

# Machine Intelligence:: Deep Learning Week

*Beate Sick, Lilach Goren, Pascal Bühler*

Institut für Datenanalyse und Prozessdesign  
Zürcher Hochschule für Angewandte Wissenschaften

# Outline of the DL Module (tentative)

The course is split in 8 sessions, each 4 lectures long. Topics might be adapted during the course

Day	Date	Time	Topic
1	15.04.2025	09:00-12:30	Introduction to Deep Learning & Keras, first NNs
-	21.04.2025	-	FRÜHLINGS-FERIEN
-	28.04.2025	-	FRÜHLINGS-FERIEN
2	06.05.2025	09:00-12:30	Loss, Optimization, Regression, Classification
3	13.05.2025	09:00-12:30	Computer vision, CNN-architecture
4	20.05.2025	09:00-12:30	DL in practice, pretrained (foundation) models
5	27.05.2025	09:00-12:30	Model evaluation, baselines, xAI, troubleshooting
6	03.06.2025	09:00-12:30	Generative Models, Transformer-architecture
7	10.06.2025	09:00-12:30	Vision Transformer
8	17.06.2025	09:00-12:30	Projects, deep Ensembling

# Supervised DL for Classification or Regression tasks

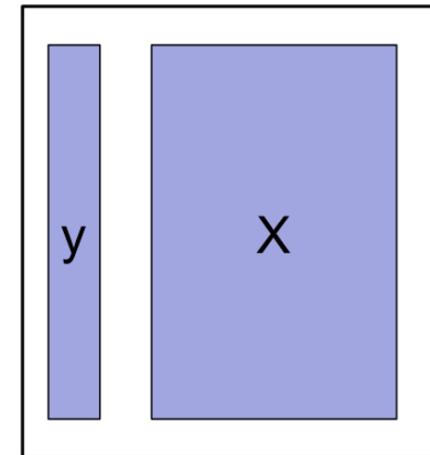
- Typical supervised DL task: predict  $y$  given  $x$

- **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



Supervised Learning

# Probabilistic vs deterministic models

Deterministic

“Classification”



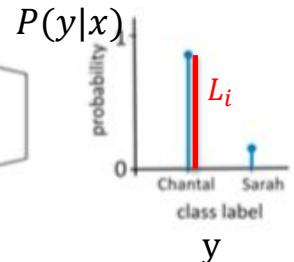
“Regression”



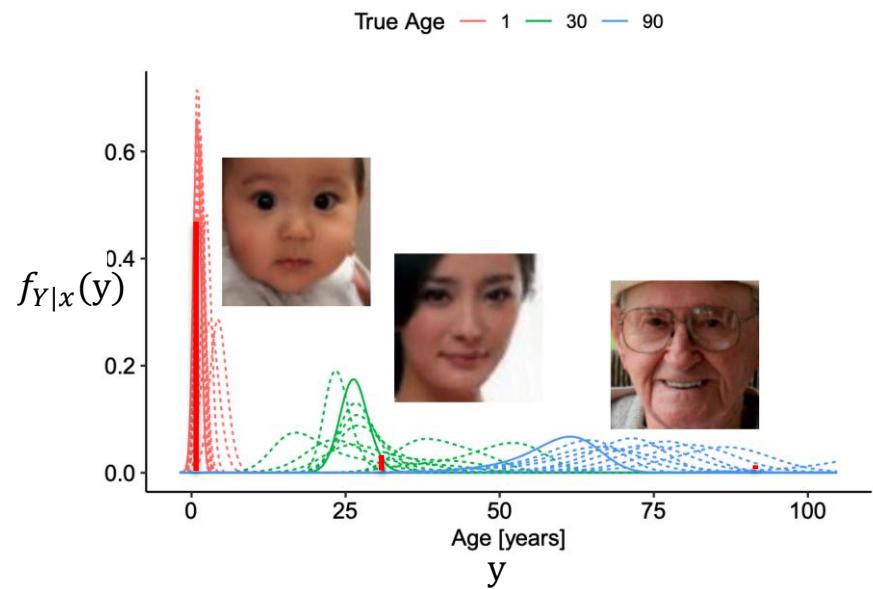
Probabilistic



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

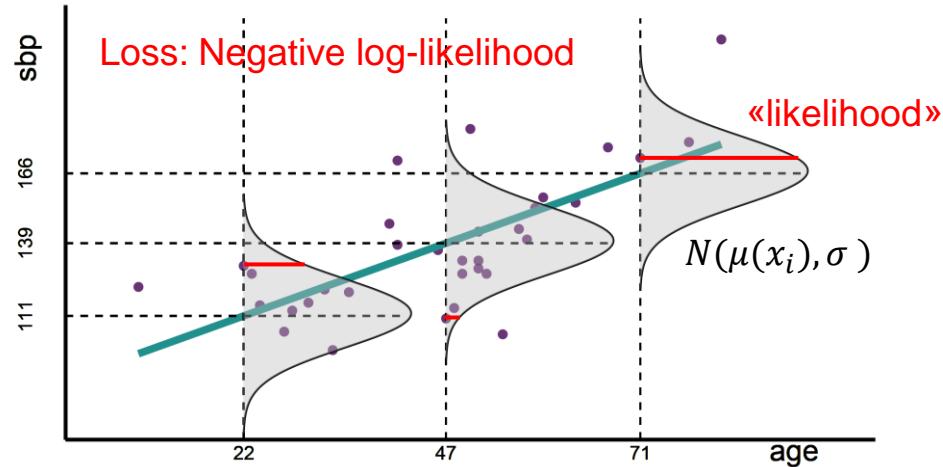
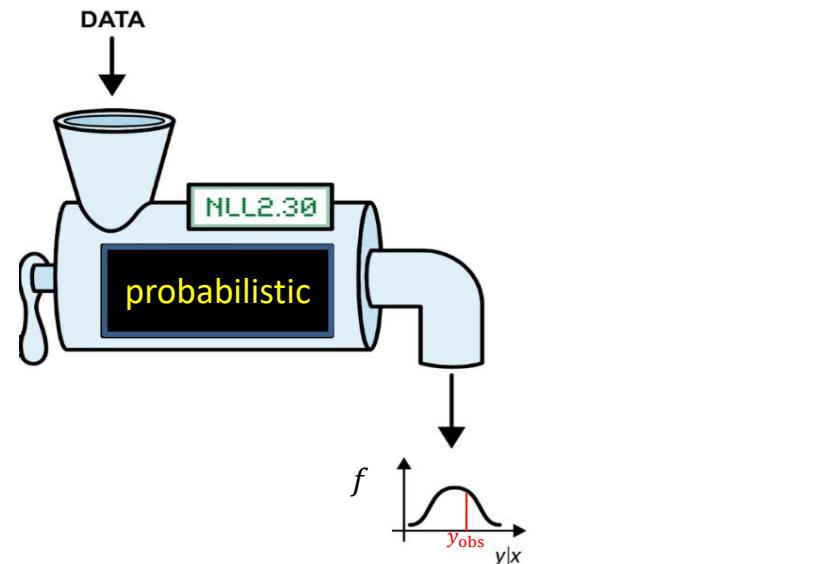
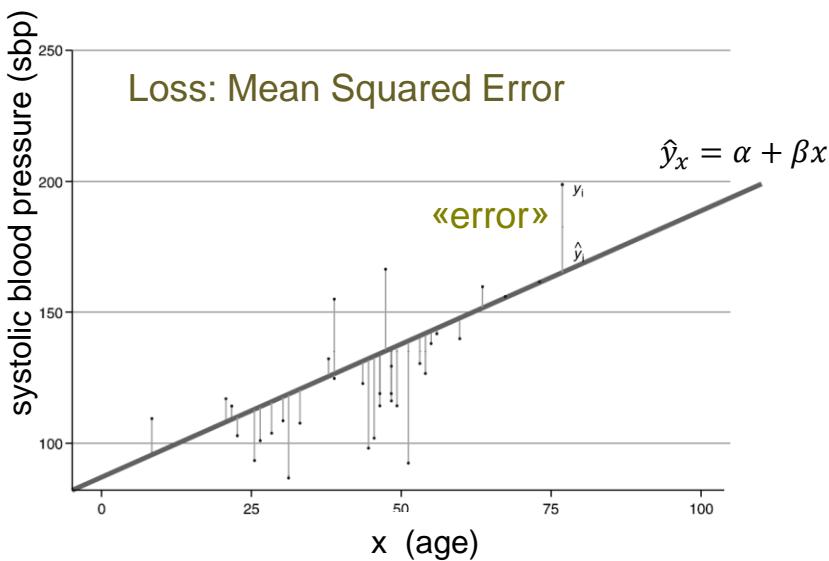
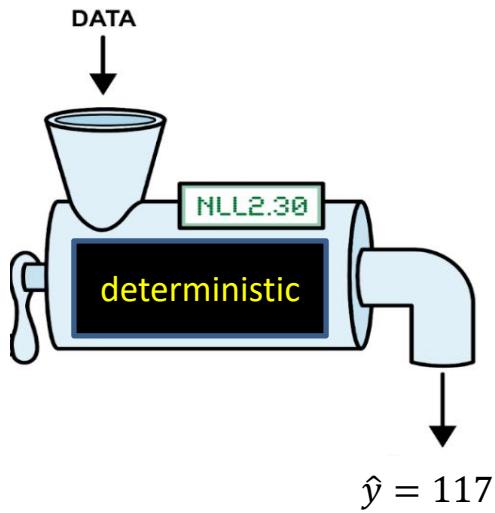


74



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

# Non-probabilistic versus probabilistic regression DL models

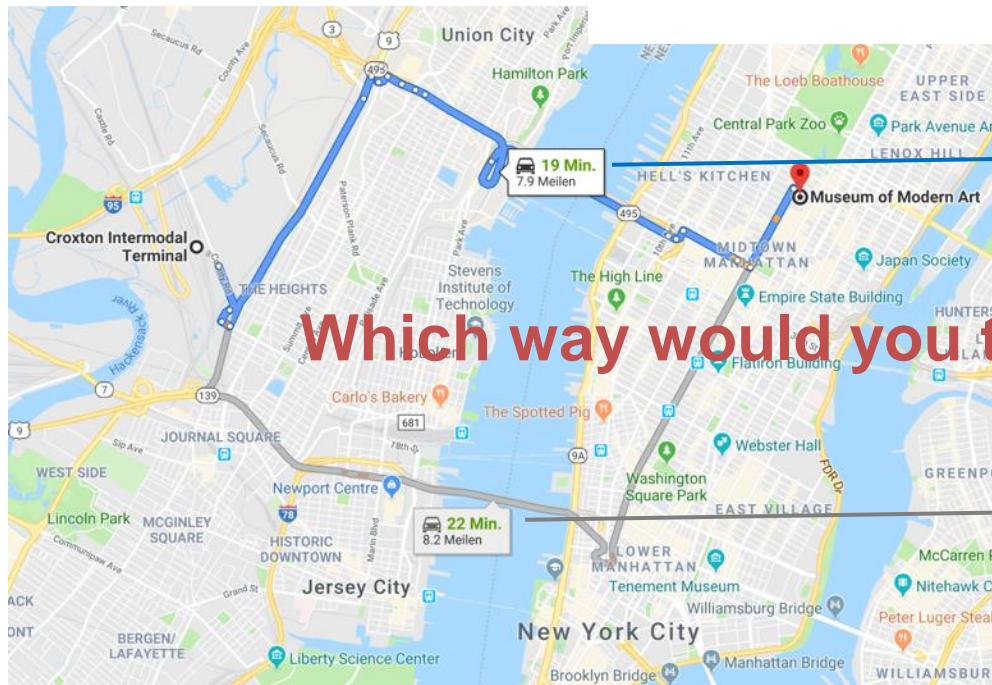


# Probabilistic models in everyday tasks

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

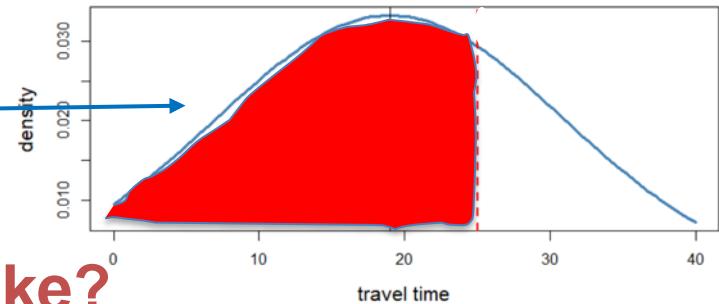


Let's use my probabilistic travel time gadget!

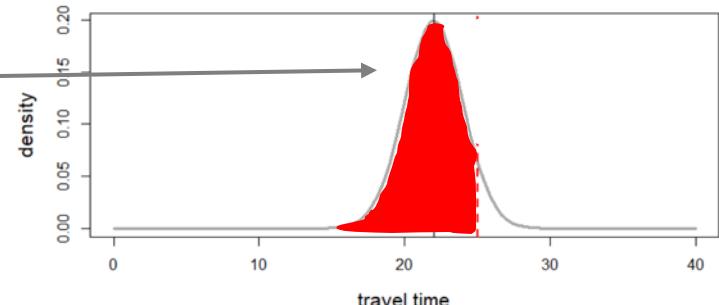


Which way would you take?

