Karlsruher Institut für Technologie

Machine Learning 1 – Fundamentals

Neural Networks – Hyperparameters

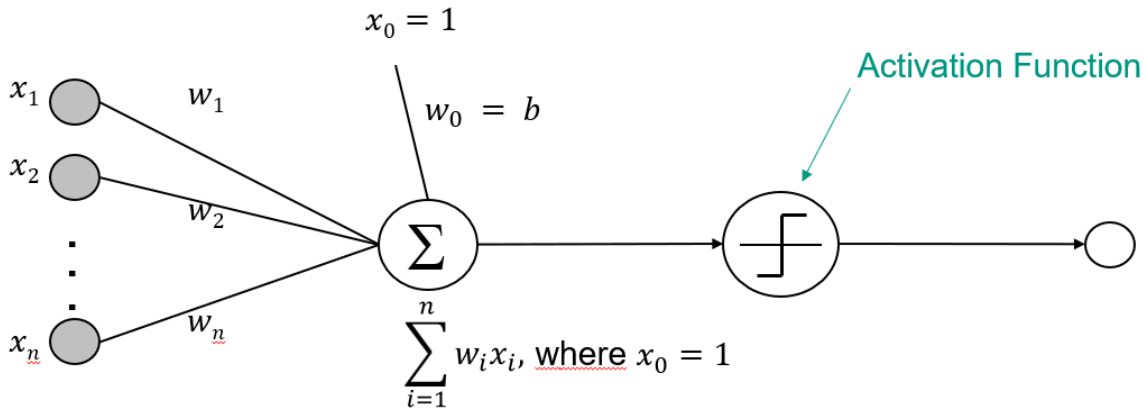
Prof. Dr. J. M. Zöllner, M.Sc. Nikolai Polley, M.Sc. Marcus Fechner

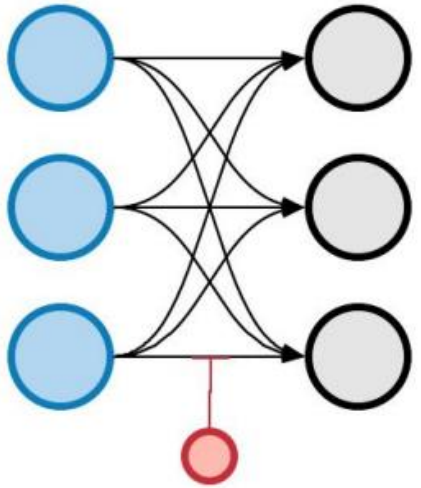
Overview

- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

Neural Networks – Architecture

- Collection of connected neurons
- Each neuron returns a weighted sum of its inputs
- An activation function is applied after each neuron to enable the network to learn non-linear functions





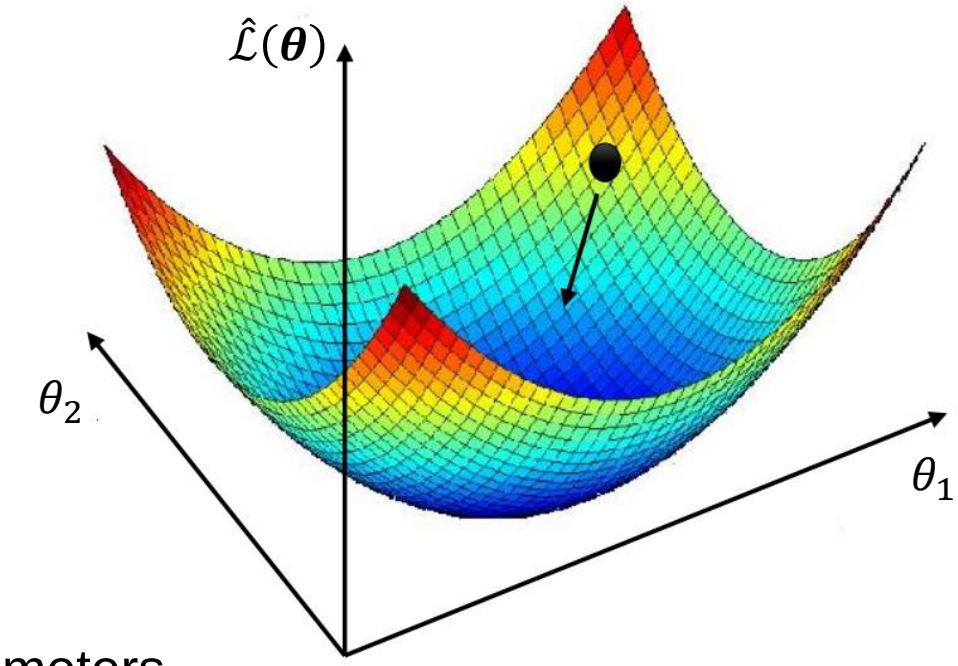
$$\boldsymbol{f(x) = Wx + b}$$

$$\sum_{i=1}^{d} \mathbf{w}_i \mathbf{x}_i + b = \sum_{i=0}^{d} \mathbf{w}_i \mathbf{x}_i \qquad \mathbf{x}_0 := 1$$

Neural Networks – Training

Recap

- **Supervised method:**
  - $(x, y) \in D$
    - x: input data
    - y: label/target of input data
  - Can be used for regression and classification



- **Empirical Loss**: $\hat{\mathcal{L}}(\boldsymbol{\theta}) = \frac{1}{|D|} \sum_{(\boldsymbol{x},y)\in D} \ell(\hat{y}_{\boldsymbol{\theta}}(\boldsymbol{x}), y)$
- **Objective**: $\boldsymbol{\theta}^* \leftarrow \arg\min_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta})$
- **Iterative Algorithm**:
  - Calculate output $\hat{y}_{\boldsymbol{\theta}}$ of input data with current parameters
  - Use output $\hat{y}_{\boldsymbol{\theta}}$ and label y to calculate loss
  - Calculate gradient for each parameter to search for a direction in which $\hat{\mathcal{L}}$ decreases
  - Update parameters: $\boldsymbol{\theta_{t+1}} \leftarrow \boldsymbol{\theta_t} - \alpha \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t)$

Neural Networks – Backpropagation

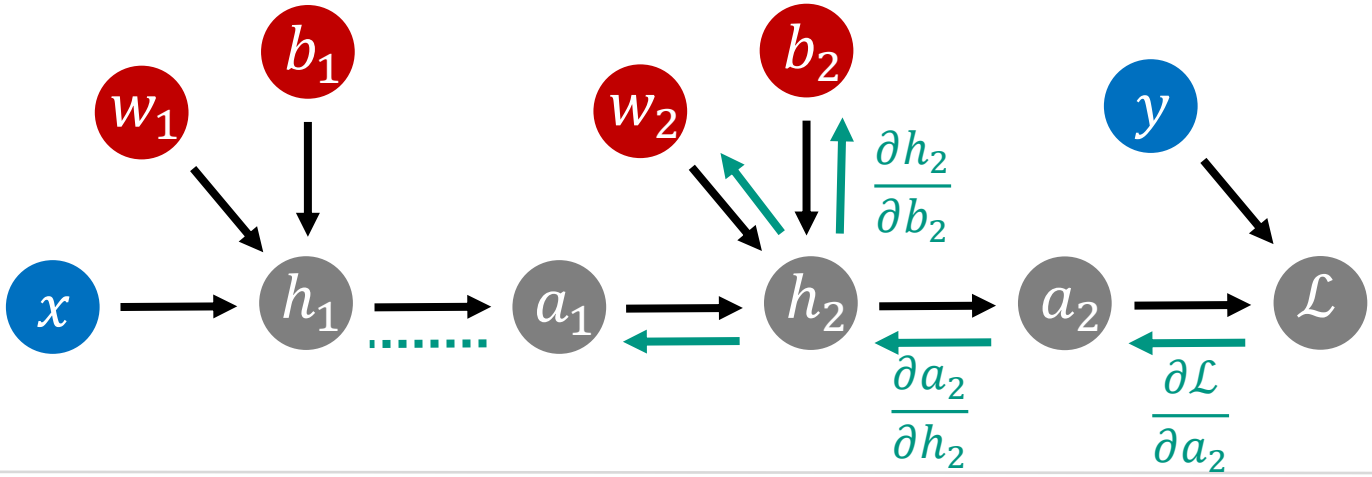
Recap

- **Backward pass**: Calculates the gradient for each parameter by propagating the error signal back through the network.
- **Example**: Two-layer neural network
  - $h_1 = w_1 x + b_1$
  - $a_1 = \sigma(h_1)$
  - $h_2 = w_2 a_1 + b_2$
  - $a_2 = \sigma(h_2)$
  - $\mathcal{L} = \ell(a_2, y)$

- **Gradients**

Already computed parts

  - $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial w_2}$
  - $\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial b_2}$
  - $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial a_1}\frac{\partial a_1}{\partial h_1}\frac{\partial h_1}{\partial w_1}$
  - $\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial h_2}\frac{\partial h_2}{\partial a_1}\frac{\partial a_1}{\partial h_1}\frac{\partial h_1}{\partial b_1}$



Stochastic Gradient Descent (SGD)

Recap + New

- **Minibatch Learning:** Approximate the expectation with subset $B \ll N$ of the data

$$\nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_t) = \nabla_{\boldsymbol{\theta}_t} \mathbb{E}_{(x,y)\sim D}[\ell(\hat{y}_i, y_i)] \approx \nabla_{\boldsymbol{\theta}_t} \frac{1}{B} \sum_{b=1}^{B} \ell(\hat{y}_b, y_b)$$

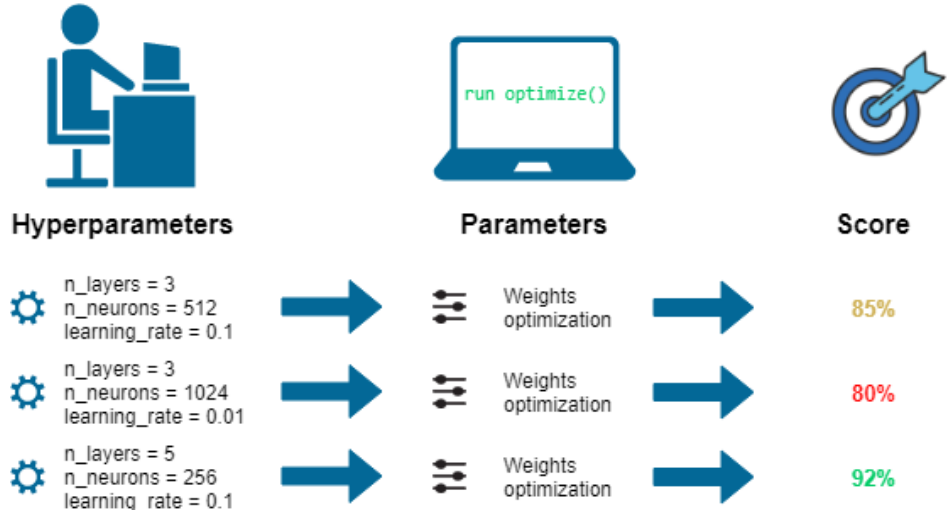
  - Use mini-batches for training instead of whole dataset
    - Usually minibatches contain between 32-512 training examples
- **Example**: We have 1000 training examples and a minibatch size of 100
  - **Iteration:** One update of model weights. Calculating loss of 100 examples, average their gradient and update weights
  - **Epoch:** One full cycle through training data, here 10 iterations
  - **Multiple Epochs:** Use the same training examples but reshuffle minibatches.
    - Often necessary because not enough data exist. Makes overfitting more likely.
    - (In computer vision oftentimes multiple hundreds of epochs)

Overview

- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

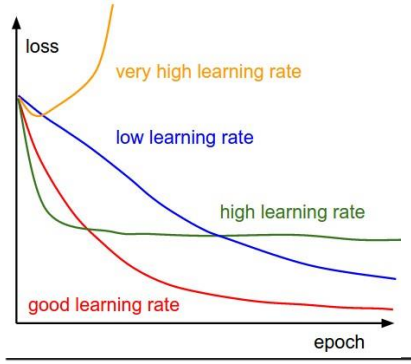
Terminology 术语

- **Parameter**
  - In the field of neural networks, this almost always refers to the weights and biases of the neurons
  - Values are learned by system and not manually set by programmer
- **Hyperparameter**
  - A setting of the training algorithm that must be defined before training
  - Choices need to be done manually



Why are Hyperparameters Important?

