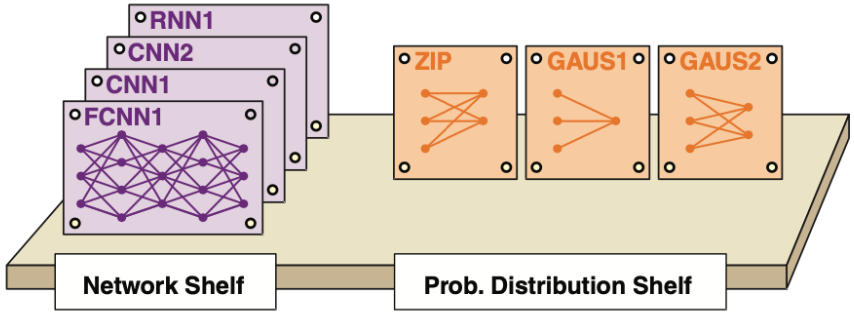
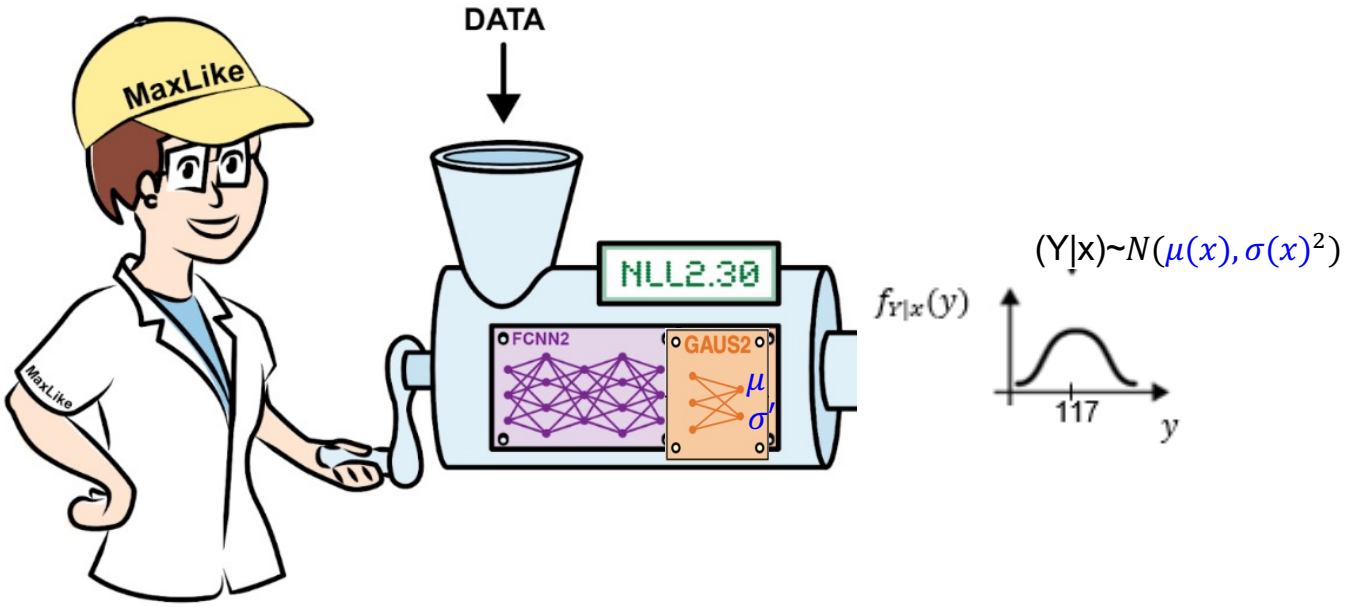

RNN1
CNN2
CNN1
FCNN1
ZIP
GAUS1
GAUS2
Network Shelf
Prob. Distribution Shelf


DATA
MaxLike
NLL2.30
FCNN2
GAUS2
μ
σ
$f_{Y|x}(y)$
(Y|x)~$N(\mu(x), \sigma(x)^2)$
117
y

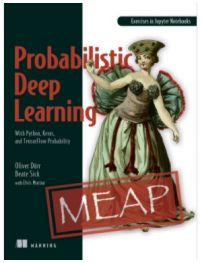
WBL Deep Learning:: Lecture 1

Beate Sick, Oliver Dürr

Lecture 1: Introduction to probabilistic deep learning

Literature TODO tensorchiefs - lit wie dort

- Additional Course website
  - https://tensorchiefs.github.io/dl_rcourse_2022/



- Probabilistic Deep Learning (Manning in production)
  - Our probabilistic take
  - https://www.manning.com/books/probabilistic-deep-learning?a_aid=probabilistic_deep_learning&a_bid=78e55885

- Deep Learning Book (DL-Book) http://www.deeplearningbook.org/. This is a quite comprehensive book which goes far beyond the scope of this course.

- Courses
  - Convolutional Neural Networks for Visual Recognition http://cs231n.stanford.edu

  - Martin Görner (very practical)
    - https://cloud.google.com/blog/products/gcp/learn-tensorflow-and-deep-learning-without-a-phd

Dates TODO

Date	Lectures
12.09.2022	L1 (10:15-12:00), L2 (14:15-16:00) and Exercises (16:15-18:00)
19.09.2022	L3 (14:15-16:00) and Exercises (16:15-18:00)
26.09.2022	L4 (14:15-16:00) and Exercises (16:15-18:00)
03.10.2022	This day is reserved to work on your projects
10.10.2022	L5 and presentatation of the projects

There will be also small exercises during the lectures

Topics TODO

- Lecture 1
  - Introduction to probabilistic DL
  - Introduction to Keras with pytorch backend
- Lecture 2
  - Convolutional Neural Networks (CNN) for image data
  - Transformer Architecture
  - Foundation Models = Pretrained models on many data
    - ImageNet models, LLM, viT, tabPFN
  - Transfer learning
- Lecture 3
  - Probabilistic DL
  - Extending the GLM with DL for scalar features and image data
- Lecture 4,5 (Not 100% sure yet)
  - Extending deep GLMs by deep transformation models
  - Deep interpretable ordinal regression models

Deep Learning vs. Machine Learning

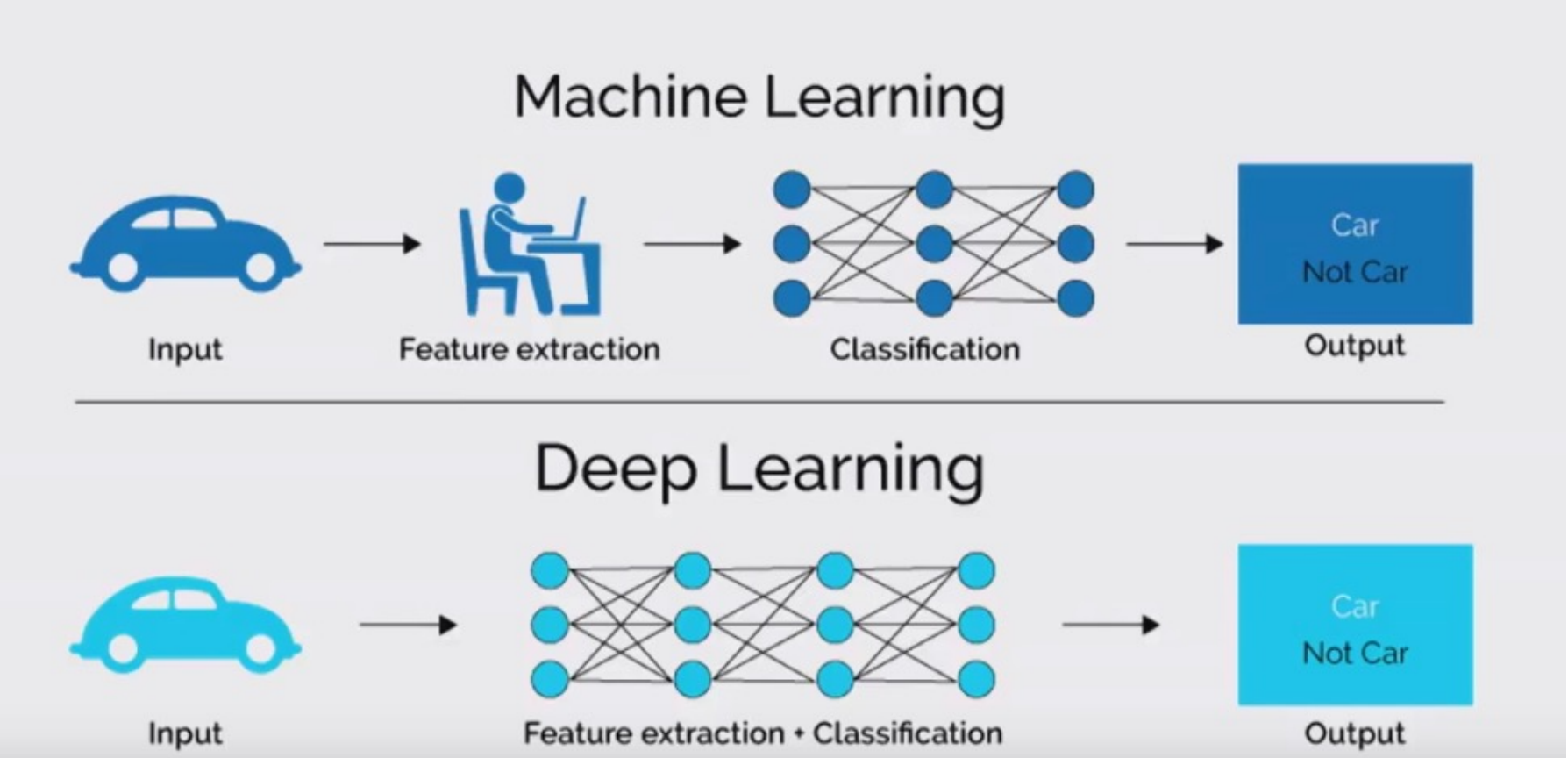
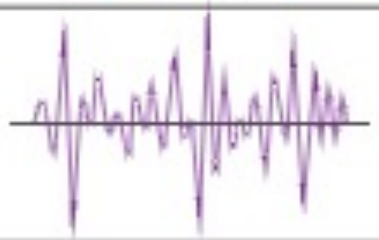


image credit: Siraj Raval

Use cases of deep learning

Input x to DL model	Output y of DL model	Application
Images 	Label "Tiger"	Image classification
Audio 	Sequence / Text "see you tomorrow"	Voice recognition
Sequence of tokens representing e.g. "Hallo, wie gehts?"	Next token	Translation text generation, QA
Sequence of tokens representing e.g. This movie was rather good	Label (Sentiment) positive	Sentiment analysis

Deep learning models can „see“, „hear“, „write“ .

Last breakthrough 2022: Large Language Models (LLMs) like ChatGPT

Course topic is part of our research focus

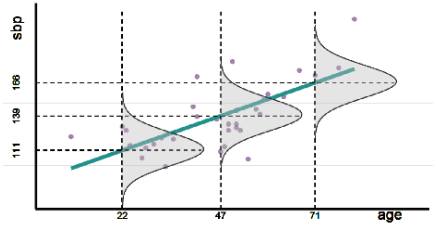
Statistics

+ transparent & interpretable

+ valid uncertainty measures

- needs structured data

- not robust vs distribution changes



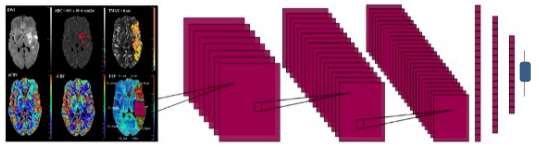
Deep Learning

+ can handle structured & unstructured data

+ high prediction performance

- black box,

- not robust vs distribution changes



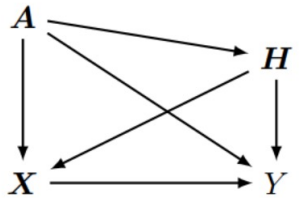
Interpretable, robust (causally inspired) & probabilistic DL

Causality

+ robust vs distribution changes
+ allows intervention planing

+ transparent & interpretable

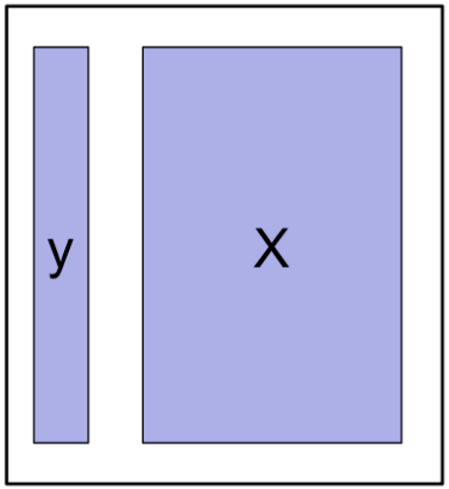
- needs stronger assumptions



Focus in this DL course:

Probabilistic Viewpoint

Tasks in supervised DL



Supervised Learning

- Typical supervised DL task: predict y given x
  - **Classification**
    - Point prediction: Predict a class label
    - Probabilistic prediction: predict a discrete probability distribution over the class labels
    - First, we focus on probabilistic binary classification where $Y \in \{0,1\}$
  - Regression
    - Point prediction: Predict a number
    - Probabilistic prediction: predict a continuous distributions

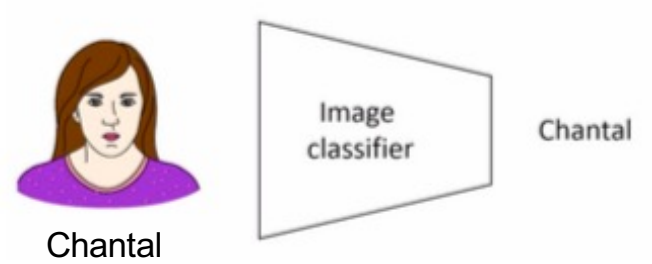
Probabilistic vs deterministic models

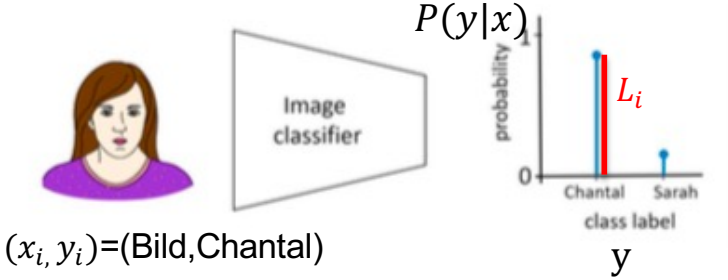
Likelihood L_i of the observed outcome y_i under the predicted conditional outcome distribution $P(Y|x_i)$

Deterministic

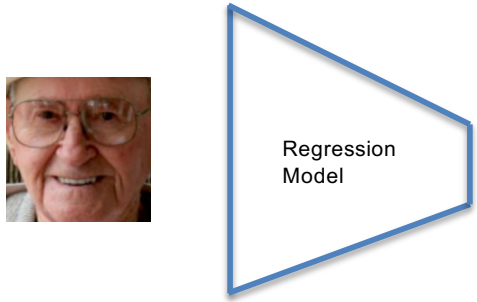
Probabilistic

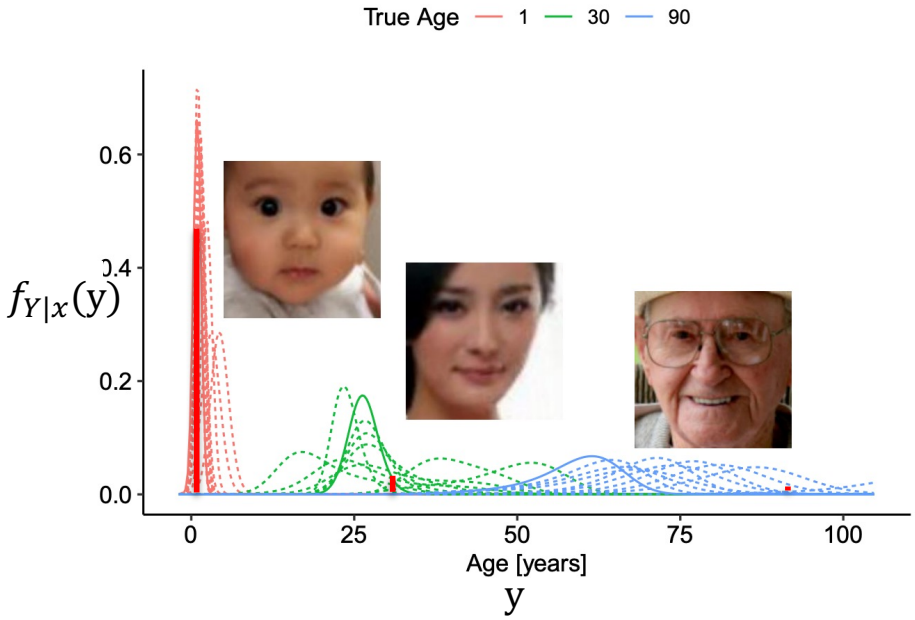
"Classification"





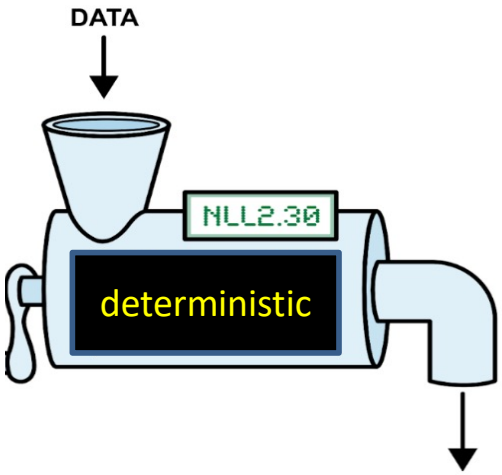
"Regression"



