



Content	Page
Introduction	2
Logging into the cluster	4
Code Package and Environment	6
SLURM User Guide	13
Guideline for Job Submission	18
Shared Resources	25
MacResource	26
Other Resources for Application	37
Important Notice	38

Quick Start Guide for Using VPN to Access NTU Systems from Off Campus

1. Access from off campus

When you are off-campus or at home, you can continue to access the NTU systems and online resources so long as you have a valid Internet connection.

For specific NTU Intranet applications or file shares, you will need to connect to the University's network using the NTU VPN (Virtual Private Network) service.

2. Use VPN only when necessary

IMPORTANT: Connect to NTU VPN only when you need to access the following Intranet applications:

- 1. Animal Research Facility Management System
- 2. ARCHIBUS
- 3. Blackbaud CRM
- 4. Blockchain eCert
- 5. COPACE, WMI, NBS CRM
- 6. Enterprise, Library, OFIN File & Print service
- 7. Enterprise Timetable Planner
- 8. Jupyter Hub
- 9. Library Facility Booking System
- 10. Northhill Gym Membership Management System
- 11. Oasis QE System
- 12. Offer Letter Generation
- 13. Opera Property Management System
- 14. PACE CRM
- 15. Phonathon System
- 16. SAP System
- 17. Staff P-File
- 18. Student P-File System
- 19. TeamMate Audit Management
- 4. Launch VPN client

NOTE: VPN is NOT required for accessing Email, Teams, Ariba, ServiceNow, Workday, NTULearn or Studentlink.

Research areas

- Generative models
- Image synthesis
- Computer vision
- Deep learning

Email: xingang.pan@ntu.edu.sg

Office: N4-02c-113

Office hour: By email appointment

Schedule

- Week 7 – Convolutional Neural Network (CNN) I
- Recess Week
- Week 8 – Convolutional Neural Network (CNN) II
- Week 9 – Recurrent Neural Networks (RNN)
- Week 10 – Attention
- Week 11 – Autoencoders
- Week 12 – Generative Adversarial Networks (GAN)
- Week 13 – Revision and Selected Topics

Every week: Tutorial of previous week + Lecture of current week

SC4001 Learning Objectives

1. Interpret artificial neuron as an abstraction of biological neurons and explain how it can be used to build deep neural networks that are trained to perform various tasks such as regression and classification
2. Identify the underlying principles, architectures, and learning algorithms of various types of neural networks;
3. Select and design a suitable neural network for a given application;
4. Implement deep neural networks that can efficiently run on computing machines.

Prerequisites: MTH1810, SC1004, SC1003, SC1007
Comfortable with some Mathematics: Linear Algebra, Basic Calculus.
Need programming skills

Course Hours

Lectures: Thursday 3:30pm – 5:30pm (LT19A)

Tutorial: Wednesday 4:30pm – 5:30pm (LT1),

Tutorials start from **2nd** week

Course Topics

First Half:

1. NN fundamentals
2. Regression and Classification
3. Neuron layer
4. Deep neural networks (DNN)
5. Model selection and overfitting
6. Convolution neural networks (CNN)

Second Half:

7. Convolution neural networks (CNN) architectures
8. Recurrent neural networks (RNN) and Gated RNN
9. Attention
10. Autoencoders
11. Generative adversarial networks (GAN)

Python and PyTorch

- Python >=3.11 is the programming language
- PyTorch >=2.0 Libraries:
 - PyTorch: <https://pytorch.org/>
 - Codes of lecture examples and tutorials will be provided
 - Codes are provided as **Jupyter Notebook (.ipynb)** files.

Assistant Professor Wei Ying (First Half)

Assistant Professor Pan Xingang (Second Half)

Instructors (First Half)

Professor Wei Ying

Office: Block N4, Room 2c-117
Telephone number: (65) 67904288
E-mail: ying.wei@ntu.edu.sg
Web: <http://weiying.net/>
Office Hours: Wednesdays 5:30 – 6:30pm

TAs: Mr Liao Chang

Email: TBD

Assessment

- Programming Assignment (25%):
 - Individual: handout Feb 16, deadline March 15
- Project (35%):
 - Group of up to 3 students: handout March 15, deadline April 12
- Final exam (40%):
 - Open book

For the Assignment and the Project, **codes** and a **report** are to be submitted to NTULearn by the deadline. Late submissions will be penalized!

Group project

- Project (35%) – Group (up to three)
 - Project ideas handout: March 15
 - Deadline: April 12

The students are to propose the project and form project groups. The topic could also be selected from given project ideas.

The project includes potential research issues related to neural networks theory/applications, literature survey, and discussion/implementation of a potential solution. Comparisons with existing solutions are to be performed.

A report, codes, and a video presentation are to be submitted to NTU Learn by the deadline of one by the group members. The project report should contain the names of all the project members. No need to inform prior to report submission.

Assignments and projects

- Python and Pytorch are recommended for assignments and projects
- PC with at least 1 GPU is recommended
- Access to SCSE GPU Cluster (GPU-T1) server for those who needs computational power. Students will have accounts after add-and-drop period is over. Email: scsegpu-tr@ntu.edu.sg
- Reports are to be submitted in pdf format and codes are to be submitted in a .zip file to NTU Learn before the deadline.
- Late submissions are penalized (each day at 5% up to 3 days)
- Assessment criteria are indicated in the handout.

Text and References

Text (for additional reading)

Deep Learning, I. Goodfellow, Y. Bengio, and A. Courville, MIT Press, 2016
<http://www.deeplearningbook.org/>

References

<http://deeplearning.net/tutorial/>
http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial

PyTorch:

<https://pytorch.org/>

TensorFlow, Keras

<https://www.tensorflow.org/>

<https://keras.io/>

$$\begin{aligned}
Y &= e^x \\
Y &= xe^x \\
Y &= \frac{1}{1+e^{-x}} \\
Y &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
Y &= 1 - \gamma^2 \quad (\text{Tanh function}) \\
u \cdot w + b &\quad \frac{\partial u}{\partial w} = x
\end{aligned}$$

Summary: SGD for neurons

neuron	$f(u_p), y_p$	$\nabla_u J$
Logistic regression neuron	$f(u_p) = \frac{1}{1+e^{-u_p}}$ $y_p = 1(f(u_p) > 0.5)$	$-(d_p - f(u_p))$
Linear neuron	$y_p = u_p$	$-(d_p - y_p)$
Perceptron	$y_p = f(u_p) = \frac{1}{1+e^{-u_p}}$	$-(d_p - y_p) \cdot f'(u_p)$

Summary: GD for neurons

neuron	$f(u), y$	$\nabla_u J$
Logistic regression neuron	$f(u) = \frac{1}{1+e^{-u}}$ $y = 1(f(u) > 0.5)$	$-(d - f(u))$
Linear neuron	$y = u$	$-(d - y)$
Perceptron	$y = f(u) = \frac{1}{1+e^{-u}}$	$-(d - y) \cdot f'(u)$

Learning in two-layer FFN

GD	SGD
(X, d)	(x, d)
$Z = XW + B$	$z = W^T x + b$
$H = g(Z)$	$h = g(z)$
$U = HV + C$	$u = V^T h + c$
$Y = f(U)$	$y = f(u)$
$\nabla_U J = \begin{cases} -(D - Y) \\ -(K - f(U)) \end{cases}$	$\nabla_u J = \begin{cases} -(d - y) \\ (1(k = d) - f(u)) \end{cases}$
$\nabla_Z J = (\nabla_U J)^T \cdot g'(Z)$	$\nabla_z J = \nabla_V J \cdot V^T \cdot g'(Z)$
$W \leftarrow W - \alpha X^T \nabla_Z J$	$W \leftarrow W - \alpha x(\nabla_z J)^T$
$b \leftarrow b - \alpha (\nabla_Z J)^T \mathbf{1}_p$	$b \leftarrow b - \alpha \nabla_b J$
$V \leftarrow V - \alpha H^T \nabla_U J$	$V \leftarrow V - \alpha h(\nabla_u J)^T$
$c \leftarrow c - \alpha (\nabla_U J)^T \mathbf{1}_p$	$c \leftarrow c - \alpha \nabla_c J$

Learning a single layer

Learning a layer of neurons	
SGD	$W = W - \alpha x(\nabla_w J)^T$ $b = b - \alpha \nabla_b J$
GD	$W = W - \alpha X^T \nabla_w J$ $b = b - \alpha (\nabla_w J)^T \mathbf{1}_p$

To learn a given layer, we need to compute $\nabla_w J$ for SGD and $\nabla_U J$ for GD.

Those gradients with respect to synaptic inputs are dependent on the types of neurons in the layer.

Learning a perceptron layer

GD	SGD
(X, D)	(x, d)
$U = XW + B$	$u = W^T x + b$
$Y = f(U)$	$y = f(u)$
$\nabla_U J = -(D - Y) \cdot f'(U)$	$\nabla_u J = -(d - y) \cdot f'(u)$
$W = W - \alpha X^T \nabla_U J$	$W = W - \alpha x(\nabla_u J)^T$
$b = b - \alpha (\nabla_U J)^T \mathbf{1}_p$	$b = b - \alpha \nabla_b J$

Summary: GD for layers

layer	$f(U), Y$	$\nabla_U J$
Linear neuron layer	$Y = f(U) = U$	$-(D - Y)$
Perceptron layer	$Y = f(U) = \frac{1}{1+e^{-u}}$	$-(D - Y) \cdot f'(U)$
Softmax layer	$f(U) = \frac{e^u}{\sum_{k=1}^K e^{u_k}}$ $y = \operatorname{argmax}_k f(u)$	$-(K - f(U))$

linear neuron

GD	SGD
(X, d)	(x_p, d_p)
$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$	$J = \frac{1}{2} (d_p - y_p)^2$
$y = u = Xw + b \mathbf{1}_p$	$y_p = u_p = x_p^T w + b$
$w \leftarrow w + \alpha X^T (d - y)$	$w \leftarrow w + \alpha (d_p - y_p) x_p$
$b \leftarrow b + \alpha \mathbf{1}_p^T (d - y)$	$b \leftarrow b + \alpha (d_p - y_p)$

Gradient descent for perceptron

GD	SGD
(X, d)	(x_p, d_p)
$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$	$J = \frac{1}{2} (d_p - y_p)^2$
$u = Xw + b \mathbf{1}_p$	$u_p = x_p^T w + b$
$y = f(u)$	$y_p = f(u_p)$
$w = w + \alpha X^T (d - y) \cdot f'(u)$	$w = w + \alpha (d_p - y_p) f'(u_p) x_p$
$b = b + \alpha \mathbf{1}_p^T (d - y) \cdot f'(u)$	$b = b + \alpha (d_p - y_p) f'(u_p)$

Learning for logistic regression neuron

GD	SGD
(X, d)	(x_p, d_p)
$J = - \sum_{p=1}^P d_p \log(f(u_p)) + (1-d_p) \log(1-f(u_p))$	$j_p = -d_p \log(f(u_p)) - (1-d_p) \log(1-f(u_p))$
$u = Xw + b \mathbf{1}_p$	$u_p = x_p^T w + b$
$f(u) = \frac{1}{1+e^{-u}}$	$f(u_p) = \frac{1}{1+e^{-u_p}}$
$y = 1(f(u) > 0.5)$	$y_p = 1(f(u_p) > 0.5)$
$w \leftarrow w + \alpha X^T (d - f(u))$	$w \leftarrow w + \alpha (d_p - f(u_p)) x_p$
$b \leftarrow b + \alpha \mathbf{1}_p^T (d - f(u))$	$b \leftarrow b + \alpha (d_p - f(u_p))$

Learning a softmax layer

GD	SGD
(X, D)	(x, d)
$U = XW + B$	$u = W^T x + b$
$f(U) = \frac{e^u}{\sum_{k=1}^K e^{u_k}}$	$f(u) = \frac{e^u}{\sum_{k=1}^K e^{u_k}}$
$y = \operatorname{argmax}_k f(U)$	$y = \operatorname{argmax}_k f(u)$
$\nabla_U J = -(D - Y) \cdot f'(U)$	$\nabla_u J = -(d - y) \cdot f'(u)$
$W = W - \alpha X^T \nabla_U J$	$W = W - \alpha x(\nabla_u J)^T$
$b = b - \alpha (\nabla_U J)^T \mathbf{1}_p$	$b = b - \alpha \nabla_b J$

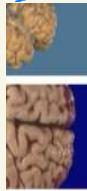
Summary: SGD for layers

layer	$f(U), Y$	$\nabla_U J$
Linear neuron layer	$y = f(U) = u$	$-(D - Y)$
Perceptron layer	$y = f(U) = \frac{1}{1+e^{-u}}$	$-(D - Y) \cdot f'(U)$
Softmax layer	$f(U) = \frac{e^u}{\sum_{k=1}^K e^{u_k}}$ $y = \operatorname{argmax}_k f(U)$	$-(K - f(U))$

Programming

material → inner product
* → dot product

Overview



Biological neural networks

Artificial neural networks are inspired by the biological neural networks in the brain. The three pounds of jelly-like material found within our brain is the most complex machine on earth and perhaps in the universe.

It consists of a **densely interconnected** set of nerve cells, or basic information-processing units, called **neurons**. Human brain incorporates nearly 10 billion neurons, each connected to about 10,000 other neurons with 60-100 trillion connections, **synapses**, between them.

By using multiple neurons simultaneously, the brain performs its functions much faster than the fastest computers in existence today. An **artificial neural network** is defined as a model of reasoning based on the principles of the brain.



Brain vs computer



- Typical operating speeds of biological neurons is in milliseconds (10^{-3} s), while silicon chip operate in nanoseconds (10^{-9} s). But Brain makes up (for slower rate of operation of a neuron) by having significant number of neurons with massive interconnections between them.

- Human brain is extremely **energy efficient**, using approximately 10^{-20} joules per operation per second, whereas the computers use around 10^{16} joules per operation per second.

- Brains have an **evolution** history of tens of millions of years, computers have been evolving for tens of decades.



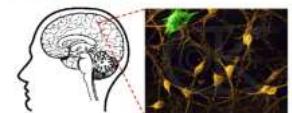
Information processing in the brain

Our brain is highly complex, non-linear parallel information-processing system.

Information is distributed throughout the whole network, rather than at specific locations and stored and processed in a neural network. Today's computers have one or several processors but each node in the brain can be considered as a single processor.

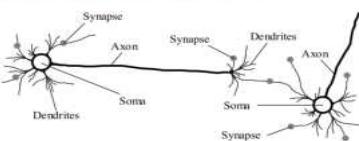
Learning is a fundamental and essential characteristic of biological neural networks. The ease with which we learn, led to attempts to emulate biological neural networks in a computer.

Biological neural networks



Each of the yellow blobs in the picture above are neuronal cell bodies (soma), and the lines are the input and output channels (dendrites and axons) which connect them.

Schematic of biological neuron



Components of biological neurons

A **biological neuron** consists of the following components:

- **Soma**: Cell body which processes incoming activation signals and converts input into output activations. The nucleus of soma contains the genetic material in the form of DNA.
- **Axon**: Transmission lines that send activation signals to other neurons
- **Dendrites**: Receptive zones that receive activation signals from other neurons
- **Synapses**: Allow weighted signal transmission between the dendrites and axons. Process of transmission is by diffusion of chemicals.

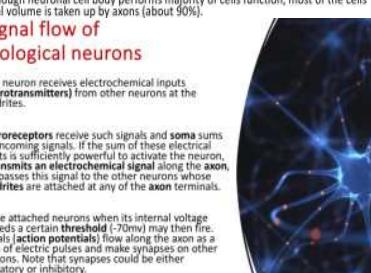
Although neuronal cell body performs majority of cells function, most of the cells total volume is taken up by axons (about 90%).

Signal flow of biological neurons

Each neuron receives electrochemical inputs (**neurotransmitters**) from other neurons at the dendrites.

Neuroreceptors receive such signals and soma sums the incoming signals. If the sum of these electrical inputs is sufficiently powerful to activate the neuron, it transmits an **electrochemical signal** along the axon, and passes this signal to the other neurons whose dendrites are attached at any of the axon terminals.

These attached neurons when its internal voltage exceeds a certain **threshold** (-70mV) may then fire. Signals (**action potentials**) flow along the axon as a form of electric pulses and make synapses on other neurons. Note that synapses could be either excitatory or inhibitory.



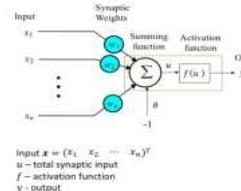
Artificial neural networks

Artificial neural networks attempt to mimic biological neural networks in the brain. The neuronal signals flow along the axon in the form of electric pulses or **action potentials**.

There are two types of artificial neural networks. One that emulates the action potentials are referred to as **spiking neural networks** and the others that emulate the aggregate of action potentials are rate-based or **activation-based neural networks**.

Neural networks discussed in this class are activation-based. However, spiking neural networks are more amenable for hardware implementations.

Artificial neuron model



Analogy between biological and artificial neurons

Biological Neuron	Artificial Neuron
Soma	Sum + Activation function
Dendrite	Input
Axon	Output
Synapse	Weight

- McCulloch-Pitts neuron (~1940) is an artificial neuron with binary inputs and outputs
- Perceptron (~1950) is another name for an artificial neuron with analog inputs

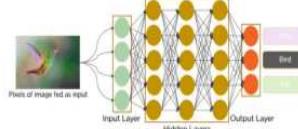
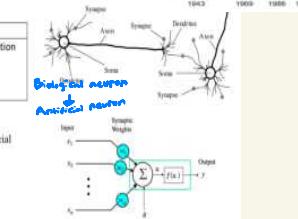
feedforward neural networks

- A **three-layer network** (two hidden layers and output layer).

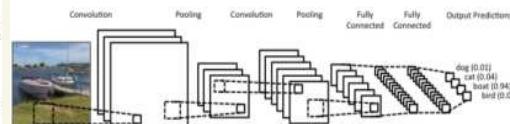
- The **input layer** consists of input nodes that receive input signals.

- The layers between input layer and output layer are referred to as **hidden layers**.

- If the **depth** (the number of layers) is large, feed forward neural networks are referred to as **deep neural networks**.



Deep convolutional neural networks



Alternate layers of convolutional and pooling, followed by fully connected layers.

Predictive analytics with neural networks

Neural network is a computational paradigm for machine learning and data analytics.



Neural networks are to be **trained** first with **training data**. Once trained, neural networks are able to **predict** labels for new data.

Predictive analytics:

Regression: outputs are continuous variables: age, height, income, etc.

Classification: outputs are discrete variables: sex, type of flowers, digits, etc.

History

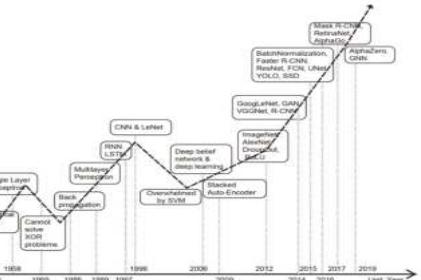
Modern view of Neural Networks (NN) began in the 1940s. – Warren McCulloch, Walter Pitts, Donald Hebb.

By late **Sixties**, most of the basic ideas and concepts necessary for neural computing had already been formulated: **perceptron**, **gradient descent learning**, etc.

Practical solution emerges for neural networks only in the mid-eighties; for example, **backpropagation algorithm** (1985).

Major reason for the delay in using large neural networks was technological: no powerful workstations to model and experiment with ANN; algorithms for learning large neural networks were unknown.

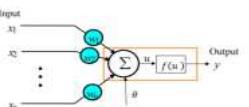
Emergence of **deep learning** since **2012**: human like performance was achieved in object recognition, using deep convolutional neural networks such as AlexNet.



Fundamentals of Neural Networks

Artificial Neuron

Notation



Vectors are denoted in bold and written as horizontally with a transpose (\top), where $1(\cdot)$ is the *Indicator function* or *Unit-step function*:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = (w_1 \ w_2 \ \cdots \ w_n)^T$$

$$\mathbf{w}^T = (w_1 \ w_2 \ \cdots \ w_n)$$

Example

$$\text{column } \mathbf{x} = \begin{pmatrix} 1.2 \\ -0.5 \\ 2.1 \end{pmatrix} = (1.2 \ -0.5 \ 3.5 \ 2.1)^T$$

$$\text{row } \mathbf{x}^T = (1.2 \ -0.5 \ 3.5 \ 2.1)$$

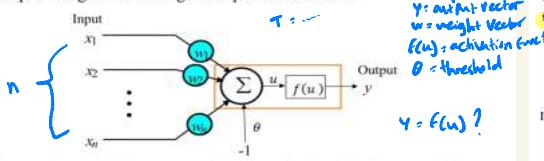
Input vector $\mathbf{x} = (x_1 \ x_2 \ \cdots \ x_n)^T$
weight vector $\mathbf{w} = (w_1 \ w_2 \ \cdots \ w_n)^T$.
 n is the number of inputs.

Artificial Neuron

An **artificial neuron** is the basic unit of neural networks.

Basic elements of an artificial neuron:

- A set of **input** signals: the input is a vector $\mathbf{x} = (x_1 \ x_2 \ \cdots \ x_n)^T$ where n is the number (or the dimension) of input signals. Inputs are also referred to as **features**.
- Inputs are connected to the neuron via synaptic connections whose strengths are represented by their **weights**.
- The weight vector $\mathbf{w} = (w_1 \ w_2 \ \cdots \ w_n)^T$ where w_i is the synaptic weight connecting i th input of the neuron.



The total **synaptic input** u to the neuron is given by the sum of the products of the inputs and their corresponding connecting weights minus the **threshold** of the neuron.

The total synaptic input to a neuron, u is given by

$$u = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n - \theta = \sum_{i=1}^n w_i x_i - \theta$$

where θ is the threshold of the neuron.

By using vector notations:

$$u = \mathbf{w}^T \mathbf{x} - \theta$$

The **activation function** f relates synaptic input to the activation of the neuron.

$f(u)$ denotes the **activation** of the neuron.

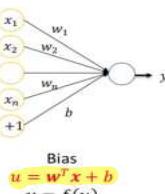
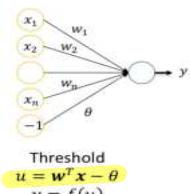
For some neurons, the **output** y is equal to the activation of the neuron.

$$y = f(u)$$

Note that activation is not generally equal to the output of the neuron.

Bias vs Threshold

The threshold is often considered as a weight with a fixed input of -1 . Often the threshold is represented as a **bias** b that receives constant $+1$ input.



$$\text{Threshold}$$

$$u = \mathbf{w}^T \mathbf{x} - \theta$$

$$y = f(u)$$

$$\text{Bias}$$

$$\text{Bias}$$

$$u = \mathbf{w}^T \mathbf{x} + b$$

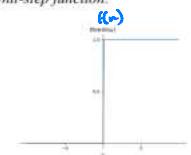
$$y = f(u)$$

Activation functions

For **threshold (unit step) activation function**, the activation is given by

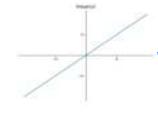
$$f(u) = \text{threshold}(u) = 1(u > 0)$$

$$1(x) = \begin{cases} 1, & x \text{ is True} \\ 0, & x \text{ is False} \end{cases}$$



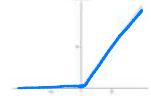
A neuron with **linear** activation function can be written as

$$f(u) = \text{linear}(u) = u$$



The **ReLU (rectified-linear unit)** activation function can be written as

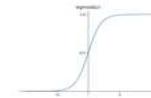
$$f(u) = \text{relu}(u) = \max(0, u)$$



Sigmoid activation function

The sigmoidal is known as the **logistic function** or simply **sigmoid function**

$$f(u) = \text{sigmoid}(u) = \frac{1}{1 + e^{-u}}$$



In general, the **sigmoid** activation function can be written as

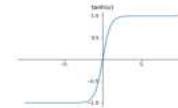
$$f(u) = \frac{a}{1 + e^{-bu}}$$

a is the gain (amplitude) and b is the slope.

But often, $a = 1.0$ and $b = 1.0$.

Tanh activation function

$$f(u) = \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$



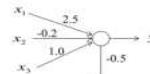
Tanh activation function has the same shape as sigmoidal and spans from -1 and $+1$. It is also known as **bipolar sigmoidal**.

Sigmoidal is the most pervasive and biologically plausible activation function. Since sigmoid function is **differentiable**, it leads to mathematically attractive neuronal models.

Example 1

The artificial neuron in the figure receives 3-dimensional inputs $\mathbf{x} = (x_1 \ x_2 \ x_3)^T$ and has an activation function given by

$$f(u) = \frac{u}{1 + e^{-1.2u}}$$



Find the synaptic input and the output of the neuron for inputs: $(0.8, -0.2, 1.0)$ and $(-0.4, 1.5, 1.0)$.

$$\mathbf{w} = \begin{pmatrix} 2.5 \\ -0.2 \\ 1.0 \end{pmatrix}, \quad b = -0.5$$

$$\text{Consider } \mathbf{x} = \begin{pmatrix} 0.8 \\ -0.2 \\ 1.0 \end{pmatrix}$$

$$\text{Synaptic input } u = \mathbf{w}^T \mathbf{x} + b = (2.5 \ -0.2 \ 1.0) \begin{pmatrix} 0.8 \\ -0.2 \\ 1.0 \end{pmatrix} - 0.5 = 0.6$$

$$\text{Output } y = f(u) = \frac{0.6}{1 + e^{-1.2 \cdot 0.6}} = 0.538$$

$$\text{Similarly, for } \mathbf{x} = \begin{pmatrix} -0.4 \\ 1.5 \\ 1.0 \end{pmatrix}, \quad u = -0.8 \text{ and output } y = f(u) = 0.222$$

$$\text{for } x_1 = 0.8, w_1 x_1 + b = (2.5 \cdot 0.8) + (-0.2) - 0.5 = 2 - 0.2 - 0.5 = 0.3$$

$$\text{for } x_2 = -0.2, w_2 x_2 + b = (2.5 \cdot -0.2) + (-0.2) - 0.5 = -1 - 0.2 - 0.5 = -1.7$$

$$\text{for } x_3 = 1.0, w_3 x_3 + b = (2.5 \cdot 1.0) + (-0.2) - 0.5 = 2.5 - 0.2 - 0.5 = 1.8$$

$$f(u) = y = \frac{0.8}{1 + e^{-1.2 \cdot 0.6}} = 0.538$$

Training (or learning) of neural networks

Neural networks attain their operating characteristics through **learning** (or **training**) with training examples. During training, the weights or the strengths of connections are gradually adjusted iteratively to achieve their desirable labels (or **targets**).

Training may be either **supervised** or **unsupervised**.



Supervised and unsupervised learning

Supervised Learning:

For each training input pattern, the network is presented with the correct **target label** (the desired output).

Unsupervised Learning:

For each training input pattern, the network adjusts weights *without* knowing the correct target.

In unsupervised training, the network **self-organizes** to classify similar input patterns into clusters.

Supervised learning

Learning of a neuron or neural network is usually performed in order to minimize a **cost function** (**loss function** or **error function**).

The cost function $J(\mathbf{W}, \mathbf{b})$ of an artificial neuron is typically a multi-dimensional function that depends on weights \mathbf{W} and the biases \mathbf{b} . The neuron learning attempts to find the optimal weights \mathbf{W}^* and biases \mathbf{b}^* that minimize the error function:

$$\mathbf{W}^*, \mathbf{b}^* = \arg \min_{\mathbf{W}, \mathbf{b}} J(\mathbf{W}, \mathbf{b})$$

Given a set of training patterns, the parameters (weights and biases) minimizing cost function are learned in an iterative procedure. In each iteration, small changes of weights $\Delta \mathbf{W}$ and biases $\Delta \mathbf{b}$ are made:

$$\begin{aligned}\mathbf{W} &\leftarrow \mathbf{W} + \Delta \mathbf{W} \\ \mathbf{b} &\leftarrow \mathbf{b} + \Delta \mathbf{b}\end{aligned}$$

Gradient descent learning

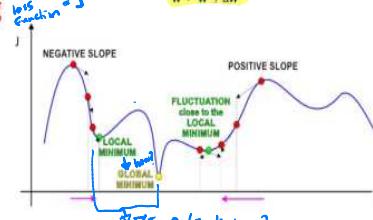
The grading descent procedure states that the weights \mathbf{W} (and biases \mathbf{b}) are updated during learning by searching in the direction of and proportional to the **negative gradient** of the cost function.

That is, the change of the weight vector:

$$\begin{aligned}\Delta \mathbf{W} &\leftarrow -\frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}} \\ \Delta \mathbf{b} &\leftarrow -\alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}\end{aligned}$$

where $\frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$ is the gradient (partial derivative) of cost with respect to weight \mathbf{W} and α is **learning factor** or **learning rate**, $\alpha \in (0, 0, 1.0)$.

The **gradient descent equations** for learning the weights is given by substituting above in



The **gradient descent equations** for the weights is given by

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$$

Similarly, for the bias

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}$$

Notation: $\nabla_{\mathbf{W}} J = \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$ and $\nabla_{\mathbf{b}} J = \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}$.

Gradient descent learning is given by

$$\begin{aligned}\mathbf{W} &\leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J \\ \mathbf{b} &\leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J\end{aligned}$$

Note: Will drop the arguments in J .

Given a set of training examples

Initialize weight \mathbf{W} and bias \mathbf{b}

Set the learning parameter α

Iterate until convergence:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$

Convergence is achieved by observing one of the following:

1. No changes in weights and biases
2. No difference between the outputs and targets
3. No decrease in the cost function J

Training dataset

Training data are also referred to as **training examples** or **training patterns**. For supervised learning, a training pattern consists of a pair consisting of input pattern and the corresponding target.

A training dataset is a set of training examples: $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$ or $\{(\mathbf{x}_1, d_1), (\mathbf{x}_2, d_2), \dots, (\mathbf{x}_p, d_p)\}$

\mathbf{x}_p is the input (features) and d_p is the target (desired label) of p th training pattern. P is the number of examples in the dataset.

The input is usually n -dimensional and written as:
 $\mathbf{x}_p = (x_{p1}, x_{p2}, \dots, x_{pn})^T$

Stochastic Gradient Descent (SGD) learning

Given training examples $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$

1 Set learning factor α

2 Initialize (\mathbf{W}, \mathbf{b})

3 Iterate until convergence:

for each pattern (\mathbf{x}_p, d_p) :

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J_p$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J_p$$

➢ In each epoch or cycle of iteration, the learning takes place individually over every pattern
 ➢ The cost J_p is individually computed from the output and the target of the p th training pattern.

Batches of Data

Inputs are often presented as a batch in a **data matrix X** and a **target vector d** . The input datapoints (or patterns) are written as rows in the data matrix and the targets are written into a single vector in the target vector.

Given a training dataset $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$

Data matrix:

$$X = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{P1} & x_{P2} & \cdots & x_{Pn} \end{pmatrix}$$

Target vector:

$$d = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$$

(Batch) Gradient descent learning

Given a set of training patterns: (X, d)

Set learning factor α

Initialize (\mathbf{W}, \mathbf{b})

Iterate until convergence:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$

The cost J is computed using all the training patterns.

That is, using (X, d)

In each epoch, the weights are updated once considering all the input patterns.

Summary

- Analogy between biological and artificial neurons
- Transfer function of artificial neuron:

$$u = \mathbf{w}^T \mathbf{x} + b$$

$$y = f(u)$$
- Types of activation functions: sigmoid, threshold, linear, ReLU, and tanh.
- Given inputs, to find the outputs for simple feedforward networks
- Supervised and unsupervised learning
- Gradient descent learning:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$
- Stochastic gradient descent (SGD) and batch gradient descent (GD)

Gradient Descent (GD) and Stochastic Gradient Descent (SGD) are optimization algorithms commonly used in machine learning for minimizing the loss function during the training of a model.

1. Gradient Descent (GD):

- GD is an iterative optimization algorithm used to minimize a cost function by adjusting the parameters (weights and biases) of a model.
- In each iteration, GD computes the gradient of the cost function with respect to the parameters of the model.
- The gradient indicates the direction of steepest ascent, so GD updates the parameters in the opposite direction to minimize the cost function.
- GD updates the parameters by taking a step proportional to the negative gradient, scaled by a learning rate hyperparameter.
- It repeats this process until convergence, where the gradient approaches zero or a predefined stopping criterion is met.

2. Stochastic Gradient Descent (SGD):

- SGD is a variant of GD that updates the parameters using a single training example (or a small batch of examples) at a time, rather than the entire dataset.
- Unlike GD, which computes the gradient using the entire dataset, SGD computes the gradient using only one example at a time.
- Since SGD updates the parameters more frequently and with noisier estimates of the gradient, it tends to converge faster but may exhibit more erratic behavior during training.
- SGD introduces randomness into the optimization process, which can help escape local minima and reach better solutions.
- SGD is particularly useful when working with large datasets or complex models where computing the gradient for the entire dataset is computationally expensive.

Differences:

1. Data Usage:

- GD computes the gradient using the entire dataset, whereas SGD computes the gradient using only one example (or a small batch of examples) at a time.

2. Convergence Behavior:

- GD typically converges to the minimum of the cost function in a more stable manner, while SGD may exhibit more erratic behavior due to the noisy estimates of the gradient.

3. Computational Cost:

- GD can be computationally expensive, especially for large datasets, since it requires computing the gradient for the entire dataset in each iteration.
- SGD is computationally cheaper than GD since it updates the parameters more frequently and processes only one example at a time.

4. Sample Efficiency:

- GD can be more sample-efficient since it utilizes information from the entire dataset in each iteration.
- SGD may require more iterations to converge but can be more efficient in terms of processing each sample.

In practice, variations of SGD, such as mini-batch SGD, which processes small batches of examples at a time, are often preferred due to their balance between computational efficiency and convergence behavior.

2. Regression n classification

Regression and classification



Primarily, neural networks are used to predict output labels from input features.

Prediction tasks can be classified into two categories:

Regression: the labels are continuous (age, income, height, etc.)

Classification: the labels are discrete (sex, digit, type of flowers, etc.).

Training finds network weights and biases that are optimal for prediction of labels from features.

Linear neuron

Synaptic input u to a neuron is given by

$$u = w^T x + b$$

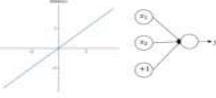
A linear neuron has a linear activation function. That is,

$$y = f(u) = u$$

A linear neuron with weights $w = (w_1, w_2, \dots, w_n)^T$ and bias b has an output:

$$y = w^T x + b$$

where input $x = (x_1, x_2, \dots, x_n)^T \in R^n$ and output $y \in R$.



Linear neuron performs linear regression

Representing a dependent (output) variable as a linear combination of independent (input) variables is known as **linear regression**.

$$\begin{aligned} y &= y_1, y_2, \dots, y_p \rightarrow \text{linear} \\ y &= y_1, y_2, \dots, y_p \rightarrow \text{plane} \end{aligned}$$

The output of a linear neuron can be written as

$$y = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

where x_1, x_2, \dots, x_n are the inputs. That is, a linear neuron performs linear regression and the weights and biases (that is, w and b) act as regression coefficients. The above function forms a **hyperplane** in Euclidean space R^P .

Given a training examples $\{(x_p, d_p)\}_{p=1}^P$, where input $x_p \in R^n$ and target $d_p \in R$, training a linear neuron finds a linear mapping $\phi: R^n \rightarrow R$ given by:

$$y = w^T x + b$$

Stochastic gradient descent (SGD) for linear neuron

The **cost function** J for regression is usually given as the **square error** (s.e.) between neuron outputs and targets.

Given a training pattern (x, d) , $\frac{1}{2}$ square error cost J is defined as

$$J = \frac{1}{2} (d - y)^2$$

where y is neuron output for input pattern x and d is the target label.

$$y = w^T x + b$$

The $\frac{1}{2}$ in the cost function is introduced to simplify learning equations and does not affect the optimal values of the parameters (weights and bias).

SGD for linear neuron

$$\begin{aligned} u &= w^T x + b = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b \\ j &= \frac{1}{2} (d - y)^2 = \frac{1}{2} (d - w^T x - b)^2 \\ y &= w^T x + b \end{aligned}$$

$$\begin{aligned} \frac{\partial J}{\partial w} &= \frac{\partial}{\partial w} \frac{1}{2} (d - w^T x - b)^2 \\ \nabla_w J &= \frac{\partial J}{\partial w} = \frac{\partial}{\partial w} (d - w^T x - b)^2 \end{aligned}$$

$$\begin{aligned} \text{Similarly, since } \frac{\partial J}{\partial b} = 1, \\ \nabla_b J &= \frac{\partial J}{\partial b} = -(d - y)x \end{aligned}$$

$$\begin{aligned} \text{Gradient learning equations:} \\ w &\leftarrow w - \alpha \nabla_w J \\ b &\leftarrow b - \alpha \nabla_b J \end{aligned}$$

By substituting from (D) and (E), SGD equations for a linear neuron are given by

$$\begin{aligned} w &\leftarrow w - \alpha(d - y)x \\ b &\leftarrow b - \alpha(d - y) \end{aligned}$$

SGD algorithm for linear neuron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$. Set learning parameter α . Initialize w and b . Repeat until convergence:

For every training pattern (x_p, d_p) :

$$\begin{aligned} y_p &= w^T x_p + b \\ w &\leftarrow w + \alpha(d_p - y_p)x_p \\ b &\leftarrow b + \alpha(d_p - y_p) \end{aligned}$$

Example 1: SGD on a linear neuron

Train a linear neuron to perform the following mapping, using stochastic gradient descent (SGD) learning:

$x^T = (x_1, x_2)$	d
(0.54, -0.96)	1.33
(0.27, 0.50)	0.45
(0.00, -0.55)	0.56
(-0.60, 0.52)	-1.66
(-0.66, -0.82)	-1.07
(0.37, 0.91)	0.30

Use a learning factor $\alpha = 0.01$.

Example 1

Gradient descent (GD) for linear neuron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$, cost function J is given by the sum of square errors (s.s.e.):

$$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$$

where y_p is the neuron output for input pattern x_p .

$$J = \sum_{p=1}^P l_p \quad (F)$$

where $l_p = \frac{1}{2} (d_p - y_p)^2$ is the square error for the p th pattern.

From (F):

$$\begin{aligned} V_w J &= \sum_{p=1}^P V_w l_p \\ &= - \sum_{p=1}^P (d_p - y_p)x_p \\ &= -(d_1 - y_1)x_1 + (d_2 - y_2)x_2 + \dots + (d_p - y_p)x_p \\ &= -(x_1 \cdot x_1 - x_2 \cdot x_2 - \dots - x_p \cdot x_p) \\ &= -x^T (d - y) \end{aligned} \quad (G)$$

where $X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}$ is the data matrix, $d = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_p \end{pmatrix}$ is the target vector, and $y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix}$ is the output vector.

Similarly, V_b can be obtained by considering inputs of +1 in (G):

$$\nabla_b J = -1_p^T (d - y) \quad (H)$$

where $1_p = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$ has P elements of 1.

The output vector y for the batch of P patterns is given by

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix} = \begin{pmatrix} x_1^T w + b \\ x_2^T w + b \\ \vdots \\ x_p^T w + b \end{pmatrix} = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_p^T \end{pmatrix} w + b \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = Xw + b 1_p$$

Substituting (G) and (H) in gradient descent equations:

$$\begin{aligned} w &\leftarrow w - \alpha \nabla_w J \\ b &\leftarrow b - \alpha \nabla_b J \end{aligned}$$

We get GD learning equations for the linear neuron as

$$\begin{aligned} w &\leftarrow w + \alpha X^T (d - y) \\ b &\leftarrow b + \alpha 1_p^T (d - y) \end{aligned}$$

And α is the learning factor.

Where:

$$y = Xw + b 1_p$$

Given a training dataset (X, d) .

Set learning parameter α .

Initialize w and b .

Repeat until convergence:

$$\begin{aligned} y &= Xw + b 1_p \\ w &\leftarrow w + \alpha X^T (d - y) \\ b &\leftarrow b + \alpha 1_p^T (d - y) \end{aligned}$$

GD

SGD

$$(X, d)$$

$$(x_p, d_p)$$

$$\frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$$

$$J = \frac{1}{2} (d_p - y_p)^2$$

$$y = Xw + b 1_p$$

$$w \leftarrow w + \alpha X^T (d - y)$$

$$b \leftarrow b + \alpha 1_p^T (d - y)$$

$$y_p = x_p^T w + b$$

$$w \leftarrow w + \alpha (d_p - y_p)x_p$$

$$b \leftarrow b + \alpha (d_p - y_p)$$

Example 1: epoch 1

x_1	x_2	y	w	b
(0.54, -0.96)	-0.19	2.29	(0.93)	0.02
(0.27, 0.50)	-1.17	0.01	(0.93)	0.03
(0.00, -0.55)	-0.37	0.87	(0.93)	0.03
(-0.60, 0.52)	0.62	0.03	(0.93)	0.02
(-0.66, -0.82)	-0.17	2.21	(0.93)	0.01
(0.37, 0.91)	0.98	0.45	(0.93)	0.00

Example 1: epoch 200

x_1	x_2	y	w	b
(0.54, -0.96)	1.49	0.03	(0.00)	-0.01
(0.27, 0.50)	0.22	0.12	(0.00)	-0.01
(0.00, -0.55)	-0.78	0.03	(0.00)	-0.01
(-0.60, 0.52)	0.31	0.00	(0.00)	-0.01
(-0.66, -0.82)	0.30	0.02	(0.00)	-0.01
(0.37, 0.91)	-1.03	0.09	(0.00)	-0.01

At convergence:

$$w = \begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$$

$$b = -0.11$$

$$\text{Mean square error: } 0.007 \text{ not zero}$$

The regression equation:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned by the linear neuron:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

Example 1

$$\begin{array}{|c|c|c|c|} \hline \text{Inputs} & \text{Predictions} & \text{Targets} & \text{Error} \\ \hline x_1 & x_2 & y & d \\ \hline 0.54 & -0.96 & 1.49 & 1.33 \\ 0.27 & 0.50 & 0.22 & 0.45 \\ 0.00 & -0.55 & -0.78 & 0.56 \\ -0.60 & 0.52 & 0.31 & -1.66 \\ -0.66 & -0.82 & 0.30 & -1.07 \\ 0.37 & 0.91 & -1.03 & 0.30 \\ \hline \end{array}$$

Targets and predictions

Targets: $y = 2.00x_1 - 0.44x_2 - 0.11$

Predictions: $y = 2.00x_1 - 0.44x_2 - 0.11$

Error: $y - y = 0.007$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.11$$

The mapping learned a hyperplane in the 3-dimensional space:

Perceptron

Perceptron is a neuron having a **sigmoid** activation function and has an output

$$y = f(u)$$

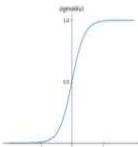
Where:

$$f(u) = \frac{1}{1+e^{-u}} = \text{sigmoid}(u)$$

And $u = w^T x + b$

The square error is used as cost function for learning.

Perceptron performs a **non-linear regression** of inputs.



SGD for perceptron



Cost function J is given by

$$J = \frac{1}{2} (d - y)^2$$

where $y = f(u)$ and $u = w^T x + b$

$$\frac{\partial J}{\partial y} = -(d - y)$$

The gradient with respect to the synaptic input:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial u} = -(d - y) f'(u) x$$

$$\text{From (C), } \frac{\partial u}{\partial w} = x \text{ and } \frac{\partial u}{\partial b} = 1.$$

$$\nabla_w J = \frac{\partial J}{\partial w} = \frac{\partial J}{\partial u} \frac{\partial u}{\partial w} = -(d - y) f'(u) x$$

$$\nabla_b J = \frac{\partial J}{\partial b} = -(d - y) f'(u)$$

Gradient learning equations:

$$w \leftarrow w - \alpha \nabla_w J$$

$$b \leftarrow b - \alpha \nabla_b J$$

Substituting gradients from (I) and (J), SGD equations for a perceptron are given by

$$w \leftarrow w + \alpha(d - y)f'(u)x$$

$$b \leftarrow b + \alpha(d - y)f'(u)$$

SGD algorithm for perceptron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$

Set learning parameter α

Initialize w and b

Repeat until convergence:

For every training pattern (x_p, d_p) :

$$u_p = w^T x_p + b$$

$$y_p = f(u_p) = \frac{1}{1+e^{-u_p}}$$

$$w \leftarrow w + \alpha(d_p - y_p)f'(u_p)x_p$$

$$b \leftarrow b + \alpha(d_p - y_p)f'(u_p)$$

GD for perceptron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$, cost function J is given by the sum of square errors (s.s.e) over all the patterns:

$$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2 = \sum_{p=1}^P J_p$$

where $J_p = \frac{1}{2} (d_p - y_p)^2$ is the square error for the p th pattern.

From (F):

$$\nabla_w J_p = \sum_{p=1}^P (\nabla_w J_p)_p$$

$$= - \sum_{p=1}^P (d_p - y_p)f'(u_p)x_p \quad \text{From (J)}$$

$$= -((d_1 - y_1)f'(u_1)x_1 + (d_2 - y_2)f'(u_2)x_2 + \dots + (d_p - y_p)f'(u_p)x_p)$$

$$= -(x_1 \cdot x_2 \cdots x_p) \cdot (d_1 - y_1)f'(u_1)$$

$$\text{product of } (d_i - y_i)f'(u_i) \text{ (element-wise product)}$$

$$\text{where } X = \begin{pmatrix} x_1 & x_2 & \cdots & x_p \end{pmatrix}, d = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_p \end{pmatrix}, y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix}, \text{ and } f'(u) = \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_p) \end{pmatrix}$$

Substituting X^T by $\mathbf{1}_p^T$ in (K), we get

$$\nabla_w J_p = -\mathbf{1}_p^T (d - y) \cdot f'(u)$$

where $\mathbf{1}_p = (1 \ 1 \ \cdots \ 1)^T$.

The gradient descent learning is given by

$$w \leftarrow w - \alpha \nabla_w J$$

$$b \leftarrow b - \alpha \nabla_b J$$

Substituting (K) and (L), we get the learning equations:

$$w \leftarrow w + \alpha X^T (d - y) \cdot f'(u)$$

$$b \leftarrow b + \alpha \mathbf{1}_p^T (d - y) \cdot f'(u)$$

Note that \cdot is the element-wise product.

Given a training dataset $\{(X, d)\}$

Set learning parameter α

Initialize w and b

Repeat until convergence:

$$u = Xw + b\mathbf{1}_p$$

$$y = f(u) = \frac{1}{1+e^{-u}}$$

$$w \leftarrow w + \alpha X^T (d - y) \cdot f'(u)$$

$$b \leftarrow b + \alpha \mathbf{1}_p^T (d - y) \cdot f'(u)$$

Gradient descent for perceptron

Example 2

Design a perceptron to learn the following mapping by using gradient descent (GD):

$x = (x_1, x_2)$	d
(0.77, 0.02)	2.91
(0.63, 0.75)	0.55
(0.50, 0.22)	1.28
(0.20, 0.76)	-0.74
(0.17, 0.09)	0.88
(0.69, 0.95)	0.30
(0.00, 0.51)	-0.28

Use learning factor $\alpha = 0.01$.

$$x = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} \quad \text{and } d = \begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix}$$

initially, $w = (0.81, 0.61)$ and $b = 0.0$
 $\alpha = 0.01$
 $\text{Output } y \in [-0.74, 2.91] \subset [-1.0, 3.0]$

Note that the sigmoid should have an amplitude ≈ 4 and shifted downwards by 1.0.
 $y = f(u) = \frac{4e^{u-1}}{1+e^{u-1}} - 1.0$

So, the activation function should be
 $y = f(u) = \frac{4e^{u-1}}{1+e^{u-1}} - 1.0$

$$f'(u) = \frac{-4e^{u-1}}{(1+e^{u-1})^2} = (y+1) \cdot \frac{-e^{u-1}}{(1+e^{u-1})^2} = (y+1)(1 - \frac{1}{1+e^{u-1}}) = \frac{1}{4}(y+1)(1-y)$$

$$\text{Epoch 1 begins...}$$

$$4w \leftarrow \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} = \begin{pmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{pmatrix}$$

$$w \leftarrow 1 + e^{-u} \cdot \begin{pmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{pmatrix} = \begin{pmatrix} 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} = \begin{pmatrix} 0.64 \\ 0.54 \\ 0.63 \\ 0.19 \\ 1.14 \\ 0.32 \end{pmatrix}$$

$$-4w \leftarrow \begin{pmatrix} 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \end{pmatrix} = \frac{4e^{u-1}}{(1+e^{u-1})^2} \cdot \begin{pmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{pmatrix} = \begin{pmatrix} 1.61 & 1.53 & 1.19 & 1.61 & 2.03 & 1.61 & 2.03 \end{pmatrix}$$

$$y = f(u) = \frac{4.0}{1+e^{-u}} - 1.0 = \begin{pmatrix} 1.61 \\ 1.53 \\ 1.19 \\ 1.61 \\ 2.03 \\ 1.61 \\ 2.03 \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.93 \\ 0.99 \\ 0.97 \\ 0.98 \\ 0.90 \\ 0.98 \end{pmatrix}$$

$$m.s.e. = \frac{1}{7} \sum_{p=1}^P (d_p - y_p)^2 = \frac{1}{7} (2.91 - 1.61)^2 + (0.55 - 1.0)^2 + \dots = 2.11$$

$$f'(u) = \frac{1}{4} (y+1) \cdot (3-y) = \frac{1}{4} \begin{pmatrix} 1.61 \\ 1.53 \\ 1.19 \\ 1.61 \\ 2.03 \\ 1.61 \\ 2.03 \end{pmatrix} = \begin{pmatrix} 3.0 & 1.90 & 3.0 & 1.53 & 3.0 & 1.61 & 3.0 \end{pmatrix}$$

$$At \text{ convergence:}$$

$$w = \begin{pmatrix} 3.35 \\ 2.80 \end{pmatrix}$$

$$b = -0.47$$

$$\text{Mean square error} = 0.01$$

$$u = x^T w + b = (x_1 \ x_2) \begin{pmatrix} 3.35 \\ 2.80 \end{pmatrix} - 0.47 = 3.35x_1 - 2.8x_2 - 0.47$$

$$y = \frac{4.0}{1+e^{-u}} - 1.0$$

$$y = \frac{4.0}{1+e^{-(3.35x_1 + 2.8x_2 - 0.47)}} - 1.0$$



Classification Example

Classification is to identify or distinguish classes or groups of objects.

Example: To identify **ballet dancers** from **rugby players**.

Two distinctive features that can aid in classification:

- weight
- height

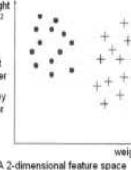


Figure: A 2-dimensional feature space

Let x_1 denote weight and x_2 denote height. Every individual is represented as a point $\mathbf{x} = (x_1, x_2)$ in the feature feature space.

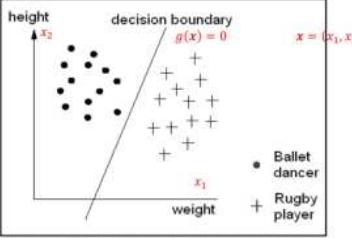


Figure: A linear classification decision boundary

Decision boundary

The **decision boundary** of the classifier, $g(\mathbf{x}) = 0$ where function $g(\mathbf{x})$ is referred to as the **discriminant function**.

A classifier finds a decision boundary separating the two classes in the feature space. On one side of the decision boundary, discriminant function is positive and on other side, discriminant function is negative.

Therefore, the following class definition may be employed:

If $g(\mathbf{x}) > 0 \Rightarrow$ Ballet dancer

If $g(\mathbf{x}) \leq 0 \Rightarrow$ Rugby player

Linear Classifier

If the two classes can be separated by a straight line, the classification is said to be **linearly separable**. For linear separable classes, one can design a **linear classifier**.

A linear classifier implements discriminant function or a decision boundary that is represented by a straight line (hyper plane) in the multidimensional **feature space**. Generally, the feature space is multidimensional. In the multidimensional space, a straight line or **hyperplane** is indicated by a linear sum of coordinates.

Given an input (features), $\mathbf{x} = (x_1 \ x_2 \ \dots \ x_n)^T$. A linear description function is given by

$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

where $w = (w_1 \ w_2 \ \dots \ w_n)^T$ are the coefficient/weights and w_0 is the constant term.

Discrete perceptron as a linear two-class classifier

The linear discriminant function can be implemented by the synaptic input to a neuron

$$g(\mathbf{x}) = u = \mathbf{w}^T \mathbf{x} + b$$

And with a threshold activation function $1(u)$:

$$u = g(\mathbf{x}) > 0 \rightarrow y = 1 \rightarrow \text{class 1}$$

$$u = g(\mathbf{x}) \leq 0 \rightarrow y = 0 \rightarrow \text{class 2}$$

That is, two-class linear classifier (or a dichotomizer) can be implemented with an artificial neuron with a threshold (unit step) activation function (**discrete perceptron**).

The output, 0 or 1, of the binary neuron represents the **label** of the class.

Classification error

In classification, the error is expressed as total mismatches between the target and output labels.

not differentiable

$$\text{Classification error} = \sum_{p=1}^P 1(d_p \neq y_p)$$

Logistic regression neuron

A **logistic regression neuron** performs a binary classification of inputs. That is, it classifies inputs into two classes with labels '0' and '1'.

The activation function of the logistic regression neuron is given by the sigmoid function:

$$f(u) = \frac{1}{1 + e^{-u}}$$

where $u = \mathbf{w}^T \mathbf{x} + b$ is the synaptic input to the neuron.

$$f(u) = \frac{1}{1 + e^{-u}}$$

The activation of a logistic regression neuron gives the **probability of the neuron output belonging to class '1'**.

$$P(y = 1|\mathbf{x}) = f(u)$$

Then

$$P(y = 0|\mathbf{x}) = 1 - P(y = 1|\mathbf{x}) = 1 - f(u)$$

A logistic regression neuron receives an input $\mathbf{x} \in R^n$ and produces a class label $y \in \{0, 1\}$ as the output.

$$f(u) = \frac{1}{1 + e^{-u}}$$

When $u = 0$, $f(u) = P(y = 1|\mathbf{x}) = P(y = 0|\mathbf{x}) = 0.5$. That is, $y = 1$ if $f(u) > 0.5$, else $y = 0$.

The output y of the neuron is given by:

$$y = 1(f(u) > 0.5) = 1(u > 0)$$

Note that for logistic neuron, the output and activation are different. It finds a linear boundary $u = 0$ separating the two classes.

SGD for logistic regression neuron

Given a training pattern (\mathbf{x}, d) where $\mathbf{x} \in R^n$ and $d \in \{0, 1\}$.

The cost function for classification is given by the **cross-entropy**:

$$J = -d \log(f(u)) - (1-d) \log(1-f(u))$$

The cost function J is minimized using the gradient descent procedure.

$$J = \begin{cases} -\log(f(u)) & \text{if } d = 1 \\ -\log(1-f(u)) & \text{if } d = 0 \end{cases}$$

$$J = -d \log(f(u)) - (1-d) \log(1-f(u))$$

where $u = \mathbf{w}^T \mathbf{x} + b$ and $f(u) = \frac{1}{1+e^{-u}}$.

Gradient with respect to u :

$$\frac{\partial J}{\partial u} = -\frac{\partial}{\partial f(u)} (\log(f(u)) + (1-d) \log(1-f(u))) \frac{\partial f(u)}{\partial u} = -\left(\frac{d}{f(u)} - \frac{(1-d)}{1-f(u)}\right) f'(u)$$

Substituting $\frac{\partial J}{\partial u}$, $\frac{\partial u}{\partial w} = \mathbf{x}$, and $\frac{\partial u}{\partial b} = 1$:

$$\nabla_w J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial w} = -(d-f(u)) \mathbf{x}$$

$$\nabla_b J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial b} = -(d-f(u))$$

Gradient learning equations:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_w J$$

$$b \leftarrow b - \alpha \nabla_b J$$

Substituting $\nabla_w J$ and $\nabla_b J$ for logistic regression neuron

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(d - f(u)) \mathbf{x}$$

$$b \leftarrow b + \alpha(d - f(u))$$

Given a training dataset $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$

Set learning rate α

Initialize \mathbf{w} and b

Iterate until convergence:

For every pattern (\mathbf{x}_p, d_p) :

$$u_p = \mathbf{w}^T \mathbf{x}_p + b$$

$$f(u_p) = \frac{1}{1+e^{-u_p}}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(d_p - f(u_p)) \mathbf{x}_p$$

$$b \leftarrow b + \alpha(d_p - f(u_p))$$

GD for logistic regression neuron

Given a training dataset $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$ where $\mathbf{x}_p \in R^n$ and $d_p \in \{0, 1\}$.

The cost function for logistic regression is given by the **cross-entropy** (or negative log-likelihood) over all the training patterns:

$$J = -\sum_{p=1}^P d_p \log(f(u_p)) + (1-d_p) \log(1-f(u_p))$$

where $u_p = \mathbf{w}^T \mathbf{x}_p + b$ and $f(u_p) = \frac{1}{1+e^{-u_p}}$.

The cost function J can be written as

$$J = \sum_{p=1}^P \nabla_w J_p$$

where $J_p = -d_p \log(f(u_p)) - (1-d_p) \log(1-f(u_p))$ is cross-entropy due to p th pattern.

$$\begin{aligned} \nabla_w J &= \sum_{p=1}^P \nabla_w J_p \\ &= -\sum_{p=1}^P (d_p - f(u_p)) \mathbf{x}_p \\ &= -((d_1 - f(u_1)) \mathbf{x}_1 + (d_2 - f(u_2)) \mathbf{x}_2 + \dots + (d_P - f(u_P)) \mathbf{x}_P) \\ &= -(x_1 \ x_2 \ \dots \ x_P) \begin{pmatrix} (d_1 - f(u_1)) \\ (d_2 - f(u_2)) \\ \vdots \\ (d_P - f(u_P)) \end{pmatrix} \\ &= -\mathbf{x}^T (\mathbf{d} - f(\mathbf{u})) \end{aligned}$$

$$\text{where } \mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix}, \mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}, \text{ and } f(\mathbf{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_P) \end{pmatrix}$$

$$\text{By substituting } \mathbf{1}_P \text{ for } \mathbf{x} \text{ in above equation:}$$

$$\nabla_b J = -\mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

Substituting the gradients in

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_w J$$

$$b \leftarrow b - \alpha \nabla_b J$$

the gradient descent learning for logistic regression neuron is given by

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u})) \\ b &\leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u})) \end{aligned}$$

Note that \mathbf{y} in the discrete perceptron is now replaced with $f(\mathbf{u})$ in logistic regression learning equations.

Given training data (\mathbf{X}, \mathbf{d})

Set learning rate α

Initialize \mathbf{w} and b

Iterate until convergence:

$$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1}_P$$

$$f(\mathbf{u}) = \frac{1}{1+e^{-\mathbf{u}}}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u}))$$

$$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

Learning for logistic regression neuron

GD	SGD
(\mathbf{x}, d)	(\mathbf{x}_p, d_p)
$J = -\sum_{p=1}^P d_p \log(f(u_p)) + (1-d_p) \log(1-f(u_p))$	$J_p = -d_p \log(f(u_p)) - (1-d_p) \log(1-f(u_p))$
$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1}_P$	$\mathbf{u}_p = \mathbf{w}^T \mathbf{x}_p + b$
$f(\mathbf{u}) = \frac{1}{1+e^{-\mathbf{u}}}$	$f(u_p) = \frac{1}{1+e^{-u_p}}$
$\mathbf{y} = 1(f(\mathbf{u}) > 0.5)$	$y_p = 1(f(u_p) > 0.5)$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u}))$	$\mathbf{w} \leftarrow \mathbf{w} + \alpha (d_p - f(u_p)) \mathbf{x}_p$
$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$	$b \leftarrow b + \alpha (d_p - f(u_p))$

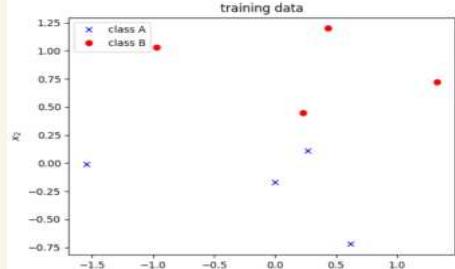
$\mathbf{G} \rightarrow \mathbf{m} \rightarrow \mathbf{n} \rightarrow \mathbf{x}$

Example 3: GD for logistic regression neuron

Train a logistic regression neuron to perform the following classification, using GD:

- $(1.33 \ 0.72) \rightarrow \text{class } B$
- $(-1.55 \ -0.01) \rightarrow \text{class } A$
- $(0.62 \ -0.72) \rightarrow \text{class } A$
- $(0.27 \ 0.11) \rightarrow \text{class } A$
- $(0.0 \ -0.17) \rightarrow \text{class } A$
- $(0.43 \ 1.2) \rightarrow \text{class } B$
- $(-0.97 \ 1.03) \rightarrow \text{class } B$
- $(0.23 \ 0.45) \rightarrow \text{class } B$

User a learning factor $\alpha = 0.04$.



Let $y = 1$ for class A and $y = 0$ for class B.

$$X = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & -0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \text{ and } d = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Initially, $w = \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix}$, $b = 0.0$ and $\alpha = 0.4$

Epoch 1:

$$u = Xw + b\mathbf{1} = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & 0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.0 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1.04 \\ -1.2 \\ 0.46 \\ 0.21 \\ 0.00 \\ 0.36 \\ -0.73 \\ 0.19 \end{pmatrix}$$

$$f(u) = \frac{1}{1 + e^{-u}} = \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix}$$

205?

$$y = 1(f(u) > 0.5) = 1 \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} > 0.5 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

d = $\begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

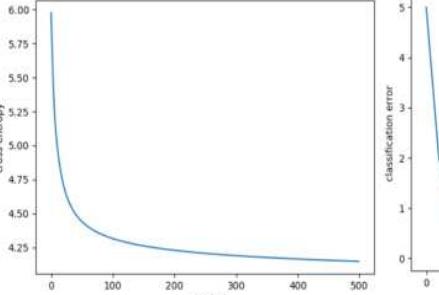
misMatch

$$\text{Classification error} = \sum_{p=1}^8 1(d_p \neq y_p) = 5$$

$$\begin{aligned} \text{Cross-entropy} &= -\sum_{p=1}^8 d_p \log(f(u_p)) + (1-d_p) \log(1-f(u_p)) \\ &= -\log(1-f(u_1)) - \log(f(u_2)) - \log(f(u_3)) - \dots - \log(1-f(u_8)) \\ &= -\log(1-0.74) - \log(0.23) - \log(0.61) - \dots - \log(1-0.55) \\ &= 6.653 \end{aligned}$$

$$\begin{aligned} w &= w + \alpha X^T(d - f(u)) \\ &= \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.04 \begin{pmatrix} 1.33 & -1.55 & 0.62 & 0.27 & 0 & 0.43 & -0.97 & 0.23 \\ -0.72 & 0.01 & -0.72 & 0.11 & -0.17 & 1.2 & 1.03 & 0.45 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0.69 \\ -0.2 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} b &= b + \alpha \mathbf{1}^T(d - f(u)) \\ &= 0.0 + 0.04(1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1) \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} = -0.09 \end{aligned}$$



At convergence, $w = \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix}$, $b = 4.47$

The decision boundary is given by: $u = x^T w + b = 0$

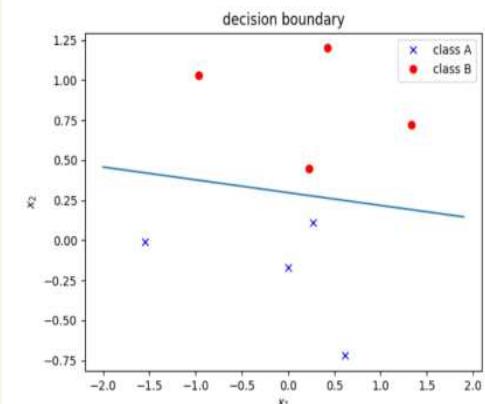
$$(x_1 \ x_2)^T \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix} + 4.47 = 0$$

$$-1.20x_1 - 15.02x_2 + 4.47 = 0$$

Decision boundary:

$$-1.20x_1 - 15.02x_2 + 4.47 = 0$$

if $x_1 = 0 \ x_2 = 0.3$
if $x_2 = 0 \ x_1 = 3.73$

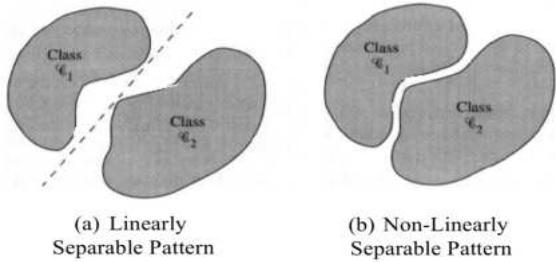


Limitations of logistic regression neuron

Learning rates

As long as the neuron is a *linear combiner* followed by a *non-linear activation function*, then regardless of the form of non-linearity used, the neuron can perform pattern classification *only on linearly separable patterns*.

Linear separability requires that the patterns to be classified must be sufficiently separated from each other to ensure that the decision boundaries are hyperplanes.



Discrete perceptron and logistic regression neuron can create only linear decision boundaries.

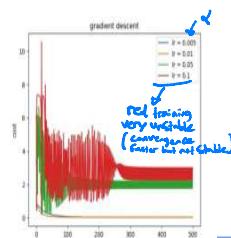
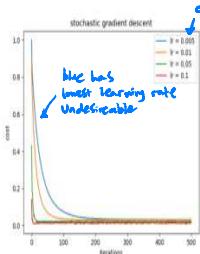
Example 4: effects of leaning rate

Design a perceptron to learn the following mapping by using gradient descent (GD):

$x = (x_1, x_2)$	d
(0.77, 0.02)	2.91
(0.63, 0.75)	0.55
(0.50, 0.22)	1.28
(0.20, 0.76)	-0.74
(0.17, 0.09)	0.88
(0.69, 0.95)	0.30
(0.00, 0.51)	-0.28

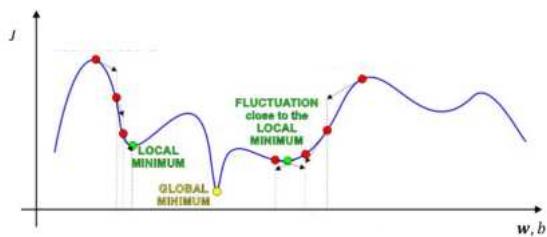
Use learning factor $\alpha = 0.01$.

Example 4: Learning rates with SGD



- At higher learning rates, convergence is faster but may not be stable.
- The *optimal learning rate* is the largest rate at which learning does not diverge.
- Generally, SGD converges to a better solution (lower error) as it capitalizes on randomness of data. SGD takes a longer time to converge.
- Usually, GD can use a higher learning rate compared to SGD; the time for one add/multiply computation per a weight update is less when patterns are trained in a (small) batch.
- In practice, *mini-batch SGD* is used. Then, the time to train a network is dependent upon
 - the learning rate
 - the batch size

Local minima problem in gradient descent learning



Algorithm may get stuck in a local minimum of error function depending on the initial weights. Gradient descent gives a suboptimal solution and does not guarantee the optimal solution.

Summary: types of neurons

Role	Neuron
Regression (one dimensional)	Linear neuron Perceptron
Classification (two classes)	Logistic regression neuron

Summary: GD for neurons

neuron	$f(u), y$	$\nabla_u f$	neuron	$f(u_p), y_p$	$\nabla_{u_p} f$
Logistic regression neuron	$f(u) = \frac{1}{1+e^{-u}}$ $y = 1(f(u) > 0.5)$	$-(d - f(u))$	Logistic regression neuron	$f(u_p) = \frac{1}{1+e^{-u_p}}$ $y_p = 1(f(u_p) > 0.5)$	$-(d_p - f(u_p))$
Linear neuron	$y = u$	$-(d - y)$	Linear neuron	$y_p = u_p$	$-(d_p - y_p)$
Perceptron	$y = f(u) = \frac{1}{1+e^{-u}}$	$-(d - y) \cdot f'(u)$	Perceptron	$y_p = f(u_p) = \frac{1}{1+e^{-u_p}}$	$-(d_p - y_p) \cdot f'(u_p)$

Summary: SGD for neurons

$$(x_p, d_p)$$

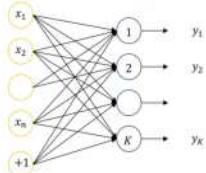
$$w = w^T x_p + b$$

$$w = w - \alpha V_{u_p} x_p$$

$$b = b - \alpha V_{u_p} b$$

Weight matrix of a layer

Consider a layer of K neurons.



Let \mathbf{w}_k and b_k denote the weight vector and bias of k th neuron. Weights connected to a neuron layer is given by a weight matrix:

$$\mathbf{W} = (\mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_K)$$

where columns are given by weight vectors of individual neurons.

And a bias vector \mathbf{b} where each element corresponds to a bias of a neuron:
 $\mathbf{b} = (b_1, b_2, \dots, b_K)^T$

Synaptic input at a layer for single input

Given an input pattern $\mathbf{x} \in \mathbb{R}^n$ to a layer of K neurons.

Synaptic input u_k to k th neuron:

$$u_k = \mathbf{w}_k^T \mathbf{x} + b_k$$

\mathbf{w}_k and b_k denote the weight vector and bias of k th neuron.

Synaptic input vector \mathbf{u} to the layer :

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{pmatrix} = \begin{pmatrix} \mathbf{w}_1^T \mathbf{x} + b_1 \\ \mathbf{w}_2^T \mathbf{x} + b_2 \\ \vdots \\ \mathbf{w}_K^T \mathbf{x} + b_K \end{pmatrix} = \begin{pmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_K^T \end{pmatrix} \mathbf{x} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{pmatrix} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$$

$$\mathbf{u} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$$

where \mathbf{W} is the weight matrix and \mathbf{b} is the bias vector of the layer.

Synaptic input at a layer for batch input

Given a set $\{\mathbf{x}_p\}_{p=1}^P$ input patterns to a layer of K neurons where $\mathbf{x}_p \in \mathbb{R}^n$.

Synaptic input \mathbf{u}_p to the layer for an input pattern \mathbf{x}_p :

$$\mathbf{u}_p = \mathbf{W}^T \mathbf{x}_p + \mathbf{b}$$

The synaptic input matrix \mathbf{U} to the layer for P patterns:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

$$\mathbf{U} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \mathbf{W} + \mathbf{b}^T \\ \mathbf{x}_2^T \mathbf{W} + \mathbf{b}^T \\ \vdots \\ \mathbf{x}_P^T \mathbf{W} + \mathbf{b}^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix} \mathbf{W} + \begin{pmatrix} \mathbf{b}^T \\ \mathbf{b}^T \\ \vdots \\ \mathbf{b}^T \end{pmatrix} = \mathbf{XW} + \mathbf{B}$$

$$\text{p} \downarrow \text{n} \quad \text{m} \downarrow \text{k} = \text{p} \times \text{n} \times \text{m} \times \text{k}$$

where rows of \mathbf{U} are synaptic inputs corresponding to individual input patterns.

The matrix $\mathbf{B} = \begin{pmatrix} \mathbf{b}^T \\ \mathbf{b}^T \\ \vdots \\ \mathbf{b}^T \end{pmatrix}$ has bias vector propagated as rows.

Activation at a layer for batch input

The synaptic input to the layer due to a batch of patterns:

$$\mathbf{U} = \mathbf{XW} + \mathbf{B}$$

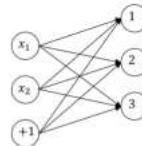
where rows of \mathbf{U} corresponds to synaptic inputs of the layer, corresponding to individual input patterns:

Activation of the layer:

$$f(\mathbf{U}) = \begin{pmatrix} f(\mathbf{u}_1^T) \\ f(\mathbf{u}_2^T) \\ \vdots \\ f(\mathbf{u}_P^T) \end{pmatrix} = \begin{pmatrix} f(\mathbf{u}_1)^T \\ f(\mathbf{u}_2)^T \\ \vdots \\ f(\mathbf{u}_P)^T \end{pmatrix}$$

where activations due to individual patterns are written as rows.

Example 1: activations and outputs of a perceptron layer



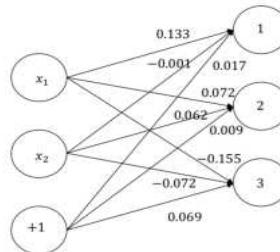
A perceptron layer of 3 neurons shown in the figure receives 2-dimensional inputs $(x_1, x_2)^T$, and has a weight matrix \mathbf{W} and a bias vector \mathbf{b} given by

$$\mathbf{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 0.017 \\ 0.009 \\ 0.069 \end{pmatrix}$$

Using batch processing, find the output for input patterns:

$$\begin{pmatrix} 0.5 \\ -1.0 \end{pmatrix}, \begin{pmatrix} 0.78 \\ -0.51 \end{pmatrix}, \begin{pmatrix} 0.64 \\ -0.65 \end{pmatrix}, \text{ and } \begin{pmatrix} 0.04 \\ -0.2 \end{pmatrix}$$

$$\mathbf{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 0.017 \\ 0.009 \\ 0.069 \end{pmatrix}$$



$$\mathbf{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \end{pmatrix}$$

Input as a batch of four patterns:

$$\mathbf{X} = \begin{pmatrix} 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}$$

The synaptic input to the layer:

$$\begin{aligned} \mathbf{U} &= \mathbf{XW} + \mathbf{B} \\ &= \begin{pmatrix} 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix} \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} + \begin{pmatrix} 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \end{pmatrix} \\ &= \begin{pmatrix} 0.085 & -0.059 & 0.111 \\ -0.115 & 0.094 & 0.26 \\ 0.121 & 0.024 & -0.005 \\ 0.022 & -0.001 & 0.077 \end{pmatrix} \end{aligned}$$

$$\mathbf{U} = \begin{pmatrix} 0.085 & -0.059 & 0.111 \\ -0.115 & 0.094 & 0.26 \\ 0.121 & 0.024 & -0.005 \\ 0.022 & -0.001 & 0.077 \end{pmatrix}$$

For a perceptron layer

$$\mathbf{Y} = f(\mathbf{U}) = \frac{1}{1 + e^{-\mathbf{U}}} = \begin{pmatrix} 0.521 & 0.485 & 0.527 \\ 0.471 & 0.476 & 0.565 \\ 0.530 & 0.506 & 0.499 \\ 0.506 & 0.500 & 0.519 \end{pmatrix}$$

For example, third row corresponding to 3rd input:

$$\mathbf{x} = \begin{pmatrix} 0.78 \\ -0.65 \end{pmatrix}$$

And the corresponding output

$$\mathbf{y} = \begin{pmatrix} 0.530 \\ 0.506 \\ 0.499 \end{pmatrix}$$

2nd neuron output for 3rd input pattern

SGD for single layer

Computational graph processing input (x, d) :



J denotes the cost function.

Need to compute gradients $\nabla_W J$ and $\nabla_b J$ to learn weight matrix W and bias vector b .

Consider k th neuron at the layer:

$$u_k = \mathbf{x}^T \mathbf{w}_k + b_k$$

And

$$\frac{\partial u_k}{\partial \mathbf{w}_k} = \mathbf{x}$$

The gradient of the cost with respect to the weight connected to k th neuron:

$$\nabla_{\mathbf{w}_k} J = \frac{\partial J}{\partial u_k} \frac{\partial u_k}{\partial \mathbf{w}_k} = \mathbf{x} \frac{\partial J}{\partial u_k} \quad (\text{A})$$

$$\nabla_{b_k} J = \frac{\partial J}{\partial u_k} \frac{\partial u_k}{\partial b_k} = \frac{\partial J}{\partial u_k} \quad (\text{B})$$

Gradient of J with respect to $W = (\mathbf{w}_1 \ \mathbf{w}_2 \ \dots \ \mathbf{w}_K)$:

$$\begin{aligned} \nabla_W J &= (\nabla_{\mathbf{w}_1} J \ \nabla_{\mathbf{w}_2} J \ \dots \ \nabla_{\mathbf{w}_K} J) \\ &= \left(\mathbf{x} \frac{\partial J}{\partial u_1} \ \mathbf{x} \frac{\partial J}{\partial u_2} \ \dots \ \mathbf{x} \frac{\partial J}{\partial u_K} \right) \quad \text{From (A)} \\ &= \mathbf{x} \left(\frac{\partial J}{\partial u_1} \ \frac{\partial J}{\partial u_2} \ \dots \ \frac{\partial J}{\partial u_K} \right) \\ &= \mathbf{x} (\nabla_u J)^T \end{aligned}$$

where

$$\nabla_u J = \frac{\partial J}{\partial \mathbf{u}} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix}$$

That is, $\nabla_W J = \mathbf{x} (\nabla_u J)^T$ (C)

Similarly, by substituting $\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial u_k}$ from (B):

$$\nabla_b J = \begin{pmatrix} \frac{\partial J}{\partial b_1} \\ \frac{\partial J}{\partial b_2} \\ \vdots \\ \frac{\partial J}{\partial b_K} \end{pmatrix} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix} = \nabla_u J \quad (\text{D})$$

$$\nabla_b J = \nabla_u J$$

From (C) and (D),

$$\begin{aligned} \nabla_W J &= \mathbf{x} (\nabla_u J)^T \\ \nabla_b J &= \nabla_u J \end{aligned}$$

That is, by computing gradient $\nabla_u J$ with respect to synaptic input u , the gradient of cost J with respect to the weights and biases is obtained.

$$\begin{aligned} \mathbf{W} &= \mathbf{W} - \alpha \mathbf{x} (\nabla_u J)^T \\ \mathbf{b} &= \mathbf{b} - \alpha \nabla_u J \end{aligned}$$

GD for single layer

Given a set of patterns $\{(x_p, d_p)\}_{p=1}^P$ where $x_p \in R^n$ and $d_p \in R^K$ for regression and $d_p \in \{1, 2, \dots, K\}$ for classification.

The cost J is given by the sum of cost due to individual patterns:

$$J = \sum_{p=1}^P J_p$$

Where Then,

$$\nabla_W J = \sum_{p=1}^P \nabla_W J_p$$

Substituting $\nabla_W J_p = x_p (\nabla_{u_p} J_p)^T$ from (C):

$$\begin{aligned} \nabla_W J &= \sum_{p=1}^P x_p (\nabla_{u_p} J_p)^T \\ &= \sum_{p=1}^P x_p (\nabla_{u_p} J)^T \quad \text{since } \nabla_{u_p} J = \nabla_{u_p} J_p \\ &= x_1 (\nabla_{u_1} J)^T + x_2 (\nabla_{u_2} J)^T + \dots + x_P (\nabla_{u_P} J)^T \\ &= (x_1 \ x_2 \ \dots \ x_P) \begin{pmatrix} (\nabla_{u_1} J)^T \\ \vdots \\ (\nabla_{u_P} J)^T \end{pmatrix} \\ &= X^T \nabla_u J \end{aligned} \quad (\text{E})$$

$$\text{Note that } X = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix} \text{ and } U = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix}$$

$$J = \sum_{p=1}^P J_p$$

$$\begin{aligned} \nabla_b J &= \sum_{p=1}^P \nabla_b J_p \\ &= \sum_{p=1}^P \nabla_{u_p} J_p \\ &= \sum_{p=1}^P \nabla_{u_p} J \\ &= \nabla_{u_1} J + \nabla_{u_2} J + \dots + \nabla_{u_P} J \quad \text{Substituting from (D)} \\ &= (\nabla_{u_1} J \ \nabla_{u_2} J \ \dots \ \nabla_{u_P} J) \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad \text{Since } \nabla_{u_p} J = \nabla_{u_p} J_p. \\ &= (\nabla_u J)^T \mathbf{1}_P \end{aligned} \quad (\text{F})$$

where $\mathbf{1}_P = (1, 1, \dots, 1)^T$ is a vector of P ones.

From (E) and (F):

$$\begin{aligned} \nabla_W J &= X^T \nabla_u J \\ \nabla_b J &= (\nabla_u J)^T \mathbf{1}_P \end{aligned}$$

That is, by computing gradient $\nabla_u J$ with respect to synaptic input, the weights and biases can be updated.

$$\begin{aligned} \mathbf{W} &= \mathbf{W} - \alpha X^T \nabla_u J \\ \mathbf{b} &= \mathbf{b} - \alpha (\nabla_u J)^T \mathbf{1}_P \end{aligned}$$

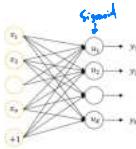
Learning a single layer

Learning a layer of neurons	
SGD	$\mathbf{W} = \mathbf{W} - \alpha x (\nabla_u J)^T$ $\mathbf{b} = \mathbf{b} - \alpha \nabla_u J$
GD	$\mathbf{W} = \mathbf{W} - \alpha X^T \nabla_u J$ $\mathbf{b} = \mathbf{b} - \alpha (\nabla_u J)^T \mathbf{1}_P$

To learn a given layer, we need to compute $\nabla_u J$ for SGD and $\nabla_u J$ for GD.

Those gradients with respect to synaptic inputs are dependent on the types of neurons in the layer.

Perceptron layer



$$y_k = f(u_k) = \frac{1}{1+e^{-u_k}}$$

Given a training pattern (\mathbf{x}, \mathbf{d})
Note $\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n$ and $\mathbf{d} = (d_1, d_2, \dots, d_K)^T \in \mathbb{R}^K$.

$$\text{The square-error cost function: } J = \frac{1}{2} \sum_{k=1}^K (d_k - y_k)^2$$

$$\text{where } y_k = f(u_k) = \frac{1}{1+e^{-u_k}} \text{ and } u_k = \mathbf{x}^T \mathbf{w}_k + b_k.$$

A layer of perceptrons performs **multidimensional non-linear regression** and learns a multidimensional non-linear mapping:

$$\phi: \mathbb{R}^n \rightarrow \mathbb{R}^K$$

Learning a perceptron layer

GD	SGD
(X, D)	(\mathbf{x}, \mathbf{d})
$U = XW + B$	$u = W^T x + b$
$Y = f(U)$	$y = f(u)$
$\nabla_U J = -(D - Y) \cdot f'(U)$	$\nabla_u J = -(d - y) \cdot f'(u)$
$W = W - \alpha X^T \nabla_U J$	$W = W - \alpha x(\nabla_u J)^T$
$b = b - \alpha (\nabla_u J)^T \mathbf{1}_p$	$b = b - \alpha \nabla_u J$

SGD for perceptron layer

Given a training pattern (\mathbf{x}, \mathbf{d})
Note $\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n$ and $\mathbf{d} = (d_1, d_2, \dots, d_K)^T \in \mathbb{R}^K$.

$$\text{The square-error cost function: } J = \frac{1}{2} \sum_{k=1}^K (d_k - y_k)^2$$

$$\text{where } y_k = f(u_k) = \frac{1}{1+e^{-u_k}} \text{ and } u_k = \mathbf{x}^T \mathbf{w}_k + b_k.$$

Gradient of J with respect to u_k :

$$\frac{\partial J}{\partial u_k} = \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial u_k} = -(d_k - y_k) \frac{\partial y_k}{\partial u_k} = -(d_k - y_k) f'(u_k) \quad (G)$$

$$\text{Substituting } \nabla_u J = \frac{\partial J}{\partial u_k} = -(d_k - y_k) f'(u_k) \text{ from (G):}$$

$$\nabla_u J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = - \begin{pmatrix} (d_1 - y_1) f'(u_1) \\ (d_2 - y_2) f'(u_2) \\ \vdots \\ (d_K - y_K) f'(u_K) \end{pmatrix} = -(d - y) \cdot f'(u) \quad (H)$$

and $\cdot \cdot^T$ denotes element-wise multiplication.

$$\nabla_u J = -(d - y) \cdot f'(u)$$

Given a training dataset $\{(\mathbf{x}, \mathbf{d})\}$

Set learning parameter α

Initialize W and b

Repeat until convergence:

For every pattern (\mathbf{x}, \mathbf{d}) :

$$u = W^T x + b$$

$$y = f(u) = \frac{1}{1+e^{-u}}$$

$$\nabla_u J = -(d - y) \cdot f'(u)$$

$$W = W - \alpha x(\nabla_u J)^T$$

$$b = b - \alpha \nabla_u J$$

GD for perceptron layer

Given a training dataset $\{(\mathbf{x}_p, \mathbf{d}_p)\}_{p=1}^P$

Note $\mathbf{x}_p = (x_{p1}, x_{p2}, \dots, x_{pn})^T \in \mathbb{R}^n$ and $\mathbf{d}_p = (d_{p1}, d_{p2}, \dots, d_{pK})^T \in \mathbb{R}^K$.

The cost function J is given by the sum of square errors (s.s.e.):

$$J = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2$$

$$J = \sum_{p=1}^P J_p$$

where $J_p = \frac{1}{2} \sum_{k=1}^K (d_{pk} - y_{pk})^2$ is the square error for the p th pattern.

$$\mathbf{U} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_p^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix} \rightarrow \nabla_{\mathbf{U}} J = \begin{pmatrix} (\nabla_{u_1} J)^T \\ (\nabla_{u_2} J)^T \\ \vdots \\ (\nabla_{u_p} J)^T \\ \vdots \\ (\nabla_{u_P} J)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{u_1} J_1)^T \\ (\nabla_{u_2} J_2)^T \\ \vdots \\ (\nabla_{u_p} J_p)^T \\ \vdots \\ (\nabla_{u_P} J_P)^T \end{pmatrix}$$

From (H), substitute $\nabla_u J = -(d - y) \cdot f'(u)$:

$$\nabla_{\mathbf{U}} J = - \begin{pmatrix} ((d_1 - y_1) \cdot f'(u_1))^T \\ ((d_2 - y_2) \cdot f'(u_2))^T \\ \vdots \\ ((d_p - y_p) \cdot f'(u_p))^T \\ \vdots \\ ((d_P - y_P) \cdot f'(u_P))^T \end{pmatrix} = - \begin{pmatrix} (d_1^T - y_1^T) \cdot f'(u_1^T) \\ (d_2^T - y_2^T) \cdot f'(u_2^T) \\ \vdots \\ (d_p^T - y_p^T) \cdot f'(u_p^T) \\ \vdots \\ (d_P^T - y_P^T) \cdot f'(u_P^T) \end{pmatrix}$$

$$\nabla_{\mathbf{U}} J = -(D - Y) \cdot f'(U)$$

$$\text{where } \mathbf{D} = \begin{pmatrix} d_1^T \\ d_2^T \\ \vdots \\ d_p^T \\ \vdots \\ d_P^T \end{pmatrix}, \mathbf{Y} = \begin{pmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_p^T \\ \vdots \\ y_P^T \end{pmatrix}, \text{ and } \mathbf{U} = \begin{pmatrix} u_1^T \\ u_2^T \\ \vdots \\ u_p^T \\ \vdots \\ u_P^T \end{pmatrix} \quad (\text{A} \otimes \text{T} \Rightarrow \text{A}^T \otimes \text{T})$$

Given a training dataset (\mathbf{X}, \mathbf{D})

Set learning parameter α

Initialize W and b

Repeat until convergence:

$$U = XW + B$$

$$Y = f(U) = \frac{1}{1+e^{-U}}$$

$$\nabla_U J = -(D - Y) \cdot f'(U)$$

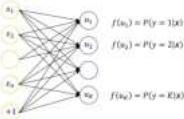
$$W = W - \alpha X^T \nabla_U J$$

$$b = b - \alpha (\nabla_U J)^T \mathbf{1}_p$$

$\text{Loc element wise product}$

Softmax layer

Softmax layer is the extension of logistic regression to **multiclass classification** problem, which is also known as *multinomial logistic regression*.



SGD for softmax layer

Given a training pattern (\mathbf{x}, d) where $\mathbf{x} \in R^n$ and $d \in \{1, 2, \dots, K\}$.

The cost function for learning is by the *multiclass cross-entropy*:

$$J = -\sum_{k=1}^K 1(d=k) \log(f(u_d))$$

where u_k is the synaptic input to the k neuron.

The cost function can also be written as

$$J = -\log(f(u_d))$$

where d is the target label of input \mathbf{x} .

Note that the logarithm here is natural: $\log = \log_e$

$$J = -\log(f(u_d))$$

The gradient with respect to u_k is given by

$$\frac{\partial J}{\partial u_k} = -\frac{1}{f(u_d)} \frac{\partial f(u_d)}{\partial u_k} \quad (I)$$

where

$$\frac{\partial f(u_d)}{\partial u_k} = \frac{\partial}{\partial u_k} \left(\frac{e^{u_d}}{\sum_{k'=1}^K e^{u_{k'}}} \right)$$

The above differentiation need to be considered separately for $k = d$ and for $k \neq d$.

If $k = d$:

$$\begin{aligned} \frac{\partial f(u_d)}{\partial u_k} &= \frac{\partial}{\partial u_k} \left(\frac{e^{u_d}}{\sum_{k'=1}^K e^{u_{k'}}} \right) \\ &= \frac{e^{u_d} e^{u_d} - e^{u_d} e^{u_d}}{\left(\sum_{k'=1}^K e^{u_{k'}} \right)^2} = \frac{\partial \left(\sum_{k'=1}^K e^{u_{k'}} \right)}{\partial u_k} = e^{u_d} \end{aligned}$$

If $k \neq d$:

$$\begin{aligned} \frac{\partial f(u_d)}{\partial u_k} &= \frac{\partial}{\partial u_k} \left(\frac{e^{u_d}}{\sum_{k'=1}^K e^{u_{k'}}} \right) \\ &= -\frac{e^{u_d} e^{u_k}}{\left(\sum_{k'=1}^K e^{u_{k'}} \right)^2} \\ &= -f(u_d) f(u_k) \quad 1(k=d) = 0 \\ &= f(u_d) (1(k=d) - f(u_k)) \\ &\quad \frac{\partial f(u_d)}{\partial u_k} = f(u_d) (1(k=d) - f(u_k)) \end{aligned}$$

Substituting in (I):

$$\nabla_u J = \frac{\partial J}{\partial u_k} = -\frac{1}{f(u_d)} \frac{\partial f(u_d)}{\partial u_k} = -(1(d=k) - f(u_k))$$

Gradient J with respect to \mathbf{u} :

$$\nabla_u J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = -\begin{pmatrix} 1(d=1) - f(u_1) \\ 1(d=2) - f(u_2) \\ \vdots \\ 1(d=K) - f(u_K) \end{pmatrix} = -(1(\mathbf{k}=d) - f(\mathbf{u})) \quad (J)$$

where $\mathbf{k} = (1 \ 2 \ \dots \ K)^T$

For a softmax layer:

$$\nabla_u J = -(1(\mathbf{k}=d) - f(\mathbf{u}))$$

where:

$$1(\mathbf{k}=d) = \begin{pmatrix} 1(d=1) \\ 1(d=2) \\ \vdots \\ 1(d=K) \end{pmatrix} \text{ and } f(\mathbf{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_K) \end{pmatrix}$$

Note that $1(\mathbf{k}=d)$ is a one-hot vector where the element corresponding to the target label d is '1' and elsewhere is '0'.

GD for softmax layer

Given a set of patterns $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$ where $\mathbf{x}_p \in R^n$ and $d_p \in \{1, 2, \dots, K\}$.

The cost function of the softmax layer is given by the *multiclass cross-entropy*:

$$J = -\sum_{p=1}^P \left(\sum_{k=1}^K 1(d_p=k) \log(f(u_{dp})) \right)$$

where u_{pk} is the synaptic input to the k neuron for input \mathbf{x}_p .

The cost function J can also be written as

$$J = -\sum_{p=1}^P \log(f(u_{pd_p}))$$

J can be written as the sum of cost due to individual patterns:

$$J = \sum_{p=1}^P J_p$$

where $J_p = -\log(f(u_{pd_p}))$ is the cross-entropy for the p th pattern.

$$\nabla_u J = \begin{pmatrix} (\nabla_{u_1} J_p)^T \\ (\nabla_{u_2} J_p)^T \\ \vdots \\ (\nabla_{u_K} J_p)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{u_1} J_1)^T \\ (\nabla_{u_2} J_2)^T \\ \vdots \\ (\nabla_{u_K} J_P)^T \end{pmatrix}$$

Substituting $\nabla_u J = -(1(\mathbf{k}=d) - f(\mathbf{u}))$ from (J):

$$\begin{aligned} \nabla_u J &= -\begin{pmatrix} (1(k=d_1) - f(u_1))^T \\ (1(k=d_2) - f(u_2))^T \\ \vdots \\ (1(k=d_p) - f(u_p))^T \end{pmatrix} \\ \nabla_u J &= -(K - f(\mathbf{u})) \end{aligned}$$

where $K = \begin{pmatrix} 1(k=d_1)^T \\ 1(k=d_2)^T \\ \vdots \\ 1(k=d_p)^T \end{pmatrix}$ is a matrix with every row is a one-hot vector.

Learning a softmax layer

GD	SGD
(X, D)	(x, d)
$U = XW + B$	$u = W^T x + b$
$f(U) = \frac{e^U}{\sum_{k'=1}^K e^{U_{k'}}}$	$f(u) = \frac{e^u}{\sum_{k'=1}^K e^{u_{k'}}}$
$y = \arg\max_k f(U)$	$y = \arg\max_k f(u)$
$\nabla_u J = -(K - f(\mathbf{u}))$	$\nabla_u J = -(1(k=d) - f(u))$
$W = W - \alpha x^T \nabla_u J$	$W = W - \alpha x (\nabla_u J)^T$
$B = B - \alpha (\nabla_u J)^T \mathbf{1}_P$	$b = b - \alpha \nabla_u J$

Example 2: GD of a softmax layer

Train a softmax regression layer of neurons to perform the following classification:

$$\begin{aligned} (0.94, 0.18) &\rightarrow \text{class A} \\ (-0.58, -0.53) &\rightarrow \text{class B} \\ (-0.23, -0.31) &\rightarrow \text{class B} \\ (0.42, -0.44) &\rightarrow \text{class A} \\ (0.5, -1.66) &\rightarrow \text{class C} \\ (-1.0, -0.51) &\rightarrow \text{class B} \\ (0.78, -0.65) &\rightarrow \text{class A} \\ (0.04, -0.20) &\rightarrow \text{class C} \end{aligned}$$

Use a learning factor $\alpha = 0.05$.

Let $y = \begin{cases} 1, \text{for class A} \\ 2, \text{for class B} \\ 3, \text{for class C} \end{cases}$

$$X = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.20 \end{pmatrix}, d = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \\ 1 \\ 3 \end{pmatrix}$$

$$K = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Initialize $W = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix}, b = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$

$$\text{Then, } B = \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

1st epoch starts ...

$$U = XW + B = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.20 \end{pmatrix} \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

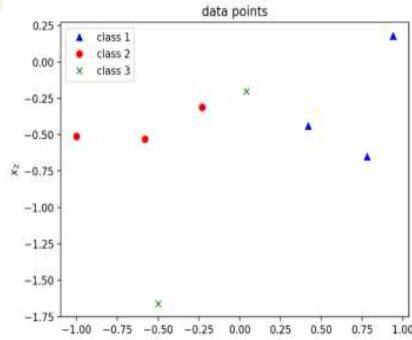
$$U = \begin{pmatrix} 0.86 & 0.11 & 0.64 \\ -0.84 & -0.28 & -0.49 \\ -0.41 & -0.16 & -0.22 \\ -0.01 & -0.21 & 0.17 \\ -0.86 & -0.82 & -0.06 \\ -1.15 & -0.27 & -0.75 \\ 0.11 & -0.31 & 0.35 \\ -0.12 & -0.10 & -0.02 \end{pmatrix} \quad f(u_{12}) = \frac{e^{0.11}}{e^{0.86} + e^{0.11} + e^{0.64}}$$

$$f(U) = \frac{e^{(U)}}{\sum_{k=1}^3 e^{(U)}} = \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix}$$

$$y = \underset{k}{\operatorname{argmax}}[f(U)] = \underset{k}{\operatorname{argmax}} \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 3 \\ 2 \\ 3 \\ 1 \end{pmatrix}$$

Errors = $\sum_{p=1}^8 1(d_p \neq y_p) = 2$

$$\begin{aligned} \text{Entropy, } J &= -\sum_{p=1}^8 \log(f(u_{pd_p})) \\ &= -\log(0.44) - \log(0.42) - \dots - \log(0.35) \\ &= 7.26 \end{aligned}$$



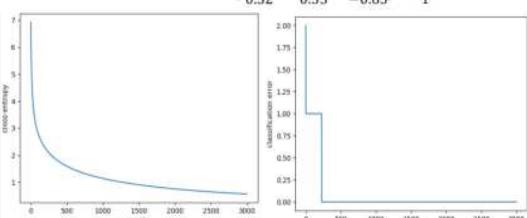
$$\nabla_U f = -(\mathbf{K} - f(\mathbf{U}))$$

$$\begin{aligned} &= - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix} \\ &= \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix} \end{aligned}$$

$$W = W - \alpha X^T \nabla_U f$$

$$\begin{aligned} &= \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.20 \end{pmatrix} - 0.05 \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix} \\ &= \begin{pmatrix} 0.85 & -0.06 & 0.63 \\ 0.76 & 0.50 & 0.22 \end{pmatrix} \end{aligned}$$

$$b = b - \alpha (\nabla_U f)^T \mathbf{1}_P = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix} - 0.05 \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.03 \\ 0.02 \\ -0.05 \end{pmatrix}$$



At convergence at 3000 iterations:

$$W = \begin{pmatrix} 14.22 & -13.04 & 0.00 \\ 4.47 & -2.05 & -0.95 \end{pmatrix}$$

$$b = \begin{pmatrix} -0.53 \\ -0.47 \\ 1.00 \end{pmatrix}$$

Entropy = 0.562

Errors = 0

Initialization of weights

Random initialization is inefficient

At initialization, it is desirable that **weights** are small and near zero

- to operate in the linear region of the activation function
- to preserve the variance of activations and gradients.

Two methods:

- Using a uniform distribution within specified limits
- Using a truncated normal distribution

Initialization from a uniform distribution (Xavier/Glorot uniform Initialization)

Uniformly draws weight samples $w \sim U(-a, +a)$:

where

$$a = \text{gain} \times \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}$$

n_{in} is the number of input nodes and n_{out} is the number of neurons in the layer.

activation	tanh	sigmoid	Tanh	ReLU	Leaky ReLU
gain	1	1	5/3	$\sqrt{2}$	$\sqrt{1 + \text{slope}^2}$

Initialization from a truncated normal distribution

$$w \sim \text{truncated_normal} \left[\text{mean} = 0, \text{std} = \frac{\text{gain}}{\sqrt{n_{\text{in}}}} \right]$$

In the truncated normal, the samples that are two s.d. away from the center are discarded and resampled again.

This is also known as **Kaiming normal initialization**.

Iris dataset

Iris dataset:
<https://archive.ics.uci.edu/ml/datasets/Iris>

Three classes of iris flower:



Four features:

Sepal length, sepal width, petal length, petal width

150 data points, 50 for each class

Features:

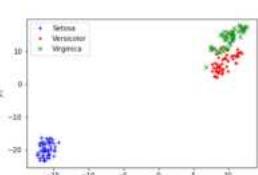
[-1.4333333e-01 4.4600000e-01 -2.3586667e+00 -9.9866667e-01]
[-0.9433333e-01 -5.4000000e-02 -2.3586667e+00 -9.9866667e-01]
[-1.1433333e+00 1.4600000e-01 -2.4586667e+00 -9.9866667e-01]

Labels:

[0 0 0 0 0 0 0 0 0 ... 1 1 1 1 1 1 1 1 ... 2 2 2 2 2 2 2 2]

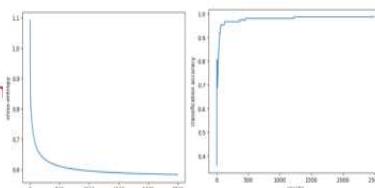
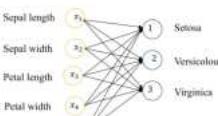
Iris data

Display of data points after dimensionality reduction (from Four dimensions to Two) by t-SNE.



Example 3: Softmax classification of iris data

Example 4: GD of a perceptron layer



Revision: neurons and layers

Classification	Logistic neurons
Two-class	Logistic regression neuron
Multiclass	Softmax layer

Regression	Linear	Non-linear
One dimensional	Linear neuron	Perceptron
Multi-dimensional	Linear neuron layer	Perceptron layer

Summary: GD for layers

$$\begin{aligned} & (x, d) \\ & u = Wx + b \\ & W = W - \alpha A^T (V_{ul}) \\ & b = b - \alpha (V_{ul})^T u \end{aligned}$$

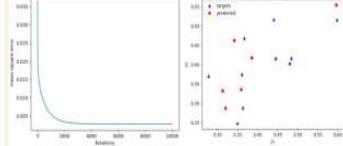
layer	$f(u) \cdot Y$	V_{ul}
Linear neuron layer	$y = f(u) = u$	$-(D - Y)$
Perceptron layer	$y = f(u) = \frac{1}{1 + e^{-u}}$	$-(D - Y) \cdot f'(u)$
Softmax layer	$f(u) = \frac{e^u}{\sum_{k=1}^K e^{u_k}}$ $y = \arg\max f(u)$	$-(K - f(u))$

Summary: SGD for layers

$$\begin{aligned} & (x, d) \\ & u = Wx + b \\ & W = W - \alpha (V_{ul})^T \\ & b = b - \alpha (V_{ul}) \end{aligned}$$

layer	$f(u) \cdot Y$	V_{ul}
Linear neuron layer	$y = f(u) = u$	$-(D - Y)$
Perceptron layer	$y = f(u) = \frac{1}{1 + e^{-u}}$	$-(D - Y) \cdot f'(u)$
Softmax layer	$f(u) = \frac{e^u}{\sum_{k=1}^K e^{u_k}}$ $y = \arg\max f(u)$	$-(1(k=d) - f(u))$

Epoch	Y	mae	W	b
2	(0.50 0.51) (0.50 0.51) (0.50 0.50) (0.50 0.51) (0.50 0.50) (0.50 0.50)	0.036 (-0.01 0.02) (0.00 0.01)	(0.02 0.03) (-0.04)	(-0.04)
30000	(0.34 0.46) (0.39 0.42) (0.32 0.29) (0.48 0.40) (0.36 0.34) (0.45 0.42) (0.59 0.57) (0.31 0.33)	0.003 (1.00 1.18) (1.42 0.60) (1.14 0.04)	(-2.24) (-1.57)	

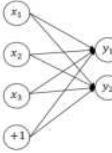


Design a perceptron layer to perform the following mapping using GD learning:

$x = (x_1, x_2, x_3)^T$	$d = (d_1, d_2)^T$
(0.77, 0.02, 0.63)	(0.37, 0.47)
(0.75, 0.50, 0.22)	(0.36, 0.38)
(0.20, 0.76, 0.17)	(0.35, 0.25)
(0.09, 0.69, 0.95)	(0.48, 0.42)
(0.00, 0.51, 0.81)	(0.36, 0.29)
(0.61, 0.72, 0.29)	(0.44, 0.52)
(0.92, 0.71, 0.54)	(0.60, 0.52)
(0.14, 0.37, 0.67)	(0.28, 0.37)

Use $\alpha = 0.1$.

$$X = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix} \quad \text{and } D = \begin{pmatrix} 0.37 & 0.47 \\ 0.36 & 0.38 \\ 0.35 & 0.25 \\ 0.48 & 0.42 \\ 0.36 & 0.29 \\ 0.44 & 0.52 \\ 0.60 & 0.52 \\ 0.28 & 0.37 \end{pmatrix}$$



Output $y_1, y_2 \in [0, 1]$

So, activation function for both neurons:

$$f(u) = \frac{1}{1 + e^{-u}}$$

$$f'(u) = \frac{e^{-u}}{(1 + e^{-u})^2}$$

Learning factor $\alpha = 0.1$.

Weights and biases are initialized:

$$W = \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} \text{ and } b = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

$$\begin{aligned} \text{Epoch 1:} \\ U &= XW + B \\ &= \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix} \\ &= \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 0.03 & 0.06 \\ 0.03 & 0.06 \\ 0.02 & 0.05 \\ 0.03 & 0.07 \\ 0.02 & 0.05 \\ 0.03 & 0.07 \\ 0.04 & 0.09 \\ 0.02 & 0.05 \end{pmatrix} \\ Y &= f(U) = \frac{1}{1 + e^{-U}} = \begin{pmatrix} 0.51 & 0.51 \\ 0.50 & 0.51 \\ 0.50 & 0.51 \\ 0.51 & 0.51 \\ 0.51 & 0.51 \\ 0.51 & 0.52 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} f'(U) &= Y \cdot (1 - Y) = \begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix} \\ \nabla_U f &= -(D - Y) \cdot f'(U) = \begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix} = \begin{pmatrix} 0.04 & 0.03 \\ 0.04 & 0.03 \\ 0.04 & 0.03 \\ 0.04 & 0.03 \\ 0.04 & 0.03 \\ 0.04 & 0.03 \\ 0.04 & 0.03 \\ 0.04 & 0.03 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} W &= W - \alpha X^T \nabla_U f \\ &= \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix}^T \begin{pmatrix} 0.04 & 0.01 \\ 0.04 & 0.03 \\ 0.04 & 0.07 \\ 0.04 & 0.03 \\ 0.04 & 0.06 \\ 0.02 & 0.00 \\ 0.06 & 0.04 \end{pmatrix} = \begin{pmatrix} 0.02 & 0.04 \\ 0.01 & 0.03 \\ 0.02 & 0.03 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} b &= b - \alpha (V_{ul})^T \mathbf{1}_F = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} - 0.1 \begin{pmatrix} 0.03 & 0.01 & 0 \\ 0.03 & 0.03 & 0 \\ 0.04 & 0.04 & 0 \\ 0.00 & 0.02 & 0 \\ 0.03 & 0.05 & 0 \\ 0.01 & 0.00 & 0 \\ -0.02 & 0.00 & 0 \\ 0.05 & 0.03 & 0 \end{pmatrix} = \begin{pmatrix} -0.02 \\ 0.01 \end{pmatrix} \end{aligned}$$

4. Deep Neural Networks

Chain rule of differentiation

Let x, y , and $f \in R$ be one-dimensional variables and

$$j = g(y)$$

$$y = f(x)$$

Chain rule of differentiation states that:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \frac{\partial y}{\partial x}$$

$$\nabla_x J = \left(\frac{\partial J}{\partial x} \right) \nabla_y j$$



Note the transfer of gradient of J from y to x .

Chain rule in multidimensions



$$x = (x_1, x_2, \dots, x_n) \in R^n, y = (y_1, y_2, \dots, y_K) \in R^K, J \in R, \text{ and}$$

$$y = f(x)$$

$$J = g(y)$$

Then, the chain rule of differentiation states that:

$$\nabla_x J = \left(\frac{\partial J}{\partial x} \right)^T \nabla_y J$$

The matrix $\frac{\partial y}{\partial x}$ is known as the **Jacobian** of the function f where $y = f(x)$.

$$\nabla_x J = \left(\frac{\partial f}{\partial x} \right)^T \nabla_y J$$

where

$$\nabla_x J = \frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \quad \text{and} \quad \nabla_y J = \frac{\partial f}{\partial y} = \begin{pmatrix} \frac{\partial f}{\partial y_1} \\ \frac{\partial f}{\partial y_2} \\ \vdots \\ \frac{\partial f}{\partial y_K} \end{pmatrix}.$$

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \frac{\partial y_K}{\partial x_2} & \dots & \frac{\partial y_K}{\partial x_n} \end{pmatrix}$$

Note that differentiation of a scalar by a vector results in a vector and differentiation of a vector by a vector results in a matrix.

Example 1: find Jacobian of a function

Let $x = (x_1, x_2, x_3)$, $y = (y_1, y_2)$, and $y = f(x)$ where f is given by

$$y_1 = 5 - 2x_1 + 3x_3$$

$$y_2 = x_1 + 5x_2^2 + x_3^3 - 1$$

Find the Jacobian of f .

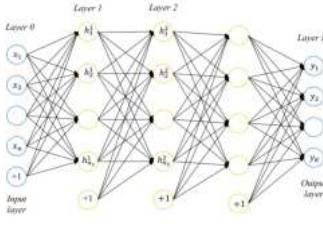
$$\begin{aligned} \frac{\partial y_1}{\partial x_1} &= -2, & \frac{\partial y_1}{\partial x_2} &= 0, & \frac{\partial y_1}{\partial x_3} &= 3 \\ \frac{\partial y_2}{\partial x_1} &= 1, & \frac{\partial y_2}{\partial x_2} &= 10x_2, & \frac{\partial y_2}{\partial x_3} &= 3x_3^2 \end{aligned}$$

Jacobian:

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{pmatrix} = \begin{pmatrix} -2 & 0 & 3 \\ 1 & 10x_2 & 3x_3^2 \end{pmatrix}$$

Deep neural networks (DNN):

Also known as feedforward networks (FFN)



L layer DNN

Feedforward Networks (FFN)

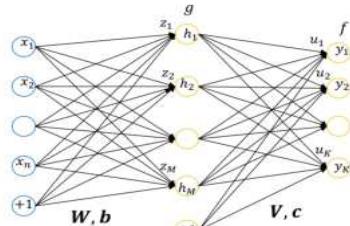
Feedforward networks (FFN) consists of several layers of neurons where activations propagate from input layer to output layer. The layers between the input and output layers are referred to as **hidden layers**.

The **number of layers** is referred to as the **depth** of the feedforward network. When a network has many hidden layers of neurons, feedforward networks are referred to as **deep neural networks (DNN)**. Learning in deep neural networks is referred to as **deep learning**. The **number of neurons** in a layer is referred to as the **width** of that layer.

The hidden layers are usually composed of perceptrons (sigmoidal units) or ReLU units and the output layer is usually

- A linear neuron layer for regression
- A softmax layer for classification

Two-layer FFN



Input $x = (x_1 \ x_2 \ \dots \ x_n)^T$

Hidden-layer output $\mathbf{h} = (h_1 \ h_2 \ \dots \ h_M)^T$

Output $\mathbf{y} = (y_1 \ y_2 \ \dots \ y_K)^T$

W, b – weight and bias of the hidden layer

V, c – weight and bias of the output layer

M is the number of hidden layer neurons

Forward propagation of activations single input pattern

Consider an input pattern (x, d) to 2-layer FFN:



Synaptic input \mathbf{z} to hidden layer:

$$\mathbf{z} = W^T \mathbf{x} + \mathbf{b}$$

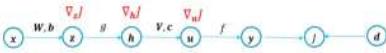
Output \mathbf{h} of hidden layer:

$$\mathbf{h} = g(\mathbf{z})$$

Output \mathbf{y} of output layer:

$$\mathbf{y} = f(\mathbf{h})$$

Backpropagation of gradients



Since the targets appear at the output, the error gradient at the output layer is $\nabla_d J$ is known. Therefore, output weights and bias, V, c , can be learnt.

To learn hidden layer weights and biases, the gradients at the output layer are to be backpropagated to hidden layers.

Derivatives

$$u = V^T h + c$$

Consider synaptic input to u_k to the k th neuron at the output layer. Let weight vector $v_k = (v_{k1} \ v_{k2} \ \dots \ v_{kM})^T$ and bias c_k .

The synaptic input u_k to h is given by

$$u_k = v_k^T h + c_k = v_{k1}h_1 + v_{k2}h_2 + \dots + v_{kM}h_M + c_k$$

Therefore:

$$\frac{\partial u}{\partial h} = \begin{pmatrix} \frac{\partial u_k}{\partial h_1} & \frac{\partial u_k}{\partial h_2} & \dots & \frac{\partial u_k}{\partial h_M} \\ \frac{\partial u_k}{\partial h_1} & \frac{\partial u_k}{\partial h_2} & \dots & \frac{\partial u_k}{\partial h_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial u_k}{\partial h_1} & \frac{\partial u_k}{\partial h_2} & \dots & \frac{\partial u_k}{\partial h_M} \end{pmatrix} = \begin{pmatrix} v_{11} & v_{12} & \dots & v_{1M} \\ v_{21} & v_{22} & \dots & v_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ v_{N1} & v_{N2} & \dots & v_{NM} \end{pmatrix} = V^T$$

That is,

$$\frac{\partial u}{\partial h} = V^T$$

$$y = f(V^T h)$$

Considering k th neuron:

$$y_k = f(u_k)$$

$$\frac{\partial y}{\partial u} = \begin{pmatrix} \frac{\partial y_k}{\partial u_1} & \frac{\partial y_k}{\partial u_2} & \dots & \frac{\partial y_k}{\partial u_N} \\ \frac{\partial y_k}{\partial u_1} & \frac{\partial y_k}{\partial u_2} & \dots & \frac{\partial y_k}{\partial u_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_k}{\partial u_1} & \frac{\partial y_k}{\partial u_2} & \dots & \frac{\partial y_k}{\partial u_N} \end{pmatrix} = \begin{pmatrix} f'(u_1) & 0 & \dots & 0 \\ 0 & f'(u_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & f'(u_N) \end{pmatrix} = diag(f'(u))$$

That is,

$$\frac{\partial y}{\partial u} = diag(f'(u))$$

where $diag(f'(u))$ is a diagonal matrix composed of derivatives corresponding to individual components of u in the diagonal.

Back-propagation of gradients: single pattern



Considering output layer,

$$\nabla_y J = \begin{cases} -(d - y) \\ -(1(k=d) - f(u)) \end{cases} \quad \text{for a linear layer} \\ \text{for a softmax layer}$$

From chain rule of differentiation,

$$\nabla_h J = \left(\frac{\partial u}{\partial h} \right)^T \nabla_u J = V \nabla_u J \quad \text{From (A)}$$

$$\nabla_z J = \left(\frac{\partial h}{\partial z} \right)^T \nabla_h J = \text{diag}(g'(z)) V \nabla_u J = V \nabla_u J \cdot g'(z)$$

(C), from (B)

Proof

For a vector x :

$$\text{diag}(f'(u))x = \begin{pmatrix} f'(u_1) & 0 & \cdots & 0 \\ 0 & f'(u_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(u_K) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix} = \begin{pmatrix} f'(u_1)x_1 \\ f'(u_2)x_2 \\ \vdots \\ f'(u_K)x_K \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix} \cdot \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_K) \end{pmatrix}$$

That is:

$$\text{diag}(f'(u))x = x \cdot f'(u) = f'(u) \cdot x$$

Back-propagation of gradients: single input pattern

From (C);

$$\nabla_z J = V \nabla_u J \cdot g'(z)$$

That is, the gradients at output layer are multiplied by V and back-propagated to hidden layer.

Note that hidden-layer activations are multiplied by V^T (Note $u = V^T h + c$) in forward propagation and in back-propagation, the gradients are multiplied by V .

Flow of gradients propagated in backward direction, hence named **back-propagation** (backprop) algorithm.

1. Forward Propagation:

- Forward propagation refers to the process of computing the output of the neural network for a given input.
- It involves passing the input data through the network layer by layer, from the input layer to the output layer, while applying the activation functions and using the current values of the parameters (weights and biases).
- Each neuron in the network receives inputs from the previous layer, computes a weighted sum of these inputs, adds a bias term, and applies an activation function to produce its output.
- The output of the last layer (the output layer) is the predicted output of the network for the given input.
- Forward propagation does not involve adjusting the parameters of the network; it is solely concerned with computing predictions based on the current parameter values.

2. Backward Propagation (Backpropagation):

- Backward propagation refers to the process of updating the parameters of the neural network based on the error between the predicted output and the true target values.
- It involves computing the gradients of the loss function with respect to each parameter of the network using techniques such as the chain rule from calculus.
- The gradients indicate how much the loss function would change with respect to small changes in each parameter.
- Using the gradients, the parameters are updated in the opposite direction of the gradient (gradient descent) to minimize the loss function.
- The updated parameters are obtained by subtracting a fraction of the gradient from the current parameter values, scaled by a learning rate hyperparameter.
- Backward propagation is typically performed iteratively for each batch of training data in a process known as stochastic gradient descent (SGD) or its variants, updating the parameters gradually to minimize the J function over the entire training dataset.

SGD of two-layer FFN

Output layer:

$$\nabla_u J = \begin{cases} -(d - y) \\ -(1(k=d) - f(u)) \end{cases} \quad \text{for linear layer} \\ \text{for softmax layer}$$

Output layer:

$$\nabla_u J = \begin{cases} -(d - Y) \\ -(K - f(U)) \end{cases} \quad \text{for linear layer} \\ \text{for softmax layer}$$

Hidden layer:

$$\nabla_z J = V \nabla_u J \cdot g'(z)$$

Given a training dataset $\{(x, d)\}$

Set learning parameter α

Initialize W, b, V, c

Repeat until convergence:

For every pattern (x, d) :

$$z = W^T x + b$$

$$h = g(z)$$

$$u = V^T h + c$$

$$y = f(u)$$

Forward propagation

Forward propagation

$$\nabla_u J = \begin{cases} -(d - y) \\ -(1(k=d) - f(u)) \end{cases}$$

$$\nabla_z J = V \nabla_u J \cdot g'(z)$$

Backward propagation

Backward propagation

$$V \leftarrow V - \alpha H^T \nabla_u J$$

$$c \leftarrow c - \alpha (V_u J)^T 1_p$$

$$W \leftarrow W - \alpha X^T \nabla_u J$$

$$b \leftarrow b - \alpha V_u J$$

Forward propagation of activations: batch of inputs

Computational graph of 2-layer FFN for a batch of patterns (X, D) :



Synaptic input Z to hidden layer:

$$Z = XW + B$$

Output H of the hidden layer:

$$H = g(Z)$$

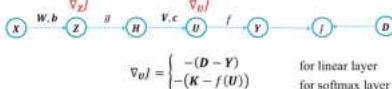
Synaptic input U to output layer:

$$U = HV + C$$

Output Y of the output layer:

$$Y = f(U)$$

Back-propagation of gradients: batch of patterns



$\nabla_u J = \begin{cases} -(D - Y) \\ -(1(k=d) - f(U)) \end{cases}$ for linear layer
for softmax layer

$$\nabla_z J = \begin{pmatrix} (\nabla_{z_1} J)^T \\ (\nabla_{z_2} J)^T \\ \vdots \\ (\nabla_{z_p} J)^T \end{pmatrix} = \begin{pmatrix} \left(V \nabla_u J \cdot g'(z_1) \right)^T \\ \left(V \nabla_u J \cdot g'(z_2) \right)^T \\ \vdots \\ \left(V \nabla_u J \cdot g'(z_p) \right)^T \end{pmatrix} \quad \text{Substituting from (C)}$$

$$\nabla_z J = \begin{pmatrix} (\nabla_{u_1} J)^T V^T \cdot (g'(z_1))^T \\ (\nabla_{u_2} J)^T V^T \cdot (g'(z_2))^T \\ \vdots \\ (\nabla_{u_p} J)^T V^T \cdot (g'(z_p))^T \end{pmatrix} \quad (XY)^T = Y^T X^T$$

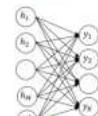
$$= \begin{pmatrix} (\nabla_{u_1} J)^T \\ (\nabla_{u_2} J)^T \\ \vdots \\ (\nabla_{u_p} J)^T \end{pmatrix} V^T \cdot \begin{pmatrix} (g'(z_1))^T \\ (g'(z_2))^T \\ \vdots \\ (g'(z_p))^T \end{pmatrix}$$

$$= (\nabla_u J) V^T \cdot g'(z)$$

$$\nabla_z J = (\nabla_u J) V^T \cdot g'(z)$$

Back-propagation

H — V — $U = HV + C$



$$\nabla_z J = (\nabla_u J) V^T \cdot g'(z) \quad \nabla_u J$$

The error gradient can be seen as propagating from the output layer to the hidden layer and so learning in feedforward networks is known as the *back-propagation* algorithm

Learning in two-layer FFN

GD	SGD
(X, D)	(x, d)
$Z = XW + B$	$z = Xw + b$
$H = g(Z)$	$h = g(x)$
$U = HV + C$	$u = V^T h + c$
$Y = f(U)$	$y = f(u)$

$\nabla_u J$	$\nabla_u J$
$\begin{cases} -(D - Y) \\ -(1(k=d) - f(U)) \end{cases}$	$\begin{cases} -(d - y) \\ -(1(k=d) - f(u)) \end{cases}$

$\nabla_z J$	$\nabla_z J$
$(\nabla_u J) V^T \cdot g'(z)$	$(\nabla_u J) V^T \cdot g'(z)$

$W \leftarrow W - \alpha X^T \nabla_u J$	$W \leftarrow W - \alpha X^T \nabla_u J$
$b \leftarrow b - \alpha V_u J$	$b \leftarrow b - \alpha V_u J$

$V \leftarrow V - \alpha H^T \nabla_u J$	$V \leftarrow V - \alpha H^T \nabla_u J$
$c \leftarrow c - \alpha (V_u J)^T 1_p$	$c \leftarrow c - \alpha (V_u J)^T 1_p$

FP → Compute output of nn for a given input
Pass the input to network to generate pred without adjusting parameters
→ Compute output of each neuron by applying activation func to weighted sums of input
→ Perform once per input to generate pred

BP → Update parameter of NN based on error between pred and true target by computing gradients and adjust params to minimize loss function

→ Compute gradients of loss func to each neuron using chain rule and update parameter with gradient descent

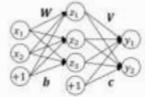
→ Perform iteratively for each batch of training data

Example 2: Two-layer FFN

Design a two-layer FFN, using gradient descent to perform the following mapping. Use a learning factor = 0.05 and three perceptrons in the hidden-layer.

