Deep Learning

Spring 2026

Course Introduction

Andreas Aakerberg

anaa@create.aau.dk

About me

- Andreas Aakerberg
- Assistant Professor at the Visual Analysis and Perception Lab
- Research consultant @ Milestone Systems A/S

VISUAL ANALYSIS & PERCEPTION LAB

milestone

- Master’s degree in Vision, Graphics and Interactive Systems (VGIS) from AAU

AALBORG UNIVERSITET

EPFL

- PhD in Deep Learning based Image Super-Resolution from AAU and EPFL
- Industry experience from Intel, HSA Systems and Milestone Systems

intel

HSA SYSTEMS

milestone

- E-mail: anaa@create.aau.dk
- Office: Rendsburggade 14, room 6.305

Instructors

VISUAL ANALYSIS & PERCEPTION LAB

Mohammad Sabet
Assistant Professor

Gala H.-Renaux
PhD Fellow

Machine Learning research group

Zheng-Hua Tan
Professor

Holger Bovbjerg
PhD Fellow

Sarthak Yadav
PhD Fellow

Course Overview

- Lecture 1: Fundamentals of Deep Learning I (Andreas)
- Lecture 2: Fund. of Deep Learning II + Mini Project Intro (Andreas)
- Lecture 3: Bias Fainess and XAI (Mohammad Naaser Sabet)
- Lecture 4: Transformers (Sarthak Yadav)
- Lecture 5: Failures, Distribution Shift and Uncertainty (Galadrielle Humblot-Renaux) + Mini Project presentation + QA session
- Lecture 6: Generative (Zheng-Hua Tan)
- Lecture 7: Adversarial Learning (Zheng-Hua Tan)
- Lecture 8: SSL 1 (Sarthak Yadav)
- Lecture 9: SSL 2 (Sarthak Yadav) + Reinforcement Learning (Zheng-Hua Tan)

Why you need to learn about Deep Learning

- **Relevance in AI:** Deep Learning is currently **the most successful approach to artificial intelligence** (Others include rule-based and genetic algorithms and knowledge graphs)
- **Career Opportunities:** Deep-learning expertise is highly sought after in the industry and academia:

What Netflix's $900,000 AI Job Offer Tells Us About The State Of Work

By Shara Hutchinson, Former Contributor. ⓘ I am a transformational strategist, speaker and found...

Published Sep 20, 2023, 08:54am EDT

Share Save

Human meets AI: A moment of connection
CANVA

LOADING VIDEO PLAYER...

FORBES' FEATURED VIDEO

Netflix's recent job posting, offering up to $900,000 for an AI-focused product manager while over 9,000 writers are on strike, sends a bold message about today's workforce: adaptability and continuous learning are paramount.

AALBORG UNIVERSITY

Agenda

- Course Introduction

Lecture 1 – Fundamentals of DL part 1

- A crash course in deep learning to bring you to a level where you can follow more advanced techniques.

Curricula Learning Goals

Knowledge

- **Models** and representation learning
- Advanced topics including **attention** and **transformer** networks, **autoencoders**, **generative** adversarial networks, adversarial attacks, **self-supervised** learning, and deep **reinforcement** learning
- **Regularization**, **optimization**, hyperparameter tuning, and data augmentation
- Bias, fairness, and **explainable AI**
- **Design** and **implementation** of deep learning for selected applications

Skills

- Must be able to apply the taught methods to **solve real-world problems**.
- Must be able to **evaluate** and **compare** the methods within a specific application problem.

Competences

- Must have competences in **analyzing** a given **problem** and identifying appropriate **deep learning methods** to the problem.
- Must have competences in understanding the strengths and weaknesses of the methods

Literature, Materials and Prerequisites

Literature:

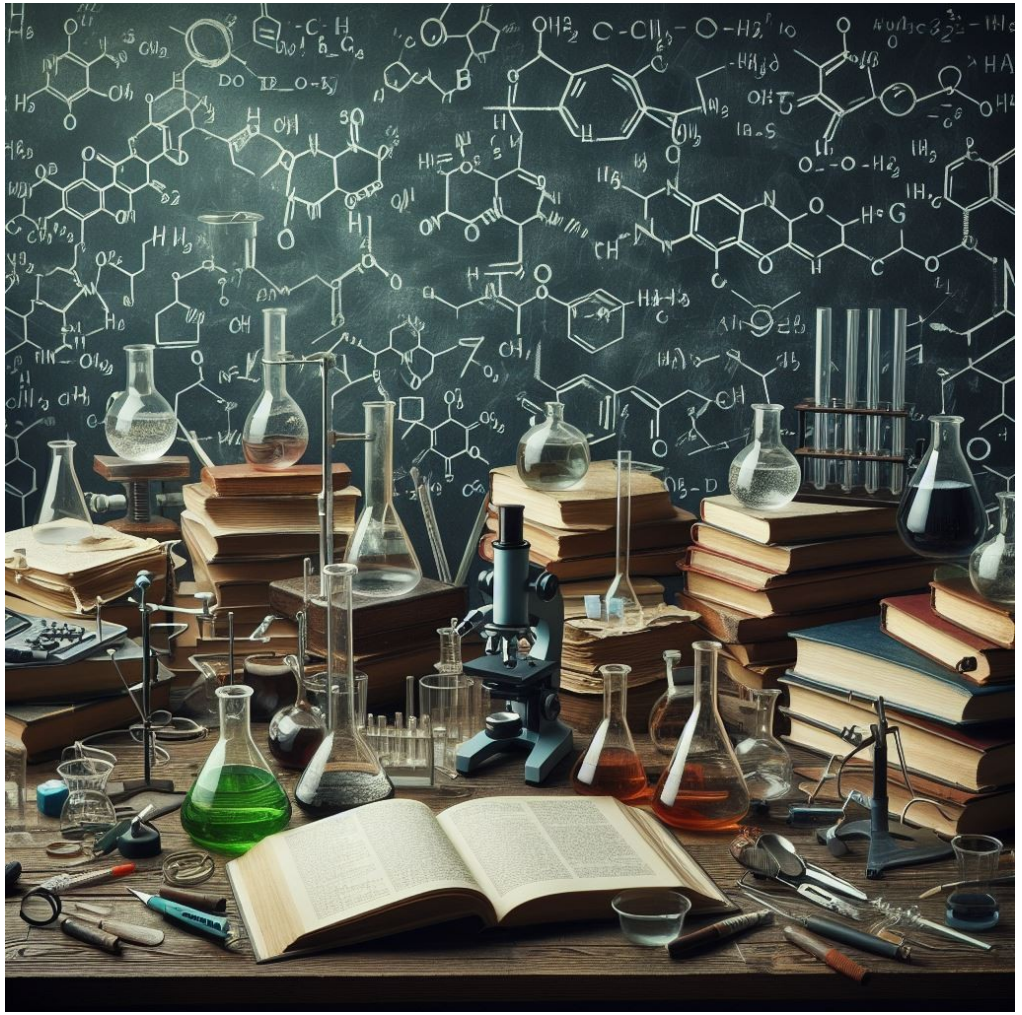
- Scientific papers
- Dive into DL (https://d2l.ai/)
- Mathematics for ML (https://mml-book.github.io/)
- Youtube videos

Materials:

- GPUs available at AI-Lab
  - https://hpc.aau.dk/ai-lab/
- PyTorch

Prerequisites:

- Linear algebra, statistics and probability theory
- Basic Python skills
- Knowledge on machine learning



Module Structure

For each module:

- Study the literature specified on Moodle prior to the lecture, and any pre-lecture assignment
  - Studies show strong correlation between learning outcome and performance and students who read before the lecture [1]
- Attend lecture in seminar room (Typically around 1.5 hours)
- Work on exercises
- Work on your mini project if time allows
- Reflect on learnings [2]

[1] Gammerdinger et al. “Understanding Student Perceptions and Practices for Pre-Lecture Content Reading in the Genetics Classroom”, JMBE 2018

[2] https://www.hubermanlab.com/episode/optimize-your-learning-and-creativity-with-science-based-tools

Mini Project

Hand-in of a completed mini-project is a prerequisite for entering the exam.

- Group based
- Deadlines on Moodle.

- More details will be given in lecture 2

Exam

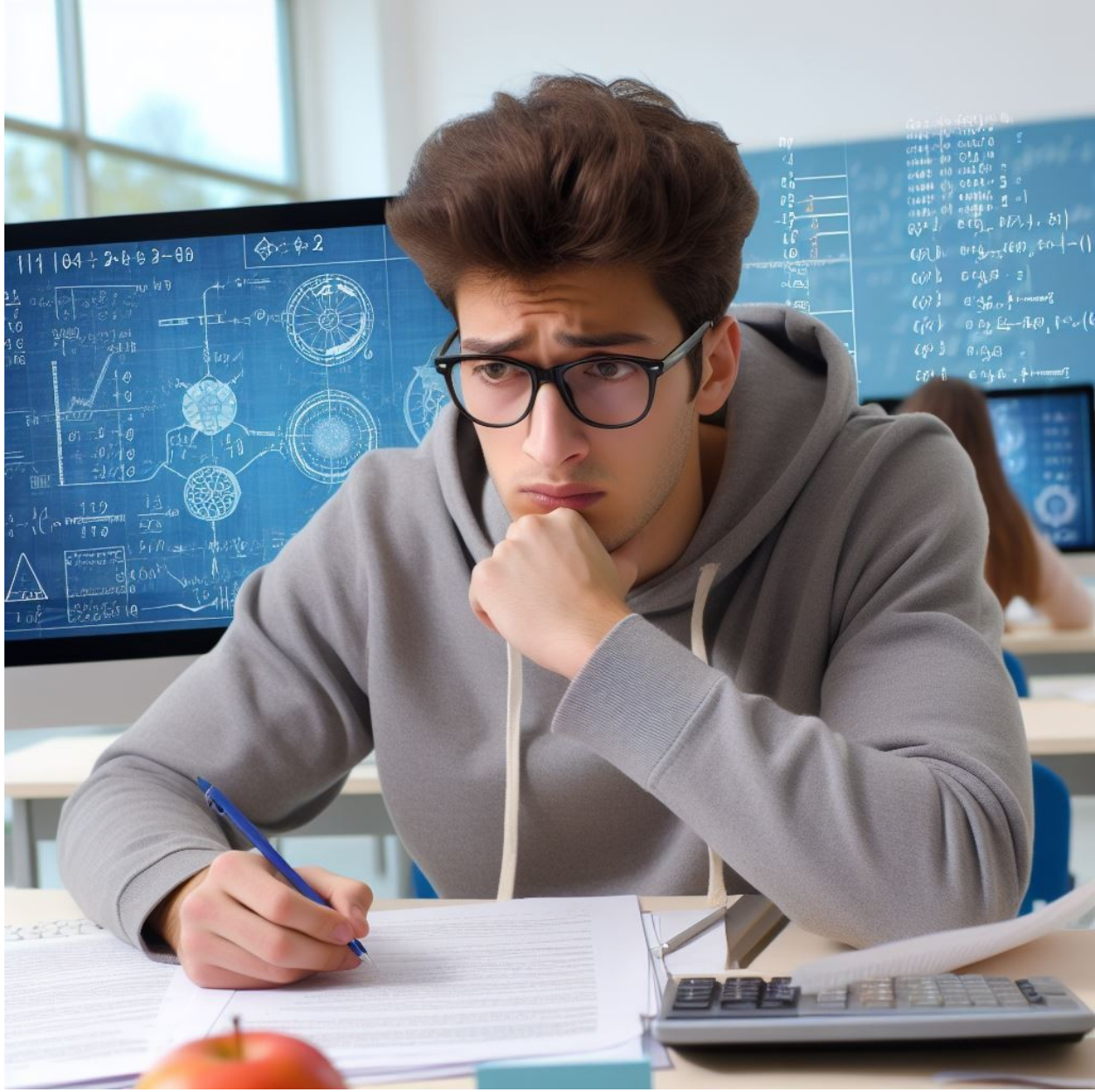
Oral examination, 20 minutes per student:

- 15 minutes examination
- 5 minutes deliberation
- Grading will be based on the 7-point scale

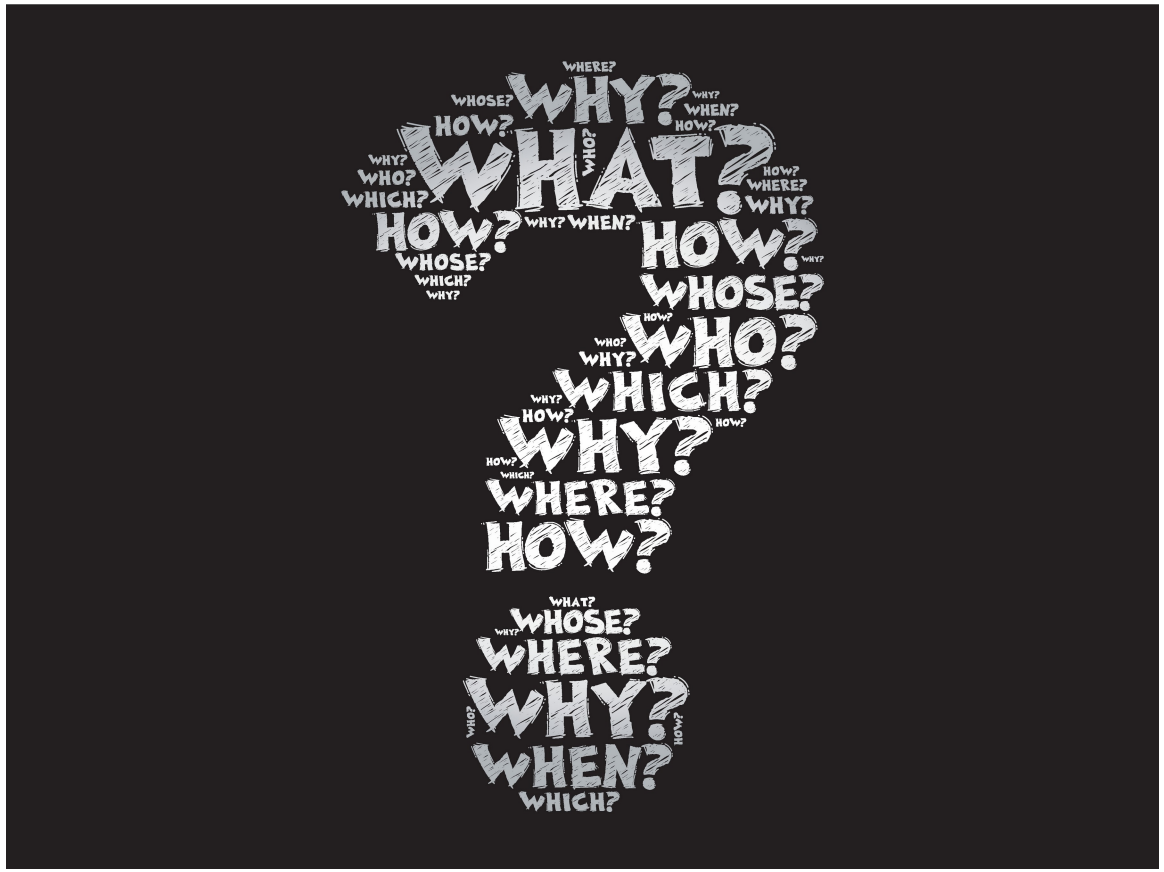
Process:

1: Present and discuss mini-project (max 5 min.)

2: Randomly choose 3 questions, from 3 separate topics (List of topics will be released). Related topics can be discussed.



Questions


WHERE?
WHOSE?
WHY?
WHY?
WHEN?
HOW?
HOW?
WHAT?
WHO?
WHY?
WHO?
WHICH?
HOW?
WHERE?
WHY?
HOW?
WHY?
WHEN?
WHOSE?
WHICH?
WHY?
HOW?
WHOSE?
WHO?
WHO?
WHY?
WHICH?
WHY?
HOW?
HOW?
WHY?
HOW?
WHICH?
WHERE?
HOW?
WHAT?
WHY?
WHOSE?
WHERE?
WHY?
WHEN?
WHICH?

Deep Learning

Spring 2026

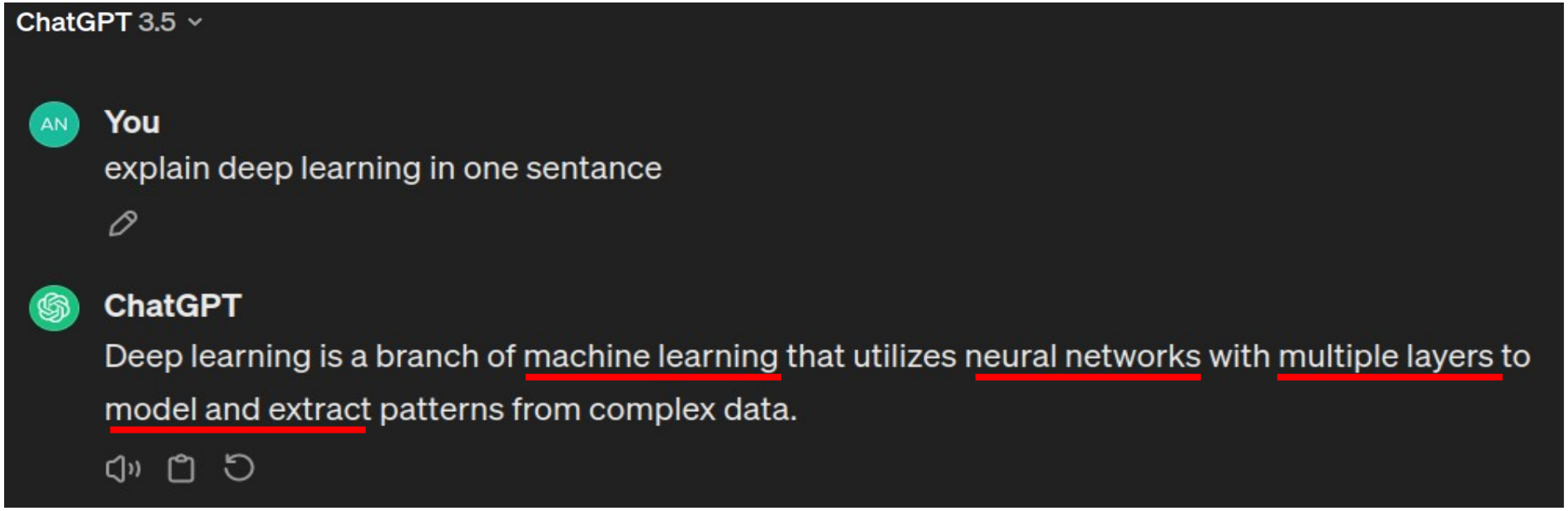
Lecture 1:
Fundamentals of Deep Learning

Andreas Aakerberg

anaa@create.aau.dk

What is Deep Learning?

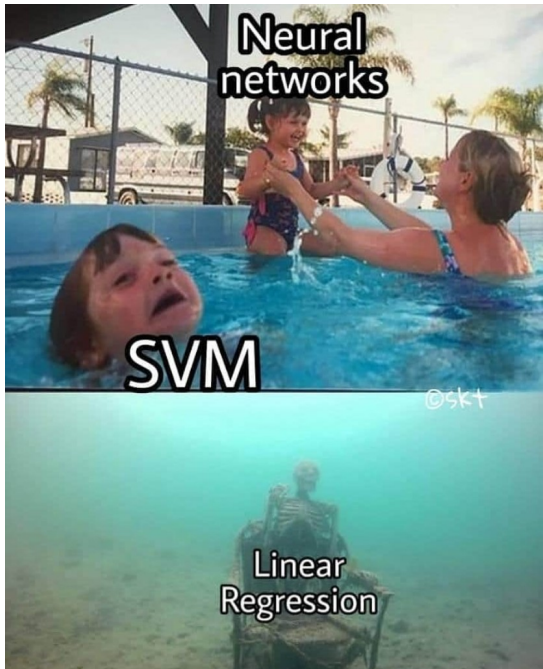
According to deep learning itself:



Google Trends: “Deep Learning”

But….. **Classical machine learning** techniques are **still relevant** and extensively used both **today** and **tomorrow**:

- Can even **outperform neural networks** (e.g. XGBoost on tabular data)
- Easier to **justify** for critical systems where **explainability** is crucial – Why?
- Often **better** when **data** is **limited**
- Suitable for **real-time**, and **tiny embedded systems** without **GPUs**, where NNs are too heavy to run



Positioning Deep Learning

Alan Turing (1950-):

- Turing Test
- Turing Machines
- Enigma codebreaking

Foundations of AI Research (1950-)

- Key concepts and methodologies in AI, such as machine learning, NLP, and computer vision and Perceptrons.

The Dartmouth Conference (1956)

- Machines that could simulate human intelligence was termed “artificial intelligence”

1950’s
Artificial Intelligence (AI)
Automate tasks that normally require human intelligence

Positioning Deep Learning

Neural Network Renaissance (1980-):

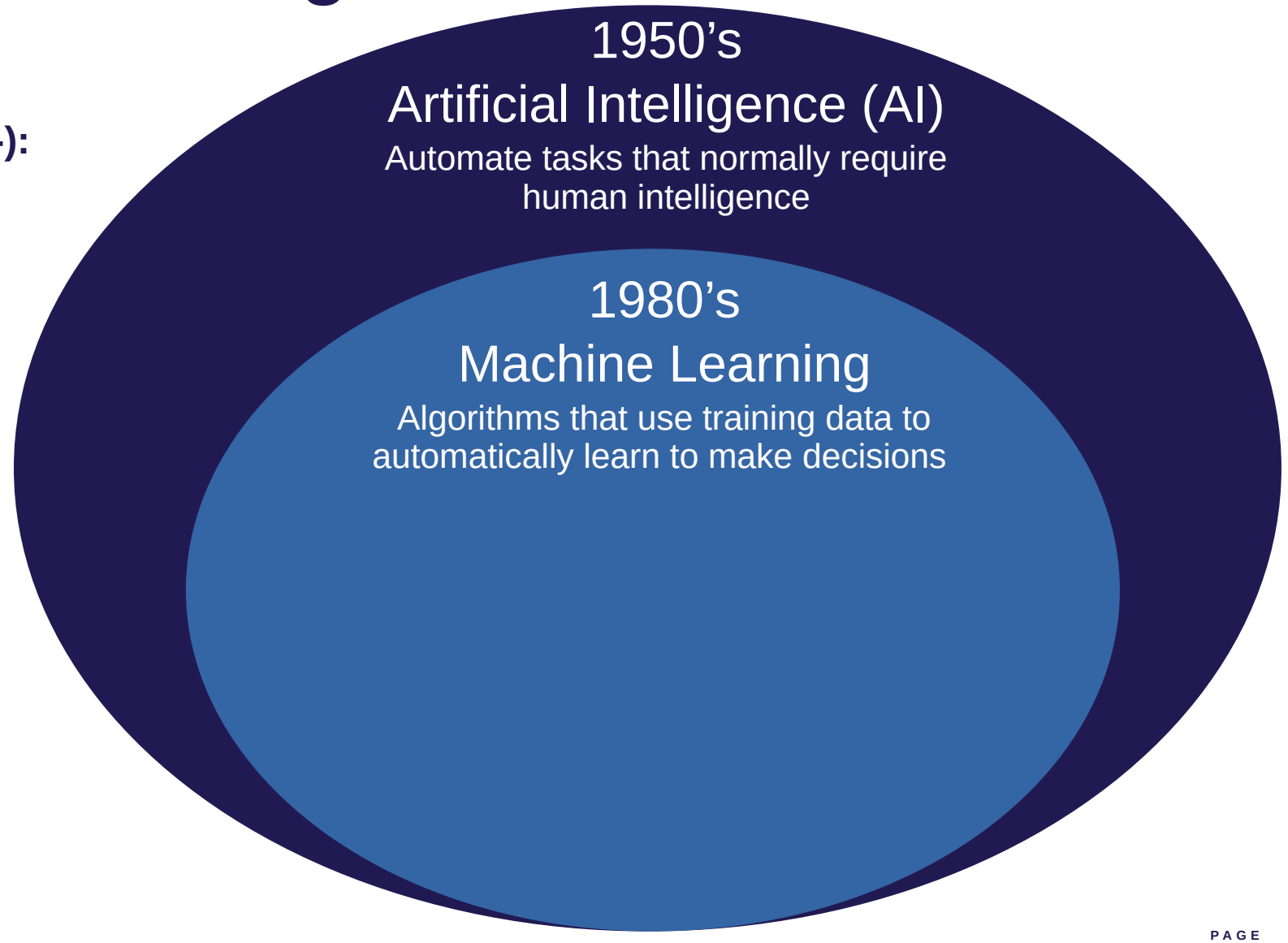
- Backpropagation to train MLPs
- Fukushima's Neocognitron
- LeCun's LeNet for digit recognition

Statistical methods and Pattern Recognition (1980-)

- Clustering, Classification, Regression
- Dimensionality reduction
- Support Vector Machines
- Decision Trees

Challenges and Limitations (1980-)

- Limited computational resources
- Data availability
- Most ML techniques was only used in research and not for practical problems



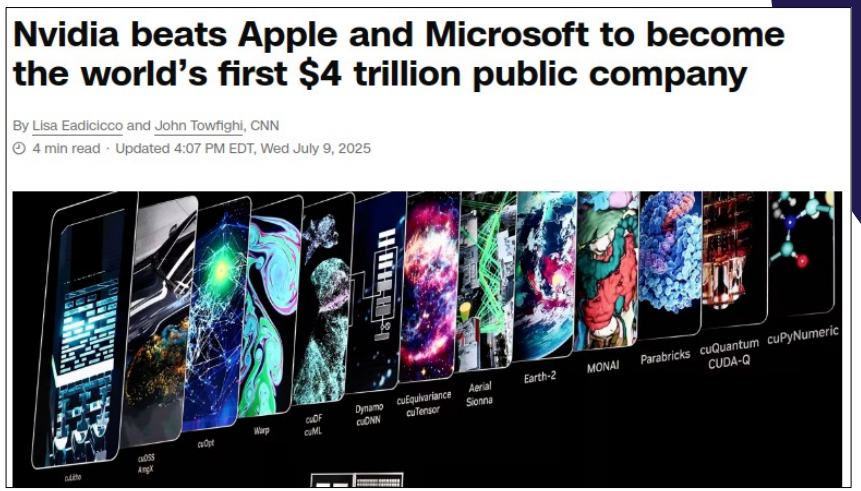
Positioning Deep Learning

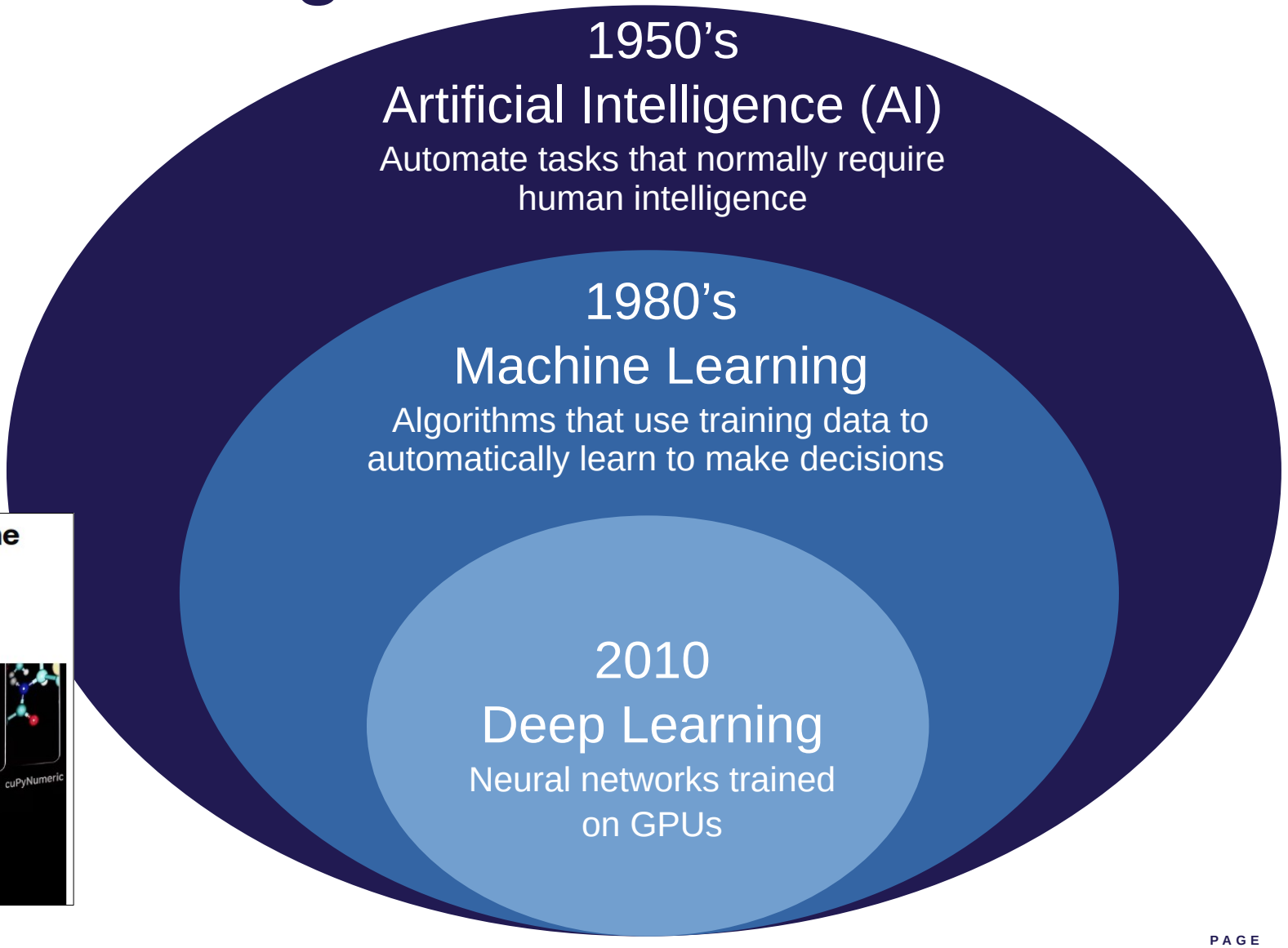
Deep Learning Revolution (2010-):

- Deep Learning beats hand-crafted methods (**AlexNet** moment)

AI in Industry and Applications

- Decision making
- Automation
- Analytics





https://edition.cnn.com/2025/07/09/investing/nvidia-is-the-first-usd4-trillion-company

Pattern Recognition with Hand-Crafted Features

Pipeline: Hand-crafted pre-processing and feature extractors combined with simple trainable classifier

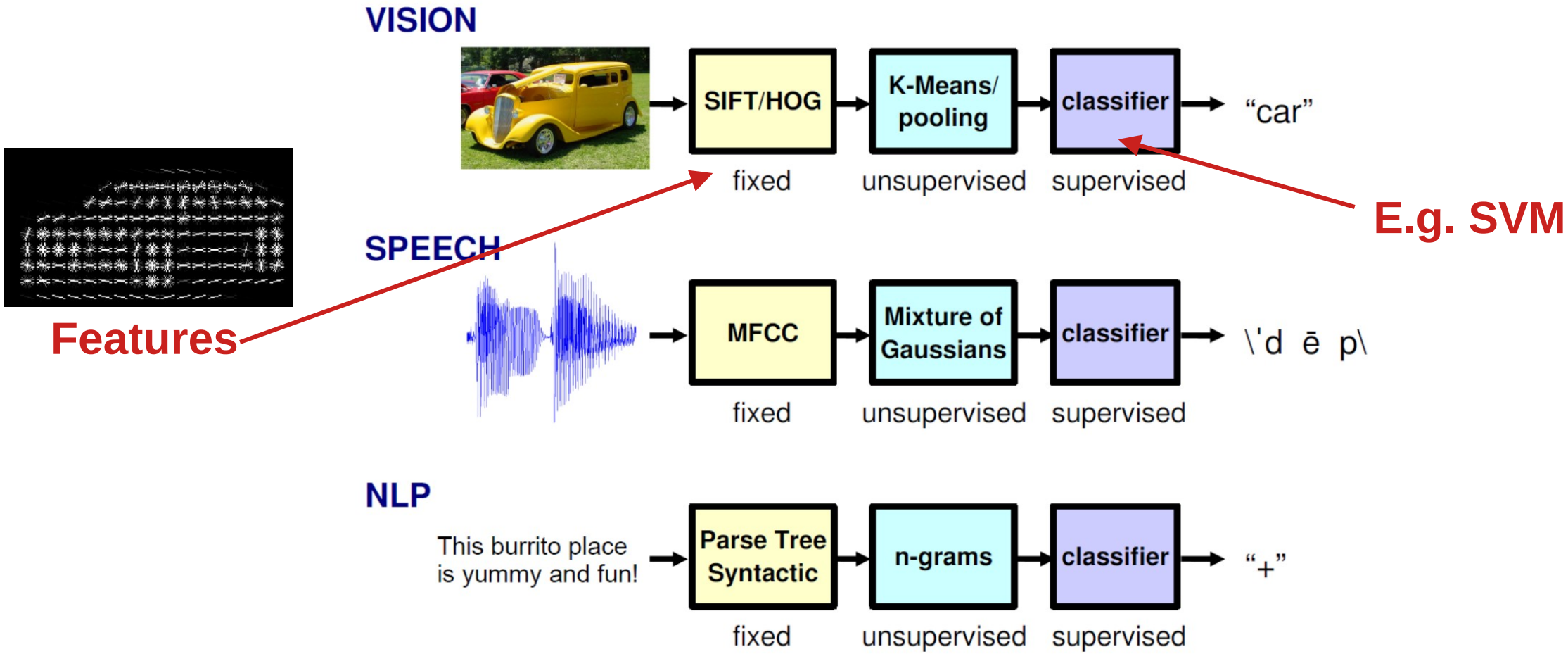
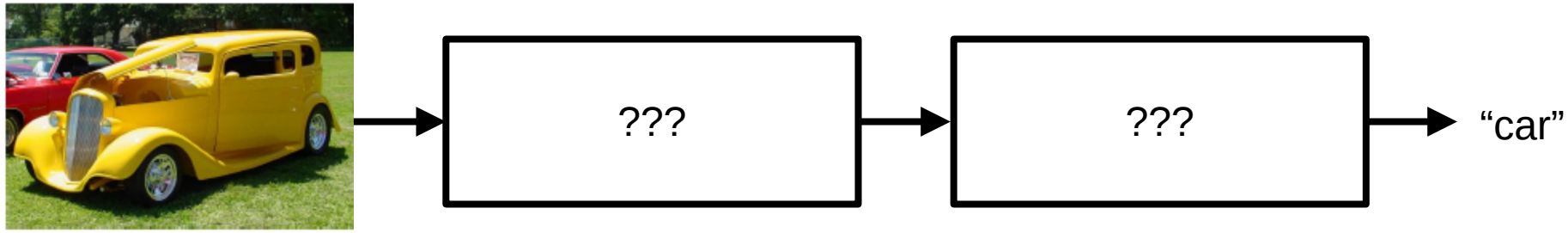


Illustration from Nojun Kwak, "Intro to CNNs", SNU 2016

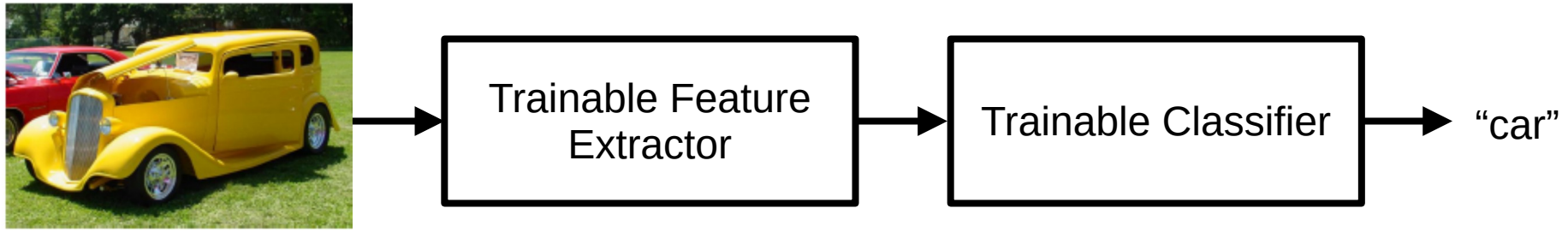
Pattern Recognition with End-to-End Learning

What if the best features could be learned automatically without human intervention?



