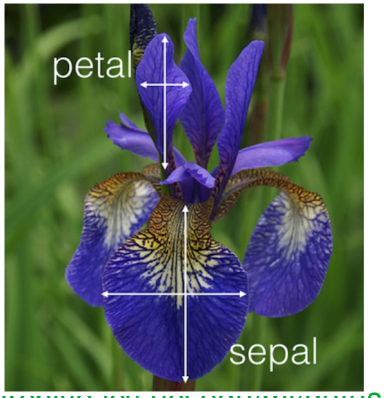
# Machine Learning Datasets

- Example datasets for classification applications

  - Iris dataset from Fisher (1936) with 3 classes, Iris Setosa, Iris Versicolour and Iris Virginica with 50 instances each. There are 4 attributes:

    - sepal length [cm]
    - sepal width [cm]
    - petal length [cm]
    - petal width [cm]

    The application is for classification

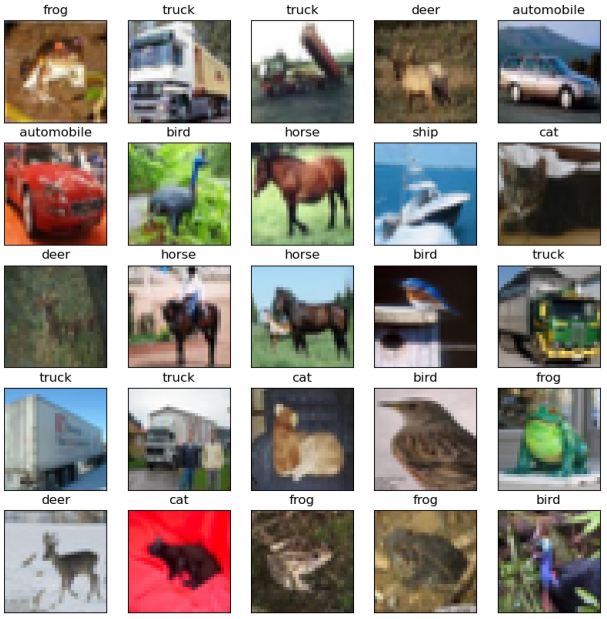



https://archive.ics.uci.edu/ml/datasets/iris

  - Cifar-10 and Cifar-100 dataset of 32 x 32 color pictures in 10 classes (100 classes) with 60000 images per class and 10000 test images for Cifar-10.

    The 100 classes in the CIFAR-100 are grouped also into 20 superclasses. There are 500 training images and 100 testing images per class

    https://www.cs.toronto.edu/~kriz/cifar.html



# Exercise 1

- Read/load the cifar10 dataset using `tf.keras.datasets`
  There are the following classes

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
               'frog', 'horse', 'ship', 'truck']
```

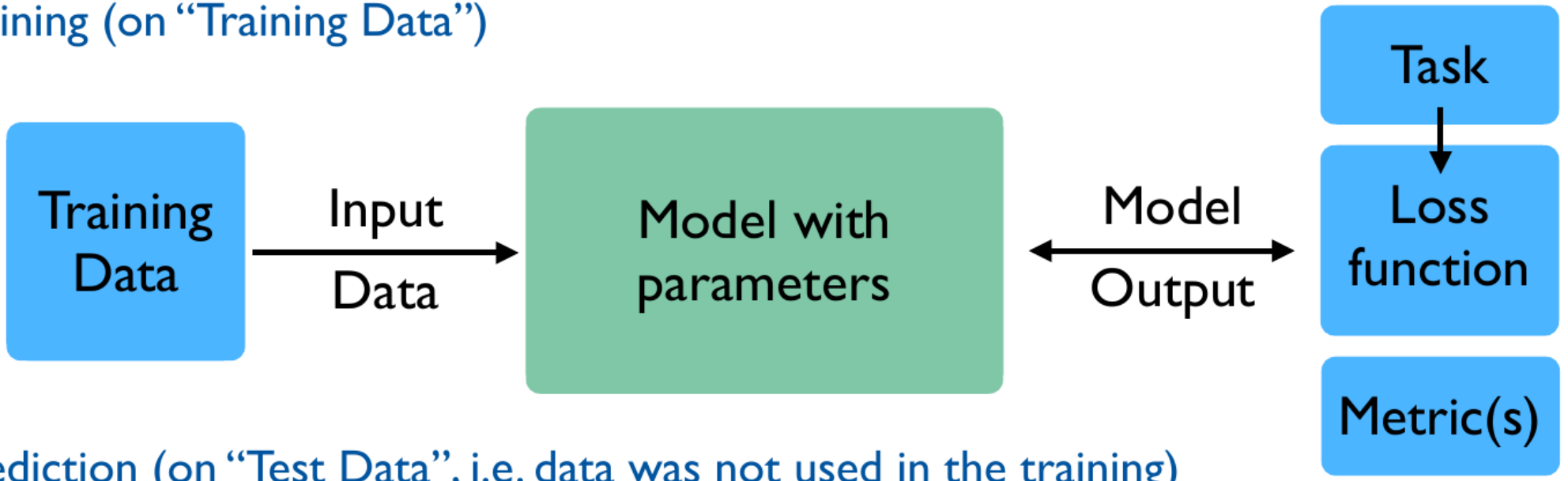
  Display the first 25 images

- They are color images, try to convert them to grey scale images by reducing the 3 colors (r,g,b) to one grey scale with the formula
  gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
  Normalize the data to the interval [0,1]
  Display the first 25 greyscale images

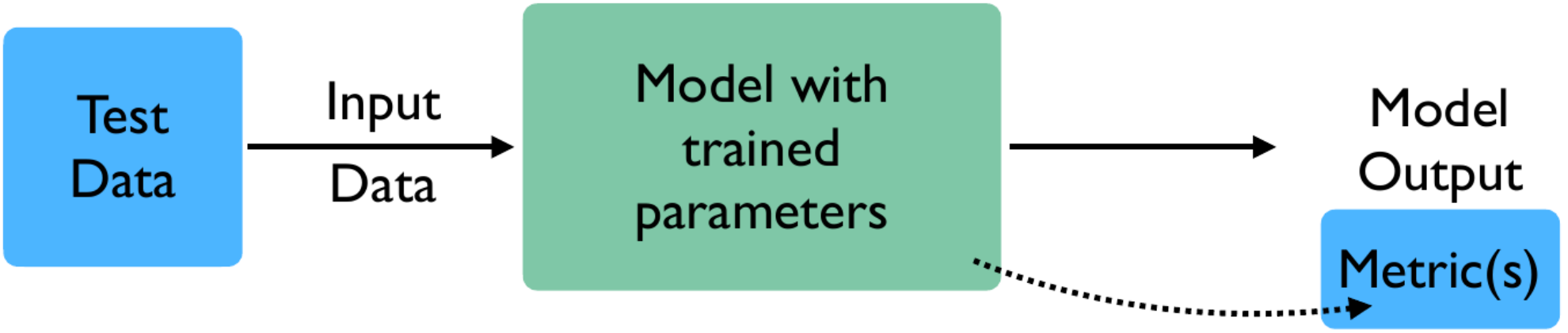
03_ml_basics_ex_2_display_Cifar10_Greyscale.ipynb

# Introduction Machine Learning

- Training (on "Training Data")




