ERUM-DATA-HUB & DIG-UM DEEP LEARNING SCHOOL “BASIC CONCEPTS”

How to fail Mastering Model Building many times and eventually succeed

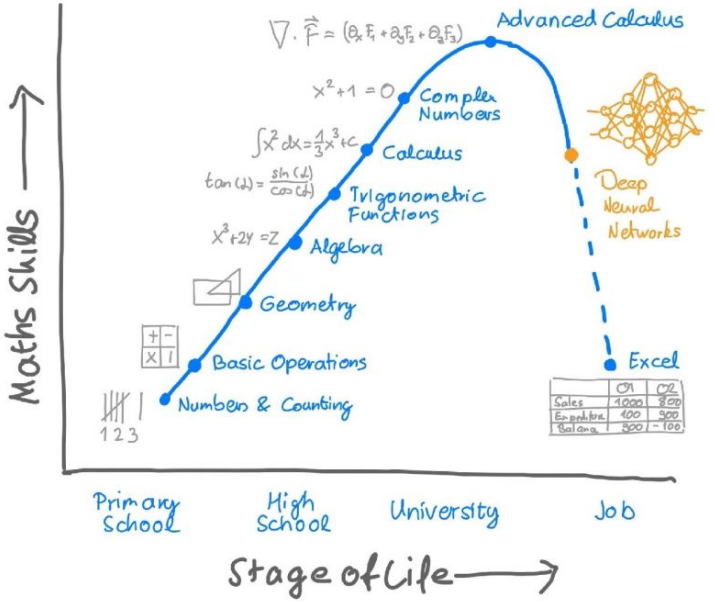
Maximilian Horzela, **Markus Pirke**

shamelessly stolen from
Much of the material ~~heavily inspired by~~ Marcel Rieger, ChatGPT, "Deep learning in Physics Research", Erdmann *et al.*

Complexity of Artificial Neural Networks (ANNs)

- NNs are not complicated
- But complex

- Well suited for complex tasks with multivariate dependencies
- Most challenges with NN models related to complexity
    A. Computational complexity
    B. Problem complexity

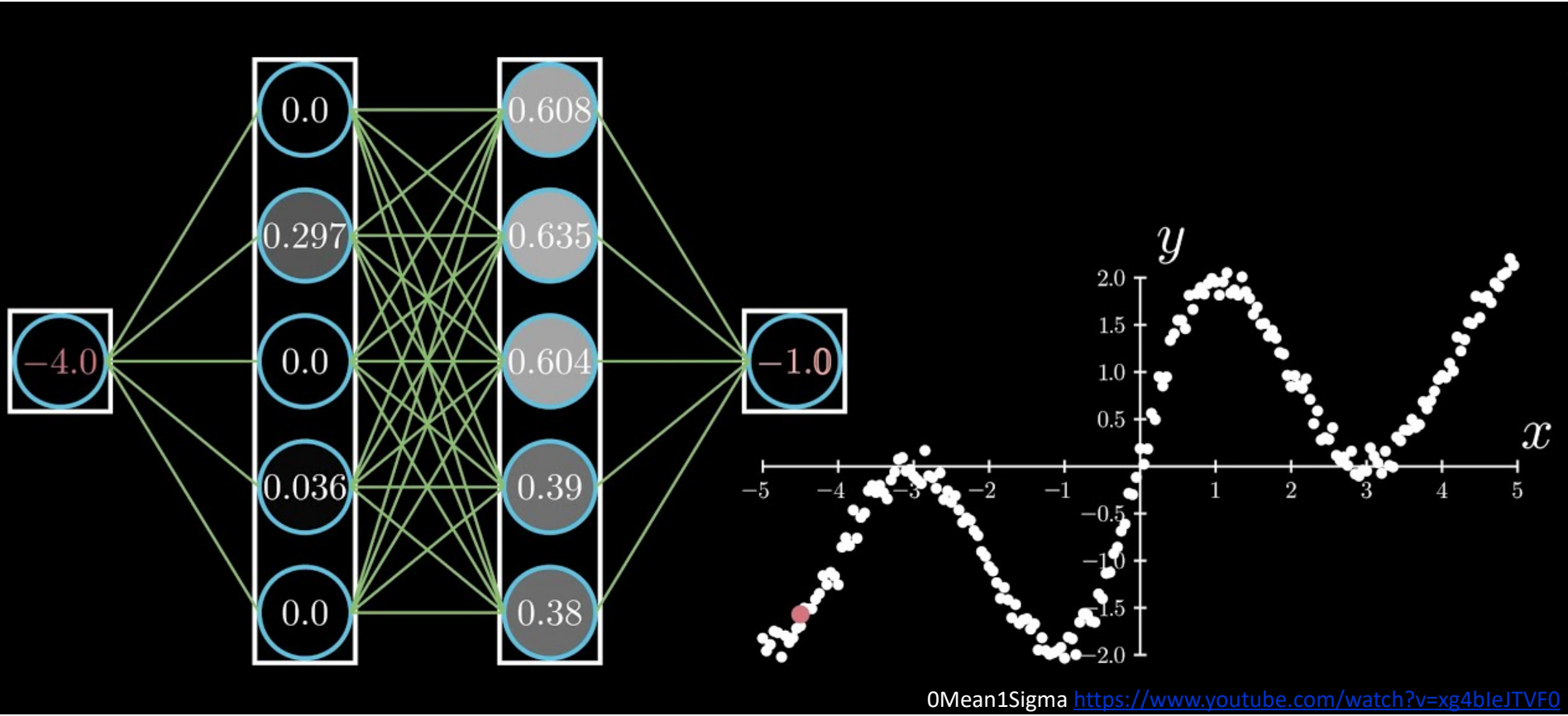


Who are we to tell you?

- **Maximilian**
  hysicists, not
  Uni Göttingen/CERN
  sts, with a mix of
  rcs
  eling
  nanagement
- are really the
  for coping with

Markus

FAU Erlangen-Nürnberg

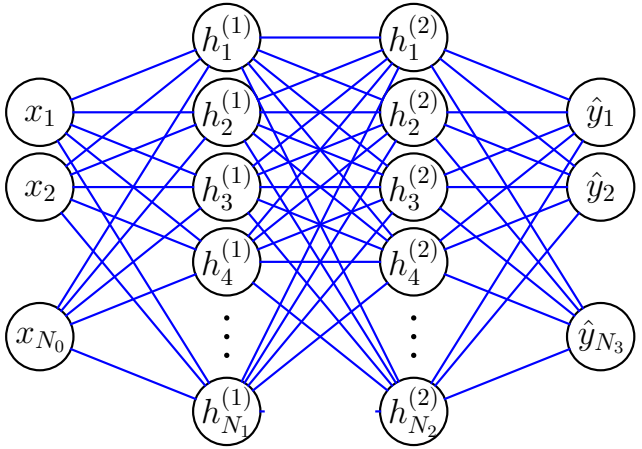


0Mean1Sigma https://www.youtube.com/watch?v=xg4bleJTVF0

What are NNs practically?