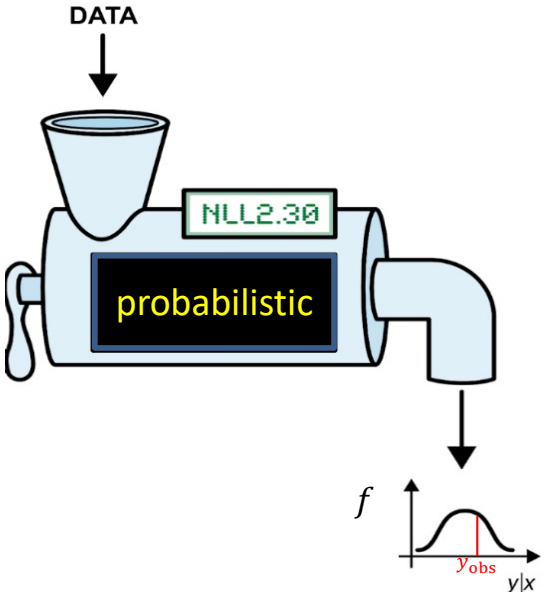


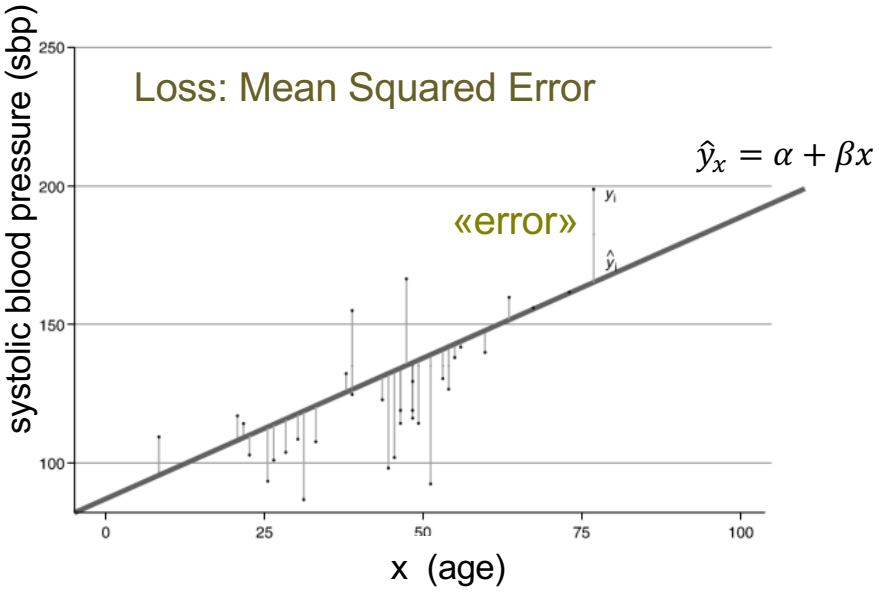
Conditional probability distribution $P(y|x)$ aka $f_{Y|x}(y)$

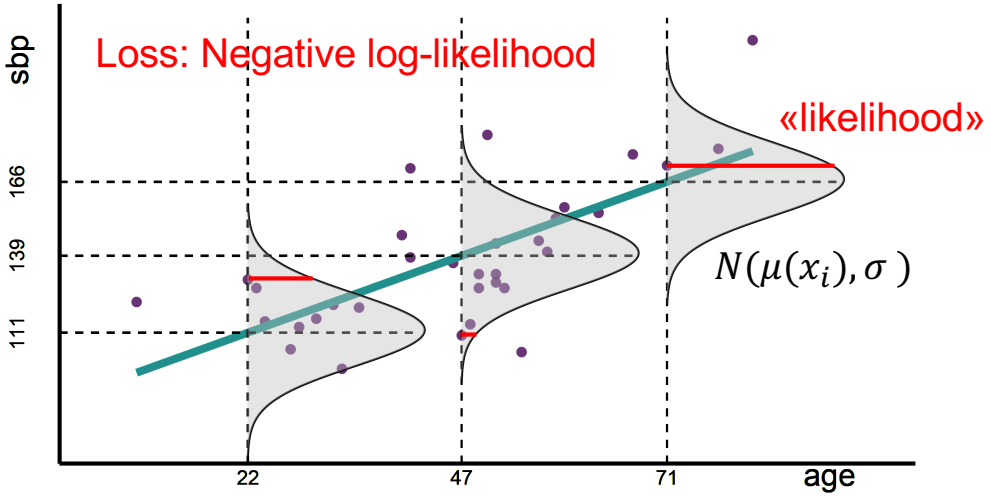
Non-probabilistic versus probabilistic regression DL models



$\hat{y} = 117$



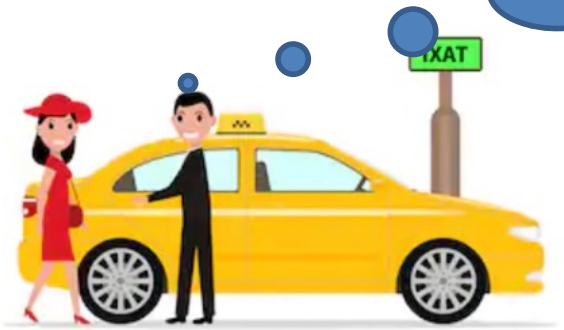




Probabilistic models in everyday tasks

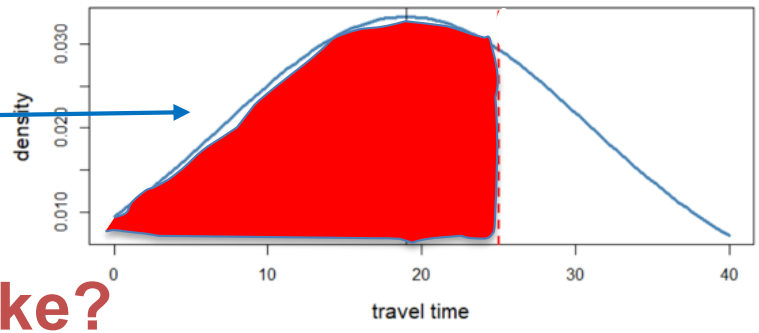
You'll get 500$ tip if I arrive at MOMA within 25 minutes!

Let's use my probabilistic travel time gadget!

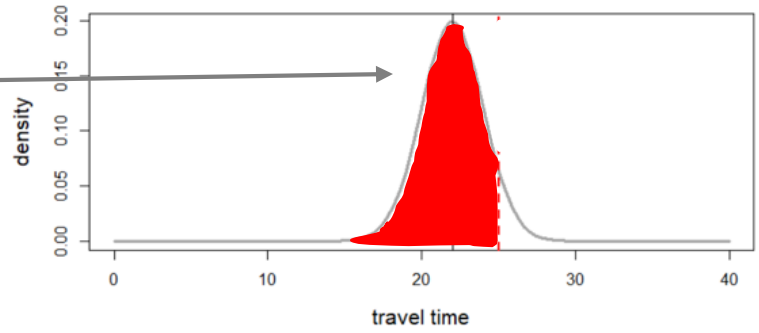


Which way would you take?

Chance to get tip: 69%



Chance to get tip: 93%

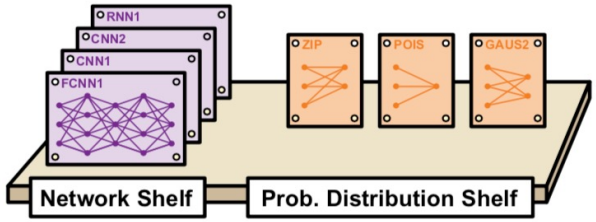


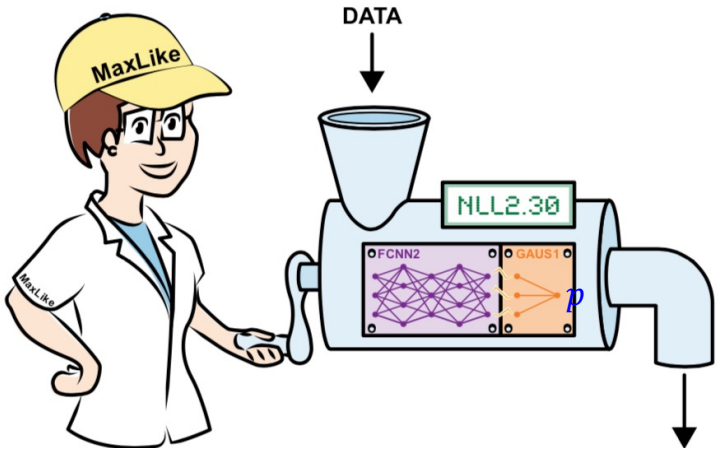
Guiding Theme of the course

- We treat DL as probabilistic models to model a conditional outcome distribution $P(y|x)$
- The models are fitted to training data with maximum likelihood (or Bayes)

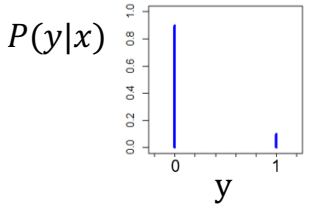
Input X determines choice of the NN architectures:

- Tabular data → fcNN
- Images → CNN
- Text → Transformer





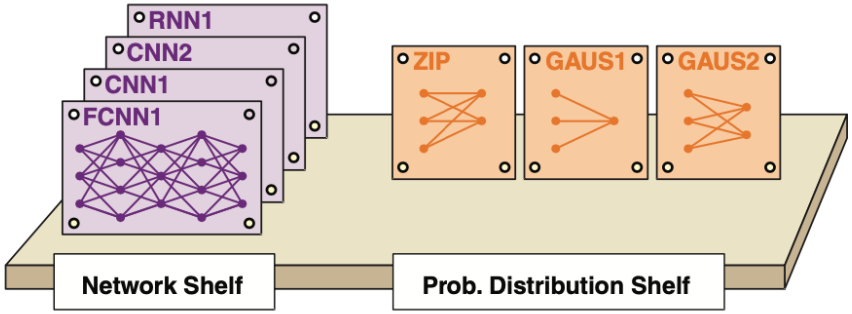
(Y|x)~Ber($p(Y = 1|x)$)



Family of outcome distribution, parameters are controlled by NN:

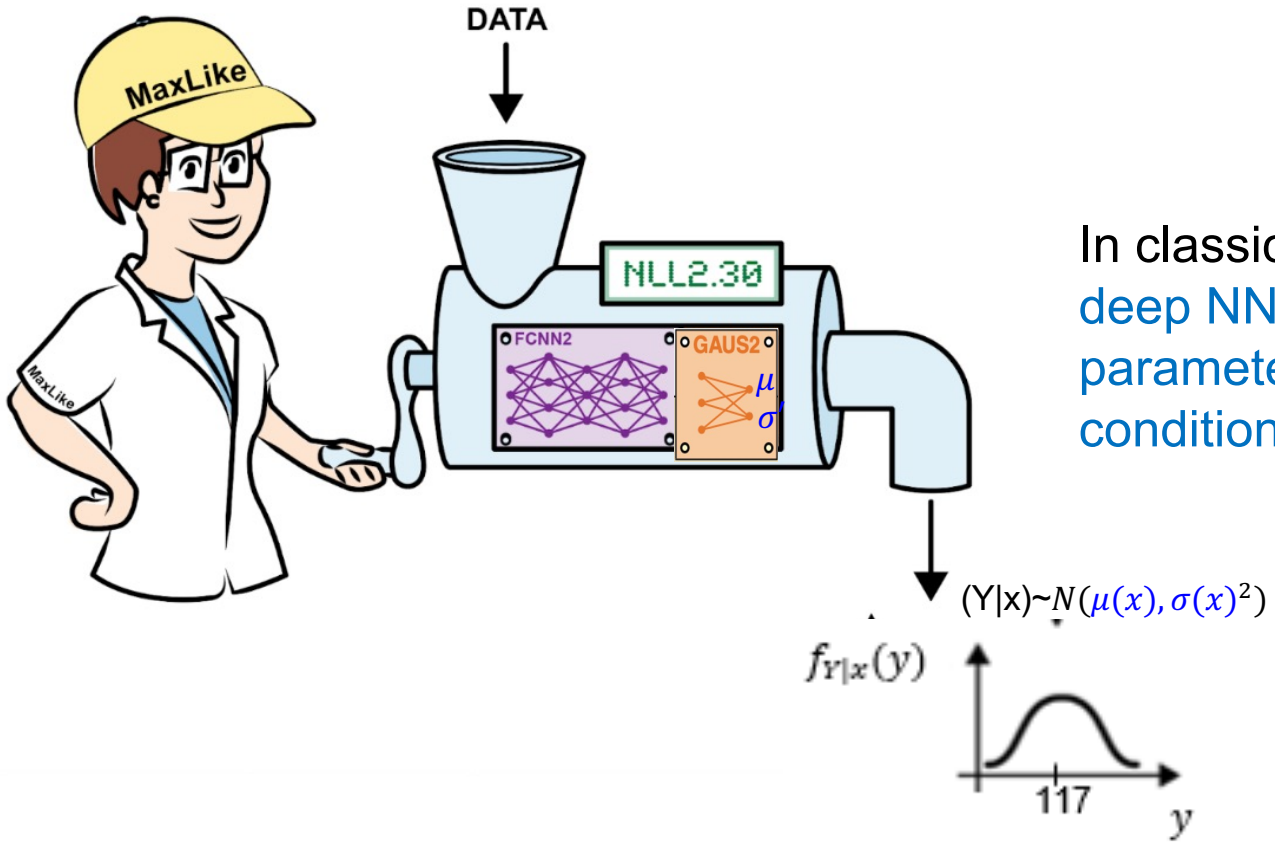
- Bernoulli (y: binary)
- Multi-Nomial (y: classes)
- Poisson (y: count)
- Negative Binomial (y: count)
- Gaussian (y: real number)
- Weibull (y: real number)
- …

Probabilistic Deep Learning



Family of outcome distribution

- Gaussian
- Bernoulli
- Poisson
- …



In classical probabilistic deep learning deep NNs are used to estimate the parameters of a pre-defined conditional outcome distribution.

O. Dürr, B. Sick, E. Murina. Probabilistic deep learning: With python, keras and tensorflow probability. Manning Book (2020)

How to use a NN to do Logistic Regression?

Recall logistic regression from statistics

$Y \in \{0,1\}$

$(\mathrm{Y}|\mathrm{X}_i) \sim \mathrm{Ber}(p_{x_i})$

$$\log\left(\frac{p_{x_i}}{1-p_{x_i}}\right) = \beta_0 + \beta_1 x_{i1}$$

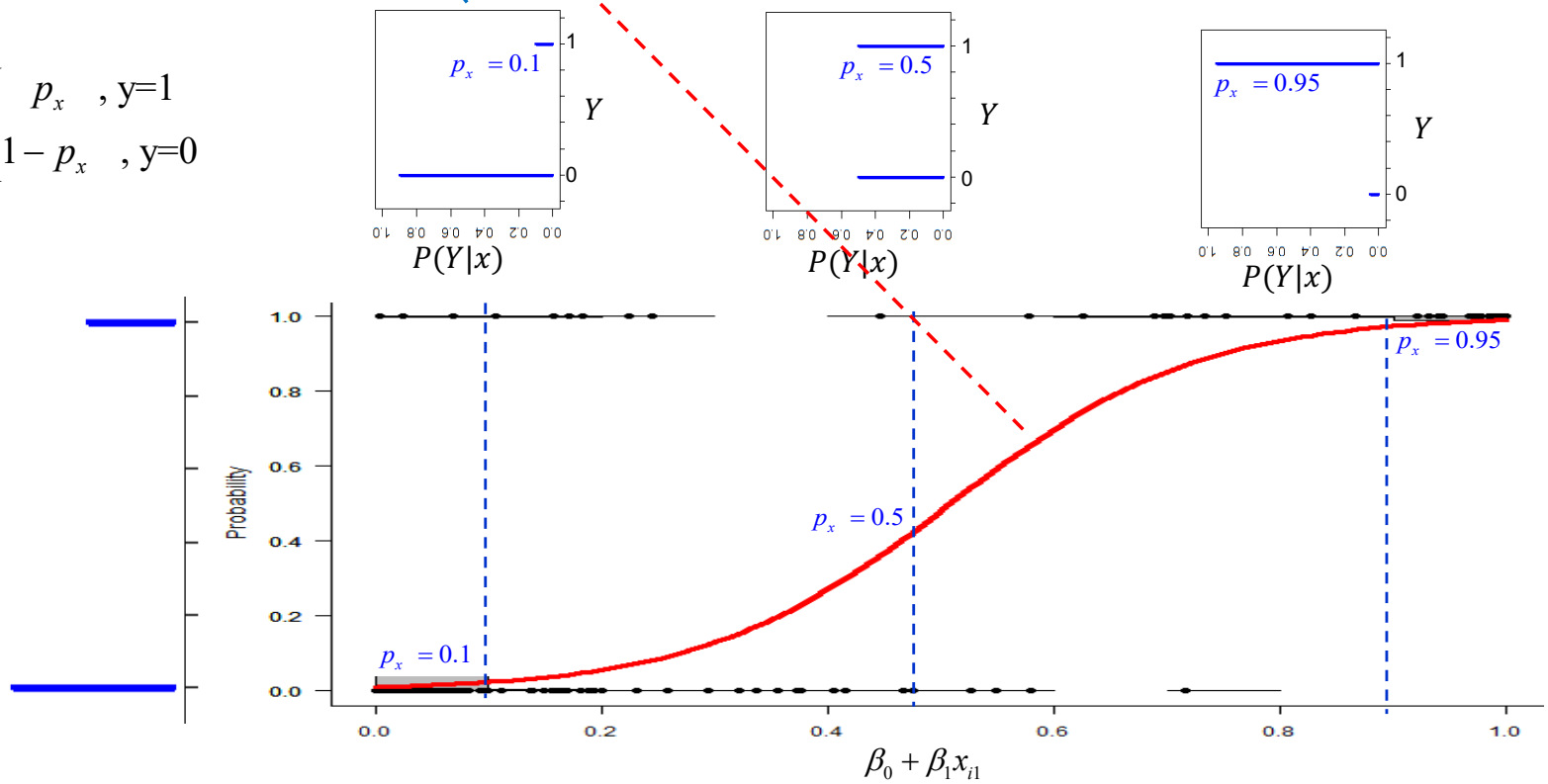
$$\mathrm{P}(Y=1|x_i) = p_{x_i} = \sigma(\beta_0 + \beta_1 x_{i1}) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_{i1})}}$$

