Karlsruher Institut für Technologie

Machine Learning

Lecture 1

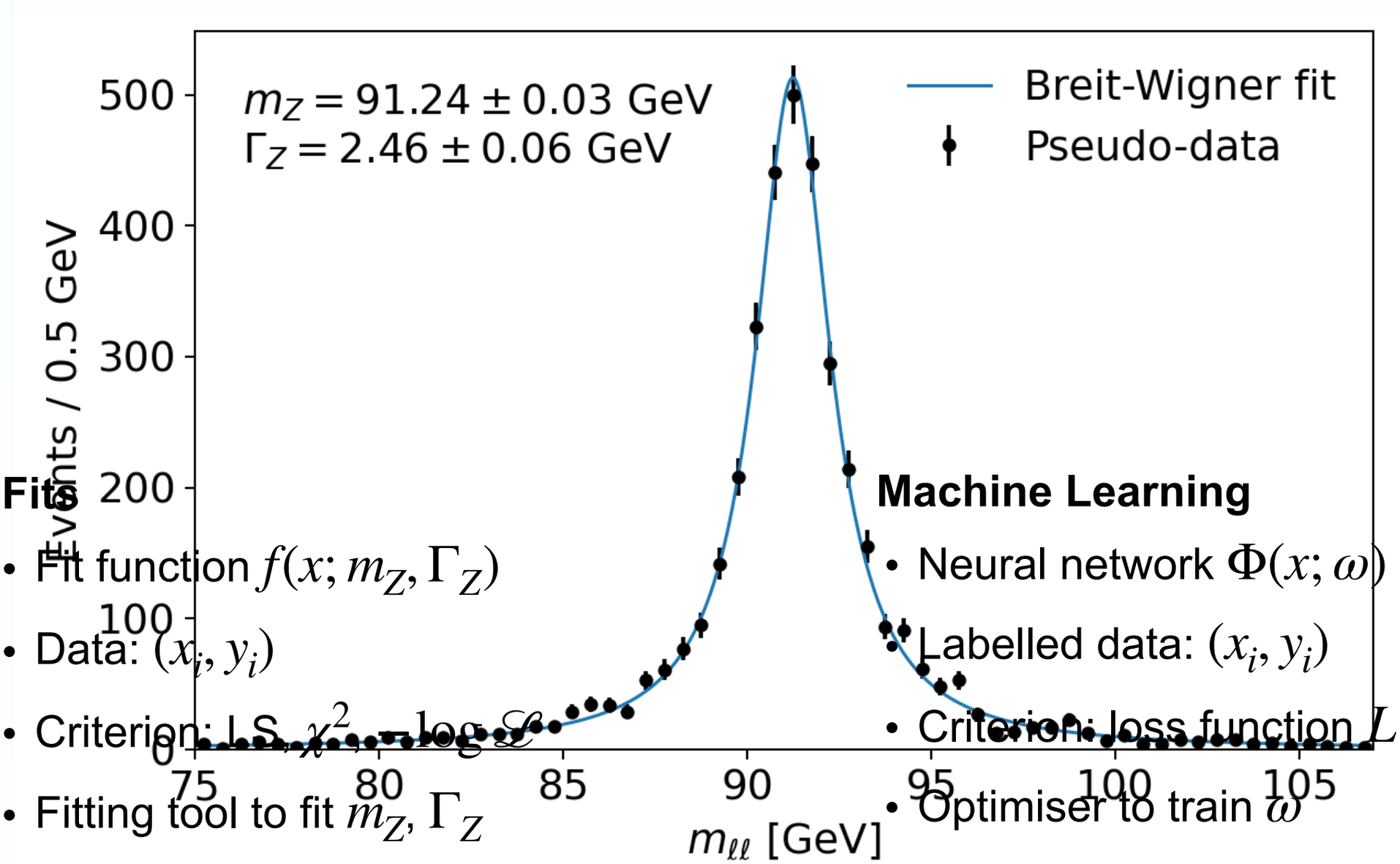
-

Learning a Function

Jan Kieseler

jan.kieseler@kit.edu

Who has performed a fit like this?



Fits

- Fit function $f(x; m_Z, \Gamma_Z)$
- Data: (x_i, y_i)
- Criterion: LS, χ^2, $-\log \mathscr{L}$
- Fitting tool to fit m_Z, Γ_Z

Machine Learning

- Neural network $\Phi(x; \omega)$
- Labelled data: (x_i, y_i)
- Criterion: loss function L
- Optimiser to train ω

➡ Measure m_Z, Γ_Z

➡ Predict $\hat{y}_j$ for new x_j with unknown y_j

Turn your own Physics Problem into a Well-Posed ML Problem

Lecture 1: Learning a Function

understand the complete basic learning loop and how
a neural network is trained to approximate a deterministic function

Lecture 2: Exploiting Structure

reason from the structure and symmetries of a problem to
an appropriate representation and architecture

Lecture 3: Learning a Process

understand how random inputs turn deterministic neural networks
into generators, and how to train and evaluate them.

Ask questions at any time

Distill the most important take-aways:
we will write the conclusions together at the end of each lecture

Topics for today

Fits

- Fit function $f(x; m_Z, \Gamma_Z)$
- Data: (x_i, y_i)
- Criterion: LS, χ^2, $-\log \mathscr{L}$
- Fitting tool to fit m_Z, Γ_Z

➡ Measure m_Z, Γ_Z

Machine Learning

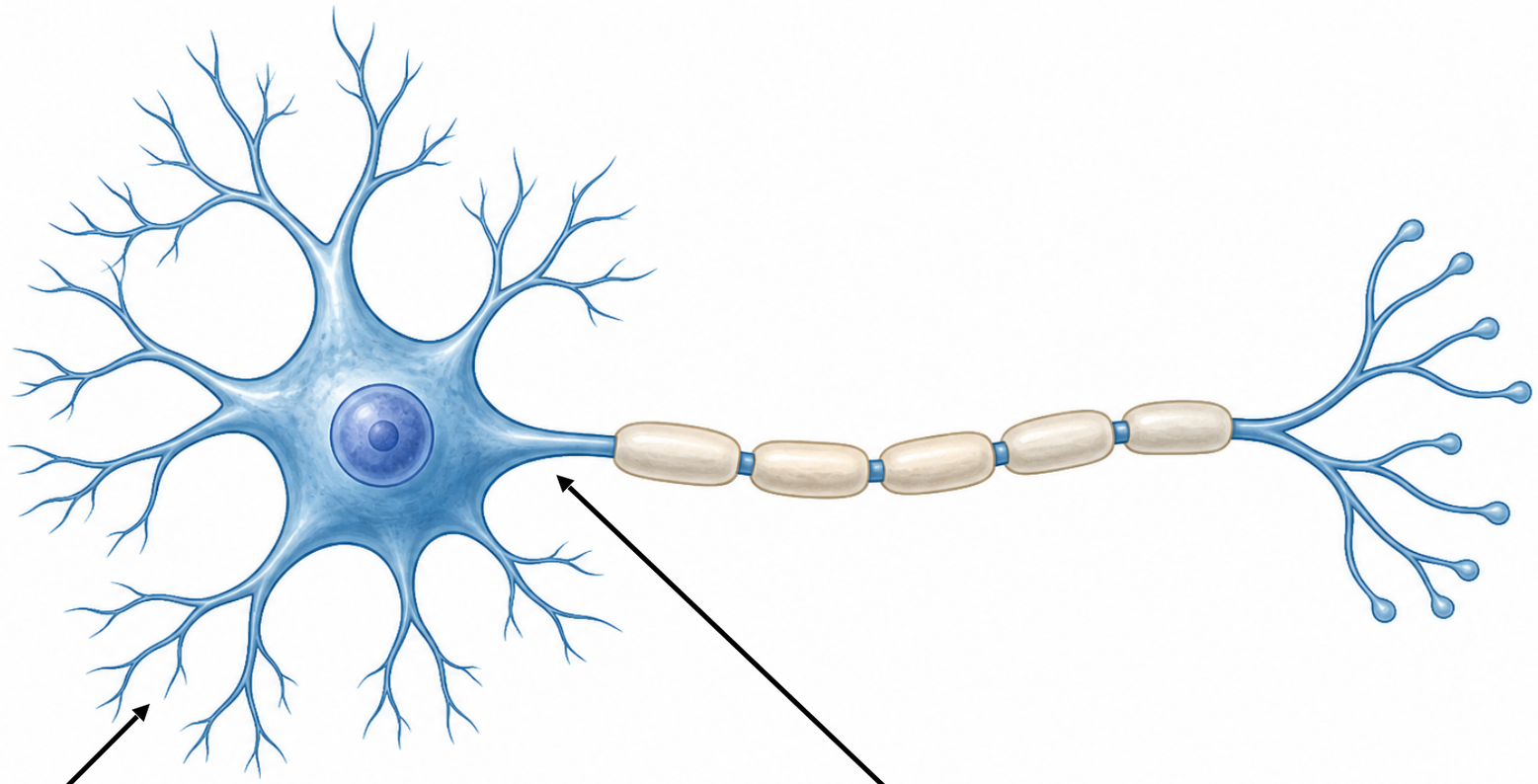
- Neural network $\Phi(x; \omega)$
- Labelled data: (x_i, y_i)
- Criterion: loss function L
- Optimiser to train ω

➡ Predict $\hat{y}_j$ for new x_j with unknown y_j

Assess NN model performance

What is a NN: Inspiration from nature

- Artificial neural networks were historically inspired by highly simplified models of biological neurons.

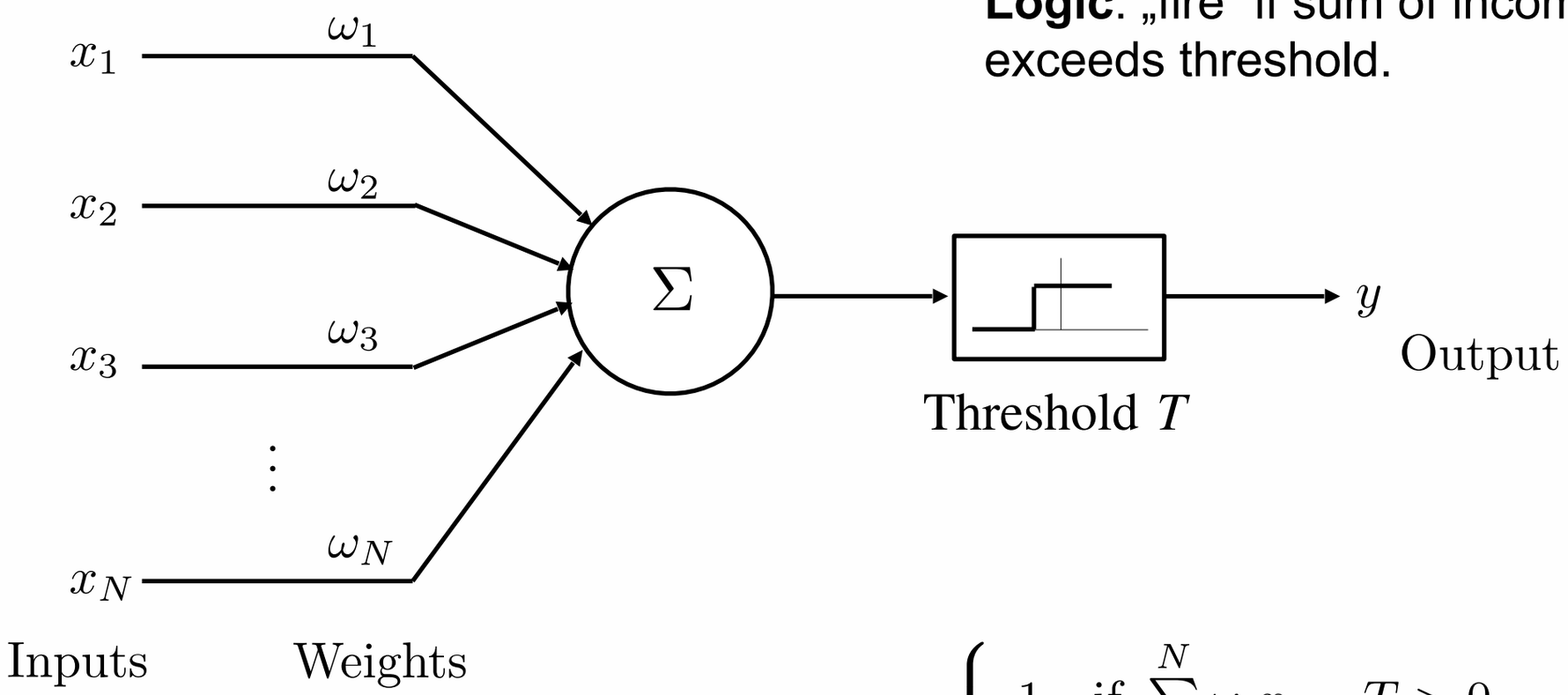


Many occasionally small signals enter here.

If their sum exceeds a threshold, the neuron fires and sends a signal along the axon.

The Perceptron

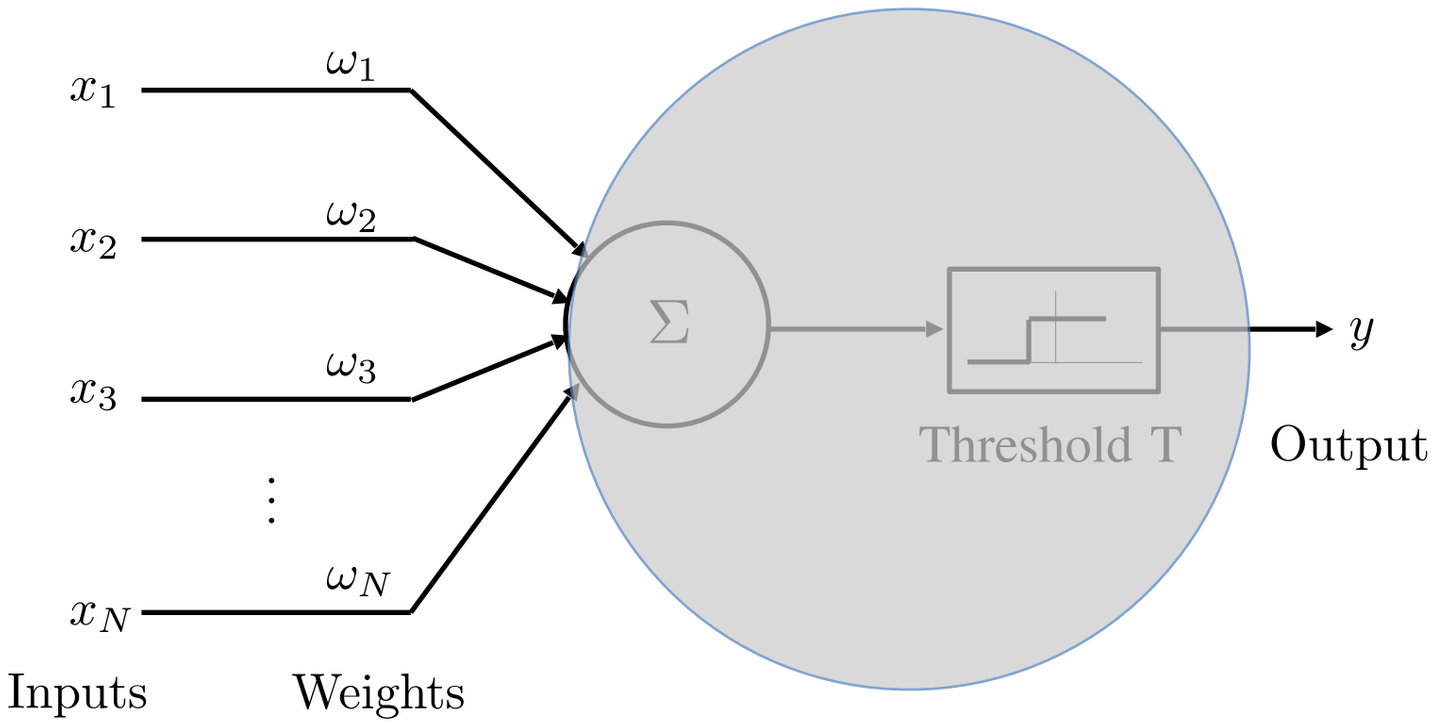
- Rosenblatt's perceptron turns this intuition into a mathematical model:

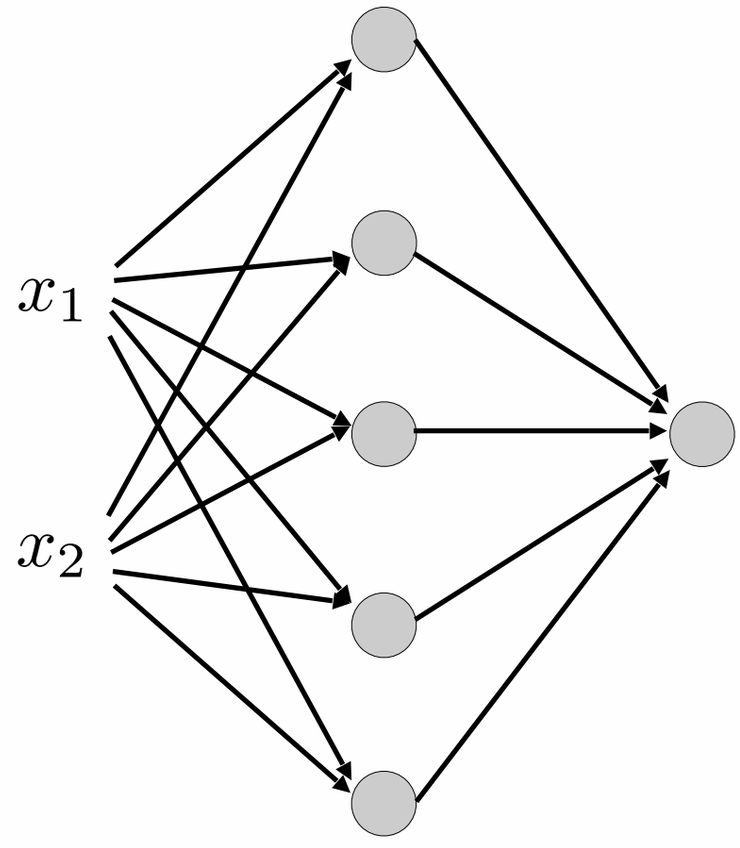


Logic: „fire" if sum of incoming signals exceeds threshold.

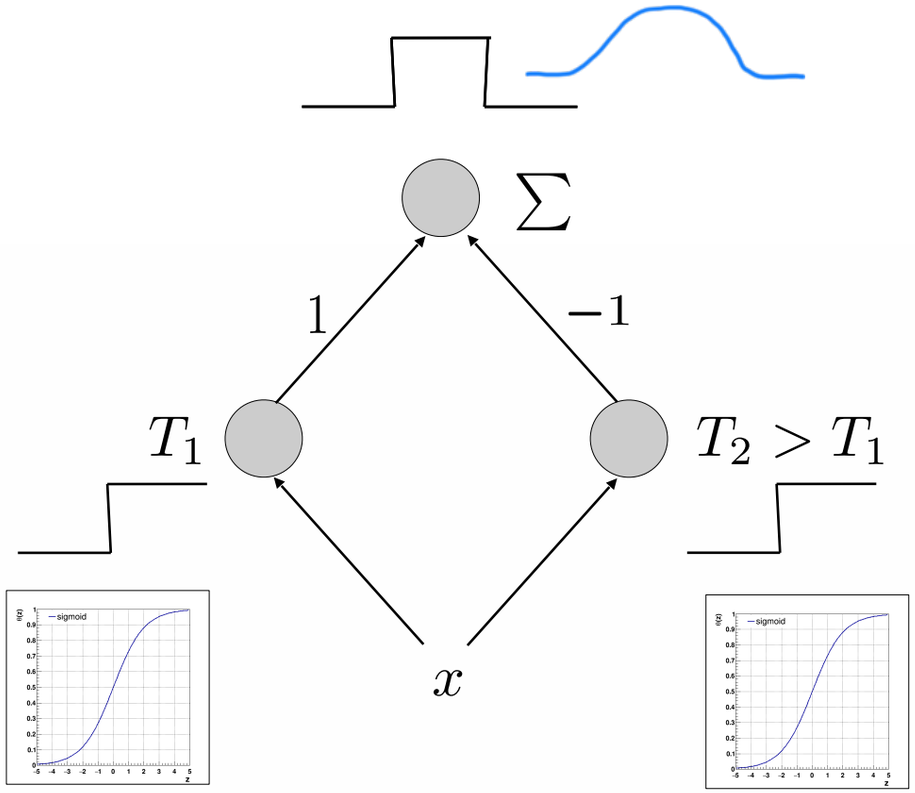
$$y = \begin{cases} 1 & \text{if } \sum_i^N \omega_i x_i - T > 0 \\ 0 & \text{else} \end{cases}$$

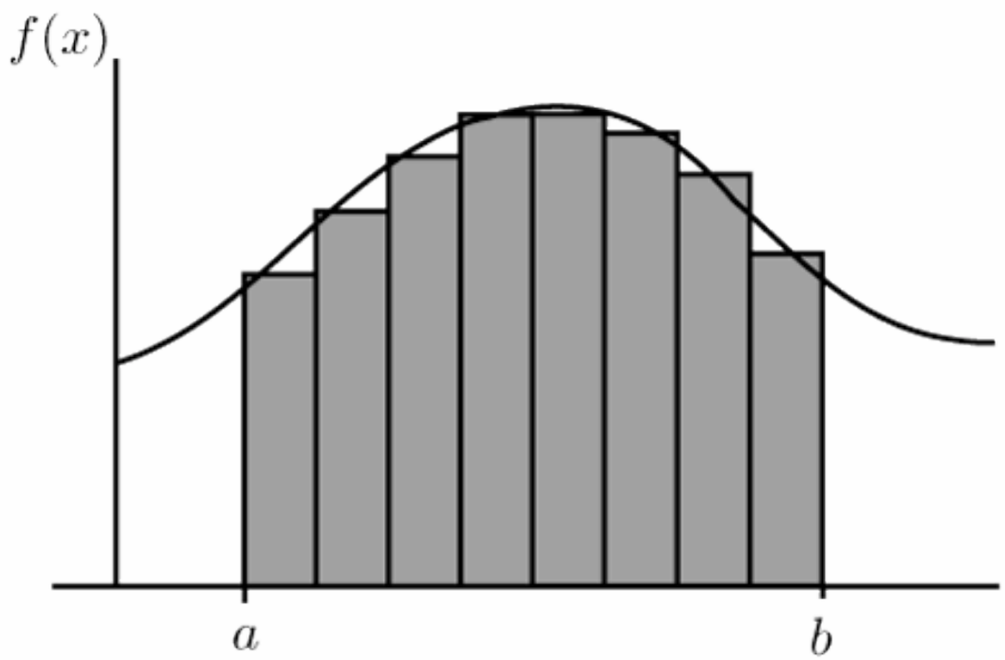
The Multi-Layer-Perceptron





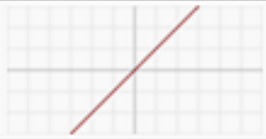
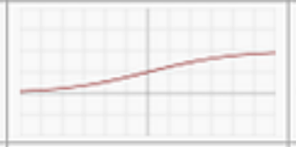
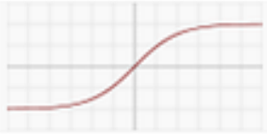
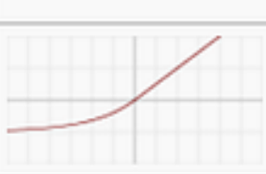
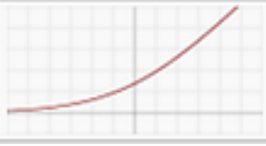
Approximation of an Arbitrary Function





Each hidden unit creates one simple contribution; the next layer weights and adds these contributions into a more complicated function.

Activation Functions

Name	Plot	Equation
Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$

- One perceptron:

$$y = \theta\left(\sum \omega_i x_i - T\right)$$

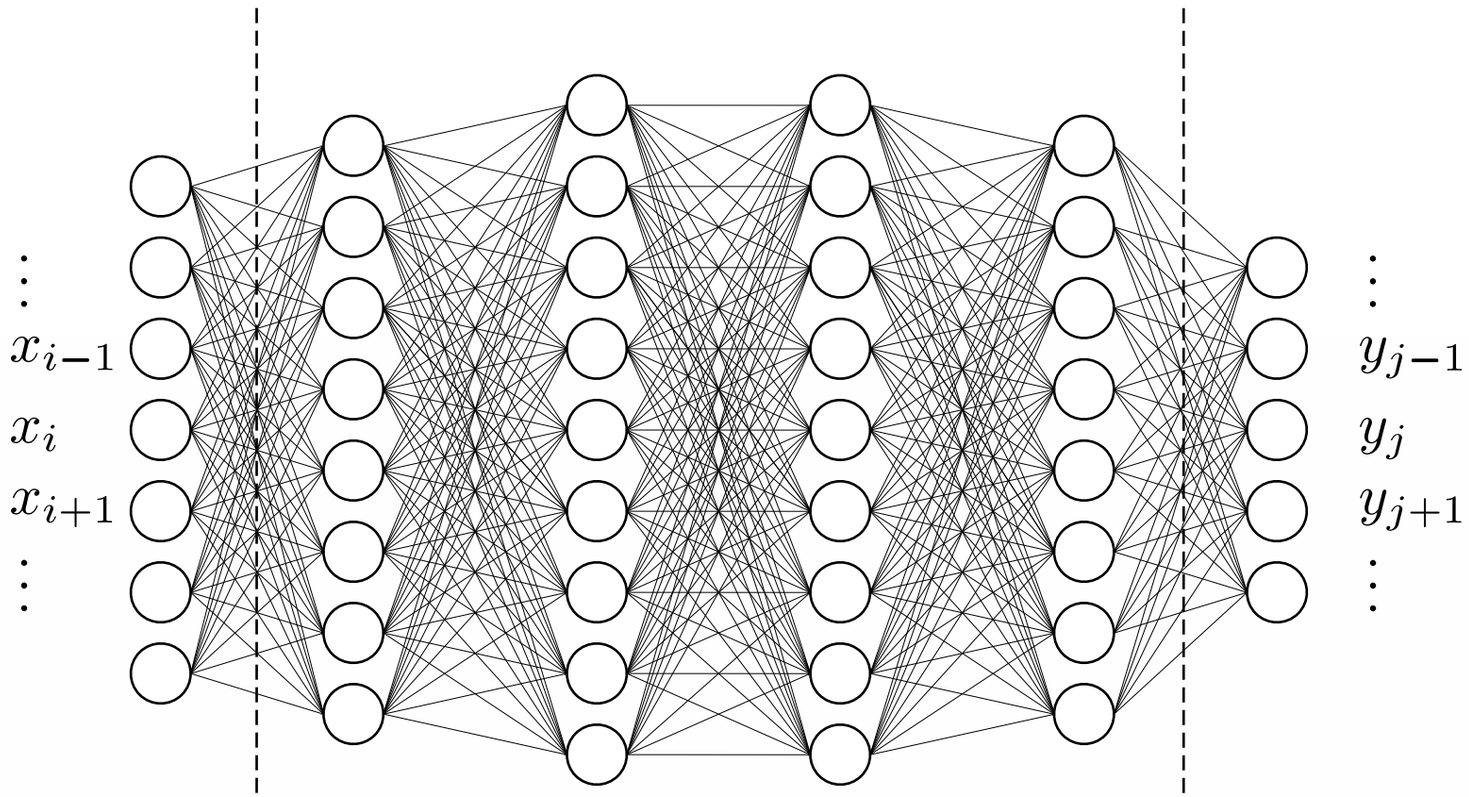
My recommendation:

Rectified Linear Unit (ReLU)	
Exponential Linear Unit (ELU) [3]	

- For ordinary MLP hidden layers, ReLU-like activations are usually a robust default.

https://patrickhoo.wixsite.com/diveindatascience/single-post/2019/06/13/activation-functions-and-when-to-use-them

MLPs: Terminology & Mathematical Representation



$\mathbf{x} \in \mathcal{X}$
Input layer

$\boldsymbol{\omega} \in \Omega$
Hidden layers

$\mathbf{y} \in \mathcal{Y}$
Output layer

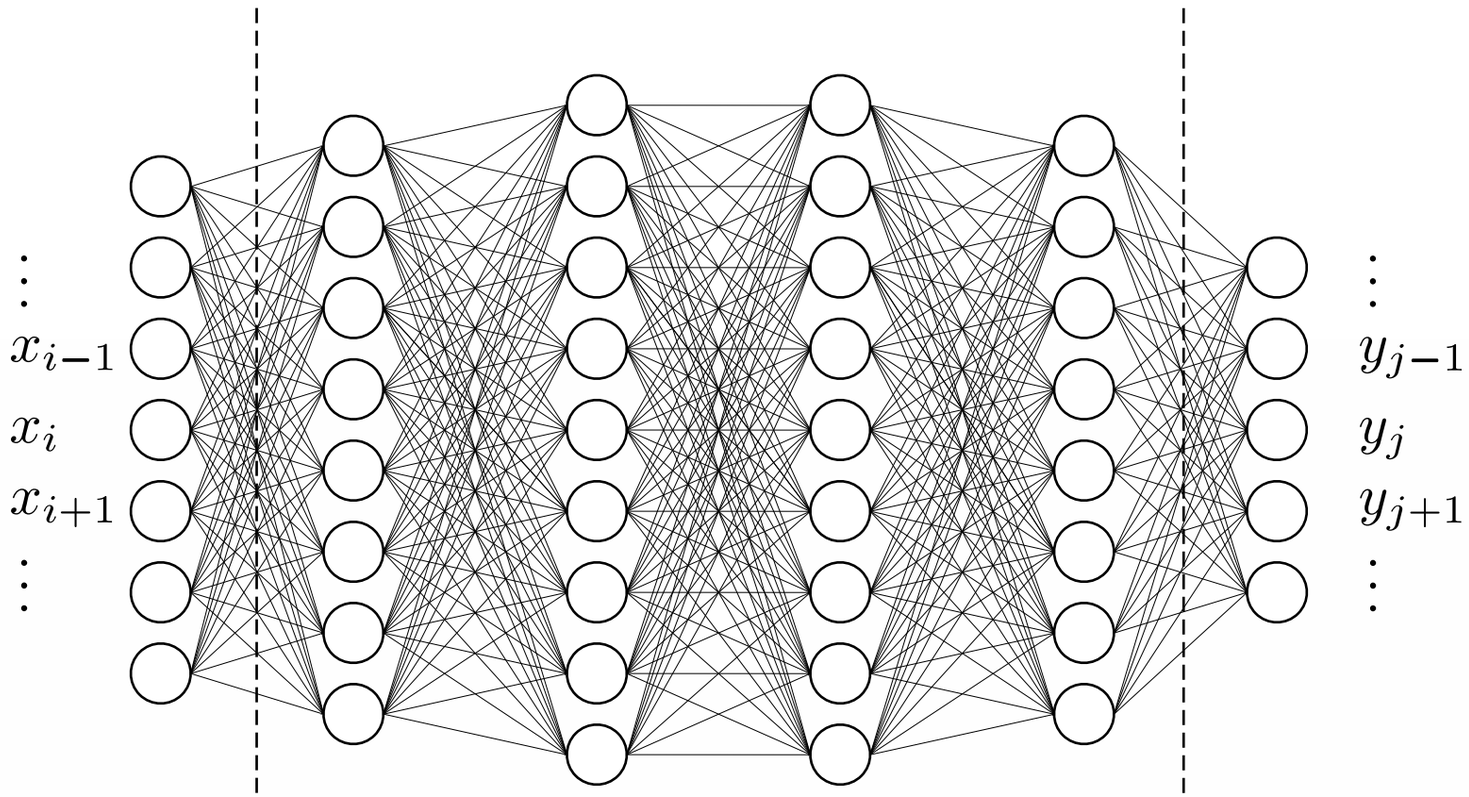
- One layer: $h^{(l)} = \theta(W_l h^{(l-1)} + b_l)$ $T = -b$