- Well-defined hyperparameters are necessary for successful training
  - Poor choice of hyperparameters can slow down or even prevent training altogether
- Optimal choice of hyperparameters can improve the performance of already excellent networks without requiring any architectural modifications to the learning system.
  - 2018: Google researchers release the BERT neural network that beats all previous benchmarks
  - 2019: Facebook publishes network with identical architecture but different hyperparameters
    → 22% improvement



Hyperparam	RoBERTa$_{\text{LARGE}}$
Number of Layers	24
Hidden size	1024
FFN inner hidden size	4096
Attention heads	16
Attention head size	64
Dropout	0.1
Attention Dropout	0.1
Warmup Steps	30k
Peak Learning Rate	4e-4
Batch Size	8k
Weight Decay	0.01
Max Steps	500k
Learning Rate Decay	Linear
Adam ϵ	1e-6
Adam β_1	0.9
Adam β_2	0.98
Gradient Clipping	0.0

Covered in ML2

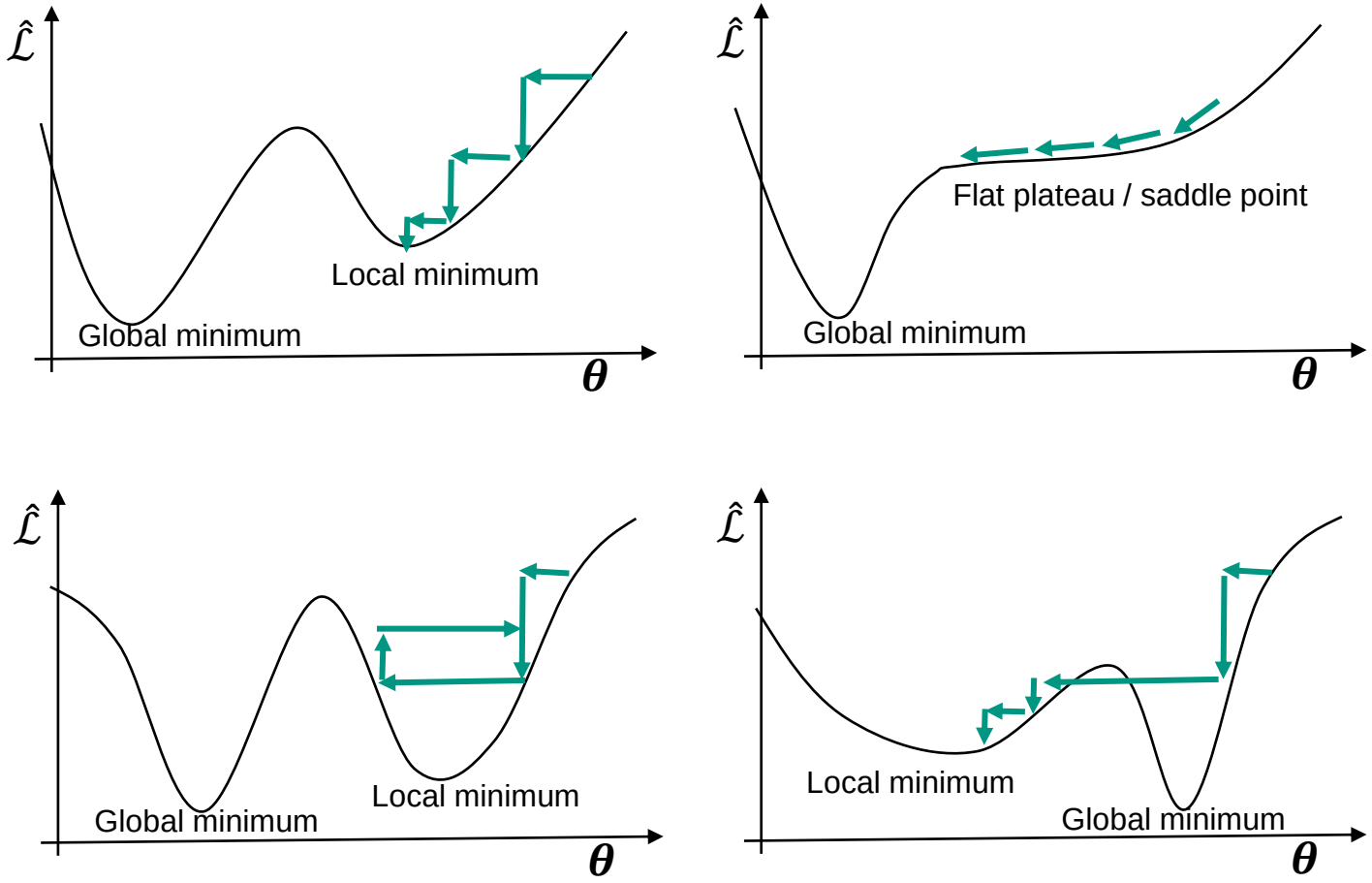
Covered in ML1

Overview

- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

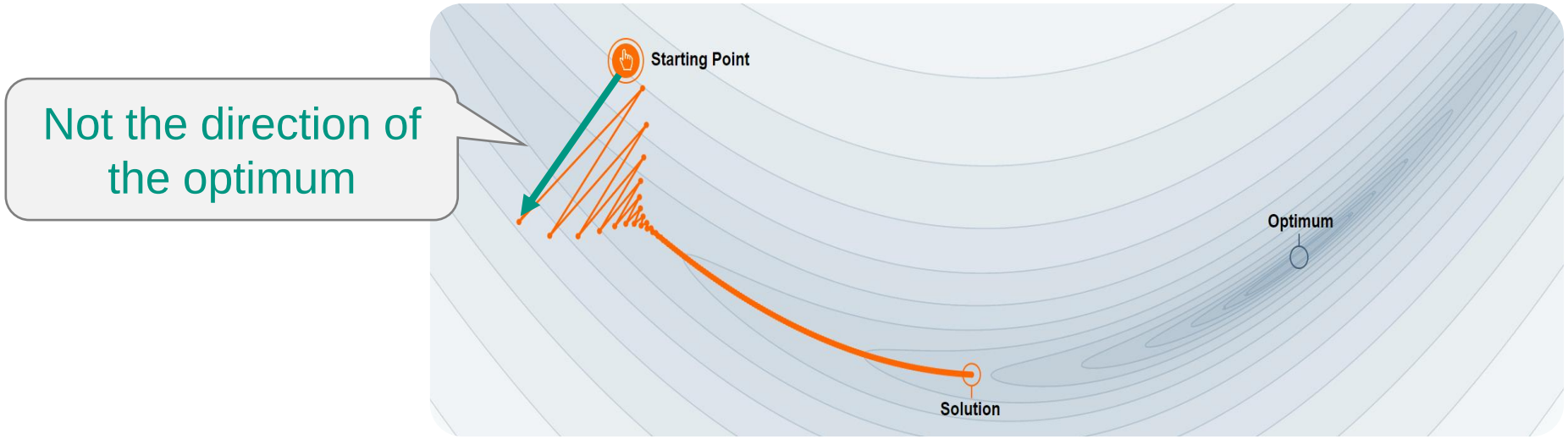
Loss Landscape

- Learning depends on various factors such as:
  - occurrence of local minima/ saddle points
  - slope of loss landscape
  - characteristics of local minima / saddle points
  - learning rate
  - …



Gradient Descent

- **Careful**: The steepest direction is not always the best!
- **Gradient Descent**: First-order iterative optimization method
  - No information about the curvature of the gradient!
- **Better?: Newton‘s Method**
  - 2nd order optimization method using first and second derivatives



Second-Order Optimization Methods

- Consider information of how the gradient itself changes
  - For example, using second-order Taylor series:
    $\theta_{k+1} = \theta_k - H(\theta)^{-1}\nabla h(\theta_k)$
    where $H(\theta)^{-1}$ is the inverse Hessian matrix
- **Problem**:
  - Finite memory: The Hessian contains $O(n^2)$ values for n parameters.
  - … and then we need to invert it
- **Solution**:
  - Apply heuristic methods that approximate higher order behavior to find the optimal solution based on the previous behavior of the gradients

Momentum - Real World Example

- Ball rolling down a mountain accumulating (physical) momentum
  - **Acceleration:**
    Even if direction of motion changes, the ball continues to accelerate downwards
  - **Smoothing/Dampening:**
    Valley keeps ball inside.

[Devin Graham, Youtube]

Momentum

- Iterative optimization:

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \boldsymbol{v}_k$$

  - with **gradient descent**

$$\boldsymbol{v}_k = -\alpha \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_k)$$

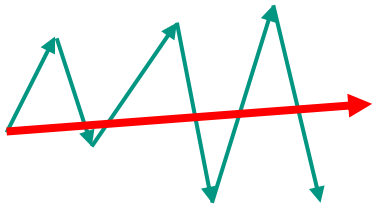
    - with **momentum**

$$\boldsymbol{v}_k = -\alpha \nabla_{\boldsymbol{\theta}} \hat{\mathcal{L}}(\boldsymbol{\theta}_k) + \beta \boldsymbol{v}_{k-1}$$

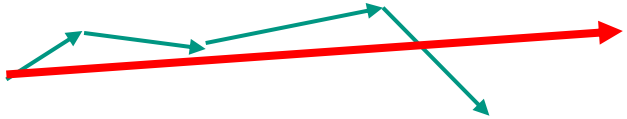
"Coefficient of momentum"

"previous direction"

- **Intuition**:
  - If subsequent gradient steps point in different directions, we should eliminate the directions that do not match.



  - If subsequent gradient steps point in similar directions, we should accelerate in that direction



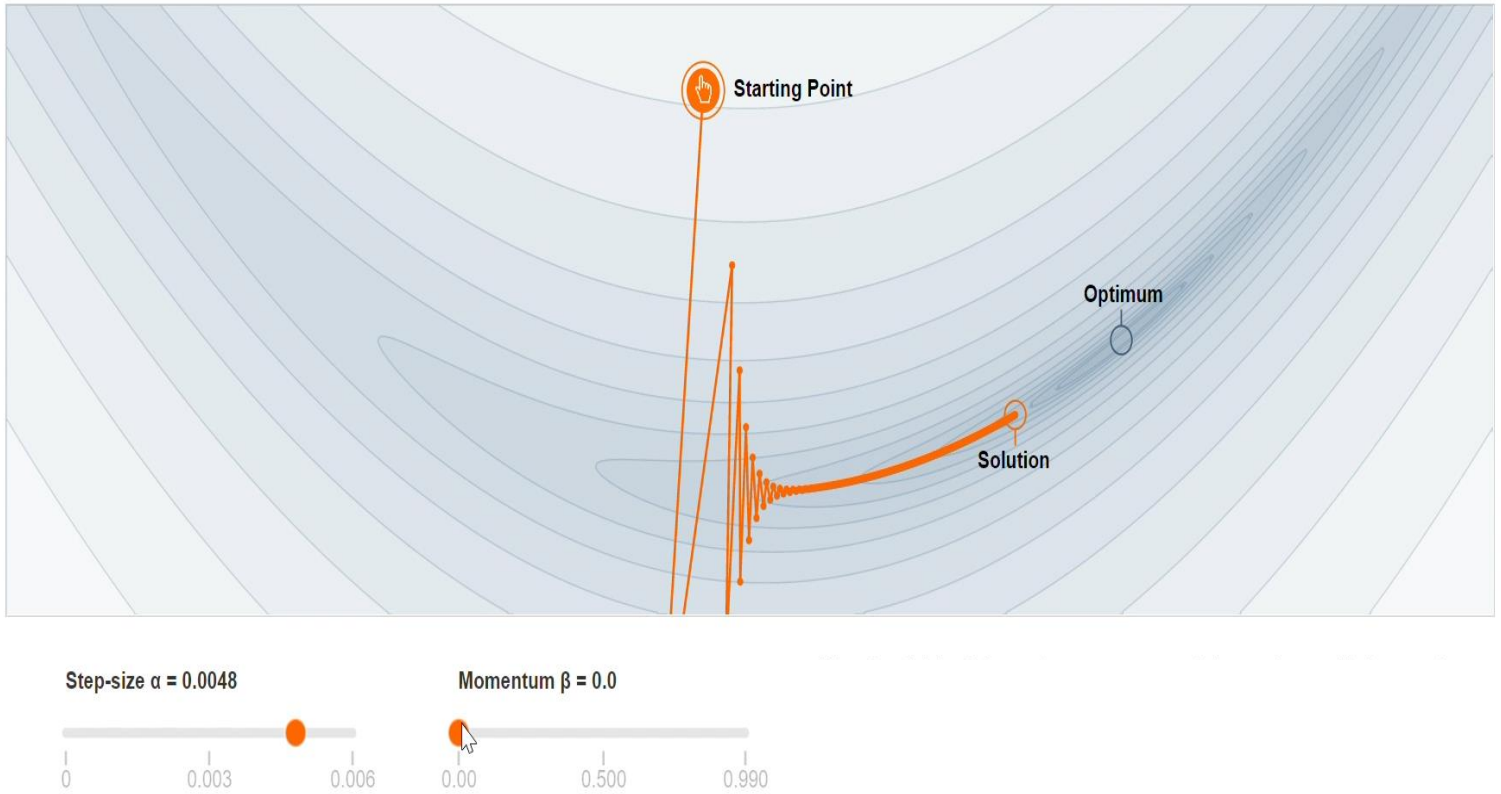
Momentum: Example

$$\beta = 0.9$$

Timestep k	$\boldsymbol{v}_k$
$k = 0$	$-\alpha \nabla_{\boldsymbol{\theta}_0} \hat{\mathcal{L}}(\boldsymbol{\theta}_0)$
$k = 1$	$-\alpha \nabla_{\boldsymbol{\theta}_1} \hat{\mathcal{L}}(\boldsymbol{\theta}_1)$ $- \ 0.9\ \alpha \nabla_{\boldsymbol{\theta}_0} \hat{\mathcal{L}}(\boldsymbol{\theta}_0)$
$k = 2$	$-\alpha \nabla_{\boldsymbol{\theta}_2} \hat{\mathcal{L}}(\boldsymbol{\theta}_2)$ $-0.9\ \alpha \nabla_{\boldsymbol{\theta}_1} \hat{\mathcal{L}}(\boldsymbol{\theta}_1)$ $- \ 0.81\ \alpha \nabla_{\boldsymbol{\theta}_0} \hat{\mathcal{L}}(\boldsymbol{\theta}_0)$

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \boldsymbol{v}_k$$

$$\boldsymbol{v}_k = -\alpha \nabla_\theta \hat{\mathcal{L}}(\theta_k) + \beta v_{k-1}$$



Adam – Adaptive Moment Estimation

- **Adam**: Normalized running estimates of mean and variance of each component of gradient. Custom learning rate for each parameter
  - Exponential moving average of gradient $\boldsymbol{m_k}$
    - (Momentum)
  - Exponential moving average of squared gradient $\boldsymbol{s_k}$
    - Scale gradient inversely proportional to squared root of history of gradients
    - Keep step sizes similar even for high differences in gradient values

Focus on direction of gradients →

Focus on magnitude of gradients →

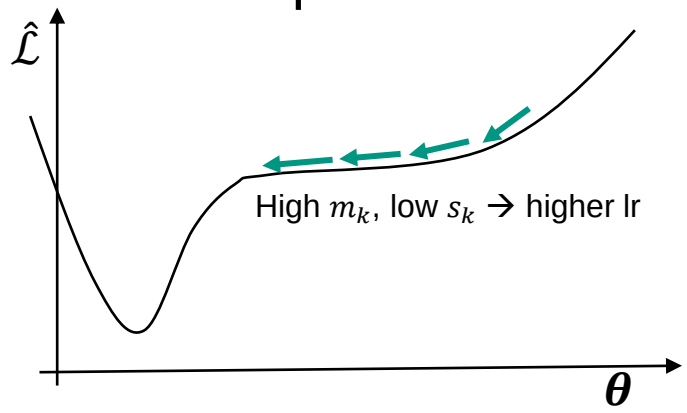
$$g_k = \nabla_\theta \hat{\mathcal{L}}(\theta_k)$$
$$m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k$$
$$s_k = \beta_2 s_{k-1} + (1-\beta_2) g_k^2$$