Predict Y=1 «neuron fires» if $p_x > 0.5$

probabilistic prediction

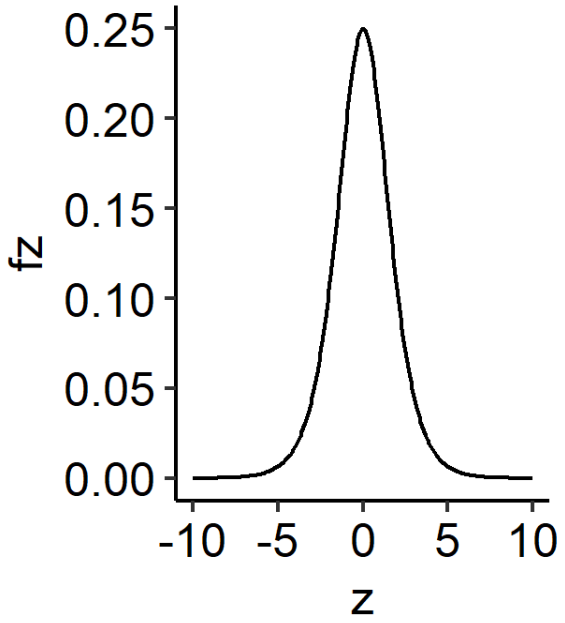
Logisic regression predicts a conditional Bernoulli $P(Y|x_i)$

$$P(\mathrm{Y} \mid \mathrm{X}=\mathrm{x}) = \begin{cases} p_x & , \mathrm{y}=1 \\ 1-p_x & , \mathrm{y}=0 \end{cases}$$



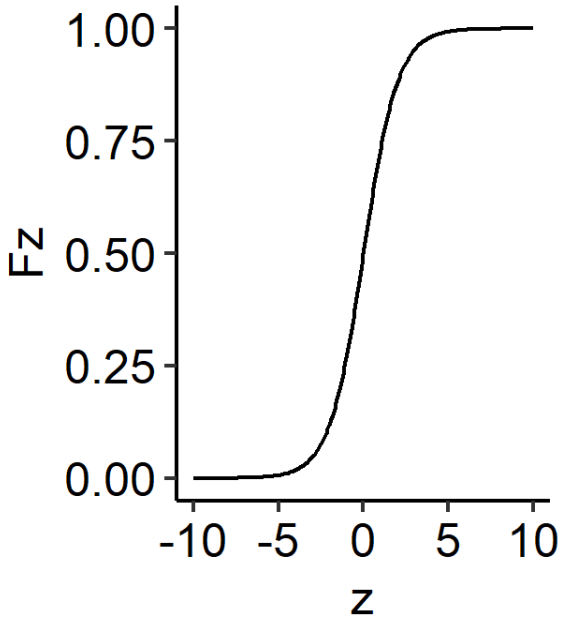
Recall the Standard Logistic Distribution

PDF (f_Z)



$$f_Z(z) = \frac{e^{-z}}{(1+e^{-z})^2}$$

CDF (F_Z)

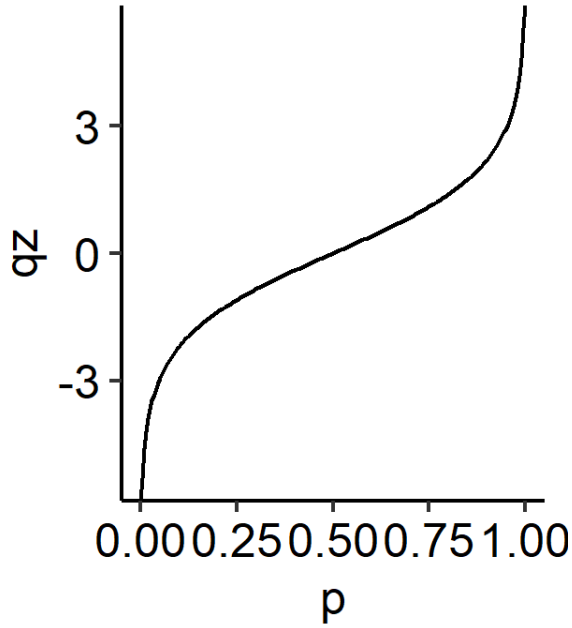


$$F_Z(z) = \frac{1}{1+e^{-z}}$$

$$F_Z(z) = \text{expit}(z)$$

aka “sigmoid”

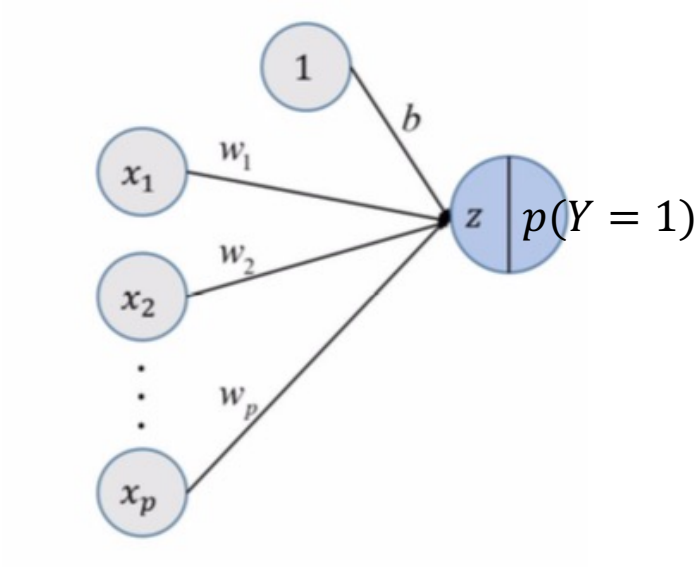
Quantiles function (F_Z^{-1})

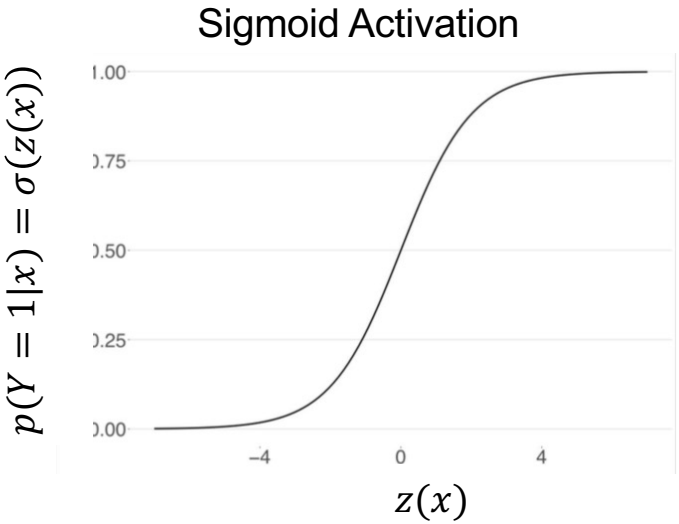


$$F_Z^{-1}(p) = \log\frac{p}{1-p}$$

$$F_Z^{-1}(p) = \log(\text{odds})$$
$$= \text{logit}(p)$$

Logistic regression as NN
Activation in the last layer: "sigmoid"



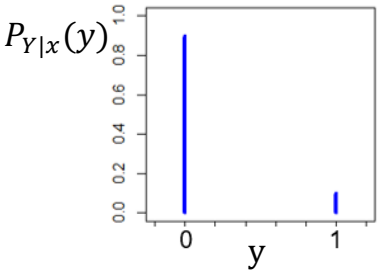


$$z(x) = b + x_1 \cdot w_1 + x_2 \cdot w_2 + \cdots x_p \cdot w_p$$

$$p(Y = 1|x) = \sigma(z) = \sigma\left(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_{ip} x_{ip}\right) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_{ip} x_{ip})}}$$

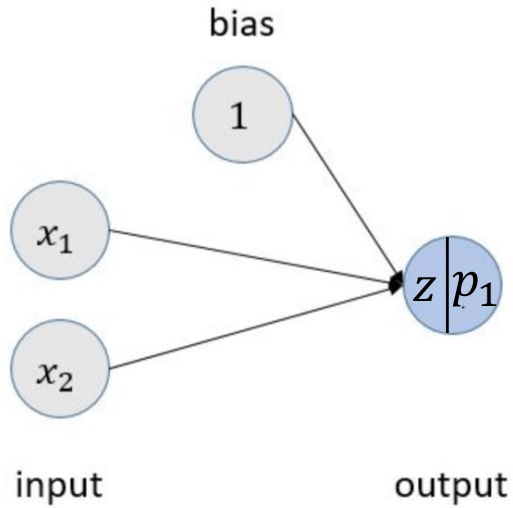
The output after the sigmoid activation:

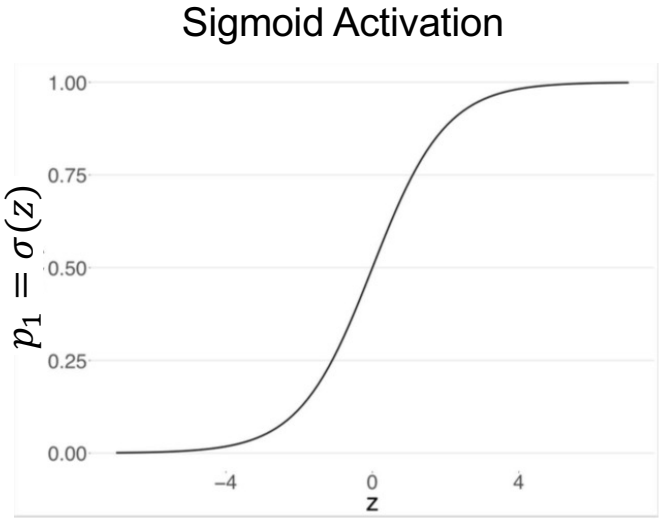
$$P(y_i = 1|x_i) = \sigma\left(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_{ip} x_{ip}\right)$$



→ A NN without hidden layers and
with one output node and a sigmoid-activation is a logistic regression model!

Exercise: Part 1





Model: The above network models the **probability** p_1 that a given banknote is false.

TASK (with pen and paper)

The weights (determined by a training procedure later) are given by

$$w_1 = 0.3, w_2 = 0.1, \text{ and } b = 1.0$$

What is the probability that a banknote, that is characterized by x_1=1 and $x_2 = 2.2$, is a faked banknote?

Colab Demo [Demo Time]

- Explain Colab based on
  - https://github.com/tensorchiefs/dlwbl_eth25/blob/master/notebooks/00_colab_intro_forR.ipynb

Exercise run your first network

Notebook: 01_fcnn_with_banknotes

Play with the code until

Calculate the probability that a banknote is fake

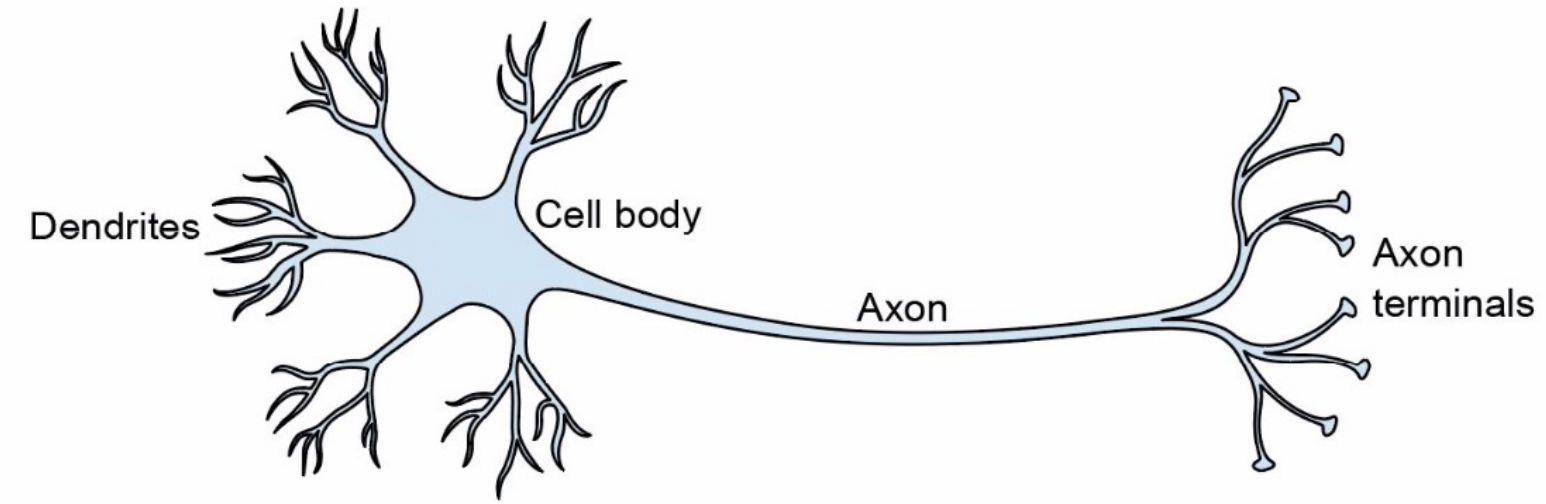
The banknote is described by two features $x_1 = 1$ and $x_2 = 2.2$ and the weights of the trained NN are $w_1 = 0.3$ and $w_2 = 0.2$ and the bias is $b = 1$. Use the following python function to calculate the probability that the banknote is fake.

```
[ ]  1 def predict_prob(x1, x2, w1, w2, b):
     2     return 1/(1+np.exp(-(w1*x1 + w2*x2 + b)))
```

Solution Code

Code anzeigen

Neuron in a neural network: Biological Motivation



Sketch shows a model of single neuron in a brain.
The Dendrites collect signal from other neurons and accumulate it.
If signal exceeds a certain value the neuron “fires”, meaning a signal is sent to other neurons that coupled via the axon terminals

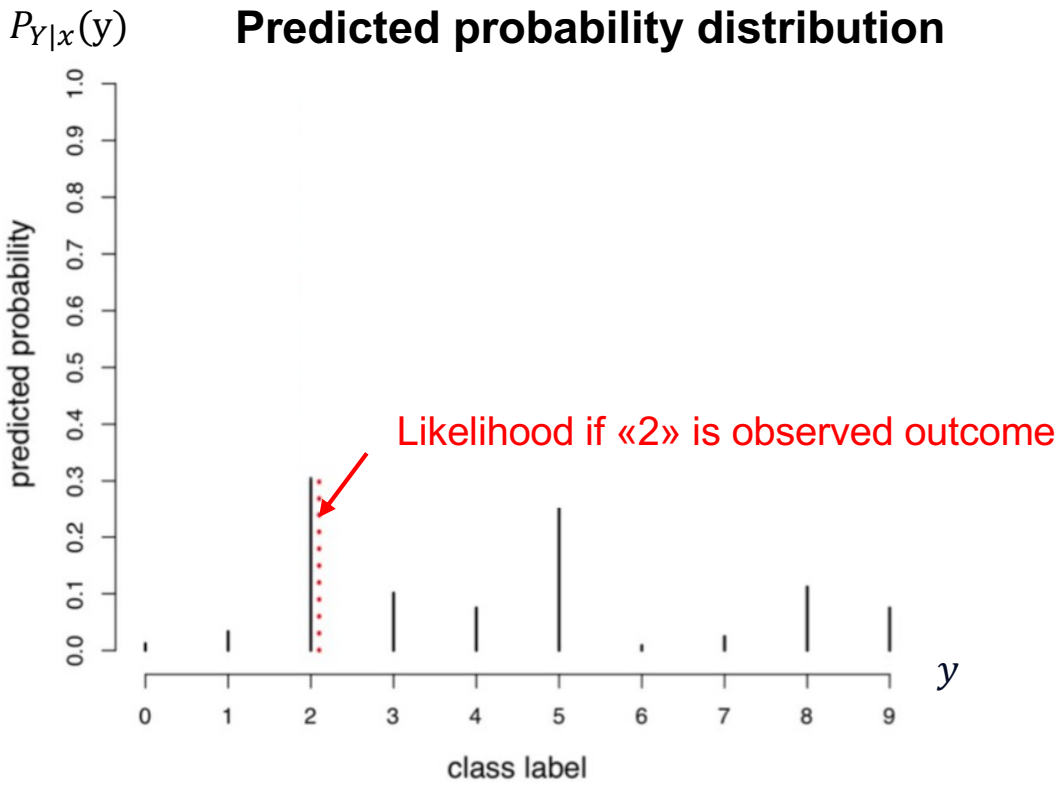
Neural networks are **loosely** inspired by how the brain works

How to use a NN to do Multinomial Regression?