Chance to get tip: 69%



Chance to get tip: 93%

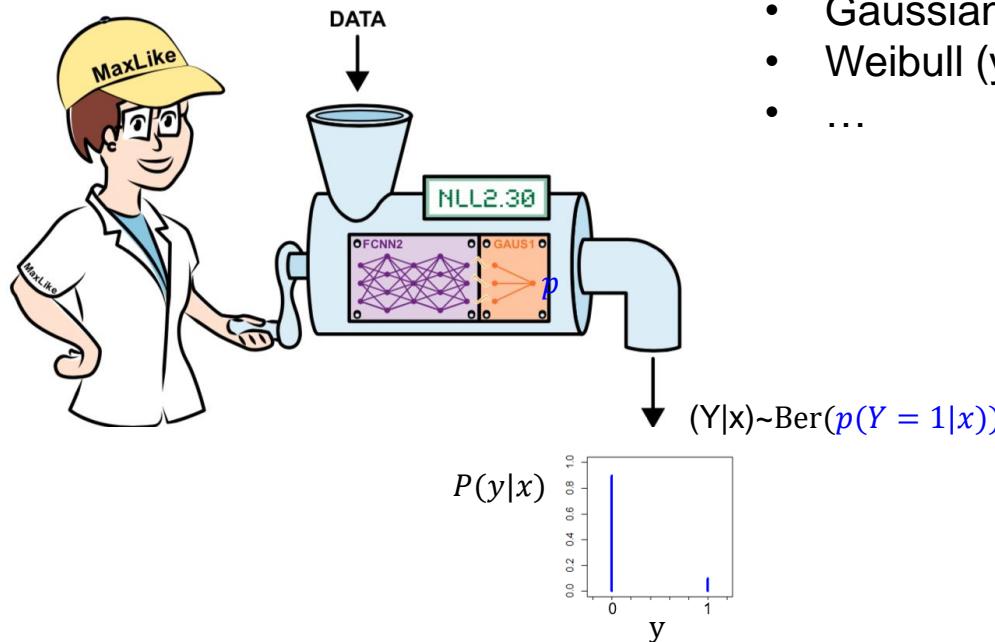
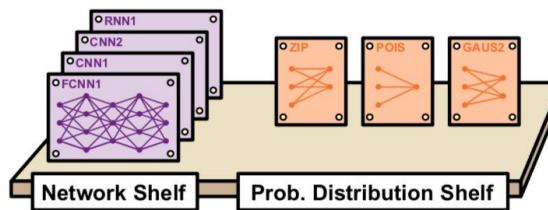


# Guiding Theme of the course

- 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:

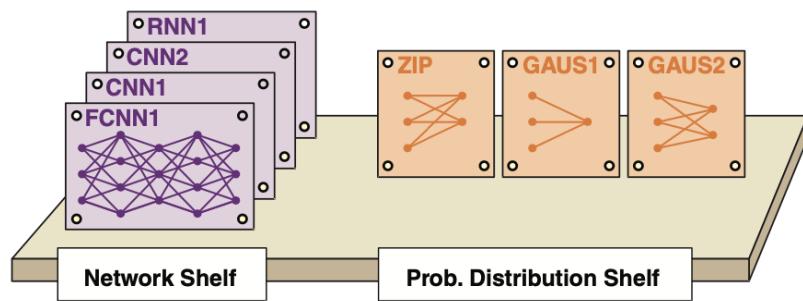
- Tabular data  $\rightarrow$  fcNN, tabPFN
- Images  $\rightarrow$  CNN, ViT
- Text  $\rightarrow$  LLMs



Family of outcome distribution, parameters are controlled by NN:

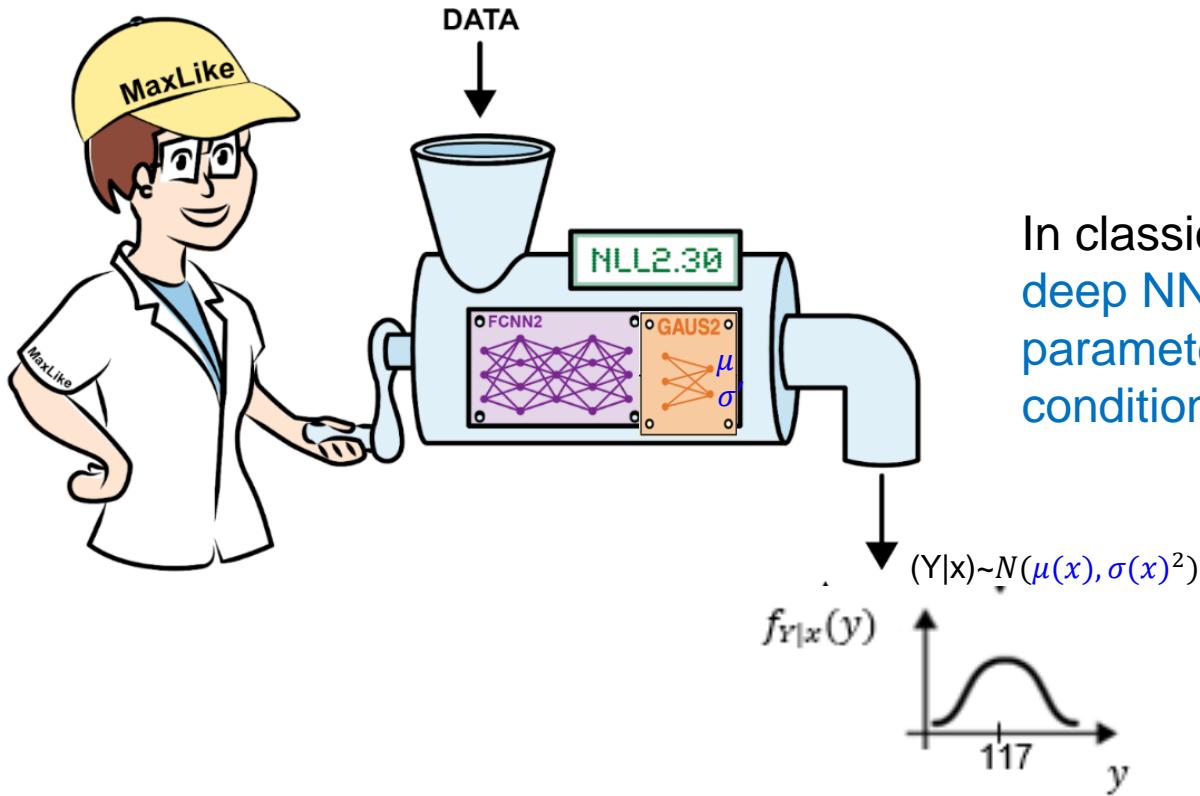
- Bernoulli ( $y$ : binary)
- Multi-Nomial ( $y$ : classes)
- Poisson ( $y$ : count)
- Negative Binomial ( $y$ : count)
- Gaussian ( $y$ : real number)
- Weibull ( $y$ : real number)
- ...

# 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.

# How to design a NN to do Logistic Regression?

Note: this is a special case of binary classification

# Recall logistic regression from statistics

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

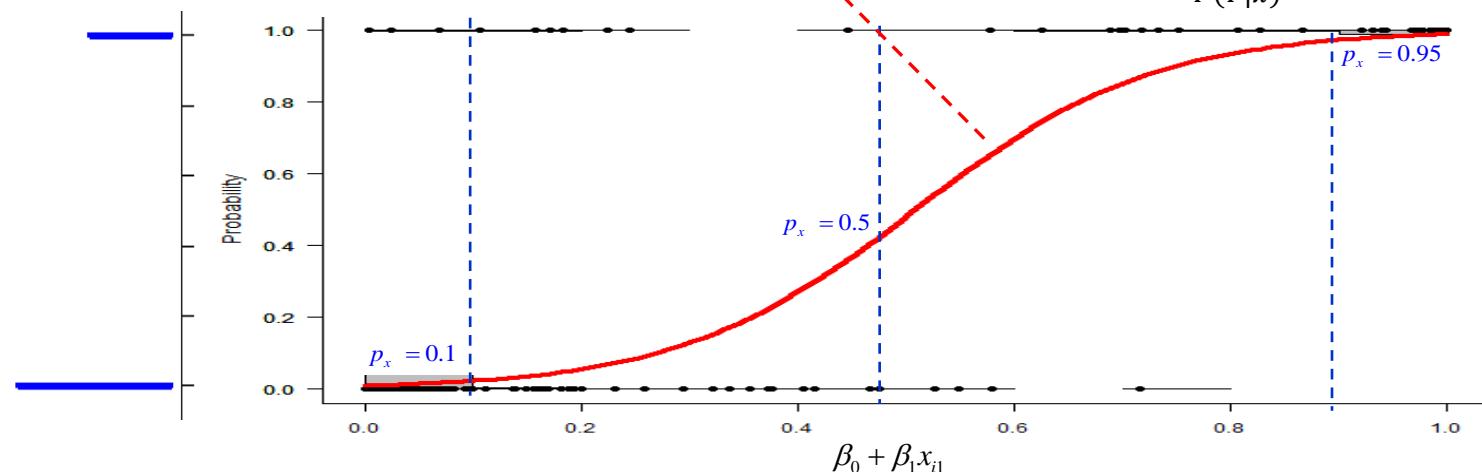
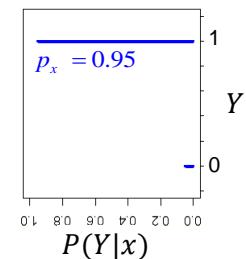
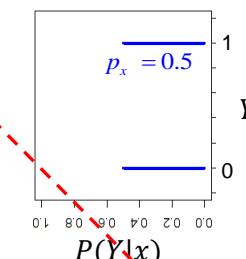
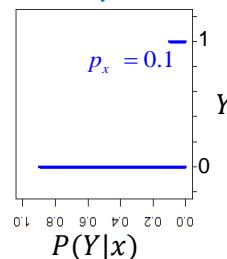
$$(Y|X_i) \sim \text{Ber}(p_{x_i})$$

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

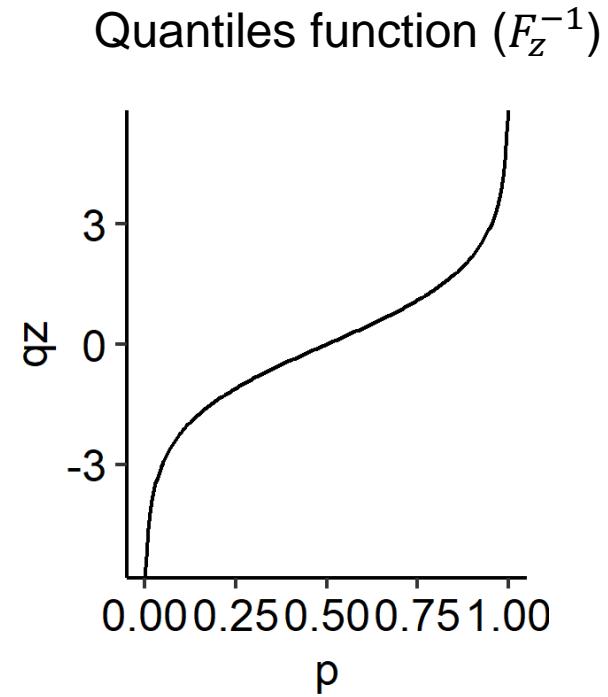
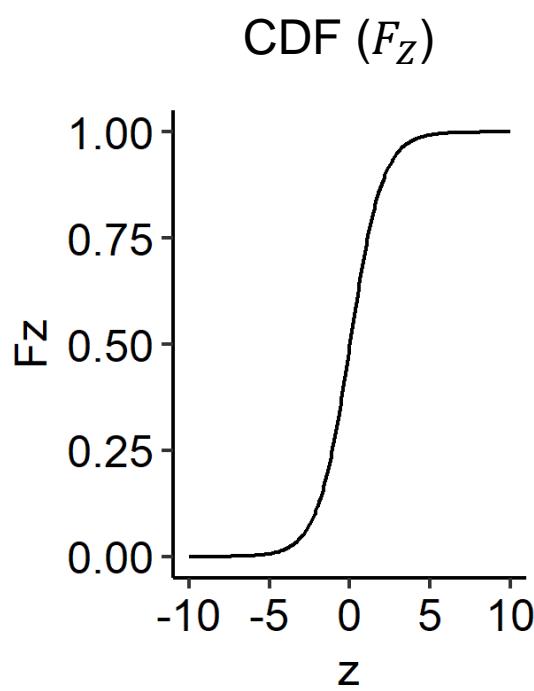
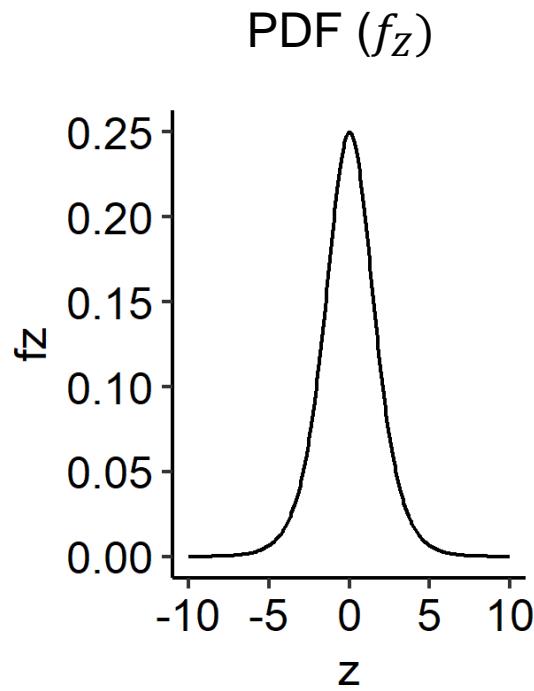
$$P(Y=1|x_i) = p_{x_i} = \sigma(\beta_0 + \beta_1 x_{i1}) = \frac{1}{1+e^{-(\beta_0 + \beta_1 x_{i1})}}$$