Activation function
$\dim(h^{(l)}) \rightarrow \dim(h^{(l)})$

Weight matrix
$\dim(h^{(l)}) \times \dim(h^{(l-1)})$

Bias vector: $\dim(h^{(l)})$

MLPs: Recursive Definition



$\mathbf{x} \in \mathcal{X}$
Input layer

$\boldsymbol{\omega} \in \Omega$
Hidden layers

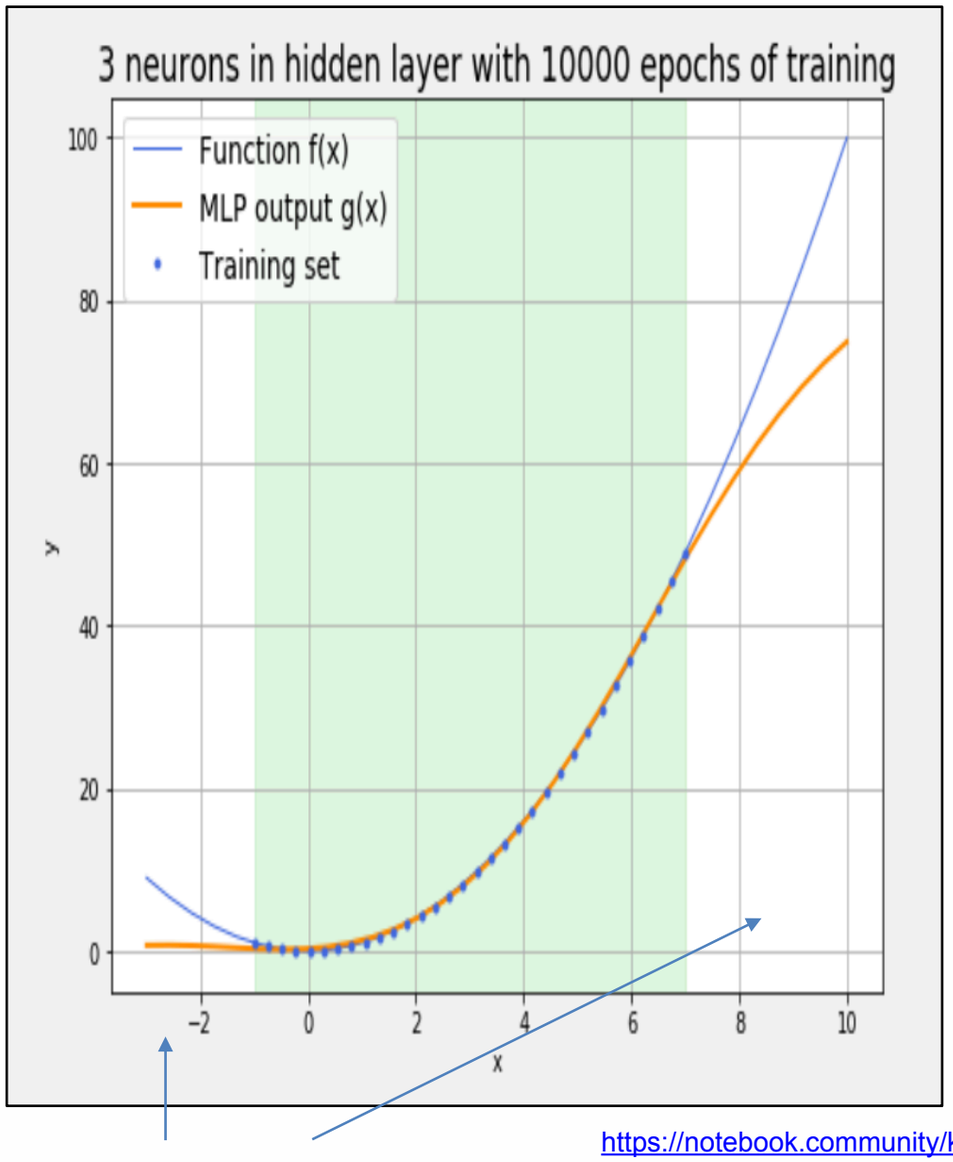
$\mathbf{y} \in \mathcal{Y}$
Output layer

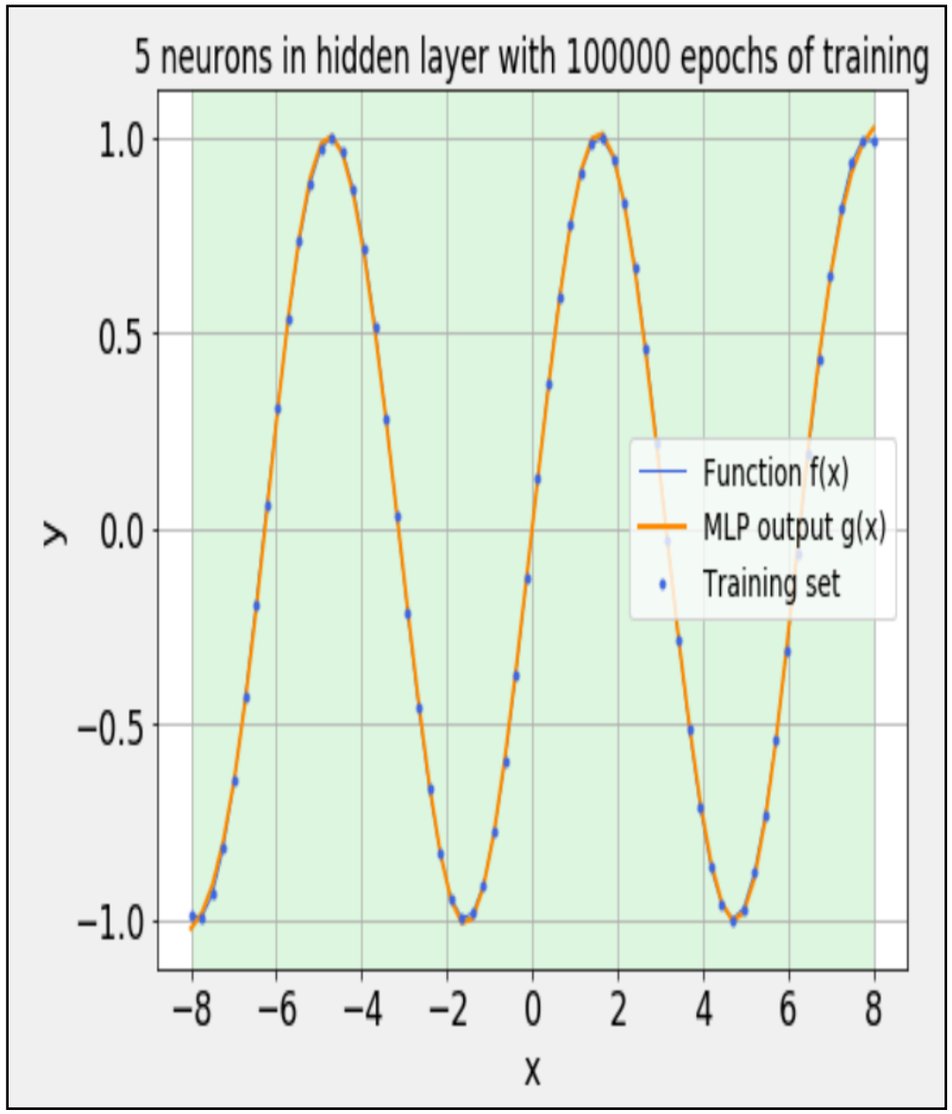
- One layer: $h^{(l)} = \theta(W_l h^{(l-1)} + b_l)$
- Full DNN: $\hat{y}(x) = \Phi(x; \omega) = h^{(L)}, h^{(0)} = x$

Powerful Approximators

- Very simple NN: one hidden layer, one input, one output, $\tanh$ activation

$$\Phi(x;\omega) = W^{(2)} \tanh(W^{(1)}x + b^{(1)}) + b^{(2)}$$

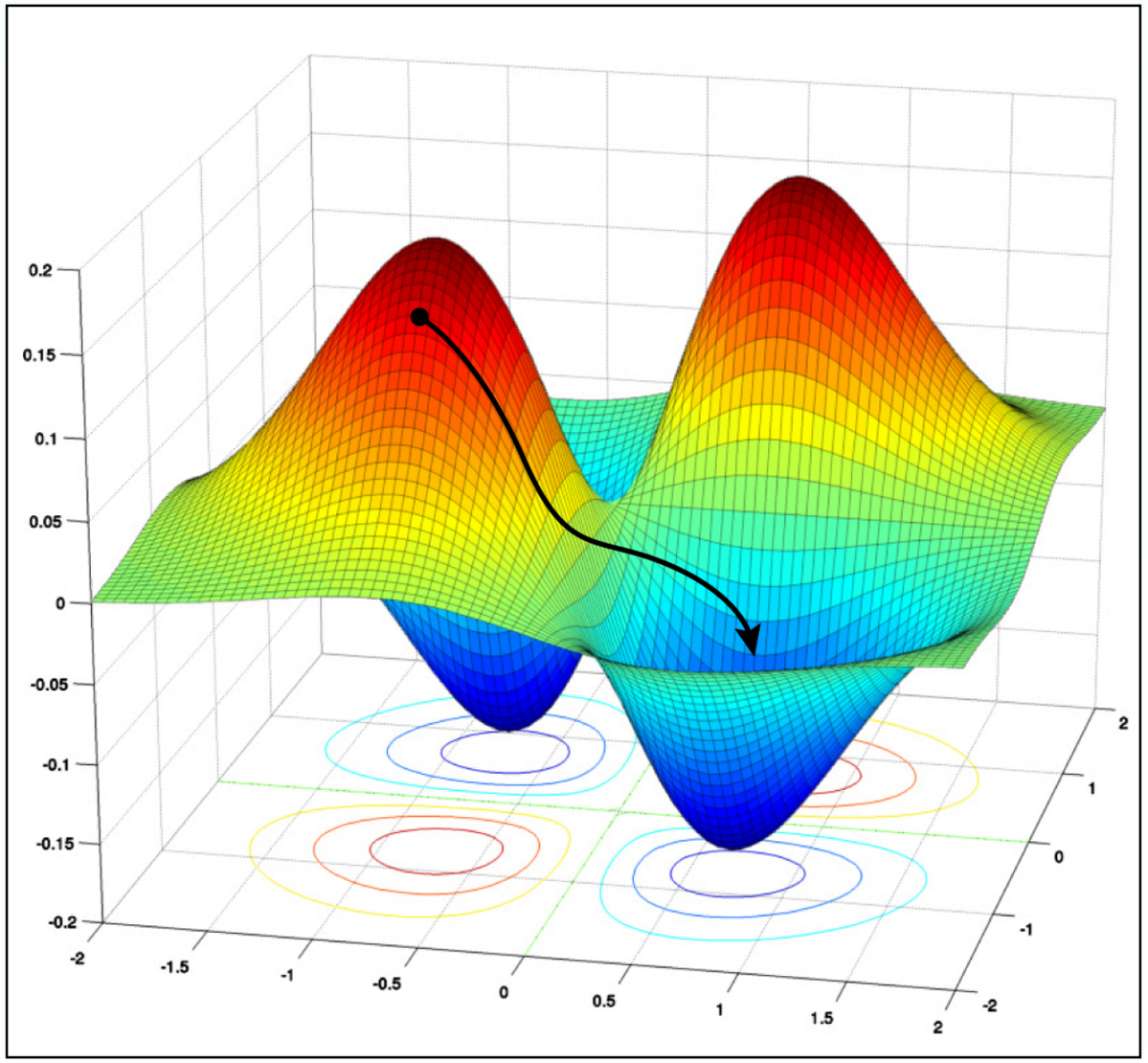




https://notebook.community/kit-cel/lecture-examples/mloc/ch3_Deep_Learning/pytorch/function_approximation_with_MLP

"Out-of-distribution"

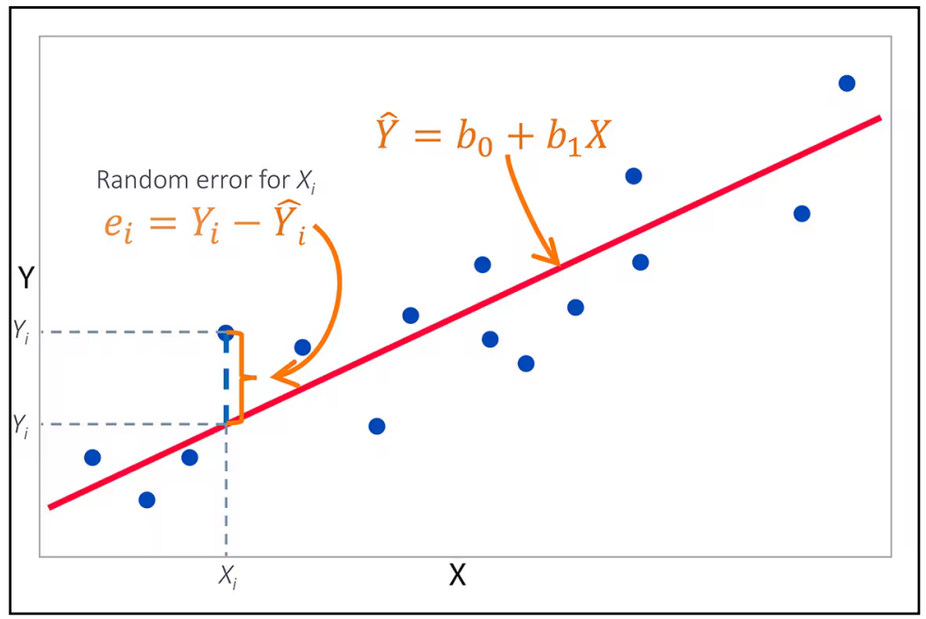
Training



https://medium.com/@tejovk311/optimization-challenges-in-deep-learning-a4b085d529b6

Loss Function

- The loss quantifies disagreement between **predictions** $\hat{y}_i$ and **truth targets** y_i on a dataset



- E.g. text-book linear regression:
  - $\hat{y} = \Phi(x; \omega = \{a, b\}) = ax + b$
  - Least-squares method (Gaussian $-\log \mathscr{L}$):

$$\sum_i^N \left((\Phi(x_i; \omega) - y_i)^2 \right) \rightarrow L_{MSE} = 1/N \sum_i^N \left((\Phi(x_i; \omega) - y_i)^2 \right)$$

