Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Introduction to Deep Learning with Keras

PO&DAS 2023, Hamburg

Karim El Morabit (UHH) (karim.el.morabit@cern.ch)

Adapted from the tutorial by: Lisa BENATO, Patrick L.S. CONNOR, Dirk KRÜCKER, Mareike MEYER, Teddy BEAR

Reminder

HTTP://CERN.CH/GO/D9BT

IT'S EVERYONE'S RESPONSIBILITY TO:

Maintain a professional environment in an atmosphere of tolerance and mutual respect.

Abstain from all forms of harassment, abuse, intimidation, bullying and mistreatment of any kind.

This includes intimidation, sexual or crude jokes or comments, offensive images, and unwelcome physical conduct.

Keep in mind that behaviour and language deemed acceptable to one person may not be to another.

Help our community adhere to the code of conduct and speak up when you see possible violations.

Outline

00 Introduction (10 mins)
- Machine learning in a nutshell
- Scope of the lab
- Setting up environment

01 Regression (50 mins)
- References
- What is machine learning
- Build a neural network
- Activation function
- TensorFlow & Keras
- Building a network with Keras
- Loss function
- Example
- Optimiser & compilation
- Training & learning curve
- Predicting

02 Classification (55 mins)
- Classification (vs. regression)
- Loss function: *cross-entropy*
- Example
- Validation sample
- Learning curves & accuracy
- Multi-class classification
- Fashion-MNIST
- Methods to prevent over-training

03 Summary & Conclusions (5 mins)
- What we have (not) covered
- Plans for the second lab

04 Back-up

Introduction

The concept of Machine Learning in a nutshell

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Typical problems in science & technology

Modelling, diagnosis/classification, pattern recognition, task automation, ...

Example: Modelling a physics phenomenon

1. Write a model with limited number of parameters $\hat{y} = f(x|p)$
2. Measure data x_i, y_i
3. Fit parameters

Examples
- Predict the temperature based on yesterday's weather
- Classify astrophysical objects
- Identifty particles at colliders
- Diagnose diseases
- Speech and face recognition
- ...

Advantages
Better understanding and control.
Parameters may have physical meaning.

Drawbacks
You need to have an idea for a model.
Difficult when many parameters are involved.

The concept of Machine Learning in a nutshell

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Typical problems in science & technology

Modelling, diagnosis/classification, pattern recognition, task automation, ...

Example: Modelling a physics phenomenon

1. *Construct a network with a large number of parameters (or weights)*
2. Measure data
3. Fit parameters

$$\hat{y} = f(x|p)$$

$$x_i, y_i$$

Machine learning means that you don't need to come up with a model: the machine will learn how to interpret the data for you

Supervised learning means that you have a measurement to test your predictions

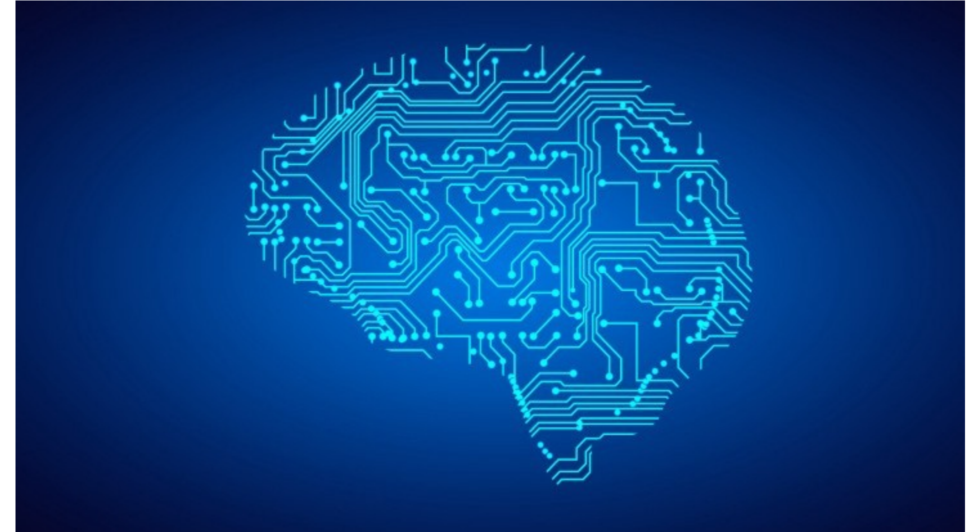
Advantages
Catch effects difficult to model.
Come up without any precise idea.

Drawbacks
Fit is not straightforward.
No direct interpretation.

Scope of this exercise

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- Machine learning is a huge field
- We only have a very limited amount of time
- We are only going to talk about:
  - (Supervised) Deep learning
  - Regression & classification
  - Loss function, optimiser, learning curve, learning rate, …
  - How to do this with TensorFlow & Keras
- There are many excellent resources available to learn about other aspects or methods

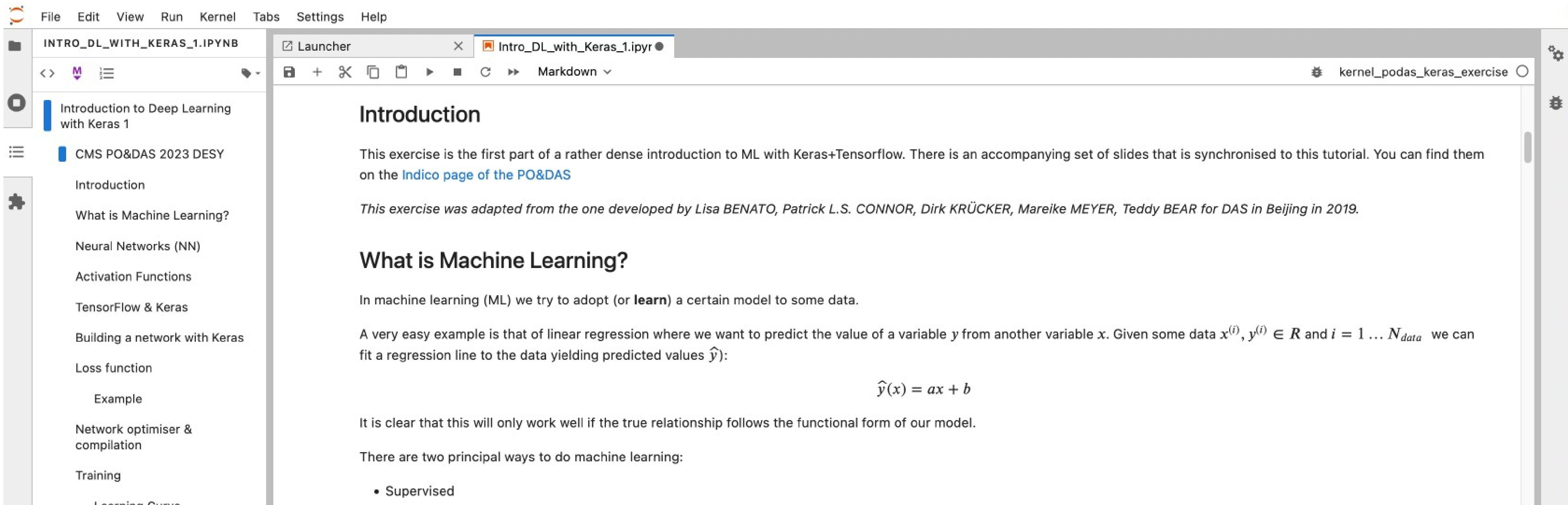


We here emphasise on practical aspects of machine learning

Practical Aspects

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- For the practical exercises go to https://gitlab.cern.ch/cms-podas23/topical/keras-exercise
- Follow the instructions in the README
- These slides and the exercise notebooks are synchronised and should be followed together



Part 1 - Regression

What is machine learning?

$$f(x) = \sum a_n x^n$$

$$f(x) = \sum \left(A_n \cos nx + B_n \sin nx \right)$$

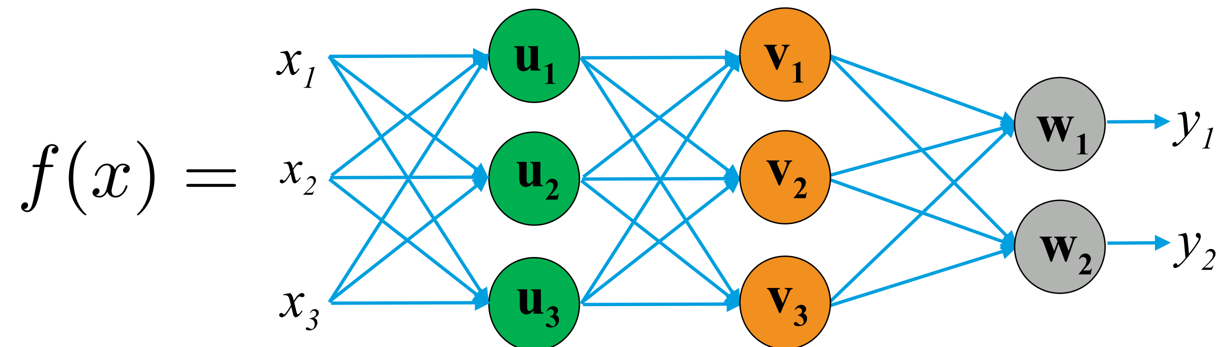
Machine learning means that you don't need to come up with a model: the machine will learn how to interpret the data for you

What is machine learning?

$$f(x) = \sum a_n x^n$$

$$f(x) = \sum \Big(A_n \cos nx + B_n \sin nx \Big)$$

$$f(x) =$$

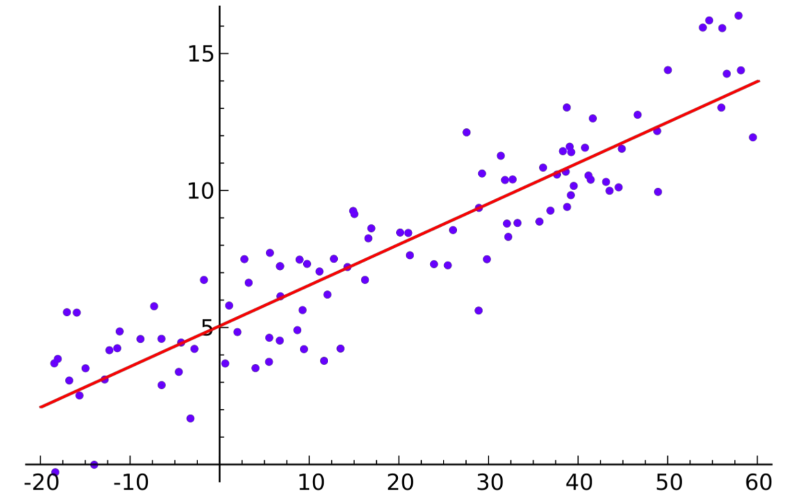


Machine learning means that you don't need to come up with a model: the machine will learn how to interpret the data for you

Neural network = Connected neurons/nodes

What is machine learning?

UHH Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



Starting point: *Linear Regression*

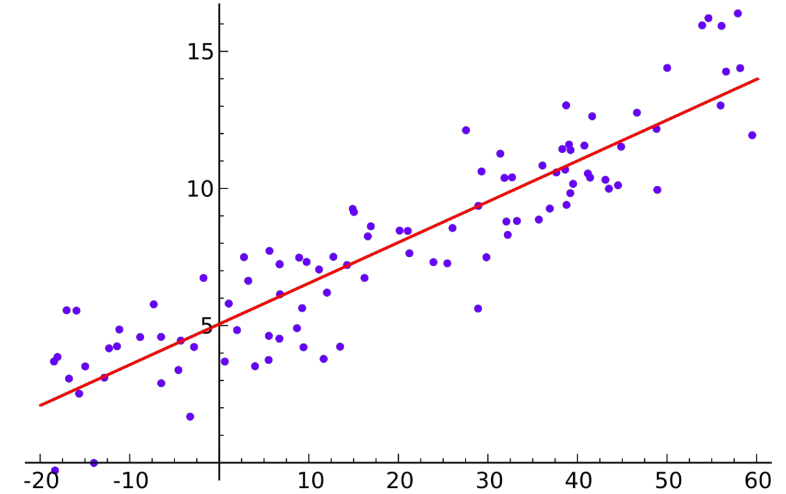
Data points $x^i, y^i \in \mathbb{R}, i = 1, \ldots, N_{\text{data}}$

Model $\hat{y} = ax + b$

→ Minimise *loss function* $\min_{a,b} \sum_{i=0}^{N_{\text{data}}} \left(\hat{y}(x_i) - y_i\right)^2$

What is machine learning?

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



Starting point: *Linear Regression*

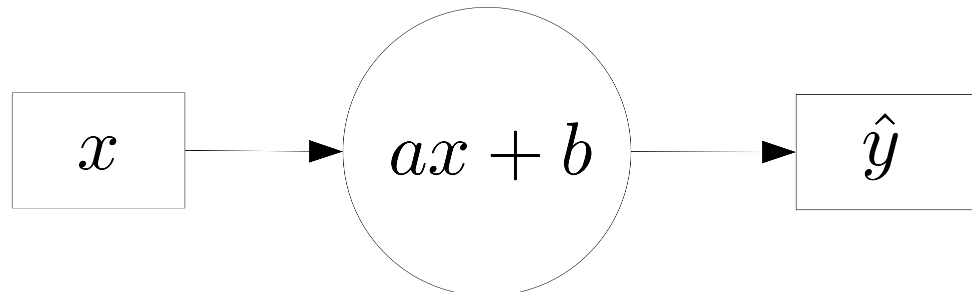
Data points $x^i, y^i \in \mathbb{R}, i = 1, ..., N_{\text{data}}$

Model $\hat{y} = ax + b$

→ Minimise *loss function* $\min_{a,b} \sum_{i=0}^{N_{\text{data}}} \left(\hat{y}(x_i) - y_i\right)^2$

In Machine Learning, we have **many (many) more parameters...**

Our first node/neuron:
(or almost)



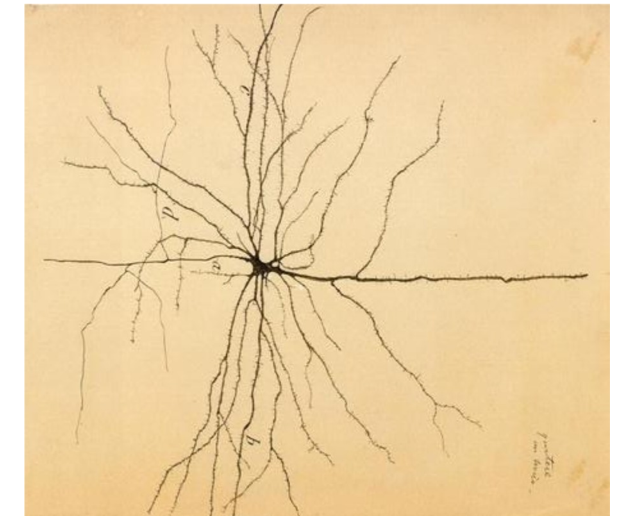
What is machine learning?

Brain = neural network*

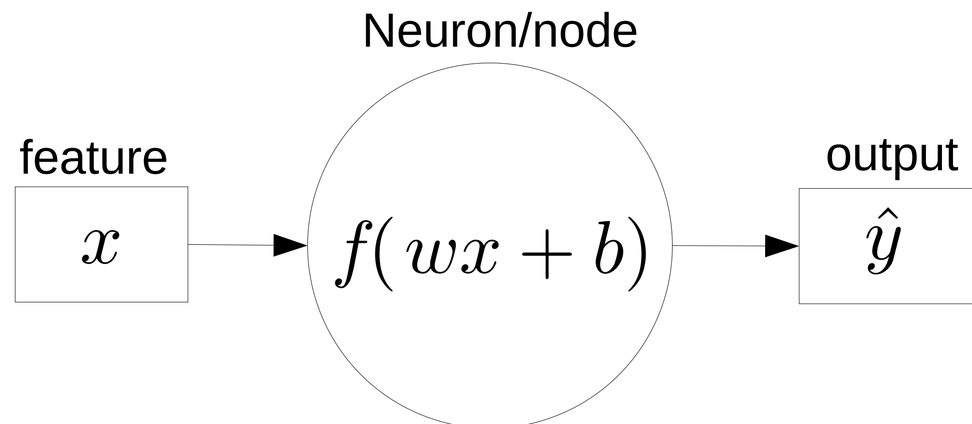
*simplistic model for real neuron

Neuron = *Linear Regression + Activation function*

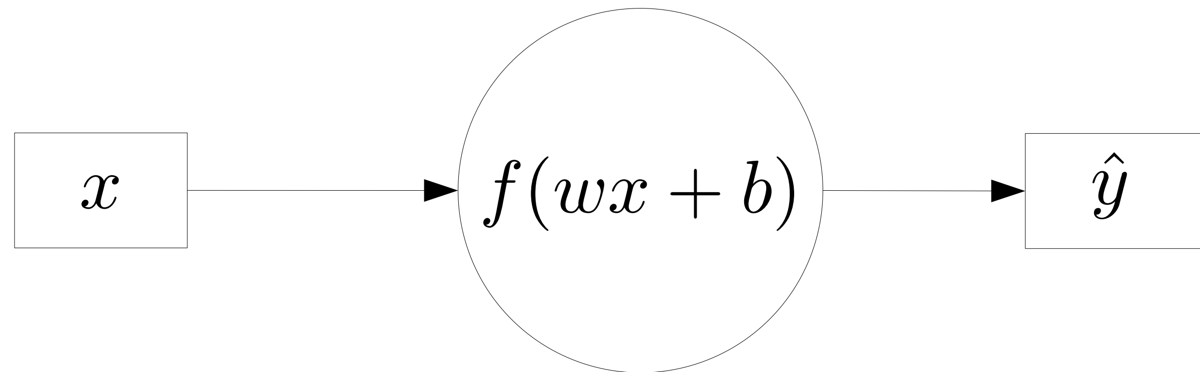
- **Generalisation** from scalar to **vectors** is straightforward
- **Neurons/nodes** are **connected** into a neural network
- The **activation function** introduces a **non-linearity**