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

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



# Recall the Standard Logistic Distribution



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

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

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

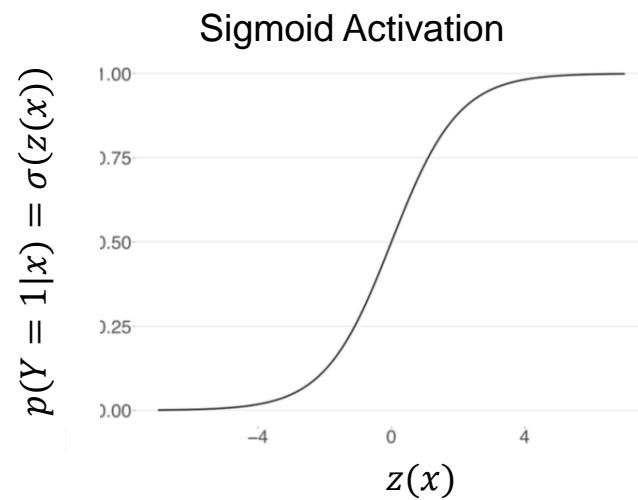
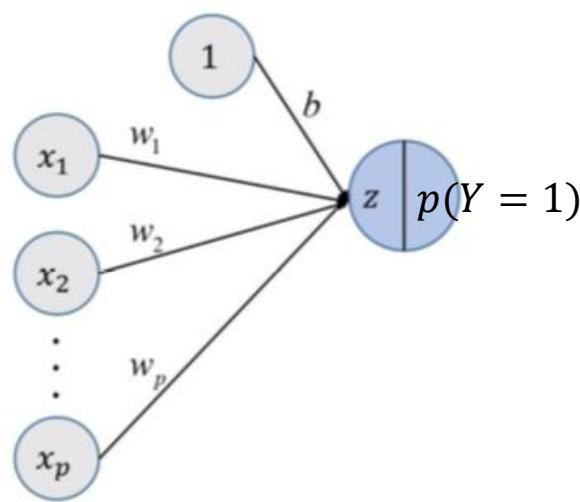
aka “sigmoid”

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

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

# Logistic regression as NN

## Activation in the last layer: “sigmoid”



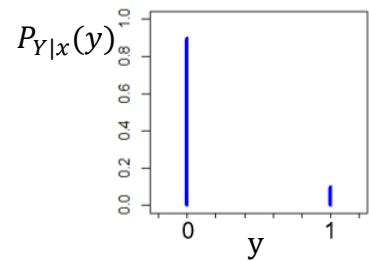
$$z(x) = b + x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_p \cdot w_p$$

$$p(Y = 1|x) = \sigma(z) = \sigma(\beta_0 + \beta_1 x_1 + \dots + \beta_p x_p) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \dots + \beta_p x_p)}}$$

The output after the sigmoid activation:

$$P(y_i = 1|x_i) = \sigma(\beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip})$$

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



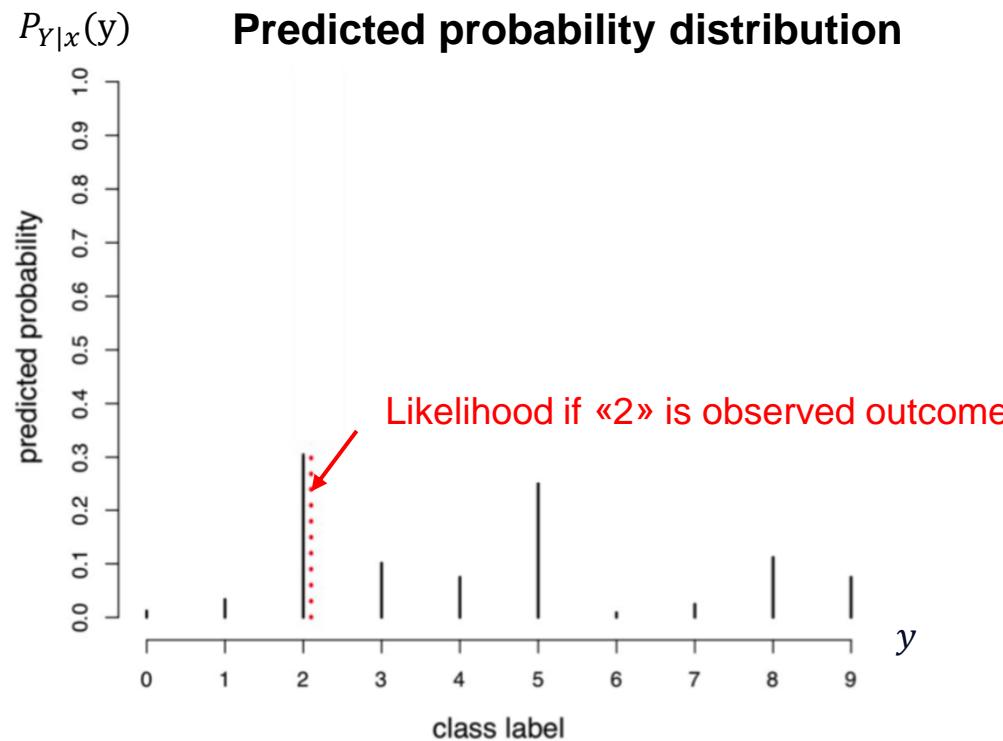
# How to design a NN to do Multinomial Regression?

Note: this is a special case of multi-class classification

# Multinomial regression aka multi-class regression

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

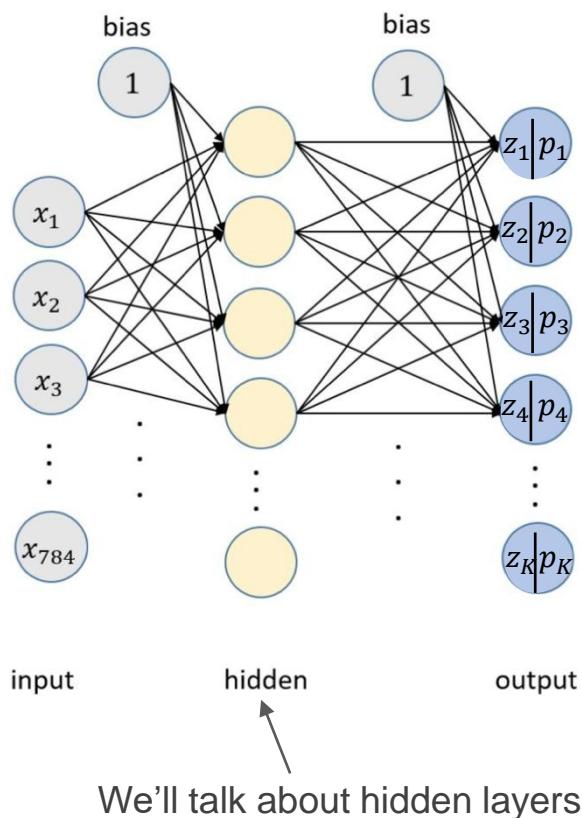
$$Y \in \{c1, c2, \dots, cK\}, P_{Y|x}(Y = c_k) = P(Y = c_k|x) = p_k(x), \sum_{k=1}^K p_k(x) = 1$$



Remark: If  $Y$  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”



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

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

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

Makes outcome positive

Ensures that  $p_i$ '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

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

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

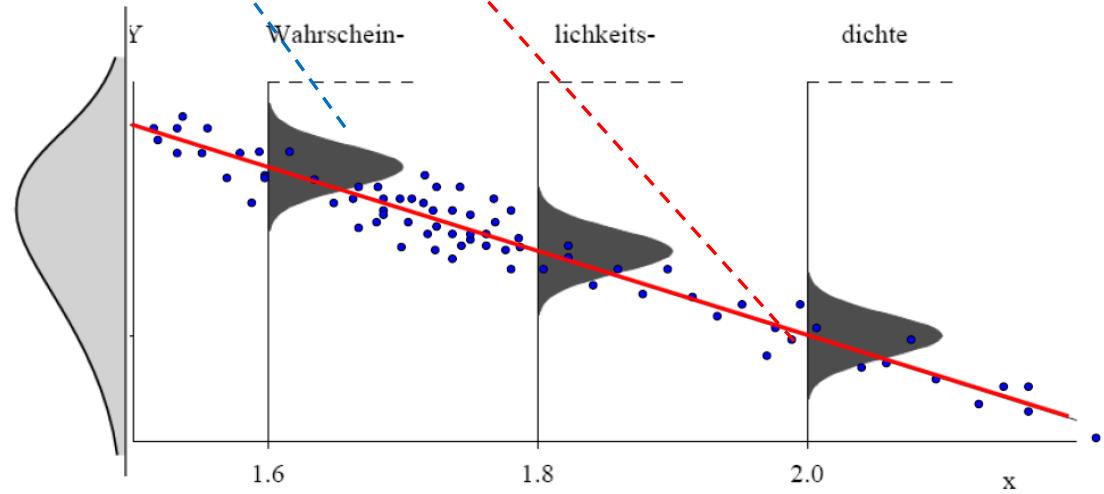
Probabilistic linear regression predicts for each input  $x_i$  a Gaussian conditional probability distribution for the output  $P(Y|x_i)$  that assigns each possible value of  $Y$  a likelihood.

point prediction  
probabilistic prediction

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

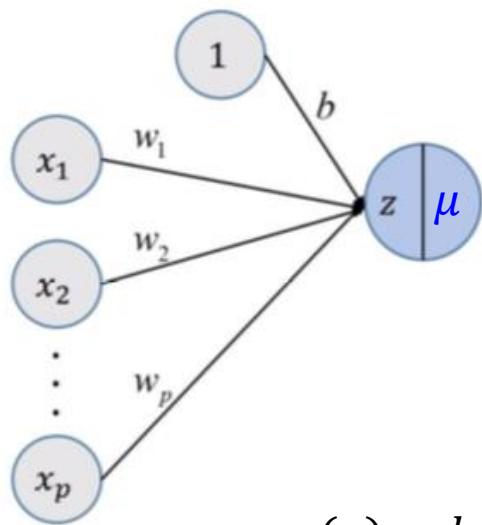
Variance is assumed to be constant

$Y$  can have an arbitrary distribution.