Multinomial regression aka multi-class regression

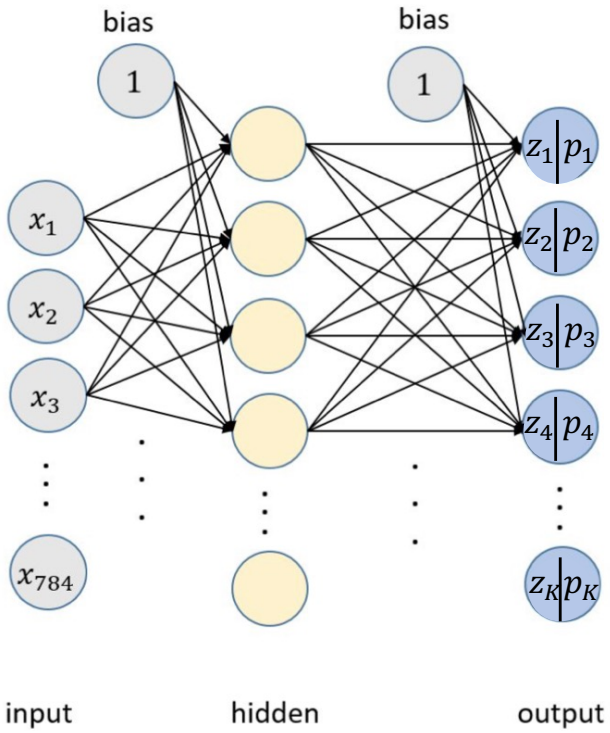
Multinomial regression is used when the outcome variable Y is categorial nominal and has more than two categories, $c_1, c_2, \ldots, c_k$, that do not have an order.

$Y \in \{c1, c2, \ldots, cK\}$, $P_{Y|x}(Y = c_k) = P(Y = \mathrm{c_k}|\mathrm{x}) = \mathrm{p_k(x)}$, $\sum_{k=1}^{K} p_k(x) = 1$



Remark: It Y is not nominal but ordinal, often a proportional odds model is fitted which does assume more than just $\sum_k p_k = 1$.

Multi-class regression as NN
Activation in the last layer: "softmax"



$p_0, p_1 \ldots p_9$ are probabilities for the classes c_1 to c_K.

Incoming to last layer z_i $i = 1 \cdots K$

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^{K} e^{z_j}}$$

Makes outcome positive

Ensures that pi's sum up to one

This activation is called *softmax*

NN with "softmax" activation in last layer outputs the probabilities for the classes.

How to use a NN to do Linear Regression?

Recall linear regression from statistics

$$(Y|X = x_i) \sim N(\mu(x_i), \sigma^2)$$

$Y \in \mathbb{R}$,
$\mu_x \in \mathbb{R}$

$$y_i = \mu(x_i) + \varepsilon_i = \beta_0 + \beta_1 \cdot x_{i1} + \varepsilon_i$$
$$\hat{E}(Y_{X_i}) = \hat{\mu}_{x_i} = \hat{\beta}_0 + \hat{\beta}_1 \cdot x_{i1}$$
$$\mathrm{Var}(Y_{X_i}) = \mathrm{Var}(\varepsilon_i) = \sigma^2$$
$$\varepsilon_i \sim N(0, \sigma^2)$$

Probabilistic linear regression predicts for each input x_i a Gaussian conditional probability distribution for the output $P(Y|x_i)$ that assigns each possible value of Y a likelihood.

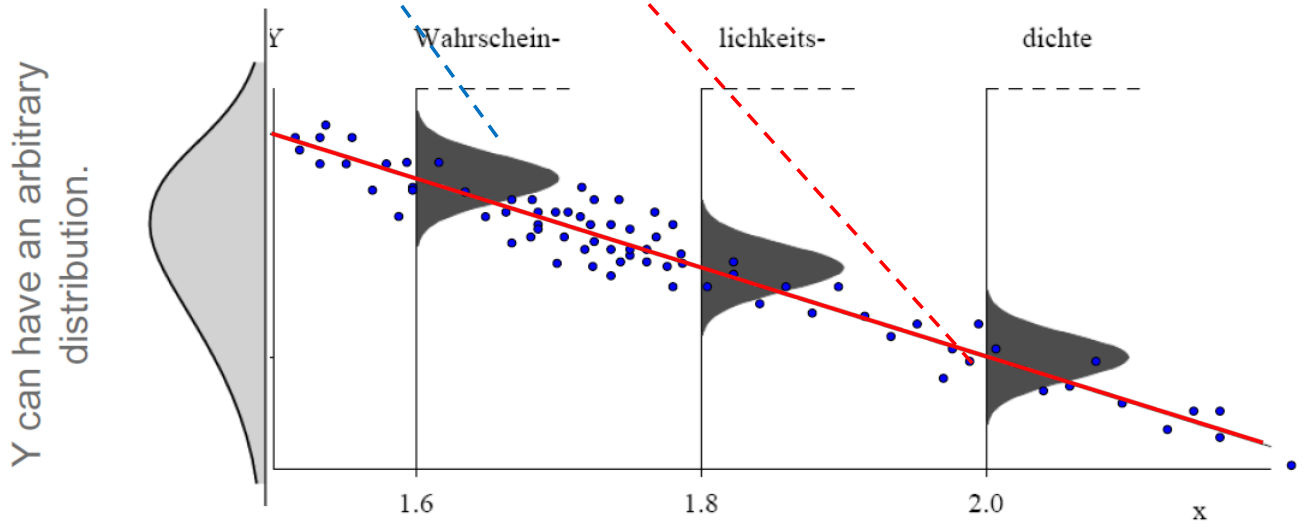
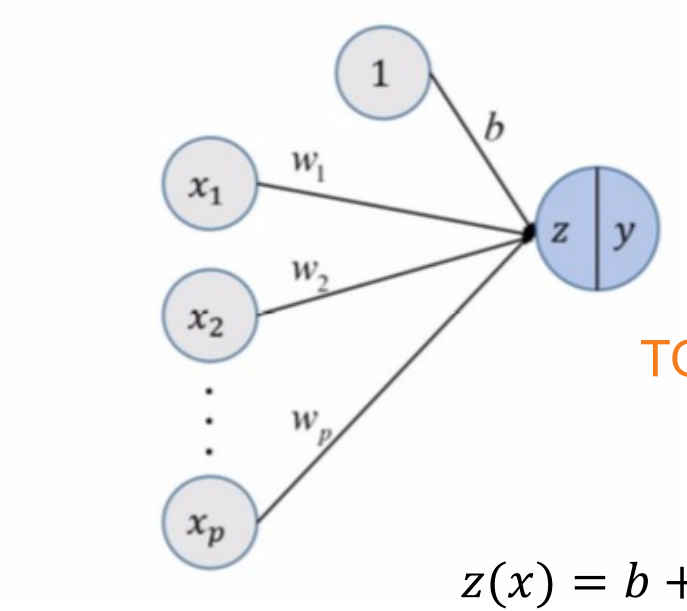
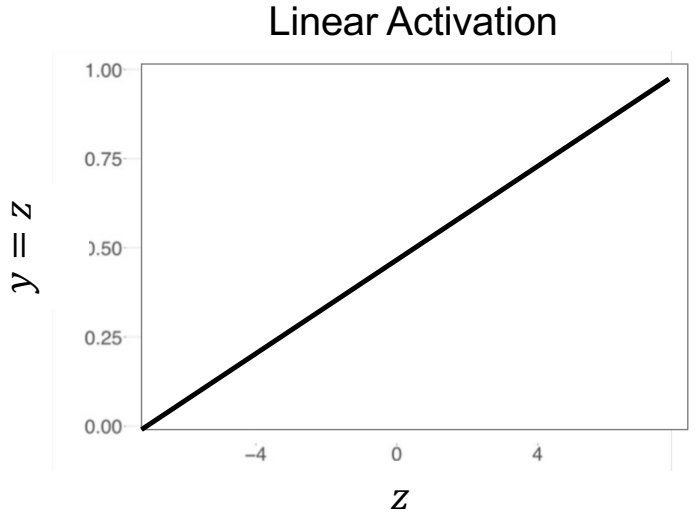


Image credits: Book of Werner Stahel: Angewandte Statistik

Linear Regression as NN
Activation in the last layer: "linear"



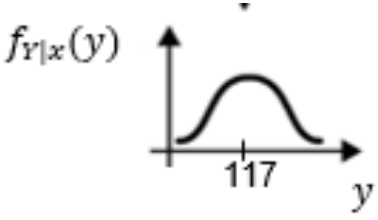
TODO mu



$$z(x) = b + x_1 \cdot w_1 + x_2 \cdot w_2 + \cdots x_p \cdot w_p$$

$$y(x) = \mu(x) = z(x)$$

$(Y|x) \sim N(\mu(x), \sigma^2)$



The output after the linear activation can be interpreted as estimated expeced value

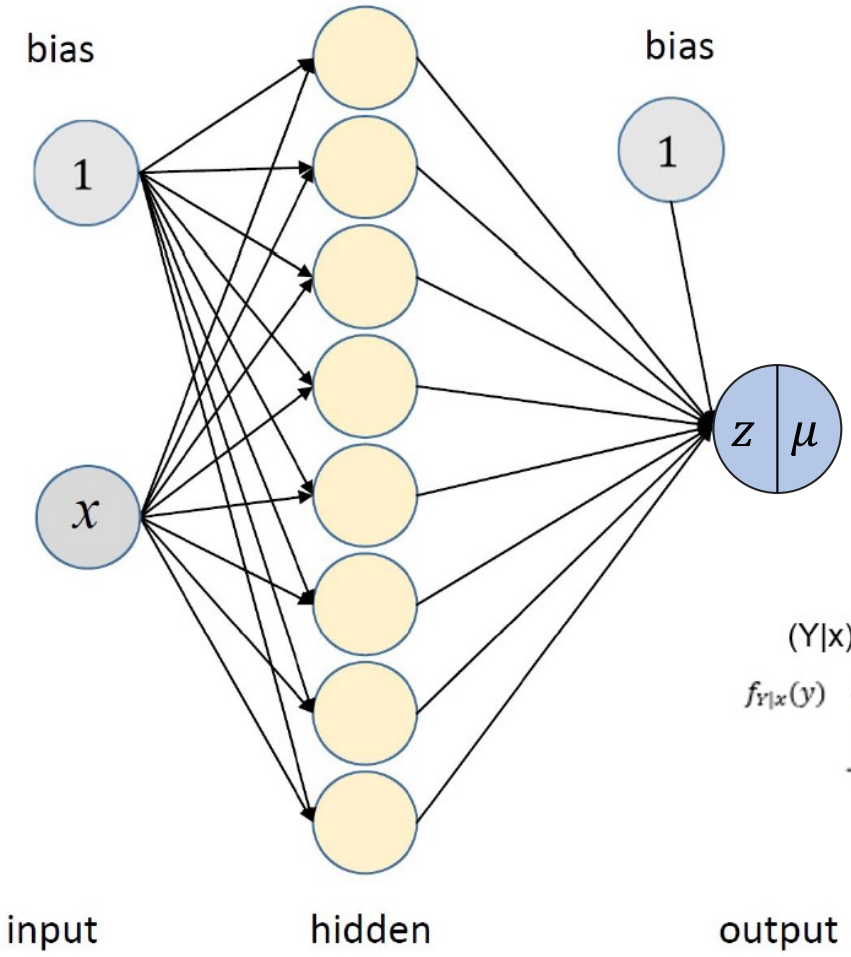
$y = E(Y|x) = \mu(x)$

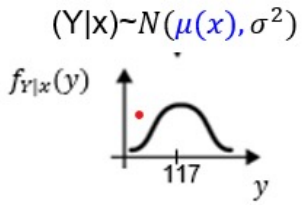
→ A fcNN without hidden layers and with one output node and a sigmoid-activation is a logistic regression model!

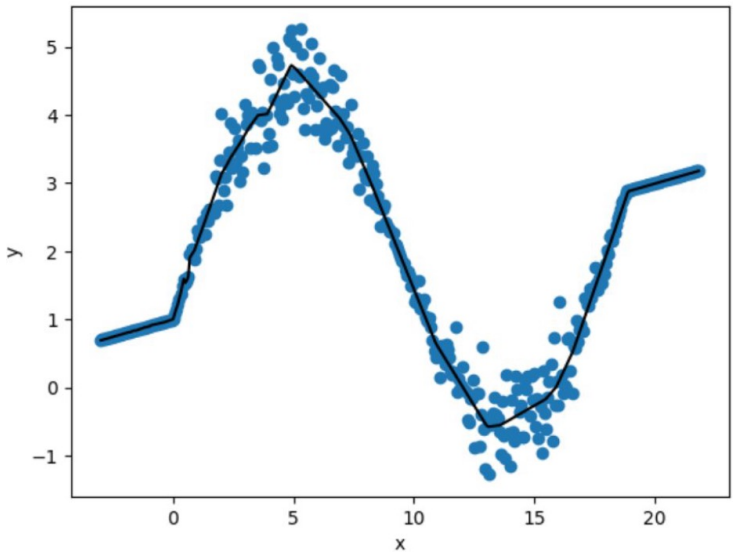
How to use a NN to do Non-linear Regression?


WE NEED TO GO
DEEPER
memegene

Non-linear Regression as NN
Activation in hidden layer "sigmoid" in last layer: "linear"







Here, the conditional outcome distribution is a Gaussian with a μ that depends on x and a constant variance σ^2

Remark: It has been shown, that a NN with one hidden layers can approximate any function, if the the hidden layer holds enough neurons.

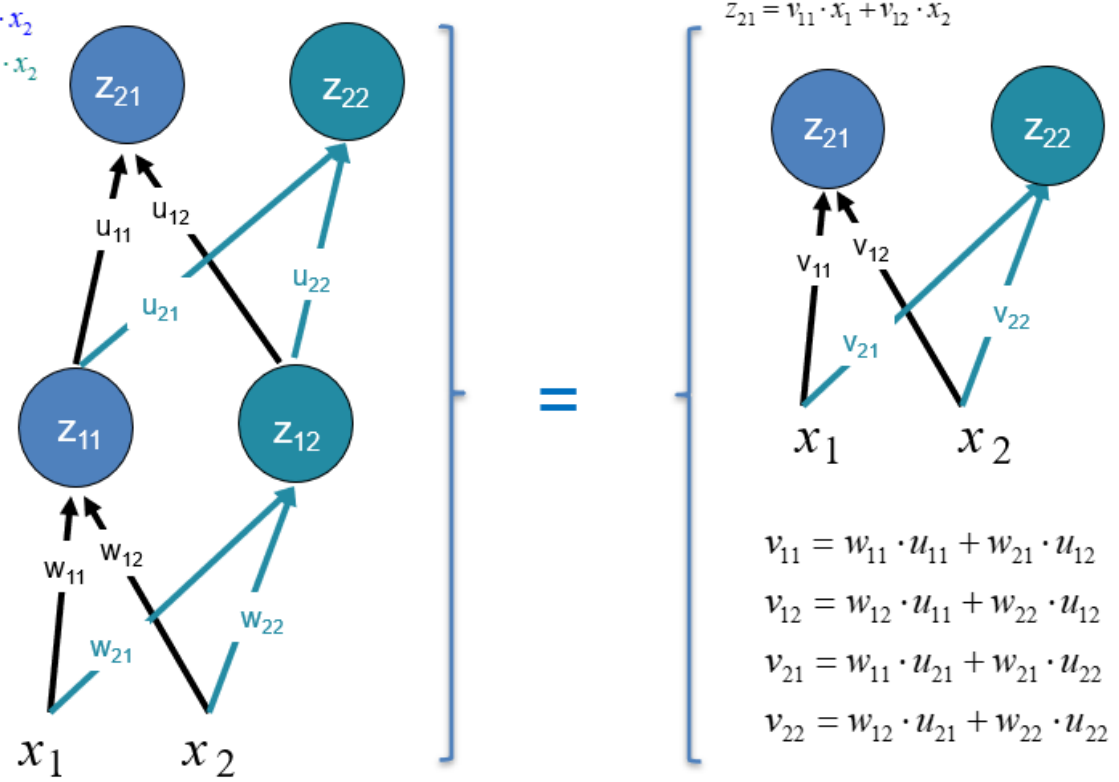
Hidden layer need non-linear activation functions

2 linear layers can be replaced by 1 linear layer -> can't go deep with linear layers!

$$z_{21} = z_{11} \cdot u_{11} + z_{12} \cdot u_{12} = (w_{11} \cdot x_1 + w_{12} \cdot x_2) \cdot u_{11} + (w_{21} \cdot x_1 + w_{22} \cdot x_2) \cdot u_{12}$$
$$= x_1 \cdot (w_{11} \cdot u_{11} + w_{21} \cdot u_{12}) + x_2 \cdot (w_{12} \cdot u_{11} + w_{22} \cdot u_{12})$$

$z_{11} = w_{11} \cdot x_1 + w_{12} \cdot x_2$

$z_{12} = w_{21} \cdot x_1 + w_{22} \cdot x_2$



$$v_{11} = w_{11} \cdot u_{11} + w_{21} \cdot u_{12}$$
$$v_{12} = w_{12} \cdot u_{11} + w_{22} \cdot u_{12}$$
$$v_{21} = w_{11} \cdot u_{21} + w_{21} \cdot u_{22}$$
$$v_{22} = w_{12} \cdot u_{21} + w_{22} \cdot u_{22}$$