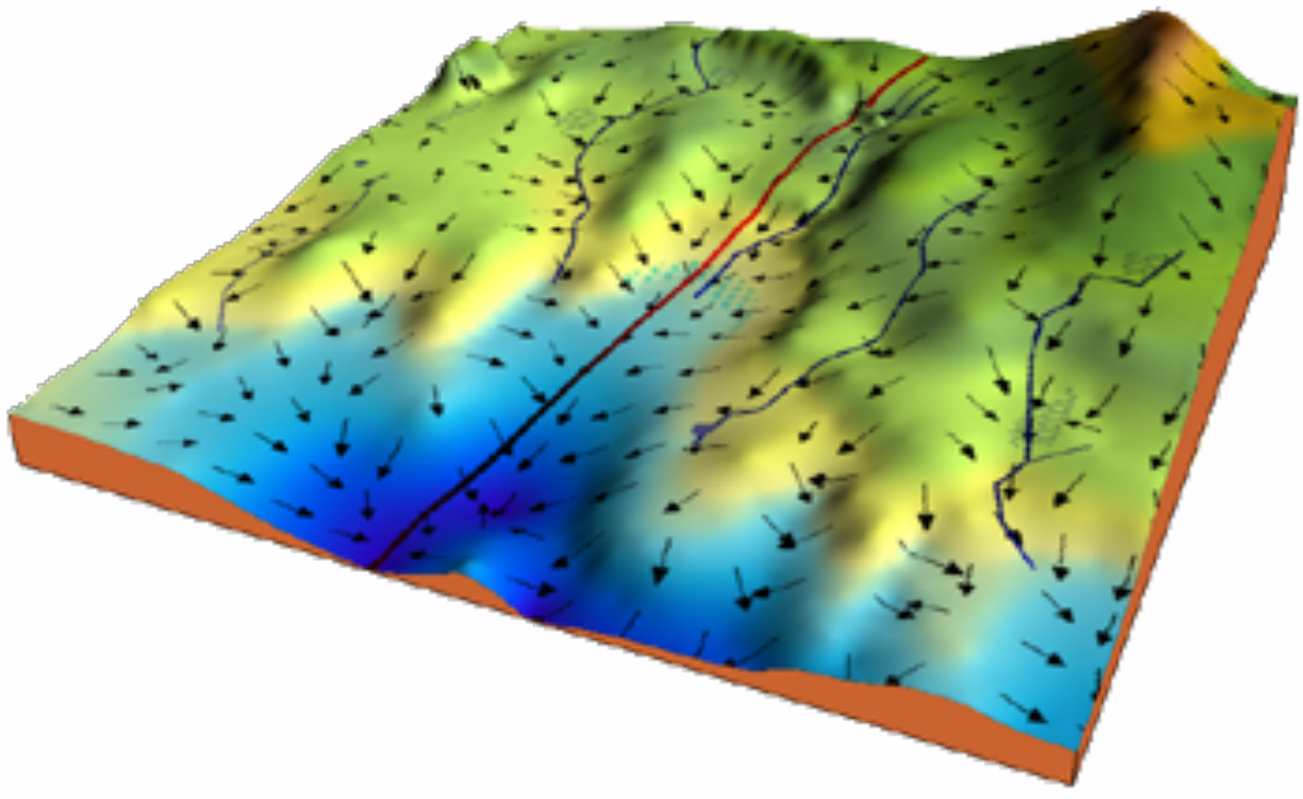
- The mean squared error loss is a standard loss for **regression** tasks

Gradient Descent

- Well established, robust numerical minimisation procedure:

$$\omega^{(s+1)} = \omega^{(s)} - \eta \nabla_{\omega^{(s)}} L\left(\Phi(x; \omega^{(s)}), y\right)$$

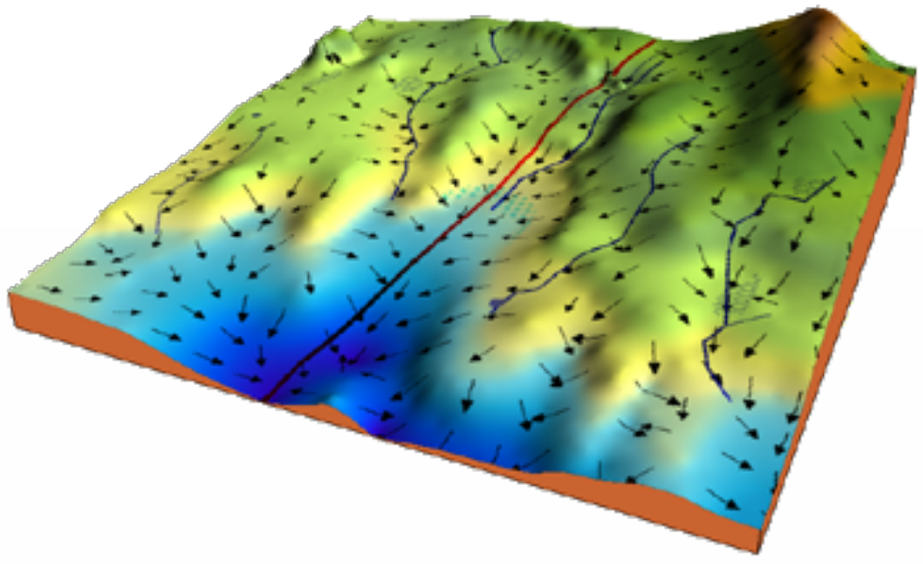
Learning rate



https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Momentum

Goodfellow et al. (2016)



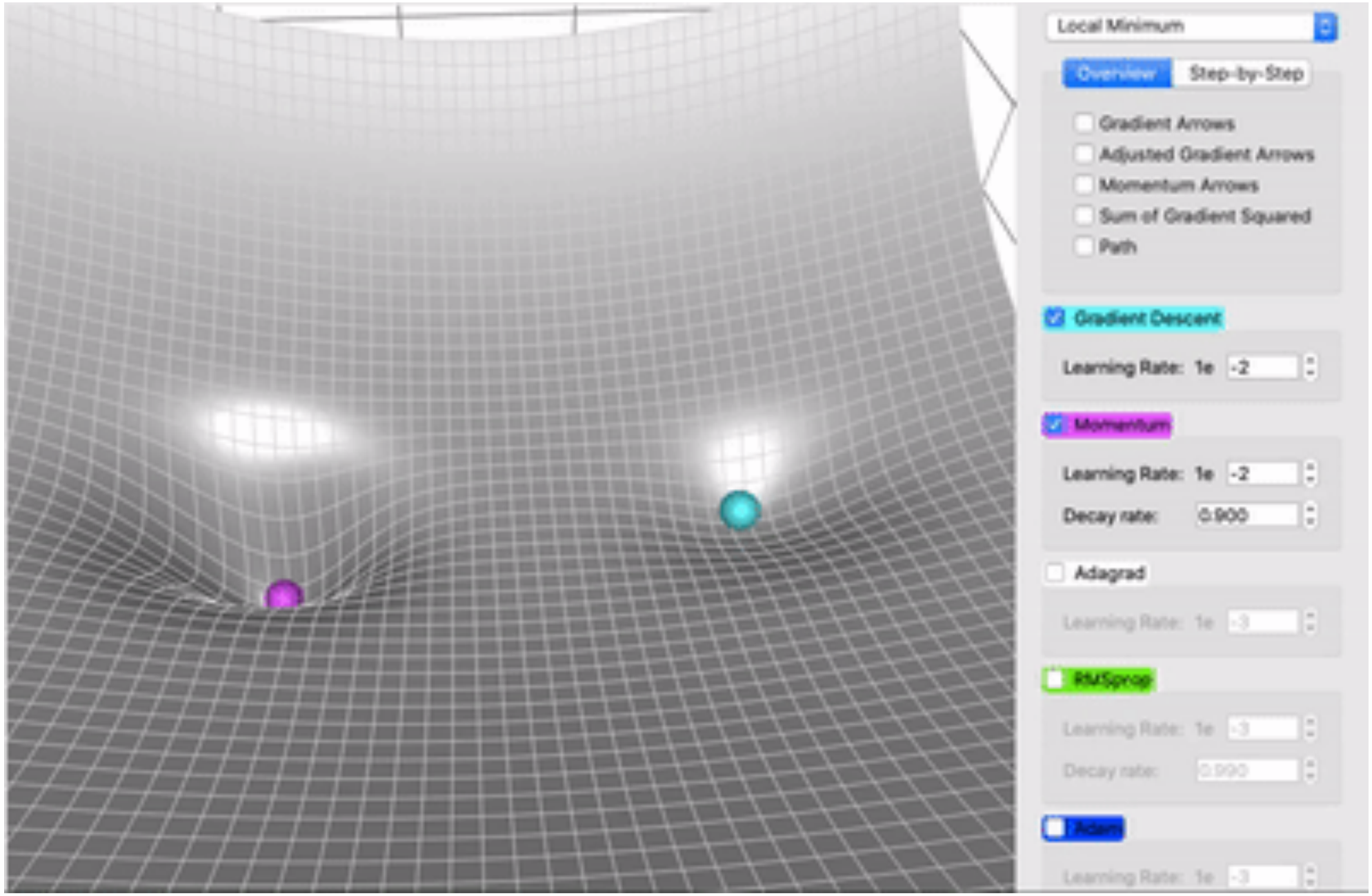
- Add a momentum/velocity that averages the general directions in parameter space

$$v^{(s)} = \alpha v^{(s-1)} - \eta \nabla_{\omega^{(s)}} L$$

$$\omega^{(s+1)} = \omega^{(s)} + v^{(s)}$$

➡ The basis for most common optimisers that are in use

Momentum in action



The above and many more details (great page)
https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

A Word on Normalisation

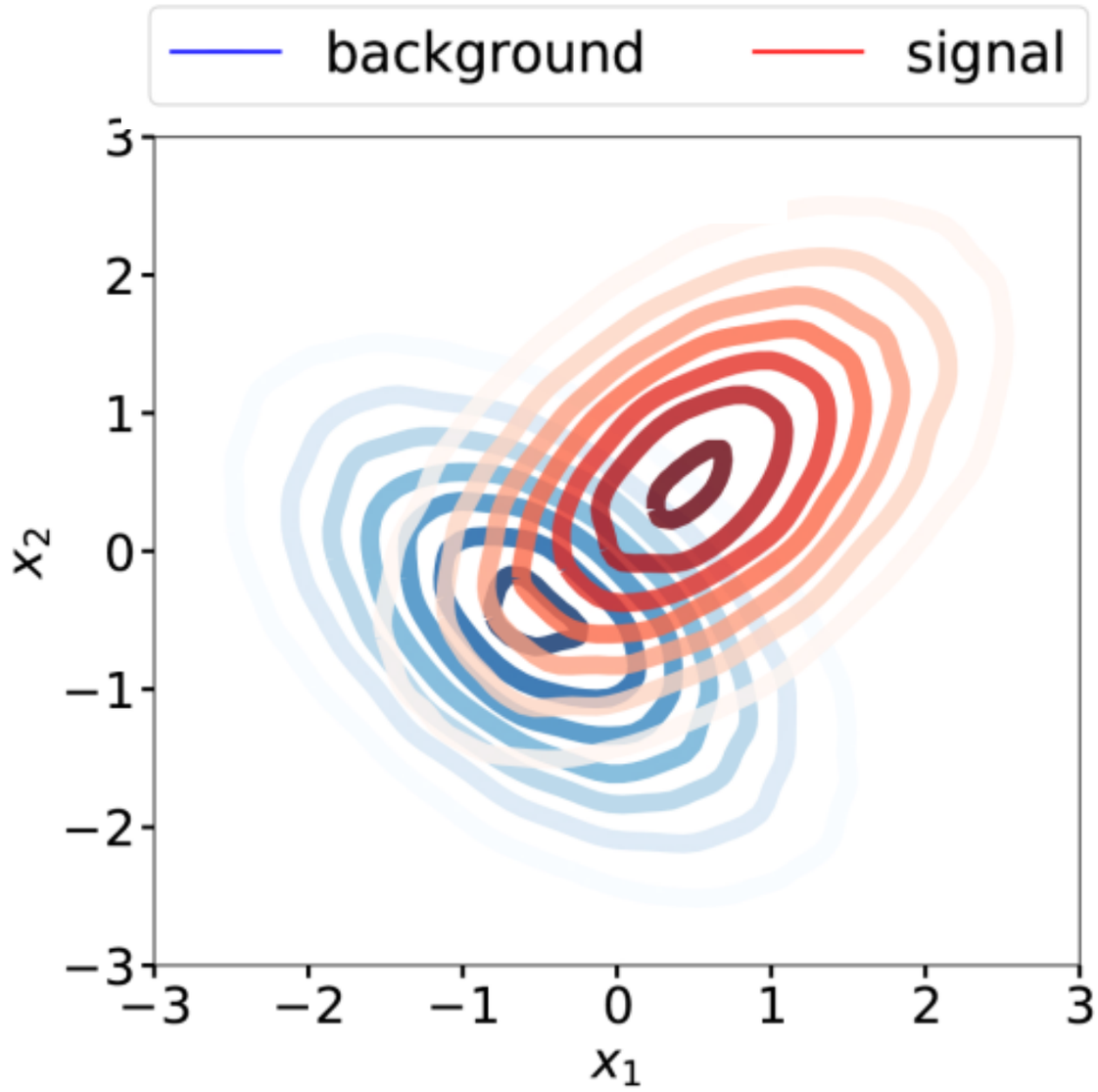
- Gradient-based optimisation is more efficient when continuous inputs (and, where useful, targets) have comparable numerical scales

- **Inputs** should be *approximately* normalised
(close to mean = 0 and variance = 1) → **needs to be done by the user**

- **Initial weights within the neural network** together with the **activation functions** should preserve that normalisation to avoid vanishing or exploding gradients

→ typically sufficiently taken care of by **default choices for initialisations** for typical activation functions

Classification



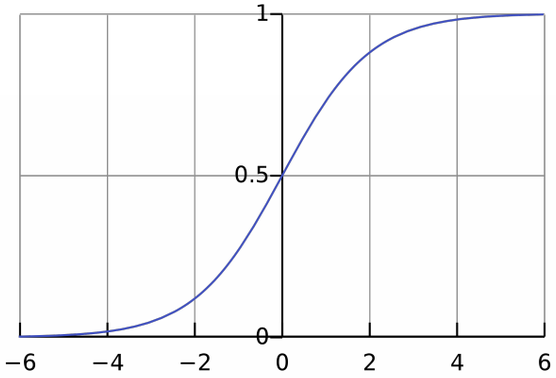
- The target is not continuous. It is either background or signal
- Encode as signal: $y = 1$, background $y = 0$

Classification loss: Binary Cross-Entropy

- $\hat{y} = \Phi(x; \omega)$ should be **probability for a single sample to be signal** (Bernoulli distribution)
  $p^k(1-p)^{1-k} \rightarrow \hat{y}^y(1-\hat{y})^{1-y}$

- Map output of Φ to 0-1 $\rightarrow$ output activation: **sigmoid**

Sigmoid



$1/(1+e^{-x})$

- The likelihood for N independent samples factorises:

$$\mathscr{L} = \Pi_{i=1}^{N} (\hat{y}^{(i)})^{y^{(i)}} (1-\hat{y}^{(i)})^{(1-y^{(i)})}$$

$$L_{BCE} = -1/N \log \mathscr{L} = -1/N \sum_{i}^{N} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1-y^{(i)}) \log(1-\hat{y}^{(i)}) \right)$$

- For multi-classification:
  - binary $\rightarrow$ categorical cross entropy (from multinomial distribution)
  - $y \rightarrow \vec{y}$, “one-hot”, $\hat{y} \rightarrow \vec{\hat{y}}$
  - Sigmoid $\rightarrow$ Softmax

And now?