- Prediction (on "Test Data", i.e. data was not used in the training)



# DNN Training

- Metric

Machine learning datasets contain truth information to allow an independent quality measure of the neural network. Metrics are typically defined in terms of the neural network model's predictions and the true labels. They quantify how well the model is doing at its task.

- accuracy: number of correctly classified data points over the total number.
- precision: ratio of true positive (TP) predictions to the total number of positive (TP + false positive) predictions.
- recall: ratio of true positives to the sum of true positives and false negatives
- F1 score: F1 score = 2 * (precision * recall) / (precision + recall)

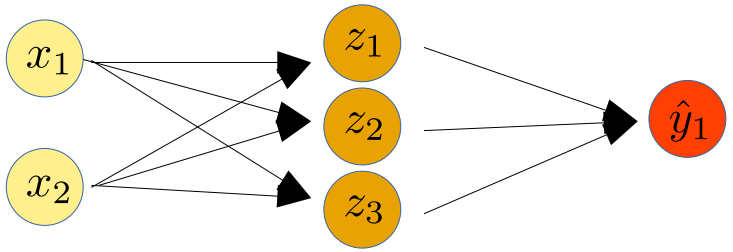
- Splitting the dataset in 2 or 3 parts

- Traing dataset – Do all the parameter determination (training) of the neural network.
- Validation dataset – Used to evaluate the model during training and to tune its hyperparameters ( e.g. handled by tensorflow during minimization)
- Test dataset – Use it for evaluating the optimized parameters and the final performance of the model after it has been fully trained.

Splitting: Traing:Validation:Test → 60%:20%:20%

# DNN Loss Function

- Quantifying quality/success of a neural network
  - Compare predicted output with the true output → loss function




$$\mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

predicted true

- Empirical loss
  total loss over the entire dataset

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

- Cross entropy loss for models