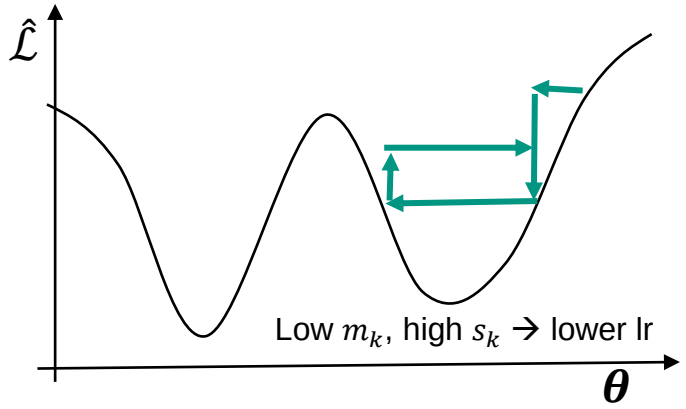
- (for completeness, m_k and v_k are initialized with zero, following is to correct bias)

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k} \qquad \hat{s}_k = \frac{s_k}{1-\beta_2^k}$$

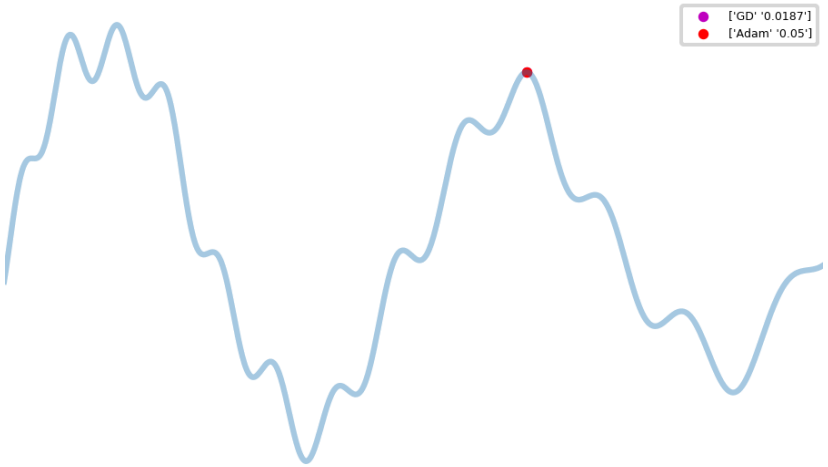
- Update parameters $\theta_{k+1} \leftarrow \theta_k + v_k$ with

$$v_k = -\alpha \frac{\hat{m}_k}{\sqrt{\hat{s}_k + \epsilon}}$$

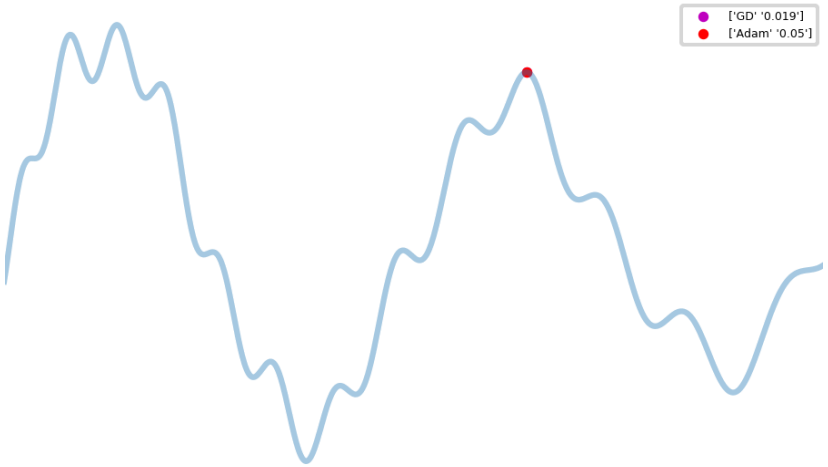




Adam – Adaptive Moment Estimation



Gradient descent with small learning rate: stuck in local minimum



Gradient descent with high learning rate: oscillating steps

- Additional hyperparameters β_1, β_2 and ϵ with default values $0.9, 0.999, 10^{-8}$.
- Requires more memory than SGD in optimization step
  - SGD calculates and stores for each parameter its gradient g_k
  - Adam needs to additionally calculate and store for each parameter its history: m_{k-1} & s_{k-1}
- **Adam approximates the diagonal of the Hessian (2nd order method)**

Optimizers Today

- **Adam**: Technique from 2015 but still used today
  - ChatGPT, LLaMA, DALL-E, Stable Diffusion …
  - One of the most cited papers of all time (200.000 citations)
    - (Nobel price winner 2016: Discovery of gravitational waves: 17.000 citations)
  - **AdamW:** Adam + **W**eight decay → see later slides
  - **Adafactor:** Adam with reduced memory requirements

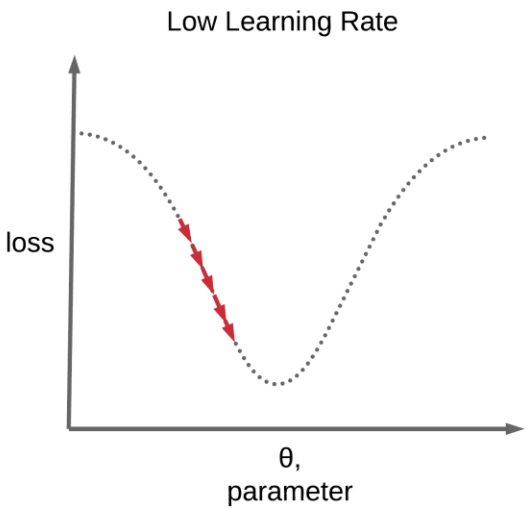
- Adam does not necessarily provide best results. SGD with/without momentum usually competitive and sometimes better.
  - But Adam is usually a good starting point!

Overview

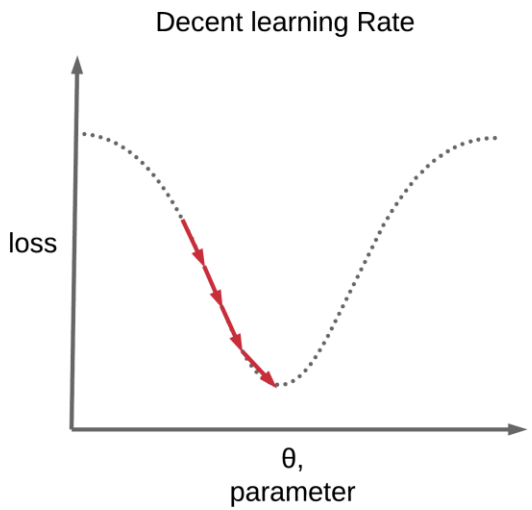
- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

Impact of Learning Rate

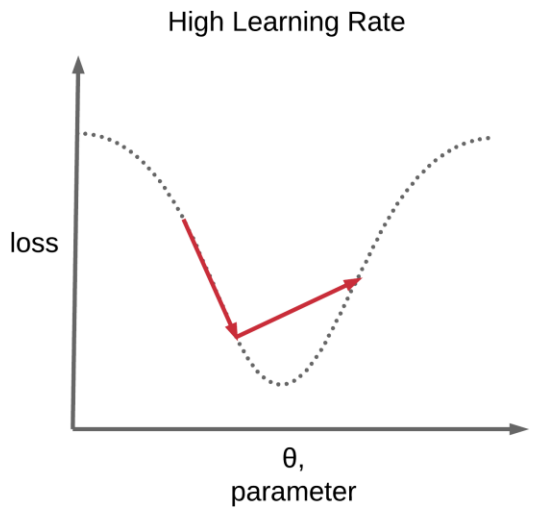
- Size of learning rate is important factor for training
  - *„In praxis, the learning rate is the most important hyperparameter“*



Too small learning rate requires lots of steps to find minimum.



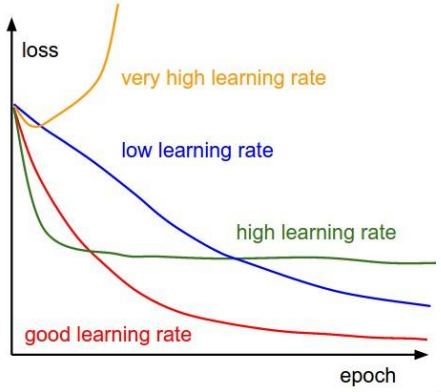
Suitable learning rate finds minimum in short time

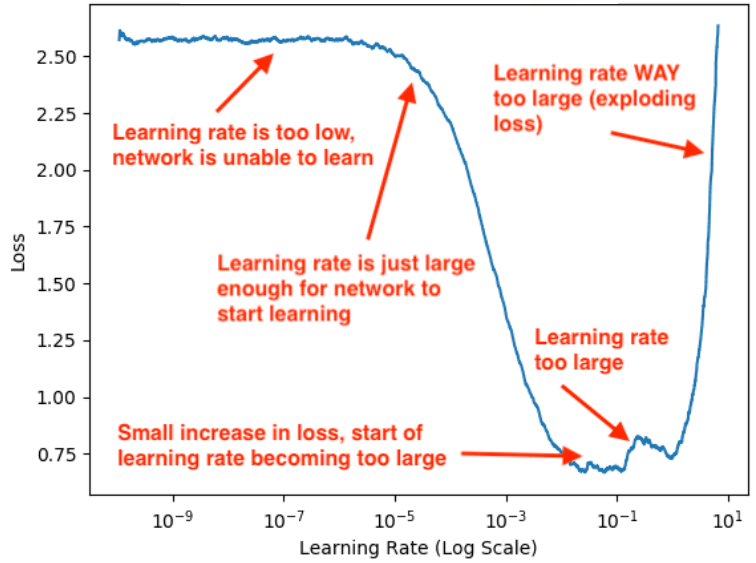


Too high learning rate can overshoot the minimum

Impact of Learning Rate

- **High learning rates**
  - Reduces loss fast, but don't find minimum
  - Too high learning rates can even diverge training
- **Small learning rates**
  - Take very long times to find minimum
  - Can get stuck at points with low gradients



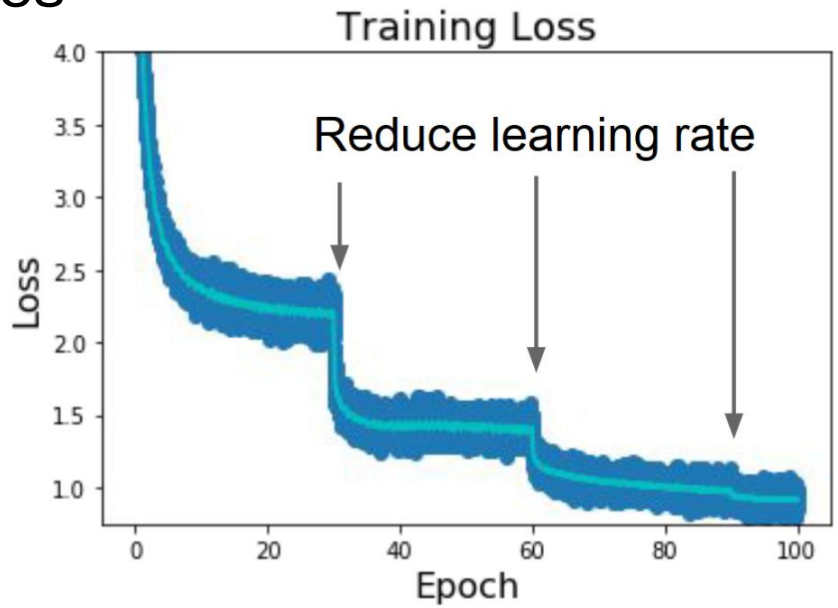


So, which Learning Rate to Choose?

- There is no optimum value
  - Values oftentimes between 0.0001 and 0.1
    - Start with 3e-4 or 1e-4
- Validate with different values
  - Goodfellow: Deep Learning Book: *„If you have time to tune only one hyperparameter, tune the learning rate“*
- **Better**: Change learning rate dynamically during training.

Dynamic Learning Rates

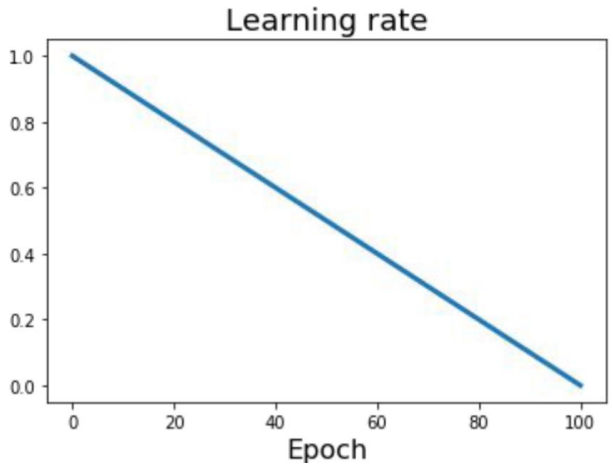
- **Idea**: Gradually reduce learning rate (learning rate decay)
  - Utilize benefits of both high and low learning rates
  - Initially employ high learning rates to obtain good parameters quickly
  - Towards the end of the training process, employ small learning rates to converge to the minimum
  - **One option**: Step-wise decay of learning rate
    - After n epochs, learning rate is reduced by a set factor

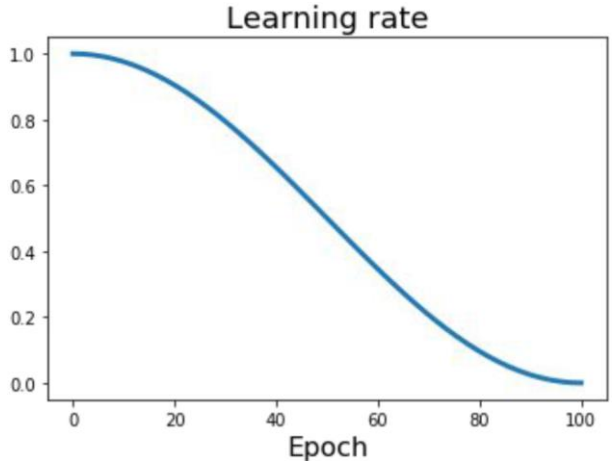


Divide learning rate by 10 after 30 epochs.
Resulting reduction in loss is easily visible.