Intelligence arises from the connections

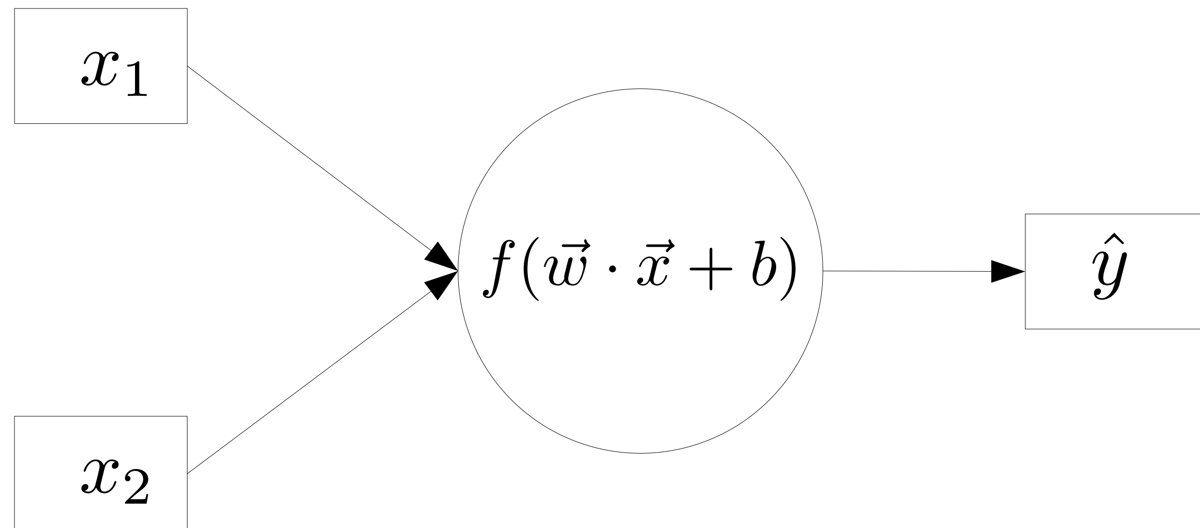


Building a Neural Network



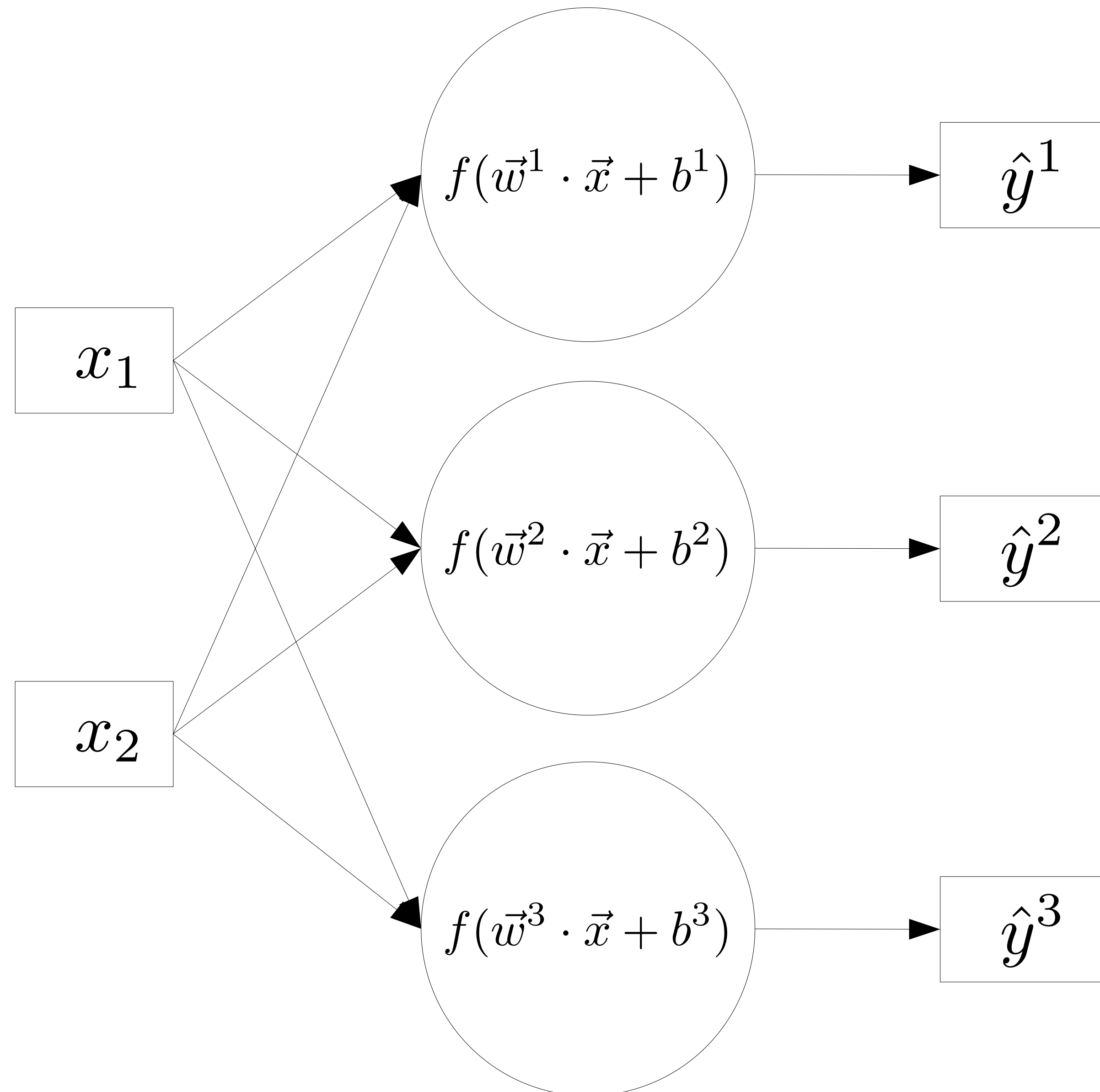
$$\hat{y} = f(wx+b)$$

Building a Neural Network



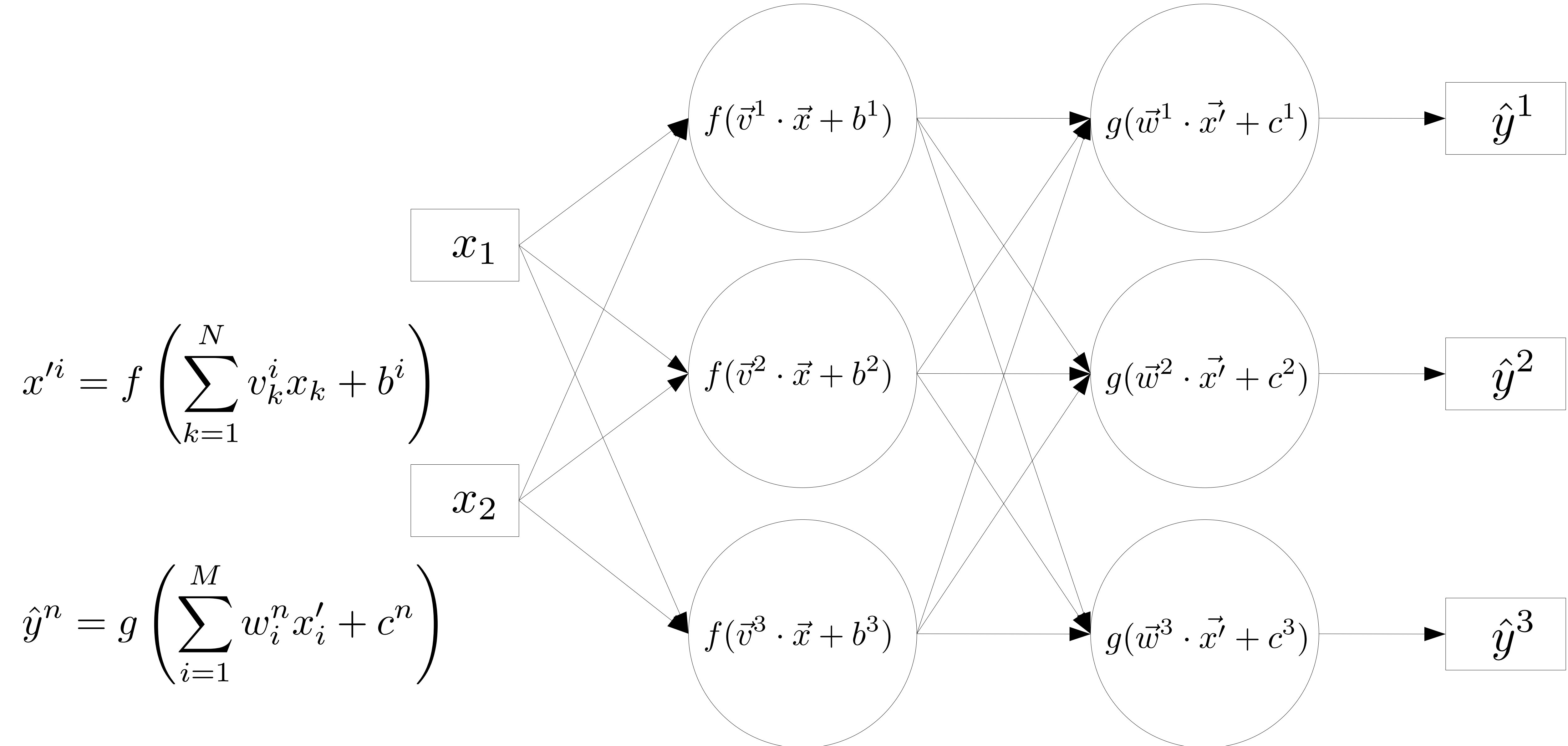
$$\hat{y} = f\left(\sum_{k=1}^{N} w_k x_k + b\right)$$

Building a Neural Network



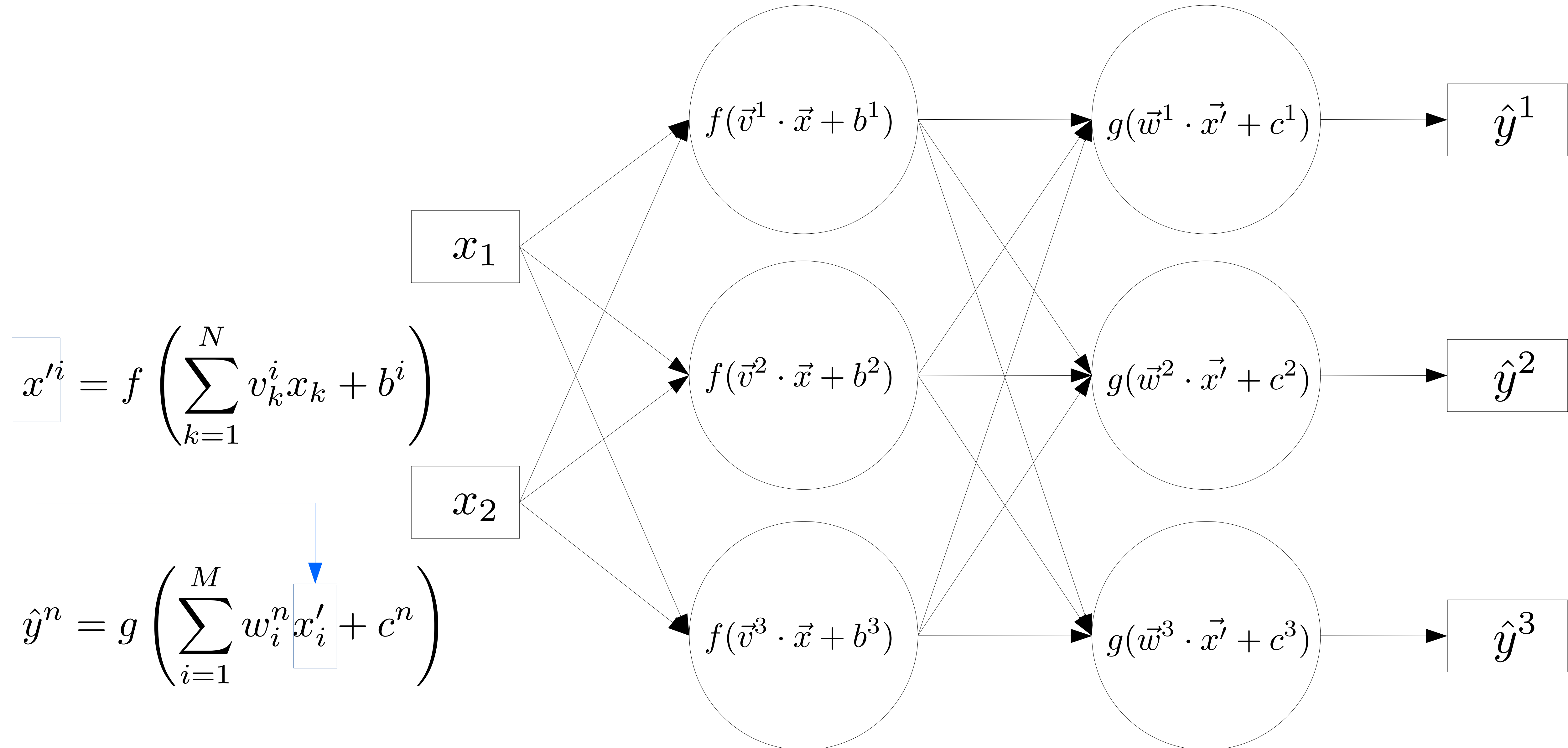
$$\hat{y}^i = f\left(\sum_{k=1}^{N} w_k^i x_k + b^i\right)$$

Building a Neural Network

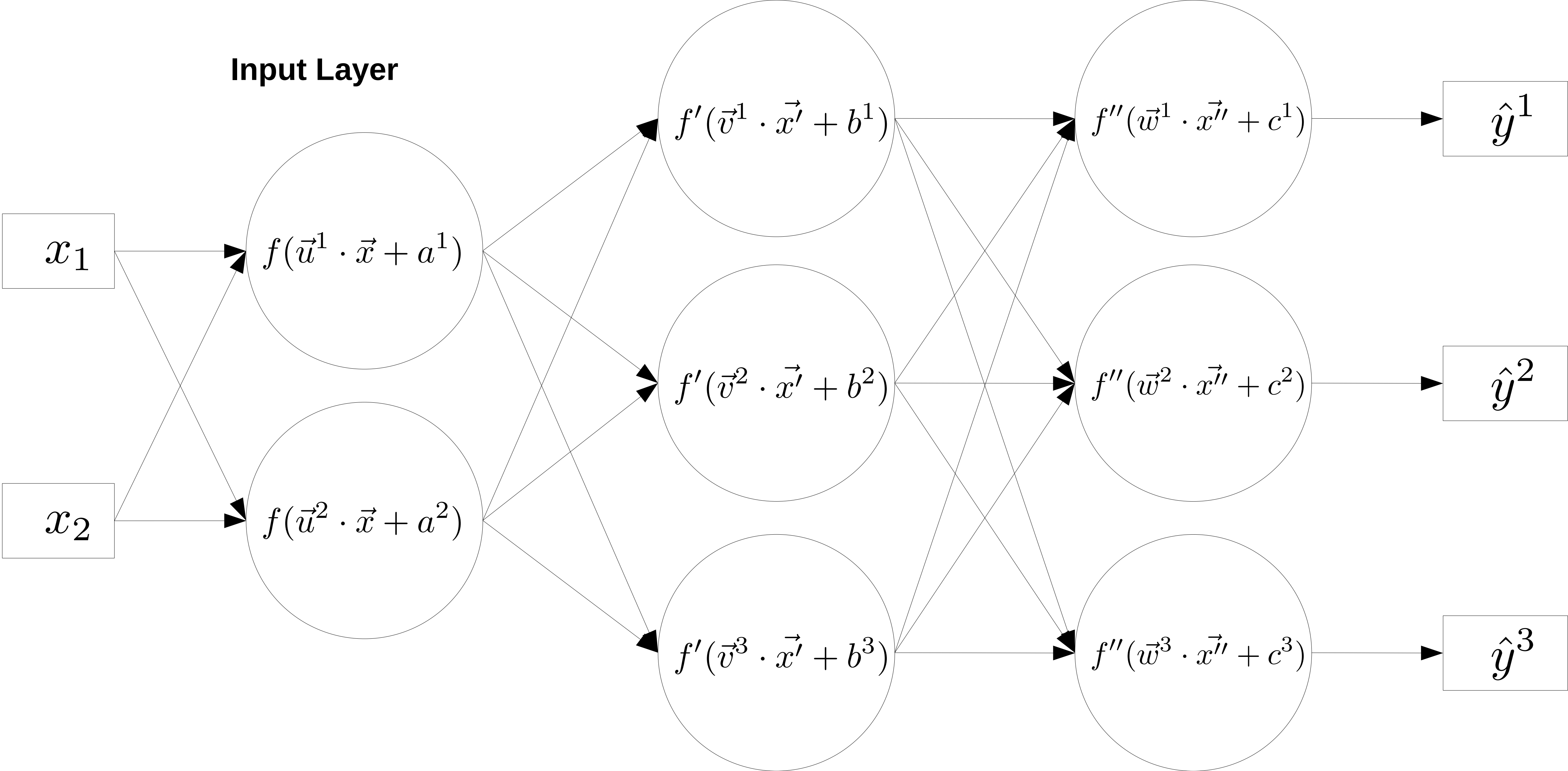


Building a Neural Network

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

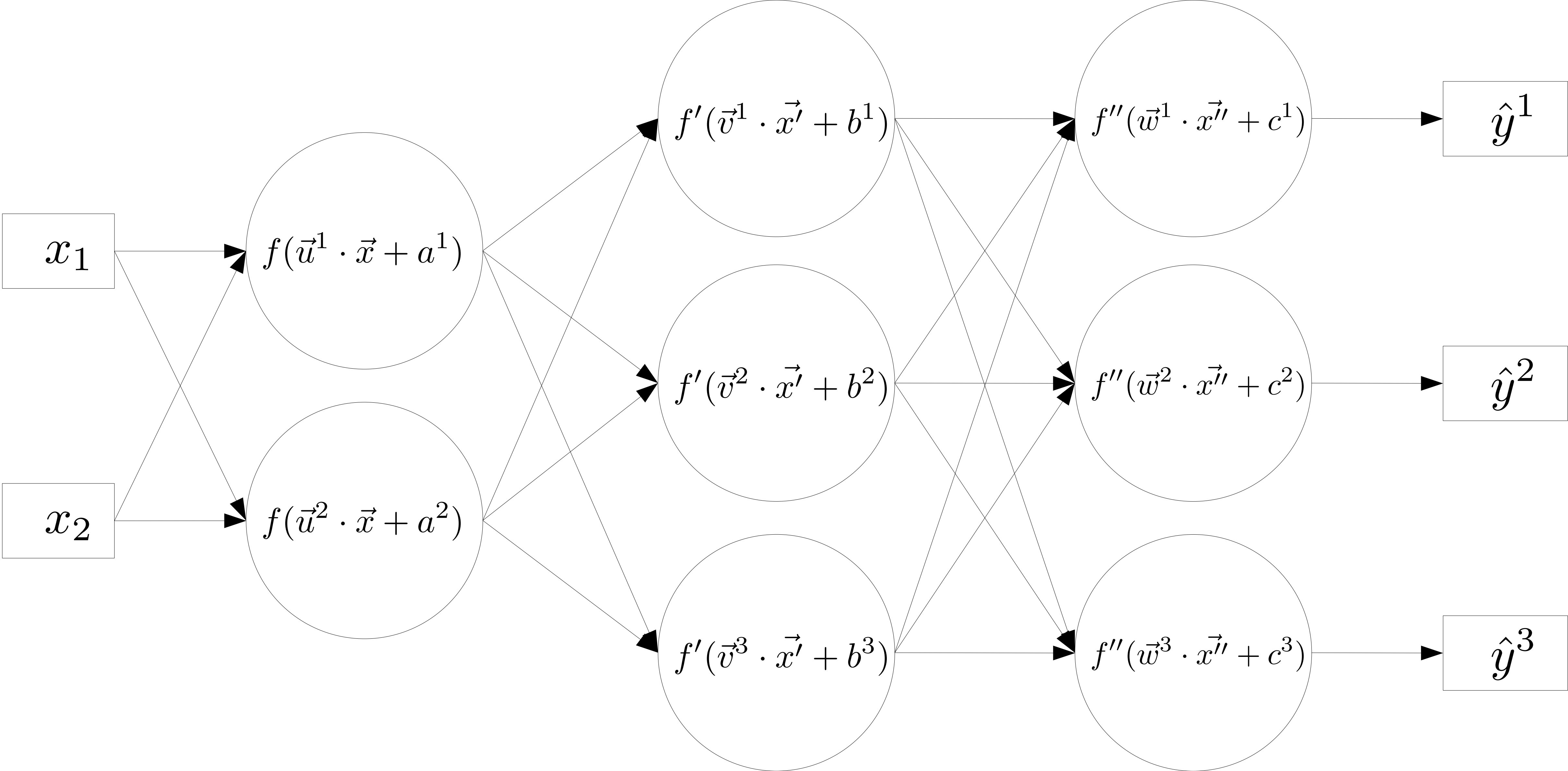


Building a Neural Network



UHH Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Building a Neural Network