A. NNs are directed weighted graphs of neurons that perform a mapping
   - Tensors model the network of nodes and edges
   - Tensor operations define mathematical dependencies between tensors

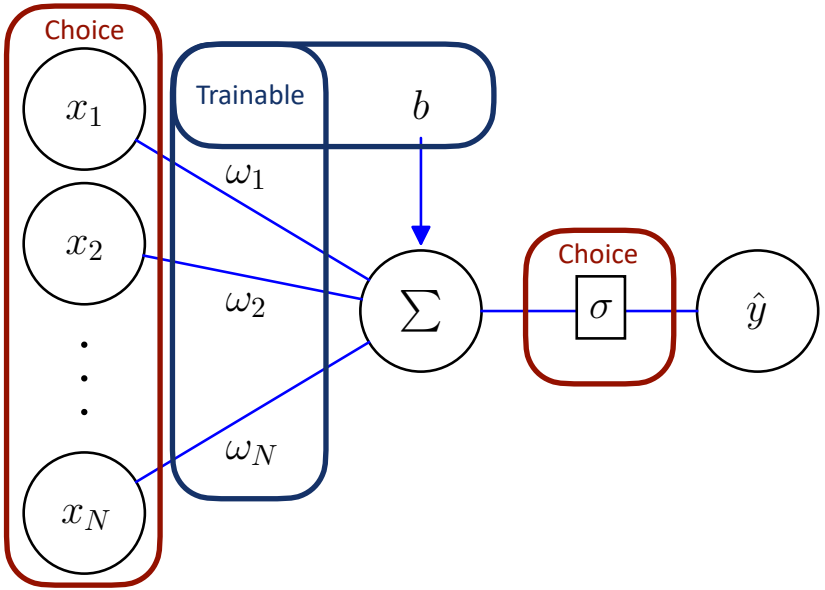
B. NNs are typically multiple layers of neurons, each layer chosen for a (specific) purpose



Perceptron

$$\hat{y} = \sigma\left(W\vec{x} + b\right)$$

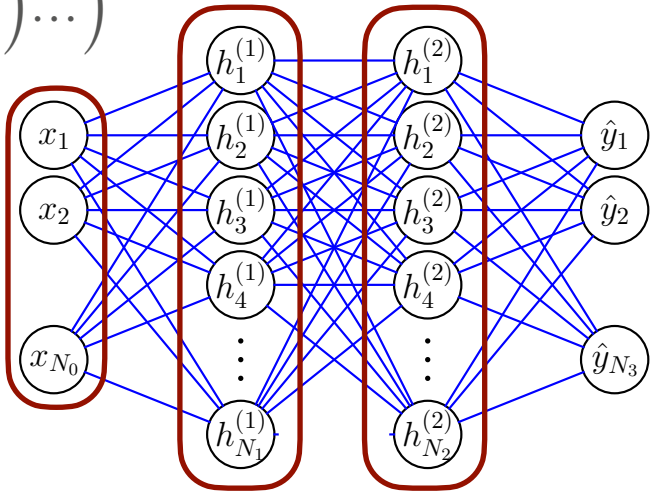
- Inputs $\vec{x}$ Choice
- Weights W Trainable
- Bias b
- Activation Function σ
- **Output** $\hat{y}$



(Deep) Neural Networks

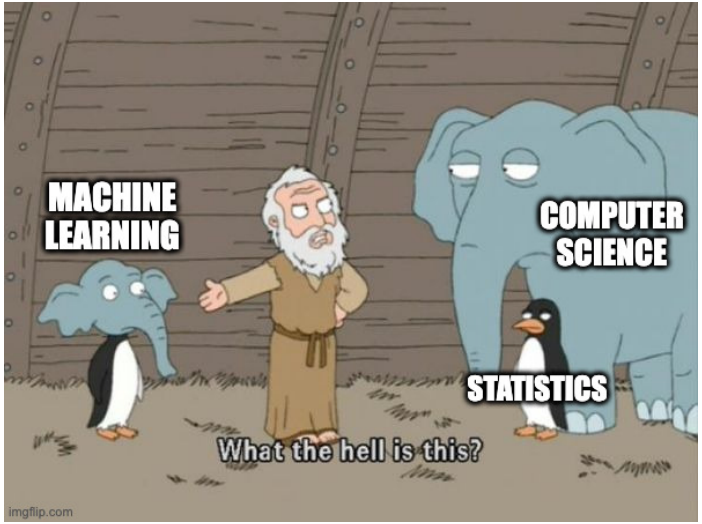
$$\vec{\hat{y}} = \ldots\vec{\sigma}_{(2)}\left(W_{(2)}\vec{\sigma}_{(1)}\left(W_{(1)}\vec{\sigma}_{(0)}\left(W_{(0)}\vec{x} + b_{(0)}\right) + b_{(1)}\right)\ldots\right)$$

- Inputs $\vec{x}$ Choice
- Weight matrices $W_{(i)}$ Trainable
- Biases $b_{(i)}$
- Hidden layers $\vec{h}^{(i)} = \vec{\sigma}_{(i-1)}\left(W_{(i-1)}\vec{h}_{(i-1)} + \vec{b}_{(i-1)}\right)$
- Activation Functions $\vec{\sigma}_{(i)}$
- **Outputs** $\vec{\hat{y}}$

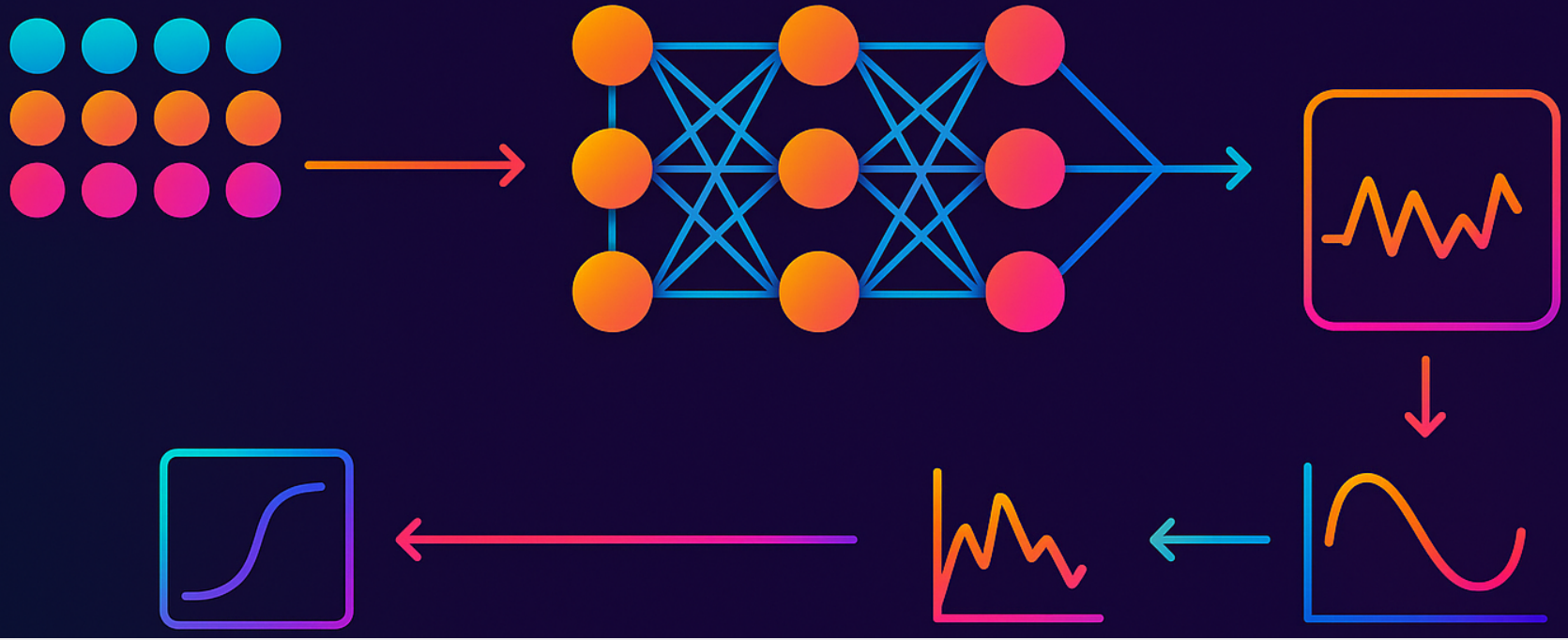


Machine Learning = NNs

- Optimization of model parameters with respect to an objective
- Challenge: Many free parameters for a statistical fit $N(\text{param}) \gtrsim N(\text{data})$
- Solution:
  - Exploit analytical differentiability at high granularity of NNs
  - Efficient numerical optimization techniques