  with output $\in [0, 1]$

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} log(f(x^{(i)}; W)) + (1 - y^{(i)}) log(1 - f(x^{(i)}; W))$$

true predicted true predicted

- Mean squared error loss for regression with continuous real numbers

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - f(x^{(i)}; W)))^2$$

true predicted

test minimizer in python:
03_ml_basics_minimizer.ipynb

# DNN Minimization

- Find the network weights such that the loss function is minimal

$$W_{min} = \underset{w}{argmin} J(W) = \underset{w}{argmin} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

Repeat until convergence

- Initialize weights randomly
- Loop until convergence:

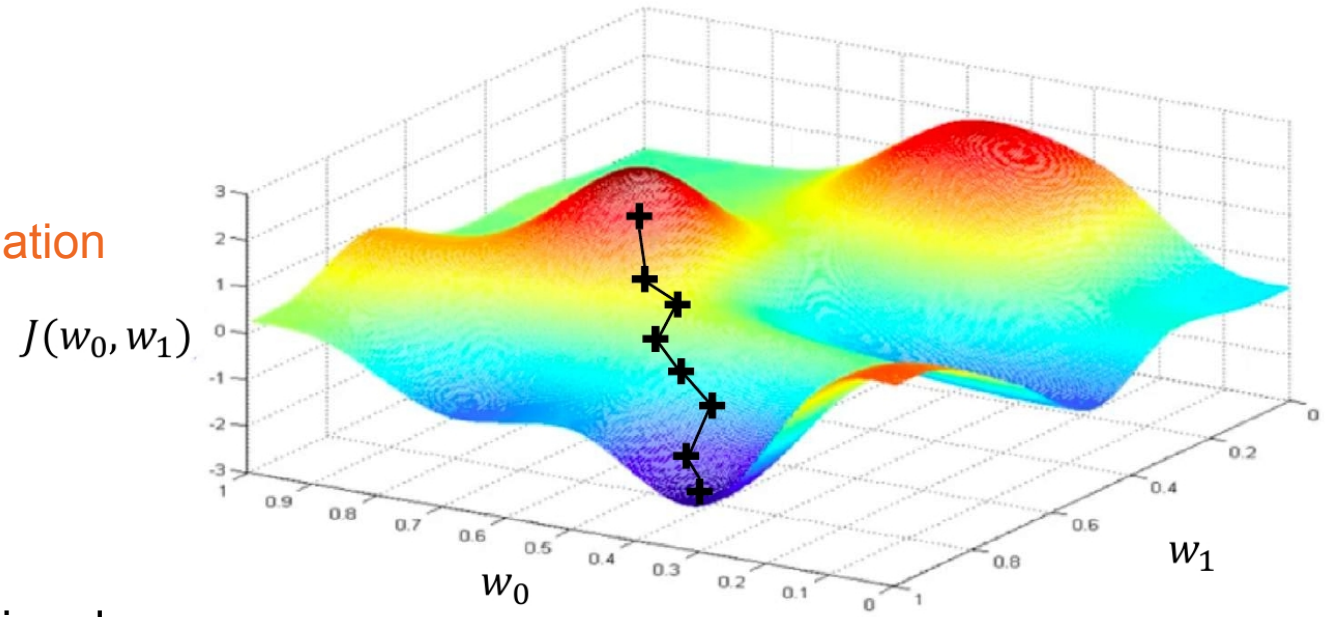
  compute $\frac{\partial J(W)}{\partial W}$ backpropagation

  update weights

  $$W \leftarrow W - \eta \cdot \frac{\partial J(W)}{\partial W}$$

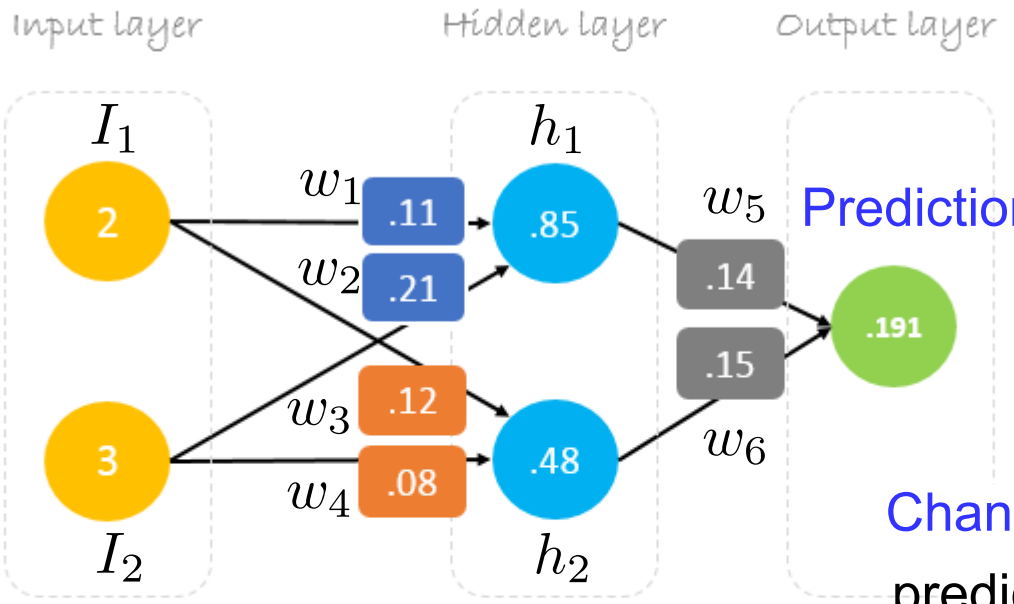
- return weights

- derivative calculation with chain rule



# Example - Backward Propagation

- Finding weights that minimizes the error function




Change weights to reduce the error function:

prediction = $(I_1 w_1 + I_2 w_2) w_5 + (I_1 w_3 + I_2 w_4) w_6$

error = 0.5 (Prediction - True)$^2$

$= \Delta$

$$[2 \quad 3] \cdot \begin{bmatrix} 0.11 & 0.12 \\ 0.21 & 0.08 \end{bmatrix} = [0.85 \quad 0.48] \cdot \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} = [0.191]$$

backward propagation updates the weights by calculating the gradient of the error function with respect to the neural network's weights.