Dynamic Learning Rates

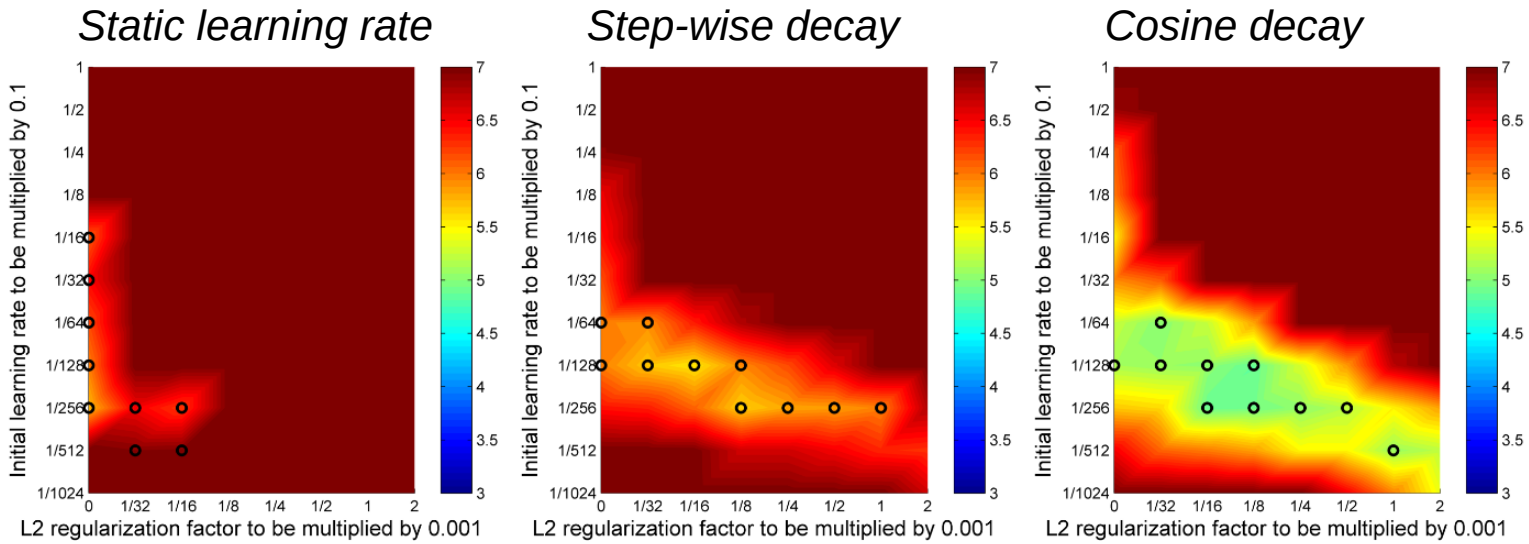
- Alternative decay methods:
  - **Linear decay**: $\alpha_t = \alpha_0(1 - \frac{t}{T})$
    - Linearly reduces learning rate in each iteration
    - α_t is value of learning rate at iteration t and T is the maximum number of steps
  - **Cosine Decay** : $\alpha_t = \alpha_0(1 + \cos\left(\frac{t\pi}{T}\right))$
    - Most used method today
- Requires a previous set ending





And with Adam?

- Adam calculates „individual learning rates“ for each parameter
  - Do we then even benefit from dynamic learning rates?
- **Yes**. Adam with dynamic learning rates achieves better results

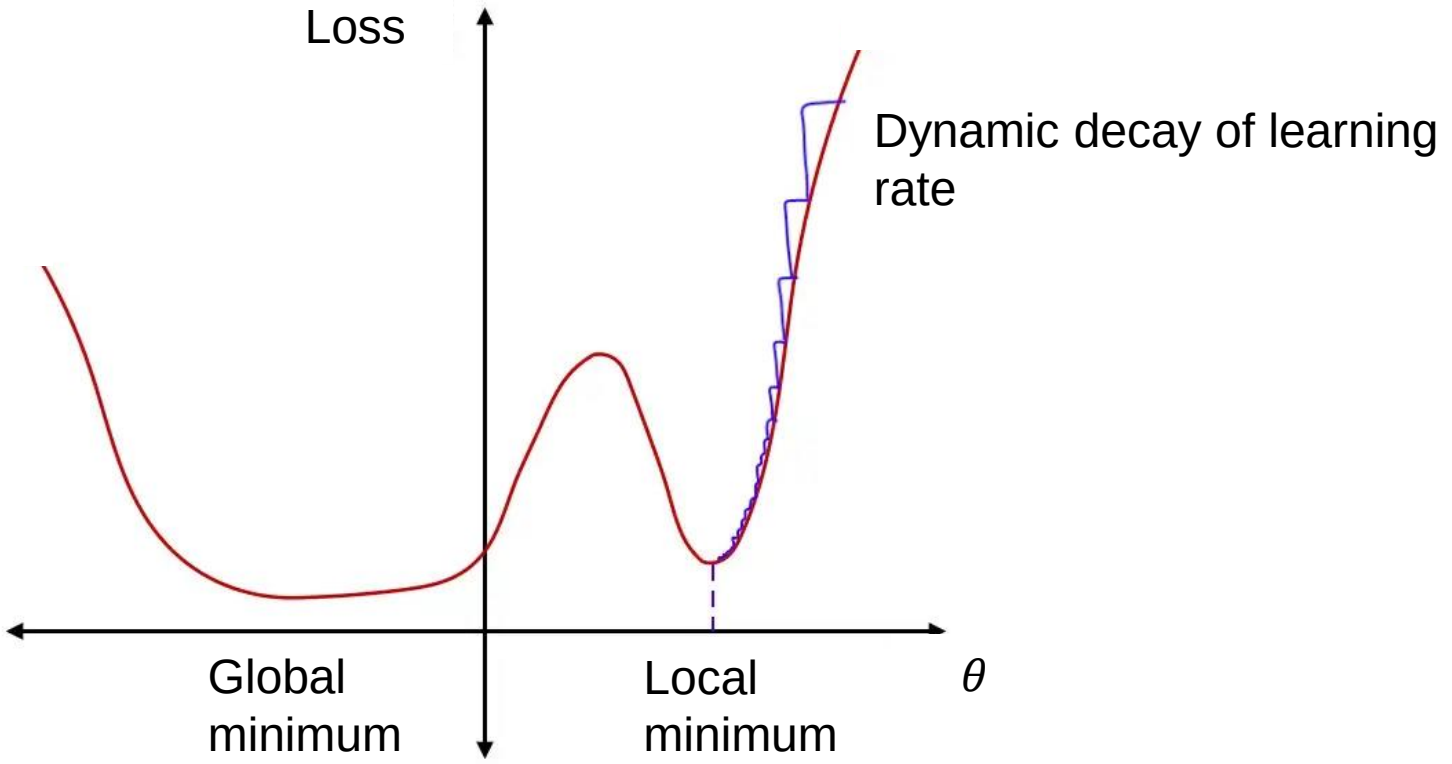


Loshchilov et. al.: *Decoupled Weight Decay Regularization*

Comparison of static with dynamic learning rates. Color shows loss for given hyperparameter combination. Cosine decay achieves generally better results for different combinations of hyperparameters

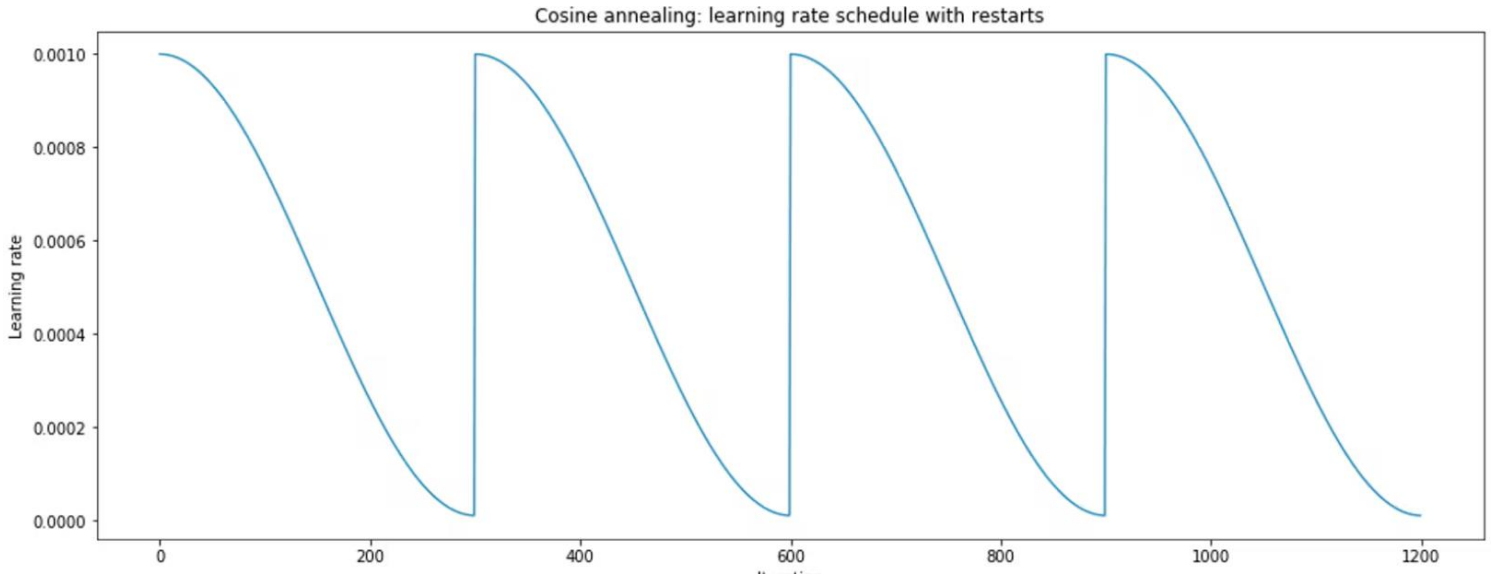
Cyclical Learning Rates

- Decay of learning rate does not prevent „getting stuck“ in local minima



Cyclical Learning Rates

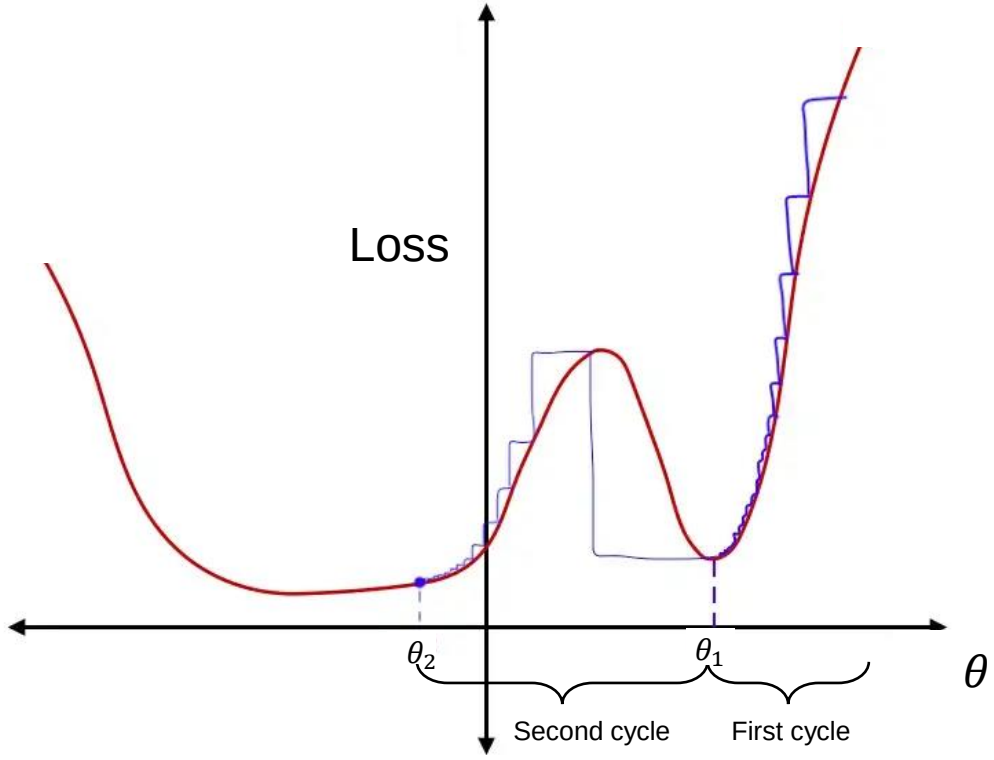
- **Idea**: Use high learning rates to escape local minima
- Use multiple cycles of decay
  - At the beginning of each cycle, the high learning rate is able to escape a possible local minimum



Cyclical Learning Rates

- Escape local minima by using high learning rates
  - At the beginning of each cycle, the loss increases because the weights jump outside local minimum
  - In the long term, however, better minima can be found
- Nowadays, regular decay is more common than cyclic learning rates

Getting stuck in a „bad“ local minima is unlikely. Active research question of why this is the case.

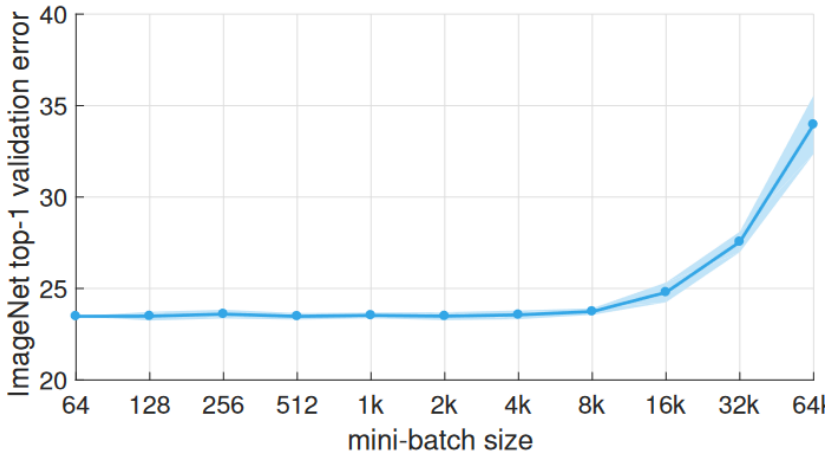


Warmup – Extension

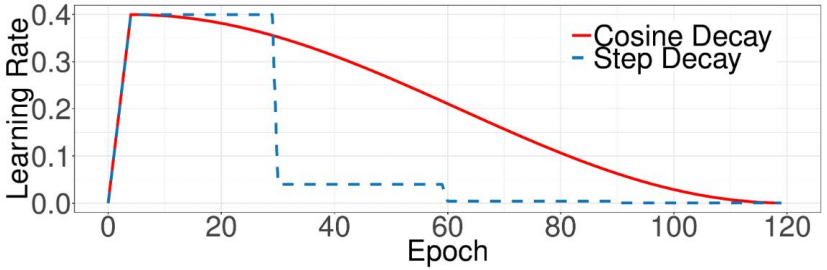
- We use small minibatches because whole dataset usually does not fit into memory
- But on supercomputers this shouldn't be a problem so can we drastically increase the minibatch-size?
  - **No:** generalization error increases
    - (the global minimum of training data is always an overfit)

小批量训练不可行的原因：
泛化误差（Generalization Error）增加

Linear Scaling Rule: When the minibatch size is multiplied by k, multiply the learning rate by k.