Machine Learning

- Neural network $\Phi(x; \omega)$
- Labelled data: (x_i, y_i)
- Criterion: loss function L
- Optimiser to train ω

➡ Predict $\hat{y}_j$ for new x_j with unknown y_j

Assess NN model performance

- We have the ingredients
- When are we satisfied with the training? What makes a good model?

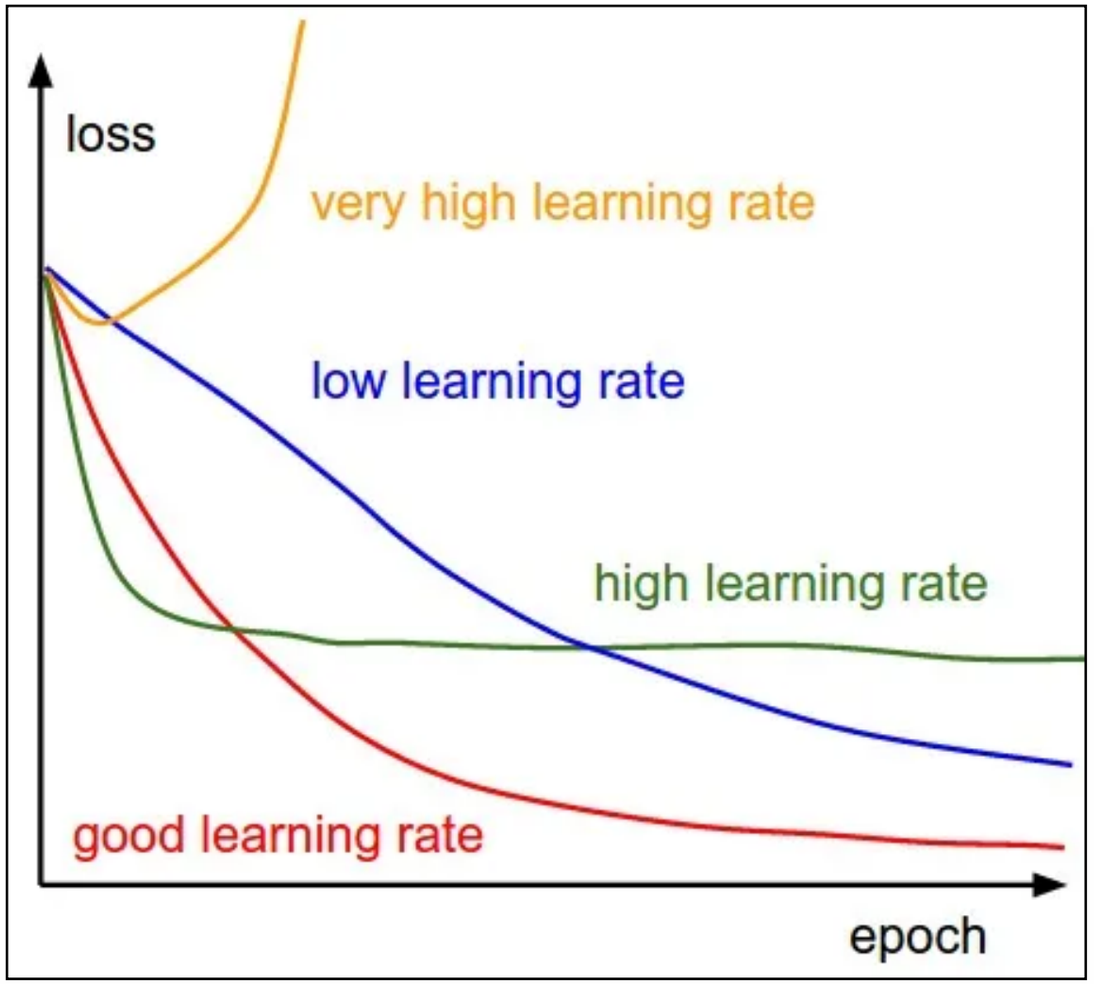
Training in Steps and Epochs

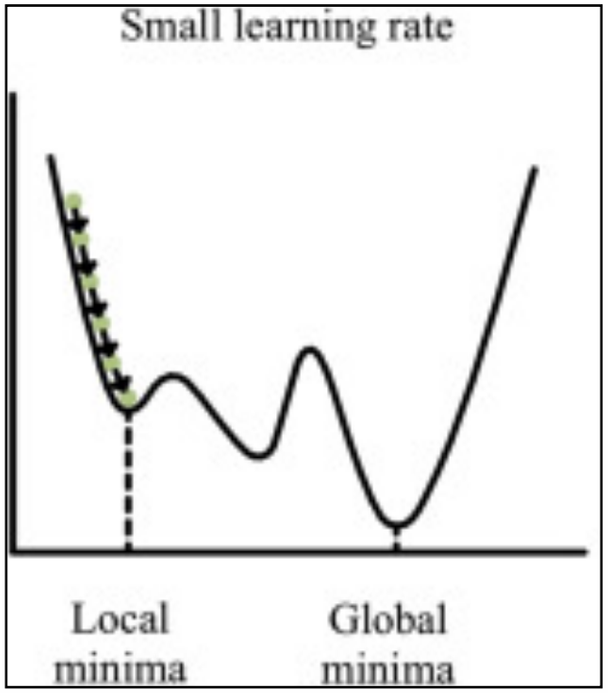
SGD: $\omega^{(s+1)} = \omega^{(s)} - \eta \nabla_{\omega^{(s)}} L\left(\Phi(x; \omega^{(s)}), y\right)$

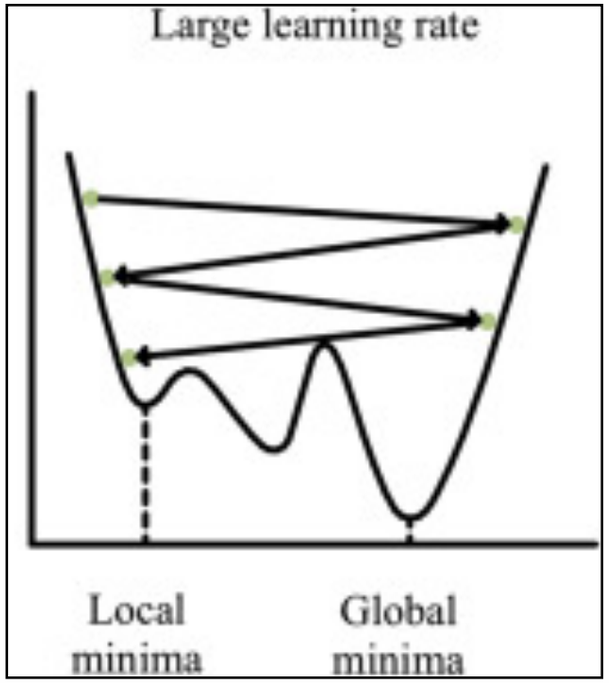
- Training once on the dataset is almost never enough

```
loss_fn   = torch.nn.MSELoss()
optimiser = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(n_epochs):

    for x, y in train_loader:          # one mini-batch

        optimiser.zero_grad()

        y_hat = model(x)               # forward pass
        loss = loss_fn(y_hat, y)

        loss.backward()                # calculate gradients
        optimiser.step()               # update parameters
```

Learning rates







- There is no universally best learning rate - always needs to be adjusted

Learning from the data: Dataset Bias









- Who thinks this is a dog?

Learning from the data: Dataset Bias








- Who thinks this is a dog?
- **The dataset has to be representative of the problem**

Learning from the data: Overfitting









- Who thinks this is a dog?

Learning from the data: Overfitting

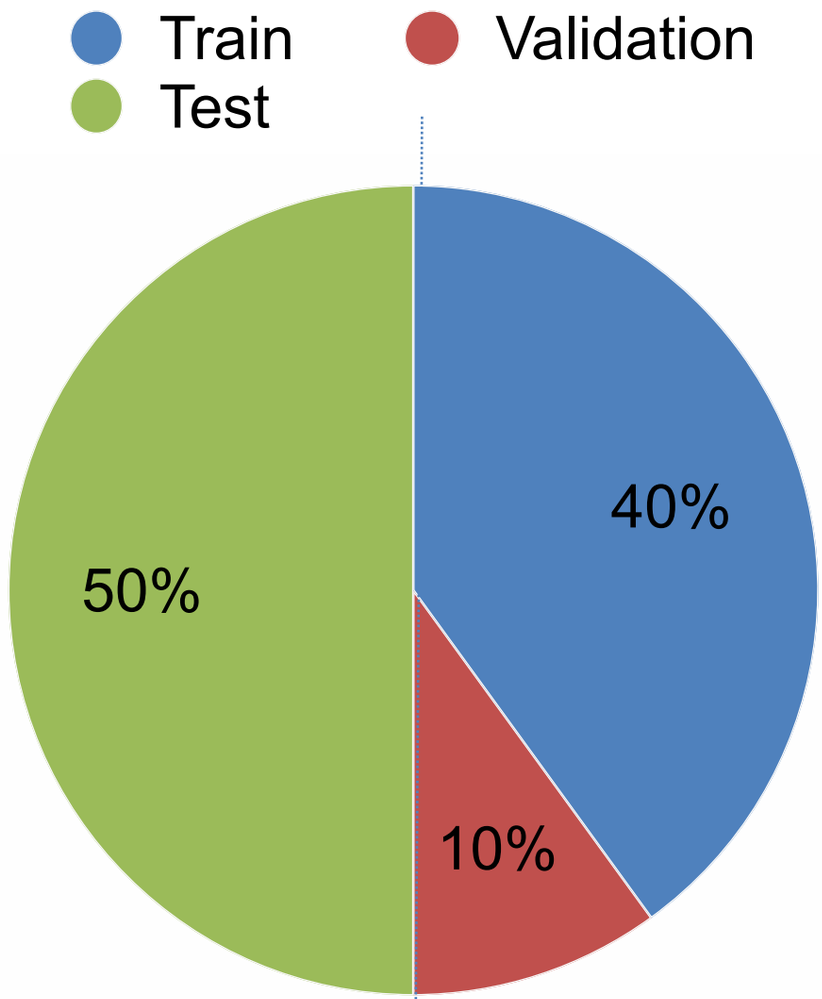







- Who thinks this is a dog??
- We learned to identify specific features of *individual samples* → **overfitting**