Matri

- Equa

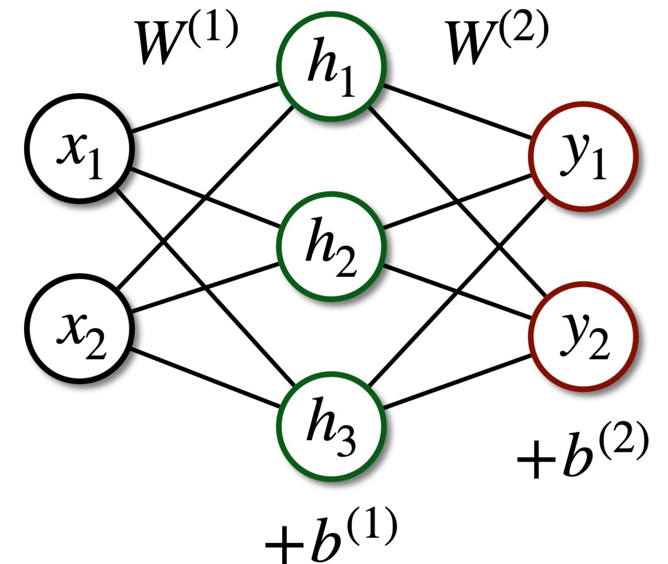
  - $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$

- Capture higher order correlations with multiple layers
- But: Without activation function, the output of the whole network is still a linear mapping

$$\vec{h} = W^{(1)}\,\vec{x} + \vec{b}^{(1)}$$

$$\vec{y} = W^{(2)}\,\vec{h} + \vec{b}^{(2)}$$

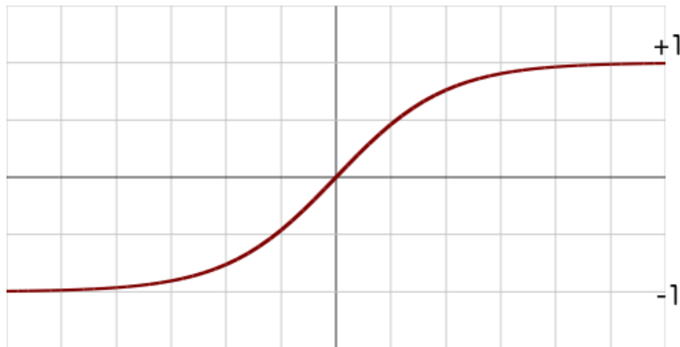
$$\vec{y} = \underbrace{W^{(2)}W^{(1)}}_{\text{W}}\,\vec{x} + \underbrace{W^{(1)}\vec{b}^{(1)} + \vec{b}^{(2)}}_{\text{b}}$$



Activation functions

The activation function introduces a *non-linearity* at each layer

Hyperbolic tangent



$$\hat{y} = \tanh x$$

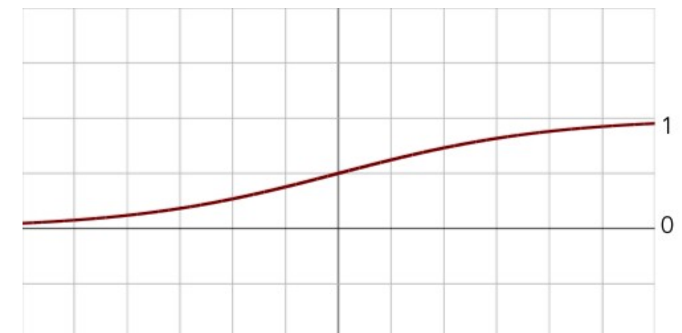
Used to be popular

Rectified **L**inear **U**nit (ReLU)

$$\hat{y} = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

Currently most popular for regression

Logistic function



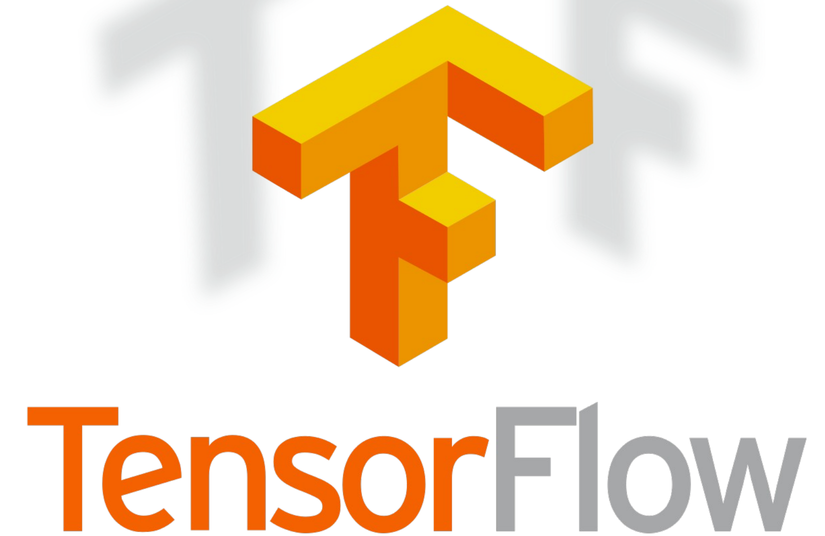
$$\hat{y} = \frac{1}{1 + \exp\left(-x\right)}$$

Important in classification (see Part Two)

Tensorflow & Keras

TensorFlow (or TensorFly)

- One standard library
- Powerful (*i.e.* complicate) language
- Other examples: PyTorch (Facebook), MXNet (Apache), Theano, ...



Keras

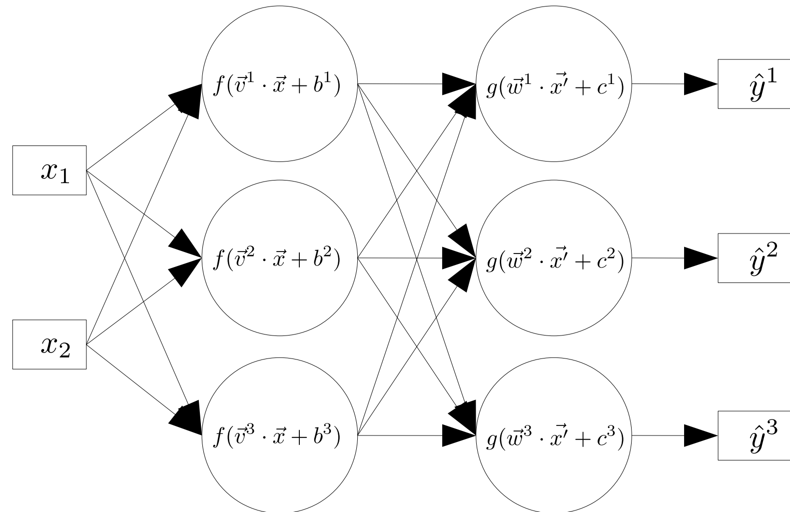
- Wrapper for TensorFlow in Python
- Similar language to NumPy
- Originally independent, now also integrated to TensorFlow package

```python
# We take the keras implementation from tensorflow
from tensorflow import keras
from tensorflow.keras import layers,models
```

In this tutorial, we will use the version of Keras included in TensorFlow

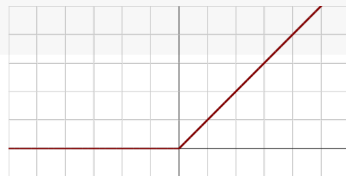
Building a network with Keras

$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$



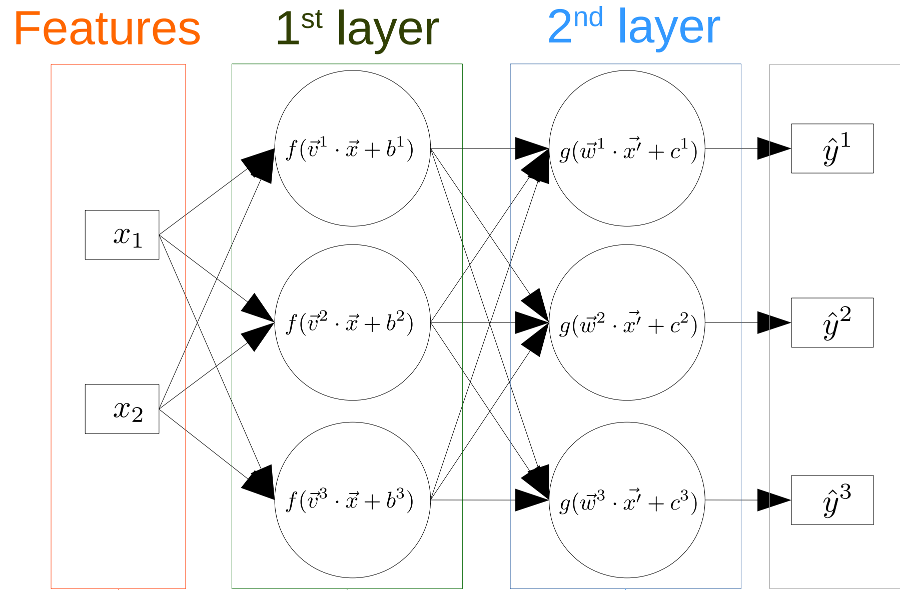

```
model = models.Sequential([
    layers.Dense(20, activation='relu', input_dim=10),
    layers.Dense(30, activation='relu')]
)
```

"*Dense*" means
that all nodes of a layer
are connected
to all nodes of the next layer

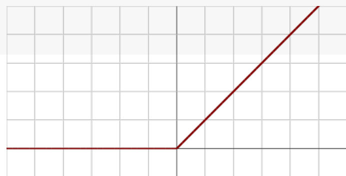


Building a network with Keras

$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$



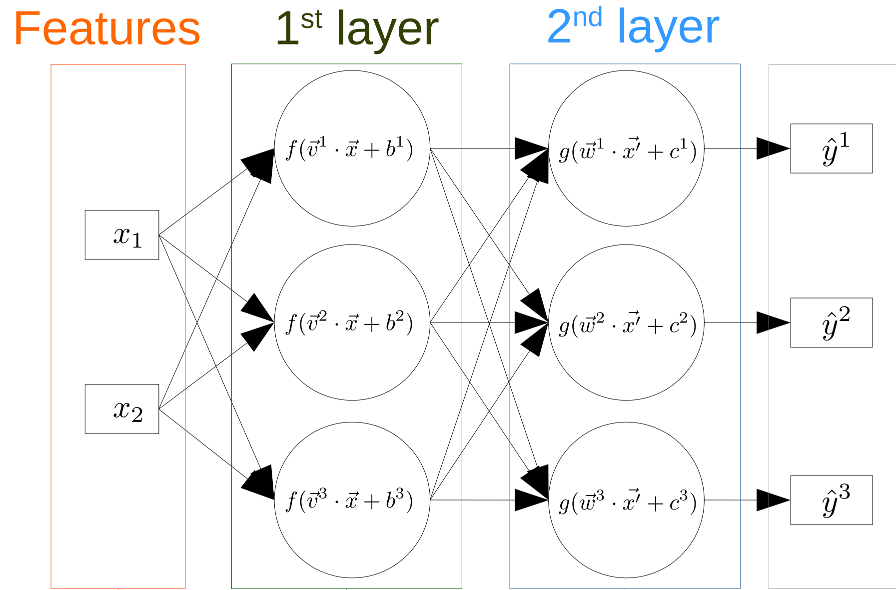

```
model = models.Sequential([
    layers.Dense(20, activation='relu', input_dim=10),
    layers.Dense(30, activation='relu')]
)
```



"*Dense*" means
that all nodes of a layer
are connected
to all nodes of the next layer

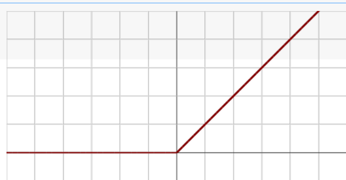
Building a network with Keras

$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$



- 10 features
- 20 nodes in the first layer
- 30 nodes in the second layer

```
model = models.Sequential([
    layers.Dense(20, activation='relu', input_dim=10),
    layers.Dense(30, activation='relu')]
)
```



"*Dense*" means that all nodes of a layer are connected to all nodes of the next layer

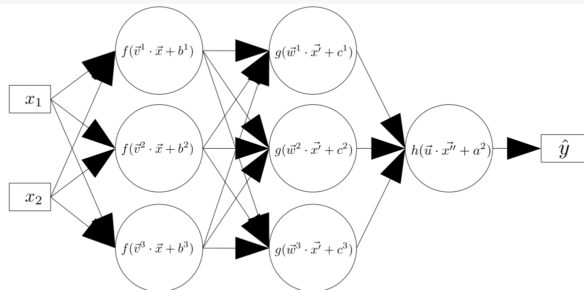
Q: how to count the number of parameters?

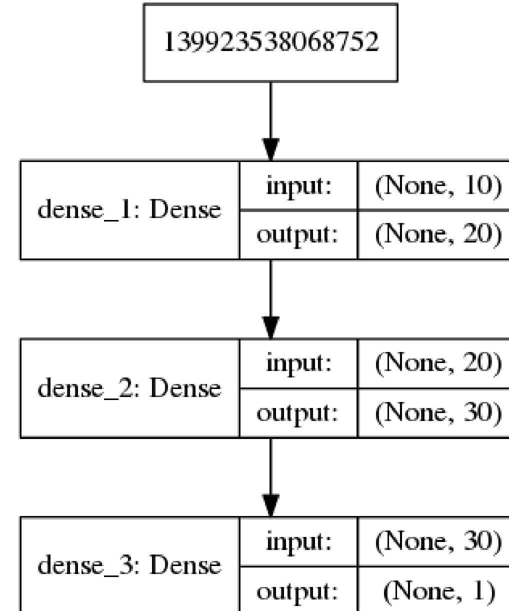
Counting the model parameters

$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$

```
_________________________________________________________________
Layer (type)                 Output Shape              Param
=================================================================
dense_1 (Dense)              (None, 20)                220
_________________________________________________________________
dense_2 (Dense)              (None, 30)                630
=================================================================
Total params: 850
Trainable params: 850
Non-trainable params: 0
_________________________________________________________________
```

```
model = models.Sequential([
    layers.Dense(20, activation='relu', input_dim=10),
    layers.Dense(30, activation='relu')]
)
```





Counting the model parameters

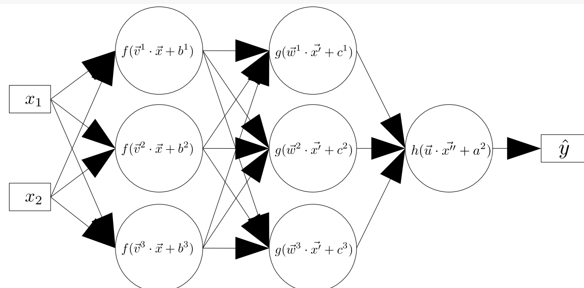
$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$

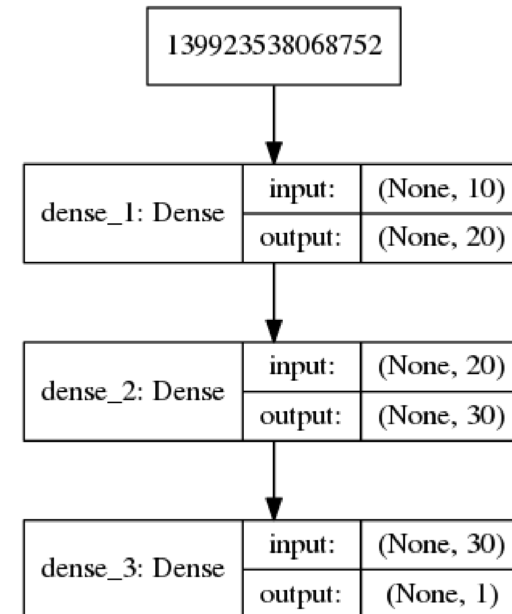
```
Layer (type)                 Output Shape              Param
=================================================================
dense_1 (Dense)              (None, 20)                220
_________________________________________________________________
dense_2 (Dense)              (None, 30)                630
=================================================================
Total params: 850
Trainable params: 850
Non-trainable params: 0
_________________________________________________________________
```

$(10 \text{ features} + 1 \text{ bias}) \times 20 \text{ nodes}$

$(20 \text{ features} + 1 \text{ bias}) \times 30 \text{ nodes}$

```
model = models.Sequential([
    layers.Dense(20, activation='relu', input_dim=10),
    layers.Dense(30, activation='relu')]
)
```





Counting the model parameters

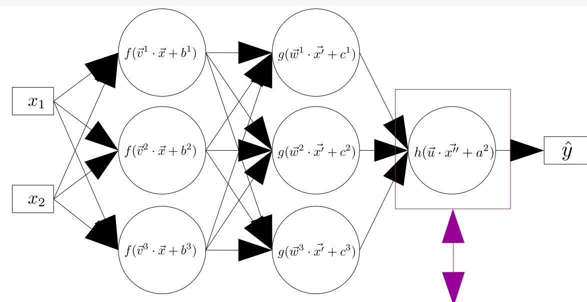
$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$

```
_________________________________________________________________
Layer (type)                 Output Shape              Param
=================================================================
dense_1 (Dense)              (None, 20)                220
_________________________________________________________________
dense_2 (Dense)              (None, 30)                630
=================================================================
Total params: 850
Trainable params: 850
Non-trainable params: 0
_________________________________________________________________
```

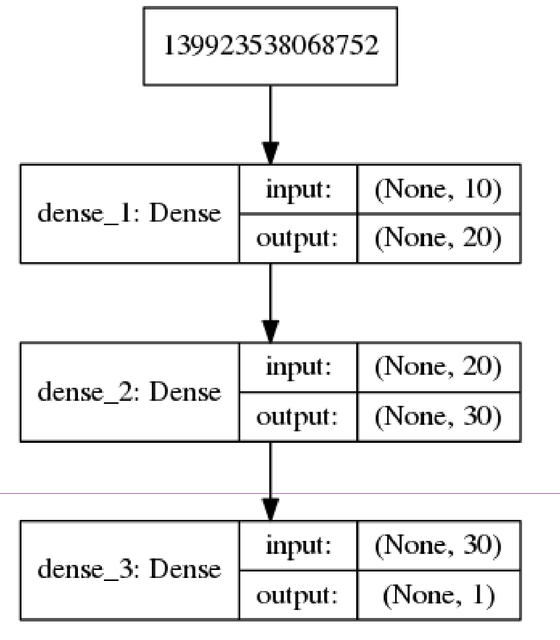
$(10 \text{ features} + 1 \text{ bias}) \times 20 \text{ nodes}$

