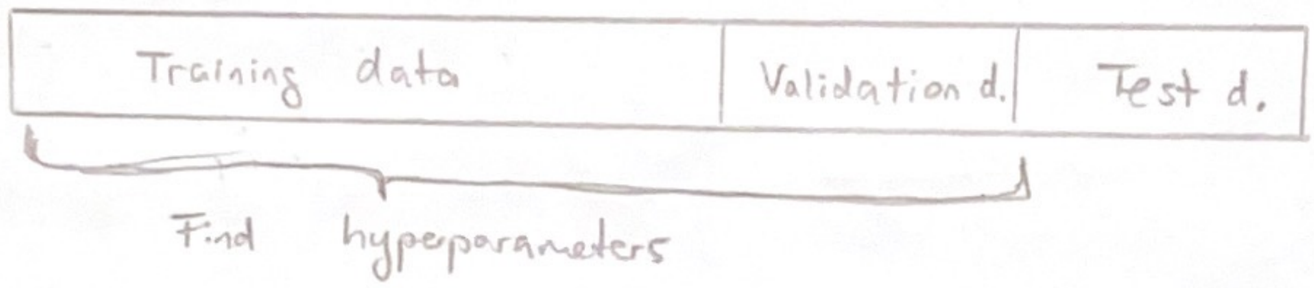
12DL 2: ML Basics

- ML: Unsupervised vs. Supervised Learning:
  - Unsupervised: * No Labels; * Find structure of the data; * Clustering
  - Supervised: * Labels
  - Reinforcement Learning: Agent ⇄ Env (Interaction / reward)

- Properties of k-NN:
  - → Hyperparameters
    - Distance metric: L_1-Norm, L_2-Norm
    - No. of neighbors k
  - → Values are Problem dependent

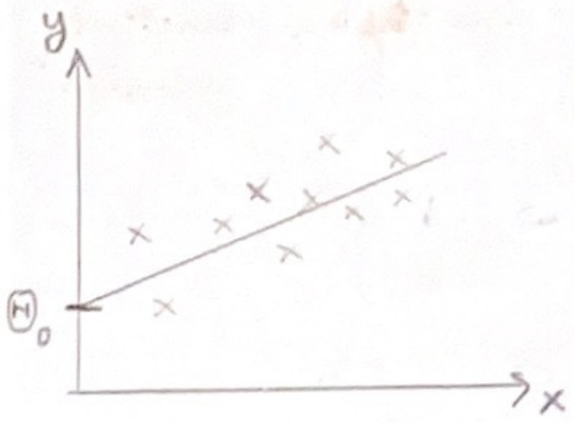


- What is Cross Validation?
  - → Split the ~~data~~ training data into N folds
  - ⇒ Train on N-1 folds and validate on 1 fold

- Linear Regression: Find a linear model that maps inputs x to target y.

Training: $\{\vec{x}_{1:n}, \vec{y}_{1:n}\} \rightarrow$ Learner $\rightarrow \textcircled{H}$ (Model parameters)
Data points

Testing: $\vec{x}_{n+1}, \textcircled{H} \rightarrow$ Predictor $\rightarrow \hat{y}_{n+1}$
new sample → new label



Linear model:

$$\hat{y}_i = \textcircled{W}_0 + \sum_{j=1}^{d} x_{ij} \textcircled{W}_j$$

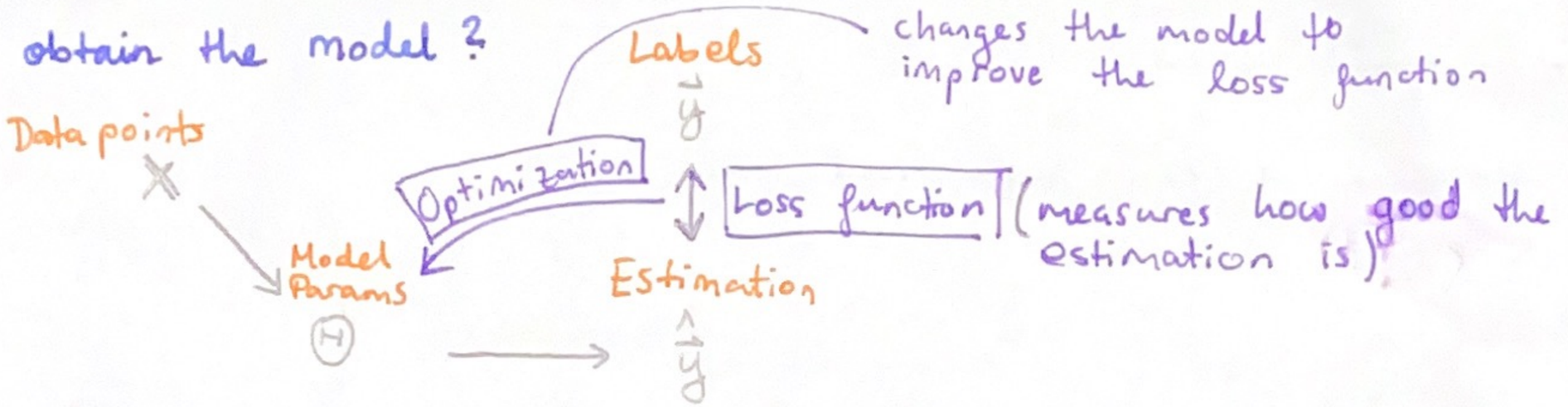
$\hat{y}_i$: Prediction for ith sample; $\textcircled{W}_0$: Bias; x_{ij}: Input sample; $\textcircled{W}_j$: weights (for each feature)

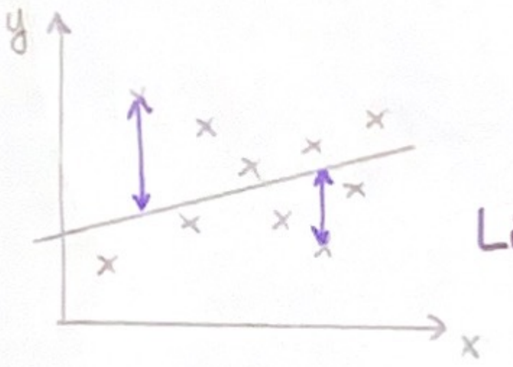
$$\Rightarrow \hat{\vec{y}} = X\vec{\textcircled{W}} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix} \begin{bmatrix} \textcircled{W}_0 \\ \textcircled{W}_1 \\ \vdots \\ \textcircled{W}_d \end{bmatrix}$$

Input sample with d features

Model parameters (d weights + 1 bias)

How to obtain the model ?





Lin. Regression loss function: (Objective function / Cost function)

$$J(\vec{\Theta}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Optimization: (Least squares estimate)

$$\min_{\vec{\Theta}} J(\vec{\Theta}) = \frac{1}{n}\sum_{i=1}^{n}(\vec{x}_i^T\vec{\Theta} - y_i)^2$$

$$= (X\vec{\Theta} - \vec{y})^T(X\vec{\Theta} - \vec{y})$$

$$\Rightarrow \frac{\partial J(\vec{\Theta})}{\partial \vec{\Theta}} = 2X^TX\vec{\Theta} - 2X^T\vec{y} = 0$$

$$\Rightarrow \vec{\Theta} = (X^TX)^{-1}X^T\vec{y}$$

- **What's the maximum likelihood estimate?** (MLE)

"Find the parameters that max. the probability that the model's predictions match the actual observations"

$$\Theta_{ML} = \underset{\Theta}{\text{argmax}}\ p_{Model}(\vec{y}\,|\,X,\Theta)$$

$\vec{y}$: the actual labels; X: The samples

observations from $p_{data}(\vec{y}\,|\,X)$

(Or "Probabilities of seeing outcomes that I actually see")

Data is i.i.d. $\Rightarrow$ independence $\Rightarrow p(a,b) = p(a)p(b)$

$$= \underset{\Theta}{\text{argmax}}\ \prod_{i=1}^{n} p_{Model}(y_i\,|\,\vec{x}_i,\Theta)$$

"Θ maximizes [·] as well as $\log(\cdot)$, since $\log$ is a monotonic function"

$\log(ab) = \log a + \log b$

$$= \underset{\Theta}{\text{argmax}}\ \sum_{i=1}^{n} \log p_{model}(y_i\,|\,\vec{x}_i,\Theta)$$

* "Gaussian Assumption"

Assume $y_i \sim \mathcal{N}(\vec{x}_i\vec{\Theta}, \sigma^2)$ — $\vec{x}_i\vec{\Theta}$: Mean

$$\Rightarrow p(y_i) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}(y_i-\mu)^2}$$

*

$$= \underset{\Theta}{\text{argmax}}\left(\sum_{i=1}^{n} -\frac{1}{2}\log(2\pi\sigma^2) + \sum_{i=1}^{n} -\frac{1}{2\sigma^2}(y_i - \vec{x}_i\vec{\Theta})\right)$$

$$= \underset{\Theta}{\text{argmax}}\left(-\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(\vec{y} - X\vec{\Theta})^T(\vec{y} - X\vec{\Theta})\right)$$

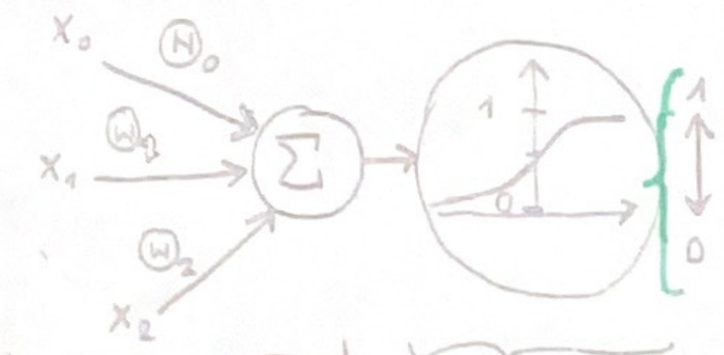
$\frac{\partial J(\Theta)}{\partial \Theta}$

$$\Rightarrow \Theta = (X^TX)^{-1}X^T\vec{y}$$

$\Rightarrow$ MLE corresponds to the Least Squares Estimate (same solution)

- What is Logistic Regression? → for binary predictions

→ Taking the output from a lin. regression and feeding it into a sigmoid function



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

(sigmoid's)

=> The label could be 0 or 1.

=> The model's prediction:

$$\hat{y}_i = \sigma(x_i \vec{w})$$

model for the

→ Is a Probability of a binary output:

Model of Coins:

$$p(z|\phi) = \phi^z (1-\phi)^{1-z} =$$

$$= \begin{cases} \phi, & z=1 \\ 1-\phi, & z=0 \end{cases}$$

The prediction of the sigmoid

$$\hat{y} = p(y=1 \mid X, \vec{w}) = \prod_{i=1}^{n} p(y_i=1 \mid x_i, \vec{w})$$

Bernoulli Trial

label (discrete)

$$= \prod_{i=1}^{n} \hat{y}_i^{\,y_i} (1-\hat{y}_i)^{(1-y_i)}$$

prediction (continuous)

$$= p(\vec{y} \mid X, \vec{w})$$

MLE: Find the $\vec{w}$ that max. the likelihood of the observations given $\vec{w}$

$$\Rightarrow \vec{w}_{ML} = \underset{\vec{w}}{\operatorname{argmax}} \log p(\vec{y} \mid X, \vec{w})$$

$$= \underset{\vec{w}}{\operatorname{argmax}} \sum_{i=1}^{n} \log\left(\hat{y}_i^{\,y_i} (1-\hat{y}_i)^{(1-y_i)}\right)$$

$$= \underset{\vec{w}}{\operatorname{argmax}} \sum_{i=1}^{n} \left(y_i \log \hat{y}_i + (1-y_i) \log(1-\hat{y}_i)\right)$$

=> Loss $\mathcal{L}(\hat{y}_i, y_i) = -\left[y_i \log \hat{y}_i + (1-y_i) \log(1-\hat{y}_i)\right]$!

binary-cross-entropy (BCE)

e.g.:

$y_i = 1 \rightarrow \mathcal{L} = \log(\hat{y}_i) \rightarrow \hat{y}_i$ should be large

$y_i = 0 \rightarrow \mathcal{L} = \log(1-\hat{y}_i) \rightarrow \hat{y}_i$ should be small

→ Optimization:

Minus in $\mathcal{L}$

Cost function: $C(\vec{w}) = +\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y_i)$

=> Minimize via Gradient descent, since there is no closed form solution.

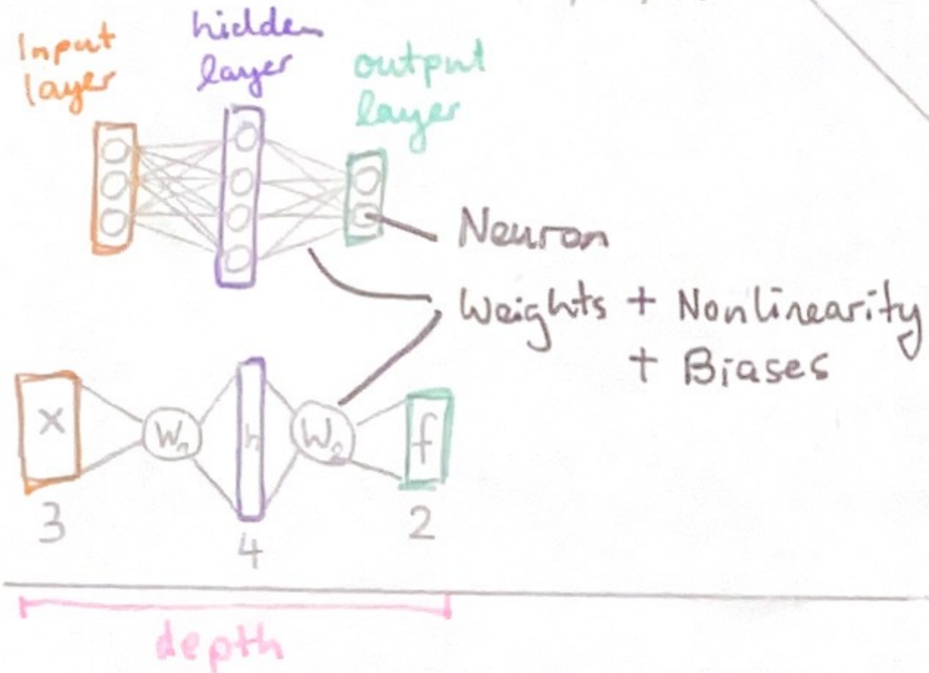
I2DL 03: Introduction to Neural Networks

↳ Can be written as non-linear score function

$$f = \ldots(\underbrace{\max(0, W_1 x)}_{\text{1. layer}}) \quad \text{weight of 1. layer}$$

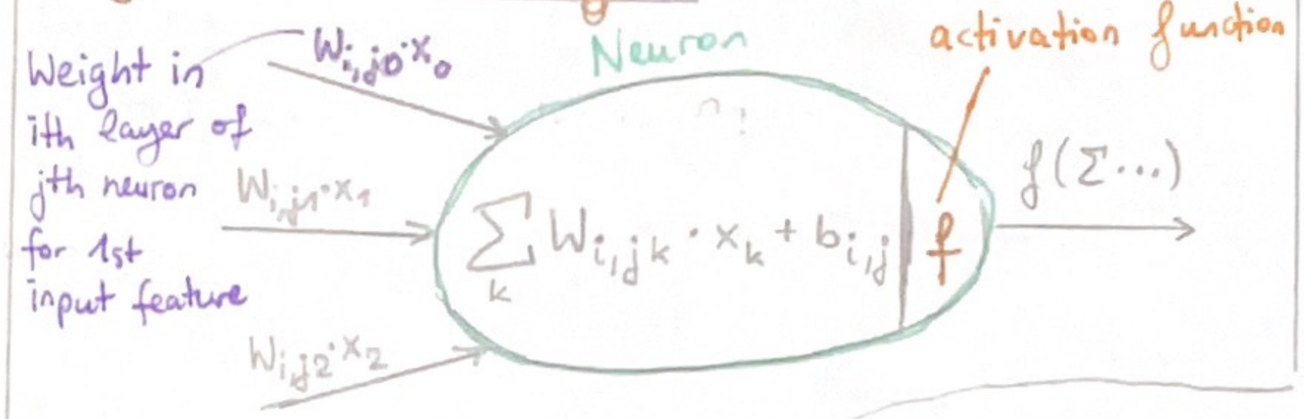
Example: 2-layer NN

$$f = W_2 \max(0, W_1 x)$$



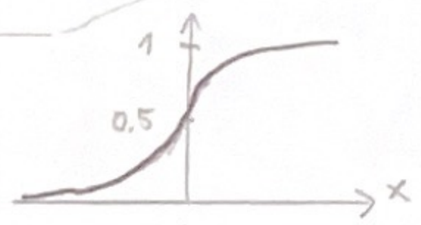
→ ANNs are inspired by the brain

Single Neuron in a layer:

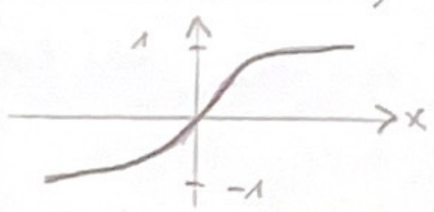


- Activation functions: ("Non-linearities")

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

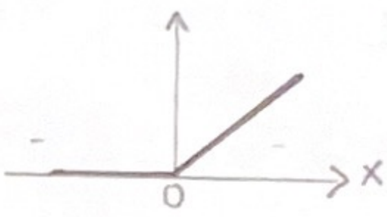


* tanh: $\tanh(x)$

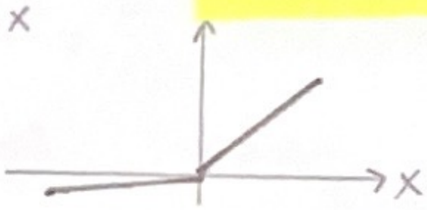


Should be steeper than Sigmoid

* ReLU: $\max(0, x)$



* Leaky ReLU: $\max(0.1x, x)$



* Parametric ReLU: $\max(\alpha x, x)$ ← learnable parameter

* Maxout: $\max(w_1^T x + b_1, w_2^T x + b_2)$

* ELU: $f(x) = \begin{cases} x & , x > 0 \\ \alpha(e^x - 1) & , x \leq 0 \end{cases}$

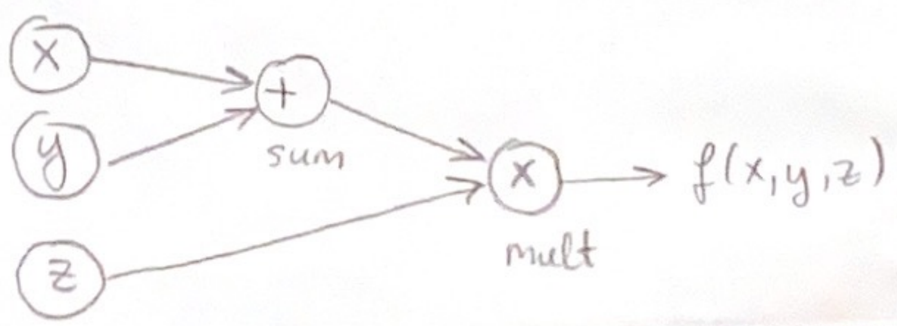
- Why is a NN structured into layers?

1) Usually all neurons per layer are the same w/ veg simple math
2) Structure allows efficient computation
3) Inspired by neurons in the brain

- What's a compute graph / computational graph?

→ Directional graph
→ Matrix operations represented as compute nodes
→ Variables or operators $(+, -, *, /, \log(), \exp())$ as vertex nodes

e.g.: $f(x, y, z) = (x + y)z$



- Loss functions: A function to measure the network performance

  * Small loss = good predictions

  * depends heavily on the problem

Ground truth / prediction

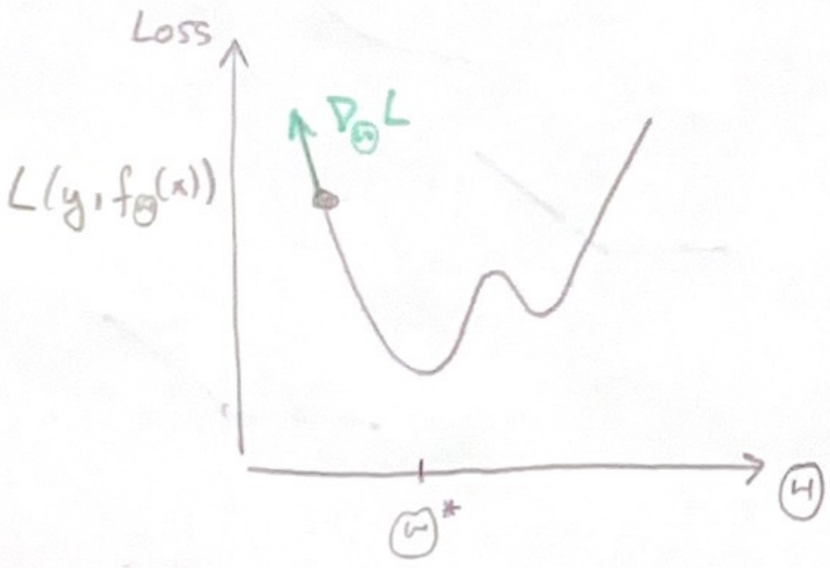
L1-Loss: $$L(y, \hat{y}; \Theta) = \frac{1}{n} \sum_{i}^{n} \| y_i - \hat{y}_i \|_1$$

MSE-Loss: $$L(y, \hat{y}; \Theta) = \frac{1}{n} \sum_{i}^{n} \| y_i - \hat{y}_i \|_2^2$$

Binary-Cross-Entropy: $$L(y, \hat{y}; \Theta) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i) \right]$$

Cross-Entropy: $$L(y, \hat{y}; \Theta) = -\sum_{i=1}^{n} \sum_{k=1}^{k} (y_{ik} \log \hat{y}_{ik})$$

- How is the NN improved?



Goal: Find $\Theta^* = \text{argmin } L(y, f_\Theta(x))$

$\Rightarrow$ Iterate: $\Theta = \Theta - \alpha \nabla_\Theta L(y, f_\Theta(x))$

step size / learning rate

Since the gradient points in the direction of the greatest ascent of L

- Why is gradient descent used?

→ NNs can be formulated as compute graphs and GD is easy to compute on a compute graph.

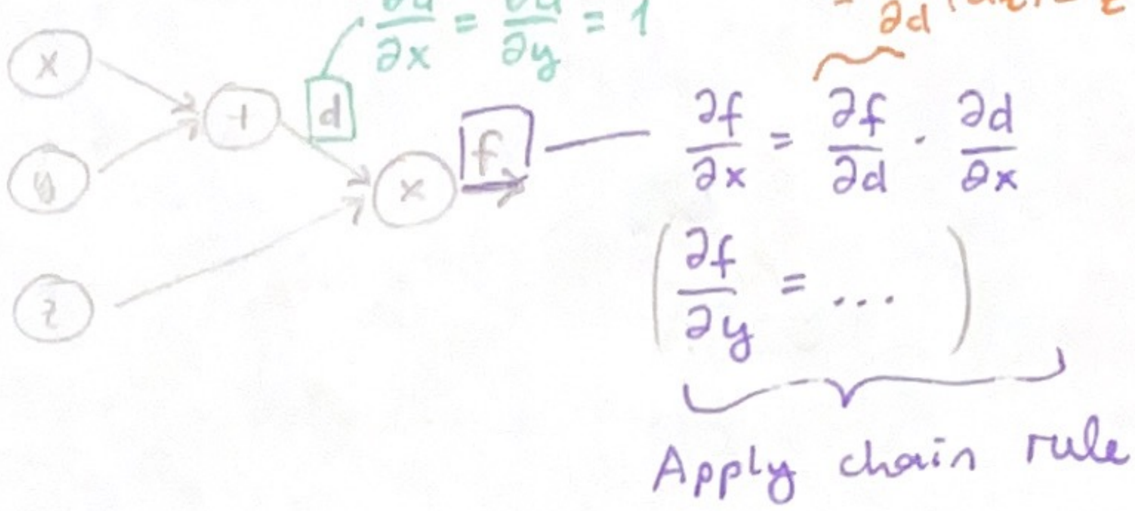
12DL 04: Optimization & Backpropagation

4

- What's the idea of Backprop?

→ ~~Compute~~ Break down the gradient computation $\nabla_{\vec{\theta}} L(\vec{\theta})$

e.g. for $f(x,y,z) = (x+y)z$



* x_k : Input variables

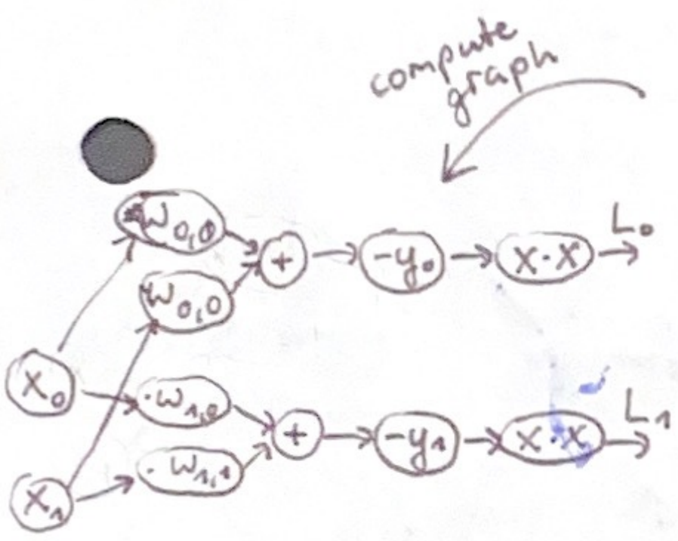
* $w_{l,m,n}$: Weights
- l: layer
- m: neuron in layer
- n: weight in neuron

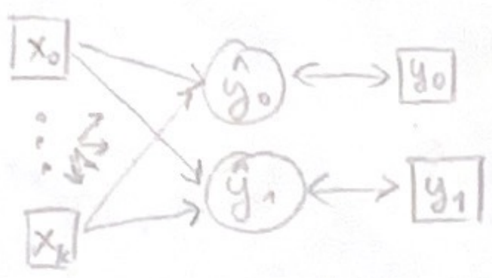
* $\hat{y}_i$: prediction

* y_i : ground truth targets

* L : loss

NN example:





$\hat{y}_i = A\left(b_i + \sum_k x_k w_{ik}\right)$

Activation func · Bias · weight

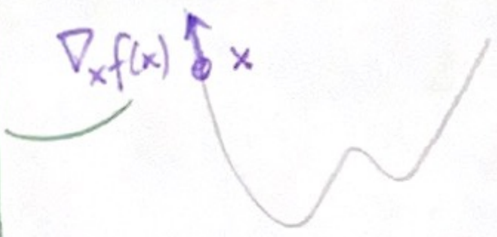
Goal: Compute $\frac{\partial L}{\partial w_k}$ (Gradient w.r.t. all weights and biases)

with $L = \sum_i L_i = \sum_i (\hat{y}_i - y_i)^2$

→ Use chain rule: $\frac{\partial L_i}{\partial w_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{i,k}}$

- What's Gradient descent?

Direction of greatest increase of the func



=> Update rule: $x' = x - \alpha \nabla_x f(x)$

learning rate

For a NN: $\vec{w}' = \vec{w} - \alpha \nabla_{\vec{w}} f_{\{\vec{x},\vec{y}\}}(\vec{w})$

with $\nabla_{\vec{w}} f_{\{\vec{x},\vec{y}\}}(\vec{w}) = \begin{pmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \vdots \\ \frac{\partial f}{\partial w_{l,m,n}} \end{pmatrix}$

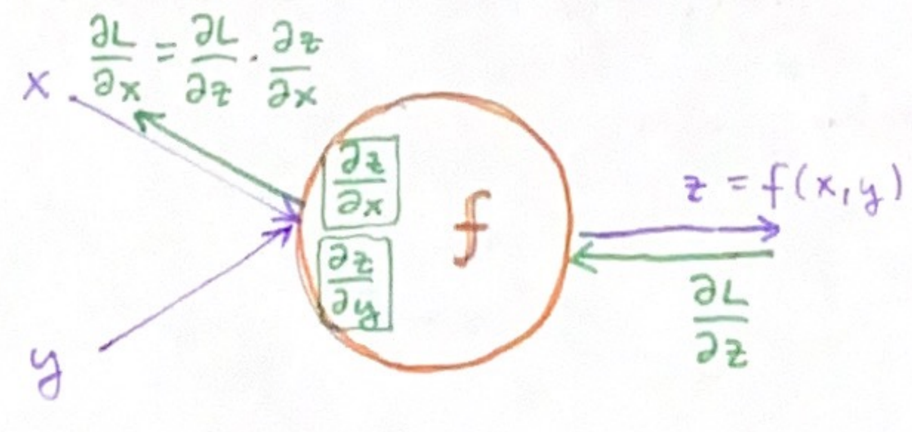
for a given training pair

- Which functions does each neuron implement?

→ Forward pass:

→ Backward pass:

Flow of the forward pass & gradients



Unknown parameters

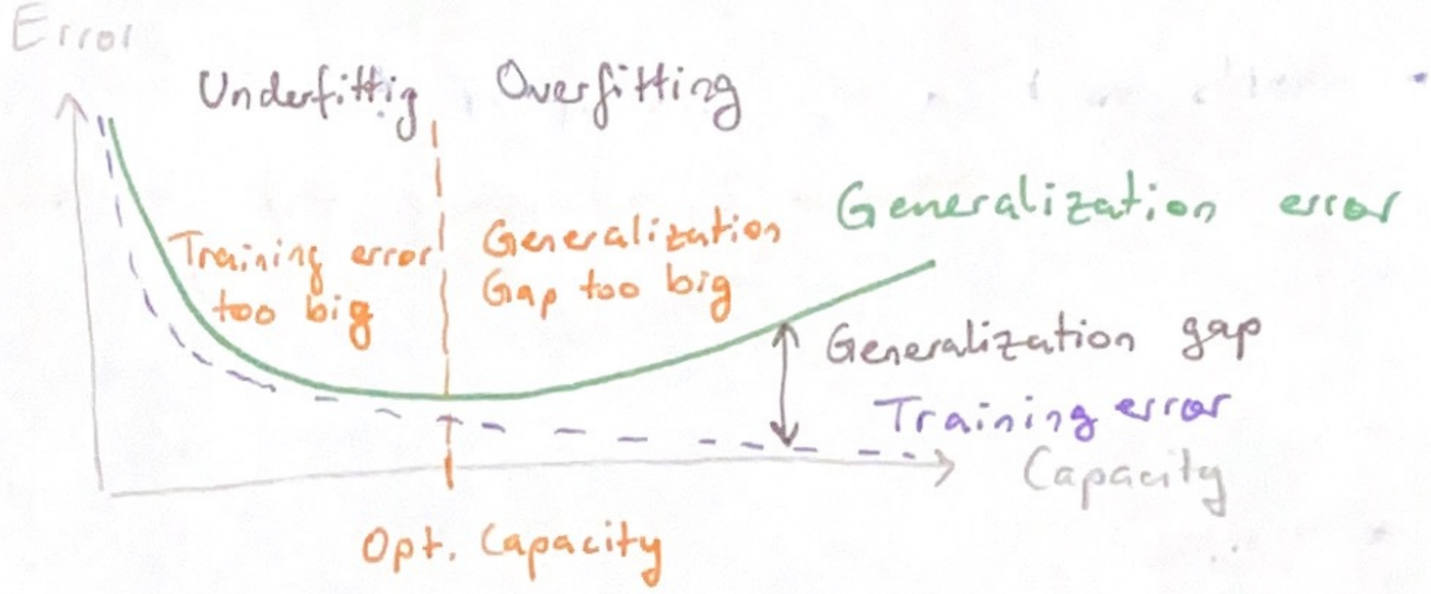
#neurons · #input channels + #biases

- What's over- & underfitting?