Pattern Recognition with End-to-End Learning

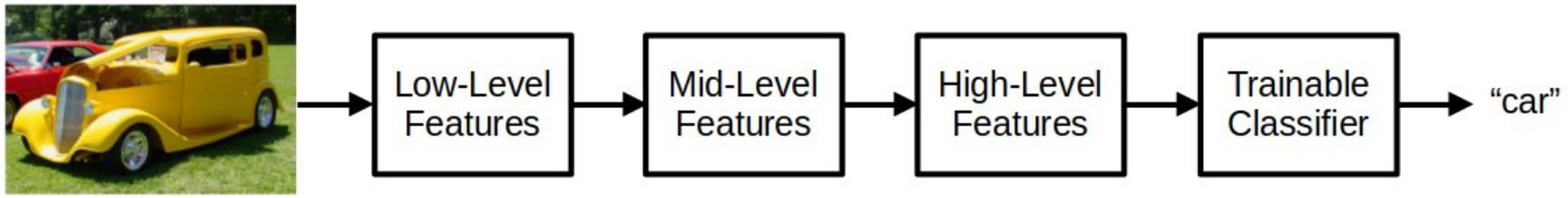
What if the best features could be learned automatically without human intervention?



End-to-end learning can be implemented with a **neural network**

Data driven approach: We learn the unknown **parameters** of the entire function f(x)=y **purely** from **data**

Learnable Feature Hierarchies

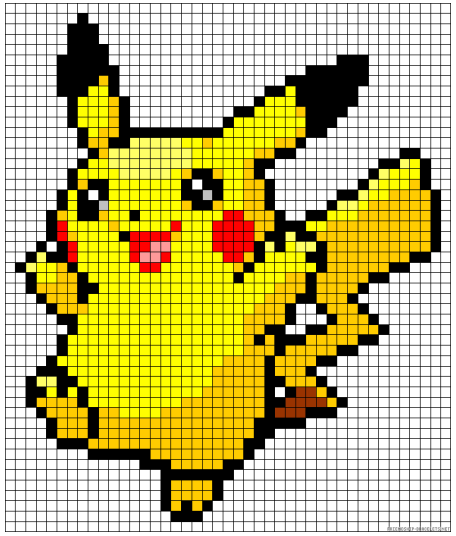


Key Properties:

- Each stage in the model is a **trainable feature transform**
- The hierarchical feature representation has **increasing level of abstraction**
- The features become increasingly **specialized in the deeper layers**

Examples of feature hierarchies:

- **Image Recognition:** Pixel → Edge → Texture → Part → Object
- **NLP:** Character → Word → Word Group → Clause → Sentence → Story



The 2012 AlexNet Moment

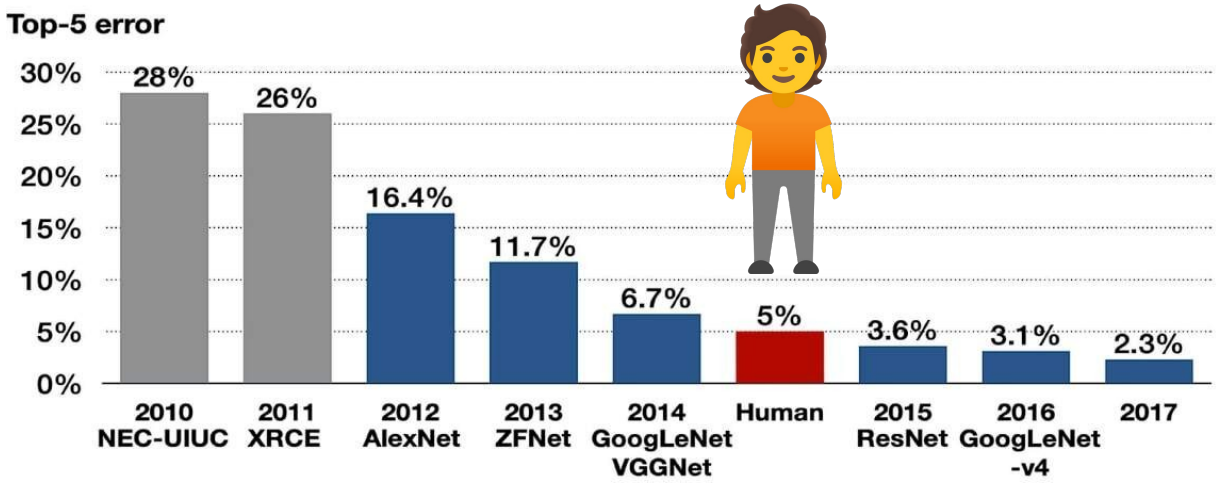
IMAGENET

Large Scale Visual Recognition Challenge 2012

Task 1: Classification

14 million images, 22k categories

Challenge subset:
1.2 million images, 1000 categories



AlexNet
Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton
University of Toronto

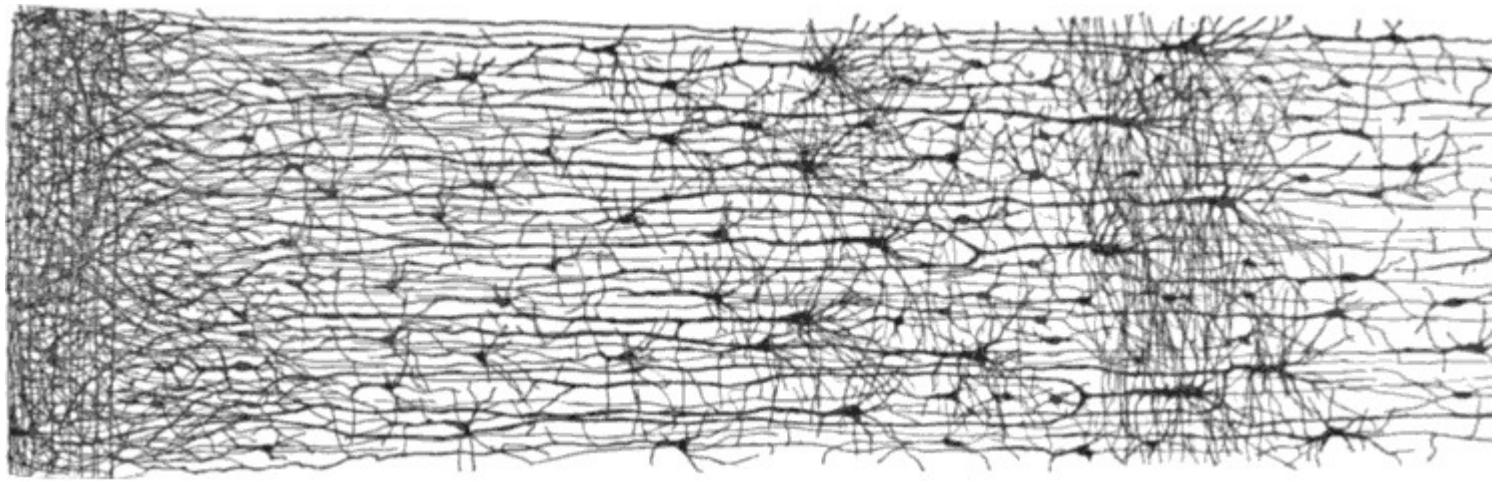
- Demonstrated efficient training of **deep neural networks** on GPUs
- Resulting in nearly halving **the error rates** of its closest competitors (More than 10% points)
- Later published as “ImageNet Classification with Deep Convolutional Neural Networks”

[1] Deng et al. “ImageNet: A Large-Scale Hierarchical Image Database”, CVPR 2009
[2] Krizhevsky, Sutskever, and Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, NeurIPS 2012

KEY QUESTION

How can we replicate the reasoning capacity of the human brain in a computer?

Biological Inspiration for Neural Nets



Mapping of the human visual cortex

The human brain:

- A neuron is the basic computational unit of the brain
- There are approx. 86 billion neurons in the human brain.
- The neurons are organized and connected in complex patterns i.e. a network
- Neurons are often organized in consecutive layers

Image source: https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch01.html

Biological Neuron

Illustration of a Biological Neuron

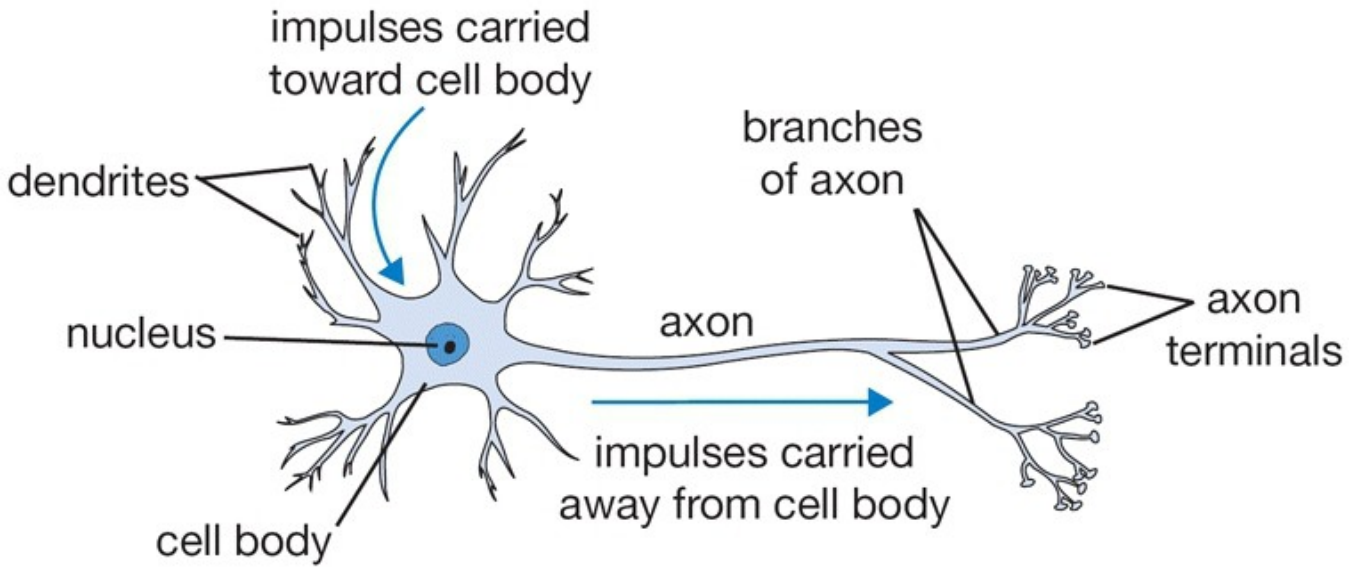
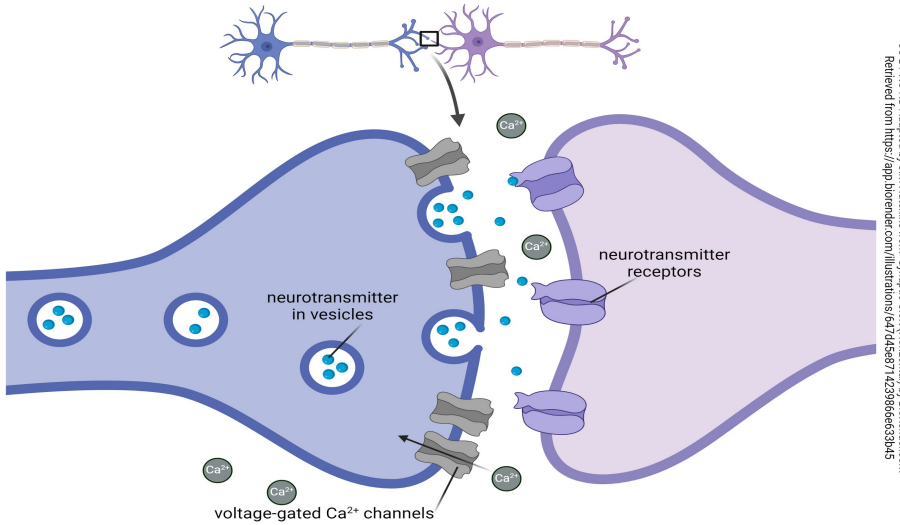


Image source: https://cs231n.github.io/neural-networks-1/
https://uen.pressbooks.pub/anatomyphysiology2/chapter/the-synapse/

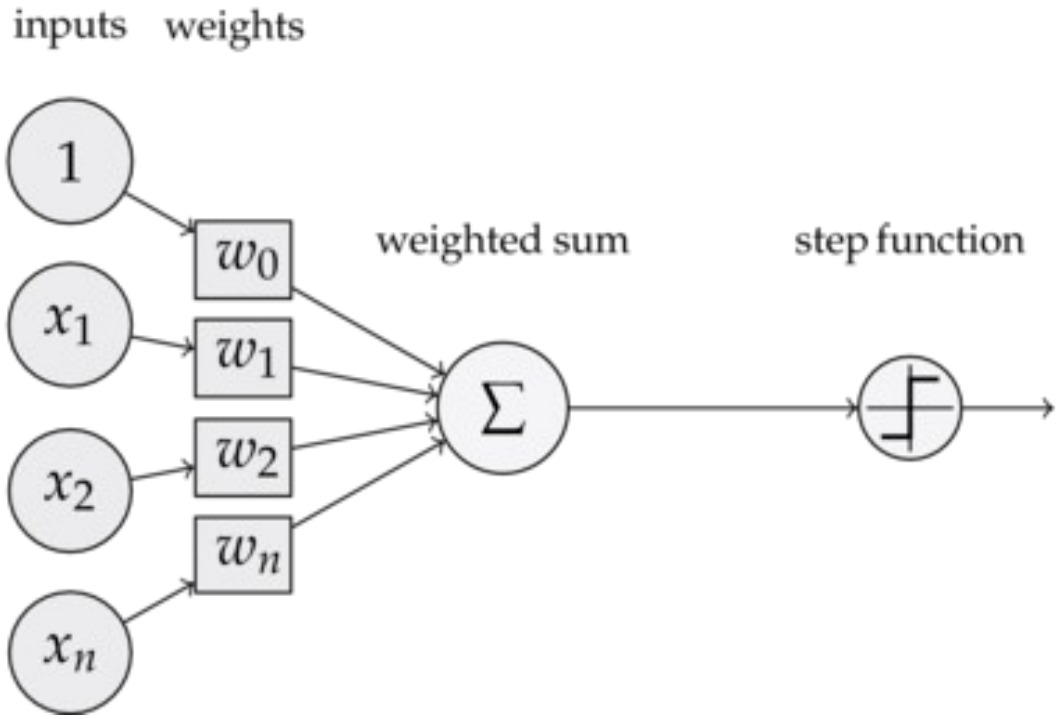
Key Properties of a Neuron:

- The basic computational unit of the brain
- Inputs are collected from **dendrites**
- The neuron **“fire”** if the total sum of inputs **exceeds a threshold**
- Output via single **axon**
- Connected to other neurons via **synapses** (More than 100 trillion)
- The strength of the connections are modifiable, also known as **synaptic plasticity**

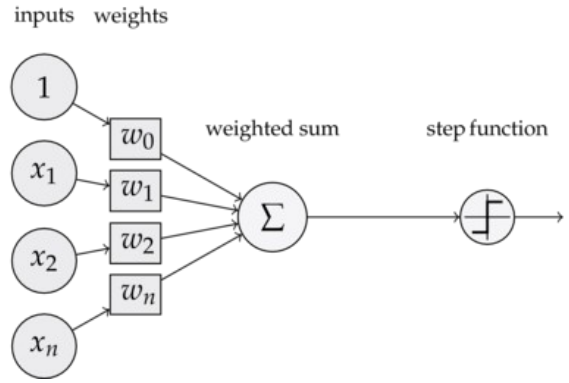


The First Artificial Neuron

- Originally presented in 1943 by **McCulloch** and **Pitts**.
- Works by **weighting** and **summing** inputs and **firing** if a **threshold** is **exceeded**
- Relied on a **step function**
- Can **only solve linearly separable** problems.
- **No learning rule**, no **automatic adjustment** of **weights**.

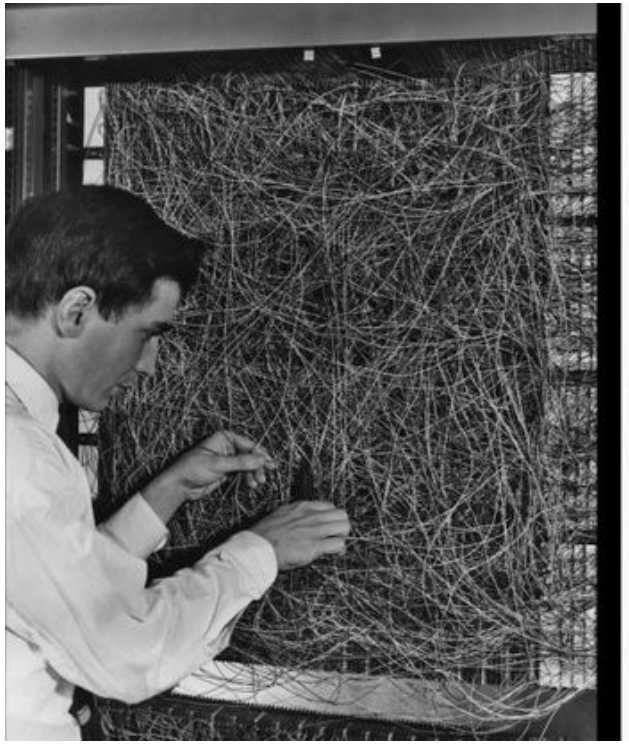


The Perceptron



Rosenblatt's Perceptron, 1957

- An **artificial neuron** similar to the **McCulloch** and **Pitts neuron.**
- A **learning algorithm** for updating a network of **Perceptrons**
- **Implementation** in **hardware.** Weights are stored in **potentiometers** that were automatically adjusted via electric motors during learning
- Connected to 20x20 **photocells** (camera)
- Used for recognizing letters of the alphabet.



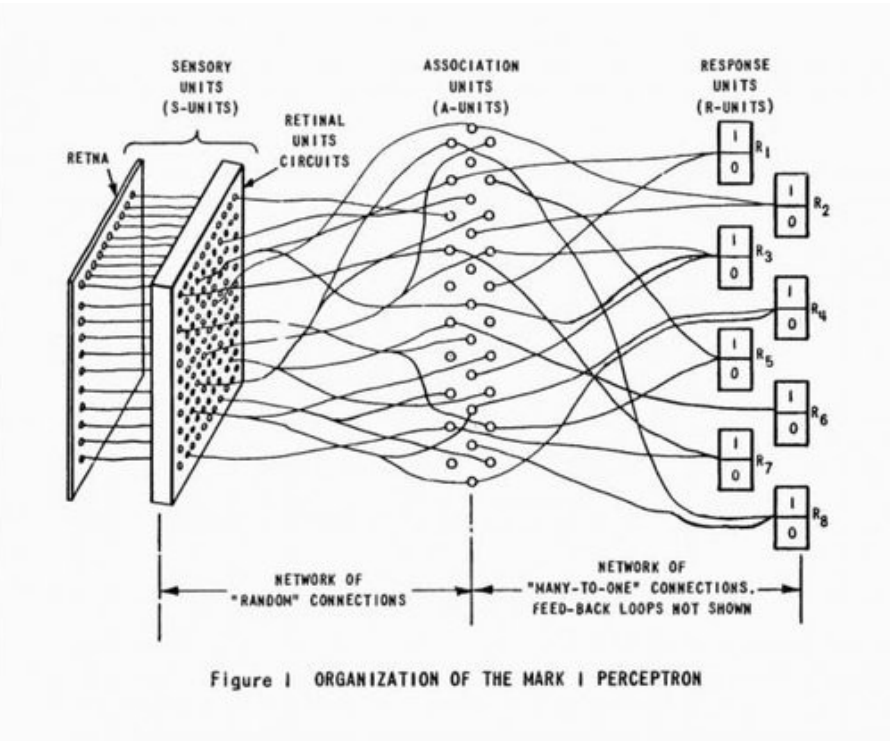
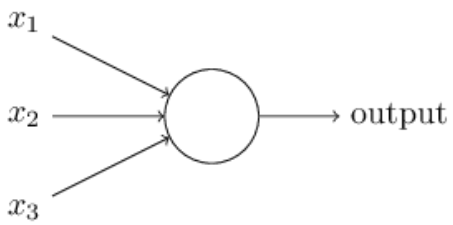


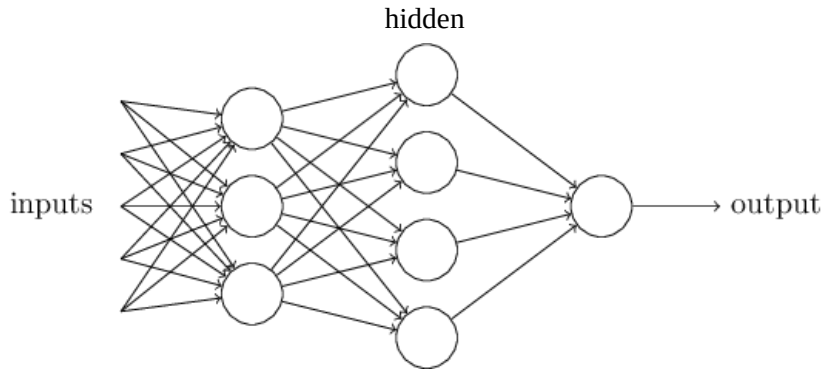
Image source: https://www.researchgate.net/figure/Frank-Rosenblatt-with-his-Mark-I-perceptronleft-and-a-graphical-representation-of_fig2_345813508

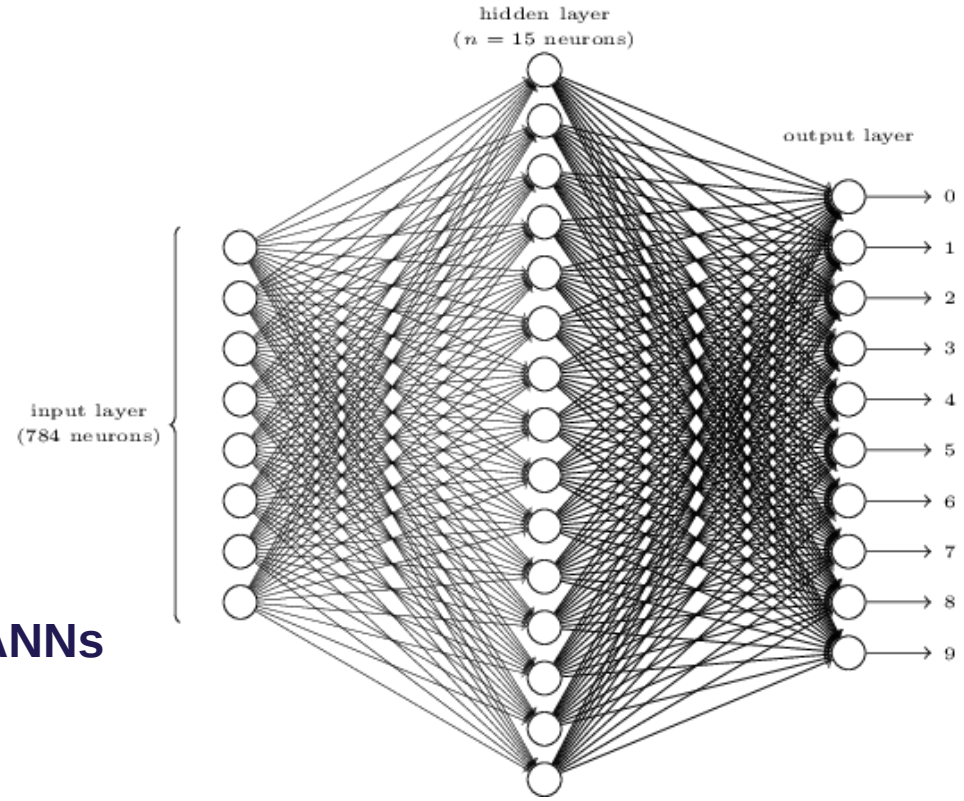
The Perceptron as Building Block

Perceptron



Multilayer Perceptron





- The **Perceptron** is the **predecessor** for **MLPs** and **ANNs**
- **Multilayer Perceptrons (MLP**s) are also called:
  Feedforward, **Fully-Connected, Dense,** or **Artificial Neural Networks**.

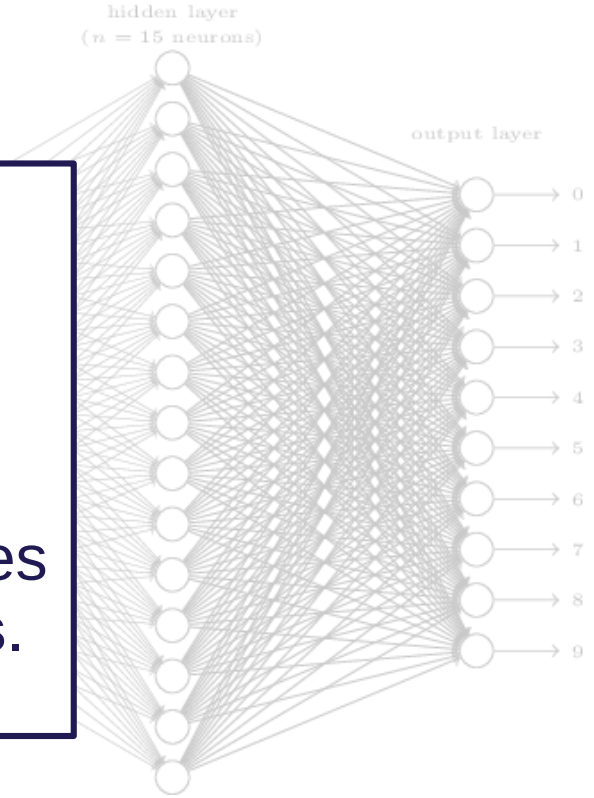
The term **Fully-Connected Neural Network** is most widely used today

Image source: http://neuralnetworksanddeeplearning.com/chap1.html

The Perceptron as Building Block

Perceptron

Multilayer Perceptron



The Perceptron have two limitations:

- The perceptron can **only** solve **linearly separable problems**
- The perceptron **optimization algorithm** does not work for **networks** with **multiple layers**.

- The **Perceptron** use
  can only be used to s

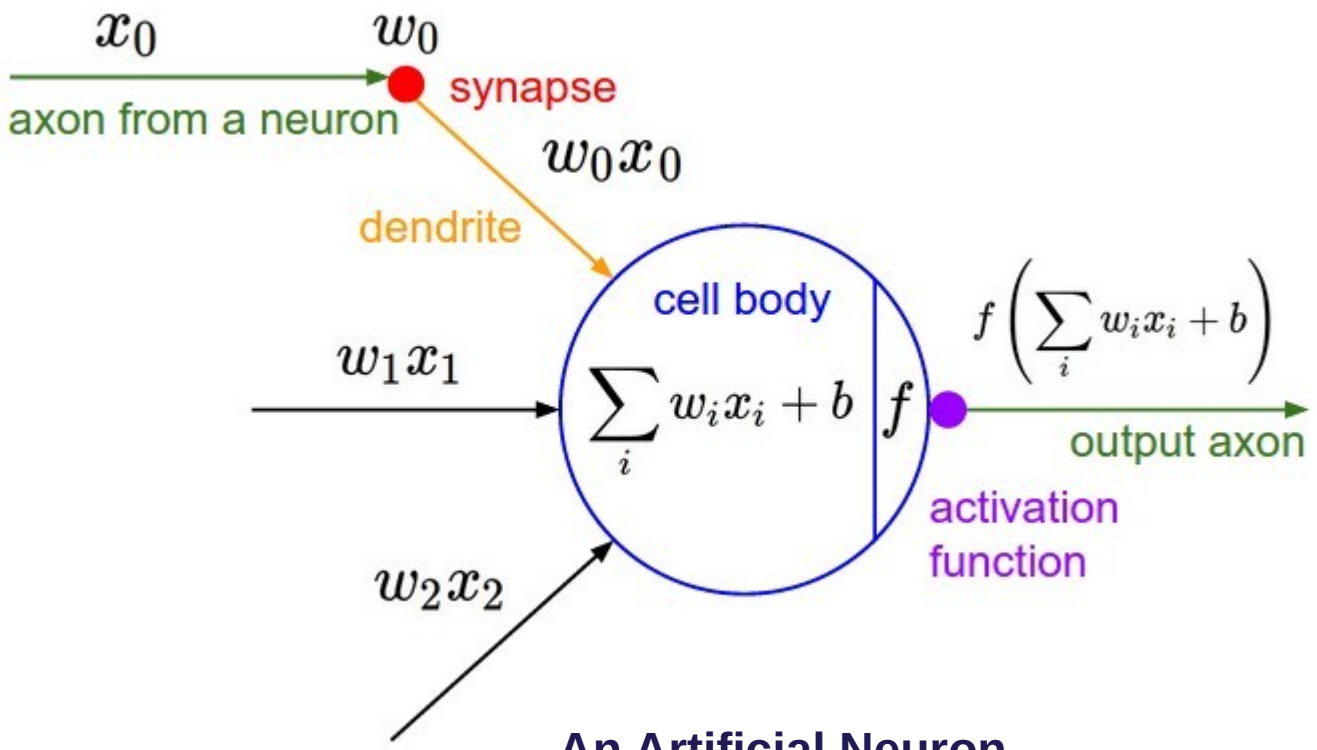
- **Multilayer Perceptrons (MLPs)** are also called:
  Feedforward, **Fully-Connected, Dense,** or **Artificial Neural Networks**.

The term **Fully-Connected Neural Network** is most widely used today

Image source: http://neuralnetworksanddeeplearning.com/chap1.html

Artificial Biological Neuron

A modernized Perceptron



An Artificial Neuron

Key Properties of an Artificial Neuron:

- Multiple inputs are received and **weighted based on importance**
- The weighted inputs are **summed and then shifted** with a single bias term
- The **firing rate** is modeled with an **non-linear activation function**
  - (traditionally a **Sigmoid**)

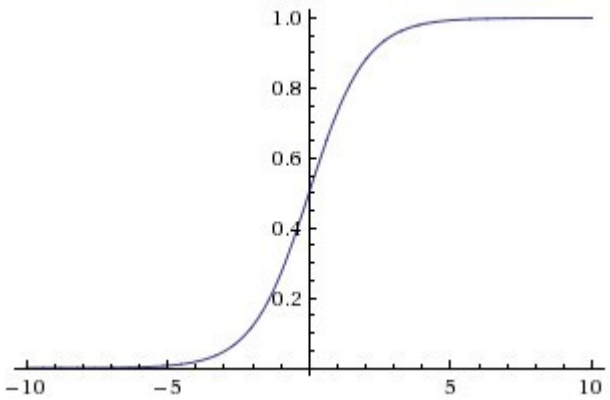


Image source: https://cs231n.github.io/neural-networks-1/

Fully Connected Neural Networks

Properties:

- **Each neuron in one layer is connected to every neuron in the next layer**
- Each **neuron** can have **direct** influence on every **output**
- Information **flows in one direction**, from input to output layer through n hidden layers
- The **depth** is determined by the **number** of **layers**
- The network is considered **deep if it has more than one hidden layer** i.e. there must be a feature hierarchy.

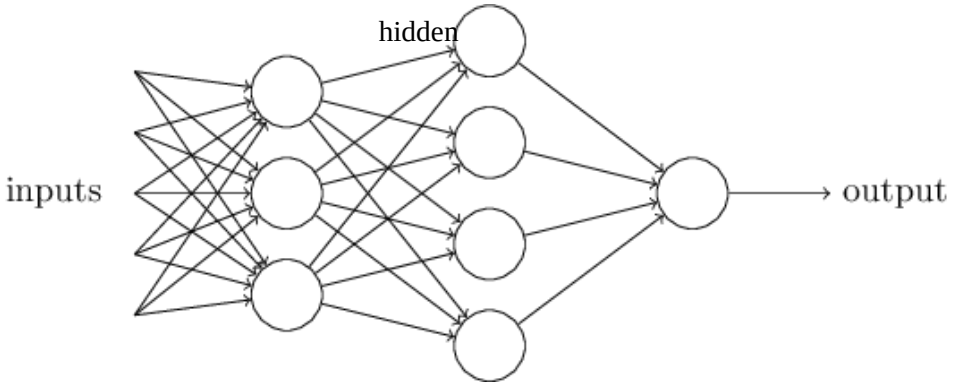


Image source: http://neuralnetworksanddeeplearning.com/chap1.html

Fully Connected Neural Networks

Properties cont.:

- Can learn **complex relationships** in data
- The learning happens by adjusting the **weights and biases** i.e. the learnable parameters
- The number of **parameters grows quadratically** with the number of **neurons**
- As the size of the input increase, so does the **computational complexity**
- **Input neurons does not perform computations** or have bias terms, while output neurons do. We can think of these as **pass-through** nodes.