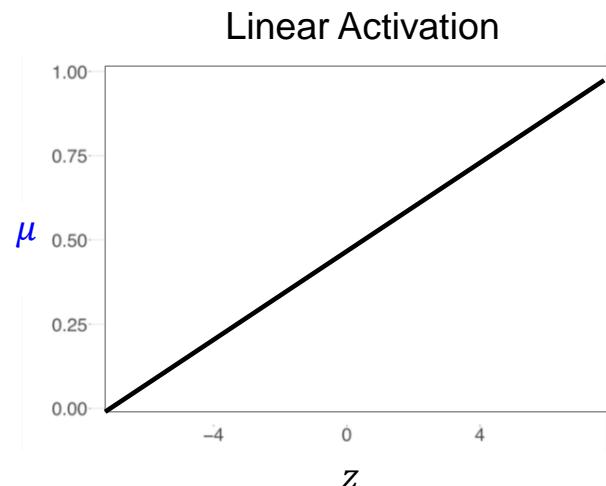
# Linear Regression as NN

## Activation in the last layer: “linear”



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

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

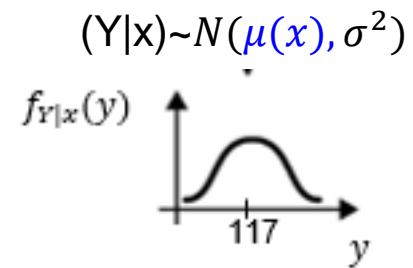


The output after the linear activation can be interpreted as estimated expected value of the Gauss

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

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

Remark: after fitting the linear predictor  $\hat{\mu}(x)$  the constant variance is estimated from the residuals



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

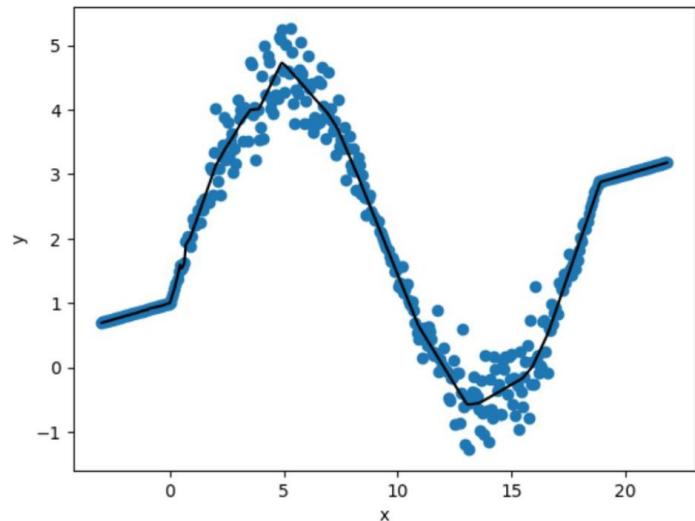
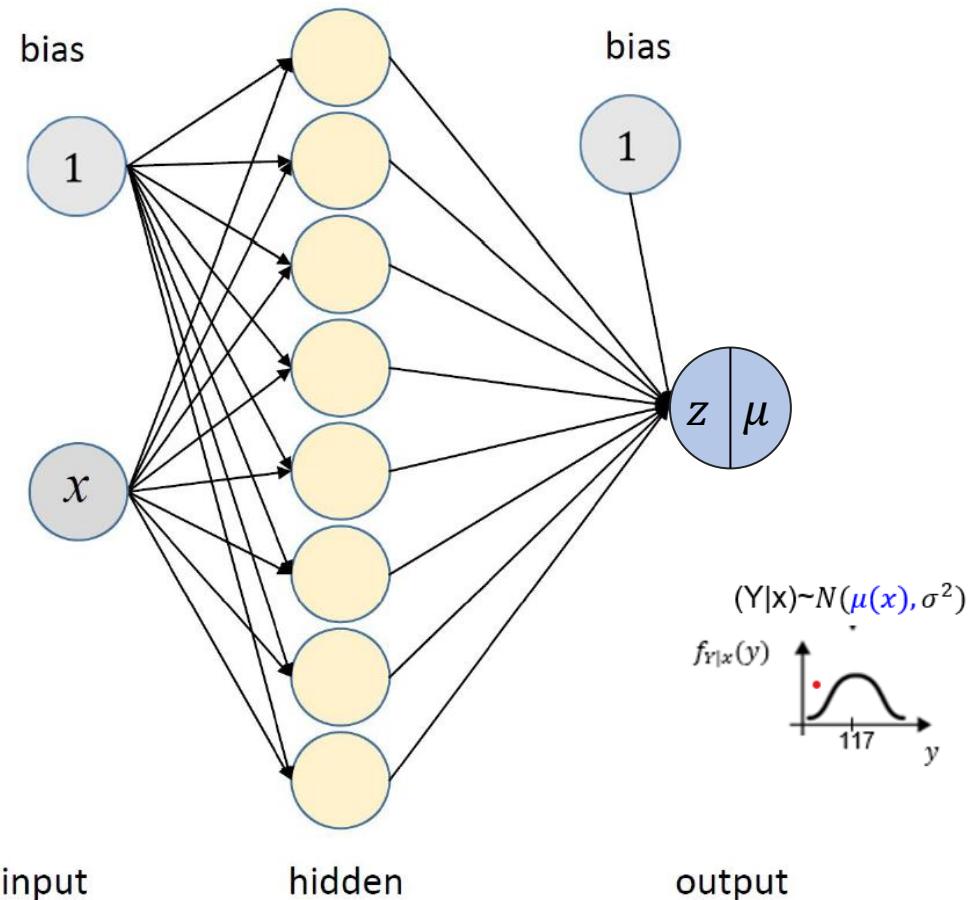
A close-up shot from the movie Inception. Two men in dark suits are facing each other. The man on the left, played by Leonardo DiCaprio, has his eyes closed and a slight smile. The man on the right is partially visible, also smiling. The lighting is warm and dramatic.

**WE NEED TO GO**

**DEEPER**

# Non-linear Regression as NN

## Activation in hidden layer non-linear, in last layer linear

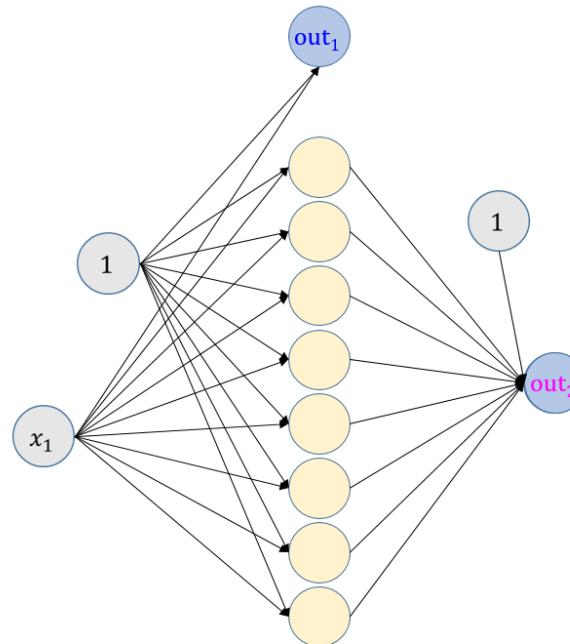
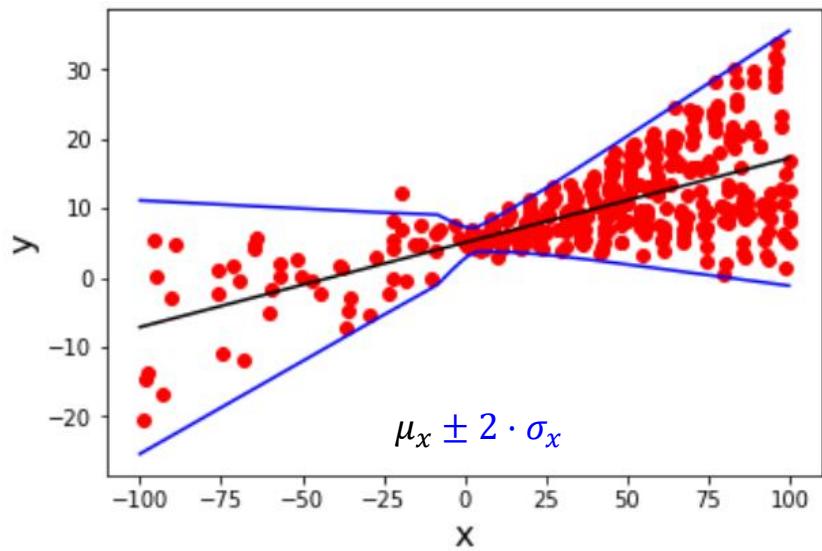


Here, the conditional outcome distribution is a Gaussian with a  $\mu$  that depends non-linearly on  $x$  😊 a variance  $\sigma^2$  that is constant ⚡

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

# NN for linear regression with not-constant variance

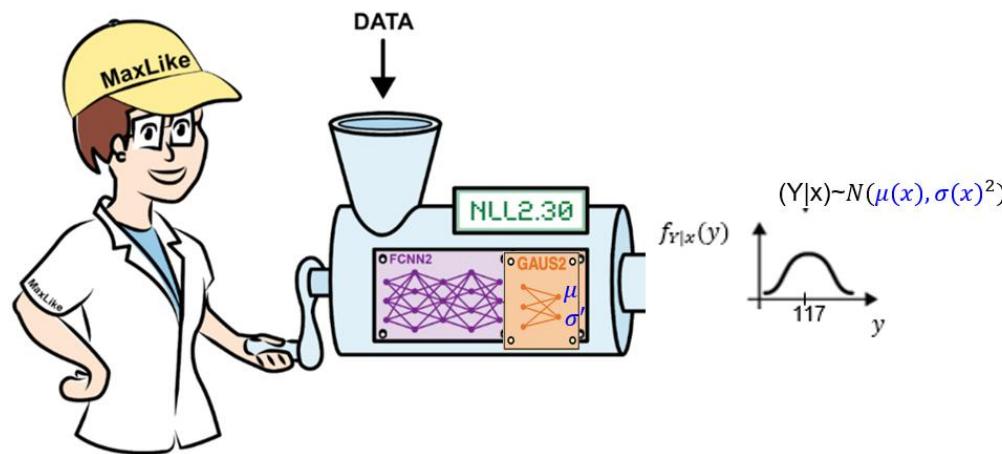
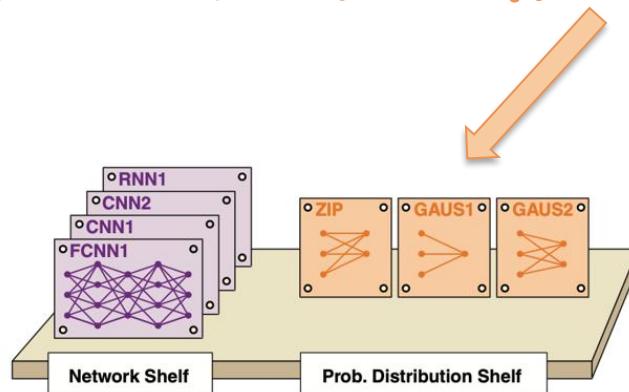
$$Y_{X_i} = (\mathbf{Y} | \mathbf{X}_i) \sim N(\mu_{x_i}, \sigma_x^2)$$



$$\mu_{x_i} = out_{1_i}$$

$$\sigma_{x_i} = e^{out_{2_i}}$$

NLL is used as loss function which depends on outcome distribution

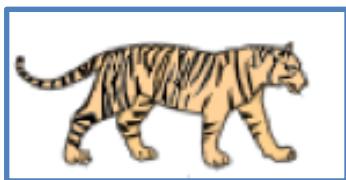


# Training: Tune the weights to minimize the loss

Input  $x^{(i)}$

True class  $y^{(i)}$

predicted class  
(class with highest predicted probability)



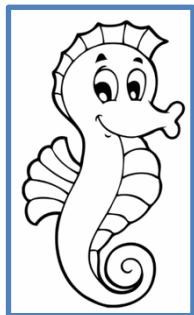
Tiger

→ Lion 🤢



Tiger

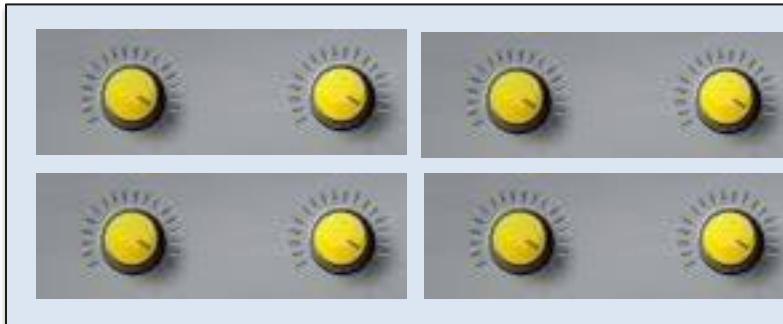
→ Tiger 🤗



Seehorse

→ Seehorse 🤗

Neural network with many weights  $w$



Trainingsprinciple:

Weights are tuned so a loss functions gets minimized.

...

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

# Which loss to use in probabilistic DL?

## Negative Log Likelihood!

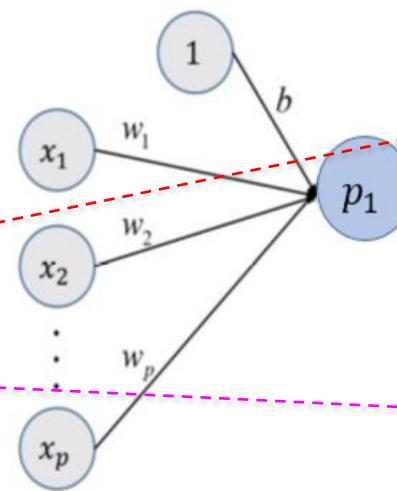


# NLL-loss for binary classification

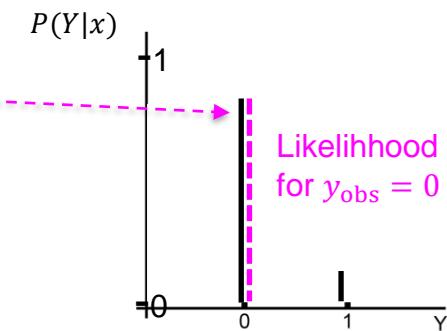
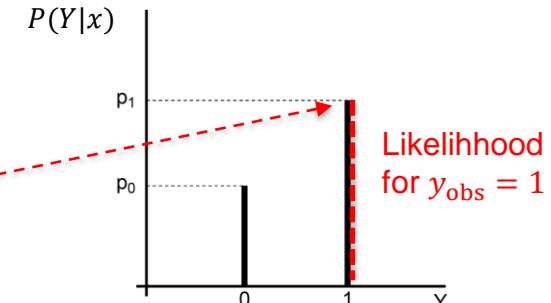
# Maximum likelihood principle in logistic regression

Training data set

<b>x1</b>	...	<b>xp</b>	<b>y</b>
0.4	...	-1.3	0
1.1	...	0.2	1
:	:	:	:
0.6	...	-0.9	0



probabilistic prediction



Given (=conditioned on) the input features  $x$  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”

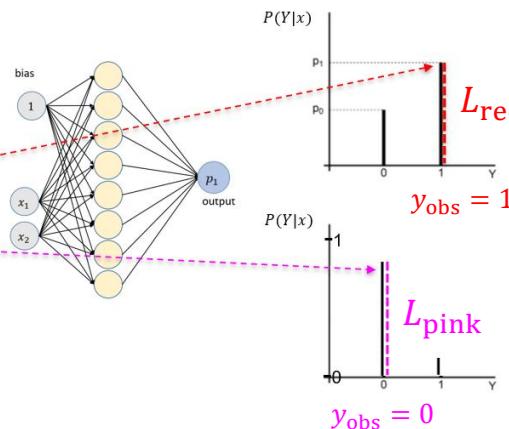
The Likelihood of an observation  $i$  is the likelihood (probability), that the predicted probability distribution  $P(Y|x_i)$  assigns to the observed outcome  $y_i$ .