- **Warmup:**
  - High learning rates at the beginning are not optimal as the loss landscape is poorly explored. Gradients do not point in meaningful directions.
  - „Don't trust the gradients" too much at the beginning and gradually warmup the learning rate before starting the decay



[2024] Why Warmup the Learning Rate

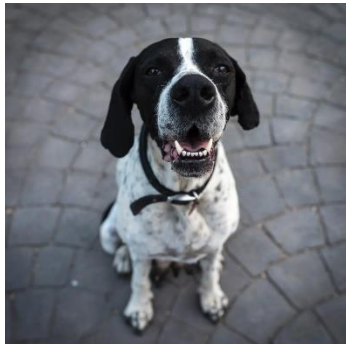
Overview

- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

Overfitting/Generalization of Neural Networks

- **Experiment**: Training and test datasets containing million images are given random (wrong) labels

Car



Cat

Dog

Dog

  - Generalization impossible because no correlation exists between image and random label
    - In training dataset, neural networks achieve $> 95\%$ accuracy.
      - In test $\sim 0.001\%$ accuracy
    - This shows that, neural networks can memorize (overfit) the labels of million of images
- **Why doesn't this happen with true labels?**
  - The following slides are an attempt, but do not sufficiently answer this fundamental question.

Regularization for Neural Networks

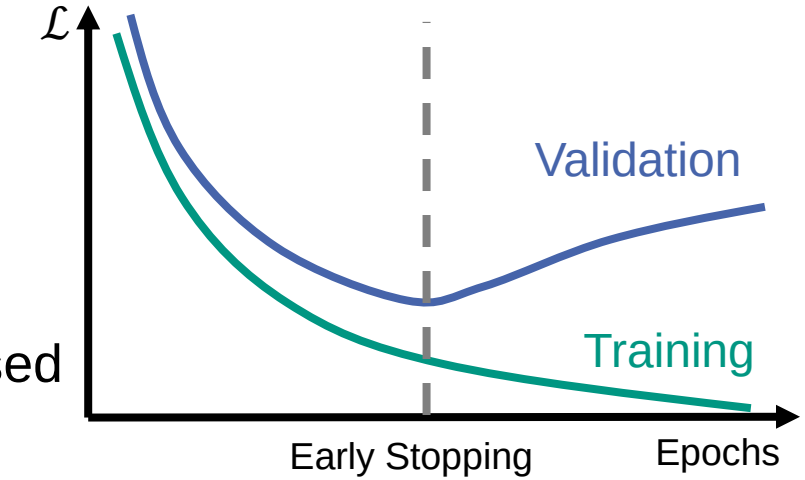
- **Critical Problem:** Create models that perform well not only on training data, but also on unseen data

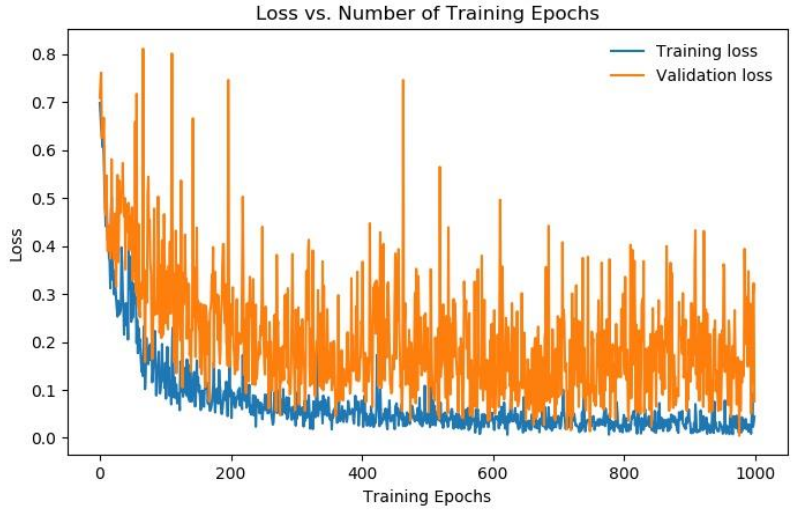
- **Definition Regularization**:
  Strategies that aim to reduce generalization error, possibly at the expense of increased training error.

- **Methods**
  - Early Stopping
  - Parameter regularization (Weight Decay)
  - Dropout
  - Data augmentation
  - …

Early Stopping

- Stop training if loss increases on validation data
  - **Problem**: There are no smooth loss curves.
    - **Solution**
      - Calculate validation loss after each epoch
      - **1.** If for n epochs, the validation loss has increased or not decreased, stop training.
      - **2.** Finish training, but store validation loss and model weights after each epoch.
        Use model weights with smallest validation loss





Parameter Regularization (Weight Decay)

- **Intuition**: Large parameter values mean that small changes in the input have a large effect on the output, which can be a sign of overfitting.
- **Idea**: Penalise large parameter values by adding the parameter norm $R(\boldsymbol{\theta})$ to loss function $\hat{\mathcal{L}}(\boldsymbol{\theta})$
- **Regularized Loss**:

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}) = \underbrace{\hat{\mathcal{L}}(\boldsymbol{\theta})}_{\text{Original Loss}} + \underbrace{\lambda R(\boldsymbol{\theta})}_{\text{Regularization Term}}$$

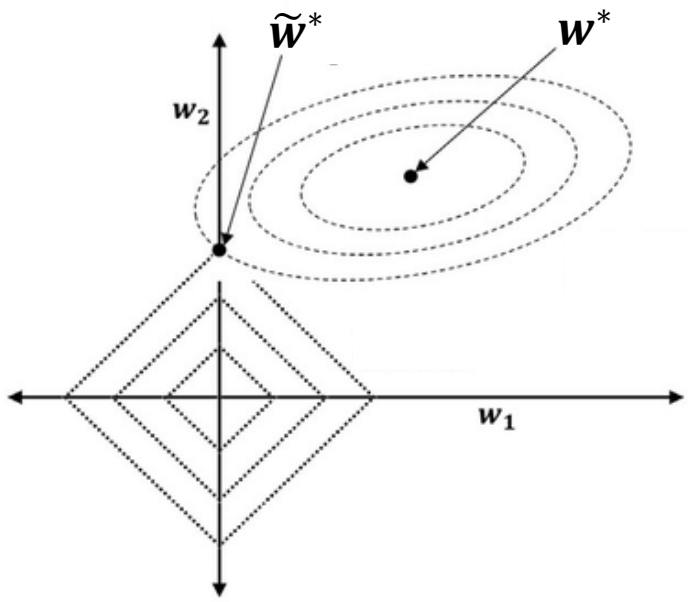
- **Hyperparameter** $\lambda \in [0, \infty)$: Controls the influence of regularization
- **Effect**: During optimization, parameters that both reduce the original loss function and reduce their parameter norm are preferred.

- **Important**: In $R(\theta)$ only the parameters $\boldsymbol{W}^{(l)}$ are considered, Bias $\boldsymbol{b}^{(l)}$ remains unregularized
  - Less likely that bias is the cause of an overfit and if regularized can lead to severe underfitting

Parameter Regularization – L^1 & L^2

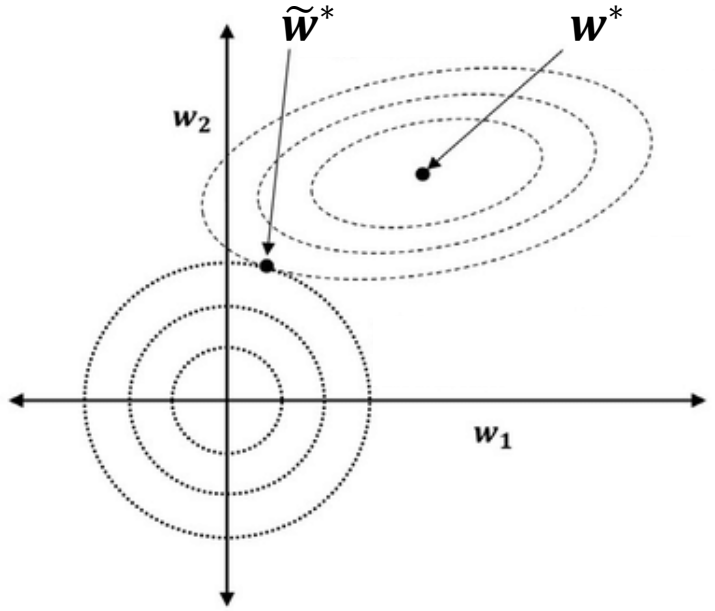
L^1 Regularization

- $R(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|$



L^2 Regularization

- $R(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$



Important:
$\boldsymbol{w}$ considers all parameters in all layers

Parameter Regularization – Example

- Input: $\mathbf{x} = [1 \quad 1 \quad 1 \quad 1]^T$
- Weight vector 1: $\boldsymbol{w}_1 = [1 \quad 0 \quad 0 \quad 0]$
- Weight vector 2: $\boldsymbol{w}_2 = [0.25 \quad 0.25 \quad 0.25 \quad 0.25]$
- Output: $\boldsymbol{w}_1 \boldsymbol{x} = \boldsymbol{w}_2 \boldsymbol{x} = 1$

- Question:
  - Which vector produces the lower regularization cost $R(\boldsymbol{\theta})$, $\boldsymbol{w}_1$ or $\boldsymbol{w}_2$ for L_1 regularisation?

$$R_{L1}(w_1) = R_{L1}(w_2)$$

  - What about L_2 regularization?

$$R_{L2}(w_1) > R_{L2}(w_2)$$

Parameter Regularization – L^1 & L^2

L^1 Regularization

- $R(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|$
- Results in sparse results
  → Some $w_i = 0$

可以自动忽略不重要的特征，仅保留对模型输出有显著影响的特征

- **Advantage**
  - **Feature selection**: Unimportant features are set to zero and ignored
- **Disadvantage**
  - Constant factor that does not scale with specific values of w_i

可以自动忽略不重要的特征，仅保留对模型输出有显著影响的特征

L^2 Regularization

More in depth in Goodfellow: Deep Learning Book Chapter 7.1

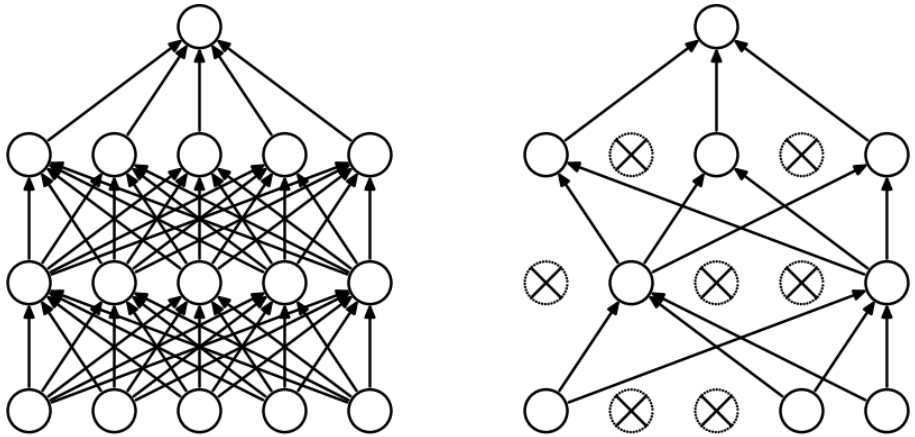
- $R(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$
- Backpropagation results in

$$\theta_{t+1} = \underbrace{(1-\alpha\lambda)}_{\text{Shrink weight in each iteration}}\underbrace{\theta - \alpha\nabla_\theta\hat{\mathcal{L}}(\theta)}_{\text{Same as before}}$$

- **Advantage**
  - Parameters that contribute significantly to output are preserved relatively intact
  - Weights that do not significantly increase the gradient are reduced
  - AdamW decouples weight decay from adaptive learning rates

Dropout

- **Reminder - Bagging (Bootstrap Aggregation)**: Reduces variance/overfitting by averaging multiple hypotheses
  - **Bootstrapping**: Create from D, m datasets D_1 to D_m with same number of instances and with layback
  - **Training**: Train new model h_{D_1} to h_{D_m} for each dataset
  - **Aggregation**: Combine models by e.g. averaging predictions
- **Challenges** of bagging deep neural networks:
  - Training usually takes longer and requires much more hardware resources
- **Solution**: Approximative and implicit realisation
  - Create inside a large model, multiple subnets

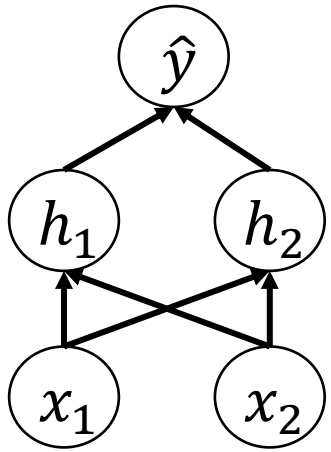


(a) Standard Neural Net (b) After applying dropout.