Underfit
→ too little capacity

Appropriate

Overfit
→ Model has too much capacity



- How to prevent overfitting? → Regularization

=D Add regularization term to loss function

=D $L(\vec{y}, \hat{\vec{y}}, \vec{\theta}) = \sum_i (x_i \theta_{ji} - y_i)^2 + \lambda R(\vec{\theta})$

Regularization term; Hyper-parameter

a.k.a. weight decay

L2 Regularization: $R(\vec{\theta}) = \sum_i \theta_i^2$ ← favours smaller weights → enforces that weights have similar values

L1 Regularization: $R(\vec{\theta}) = \sum_i |\theta_i|$ → considers all features

(only in regularization i.e. if the L-norm is taken of the weights ||w||)

↑ enforces sparsity of weights (a few large ones)
→ Considers key features

→ Large regularization smoothes the decision / prediction

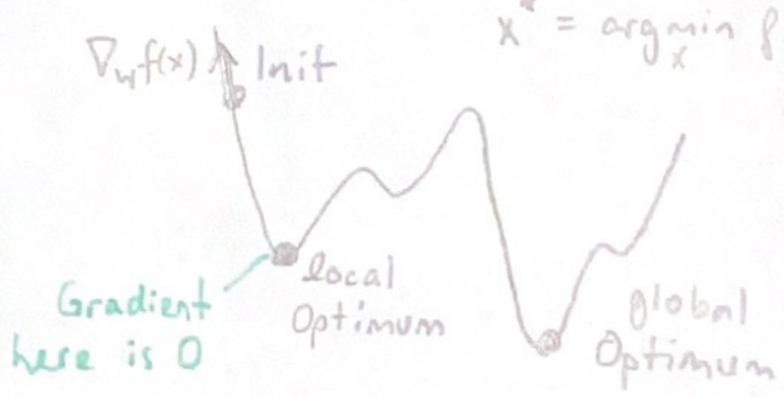
Problem: GD possibly only optimizes for the regulariz. term

Aim of regularization: Training error ↑
Validation error ↓

Regularization = Any technique that lowers the validation error (variance) at the cost of increasing the training error (bias)

12DL 05: Scaling Optimization

- How do we know if the local minimum found with G.D. is a global minimum?



→ Use Convexity

↳ If f is convex, then all local minima are global minima

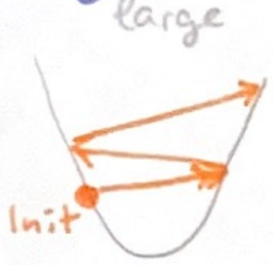
→ In general the Loss function of a NN is not convex

→ No (practical) method to know this

- Small vs large learning rate ?

in 1-D

large



⊖ Can overshoot

⊕ good progress

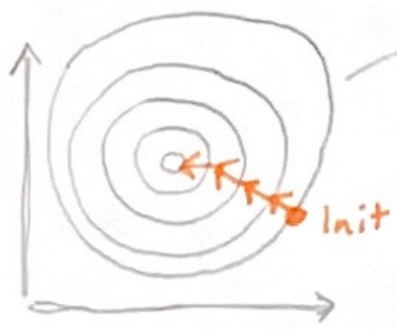
⊖ slow

⊕ does not overshoot

same

* Also the learning rate may cause slow convergence during plateaus but still overshoot during steep drops of the Loss curve

In 2-D:



- What is the equation for G.D. for a single training sample and a batch ?

Single Sample:

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha \nabla_{\vec{\theta}} L_i(\vec{\theta}^{k}, x_i, y_i)$$

- $\vec{\theta}^{k+1}$, $\vec{\theta}^{k}$: Weights & Biases at training step k and $k+1$
- α: Learning rate
- $\nabla_{\vec{\theta}} L_i$: Gradient w.r.t. θ (Computed via Backprop)
- L_i: Loss function
- x_i, y_i: Training sample

Batch:

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha \underbrace{\nabla_{\vec{\theta}} L(\vec{\theta}^{k}, x_{\{1..n\}}, y_{\{1..n\}})}_{\text{avg. Gradient}} = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\theta} L_i(\theta^{k}, x_i, y_i)$$

Abbreviation "Slang" $= \nabla L = \sum_{i=1}^{n} \nabla_{\theta} L_i$

↑ omitting $\frac{1}{n}$ rescales the learning rate

- Can one compute the optimal learning rate ?

→ Optimize for α:

1) $\alpha^* = \arg\min_{\alpha} L(\underbrace{\vec{\theta}^{k} - \alpha \nabla_{\vec{\theta}} L}_{\vec{\theta}^{k+1}})$

2) $\vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha^* \nabla_{\vec{\theta}} L$

} In practice, this is too computationally expensive

- How to tackle the computational complexity of naive G.D.?

→ has $\mathcal{O}(n)$ per training step

→ Use Stochastic Gradient Descent (SGD) Gradient for n samples

+ can help escape local minima and saddle points

$$\frac{1}{n}\sum_{i=1}^{n} L_i(\vec{\theta}, x_i, y_i) = E_{i\sim[1..n]}\left[L_i(\vec{\theta}, x_i, y_i)\right] \approx \frac{1}{|S|}\sum_{j\in S} L_j(\theta, x_j, y_j)$$

express total loss over training data as expectation

approximate on a subset of the training data

with $S \subseteq \{1..n\}$

→ Minibatches w/ length m and $m \ll n$

→ What's a good minibatch size?

Typically power of 2: 8 16 32 64 128

small ——— large

⊖ noisy updates
⊕ fast

⊕ less noisy
⊖ slow

→ limited by GPU memory

- Iteration: Single gradient update step with one minibatch
  Epoch: Complete pass through training set

⊖ When does SGD converge?

Robbins & Monro Condition

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha_k H(\vec{\theta}^{k}, X)$$

estimates $\nabla F(\theta^k)$

converges to a local minimum if:

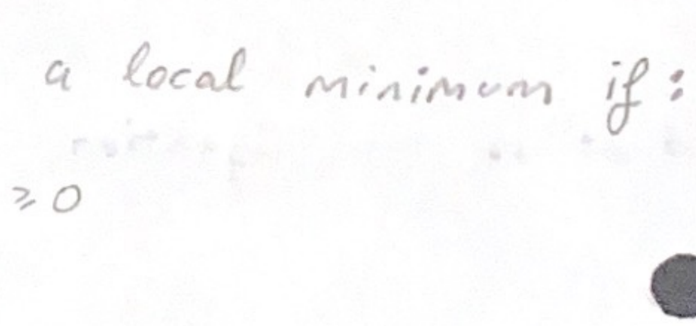
1) $\alpha_n \geq 0 \quad \forall n \geq 0$
2) $\sum_{n=1}^{\infty} \alpha_n = \infty$
3) $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$

If also 4) $F(\vec{\theta})$ is convex → global minimum

- What are problems of SGD?

→ Gradient is scaled equally across dimensions

→ Difficult to find a good learning rate



→ Solution to equally scaled gradients?

→ Gradient Descent with Momentum

① Gradient Descent w/ Momentum:

$$\vec{v}^{k+1} = \beta \vec{v}^{k} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}^{k})$$

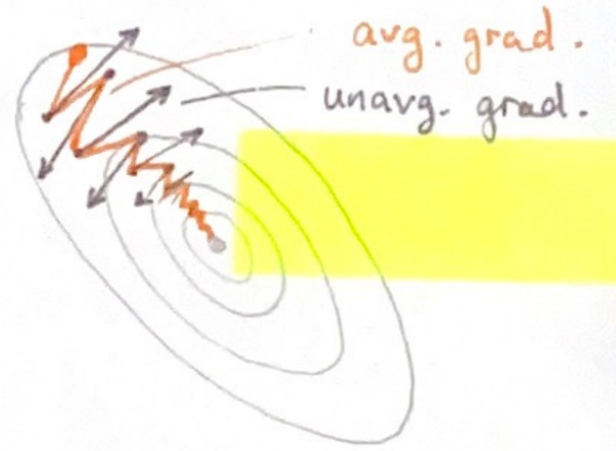
($\vec{v}$: velocity; β: accumulation rate (e.g. = 0.9))

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} + \vec{v}^{k+1}$$

* If gradient steps in a dim. always go in the same direction → larger step
* If steps go back & forth → smaller steps

⇒ Exponentially weighted avg. of the gradient

because previous gradients "decay" exponentially with β (i.e. naive G.D. for $\beta = 0$)



→ it can overcome local minima + accelerate optimization

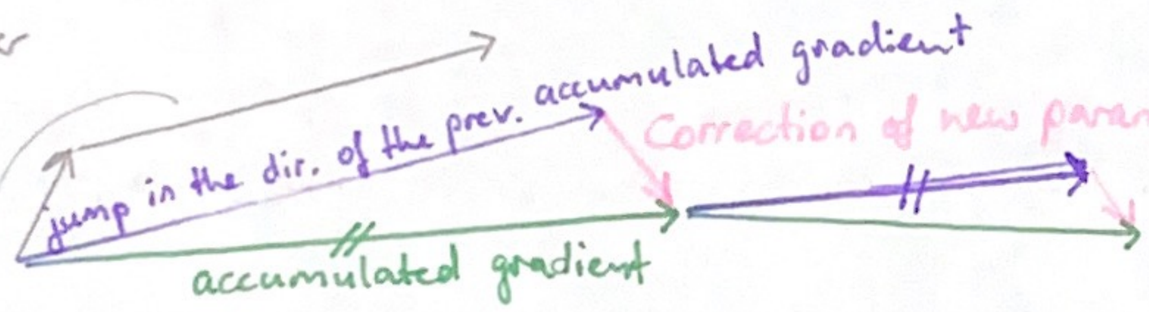
- What's the Nesterov Momentum? ("Look-ahead momentum")

$$\vec{v}^{k+1} = \beta \vec{v}^{k} - \alpha \nabla_{\vec{\theta}} L(\tilde{\vec{\theta}}^{k+1}) \quad \text{with} \quad \tilde{\vec{\theta}}^{k+1} = \vec{\theta}^{k} + \beta \vec{v}^{k}$$

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} + \vec{v}^{k+1}$$

↳ Motivation: Computing the gradient at the next look-ahead-step gives us a better gradient

In parameter space:



standard momentum

- What's RMS Prop? (Root Mean Squared Propagation)

→ If gradients in a dim. have a high variance, (oscillations in this direction) the $\vec{v}$ are dampened

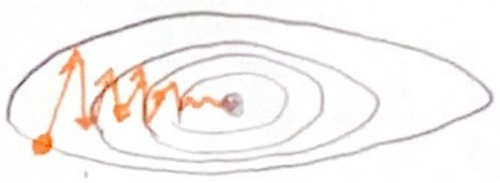
$$\vec{s}^{k+1} = \beta \cdot \vec{s}^{k} + (1-\beta)\left[\nabla_{\vec{\theta}} L \circ \nabla_{\vec{\theta}} L\right]$$

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha \cdot \frac{\nabla_{\vec{\theta}} L}{\sqrt{\vec{s}^{k+1}} + \varepsilon}$$

($\circ$: Element-wise operations; ε: e.g. 10^{-8})

divide learning rate by exp. decaying avg. of squared gradients

Example:



Divide by squared gradients:

- In Y-direction: divide by large number
- In X-direction: divide by small number

⇒ allows for larger learning rates

- What's Adaptive Momentum estimation? (ADAM)

→ Combines Momentum and RMSProp

1. Momentum: $\vec{m}^{k+1} = \beta_1 \cdot \vec{m}^{k} + (1-\beta_1)\,\nabla_{\vec{\theta}} L(\vec{\theta}^{k})$ ($\beta_1 = 0.9$)

2. Momentum: $\vec{v}^{k+1} = \beta_2 \cdot \vec{v}^{k} + (1-\beta_2)\,[\nabla_{\vec{\theta}} L(\vec{\theta}^{k}) \circ \nabla_{\vec{\theta}} L(\vec{\theta}^{k})]$ ($\beta_2 = 0.999$)

$$\vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha \cdot \frac{\vec{m}^{k+1}}{\sqrt{\vec{v}^{k+1}} + \varepsilon}$$

α: Needs tuning

$\vec{m}^{k+1}$: mean of gradients (exp. decaying)

$\sqrt{\vec{v}^{k+1}}$: Variance of gradients (exp. decaying)

$\varepsilon = 10^{-8}$

At the start: $\vec{v}^{0} = \vec{m}^{0} = \vec{0}$

For stable training, we need "bias correction" (explained on StackOverflow)

Actual update rule in ADAM:

$$\Rightarrow \hat{\vec{m}}^{k+1} = \frac{\vec{m}^{k+1}}{1-\beta_1^{k+1}} \quad \& \quad \hat{\vec{v}}^{k+1} = \frac{\vec{v}^{k+1}}{1-\beta_2^{k+1}}$$

$\vec{v}^{k+1}$ — in the $(k+1)$th step; β_2^{k+1} — to the power of $k+1$

$$\Rightarrow \vec{\theta}^{k+1} = \vec{\theta}^{k} - \alpha \frac{\hat{\vec{m}}^{k+1}}{\sqrt{\hat{\vec{v}}^{k+1}} + \varepsilon}$$

- What's Newton's Method?

→ Approx. loss func. by second-order derivative (taylor expansion)

$$L(\vec{\theta}) \approx L(\vec{\theta}_0) + (\vec{\theta}-\vec{\theta}_0)^T \nabla_{\vec{\theta}} L(\vec{\theta}_0) + \frac{1}{2}(\vec{\theta}-\vec{\theta}_0)^T H (\vec{\theta}-\vec{\theta}_0)$$

Diff. and set equal to 0 ⇒

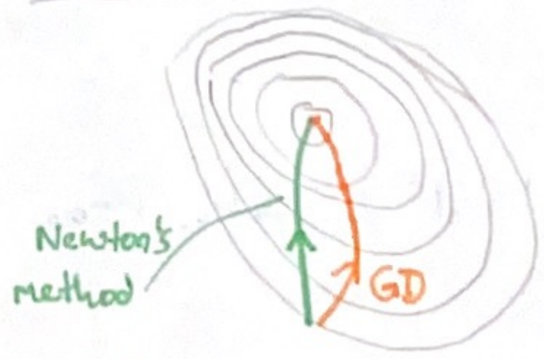
$$\theta^{*} = \theta_0 - H^{-1} \nabla_{\vec{\theta}} L(\vec{\theta})$$

Hessian replaces the learning rate!

→ Scales each gradient dim. by the curvature

Faster convergence in terms of number of iterations.

Example:



→ Not used in DL, because of high complexity (inverting is $\mathcal{O}(k^3)$)

⇒ Quasi-Newton methods approximate H^{-1}

↳ BFGS & L-BFGS ($\mathcal{O}(n^2)$ & $\mathcal{O}(n)$)

→ Gauss-Newton

→ Levenberg

→ Levenberg-Marquardt

- Which optimizer to use:

Standard: Adam

Fallback: SGD with momentum

→ Newton, L-BFGS, ... don't work on minibatches

- Which (general) optimization algorithms exist?

  - for lin. systems ($Ax = b$)
    - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG
  - Non-lin. (gradient-based)
    - Newton, Gauss-Newton, (L)BFGS ← second order derivative
    - GD, SGD ← first order derivative
  - Others
    - Genetic algorithms, ...

