Introduction to Deep Learning

Prof. Dr. David Rügamer

LMU Munich, MCML

2023

INTRODUCTION TO DEEP LEARNING

Introduction & MLPs

Convolutional Neural Networks

Recurrent Neural Networks

Practical Considerations for Training Neural Networks

INTRODUCTION TO DEEP LEARNING

Introduction & MLPs

- Introduction
- A Single Neuron
- Single Hidden Layer Networks
- Multiclass Classification
- Multilayer MLPs
- Basic Training

Convolutional Neural Networks

- Introduction
- Conv2D
- Properties of Convolutional Layers
- Components of Convolution Layers

Recurrent Neural Networks

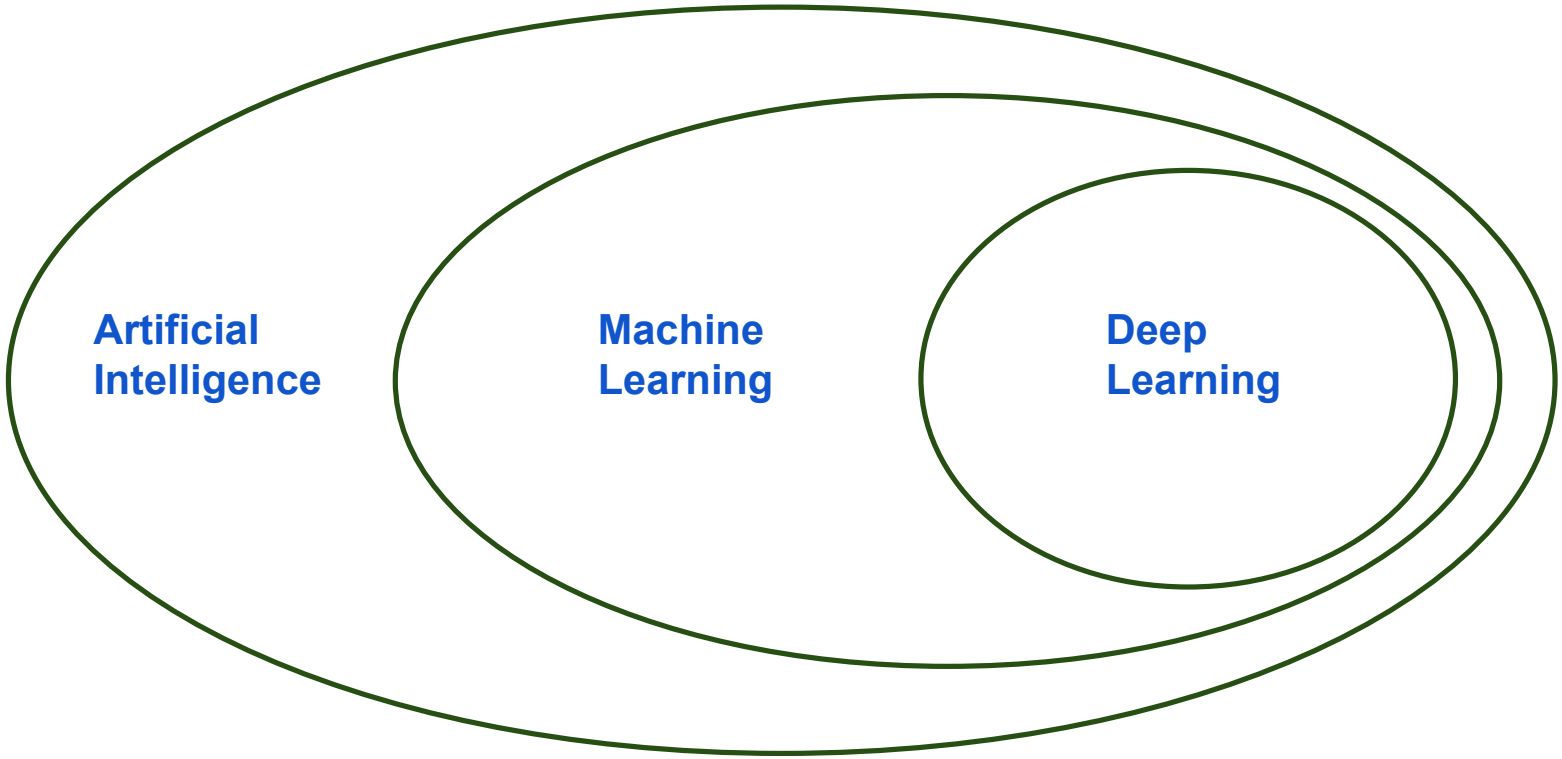
- Introduction
- Modern RNNs
- Encoder-Decoder Networks and Attention

Practical Considerations for Training Neural Networks

- Hardware and Software
- Regularization
- Advanced Training
- Activation Functions
- Residual Connections

Introduction

WHAT IS DEEP LEARNING



- Deep learning is a subfield of ML based on artificial neural networks.

DEEP LEARNING AND NEURAL NETWORKS

- Deep learning itself is not *new*:
  - Neural networks have been around since the 70s.
  - *Deep* neural networks, i.e., networks with multiple hidden layers, are not much younger.

- Why everybody is talking about deep learning now:
  1. Specialized, powerful hardware allows training of huge neural networks to push the state-of-the-art on difficult problems.
  2. Large amount of data is available.
  3. Special network architectures for image/text data.
  4. Better optimization and regularization strategies.

IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio

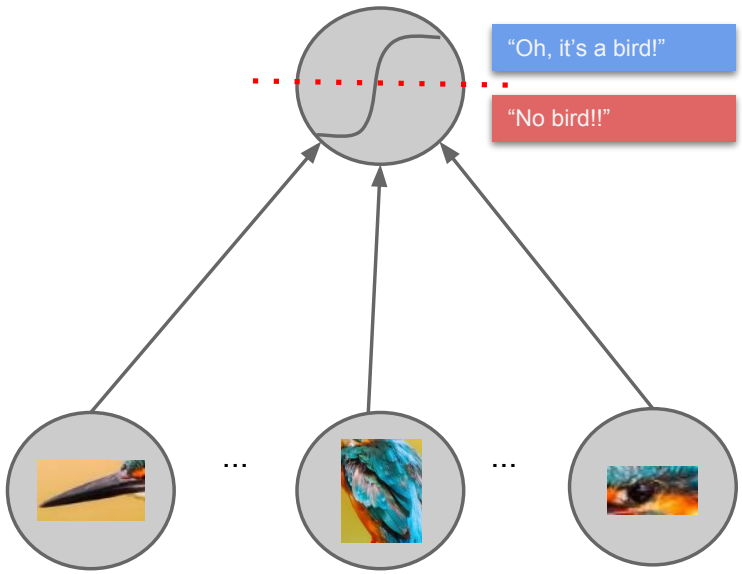


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio

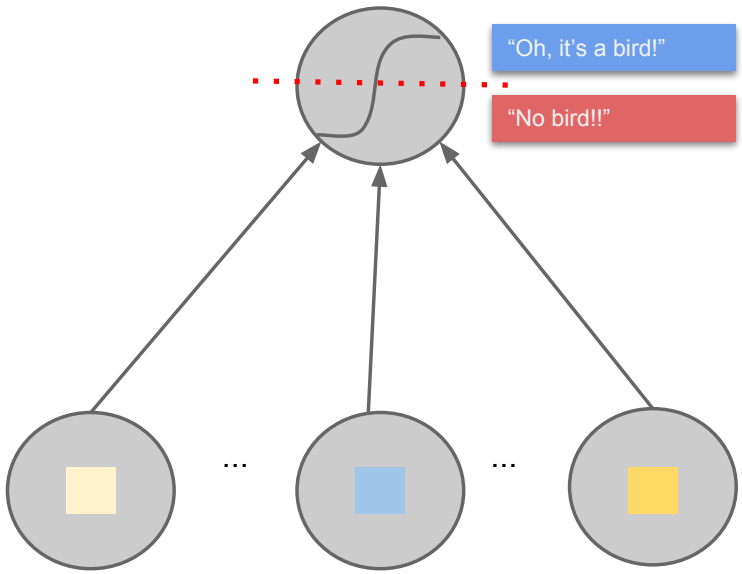
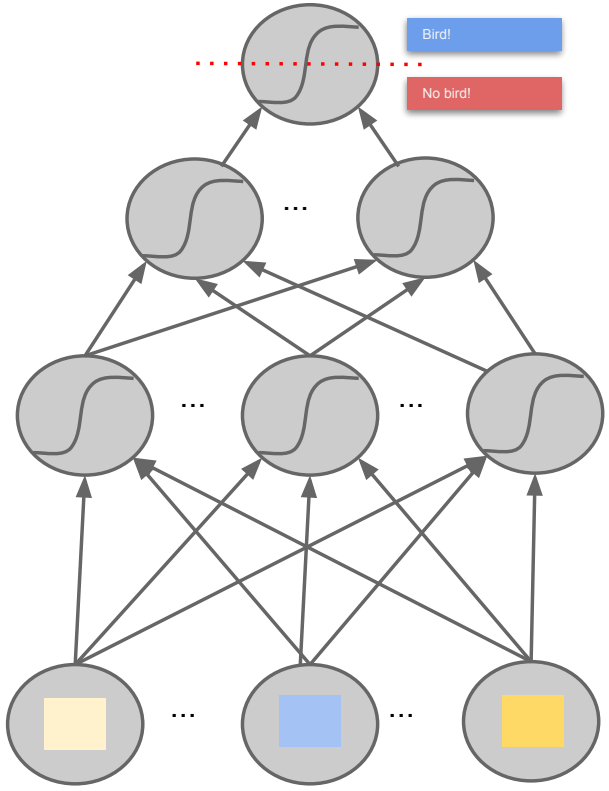


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

Y. Bengio



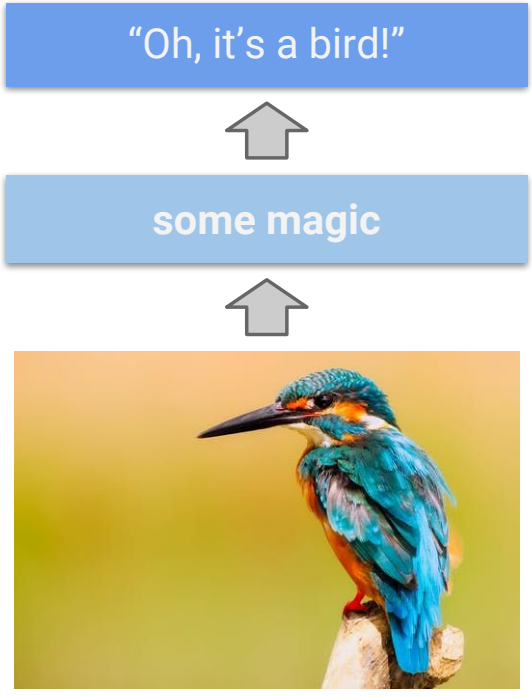
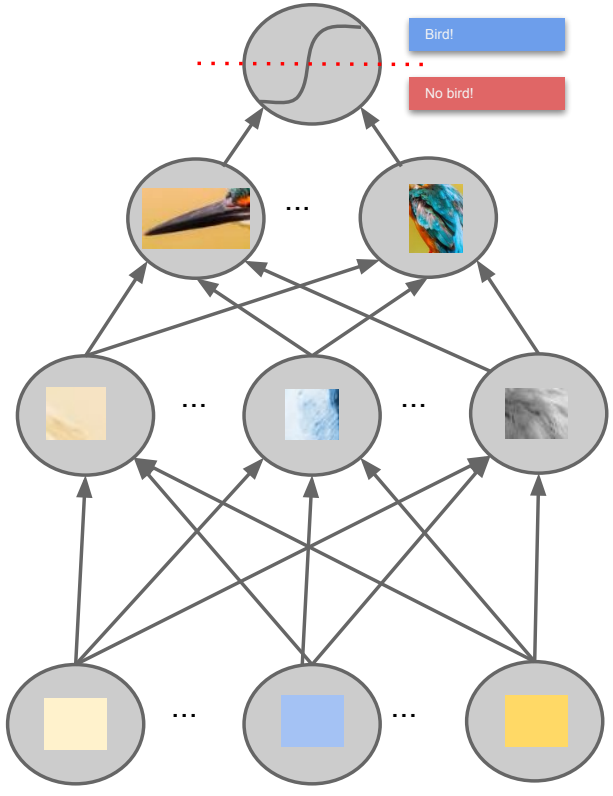
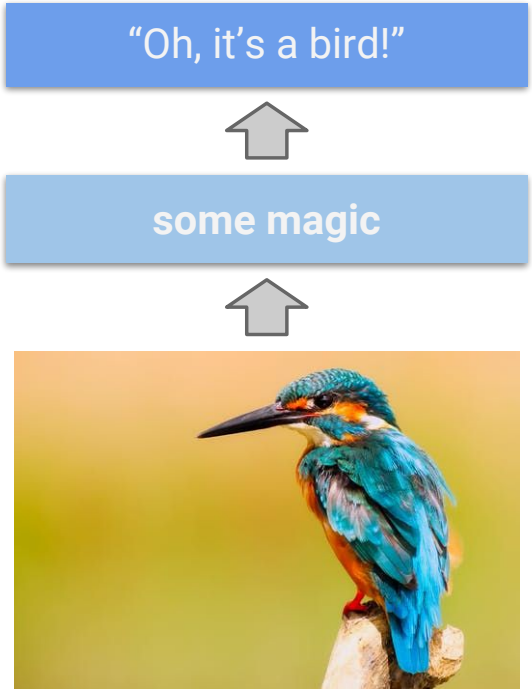


IMAGE CLASSIFICATION WITH NEURAL NETWORKS

"Machine learning algorithms, inspired by the brain, based on learning multiple levels of representation/abstraction."

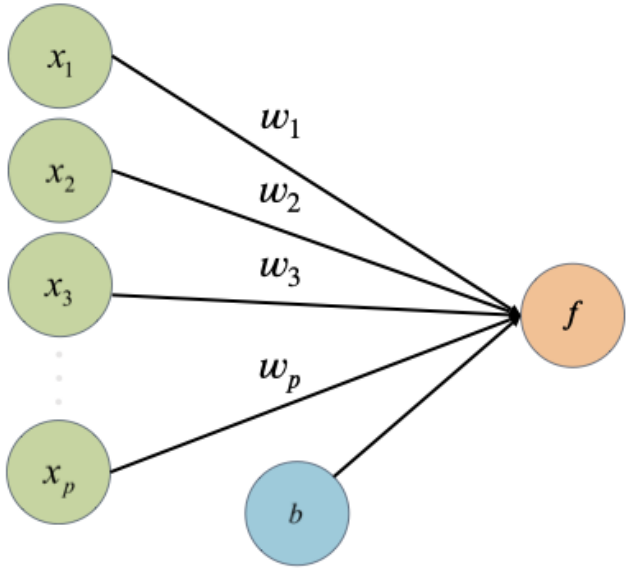
Y. Bengio





A Single Neuron

A SINGLE NEURON



Perceptron with **input features** $x_1, x_2, \ldots, x_p$, **weights** $w_1, w_2, \ldots, w_p$, **bias term** b, and **activation function** τ.

- The perceptron is a single artificial neuron and the basic computational unit of neural networks.
- It is a weighted sum of input values, transformed by τ:

$$f(x) = \tau(w_1 x_1 + \ldots + w_p x_p + b) = \tau(\mathbf{w}^T \mathbf{x} + b)$$

A SINGLE NEURON

Activation function τ: a single neuron represents different functions depending on the choice of activation function.

- The identity function gives us the simple **linear regression**:

$$f(x) = \tau(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

- The logistic function gives us the **logistic regression**:

$$f(x) = \tau(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})}$$

A SINGLE NEURON

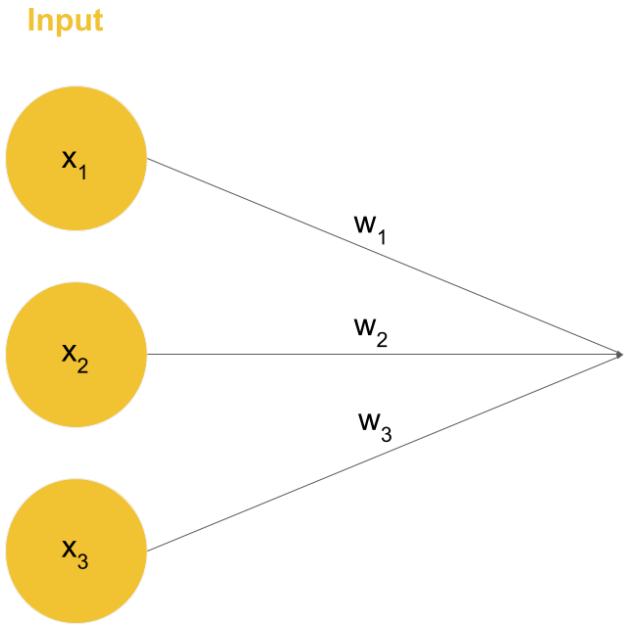
We consider a perceptron with 3-dimensional input, i.e.
$f(\mathbf{x}) = \tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$.

- Input features $\mathbf{x}$ are represented by nodes in the "input layer".

- In general, a p-dimensional input vector $\mathbf{x}$ will be represented by p nodes in the input layer.

A SINGLE NEURON

- Weights **w** are connected to edges from the input layer.



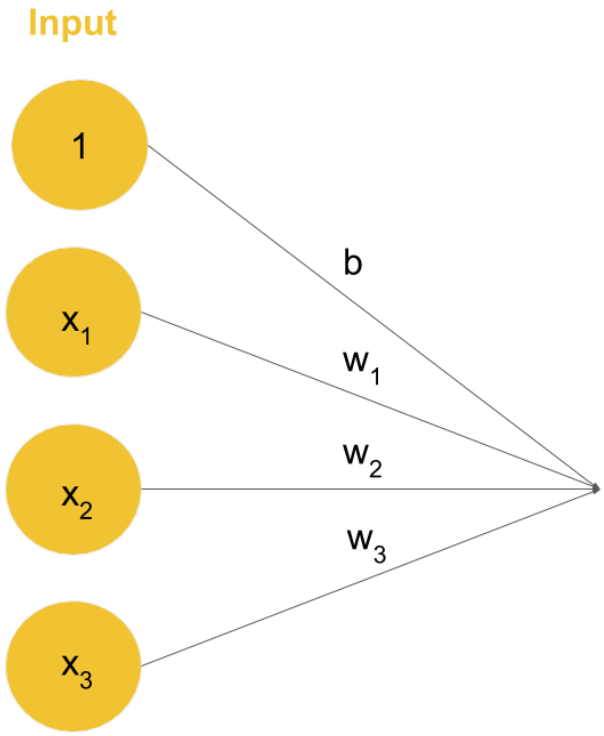
- The bias term *b* is implicit here. It is often not visualized as a separate node.

A SINGLE NEURON

For an explicit graphical representation, we do a simple trick:

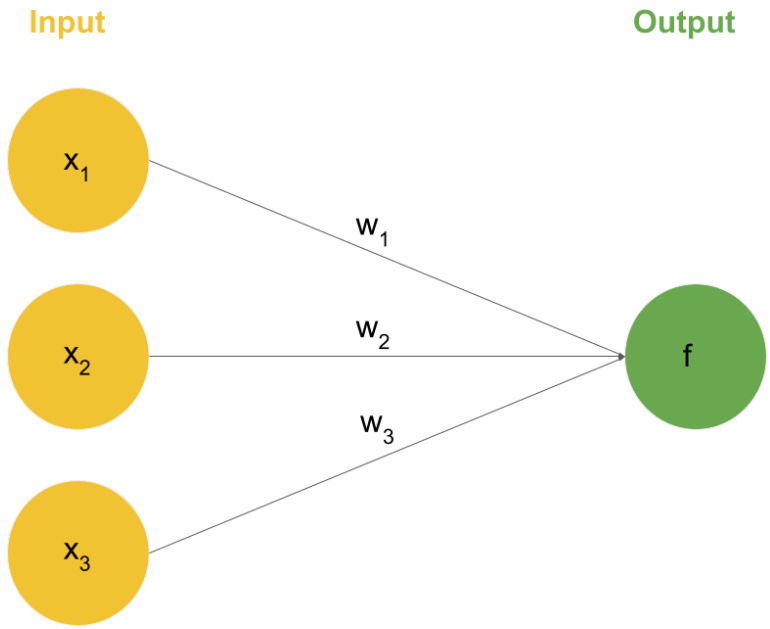
- Add a constant feature to the inputs $\tilde{\mathbf{x}} = (1, x_1, ..., x_p)^T$
- and absorb the bias into the weight vector $\tilde{\boldsymbol{w}} = (b, w_1, ..., w_p)$.

The graphical representation is then:



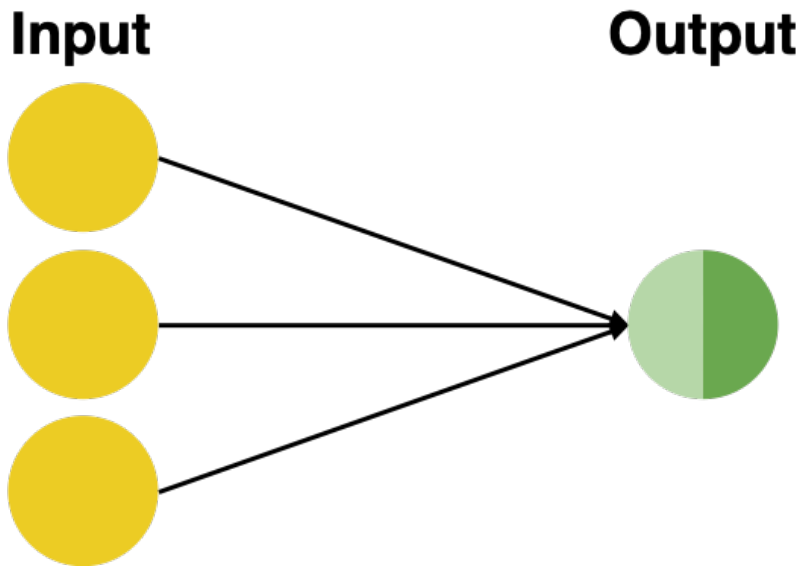
A SINGLE NEURON

- The computation $\tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$ is represented by the neuron in the "output layer".



A SINGLE NEURON

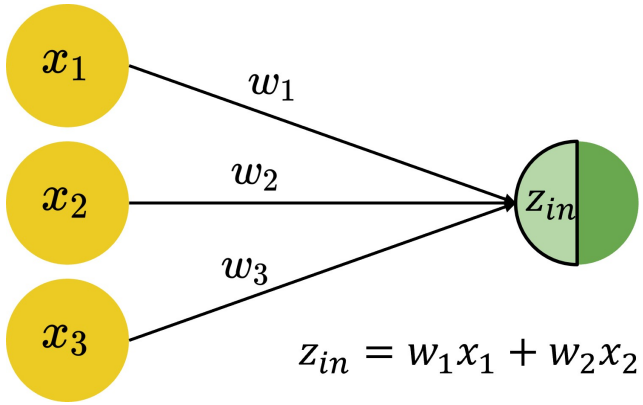
- You can picture the input vector being "fed" to neurons on the left followed by a sequence of computations performed from left to right. This is called a **forward pass**.



A SINGLE NEURON

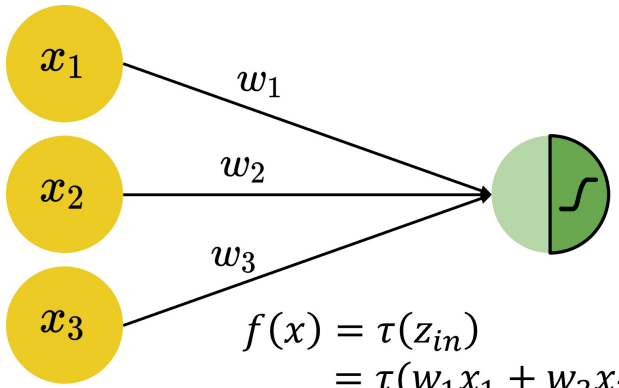
A neuron performs a 2-step computation:

1. **Affine Transformation:** weighted sum of inputs plus bias.



$$z_{in} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

2. **Non-linear Activation:** a non-linear transformation applied to the weighted sum.



$$f(x) = \tau(z_{in})$$
$$= \tau(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$$

A SINGLE NEURON: HYPOTHESIS SPACE

- The hypothesis space that is formed by single neuron is

$$\mathcal{H} = \left\{ f : \mathbb{R}^p \to \mathbb{R} \;\middle|\; f(\mathbf{x}) = \tau\left(\sum_{j=1}^{p} w_j x_j + b\right), \mathbf{w} \in \mathbb{R}^p, b \in \mathbb{R} \right\}.$$

- If τ is the logistic sigmoid or identity function, $\mathcal{H}$ corresponds to the hpothesis space of logistic or linear regression, respectively.

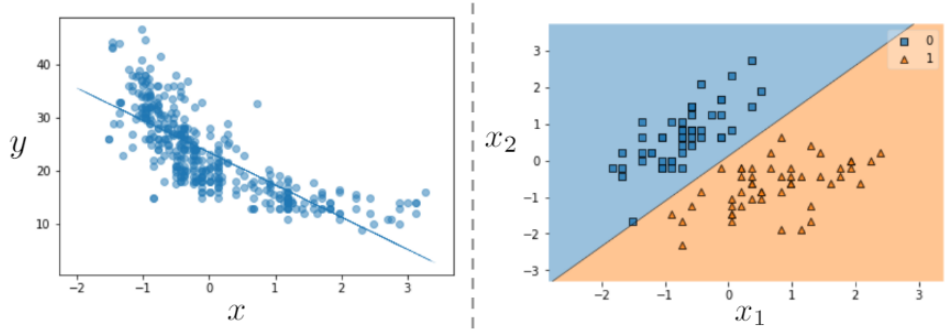


Figure: *Left*: A regression line learned by a single neuron. *Right*: A decision-boundary learned by a single neuron in a binary classification task.

A SINGLE NEURON: OPTIMIZATION

- To optimize this model, we minimize the empirical risk

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right),$$

  where $L(y, f(\mathbf{x}))$ is a loss function. It compares the network's predictions $f(\mathbf{x})$ to the ground truth y.
- For regression, we typically use the L2 loss (rarely L1):

$$L(y, f(\mathbf{x})) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- For binary classification, we typically apply the cross entropy loss (also known as Bernoulli loss):

$$L(y, f(\mathbf{x})) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

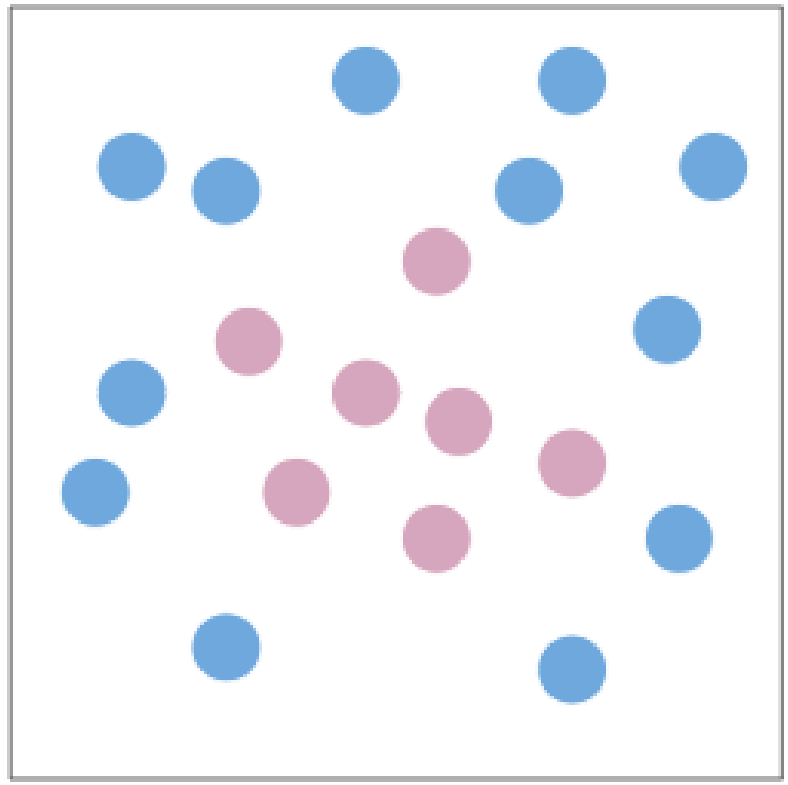
A SINGLE NEURON: OPTIMIZATION

- For a single neuron and both choices of τ the loss function is convex.
- The global optimum can be found with an iterative algorithm like gradient descent.
- A single neuron with logistic sigmoid function trained with the Bernoulli loss yields the same result as logistic regression when trained until convergence.
- Note: In the case of regression and the L2-loss, the solution can also be found analytically using the “normal equations”. However, in other cases a closed-form solution is usually not available.

Single Hidden Layer Networks

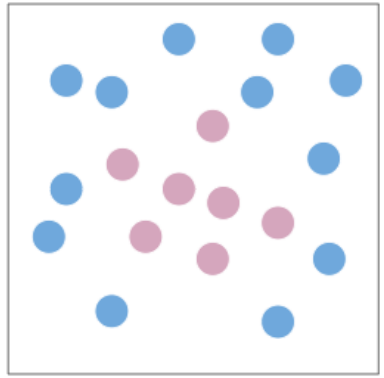
MOTIVATION

Can a single neuron perform binary classification of these points?



MOTIVATION

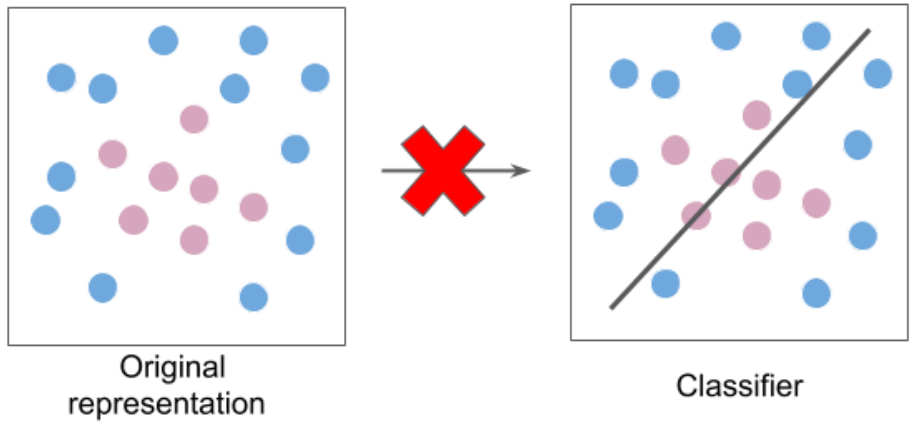
- As a single neuron is restricted to learning only linear decision boundaries, its performance on the following task is quite poor:



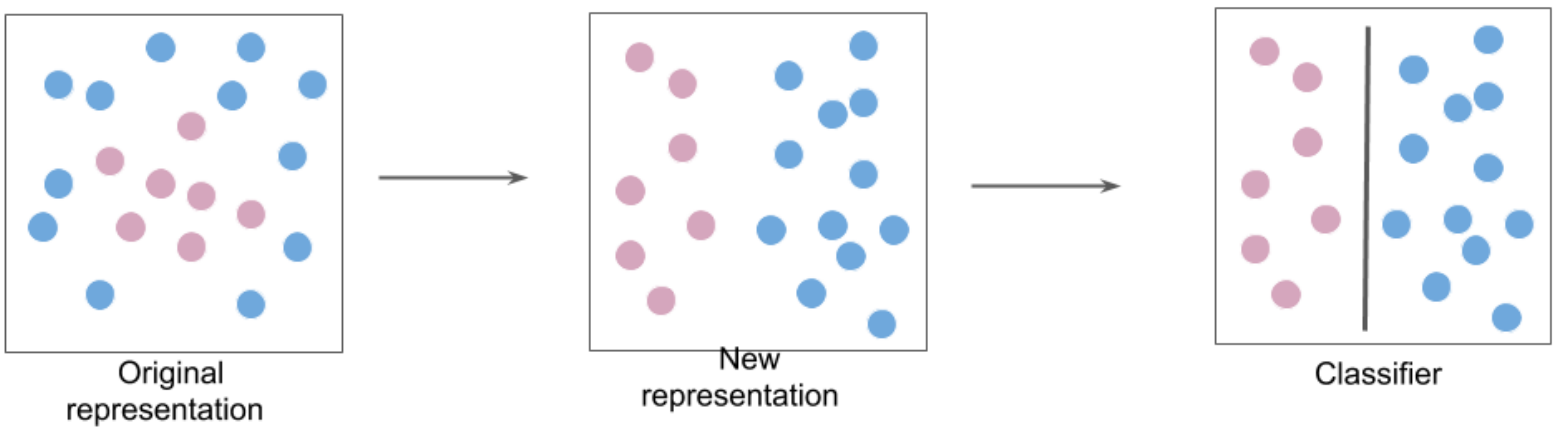
- However, the neuron can easily separate the classes if the original features are transformed (e.g., from Cartesian to polar coordinates):

MOTIVATION

- Instead of classifying the data in the original representation,

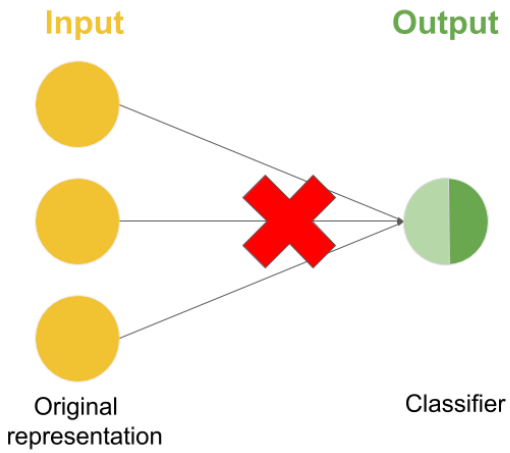


- we classify it in a new feature space.

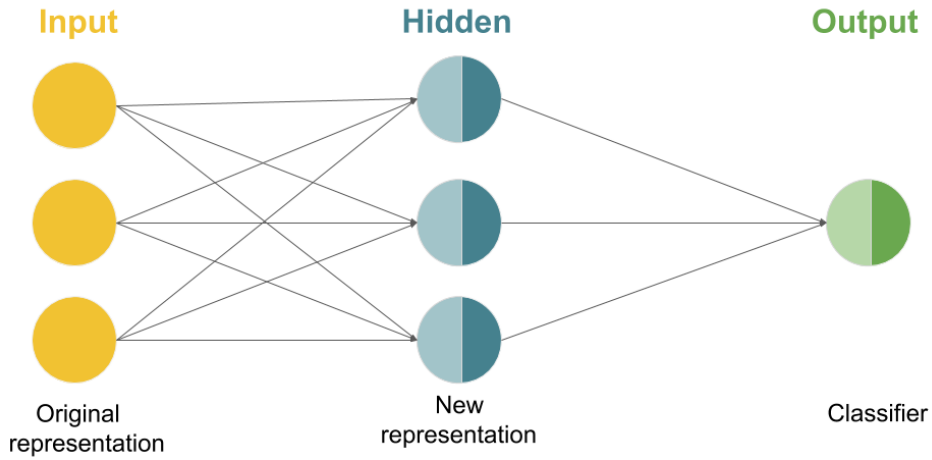


MOTIVATION

- Analogously, instead of a single neuron,



- we use more complex networks.



REPRESENTATION LEARNING

- It is *very* critical to feed a classifier the “right” features in order for it to perform well.

- Before deep learning took off, features for tasks like machine vision and speech recognition were “hand-designed” by domain experts. This step of the machine learning pipeline is called **feature engineering**.

- DL automates feature engineering. This is called **representation learning**.

SINGLE HIDDEN LAYER NETWORKS

Single neurons perform a 2-step computation:

1. **Affine Transformation:** a weighted sum of inputs plus bias.
2. **Activation:** a non-linear transformation on the weighted sum.

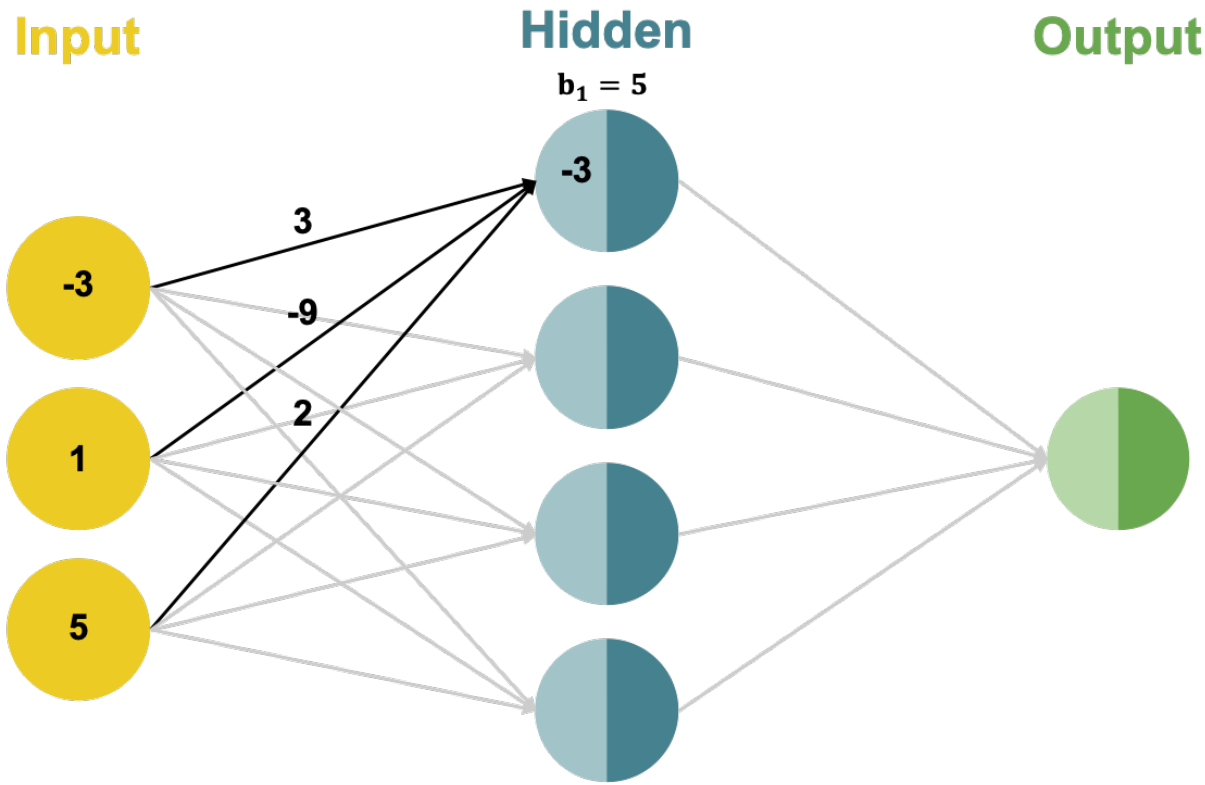
Single hidden layer networks consist of two layers (without input layer):

1. **Hidden Layer:** having a set of neurons.
2. **Output Layer:** having one or more output neurons.

- Multiple inputs are simultaneously fed to the network.
- Each neuron in the hidden layer performs a 2-step computation.
- The final output of the network is then calculated by another 2-step computation performed by the neuron in the output layer.

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

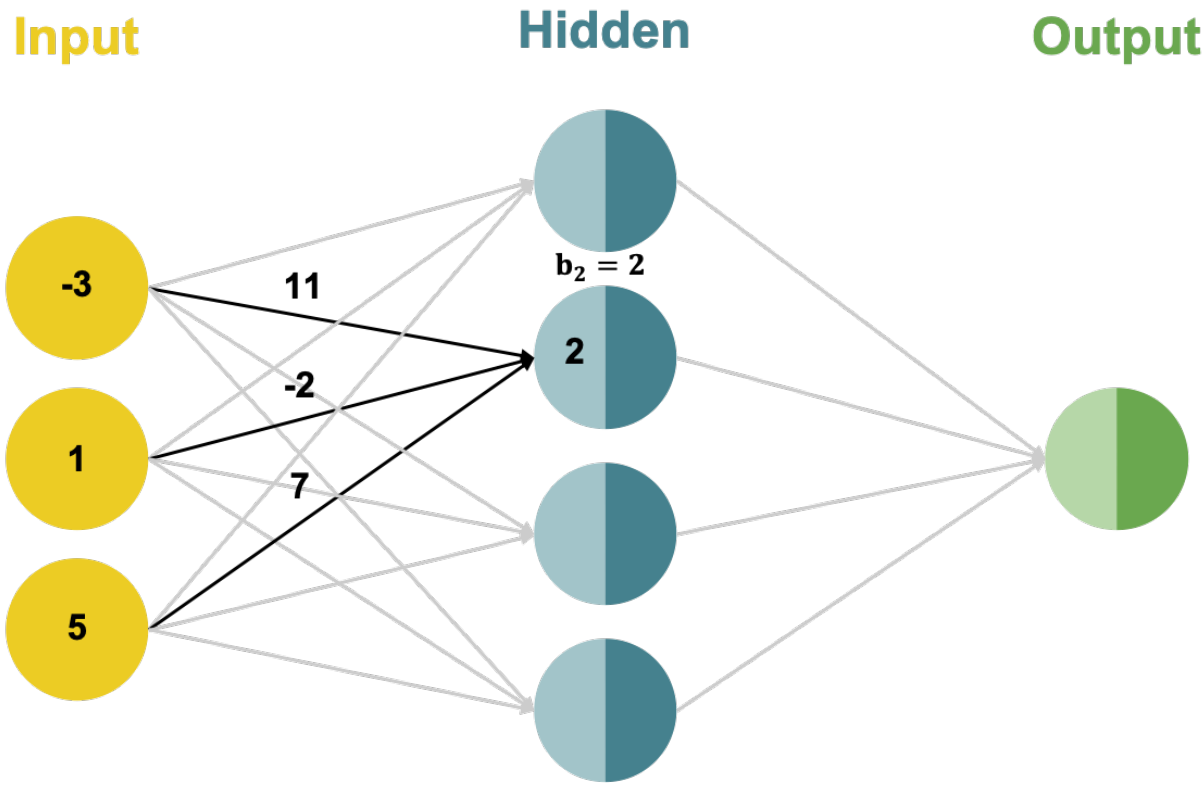
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(1)} = w_{11}x^{(1)} + w_{21}x^{(2)} + w_{31}x^{(3)} + b_1$$
$$z_{\text{in}}^{(1)} = 3 * (-3) + (-9) * 1 + 2 * 5 + 5 = -3$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

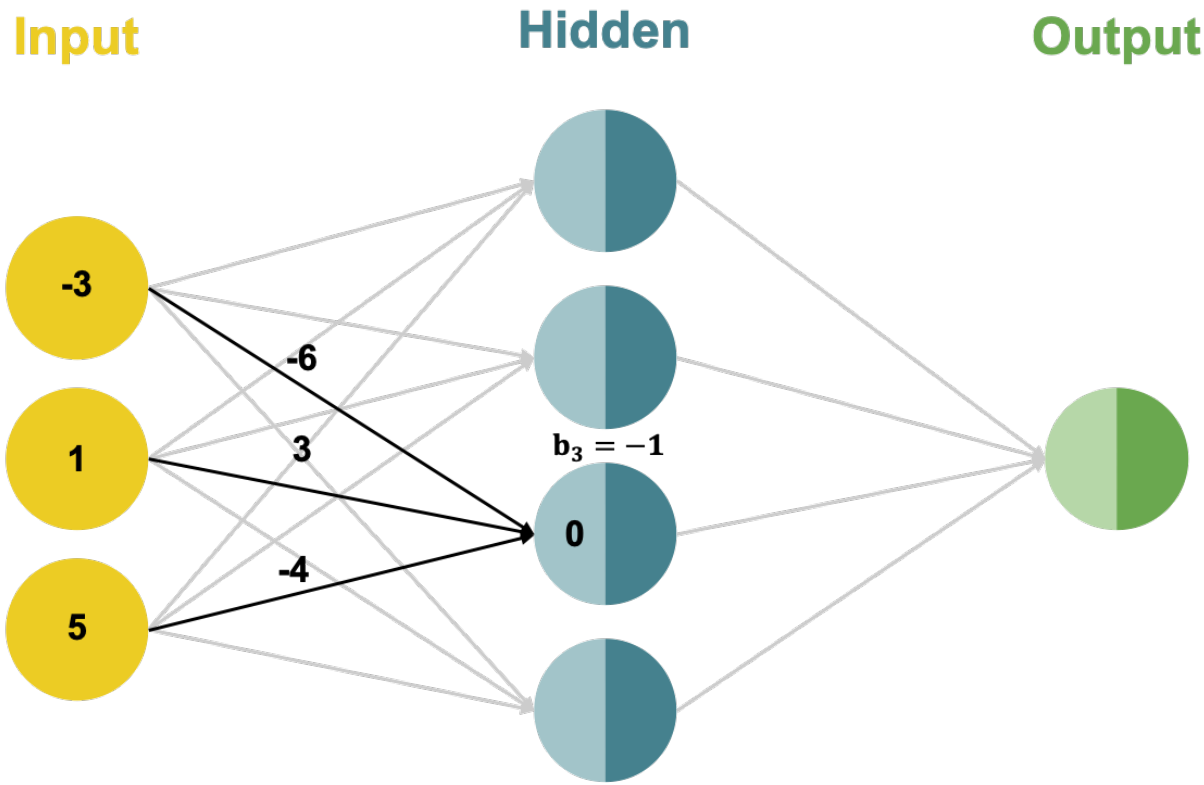
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(2)} = w_{12}x^{(1)} + w_{22}x^{(2)} + w_{32}x^{(3)} + b_2$$
$$z_{\text{in}}^{(2)} = 11 * (-3) + (-2) * 1 + 7 * 5 + 2 = 2$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

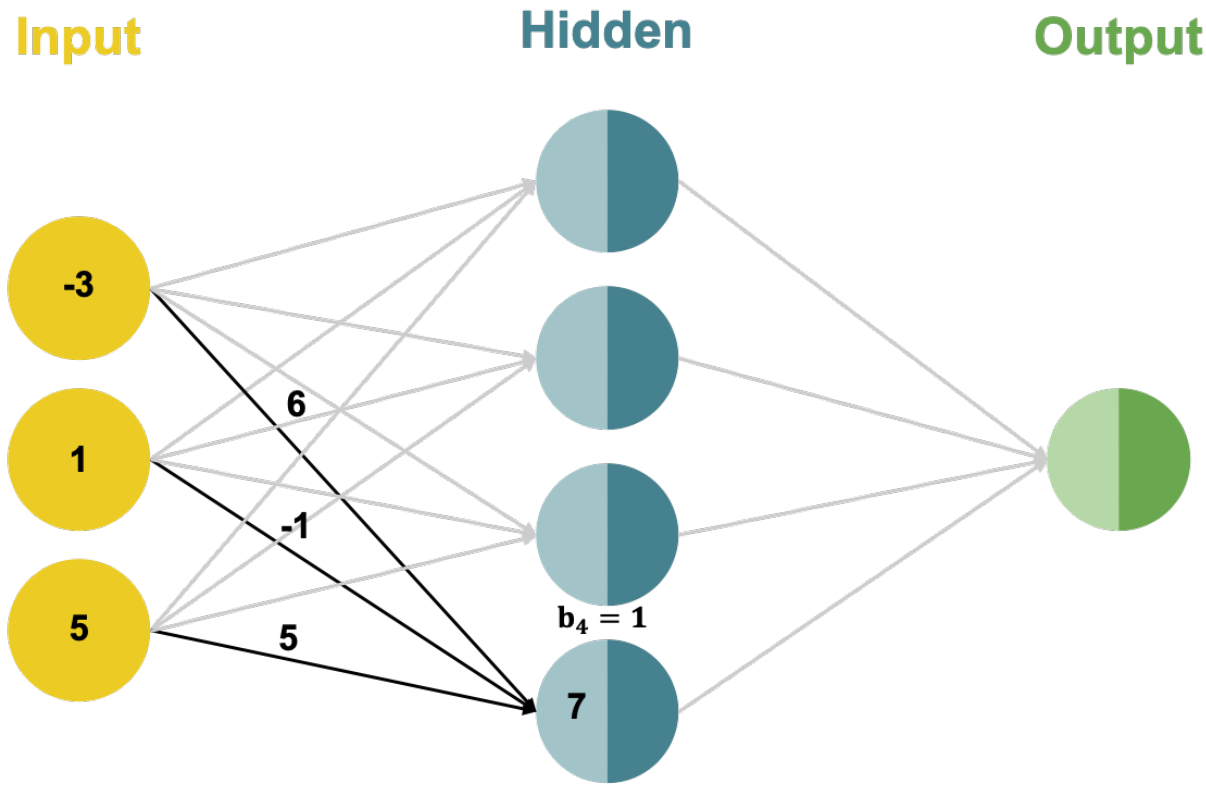
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(3)} = w_{13}x^{(1)} + w_{23}x^{(2)} + w_{33}x^{(3)} + b_3$$
$$z_{\text{in}}^{(3)} = (-6) * (-3) + 3 * 1 + (-4) * 5 - 1 = 0$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