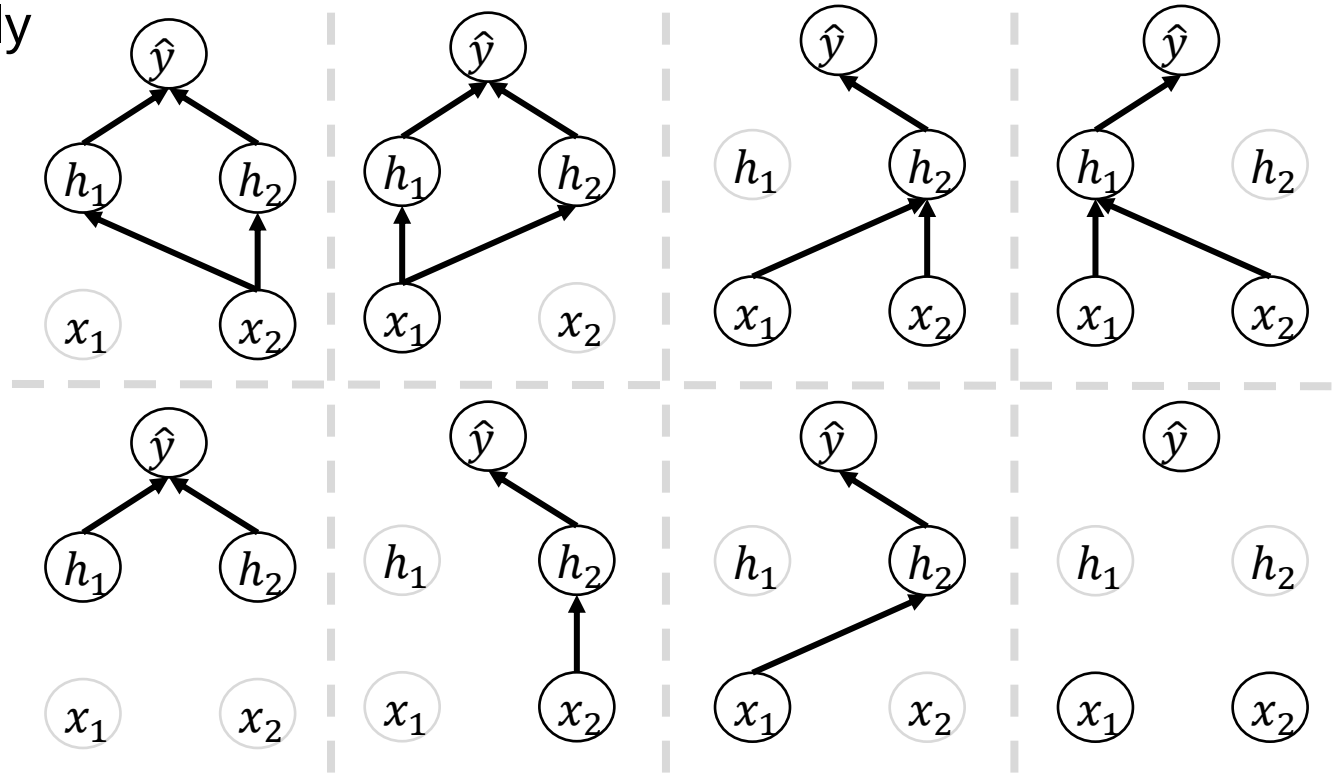
- **Dropout**: Randomly deactivate neurons (and their connections) during training with probability p.

Dropout

- Masking (deactivation) of neurons during learning
  - (Usually multiply the outputs of the deactivated neuron by zero)
- → learn all subnets simultaneously
- Each sample inside minibatch samples neurons for dropping



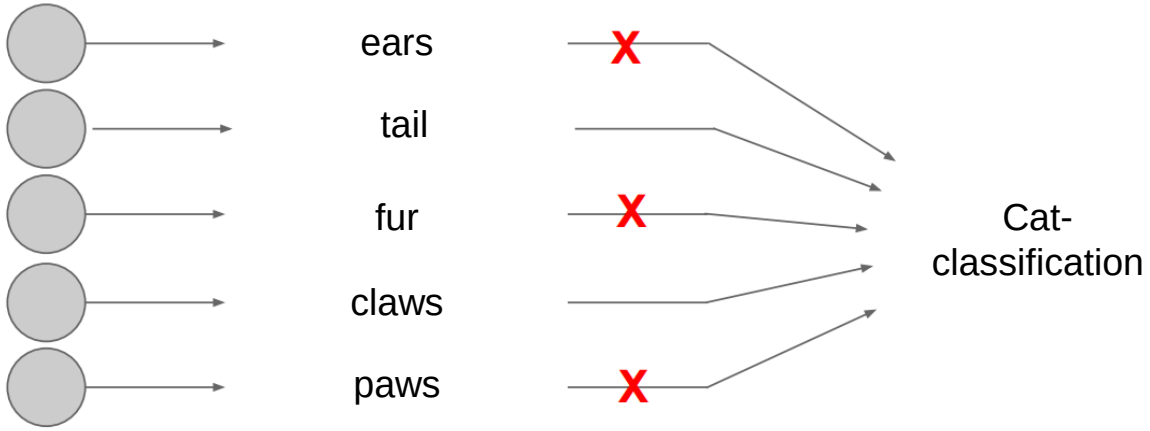
Full network



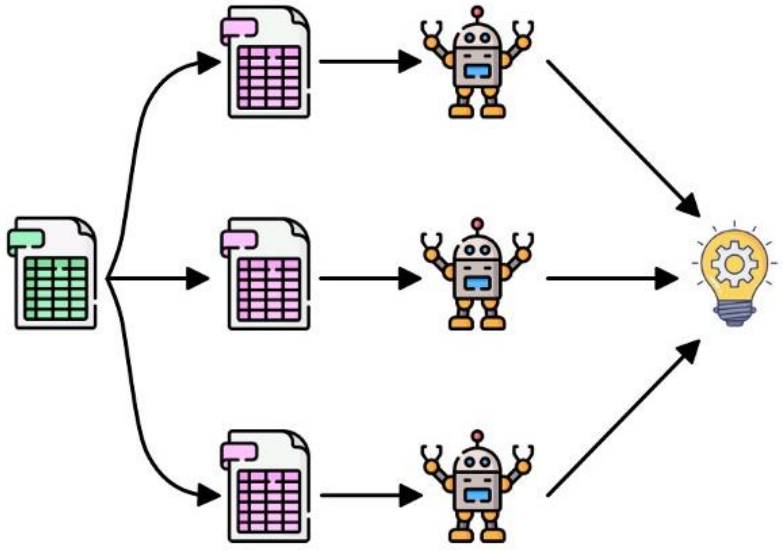
Creating subnets by deactivating neurons

Dropout – Interpretation

- Prevents an overspecification of neurons and promotes a redundant representation of neurons



- Training of a large ensemble of different neural networks that share parameters, i.e. the models are not independent of each other.
  - Different to bagging, there all models might share the training data, but do not share their parameters



Dropout – Implementation during Training

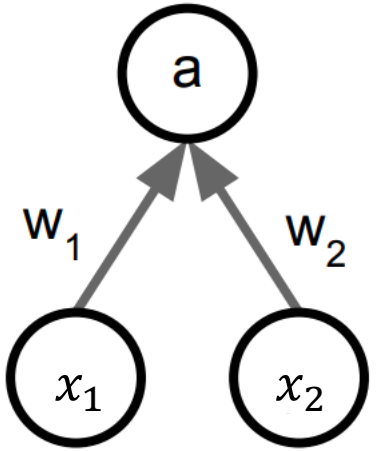
- A masking $\boldsymbol{\mu} \sim p(\boldsymbol{\mu})$ (inactivation) of the neurons is defined for each sample in minibatch
- (Usually, we choose $\mathrm{p} = 0.5$ to deactivate a given neuron)
- Backpropagation averages gradients of all different subnets used in minibatch and updates the weights.

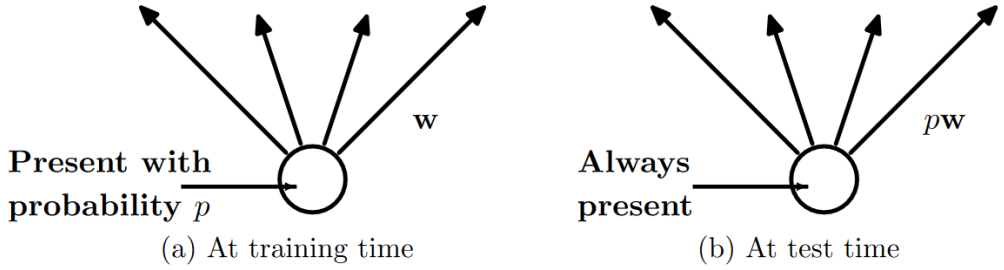
Dropout – Implementation during Testing

- How to combine the different outputs of our subnets?
  - $\hat{\boldsymbol{y}}^{(i)} = h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}, \boldsymbol{\mu})$
- During inference, we want to make one final prediction.
- **Idea**: $\hat{\boldsymbol{y}} = \mathbb{E}_{\boldsymbol{\mu} \sim p(\boldsymbol{\mu})}\left[h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}, \boldsymbol{\mu})\right] = \int_{\boldsymbol{\mu}} h_{\boldsymbol{\theta}}\left(\mathbf{x}^{(i)}, \boldsymbol{\mu}\right) p(\boldsymbol{\mu}) d\boldsymbol{\mu}$
- **Problem**: Solving this integral is very complex
  → Let's approximate it

Dropout – Implementation during Testing

- **Now**: Consider each neuron separately
- **During Training**
  - For $p = 0.5$ (each neuron has a 50% chance of being dropped)
  - $\mathbb{E}[a] = \frac{1}{4}(\mathrm{w}_1 x_1 + \mathrm{w}_2 x_2) + \frac{1}{4}(\mathrm{w}_1 x_1 + 0 x_2) + \frac{1}{4}(0 x_1 + \mathrm{w}_2 x_2) + \frac{1}{4}(0 x_1 + 0 x_2)$
  - $\mathbb{E}_{train}[a] = \frac{1}{2}(w_1 x_1 + w_2 x_2)$
- **Testing**
  - $\mathbb{E}_{test}[a] = \mathrm{w}_1 x_1 + \mathrm{w}_2 x_2$
- **Solution:** During inference/testing, multipy the output of each neuron with its dropout probability p
  - **More common**: Divide by p during training





(a) At training time (b) At test time

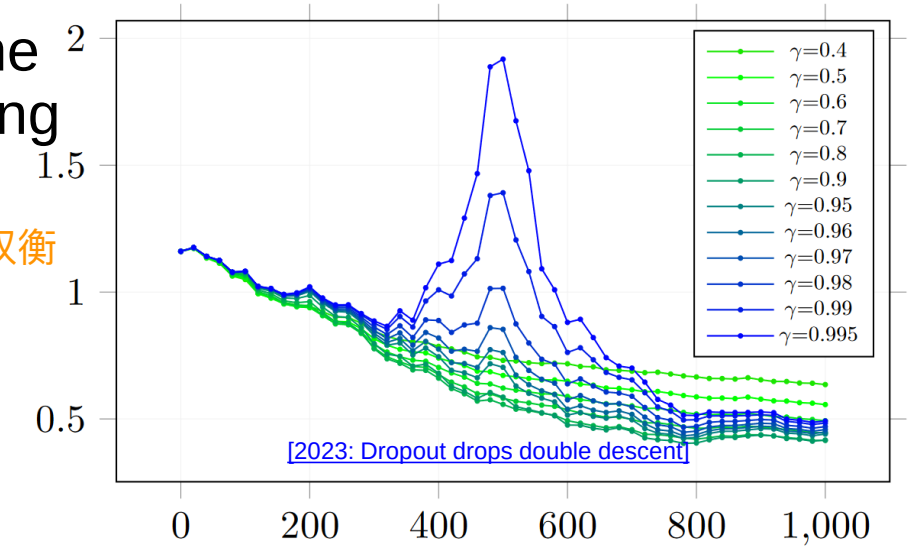
Regularization Prevents Double Descent

这个现象可以类似于过拟合阶段，可理解为过拟合造成的

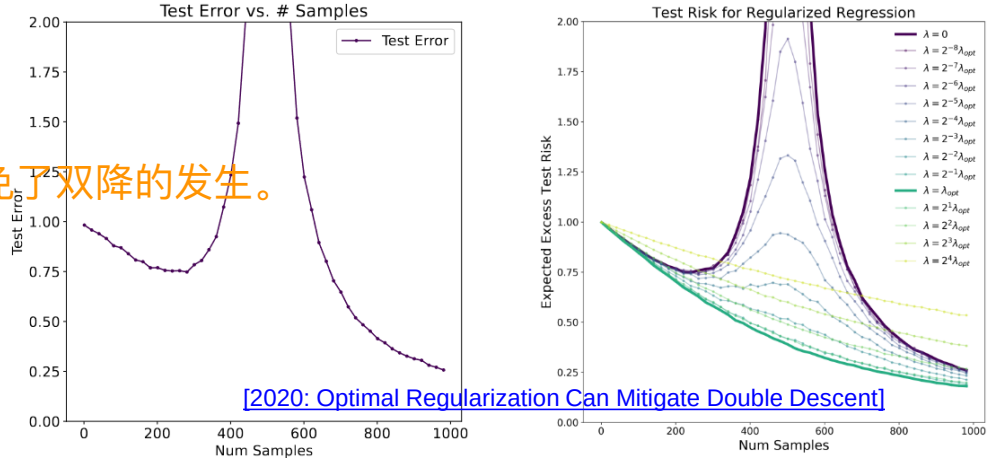
- **Traditional U-shape for learning:** At the beginning, the loss is decreasing. After a while the loss starts increasing again.
  Theory credits bias-variance tradeoff 可归结于此，偏差-方差权衡
- **Double Descent:** Loss starts decreasing again, final loss is lower than the minimum in the u-shape
- **Dropout drops double descent**
  - $\gamma = 1 - p =$ probability we keep the neuron

  Dropout 通过随机失活部分神经元，减少过拟合现象，从而避免了双降的发生。
- Similar observations have been made for **weight decay**