$$w_6^* = w_6 - a\left(\frac{\partial Error}{\partial w_6}\right) \quad \Rightarrow \quad \frac{\partial Error}{\partial w_6} = \frac{\partial Error}{\partial pred} \cdot \frac{\partial pred}{\partial w_6} \quad \Rightarrow \quad \frac{\partial Error}{\partial w_6} = \Delta \cdot h_2$$

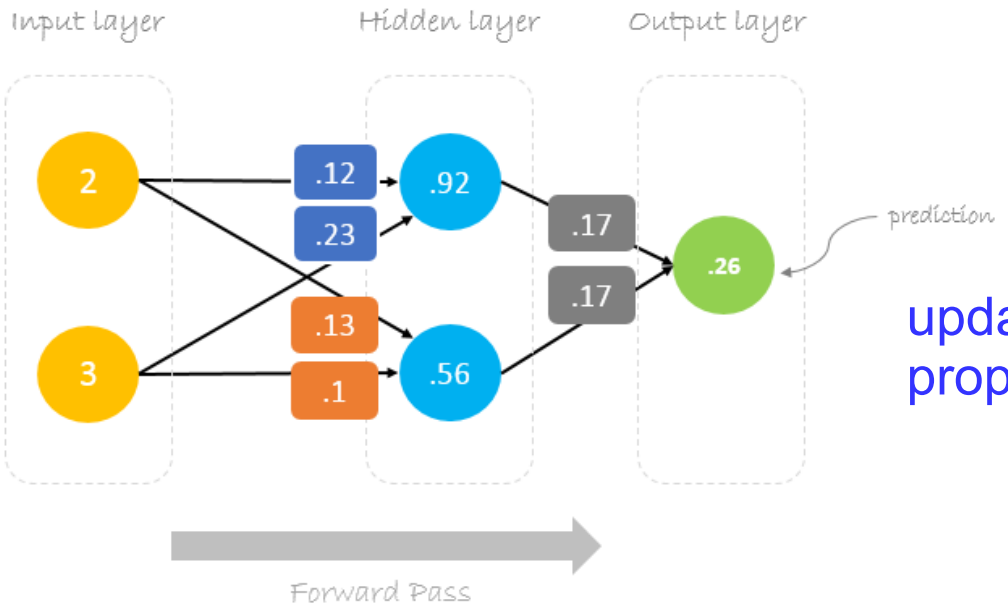
# Example - Backward Propagation

- New weights after the backward propagation

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - a\,\Delta \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - \begin{bmatrix} a h_1 \Delta \\ a h_2 \Delta \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - a\,\Delta \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} . [w_5 \quad w_6] = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \begin{bmatrix} a\, i_1 \Delta w_5 & a\, i_1 \Delta w_6 \\ a\, i_2 \Delta w_5 & a\, i_2 \Delta w_6 \end{bmatrix}$$

$$^{*}w_6 = w_6 - a\,(h_2 \,.\, \Delta)$$
$$^{*}w_5 = w_5 - a\,(h_1 \,.\, \Delta)$$
$$^{*}w_4 = w_4 - a\,(i_2 \,.\, \Delta w_6)$$
$$^{*}w_3 = w_3 - a\,(i_1 \,.\, \Delta w_6)$$
$$^{*}w_2 = w_2 - a\,(i_2 \,.\, \Delta w_5)$$
$$^{*}w_1 = w_1 - a\,(i_1 \,.\, \Delta w_5)$$




$$[2 \quad 3] . \begin{bmatrix} 0.12 & 0.13 \\ 0.23 & 0.10 \end{bmatrix} = [0.92 \quad 0.56] . \begin{bmatrix} 0.17 \\ 0.17 \end{bmatrix} = [0.26]$$

updated prediction after backward propagation

# Introduction to Deep Neural Networks

- Find the network weights such that the loss function is minimal

$$W_{min} = \underset{w}{argmin} J(W) = argmin \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

- Initialize weights randomly