The Dataset Split

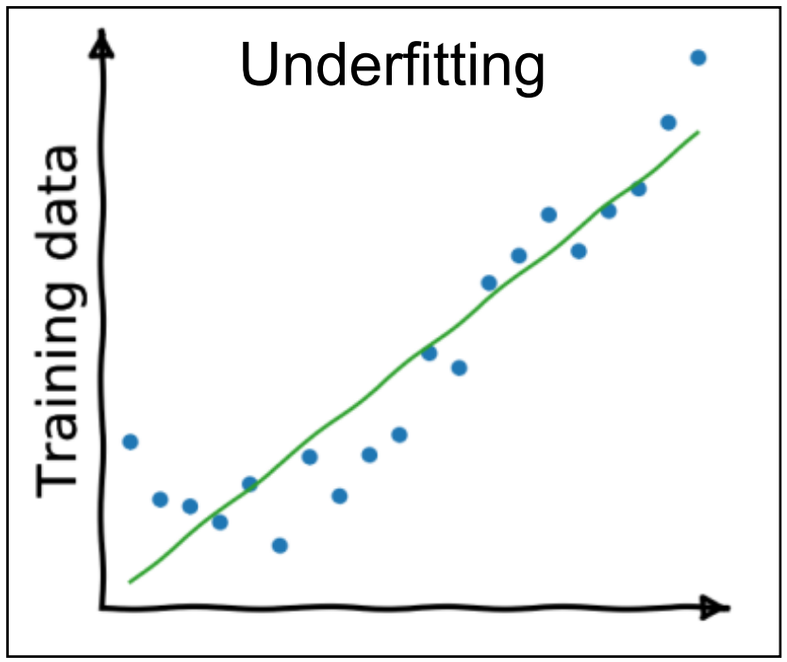


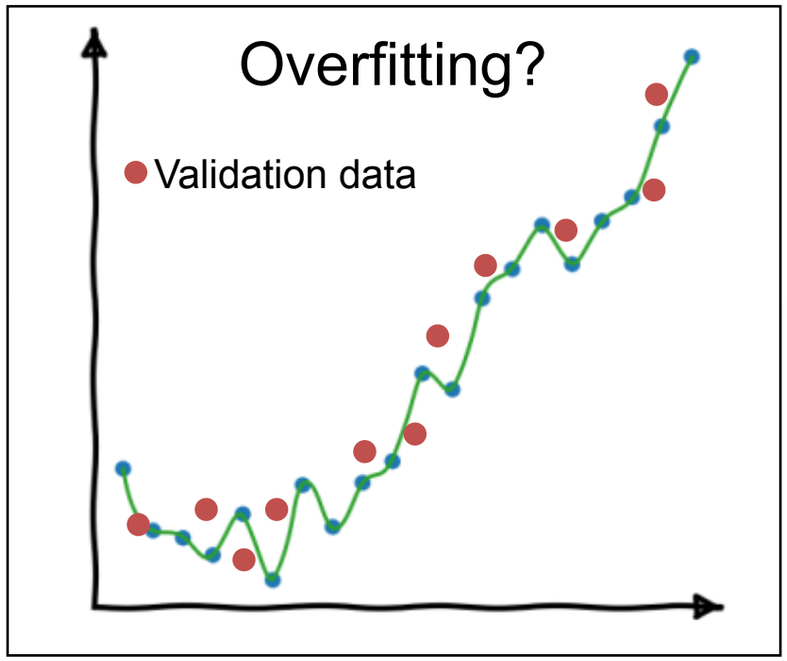
Used to train the weights

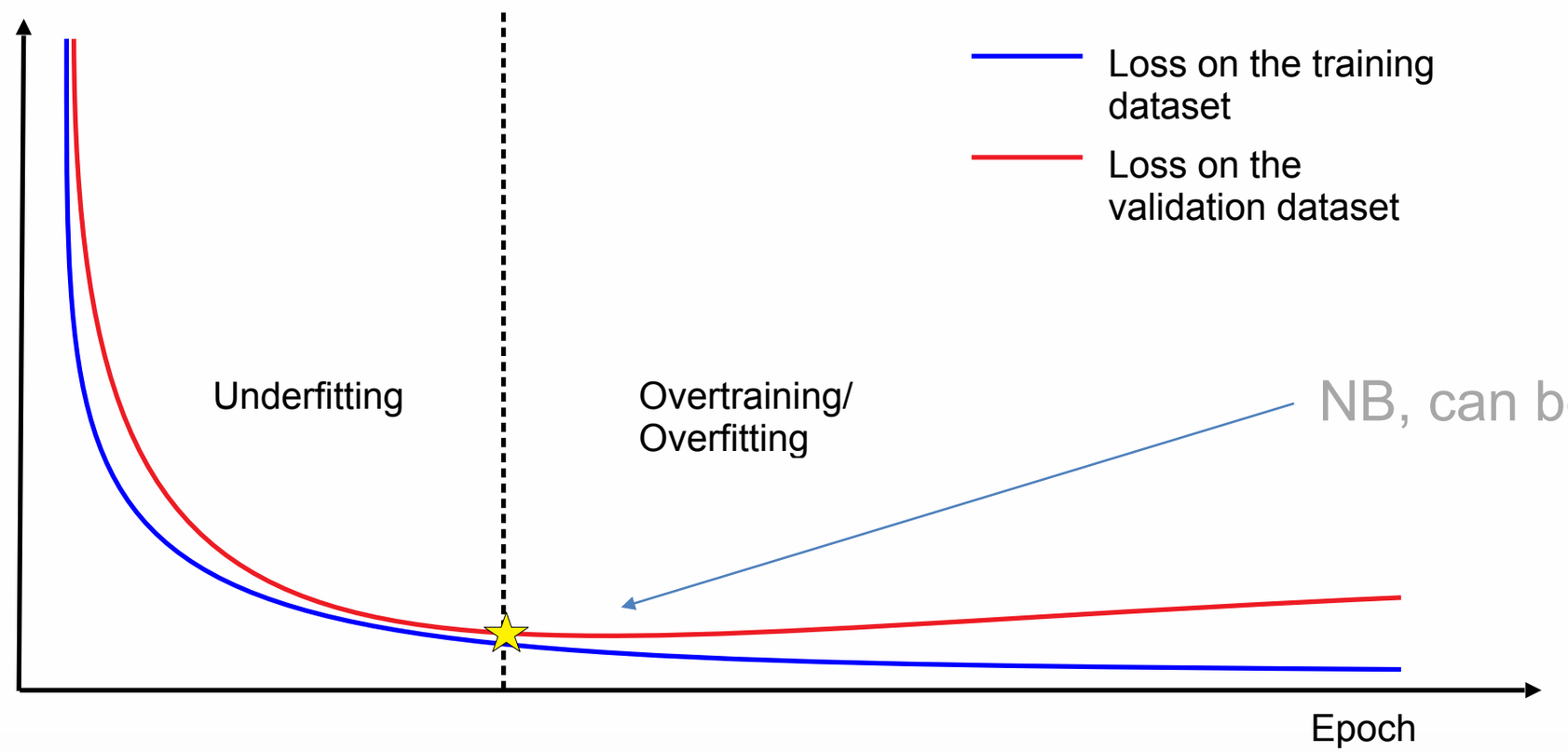
Must not influence the model at all

Used to decide hyper parameters

Identifying Overfitting





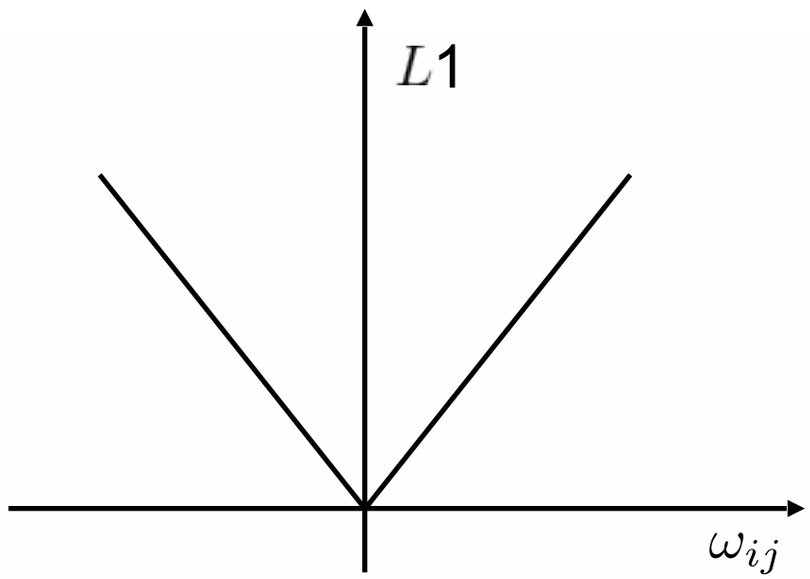


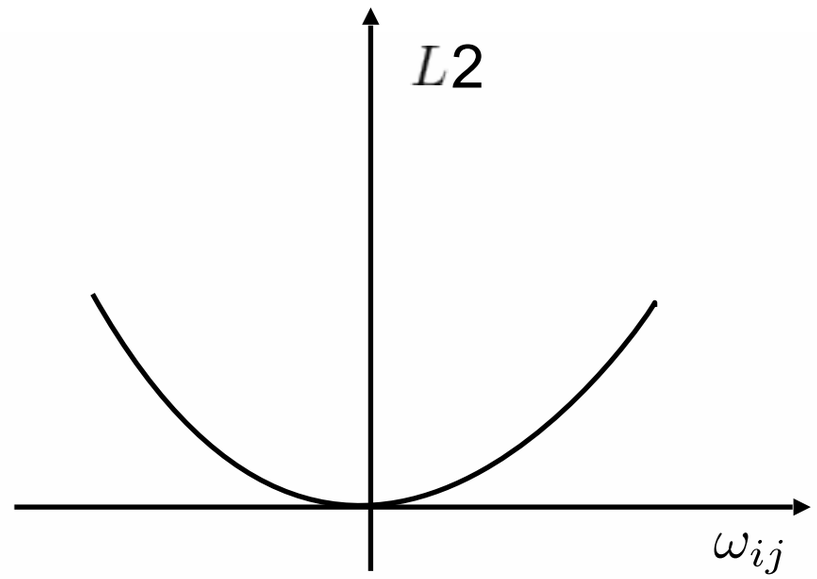
NB, can be used for *early stopping*

Prevent Overfitting

➡Use/create more data!

- L1/L2 regularisation





- **Erases** single weights, creates sparse models

  NB: this can be useful e.g. when speeding up models

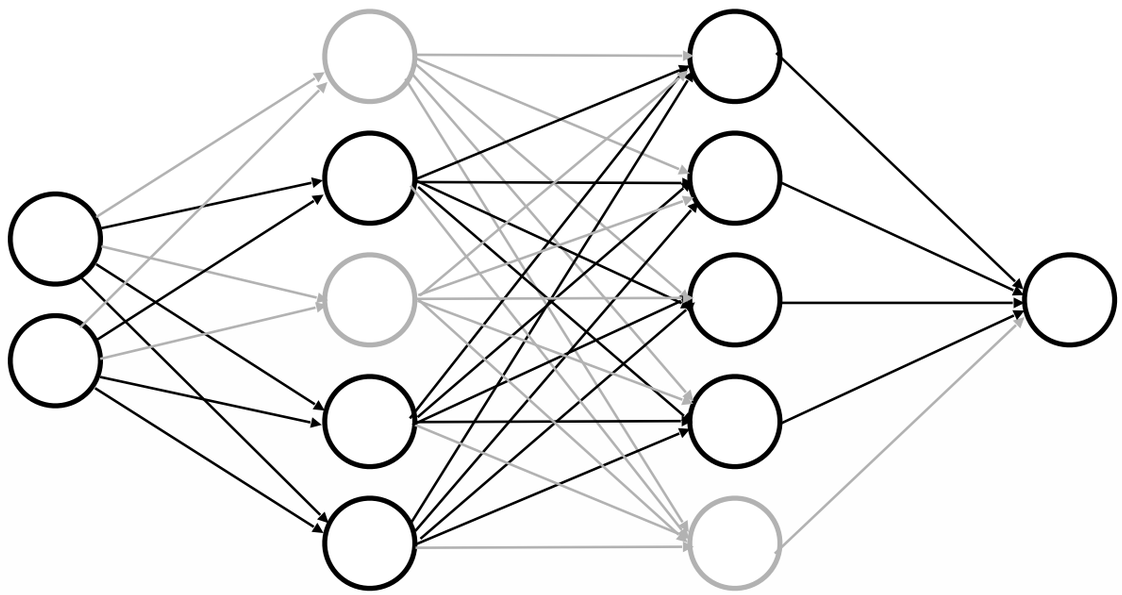
- **Reduces** contributions from individual weights.

$$L \rightarrow L + \lambda L_{\text{reg}}$$

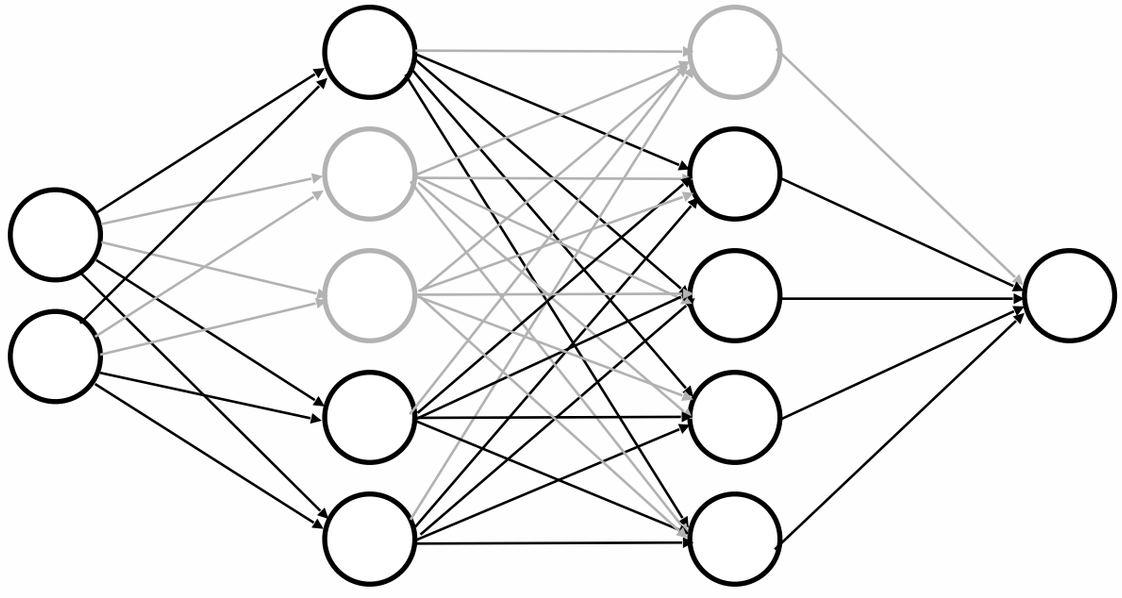
- Tune λ carefully using validation data
- **If you can, use/create more data**

Dropout

Train step k



Train step $k + 1$



- Mask different units each step
- Model cannot (over)focus on individual weights
- **If you can, use/create more data**

Usual choices of dropout probabilities are

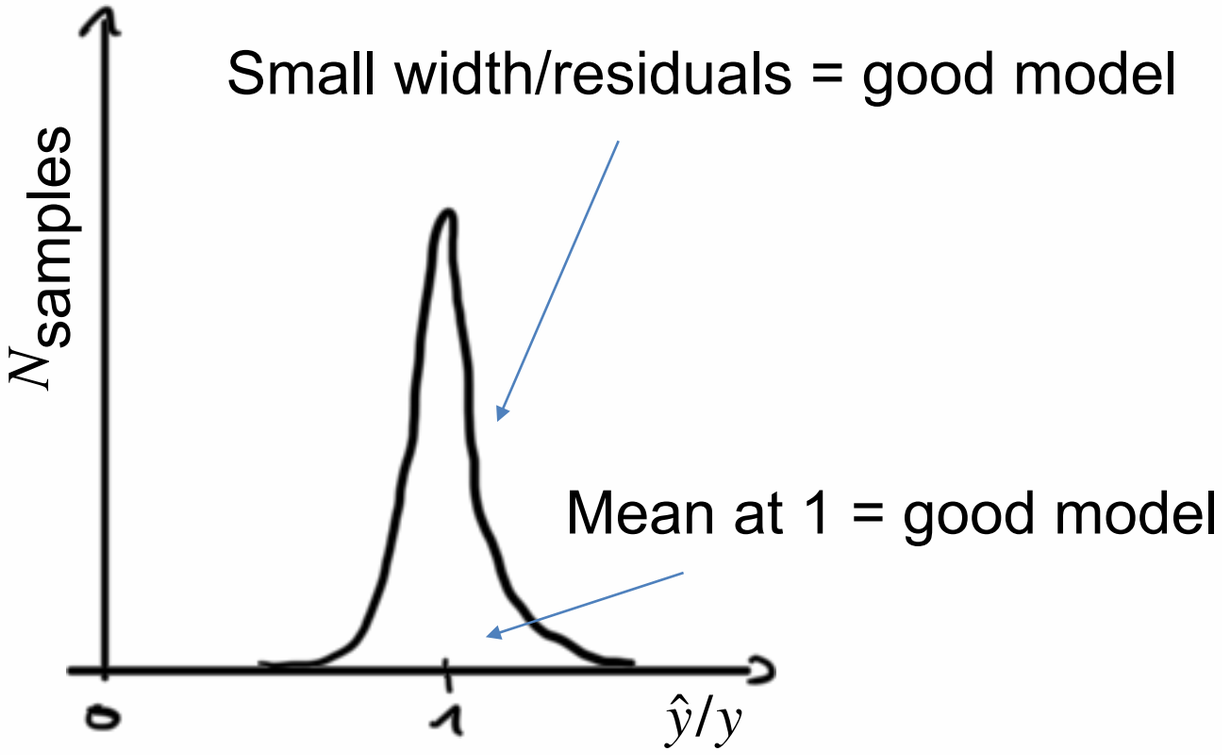
- $d = 0.3 \ldots 0.5$

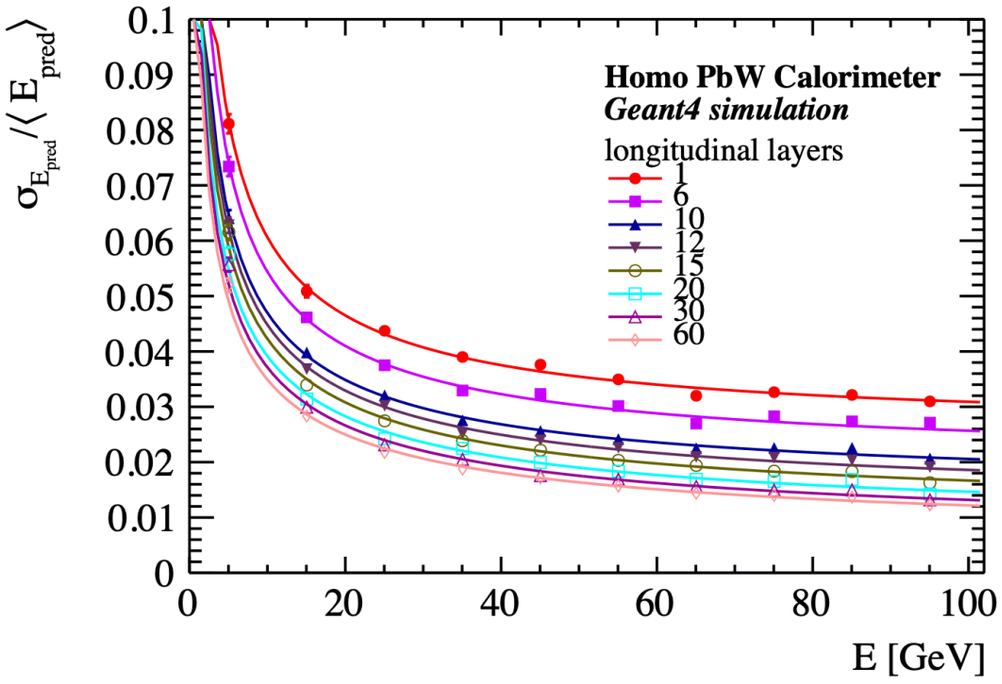
Is it a Good Model?

Regression Metrics

- For regression tasks, it depends a lot on the regression target which metrics can help us estimate the DNN performance.

