PROBABILISTIC AND DIFFERENTIABLE PROGRAMMING

V3: Deep Learning I

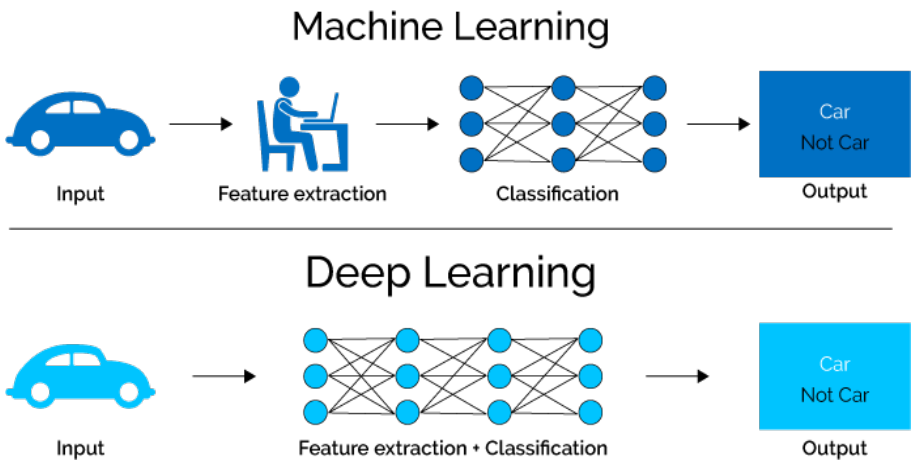
Özgür L. Özçep

Universität zu Lübeck

Institut für Informationssysteme

Today's Agenda

1. Need for Deep



2. Tackling Deep Problems
   - 2.1 Instable Gradient
   - 2.2 Overfitting

3. Convolutional networks

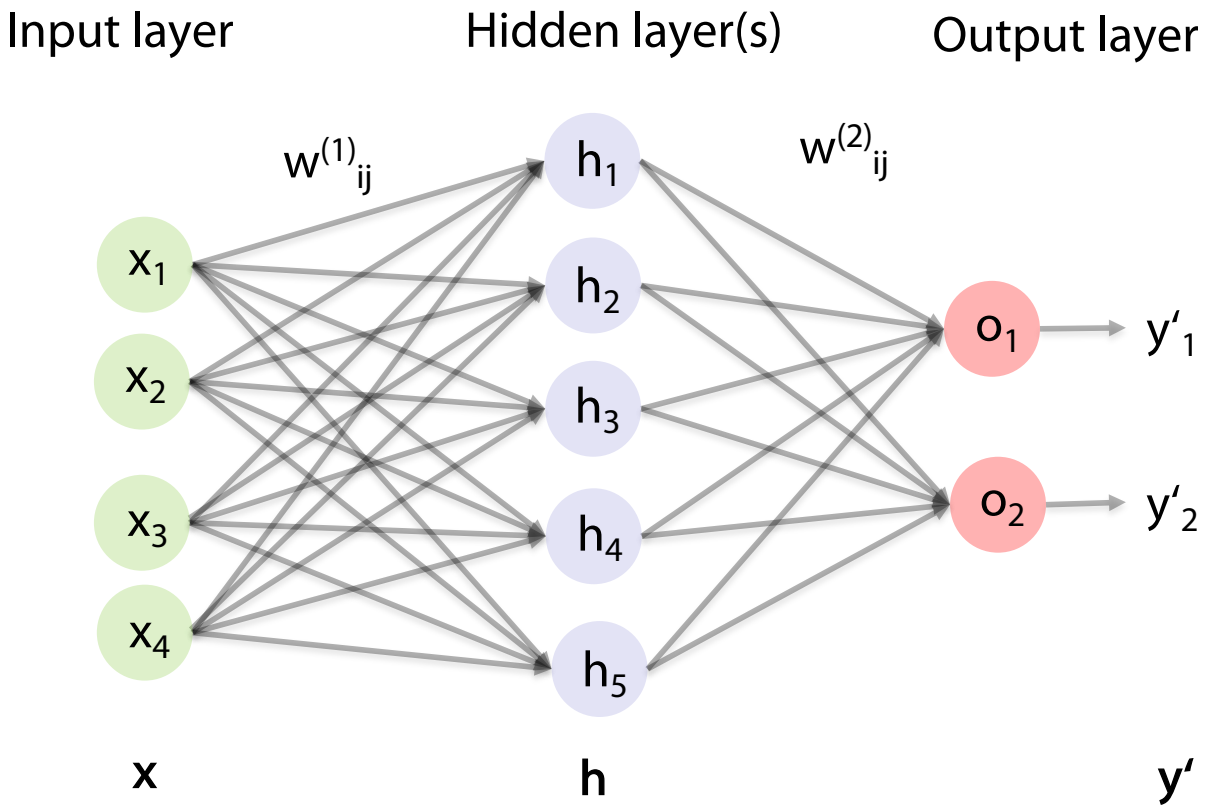
What is Deep Learning (DL)?

- Deep learning is based on function composition
  - Feedforward networks: $\boldsymbol{y} = f\left(g\left(\boldsymbol{x}, \boldsymbol{\theta}_g\right), \boldsymbol{\theta}_f\right)$
    Often with relatively simple functions
    (e.g. $f\left(\boldsymbol{x}, \boldsymbol{\theta}_f\right) = \sigma(\boldsymbol{x}^\top \boldsymbol{\theta}_f)$)
  - Recurrent networks:
    $\boldsymbol{y}_t = f(\boldsymbol{y}_{t\_1}, \boldsymbol{x}_t, \boldsymbol{\theta}) = f(f(\boldsymbol{y}_{t\_2}, \boldsymbol{x}_{t\_1}, \boldsymbol{\theta}), \boldsymbol{x}_t, \boldsymbol{\theta}) = \ldots$

- In early days focus of DL on functions for classification

- Nowadays the functions are much more general in their inputs and outputs.