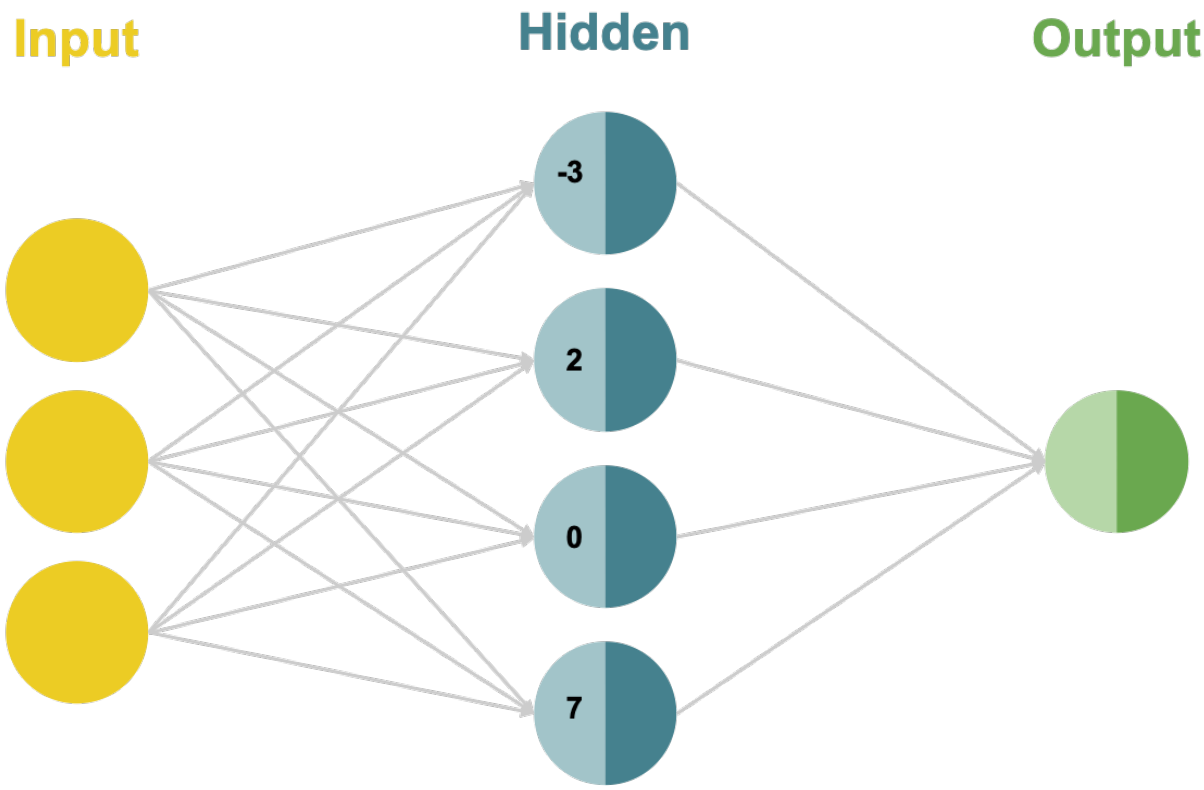
Each neuron in the hidden layer performs an **affine transformation** on the inputs:



$$z_{\text{in}}^{(4)} = w_{14}x^{(1)} + w_{24}x^{(2)} + w_{34}x^{(3)} + b_4$$
$$z_{\text{in}}^{(4)} = 6 * (-3) + (-1) * 1 + 5 * 5 + 1 = 7$$

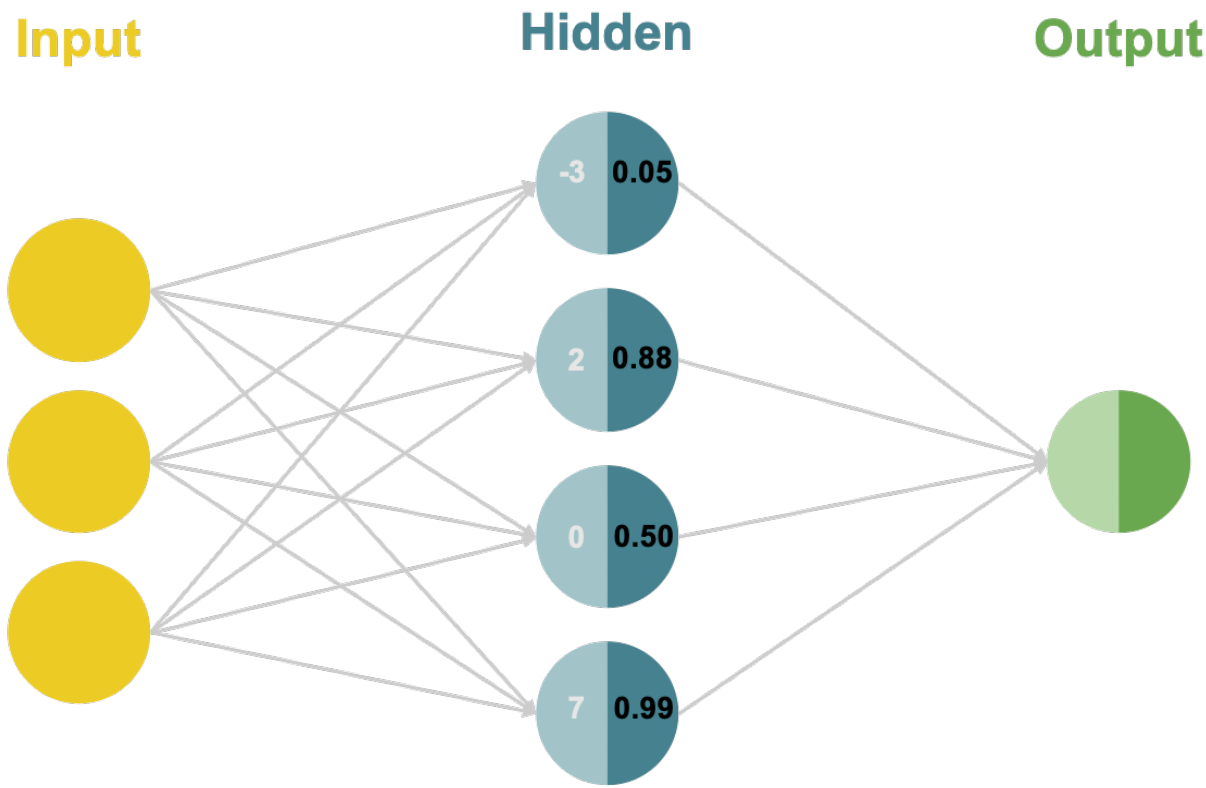
SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

Each neuron in the hidden layer performs an **affine transformation** on the inputs:



SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

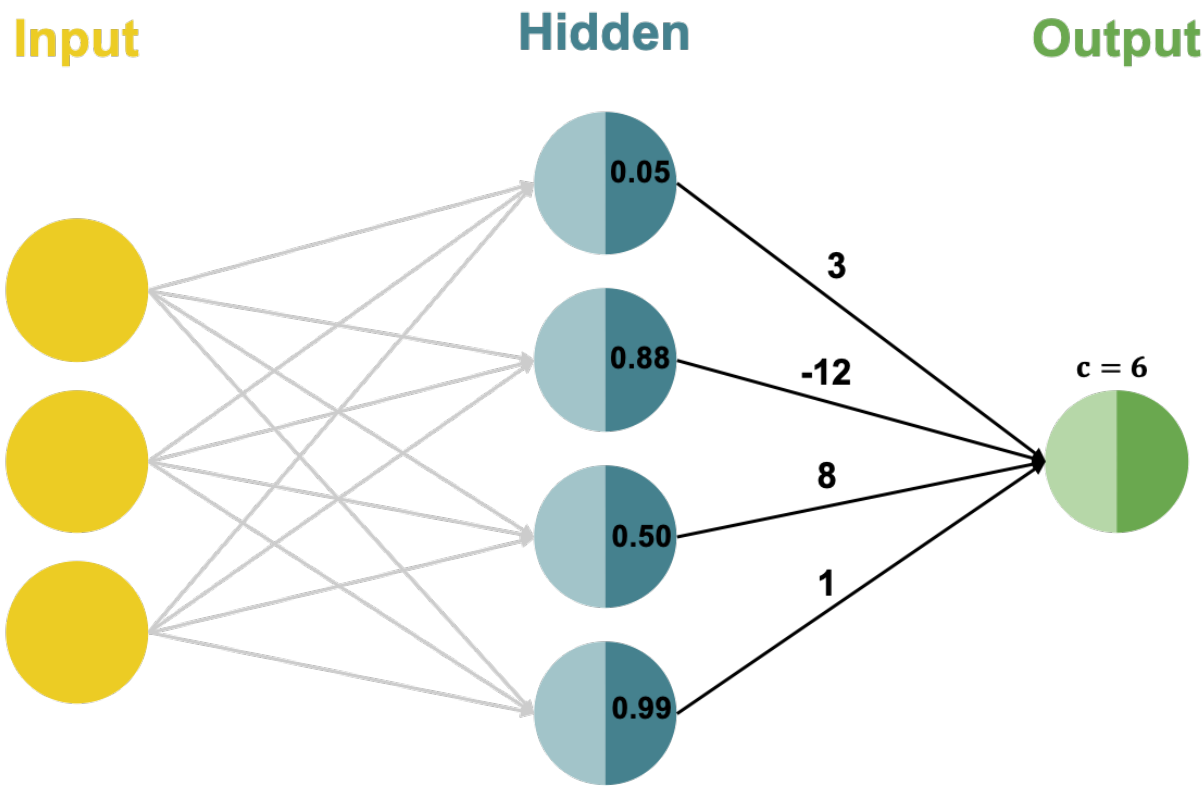
Each hidden neuron performs a non-linear **activation** transformation on the weight sum:



$$z_{\text{out}}^{(i)} = \sigma\left(z_{\text{in}}^{(i)}\right) = \frac{1}{1+e^{-z_{\text{in}}^{(i)}}}$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

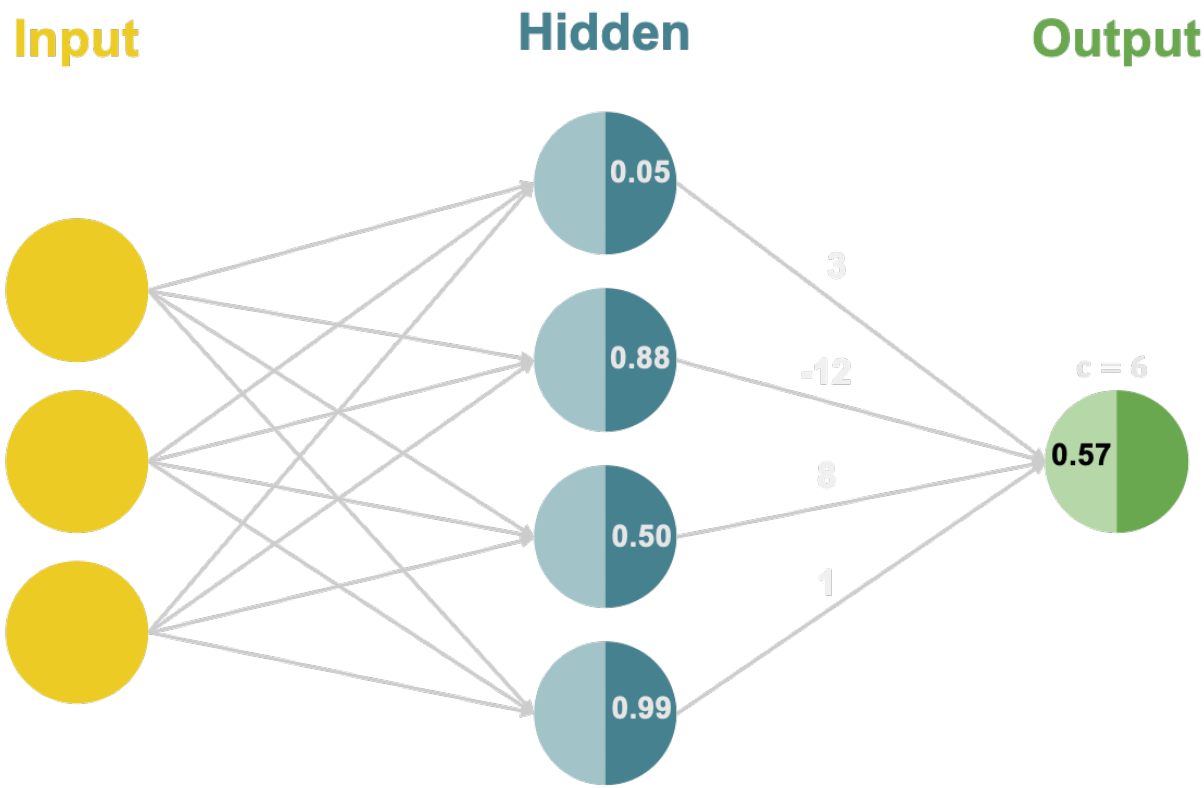
The output neuron performs an **affine transformation** on its inputs:



$$f_{\text{in}} = u_1 z_{\text{out}}^{(1)} + u_2 z_{\text{out}}^{(2)} + u_3 z_{\text{out}}^{(3)} + u_4 z_{\text{out}}^{(4)} + c$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

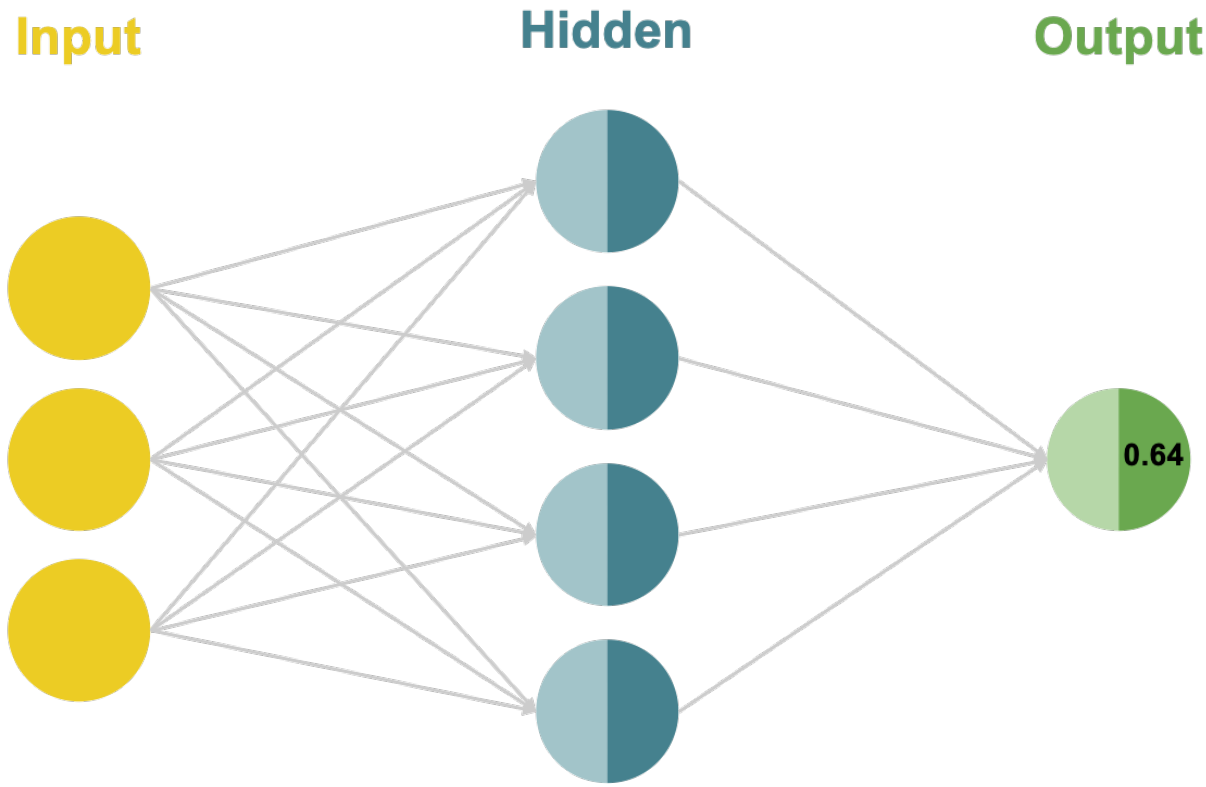
The output neuron performs an **affine transformation** on its inputs:



$$f_{\text{in}} = u_1 z_{\text{out}}^{(1)} + u_2 z_{\text{out}}^{(2)} + u_3 z_{\text{out}}^{(3)} + u_4 z_{\text{out}}^{(4)} + c$$
$$f_{\text{in}} = 3 * 0.05 + (-12) * 0.88 + 8 * 0.50 + 1 * 0.99 + 6 = 0.57$$

SINGLE HIDDEN LAYER NETWORKS: EXAMPLE

The output neuron performs a non-linear **activation** transformation on the weight sum:



$$f_{\text{out}} = \sigma(f_{\text{in}}) = \frac{1}{1+e^{-f_{\text{in}}}}$$
$$f_{\text{out}} = \frac{1}{1+e^{-0.57}} = 0.64$$

HIDDEN LAYER: ACTIVATION FUNCTION

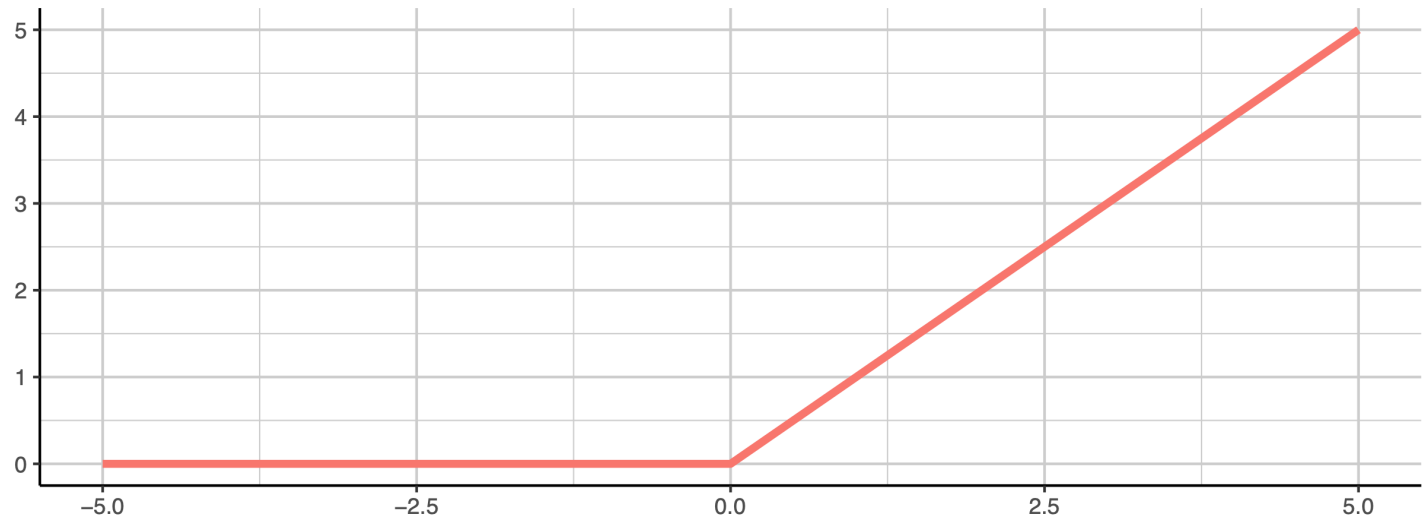
- If the hidden layer does not have a non-linear activation, the network can only learn linear decision boundaries.
- A lot of different activation functions exist.

HIDDEN LAYER: ACTIVATION FUNCTION

ReLU Activation:

- Currently the most popular choice is the ReLU (rectified linear unit):

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

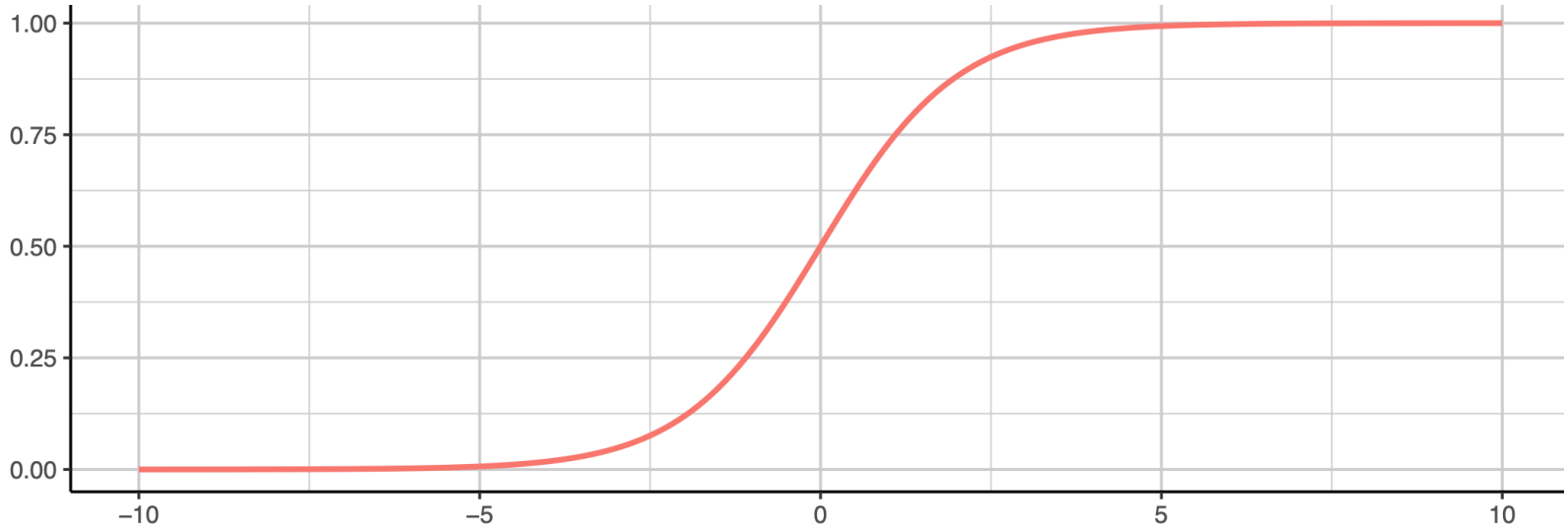


HIDDEN LAYER: ACTIVATION FUNCTION

Sigmoid Activation Function:

- The sigmoid function can be used even in the hidden layer:

$$\sigma(v) = \frac{1}{1 + \exp(-v)}$$



Multiclass Classification

MULTI-CLASS CLASSIFICATION

- We have only considered regression and binary classification problems so far.
- How can we get a neural network to perform multiclass classification?

MULTI-CLASS CLASSIFICATION

- The first step is to add additional neurons to the output layer.
- Each neuron in the layer will represent a specific class (number of neurons in the output layer = number of classes).

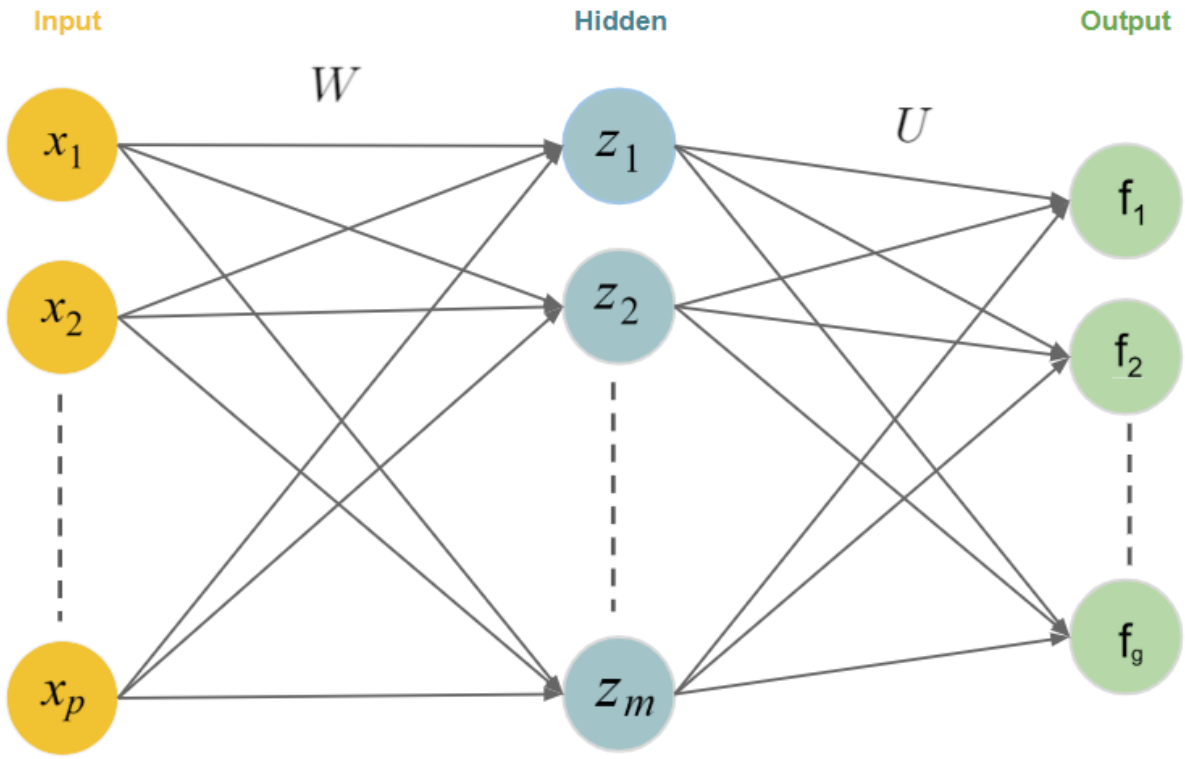


Figure: Structure of a single hidden layer, feed-forward neural network for g-class classification problems (bias term omitted).

MULTI-CLASS CLASSIFICATION

Notation:

- For g-class classification, g output units:

$$\mathbf{f} = (f_1, \ldots, f_g)$$

- m hidden neurons $z_1, \ldots, z_m$, with

$$z_j = \sigma(\mathbf{W}_j^T \mathbf{x}), \quad j = 1, \ldots, m.$$

- Compute linear combinations of derived features z:

$$f_{in,k} = \boldsymbol{U}_k^T \mathbf{z}, \quad \mathbf{z} = (z_1, \ldots, z_m)^T, \quad k = 1, \ldots, g$$

MULTI-CLASS CLASSIFICATION

- The second step is to apply a **softmax** activation function to the output layer.

- This gives us a probability distribution over g different possible classes:

$$f_{out,k} = \tau_k(f_{in,k}) = \frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})}$$

- This is the same transformation used in softmax regression!

- Derivative $\frac{\partial \tau(\mathbf{f}_{in})}{\partial \mathbf{f}_{in}} = \text{diag}(\tau(\mathbf{f}_{in})) - \tau(\mathbf{f}_{in})\tau(\mathbf{f}_{in})^T$

- It is a "smooth" approximation of the argmax operation, so $\tau((1, 1000, 2)^T) \approx (0, 1, 0)^T$ (picks out 2nd element!).

OPTIMIZATION: SOFTMAX LOSS

- The loss function for a softmax classifier is

$$L(y, f(\mathbf{x})) = -\sum_{k=1}^{g} [y = k] \log \left(\frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})} \right)$$

where $[y = k] = \begin{cases} 1 & \text{if } y = k \\ 0 & \text{otherwise} \end{cases}$.

- This is equivalent to the cross-entropy loss when the label vector $\boldsymbol{y}$ is one-hot coded (e.g. $\mathbf{y} = (0, 0, 1, 0)^T$).
- Optimization: Again, there is no analytic solution.

Multilayer MLPs

FEEDFORWARD NEURAL NETWORKS

- We will now extend the model class once again, such that we allow an arbitrary amount l of hidden layers.

- The general term for this model class is (multi-layer) **feedforward networks** (inputs are passed through the network from left to right, no feedback-loops are allowed)

FEEDFORWARD NEURAL NETWORKS

- We can characterize those models by the following chain structure:

$$f(\mathbf{x}) = \tau \circ \phi \circ \sigma^{(l)} \circ \phi^{(l)} \circ \sigma^{(l-1)} \circ \phi^{(l-1)} \circ \ldots \circ \sigma^{(1)} \circ \phi^{(1)}$$

  where $\sigma^{(i)}$ and $\phi^{(i)}$ are the activation function and the weighted sum of hidden layer i, respectively. τ and ϕ are the corresponding components of the output layer.

- Each hidden layer has:
  - an associated weight matrix $\mathbf{W}^{(i)}$, bias $\mathbf{b}^{(i)}$, and activations $\mathbf{z}^{(i)}$ for $i \in \{1 \ldots l\}$.
  - $\mathbf{z}^{(i)} = \sigma^{(i)}(\phi^{(i)}) = \sigma^{(i)}(\mathbf{W}^{(i)T}\mathbf{z}^{(i-1)} + \mathbf{b}^{(i)})$, where $\mathbf{z}^{(0)} = \mathbf{x}$.

- Again, without non-linear activations in the hidden layers, the network can only learn linear decision boundaries.

FEEDFORWARD NEURAL NETWORKS

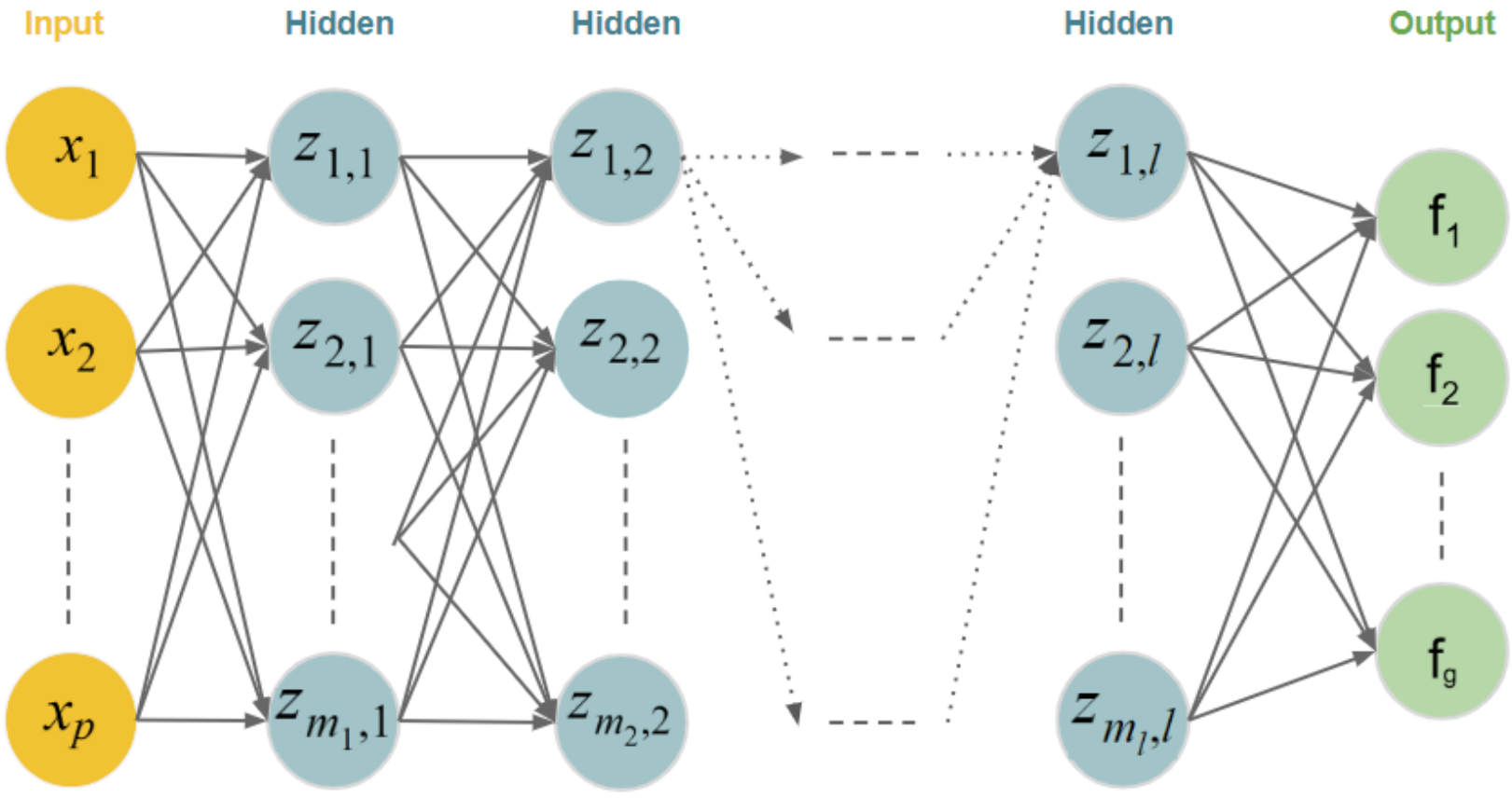


Figure: Structure of a deep neural network with *l* hidden layers (bias terms omitted).

WHY ADD MORE LAYERS?

- Multiple layers allow for the extraction of more and more abstract representations.
- Each layer in a feed-forward neural network adds its own degree of non-lnearity to the model.

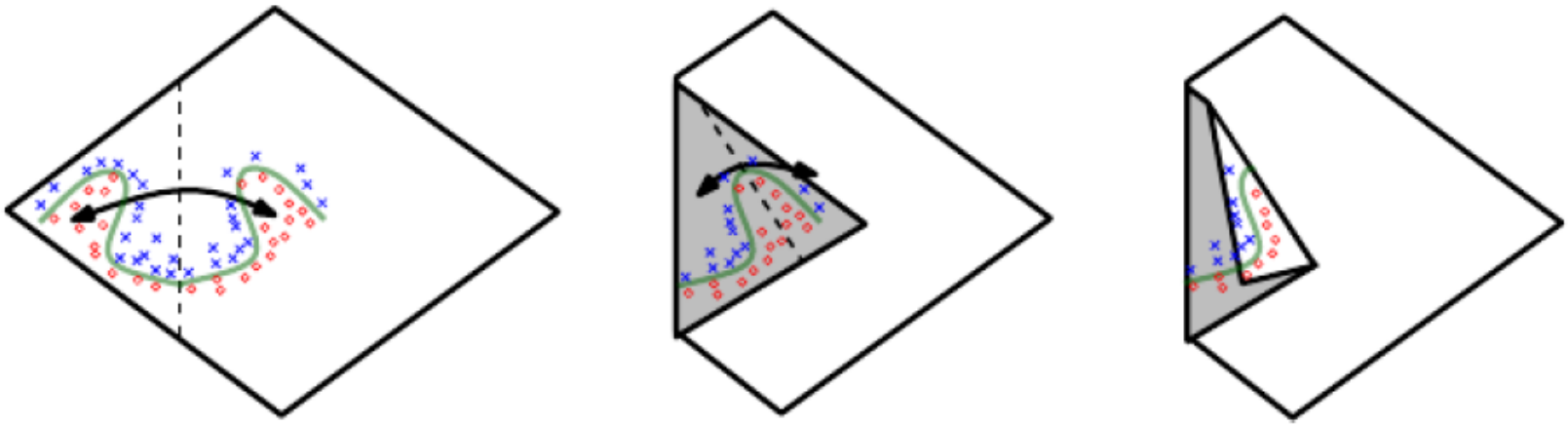


Figure: An intuitive, geometric explanation of the exponential advantage of deeper networks formally (Montúfar et al. (2014)).

Basic Training

TRAINING NEURAL NETWORKS

- In ML we use **empirical risk minimization** to minimize prediction losses over the training data

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right)$$

- In DL, $\boldsymbol{\theta}$ represents the weights (and biases) of the NN.
- Often, L2 in regression:

$$L\left(y, f(\mathbf{x})\right) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- or cross-entropy for binary classification

$$L\left(y, f(\mathbf{x})\right) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

- ERM can be implemented by **gradient descent**.

GRADIENT DESCENT

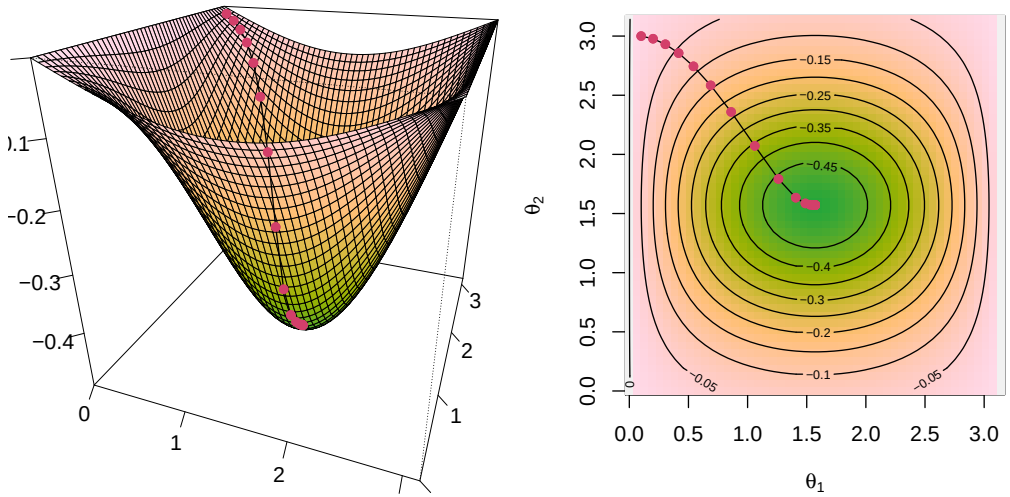
- Neg. risk gradient points in the direction of the **steepest descent**

$$-\mathbf{g} = -\nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -\left(\frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_1}, \ldots, \frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_d} \right)^\top$$

- "Standing" at a point $\boldsymbol{\theta}^{[t]}$, we locally improve by:

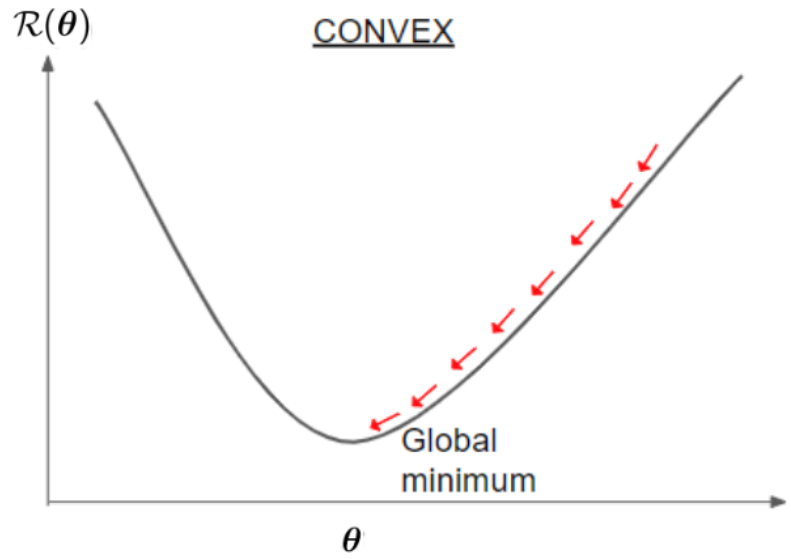
$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \mathbf{g},$$

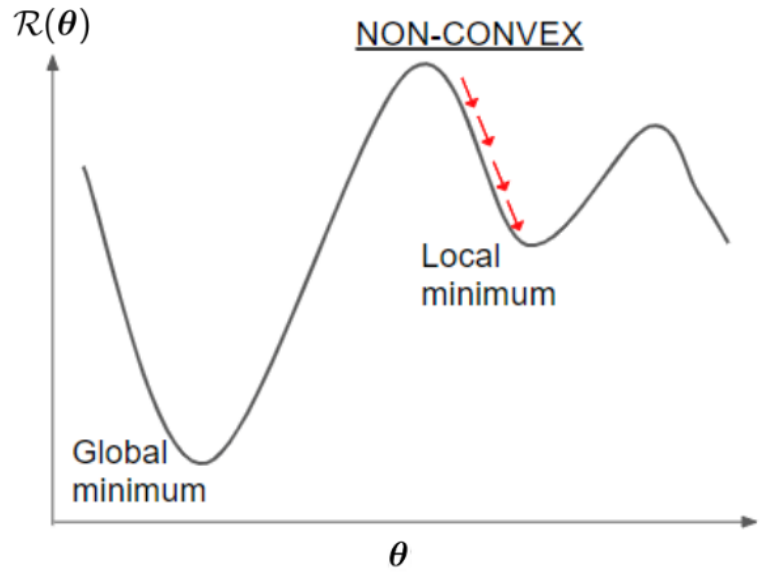
- α is called **step size** or **learning rate**.



GRADIENT DESCENT AND OPTIMALITY

- GD is a greedy algorithm: In every iteration, it makes locally optimal moves.
- If $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ is **convex** and **differentiable**, and its gradient is Lipschitz continuous, GD is guaranteed to converge to the global minimum (for small enough step-size).
- However, if $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ has multiple local optima and/or saddle points, GD might only converge to a stationary point (other than the global optimum), depending on the starting point.





GRADIENT DESCENT AND OPTIMALITY

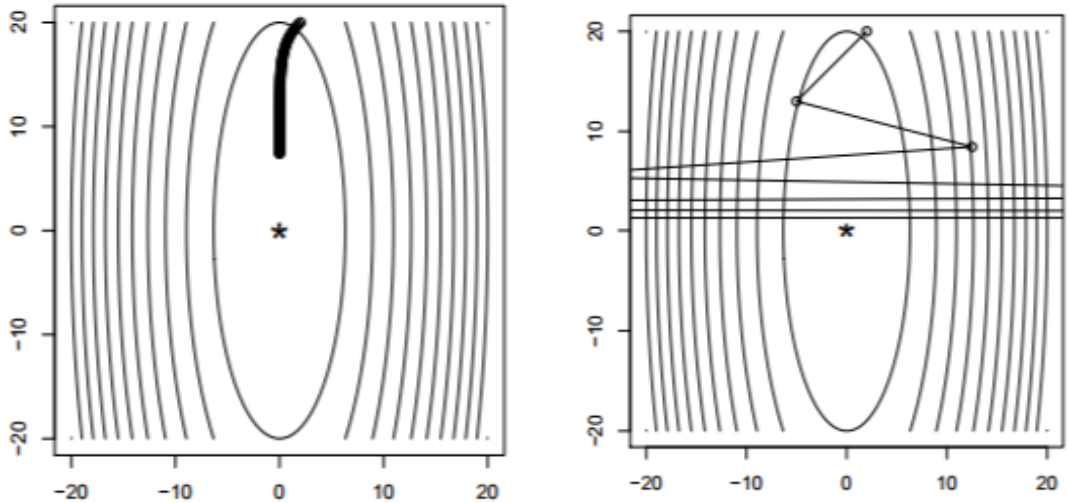
Note: It might not be that bad if we do not find the global optimum:

- We don't optimize the (theoretical) risk, but only an approximate version, i.e. the empirical risk.
- For very flexible models, aggressive optimization might overfitting.
- Early-stopping might even increase generalization performance.

LEARNING RATE

The step-size α plays a key role in the convergence of the algorithm.

If the step size is too small, the training process may converge **very** slowly (see left image). If the step size is too large, the process may not converge, because it **jumps** around the optimal point (see right image).

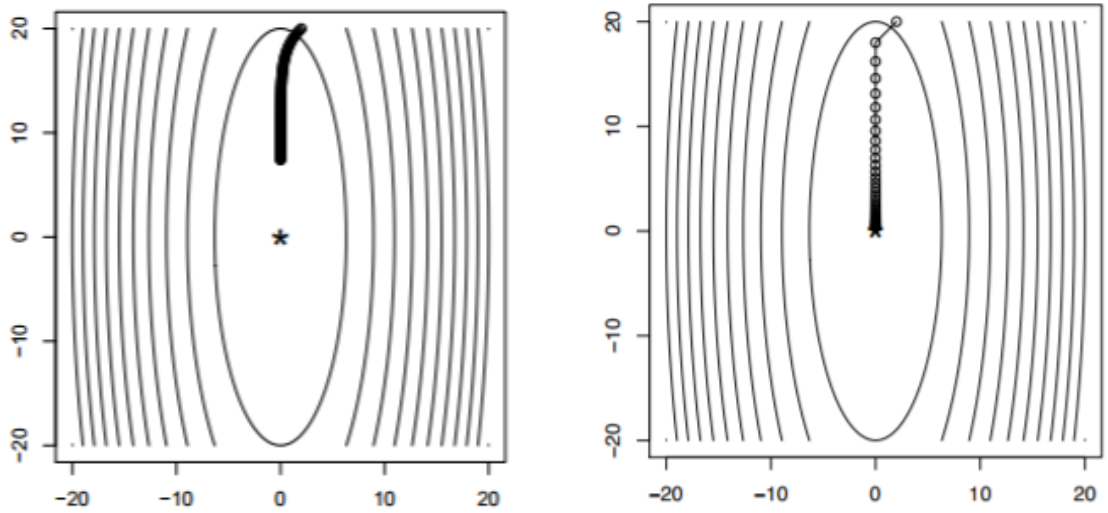


LEARNING RATE

So far we have assumed a fixed value of α in every iteration:

$$\alpha^{[t]} = \alpha \quad \forall t = \{1, \ldots, T\}$$

However, it makes sense to adapt α in every iteration:



Steps of gradient descent for $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = 10\,\theta_1^2 + 0.5\,\theta_2^2$. Left: 100 steps for with a fixed learning rate. Right: 40 steps with an adaptive learning rate.

WEIGHT INITIALIZATION

- Weights (and biases) of an NN must be initialized in GD.
- We somehow must "break symmetry" – which would happen in full-0-initialization. If two neurons (with the same activation) are connected to the same inputs and have the same initial weights, then both neurons will have the same gradient update and learn the same features.
- Weights are typically drawn from a uniform a Gaussian distribution (both centered at 0 with a small variance).
- Two common initialization strategies are 'Glorot initialization' and 'He initialization' which tune the variance of these distributions based on the topology of the network.