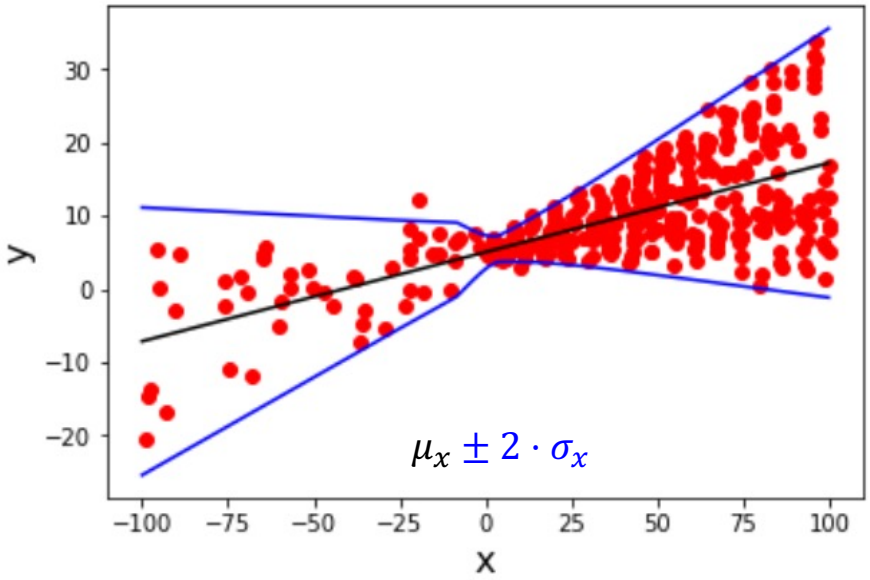
Computation NNs with linear activation are matrix multiplications → GPUs

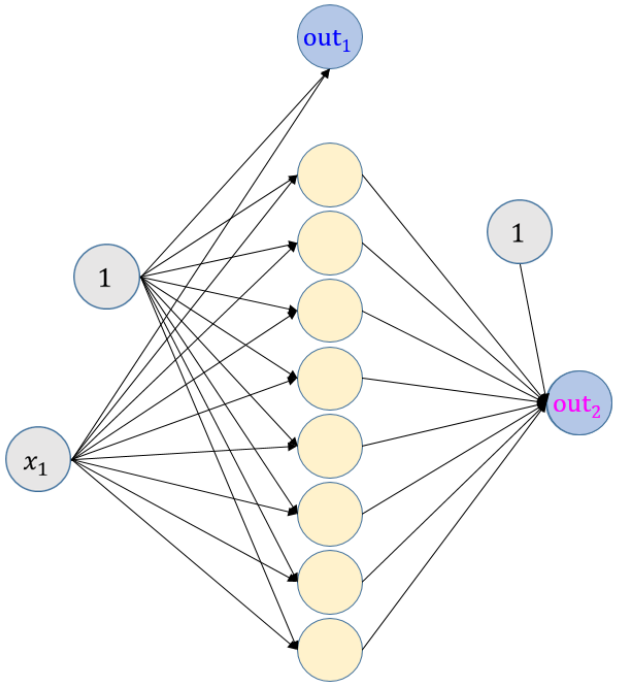
$$z = (x \cdot W) \cdot U = x \cdot (W \cdot U) = x \cdot V$$

Remark: biases are ignored here, but do not change fact

NN for linear regression with not-constant variance

$$Y_{X_i} = (\mathrm{Y}|\mathrm{X}_i) \sim \mathrm{N}(\mu_{x_i}, \sigma_x^2)$$



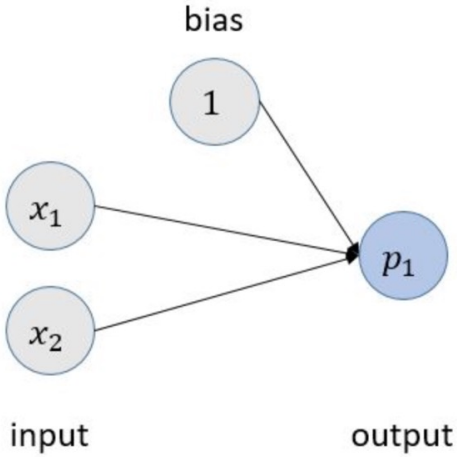


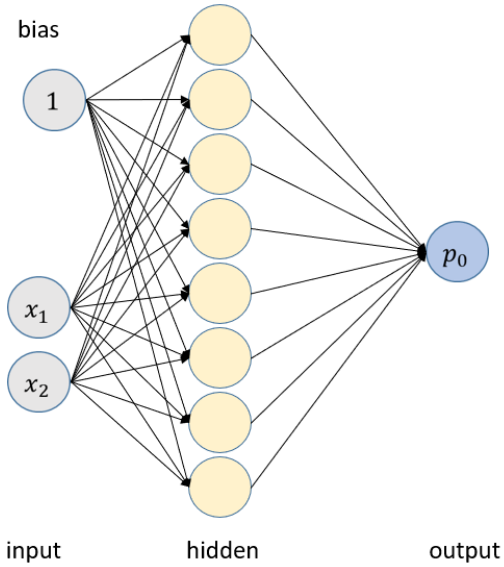
$$\mu_{x_i} = \mathrm{out}_{1_i}$$

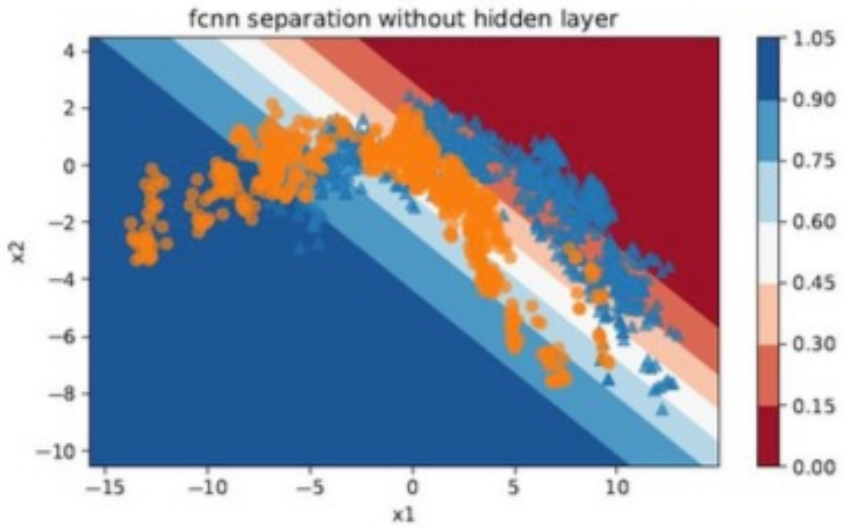
$$\sigma_{x_i} = e^{\mathrm{out}_{2_i}}$$

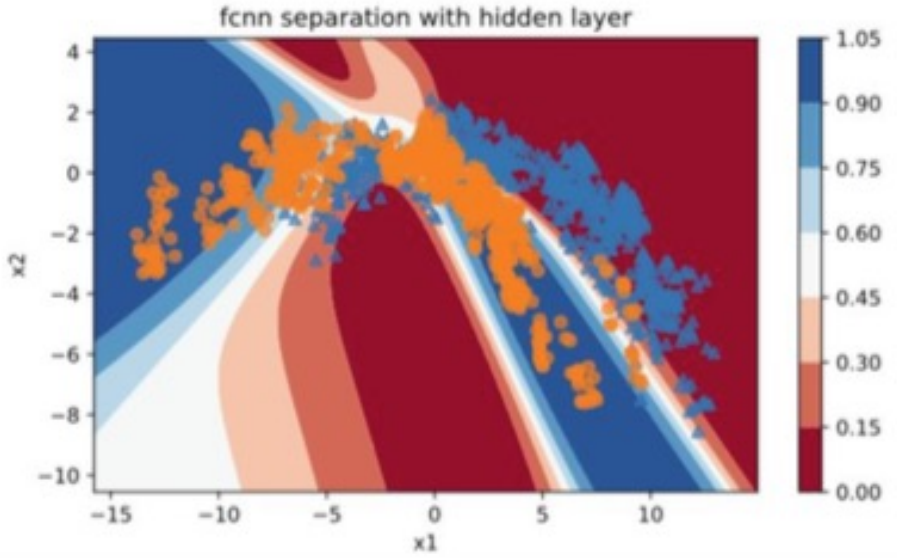
Note: You will see soon that the maximum likelihood principle allows to learn the variance σ_x of the conditional Gaussian without requiring a ground truth for σ_x.

Non-linear decision boundary in binary classification
Activation in hidden layer "ReLu" in last layer: "linear"









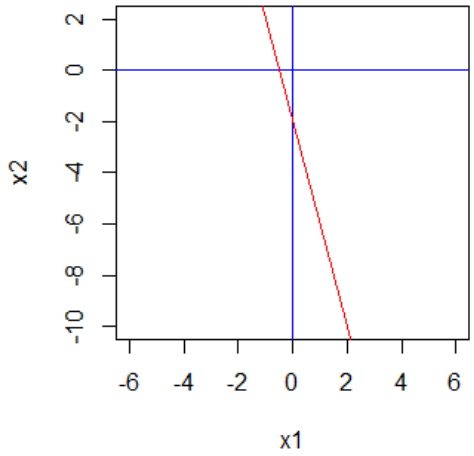
Logistic regression yield linear/planar decision curves

Logistic regression model: $\ln\left(\frac{p}{1-p}\right) = 1 + 2x_1 + 0.5x_2$

Determine the separation curve between Y=1 and Y=0 in the feature room which is spanned by x_1 and x_2 and draw it in the following plot x_2 and x_2.

Hint: on the separation curve should hold: $P(Y = 1|x) = 0.5$

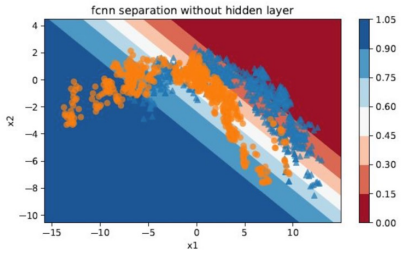
-> plug in 0.5 for p and solve for x_2.



$$\ln\left(\frac{0.5}{1-0.5}\right) = 0 = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

$$x_2 = -\frac{\beta_0}{\beta_2} - \frac{\beta_1}{\beta_2} \cdot x_1$$

$$x_2 = -2 - 4 \cdot x_1$$



Hence a NN with 1 output neuron with sigmoid-activation w/o hidden layer have linear decision boundary

Deep Learning Libraries

Deep-learners talk about tensors - What is a tensor?

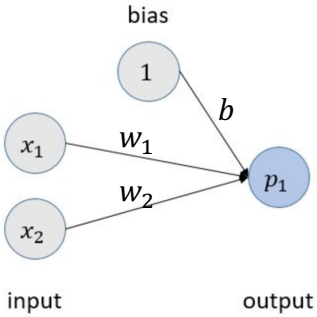
For Deep Learning: A tensor is an array with several indices (like in numpy). Order (a.k.a rank) is the number of indices and shape is the range.

```
In [1]: import numpy as np

In [2]: T1 = np.asarray([1,2,3]) #Tensor of order 1 aka Vector
        T1

Out[2]: array([1, 2, 3])

In [3]: T2 = np.asarray([[1,2,3],[4,5,6]]) #Tensor of order 2 aka Matrix
        T2

Out[3]: array([[1, 2, 3],
               [4, 5, 6]])

In [4]: T3 = np.zeros((10,2,3)) #Tensor of order 3 (Volume like objects)

In [6]: print(T1.shape)
        print(T2.shape)
        print(T3.shape)

        (3,)
        (2, 3)
        (10, 2, 3)
```

Tensors (as a generalization of matrix and vectors) are the basic quantities in DL.

GPUs love Vectors

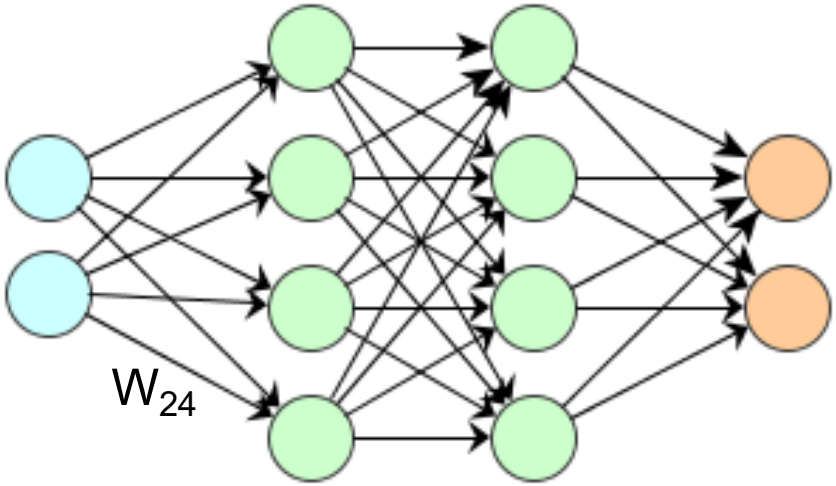
$F^{\mu\nu}$



$$p_1 = \text{sigmoid}\left(\begin{pmatrix} x_1 & x_2 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} + b \right)$$

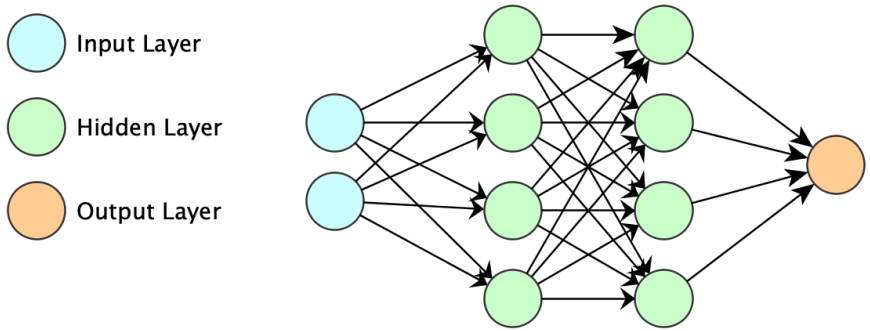
DL: better to have column vectors

Typical Tensors in Deep Learning



- The input can be understood as a vector
- A mini-batch of size 64 of input vectors can be understood as tensor of order 2
  - (index in batch, x_j)
- The weights going from e.g. Layer L_1 to Layer L_2 can be written as a matrix (often called W)
- A mini-batch of size 64 images with 256,256 pixels and 3 color-channels can be understood as a tensor of order 4.

Structure of the network



In code:

```
## Solution 2 hidden layers
def predict_hidden_2(X):
    hidden_1=sigmoid(np.matmul(X,W1)+b1)
    hidden_2=sigmoid(np.matmul(hidden_1,W2)+b2)
    return(sigmoid(np.matmul(hidden_2,W3)+b3))
```

In math (f = sigmoid) and b1=b2=b3=0

$$p = f(f(f(\mathrm{x}\,\mathrm{W}^1)\mathrm{W}^2))$$

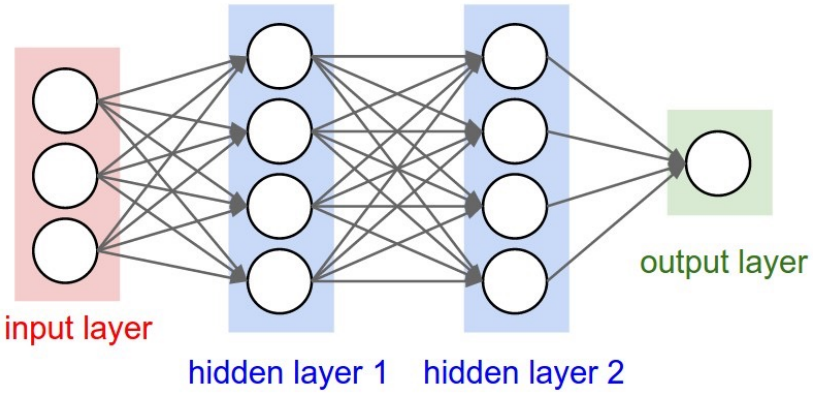
Looks a bit like onions, matryoshka (Russian Dolls) or lego bricks

Keras as High-Level library

Keras

We use Keras as high-level library

Libraries make use of the Lego like block structure of networks



High Level Libraries

Keras

Under the “hood” of Keras runs the engine;

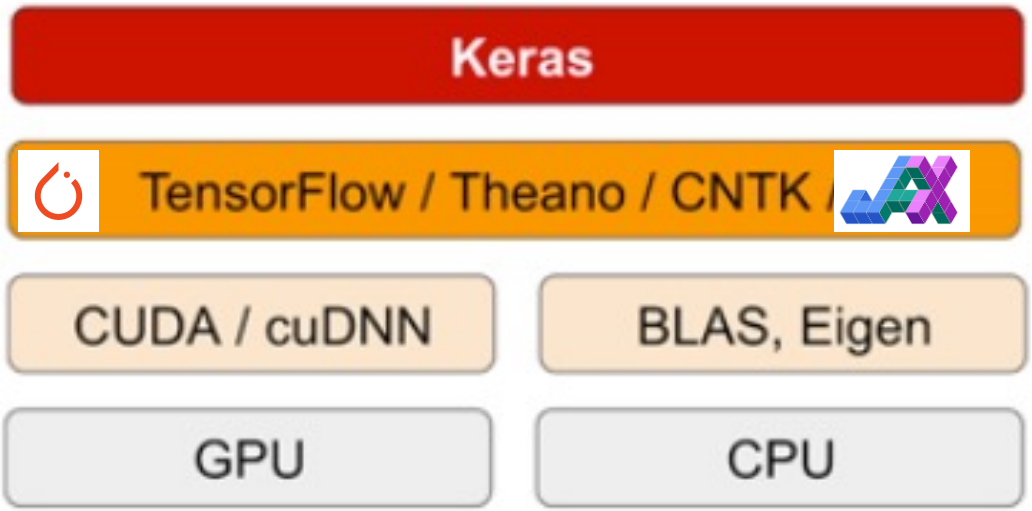


Figure: From Deep Learning with Python, Francois Chollett

Keras Workflow

Keras

Load packages
- Select “Engine” (before `import keras`)

PyTorch JAX TensorFlow

```
import os
os.environ["KERAS_BACKEND"] = "torch"    # or 'jax' or 'tensorflow'
import keras
```

Load building blocks
- Layers, functions etc.