- Often, we are interested in the response of the network: $\hat{y}/y$



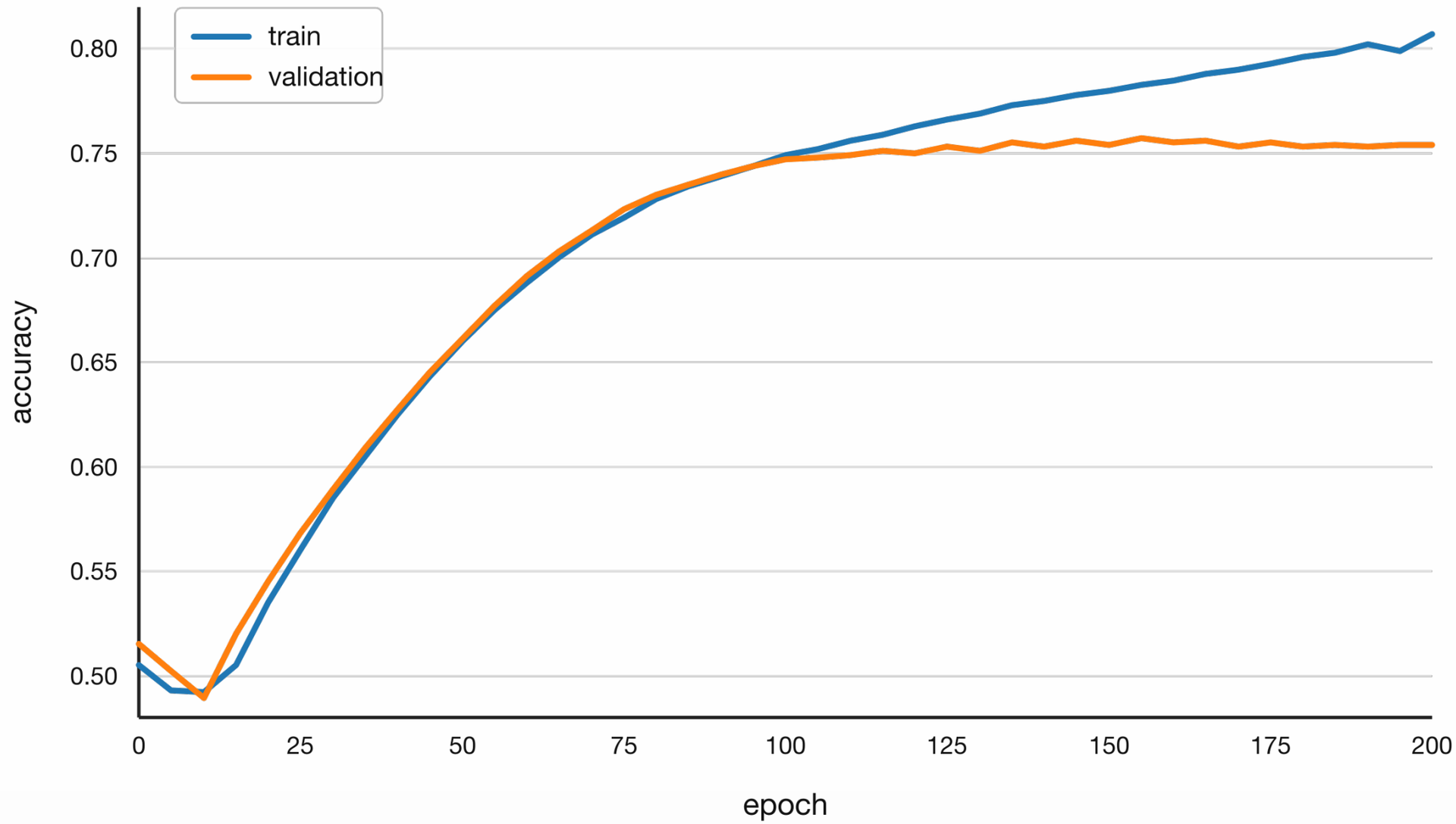


C. Neubüser, et al., arXiv:2101.08150

- This can differ across relevant observables - checking only the inclusive distribution is often not enough

Classification Metrics: Accuracy

- $\hat{y}$ (the prediction) is a continuous score between 0 and 1
- y (the truth) is discrete and either 0 or 1
- We need a cut-off (working point) for matching, e.g. 0.5



Continuous View: ROC curves

$$t_P = \frac{T_P}{\Sigma\, y}, \quad f_P = \frac{F_P}{\Sigma\, (1-y)}$$

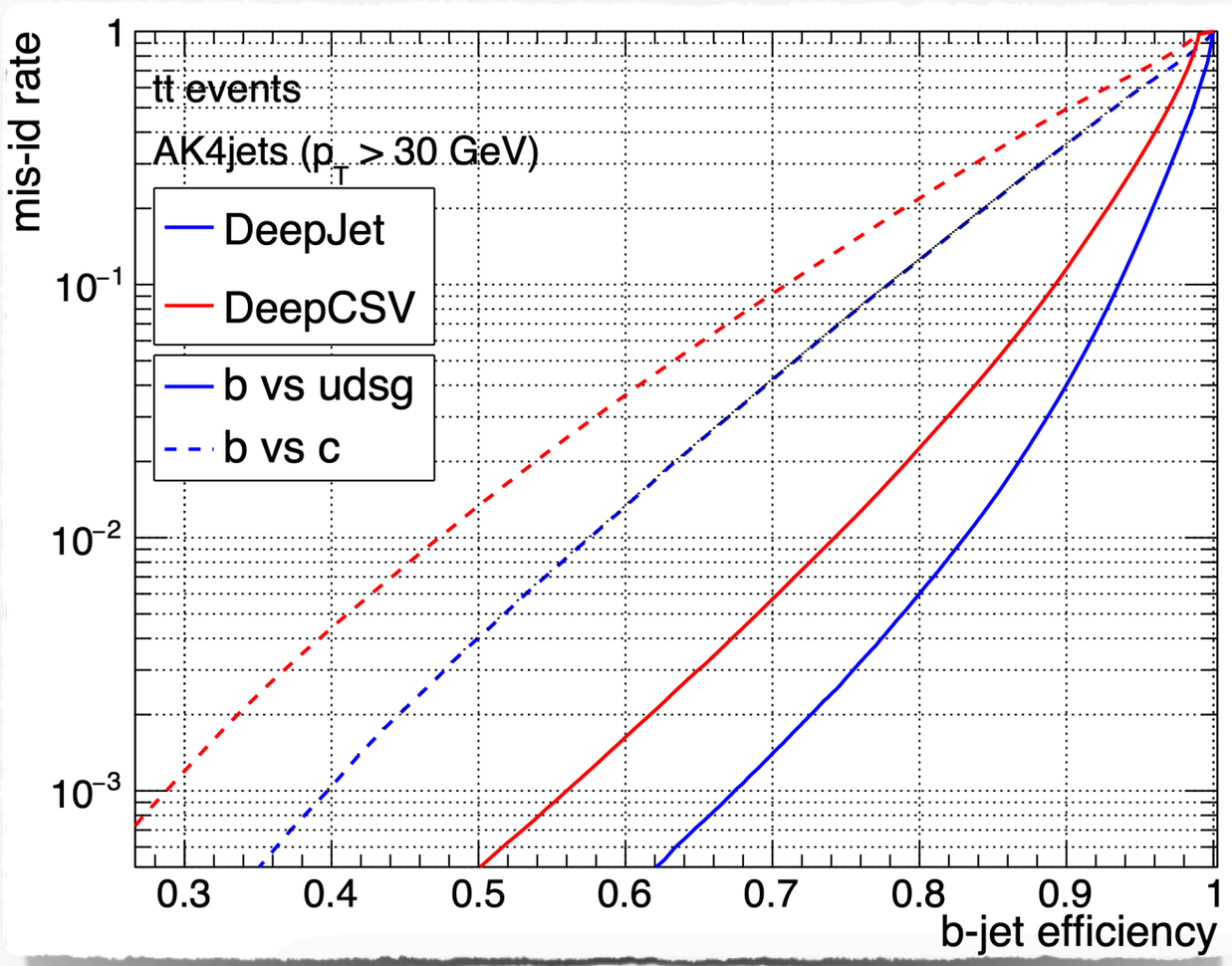
Explicit dependence on score threshold t

$$t_{P(t)} = \frac{T_{P(t)}}{\Sigma\, y}, \quad f_{P(t)} = \frac{F_{P(t)}}{\Sigma\, (1-y)}$$

- Often, we also refer to **efficiency** and **fake rate,** or **misidentification rate**

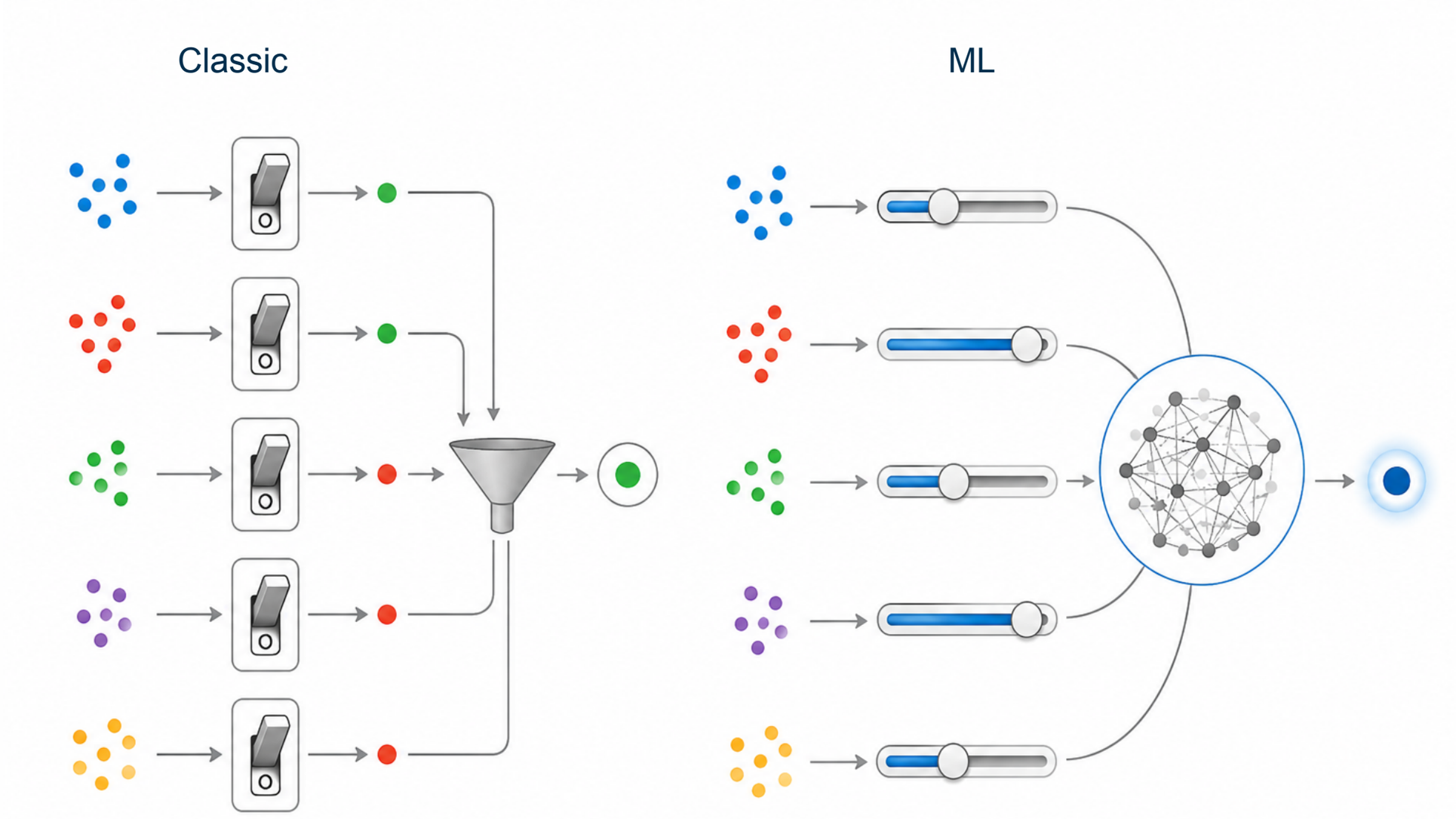
- **Efficiency:** t_P
- **Fake/misidentification rate:** f_P

- Which algorithm is better?



E. Bols, J.K, et al. , arXiv:2008.10519

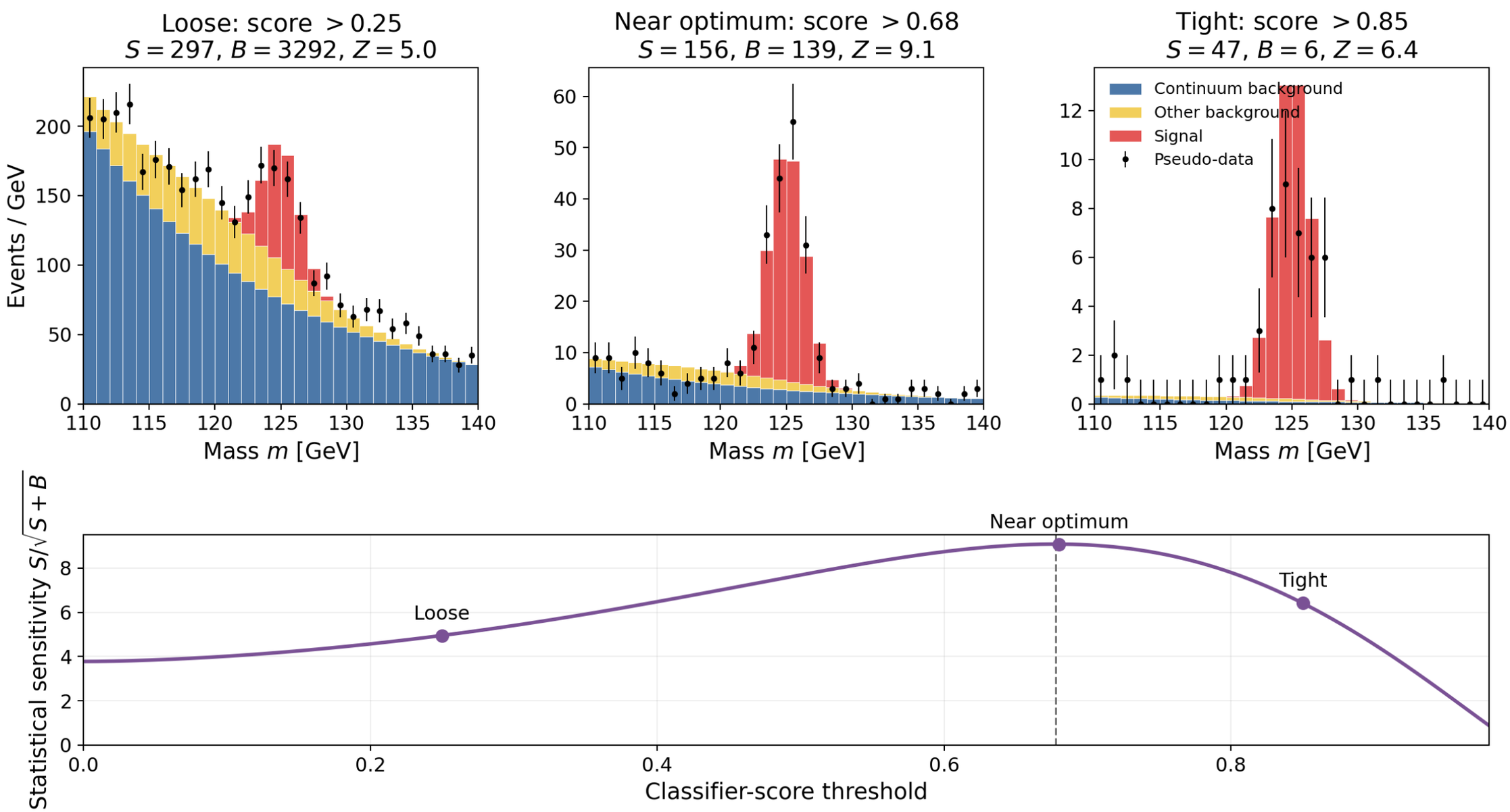
Why ML (1)?