Universal Approximation Theorem:

Any function on a compact set can be approximated within an arbitrary error $\varepsilon > 0$, using a single-layer neural network (1 hidden layer + sigmoid)

↳ (but the layer may be extremely large)

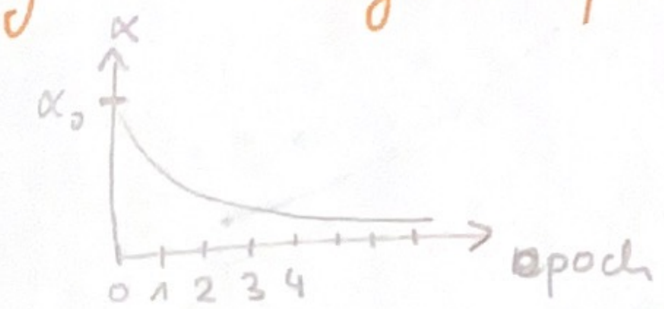
↳ inefficient use of comp. power

12DL 06 : Training NNs

- Learning rate decay: → decrease stepsize during training to converge to optimum

different Options:

- $\alpha = \frac{1}{1 + t \cdot epoch} \cdot \alpha_0$ — t: decay rate, $epoch$: number of epoch



- Step decay: $\alpha = \alpha - t \cdot \alpha$ (apply only every n steps)
- exp. decay: $\alpha = t^{epoch} \cdot \alpha_0$ ($t<1$)
- $\alpha = \frac{t}{\sqrt{epoch}} \cdot \alpha_0$

- Training Schedule (specify α for entire training process)
  → every n-epochs

- Basic training recipe:

Given: - Dataset $\{x_i, y_i\}$ with usually $dim(x) >> dim(y)$
- Network f with parameters $\vec{w}, \vec{b}$

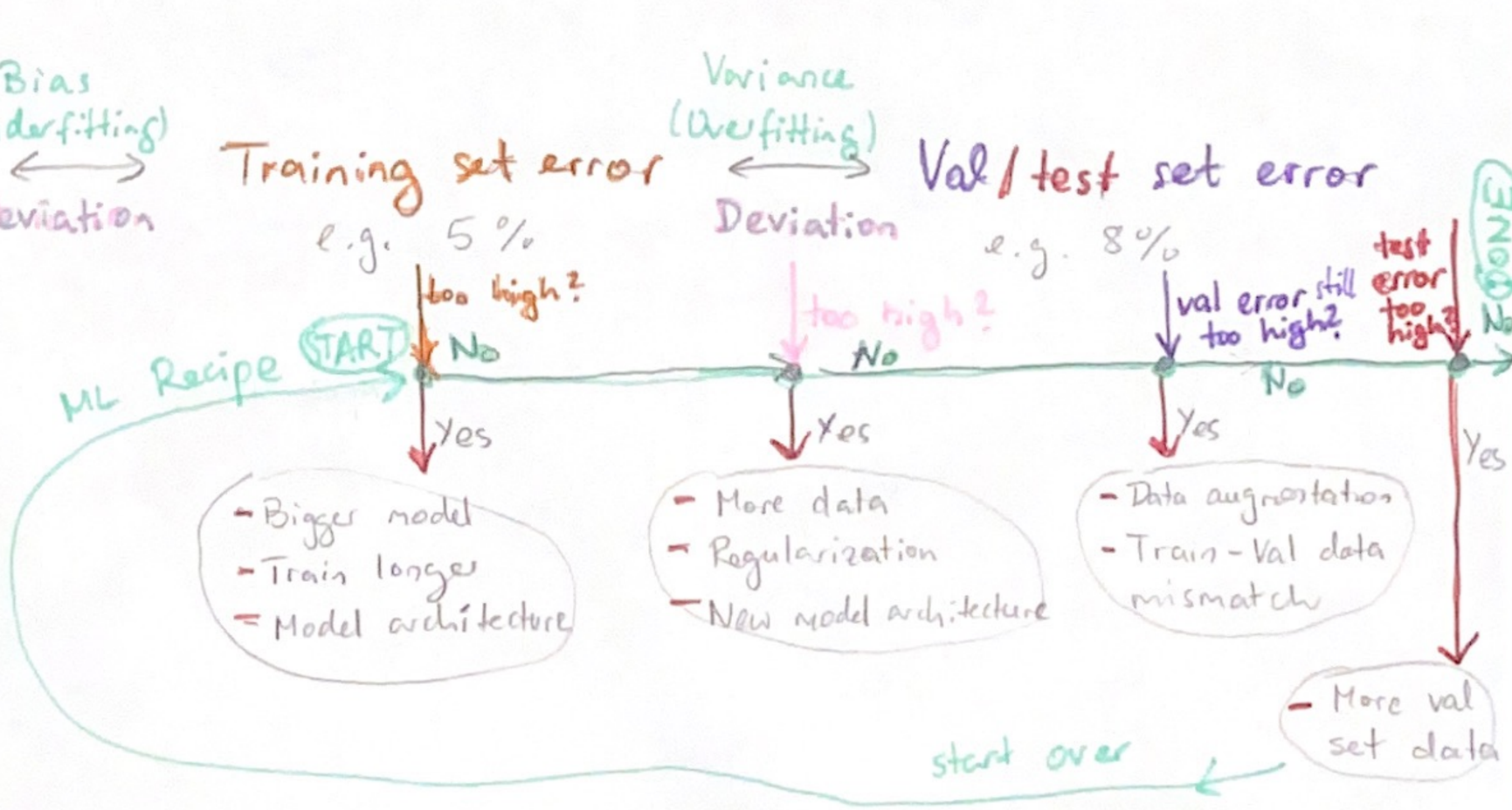
Method: Use SGD to optimize parameters

Learning means generalization to unknown dataset.

Procedure:
1. Train on training set → Training error is avg minibatch error
2. Optimize hyperparameters & check generalization on validation set → Evaluate loss on subset of validation set every n iterations
3. After developing, test performance on test set

- Error:

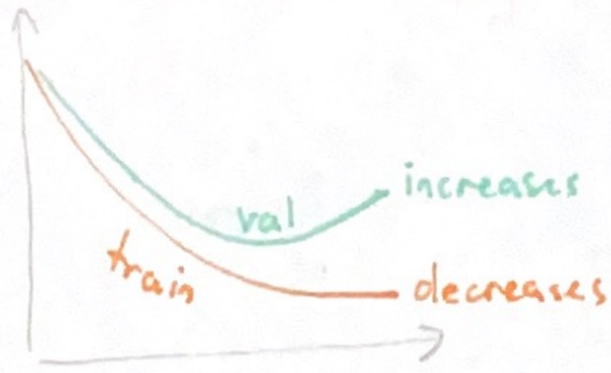
mislabelled data — Ground truth error e.g. 1% — Bias (underfitting) ⟷ Deviation — Training set error e.g. 5% — Deviation ⟷ Variance (overfitting) — Val/test set error e.g. 8%



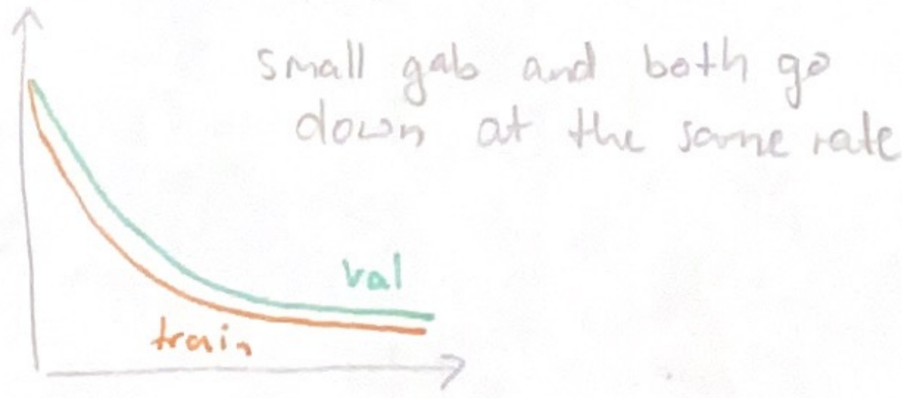
- Training curves :

Underfitting

Overfitting



Ideal



- Hyperparameter tuning methods :

→ Manual search

→ Grid Search (Select points from parameter spaces within a range at fixed distances between them)

→ Random search (Select random points)

- Various Tips for DL :

1. Start with a single sample to check if the model works and overfits (train accuracy should be 100%)

→ Then increase to a few samples → Then move from overfitting to more samples (see generalization)

2. Find learning rate so that loss drops exponentially within 100 iterations

3. Use coarse grid search and then refine grid around best hyperparameters (if the compute budget allows)

4. Get precise timings (find performance bottleneck)

→ An iteration should not exceed 500 ms

→ How long until convergence

→ Dataloading

→ Backprop

5. Start with the simplest network possible

→ Rule of thumb : # of layers divided by 5

6. Make on change at a time.

12DL 07: Training NNs 2

- Loss functions:

  • L^1-Loss: $\sum_{i=1}^{n} |y_i - f(\vec{x}_i)|$ — + Robust to outliers; − costly to optimize; → optimum is median

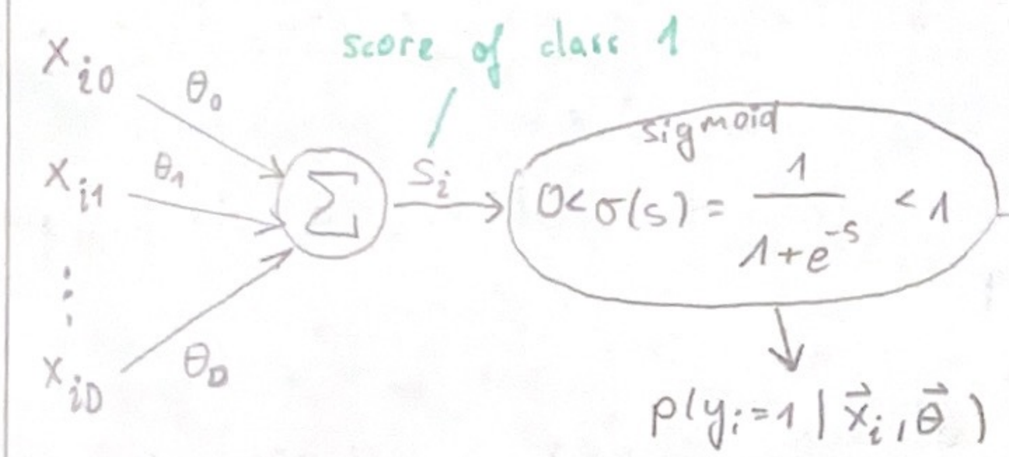
  • L^2-Loss: $\sum_{i=1}^{n} (y_i - f(\vec{x}_i))^2$ — + prone to outliers; − compute-efficient optimization; → optimum is mean

  • Cross-Entropy Loss: $-\frac{1}{N}\sum_{i=1}^{n}\sum_{k=1}^{k}\left[y_{ik} \cdot \log\left(\frac{e^{s_{ik}}}{\sum_{k'=1}^{k'} e^{s_{ik'}}}\right)\right]$ } general equation

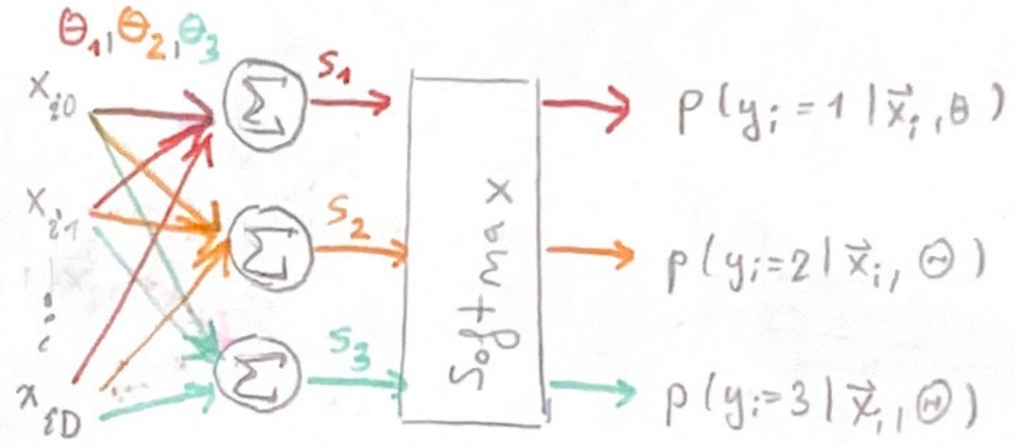
  used in the lecture { or $L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$ for every class y_i and use only the L_i from the correctly classified class.

Network Outputs:

• Binary classification ($k=2$)



• Multiclass Classification ($k>2$)



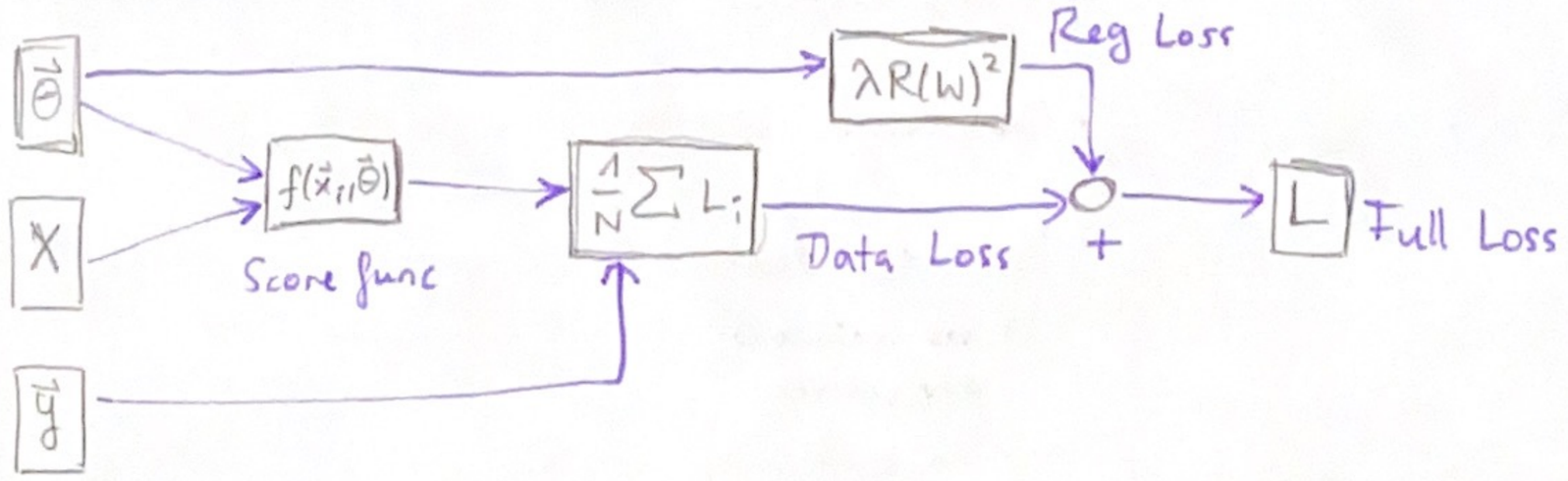
• Hinge Loss: $L_i = \sum_{k \neq y_i} \max(0, s_k - s_{k_i} + 1)$

s_{k_i}: score of the "label class"

→ Once the NN is very sure about its decision, the Loss becomes 0 and learning stops
⇒ „The Hinge loss saturates"
↳ stops optimizing (as opposed to CE)

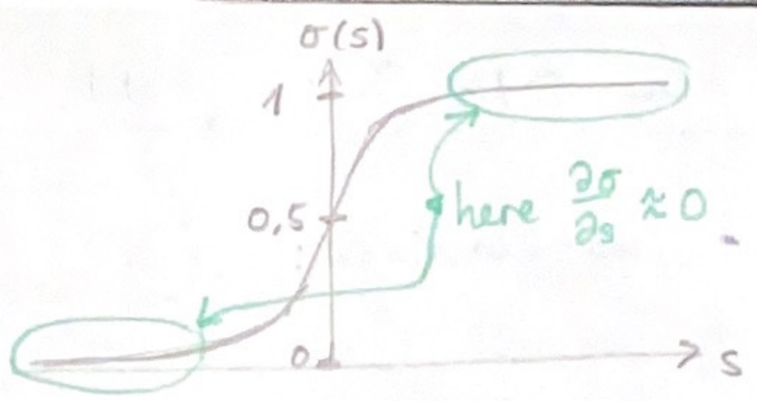
• Regularization-Loss: $\frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R^2(W)$

Data Loss + Reg Loss



- Activation functions: • Sigmoid: $\sigma(s) = \frac{1}{1+e^{-s}}$

not really used

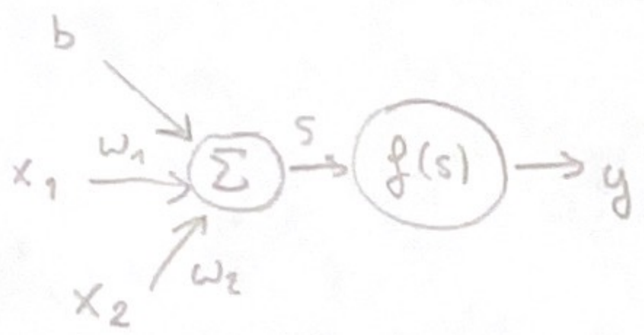


1st problem:
In the backward pass: $\frac{\partial L}{\partial s} = \frac{\partial \sigma}{\partial s} \cdot \frac{\partial L}{\partial \sigma} \approx 0$