Note: the predicted  $P(Y|x_i)$  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  $P(Y|x)$

Training data set

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



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

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

## Maximum likelihood principle:

Statistical models are fit to maximize the average LogLikelihood

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

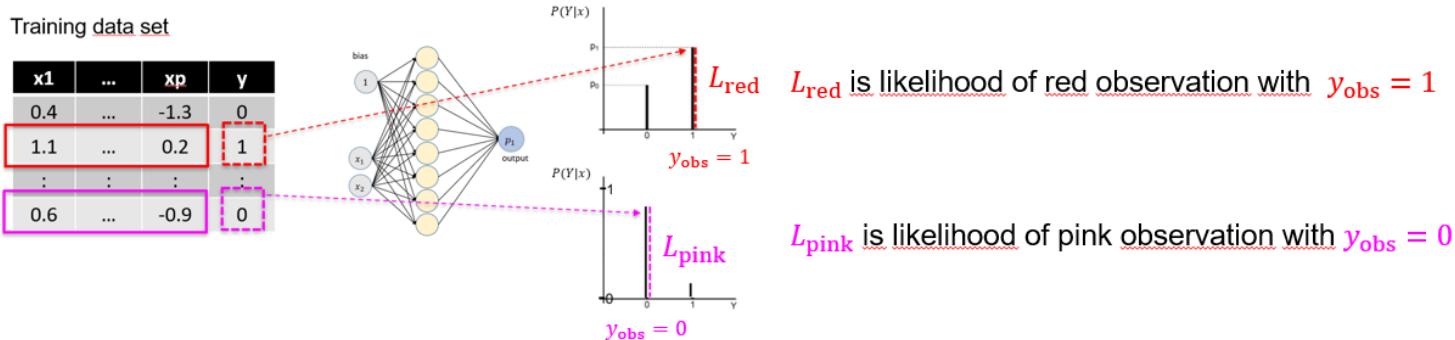
we often use the simplified notation average logLik :  $L = \frac{1}{N} \sum \log(p_i)$

Predicted probability for  
the observed outcome  $y_{\text{obs}}$

# 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):

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



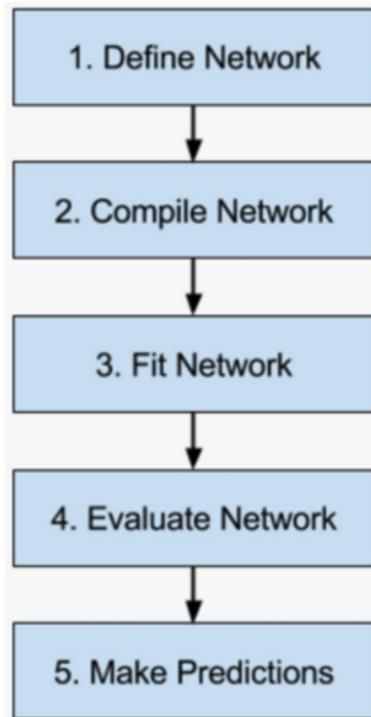
The best possible value of the NLL contribution of an observation  $i$  is  $-\log(L_i) = -\log(1) = 0$

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

Note: In Keras the NLL for binary outcome is called 'binary\_crossentropy',  
if we do probabilistic binary classification with 1 output node with sigmoid activation.  
If we would use 2 output nodes and softmax than the NLL is called 'categorical\_crossentropy'

# Define the network

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



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

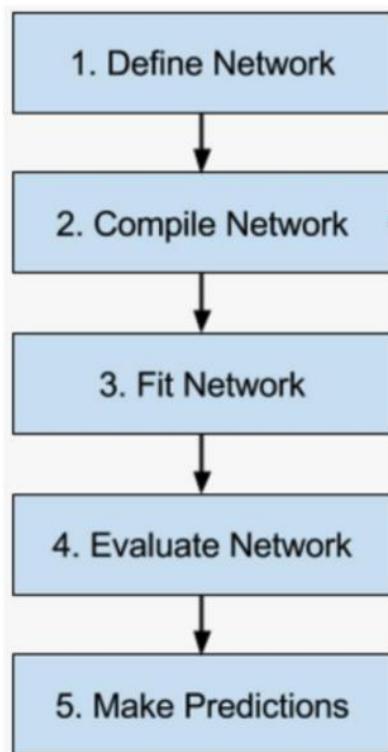
Hidden layer with 8 Neurons,  
Activation function: sigmoid

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



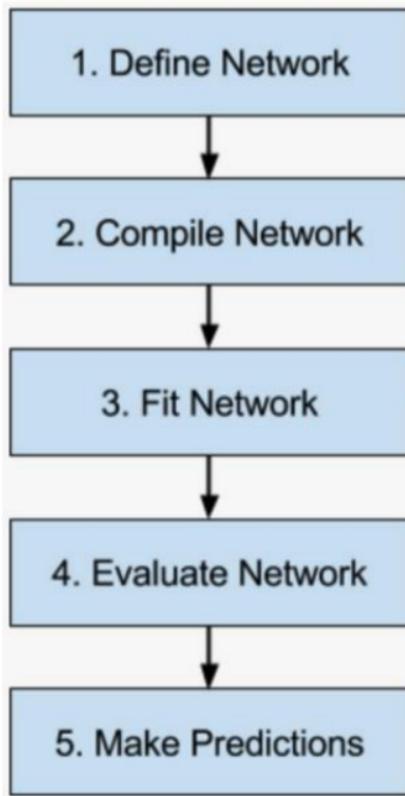
```
model.compile(optimizer=SGD(learning_rate=0.01),  
              loss='binary_crossentropy',  
              metrics=['accuracy'])
```

Loss Function to optimize  
Here: Binary CE

Which optimizer should be used?  
Here Stochastic Gradient Descent

Which metrics do we want to track,  
Here: Accuracy

# Fit the network



Data Tensor X for training

Label Tensor Y

Validation at  
End of each epoch

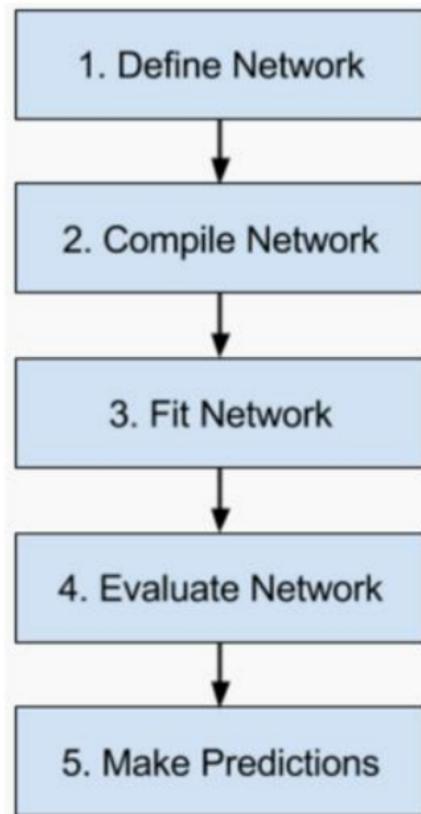
```
history = model.fit(x_train,  
                     y_train,  
                     validation_data=(x_val, y_val),  
                     epochs=1000,  
                     batch_size=10,  
                     verbose=1)
```

How many time do  
we feed the whole  
dataset to the  
model

Should we see the whole  
output? If no verbose = 0

How many samples per  
batch, 1 batch = 1 iteration  
of weight updates

# Evaluate the network



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

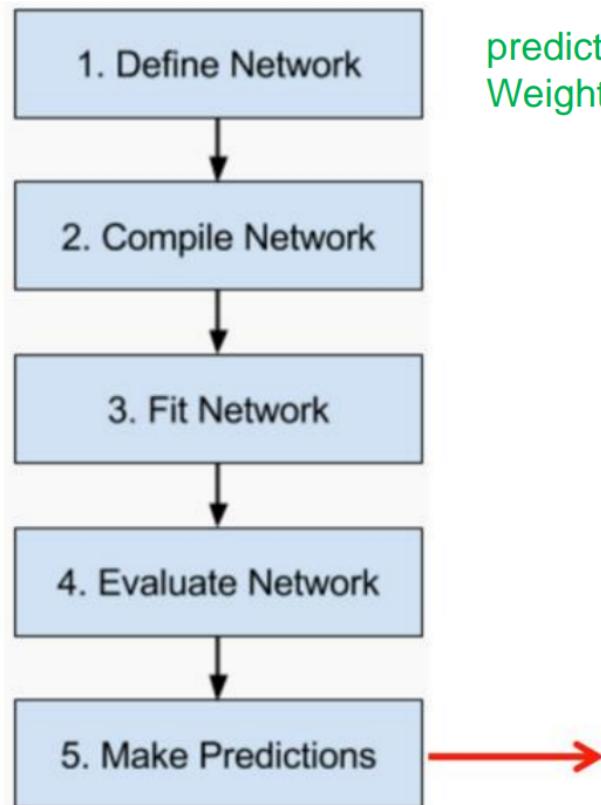
```
model.evaluate(x_test, y_test, verbose=0)
```

Test Loss: 0.33591118454933167, Test Accuracy: 0.8581818342208862

Remember from the  
.compile

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

# Make Predictions



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

# Losses in Keras

The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.

## Available losses

Note that all losses are available both via a class handle and via a function handle. The class handles enable you to pass configuration arguments to the constructor (e.g. `loss_fn = CategoricalCrossentropy(from_logits=True)`), and they perform reduction by default when used in a standalone way (see details below).

▶ [Keras 3 API documentation / Losses / Probabilistic losses](#)

### Probabilistic losses

- `BinaryCrossentropy class`
- `BinaryFocalCrossentropy class`
- `CategoricalCrossentropy class`
- `CategoricalFocalCrossentropy class`
- `SparseCategoricalCrossentropy class`
- `Poisson class`
- `CTC class`
- `KLDivergence class`
- `binary_crossentropy function`
- `categorical_crossentropy function`
- `sparse_categorical_crossentropy function`
- `poisson function`
- `ctc function`
- `kl_divergence function`

### Regression losses

- `MeanSquaredError class`
- `MeanAbsoluteError class`
- `MeanAbsolutePercentageError class`
- `MeanSquaredLogarithmicError class`
- `CosineSimilarity class`

▶ [Keras 3 API documentation / Losses / Regression losses](#)