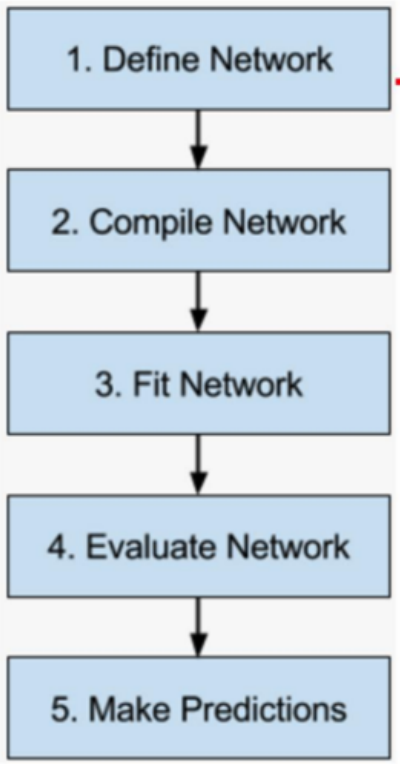
```
from keras.models import Sequential
from keras.layers import Dense,
from keras.optimizers import SGD
```

Define the network

Sequential API, layers output are the input for the next layer, Alternative Functional API

Keras

Hidden layer with 8 Neurons, Activation function: sigmoid




```
# Define fcNN
model = Sequential()
model.add(Dense(8,
                input_shape=(2,),
                activation='sigmoid'))
model.add(Dense(1,
                activation='sigmoid'))
```

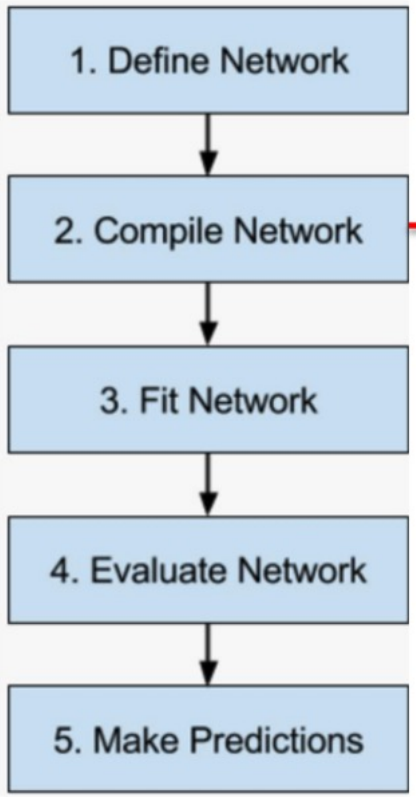
Last layer with one output neuron for binary classification

Input shape needs to be defined only at the beginning.

Alternative: `input_dim=2`, `Functional API` or `Sequential API`

Compile the network

Keras




```
model.compile(optimizer=SGD(learning_rate=0.01)
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

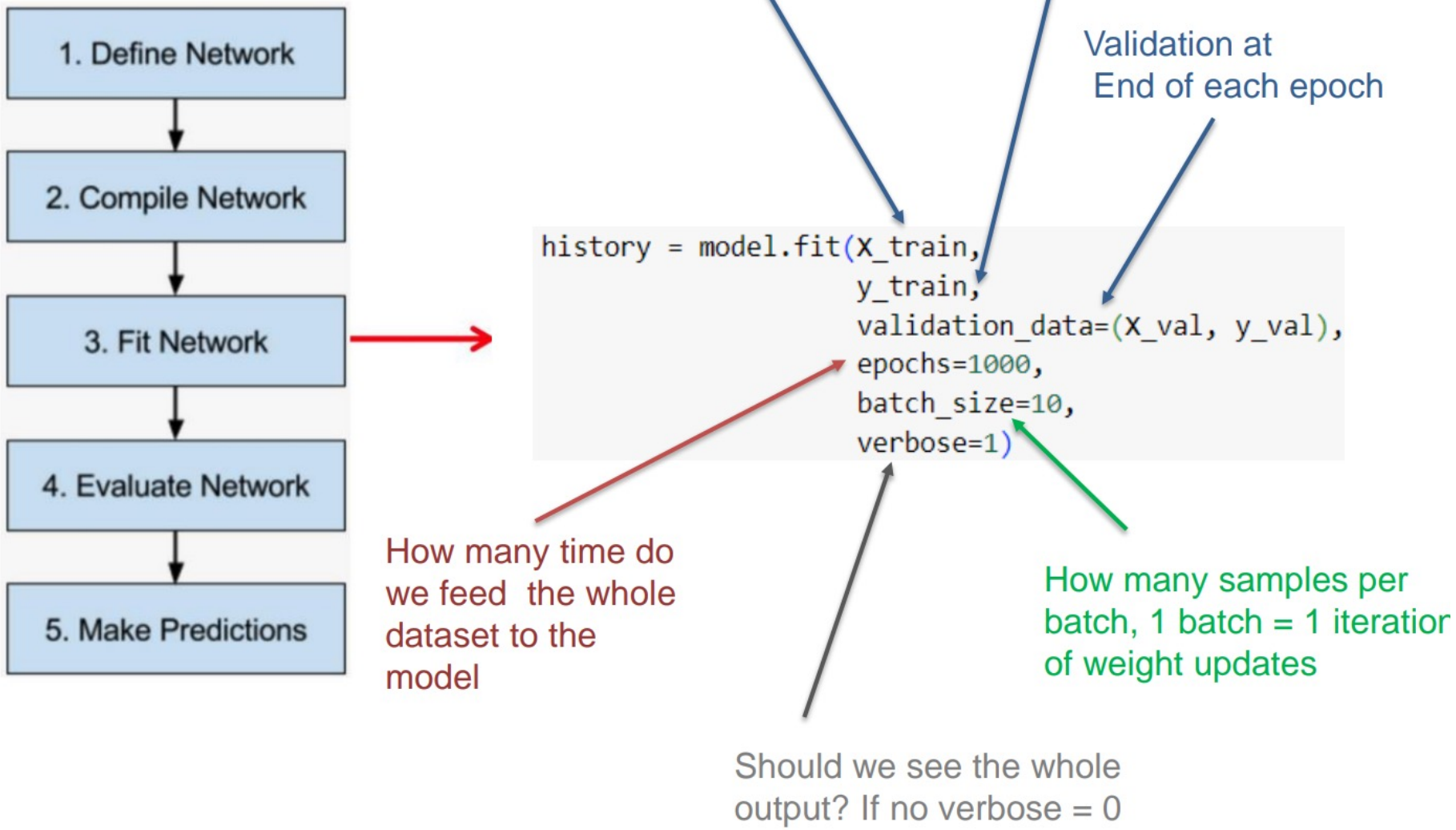
Which optimizer should be used?
Here Stochastic Gradient Descent

Loss Function to optimize
Here: Binary CE

Which metrics do we want to track,
Here: Accuracy

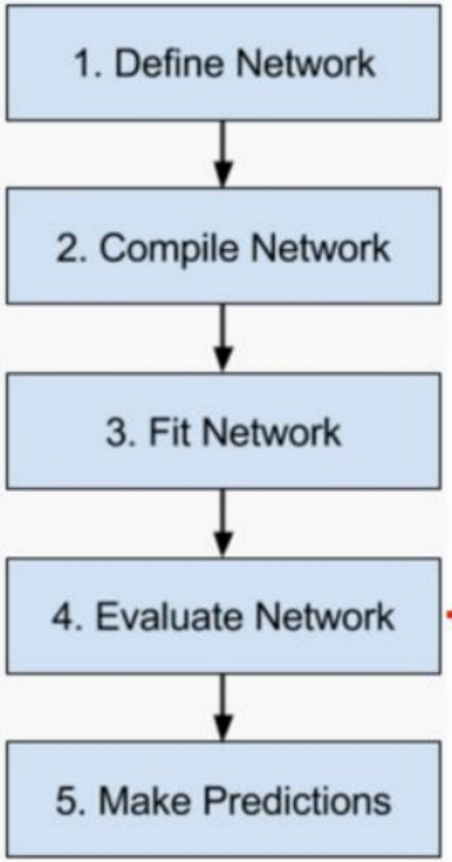
Fit the network

Keras



Evaluate the network

Keras



Unseen (to the model) Test Data X with labels Y

```
model.evaluate(X_test, y_test, verbose=0)
```

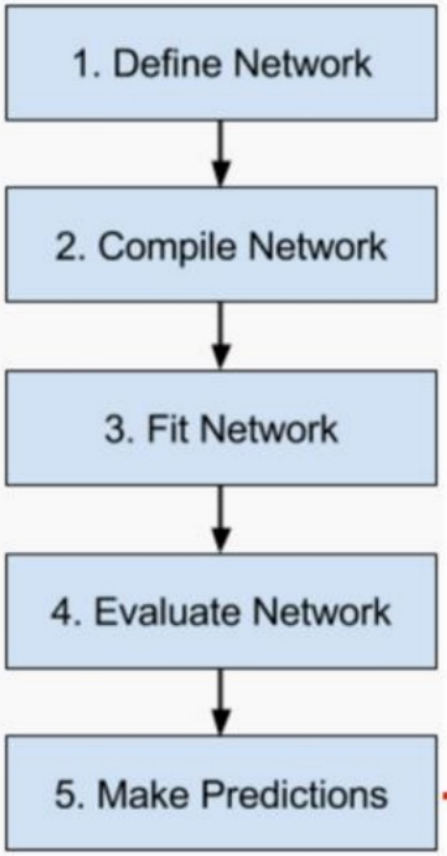
```
Test Loss: 0.33591118454933167, Test Accuracy: 0.8581818342208862
```

Remember from the .compile

```
loss='binary_crossentropy',
metrics=['accuracy'])
```

Make Predictions

Keras



predict with the model,
Weights are fixed now

First 10 rows
(observations) from
Testdata

All features, here: 2

```
model.predict(X_test[0:10,:])
```

```
1/1 [==============================] - 0s 148ms/step
array([[5.9251189e-01],
       [8.8978779e-01],
       [7.8563684e-01],
       [9.1934395e-01],
       [6.1078304e-01],
       [8.2838058e-01],
       [2.5862646e-01],
       [3.1314052e-05],
       [2.6938611e-01],
       [3.7005499e-02]], dtype=float32)
```

Each of the observations
Gets a probability to
Belong to class 1

Building a network (API Styles)

Three API styles

- The Sequential Model
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks
  - Good for 70+% of use cases
- The functional API
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies
  - Good for 95% of use cases
- Model subclassing
  - Maximum flexibility
  - Larger potential error surface

Slides from Chollet (founder of Keras)

Sequential API

The Sequential API

```
import keras
from keras import layers

model = keras.Sequential()
model.add(layers.Dense(20, activation='relu', input_shape=(10,)))
model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.fit(x, y, epochs=10, batch_size=32)
```

Slides from Chollet (founder of Keras)

Functional (do you spot the error?)

The functional API

```
import keras
from keras import layers

inputs = keras.Input(shape=(10,))
x = layers.Dense(20, activation='relu')(x)
x = layers.Dense(20, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)
model.fit(x, y, epochs=10, batch_size=32)
```

Slides from Chollet (founder of Keras)

Functional (do you spot the error?)

The functional API

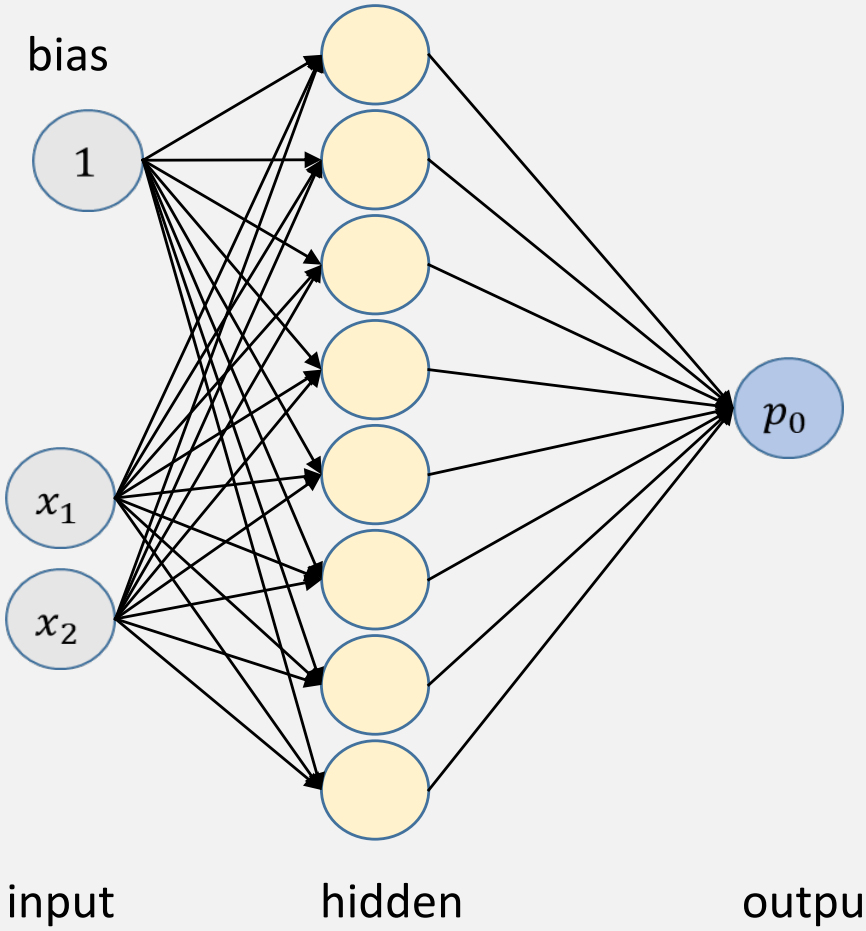
```
import keras
from keras import layers