Network view of composed functions



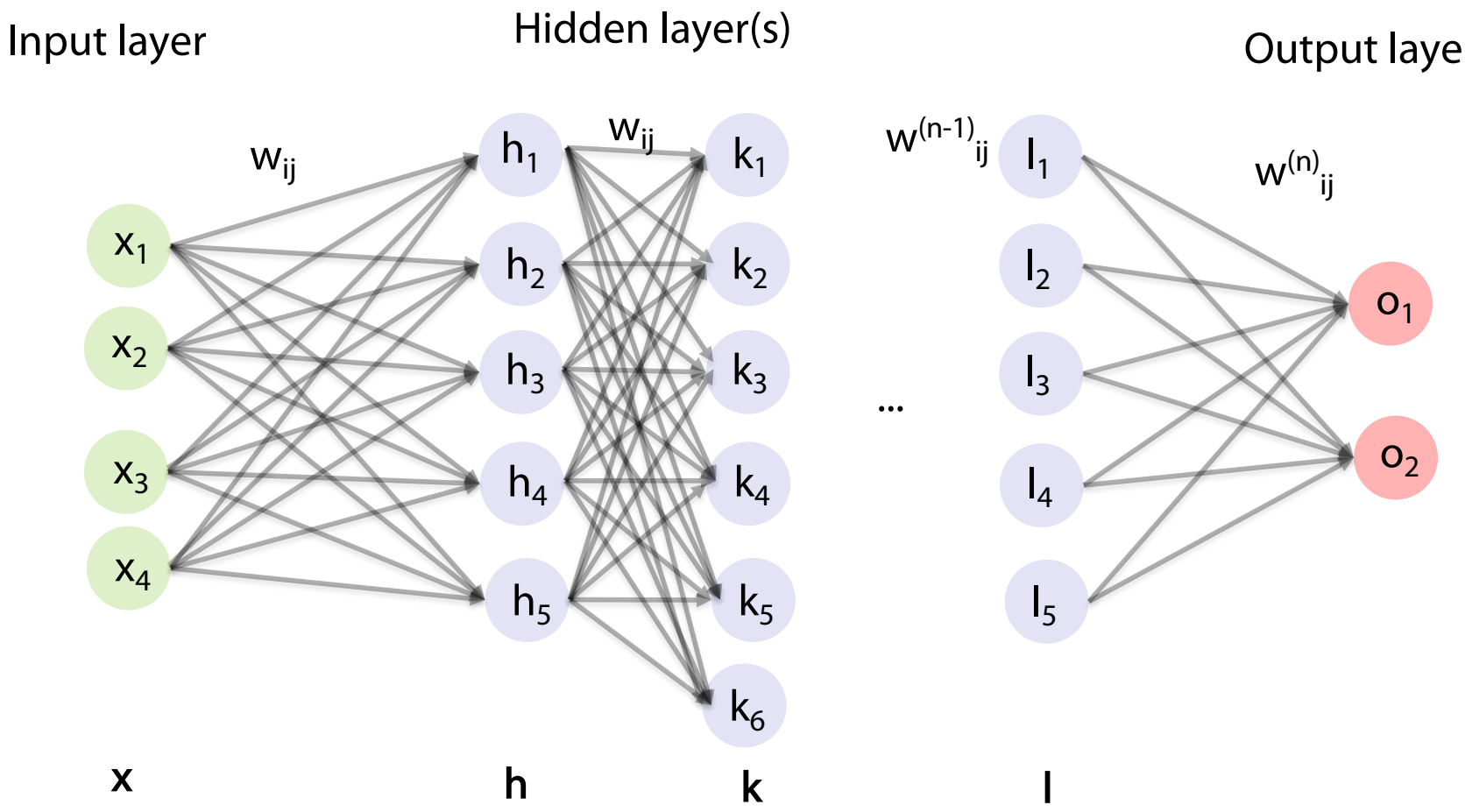
i:	layer i
b_i:	Bias in i
$W^{(i)}$:	weight matrix in i
σ_i:	activation function in i

„States“

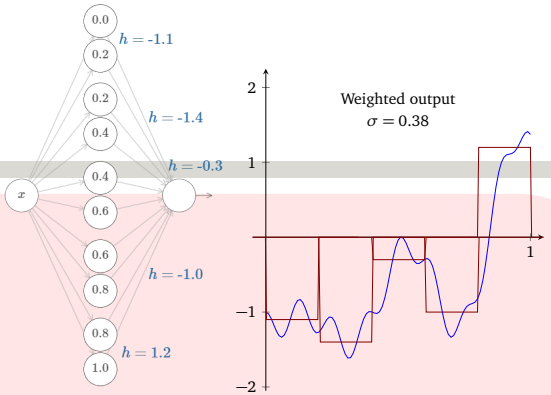
Functions

$$y' = f(g(x; W^{(1)}, b_1); W^{(2)}, b_2) = \sigma_2(W^{(2)} \sigma_1(W^{(1)} x + b_1) + b_2)$$

DL is based on Deep networks



Do we need to go deep?



Universal Approximation Theorem

- Given
  - σ: $\mathbb{R} \rightarrow \mathbb{R}$, nonconstant, bounded, continuous function
  - I_m: m-dimensional hypercube $[0,1]^m$
  - $C(I_m)$: real-valued continuous functions on I_m
- Any function f $\in C(I_m)$ can be approximated by a function of the form $F(x) = \sum_{i=1}^{N} v_i \, \sigma\left(w_i^{\top} x + b_i\right)$

(constants v_i, b_i ,vectors w_i)

i.e. $|F(x) - f(x)| < \varepsilon$ for all $x \in I_m$.

Hence : single-layer networks can represent a wide variety of interesting functions

You need to go deep!

1. High precision (small ε) leads to large number of components in single layer (very large N).
   - worst-case: exponential blow up
   - E.g., parity function better implemented in a deep structure
2. All to good approximation not always good:

   Overfitting and missing generalizability (still holds for DL)
3. We should care about the data generating distribution
   - Real-world data has significant structure
   - Model this structure with hierarchy of increasingly more abstract latent (hidden) factors
4. Experience shows: Deeper networks better to handle

Slide reminder: What is Deep Learning?

- *Deep learning* systems are neural network models similar to those popular in the ’80s and ’90s, with:
  1. some architectural and algorithmic innovations (e.g. many layers, ReLUs, dropout, LSTMs)
  2. vastly larger data sets (web-scale)
  3. vastly larger-scale compute resources (GPU, cloud)
  4. much better software tools (Theano, Torch, TensorFlow)
  5. vastly increased industry investment and media hype

Let‘s have a look into some of the innovations mentioned above in 1.

No problem - no innovation

- We already saw some problems in DL
- We look again into them and others

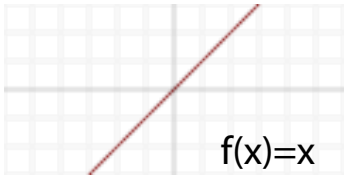
  - Gradient descent is hard
    - Slow learning with MSE
    - vanishing/exploding gradient
  - Overfitting
  - (Horrible symmetries in the data)

INSTABLE GRADIENT

Lets talk about loss

- The choice of loss function depends on the task (e.g. classification/regression/something else)
- Depends also on the activation function of the last layer
  - For numerical reasons many times the activation is computed directly within the loss rather than being part of the model
  - Some classification losses require raw outputs (e.g. a linear layer) of the network as their input (logits)
  - There are are different loss functions for different tasks (MSE, Cross-Entropy, ...)

Loss functions and output

	Multilabel (overlapping)/ Multiclass (disjoint) Classification	Regression
Training examples	$\mathbb{R}^n$ x {class_1, ..., class_n}	$\mathbb{R}^n \, x \mathbb{R}^m$
Output Layer	Sigmoid $f(x) = \frac{1}{1+e^{-x}}$ Soft-max [map $\mathbb{R}^n$ to a probability distribution] $P(y = j \mid \boldsymbol{x}) = \frac{e^{\boldsymbol{x}^\top \boldsymbol{w}_j}}{\sum_{k=1}^{K} e^{\boldsymbol{x}^\top w_k}}$	Linear (Identity) or Sigmoid 
Cost (loss) function	Cross-entropy $L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right)\right]$	Mean Squared Error $L(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$ Mean Absolute Error $L(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left\lvert y^{(i)} - \hat{y}^{(i)}\right\rvert$

Slow learning with MSE

Mean squared error on one single training example

$$l_{MSE}: \quad \mathbb{R}^n \times \mathbb{R}^n \xrightarrow{l} \mathbb{R}$$
$$(\hat{y}, y) \mapsto \left|\left|\hat{y} - y\right|\right|^2$$

- l_{MSE} is the predominant choice for regression problems with linear activation in the last layer
- MSE can cause slow learning (small $\nabla_{W,b} l_{MSE}$), especially if the predictions are very far off the targets : $\nabla_{W,b} l_{MSE} \sim \left|\left|\hat{y} - y\right|\right| \cdot \sigma'(z)$
- Counter intuitive: Big mistakes should lead to big learning steps