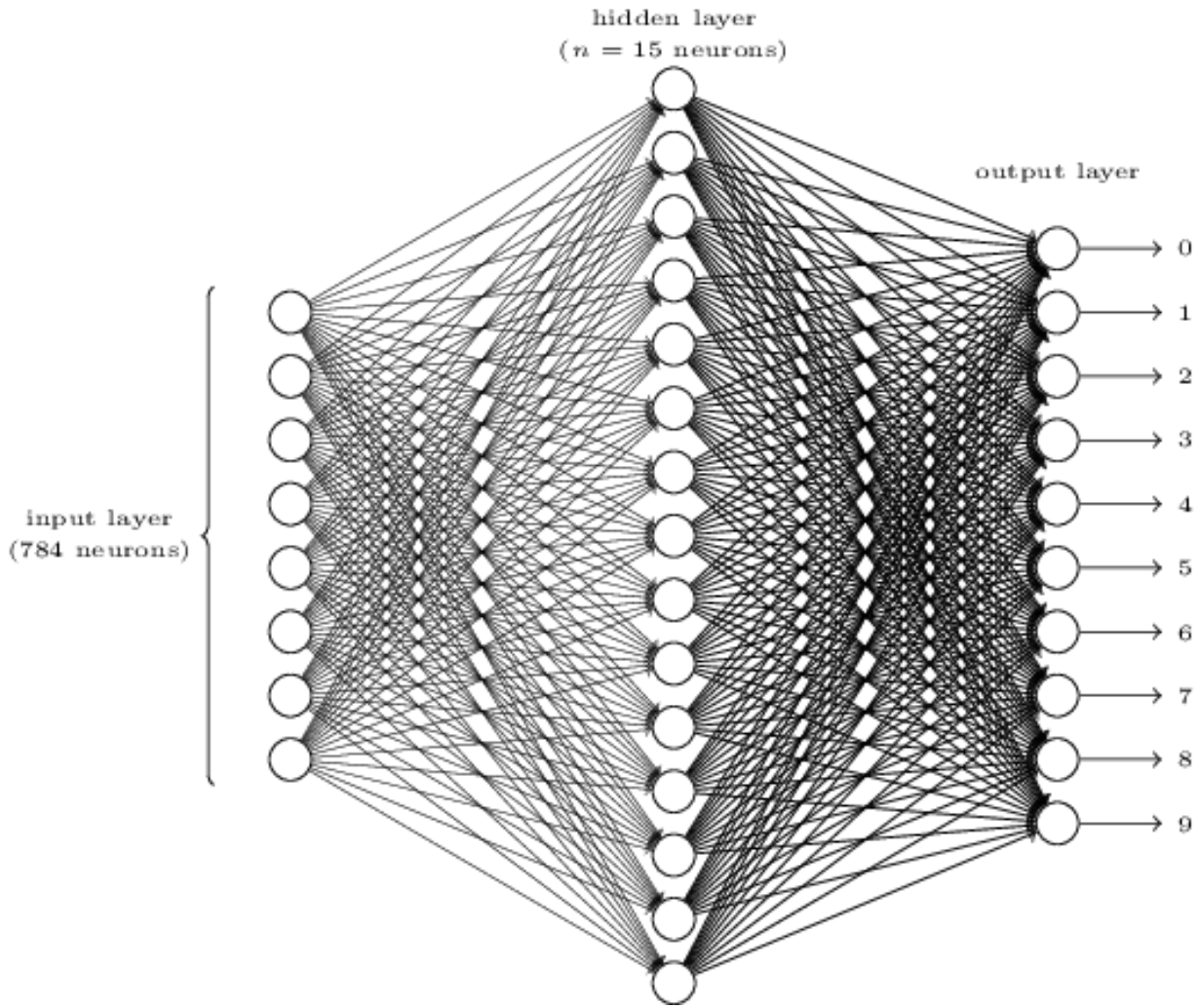
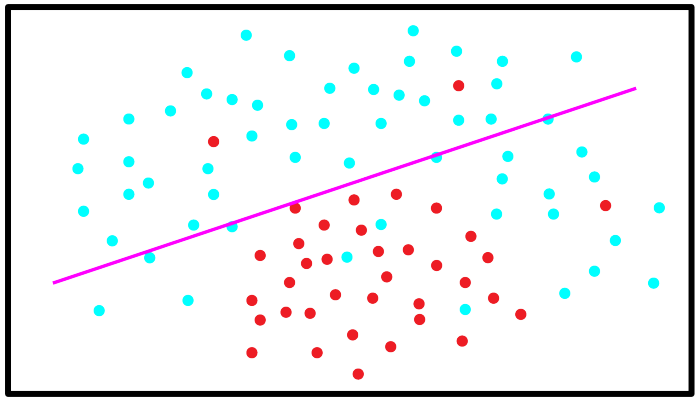


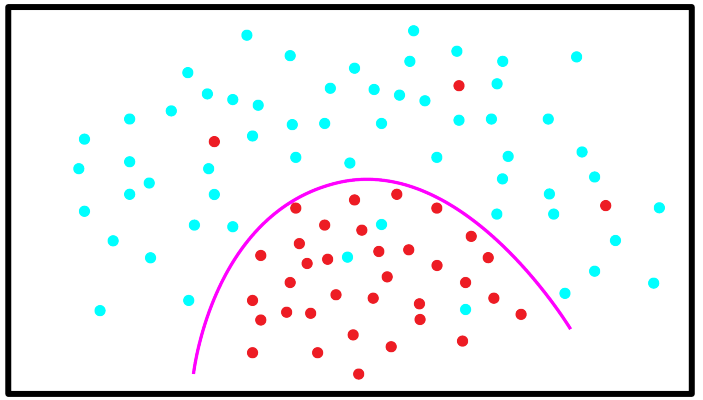
Image source: http://neuralnetworksanddeeplearning.com/chap1.html

The Need for Nonlinearity

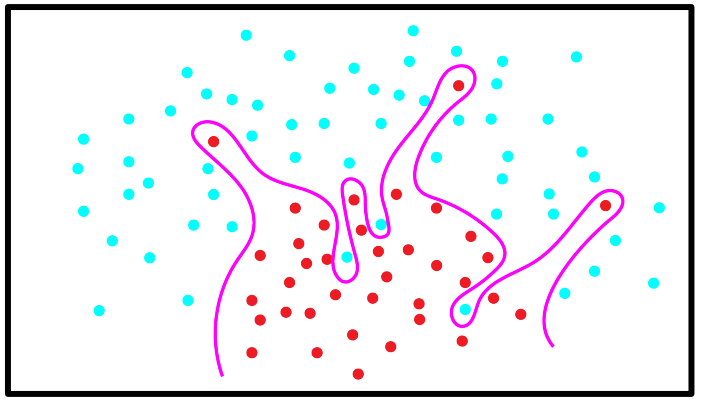
- A model based on **scaling** and **shifting operations**, defined by **weights** and **biases**, will only be able to perform **linear transformations** of the input (Weak expressive power).
- We need the model to be **more powerful** and **non-linear** (But not too powerful)



- Linear decision boundary
- The model is too simple and underfits to the distribution



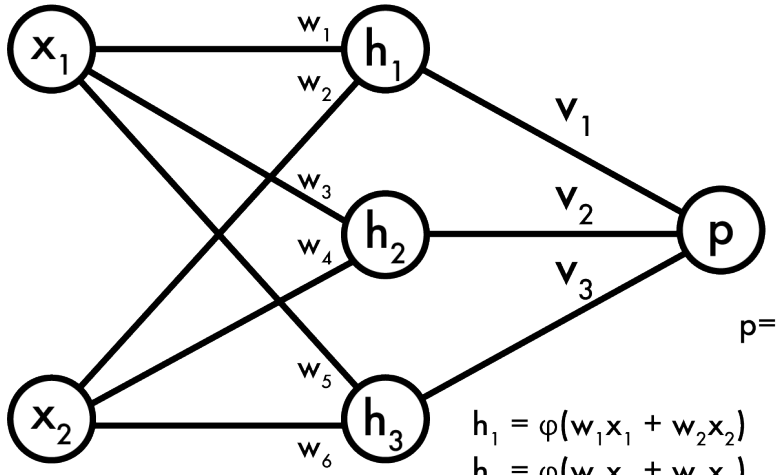
Good tradeoff?



- Complex decision boundary
- The model is too powerful and overfits to the distribution

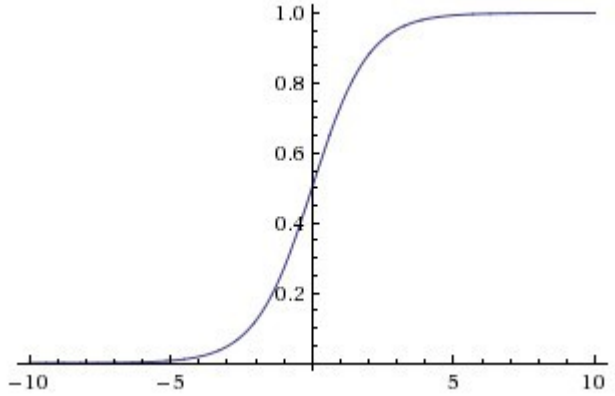
Image source: https://docs.google.com/presentation/d/1HQOET7oM3fzQba2KAcw20yEtqe_3ZLE9VTtfoPBgb8A/

Adding Activation Functions



$p = \varphi(v_1 h_1 + v_2 h_2 + v_3 h_3)$

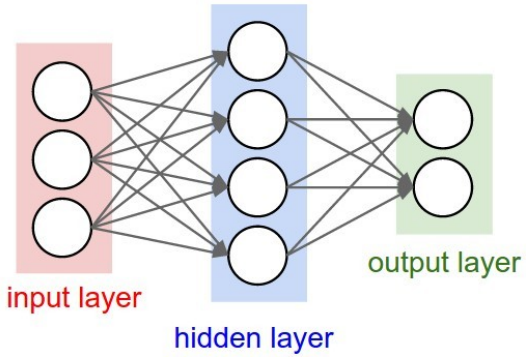
$h_1 = \varphi(w_1 x_1 + w_2 x_2)$
$h_2 = \varphi(w_3 x_1 + w_4 x_2)$
$h_3 = \varphi(w_5 x_1 + w_6 x_2)$



- **φ: An activation function which must be non-constant and differentiable.**
- P is now a result of the **activations** of **each neuron**
- The neural network is now a **non-linear function approximator**

Image source: https://docs.google.com/presentation/d/1HQOET7oM3fzQba2KAcw20yEtqe_3ZLE9VTtfoPBgb8A/

Universal Approximation Theorem



- A **feedforward neural network** with a **single** hidden layer containing a **finite number of neurons** with **activation functions** can **approximate any continuous function**.
- In **theory**, a neural network can learn **anything**!
- In **practice**, it would often require too many neurons and data.
- Other network archs, like **DNNs** and **CNNs**, exists because we do not have unlimited data and compute resources.

- **Proof**: http://neuralnetworksanddeeplearning.com/chap4.html

AALBORG UNIVERSITY

Activation Functions

- In relation to the brain **analogy**, the **activation** function **models** the **firing** mechanism of a biological neuron i.e. **should we activate or not**.
- **In practice, the activation function introduces nonlinearity to the network.**
- There are many different types of activation functions, each with their own **advantages** and **drawbacks**.

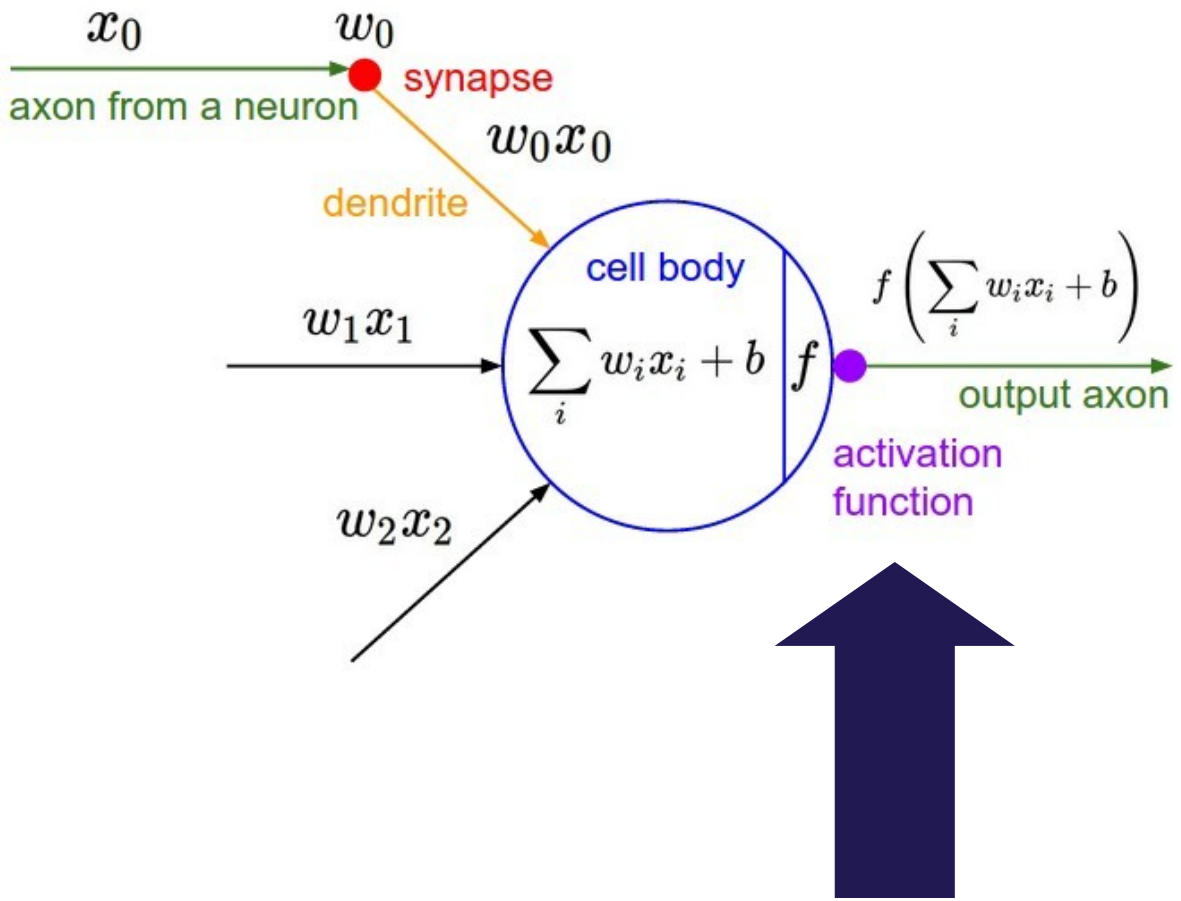
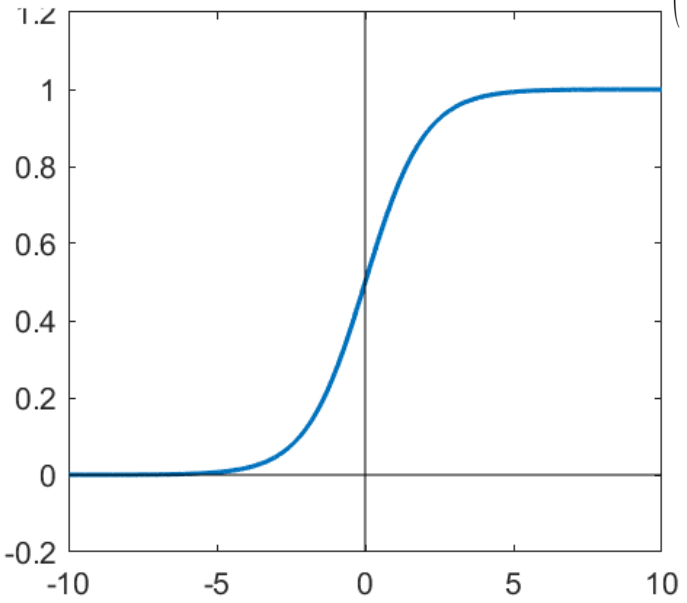


Image source: https://cs231n.github.io/neural-networks-1/

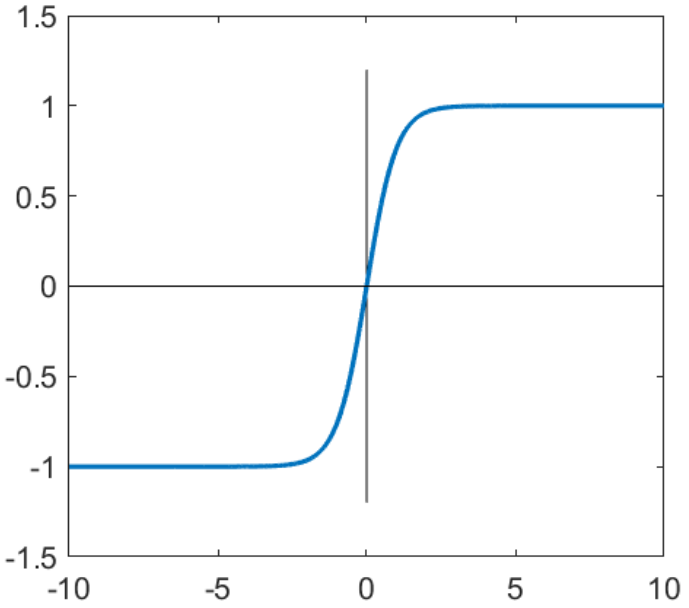
Examples of Activation Functions

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



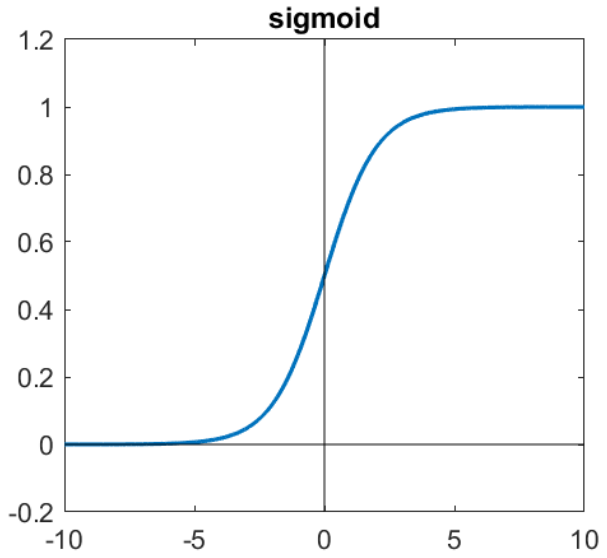
- Bound in the range [0,1]
- Prone to the vanishing gradient problem i.e. the gradient of the function approaches zero at low/high values
- Not zero-centered I.e outputs are always positive

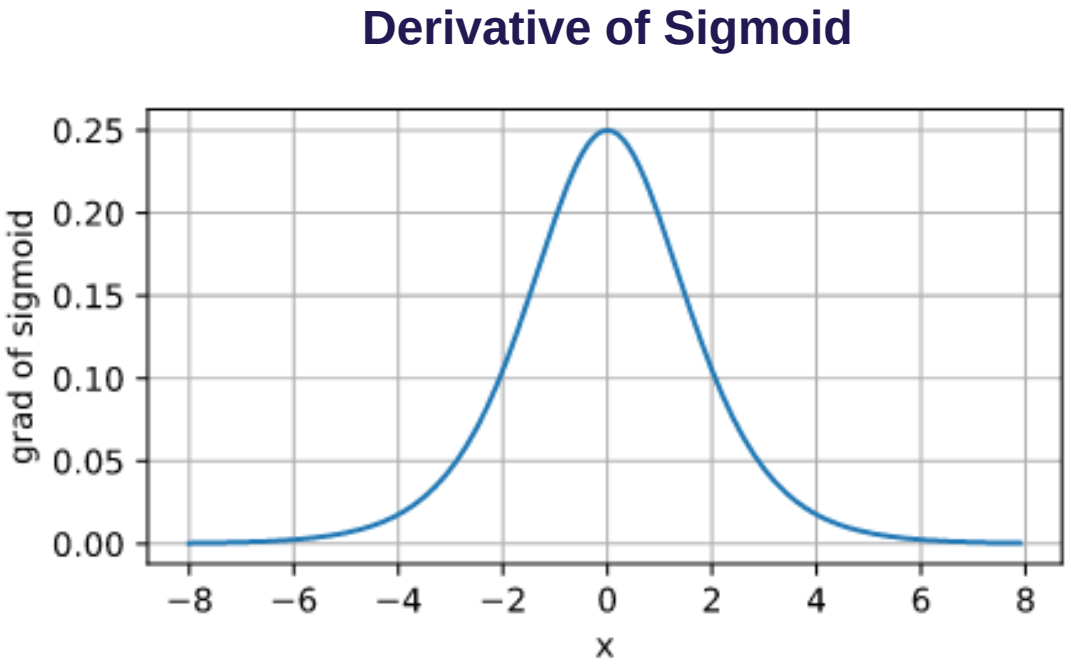
Tanh (Hyperbolic Tangent)

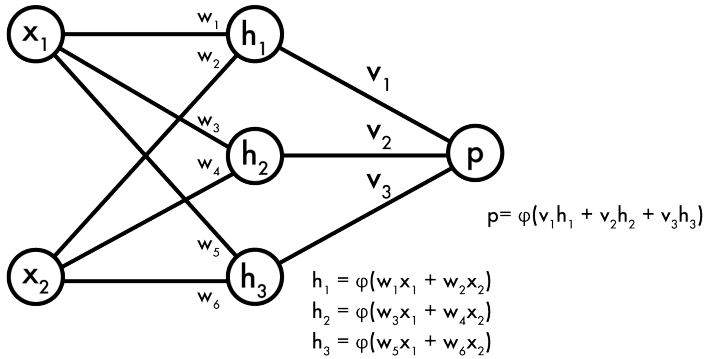


- Bound in the range [-1,1]
- Prone to the vanishing gradient problem i.e. the gradient of the function approaches zero at low/high values
- Zero-centered i.e. mean is close to zero

Vanishing Gradients





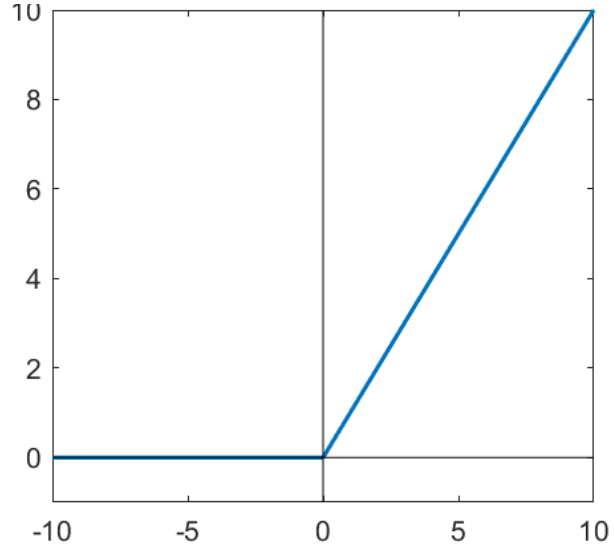


- When the input is 0, the **derivative** is 0.25
- When the input diverges from 0, the **derivative approaches 0**

- When **multiplying gradient values** of **multiple Sigmoids**, the overall product can **vanish** e.g. 0.05*0.05*0.05*0.05 = 0.00000625
- Hence the gradient might be **cut off** at some layer in the network, **hindering further optimizations** of the earlier weights

Examples of Activation Functions

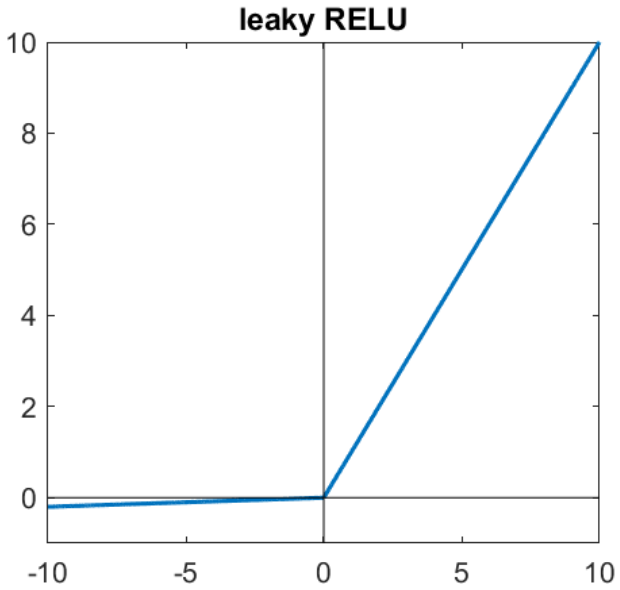
ReLU (Rectified Linear Unit) $ReLU(x) = max(0, x)$



- Thresholded to zero
- Faster convergence than Sigmoid/Tanh
- More robust against the vanishing gradient problem
- Risk of dying neurons (stuck on negative side)
- Unbounded which can cause exploding gradients

Leaky ReLU

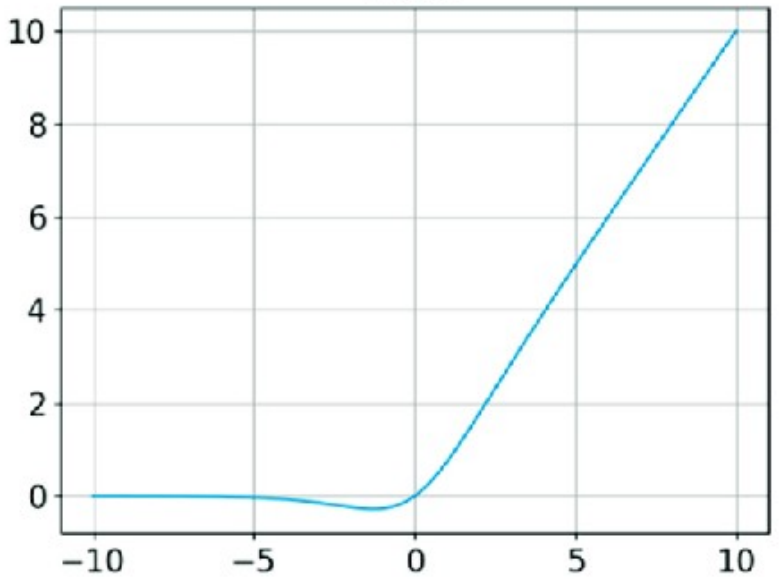
$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$



- Address the dying neuron problem
- Introduce an additional hyperparameter that needs to be tuned

Examples of Activation Functions

SiLU (Sigmoid Linear Unit) / Swish

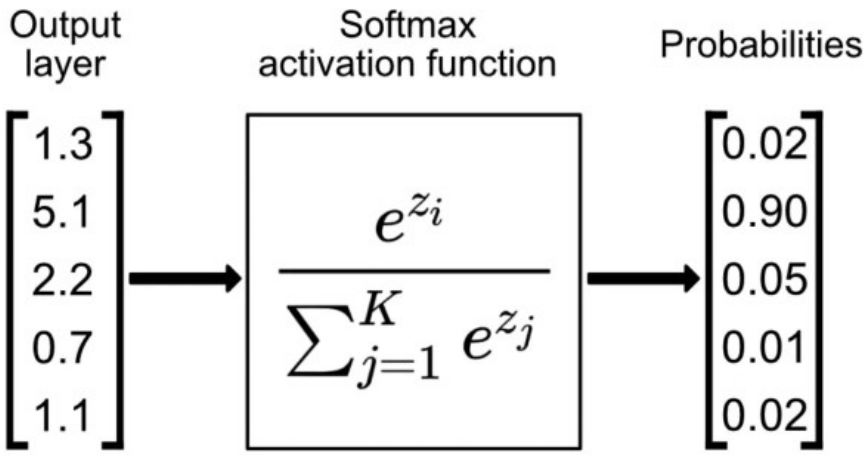


$$SiLU(x) = x \cdot \sigma(x), \text{ where } \sigma(x) \text{ is the logistic sigmoid}$$

- Self gating i.e. adaptive scaling by original input
- Sometimes better performance than ReLU and leaky ReLU
- No additional hyperparameters
- Slightly more expensive to compute than other activation functions like ReLU

Examples of Activation Functions

Softmax



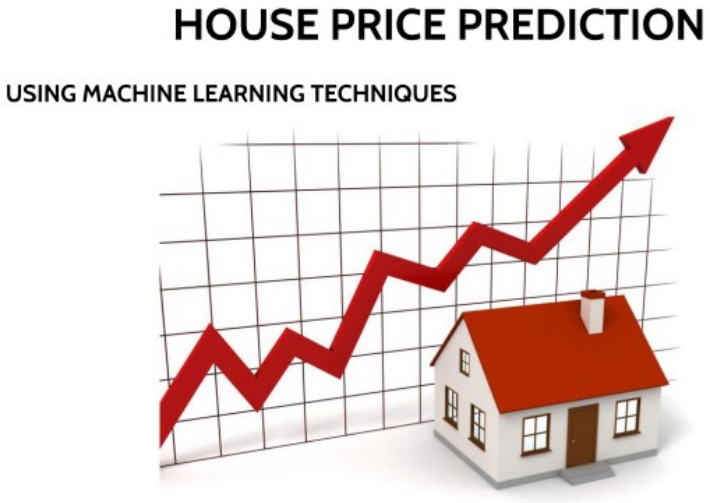
$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

- **Typically not used within the network**
- Rescales tensor elements to lie in the range [0,1] and sum to 1
- Typically used after the **last fully-connected layer** in a **multi-class classification networks**
- From **scores** to a **probability distribution** over K classes.

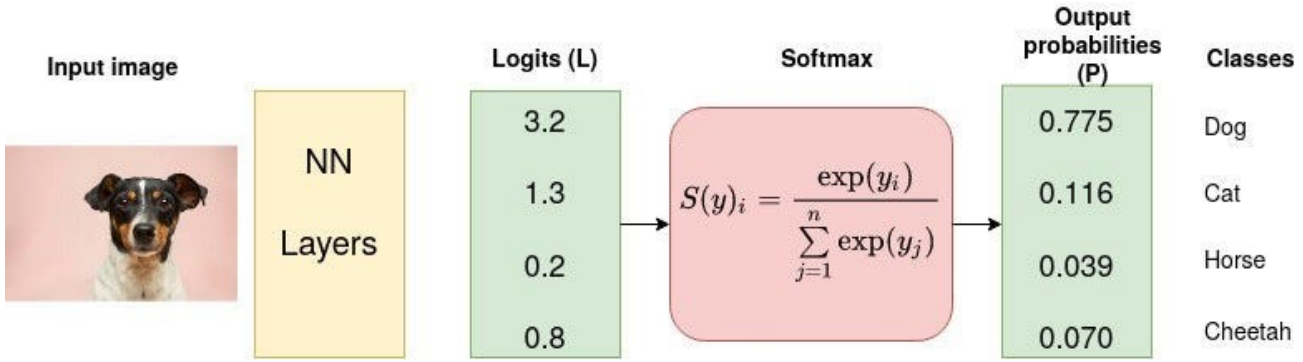
Activation Function After All Last Layers?

It depends…

- For **regression** tasks, where we want to predict a continous value, the last layer is typically kept "**linear**", as we don't want to constrain the output range (e.g. [0,1] for Sigmoid)



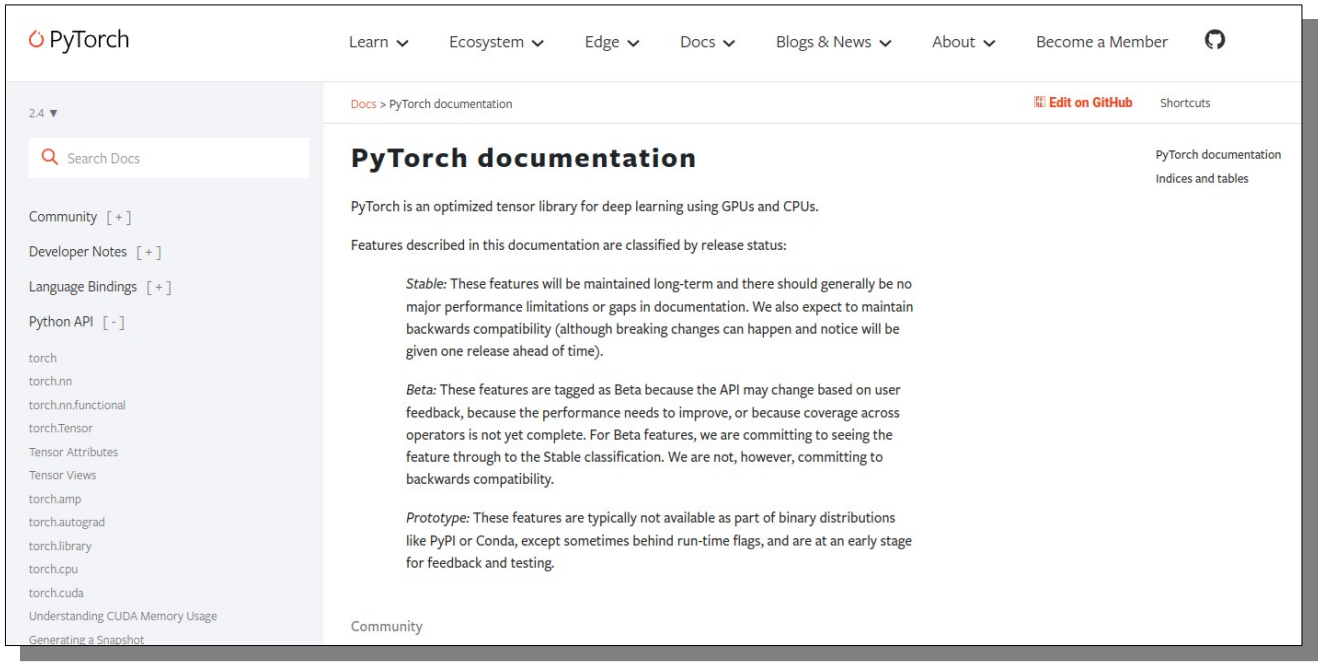
- For **classification** tasks, we typically use a **Softmax** activation to output a PDF




Micro Break

The Tool Box for Building Neural Nets

PyTorch



https://pytorch.org/docs/stable/index.html

Fully-Connected Layers in PyTorch

PyTorch **EXAMPLE**

Linear

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

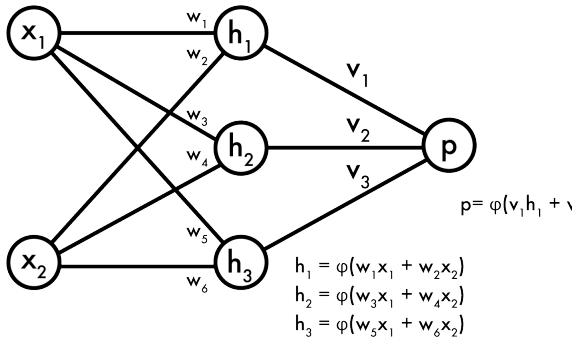
Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Corresponds to the **number** of **neurons** in the **layer** **(**It's sometimes easier to start from the end)

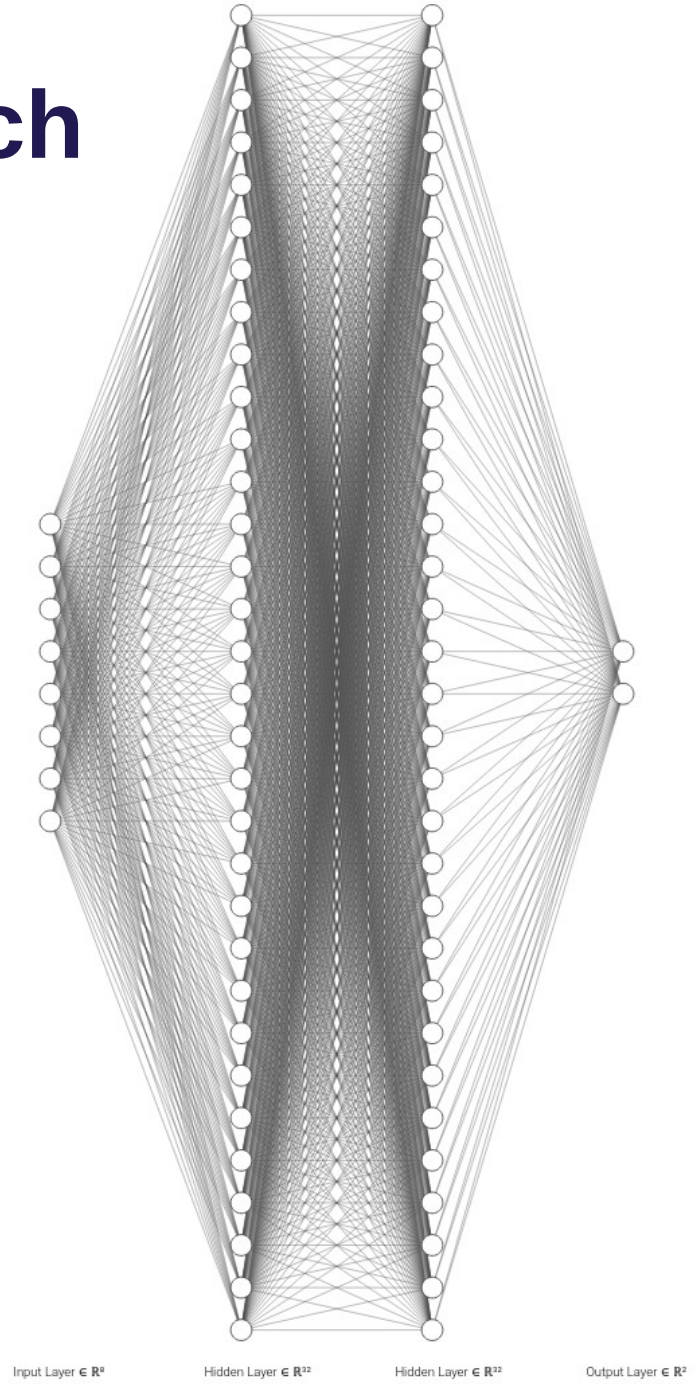


Putting it All Together in PyTorch

PyTorch **EXAMPLE**

Simple NN model definition:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

#Define feed-forward neural net class
#that inherits from nn.Module
class Net(nn.Module):
    #Define constructor
    def __init__(self):
        #Initialize the base class nn.Module
        super(Net, self).__init__()

        #Define 3 fully conncted layers
        self.fc1 = nn.Linear(8, 32)
        self.fc2 = nn.Linear(32,32)
        self.fc3 = nn.Linear(32,2)

    #Override the forward class to call
    #the layers with the input data
    def forward(self, input):
        f1 = F.relu(self.fc1(input))
        f2 = F.relu(self.fc2(f1))
        out = self.fc3(f2)

        return out
```



Putting it All Together in PyTorch

PyTorch EXAMPLE

Model definition:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

#Define feed-forward neural net class
#that inherits from nn.Module
class Net(nn.Module):
    #Define constructor
    def __init__(self):
        #Initialize the base class nn.Module
        super(Net, self).__init__()

        #Define 3 fully conncted layers
        self.fc1 = nn.Linear(8, 32)
        self.fc2 = nn.Linear(32,32)
        self.fc3 = nn.Linear(32,2)

    #Override the forward class to call
    #the layers with the input data
    def forward(self, input):
        f1 = F.relu(self.fc1(input))
        f2 = F.relu(self.fc2(f1))
        out = self.fc3(f2)

        return out
```