inputs = keras.Input(shape=(10,))
x = layers.Dense(20, activation='relu')(x)  ←'inputs'
x = layers.Dense(20, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)
model.fit(x, y, epochs=10, batch_size=32)
```

Slides from Chollet (founder of Keras)

Exercise: use keras for a NN with hidden layers

Notebook: 01_fcnn_with_banknotes

Play around with the code!
We discuss afterwards the loss function and how fitting works



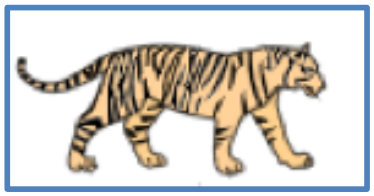
Fitting NNs via Maximum Likelihood principle

Training: Tune the weights to minimize the loss

Input $x^{(i)}$

True class $y^{(i)}$

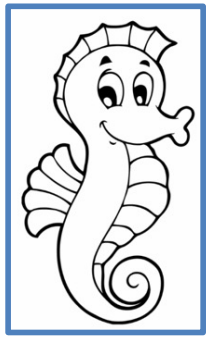
predicted class
(class with highest predicted probability)



Tiger → Lion 👎

Neural network with many weights w

Tiger → Tiger 👍



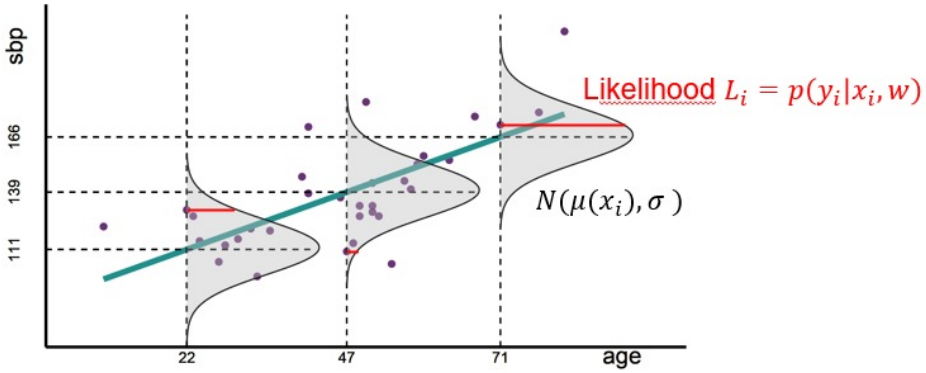
Seehorse → Seehorse 👍

...

Trainingsprinciple:
Weights are tuned so a loss functions gets minimized.

$$\hat{w} = \underset{w}{\operatorname{argmin}}\ \operatorname{loss}\left(\{y^{(i)}, x^{(i)}\}, w\right)$$

Recap: Maximum Likelihood (one of the most beautiful ideas in statistics)



Ronald Fisher in 1913
Also used before by Gauss, Laplace

Tune the parameters weights of the network such, that observed data (training data) is most likely under the predicted outcome distribution.

We assume iid training data → multiplication of likelihood over all data points:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{N} L_i = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{N} p(y_i|x_i, w)$$

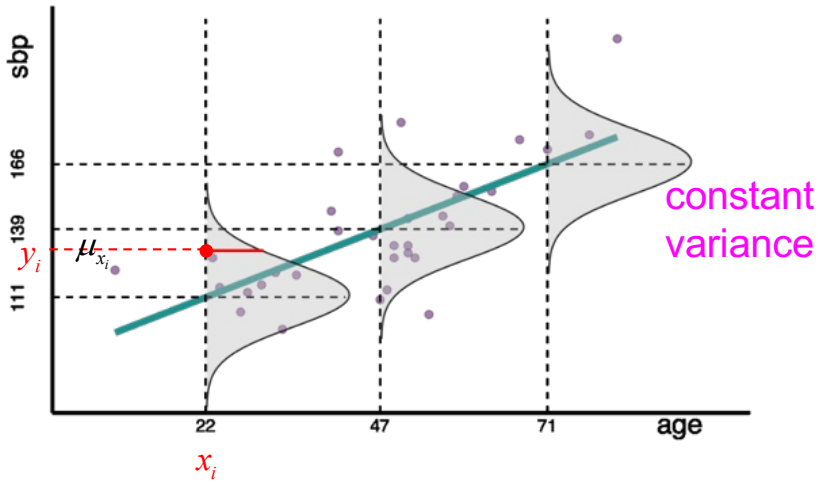
Practically: Use Negative Log-Likelihood NLL as loss and minimize NLL

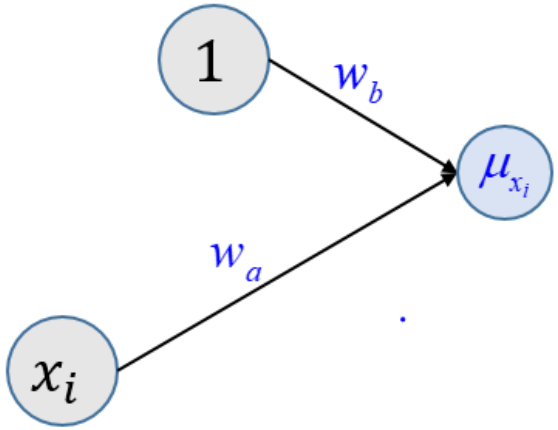
take log; minimize after multiplication with -1, note that NLL in DL is usually the mean-negative log-likelihodd

$$\hat{w} = \underset{w}{\operatorname{argmin}} \operatorname{NLL}\left(y^{(i)}, x^{(i)}, w\right) = \underset{w}{\operatorname{argmin}} \frac{1}{N} \sum_{i=i}^{N} -\log(p(y_i|x_i, w))$$

Fit "linear regression NN" via Maximum likelihood principle

$$Y_{X_i} \sim \mathrm{N}(\mu_{x_i} = w_a \cdot x_i + w_b \ , \sigma^2)$$





ML-principle:

$$\mathbf{w}_{\mathrm{ML}} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \, \mathrm{e}^{-\frac{\left(y_i-\mu_{x_i}\right)^2}{2\sigma^2}}$$

$$= \underset{w}{\operatorname{argmin}} \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \frac{\left(y_i-\mu_{x_i}\right)^2}{2\sigma^2}$$

Negative Log-Likelihood (NLL)

$$(\hat{w}_a, \hat{w}_b)_{\mathrm{ML}} = \underset{w_a, w_b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - (w_a \cdot x_i + w_b)\right)^2$$

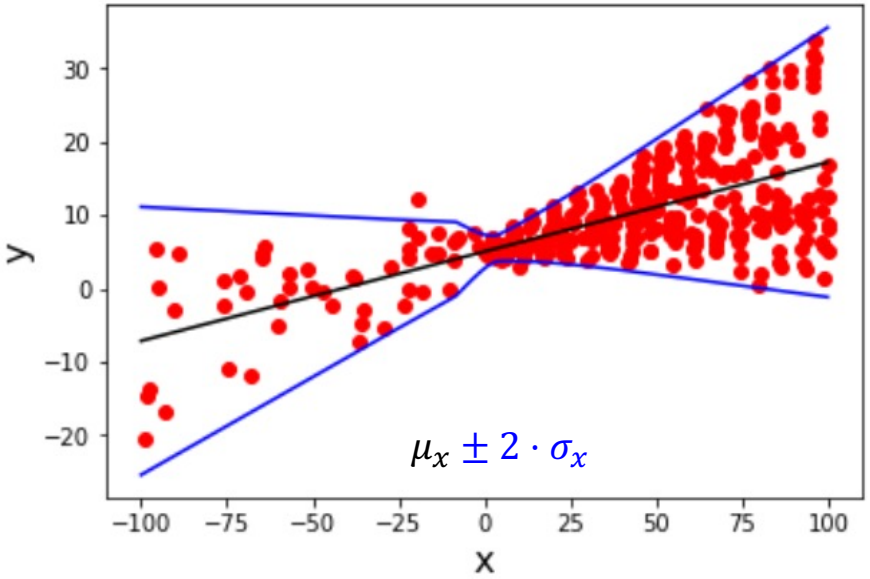
Minimize MSE loss

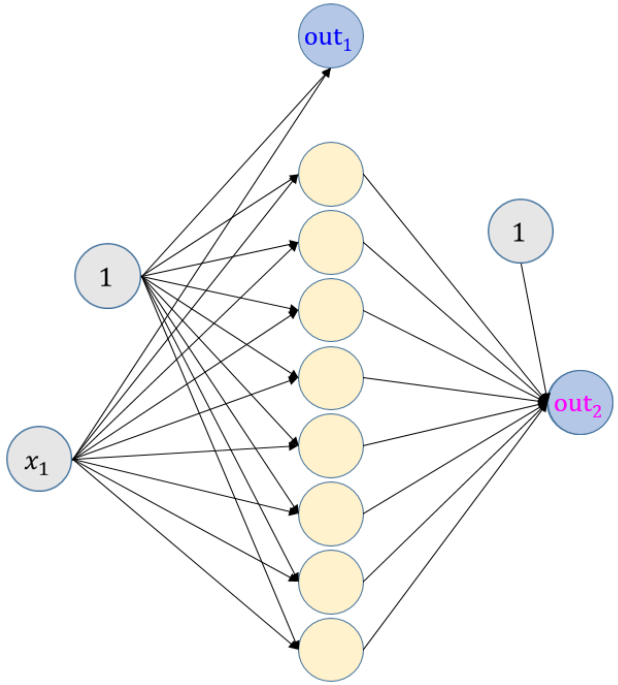
$\hat{w}_a$ $\hat{w}_b$

Minimizing NLL loss $\overset{\sigma \ const}{\Longrightarrow}$ Minimizing MSE loss

Linear regression with flexible non-constant variance

$$Y_{X_i} = (\mathrm{Y}|\mathrm{X}_i) \sim \mathrm{N}(\mu_{x_i}, \sigma_x^2)$$





$$\mu_{x_i} = \mathrm{out}_{1_i}$$

$$\sigma_{x_i} = e^{\mathrm{out}_{2_i}}$$

Minimize the mean negative log-likelihood (NLL) on train data:

$$NLL(w) = \frac{1}{n} \sum_{train-data} -log(p(y_i|x_i, w)$$

gradient descent with NLL loss

$$\hat{\mathbf{w}}_{\mathrm{ML}} = \underset{w}{\mathrm{argmin}} \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma_{x_i}^2}}\right) + \frac{\left(y_i - \mu_{x_i}\right)^2}{2\sigma_{x_i}^2}$$

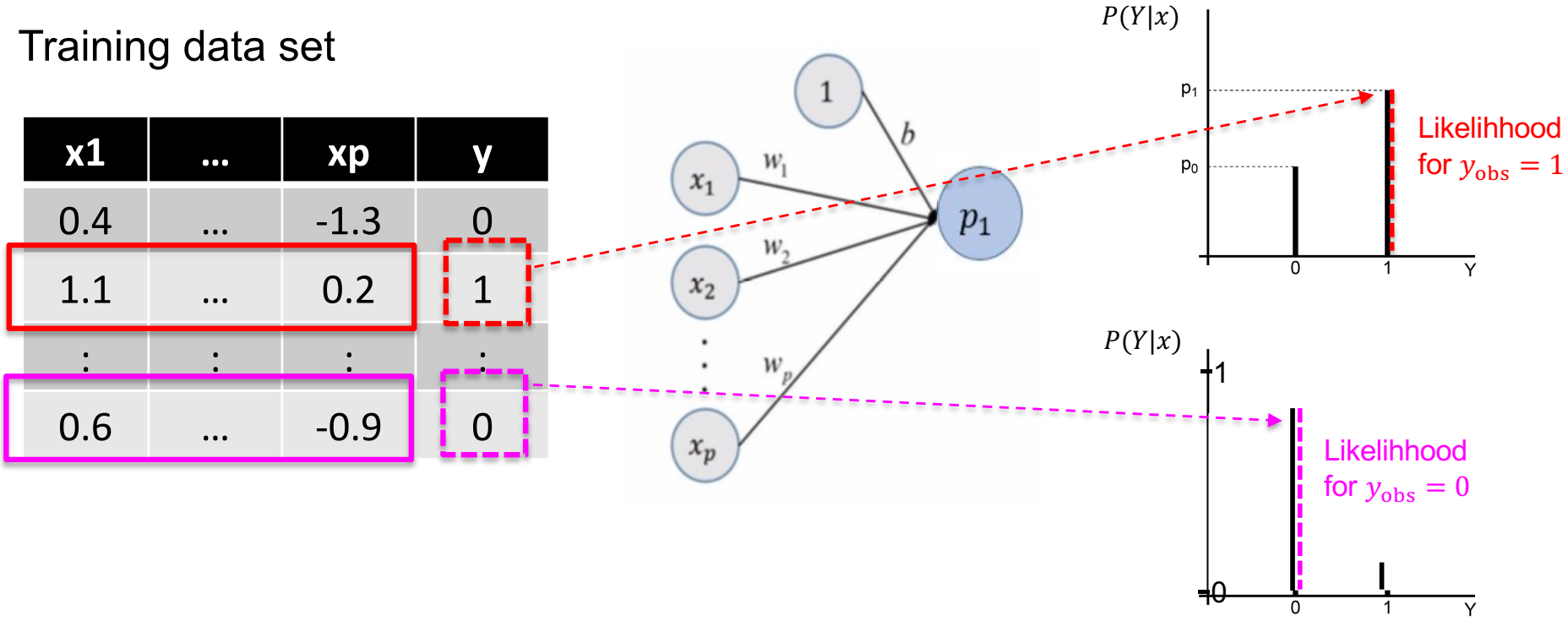
Note: we do not need to know the "ground truth for σ" – the likelihood does the job!

Maximum likelihood principle in logistic regression

probabilistic prediction

Training data set

x1	...	xp	y
0.4	...	-1.3	0
1.1	...	0.2	1
:	:	:	:
0.6	...	-0.9	0



Given (=conditioned on) the input featurs x of an observation i, a well trained NN

- should predict large $p_1 = P(Y = 1|x)$ if the observed class is $y_i = 1$
- should predict small p_1 hence large $p_0 = P(Y = 0|x)$ if observed is $y_i = 0$

→ The likelihood (for the observed outcome) or LogLikelihood should be large

$$\text{LogLikelihood} = \sum_{i=1}^{n} [y_i \log(p_{1i}) + (1 - y_i) \log(1 - p_{1i})]$$

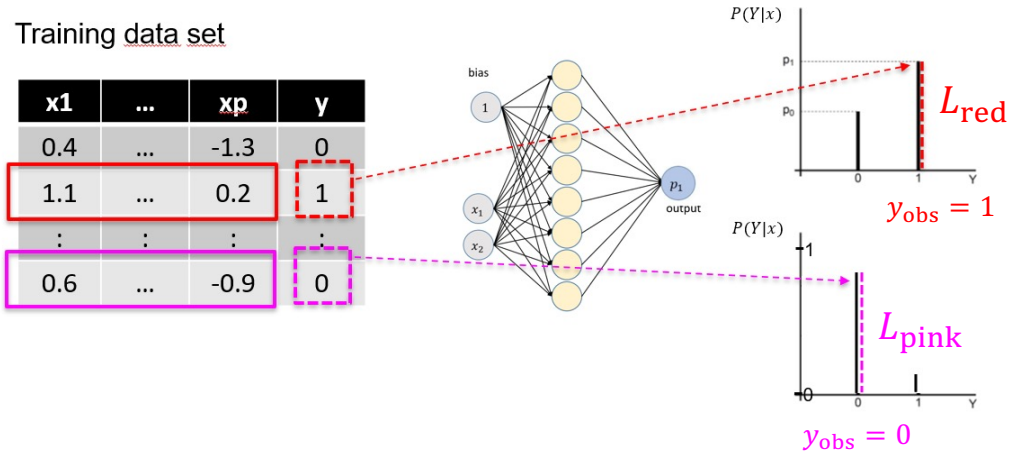