If you are curious: the argument in detail

- $\nabla_{W,b} l_{MSE} \sim \left||\hat{y} - y|\right| \cdot \sigma'(z)$
  $= \left||\sigma(z) - y|\right| \cdot \sigma(z)\big(1 - \sigma(z)\big)$

- and $\sigma(z) \in [0,1]$.
- Extreme difference e.g., when $y = 1$ and $\sigma(z) = 0$
- Then z must be negatively large, so $\nabla_{W,b} l_{MSE} = 0$

Better Choice: Binary Cross-Entropy

Binary Cross entropy for case of binary classification

$$l_{BCE} = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

- Properties we expect from a cost function hold
  - $l_{BCE} > 0$
  - $l_{BCE} \approx 0$ iff $\mathrm{y} = 1 \approx \hat{y}$ or $\mathrm{y} = 0 \approx \hat{y}$

- The nice property of BCE in contrast to MSE for sigmoidal layer

$$\frac{\partial l_{BCE}}{\partial w_i} \sim (\hat{y} - y)$$

(The larger the error, the more you learn)

If you are curious $\frac{\partial l_{BCE}}{\partial w_i} \sim (\hat{y} - y)$

- Assume single layer $\hat{y} = \sigma(z) = \sigma(wx + b)$ with sigmoidal activation
- $\frac{\partial l_{BCE}}{\partial w_i} = \frac{\partial}{\partial w_i}(-y \log \hat{y} - (1-y)\log(1-\hat{y}))$

$$
\begin{aligned}
&= \frac{\partial}{\partial w_i}(-y \log \sigma(z) - (1-y)\log(1-\sigma(z))) \\
&= -y\frac{1}{\sigma(z)}\frac{\partial}{\partial w_i}\sigma(z)) - (1-y)\frac{1}{1-\sigma(z)}\left(\frac{\partial}{\partial w_i}(1-\sigma(z)\right) \\
&= -y\frac{1}{\sigma(z)}\sigma'(z)x_i - (1-y)\frac{1}{1-\sigma(z)}(-\sigma'(z)x_i) \\
&= -y\frac{1}{\sigma(z)}\sigma(z)\big(1-\sigma(z)\big)x_i - (1-y)\frac{1}{1-\sigma(z)}(-\sigma(z)(1-\sigma(z)))\,x_i \\
&= -y\,\big(1-\sigma(z)\big)x_i + (1-y)\sigma(z)x_i = x_i(\sigma(z)-y) = x_i(\hat{y}-y)
\end{aligned}
$$

BCE: Intuition

The cross-entropy can be thought of as a measure of surprise.

- $\widehat{y_i}$: probability that some input x_i is of class 1
- $1 - \widehat{y_i}$: probability that x_i is of class 0

- Extreme case $l_{BCE} = \infty$
  - Model believes prob = 0 for some class, yet this class appears

BCE for multiple labels

- In the case of multi-label classification with a network with multiple sigmoidal outputs you just sum the BCE over the outputs:

Binary Cross entropy for case of binary classification

$$l_{BCE} = \sum_{k=1}^{K} -y \log \hat{y}_k \; - (1 - y) \log(1 - \hat{y}_k)$$

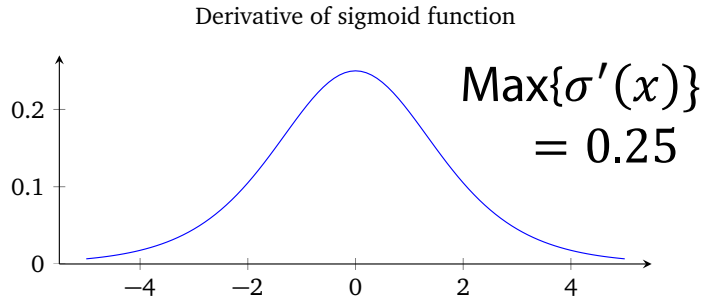
where K is the number of classes of the classification problem, $\hat{y} \in R^K$.

Vanishing and exploding (unstable) gradients

- In training, the gradient may become vanishingly small (or large), effectively preventing the weight from changing its value (or exploding in value).
- This leads to the neural network not being able to train.
- This issue affects many-layered networks (feed-forward), as well as recurrent networks.
- In principle, optimisers that rescale the gradients of each weight should be able to deal with this issue

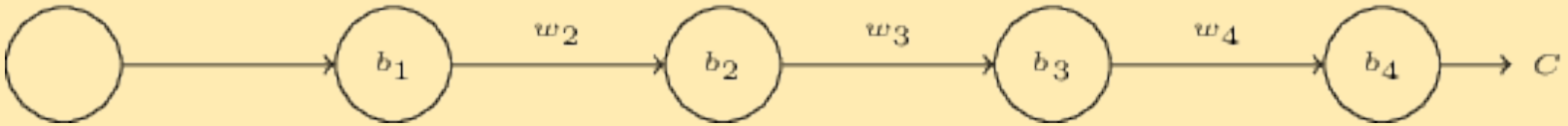
Reminder: Vanishing gradient for sigmoid σ

- Gradient of sigmoid:
  $\sigma'(x) = \sigma(x)(1 - \sigma(x))$



Gradients in linear network of depth 4

$$\frac{\partial C}{\partial b_1} = \underbrace{\sigma'(z_1)}_{\leq 0{,}25} \times w_2 \times \underbrace{\sigma'(z_2)}_{\leq 0{,}25} \times w_3 \times \underbrace{\sigma'(z_3)}_{\leq 0{,}25} \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

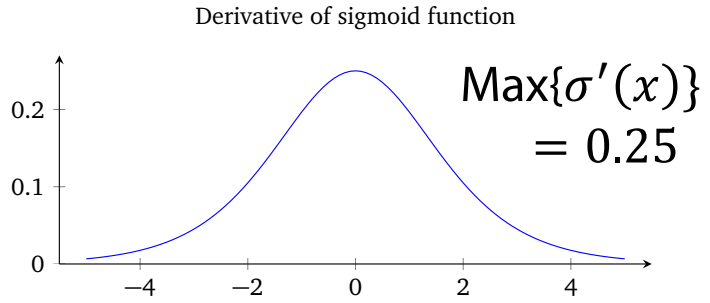


- Assume $|w_i| \leq 1$ (e.g. $w_i \sim N(0,1)$)
- Then: $|w_i\sigma'(z_i)| \leq 0{,}25$
- Exponential decrease from later derivatives to earlier ones due to chain rule

Gradient vanishes moving backwards

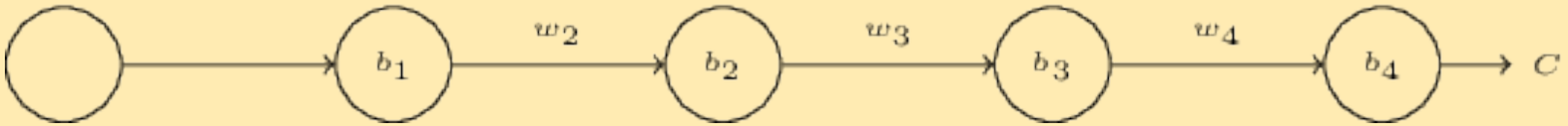
Problem: Exploding gradient for sigmoid σ