- Loop until convergence:

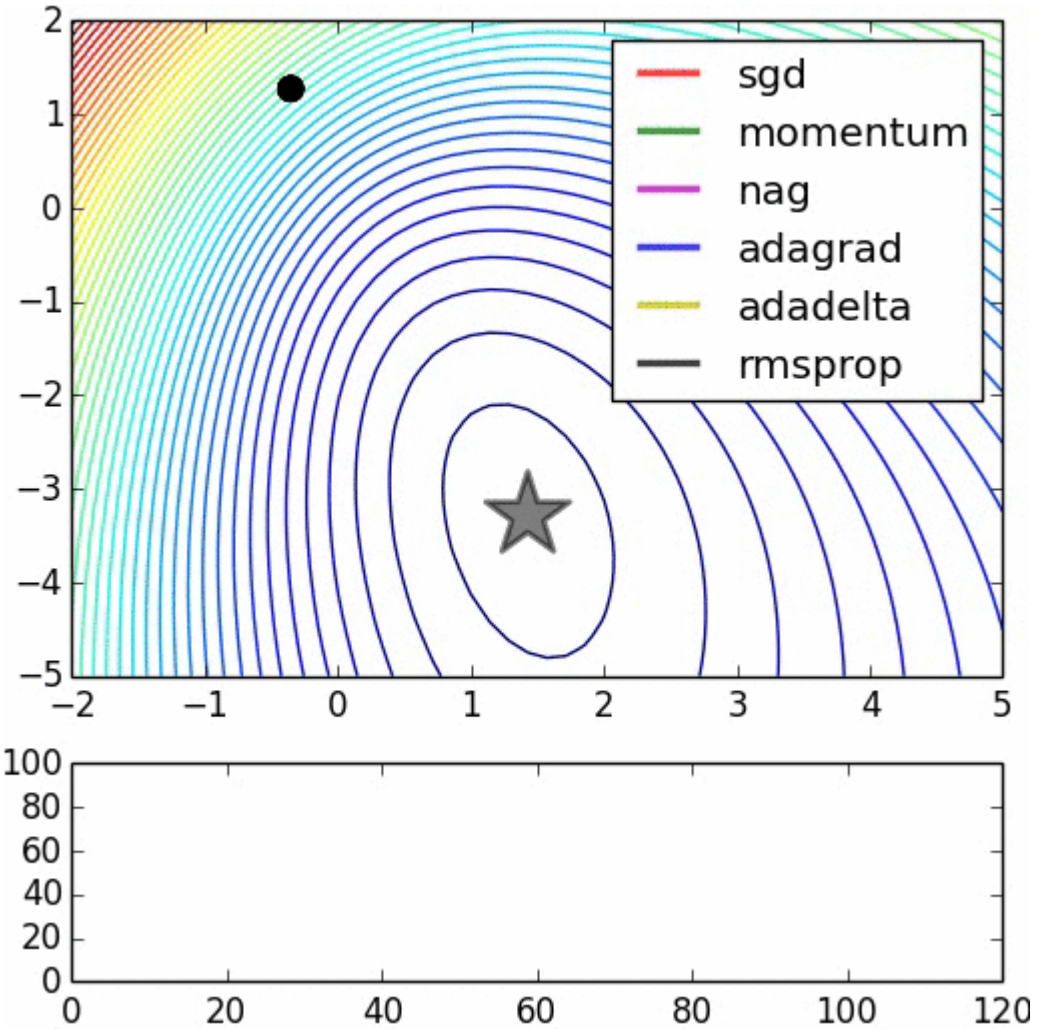
  compute $\frac{\partial J(W)}{\partial W}$ backpropagation

  update weights

  $$W \leftarrow W - \eta \cdot \frac{\partial J(W)}{\partial W}$$

- return weights

- derivative calculation with chain rule

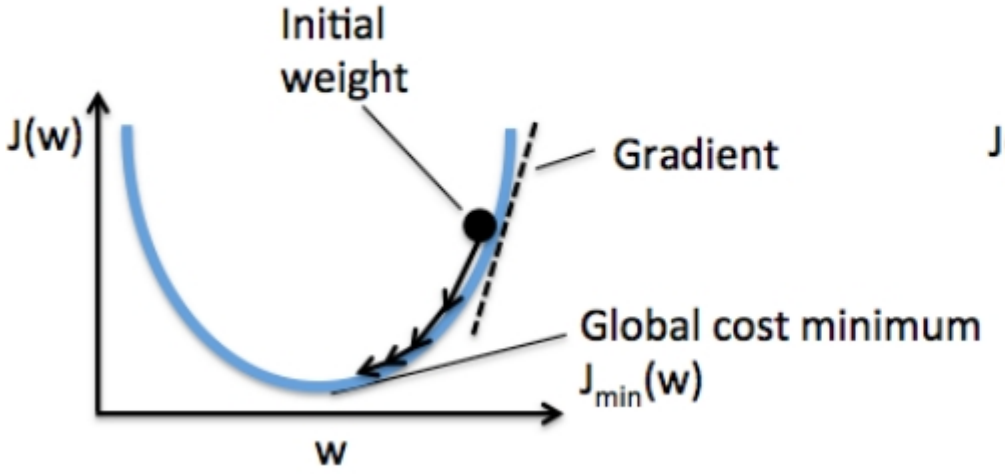


# DNN Minimization

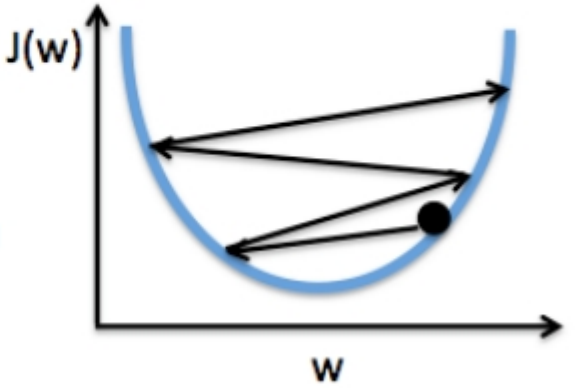
- Training strategies
  - Minimization: difficult to converge to a global minimum with large number of parameters.
  - Learning rate

$$W \rightarrow W - \alpha \cdot \frac{\partial J(W)}{\partial W}$$

$\alpha$ is called learning rate




choosing the right learning rate is important for convergence




learning rate is too large

[https://raw.githubusercontent.com/rasbt/python-machine-learning-book/master/code/ch02/images/02_12.png, MIT License]

- Adaptive moment estimation (Adam) uses adaptive learning rates which reduces α during minimization for each parameter separately

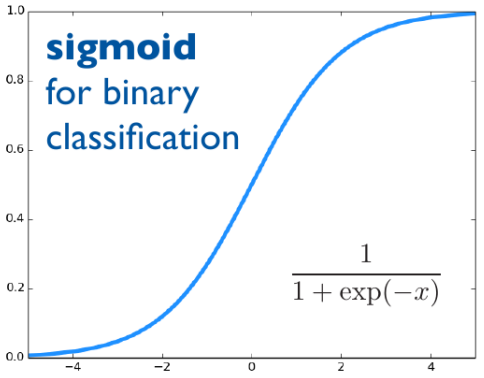
$$W \rightarrow W - (1-\beta) \cdot \alpha \cdot \frac{\partial J(W)}{\partial W}$$

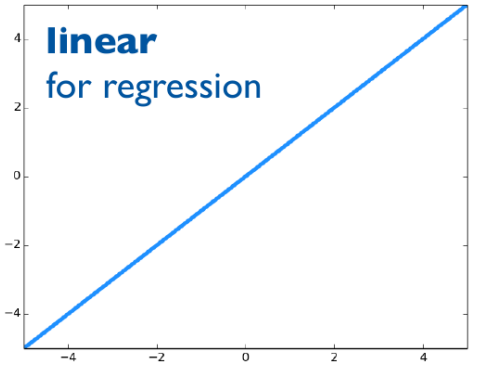
β change in W from previous step

# DNN Minimization

- Output to the last node(s)

  - The output function(s) of the neural network (nodes in the final layer) are determined by activation functions. Their choice depends on the problem to be solved.
  - For binary classification problems the sigmoid function is used and interpreted as probability.
  - For regression problems a linear activation function or no activation function is used.