⇒ Saturated neurons kill the gradient flow

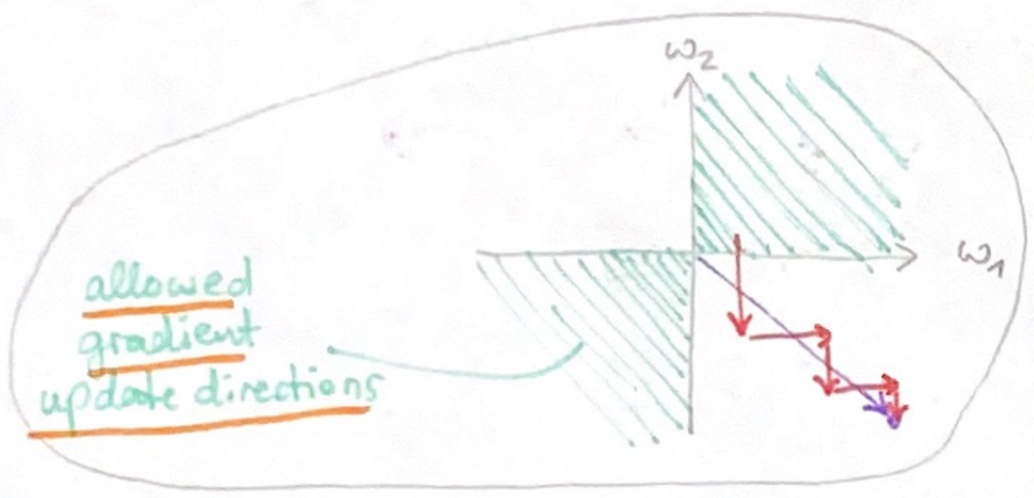
⇒ Sigmoids have a small active region / not ideal

2nd problem:



and assume all positive data

$x_1, x_2 > 0$



* $\sigma(s) > 0 \quad \forall s$ (sigmoid is always positive)

⇒ $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial w_1}$

$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial s} \cdot \frac{\partial s}{\partial w_2}$

either >0 or <0

>0 for all positive data

or sigmoid input from previous layer (>0)

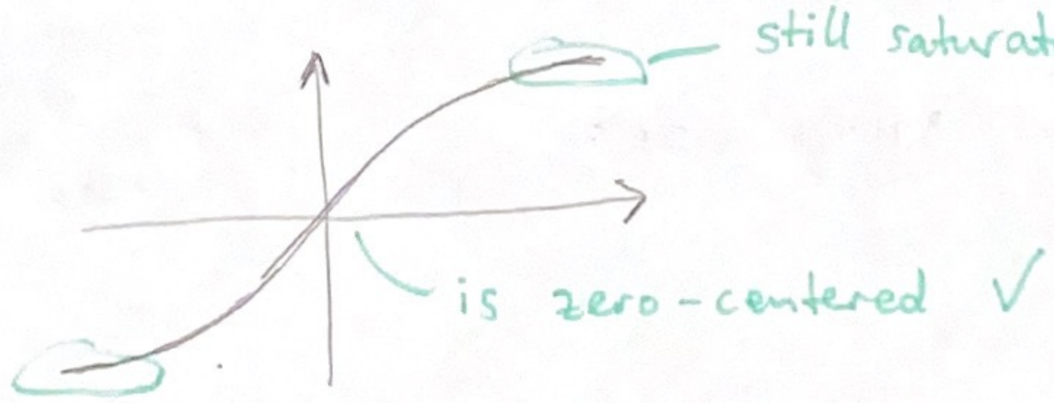
All ⇒ Gradients are simultaneously positive or neg.

* The hypothetical optimal parameters are only reached via a zig zag path

also ⇒ For this reason we use zero-centered data! (and a zero-centered activation)

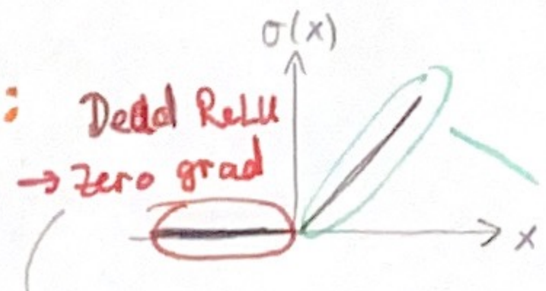
• Tanh:

used in RNNs



• ReLU:

standard choice



$\sigma(x) = \max(0, x)$

large & consitent gradients

→ Initialising ReLUs with sligthly pos. bias makes it likely for them to stay active

second choice to ReLU

Rectified lin. unit

• Leaky ReLU: $\sigma(x) = \max(0.01x, x)$

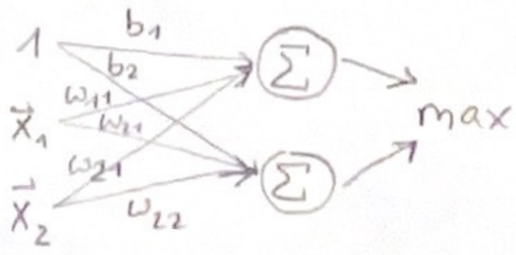
• Parametric ReLU: $\sigma(x) = \max(\alpha x, x)$

} To avoid dead ReLUs

• Maxout Units: $\text{Maxout} = \max(\vec{w}_1^T \vec{x} + b_1, \vec{w}_2^T \vec{x} + b_2)$

↳ has trainable parameters

→ can approximate convex functions (e.g. Absolut val)



example with two functions

Generalization of ReLUs ✓

Lin. regimes (nice grads) ✓

Does not die ✓

Does not saturate ✓

Increases the # of params ✗

- Weight Initialization:

→ Why not all zero? → All hidden units compute the same function
→ same gradients for all neurons
→ "no symmetry-breaking" / weights updated equally

→ Why not with small random numbers?

↳ Cause a small output of each layer

when using tanh as an activation func

$$f_\ell\left(\sum_i w_i^\ell x_i^\ell + b^\ell\right) \approx 0 = x_i^{\ell+1}$$

↳ The input to the next layer is small, but

even if the grad. of the activation func. is > 0 → the grad. of the affine layer are the inputs!

$$\Rightarrow \frac{\partial L}{\partial w_i^{\ell+1}} = \frac{\partial L}{\partial f_{\ell+1}} \cdot \frac{\partial f_{\ell+1}}{\partial w_i^{\ell+1}} = \frac{\partial L}{\partial f_{\ell+1}} \cdot x_i^{\ell+1} \approx 0$$

⇒ vanishing gradients in the affine layers caused by small output

→ Why not with large random numbers?

↳ When using tanh, the output is saturated to -1 and +1.

⇒ Gradient of the activation func ≈ 0

⇒ Vanishing gradient, caused by saturated activation function.

* Xavier Initialization: (and w/ tanh)

→ Use Gaussian with zero mean and # of input neurons

$$Var(s) = Var\left(\sum_{i=1}^{n} w_i x_i\right) = \sum_{i=1}^{n} Var(w_i x_i)$$

$$= \sum_{i=1}^{n} E[x_i]^2 Var(x_i) + E[w_i]^2 Var(x_i) + Var(x_i) Var(w_i)$$

x_i: 0, zero-mean; w_i: 0, zero-mean

i.i.d.

$$= n\,(Var(w)\,Var(x))$$

⇒ We want the variance of the output to equal the var. of the input

$$Var(s) = Var(x) \Rightarrow \boxed{Var(w) = \frac{1}{n}} \text{ (w/ tanh)}$$

⇒ The layer output does not collapse to 0 or saturate.

⇒ The gradient flow is not killed

When using ReLU: since half of the inputs are set to zero

$$\boxed{Var(w) = \frac{2}{n}}$$

Xavier-Haff initialization

12DL 08: Training NNs

- Data Augmentation: "is regularisation"

→ A classifier should be invariant to many transformations (pose, appearance, illumination of obj)

⇒ Synthesize data simulating such plausible transformations.

Methods:
- * Random brightness and contrast changes
- * Random crops of larger image (→ fixed crops in testing)
- * Flipping

→ Use the same data augmentation when comparing networkes

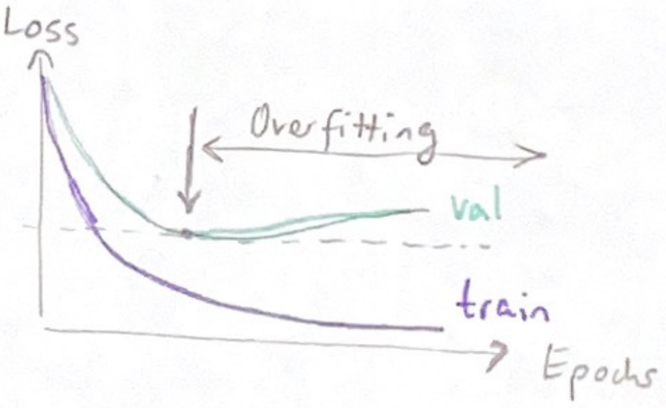
- Advanced Regularization:

* L_2-Regularization: $\omega_{k+1} = \omega_k - \varepsilon \left(\nabla_\omega (\omega_k, x, y) + \underbrace{\lambda \omega_k}_{\text{gradient of } L_2\text{-Reg.}} \right)$

↳ penalizes large weights

→ Improves generalization

* Early stopping



* Bagging & Ensemble methods

Use a different dataset for each model (is an ensemble method)

train multiple models and average their results

→ change optimizer / architecture / ...

→ If the errors are uncorrelated, the expected combined error will decrease lin. w/ the ensemble size.

model ensemble

* Dropout → Disable a set of random neurons (e.g. 50%)

↳ In the forward pass: fewer neurons impact the decision } half the capacity
(→ fewer information → harder)

→ In the backward pass: Gradients are only applied to active neurons

→ creates redundant representations (e.g. neurons for detecting tails are not active, so I create a new one)

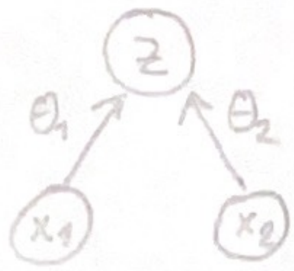
reduces co-adaptation between neurons / scores based on more features

→ different set of neurons w/ each batch (diff. data), but shared params.

* At test time the conditions in the NN during training time must be "simulated"

→ Weight scaling inference rule

During testing:



$$E[z] = \frac{1}{4}(\theta_1 \cdot 0 + \theta_2 \cdot 0) + \frac{1}{4}(\theta_1 x_1 + \theta_2 \cdot 0) + \frac{1}{4}(\theta_1 \cdot 0 + \theta_2 x_2) + \frac{1}{4}(\theta_1 x_1 + \theta_2 x_2) = \frac{1}{2}(\theta_1 x_1 + \theta_2 x_2)$$

Dropout probability $p = 0.5$

⇒ At test time the weights need to be adapted to match the conditions at training time

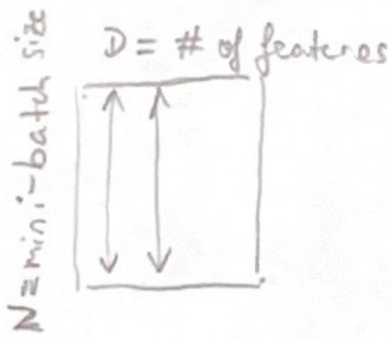
→ Dropout improves accuracy but reduces model capacity

* **Batch Normalization:**

→ Scale mini-batches to unit gaussian in each feature/dim.

$$E[\vec{x}] = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$Var[\vec{x}] = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$



$$\hat{\vec{x}}^{(k)} = \frac{\vec{x}^{(k)} - E[\vec{x}^{(k)}]}{\sqrt{Var[\vec{x}^{(k)}]}}$$

* BN layer is applied between affine layer and activation

↳ 1. Normalize

2. Allow netw. to change the range $\vec{y}^{(k)} = \gamma^{(k)} \hat{\vec{x}}^{(k)} + \beta^{(k)}$

→ possibility to undo normalization

$\gamma^{(k)}, \beta^{(k)}$: optimized in backprop

* Empirical results show, it's ok to treat dims separately

since we often don't test a mini-batch, but single samples

⇒ At test time use a mean and variance obtained from an exponentially weighted running average during training

Benefits:

→ More stable gradients w/ deeper nets

→ larger range of good hyperparams

Drawbacks:

→ cannot be used w/ small batches → Group norm instead of Batch norm

12DL 08: Convolutional neural networks

* FC layers with image data have too many weights ("brute force approach")
  → share the weights for different parts of the image ("use structured approach")

- What are convolutions? "Applying a filter to a function"

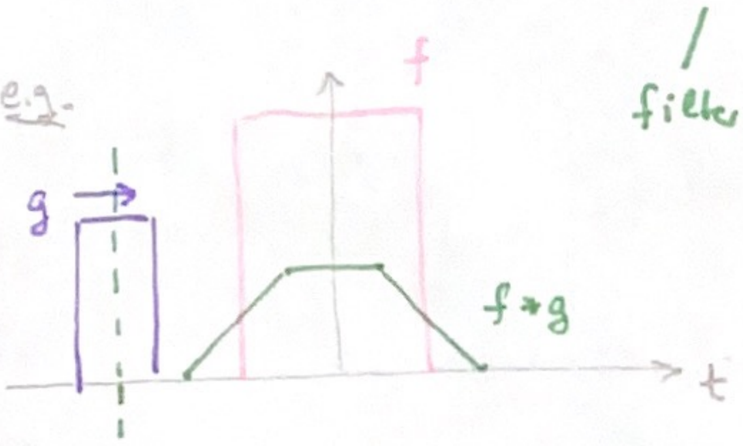
in conv layers

Continuous: $f * g = \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$:= "the area underneath the product" ②

- g: filter/kernel
- flip kernel
- slide kernel/shift kernel w/ t

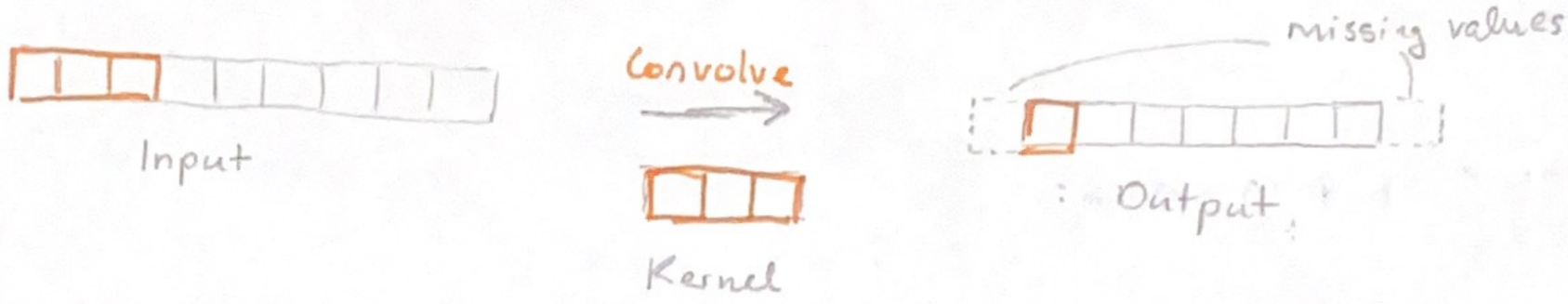
① multiply flipped and shifted kernel w/ func

e.g.