## Probabilistic losses

### BinaryCrossentropy class

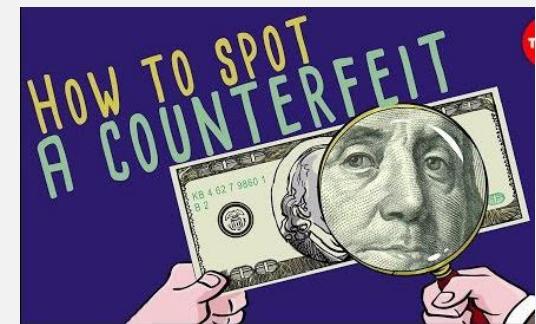
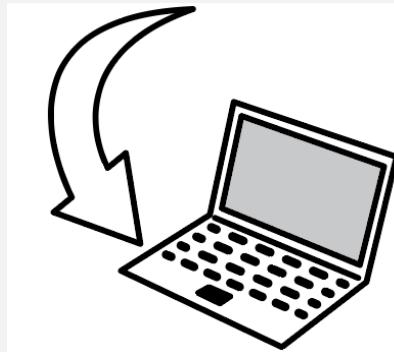
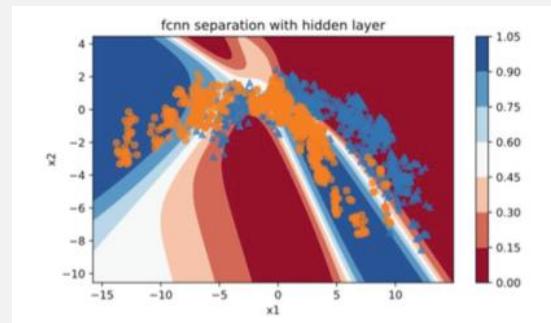
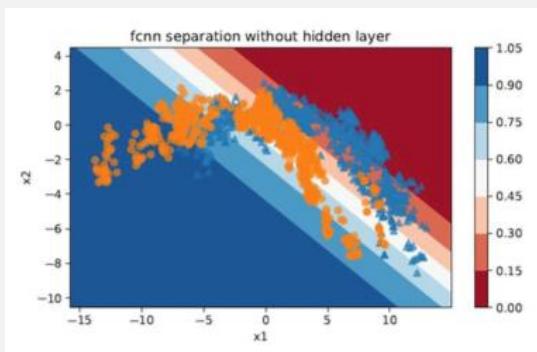
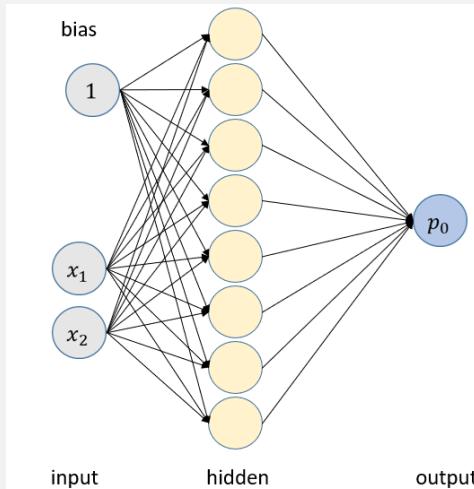
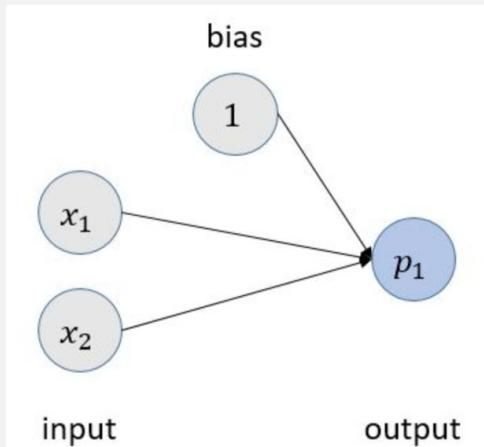
```
keras.losses.BinaryCrossentropy(  
    from_logits=False,  
    label_smoothing=0.0,  
    axis=-1,  
    reduction="sum_over_batch_size",  
    name="binary_crossentropy",  
    dtype=None,  
)
```

## Regression losses

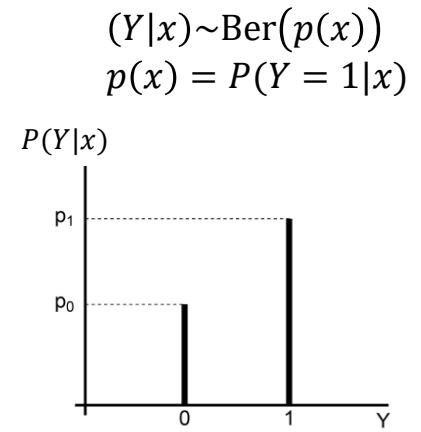
### MeanSquaredError class

```
keras.losses.MeanSquaredError(  
    reduction="sum_over_batch_size", name="mean_squared_error", dtype=None  
)
```

# Exercise: Keras for banknote classification

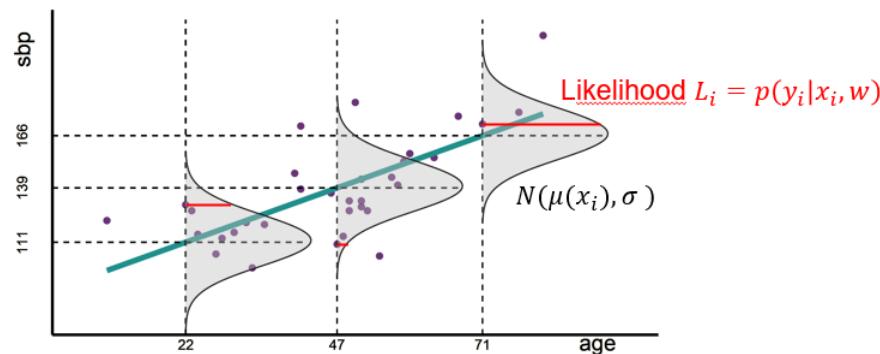


Open NB [02\\_fcnn\\_with\\_banknote\\_keras\\_torch.ipynb](#)  
and use Keras to train a NN with and w/o hidden layer



# NLL-loss for regression with constant variance

# 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  $\rightarrow$  multiplication of likelihood over all data points:

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

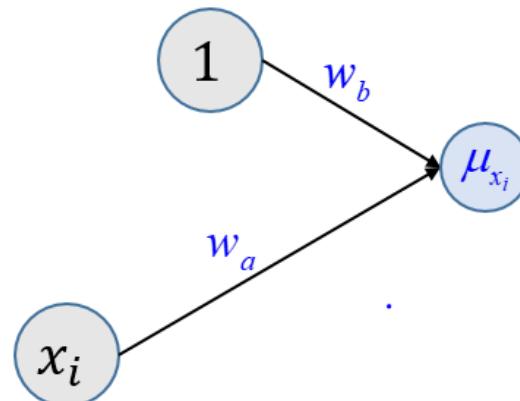
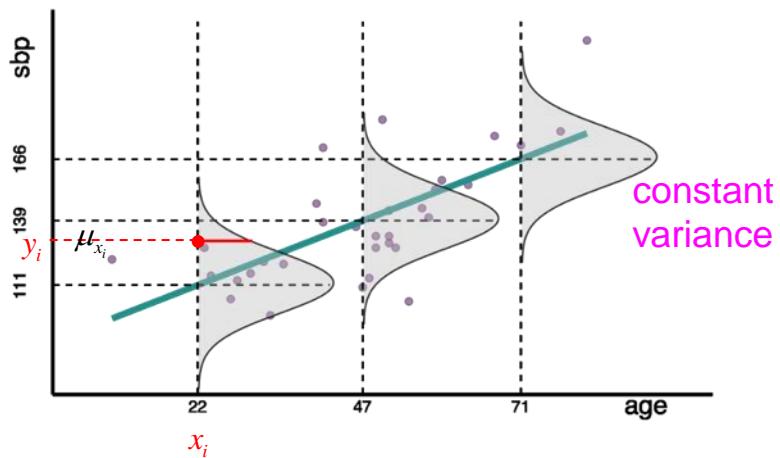
Practically: Use Negative Log-Likelihood NLL as loss and minimize NLL

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

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

# Fit “linear regression NN” via Maximum likelihood principle

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



ML-principle:

$$\begin{aligned} \mathbf{w}_{\text{ML}} &= \underset{w}{\operatorname{argmax}} \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \mu_{x_i})^2}{2\sigma^2}} \\ &= \underset{w}{\operatorname{argmin}} \sum_{i=1}^n -\log \left( \frac{1}{\sqrt{2\pi\sigma^2}} \right) + \frac{(y_i - \mu_{x_i})^2}{2\sigma^2} \end{aligned}$$

Negative Log-Likelihood (NLL)

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

Minimize MSE loss

$\hat{w}_a$        $\hat{w}_b$

Minimizing NLL loss  $\xrightarrow{\sigma \text{ const}}$  Minimizing MSE loss

# Losses in Keras

keras.io/api/losses/

Developer guides

Code examples

Keras 3 API documentation

Models API

Layers API

Callbacks API

Ops API

Optimizers

Metrics

**Losses**

Probabilistic losses

Regression losses

Hinge losses for "maximum-margin" classification

Data loading

Built-in small datasets

Keras Applications

Mixed precision

Multi-device distribution

RNG API

Utilities

The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.

## Available losses

Note that all losses are available both via a class handle and via a function handle. The class handles enable you to pass configuration arguments to the constructor (e.g. `loss_fn = CategoricalCrossentropy(from_logits=True)`), and they perform reduction by default when used in a standalone way (see details below).

▶ [Keras 3 API documentation / Losses / Probabilistic losses](#)

## Probabilistic losses

- `BinaryCrossentropy` class
- `BinaryFocalCrossentropy` class
- `CategoricalCrossentropy` class
- `CategoricalFocalCrossentropy` class
- `SparseCategoricalCrossentropy` class
- `Poisson` class
- `CTC` class
- `KLDivergence` class
- `binary_crossentropy` function
- `categorical_crossentropy` function
- `sparse_categorical_crossentropy` function
- `poisson` function
- `ctc` function
- `kl_divergence` function

## Regression losses

- `MeanSquaredError` class
- `MeanAbsoluteError` class
- `MeanAbsolutePercentageError` class
- `MeanSquaredLogarithmicError` class
- `CosineSimilarity` class

▶ [Keras 3 API documentation / Losses / Regression losses](#)

## Probabilistic losses

### `BinaryCrossentropy` class

```
keras.losses.BinaryCrossentropy(  
    from_logits=False,  
    label_smoothing=0.0,  
    axis=-1,  
    reduction="sum_over_batch_size",  
    name="binary_crossentropy",  
    dtype=None,  
)
```

▶ [Keras 3 API documentation / Losses / Regression losses](#)

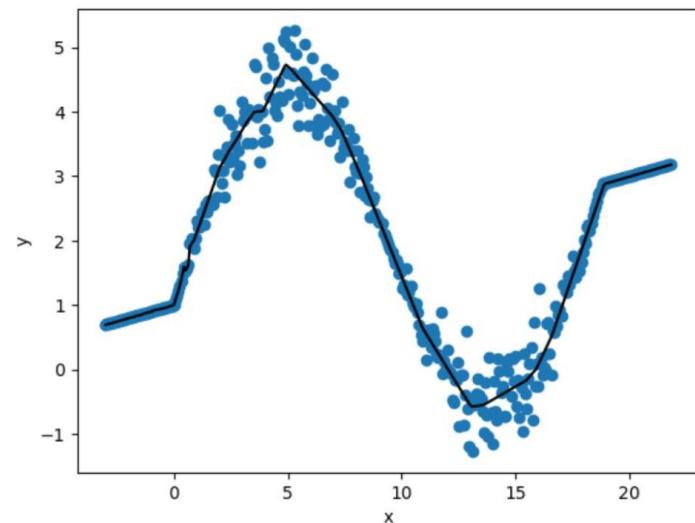
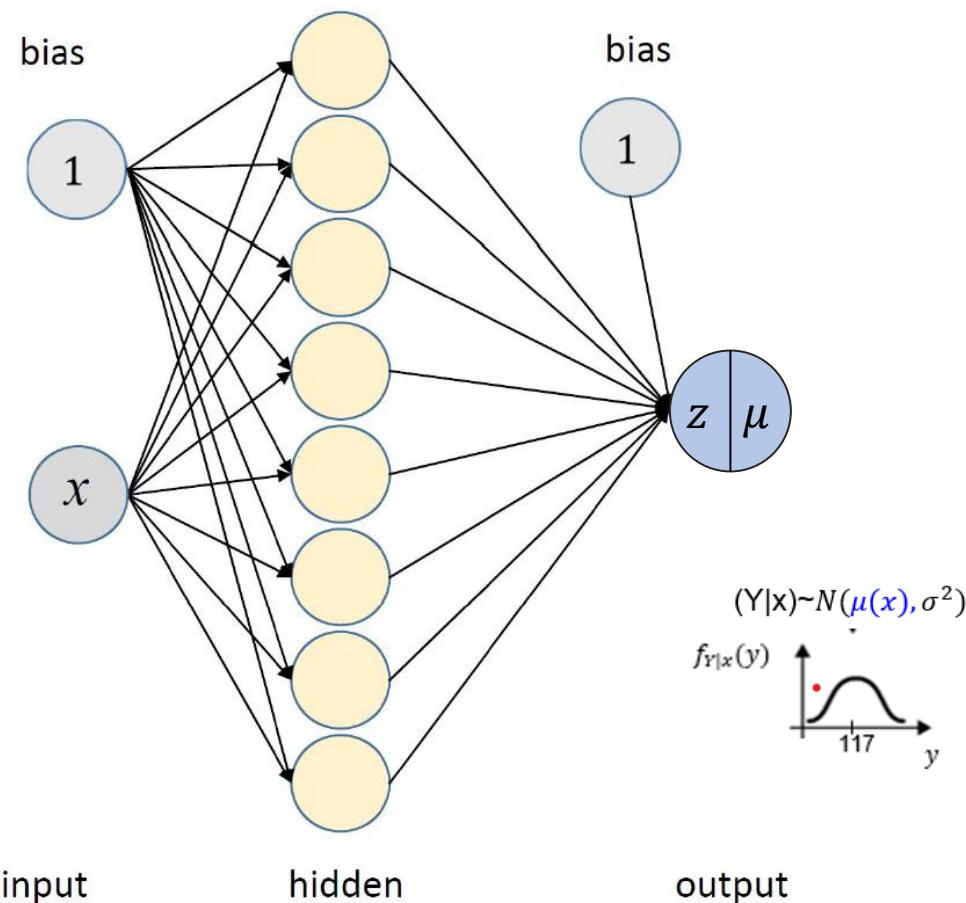
## Regression losses