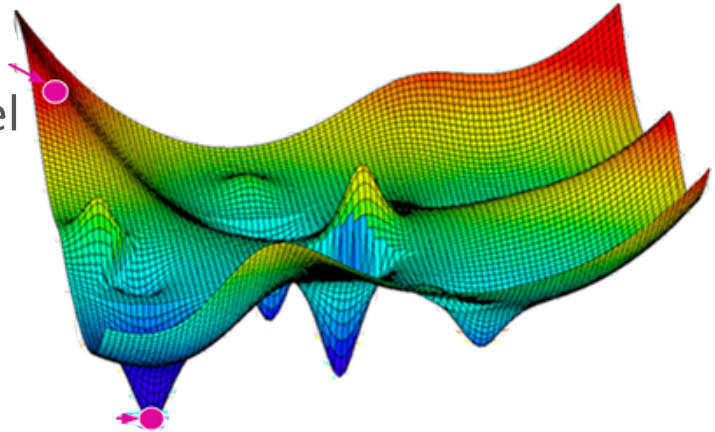


Building Blocks, Again, but with More Detail



Objective Functions

- Define your model's task and how it learns
  - Reinforcement Learning: Reward
  - Regression: Difference between model prediction and target
  - Classification: Metric of similarity or distance
  - …
- Often heavily inspired by traditional statistical methods



Regression

- Find the set of model parameters that leads to a prediction that is as close as possible to a target
  - Minimize the (absolute) difference between prediction and target
- Many differentiable candidates available

Mean Squared Error (MSE)	$\frac{1}{N}\sum_{i+1}^{N}(y_i-\hat{y}_i)^2$	Smooth	Sensitive to outliers
Mean Absolute Error (MAE)	$\frac{1}{N}\sum_{i+1}^{N}\lvert y_i-\hat{y}_i\rvert$	Robust to outliers	Constant gradients
Log-Cosh Loss	$\sum_{i+1}^{N}\log(\cosh(y_i-\hat{y}_i))$	MSE for small, MAE for large values	Computational overhead
Quantile Loss	$\sum_{i+1}^{N}\max(q(y_i-\hat{y}_i),(q-1)(y_i-\hat{y}_i))$	Not just the mean	Requires choices, constant gradients

Classification = Categorical Targets

- Classification targets different categories that can be labelled with discrete values
  - But no information in absolute values
  - Differences $(y_i - \hat{y}_i)$ don't work
- For two classes we want that something is either *cat/"signal"/1* or *dog/"background"/0*
  - But this is maybe too simplistic/idealistic
  - Instead let's think in probabilities

Cross-Entropy

- Predict probabilities $\hat{y}_i \in [0,1]$ with $\Sigma_i^C \hat{y}_i = 1$ for C classes
  - Softmax function gives the correct mapping: $\hat{y}_i = f(\vec{s})_i = \frac{e^{s_i}}{\sum_j^C s_j}$
- Quantify likeliness of target and prediction with a goodness-of-fit test
  - e.g. Kullback-Leibler distance $D(t(x), p(x)) = \Sigma_x t(x) \log \frac{t(x)}{p(x)}$
  - … roughly equivalently: cross entropy $H(t(x), p(x)) = -\Sigma_x t(x) \log p(x)$
- Put this all together and you get the Cross Entropy Loss for target t and prediction s

$$L(\vec{t}, \vec{s}) = -\sum_i^C w_i t_i \log \frac{e^{s_i}}{\sum_j^C s_j}$$

Activation Functions

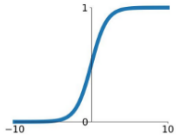
- Without activation functions $\vec{\sigma}$ or linear $\vec{\sigma}(\propto \vec{x})$

$$\vec{\hat{y}} = \ldots \vec{\sigma}_{(2)} \left(W_{(2)} \vec{\sigma}_{(1)} \left(W_{(1)} \vec{\sigma}_{(0)} \left(W_{(0)} \vec{x} + b_{(0)} \right) + b_{(1)} \right) \ldots \right) \quad \rightarrow \quad \vec{\hat{y}} = W_{eff} \vec{x}$$

- Non-linear function needed and we want them to be differentiable, commonly used ones are

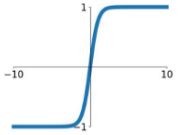
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



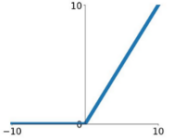
tanh

$$\tanh(x)$$



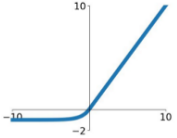
ReLU

$$\max(0, x)$$



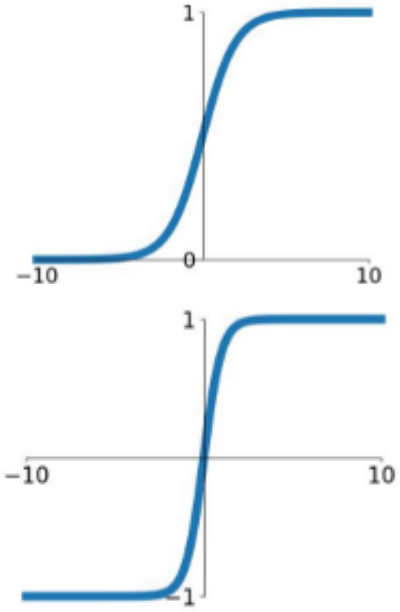
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



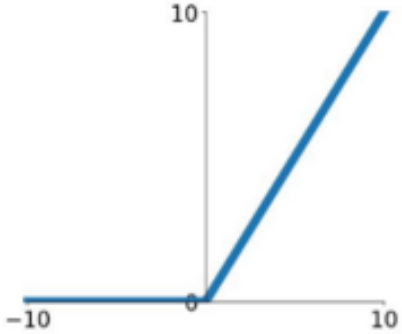
Discussion of Activation Functions

- Sigmoid historically used a lot because maps $\mathbb{R} \to (0,1)$, useful for probabilities
  - But $d\sigma(x)/dx = \sigma(x)(1-\sigma(x))$ vanishes for small and large values
- Tanh maps $\mathbb{R} \to (-1,1)$, useful for classification, better than sigmoid in many cases
  - But still gradient can approach zero and computationally quite expensive
- Both: Lead to static behavior at large and small inputs