Inputs $x = (x_1, x_2)$	Targets $d = (d_1, d_2)$
(0.77, 0.02)	(0.44, -0.42)
(0.63, 0.75)	(0.84, 0.43)
(0.50, 0.22)	(0.09, -0.72)
(0.20, 0.76)	(-0.25, 0.35)
(0.17, 0.09)	(-0.12, -0.13)
(0.69, 0.95)	(0.24, 0.03)
(0.00, 0.51)	(0.30, 0.20)
(0.81, 0.61)	(0.61, 0.04)

$$X = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix}$$



Output layer is a linear neuron layer
Hidden layer is a sigmoidal layer

Initialized (weights using a uniform distribution):

$$W = \begin{pmatrix} -3.97 & 1.10 & 0.42 \\ 2.79 & -2.64 & 3.13 \end{pmatrix}, b = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}, V = \begin{pmatrix} 3.58 & -1.58 \\ -3.58 & -1.75 \\ -3.38 & 2.88 \end{pmatrix}, c = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

Epoch 1:

$$Z = XW + b = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.49 & 0.15 & 0.59 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -3.00 & 0.0 & 0.39 \\ -0.42 & -1.27 & 2.61 \\ -1.35 & -0.04 & 0.91 \\ 1.34 & -1.79 & 2.46 \\ -0.45 & -0.45 & 0.35 \\ -0.05 & -1.76 & 3.27 \\ 1.42 & -1.34 & 1.60 \\ -1.51 & -0.72 & 2.25 \end{pmatrix}$$

$$H = g(Z) = \frac{1}{1+e^{-Z}} = \begin{pmatrix} 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.49 & 0.15 & 0.59 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix}$$

$$Y = HV + C = \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.49 & 0.15 & 0.59 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix} \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -4.32 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ 3.59 & 1.75 \end{pmatrix}$$

$$\nabla_H = -(D - Y) = \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix} - \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix} = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.79 & 1.15 \\ -0.54 & 0.80 \\ -2.32 & 0.34 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix}$$

$$g'(Z) = H \cdot (1 - H) = \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

$$\nabla_Z = (\nabla_H) V^T \cdot g'(Z) = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.54 & 0.80 \\ -2.32 & 0.34 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix} \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}^T \cdot \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

$$= \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.54 & 0.80 \\ -2.32 & 0.34 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix} \begin{pmatrix} 3.58 & -1.58 \\ -3.58 & -1.75 \\ -3.38 & 2.88 \end{pmatrix}^T \cdot \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

- Escape from a few layers
- ReLU nonlinearity for solving gradient vanishing (better)
- Data augmentation
- Dropout
- Outperformed all previous models on ILSVRC by 10%

Output layer:

$$\nabla_V J = H^T \nabla_U J = \begin{pmatrix} -6.23 & 3.22 \\ -8.81 & 2.85 \\ -17.07 & 7.40 \end{pmatrix}$$

$$\nabla_U J = (\nabla_U J)^T \mathbf{1}_P = \begin{pmatrix} -21.83 \\ 8.81 \end{pmatrix}$$

Hidden layer:

$$\nabla_W J = X^T \nabla_Z J = \begin{pmatrix} -8.43 & 7.81 & 7.79 \\ -8.21 & 4.56 & 3.76 \end{pmatrix}$$

$$\nabla_b J = (\nabla_Z J)^T \mathbf{1}_P = \begin{pmatrix} -15.22 \\ 13.11 \\ 13.92 \end{pmatrix}$$

Layer	Weights
L1 (Conv)	
L2 (Conv)	
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels

96 * (11 * 11 * 3 + 1)

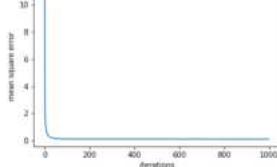
$$V \leftarrow V - \alpha \nabla_V J = \begin{pmatrix} 3.89 & -1.74 \\ -3.14 & -1.89 \\ -2.53 & 2.51 \end{pmatrix}$$

$$c \leftarrow c - \alpha \nabla_c J = \begin{pmatrix} 1.09 \\ -0.44 \end{pmatrix}$$

$$W \leftarrow W - \alpha \nabla_W J = \begin{pmatrix} -3.55 & 0.72 & 0.03 \\ 3.21 & -2.87 & 2.94 \end{pmatrix}$$

$$b \leftarrow b - \alpha \nabla_b J = \begin{pmatrix} -0.66 \\ -0.70 \end{pmatrix}$$

GD learning



After 1,000 epochs,

m. s. e. = 0.107

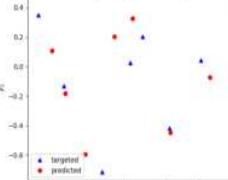
Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels

Number of parameters = (11x11x3 + 1)*96 = 34,944

Note: There are no parameters associated with a pooling layer. The pool size, stride, and padding are hyperparameters.

targets and predicted outputs



After 20,000 iterations

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256

Size (width) of the output image of the previous Conv Layer = 6

Number of parameters = (6x6*256*4096)+4096 = 37,752,832

Number of neurons = 1000

Number of neurons in the previous FC Layer = 4096

Number of parameters = (1000*4096)+1000 = 4,097,000

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

The last FC layer:
Number of neurons = 1000
Number of neurons in the previous FC Layer = 4096

Number of parameters = (1000*4096)+1000 = 4,097,000

Preprocessing of inputs

If inputs have similar variations, better approximation of inputs or prediction of outputs is achieved. Mainly, there are two approaches to normalization of inputs.

Suppose the input $x_i \in [x_{i,\min}, x_{i,\max}]$ and has a mean μ_i and a standard deviation σ_i .

If \tilde{x}_i denotes the normalized input,

$$1. \text{ Scaling} \quad \text{the inputs such that } \tilde{x}_i \in [0, 1]: \\ \tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

$$2. \text{ Normalizing} \quad \text{the input to have standard normal distributions } \tilde{x}_i \sim N(0, 1): \\ \tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

Linear activation function:

The convergence is usually improved if each output is **normalized** to have zero mean and unit standard deviation: $\tilde{y}_k \sim N(0, 1)$

$$\tilde{y}_k = \frac{y_k - \mu_k}{\sigma_k}$$

Sigmoid activation function:

Since sigmoidal activation range from 0 to 1.0, you can scale $\tilde{y}_k \in [0, 1]$:

$$\tilde{y}_k = \frac{y_k - y_{k,\min}}{y_{k,\max} - y_{k,\min}} = \frac{1}{y_{k,\max} - y_{k,\min}} y_k - \frac{y_{k,\min}}{y_{k,\max} - y_{k,\min}}$$

California housing dataset

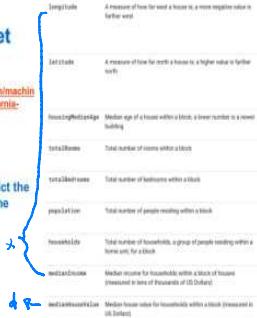
<https://developers.google.com/machine-learning/crash-course/california-housing-data-description>

9 variables

The problem is to predict the housing prices using the other 8 variables.

20540 samples

Train: 14448 samples
Test: 6192 samples



Example 3: Two-layer FFN predicting housing prices in California

Thirteen input variables, One output variable

We use FFN with one hidden layer with 10 neurons

Network size: [8, 10, 1]

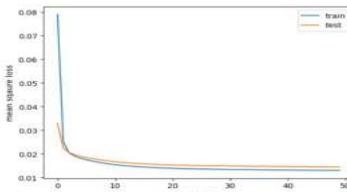
Feature 1: $\# \text{bedrooms}$

class FFN(nn.Module):

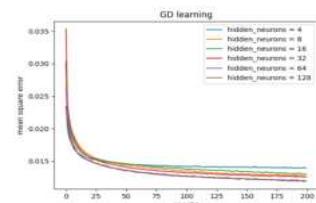
```
def __init__(self, no_features, no_hidden, no_labels):
    super().__init__()
    self.relu_stack = nn.Sequential(
        nn.Linear(no_features, no_hidden),
        nn.ReLU(),
        nn.Linear(no_hidden, no_labels),
```

```
def forward(self, x):
    logits = self.relu_stack(x)
    return logits
```

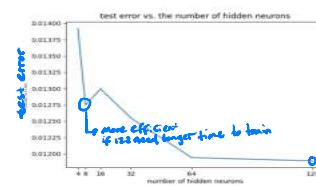
Example 3a



Example 3b: no of hidden neurons



Example 3b: width of the hidden layer



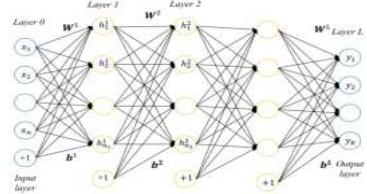
Optimum number of hidden neurons: 64

Width of hidden Layers

The number of parameters of the network increases with the **width** of layers. Therefore, the network attempts to remember the training patterns with increasing number of parameters. In other words, the network aims at minimizing the training error at the expense of its generalization ability on unseen data.

As the number of hidden units increases, the test error decreases initially but tends to increase at some point. The optimal number of hidden units is often determined empirically (that is, by trial and error).

Deep neural networks (DNN)



DNN notations

Input layer $l = 0$

Width: n_0

Input x, X

Hidden layers $l = 1, 2, \dots, L-1$

Width: n_l

Weight matrix W^l , bias vector b^l

Synaptic input u^l, U^l

Activation function f^l

Output y, Y

Desired output d, D

Forward propagation of activation in DNN: single pattern

Input (x, d)

$u^l = W^{l,T} x + b^l$

For layers $l = 1, 2, \dots, L-1$:

$h^l = f^l(u^l)$

$u^{l+1} = W^{l+1,T} h^l + b^{l+1}$

$y = f^L(u^L)$

Back-propagation of gradients in DNN: single pattern

If $l = L$:

$$\nabla_w J = \begin{cases} -(D - y) \\ -(1(k=d) - f^L(u^L)) \end{cases}$$

for linear layer

for softmax layer

else:

$$\nabla_w J = W^{l+1} (\nabla_u J) \cdot f'^T (u^l)$$

from (C)

$$\nabla_w J = H^{l-1} (\nabla_u J)^T$$

$$\nabla_b J = \nabla_u J$$

Gradients are backpropagated from the output layer to the input layer

Forward propagation of activation in DNN: batch of patterns

Input (X, D)

$U^l = X W^l + B^l$

For layers $l = 1, 2, \dots, L-1$:

$H^l = f^l(U^l)$

$U^{l+1} = H^l W^{l+1} + B^{l+1}$

$Y = f^L(U^L)$

Back-propagation of gradients in DNN: batch of patterns

If $l = L$:

$$\nabla_w J = \begin{cases} -(D - Y) \\ -(K - f^L(U^L)) \end{cases}$$

Else:

$$\nabla_w J = (V_w J) W^{l+1,T} \cdot f'^T (U^l)$$

from (D)

$$\nabla_w J = H^{l-1} (V_w J)^T$$

$$\nabla_b J = (V_w J)^T I_p$$

Gradients are backpropagated from the output layer to the input layer

Example 4: DNN on California Housing data

Depth of DNN

Example 5

California housing data:

<https://developers.google.com/machine-learning/crash-course/california-housing-data-description>

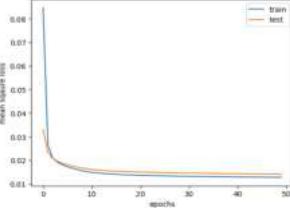
Predicting housing price from other 8 variables

DNN with two hidden layers:

[8, 10, 5, 1]

```
relu_stack = nn.Sequential(
    nn.Linear(in_features, no_hidden),
    nn.ReLU(),
    nn.Linear(no_hidden, no_hidden2),
    nn.ReLU(),
    nn.Linear(no_hidden2, no_labels),
)
```

Example 4

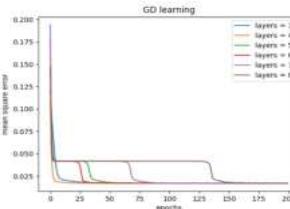


Example 4b: Varying the depth of DNN

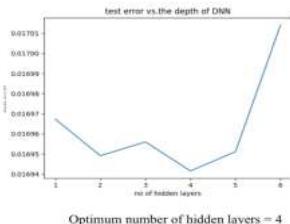
Architectures:

- One hidden layer: [8, 5, 1]
- Two hidden layers: [8, 5, 5, 1]
- Three hidden layers: [8, 5, 5, 5, 1]
- Four hidden layers: [8, 5, 5, 5, 5, 1]
- Five hidden layers: [8, 5, 5, 5, 5, 5, 1]

Example 4



Example 4



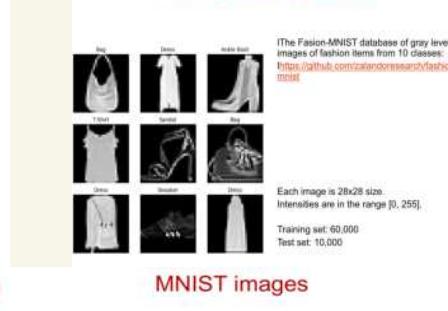
Optimum number of hidden layers = 4

The deep networks extract features at different levels of complexity for regression or classification. However, the **depth** or the number of layers that you can have for the networks depend on the number of training patterns available. The deep networks have more parameters (weights and biases) to learn, so need more data to train. Deep networks can learn complex mapping accurately if sufficient training data is available.

The optimal number of layers is determined usually through experiments. The optimal architecture minimizes the error (training, test, and validation).

at least 1 training point for 1 parameter

Fashion MNIST dataset

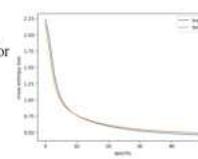


MNIST images

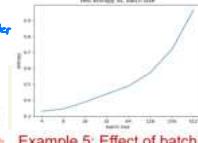


An image is divided into rows and columns and defined by its pixels.

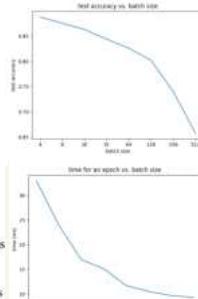
Size of the image = rows x columns pixels



Example 5b: Effect of batch size



Example 5: Effect of batch size



Mini-batch SGD

In practice, gradient descent is performed on *mini-batch* updates of gradients within a *batch* or *block* of data of size *B*. In mini-batch SGD, the data is divided into blocks and the gradients are evaluated on blocks in an epoch in random order.

- $B = 1$: stochastic (online) gradient descent
- $B = P$ (size of training data): (batch) gradient descent
- $1 < B < P \rightarrow$ mini-batch stochastic gradient descent

When *B* increases, more add-multiply operations per second, taking advantages of parallelism and matrix computations. On the other hand, as *B* increases, the number of computations per update (of weights, biases) increases.

Therefore, the curve of the time for weight update against batch size usually take a U-shape curve. There exists an optimal value of *B* – that depends on the sizes of the caches as well.

Selection of batch size

For SGD, it is desirable to randomly sample the patterns from training data in each epoch. In order to efficiently sample blocks, the training patterns are shuffled at the beginning of every training epoch and then blocks are sequentially fetched from memory.

Typical batch sizes: 16, 32, 64, 128, and 256.

The batch size is dependent on the size of caches of CPU and GPUs.

Summary

- Chain rule for backpropagation of gradients: $\nabla_z J = (\frac{\partial J}{\partial x})^T \cdot \nabla_y$
- FFN with one hidden layer (Shallow FFN)
- Backpropagation for FFN with one hidden layer
- Backpropagation learning for deep FFN (DNN)
- Training deep neural networks (GD and SGD):
 - Forward propagation of activation
 - Backpropagation of gradients
 - Updating weights
- Parameters to be decided: depth, width, and batch size

5 Model Selection and overfitting

Model Selection



In neural networks, there exist several free parameters: learning rate, batch size, no. of layers, number of neurons, etc.

Every set of parameters of the network leads to a specific model.

How do we determine the "optimum" parameter(s) or the model for a given regression or classification problem?

Selecting the best model with the best parameter values

Performance estimates



How do we measure the performance of the network?

Some metrics:

- Mean-square error/Root-mean square error for regression — the mean-squared error or its square root. A measure of the deviation from actual.

$$MSE = \frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2 \text{ and } RMSE = \sqrt{\frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2}$$

- Classification error of a classifier. ~~choose that way we need use not~~

$$\text{classification error} = \sum_{p=1}^P \frac{1}{K} \sum_{k=1}^K \delta(d_p \neq y_p)$$

where d_p is the target and y_p is the predicted output of pattern p . $\delta(\cdot)$ is the indicator function.

True error or apparent error?

Apparent error (training error): the error on the training data. What the learning algorithm tries to optimize.

True error: the error that will be obtained in use (i.e., over the whole sample space). What we want to optimize but unknown.

However, the **apparent error** is not always a good estimate of the **true error**. It is just an optimistic measure of it.

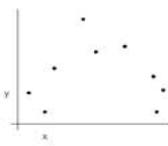
Validation error

Test error: (out-of-sample error) an estimate of the true error obtained by testing the network on some independent data.

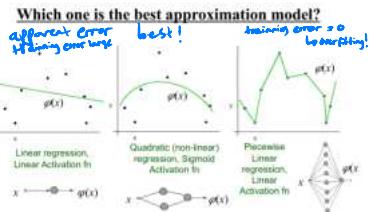
Generally, a larger test set helps provide a greater confidence on the accuracy of the estimate.

Example: Regression

Given the following sample data:

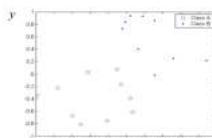


Approximate $y \approx \varphi(x)$ using an NN



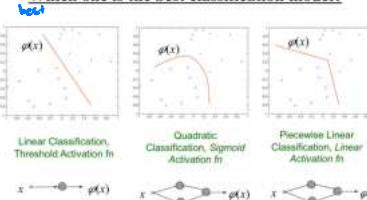
Example: Classification

Given the following sample data:



Classify the following data using an NN.

Which one is the best classification model?



Estimation of true error

Intuition: Choose the model with the best fit to the data?

Meaning: Choose the model that provides the lowest error rate on the entire sample population. Of course, that error rate is the *true error rate*.

"However, to choose a model, we must first know how to **estimate the error** of a model."

The entire **sample population** is often unavailable and only **example data** is available.

Validation

In real applications, we only have access to a finite set of examples, usually smaller than we wanted.

Validation is the approach to use the entire example data available to build the model and estimate the error rate. The validation uses a part of the data to select the model, which is known as the *validation set*.

Validation attempts to solve fundamental problems encountered:

- The model tends to *overfit* the training data. It is not uncommon to have 100% correct classification on training data.
- There is no way of knowing how well the model performs on *unseen data*
- The *error rate estimate* will be overly optimistic (usually lower than the true error rate). Need to get an unbiased estimate.

Validation Methods

An effective approach is to split the entire data into subsets, i.e., Training/Validation/Testing datasets.

Some Validation Methods:

- The Holdout (1/3 - 2/3 rule for test and train partitions)
- Re-sampling techniques
 - Random Subsampling
 - K-fold Cross-Validation
 - Leave-one-out Cross-Validation
- Three-way data splits (train-validation-test partitions)

Solution:

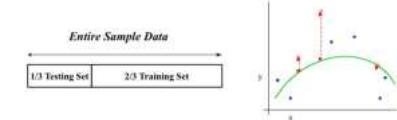
K-fold cross validation

Problem! : there will be overlap
Here will be data that never be used for training

Holdout Method

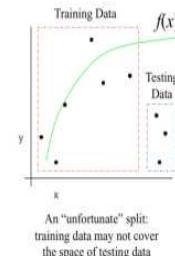
Split entire dataset into two sets:

- Training set (blue): used to train the classifier
- Testing set (red): used to estimate the error rate of the trained classifier on unseen data samples



The holdout method has two basic drawbacks:

- By setting some samples for testing, the training dataset becomes smaller (*bias distribution*)
- Use of a single train-and-test experiment, could lead to misleading estimate if an "unfortunate" split happens



Random Sampling Methods

Limitations of the holdout can be overcome with a family of resampling methods at the expense of more computations:

- Random Subsampling
- K-Fold Cross-Validation
- Leave-one-out (LOO) Cross-Validation

K Data Splits Random SubSampling

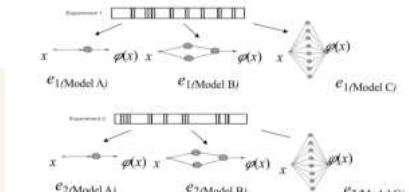
Random Subsampling performs K data splits of the dataset for training and testing.



Each split randomly selects a (fixed) no. of examples. For each data split we retrain the classifier from scratch with the training data. Let the error estimate obtained for i th split (experiments) be e_i .

$$\text{Average test error} = \frac{1}{K} \sum_{i=1}^K e_i$$

Example: K=2 Data Splits Random SubSampling



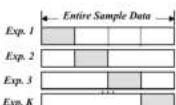
$$\text{Test error}_{(M)} = \frac{1}{2} (e_{1(M)} + e_{2(M)})$$

Choose the model with the best test error, i.e., lowest average test error!

K-fold Cross-Validation

Create a K-fold partition of the dataset:

- For each of K experiments, use $K-1$ folds for training and the remaining one-fold for testing

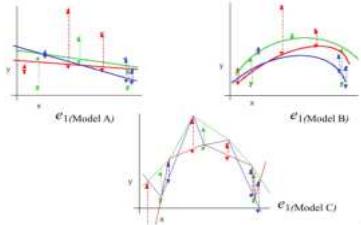


Let the error estimate for i th experiment on test partition be e_i

$$\text{Cross-validation error} = \frac{1}{K} \sum_{i=1}^K e_i$$

K-fold cross validation is similar to Random Subsampling. The advantage of K-fold Cross validation is that all examples in the dataset are eventually used for both training and testing.

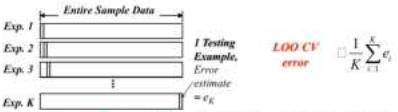
Example: 3-fold Cross-Validation



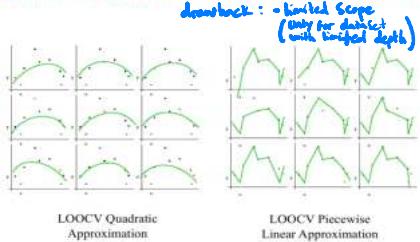
Leave-One-Out (LOO) Cross-Validation

Leave-One-Out is the degenerate case of **K-Fold Cross Validation**, where K is chosen as the total number of examples:

- For a dataset with N examples, perform N experiments, i.e., $N = K$.
- For each experiment use $N-1$ examples for training and the remaining one example for testing.

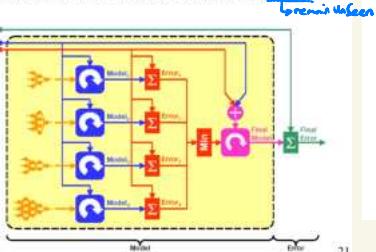


Example: Leave-One-Out Cross-Validation



Three-Way Data Splits Method

Dataset is partitioned in to training set, validation set, and testing set.



Three-Way Data Splits Method

If model selection and true error estimates are to be computed simultaneously, the data needs to be divided into three disjoint sets:

- Training set:** examples for learning to fit the parameters of several possible classifiers. In the case of DNN, we would use the training set to find the "optimal" weights with the gradient descent rule.
- Validation set:** examples to determine the error J_m of different models m , using the validation set. The optimal model m^* is given by
$$m^* = \underset{m}{\operatorname{argmin}} J_m$$
- Test set:** examples used only to assess the performance of a trained model m^* . We will use the test data to estimate the error rate after we have trained the final model with train + validation data.

Why separate test and validation sets?

- The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is also used to select in the process of final model selection.
- After assessing the final model, an independent test set is required to estimate the performance of the final model.

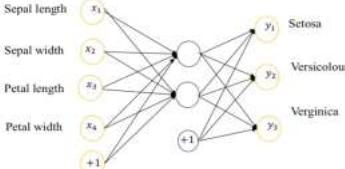
"NO FURTHERING TUNING OF THE MODEL IS ALLOWED!"

Examples 1-3: Iris dataset

<https://archive.ics.uci.edu/ml/datasets/Iris>

Three classes of iris flower: Setosa, versicolour, and virginica
Four features: Sepal length, sepal width, petal length, petal width
150 data points

DNN with one hidden layer. Determine number of hidden neurons?

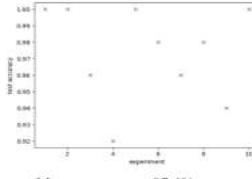


Example 1a: Random subsampling

150 data points

In each experiment, 50 points for testing and 100 for training

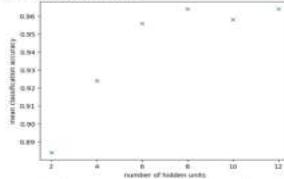
Example: 5 hidden neurons, 10 experiments



Mean accuracy = 97.4%

Example 1b

For different number of hidden units, misclassification errors in 10 experiments



Optimum number of hidden units = 8
Accuracy = 96.4%

↓ because simpler model → avoids overfitting issues run quicker

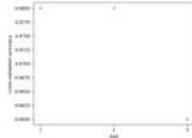
Example 2a: Cross validation

150 data points

3-fold cross validation: 50 data points in one fold.

Two folds are used for training and the remaining fold for testing

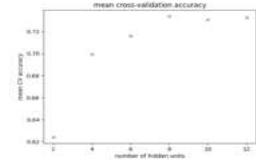
Example: hidden number of units = 5



3-fold cross-validation (CV) accuracy = 97.3%

Example 2b

Mean CV error for 10 experiments for different number of hidden units:

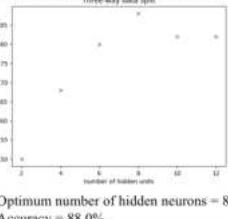


Optimum number of hidden neurons = 8
Cross-validation accuracy = 73.4%

Example 3a: Three-way data splits

150 data points

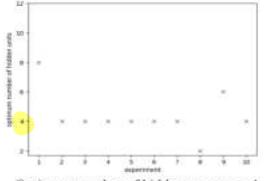
50 data points each for training, for validation, and for testing.



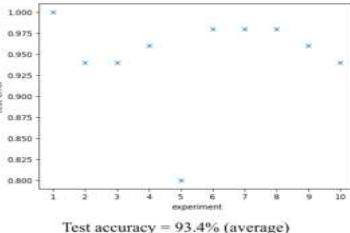
Optimum number of hidden neurons = 8
Accuracy = 88.0%

Example 3b

One way to further improve the results is to repeat it for many experiments, say for 10 experiments:



Optimum number of hidden neurons = 4



Test accuracy = 93.4% (average)

Optimum number of hidden units = 8

Accuracy = 96.4%

↓ because simpler model → avoids overfitting issues run quicker

Model Complexity

Complex models: models with many adjustable weights and biases will

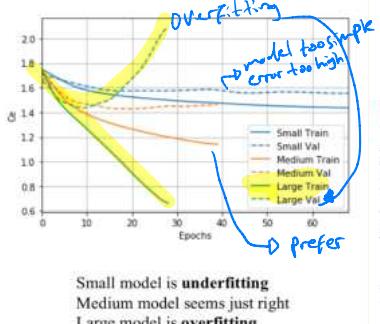
- more likely to be able to solve the required task,
- more likely to memorize the training data without solving the task.

Simple models: The simpler the model that can learn from the training data is more likely to generalize over the entire sample space. May not learn the problem well.

This is the fundamental trade-off:

- Too simple — cannot do the task because not enough parameters to learn. e.g., 5 hidden neurons. (*underfitting*)
- Too complex — cannot generalize from small and noisy datasets well, e.g., 20 hidden neurons. (*overfitting*)

Overfitting and underfitting



Overfitting

Overfitting is one of the problems that occur during training of neural networks, which drives the training error of the network to a very small value at the expense of the test error. The network learns to respond correctly to the training inputs by remembering them too much but is unable to generalize to produce correct outputs to novel inputs.

- Overfitting happens when the amount of training data is inadequate in comparison to the number of network parameters to learn.
- Overfitting occurs when the weights and biases become too large and are fine-tuned to remember the training patterns too much.
- Even your model is right, too much training can cause overfitting.

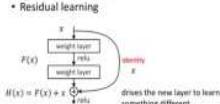
architectures - GoogLeNet

- An important lesson - go deeper
- Inception structures (v2, v3, v4)
 - Reduce parameters (4M vs 60M in AlexNet)



- Batch normalization
 - Normalizes the activation for each training mini-batch
 - Allows us to use much higher learning rates and be less careful about initialization

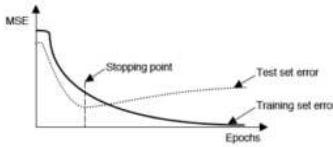
- An important lesson - go deeper
- Escape from 100 layers
 - Residual learning



Methods to overcome overfitting

- Early stopping
- Regularization of weights
- Dropouts

Early stopping



Training of the network is to be stopped when the validation error starts increasing. Early stopping can be used in test/validation by stopping when the validation error is minimum.

Regularization of weights

During overfitting, some weights attain large values to reduce training error, jeopardizing its ability to generalize. In order to avoid this, a *penalty term* (*regularization term*) is added to the cost function.

For a network with weights $\mathbf{W} = [w_{ij}]$ and bias \mathbf{b} , the penalized (or regularized) cost function $J(\mathbf{W}, \mathbf{b})$ is defined as

$$J = f + \beta_1 \sum_{ij} |w_{ij}| + \beta_2 \sum_{ij} (w_{ij})^2$$

where $J(\mathbf{W}, \mathbf{b})$ is the standard cost function (i.e., m.s.e. or cross-entropy).

$$L^1 - \text{norm} = \sum_{ij} |w_{ij}|$$

$$L^2 - \text{norm} = \sqrt{\sum_{ij} (w_{ij})^2}$$

And β_1 and β_2 are known as L^1 and L^2 regularization (penalty) constants, respectively. These penalties discourage weights from attaining large values.

L2 regularization of weights

Regularization is usually not applied on bias terms. L^2 regularization is most popular on weights.

$$J_2 = J + \beta_2 \sum_{ij} (w_{ij})^2$$

Gradient of the regularized cost wrt weights:

$$\nabla_{\mathbf{W}} J_2 = \nabla_{\mathbf{W}} J + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial \mathbf{W}}$$

(A)

$$\begin{aligned} L^2 \text{ norm} &= \sum_{ij} (w_{ij})^2 = w_{11}^2 + w_{12}^2 + \dots + w_{R1}^2 \\ \frac{\partial (\sum_{ij} (w_{ij})^2)}{\partial \mathbf{W}} &= \begin{pmatrix} \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{11}} & \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{12}} & \dots & \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{R1}} \\ \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{21}} & \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{22}} & \dots & \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{R2}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{n1}} & \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{n2}} & \dots & \frac{\partial \sum_{ij} (w_{ij})^2}{\partial w_{Rn}} \end{pmatrix} = 2\mathbf{W} \end{aligned}$$

Substituting in (A):

$$\nabla_{\mathbf{W}} J_2 = \nabla_{\mathbf{W}} J + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial \mathbf{W}} = \nabla_{\mathbf{W}} J + 2\beta_2 \mathbf{W}$$

For gradient decent learning that uses regularized cost function:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J_2$$

Substituting above:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha (\nabla_{\mathbf{W}} J + 2\beta_2 \mathbf{W})$$

where $\beta = 2\beta_2$

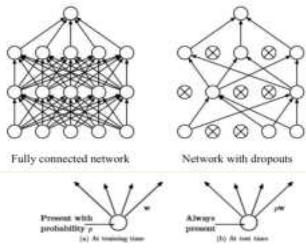
β is known as the *weight decay parameter*.

That is for L^2 regularization, the weight matrix is weighted by decay parameter and added to the gradient term.

Dropouts

Overfitting can be avoided by training only a fraction of weights in each iteration. The key idea of ‘dropouts’ is to randomly drop neurons (along with their connections) from the networks during training.

This prevents neurons from co-adapting and thereby reduces overfitting.



48

At the training time, the units (neurons) are present with a probability p and presented to the next layer with weights multiplied by probability p . That is, The output at the test time is multiplied by $\frac{1}{p}$.

This results in a scenario that at test time, the weights are always present and presented to the network with weights multiplied by probability p . Applying dropouts result in a ‘thinned network’ that consists of only neurons that survived. This minimizes the redundancy in the network.

Dropout ratio

Dropout ratio is the fraction of neurons to be dropped out at one forward step.

Dropout ratio (p) has to be specified with `nn.Dropout()` in torch after activation function.

`nn.Dropout(p=0.2)`

Example 5: dropouts

When training with dropouts, we train on a thinned network dropping out units in each mini-batch.

```
class NeuralNetwork(nn.Module):
    def __init__(self, batch_size=100, drop_out=0.5):
        super(NeuralNetwork, self).__init__()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(128*32*3),
            nn.ReLU(),
            nn.Linear(128),
            nn.ReLU(),
            nn.Linear(10),
            nn.ReLU(),
            nn.Dropout(p=drop_out),
            nn.Linear(10),
            nn.Softmax(dim=1)
        )
    def forward(self, x):
        x = x.flatten(start_dim=1)
        logits = self.linear_relu_stack(x)
        return logits
```

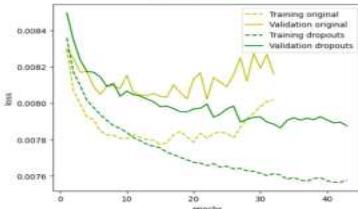
`def forward(self, x):`

`x = x.flatten(start_dim=1)`

`logits = self.linear_relu_stack(x)`

`return logits`

Example 5



Summary

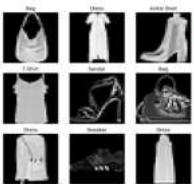
• Model selection

- Holdout method
- Resampling methods
 - Random subsampling
 - K-fold cross-validation
 - LOO cross-validation
- Three-way data split

• Methods to overcome overfitting

- Early stopping
- Weight regularization
- Dropouts

Fashion MNIST dataset



<https://github.com/zalandoresearch/fashion-mnist>

Example 4: Early stopping and weight decay

DNN with [784, 400, 400, 400, 10] architecture.
Let's implement L2 regularization with $\beta = 0.0001$
Use early stopping to terminate learning.

```
class NeuralNetwork(Module):
    def __init__(self, hidden_size=500):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Example 4

```
class EarlyStopper:
    def __init__(self, patience=5, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_validation_loss = np.inf
```

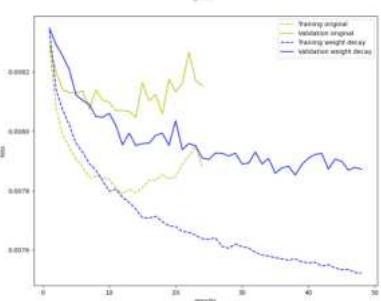
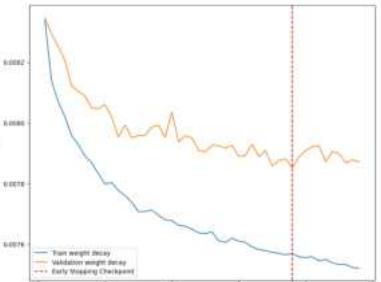
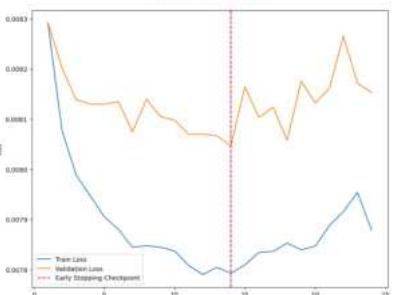
```
def early_stop(self, validation_loss):
    if validation_loss < self.min_validation_loss:
        self.min_validation_loss = validation_loss
        self.counter = 0
    elif validation_loss > (self.min_validation_loss + self.min_delta):
        self.counter += 1
    if self.counter >= self.patience:
        return True
    return False
```

Example 4

```
model = NeuralNetwork()
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.001)
early_stopper = EarlyStopper(patience=patience, min_delta)
```

```
if early_stopper.early_stop(loss):
    print("Done!")
    break
```

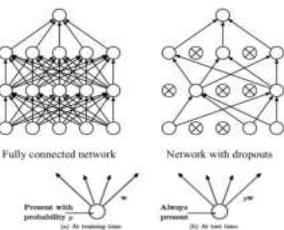
Example 4



Dropouts

Oversfitting can be avoided by training only a fraction of weights in each iteration. The key idea of "dropouts" is to randomly drop neurons (along with their connections) from the networks during training.

This prevents neurons from co-adapting and thereby reduces overfitting.



At the training time, the units (neurons) are present with a probability p and presented to the next layer with weight \mathbf{W} to the next layer.

This results in a scenario that at test time, the weights are always present, and presented to the network with weights multiplied by $\frac{1}{p}$.

Applying dropouts result in a "thinned network" that consists of only neurons that survived. This minimizes the redundancy in the network.

Dropout ratio

Dropout ratio is the fraction of neurons to be dropped out at one forward step.

Dropout ratio (p) has to be specified with `nn.Dropout()` in torch after activation function.

`nn.Dropout(p=0.2)`

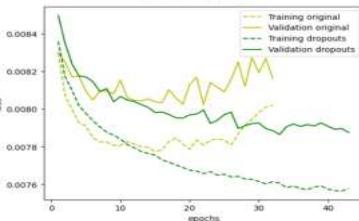
Example 5: dropouts

When training with dropouts, we train on a thinned network dropping out units in each mini-batch.

```
class NeuralNetwork_Dropouts(Module):
    def __init__(self, hidden_size=100, drop_out=0.5):
        super(NeuralNetwork_Dropouts, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(p=drop_out),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Example 5



Summary

• Model selection

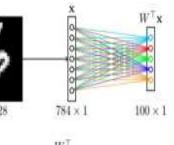
- Holdout method
- Resampling methods
 - Random subsampling
 - K-fold cross-validation
 - LOO cross-validation
- Three-way data split

• Methods to overcome overfitting

- Early stopping
- Weight regularization
- Dropouts

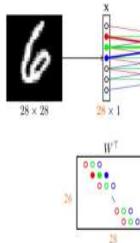
6 CNN

Motivation



- Spatial organization of the input is destroyed.
- The network is not invariant/equivariant to transformations (e.g., translation).

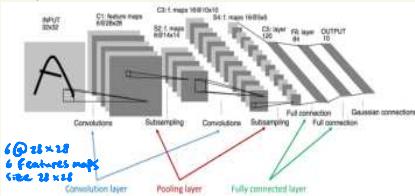
Locally connected networks



Outline

- Basic components in CNN
- Training a classifier
- Optimizers

Basic Components in CNN
example of CNN (Convolutional network)
- LeNet 5

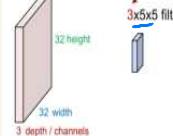


As we go deeper [left to right] the height and width tend to go down and the number of channels increased.

Convolution layer

Filters always extend the full depth of the input volume

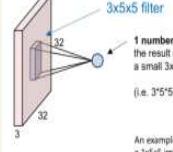
$3 \times 32 \times 32$ image



Weights of filters are learned using backpropagation algorithm.
The weights learned are known as **filters** or **kernels**

Convolve the filter with the image i.e. "slide over the image spatially, computing dot products"

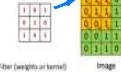
$3 \times 32 \times 32$ image



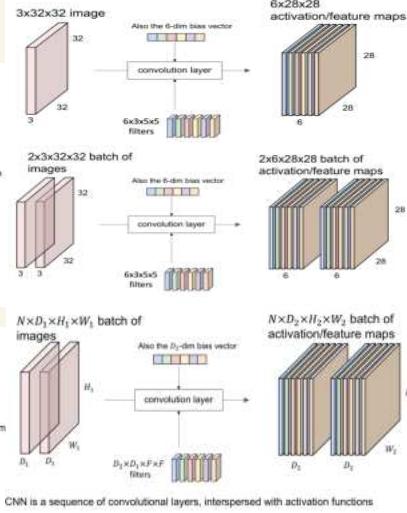
1 number:
the result of taking a dot product between the filter and a small 3×5 chunk of the image

(i.e. $3 \times 5 = 75$ -dimensional dot product + bias)

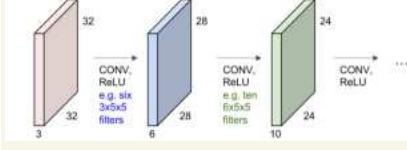
An example of convolving a 3×5 image with a $1 \times 3 \times 5$ filter



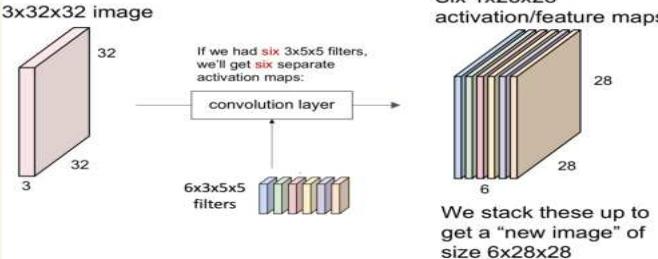
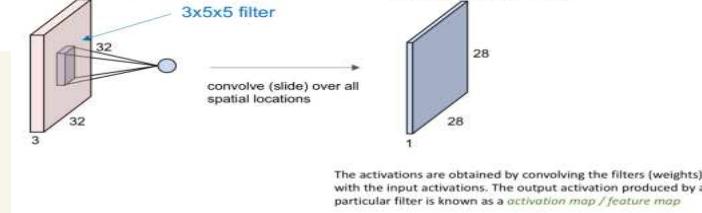
$1 \times 28 \times 28$ activation/feature map



CNN is a sequence of convolutional layers, interspersed with activation functions



Pattern = edge



Convolution layer

Consider a kernel $w = \{w(l, m)\}$, which has a size of $L \times M$, $L = 2a + 1$, $M = 2b + 1$

Synaptic input at location $p = (i, j)$ of the first hidden layer due to a kernel is given by

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i+l, j+m)w(l, m) + \text{bias}$$

For instance, given $L = 3, M = 3$

$$\rightarrow 3 \times 3$$

$$u(i, j) = x(i - 1, j - 1)w(-1, -1) + x(i - 1, j)w(-1, 0) + \dots + x(i, j)w(0, 0) + x(i + 1, j + 1)w(1, 1) + \text{bias}$$

The output of the neuron at (i, j) of the convolution layer

$$y(i, j) = f(u(i, j))$$

where f is an activation function. For deep CNN, we typically use ReLU, $f(x) = \max(0, x)$.

Note that one weight tensor $w_k = \{w_k(l, m)\}$ or kernel (filter) creates one feature map:

$$y_k = \{y_k(i, j)\}$$

If there are K weight vectors $\{w_k\}_{k=1}^K$, the convolutional layer is formed by K feature maps

$$y = \{y_k\}_{k=1}^K$$

Convolution by doing a sliding window



As a guiding example, let us consider the convolution of single-channel tensors $x \in \mathbb{R}^{4 \times 4}$ and $w \in \mathbb{R}^{3 \times 3}$:

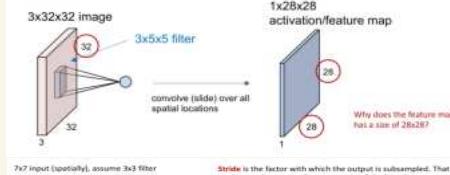
$$w \cdot x = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 8 & 8 \\ 1 & 8 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 & 8 \\ 6 & 5 & 7 & 8 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$w \cdot x = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i+l, j+m)w(l, m) + \text{bias}.$$

$$u(1,1) = 4 \cdot 1 + 5 \cdot 4 + 8 \times 1 + 1 \times 1 + 8 \cdot 4 + 8 \times 3 + 3 \times 3 + 1 \times 3 + 6 \times 1 = 122 \\ u(1,2) = 4 \cdot 1 + 5 \cdot 4 + 8 \cdot 7 + 1 \times 1 + 8 \cdot 4 + 8 \times 3 + 6 \times 3 + 1 \times 3 + 6 \times 1 = 148 \\ u(2,1) = 1 \cdot 1 + 8 \cdot 4 + 8 \times 1 + 3 \times 1 + 6 \times 4 + 6 \times 3 + 6 \times 3 + 5 \times 3 + 7 \times 1 = 126 \\ u(2,2) = 8 \cdot 1 + 8 \cdot 4 + 8 \times 1 + 6 \times 1 + 6 \times 4 + 4 \times 3 + 5 \times 3 + 7 \times 3 + 8 \times 1 = 134$$

Convolution layer – spatial dimensions



7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.



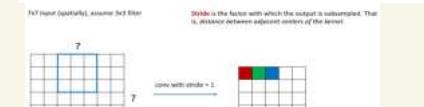
7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.



7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.

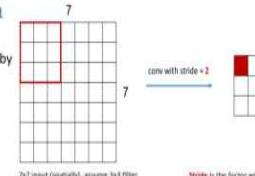


7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.

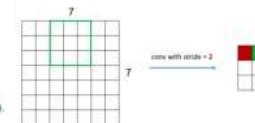
7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.



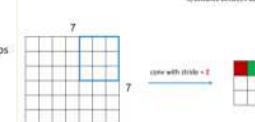
7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.



7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.



7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.

Example

Input volume: 3x32x32

Ten 3x5x5 filters with stride 1, pad 2

Output volume size: ?

Output size =

$$\frac{N - f + 2P}{S} + 1$$

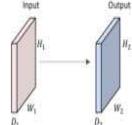
$$\frac{32 - 5 + 2 \cdot 2}{5} + 1 = 13 \text{ spatially}$$

The output volume size is 10x32x32

Convolution layer - summary

A convolution layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters:
 - Number of filters K
 - Their spatial extent F
 - The stride S
 - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$ where
 - $D_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d -th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias



$N \times N$ input (spatially), assume $F \times F$ filter, and S stride

$$\text{Output size} = \frac{N - F}{S} + 1$$

e.g. $N = 7, F = 3$

stride 1 $\Rightarrow (7 - 3)/1 + 1 = 5$

stride 2 $\Rightarrow (7 - 3)/2 + 1 = 3$

stride 3 $\Rightarrow (7 - 3)/3 + 1 = 2$

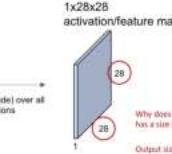
\downarrow



3x32x32 image

3x5x5 filter

convolve (slide) over all spatial locations



1x28x28 activation/feature map

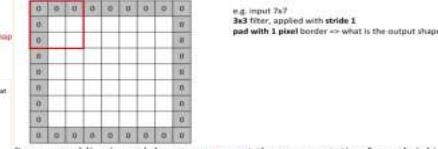
Why does the feature map have a size of 28x28?

$$\text{Output size} = \frac{N - F}{S} + 1$$

The valid feature map is smaller than the input after convolution. Without zero-padding, the width of the representation shrinks by the $F - 1$ at each layer. To avoid shrinking the spatial extent of the network rapidly, small filters have to be used.

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized).



e.g. input 3x3
3x3 filter, applied with stride 1
pad with 1 pixel border \Rightarrow what is the output shape?

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized).



e.g. input 3x3
3x3 filter, applied with stride 1
pad with 1 pixel border \Rightarrow what is the output shape?

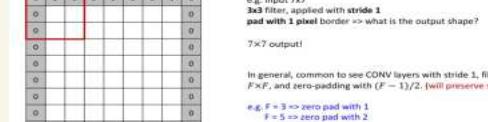
$$\text{Recall that without padding, output size} = \frac{N - F}{S} + 1$$

$$\text{With padding, output size} = \frac{N - F + 2P}{S} + 1$$

e.g. $N = 7, F = 3$

$$\text{stride 1} \Rightarrow (7 - 3 + 2 \cdot 1)/1 + 1 = 7$$

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized).



e.g. input 3x3
3x3 filter, applied with stride 1
pad with 2 pixel border \Rightarrow what is the output shape?

7x7 output!

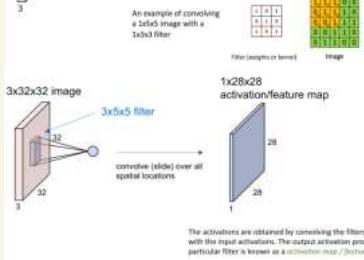
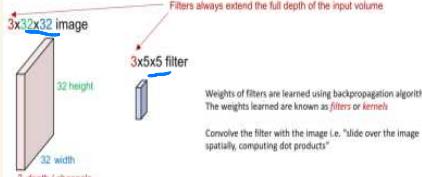
In general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F - 1)/2$. (will preserve size spatially)

e.g. $F = 3 \Rightarrow$ zero pad with 1

$F = 5 \Rightarrow$ zero pad with 2

$F = 7 \Rightarrow$ zero pad with 3

Convolution layer



Consider a second, green filter

3x32x32 image

3x5x5 filter

convolve (slide) over all spatial locations

Two 1x28x28 activation/feature maps

Six 1x28x28 activation/feature maps

If we had six 3x5x5 filters, we'll get six separate activation maps:

convolution layer

6x3x5x5 filters

We stack these up to get a "new image" of size 6x28x28

2x3x32x32 batch of images

Also the 8-dim bias vector:

convolution layer

8x3x5x5 filters

NxD₁xH₁xW₁ batch of images

Also the D₁-dim bias vector

convolution layer

D₂xH₂xW₂ batch of activation/feature maps

H₂

CNN is a sequence of convolutional layers, interspersed with activation functions:

CONV, ReLU e.g. six 3x5x5 filters

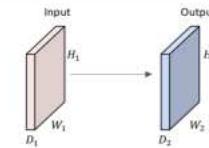
CONV, ReLU e.g. ten 6x5x5 filters

CONV, ReLU

Convolution layer - summary

A convolution layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
 - Number of filters K
 - Their spatial extent F
 - The stride S
 - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $D_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d -th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias



Pattern = edge

Softmax function

The softmax step can be seen as a generalized logistic function input a vector of scores $z \in \mathbb{R}^n$ and outputs a vector of output

$p \in \mathbb{R}^n$ through a softmax function at the end of the architecture

$$z = [2.0, 1.0, 0.1] \quad S(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$p =$

$$p = \frac{e^{2.0}}{e^{2.0} + e^{1.0} + e^{0.1}}$$

$p =$

$$p = \frac{e^{1.0}}{e^{2.0} + e^{1.0} + e^{0.1}}$$

$p =$

$$p = \frac{e^{0.1}}{e^{2.0} + e^{1.0} + e^{0.1}}$$

Where does the Softmax function fit in a CNN arch?

Softmax's input is the output of the fully connected immediately preceding it, and it outputs the final or entire neural network. This output is a probability distribution of all the label class candidates.

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet) ($\text{mean image} = [32,32,3]$ array)
 - Subtract per-channel mean (e.g. VGGNet) ($\text{mean along each channel} = 3$ numbers)
 - Subtract per-channel mean and Divide by per-channel std (e.g. ResNet) ($\text{mean along each channel} = 3$ numbers)
- Parameters $w = (w_1, w_2, \dots, w_L)$

Empirical loss function

$$L(w) = \frac{1}{n} \sum_i l(y_i, f_w(x_i))$$

$$w^* = \operatorname{argmin}_w L(w)$$

- In 1-dimension, the derivative of a function gives the slope:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$



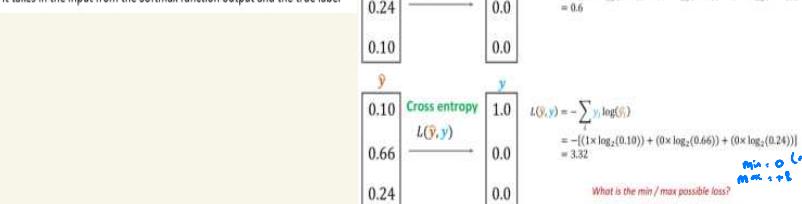
- In multiple dimensions, the gradient is the vector of (partial derivatives) along each dimension
- The direction of steepest descent is the negative gradient

Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label

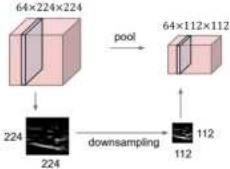


Random search is a bad idea

Follow the slope



Pooling layer



- Operates over each activation map independently
- Either 'max' or 'average' pooling is used at the pooling layer. That is, the convolved features are divided into disjoint regions and pooled by taking either maximum or averaging.

MAX Pooling

Single depth slice

dim 1	1	1	2	4
dim 2	5	6	7	8
	3	2	1	0
	1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4

Pooling is intended to subsample the convolution layer.
The default stride for pooling is equal to the filter width.

Consider pooling with non-overlapping windows $\{y_{l,m}\}_{l=1}^{L/2M/2},_{m=-L/2-M/2}$, of size $L \times M$

The **max pooling** output is the maximum of the activation inside the pooling window. Pooling of a feature map y at $p = (l, m)$ produce pooled feature

$$z(l, m) = \max_{i,j} y(i + l, j + m)$$

Pooling layer - summary

The **mean pooling** output is the mean of activations in the pooling window

$$z(l, m) = \frac{1}{L \times M} \sum_l \sum_m y(i + l, j + m)$$

Why pooling?

A function f is invariant to g if $f(g(x)) = f(x)$

- Pooling layers can be used for building inner activations that are (slightly) invariant to small variations of the input.
- Invariance to local translation is helpful if we care more about the presence of a pattern rather than its exact position.

Example 1

Given an input pattern X :

$$X = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.1 & -0.3 \\ 0.1 & -0.1 & 0.5 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}$$

The input pattern is received by a convolution layer consisting of one kernel (filter)

$$W = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

and bias = 0.05.

- If convolution layer has a sigmoid activation function,
- Find the outputs of the convolution layer if the padding is VALID at strides = 1
 - Assume the pooling layer uses max pooling, has a pooling window size of 2x2, and strides = 2, find the activations at the pooling layer
 - Repeat (a) and (b) using convolution layer with SAME padding.

Find the outputs of the convolution layer if the padding is VALID at strides = 1

$$I = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.1 & -0.3 \\ 0.1 & -0.1 & 0.5 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}, W = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

Synaptic input to the pooling-layer:

$$u(l, m) = \sum_i \sum_j x(i + l, j + m)w(i, m) + b$$

For VALID padding:

$$\begin{aligned} u(1,1) &= 0.5 \times 0 + 0.1 \times 1 + 0.2 \times 1 + 0.3 \times 1 + 0.1 \times 0 - 0.5 \times 1 - 0.1 \times 1 + 0.2 \times 1 + 0.3 \times 1 + 0.0 \times 0 + 0.05 = -0.35 \\ u(1,2) &= -0.1 \times 0 + 0.2 \times 1 + 0.3 \times 1 + 0.1 \times 1 - 0.5 \times 0 + 0.5 \times 1 + 0.2 \times 1 + 0.0 \times 1 + 0.3 \times 0 + 0.05 = 1.25 \\ u(2,1) &= 0.2 \times 0 + 0.3 \times 1 + 0.5 \times 1 + 0.0 \times 1 + 0.1 \times 1 + 0.0 \times 1 + 0.3 \times 1 + 0.2 \times 0 + 0.05 = 0.75 \\ u(2,2) &= 0.0 \times 0 + 0.1 \times 1 - 0.5 \times 1 - 0.1 \times 1 + 0.2 \times 0 + 0.0 \times 1 + 0.7 \times 1 + 0.2 \times 1 + 0.2 \times 0 + 0.05 = -0.55 \end{aligned}$$

Example 2

Inputs are digit images from MNIST database: <http://yann.lecun.com/exdb/mnist/>

Input image size = 28x28

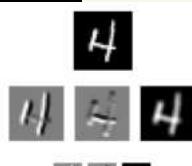
First convolution layer consists of three filters:

$$w_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}, w_2 = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, w_3 = \begin{pmatrix} 3 & 4 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

Find the feature maps at the convolution layer and pooling layer. Assume zero bias

For convolution layer, use a stride = 1 (default) and padding = 'VALID';

For pooling layer, use a window of size 2x2 and a stride if 2.



Original image 28 x 28

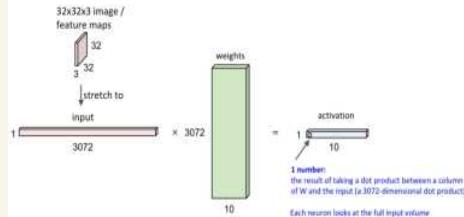
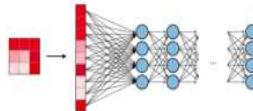
Output of convolution layer 9 x 26 x 26

Output of pooling layer 3 x 13 x 13

Fully connected layer

The fully connected layer (FC) operates on a **flattened input** where each input is connected to all neurons.

If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.

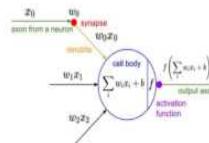


Activation function

Recall that the output of the neuron at (i, j) of the convolution layer

$$y(i, j) = f(u(i, j))$$

where u is the synaptic input and f is an activation function (it aims at introducing non-linearities to the network).



Sigmoid

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

tanh

$$\tanh(x)$$

ReLU

$$\max(0, x)$$

Leaky ReLU

$$\max(0.1x, x)$$

Maxout

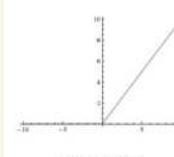
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

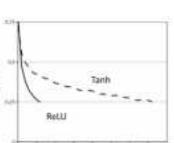
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

See eg. 1.1 pyth. The activation function is usually an abstraction representing the rate of firing in the cell

Activation function – ReLU



- Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$



- (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.

- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.

See eg. 1.1 pyth.

Training a classifier

Multi-class classification



CIFAR-10

- 10 classes
- 6000 images per class
- 60000 images - 50000 training images and 10000 test images
- Each image has a size of 3x32x32, that is 3-channel color images of 32x32 pixels in size



MNIST

- Size-normalized and centred 1x28x28
- 784 inputs
- Training set = 60,000 images
- Testing set = 10,000 images

Softmax function

The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $\mathbf{z} \in \mathbb{R}^n$ and outputs a vector of output probability $\mathbf{p} \in \mathbb{R}^n$ through a softmax function at the end of the architecture.

$$\begin{array}{c} z \\ \hline \begin{matrix} 2.0 \\ 1.0 \\ 0.1 \end{matrix} \end{array} \quad S(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad \begin{array}{l} p = 0.66 \\ p = 0.24 \\ p = 0.10 \end{array}$$

Numeric output of the last linear layer of a multi-class classification neural network



Where does the Softmax function fit in a CNN architecture?

Softmax's input is the output of the fully-connected layer immediately preceding it, and it outputs the final output of the entire neural network. This output is a probability distribution of all the label class candidates.

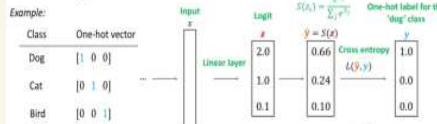
Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label

One-hot encoded ground truth



$$\begin{array}{cc} \hat{y} & y \\ \hline 0.66 & 1.0 \\ 0.24 & 0.0 \\ 0.10 & 0.0 \end{array} \quad L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i) = -(1 \times \log_2(0.66)) + (0 \times \log_2(0.24)) + (0 \times \log_2(0.10)) = 0.6$$

$$\begin{array}{cc} \hat{y} & y \\ \hline 0.10 & 1.0 \\ 0.66 & 0.0 \\ 0.24 & 0.0 \end{array} \quad L(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i) = -(1 \times \log_2(0.10)) + (0 \times \log_2(0.66)) + (0 \times \log_2(0.24)) = 3.32$$

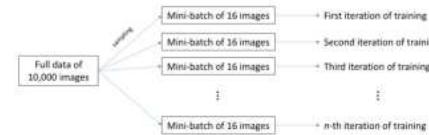
min: 0 max: 1.32

What is the min / max possible loss?

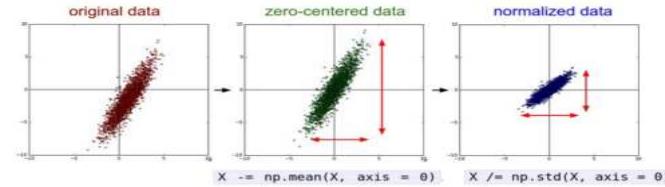
Epoch and mini-batch

- **Epoch** – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

- **Mini-batch** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on **mini batches**, where the number of data points in a batch is a hyperparameter that we can tune.



Data pre-processing



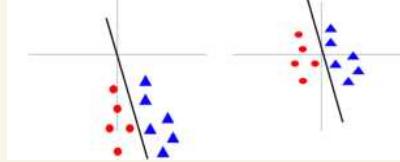
(Assume $X [NxD]$ is data matrix, each example in a row)

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet) (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet) (mean along each channel = 3 numbers)
- Subtract per-channel mean and Divide by per-channel std (e.g. ResNet) (mean along each channel = 3 numbers)

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

After normalization: less sensitive to small changes in weights; easier to optimize



Optimization

- A CNN as composition of functions $f_w(x) = f_L(\dots(f_2(f_1(x; w_1); w_2) \dots; w_L)$

$$w = (w_1, w_2, \dots, w_L)$$

- Parameters

$$L(w) = \frac{1}{n} \sum_i l(y_i, f_w(x_i))$$

$$w^* = \operatorname{argmin}_w L(w)$$

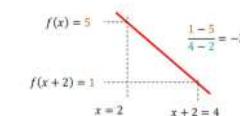
Random search is a bad idea

Follow the slope



- In 1-dimension, the derivative of a function gives the slope:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$



- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

- The direction of steepest descent is the **negative gradient**

Gradient descent (GD)

- Iteratively step in the direction of the negative gradient

- Gradient descent

$$w^{t+1} = w^t - \eta_t \frac{\partial f(w^t)}{\partial w}$$

New weight
Old weight
Learning rate
Gradient



- Batch Gradient Descent

- Full sum is expensive when N is large

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

- Stochastic Gradient Descent (SGD)

- Approximate sum using a minibatch of examples
- $32 / 64 / 128$ common minibatch size
- Additional hyperparameter on batch size