### `MeanSquaredError` class

```
keras.losses.MeanSquaredError(  
    reduction="sum_over_batch_size", name="mean_squared_error", dtype=None  
)
```

# Non-linear Regression as NN

## Activation in hidden layer non-linear, in last layer linear

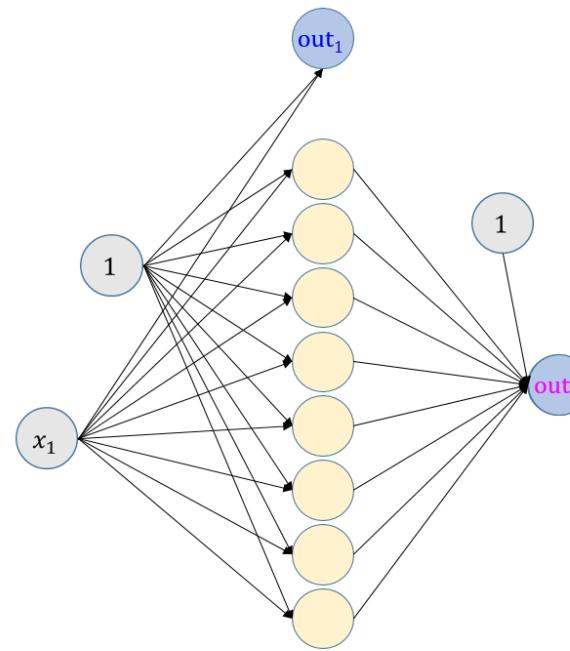
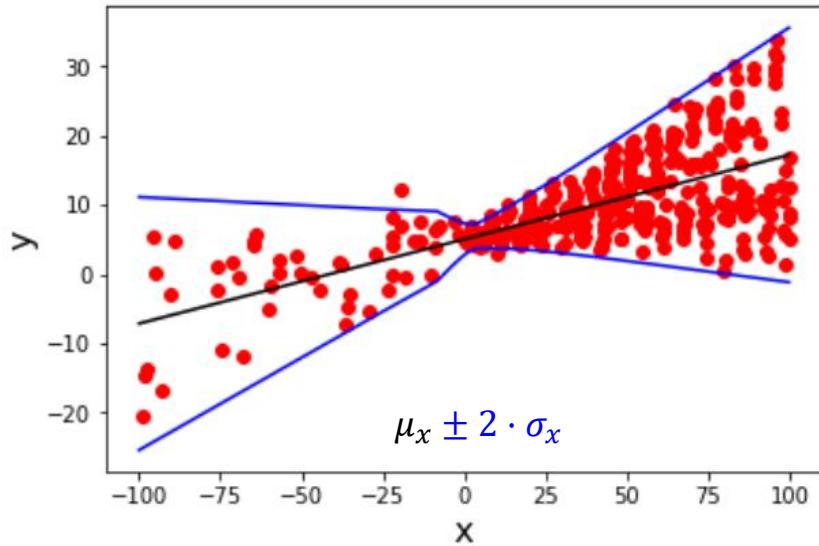


Here, the conditional outcome distribution is a Gaussian with a  $\mu$  that depends non-linearly on  $x$  😊 a variance  $\sigma^2$  that is constant ⚡

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

# Linear regression with flexible non-constant variance

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



$$\mu_{x_i} = out_{1_i}$$

$$\sigma_{x_i} = e^{out_{2_i}}$$

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

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

Note: we do not need to know the “ground truth for  $\sigma$ ” – the likelihood does the job!

# Define custom NLL loss for a probabilistic model

- Some keras losses correspond to NLL, e.g.
  - Linear Regression with constant variance
    - Mean Squared Error (keras.losses.mse)
  - Multiclass classification
    - Categorical\_crossentropy (keras.losses.categorical\_crossentropy)
- For a given outcome distribution family, you can build your custom NLL loss even if keras does not provide it
  1. Define a model with as many outputs as the number of parameters of the outcome distribution family
  2. Link the outputs of the NN to the parameters of the outcome distribution
  3. Use negative log likelihood as loss function

## 1. Define a model with as many outputs as the number of parameters of the distribution



```
# Define the model
inputs = Input(shape=(1,))
hidden = layers.Dense(10, activation="relu")(inputs)
# ... more hidden layers ...
outputs = layers.Dense(2)(hidden) # <--- Outputs mean and log(sd)
model = Model(inputs=inputs, outputs=outputs)
```

[8] ✓ 0.0s

Python

## 2. Link the outputs of the model to the parameters of the distribution

```
# Wrapper function to convert model output to a PyTorch Normal distribution
@staticmethod
def output_to_gaussian_distribution(output):
    mean = output[:, 0:1] # Gets the first column while keeping dimensions (like output[, 1, drop=FALSE] in R)
    log_sd = output[:, 1:2] # Gets the second column while keeping dimensions
    scale = torch.exp(log_sd) # Inverse link function to ensure positive scale
    return Normal(loc=mean, scale=scale)
```

[7] ✓ 0.0s

Python

## 3. Define NLL

```
# Custom Negative Log-Likelihood Loss
def negative_log_likelihood(y_true, output):
    dist = output_to_gaussian_distribution(output)
    return -dist.log_prob(torch.tensor(y_true)).mean()

# Compile the model
model.compile(optimizer="adam", loss=negative_log_likelihood)
```

[10] ✓ 0.0s

Python

# Exercise: Define a custom loss function

Notebook: [02\\_custom\\_loss.ipynb](#)

[https://github.com/tensorchiefs/dl\\_course\\_2025/blob/main/notebooks/02\\_custom\\_loss.ipynb](https://github.com/tensorchiefs/dl_course_2025/blob/main/notebooks/02_custom_loss.ipynb)

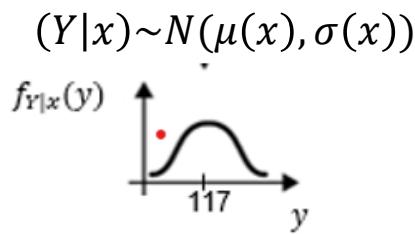
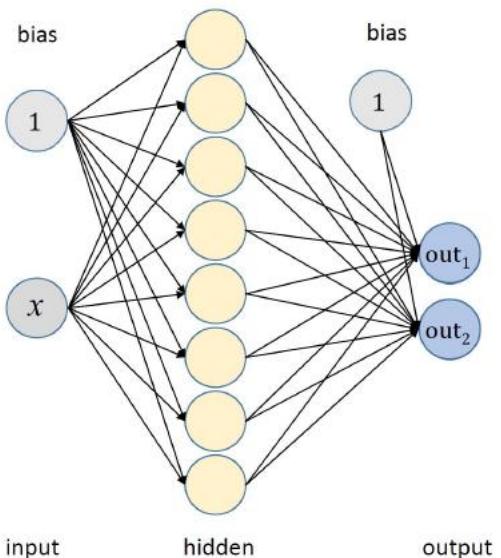
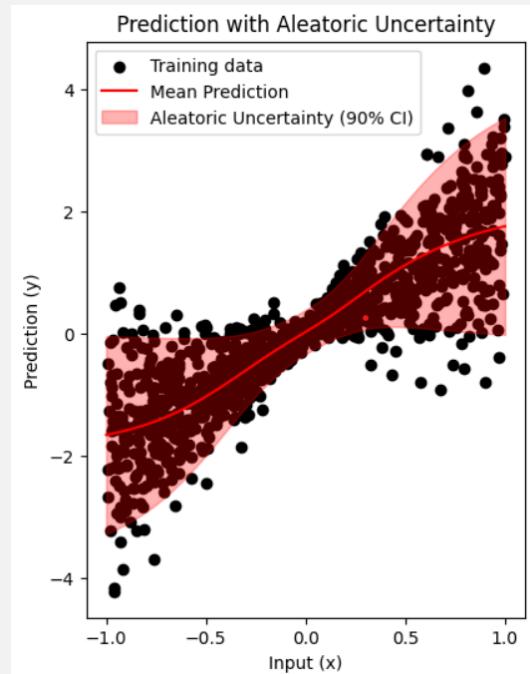
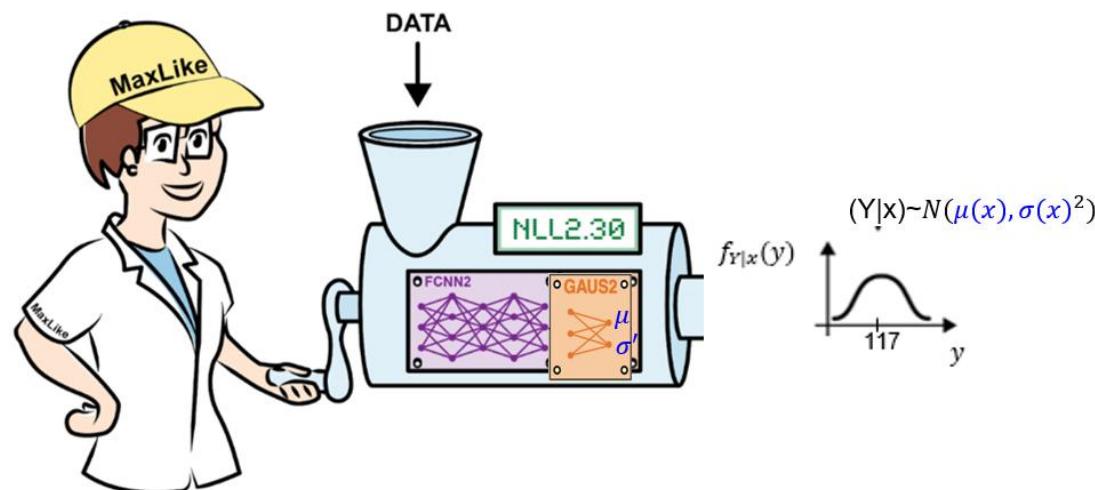
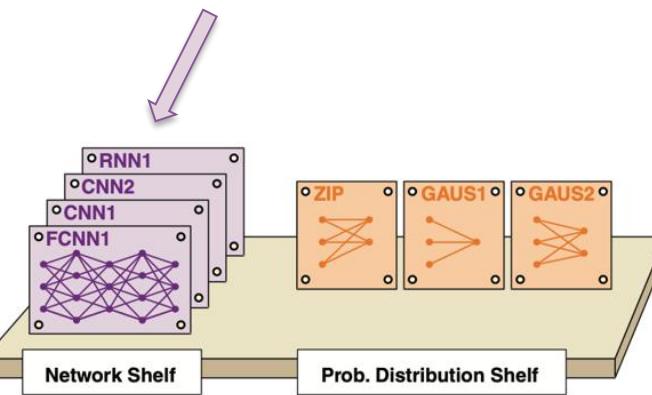


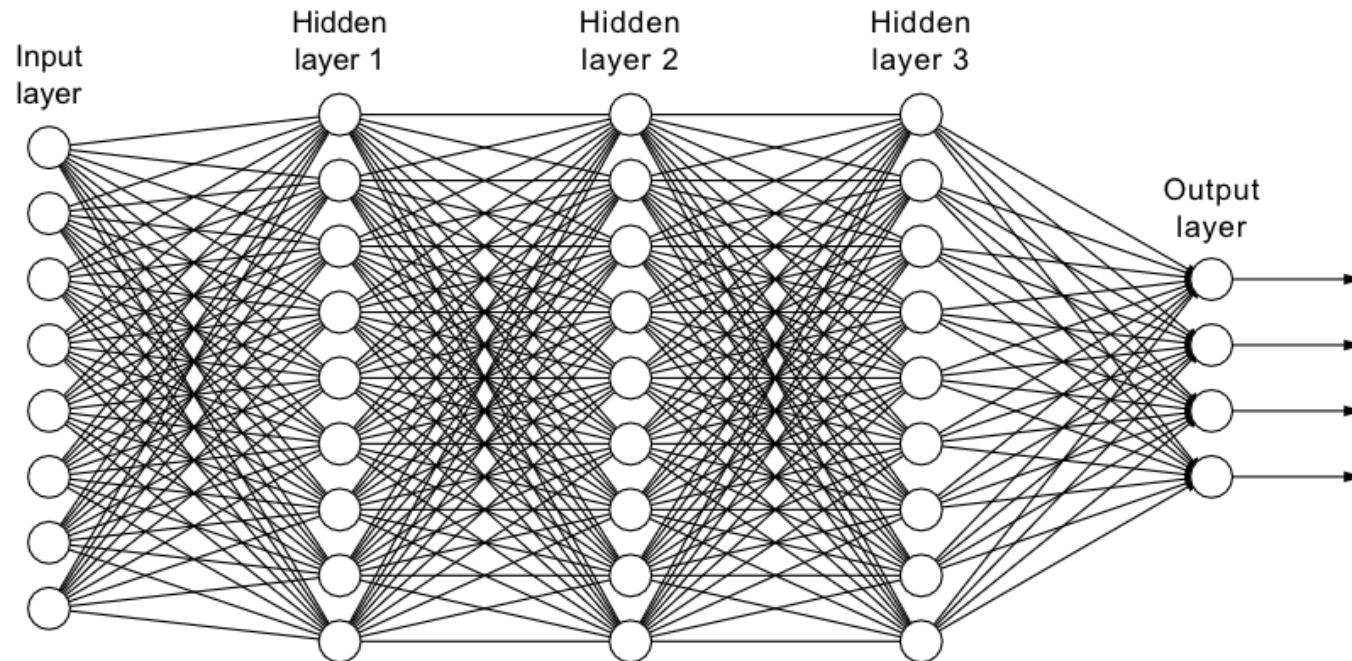
Figure 4.19 A NN with two output nodes can be used to control the parameter  $\mu_x$  and  $\sigma_x$  of the conditional outcome distribution  $N(\mu_x, \sigma_x)$  for regression tasks with non-constant variance