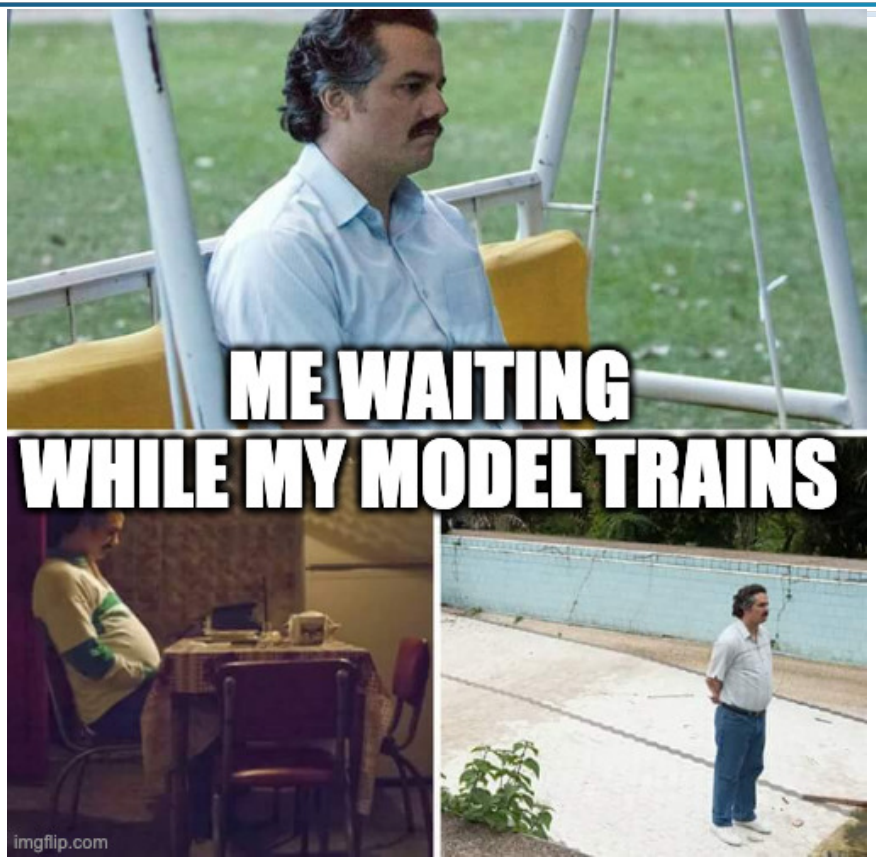


Minimal Amount of Non-linearity: ReLU

- Computationally simple and non-linear
- Threshold behavior
  - Constant gradient of 1 for $x > 0$, never zero
  - Gradient 0 for $x < 0$, unit dead
    - Can be desired turning off unnecessary neurons
    - Network can “get stuck”, neurons dying out
- ELU is one variant that fixes dying ReLU
  - Also smooth transition at 0 and gives negative values for negative inputs



Model Training

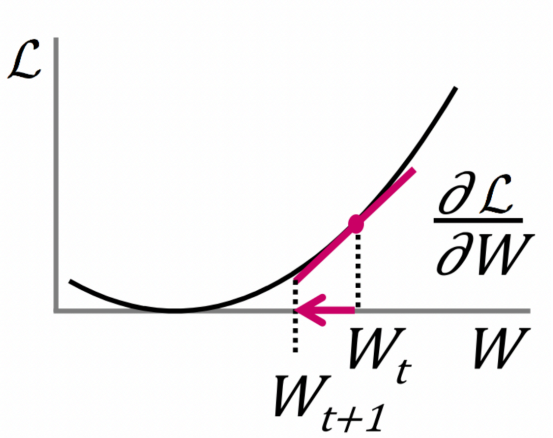


Gradient Descent

- How can we optimize the parameters of the model to optimize the associated loss?
  - Minimization problem, but the loss has many parameters
- Start randomly, measure the local gradient, go a step in direction of gradient, repeat!

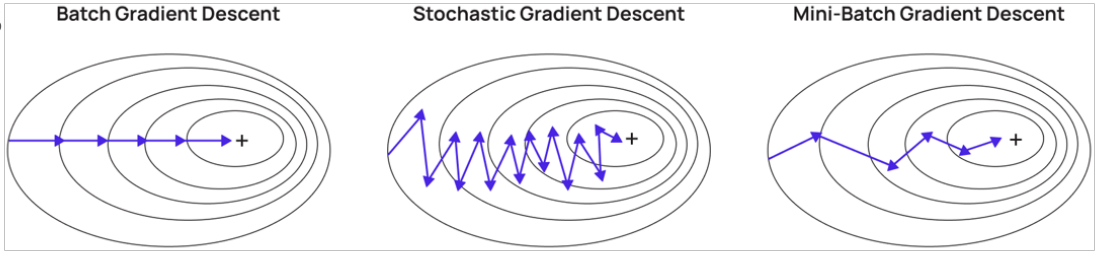
$$W_{t+1} = W_t - \alpha \frac{\partial L}{\partial W}\Big|_{W_t}$$ with learning rate α

- Iterative numeric minimization of the loss



Stochastic and Batch Gradient Descent

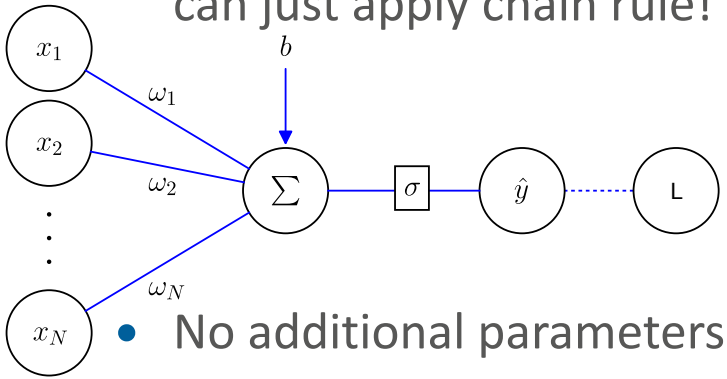
- Computing the gradients and updating the model parameters is computationally expensive
  - Do it for each data point → *Stochastic:* Noisy, fast convergence
  - Only once for all data → *Batch* $L := \langle L \rangle_{\text{batch}}$ Fluctuations average out, big steps



  - We call one full pass through all data (rather than a batch) an *epoch*

Backpropagation

- How do we get the gradient with respect to the model parameters?
- Remember we have chosen all ingredients to be differentiable. So we can just apply chain rule!

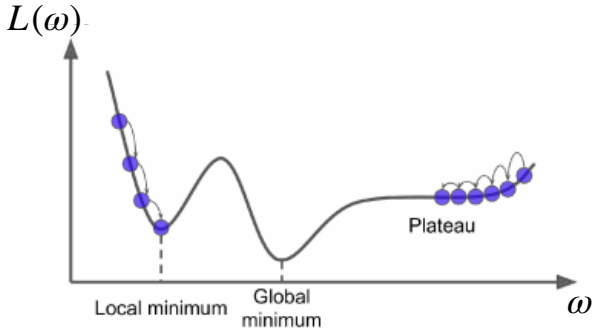


$$\leftrightarrow \quad \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \sigma} \frac{\partial \sigma}{\partial \Sigma} \frac{\partial \Sigma}{\partial w_1}$$

- No additional parameters needed!
- Implemented as exact “automatic differentiation” , e.g. Autograd

Optimizer

- Backpropagation and Gradient Descent are the backbone of model training
  - Learning rate and extensions with other parameters need to be set
  - All together is called the optimizer
- But real Loss Landscapes can get quite complicated, there can be
  - Plateaus $\rightarrow \partial L = 0$
  - Local minima $\rightarrow \partial L = 0$
  - Fluctuations
  - Narrow spikes and minima