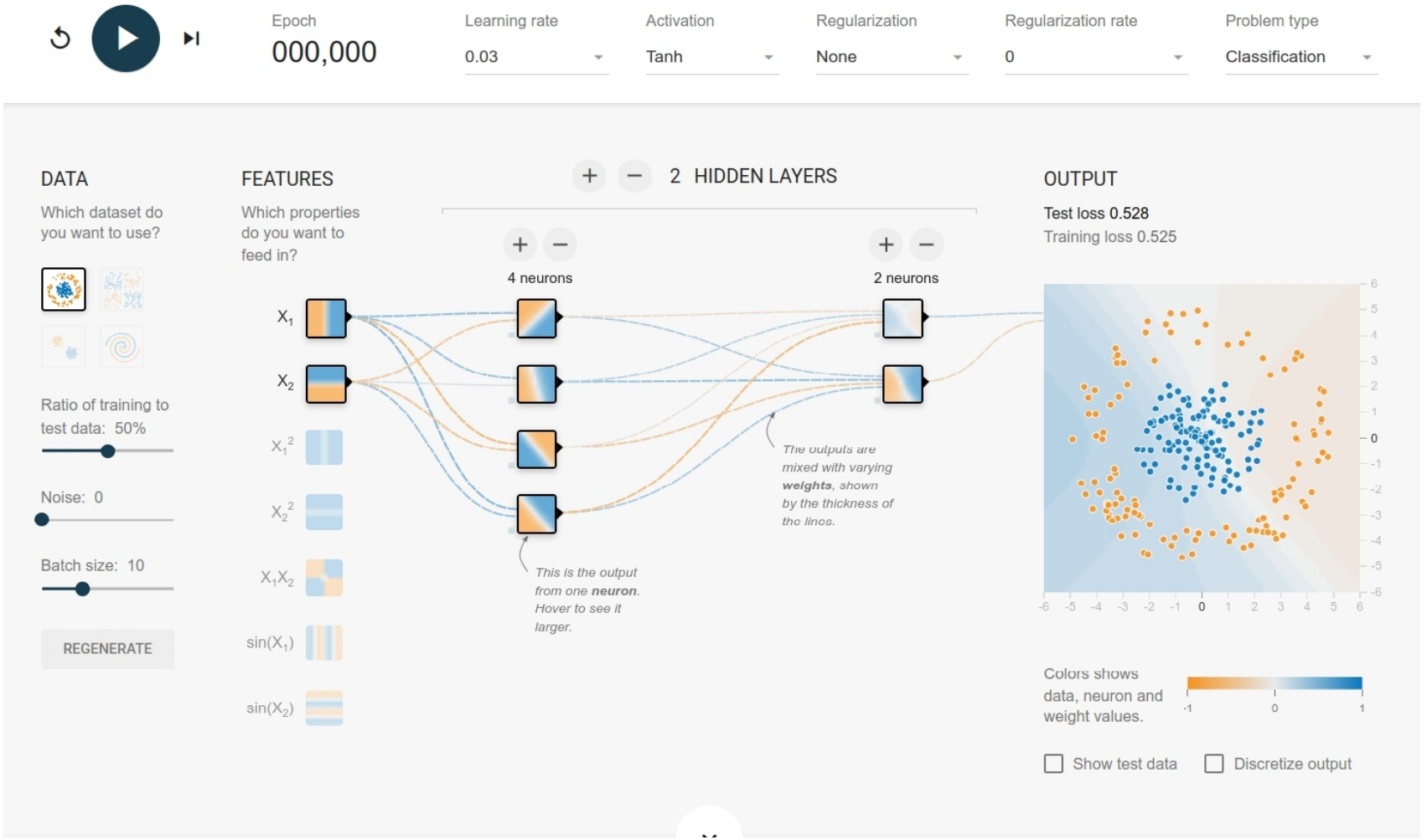

  - For multi-class classification problems the softmax function is mostly in use and gives a probability distribution over the classes.
    Softmax assigns decimal probabilities to each class i of a multi class problem with its N multiclass output values:

$$\mathrm{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

# Exercise 2

The following page illustrates a classification task which can be configured online via the web page https://playground.tensorflow.org/

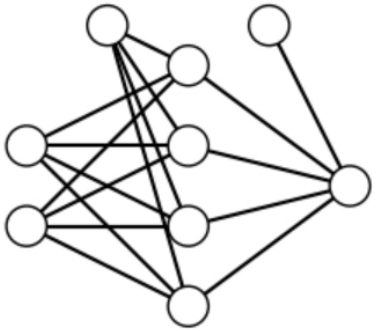
- Try different activation functions and change the learning rate



# Example Python Code of a Neural Network

- A simple feed forward neural network with one hidden layer is programmed in pure python without using machine learning libraries

  03_ml_basics_simple_neural_network.ipynb



# Exercise 3

- Modify the activation function of the simple neural network in python from the sigmoid function to the Rectified Linear Unit (ReLU) function

  03_ml_basics_simple_neural_network_exercise_solution.ipynb

# DNN Minimization

- Minimizer algorithms: difficult to converge to global minimum with large number of parameters
  - various minimizers are available in the `tf.keras.optimizers` module of TensorFlow
    - Stochastic Gradient Descent (SGD)
      gradient descent algorithm updates the model parameters (weights and biases) in small batches based on the gradients computed on each batch in the direction of the negative gradient scaled by α
    - Momentum optimizer
      updates the parameters by computing the gradient of the loss function with respect to the parameters and adjusting the parameters in the direction of the negative gradient taking into account previous updates to get the direction and magnitude
    - AdaGrad optimizer
      adapt the learning rate for each parameter based on the historical gradient by keeping a running sum of the squared gradients for each parameter (accumulator) and divide the learning rate by the square root of the accumulator for each parameter, which adapts it.
    - RMSProp optimizer
      In RMSprop (Root Mean Square Propagation) the learning rate is adaptively scaled for each weight parameter based on the root mean square (RMS) of the gradients. This reduces the learning rate for parameters with large and rapidly changing gradients
    - Adaptive Moment Estimation (Adam)
    - AdaDelta optimizer
    - Ftrl optimizer
  - Choice of optimizer often depends on the specifics of the problem
    → experiment with different optimizers

03_ml_basics_minimizer_methods.ipynb

# DNN Minimization

- Start values for the neural network parameters are needed to achieve good performance and convergence speed
  - Random initialization: sample from a uniform or normal distribution with zero mean and a small standard deviation.
  - Xavier initialization: randomly sampled from a normal distribution with zero mean, the standard deviation scales with 1 / sqrt (number of input layers)
  - Pretraining initialization: If a model has been pretrained, the weights can be initialized from the pretrained model.