# NN architectures

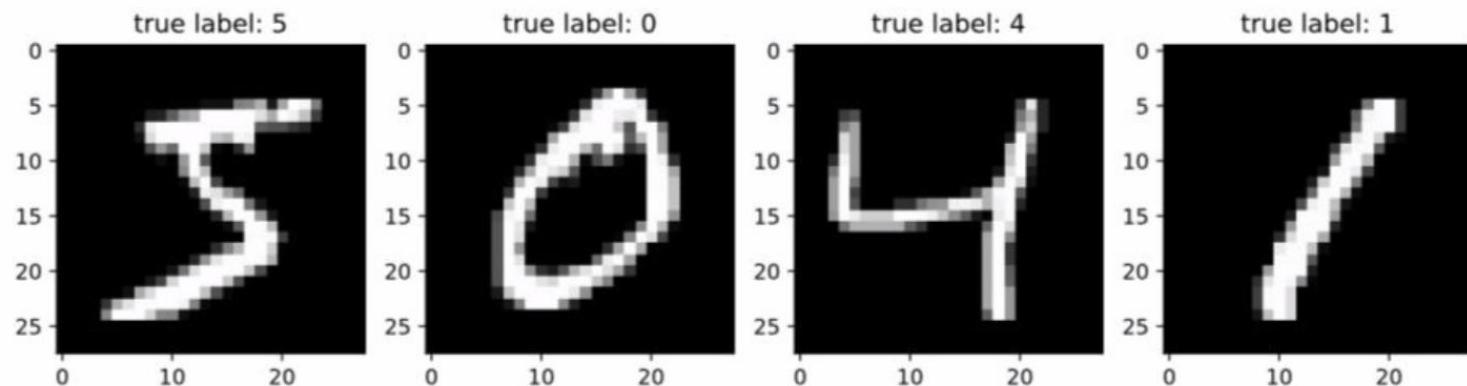


# A fully connected NN

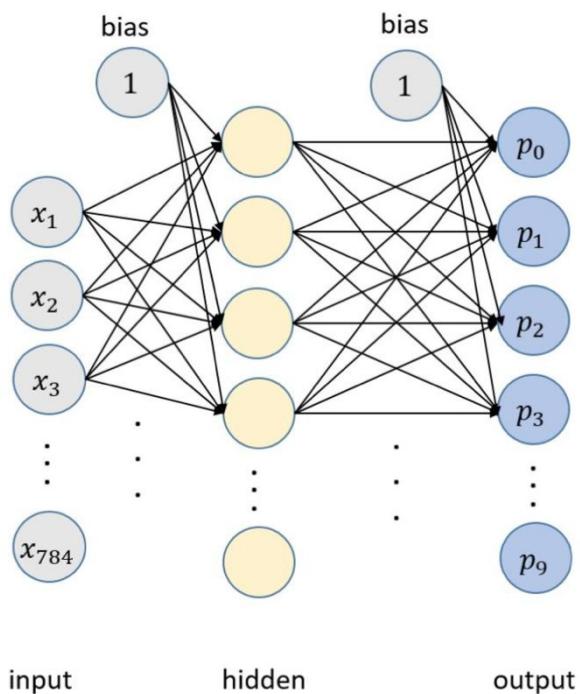


# Probabilistic classification

- Usually in DL the model predicts a probability for each possible class
- Example:
  - Banknote from exercise – classes: “real” or “fake”
  - Typical example Number from hand-written digit – classes: 0, 1, ..., 9



# Classification: Softmax Activation



$p_0, p_1 \dots p_9$  are probabilities for the classes 0 to 9.

Activation of last layer  $z_i$  incoming

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^9 e^{z_j}}$$

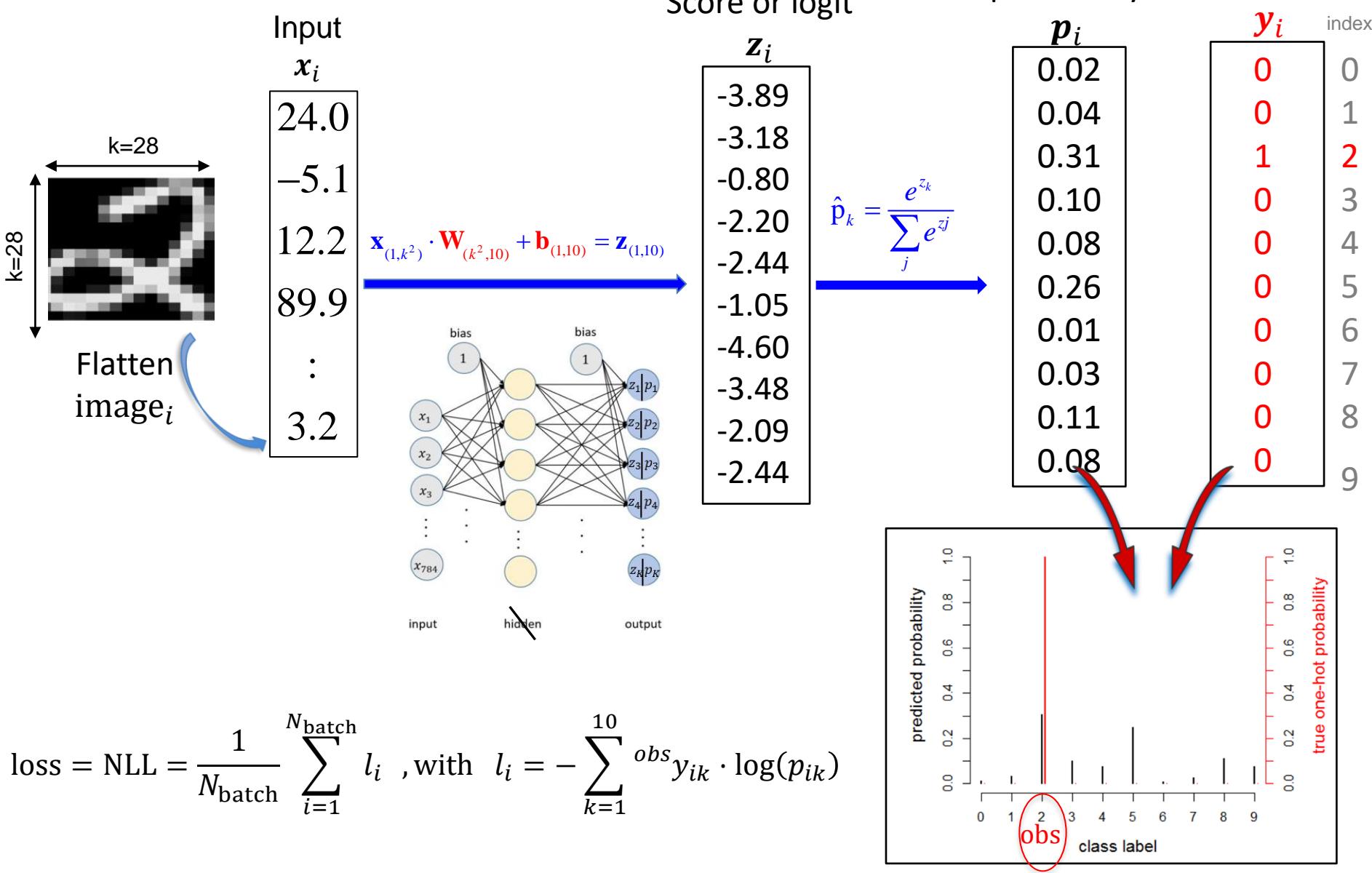
Makes outcome positive

Ensures that  $p_i$ 's sum up to one

This activation is called softmax

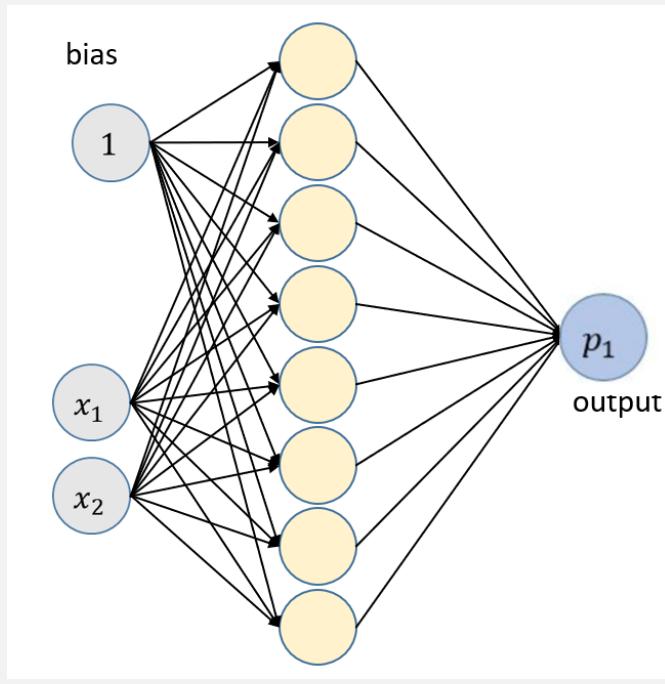
Figure 2.12: A fully connected NN with 2 hidden layers. For the MNIST example, the input layer has 784 values for the 28 x 28 pixels and the output layer out of 10 nodes for the 10 classes.

# NLL for multi-class = cross-entropy



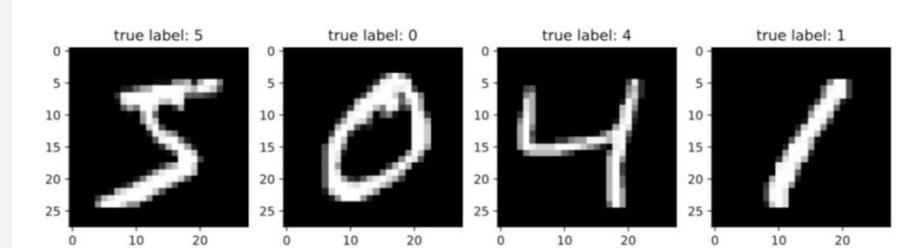
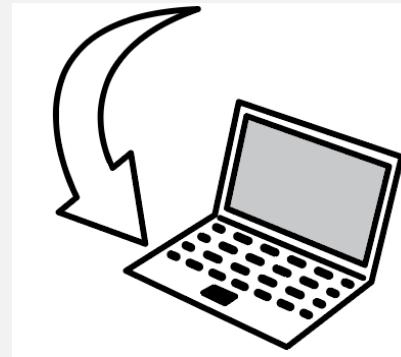
Loss = NLL = cross-entropy ( $-\sum p_i^{\text{obs}} \log p_i^{\text{pred}}$ ) averaged over all images in mini-batch

# Exercise:



input

hidden



Use a fully connected NN to classify MNIST data to 10 possible classes

[03\\_fcnn\\_mnist\\_keras\\_torch.ipynb](#)

# Summary

- If the outcome is continuous we say we do regression
- If the outcome is categorical we say we do classification
- Probabilistic models predict an outcome distribution that assigns for each possible outcome a probability/likelihood
- When using NNs to fit a probabilistic model we use the NLL as loss
- The NLL-contribution of one data point  $(x_i, y_i)$  is  $-\log(L_i)$ : the negative log of the likelihood  $L_i$  which is by the predicted outcome distribution assigned to the observed value  $y_i$  (see visualization).
- Hidden layers allow for
  - Non linear decision boundaries if the outcome is categorical
  - For non-linear regression modeling with flexible variance if the outcome is continuous
- For a binary classification task, we can use
  - a single neuron in the last layer with the sigmoid activation and the Keras 'binary\_crossentropy'.
  - Two neurons in the last layer with the softmax activation and the Keras loss "categorical\_crossentropy"
- For a regression task where the outcome distribution is a Gaussian  $N(\mu(x), \sigma(x))$ , we can use a NN with two neurons in the last layer and a custom NLL loss using one NN-output to determine  $\mu(x)$ , and the other NN-output to determine  $\sigma(x)$ , needed to describe the predicted outcome distribution, under which the likelihood of the observation needs to be determined