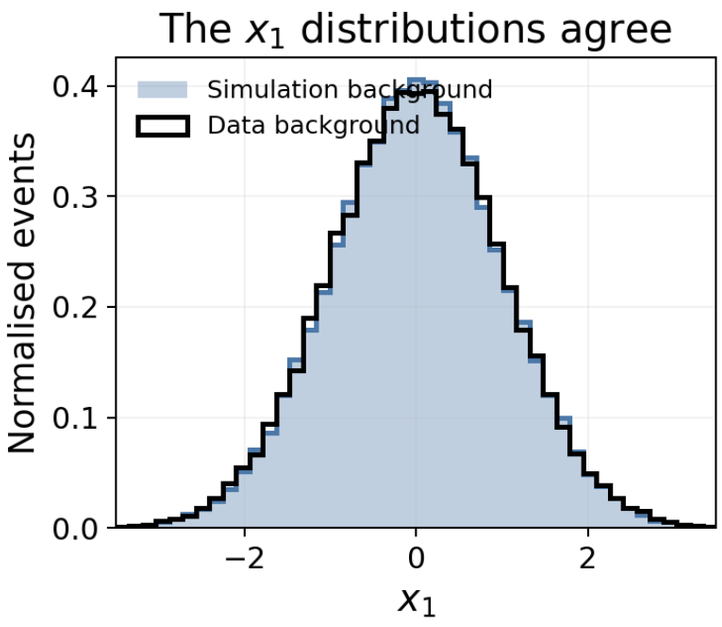
- Can exploit **non-trivial dependencies** of many input variables
- Well designed and trained algorithms typically outperform classic algorithms w.r.t. physics performance

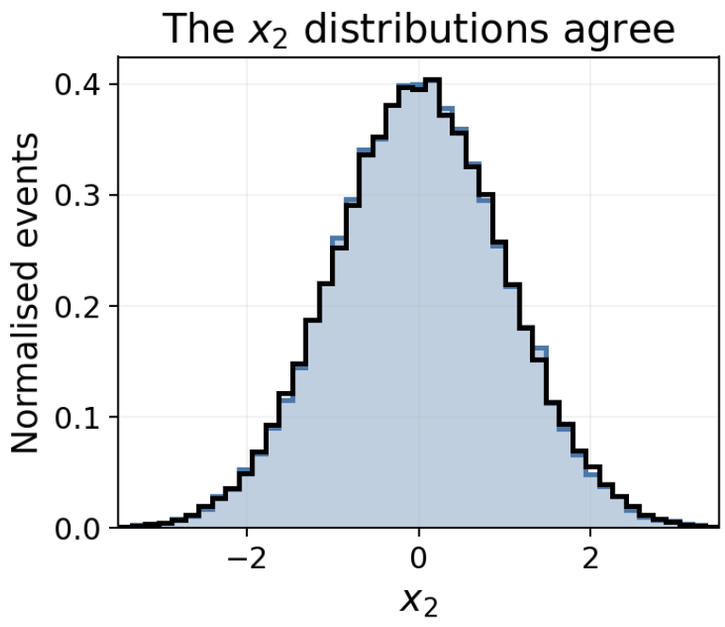
Physics Sensitivity decides the Choice of Working Points

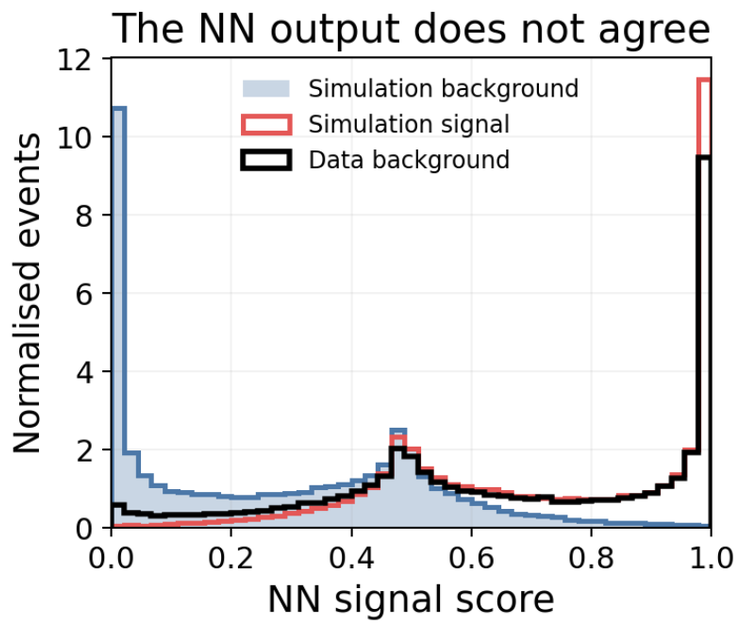
Mock analysis



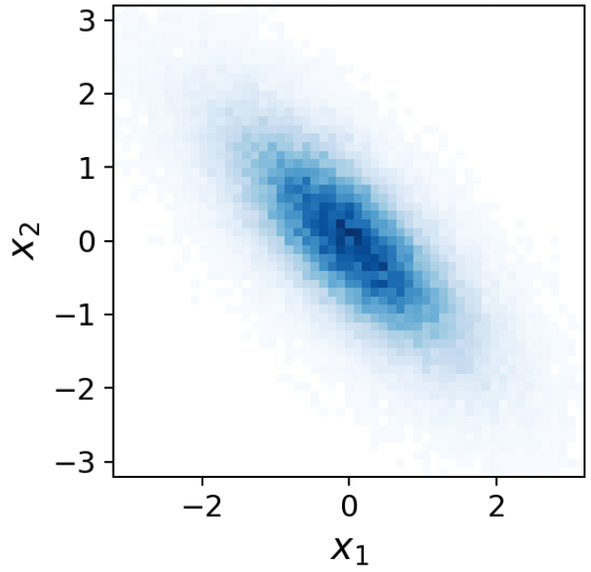
Validation is Key: Also Correlations Matter



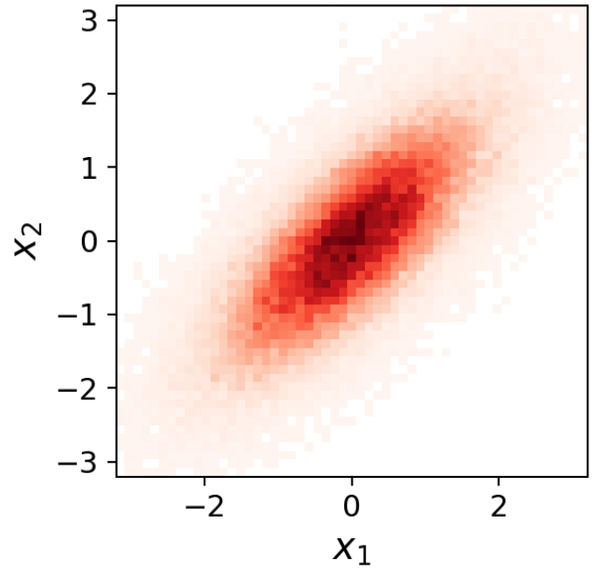




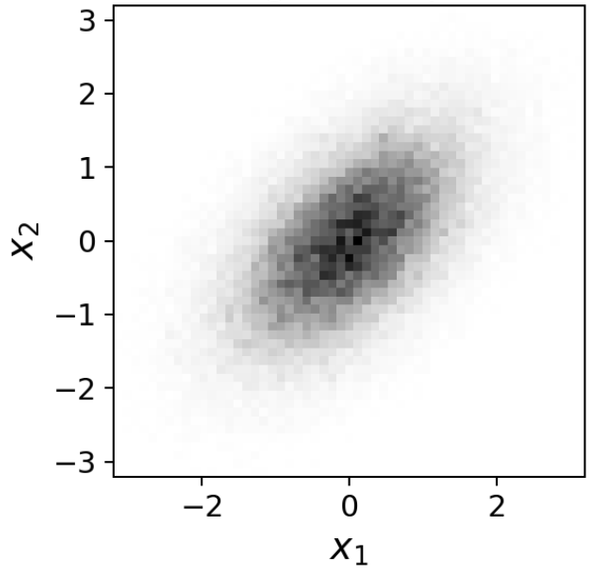
Simulation background; $\rho = -0.75$



Simulation signal; $\rho = +0.75$



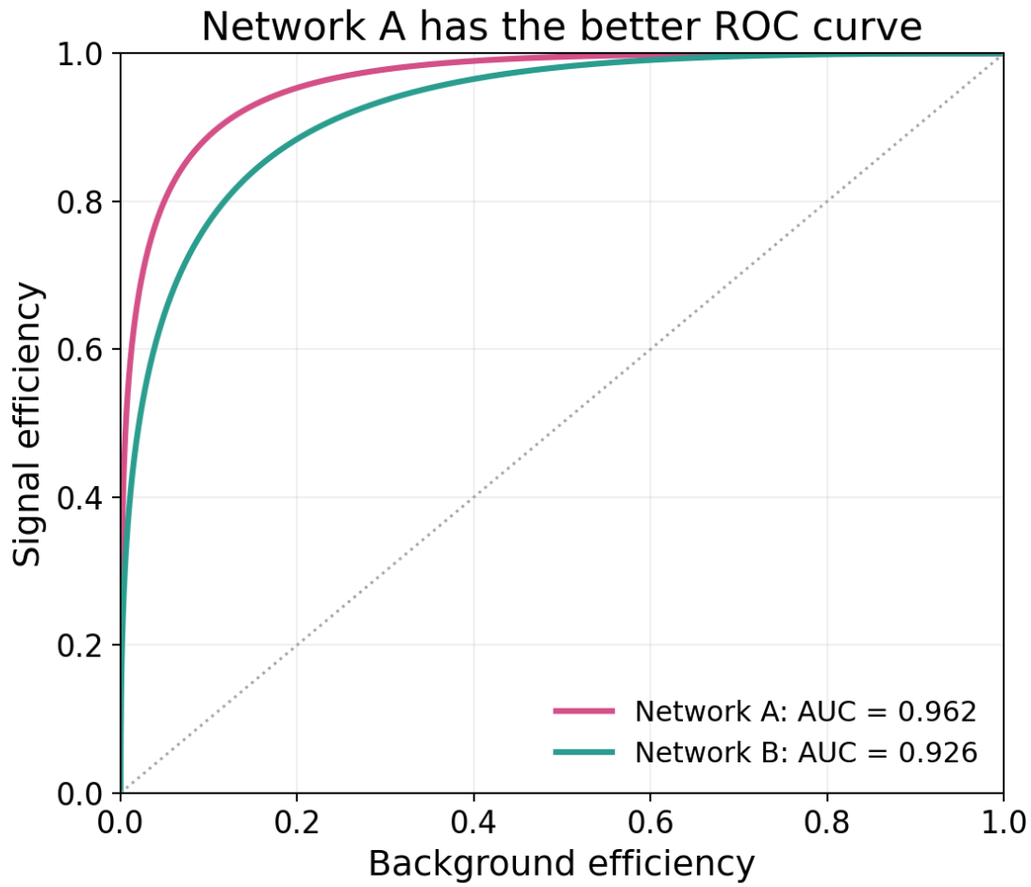
Data background; $\rho = +0.55$

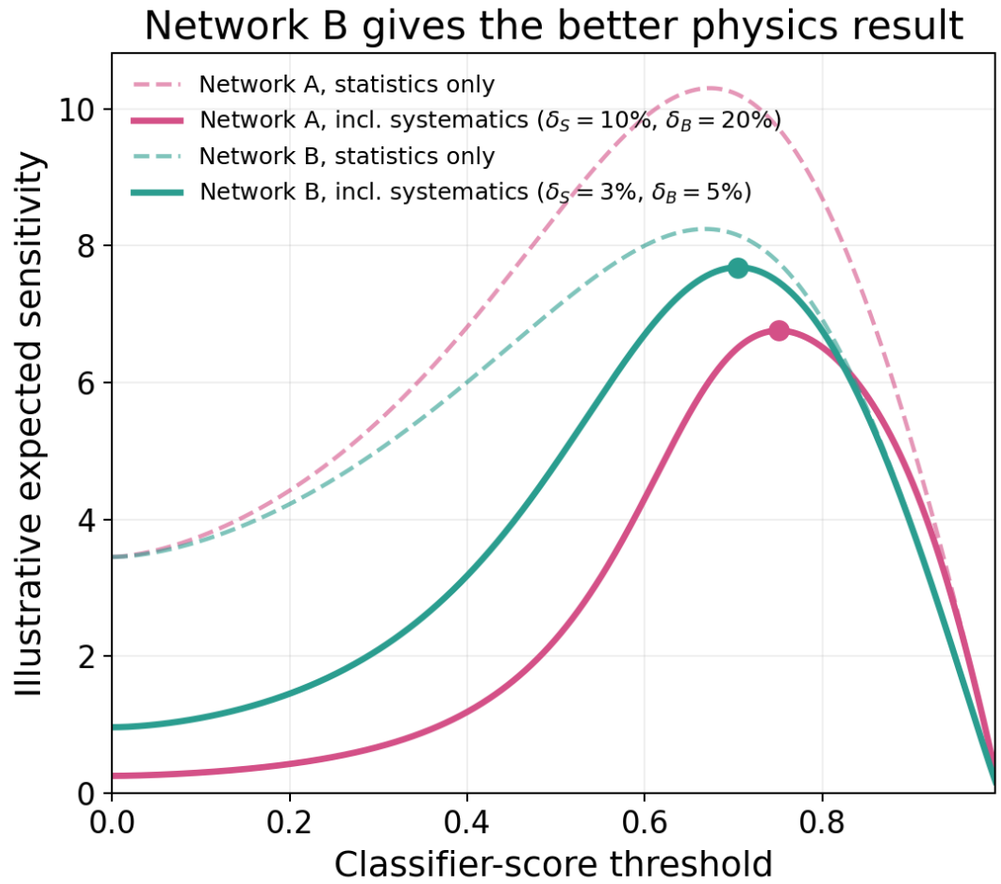


- Validate or calibrate the model output in data wherever possible; otherwise validate the inputs and their correlations in suitable control regions.

The ML Metric is not the Physics Objective

- A model with better statistical performance can still give a worse physics result once modelling and calibration uncertainties are included.

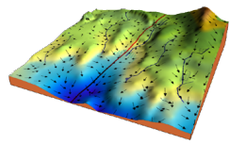


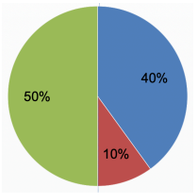


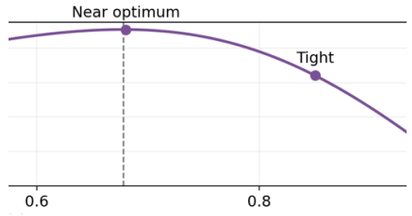
Illustrative proxy: $Z = S/\sqrt{S + B + (\delta_S S)^2 + (\delta_B B)^2}$

DIY Conclusions

$\Phi(x;\omega)$ $L(\hat{y},y)$







- Neural networks are universal function approximators

- The loss quantifies the discrepancy of the truth value and the NN output; often comes from a likelihood

- If validation loss increases over training loss, we overfit; prevent by using more data ;)

- We find the minimum with gradient descent
- The learning rate is an important hyper parameter

- Physics performance decides what the best model is, not just ROC curves etc.

- The dataset should be representative of the problem