- Gradient of sigmoid:
  $\sigma'(x) = \sigma(x)(1 - \sigma(x))$



Gradients in linear network of depth 4

$$\frac{\partial C}{\partial b_1} = \underbrace{\sigma'(z_1)}_{=25} \times w_2 \times \underbrace{\sigma'(z_2)}_{=25} \times w_3 \times \underbrace{\sigma'(z_3)}_{=25} \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



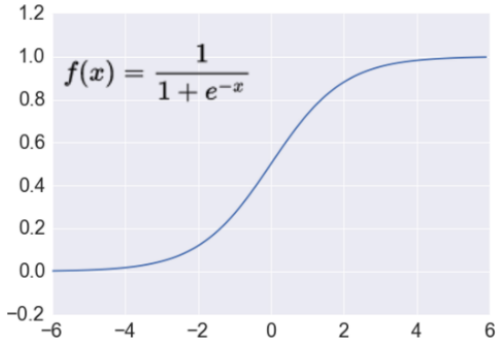
- Assume $w_i = 100$
- Choose b_i such that $z_i = 0$, hence $\sigma'(z_i) = 0.25$
- So $w_i\sigma'(z_i) = 25$
- Exponential increase from later derivatives to earlier ones due to chain rule

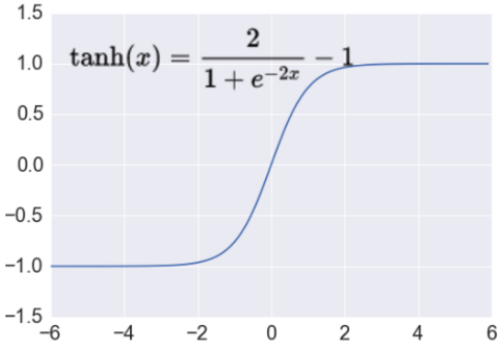
But the main point is instabiliy!
In large networks no
easily balanceing out learning
Rates due to multiplicaiton

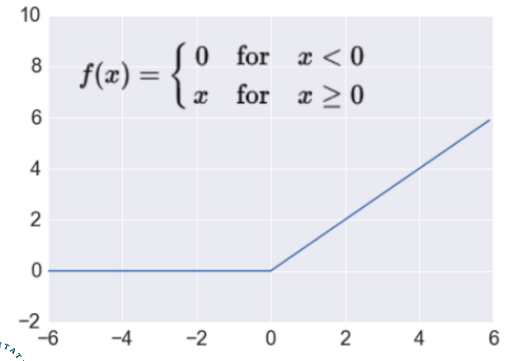
Gradient explodes moving backwards

Tackling vanishing gradients II

http://adilmoujahid.com/images/activation.png







Sigmoid $\mathbb{R}^n \rightarrow [0,1]$

- Takes a real-valued number and "squashes" it into range between 0 and 1.
- Earliest used activation function (neuron)
- Leads to vanishing gradient problem

Tanh: $\mathbb{R}^n \rightarrow [-1,1]$

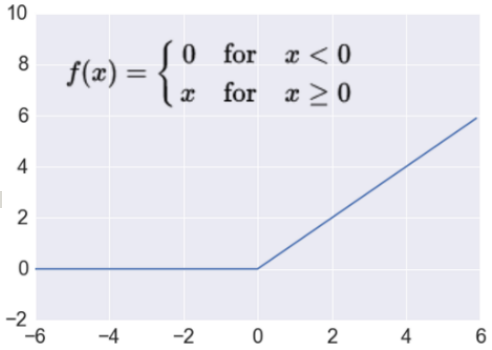
- Takes a real-valued number and "squashes" it into range between -1 and 1
- Same problem of vanishing gradient
- $\tanh(x) = 2sigm(2x) - 1$

Rectified Linear Unit ReLu: $\mathbb{R}^n \rightarrow \mathbb{R}_+^n$

- Takes a real-valued number and thresholds it at 0 Used in Deep Learning
  - Like linear function
  - No vanishing gradient
- But: Dying ReLU
- But: it is not differentiable (need relaxation)

Because no saturation (for positive x)

Problem: Dying ReLUs

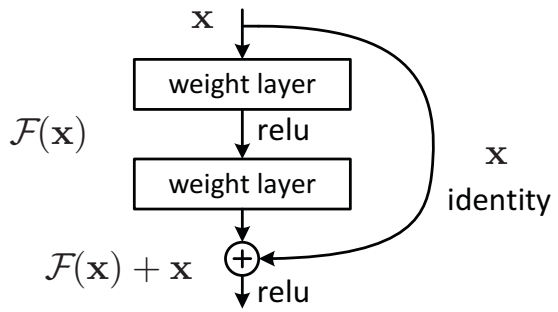


- Gradient is 1 $for\ x > 0$ and 0 otherwise
- Consider $ReLU(\boldsymbol{w}^\top \boldsymbol{x})$
  - What happens if $\boldsymbol{w}$ is initialised badly?
  - What happens if $\boldsymbol{w}$ receives an update that means that $\boldsymbol{w}\top\boldsymbol{x} < 0\ \forall\ \boldsymbol{x}$?
  - These are dead ReLUs - ones that never fire for all training data
- How to tackle?
  - If you get those from the beginning: weight initialisation and data normalisation
  - During training: Maybe η is too big?
  - Leaky ReLU: For x < 0 define with small gradient, e.g. 0.01

x

Tackling vanishing gradients II

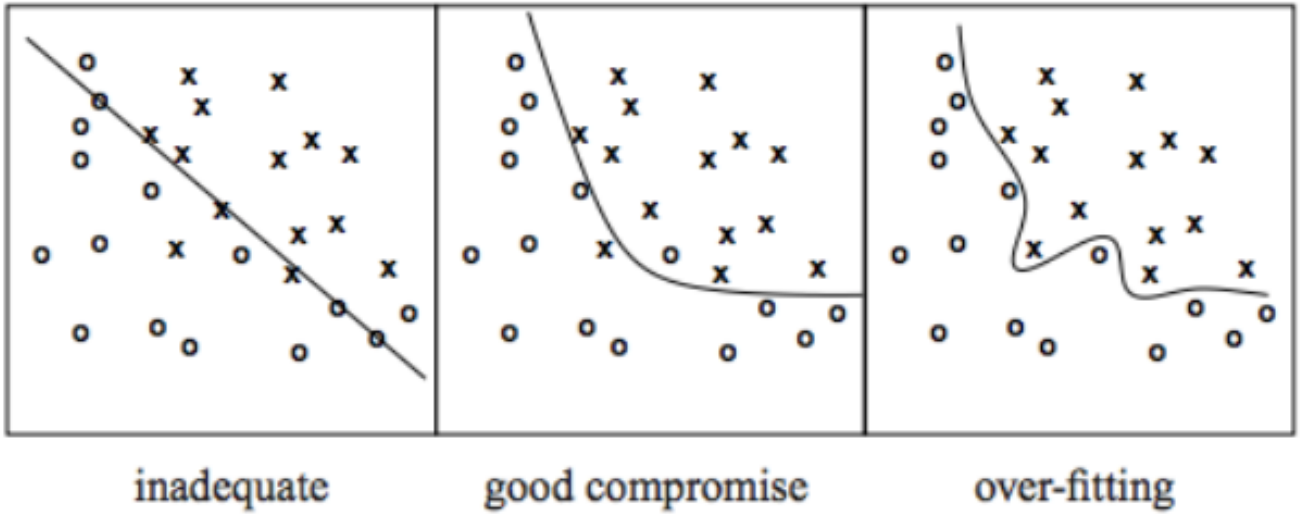
- Residual Networks (ResNets) use skip connections to jump over layers.
- The vanishing gradient problem is mitigated in ResNets by reusing activations from a previous layer.
- Is this the full story though? Skip connections also break symmetries, which could be much more important...