STOCHASTIC GRADIENT DESCENT

GD for ERM was:

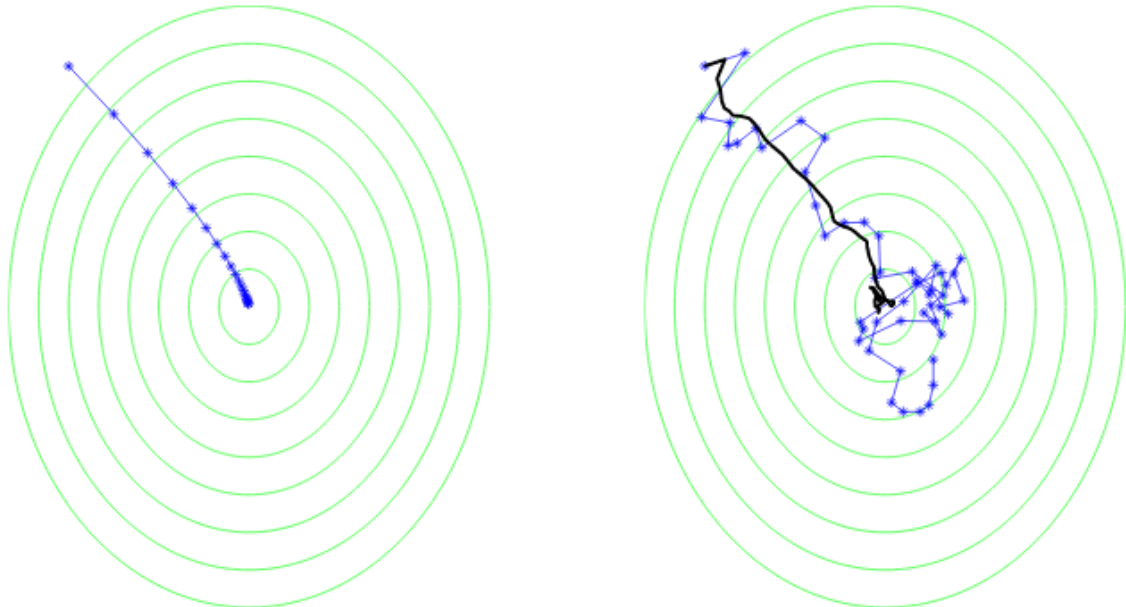
$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_\theta L \left(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t]}) \right)$$

- Using the entire training set in GD to is called **batch** or **deterministic** or **offline training**. This can be computationally costly or impossible, if data does not fit into memory.
- **Idea:** Instead of letting the sum run over the whole dataset, use small stochastic subsets (**minibatches**), or only a single $\mathbf{x}^{(i)}$.
- If batches are uniformly sampled from $\mathcal{D}_{\text{train}}$, our stochastic gradient is in expectation the batch gradient $\nabla_\theta \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$.
- The gradient w.r.t. a single $\mathbf{x}$ is fast to compute but not reliable. But its often used in a theoretical analysis of SGD.

$\rightarrow$ We have a **stochastic**, noisy version of the batch GD.

STOCHASTIC GRADIENT DESCENT

SGD on function $1.25(x_1 + 6)^2 + (x_2 - 8)^2$.



Source : Shalev-Shwartz and Ben-David. Understanding machine learning: From theory to algorithms. Cambridge University Press, 2014.

Figure: Left = GD, right = SGD. The black line is a smoothed θ.

STOCHASTIC GRADIENT DESCENT

Algorithm Basic SGD pseudo code

Initialize parameter vector $\boldsymbol{\theta}^{[0]}$
$t \leftarrow 0$
while stopping criterion not met **do**
    Randomly shuffle data and partition into minibatches $J_1, ..., J_K$ of size m
    for $k \in \{1, ..., K\}$ **do**
        $t \leftarrow t + 1$
        Compute gradient estimate with J_k: $\hat{g}^{[t]} \leftarrow \frac{1}{m} \sum_{i \in J_k} \nabla_\theta L(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t-1]}))$
        Apply update: $\boldsymbol{\theta}^{[t]} \leftarrow \boldsymbol{\theta}^{[t-1]} - \alpha \hat{g}^{[t]}$
    end for
end while

- A full SGD pass over data is an **epoch**.
- Minibatch sizes are typically between 50 and 1000.

STOCHASTIC GRADIENT DESCENT

- SGD is the most used optimizer in ML and especially in DL.
- We usually have to add a considerable amount of tricks to SGD to make it really efficient (e.g. momentum). More on this later.
- SGD with (small) batches has a high variance, although is unbiased. Hence, the LR α is smaller than in the batch mode.
- When LR is slowly decreased, SGD converges to local minimum.
- Recent results indicate that SGD often leads to better generalization than GD, and may result in indirect regularization.

INTRODUCTION TO DEEP LEARNING

Introduction

CONVOLUTIONAL NEURAL NETWORKS

- Convolutional Neural Networks (CNN, or ConvNet) are a powerful family of neural networks that are inspired by biological processes in which the connectivity pattern between neurons resembles the organization of the mamel visual cortex.

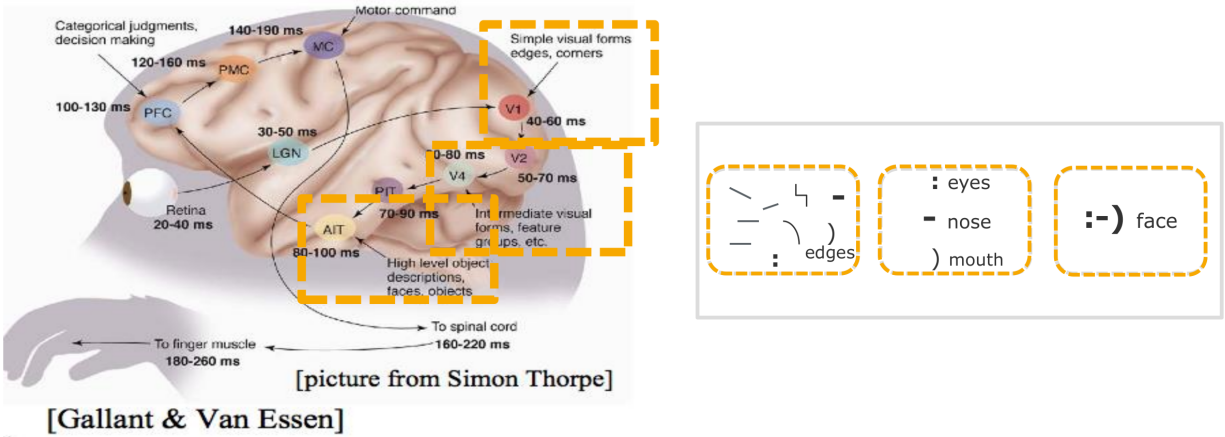


Figure: The ventral (recognition) pathway in the visual cortex has multiple stage: Retina - LGN - V1 - V2 - V4 - PIT - AIT etc., which consist of lots of intermediate representations.

CONVOLUTIONAL NEURAL NETWORKS

- Since 2012, given their success in the ILSVRC competition, CNNs are popular in many fields.
- Common applications of CNN-based architectures in computer vision are:
  - Image classification.
  - Object detection / localization.
  - Semantic segmentation.
- CNNs are widely applied in other domains such as natural language processing (NLP), audio, and time-series data.
- Basic idea: a CNN automatically extracts visual, or, more generally, spatial features from an input data such that it is able to make the optimal prediction based on the extracted features.
- It contains different building blocks and components.

CNNS - WHAT FOR?

Figure: All Tesla cars being produced now have full self-driving hardware (Source: Tesla website). A convolutional neural network is used to map raw pixels from a single front-facing camera directly into steering commands. The system learns to drive in traffic, on local roads, with or without lane markings as well as on highways.

CNNS - WHAT FOR?

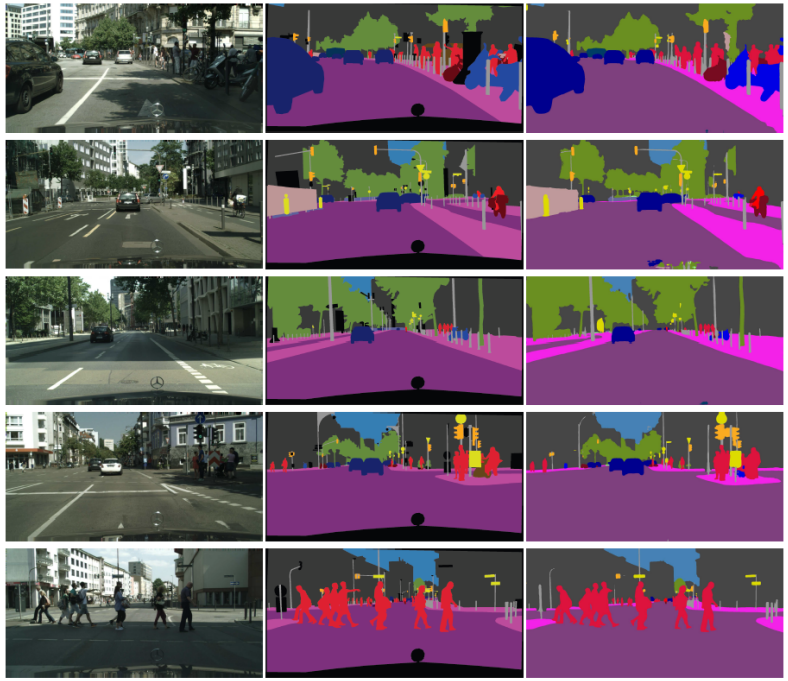


Figure: Given an input image, a CNN is first used to get the feature map of the last convolutional layer, then a pyramid parsing module is applied to harvest different sub-region representations, followed by upsampling and concatenation layers to form the final feature representation, which carries both local and global context information. Finally, the representation is fed into a convolution layer to get the final per-pixel prediction. (Source: pyramid scene parsing network, by Zhao et. al, CVPR 2017)

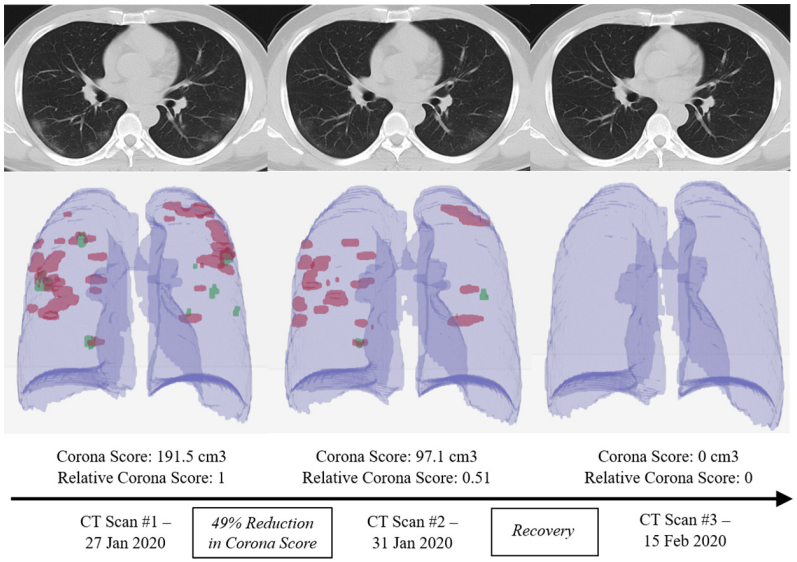
CNNS - WHAT FOR?

Figure: Road segmentation (Mnih Volodymyr (2013)). Aerial images and possibly outdated map pixels are segmented.

CNNS - WHAT FOR?

CNN for personalized medicine

- Examples:
  Tracking, diagnosis and localization of Covid-19 patients.
- CNN
  based method (RADLogists) for personalized Covid-19 detection: three CT scans from a single Corona virus patient diagnosed by RADLogists.



CNNS - WHAT FOR?

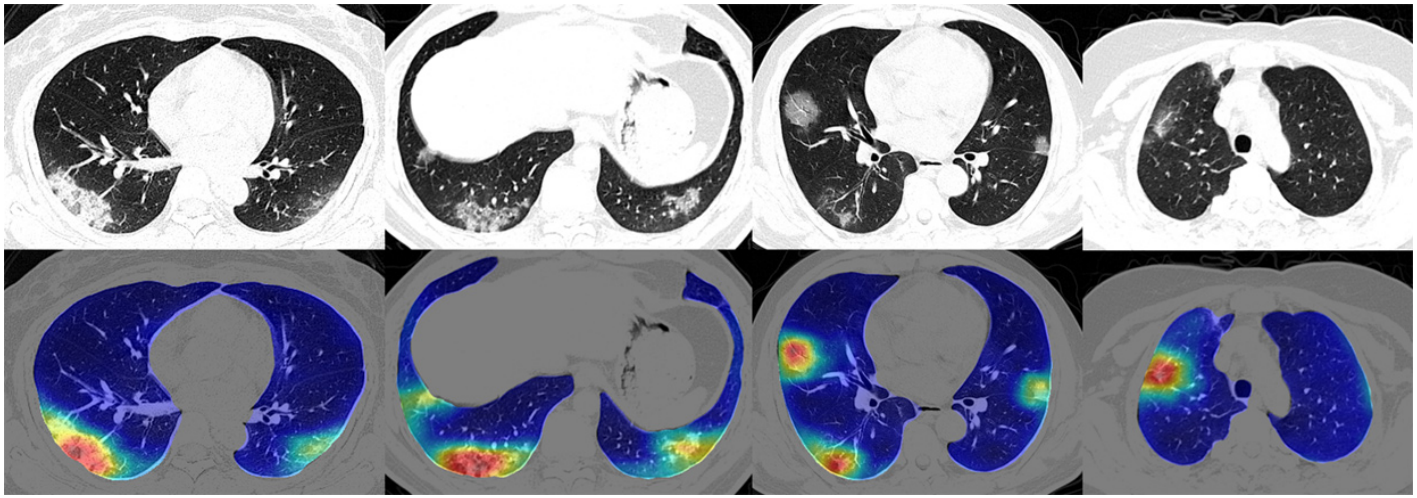


Figure: Four COVID-19 lung CT scans at the top with corresponding colored maps showing Corona virus abnormalities at the bottom (Source: Megan Scudellari, IEEE Spectrum 2021).

CNNS - WHAT FOR?

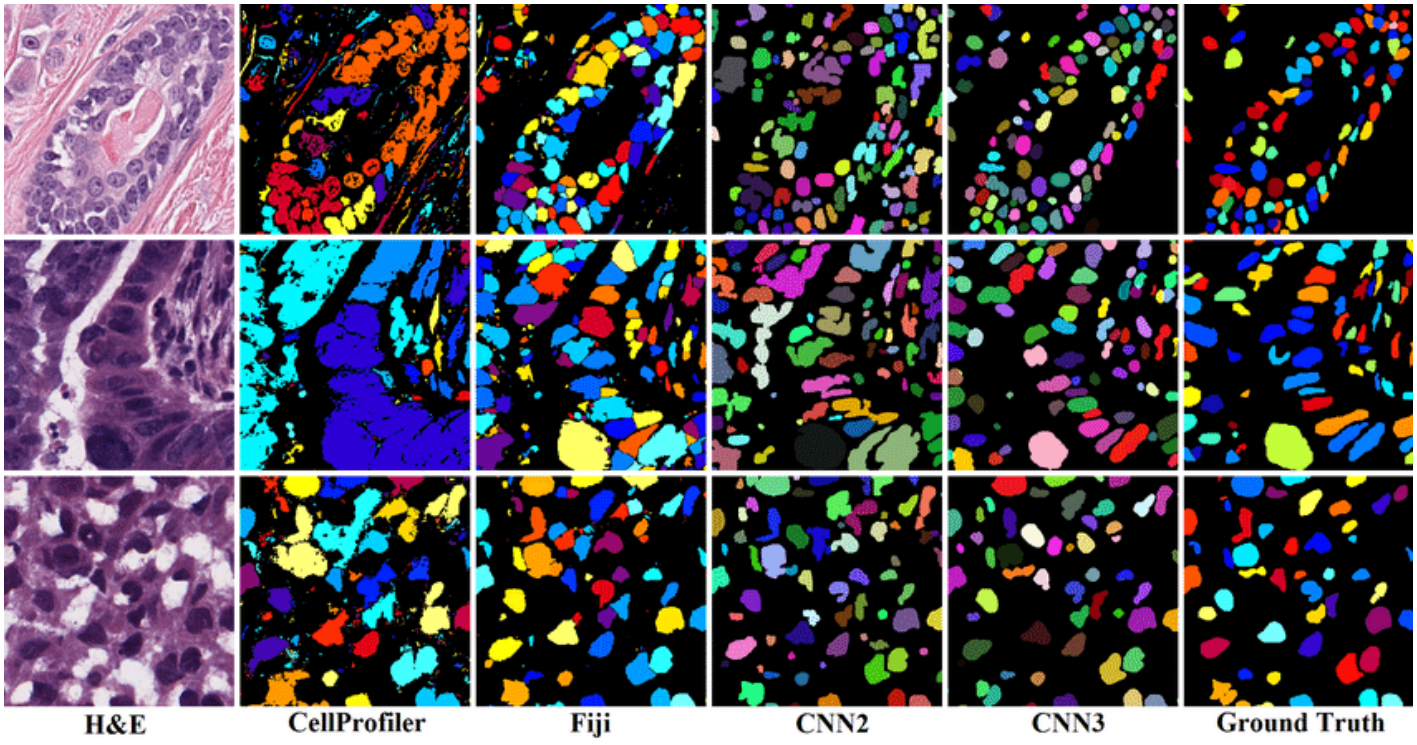


Figure: Various analyses in computational pathology are possible. For example, nuclear segmentation in digital microscopic tissue images enable extraction of high-quality features for nuclear morphometrics (Source: Kummar et. al. IEEE Transaction Medical Imaging).

CNNS - WHAT FOR?

Figure: Image Colorization is another interesting application of CNN in computer vision (Zhang et al. (2016)). Given a grayscale photo as the input (top row), this network solves the problem of hallucinating a plausible color version of the photo (bottom row, i.e. the prediction of the network).

CNNS - WHAT FOR?

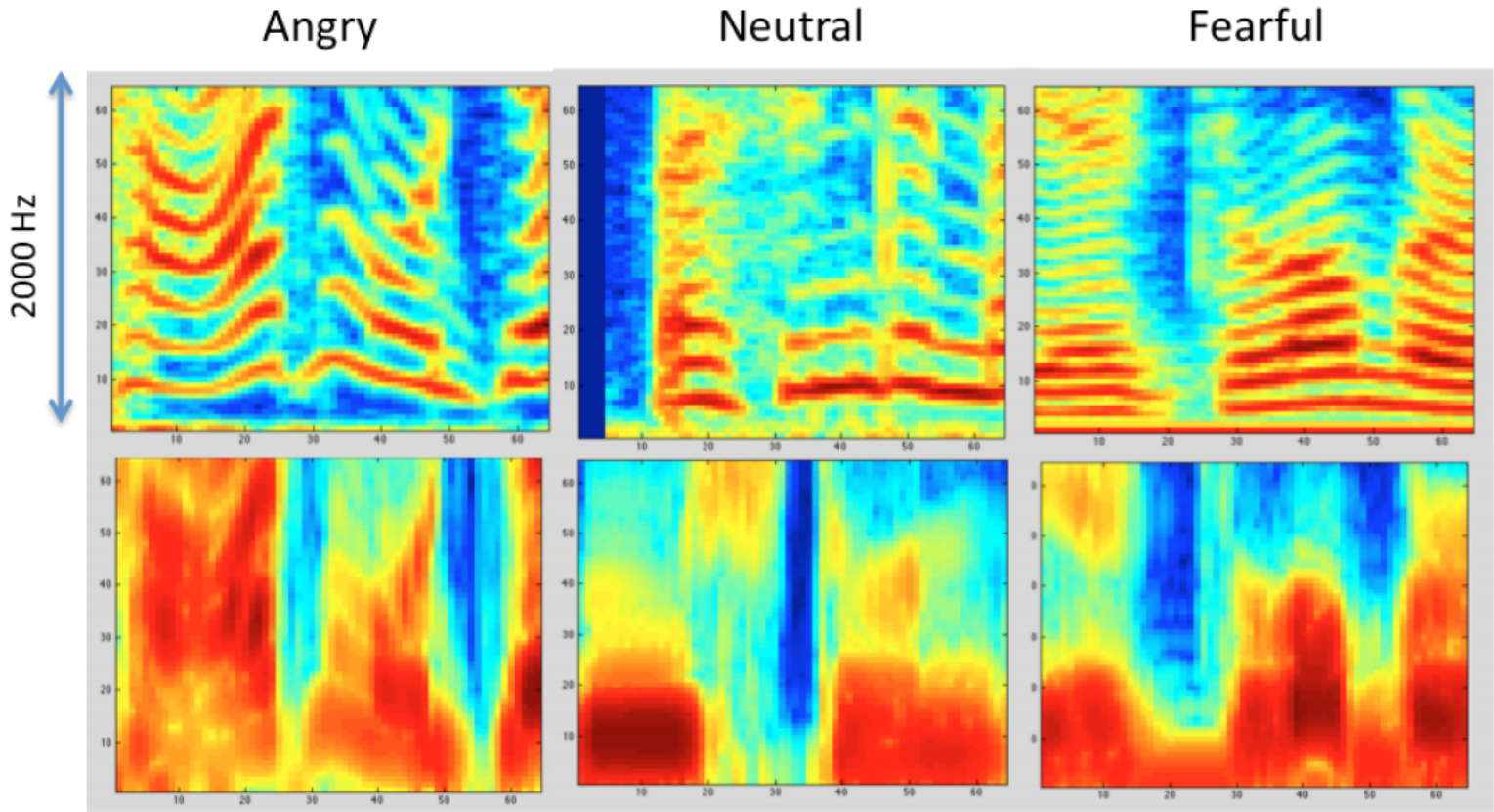
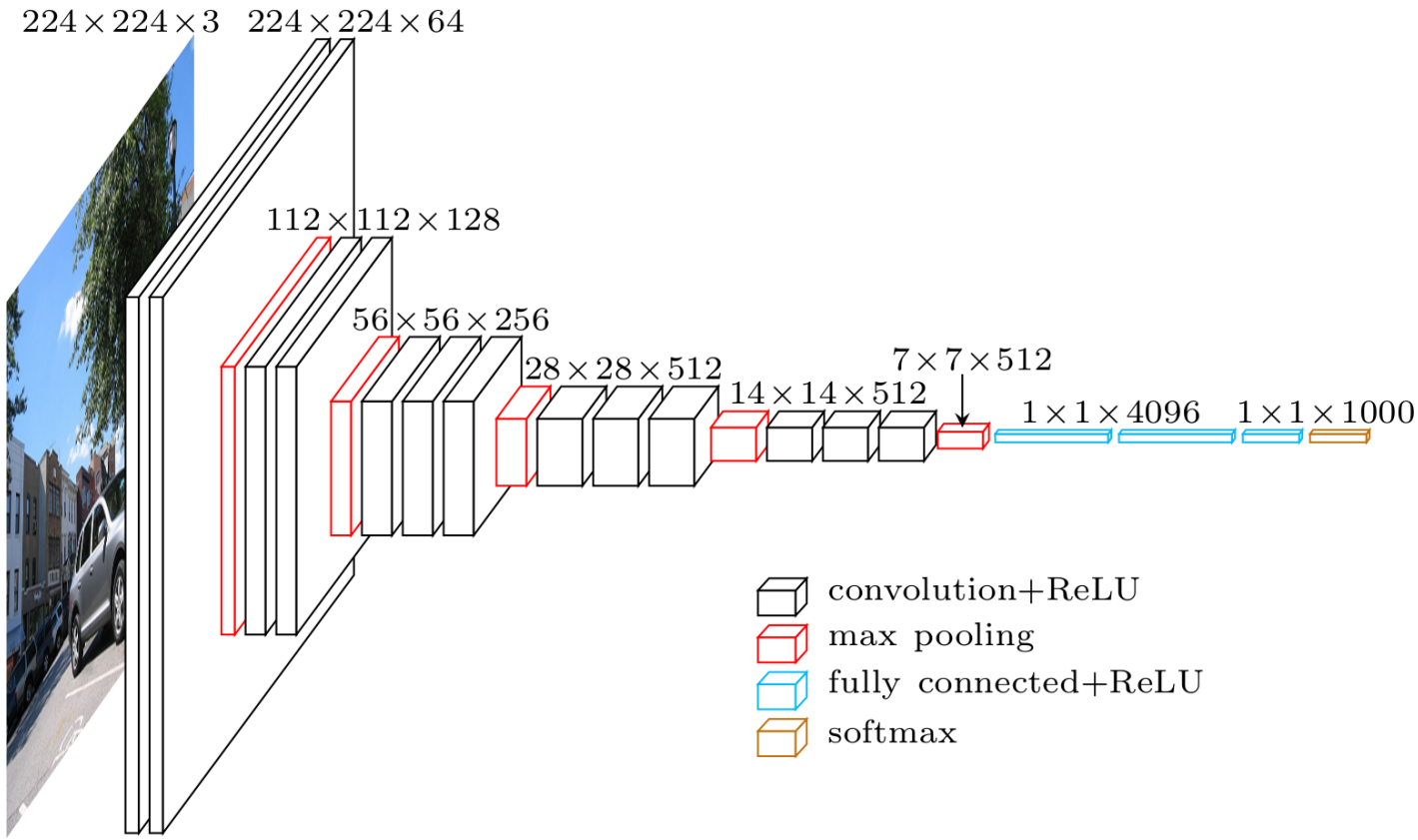


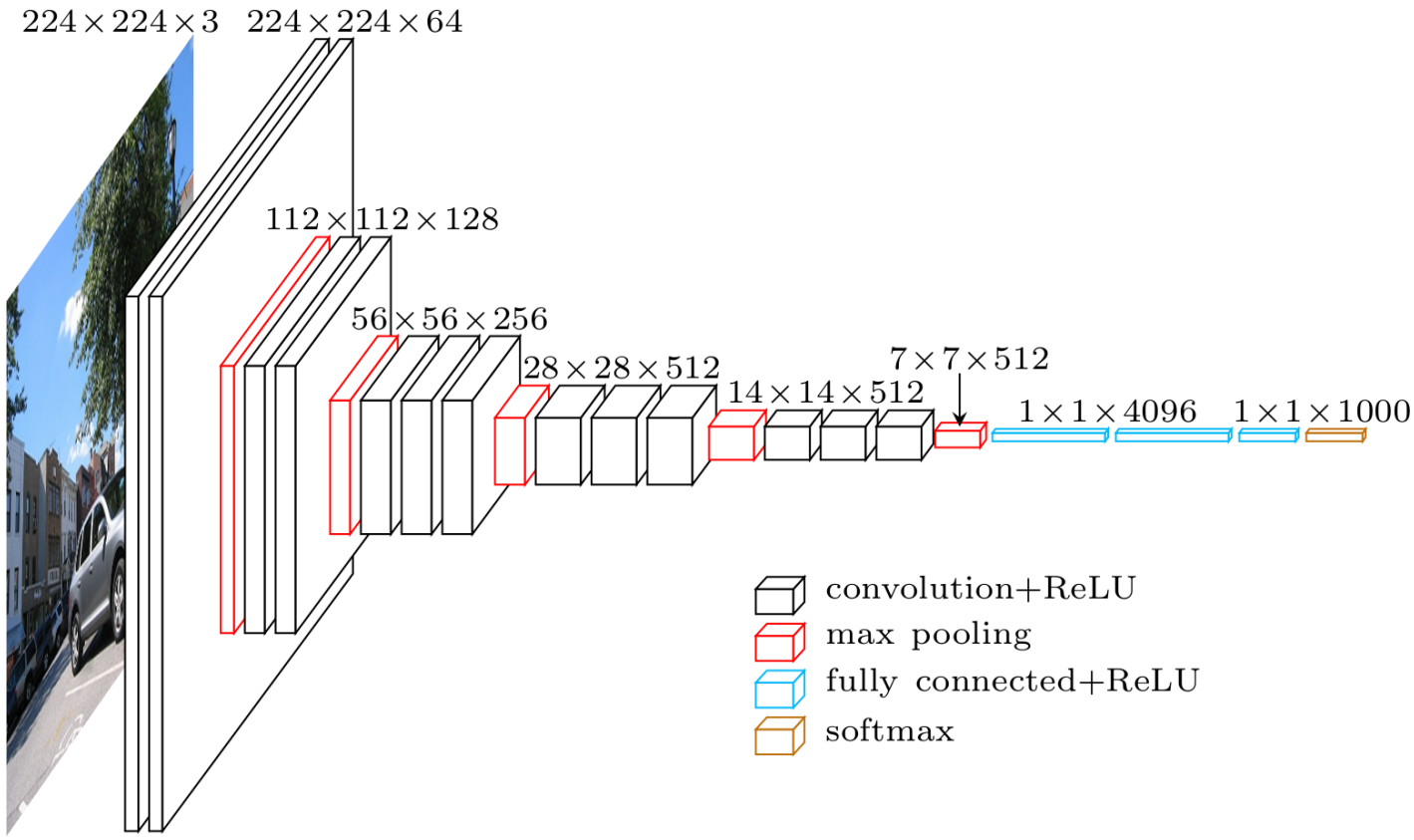
Figure: Speech recognition (Anand & Verma (2015)). Convolutional neural network is used to learn features from the audio data in order to classify emotions.

CNNS - A FIRST GLIMPSE



- **Input layer** takes input data (e.g. image, audio).
- **Convolution layers** extract feature maps from the previous layers.
- **Pooling layers** reduce the dimensionality of feature maps and filter meaningful features.

CNNS - A FIRST GLIMPSE



- **Fully connected layers** connect feature map elements to the output neurons.
- **Softmax** converts output values to probability scores.

Conv2D

FILTERS TO EXTRACT FEATURES

- Filters are widely applied in Computer Vision (CV) since the 70's.
- One prominent example: **Sobel-Filter**.
- It detects edges in images.

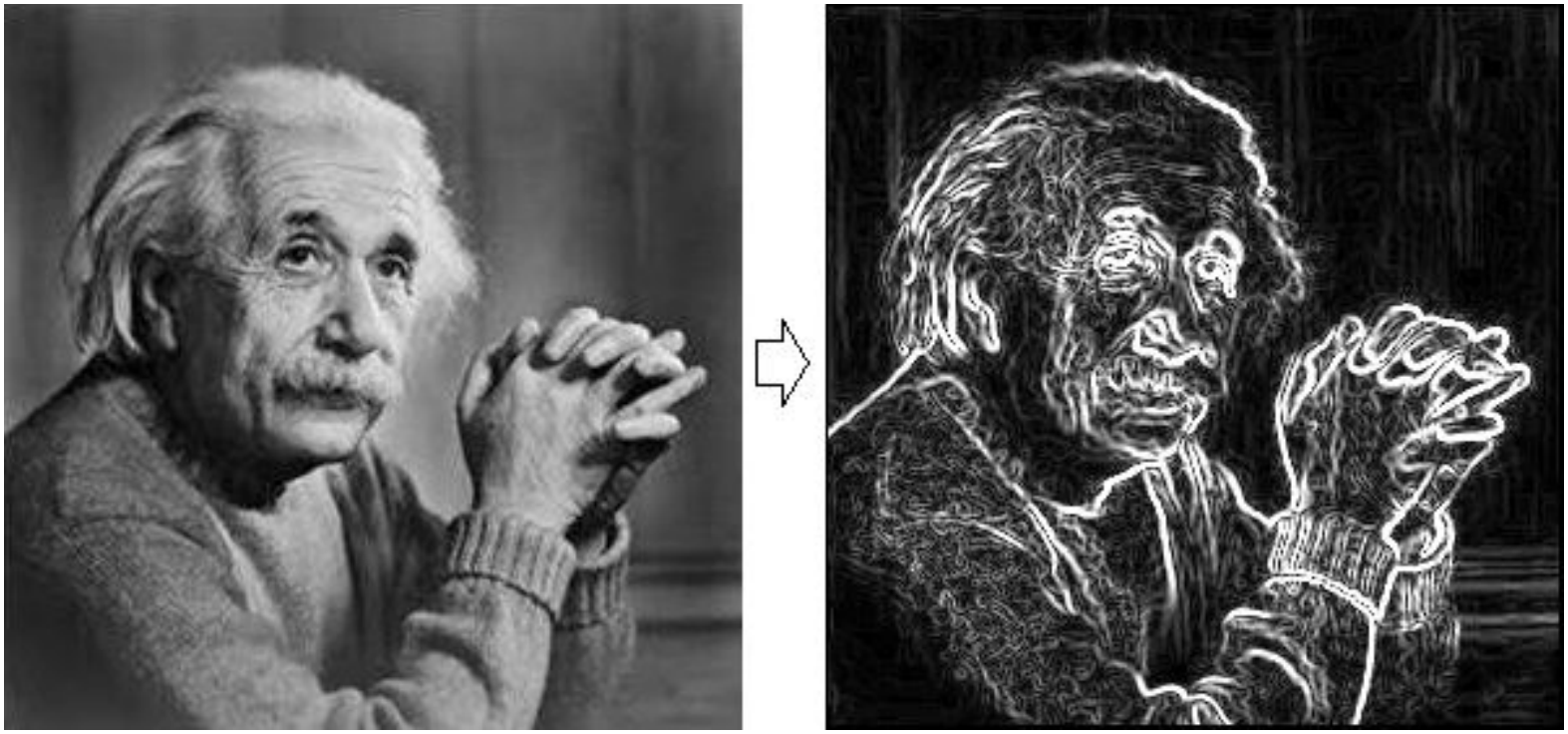


Figure: Sobel-filtered image.

FILTERS TO EXTRACT FEATURES

- Edges occur where the intensity over neighboring pixels changes fast.
- Thus, approximate the gradient of the intensity of each pixel.
- Sobel showed that the gradient image $\mathbf{G}_x$ of original image $\mathbf{A}$ in x-dimension can be approximated by:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} = \mathbf{S}_x * \mathbf{A}$$

  where $*$ indicates a mathematical operation known as a **convolution**, not a traditional matrix multiplication.
- The filter matrix $\mathbf{S}_x$ consists of the product of an **averaging** and a **differentiation** kernel:

$$\underbrace{\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}^T}_{\textit{averaging}} \underbrace{\begin{bmatrix} -1 & 0 & +1 \end{bmatrix}}_{\textit{differentiation}}$$

FILTERS TO EXTRACT FEATURES

- Similarly, the gradient image $\mathbf{G}_y$ in y-dimension can be approximated by:

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} = \mathbf{S}_y * \mathbf{A}$$

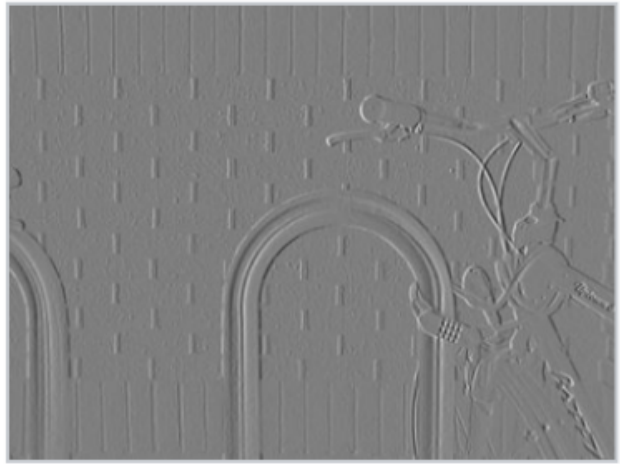
- The combination of both gradient images yields a dimension-independent gradient information $\mathbf{G}$:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

- These matrix operations were used to create the filtered picture of Albert Einstein.

HORIZONTAL VS VERTICAL EDGES

Input



Vertical edges detected by S_x

Horizontal edges detected by S_y

Combined

Source: Wikipedia

Figure: Sobel filtered images. Outputs are normalized in each case.

FILTERS TO EXTRACT FEATURES

- Let's do this on a dummy image.
- How to represent a digital image?

FILTERS TO EXTRACT FEATURES

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

- Basically as an array of integers.

FILTERS TO EXTRACT FEATURES

Sobel-Operator

$$S_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & \boxed{0} & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

- $\mathbf{S}_X$ enables us to to detect vertical edges!

FILTERS TO EXTRACT FEATURES

Sobel-Operator

$$S_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

FILTERS TO EXTRACT FEATURES

Sobel-Operator

$$S_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

0	0	0	0	255	255	0	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	0	255	255	0	0	0	0
255	0	0	255	255	255	255	0	0	0
0	0	255	255	255	255	255	255	0	0
0	255	0	255	255	255	255	0	255	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	255	255	255	0	0	0
0	0	0	255	0	0	255	0	0	255
0	0	255	255	0	0	255	255	0	0

$$\begin{aligned} (\mathbf{G}_x)_{(i,j)} = (\mathbf{I} \star \mathbf{S}_x)_{(i,j)} = & -1 \cdot 0 + 0 \cdot 255 + \mathbf{1 \cdot 255} \\ & -2 \cdot 0 + 0 \cdot 0 + \mathbf{2 \cdot 255} \\ & -1 \cdot 0 + 0 \cdot 255 + \mathbf{1 \cdot 255} \end{aligned}$$

FILTERS TO EXTRACT FEATURES

0	510	1020	510	-510	-1020	-510	0
-255	510	1020	510	-510	-1020	-510	0
-255	765	765	255	-255	-765	-765	-255
255	765	510	0	0	-510	-765	-510
255	510	765	0	0	-765	-510	-255
0	765	1020	0	0	-1020	-765	0
0	1020	765	-255	255	-765	-1020	255
255	1020	0	-765	765	0	-1020	255

- Applying the Sobel-Operator to every location in the input yields the **feature map**.

FILTERS TO EXTRACT FEATURES

128	191	255	191	64	0	64	128
96	191	255	191	64	0	64	128
96	223	223	159	96	32	32	96
159	223	191	128	128	64	32	64
159	191	223	128	128	32	64	96
128	223	255	128	128	0	32	128
128	255	223	96	159	32	0	159
159	255	128	32	223	128	0	159

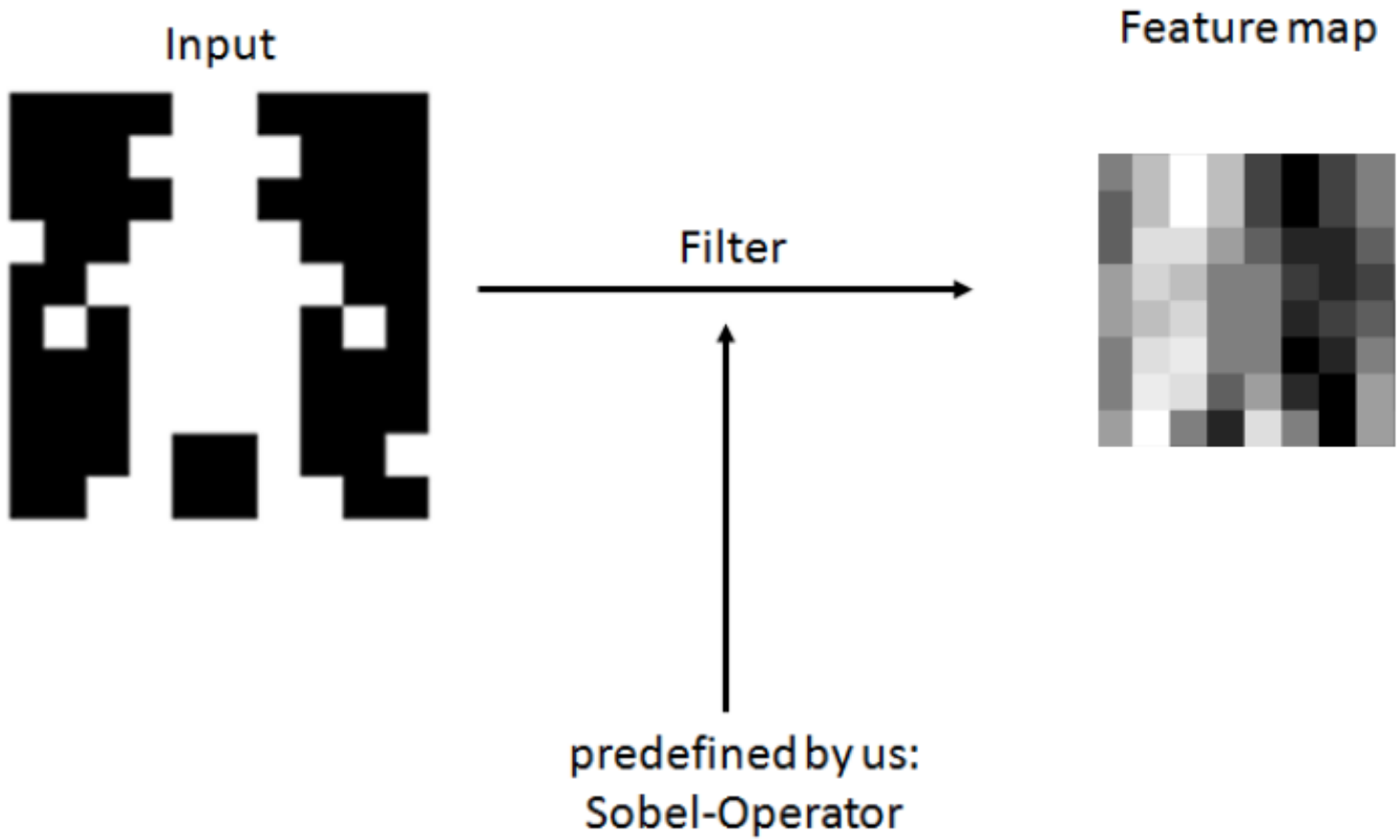
- Normalized feature map reveals vertical edges.
- Note the dimensional reduction compared to the dummy image.

WHY DO WE NEED TO KNOW ALL OF THAT?

- What we just did was extracting **pre-defined** features from our input (i.e. edges).
- A convolutional neural network does almost exactly the same: “extracting features from the input”.

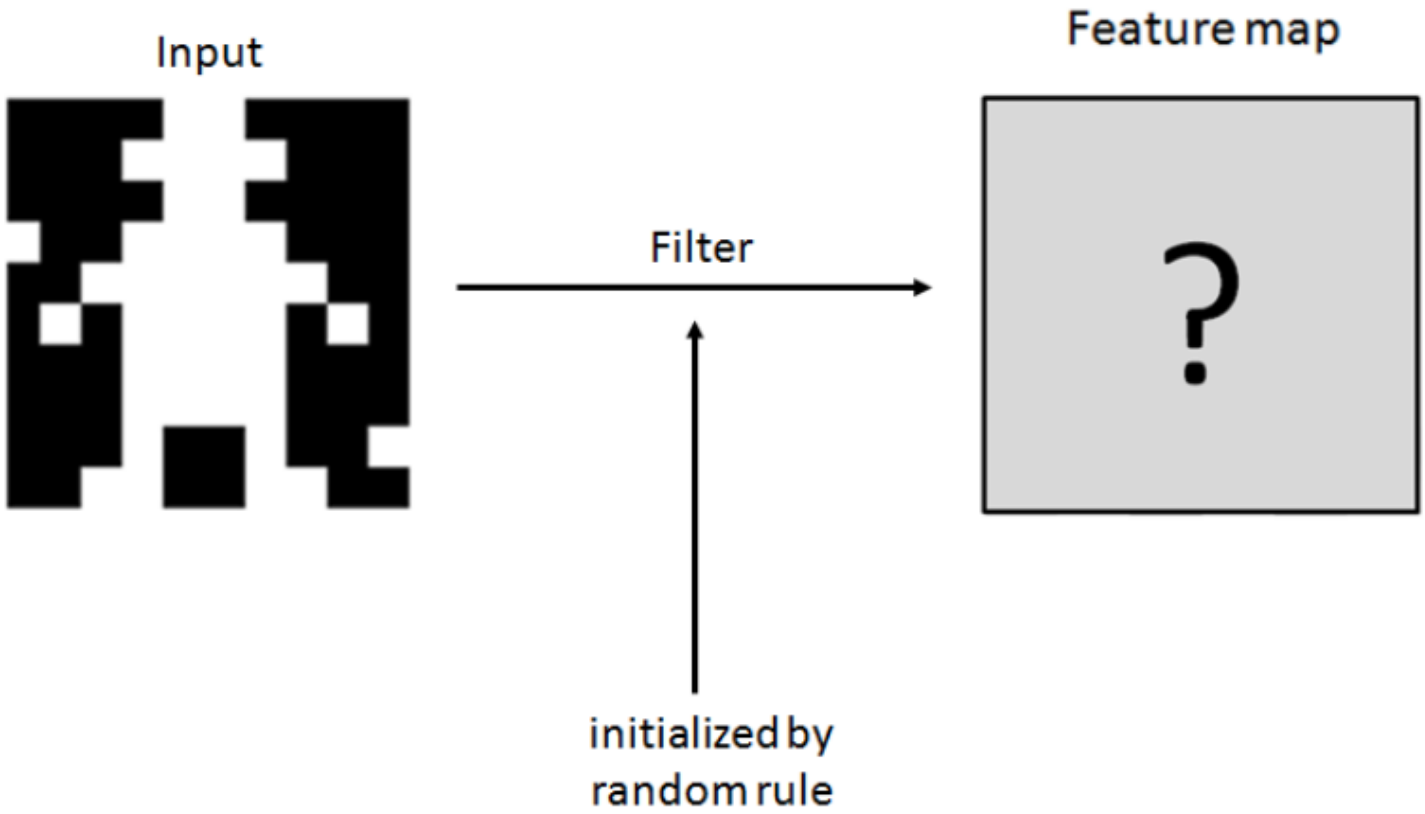
  $\Rightarrow$ The main difference is that we usually do not tell the CNN what to look for (pre-define them), **the CNN decides itself**.
- In a nutshell:
  - We initialize a lot of random filters (like the Sobel but just random entries) and apply them to our input.
  - Then, a classifier which (e.g. a feed forward neural net) uses them as input data.
  - Filter entries will be adjusted by common gradient descent methods.

WHY DO WE NEED TO KNOW ALL OF THAT?



$$S_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

WHY DO WE NEED TO KNOW ALL OF THAT?



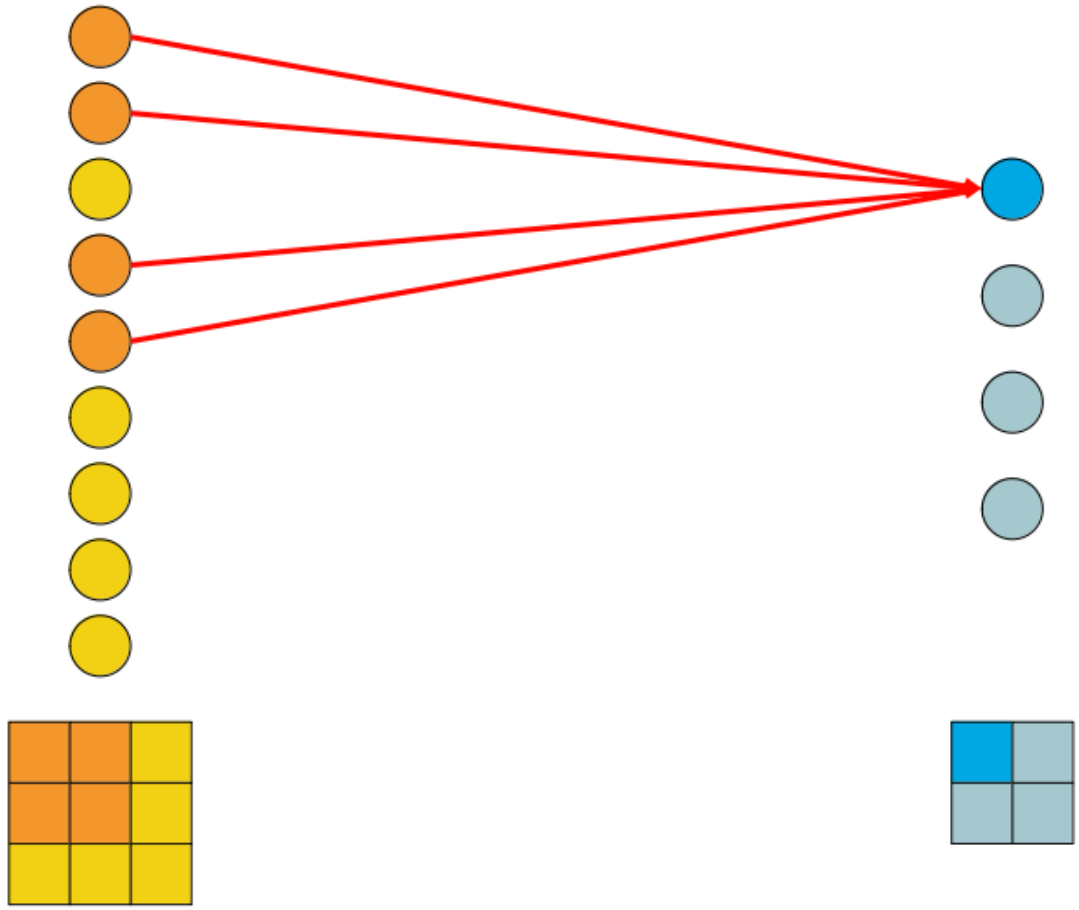
$$S_X = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

WORKING WITH IMAGES

- In order to understand the functionality of CNNs, we have to familiarize ourselves with some properties of images.
- Grey scale images:
  - Matrix with dimensions **h**eight $\times$ **w**idth $\times$ 1.
  - Pixel entries differ from 0 (black) to 255 (white).
- Color images:
  - Tensor with dimensions **h**eight $\times$ **w**idth $\times$ 3.
  - The depth 3 denotes the RGB values (red - green - blue).
- Filters:
  - A filter's depth is **always** equal to the input's depth!
  - In practice, filters are usually square.
  - Thus we only need one integer to define its size.
  - For example, a filter of size 2 applied on a color image actually has the dimensions $2 \times 2 \times 3$.