$(20 \text{ features} + 1 \text{ bias}) \times 30 \text{ nodes}$

```
model = models.Sequential([
    layers.Dense(20, activation='relu', input_dim=10),
    layers.Dense(30, activation='relu')]
)
```



```
model.add(layers.Dense(1))
```



Training the model

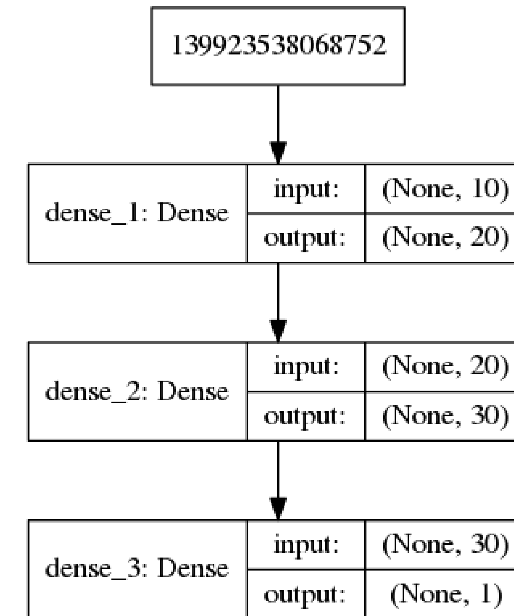
Reminder:
we are doing
supervised learning

Compare output of the model
with real (measured) values
and **minimise loss function**

Example: mean-square error (MSE)

$$\min_{w_i} \sum_{i=0}^{N_{\text{data}}} \left(\hat{y}(x_i) - y_i\right)^2$$

$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$



Training the model

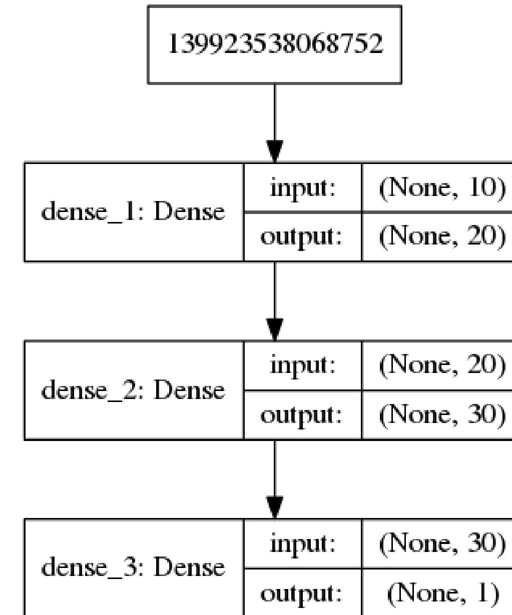
Reminder:
we are doing
supervised learning

Compare output of the model
with real (measured) values
and **minimise loss function**

Example: mean-square error (MSE)

$$\min_{w_i} \sum_{i=0}^{N_{\mathrm{data}}} \left(\hat{y}(x_i) - y_i\right)^2$$

$$\text{model} = \text{ReLu}\left[W_2 \cdot \text{ReLu}\left[W_1 \cdot x + b_1\right] + b_2\right]$$



- To train the network we also need some **labeled** data (data where we know the true output values)
- In HEP most often from MC simulation
- So let's simulate some data!

Toy example

Generate randomly 10k events of 10-tuples, such that

$$y = \sin \sum_{k=1}^{10} x_k^2$$

$$x_k \in [0, 1)$$

```python
import numpy as np
# We create 10000 random vectors each 10-dim
N_samples=10000
N_in=10
# A matrix N_samplesxN_in, uniform in [0,1)
x_train=np.random.rand(N_samples,N_in)
# Sum of squares along N_in
z = np.sum( np.square(x_train),axis=1)
y_train = np.sin(z)
```

Toy example

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Generate randomly 10k events of 10-tuples, such that

$$y = \sin \sum_{k=1}^{10} x_k^2$$

$$x_k \in [0, 1)$$

```
import numpy as np
# We create 10000 random vectors each 10-dim
N_samples=10000
N_in=10
# A matrix N_samplesxN_in, uniform in [0,1)
x_train=np.random.rand(N_samples,N_in)
# Sum of squares along N_in
z = np.sum( np.square(x_train),axis=1)
y_train = np.sin(z)
```

Toy example

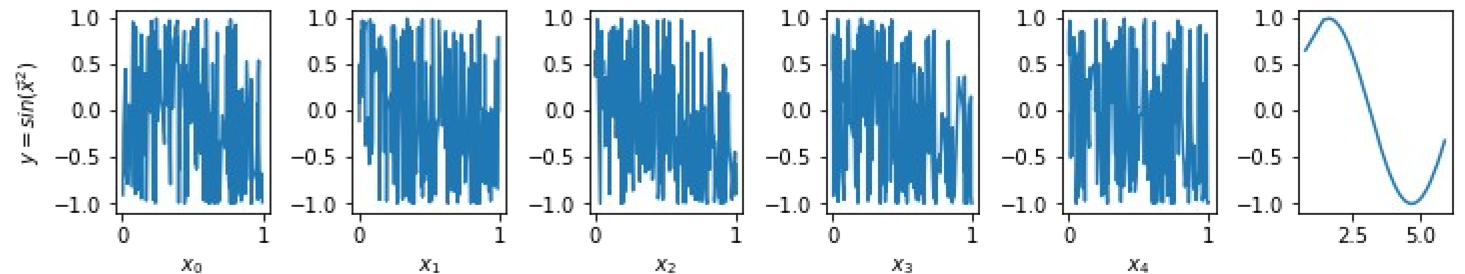
Generate randomly 10k events of 10-tuples, such that

$$y = \sin \sum_{k=1}^{10} x_k^2$$

$$x_k \in [0, 1)$$

```python
import numpy as np
# We create 10000 random vectors each 10-dim
N_samples=10000
N_in=10
# A matrix N_samplesxN_in, uniform in [0,1)
x_train=np.random.rand(N_samples,N_in)
# Sum of squares along N_in
z = np.sum( np.square(x_train),axis=1)
y_train = np.sin(z)
```

Looks quite random: **hidden structure**



Toy example

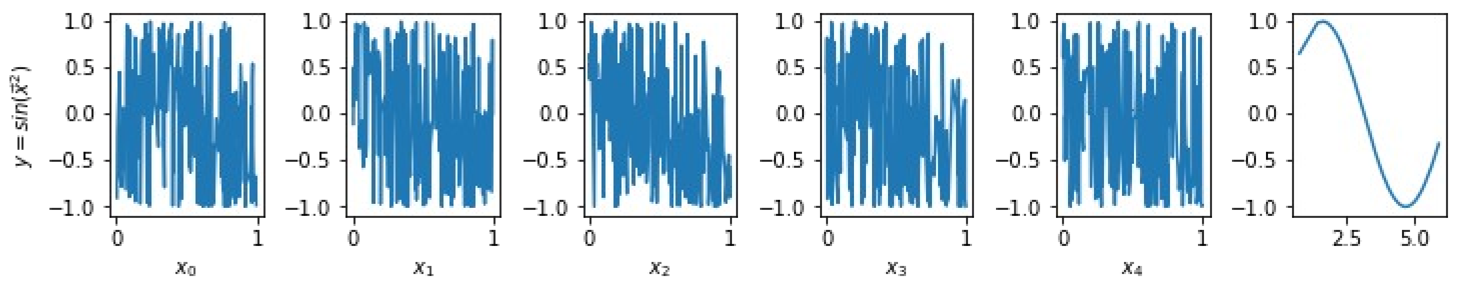
Generate randomly 10k events of 10-tuples, such that

$$y = \sin \sum_{k=1}^{10} x_k^2$$

$$x_k \in [0,1)$$

```
import numpy as np
# We create 10000 random vectors each 10-dim
N_samples=10000
N_in=10
# A matrix N_samplesxN_in, uniform in [0,1)
x_train=np.random.rand(N_samples,N_in)
# Sum of squares along N_in
z = np.sum( np.square(x_train),axis=1)
y_train = np.sin(z)
```

Looks quite random: **hidden structure**

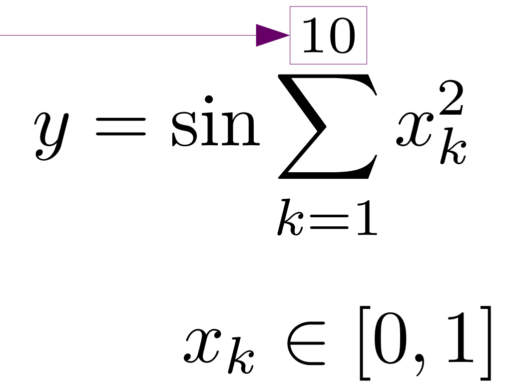


- Now we need a way to fit a large number of parameters

Network optimizer and compilation


```
from tensorflow.keras import models, layers, losses, optimizers
from time import time
model = models.Sequential(
    [
        layers.Dense(20, activation='relu', input_dim=10),
        layers.Dense(30, activation='relu'),
        layers.Dense(1)
    ]
)
optimizer = optimizers.SGD(lr=0.01)

model.compile(optimizer=optimizer,loss='mse')
```



$$y = \sin \sum_{k=1}^{10} x_k^2$$

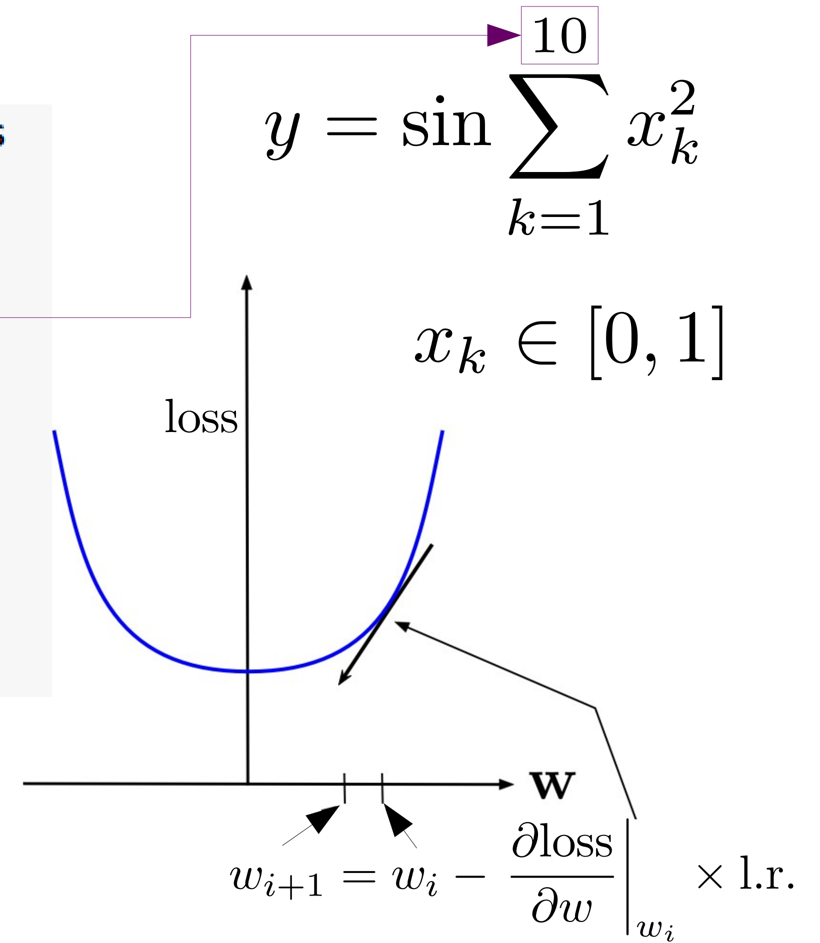
$$x_k \in [0, 1]$$

Network optimizer and compilation


```
from tensorflow.keras import models, layers, losses, optimizers
from time import time
model = models.Sequential(
    [
        layers.Dense(20, activation='relu', input_dim=10),
        layers.Dense(30, activation='relu'),
        layers.Dense(1)
    ]
)
optimizer = optimizers.SGD(lr=0.01)

model.compile(optimizer=optimizer,loss='mse')
```

Optimiser: Stochastic Gradient Descent



- The *minimisation* of so many parameters is performed *iteratively* (“**descent**”)
- Use the **gradient** of the loss function w.r.t. NN parameters
- **Stochastic** means that only a subsample of the data is used at each iteration

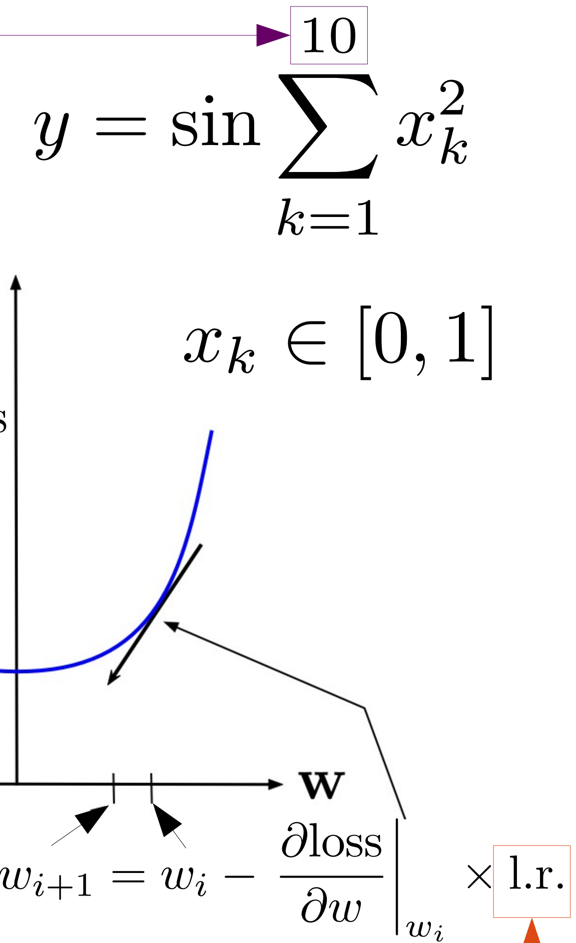
Network optimizer and compilation


```
from tensorflow.keras import models, layers, losses, optimizers
from time import time
model = models.Sequential(
    [
        layers.Dense(20, activation='relu', input_dim=10),
        layers.Dense(30, activation='relu'),
        layers.Dense(1)
    ]
)
optimizer = optimizers.SGD(lr=0.01)

model.compile(optimizer=optimizer,loss='mse')
```

Optimiser: Stochastic Gradient Descent

Learning rate



- The *minimisation* of so many parameters is performed *iteratively* (“**descent**”)
- Use the **gradient** of the loss function w.r.t. NN parameters
- **Stochastic** means that only a subsample of the data is used at each iteration

Network optimizer and compilation

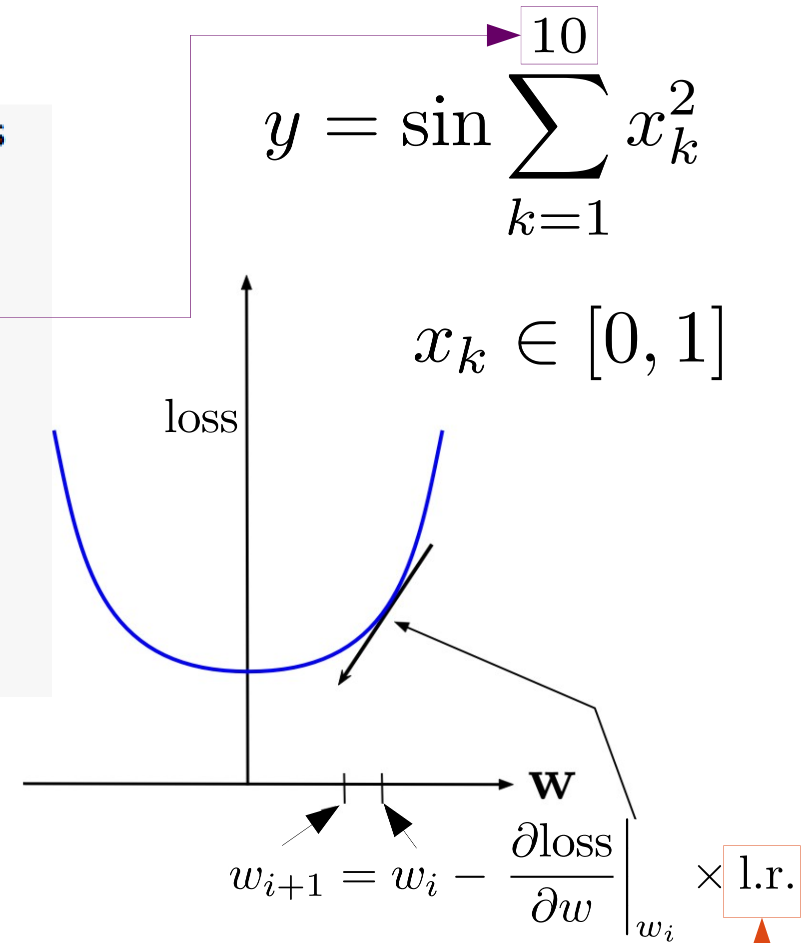

```
from tensorflow.keras import models, layers, losses, optimizers
from time import time
model = models.Sequential(
    [
        layers.Dense(20, activation='relu', input_dim=10),
        layers.Dense(30, activation='relu'),
        layers.Dense(1)
    ]
)
optimizer = optimizers.SGD(lr=0.01)

model.compile(optimizer=optimizer,loss='mse')
```

Optimiser: **S**tochastic **G**radient **D**escent

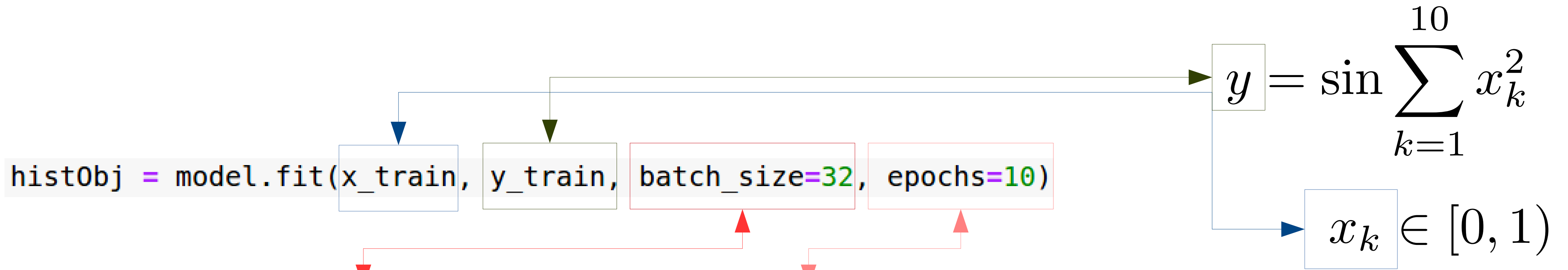
Compile the network before fitting anything