Output from interactive Python session:

```
>>> import torchsummary
>>> net = Net()
>>> data = torch.randn(8)
>>> res = net(data)
>>> res
tensor([-0.0052,  0.0271], grad_fn=<ViewBackward0>)
>>> torchsummary.summary(net, input_size=(1,8))
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 32]             288
            Linear-2               [-1, 1, 32]           1,056
            Linear-3                [-1, 1, 2]              66
================================================================
Total params: 1,410
Trainable params: 1,410
```

Putting it All Together in PyTorch

PyTorch **EXAMPLE**

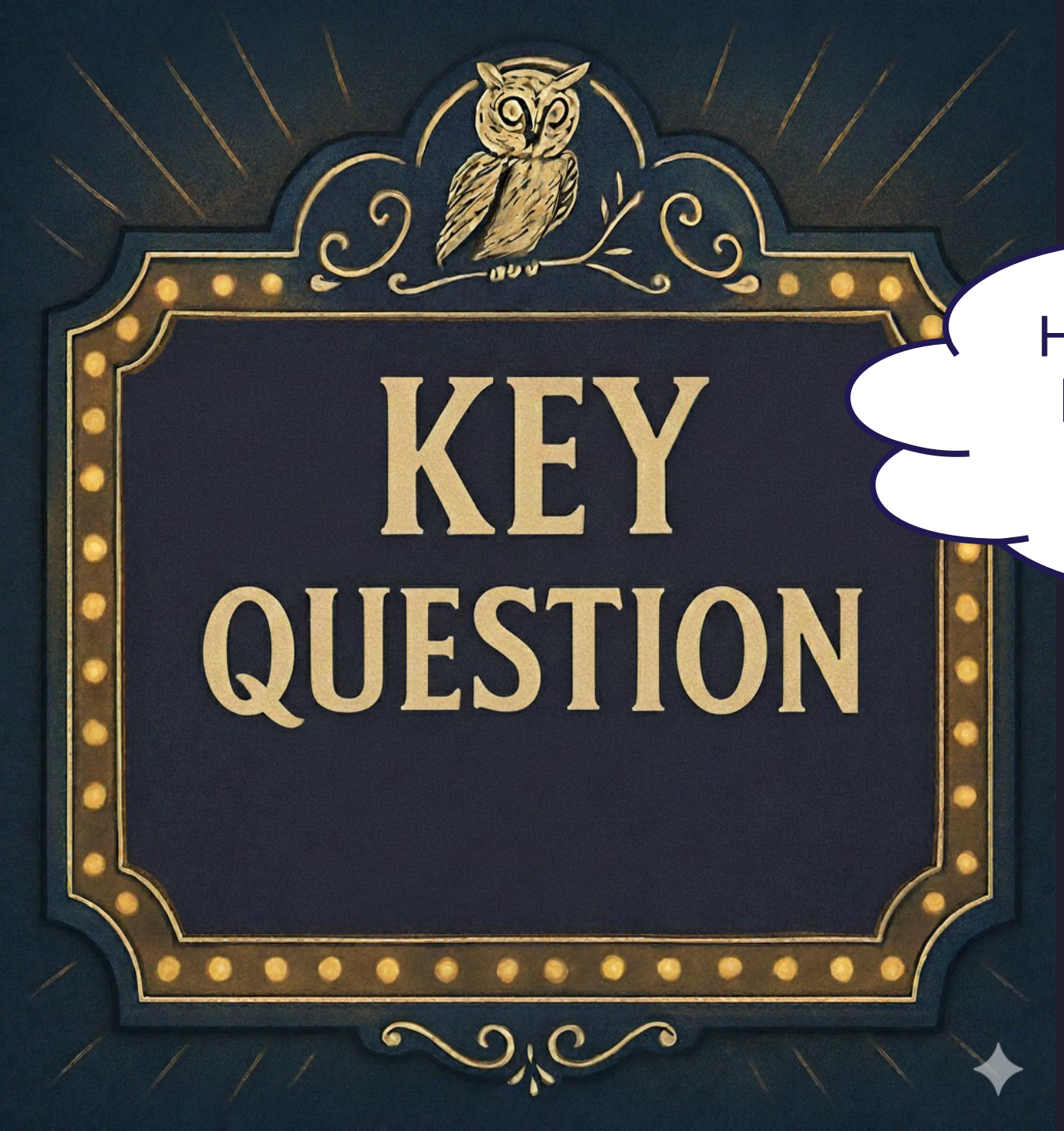
Model definition:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

#Define feed-forward neural net class
#that inherits from nn.Module
class Net(nn.Module):
    #Define constructor
    def __init__(self):
        #Initialize the base class nn.Module
        super(Net, self).__init__()

        #Define 3 fully conncted layers
        self.fc1 = nn.Linear(8, 32)
        self.fc2 = nn.Linear(32,32)
        self.fc3 = nn.Linear(32,2)

    #Override the forward class to call
    #the layers with the input data
    def forward(self, input):
        f1 = F.relu(self.fc1(input))
        f2 = F.relu(self.fc2(f1))
        out = self.fc3(f2)

        return out
```

Is this correct?

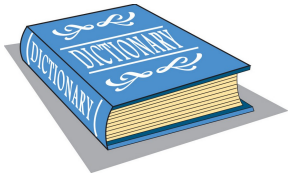
Output from interactive Python session:

```
>>> import torchsummary
>>> net = Net()
>>> data = torch.randn(8)
>>> res = net(data)
>>> res
tensor([-0.0052,  0.0271], grad_fn=<ViewBackward0>)
>>> torchsummary.summary(net, input_size=(1,8))
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 32]             288
            Linear-2               [-1, 1, 32]           1,056
            Linear-3                [-1, 1, 2]              66
================================================================
Total params: 1,410
Trainable params: 1,410
```



How can we process grid-like data (e.g. an image) with a neural network?

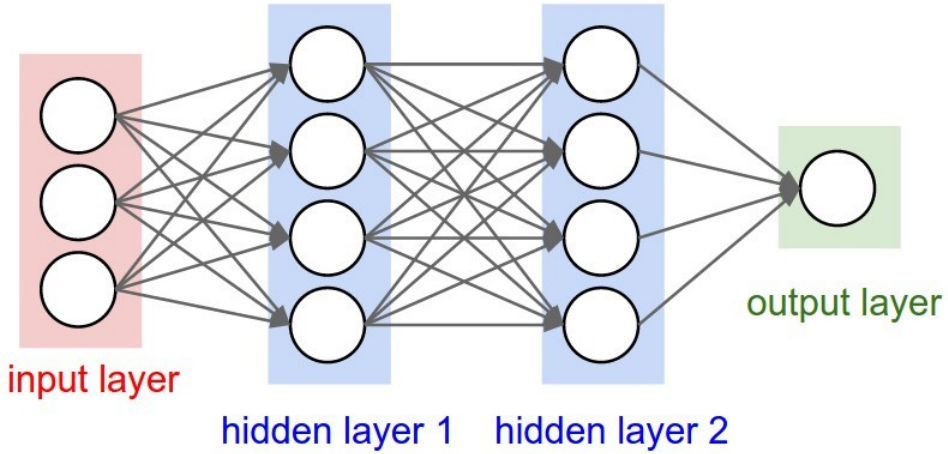
Terminology in Deep Learning

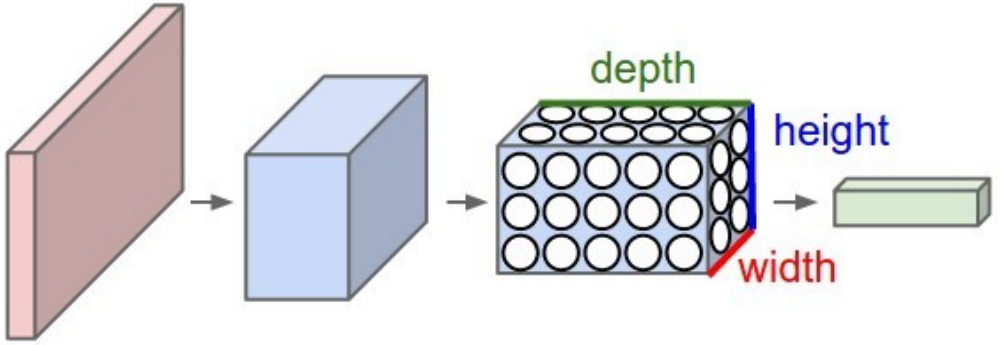


- Artificial Neural Network (ANN), Neural Network (NN), Fully-connected Net (FCN), Feedforward NN, Multi-layer-perceptron (MLP)
  - Dense layer, fully connected layer
- Activation function, non-linearity
- Weights, parameters, synapse
- **Convolutional Neural Network (CNN), Convnet**
  - **Conv layer, convolutional layer**
- **Kernel, Filter**
- **Feature map, activation map, output map**
- **Pooling, downsampling, subsampling layer**

Note that while many of these terms are used interchangebly, they are technically not the same!

From Neurons to Tensors





Scalars and vectors

1

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix},$$

Matrices in multiple dimensions

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

Tensors are containers describing multidimensional arrays

Image source: https://cs231n.github.io/convolutional-networks

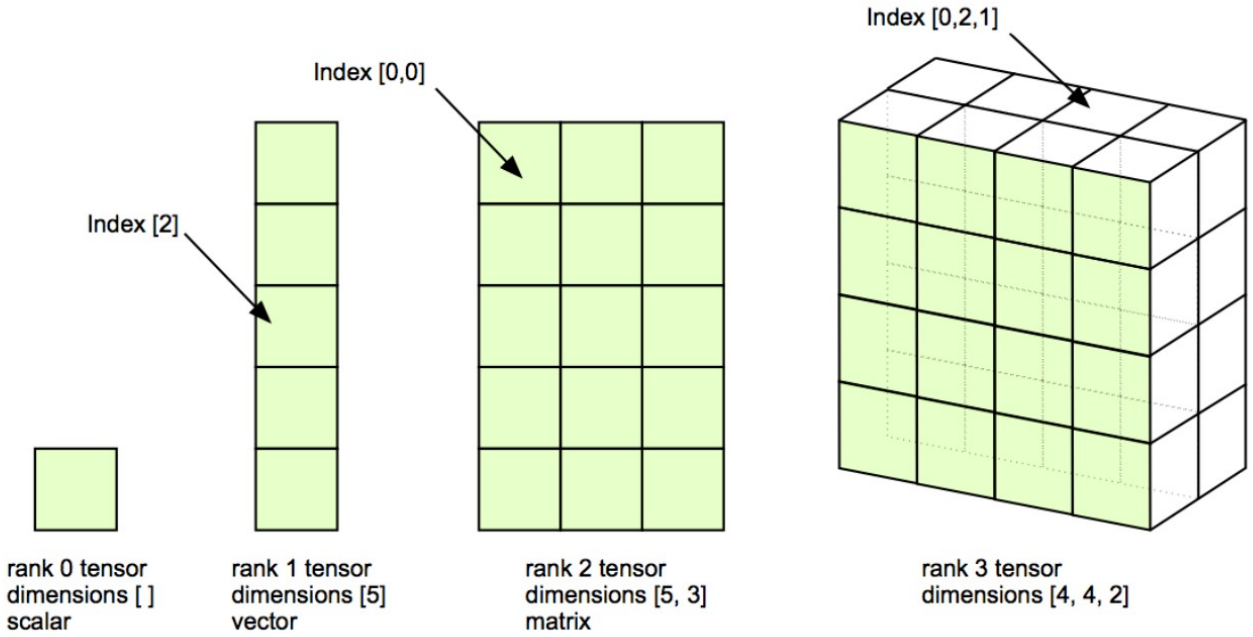
Tensors

Rank 0 Tensor, Scalar, $\mathbb{R}$

Rank 1 Tensor, Vector, $\mathbb{R}^n$

Rank 2 Tensor, Matrix, $\mathbb{R}^n \times \mathbb{R}^m$

Rank 3 Tensor, Tensor, $\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p$



Tensors

Image source: https://leanpub.com/ann-and-deeplearning

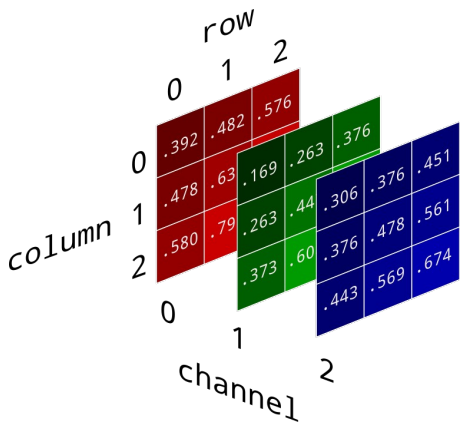
Tensors

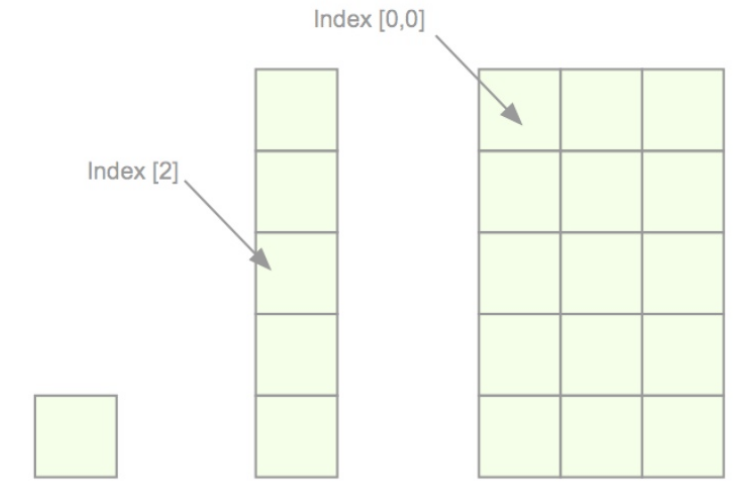
Rank 0 Tensor, Scalar, $\mathbb{R}$

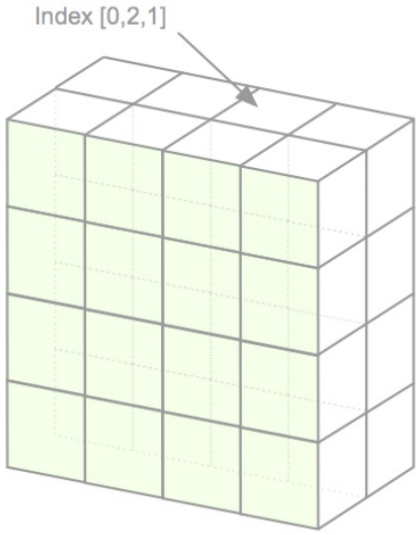
Rank 1 Tensor, Vector, $\mathbb{R}^n$

Rank 2 Tensor, Matrix, $\mathbb{R}^n \times \mathbb{R}^m$

Rank 3 Tensor, Tensor, $\mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p$







Tensors

PyTorch **EXAMPLE**

Typical notation: (N, C, H, W)
N=batch, C=channels, Height, Width)

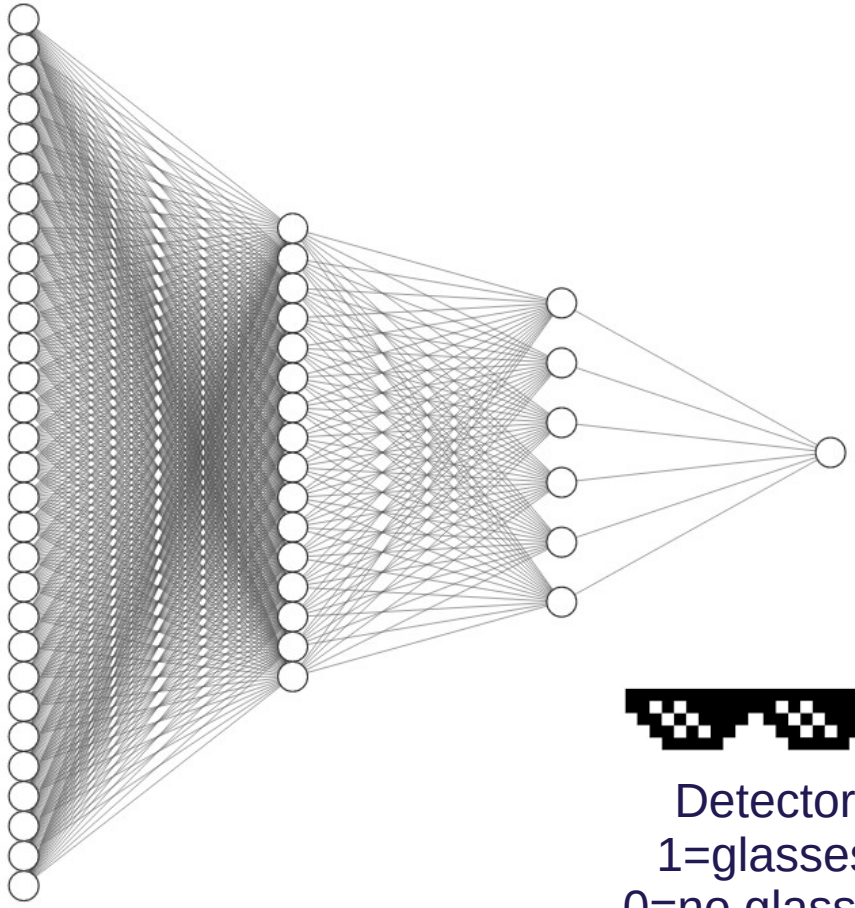
```
>>> import torch
>>> t = torch.ones(1,3,8,8)
>>> t
tensor([[[[1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.]],