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
    minibatch = sample_data(data, batch_size)
    dw = compute_gradient(loss_fn, minibatch, w)
    w -= learning_rate * dw
```

GD with Momentum

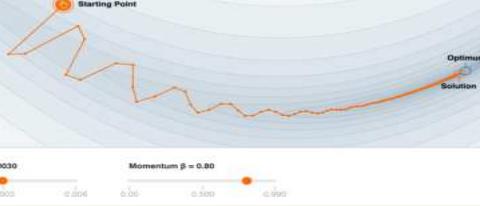
Deep neural networks have very complex error profiles. The method of momentum is designed to **accelerate learning**, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to **oscillate** near the optimum. This leads to very slow converging **rates**. This problem is typical in deep learning architecture.

Momentum is one method of **speeding** the convergence along a narrow ravine.



Step-size $\alpha = 0.0030$ Momentum $\beta = 0.0$



Momentum update is given by:

$$\begin{aligned} V &\leftarrow \gamma V - \alpha \nabla W \\ W &\leftarrow W + V \end{aligned}$$

where V is known as the **velocity** term and has the same dimension as the weight vector W .

The momentum parameter $\gamma \in [0, 1]$ indicates how many iterations the previous gradients are incorporated into the current update.

The momentum algorithm **accumulates an exponentially decaying moving average of past gradients** and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

Learning rate

$$w^{t+1} = w^t - \eta_t \frac{\partial f(w^t)}{\partial w}$$

New weight
Old weight
Learning rate
Gradient



Learning rate

The learning rate, often noted α or sometimes η , indicates at **which pace** the weights get updated. It can be fixed or adaptively changed.

The current most popular method is called **Adam**, which is a method that adapts the learning rate.

Adaptive learning rates

- Letting the learning rate vary when training a model can **reduce** the training time and **improve** the numerical optimal solution.
- While Adam optimizer is the most commonly used technique, others can also be useful.

Algorithms with adaptive learning rates:

- AdaGrad torch.optim.Adagrad()
- RMSprop torch.optim.RMSprop()
- Adam torch.optim.Adam()

Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to **start with a large learning factor and then gradually reducing it**.

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.

Learning trajectory of a neural network **minimizing non-convex cost function** passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by **scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient**. This improves the learning rates, especially in the convex regions of error function.

$$r \leftarrow r + (\nabla_W)^2$$

$$W \leftarrow W - \frac{\alpha}{\varepsilon + \sqrt{r}} \cdot (\nabla_W)$$

In other words, learning rate:

$$\bar{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{r}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to **discard the history from extreme past** so that it can converge rapidly after finding a convex region.

$$\begin{aligned} r &\leftarrow \rho r + (1 - \rho)(\nabla_W)^2 \\ W &\leftarrow W - \frac{\alpha}{\sqrt{\varepsilon + r}} \cdot (\nabla_W) \end{aligned}$$

The decay constant ρ controls the length of the moving average of gradients. Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer **combines RMSprop and momentum** methods. Adam is generally regarded as **fairly robust** to hyperparameters and works well on many applications.

$$\begin{aligned} \text{Momentum term: } s &\leftarrow \rho_1 s + (1 - \rho_1)\nabla_W \\ \text{Learning rate term: } r &\leftarrow \rho_2 r + (1 - \rho_2)(\nabla_W)^2 \\ s &\leftarrow \frac{s}{1 - \rho_1} \\ r &\leftarrow \frac{r}{1 - \rho_2} \\ W &\leftarrow W - \frac{\alpha}{\varepsilon + \sqrt{r}} \cdot s \end{aligned}$$

Note that s adds the momentum and r contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001$, $\rho_1 = 0.9$, $\rho_2 = 0.999$, and $\varepsilon = 10^{-8}$

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.
Learning trajectory of a neural network **minimizing non-convex cost function** passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. This improves the learning rates, especially in the convex regions of error function.

$$r \leftarrow r + (\nabla_W)^2$$

$$W \leftarrow W - \frac{\alpha}{\varepsilon + \sqrt{r}} \cdot (\nabla_W)$$

In other words, learning rate:

$$\hat{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{r}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to **discard the history from extreme past** so that it can converge rapidly after finding a convex region.

$$r \leftarrow \rho r + (1 - \rho)(\nabla_W)^2$$

$$W \leftarrow W - \frac{\alpha}{\sqrt{\varepsilon + r}} \cdot (\nabla_W)$$

The decay constant ρ controls the length of the moving average of gradients.
Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer **combines RMSprop and momentum** methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

Momentum term: $s \leftarrow \rho_1 s + (1 - \rho_1)(\nabla_W)$
 Learning rate term: $r \leftarrow \rho_2 r + (1 - \rho_2)(\nabla_W)^2$

$$s \leftarrow \frac{s}{1 - \rho_1}$$

$$r \leftarrow \frac{r}{1 - \rho_2}$$

$$W \leftarrow W - \frac{\alpha}{\sqrt{\varepsilon + r}} \cdot s$$

Note that s adds the momentum and r contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001$, $\rho_1 = 0.9$, $\rho_2 = 0.999$, and $\varepsilon = 10^{-8}$

Example 3: MNIST digit recognition

MNIST database: 28x28 = 784 inputs

Training set = 12000 images

Testing set = 2000 images

Input pixel values were normalized to [0, 1]

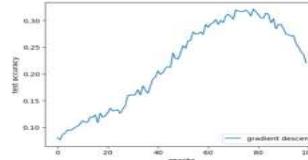


Example 3: Architecture of CNN



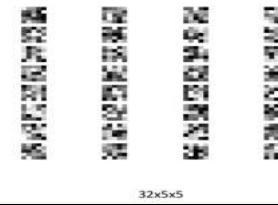
ReLU neurons
Gradient descent optimizer with batch-size = 128
[See eg3.ipynb](#)

Example 3: Training Curve



Example 3: Weights learned

Weights learned at convolution layer 1



32x5x5

Example 3: Feature maps

Input image
28x28



Feature maps at conv1
32x28x28



Feature maps at pool1
32x14x14

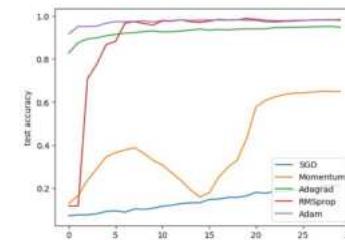


Feature maps at conv2
64x14x14



Feature maps at pool2
64x7x7

Example 4: MNIST recognition with CNN with different learning algorithms



See eg4.ipynb

7 CNN II

Next lecture

CNN Architectures

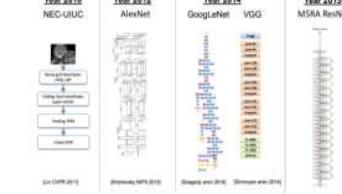
- More on convolution
 - How to calculate FLOPs
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution

Batch normalization

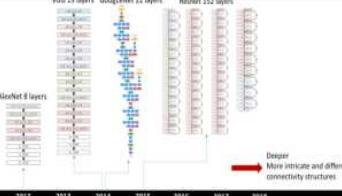
- Prevent overfitting
 - Transfer learning
 - Data augmentation

CNN Architectures

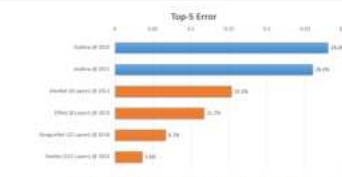
Deep networks for ImageNet



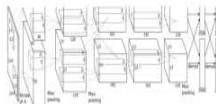
Deep architectures



Performance of previous years on ImageNet



Deep architectures - AlexNet



Escape from a few layers

- ReLU nonlinearity for solving gradient vanishing (**better**)
- Data augmentation
- Dropout
- Outperformed all previous models on ILSVRC by 10%

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels *always bias?*

$96 \times (11 \times 11 \times 3 + 1)$

Deep architectures - GoogLeNet

- An important lesson - go deeper
- Inception structures (v2, v3, v4)
- Reduce parameters (4M vs 60M in AlexNet)



The $[x]$ convolutions are performed to reduce the dimensions of input/output.

Batch normalization

- Normalization the activation for each training mini-batch
- Allows us to use much higher learning rates and be less careful about initialization

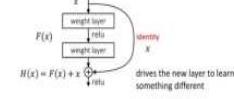
Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

An important lesson - go deeper

- 140M parameters
- Now commonly used for computing perceptual loss

Escape from 100 layers

- Residual learning



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels

Number of parameters = $(11 \times 11 \times 3 + 1) \times 96 = 34,944$

Note: There are no padding layers associated with a pooling layer. The pool size, stride and padding are hyperparameters.

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

Second convolution layer: 256 kernels of size 5x5x96

Number of parameters = $(5 \times 5 \times 96 + 1) \times 256 = 614,656$

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256
Size (width) of the output image of the previous Conv Layer = 6

Number of parameters = $(6 \times 6 \times 4096 + 1) \times 4096 = 37,752,832$

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

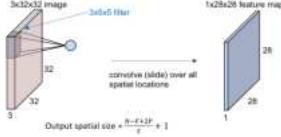
The last FC layer:
Number of neurons = 1000
Number of neurons in the previous FC Layer = 4096
Number of parameters = $(1000 \times 4096 + 1) \times 1000 = 4,097,000$

Standard convolution



- How to calculate the spatial size of output? [Lecture 6](#)
- How to calculate the number of parameters? [Lecture 6](#)
- How to calculate the computations involved?

Recap: How to calculate the spatial size of output?

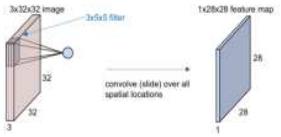


$$\text{Output spatial size} = \frac{N - F + P}{S} + 1$$

In this example, $N = 32$, $F = 5$, $P = 0$, $S = 1$

Thus, output spatial size = $\frac{(32-5+0)}{1} + 1 = 28$

Recap: How to calculate the number of parameters?



Let say we have ten 3x5x5 filters with stride 1, pad 0

Input volume: 3x32x32

Ten 3x5x5 filters with stride 1, pad 0

Number of parameters in this layer: ?



How to calculate the computations involved?

Let's focus on one input channel first

$$u(l, j) = \sum_{i=0}^{F-1} \sum_{m=0}^{S-1} x(l+i, j+m)w(i, m) + b$$

Element-wise multiplication of input and weights

$$t_1 w_1 \\ t_2 w_2 \\ t_3 w_3 \\ t_4 w_4 \\ t_5 w_5$$

How many multiplication operations?
In general, there are $F^2 - 1$ addition operations, where F is the filter spatial size

$$t_1 w_1 \\ t_2 w_2 \\ t_3 w_3 \\ t_4 w_4 \\ t_5 w_5$$

How many padding operations?
Adding the elements after multiplication, we need $n - 1$ addition operations for n elements

$$t_1 w_1 \\ t_2 w_2 \\ t_3 w_3 \\ t_4 w_4 \\ t_5 w_5$$

Let's say we have D_1 input channels now

$$D_1 \times F^2 \text{ filter}$$

Element-wise multiplication of input and weights, for each channel

$$D_1 \times F^2 \text{ filter}$$

In general, there are $D_1 \times F^2 - 1$ addition operations, where D_1 is the number of input channels

$$D_1 \times F^2 \text{ filter}$$

If we have only one filter, and apply it to generate output at one spatial location, the total operations

$$D_1 \times F^2 \text{ filter}$$

The total cost is
 $(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$

$$D_1 \times F^2 \text{ filter}$$

If we have D_1 filter, and apply it to generate output at one spatial location, the total operations

$$D_1 \times F^2 \text{ filter}$$

The total cost is
 $((D_1 \times F^2) + (D_1 \times F^2 - 1) + 1) \times D_2 \times H_2 \times W_2$

$$= (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

Summary

FLOPs (floating point operations)

Not FLOPs (floating point operations per second)

Assume: Filter size F

Accepts a volume of size $D_1 \times H_1 \times W_1$

Produces an output volume of size $D_2 \times H_2 \times W_2$

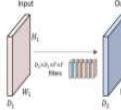
The FLOPs of the convolution layer is given by

$$\text{FLOPs} = ((D_1 \times F^2) + (D_1 \times F^2 - 1) + 1) \times D_2 \times H_2 \times W_2$$

Intermediate: Adding the results after each filter pass, we need $n - 1$ additions for n elements

Add the bias: We need D_1 additions for D_1 bias terms

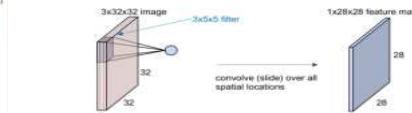
Final result: Add the results from all the filters



Standard convolution

Standard convolution

- The input and output are locally connected in spatial domain
- In **channel** domain, they are **fully connected**



Try this

$$\begin{aligned} & \frac{5}{3} \times 3 \\ & (2 \times 3 \times (\frac{5}{3})^2) \times 1 \times 1 \times 1 \times 28 = 11760 \end{aligned}$$

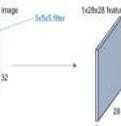
Assume:

One filter with spatial size of 5×5

Accepts a volume of size $3 \times 32 \times 32$

Produces an output volume of size $1 \times 28 \times 28$

The FLOPs of the convolution layer is given by



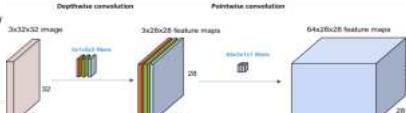
Standard convolution

Standard convolution

Standard convolution



Depthwise convolution + Pointwise convolution



Pointwise convolution

We have seen convolution with spatial size of $3 \times 3, 5 \times 5, 7 \times 7$

Can we have other sizes?

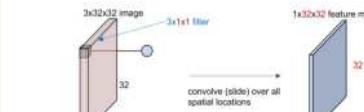
Can we have filter of spatial size 1×1 ?



We have seen convolution with spatial size of $3 \times 3, 5 \times 5, 7 \times 7$

Can we have other sizes?

Can we have filter of spatial size 1×1 ?



Why replacing standard convolution with depthwise convolution + pointwise convolution?

The computational cost reduction rate is roughly $1/8 - 1/9$ at only a small reduction in accuracy

Good if you want to deploy small networks on devices with CPU.

Used in network such as MobileNet

Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = (D_1 \times F^2) + (D_1 \times F^2 - 1) + 1 \times D_2 \times H_2 \times W_2$$

Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = (D_1 \times F^2 - 1) \times D_1 \times H_1 \times W_1$$

Each filter is applied only to one channel

The number of output channels result in the number of input channels

Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = (D_1 \times F^2) + (D_1 \times F^2 - 1) = 1 \times D_1 \times H_1 \times W_1$$

Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = (D_1 \times F^2) + (D_1 \times F^2 - 1) = 1 \times D_1 \times H_1 \times W_1$$

Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = (D_1 \times F^2 - 1) \times D_1 \times H_1 \times W_1$$

The bias operation is D_1

How much computation do you save by replacing standard convolution with depthwise convolution?

Reduction = $\frac{\text{Cost of depthwise convolution} + \text{pointwise convolution}}{\text{Cost of standard convolution}}$

$$\text{Reduction} = \frac{(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1}{(D_1 \times F^2) + (D_1 \times F^2 - 1)} = \frac{(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1}{(D_1 \times F^2) + (D_1 \times F^2 - 1)}$$

$$\text{Reduction} = \frac{1}{1} = 1$$

D_1 is usually large. Reduction can be as high as $(N - 1)(N - 2)/2$ multiply-add operations per pixel

More on convolution

Learn how to calculate computation complexity of convolutional layer

Parameter convolution can be used to change the use of this layer. This can be used to achieve channel reduction and thus saving computational cost

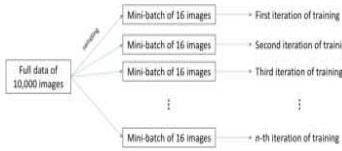
Depthwise convolution + Pointwise convolution costs fewer computations than standard convolution



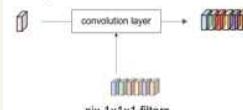
Proposed by Google in 2014, winning the ImageNet competition that year.
Apart from the inception structure, there is another important technique called batch normalization.
Problem: deep networks are very hard to train!
Main idea: "Normalize" the outputs of a layer so they have zero mean and unit variance.
Effect: allowing higher learning rates and reducing the strong dependence on initialization.

Mini-batch

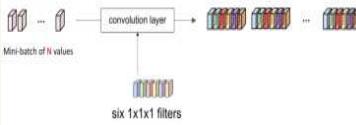
- What is mini-batch?
- A subset of all data during one iteration to compute the gradient



1x1x1 input



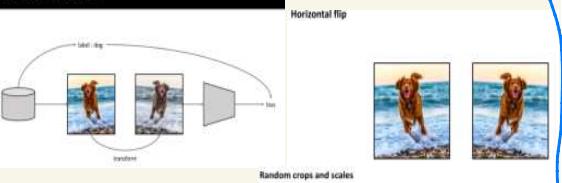
Nx1x1 input



Transfer learning is pervasive



Data augmentation



- Training:**
1. Pick random L in range [356, 480]
 2. Resize training image, short side = L
 3. Sample random 224 x 224 patch

There are many more data augmentation schemes.

Random mix/combinations of:

translation · rotation · stretching · shearing · lens distortions ...



Color jitter

Simple

1. Randomize contrast and brightness

Complex

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

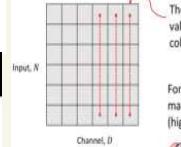
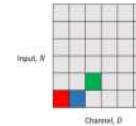
Used in AlexNet, ResNet, etc.

Batch Normalization

Let's arrange the activations of the mini batch in a matrix of $N \times D$

Nx6x1x1 activation/feature maps

Activation of the first row in the activations
Activation of the second row in the activations
Activation of the last row in the activations

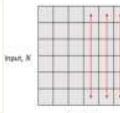


The goal of batch normalization is to normalize the values across each column so that the values of the column have zero mean and unit variance.

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below:



Batch Normalization - Training



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{ij}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{ij} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{ij} = \gamma_j x'_{ij} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization - Test Time



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{ij}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{ij} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{ij} = \gamma_j x'_{ij} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{ij} = \gamma_j x'_{ij} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{ij} = \gamma_j x'_{ij} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization

Batch Normalization for fully-connected networks

$$\mathbf{x}: N \times D$$

$$\mu, \sigma: 1 \times D$$

$$\gamma, \beta: 1 \times D$$

$$y = \gamma(\mathbf{x} - \mu) / \sigma + \beta$$

Normalize

• It is usually done after a fully connected/convolutional layer and before a non-linearity layer

• Zero overhead at test-time: can be fused with conv

• Allows higher learning rates, faster convergence

• Networks becomes more robust to initialization

Batch Normalization for convolutional networks (Spatial Batchnorm, BatchNorm2D)

$$\mathbf{x}: N \times C \times H \times W$$

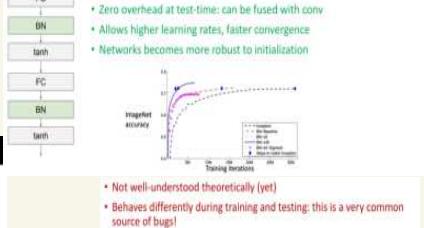
$$\mu, \sigma: 1 \times C \times 1 \times 1$$

$$\gamma, \beta: 1 \times C \times 1 \times 1$$

$$y = \gamma(\mathbf{x} - \mu) / \sigma + \beta$$

Normalize

• Normalize also on the spatial dimensions



prevent overfitting

Why overfitting?

- This happens when our model is too complex and too specialized on a small number of training data.
- Increase the size of the data, remove outliers in data, reduce the complexity of the model, reduce the feature dimension

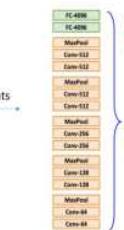
Transfer learning



In a network with an N -dimensional softmax output layer that has been successfully trained toward a supervised classification objective, each output unit will be **specific** to a particular class

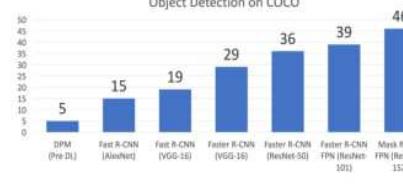
Pre-training + Fine-tuning

- Step 1: Pre-training on large-scale dataset like ImageNet
- Step 2: Use pre-trained network as initialization
- Step 3: Fine-tuning



Add new layer correspond to the target class number
Train on new data

Architecture matters



8. Recurrent Neural Networks (RNN)

Last week

- CNN Architectures
- More on convolution
 - How to calculate FLOPs
 - Poatwise convolution
 - Depthwise convolution
 - Depthwise convolution + Poatwise convolution
- Batch normalization
- Prevent overfitting
 - Transfer learning
 - Data augmentation

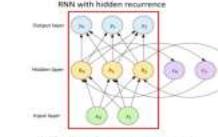
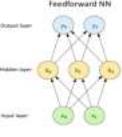
You learn some classic architectures.

You learn how to calculate interpretation complexity of convolutional layer and how to design a lightweight network.

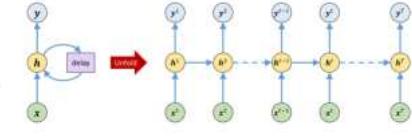
You learn an important technique to improve the training of modern neural networks.

You learn few important techniques to prevent overfitting in neural networks.

Recurrent Neural Networks (RNN)



RNN with hidden recurrence



The recurrent connections in the hidden-layer can be unfolded to process sequences of arbitrary length.

By considering the unfolded structure, $h(t)$ is dependent on all the inputs at time t and before time t :

$$h(t) = f^t(x(t), x(t-1), \dots, x(2), x(1))$$

The function f^t takes the whole past sequence $(x(t), x(t-1), \dots, x(2), x(1))$ as input and produce the hidden layer activation.

Not an efficient way! The function f^t is dependent to the sequence length.

Let's represent the past inputs by the hidden-layer activations in the previous instant.

$$h(t) = f^t(x(t), x(t-1), \dots, x(2), x(1))$$

$h(t-1)$



The recurrent structure allows us to factorize f^t into repeated applications of a function f . We can write:

$$\begin{aligned} \text{new state } h(t) &= f(h(t-1), x(t)) \\ \text{some function } f &\text{ old state } h(t-1) \\ &\text{input vector at } x(t) \end{aligned}$$

The folded structure introduces two major advantages:

$$\begin{aligned} \text{new state } h(t) &= f(h(t-1), x(t)) \\ \text{some function } f &\text{ old state } h(t-1) \\ &\text{input vector at } x(t) \end{aligned}$$

1. Regardless of the sequence length, the learned model always has the same size, rather than specified in terms of a variable-length history of states.

2. It is possible to use same transition function f with the same parameters at every time step.

U : weight vector that transforms raw inputs to the hidden-layer

W : recurrent weight vector connecting previous hidden-layer output to hidden input

V : weight vector of the output layer

b : bias connected to hidden layer

c : bias connected to the output layer

ϕ : the tanh hidden-layer activation function
 σ : the linear/softmax output-layer activation function

Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Elman-type RNN with one hidden-layer is given by:

$$\begin{aligned} h(t) &= \phi(U^T x(t) + W^T h(t-1) + b) \\ y(t) &= \sigma(V^T h(t) + c) \end{aligned}$$

σ is a softmax function for classification and a linear function for regression.

RNN with hidden recurrence: batch processing

Given P patterns $\{x_p\}_{p=1}^P$ where $x_p = (x_p(t))_{t=1}^T$,

$$X(t) = \begin{pmatrix} x_1(t)^T \\ x_2(t)^T \\ \vdots \\ x_P(t)^T \end{pmatrix}$$

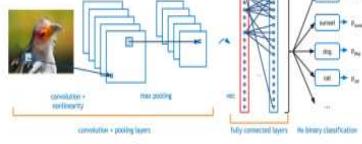
Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t .

Activation of the three-layer Elman-type RNN is given by:

$$H(t) = \phi(X(t)U + H(t-1)W + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Previous: Convolutional Neural Networks (CNN)



How about sequential information?

- Turn a sequence of sound pressures into a sequence of word identities?
- Speech recognition?
- Video prediction?

Outline

- Recurrent Neural Network (RNN)
 - Hidden recurrence
 - Top-down recurrence
- Long Short-Term Memory (LSTM)
 - Long-term dependency
 - Structure of LSTM
- Example Applications

Recurrent Neural Networks (RNN)

Recurrent neural networks (RNN) are designed to process **sequential information**. That is, the data presented in a sequence.

The next data point in the sequence is usually **dependent** on the current data point.

Examples:

- Natural language processing (spoken words and written sentences). The next word in a sentence depends on the word which comes before it.
- Genomic sequences: a nucleotide in a DNA sequence is dependent on its neighbors.

RNN attempts to **capture dependency** among the data points in the sequence.

Text Generation with an RNN

QUEENE:

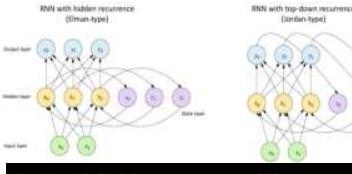
I had thought thou hadst a Roman; for the oracle,
 Thus by All bids the man against the word,
 Which are so weak of care, by old care done;
 Your children were in your holy love,
 And the precipitation through the bleeding throne.

in google
 search
 image
 captions

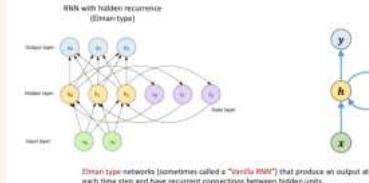
BISHOP OF ELY:

Marry, and will, my lord, to weep in such a one were pretties;
 Yet now I was adopted heir
 Of the world's lamentable day,
 To watch the next way with his father with his face?

Types of RNN



RNN with hidden recurrence (Elman type)



Elman-type networks (sometimes called "classic RNN") that produce an output at each time step and have recurrent connections between hidden units.

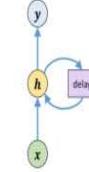
3. The hidden-layer activation at time $t-1$ is kept by the delay unit and fed to the hidden layer at time t together with the raw input $x(t)$.

4. The delay unit represents that the activation is held for one time unit until the next time instance.

Here, one unit represents the time between two adjacent data points in the sequence.

1. Data is presented as a sequence of instance t [time] that is discretized

2. Activations are updated at each time instant t .



initial hidden state
 Either set it to all zero or learn it.

Example 1

A recurrent neural network with hidden recurrence has two input neurons, three hidden neurons, and two output neurons. The parameters of the network are initialized as $U = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \end{pmatrix}$, $W = \begin{pmatrix} 0.5 & 0.0 & -0.2 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix}$, and $V = \begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}$. Bias to the hidden layer $b = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$ and to the output layer $c = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$.

For a sequence of inputs $(x(1), x(2), x(3), x(4))$ where

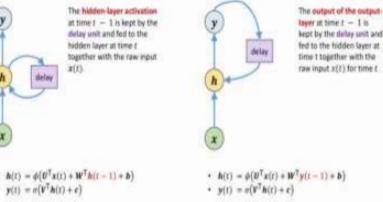
$$x(1) = \begin{pmatrix} 0.2 \\ 1.2 \end{pmatrix}, x(2) = \begin{pmatrix} 0.2 \\ 1 \end{pmatrix}, x(3) = \begin{pmatrix} 0 \\ 3 \end{pmatrix}, \text{ and } x(4) = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

find the output of RNN.

Assume that hidden layer activations are initialized to zero and tanh and sigmoid functions for the hidden and output layer activation functions, respectively.

RNN with top-down recurrence (Jordan type)

$$\text{At } t=1 : X(1) = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}$$



$$\begin{aligned} H(1) &= \tanh(X(1)U + Y(0)W + B) \\ &= \tanh\left(\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}\begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \end{pmatrix} + \begin{pmatrix} 0.2 & 0.2 & 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.2 \\ 0.72 \\ -1.0 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} Y(1) &= \text{sigmoid}(H(1)V + C) \\ &= \text{sigmoid}\left(\begin{pmatrix} 0.2 & 0.72 & -1.0 \end{pmatrix}\begin{pmatrix} 2.0 \\ -1.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.31 \\ 0.90 \end{pmatrix} \end{aligned}$$

$$\text{At } t=2 : X(2) = \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}$$

$$\begin{aligned} H(2) &= \tanh(X(2)U + Y(1)W + B) \\ &= \tanh\left(\begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.31 \\ 0.9 \end{pmatrix}\begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \end{pmatrix} + \begin{pmatrix} 0.2 & 0.2 & 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} Y(2) &= \text{sigmoid}(H(2)V + C) \\ &= \text{sigmoid}\left(\begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix}\begin{pmatrix} 2.0 \\ -1.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix} \end{aligned}$$

$$\text{At } t=3 : X(3) = \begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}$$

$$\begin{aligned} H(3) &= \tanh(X(3)U + Y(2)W + B) \\ &= \tanh\left(\begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix}\begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \end{pmatrix} + \begin{pmatrix} 0.2 & 0.2 & 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.00 & 0.94 & 0.99 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} Y(3) &= \text{sigmoid}(H(3)V + C) \\ &= \text{sigmoid}\left(\begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.00 & 0.94 & 0.99 \end{pmatrix}\begin{pmatrix} 2.0 \\ -1.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.63 \\ 0.04 \end{pmatrix} \end{aligned}$$

$$\text{Output (batch): } \mathbf{Y} = (\mathbf{Y}(1), \mathbf{Y}(2), \mathbf{Y}(3)) = \begin{pmatrix} 0.31 \\ 0.11 \\ 0.04 \end{pmatrix}$$

Outputs for inputs

$$x_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad x_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$y_1 = (0.31, 0.83, 0.63), \quad y_2 = (0.9, 0.11, 0.04)$$

Backpropagation through time (BPTT)

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

$$\begin{aligned} h(t) &= \tanh(U^T x(t) + \dots + U^T x(1) + b) \\ u(t) &= V^T h(t) + c \end{aligned}$$

The gradients propagate backward, starting at the end of the sequence from two directions:

- Top-down direction
- Reverse sequence direction:

For classification,

$$y(t) = \text{softmax}(u(t))$$

For regression,

$$y(t) = u(t)$$

Learnable parameters

- W : weight vector that transforms raw inputs to the hidden layer
- W' : recurrent weight vector connecting previous hidden-layer output to hidden input
- V : weight vector of the output layer
- b : bias connected to hidden layer
- c : bias to the output layer

The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph, followed by a backpropagation pass moving right to left through the graph. The gradients are propagated from the final time point to the initial time points.

The runtime is $O(T)$ where T is the length of input sequence and cannot be reduced by parallelization because the forward propagation is inherently sequential. The back-propagation algorithm applied to the unrolled graph with $O(T)$ cost is called back-propagation through time (BPTT).

The network with recurrence between hidden units is thus very powerful but also expensive to train.

RNN with top-down recurrence: batch processing

Given P patterns $\{x_p\}_{p=1}^P$ where $x_p = (x_p(t))_{t=1}^T$,

$$X(t) = \begin{pmatrix} x_1(t)^T \\ x_2(t)^T \\ \vdots \\ x_P(t)^T \end{pmatrix}$$

Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Jordan-type RNN is given by:

$$H(t) = \phi(X(t)U + Y(t-1)W + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Example 2

A recurrent neural network with top-down recurrence receives 2-dimensional input sequences and produce 1-dimensional output sequences. It has three hidden neurons and following weight matrices and biases:

$$\text{Weight matrix connecting input to hidden layer: } U = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \end{pmatrix}$$

$$\text{Recurrence weight matrix connecting previous output to the hidden layer: } W = \begin{pmatrix} 2.0 & -1.5 \\ -1.5 & 0.2 \end{pmatrix}$$

$$\text{Weight matrix connecting hidden layer to the output: } V = \begin{pmatrix} 2.0 \\ -1.5 \end{pmatrix}$$

$$\text{Bias to the hidden layer: } b = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$$

$$\text{and bias to the output layer: } c = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}$$

Given the following two input sequences:

$$x_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad x_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

Using batch processing, find output sequences. Assume outputs are initialized to zero at the beginning. Assume tanh and sigmoid activations for hidden and output layer, respectively.

$$x_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad x_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

Two sequences as batch of sequences:

$$X = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}$$

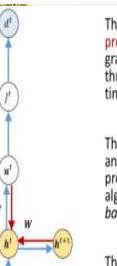
Three hidden neurons and one output neuron.

$$H(t) = \phi(X(t)U + Y(t-1)W + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Initially,

$$Y(0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$



Backpropagation through time (BPTT)

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

$$\begin{aligned} h(t) &= \tanh(U^T x(t) + W^T h(t-1) + b) \\ u(t) &= V^T h(t) + c \end{aligned}$$

For classification,

$$y(t) = \text{softmax}(u(t))$$

For regression,

$$y(t) = u(t)$$



- Learnable parameters**
- U: weight vector that transforms raw inputs to the hidden-layer
 - W: recurrent weight vector connecting previous hidden-layer output to hidden input
 - V: weight vector of the output layer
 - b: bias connected to hidden layer
 - c: bias connected to output layer

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

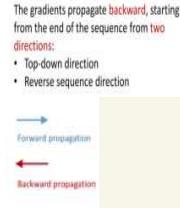
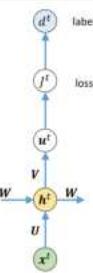
$$\begin{aligned} h(t) &= \tanh(U^T x(t) + W^T h(t-1) + b) \\ u(t) &= V^T h(t) + c \end{aligned}$$

For classification,

$$y(t) = \text{softmax}(u(t))$$

For regression,

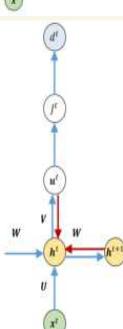
$$y(t) = u(t)$$



The gradient computation involves performing a **forward propagation** pass moving left to right through the unfolded graph, followed by a **backward propagation** pass moving right to left through the graph. The gradients are propagated from the final time point to the initial time points.

The runtime is $O(T)$ where T is the length of input sequence and cannot be reduced by parallelization because the forward propagation is inherently **sequential**. The back-propagation algorithm applied to the unrolled graph with $O(T)$ cost is called **back-propagation through time (BPTT)**.

The network with recurrence between hidden units is thus very powerful but also **expensive** to train.



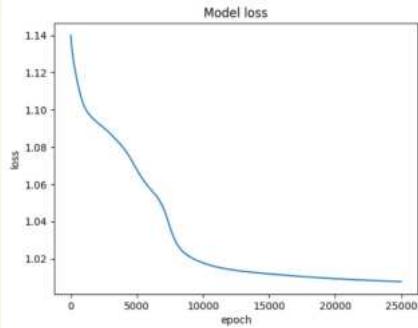
Example 3 eg3.ipynb

Generate 8-dimensional 16 input sequences of 64 time-steps with each input is a random number between [0.0, 1.0].

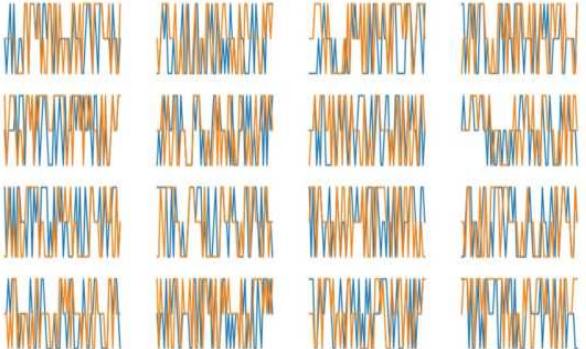
Generate the corresponding 1-dimensional labels by randomly generating a number from [0, 1, 2].

Create an RNN with one hidden layer of 5 neurons.

Plot the learning curves and predicted labels.



Prediction (orange) vs Groundtruth (blue)



Long Short-term Memory (LSTM)

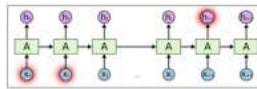
Long-term dependency

"The man who ate my pizza has purple hair"

Purple hair is for the man and not the pizza!
This is a long term dependency.

There are cases where we need even more context:

- To predict last word in "I grew up in France...long paragraph...I speak French"
- Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is French
- RNNs work upon the fact that the result of an information is dependent on its previous state or previous n time steps
- RNNs have difficulty in learning long range dependencies
- Gap between relevant information and where it is needed is large



Exploding and vanishing gradients in RNN

During gradient back-propagation learning of RNN, the gradient can end up being multiplied a large number of times (as many as the number of time steps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer.

Note that each time the activations are forward propagated in time, the activations are multiplied by W and each time the gradients are back propagated, the gradients are multiplied by W^T .

$$\nabla_{W(t)} = W \text{diag}(1 - h^2(t+1)) \nabla_{W(t+1)} + V^T \nabla_{V(t)}$$

Gradient from reverse direction

If the weights in this matrix are small, the recursive derivative can lead to a situation called **vanishing gradients** where the gradient signal gets so small that learning either becomes very slow or stops working altogether



Contribution from the earlier steps becomes insignificant in the gradients.

Conversely, if the weights in this matrix are large, it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as **exploding gradients**.

Exploding gradient easily solved by clipping the gradients at a predefined threshold value.

Vanishing gradient is more concerning

Gradient Clipping



Due to long term dependencies, RNN tend to have gradients having very large or very small magnitudes. The large gradients resemble cliffs in the error landscape and when the gradient descent encounters the gradient updates can move the parameters away from true minimum.

A gradient clipping is employed to avoid the gradients to become too large.

Commonly, two methods are used:

1. Clip the gradient g when it exceeds a threshold:

$$\|g\| > v: \|g\| \leftarrow \frac{v}{\|g\|} v$$

2. Normalize the gradient when it exceeds a threshold:

$$\|g\| > v: \|g\| \leftarrow \frac{v}{\|g\|} v$$

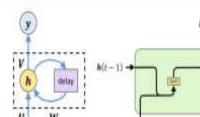
Basic RNN unit

The RNN cell (memory unit) is characterized by

$$h(t) = \phi(W^T x(t) + W^T h(t-1) + b)$$

where ϕ is the tanh activation function and RNN cell is referred to as tanh units.

RNN build with simple tanh units are also referred to as **vanilla RNN**.

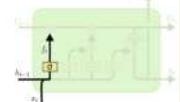


Forget gate

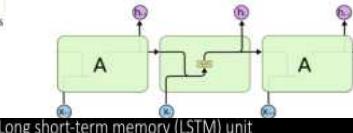
The forget gate can modulate the memory cell's self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.

$$f(t) = \sigma(W_f^T x(t) + W_f^T h(t-1) + b_f)$$

Value of $f(t)$ determines if $c(t-1)$ is to be remembered or not.



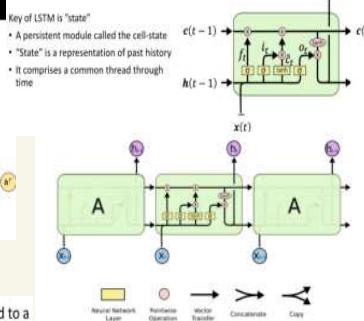
Basic RNN layer



Long short-term memory (LSTM) unit

Key of LSTM is "state"

- A persistent module called the cell-state
- "State" is a representation of past history
- It comprises a common thread through time



Cells are connected recurrently to each other - Replacing hidden units of ordinary recurrent networks

LSTMs provide a solution by incorporating memory units that allow the network to learn when to **forget previous hidden states** and when to **update hidden states** given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.

The key to LSTM is the **cell state** $c(t)$ - the horizontal line through the top of the diagram.

The long-term memory.

Like a conveyor belt that runs through entire chain with minor interactions

The LSTM has the ability to add and remove information to the cell states through gates.

Gates are a way to optionally let information through.

They are composed of sigmoid neural net layer and pointwise multiplication operations.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.

A value of zero means "let nothing through," while a value of one means "let everything through."



Input gate

The input gate can allow incoming signal to alter the state of the memory cell or block it. It decides what new information to store in the cell stage.

This has two parts:

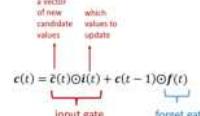
A sigmoid input gate layer decides which values to update;

A tanh layer creates a vector of new candidate values $\tilde{c}(t)$ that could be added to the state.

$$i(t) = \sigma(W_i^T x(t) + W_i^T h(t-1) + b_i)$$

$$\tilde{c}(t) = \phi(W_t^T x(t) + W_t^T h(t-1) + b_t)$$

Cell state



Example: Language modeling

In the language model, this is where we'll actually drop the information about the old subject's gender and add the new information, as we decided in previous steps.

Output gate

The output gate can allow the state of the memory cell to have an effect on other neurons or prevent it.

$$o(t) = \sigma(W_o^T x(t) + W_o^T h(t-1) + b_o)$$

$$h(t) = \phi(c(t)) \odot o(t)$$

LSTM unit

~~test control~~
~~not calculation~~

$$l(t) = \sigma(W_l^T x(t) + W_l^T h(t-1) + b_l)$$

$$f(t) = \sigma(W_f^T x(t) + W_f^T h(t-1) + b_f)$$

$$a(t) = \sigma(W_a^T x(t) + W_a^T h(t-1) + b_a)$$

$$c(t) = \tilde{c}(t) \odot l(t) + c(t-1) \odot f(t)$$

$$h(t) = \phi(c(t)) \odot o(t)$$

LSTM introduces a gated cell where the information flow can be controlled.

The most important component is the state unit $c_i(t)$ (for time step t and cell i) which has a linear self-loop.

The self-loop weight is controlled by a forget gate unit, which sets this weight to a value between 0 and 1 via a sigmoid unit.

Clever idea!

The self-loop inside the cell is able to produce paths where the gradient can flow for long duration and to make the weight on the time scale of integration can be changed dynamically based on the input sequence.

In this way, LSTM will preserve error terms that can be propagated through many layers and time steps.

Output gate

Example 4

The output gate can allow the state of the memory cell to have an effect on other neurons or prevent it.

$$o(t) = \sigma(U_o^T x(t) + W_o^T h(t-1) + b_o)$$

$$h(t) = \phi(c(t)) \odot o(t)$$

LSTM unit

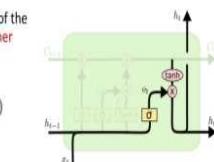
$$i(t) = \sigma(U_i^T x(t) + W_i^T h(t-1) + b_i)$$

$$f(t) = \sigma(U_f^T x(t) + W_f^T h(t-1) + b_f)$$

$$o(t) = \sigma(U_o^T x(t) + W_o^T h(t-1) + b_o)$$

$$c(t) = \phi(U_c^T x(t) + W_c^T h(t-1) + b_c)$$

$$h(t) = \phi(c(t)) \odot o(t)$$



Generate 64 2-dimensional input sequences $(x(t))_{t=1}^{16}$, where $(x_1(t), x_2(t)) \in [0, 1]^2$ by randomly generating numbers uniformly.

Generate 1-dimensional output sequences $(y(t))_{t=1}^{16}$ where $y(t) \in R$ by following the following recurrent relation:

$$y(t) = 5x_1(t-1)x_2(t-2) - 2x_1(t-7) + 3.5x_2^2(t-5) + 0.1e$$

where $e \sim N(0, 1)$.

Train a LSTM layer to learn the mapping between input and output sequences.

Use a learning factor $\alpha = 0.001$.

Repeat the above using RNN and GRU and compare the performances.

Sentiment classification

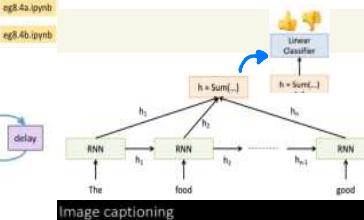
"The food was really good"

"The chicken crossed the road because it was uncooked"

- Classify a restaurant review from Yelp! OR movie review from IMDB OR ...

as positive or negative

- Inputs: Multiple words, one or more sentences
- Outputs: Positive / Negative classification



- Given an image, produce a sentence describing its contents
- Inputs: Image feature from a CNN
- Outputs: Multiple words (let's consider one sentence)



LSTM introduces a gated cell where the information flow can be controlled.

The most important component is the state unit $c_i(t)$ (for time step i and cell i) which has a linear self-loop.

The self-loop weight is controlled by a forget gate unit, which sets this weight to a value between 0 and 1 via a sigmoid unit.

Clever idea!

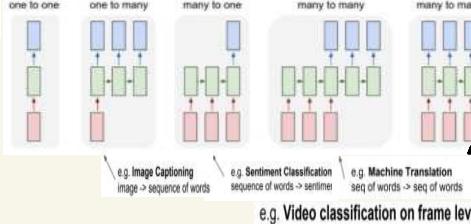
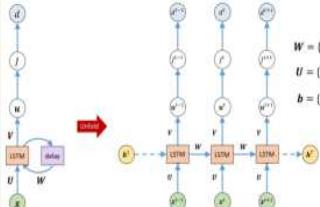
The self-loop inside the cell is able to produce paths where the gradient can flow for long duration and to make the weight on this self-loop conditioned on the context rather than fixed.

By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically based on the input sequence.

In this way, LSTM help preserve error terms that can be propagated through many layers and time steps.

Training

Training LSTM networks



Text classification: Dbpedia dataset

This dataset contains first paragraph of Wikipedia page of 56000 entities and label them as one of 15 categories like people, company, schools, etc.

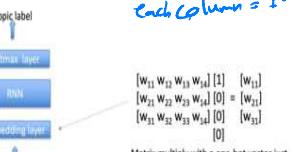


An input is a text. Requires to find all words in the text and remap them into word IDs, a number per each unit word (word-level inference).

my work is cool! $\Rightarrow [23, 500, 5, 1402, 17]$

We need to make sure that each sentence is of same length (no_words). The maximum document length is fixed and longer sentences will be truncated and shorter once are padded with zeros.

Text classification



Each column = 1 word

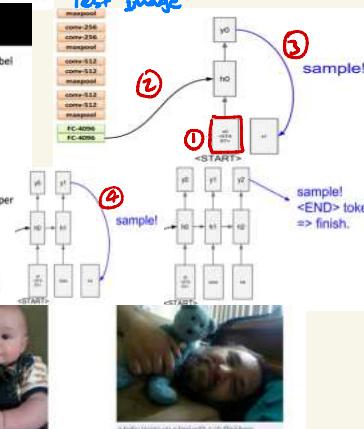
my work is cool! $\Rightarrow [23, 500, 5, 1402, 17]$

We need to make sure that each sentence is of same length (no_words). The maximum document length is fixed and longer sentences will be truncated and shorter once are padded with zeros.

The embedding layer converts one-hot vector of word representations (of size of the vocabulary) to a vector of fixed length (embedding_size vectors).

Embedding layer learns a weight matrix of $|embedding_size| \times |vocab_size|$

And then maps word indexes of the sequences into $[batch_size, sequence_length, embedding_size]$ input to the RNN.



9. Attention

Outline

- Attention
- Transformers
- Vision Transformers

Attention

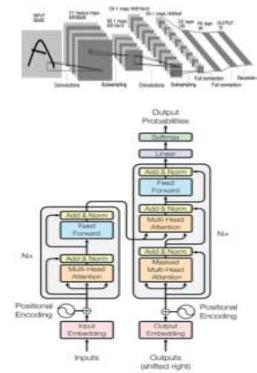
Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified by learned weights** and thus the output is usually formed as a **weighted average**.

Transformer

Background

- Convolutional neural networks (CNN) has been dominating
 - Greater scale
 - More extensive connections
 - More sophisticated forms of convolution



Ashish Vaswani et al., [Attention Is All You Need](#), NIPS 2017 (from Google)

→ GPT-4

Figure 1: The Transformer - model architecture.



What is the best way to learn knowledge about Transformers (a machine learning model)?



Transformers are an important concept in modern machine learning, especially in the field of natural language processing. They are the key model behind the technology in GPT-3 and GPT-4, BERT, T5, and many other influential architectures.

Here is a suggested path for learning about Transformers:

1. Understand the Basics of Machine Learning and Deep Learning

Before you get into Transformers, it's crucial to have a solid understanding of Machine Learning and Deep Learning. Knowing the fundamental concepts of Deep Learning such as neural networks, activation functions, loss functions, backpropagation, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Long Short Term Memory (LSTM) units is very helpful.

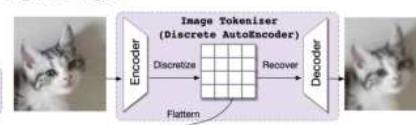


Input Text: Input Image:

(The head of a lovely cat.)
一只可爱的 小猫的 头像。

Text Tokenizer (sentence pieces)

一只 可爱 的 小 猫 的 头 像



Text tokens, ranging from 8192 to 58192.

1024 Image tokens, ranging from 0 to 8192.

Transformer (GPT)

Transformers

Notable for its use of **attention** to model long-range dependencies in data



A sequence-to-sequence model

Model of choice in natural language processing (NLP)

Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder)

But it differs from the existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).

- During training, layer outputs can be calculated in parallel, instead of a series like an RNN
- Attention-based models allow modeling of dependencies without regard to their distance in the input or output sequences

Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified** by learned weights and thus the output is usually formed as a weighted average.

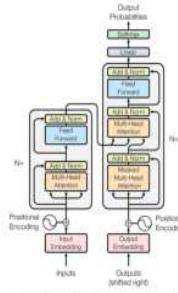


Figure 1: The Transformer - model architecture

Input Text:

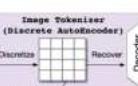
(The head of a lovely cat.)

一只可爱的 小猫的 头像

Input Image:



Image Tokenizer



Discretize
Recover
Flatten
[END]

1024 Image tokens, ranging from 0 to 8192.

Transformer (GPT)

Output: I am a student

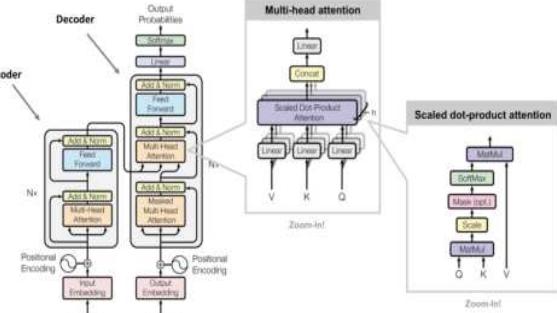
Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified** by learned weights and thus the output is usually formed as a weighted average.

Attention = WV



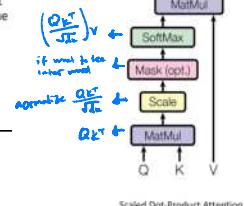
The model attends to image regions that are semantically relevant for classification



Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

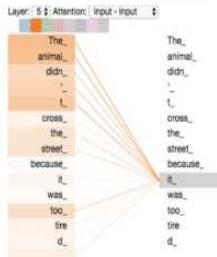
$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



It is entirely built on the **self-attention**

mechanisms without using sequence-aligned recurrent architecture

Self-attention is a type of attention mechanism where the model makes prediction for one part of a data sample using other parts of the observation about the same sample.



What does "it" in this sentence refer to? Is it referring to the street or to the animal?

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".

The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word

The outputs of the self-attention layer are fed to a feed-forward neural network.

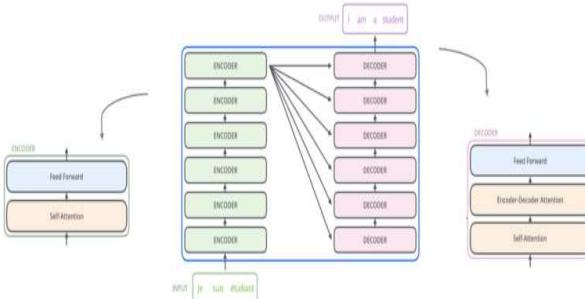
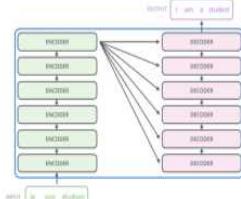
The exact same feed-forward network is independently applied to each position (each word/token).

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence.

The encoding component is a stack of encoders



The decoding component is a stack of decoders of the same number



Self-Attention in Detail

Matrix Calculation of Self-Attention

First Step

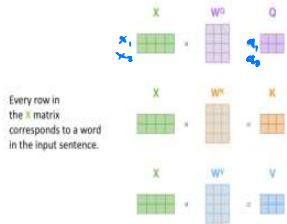
Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).



First Step

Calculate the Query, Key, and Value matrices.

Pack our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V)



So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

What are the "query", "key", and "value" vectors?

The names "query", "key" are inherited from the field of **information retrieval**

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{w_{ij}}{\sqrt{d_k}}$ depend on the relative similarities between the i -th query and all of the keys

The softmax function means that the **key** vectors "compete" with one another to contribute to the final result.

$$d_K = \text{key vector (k)} \\ \text{dimension} (-o)$$

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.

Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)

Fourth Step

Softmax for normalization

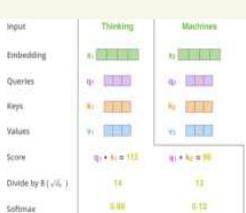
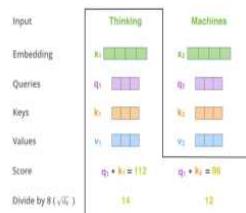
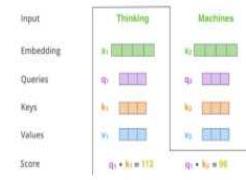
$$z_1 = q_1 \cdot k_1 / \sqrt{d_k} = 112 / \sqrt{64} = 112/8 = 14, z_2 = q_1 \cdot k_2 / \sqrt{d_k} = 96 / \sqrt{64} = 96/8 = 12 \\ \text{softmax}(z_1) = \exp(z_1) / \sum_{i=1}^2 (\exp(z_i)) = 0.88, \text{softmax}(z_2) = \exp(z_2) / \sum_{i=1}^2 (\exp(z_i)) = 0.12$$

Fifth Step

Multiply each value vector by the softmax score

Sixth Step

Sum up the weighted value vectors to get the output of the self-attention layer at this position (for the first word)



$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

First Step

Calculate the Query, Key, and Value matrices.

Pack our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V)

Every row in the X matrix corresponds to a word in the input sentence.

Second Step

Calculate the outputs of the self-attention layer. SoftMax is **row-wise**

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Example

Assuming we have two sequences:
 $(1, 2, 3)$
 $(4, 5, 6)$

And the given W^Q, W^K, W^V matrices are respectively given as
 $\begin{pmatrix} 0.01 & 0.03 \end{pmatrix}, \begin{pmatrix} 0.05 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.02 & 0.02 \end{pmatrix}$
 $\begin{pmatrix} 0.02 & 0.02 \end{pmatrix}, \begin{pmatrix} 0.06 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.01 & 0.02 \end{pmatrix}$
 $\begin{pmatrix} 0.03 & 0.01 \end{pmatrix}, \begin{pmatrix} 0.07 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.01 & 0.01 \end{pmatrix}$

Compute the output of the scaled-dot product attention, $\text{Attention}(Q, K, V)$.

Step 1: Find Q, K, V

$$Q = XW^Q = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix}$$

$$K = XW^K = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.5 & 0.5 \\ 0.6 & 0.5 \\ 0.7 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.38 & 0.30 \\ 0.92 & 0.75 \end{pmatrix}$$

$$V = XW^V = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix}$$

Step 2: Find $\text{Attention}(Q, K, V)$

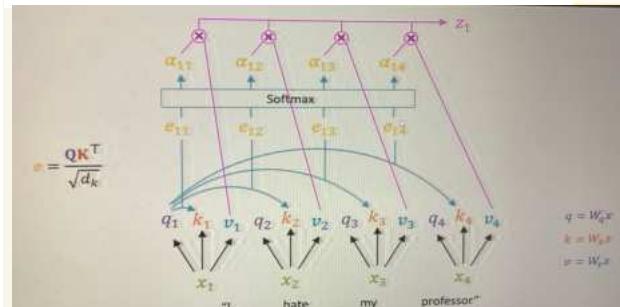
$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$\text{Perform row-wise SoftMax}$$

$$\frac{e^{x_j}}{\sum_i e^{x_i}} = \frac{e^{0.588}}{e^{0.0588} + e^{0.1441}} = 0.4787$$

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) = \text{Softmax}\left(\frac{(0.14 \cdot 0.10)(0.38 \cdot 0.92)}{\sqrt{2}}\right) = \text{Softmax}\left(0.0588 \cdot 0.1441\right) = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix}$$

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix} \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix} = \begin{pmatrix} 0.1326 & 0.1682 \\ 0.1363 & 0.1729 \end{pmatrix}$$



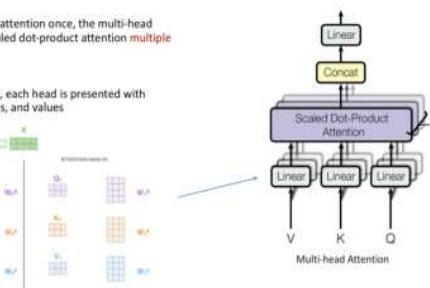
eg9.1.ipynb

Multi-Head Self-Attention

Multi-Head Self-Attention

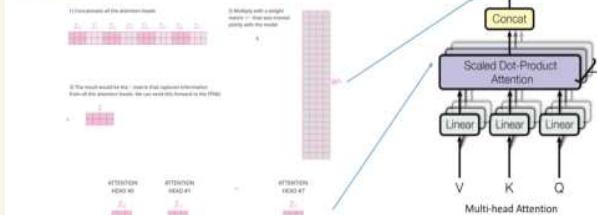
Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention **multiple times in parallel**.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values.



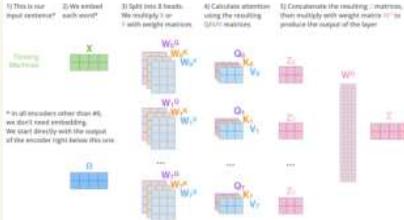
Multi-Head Self-Attention

The independent attention outputs are simply **concatenated** and linearly transformed into the expected dimensions.



Multi-Head Self-Attention

The big picture. Note that after the split each head can have a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.



Why?

"Multi-head attention allows the model to jointly attend to information from different representation **subspaces** at different positions."

SCALE UP MODEL

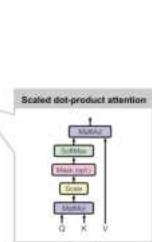
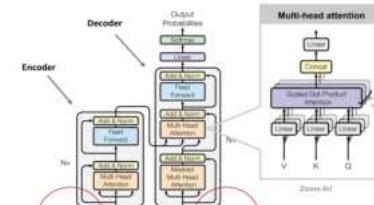
An Intuitive Example

Given a sentence:

"Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning."

Given "representation learning," the first head attends to "Deep learning," while the second head attends to the more general term "machine learning methods".

Positional Encoding



Self-attention layer works on sets of vectors and it doesn't know the order of the vectors it is processing

The positional encoding has the **same dimension** as the input embedding

Adds a vector to each input embedding to give information about the **relative or absolute position** of the tokens in the sequence

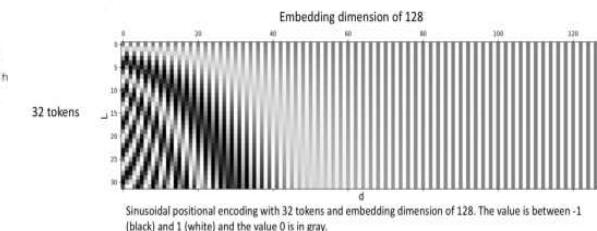
These vectors follow a specific pattern

What might this pattern look like?

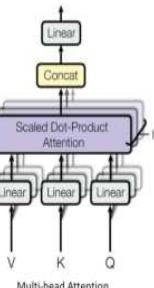
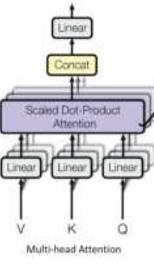
Each row corresponds to the positional encoding of a vector.

The first row would be the vector we'd add to the embedding of the first word in an input sequence.

Each position is uniquely encoded and the encoding can deal with sequences longer than any sequence seen in the training time.



Sinusoidal positional encoding with 32 tokens and embedding dimension of 128. The value is between -1 (black) and 1 (white) and the value 0 is gray.



Sinusoidal positional encoding - interweaves the two signals (sine for even indices and cosine for odd indices)

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

where pos is the position and i is the dimension, $[0, \dots, d_{model}/2]$

Example: Positional Encoding

Example:

Given the following **Sinusoidal positional encoding**, calculate the $PE(pos=1)$ for the first five dimensions $[0, 1, 2, 3, 4]$. Assume $d_{model} = 512$

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Solution:

Given $pos = 1$ and $d_{model} = 512$

$$At\ dimension\ 0, 2i = 0\ thus\ i = 0, therefore\ PE_{(pos,2i)} = PE_{(1,0)} = \sin(1/10000^0/512)$$

$$At\ dimension\ 1, 2i + 1 = 1\ thus\ i = 0, therefore\ PE_{(pos,2i+1)} = PE_{(1,1)} = \cos(1/10000^0/512)$$

$$At\ dimension\ 2, 2i = 2\ thus\ i = 1, therefore\ PE_{(pos,2i)} = PE_{(1,2)} = \sin(1/10000^2/512)$$

$$At\ dimension\ 3, 2i + 1 = 3\ thus\ i = 1, therefore\ PE_{(pos,2i+1)} = PE_{(1,3)} = \cos(1/10000^2/512)$$

$$At\ dimension\ 4, 2i = 4\ thus\ i = 2, therefore\ PE_{(pos,2i)} = PE_{(1,4)} = \sin(1/10000^4/512)$$

Implementation

The constructor initializes the module and sets up the dropout layer with the given dropout probability.

position is a tensor containing integers from 0 to `max_len-1`, representing each position in the sequence.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Arguments:
            x: Tensor, shape "[seq_len, batch_size, embedding_dim]"
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

The sinusoidal functions (sine for even indices and cosine for odd indices) are applied to the positions and the results are stored in `pe`. This creates a unique positional encoding for each position.

`max_len` is the maximum expected length of the sequences.

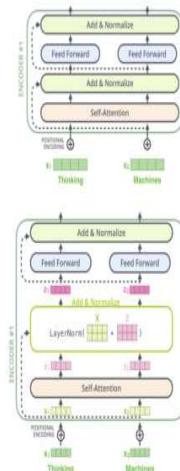
`div_term` is a scaling term used to adjust the rate of the sinusoidal functions.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

The positional encoding (`self.pe`) corresponding to the length of the input sequence is added to the input tensor `x`.

Transformer Encoder



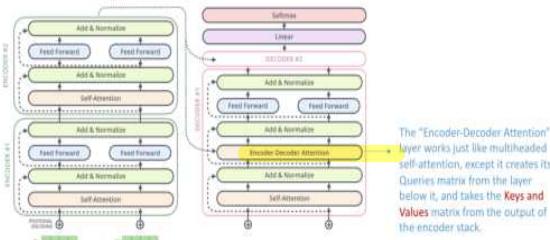
Encoder

If we're to visualize the vectors and the layer norm operation associated with self-attention, it would look like this:

- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.
- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

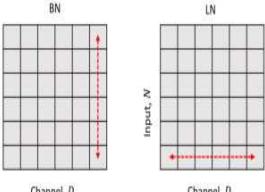
Transformer Decoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



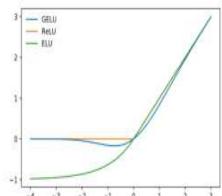
How encoder and decoder work together

- The encoder starts by processing the input sequence.
- The output of the top encoder is then transformed into a set of attention vectors K and V .
- These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence.



Layer Normalization (LN)¹

- The pixels along the red arrow are normalized by the same mean and variance, computed by aggregating the values of these pixels.
- BN is found unstable in Transformers²
- Works well with RNN and now being used in Transformers



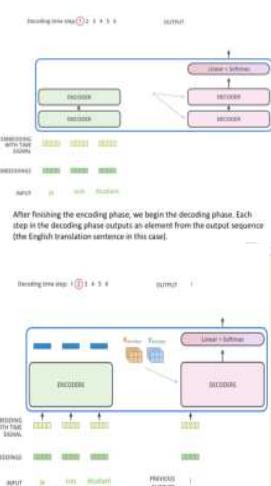
Gaussian Error Linear Units (GELU)³

- Can be thought of as a smoother ReLU
- Used in GPT-3, BERT, and most other Transformers

How encoder and decoder work together

- The output of each step is fed to the bottom decoder in the next time step.
- Embed and add positional encoding to those decoder inputs. Process the inputs.
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.

Note:
In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.



Vision Transformer (ViT)

- Do not have decoder

- Reshape the image $X \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
- (H, W) is the resolution of the original image
- C is the number of channels
- (P, P) is the resolution of each image patch
- $N = HW/P^2$ is the resulting number of patches

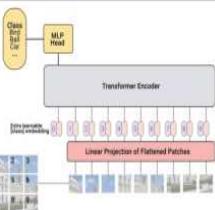


image \rightarrow word seqs

- Prepend a **learnable embedding** ($\mathbf{z}_0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_1^1 \mathbf{E}; \mathbf{x}_2^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{\text{pos}}$$

- Patch embedding** - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(H^2 \cdot C) \times D}$

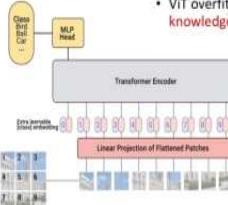
Add learnable position embeddings $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information

Feed the resulting sequence of vectors \mathbf{z}_0 to a standard Transformer encoder

$$\begin{aligned} \text{Transformer Encoder: } & \mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_1^1 \mathbf{E}; \mathbf{x}_2^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{\text{pos}}, \quad \mathbf{E} \in \mathbb{R}^{(H^2 \cdot C) \times D}, \quad \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D} \\ & \mathbf{z}'_l = \text{MSA}(\text{LN}(\mathbf{z}_{l-1})) + \mathbf{z}_{l-1}, \quad l = 1, \dots, L \\ & \mathbf{z}_l = \text{MLP}(\text{LN}(\mathbf{z}'_l)) + \mathbf{z}'_l, \quad l = 1, \dots, L \end{aligned}$$

Classification Head: $y = \text{MLP}(\mathbf{z}_L^1)$

The output of the additional [class] token is transformed into a class prediction via a small multi-layer perceptron (MLP) with tanh as non-linearity in the single hidden layer.

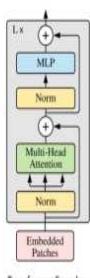


Transformer Encoder

- Consists of a multi-head self-attention module (MSA), followed by a 2-layer MLP (with GELU)
- LayerNorm (LN) is applied before MSA module and MLP, and a residual connection is applied after each module.

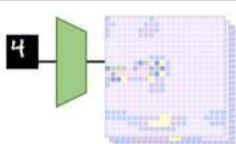
$$\mathbf{z}'_l = \text{MSA}(\text{LN}(\mathbf{z}_{l-1})) + \mathbf{z}_{l-1},$$

$$\mathbf{z}_l = \text{MLP}(\text{LN}(\mathbf{z}'_l)) + \mathbf{z}'_l,$$



ViT conducts global self-attention

- Relationships between a token and all other tokens are computed
- Quadratic complexity with respect to the number of tokens
- Not tractable for dense prediction or to represent a high-resolution image



An equivariant mapping is a mapping which preserves the algebraic structure of a transformation.

A translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping.

Inductive Bias

- ViT has much less image-specific inductive bias than CNNs

Inductive bias in CNN

- Locality
- Two-dimensional neighborhood structure
- Translation equivariance

ViT

- Only MLP layers are local and translationally equivariant. Self-attention is global
- Two dimensional neighborhood is used sparingly – i) only at the beginning where we cut image into patches, ii) learnable position embedding (spatial relations have to be learned from scratch)

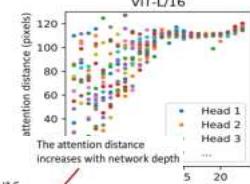
Examine the attention distance, analogous to receptive field in CNN

Compute the average distance in image space across which information is integrated, based on the attention weights.

Attention distance was computed for 128 example images by averaging the distance between the query pixel and all other pixels, weighted by the attention weight.

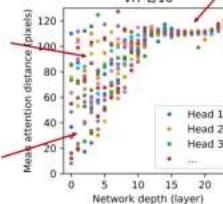
Each dot shows the mean attention distance across images for one of 16 heads at one layer. Image width is 224 pixels.

An example: if a pixel is 20 pixels away and the attention weight is 0.5 the distance is 10.



Some heads attend to most of the image already in the lowest layers, showing the capability of ViT in integrating information globally

Other attention heads have consistently small attention distances in the low layers

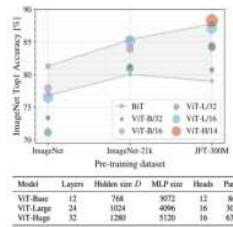


Performance of ViT

- ViT performs significantly worse than the CNN equivalent (BiT) when trained on ImageNet (1M images).

- However, on ImageNet-21k (14M images) performance is comparable, and on JFT (300M images), ViT outperforms BiT.

- ViT overfits the ImageNet task due to its lack of inbuilt knowledge about images

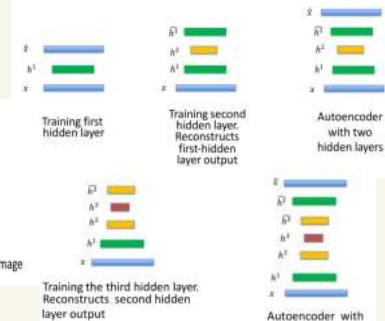


Deep stacked autoencoders

Deep autoencoders can be built by **stacking autoencoders** one after the other.

Training of deep autoencoders is done in a step-by-step fashion **one layer at a time**:

- After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder.
- The second level hidden representation can be used to train the third level of the encoders.
- This process is repeated and deep stacked autoencoder can be realized.

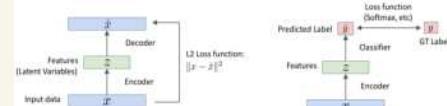


Semi-Supervised Classification

- Many images, but few ground truth labels

start unsupervised
train autoencoder on many images

supervised fine-tuning
train classification network on labeled images



10. Autoencoders

Supervised Learning

Data: (x, y)
 x is data, y is label

Goal: Learn a function to map $x \rightarrow y$

Examples: Classification, regression, object detection, semantic segmentation, Image captioning, etc.

Density Estimation

Feature Learning (e.g. autoencoders)

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden structure of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden structure of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Training autoencoders

The cost function of reconstruction can be measured by many ways, depending on the appropriate distributional assumptions on the inputs.

Learning of autoencoders is **unsupervised** as no specific targets are given.

Given a training set $\{x_p\}_{p=1}^P$.

The mean-square-error cost is usually used if the data is assumed to be continuous and Gaussian distributed:

$$J_{mse} = \frac{1}{P} \sum_{p=1}^P \|y_p - x_p\|^2$$

where y_p is the output for input x_p and $\|\cdot\|$ denotes the magnitude of the vector.

If the inputs are interpreted as bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

$$J_{cross\text{-}entropy} = - \sum_{p=1}^P [x_p \log y_p + (1 - x_p) \log(1 - y_p)]$$

Learning are done by using gradient descent:

$$\begin{aligned} W &\leftarrow W - \alpha \nabla_W J \\ b &\leftarrow b - \alpha \nabla_b J \\ c &\leftarrow c - \alpha \nabla_c J \end{aligned}$$

Denoising Autoencoders (DAE)

A denoising autoencoder (DAE) receives corrupted data points as inputs. It is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to force the hidden layer to identify. We corrupt the input data and then train the model to learn a denoised version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to **undo** the effect of corruption process applied to the input of the autoencoder.

For training DAE:

- First, input data is corrupted to mimic the noise in the images: $x \rightarrow \tilde{x}$
- \tilde{x} is the corrupted version of data. The corruption process simulates the distribution of data

DAE: Corrupting inputs

To obtain corrupted version of input data, each input x_i of input data is added with additive or multiplicative noise.

Additive noise:

$$\tilde{x}_i = x_i + \epsilon$$

where noise ϵ is Gaussian distributed: $\epsilon \sim N(0, \sigma^2)$. And σ is the standard deviation that determines the S/N ratio. Usually used for continuous data.

Multiplicative noise:

$$\tilde{x}_i = cx_i$$

where noise c could be Binomially distributed: $c \sim \text{Binomial}(p)$. And p is the probability of ones and $1 - p$ is the probability of zeros (noise). Usually, used for binary data.

Autoencoders

Input dimension n and hidden dimension M :

- If $M < n$, **undercomplete** autoencoders
- If $M > n$, **overcomplete** autoencoders

Undercomplete autoencoders

In undercomplete autoencoders, the hidden-layer has a **lower dimension** than the input layer.

By learning to approximate an n -dimensional inputs with M ($< n$) number of hidden units, we obtain a **lower dimensional representation** of the input signals. The network reconstructs the input signals from the reduced-dimensional hidden representation.

Learning an undercomplete representation forces the autoencoder to capture the most salient features.

By limiting the number of hidden neurons, hidden structures of input data can be inferred from autoencoders. For example, correlations among input variables, learning principal components of data, etc.

Overcomplete autoencoders

In overcomplete autoencoders, the hidden-layer has a **higher dimension** than the dimension of the input.

In order to learn useful information from overcomplete autoencoders, it is necessary use **some constraints** on its characteristics.

Even when the hidden dimensions are large, one can still explore interesting structures of inputs by introducing other constraints such as **'sparsity'** of input data.

Sparse Autoencoders (SAE)

A sparse autoencoder (SAE) is simply an autoencoder whose training criterion involves the sparsity penalty $\Omega_{sparse}(h)$ in the hidden layer:

$$J_s = J + \beta \Omega_{sparse}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be **sparse**.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be inactive for most of the time.

We say that the neuron is "active" when its output is close to 1 and the neuron is "inactive" when its output is close to 0.

Sparcity constraint

For a set $\{x_p\}_{p=1}^P$ of input patterns, the **average activation** ρ_j of neuron j at the hidden-layer is given by:

$$\rho_j = \frac{1}{P} \sum_{p=1}^P h_{pj}$$

where h_{pj} is the activation of hidden neuron j for p th pattern, and w_j and b_j are the weights and biases of the hidden neuron j .

We would like to enforce the constraint: $\rho_j = \rho$ such that the **sparsity parameter** ρ is set to a **small value** close to zero (say 0.05).

That is, most of the time, hidden neuron activations are maintained at ρ on average. By choosing a smaller value for ρ , the neurons are **activated selectively** to patterns and thereby learn sparse features.

To achieve sparse activations at the hidden-layer, the Kullback-Leibler (KL) divergence is used as the sparsity constraint:

$$D_{KL}(\rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j})$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the **deviation of the distribution** $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ .

The KL divergence is **minimum** when $\rho_j = \rho$ for all j . That is, when the average activations are uniform and equal to very low value ρ .

Regularizing autoencoders

Both undercomplete and overcomplete autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. Some form of constraints are needed in order to make them useful.

Deep autoencoders are only possible when some **constraints** are imposed on the cost function.

Regularized autoencoders incorporate a penalty to the cost function to learn interesting features from the input.

Regularized autoencoders provide the ability to train any autoencoder architecture successfully by choosing suitable code dimension and the capacity of the encoder and decoder.

With regularized autoencoders, one can use **larger model capacity** (for example, deeper autoencoders) and **large code size**.

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

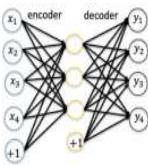
$$J_r = J + \beta \Omega(h)$$

where β is the penalty or regularization parameter. The penalty is usually imposed on the hidden activations.

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn trivial identity function.

The regularized loss function encourages the model to have other properties besides the ability to copy its input to its output.

Autoencoders

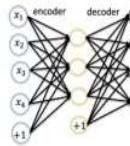


Input dimension n and hidden dimension M :

- If $M < n$, **undercomplete** autoencoders
- If $M > n$, **overcomplete** autoencoders

Undercomplete autoencoders

In undercomplete autoencoders, the hidden-layer has a **lower dimension** than the input layer.

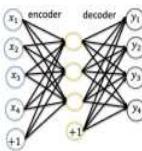


By learning to approximate an n -dimensional inputs with M ($< n$) number of hidden units, we obtain a **lower dimensional representation** of the input signals. The network reconstructs the input signals from the reduced-dimensional hidden representation.

Learning an undercomplete representation forces the autoencoder to capture the most salient features.

By limiting the number of hidden neurons, **hidden structures** of input data can be inferred from autoencoders. For example, correlations among input variables, learning principal components of data, etc.

Overcomplete autoencoders

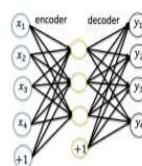


In overcomplete autoencoders, the hidden-layer has a **higher dimension** than the dimension of the input.

In order to learn useful information from overcomplete autoencoders, it is **necessary** use **some constraints** on its characteristics.

Even when the hidden dimensions are large, one can still explore interesting structures of inputs by introducing other constraints such as '**sparsity**' of input data.

Regularizing autoencoders



Both undercomplete and overcomplete autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. Some form of constraints are needed in order to make them useful.

Deep autoencoders are only possible when some **constraints** are imposed on the **cost function**.

Regularized autoencoders incorporate a penalty to the cost function to learn interesting features from the input.

Regularized autoencoders provide the ability to train any autoencoder architecture successfully by choosing suitable code dimension and the capacity of the encoder and decoder.

With regularized autoencoders, one can use **larger model capacity** (for example, deeper autoencoders) and large code size.

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

$$J_1 = J + \beta \Omega(h)$$

where β is the penalty or regularization parameter. The penalty is usually **imposed** on the **hidden activations**.

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn trivial identity function.

The regularized loss function encourages the model to have other properties besides the ability to copy its input to its output.

Sparse Autoencoders (SAE)

A **sparse autoencoder** (SAE) is simply an autoencoder whose training criterion involves the **sparsity penalty** Ω_{sparsity} at the hidden layer:

$$J_1 = J + \beta \Omega_{\text{sparsity}}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be **sparse**.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be **inactive for most of the time**.

We say that the neuron is "active" when its output is close to 1 and the neuron is "inactive" when its output is close to 0.

Sparsity constraint

For a set $\{x_p\}_{p=1}^P$ of input patterns, the **average activation** ρ_j of neuron j at the hidden-layer is given by:

$$\rho_j = \frac{1}{P} \sum_{p=1}^P h_{pj} = \frac{1}{P} \sum_{p=1}^P f(x_p^\top w_j + b_j)$$

where h_{pj} is the activation of hidden neuron j for p th pattern, and w_j and b_j are the weights and biases of the hidden neuron j .

We would like to enforce the constraint: $\rho_j = p$ such that the **sparsity parameter** p is set to a **small value** close to zero (say 0.05).

That is, most of the time, hidden neuron activations are maintained at p on average. By choosing a smaller value for p , the neurons are **activated selectively** to patterns and thereby learn **sparse features**.

To achieve sparse activations at the hidden-layer, the **Kullback-Leibler (KL)** divergence is used as the sparsity constraint:

$$D(h) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the **deviation** of the distribution $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ .

The KL divergence is **minimum** when $\rho_j = \rho$ for all j .

That is, when the average activations are uniform and equal to very low value ρ .

Sparse Autoencoder (SAE)

The cost function for the sparse autoencoder (SAE) is given by

$$J_1 = J + \beta D(h)$$

where $D(h)$ is the KL divergence of hidden-layer activations:

$$D(h) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

For gradient descent,

$$\nabla_w f_1 = \nabla_w J + \beta \nabla_w D(h)$$

By chain rule:

$$\frac{\partial D(h)}{\partial w_j} = \frac{\partial D(h)}{\partial \rho_j} \frac{\partial \rho_j}{\partial w_j} \quad (\text{A})$$

where

$$D(h) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

and

$$\begin{aligned} \frac{\partial D(h)}{\partial \rho_j} &= \rho \frac{1}{\rho_j} \left(-\frac{\rho}{\rho_j^2} \right) + (1 - \rho) \left(\frac{1}{1 - \rho_j} \right) \left(\frac{-(1 - \rho)}{(1 - \rho_j)^2} \right) \\ &= -\frac{\rho}{\rho_j} + \frac{1 - \rho}{1 - \rho_j} \end{aligned} \quad (\text{B})$$

Note:

$$\rho_j = \frac{1}{P} \sum_{p=1}^P f(u_{pj})$$

where $u_{pj} = x_p^\top w_j + b_j$ is the synaptic input of the j th neuron due to p th pattern

$$\frac{\partial \rho_j}{\partial w_j} = \frac{1}{P} \sum_p f'(u_{pj}) \frac{\partial (u_{pj})}{\partial w_j} = \frac{1}{P} \sum_p f'(u_{pj}) x_p \quad (\text{C})$$

Substituting (B) and (C) in (A):

$$\frac{\partial D(h)}{\partial w_j} = \frac{1}{P} \left(\frac{\rho}{\rho_j} + \frac{1 - \rho}{1 - \rho_j} \right) \sum_p f'(u_{pj}) x_p$$

Substituting in (A), we can find:

$$\nabla_w D(h) = (\nabla_{w_1} D(h) \quad \nabla_{w_2} D(h) \quad \dots \quad \nabla_{w_M} D(h))$$

The gradient of the constrained cost function:

$$\nabla_w f_1 = \nabla_w J + \beta \nabla_w D(h)$$

Similarly, $\nabla_b f_1$ and $\nabla_c f_1$ can be derived.

Learning can be done as

$$\begin{aligned} W &\leftarrow W - \alpha \nabla_w J \\ b &\leftarrow b - \alpha \nabla_b J \\ c &\leftarrow c - \alpha \nabla_c J \end{aligned}$$

Design the following autoencoder to process MNIST images:

1. Undercomplete autoencoder with 100 hidden units
2. Overcomplete autoencoder with 900 hidden units
3. Sparse autoencoder with 900 hidden units. Use sparsity parameter = 0.5

The MNIST database of gray level images of handwritten digits:
<http://yann.lecun.com/exdb/mnist/>

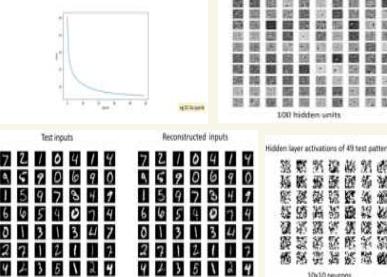
Each image is 28x28 size.

Intensities are in the range [0, 255].

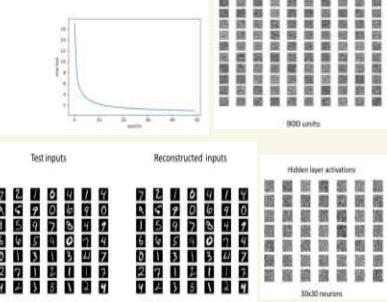
Training set: 60,000

Test set: 10,000

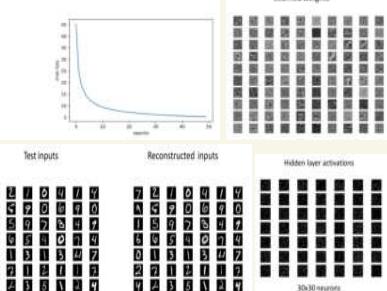
Example 3a: Undercomplete autoencoder



Example 3b: Overcomplete autoencoder



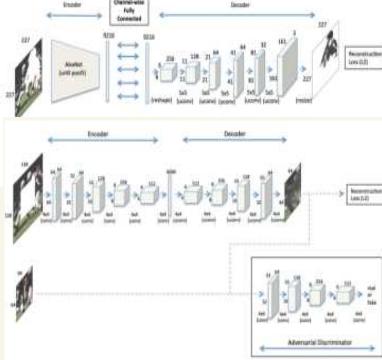
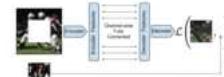
Example 3c: Sparse autoencoder



Deep autoencoders can be built by stacking autoencoders one after the other.

Training of deep autoencoders is done in a step-by-step fashion **one layer at a time**:

- After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder.
- The second level hidden representation can be used to train the third level of the encoders.
- This process is repeated and deep stacked autoencoder can be realized.

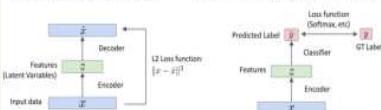


Semi-Supervised Classification

- Many images, but few ground truth labels

start unsupervised
train autoencoder on many images

supervised fine-tuning
train classification network on labeled images



11. Generative Adversarial Networks

Generative Adversarial Networks (GAN)

GAN Training

Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Applications of supervised learning



Why generative models?

We've only seen discriminative models so far

- Given an data x , predict a label y
- Estimates $p(y|x)$

Discriminative models have several key limitations

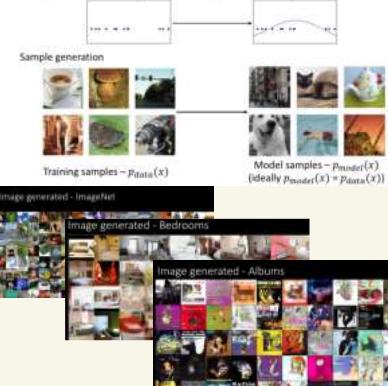
- Can't model $p(x)$, i.e., the probability of seeing a certain data
- Thus, can't sample from $p(x)$, i.e., can't generate new data

Generative models (in general) cope with all of above

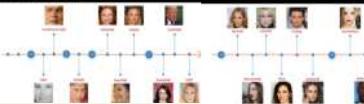
- Can model $p(x)$
- Can generate new data or images

Generative modeling

Density estimation – a core problem in unsupervised learning



The GAN Revolution 2014-2017



Results STARGAN

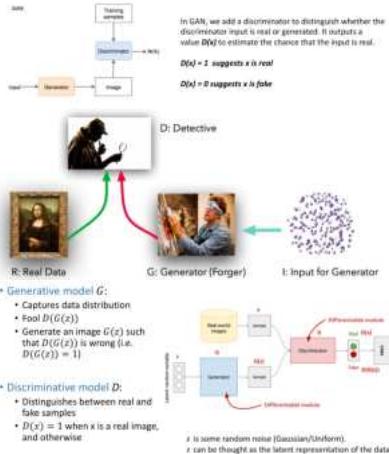


Generative Adversarial Networks (GAN)

Training procedure



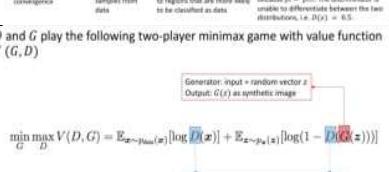
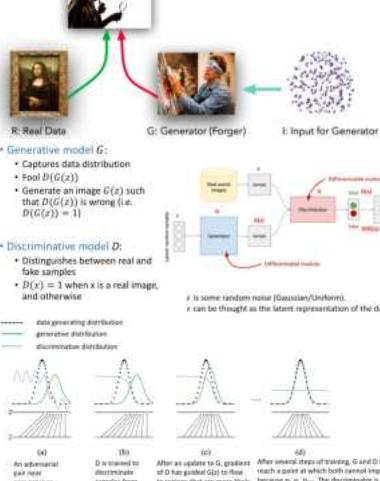
GAN samples noise z using normal or uniform distribution and utilizes a deep network generator G to create an image \hat{x} [Eqn 9]



In GAN, we add a discriminator to distinguish whether the discrimination input is real or generated. It outputs a value $D(x)$ to estimate the chance that the input is real.

$$D(x) = 2 \text{ suggests } x \text{ is real}$$

$$D(x) = 0 \text{ suggests } x \text{ is fake}$$



$$\min_{G} V(D, G) = \mathbb{E}_{x \sim P_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim P_{\text{noise}}(z)} [\log (1 - D(G(z)))]$$

Generator: input = random vector z
Output: $G(z)$ = synthetic image

Discriminator: input = generated/real images
Output: $D(x) = 1 \text{ if real image}$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial net. The number of steps to apply to the discriminator, K , is a hyperparameter. We used $K = 1$, the least expensive option, in our experiments.

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Dot = probably that x is not real

0 = generated image, 1 = real image

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

Average over m samples

Let's do an example

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

Average over m samples

Imagine for a real image D(x) scores 0.8

Imagine for a generated image D(G(z)) = 0.2

Note showing D(x) not logit

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

Average over m samples

Then for a generated image, D(x) score 0.2 that it is a real image (good)

Note showing D(x) not logit

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

Average over m samples

Then for a generated image, D(G(z)) = 0.2 that it is a real image (good)

Note showing D(G(z)) not logit

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

Average over m samples

Then for a generated image, D(x) = 0.2 that it is a real image (good)

Note showing D(x) not logit

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $P_{\text{noise}}(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $P_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))].$$

end for

Average over m samples

Then for a generated image, D(G(z)) = 0.2 that it is a real image (good)

Note showing D(G(z)) not logit

11. Generative Adversarial Networks

Why generative models?

- We've only seen **discriminative models** so far
 - Given an data x , predict a label y
 - Estimates $p(y|x)$

- Discriminative models have several key limitations**
 - Can't model $p(x)$, i.e., the probability of seeing a certain data
 - Thus, can't sample from $p(x)$, i.e., **can't generate new data**

- Generative models** (in general) cope with all of above
 - Can model $p(x)$
 - Can generate new data or images

Generative modeling

Density estimation – a core problem in unsupervised learning:

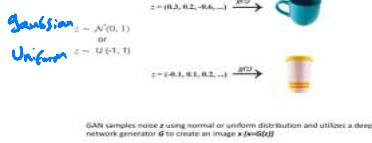


Image-to-Image Translation

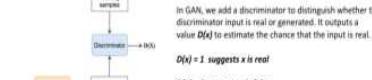


Inria et al., "Image-to-Image Translation with Conditional Adversarial Networks", arXiv 1611.07004

Generative Adversarial Networks (GAN)



GAN samples noise z using normal or uniform distribution and utilizes a deep network generator G to create an image $\{x \sim G\}$



In GAN, we add a discriminator to distinguish whether the discriminator input is real or generated. It outputs a value $D(x)$ to estimate the chance that the input is real.

$D(x) \approx 1$ suggests x is real

$D(x) \approx 0$ suggests x is fake

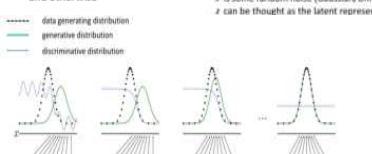
- Generative model G :**
 - Captures data distribution
 - Fool $D(G(z))$
 - Generate an image $G(z)$ such that $D(G(z))$ is wrong (i.e. $D(G(z)) = 1$)

- Discriminative model D :**
 - Distinguishes between real and fake samples
 - $D(x) = 1$ when x is a real image, and otherwise

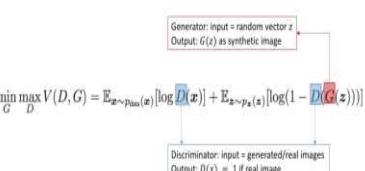
----- data generating distribution

generative distribution

discriminative distribution



D and G play the following two-player minimax game with value function $V(D, G)$



Example 1

Design a GAN to learn Gaussian distributed data with mean = 1.0 and standard deviation = 1.0. The generator receives uniformly distributed 1-dimensional inputs in the range [-1, +1].

Discriminator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden layer 1 = 10 (activation: Tanh)
- Number of hidden neurons in hidden layer 2 = 10 (activation: Tanh)
- Output dimension = 1 (binary classification)

Generator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden layer 1 = 5 (activation: Tanh)
- Number of hidden neurons in hidden layer 2 = 5 (activation: Tanh)
- Output dimension = 1 (activation: None)

GAN Training

Training procedure

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z^{(i)})))$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Dot product $\hat{x} \cdot \hat{y}$ is not $x \cdot y$

$\hat{x} \cdot \hat{y}$ = generated image, $x \cdot y$ = real image

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t. the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_i [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

Average over m samples

Discriminator: input=generated image, output=logit of real image

Generator: input=generated image, output=logit of real image

Note showing logit not logprob

Let's do an example

Dot product $\hat{x} \cdot \hat{y}$ is not $x \cdot y$

$\hat{x} \cdot \hat{y}$ = generated image, $x \cdot y$ = real image

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t. the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_i [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

Average over m samples

Discriminator: input=generated image, output=logit of real image

Note showing logit not logprob

Let's do an example

Dot product $\hat{x} \cdot \hat{y}$ is not $x \cdot y$

$\hat{x} \cdot \hat{y}$ = generated image, $x \cdot y$ = real image

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t. the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_i [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

Average over m samples

Discriminator: input=generated image, output=logit of real image

Note showing logit not logprob

Let's do an example

Dot product $\hat{x} \cdot \hat{y}$ is not $x \cdot y$

$\hat{x} \cdot \hat{y}$ = generated image, $x \cdot y$ = real image

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t. the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_i [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

Average over m samples

Discriminator: input=generated image, output=logit of real image

Note showing logit not logprob

Let's do an example

Dot product $\hat{x} \cdot \hat{y}$ is not $x \cdot y$

$\hat{x} \cdot \hat{y}$ = generated image, $x \cdot y$ = real image

for number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\text{Gradient w.r.t. the parameters of the Discriminator: } \nabla_{\theta_D} \frac{1}{m} \sum_i [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

Average over m samples

Discriminator: input=generated image, output=logit of real image

Note showing logit not logprob

Let's do another example

$D(x)$ = probability that x is a real image
 $P_{\text{gen}}(x)$ = generated image, $I = 1$ = real image

for number of training iterations do

- for i steps do
 - Sample minibatch of m noise samples $\{\tilde{z}^{(1)}, \dots, \tilde{z}^{(m)}\}$ from noise prior $p_{\theta}(\tilde{z})$.
 - Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$. [minibatch] [real image]
 - Update the discriminator by ascending its stochastic gradient:

Gradients w.r.t. the parameters of the Discriminator: $\nabla_{\theta_D} \left[\frac{1}{m} \sum_{i=1}^m \left[\log D(\tilde{z}^{(i)}) + \log \left(1 - D(x^{(i)})\right) \right] \right]$

Discriminator: Empirical log-likelihood loss,
superfunction of real image.

end for [Average over m samples]

- Sample minibatch of m noise samples $\{\tilde{z}^{(1)}, \dots, \tilde{z}^{(m)}\}$ from noise prior $p_{\theta}(\tilde{z})$.
- Update the generator by descending its stochastic gradient:

$D(G(\tilde{z})) = 0.2$
 $\log(1-0.2) = \log(0.8) = -0.2$ [D(G(\tilde{z}^{(i)}))]

$\nabla_{\theta_G} \left[\frac{1}{m} \sum_{i=1}^m \log \left(1 - D(\tilde{z}^{(i)})\right) \right]$

end for [Note showing In(x) not log(x)]

Get assigned a loss of -0.2

D(x) = probability that x is a real image
 θ = generated image; $1 - \epsilon$ = real image

for number of training iterations do

- for** k steps **do**
- Sample minibatch of m noise samples $\{z^{(1)}, z^{(2)}, \dots, z^{(m)}\}$ from noise prior $p_{\theta}(z)$.
- Sample minibatch of m examples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$ from data generating distribution $p_{\text{Data}}(x, y)$.
- Update the discriminator by **ascending** its stochastic gradient:

end for

Average over m samples

Discriminator: input=generated image, output=probability of real image.

Generator: input=generated image, output=synthetic image

Note showing $\ln(\epsilon)$ (not $\log(\epsilon)$)

D generated images - Real image

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_\theta(\mathbf{z})$.
- Sample minibatch of n examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by minimizing its stochastic gradient:

$$\text{Gradient w.r.t. } D: \nabla_D \left[\sum_i^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(G(\mathbf{z}^{(i)}))\right) \right] \right].$$

end for Average over m samples

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_\theta(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

$$\begin{aligned} \text{D}(G(\mathbf{z})) &= 0.2 \\ \log(D(\mathbf{z})) &+ \log(0.8) = -0.2 \\ \nabla_{\theta_g} \left[\sum_i^m \log \left(1 - D(G(\mathbf{z}^{(i)}))\right) \right]. \end{aligned}$$

end for Notice here we want to minimize this loss function (not maximize like before)

More examples

D(x) = probability that x is a real image
 $\neg D(x)$ = probability that x is a fake image

for n steps do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_\theta(z)$
- Sample minibatch of n examples $\{(x^{(1)}, \dots, x^{(n)})\}$ from data generating distribution $p_{\text{data}}(x)$
- Update the discriminator by ascending its stochastic gradient:

Gradient w.r.t. the parameters of the Discriminator: $-\nabla_{\theta^D} L = \sum_{i=1}^m \log D(x^{(i)}) + \log(1 - D(\tilde{x}^{(i)}))$

Discriminator: $D(x) = \text{softmax}(\text{linear}(x) + \text{bias})$
 Input: x (real/fake image)
 Output: prediction of real/fake image

Generator: $\tilde{x} = \text{softmax}(\text{linear}(z) + \text{bias})$
 Input: z (uniformly distributed random numbers)
 Output: synthetic image

end for

Average over m samples

$D(x) = 0.8$
 $\log(0.8) = -0.2$

$D(\tilde{x}|z) = 0.2$
 $\log(0.2) = -1.3$

$D(x) = 0.2$
 $\log(0.2) = -1.3$

$D(\tilde{x}|z) = 0.8$
 $\log(1-0.8) = \log(0.2) = -1.3$

Note showing $\neg D(x)$ not $D(x)$
 These "bad" predictions combined give a loss of -3.2

$-1.6 - 1.6 = -3.2$

end for

$D(x) = \text{probability that } x \text{ is a real image}$

for more training iterations do

- for j steps do
 - Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_{\theta}(\mathbf{z})$.
 - Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
 - Update the discriminator by ascending its stochastic gradient:

Gradient w.r.t. the parameters of the Discriminator: $-\nabla_{\theta_D} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(\mathbf{z}^{(i)})\right) \right]$

Discriminator (Supervised learning)
Output: classification of real image

end for

- Average over m samples
- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_{\theta}(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

$D(\mathbf{G}(\mathbf{z})) = 0.2$
 $\log(1-0.2) = \log(0.8) = -0.2$
 $\nabla_{\theta_G} \sum_{i=1}^m \log \left(1 - D(\mathbf{G}(\mathbf{z}^{(i)}))\right)$

$D(\mathbf{G}(\mathbf{z})) = 0.8$
 $\log(1-0.8) = \log(0.2) = -1.6$

This gives loss of -1.6

Notes showing $\text{log}(x)$

for $k = 1, \dots, m$ iterations do

- Sample minibatch of m noise samples $\{\bar{z}^{(1)}, \dots, \bar{z}^{(m)}\}$ from noise prior $p_{\theta}(\bar{z})$.
- Sample minibatch of m examples $\{\text{img}^{(1)}, \dots, \text{img}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\text{img})$.
- Update the discriminator by ascertaining its stochastic gradient.

Gradient w.r.t the parameters of the Discriminator: $-\nabla_{\theta} \left[\sum_{i=1}^m \left[\log D(\bar{z}^{(i)}) + \log \left(1 - D(\bar{z}^{(i)}) \right) \right] \right]$

end for

Average over m samples

- Sample minibatch of m noise samples $\{\bar{z}^{(1)}, \dots, \bar{z}^{(m)}\}$ from noise prior $p_{\theta}(\bar{z})$.
- Update the generator by descending its stochastic gradient:

Generator: reproducing synthesized images
Discriminator: Input: generated image
Output: probability of real image

Note showing Intro not log loss

for number of training iterations **do**

- for** k steps **do**
 - Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
 - Sample minibatch of m examples $\{\bar{x}^{(1)}, \dots, \bar{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\bar{x})$.
 - Update the discriminator by ascending its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator: $-\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(\bar{x}^{(i)}) + \log (1 - D(G(z^{(i)})))]$

Discriminator: (rep-)generative strategy, independent of real sample

end for Average over m samples.

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

$D(G(z)) = 0.8$

$\log(1.0 - 0.2) = \log(0.8) = -0.2$

$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)})))$

So minimizing this loss

Q(z) = probability that z is a real image
 \hat{z} = generated image; $\hat{z} \sim$ real image

Number of training iterations do

for k steps do

- Sample minibatch of m noise samples $\{\hat{z}^{(1)}, \dots, \hat{z}^{(m)}\}$ from noise prior $p_{\theta}(z)$.
- Sample minibatch of n examples $\{x^{(1)}, \dots, x^{(n)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by descending its stochastic gradient.

Generator: update random numbers, output synthetic image

Gradient w.r.t the parameters of D : $\nabla_{\theta_D} \sum_{i=1}^m \log \left(1 - D(G(\hat{z}^{(i)})) \right)$

The Discriminator: $\log(p_{\text{data}}(x^{(i)})) + \log(1 - D(G(\hat{z}^{(i)})))$

Discriminator loss: (real-generated image, expectation of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{\hat{z}^{(1)}, \dots, \hat{z}^{(m)}\}$ from noise prior $p_{\theta}(z)$.
- Update the generator by descending its stochastic gradient:

$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^m \log \left(D(G(\hat{z}^{(i)})) \right)$

Notes showing (not) log loss

for number of training iterations **do**

- for** k steps **do**
 - Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_{\theta}(\mathbf{z})$.
 - Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
 - Update the discriminator by **descending** its stochastic gradient:

Discriminator = π_1 the parameters of the Discriminator

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(G(\mathbf{z}^{(i)})) \right) \right]$$

Discriminator: Input-generated image
Output: prediction of real image

end for

Average over m samples

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_{\theta}(\mathbf{z})$.
- Update the generator by **descending** its stochastic gradient:

minimize

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D(G(\mathbf{z}^{(i)})) \right)$$

Note showing in

end for

means to generate images that fool D_G

$D(x) = \text{probability that } x \text{ is a real image}$
 \rightarrow generated image: 1 = real image

Uniform noise vector (random numbers)

Generator input noise output generated image

Training a system for rendering arbitrary images (arbitrary within the bounds of the subject matter that we trained the

Results from GAN



Example 1

Design a GAN to learn Gaussian distributed data with mean = 1.0 and standard deviation = 1.0. The generator receives uniformly distributed 1-dimensional inputs in the range [-1, +1].

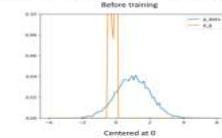
Discriminator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 10 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 10 (activation: Tanh)
- Output dimension = 1 (activation: None)

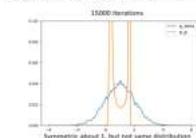
Generator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 5 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 5 (activation: Tanh)
- Output dimension = 1 (activation: None)

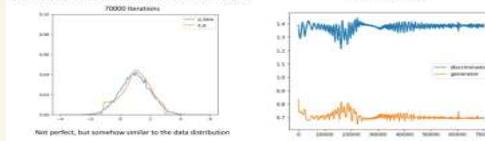
Histogram of 10000 randomly drawn points from $U[-1, 1]$



Histogram of 10000 randomly drawn points from $U[-1, 1]$

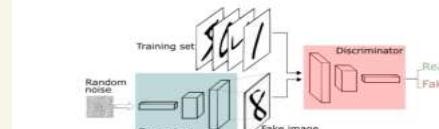


Histogram of 10000 randomly drawn points from $U[-1, 1]$



Example 2

Design a GAN to generate MNIST images from a uniformly distributed noise vector of 100 dimensions.



Example 2

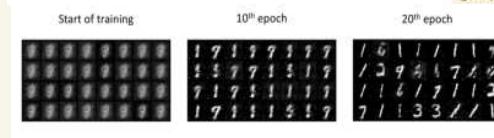
Generator:

- Input dimension = 100, drawn from a uniform distribution $U[-1, 1]$
- Hidden layers:
 - Number of hidden neurons = 256 (activation: ReLU)
 - Number of hidden neurons = 512 (activation: ReLU)
 - Number of hidden neurons = 1024 (activation: ReLU)
- Output dimension = 784 (activation: Tanh)

Discriminator:

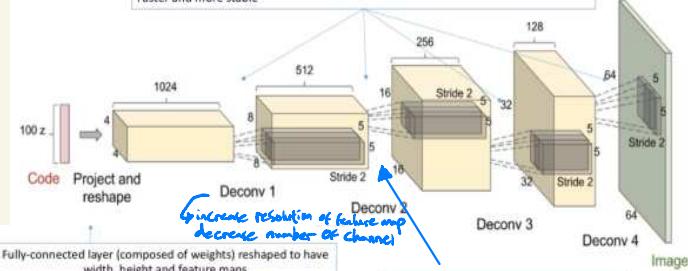
- Input dimension = 784
- Hidden layers:
 - Number of hidden neurons = 1024 (activation: ReLU)
 - Number of hidden neurons = 512 (activation: ReLU)
 - Number of hidden neurons = 256 (activation: ReLU)
- Output dimension = 1 (activation: Sigmoid, binary classification)

eg11.2.ipynb



Deep Convolutional GAN (DCGAN)

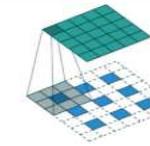
Most "deconv"s are batch normalized:
Normalize responses to have zero mean and unit variance over the entire mini-batch
Faster and more stable



We will go through the code in the tutorial

Fractionally-strided convolutions

<https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html>



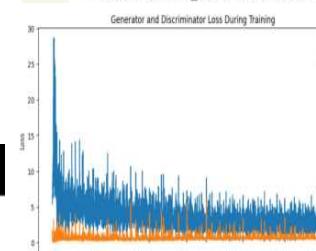
Up-sampling by fractional striding

Input 3x3, output = 5x5

stride = 1 convolution window = 3 x 3 (with respect to output)

Other tricks proposed by DCGAN

- Adam optimizer (adaptive moment estimation) = similar to SGD but with less parameter tuning
- Momentum = 0.5 (usually is 0.9 but training oscillated and was unstable)
- Low learning rate = 0.0002



Sanity check by walking on the manifold



Interpolating between a series of random points in \mathbf{z}
(\mathbf{z} = 100-d random numbers)
e.g.,
 $\mathbf{z}_{\text{new}} = m \mathbf{z}_1 + (1 - m) \mathbf{z}_2$

Note the smooth transition between each scene
← the window slowly appearing

This indicates that the space learned has smooth transitions!
(and is not simply memorizing)



Advantages of GANs

• Plenty of existing work on Deep Generative Models

- Boltzmann Machine
- Deep Belief Nets
- Variational AutoEncoders (VAE)

• Why GANs?

- Sampling (or generation) is straightforward.
- Training doesn't involve Maximum Likelihood estimation (i.e. finding the best model parameters that fit the training data the most), believed to cause poorer quality of images (blurry images)
- Robust to overfitting since the generator never sees the training data.

Problems with GAN

• Probability Distribution is Implicit

- Not straightforward to compute $p(x)$
- Thus Vanilla GANs are only good for Sampling/Generation.

• Training is Hard

- Non-Convergence
- Mode-Collapse

Non-convergence

• Deep Learning models (in general) involve a single player

- The player tries to maximize its reward (minimize its loss).
- Use SGD (with Backpropagation) to find the optimal parameters.
- SGD has convergence guarantees (under certain conditions).
- **Problem:** With non-convexity, we might converge to local optima.

$$\min_g L(g)$$

• GANs instead involve two (or more) players

- Discriminator is trying to maximize its reward.
- Generator is trying to minimize Discriminator's reward.

$$\min_g \max_D V(D, g)$$

- SGD was not designed to find the Nash equilibrium of a game.
- **Problem:** We might not converge to the Nash equilibrium at all.

$$\min_x \max_y V(x, y)$$

Let $V(x, y) = xy$

• State 1:	<table border="1"><tr><td>x>0</td><td>y>0</td><td>v>0</td></tr></table>	x>0	y>0	v>0	<table border="1"><tr><td>Increase y</td><td>Decrease x</td></tr></table>	Increase y	Decrease x
x>0	y>0	v>0					
Increase y	Decrease x						
• State 2:	<table border="1"><tr><td>x<0</td><td>y>0</td><td>v<0</td></tr></table>	x<0	y>0	v<0	<table border="1"><tr><td>Decrease y</td><td>Decrease x</td></tr></table>	Decrease y	Decrease x
x<0	y>0	v<0					
Decrease y	Decrease x						
• State 3:	<table border="1"><tr><td>x<0</td><td>y<0</td><td>v>0</td></tr></table>	x<0	y<0	v>0	<table border="1"><tr><td>Decrease y</td><td>Increase x</td></tr></table>	Decrease y	Increase x
x<0	y<0	v>0					
Decrease y	Increase x						
• State 4:	<table border="1"><tr><td>x>0</td><td>y<0</td><td>v<0</td></tr></table>	x>0	y<0	v<0	<table border="1"><tr><td>Increase y</td><td>Increase x</td></tr></table>	Increase y	Increase x
x>0	y<0	v<0					
Increase y	Increase x						
• State 5:	<table border="1"><tr><td>x>0</td><td>y>0</td><td>v>0</td></tr></table>	x>0	y>0	v>0	== State 1		
x>0	y>0	v>0					

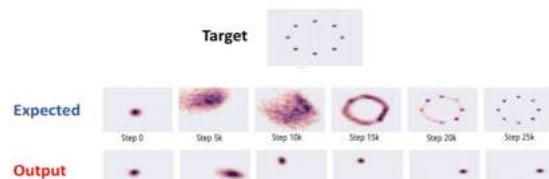
MNIST digits generated conditioned on their class label.

[1,0,0,0,0,0,0,0,0,0]	
[0,1,0,0,0,0,0,0,0,0]	
[0,0,1,0,0,0,0,0,0]	
[0,0,0,1,0,0,0,0,0]	
[0,0,0,0,1,0,0,0,0]	
[0,0,0,0,0,1,0,0,0]	
[0,0,0,0,0,0,1,0,0,0]	
[0,0,0,0,0,0,0,1,0,0]	
[0,0,0,0,0,0,0,0,1,0]	
[0,0,0,0,0,0,0,0,0,1]	

Figure 2 in the original paper.

Mode Collapse = Prob in Gan

Generator fails to output diverse samples



How to reward sample diversity

• At Mode Collapse,

- Generator produces good samples, but a very few of them.
- Thus, Discriminator can't tag them as fake.

• To address this problem,

- Let the Discriminator know about this edge-case.

• More formally,

- Let the Discriminator look at the entire batch instead of single examples
- If there is lack of diversity, it will mark the examples as fake

• Thus,

- Generator will be forced to produce diverse samples.

Mini-Batch GANs

• Extract features that capture diversity in the mini-batch

- For e.g. L2 norm of the difference between all pairs from the batch
- That is, the errors at an intermediate layer for real and fake samples are minimized.

• Feed those features to the discriminator along with the image

• Feature values will differ b/w diverse and non-diverse batches

- Thus, Discriminator will rely on those features for classification

• This in turn,

- Will force the Generator to match those feature values with the real data
- Will generate diverse batches

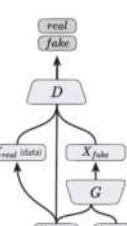
Supervision with Labels

• Label information of the real data might help



• Empirically generates much better samples

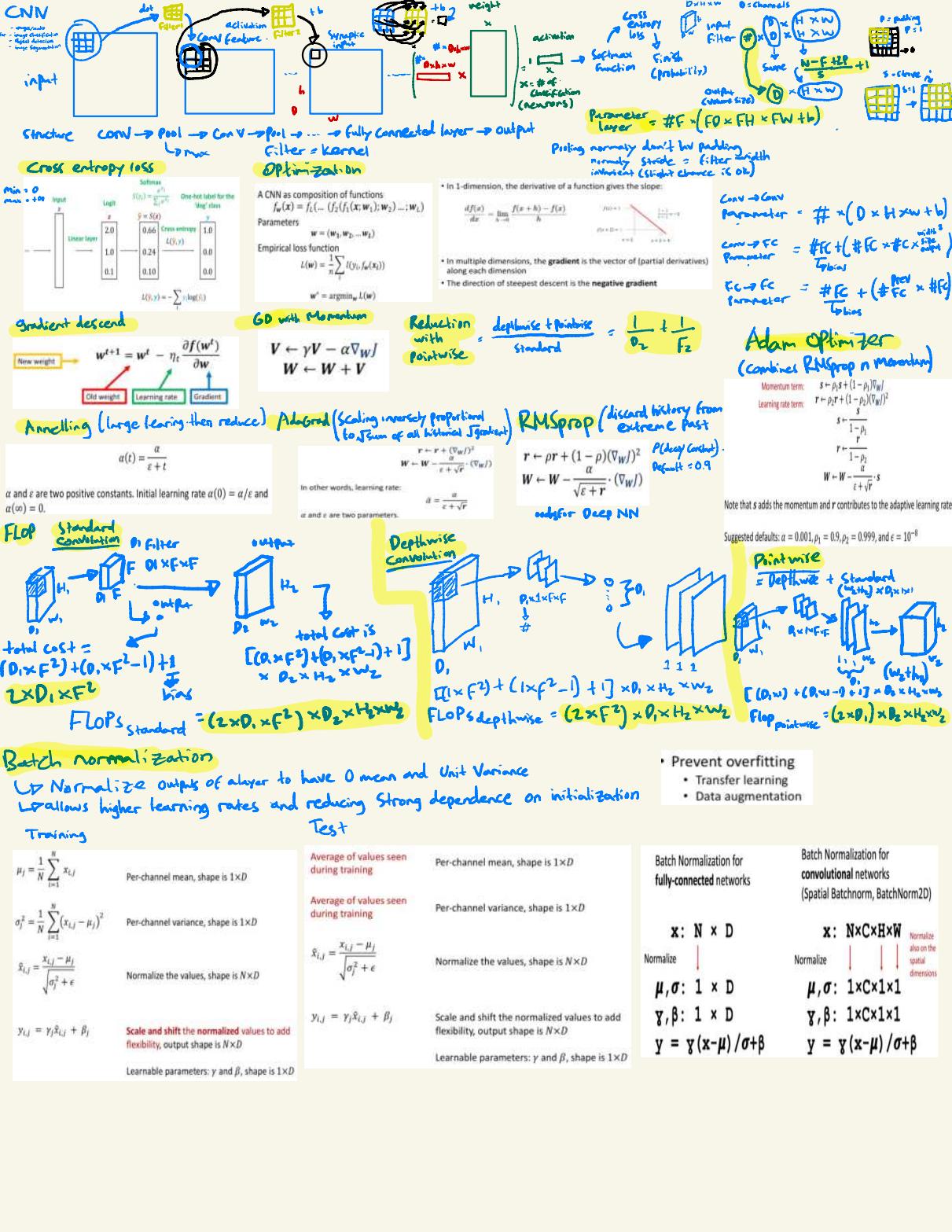
Conditional GANs



Conditions the inputs to the generator and discriminator with the labels

- Simple modification to the original GAN framework that conditions the model on *additional information* for better multi-modal learning.

- Lends to many practical applications of GANs when we have explicit *supervision* available.

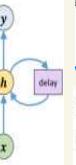


RNN
sequential data
for NLP
speech recognition
time series prediction
sentiment analysis
Machine translation

RNN hidden recurrence / Elman / Vanilla

The folded structure introduces two major advantages:

- new state $h(t) = f(h(t-1), x(t))$
- some function old state input vector at same time step
- 1. Regardless of the sequence length, the learned model always has the same size, rather than specified in terms of a variable-length history of states.
- 2. It is possible to use same transition function f with the same parameters at every time step.



Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Elman-type RNN with one hidden-layer is given by:

$$h(t) = \phi(U^T x(t) + W^T h(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

Batch Processing

Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Elman-type RNN is given by:

$$H(t) = \phi(X(t)U + H(t-1)W + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

U : weight vector that transforms raw inputs to the hidden-layer

W : recurrent weight vector connecting previous hidden-layer output to hidden input

V : weight vector of the output layer

b : bias connected to hidden layer

c : bias connected to the output layer

ϕ : the tanh hidden-layer activation function

σ : the linear/softmax output-layer activation function

$$H(t) = (\cdot) \quad Y(t) = (\cdot)$$

RNN top down / Jordan

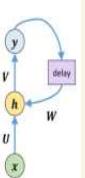
Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Jordan-type RNN with one hidden-layer is given by:

$$h(t) = \phi(U^T x(t) + W^T y(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

Note that output of the previous time instant is fed back to the hidden layer and W represents the recurrent weight matrix connecting previous output to the current hidden input



Batch Processing

Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Jordan-type RNN is given by:

$$H(t) = \phi(X(t)U + Y(t-1)W + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Backpropagation through time

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

$$h(t) = \tanh(U^T x(t) + W^T h(t-1) + b)$$

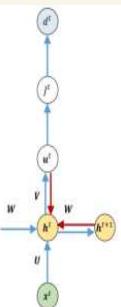
$$u(t) = V^T h(t) + c$$

For classification,

$$y(t) = \text{softmax}(u(t))$$

For regression,

$$y(t) = u(t)$$



The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph, followed by a backpropagation pass moving right to left through the graph. The gradients are propagated from the final time point to the initial time points.

The runtime is $O(T)$ where T is the length of input sequence and cannot be reduced by parallelization because the forward propagation is inherently sequential. The back-propagation algorithm applied to the unrolled graph with $O(T)$ cost is called back-propagation through time (BPTT).

The network with recurrence between hidden units is thus very powerful but also expensive to train.

Long Short-Term Memory (LSTM)

weights large \rightarrow Exploding gradient (learning diverge) \rightarrow use Gradient Clipping

clip gradient g when exceed threshold if $\|g\|_2 > V$: $\frac{\|g\|_2}{V} \cdot g$
Normalize the gradient when exceeds threshold if $\|g\|_2 > V$: $\frac{\|g\|_2 - V}{\|g\|_2} \cdot g$

Gates

\hookrightarrow Forget gate (mb/forget prev state)

Gated RNN \rightarrow LSTM, GR

$$f(t) = \sigma(U_f^T x(t) + W_f^T h(t-1) + b_f)$$

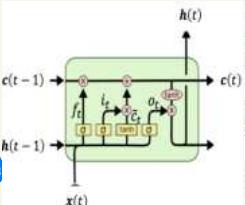
example:
language modelling

Value of $f(t)$ determines if $c(t-1)$ is to be remembered or not.

\hookrightarrow Input gate (allow incoming signal to alter the state of memory cell/block it)

Clever idea!

The self-loop inside the cell is able to produce paths where the gradient can flow for long duration and to make the weight of this self-loop conditioned on the context rather than fixed.



By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically based on the input sequence.

In this way, LSTM help preserve error terms that can be propagated through many layers and time steps.

Cell state

This has two parts:

A sigmoid input gate decides which values to update;

A tanh layer creates a vector of new candidate values $\tilde{c}(t)$ that could be added to the state.

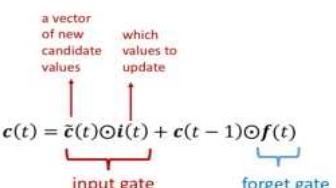
$$i(t) = \sigma(U_i^T x(t) + W_i^T h(t-1) + b_i)$$

$$\tilde{c}(t) = \phi(U_c^T x(t) + W_c^T h(t-1) + b_c)$$

\hookrightarrow Output gate (allow state to have effect on other neurons)

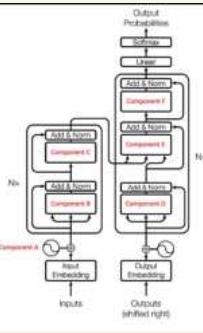
$$o(t) = \sigma(U_o^T x(t) + W_o^T h(t-1) + b_o)$$

$$h(t) = \phi(c(t)) \odot o(t)$$



Want test Computation Only theory

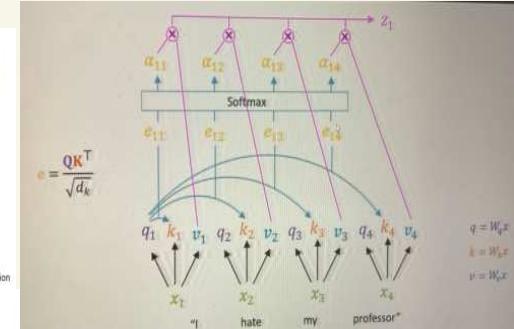
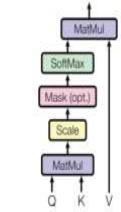
Transformer



Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys.

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Self-attention:

① Calculate Q, K, V $q_i = x_i W_Q$
From each encoder input vector $k_i = x_i W_K$
 $v_i = x_i W_V$

② Calculate Score $q_i \cdot k_i$
↓ how much attention

③ divide score by $\sqrt{d_k}$
for having more stable gradient
(large similarities cause softmax to saturate → vanishing gradient)

④ Softmax for normalization

⑤ multiply each value Vector $v_i = \text{softmax } \cdot v_i$

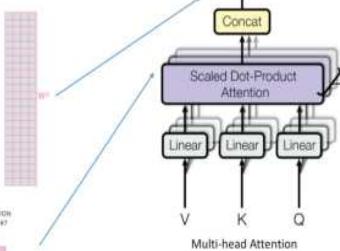
⑥ sum up weighted value vector $z_i = v_1 + v_2 + \dots$

Multi-Head Self-Attention

The independent attention outputs are simply concatenated and linearly transformed into the expected dimensions.



3) The result would be the Z matrix that captures information from all the attention heads. We can then feed it to the FFN.



1) This is our input sentence*

2) We embed each word*

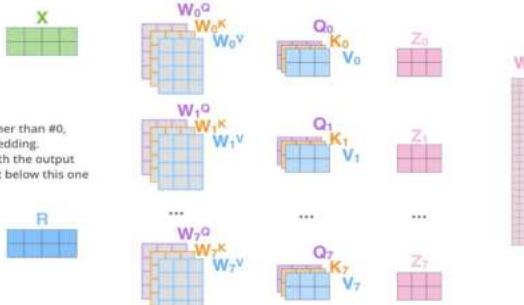
3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting $Q/K/V$ matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer

Thinking
Machines

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one.



Matrix Calculation of Self Attention

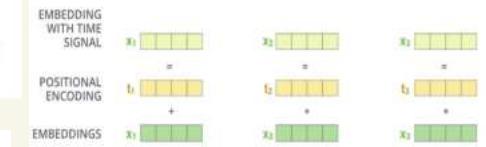
① Calculate Q, K, V



② Calculate Outputs of Self-attention layer
SoftMax is row-wise

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = Z$$

Positional Encoding



$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

where pos is the position and i is the dimension, $[0, \dots, d_{model}/2]$

Example: Positional Encoding

Example:

Given the following Sinusoidal positional encoding, calculate the PE ($pos = 1$) for the first five dimensions $[0, 1, 2, 3, 4]$. Assume $d_{model} = 512$

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Solution:

Given $pos = 1$ and $d_{model} = 512$

$$\text{At dimension } 0, 2i = 0 \text{ thus } i = 0, \text{ therefore } PE_{(pos, 2i)} = PE_{(1, 0)} = \sin(1/10000^{0/512})$$

$$\text{At dimension } 1, 2i + 1 = 1 \text{ thus } i = 0, \text{ therefore } PE_{(pos, 2i+1)} = PE_{(1, 1)} = \cos(1/10000^{0/512})$$

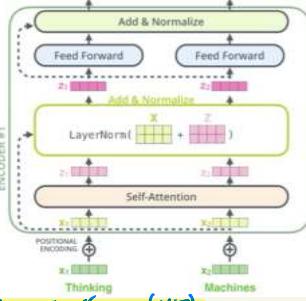
$$\text{At dimension } 2, 2i = 2 \text{ thus } i = 1, \text{ therefore } PE_{(pos, 2i)} = PE_{(1, 2)} = \sin(1/10000^{2/512})$$

$$\text{At dimension } 3, 2i + 1 = 3 \text{ thus } i = 1, \text{ therefore } PE_{(pos, 2i+1)} = PE_{(1, 3)} = \cos(1/10000^{2/512})$$

$$\text{At dimension } 4, 2i = 4 \text{ thus } i = 2, \text{ therefore } PE_{(pos, 2i)} = PE_{(1, 4)} = \sin(1/10000^{4/512})$$

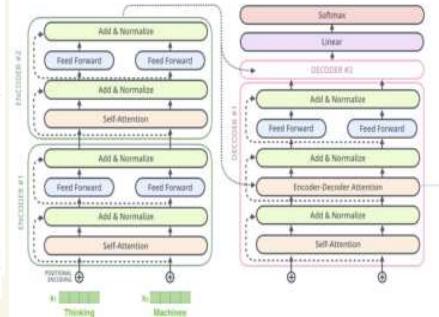
Transformer Encoder

Encoder



Transformer Decoder

KV



Vision Transformer (ViT)

Prepend a **learnable embedding** ($\mathbf{x}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches:

$$\mathbf{x}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_1^1 \mathbf{E}; \mathbf{x}_2^2 \mathbf{E}; \dots; \mathbf{x}_N^N \mathbf{E}] + \mathbf{E}_{\text{pos}}$$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(N+1) \times D}$

Add learnable position embeddings $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information

Training autoencoders

$$J_{mse} = \frac{1}{P} \sum_{p=1}^P \|y_p - x_p\|^2$$

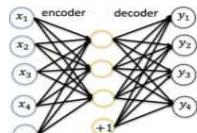
where y_p is the output for input x_p and $\|\cdot\|$ denotes the magnitude of the vector.

If the inputs are interpreted as bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

$$J_{cross\text{-}entropy} = - \sum_{p=1}^P (x_p \log y_p + (1 - x_p) \log (1 - y_p))$$

Learning are done by using gradient descent:

$$\begin{aligned} W &\leftarrow W - \alpha \nabla_W J \\ b &\leftarrow b - \alpha \nabla_b J \\ c &\leftarrow c - \alpha \nabla_c J \end{aligned}$$



Undercomplete Autoencoder:

- An undercomplete autoencoder is characterized by having an encoder that maps the input data to a lower-dimensional latent space (encoder's output) with fewer dimensions than the input data.
- The reduction in dimensionality forces the autoencoder to learn a compressed representation of the input data. As a result, undercomplete autoencoders are effective at capturing the most salient features of the data while ignoring noise and less relevant information.
- Undercomplete autoencoders are commonly used for dimensionality reduction, feature extraction, and data compression tasks.

Overcomplete Autoencoder:

- In contrast to undercomplete autoencoders, overcomplete autoencoders have an encoder that maps the input data to a higher-dimensional latent space (encoder's output) with more dimensions than the input data.
- Overcomplete autoencoders have more capacity to learn complex representations of the input data, potentially capturing more detailed information and capturing nonlinear relationships between features.
- However, overcomplete autoencoders are more prone to overfitting, as they can learn to encode noise and irrelevant details from the input data.

Regularized Autoencoder:

- Regularized autoencoders incorporate regularization techniques to control the complexity of the learned representations and prevent overfitting.
- Common regularization techniques used in autoencoders include L1 or L2 regularization, dropout, sparsity constraints, and denoising (adding noise to the input data during training).
- Regularized autoencoders strike a balance between capturing relevant information in the input data and preventing the model from memorizing noise or overfitting to the training data.

Sparse Autoencoder:

- Sparse autoencoders enforce sparsity in the learned representations by penalizing the activation of neurons in the hidden layers.
- Sparsity encourages the autoencoder to learn a compact and efficient representation of the input data, focusing on the most informative features while ignoring irrelevant details.
- Sparse autoencoders are particularly useful for tasks where feature interpretability or dimensionality reduction is important, as they tend to learn sparse, easily interpretable representations of the data.

Substituting in (A) , we can find:

$$\nabla_W D(\mathbf{h}) = (\nabla_{w_1} D(\mathbf{h}), \nabla_{w_2} D(\mathbf{h}), \dots, \nabla_{w_N} D(\mathbf{h}))$$

The gradient of the constrained cost function:

$$\nabla_{W/J} = \nabla_W / \beta \nabla_W D(\mathbf{h})$$

Similarly, ∇_b/J and ∇_c/J can be derived.

Learning can be done as

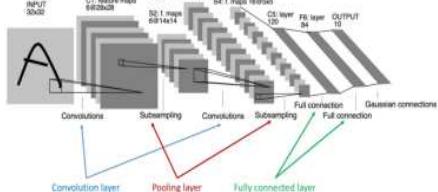
$$\begin{aligned} W &\leftarrow W - \alpha \nabla_{W/J} \\ b &\leftarrow b - \alpha \nabla_{b/J} \\ c &\leftarrow c - \alpha \nabla_{c/J} \end{aligned}$$

$$J_1 = J + \beta \Omega(\mathbf{h})$$

Revision & Applications of GANs

- Week 7 – Convolutional Neural Network (CNN) I
- Recess Week
- Week 8 – Convolutional Neural Network (CNN) II
- Week 9 – Recurrent Neural Networks (RNN)
- Week 10 – Attention
- Week 11 – Autoencoders
- Week 12 – Generative Adversarial Networks (GAN)
- Week 13 – Revision and Selected Topics

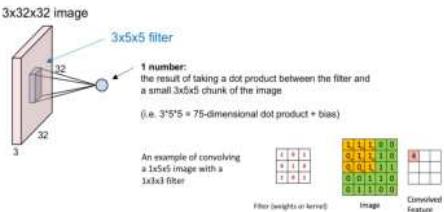
An example of convolutional network: LeNet 5



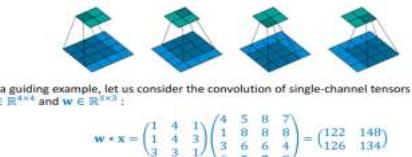
As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

Common layer arrangement: Conv → pool → Conv → pool → fully connected → fully connected → output

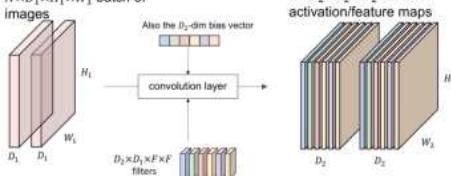
Convolution layer



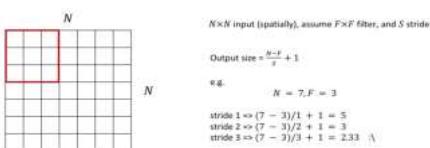
Convolution by doing a sliding window



$N \times D_1 \times H_1 \times W_1$ batch of images



Convolution layer – spatial dimensions

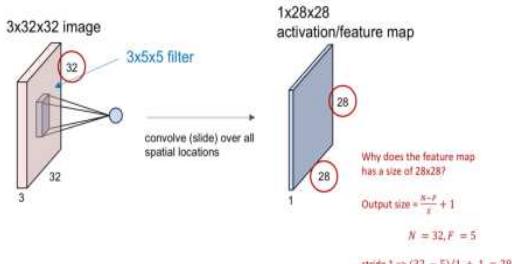


basic components
Training a classifier
optimizer → momentum + Adam + SGD

Cross entropy + softmax

(Conception no calculate)

Convolution layer – spatial dimensions



Why does the feature map has a size of 28x28?

$$Output\ size = \frac{N-F}{S} + 1$$

$$N = 32, F = 5$$

$$stride\ 1 \Rightarrow (32 - 5)/1 + 1 = 28$$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized, e.g., 'constant', 'reflect', and 'replicate').

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

e.g. input 7×7

3×3 filter, applied with stride 1
pad with 1 pixel border => what is the output shape?

Recall that without padding, output size = $\frac{N-F}{S} + 1$

With padding, output size = $\frac{N-F+2P}{S} + 1$

e.g.: $N = 7, F = 3$

stride 1 $\Rightarrow (7 - 3 + 2(1))/1 + 1 = 7$

7×7 output!

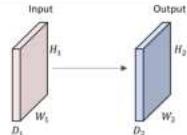
In general, common to see CONV layers with stride 1, Filters of size $F \times F$, and zero-padding with $(F - 1)/2$. (will preserve size spatially)

e.g. $F = 3 \Rightarrow$ zero pad with 1
 $F = 5 \Rightarrow$ zero pad with 2
 $F = 7 \Rightarrow$ zero pad with 3

Convolution layer - summary

A convolution layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
 - Number of filters K
 - Their spatial extent F
 - The stride S
 - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $D_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d -th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the



Pooling layer - summary

A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
 - Their spatial extent F
 - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $D_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

Outline – CNN II

- CNN Architectures
- More on convolution
 - How to calculate FLOPs!
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution

! computing

You learn some classic architectures

You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network

Batch normalization

- Prevent overfitting
 - Transfer learning
 - Data augmentation

3 conceptual levels

You learn an important technique to improve the training of modern neural networks

You learn two important techniques to prevent overfitting in neural networks

How to calculate the computations of Convolution?

FLOPs (floating point operations)

Not FLOPs (floating point operations per second)

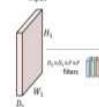
Assume:

- Filter size F
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Produces a output volume of size $D_2 \times H_2 \times W_2$

The FLOPs of the convolution layer is given by

$$\text{FLOPs} = \frac{[(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2}{\text{Element-wise multiplication of each filter in spatial location}} = [(D_1 \times F^2) \times D_2 \times H_2 \times W_2]$$

Adding the elements after multiplication, we are doing additions for the elements.



Pointwise convolution

Why having filter of spatial size 1×1 ?

- Change the size of channels
- "Blend" information among channels by linear combination



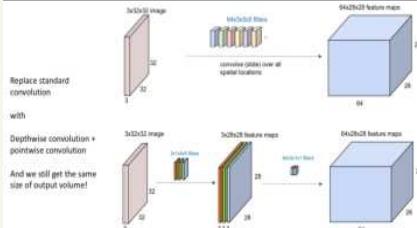
Depthwise convolution

Depthwise convolution

- Convolution is performed independently for each of input channels



Depthwise convolution + Pointwise convolution



How much computation do you save by replacing standard convolution with depthwise+pointwise?

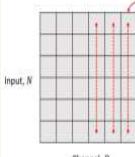
$$\text{Ratio} = \frac{\text{Cost of depthwise convolution + pointwise convolution}}{\text{Cost of standard convolution}}$$

$$\text{Ratio} = \frac{(2 \times F^2) \times D_1 \times H_1 \times W_1}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2} + \frac{(2 \times D_1) \times D_2 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2}$$

$$\text{Ratio} = \frac{1}{D_2} + \frac{1}{F^2}$$

D_2 is usually large. Reduction rate is roughly 3/8-1/9 if 3x3 depthwise separable convolutions are used

Batch Normalization



The goal of batch normalization is to normalize the values across each column so that the values of the column have zero mean and unit variance

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below



Training

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\tilde{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \tilde{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Test time

Input Channel, D

Problem: Estimates depend on minibatch; can't do this at test-time.

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\tilde{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \tilde{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

RNN with hidden recurrence

Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Elman-type RNN with one hidden-layer is given by:

$$h(t) = \phi(U^T x(t) + W^T h(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

σ is a softmax function for classification and a linear function for regression.

RNN with top-down recurrence

Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Jordan-type RNN with one hidden-layer is given by:

$$h(t) = \phi(U^T x(t) + W^T y(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

Note that output of the previous time instant is fed back to the hidden layer and W represents the recurrent weight matrix connecting previous output to the current hidden input.

Long short-term memory (LSTM) unit

LSTMs provide a solution by incorporating memory units that allow the network to learn when to forget previous hidden states and when to update hidden states given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.

Hochreiter & Schmidhuber (1997)

$$i(t) = \sigma(U_i^T x(t) + W_i^T h(t-1) + b_i)$$

$$f(t) = \sigma(U_f^T x(t) + W_f^T h(t-1) + b_f)$$

$$o(t) = \sigma(U_o^T x(t) + W_o^T h(t-1) + b_o)$$

$$\tilde{c}(t) = \phi(U_c^T x(t) + W_c^T h(t-1) + b_c)$$

$$c(t) = \tilde{c}(t) \odot i(t) + c(t-1) \odot f(t)$$

$$h(t) = \phi(c(t)) \odot o(t)$$

becomes $i(t)$ and $f(t)$

$c(t-1)$ and $h(t-1)$

$x(t)$ and $c(t)$

Outline – RNN

Recurrent Neural Network (RNN)

- Hidden recurrence
- Top-down recurrence

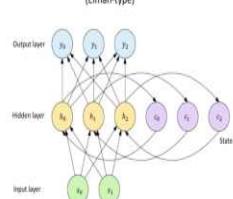
Long Short-Term Memory (LSTM)

- Long-term dependency
- Structure of LSTM

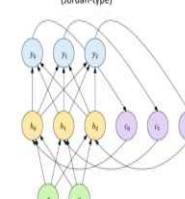
Example Applications

Types of RNN

RNN with hidden recurrence (Elman-type)



RNN with top-down recurrence (Jordan-type)



Outline – Attention

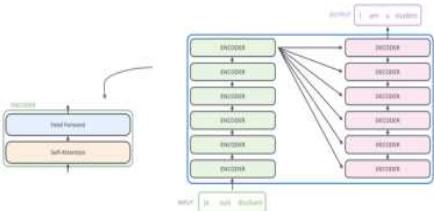
- Attention
- Transformers
- Vision Transformers

Transformers

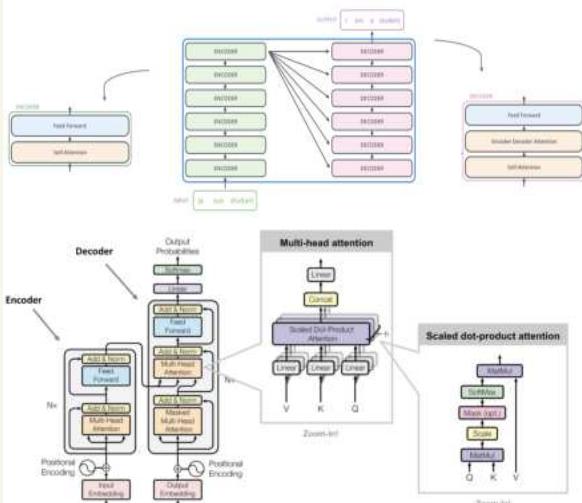
The encoder's inputs first flow through a self-attention layer – a layer that **helps the encoder look at other words in the input sentence** as it encodes a specific word

The outputs of the self-attention layer are fed to a **feed-forward neural network**.

The exact same feed-forward network is independently applied to each position (each word/token).



The decoder has both those layers, but between them is an attention layer that **helps the decoder focus on relevant parts of the input sentence**



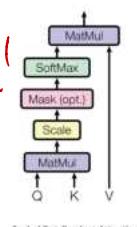
Transformers

Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

→ Computation!



Self-Attention in Detail

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).

Input	Thinking	Machines
Embedding	\mathbf{x}_1	\mathbf{x}_2
Queries	\mathbf{q}_1	\mathbf{q}_2
Keys	\mathbf{k}_1	\mathbf{k}_2
Values	\mathbf{v}_1	\mathbf{v}_2

So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

What are the "query", "key", and "value" vectors?

The names "query", "key" are inherited from the field of **information retrieval**

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{\mathbf{q}\mathbf{k}^T}{\sqrt{d_k}}$ depend on the relative similarities between the n -th query and all of the keys

The softmax function means that the **key vectors** "compete" with one another to contribute to the final result.

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.

Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)

Fourth Step

Softmax for normalization

Input	Thinking	Machines
Embedding	\mathbf{x}_1	\mathbf{x}_2
Queries	\mathbf{q}_1	\mathbf{q}_2
Keys	\mathbf{k}_1	\mathbf{k}_2
Values	\mathbf{v}_1	\mathbf{v}_2
Score	$\mathbf{q}_1 \cdot \mathbf{k}_1 = 112$	$\mathbf{q}_1 \cdot \mathbf{k}_2 = 96$

$$z_1 = \mathbf{q}_1 \cdot \mathbf{k}_1 / \sqrt{d_k} = 112/\sqrt{64} = 14, z_2 = \mathbf{q}_1 \cdot \mathbf{k}_2 / \sqrt{64} = 96/\sqrt{64} = 96/8 = 12$$

$$\text{softmax}(z_i) = \exp(z_i) / \sum_{i=1}^2 \exp(z_i) = 0.88, \text{softmax}(z_2) = \exp(z_2) / \sum_{i=1}^2 \exp(z_i) = 0.12$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Fifth Step

Multiply each value vector by the softmax score

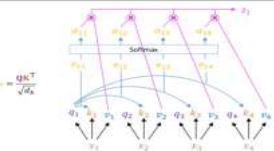
Input	Thinking	Machines
Embedding	\mathbf{x}_1	\mathbf{x}_2
Queries	\mathbf{q}_1	\mathbf{q}_2
Keys	\mathbf{k}_1	\mathbf{k}_2
Values	\mathbf{v}_1	\mathbf{v}_2
Score	$\mathbf{q}_1 \cdot \mathbf{k}_1 = 112$	$\mathbf{q}_1 \cdot \mathbf{k}_2 = 96$
Divide by $8 (\sqrt{64})$	14	12
Softmax	0.88	0.12

Sixth Step

Sum up the weighted value vectors to get the output of the self-attention layer at this position (for the first word)

Input	Thinking	Machines
Embedding	\mathbf{x}_1	\mathbf{x}_2
Queries	\mathbf{q}_1	\mathbf{q}_2
Keys	\mathbf{k}_1	\mathbf{k}_2
Values	\mathbf{v}_1	\mathbf{v}_2
Score	$\mathbf{q}_1 \cdot \mathbf{k}_1 = 112$	$\mathbf{q}_1 \cdot \mathbf{k}_2 = 96$
Divide by $8 (\sqrt{64})$	14	12
Softmax	0.88	0.12
Softmax X Value	$0.88 \cdot \mathbf{v}_1$	$0.12 \cdot \mathbf{v}_2$
Sum	$\mathbf{z}_1 = 0.88\mathbf{v}_1 + 0.12\mathbf{v}_2$	\mathbf{z}_2
Attention($\mathbf{Q}, \mathbf{K}, \mathbf{V}$) = $\text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$	\mathbf{z}_1	\mathbf{z}_2

Self-Attention in Detail



Multi-Head Self-Attention

Multi-Head Self-Attention
Rather than only computing the attention once, the multi-head architecture iterates through the scaled-dot product attention multiple times in parallel.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values



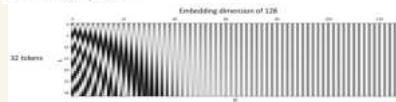
Positional Encoding

Self-attention layer works on sets of vectors and it doesn't know the order of the vectors as the input embedding

The positional encoding has the same dimension as the input embedding

Add a vector to each input embedding according to the relative or absolute position of the tokens in the sequence

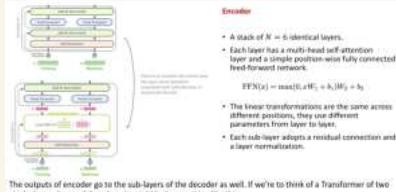
These vectors follow a specific pattern



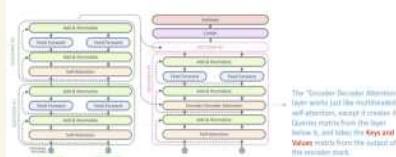
$$\begin{aligned} PE_{(pos),d} &= \sin((pos)/10000^{(d/n)}) \\ PE_{(pos+1),d} &= \cos((pos)/10000^{(d/n)}) \end{aligned}$$

where pos is the position and d is the dimension, $n = d_{\text{model}}/2$

Transformer Encoder



The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



Vision Transformer (ViT)

Do not have decoder

Reshape the image $x \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $x \in \mathbb{R}^{N \times (P^2 \cdot C)}$
 (H, W) is the resolution of the original image
 C is the number of channels
 (P, P) is the resolution of each image patch
 $N = RW/P^2$ is the resulting number of patches

Prepend a learnable embedding ($\mathbf{z}_0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

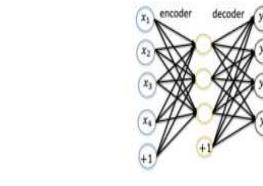
$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_1^1 \mathbf{E}; \mathbf{x}_1^2 \mathbf{E}; \dots; \mathbf{x}_1^N \mathbf{E}] + \mathbf{E}_{\text{pos}}$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

Add learnable position embeddings $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information

Outline – Autoencoders

- Supervised vs unsupervised learning
- Autoencoders
 - Denoising autoencoders
 - Undercomplete and overcomplete autoencoders
 - Sparse autoencoders
 - Other encoders



An autoencoder is a neural network that is trained to attempt to copy its input to its output. Its hidden layer describes a **code** that represents the input.

The network consists of two parts: an **encoder** and a **decoder**.

Reverse mapping from the hidden layer to the output can be **optionally** constrained to be the same as the input to hidden-layer mapping (**ties weights**). That is, if encoder weight matrix is W , the decoder weight matrix is W^T .

Hidden layer and output layer activation can then be written as:

$$\begin{aligned} h &= f(W^T x + b) \\ y &= f(W h + c) \end{aligned}$$

f is usually a sigmoid.

Denoising Autoencoders (DAE)

A **denoising autoencoder** (DAE) receives corrupted data points as inputs.

It is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to force the hidden layer to **discover more robust features** and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to **undo the effect of corruption process** applied to the input of the autoencoder.

Input dimension n and hidden dimension M :

If $M < n$, **undercomplete** autoencoders

If $M > n$, **overcomplete** autoencoders

Sparse Autoencoders (SAE)

A sparse autoencoder (SAE) is simply an autoencoder whose training criterion involves the **sparsity penalty** Ω applied at the hidden layer:

$$J = J + \beta \Omega_{\text{sparsity}}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden layers to be **sparsified**.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be **inactive for most of the time**.

We say that the neuron is "active" when its output is close to 1 and the neuron is "inactive" when its output is close to 0.

Sparsity constraint

To achieve sparse activations at the hidden-layer, the **Kullback-Leibler (KL) divergence** is used as the sparsity constraint:

$$D(h) = \sum_{j=1}^M p \log \frac{p}{p_j} + (1-p) \log \frac{1-p}{1-p_j}$$

where M is the number of hidden neurons and p is the sparsity parameter.

KL divergence measures the **deviation of the distribution** $\{p_j\}$ of activations at the hidden-layer from the uniform distribution of p .

The KL divergence is **minimum** when $p_j = p$ for all j . That is, when the average activations are uniform and equal to very low value p .

Outline – GAN

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Generative adversarial networks (GAN)

Generative model G :

- Captures data distribution
- Fool $D(G(z))$
- Generate an image $G(z)$ such that $D(G(z)) = 1$

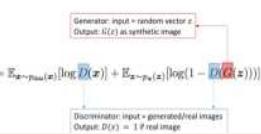


Discriminative model D :

- Distinguishes between real and fake samples
- $D(x) = 1$ when x is a real image, and otherwise $D(x) = 0$

x is some random noise (Gaussian/Uniform). x can be thought as the latent representation of the data

D and G play the following two-player minimax game with value function $V(D, G)$



Training procedure

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

For number of training iterations do

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_\theta(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D\left(G\left(z^{(i)} \right) \right) \right).$$

and for

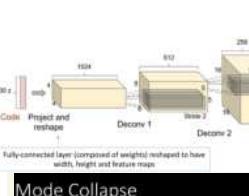
- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_\theta(z)$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D\left(G\left(z^{(i)} \right) \right) \right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Deep Convolutional GAN (DCGAN)



Mode Collapse

Generator fails to output diverse samples



Go if asked how to deal with mode collapse
↓
Write down the idea of minibatch
of minibatch to reflect the mini-batch feature of mini-batch as input

SC4001/4042 Overview

Neural Networks and Deep Learning

**Assistant Professor
Ying Wei**



Biological neural networks

Artificial neural networks are inspired by the biological neural networks in the brain. The three pounds of jelly-like material found within our brain is the most complex machine on earth and perhaps in the universe.

It consists of a **densely interconnected** set of nerve cells, or basic information-processing units, called **neurons**. Human brain incorporates nearly 10 billion neurons, each connected to about 10,000 other neurons with 60-100 trillion **connections**, synapses, between them.

By using multiple neurons simultaneously, the brain performs its functions much faster than the fastest computers in existence today. An **artificial neural network** is defined as a model of reasoning based on the principles of the human brain.

Brain vs computer



- Typical **operating speeds** of biological neurons is in milliseconds (10^{-3} s), while silicon chip operate in nanoseconds (10^{-9} s). But Brain makes up (for slower rate of operation of a neuron) by having significant number of neurons with **massive interconnections** between them.
- Human brain is extremely **energy efficient**, using approximately 10^{-16} joules per operation per second, whereas the computers use around 10^{-6} joules per operation per second.
- Brains have an **evolution** history of tens of millions of years, computers have been evolving for tens of decades.



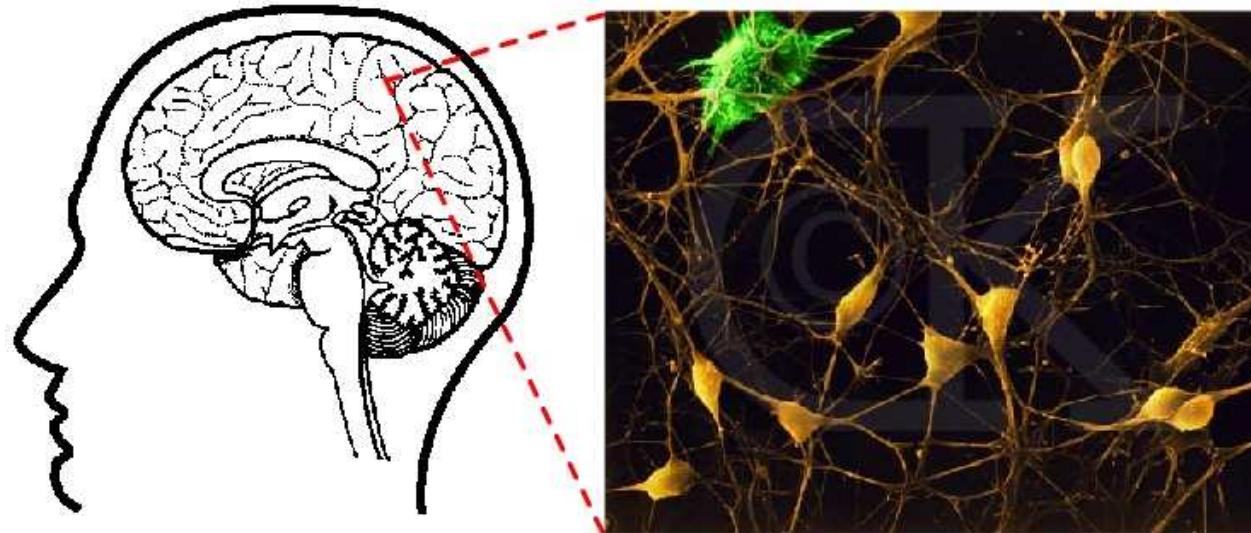
Information processing in the brain

Our brain is highly complex, **non-linear parallel information-processing** system.

Information is distributed throughout the whole network, rather than at specific locations and stored and processed in a neural network simultaneously. Today's computers have one or several processors but each neuron in the brain can be considered as a simple processor.

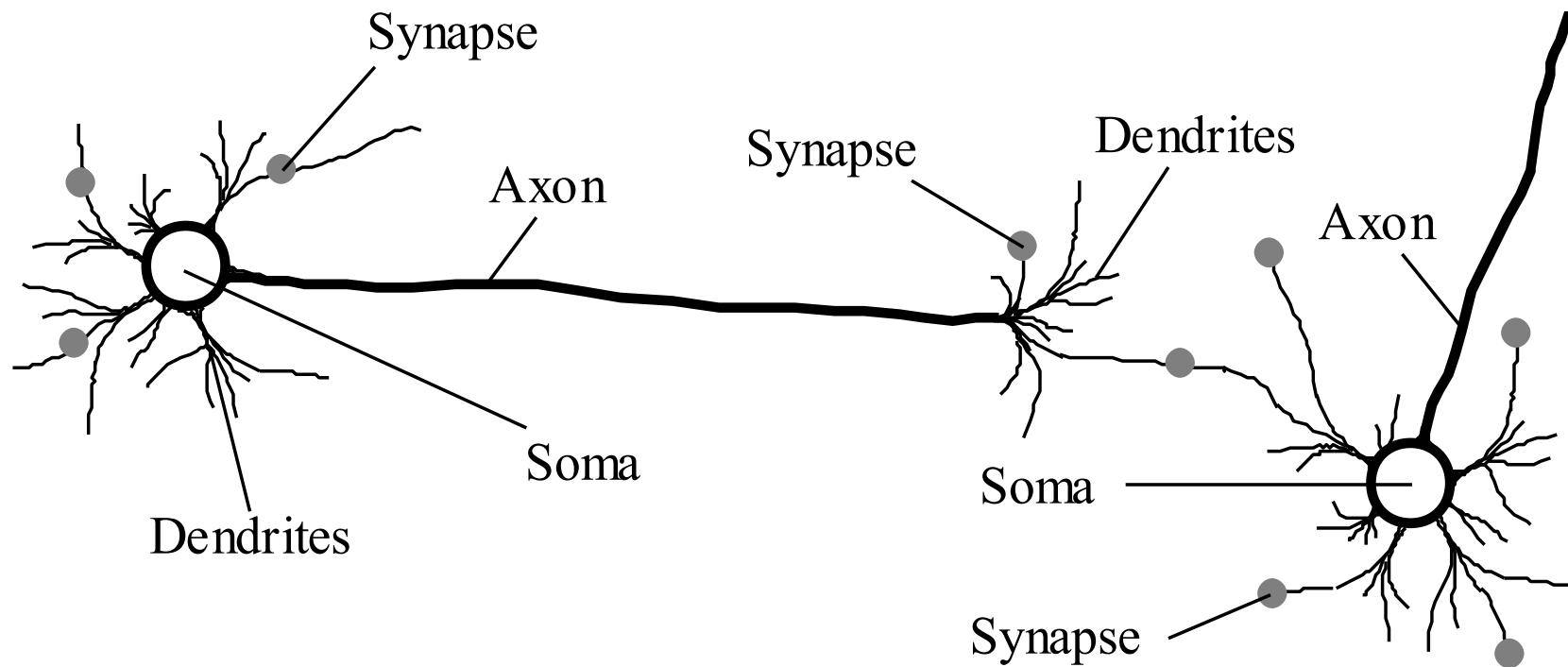
Learning is a fundamental and essential characteristic of biological neural networks. The ease with which they can learn, led to attempts to emulate biological neural networks in a computer.

Biological neural networks



Each of the yellow blobs in the picture above are neuronal cell bodies (soma), and the lines are the input and output channels (dendrites and axons) which connect them.

Schematic of biological neuron



Components of biological neurons

A **biological neuron** consists of the following components:

- **Soma**: Cell body which processes incoming activation signals and converts input into output activations. The nucleus of soma contains the genetic material in the form of DNA.
- **Axon**: Transmission lines that send activation signals to other neurons
- **Dendrites**: Receptive zones that receive activation signals from other neurons
- **Synapses**: Allow weighted signal transmission between the dendrites and axons. Process of transmission is by diffusion of chemicals.

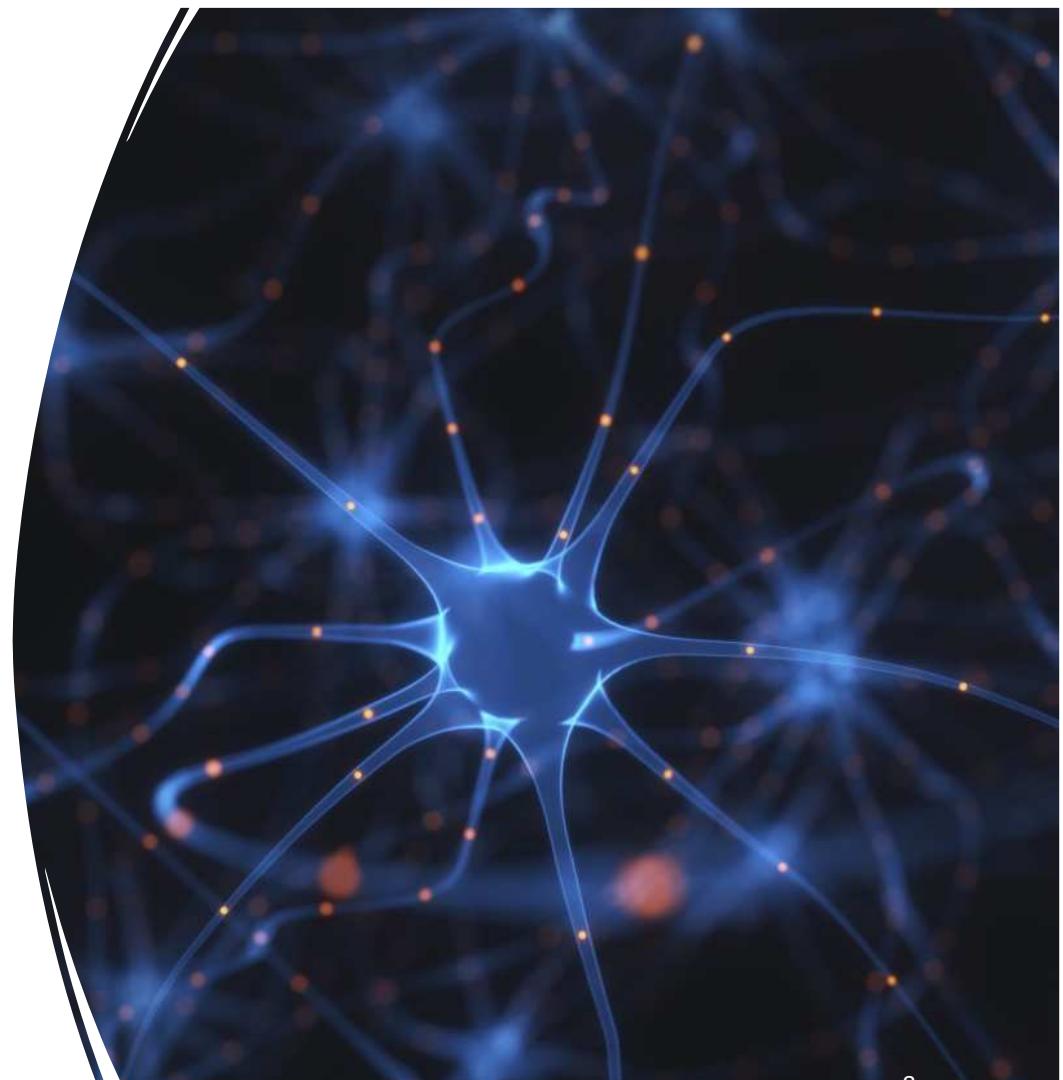
Although neuronal cell body performs majority of cells function, most of the cells total volume is taken up by axons (about 90%).

Signal flow of biological neurons

Each neuron receives electrochemical inputs (**neurotransmitters**) from other neurons at the dendrites.

Neuroreceptors receive such signals and **soma** sums the incoming signals. If the sum of these electrical inputs is sufficiently powerful to activate the neuron, it **transmits an electrochemical signal** along the **axon**, and passes this signal to the other neurons whose **dendrites** are attached at any of the **axon** terminals.

These attached neurons when its internal voltage exceeds a certain **threshold** (-70mv) may then fire. Signals (**action potentials**) flow along the axon as a form of electric pulses and make synapses on other neurons. Note that synapses could be either excitatory or inhibitory.



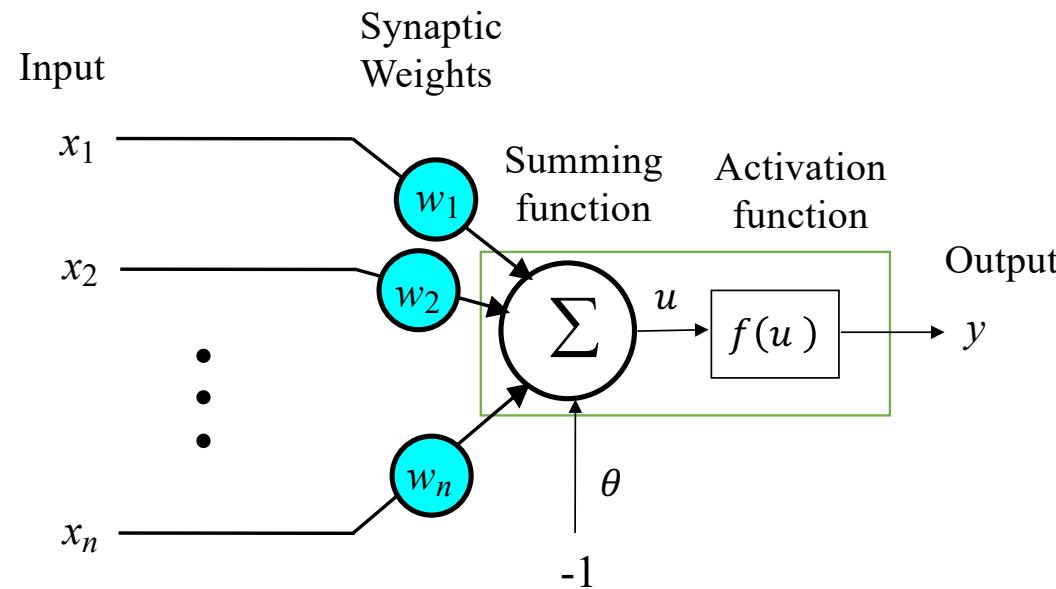
Artificial neural networks

Artificial neural networks attempt to mimic biological neural networks in the brain. The neuronal signals flow along the axon in the form of electric pulses or **action potentials**.

There are two types of artificial neural networks. One that emulates the action potentials are referred to as **spiking neural networks** and the others that emulate the aggregate of action potentials are rate-based or **activation-based neural networks**.

Neural networks discussed in this class are activation-based. However, spiking neural networks are more amenable for hardware implementations.

Artificial neuron model



Input $x = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$

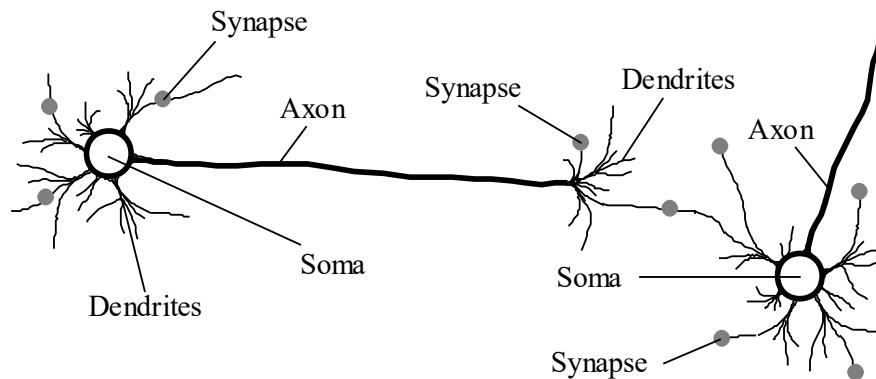
u – total synaptic input

f – activation function

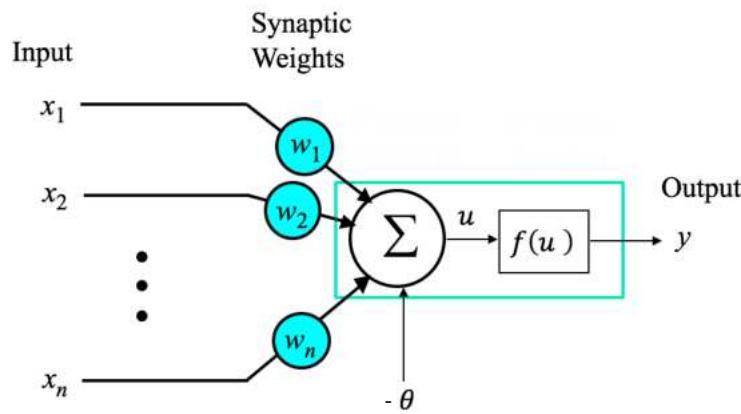
y - output

Analogy between biological and artificial neurons

Biological neuron

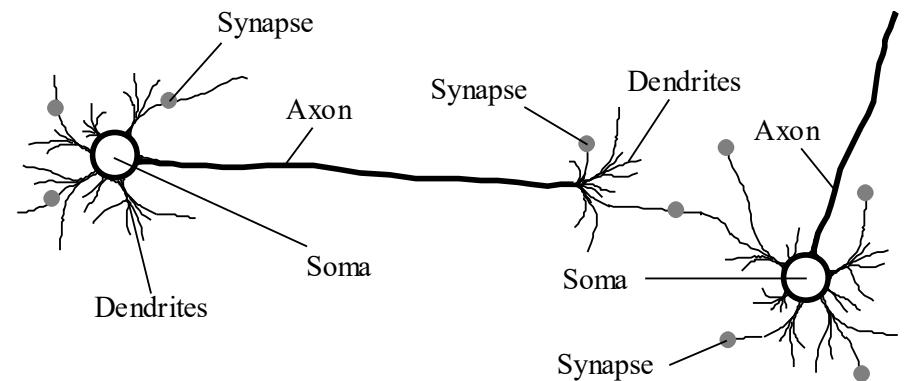


Artificial neuron

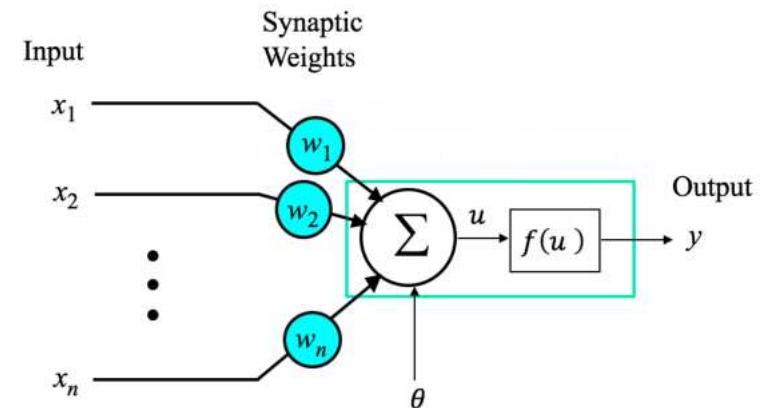


Analogy between biological and artificial neurons

Biological Neuron	Artificial Neuron
Soma	Sum + Activation function
Dendrite	Input
Axon	Output
Synapse	Weight

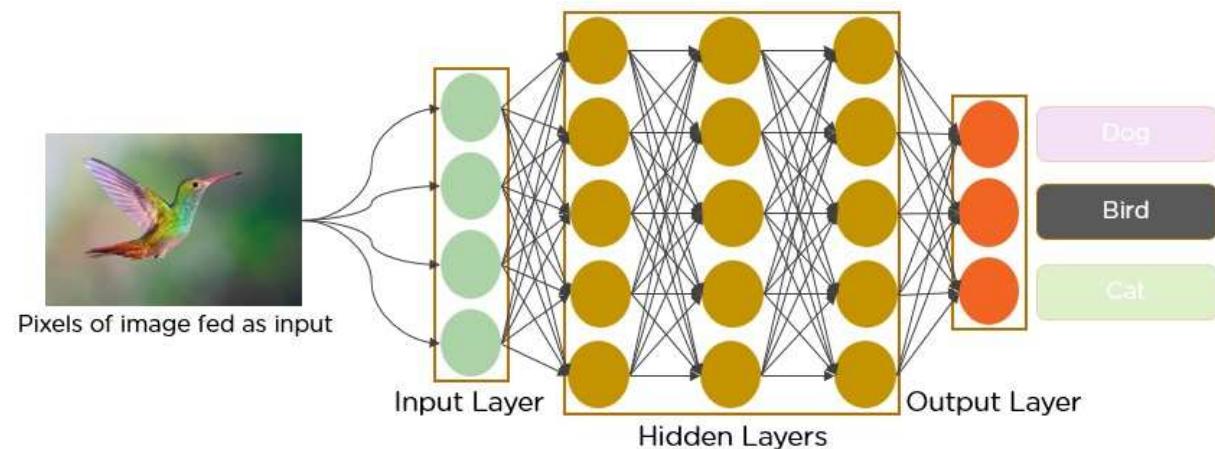


- *McCulloch-Pitts neuron* (~ 1940) is an artificial neuron with binary inputs and outputs
- *Perceptron* (~ 1950) is another name for an artificial neuron with analog inputs

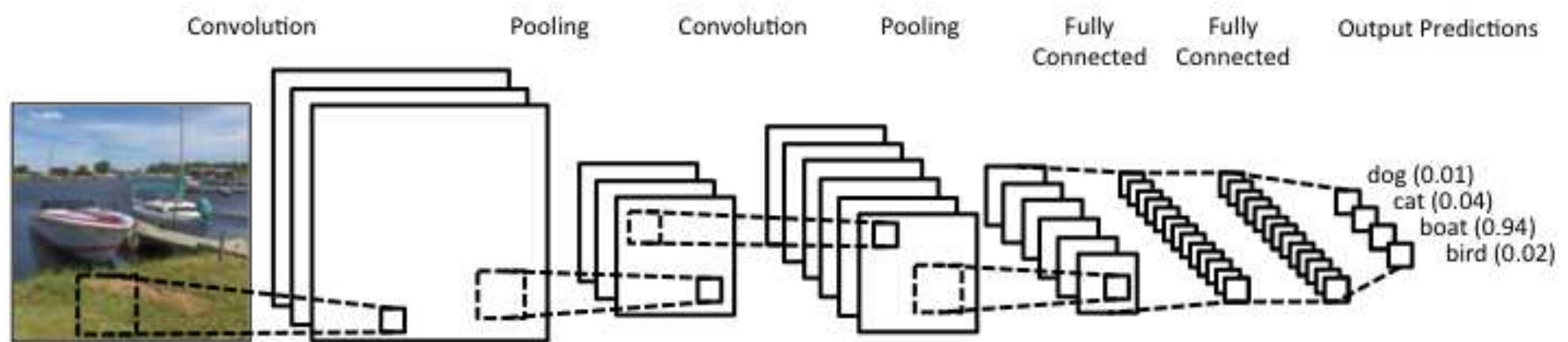


feedforward neural networks

- A **three-layer network** (two hidden layers and output layer).
- The **input layer** consists of input nodes that receive input signals.
- The layers between input layer and **output layer** are referred to as **hidden layers**.
- If the **depth** (the number of layers) is large, feed forward neural networks are referred to as **deep neural networks**.



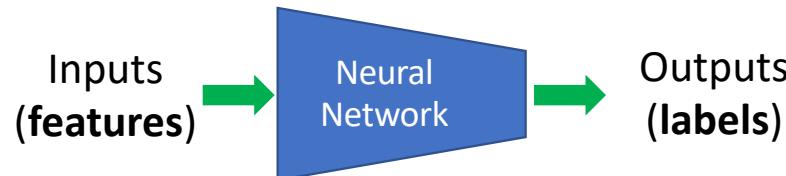
Deep convolutional neural networks



Alternate layers of convolutional and pooling, followed by fully connected layers.

Predictive analytics with neural networks

Neural network is a computational paradigm for machine learning and data analytics.



Neural networks are to be **trained** first with **training data**. Once trained, neural networks are able to **predict** labels for new data.

Predictive analytics:

Regression: outputs are continuous variables: age, height, income, etc.

Classification: outputs are discrete variables: sex, type of flowers, digits, etc.

History

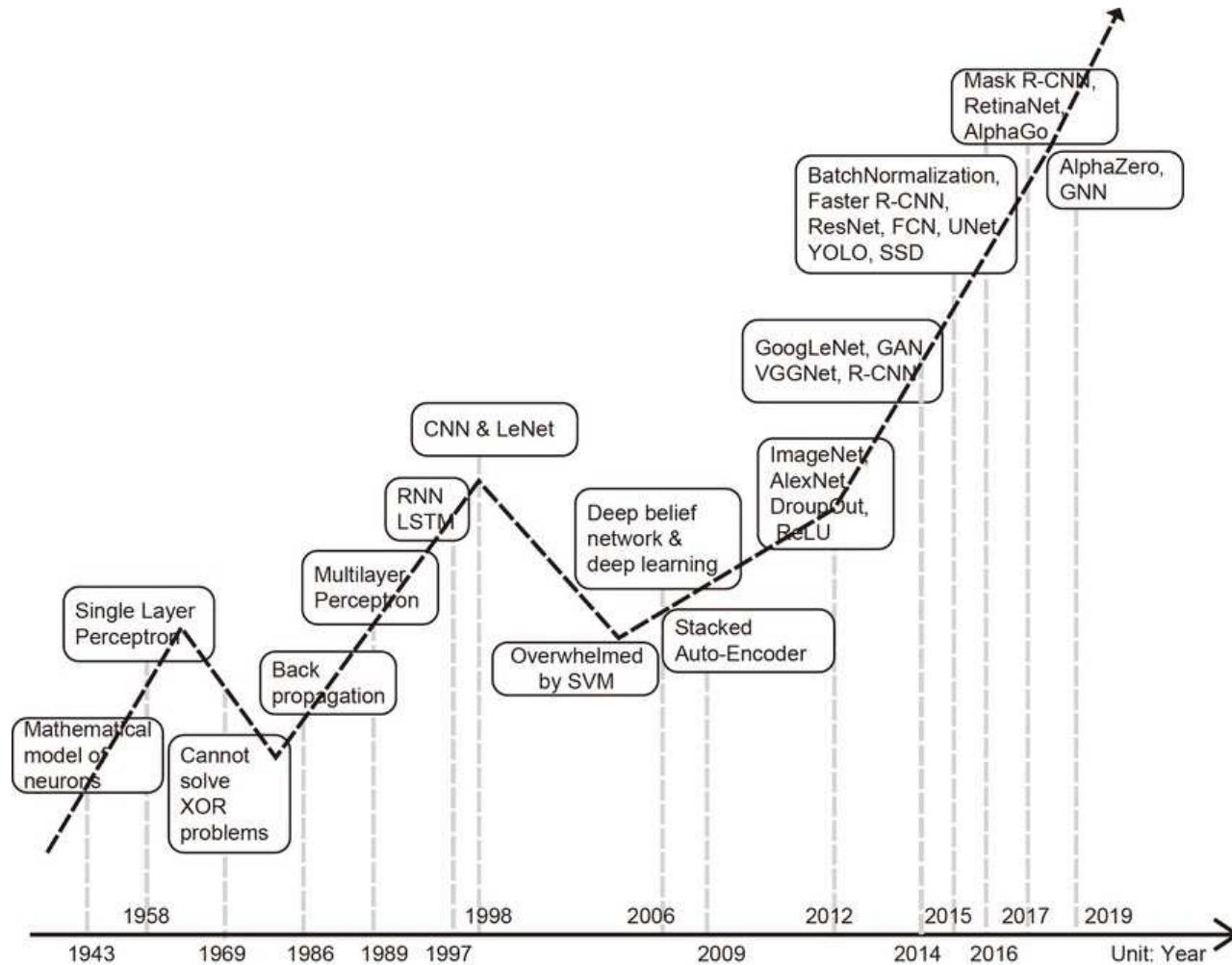
Modern view of Neural Networks (NN) began in the **1940s**. – **Warren McCulloch, Walter Pitts, Donald Hebb**.

By **late Sixties**, most of the basic ideas and concepts necessary for neural computing had already been formulated: **perceptron, gradient descent learning**, etc.

Practical solution emerges for neural networks only in the mid-eighties; for example, **backpropagation algorithm (1985)**.

Major reason for the delay in using large neural networks was technological: no powerful workstations to model and experiment with ANN; algorithms for learning large neural networks were unknown.

Emergence of **deep learning since 2012**: human like performance was achieved in object recognition, using deep convolutional neural networks such as AlexNet.

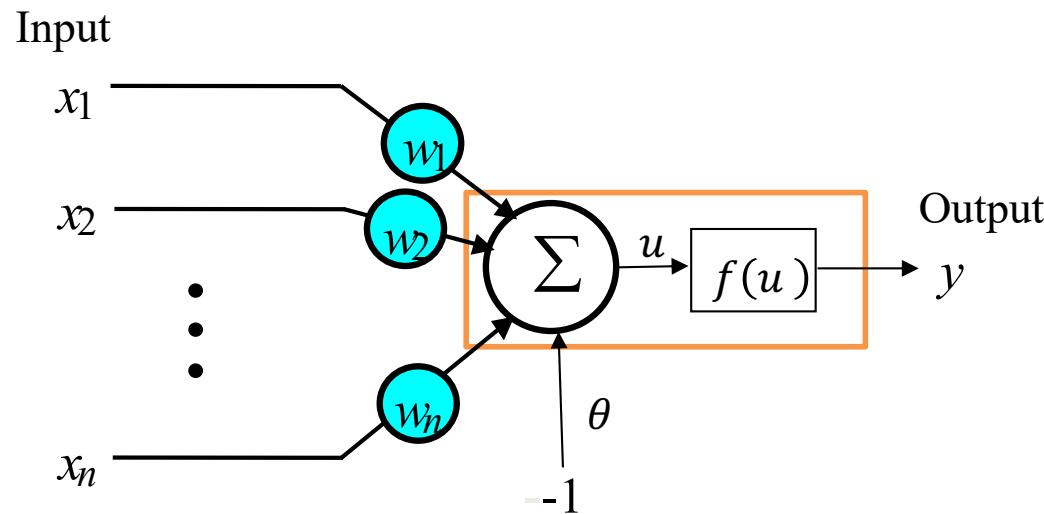


Chapter 1

Fundamentals of Neural Networks

Neural networks and deep learning

Artificial Neuron



Input vector $\mathbf{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$
weight vector $\mathbf{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$
 n is the number of inputs.

Artificial Neuron

An **artificial neuron** is the basic unit of neural networks.

Basic elements of an artificial neuron:

- A set of **input** signals: the input is a vector $\mathbf{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$ where n is the number (or the dimension) of input signals. Inputs are also referred to as **features**.
- Inputs are connected to the neuron via synaptic connections whose strengths are represented by their **weights**.
- The weight vector $\mathbf{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$ where w_i is the synaptic weight connecting i th input of the neuron.

Notation

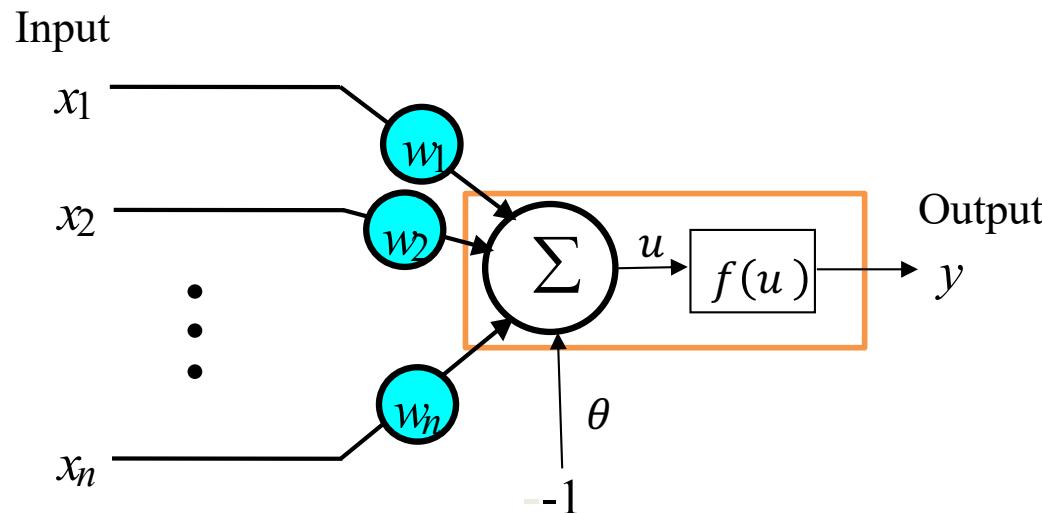
Vectors are denoted in bold and written as horizontally with a transpose (^T).

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$$

Example

$$\mathbf{x} = \begin{pmatrix} 1.2 \\ -0.5 \\ 3.5 \\ 2.1 \end{pmatrix} = (1.2 \quad -0.5 \quad 3.5 \quad 2.1)^T$$
$$\mathbf{x}^T = (1.2 \quad -0.5 \quad 3.5 \quad 2.1)$$

Artificial Neuron



The total **synaptic input** u to the neuron is given by the sum of the products of the inputs and their corresponding connecting weights minus the **threshold** of the neuron.

The total synaptic input to a neuron, u is given by

$$u = w_1x_1 + w_2x_2 + \cdots w_nx_n - \theta = \sum_{i=1}^n w_i x_i - \theta$$

where θ is the threshold of the neuron.

By using vector notations:

$$u = \mathbf{w}^T \mathbf{x} - \theta$$

Artificial Neuron

The **activation function** f relates synaptic input to the activation of the neuron.

$f(u)$ denotes the **activation** of the neuron.

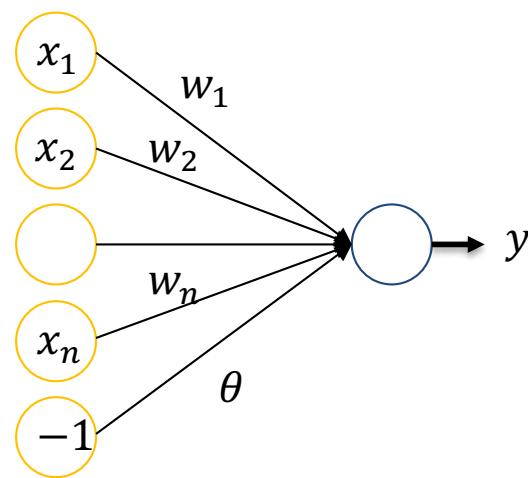
For some neurons, the **output** y is equal to the activation of the neuron.

$$y = f(u)$$

Note that activation is not generally equal to the output of the neuron.

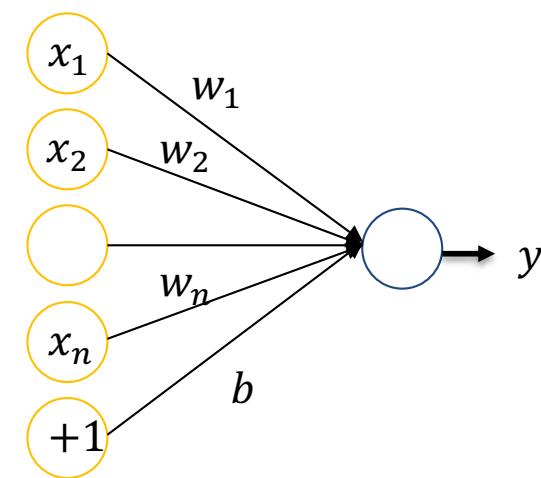
Bias vs Threshold

The threshold is often considered as a weight with a fixed input of -1 . Often the threshold is represented as a **bias** b that receives constant $+1$ input.



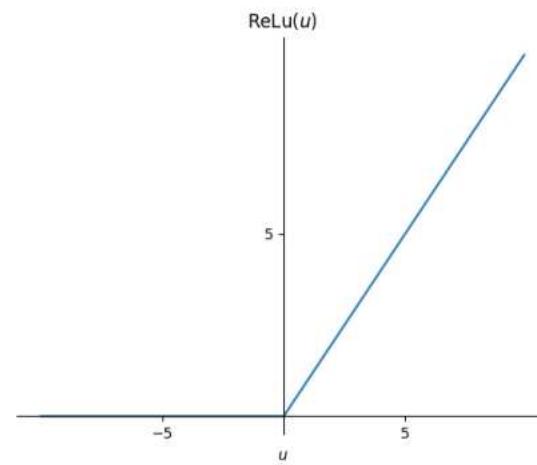
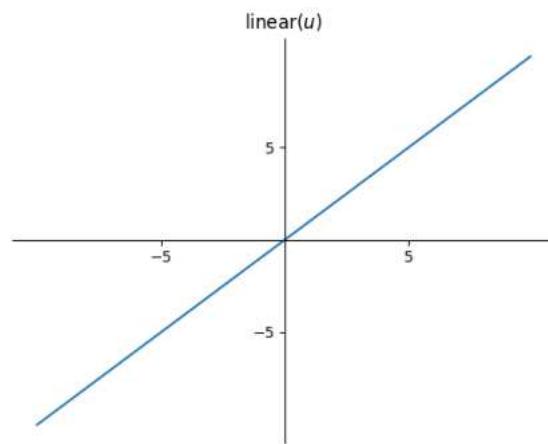
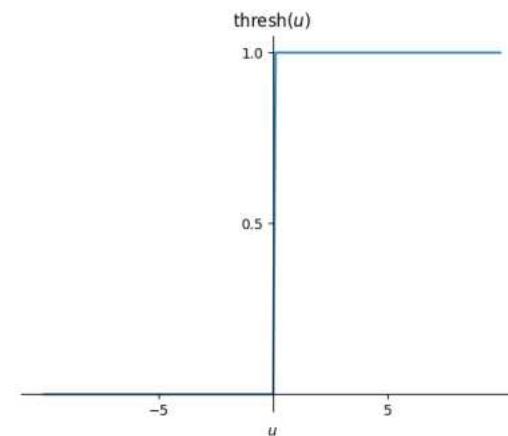
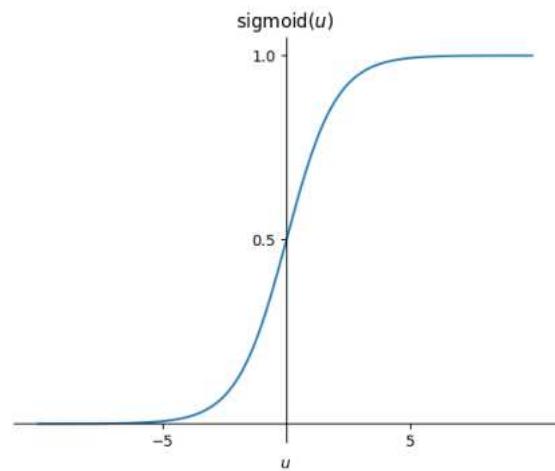
Threshold
 $u = \mathbf{w}^T \mathbf{x} - \theta$
 $y = f(u)$

Bias $b = -\theta$



Bias
 $u = \mathbf{w}^T \mathbf{x} + b$
 $y = f(u)$

Activation functions



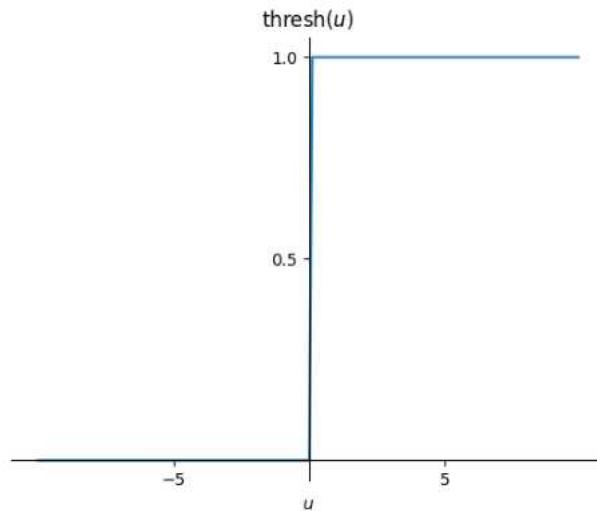
Activation functions

For **threshold (unit step) activation function**, the activation is given by

$$f(u) = \text{threshold}(u) = 1(u > 0)$$

where $1(\cdot)$ is the *Indictor function* or *Unit-step function*:

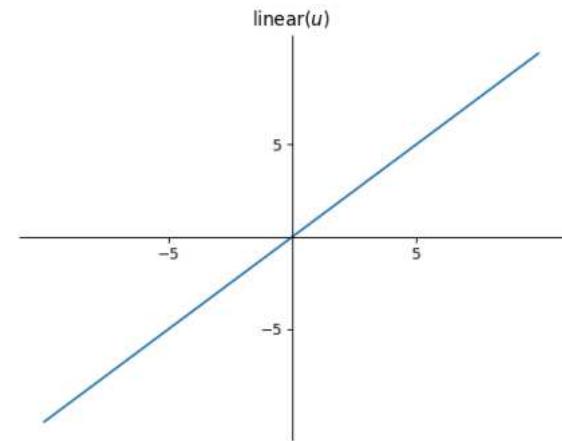
$$1(x) = \begin{cases} 1, & x \text{ is True} \\ 0, & x \text{ is False} \end{cases}$$



Activation functions

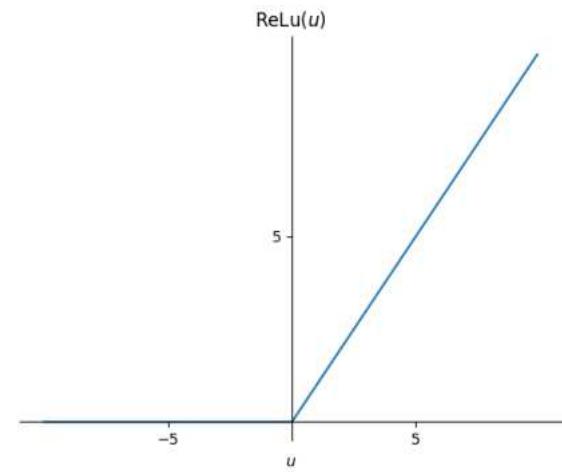
A neuron with *linear* activation function can be written as

$$f(u) = \text{linear}(u) = u$$



The **ReLU** (rectified-linear unit) activation function can be written as

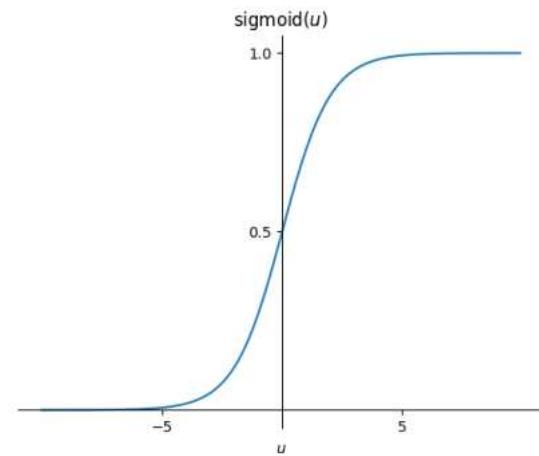
$$f(u) = \text{relu}(u) = \max\{0, u\}$$



Sigmoid activation function

The sigmoidal is known as the **logistic function** or simply **sigmoid function**

$$f(u) = \text{sigmoid}(u) = \frac{1}{1 + e^{-u}}$$



In general, the **sigmoid** activation function can be written as

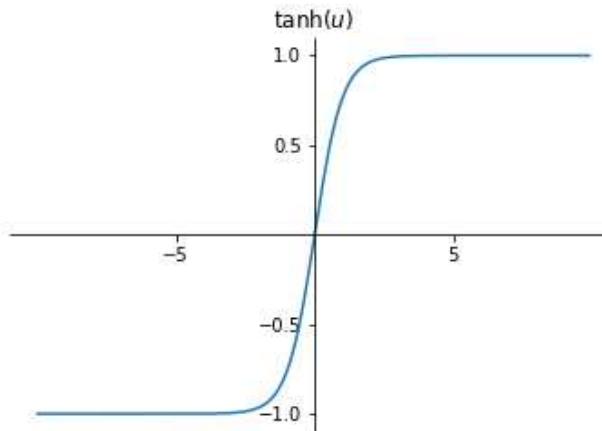
$$f(u) = \frac{a}{1 + e^{-bu}}$$

a is the gain (amplitude) and b is the slope.

But often, $a = 1.0$ and $b = 1.0$.

Tanh activation function

$$f(u) = \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$



Tanh activation function has the same shape as sigmoidal and spans from -1 and +1. It is also known as **bipolar sigmoidal**.

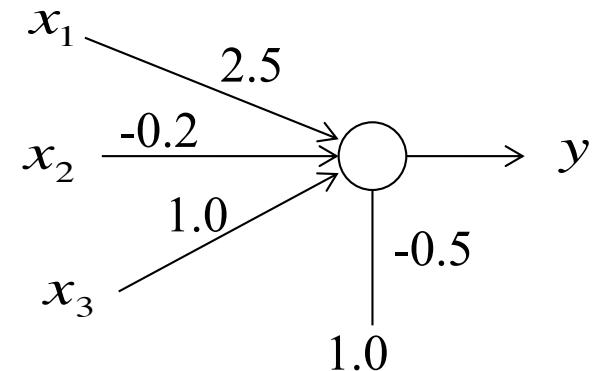
Sigmoidal is the most pervasive and biologically plausible activation function. Since sigmoid function is *differentiable*, it leads to mathematically attractive neuronal models.

Example 1

The artificial neuron in the figure receives 3-dimensional inputs $x = (x_1 \ x_2 \ x_3)^T$ and has an activation function given by

$$f(u) = \frac{0.8}{1+e^{-1.2u}}.$$

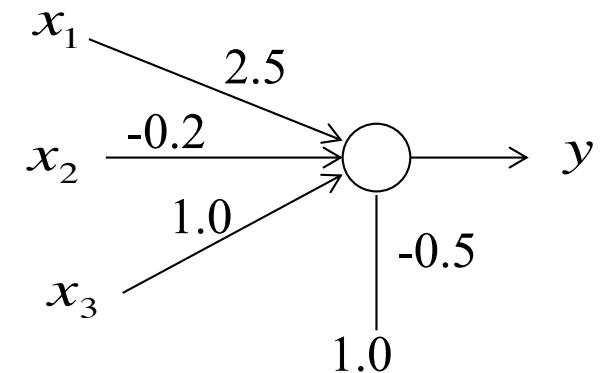
Find the synaptic input and the output of the neuron for inputs: $\begin{pmatrix} 0.8 \\ 2.0 \\ -0.5 \end{pmatrix}$ and $\begin{pmatrix} -0.4 \\ 1.5 \\ 1.0 \end{pmatrix}$.



Example 1

$$\mathbf{w} = \begin{pmatrix} 2.5 \\ -0.2 \\ 1.0 \end{pmatrix}, \quad b = -0.5$$

Consider $\mathbf{x} = \begin{pmatrix} 0.8 \\ 2.0 \\ -0.5 \end{pmatrix}$



Synaptic input $u = \mathbf{w}^T \mathbf{x} + b = (2.5 \quad -0.2 \quad 1.0) \begin{pmatrix} 0.8 \\ 2.0 \\ -0.5 \end{pmatrix} - 0.5 = 0.6$

Output $y = f(u) = \frac{0.8}{1+e^{-1.2u}} = \frac{0.8}{1+e^{-1.2 \times 0.6}} = 0.538$

Similarly, for $\mathbf{x} = \begin{pmatrix} -0.4 \\ 1.5 \\ 1.0 \end{pmatrix}$, $u = -0.8$ and output $y = f(u) = 0.222$

PyTorch 2.0

PyTorch/Tensorflow is about processing of **tensors**. Tensor is a multidimensional array.

Rank refers to the number of dimensions and **shape** gives the sizes of each dimension of the tensor.

3. # a rank 0 tensor; a scalar with shape [],

[1., 2., 3.] # a rank 1 tensor; a vector with shape [3]

[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]

[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]

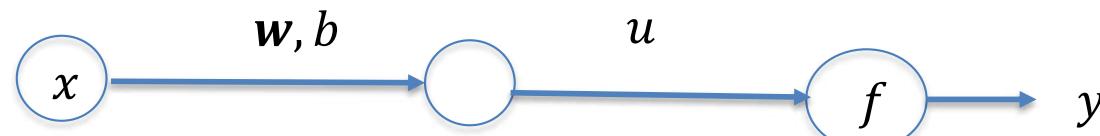
Computational Graph

PyTorch program involves building and evaluation of a **computational graph**.

A computational graph is a series of tensor **operations** arranged into a graph.

- Nodes of the graph represent tensor operations
- Edges represent values (tensors) that follow through the graph

Computational graph of a neuron



multiply
add

$$u = \mathbf{w}^T \mathbf{x} + b$$
$$y = f(u)$$

Torch Implementation of Example 1