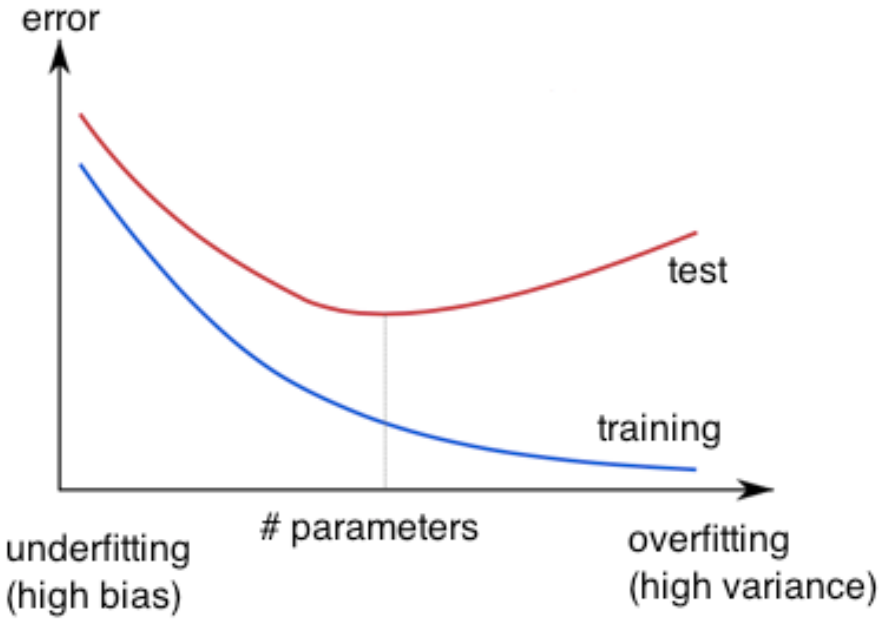
K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. arXiv e-prints, page arXiv:1512.03385, Dec. 2015.

OVERFITTING

Overfitting



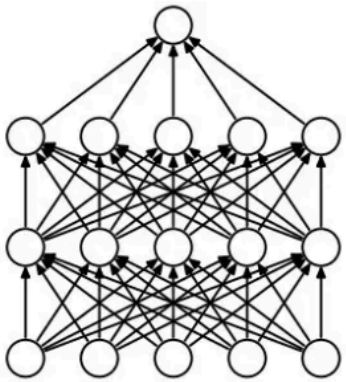
http://wiki.bethanycrane.com/overfitting-of-data

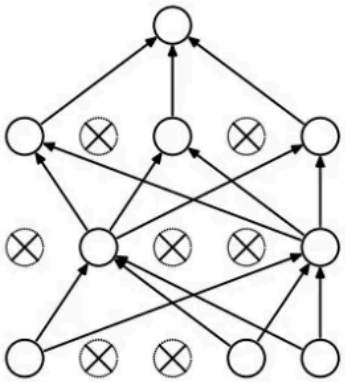


https://www.neuraldesigner.com/images/learning/selection_error.svg

Learned hypothesis may **fit** the training data very well, even outliers (**noise**) but fail to **generalize** to new examples (test data)

Against Overfitting: Regularization





Dropout

- Randomly drop units (along with their connections) during training
- Each unit retained with fixed probability p, independent of other units
- Hyper-parameter p to be chosen (tuned)

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." Journal of machine learning research (2014)

L2 weight decay

- Regularization term that penalizes big weights, added to the objective

$$L_{reg}(\theta) = L(\theta) + \lambda \sum_k \theta_k^2$$

- Weight decay value λ determines how dominant regularization is during gradient computation
- Big weight decay coefficient → big penalty for big weights

Early-stopping

- Use validation error to decide when to stop training
- Stop when monitored quantity has not improved after n (patience) subsequent epochs

Smooting landscape

- E.g. batch normalisation

How and why Dropout works

- HOW
  - In the learning phase, we set a dropout probability for each layer in the network.
  - For each batch we then randomly decide whether or not a given neuron in a given layer is removed.
- WHY
  - Neurons cannot co-adapt to other units (they cannot assume that all of the other units will be present)
  - By breaking co-adaptation, each unit will ultimately find more general features

Dropout = Backprop with random masking

1. Input: random binary mask $\boldsymbol{m}^i$; input vector $\boldsymbol{x} = \boldsymbol{a}^0$
2. Feedforward: For layers i = 1,2, … , M-1

$$\boldsymbol{z}^{\boldsymbol{i}} = \boldsymbol{W}^{(\boldsymbol{i})}\boldsymbol{a}^{\boldsymbol{i-1}} + \boldsymbol{b}_{\boldsymbol{i}} \text{ and } \boldsymbol{a}^{\boldsymbol{i}} = \boldsymbol{\sigma}_{\boldsymbol{i}}(\boldsymbol{z}^{\boldsymbol{i}}) \odot \boldsymbol{m}^i$$

(Remember: ⊙ Hadamard product)

For layer i = M

$$\boldsymbol{z}^{\boldsymbol{M}} = \boldsymbol{W}^{(\boldsymbol{i})}\boldsymbol{a}^{\boldsymbol{i-1}} + \boldsymbol{b}_{\boldsymbol{i}} \text{ and } \boldsymbol{a}^{\boldsymbol{M}} = \boldsymbol{\sigma}_{\boldsymbol{i}}(\boldsymbol{z}^{\boldsymbol{M}}) \odot \boldsymbol{m}^i$$

1. Compute error on last layer

$$\boldsymbol{\delta}^M = \nabla_{\hat{\boldsymbol{y}}} C \odot \sigma'(\boldsymbol{z}^M) \qquad \text{(BP1)}$$

2. Backpropagate error: For i = M-1, M-2, …,

$$\boldsymbol{\delta}^i = (\boldsymbol{w}^{\boldsymbol{i+1}})^\top \boldsymbol{\delta}^{i+1} \odot \sigma'(\boldsymbol{z}^i) \odot \boldsymbol{m}^i \qquad \text{(BP2)}$$