Learning rate



- The *minimisation* of so many parameters is performed *iteratively* (“**descent**”)
- Use the **gradient** of the loss function w.r.t. NN parameters
- **Stochastic** means that only a subsample of the data is used at each iteration

Training and the loss curve



Batch size

Size of the sample used by the optimiser

Number of epochs

Number of times that the whole sample will be used

Monitor the training

Learning rate

Internal parameter

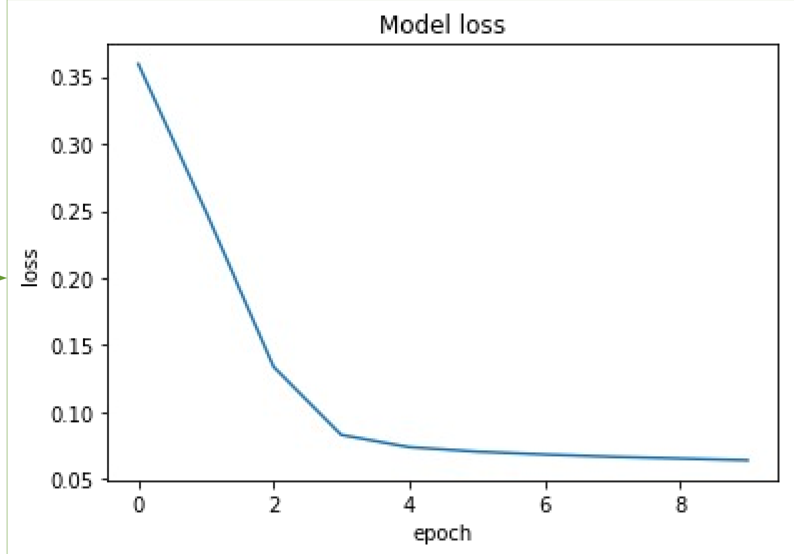
- Too small = slow convergence
- Too high = may diverge

```
optimizer = optimizers.SGD(lr=0.01)
```

Learning Curve

Loss value along epochs

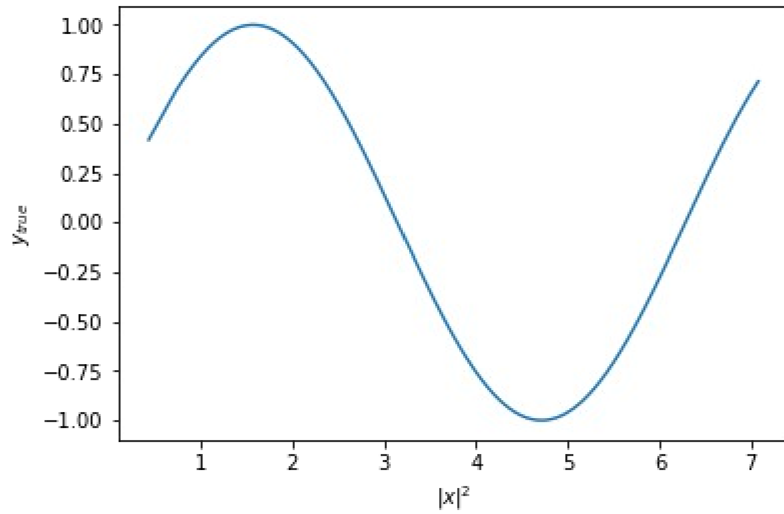
(see also Part Two!)

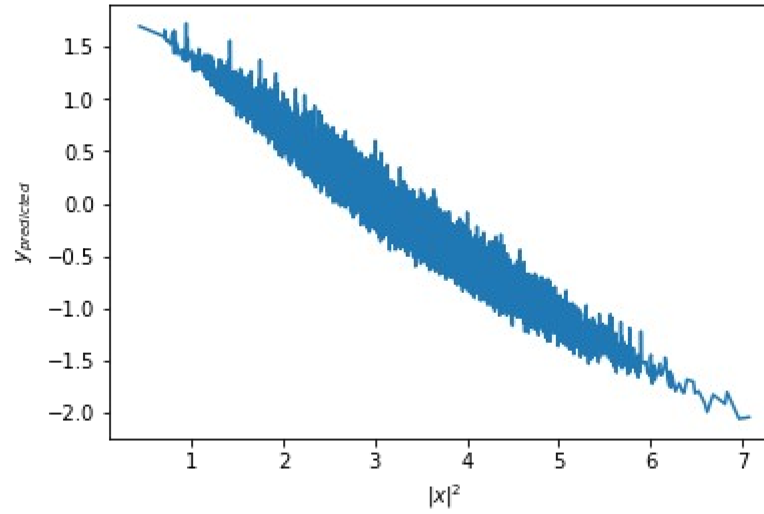


How to get a good prediction?

$$y = \sin \sum_{k=1}^{10} x_k^2$$

$$x_k \in [0, 1)$$





Deep Neural Network
- Number of layers
- Number of nodes for each layer
- Activation function(s)

Training
- Sample size
- Batch size
- Optimiser
- Number of epochs

Play with the parameters...

Try to use the Adam optimiser...

Part 2 - Classification

Classification

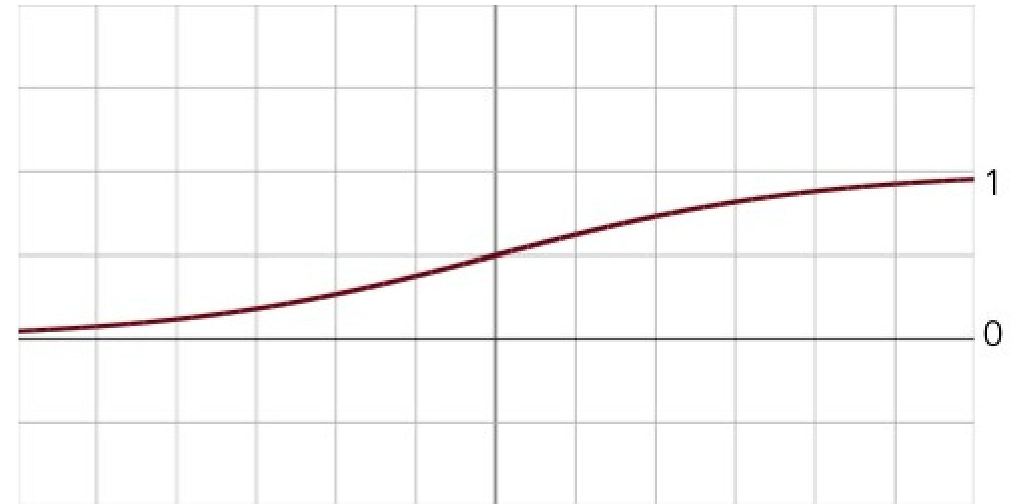
1. Discrete class labels $y_k = 0, 1$
2. Probabilist prediction $\hat{y} \in [0, 1]$

Activation function
For last node/layer:
Logistic function

$$f(x) = \frac{1}{1 + \exp(-x)}$$

Binary* classification:

Last layer must be made of a single node with output between 0 and 1



* we discuss later on multi-classification

Loss function: cross entropy

Jacob Bernoulli 1654-1705

1. Discrete class labels $y_k = 0, 1$
2. Probabilist prediction $\hat{y} \in [0, 1]$

Bernoulli trial $p, q \,|\, p + q = 1$

→ two possible outcomes

$$\mathcal{L} = \prod_{k \text{ in class } 1}^{n_{\text{batch}}} \hat{y}(x_k) \cdot \prod_{k \text{ in class } 0}^{n_{\text{batch}}} (1 - \hat{y}(x_k))$$

→ *Cross entropy* is found from this likelihood

Reminder

The batch corresponds to the data subsample used at each iteration to minimise the loss function

Loss function: cross entropy

Jacob Bernoulli 1654-1705

1. Discrete class labels $y_k = 0, 1$
2. Probabilist prediction $\hat{y} \in [0, 1]$

Bernoulli trial
→ two possible outcomes $p, q \,|\, p + q = 1$

Reminder

The batch corresponds to the data subsample used at each iteration to minimise the loss function

$$\mathcal{L} = \prod_{k}^{n_{\text{batch}}} \Big(y_k \cdot \hat{y}(x_k) + \left(1 - y_k\right) \cdot \left(1 - \hat{y}(x_k)\right) \Big)$$

normalisation

Cross-entropy

$$\rightarrow -\frac{1}{n_{\text{batch}}} \log \mathcal{L} = -\frac{1}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \Big(y_k \log\left(\hat{y}(x_k)\right) + \left(1 - y_k\right) \log\left(1 - \hat{y}(x_k)\right) \Big)$$

Loss function: cross entropy

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

$$\text{entropy} = \int p(x) \log\left(p(x)\right) dx$$

$$\text{cross-entropy} = \int q(x) \log\left(p(x)\right) dx$$

$$-\frac{1}{n_{\text{batch}}} \log \mathcal{L} = -\frac{1}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \left(y_k \log\left(\hat{y}(x_k)\right) + \left(1 - y_k\right) \log\left(1 - \hat{y}(x_k)\right) \right)$$

Loss function: cross entropy

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

$$\text{entropy} = \int p(x) \log\left(p(x)\right) dx$$

$$\text{cross-entropy} = \int q(x) \log\left(p(x)\right) dx$$

data

prediction

$$-\frac{1}{n_{\text{batch}}} \log \mathcal{L} = -\frac{1}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \left(y_k \log\left(\hat{y}(x_k)\right) + \left(1 - y_k\right) \log\left(1 - \hat{y}(x_k)\right) \right)$$

Loss function: cross entropy

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

$$\text{entropy} = \int p(x) \log\left(p(x)\right) dx$$

$$\text{cross-entropy} = \int q(x) \log\left(p(x)\right) dx$$

data

prediction

Gibbs inequality

Cross-entropy is minimal if $p = q$

Hence the name...

$$-\frac{1}{n_{\text{batch}}} \log \mathcal{L} = -\frac{1}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \left(y_k \log\left(\hat{y}(x_k)\right) + \left(1 - y_k\right) \log\left(1 - \hat{y}(x_k)\right) \right)$$

Toy Example


```python
import numpy as np
# We create 10000 random vectors each 10-dim
N_samples=10000
N_in=10
# A matrix N_samplesxN_in, uniform in [0,1)
x_train=np.random.rand(N_samples,N_in)
# Sum of squares along N_in
z = np.sum( np.square(x_train),axis=1)
y_train = (np.sin(z) >= 0).astype(np.int32)
```

$$\hat{y} = \begin{cases} 1 & \text{if } \sin\sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Recycle the example from Part One

$$-\frac{1}{n_{\text{batch}}}\log\mathcal{L} = -\frac{1}{n_{\text{batch}}}\sum_{k=1}^{n_{\text{batch}}}\Big(y_k\log\left(\hat{y}(x_k)\right) + (1-y_k)\log\left(1-\hat{y}(x_k)\right)\Big)$$

Toy Example

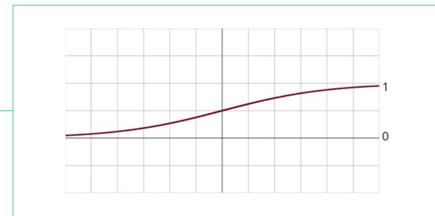
```python
import numpy as np
# We create 10000 random vectors each 10-dim
N_samples=10000
N_in=10
# A matrix N_samplesxN_in, uniform in [0,1)
x_train=np.random.rand(N_samples,N_in)
# Sum of squares along N_in
z = np.sum( np.square(x_train),axis=1)
y_train = (np.sin(z) >= 0).astype(np.int32)
```

$$\hat{y} = \begin{cases} 1 & \text{if } \sin \sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Recycle the example from Part One

```python
model = models.Sequential(
    [
        layers.Dense(100, activation='relu', input_dim=10),
        layers.Dense(100, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ]
)
optimizer = optimizers.Adam()
model.compile(optimizer=optimizer,loss='binary_crossentropy', metrics=['acc'])
```

Accuracy is used to Monitor the training



$$-\frac{1}{n_{\text{batch}}} \log \mathcal{L} = -\frac{1}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \Big(y_k \log \left(\hat{y}(x_k)\right) + (1 - y_k) \log \left(1 - \hat{y}(x_k)\right) \Big)$$

Validation sample

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

$$\hat{y} = \begin{cases} 1 & \text{if } \sin \sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               1100
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 11,301
Trainable params: 11,301
Non-trainable params: 0
```

```
model.fit(x_train, y_train, batch_size=256, epochs=25, validation_split=0.2)
```

Validation sample
Keep 20% of the data to control that the machine has modelled what it was supposed to model

Control **overfitting / overtraining**

Loss and accuracy

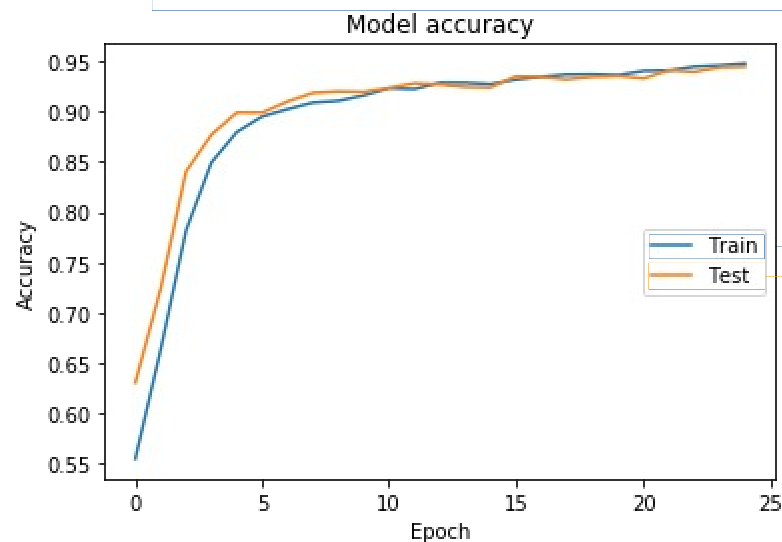
$$\text{accuracy} = \frac{\text{correctly classified data}}{\text{all data}}$$

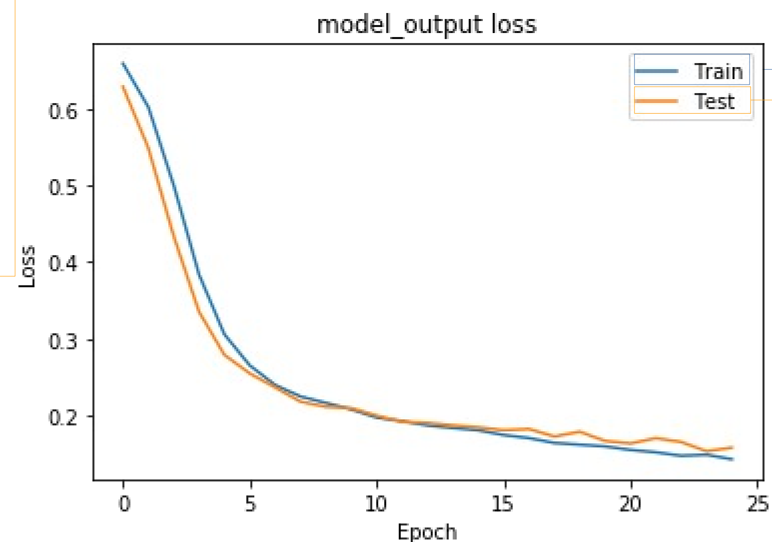
$$\hat{y} = \begin{cases} 1 & \text{if } \sin \sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

```
histObj.history.keys()
```

```
['acc', 'loss', 'val_acc', 'val_loss']
```

- With **too few** parameters, the NN may not be able to model the data.
- With **too many** parameters, the NN may start *memorising* the training data.





Loss and accuracy

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

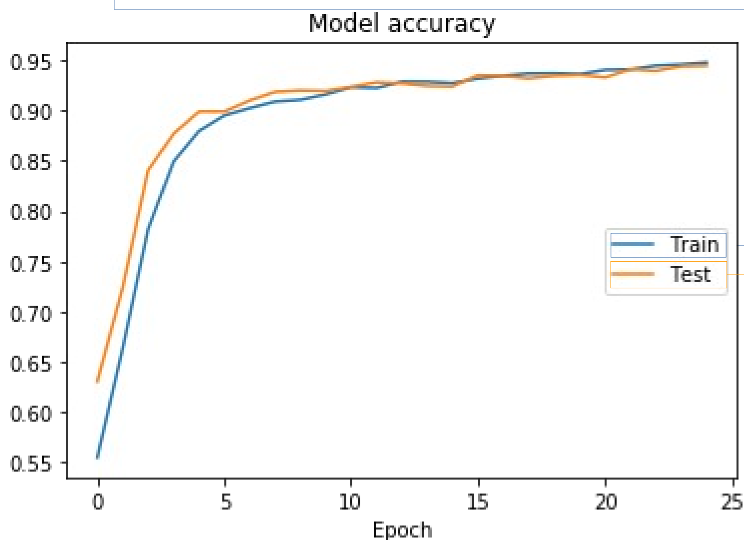
$$\text{accuracy} = \frac{\text{correctly classified data}}{\text{all data}}$$

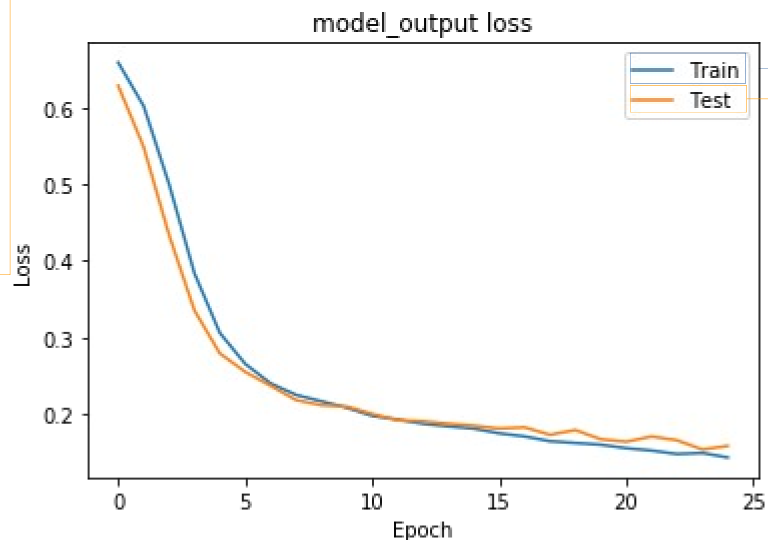
$$\hat{y} = \begin{cases} 1 & \text{if } \sin \sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

```
histObj.history.keys()