```
import torch

# a class for neuron
class Neuron():
    # initiate a neuron class with weights and biases (initiate the object)
    def __init__(self):
        self.w = torch.tensor([2.5, -0.2, 1.0])
        self.b = torch.tensor(-0.5)

    # evaluate the neuron (implement a function)
    def __call__(self, x):
        u = torch.inner(self.w, x) + self.b
        y = 0.8/(1+torch.exp(-1.2*u))
        return u, y

# create a neuron
neuron = Neuron()

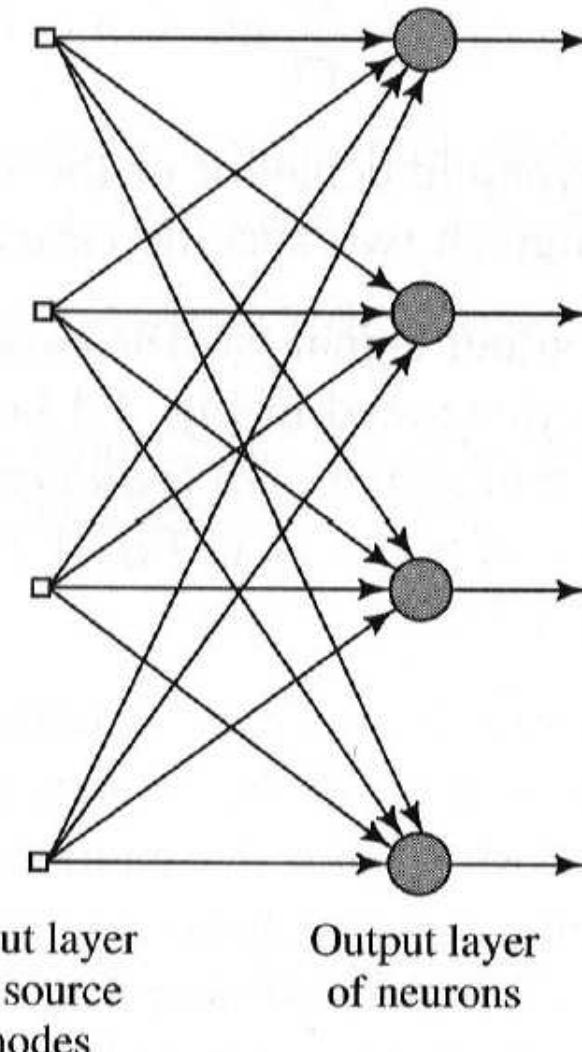
# evaluate
u, y = neuron(torch.tensor([0.8, 2.0, -0.5]))

# print: u = 0.600, y = 0.538
```

NN Architectures: neuron layers

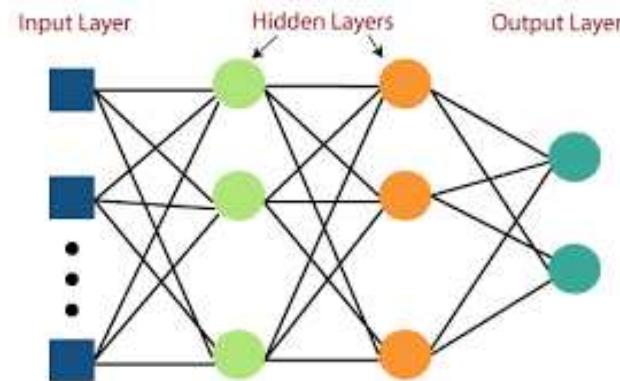
Single – layer of neurons

- Comprised of an input layer of source units that inject into an output layer of neurons.
- A fully-connected layer



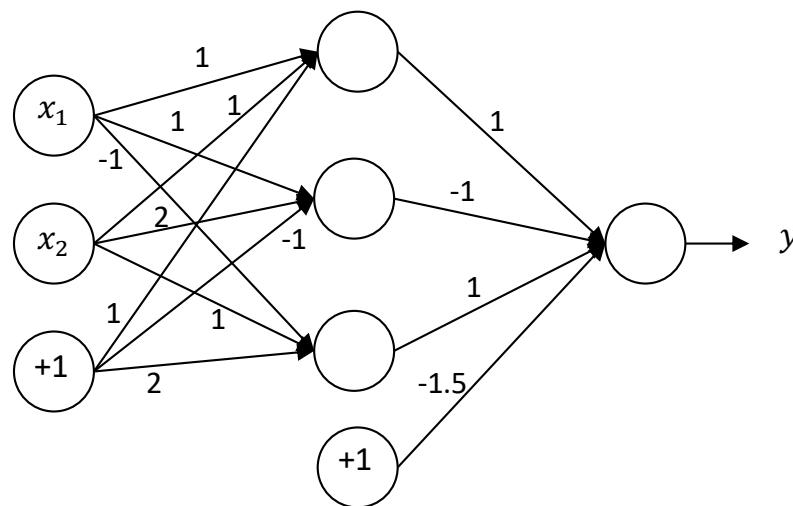
NN architectures: multilayer feedforward networks

- Comprised of more than one layer of neurons. Layers between input source nodes and output layer is referred to as *hidden layers*.
- Multilayer neural networks can handle *more complicated* and *larger scale problems* better than single-layer networks.
- However, training multilayer network may be *more difficult* and *time-consuming*.



Three-layer network

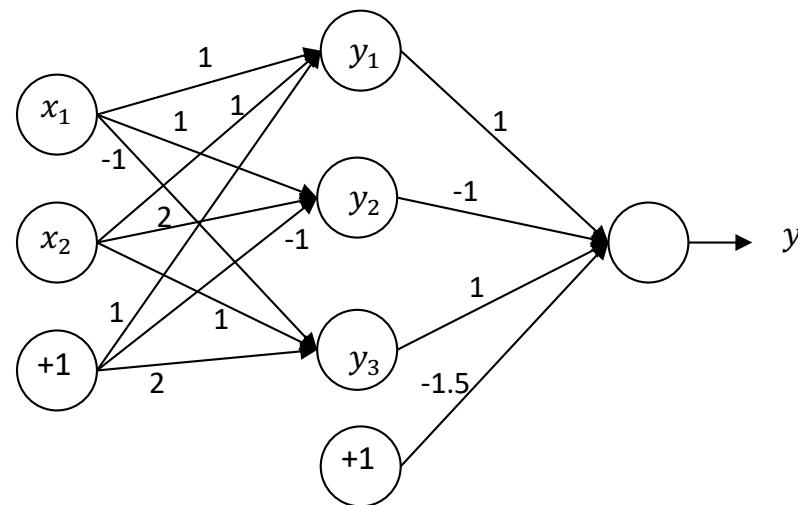
Example 2



Two-layer neural network receives 2-dimensional inputs $(x_1, x_2) \in \mathbb{R}^2$ and has one output neuron and three hidden neurons. All the neurons have unit step activation functions. The weights of the connections are given in the figure. Find the space of inputs for which the output $y = 1.0$.

Find the output for inputs $(0.0, 0.0)$, $(2.0, 2.0)$, and $(-1.0, 1.0)$

Example 2



Synaptic inputs:

$$u_1 = x_1 + x_2 + 1$$

$$\text{Output } y_1 = 1(u_1 > 0)$$

$$u_2 = x_1 + 2x_2 - 1$$

$$y_2 = 1(u_2 > 0)$$

$$u_3 = -x_1 + x_2 + 2$$

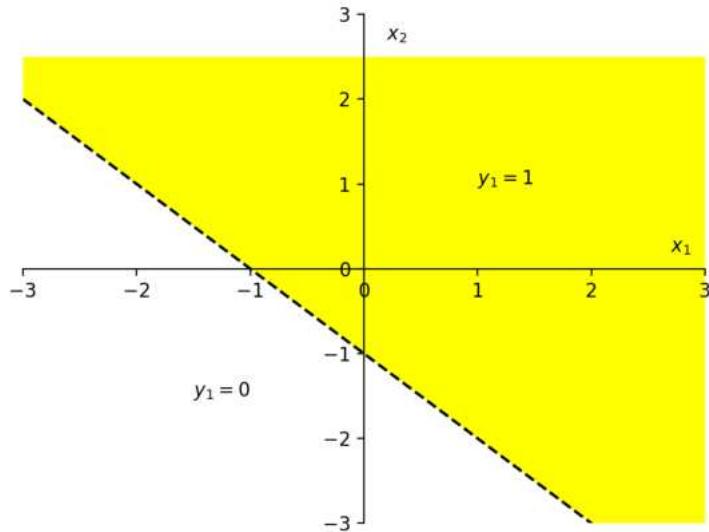
$$y_3 = 1(u_3 > 0)$$

$$u = y_1 - y_2 + y_3 - 1.5$$

$$y = 1(u > 0)$$

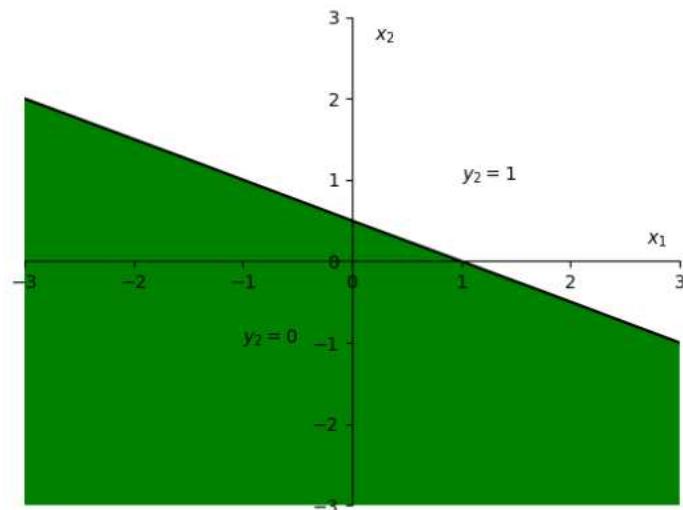
$$y_1 = f(u_1) = 1(u_1 > 0)$$

Boundary: $u_1 = x_1 + x_2 + 1 = 0 \rightarrow x_2 = -x_1 - 1$

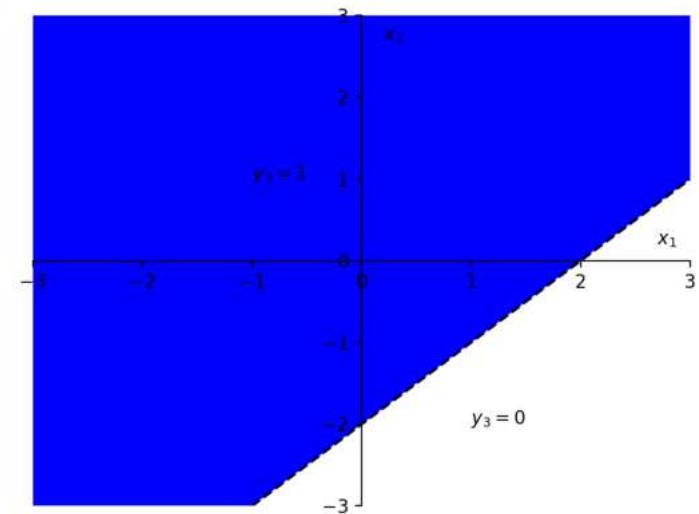


The boundary line is obtained by setting $u_1 = 0$; and for one side of the boundary, $y = 1$ and on other side $y_1 = 0$.

$$u_2 = 2x_2 + x_1 - 1 = 0 \rightarrow x_2 = -0.5x_1 + 0.5$$



$$u_3 = x_2 - x_1 + 2 = 0 \rightarrow x_2 = x_1 - 2$$



Output layer neuron:

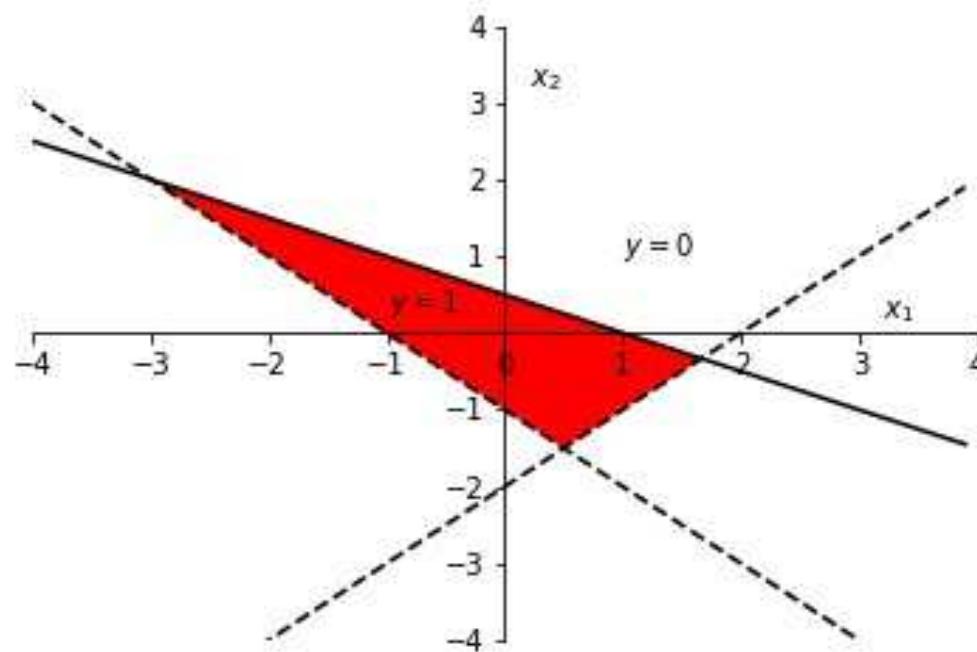
$$u = y_1 - y_2 + y_3 - 1.5$$
$$y = 1(u > 0)$$

Note that $y_1, y_2, y_3 \in \{0, 1\}$

y_1	y_2	y_3	u	y
0	0	0	-1.5	0
0	0	1	-0.5	0
0	1	0	-2.5	0
0	1	1	-1.5	0
1	0	0	-0.5	0
1	0	1	0.5	1
1	1	0	-1.5	0
1	1	1	-0.5	0

$$Y = Y_1 \bar{Y}_2 Y_3$$

$y = 1$ region is given by the intersection of regions: $y_1=1$, $y_2=0$, and $y_3=1$.



$$x = (0.0, 0.0) \rightarrow y = 1$$

$$x = (2.0, 2.0) \rightarrow y = 0$$

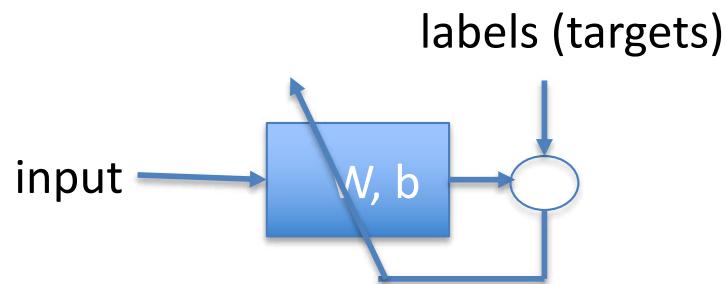
$$x = (-1.0, 1.0) \rightarrow y = 1$$

Note that networks of *discrete perceptrons* (neurons with threshold activation functions) can implement Boolean functions.

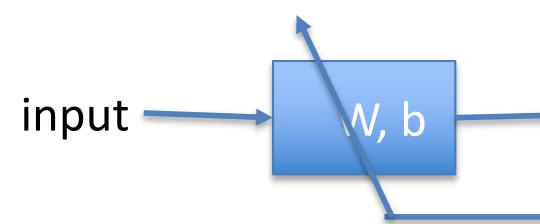
Training (or learning) of neural networks

Neural networks attain their operating characteristics through **learning** (or **training**) with training examples. During training, the weights or the strengths of connections are gradually adjusted iteratively to achieve their desirable labels (or **targets**).

Training may be either **supervised** or **unsupervised**.



Supervised Learning



Unsupervised Learning

Supervised and unsupervised learning

Supervised Learning:

For each training input pattern, the network is presented with the correct **target label** (the desired output).

Unsupervised Learning:

For each training input pattern, the network adjusts weights *without knowing* the correct target.

In unsupervised training, the network **self-organizes** to classify similar input patterns into clusters.

Supervised learning

Learning of a neuron or neural network is usually performed in order to minimize a **cost function (loss function or error function)**.

The cost function $J(\mathbf{W}, \mathbf{b})$ of an artificial neuron is typically a multi-dimensional function that depends on weights \mathbf{W} and the biases \mathbf{b} . The neuron learning attempts to find the optimal weights \mathbf{W}^* and biases \mathbf{b}^* that minimize the error function:

$$\mathbf{W}^*, \mathbf{b}^* = \arg \min_{\mathbf{W}, \mathbf{b}} J(\mathbf{W}, \mathbf{b})$$

Given a set of training patterns, the parameters (weights and biases) minimizing cost function are learned in an iterative procedure. In each iteration, small changes of weights $\Delta\mathbf{W}$ and biases $\Delta\mathbf{b}$ are made:

$$\mathbf{W} \leftarrow \mathbf{W} + \Delta\mathbf{W}$$

$$\mathbf{b} \leftarrow \mathbf{b} + \Delta\mathbf{b}$$

Gradient descent learning

The grading descent procedure states that the weights \mathbf{W} (and biases \mathbf{b}) are updated during learning by searching in the direction of and proportional the **negative gradient** of the cost function.

That is, the change of the weight vector:

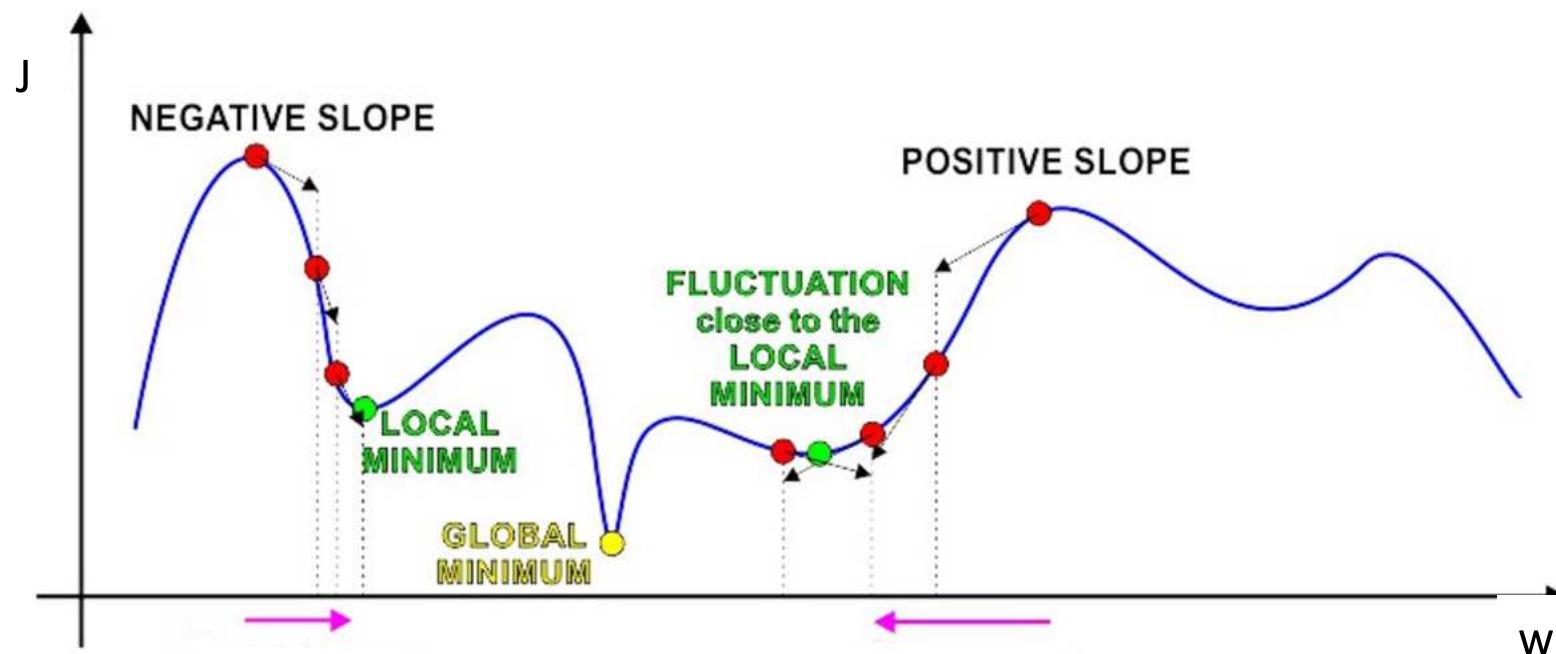
$$\Delta \mathbf{W} \propto -\frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{w}}$$
$$\Delta \mathbf{W} = -\alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$$

where $\frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$ is the gradient (partial derivative) of cost with respect to weight \mathbf{W} and α is *learning factor* or *learning rate*. $\alpha \in (0.0, 1.0]$.

The **gradient descent equations** for learning the weights is given by substituting above in

$$\mathbf{W} \leftarrow \mathbf{W} + \Delta \mathbf{W}$$

Gradient Descent Leaning



Gradient descent learning

The **gradient descent equations** for the weights is given by

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$$

Similarly, for the bias

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}$$

Notation: $\nabla_{\mathbf{W}} J = \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}$ and $\nabla_{\mathbf{b}} J = \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}$.

Gradient descent learning is given by

$$\begin{aligned}\mathbf{W} &\leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J \\ \mathbf{b} &\leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J\end{aligned}$$

Note: Will drop the arguments in J .

Gradient descent learning

Given a set of training examples

Initialize weight \mathbf{W} and bias \mathbf{b}

Set the learning parameter α

Iterate until convergence:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$

Convergence is achieved by observing one of the following:

1. No changes in weights and biases
2. No difference between the outputs and targets
3. No decrease in the cost function J

Training dataset

Training data are also referred to as **training examples** or **training patterns**. For supervised learning, a training pattern consists of a pair consisting of input pattern and the corresponding target.

A training dataset is a set of training examples: $\{(x_p, d_p)\}_{p=1}^P$ or $\{(x_1, d_1), (x_2, d_2), \dots (x_P, d_P)\}$

x_p is the input (features) and d_p is the target (desired label) of p th training pattern. P is the number of examples in the dataset.

The input is usually n -dimensional and written as:

$$x_p = (x_{p1}, x_{p2}, \dots x_{pn})^T$$

Stochastic Gradient Descent (SGD) learning

Given training examples $\{(x_p, d_p)\}_{p=1}^P$

Set learning factor α

Initialize (W, b)

Iterate until convergence:

for each pattern (x_p, d_p) :

$$W \leftarrow W - \alpha \nabla_W J_p$$

$$b \leftarrow b - \alpha \nabla_b J_p$$

- In each **epoch** or cycle of iteration, the learning takes place individually over every pattern
- The **cost** J_p is individually computed from the output and the target of the p th training pattern.

Batches of Data

Inputs are often presented as a batch in a **data matrix X** and a **target vector d** . The input datapoints (or patterns) are written as rows in the data matrix and the targets are written into a single vector in the target vector.

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$.

Data matrix:

$$X = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_P^T \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{P1} & x_{P2} & \cdots & x_{Pn} \end{pmatrix}$$

Target vector:

$$d = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$$

(Batch) Gradient descent learning

Given a set of training patterns: (X, d)

Set learning factor α

Initialize (W, b)

Iterate until convergence:

$$W \leftarrow W - \alpha \nabla_W J$$

$$b \leftarrow b - \alpha \nabla_b J$$

The cost J is computed using all the training patterns.

That is, using (X, d)

In each epoch, the weights are updated once considering all the input patterns.

Summary

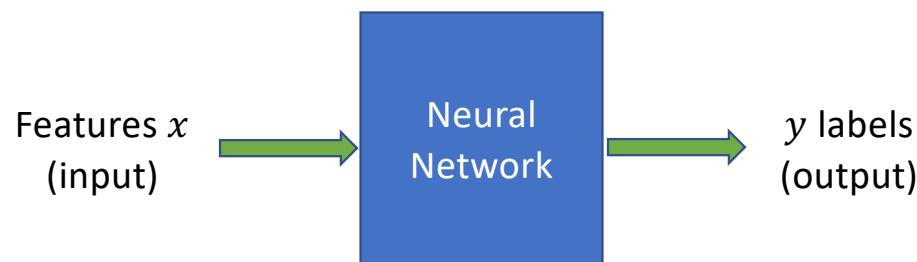
- Analogy between biological and artificial neurons
- Transfer function of artificial neuron:
$$u = \mathbf{w}^T \mathbf{x} + b$$
$$y = f(u)$$
- Types of activation functions: sigmoid, threshold, linear, ReLU, and tanh.
- Given inputs, to find the outputs for simple feedforward networks
- Supervised and unsupervised learning
- Gradient descent learning:
$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$$
$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$
- Stochastic gradient descent (SGD) and batch gradient descent (GD)

Chapter 2

Regression and classification

Neural networks and deep learning

Regression and classification



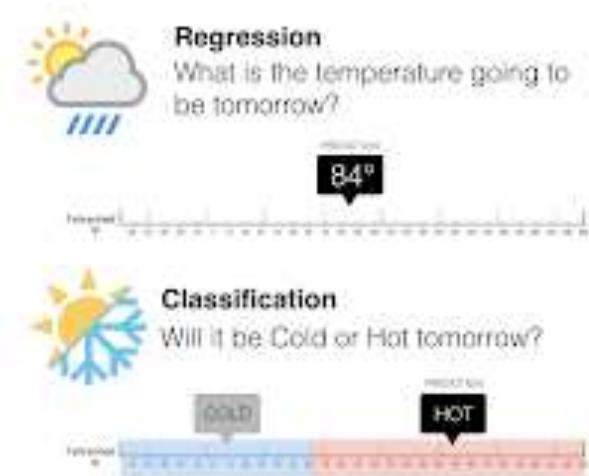
Primarily, neural networks are used to *predict* output **labels** from input **features**.

Prediction tasks can be classified into two categories:

Regression: the labels are continuous (age, income, height, etc.)

Classification: the labels are discrete (sex, digits, type of flowers, etc.).

Training finds network weights and biases that are optimal for **prediction** of labels from features.



Linear neuron

Synaptic input u to a neuron is given by

$$u = \mathbf{w}^T \mathbf{x} + b$$

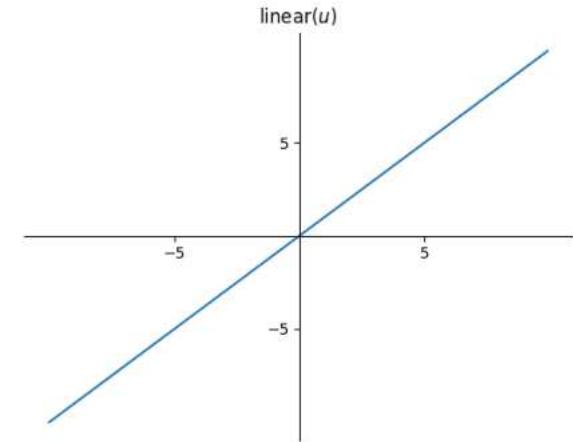
A linear neuron has a *linear activation function*. That is,

$$y = f(u) = u$$

A linear neuron with weights $\mathbf{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$ and bias b has an output:

$$y = \mathbf{w}^T \mathbf{x} + b$$

where input $\mathbf{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T \in \mathbf{R}^n$ and output $y \in \mathbf{R}$.



Linear neuron performs linear regression

Representing a dependent (output) variable as a linear combination of independent (input) variables is known as **linear regression**.

The output of a linear neuron can be written as

$$y = w_1x_1 + w_2x_2 \cdots + w_nx_n + b$$

where x_1, x_2, \dots, x_n are the inputs. That is, a linear neuron performs linear regression and the weights and biases (that is, b and w_1, \dots, w_n) act as regression coefficients. The above function forms a **hyperplane** in Euclidean space \mathbf{R}^n .

Given a training examples $\{(x_p, d_p)\}_{p=1}^P$ where input $x_p \in \mathbf{R}^n$ and target $d_p \in \mathbf{R}$, training a linear neuron finds a linear mapping $\phi: \mathbf{R}^n \rightarrow \mathbf{R}$ given by:

$$y = \mathbf{w}^T \mathbf{x} + b$$

Stochastic gradient descent (SGD) for linear neuron

The **cost function** J for regression is usually given as the *square error* (s.e.) between neuron outputs and targets.

Given a training pattern (\mathbf{x}, d) , $\frac{1}{2}$ square error cost J is defined as

$$J = \frac{1}{2} (d - y)^2$$

where y is neuron output for input pattern \mathbf{x} and d is the target label.

$$y = \mathbf{w}^T \mathbf{x} + b$$

The $\frac{1}{2}$ in the cost function is introduced to simplify learning equations and does not affect the optimal values of the parameters (weights and bias).

SGD for linear neuron



$$J = \frac{1}{2}(d - y)^2$$
$$y = u = \mathbf{w}^T \mathbf{x} + b$$

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} = -(d - y) \quad (\text{A})$$

$$\nabla_{\mathbf{w}} J = \frac{\partial J}{\partial \mathbf{w}} = \frac{\partial J}{\partial u} \frac{\partial u}{\partial \mathbf{w}} \quad (\text{B})$$

SGD for linear neuron

$$u = \mathbf{w}^T \mathbf{x} + b = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

$$\frac{\partial u}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial u}{\partial w_1} \\ \frac{\partial u}{\partial w_2} \\ \vdots \\ \frac{\partial u}{\partial w_n} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \mathbf{x} \quad (\text{C})$$

$$\frac{\partial u}{\partial \mathbf{w}} = \mathbf{x}$$

Substituting (A) and (C) in (B),

$$\nabla_{\mathbf{w}} J = -(d - y) \mathbf{x} \quad (\text{D})$$

Similarly, since $\frac{\partial u}{\partial b} = 1$,

$$\nabla_b J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial b} = -(d - y) \quad (\text{E})$$

SGD for linear neuron

Gradient learning equations:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J \\ b &\leftarrow b - \alpha \nabla_b J\end{aligned}$$

By substituting from (D) and (E), SGD equations for a linear neuron are given by

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} + \alpha(d - y)\mathbf{x} \\ b &\leftarrow b + \alpha(d - y)\end{aligned}$$

SGD algorithm for linear neuron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$

Set learning parameter α

Initialize w and b

Repeat until convergence:

For every training pattern (x_p, d_p) :

$$y_p = w^T x_p + b$$

$$w \leftarrow w + \alpha(d_p - y_p)x_p$$

$$b \leftarrow b + \alpha(d_p - y_p)$$

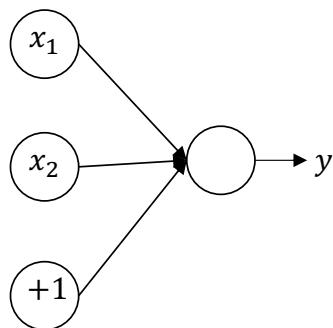
Example 1: SGD on a linear neuron

Train a linear neuron to perform the following mapping, using stochastic gradient descent (SGD) learning:

$x^T = (x_1, x_2)$	d
(0.54, -0.96)	1.33
(0.27, 0.50)	0.45
(0.00, -0.55)	0.56
(-0.60, 0.52)	-1.66
(-0.66, -0.82)	-1.07
(0.37, 0.91)	0.30

Use a learning factor $\alpha = 0.01$.

Example 1



Let's initialize weights randomly and biases to zeros

$$\mathbf{w} = \begin{pmatrix} 0.92 \\ 0.71 \end{pmatrix} \text{ and } b = 0.0$$

$$\alpha = 0.01$$

Need to shuffle the patterns before every epoch.

Example 1: epoch 1

Epoch 1 begins

Shuffle the patterns

First pattern $p = 1$ is applied:

$$\mathbf{x}_p = \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix} \text{ and } d_p = 1.33$$

$$y_p = \mathbf{w}^T \mathbf{x}_p + b = (0.92 \quad 0.71) \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix} + 0.0 = -0.19$$

$$\text{s.e.} = (d_p - y_p)^2 = 2.292$$

$$\mathbf{w} = \mathbf{w} + \alpha(d_p - y_p)\mathbf{x}_p = \begin{pmatrix} 0.92 \\ 0.71 \end{pmatrix} + 0.01 \times (1.33 + 0.19) \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix} = \begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$$

$$b = b + \alpha(d_p - y_p) = 0 + 0.01 \times (1.33 + 0.19) = 0.02$$

Example 1: epoch 1

Second pattern $p = 2$ is applied:

$$\mathbf{x}_p = \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix} \text{ and } d_p = -1.07$$

$$y_p = \mathbf{w}^T \mathbf{x}_p + b = (0.93 \quad 0.70) \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix} + 0.02 = -0.19$$

$$\text{s.e.} = (d_p - y_p)^2 = 0.01$$

$$\mathbf{w} = \mathbf{w} + \alpha(d_p - y_p)\mathbf{x}_p = \begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix} + 0.01 \times (-1.07 + 0.19) \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix} = \begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$$

$$b = b + \alpha(d_p - y_p) = 0.02 + 0.01 \times (1.33 + 0.19) = 0.02$$

Iterations continues for patterns $p = 3, \dots, 6$.

the second epoch starts

Shuffle the patterns

Apply patterns $p = 1, 2, \dots, 6$

Training epochs continue until convergence.

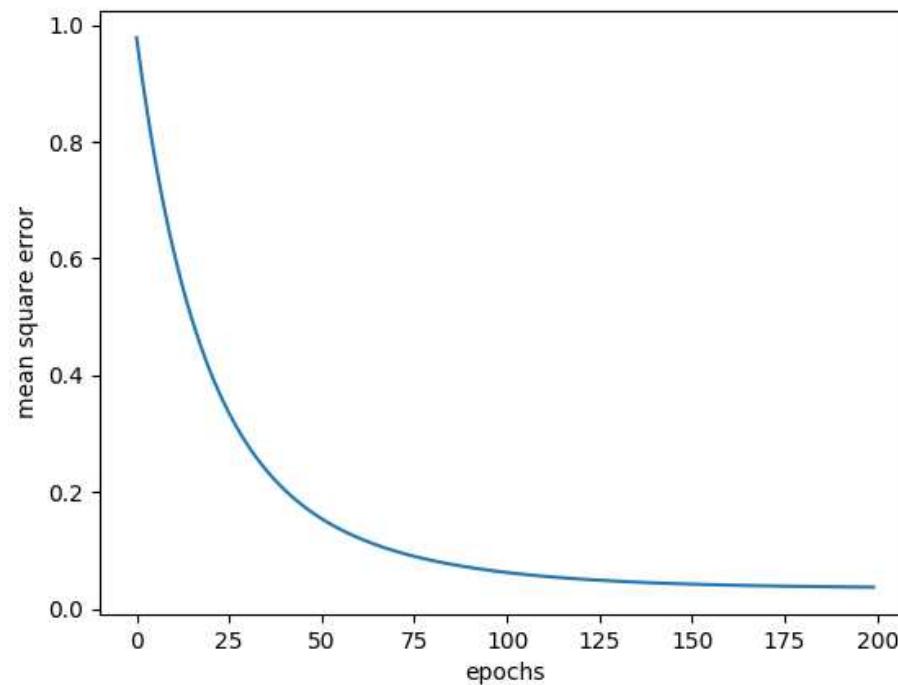
Example 1: epoch 1

x_p	y_p	s.e.	w	b
$x_1 = \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix}$	-0.19	2.29	$\begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$	0.02
$x_2 = \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix}$	-1.17	0.01	$\begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$	0.03
$x_3 = \begin{pmatrix} 0.00 \\ -0.55 \end{pmatrix}$	-0.37	0.87	$\begin{pmatrix} 0.93 \\ 0.69 \end{pmatrix}$	0.03
$x_4 = \begin{pmatrix} 0.27 \\ 0.50 \end{pmatrix}$	0.62	0.03	$\begin{pmatrix} 0.92 \\ 0.69 \end{pmatrix}$	0.02
$x_5 = \begin{pmatrix} -0.60 \\ 0.52 \end{pmatrix}$	-0.17	2.21	$\begin{pmatrix} 0.93 \\ 0.69 \end{pmatrix}$	0.01
$x_6 = \begin{pmatrix} 0.37 \\ 0.91 \end{pmatrix}$	0.98	0.45	$\begin{pmatrix} 0.93 \\ 0.68 \end{pmatrix}$	0.00

Example 1: epoch 200

x_p	y_p	s.e.	w	b
$x_1 = \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix}$	1.49	0.03	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	-0.01
$x_2 = \begin{pmatrix} 0.00 \\ -0.55 \end{pmatrix}$	0.22	0.12	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	-0.01
$x_3 = \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix}$	-0.98	0.01	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	-0.01
$x_4 = \begin{pmatrix} 0.37 \\ 0.91 \end{pmatrix}$	0.33	0.00	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	-0.01
$x_5 = \begin{pmatrix} 0.27 \\ 0.50 \end{pmatrix}$	0.30	0.02	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	-0.01
$x_6 = \begin{pmatrix} -0.60 \\ 0.52 \end{pmatrix}$	-1.45	0.04	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	-0.01

Example 1



$$\text{m.s.e.} = \frac{1}{6} \sum_{p=1}^6 (d_p - y_p)^2$$

Example 1

At convergence:

$$\begin{aligned}\mathbf{w} &= \begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix} \\ b &= -0.013\end{aligned}$$

Mean square error = 0.037

The regression equation:

$$y = \mathbf{x}^T \mathbf{w} + b = (x_1 \quad x_2) \begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix} - 0.013$$

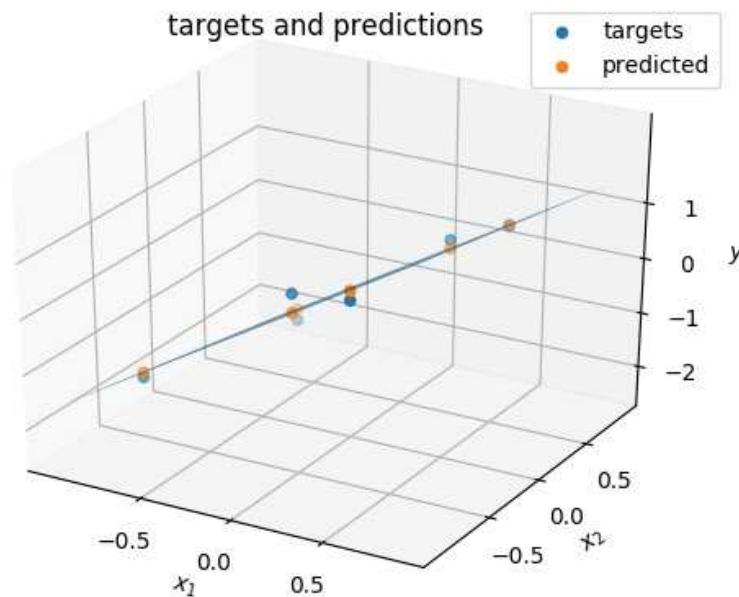
The mapping learnt by the linear neuron:

$$y = 2.00x_1 - 0.44x_2 - 0.013$$

Example 1

<i>inputs</i> $x = (x_1, x_2)$	<i>predictions</i> $y = 2.00x_1 - 0.44x_2 - 0.01$	<i>targets</i> d
(0.54, -0.96)	1.49	1.33
(0.27, 0.50)	0.30	0.45
(0.00, -0.55)	0.22	0.56
(-0.60, 0.52)	-1.45	-1.66
(-0.66, -0.82)	-0.98	-1.07
(0.37, 0.91)	0.33	0.30

Example 1



The mapping portrays a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.013$$

Gradient descent (GD) for linear neuron

Given a training dataset $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$, cost function J is given by the sum of square errors (s.s.e.):

$$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$$

where y_p is the neuron output for input pattern \mathbf{x}_p .

$$J = \sum_{p=1}^P J_p \tag{F}$$

where $J_p = \frac{1}{2} (d_p - y_p)^2$ is the square error for the p th pattern.

GD for linear neuron

From (F):

$$\begin{aligned}\nabla_{\mathbf{w}} J &= \sum_{p=1}^P \nabla_{\mathbf{w}} J_p \\ &= - \sum_{p=1}^P (d_p - y_p) \mathbf{x}_p && \text{from (D)} \\ &= -((d_1 - y_1) \mathbf{x}_1 + (d_2 - y_2) \mathbf{x}_2 + \dots + (d_P - y_P) \mathbf{x}_P) \\ &= -(\mathbf{x}_1 \quad \mathbf{x}_2 \quad \dots \quad \mathbf{x}_P) \begin{pmatrix} (d_1 - y_1) \\ (d_2 - y_2) \\ \vdots \\ (d_P - y_P) \end{pmatrix} \\ &= -\mathbf{X}^T (\mathbf{d} - \mathbf{y}) && (G)\end{aligned}$$

where $\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix}$ is the data matrix, $\mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$ is the target vector, and $\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix}$ is the output vector.

GD for linear neuron

Similarly, $\nabla_b J$ can be obtained by considering inputs of +1 and substituting a vector of +1 in (G):

$$\nabla_b J = -\mathbf{1}_P^T (\mathbf{d} - \mathbf{y}) \quad (\text{H})$$

where $\mathbf{1}_P = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$ has P elements of 1.

The output vector \mathbf{y} for the batch of P patterns is given by

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_P \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \mathbf{w} + b \\ \mathbf{x}_2^T \mathbf{w} + b \\ \vdots \\ \mathbf{x}_P^T \mathbf{w} + b \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix} \mathbf{w} + b \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \mathbf{X}\mathbf{w} + b \mathbf{1}_P$$

GD for linear neuron

Substituting (G) and (H) in gradient descent equations:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J$$

$$b \leftarrow b - \alpha \nabla_b J$$

We get GD learning equations for the linear neuron as

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - \mathbf{y})$$

$$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - \mathbf{y})$$

And α is the learning factor.

Where:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b \mathbf{1}_P$$

GD for linear neuron

Given a training dataset (X, d)

Set learning parameter α

Initialize w and b

Repeat until convergence:

$$y = Xw + b\mathbf{1}_P$$

$$w \leftarrow w + \alpha X^T(d - y)$$

$$b \leftarrow b + \alpha \mathbf{1}_P^T(d - y)$$

GD and SGD for a linear neuron

GD	SGD
(X, d)	(x_p, d_p)
$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$	$J = \frac{1}{2} (d_p - y_p)^2$
$\mathbf{y} = \mathbf{u} = X\mathbf{w} + b\mathbf{1}_P$	$y_p = u_p = \mathbf{x}_p^T \mathbf{w} + b$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - \mathbf{y})$	$\mathbf{w} \leftarrow \mathbf{w} + \alpha (d_p - y_p) \mathbf{x}_p$
$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - \mathbf{y})$	$b \leftarrow b + \alpha (d_p - y_p)$

Perceptron

Perceptron is a neuron having a **sigmoid** activation function and has an output

.

$$y = f(u)$$

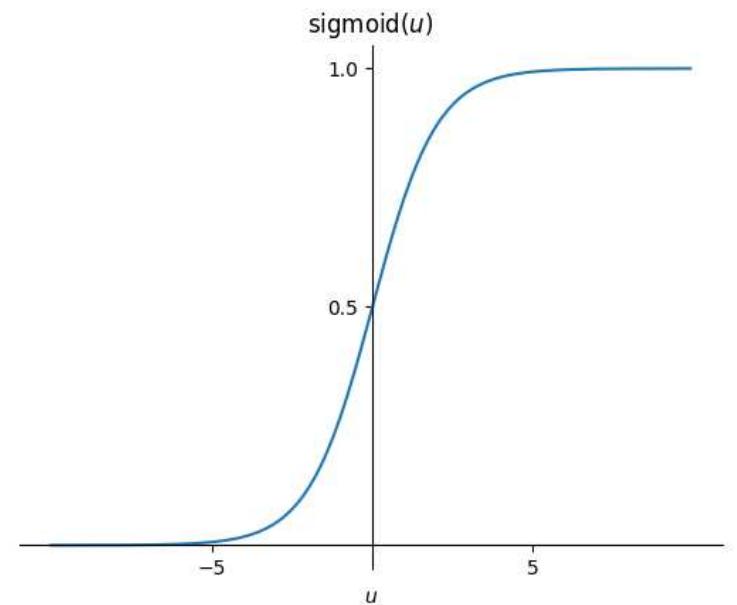
Where:

$$f(u) = \frac{1}{1 + e^{-u}} = \text{sigmoid}(u)$$

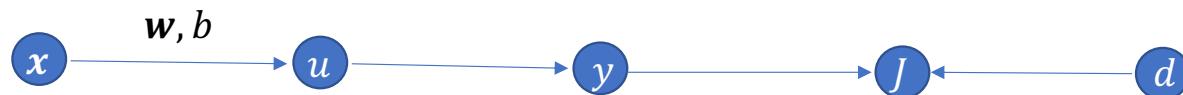
And $u = \mathbf{w}^T \mathbf{x} + b$

The square error is used as cost function for learning.

Perceptron performs a *non-linear regression* of inputs.



SGD for perceptron



Cost function J is given by

$$J = \frac{1}{2} (d - y)^2$$

where $y = f(u)$ and $u = \mathbf{w}^T \mathbf{x} + b$

The gradient with respect to the synaptic input:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial u} = -(d - y)f'(u)$$

From (C), $\frac{\partial u}{\partial w} = \mathbf{x}$ and $\frac{\partial u}{\partial b} = 1$.

$$\nabla_w J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial w} = -(d - y)f'(u)\mathbf{x} \quad (\text{I})$$

$$\nabla_b J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial b} = -(d - y)f'(u) \quad (\text{J})$$

SGD for perceptron

Gradient learning equations:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J \\ b &\leftarrow b - \alpha \nabla_b J\end{aligned}$$

Substituting gradients from (I) and (J), SGD equations for a perceptron are given by

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} + \alpha(d - y)f'(u)\mathbf{x} \\ b &\leftarrow b + \alpha(d - y)f'(u)\end{aligned}$$

SGD algorithm for perceptron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$

Set learning parameter α

Initialize w and b

Repeat until convergence:

For every training pattern (x_p, d_p) :

$$u_p = w^T x_p + b$$

$$y_p = f(u_p) = \frac{1}{1+e^{-u_p}}$$

$$w \leftarrow w + \alpha(d_p - y_p)f'(u_p)x_p$$

$$b \leftarrow b + \alpha(d_p - y_p)f'(u_p)$$

GD for perceptron

Given a training dataset $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$, cost function J is given by the sum of square errors (s.s.e) over all the patterns:

$$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2 = \sum_{p=1}^P J_p \quad (\text{F})$$

where $J_p = \frac{1}{2} (d_p - y_p)^2$ is the square error for the p th pattern.

GD for perceptron

From (F):

$$\begin{aligned}
 \nabla_{\mathbf{w}} J &= \sum_{p=1}^P \nabla_{\mathbf{w}} J_p \\
 &= - \sum_{p=1}^P (d_p - y_p) f'(u_p) \mathbf{x}_p && \text{From (J)} \\
 &= -((d_1 - y_1) f'(u_1) \mathbf{x}_1 + (d_2 - y_2) f'(u_2) \mathbf{x}_2 + \cdots + (d_P - y_P) f'(u_P) \mathbf{x}_P) \\
 &= -(\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_P) \begin{pmatrix} (d_1 - y_1) f'(u_1) \\ (d_2 - y_2) f'(u_2) \\ \vdots \\ (d_P - y_P) f'(u_P) \end{pmatrix} \\
 &= -\mathbf{X}^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u}) && \text{(K)}
 \end{aligned}$$

where $\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix}$, $\mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$, $\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix}$, and $f'(\mathbf{u}) = \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_P) \end{pmatrix}$

GD for perceptron

Substituting \mathbf{X}^T by $\mathbf{1}_P^T$ in (K), we get

$$\nabla_b J = -\mathbf{1}_P^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u}) \quad (\text{L})$$

where $\mathbf{1}_P = (1 \ 1 \ \dots \ 1)^T$.

The gradient descent learning is given by

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J \\ b &\leftarrow b - \alpha \nabla_b J\end{aligned}$$

Substituting (K) and (L), we get the learning equations:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u}) \\ b &\leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})\end{aligned}$$

Note that \cdot is the element-wise product.

GD for perceptron

Given a training dataset(\mathbf{X}, \mathbf{d})

Set learning parameter α

Initialize \mathbf{w} and b

Repeat until convergence:

$$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1}_P$$

$$\mathbf{y} = f(\mathbf{u}) = \frac{1}{1+e^{-\mathbf{u}}}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$$

$$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$$

Gradient descent for perceptron

GD	SGD
(X, d)	(x_p, d_p)
$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$	$J = \frac{1}{2} (d_p - y_p)^2$
$\mathbf{u} = X\mathbf{w} + b\mathbf{1}_P$	$u_p = \mathbf{x}_p^T \mathbf{w} + b$
$\mathbf{y} = f(\mathbf{u})$	$y_p = f(u_p)$
$\mathbf{w}' = \mathbf{w} + \alpha X^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$	$\mathbf{w}' = \mathbf{w} + \alpha (d_p - y_p) f'(u_p) \mathbf{x}_p$
$b' = b + \alpha \mathbf{1}_P^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$	$b' = b + \alpha (d_p - y_p) f'(u_p)$

Derivatives of Sigmoid

The activation function of the (continuous) perceptron is *sigmoid function* (i.e., unipolar sigmoidal function with $a = 1.0$ and $b = 1.0$) :

$$y = f(u) = \frac{1}{1 + e^{-u}}$$

The derivative is given by

$$f'(u) = \frac{-1}{(1+e^{-u})^2} \frac{\partial(e^{-u})}{\partial u} = \frac{e^{-u}}{(1+e^{-u})^2} = \frac{1}{1+e^{-u}} - \frac{1}{(1+e^{-u})^2} = y(1-y)$$

For *Tanh* function (bipolar sigmoid):

$$y = f(u) = \frac{e^{+u} - e^{-u}}{e^{+u} + e^{-u}}$$
$$f'(u) = \frac{(e^{+u}+e^{-u})(e^{+u}+e^{-u}) - (e^{+u}-e^{-u})(e^{+u}-e^{-u})}{(e^{+u}+e^{-u})^2} = \left(1 - \left(\frac{e^{+u}-e^{-u}}{e^{+u}+e^{-u}}\right)^2\right) = 1 - y^2$$

Example 2

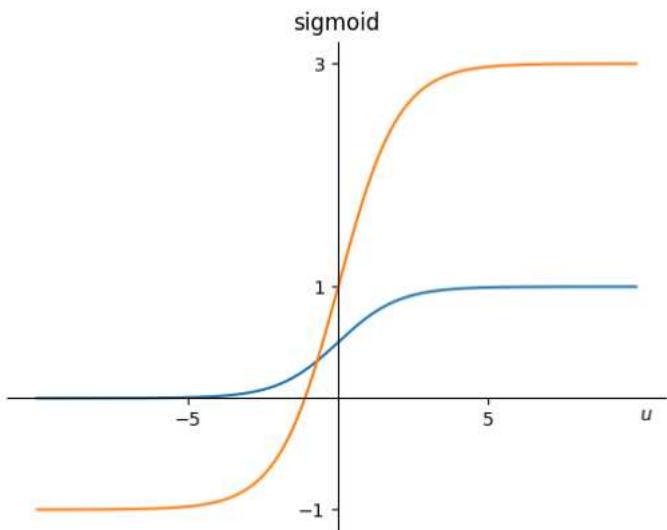
Design a perceptron to learn the following mapping by using gradient descent (GD):

$x = (x_1, x_2)$	d
(0.77, 0.02)	2.91
(0.63, 0.75)	0.55
(0.50, 0.22)	1.28
(0.20, 0.76)	-0.74
(0.17, 0.09)	0.88
(0.69, 0.95)	0.30
(0.00, 0.51)	-0.28

Use learning factor $\alpha = 0.01$.

Example 2

$$X = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} \text{ and } \mathbf{d} = \begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix}$$



Initially, $\mathbf{w} = \begin{pmatrix} 0.81 \\ 0.61 \end{pmatrix}$ and $b = 0.0$
 $\alpha = 0.01$

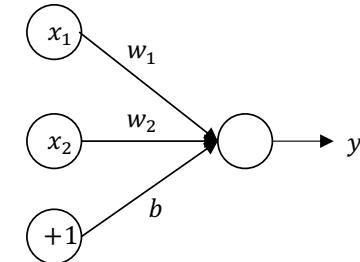
Output $y \in [-0.74, 2.91] \subset [-1.0, 3.0]$

Note that the sigmoidal should have an amplitude = 4 and shifted downwards by 1.0.

So, the activation function should be

$$y = f(u) = \frac{4.0}{1 + e^{-u}} - 1.0$$

$$f'(u) = \frac{4e^{-u}}{(1+e^{-u})^2} = (y+1) \frac{e^{-u}}{(1+e^{-u})} = (y+1) \left(1 - \frac{1}{1+e^{-u}}\right) = \frac{1}{4} (y+1)(3-y)$$



Example 2

Epoch 1 begins ...

$$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1}_P = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} \begin{pmatrix} 0.81 \\ 0.61 \end{pmatrix} + 0.0 \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} = \begin{pmatrix} 0.64 \\ 0.97 \\ 0.54 \\ 0.63 \\ 0.19 \\ 1.14 \\ 0.32 \end{pmatrix}$$

$$y = f(\mathbf{u}) = \frac{4.0}{1 + e^{-\mathbf{u}}} - 1.0 = \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix}$$

$$m.s.e. = \frac{1}{7} \sum_{p=1}^7 (d_p - y_p)^2 = \frac{1}{7} ((2.91 - 1.61)^2 + (0.55 - 1.90)^2 + \dots) = 2.11$$

Example 2

$$f'(\mathbf{u}) = \frac{1}{4}(\mathbf{y} + \mathbf{1}) \cdot (\mathbf{3} - \mathbf{y}) = \frac{1}{4} \left(\begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix} + \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \right) \cdot \left(\begin{pmatrix} 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \end{pmatrix} - \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix} \right) = \begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix}$$

Example 2

$$\mathbf{w} = \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$$

$$= \begin{pmatrix} 0.81 \\ 0.61 \end{pmatrix} + 0.01 \begin{pmatrix} 0.77 & 0.63 & 0.50 & 0.20 & 0.17 & 0.69 & 0.00 \\ 0.02 & 0.75 & 0.22 & 0.76 & 0.09 & 0.95 & 0.51 \end{pmatrix} \left(\begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix} - \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix} \right) \cdot \begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix} = \begin{pmatrix} 0.80 \\ 0.57 \end{pmatrix}$$

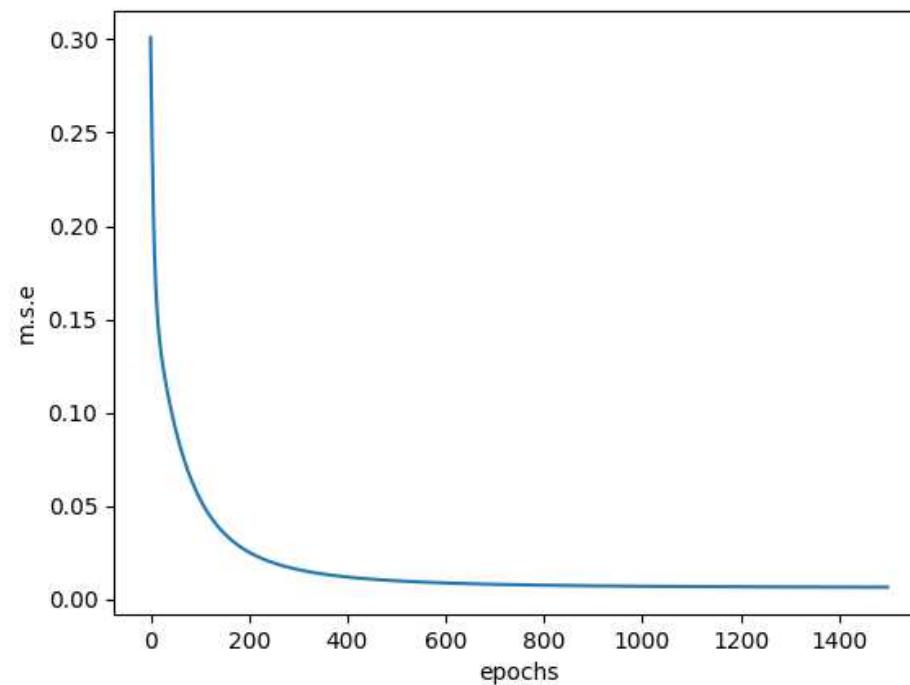
$$b = b + \alpha \mathbf{1}_P^T (\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$$

$$= 0.0 + 0.01 \times (1.0 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0) \left(\begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix} - \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix} \right) \cdot \begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix} = -0.05$$

Example 2

<i>iter</i>	<i>u</i>	<i>y</i>	<i>f'(u)</i>	<i>mse</i>	<i>w</i>	<i>b</i>
1	$\begin{pmatrix} 0.64 \\ 0.97 \\ 0.54 \\ 0.63 \\ 0.19 \\ 1.14 \\ 0.32 \end{pmatrix}$	$\begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix}$	$\begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix}$	2.11	$\begin{pmatrix} 0.80 \\ 0.57 \end{pmatrix}$	-0.05
2	$\begin{pmatrix} 0.57 \\ 0.88 \\ 0.47 \\ 0.54 \\ 1.04 \\ 1.95 \\ 0.24 \end{pmatrix}$	$\begin{pmatrix} 1.56 \\ 1.83 \\ 1.46 \\ 1.52 \\ 1.13 \\ 1.95 \\ 1.24 \end{pmatrix}$	$\begin{pmatrix} 0.92 \\ 0.83 \\ 0.95 \\ 0.93 \\ 1.00 \\ 0.77 \\ 0.99 \end{pmatrix}$	1.96	$\begin{pmatrix} 0.79 \\ 0.52 \end{pmatrix}$	-0.11
1500	$\begin{pmatrix} 2.05 \\ -0.45 \\ 0.56 \\ -1.94 \\ -0.16 \\ -0.85 \\ -1.89 \end{pmatrix}$	$\begin{pmatrix} 2.54 \\ 0.56 \\ 1.55 \\ -0.50 \\ 0.84 \\ 0.20 \\ -0.48 \end{pmatrix}$	$\begin{pmatrix} 0.40 \\ 0.95 \\ 0.92 \\ 0.44 \\ 0.99 \\ 0.84 \\ 0.45 \end{pmatrix}$	0.046	$\begin{pmatrix} 3.35 \\ -2.80 \end{pmatrix}$	-0.47

Example 2



Example 2

At convergence:

$$\begin{aligned}\mathbf{w} &= \begin{pmatrix} 3.35 \\ -2.80 \end{pmatrix} \\ b &= -0.47\end{aligned}$$

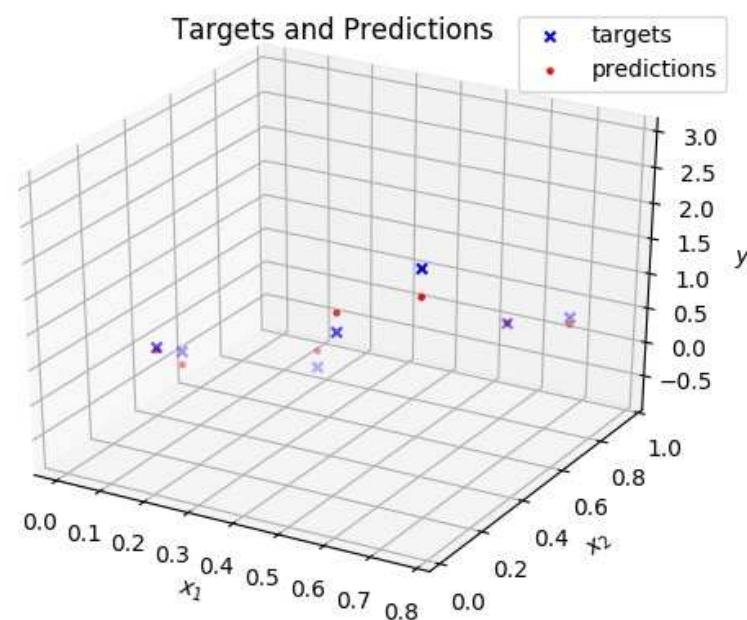
Mean square error = 0.01

$$u = \mathbf{x}^T \mathbf{w} + b = (x_1 \quad x_2) \begin{pmatrix} 3.35 \\ -2.80 \end{pmatrix} - 0.47 = 3.35x_1 - 2.8x_2 - 0.47$$

$$y = \frac{4.0}{1 + e^{-u}} - 1.0$$

$$y = \frac{4.0}{1 + e^{-3.35x_1 + 2.8x_2 + 0.47}} - 1.0$$

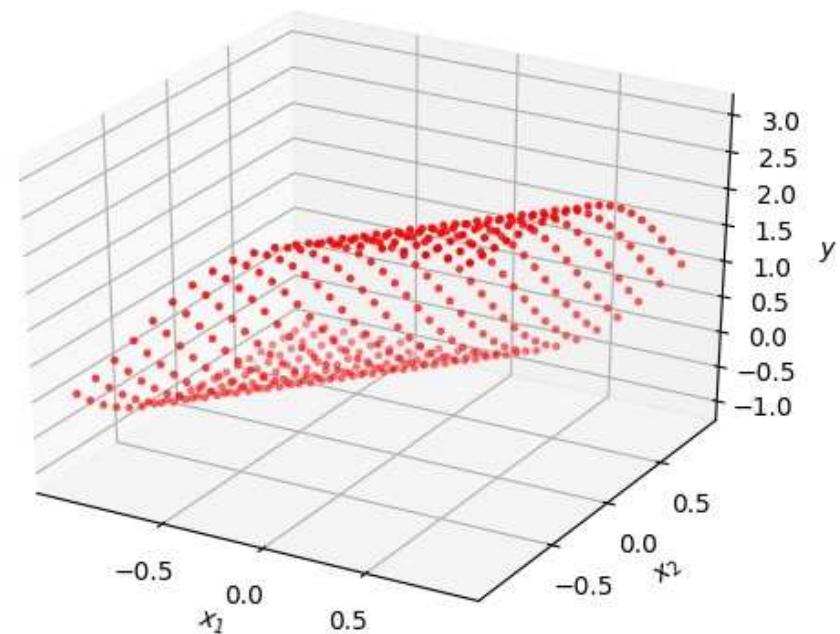
Example 2



Example 2

Non-linear function learnt by the perceptron

$$y = \frac{4.0}{1 + e^{-3.35x_1 + 2.8x_2 + 0.47}} - 1.0$$



Classification Example

Classification is to identify or distinguish classes or groups of objects.

Example: To identify *ballet dancers* from *rugby players*.

Two distinctive *features* that can aid in classification:

- ***weight***
- ***height***

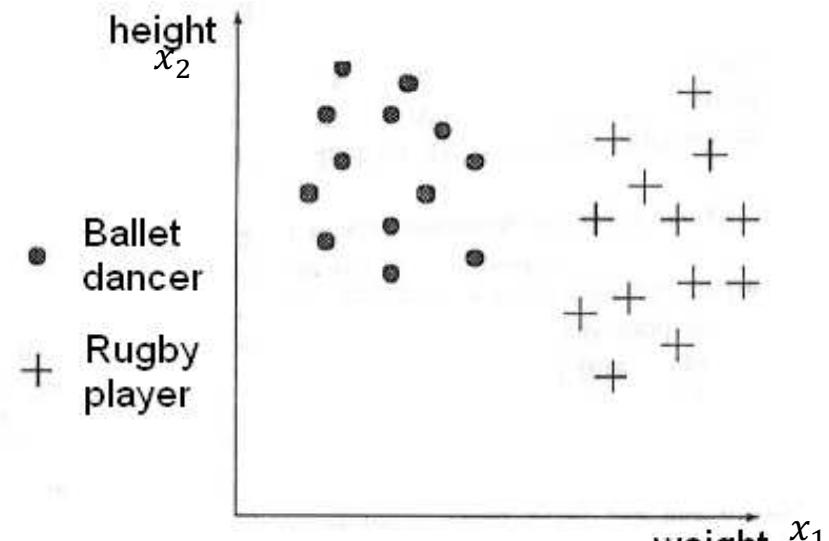


Figure: A 2-dimensional feature space

Let x_1 denote weight and x_2 denote height. Every individual is represented as a point $\mathbf{x} = (x_1, x_2)$ in the feature space.

Classification Example

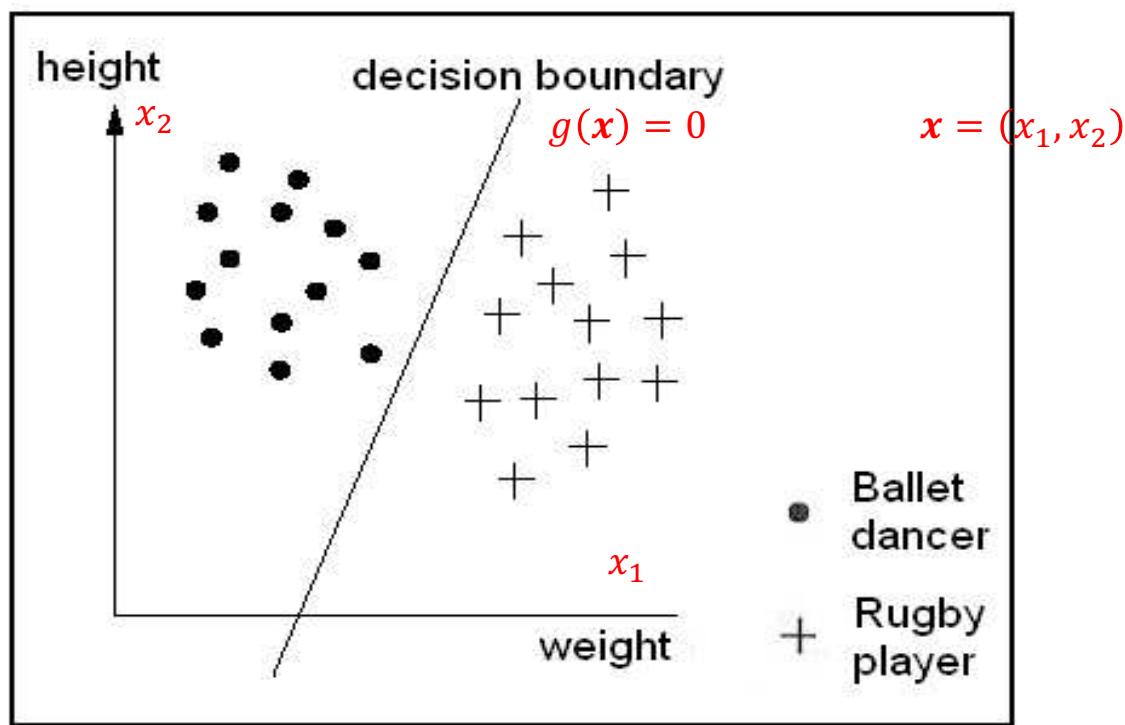


Figure: A linear classification decision boundary

Decision boundary

The **decision boundary** of the classifier, $g(\mathbf{x}) = 0$ where function $g(\mathbf{x})$ is referred to as the **discriminant function**.

A classifier finds a decision boundary separating the two classes in the feature space. On one side of the decision boundary, discriminant function is positive and on other side, discriminant function is negative.

Therefore, the following class definition may be employed:

If $g(\mathbf{x}) > 0 \Rightarrow$ Ballet dancer

If $g(\mathbf{x}) \leq 0 \Rightarrow$ Rugby player

Linear Classifier

If the two classes can be separated by a straight line, the classification is said to be **linearly separable**. For linear separable classes, one can design a **linear classifier**.

A linear classifier implements discriminant function or a decision boundary that is represented by a straight line (hyper plane) in the multidimensional **feature space**. Generally, the feature space is multidimensional. In the multidimensional space, a straight line or **hyperplane** is indicated by a linear sum of coordinates.

Given an input (features), $\mathbf{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$. A linear description function is given by

$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

where $\mathbf{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$ are the coefficient/weights and w_0 is the constant term.

Discrete perceptron as a linear two-class classifier

The linear discriminant function can be implemented by the synaptic input to a neuron

$$g(\mathbf{x}) = u = \mathbf{w}^T \mathbf{x} + b$$

And with a threshold activation function $1(u)$:

$$\begin{aligned} u = g(\mathbf{x}) > 0 &\rightarrow y = 1 \rightarrow \text{class1} \\ u = g(\mathbf{x}) \leq 0 &\rightarrow y = 0 \rightarrow \text{class2} \end{aligned}$$

That is, two-class linear classifier (or a dichotomizer) can be implemented with an artificial neuron with a threshold (unit step) activation function (**discrete perceptron**).

The output, 0 or 1, of the binary neuron represents the **label** of the class.

Classification error

In classification, the error is expressed as total mismatches between the target and output labels.

$$\text{Classification error} = \sum_{p=1}^P \mathbf{1}(d_p \neq y_p)$$

Logistic regression neuron

A **logistic regression neuron** performs a binary classification of inputs. That is, it classifies inputs into two classes with labels '0' and '1'.

The activation function of the logistic regression neuron is given by the sigmoid function:

$$f(u) = \frac{1}{1 + e^{-u}}$$

where $u = \mathbf{w}^T \mathbf{x} + b$ is the synaptic input to the neuron.

The activation of a logistic regression neuron gives the probability of the neuron output belonging to class '1'.

$$P(y = 1|\mathbf{x}) = f(u)$$

Then

$$P(y = 0|\mathbf{x}) = 1 - P(y = 1|\mathbf{x}) = 1 - f(u)$$

Logistic Regression Neuron

A logistic regression neuron receives an input $x \in \mathbf{R}^n$ and produces a class label $y \in \{0, 1\}$ as the output.

$$f(u) = \frac{1}{1 + e^{-u}}$$

When $u = 0$, $f(u) = P(y = 1|x) = P(y = 0|x) = 0.5$.
That is, $y = 1$ if $f(u) > 0.5$, else $y = 0$.

The output y of the neuron is given by:

$$y = 1(f(u) > 0.5) = 1(u > 0)$$

Note that for logistic neuron, the output and activation are different. It finds a linear boundary $u = 0$ separating the two classes.

SGD for logistic regression neuron

Given a training pattern (x, d) where $x \in R^n$ and $d \in \{0,1\}$.

The cost function for classification is given by the **cross-entropy**:

$$J = -d\log(f(u)) - (1 - d)\log(1 - f(u))$$

where $u = w^T x + b$ and $f(u) = \frac{1}{1+e^{-u}}$.

The cost function J is minimized using the gradient descent procedure.

$$J = \begin{cases} -\log(f(u)) & \text{if } d = 1 \\ -\log(1 - f(u)) & \text{if } d = 0 \end{cases}$$

SGD for logistic regression neuron

$$J = -d \log(f(u)) - (1-d) \log(1-f(u))$$

where $u = \mathbf{w}^T \mathbf{x} + b$ and $f(u) = \frac{1}{1+e^{(-u)}}$.

Gradient with respect to u :

$$\begin{aligned}\frac{\partial J}{\partial u} &= -\frac{\partial}{\partial f(u)} \left(d \log(f(u)) + (1-d) \log(1-f(u)) \right) \frac{\partial f(u)}{\partial u} \\ &= -\left(\frac{d}{f(u)} - \frac{(1-d)}{1-f(u)} \right) f'(u)\end{aligned}$$

Substituting $f'(u) = f(u)(1-f(u))$ for sigmoid activation function,

$$\frac{\partial J}{\partial u} = -\frac{d - f(u)}{f(u)(1-f(u))} f(u)(1-f(u)) = -(d - f(u))$$

SGD for logistic regression neuron

Substituting $\frac{\partial J}{\partial u}$, $\frac{\partial u}{\partial w} = \mathbf{x}$, and $\frac{\partial u}{\partial b} = 1$.

$$\nabla_w J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial w} = -(d - f(u))\mathbf{x} \quad (\text{A})$$

$$\nabla_b J = \frac{\partial J}{\partial u} \frac{\partial u}{\partial b} = -(d - f(u)) \quad (\text{B})$$

Gradient learning equations:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \nabla_w J \\ b &\leftarrow b - \alpha \nabla_b J\end{aligned}$$

Substituting $\nabla_w J$ and $\nabla_b J$ for logistic regression neuron

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} + \alpha(d - f(u))\mathbf{x} \\ b &\leftarrow b + \alpha(d - f(u))\end{aligned}$$

SGD for logistic regression neuron

Given a training dataset $\{(x_p, d_p)\}_{p=1}^P$

Set learning rate α

Initialize w and b

Iterate until convergence:

For every pattern (x_p, d_p) :

$$u_p = w^T x_p + b$$

$$f(u_p) = \frac{1}{1+e^{-u_p}}$$

$$w \leftarrow w + \alpha (d_p - f(u_p)) x_p$$

$$b \leftarrow b + \alpha (d_p - f(u_p))$$

GD for logistic regression neuron

Given a training dataset $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$ where $\mathbf{x}_p \in \mathbb{R}^n$ and $d_p \in \{0,1\}$.

The cost function for logistic regression is given by the *cross-entropy* (or *negative log-likelihood*) over all the training patterns:

$$J = - \sum_{p=1}^P d_p \log(f(u_p)) + (1 - d_p) \log(1 - f(u_p))$$

where $u_p = \mathbf{w}^T \mathbf{x}_p + b$ and $f(u_p) = \frac{1}{1+e^{-u_p}}$.

The cost function J can be written as

$$J = \sum_{p=1}^P J_p$$

where $J_p = -d_p \log(f(u_p)) - (1 - d_p) \log(1 - f(u_p))$ is cross-entropy due to p th pattern.

GD for logistic regression neuron

$$\begin{aligned}
 \nabla_{\mathbf{w}} J &= \sum_{p=1}^P \nabla_{\mathbf{w}} J_p \\
 &= - \sum_{p=1}^P (d_p - f(u_p)) \mathbf{x}_p && \text{From (A)} \\
 &= - \left((d_1 - f(u_1)) \mathbf{x}_1 + (d_2 - f(u_2)) \mathbf{x}_2 + \cdots + (d_P - f(u_P)) \mathbf{x}_P \right) \\
 &= -(\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_P) \begin{pmatrix} (d_1 - f(u_1)) \\ (d_2 - f(u_2)) \\ \vdots \\ (d_P - f(u_P)) \end{pmatrix} \\
 &= -\mathbf{X}^T (\mathbf{d} - f(\mathbf{u}))
 \end{aligned}$$

where $\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix}$, $\mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$, and $f(\mathbf{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_P) \end{pmatrix}$

GD for logistic regression neuron

By substituting $\mathbf{1}_P$ for \mathbf{X} in above equation:

$$\nabla_b J = -\mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

Substituting the gradients in

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} J$$

$$b \leftarrow b - \alpha \nabla_b J$$

the gradient descent learning for logistic regression neuron is given by

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u}))$$

$$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

Note that \mathbf{y} in the discrete perceptron is now replaced with $f(\mathbf{u})$ in logistic regression learning equations.

GD for logistic regression neuron

Given training data (\mathbf{X}, \mathbf{d})

Set learning rate α

Initialize \mathbf{w} and b

Iterate until convergence:

$$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1}_P$$

$$f(\mathbf{u}) = \frac{1}{1+e^{-\mathbf{u}}}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u}))$$

$$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

Learning for logistic regression neuron

GD	SGD
(\mathbf{X}, \mathbf{d})	(\mathbf{x}_p, d_p)
$J = - \sum_{p=1}^P d_p \log(f(u_p)) + (1 - d_p) \log(1 - f(u_p))$	$J_p = -d_p \log(f(u_p)) - (1 - d_p) \log(1 - f(u_p))$
$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1}_P$	$u_p = \mathbf{w}^T \mathbf{x}_p + b$
$f(\mathbf{u}) = \frac{1}{1 + e^{-\mathbf{u}}}$	$f(u_p) = \frac{1}{1 + e^{-u_p}}$
$\mathbf{y} = 1(f(\mathbf{u}) > 0.5)$	$y_p = 1(f(u_p) > 0.5)$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u}))$	$\mathbf{w} \leftarrow \mathbf{w} + \alpha (d_p - f(u_p)) \mathbf{x}_p$
$b \leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$	$b \leftarrow b + \alpha (d_p - f(u_p))$

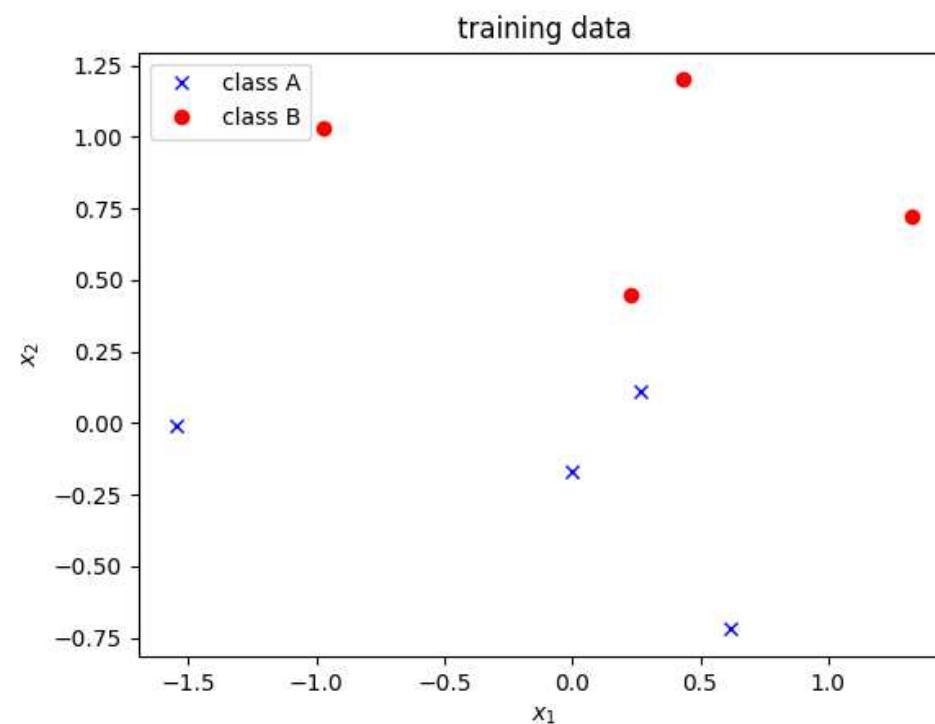
Example 3: GD for logistic regression neuron

Train a logistic regression neuron to perform the following classification, using GD:

- (1.33 0.72) \rightarrow class B
- (-1.55 -0.01) \rightarrow class A
- (0.62 -0.72) \rightarrow class A
- (0.27 0.11) \rightarrow class A
- (0.0 -0.17) \rightarrow class A
- (0.43 1.2) \rightarrow class B
- (-0.97 1.03) \rightarrow class B
- (0.23 0.45) \rightarrow class B

User a learning factor $\alpha = 0.04$.

Example 3



Example 3

Let $y = 1$ for class A and $y = 0$ for class B.

$$X = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & -0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \text{ and } d = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Initially, $w = \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix}$, $b = 0.0$ and $\alpha = 0.4$

Example 3

Epoch 1:

$$\mathbf{u} = \mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & 0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.0 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.04 \\ -1.2 \\ 0.46 \\ 0.21 \\ 0.00 \\ 0.36 \\ -0.73 \\ 0.19 \end{pmatrix}$$

$$f(\mathbf{u}) = \frac{1}{1 + e^{(-\mathbf{u})}} = \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix}$$

Example 3

$$\mathbf{y} = \mathbf{1}(f(\mathbf{u}) > 0.5) = \mathbf{1} \left(\begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} > 0.5 \right) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

$$\mathbf{d} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\text{Classification error} = \sum_{p=1}^8 \mathbf{1}(d_p \neq y_p) = 5$$

$$\begin{aligned} \text{Cross-entropy} &= -\sum_{p=1}^P d_p \log(f(u_p)) + (1 - d_p) \log(1 - f(u_p)) \\ &= -\log(1 - f(u_1)) - \log f(u_2) - \log f(u_3) - \dots - \log(1 - f(u_8)) \\ &= -\log(1 - 0.74) - \log 0.23 - \log 0.61 - \dots - \log(1 - 0.55) \\ &= 6.653 \end{aligned}$$

Example 3

$$\mathbf{w} = \mathbf{w} + \alpha X^T (\mathbf{d} - f(\mathbf{u}))$$

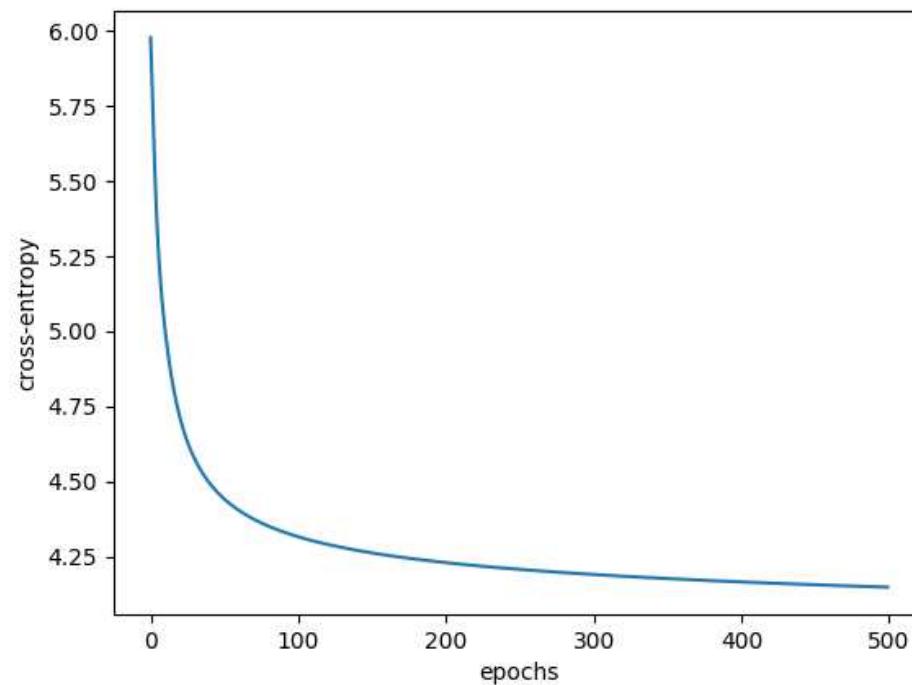
$$= \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.04 \begin{pmatrix} 1.33 & -1.55 & 0.62 & 0.27 & 0 & 0.43 & -0.97 & 0.23 \\ 0.72 & -0.01 & -0.72 & 0.11 & -0.17 & 1.2 & 1.03 & 0.45 \end{pmatrix} \left(\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} \right)$$

$$= \begin{pmatrix} 0.69 \\ -0.2 \end{pmatrix}$$

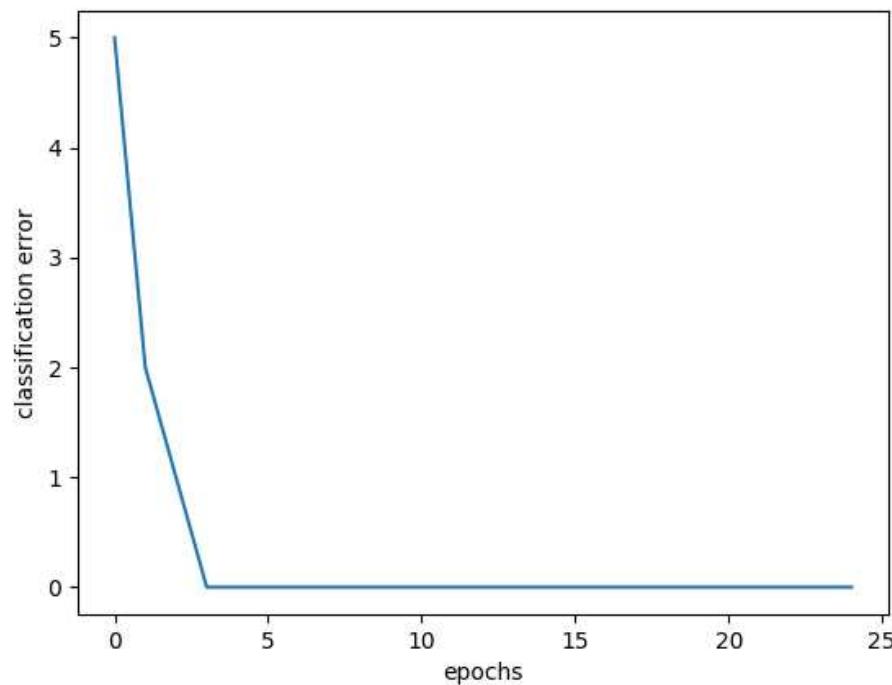
$$b = b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

$$= 0.0 + 0.04(1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1) \left(\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} \right) = -0.09$$

Example 3



Example 3



Example 3

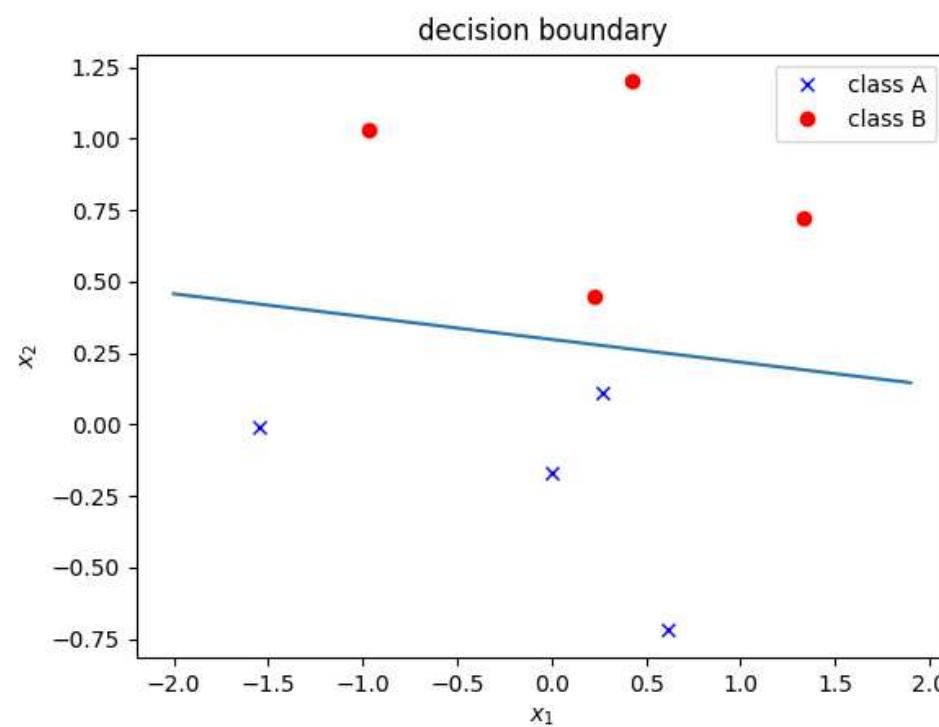
At convergence, $\mathbf{w} = \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix}, b = 4.47$

The decision boundary is given by: $u = \mathbf{x}^T \mathbf{w} + b = 0$

$$\begin{aligned}(x_1 & \quad x_2)^T \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix} + 4.47 = 0 \\ -1.20x_1 - 15.02x_2 + 4.47 & = 0\end{aligned}$$

Decision boundary:

$$-1.20x_1 - 15.02x_2 + 4.47 = 0$$

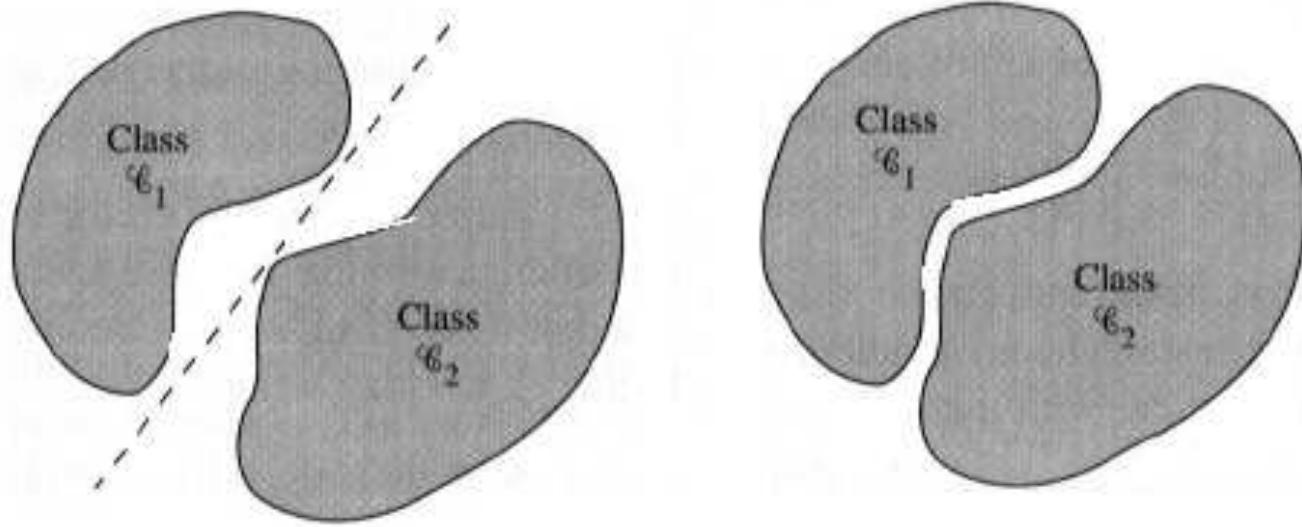


Limitations of logistic regression neuron

As long as the neuron is a *linear combiner* followed by a *non-linear activation function*, then regardless of the form of non-linearity used, the neuron can perform pattern classification *only on linearly separable* patterns.

Linear separability requires that the patterns to be classified must be sufficiently separated from each other to ensure that the decision boundaries are hyperplanes.

Limitations of logistic regression neuron



(a) Linearly
Separable Pattern

(b) Non-Linearly
Separable Pattern

Discrete perceptron and logistic regression neuron can create only linear decision boundaries.

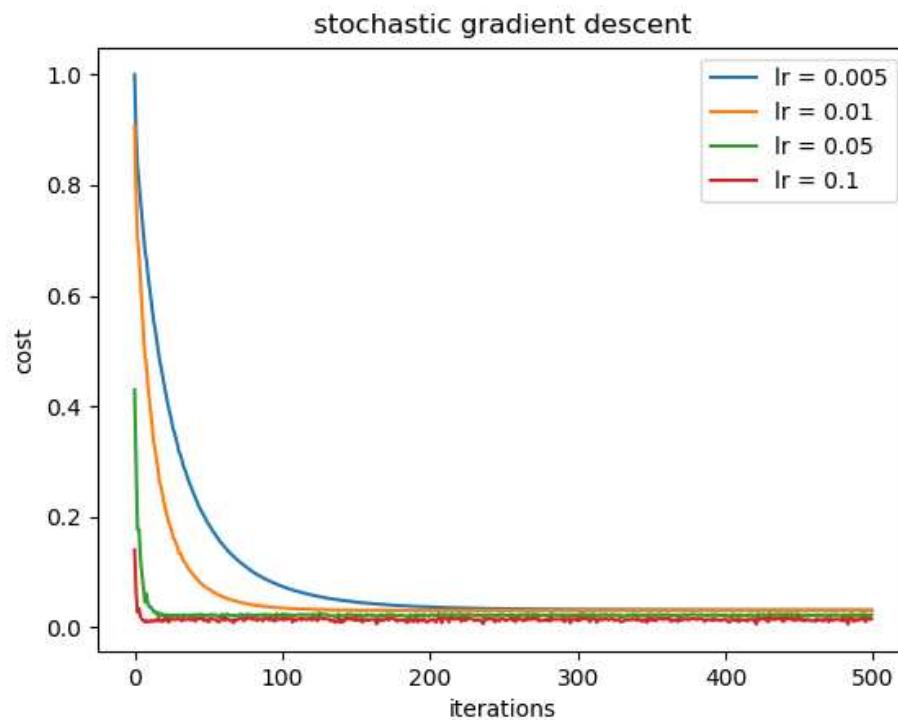
Example 4: effects of leaning rate

Design a perceptron to learn the following mapping by using gradient descent (GD):

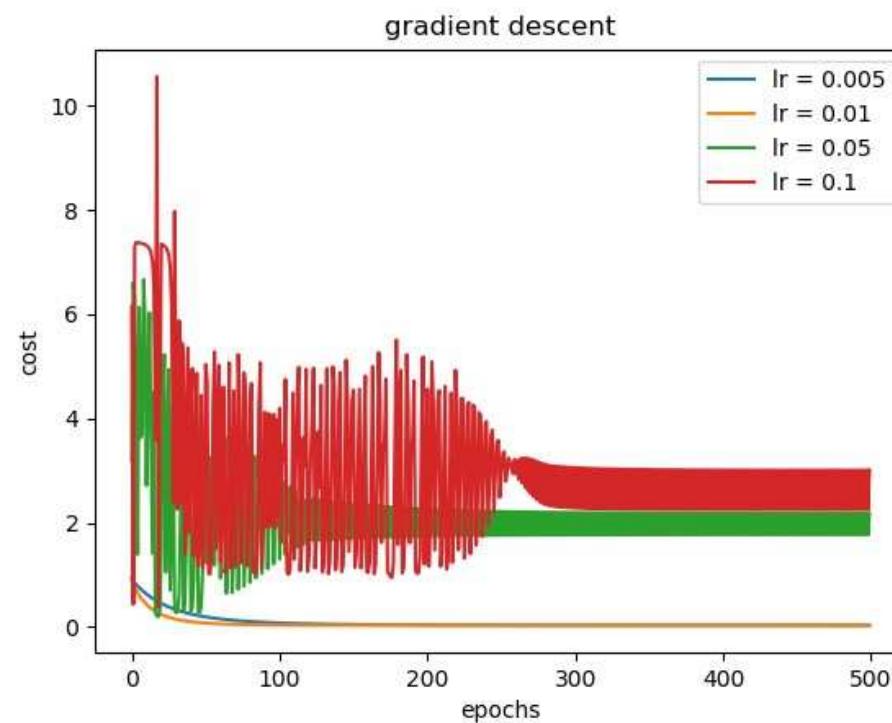
$x = (x_1, x_2)$	d
(0.77, 0.02)	2.91
(0.63, 0.75)	0.55
(0.50, 0.22)	1.28
(0.20, 0.76)	-0.74
(0.17, 0.09)	0.88
(0.69, 0.95)	0.30
(0.00, 0.51)	-0.28

Use learning factor $\alpha = 0.01$.

Example 4: Learning rates with SGD



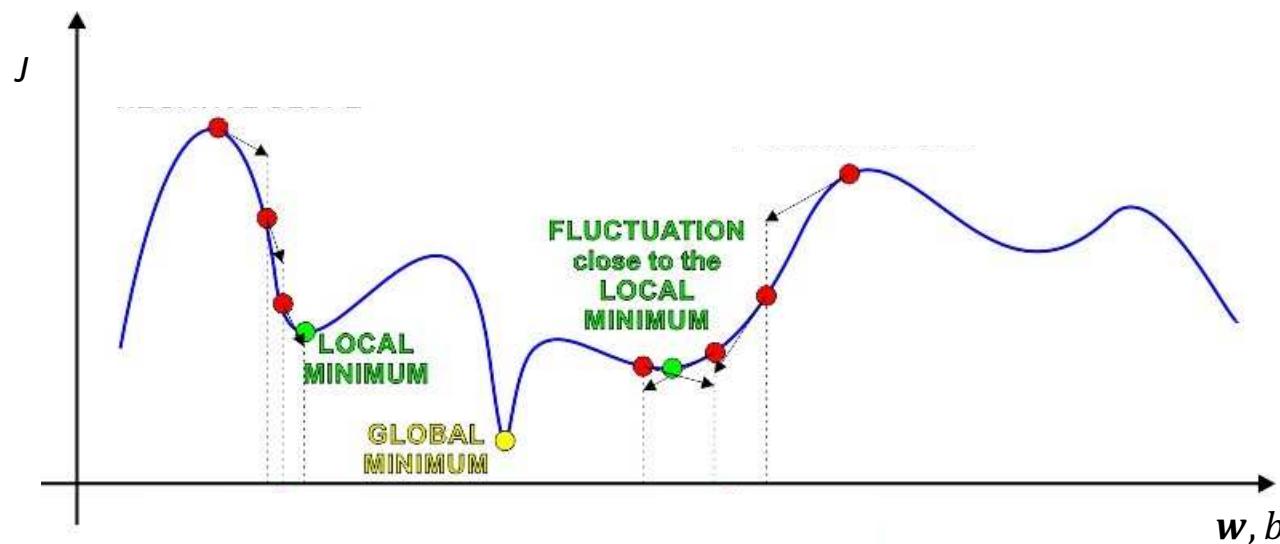
Example 4: Learning rates with GD



Learning rates

- At higher learning rates, convergence is faster but may not be stable.
- The *optimal learning rate* is the largest rate at which learning does not diverge.
- Generally, SGD converges to a better solution (lower error) as it capitalizes on randomness of data. SGD takes a longer time to converge
- Usually, GD can use a higher learning rate compared to SGD; The time for one add/multiply computation per a weight update is less when patterns are trained in a (small) batch.
- In practice, *mini-batch SGD* is used. Then, the time to train a network is dependent upon
 - the learning rate
 - the batch size

Local minima problem in gradient descent learning



Algorithm may get stuck in a local minimum of error function depending on the initial weights. Gradient descent gives a suboptimal solution and does not guarantee the optimal solution.

Summary: types of neurons

Role	Neuron
Regression (one dimensional)	Linear neuron
	Perceptron
Classification (two classes)	Logistic regression neuron

Summary: GD for neurons

$$\begin{aligned}
 & (X, d) \\
 & u = Xw + b\mathbf{1}_P \\
 & w = w - \alpha X^T \nabla_u J \\
 & b = b - \alpha \mathbf{1}_P^T \nabla_u J
 \end{aligned}$$

neuron	$f(u), y$	$\nabla_u J$
Logistic regression neuron	$f(u) = \frac{1}{1 + e^{-u}}$ $y = 1(f(u) > 0.5)$	$-(d - f(u))$
Linear neuron	$y = u$	$-(d - y)$
Perceptron	$y = f(u) = \frac{1}{1 + e^{-u}}$	$-(d - y) \cdot f'(u)$

Summary: SGD for neurons

$$\begin{aligned}
 & (\mathbf{x}_p, d_p) \\
 & u_p = \mathbf{w}^T \mathbf{x}_p + b \\
 & \mathbf{w} = \mathbf{w} - \alpha \nabla_{u_p} J \mathbf{x}_p \\
 & b = b - \alpha \nabla_{u_p} J
 \end{aligned}$$

neuron	$f(u_p), y_p$	$\nabla_{u_p} J$
Logistic regression neuron	$f(u_p) = \frac{1}{1 + e^{-u_p}}$ $y_p = 1(f(u_p) > 0.5)$	$-(d_p - f(u_p))$
Linear neuron	$y_p = u_p$	$-(d_p - y_p)$
Perceptron	$y_p = f(u_p) = \frac{1}{1 + e^{-u_p}}$	$-(d_p - y_p) \cdot f'(u_p)$

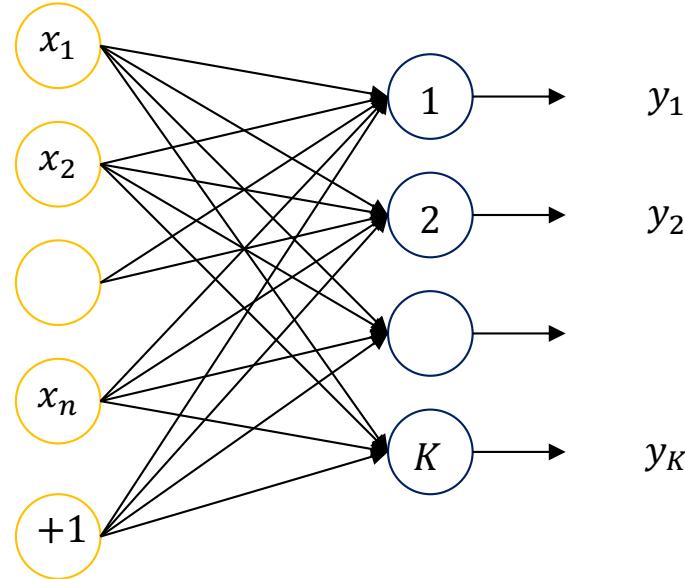
Chapter 3

Neuron Layers

Neural networks and deep learning

Weight matrix of a layer

Consider a layer of K neurons.



Let \mathbf{w}_k and b_k denote the weight vector and bias of k th neuron. Weights connected to a neuron layer is given by a weight matrix:

$$\mathbf{W} = (\mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_K)$$

where columns are given by weight vectors of individual neurons.

And a bias vector \mathbf{b} where each element corresponds to a bias of a neuron:

$$\mathbf{b} = (b_1, b_2, \dots, b_K)^T$$

Synaptic input at a layer for single input

Given an input pattern $\mathbf{x} \in \mathbf{R}^n$ to a layer of K neurons.

Synaptic input u_k to k th neuron:

$$u_k = \mathbf{w}_k^T \mathbf{x} + b_k$$

\mathbf{w}_k and b_k denote the weight vector and bias of k th neuron.

Synaptic input vector \mathbf{u} to the layer :

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{pmatrix} = \begin{pmatrix} \mathbf{w}_1^T \mathbf{x} + b_1 \\ \mathbf{w}_2^T \mathbf{x} + b_2 \\ \vdots \\ \mathbf{w}_K^T \mathbf{x} + b_k \end{pmatrix} = \begin{pmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_K^T \end{pmatrix} \mathbf{x} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$$

$$\mathbf{u} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$$

where \mathbf{W} is the weight matrix and \mathbf{b} is the bias vector of the layer.

Synaptic input to a layer for batch input

Given a set $\{\mathbf{x}_p\}_{p=1}^P$ input patterns to a layer of K neurons where $\mathbf{x}_p \in \mathbb{R}^n$.

Synaptic input \mathbf{u}_p to the layer for an input pattern \mathbf{x}_p :

$$\mathbf{u}_p = \mathbf{W}^T \mathbf{x}_p + \mathbf{b}$$

The synaptic input matrix \mathbf{U} to the layer for P patterns:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

$$\mathbf{U} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \mathbf{W} + \mathbf{b}^T \\ \mathbf{x}_2^T \mathbf{W} + \mathbf{b}^T \\ \vdots \\ \mathbf{x}_P^T \mathbf{W} + \mathbf{b}^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix} \mathbf{W} + \begin{pmatrix} \mathbf{b}^T \\ \mathbf{b}^T \\ \vdots \\ \mathbf{b}^T \end{pmatrix} = \mathbf{XW} + \mathbf{B}$$

where rows of \mathbf{U} are synaptic inputs corresponding to individual input patterns.

The matrix $\mathbf{B} = \begin{pmatrix} \mathbf{b}^T \\ \mathbf{b}^T \\ \vdots \\ \mathbf{b}^T \end{pmatrix}$ has bias vector propagated as rows.

Activation at a layer for batch input

The synaptic input to the layer due to a batch of patterns:

$$\mathbf{U} = \mathbf{X} \mathbf{W} + \mathbf{B}$$

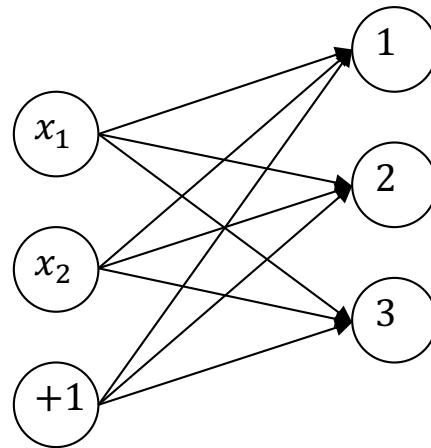
where rows of \mathbf{U} corresponds to synaptic inputs of the layer, corresponding to individual input patterns:

Activation of the layer:

$$f(\mathbf{U}) = \begin{pmatrix} f(\mathbf{u}_1^T) \\ f(\mathbf{u}_2^T) \\ \vdots \\ f(\mathbf{u}_P^T) \end{pmatrix} = \begin{pmatrix} f(\mathbf{u}_1)^T \\ f(\mathbf{u}_2)^T \\ \vdots \\ f(\mathbf{u}_P)^T \end{pmatrix}$$

where activations due to individual patterns are written as rows.

Example 1: activations and outputs of a perceptron layer



A perceptron layer of 3 neurons shown in the figure receives 2-dimensional inputs $(x_1, x_2)^T$, and has a weight matrix \mathbf{W} and a bias vector \mathbf{b} given by

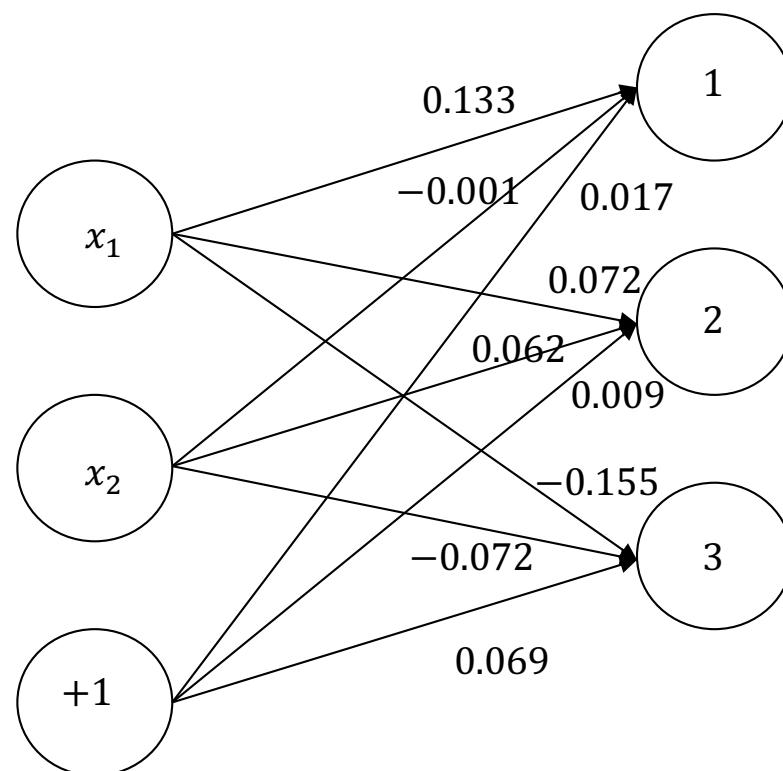
$$\mathbf{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 0.017 \\ 0.009 \\ 0.069 \end{pmatrix}$$

Using batch processing, find the output for input patterns:

$$\begin{pmatrix} 0.5 \\ -1.0 \\ -1.66 \end{pmatrix}, \begin{pmatrix} 0.78 \\ -0.51 \\ -0.65 \end{pmatrix}, \text{ and } \begin{pmatrix} 0.04 \\ -0.2 \end{pmatrix}.$$

Example 1

$$\mathbf{w} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 0.017 \\ 0.009 \\ 0.069 \end{pmatrix}.$$



Example 1

$$\mathbf{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \end{pmatrix}.$$

Input as a batch of four patterns:

$$\mathbf{X} = \begin{pmatrix} 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}$$

The synaptic input to the layer:

$$\begin{aligned} \mathbf{U} &= \mathbf{X}\mathbf{W} + \mathbf{B} \\ &= \begin{pmatrix} 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix} \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} + \begin{pmatrix} 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \end{pmatrix} \\ &= \begin{pmatrix} 0.085 & -0.059 & 0.111 \\ -0.115 & 0.094 & 0.26 \\ 0.121 & 0.024 & -0.005 \\ 0.022 & -0.001 & 0.077 \end{pmatrix} \end{aligned}$$

Example 1

$$\mathbf{U} = \begin{pmatrix} 0.085 & -0.059 & 0.111 \\ -0.115 & 0.094 & 0.26 \\ 0.121 & 0.024 & -0.005 \\ 0.022 & -0.001 & 0.077 \end{pmatrix}$$

For a perceptron layer

$$\mathbf{Y} = f(\mathbf{U}) = \frac{1}{1 + e^{-\mathbf{U}}} = \begin{pmatrix} 0.521 & 0.485 & 0.527 \\ 0.471 & 0.476 & 0.565 \\ 0.530 & 0.506 & 0.499 \\ 0.506 & 0.500 & 0.519 \end{pmatrix}$$

For example, third row corresponding to 3rd input:

$$\mathbf{x} = \begin{pmatrix} 0.78 \\ -0.65 \end{pmatrix}$$

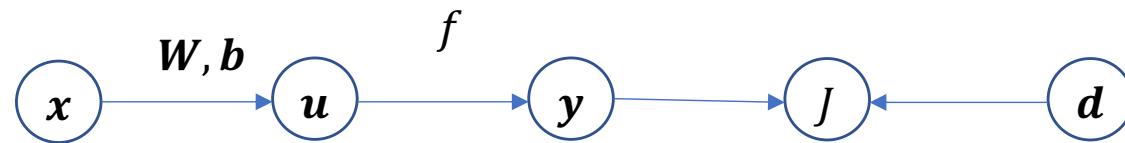
And the corresponding output

$$\mathbf{y} = \begin{pmatrix} 0.530 \\ 0.506 \\ 0.499 \end{pmatrix}$$

2nd neuron output
for 3rd input pattern

SGD for single layer

Computational graph processing input (\mathbf{x}, \mathbf{d}) :



J denotes the cost function.

Need to compute gradients $\nabla_{\mathbf{W}} J$ and $\nabla_{\mathbf{b}} J$ to learn weight matrix \mathbf{W} and bias vector \mathbf{b} .

SGD for single layer

Consider k th neuron at the layer:

$$u_k = \mathbf{x}^T \mathbf{w}_k + b_k$$

And

$$\frac{\partial u_k}{\partial \mathbf{w}_k} = \mathbf{x}$$

The gradient of the cost with respect to the weight connected to k th neuron:

$$\nabla_{\mathbf{w}_k} J = \frac{\partial J}{\partial u_k} \frac{\partial u_k}{\partial \mathbf{w}_k} = \mathbf{x} \frac{\partial J}{\partial u_k} \quad (\text{A})$$

$$\nabla_{b_k} J = \frac{\partial J}{\partial u_k} \frac{\partial u_k}{\partial b_k} = \frac{\partial J}{\partial u_k} \quad (\text{B})$$

SGD for single layer

Gradient of J with respect to $\mathbf{W} = (\mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_K)$:

$$\begin{aligned}\nabla_{\mathbf{W}} J &= (\nabla_{\mathbf{w}_1} J \quad \nabla_{\mathbf{w}_2} J \quad \cdots \quad \nabla_{\mathbf{w}_K} J) \\ &= \left(\mathbf{x} \frac{\partial J}{\partial u_1} \quad \mathbf{x} \frac{\partial J}{\partial u_2} \quad \cdots \quad \mathbf{x} \frac{\partial J}{\partial u_K} \right) \quad \text{From (A)} \\ &= \mathbf{x} \left(\frac{\partial J}{\partial u_1} \quad \frac{\partial J}{\partial u_2} \quad \cdots \quad \frac{\partial J}{\partial u_K} \right) \\ &= \mathbf{x}(\nabla_{\mathbf{u}} J)^T\end{aligned}$$

where

$$\nabla_{\mathbf{u}} J = \frac{\partial J}{\partial \mathbf{u}} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix}$$

That is, $\nabla_{\mathbf{W}} J = \mathbf{x}(\nabla_{\mathbf{u}} J)^T \quad (C)$

SGD for single layer

Similarly, by substituting $\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial u_k}$ from (B):

$$\nabla_{\mathbf{b}} J = \begin{pmatrix} \frac{\partial J}{\partial b_1} \\ \frac{\partial J}{\partial b_2} \\ \vdots \\ \frac{\partial J}{\partial b_K} \end{pmatrix} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix} = \nabla_{\mathbf{u}} J \quad (\text{D})$$

$$\nabla_{\mathbf{b}} J = \nabla_{\mathbf{u}} J$$

SGD for single layer

From (C) and (D),

$$\begin{aligned}\nabla_{\mathbf{w}} J &= \mathbf{x}(\nabla_{\mathbf{u}} J)^T \\ \nabla_{\mathbf{b}} J &= \nabla_{\mathbf{u}} J\end{aligned}$$

That is, by computing gradient $\nabla_{\mathbf{u}} J$ with respect to synaptic input \mathbf{u} , the gradient of cost J with respect to the weights and biases is obtained.

$$\begin{aligned}\mathbf{W} &= \mathbf{W} - \alpha \mathbf{x}(\nabla_{\mathbf{u}} J)^T \\ \mathbf{b} &= \mathbf{b} - \alpha \nabla_{\mathbf{u}} J\end{aligned}$$

GD for single layer

Given a set of patterns $\{(\mathbf{x}_p, \mathbf{d}_p)\}_{p=1}^P$ where $\mathbf{x}_p \in \mathbf{R}^n$ and $\mathbf{d}_p \in \mathbf{R}^K$ for regression and $d_p \in \{1, 2, \dots, K\}$ for classification.

The cost J is given by the sum of cost due to individual patterns:

$$J = \sum_{p=1}^P J_p$$

Where Then,

$$\nabla_{\mathbf{w}} J = \sum_{p=1}^P \nabla_{\mathbf{w}} J_p$$

GD for single layer

Substituting $\nabla_{\mathbf{W}} J_p = \mathbf{x}_p (\nabla_{\mathbf{u}_p} J_p)^T$ from (C) :

$$\begin{aligned}
 \nabla_{\mathbf{W}} J &= \sum_{p=1}^P \mathbf{x}_p (\nabla_{\mathbf{u}_p} J_p)^T \\
 &= \sum_{p=1}^P \mathbf{x}_p (\nabla_{\mathbf{u}_p} J)^T && \text{since } \nabla_{\mathbf{u}_p} J = \nabla_{\mathbf{u}_p} J_p. \\
 &= \mathbf{x}_1 (\nabla_{\mathbf{u}_1} J)^T + \mathbf{x}_2 (\nabla_{\mathbf{u}_2} J)^T + \dots + \mathbf{x}_P (\nabla_{\mathbf{u}_P} J)^T \\
 &= (\mathbf{x}_1 \quad \mathbf{x}_2 \quad \dots \quad \mathbf{x}_P) \begin{pmatrix} (\nabla_{\mathbf{u}_1} J)^T \\ (\nabla_{\mathbf{u}_2} J)^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J)^T \end{pmatrix} \\
 &= \mathbf{X}^T \nabla_{\mathbf{U}} J
 \end{aligned} \tag{E}$$

Note that $\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_P^T \end{pmatrix}$ and $\mathbf{U} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix}$

GD for single layer

$$J = \sum_{p=1}^P J_p$$

$$\begin{aligned}\nabla_{\mathbf{b}} J &= \sum_{p=1}^P \nabla_{\mathbf{b}} J_p \\ &= \sum_{p=1}^P \nabla_{\mathbf{u}_p} J_p && \text{Substituting from (D)} \\ &= \sum_{p=1}^P \nabla_{\mathbf{u}_p} J && \text{Since } \nabla_{\mathbf{u}_p} J = \nabla_{\mathbf{u}_p} J_p. \\ &= \nabla_{\mathbf{u}_1} J + \nabla_{\mathbf{u}_2} J + \cdots + \nabla_{\mathbf{u}_P} J \\ &= (\nabla_{\mathbf{u}_1} J \quad \nabla_{\mathbf{u}_2} J \quad \cdots \quad \nabla_{\mathbf{u}_P} J) \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \\ &= (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P && (\text{F})\end{aligned}$$

where $\mathbf{1}_P = (1, 1, \dots, 1)^T$ is a vector of P ones.

GD for single layer

From (E) and (F):

$$\begin{aligned}\nabla_{\mathbf{W}} J &= \mathbf{X}^T \nabla_{\mathbf{U}} J \\ \nabla_{\mathbf{b}} J &= (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P\end{aligned}$$

That is, by computing gradient $\nabla_{\mathbf{U}} J$ with respect to synaptic input, the weights and biases can be updated.

$$\begin{aligned}\mathbf{W} &= \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{U}} J \\ \mathbf{b} &= \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P\end{aligned}$$

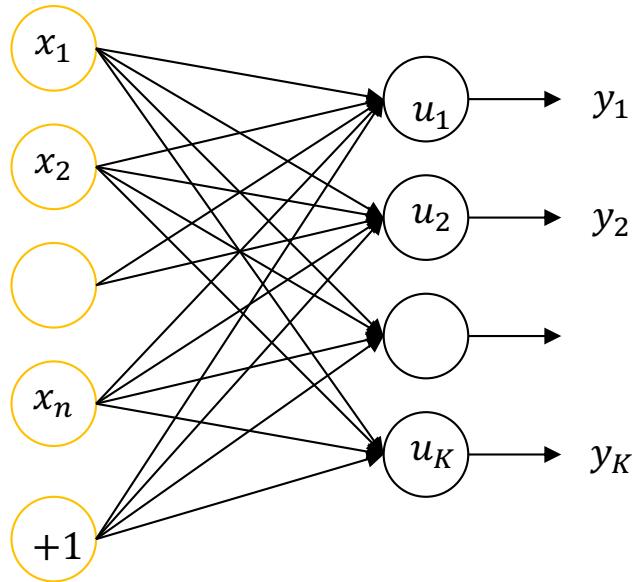
Learning a single layer

Learning a layer of neurons	
SGD	$\mathbf{W} = \mathbf{W} - \alpha \mathbf{x} (\nabla_{\mathbf{u}} J)^T$ $\mathbf{b} = \mathbf{b} - \alpha \nabla_{\mathbf{u}} J$
GD	$\mathbf{W} = \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{U}} J$ $\mathbf{b} = \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P$

To learn a given layer, we need to compute
 $\nabla_{\mathbf{u}} J$ for SGD and $\nabla_{\mathbf{U}} J$ for GD.

Those gradients with respect to synaptic inputs are dependent on the types of neurons in the layer.

Perceptron layer



$$y_k = f(u_k) = \frac{1}{1 + e^{-u_k}}$$

A layer of perceptrons performs **multidimensional non-linear regression** and learns a multidimensional non-linear mapping:

$$\phi: \mathbf{R}^n \rightarrow \mathbf{R}^K$$

SGD for perceptron layer

Given a training pattern (\mathbf{x}, \mathbf{d})

Note $\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbf{R}^n$ and $\mathbf{d} = (d_1, d_2, \dots, d_K)^T \in \mathbf{R}^K$.

The square-error cost function:

$$J = \frac{1}{2} \sum_{k=1}^K (d_k - y_k)^2$$

where $y_k = f(u_k) = \frac{1}{1+e^{-u_k}}$ and $u_k = \mathbf{x}^T \mathbf{w}_k + b_k$.

Gradient of J with respect to u_k :

$$\frac{\partial J}{\partial u_k} = \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial u_k} = -(d_k - y_k) \frac{\partial y_k}{\partial u_k} = -(d_k - y_k) f'(u_k) \quad (\text{G})$$

SGD for perceptron layer

Substituting $\nabla_{u_k} J = \frac{\partial J}{\partial u_k} = -(d_k - y_k)f'(u_k)$ from (G):

$$\nabla_{\mathbf{u}} J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = - \begin{pmatrix} (d_1 - y_1)f'(u_1) \\ (d_2 - y_2)f'(u_2) \\ \vdots \\ (d_K - y_K)f'(u_K) \end{pmatrix} = -(\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u}) \quad (\text{H})$$

and ‘.’ denotes element-wise multiplication.

$$\nabla_{\mathbf{u}} J = -(\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$$

SGD for perceptron layer

Given a training dataset $\{(x, d)\}$

Set learning parameter α

Initialize W and b

Repeat until convergence:

For every pattern (x, d) :

$$u = W^T x + b$$

$$y = f(u) = \frac{1}{1+e^{-u}}$$

$$\nabla_u J = -(d - y) \cdot f'(u)$$

$$W = W - \alpha x (\nabla_u J)^T$$

$$b = b - \alpha \nabla_u J$$

GD for perceptron layer

Given a training dataset $\{(\mathbf{x}_p, \mathbf{d}_p)\}_{p=1}^P$

Note $\mathbf{x}_p = (x_{p1}, x_{p2}, \dots, x_{pn})^T \in \mathbb{R}^n$ and $\mathbf{d}_p = (d_{p1}, d_{p2}, \dots, d_{pK})^T \in \mathbb{R}^K$.

The cost function J is given by the sum of square errors (s.s.e.):

$$J = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2$$

J can be written as the sum of cost due to individual patterns:

$$J = \sum_{p=1}^P J_p$$

where $J_p = \frac{1}{2} \sum_{k=1}^K (d_{pk} - y_{pk})^2$ is the square error for the p th pattern.

GD for perceptron layer

$$\mathbf{U} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix} \rightarrow \nabla_{\mathbf{U}} J = \begin{pmatrix} (\nabla_{\mathbf{u}_1} J)^T \\ (\nabla_{\mathbf{u}_2} J)^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{\mathbf{u}_1} J_1)^T \\ (\nabla_{\mathbf{u}_2} J_2)^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J_P)^T \end{pmatrix}$$

From (H), substitute $\nabla_{\mathbf{u}} J = -(\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$:

$$\nabla_{\mathbf{U}} J = - \begin{pmatrix} ((\mathbf{d}_1 - \mathbf{y}_1) \cdot f'(\mathbf{u}_1))^T \\ ((\mathbf{d}_2 - \mathbf{y}_2) \cdot f'(\mathbf{u}_2))^T \\ \vdots \\ ((\mathbf{d}_P - \mathbf{y}_P) \cdot f'(\mathbf{u}_P))^T \end{pmatrix} = - \begin{pmatrix} (\mathbf{d}_1^T - \mathbf{y}_1^T) \cdot f'(\mathbf{u}_1^T) \\ (\mathbf{d}_2^T - \mathbf{y}_2^T) \cdot f'(\mathbf{u}_2^T) \\ \vdots \\ (\mathbf{d}_P^T - \mathbf{y}_P^T) \cdot f'(\mathbf{u}_P^T) \end{pmatrix}$$

$$\nabla_{\mathbf{U}} J = -(\mathbf{D} - \mathbf{Y}) \cdot f'(\mathbf{U})$$

$$\text{where } \mathbf{D} = \begin{pmatrix} \mathbf{d}_1^T \\ \mathbf{d}_2^T \\ \vdots \\ \mathbf{d}_P^T \end{pmatrix}, \mathbf{Y} = \begin{pmatrix} \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_P^T \end{pmatrix}, \text{ and } \mathbf{U} = \begin{pmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_P^T \end{pmatrix}$$

GD for perceptron layer

Given a training dataset (X, D)

Set learning parameter α

Initialize W and b

Repeat until convergence:

$$U = XW + B$$

$$Y = f(U) = \frac{1}{1+e^{-U}}$$

$$\nabla_U J = -(D - Y) \cdot f'(U)$$

$$W = W - \alpha X^T \nabla_U J$$

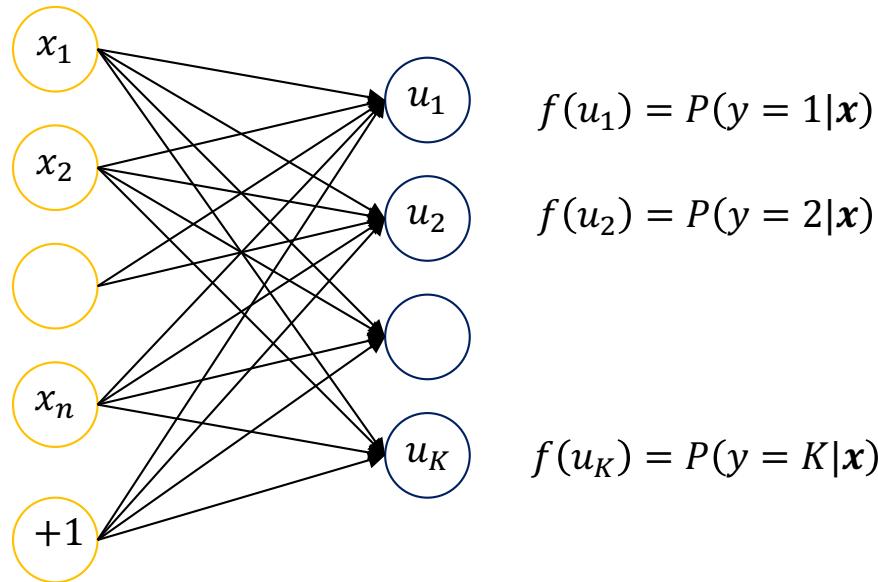
$$b = b - \alpha (\nabla_U J)^T \mathbf{1}_P$$

Learning a perceptron layer

GD	SGD
(X, D)	(x, d)
$U = XW + B$	$u = W^T x + b$
$Y = f(U)$	$y = f(u)$
$\nabla_U J = -(D - Y) \cdot f'(U)$	$\nabla_u J = -(d - y) \cdot f'(u)$
$W = W - \alpha X^T \nabla_U J$	$W = W - \alpha x (\nabla_u J)^T$
$b = b - \alpha (\nabla_U J)^T \mathbf{1}_P$	$b = b - \alpha \nabla_u J$

Softmax layer

Softmax layer is the extension of logistic regression to **multiclass classification** problem, which is also known as *multinomial logistic regression*.



Each neuron in the softmax layer corresponds to one class label. The activation of a neuron gives the probability of the input belonging to that class label. Output is the class label having the maximum probability.

Softmax layer

The K neurons in the softmax layer performs K class classification and represent K classes.

The activation of each neuron k estimates the probability $P(y = k|x)$ that the input \mathbf{x} belongs to the class k :

$$P(y = k|\mathbf{x}) = f(u_k) = \frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}}$$

where $u_k = \mathbf{w}_k^T \mathbf{x} + b_k$, and \mathbf{w}_k is weight vector and b_k is bias of neuron k .

The above activation function f is known as **softmax activation function**.

Softmax layer

The output y denotes the class label of the input pattern, which is given by

$$y = \operatorname{argmax}_k P(y = k | \mathbf{x}) = \operatorname{argmax}_k f(u_k)$$

That is, the class label is assigned to the class with the maximum activation.

SGD for softmax layer

Given a training pattern (\mathbf{x}, d) where $\mathbf{x} \in \mathbb{R}^n$ and $d \in \{1, 2, \dots, K\}$.

The cost function for learning is by the *multiclass cross-entropy*:

$$J = - \sum_{k=1}^K 1(d = k) \log(f(u_k))$$

where u_k is the synaptic input to the k the neuron.

The cost function can also be written as

$$J = -\log(f(u_d))$$

where d is the target label of input \mathbf{x} .

Note that the logarithm here is natural: $\log = \log_e$

SGD for softmax layer

$$J = -\log(f(u_d))$$

The gradient with respect to u_k is given by

$$\frac{\partial J}{\partial u_k} = -\frac{1}{f(u_d)} \frac{\partial f(u_d)}{\partial u_k} \quad (\text{I})$$

where

$$\frac{\partial f(u_d)}{\partial u_k} = \frac{\partial}{\partial u_k} \left(\frac{e^{u_d}}{\sum_{k'=1}^K e^{u_{k'}}} \right)$$

The above differentiation need to be considered separately for $k = d$ and for $k \neq d$.

SGD for softmax layer

If $k = d$:

$$\begin{aligned}\frac{\partial f(u_d)}{\partial u_k} &= \frac{\partial}{\partial u_k} \left(\frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}} \right) \\ &= \frac{(\sum_{k'=1}^K e^{u_{k'}}) e^{u_k} - e^{u_k} e^{u_k}}{(\sum_{k'=1}^K e^{u_{k'}})^2} \\ &= \frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}} \left(1 - \frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}} \right) \\ &= f(u_k)(1 - f(u_k)) \\ &= f(u_d)(1(k=d) - f(u_k))\end{aligned}$$

$\frac{\partial (\sum_{k'=1}^K e^{u_{k'}})}{\partial u_k} = e^{u_k}$

If $k \neq d$:

$$\begin{aligned}\frac{\partial f(u_d)}{\partial u_k} &= \frac{\partial}{\partial u_k} \left(\frac{e^{u_d}}{\sum_{k'=1}^K e^{u_{k'}}} \right) \\ &= -\frac{e^{u_d} e^{u_k}}{(\sum_{k'=1}^K e^{u_{k'}})^2} \\ &= -f(u_d)f(u_k) \\ &= f(u_d)(1(k=d) - f(u_k))\end{aligned}$$

$1(k=d) = 0$

SGD for softmax layer

$$\frac{\partial f(u_d)}{\partial u_k} = f(u_d)(1(d=k) - f(u_k))$$

Substituting in (I):

$$\nabla_{u_k} J = \frac{\partial J}{\partial u_k} = -\frac{1}{f(u_d)} \frac{\partial f(u_d)}{\partial u_k} = -(1(d=k) - f(u_k))$$

Gradient J with respect to \mathbf{u} :

$$\nabla_{\mathbf{u}} J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = - \begin{pmatrix} 1(d=1) - f(u_1) \\ 1(d=2) - f(u_2) \\ \vdots \\ 1(d=K) - f(u_K) \end{pmatrix} = -(1(\mathbf{k}=d) - f(\mathbf{u})) \quad (\text{J})$$

where $\mathbf{k} = (1 \quad 2 \quad \dots \quad K)^T$

SGD for softmax layer

For a softmax layer:

$$\nabla_{\mathbf{u}} J = - (1(\mathbf{k} = d) - f(\mathbf{u}))$$

where:

$$1(\mathbf{k} = d) = \begin{pmatrix} 1(d = 1) \\ 1(d = 2) \\ \vdots \\ 1(d = K) \end{pmatrix} \text{ and } f(\mathbf{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_K) \end{pmatrix}$$

Note that $1(\mathbf{k} = d)$ is a one-hot vector where the element corresponding to the target label d is ‘1’ and elsewhere is ‘0’.

GD for softmax layer

Given a set of patterns $\{(\mathbf{x}_p, d_p)\}_{p=1}^P$ where $\mathbf{x}_p \in \mathbb{R}^n$ and $d_p \in \{1, 2, \dots, K\}$.

The cost function of the *softmax layer* is given by the *multiclass cross-entropy*:

$$J = - \sum_{p=1}^P \left(\sum_{k=1}^K 1(d_p = k) \log(f(u_{pk})) \right)$$

where u_{pk} is the synaptic input to the k the neuron for input \mathbf{x}_p .

The cost function J can also be written as

$$J = - \sum_{p=1}^P \log(f(u_{pd_p}))$$

GD for softmax layer

J can be written as the sum of cost due to individual patterns:

$$J = \sum_{p=1}^P J_p$$

where $J_p = -\log(f(u_{pd_p}))$ is the cross-entropy for the p th pattern.

GD for softmax layer

$$\nabla_{\mathbf{U}} J = \begin{pmatrix} (\nabla_{\mathbf{u}_1} J)^T \\ (\nabla_{\mathbf{u}_2} J)^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{\mathbf{u}_1} J_1)^T \\ (\nabla_{\mathbf{u}_2} J_2)^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J_P)^T \end{pmatrix}$$

Substituting $\nabla_{\mathbf{u}} J = -(1(\mathbf{k} = d) - f(\mathbf{u}))$ from (J):

$$\nabla_{\mathbf{U}} J = - \begin{pmatrix} (1(\mathbf{k} = d_1) - f(\mathbf{u}_1))^T \\ (1(\mathbf{k} = d_2) - f(\mathbf{u}_2))^T \\ \vdots \\ (1(\mathbf{k} = d_P) - f(\mathbf{u}_P))^T \end{pmatrix}$$

$$\nabla_{\mathbf{U}} J = -(\mathbf{K} - f(\mathbf{U}))$$

where $\mathbf{K} = \begin{pmatrix} 1(\mathbf{k} = d_1)^T \\ 1(\mathbf{k} = d_2)^T \\ \vdots \\ 1(\mathbf{k} = d_P)^T \end{pmatrix}$ is a matrix with every row is a one-hot vector.

Learning a softmax layer

GD	SGD
(X, \mathbf{D})	(\mathbf{x}, d)
$\mathbf{U} = \mathbf{XW} + \mathbf{B}$	$\mathbf{u} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$
$f(\mathbf{U}) = \frac{e^{\mathbf{U}}}{\sum_{k'=1}^K e^{\mathbf{U}_{k'}}}$	$f(\mathbf{u}) = \frac{e^{u_k}}{\sum_{k'=1}^K e^{u_{k'}}}$
$\mathbf{y} = \underset{k}{\operatorname{argmax}} f(\mathbf{U})$	$y = \underset{k}{\operatorname{argmax}} f(\mathbf{u})$
$\nabla_{\mathbf{U}} J = -(\mathbf{K} - f(\mathbf{U}))$	$\nabla_{\mathbf{u}} J = -(1(\mathbf{k} = d) - f(\mathbf{u}))$
$\mathbf{W} = \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{U}} J$	$\mathbf{W} = \mathbf{W} - \alpha \mathbf{x} (\nabla_{\mathbf{u}} J)^T$
$\mathbf{b} = \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P$	$\mathbf{b} = \mathbf{b} - \alpha \nabla_{\mathbf{u}} J$

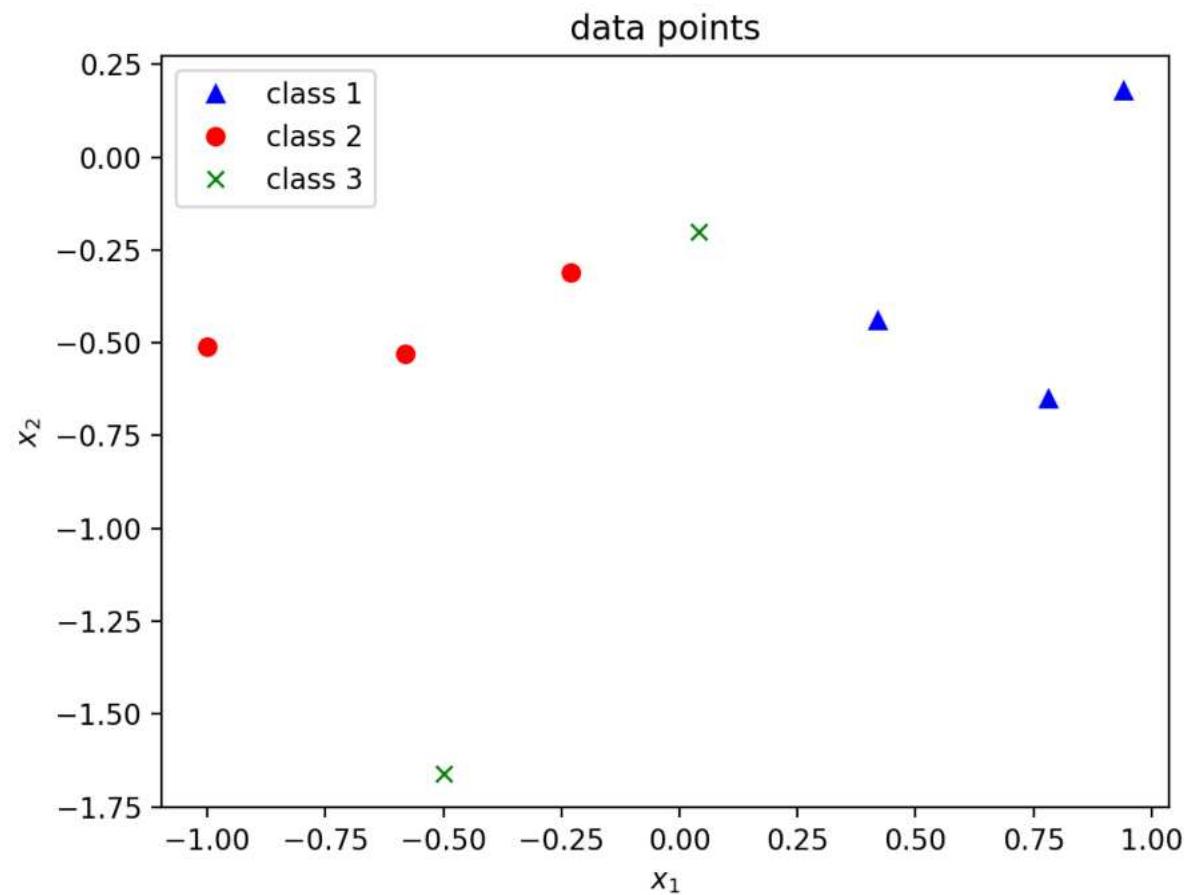
Example 2: GD of a softmax layer

Train a softmax regression layer of neurons to perform the following classification:

- (0.94 0.18) → *class A*
- (−0.58 −0.53) → *class B*
- (−0.23 −0.31) → *class B*
- (0.42 −0.44) → *class A*
- (0.5 −1.66) → *class C*
- (−1.0 −0.51) → *class B*
- (0.78 −0.65) → *class A*
- (0.04 −0.20) → *class C*

Use a learning factor $\alpha = 0.05$.

Example 2

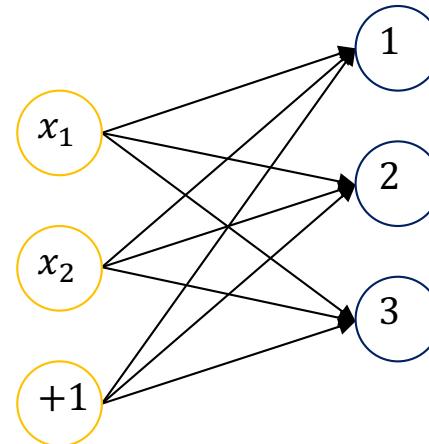


Example 2

Let $y = \begin{cases} 1, & \text{for class } A \\ 2, & \text{for class } B \\ 3, & \text{for class } C \end{cases}$

$$X = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}, d = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \\ 1 \\ 3 \end{pmatrix}$$

$$K = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



Example 2

Initialize $\mathbf{W} = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix}$, $\mathbf{b} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$,

1st epoch starts ...

Example 2

$$U = \begin{pmatrix} 0.86 & 0.11 & 0.64 \\ -0.84 & -0.28 & -0.49 \\ -0.41 & -0.16 & -0.22 \\ -0.01 & -0.21 & 0.17 \\ -0.86 & -0.82 & -0.06 \\ -1.15 & -0.27 & -0.75 \\ 0.11 & -0.31 & 0.35 \\ -0.12 & -0.10 & -0.02 \end{pmatrix}$$

$$f(u_{12}) = \frac{e^{0.11}}{e^{0.86} + e^{0.11} + e^{0.64}}$$

$$f(U) = \frac{e^{(U)}}{\sum_{k=1}^K e^{(U)}} = \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix}$$

Example 2

$$\mathbf{y} = \underset{k}{\operatorname{argmax}} \{f(\mathbf{U})\} = \underset{k}{\operatorname{argmax}} \left\{ \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix} \right\} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 3 \\ 2 \\ 3 \\ 3 \end{pmatrix}$$

$$\text{Errors} = \sum_{p=1}^8 1(d_p \neq y_p) = 2$$

$$\begin{aligned} \text{Entropy, } J &= - \sum_{p=1}^8 \log \left(f(u_{pd_p}) \right) \\ &= -\log(0.44) - \log(0.42) - \dots - \log(0.35) \\ &= 7.26 \end{aligned}$$

$$\mathbf{d} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \\ 1 \\ 3 \end{pmatrix}$$

Example 2

$$\nabla_U J = -(\mathbf{K} - f(\mathbf{U}))$$

$$= - \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix} \right)$$

$$= \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix}$$

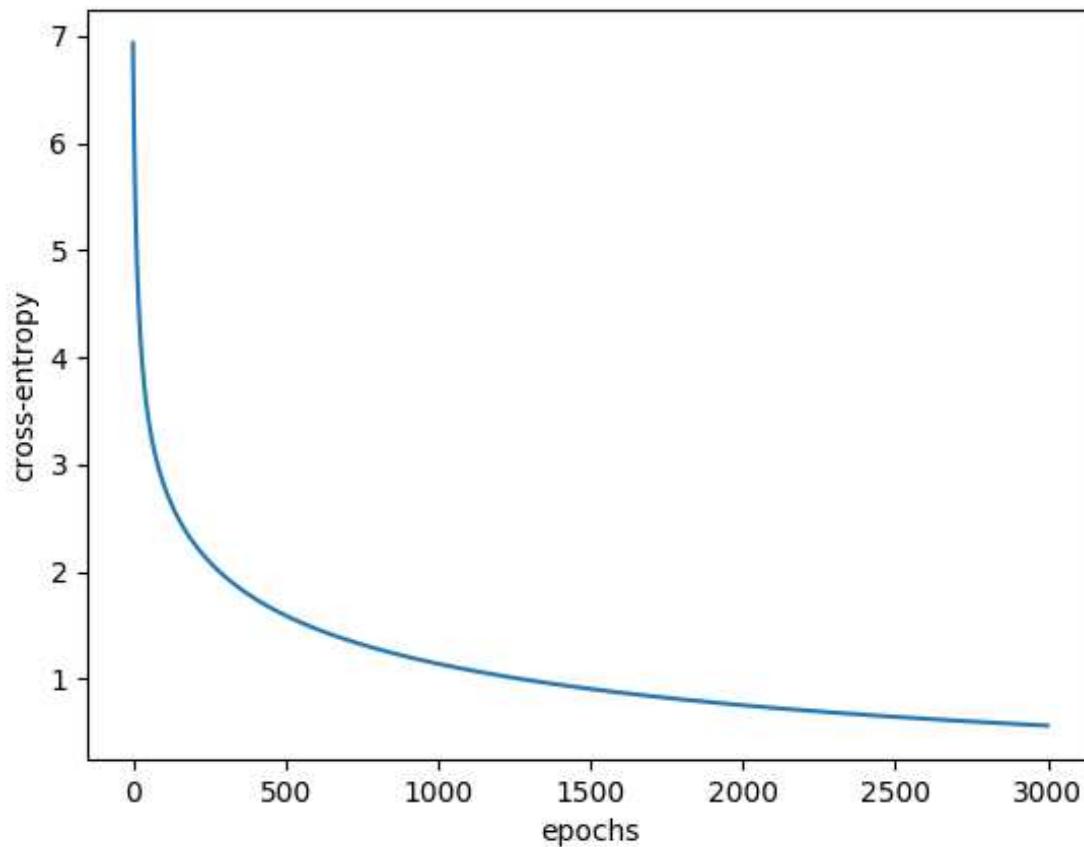
$$\mathbf{W} = \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{U}} J$$

$$= \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix} - 0.05 \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}^T \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix}$$

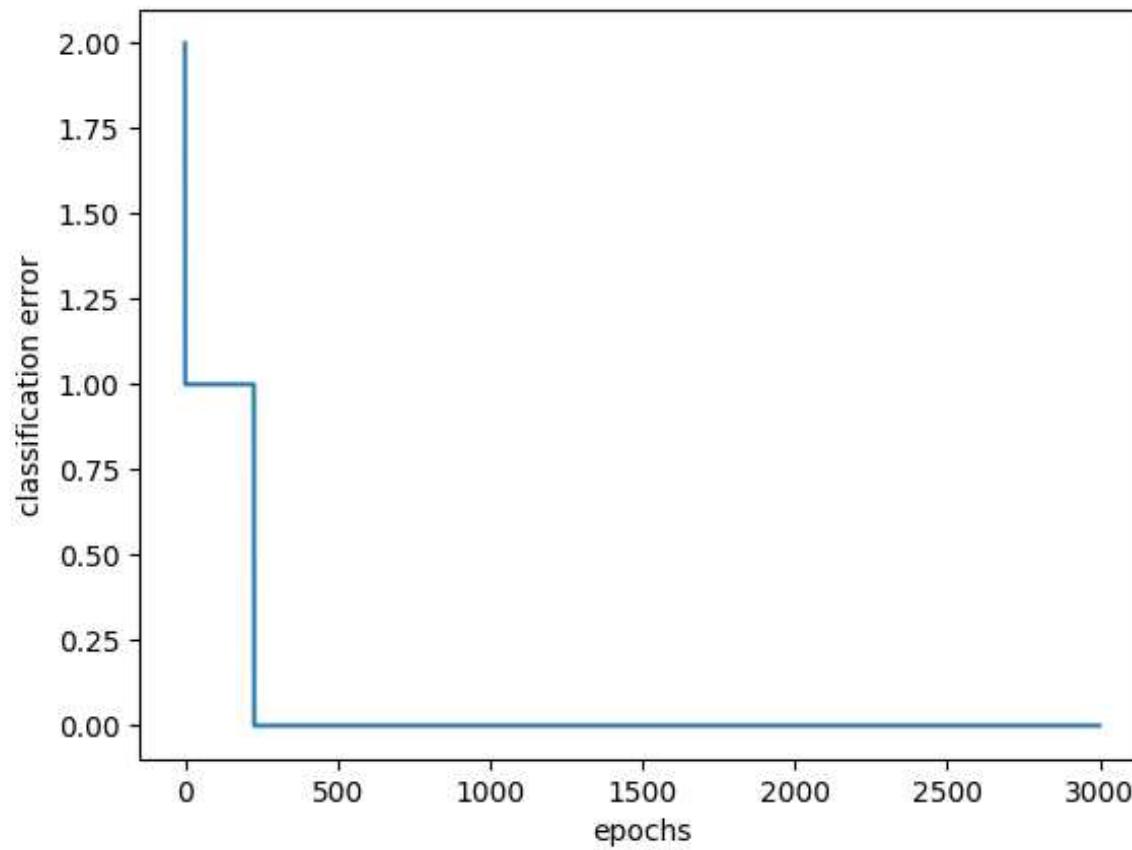
$$= \begin{pmatrix} 0.85 & -0.06 & 0.63 \\ 0.76 & 0.50 & 0.22 \end{pmatrix}$$

$$\mathbf{b} = \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix} - 0.05 \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.03 \\ 0.02 \\ -0.05 \end{pmatrix}$$

Example 2



Example 2



Example 2

At convergence at 3000 iterations:

$$\mathbf{w} = \begin{pmatrix} 14.22 & -13.04 & 0.00 \\ 4.47 & -2.05 & -0.95 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} -0.53 \\ -0.47 \\ 1.00 \end{pmatrix}$$

Entropy = 0.562

Errors = 0

Initialization of weights

Random initialization is inefficient

At initialization, it is desirable that weights are small and near zero

- to operate in the linear region of the activation function
- to preserve the variance of activations and gradients.

Two methods:

- Using a uniform distribution within specified limits
- Using a truncated normal distribution

Initialization from a uniform distribution (Xavier/Glorot uniform Initialization)

Uniformly draws weight samples $w \sim U(-a, +a)$:

where

$$a = gain \times \sqrt{\frac{6}{n_{in} + n_{out}}}$$

n_{in} is the number of input nodes and n_{out} is the number of neurons in the layer.

activation	linear	sigmoid	Tanh	ReLU	Leaky ReLU
<i>gain</i>	1	1	5/3	$\sqrt{2}$	$\sqrt{\frac{1}{1 + slope^2}}$

Initialization from a truncated normal distribution

$$w \sim \text{truncated_normal} \left[\text{mean} = 0, \text{std} = \frac{\text{gain}}{\sqrt{n_{in}}} \right]$$

In the truncated normal, the samples that are two s.d. away from the center are discarded and resampled again.

This is also known as **Kaiming normal initialization**.

Iris dataset

Iris dataset:

<https://archive.ics.uci.edu/ml/datasets/Iris>

Three classes of iris flower:



Setosa



Versicolour



Virginica

Four features:

Sepal length, sepal width, petal length, petal width

Iris dataset

150 data points, 50 for each class

Features:

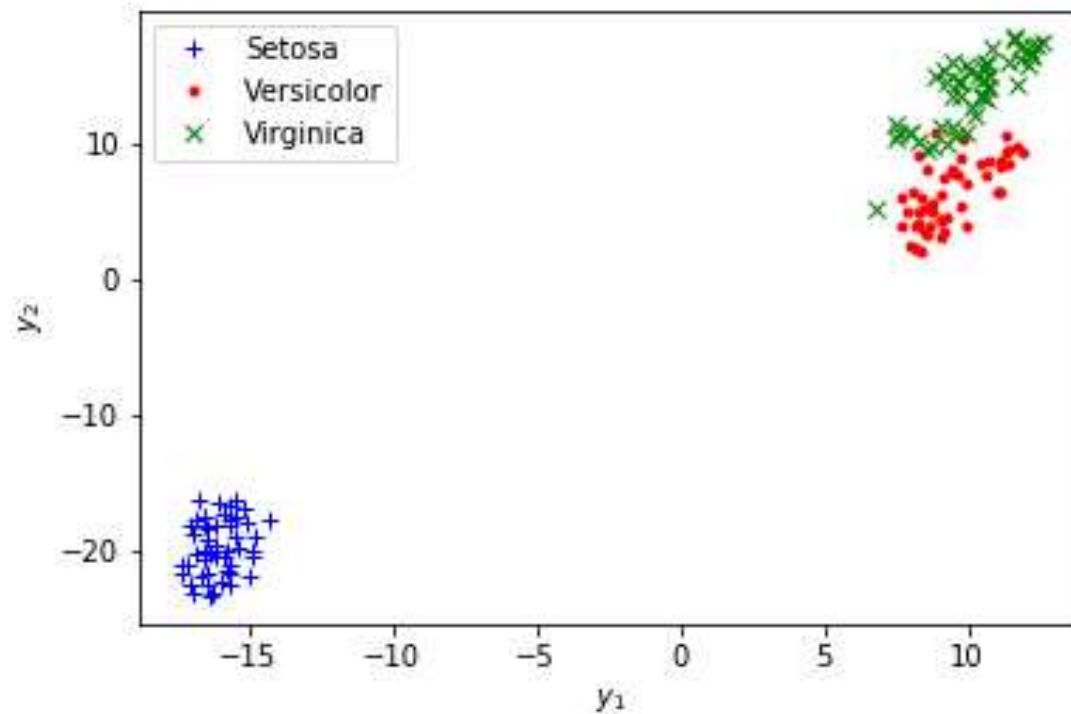
```
[ -7.4333333e-01  4.4600000e-01 -2.35866667e+00 -9.98666667e-01]
[ -9.4333333e-01 -5.4000000e-02 -2.35866667e+00 -9.98666667e-01]
[ -1.1433333e+00  1.4600000e-01 -2.45866667e+00 -9.98666667e-01]
```

Labels:

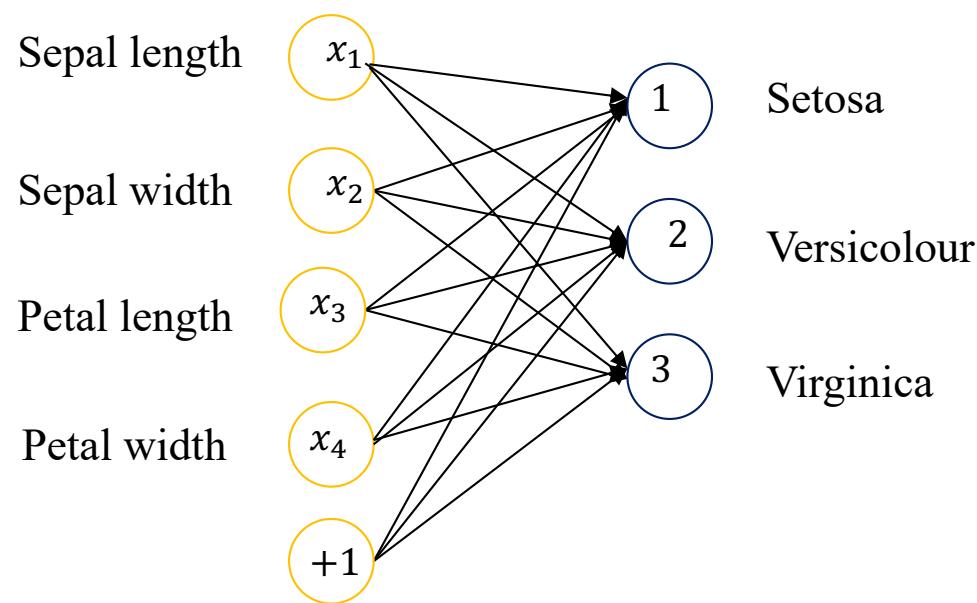
[0 0 0 0 0 0 0 0 0 0 ..1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2]

Iris data

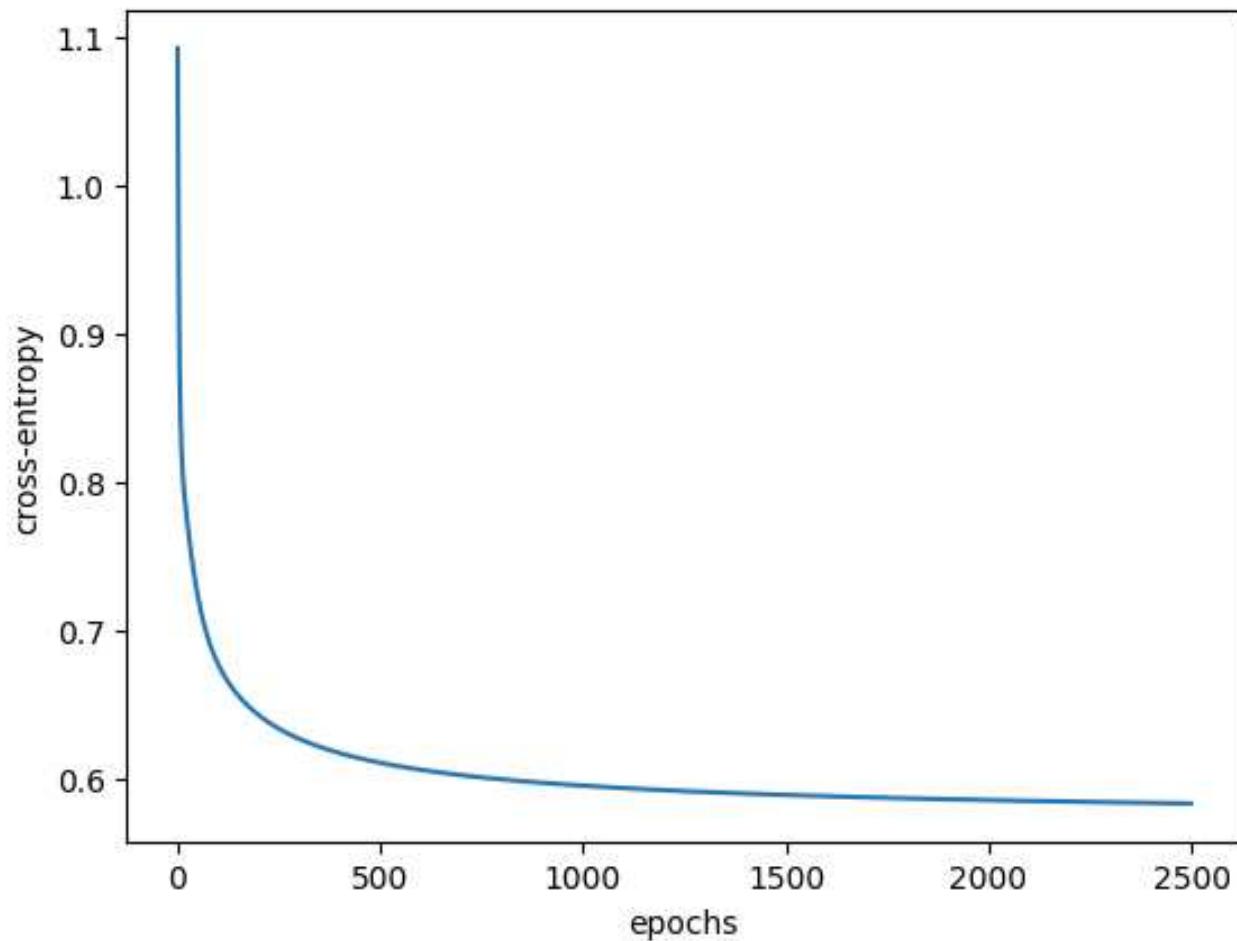
Display of data points after dimensionality reduction (from Four dimensions to Two) by t-SNE.



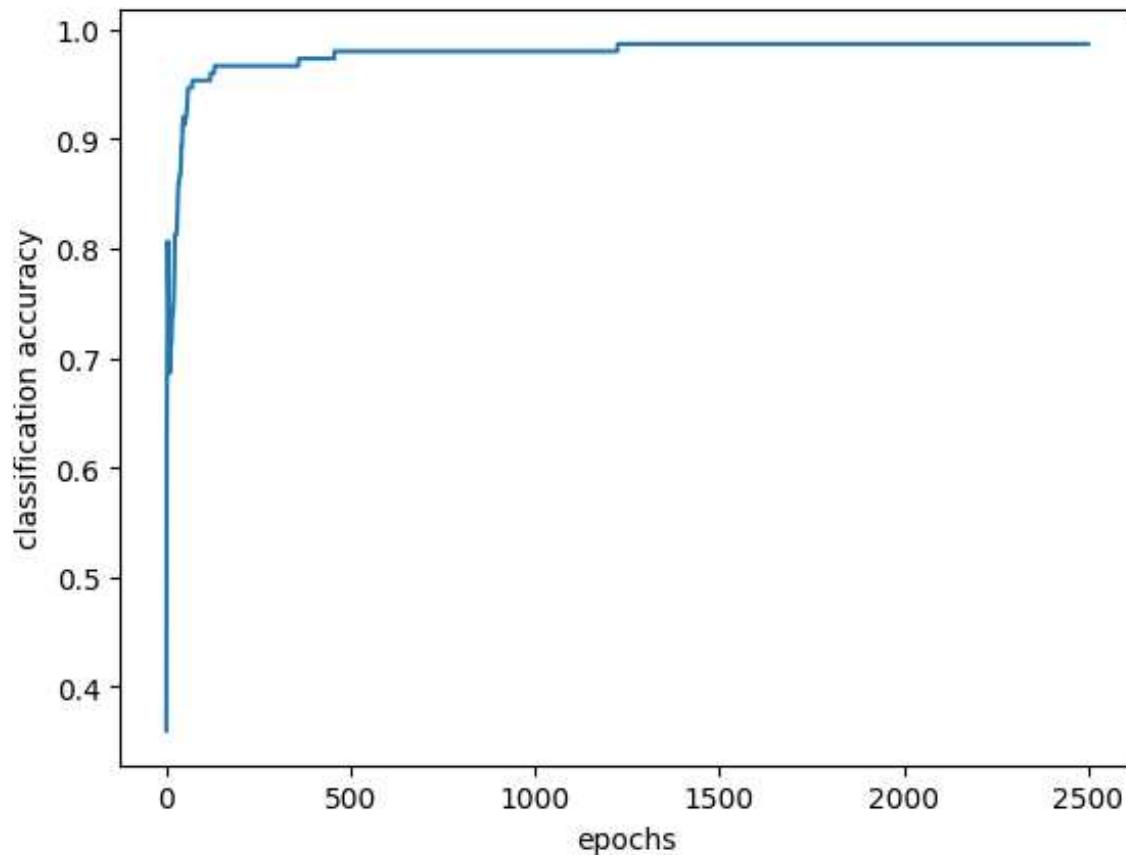
Example 3: Softmax classification of iris data



Example 3



Example 3



Final classification error = 5

Revision: neurons and layers

Classification	Logistic neurons
Two-class	Logistic regression neuron
Multiclass	Softmax layer

Regression	Linear	Non-linear
One dimensional	Linear neuron	Perceptron
Multi-dimensional	Linear neuron layer	Perceptron layer

Summary: GD for layers

$$\begin{aligned}
 & (\mathbf{X}, \mathbf{D}) \\
 & \mathbf{U} = \mathbf{XW} + \mathbf{B} \\
 & \mathbf{W} = \mathbf{W} - \alpha \mathbf{X}^T (\nabla_{\mathbf{U}} J) \\
 & \mathbf{b} = \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P
 \end{aligned}$$

layer	$f(\mathbf{U}), \mathbf{Y}$	$\nabla_{\mathbf{U}} J$
Linear neuron layer	$\mathbf{Y} = f(\mathbf{U}) = \mathbf{U}$	$-(\mathbf{D} - \mathbf{Y})$
Perceptron layer	$\mathbf{Y} = f(\mathbf{U}) = \frac{1}{1 + e^{-\mathbf{U}}}$	$-(\mathbf{D} - \mathbf{Y}) \cdot f'(\mathbf{U})$
Softmax layer	$f(\mathbf{U}) = \frac{e^{\mathbf{U}}}{\sum_{k=1}^K e^{\mathbf{U}_k}}$ $\mathbf{y} = \underset{k}{\operatorname{argmax}} f(\mathbf{U})$	$-(\mathbf{K} - f(\mathbf{U}))$

Summary: SGD for layers

$$\begin{aligned}
 & (\mathbf{x}, \mathbf{d}) \\
 \mathbf{u} &= \mathbf{W}^T \mathbf{x} + \mathbf{b} \\
 \mathbf{W} &= \mathbf{W} - \alpha \mathbf{x} (\nabla_{\mathbf{u}} J)^T \\
 \mathbf{b} &= \mathbf{b} - \alpha (\nabla_{\mathbf{u}} J)
 \end{aligned}$$

layer	$f(\mathbf{u}), \mathbf{y}$	$\nabla_{\mathbf{u}} J$
Linear neuron layer	$\mathbf{y} = f(\mathbf{u}) = \mathbf{u}$	$-(\mathbf{d} - \mathbf{y})$
Perceptron layer	$\mathbf{y} = f(\mathbf{u}) = \frac{1}{1 + e^{-\mathbf{u}}}$	$-(\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u})$
Softmax layer	$f(\mathbf{u}) = \frac{e^{\mathbf{u}}}{\sum_{k'=1}^K e^{k'}}$ $\mathbf{y} = \underset{k}{\operatorname{argmax}} f(\mathbf{u})$	$-(1(\mathbf{k} = d) - f(\mathbf{u}))$

Thank you.

Example 4: GD of a perceptron layer

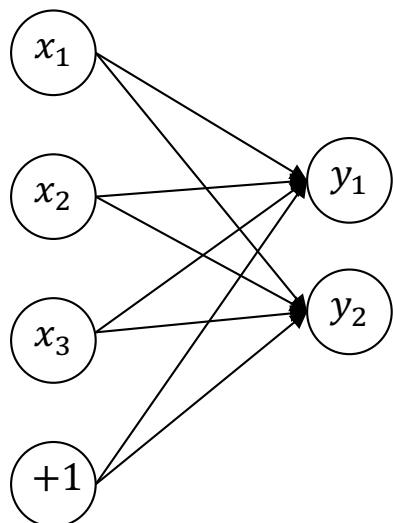
Design a perceptron layer to perform the following mapping using GD learning:

$x = (x_1, x_2, x_3)$	$d = (d_1, d_2)$
(0.77, 0.02, 0.63)	(0.37, 0.47)
(0.75, 0.50, 0.22)	(0.36, 0.38)
(0.20, 0.76, 0.17)	(0.35, 0.25)
(0.09, 0.69, 0.95)	(0.48, 0.42)
(0.00, 0.51, 0.81)	(0.36, 0.29)
(0.61, 0.72, 0.29)	(0.44, 0.52)
(0.92, 0.71, 0.54)	(0.60, 0.52)
(0.14, 0.37, 0.67)	(0.28, 0.37)

Use $\alpha = 0.1$.

Example 4

$$X = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0.37 & 0.47 \\ 0.36 & 0.38 \\ 0.35 & 0.25 \\ 0.48 & 0.42 \\ 0.36 & 0.29 \\ 0.44 & 0.52 \\ 0.60 & 0.52 \\ 0.28 & 0.37 \end{pmatrix}$$



Output $y_1, y_2 \in [0, 1]$

So, activation function for both neurons:

$$f(u) = \frac{1}{1 + e^{-u}}$$
$$f'(u) = y(1 - y)$$

Learning factor $\alpha = 0.1$.

Weights and biases are initialized:

$$W = \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} \text{ and } b = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

Example 4

Epoch 1:

$$\mathbf{U} = \mathbf{XW} + \mathbf{B}$$

$$\mathbf{U} = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix} \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 0.03 & 0.06 \\ 0.03 & 0.06 \\ 0.02 & 0.05 \\ 0.03 & 0.07 \\ 0.02 & 0.05 \\ 0.03 & 0.07 \\ 0.04 & 0.09 \\ 0.02 & 0.05 \end{pmatrix}$$

$$\mathbf{Y} = f(\mathbf{U}) = \frac{1}{1+e^{-\mathbf{U}}} = \begin{pmatrix} 0.51 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \end{pmatrix}$$

$$\text{Mean square error} = \frac{1}{8} \sum_{p=1}^8 \sum_{k=1}^2 (d_{pk} - y_{pk})^2 = \frac{1}{8} \sum_{p=1}^8 (d_{p1} - y_{p1})^2 + (d_{p2} - y_{p2})^2 = 0.04$$

Example 4

$$f'(U) = Y \cdot (1 - Y) = \begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix}$$

$$\begin{aligned} \nabla_U J &= -(D - Y) \cdot f'(U) \\ &= - \left(\begin{pmatrix} 0.37 & 0.47 \\ 0.36 & 0.38 \\ 0.35 & 0.25 \\ 0.48 & 0.42 \\ 0.36 & 0.29 \\ 0.44 & 0.52 \\ 0.60 & 0.52 \\ 0.28 & 0.37 \end{pmatrix} - \begin{pmatrix} 0.51 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \end{pmatrix} \right) \cdot \begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix} = \begin{pmatrix} 0.04 & 0.01 \\ 0.04 & 0.03 \\ 0.04 & 0.07 \\ 0.01 & 0.03 \\ 0.04 & 0.06 \\ 0.02 & 0.00 \\ -0.02 & 0.00 \\ 0.06 & 0.04 \end{pmatrix} \end{aligned}$$

Example 4

$$\mathbf{W} = \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{U}} J$$

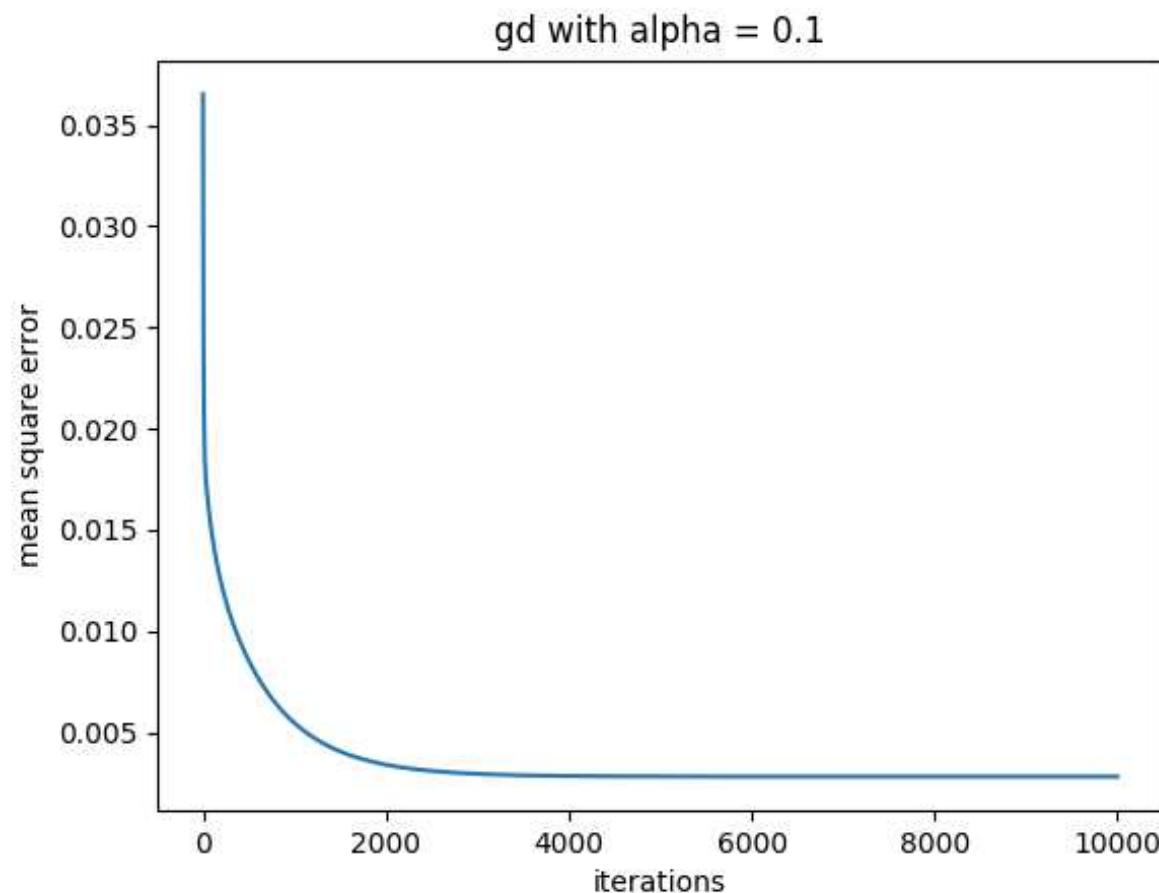
$$= \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} - 0.1 \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix}^T \begin{pmatrix} 0.04 & 0.01 \\ 0.04 & 0.03 \\ 0.04 & 0.07 \\ 0.01 & 0.03 \\ 0.04 & 0.06 \\ 0.02 & 0.00 \\ -0.02 & 0.00 \\ 0.06 & 0.04 \end{pmatrix} = \begin{pmatrix} 0.02 & 0.04 \\ 0.00 & 0.03 \\ 0.01 & 0.03 \end{pmatrix}$$

$$\mathbf{b} = \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} - 0.1 \begin{pmatrix} 0.03 & 0.01 \\ 0.03 & 0.03 \\ 0.04 & 0.06 \\ 0.00 & 0.02 \\ 0.03 & 0.05 \\ 0.01 & 0.00 \\ -0.02 & 0.00 \\ 0.05 & 0.03 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.02 \\ -0.02 \end{pmatrix}$$

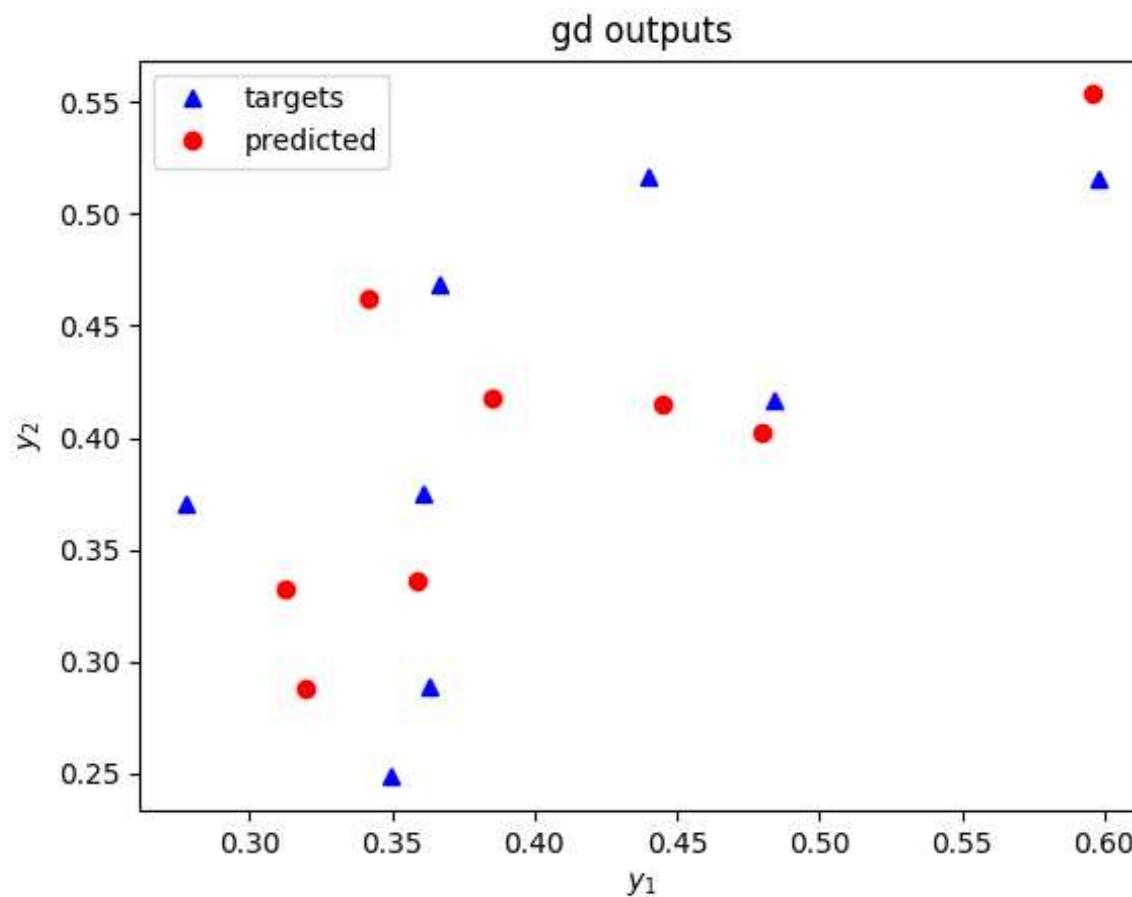
Example 4

Epoch	Y	mse	W	b
2	$\begin{pmatrix} 0.50 & 0.51 \\ 0.50 & 0.51 \\ 0.50 & 0.50 \\ 0.50 & 0.51 \\ 0.50 & 0.50 \\ 0.50 & 0.51 \\ 0.50 & 0.51 \\ 0.50 & 0.50 \end{pmatrix}$	0.036	$\begin{pmatrix} 0.02 & 0.03 \\ -0.01 & 0.02 \\ 0.00 & 0.01 \end{pmatrix}$	$\begin{pmatrix} -0.04 \\ -0.04 \end{pmatrix}$
10000	$\begin{pmatrix} 0.34 & 0.46 \\ 0.39 & 0.42 \\ 0.32 & 0.29 \\ 0.48 & 0.40 \\ 0.36 & 0.34 \\ 0.45 & 0.42 \\ 0.59 & 0.55 \\ 0.31 & 0.33 \end{pmatrix}$	0.003	$\begin{pmatrix} 1.08 & 1.14 \\ 1.42 & 0.40 \\ 1.14 & 0.84 \end{pmatrix}$	$\begin{pmatrix} -2.24 \\ -1.57 \end{pmatrix}$

Example 4



Example 4



Chapter 4

Deep neural networks

Neural networks and deep learning

Chain rule of differentiation

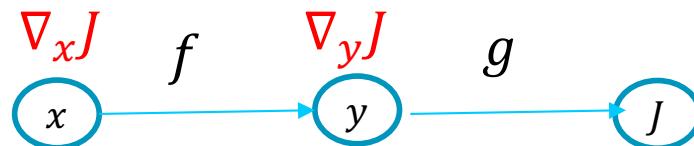
Let x , y , and $J \in \mathbb{R}$ be one-dimensional variables and

$$\begin{aligned} J &= g(y) \\ y &= f(x) \end{aligned}$$

Chain rule of differentiation states that:

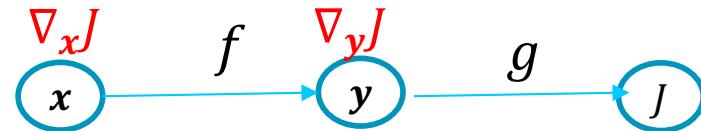
$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial x}$$

$$\nabla_x J = \left(\frac{\partial y}{\partial x} \right) \nabla_y J$$



Note the transfer of gradient of J from y to x .

Chain rule in multidimensions



$\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbf{R}^n$, $\mathbf{y} = (y_1, y_2, \dots, y_K) \in \mathbf{R}^K$, $J \in \mathbf{R}$, and

$$\begin{aligned}\mathbf{y} &= f(\mathbf{x}) \\ J &= g(\mathbf{y})\end{aligned}$$

Then, the chain rule of differentiation states that:

$$\nabla_{\mathbf{x}} J = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{y}} J$$

The matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is known as the **Jacobian** of the function f where $\mathbf{y} = f(\mathbf{x})$.

Chain rule in multidimensions

$$\nabla_{\mathbf{x}} J = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{y}} J$$

where

$$\nabla_{\mathbf{x}} J = \frac{\partial J}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_n} \end{pmatrix} \text{ and } \nabla_{\mathbf{y}} J = \frac{\partial J}{\partial \mathbf{y}} = \begin{pmatrix} \frac{\partial J}{\partial y_1} \\ \frac{\partial J}{\partial y_2} \\ \vdots \\ \frac{\partial J}{\partial y_K} \end{pmatrix},$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \frac{\partial y_K}{\partial x_2} & \dots & \frac{\partial y_K}{\partial x_n} \end{pmatrix}$$

Note that differentiation of a scalar by a vector results in a vector and differentiation of a vector by a vector results in a matrix.

Example 1: find Jacobian of a function

Let $\mathbf{x} = (x_1, x_2, x_3)$, $\mathbf{y} = (y_1, y_2)$, and $\mathbf{y} = f(\mathbf{x})$ where f is given by

$$\begin{aligned}y_1 &= 5 - 2x_1 + 3x_3 \\y_2 &= x_1 + 5x_2^2 + x_3^3 - 1\end{aligned}$$

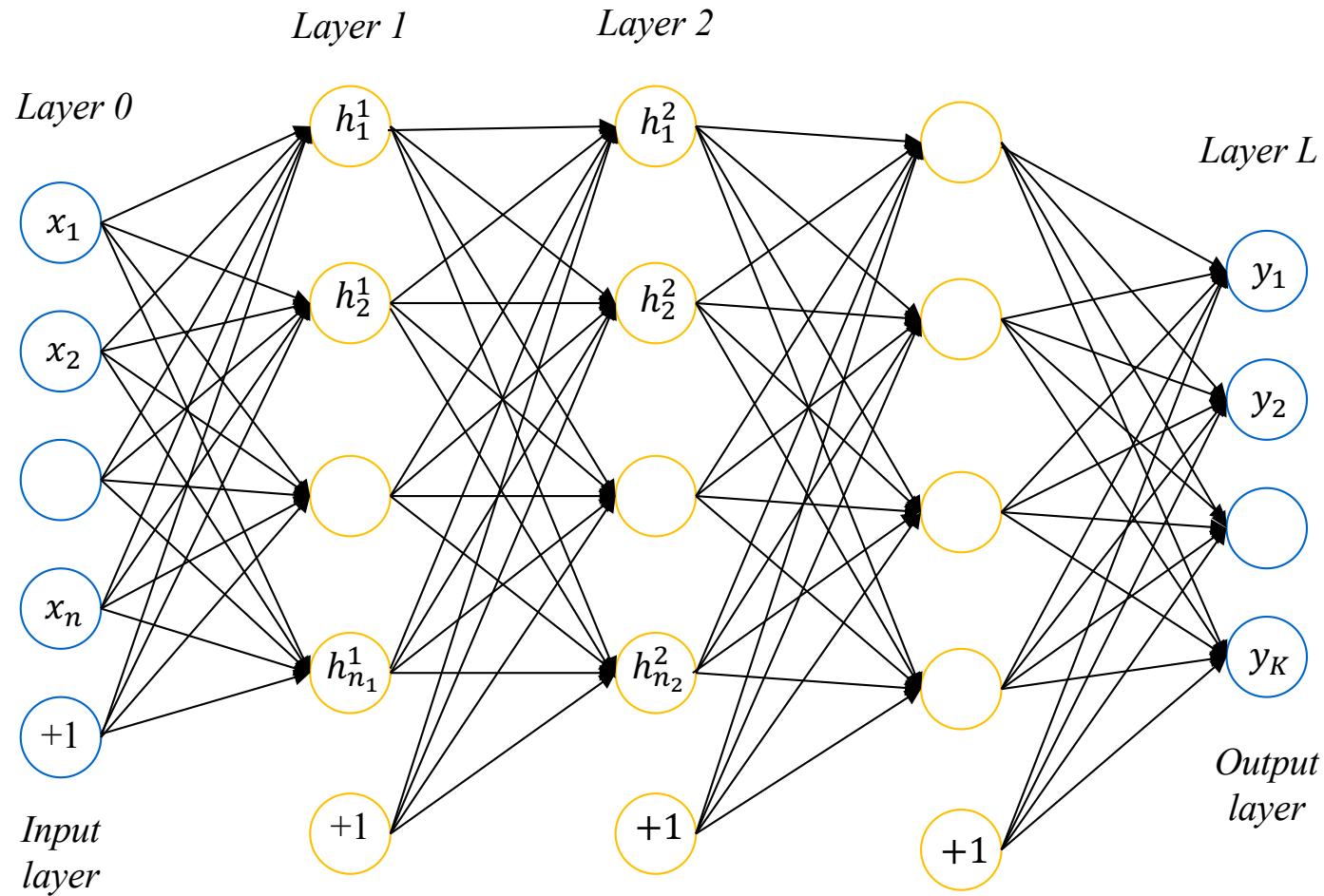
Find the Jacobian of f .

$$\begin{aligned}\frac{\partial y_1}{\partial x_1} &= -2, & \frac{\partial y_1}{\partial x_2} &= 0, & \frac{\partial y_1}{\partial x_3} &= 3 \\ \frac{\partial y_2}{\partial x_1} &= 1, & \frac{\partial y_2}{\partial x_2} &= 10x_2, & \frac{\partial y_2}{\partial x_3} &= 3x_3^2\end{aligned}$$

Jacobian:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{pmatrix} = \begin{pmatrix} -2 & 0 & 3 \\ 1 & 10x_2 & 3x_3^2 \end{pmatrix}$$

Deep neural networks (DNN): Also known as feedforward networks (FFN)



L layer DNN

Feedforward Networks (FFN)

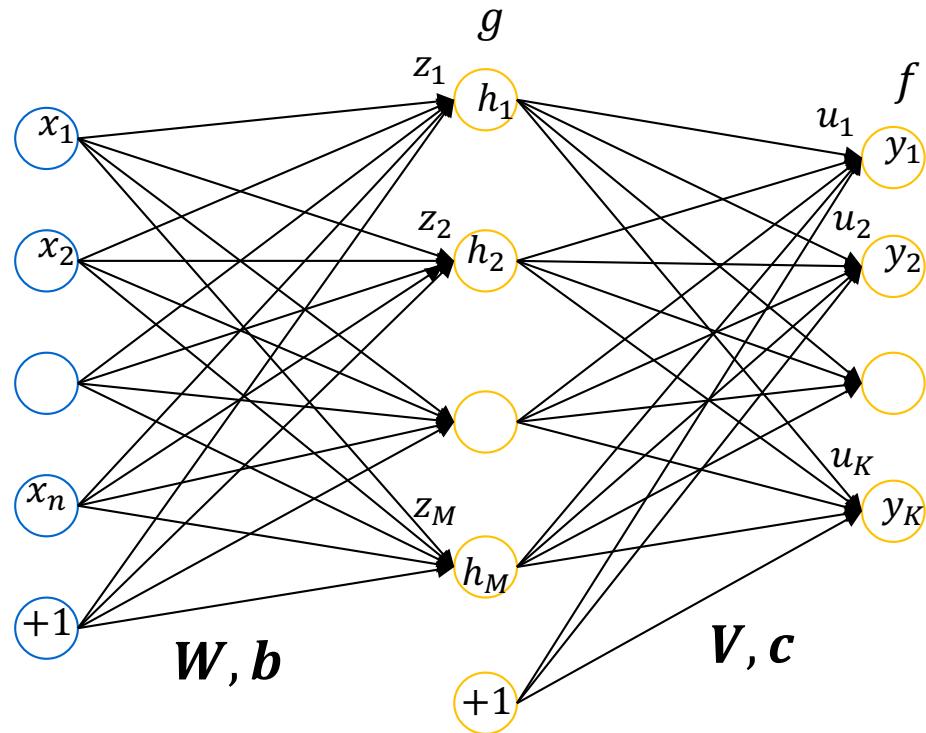
Feedforward networks (FFN) consists of several layers of neurons where activations propagate from input layer to output layer. The layers between the input and output layers are referred to as **hidden layers**.

The number of layers is referred to as the **depth** of the feedforward network. When a network has many hidden layers of neurons, feedforward networks are referred to as **deep neural networks (DNN)**. Learning in deep neural networks is referred to as **deep learning**. The number of neurons in a layer is referred to as the **width** of that layer.

The hidden layers are usually composed of perceptrons (sigmoidal units) or ReLU units and the output layer is usually

- A linear neuron layer for regression
- A softmax layer for classification

Two-layer FFN



Input $\mathbf{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$

Hidden-layer output $\mathbf{h} = (h_1 \quad h_2 \quad \cdots \quad h_M)^T$

Output $\mathbf{y} = (y_1 \quad y_2 \quad \cdots \quad y_K)^T$

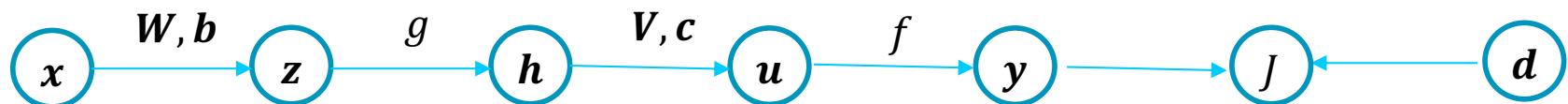
W, b – weight and bias of the hidden layer

V, c – weight and bias of the output layer

M is the number of hidden layer neurons

Forward propagation of activations: single input pattern

Consider an input pattern (x, d) to 2-layer FFN:



Synaptic input \mathbf{z} to hidden layer:

$$\mathbf{z} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$$

Output \mathbf{h} of hidden layer:

$$\mathbf{h} = g(\mathbf{z})$$

g is hidden layer activation function.

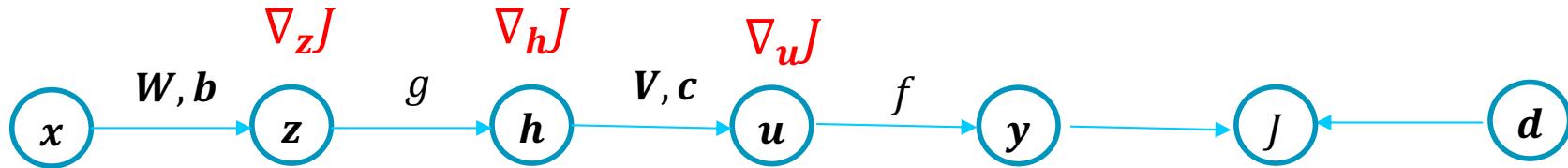
Synaptic input \mathbf{u} to output layer:

$$\mathbf{u} = \mathbf{V}^T \mathbf{h} + \mathbf{c}$$

Output \mathbf{y} of output layer:

$$\mathbf{y} = f(\mathbf{u})$$

Backpropagation of gradients



Since the targets appear at the output, the error gradient at the output layer is $\nabla_u J$ is known. Therefore, output weights and bias, V, c , can be learnt.

To learn hidden layer weights and biases, the gradients at the output layer are to be backpropagated to hidden layers.

Derivatives

$$\mathbf{u} = \mathbf{V}^T \mathbf{h} + \mathbf{c}$$

Consider synaptic input to u_k to the k th neuron at the output layer. Let weight vector $\mathbf{v}_k = (v_{k1} \quad v_{k2} \quad \cdots \quad v_{kM})^T$ and bias c_k .

The synaptic input u_k due to \mathbf{h} is given by

$$u_k = \mathbf{v}_k^T \mathbf{h} + c_k = v_{k1} h_1 + v_{k2} h_2 + \cdots + v_{kM} h_M + c_k$$

$$\frac{\partial u_k}{\partial h_j} = v_{kj} \quad \text{for all } j = 1, 2, \dots, K$$

Therefore

$$\frac{\partial \mathbf{u}}{\partial \mathbf{h}} = \begin{pmatrix} \frac{\partial u_1}{\partial h_1} & \frac{\partial u_1}{\partial h_2} & \cdots & \frac{\partial u_1}{\partial h_M} \\ \frac{\partial u_2}{\partial h_1} & \frac{\partial u_2}{\partial h_2} & \cdots & \frac{\partial u_2}{\partial h_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial u_K}{\partial h_1} & \frac{\partial u_K}{\partial h_2} & \cdots & \frac{\partial u_K}{\partial h_K} \end{pmatrix} = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1M} \\ v_{21} & v_{22} & \cdots & v_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ v_{K1} & v_{K2} & \cdots & v_{KM} \end{pmatrix} = \mathbf{V}^T$$

That is,

$$\frac{\partial \mathbf{u}}{\partial \mathbf{h}} = \mathbf{V}^T \quad (\text{A})$$

Derivatives

$$\mathbf{y} = f(\mathbf{u})$$

Considering k th neuron:

$$y_k = f(u_k)$$

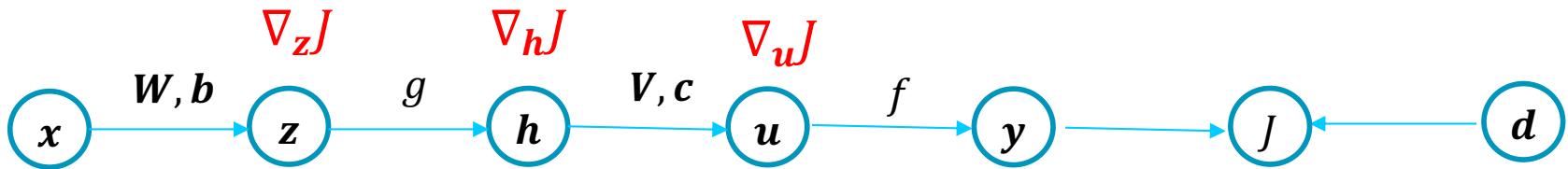
$$\frac{\partial \mathbf{y}}{\partial \mathbf{u}} = \begin{pmatrix} \frac{\partial y_1}{\partial u_1} & \frac{\partial y_1}{\partial u_2} & \dots & \frac{\partial y_1}{\partial u_K} \\ \frac{\partial y_2}{\partial u_1} & \frac{\partial y_2}{\partial u_2} & \dots & \frac{\partial y_2}{\partial u_K} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial y_K}{\partial u_1} & \frac{\partial y_K}{\partial u_2} & \dots & \frac{\partial y_K}{\partial u_K} \end{pmatrix} = \begin{pmatrix} f'(u_1) & 0 & \dots & 0 \\ 0 & f'(u_2) & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & f'(u_K) \end{pmatrix} = \text{diag}(f'(\mathbf{u}))$$

That is,

$$\frac{\partial \mathbf{y}}{\partial \mathbf{u}} = \text{diag}(f'(\mathbf{u})) \quad (\text{B})$$

where $\text{diag}(f'(\mathbf{u}))$ is a diagonal matrix composed of derivatives corresponding to individual components of \mathbf{u} in the diagonal.

Back-propagation of gradients: single pattern



Considering output layer,

$$\nabla_u J = \begin{cases} -(d - y) & \text{for a linear layer} \\ -(1(k = d) - f(u)) & \text{for a softmax layer} \end{cases}$$

From chain rule of differentiation,

$$\nabla_h J = \left(\frac{\partial u}{\partial h} \right)^T \nabla_u J = V \nabla_u J \quad \text{From (A)}$$

$$\nabla_z J = \left(\frac{\partial h}{\partial z} \right)^T \nabla_h J = \text{diag}(g'(\mathbf{z})) V \nabla_u J = V \nabla_u J \cdot g'(\mathbf{z})$$

(C), from (B)

Proof

For a vector \mathbf{x} :

$$diag(f'(\mathbf{u}))\mathbf{x} = \begin{pmatrix} f'(u_1) & 0 & \cdots & 0 \\ 0 & f'(u_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(u_K) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix} = \begin{pmatrix} f'(u_1)x_1 \\ f'(u_2)x_2 \\ \vdots \\ f'(u_K)x_K \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix} \cdot \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_K) \end{pmatrix}$$

That is:

$$diag(f'(\mathbf{u}))\mathbf{x} = \mathbf{x} \cdot f'(\mathbf{u}) = f'(\mathbf{u}) \cdot \mathbf{x}$$

Back-propagation of gradients: single input pattern

From (C);

$$\nabla_{\mathbf{z}} J = \mathbf{V} \nabla_{\mathbf{u}} J \cdot g'(\mathbf{z})$$

That is, the gradients at output layer are multiplied by \mathbf{V} and back-propagated to hidden layer.

Note that hidden-layer activations are multiplied by \mathbf{V}^T (Note $\mathbf{u} = \mathbf{V}^T \mathbf{h} + \mathbf{c}$) in forward propagation and in back-propagation, the gradients are multiplied by \mathbf{V} .

Flow of gradients propagated in backward direction, hence named **back-propagation** (backprop) algorithm.

SGD of two-layer FFN

Output layer:

$$\nabla_{\mathbf{u}} J = \begin{cases} -(\mathbf{d} - \mathbf{y}) & \text{for linear layer} \\ -(1(\mathbf{k} = d) - f(\mathbf{u})) & \text{for softmax layer} \end{cases}$$

Hidden layer:

$$\nabla_{\mathbf{z}} J = \mathbf{V} \nabla_{\mathbf{u}} J \cdot g'(\mathbf{z})$$

SGD for a two-layer FFN

Given a training dataset $\{(x, d)\}$

Set learning parameter α

Initialize W, b, V, c

Repeat until convergence:

For every pattern (x, d) :

$$z = W^T x + b$$

$$h = g(z)$$

$$u = V^T h + c$$

$$y = f(u)$$

Forward propagation

$$\nabla_w J = \begin{cases} -(d - y) \\ -(1(k=d) - f(u)) \end{cases}$$

$$\nabla_z J = V \nabla_w J \cdot g'(z)$$

Backward propagation

$$V \leftarrow V - \alpha h (\nabla_w J)^T$$

$$c \leftarrow c - \alpha \nabla_w J$$

$$W \leftarrow W - \alpha x (\nabla_z J)^T$$

$$b \leftarrow b - \alpha \nabla_z J$$

Forward propagation of activations: batch of inputs

Computational graph of 2-layer FFN for a batch of patterns (X, D) :



Synaptic input Z to hidden layer:

$$Z = XW + B$$

Output H of the hidden layer:

$$H = g(Z)$$

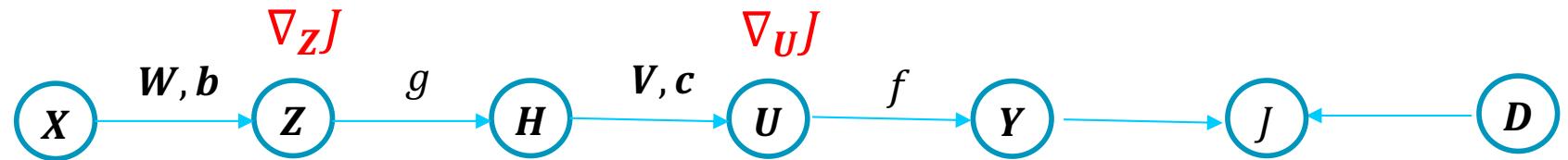
Synaptic input U to output layer:

$$U = HV + C$$

Output Y of the output layer:

$$Y = f(U)$$

Back-propagation of gradients: batch of patterns



$$\nabla_U J = \begin{cases} -(D - Y) & \text{for linear layer} \\ -(K - f(U)) & \text{for softmax layer} \end{cases}$$

$$\nabla_Z J = \begin{pmatrix} (\nabla_{z_1} J)^T \\ (\nabla_{z_2} J)^T \\ \vdots \\ (\nabla_{z_P} J)^T \end{pmatrix} = \begin{pmatrix} \left(V \nabla_{u_1} J \cdot g'(\mathbf{z}_1) \right)^T \\ \left(V \nabla_{u_2} J \cdot g'(\mathbf{z}_2) \right)^T \\ \vdots \\ \left(V \nabla_{u_P} J \cdot g'(\mathbf{z}_P) \right)^T \end{pmatrix}$$

Substituting
from (C)

Back-propagation of gradients: batch of patterns

$$\nabla_{\mathbf{Z}} J = \begin{pmatrix} (\nabla_{\mathbf{u}_1} J)^T V^T \cdot (g'(\mathbf{z}_1))^T \\ (\nabla_{\mathbf{u}_2} J)^T V^T \cdot (g'(\mathbf{z}_2))^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J)^T V^T \cdot (g'(\mathbf{z}_P))^T \end{pmatrix} \quad (XY)^T = Y^T X^T$$

$$= \begin{pmatrix} (\nabla_{\mathbf{u}_1} J)^T \\ (\nabla_{\mathbf{u}_2} J)^T \\ \vdots \\ (\nabla_{\mathbf{u}_P} J)^T \end{pmatrix} V^T \cdot \begin{pmatrix} (g'(\mathbf{z}_1))^T \\ (g'(\mathbf{z}_2))^T \\ \vdots \\ (g'(\mathbf{z}_P))^T \end{pmatrix}$$

$$= (\nabla_{\mathbf{U}} J) V^T \cdot g'(\mathbf{Z})$$

$$\nabla_{\mathbf{Z}} J = (\nabla_{\mathbf{U}} J) V^T \cdot g'(\mathbf{Z})$$

GD of two-layer FFN

Output layer:

$$\nabla_{\mathbf{U}} J = \begin{cases} -(\mathbf{D} - \mathbf{Y}) & \text{for linear layer} \\ -(\mathbf{K} - f(\mathbf{U})) & \text{for softmax layer} \end{cases}$$

Hidden layer:

$$\nabla_{\mathbf{Z}} J = (\nabla_{\mathbf{U}} J) \mathbf{V}^T \cdot g'(\mathbf{Z}) \quad (\text{D})$$

GD for a two-layer FFN

Given a training dataset (\mathbf{X}, \mathbf{D})

Set learning parameter α

Initialize $\mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c}$

Repeat until convergence:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B}$$

$$\mathbf{H} = g(\mathbf{Z})$$

$$\mathbf{U} = \mathbf{H}\mathbf{V} + \mathbf{C}$$

$$\mathbf{Y} = f(\mathbf{U})$$

Forward propagation

$$\nabla_{\mathbf{U}} J = \begin{cases} -(\mathbf{D} - \mathbf{Y}) \\ -(\mathbf{K} - f(\mathbf{U})) \end{cases}$$

$$\nabla_{\mathbf{Z}} J = (\nabla_{\mathbf{U}} J) \mathbf{V}^T \cdot g'(\mathbf{Z})$$

Backward propagation

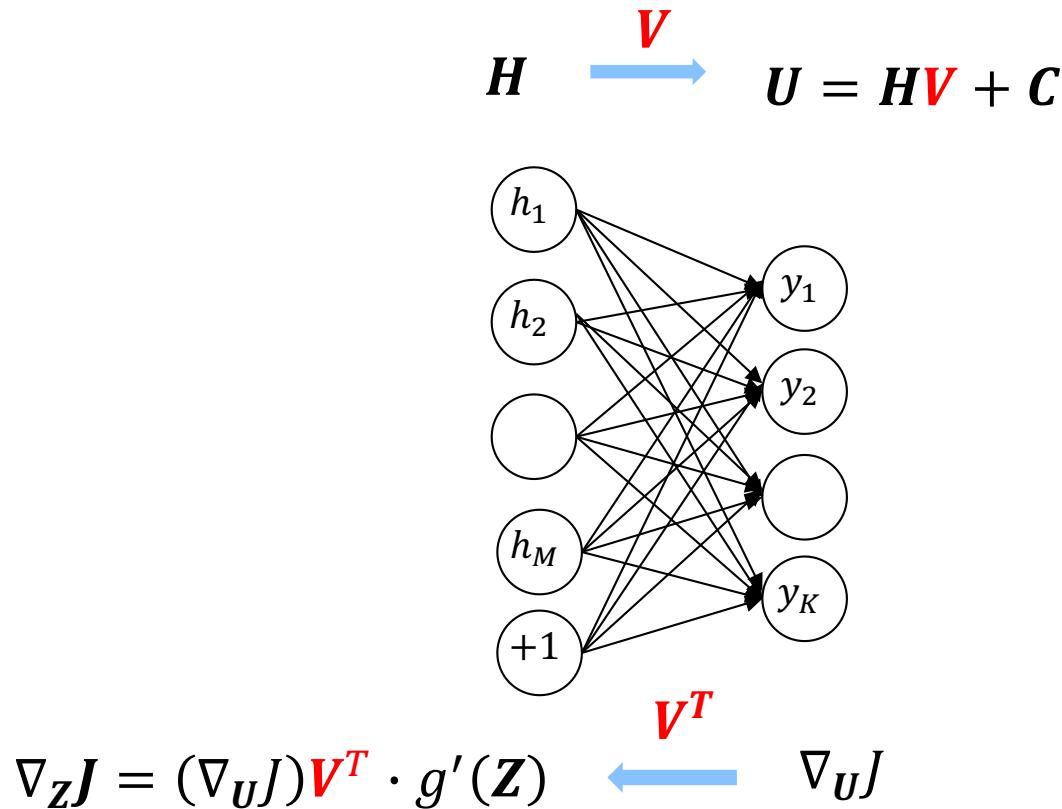
$$\mathbf{V} \leftarrow \mathbf{V} - \alpha \mathbf{H}^T \nabla_{\mathbf{U}} J$$

$$\mathbf{c} \leftarrow \mathbf{c} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P$$

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{Z}} J$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha (\nabla_{\mathbf{Z}} J)^T \mathbf{1}_P$$

Back-propagation



The error gradient can be seen as propagating from the output layer to the hidden layer and so learning in feedforward networks is known as the *back-propagation* algorithm

Learning in two-layer FFN

GD	SGD
(X, D)	(x, d)
$Z = XW + B$	$z = W^T x + b$
$H = g(Z)$	$h = g(z)$
$U = HV + C$	$u = V^T h + c$
$Y = f(U)$	$y = f(u)$
$\nabla_U J = \begin{cases} -(D - Y) \\ -(K - f(U)) \end{cases}$	$\nabla_w J = \begin{cases} -(d - y) \\ (1(k = d) - f(u)) \end{cases}$
$\nabla_Z J = (\nabla_U J)V^T \cdot g'(Z)$	$\nabla_z J = V \nabla_w J \cdot g'(z)$
$W \leftarrow W - \alpha X^T \nabla_Z J$	$W \leftarrow W - \alpha x (\nabla_z J)^T$
$b \leftarrow b - \alpha (\nabla_Z J)^T \mathbf{1}_P$	$b \leftarrow b - \alpha \nabla_z J$
$V \leftarrow V - \alpha H^T \nabla_U J$	$V \leftarrow V - \alpha h (\nabla_w J)^T$
$C \leftarrow C - \alpha (\nabla_U J)^T \mathbf{1}_P$	$c \leftarrow c - \alpha \nabla_w J$

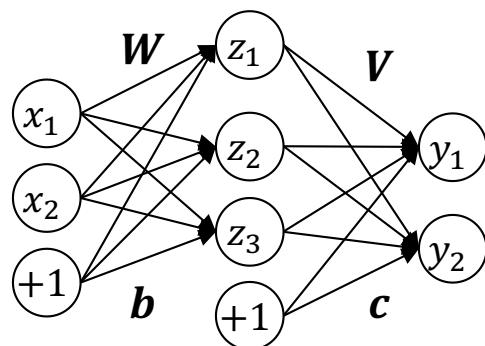
Example 2: Two-layer FFN

Design a two-layer FFN, using gradient descent to perform the following mapping. Use a learning factor = 0.05 and three perceptrons in the hidden-layer.

Inputs $x = (x_1, x_2)$	Targets $d = (d_1, d_2)$
(0.77, 0.02)	(0.44, -0.42)
(0.63, 0.75)	(0.84, 0.43)
(0.50, 0.22)	(0.09, -0.72)
(0.20, 0.76)	(-0.25, 0.35)
(0.17, 0.09)	(-0.12, -0.13)
(0.69, 0.95)	(0.24, 0.03)
(0.00, 0.51)	(0.30, 0.20)
(0.81, 0.61)	(0.61, 0.04)

Example 2

$$X = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix}$$



Output layer is a linear neuron layer

Hidden layer is a sigmoidal layer

Initialized (weights using a uniform distribution):

$$W = \begin{pmatrix} -3.97 & 1.10 & 0.42 \\ 2.79 & -2.64 & 3.13 \end{pmatrix}, b = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}, V = \begin{pmatrix} 3.58 & -1.58 \\ -3.58 & -1.75 \\ -3.38 & 2.88 \end{pmatrix}, c = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

Example 2

Epoch 1:

$$\mathbf{Z} = \mathbf{XW} + \mathbf{B} = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \begin{pmatrix} -3.97 & 1.10 & 0.42 \\ 2.79 & -2.64 & 3.13 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -3.00 & 0.8 & 0.39 \\ -0.42 & -1.27 & 2.61 \\ -1.35 & -0.04 & 0.91 \\ 1.34 & -1.79 & 2.46 \\ -0.42 & -0.05 & 0.35 \\ -0.05 & -1.76 & 3.27 \\ 1.42 & -1.34 & 1.60 \\ -1.51 & -0.72 & 2.25 \end{pmatrix}$$

$$\mathbf{H} = g(\mathbf{Z}) = \frac{1}{1 + e^{-\mathbf{z}}} = \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.40 & 0.49 & 0.59 \\ 0.49 & 0.15 & 0.96 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix}$$

Example 2

$$Y = HV + C = \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.40 & 0.49 & 0.59 \\ 0.49 & 0.15 & 0.96 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix} \begin{pmatrix} 3.58 & -1.18 \\ -3.58 & -1.78 \\ -3.38 & 2.88 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -4.32 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix}$$

$$\nabla_u J = -(D - Y) = - \left(\begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix} - \begin{pmatrix} -4.32 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix} \right) = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix}$$

Example 2

$$g'(\mathbf{Z}) = \mathbf{H} \cdot (\mathbf{1} - \mathbf{H}) = \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

$$\nabla_{\mathbf{Z}} J = (\nabla_{\mathbf{U}} J) \mathbf{V}^T \cdot g'(\mathbf{Z})$$

$$= \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix} \begin{pmatrix} 3.58 & -1.58 \\ -3.58 & -1.75 \\ -3.38 & 2.88 \end{pmatrix}^T \cdot \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

Example 2

Output layer:

$$\nabla_{\mathbf{V}} J = \mathbf{H}^T \quad \nabla_{\mathbf{U}} J = \begin{pmatrix} -6.23 & 3.22 \\ -8.81 & 2.85 \\ -17.07 & 7.40 \end{pmatrix}$$

$$\nabla_{\mathbf{c}} J = (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P = \begin{pmatrix} -21.83 \\ 8.81 \end{pmatrix}$$

Hidden layer:

$$\nabla_{\mathbf{W}} J = \mathbf{X}^T \quad \nabla_{\mathbf{z}} J = \begin{pmatrix} -8.43 & 7.81 & 7.79 \\ -8.21 & 4.56 & 3.76 \end{pmatrix}$$

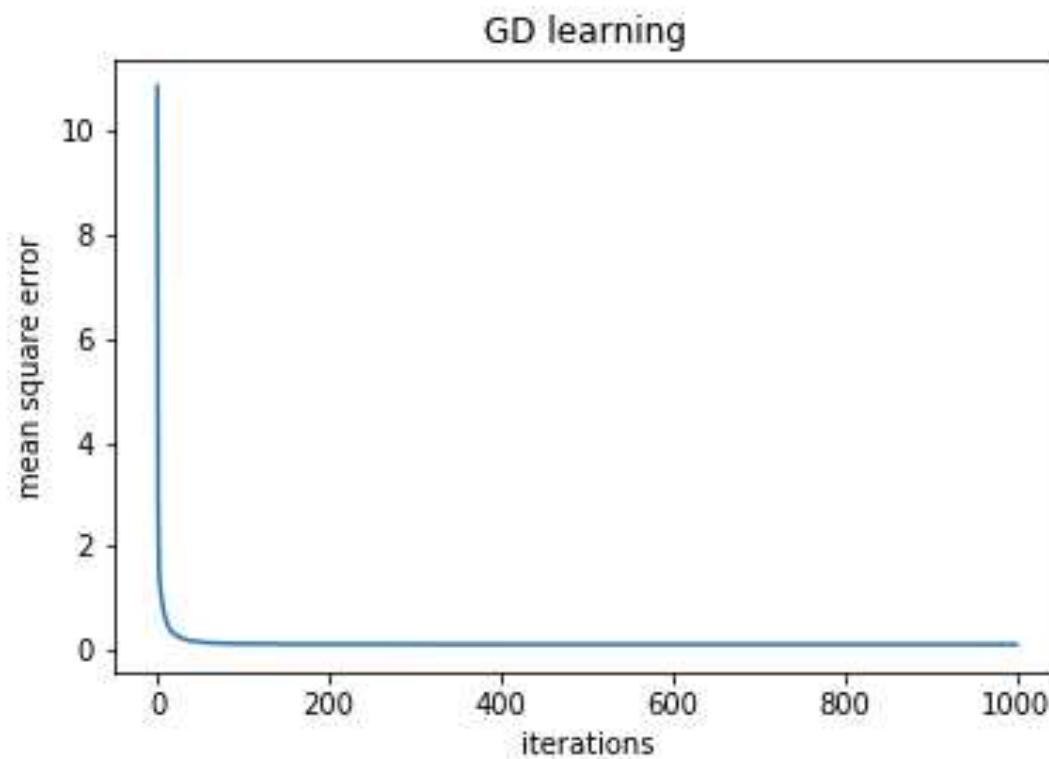
$$\nabla_{\mathbf{b}} J = (\nabla_{\mathbf{z}} J)^T \mathbf{1}_P = \begin{pmatrix} -15.22 \\ 13.11 \\ 13.92 \end{pmatrix}$$

Example 2

$$\mathbf{V} \leftarrow \mathbf{V} - \alpha \nabla_{\mathbf{V}} J = \begin{pmatrix} 3.89 & -1.74 \\ -3.14 & -1.89 \\ -2.53 & 2.51 \end{pmatrix}$$
$$\mathbf{c} \leftarrow \mathbf{c} - \alpha \nabla_{\mathbf{c}} J = \begin{pmatrix} 1.09 \\ -0.44 \end{pmatrix}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J = \begin{pmatrix} -3.55 & 0.72 & 0.03 \\ 3.21 & -2.87 & 2.94 \end{pmatrix}$$
$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J = \begin{pmatrix} 0.76 \\ -0.66 \\ -0.70 \end{pmatrix}$$

Example 2



Example 2

After 1,000 epochs,

$$m. \ s. \ e = 0.107$$

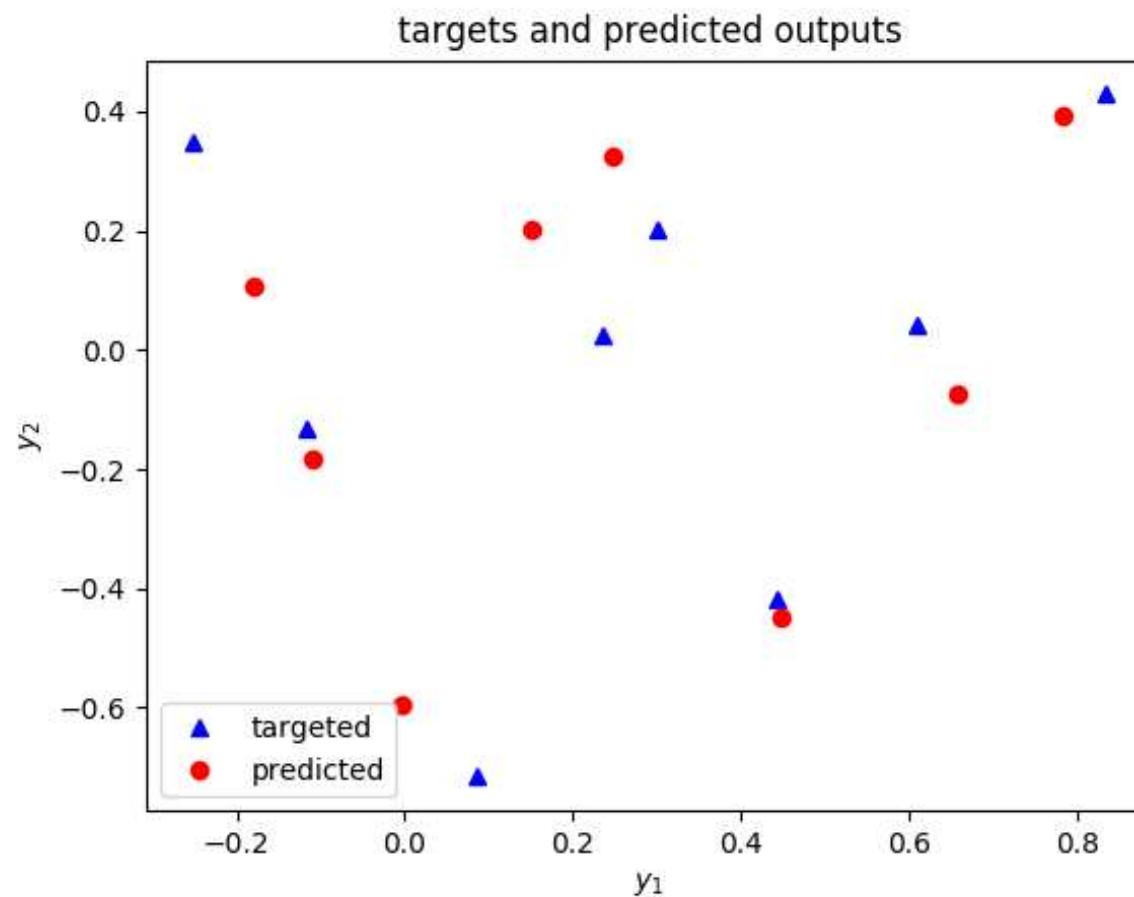
$$\mathbf{W} = \begin{pmatrix} -2.04 & -0.52 & -1.88 \\ 3.55 & -2.6 & 2.43 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 0.22 \\ -0.93 \\ 0.15 \end{pmatrix}$$

$$\mathbf{V} = \begin{pmatrix} 2.14 & -1.14 \\ -2.93 & -1.78 \\ 3.59 & 2.28 \end{pmatrix}$$

$$\mathbf{c} = \begin{pmatrix} 1.25 \\ -0.43 \end{pmatrix}$$

Example 2



After 20,000 iterations

Preprocessing of inputs

If inputs have similar variations, better approximation of inputs or prediction of outputs is achieved. Mainly, there are two approaches to normalization of inputs.

Suppose i th input $x_i \in [x_{i,min}, x_{i,max}]$ and has a mean μ_i and a standard deviation σ_i .

If \tilde{x}_i denotes the normalized input.

1. **Scaling** the inputs such that $\tilde{x}_i \in [0, 1]$:

$$\tilde{x}_i = \frac{x_i - x_{i,min}}{x_{i,max} - x_{i,min}}$$

2. **Normalizing** the input to have standard normal distributions $\tilde{x}_i \sim N(0, 1)$:

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

Preprocessing of Outputs

Linear activation function:

The convergence is usually improved if each output is **normalized** to have zero mean and unit standard deviation: $\tilde{y}_k \sim N(0,1)$

$$\tilde{y}_k = \frac{y_k - \mu_k}{\sigma_k}$$

Sigmoid activation function:

Since sigmoidal activation range from 0 to 1.0, you can **scale** $\tilde{y}_k \in [0,1]$:

$$\begin{aligned}\tilde{y}_k &= \frac{y_k - y_{k,min}}{y_{k,max} - y_{k,min}} \\ &= \frac{1}{y_{k,max} - y_{k,min}} y_k - \frac{y_{k,min}}{y_{k,max} - y_{k,min}}\end{aligned}$$

California housing dataset

<https://developers.google.com/machine-learning/crash-course/california-housing-data-description>

9 variables

The problem is to predict the housing prices using the other 8 variables.

20540 samples

Train: 14448 samples

Test: 6192 samples

longitude	A measure of how far west a house is; a more negative value is farther west
latitude	A measure of how far north a house is; a higher value is farther north
housingMedianAge	Median age of a house within a block; a lower number is a newer building
totalRooms	Total number of rooms within a block
totalBedrooms	Total number of bedrooms within a block
population	Total number of people residing within a block
households	Total number of households, a group of people residing within a home unit, for a block
medianIncome	Median income for households within a block of houses (measured in tens of thousands of US Dollars)
medianHouseValue	Median house value for households within a block (measured in US Dollars)

Example 3: Two-layer FFN predicting housing prices in California

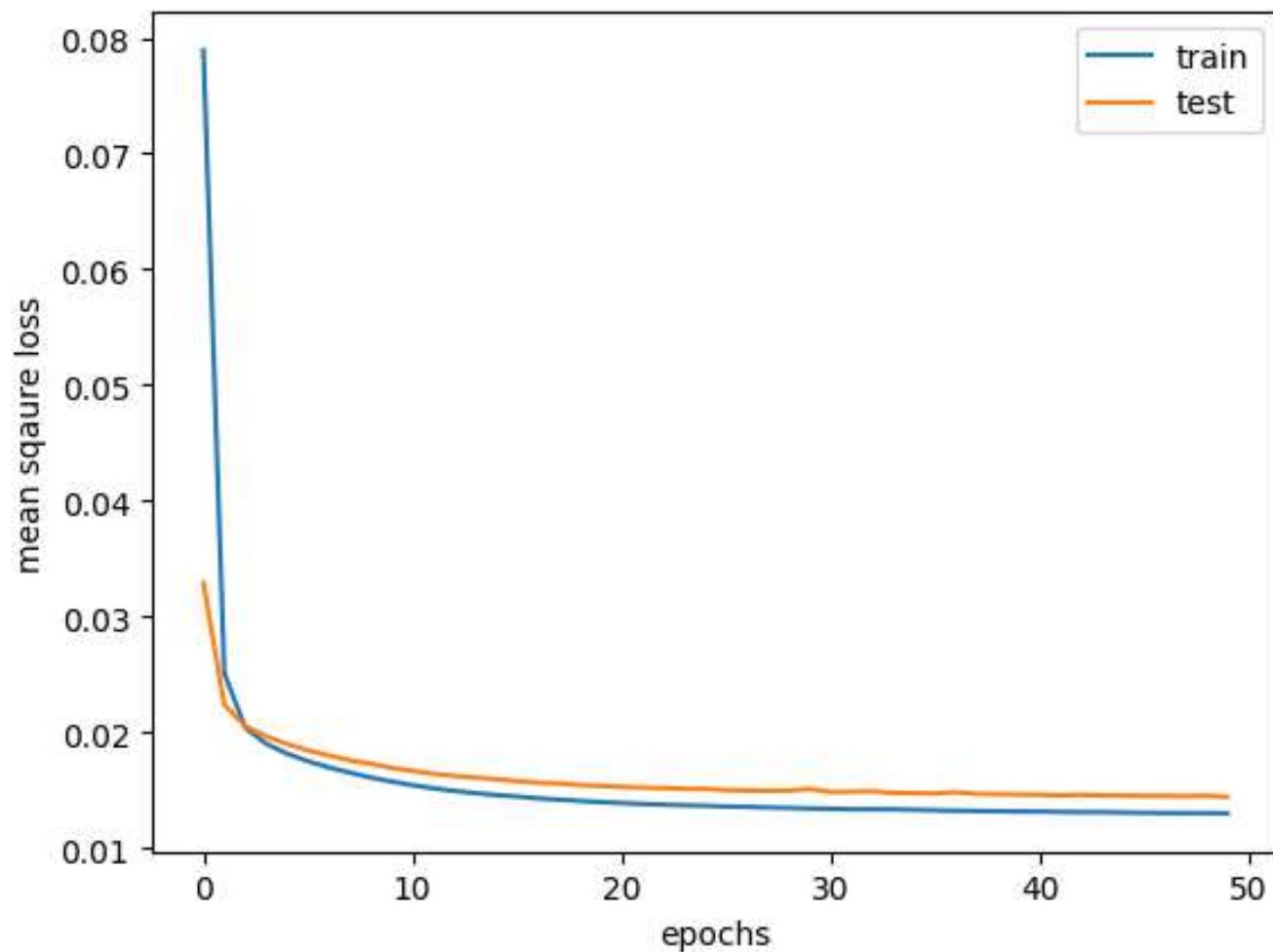
Thirteen input variables, One output variable

We use FFN with one hidden layer with 10 neuron
Network size: [8, 10, 1]

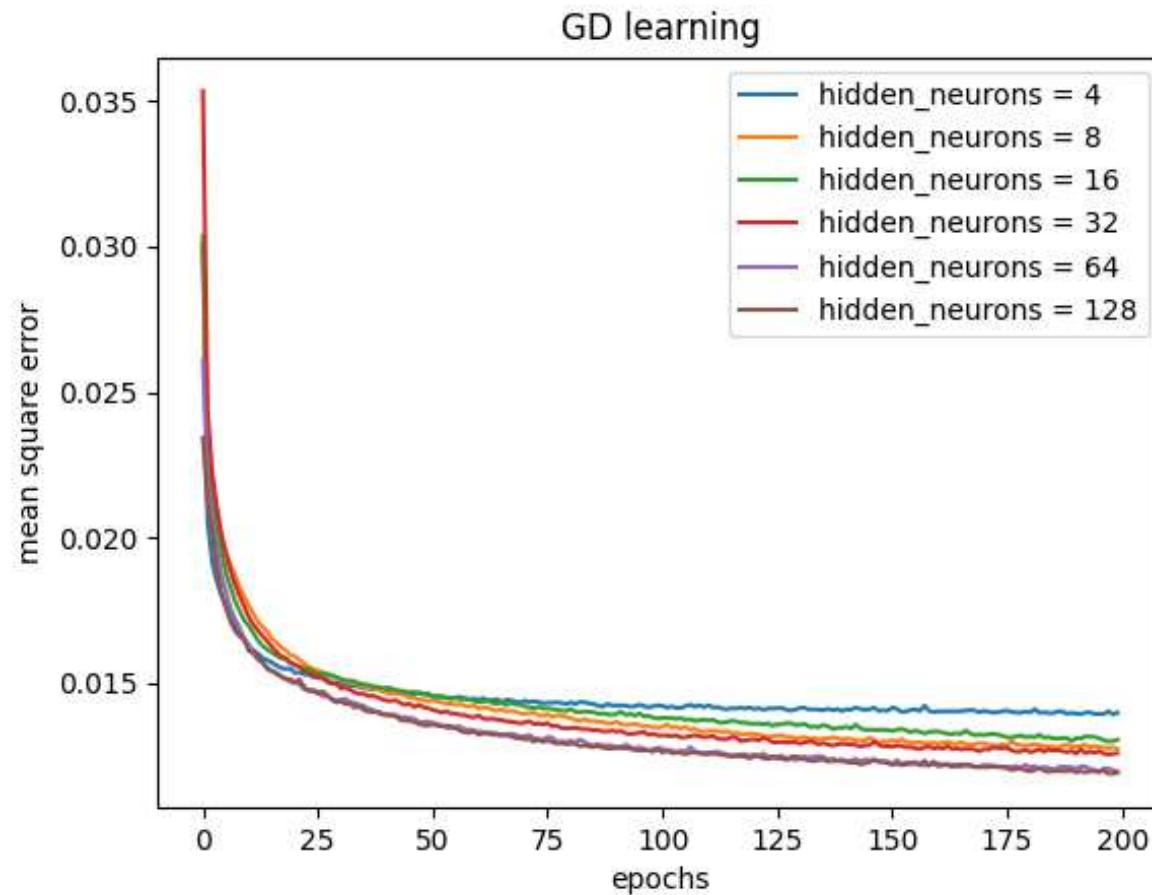
```
class FFN(nn.Module):
    def __init__(self, no_features, no_hidden, no_labels):
        super().__init__()
        self.relu_stack = nn.Sequential(
            nn.Linear(no_features, no_hidden),
            nn.ReLU(),
            nn.Linear(no_hidden, no_labels),
        )