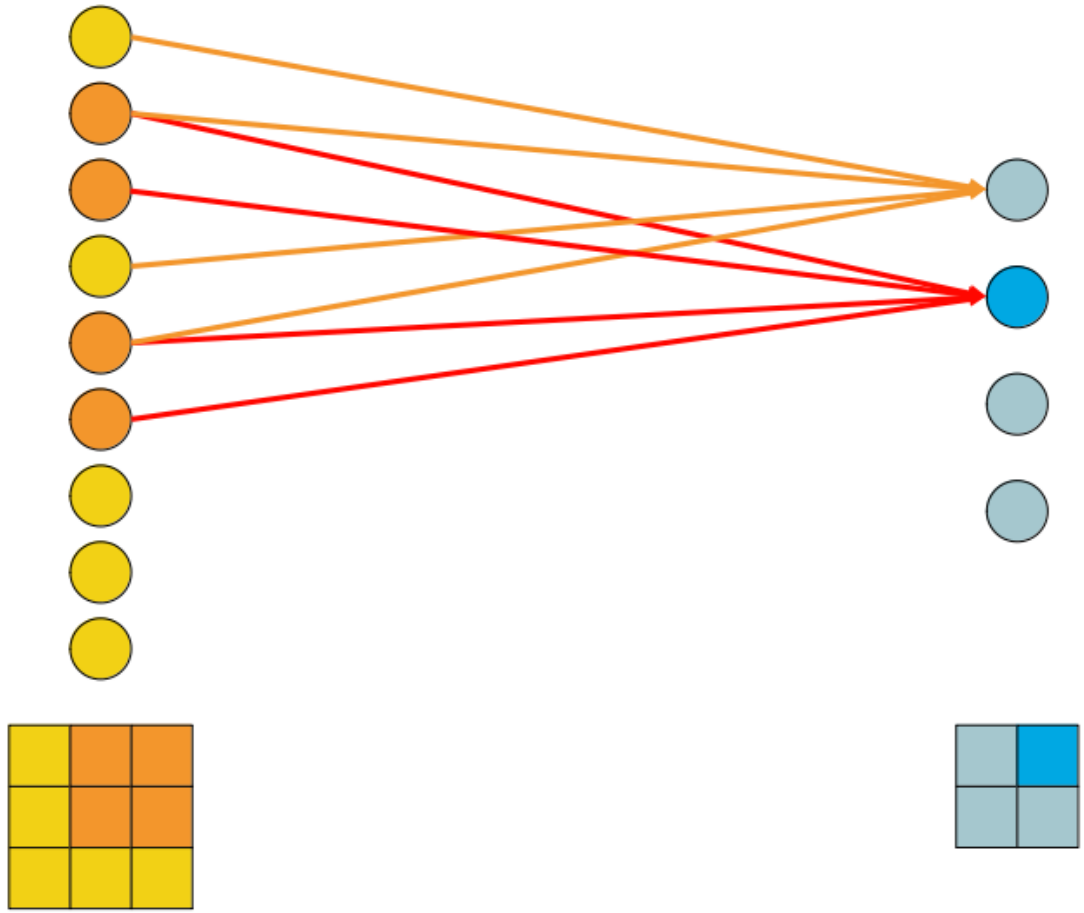
Properties of Convolutional Layers

SPARSE INTERACTIONS



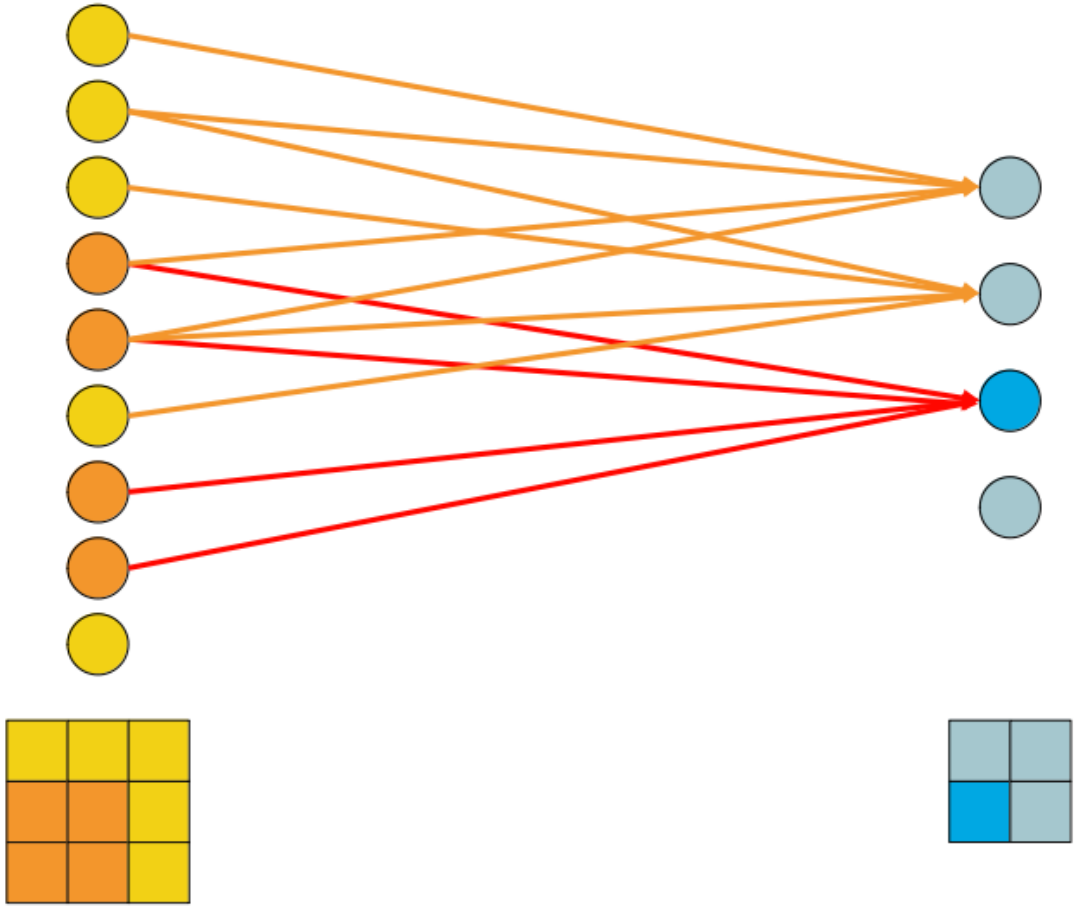
- We want to use the “neuron-wise” representation of our CNN.
- Moving the filter to the first spatial location yields the first entry of the feature map which is composed of these four connections.

SPARSE INTERACTIONS



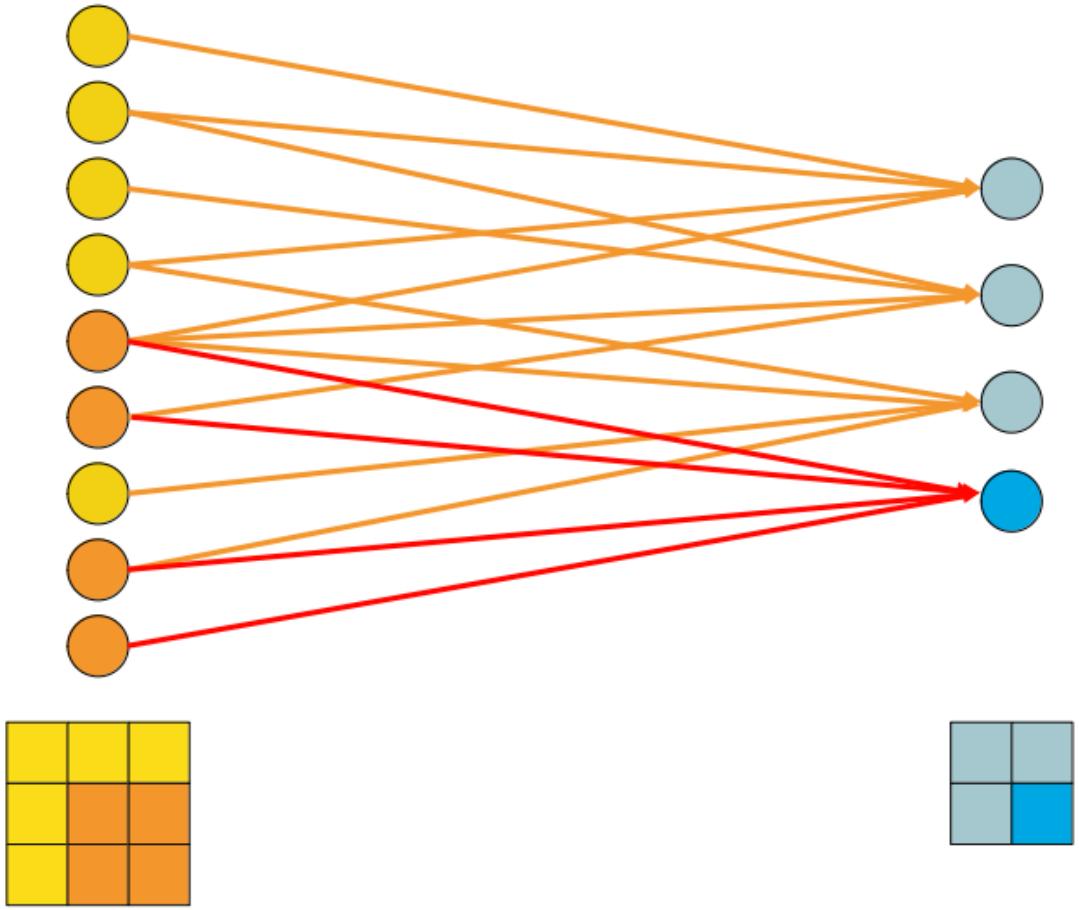
- Similarly...

SPARSE INTERACTIONS



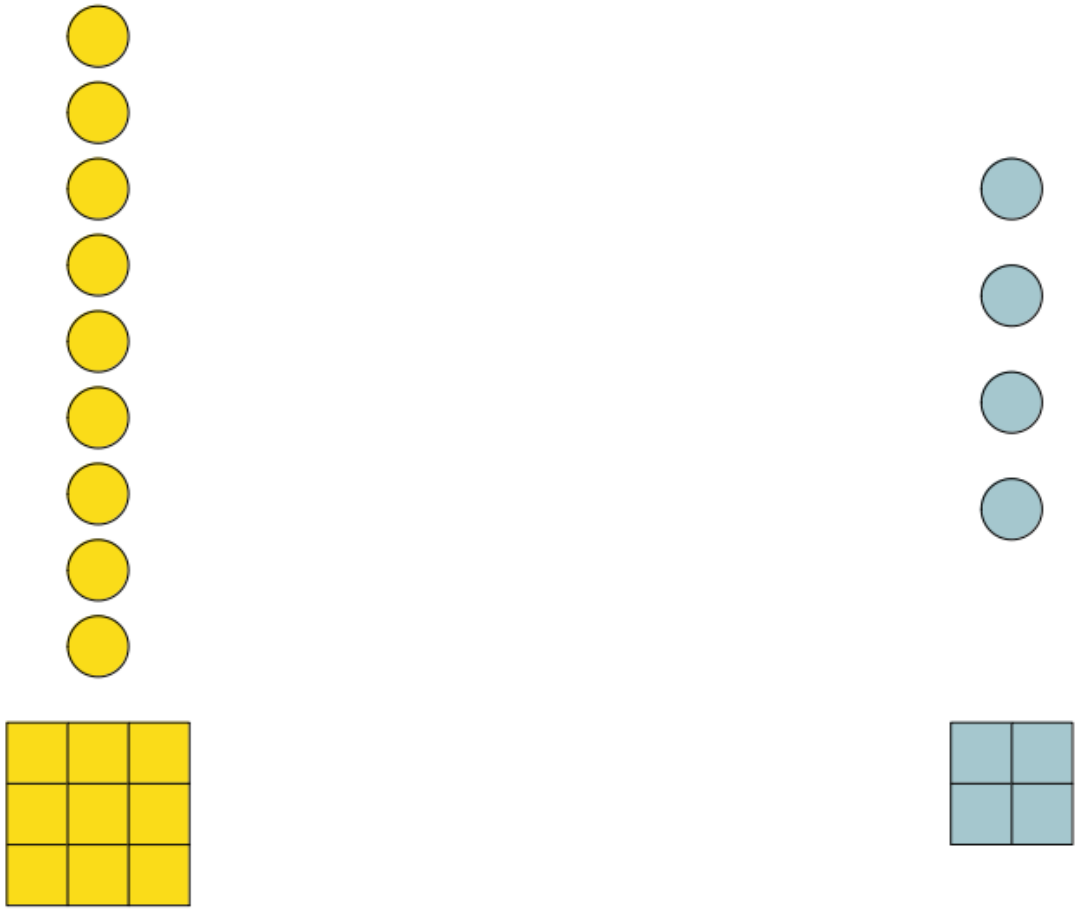
- Similarly...

SPARSE INTERACTIONS



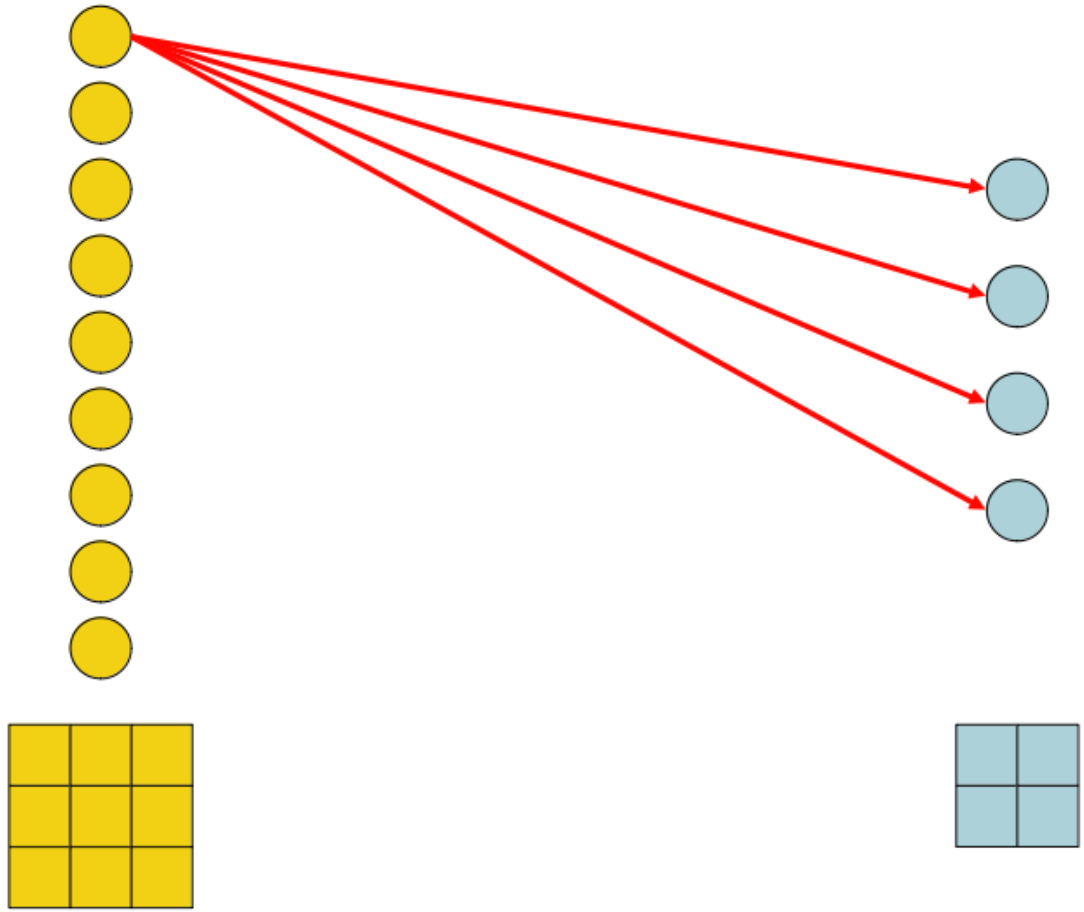
- and finally s_{22} by these and in total, we obtain 16 connections!

SPARSE INTERACTIONS



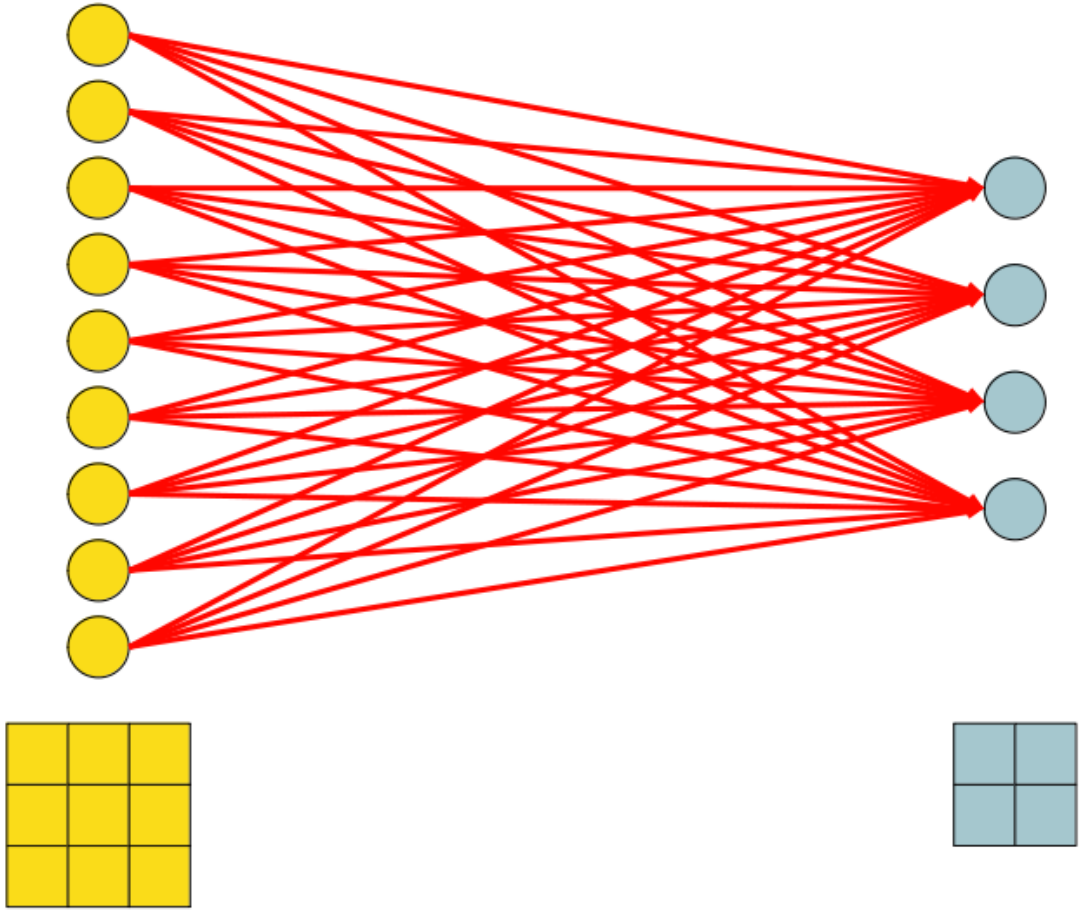
- Assume we would replicate the architecture with a dense net.

SPARSE INTERACTIONS



- Each input neuron is connected with each hidden layer neuron.

SPARSE INTERACTIONS

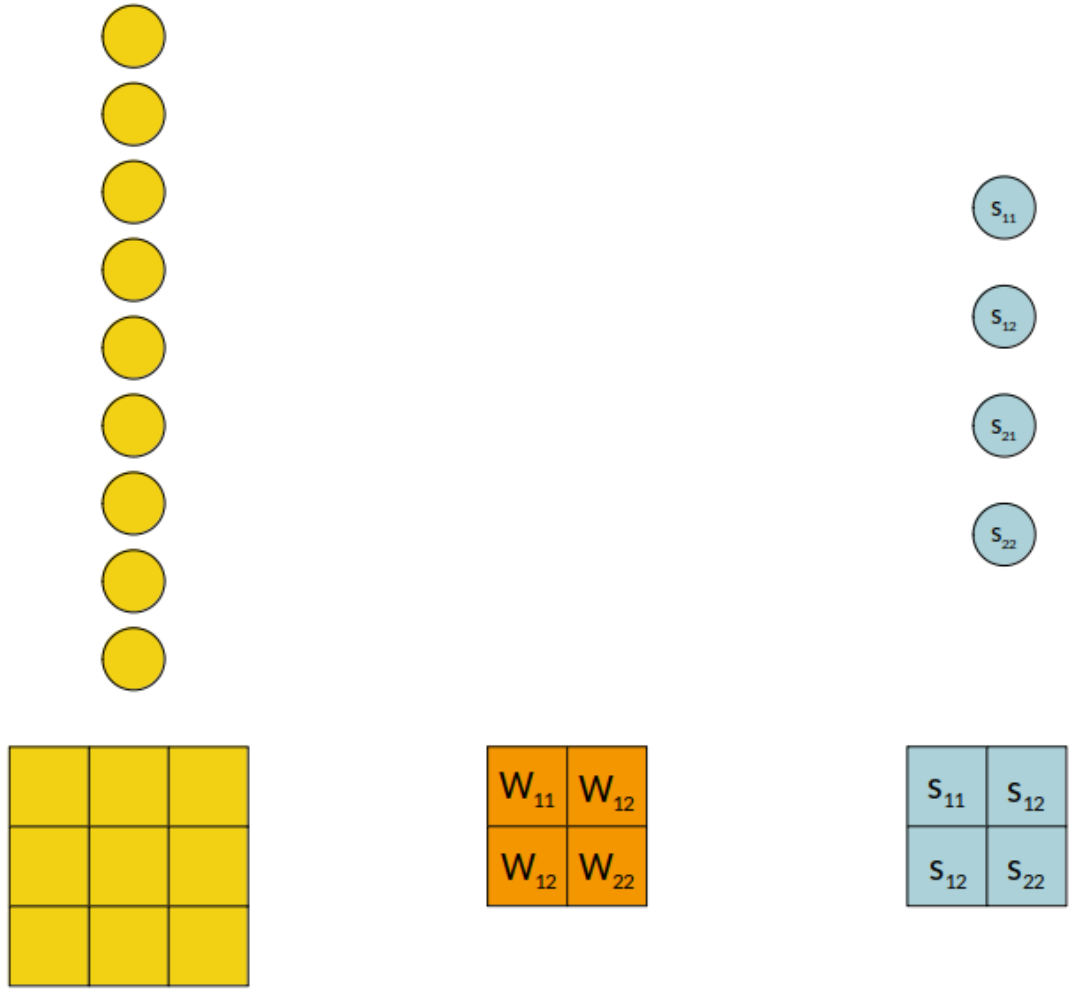


- In total, we obtain 36 connections!

SPARSE INTERACTIONS

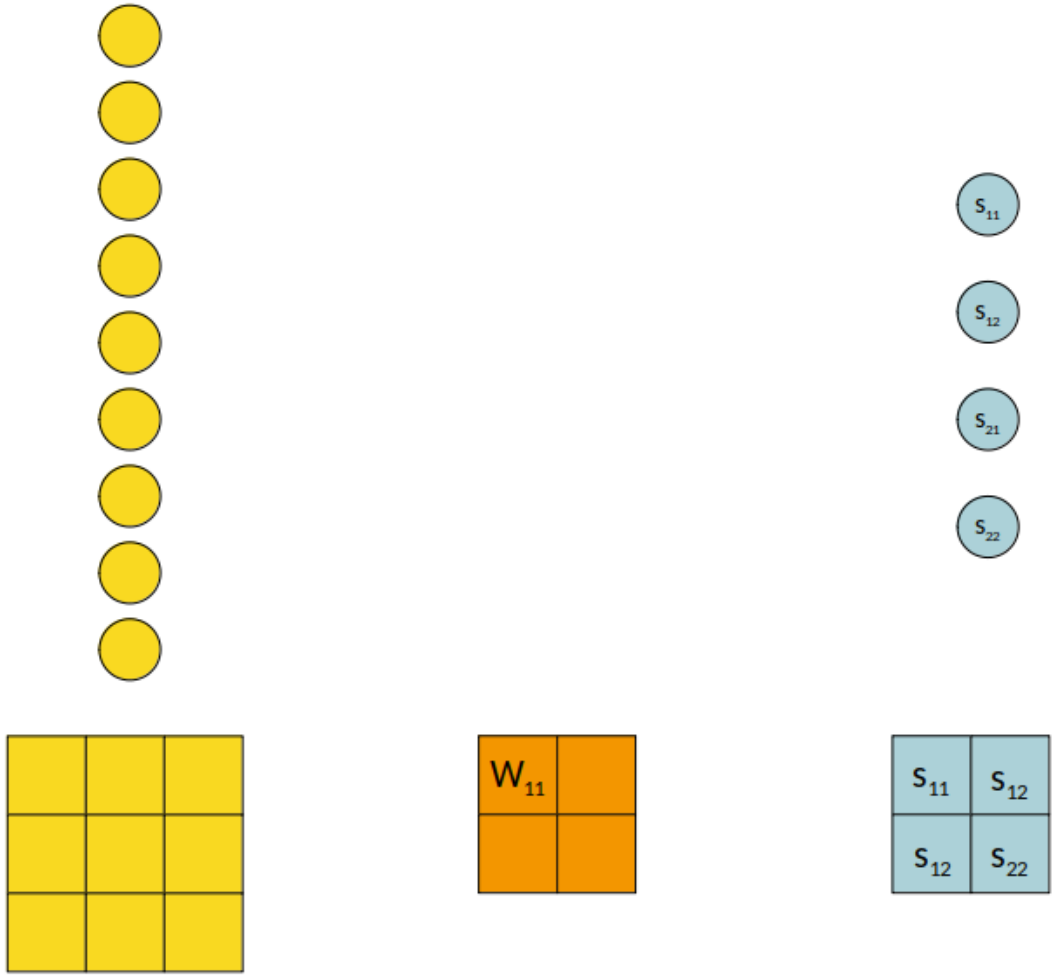
- What does that mean?
  - Our CNN has a **receptive field** of 4 neurons.
  - That means, we apply a “local search” for features.
  - A dense net on the other hand conducts a “global search”.
  - The receptive field of the dense net are 9 neurons.
- When processing images, it is more likely that features occur at specific locations in the input space.
- For example, it is more likely to find the eyes of a human in a certain area, like the face.
  - A CNN only incorporates the surrounding area of the filter into its feature extraction process.
  - The dense architecture on the other hand assumes that every single pixel entry has an influence on the eye, even pixels far away or in the background.

PARAMETER SHARING



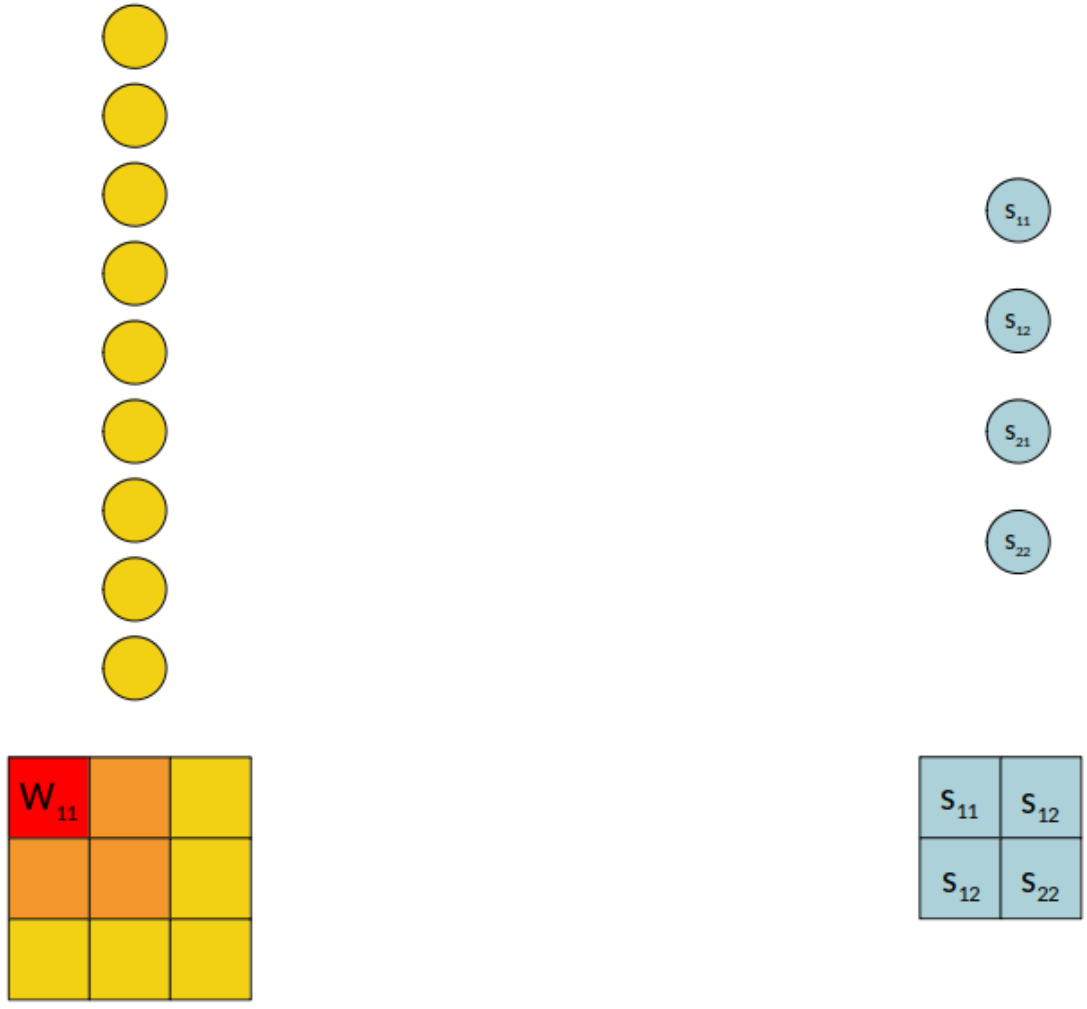
- For the next property we focus on the filter entries.

PARAMETER SHARING



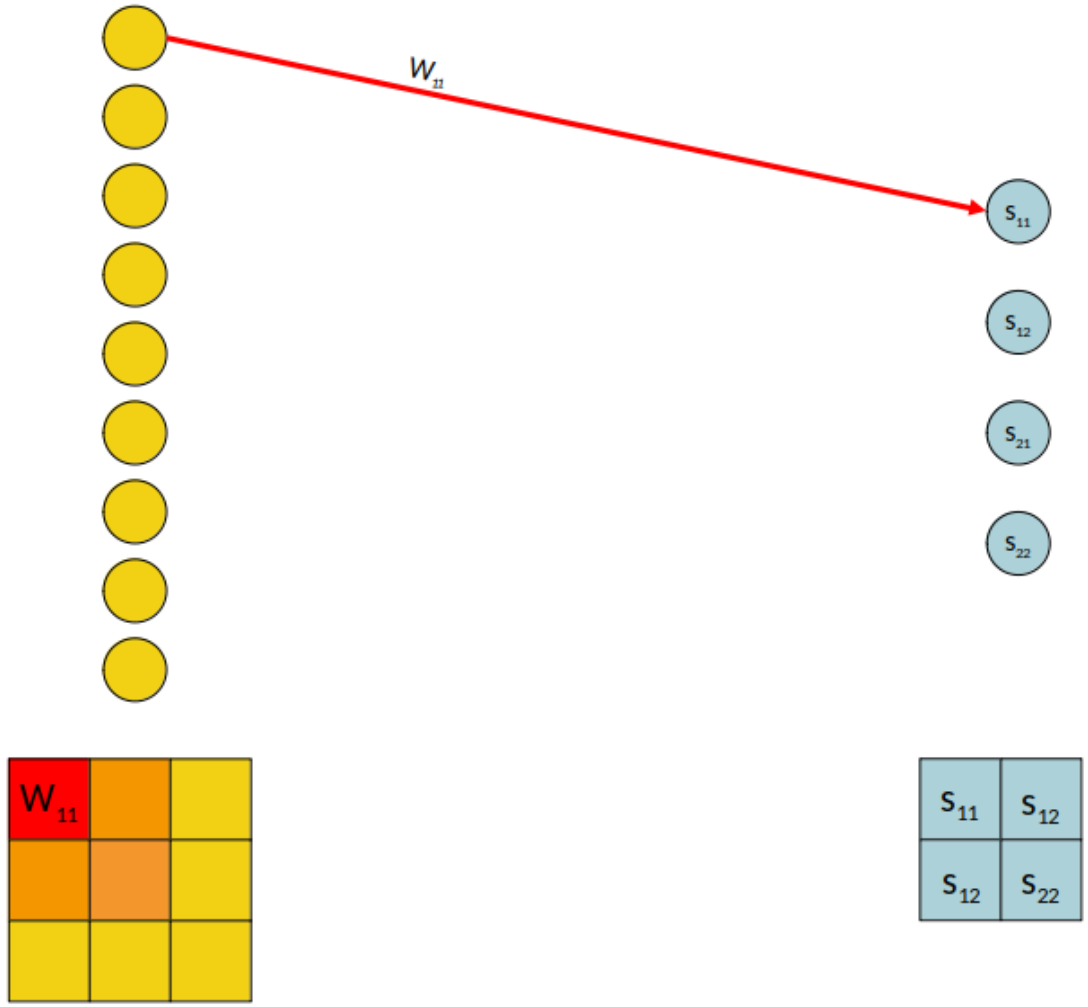
- In particular, we consider weight w_{11}

PARAMETER SHARING



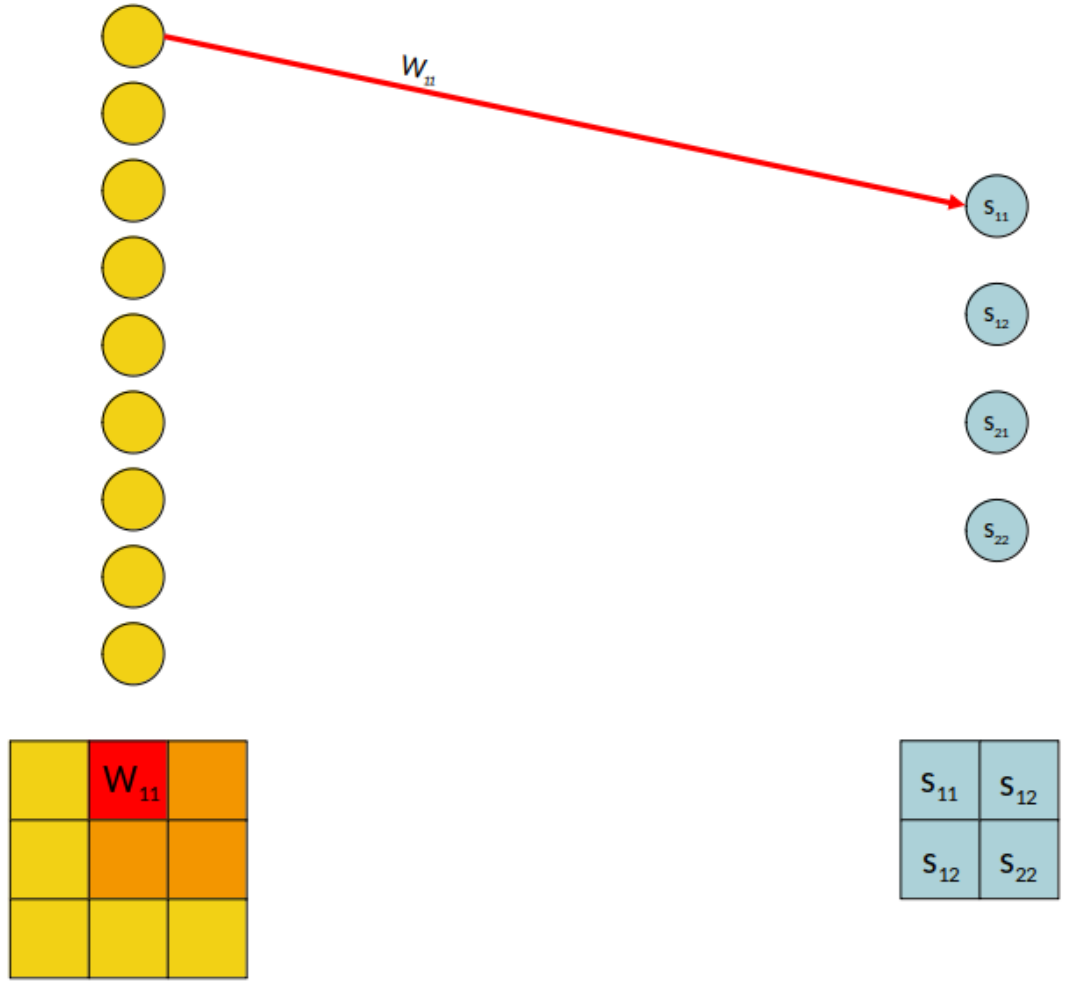
- As we move the filter to the first spatial location..

PARAMETER SHARING



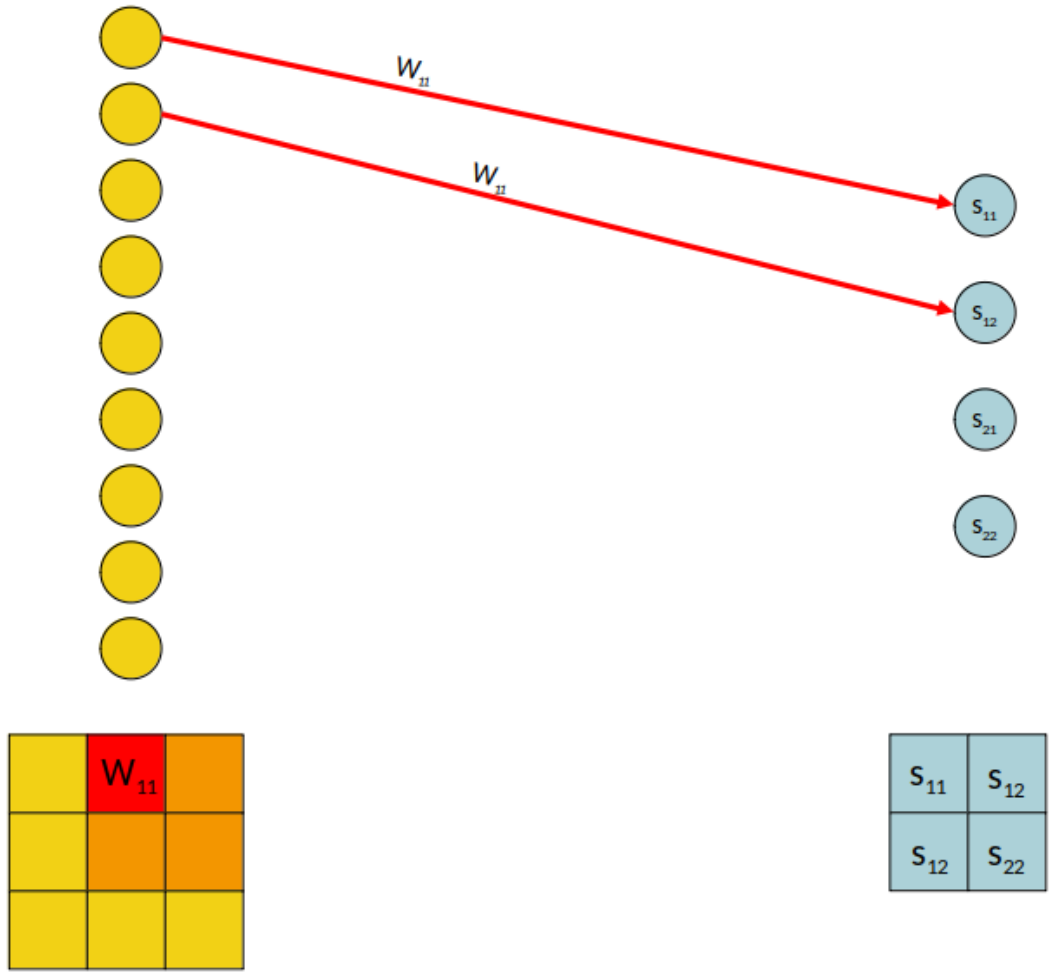
- ...we observe the following connection for weight w_{11}

PARAMETER SHARING



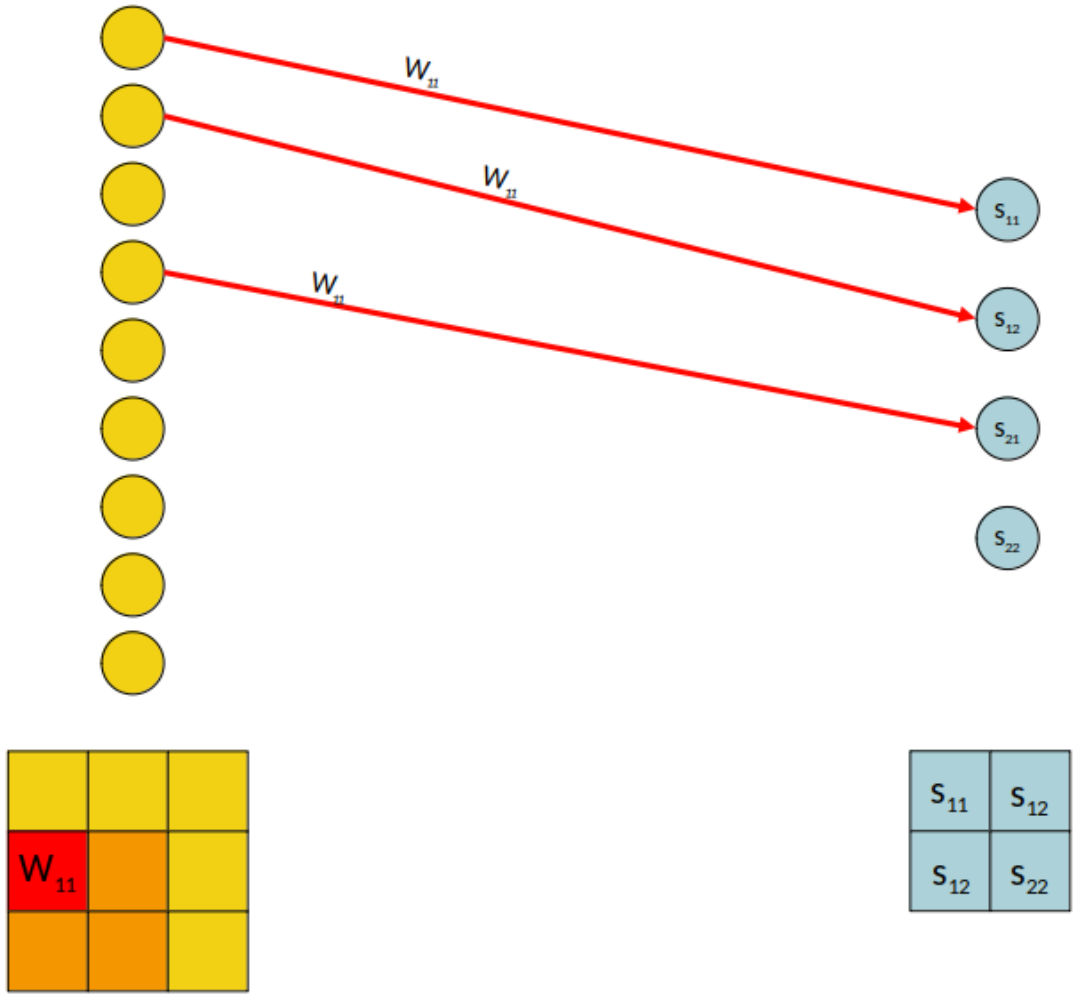
- Moving to the next location...

PARAMETER SHARING



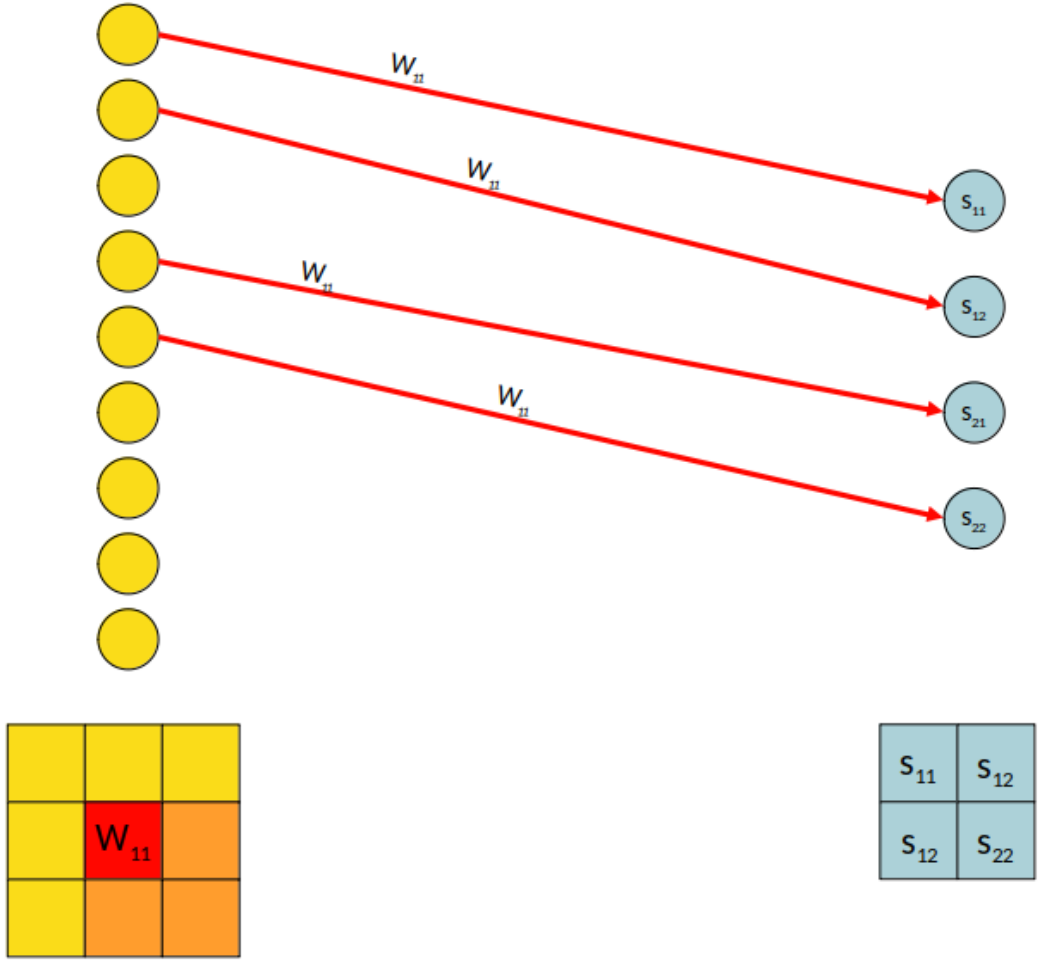
- ...highlights that we use the same weight more than once!