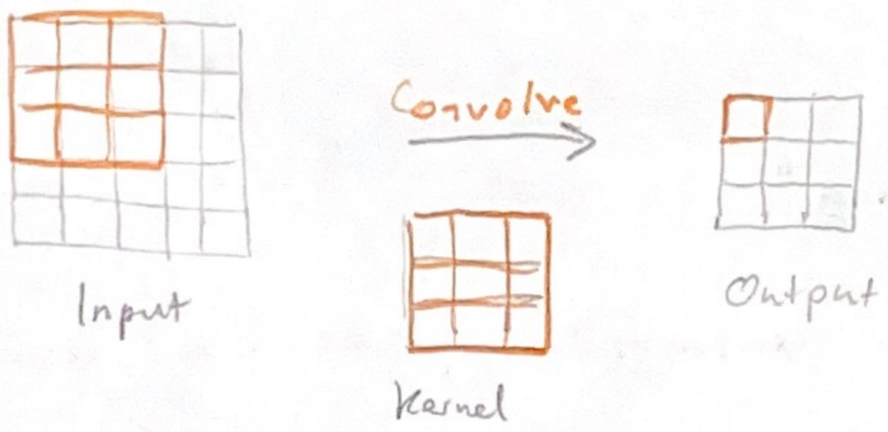


Discrete:

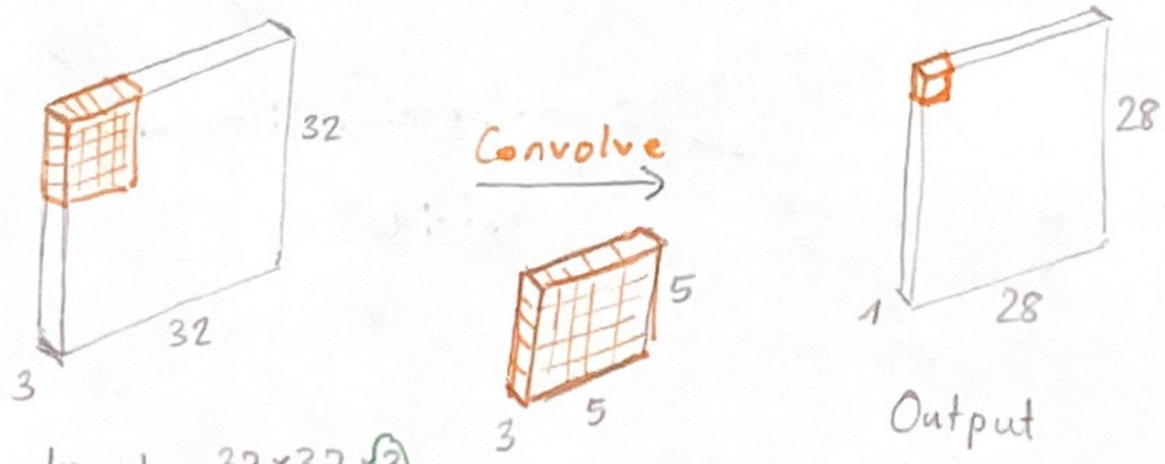
1D: (w/o padding)



2D: (w/o padding)



RGB images 3D: (W/o padding)



Input: 32×32×(3)

Kernel/Filter 5×5×(3)

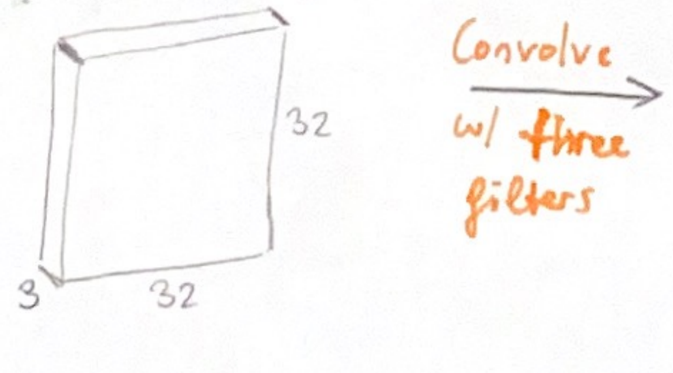
"depth" dim of input and filter must match (the number of channels)

Output 28×28×1

One channel/~~slice~~ one Activation map / Feature map

Using multiple filters: (a convolutional layer)

32×32×3 image



→ layer is defined by
1. kernel height/width
2. number of different kernels

⇒ Each filter captures a different characteristic

Activation maps 28×28×3

output channels

- Conv. layer dimensions:

↓

does "Feature extraction"

(computes feature in a given region)

Input: $N \times N$

Filter: $F \times F$ ⇒ Output: $\left(\frac{N-F}{S}+1\right) \times \left(\frac{N-F}{S}+1\right)$

Stride: S

Fractions are illegal

→ certain strides not possible

- Padding:

→ To avoid feature maps shrinking too quickly

→ To use corner pixels multiple times

$$\text{output size} = \left(\left\lfloor \frac{N+2P-F}{S} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{N+2P-F}{S} \right\rfloor + 1\right)$$

floor operation

e.g. zero padding

→ Valid convolution: no padding

→ Same convolution: output = input size ⇒ $P = \frac{F-1}{2}$

- Number of parameters:

Each filter has $\underbrace{F \times F \times \text{in\_channels}}_{\text{weights}} + \underbrace{1}_{\text{bias}}$ parameters

- Pooling: "Is downsampling"

↳ does "Feature selection" (picks the strongest activation in a region)

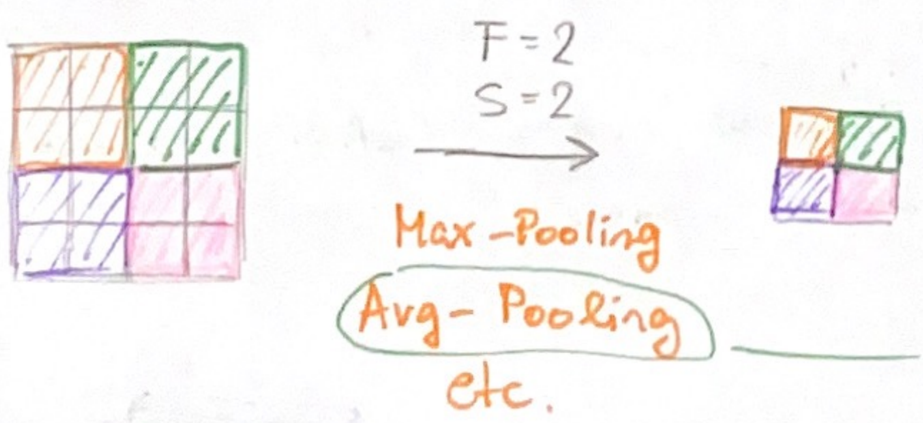
Input: size $W_{in} \times H_{in} \times D_{in}$

Output: $W_{out} = \frac{W_{in}-F}{S}+1$ (F: spatial filter extent, S: stride)

$H_{out} = \frac{H_{in}-F}{S}+1$

$D_{out} = D_{in}$

} 2 hyperparameters



typically used deeper in the network

- Final FC layer:

↔ Typically one or two FC layers to make final decision

- Receptive field: "the input pixels that are connected to a single output pixel"

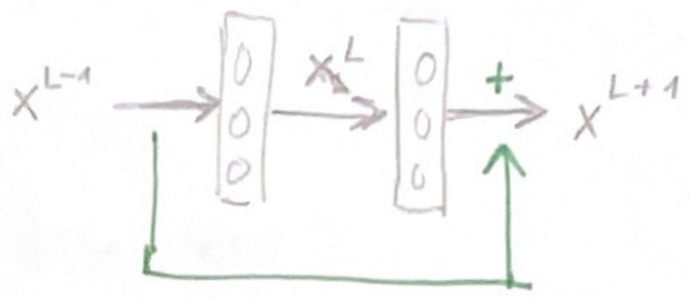
↔ Spatial extent of the connectivity of a convolutional filter

⇒ "Information from larger parts of an image are condensed into a single pixel"

12DL 10: CNN-2

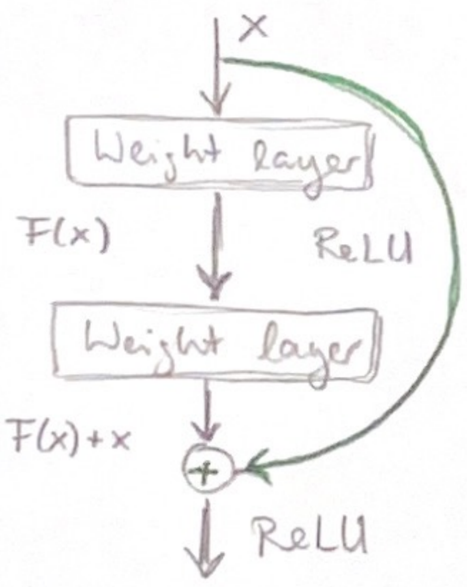
- Common performance metrics in image detection:
  - ✱ Top-1 score: If the predicted class is the target label
  - ✱ Top-5 score: If the top 5 predicted classe contain the target
  - ✱ Top-5 error: percentage of samples for which the top 5 predicted classes did not contain the target

- Skip connections:
  - ✱ Problem of vanishing & exploding gradients in deep nets (with normal nets, the performance degrades at some depth)
    - → Residual Block (used in ResNet)



$$\Rightarrow x^{L+1} = f(W^{L+1} x^L + b^{L+1} + x^{L-1})$$

  - ✱ The convolution within the block must be SAME, such that the size of x^{L-1} is the size of x^{L+1}
    - → Otherwise convert the dims w/ a matrix of learned weights or zero padding

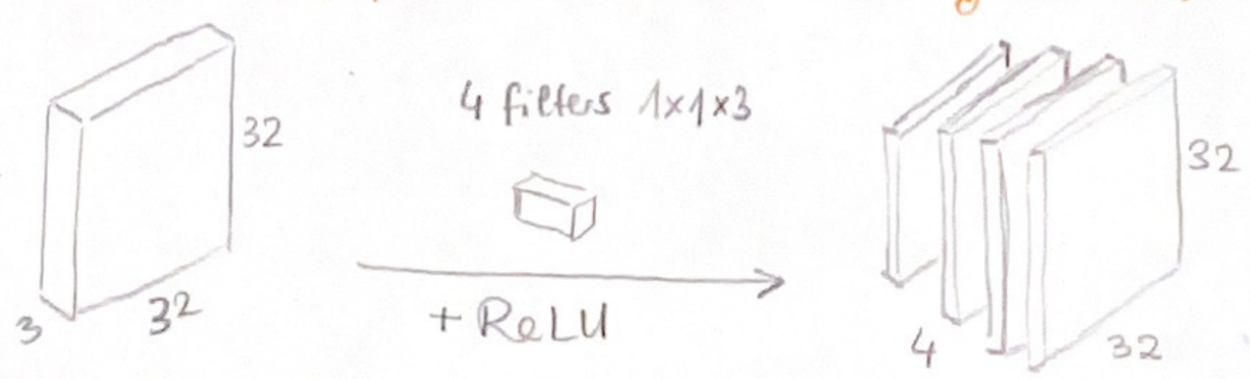


  - ✱ Why do ResNets work?
    - If the layers in the ResBlock hurt performance, the network can "turn them off" by learning to produce the identity (if all weights & biases are 0)

    ⇒ Guaranteed to not hurt performance (can only improve)

⇒ "Revolution of depth"

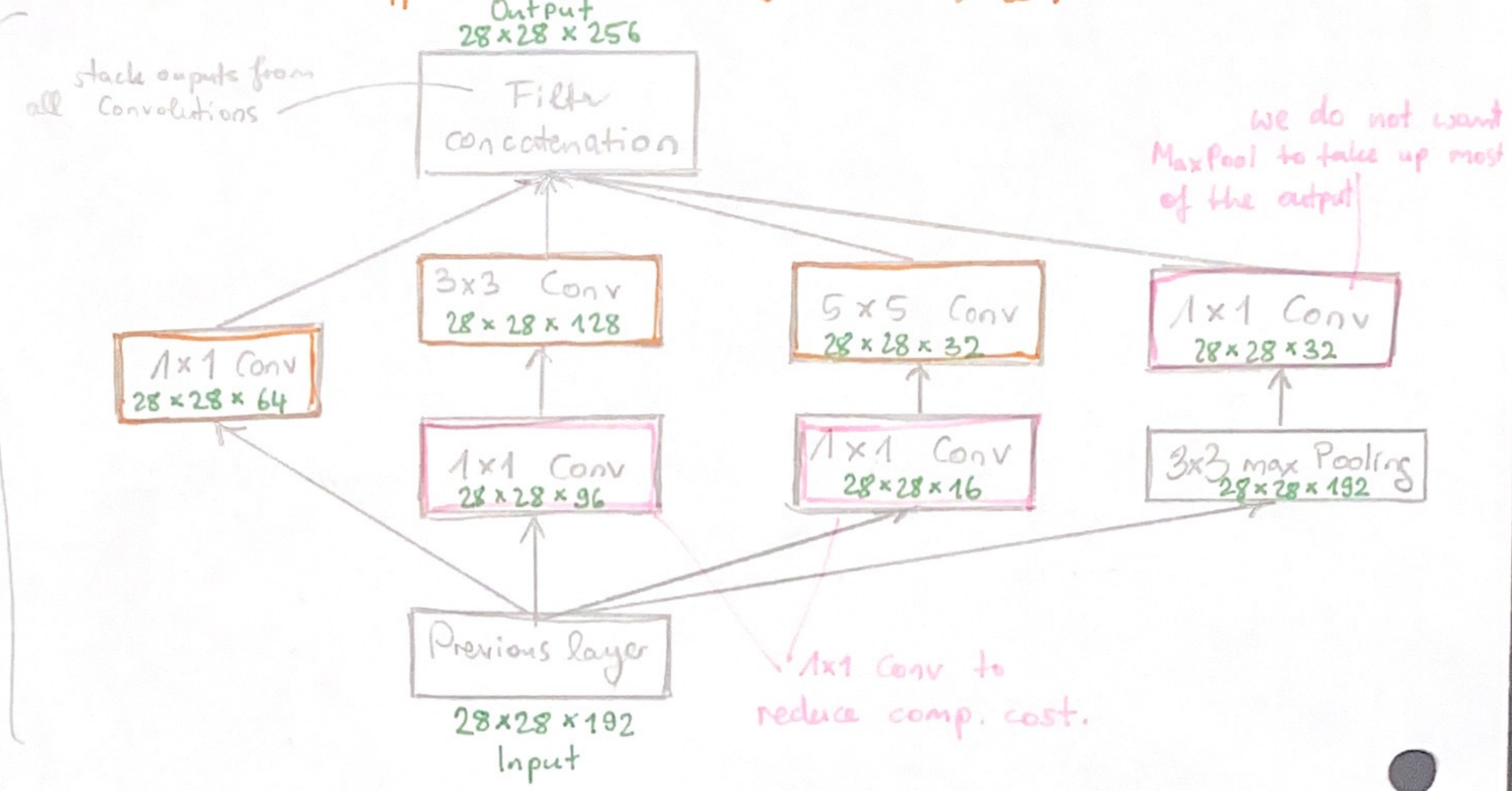
- 1×1 Convolutions:
  - → 1×1 Kernel: Keeps the dims & scales the input



  - ⇒ Used to shrink the num of channels but keep spatial resolution
    - + Adds non-lin.
      - → More complex func can be learnt
      - → Subsequent computations are cheaper

- Inception layer:

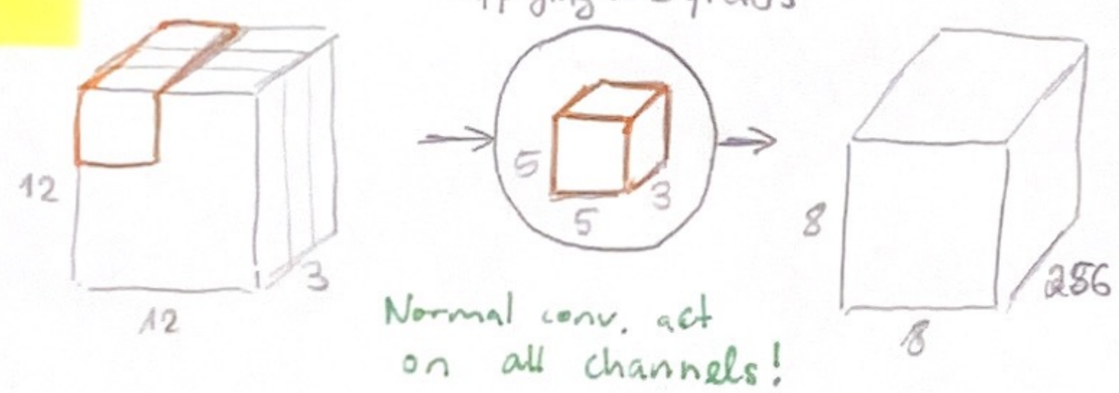
→ Use filters w/ different sizes at the same time!