         [[1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.]],

         [[1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1., 1., 1., 1.]]]])
>>> t.shape
torch.Size([1, 3, 8, 8])
```

Image source: https://leanpub.com/ann-and-deeplearning

Fully Conected NNs and Grid Data

256x256 pixels

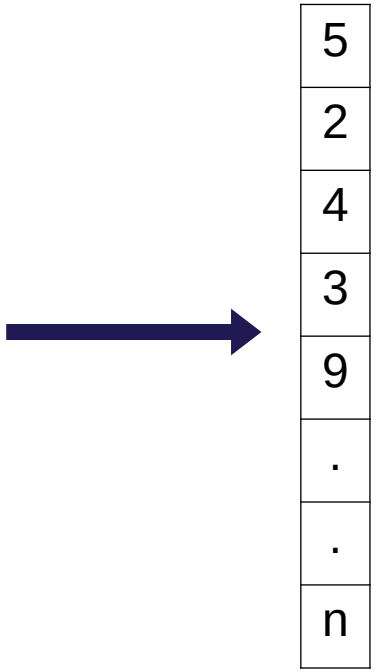
?



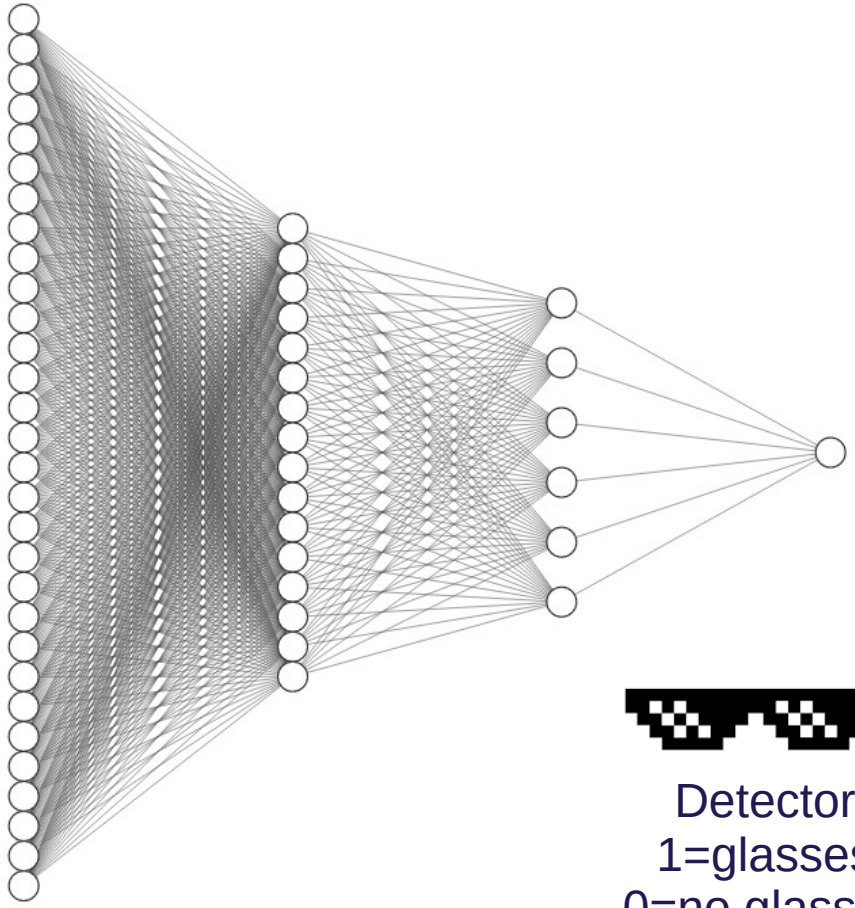
Detector
1=glasses
0=no glasses

Fully Conected NNs and Grid Data

256x256 pixels



We can flatten the image to a vector, and discard the grid-like structure 😞



Detector
1=glasses
0=no glasses

Poor Scaling of Fully Connected Neural Networks

Image example:

Input image:
`256x256` pixels, 1-channel grayscale

Network layout:
`256x256=65536` weights pr. neuron

If input = neurons in hidden layer:
~4.3 billion learnable parameters + bias

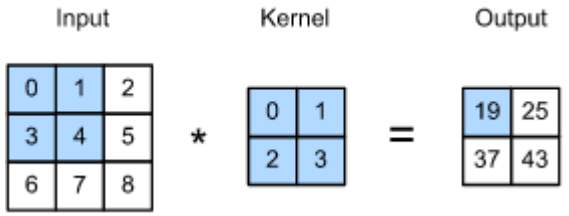
(Recall that AlexNet had around 61M params)

256x256 pixels

Not feasible!

Recall: Spatial Convolution / Cross Correlation

- We slide a kernel (filter) across a 2D grid (image) and compute dot products (multiply and sum the entries) to produce the output.

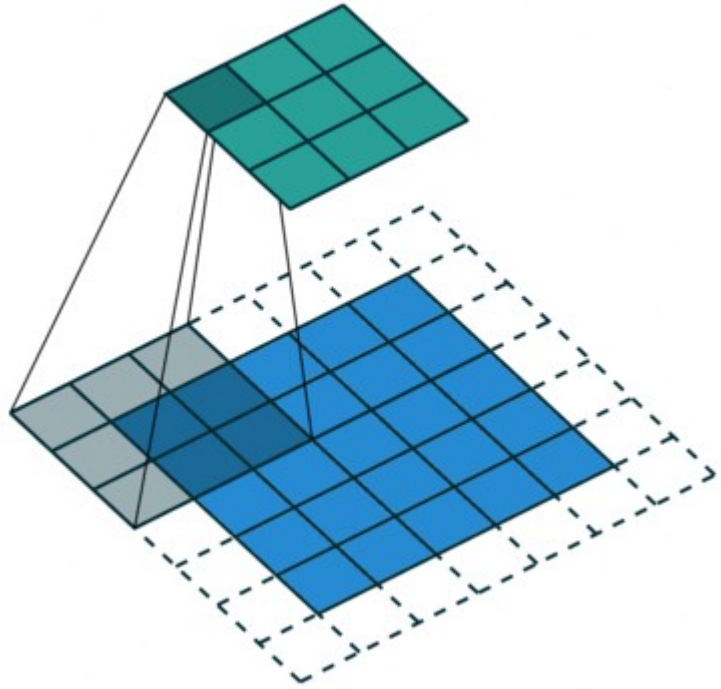


- The kernel can be hand-crafted for edge detection e.g. Sobel:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

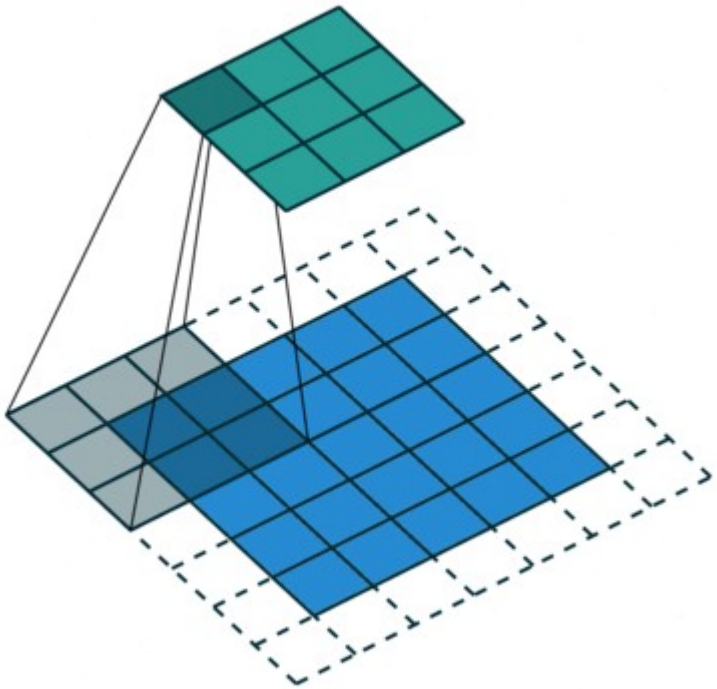
- Or for blurring e.g. Gaussian:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

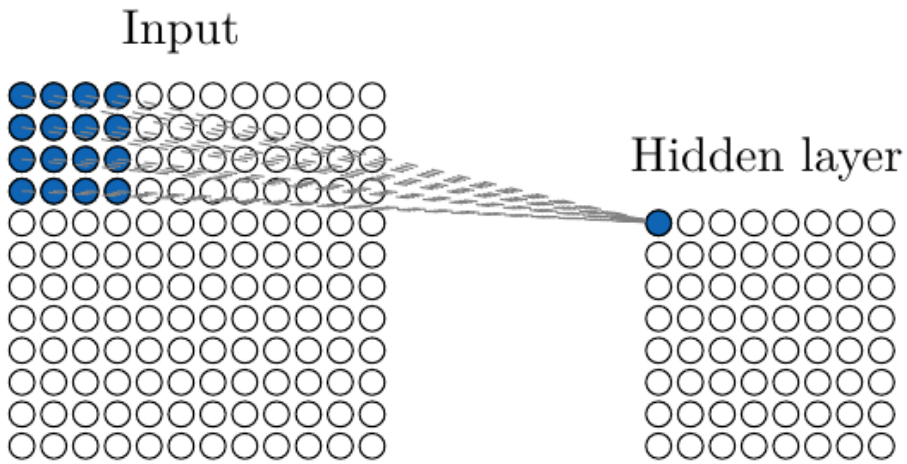


- Convolution uses a flipped filter
- Cross-Correlation use the unmodfied filter
- Both operations can be normalized
- PyTorch and Tensorflow implements cross-correlation without normalization pr. default

Convolution in CNNs



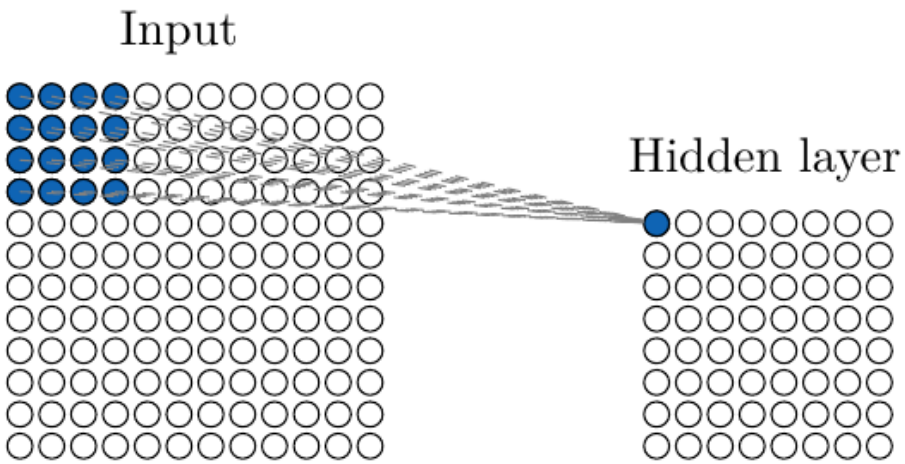
- **In a CNN the weights of the kernel are learned automatically!**
- Based on **local connectivity** opposed to global connectivity in fully connected NNs.
- The size of the kernel determines the local **receptive field** i.e. how large a neighborhood is used to compute the output.

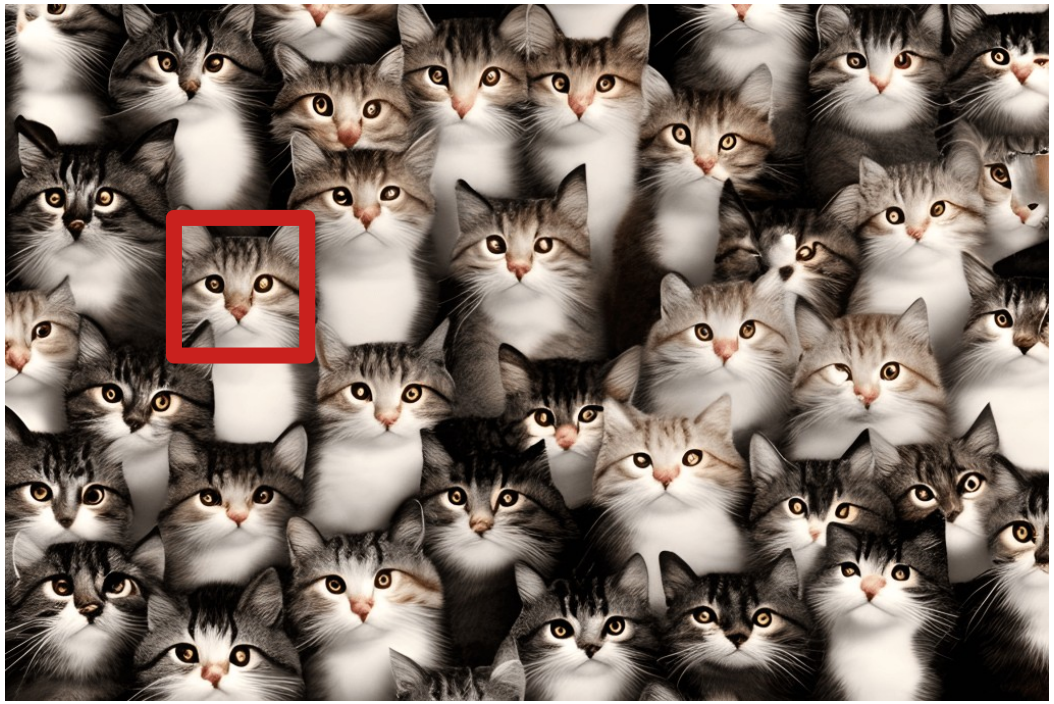


Weight Sharing in CNNs

- The same kernel weights are used across the input, also known as **weight sharing,** which **reduce** the total number of model **parameters**.

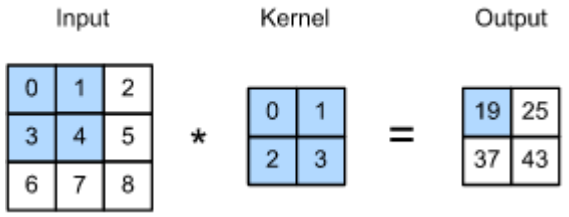
- Since each **filter** will be able to **detect** the **same feature** across the input, the model becomes **translation equivariant** (the output change in response to the input).





Example: A cat detector kernel (filter) works equally well across the image and is therefore reusable

The Output of a CNN Layer



- The output of a convolution in a CNN is the the filter response which is called:
  - A **feature map**

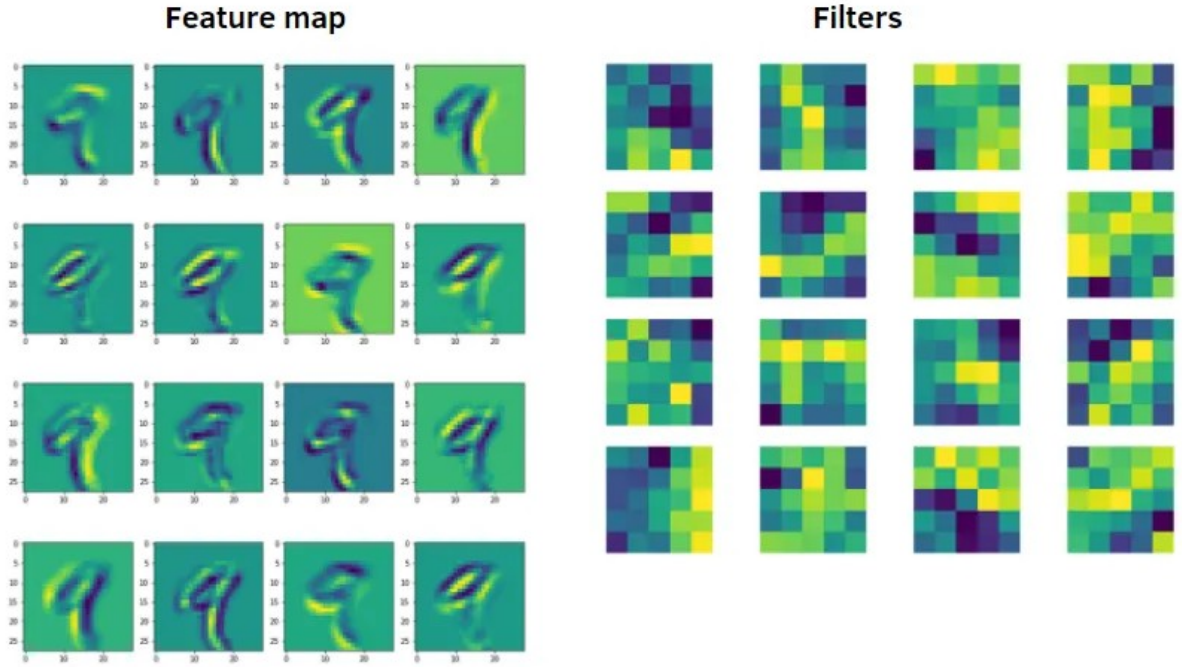
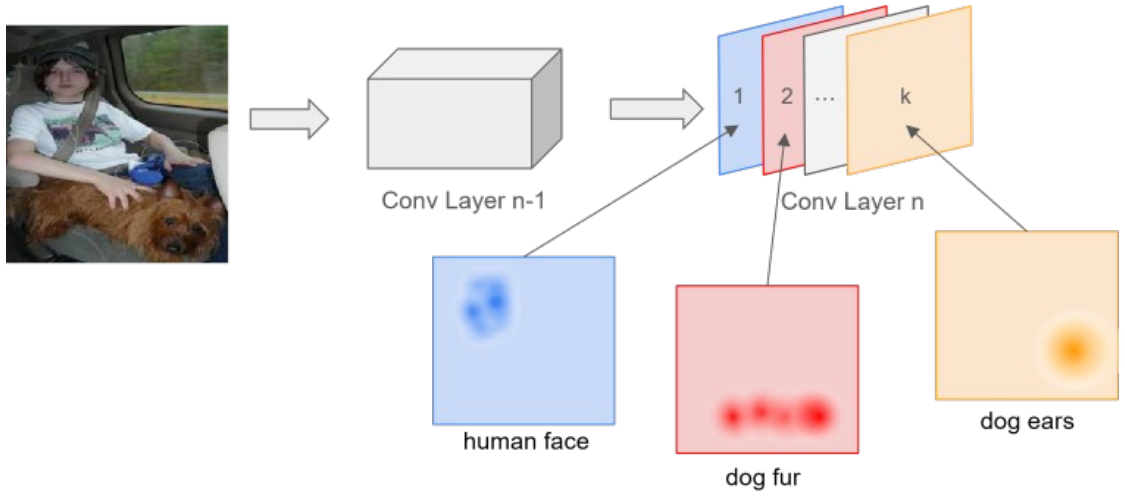


Image source: https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e

Activation Functions in CNNs

- As with fully connected neural networks **CNNs** also use **activation functions** to introduce **non-linearity** and help the network **focus** more on **strong features**.
- The activation function is **applied for each element** in the feature map
- We can use the **same act. functions** as for FCNNs, e.g. ReLU(x) = max(0,x)
- The resulting feature map **after the act. function** is called an **activation map**



Example: The human face detector filter might also react to the dogs face, but the weaker response can be **thresholded** by the act. function.

Image source: https://johfischer.com/2022/01/27/class-activation-maps/

Multiple Input/Output Channels

Multiple Inputs:

- If the input tensor have **multiple channels** the filter, or convolutional kernel, needs to have a **similar depth**.

- The **output** of a **single filter**, regardless of its depth, is **always** a **single feature map**.

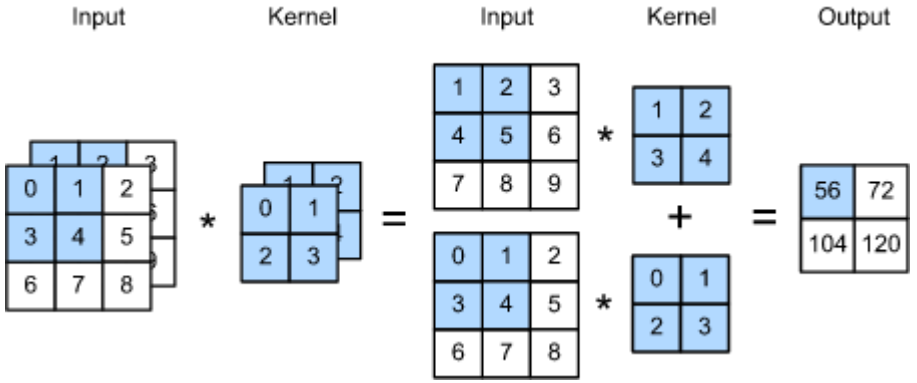
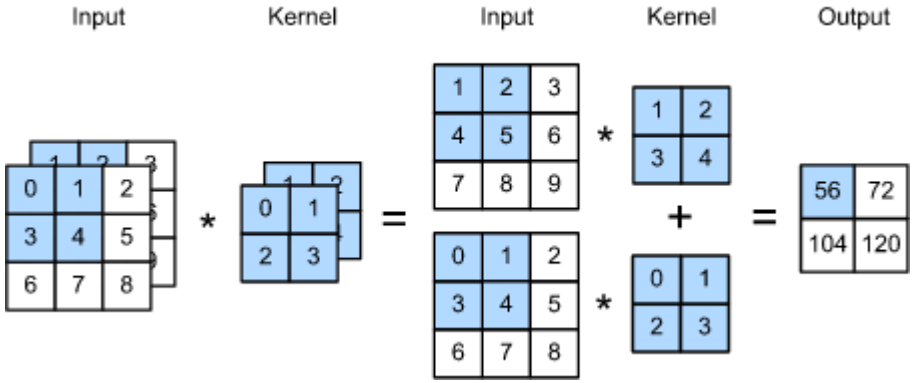


Image source: https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e

Multiple Input/Output Channels

Multiple Inputs:

- If the input tensor have **multiple channels** the filter, or convolutional kernel, needs to have a **similar depth**.



- The **output** of a **single filter**, regardless of its depth, is **always** a **single feature map**.

Multiple Outputs:

- We can extract **multiple feature maps** from a single input channel by using **multiple filters**.

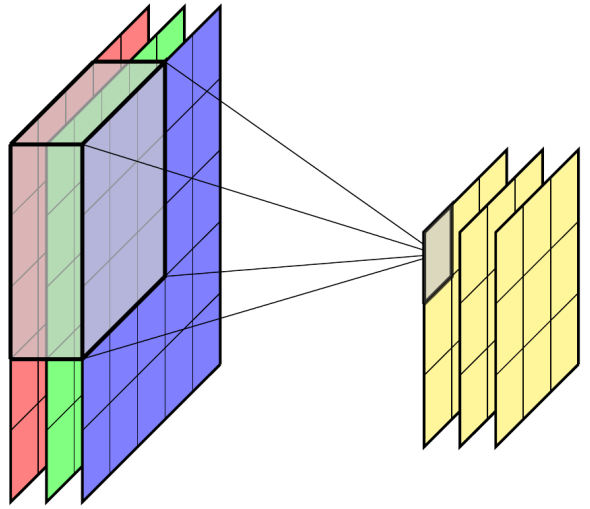
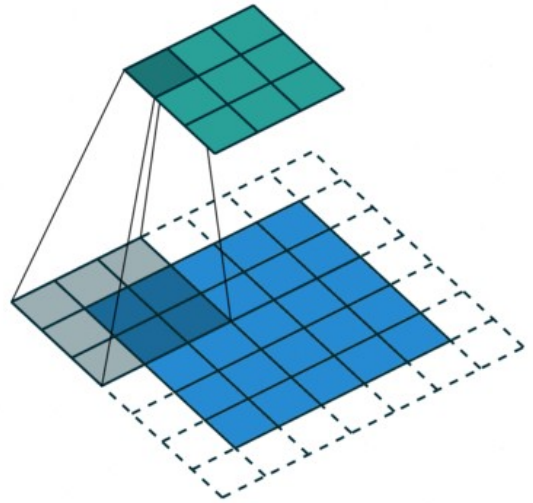


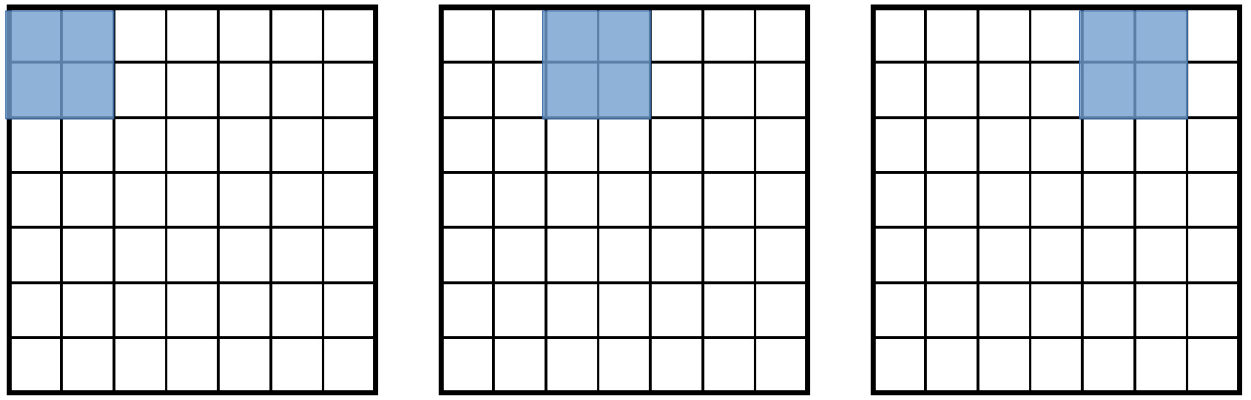
Image source: https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e

Padding and Stride

Recall that:

- **Padding** is used to overcome the border problem when convolving.
- **Stride** is a number that controls how much the kernel/filter is moved for each step.
- Suitable numbers must be used to ensure outputs with integer dimensions.





Stride 2 example

Convolutional Layer

- In **convnets** the **width** and **height** of the input can **change without modifying the architecture**

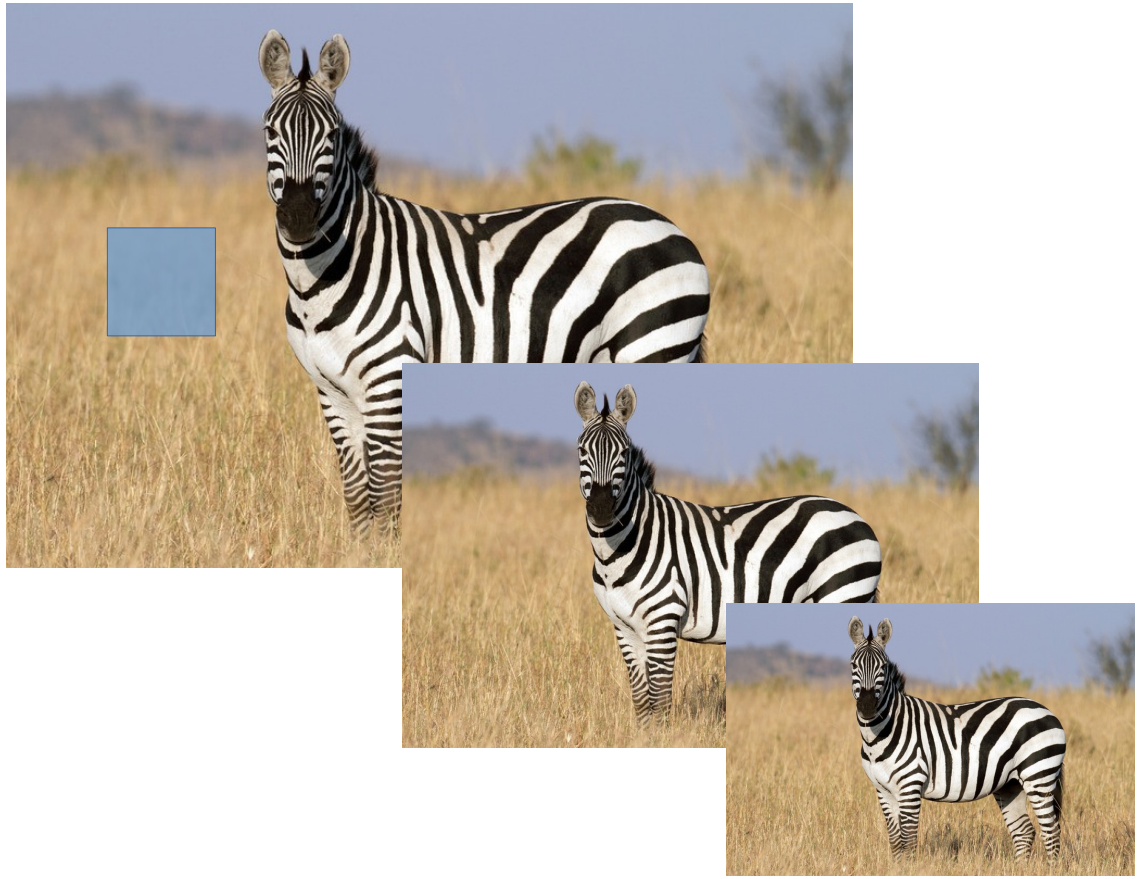
- The output shape of a convolutional operation on X is computed as:

$$X_{out} = \frac{X_{in} - K + 2P}{s} + 1$$

Where K is the kernel size, P is the padding and S is the stride

- To obtain outputs with same shape as inputs set padding to:

$$P = \frac{(K-1)}{2}$$



Convolutional Layer

- In **convnets** the **width** and **height** of the input can **change without modifying the architecture**

- The output shape of a convolutional operation on X is computed as:

$$X_{out} = \frac{X_{in} - K + 2P}{s} + 1$$

Where K is the kernel size, P is the padding and S is the stride

- To obtain outputs with same shape as inputs set padding to:

$$P = \frac{(K-1)}{2}$$

PyTorch **EXAMPLE**

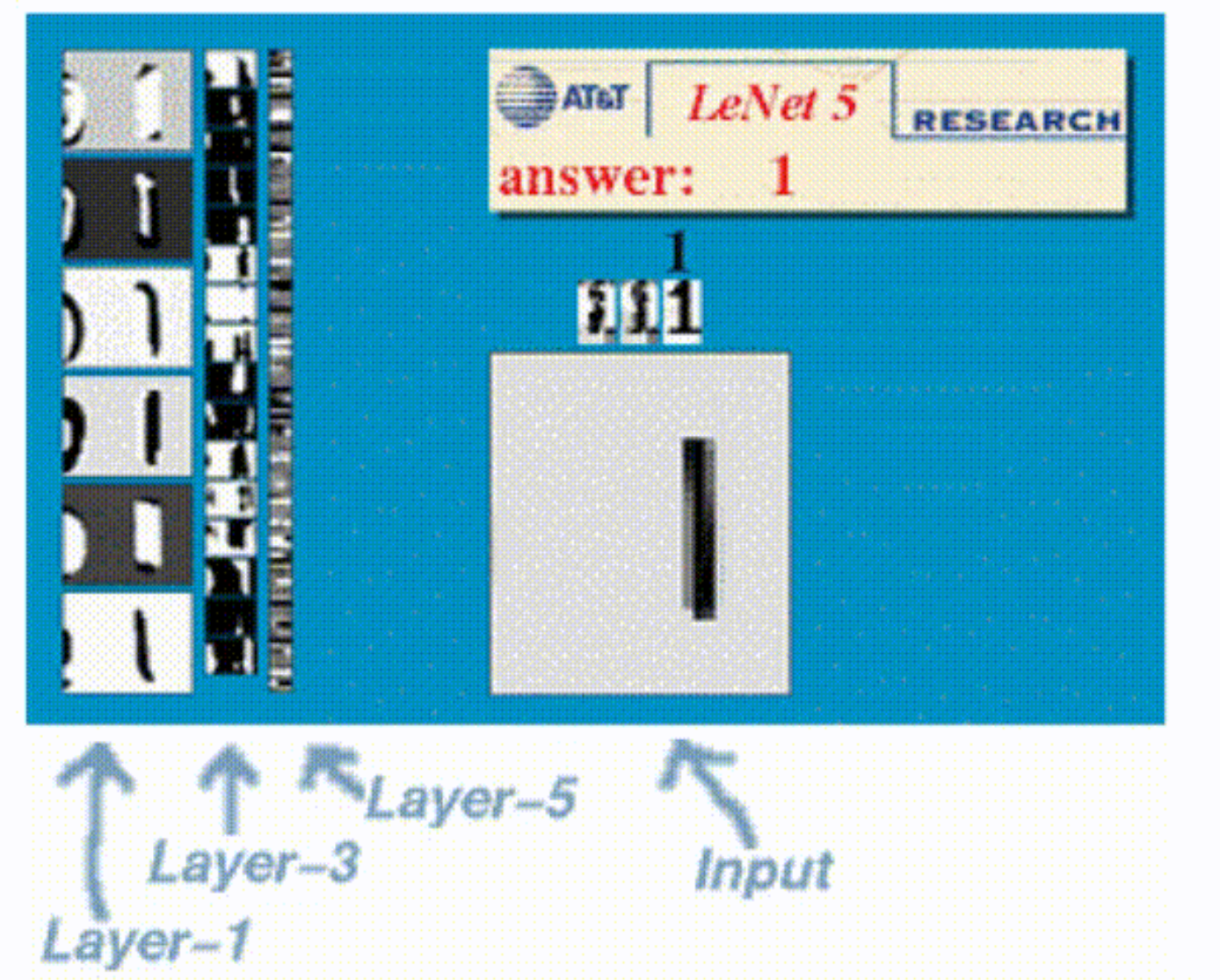
Conv2d

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

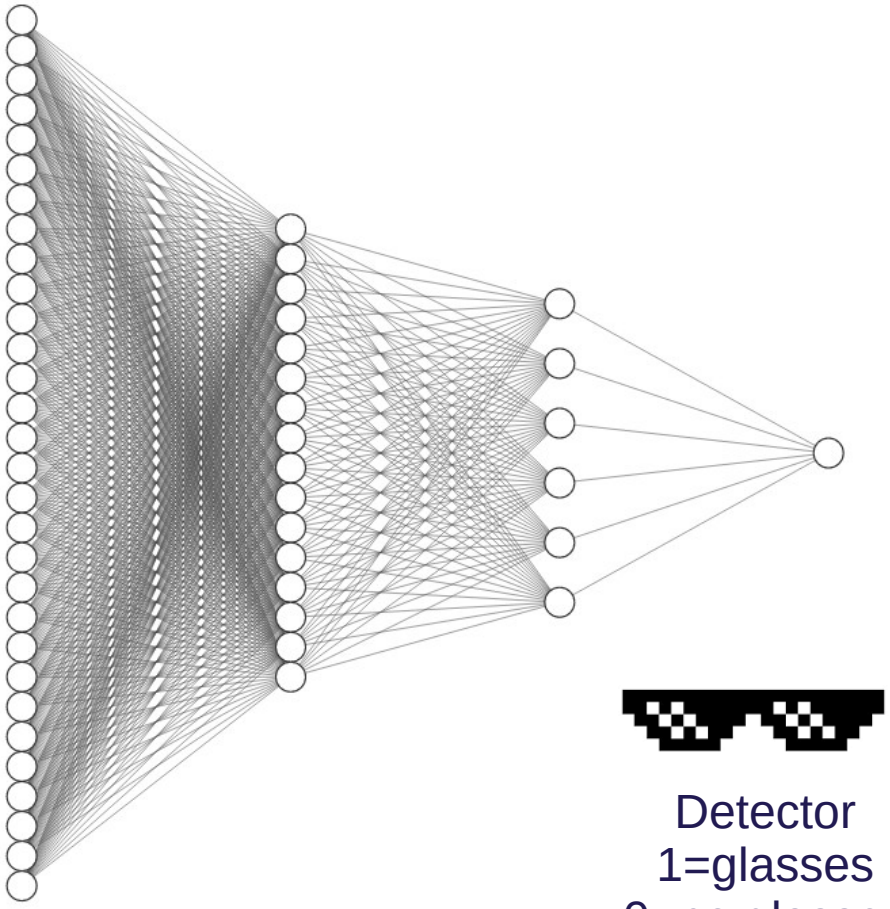
```
>>> input = torch.randn(1,16,32,32)
>>> conv = torch.nn.Conv2d(16,32,5,1,2)
>>> output = conv(input)
>>> output.shape
torch.Size([1, 32, 32, 32])
```

Recall the notation: (N, C, H, W)
N=batch, C=channels, Height, Width)

Break – 10 minutes



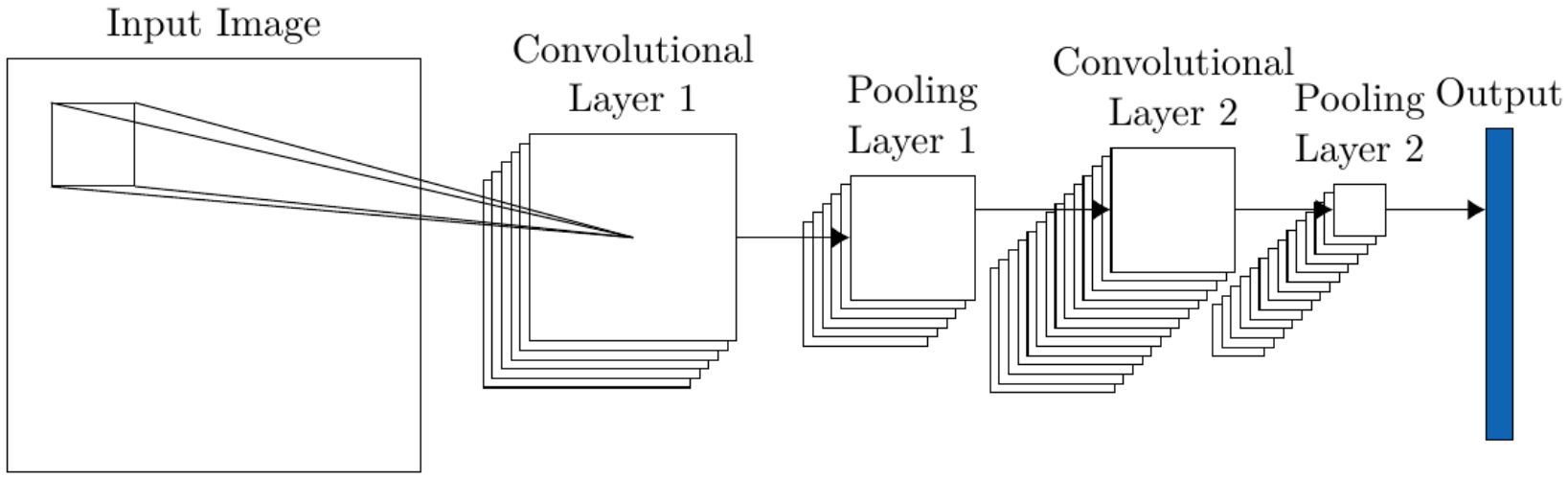
Pooling Layers



Detector
1=glasses
0=no glasses

We typically need to map to an output < input

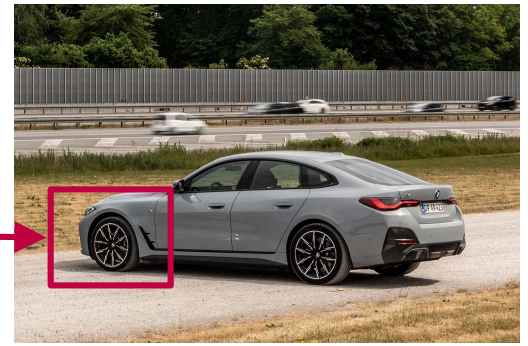
Pooling Layers



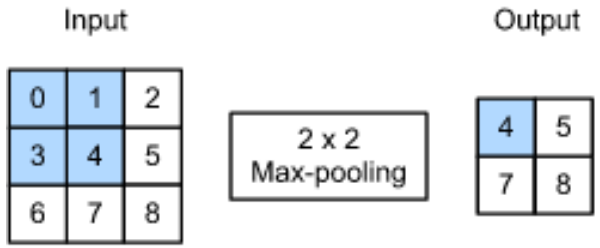
From 3D input to single prediction

The pooling layer:

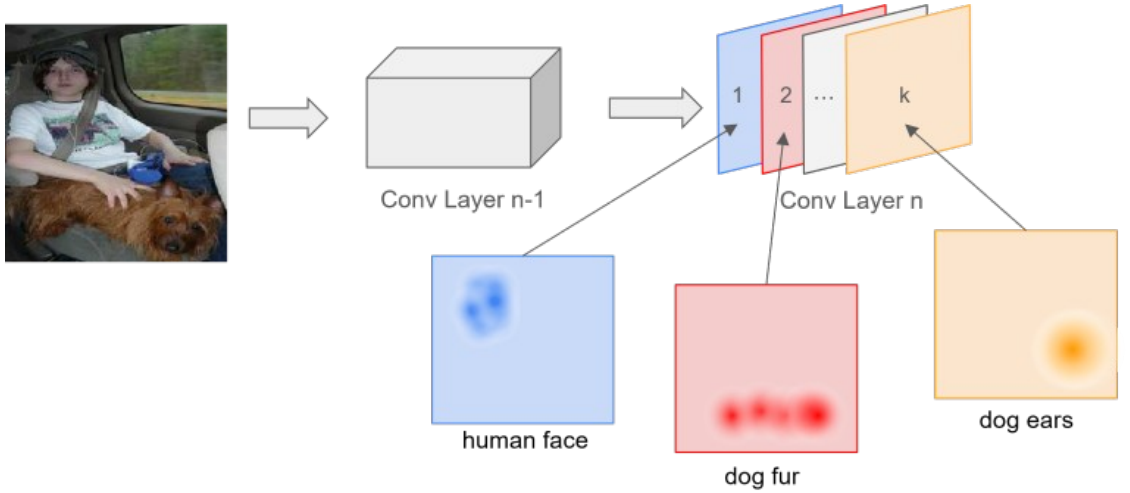
- **Aggregates** information in the **activation map**
- Performs **spatial downsampling**
- Allows for **greater channel depth** and **larger effective receptive fields**
- Adds **invariance** to small shifts within the pooling window I.e **the exact location of a feature is not critical**.



Pooling Layers

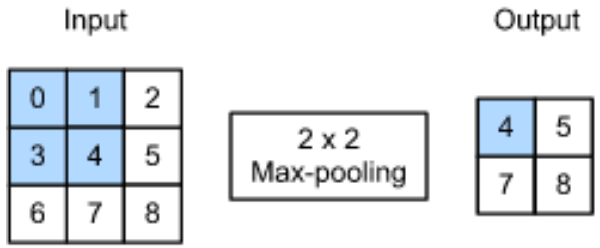


- Common **pooling** types:
  - **Max** pooling
  - **Avg**. pooling
- The pooling **window** is typically of size **2x2**
- Applied in **sliding-window** fashion (like conv)
- There are **no learnable parameters**



A **neuron** in the next layer **doesn't need** the **precise location** of **features** within the pooling window. Whether a feature (like an edge) is slightly shifted to the left or right, max pooling will capture the strongest activation in that region

Pooling Layers



- Common **pooling** types:
  - **Max** pooling
  - **Avg**. pooling
- The pooling **window** is typically of size **`2x2`**
- Applied in **sliding-window** fashion (like conv)
- There are **no learnable parameters**

PyTorch **EXAMPLE**

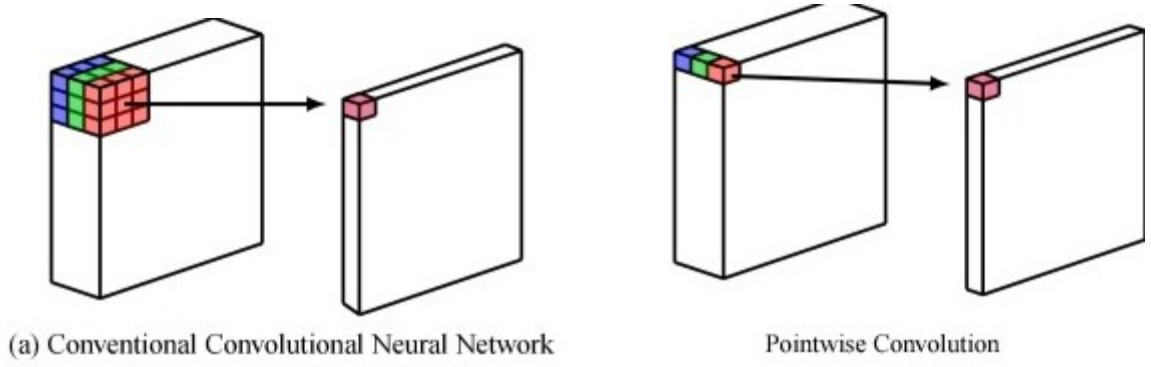
MaxPool2d

CLASS torch.nn.MaxPool2d(*kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False*) [SOURCE]

```
>>> fea = torch.randn(1,16,32,32)
>>> pool = torch.nn.MaxPool2d(3, stride=2, padding=0)
>>> act = pool(fea)
>>> act.shape
torch.Size([1, 16, 15, 15])
```

Pointwise Pooling / Channel Mixing

- **1x1 Convolutions**, or pointwise convolutions, can be used to **shrink the number of channels** I.e channel mixing
- Using a filter size of 1x1, this operation learns **a weighted combination of the channels**
- The number of filters, or output size, defines the reduction level



(a) Conventional Convolutional Neural Network

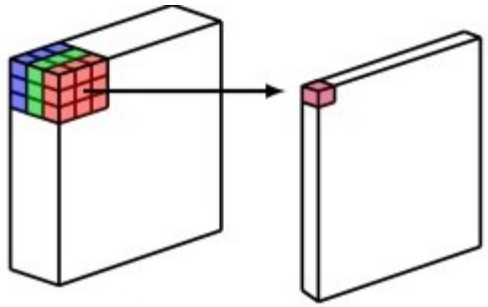
Pointwise Convolution

Pointwise Pooling / Channel Mixing

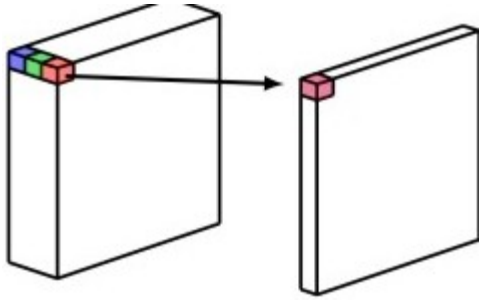
- **1x1 Convolutions**, or pointwise convolutions, can be used to **shrink the number of channels** I.e channel mixing
- Using a filter size of 1x1, this operation learns **a weighted combination of the channels**
- The number of filters, or output size, defines the reduction level

PyTorch **EXAMPLE**

```
>>> x = torch.ones(64,32,32)
>>> conv1x1 = nn.Conv2d(64, 32, kernel_size=1, stride=1, padding=0)
>>> y = conv1x1(x)
>>> y.shape
torch.Size([32, 32, 32])
```



(a) Conventional Convolutional Neural Network



Pointwise Convolution

Transposed Convolutions

Think of **inverting** the **convolution operation**

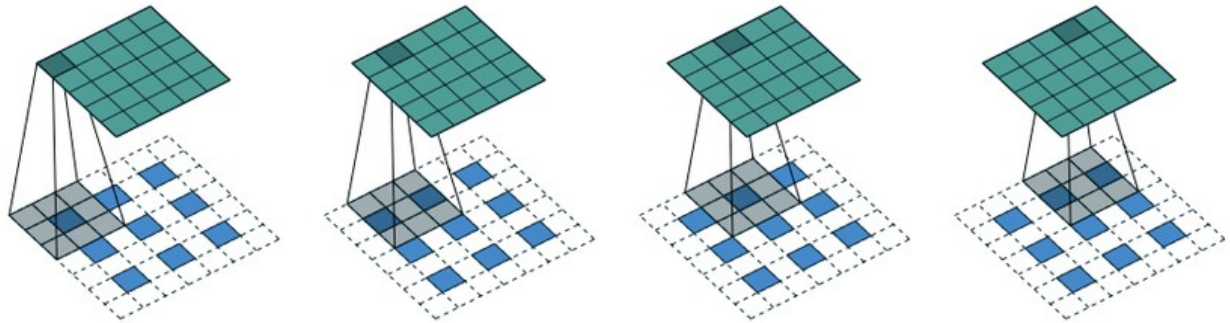
- **Transposed convolution**, or **deconvolution**, is a **learned upsampling** operation.
- Used to increase the **spatial resolution** of feature **maps** while also **learning** filter **weights**.
- Widely used in **decoder-style networks** (autoencoders, U-Net).

PyTorch **EXAMPLE**