PARAMETER SHARING



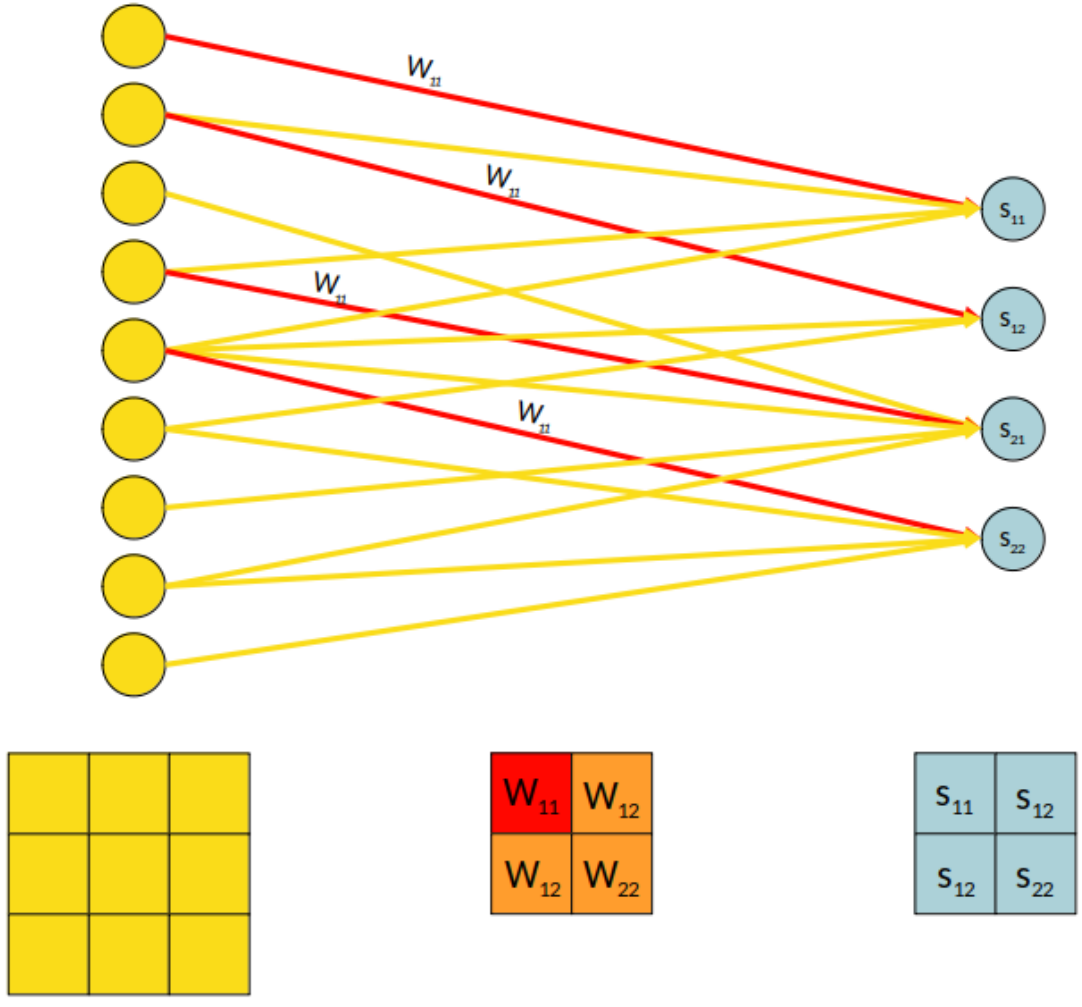
- Even three...

PARAMETER SHARING



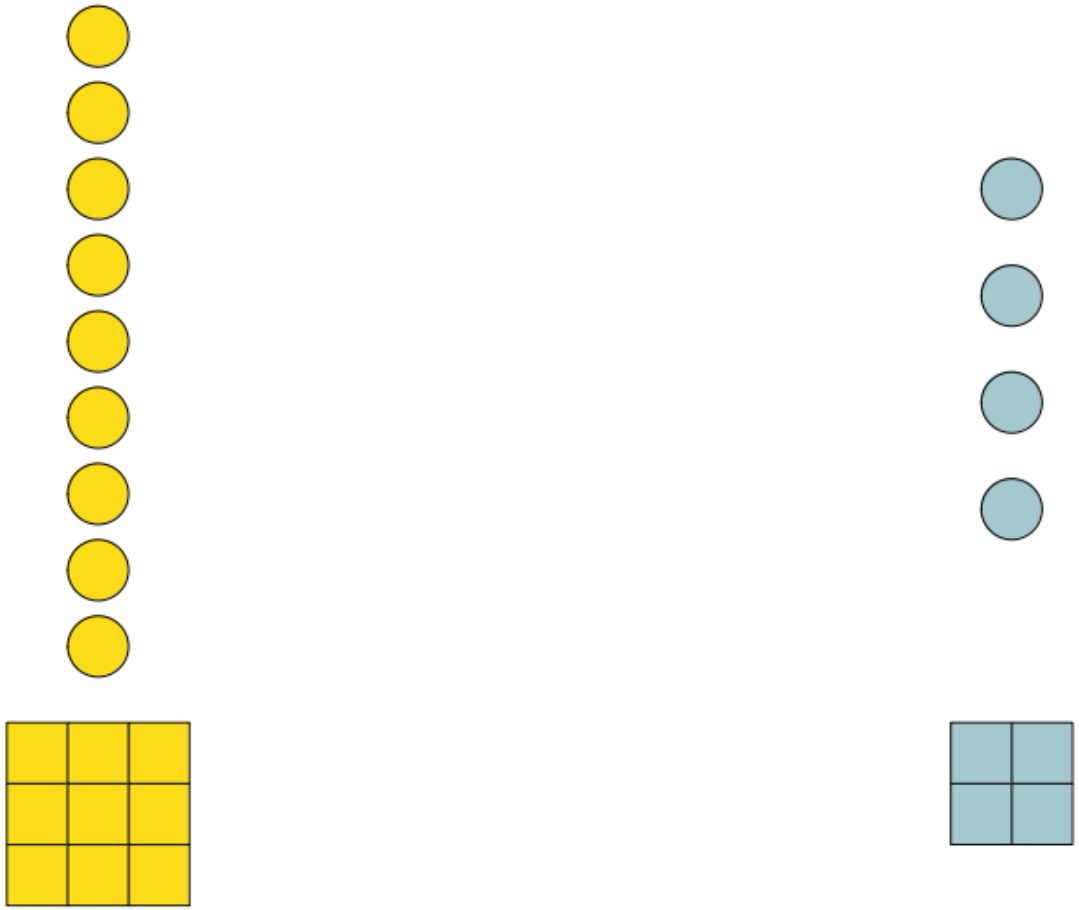
- And in total four times.

PARAMETER SHARING



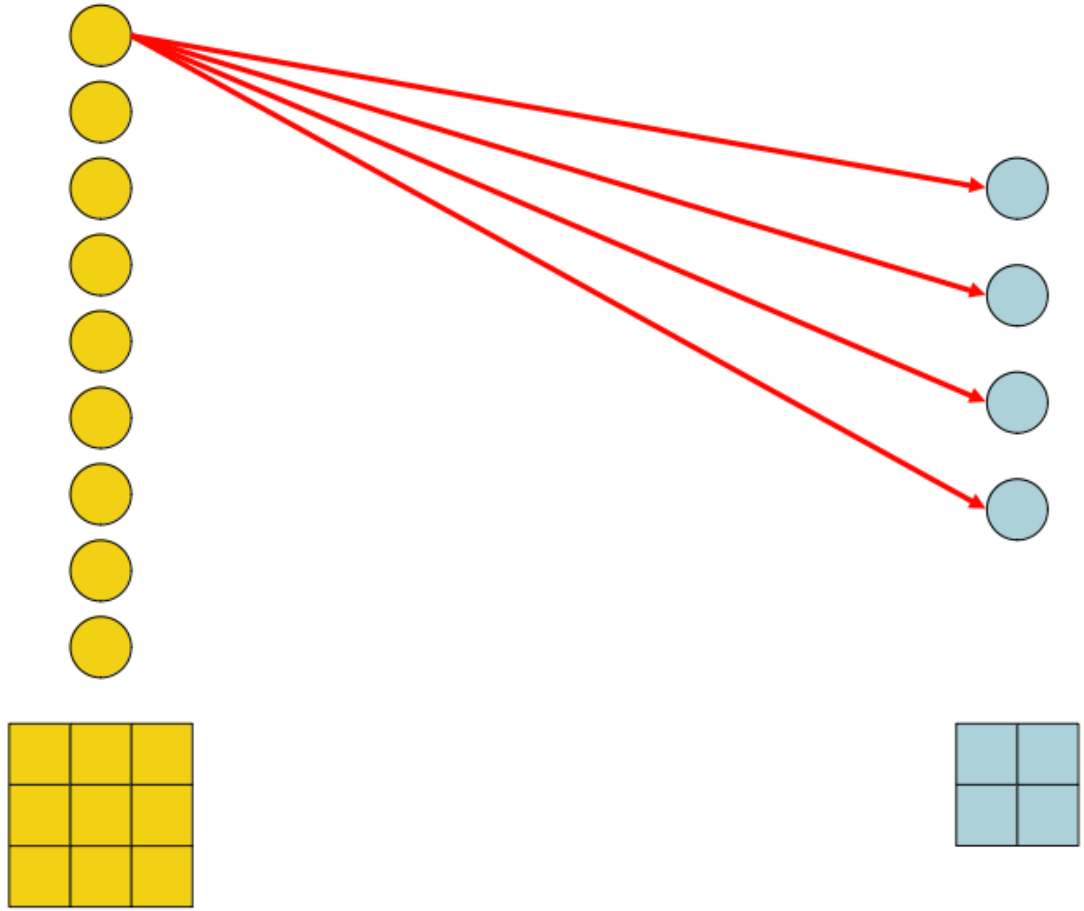
- All together, we have just used four weights.

PARAMETER SHARING



- How many weights does a corresponding dense net use?

PARAMETER SHARING



- $9 \cdot 4 = 36$! That is 9 times more weights!

SPARSE CONNECTIONS AND PARAMETER SHARING

- Why is that good?
- Less parameters drastically reduce memory requirements.
- Faster runtime:
  - For m inputs and n outputs, a fully connected layer requires $m \times n$ parameters and has $\mathcal{O}(m \times n)$ runtime.
  - A convolutional layer has limited connections $k << m$, thus only $k \times n$ parameters and $\mathcal{O}(k \times n)$ runtime.
- Less parameters mean less overfitting and better generalization!

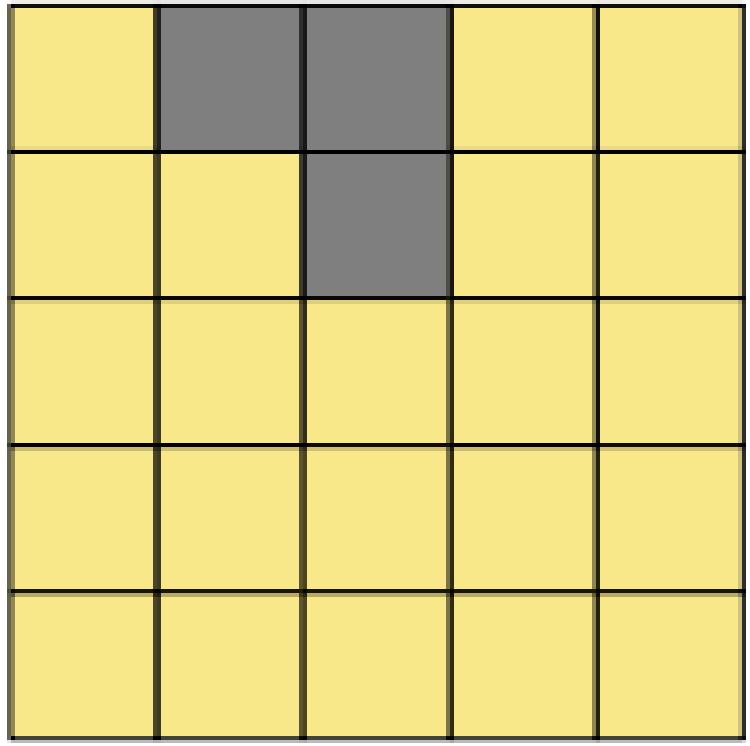
SPARSE CONNECTIONS AND PARAMETER SHARING

- Example: consider a color image with size 100×100.
- Suppose we would like to create one single feature map with a "same padding" (i.e. the hidden layer is of the same size).
  - Choosing a filter with size 5 means that we have a total of $5 \cdot 5 \cdot 3 = 75$ parameters (bias unconsidered).
  - A dense net with the same amount of "neurons" in the hidden layer results in

$$\underbrace{(100^2 \cdot 3)}_{\text{input}} \cdot \underbrace{(100^2)}_{\text{hidden layer}} = 300.000.000$$

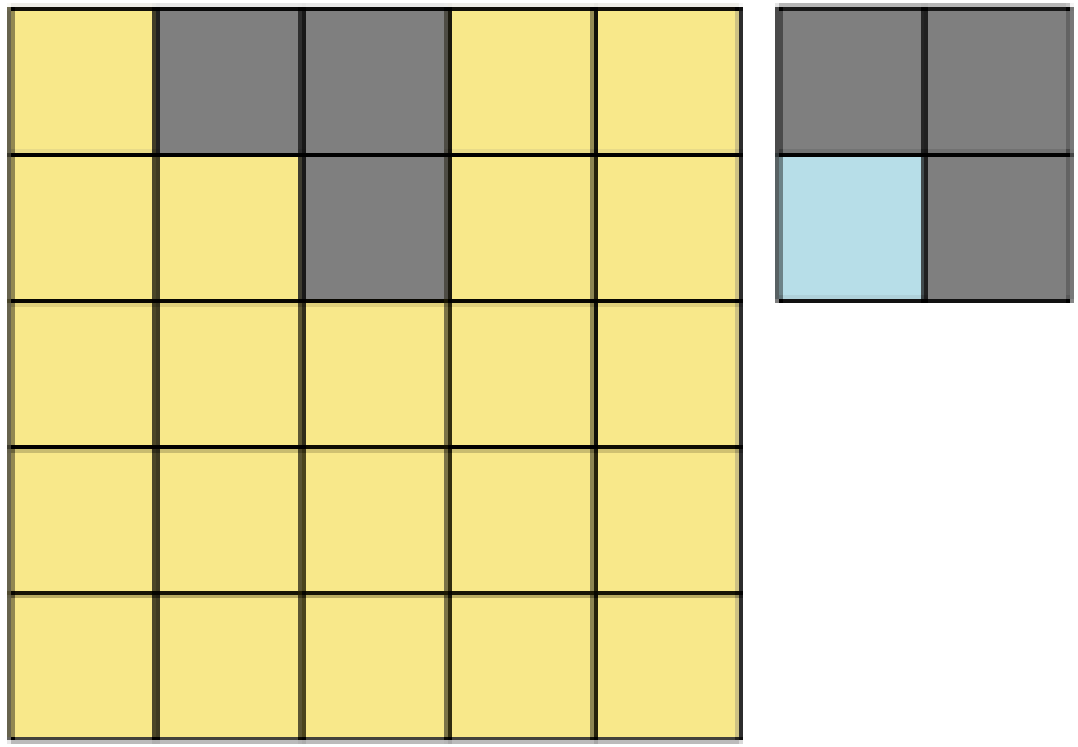
    parameters.
- Note that this was just a fictitious example. In practice we normally do not try to replicate CNN architectures with dense networks.

EQUIVARIANCE TO TRANSLATION



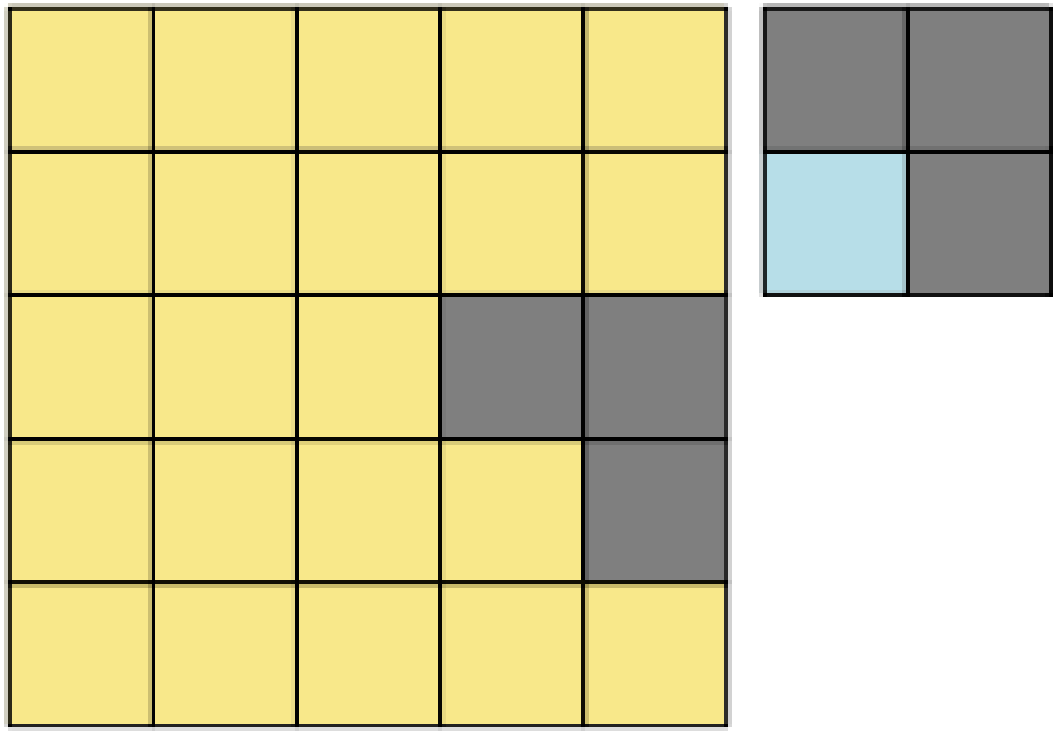
- Think of a specific feature of interest, here highlighted in grey.

EQUIVARIANCE TO TRANSLATION



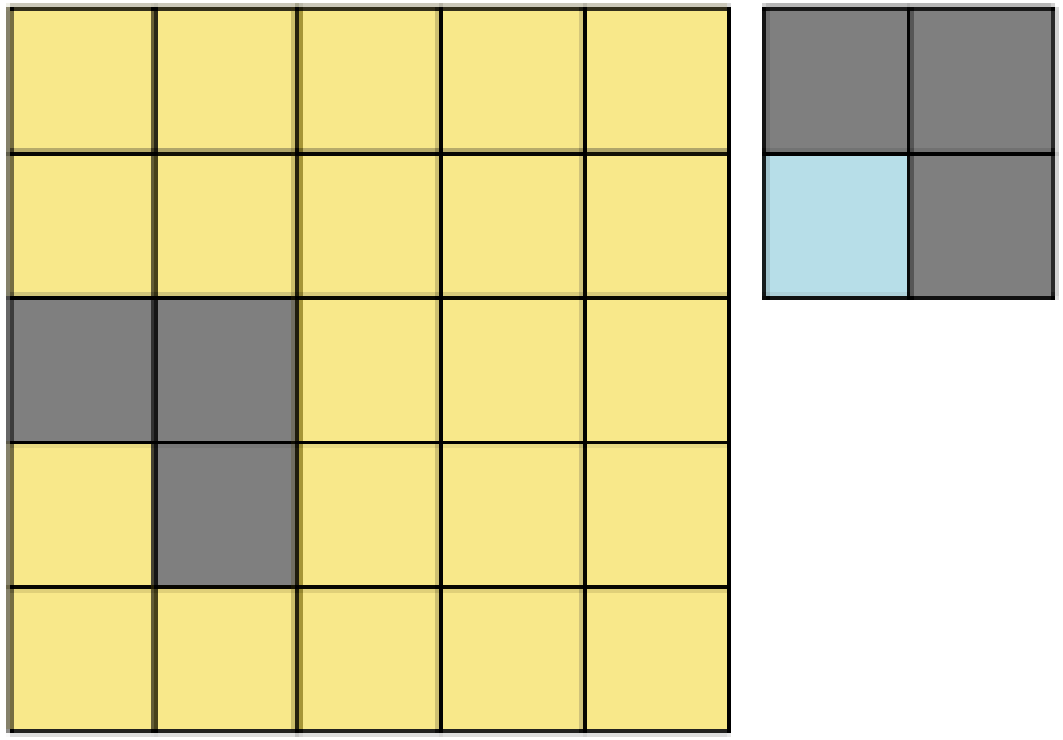
- Furthermore, assume we had a tuned filter looking for exactly that feature.

EQUIVARIANCE TO TRANSLATION



- The filter does not care at what location the feature of interest is located at.

EQUIVARIANCE TO TRANSLATION



- It is literally able to find it anywhere! That property is called **equivariance to translation**.

  Note: A function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$.

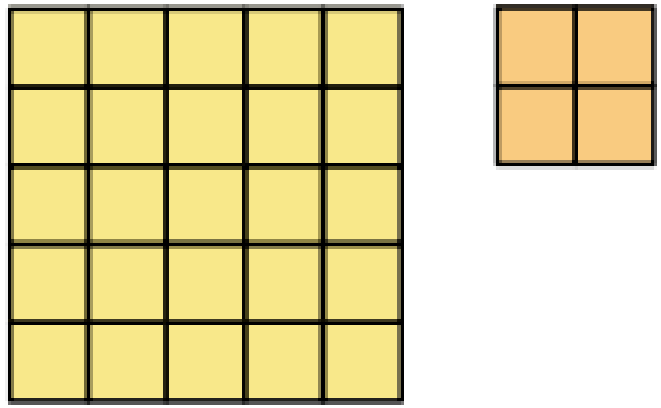
NONLINEARITY IN FEATURE MAPS

- As in dense nets, we use activation functions on all feature map entries to introduce nonlinearity in the net.
- Typically rectified linear units (ReLU) are used in CNNs:
  - They reduce the danger of saturating gradients compared to sigmoid activations.
  - They can lead to *sparse activations*, as neurons ≤ 0 are squashed to 0 which increases computational speed.
- As seen in the last chapter, many variants of ReLU (Leaky ReLU, ELU, PReLU, etc.) exist.

Components of Convolution Layers

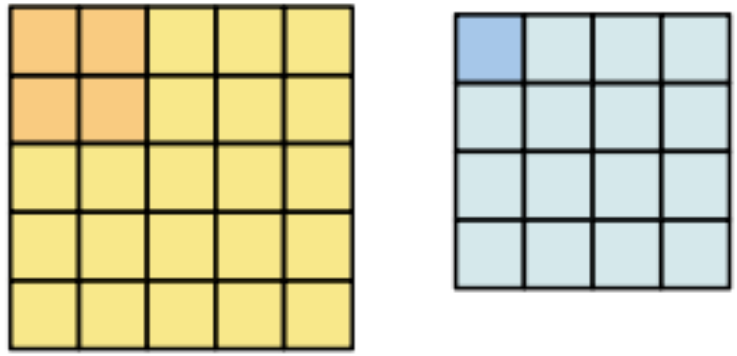
VALID PADDING

Suppose we have an input of size 5×5 and a filter of size 2×2.



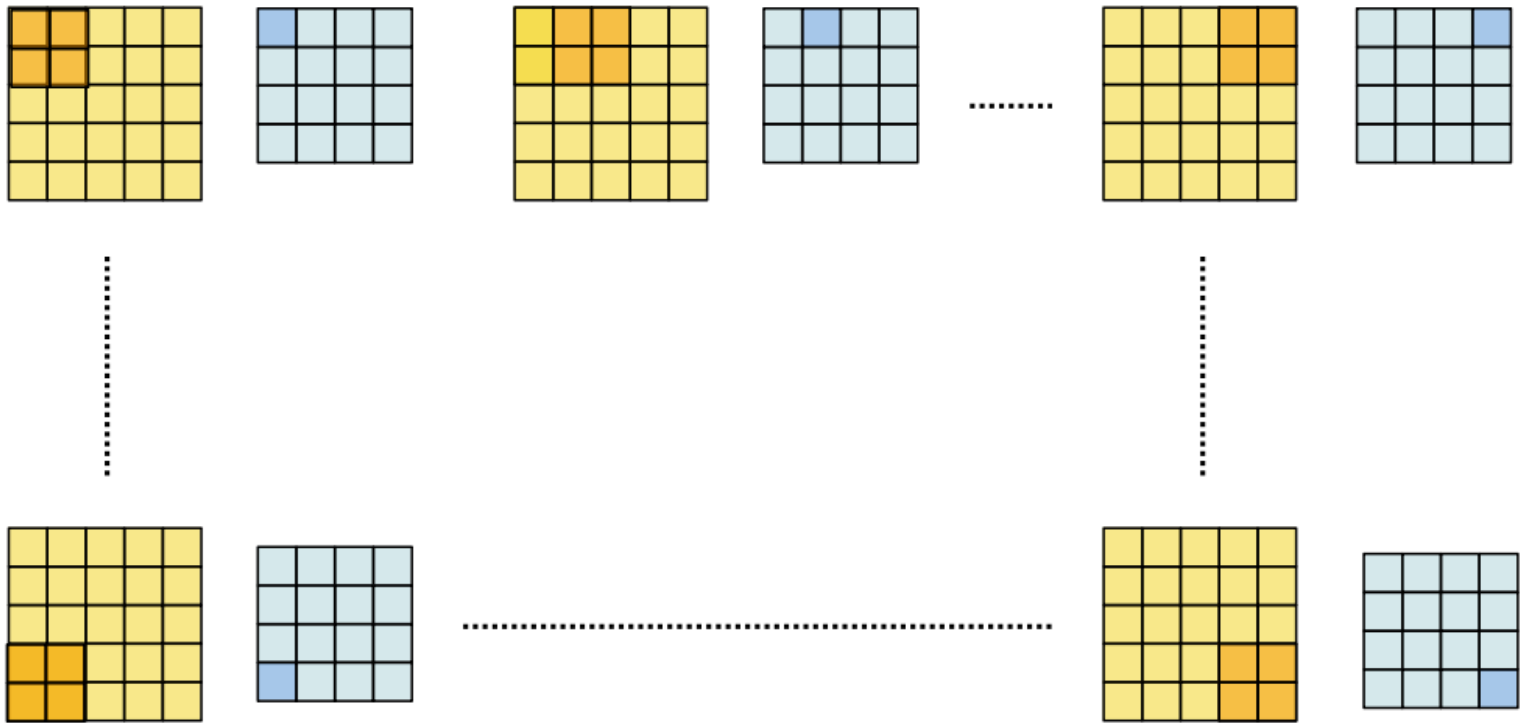
VALID PADDING

The filter is only allowed to move inside of the input space.



VALID PADDING

That will inevitably reduce the output dimensions.

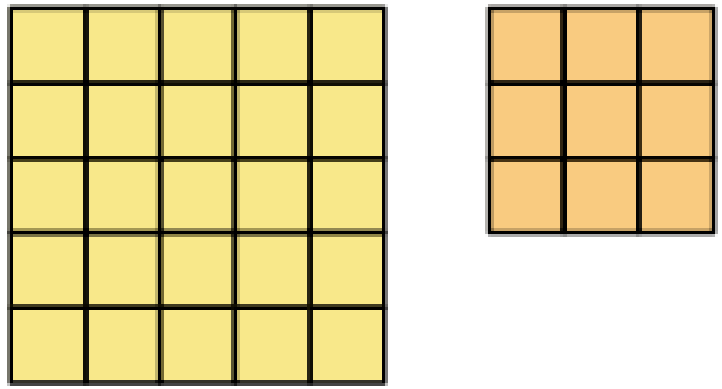


In general, for an input of size i ($\times i$) and filter size k ($\times k$), the size of the output feature map o ($\times o$) claculated by:

$$o = i - k + 1$$

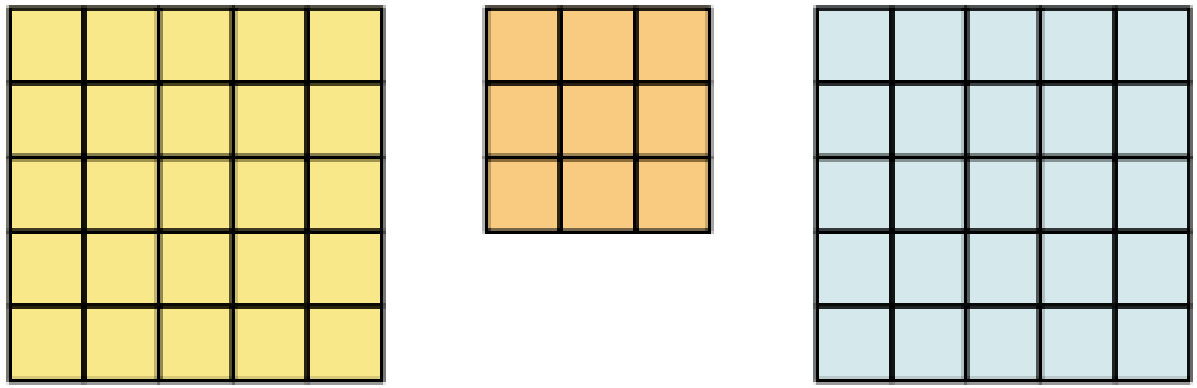
SAME PADDING

Suppose the following situation: an input with dimensions 5×5 and a filter with size 3×3.



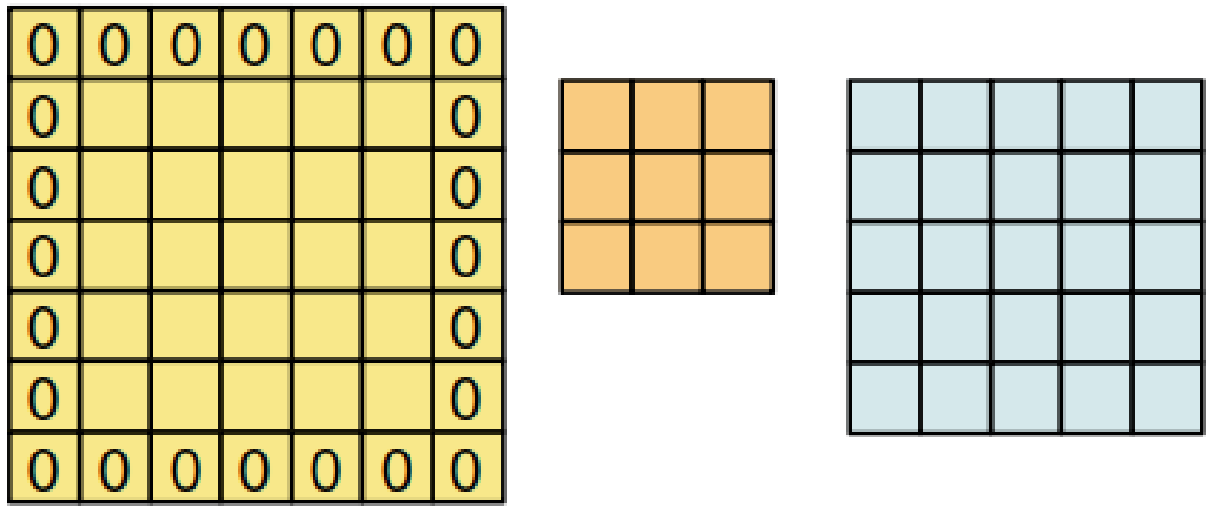
SAME PADDING

We would like to obtain an output with the same dimensions as the input.



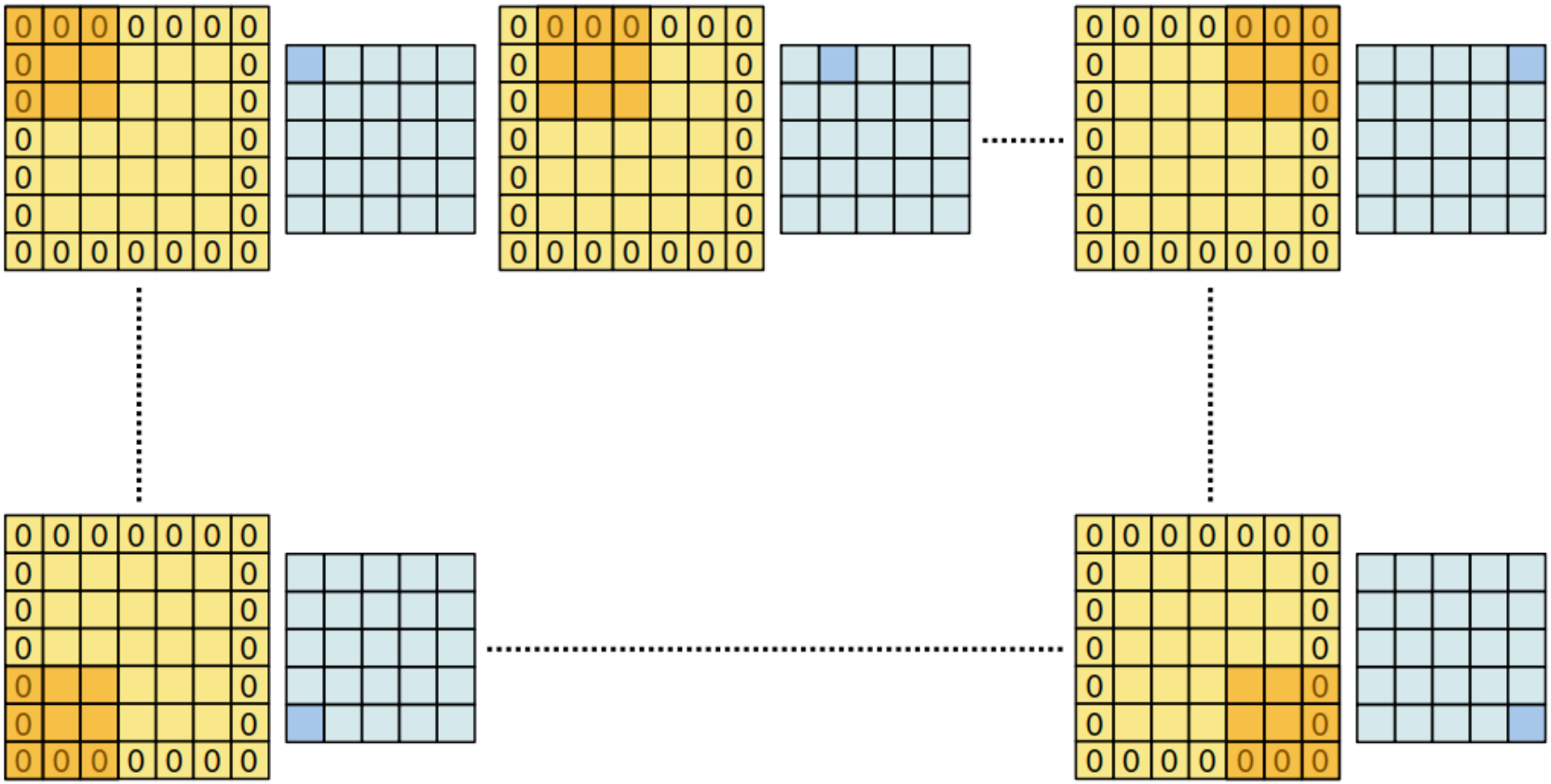
SAME PADDING

Hence, we apply a technique called zero padding. That is to say “pad” zeros around the input:



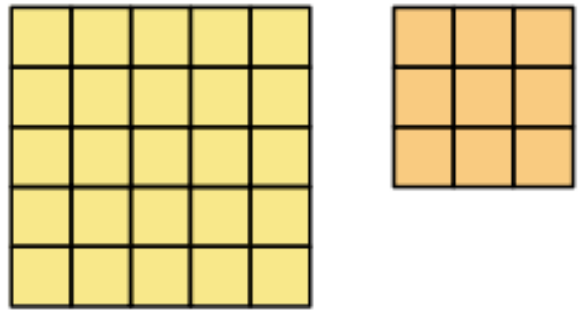
SAME PADDING

That always works! We just have to adjust the zeros according to the input dimensions and filter size (ie. one, two or more rows).



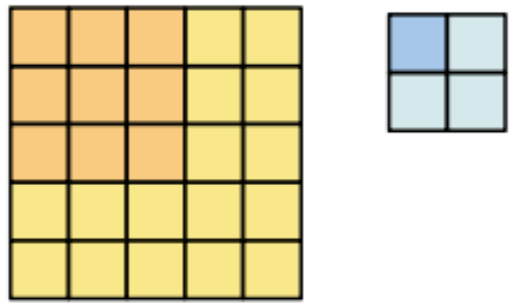
STRIDES

- Stepsize “strides” of our filter (stride = 2 shown below).



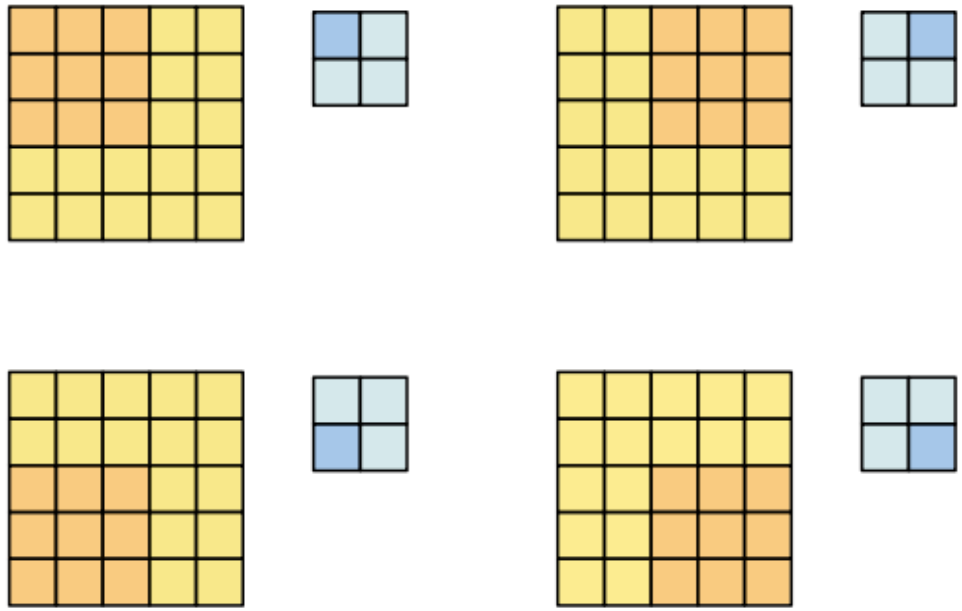
STRIDES

- Stepsize “strides” of our filter (stride = 2 shown below).



STRIDES

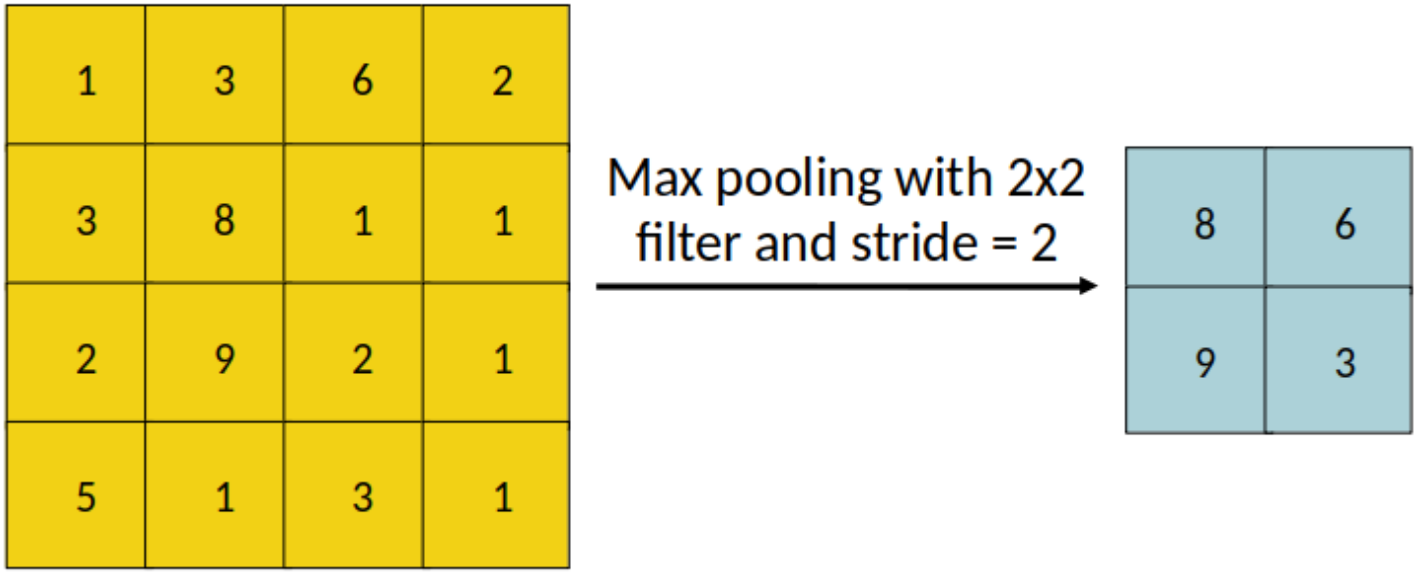
- Stepsize “strides” of our filter (stride = 2 shown below).



In general, when there is no padding, for an input of size i, filter size k and stride s, the size o of the output feature map is:

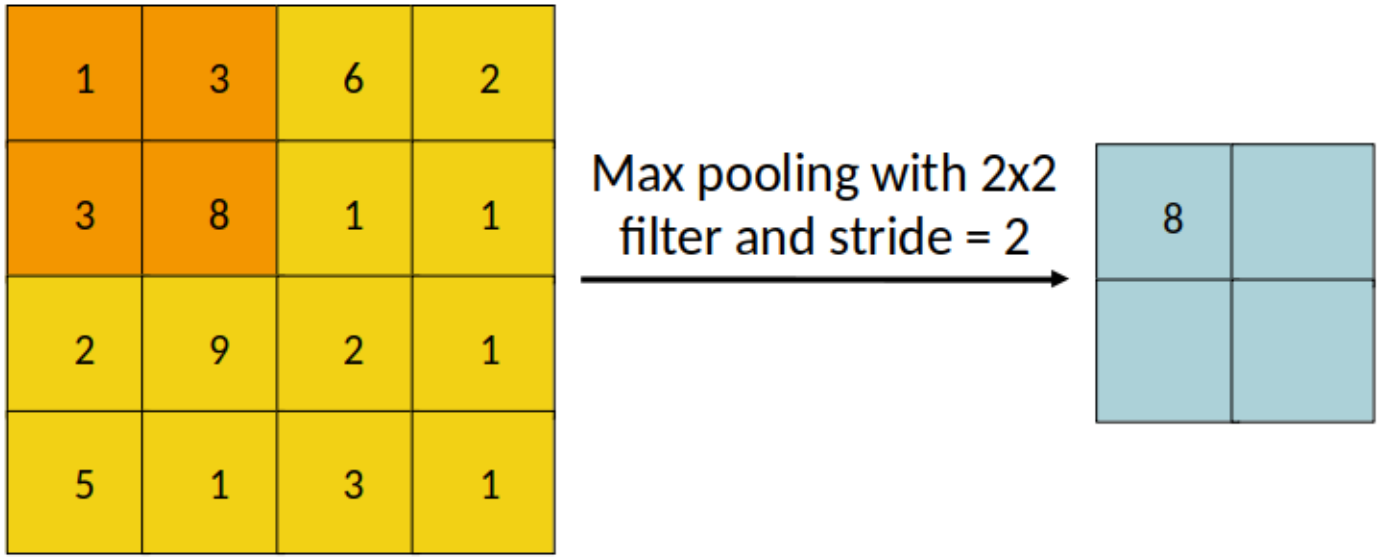
$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1$$

MAX POOLING



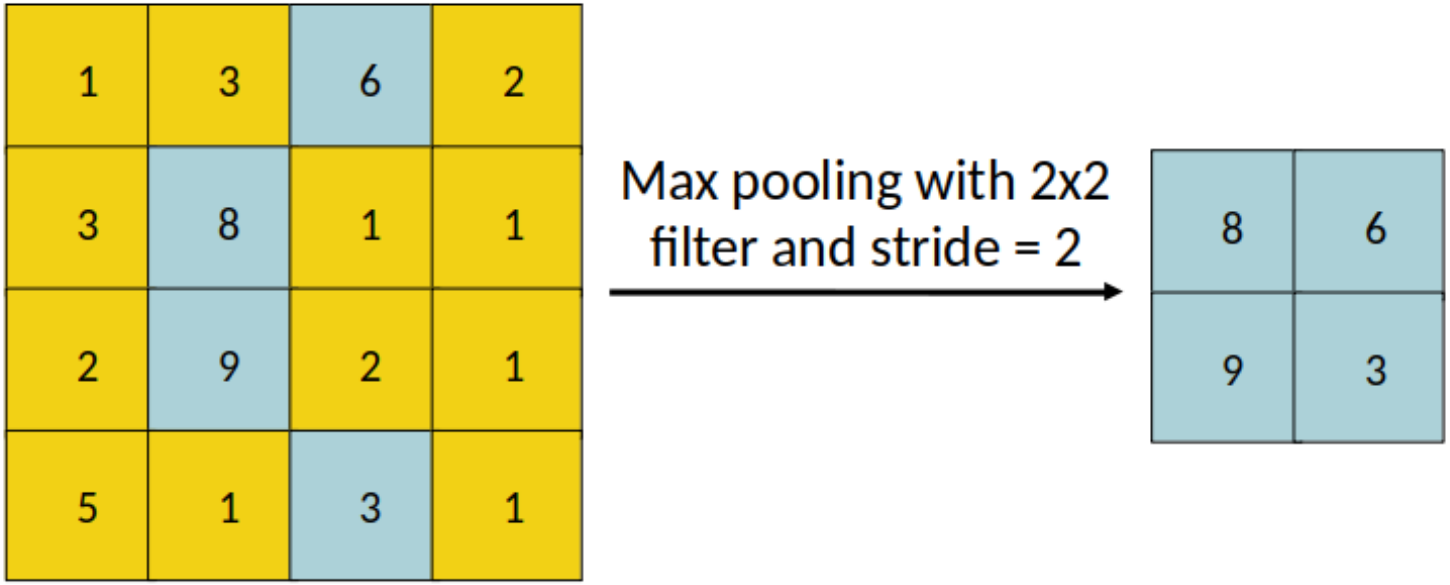
- We've seen how convolutions work, but there is one other operation we need to understand.
- We want to downsample the feature map but optimally lose no information.

MAX POOLING



- Applying the max pooling operation, we simply look for the maximum value at each spatial location.
- That is 8 for the first location.
- Due to the filter of size 2 we have the dimensions of the original feature map and obtain downsampling.

MAX POOLING



- The final pooled feature map has entries 8, 6, 9 and 3.
- Max pooling brings us 2 properties: 1) dimention reduction and 2) spatial invariance.
- Popular pooling functions: max and (weighted) average.

INTRODUCTION TO DEEP LEARNING

Introduction

MOTIVATION FOR RECURRENT NETWORKS

- The two types of neural network architectures that we've seen so far are fully-connected networks and CNNs.
- Their input layers have a fixed size and (typically) only handle fixed-length inputs.
- The primary reason: if we vary the size of the input layer, we would also have to vary the number of learnable weights in the network.
- This in particular relates to **sequence data** such as time-series, audio and text.
- **Recurrent Neural Networks (RNNs)** is a class of architectures that allows varying input lengths and properly accounts for the ordering in sequence data.