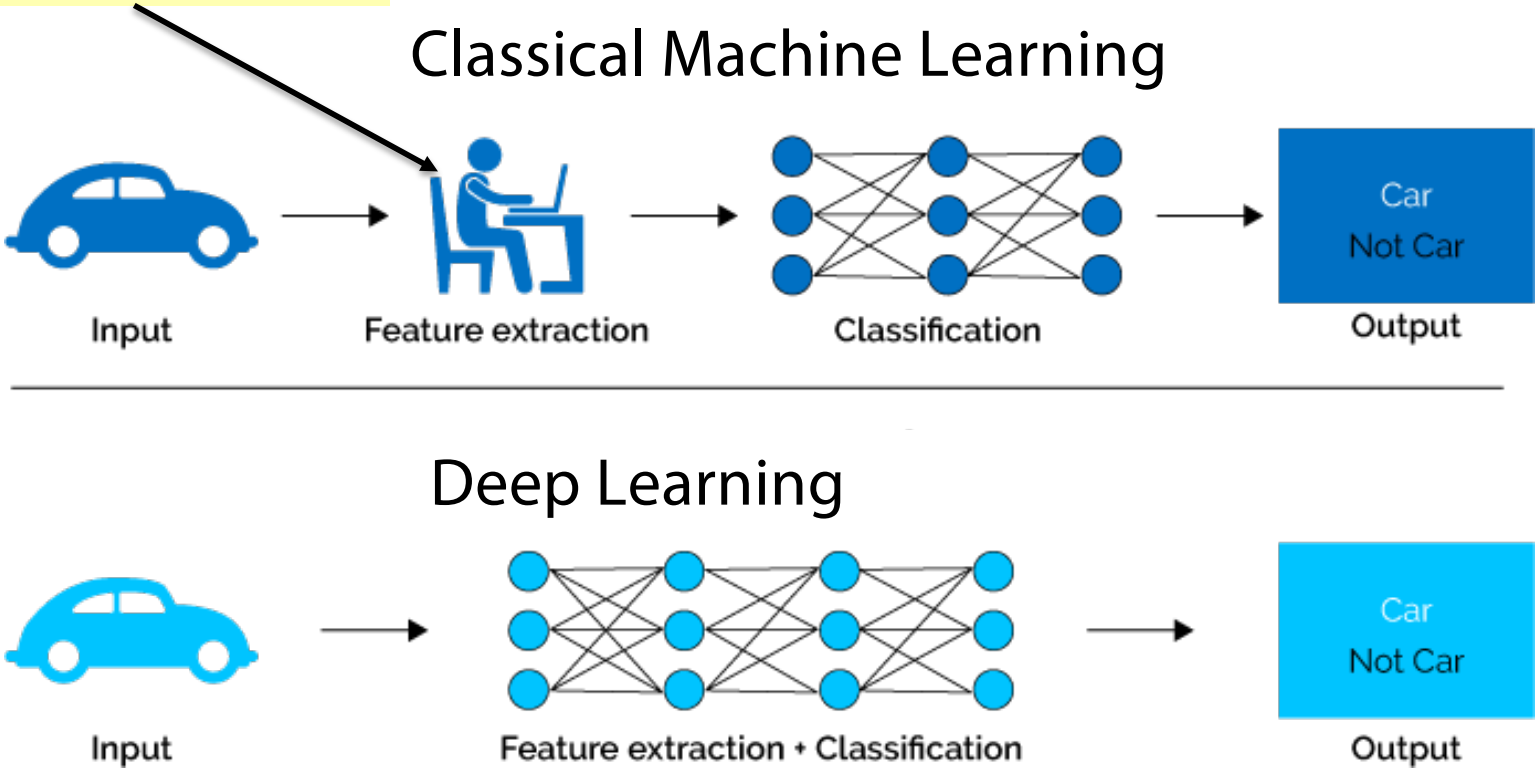
3. Compute gradients

$$\frac{\partial C}{\partial w^i_{jk}} = a^{i-1}_k \delta^i_j \quad \text{and} \quad \frac{\partial C}{\partial b^i_j} = \delta^i_j \qquad \text{(BP3/4)}$$

CONVOLUTIONAL NETWORKS

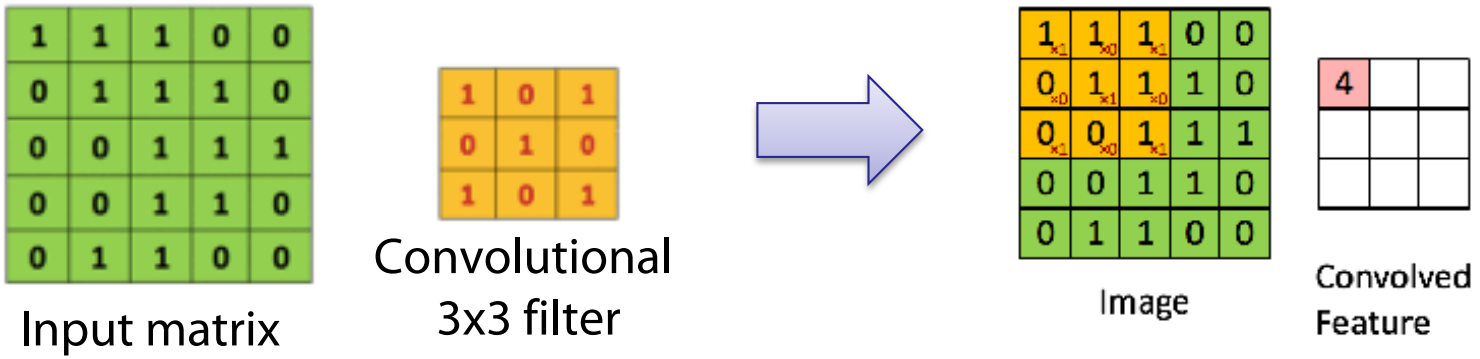
Deep Learning (ad 1.)

Example family car:
we presumed features
price and mileage



Convolutional networks

- More structure: structure of input, local receptive fields (the 1. “rf”)
- Less parameters:
  - weight tying (replication features 2. “rf”),
  - pooling



http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

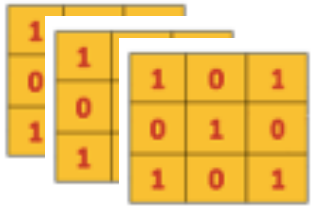
Realization in network: structure of input

- Work with tensors of any number of dimensions
  - (Audio-mono (1D, 1 Channel); audio stereo (1D, 2 channels) colour-video data (3D, color-dim)
  - input is,say, N x P x Q
    - N is the “channels” dimension
    - P , Q are the spatial dimensions,
  - Define convolutional kernel of size N x K x L
- Usually more than single feature -> many kernels -> many feature maps
- We can just add another dimension to the kernel tensor to incorporate convolution with all kernels in one operation:

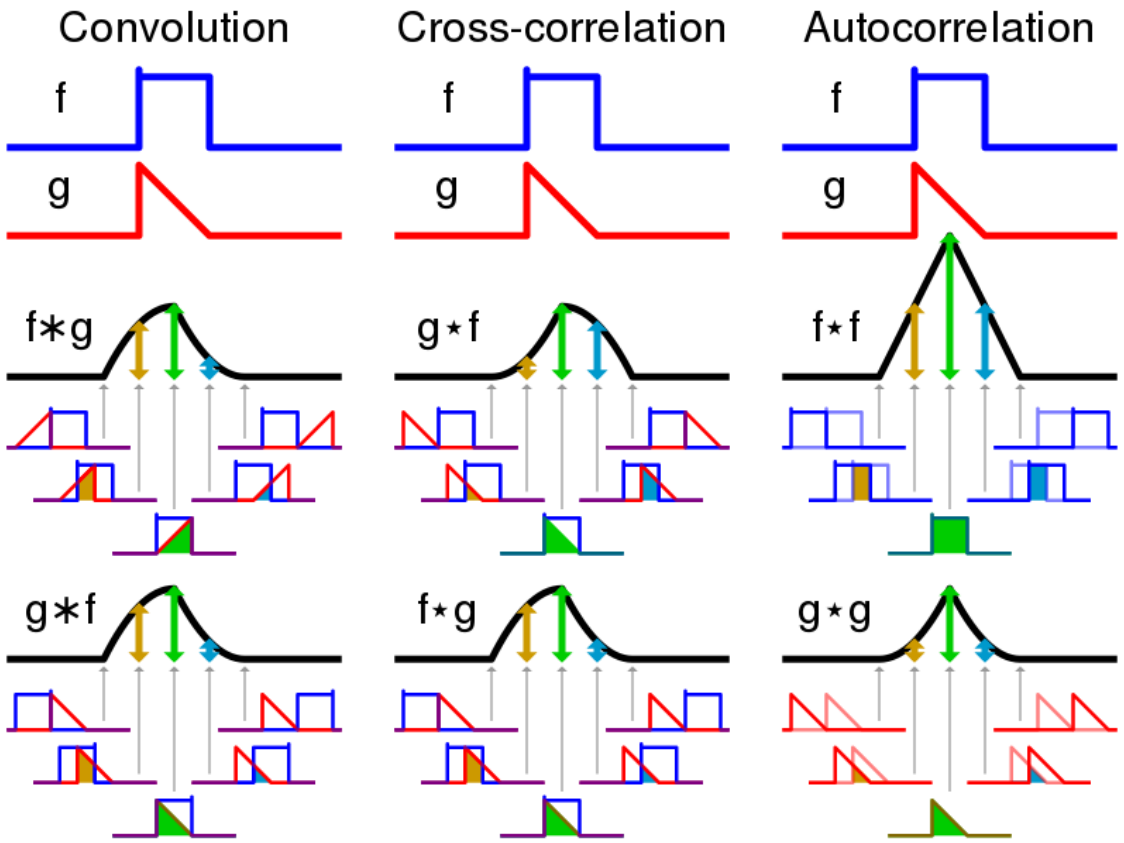
$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

Realization in network: RF and weight tying

- Receptive field: no full meshing inbetween layers
  - In case of example at each step of sliding:
    - consider only 3 x 3 submatrix of input matrix with (formally other weights are set to zero)
  - The weights for all the 3x3 submatrices of the input matrix get the same (!) weight (given by the convolutional filter/kernel)
  - Considering all timesteps instantaneously gives you a single layer (restructured as matrix „Convoluted Feature“ with smaller dimension)
- Considering many kernels gives you stacked layers



Terminology stems from signal processing



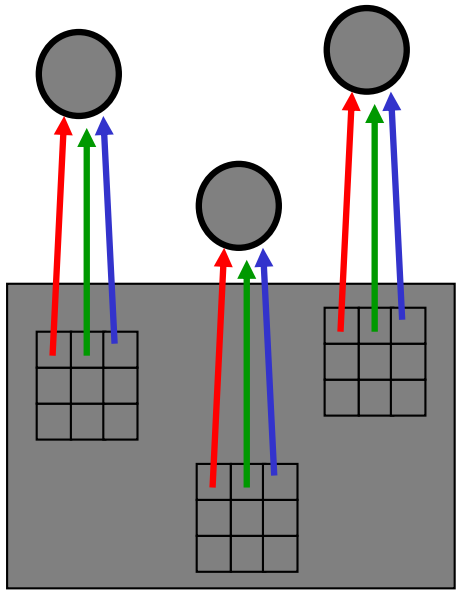
But: „Convolution“ in neural networks corresponds to cross-correlation

$$f \star g(x) = \int_{-\infty}^{\infty} \overline{f(t)} g(x+t) dx$$

The replicated feature approach
(currently the dominant approach for neural networks)

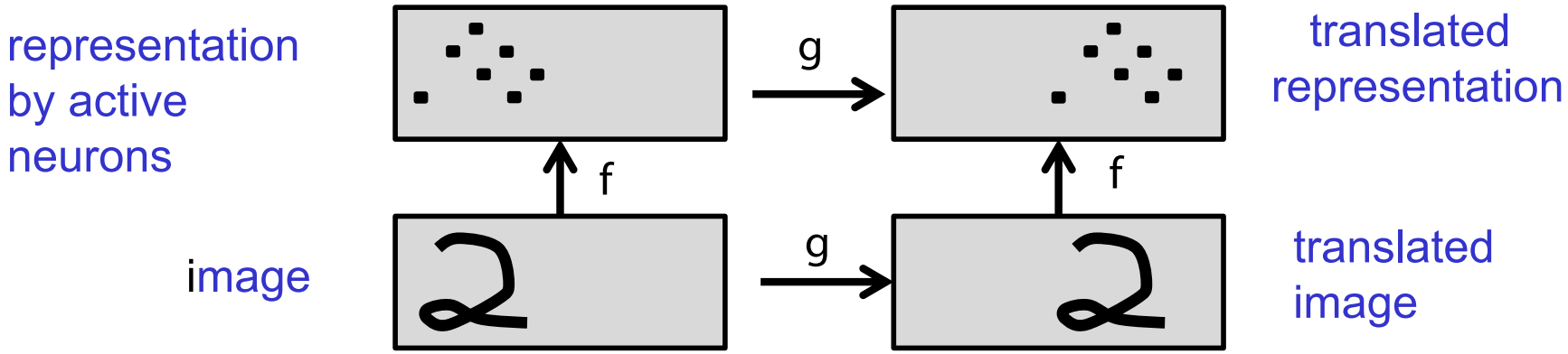
- Use many different copies of the same feature detector with different positions.
  - Could also replicate across scale and orientation (tricky and expensive)
  - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
  - Allows each patch of image to be represented in several ways.

The red connections all have the same weight.



Why replicating the feature detectors?

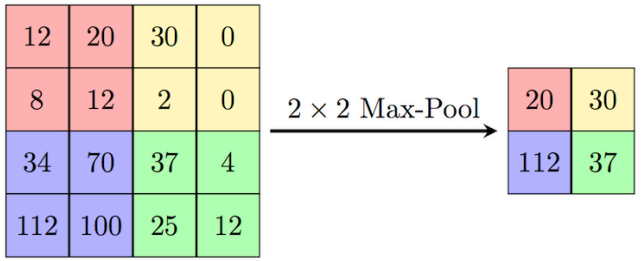
- Equivariant activities: The activities are translation equivariant.



- Invariant knowledge: If a feature is useful in some locations during training, detectors for that feature will be available in all locations during testing.
- Mathematically: f is equivariant w.r.t. g iff f(g(x)) = g(f(x)) (diagram above commutes)

Pooling and Striding

- Get a small amount of translational invariance at each level by averaging four neighboring replicated detectors to give a single output to the next level.
  - This reduces the number of inputs to the next layer of feature extraction, thus allowing us to have many more different feature maps.
  - Taking the maximum of the four works slightly better.
- Modern deep networks also use striding as dimension reduction: sliding kernel window with slide/step larger than 1