权重衰减(weight decay)也可以抑制double descent



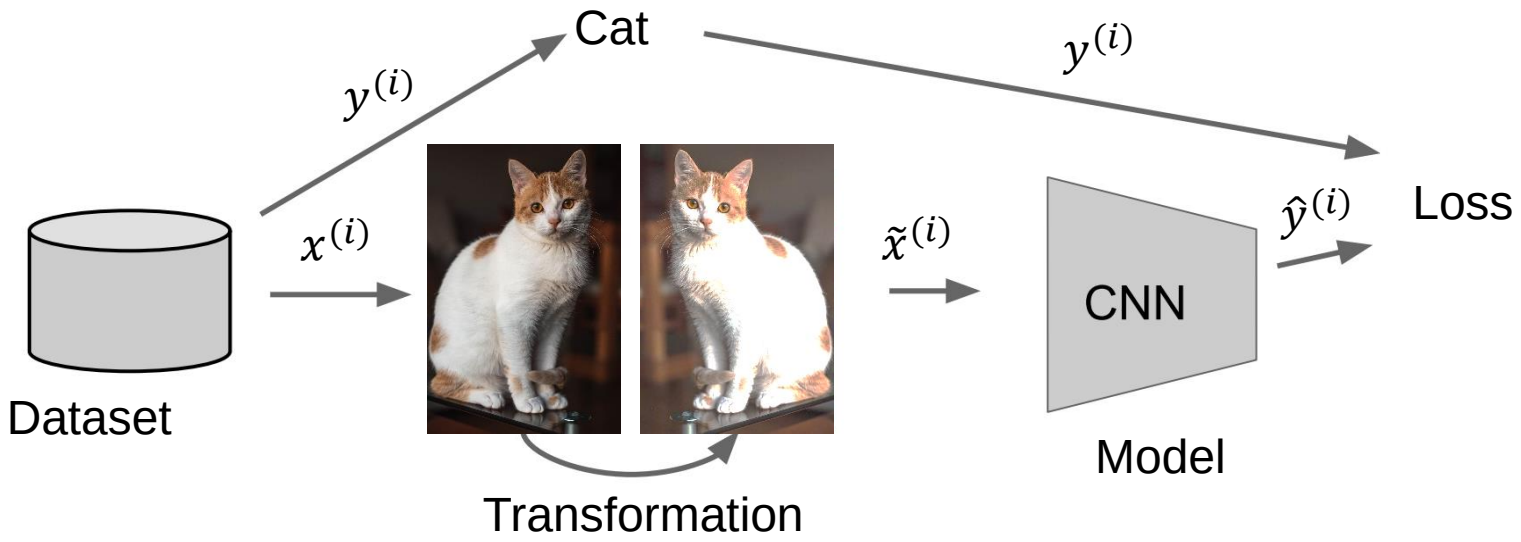
[2023: Dropout drops double descent]



[2020: Optimal Regularization Can Mitigate Double Descent]

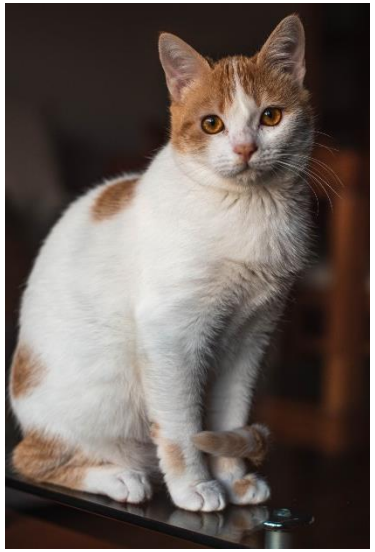
Data Augmentation 数据增强

- **Data Augmentations**: Methods to synthetically but plausibly increase the size and diversity of the dataset.
- **Objective**: Reduce overfitting. As the size of the data set increases, it becomes more difficult for the model to memorize the input data.
  - → Modell is forced to generalize

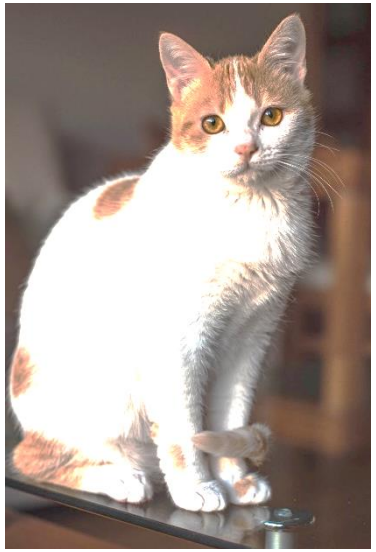


Data Augmentation

- **Challenges**: Data augmentation cannot be used for every problem and must be adapted to the specific domain.
- **Example: Image classification**:
  - Brightness & Contrast
  - Translation
  - Rotation
  - Distortion
  - Mirroring
  - Cutouts
  - Combinations of these methods
  - …

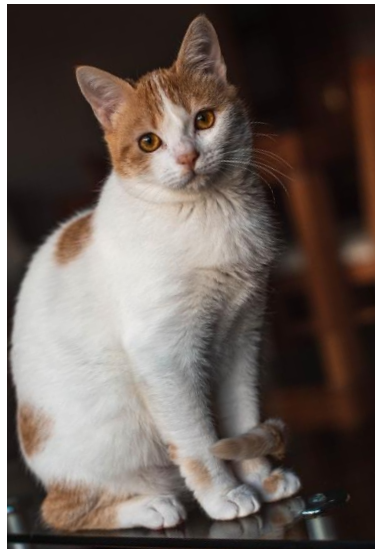


Original



Brightness & Contrast

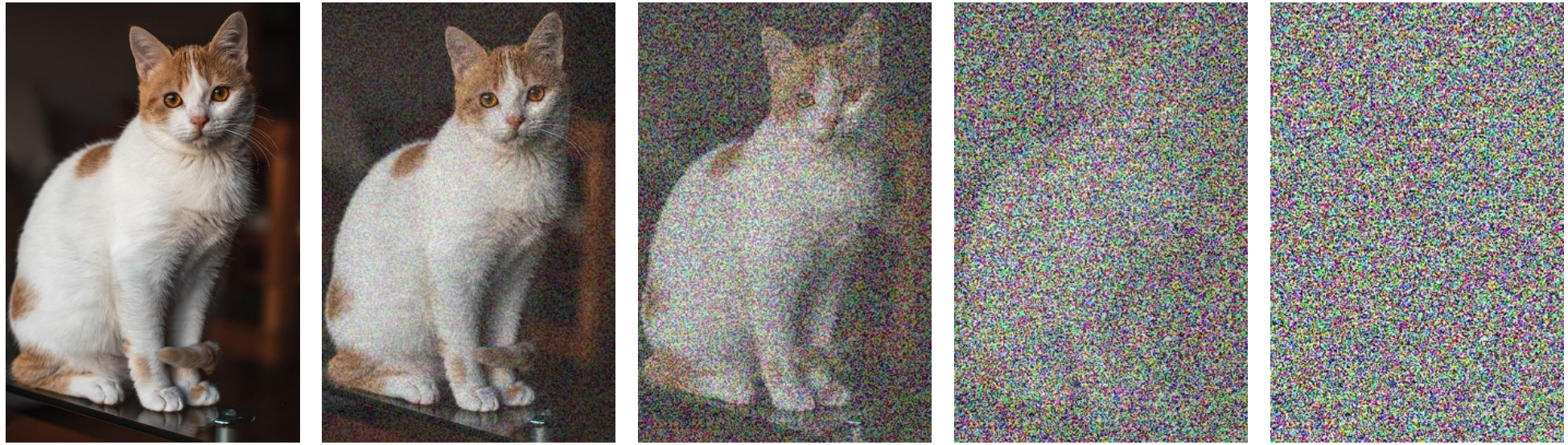
Translation



Rotation

Data Augmentation

- **General Augmentation: Adding Noise**: Increases robustness of models
- **Example Gaussian Noise**: Input: $\tilde{\boldsymbol{x}}^{(i)} = \boldsymbol{x}^{(i)} + \boldsymbol{\epsilon},\ \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \eta \boldsymbol{I})$



Data Augmentation

标签必须保持正确，否则会引入误差

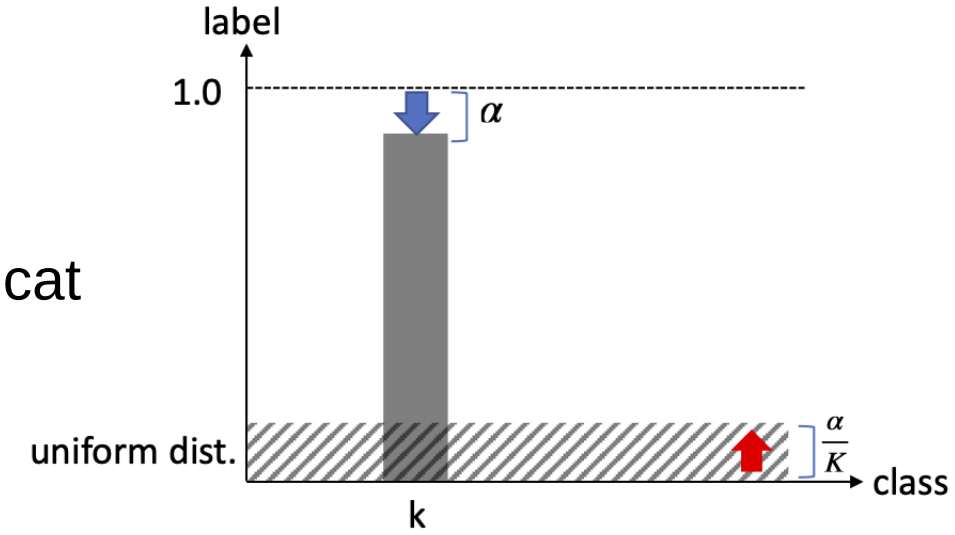
- **Problems**: Data augmentation may only change the data to such an extent that the original meaning of the data is not altered! The labels must still be correct for the changed data.
- **Example: Number classification**:
  - Original image: $x^{(i)}$
  - Label: $y^{(i)} = 6$
  - Transformation: Rotation 180°
  - Augmented image: $\tilde{x}^{(i)}$
  - Label: $y^{(i)} = 6$
  - Label should actually be $\tilde{y}^{(i)} = 9$
- **Example from our research**
  - Autonomous driving
    - Vertical mirroring of images: Network learns left-hand traffic in Karlsruhe

$(x^{(i)}, y^{(i)} = 6)$

$(\tilde{x}^{(i)}, y^{(i)} = 6)$

Label Smoothing

- Networks that have memorised the training data are very confident because they already know the solution.
  - Generalized networks are less confident
- Penalize network if it is too confident in its prediction during training
- Add „mistakes“ to labels
  - Example: Label is 95% cat and 5% dog
  - Loss is increased, if network predicts 100% cat
    - Keeps network from being to confident



[2020: When does Label Smoothing Help?]

Overview

- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

Parameter Initialization

- Why is initialization important?
  - Iterative optimizing methods like gradient descent need an **initial starting point** for their parameters
  - **Training** deep neural networks is a difficult task and can be **highly dependent** on this initialisation
  - Parameter initialisation can determine **how quickly** training converges, or **whether it converges at all.**

Exploding/Vanishing-Gradient

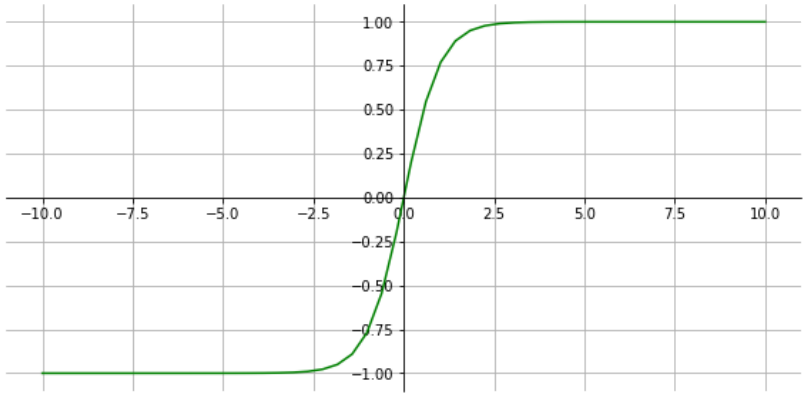
梯度爆炸

- **Exploding Gradient**: Gradient is very large and we take huge steps even with small learning rates.
  - This leads to diverging or instable training.
  - **Gradient Clipping:** Can be prevented by setting a maximum value and reducing the gradient to that value if it exceeds it.
- **Vanishing Gradient**: Gradient is close to zero. The weights are not updated.
  - No training is possible, because weights won‘t change
  - Generally harder to solve than exploding gradients

梯度消失

Parameter Initialization – Desired Properties

- **Objective**: Certain features of the activation distribution should be upheld at the start and throughout training for all shifts!
- **Careful**: Initialization is dependent on activation functions and network architecture
- **Desired properties**
  - Reminder - layer: $\boldsymbol{a}^{(l)} = \boldsymbol{\sigma}\left(\boldsymbol{h}^{(l)}\right), \quad \boldsymbol{h}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$
  - Break symmetries between neurons
  - Keep inputs for activation function in „workable“ space
  - $\mathbb{E}\left[\boldsymbol{a}^{[l-1]}\right] = \mathbb{E}\left[\boldsymbol{a}^{[l]}\right] \approx 0$
  - $Var\left(\boldsymbol{a}^{[l-1]}\right) = Var\left(\boldsymbol{a}^{[l]}\right) > 0$



tanh()

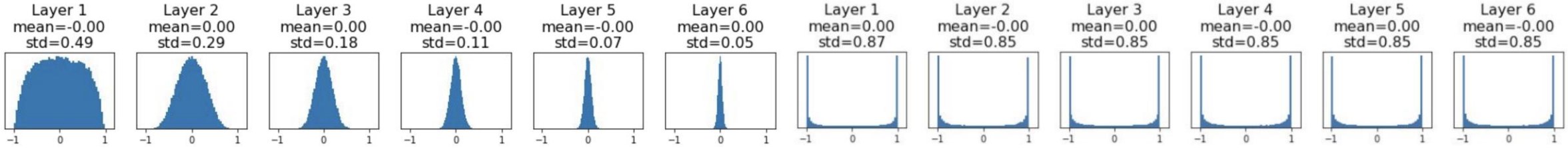
Parameter Initialization – Vanishing Gradient

梯度接近于零，这是由于小权重的乘积导致的。

Example 1

- **NN Architecture**: 6 layers with $\tanh()$ activation functions and random inputs
- Initialization with small weights $\theta_i \in \mathcal{N}(0, \boldsymbol{I} \cdot 0.01)$
- The standard deviation of the outputs collapse to zero for later layers $\boldsymbol{a}^{(l)}$
- **Gradients**: Close to zero (multiplication of small weights)

$$\frac{\partial \mathcal{L}}{\partial w_o} = \frac{\partial \mathcal{L}}{\partial a_6} \frac{\partial a_6}{\partial h_6} \frac{\partial h_6}{\partial a_5} \frac{\partial a_5}{\partial h_5} \frac{\partial h_5}{\partial a_4} \dots$$