    def forward(self, x):
        logits = self.relu_stack(x)
        return logits
```

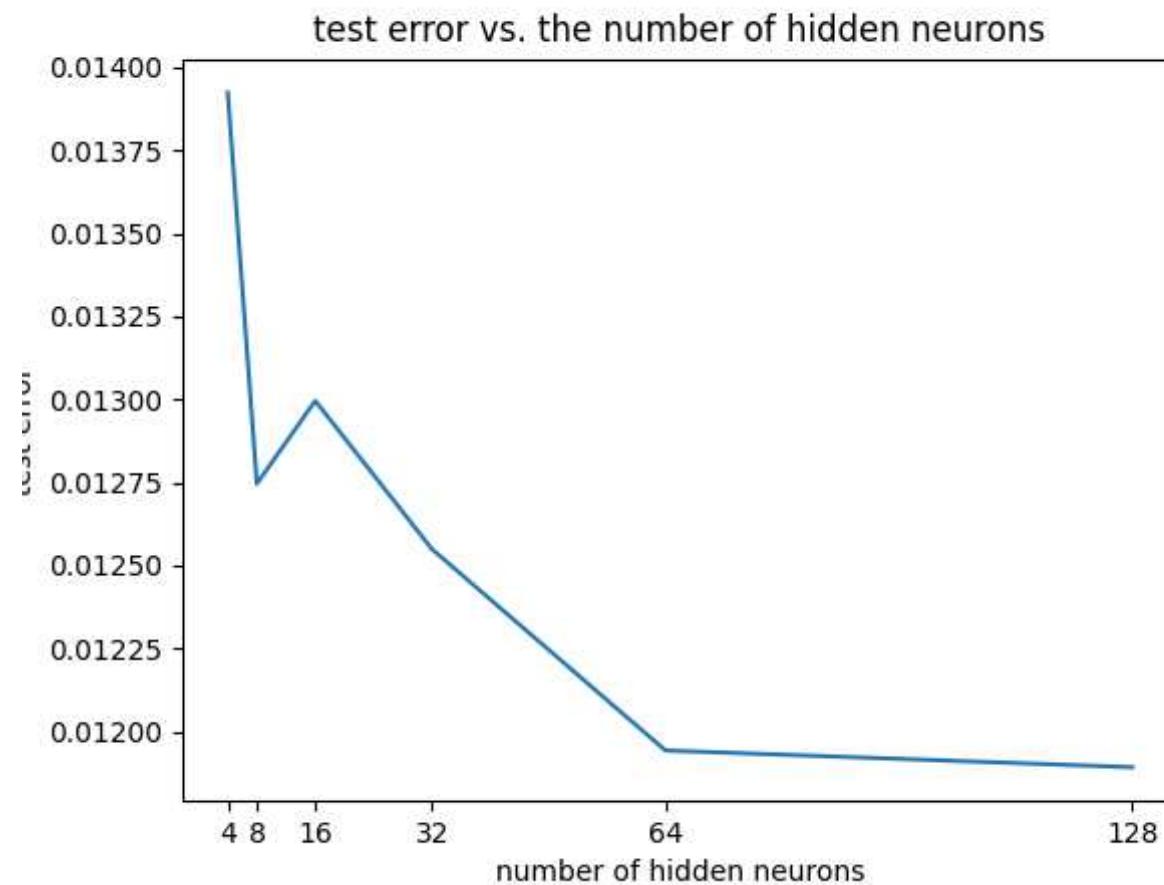
Example 3a



Example 3b: no of hidden neurons



Example 3b: width of the hidden layer



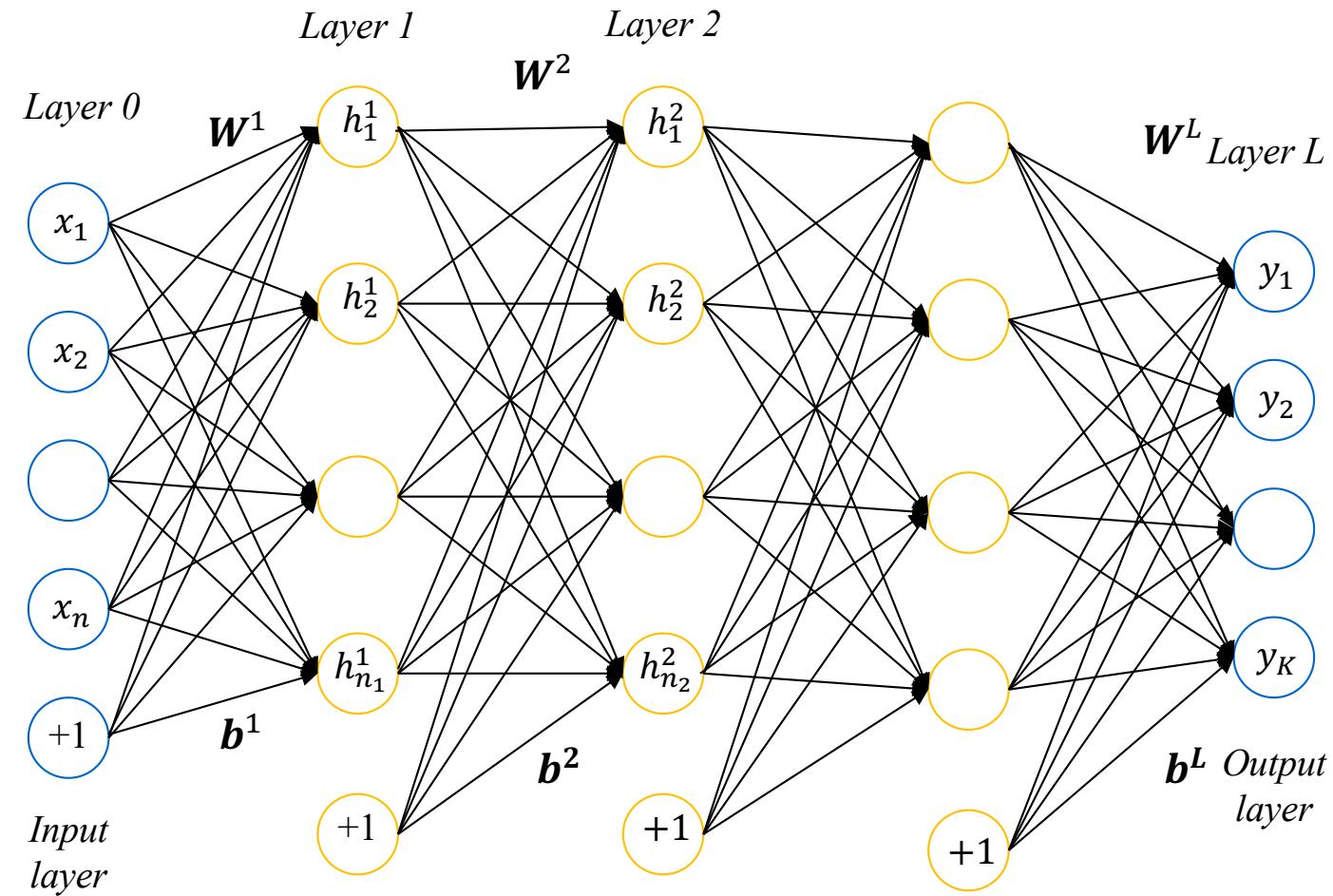
Optimum number of hidden neurons 64

Width of hidden Layers

The number of parameters of the network increases with the **width** of layers. Therefore, the network attempts to remember the training patterns with increasing number of parameters. In other words, the network aims at minimizing the training error at the expense of its generalization ability on unseen data.

As the number of hidden units increases, the test error decreases initially but tends to increase at some point. The optimal number of hidden units is often determined empirically (that is, by trial and error).

Deep neural networks (DNN)



Depth = L

DNN notations

Input layer $l = 0$:

Width = n

Input \mathbf{x}, \mathbf{X}

Hidden layers $l = 1, 2, \dots, L - 1$

Width: n_l

Weight matrix \mathbf{W}^l , bias vector \mathbf{b}^l

Synaptic input $\mathbf{u}^l, \mathbf{U}^l$

Activation function f^l

Output $\mathbf{h}^l, \mathbf{H}^l$

Output layer $l = L$

Width: K

Synaptic input $\mathbf{u}^L, \mathbf{U}^L$

Activation function f^L

Output \mathbf{y}, \mathbf{Y}

Desired output \mathbf{d}, \mathbf{D}

Forward propagation of activation in DNN: single pattern

Input (\mathbf{x}, \mathbf{d})

$$\mathbf{u}^1 = \mathbf{W}^{1T} \mathbf{x} + \mathbf{b}^1$$

For layers $l = 1, 2, \dots, L - 1$:

$$\mathbf{h}^l = f^l(\mathbf{u}^l)$$

$$\mathbf{u}^{l+1} = \mathbf{W}^{l+1T} \mathbf{h}^l + \mathbf{b}^{l+1}$$

$$\mathbf{y} = f^L(\mathbf{u}^L)$$

Back-propagation of gradients in DNN: single pattern

if $l = L$:

$$\nabla_{\mathbf{u}^l} J = \begin{cases} -(\mathbf{d} - \mathbf{y}) & \text{for linear layer} \\ -(1(\mathbf{k} = d) - f^L(\mathbf{u}^L)) & \text{for softmax layer} \end{cases}$$

else:

$$\nabla_{\mathbf{u}^l} J = \mathbf{W}^{l+1} (\nabla_{\mathbf{u}^{l+1}} J) \cdot f^l(\mathbf{u}^l) \quad \text{from (C)}$$

$$\nabla_{\mathbf{W}^l} J = \mathbf{h}^{l-1} (\nabla_{\mathbf{u}^l} J)^T$$

$$\nabla_{\mathbf{b}^l} J = \nabla_{\mathbf{u}^l} J$$

Gradients are backpropagated from the output layer to the input layer

Forward propagation of activation in DNN: batch of patterns

Input (\mathbf{X}, \mathbf{D})

$$\mathbf{U}^1 = \mathbf{X}\mathbf{W}^1 + \mathbf{B}^1$$

For layers $l = 1, 2, \dots, L - 1$:

$$\mathbf{H}^l = f^l(\mathbf{U}^l)$$

$$\mathbf{U}^{l+1} = \mathbf{H}^l \mathbf{W}^{l+1} + \mathbf{B}^{l+1}$$

$$\mathbf{Y} = f^L(\mathbf{U}^L)$$

Back-propagation of gradients in DNN: batch of patterns

If $l = L$:

$$\nabla_{\mathbf{U}^l} J = \begin{cases} -(\mathbf{D} - \mathbf{Y}) \\ -(\mathbf{K} - f^L(\mathbf{U}^L)) \end{cases}$$

Else:

$$\nabla_{\mathbf{U}^l} J = (\nabla_{\mathbf{U}^{l+1}} J) \mathbf{W}^{l+1^T} \cdot f^{l'}(\mathbf{U}^l) \quad \text{from (D)}$$

$$\nabla_{\mathbf{W}^l} J = \mathbf{H}^{l-1^T} (\nabla_{\mathbf{U}^l} J)$$

$$\nabla_{\mathbf{b}^l} J = (\nabla_{\mathbf{U}^l} J)^T \mathbf{1}_P$$

Gradients are backpropagated from the output layer to the input layer

Example 4: DNN on California Housing data

California housing data:

<https://developers.google.com/machine-learning/crash-course/california-housing-data-description>

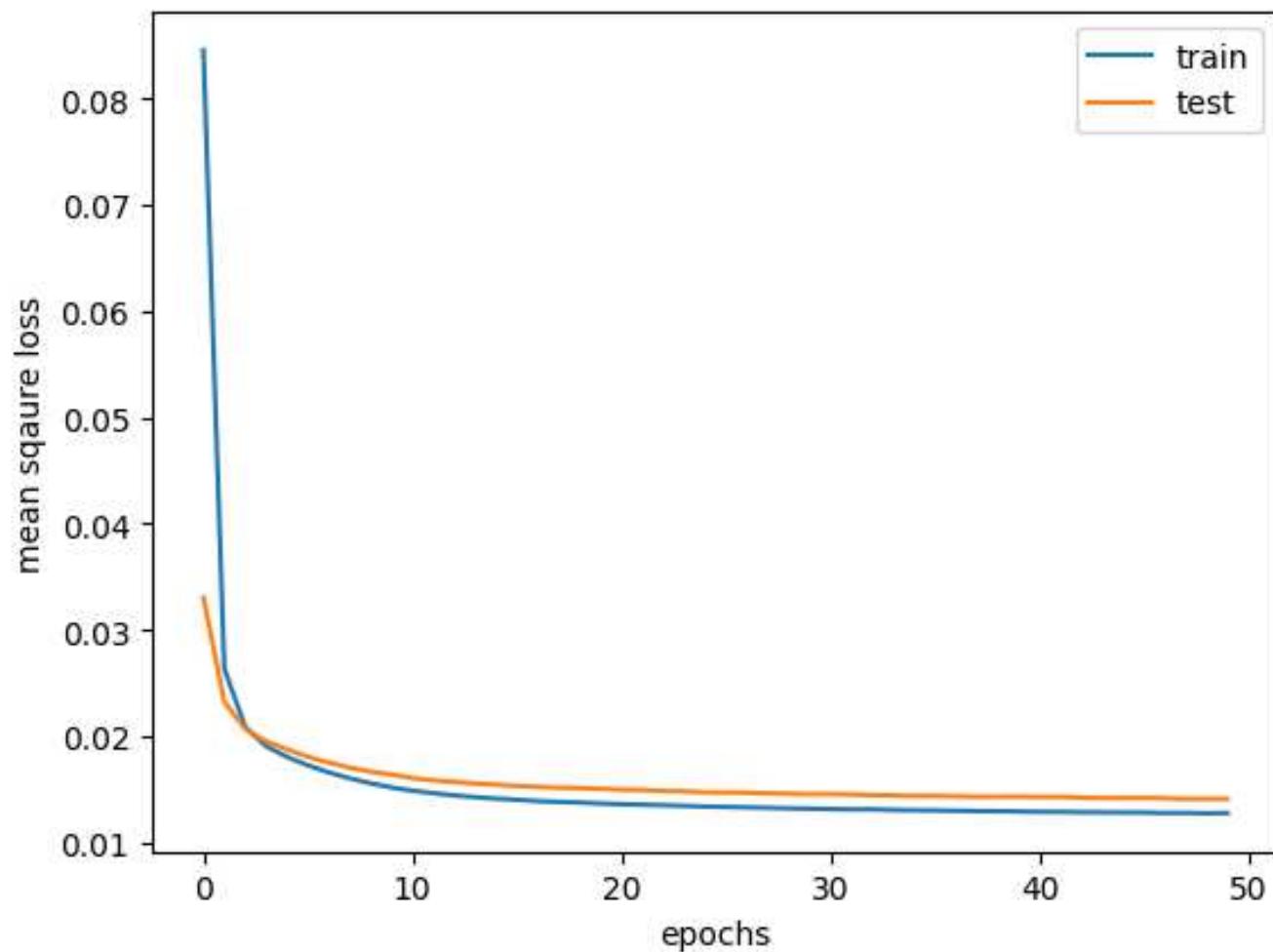
Predicting housing price from other 8 variables

DNN with two hidden layers:

[8, 10, 5, 1]