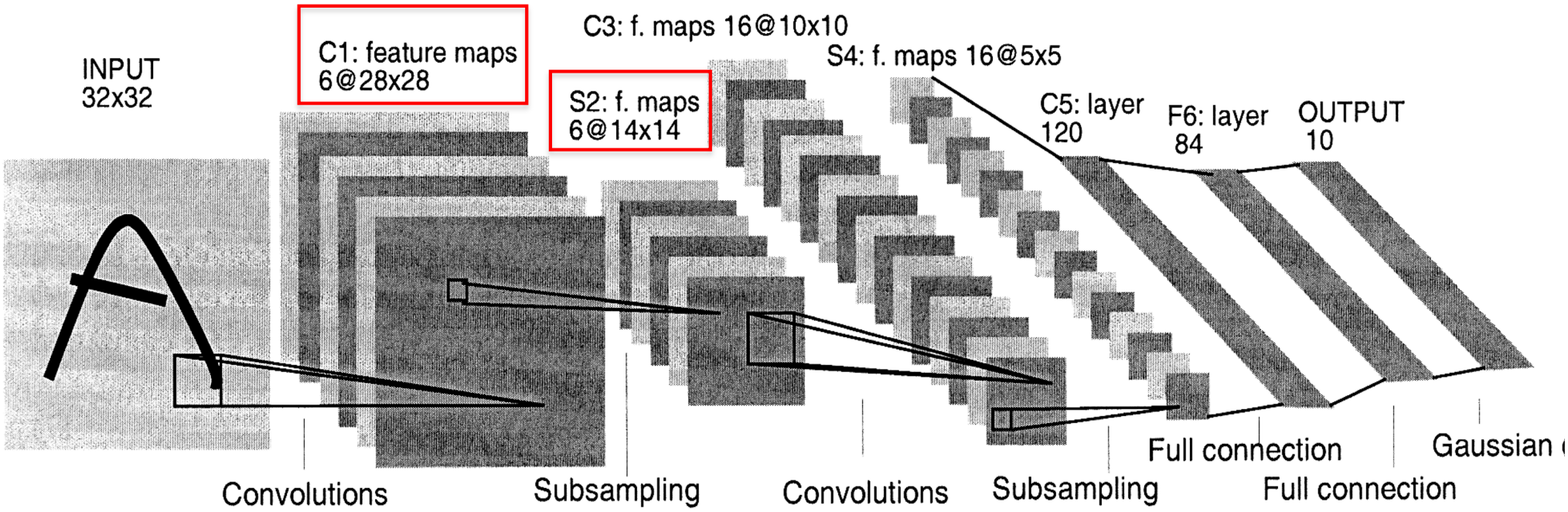


Maxpool with Stride 2

Le Net

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
  - Many hidden layers
  - Many maps of replicated units in each layer.
  - Pooling of the outputs of nearby replicated units.
  - A wide net that can cope with several characters at once even if they overlap.
  - A clever way of training a complete system, not just a recognizer.
- This net was used for reading ~10% of the checks in North America.
- Look the impressive demos of LENET at http://yann.lecun.com

The architecture of LeNet5



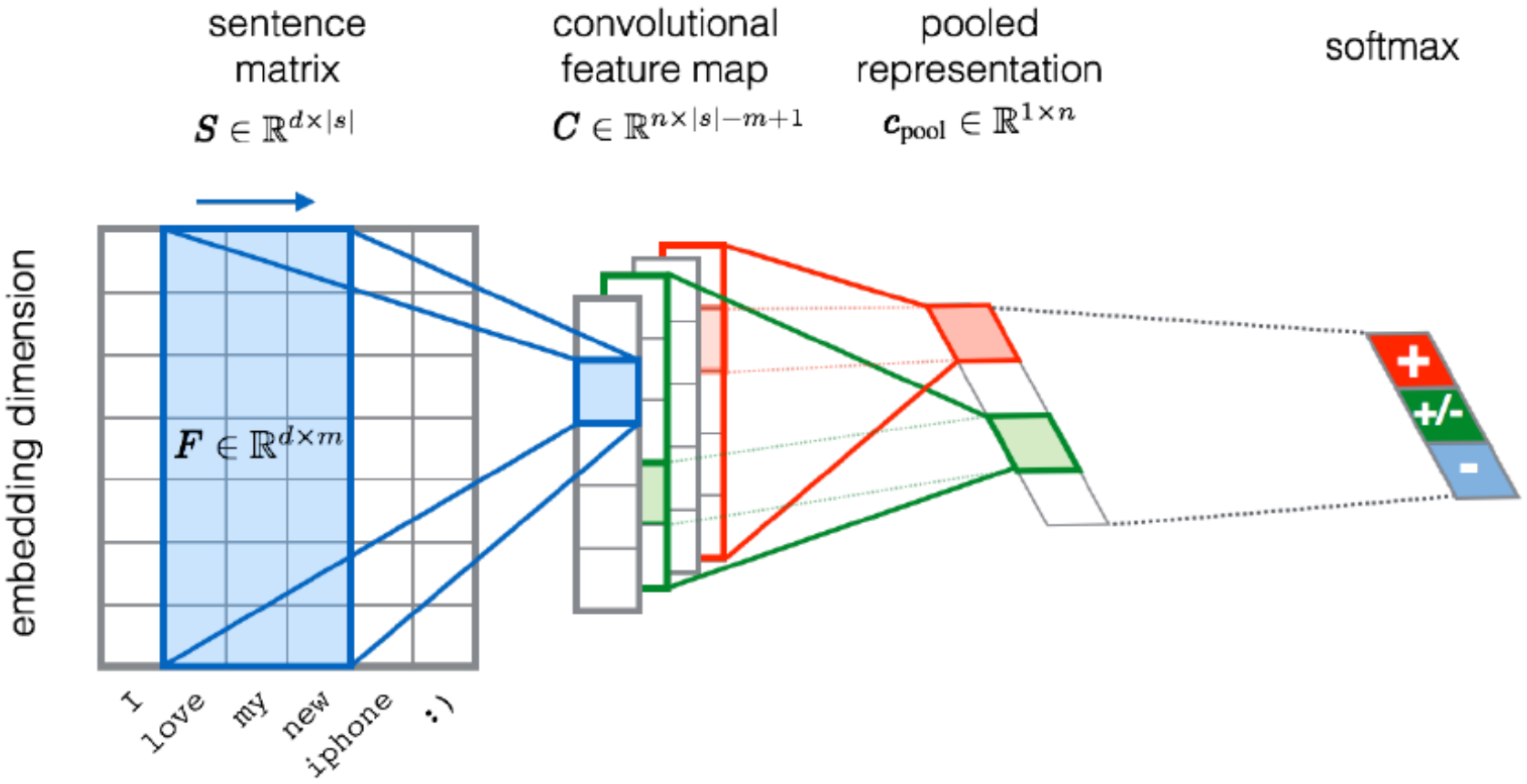
The 82 errors made by LeNet5

Notice that most of the errors are cases that people find quite easy.

The human error rate is probably 20 to 30 errors but nobody has had the patience to measure it.

There has been many further improvements with this application in mind
Ranzato 2008: use tricks to lower to 40
Ciresan et al. 2010: Enlarge dataset

CNN for text classification



Severyn, Aliaksei, and Alessandro Moschitti. "UNITN: Training Deep Convolutional Neural Network for Twitter Sentiment Classification." *SemEval@ NAACL-HLT*. 2015.

Uhhh, a lecture with a hoepfully useful

APPENDIX

Color Convention in this course

- Formulae, when occurring inline
- Newly introduced terminology and definitions
- Important results (observations, theorems) as well as emphasizing some aspects
- Examples are given with standard orange with possibly light orange frame
- Comments and notes
- Algorithms

Todays lecture is based on the following

- Jonathon Hare: Lectures 2,7,10 of course „COMP6248 Differentiable Programming (and some Deep Learning)“ http://comp6248.ecs.soton.ac.uk/
- Nielsen: Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/, chapter 6
- Geoofrey Hinton: Lecture 6a, Convolutional neural networks for hand-written digit recognition CSC2535: Advanced Machine Learning