SOME MORE SOPHISTICATED APPLICATIONS

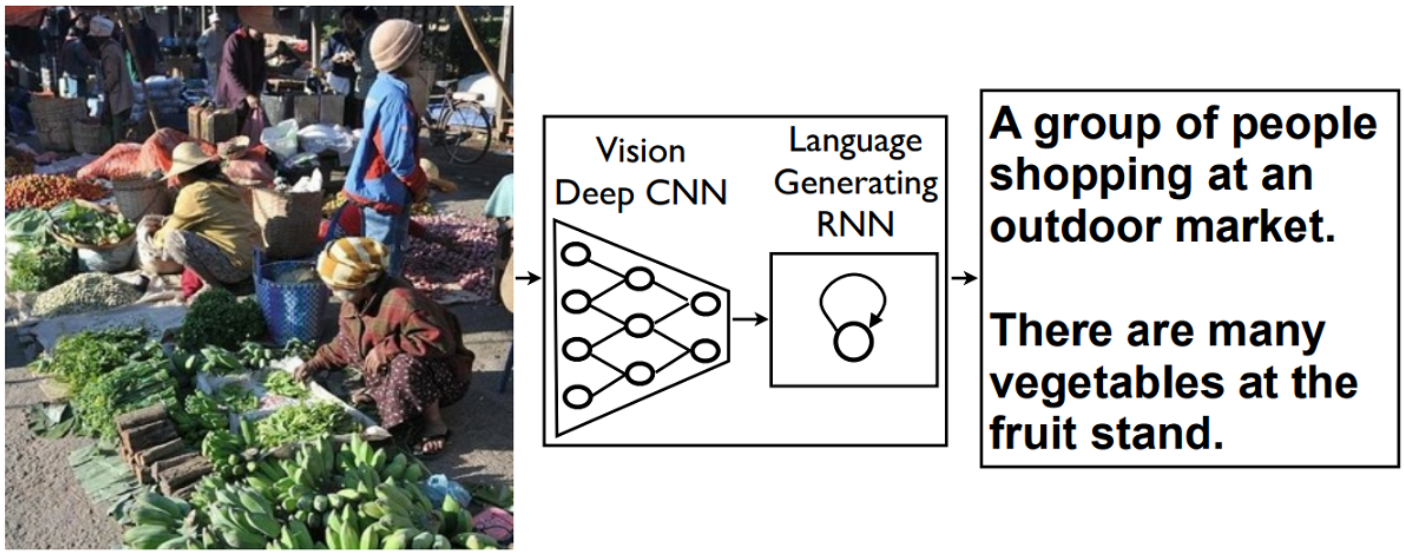


Figure: Show and Tell: A Neural Image Caption Generator (Oriol Vinyals et al. 2014). A language generating RNN tries to describe in brief the content of different images.

SOME MORE SOPHISTICATED APPLICATIONS

Figure: Show and Tell: A Neural Image Caption Generator (Oriol Vinyals et al. 2014). A language generating RNN tries to describe in brief the content of different images.

SOME MORE SOPHISTICATED APPLICATIONS

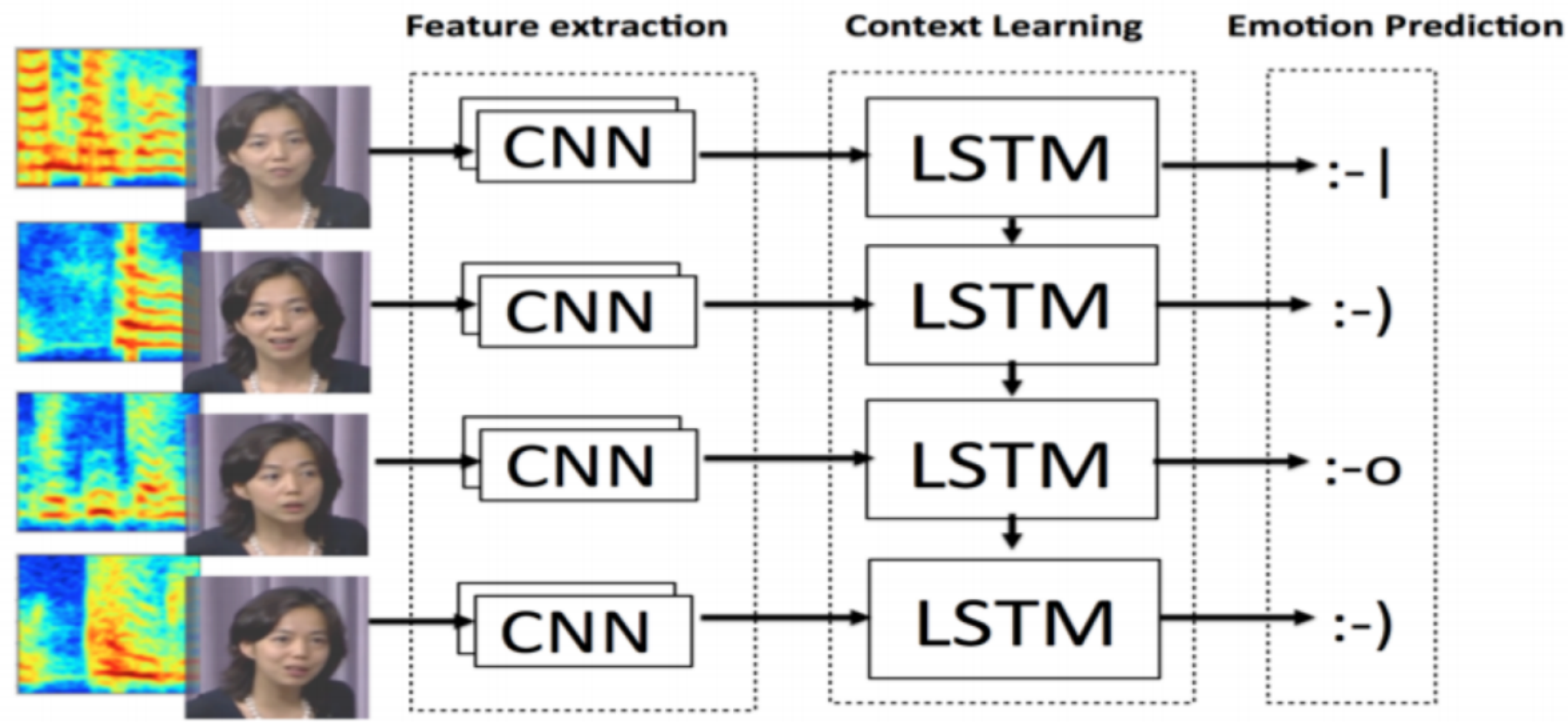


Figure: Convolutional and recurrent nets for detecting emotion from audio data (Namrata Anand & Prateek Verma, 2016). We already had this example in the CNN chapter!

SOME MORE SOPHISTICATED APPLICATIONS

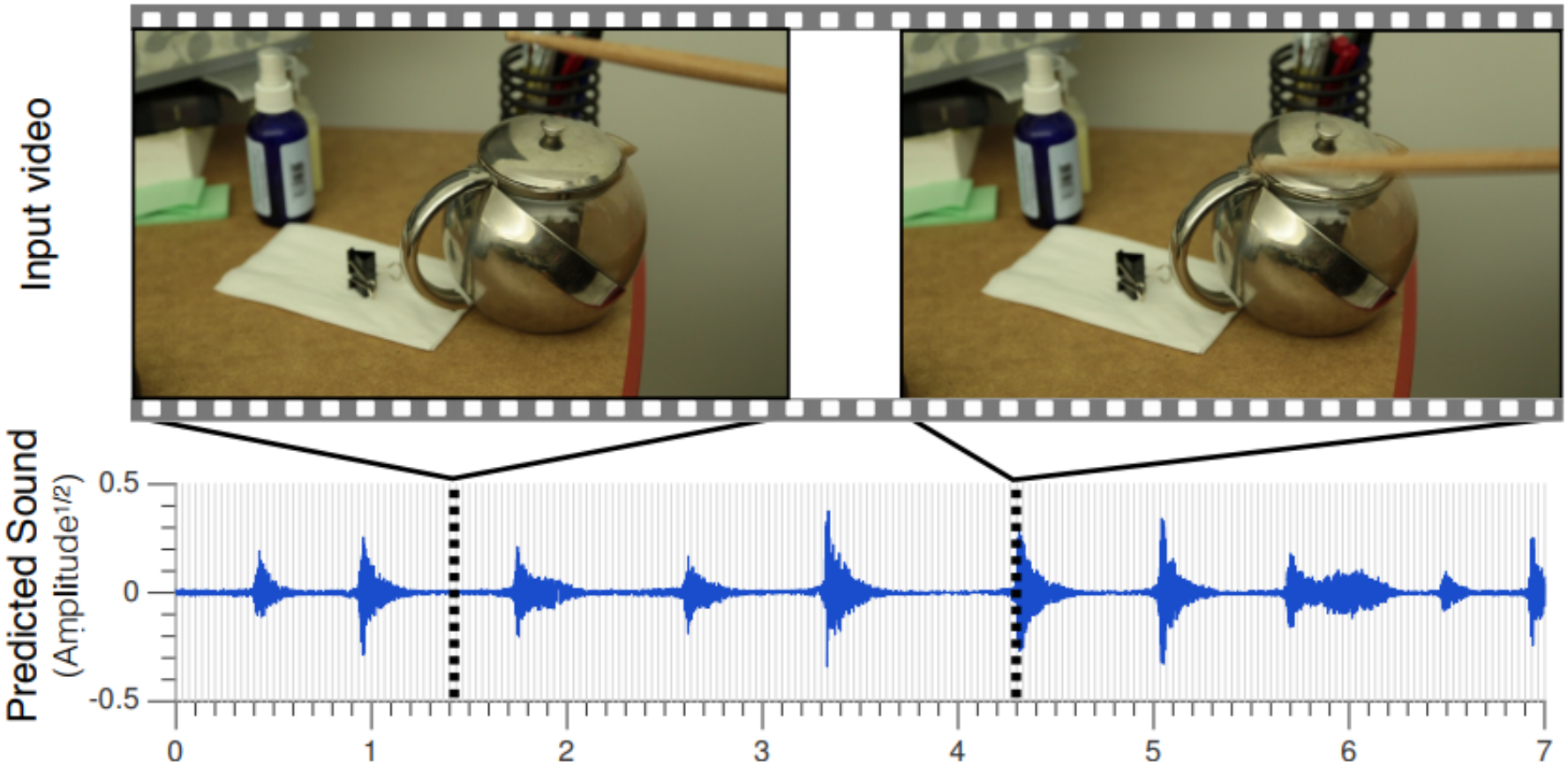


Figure: Visually Indicated Sounds (Andrew Owens et al. 2016). A model to synthesize plausible impact sounds from silent videos. Click here

RNNS - INTRODUCTION

- Suppose we have some text data and our task is to analyse the *sentiment* in the text.
- For example, given an input sentence, such as "This is good news.", the network has to classify it as either 'positive' or 'negative'.
- We would like to train a simple neural network (such as the one below) to perform the task.

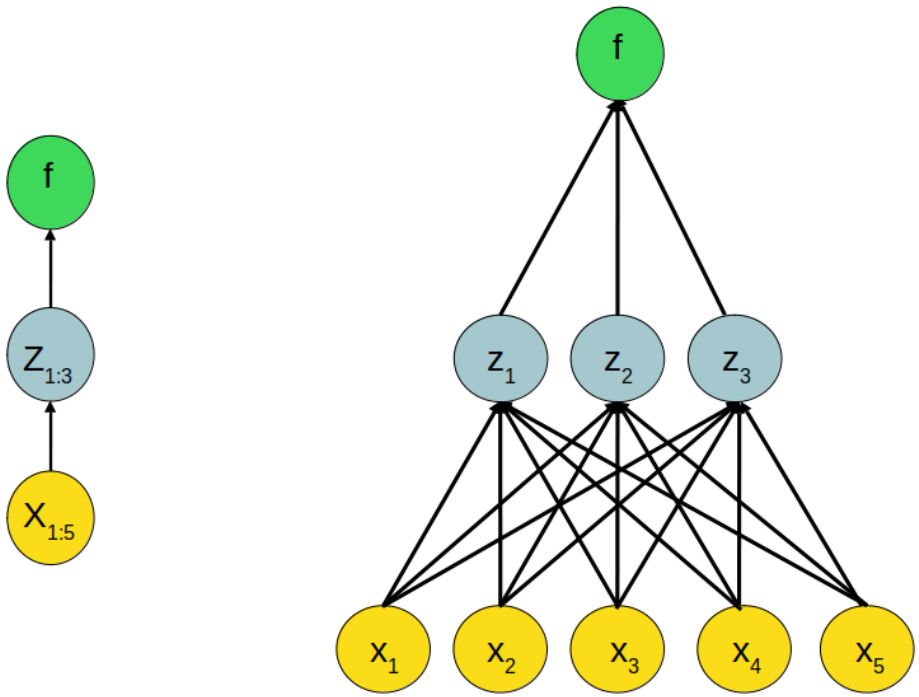


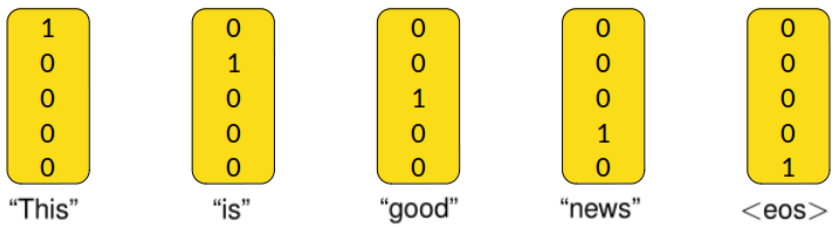
Figure: Two equivalent visualizations of a dense net with a single hidden layer, where the left is more abstract showing the network on a layer point-of-view

RNNS - INTRODUCTION

- Because sentences can be of varying lengths, we need to modify the dense net architecture to handle such a scenario.
- One approach is to draw inspiration from the way a human reads a sentence; that is, one word at a time.
- An important cognitive mechanism that makes this possible is "**short-term memory**".
- As we read a sentence from beginning to end, we retain some information about the words that we have already read and use this information to understand the meaning of the entire sentence.
- Therefore, in order to feed the words in a sentence sequentially to a neural network, we need to give it the ability to retain some information about past inputs.

RNNS - INTRODUCTION

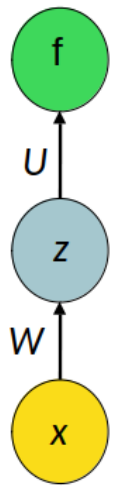
- When words in a sentence are fed to the network one at a time, the inputs are no longer independent. It is much more likely that the word "good" is followed by "morning" rather than "plastic". Hence, we also need to model this **(long-term) dependency**.
- Each word must still be encoded as a fixed-length vector because the size of the input layer will remain fixed.
- Here, for the sake of the visualization, each word is represented as a 'one-hot coded' vector of length 5. (<eos> = 'end of sequence')

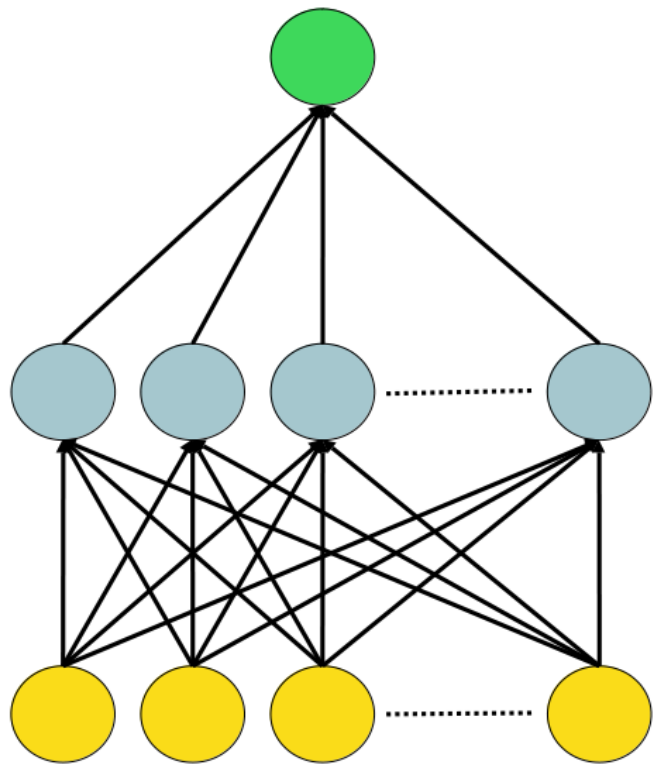


While this is one option to represent words in a network, the standard approach are word embeddings (more on this later).

RNNS - INTRODUCTION

- Our goal is to feed the words to the network sequentially in discrete time-steps.
- A regular dense neural network with a single hidden layer only has two sets of weights: 'input-to-hidden' weights **W** and 'hidden-to-output' weights **U**.





RNNS - INTRODUCTION

- In order to enable the network to retain information about past inputs, we introduce an **additional set of weights V**, from the hidden neurons at time-step t to the hidden neurons at time-step $t + 1$.
- Having this additional set of weights makes the activations of the hidden layer depend on **both** the current input and the activations for the *previous* input.

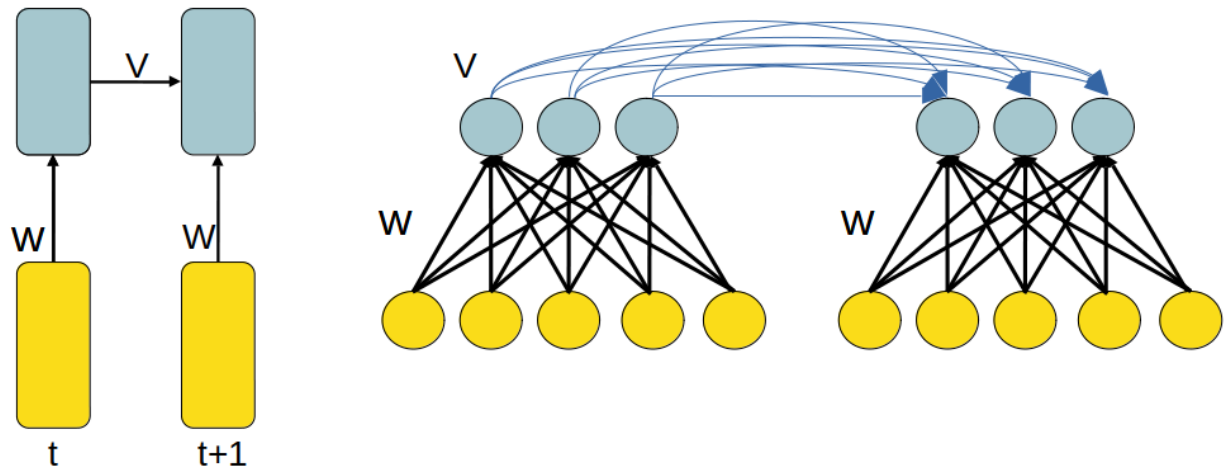


Figure: Input-to-hidden weights **W** and **hidden-to-hidden** weights **V**. The hidden-to-output weights **U** are not shown for better readability.

RNNS - INTRODUCTION

- With this additional set of hidden-to-hidden weights $\mathbf{V}$, the network is now a Recurrent Neural Network (RNN).
- In a regular feed-forward network, the activations of the hidden layer are only computed using the input-hidden weights $\mathbf{W}$ (and bias $\mathbf{b}$).

$$\mathbf{z} = \sigma(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$$

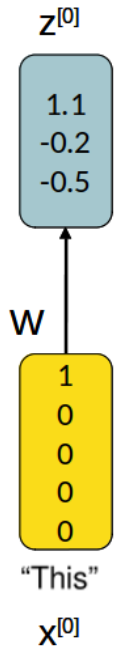
- In an RNN, the activations of the hidden layer (at time-step t) are computed using *both* the input-to-hidden weights W and the hidden-to-hidden weights $\mathbf{V}$.

$$\mathbf{z}^{[t]} = \sigma(\mathbf{V}^\top \mathbf{z}^{[t-1]} + \mathbf{W}^\top \mathbf{x}^{[t]} + \mathbf{b})$$

- The vector $\mathbf{z}^{[t]}$ represents the short-term memory of the RNN because it is a function of the current input $\mathbf{x}^{[t]}$ and the activations $\mathbf{z}^{[t-1]}$ of the previous time-step.
- Therefore, by recurrence, it contains a "summary" of *all* previous inputs.

APPLICATION EXAMPLE - SENTIMENT ANALYSIS

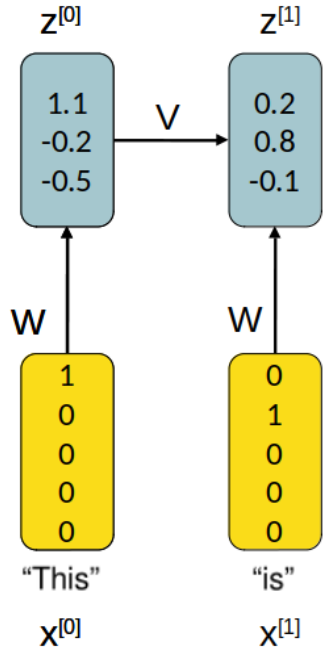
- At $t = 0$, we feed the word "This" to the network and obtain $\mathbf{z}^{[0]}$.
- $\mathbf{z}^{[0]} = \sigma(\mathbf{W}^\top \mathbf{x}^{[0]} + \mathbf{b})$



Because this is the very first input, there is no past state (or, equivalently, the state is initialized to 0).

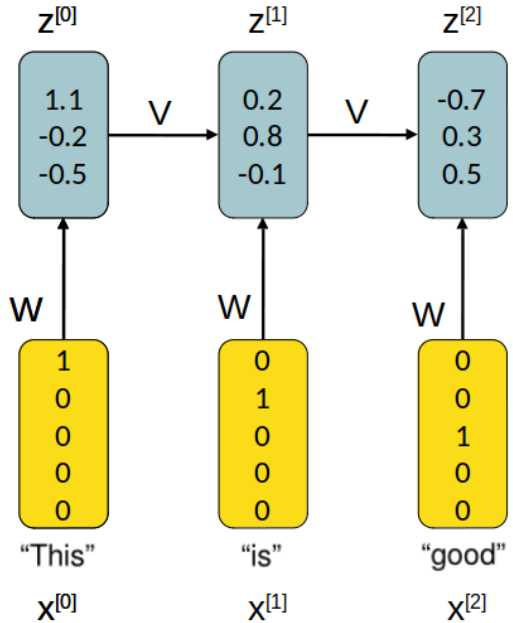
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- At $t = 1$, we feed the second word to the network to obtain $\mathbf{z}^{[1]}$.
- $\mathbf{z}^{[1]} = \sigma(\mathbf{V}^\top \mathbf{z}^{[0]} + \mathbf{W}^\top \mathbf{x}^{[1]} + \mathbf{b})$



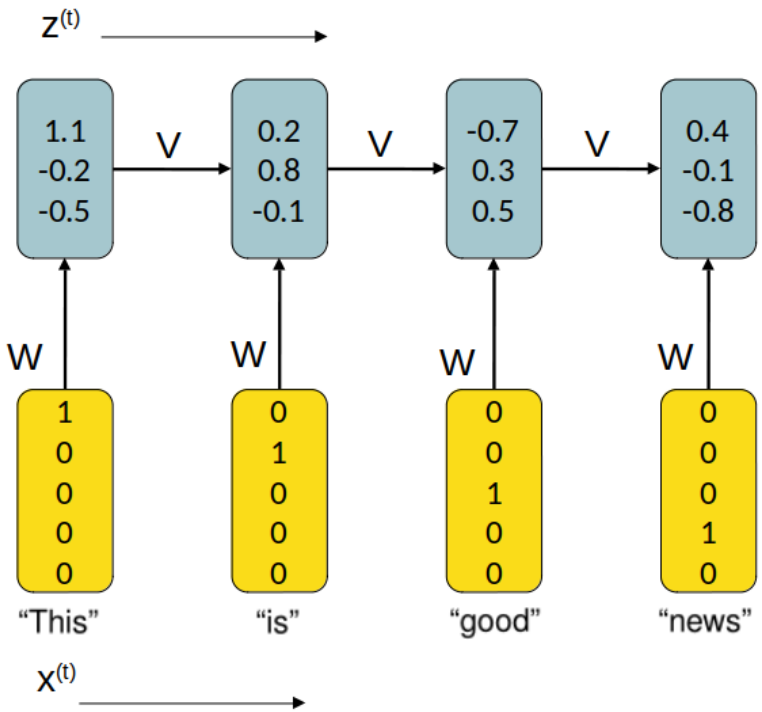
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- At $t = 2$, we feed the next word in the sentence.
- $\mathbf{z}^{[2]} = \sigma(\mathbf{V}^\top \mathbf{z}^{[1]} + \mathbf{W}^\top \mathbf{x}^{[2]} + \mathbf{b})$



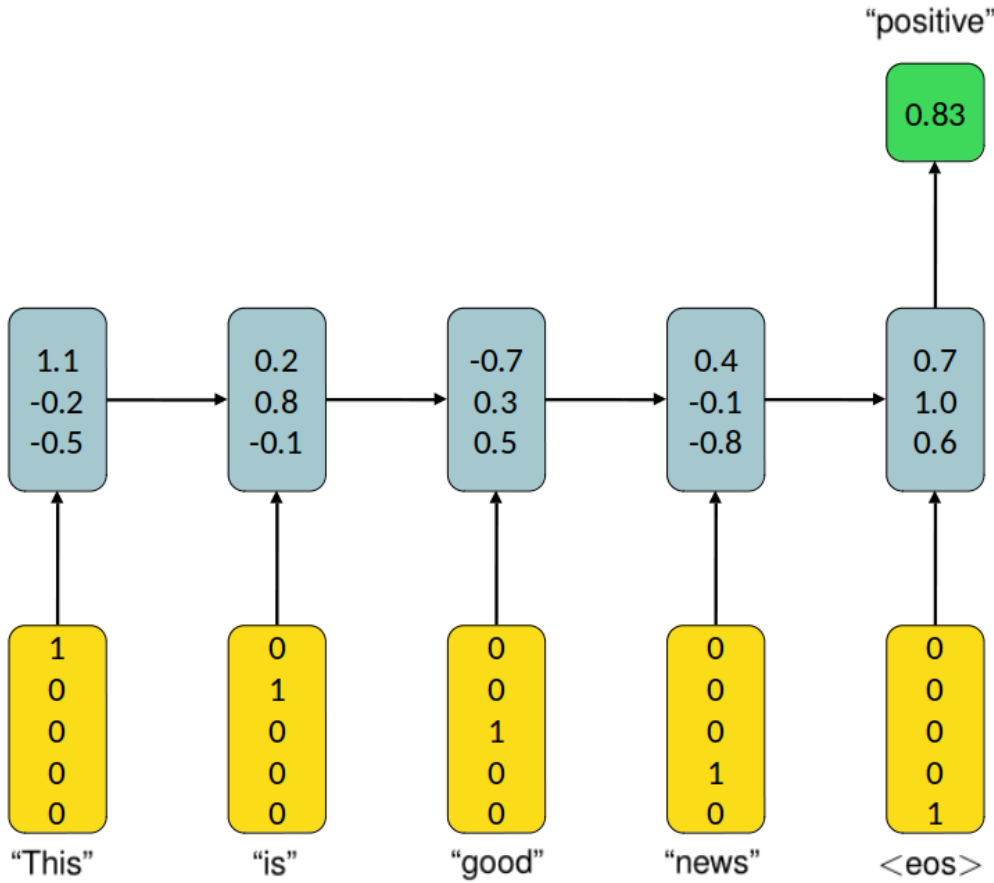
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- At $t = 3$, we feed the next word ("news") in the sentence.
- $\mathbf{z}^{[3]} = \sigma(\mathbf{V}^\top \mathbf{z}^{[2]} + \mathbf{W}^\top \mathbf{x}^{[3]} + \mathbf{b})$



APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- Once the entire input sequence has been processed, the prediction of the network can be generated by feeding the activations of the final time-step to the output neuron(s).
- $f = \sigma(\mathbf{U}^\top \mathbf{z}^{[4]} + c)$, where c is the bias of the output neuron.



PARAMETER SHARING

- This way, the network can process the sentence one word at a time and the length of the network can vary based on the length of the sequence.
- It is important to note that no matter how long the input sequence is, the matrices **W** and **V** are the same in every time-step. This is another example of **parameter sharing**.
- Therefore, the number of weights in the network is independent of the length of the input sequence.

RNNS - USE CASE SPECIFIC ARCHITECTURES

RNNs are very versatile. They can be applied to a wide range of tasks.

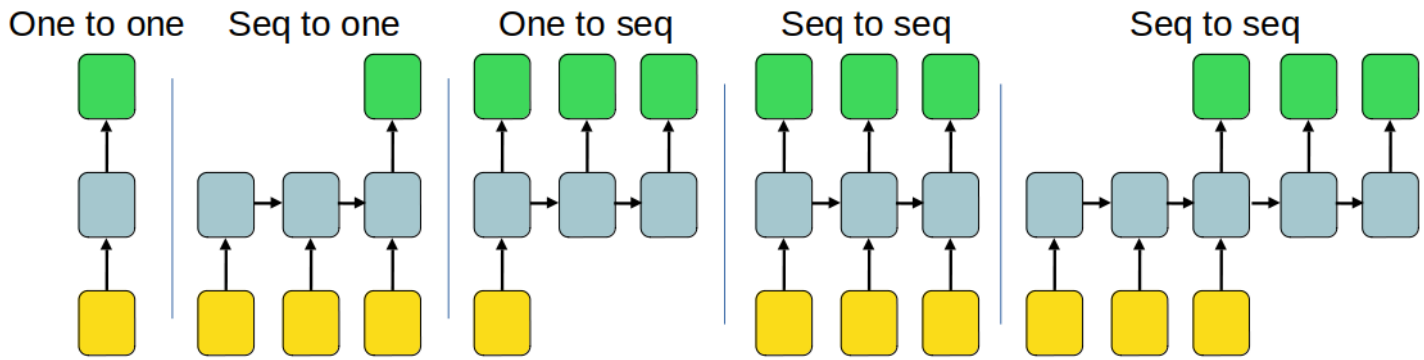


Figure: RNNs can be used in tasks that involve multiple inputs and/or multiple outputs.

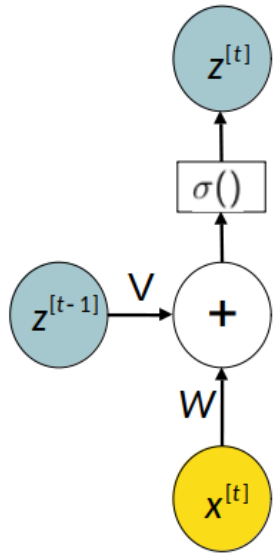
Examples:

- Sequence-to-One: Sentiment analysis, document classification.
- One-to-Sequence: Image captioning.
- Sequence-to-Sequence: Language modelling, machine translation, time-series prediction.

Modern RNNs

LONG SHORT-TERM MEMORY (LSTM)

The LSTM provides a way of dealing with vanishing gradients and modelling long-term dependencies.



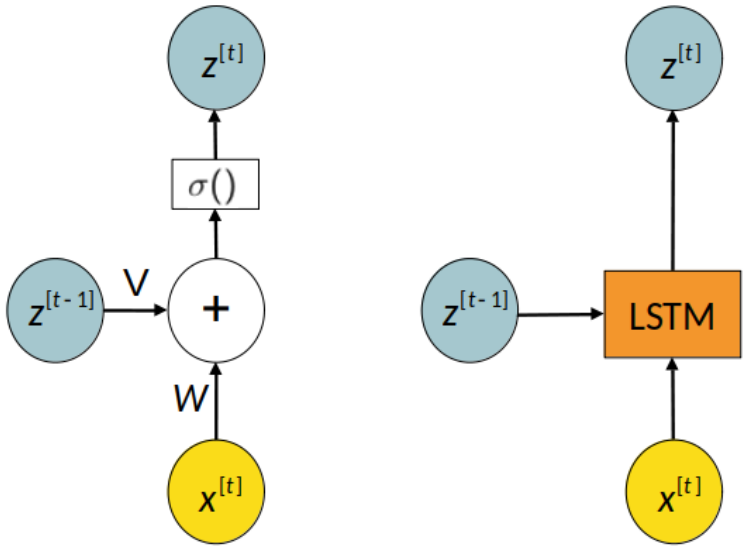
A simple RNN mechanism;

- Until now, we simply computed

$$\mathbf{z}^{[t]} = \sigma(\mathbf{b} + \mathbf{V}^\top \mathbf{z}^{[t-1]} + \mathbf{W}^\top \mathbf{x}^{[t]})$$

LONG SHORT-TERM MEMORY (LSTM)

The LSTM provides a way of dealing with vanishing gradients and modelling long-term dependencies.



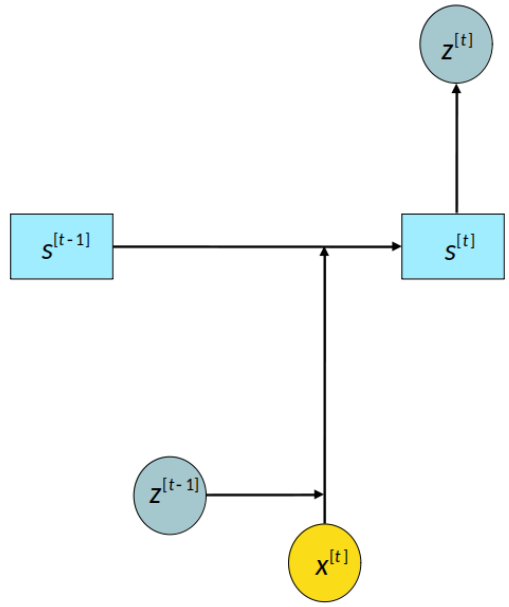
Left: A simple RNN mechanism; Right: An LSTM cell

- Until now, we simply computed

$$\mathbf{z}^{[t]} = \sigma(\mathbf{b} + \mathbf{V}^\top \mathbf{z}^{[t-1]} + \mathbf{W}^\top \mathbf{x}^{[t]})$$

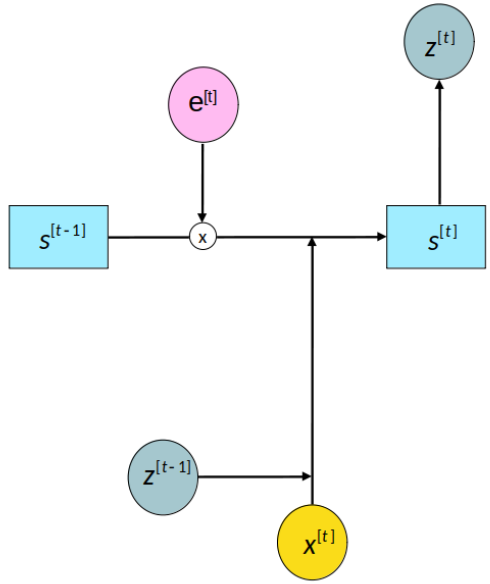
- Now we introduce the LSTM cell, a small network on its own.

LONG SHORT-TERM MEMORY (LSTM)



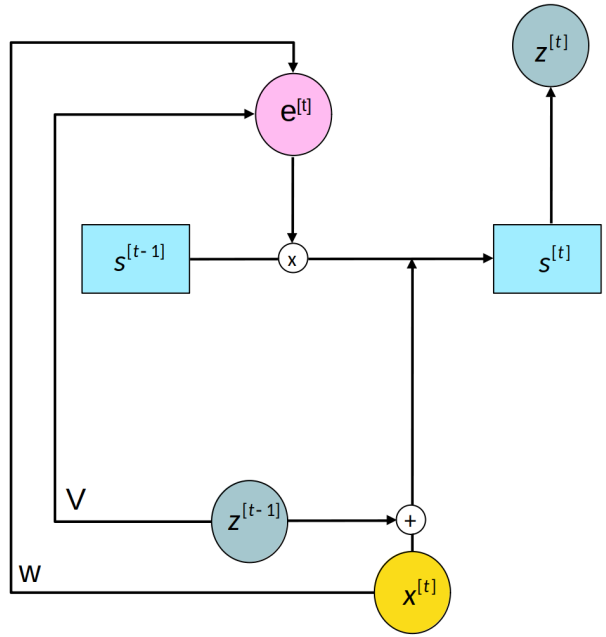
- The key to LSTMs is the **cell state** $\mathbf{s}^{[t]}$.
- $\mathbf{s}^{[t]}$ can be manipulated by different **gates** to forget old information, add new information, and read information out of it.
- Each gate is a vector of the same size as $\mathbf{s}^{[t]}$ with elements between 0 ("let nothing pass") and 1 ("let everything pass").

LONG SHORT-TERM MEMORY (LSTM)



- **Forget gate $\mathbf{e}^{[t]}$**: indicates which information of the old cell state we should forget.
- Intuition: Think of a model trying to predict the next word based on all the previous ones. The cell state might include the gender of the present subject, so that the correct pronouns can be used. When we now see a new subject, we want to forget the gender of the old one.

LONG SHORT-TERM MEMORY (LSTM)

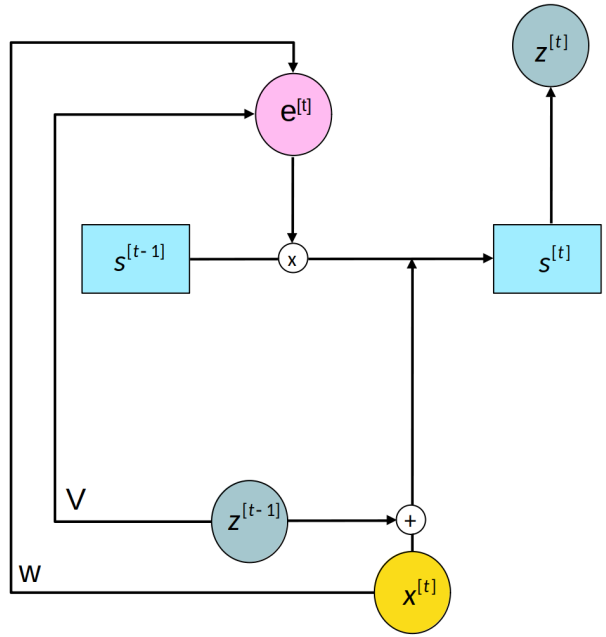


- We obtain the forget gate by computing

$$\mathbf{e}^{[t]} = \sigma(\mathbf{b}_e + \mathbf{V}_e^\top \mathbf{z}^{[t-1]} + \mathbf{W}_e^\top \mathbf{x}^{[t]})$$

- $\sigma()$ is a sigmoid and $\mathbf{V}_e, \mathbf{W}_e$ are forget gate specific weights.

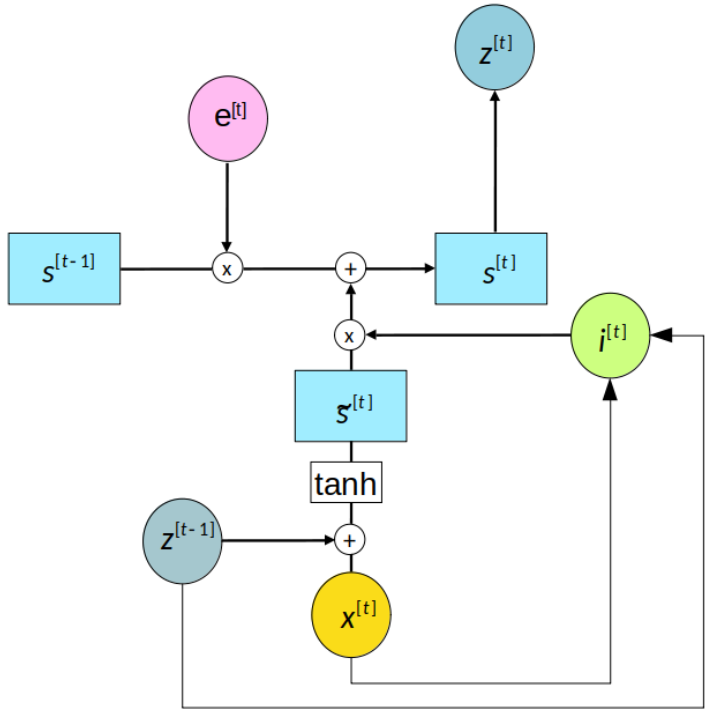
LONG SHORT-TERM MEMORY (LSTM)



- To compute the cell state $\mathbf{s}^{[t]}$, the first step is to multiply (element-wise) the previous cell state $\mathbf{s}^{[t-1]}$ by the forget gate $\mathbf{e}^{[t]}$.

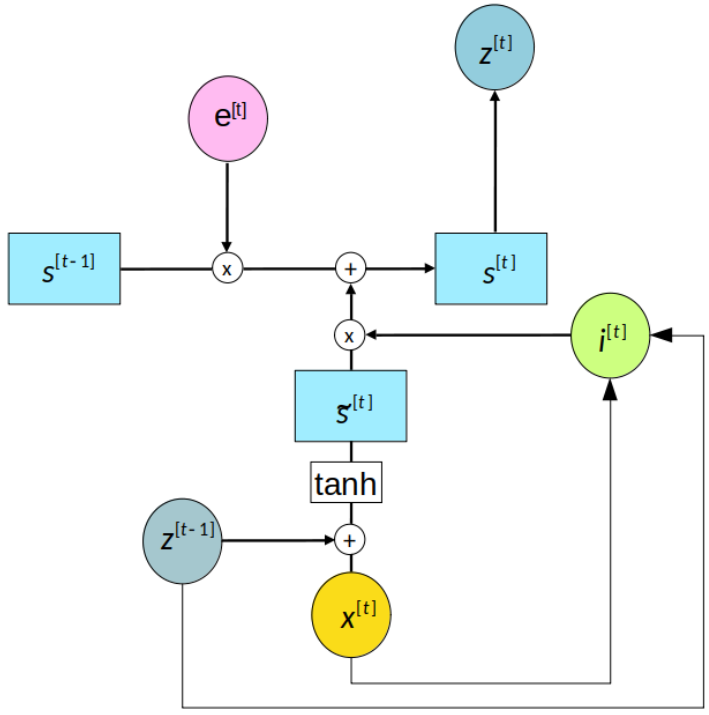
$$\mathbf{e}^{[t]} \odot \mathbf{s}^{[t-1]}, \text{ with } \mathbf{e}^{[t]} \in [0, 1]$$

LONG SHORT-TERM MEMORY (LSTM)



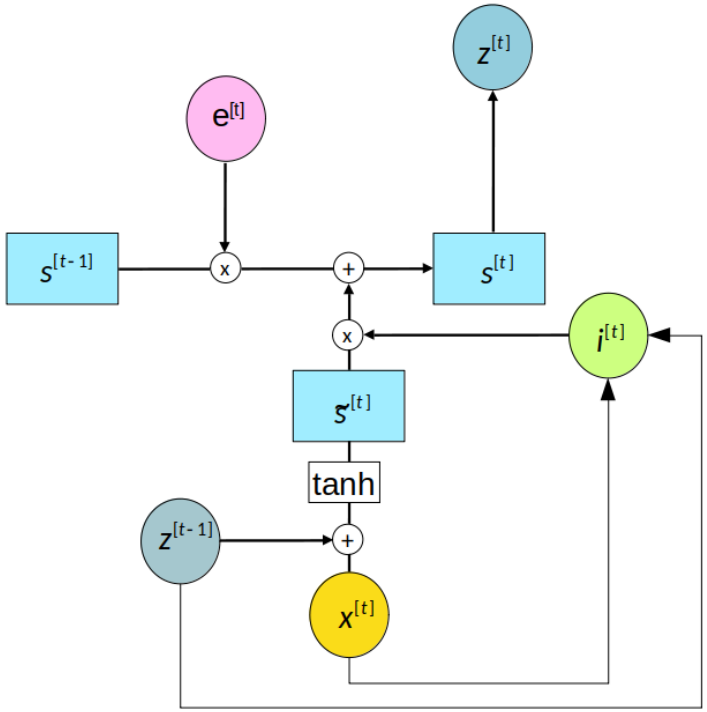
- **Input gate $\mathbf{i}^{[t]}$**: indicates which new information should be added to $\mathbf{s}^{[t]}$.
- Intuition: In our example, this is where we add the new information about the gender of the new subject.

LONG SHORT-TERM MEMORY (LSTM)



- The new information is given by $\tilde{\mathbf{s}}^{[t]} = \tanh(\mathbf{b} + \mathbf{V}^\top \mathbf{z}^{[t-1]} + \mathbf{W}^\top \mathbf{x}^{[t]}) \in [-1, 1]$.
- The input gate is given by $\mathbf{i}^{[t]} = \sigma(\mathbf{b}_i + \mathbf{V}_i^\top \mathbf{z}^{[t-1]} + \mathbf{W}_i^\top \mathbf{x}^{[t]}) \in [0, 1]$.
- $\mathbf{W}$ and $\mathbf{V}$ are weights of the new information, $\mathbf{W}_i$ and $\mathbf{V}_i$ the weights of the input gate.

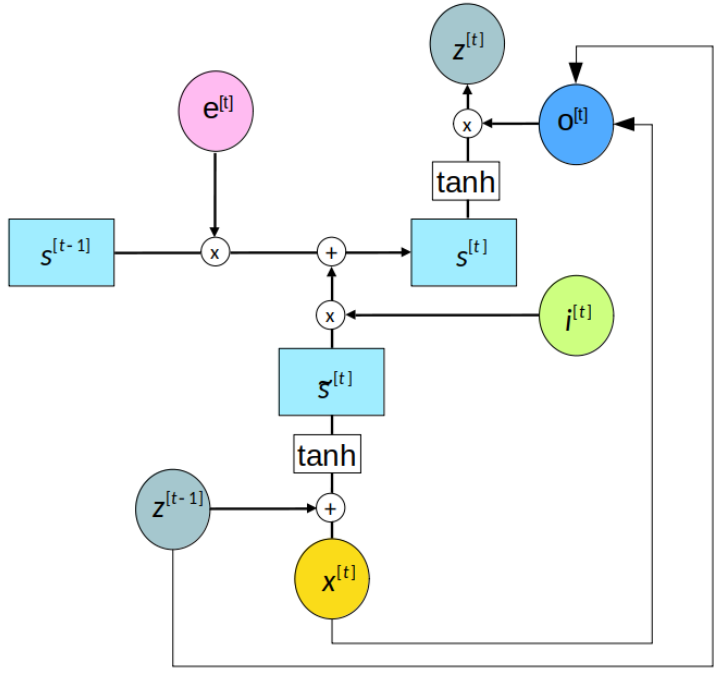
LONG SHORT-TERM MEMORY (LSTM)



- Now we can finally compute the cell state $\mathbf{s}^{[t]}$:

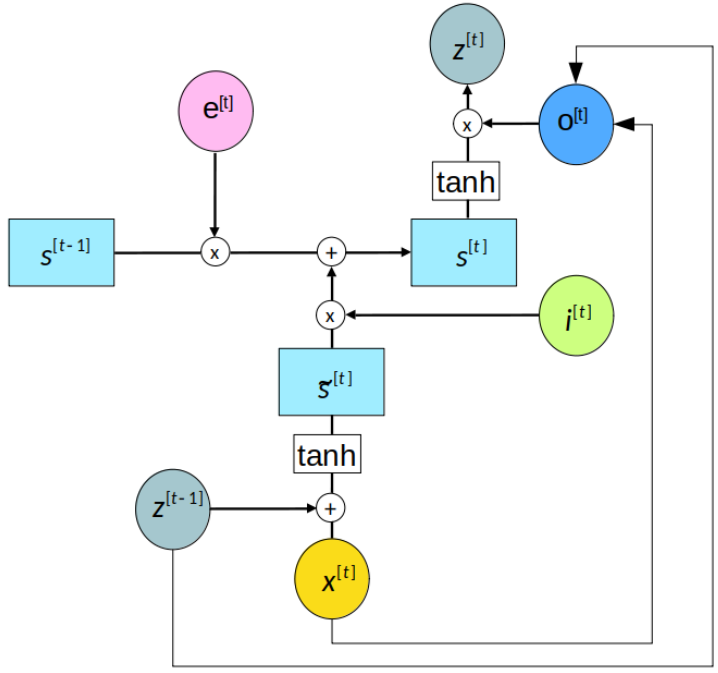
$$\mathbf{s}^{[t]} = \mathbf{e}^{[t]} \odot \mathbf{s}^{[t-1]} + \mathbf{i}^{[t]} \odot \tilde{\mathbf{s}}^{[t]}$$

LONG SHORT-TERM MEMORY (LSTM)



- **Output gate $\mathbf{o}^{[t]}$**: Indicates which information form the cell state is filtered.
- It is given by $\mathbf{o}^{[t]} = \sigma(\mathbf{b}_o + \mathbf{V}_o^\top \mathbf{z}^{[t-1]} + \mathbf{W}_o^\top \mathbf{x}^{[t]})$, with specific weights $\mathbf{W}_o, \mathbf{V}_o$.