Optimizer Optimization — Adaptive Learning

- Individual parts of the network might learn different things and therefore benefit from individual learning rates
- Measure the variance of all local gradients accounted
- And reduce learning rate when variance is high
  - Put more emphasis on features containing important information
- Implemented in Adagrad

$$\Delta_t = \alpha \frac{\partial L}{\partial W}\Big|_{W_t}$$

$$v_t = \sum_t \left(\frac{\partial L}{\partial W}\Big|_{W_t} \right)^2$$

$$\alpha \to \alpha_t = \frac{\alpha}{\sqrt{v_t} + \epsilon}$$

$\epsilon > 0$ for stability

Optimizer Optimization — Adaptive Learning

- Might lead to very small learning rates over time when v_t grows indefinitely
- Try regularizing with a moving average $v_t = \rho v_{t-1} + (1-\rho)\left(\frac{\partial L}{\partial W}\Big|_{W_t}\right)^2$
  - Not full history, but at least previous cycle

  with $0 < \rho < 1$
- Implemented in RMSprob

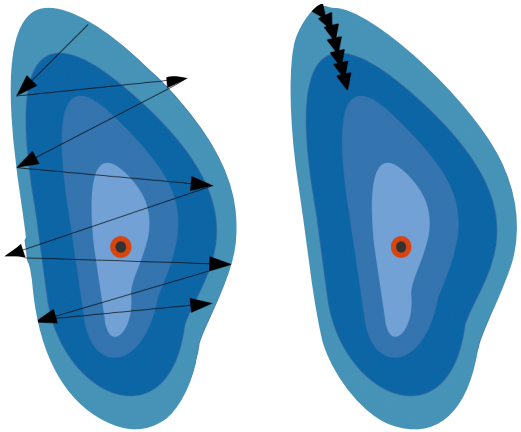
Optimizer Optimization — Adding Momentum

- Often biggest improvements in the loss at the beginning
- Small local bumps and plateaus should not suddenly stop optimization "in right direction"
- Keep some speed $\beta < 1$ from the previous cycles $\Delta_t = \beta\Delta_{t-1} + \alpha \frac{\partial L}{\partial W}\Big|_{W_t}$

- Combine this with Adaptive Learning and you get Adam
  - Most used optimizer in modern model training
  - Always the first choice

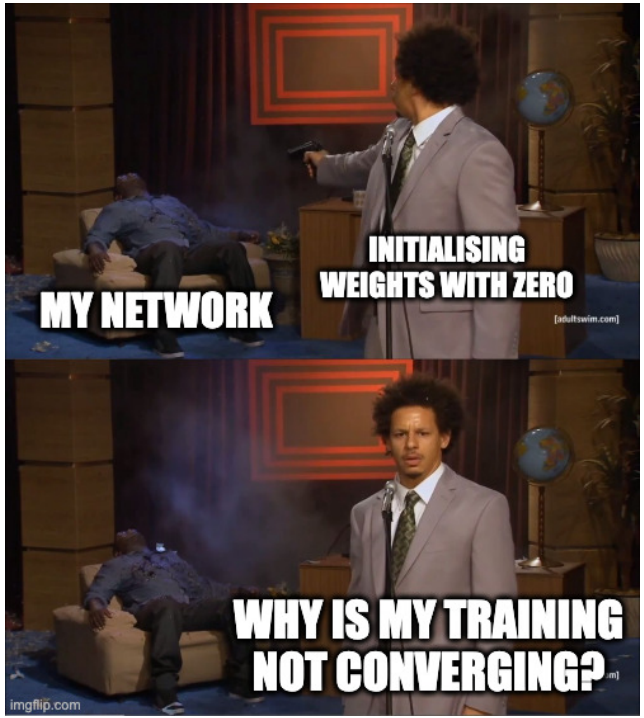
Learning Rate Decay

- Even with Adam optimization can circle around minimum
- Reduce the learning rate according to some schedule
  - Multiply by some ρ every N steps
  - Check for plateauing loss with history of last n losses and reduce k times
  - …
- Tools often called Schedulers



The Symmetry Problem

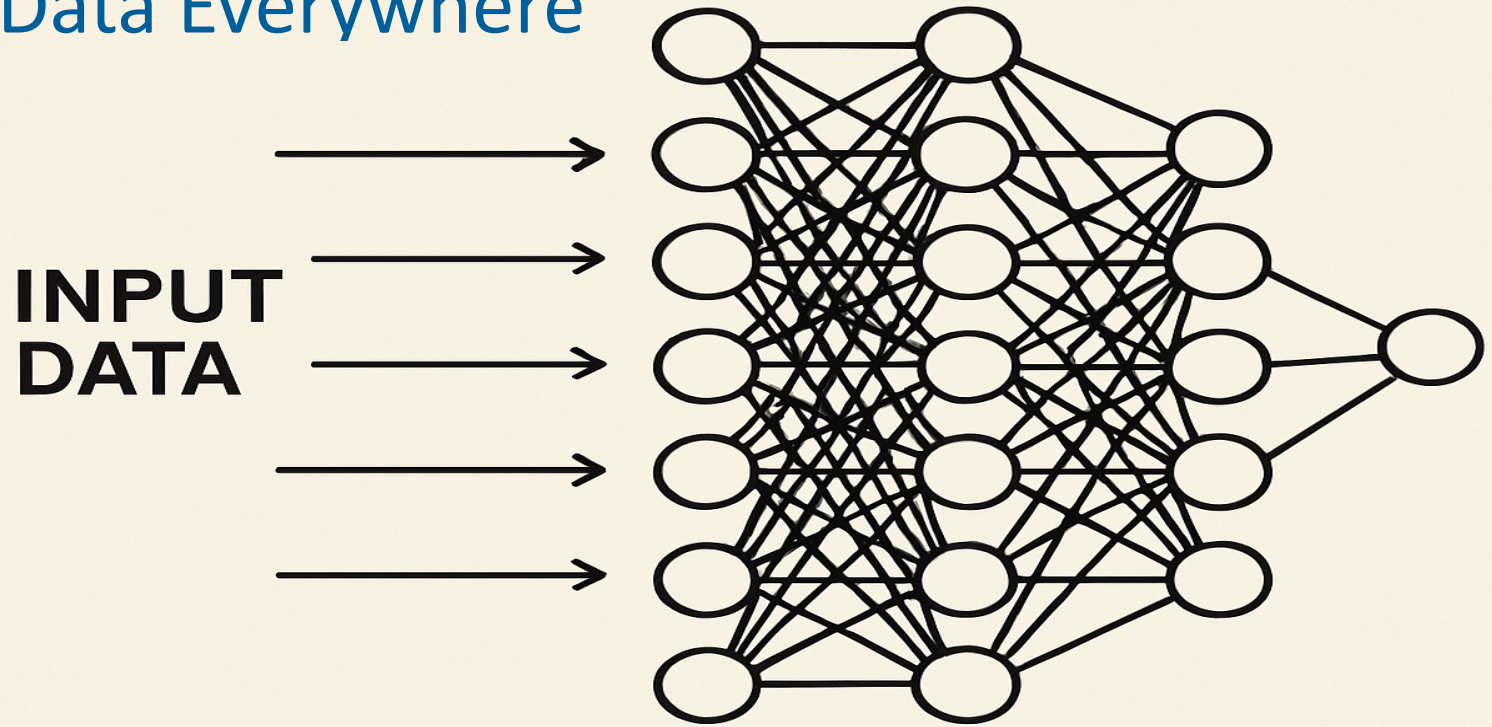
- All weights are zero
  - All activations are zero
  - Gradients are zero
- Similar for setting all weights to same constant
- Symmetries $\mathcal{S}$ in weight matrix W
  $\mathcal{S}W = W \rightarrow \mathcal{S}\partial L \approx \partial L$ limit explorable space
- Better: Random initialization