* Extra max pool layers between Inception Blocks to reduce dimensionality
→ to avoid choosing a kernel size + reduces computational cost

- Xception: "Extreme version of an inception."

→ Uses depth wise separable convolutions instead of normal convolutions



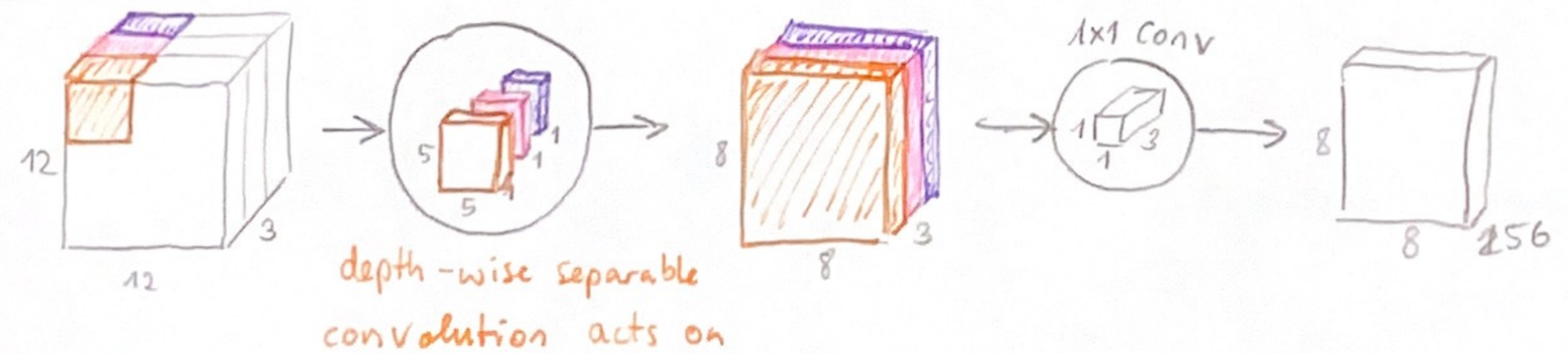
256 kernels of size 5×5×3 in 8×8 locations

Multiplications: $256 \times 5 \times 5 \times 3 \times 8 \times 8$

$= 1\,228\,800$

Same input

Same output



(used in Xception Net)

Multiplications: $5 \times 5 \times 3 \times 8 \times 8 = 4800$

3 kernels of size 5×5×1 in 8×8 locations

$256 \times 1 \times 1 \times 3 \times (8 \times 8) = 49\,152$

256 kernels of size 1×1×3 in 8×8 locations

- Fully convolutional network: → Replace lin. layers with equivalent conv. layers

↳ Continuously reduces the width and height

→ In semantic segmentation we want to output the input size

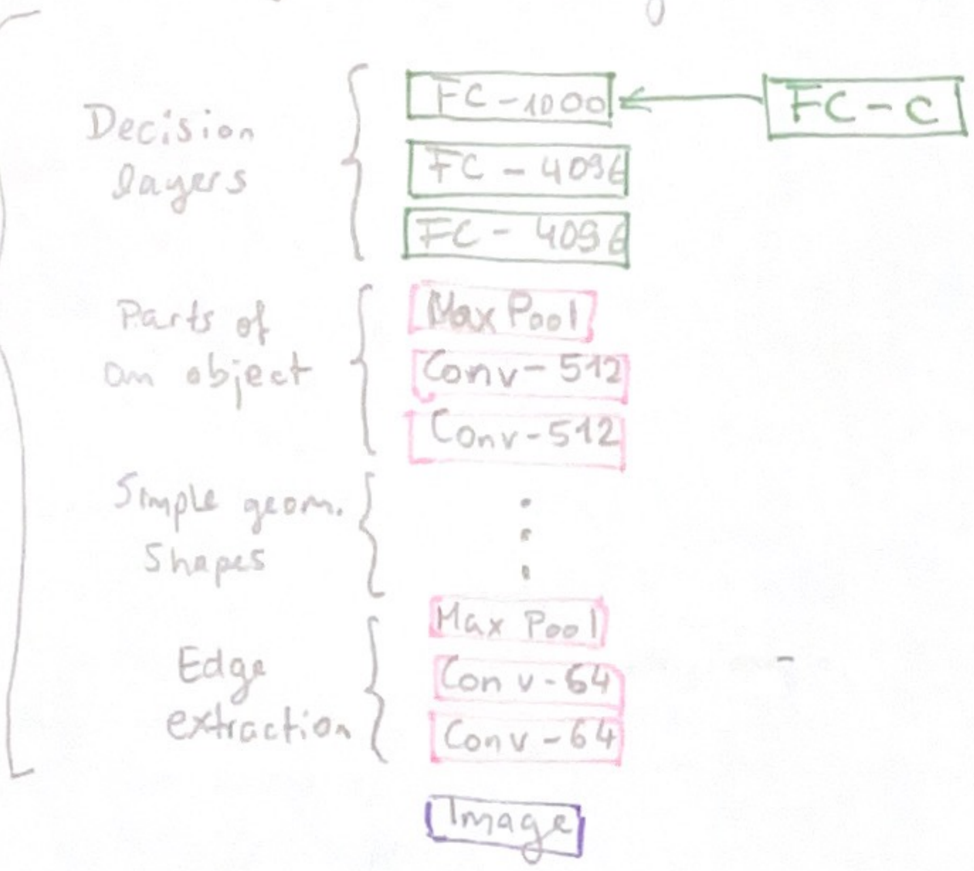
↳ Upsampling: 1) Interpolation (few artefacts)

2) Transposed convolution (unpooling + learned filter → artefacts)

12DL 11 : RNNs

- Transfer learning: · → Use pretrained model learned for another setting
  → Finetune it on small dataset

(eg. low-level feature extraction is similar for all RGB images)

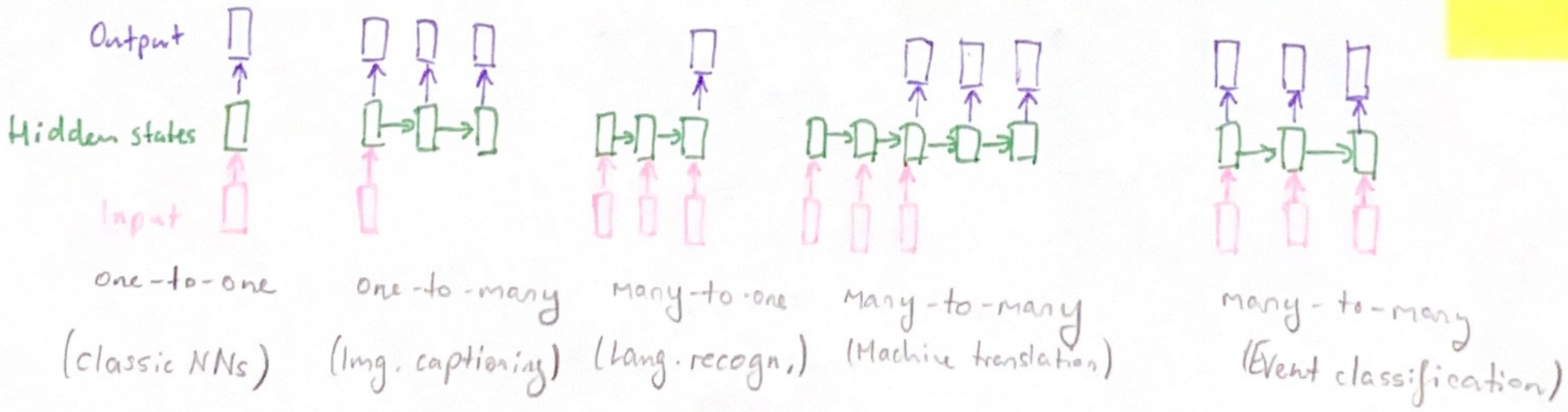


→ When to use transfer learning:
1) If both tasks have the same input
2) When we have more data for task 1 than for task 2
3) When low-level features from T1 could be useful for T2

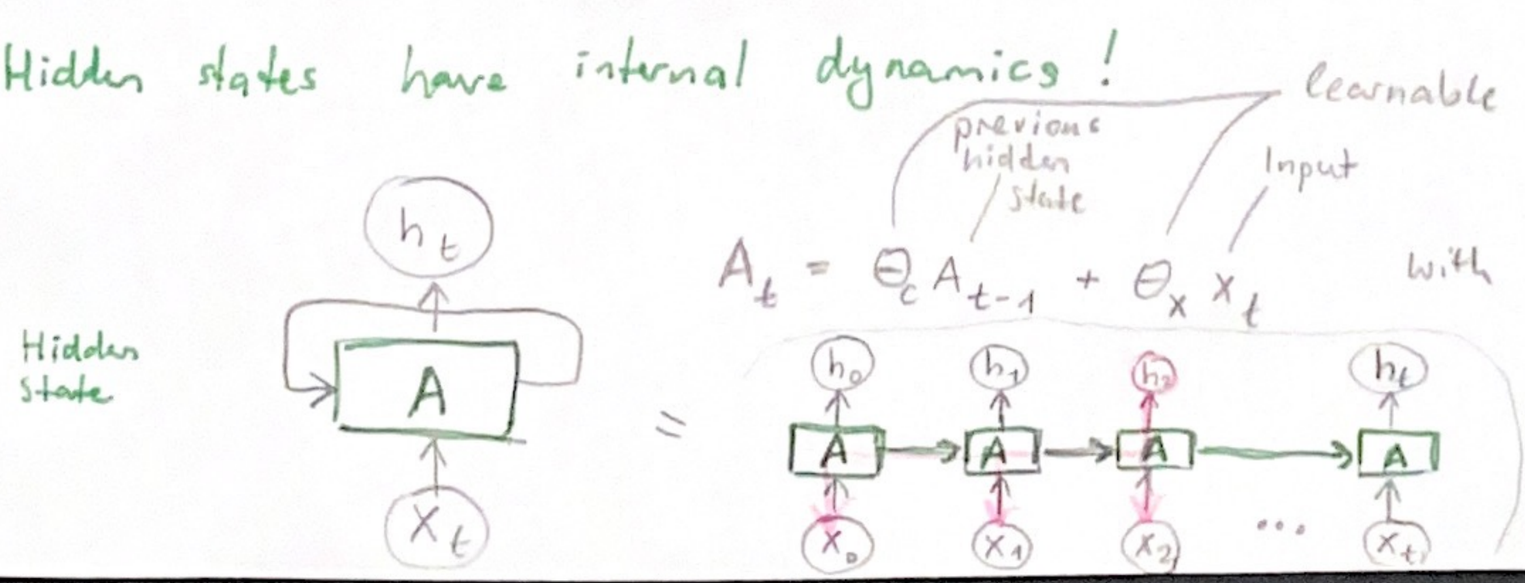
- Recurrent Neural Networks: RNNs

→ Process sequence data (Input/output can be sequences)

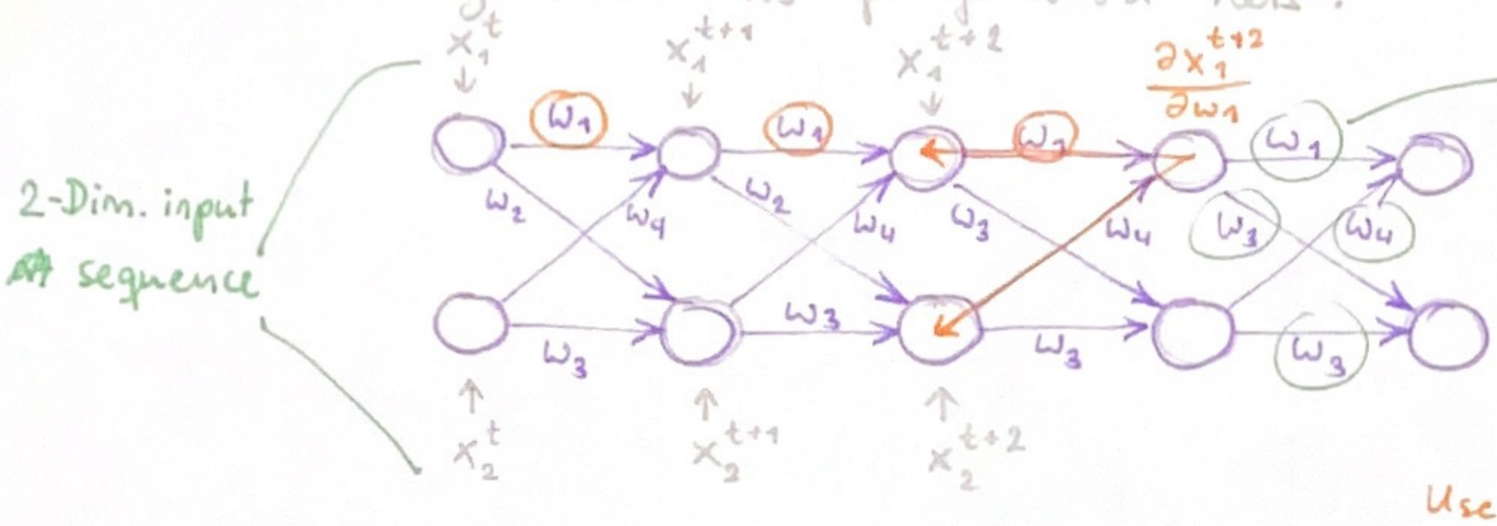
Input-Output-mappings:



→ Hidden states have internal dynamics!



Unrolling RNNs as feed forward nets:



the same weights at every time step

all the way to t=0

Use the chain rule in Backprop and add the derivatives at different time steps for each weight.

* It is a major challenge to keep enough information in the hidden states for long-term-dependencies (when an output depends on input long before it)

$$A_t = \Theta_c A_{t-1} + \Theta_x X_t \overset{\text{forgetting the input}}{\approx} \Theta_c^t A_0$$

(~~grad~~ same weights are multiplied over and over again)

small weights → vanishing grad.

large weights → exploding grad.