LONG SHORT-TERM MEMORY (LSTM)



- Finally, the new state $\mathbf{z}^{[t]}$ of the LSTM is a function of the cell state, multiplied by the output gate:

$$\mathbf{z}^{[t]} = \mathbf{o}^{[t]} \odot \tanh(\mathbf{s}^{[t]})$$

GATED RECURRENT UNITS (GRU)

- The key distinction between regular RNNs and GRUs is that the latter support gating of the hidden state.
- Here, we have dedicated mechanisms for when a hidden state should be updated and also when it should be reset.
- These mechanisms are learned to:
  - avoid the vanishing/exploding gradient problem which comes with a standard recurrent neural network.
  - solve the vanishing gradient problem by using an update gate and a reset gate.
  - control the information that flows into (update gate) and out of (reset gate) memory.

GATED RECURRENT UNITS (GRU)

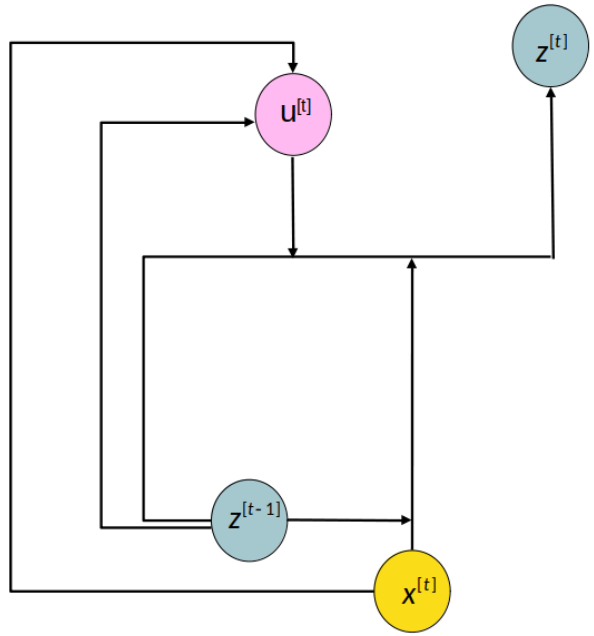


Figure: Update gate in a GRU.

- For a given time step t, the hidden state of the last time step is $\mathbf{z}^{[t-1]}$. The update gate $\mathbf{u}^{[t]}$ is computed as follows:
- $\mathbf{u}^{[t]} = \sigma(\mathbf{W}_u^\top \mathbf{x}^{[t]} + \mathbf{V}_u^\top \mathbf{z}^{[t-1]} + \mathbf{b}_u)$
- We use a sigmoid to transform input values to $(0, 1)$.

GATED RECURRENT UNITS (GRU)

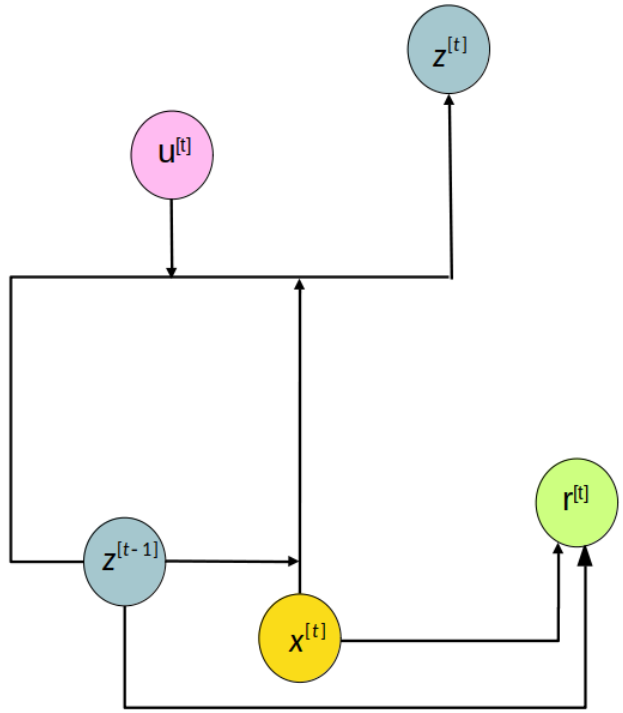


Figure: Reset gate in a GRU.

- Similarly, the reset gate $\mathbf{r}^{[t]}$ is computed as follows:
- $\mathbf{r}^{[t]} = \sigma(\mathbf{W}_r^\top \mathbf{x}^{[t]} + \mathbf{V}_r^\top \mathbf{z}^{[t-1]} + \mathbf{b}_r)$

GATED RECURRENT UNITS (GRU)

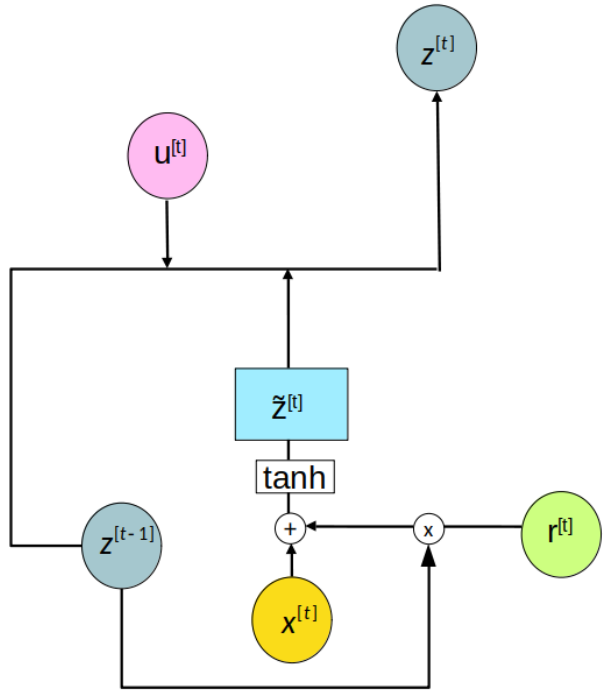


Figure: Hidden state computation in GRU. Multiplication is carried out elementwise.

- $\tilde{\mathbf{z}}^{[t]} = \tanh(\mathbf{W}_z^\top \mathbf{x}^{[t]} + \mathbf{V}_z^\top \left(\mathbf{r}^{[t]} \odot \mathbf{z}^{[t-1]}\right) + \mathbf{b}_z)$.
- In a conventional RNN, we would have an hidden state update of the form: $\mathbf{z}^{[t]} = \tanh(\mathbf{W}_z^\top \mathbf{x}^{[t]} + \mathbf{V}_z^\top \mathbf{z}^{[t-1]} + \mathbf{b}_z)$.

GATED RECURRENT UNITS (GRU)

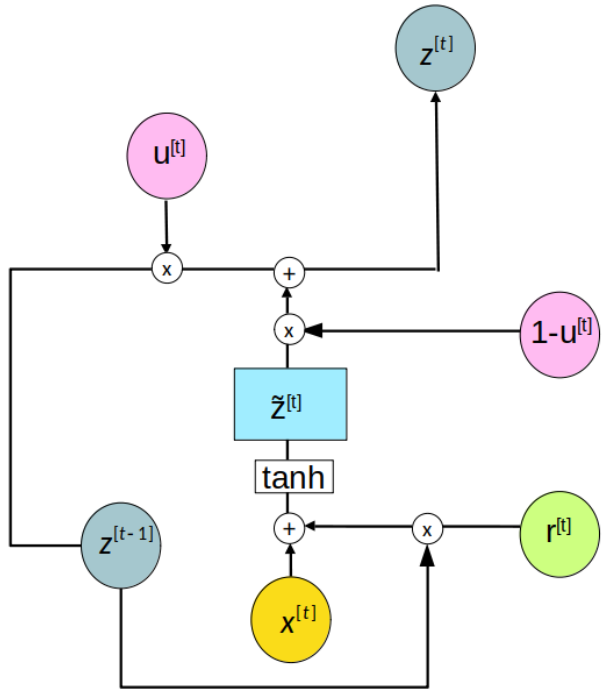


Figure: Update gate in a GRU. The multiplication is carried out elementwise.

- The update gate $\mathbf{u}^{[t]}$ determines how much the old state $\mathbf{z}^{[t-1]}$ and the new candidate state $\tilde{\mathbf{z}}^{[t]}$ is used.
- $\mathbf{z}^{[t]} = \mathbf{u}^{[t]} \odot \mathbf{z}^{[t-1]} + (1 - \mathbf{u}^{[t]}) \odot \tilde{\mathbf{z}}^{[t]}$.

GATED RECURRENT UNITS (GRU)

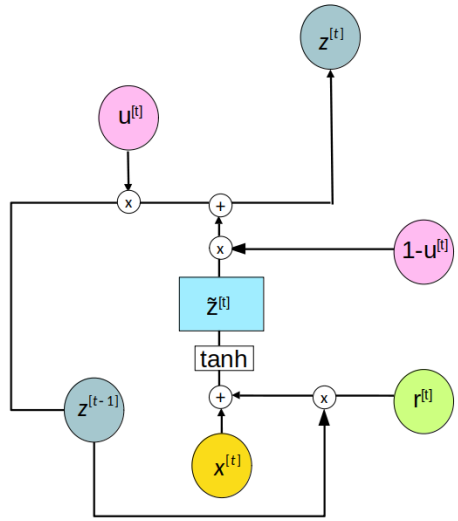


Figure: GRU

These designs can help us to eleminate the vanishing gradient problem in RNNs and capture better dependencies for time series with large time step distances. In summary, GRUs have the following two distinguishing features:

- Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series.

GRU VS LSTM

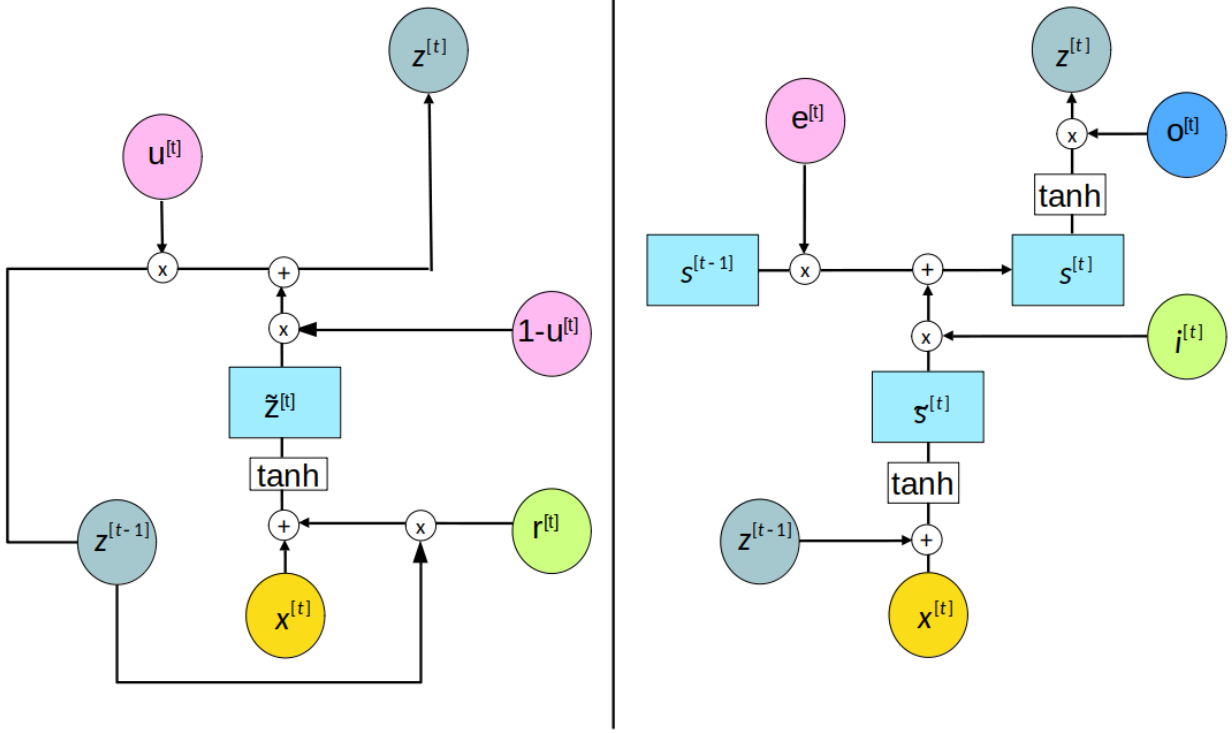


Figure: GRU vs LSTM

BIDIRECTIONAL RNNS

- Another generalization of the simple RNN are bidirectional RNNs.
- These allow us to process sequential data depending on both past and future inputs, e.g. an application predicting missing words, which probably depend on both preceding and following words.
- One RNN processes inputs in the forward direction from $\mathbf{x}^{[1]}$ to $\mathbf{x}^{[T]}$ computing a sequence of hidden states $(\mathbf{z}^{[1]}, \ldots, \mathbf{z}^{(T)})$, another RNN in the backward direction from $\mathbf{x}^{[T]}$ to $\mathbf{x}^{[1]}$ computing hidden states $(\mathbf{g}^{[T]}, \ldots, \mathbf{g}^{[1]})$
- Predictions are then based on both hidden states, which could be **concatenated**.
- With connections going back in time, the whole input sequence must be known in advance to train and infer from the model.
- Bidirectional RNNs are often used for the encoding of a sequence in machine translation.

BIDIRECTIONAL RNNS

Computational graph of a bidirectional RNN:

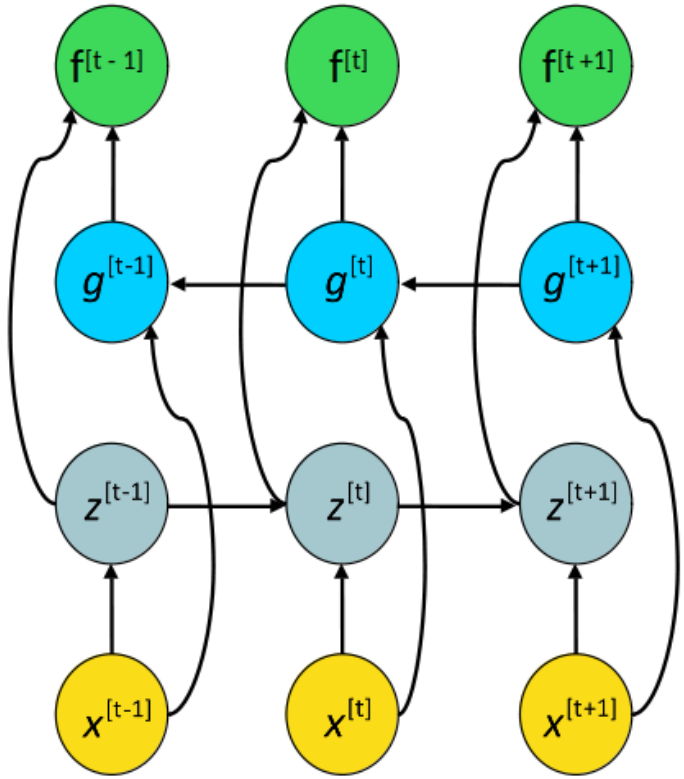


Figure: A bidirectional RNN consists of a forward RNN processing inputs from left to right and a backward RNN processing inputs backwards in time.

Encoder-Decoder Networks and Attention

ENCODER-DECODER NETWORK

- For many interesting applications such as question answering, dialogue systems, or machine translation, the network needs to map an input sequence to an output sequence of different length.
- This is what an encoder-decoder (also called sequence-to-sequence architecture) enables us to do!

ENCODER-DECODER NETWORK

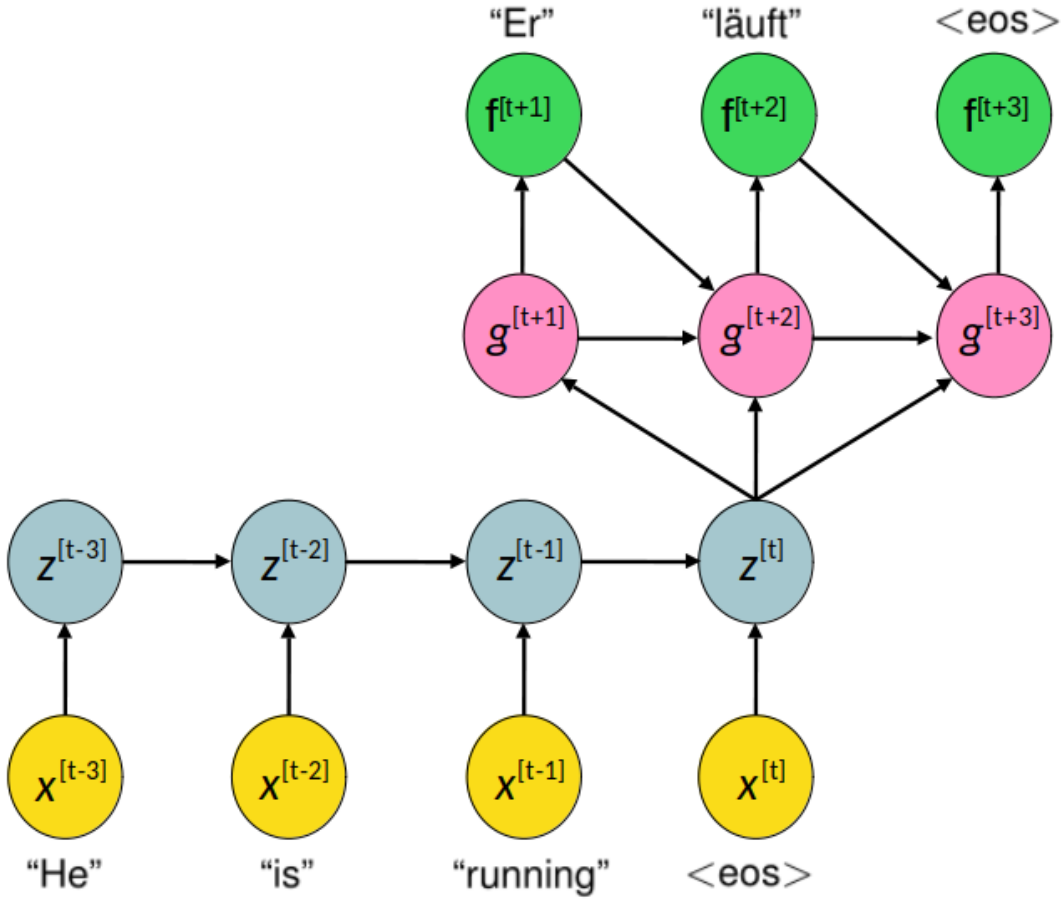


Figure: In the first part of the network, information from the input is encoded in the context vector, here the final hidden state, which is then passed on to every hidden state of the decoder, which produces the target sequence.

ENCODER-DECODER NETWORK

- An input/encoder-RNN processes the input sequence of length n_x and computes a fixed-length context vector C, usually the final hidden state or simple function of the hidden states.
- One time step after the other information from the input sequence is processed, added to the hidden state and passed forward in time through the recurrent connections between hidden states in the encoder.
- The context vector summarizes important information from the input sequence, e.g. the intent of a question in a question answering task or the meaning of a text in the case of machine translation.
- The decoder RNN uses this information to predict the output, a sequence of length n_y, which could vary from n_x.

ENCODER-DECODER NETWORK

- In machine translation, the decoder is a language model with recurrent connections between the output at one time step and the hidden state at the next time step as well as recurrent connections between the hidden states:

$$\mathbb{P}(y^{[1]}, \ldots, y^{[y_n]} | \mathbf{x}^{[1]}, \ldots, \mathbf{x}^{[x_n]}) = \prod_{t=1}^{n_y} p(y^{[t]} | C; y^{[1]}, \ldots, y^{[t-1]})$$

  with C being the context-vector.

- This architecture is now jointly trained to minimize the translation error given a source sentence.
- Each conditional probability is then

$$p(y^{[t]} | y^{[1]}, \ldots, y^{[t-1]}; C) = f(y^{[t-1]}, g^{[t]}, C)$$

  where f is a non-linear function, e.g. the tanh and $g^{[t]}$ is the hidden state of the decoder network.

ATTENTION

- In a classical decoder-encoder RNN all information about the input sequence must be incorporated into the final hidden state, which is then passed as an input to the decoder network.
- With a long input sequence this fixed-sized context vector is unlikely to capture all relevant information about the past.
- Each hidden state contains mostly information from recent inputs.
- Key idea: Allow the decoder to access all the hidden states of the encoder (instead of just the final one) so that it can dynamically decide which ones are relevant at each time-step in the decoding.
- This means the decoder can choose to "focus" on different hidden states (of the encoder) at different time-steps of the decoding process similar to how the human eye can focus on different regions of the visual field.
- This is known as an **attention mechanism**.

ATTENTION

- The attention mechanism is implemented by an additional component in the decoder.
- For example, this can be a simple single-hidden layer feed-forward neural network which is trained along with the RNN.
- At any given time-step *i* of the decoding process, the network computes the relevance of encoder state $\mathbf{z}^{[j]}$ as:

$$rel(\mathbf{z}^{[j]})^{[i]} = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{g}^{[i-1]}; \mathbf{z}^{[j]}])$$

  where $\mathbf{v}_a$ and $\mathbf{W}_a$ are the parameters of the feed-forward network, $\mathbf{g}^{[i-1]}$ is the decoder state from the previous time-step and ';' indicates concatenation.
- The relevance scores (for all the encoder hidden states) are then normalized which gives the *attention weights* $(\alpha^{[j]})^{[i]}$:

$$(\alpha^{[j]})^{[i]} = \frac{\exp(rel(\mathbf{z}^{[j]})^{[i]})}{\sum_{j'} \exp(rel(\mathbf{z}^{[j']})^{[i]})}$$

ATTENTION

- The attention mechanism allows the decoder network to focus on different parts of the input sequence by adding connections from all hidden states of the encoder to each hidden state of the decoder.

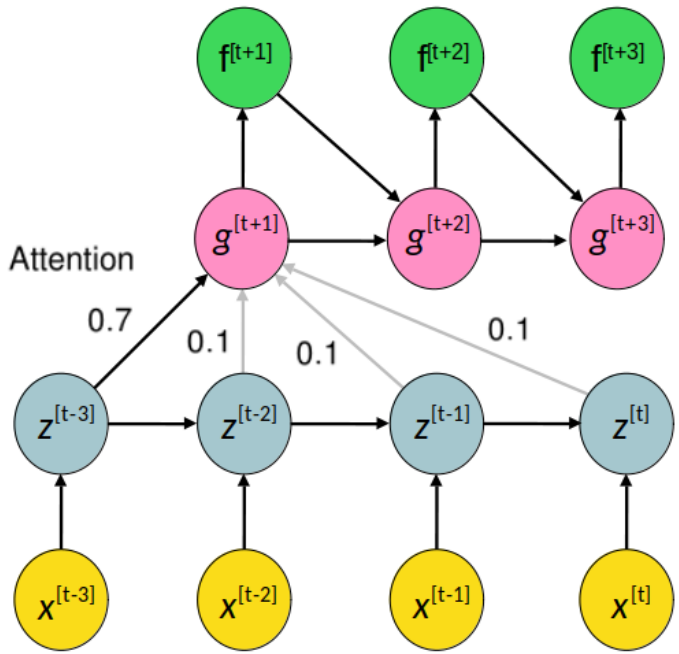


Figure: Attention at $i = t + 1$

ATTENTION

- At each time step i, a set of weights $(\alpha^{[j]})^{[i]}$ is computed which determine how to combine the hidden states of the encoder into a context vector $\mathbf{g}^{[i]} = \sum_{j=1}^{n_x} (\alpha^{[j]})^{[i]} \mathbf{z}^{[j]}$, which holds the necessary information to predict the correct output.

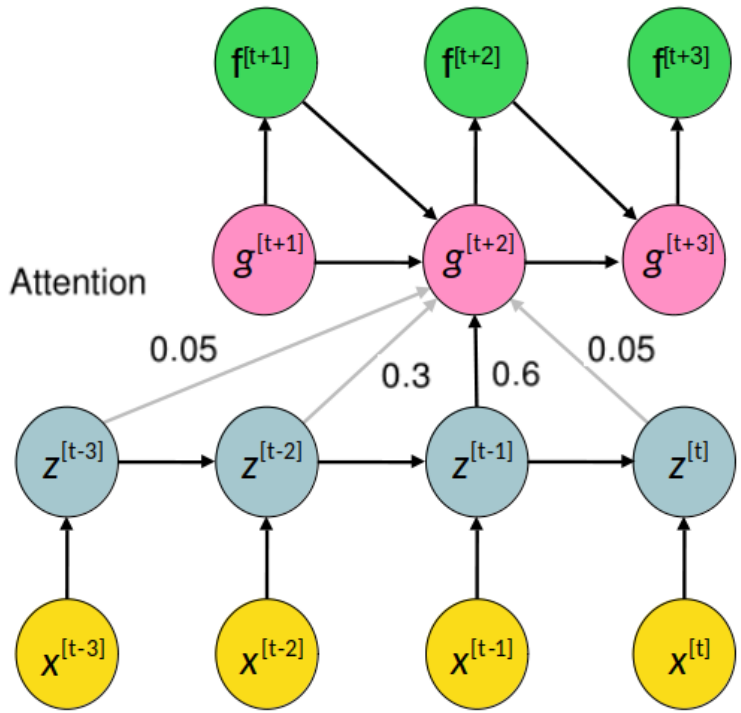


Figure: Attention at $i = t + 2$

TRANSFORMERS

- Advanced RNNs have similar limitations as vanilla RNN networks:
  - RNNs process the input data sequentially.
  - Difficulties in learning long term dependency (although GRU or LSTM perform better than vanilla RNNs, they sometimes struggle to remember the context introduced earlier in long sequences).
- These challenges are tackled by transformer networks.

TRANSFORMERS

- Transformers are solely based on attention (no RNN or CNN).
- In fact, the paper which coined the term *transformer* is called *Attention is all you need*.
- They are the state-of-the-art networks in natural language processing (NLP) tasks since 2017.
- Transformer architectures like BERT (Bidirectional Encoder Representations from Transformers, 2018) and GPT-3 (Generative Pre-trained Transformer-3, 2020) are pre-trained on a large corpus and can be fine-tuned to specific language tasks.

TRANSFORMERS

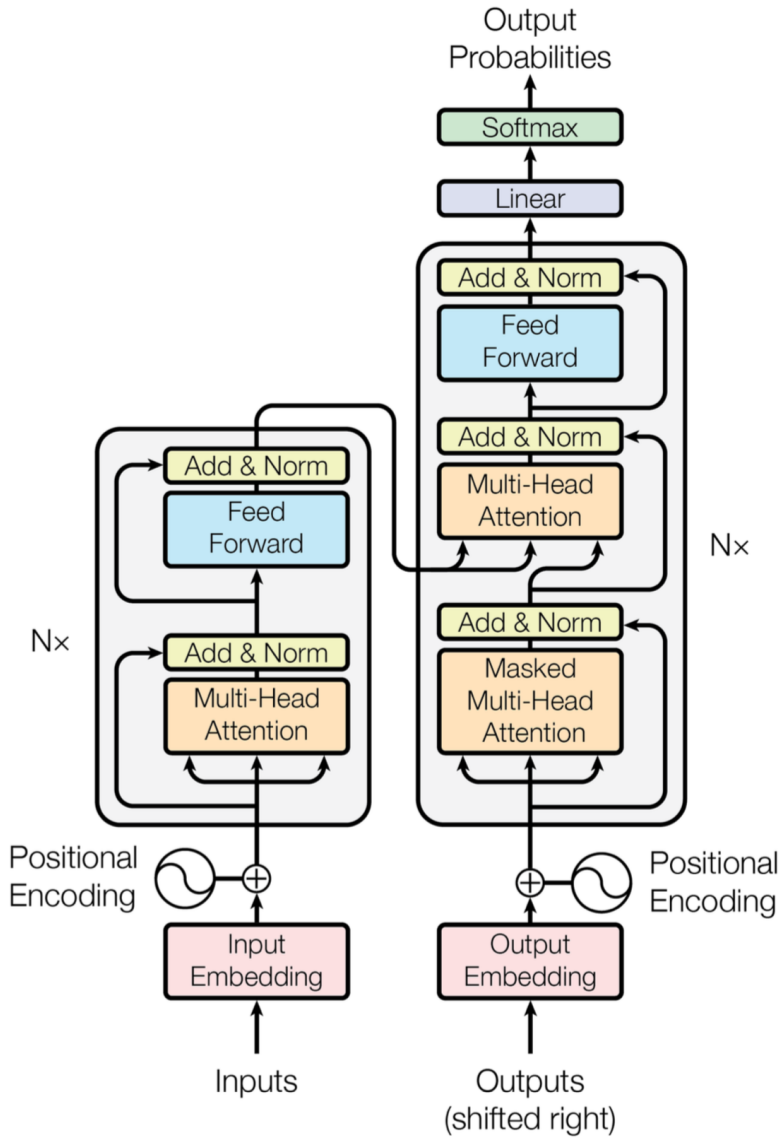


Figure 1: The Transformer - model architecture.

INTRODUCTION TO DEEP LEARNING

Introduction & MLPs

- Introduction
- A Single Neuron
- Single Hidden Layer Networks
- Multiclass Classification
- Multilayer MLPs
- Basic Training

Convolutional Neural Networks

- Introduction
- Conv2D
- Properties of Convolutional Layers
- Components of Convolution Layers

Recurrent Neural Networks

- Introduction
- Modern RNNs
- Encoder-Decoder Networks and Attention

Practical Considerations for Training Neural Networks

- Hardware and Software
- Regularization
- Advanced Training
- Activation Functions
- Residual Connections

Hardware and Software

HARDWARE FOR DEEP LEARNING

- Deep neural networks require special hardware to be trained efficiently.
- The training is done using **G**raphics **P**rocessing **U**nits (GPUs) and a special programming language called CUDA.
- Training on standard CPUs takes a very long time.

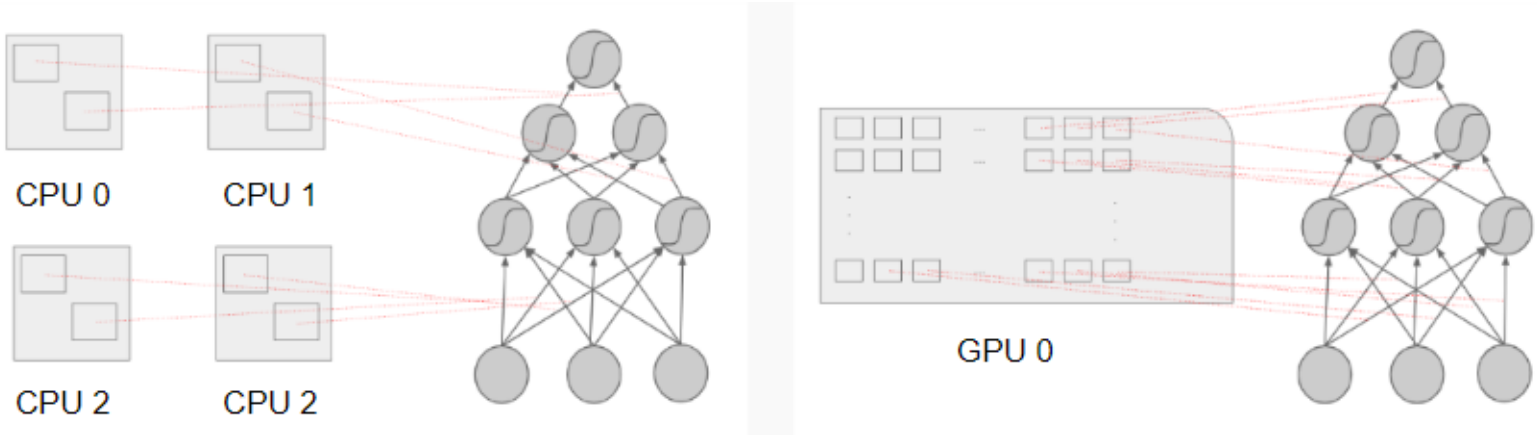


Figure: *Left:* Each CPU can do 2-8 parallel computations. *Right:* A single GPU can do thousands of simple parallel computations.

GRAPHICS PROCESSING UNITS (GPUS)

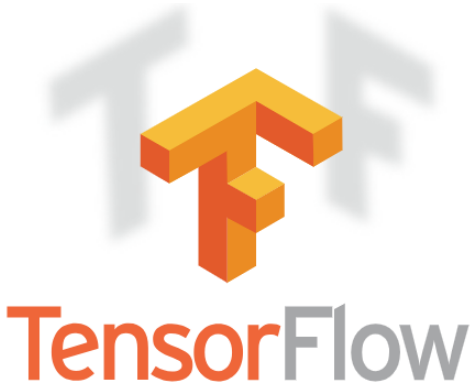
- Initially developed to accelerate the creation of graphics
- Massively parallel: identical and independent computations for every pixel
- Computer Graphics makes heavy use of linear algebra (just like neural networks)
- Less flexible than CPUs: all threads in a core concurrently execute the same instruction on different data.
- Very fast for CNNs, RNNs need more time
- Popular ones: GTX 1080 Ti, RTX 3080 / 2080 Ti, Titan RTX, Tesla V100 / A100
- Hundreds of threads per core, few thousands cores, around 10 teraFLOPS in single precision, some 10s GBs of memory
- Memory is important - some SOTA architectures do not fit GPUs with <10 GB

TENSOR PROCESSING UNITS (TPUS)

- Specialized and proprietary chip for deep learning developed by Google
- Hundreds of teraFLOPS per chip
- Can be connected together in *pods* of thousands TPUs each (result: hundreds of **peta**FLOPS per pod)
- Not a consumer product! Can be used in the Google Cloud Platform (from 1.35 USD / TPU / hour) or Google Colab (free!)
- Enables DeepMind to make impressive progress : AlphaZero for Chess became world champion after just 4 hours of training concurrently on 5064 TPUs

SOFTWARE FOR DEEP LEARNING

Tensorflow



- Popular in the industry
- Developed by Google and open source community
- Python, R, C++ and Javascript APIs
- Distributed training on GPUs and TPUs
- Tools for visualizing neural nets, running them efficiently on phones and embedded devices.

Keras

- Intuitive, high-level **wrapper** on Tensorflow for rapid prototyping
- Python and (unofficial) R APIs

SOFTWARE FOR DEEP LEARNING

Pytorch

- Popular in academia
- Supported by Facebook
- Python and C++ APIs
- Distributed training on GPUs

MXNet

- Open-source deep learning framework written in C++ and cuda (used by Amazon for their Amazon Web Services)
- Scalable, allowing fast model training
- Supports flexible model programming and multiple languages (C++, Python, Julia, Matlab, JavaScript, Go, **R**, Scala, Perl)

Regularization

DROPOUT

- Idea: reduce overfitting in neural networks by preventing complex co-adaptations of neurons.
- Method: during training, random subsets of the neurons are removed from the network (they are "dropped out").This is done by artificially setting the activations of those neurons to zero.
- Whether a given unit/neuron is dropped out or not is completely independent of the other units.
- If the network has N (input/hidden) units, applying dropout to these units can result in 2^N possible 'subnetworks'.
- Because these subnetworks are derived from the same 'parent' network, many of the weights are shared.
- Dropout can be seen as a form of "model averaging".

DROPOUT: ALGORITHM

- To train with dropout a minibatch-based learning algorithm such as stochastic gradient descent is used.
- For each training case in a minibatch, we randomly sample a binary vector/mask μ with one entry for each input or hidden unit in the network. The entries of μ are sampled independently from each other.
- The probability of sampling a mask value of 0 (dropout) for one unit is a hyperparameter known as the 'dropout rate'.
- A typical value for the dropout rate is 0.2 for input units and 0.5 for hidden units.
- Each unit in the network is multiplied by the corresponding mask value resulting in a subnet_{μ}.
- Forward propagation, backpropagation, and the learning update are run as usual.

DROPOUT: ALGORITHM

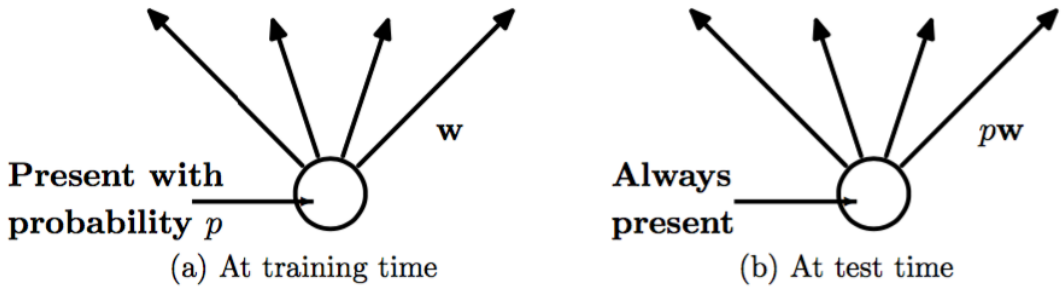
Algorithm Training a (parent) neural network with dropout rate p

1: Define parent network and initialize weights
2: **for** each minibatch: **do**
3: **for** each training sample: **do**
4: Draw mask μ using p
5: Compute forward pass for $subnet_\mu$
6: **end for**
7: Update the weights of the (parent) network by performing a gradient descent step with weight decay
8: **end for**

- The derivatives wrt. each parameter are averaged over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient of zero for that parameter.

DROPOUT: WEIGHT SCALING

- The weights of the network will be larger than normal because of dropout. Therefore, to obtain a prediction at test time the weights must be first scaled by the chosen dropout rate.
- This means that if a unit (neuron) is retrained with probability p during training, the weight at test time of that unit is multiplied by p.



Credit: Srivastava et. al. (2014)

- Weight scaling ensures that the expected total input to a neuron/unit at test time is roughly the same as the expected total input to that unit at train time, even though many of the units at train time were missing on average

DROPOUT: WEIGHT SCALING

- Rescaling of the weights can also be performed at training time instead, after each weight update at the end of the mini-batch. This is sometimes called 'inverse dropout'. Keras and PyTorch deep learning libraries implement dropout in this way.

DATASET AUGMENTATION

- Problem: low generalization because high ratio of

$$\frac{\text{complexity of the model}}{\text{\#train data}}$$

- Idea: artificially increase the train data.
  - Limited data supply $\rightarrow$ create “fake data”!
- Increase variation in inputs **without** changing the labels.
- Application:
  - Image and Object recognition (rotation, scaling, pixel translation, flipping, noise injection, vignetting, color casting, lens distortion, injection of random negatives)
  - Speech recognition (speed augmentation, vocal tract perturbation)

DATASET AUGMENTATION

(a) Original **(b)** Color **(c)** Rotate **(d)** Horizontal Stretch

Figure: (Wu et al. (2015))

DATASET AUGMENTATION

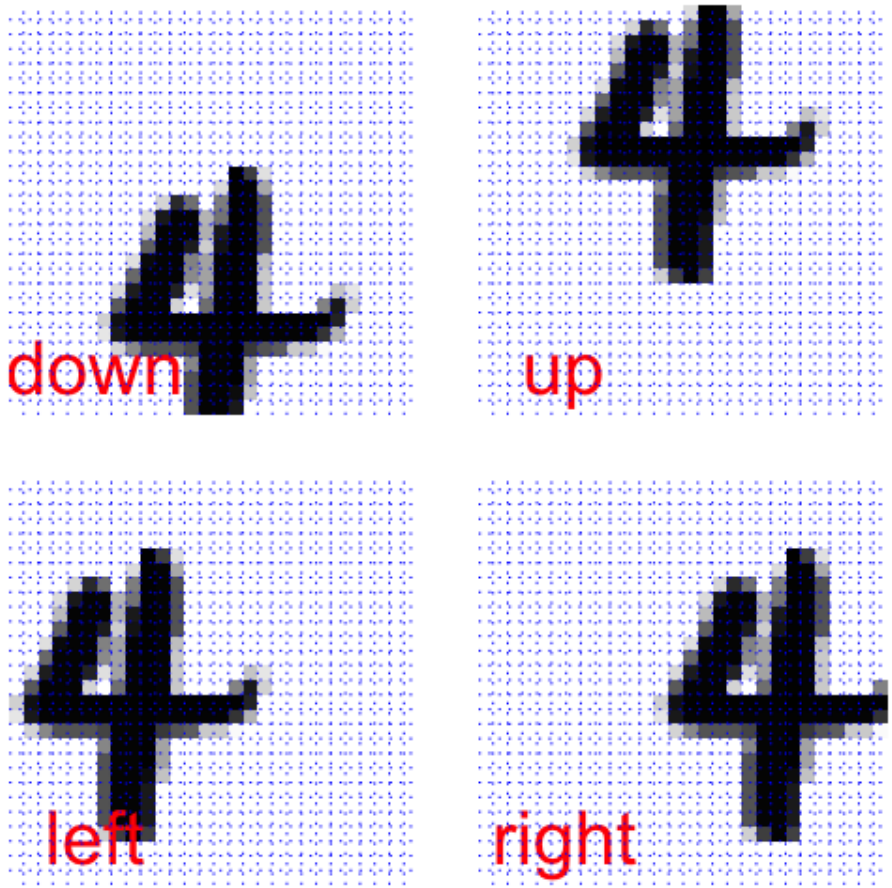


Figure: (Wu et al. (2015))

$\Rightarrow$ careful when rotating digits (6 will become 9 and vice versa)!

Advanced Training

MOMENTUM

- While SGD remains a popular optimization strategy, learning with it can sometimes be slow.
- Momentum is designed to accelerate learning, especially when facing high curvature, small but consistent or noisy gradients.
- Momentum accumulates an exponentially decaying moving average of past gradients:

$$\begin{aligned} \boldsymbol{\nu} &\leftarrow \varphi\boldsymbol{\nu} - \alpha \underbrace{\nabla_{\boldsymbol{\theta}} \Big[\frac{1}{m} \sum_i L(y^{(i)}, f(x^{(i)}, \boldsymbol{\theta})) \Big]}_{\mathbf{g}_{\boldsymbol{\theta}}} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{\nu} \end{aligned}$$

- We introduce a new hyperparameter $\varphi \in [0, 1)$, determining how quickly the contribution of previous gradients decay.
- $\boldsymbol{\nu}$ is called “velocity” and derives from a physical analogy describing how particles move through a parameter space (Newton’s law of motion).

MOMENTUM

- So far the step size was simply the gradient **g** multiplied by the learning rate α.
- Now, the step size depends on how **large** and how **aligned** a sequence of gradients is. The step size grows when many successive gradients point in the same direction.
- Common values for φ are $0.5, 0.9$ and even 0.99.
- Generally, the larger φ is relative to the learning rate α, the more previous gradients affect the current direction.
- A very good website with an in-depth analysis of momentum: `https://distill.pub/2017/momentum/`

SGD WITH MOMENTUM

Algorithm Stochastic gradient descent with momentum

1: **require** learning rate α and momentum φ
2: **require** initial parameter $\boldsymbol{\theta}$ and initial velocity $\boldsymbol{\nu}$
3: **while** stopping criterion not met **do**
4: Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
5: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
6: Compute velocity update: $\boldsymbol{\nu} \leftarrow \varphi \boldsymbol{\nu} - \alpha \hat{\mathbf{g}}$
7: Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\nu}$
8: **end while**

SGD WITH MOMENTUM

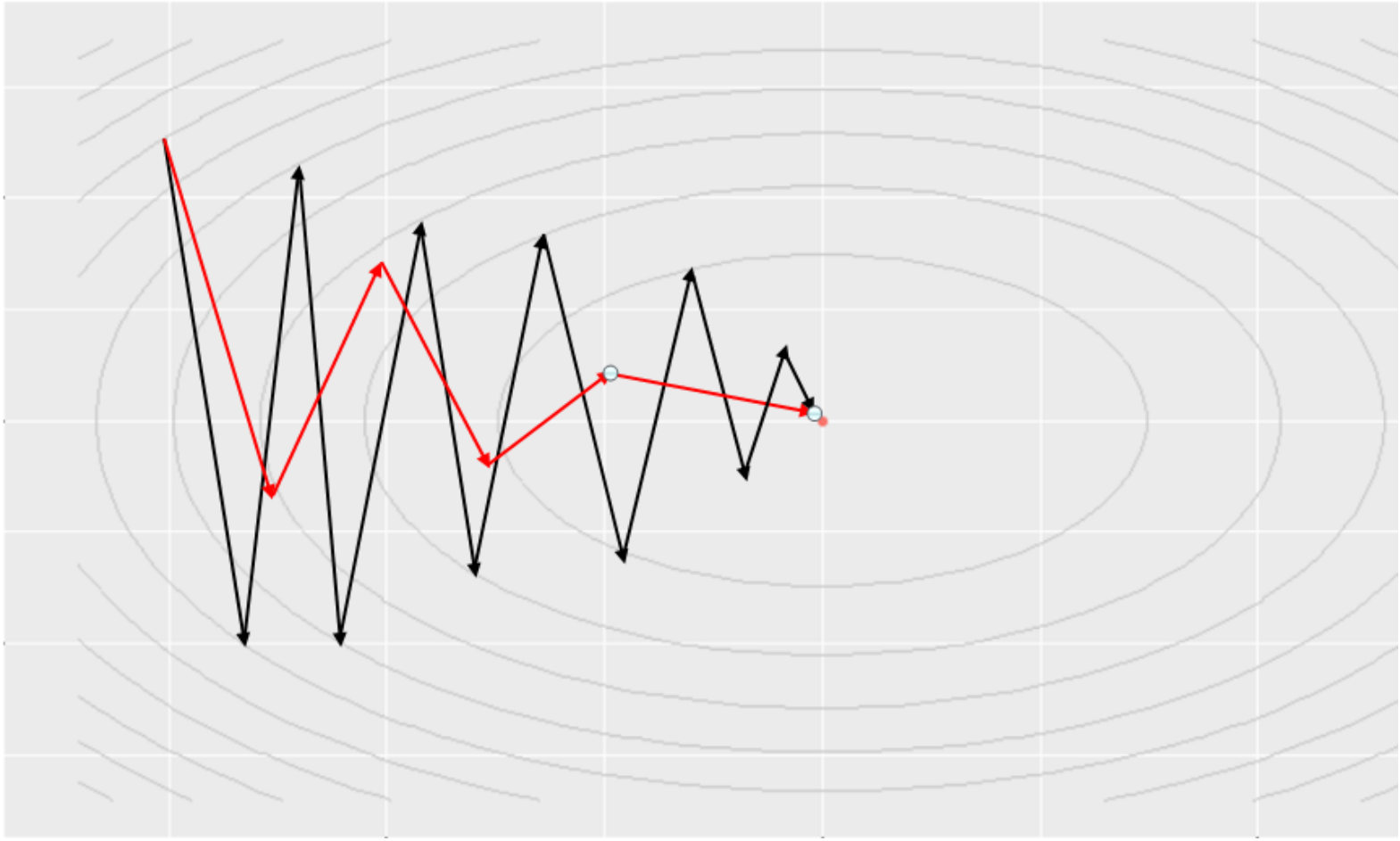
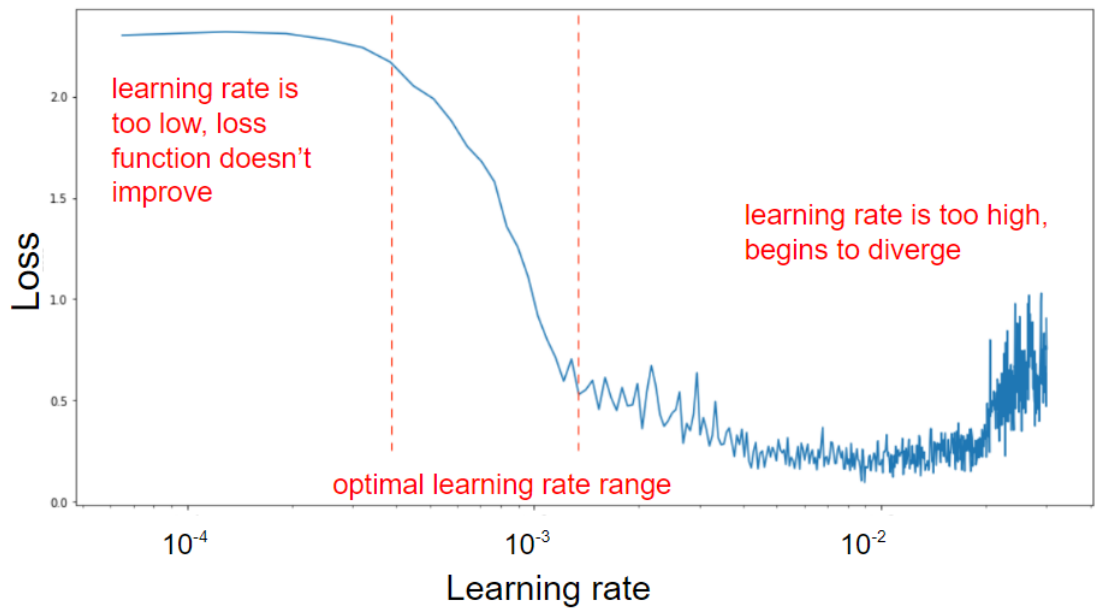


Figure: The contour lines show a quadratic loss function with a poorly conditioned Hessian matrix. The two curves show how standard gradient descent (black) and momentum (red) learn when dealing with ravines. Momentum reduces the oscillation and accelerates the convergence.