Xavier Initialization

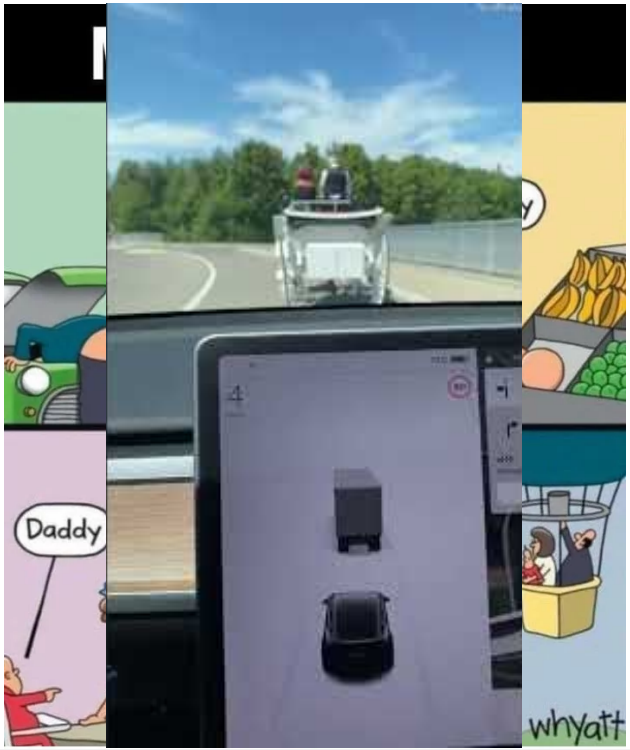
- NNs are all about the weights
  - Too large: Saturation
  - Too small: Might get stuck
- In back-propagation the gradient tends to get smaller the deeper the network
- Extensively studied: Glorot et al., arXiv:1502.01852
  - Use the established defaults
    - PyTorch even hides the functionality, so you don’t get easy access

Data Everywhere



Input Data

- Data is the most crucial part of the neural network training
  - Biases that are in data will end up in the model
  - But also numerics and types of data play a role



Data Normalization / Scaling

- Large input data might lead to vanishing gradients, saturation, numerical instability, …
  - Weights from left to right decrease
- What if large output data is expected?
  - Weights from left to right have to increase
- Normalize data to same value range

$$f \to f' = \frac{f - \mu}{\sigma} \qquad \text{with} \quad \mu = \frac{1}{m}\sum^{m} x_i \,, \quad \sigma^2 = \frac{1}{m}\sum^{m} (x_i - \mu)^2 + \epsilon$$

- For outputs retransform $f' \to f = f'\sigma + \mu$

Batch Normalization

- Apply normalization batch-wise before each layer input $\hat{x}_i = \frac{x_i - \mu}{\sigma}$
  - Reduces sensitivity to initial weights
  - Improves numerical stability
  - But forces $\langle x \rangle = 0$ and $\sigma_x = 1$
- Reintroduce sensitivity by reshaping the inputs $\hat{x}_i' = \gamma \hat{x}_i + \beta$ with learnable parameters γ and β
  - Allow the model to “learn it back”

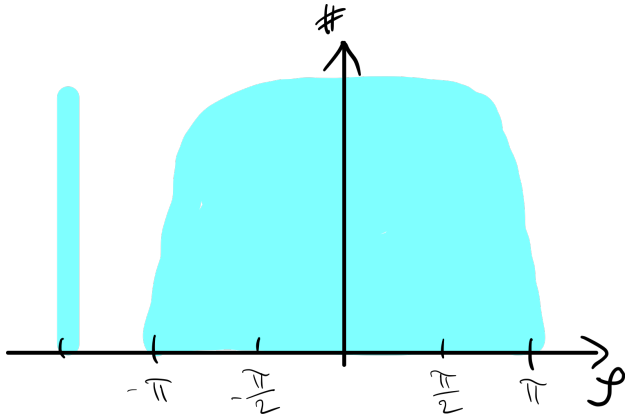
Categorical Input Data

- Categorical flags can be easily mapped by integers
  - But adjacent values do not carry additional info $0_c - 1_c = 0_c - 2_c$
- Create a vector with $d = N_c$ with one position equals 1 others 0:
  One-hot encoding
  - High-dimensional and sparse (mostly zeros), unrelated categories
- Make categories learnable with embedding layer
  - Dimension of embedding vectors is a parameter
  - Similar categories will end up with similar vectors

Index	Category	Embedding
0	Red	[0.12, 0.67, 0.21]
1	Blue	[0.11, 0.68, 0.19]
2	Green	[0.95, 0.05, 0.88]

Missing Input Data

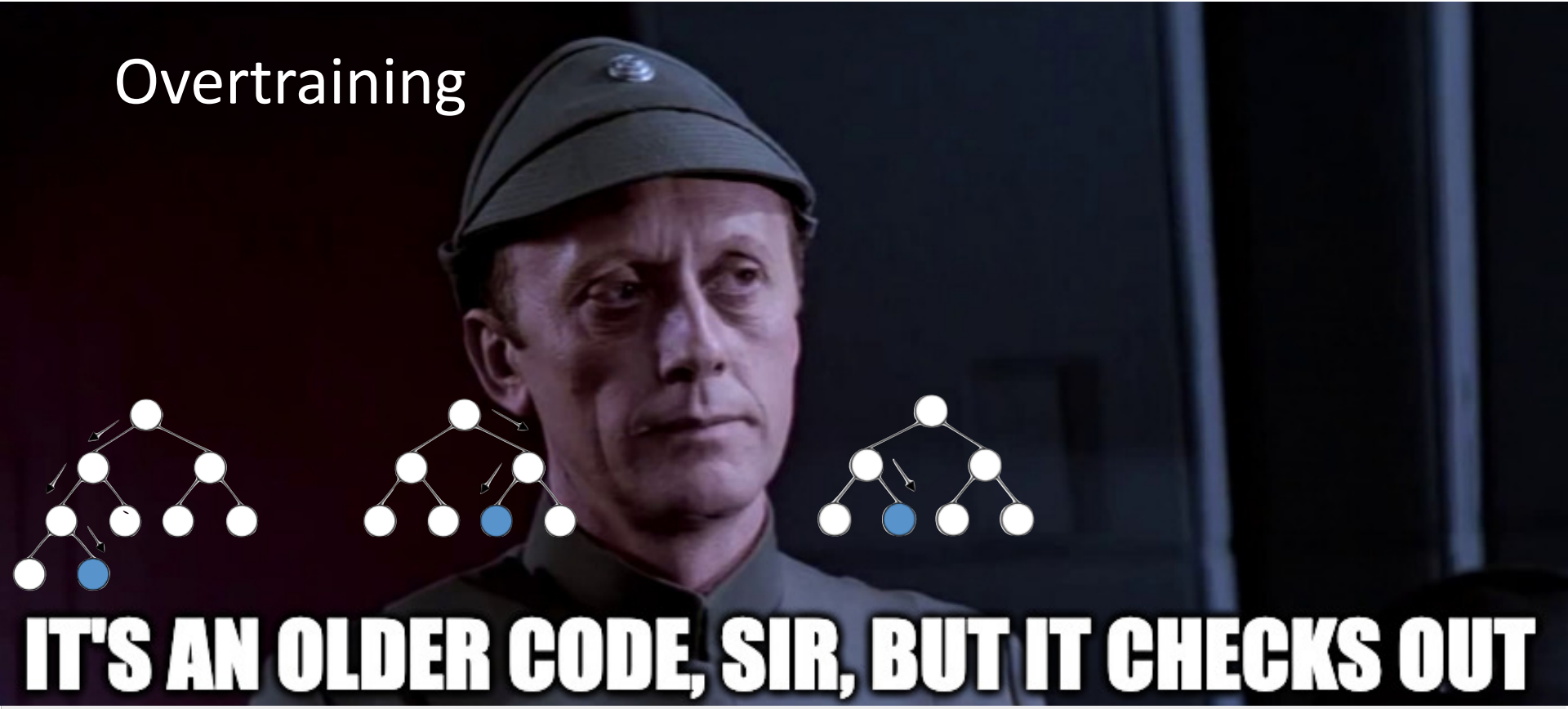
- Input features might be missing in some data (not every collision event contains a charged lepton)
- Train multiple networks
  - Computationally expensive
  - Less data for training of each
- Encode missing features in “default” values
  - Capture additional information from the absence of data
  - Hard to define