['acc', 'loss', 'val_acc', 'val_loss']
```

- With **too few** parameters, the NN may not be able to model the data.
- With **too many** parameters, the NN may start *memorising* the training data.





Q: when should one interrupt training?

Loss and accuracy

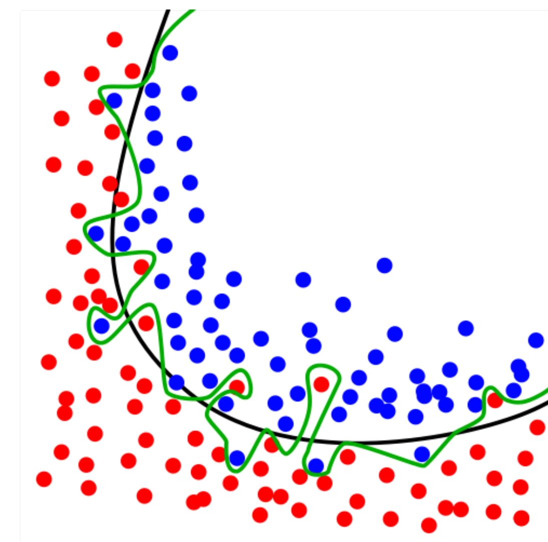
$$\text{accuracy} = \frac{\text{correctly classified data}}{\text{all data}}$$

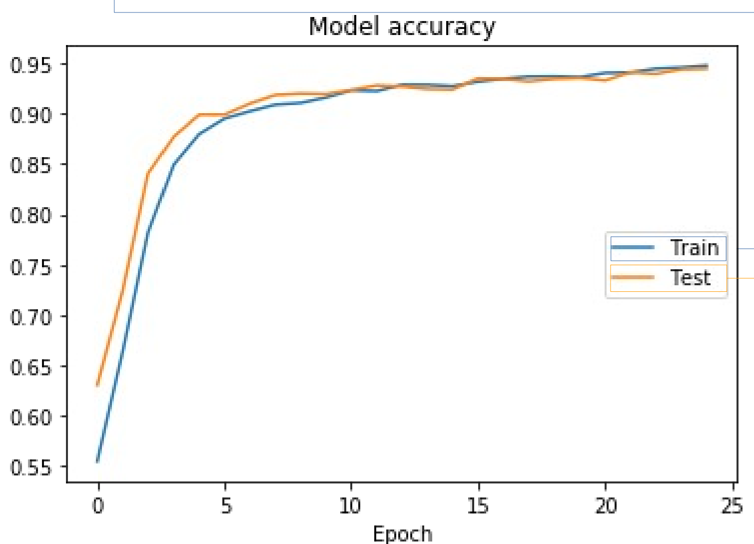
$$\hat{y} = \begin{cases} 1 & \text{if } \sin \sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

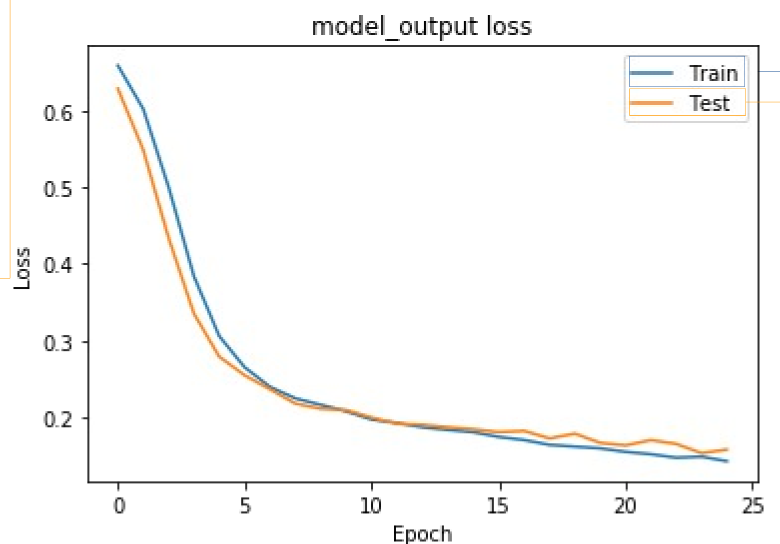
```
histObj.history.keys()
```

```
['acc', 'loss', 'val_acc', 'val_loss']
```

- With **too few** parameters, the NN may not be able to model the data.
- With **too many** parameters, the NN may start *memorising* the training data.







Q: when should one interrupt training?

Loss and accuracy

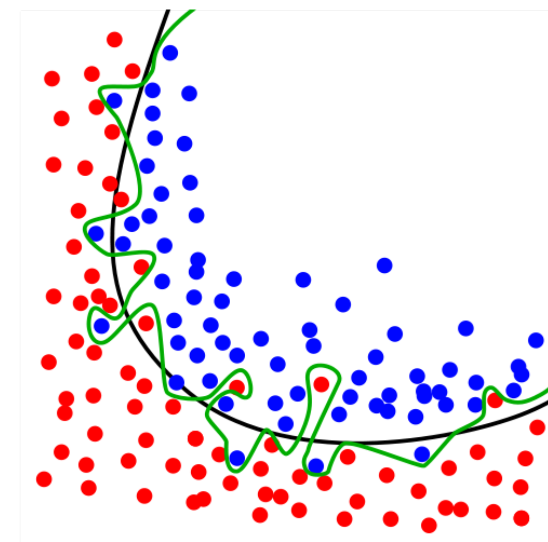
$$\text{accuracy} = \frac{\text{correctly classified data}}{\text{all data}}$$?

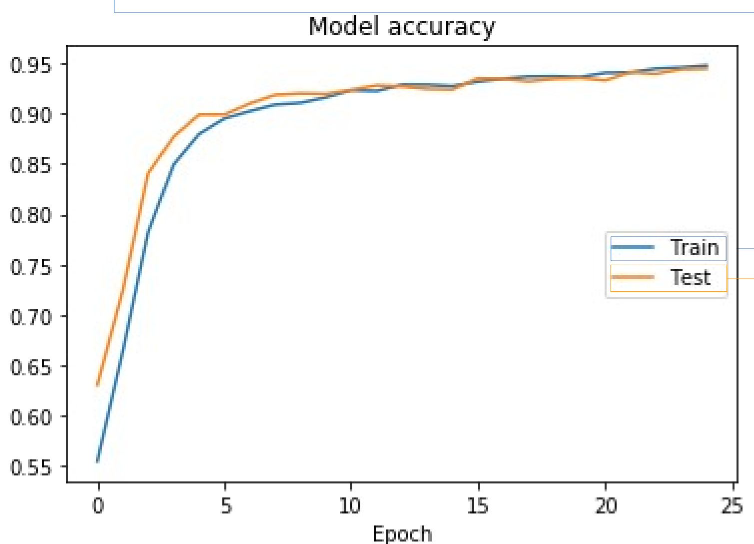
$$\hat{y} = \begin{cases} 1 & \text{if } \sin \sum_{k=1}^{10} x_k^2 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

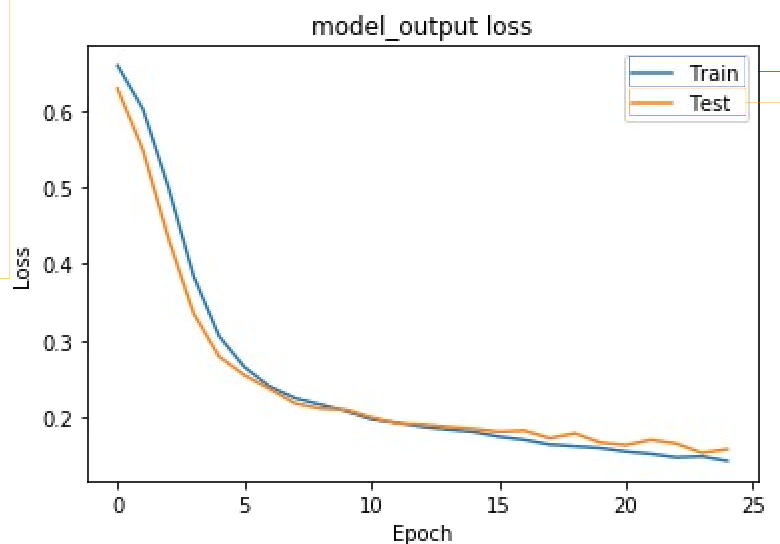
```
histObj.history.keys()
```

```
['acc', 'loss', 'val_acc', 'val_loss']
```

- With **too few** parameters, the NN may not be able to model the data.
- With **too many** parameters, the NN may start *memorising* the training data.

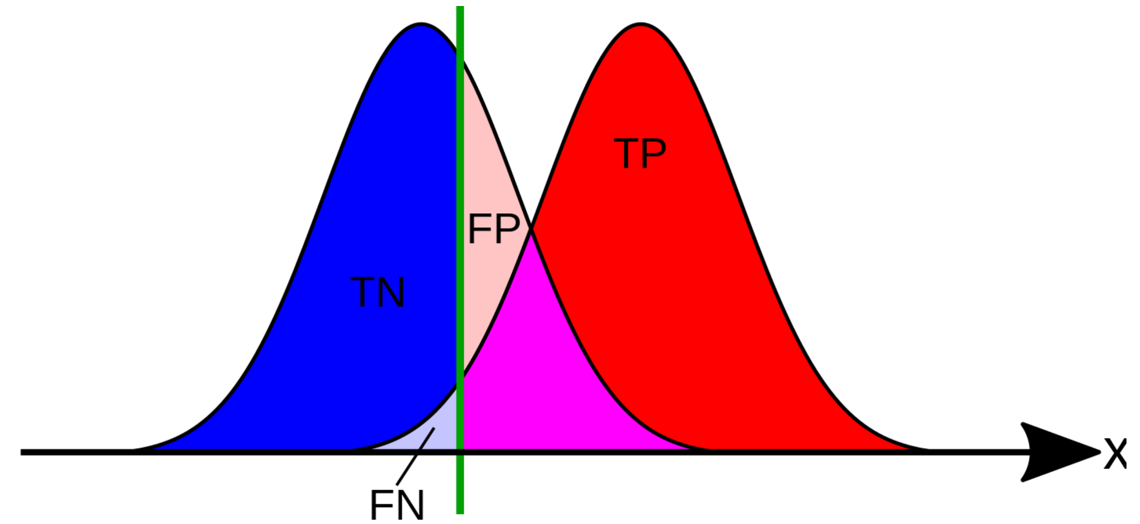






Q: when should one interrupt training?

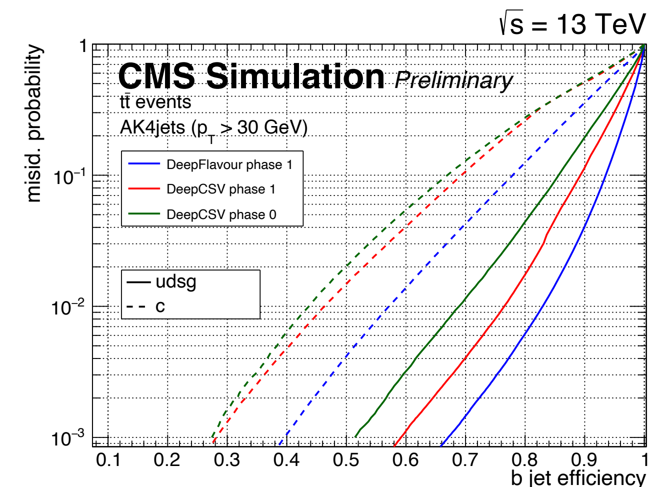
- The NN outputs a score for belonging to class „signal“ or not (class „background)
- To make a decision we need to choose a threshold of that score
- Key quantities:
  - True Positives (TP)
  - False Positives (FP)
  - True Negatives (TN)
  - False Negatives (FN)



- Purity/Precision = TP/(TP+FP): How pure is the sample predicted as signal?
- True Positive Rate/Sensitivity/Efficiency/Recall = TP/(TP+FN): What fraction of the signal is classified correctly?
- False Positive Rate = FP/(FP +TN)
- Accuracy = (TP+TN)/(TP+FP+TN+FN): What fraction of all data is classified correctly?

R(

- Overall performance of a classifier is usually visualized using the ROC curve:
  - Scan all possible thresholds and evaluate True Positive Rate and False Positive Rate
  - Plot one as a function of the other
- Area under the curve (AUC) often quoted as performance metric
- ROCs can aide in choosing a classifier and threshold
- Note: Sometimes ROCs of different classifiers cross each other -> they are better at different TPR regions



Multiclass classification

If there are more than 2 classes, we need to extend some of the concepts given above

Pedagogical examples

- Identify hand-written numbers
- Recognise clothes from catalogue

Multiclass classification

If there are more than 2 classes, we need to extend some of the concepts given above

Pedagogical examples
- Identify hand-written numbers
- Recognise clothes from catalogue

NIST

Categorical cross-entropy* (CCE)

Just generalisation to multi-nominal case

*we skip the mathematical details

Softmax activation function

$$f_k(\vec{z}) = \frac{\exp z_k}{\sum_{k=1}^{K} \exp z_k}$$

Example: Fashion MNIST

All pictures are **prepared** in a comparable and simple format

28 pixels

28 pixels

60k pictures

Grayscale 0-255

```
import tensorflow as tf
dataset = tf.keras.datasets.fashion_mnist
```

```
(x_train, y_train),(x_test, y_test) = dataset.load_data()
print(x_train.shape)
print(x_train.dtype)
```

→ (60000, 28, 28)
→ uint8

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Example: Fashion MNIST

```
import tensorflow as tf
dataset = tf.keras.datasets.fashion_mnist
```

All pictures are **prepared** in a comparable and simple format

T-shirt/top

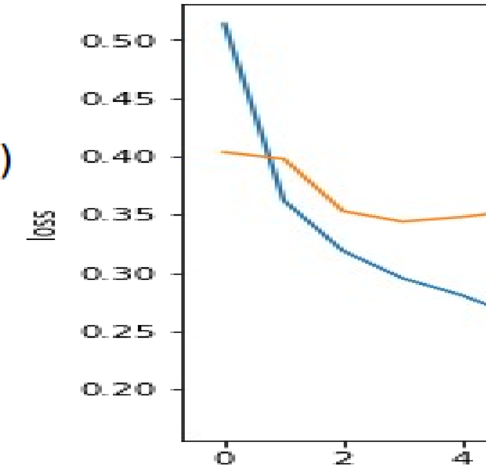
60k pictures

Grayscale 0-255

```
(x_train, y_train),(x_test, y_test) = dataset.load_data()
print(x_train.shape)
print(x_train.dtype)
```

→ (60000, 28, 28)
→ uint8

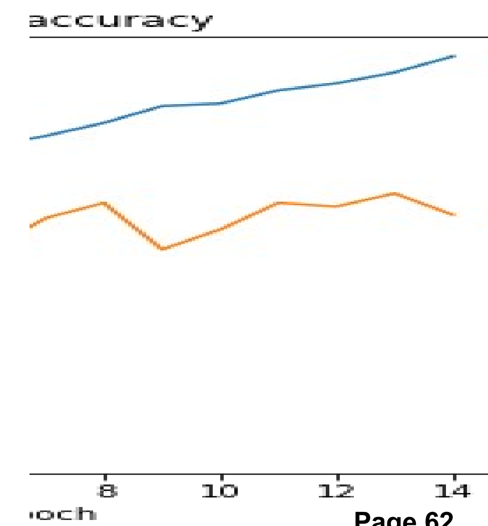
```
x_train, x_test = x_train / 255.0, x_test / 255.0
```



```
model1 = tf.keras.models.Sequential([
  layers.Flatten(input_shape=(28, 28)),
  layers.Dense(512, activation='relu'),
  layers.Dense(512, activation='relu'),
  layers.Dense(10, activation='softmax')
])
model1.summary()

model1.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['acc'])
```

Improve it!



Example: Fashion MNIST

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Reminder

- With **too few** parameters, the NN may not be able to model the data.
- With **too many** parameters, the NN may start *memorising* the training data.

```
from tensorflow.keras import models, layers, losses, optimizers, regularizers
```

Weight regularisation

Constrain weights in successive layers with penalty term in the loss

```
kernel_regularizer=
          =regularizers.l2(0.0001)
```

Batch normalisation

Normalise the data between the layers and for each batch

```
layers.BatchNormalization()
```

Dropout

Remove randomly a certain number of nodes from iteration to iteration

```
layers.Dropout(rate=0.3)
```

Play with them!

Summary

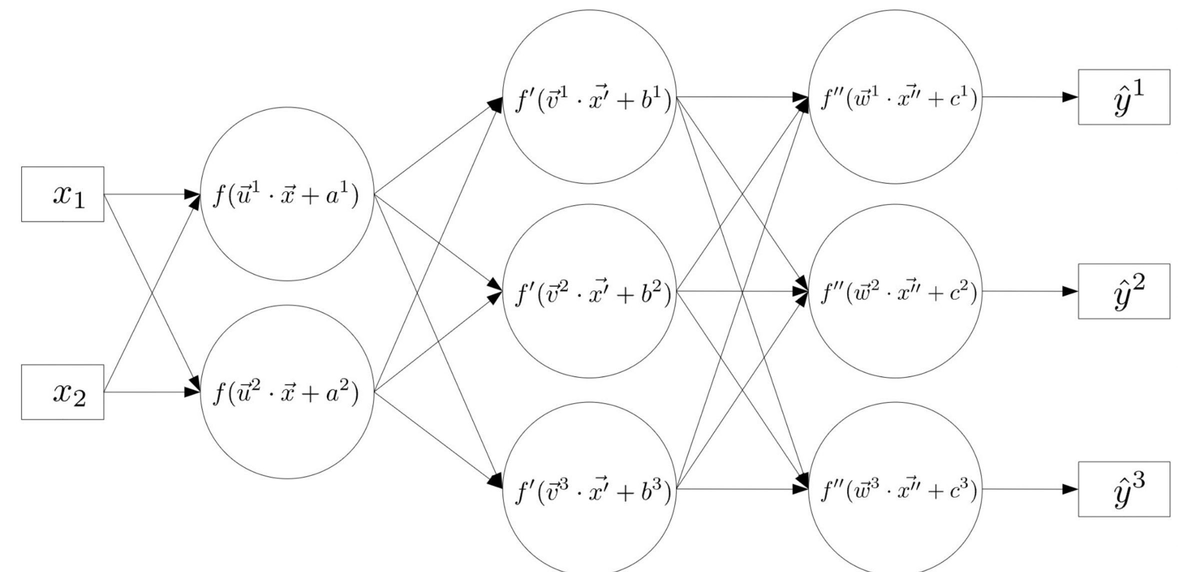
Summary

Covered
- Supervised learning
- Dense neural network
  - Activation function
    - ReLU
    - Softmax
  - Node/neuron
  - Layer
- Optimiser
  - SGD
  - Adam
- Monitoring
  - Learning curve
  - Accuracy
- Control sample
- Regression
- Classification

Not covered
- Convolutional NN (CNN)
- Generative NN
- Auto-encoders
- Initial value for the weights
- Backpropagation
- Callbacks
- Unsupervised learning
- Alternative libraries
- Data preprocessing
- Input normalisation
- Regularisation
- Overfitting mitigation
- ...