LEARNING RATE

- The learning rate is a very important hyperparameter.
- To systematically find a good learning rate, we can start at a very low learning rate and gradually increase it (linearly or exponentially) after each mini-batch.
- We can then plot the learning rate and the training loss for each batch.
- A good learning rate is one that results in a steep decline in the loss.



Credit: jeremyjordan

LEARNING RATE SCHEDULE

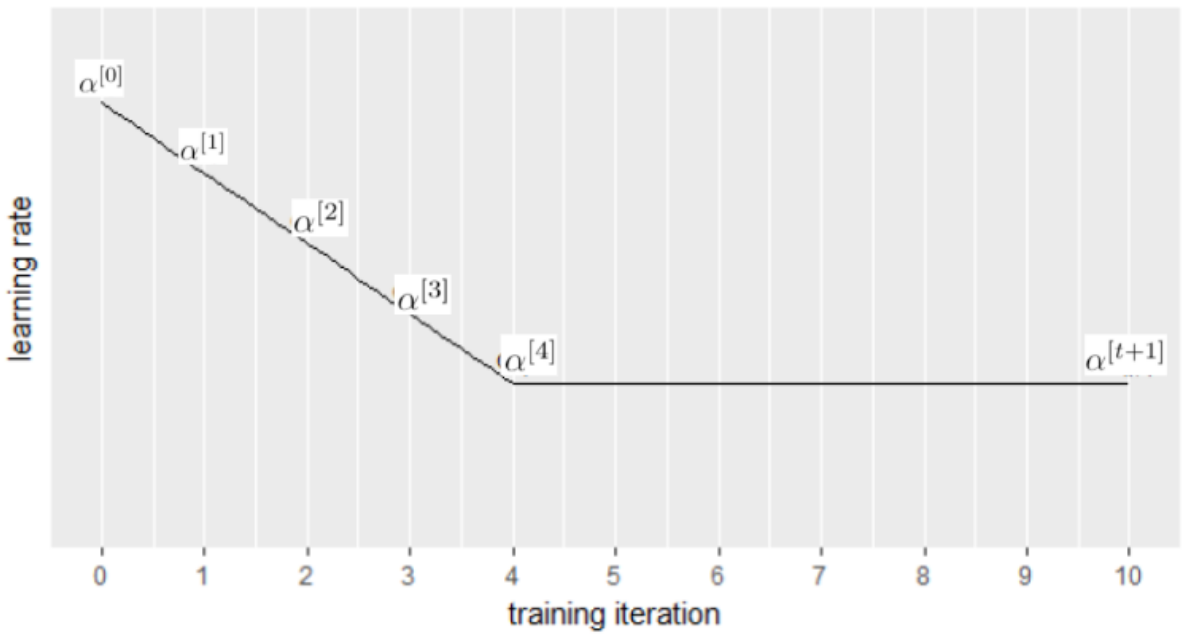
- We would like to force convergence until reaching a local minimum.
- Applying SGD, we have to decrease the learning rate over time, thus $\alpha^{[t]}$ (learning rate at training iteration t).
  - The estimator $\hat{\mathbf{g}}$ is computed based on small batches.
  - Random sampling m training samples introduces noise, that does not vanish even if we find a minimum.
- In practice, a common strategy is to decay the learning rate linearly over time until iteration τ:

$$\alpha^{[t]} = \begin{cases} \left(1 - \frac{t}{\tau}\right)\alpha^{[0]} + \frac{t}{\tau}\alpha^{[\tau]} = t\left(-\frac{\alpha^{[0]}+\alpha^{[\tau]}}{\tau}\right) + \alpha^{[0]} & \text{for } t \leq \tau \\ \alpha^{[\tau]} & \text{for } t > \tau \end{cases}$$

LEARNING RATE SCHEDULE

Example for $\tau = 4$:

iteration t	t/τ	$\alpha^{[t]}$
1	0.25	$\left(1 - \frac{1}{4}\right)\alpha^{[0]} + \frac{1}{4}\alpha^{[\tau]} = \frac{3}{4}\alpha^{[0]} + \frac{1}{4}\alpha^{[\tau]}$
2	0.5	$\frac{2}{4}\alpha^{[0]} + \frac{2}{4}\alpha^{[\tau]}$
3	0.75	$\frac{1}{4}\alpha^{[0]} + \frac{3}{4}\alpha^{[\tau]}$
4	1	$0 + \alpha^{[\tau]}$
...		$\alpha^{[\tau]}$
$t+1$		$\alpha^{[\tau]}$

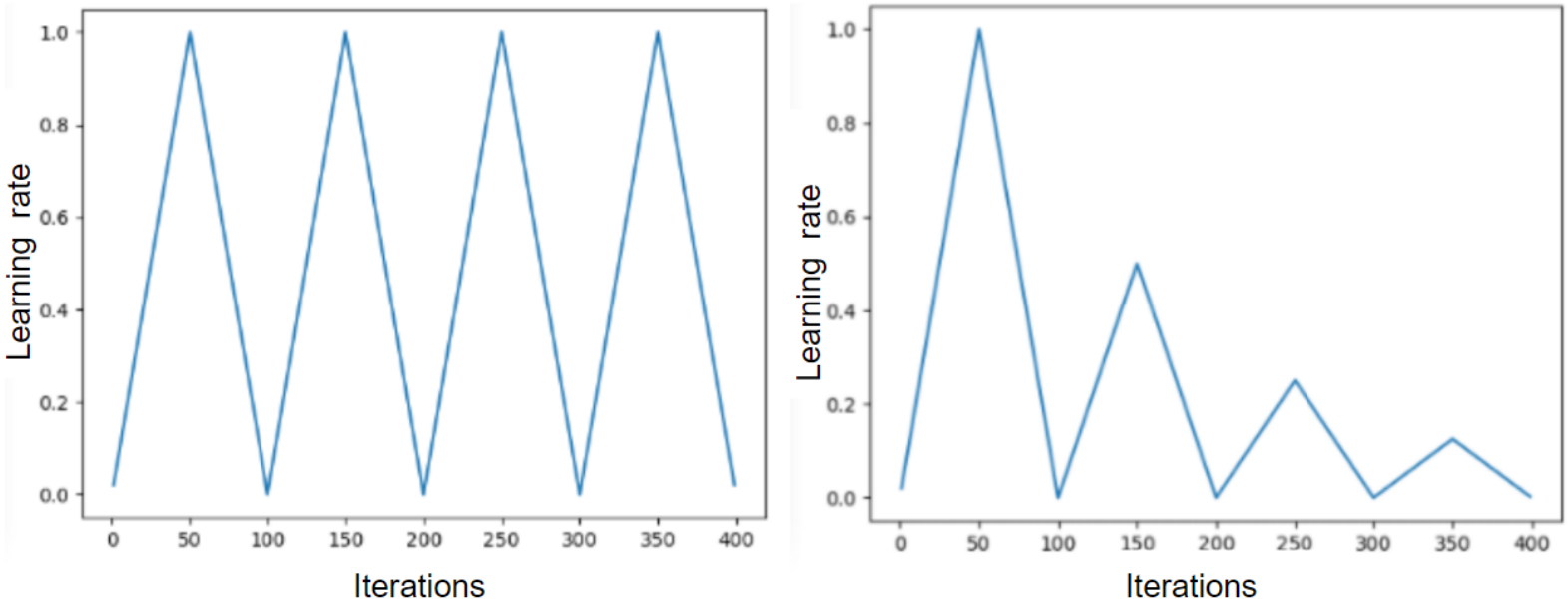


CYCLICAL LEARNING RATES

- Another option is to have a learning rate that periodically varies according to some cyclic function.
- Therefore, if training does not improve the loss anymore (possibly due to saddle points), increasing the learning rate makes it possible to rapidly traverse such regions.
- Recall, saddle points are far more likely than local minima in deep nets.
- Each cycle has a fixed length in terms of the number of iterations.

CYCLICAL LEARNING RATES

- One such cyclical function is the "triangular" function.

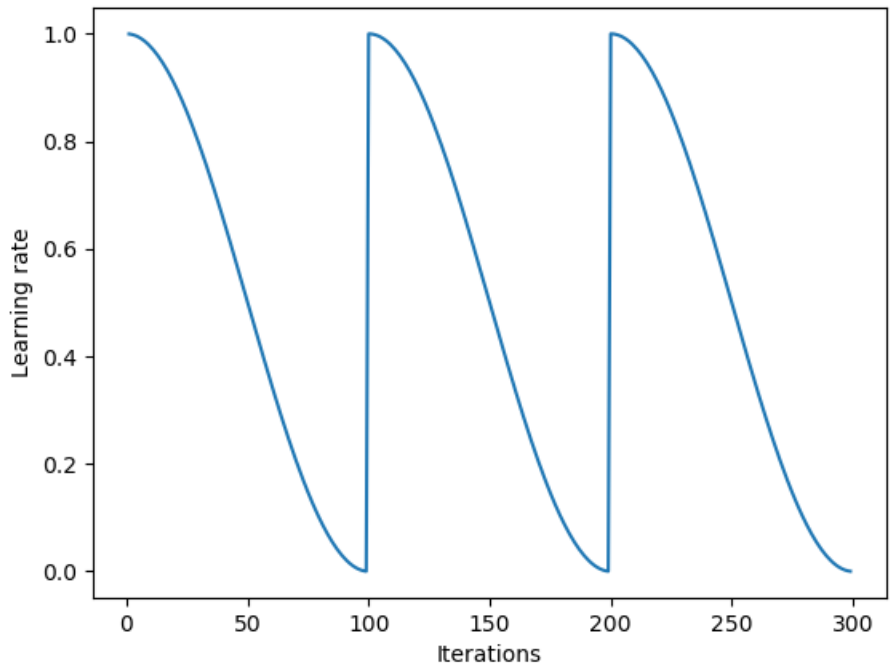


Credit: Hafidz Zulkifli

- In the right image, the range is cut in half after each cycle.

CYCLICAL LEARNING RATES

- Yet another option is to abruptly "restart" the learning rate after a fixed number of iterations.
- Loshchilov et al. (2016) proposed "cosine annealing" (between restarts).



Credit: Hafidz Zulkifli

ADAPTIVE LEARNING RATES

- The learning rate is reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on the models performance.
- Naturally, it might make sense to use a different learning rate for each parameter, and automatically adapt them throughout the training process.

ADAGRAD

- Adagrad adapts the learning rate to the parameters.
- In fact, Adagrad scales learning rates inversely proportional to the square root of the sum of the past squared derivatives.
  - Parameters with large partial derivatives of the loss obtain a rapid decrease in their learning rate.
  - Parameters with small partial derivatives on the other hand obtain a relatively small decrease in their learning rate.
- For that reason, Adagrad might be well suited when dealing with sparse data.
- Goodfellow et al. (2016) say that the accumulation of squared gradients can result in a premature and overly decrease in the learning rate.

ADAGRAD

Algorithm Adagrad

require Global learning rate α
require Initial parameter $\boldsymbol{\theta}$
require Small constant β, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Accumulate squared gradient $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Compute update: $\nabla\boldsymbol{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ (division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

- "$\odot$" is called Hadamard or element-wise product.
- Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \; B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, \; \text{then } A \odot B = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix}$$

RMSPROP

- RMSprop is a modification of Adagrad.
- It's intention is to resolve Adagrad's radically diminishing learning rates.
- The gradient accumulation is replaced by an exponentially weighted moving average.
- Theoretically, that leads to performance gains in non-convex scenarios.
- Empirically, RMSProp is a very effective optimization algorithm. Particularly, it is employed routinely by deep learning practitioners.

RMSPROP

Algorithm RMSProp

require Global learning rate α and decay rate $\rho \in [0, 1)$
require Initial parameter $\boldsymbol{\theta}$
require Small constant β, perhaps 10^{-6}, for numerical stability
Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Accumulate squared gradient $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Compute update: $\nabla\boldsymbol{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

ADAM

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
- Adam uses the first and the second moments of the gradients.
  - Adam keeps an exponentially decaying average of past gradients (first moment).
  - Like RMSProp it stores an exponentially decaying average of past squared gradients (second moment).
  - Thus, it can be seen as a combination of RMSProp and momentum.
- Basically Adam uses the combined averages of previous gradients at different moments to give it more “persuasive power” to adaptively update the parameters.

ADAM

Algorithm Adam

require Step size α (suggested default: 0.001)
require Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$ (suggested defaults: 0.9 and 0.999 respectively)
require Small constant β (suggested default 10^{-8})
require Initial parameters $\boldsymbol{\theta}$
Initialize time step $t = 0$
Initialize 1st and 2nd moment variables $\mathbf{s}^{[0]} = 0, \mathbf{r}^{[0]} = 0$
while stopping criterion not met **do**
    $t \leftarrow t + 1$
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}}^{[t]} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Update biased first moment estimate: $\mathbf{s}^{[t]} \leftarrow \rho_1 \mathbf{s}^{[t-1]} + (1 - \rho_1)\hat{\mathbf{g}}^{[t]}$
    Update biased second moment estimate: $\mathbf{r}^{[t]} \leftarrow \rho_2 \mathbf{r}^{[t-1]} + (1 - \rho_2)\hat{\mathbf{g}}^{[t]} \odot \hat{\mathbf{g}}^{[t]}$
    Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}^{[t]}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}^{[t]}}{1-\rho_2^t}$
    Compute update: $\nabla\boldsymbol{\theta} = -\alpha \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\beta}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

BATCH NORMALIZATION

- Batch Normalization (BatchNorm) is an extremely popular technique that improves the training speed and stability of deep neural nets.
- It is an extra component that can be placed between each layer of the neural network.
- It works by changing the "distribution" of activations at each hidden layer of the network.
- We know that it is sometimes beneficial to normalize the inputs to a learning algorithm by shifting and scaling all the features so that they have 0 mean and unit variance.
- BatchNorm applies a similar transformation to the activations of the hidden layers (with a couple of additional tricks).

BATCH NORMALIZATION

- For a hidden layer with neurons z_j, $j = 1, \ldots, J$, BatchNorm is applied to each z_j by considering the activations of z_j **over a given minibatch** of inputs.
- Let $z_j^{(i)}$ denote the activation of z_j for input $x^{(i)}$ in the minibatch (of size m).
- The mean and variance of the activations are

$$\mu_j = \frac{1}{m} \sum_i^m z_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m} \sum_i^m (z_j^{(i)} - \mu_j)^2$$

- Each $z_j^{(i)}$ is then normalized

$$\tilde{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where a small constant, ϵ, is added for numerical stability.

BATCH NORMALIZATION

- It may not be desirable to normalize the activations in such a rigid way because potentially useful information can be lost in the process.
- Therefore, we commonly let the training algorithm decide the "right amount" of normalization by allowing it to re-shift and re-scale $\tilde{z}_j^{(i)}$ to arrive at the batch normalized activation $\hat{z}_j^{(i)}$:

$$\hat{z}_j^{(i)} = \gamma_j \tilde{z}_j^{(i)} + \beta_j$$

- γ_j and β_j are learnable parameters that are also tweaked by backpropogation.
- $\hat{z}_j^{(i)}$ then becomes the input to the next layer.
- Note: The algorithm is free to scale and shift each $\tilde{z}_j^{(i)}$ back to its original (unnormalized) value.

BATCH NORMALIZATION: ILLUSTRATION

- Recall: $z_j = \sigma(W_j^T x + b_j)$
- So far, we have applied batch-norm to the activation z_j. It is possible (and more common) to apply batch norm to $W_j^T x + b_j$ *before* passing it to the nonlinear activation σ.

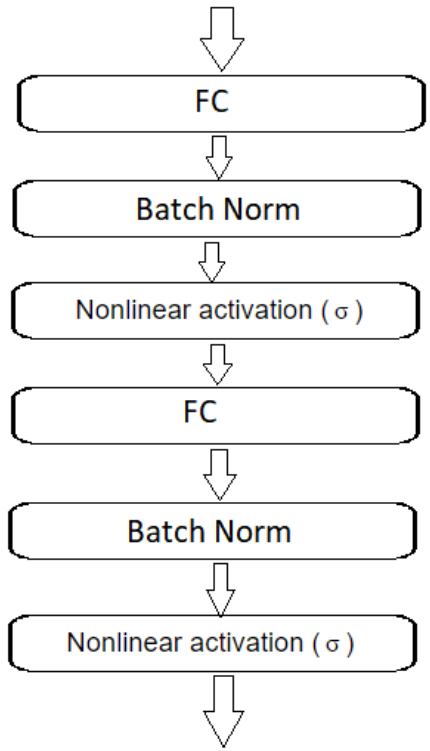
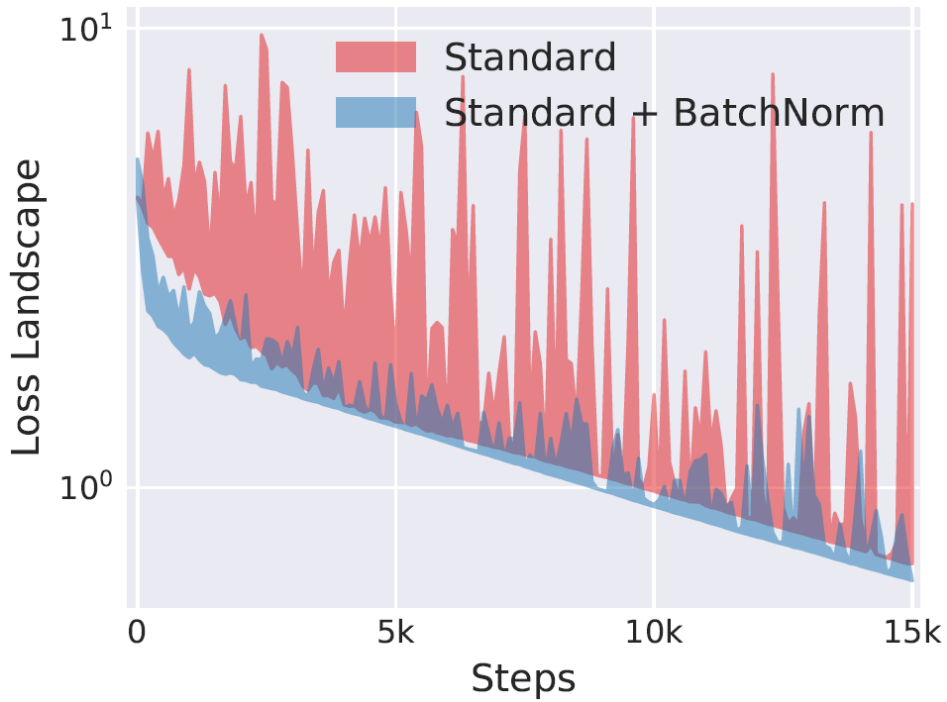


Figure: FC = Fully Connected layer. BatchNorm is applied *before* the nonlinear activation function.

BATCH NORMALIZATION

- The key impact of BatchNorm on the training process is this: It reparametrizes the underlying optimization problem to **make its landscape significantly more smooth**.
- One aspect of this is that the loss changes at a smaller rate and the magnitudes of the gradients are also smaller (see Santurkar et al. 2018).



BATCH NORMALIZATION: PREDICTION

- Once the network has been trained, how can we generate a prediction for a single input (either at test time or in production)?
- One option is to feed the entire training set to the (trained) network and compute the means and standard deviations.
- More commonly, during training, an exponentially weighted running average of each of these statistics over the minibatches is maintained.
- The learned γ and β parameters are then used (in conjunction with the running averages) to generate the output.

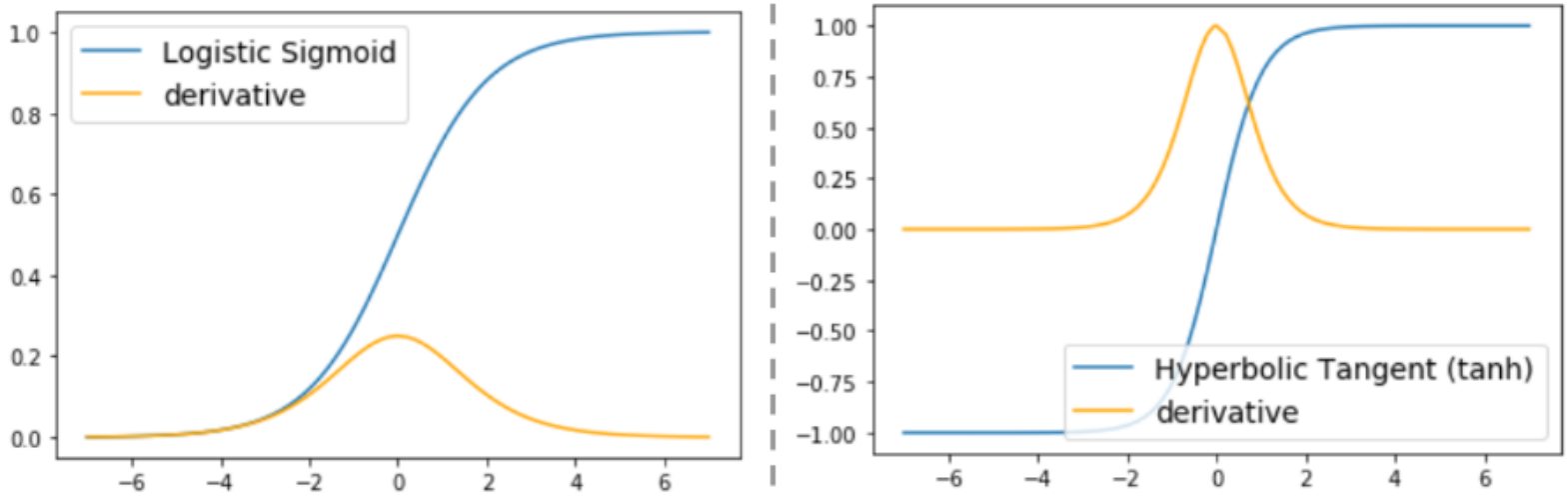
Activation Functions

HIDDEN ACTIVATIONS

- Recall, hidden-layer activation functions make it possible for deep neural nets to learn complex non-linear functions.
- The design of hidden units is an extremely active area of research. It is usually not possible to predict in advance which activation will work best. Therefore, the design process often consists of trial and error.
- In the following, we will limit ourselves to the most popular activations - Sigmoidal activation and ReLU.
- It is possible for many other functions to perform as well as these standard ones. An overview of further activations can be found ▸ here.

SIGMOIDAL ACTIVATIONS

- Sigmoidal functions such as *tanh* and the *logistic sigmoid* bound the outputs to a certain range by "squashing" their inputs.



- In each case, the function is only sensitive to its inputs in a small neighborhood around 0.
- Furthermore, the derivative is never greater than 1 and is close to zero across much of the domain.

SIGMOIDAL ACTIVATION FUNCTIONS

1. **Saturating Neurons:**
   - We know: $\sigma'(z_{in}) \to 0$ for $|z_{in}| \to \infty$.

   $\to$ Neurons with sigmoidal activations "saturate" easily, that is, they stop being responsive when $|z_{in}| \gg 0$.

SIGMOIDAL ACTIVATION FUNCTIONS

2. **Vanishing Gradients:** Consider the vector of error signals $\boldsymbol{\delta}^{(i)}$ in layer i

$$\boldsymbol{\delta}^{(i)} = \mathbf{W}^{(i+1)}\boldsymbol{\delta}^{(i+1)} \odot \sigma'\left(\boldsymbol{z}_{in}^{(i)}\right), i \in \{1, ..., O\}.$$

Each k-th component of the vector expresses how much the loss L changes when the input to the k-th neuron $z_{k,in}^{(i)}$ changes.

- We know: $\sigma'(z) < 1$ for all $z \in \mathbb{R}$.

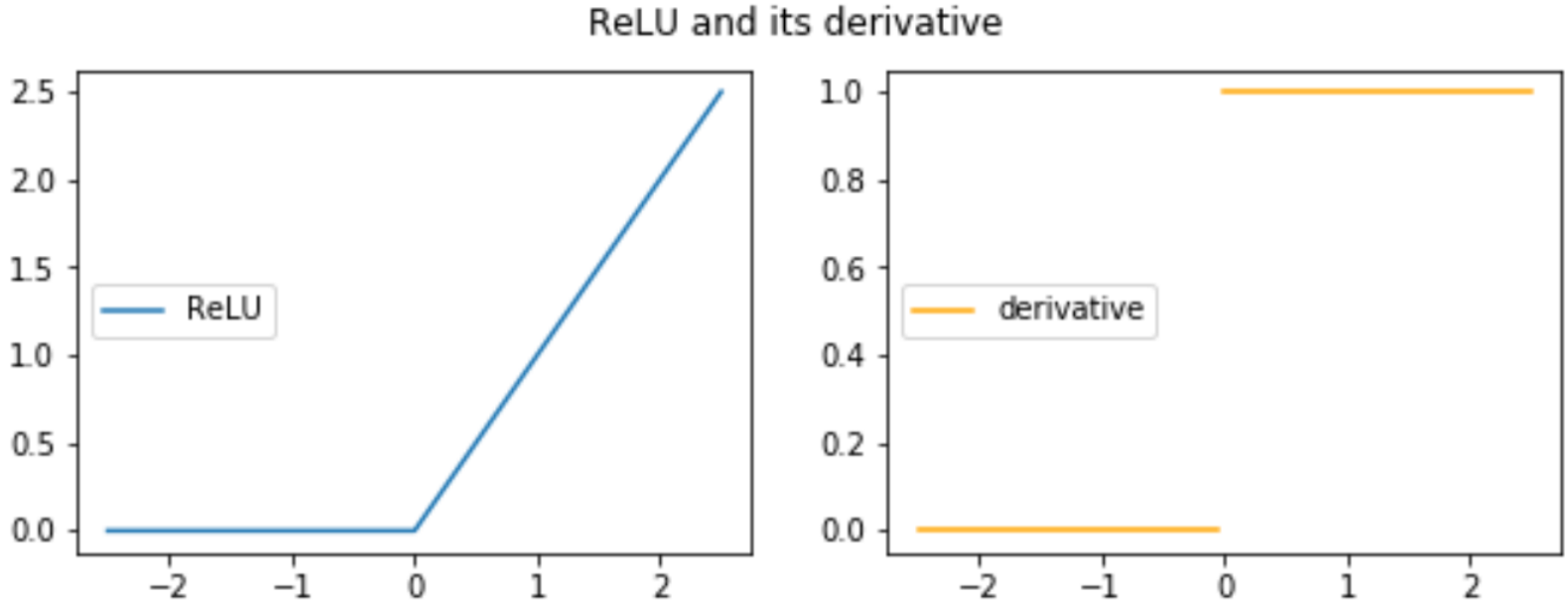
$\rightarrow$ In each step of the recursive formula above, the value will be multiplied by a value smaller than one

$$\begin{aligned}
\boldsymbol{\delta}^{(1)} &= \mathbf{W}^{(2)}\boldsymbol{\delta}^{(2)} \odot \sigma'\left(z_{in}^{(i)}\right) \\
&= \mathbf{W}^{(2)}\left(\mathbf{W}^{(3)}\boldsymbol{\delta}^{(3)} \odot \sigma'\left(z_{in}^{(i)}\right)\right) \odot \sigma'\left(z_{in}^{(i)}\right) \\
&= \ldots
\end{aligned}$$

- When this occurs, earlier layers train *very* slowly (or not at all).

RECTIFIED LINEAR UNITS (RELU)

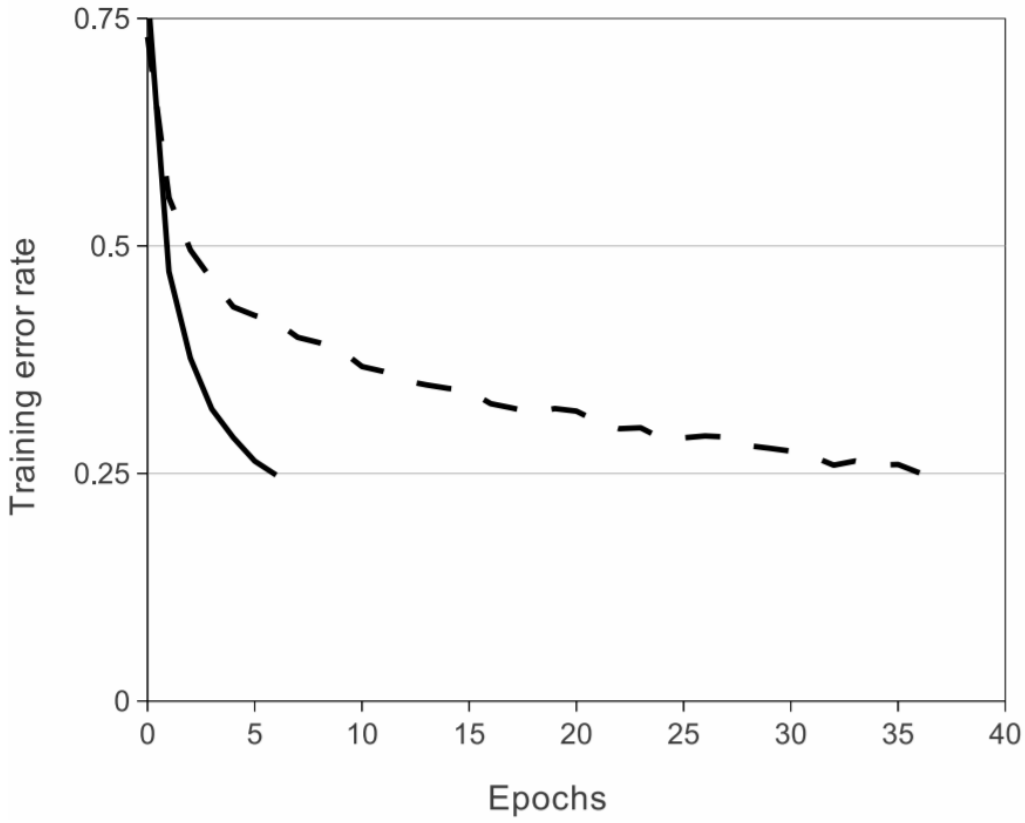
- The ReLU activation solves the vanishing gradient problem.



- In regions where the activation is positive, the derivative is 1.
- As a result, the derivatives do not vanish along paths that contain such "active" neurons even if the network is deep.
- Note that the ReLU is not differentiable at 0 (Software implementations return either 0 or 1 for the derivative at this point).

RECTIFIED LINEAR UNITS (RELU)

- ReLU units can significantly speed up training compared to units with saturating activations.

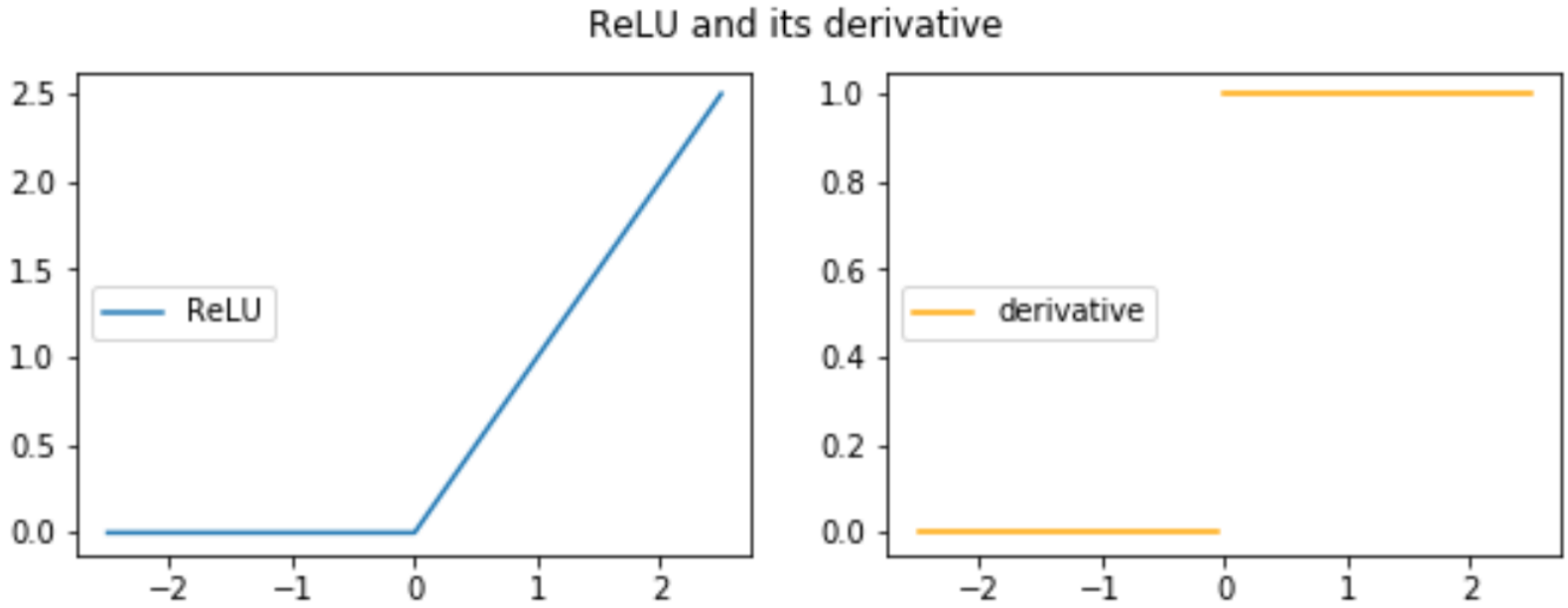


Source : Krizhevsky et al. (2012)

Figure: A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on the CIFAR-10 dataset six times faster than an equivalent network with tanh neurons (dashed line).

RECTIFIED LINEAR UNITS (RELU)

- A downside of ReLU units is that when the input to the activation is negative, the derivative is zero. This is known as the "dying ReLU problem".

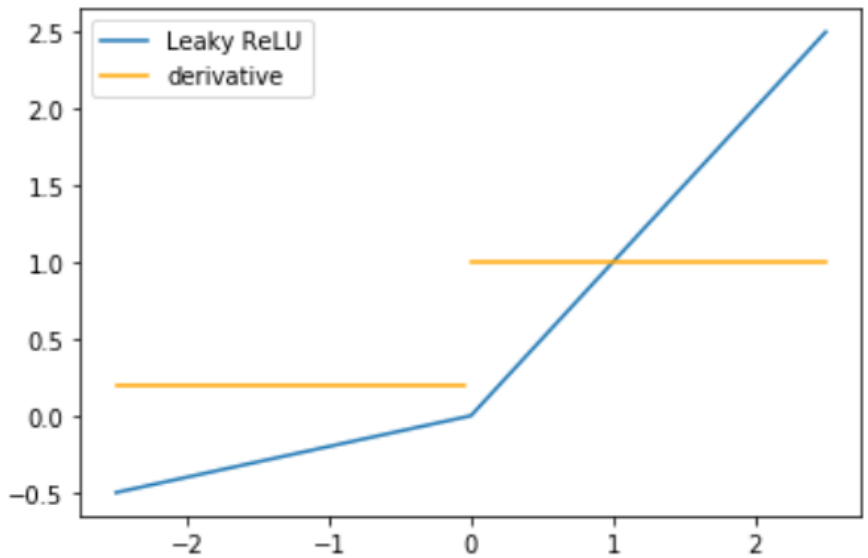


- When a ReLU unit "dies", that is, when its activation is 0 for all datapoints, it kills the gradient flowing through it during backpropogation.
- This means such units are never updated during training and the problem can be irreversible.

GENERALIZATIONS OF RELU

- There exist several generalizations of the ReLU activation that have non-zero derivatives throughout their domains.
- *Leaky ReLU*:

$$LReLU(v) = \begin{cases} v & v \geq 0 \\ \alpha v & v < 0 \end{cases}$$

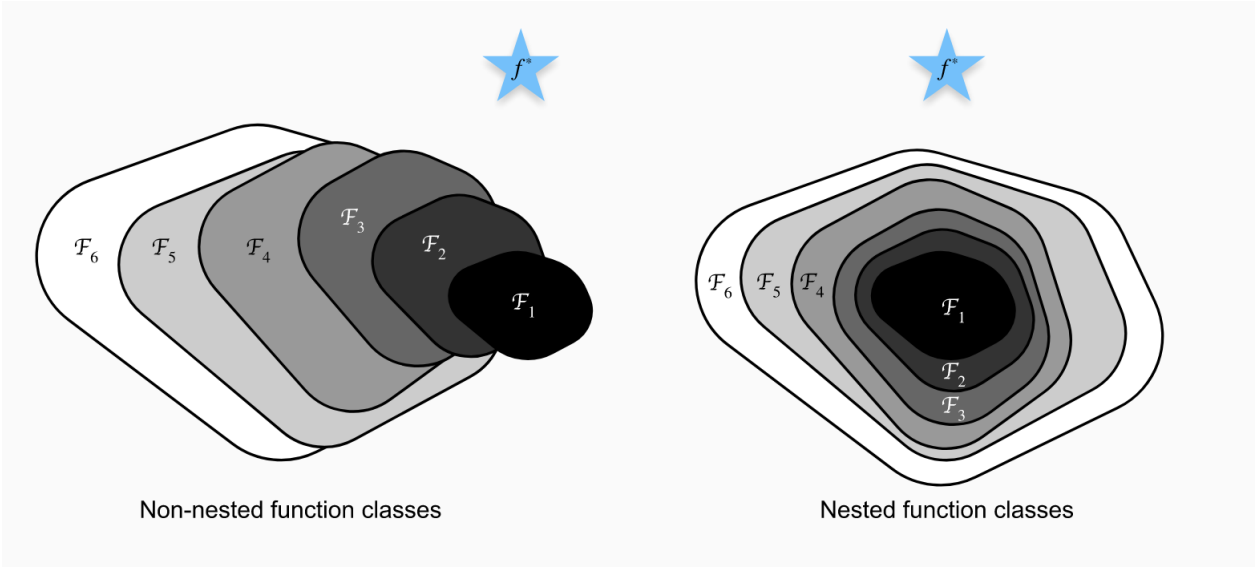


- Unlike the ReLU, when the input to the Leaky ReLU activation is negative, the derivative is α which is a small positive value (such as 0.01).

Residual Connections

RESIDUAL BLOCK (SKIP CONNECTIONS)

Problem setting: theoretically, we could build infinitely deep architectures as the net should learn to pick the beneficial layers and skip those that do not improve the performance automatically.



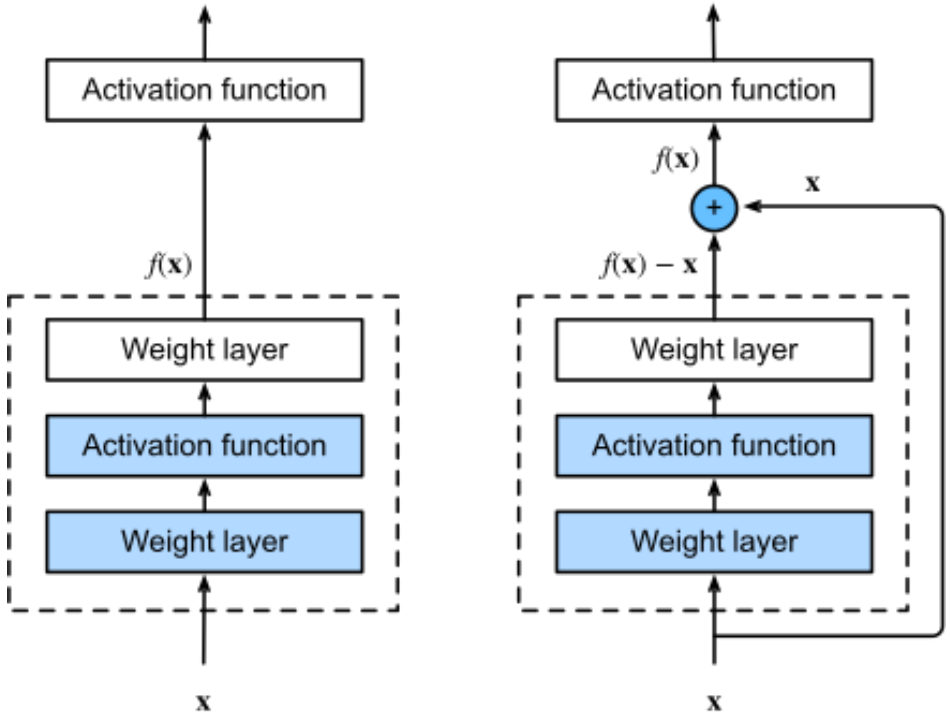
credit : D2DL

Figure: For non-nested function classes, a larger function class does not guarantee to get closer to the "truth" function ($\mathcal{F}*$). This does not happen in nested function classes.

RESIDUAL BLOCK (SKIP CONNECTIONS)

- But: this skipping would imply learning an identity mapping $\mathbf{x} = \mathcal{F}(\mathbf{x})$. It is very hard for a neural net to learn such a 1:1 mapping through the many non-linear activations in the architecture.
- Solution: offer the model explicitly the opportunity to skip certain layers if they are not useful.
- Introduced in *He et. al , 2015* and motivated by the observation that stacking evermore layers increases the test- as well as the train-error ($\neq$ overfitting).

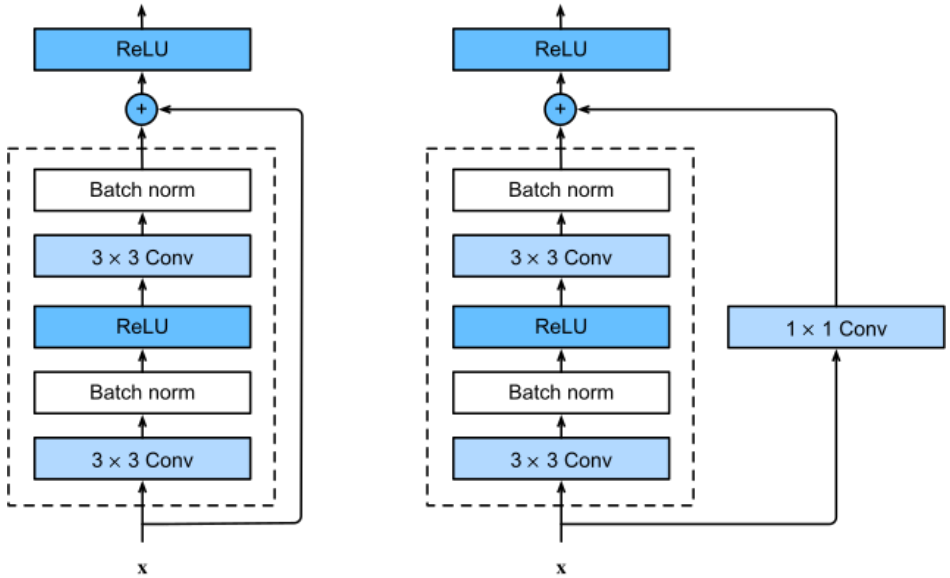
RESIDUAL BLOCK (SKIP CONNECTIONS)



credit : D2DL

Figure: A regular block (left) and a residual block (right).

RESIDUAL BLOCK (SKIP CONNECTIONS)



credit : D2DL

Figure: ResNet block with and without 1×1 convolution.The information flows through two layers and the identity function. Both streams of information are then element-wise summed and jointly activated.

RESIDUAL BLOCK (SKIP CONNECTIONS)

- Let $\mathcal{H}(\mathbf{x})$ be the optimal underlying mapping that should be learned by (parts of) the net.
- **x** is the input in layer *l* (can be raw data input or the output of a previous layer).
- $\mathcal{H}(\mathbf{x})$ is the output from layer *l*.
- Instead of fitting $\mathcal{H}(\mathbf{x})$, the net is ought to learn the residual mapping $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$ whilst **x** is added via the identity mapping.
- Thus, $\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$, as formulated on the previous slide.
- The model should only learn the **residual mapping** $\mathcal{F}(\mathbf{x})$
- Thus, the procedure is also referred to as **Residual Learning**.

RESIDUAL BLOCK (SKIP CONNECTIONS)

- The element-wise addition of the learned residuals $\mathcal{F}(\mathbf{x})$ and the identity-mapped data $\mathbf{x}$ requires both to have the same dimensions.
- To allow for downsampling within $\mathcal{F}(\mathbf{x})$ (via pooling or valid-padded convolutions), the authors introduce a linear projection layer W_s .
- W_s ensures that $\mathbf{x}$ is brought to the same dimensionality as $\mathcal{F}(\mathbf{x})$ such that:

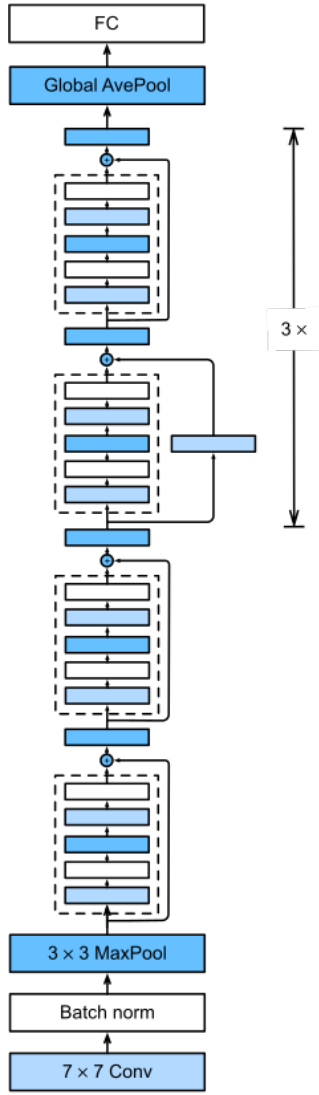
$$y = \mathcal{F}(\mathbf{x}) + W_s\mathbf{x},$$

- y is the output of the skip module and W_s represents the weight matrix of the linear projection (# rows of W_s = dimensionality of $\mathcal{F}(\mathbf{x})$).
- This idea applies to fully connected layers as well as to convolutional layers.

RESNET ARCHITECTURE

- The residual mapping can learn the identity function more easily, such as pushing parameters in the weight layer to zero.
- We can train an effective deep neural network by having residual blocks.
- Inputs can forward propagate faster through the residual connections across layers.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

RESNET ARCHITECTURE



credit : D2DL

Figure: The ResNet-18 architecture.