```
relu_stack = nn.Sequential(  
    nn.Linear(no_features, no_hidden1),  
    nn.ReLU(),  
    nn.Linear(no_hidden1, no_hidden2),  
    nn.ReLU(),  
    nn.Linear(no_hidden2, no_labels),  
)
```

Example 4



Example 4b: Varying the depth of DNN

Architectures:

One hidden layer: [8, 5, 1]

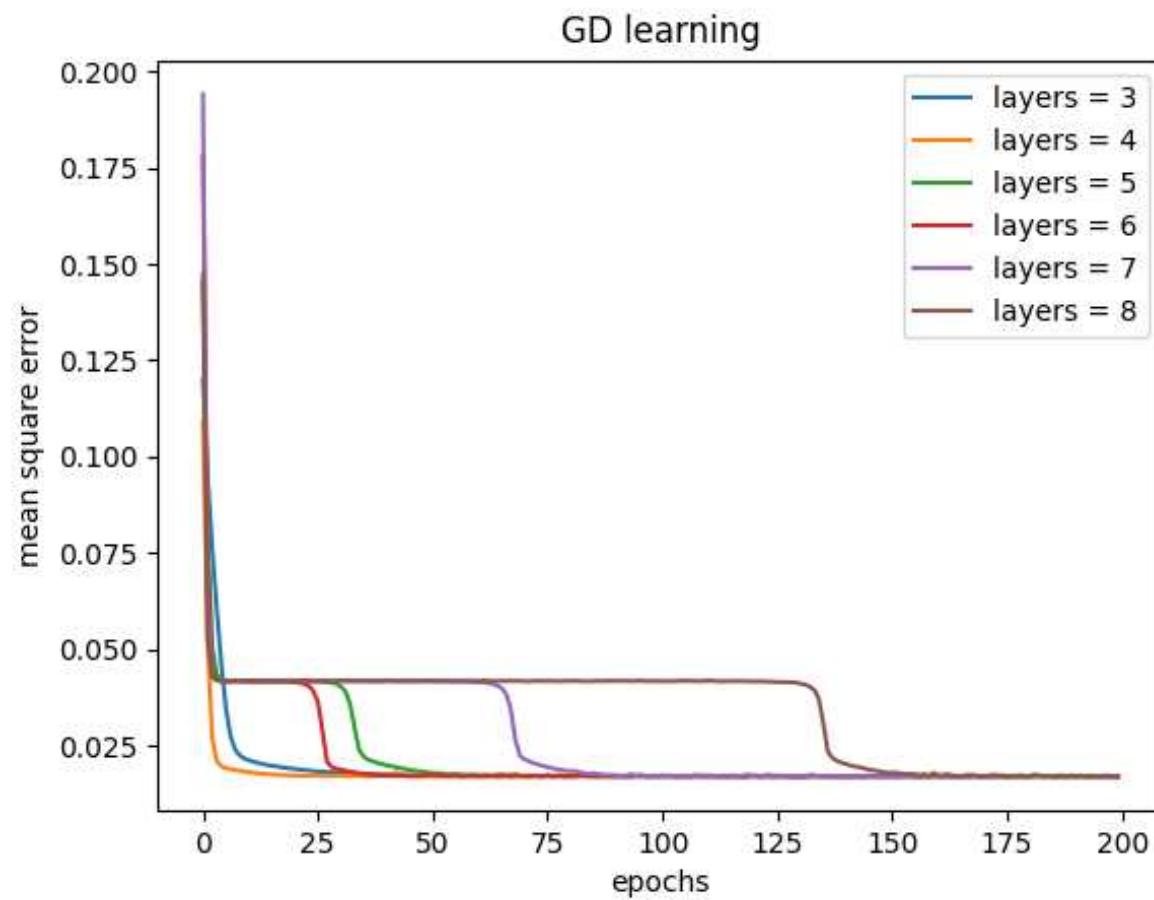
Two hidden layers: [8, 5, 5, 1]

Three hidden layers: [8, 5, 5, 5, 1]

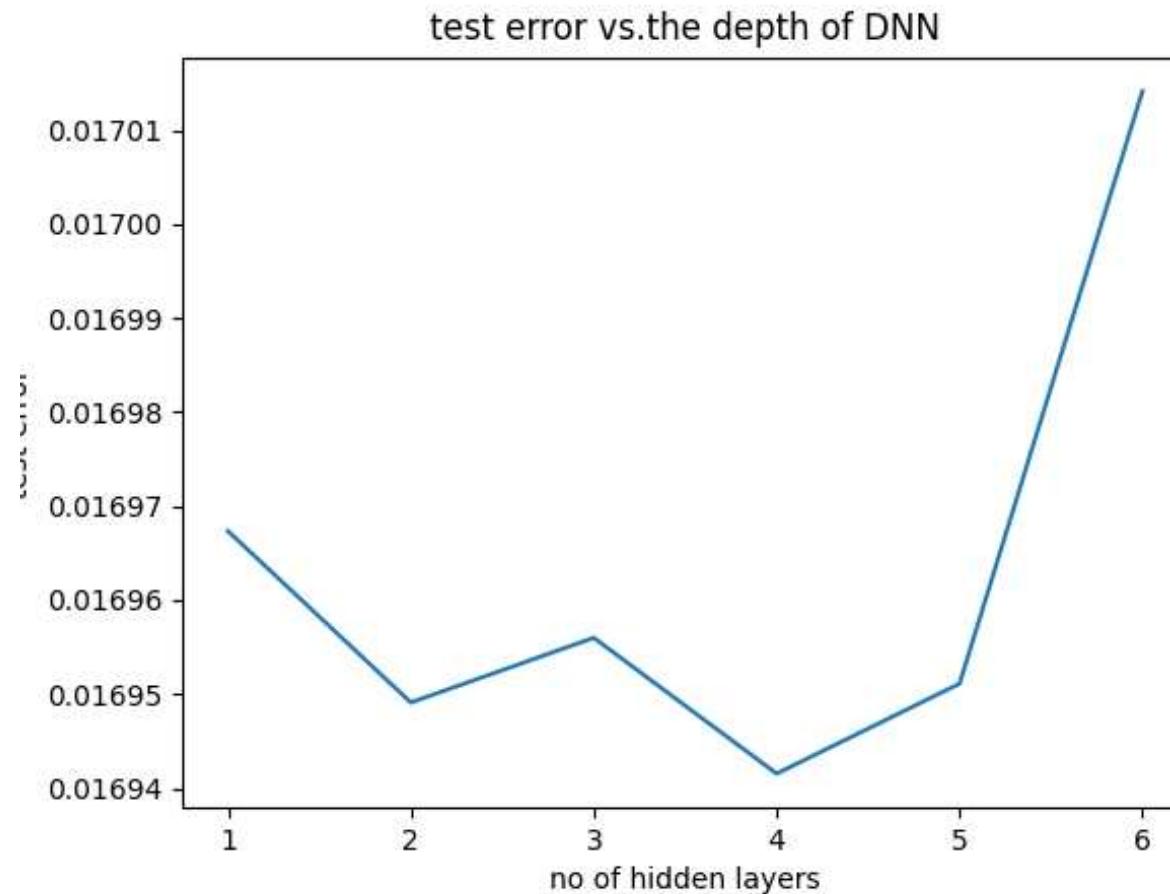
Four hidden layers: [8, 5, 5, 5, 5, 1]

Five hidden layers: [8, 5, 5, 5, 5, 5, 1]

Example 4



Example 4



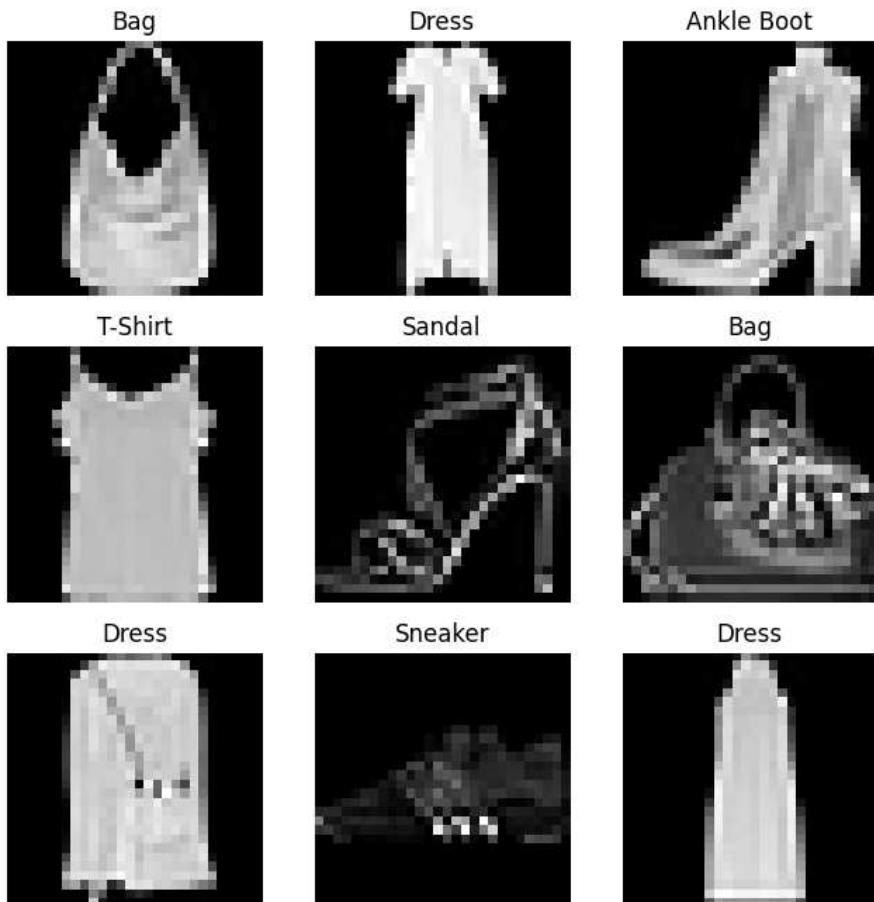
Optimum number of hidden layers = 4

Depth of DNN

The deep networks extract features at different levels of complexity for regression or classification. However, the **depth** or the number of layers that you can have for the networks depend on the number of training patterns available. The deep networks have more parameters (weights and biases) to learn, so need more data to train. Deep networks can learn complex mapping accurately if sufficient training data is available.

The optimal number of layers is determined usually through experiments. The optimal architecture minimizes the error (training, test, and validation).

Fashion MNIST dataset

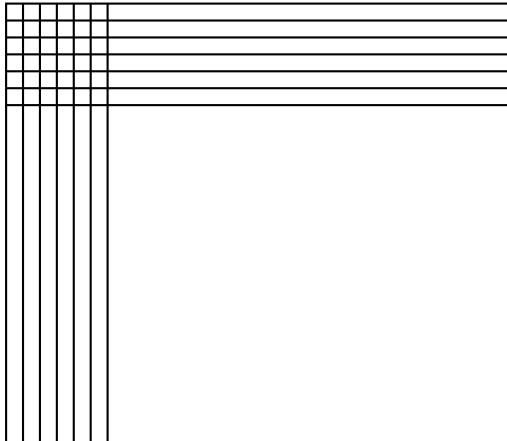


The Fashion-MNIST database of gray level images of fashion items from 10 classes:
<https://github.com/zalandoresearch/fashion-mnist>

Each image is 28x28 size.
Intensities are in the range [0, 255].

Training set: 60,000
Test set: 10,000

MNIST images



An image is divided into rows and columns and defined by its pixels.

Size of the image = rows x columns pixels

Pixels of **grey-level image** are assigned intensity values: For example, integer values between 0 and 255 assigned as intensities (grey-values) for pixels with 0 representing ‘black’ and 255 representing ‘white’.

Color images has three color channels: red, green, and blue. A pixel in a color image is a vector (r, g, b) denoting intensity in red, green, and blue channels.

Example 5: Classification of Fashion-MNIST images

No of inputs $n = 28 \times 28 = 784$ (after flattening)

Inputs were normalized to $[0.0, 1.0]$

Use a 3-layer FFN

- Hidden-layer-1 is a perceptron layer
- Hidden-layer-2 is perceptron layer
- Output-layer is a softmax layer

Input-layer size $n = 784$

Hidden-layer-1 size $n_1 = 512$

Hidden-layer-2 size $n_2 = 512$

Output-layer size $K = 10$

Training:

Batch size = 64

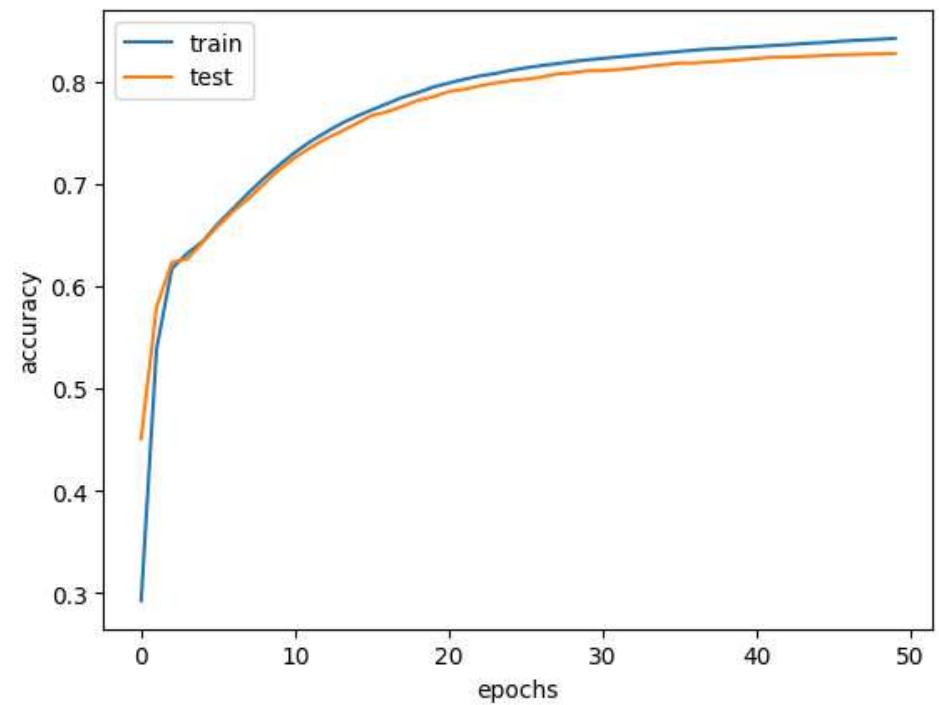
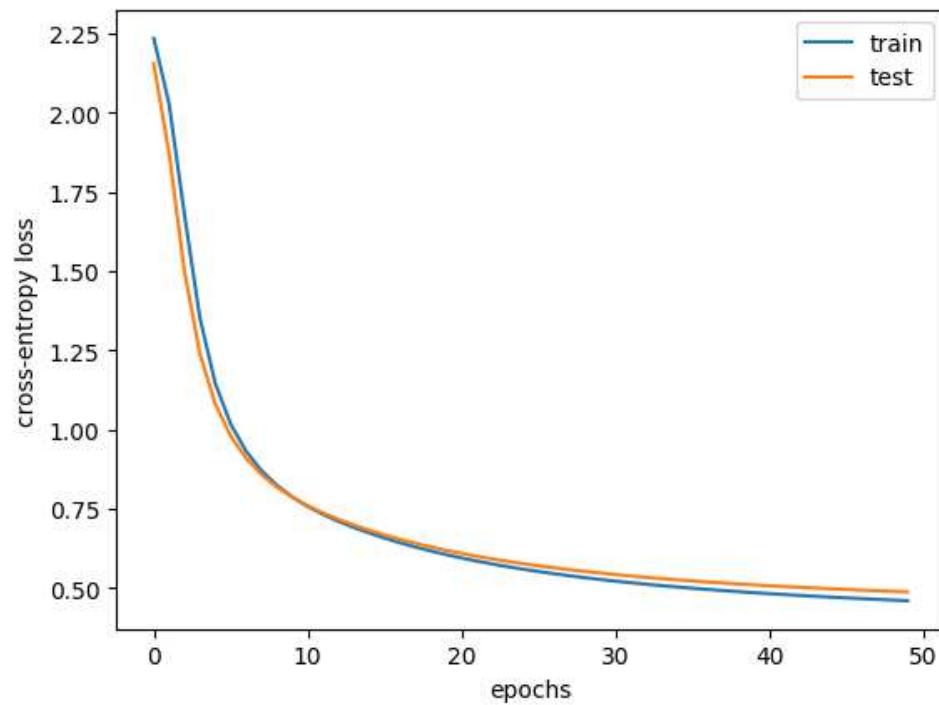
Learning rate $\alpha = 0.001$

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.softmax_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.Softmax(dim=1)
        )

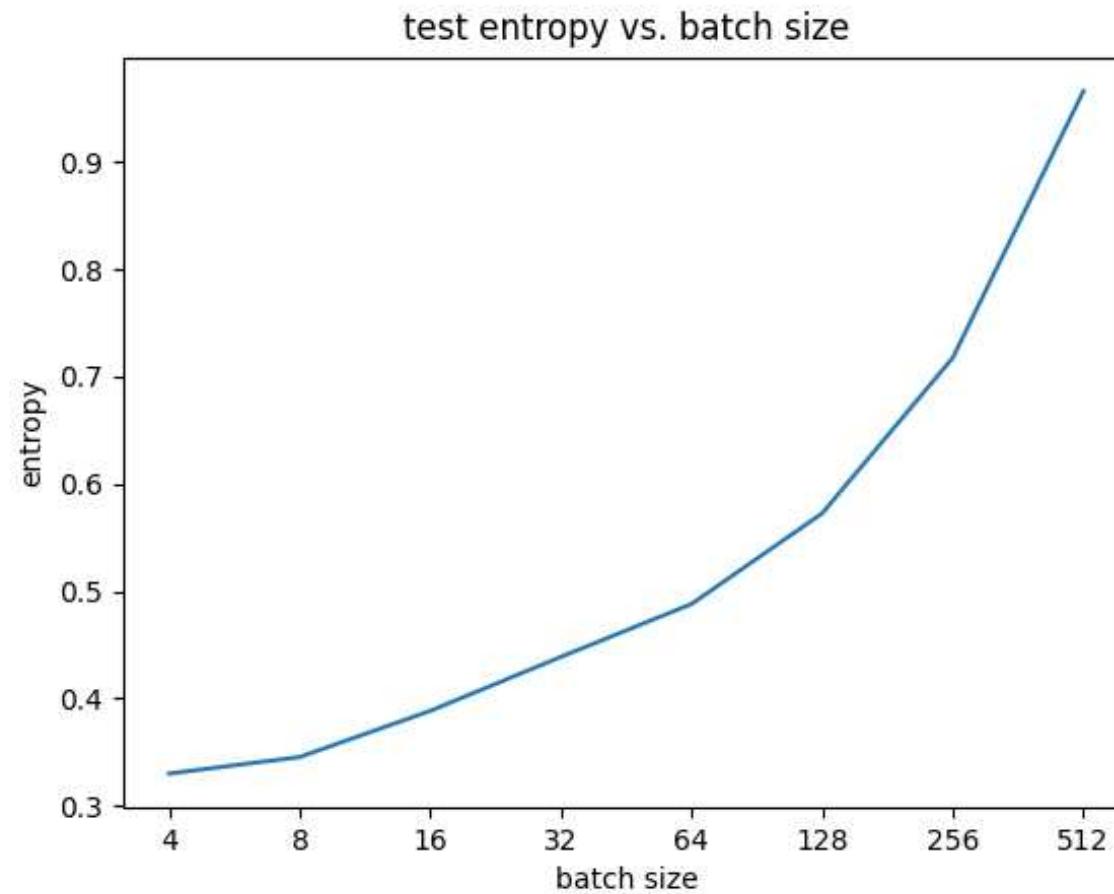
    def forward(self, x):
        x = self.flatten(x)
        logits = self.softmax_relu_stack(x)
        return logits

model = NeuralNetwork()
```

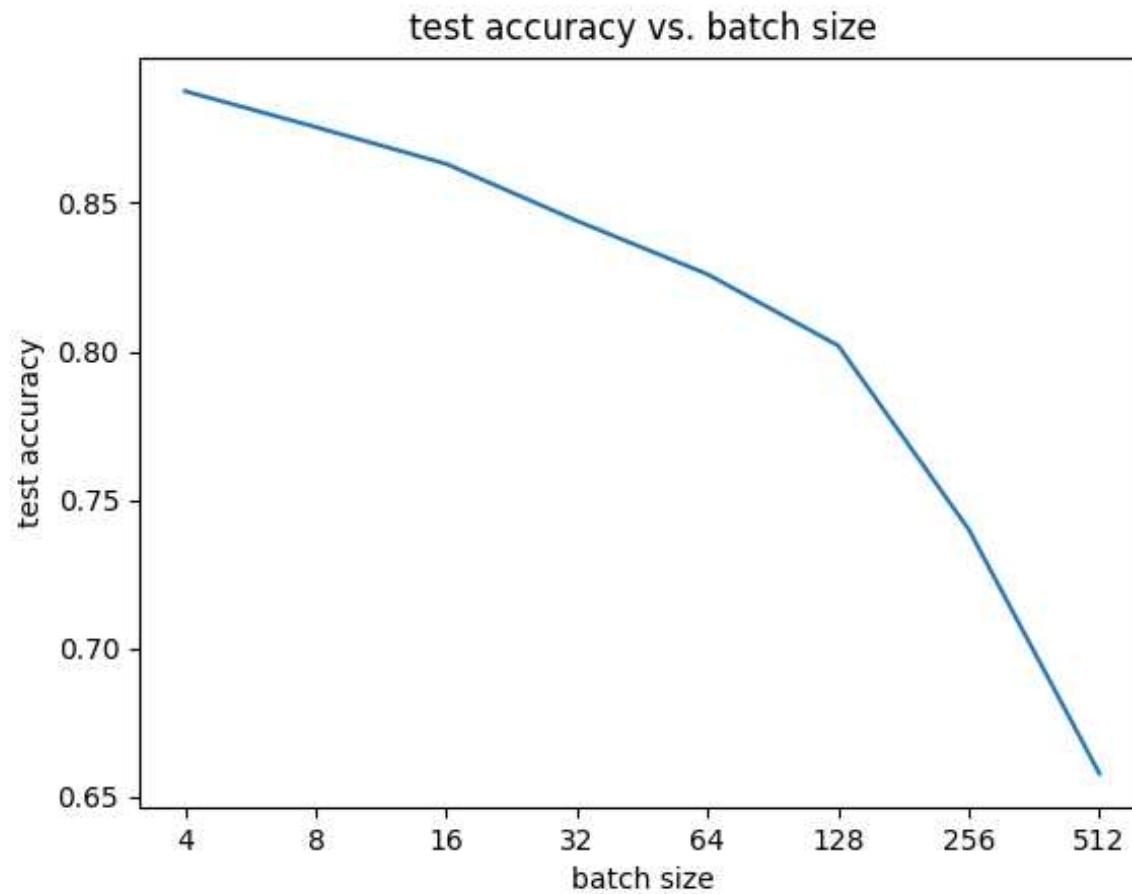
Example 5



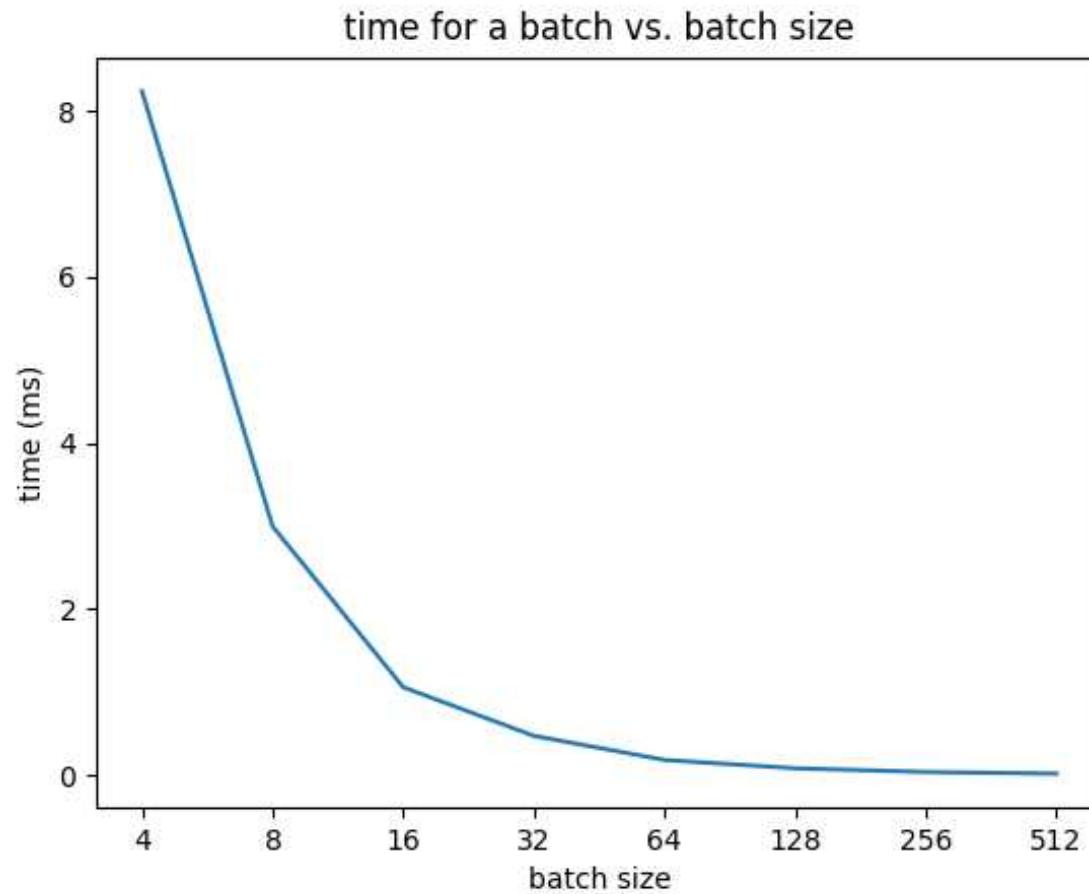
Example 5b: Effect of batch size



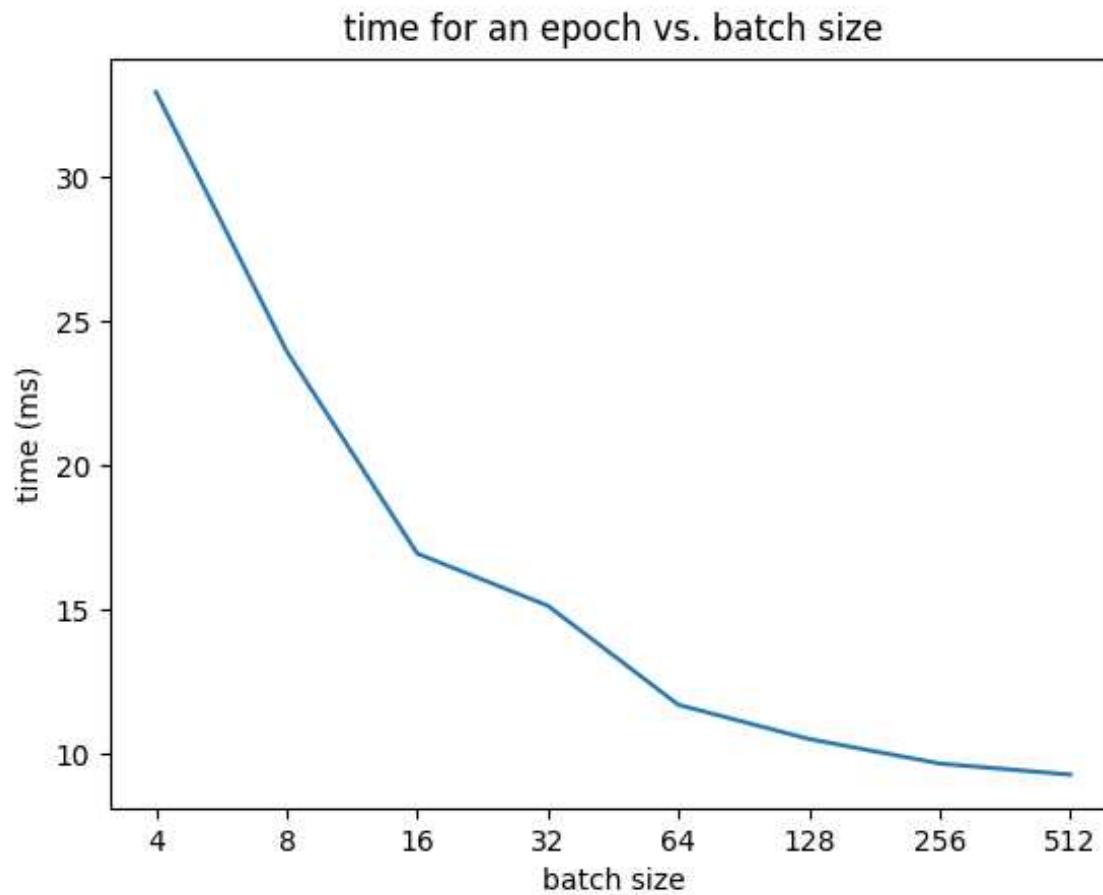
Example 5: Effect of batch size



Example 5: Effect of batch size



Example 5: Effect of batch size



Mini-batch SGD

In practice, gradient descent is performed on *mini-batch* updates of gradients within a *batch* or *block* of data of size B . In mini-batch SGD, the data is divided into blocks and the gradients are evaluated on blocks in an epoch in random order.

$B = 1$: stochastic (online) gradient descent

$B = P$ (size of training data): (batch) gradient descent

$1 < B < P \rightarrow$ mini-batch stochastic gradient descent

When B increases, more add-multiply operations per second, taking advantages of parallelism and matrix computations. On the other hand, as B increases, the number of computations per update (of weights, biases) increases.

Therefore, the curve of the time for weight update against batch size usually take a U-shape curve. There exists an optimal value of B – that depends on the sizes of the caches as well.

Selection of batch size

For SGD, it is desirable to randomly sample the patterns from training data in each epoch. In order to efficiently sample blocks, the training patterns are shuffled at the beginning of every training epoch and then blocks are sequentially fetched from memory.

Typical batch sizes: 16, 32, 64, 128, and 256.

The batch size is dependent on the size of caches of CPU and GPUs.

Summary

- Chain rule for backpropagation of gradients: $\nabla_{\mathbf{x}}J = \left(\frac{\partial y}{\partial x}\right)^T \nabla_{\mathbf{y}}J$
- FFN with one hidden layer (Shallow FFN)
- Backpropagation for FFN with one hidden layer:

$$\nabla_{\mathbf{z}}J = (\nabla_{\mathbf{U}}J) \mathbf{V}^T \cdot g'(\mathbf{Z})$$

- Backpropagation learning for deep FFN (DNN)
- Training deep neural networks (GD and SGD):
 - Forward propagation of activation
 - Backpropagation of gradients
 - Updating weights
- Parameters to be decided: depth, width, and batch size

Chapter 5

Model selection and overfitting

Neural networks and deep learning

Model Selection



In neural networks, there exist several free parameters: learning rate, batch size, no of layers, number of neurons, etc.

Every set of parameters of the network leads to a specific model.

How do we determine the “optimum” parameter(s) or the model for a given regression or classification problem?

Selecting the best model with the best parameter values

Performance estimates



How do we measure the performance of the network?

Some metrics:

1. Mean-square error/Root-mean square error for **regression** — the mean-squared error or its square root. A measure of the deviation from actual.

$$MSE = \frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2 \text{ and } RMSE = \sqrt{\frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K (d_{pk} - y_{pk})^2}$$

2. Classification error of a **classifier**.

$$\text{classification error} = \sum_{p=1}^P 1(d_p \neq y_p)$$

where d_p is the target and y_p is the predicted output of pattern p . $1(\cdot)$ is the indicator function.

True error or apparent error?

Apparent error (training error): the error on the training data.
What the learning algorithm tries to optimize.

True error: the error that will be obtained in use (i.e., over the whole *sample space*). What we want to optimize *but* unknown.

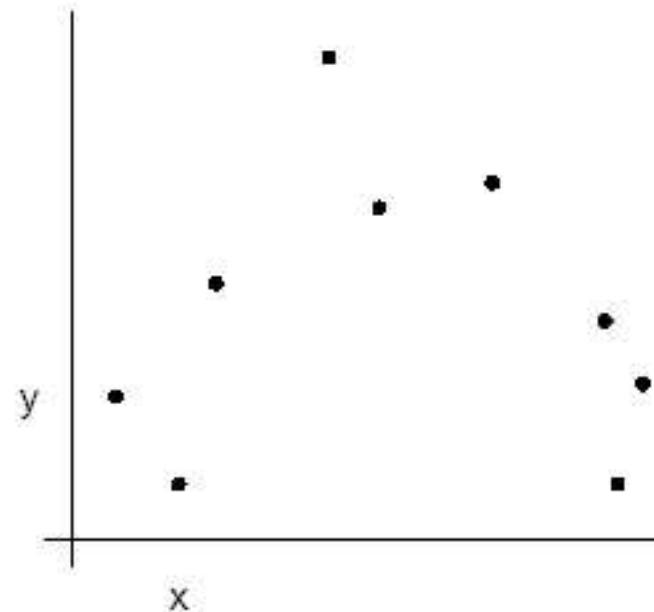
However, the **apparent error** is not always a good estimate of the **true error**. It is just an optimistic measure of it.

Test error: (out-of-sample error) an estimate of the true error obtained by testing the network on some independent data.

Generally, a larger test set helps provide a greater confidence on the accuracy of the estimate.

Example: Regression

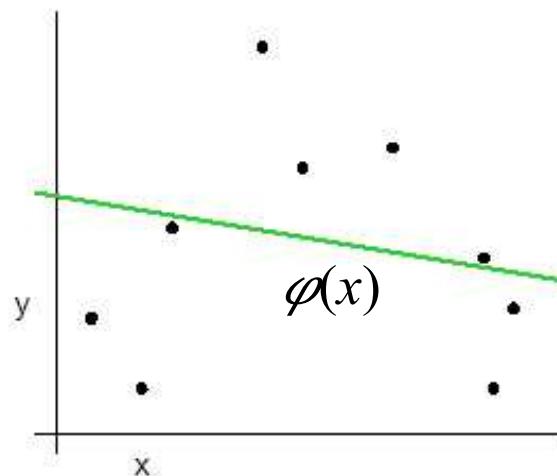
Given the following sample data:



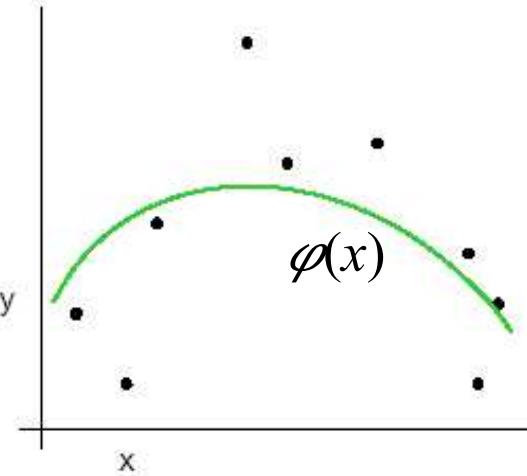
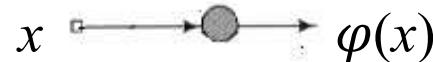
Approximate $y \approx \varphi(x)$ using an NN

Example: Regression

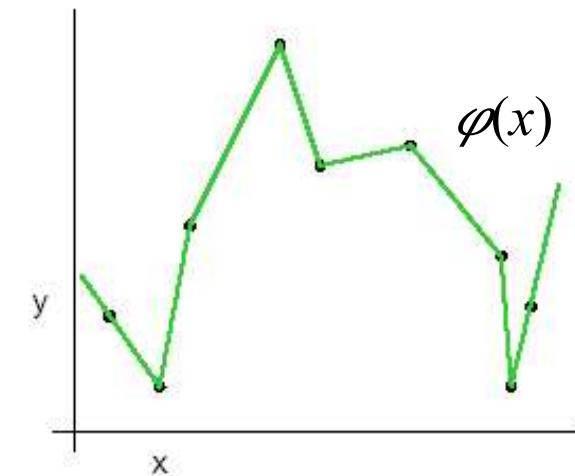
Which one is the best approximation model?



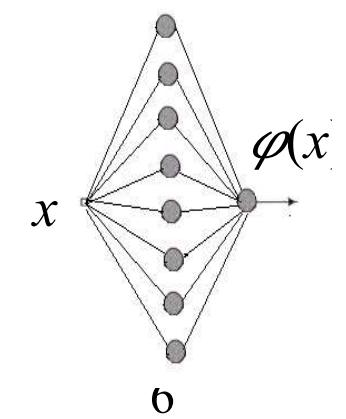
Linear regression,
Linear Activation fn



Quadratic (non-linear)
regression, Sigmoid
Activation fn

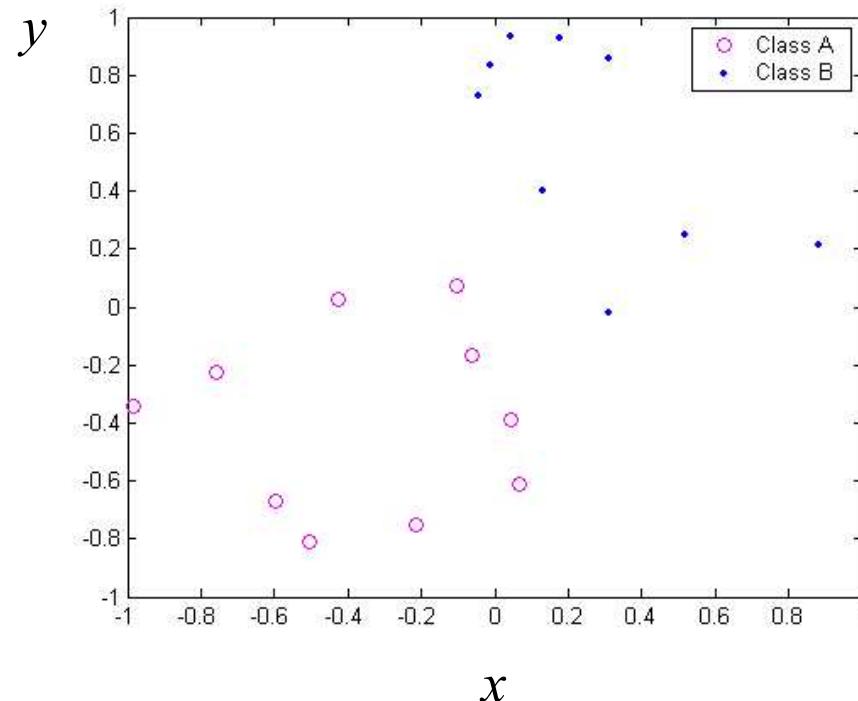


Piecewise
Linear
regression,
Linear
Activation fn



Example: Classification

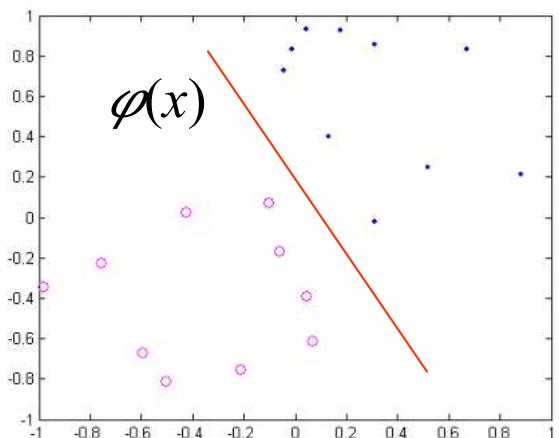
Given the following sample data:



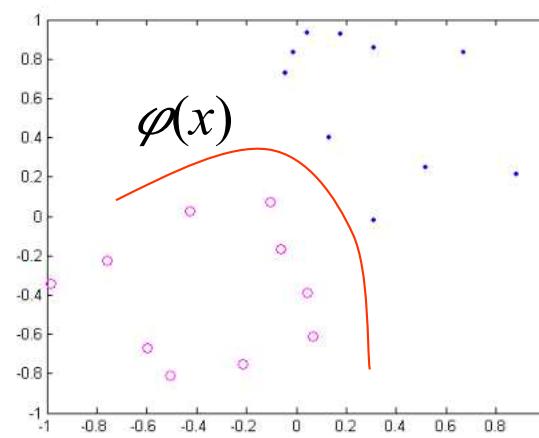
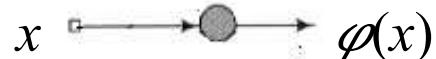
Classify the following data using an NN.

Example: Classification

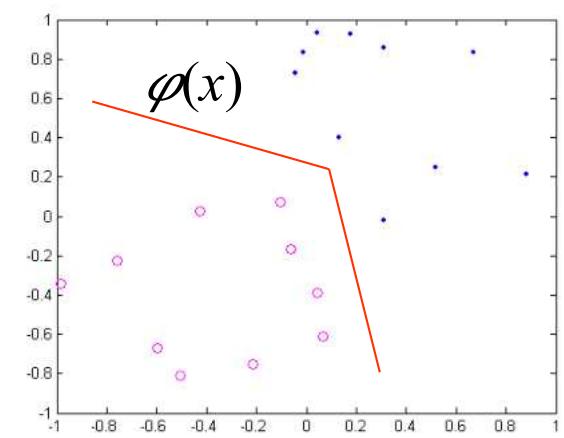
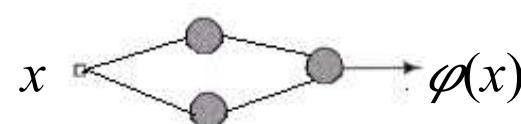
Which one is the best classification model?



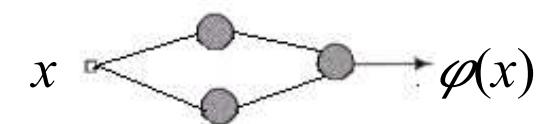
Linear Classification,
Threshold Activation fn



Quadratic
Classification, Sigmoid
Activation fn



Piecewise Linear
Classification, Linear
Activation fn



Estimation of true error

Intuition: Choose the model with the best fit to the data?

Meaning: Choose the model that provides the lowest error rate on the entire sample population. Of course, that error rate is the *true error rate*.

“However, to choose a model, we must first know how to **estimate the error** of a model.”

The entire **sample population** is often unavailable and only **example data** is available.

Validation

In real applications, we only have access to a finite set of examples, usually smaller than we wanted.

Validation is the approach to use the entire example data available to build the model and estimate the error rate. The validation uses a part of the data to select the model, which is known as the *validation set*.

Validation attempts to solve fundamental problems encountered:

- The model tends to *overfit* the training data. It is not uncommon to have 100% correct classification on training data.
- There is no way of knowing how well the model performs on *unseen data*
- The *error rate estimate* will be overly optimistic (usually lower than the true error rate) . Need to get an unbiased estimate.

Validation Methods

An effective approach is to split the entire data into subsets, i.e., Training/Validation/Testing datasets.

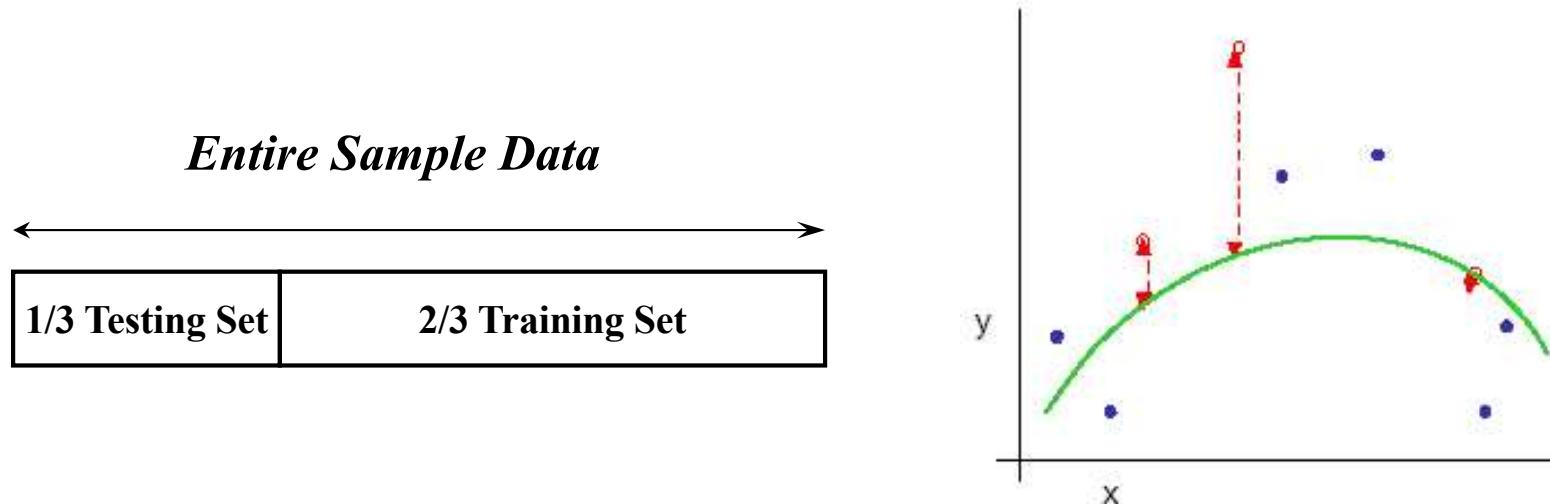
Some Validation Methods:

- The Holdout (1/3 - 2/3 rule for test and train partitions)
- Re-sampling techniques
 - Random Subsampling
 - K-fold Cross-Validation
 - Leave one out Cross-Validation
- Three-way data splits (train-validation-test partitions)

Holdout Method

Split entire dataset into two sets:

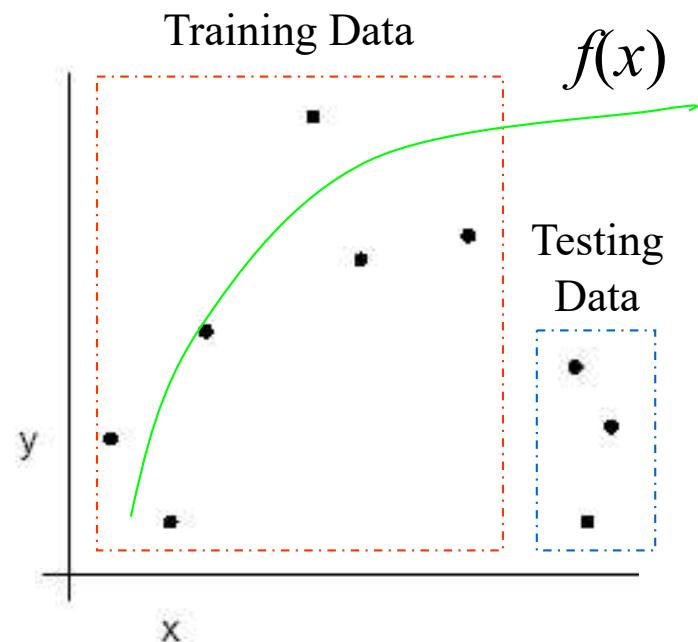
- **Training set** (blue): used to train the classifier
- **Testing set** (red): used to estimate the error rate of the trained classifier on unseen data samples



Holdout Method

The holdout method has two basic drawbacks:

- By setting some samples for testing, the training dataset becomes smaller
- Use of a single train-and-test experiment, could lead to misleading estimate if an “unfortunate” split happens



An “unfortunate” split:
training data may not cover
the space of testing data

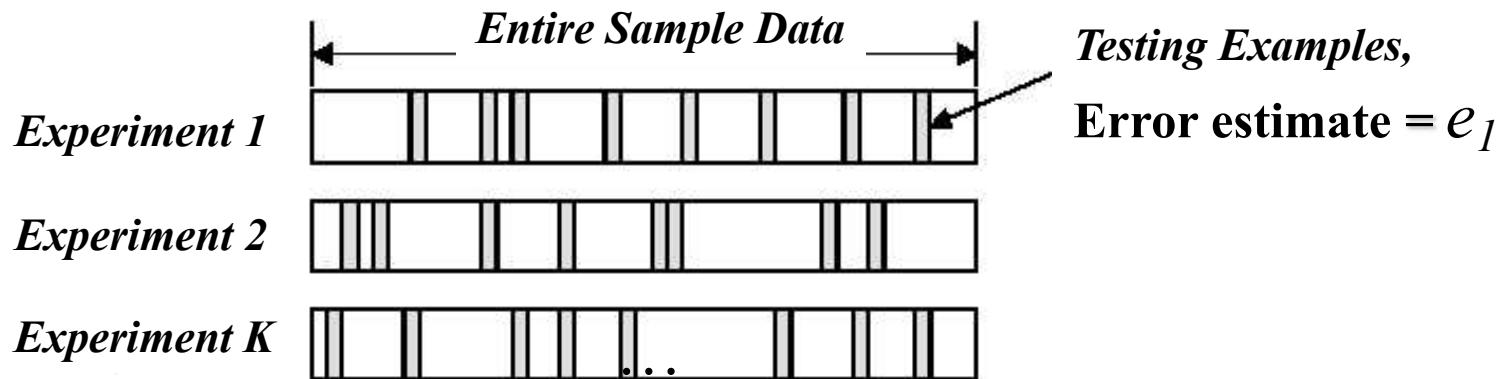
Random Sampling Methods

Limitations of the holdout can be overcome with a family of resampling methods at the expense of more computations:

- Random Subsampling
- K-Fold Cross-Validation
- Leave-one-out (LOO) Cross-Validation

K Data Splits Random SubSampling

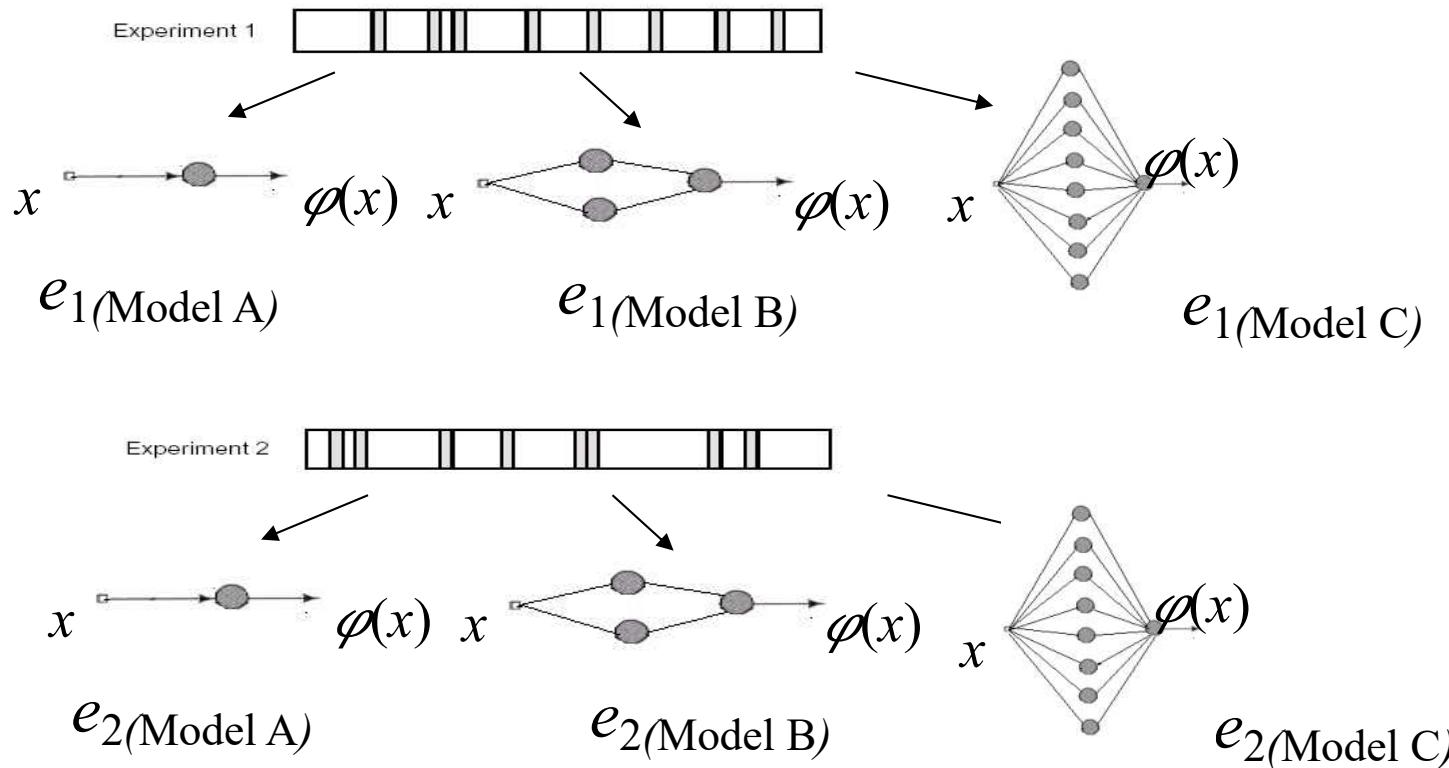
Random Subsampling performs K data splits of the dataset for training and testing.



Each split randomly selects a (fixed) no. of examples. For each data split we retrain the classifier from scratch with the training data. Let the error estimate obtained for i th split (experiments) be e_i .

$$\text{Average test error} = \frac{1}{K} \sum_{i=1}^K e_i$$

Example: K=2 Data Splits Random SubSampling



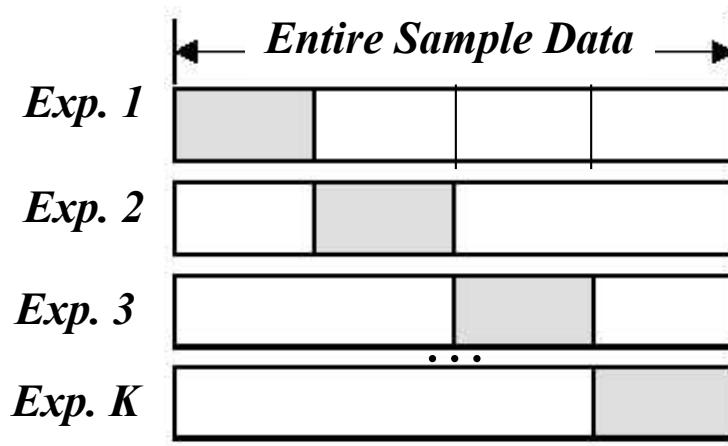
$$\text{Test error}_{(M)} = \frac{1}{2} (e_{1(M)} + e_{2(M)})$$

Choose the model with the best test error, i.e., lowest average test error!

K-fold Cross-Validation

Create a **K-fold** partition of the dataset:

- For each of K experiments, use $K-1$ folds for training and the remaining one-fold for testing



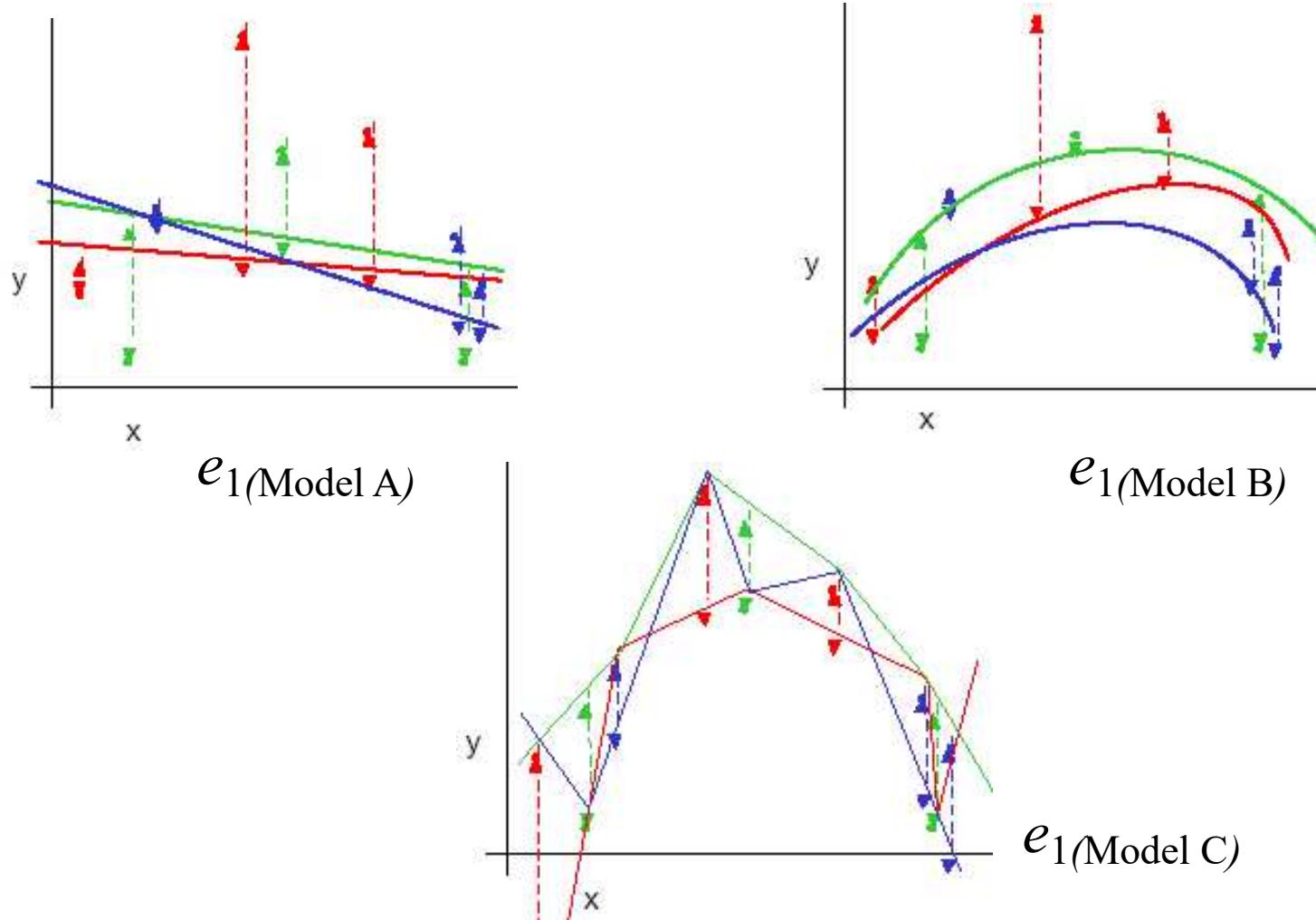
Let the error estimate for i th experiment on test partition be e_i .

Cross-validation error

$$\frac{1}{K} \sum_{i=1}^K e_i$$

K -fold cross validation is similar to Random Subsampling. The *advantage* of K -Fold Cross validation is that all examples in the dataset are eventually used for both training and testing

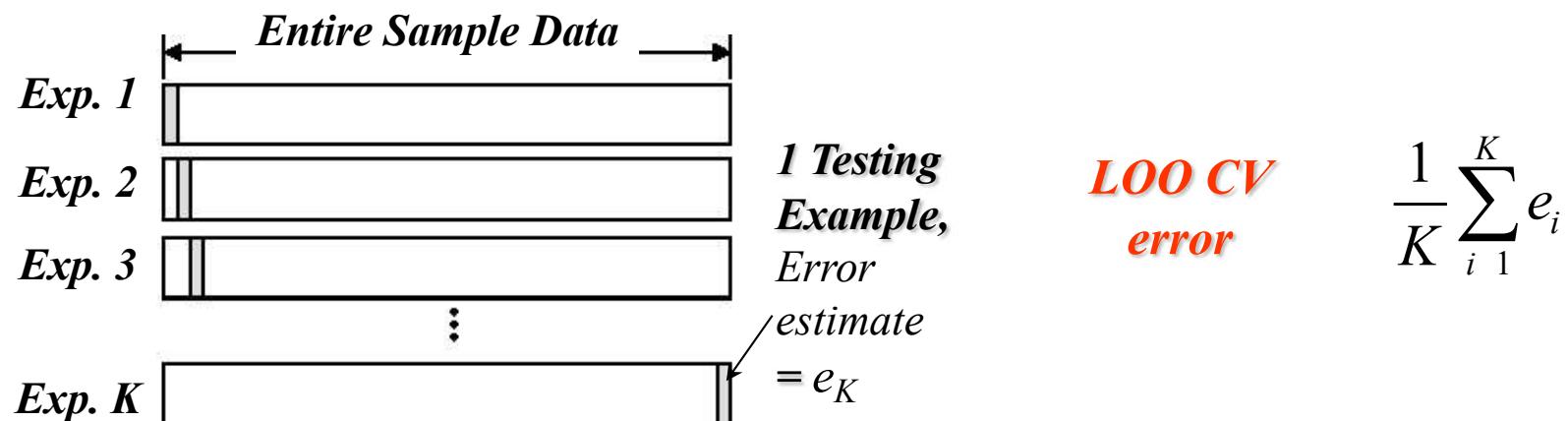
Example: 3-fold Cross-Validation



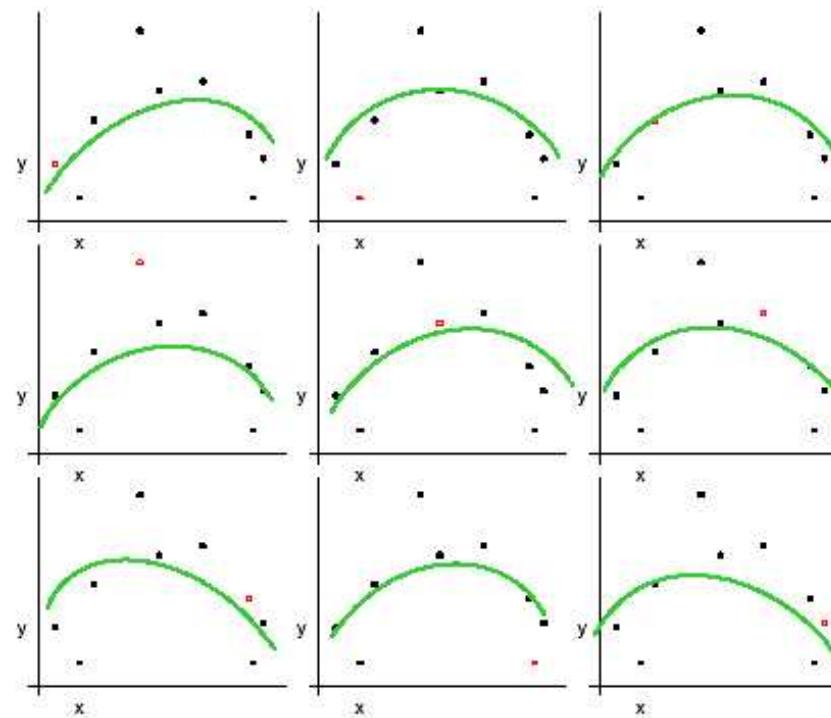
Leave-One-Out (LOO) Cross-Validation

Leave-One-Out is the degenerate case of K -Fold Cross Validation, where K is chosen as the total number of examples:

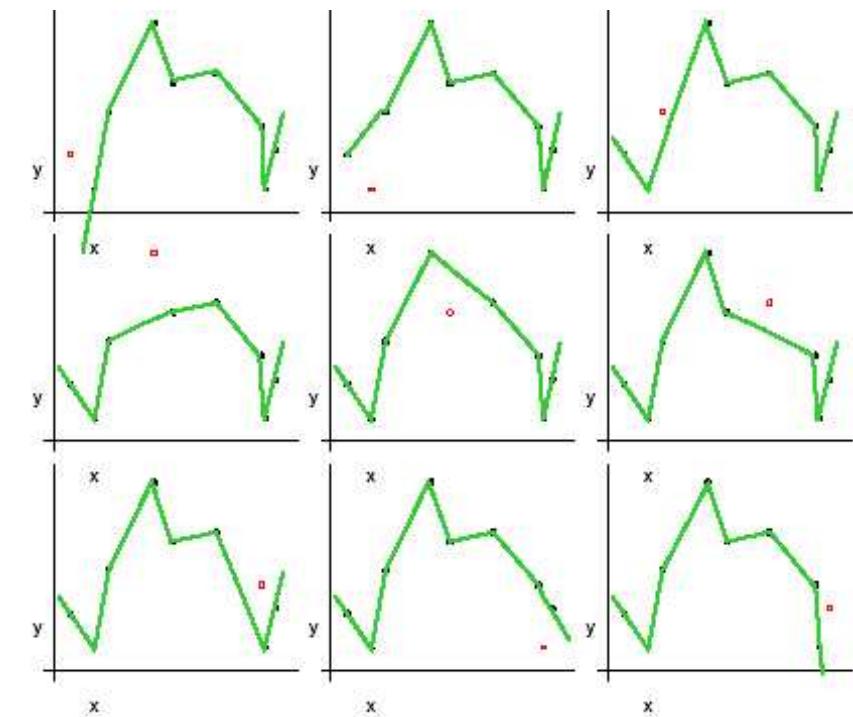
- For a dataset with N examples, perform N experiments, i.e., $N = K$.
- For each experiment use $N-1$ examples for training and the remaining one example for testing.



Example: Leave-One-Out Cross-Validation



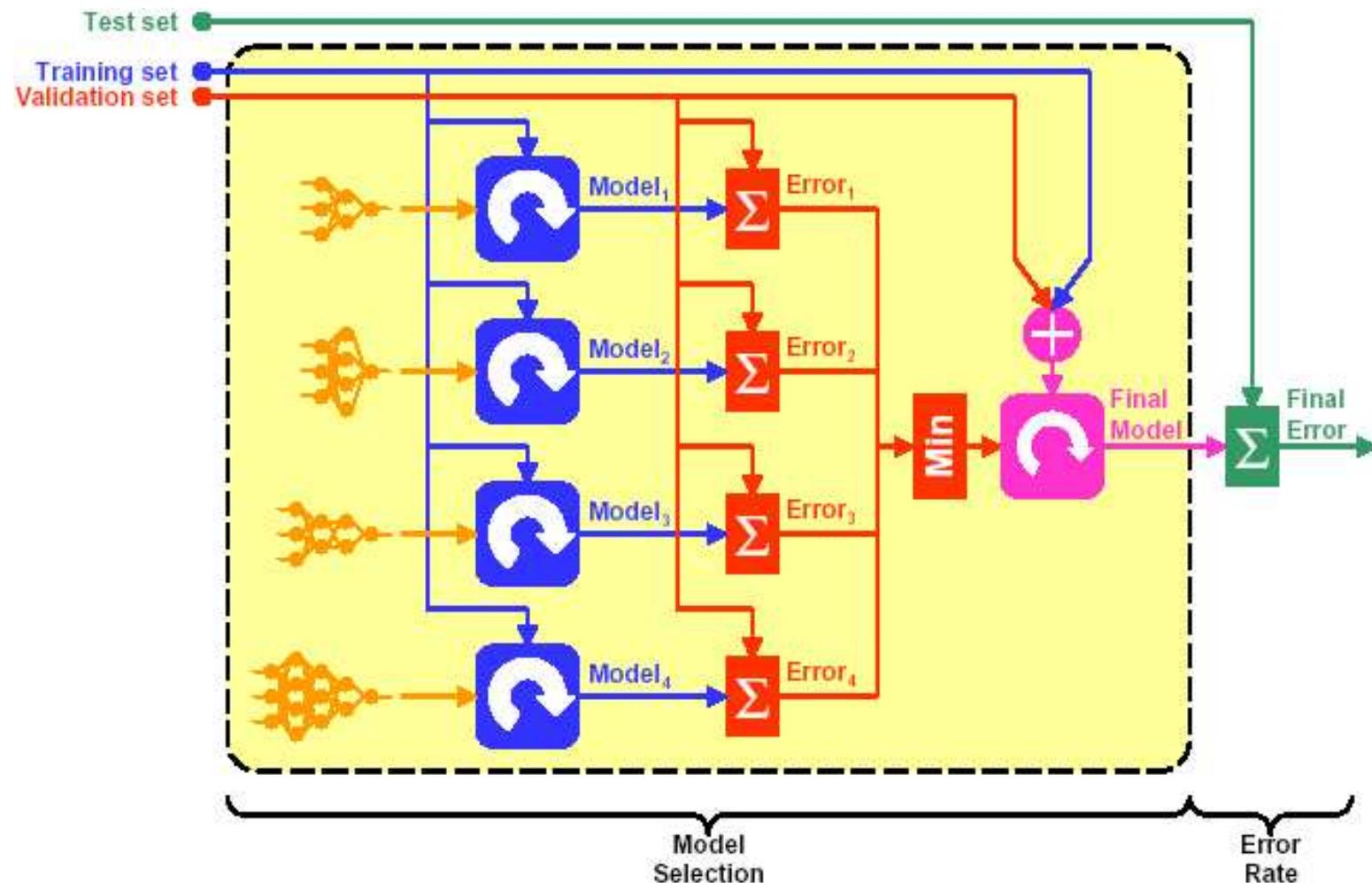
LOOCV Quadratic
Approximation



LOOCV Piecewise
Linear Approximation

Three-Way Data Splits Method

Dataset is partitioned in to training set, **validation set**, and testing set.



Three-Way Data Splits Method

If model selection and true error estimates are to be computed simultaneously, the data needs to be divided into three disjoint sets:

- **Training set:** examples for *learning* to fit the parameters of several possible classifiers. In the case of DNN, we would use the training set to find the “optimal” weights with the gradient descent rule.
- **Validation set:** examples to *determine* the error J_m of different models m , using the validation set. The optimal model m^* is given by
$$m^* = \operatorname{argmin}_m J_m$$
- **Training + Validation set:** combine examples used to re-train/redesign $model_{m^*}$, and find new “optimal” weights and biases.
- **Test set:** examples used only to *assess* the performance of a *trained model* m^* . We will use the test data to estimate the error rate after we have trained the final model with train + validation data.

Three-Way Data Splits Method

Why separate test and validation sets?

- The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is also used to select in the process of final model selection.
- After assessing the final model, an independent test set is required to estimate the performance of the final model.

“NO FURTHERING TUNNING OF THE MODEL IS ALLOWED!”

Examples 1-3: Iris dataset

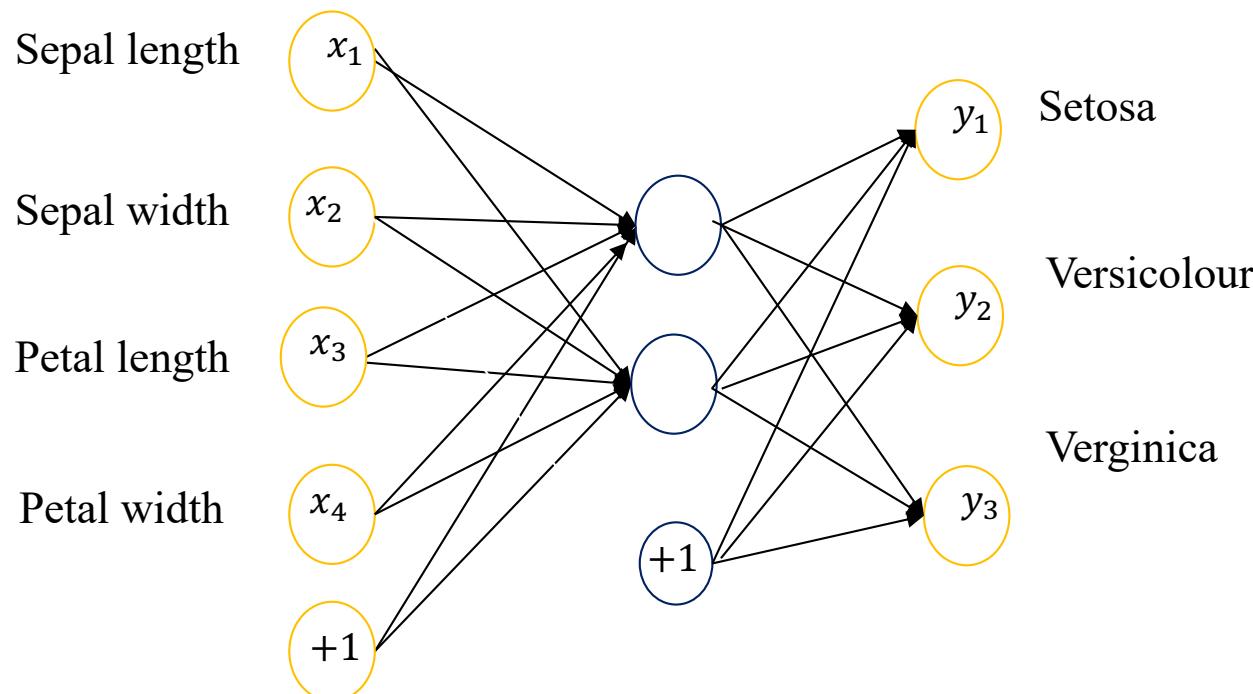
<https://archive.ics.uci.edu/ml/datasets/Iris>

Three classes of iris flower: Setosa, versicolour, and virginica

Four features: Sepal length, sepal width, petal length, petal width

150 data points

DNN with one hidden layer. **Determine number of hidden neurons?**

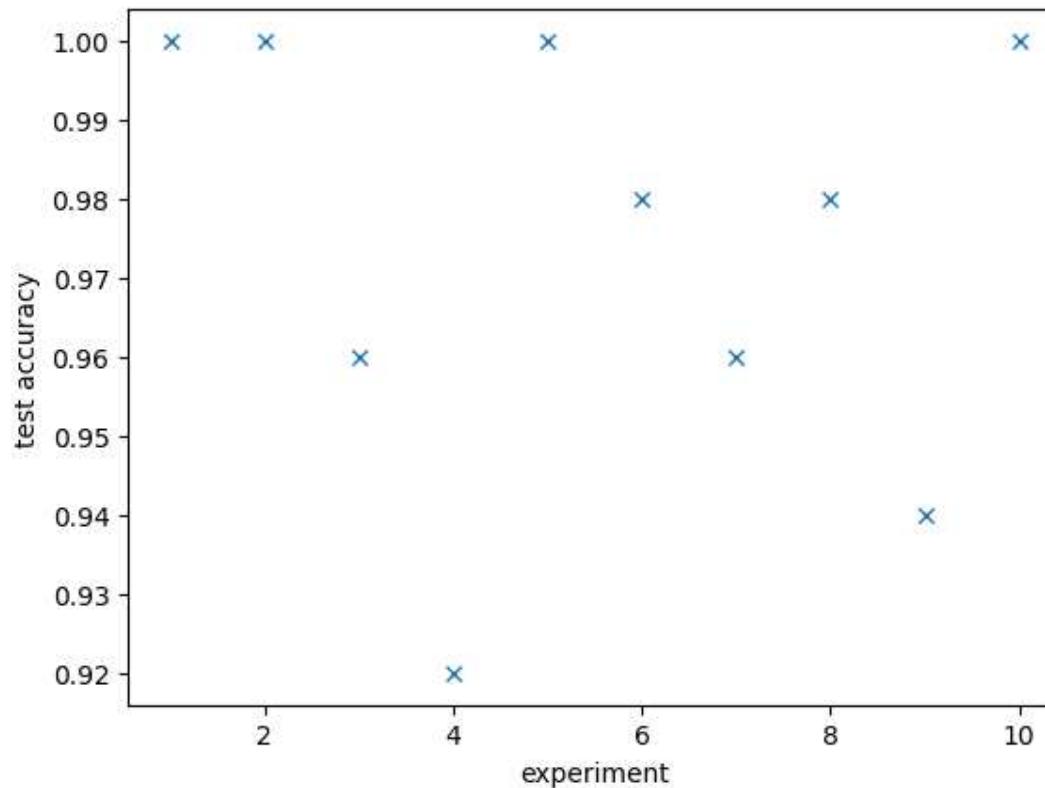


Example 1a: Random subsampling

150 data points

In each experiment, 50 points for testing and 100 for training

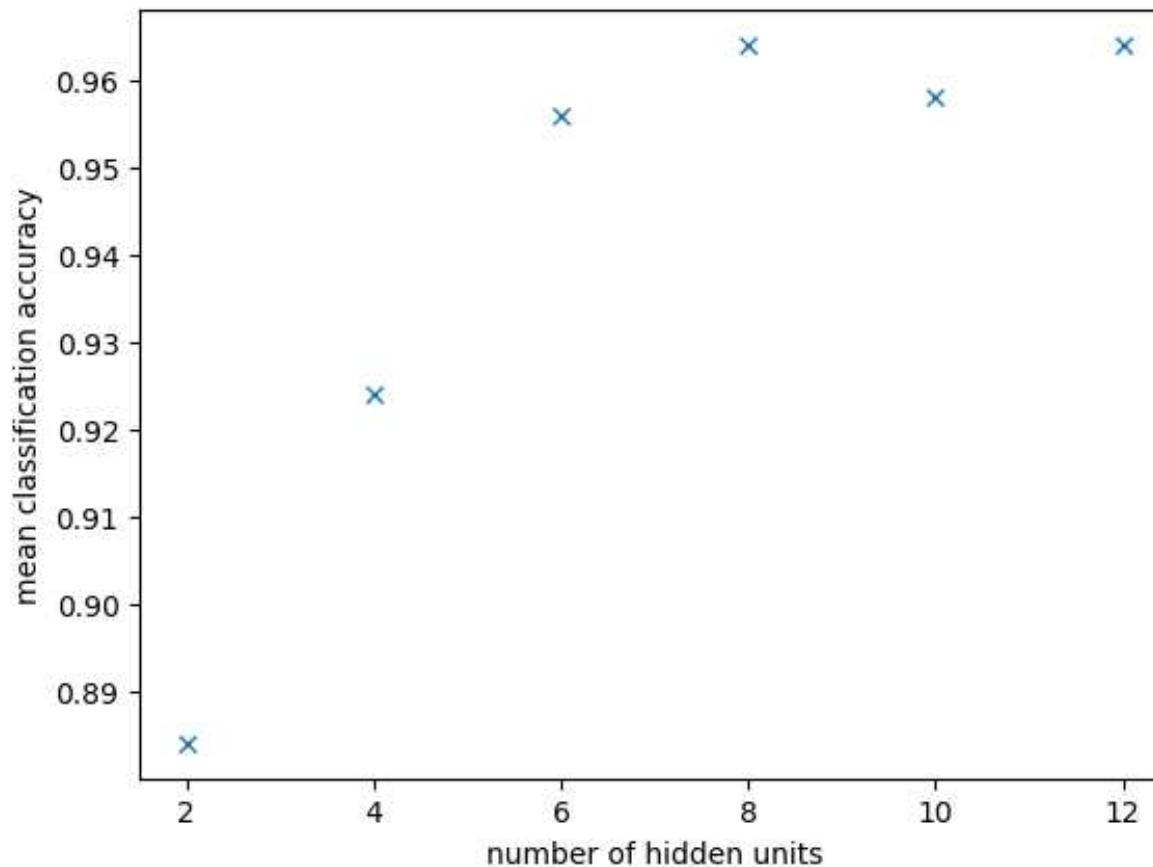
Example: 5 hidden neurons, 10 experiments



Mean accuracy = 97.4%

Example 1b

For different number of hidden units, misclassification errors in 10 experiments



Optimum number of hidden units = 8

Accuracy = 96.4%

Example 2a: Cross validation

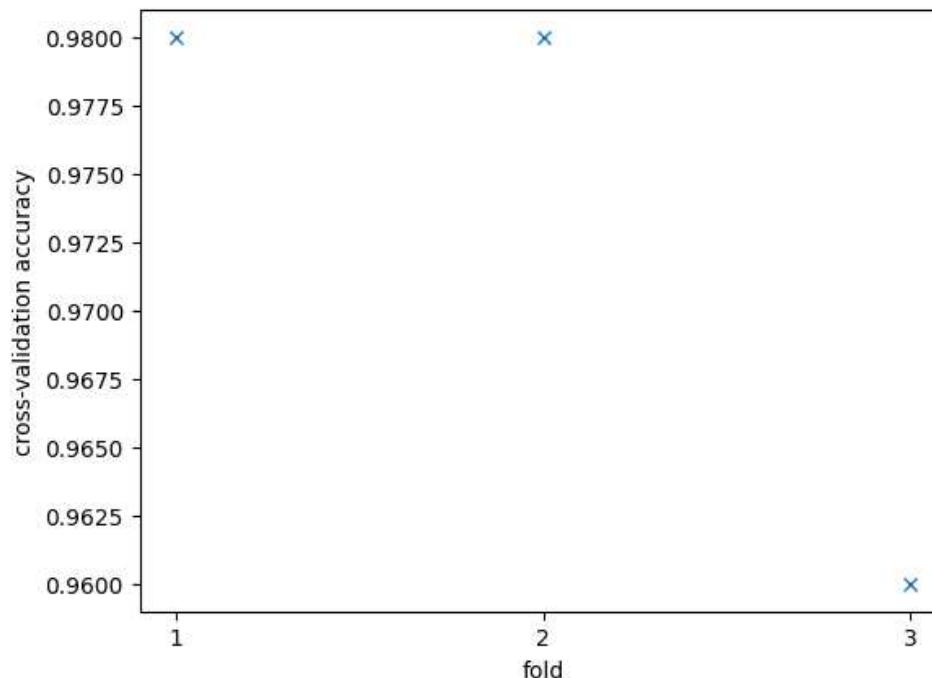
150 data points

50 50 50

3-fold cross validation: 50 data points in one fold.

Two folds are used for training and the remaining fold for testing

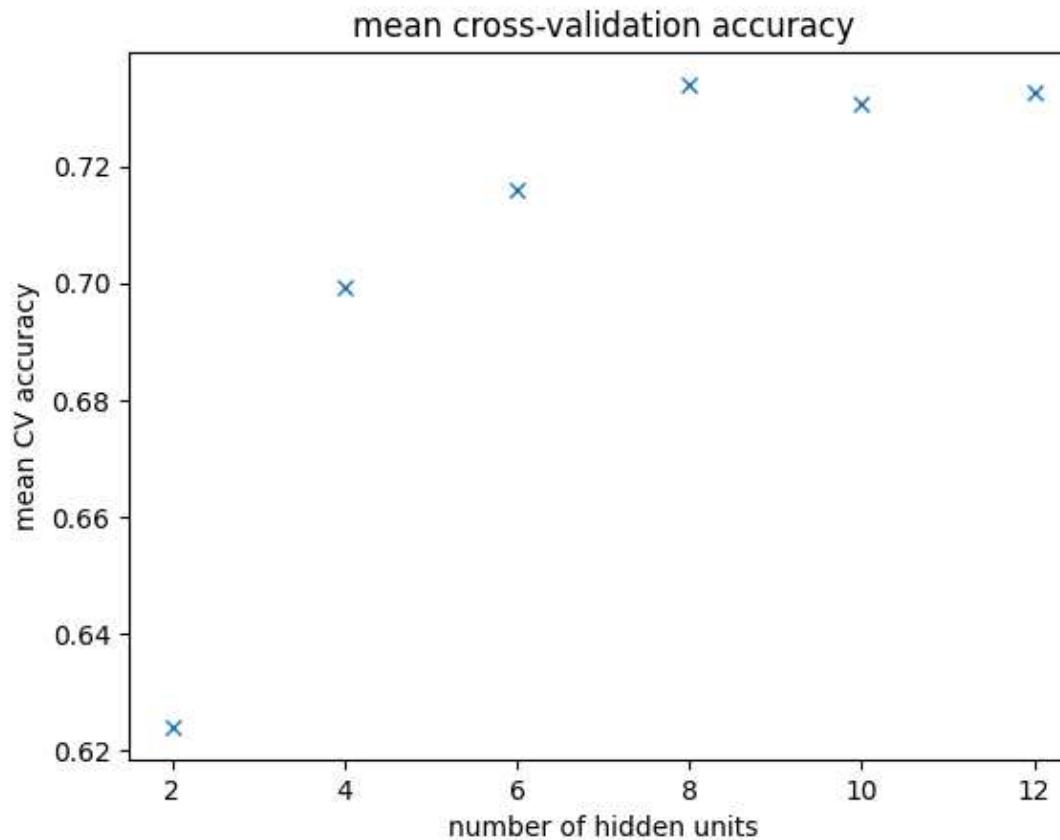
Example: hidden number of units = 5



3-fold cross-validation (CV) accuracy = 97.3%

Example 2b

Mean CV error for 10 experiments for different number of hidden units:

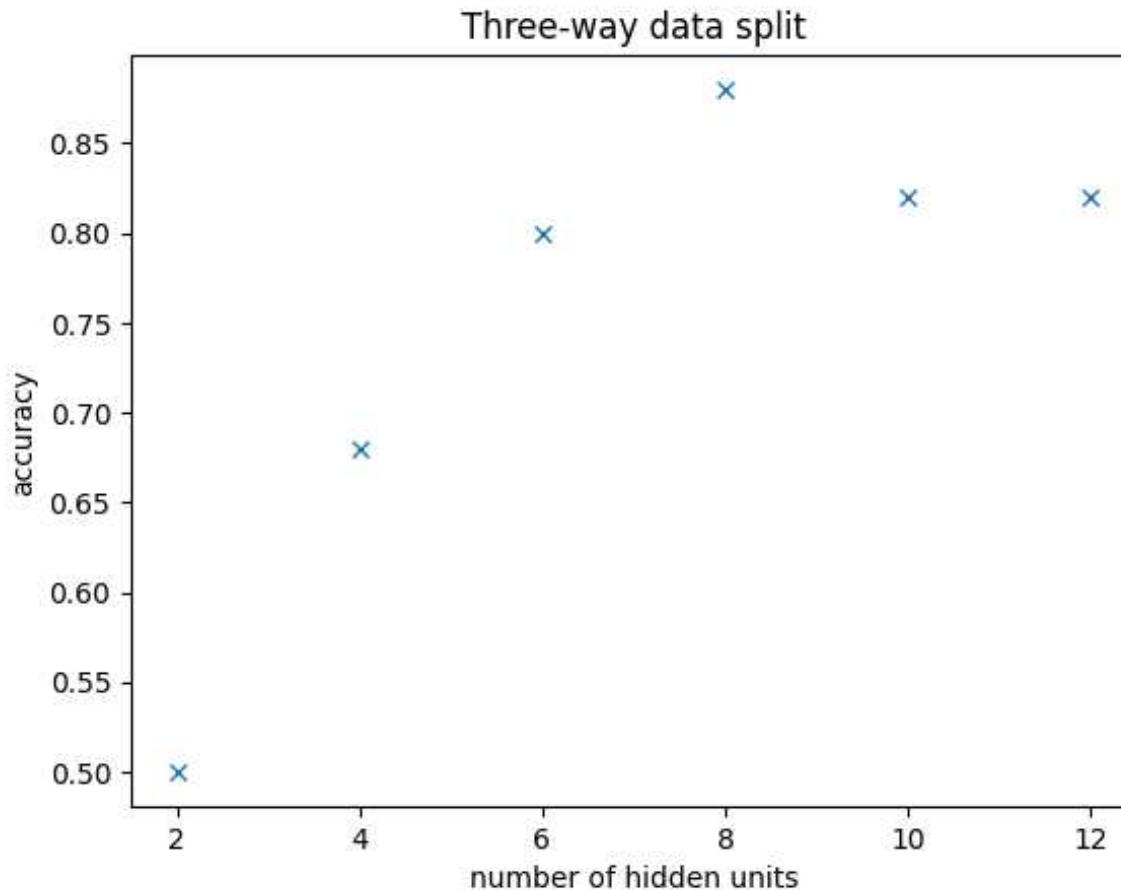


Optimum number of hidden neurons = 8
Cross-validation accuracy = 73.4%

Example 3a: Three-way data splits

150 data points

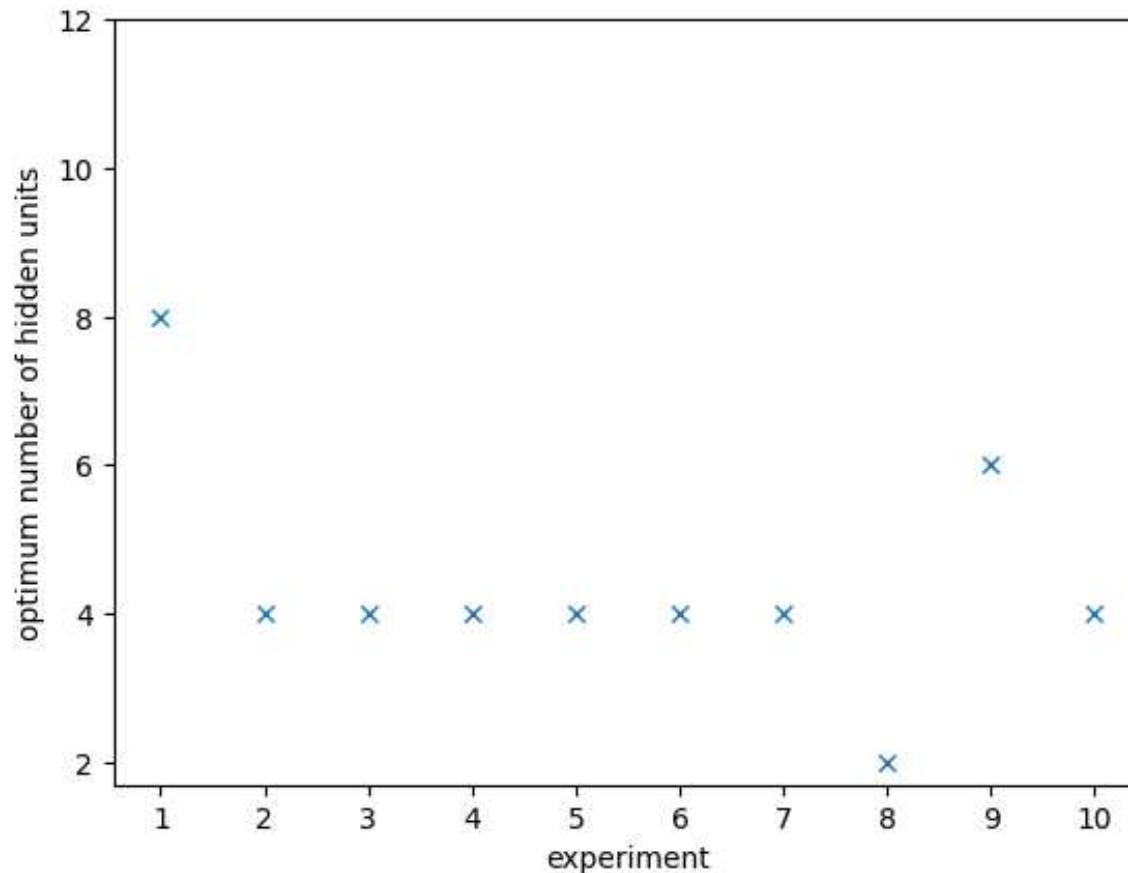
50 data points each for training, for validation, and for testing.



Optimum number of hidden neurons = 8
Accuracy = 88.0%

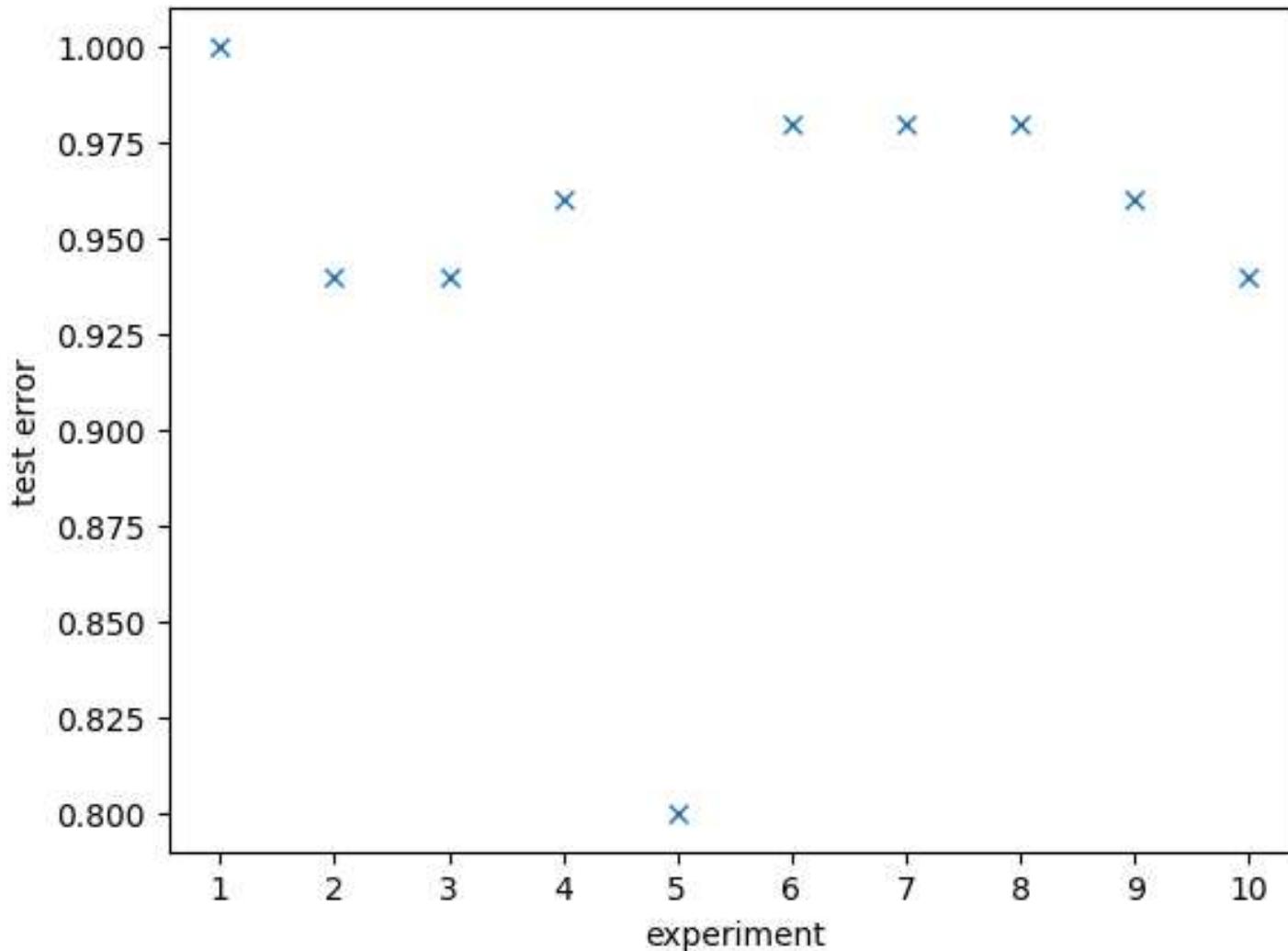
Example 3b

One way to further improve the results is to repeat it for many experiments, say for 10 experiments:



Optimum number of hidden neurons = 4

Example 3b



Test accuracy = 93.4% (average)

Model Complexity

Complex models: models with many adjustable weights and biases will

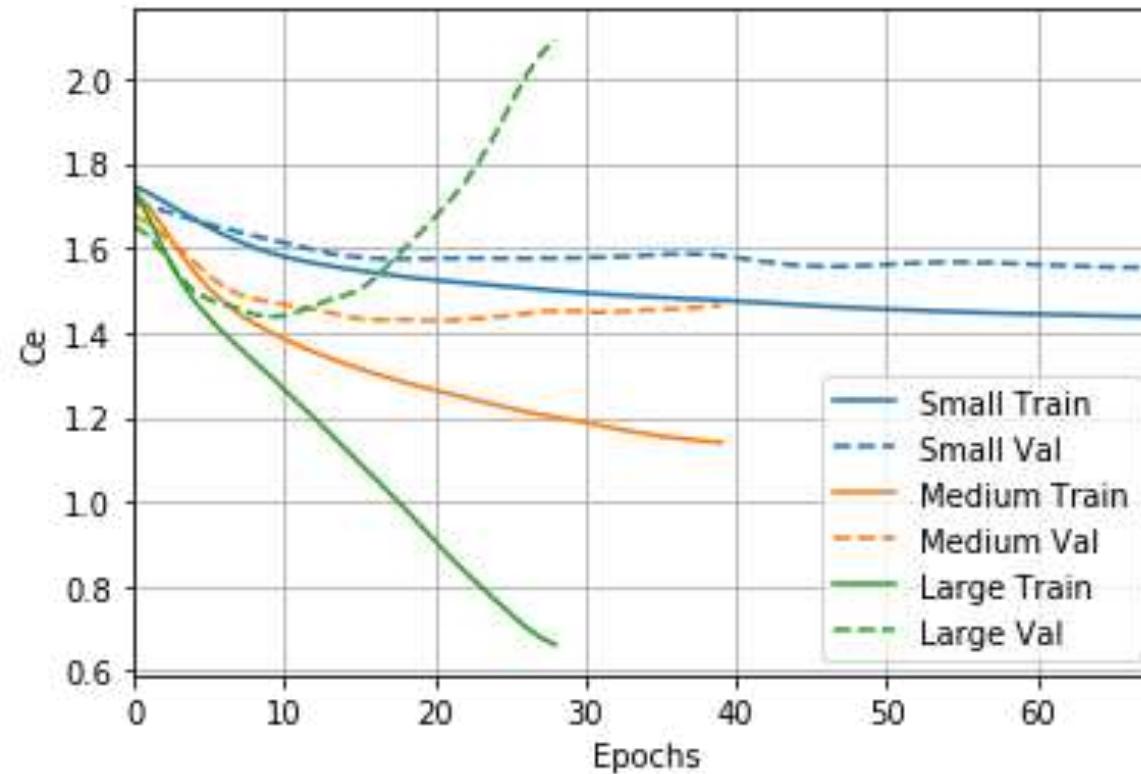
- more likely to be able to solve the required task,
- more likely to memorize the training data without solving the task.

Simple models: The simpler the model that can learn from the training data is more likely to generalize over the entire sample space. May not learn the problem well.

This is the fundamental trade-off:

- Too simple — cannot do the task because not enough parameters to learn. e.g., 5 hidden neurons. (*underfitting*)
- Too complex — cannot generalize from small and noisy datasets well, e.g., 20 hidden neurons. (*overfitting*)

Overfitting and underfitting



Small model is **underfitting**
Medium model seems just right
Large model is **overfitting**

Overfitting

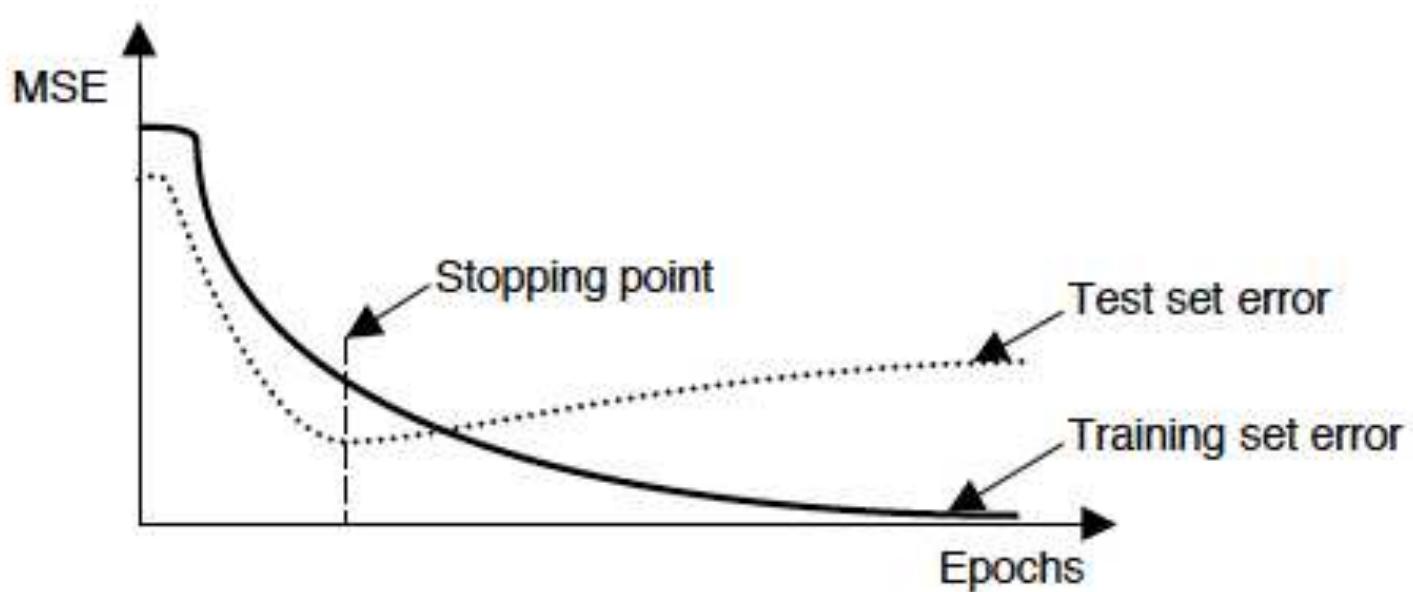
Overfitting is one of the problems that occur during training of neural networks, which drives the training error of the network to a very small value at the expense of the test error. The network learns to respond correctly to the training inputs by remembering them too much but is unable to generalize to produce correct outputs to novel inputs.

- Overfitting happens when the amount of training data is inadequate in comparison to the number of network parameters to learn.
- Overfitting occurs when the weights and biases become too large and are fine-tuned to remember the training patterns too much.
- Even your model is right, too much training can cause overfitting.

Methods to overcome overfitting

1. Early stopping
2. Regularization of weights
3. Dropouts

Early stopping



Training of the network is to be stopped when the validation error starts increasing. Early stopping can be used in test/validation by stopping when the validation error is minimum.

Regularization of weights

During overfitting, some weights attain large values to reduce training error, jeopardizing its ability to generalize. In order to avoid this, a *penalty* term (*regularization* term) is added to the cost function.

For a network with weights $\mathbf{W} = \{w_{ij}\}$ and bias \mathbf{b} , the penalized (or regularized) cost function $J_1(\mathbf{W}, \mathbf{b})$ is defined as

$$J_1 = J + \beta_1 \sum_{i,j} |w_{ij}| + \beta_2 \sum_{ij} (w_{ij})^2$$

where $J(\mathbf{W}, \mathbf{b})$ is the standard cost function (i.e., m.s.e. or cross-entropy),

$$L^1 - norm = \sum_{i,j} |w_{ij}|$$

$$L^2 - norm = \sum_{ij} (w_{ij})^2$$

And β_1 and β_2 are known as L^1 and L^2 regularization (penalty) constants, respectively. These penalties discourage weights from attaining large values.

L2 regularization of weights

Regularization is usually not applied on bias terms. L^2 regularization is most popular on weights.

$$J_1 = J + \beta_2 \sum_{ij} (w_{ij})^2$$

Gradient of the regularized cost wrt weights:

$$\nabla_{\mathbf{W}} J_1 = \nabla_{\mathbf{W}} J + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial \mathbf{W}} \quad (\text{A})$$

L2 regularization of weights

$$L^2 \text{ norm} = \sum_{ij} (w_{ij})^2 = w_{11}^2 + w_{12}^2 + \dots + w_{Kn}^2$$
$$\frac{\partial (\sum_{ij} (w_{ij})^2)}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial \sum (w_{ij})^2}{\partial w_{11}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{12}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{1k}} \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{21}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{22}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{2k}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{n1}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{n2}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{nK}} \end{pmatrix} = 2\mathbf{W}$$

L2 regularization of weights

Substituting in (A):

$$\nabla_{\mathbf{W}} J_1 = \nabla_{\mathbf{W}} J + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial \mathbf{W}} = \nabla_{\mathbf{W}} J + 2\beta_2 \mathbf{W}$$

For gradient decent learning that uses regularized cost function:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J_1$$

Substituting above:

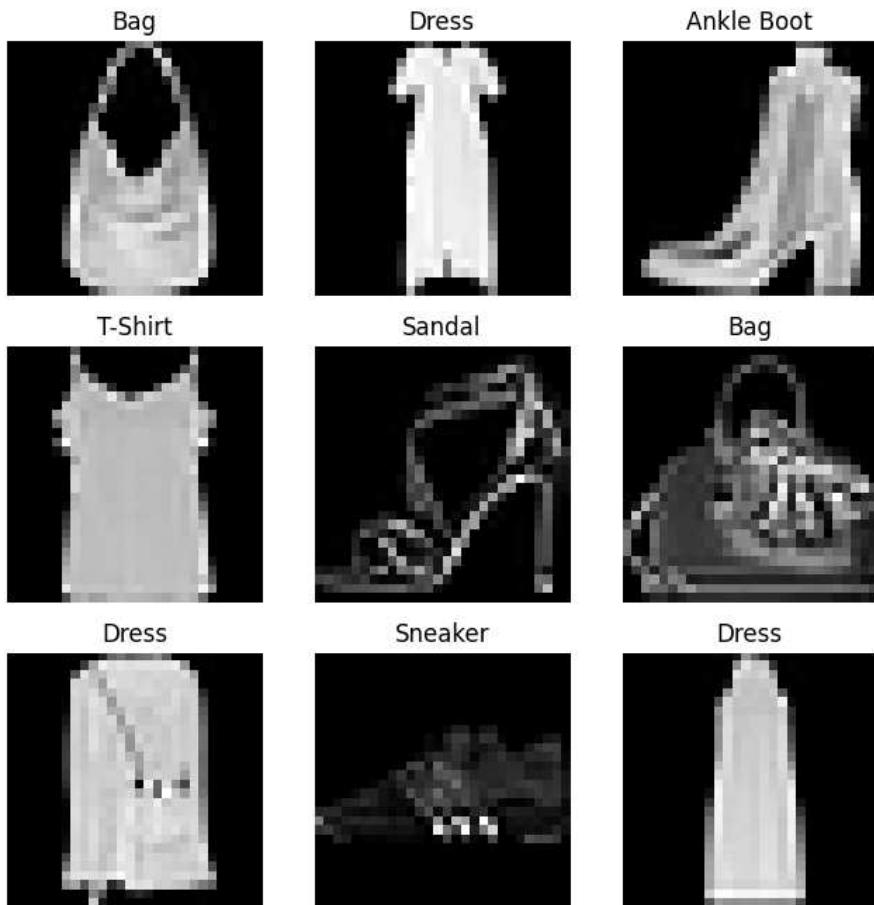
$$\mathbf{W} \leftarrow \mathbf{W} - \alpha (\nabla_{\mathbf{W}} J + \beta \mathbf{W})$$

where $\beta = 2\beta_2$

β is known as the *weight decay parameter*.

That is for L^2 regularization, the weight matrix is weighted by decay parameter and added to the gradient term.

Fashion MNIST dataset



<https://github.com/zalandoresearch/fashion-mnist>

Example 4: Early stopping and weight decay

DNN with [784, 400, 400, 400, 10] architecture.

Let's implement L2 regularization with $\beta = 0.0001$

Use early stopping to terminate learning.

```
class NeuralNetwork(nn.Module):
    def __init__(self, hidden_size=500):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Example 4

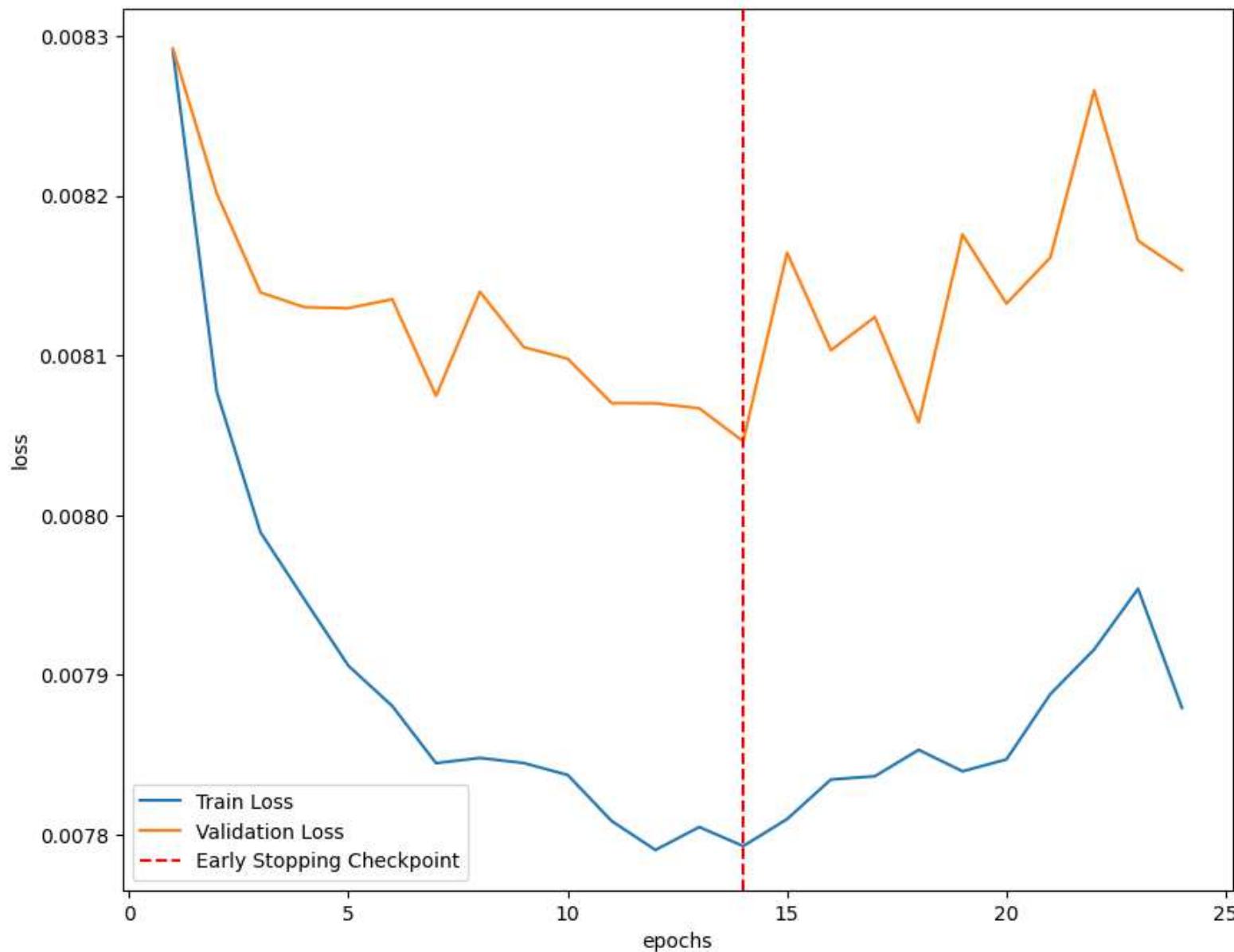
```
class EarlyStopper:  
    def __init__(self, patience=5, min_delta=0):  
        self.patience = patience  
        self.min_delta = min_delta  
        self.counter = 0  
        self.min_validation_loss = np.inf  
  
    def early_stop(self, validation_loss):  
        if validation_loss < self.min_validation_loss:  
            self.min_validation_loss = validation_loss  
            self.counter = 0  
        elif validation_loss > (self.min_validation_loss + self.min_delta):  
            self.counter += 1  
        if self.counter >= self.patience:  
            return True  
        return False
```

Example 4

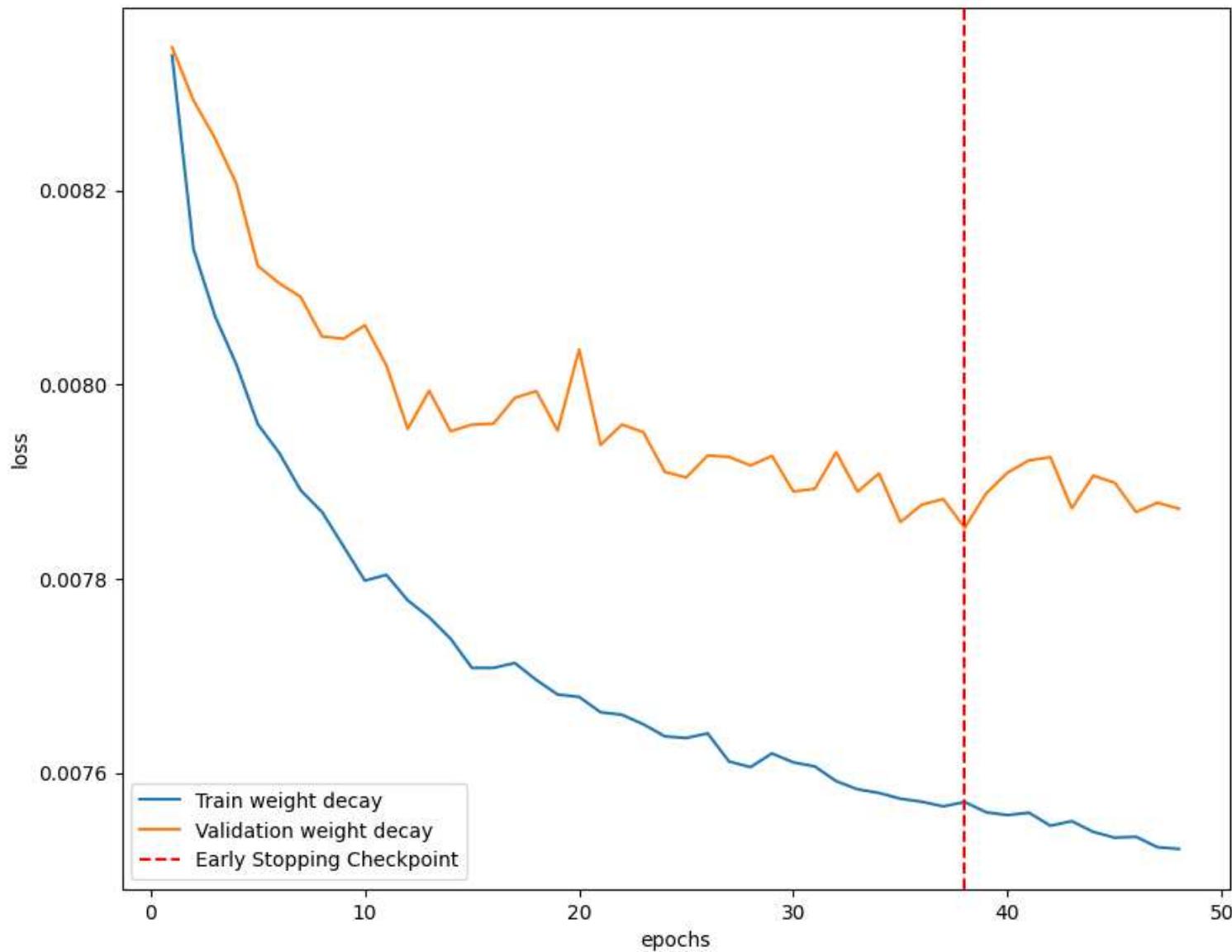
```
model = NeuralNetwork()  
loss_fn = nn.CrossEntropyLoss()  
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.001)  
early_stopper = EarlyStopper(patience=patience, min_delta)
```

```
if early_stopper.early_stop(test_loss):  
    print("Done!")  
    break
```

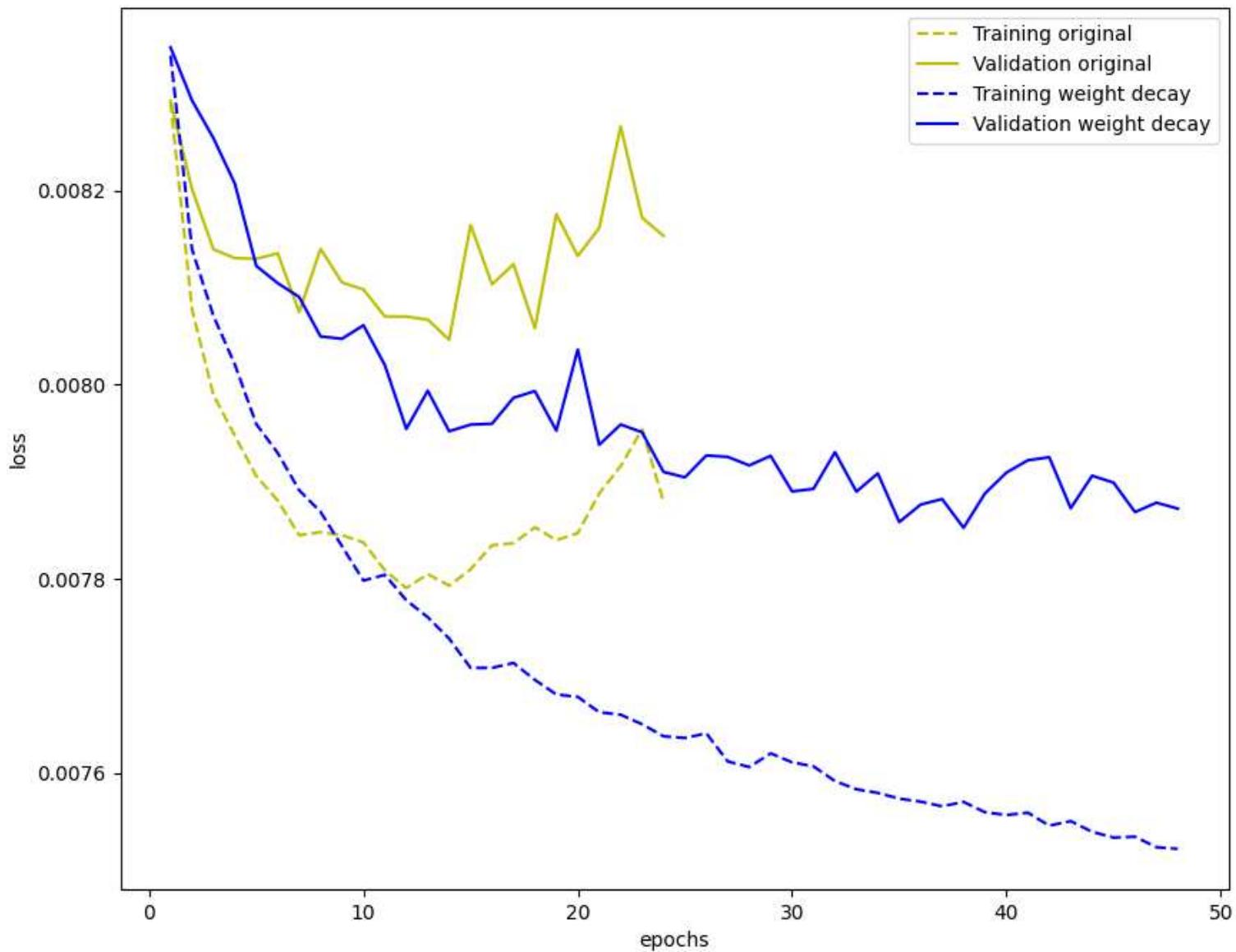
Example 4



Example 4



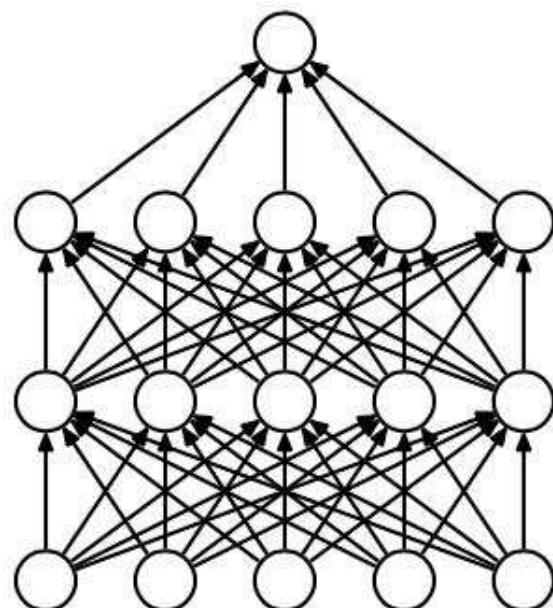
Example 4



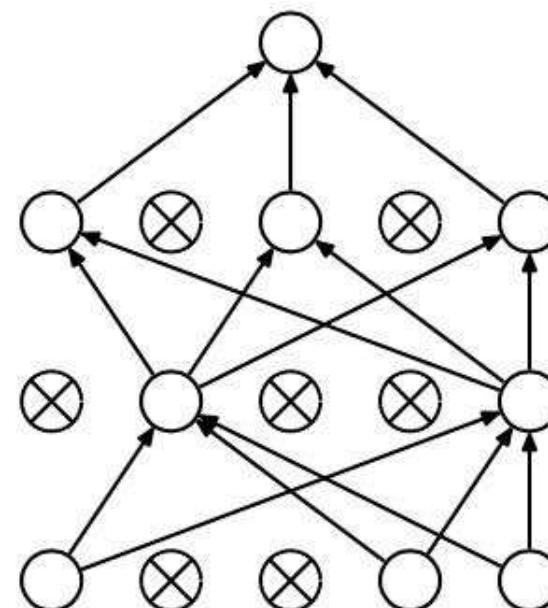
Dropouts

Overfitting can be avoided by training only a fraction of weights in each iteration. The key idea of ‘dropouts’ is to randomly drop neurons (along with their connections) from the networks during training.

This prevents neurons from co-adapting and thereby reduces overfitting.

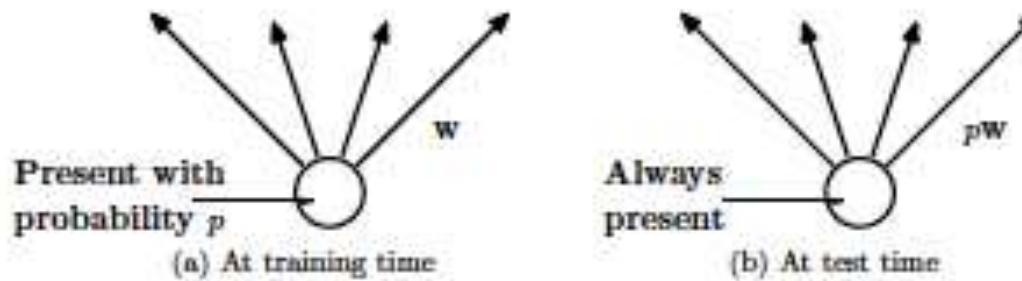


Fully connected network



Network with dropouts

Dropouts



At the training time, the units (neurons) are present with a probability p and presented to the next layer with weight W to the next layer.

This results in a scenario that at test time, the weights are always present, and presented to the network with weights multiplied by probability p . That is, The output at the test time is multiplied by $\frac{1}{p}$.

Applying dropouts result in a ‘thinned network’ that consists of only neurons that survived. This minimizes the redundancy in the network.

Dropout ratio

Dropout ratio is the fraction of neurons to be dropped out at one forward step.

Dropout ratio (p) has to be specified with `nn.Dropout()` in torch after activation function.

`nn.Dropout(p=0.2)`

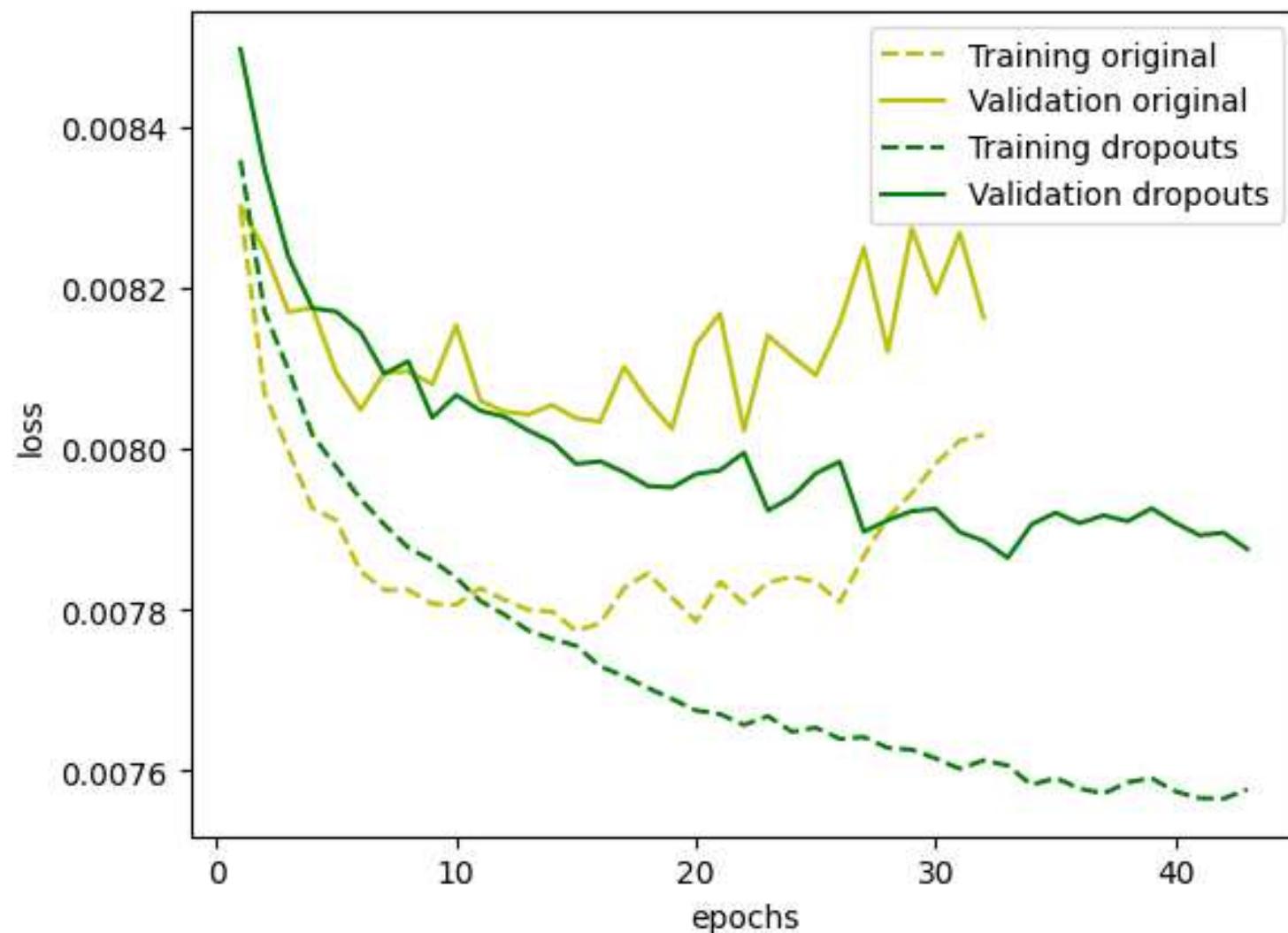
Example 5: dropouts

When training with dropouts, we train on a thinned network dropping out units in each mini-batch.