```
from tensorflow.keras import models, layers, losses, optimizers
from time import time
model = models.Sequential(
    [
        layers.Dense(20, activation='relu', input_dim=10),
        layers.Dense(30, activation='relu'),
        layers.Dense(1)
    ]
)
optimizer = optimizers.SGD(lr=0.01)

model.compile(optimizer=optimizer,loss='mse')
```

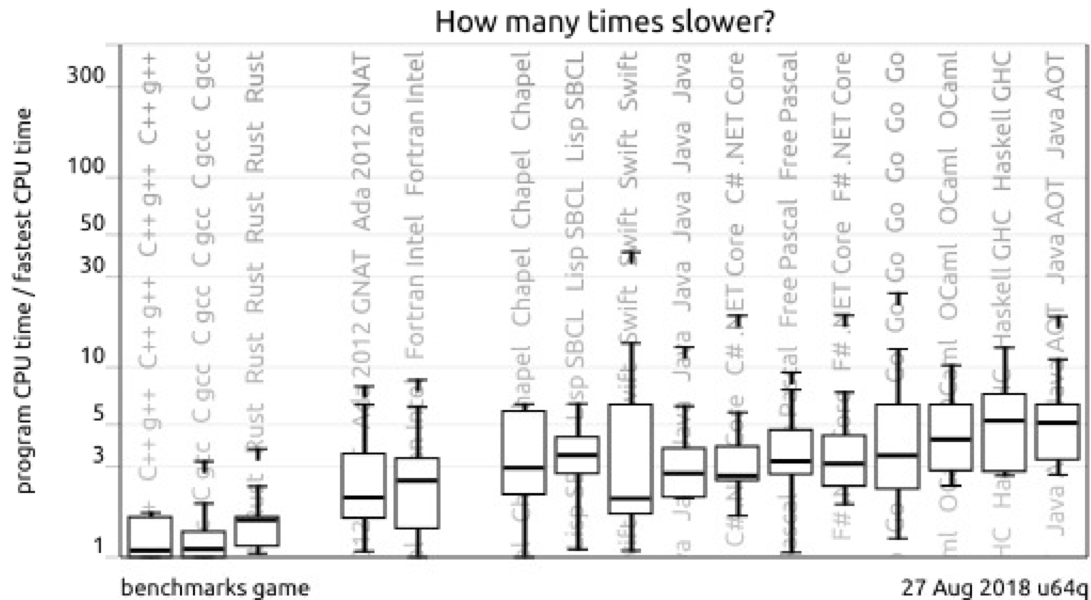


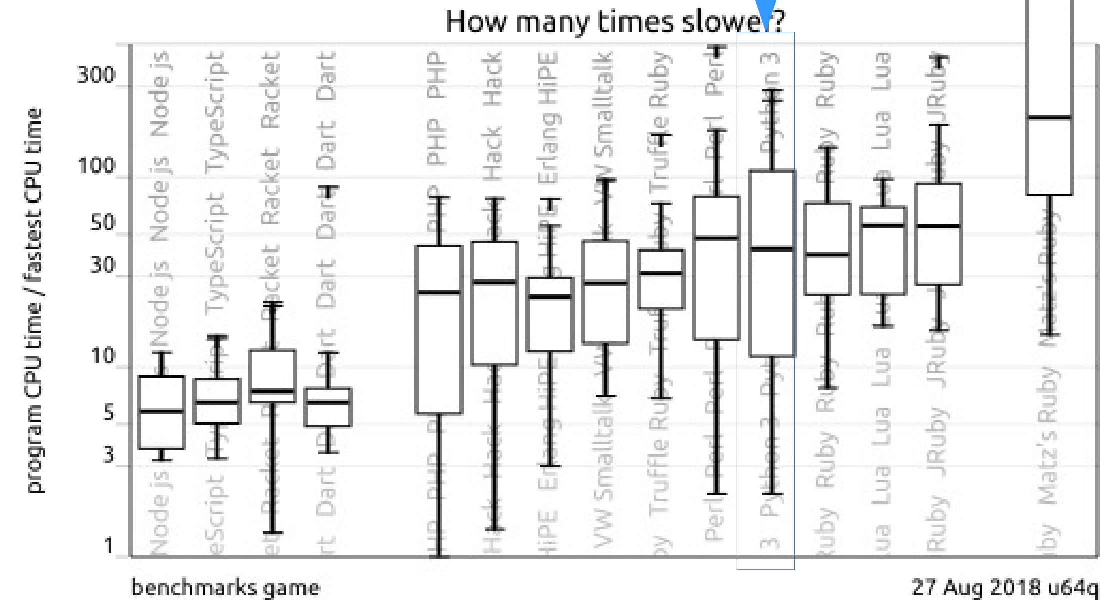
Backup

Why Python?

But don’t use Python blindly!
In DL, it is “just”
a powerful interface.

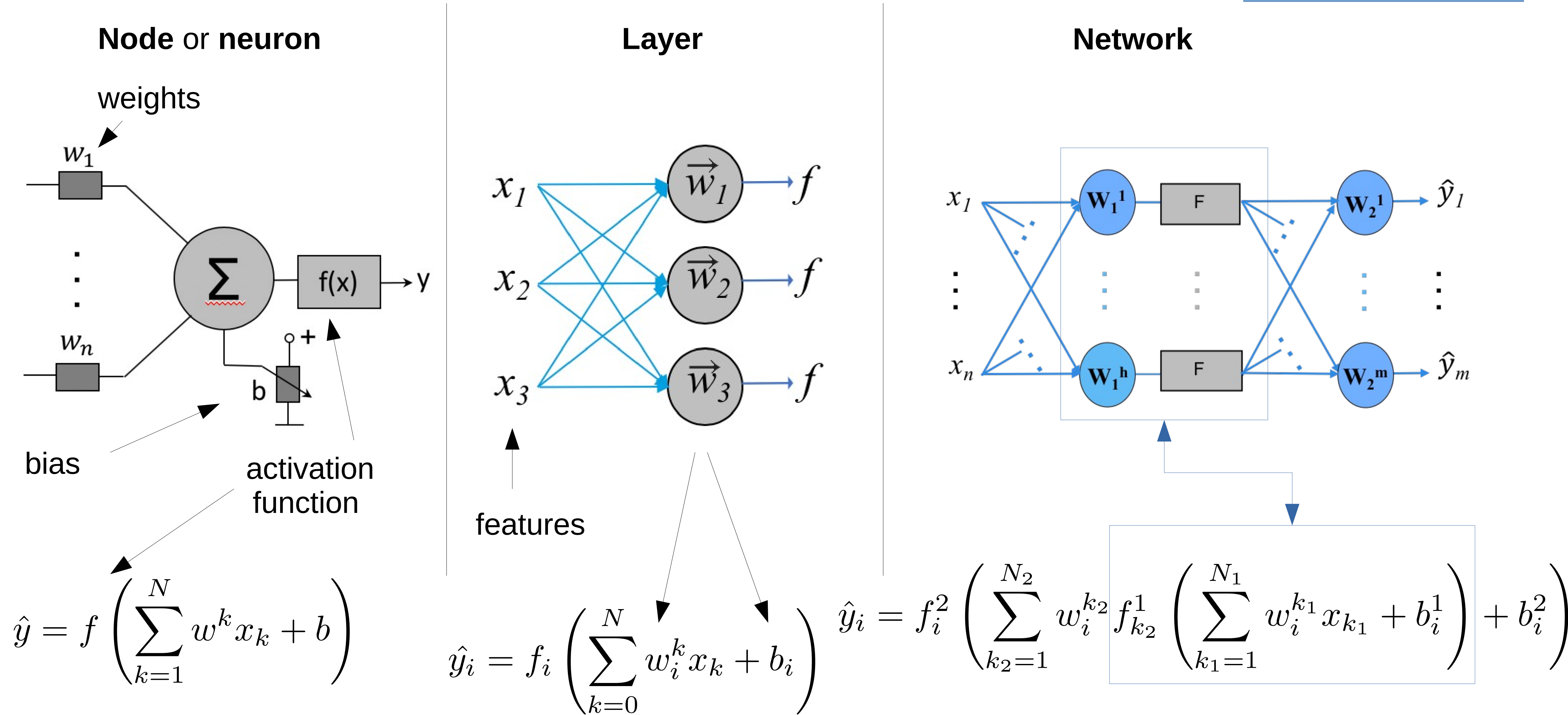
- Not always the fastest language itself
- But Simple programming syntax (close to human language)
- No need to worry about pointers, references, etc.
- Powerful libraries and interfaces





Building a Neural Network

Deep Learning means many hidden layers

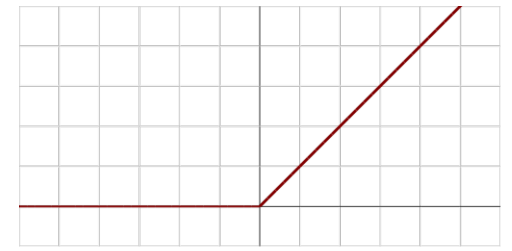


$$\hat{y} = f\left(\sum_{k=1}^{N} w^k x_k + b\right)$$

$$\hat{y}_i = f_i\left(\sum_{k=0}^{N} w_i^k x_k + b_i\right)$$

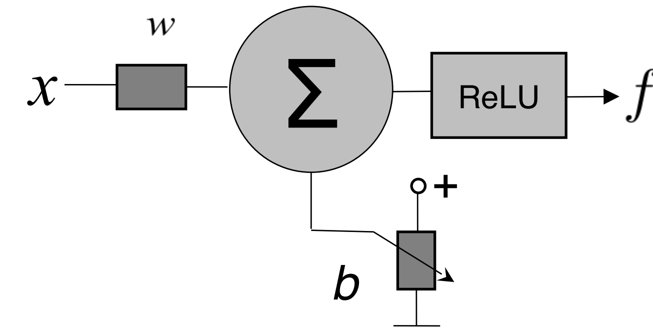
$$\hat{y}_i = f_i^2\left(\sum_{k_2=1}^{N_2} w_i^{k_2} f_{k_2}^1\left(\sum_{k_1=1}^{N_1} w_i^{k_1} x_{k_1} + b_i^1\right) + b_i^2\right)$$

Bias parameters: ReLU



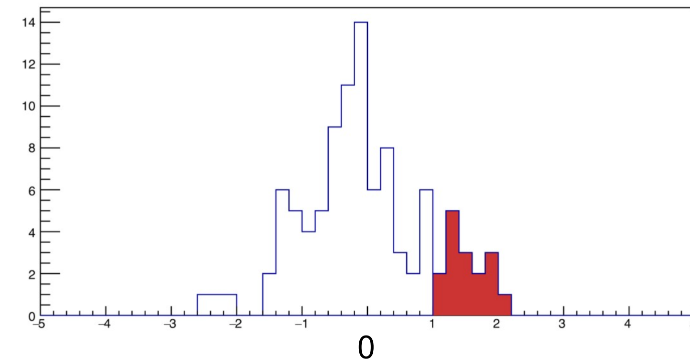
- The condition defines a cut on the input variable x

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

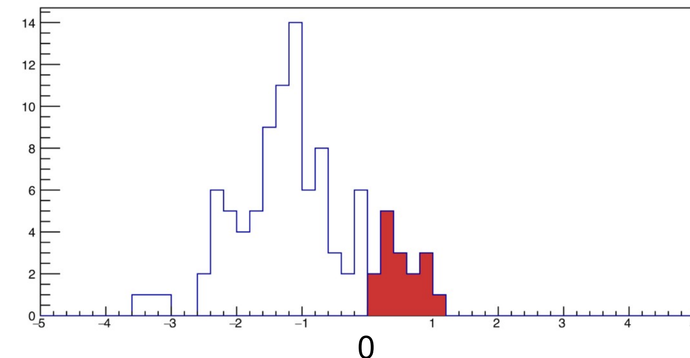


- The cut on *x* is defined by the weight *w* and the bias *b*
- Instead of cutting on $x_{in}>x_{cut}$ we scale x_{in} by *w* and shift the input distribution by *b*:

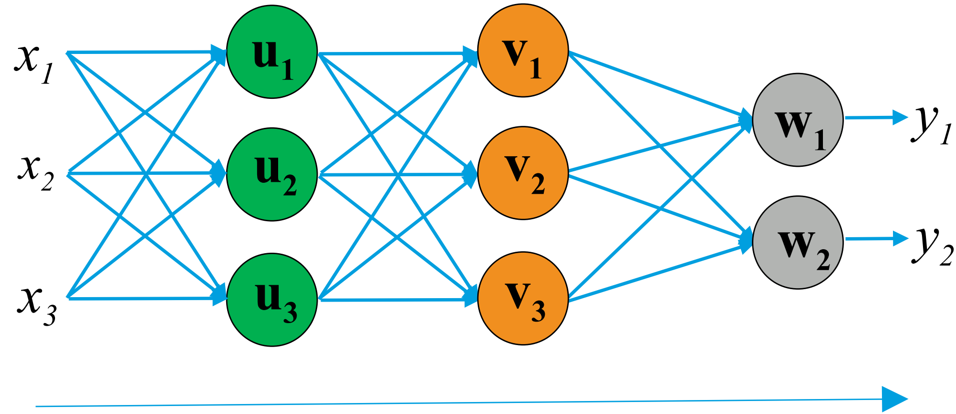
$$f(wx_{\text{in}} + b)$$



- *"We move the distribution to the cut"*
- In a ReLU layer the **bias defines the scale**

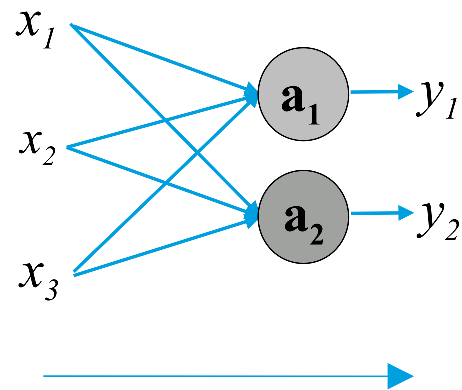


Activations and linearity



$$\underbrace{\vec{\mathbf{y}}}_{2\times1} = \overbrace{\mathbf{W}}^{2\times3}\underbrace{\mathbf{V}}_{3\times3}\overbrace{\mathbf{U}}^{3\times3}\underbrace{\vec{\mathbf{x}}}_{3\times1}$$

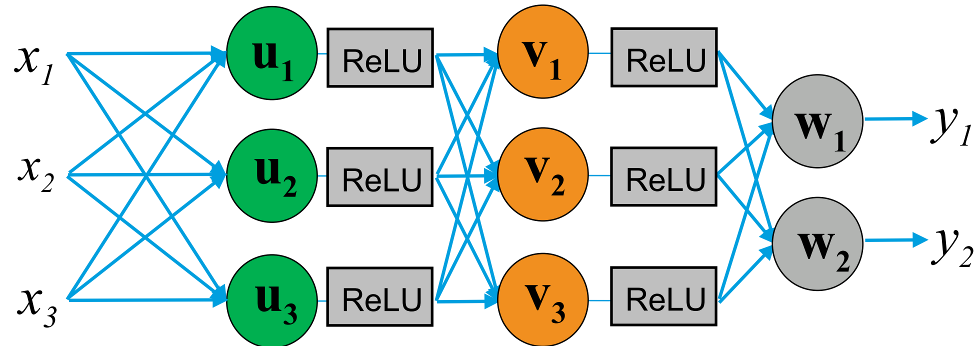
If you remove
the activation function,
layers just contract...



$$\underbrace{\vec{\mathbf{y}}}_{2\times1} = \overbrace{\mathbf{A}}^{2\times3}\underbrace{\vec{\mathbf{x}}}_{3\times1}$$

The activation function
is a <u>fundamental</u>
component
of a Neural Network

Activations and linearity



$$\underbrace{\vec{\mathbf{y}}}_{2\times 1} = \overbrace{\mathbf{W}}^{2\times 3} F(\underbrace{\mathbf{V}}_{3\times 3} \underbrace{F(\mathbf{U}\vec{\mathbf{x}})}_{3\times 1})$$

with $F(\vec{\mathbf{x}}) := \left[F(x_i)\right]$

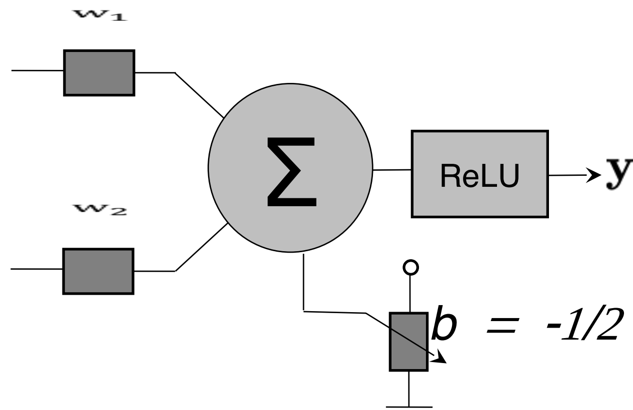
i.e. F applied to each component of $\vec{\mathbf{x}}$

and F=ReLU

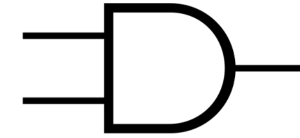
If you remove the activation function, layers just contract...

The activation function is a <u>fundamental</u> component of a Neural Network

ReLU as logic gate

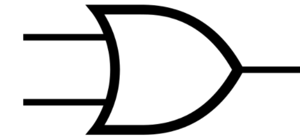


$w_1 = w_2 = 1/2$



x	y	Σ	y
0	0	0	0
1	0	1/2	0
0	1	1/2	0
1	1	1	1

$w_1 = w_2 = 1$



x	y	Σ	y
0	0	0	0
1	0	1	1
0	1	1	1
1	1	2	1

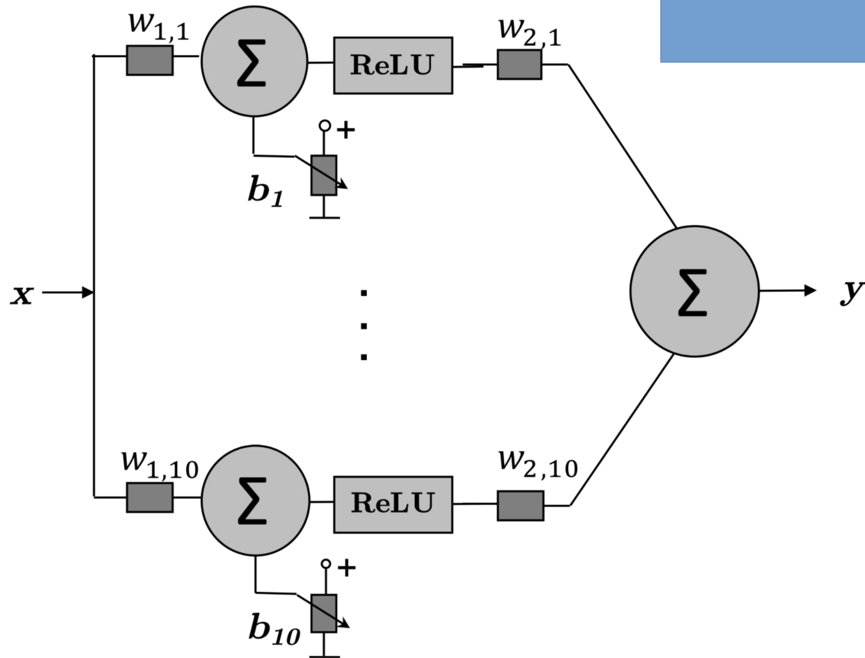
All kinds of can data manipulations can be realize by ReLU networks

negative weight ⟶ logic not

Universal Approximation

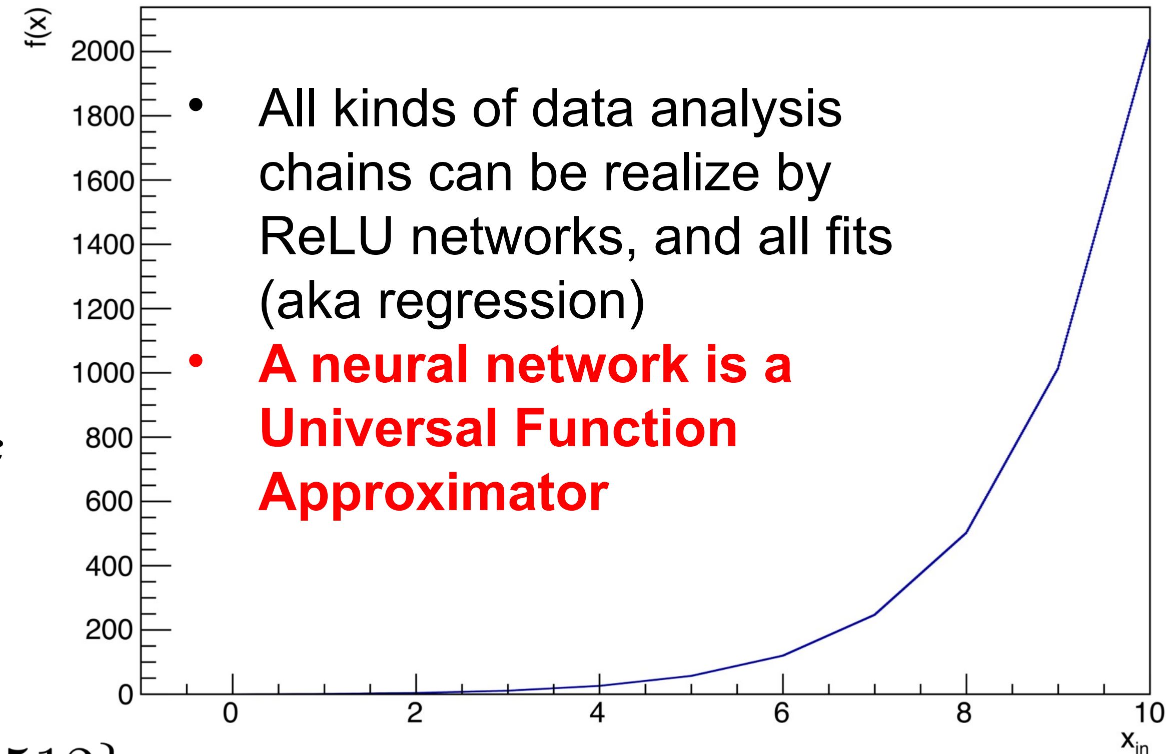
Any "reaonable" funtion f can be approximated with a layers FA

$$\int_{\mathbb{R}^n} |f(x) - F_A(x)|\, \mathrm{d}x < \epsilon$$



Example

$$f(x_{\text{in}}) \approx 2^x$$



- All kinds of data analysis chains can be realize by ReLU networks, and all fits (aka regression)
- **A neural network is a Universal Function Approximator**

$$\{w_{1,1}, \ldots, w_{1,10}\} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$
$$\{w_{2,1}, \ldots, w_{2,10}\} = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$$
$$\{b_1, \ldots, b_{10}\} = \{0, -1, -2, -3, -4, -5, -6, -7, -8, -9\}$$

Loss minimisation

$$loss(\mathbf{y}, \hat{\mathbf{y}}) = (\hat{\mathbf{y}} - \mathbf{y})^2$$ with

$$\underset{m\times 1}{\hat{\mathbf{y}}} = \underset{m\times h}{\mathbf{W}_2} F(\underset{h\times n}{\mathbf{W}_1} \underset{n\times 1}{\mathbf{x}})$$

Back-propagation

- To minimize the loss we take the derivative wrt. to ***w***

$$\frac{\partial loss}{\partial w_i} = 0 \Rightarrow \frac{\partial(\hat{\mathbf{y}} - \mathbf{y})^2}{\partial \mathbf{W}_i} = 2(\hat{\mathbf{y}} - \mathbf{y})\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}_i} = 0$$

E.g. for W_2 -
Similar for W_2

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}_1} = \mathbf{W}_2 \frac{\partial F(z)}{\partial z}\mathbf{x}$$

- Equation to be solved for $W_{1,2}$
- **Chain rule – from the end to the beginning**

Loss minimisation

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- The previous chain rule calculation is known as **Backpropagation**
- The deviation between true and estimated $\boldsymbol{y}$, i.e. the loss, is back-propagated to a linear change of the weights.
  - Invented by different people in the 1960/70s
- **NB: The chain rule creates a chain of factors that can be evaluated numerically**

$$\Delta loss(\hat{\mathbf{y}}) \approx \frac{\partial loss}{\partial \mathbf{W}} \Delta \mathbf{W}$$

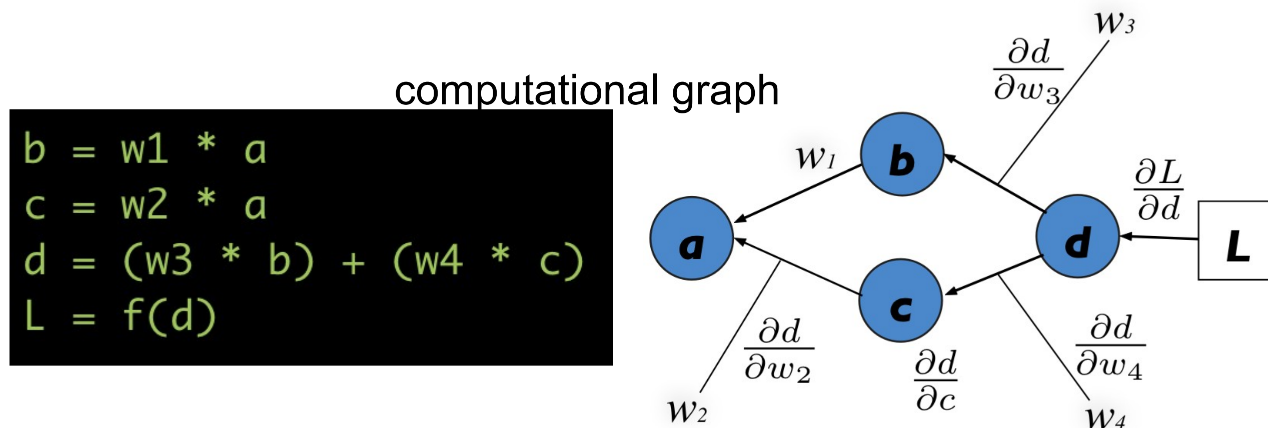
We calculate a **gradient** with respect to the weights/biases

$$\frac{\partial loss(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{W}_i} = \frac{\partial loss(\hat{\mathbf{y}}, \mathbf{y})}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial F} \frac{\partial F(z)}{\partial z} \frac{\partial z}{\partial \mathbf{W}_i}$$

- These are vector and matrix multiplication. If you need a Matrix calculus primer or work it out with tensor indices but in reality …

Automatic differentiation

- Deep Learning libraries are able to calculate the derivative of a piece of code
  - This is **not** a numerical approximation (no small epsilon)
  - This **not** symbolic differentiation (not as in e.g. Mathematica)
  - Think about it as **a derivative of your python code**

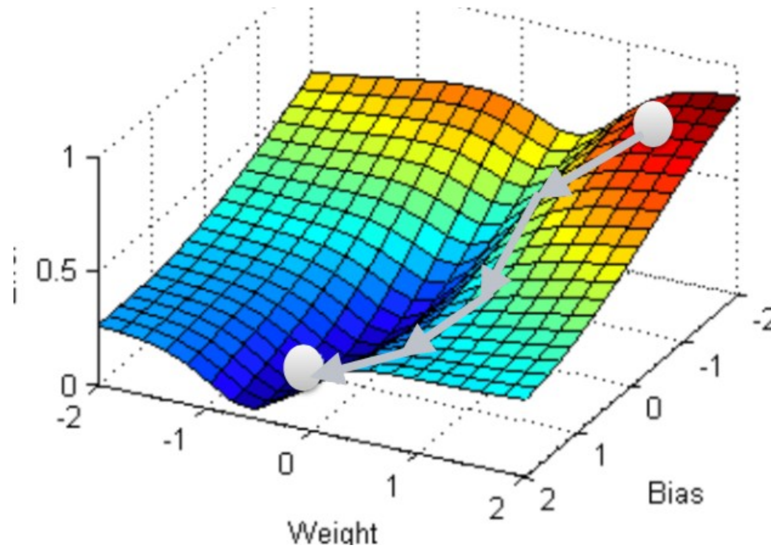


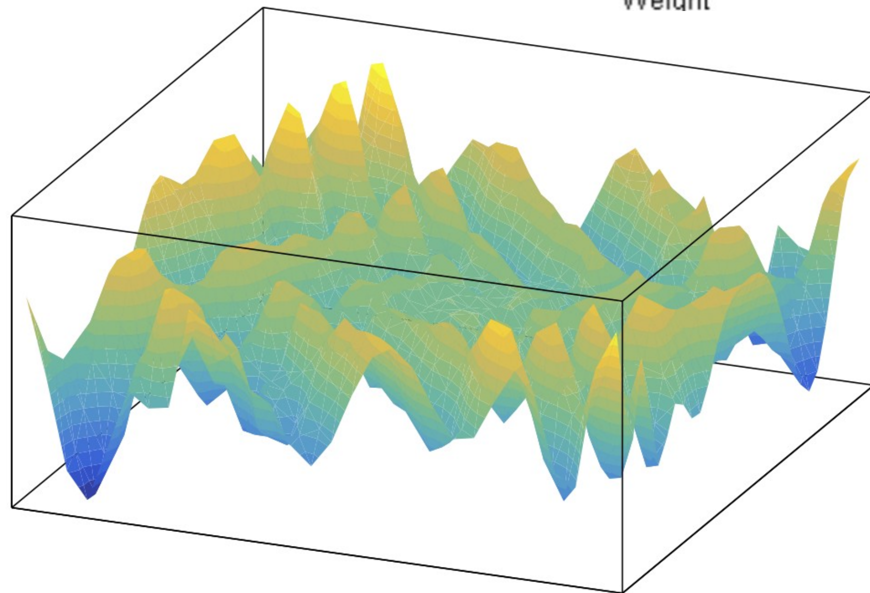
- It records your calculations (Forward step)
- It then calculates by applying the chain rule (Backward step) the gradients at the same numerical value
- The DL library keeps track of **hundreds of thousands of weights and more**
- There are different approaches to automatic differentiation

The different libraries e.g. Tensorflow, Pytorch etc. follow slightly different concepts

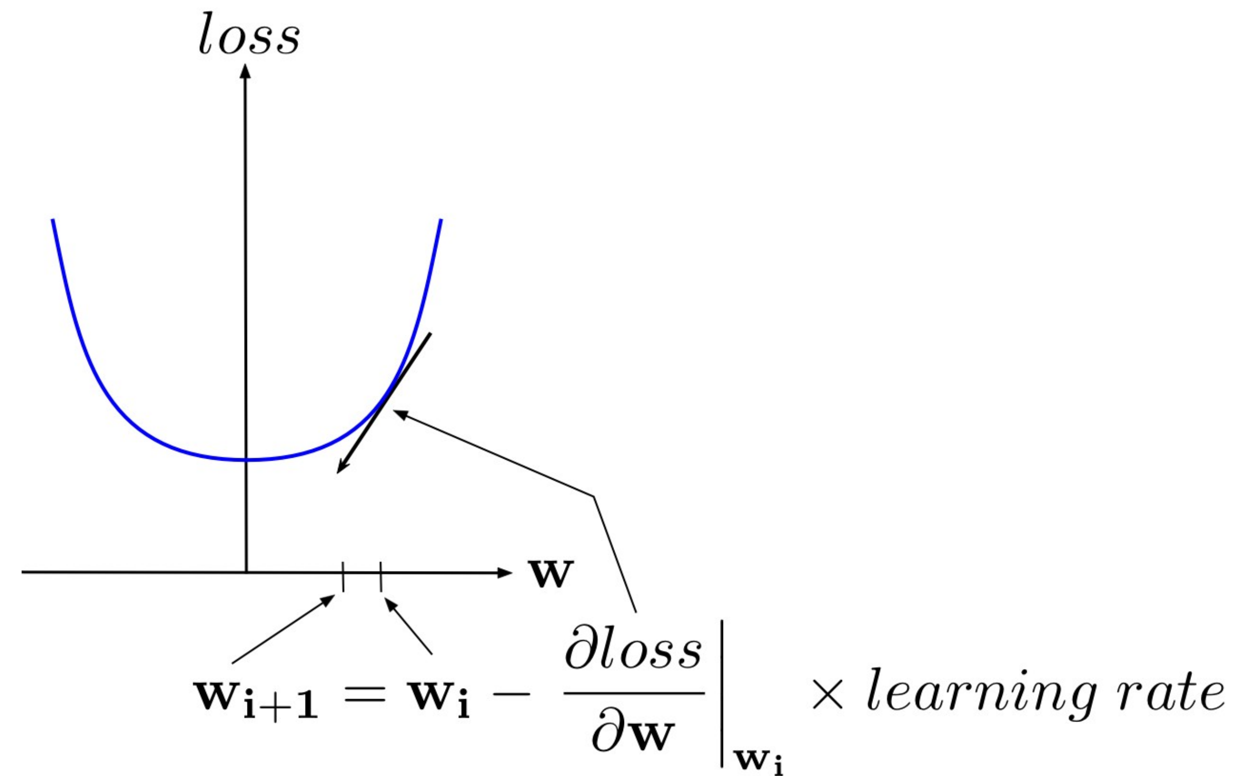
Important to understand if you want to define your own special layers and loss function

Gradient descent and learning rate





$$\Delta loss(\hat{\mathbf{y}}) \approx \frac{\partial loss}{\partial \mathbf{W}} \Delta \mathbf{W}$$



$$\mathbf{w_{i+1}} = \mathbf{w_i} - \left.\frac{\partial loss}{\partial \mathbf{w}}\right|_{\mathbf{w_i}} \times learning\ rate$$

Adam optimizer

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Overview: https://arxiv.org/abs/1609.04747
Seminal paper by LeCun (1998)

GPUs

Graphical
Processor
Unit

GPUs for video games can be used for DL

Applications

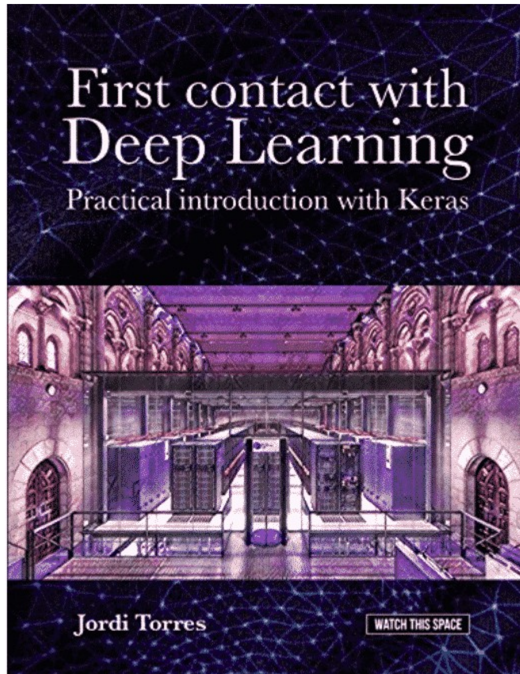
- 3D graphics
- Cryptocurrencies
- Vectorial calculation
- *Deep learning*