```
>>> features = torch.randn(3, 256, 256)
>>> down = torch.nn.Conv2d(3, 16, 3, stride=2, padding=1)
>>> up = torch.nn.ConvTranspose2d(16, 3, 4, stride=2, padding=1)
>>> downsampled = down(features)
>>> downsampled.shape
torch.Size([16, 128, 128])
>>> upsampled = up(downsampled)
>>> upsampled.shape
torch.Size([3, 256, 256])
```

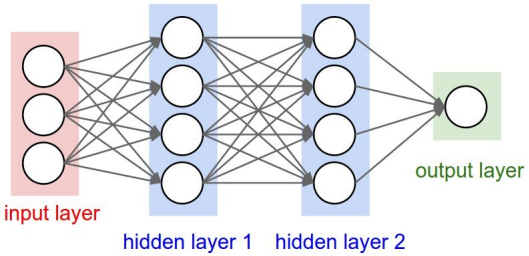
For a 2D input of shape (H,W) the output H and W is computed as:

$$H_{out} = (H-1) \times stride - 2 \times padding + kernel\_size + output\_padding$$



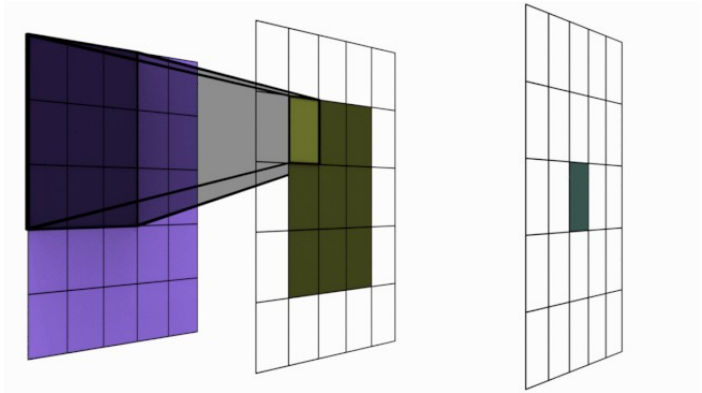
https://arxiv.org/pdf/1603.07285

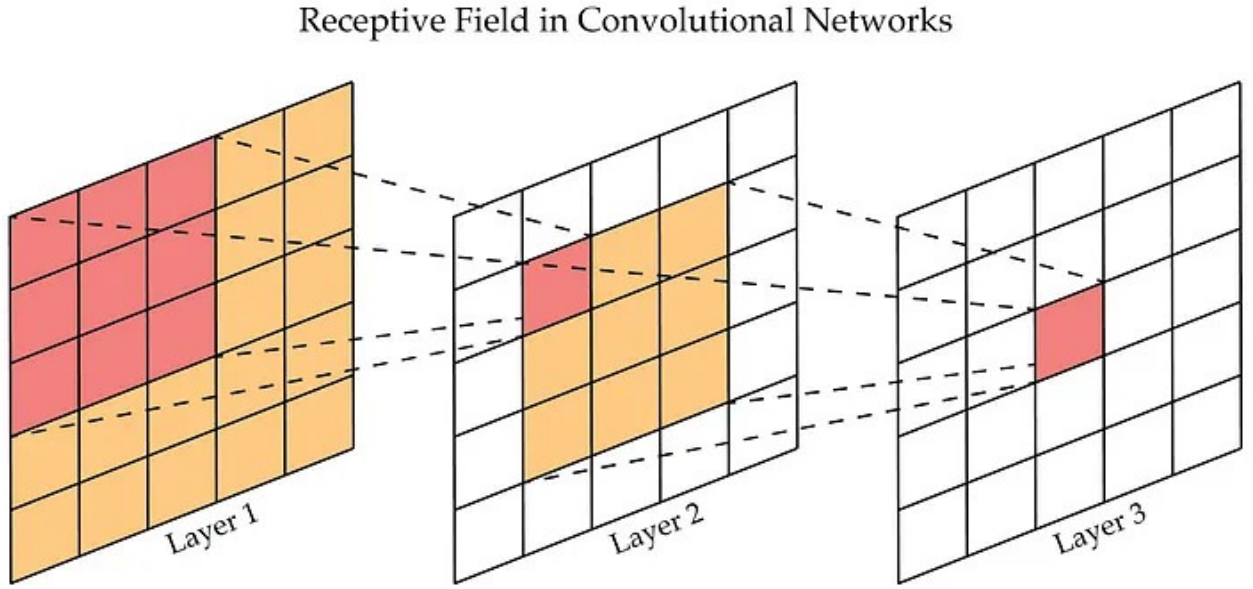
Receptive Fields



Local receptive fields: how much does a particular neuron "see"

- For the **first layer**, the local receptive field is **equal to the filter size**, e.g. the image area a neuron "sees".
- Each conv layer adds K-1 to the receptive field size.
- Besides increasing the **filter size**, **Pooling** or **strided** convolution can be used to increase the receptive field earlier.



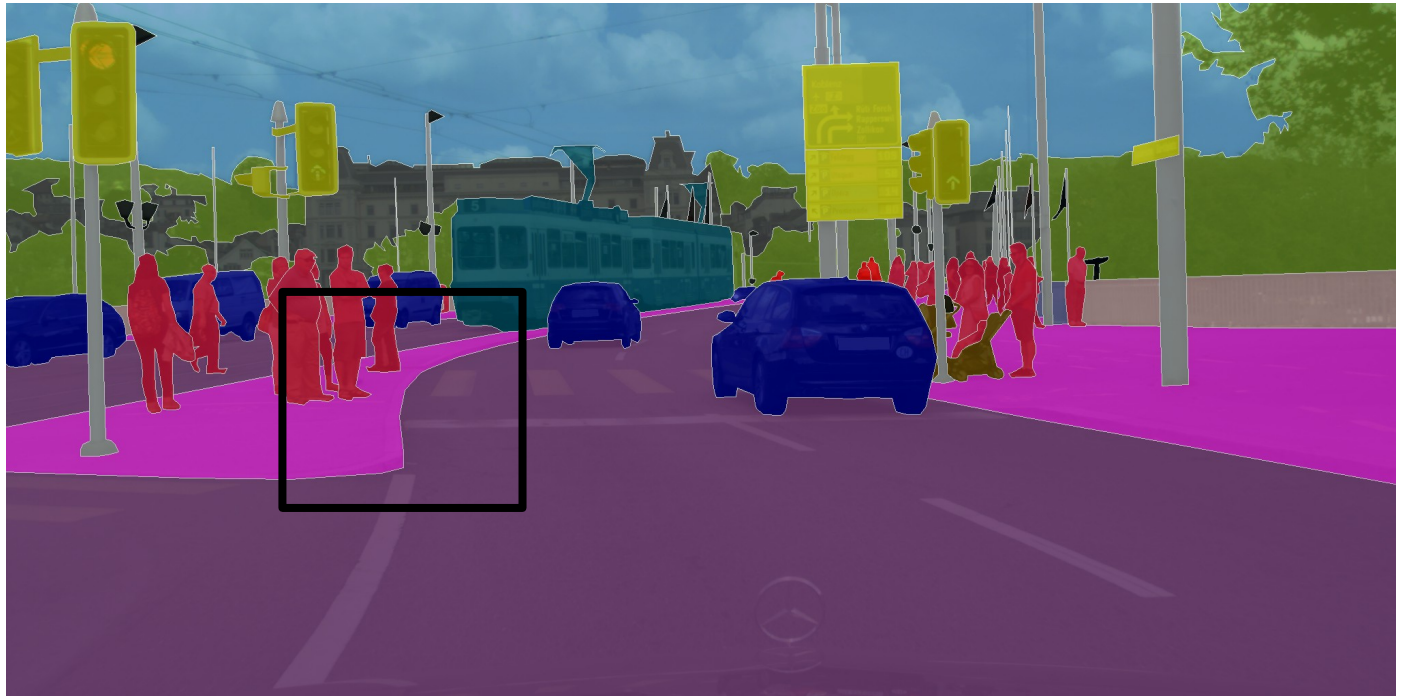


Receptive field example: In layer 1, assuming stride=1, the receptive field covers **3x3** pixels of the input, while layer 2 covers **5x5** pixels (3+3-1) of the input, and so on.

Receptive Fields

Final receptive field:

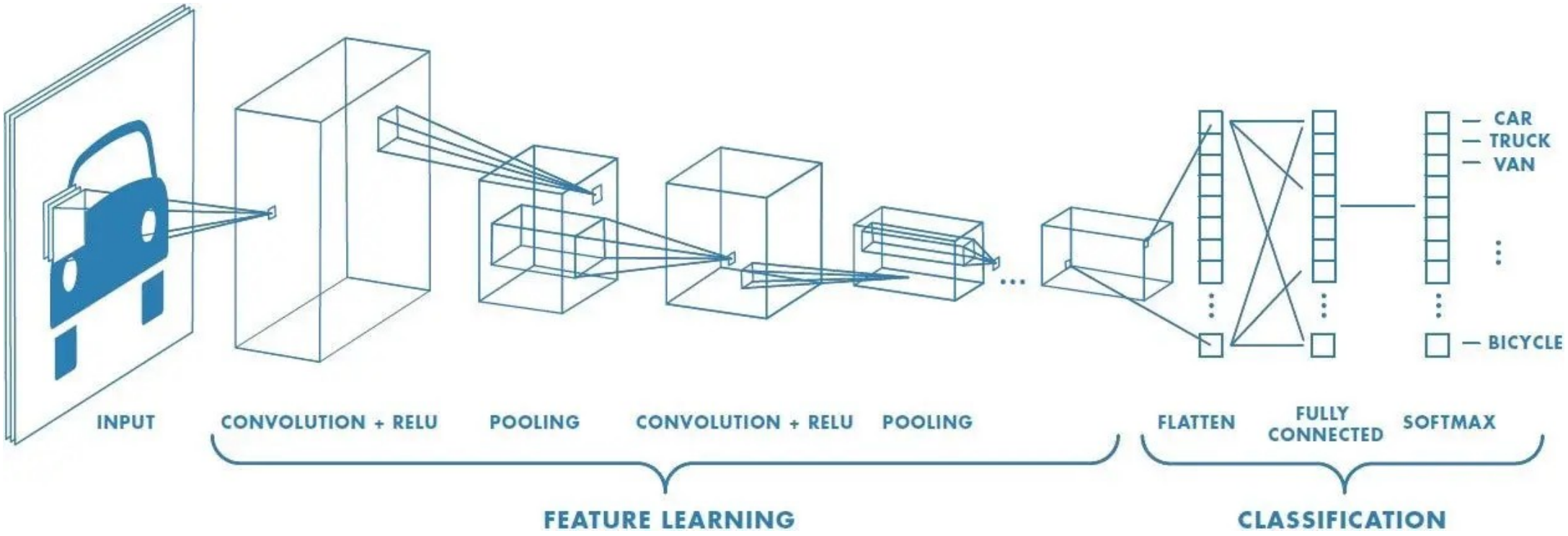
- A result of the architecture, which defines **how large an area** is used to **compute predictions** for a **single unit**, e.g. pixel in the output.
- The receptive field **grows** with the **depth** of the network, influenced by **kernel** size, **stride** and **padding**
- The layer-wise or **final receptive field size** can be calculated by hand or using a package e.g. **torch_receptive_field**
- The final **receptive field** can end up being **global**



Are global receptive fields necessary?

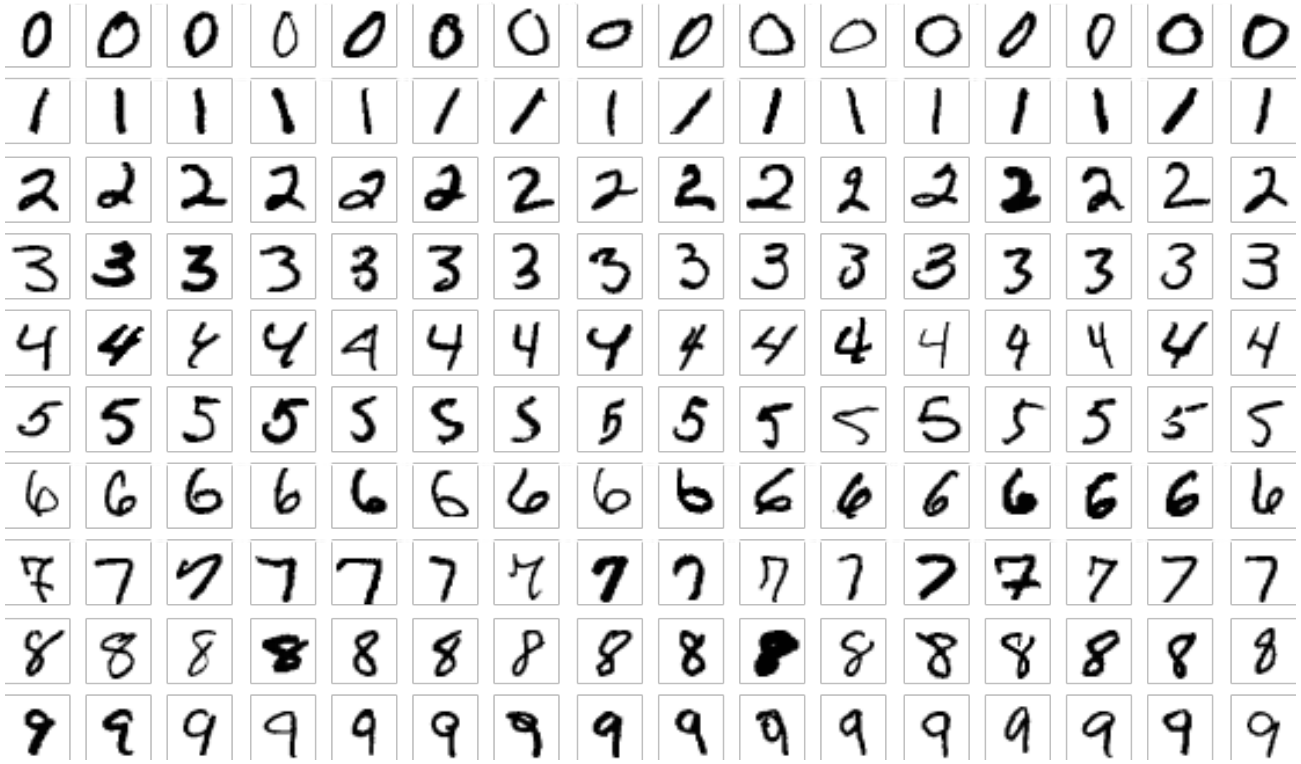
Image source: https://medium.com/@rekalantar/receptive-fields-in-deep-convolutional-networks-43871d2ef2e9

Comparing to Fully Connected Networks



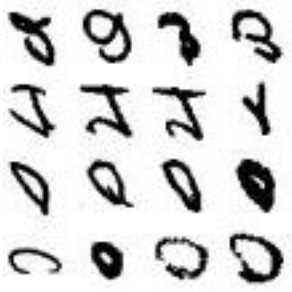
We traditionally use the same **two-stage approach** of initial **feature extraction** and **classification** as in **FCNNs**, but replace the first part with the more **efficient conv** layers.

Comparing to Fully Connected Networks



MNIST Database

- Handwritten digits 0-9
- 60000 training images
- 10000 testing images
- 28x28 pixels



Difficult examples

Deng et al, “Deng, Li. "The mnist database of handwritten digit images for machine learning research”, IEEE Signal Proc. 2012

Comparing to Fully Connected Networks

Fully Connected version

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
dense_7 (Dense)              (None, 256)               200960
dense_8 (Dense)              (None, 128)               32896
dense_9 (Dense)              (None, 64)                8256
dense_10 (Dense)             (None, 10)                650


=================================================================
Total params: 242762 (948.29 KB)
_________________________________________________________________
```

Test accuracy: 0.9786999821662903

Comparing to Fully Connected Networks

Convnet version

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
flatten_1 (Flatten)          (None, 1600)              0
dense_1 (Dense)              (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34826 (136.04 KB)
_________________________________________________________________
```

Test accuracy: 0.9882000088691711

Fully Connected version

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
dense_7 (Dense)              (None, 256)               200960
dense_8 (Dense)              (None, 128)               32896
dense_9 (Dense)              (None, 64)                8256
dense_10 (Dense)             (None, 10)                650
=================================================================
Total params: 242762 (948.29 KB)
_________________________________________________________________
```

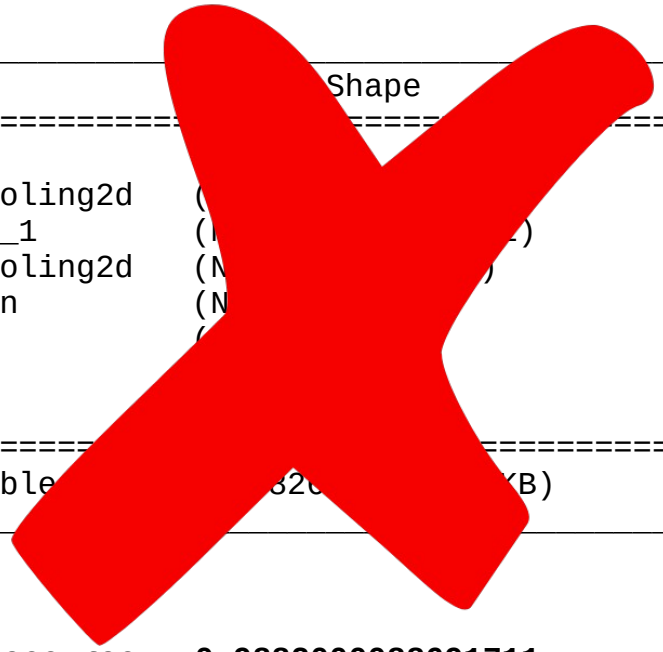
Test accuracy: 0.9786999821662903

More efficient and accurate!

Removing the Fully Connected Layer

We can replace the fully connected layer with global average pooling

Convnet version



Layer	Shape	Param #
conv2d		320
max_pooling2d		0
conv2d_1		18496
max_pooling2d		0
flatten		0
dense		16010

Trainable [illegible] KB)

Test accuracy: 0.9882000088691711

Fully Convolutional version

Layer (type)	Output Shape	Param #
conv2d	(None, 32, 26, 26)	320
max_pooling2d	(None, 32, 13, 13)	0
conv2d	(None, 64, 11, 11)	18496
max_pooling2d	(None, 64, 5, 5)	0
conv2d	(None, 10, 5, 5)	650
global_avg_pooling2d	(None, 10)	0

Total params: 19466 (76.04 KB)

Test accuracy: 0.8432000279426575

Scales to any input size, but less accurate due to fewer params

CNNs - Summary

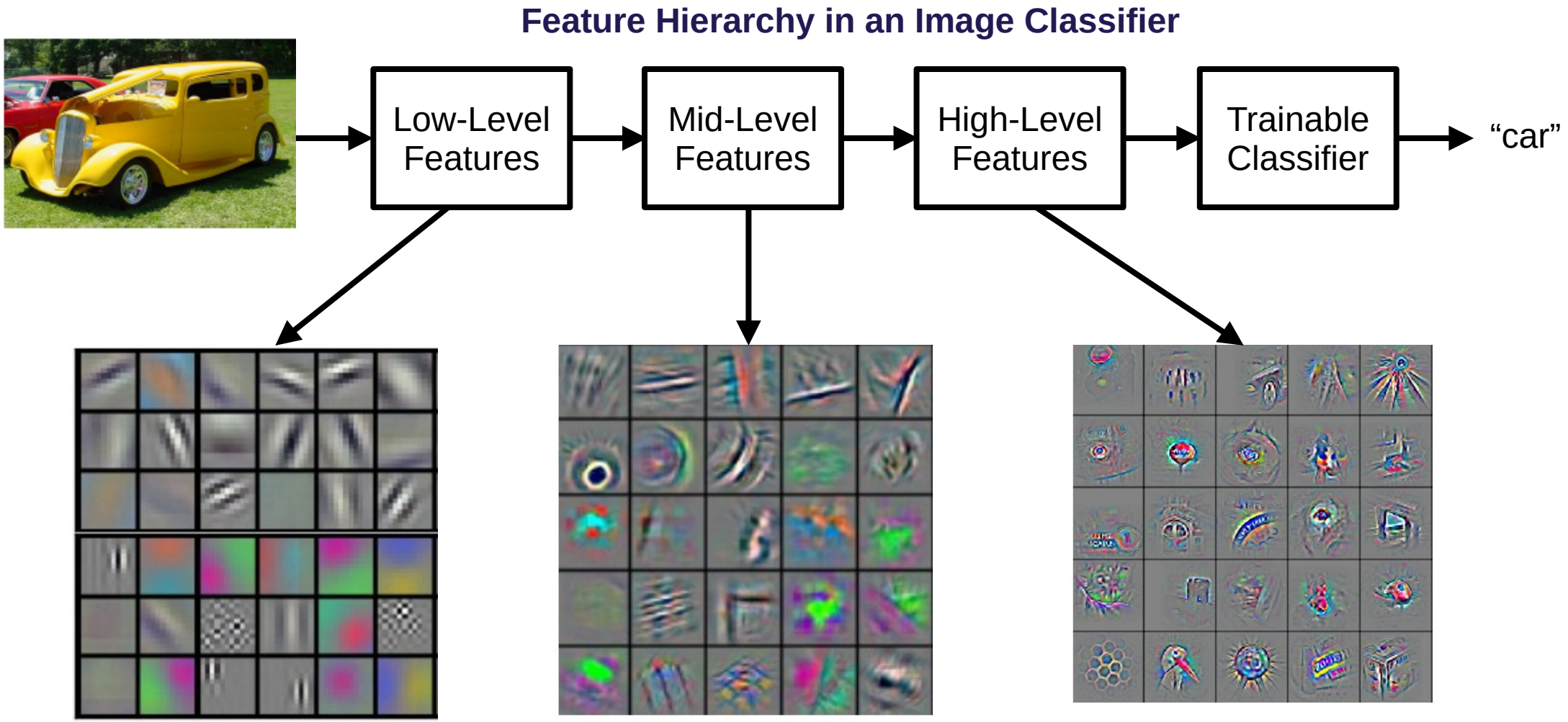
- **Neurons** in a convolutional layer are **not** connected to every single element in the input, but **only** to **their local receptive field**
- **Local connectivity** allows to make assumptions about feature locality e.g. neighbour pixels are likely similar (**inductive bias**)
- The result of a convolution of an input with a filter is a **feature map**, containing the filter response i.e. **high values = good match**

CNNs – Summary cont.

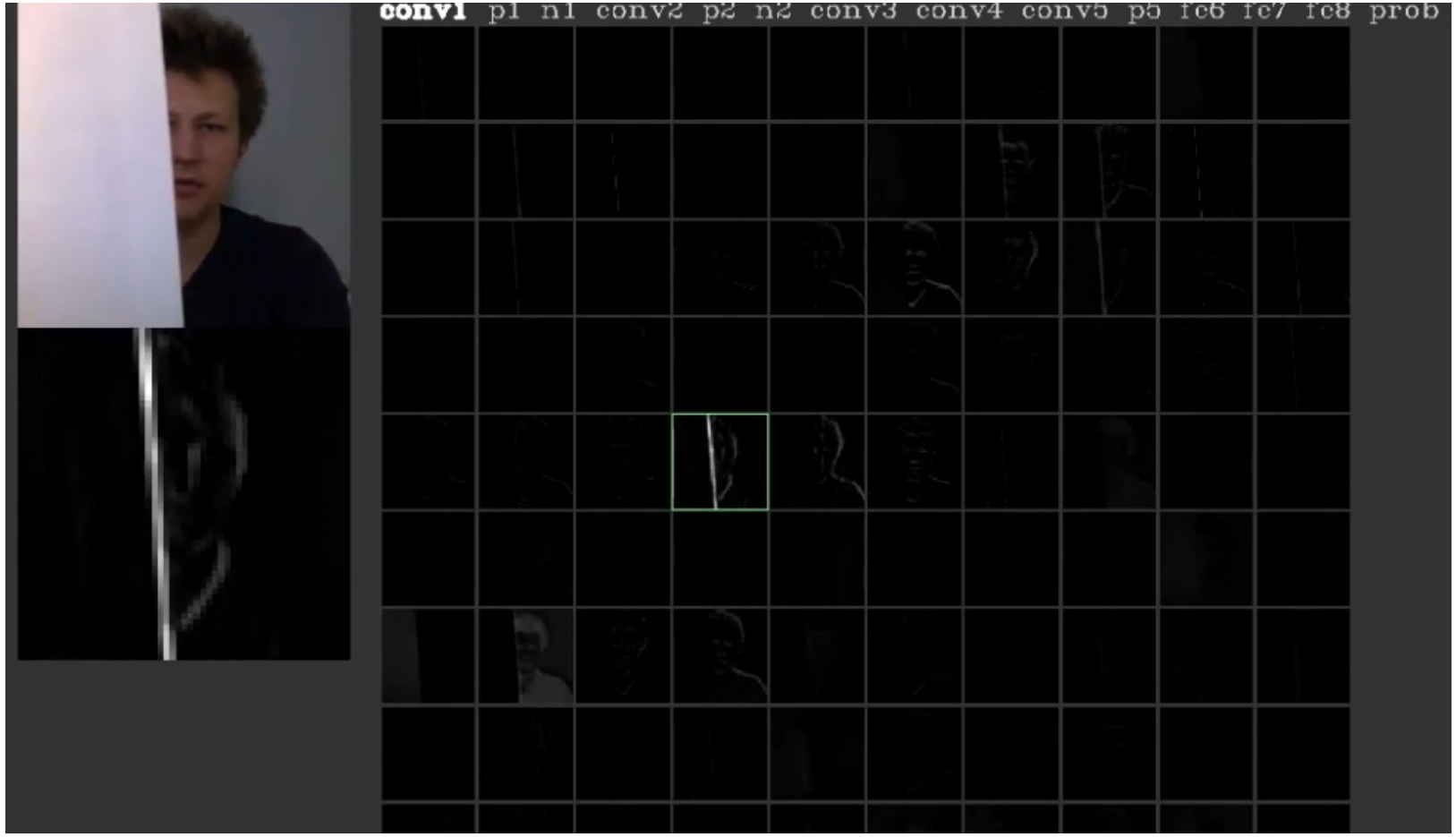
- The same **filter** is used across the image. This **weight sharing** reduce the number of parameters and introduce **translation equivariance**.
- A **hierarchical structure** of repeated conv layers allows to decompose complex patterns into simpler ones (edges → texture → shapes etc.)
- The receptive field is typically increased along the depth using **pooling** layers, which allows to integrate information over larger regions allowing for **global reasoning**.


Micro Break

What Does the CNN Learn?



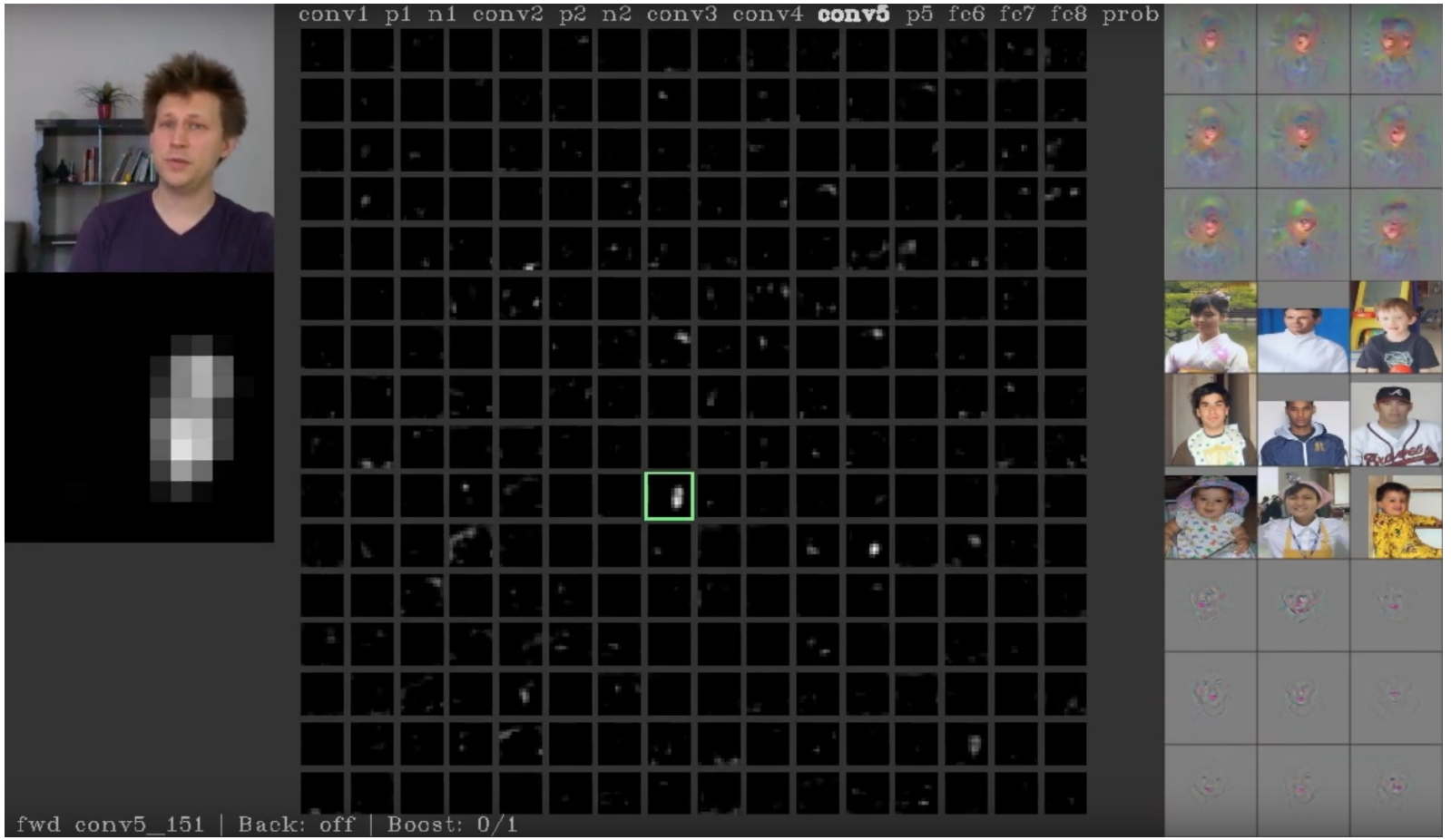
Visualizing Layer Activations



This filter in conv1 responds strongly to light-to-dark transitions

Deep Visualization Toolbox: https://www.youtube.com/watch?v=AgkfIQ4IGaM

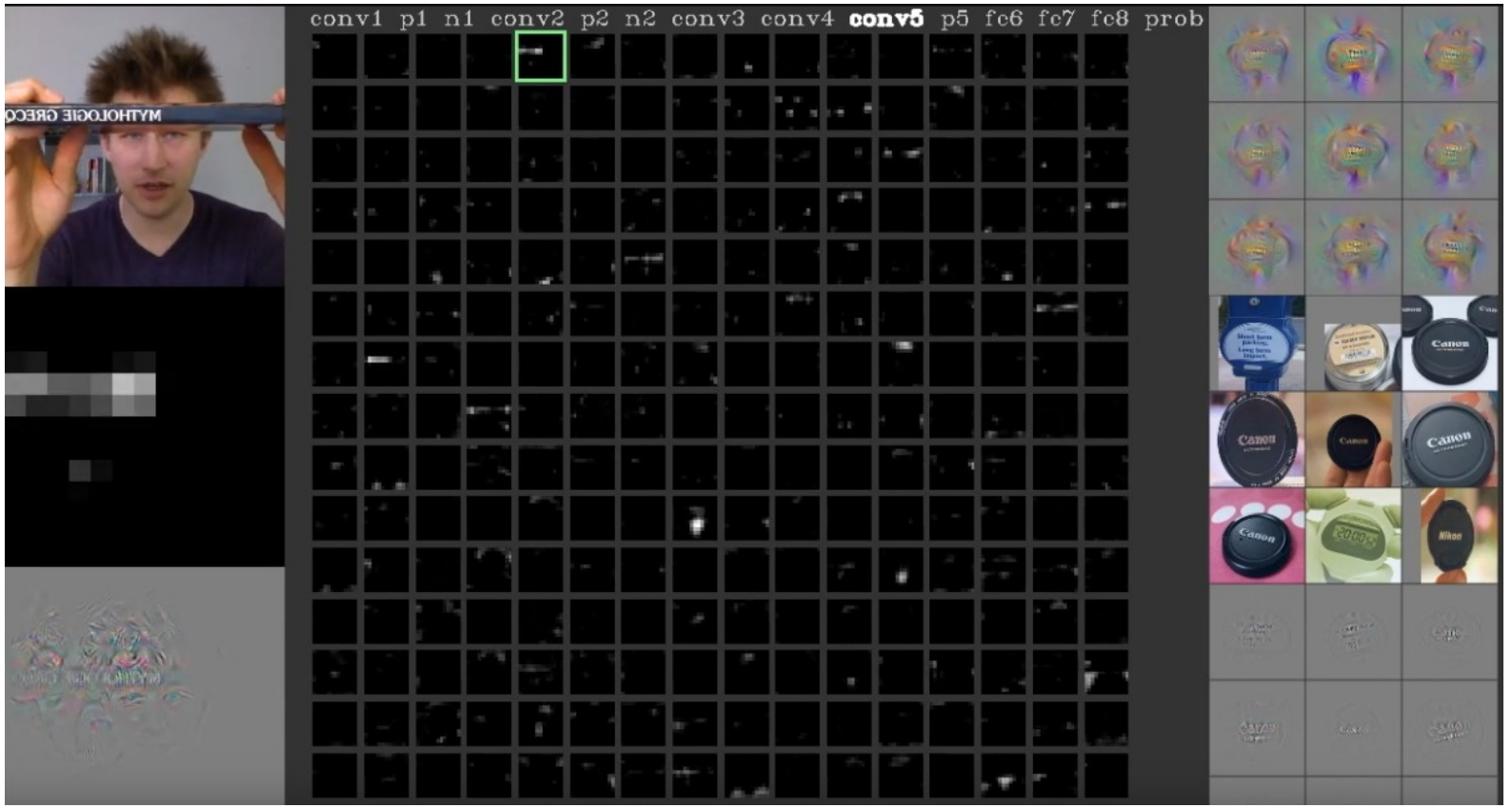
Visualizing Layer Activations



This filter in conv5 responds to faces

Deep Visualization Toolbox: https://www.youtube.com/watch?v=AgkfIQ4IGaM

Visualizing Layer Activations



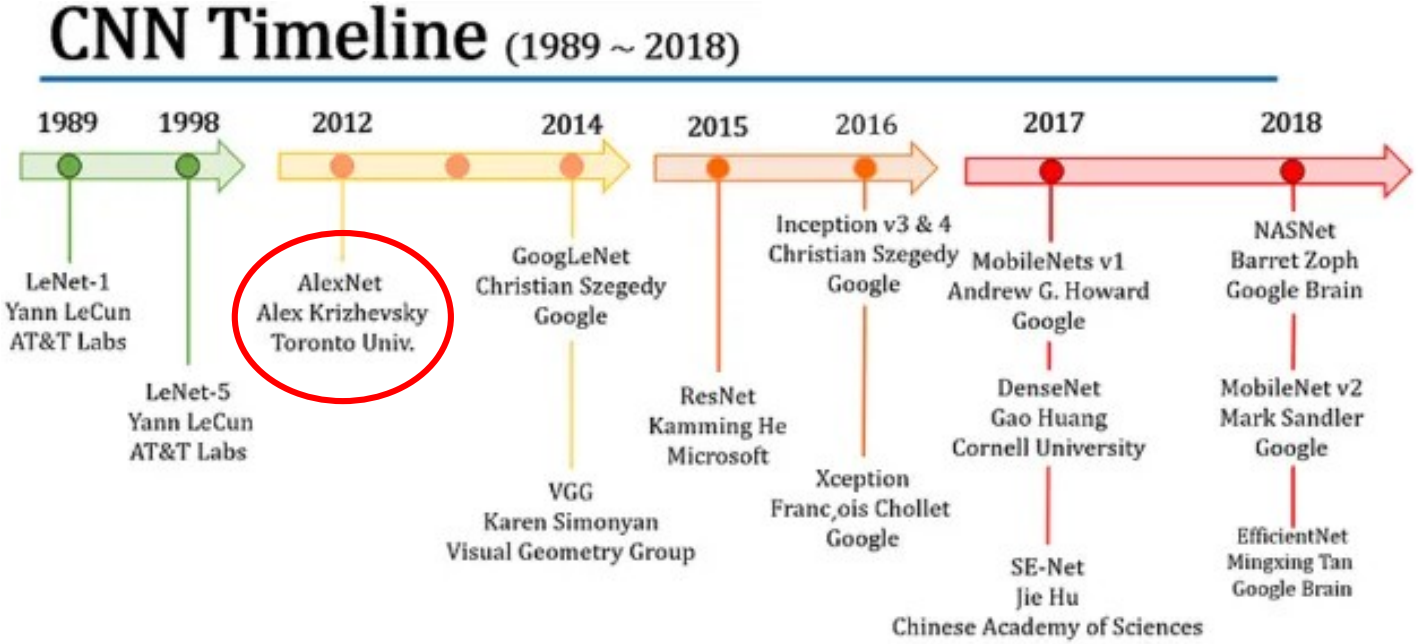
This filter in conv5 responds to printed text.

- Although not beeing a class in ImageNet, detecting **printed text** can **help classify** certain images e.g **bookshelves**.
- The filters in a **CNNs** are **automatically learned** to become **optimal** for a given task.

Deep Visualization Toolbox: https://www.youtube.com/watch?v=AgkfIQ4IGaM

Overview of Well Established CNN Architectures

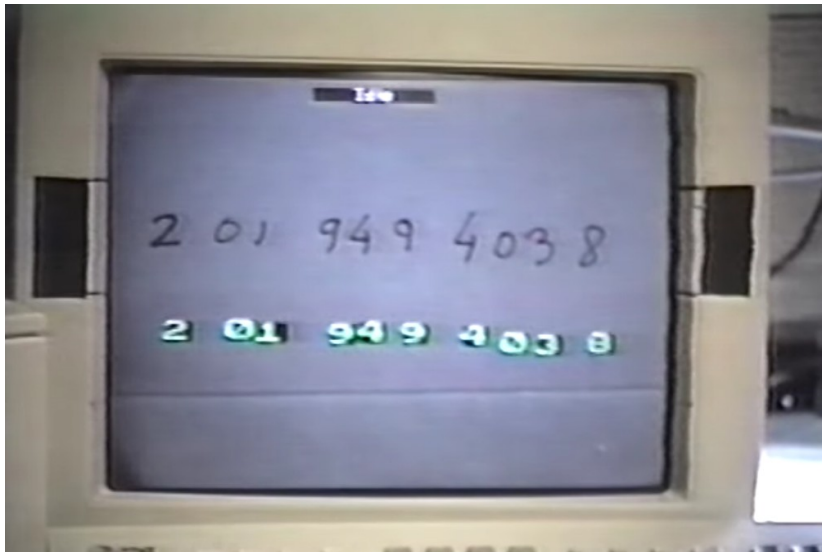
- LeNet-5, 1998
- AlexNet, 2012
- VGGNet, 2014
- GoogLeNet, 2014
- ResNet, 2015



https://blog.gopenai.com/wrapping-up-cnn-models-shifting-focus-to-attention-based-architectures-5029b87034d9

LeNet (1989)

- First “real-time” CNN implementation for hand-written digit recognition.
- Developed in 1989 by Yann LeCun et al. At Bell Labs.
- Executed on a DSP card (12.5 million multiply-accumulate operations per second) in a 486 PC with a video camera and frame grabber card.
- 4 layers: 2 Conv layers with `5x5` filters, **stride 2** and sigmoid activations and 2 fully-connected layers on top.