```
class NeuralNetwork_dropout(nn.Module):
    def __init__(self, hidden_size = 100, drop_out=0.5):
        super(NeuralNetwork_dropout, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(p=drop_out),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Example 5



Summary

- **Model selection**

- Holdout method
- Resampling methods
 - Random subsampling
 - K-fold cross-validation
 - LOO cross-validation
- Three-way data split

- **Methods to overcome overfitting**

- Early stopping
- Weight regularization
- Dropouts

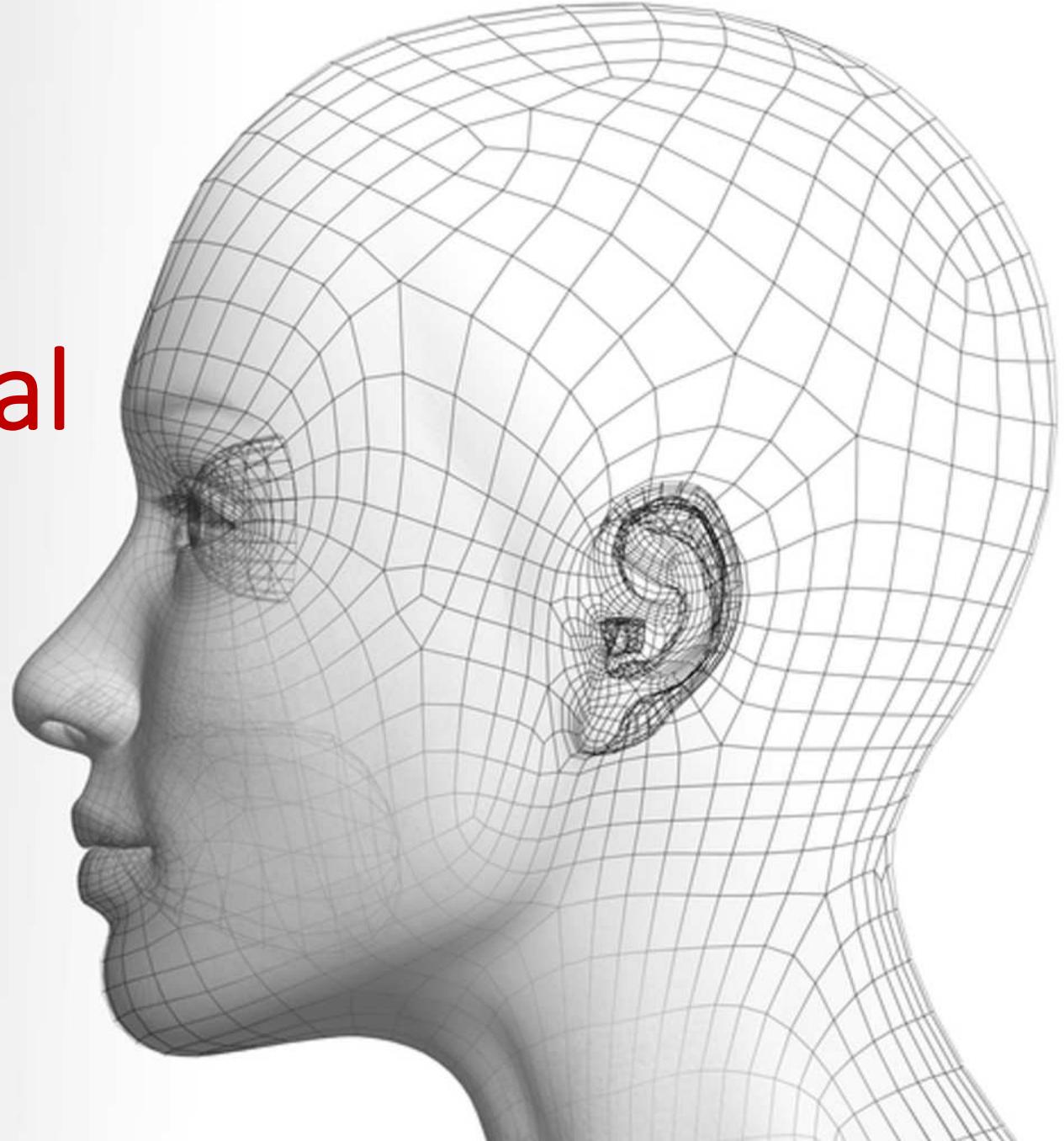
Convolutional Neural Networks I

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

<https://twitter.com/XingangP>



Instructors



Xingang Pan

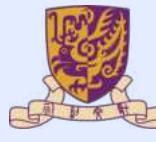
2012-2016



B.E.

Tsinghua University

2017-2021



PhD

The Chinese University of Hong Kong

Advisor : Xiaoou Tang

2021-2023



Post-doctoral Fellow

Department of Visual Computing and AI

Max Planck Institute for Informatics

2023-Now



Assistant Professor

School of Computer Science and Engineering

Nanyang Technological University

Research areas

- Generative models
- Image synthesis
- Computer vision
- Deep learning

Email: xingang.pan@ntu.edu.sg

Office: N4-02c-113

Office hour: By email appointment

Schedule

- **Week 7** – Convolutional Neural Network (CNN) I
- **Recess Week**
- **Week 8** – Convolutional Neural Network (CNN) II
- **Week 9** – Recurrent Neural Networks (RNN)
- **Week 10** – Attention
- **Week 11** – Autoencoders
- **Week 12** – Generative Adversarial Networks (GAN)
- **Week 13** – Revision and Selected Topics

Every week: Tutorial of previous week + Lecture of current week

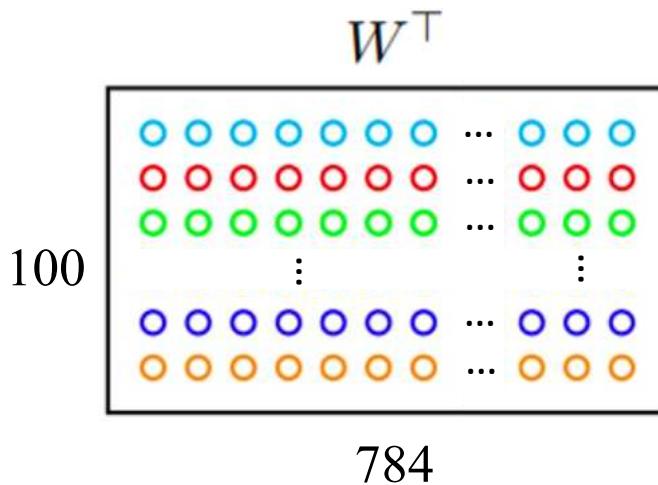
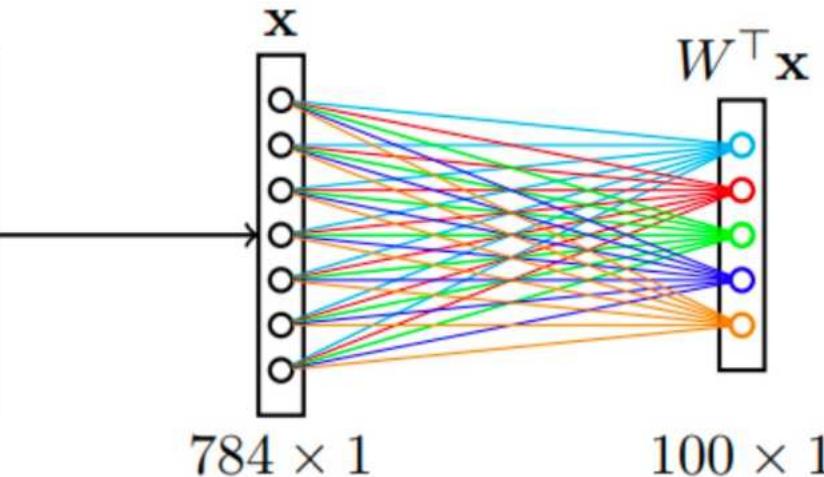
Outline

- Basic components in CNN
- Training a classifier
- Optimizers

Motivation



28×28

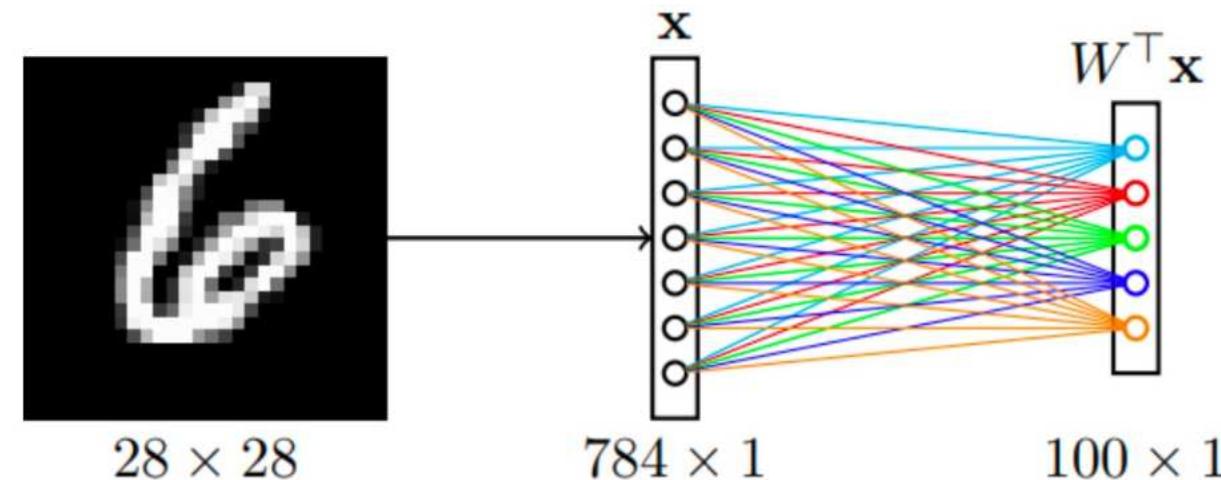
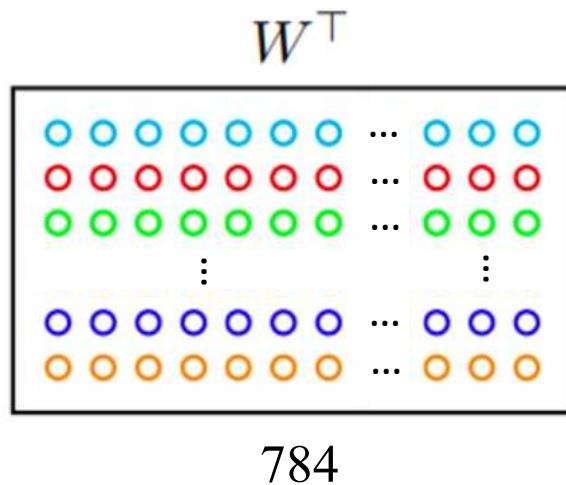


Let us consider the first layer of a MLP taking images as input. What are the problems with this architecture?

Motivation



28×28



Issues

- Too many parameters: $100 \times 784 + 100$.
 - What if images are $640 \times 480 \times 3$?
 - What if the first layer counts 1000 units?

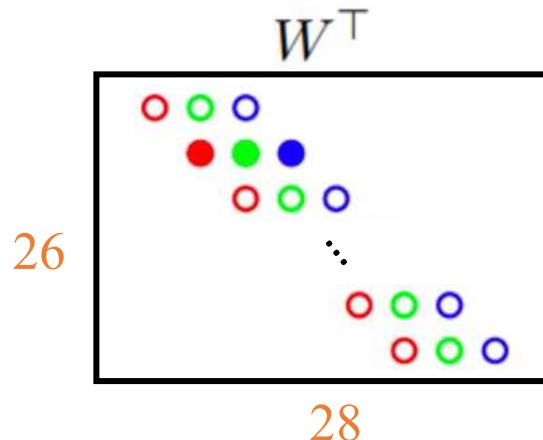
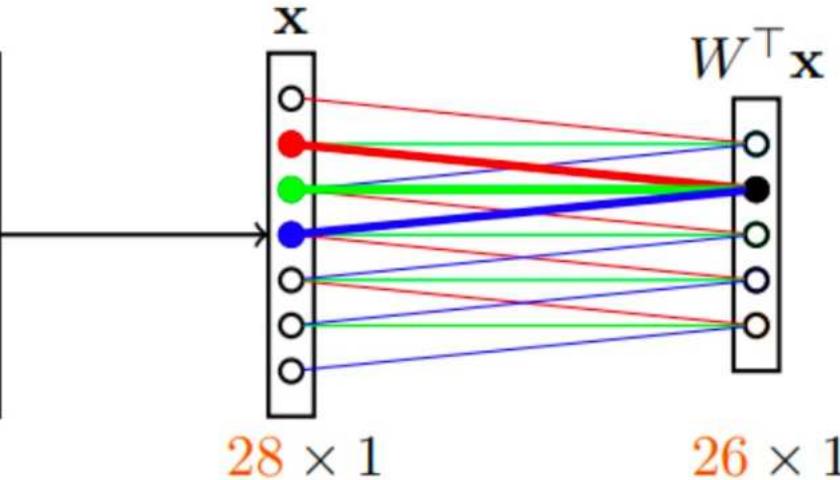
Fully connected networks where neurons at one level are connected to the neurons in other layers are not feasible for signals of large resolutions and are **computationally expensive** in feedforward and backpropagation computations.

- Spatial organization of the input is destroyed.
 - The network is not invariant/equivariant to transformations (e.g., translation).

Locally connected networks



28×28



Instead, let us only keep a **sparse** set of connections, where all weights having the same color are **shared**.

- The resulting operation can be seen as **shifting** the same weight triplet (**kernel**).
- The set of inputs seen by each unit is its receptive field.

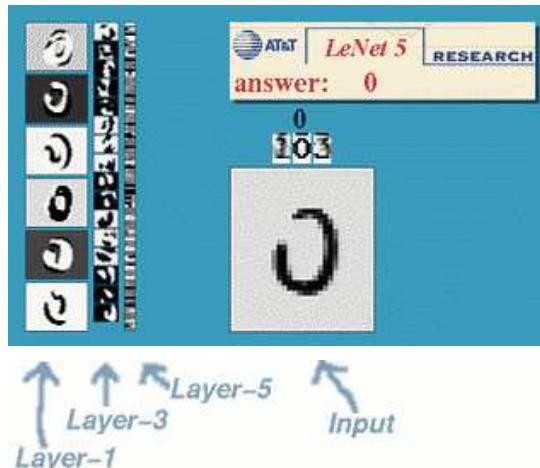
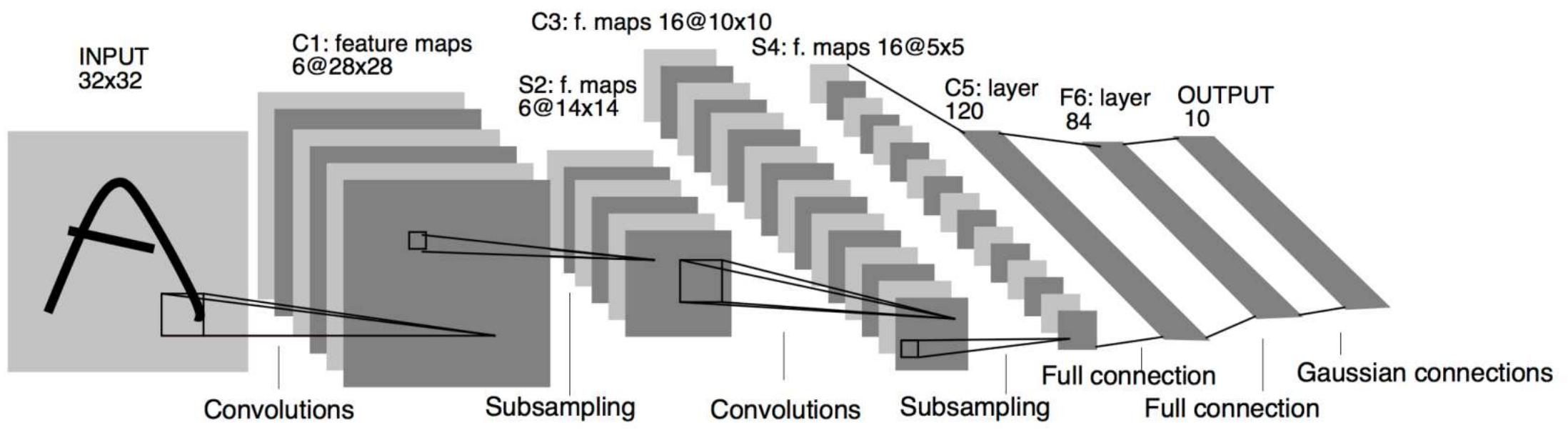
This is a **1D convolution**, which can be generalized to more dimensions.

Basic Components in CNN

Credits

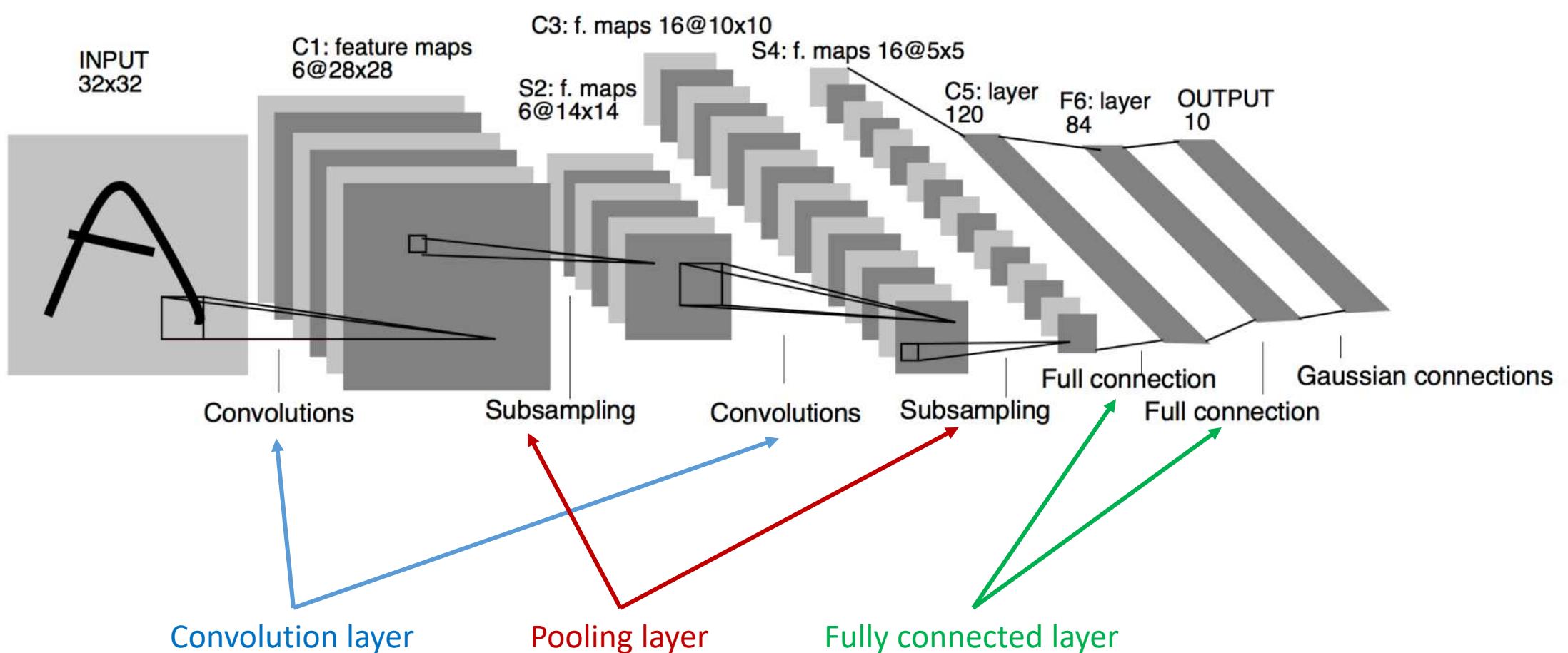
- CS 230: Deep Learning
 - <https://stanford.edu/~shervine/teaching/cs-230.html>
- CS231n: Convolutional Neural Networks for Visual Recognition
 - <http://cs231n.stanford.edu/syllabus.html>

An example of convolutional network: LeNet 5



LeCun et al., 1998
Turing Award (2018)

An example of convolutional network: LeNet 5

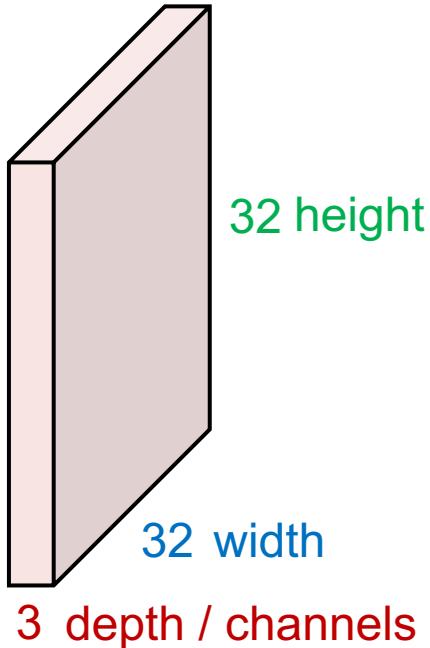


As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

Common layer arrangement: Conv → pool → Conv → pool → fully connected → fully connected → output

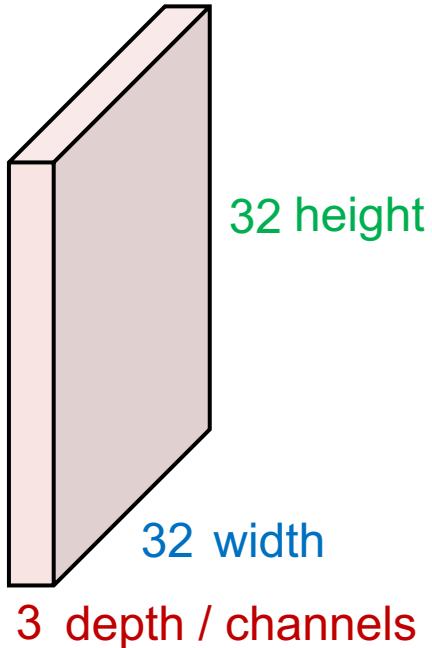
Convolution layer

3x32x32 image

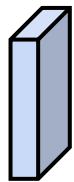


Convolution layer

3x32x32 image



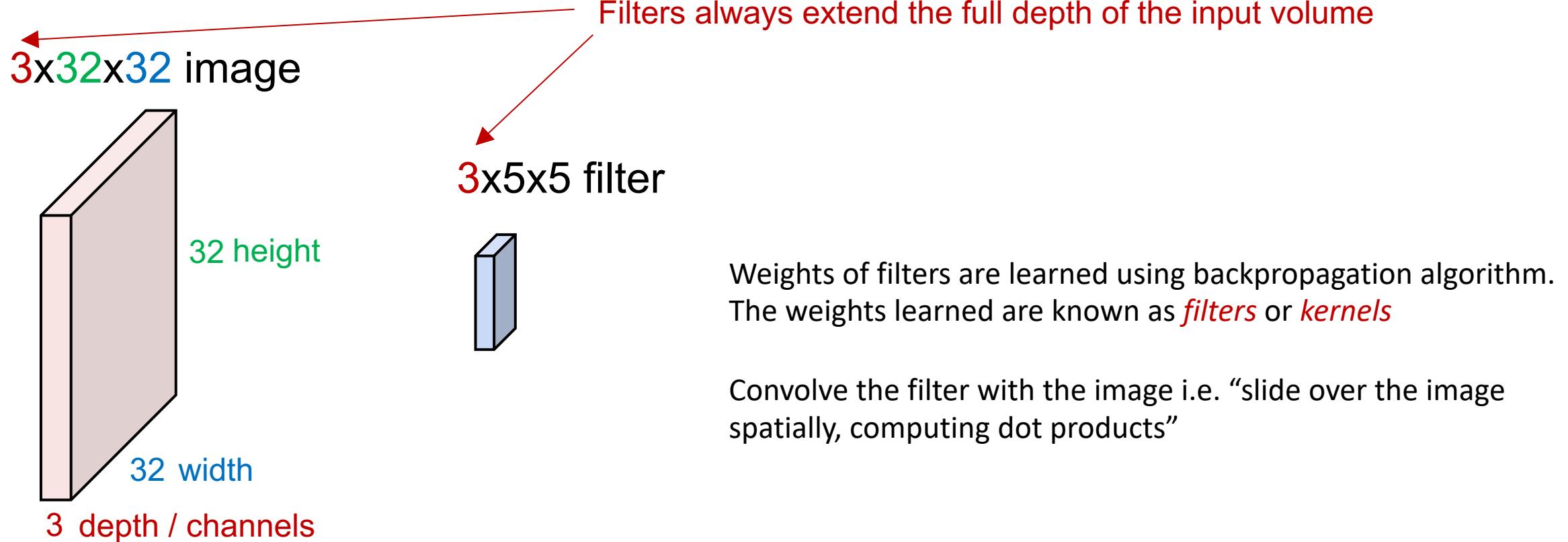
3x5x5 filter



Weights of filters are learned using backpropagation algorithm.
The weights learned are known as *filters* or *kernels*

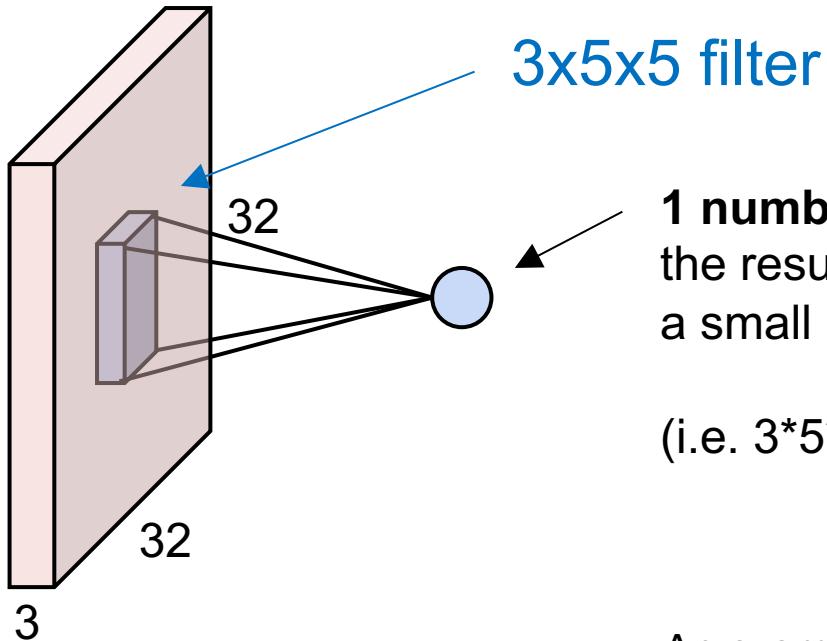
Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution layer



Convolution layer

3x32x32 image



1 number:

the result of taking a dot product between the filter and a small 3x5x5 chunk of the image

(i.e. $3 \times 5 \times 5 = 75$ -dimensional dot product + bias)

An example of convolving
a 1x5x5 image with a
1x3x3 filter

1	0	1
0	1	0
1	0	1

Filter (weights or kernel)

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

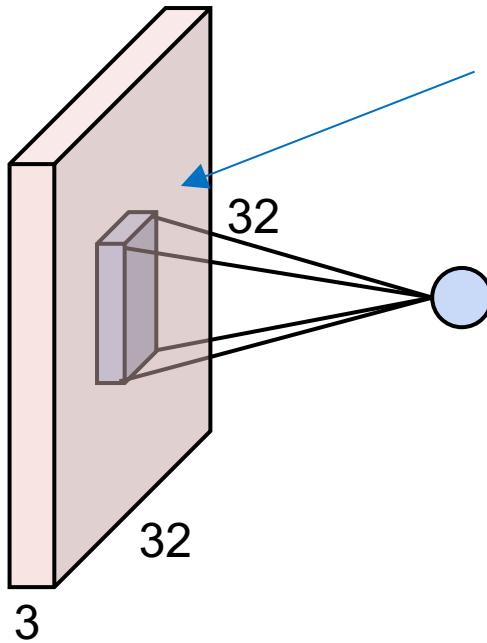
Image

4		

Convolved
Feature

Convolution layer

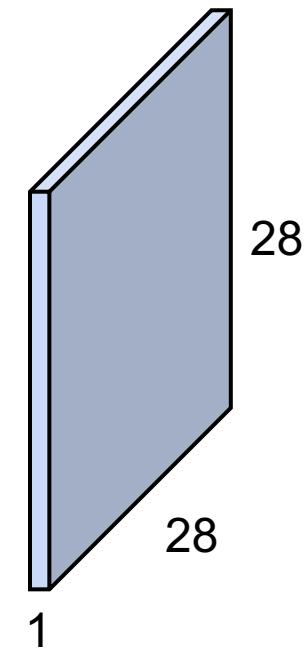
3x32x32 image



3x5x5 filter

convolve (slide) over all
spatial locations

1x28x28
activation/feature map

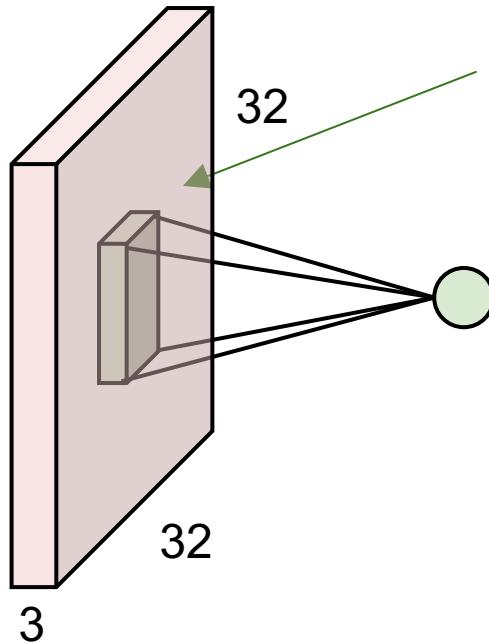


The activations are obtained by convolving the filters (weights) with the input activations. The output activation produced by a particular filter is known as a *activation map / feature map*

Convolution layer

Consider a second, green filter

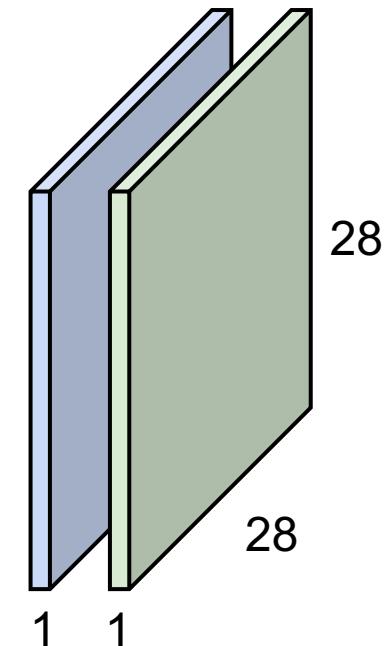
3x32x32 image



3x5x5 filter

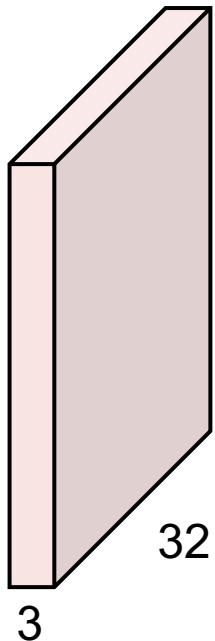
convolve (slide) over all
spatial locations

Two 1x28x28
activation/feature maps



Convolution layer

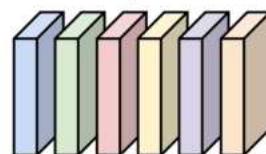
3x32x32 image



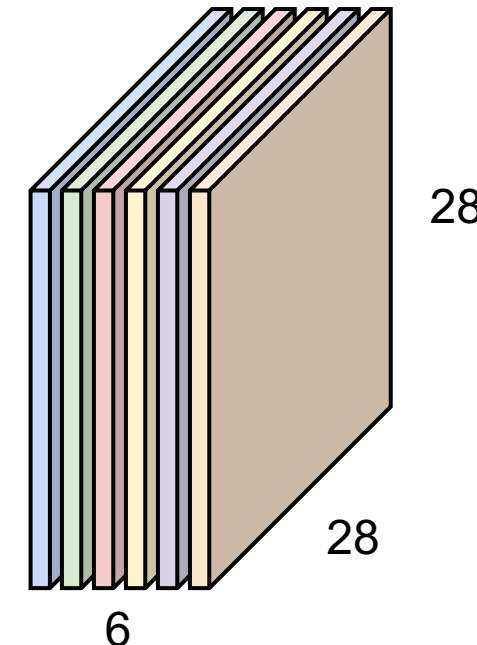
If we had **six** 3x5x5 filters,
we'll get **six** separate
activation maps:

convolution layer

6x3x5x5
filters



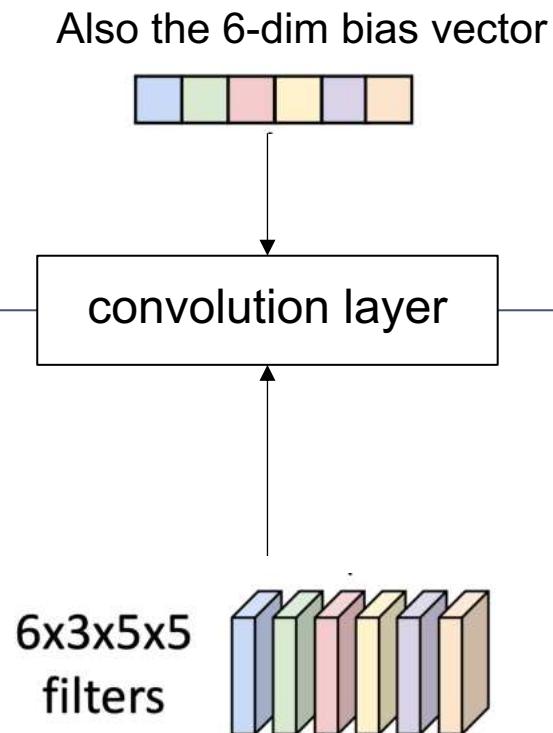
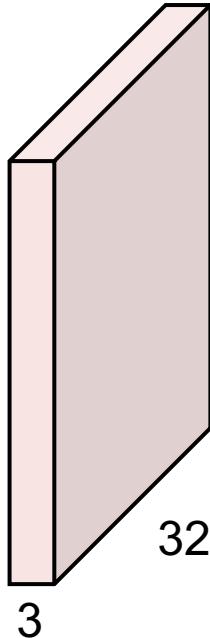
Six 1x28x28
activation/feature maps



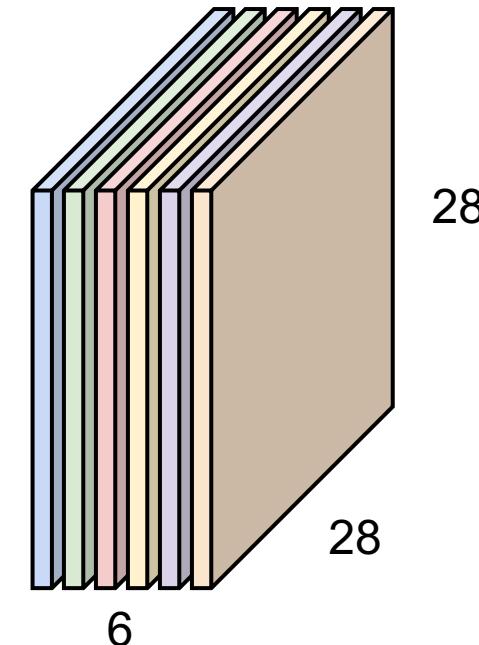
We stack these up to
get a “new image” of
size 6x28x28

Convolution layer

3x32x32 image

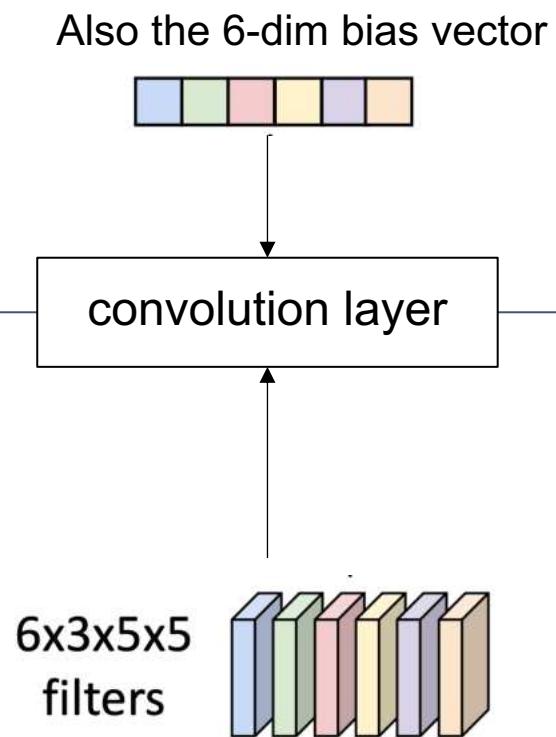
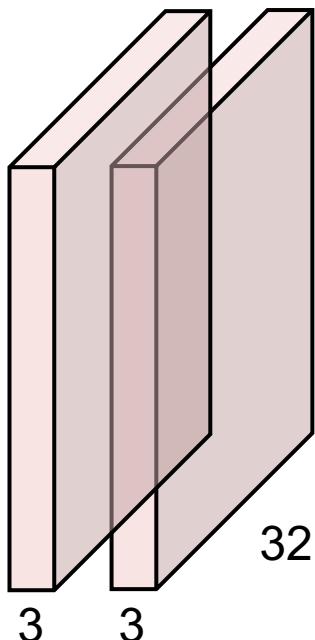


6x28x28
activation/feature maps

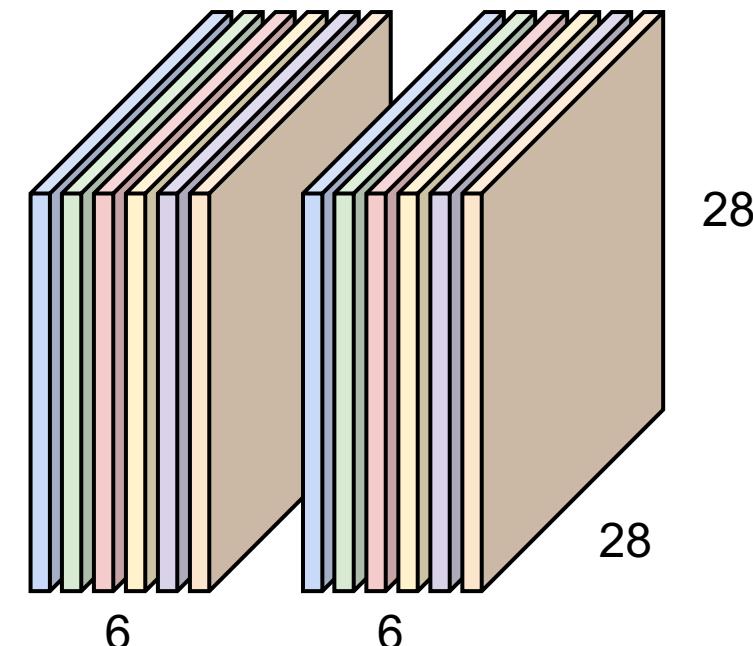


Convolution layer

2x3x32x32 batch of images

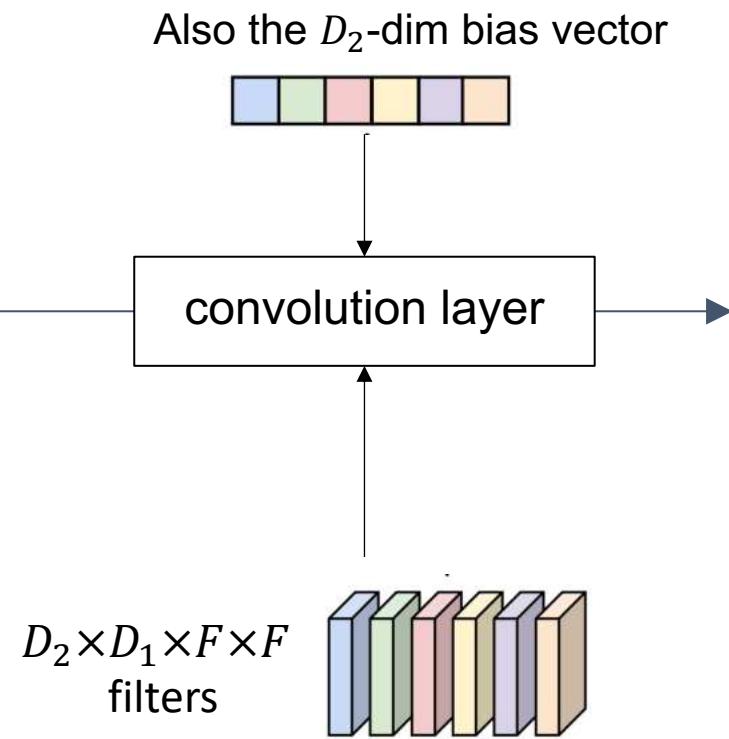
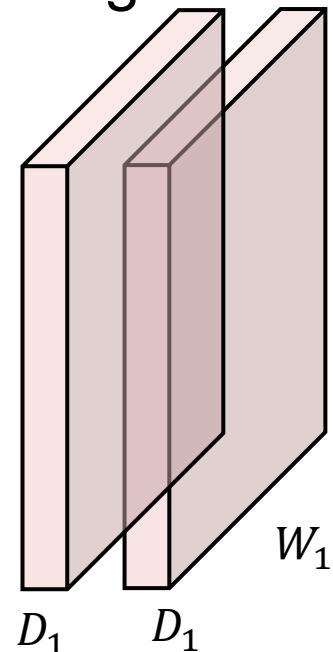


2x6x28x28 batch of activation/feature maps

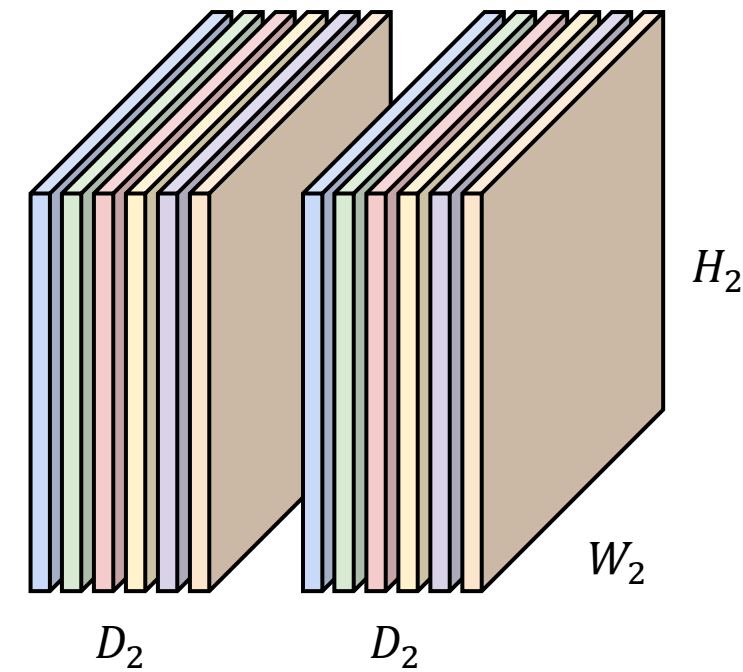


Convolution layer

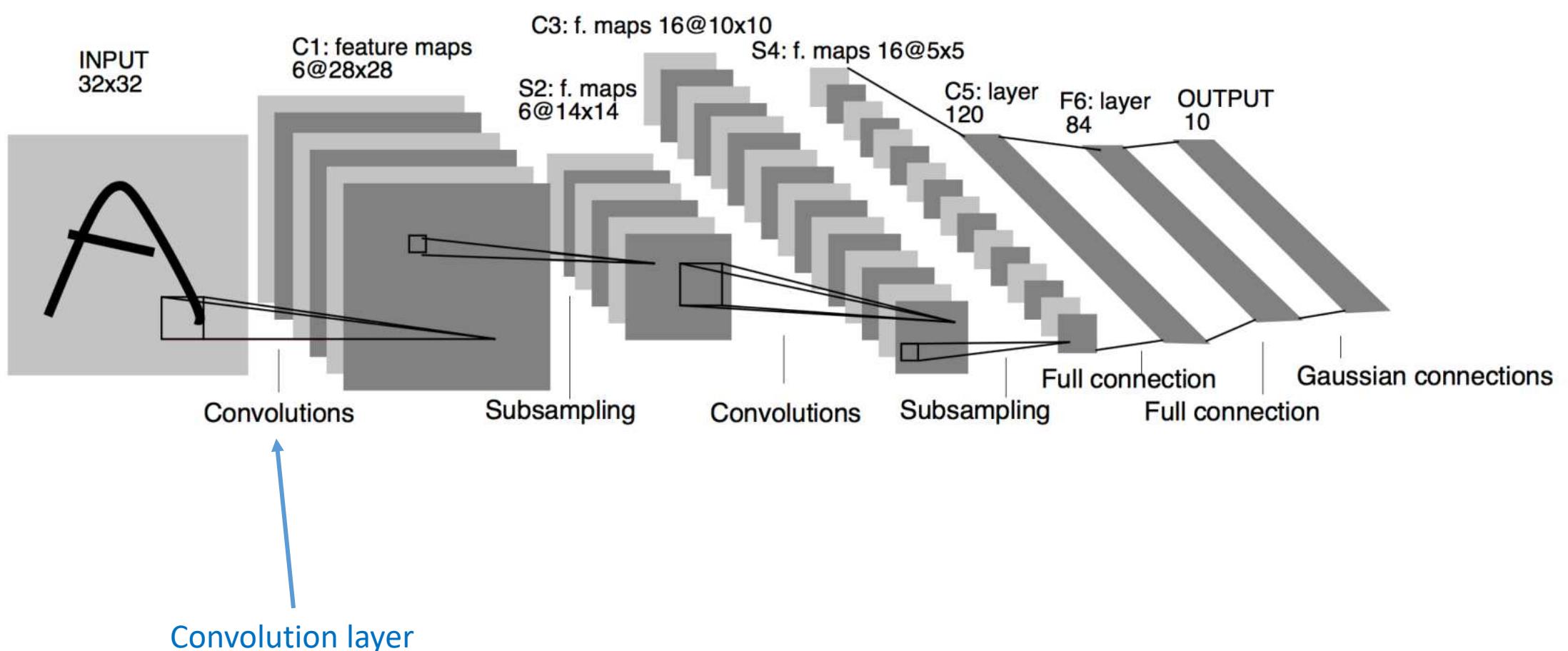
$N \times D_1 \times H_1 \times W_1$ batch of images



$N \times D_2 \times H_2 \times W_2$ batch of activation/feature maps



An example of convolutional network: LeNet 5

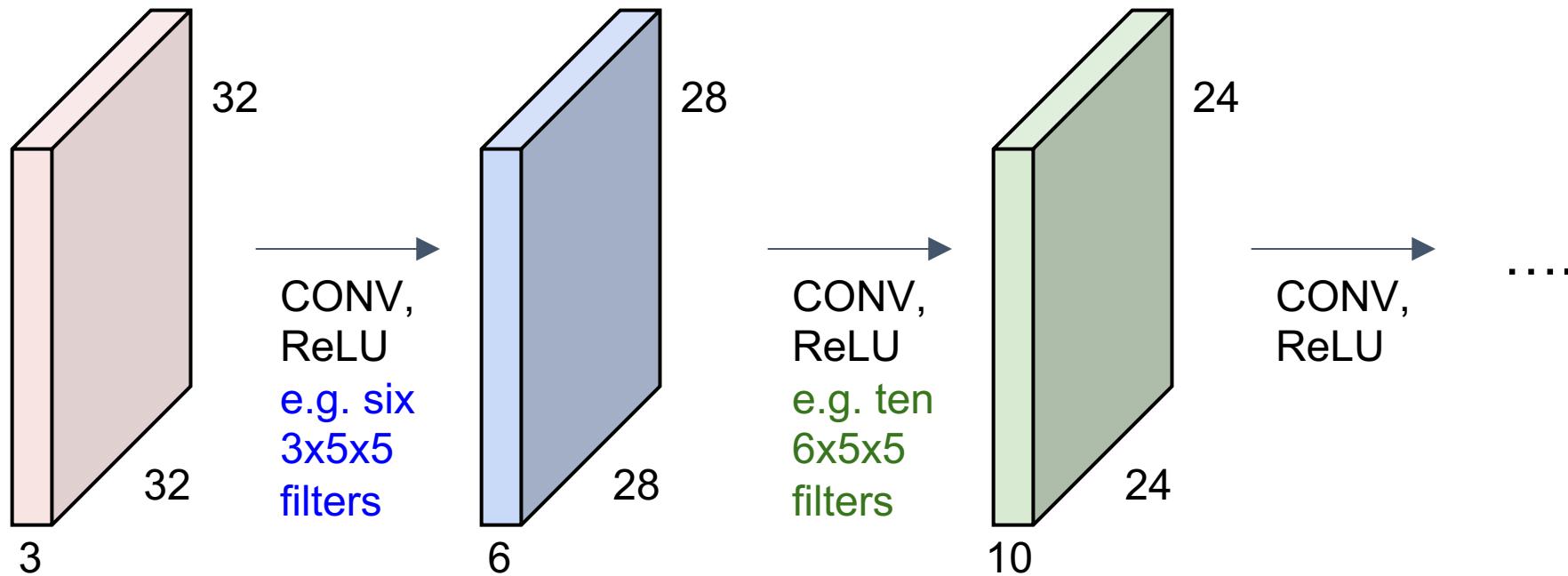


Back to this example, 6@28x28 means that we have 6 feature maps of size 28*28

You can imagine that the first convolutional layer uses 6 filters and each filter is of size 5 x 5 (how do we know that? We will discuss that later)

Convolution layer

CNN is a sequence of convolutional layers, interspersed with activation functions



Convolution layer

Consider a kernel $\mathbf{w} = \{w(l, m)\}$, which has a size of $L \times M, L = 2a + 1, M = 2b + 1$

Synaptic input at location $p = (i, j)$ of the first hidden layer due to a kernel is given by

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i + l, j + m)w(l, m) + bias$$

For instance, given $L = 3, M = 3$

$$\begin{aligned} u(i, j) = & x(i - 1, j - 1)w(-1, -1) + x(i - 1, j)w(-1, 0) + \dots \\ & + x(i, j)w(0, 0) + x(i + 1, j + 1)w(1, 1) + bias \end{aligned}$$

Convolution layer

The output of the neuron at (i, j) of the convolution layer

$$y(i, j) = f(u(i, j))$$

where f is an activation function. For deep CNN, we typically use ReLU, $f(x) = \max(0, x)$.

Note that one weight tensor $\mathbf{w}_k = \{w_k(l, m)\}$ or kernel (filter) creates one feature map:

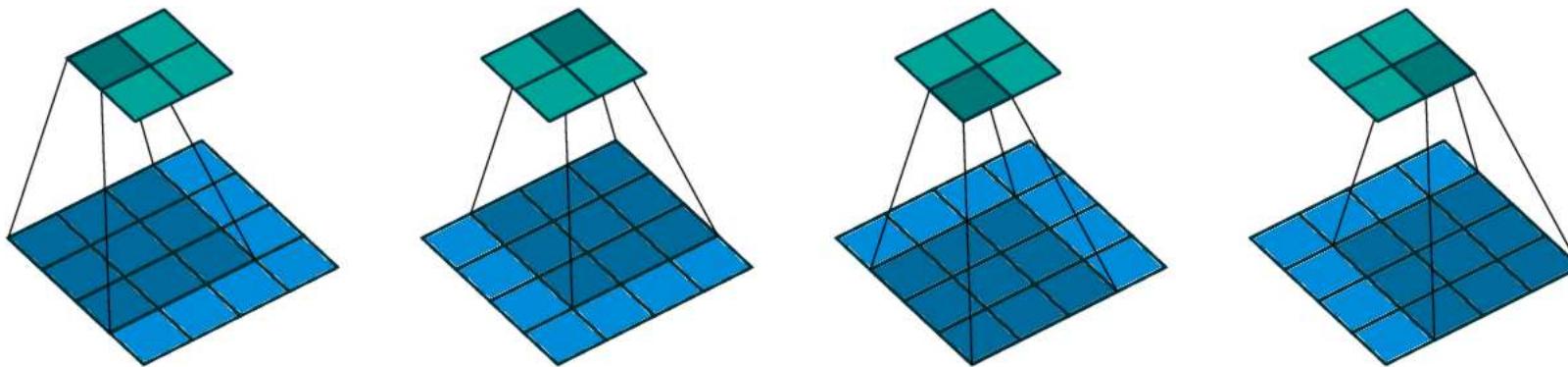
$$\mathbf{y}_k = \{y_k(i, j)\}$$

If there are K weight vectors $(\mathbf{w}_k)_{k=1}^K$, the convolutional layer is formed by K feature maps

$$\mathbf{y} = (\mathbf{y}_k)_{k=1}^K$$

Convolution layer

Convolution by doing a sliding window



As a guiding example, let us consider the convolution of single-channel tensors $\mathbf{x} \in \mathbb{R}^{4 \times 4}$ and $\mathbf{w} \in \mathbb{R}^{3 \times 3}$:

$$\mathbf{w} * \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i + l, j + m) w(l, m) + bias$$

$$u(1,1) = 4 \times 1 + 5 \times 4 + 8 \times 1 + 1 \times 1 + 8 \times 4 + 8 \times 3 + 3 \times 3 + 6 \times 3 + 6 \times 1 = 122$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i + l, j + m) w(l, m) + bias$$

$$u(1,1) = 4 \times 1 + 5 \times 4 + 8 \times 1 + 1 \times 1 + 8 \times 4 + 8 \times 3 + 3 \times 3 + 6 \times 3 + 6 \times 1 = 122$$

$$u(1,2) = 5 \times 1 + 8 \times 4 + 7 \times 1 + 8 \times 1 + 8 \times 4 + 8 \times 3 + 6 \times 3 + 6 \times 3 + 4 \times 1 = 148$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i + l, j + m) w(l, m) + bias$$

$$u(1,1) = 4 \times 1 + 5 \times 4 + 8 \times 1 + 1 \times 1 + 8 \times 4 + 8 \times 3 + 3 \times 3 + 6 \times 3 + 6 \times 1 = 122$$

$$u(1,2) = 5 \times 1 + 8 \times 4 + 7 \times 1 + 8 \times 1 + 8 \times 4 + 8 \times 3 + 6 \times 3 + 6 \times 3 + 4 \times 1 = 148$$

$$u(2,1) = 1 \times 1 + 8 \times 4 + 8 \times 1 + 3 \times 1 + 6 \times 4 + 6 \times 3 + 6 \times 3 + 5 \times 3 + 7 \times 1 = 126$$

Convolution layer

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

$$u(i,j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i+l, j+m) w(l, m) + bias$$

$$u(1,1) = 4 \times 1 + 5 \times 4 + 8 \times 1 + 1 \times 1 + 8 \times 4 + 8 \times 3 + 3 \times 3 + 6 \times 3 + 6 \times 1 = 122$$

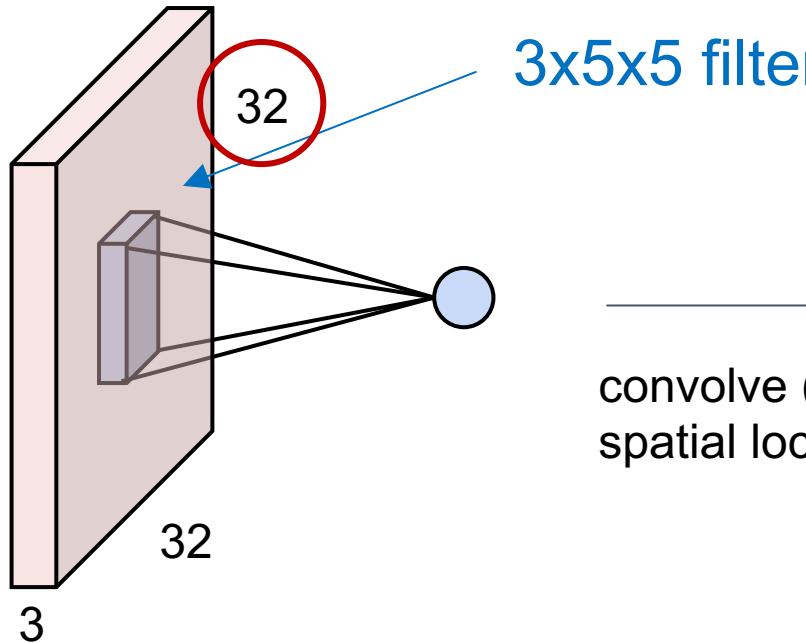
$$u(1,2) = 5 \times 1 + 8 \times 4 + 7 \times 1 + 8 \times 1 + 8 \times 4 + 8 \times 3 + 6 \times 3 + 6 \times 3 + 4 \times 1 = 148$$

$$u(2,1) = 1 \times 1 + 8 \times 4 + 8 \times 1 + 3 \times 1 + 6 \times 4 + 6 \times 3 + 6 \times 3 + 5 \times 3 + 7 \times 1 = 126$$

$$u(2,2) = 8 \times 1 + 8 \times 4 + 8 \times 1 + 6 \times 1 + 6 \times 4 + 4 \times 3 + 5 \times 3 + 7 \times 3 + 8 \times 1 = 134$$

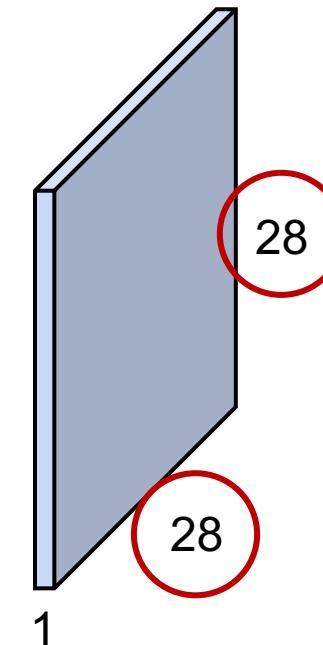
Convolution layer – spatial dimensions

3x32x32 image



convolve (slide) over all
spatial locations

1x28x28
activation/feature map

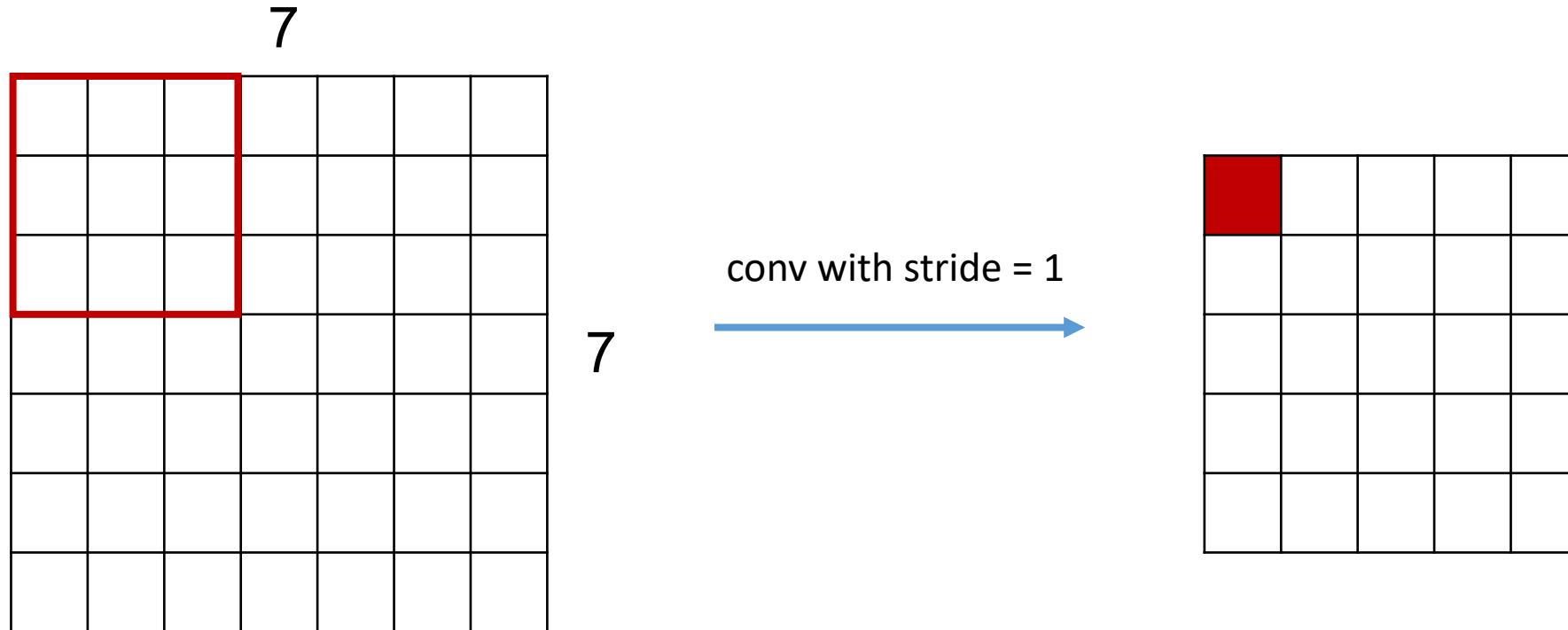


Why does the feature map
has a size of 28x28?

Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

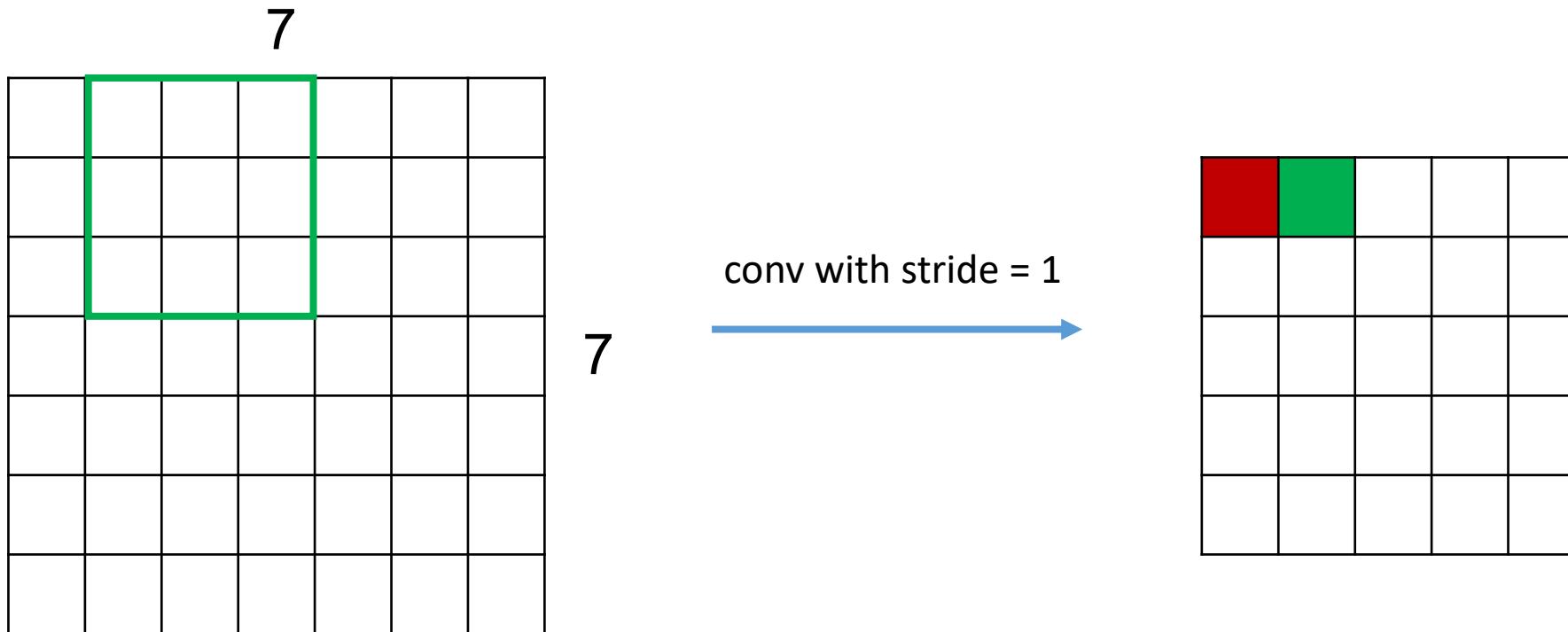
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

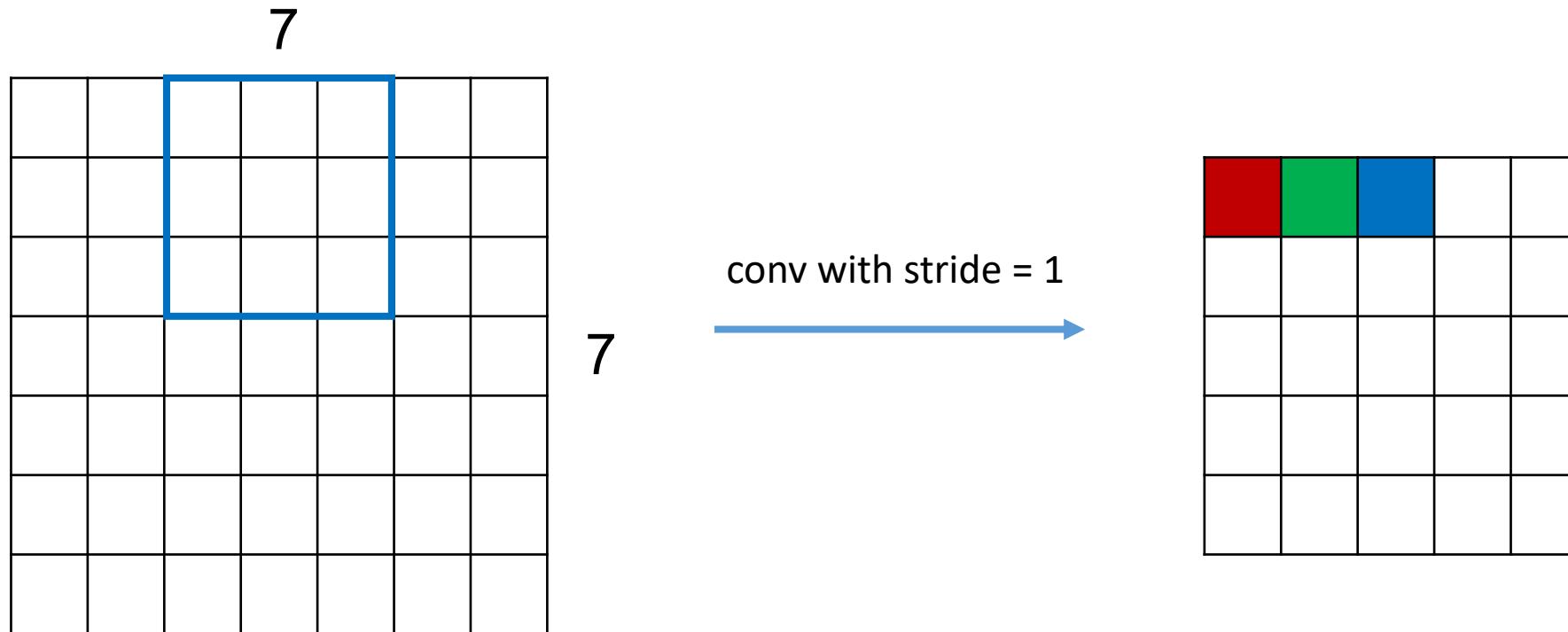
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

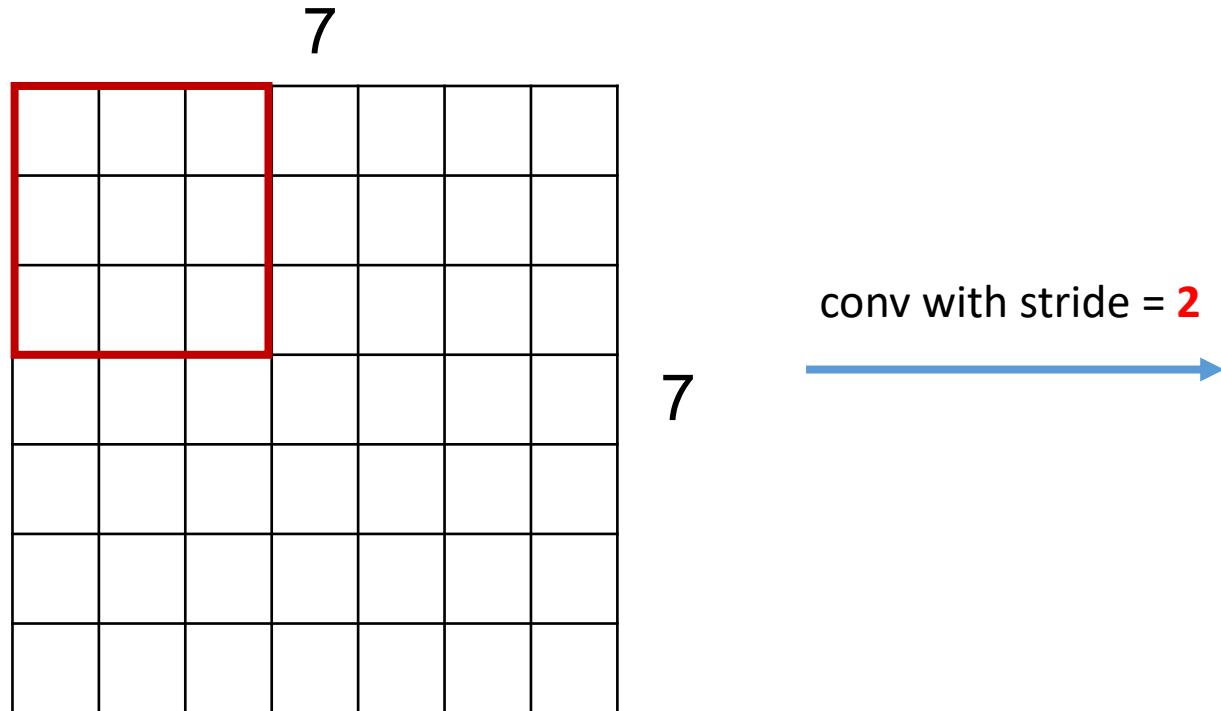
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

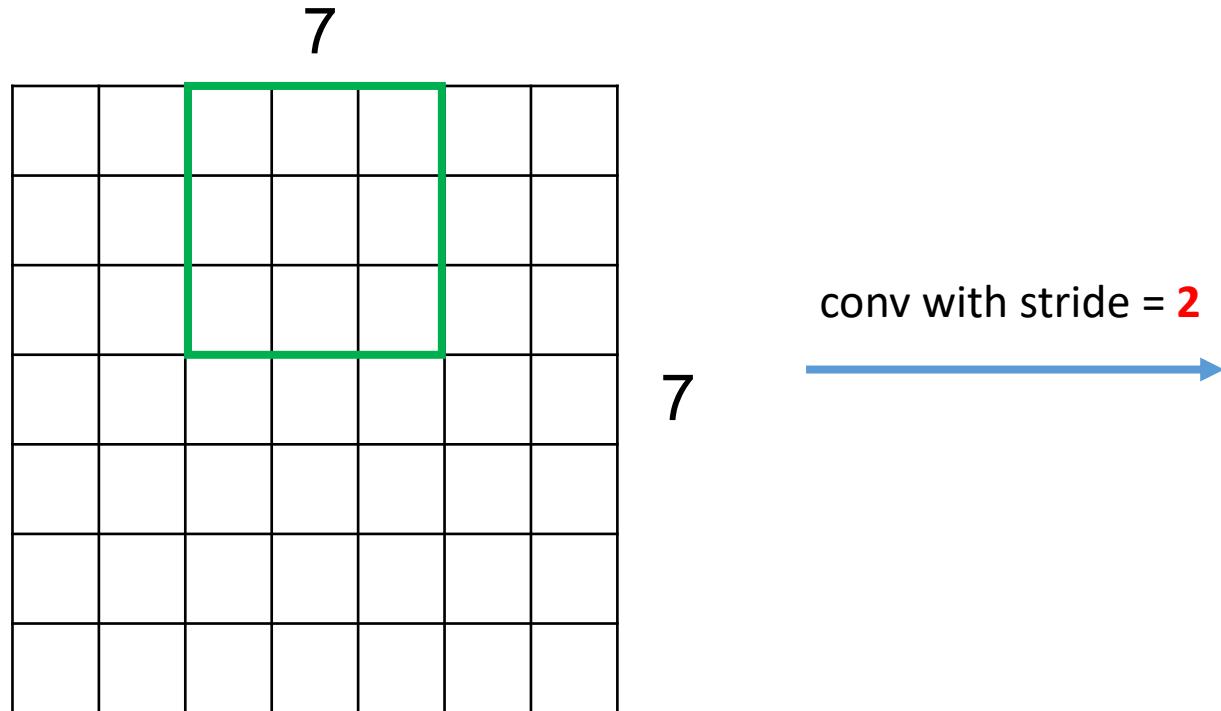
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

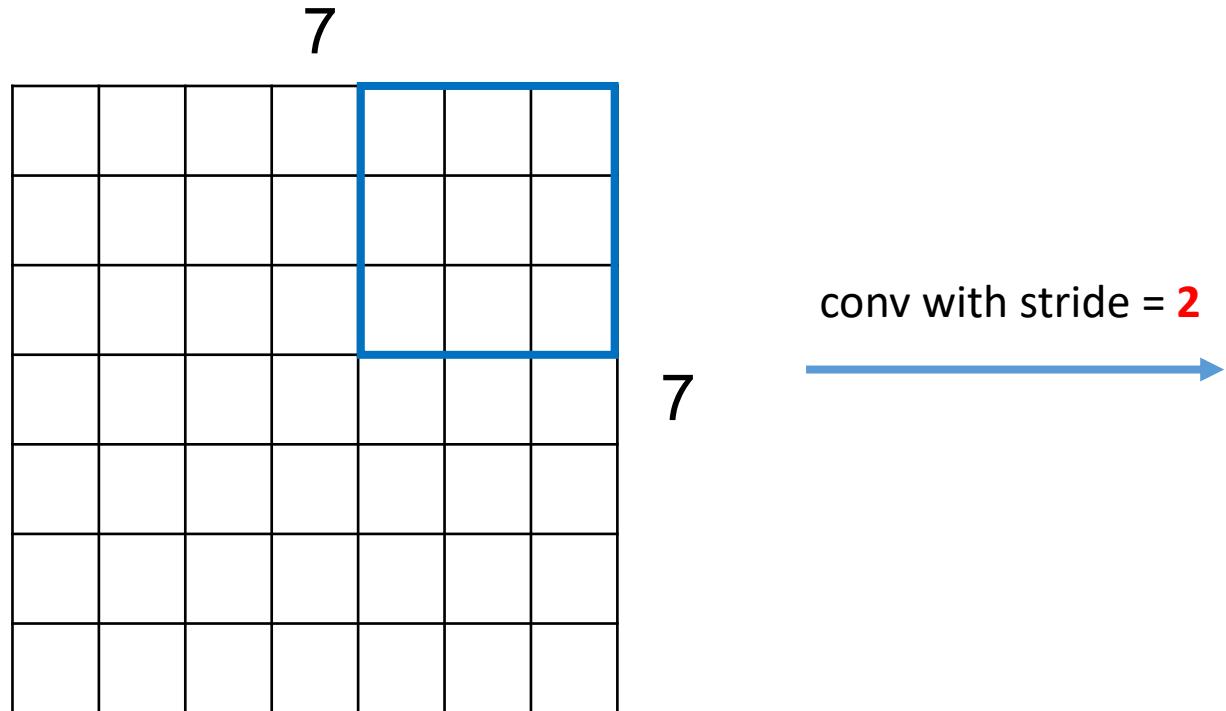
Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



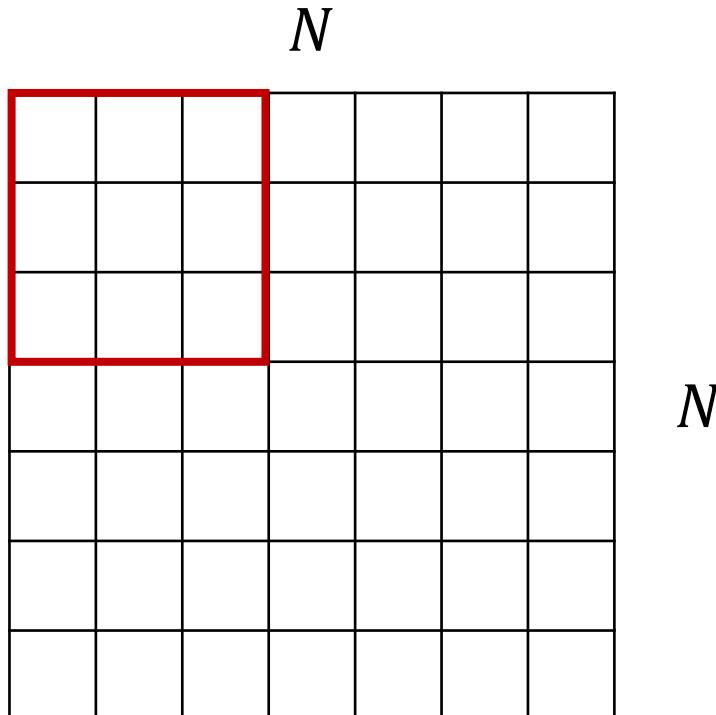
Convolution layer – spatial dimensions

7x7 input (spatially), assume 3x3 filter

Stride is the factor with which the output is subsampled. That is, *distance between adjacent centers of the kernel*.



Convolution layer – spatial dimensions



$N \times N$ input (spatially), assume $F \times F$ filter, and S stride

$$\text{Output size} = \frac{N-F}{S} + 1$$

e.g.

$$N = 7, F = 3$$

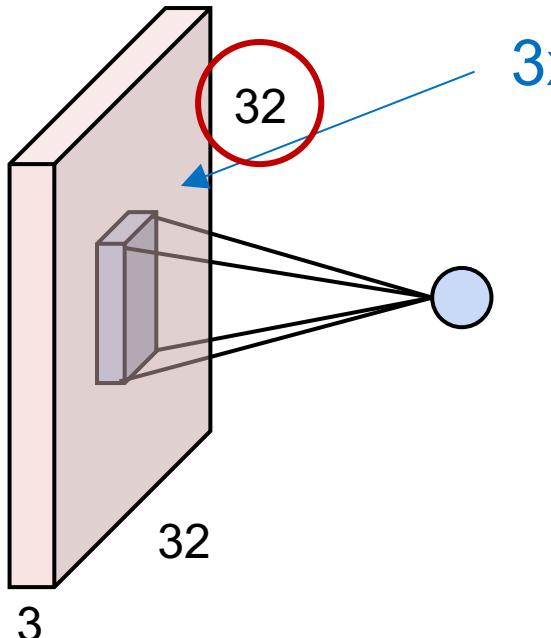
$$\text{stride 1} \Rightarrow (7 - 3)/1 + 1 = 5$$

$$\text{stride 2} \Rightarrow (7 - 3)/2 + 1 = 3$$

$$\text{stride 3} \Rightarrow (7 - 3)/3 + 1 = 2.33 : \backslash$$

Convolution layer – spatial dimensions

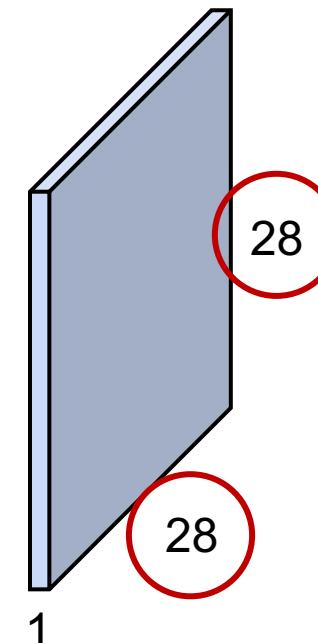
3x32x32 image



3x5x5 filter

convolve (slide) over all
spatial locations

1x28x28
activation/feature map



Why does the feature map
has a size of 28x28?

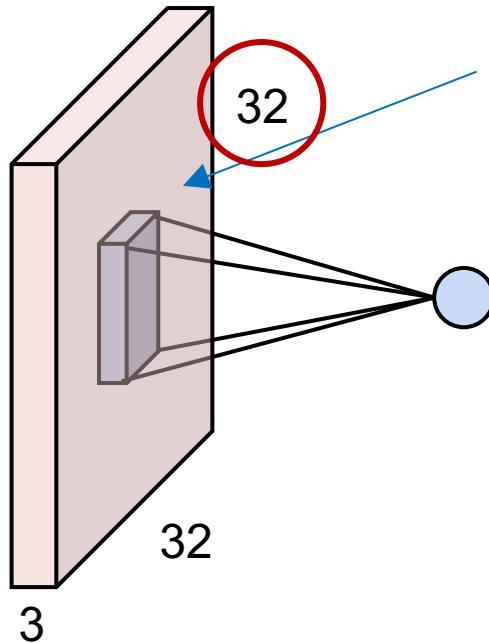
$$\text{Output size} = \frac{N-F}{S} + 1$$

$$N = 32, F = 5$$

$$\text{stride } 1 \Rightarrow (32 - 5)/1 + 1 = 28$$

Convolution layer – spatial dimensions

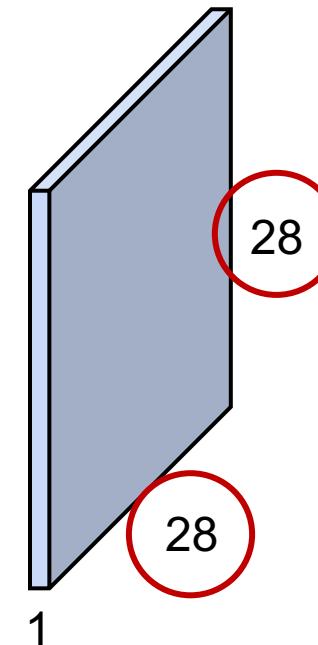
3x32x32 image



3x5x5 filter

convolve (slide) over all
spatial locations

1x28x28
activation/feature map



The valid feature map is smaller than the input after convolution.

Without zero-padding, the width of the representation shrinks by the $F - 1$ at each layer
To avoid shrinking the spatial extent of the network rapidly, small filters have to be used

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In **PyTorch**, by default, we pad top, bottom, left, right with zeros (this can be customized).

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output shape?

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In **PyTorch**, by default, we pad top, bottom, left, right with zeros (this can be customized, e.g., 'constant', 'reflect', and 'replicate').

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output shape?

Recall that without padding, output size = $\frac{N-F}{S} + 1$

With padding, output size = $\frac{N-F+2P}{S} + 1$

e.g.

$$N = 7, F = 3$$

$$\text{stride 1} \Rightarrow (7 - 3 + 2(1))/1 + 1 = 7$$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In **PyTorch**, by default, we pad top, bottom, left, right with zeros (this can be customized).

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output shape?

7×7 output!

In general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F - 1)/2$. (**will preserve size spatially**)

e.g. $F = 3 \Rightarrow$ zero pad with 1

$F = 5 \Rightarrow$ zero pad with 2

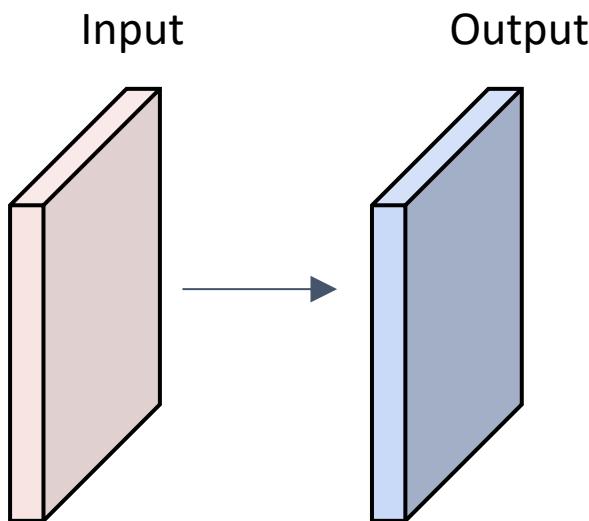
$F = 7 \Rightarrow$ zero pad with 3

Example

Input volume: $3 \times 32 \times 32$

Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

Output volume size: ?



Example

Input volume: $3 \times 32 \times 32$

Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

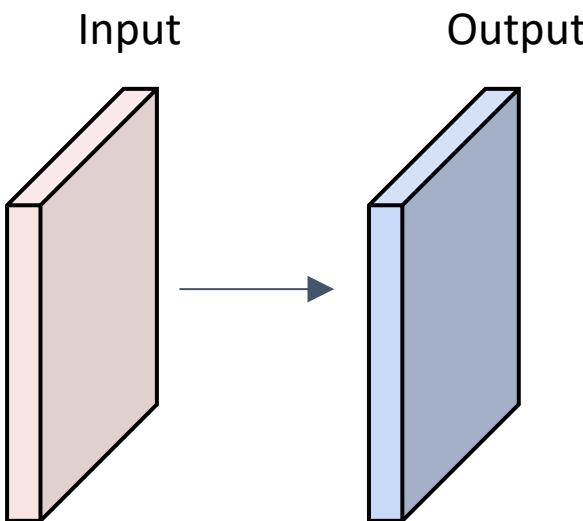
Output volume size: ?

Output size =

$$\frac{N - F + 2P}{S} + 1$$

$$\frac{32 - 5 + 2(2)}{1} + 1 = 32 \text{ spatially}$$

The output volume size is $10 \times 32 \times 32$

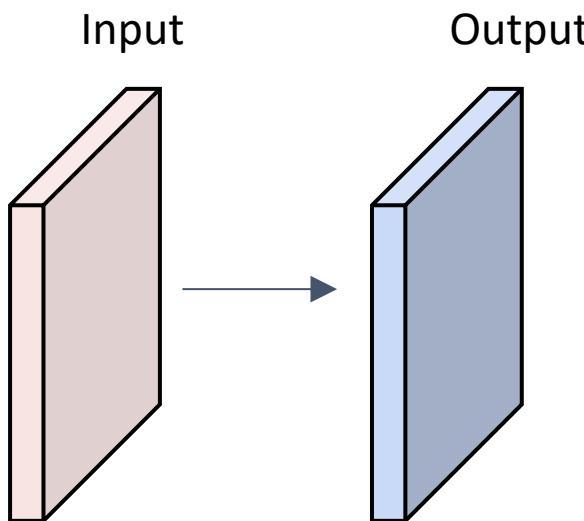


Example

Input volume: $3 \times 32 \times 32$

Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

Number of parameters in this layer: ?

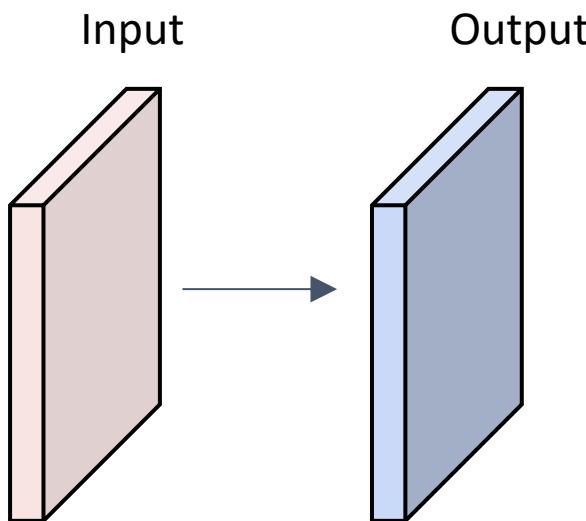


Example

Input volume: $3 \times 32 \times 32$

Ten $3 \times 5 \times 5$ filters with stride 1, pad 2

Number of parameters in this layer: ?

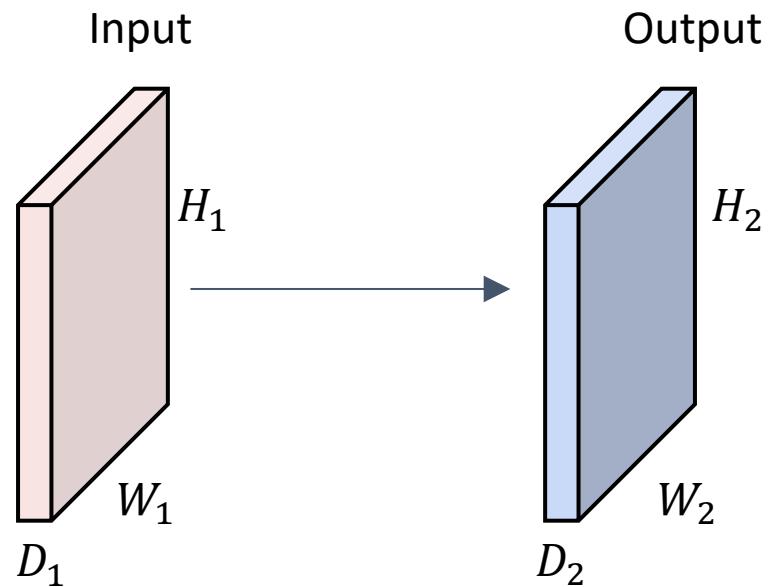


Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)
=> $76 \times 10 = 760$

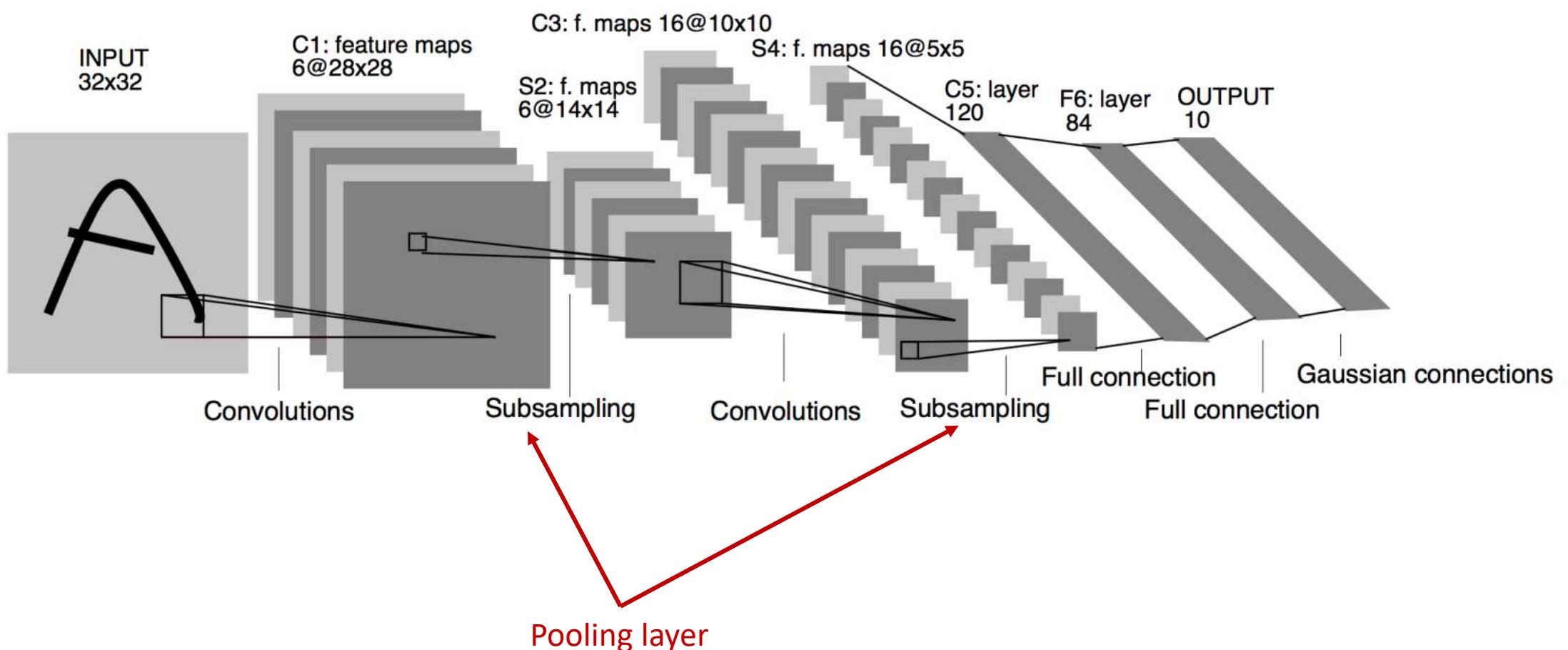
Convolution layer - summary

A convolution layer

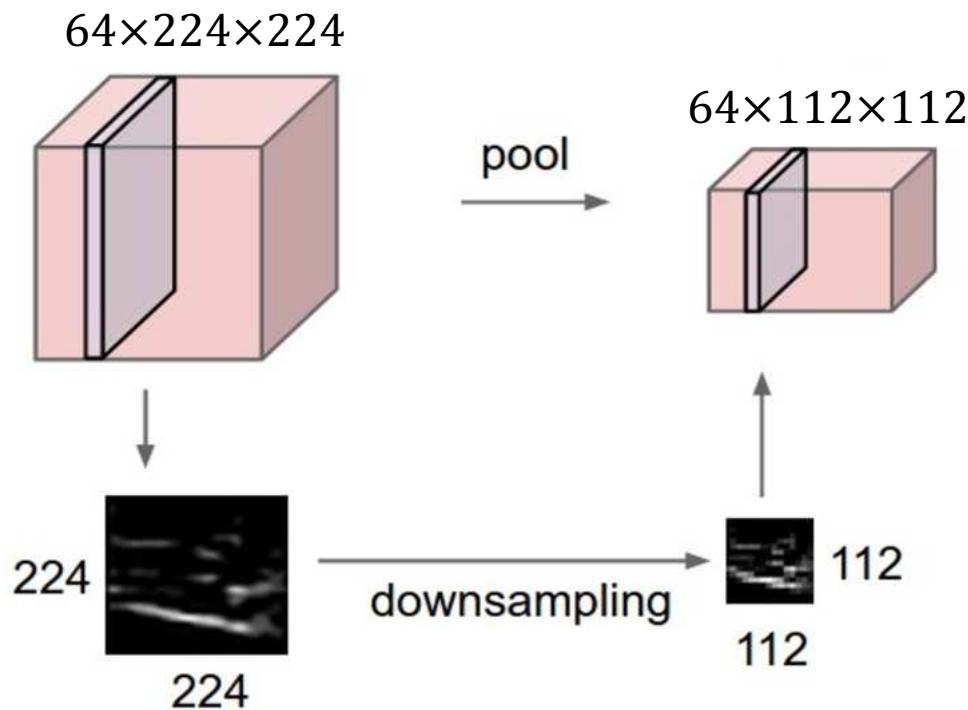
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
 - Number of filters K
 - Their spatial extent F
 - The stride S
 - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $W_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d -th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias



An example of convolutional network: LeNet 5



Pooling layer

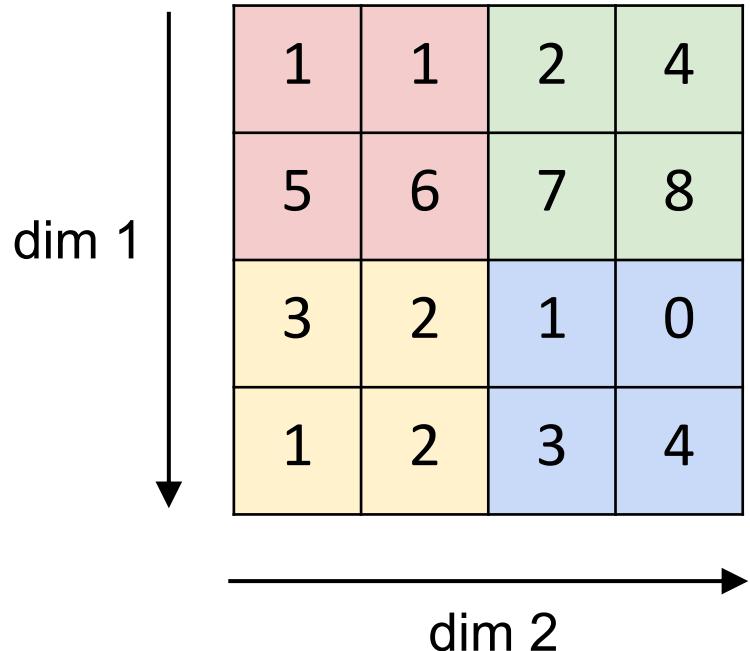


- Operates over each activation map independently
- Either '**max**' or '**average**' pooling is used at the pooling layer. That is, the convolved features are divided into disjoint regions and pooled by taking either maximum or averaging.

Pooling layer

MAX Pooling

Single depth slice



max pool with 2x2 filters
and stride 2

The result of the MAX Pooling operation is a 2x2 output tensor. It has two rows and two columns. The top-left cell contains the value 6 (light red), the top-right cell contains 8 (light green), the bottom-left cell contains 3 (light orange), and the bottom-right cell contains 4 (light blue). An arrow points from the input tensor to this output tensor.

6	8
3	4

*Pooling is intended to subsample the convolution layer.
The default stride for pooling is equal to the filter width.*

Pooling layer

Consider pooling with non-overlapping windows $\{(l, m)\}_{l,m=-L/2,-M/2}^{L/2,M/2}$, of size $L \times M$

The **max pooling** output is the maximum of the activation inside the pooling window. Pooling of a feature map y at $p = (i, j)$ produce pooled feature

$$z(i, j) = \max_{l,m} \{y(i + l, j + m)\}$$

The **mean pooling** output is the mean of activations in the pooling window

$$z(i, j) = \frac{1}{L \times M} \sum_l \sum_m y(i + l, j + m)$$

Pooling layer

Why pooling?

A function f is **invariant** to g if $f(g(x)) = f(x)$.

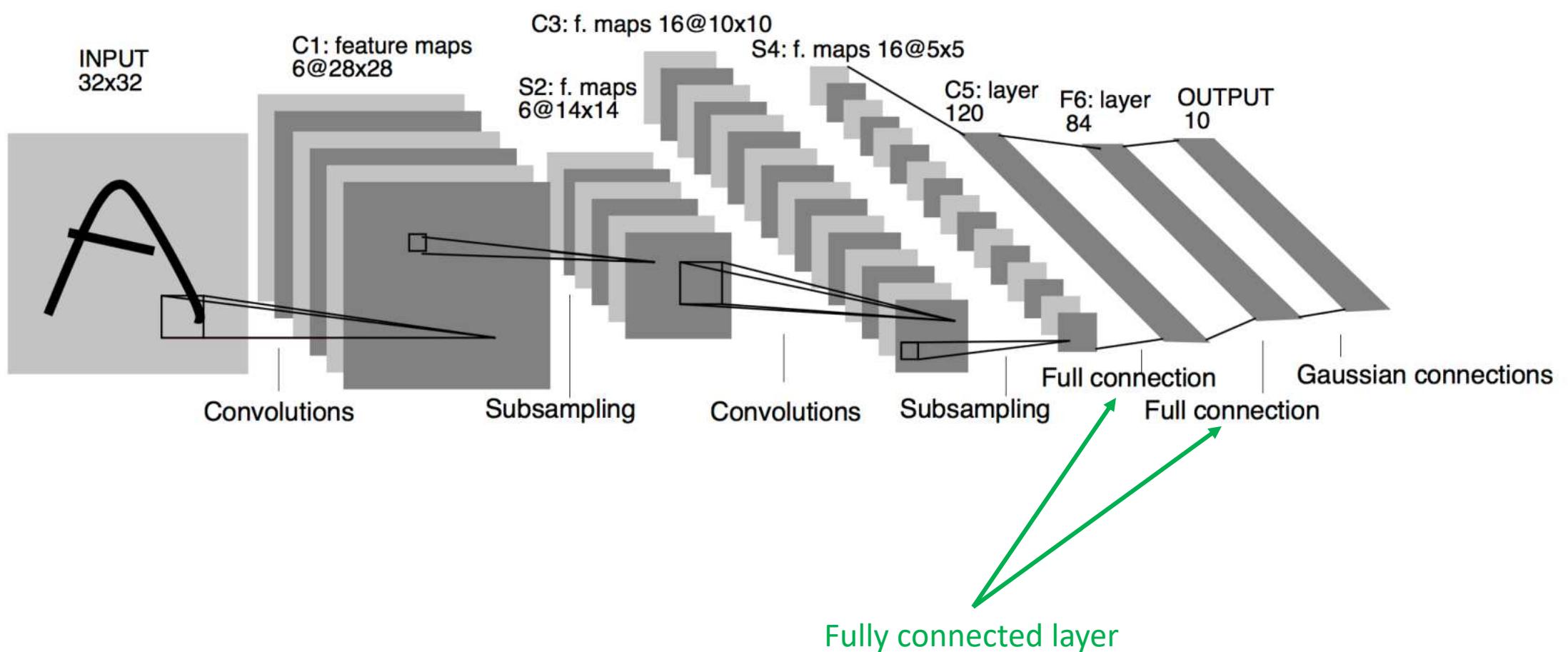
- Pooling layers can be used for building inner activations that are (slightly) invariant to small translations of the input.
- Invariance to local translation is helpful if we care more about the presence of a pattern rather than its exact position.

Pooling layer - summary

A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
 - Their spatial extent F
 - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $W_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

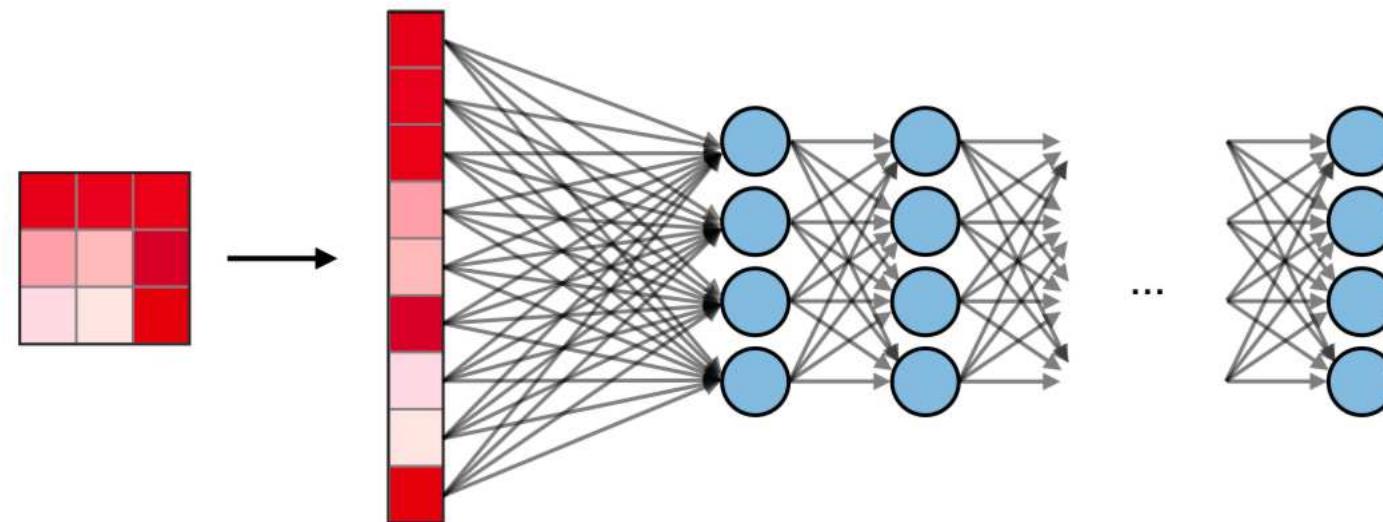
An example of convolutional network: LeNet 5



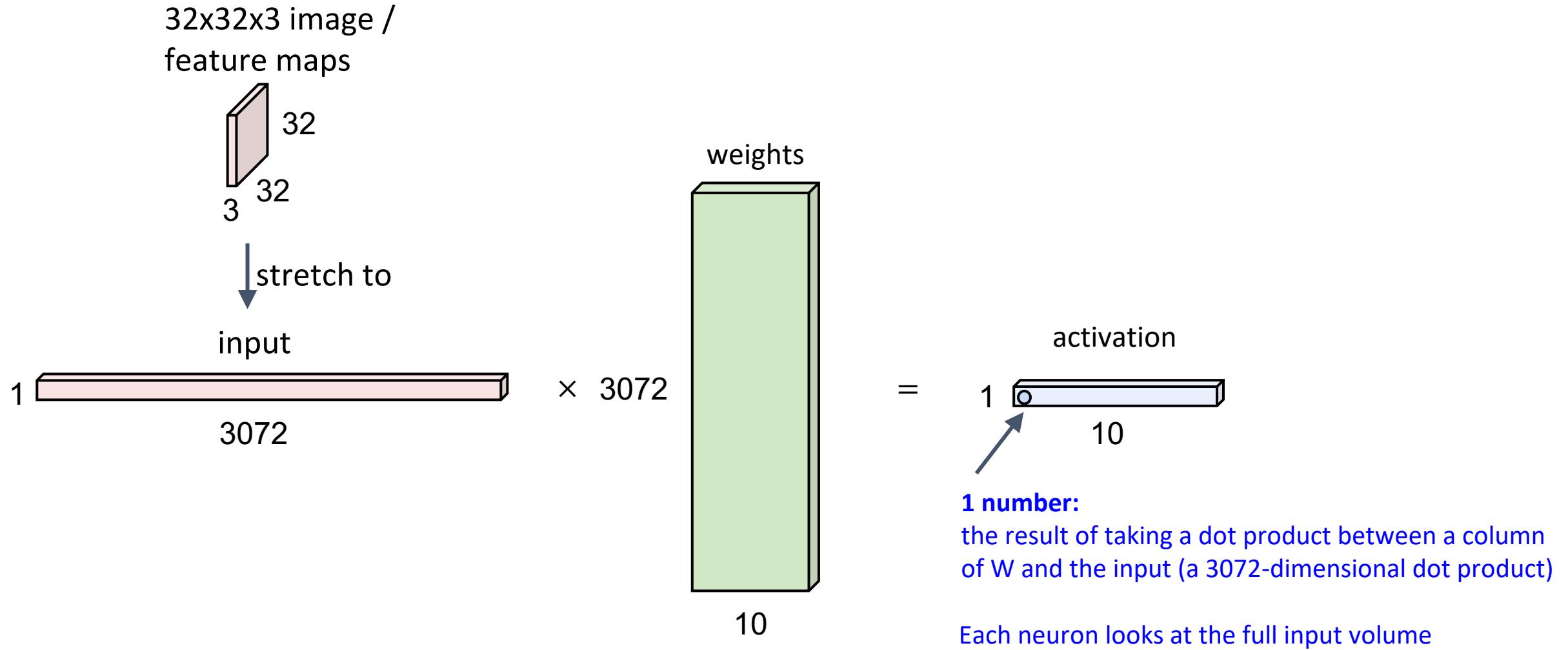
Fully connected layer

The fully connected layer (FC) operates on a **flattened input** where each input is connected to all neurons.

If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



Fully connected layer

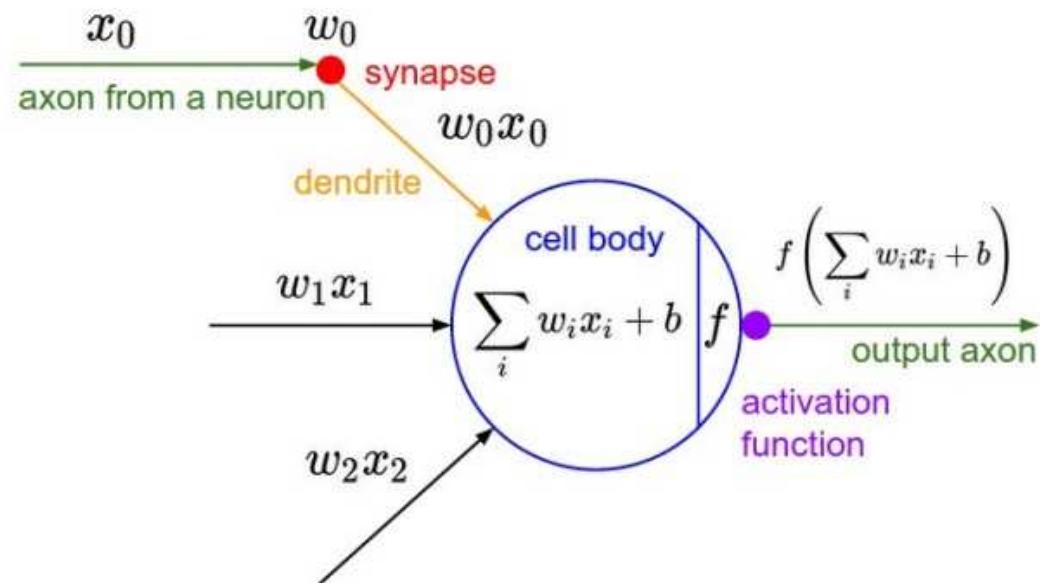


Activation function

Recall that the output of the neuron at (i, j) of the convolution layer

$$y(i, j) = f(u(i, j))$$

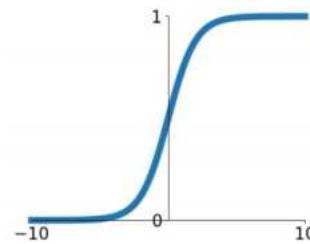
where u is the synaptic input and f is an activation function (It aims at introducing non-linearities to the network.).



Activation function

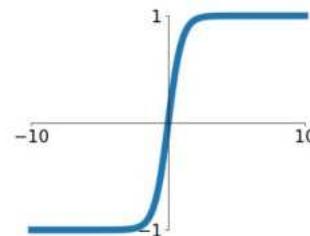
Sigmoid

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



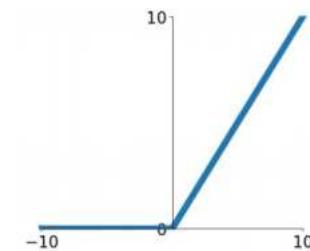
tanh

$$\tanh(x)$$



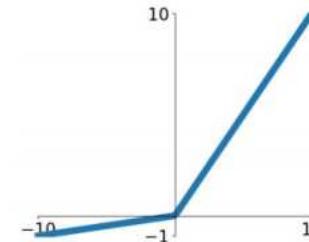
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

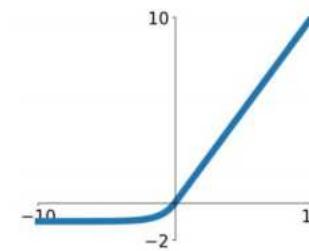


Maxout

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

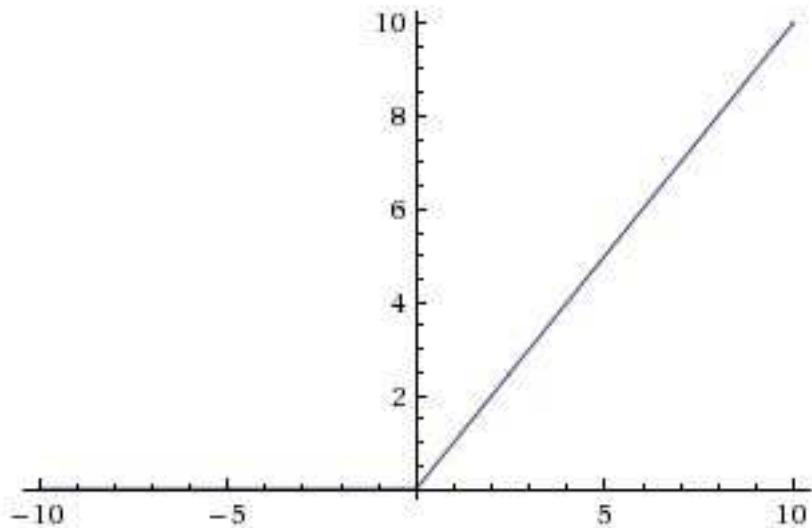
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



The activation function is usually an abstraction representing the rate of firing in the cell

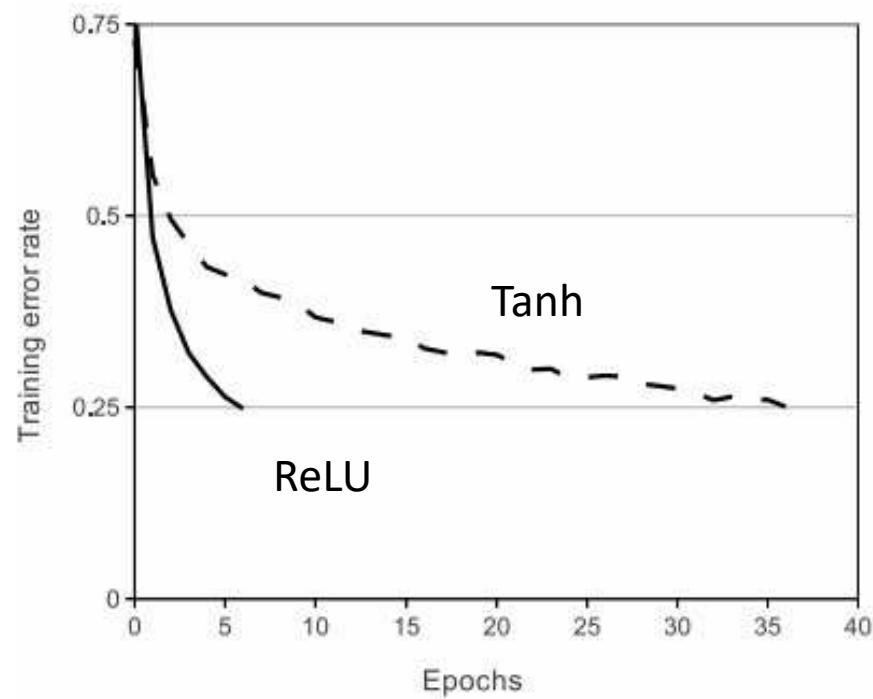
Activation function – ReLU



- Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$

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

Activation function – ReLU



- (+) It was found to greatly accelerate (e.g. a factor of 6 in [Krizhevsky et al.](#)) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.

Example 1

Given an input pattern X :

$$X = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}$$

The input pattern is received by a convolution layer consisting of one kernel (filter)

$$w = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \text{ and bias} = 0.05.$$

If convolution layer has a sigmoid activation function,

- Find the outputs of the convolution layer if the padding is VALID at strides = 1
- Assume the pooling layer uses max pooling, has a pooling window size of 2x2, and strides = 2, find the activations at the pooling layer.
- Repeat (a) and (b) using convolution layer with SAME padding

Example 1

Find the outputs of the convolution layer if the padding is VALID at strides = 1

$$\mathbf{I} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}; \mathbf{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Synaptic input to the pooling-layer:

$$u(i,j) = \sum_l \sum_m x(i + l, j + m)w(l, m) + b$$

For VALID padding:

$$u(1,1) = 0.5 \times 0 - 0.1 \times 1 + 0.2 \times 1 + 0.8 \times 1 + 0.1 \times 0 - 0.5 \times 1 - 1.0 \times 1 + 0.2 \times 1 + 0.0 \times 0 + 0.05 = -0.35$$

$$u(1,2) = -0.1 \times 0 + 0.2 \times 1 + 0.3 \times 1 + 0.1 \times 1 - 0.5 \times 0 + 0.5 \times 1 + 0.2 \times 1 + 0.0 \times 1 + 0.3 \times 0 + 0.05 = 1.35$$

$$u(1,3) = 0.2 \times 0 + 0.3 \times 1 + 0.5 \times 1 - 0.5 \times 1 + 0.5 \times 0 + 0.1 \times 1 + 0.0 \times 1 + 0.3 \times 1 - 0.2 \times 0 + 0.05 = 0.75$$

$$u(2,1) = 0.8 \times 0 + 0.1 \times 1 - 0.5 \times 1 - 0.1 \times 1 + 0.2 \times 0 + 0.0 \times 1 + 0.7 \times 1 + 0.1 \times 1 + 0.2 \times 0 + 0.05 = -0.55$$

⋮

Example 1

Synaptic input to the convolution layer:

$$\mathbf{U} = \begin{pmatrix} -0.35 & 1.35 & 0.75 \\ -0.55 & 0.85 & 0.05 \\ 0.75 & 0.05 & 1.15 \end{pmatrix}$$

Output of the convolution layer:

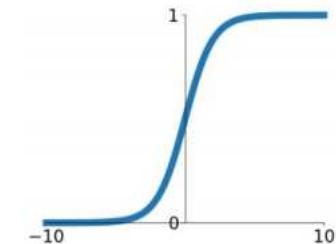
$$f(\mathbf{U}) = \frac{1}{1 + e^{-\mathbf{U}}} = \begin{pmatrix} 0.413 & 0.794 & 0.679 \\ 0.366 & 0.701 & 0.512 \\ 0.679 & 0.512 & 0.76 \end{pmatrix}$$

Output of the max pooling layer:

(0.794)

Now find the outputs of the convolution layer if the padding is SAME at strides = 1

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



See eg6.1.ipynb

Example 2

Inputs are digit images from MNIST database: <http://yann.lecun.com/exdb/mnist/>

Input image size = 28x28



First convolution layer consists of three filters:

$$w_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \quad w_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \quad w_3 = \begin{pmatrix} 3 & 4 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 3 \end{pmatrix}$$

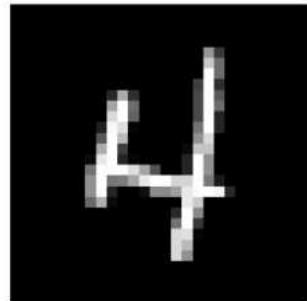
Find the feature maps at the convolution layer and pooling layer. Assume zero bias

For convolution layer, use a stride = 1 (default) and padding = 'VALID'.

For pooling layer, use a window of size 2x2 and a stride of 2.

See eg6.2.ipynb

Example 2



Original Image 28×28



Output of convolution layer
 $3 \times 26 \times 26$



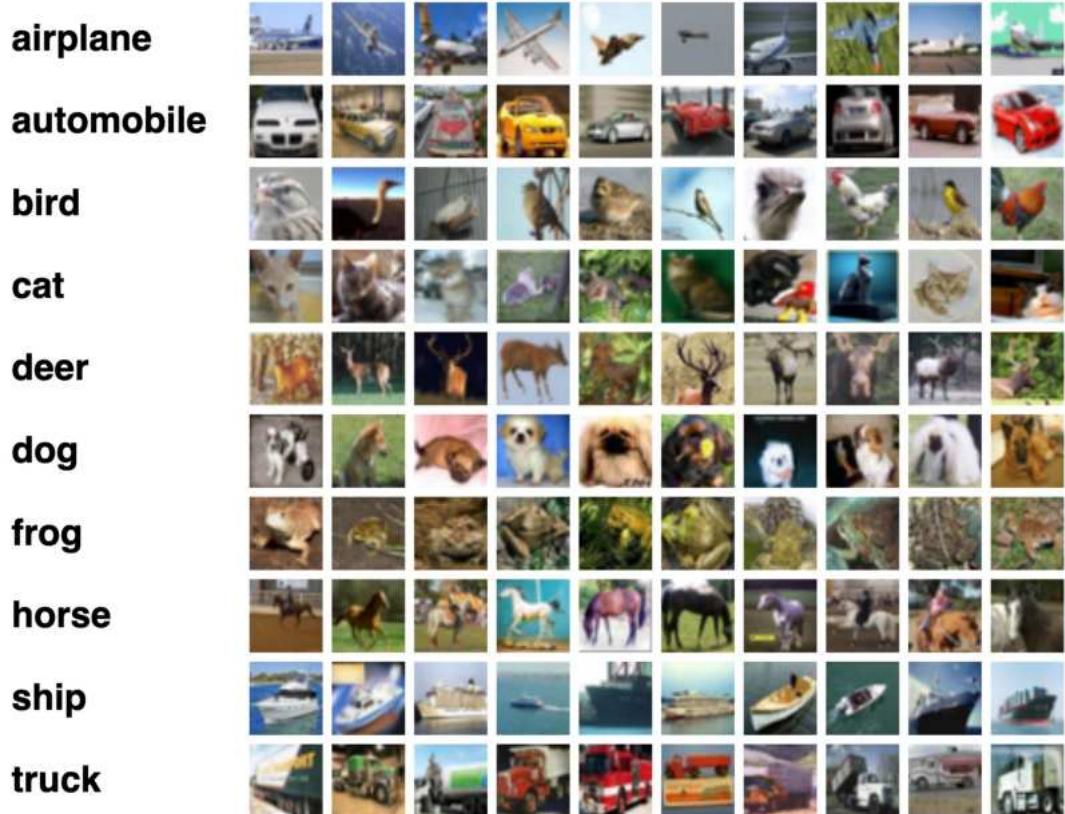
Output of pooling layer
 $3 \times 13 \times 13$

Outline

- Basic components in CNN
- Training a classifier
- Optimizers

Training a Classifier

Multi-class classification



CIFAR-10

- 10 classes
- 6000 images per class
- 60000 images - 50000 training images and 10000 test images
- Each image has a size of 3x32x32, that is 3-channel color images of 32x32 pixels in size

Multi-class classification

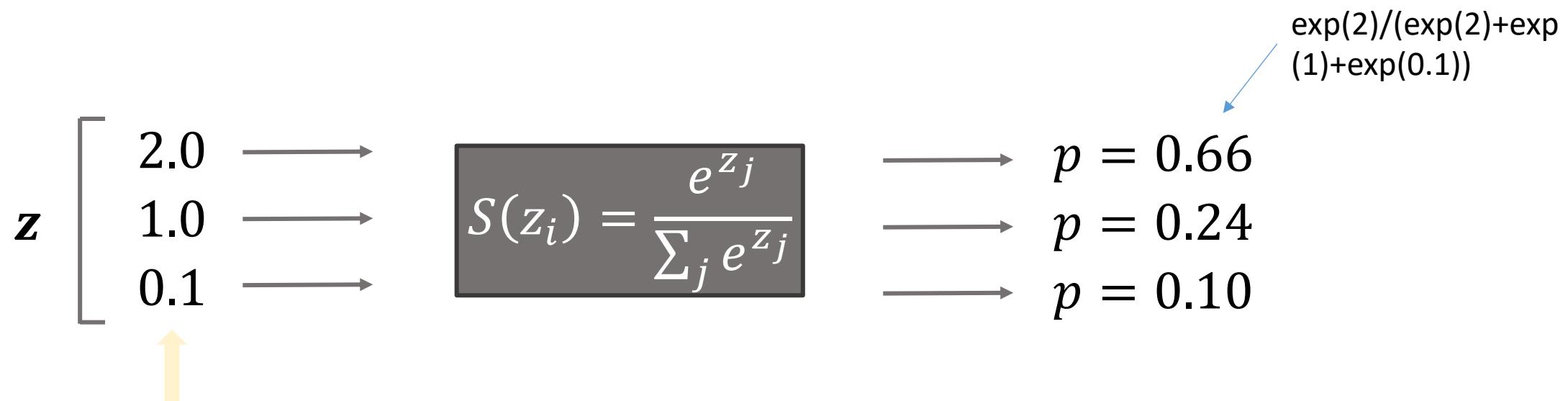


MNIST

- Size-normalized and centred 1x28x28 =784 inputs
- Training set = 60,000 images
- Testing set = 10,000 images

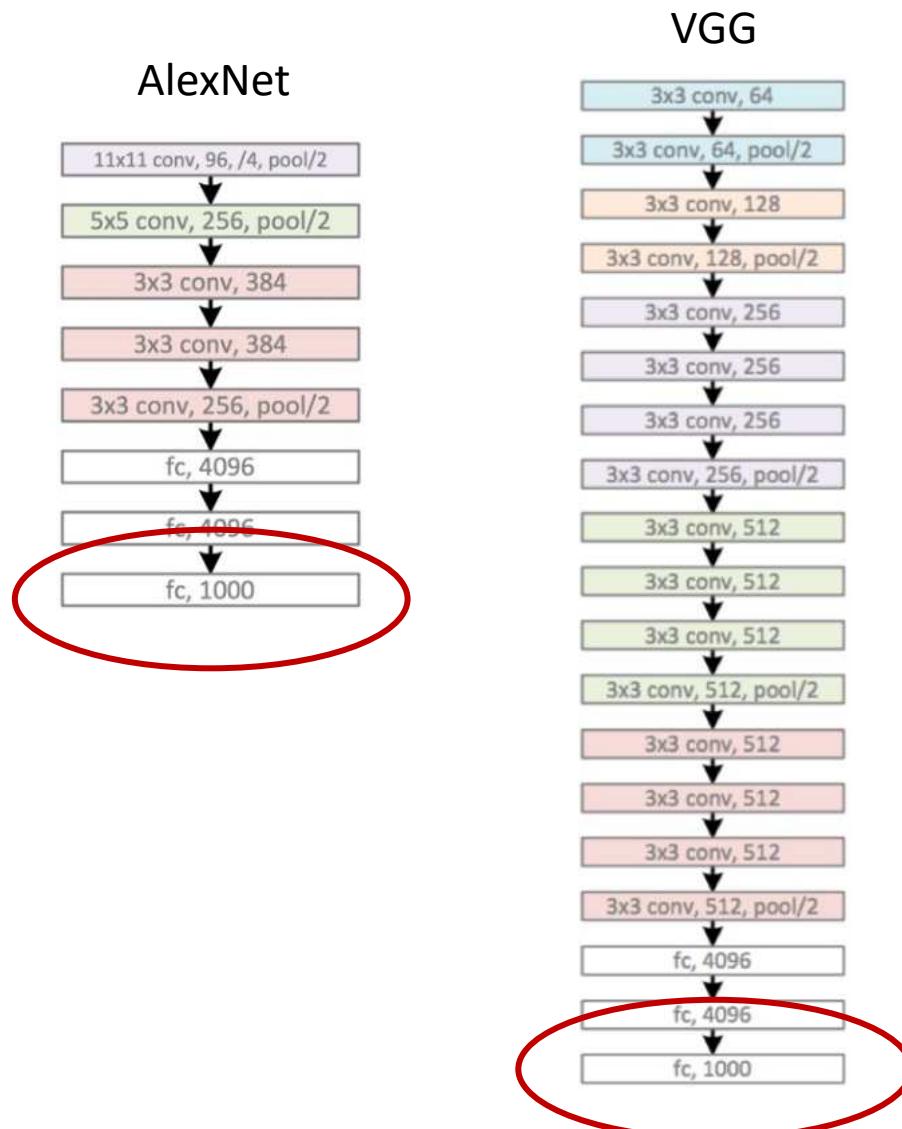
Softmax function

The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $\mathbf{z} \in \mathbb{R}^n$ and outputs a vector of output probability $\mathbf{p} \in \mathbb{R}^n$ through a softmax function at the end of the architecture.