Data Imbalance

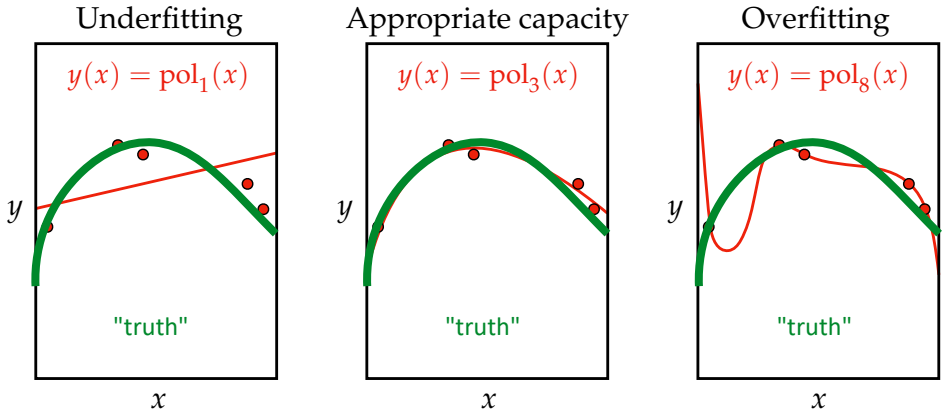
- Data is not uniformly distributed
  - One class more likely, peaked distribution
  - Network will be less sensitive to regions/classes that are sparsely populated
- Possible workarounds:
  - *Down-sampling*: Remove some data from overly populated areas
  - *Collocation*: Let training batches to be uniformly distributed
  - *Sample-weights*: weight each data item to create uniformity
    - Loss function implementations support this

Overtraining



Model Capacity and Capability

- Many free parameters leads to high capacity
  - With enough parameters, any data can be fitted
  - “Memorize the data”, no generalization! → *Overtraining*
- Too few parameters will restrict capability
  - The model cannot capture the complexity
- Luckily, with NNs we don’t need to restrict ourselves



Regularization

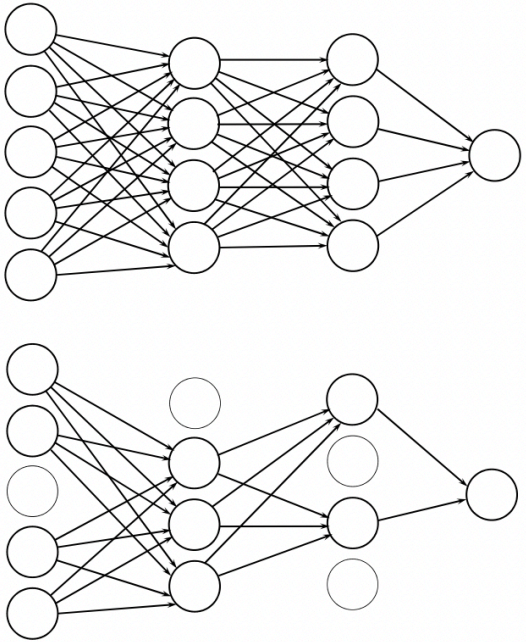
- Often volatile behavior or large values alternations are necessary
  - Some weights need to become larger than others
  - But we want to do this moderately
- Regularize the loss with penalty term $L_1 = \Sigma_W w_i$ or $L_2 = \Sigma_W w_i^2$

$$L_{\text{tot}} = L + \lambda L_{1/2}$$

  - If really needed the training will slowly enforce higher weights
  - Tuning the mediator λ can help

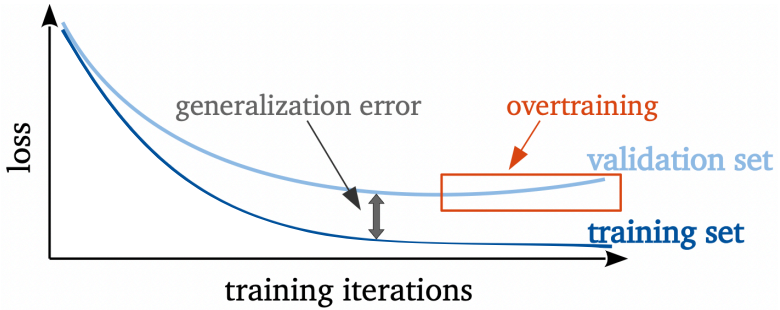
Dropout

- When one weight becomes large, others might adapt
- Restrict “undesired reliance” of weights on each other by stochastically switching parts off
  - Randomly turn off neurons with probability p during processing
  - Scale up next neurons inputs by $1/(1-p)$ to account for missing neuron
  - Next iteration pick different neurons



Validation and Test

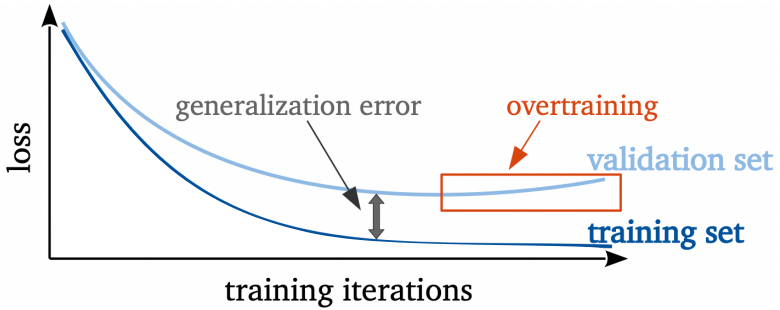
- How can we be sure we are not screwing up? Validate!
- Test performance of your NN on independent *validation data*
  - If loss diverges, the model overtrains
  - Guides the selection of the model
- Also keep another independent set of *test data* for final true measure of generalization
  - Overall reduction of data for training



Early Stopping

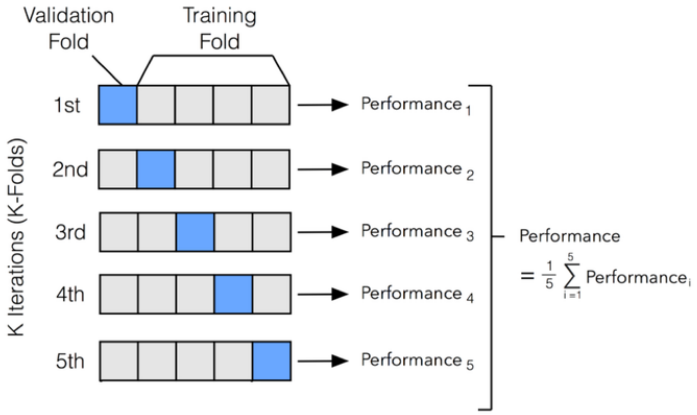
- Simplest method to fight overtraining
- Stop as soon as validation data reliably shows no improvement or overtraining
- Or as soon as generalization error becomes too large

- Pick the model that has performed the best on validation data



Cross Validation

- Splitting the available data when it is sparse can be painful
  - We can trade data (use all data for training and validation) against compute time
- k-fold cross validation
  - Split data into k sub-samples (folds)
  - Train on $k-1$, validate on remaining
  - Rotate folds and repeat (total k times)
  - Average the performance across all folds
- Also reduces dependence on train/validation split