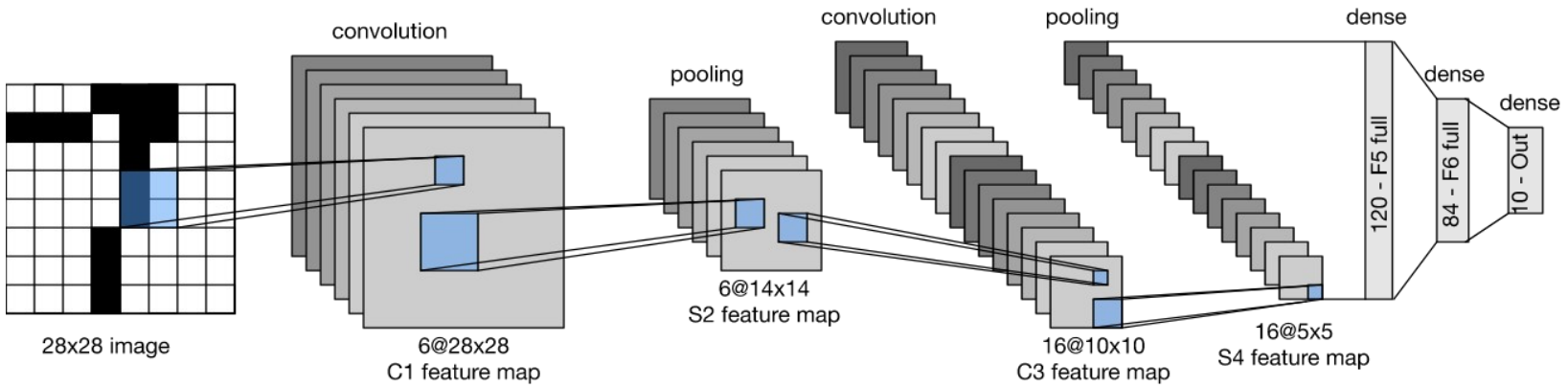
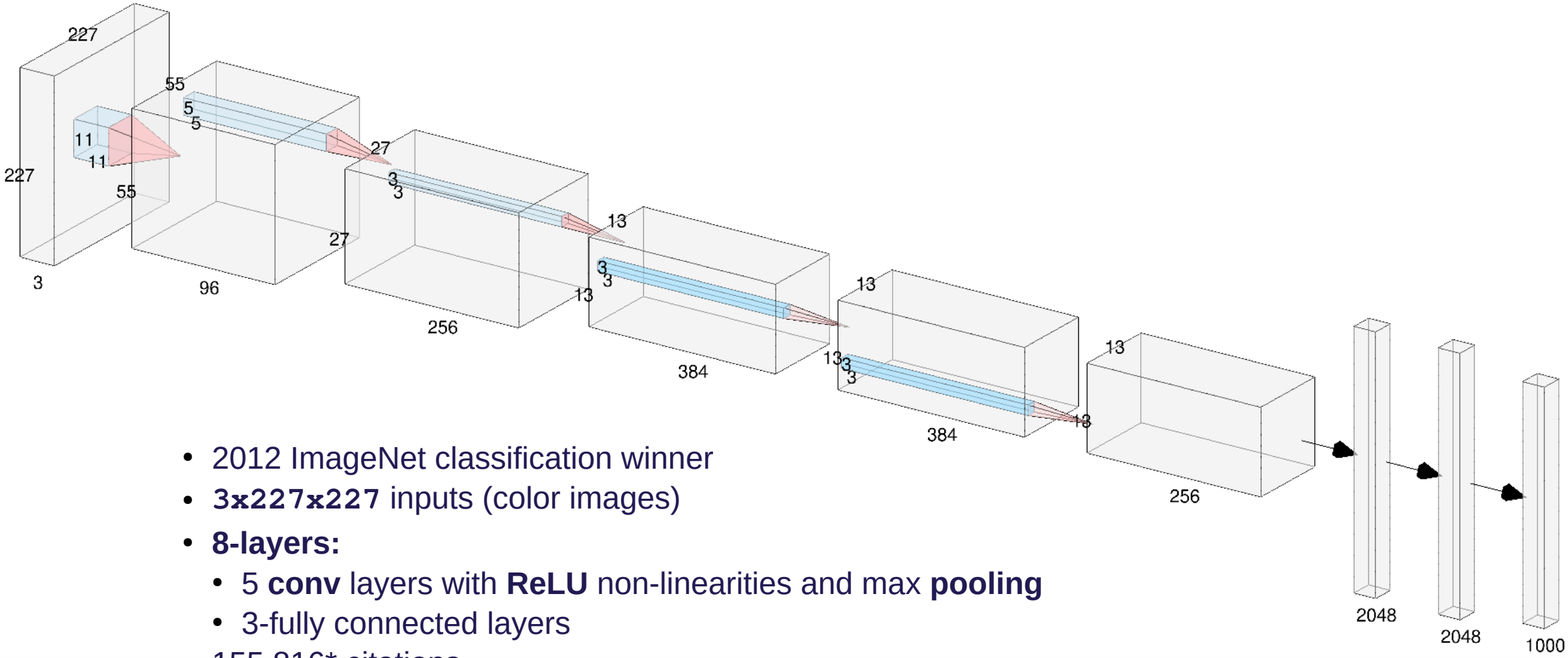


Image source: https://www.youtube.com/watch?v=FwFduRA_L6Q

AlexNet (2012)

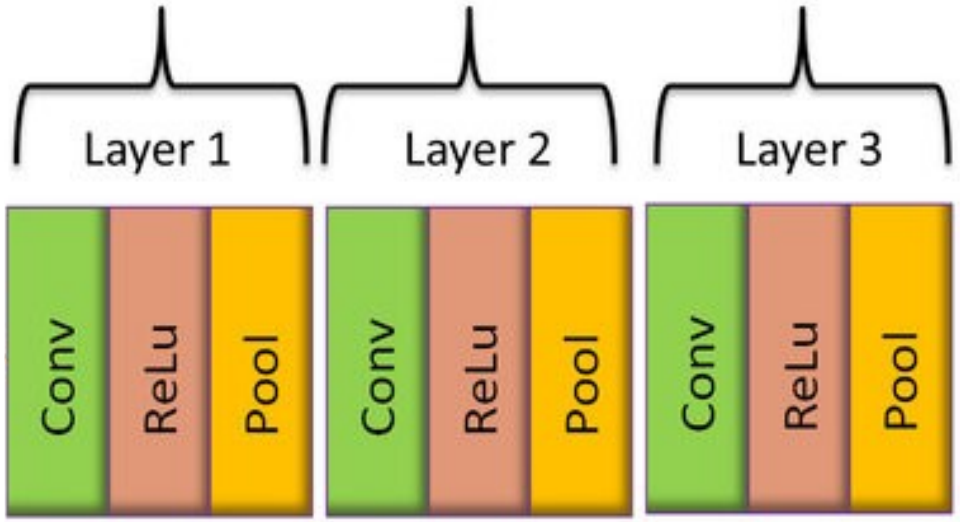


- 2012 ImageNet classification winner
- **`3x227x227`** inputs (color images)
- **8-layers:**
  - 5 **conv** layers with **ReLU** non-linearities and max **pooling**
  - 3-fully connected layers
- 155,816* citations

* According to Google Scholar, May 2024

AlexNet (2012)

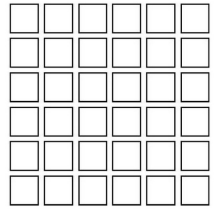
The basic CNN recipe: Conv, Relu, Pool



AlexNet Architecture (2012)

	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0

Channel num. increased



Spatial size decreased

AlexNet Architecture (2012)

	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38

Number of parameters in the first fully connected layer = channels in * channels out + channels out (bias)

= `9126*4096+4096`

= `37,725,832` 😮

AlexNet Architecture (2012)

	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38
fc7	4096		4096				4096		16	16,777	17
fc8	4096		1000				1000		4	4,096	4

Final mapping to 1000-dim class prediction (ImageNet)

AlexNet Architecture (2012)

	Input size		Layer			
Layer	C	H / W	filters	kernel	stride	pad
conv1	3	227	64	11	4	2
pool1	64	56		3	2	0
conv2	64	27	192	5	1	2
pool2	192	27		3	2	0
conv3	192	13	384	3	1	1
conv4	384	13	256	3	1	1
conv5	256	13	256	3	1	1
pool5	256	13		3	2	0
flatten	256	6				
fc6	9216		4096			
fc7	4096		4096			
fc8	4096		1000			

The parameters are chosen by a mixture of rule-of-thumbs and trial and error

AlexNet Architecture (2012)

Layer	memory (KB)	params (k)	flop (M)
conv1	784	23	73
pool1	182	0	0
conv2	547	307	224
pool2	127	0	0
conv3	254	664	112
conv4	169	885	145
conv5	169	590	100
pool5	36	0	0
flatten	36	0	0
fc6	16	37,749	38
fc7	16	16,777	17
fc8	4	4,096	4

Some interesting insights are:

- Most memory is in the early conv layers
  - (feature maps are large in wxh)

AlexNet Architecture (2012)

Layer	memory (KB)	params (k)	flop (M)
conv1	784	23	73
pool1	182	0	0
conv2	547	307	224
pool2	127	0	0
conv3	254	664	112
conv4	169	885	145
conv5	169	590	100
pool5	36	0	0
flatten	36	0	0
fc6	16	37,749	38
fc7	16	16,777	17
fc8	4	4,096	4

Some interesting insights are:

- Most memory is in the early conv layers
- Most flops occur in the conv layers
  - (many channels and large input)

AlexNet Architecture (2012)

Layer	memory (KB)	params (k)	flop (M)
conv1	784	23	73
pool1	182	0	0
conv2	547	307	224
pool2	127	0	0
conv3	254	664	112
conv4	169	885	145
conv5	169	590	100
pool5	36	0	0
flatten	36	0	0
fc6	16	37,749	38
fc7	16	16,777	17
fc8	4	4,096	4

Some interesting insights are:

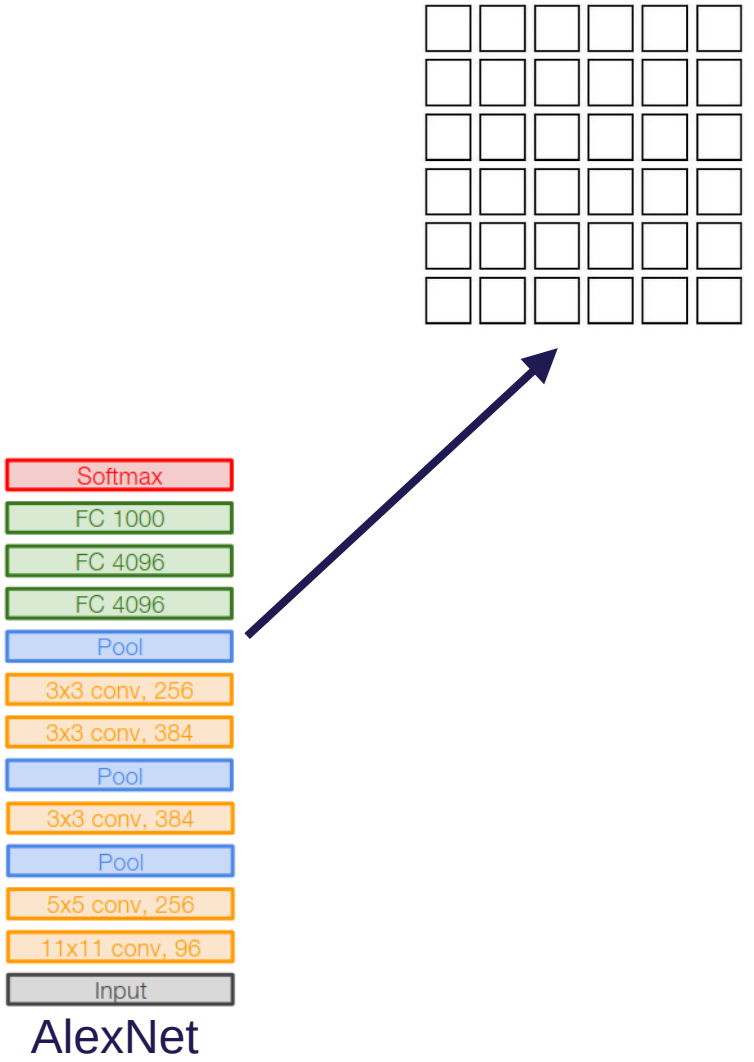
- Most memory is in the early conv layers
- Most flops occur in the conv layers
- Almost all the parameters are in the fully-connected layers

VGGNet (2014)

Challenge:

Recall that after the pool5 layer in Alexnet the **spatial resolution** was only **`6x6`**

How can we overcome this, and design a **deeper** convnet?



Simonyan et al. “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015

VGGNet (2014)

Challenge:

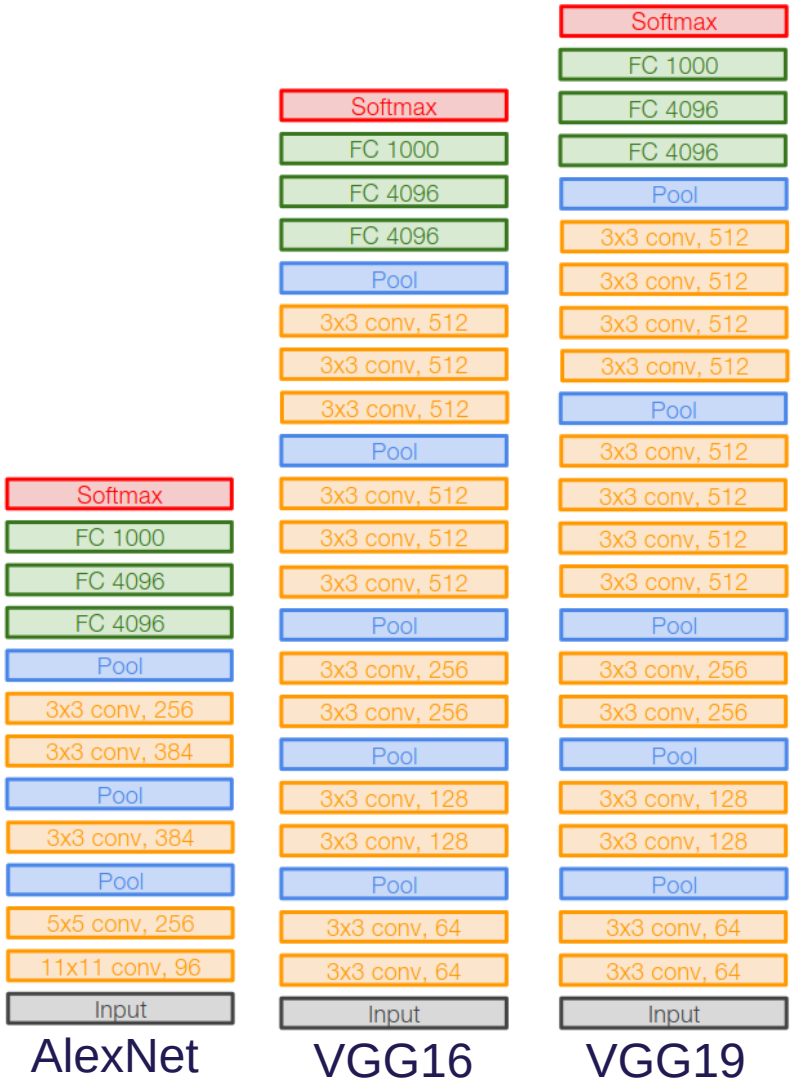
Recall that after the pool5 layer in Alexnet the **spatial resolution** was only **`6x6`**

How can we overcome this, and design a **deeper** convnet?

Solution proposed by Simonyan and Zisserman:

Move away from the typical **conv → ReLU → pool** approach and use a **sequence** of conv layers before pooling. E.g. VGG19:

Stage 1: conv → conv → pool
Stage 2: conv → conv → pool
Stage 3: conv → conv → pool
Stage 4: conv → conv → conv → conv → pool
Stage 5: conv → conv → conv → conv → pool



Simonyan et al. “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015

VGGNet (2014)

Design rules:

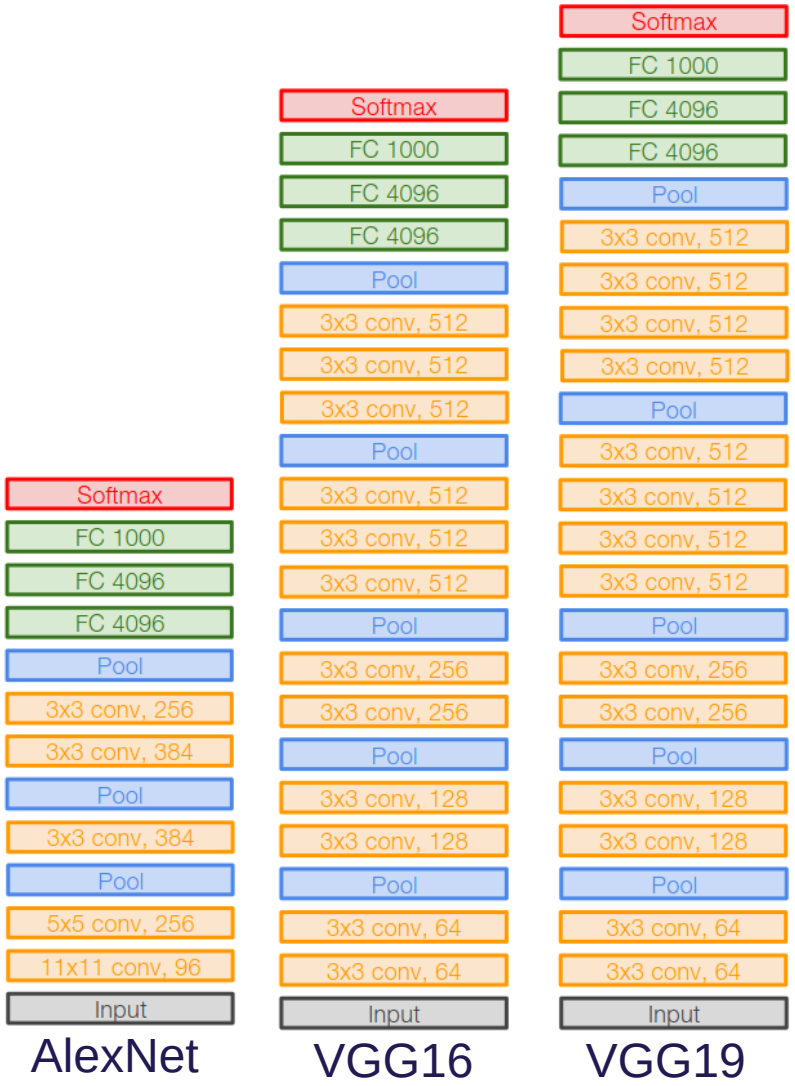
- All conv layers are use **3x3** filters, stride **1**, pad **1**
- All max pooling are **2x2**, stride **2**
- After pooling, double the channels

Rationale:

Two **3x3** conv layers have the same receptive field as one **5x5** conv, but fewer parameters and less flops. (VGG19 have **144** mio parameters)

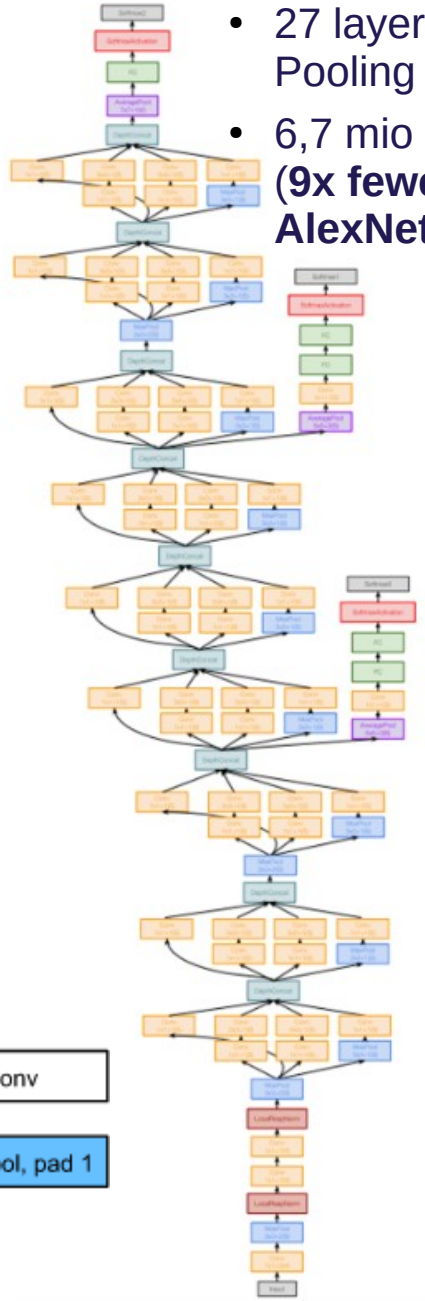
Result:
Error rate on ImageNet:
- AlexNet 8-layer: 16.4%
- VGG 19-layer: 7.3%

But the VGGNet is not very efficient (params vs. performance)



Simonyan et al. “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015

GoogLeNet / Inception v1 (2014)



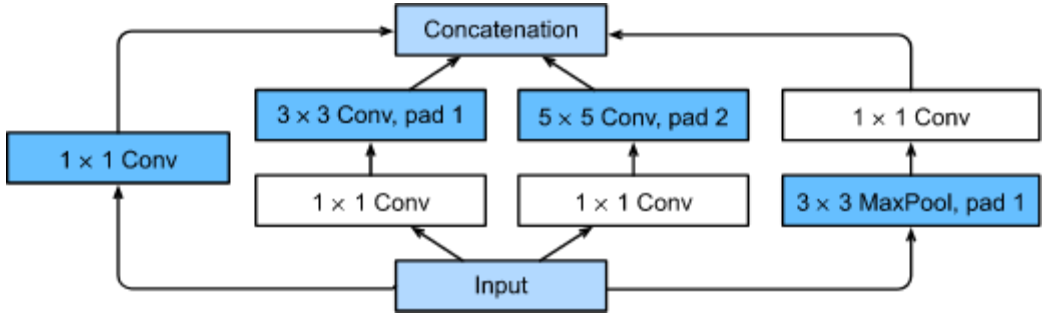
- 27 layers incl. Pooling
- 6,7 mio params (**9x fewer than AlexNet**)

A large focus on efficiency:

- **Stem → Body → Head**, a design pattern still used today
  - The **stem** extracts **low-level features** from the input and performs **aggressive spatial downsampling** to reduce **compute**
  - The **body** performs **feature transformations**
  - The **heads** maps features to the **task** at hand (e.g. classification)

- **Global average pooling,** instead of fully connected layers at the end

- **Inception modules (multi-branch net)**: concatenation of conv layers with different kernel sizes instead of relying on a single size (wider instead of deeper).



Result:
Error rate on ImageNet:
- VGG 19-layer: 7.3%
- GoogLeNet 22-layer: 6.7%

Szegedy et al, “Goind deeper with convolutions”, CVPR 2015

ResNet - Microsoft Research (2015)

Challenge:

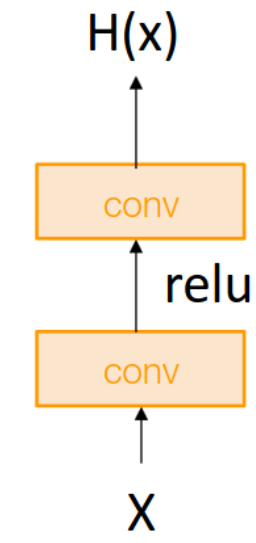
- If we keep **adding layers**, the deeper model **performs worse** than the shallow model.
- The reason was found to be **underfitting** rather than **overfitting**, mainly because the error gradient failed to propagate.

Solution:

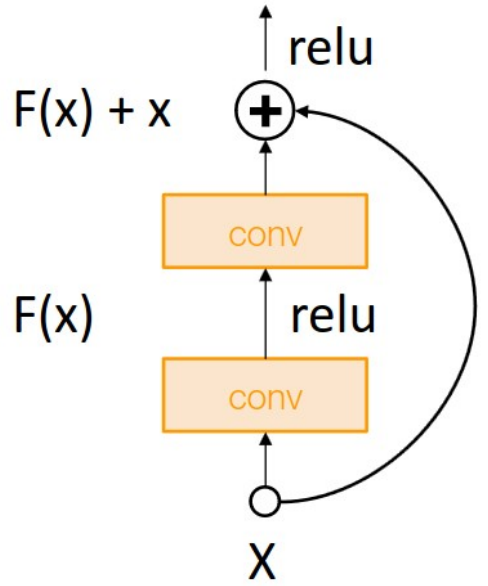
- Allow the **deep model** to **emulate a shallow** one by disabling **redundant** layers.
- If weights in the residual block are **set to 0**, the block computes the identity function.

ResNet variants:

- ResNet18
- Resnet34
- **Resnet50**
- Resnet101
- Resnet152



Vanilla block



Additive shortcut or skip-connection

Residual block

He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016


Micro Break

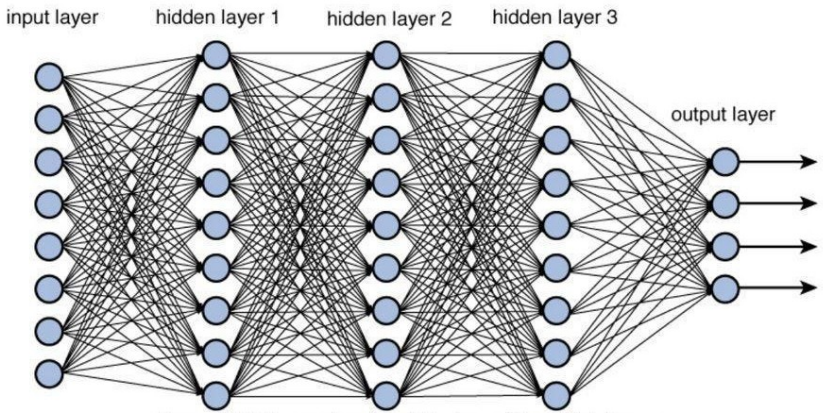
How can we train Neural Networks with millions of parameters effectively?

Overview of the Training Pipeline

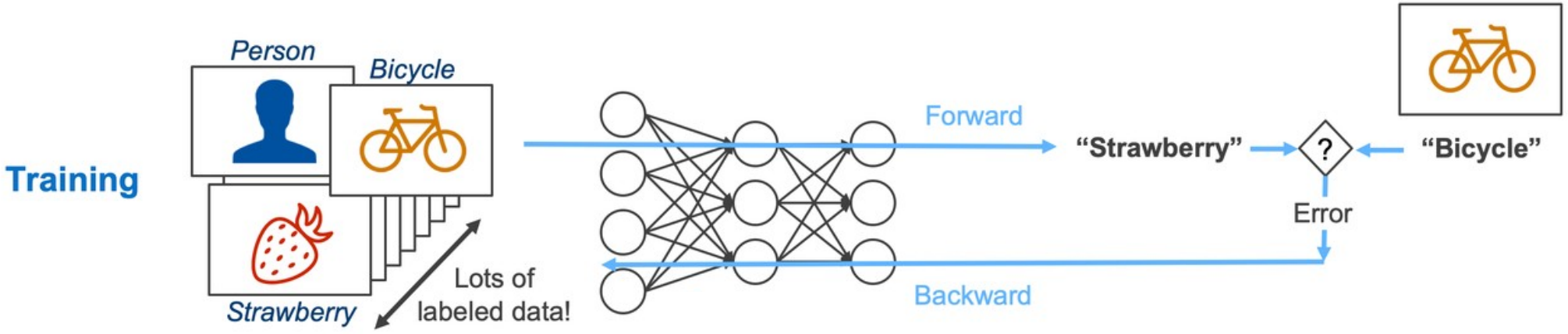
1. **Setup**
   Data pre-processing, network design, weight init., regularization, etc.

2. **Training Dynamics**
   Learning rate schedules, optimizer, hyperparameter optimization

3. **Post Training**
   Model ensembles, transfer learning, inference, deployment



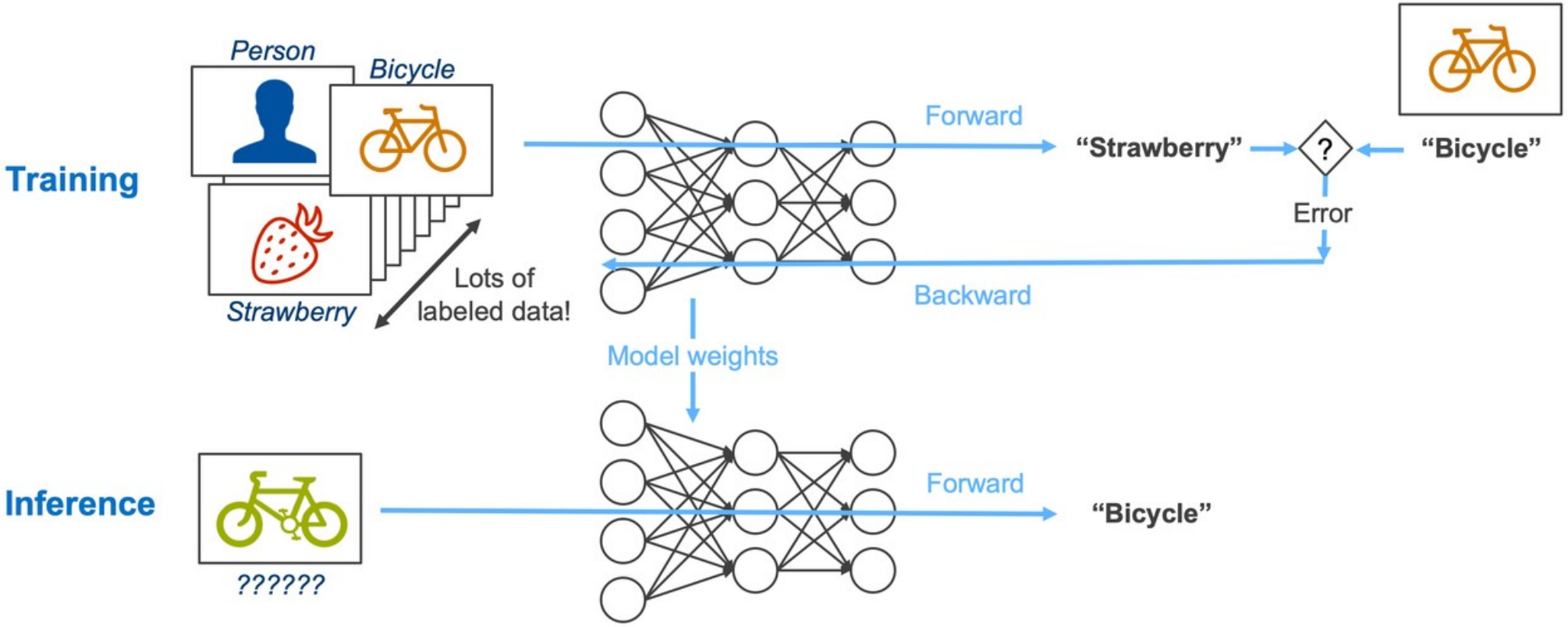
Training vs. Inference



Training refers to optimizing the weights / learning the model parameters.

https://www.linkedin.com/pulse/difference-between-deep-learning-training-inference-mark-robins-mdq8c/

Training vs. Inference



Training refers to optimizing the weights / learning the model parameters.

Inference refers to using the trained model to make predictions.

https://www.linkedin.com/pulse/difference-between-deep-learning-training-inference-mark-robins-mdq8c/

Training vs. Inference in Practice

We can control:

- **Training / Inference** via the **train()** and **eval()** methods provided through nn.Module
- **Gradient computation** turned **off** via the **no_grad()** context

Forgetting to set train() or eval() can result in **no model optimization** or **poor inference accuracy.**

PyTorch **EXAMPLE**

```
def train(args, model, train_loader):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        ## Train model using forward/backward propagation


def test(model, test_loader):
    model.eval()
    with torch.no_grad():
        for data, target in test_loader:
            ## Inference using trained model
```

Training Hyperparameters

Training Hyperparameters are parameters which are **defined before the training** is started, and therefore **not learned** from data during training.

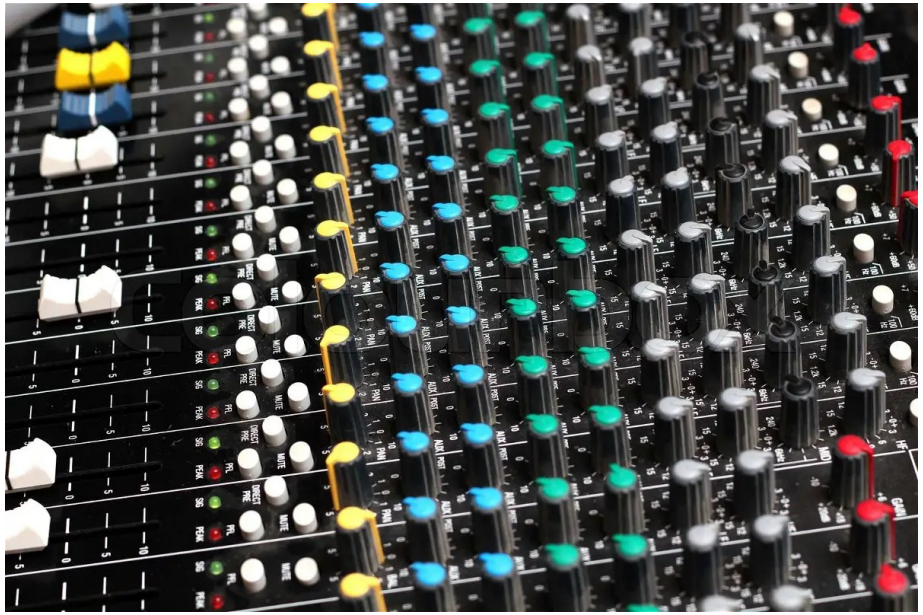
They control the training process by defining:

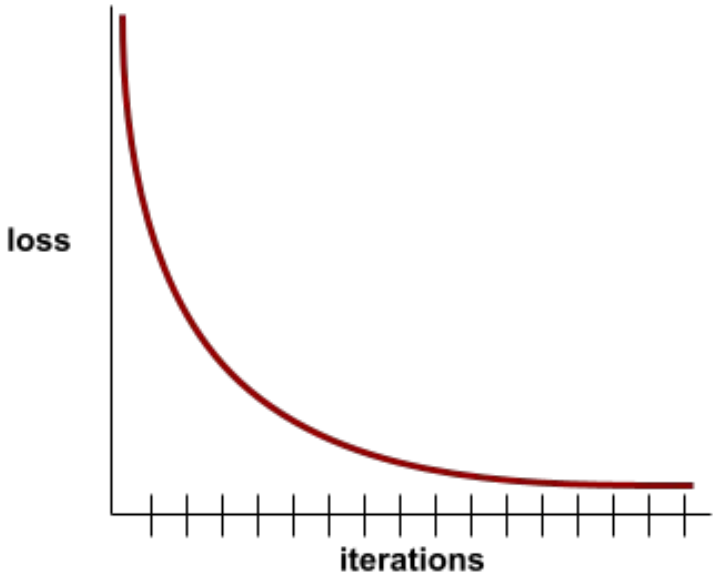
- Learning rate
- Batch size
- Number of Epochs
- Initialization method
- Optimizer strategy
- Scheduler
- And more...

Model params are learned while **hyperparams** are predefined

Intuition of Training a Neural Network

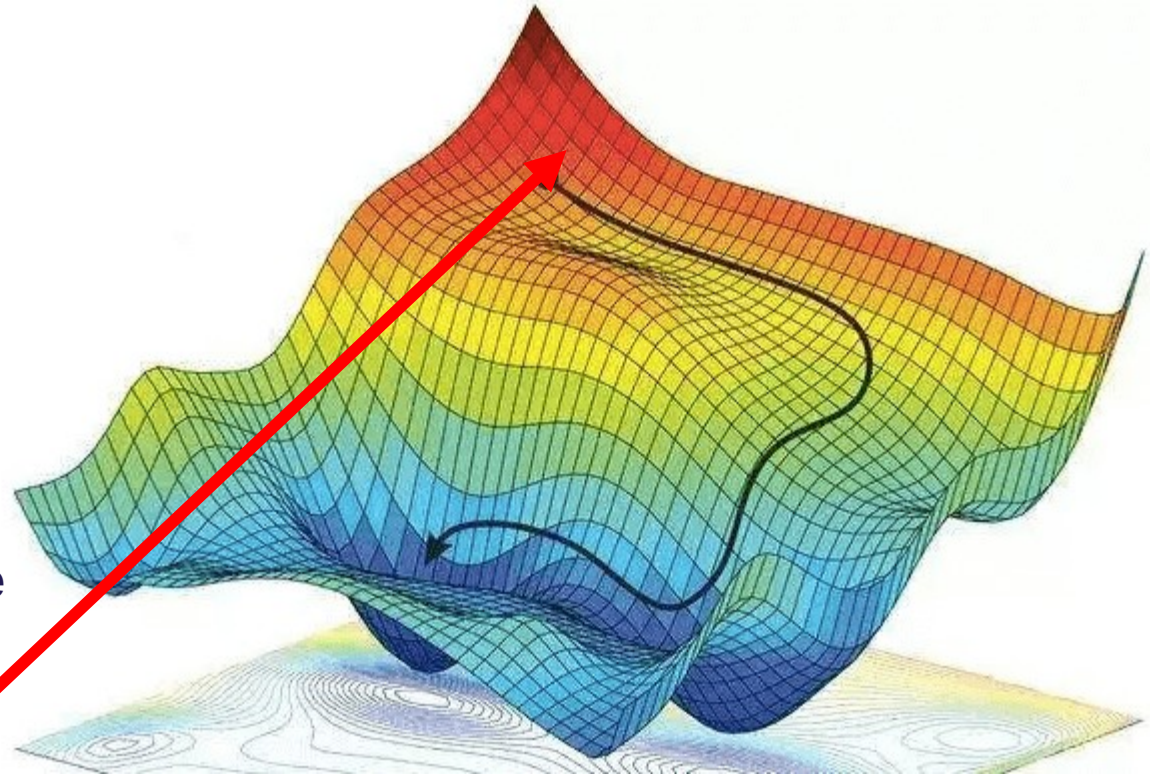
How to adjust the millions of learnable parameters to minimize the loss function?



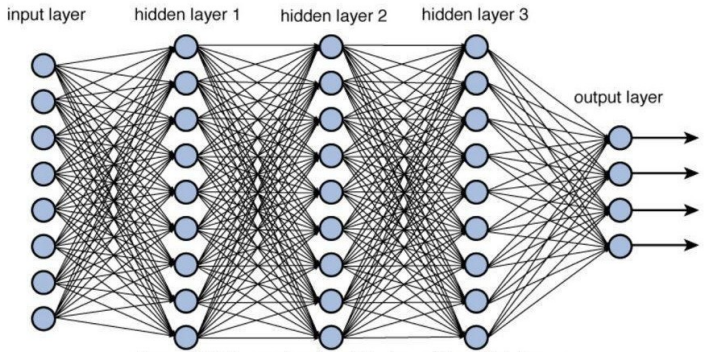


Intuition of Training a Neural Network

- In a **neural net**, the loss landscape is **complex**, due to many **composite functions** (multiple layers and nonlinear activations). We **cannot** easily **compute** the **entire loss landscape**, but only a **local area**.
- Changing **one weight** early in the **net affects later layers** on **complicated ways**
- **Computing** the **best weights** in **one go** is **not feasible** and **error prone**.