```
from tensorflow.python.client import device_lib
import tensorflow as tf
print(tf.__version__)
print(device_lib.list_local_devices())
```

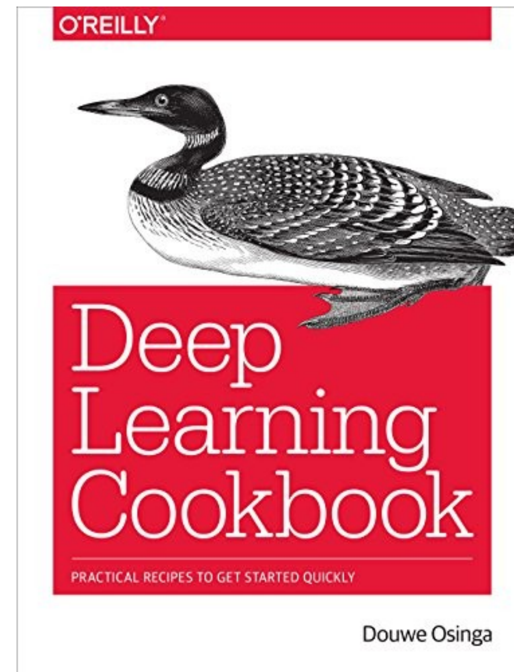
Introduction to Deep Learning with Keras

References

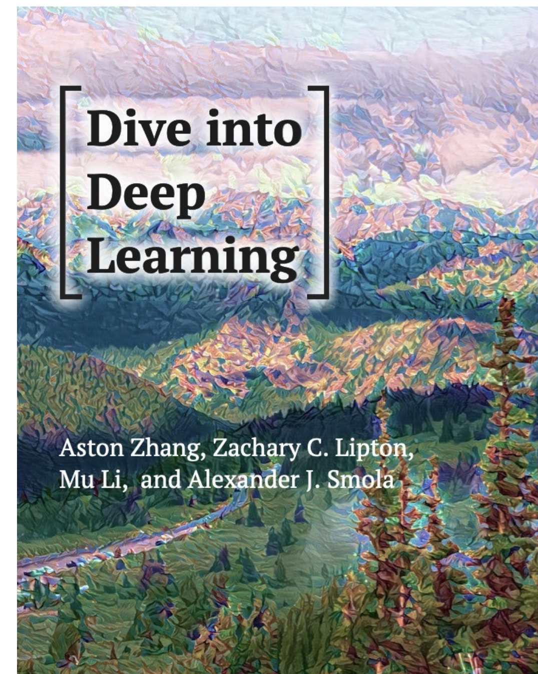
Good for a quick intro



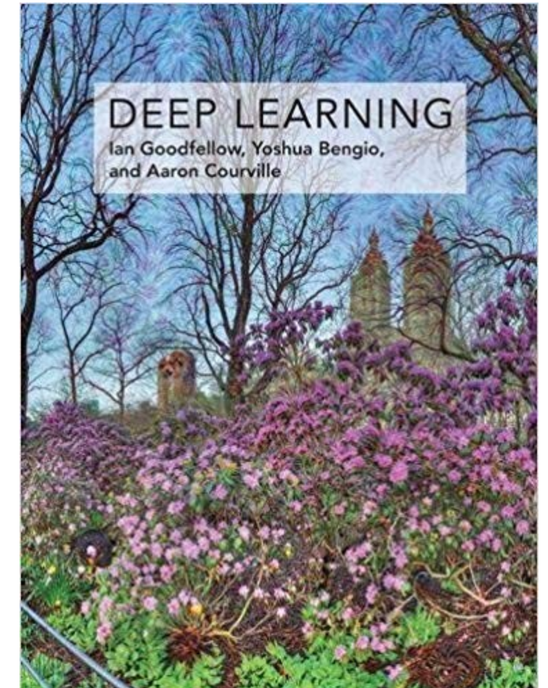
Similar style, but many more details



Amazon reference



THE standard reference



… and many more, also for free & online!