  - In general:
    var(input) ≈ var(output) with var ≈ 2 / ($N_{\text{input nodes}}$ + $N_{\text{output nodes}}$)
    draw from gaussian or uniform distributions within a range $\pm\sqrt{3var}$
- Usually the input range of the variables differ largely → scale input variables
  - Important because the model converges faster and improves its accuracy.
  - Standardization: scales the input variables so that they have a mean of zero and a standard deviation of one.
  - Min-Max scaling: scales the input variables to a fixed range, usually between 0 and 1.
  - Robust scaling: scales the input variables using the median instead of the mean and standard deviation. To be used when the input variables have outliers or are not normally distributed.
  - Perform de-correlation of input data

# Examples - Deep Neural Networks

- Example of a binary classification using TensorFlow (TF)

  - Generate toy sample with 2 normalized gaussian distributions with mean (-1,-1) and (1,1)
  - Each sample gets a label to belong to one of the distributions
  - The data is combined to a training dataset
  - The data is plotted and colored in blue and red according to their labels
  - In TensorFlow a model is set up specifying the network → shape of the input data, number of neurons in each layer, the activation functions in each layer
  - The model is a sequential MLP model that consists of a series of layers stacked one on top of the other.
  - In the next step the model is compiled → here we choose the optimizer, the loss function, and the metrics
    As optimizer we use `adam`
    As loss function we choose `BinaryCrossentropy()` because we have 2 classes of data. It measures the dissimilarity between the predicted probabilities of the model and the true labels, which are either 0 or 1 for each example. The goal is to minimize this dissimilarity, or loss, during the training process.
    `accuracy` as metric is calculated as the number of correct predictions divided by the total number of predictions made by the model.
  - The data and labels are given to the fit method, also the training iterations and the batch size. The model doesn't process the entire dataset at once, but rather in batches.
  - Loss and accuracy are plotted as number of iterations
  - Display classification results for sample points together with labeled data points

03_ml_basics_tf_binary_classification_example.ipynb

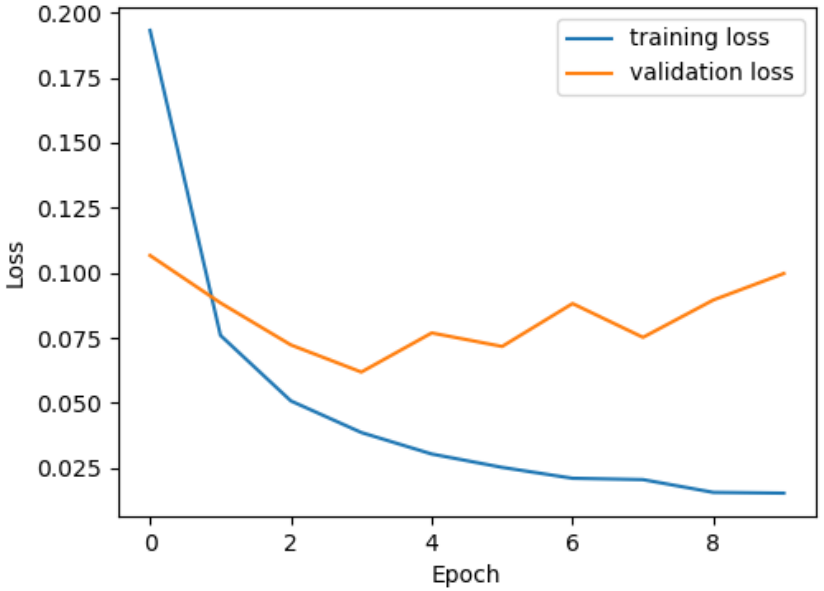
# Examples - Deep Neural Networks

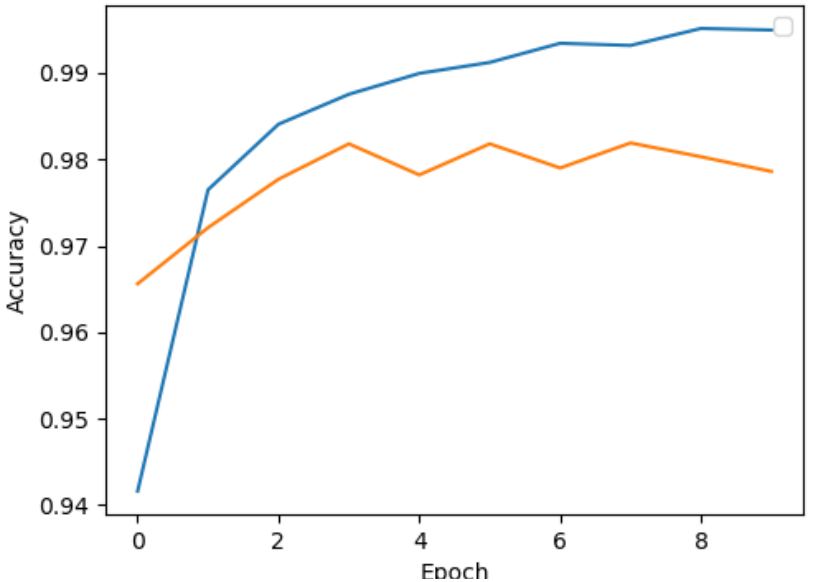
- Using the handwriting MNIST example with TensorFlow (TF)
  - Read in the dataset with a pre defined function in TF. There are 10 categories
  - Each sample has a label and the 28 x 28 pixel gray scale data array
  - The data is normalized and reshaped to a 1D array.
  - In TensorFlow a model is set up specifying a first dense layer with 512 nodes and a Rectified Linear Unit (ReLU) activation function. Dense means all nodes are fully connected. There is a second layer also with 512 nodes and ReLU activation. The 3rd layer is the output layer with 10 nodes and Softmax is used as activation.
  - As optimizer adam is used to find the minimum and accuracy as metrics
  - The data and labels are given to the fit method, also the training iterations and the batch size. The model doesn't process the entire dataset at once, but rather in batches.