Notation trick in statistics – it selects correct log-probability since $y_i \in \{0,1\}$

Fitting a "logistic regression NN"

The Likelihood of an observation i is the likelihood (probability), that the predicted probability distribution $P(Y|x_i)$ assigns to the observed outcome y_i.

Note: the predicted $P(Y \mid x_i)$ and the corresponding likelihood for the observed y_i depends on the data-point (x_i, y_i) and the model parameter values

→ The higher the likelihood, the better is the model prediction $P(Y|x)$



L_{red} is likelihood of red observation with $y_{obs} = 1$

L_{pink} is likelihood of pink observation with $y_{obs} = 0$

Maximum likelihood principle:

Statistical models are fit to maximize the average LogLikelihood

$$L = \frac{1}{N}\sum L_i = \frac{1}{N}\sum [y_i \log(p_{1i}) + (1 - y_i)\log(1 - p_{1i})]$$

we often use the simplified notation average logLik : $\mathbf{L} = \frac{1}{N}\sum \boldsymbol{log}(\boldsymbol{p_i})$

Predicted probability for the observed outcome y_{obs}

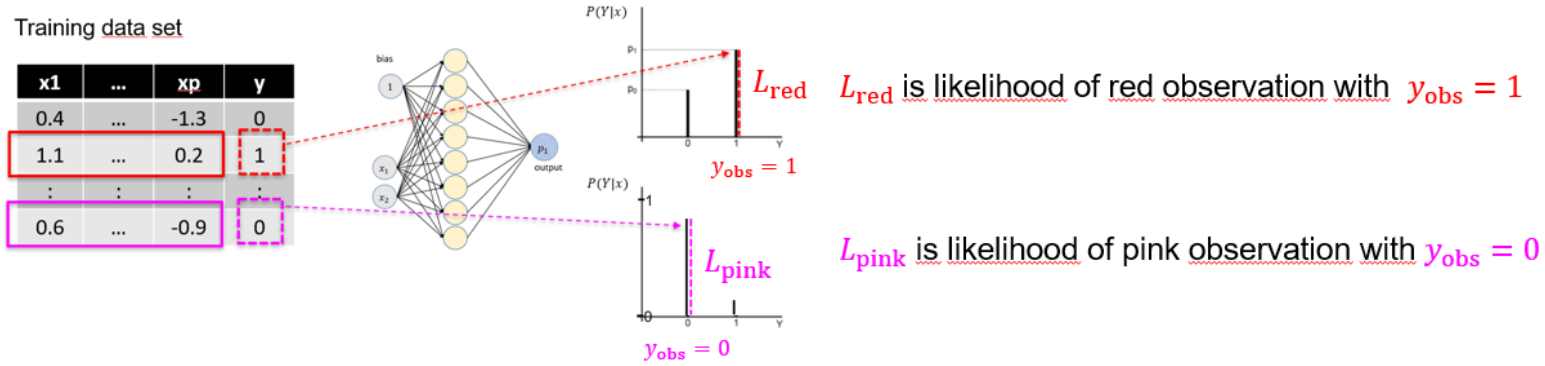
NLL Loss for probabilistic binary classification

In DL we aim to minimize a loss function $L(\text{data}, w)$ which depends on the weights

→ Instead of maximizing the average LogLikelihood

we minimize the averaged Negative LogLikelihood (NLL):

$$\textbf{loss} = \textbf{NLL} = -\frac{1}{N}\sum \log(L_i)$$



The best possible value of the NLL contribution of an observation i is $\log(\mathrm{L_i}) = -\log(1) = 0$

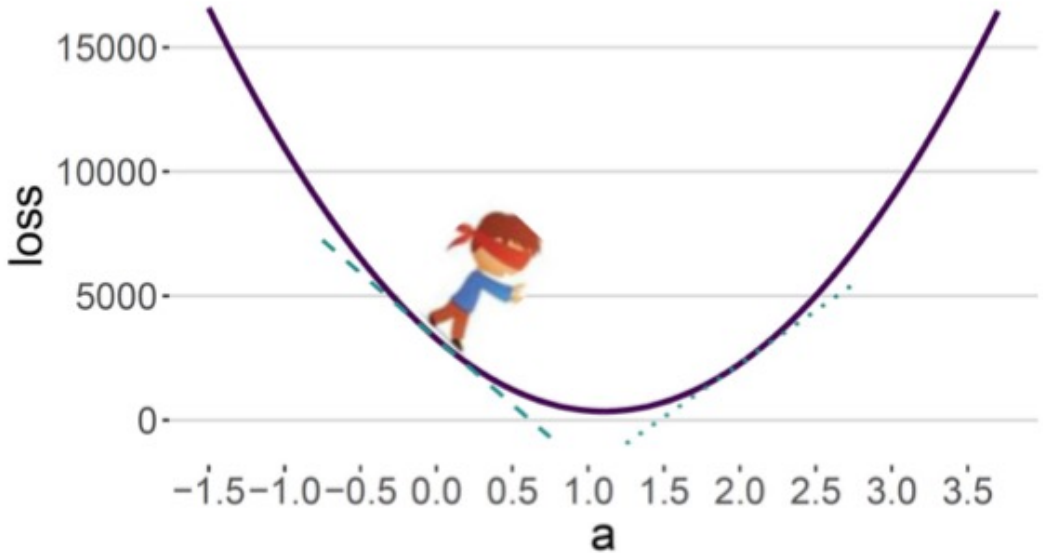
The worst possible value of the NLL contribution of an observation i $\log(\mathrm{L_i}) = -\log(0) = \infty$

Note: In Keras we use the loss 'binary_crossentropy', if we do probabilistc binary classification with one output node with sigmoid activation.

Fit a NN via Stochastic Gradient Descent

Idea of gradient descent

- Shown loss function for a single parameter a

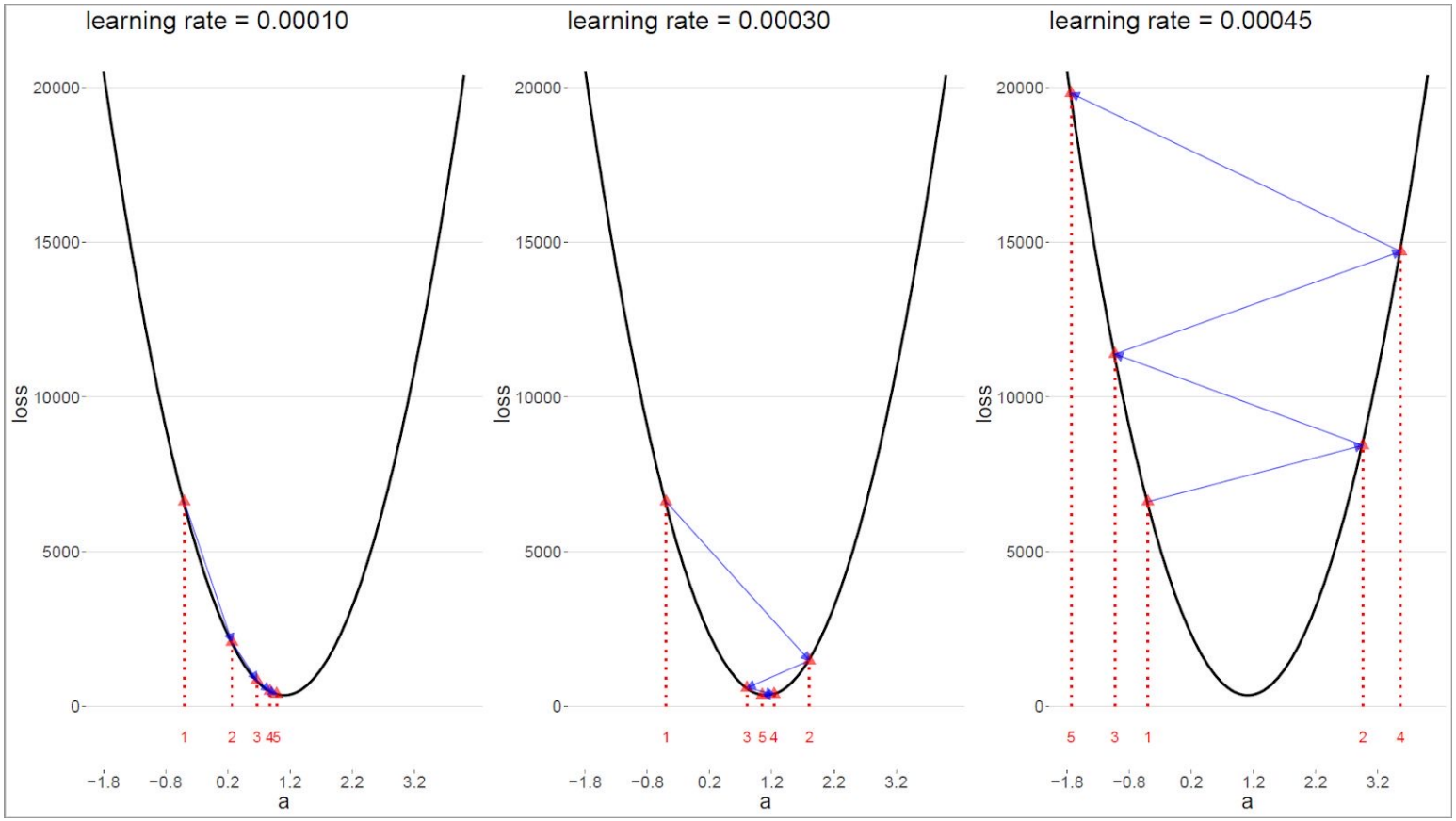


- Take a large step if slope is steep (you are away from minimum)
- Slope of loss function is given by gradient of the loss w.r.t. a
- Iterative update of the parameter a

$$a^{(t)} = a^{(t-1)} - \varepsilon^{(t)} \left. \frac{\partial L(a)}{\partial a} \right|_{a=a^{(t-1)}}$$

learning rate

The learning rate is a very important parameter for DL



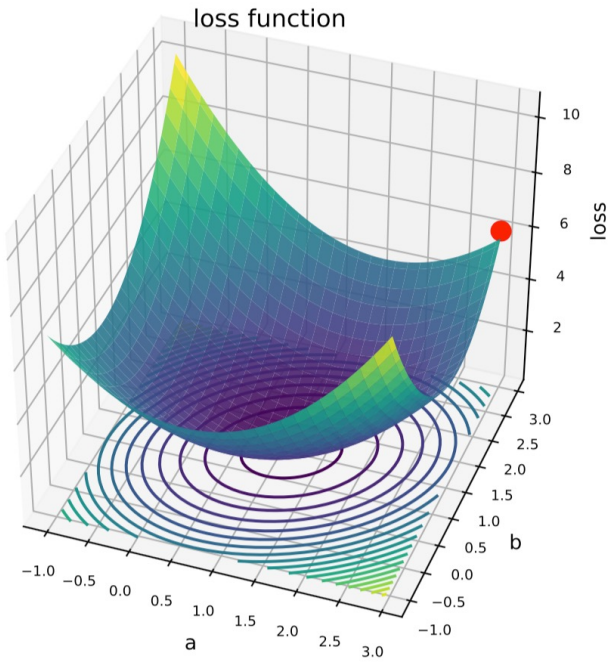
good – little slow

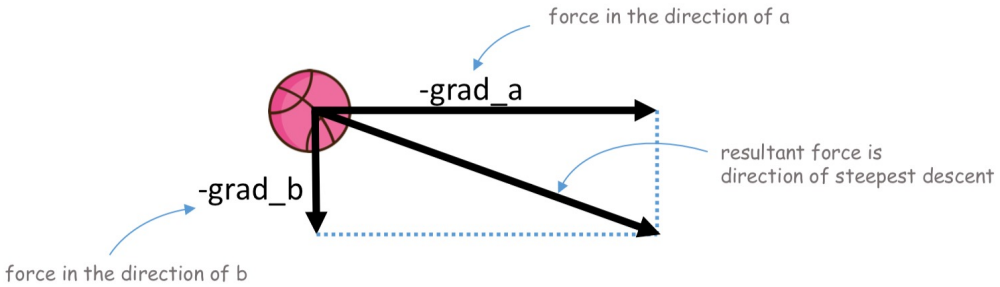
good

learning rate too high

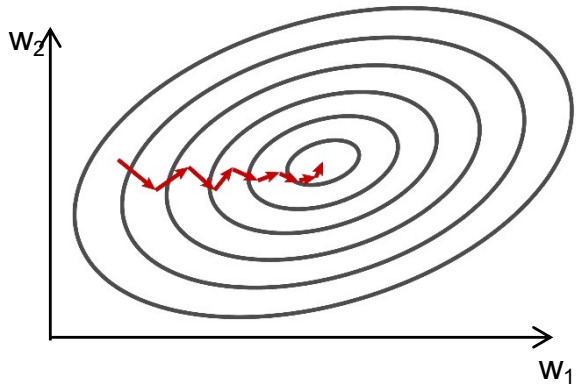
If the loss diverges to infinity: Don't panic, lower the learning rate!

In two dimensions





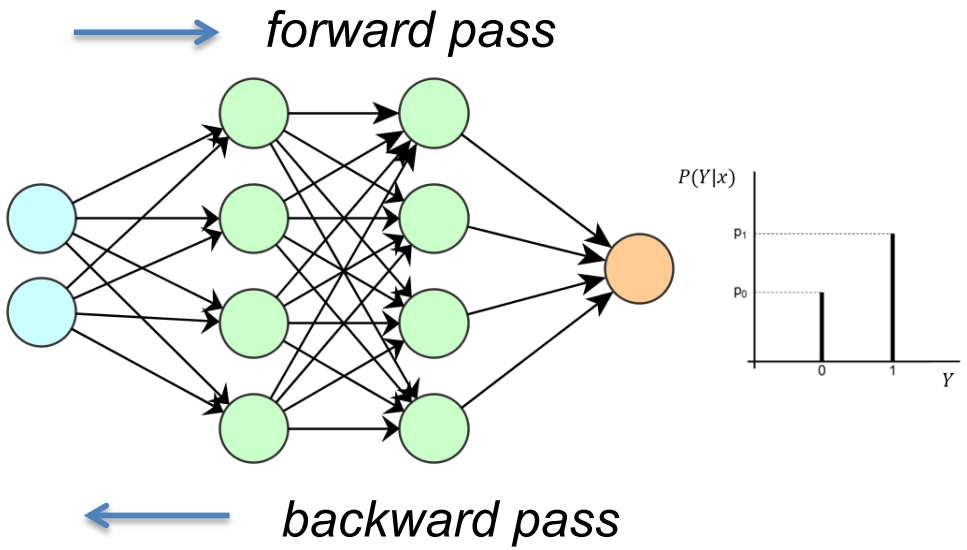
Gradient is perpendicular to contour lines



$$w_i^{(t)} = w_i^{(t-1)} - \varepsilon^{(t)} \left. \frac{\partial L(\mathbf{w})}{\partial w_i} \right|_{w_i = w_i^{(t-1)}}$$

Backpropagation

- We efficiently train the weights in a NN via forward and backward pass
  - Forward Pass propagate training example through network
    - Predicts as output $P(Y|x_i)$ for each input x_i in the batch given the NN weights w
      → With $P(Y|x_i)$ and the observed y_i we compute the loss $L = \left(\text{NLL} = -\frac{1}{N}\sum log(L_i)\right)$
  - Backward pass propagate gradients through network
    - Via chain rule all gradients $\frac{\partial L(\mathbf{w})}{\partial w_k}$ are determined
      → update the weights $w_i^{(t)} = w_i^{(t-1)} - \varepsilon^{(t)} \frac{\partial L(\mathbf{w})}{\partial w_i}\Big|_{w_i = w_i^{(t-1)}}$



For more see lecture 4 (or chapter 3 in our DL book)

The miracle of gradient descent in DL

Loss surface in DL (is not convex) but SGD magically also works for non-convex problems.

Source: chapter 3 in our DL Book

Typical Training Curve / ReLU

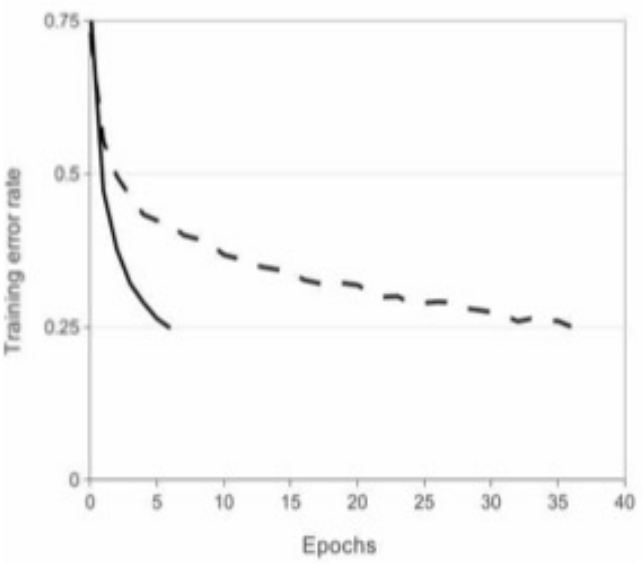
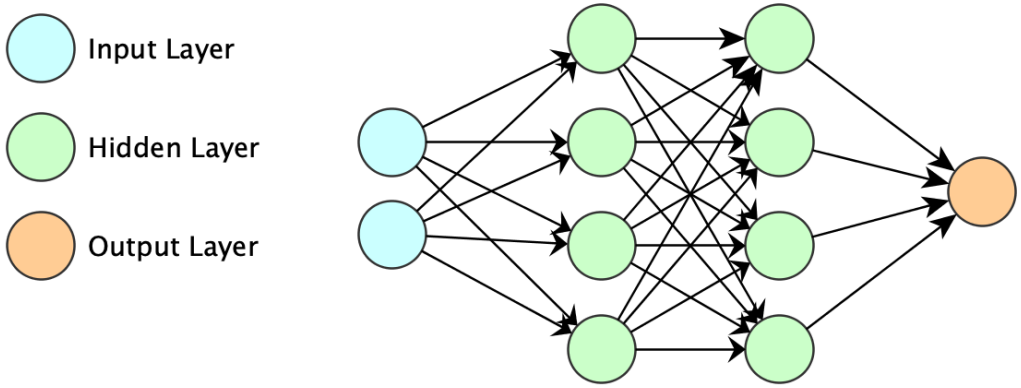
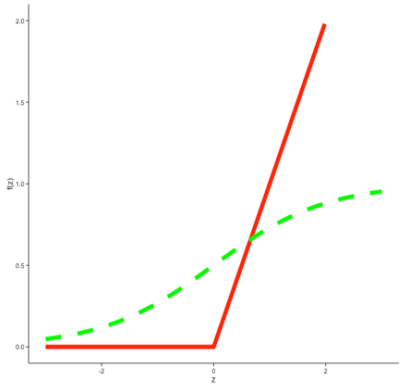


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

Source:
Alexnet
Krizhevsky et al 2012

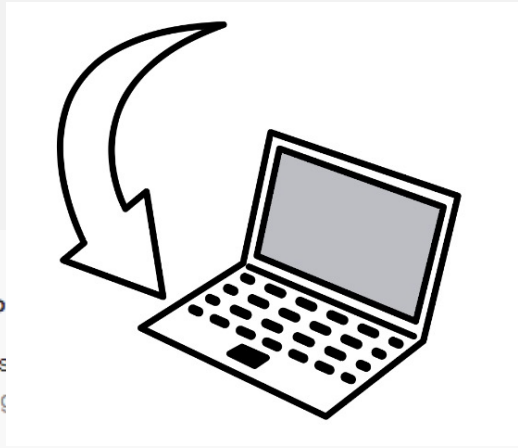


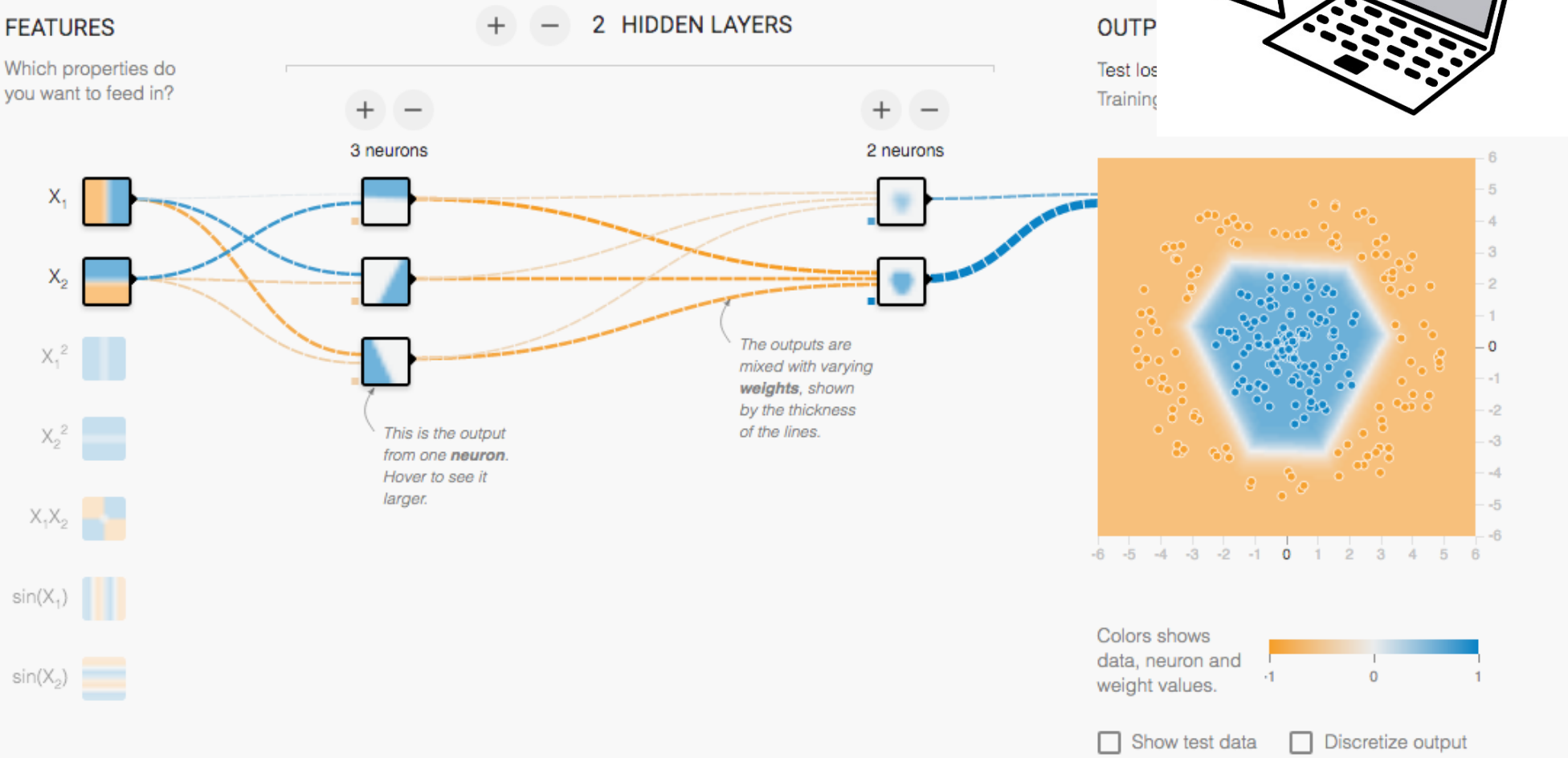


Motivation:
Green:
sigmoid.
Red:
ReLU faster
convergence

Epochs: ”each training examples is used once”

Experiment yourself, play at home





http://playground.tensorflow.org

Let's you explore the effect of hidden layers