We therefore **learn from examples**, and **make small local improvements**, and **break down the gradient computation** in that process into **smaller parts**



Gradient Descent

Imagine **walking down** a **mountain step** by **step**, but blindfolded.

We can control:

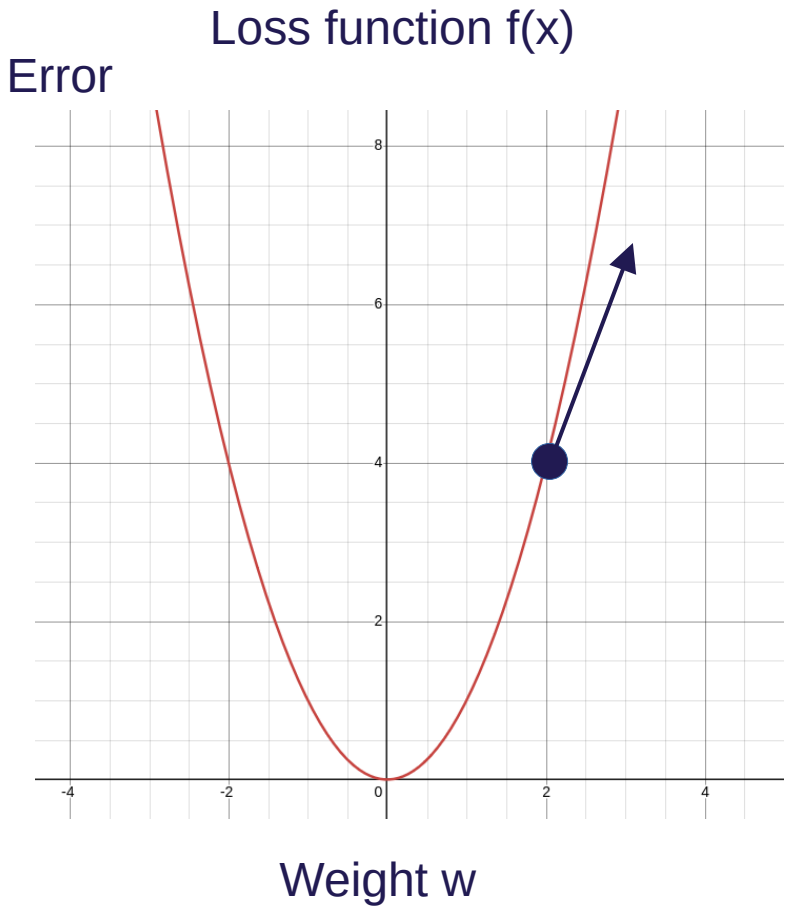
- The **direction**
- The **step size**

Which will lead us to the **bottom** of the **valley**

Gradient Descent

- An **optimization algorithm** to **minimize** a **loss function** f(x) by moving in the direction of its **steepest descent step by step**.
- The **direction** is found by taking the **derivative** of the **loss function** w.r.t the **weights**.
- Recall that the gradient of a function points in the direction of the **steepest increase** of the function.
- **Hence, the weight update of w is defined as:**

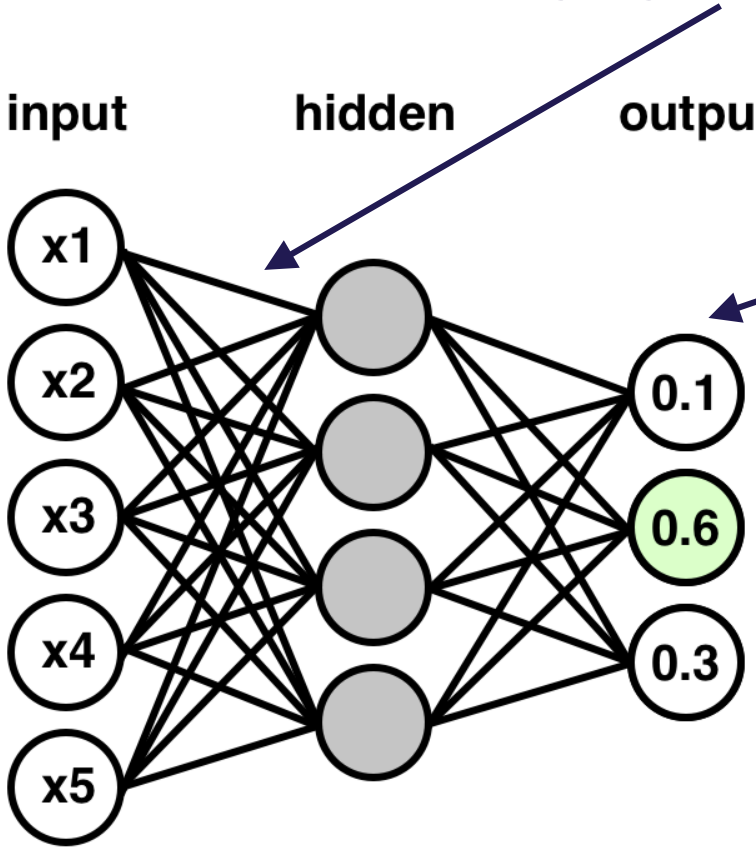
```
w_new = w - learning_rate * loss_rate_of_change_w
```

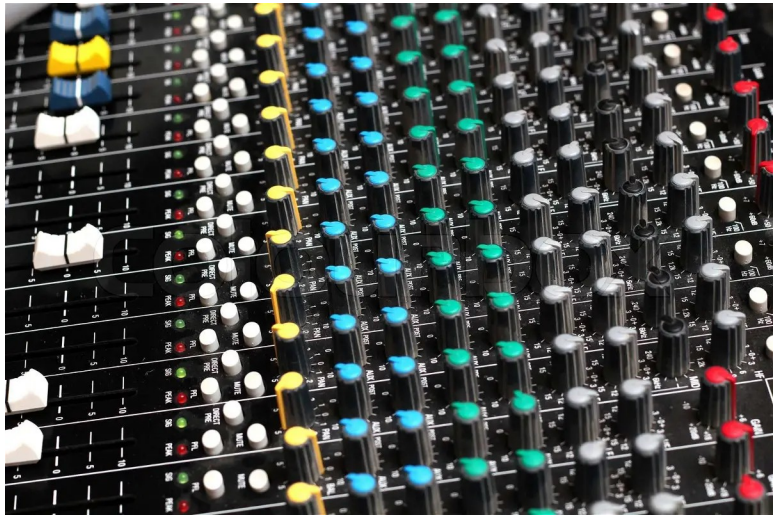


The **gradients** (partial derivatives) shows both the **direction**, and **magnitude**, of the **weight updates**.

Gradient Descent

In simple terms: By changing weight_n, what is the effect on the output_n?

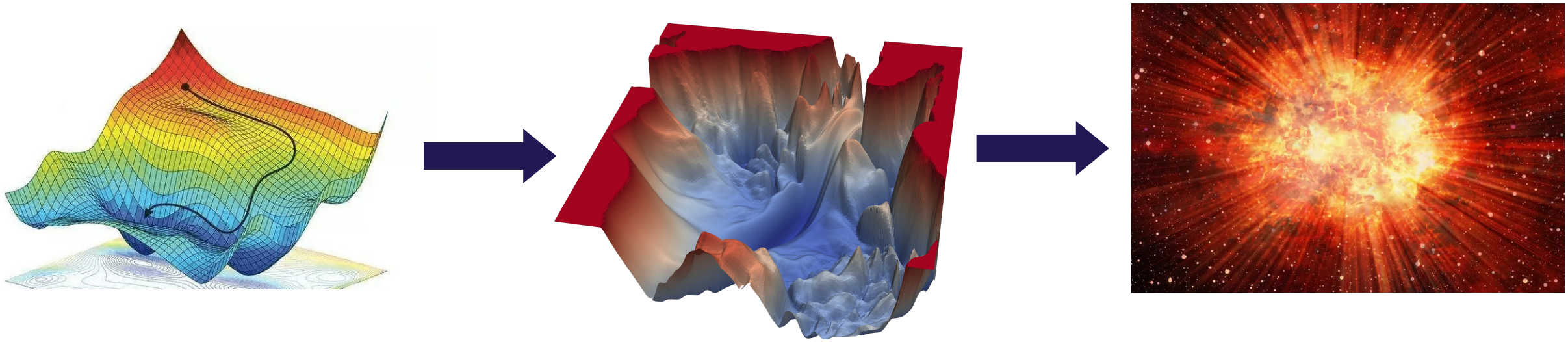




It could be a lot, or nothing! We adjust all weights according to the desired outcome

Backpropagation

When we move to models with millions of parameters, backprop can help us compute them efficiently via the chain rule.



Complexity of loss landscapes

Backpropagation

Rumelhart, Hinton, and Williams, 1986

Gradient Descent:
A general optimization algorithm that uses **gradients** to **update parameters** and hereby **minimize** a **loss** function.

Backpropagation

Rumelhart, Hinton, and Williams, 1986

The optimization strategy

Gradient Descent:
A general optimization algorithm that uses **gradients** to **update parameters** and hereby **minimize** a **loss** function.

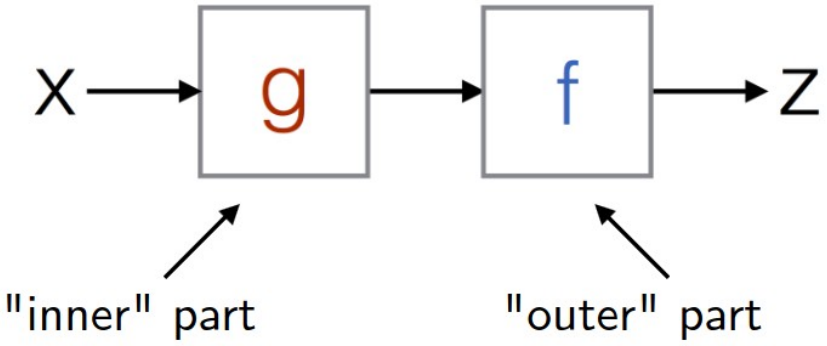
The tool to effectively compute gradients in NNs and apply the strategy

Backpropagation:
A method to **efficiently** compute gradients of the loss function w.r.t. each learnable parameters in a neural net by using the **chain-rule** starting **from the end of the network**.

Recap: Chain Rule

Decomposition of the nested function F(x):

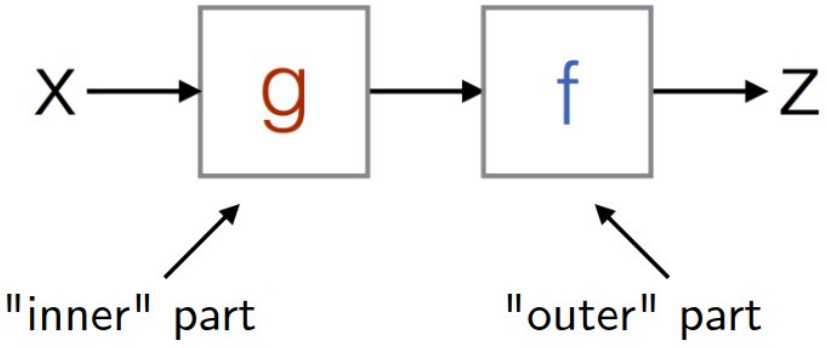
$$F(x) = f(g(x)) = z$$



Recap: Chain Rule

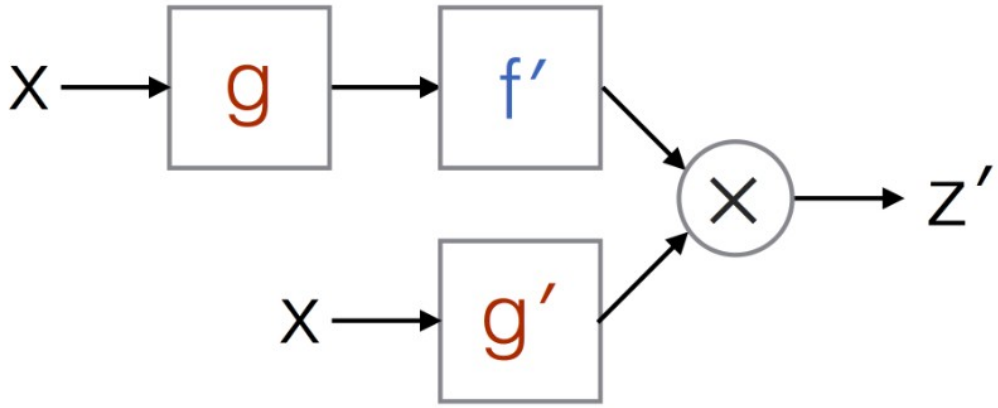
Decomposition of the nested function F(x):

$$F(x) = f(g(x)) = z$$



Derivative of F(x)

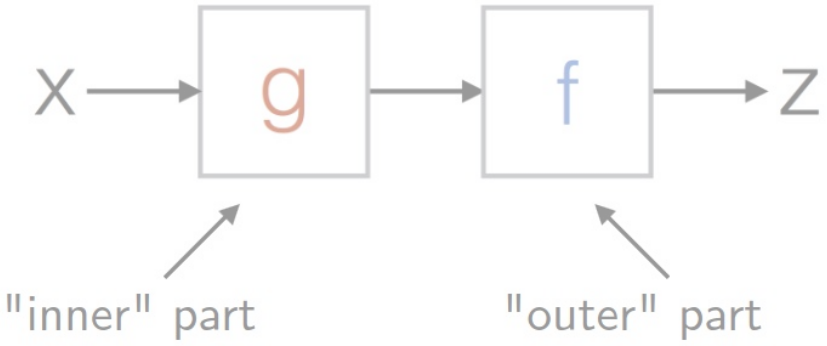
$$F'(x) = f'(g(x))\,g'(x) = z'$$



Recap: Chain Rule

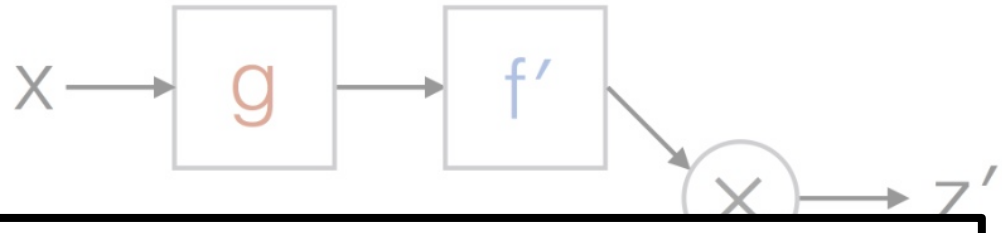
Decomposition of the nested function F(x):

$$F(x) = f(g(x)) = z$$



Derivative of F(x)

$$F'(x) = f'(g(x))\,g'(x) = z'$$



In relation to optimizing a neural with using a loss function:

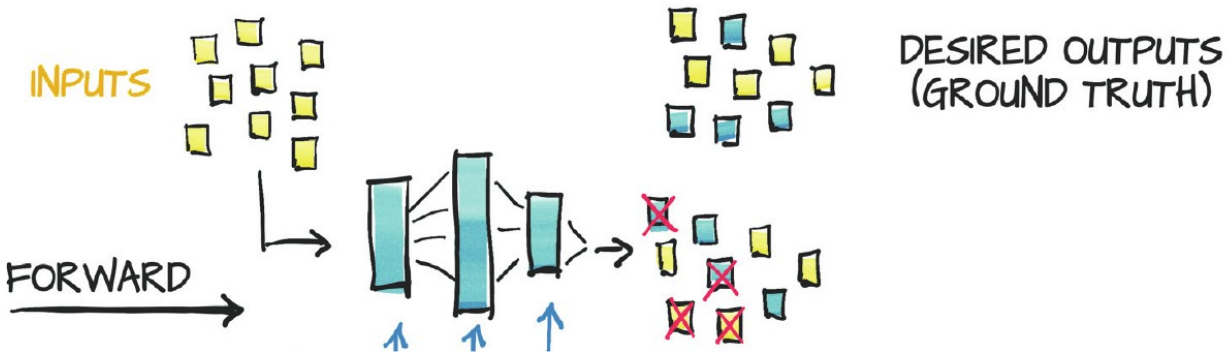
- F(x) is the loss function e.g. MSE
- G(x) is the function that maps x to y e.g. a fully connected neural net.

Backpropagation

Rumelhart, Hinton, and Williams, 1986

Backprop steps:

1. **Forward propagation:** We draw a training sample and run it through the network and compute the loss of the prediction against the ground-truth.



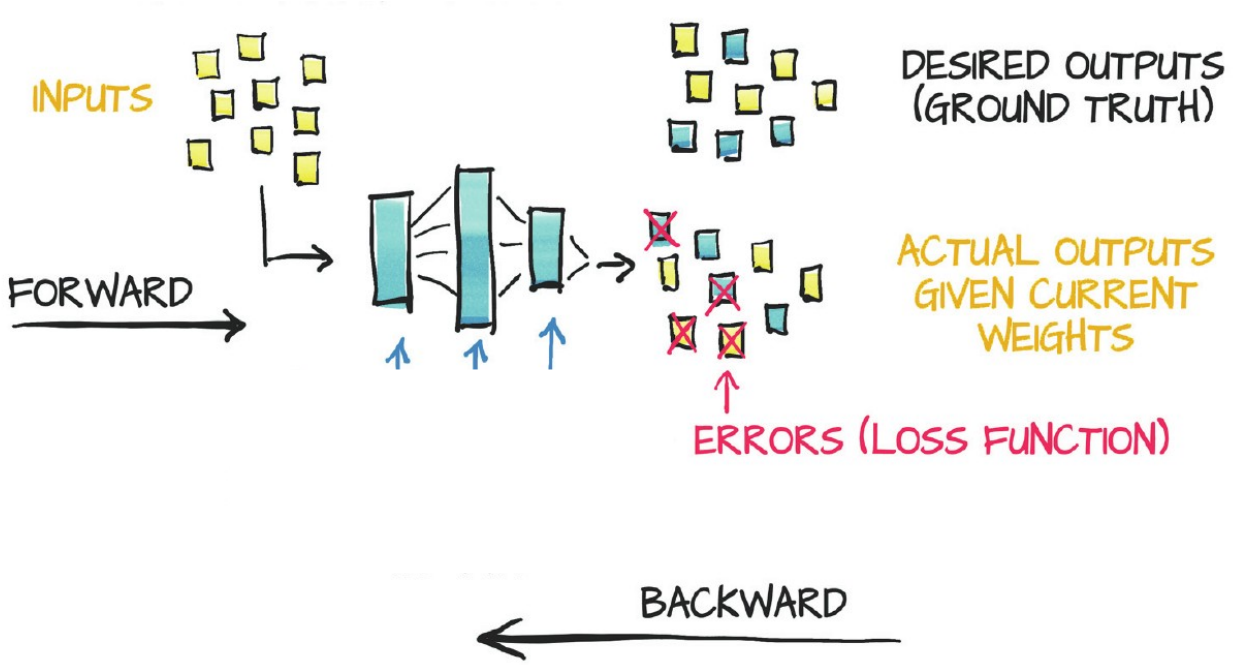
Backpropagation

Rumelhart, Hinton, and Williams, 1986

Backprop steps:

1. **Forward propagation:** We draw a training sample and run it through the network and compute the loss of the prediction against the ground-truth.

2. **Backward propagation:** We take the partial derivatives of the loss w.r.t the weights



Backpropagation

Rumelhart, Hinton, and Williams, 1986

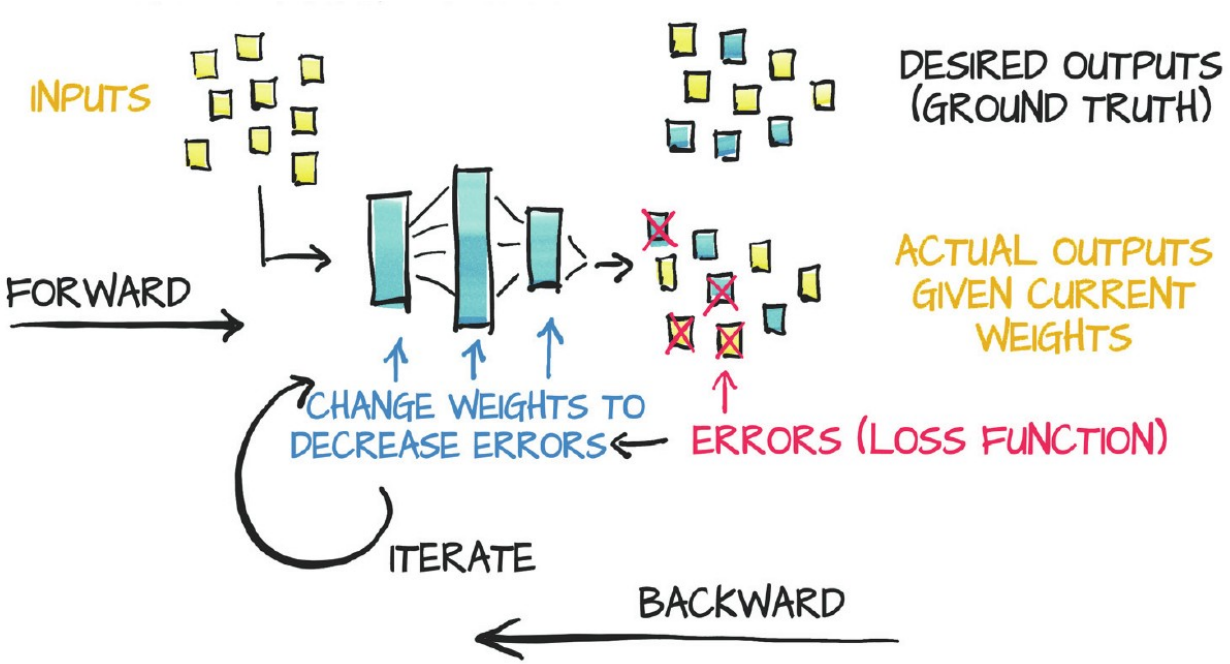
Backprop steps:

1. **Forward propagation:** We draw a training sample and run it through the network and compute the loss of the prediction against the ground-truth.

2. **Backward propagation:** We take the partial derivatives of the loss w.r.t the weights

3. **Weight Update:** We update the weights (parameters) according to the loss.

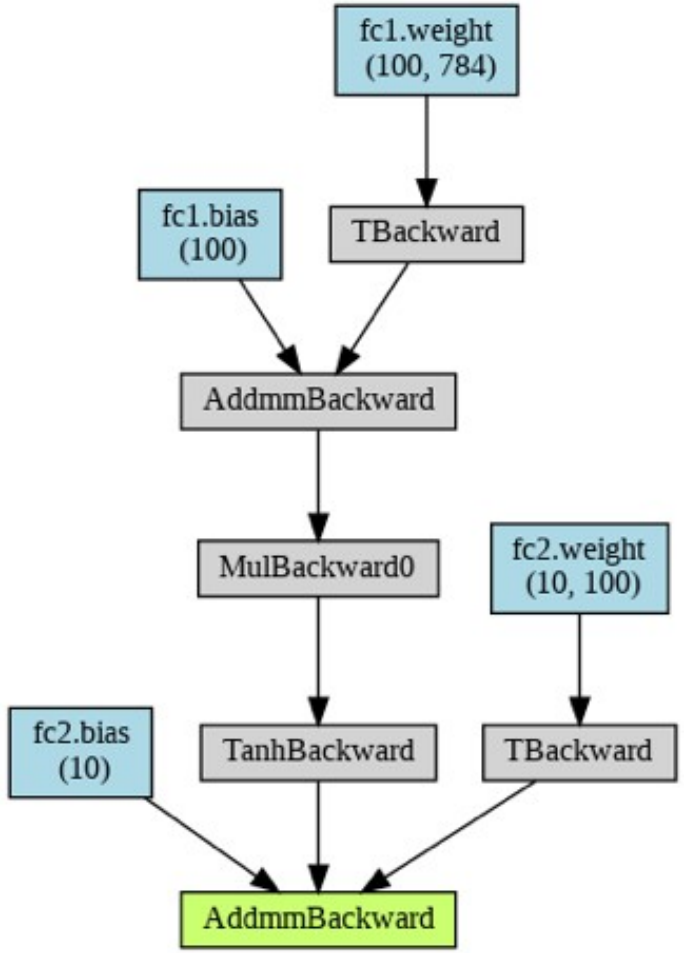
```
w_new = w - learning_rate * loss_rate_of_change_w
```



Automatic Gradient Computation

Automatic gradient (Autograd) engines to the rescue!

- **PyTorch and other frameworks automatically builds a computational graph if Autograd is enabled**
- Data and operations hereon are stored in a **directed acyclic graph** (DAG).
- In the DAG, the leaves are input tensors and the roots are output tensors, and edges are tensor operations.
- Some frameworks call this **autodiff** (e.g. Tensorflow and Keras)



https://mlconference.ai/blog/neural-networks-with-pytorch/

Utilizing Autograd

Providing `requires_grad=True` to the tensor constructor activates the autograd:

- The resulting tensors will remember where they came from (all parents and operations)
- Gradients will automatically be computed when calling backward()
- Note: requires_grad should only be set on weights and not on the data
- If you use blocks from torch.nn the `requires_grad` is already set!

PyTorch **EXAMPLE**

```
import torch

def model(input_data, model_params):
    return input_data * model_params

def loss_fn(prediction, target):
    squared_diffs = (prediction - target)**2
    return squared_diffs.mean()

learning_rate = 1e-2
input_data = torch.randn(8)
model_params = torch.randn((8), requires_grad=True)
ground_truth = torch.randn(8)

loss = loss_fn(model(input_data, model_params), ground_truth)
print("Loss:", loss.item())
print("Gradients:", model_params.grad)

loss.backward()
print("Gradients:", model_params.grad)

model_params = model_params -learning_rate * model_params.grad
loss = loss_fn(model(input_data, model_params), ground_truth)
print("Updated loss:", loss.item())
```

What do we print before and after the manual weight update?

Utilizing Autograd

Providing `requires_grad=True` to the tensor constructor activates the autograd:

- The resulting tensors will remember where they came from (all parents and operations)
- Gradients will automatically be computed
- Note: requires_grad should only be set on weights and not on the data
- If you use blocks from torch.nn the `requires_grad` is already set!

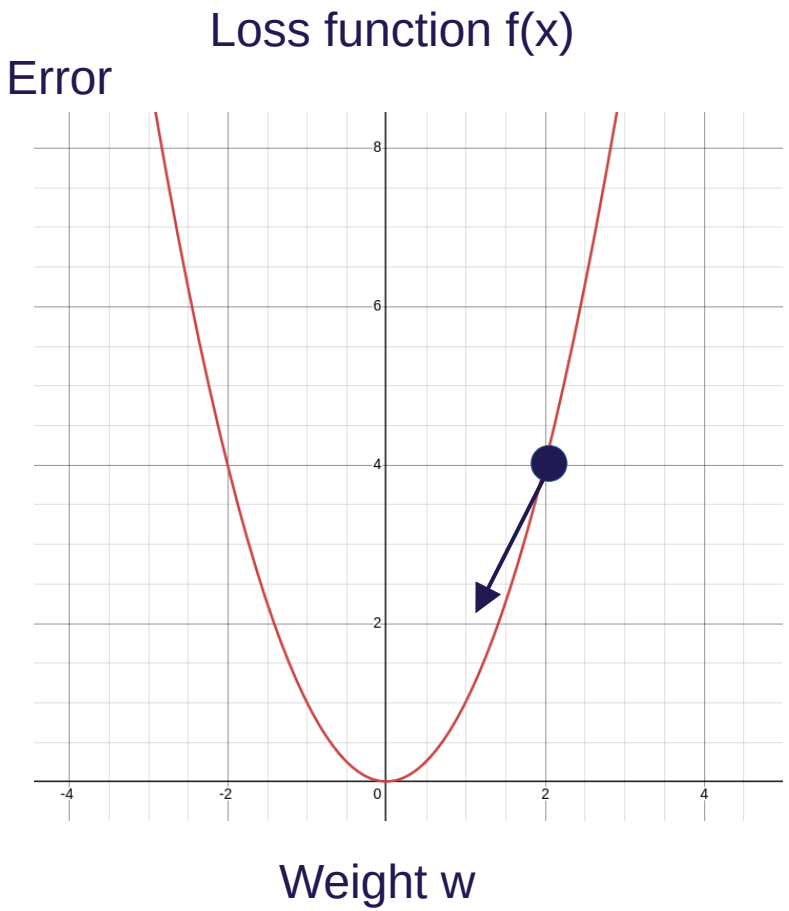
A single step →

PyTorch **EXAMPLE**

```python
import torch

def model(input_data, model_params):
    return input_data * model_params

def loss_fn(prediction, target):
    squared_diffs = (prediction - target)**2
    return squared_diffs.mean()

learning_rate = 1e-2
input_data = torch.randn(8)
model_params = torch.randn((8), requires_grad=True)
ground_truth = torch.randn(8)

loss = loss_fn(model(input_data, model_params), ground_truth)
print("Loss:", loss.item())
print("Gradients:", model_params.grad)

loss.backward()
print("Gradients:", model_params.grad)

model_params = model_params -learning_rate * model_params.grad
loss = loss_fn(model(input_data, model_params), ground_truth)
print("Updated loss:", loss.item())
```

```
andreas@aaa-thinkpad:~/Documents/teaching/DAKI-Deep-Learning$ python3 autograd.py
Loss: 1.8885011672973633
Gradients: None
Gradients: tensor([-0.0192, -0.2630,  0.2078, -0.0344, -1.1968,  0.6096, -0.0567,  0.8915])
Updated loss: 1.8615024089813232
```

Utilizing Autograd



What happens if we increase the learning rate? (lr=1, lr=10)

PyTorch **EXAMPLE**

```python
import torch

def model(input_data, model_params):
    return input_data * model_params

def loss_fn(prediction, target):
    squared_diffs = (prediction - target)**2
    return squared_diffs.mean()

learning_rate = 1
input_data = torch.randn(8)
model_params = torch.randn((8), requires_grad=True)
ground_truth = torch.randn(8)

pred = model(input_data, model_params)
loss = loss_fn(pred, ground_truth)
print("Loss:", loss.item())
print("Gradients:", model_params.grad)

loss.backward()
print("Gradients:", model_params.grad)

model_params = model_params -learning_rate * model_params.grad
loss = loss_fn(model(input_data, model_params), ground_truth)
print("Updated loss:", loss.item())
```

The Training Loop

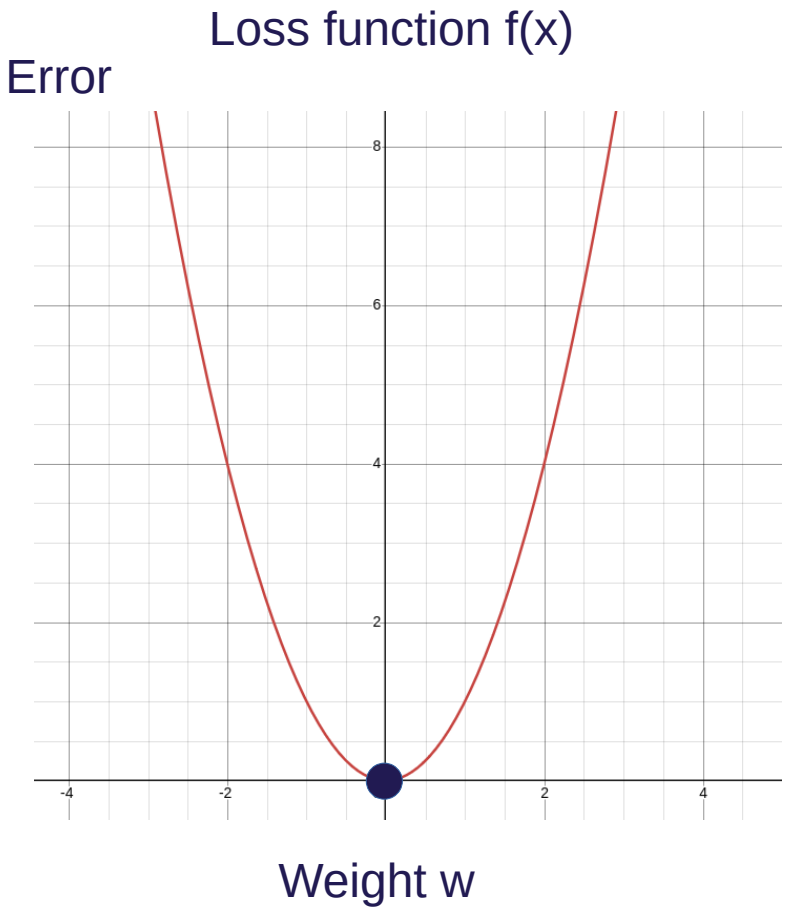
We want to **iterate** the forward/backward passes over the training data until **convergence**

```
def train_loop(n_epochs):
    for epoch in range(1, n_epochs+1):
        for i, train_data in enumerate(dataloader):

            #Forward pass
            #Loss computation
            #Backward pass
            #Weight update
```

Convergence

When further iterations yield minimal or no significant changes, specifically:

- **Convergence of the Objective Function:** This occurs when the **value of the objective function approaches a minimum value**, indicating that the algorithm has effectively found an optimal solution.
- **Convergence of Parameters:** This occurs when the **changes in the model parameters become very small**, suggesting that the optimization process has stabilized.



Simple Train Loop

PyTorch EXAMPLE

```python
import torch

def model(input_data, model_params):
    return input_data * model_params

def loss_fn(prediction, target):
    squared_diffs = (prediction - target)**2
    return squared_diffs.mean()

learning_rate = 1
iterations  = 100
input_data = torch.randn(8)
model_params = torch.randn((8), requires_grad=True)
ground_truth = torch.randn(8)

print("Initial prediction:", model(input_data, model_params))

for i in range(iterations):
    pred = model(input_data, model_params)
    loss = loss_fn(pred, ground_truth)
    if model_params.grad is not None:
        model_params.grad.zero_()

    loss.backward()

    model_params = (model_params -learning_rate * model_params.grad).detach().requires_grad_()
    loss = loss_fn(model(input_data, model_params), ground_truth)
    if i % 10 == 0:
        print("Updated loss:", loss.item())

print("Input", input_data)
print("Ground truth", ground_truth)
print("Final prediction:", model(input_data, model_params))
```

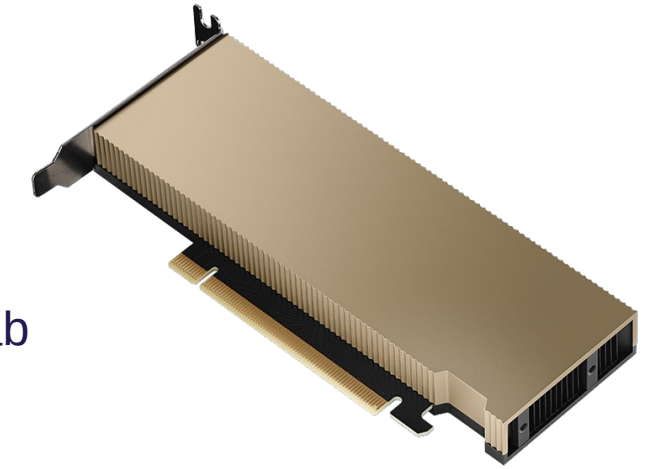
Next time:

L2: Fundamentals of Deep Learning II

- Optimizers and Schedulers
- Loss Functions
- Regularization
- Transfer Learning
- Debugging and visualization
- Advanced architectures

Introduction to Exercises

Exercises description on Moodle



- Some exercises are to be solved on AI-Lab
  - https://hpc.aau.dk/ai-lab/
- Modify existing architectures and debugging

Contact me at: anaa@create.aau.dk if you need help

Tip: Use an interactive session (not exec or squeue) in AI-Lab e.g:

```
srun --gres=gpu:1 --pty singularity shell --nv /ceph/container/pytorch/pytorch_25.02.sif
```