  - The weights of the output layer are plotted
  - Loss and Accuracy are plotted as number of iterations (epoch)
  - Test accuracy = 0.98
  - The data is plotted and true and predicted label are shown

03_ml_basics_tf_mlp_mnist_digits.ipynb

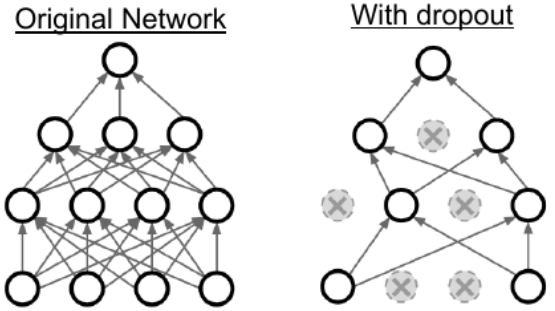
# Examples - Deep Neural Networks

- Overtraining: We observe a difference in loss and accuracy in training and test data



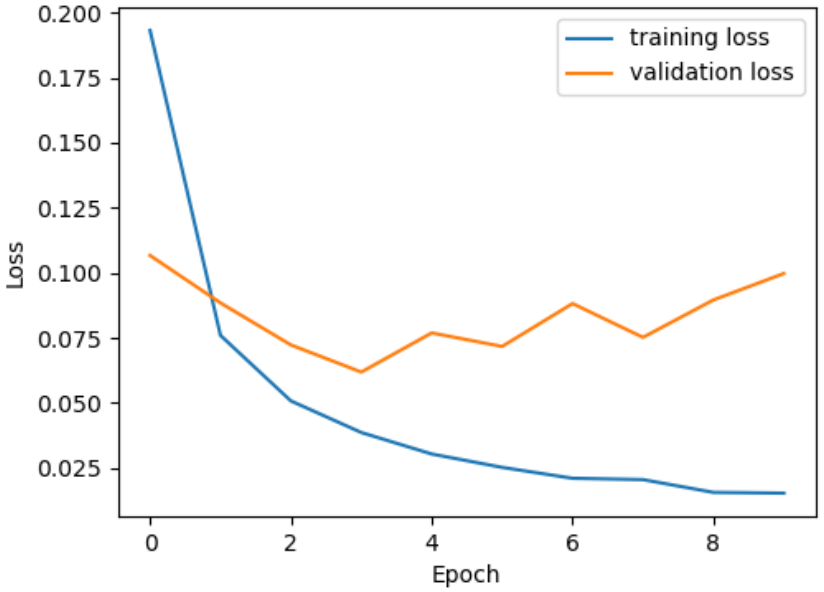



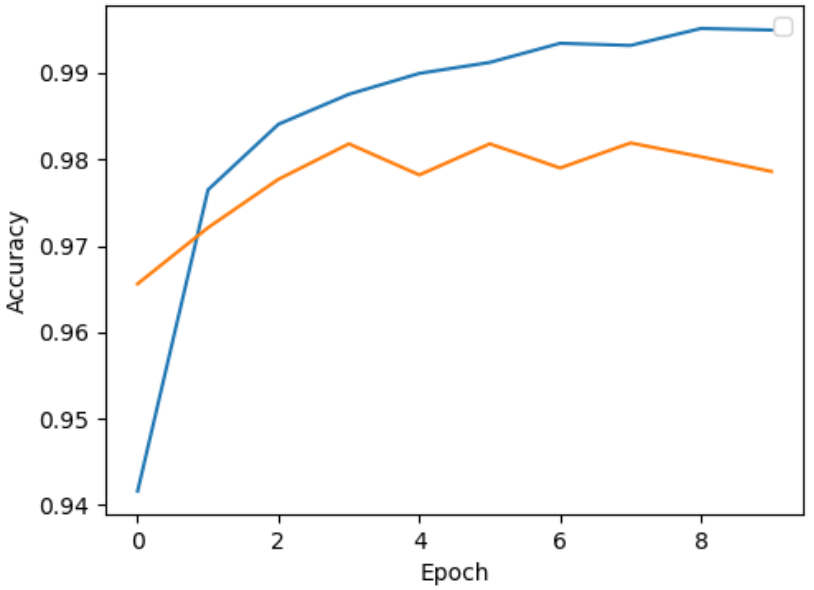

- Don't want the NN to rely heavily on individual features
- The neural net learns features of the data which are not existing. To prevent this reduce the number of nodes or refine the model.
- There are methods to reduce the number input nodes nodes:
  `tf.keras.layers.Dropout(p)`
  p = 0.5 half of the input nodes are dropped.

# Examples - Deep Neural Networks

- Overtraining: We observe a difference in loss and accuracy in training and test data




General idea: When multiple models describe the training data choose the simplest.

Add penalty terms to the loss function → L2 regularization $J_\beta(w) = J(w) + \beta|w|^2$

$$w = w - \alpha\nabla_w J = w - \alpha\nabla_w(J + \beta w^2) = (1 - 2\beta)w - \alpha\nabla_w J$$

weight decay

With `kernel_regularizer=tf.keras.regularizers.l2(0.001)` L2 regularization can be added to the layer which applies a penalty term to the loss function of the model, based on the squared magnitude of the weights of the layer.

Both methods can be switched on in the example 03_ml_basics_tf_mlp_mnist_digits.ipynb

# Exercise 4

Use the fashion mnist dataset and read it in tensorflow. Learn the different clothing categories.

- reshape the data to a 1D array and normalize it
- set up an MLP with 10 output categories
- plot the loss function and accuracy
- Do you see overfitting? Try to get rid of overfitting effects

03_ml_basics_ex_4_mlp_clothing.ipynb