Eigen-decomposition

$$\Theta = Q \Lambda Q^T$$

$$\Rightarrow A_t = Q \Lambda^t Q^T A_0$$

↳ if largest eigenvalues have magnitude $\begin{cases} < 1 \Rightarrow \text{vanishing grad} \\ > 1 \Rightarrow \text{expl. grad.} \end{cases}$

↳ grad. clipping

⇒ Set weight / eigenvalues to 1

- Long-Short-Term-Memory: (LSTM)

element-wise operations:

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

previous state, current state

Forget gate: decides when to erase the cell

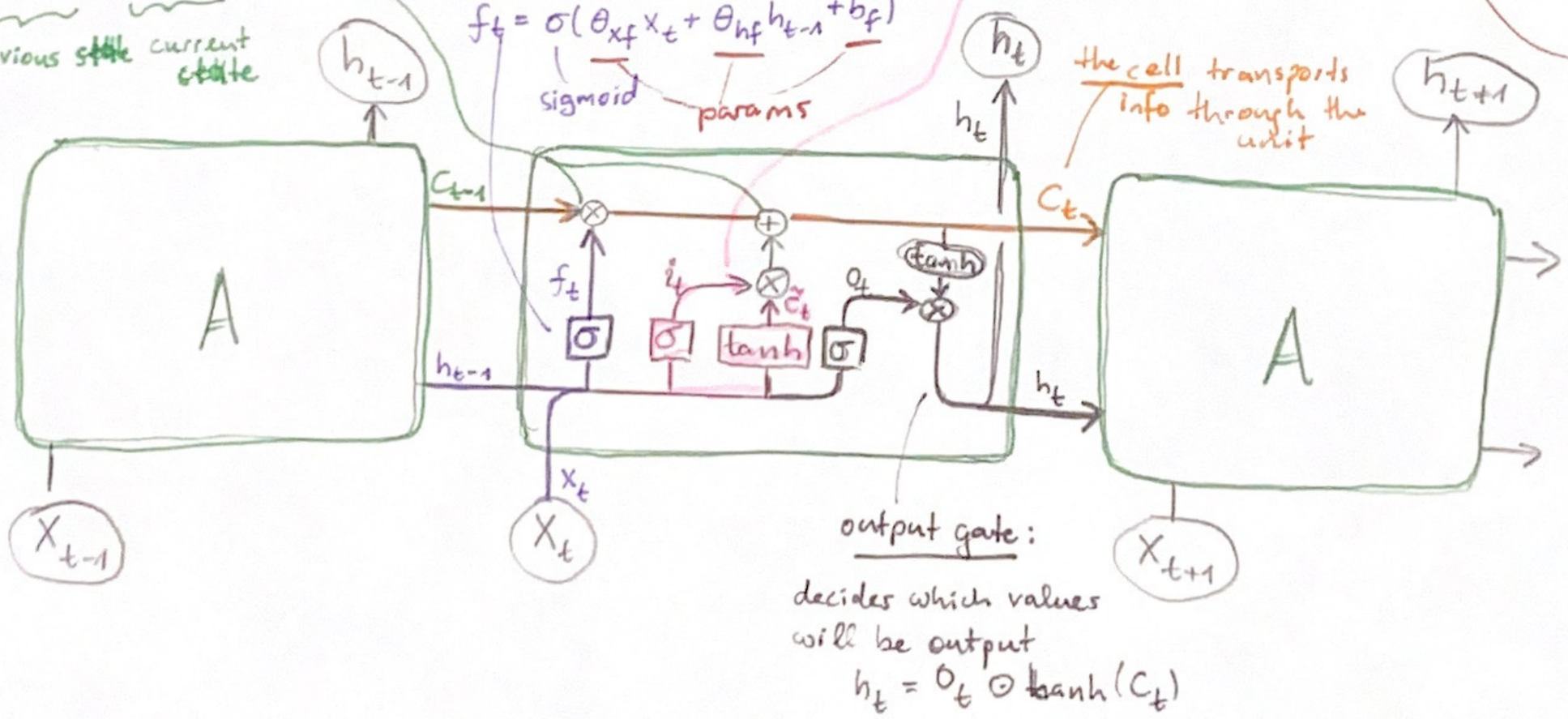
$$f_t = \sigma(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f)$$

sigmoid, params

Input gate: decides which values will be updated. $i_t = \sigma(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i)$

parameters

the cell transports info through the unit



output gate:

decides which values will be output

$h_t = o_t \odot \tanh(C_t)$

All inputs into a cell must have the same dims

The orange line is a highway for the gradient in backprop.

cell state

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

f_t is 1 for important info

forget gate deals with past information (only here over and over multiplication w/ weights)

only here vanishing grad. from activation

⇒ No problems w/ vanishing gradients

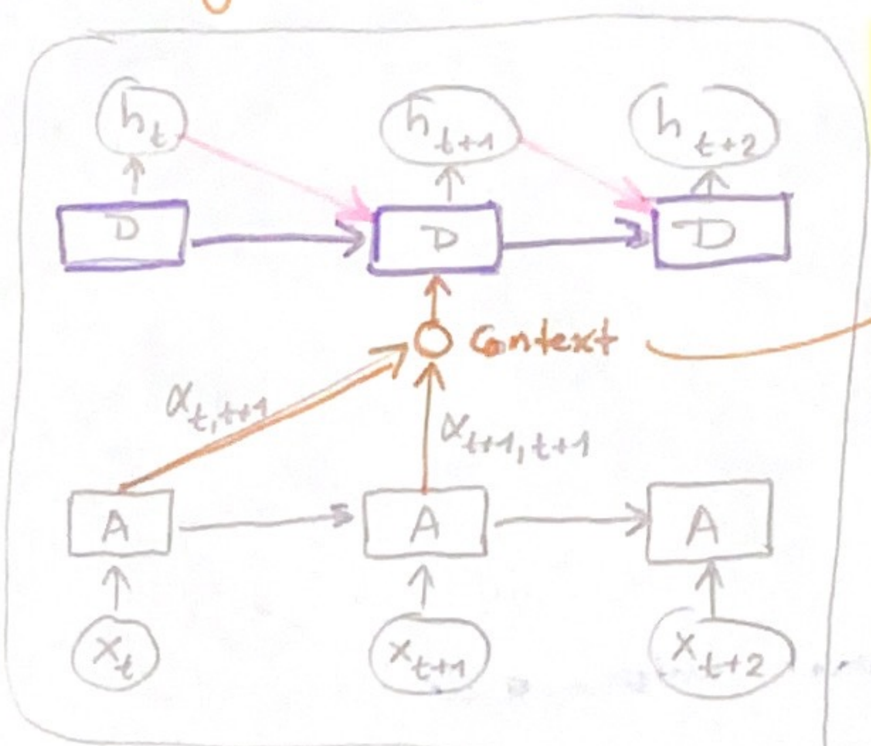
12DL 12: Advanced deep learning topics

- **Attention:** → wants to tackle the problem w/ long term dependencies (similar to LSTM)

Architecture:

* Input to decoders:
  - prev. decoder hidden state
  - prev decoder output
  - attention / context

=D The decoder then processes the information.



relationship of different time steps w.r.t. the current output at t+1

- **Transformers:** → get rid of the RNN and only use attention

RNN: No connections between the value at a time step w/ the val. at a late time step

Transformers: All values in the sequence are connected to all other values!

- **Graph neural networks:**

Node = a concept, Edge = connection between concepts

* General idea: - Input is a graph w/ optional node & edge feature vectors
  → Info is propagated across the graph for several iterations (each iteration is a "hidden layer")
  → Output is a graph w/ updated, context-aware node and edge feature vectors.

(e.g for multi-object tracking)

* During propagation, we have

- "node to edge" updates:

$$h_{(i,j)}^{(\ell)} = \mathcal{N}_e^{(\ell)}\left(\left[h_i^{(\ell-1)}, h_{(i,j)}^{(\ell-1)}, h_j^{(\ell-1)}\right]\right)$$

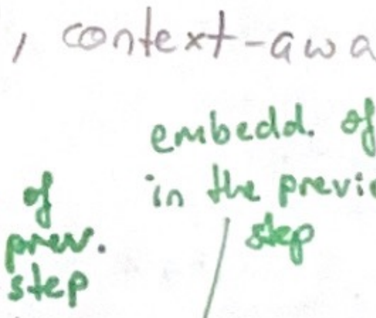
updated embedding of edge (i,j) | embedding of node i in the prev. msg. passing step | embedd. of edge (i,j) in the previous update step | node j | learnable func w. shared weights across the entire graph

- "edge to node" update:

→ After an edge update, each edge embedding has info about its incident nodes.
⇒ Use edge embedding to update nodes.

$$m_i^{(\ell)} = \Phi\left(\left\{h_{(i,j)}^{(\ell)}\right\}_{j \in N_i}\right)$$

chosen msg for node i | order-invariant operation (sum/max/mean) | neighbors of node i

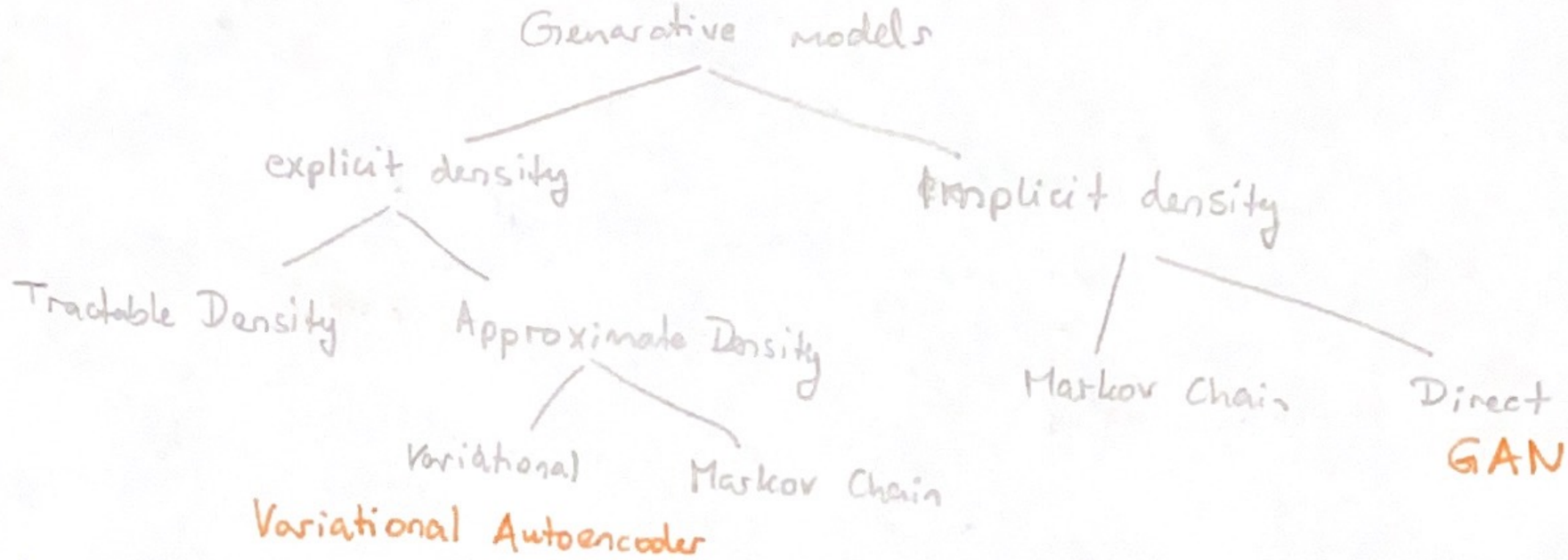


$$h_i^{(\ell)} = \mathcal{N}_v^{(\ell)}\left(\left[m_i^{(\ell)}, h_i^{(\ell-1)}\right]\right)$$

new embedding for node i | learnable func with weights shared for all nodes | previous embedding for node i

- Generative models:

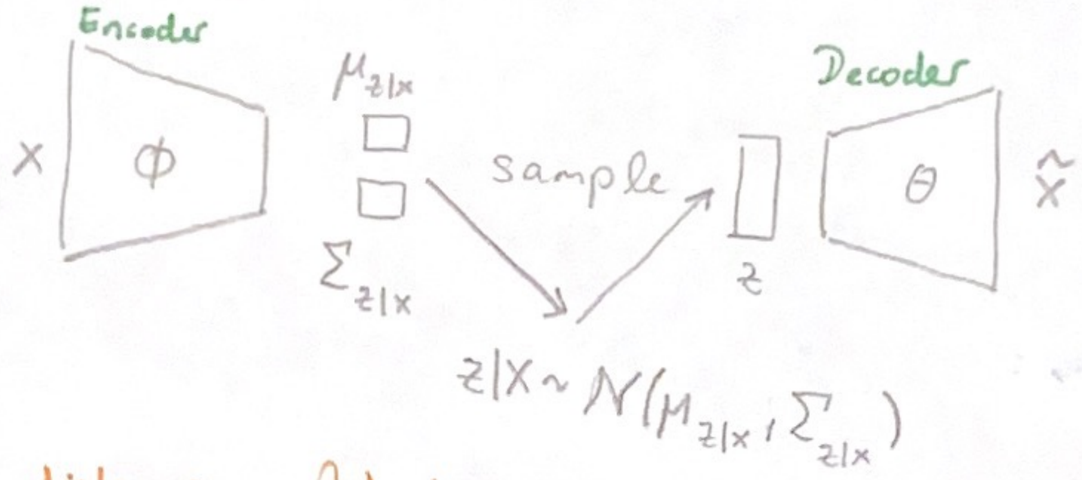
→ Given training data generate "realistic" models from the same distribution



→ Variational autoencoder:

Goal: Sample from latent distr. to generate new outputs.

=D Latent space is Gaussian



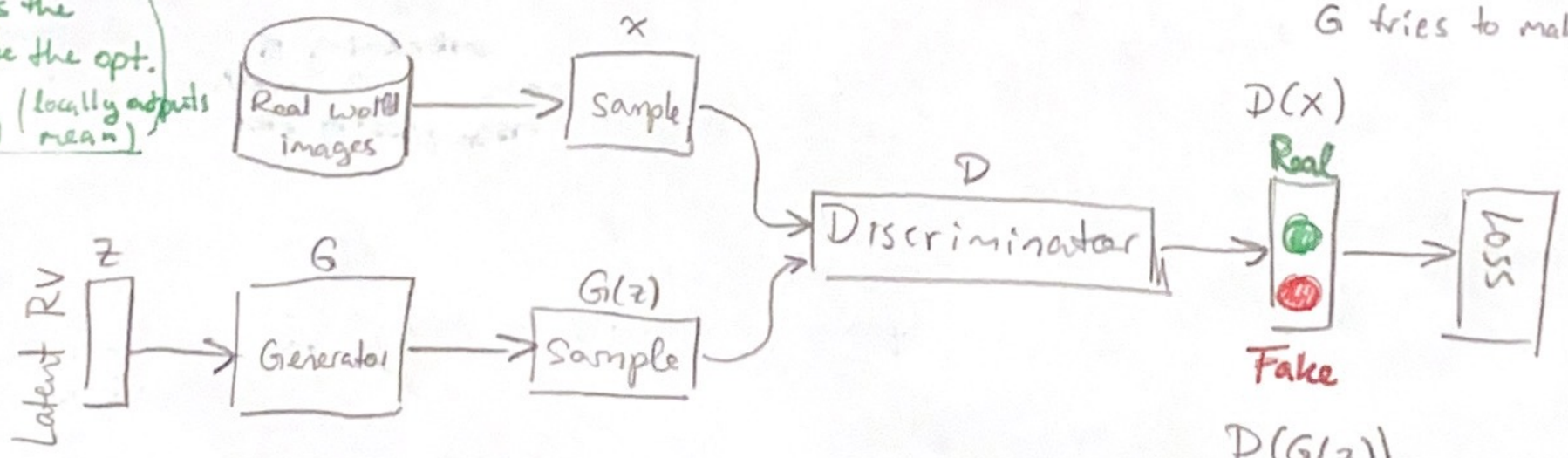
→ Prob. distr. in latent space → sample from it
→ interpretable (sample specific head pose / degree of smile)

- Generative Adversarial Networks: (GANs)

The L2-loss in Autoencoders distributes the error equally among the image
→ for constant imgs the mean would be the opt.
=D Result is blurry (locally outputs mean)

→ Learn the loss function

On the real data: D(x) tries to be near 1
On the fake data: D tries to make D(G(z)) ≈ 0
G tries to make D(G(z)) ≈ 1



=D Discriminator loss: $J^{(D)} = -\frac{1}{2} E_{x \sim p_{data}} \log D(\vec{x}) - \frac{1}{2} E_z \log(1 - D(G(z)))$

real image should be real · gen. img. should be fake

BCE

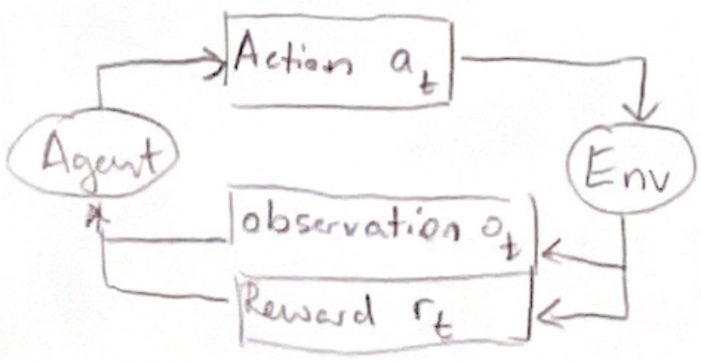
Generator loss: $J^{(G)} = -J^{(D)}$

Components of RL agents

* Policy: mapping from state to action $\pi(s) = a$
* Value-/Q-func: How good is a state/state-action-pair → future reward

- Reinforcement Learning:

→ Given sequential data, learn by interaction with the environment



* data is non i.i.d. (time matters / it is sequential)
* Actions affect the env (change future input)
* No supervisor (target is approximated by the reward signal)
* Markov assumption: "The state only depends on the previous state"