Other Optimizations

Miscellaneous Optimizations

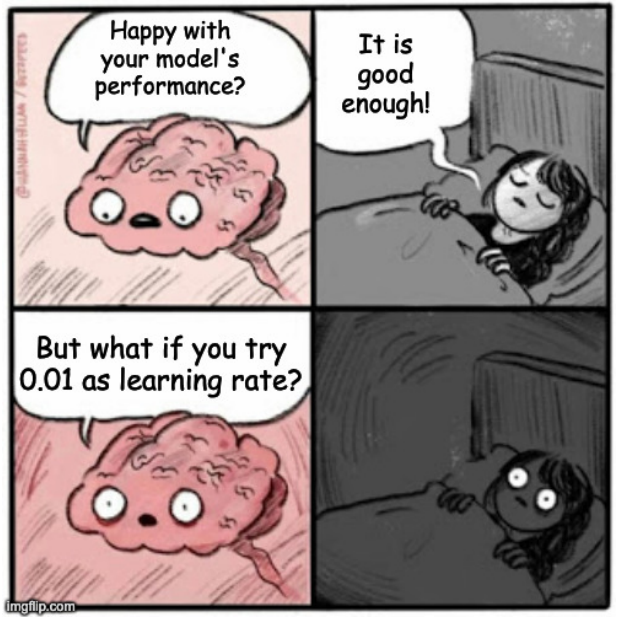
- More data
- Data augmentation (more data and free, but situational)
- Noise injection (can be more data, but loss in performance)
- Ensemble training (averaging over many networks)
- Model parameter reduction
  - Pruning (Features & Parameters)
  - Shared Parameters (i.e. CNNs, Transformer)

Hyper-Parameters

- Many parameters introduced to the various building blocks of a model
  - Architecture: number of layers & nodes per layer; types of layers, batch normalization; activation; weight initialization
  - Optimizer: learning rate α, momentum β, decay ρ
  - Training: Batch size, folds, splitting fractions, regularization, dropout
- Some have well tested defaults
- Some are independent and can be optimized individually
- Remainders have to be optimized simultaneously

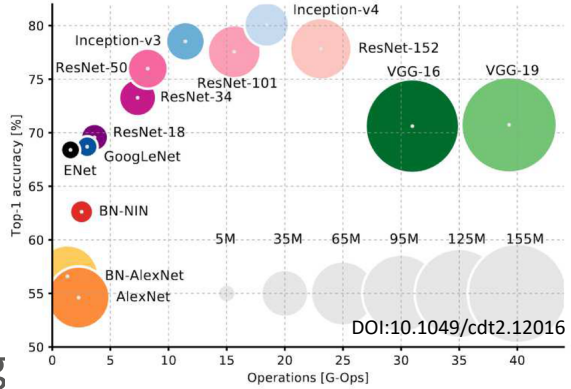
Hyper-Parameter Optimization

- Many approaches exist to explore hyper-parameter space
  - Random search
  - Grid search
  - Bayesian optimization
- Overall a very brute-force and expensive procedure but sometimes necessary
  - Definitely necessary for highest performance



Remark: Resource Availability & Optimization

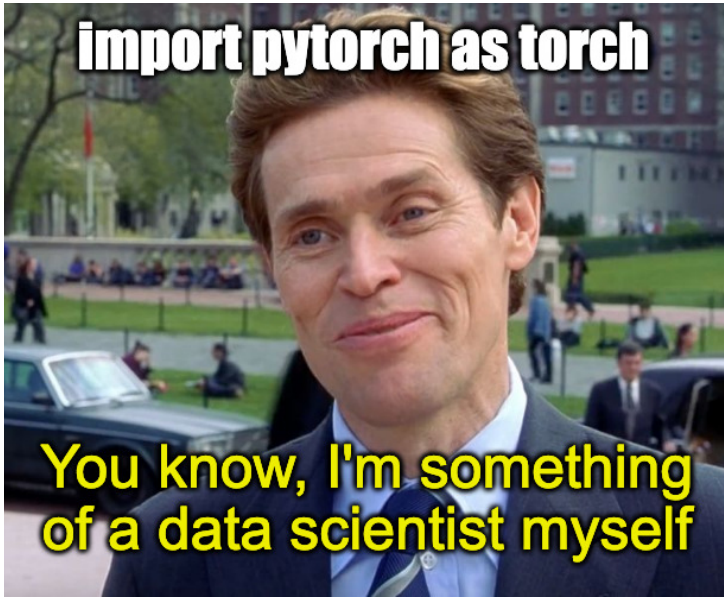
- Training (and evaluating) NNs is computationally expensive
  - Data needs to be stored and moved
  - A ton of algebraic operations are performed
  - Similar amount of gradients are calculated and parameters updated
  - Repeated many times
- Software and hardware ensure efficient processing
  - But hardware is expensive, and running it energy intensive
- Think twice before you heat someone's server! Be resource-efficient!



Finally, some hands-on

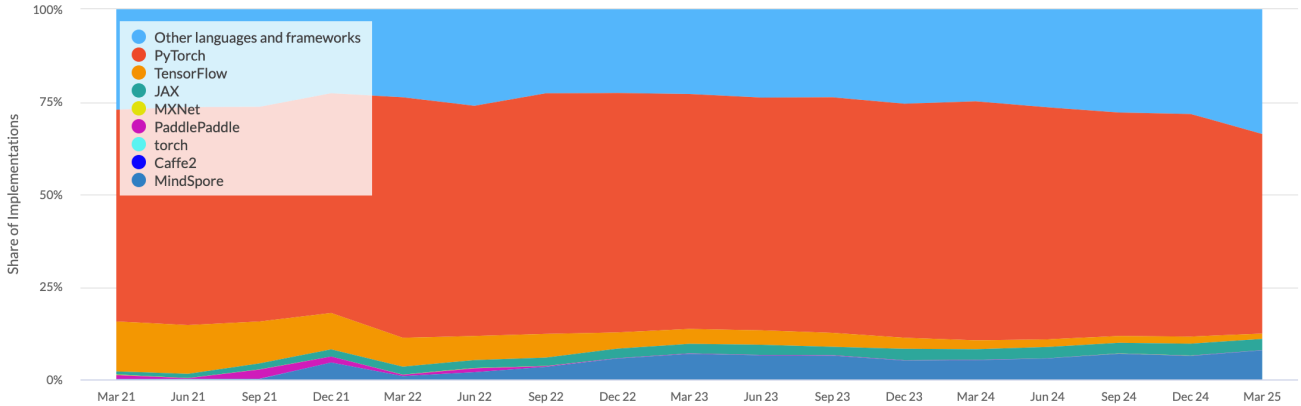
NNs as Industry Standard

- Wide and active community develops standard tools, e.g. PyTorch, tensorflow, …
  - Standard features already implemented
  - Optimized computation, memory consumption, data handling
- Collaborative development and usage of common open tools enabled the recent success of ML



PyTorch

- You will continue using PyTorch for the exercises



- It has become the most widespread tool for ML applications
  - Benefit from the work of the community and be a part of it!

Today's exercise

- Will depend on the previous exercises, we will continue there
- Open the DNN4HEP_exercise.ipynb notebook, also uploaded on the Indico, in Google Colab
  - There are code blocks marked with

    This is for the Mastering Model Building exercise. Skip this block in your first pass!

    Ignore them on your first pass!
  - The instructions should be self-explanatory. If not, feel free to ask!
- Have fun!