梯度接近于零，这是由于 tanh() 的导数在饱和区域非常小。

Example 2

[Code]

- **NN Architecture**: 6 layers with $\tanh()$ activation functions and random inputs
- Initialization with larger weights $\theta_i \in \mathcal{N}(0, \boldsymbol{I} \cdot 1)$
- $\boldsymbol{a}^{(l)}$ saturates the $\tanh()$
- **Gradients**: Close to zero (multiplications of small derivatives of $\tanh()$)

$$\frac{\partial \mathcal{L}}{\partial w_o} = \frac{\partial \mathcal{L}}{\partial a_6} \frac{\partial a_6}{\partial h_6} \frac{\partial h_6}{\partial a_5} \frac{\partial a_5}{\partial h_5} \frac{\partial h_5}{\partial a_4} \dots$$

Parameter Initialization – Xavier

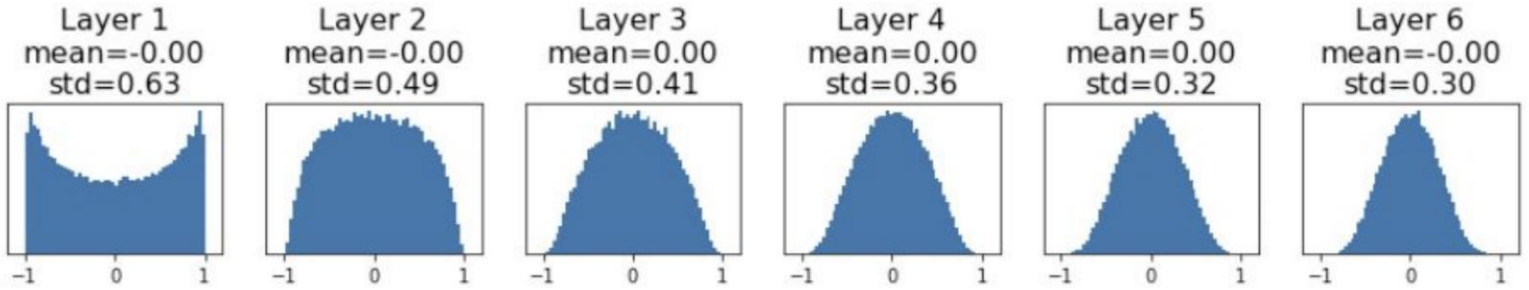
一种参数初始化方法，规避梯度消失/爆炸

- **Xavier – Initialization** (Xavier Glorot & Yoshua Bengio)
  - Optimized initialization of parameters that prevents vanishing and exploding gradients
  - **Activation Function**: Xavier is suitable for $\tanh()$ or $sigmoid()$ 只适用于tanh,sigmoid激活函数
  - **Objective**: Keep variance of activation function constant for each layer (variance of input = variance of output)
    - Variance increases or decreases by adding or removing neurons of layer
  - **Intuition**
    - Fewer incoming neurons to the layer → Higher weights to achieve the same neuron output.
    - More incoming neurons to the layer → Smaller weights to achieve same neuron output.

即当神经元过少时，加大权重
当神经元数量过多时，减小权重

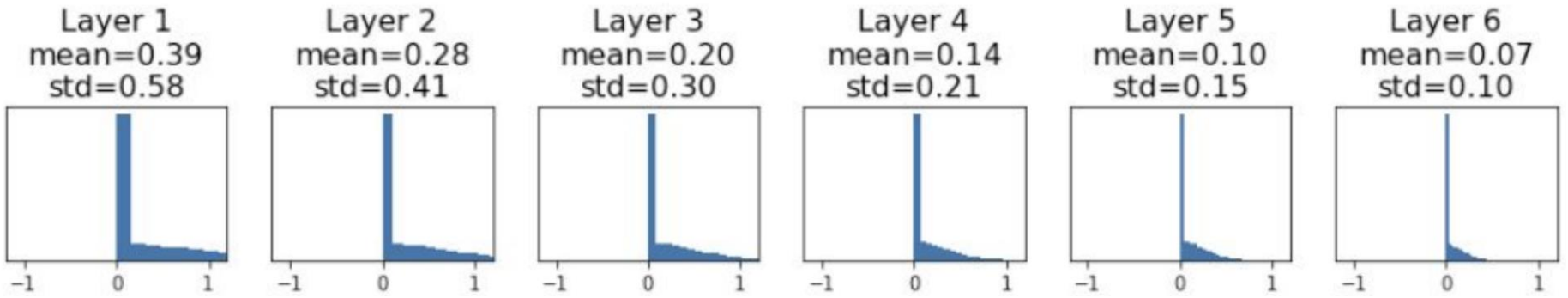
Parameter Initialization – Xavier

- **Xavier – Initialization** (Xavier Glorot & Yoshua Bengio)
  - Initialize bias $\boldsymbol{b}^{(l)}$ with 0
  - Initialize $\boldsymbol{W}^{(l)}$ from Gaussian: $w_{ij}^{(l)} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \sqrt{\frac{2}{n_{in}^{(l)} + n_{out}^{(l)}}}^2\right)$
    - Whereas $n_{in}^{(l)}$ is number of inputs of neuron and $n_{out}^{(l)}$ the number of output neurons of layer l



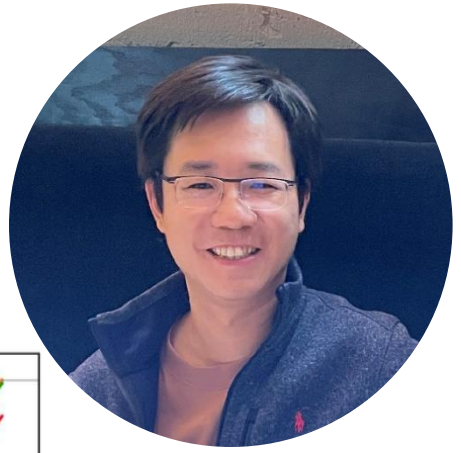
Parameter Initialization – Xavier

- **Xavier – Problem**
  - Not usable for all activation functions
  - **Previous**: NN architecture - 6 layers with $\tanh()$ activation function
  - **Now**: NN architecture - 6 layer with **ReLU activation function**
  - **Problem**: Variance of output of the activation function $\boldsymbol{a}^{(l)}$ decreases
    - ReLU sets about half of the outputs to zero → variance is halved for each layer

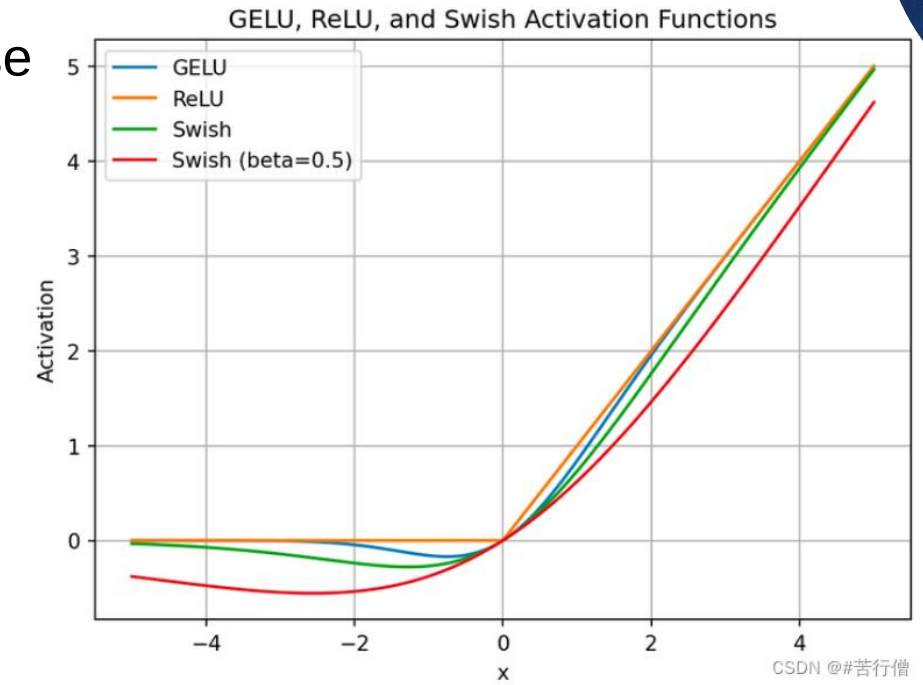


Parameterinitialisierung – Kaiming He

借此He初始化，可以使用ReLU激活函数

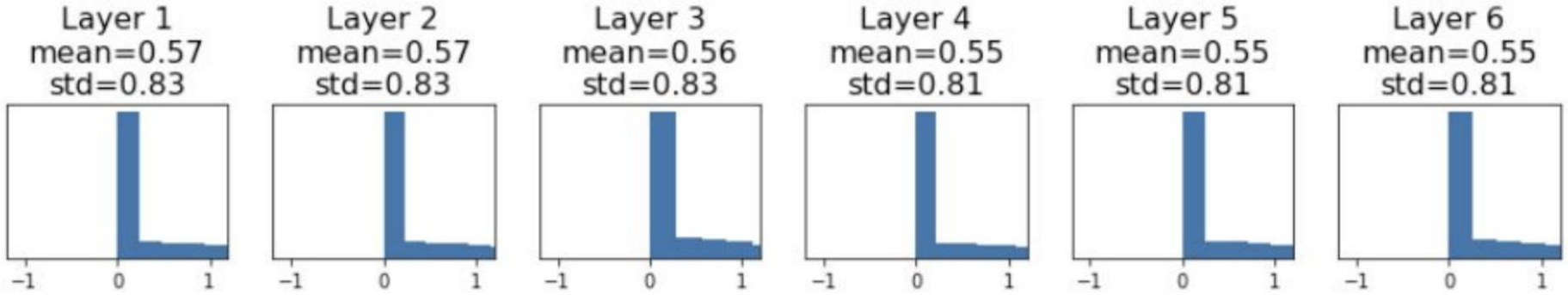


- **Kaiming He – Initialization** (after Kaiming He)
  - Considers the properties of ReLU
  - **Intuition:** If ReLU halves the variance in each layer, then manually increase the variance by that factor.



Parameterinitialisierung – Kaiming He

- **Kaiming He – Initialization**
  - Initialize bias $\boldsymbol{b}^{(l)}$ with 0
  - Initialize $\boldsymbol{W}^{(l)}$ from Gaussian: $w_{ij}^{(l)} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \sqrt{\frac{2}{n_{in}^{(l)}}}^2\right)$
- **Careful:** Tensorflow uses Xavier initialization by default, although almost all networks today use ReLU or ReLU derivatives



Overview

- Recap – Neural Networks
- Motivation
- Optimization Algorithms
- Learning Rate
- Regularization
- Parameter Initialization
- Tips and Tricks

Tips and Tricks

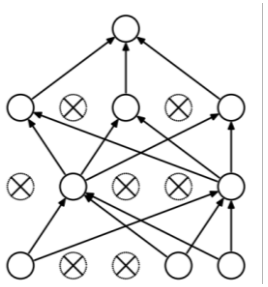
- Use existing **frameworks** for neural networks
  - We recommend PyTorch (but TensorFlow/Jax are also good)
- Initially use **AdamW** optimizer (Adam with weight decay)
  - At the beginning with static learning rate (e.g. $3e-4$)
  - Try out different static learning rates.
  - Implement dynamic learning rates only after successes with static learning rates
- Use **ReLU** or **ReLU-like** activation functions
  - $sigmoid$ and $\tanh$ functions are too disadvantageous for training
  - And then initialize weights with **Kaiming-He**
    - Or better: **Transfer learning**

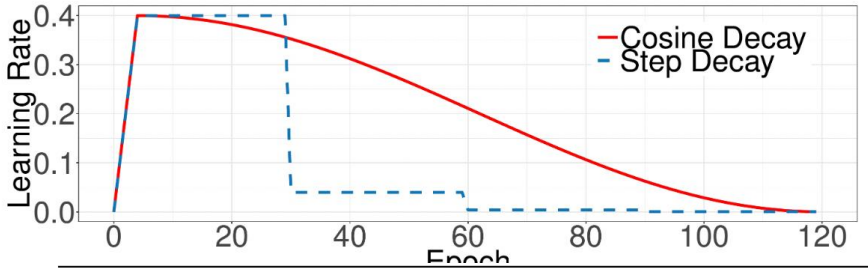
Tips and Tricks

- **Supervise** training and validation loss
  - Makes it possible to recognize problems early
- Initially use **very small subset of dataset for training** and check if model reaches ~100% training accuracy
  - If the neural network is unable to memorise a tiny data set, then something is usual incorrectly
    - Layers not connected, no activation function, labels do not fit to output of model
    - Forgot to update the weights
    - Accumulates gradients over each training iteration instead of calculating new gradients for each minibatch
    - Learning rate too high or too low
    - Network produces NaNs in weights/gradients/outputs
- Use **GPU** to decrease training time

Let‘s revisit RoBERTa

- Dropout
- Warmup
- Peak Learning Rate
- Batch Size
- Weight Decay
- Max Steps
- Learning Rate Decay
- Adam $\epsilon, \beta_1, \beta_2$
- Gradient Clipping





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

Hyperparam	RoBERTa$_{\text{LARGE}}$
Number of Layers	24
Hidden size	1024
FFN inner hidden size	4096
Attention heads	16
Attention head size	64
Dropout	0.1
Attention Dropout	0.1
Warmup Steps	30k
Peak Learning Rate	4e-4
Batch Size	8k
Weight Decay	0.01
Max Steps	500k
Learning Rate Decay	Linear
Adam ϵ	1e-6
Adam β_1	0.9
Adam β_2	0.98
Gradient Clipping	0.0

Covered in ML2

Covered in ML1

Literature

- Ian Goodfellow and Yoshua Bengio and Aaron Courville: ***Deep Learning***, *MIT Press, 2016,* *https://www.deeplearningbook.org/*
- François Fleuret: *The little book of Deep Learning 2023* *https://fleuret.org/public/lbdl.pdf*
- Loshchilov: *Decoupled Weight Decay Regularization*, 2019
- Xavier Glorot: *Understanding the difficulty of training deep feedforward neural networks* 2010
- Kaiming He: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* 2015
- Diederik P. Kingma: *Adam: A Method for Stochastic Optimization* 2015