Numeric output of the last linear layer of a multi-class classification neural network

Softmax function



Where does the Softmax function fit in a CNN architecture?

Softmax's input is the output of the fully connected layer immediately preceding it, and it outputs the final output of the entire neural network. This output is a probability distribution of all the label class candidates.

Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label

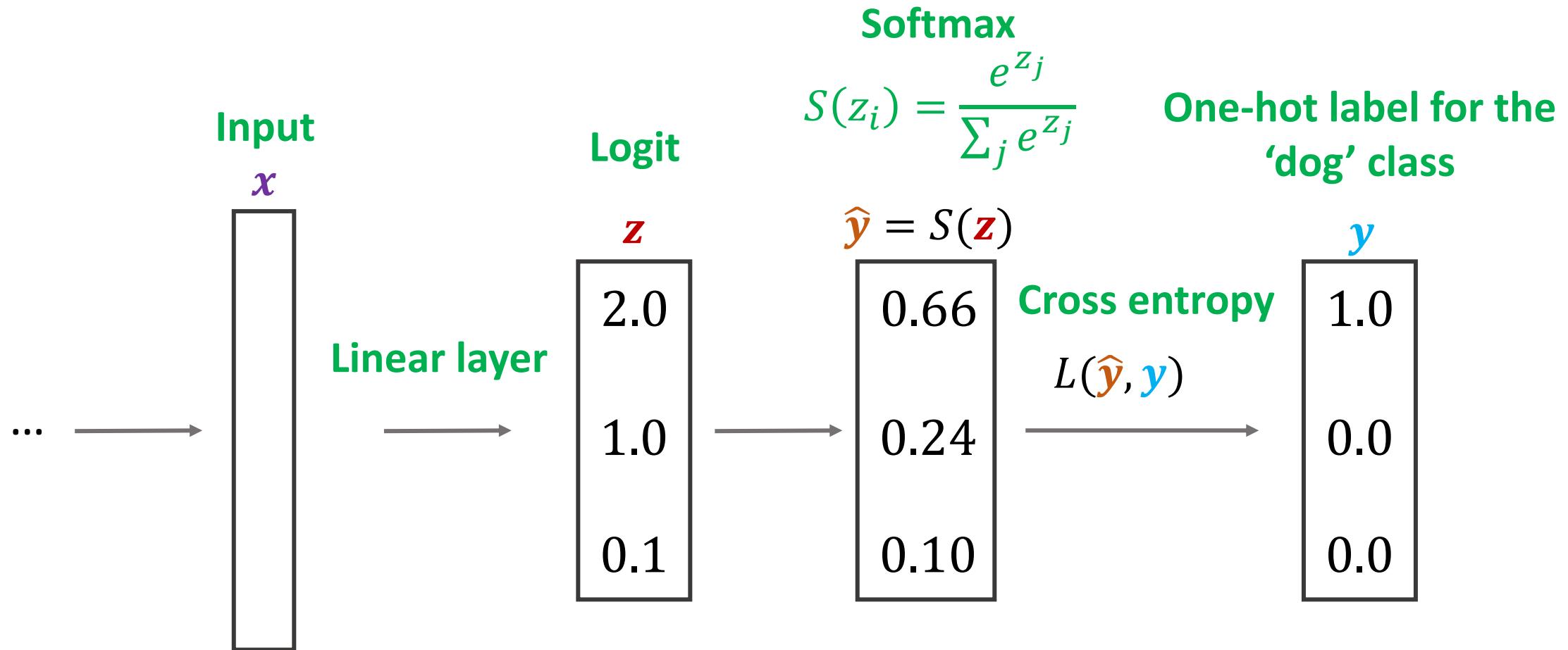
Cross entropy loss

One-hot encoded ground truth

Example:

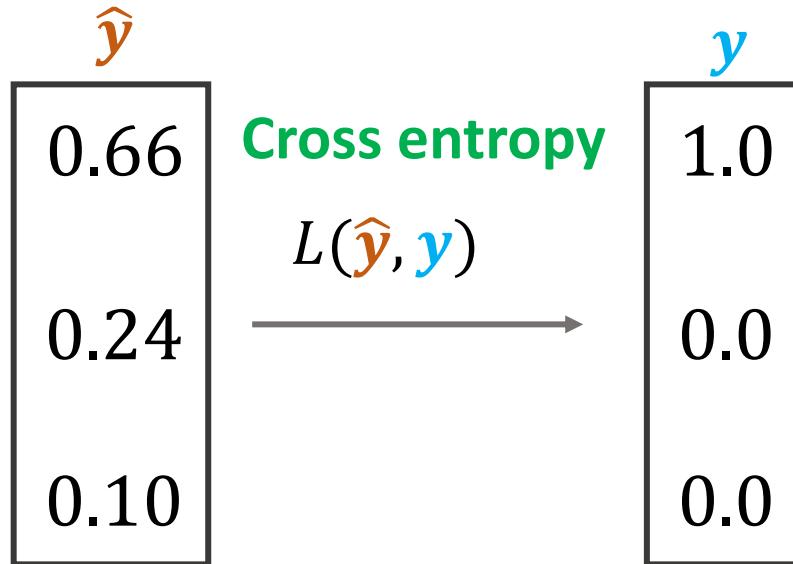
Class	One-hot vector
Dog	[1 0 0]
Cat	[0 1 0]
Bird	[0 0 1]

Cross entropy loss

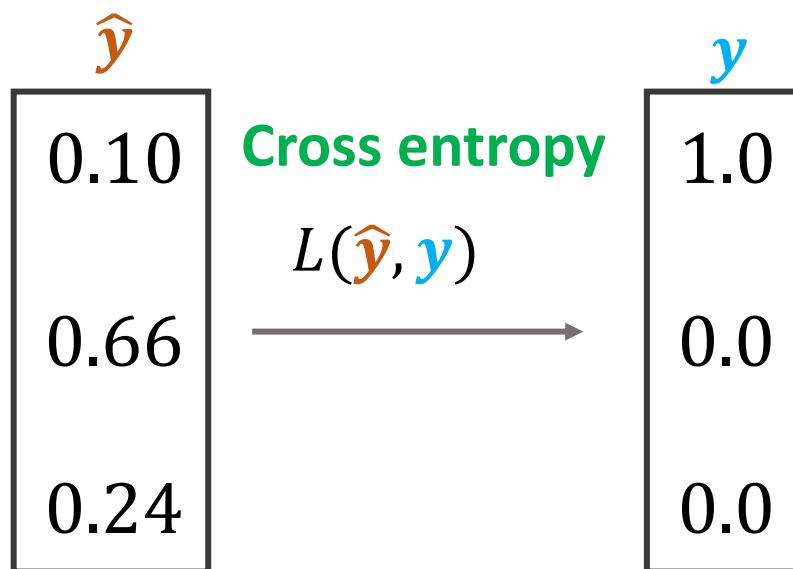


$$L(\hat{y}, y) = - \sum_i y_i \log(\hat{y}_i)$$

Cross entropy loss



$$\begin{aligned}L(\hat{y}, y) &= - \sum_i y_i \log(\hat{y}_i) \\&= -[(1 \times \log_2(0.66)) + (0 \times \log_2(0.24)) + (0 \times \log_2(0.10))] \\&= 0.6\end{aligned}$$

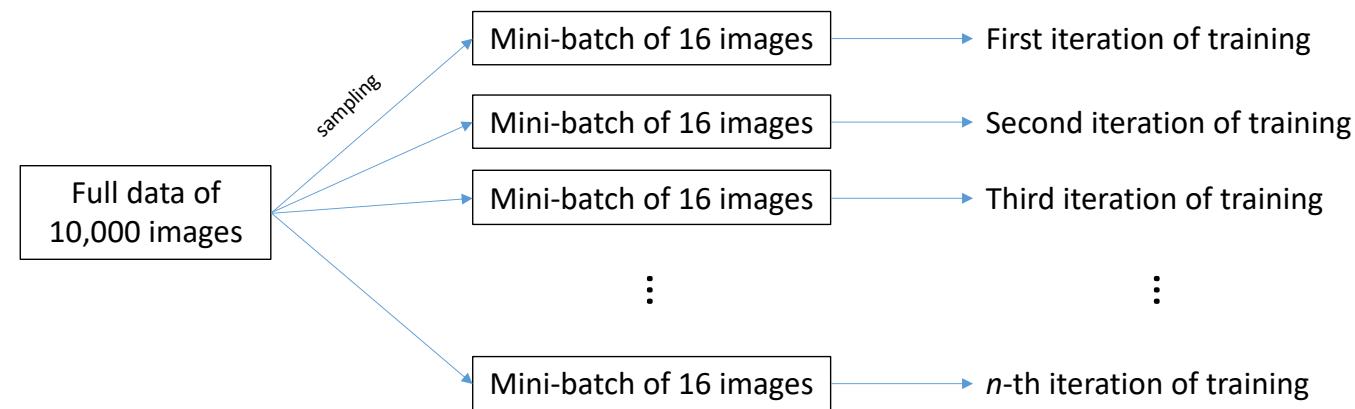


$$\begin{aligned}L(\hat{y}, y) &= - \sum_i y_i \log(\hat{y}_i) \\&= -[(1 \times \log_2(0.10)) + (0 \times \log_2(0.66)) + (0 \times \log_2(0.24))] \\&= 3.32\end{aligned}$$

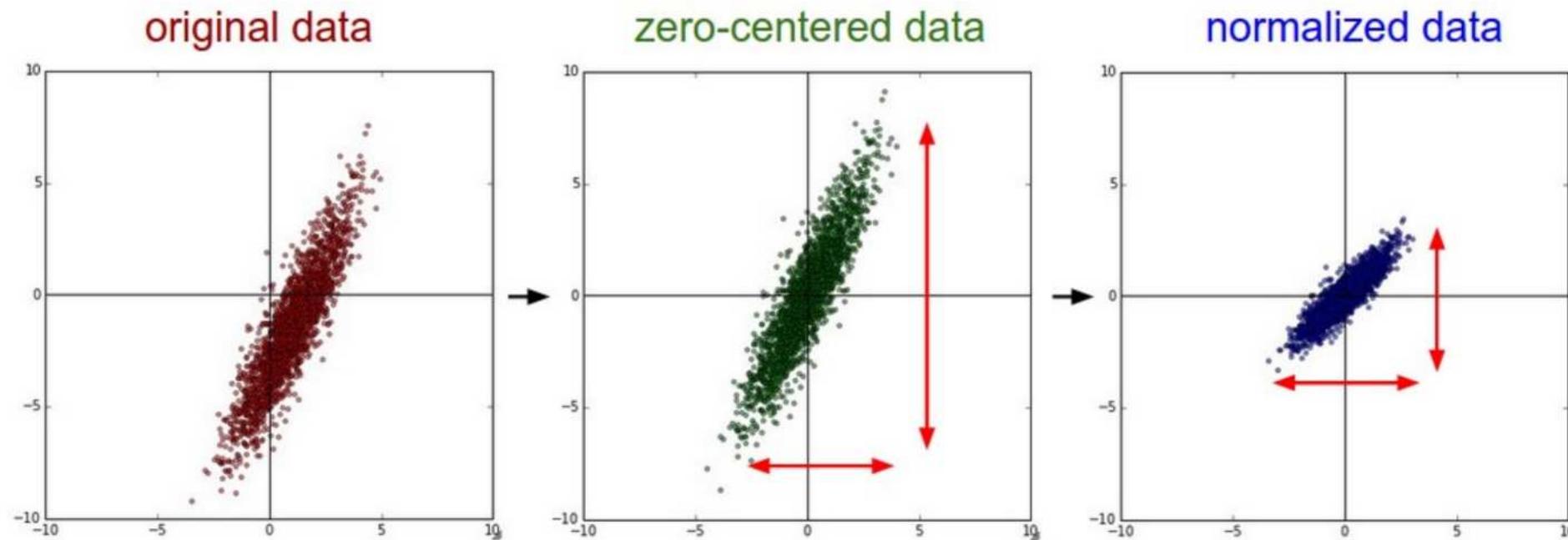
What is the min / max possible loss?

Epoch and mini-batch

- **Epoch** – In the context of training a model, epoch is a term used to refer to **one iteration where the model sees the whole training set to update its weights.**
- **Mini-batch** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on **mini batches**, where the number of data points in a batch is a hyperparameter that we can tune.



Data pre-processing



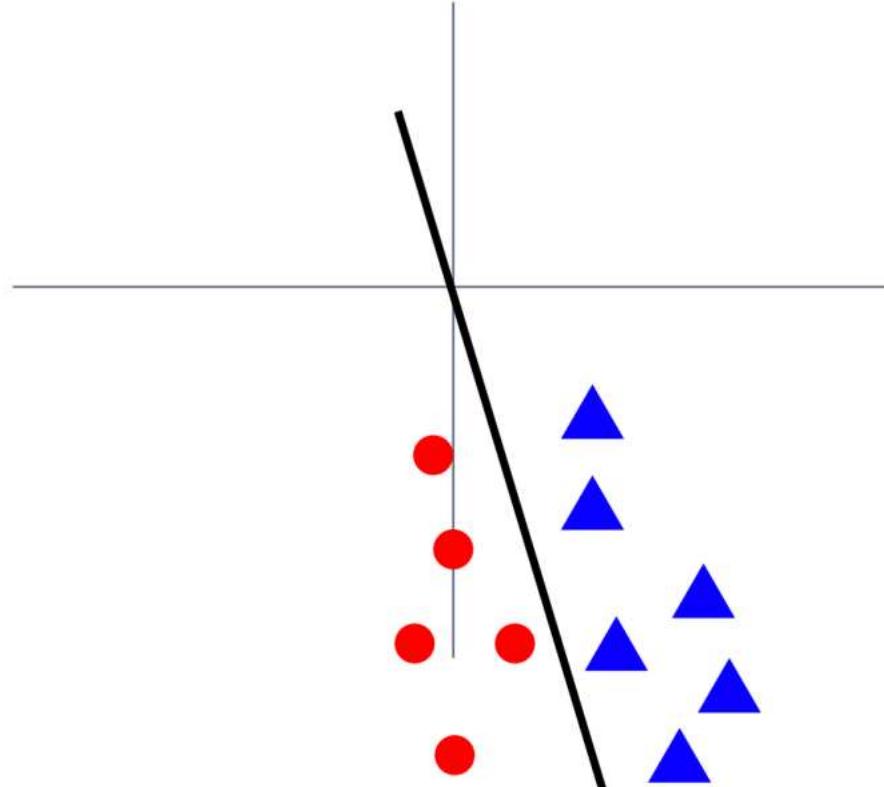
```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

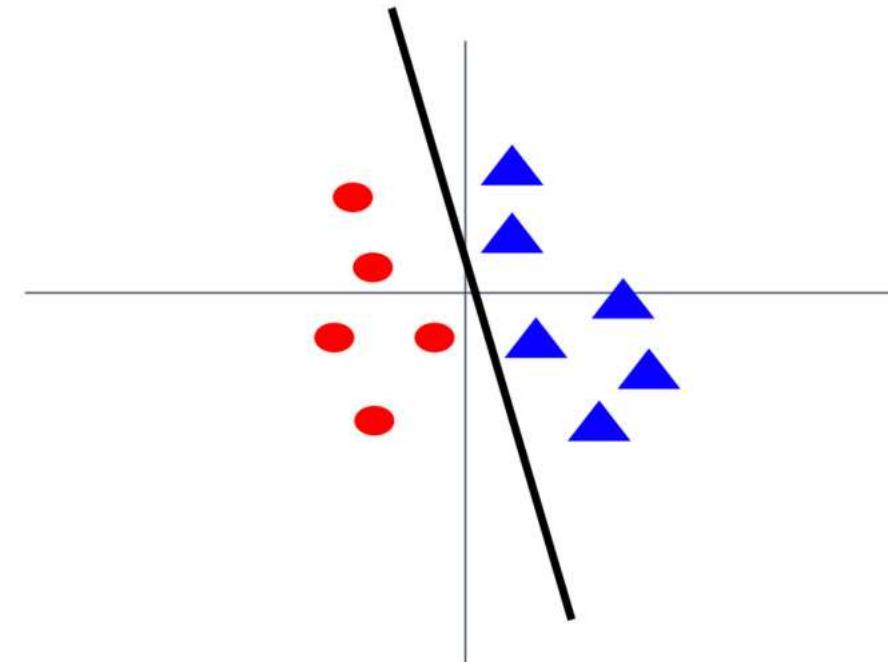
(Assume X [NxD] is data matrix,
each example in a row)

Data pre-processing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data pre-processing for images

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
(mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
(mean along each channel = 3 numbers)
- Subtract per-channel mean and
Divide by per-channel std (e.g. ResNet)
(mean along each channel = 3 numbers)

Outline

- Basic components in CNN
- Training a classifier
- Optimizers

Optimizers

Optimization

- A CNN as composition of functions

$$f_{\mathbf{w}}(\mathbf{x}) = f_L(\dots (f_2(f_1(\mathbf{x}; \mathbf{w}_1); \mathbf{w}_2) \dots; \mathbf{w}_L)$$

- Parameters

$$\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_L)$$

- Empirical loss function

$$L(\mathbf{w}) = \frac{1}{n} \sum_i l(y_i, f_{\mathbf{w}}(\mathbf{x}_i))$$

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} L(\mathbf{w})$$

Random search is a bad idea

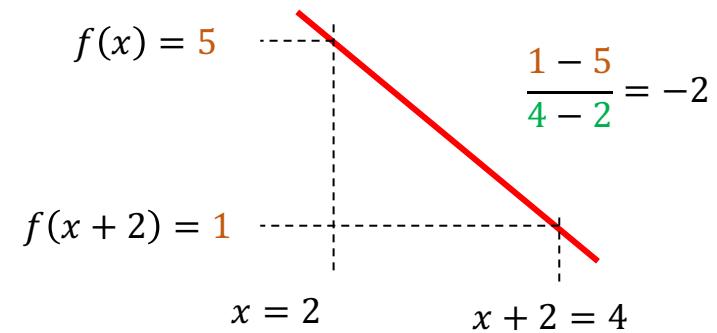
Follow the slope



Optimization

- In 1-dimension, the derivative of a function gives the slope:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$



- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension
- The direction of steepest descent is the **negative gradient**

Gradient descent (GD)

- Iteratively step in the direction of the negative gradient
- Gradient descent

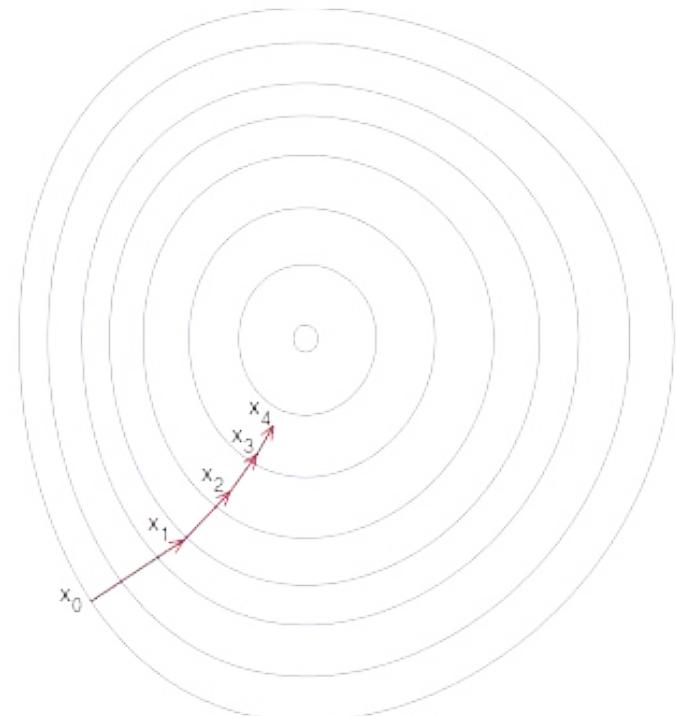
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \frac{\partial f(\mathbf{w}^t)}{\partial \mathbf{w}}$$

Diagram illustrating the components of the gradient descent update rule:

- New weight (Yellow box)
- Old weight (Red box)
- Learning rate (Green box)
- Gradient (Blue box)

Arrows point from the boxes to their corresponding terms in the equation:

- A yellow arrow points from the "New weight" box to the term \mathbf{w}^{t+1} .
- A red arrow points from the "Old weight" box to the term \mathbf{w}^t .
- A green arrow points from the "Learning rate" box to the term η_t .
- A blue arrow points from the "Gradient" box to the term $\frac{\partial f(\mathbf{w}^t)}{\partial \mathbf{w}}$.



Gradient descent (GD)

- Batch Gradient Descent
 - Full sum is *expensive* when N is large
- Stochastic Gradient Descent (SGD)
 - Approximate sum using a minibatch of examples
 - 32 / 64 / 128 common minibatch size
 - Additional hyperparameter on batch size

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

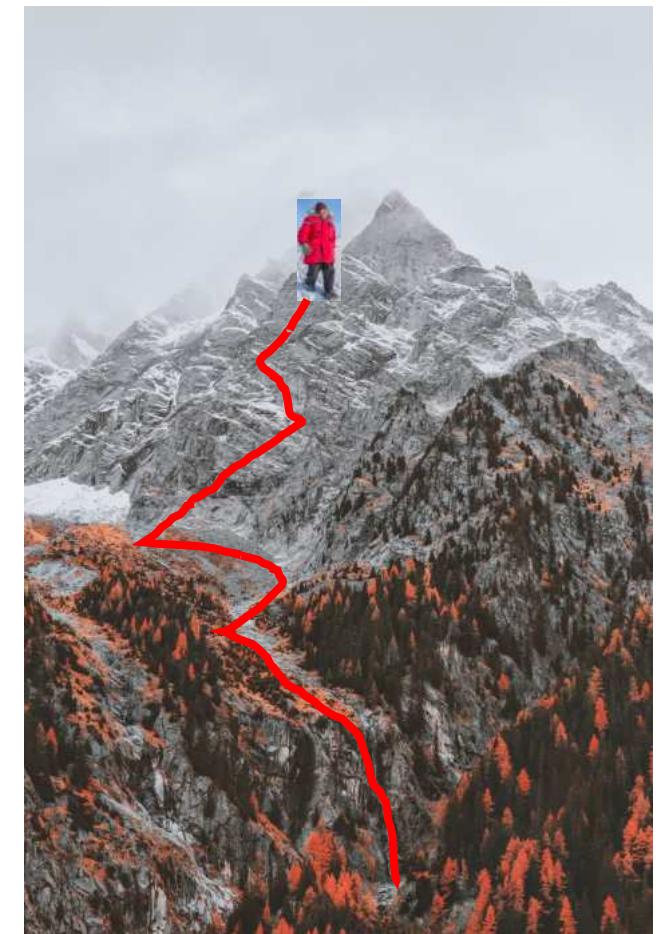
```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
    minibatch = sample_data(data, batch_size)
    dw = compute_gradient(loss_fn, minibatch, w)
    w -= learning_rate * dw
```

GD with Momentum

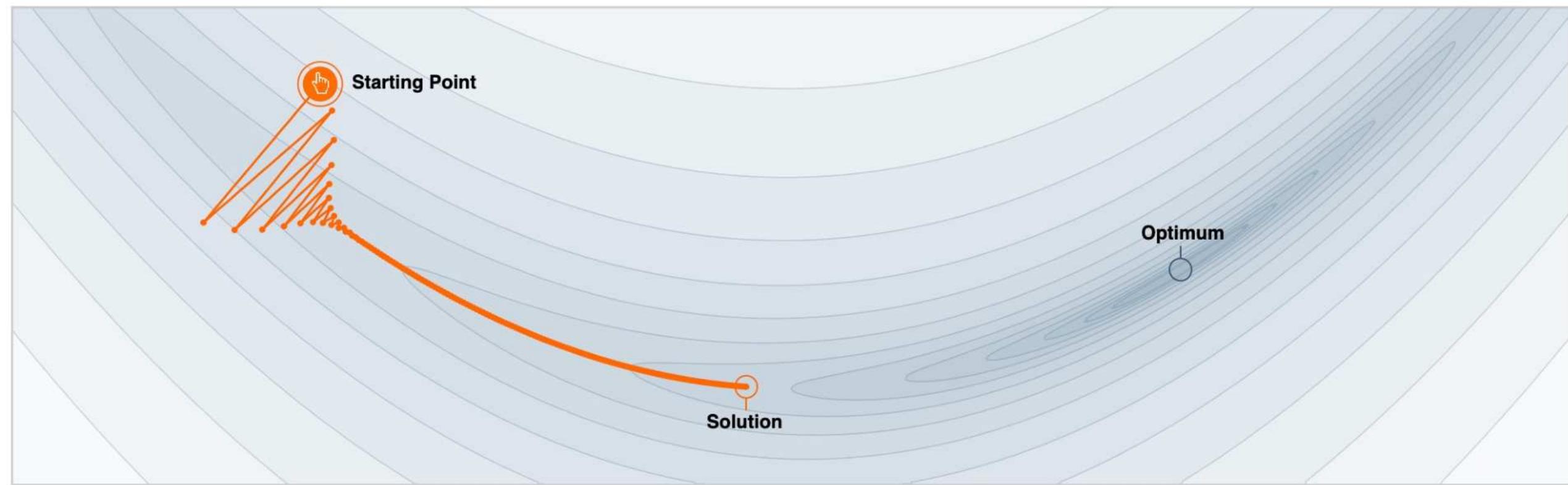
Deep neural networks have very complex error profiles. The method of momentum is designed to **accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.**

When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to **oscillate near the optimum**. This leads to **very slow converging rates**. This problem is typical in deep learning architecture.

Momentum is one method of **speeding the convergence along a narrow ravine**.



GD with Momentum



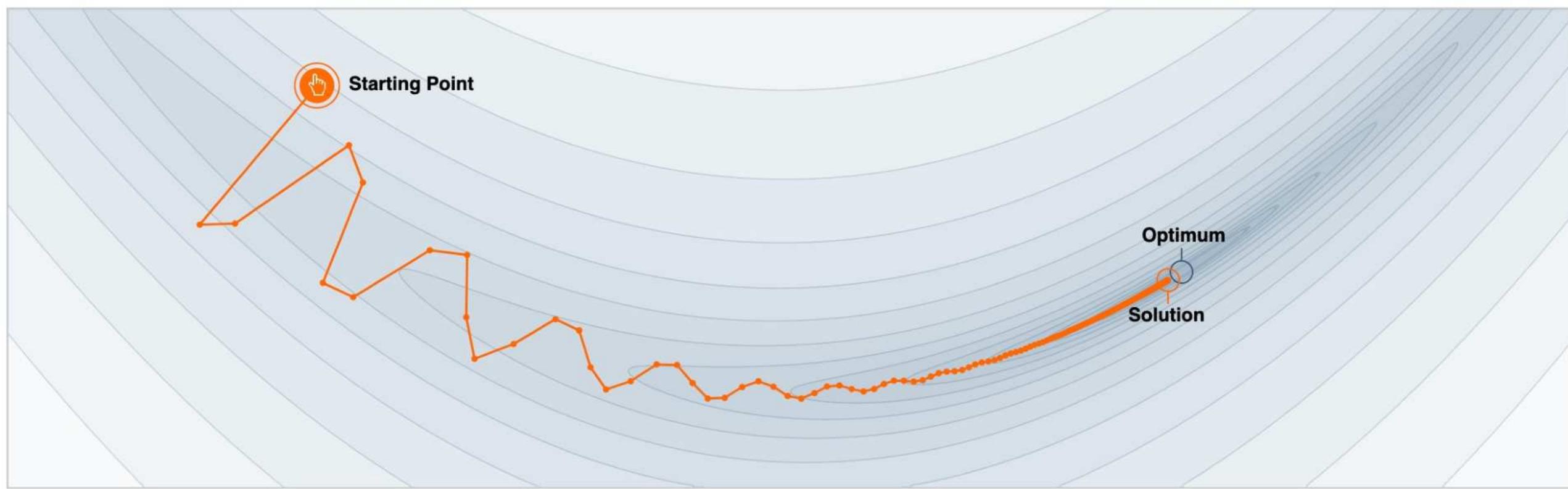
Step-size $\alpha = 0.0030$



Momentum $\beta = 0.0$



GD with Momentum



Step-size $\alpha = 0.0030$

0 0.003 0.006

Momentum $\beta = 0.80$

0 0.500 0.990

GD with Momentum

Momentum update is given by:

$$\begin{aligned} \mathbf{V} &\leftarrow \gamma \mathbf{V} - \alpha \nabla_{\mathbf{W}} J \\ \mathbf{W} &\leftarrow \mathbf{W} + \mathbf{V} \end{aligned}$$

where \mathbf{V} is known as the **velocity** term and has the same dimension as the weight vector \mathbf{W} .

The momentum parameter $\gamma \in [0,1]$ indicates how many iterations the previous gradients are incorporated into the current update.

The momentum algorithm **accumulates an exponentially decaying moving average of past gradients** and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

Learning rate

$$w^{t+1} = w^t - \eta_t \frac{\partial f(w^t)}{\partial w}$$

Diagram illustrating the update rule for learning weights:

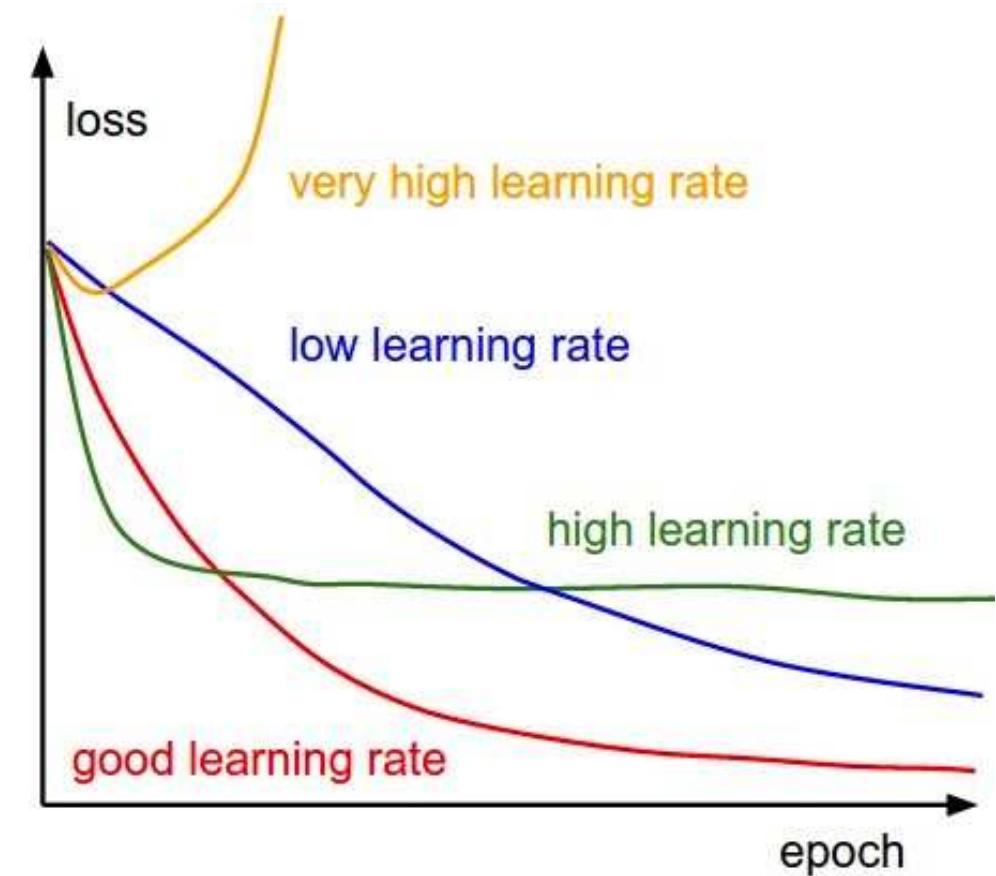
- New weight (Yellow box)
- Old weight (Red box)
- Learning rate (Green box)
- Gradient (Blue box)

The diagram shows the formula for calculating the new weight at time step $t+1$ based on the old weight, learning rate, and gradient.

Learning rate

The learning rate, often noted α or sometimes η , indicates at **which pace the weights get updated**. It can be fixed or adaptively changed.

The current most popular method is called **Adam**, which is a method that adapts the learning rate.



Learning rate

- **Adaptive learning rates**
 - Letting the learning rate vary when training a model can **reduce the training time and improve the numerical optimal solution.**
 - While **Adam** optimizer is the most commonly used technique, others can also be useful.
- **Algorithms with adaptive learning rates:**
 - AdaGrad `torch.optim.Adagrad()`
 - RMSprop `torch.optim.RMSprop()`
 - Adam `torch.optim.Adam()`

Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to **start with a large learning factor and then gradually reducing it.**

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and $\alpha(\infty) = 0$.

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.

Learning trajectory of a neural network **minimizing non-convex cost function** passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by **scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient**. This improves the learning rates, especially in the convex regions of error function.

$$\begin{aligned} \mathbf{r} &\leftarrow \mathbf{r} + (\nabla_{\mathbf{W}} J)^2 \\ \mathbf{W} &\leftarrow \mathbf{W} - \frac{\alpha}{\varepsilon + \sqrt{\mathbf{r}}} \cdot (\nabla_{\mathbf{W}} J) \end{aligned}$$

In other words, learning rate:

$$\tilde{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{\mathbf{r}}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to **discard the history from extreme past** so that it can converge rapidly after finding a convex region.

$$\begin{aligned}\mathbf{r} &\leftarrow \rho \mathbf{r} + (1 - \rho)(\nabla_{\mathbf{W}} J)^2 \\ \mathbf{W} &\leftarrow \mathbf{W} - \frac{\alpha}{\sqrt{\varepsilon + \mathbf{r}}} \cdot (\nabla_{\mathbf{W}} J)\end{aligned}$$

The decay constant ρ controls the length of the moving average of gradients.

Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer **combines RMSprop and momentum** methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

Momentum term: $s \leftarrow \rho_1 s + (1 - \rho_1) \nabla_{\mathbf{W}} J$

Learning rate term: $r \leftarrow \rho_2 r + (1 - \rho_2) (\nabla_{\mathbf{W}} J)^2$

$$s \leftarrow \frac{s}{1 - \rho_1}$$

$$r \leftarrow \frac{r}{1 - \rho_2}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\alpha}{\varepsilon + \sqrt{r}} \cdot s$$

Note that s adds the momentum and r contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001$, $\rho_1 = 0.9$, $\rho_2 = 0.999$, and $\varepsilon = 10^{-8}$

Example 3: MNIST digit recognition

MNIST database: $28 \times 28 = 784$ inputs

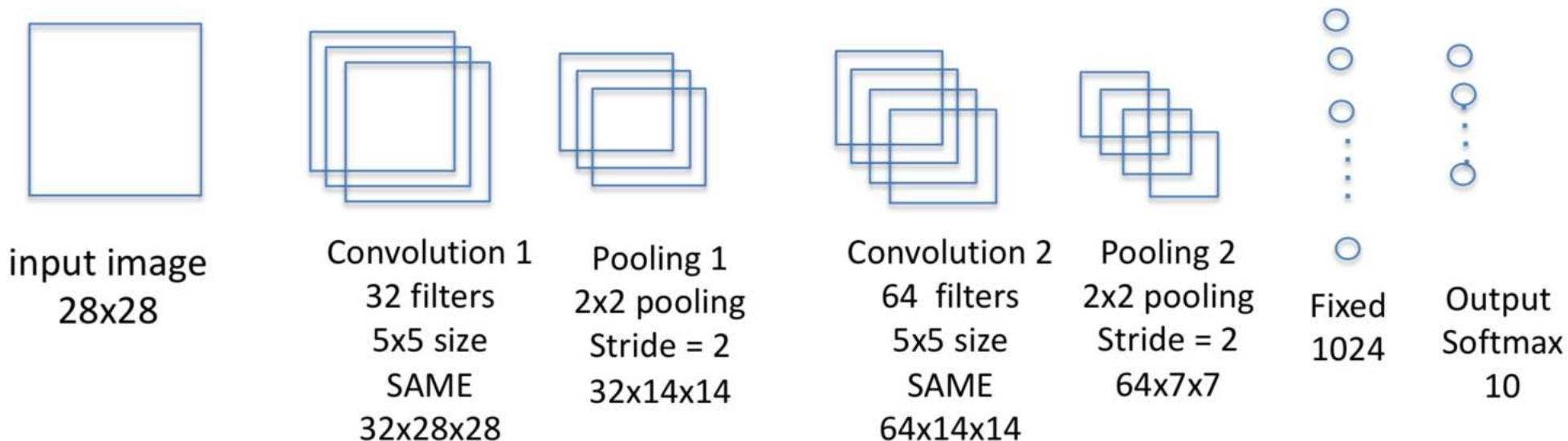
Training set = 12000 images

Testing set = 2000 images

Input pixel values were normalized to $[0, 1]$



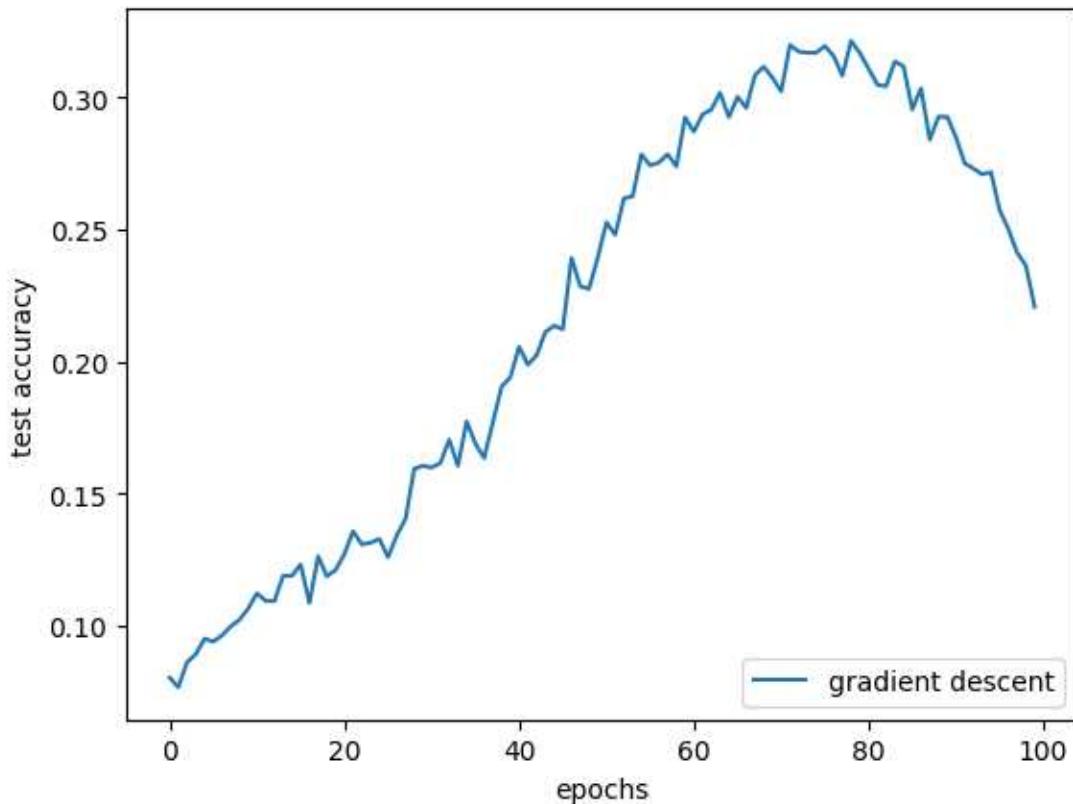
Example 3: Architecture of CNN



ReLU neurons
Gradient descent optimizer with batch-size = 128
Learning parameter = 10^{-3}

See eg6.3.ipynb

Example 3: Training Curve



Example 3: Weights learned

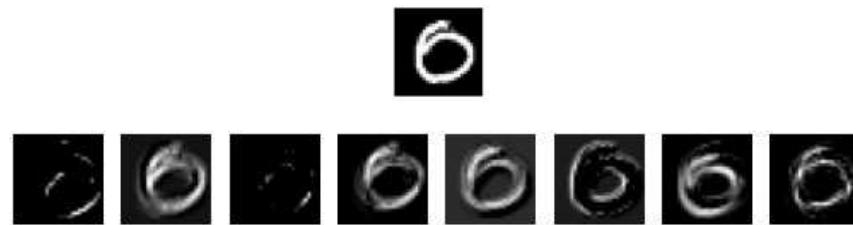
Weights learned at convolution layer 1



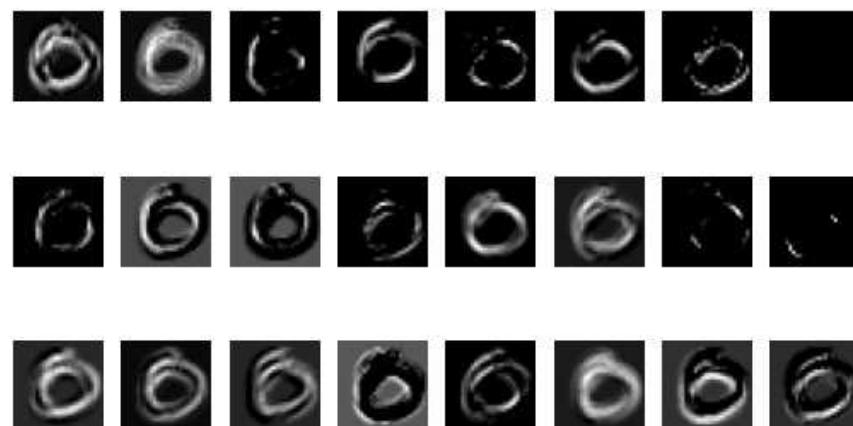
32x5x5

Example 3: Feature maps

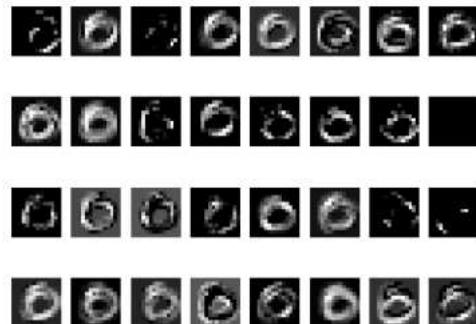
Input image
28x28



Feature maps at conv1
32x28x28

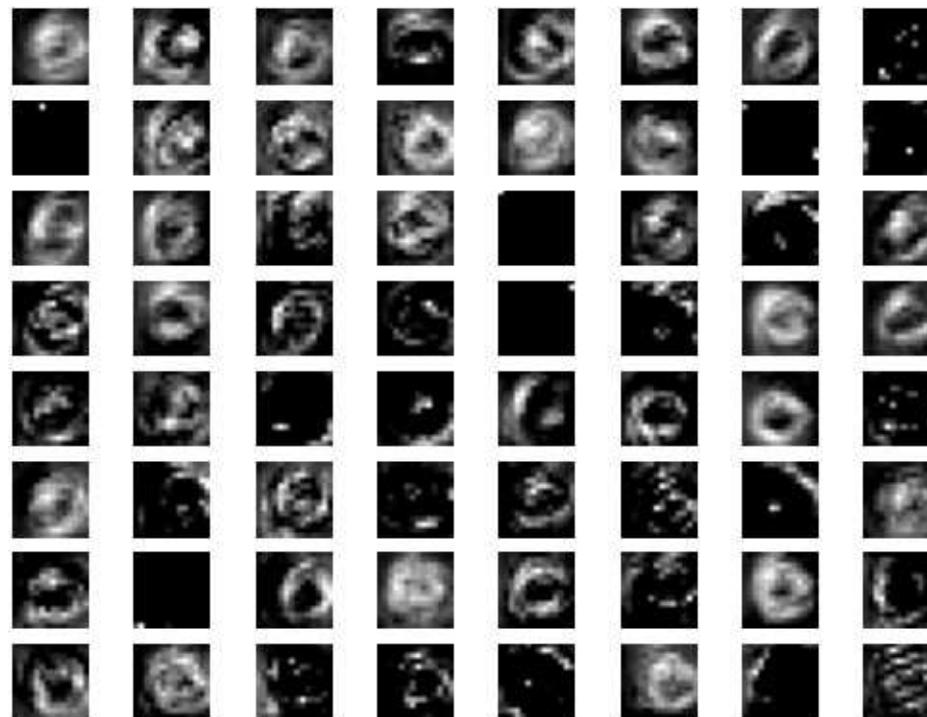


Feature maps at pool1
32x14x14



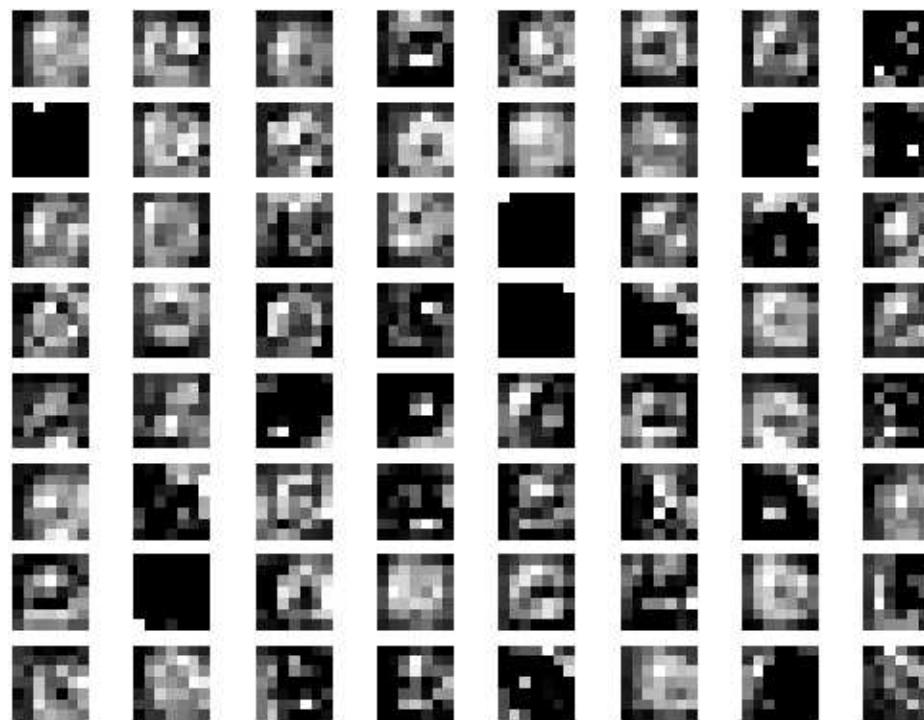
Example 3: Feature maps

Feature maps at conv2
64x14x14

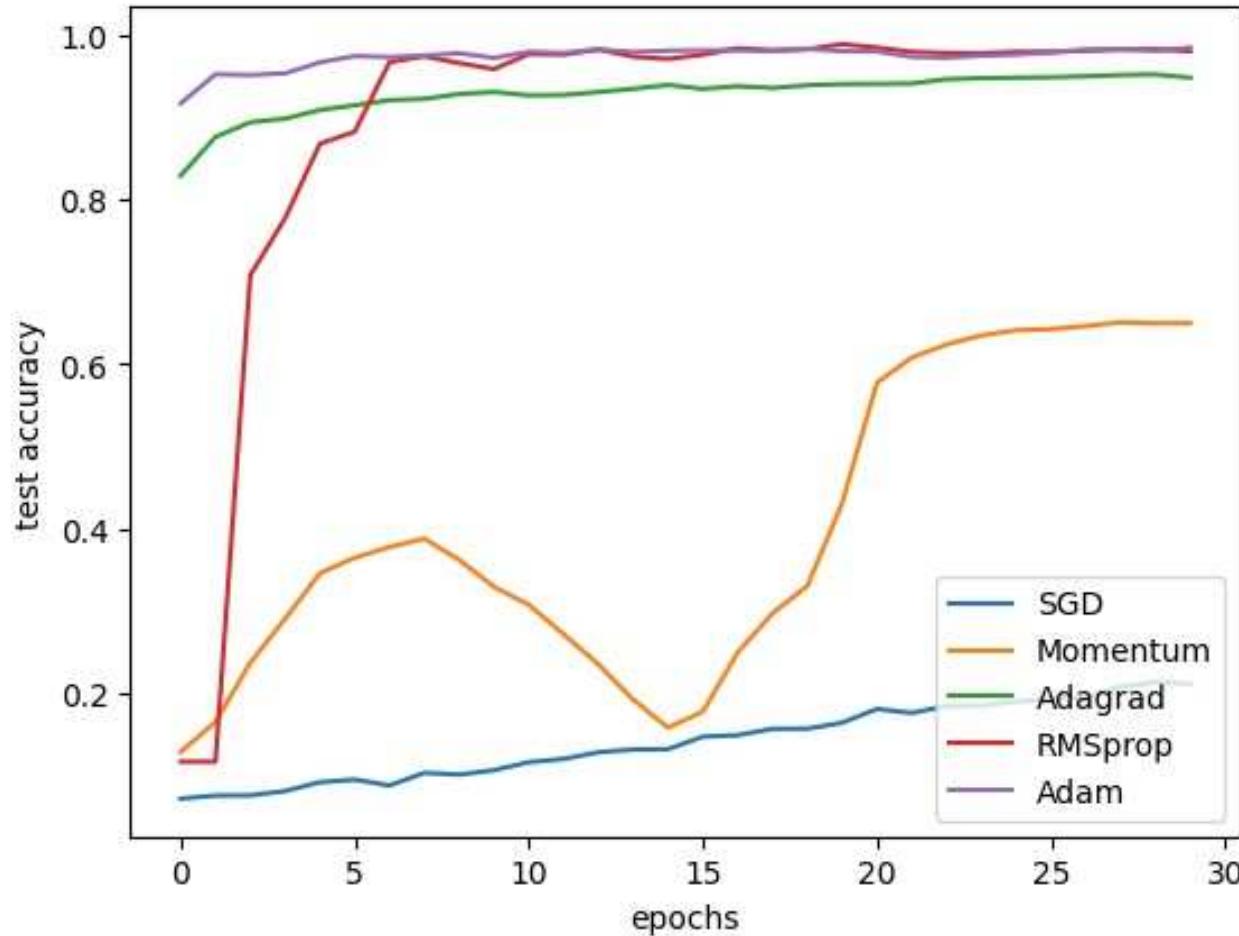


Example 3: Feature maps

Feature maps at pool2
64x7x7



Example 4: MNIST recognition with CNN with different learning algorithms



See eg6.4.ipynb

Next lecture

- CNN Architectures
 - You learn some classic architectures
- More on convolution
 - How to calculate FLOPs
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution
 - You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network
- Batch normalization
 - You learn an important technique to improve the training of modern neural networks
- Prevent overfitting
 - Transfer learning
 - Data augmentation
 - You learn two important techniques to prevent overfitting in neural networks

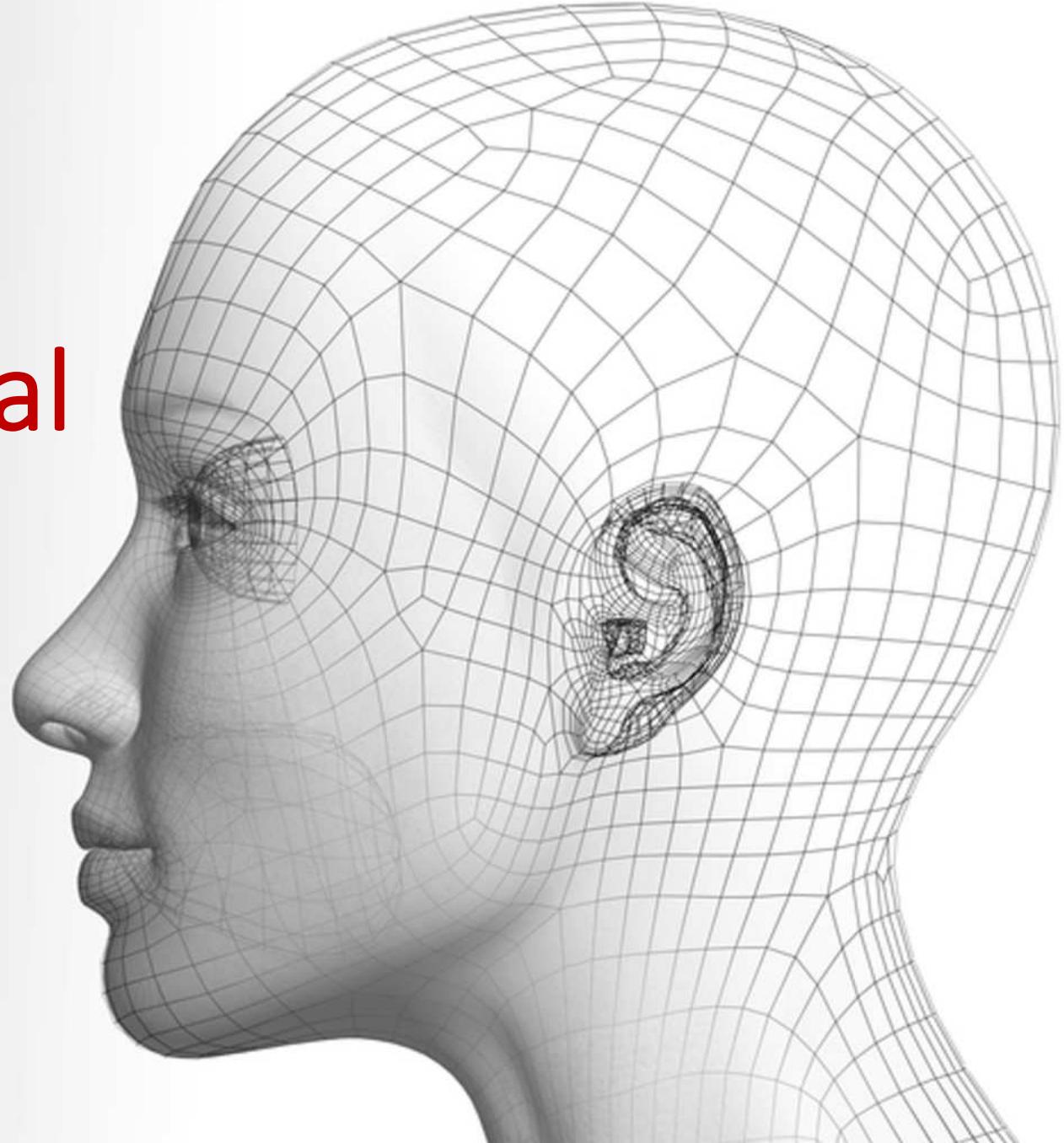
Convolutional Neural Networks II

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

<https://twitter.com/XingangP>



Outline

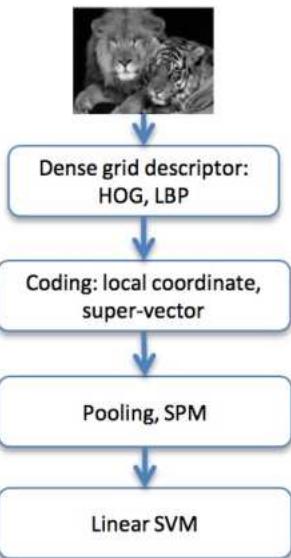
- CNN Architectures
 - You learn some classic architectures
- More on convolution
 - How to calculate FLOPs
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution
 - You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network
- Batch normalization
 - You learn an important technique to improve the training of modern neural networks
- Prevent overfitting
 - Transfer learning
 - Data augmentation
 - You learn two important techniques to prevent overfitting in neural networks

CNN Architectures

Deep networks for ImageNet

Year 2010

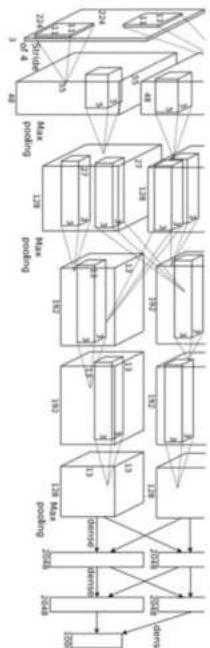
NEC-UIUC



[Lin CVPR 2011]

Year 2012

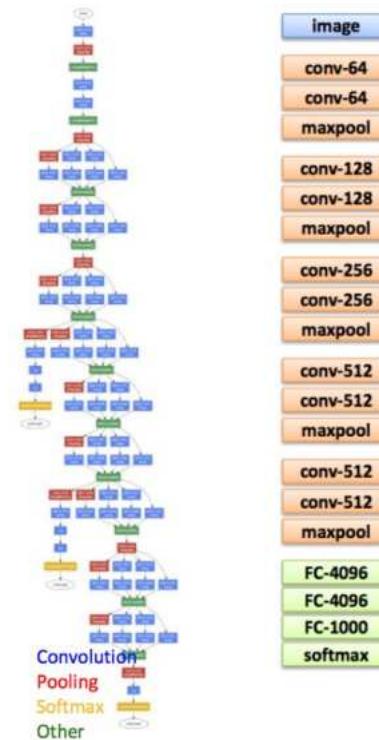
AlexNet



[Krizhevsky NIPS 2012]

Year 2014

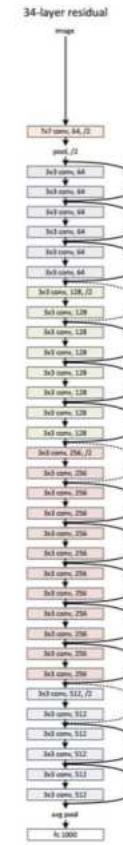
GoogLeNet VGG



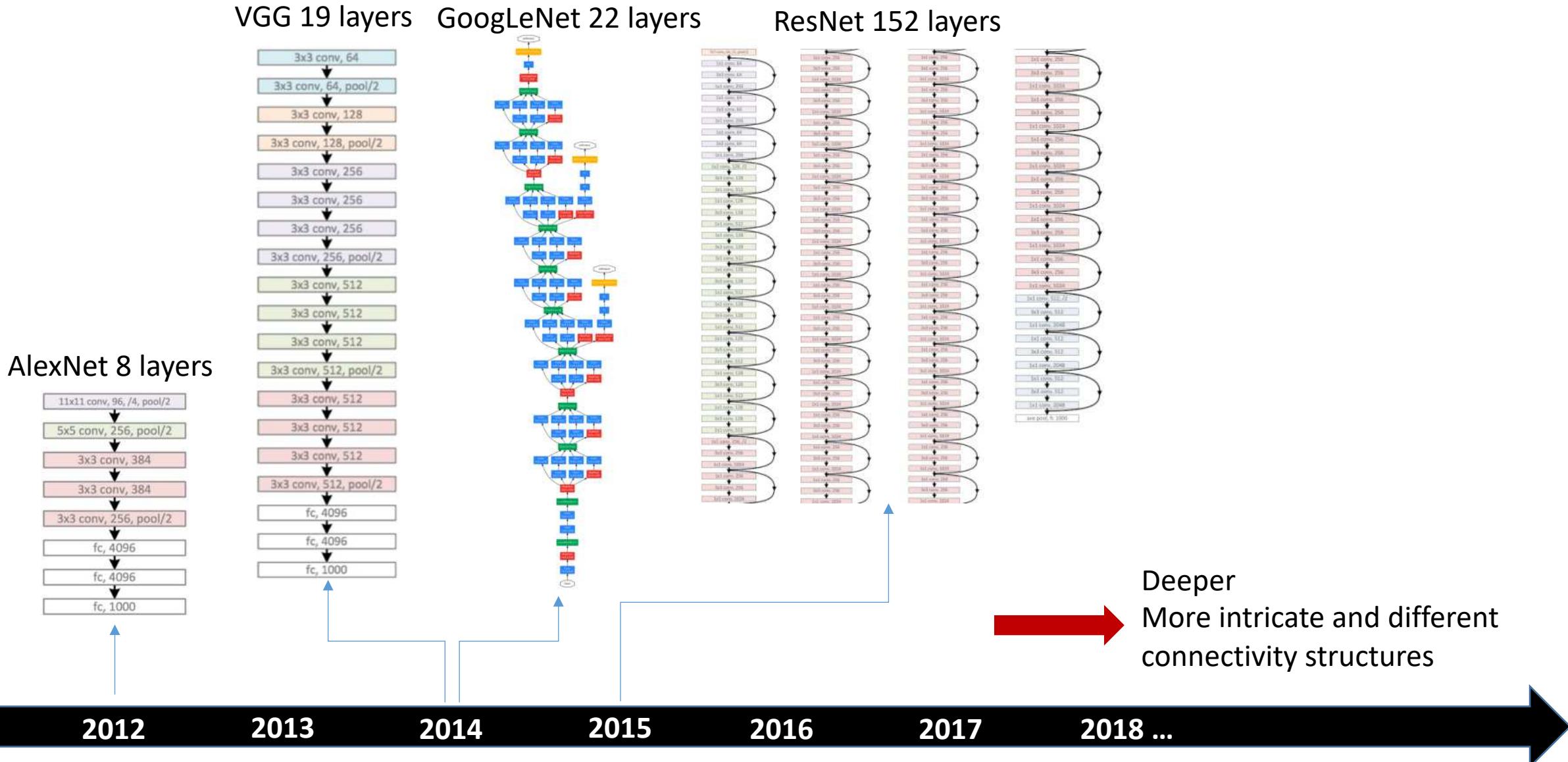
[Szegedy arxiv 2014] [Simonyan arxiv 2014]

Year 2015

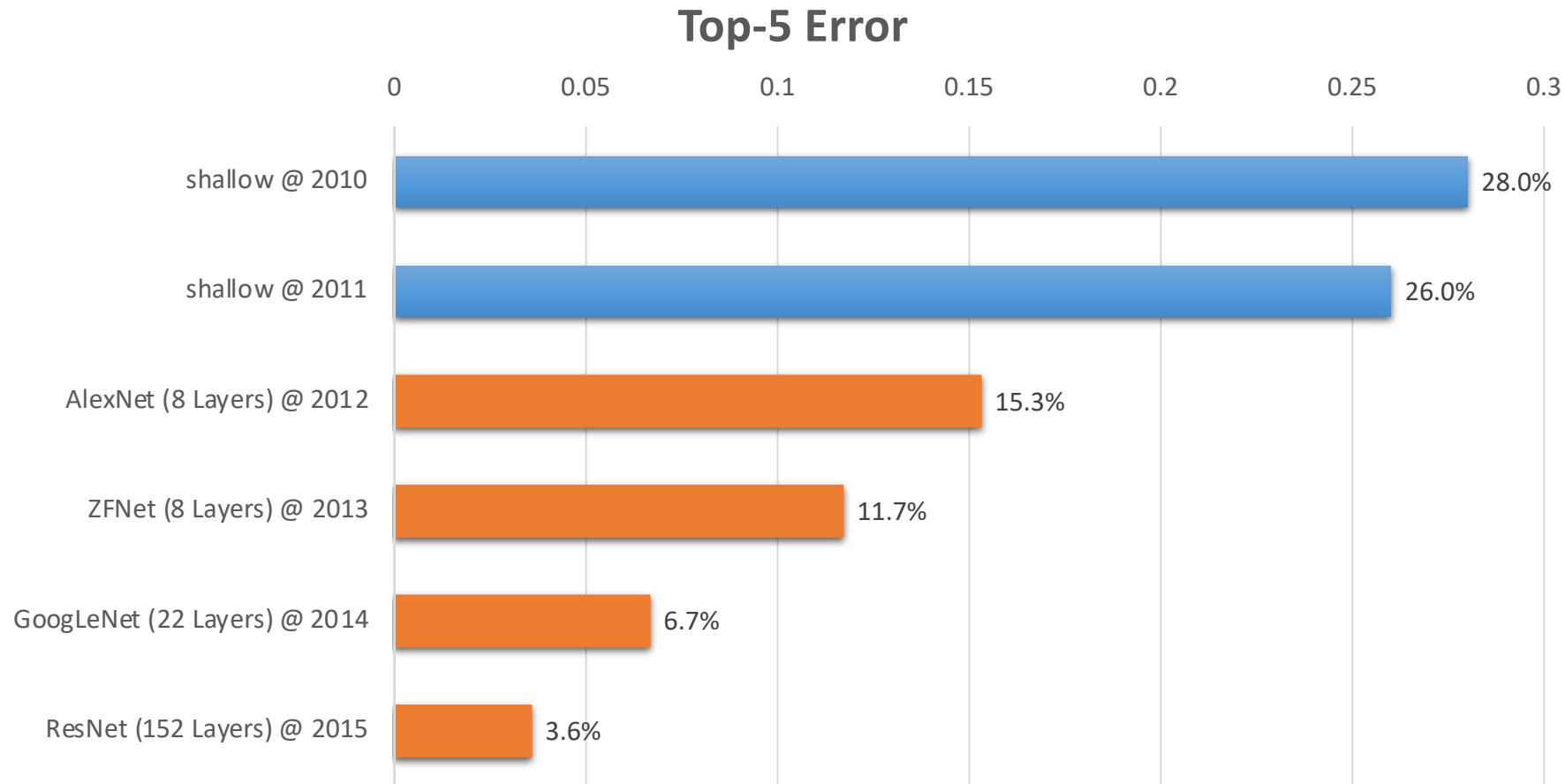
MSRA ResNet



Deep architectures

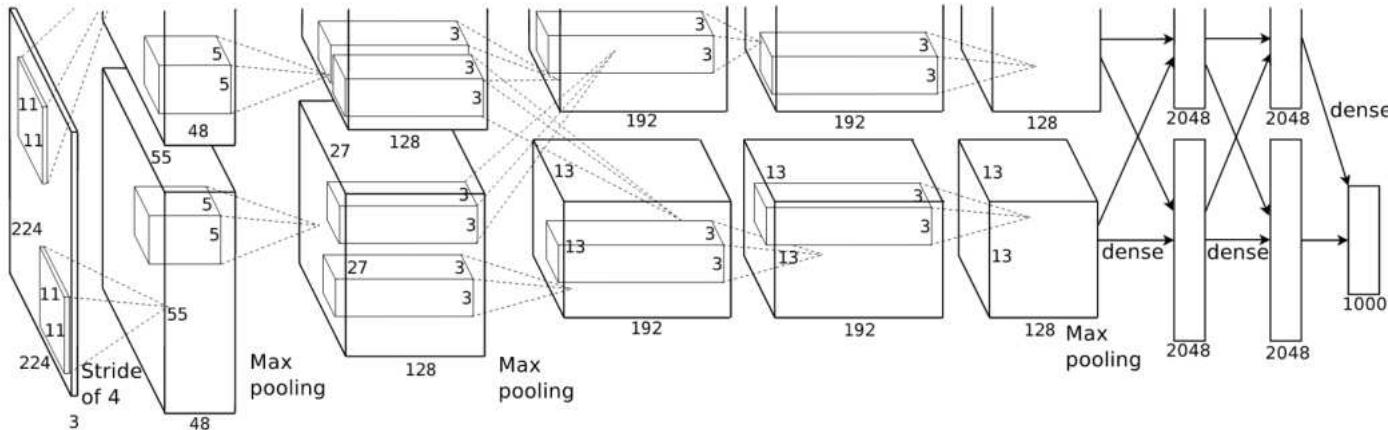
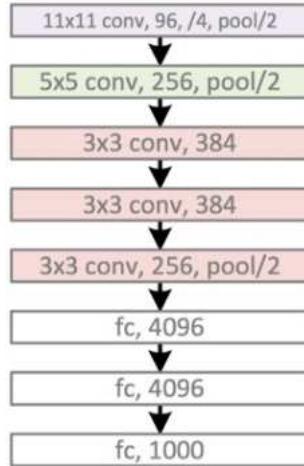


Performance of previous years on ImageNet



Depth is the key to high classification accuracy.

Deep architectures - AlexNet



- The split (i.e. two pathways) in the image above are the split between two GPUs.
- Trained for about a week on two NVIDIA GTX 580 3GB GPU
- 60 million parameters
- Input layer: size 227x227x3
- 8 layers deep: 5 convolution and pooling layers and 3 fully connected layers

2012

2013

2014

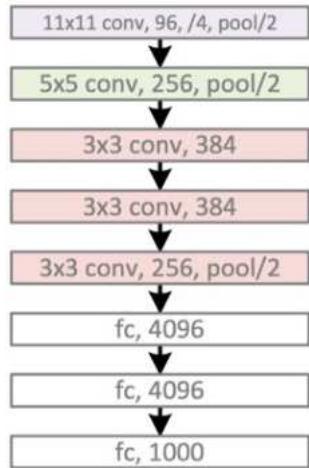
2015

2016

2017

2018 ...

Deep architectures - AlexNet



96 kernels learned by first convolution layer; 48 kernels were learned by each GPU

2012

2013

2014

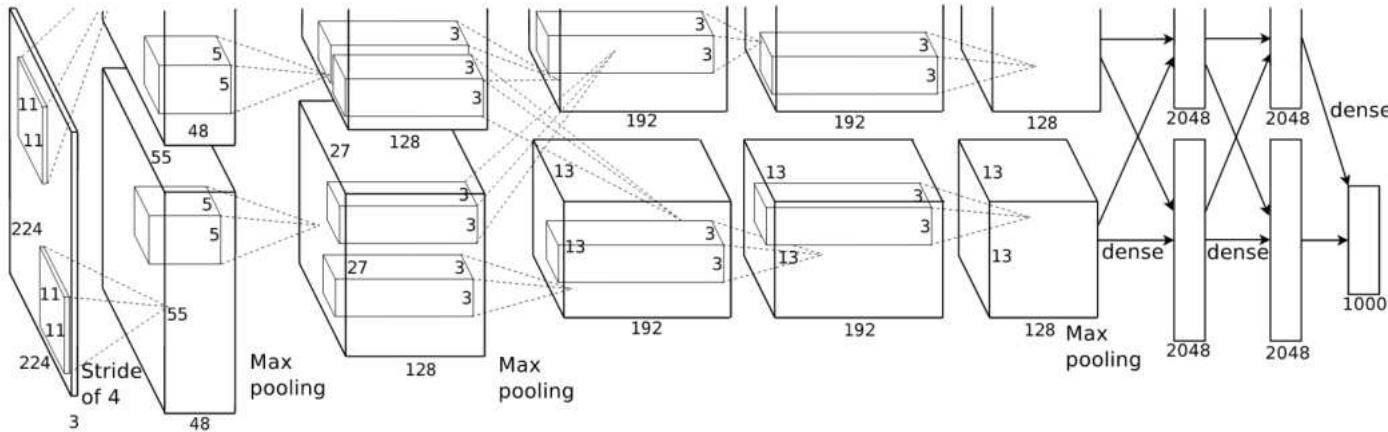
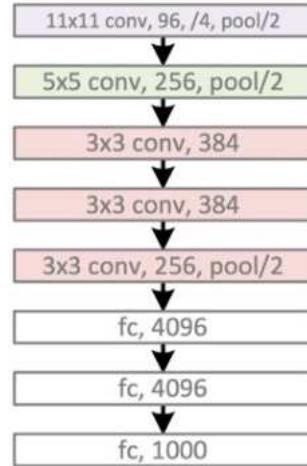
2015

2016

2017

2018 ...

Deep architectures - AlexNet



- Escape from a few layers
 - ReLU nonlinearity for solving gradient vanishing
 - Data augmentation
 - Dropout
 - Outperformed all previous models on ILSVRC by 10%

2012

2013

2014

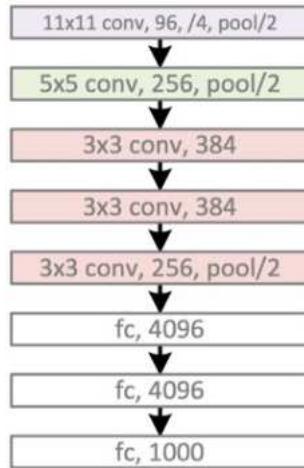
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	
L2 (Conv)	
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

← First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels

2012

2013

2014

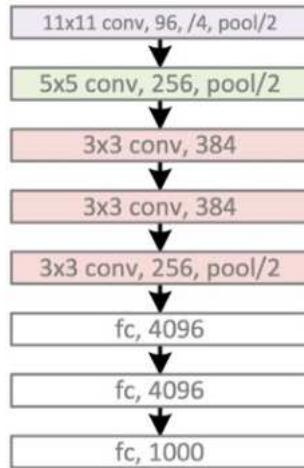
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

First convolution layer: 96 kernels of size $11 \times 11 \times 3$, with a stride of 4 pixels

Number of parameters = $(11 \times 11 \times 3 + 1) * 96 = 34,944$

Note: There are no parameters associated with a pooling layer. The pool size, stride, and padding are hyperparameters.

2012

2013

2014

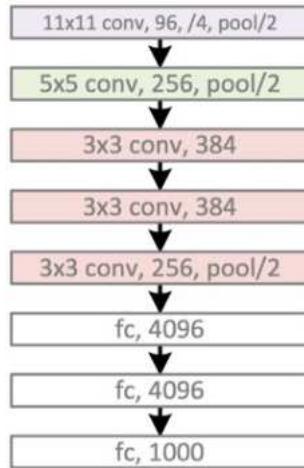
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

← Second convolution layer: 256 kernels of size 5x5x96

2012

2013

2014

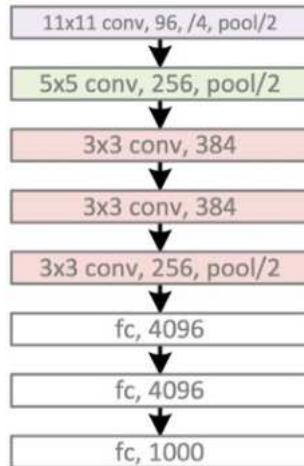
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

← Second convolution layer: 256 kernels of size 5x5x96

Number of parameters = $(5 \times 5 \times 96 + 1) * 256 = 614,656$

2012

2013

2014

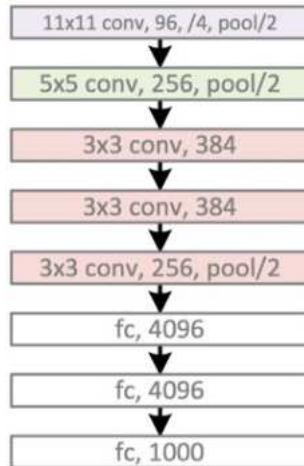
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

← First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256
Size (width) of the output image of the previous Conv Layer = 6

2012

2013

2014

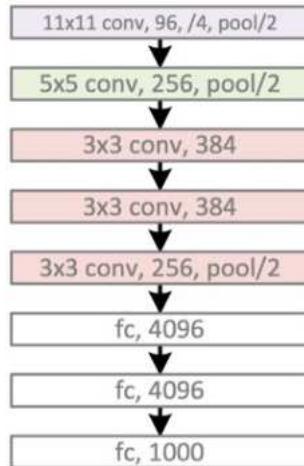
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

← First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256
Size (width) of the output image of the previous Conv Layer = 6

Number of parameters = $(6 \times 6 \times 256 \times 4096) + 4096 = 37,752,832$

2012

2013

2014

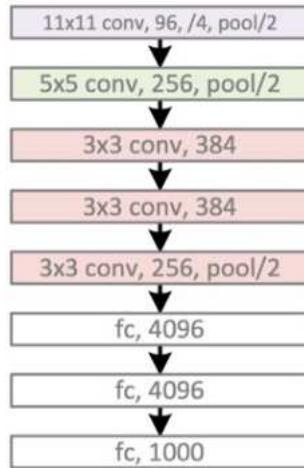
2015

2016

2017

2018 ...

Deep architectures - AlexNet



Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

The last FC layer:

Number of neurons = 1000

Number of neurons in the previous FC Layer = 4096

Number of parameters = $(1000 * 4096) + 1000 = 4,097,000$

2012

2013

2014

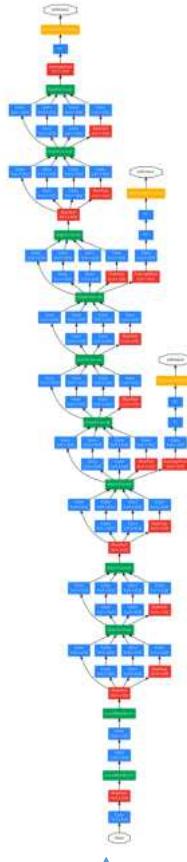
2015

2016

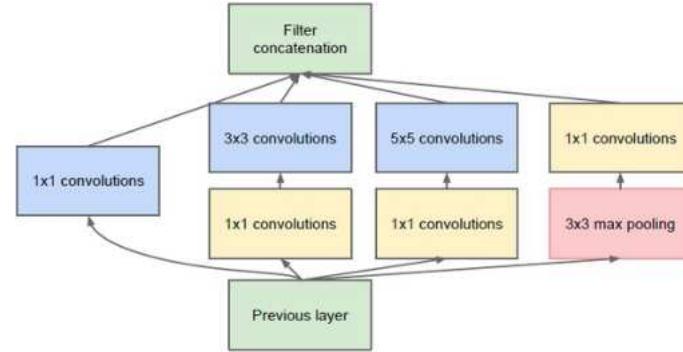
2017

2018 ...

Deep architectures - GoogLeNet



- An important lesson - go deeper
- Inception structures (v2, v3, v4)
 - Reduce parameters (4M vs 60M in AlexNet)



The 1x1 convolutions are performed to reduce the dimensions of input/output

- Batch normalization
 - Normalization the activation for each training mini-batch
 - Allows us to use much higher learning rates and be less careful about initialization

2012

2013

2014

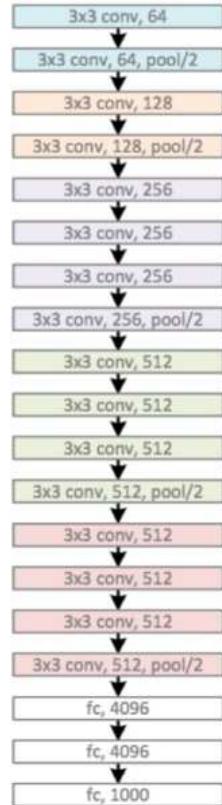
2015

2016

2017

2018 ...

Deep architectures - VGG



- An important lesson - go deeper
- 140M parameters
- Now commonly used for computing perceptual loss

2012

2013

2014

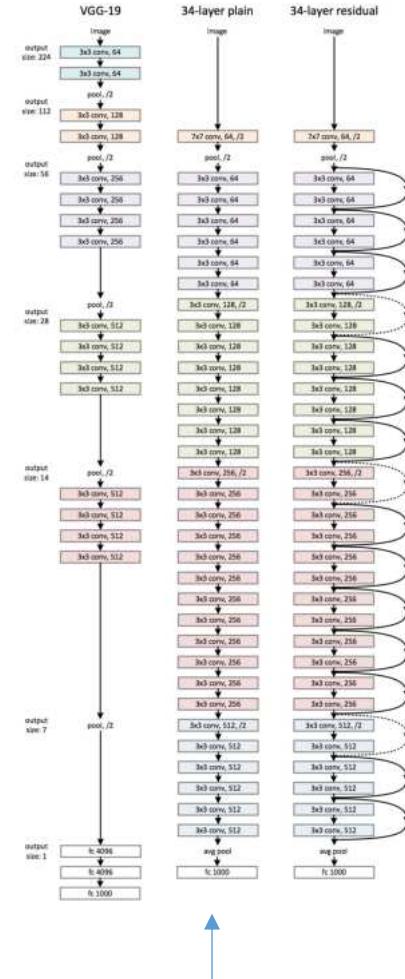
2015

2016

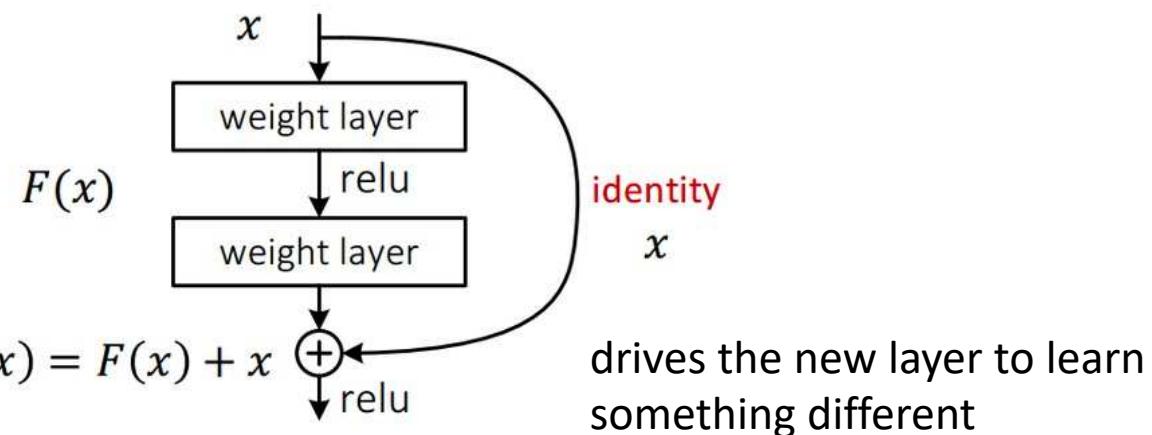
2017

2018 ...

Deep architectures - ResNet



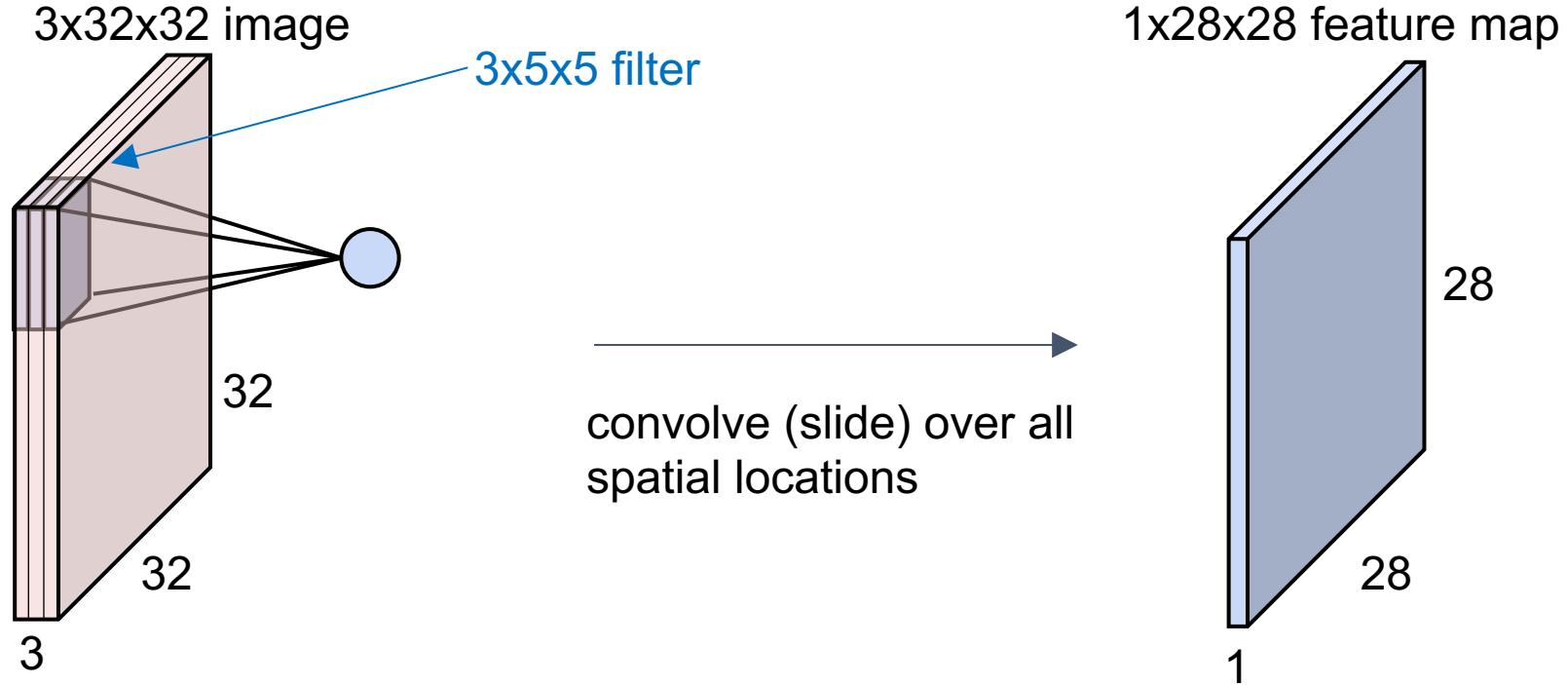
- An important lesson - go deeper
 - Escape from 100 layers
 - Residual learning



drives the new layer to learn something different

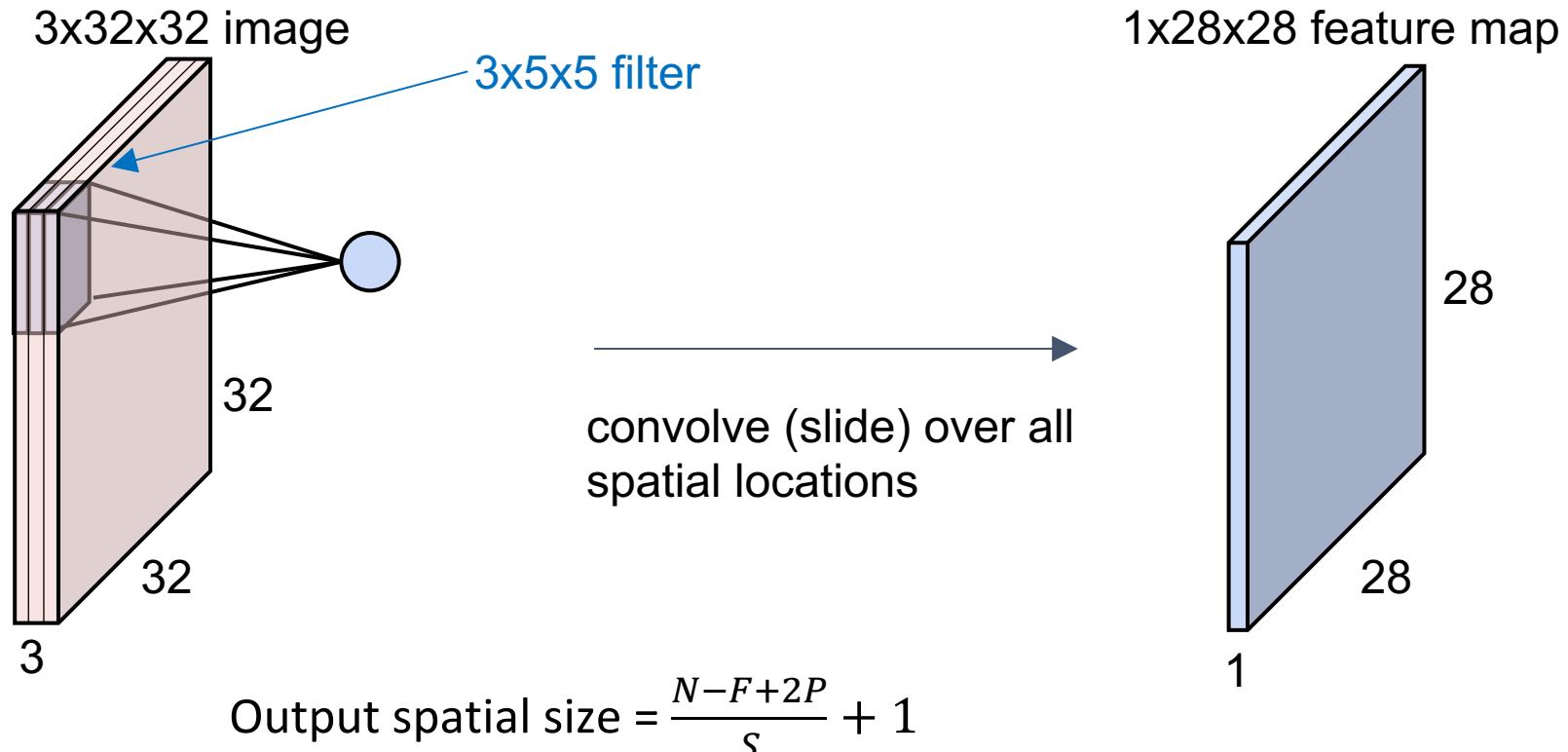
More on Convolution

Standard convolution



- 😊 How to calculate the spatial size of output? [Lecture 6](#)
- 😊 How to calculate the number of parameters? [Lecture 6](#)
- 😅 How to calculate the computations involved?

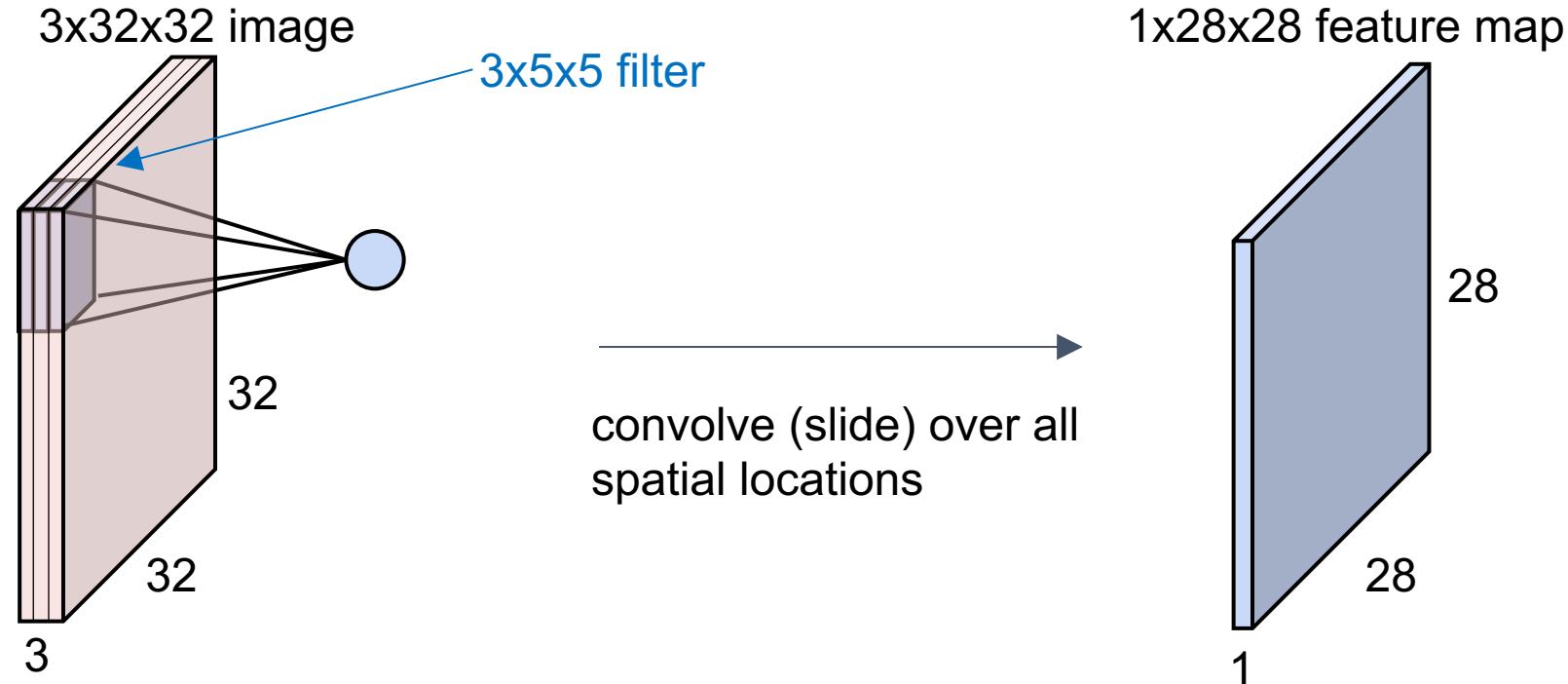
Recap: How to calculate the spatial size of output?



In this example, $N = 32, F = 5, P = 0, S = 1$

$$\text{Thus, output spatial size} = \frac{32-5+2(0)}{1} + 1 = 28$$

Recap: How to calculate the number of parameters?



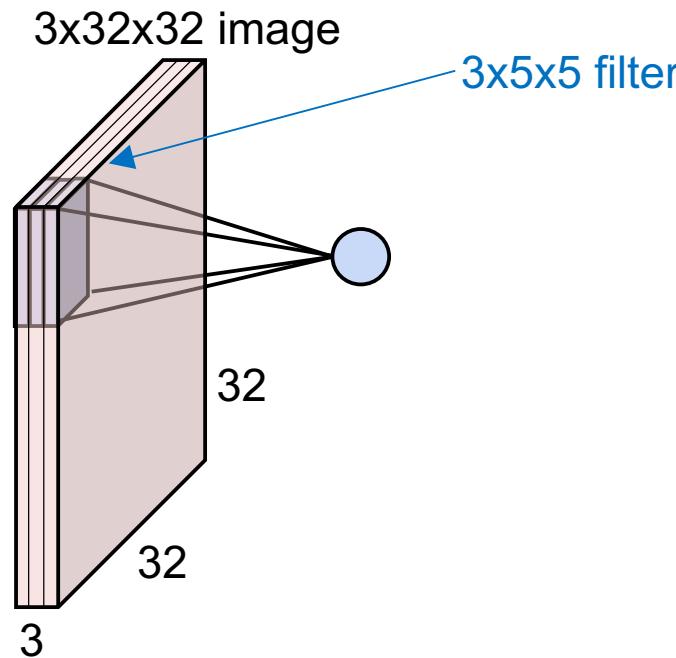
Let say we have **ten** $3 \times 5 \times 5$ filters with stride **1**, pad **0**

Recap: How to calculate the number of parameters?

Input volume: $3 \times 32 \times 32$

Ten $3 \times 5 \times 5$ filters with stride 1, pad 0

Number of parameters in this layer: ?



Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)

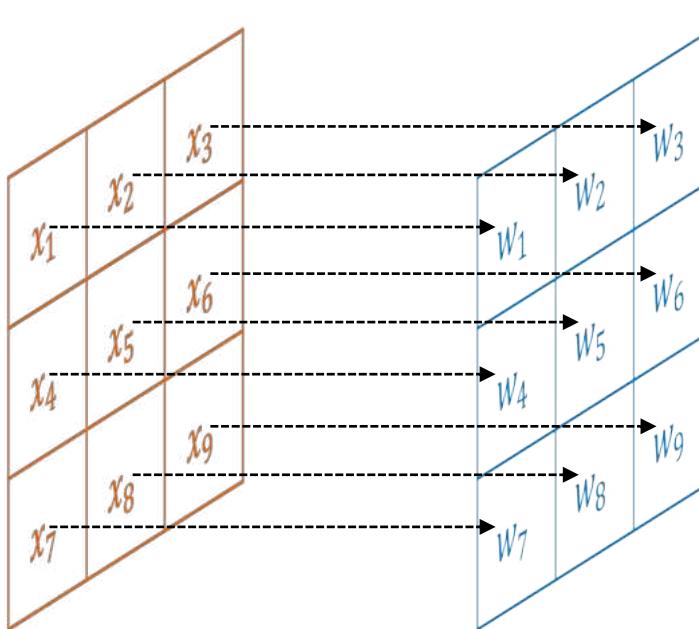
Ten filters so $76 \times 10 = 760$ parameters

How to calculate the computations involved?

Let's focus on one input channel first

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i + l, j + m)w(l, m) + b$$

Element-wise multiplication of input and weights



$x_1 w_1$

$x_2 w_2$

$x_3 w_3$

$x_4 w_4$

$x_5 w_5$

$x_6 w_6$

$x_7 w_7$

$x_8 w_8$

$x_9 w_9$

How many multiplication operations?

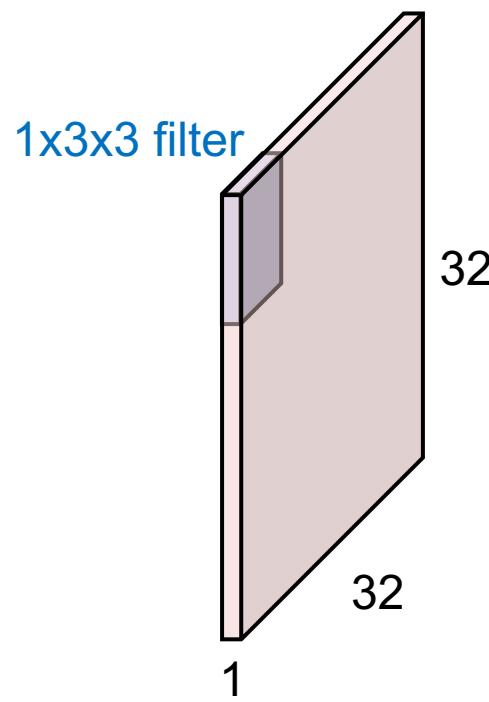
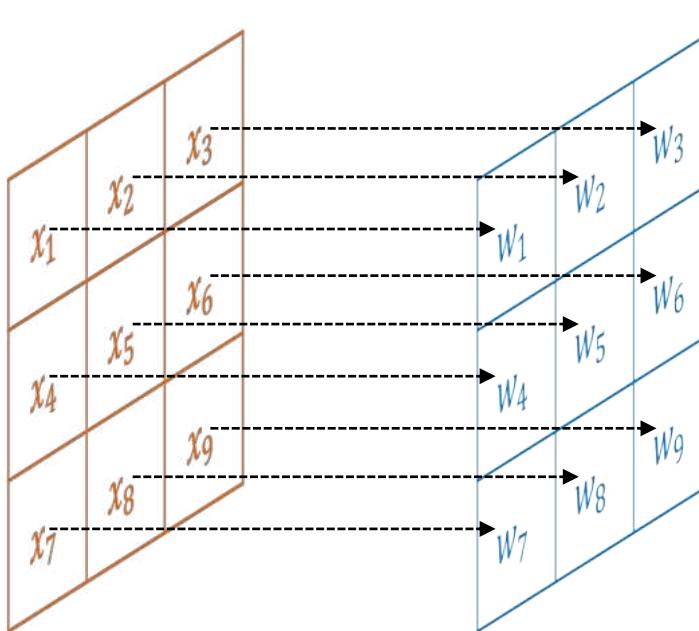
In general, there are F^2 multiplication operations, where F is the filter spatial size

How to calculate the computations involved?

Let's focus on one input channel first

$$u(i, j) = \sum_{l=-a}^a \sum_{m=-b}^b x(i + l, j + m)w(l, m) + b$$

Element-wise multiplication of input and weights



$x_1 w_1$

+

$x_2 w_2$

+

$x_3 w_3$

+

$x_4 w_4$

+

$x_5 w_5$

+

$x_6 w_6$

+

$x_7 w_7$

+

$x_8 w_8$

+

$x_9 w_9$

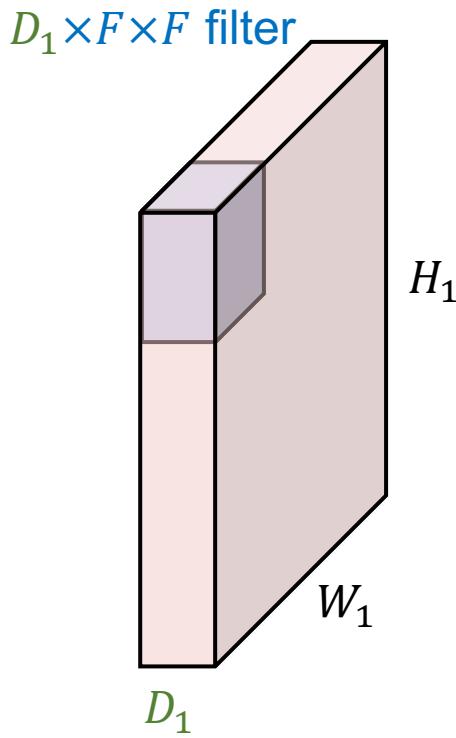
How many adding operations?

Adding the elements after multiplication, we need $n - 1$ adding operations for n elements

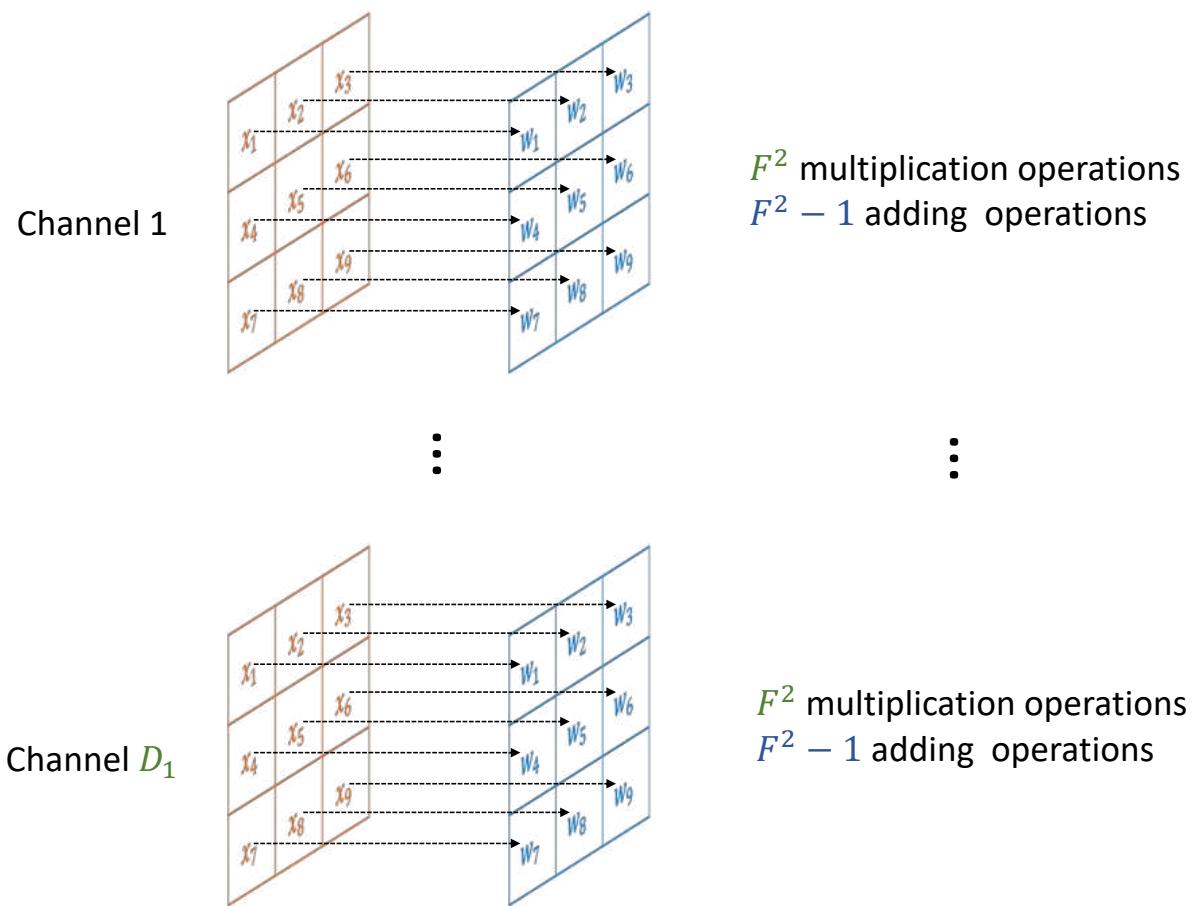
In general, there are $F^2 - 1$ adding operations, where F is the filter spatial size

How to calculate the computations involved?

Let's say we have D_1 input channels now



Element-wise multiplication of input and weights, for each channel



If we have only **one filter**, and apply it to generate **output of one spatial location**, the total operations

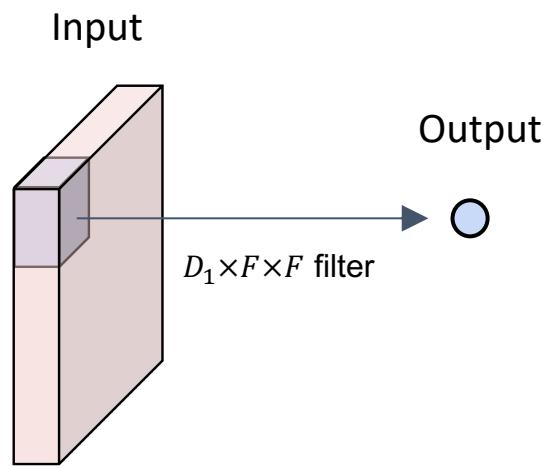
The total cost is

$$(D_1 \times F^2) + (D_1 \times F^2 - 1)$$

Let's not forget to add the **bias**

$$(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$$

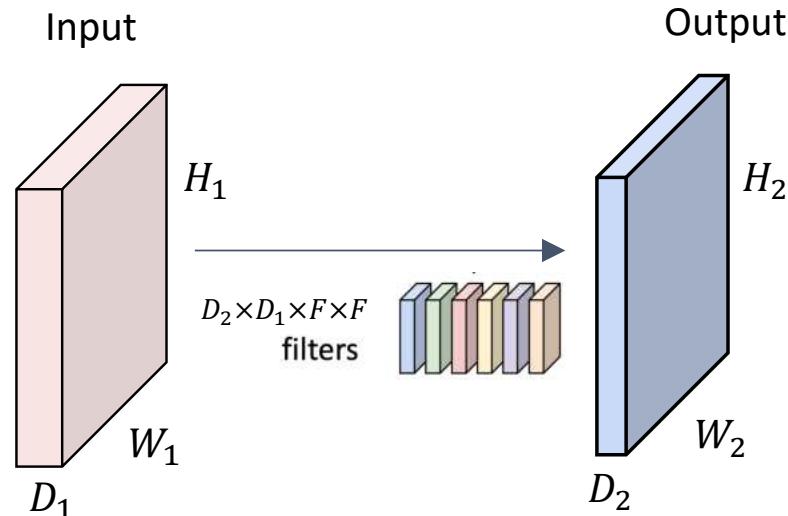
How to calculate the computations involved?



If we have **only one filter**, and apply it to generate **output of one spatial location**, the total operations

The total cost is

$$(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1$$



If we have **D_2 filter**, and apply it to generate **output of $H_2 \times W_2$ spatial location**, the total operations

The total cost is

$$\begin{aligned} & [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2 \\ & = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2 \end{aligned}$$

Summary

FLOPs (floating point operations)

Not FLOPS (floating point operations per second)

Assume:

- Filter size F
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Produces a output volume of size $D_2 \times H_2 \times W_2$

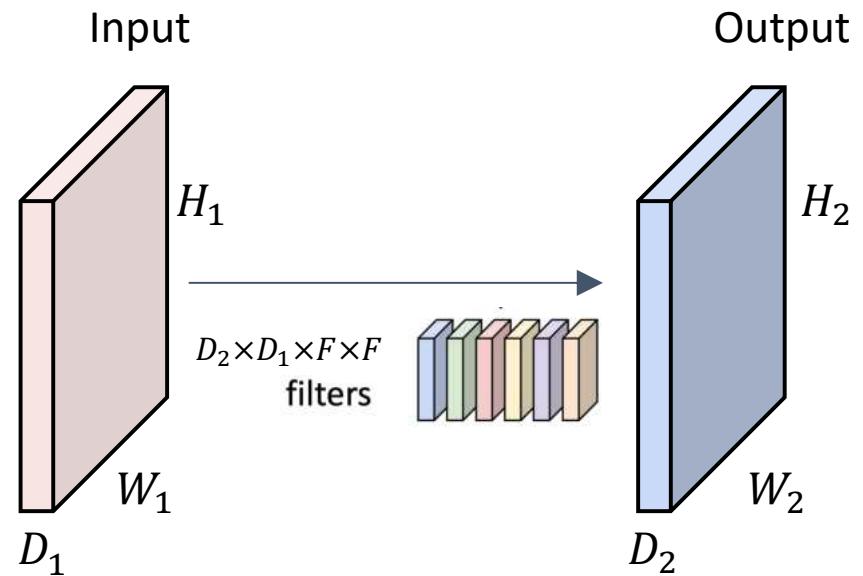
The FLOPs of the convolution layer is given by

$$\text{FLOPs} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2 = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

Elementwise multiplication of each filter on a spatial location

Adding the elements after multiplication, we need $n - 1$ adding operations for n elements

Add the bias

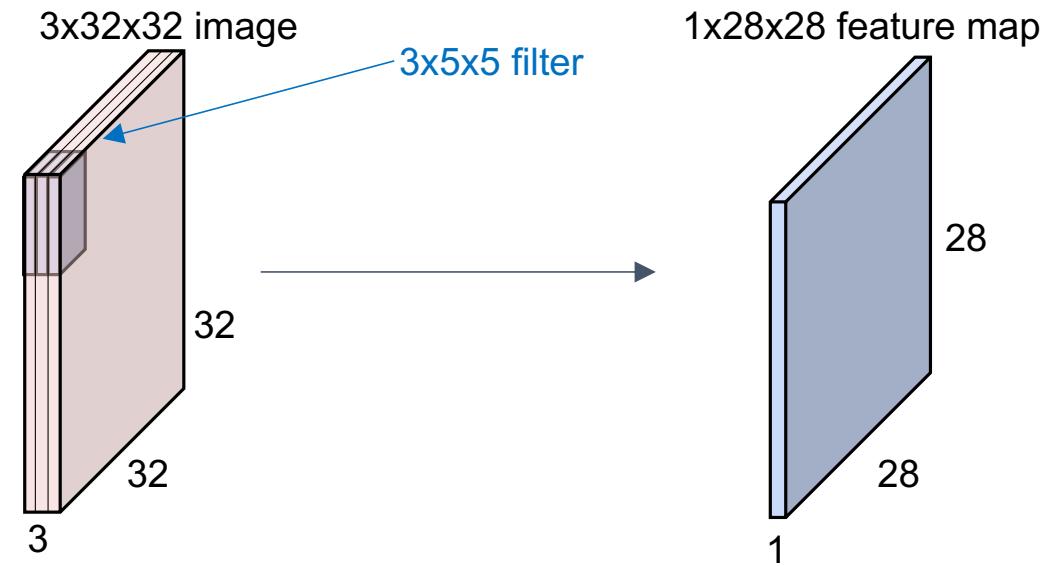


Try this

Assume:

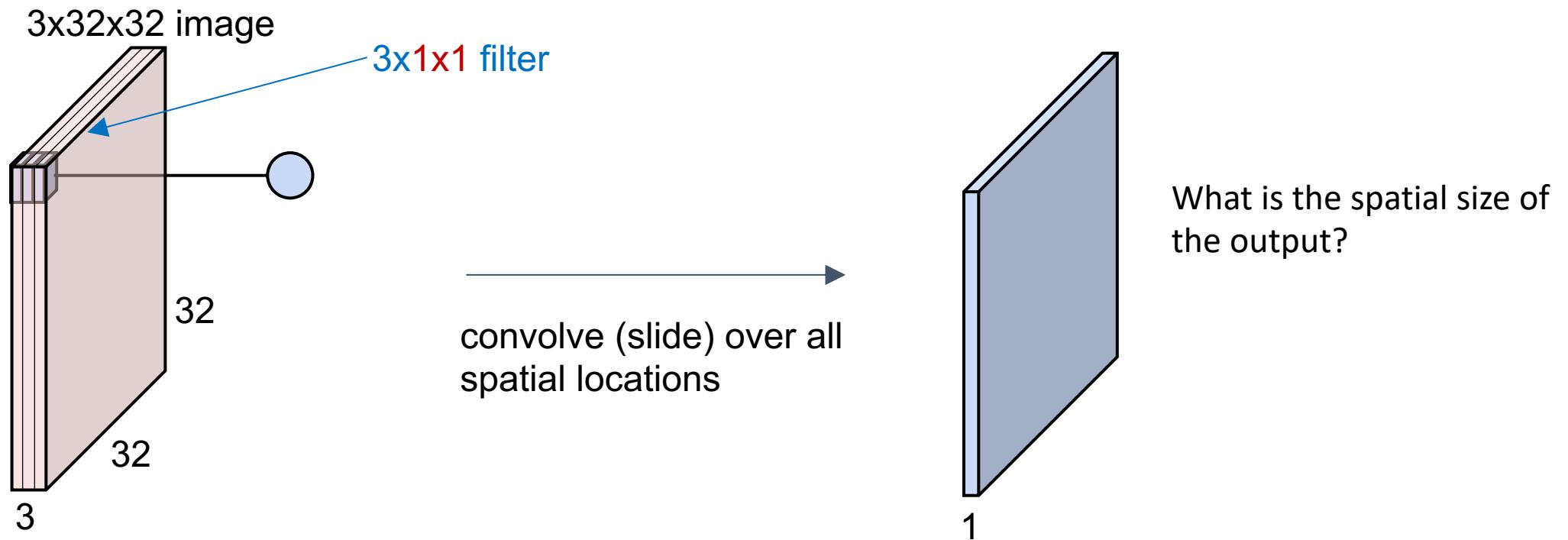
- One filter with spatial size of 5×5
- Accepts a volume of size $3 \times 32 \times 32$
- Produces a output volume of size $1 \times 28 \times 28$

The FLOPs of the convolution layer is given by?



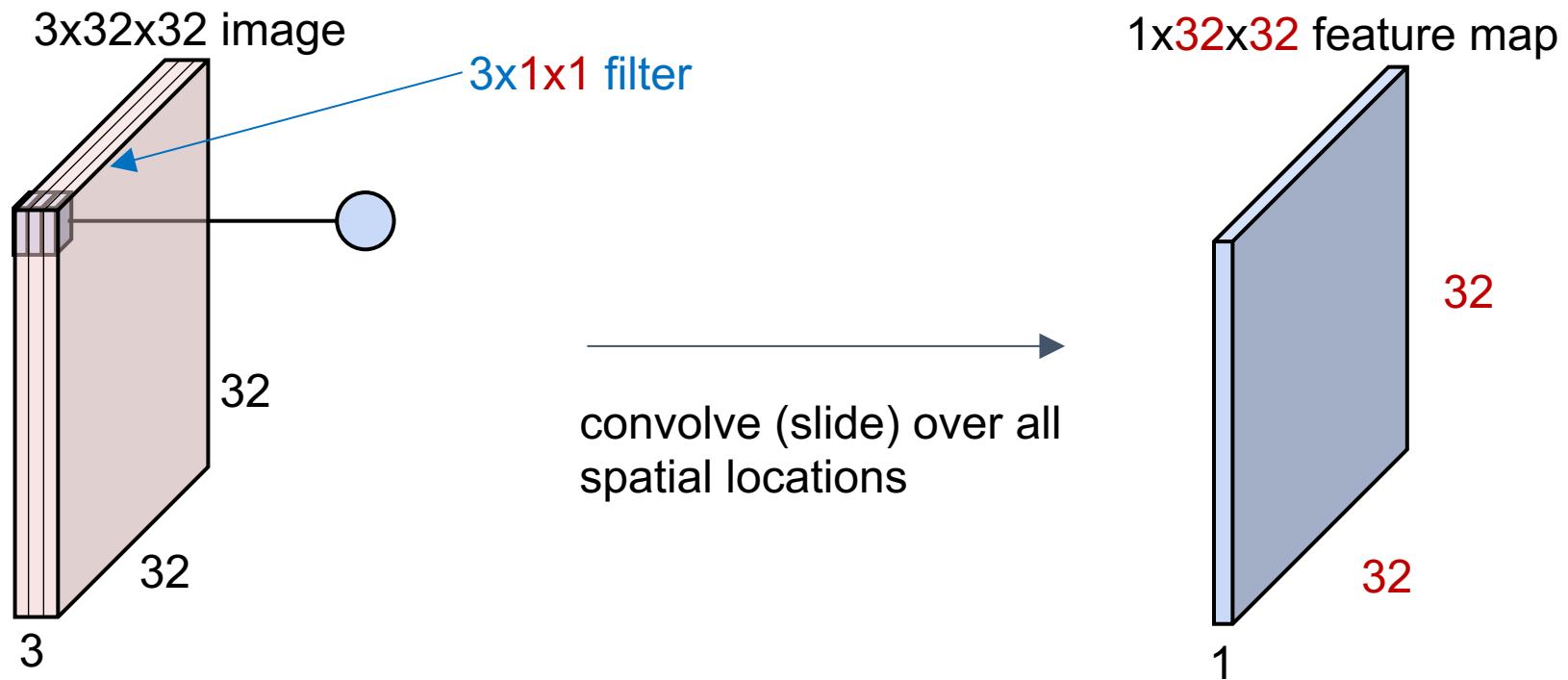
Pointwise convolution

- We have seen convolution with spatial size of 3×3 , 5×5 , 7×7
 - Can we have other sizes?
 - Can we have filter of spatial size 1×1 ?



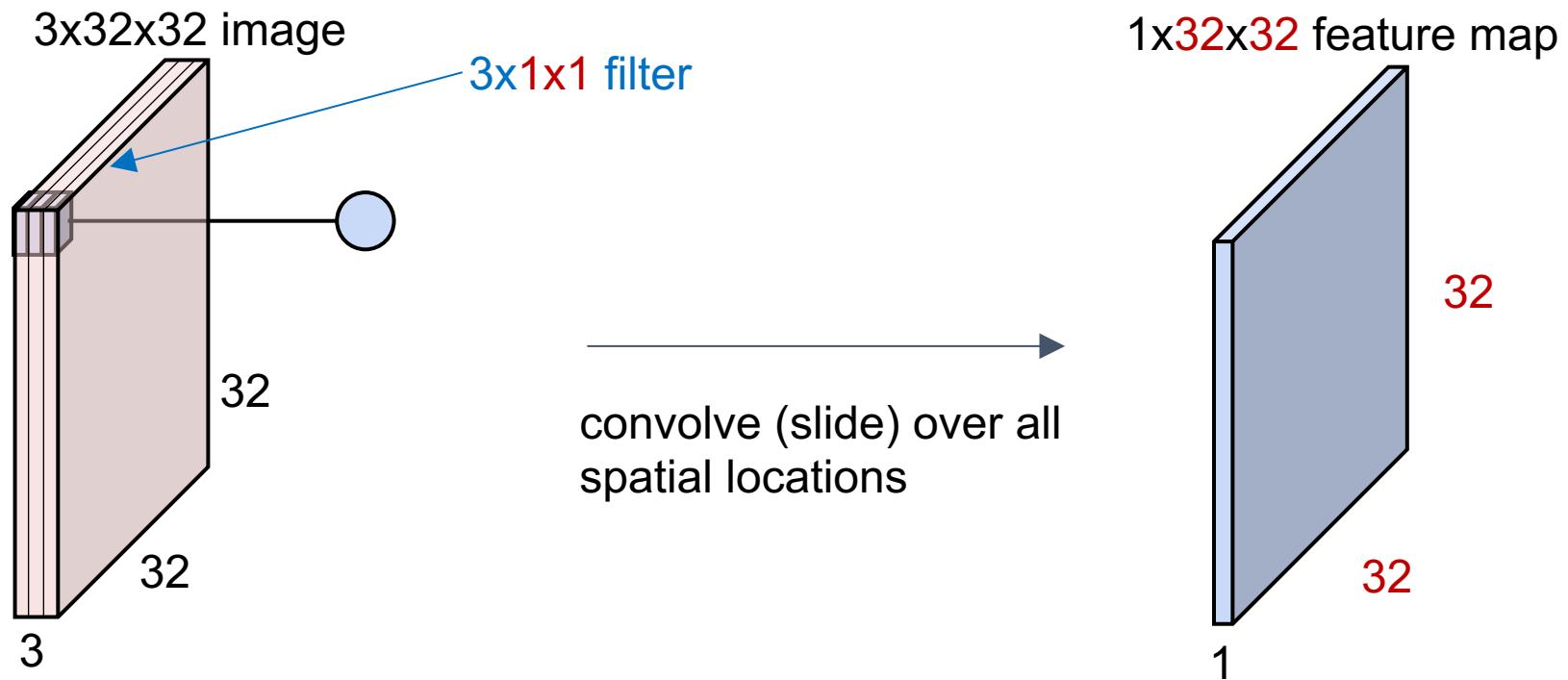
Pointwise convolution

- We have seen convolution with spatial size of 3×3 , 5×5 , 7×7
 - Can we have other sizes?
 - Can we have filter of spatial size 1×1 ?



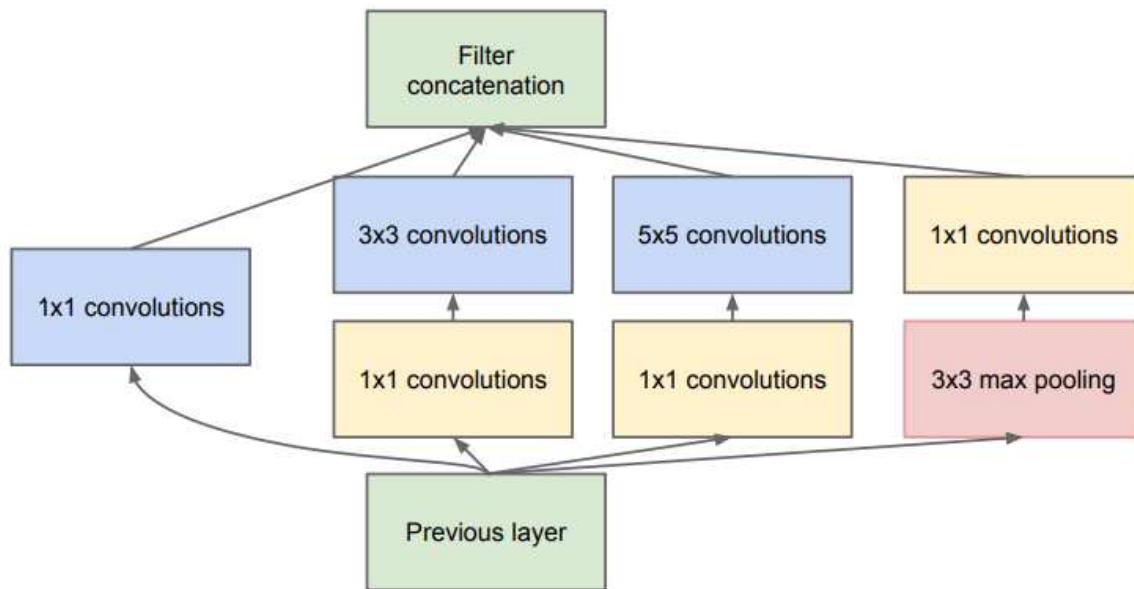
Pointwise convolution

- Why having filter of spatial size 1×1 ?
 - Change the size of channels
 - “Blend” information among channels by linear combination



Pointwise convolution

- A real-world example
 - 1×1 convolutions are used for compute reductions before the expensive 3×3 and 5×5 convolutions



E.g., Given the output from the previous layer, reduce the number of channels of from $256 \times 32 \times 32$ to $128 \times 32 \times 32$ before the 5×5 convolutions

Referring to the FLOPs equation, reducing input channels help reduce the FLOPs

$$\text{FLOPs} = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

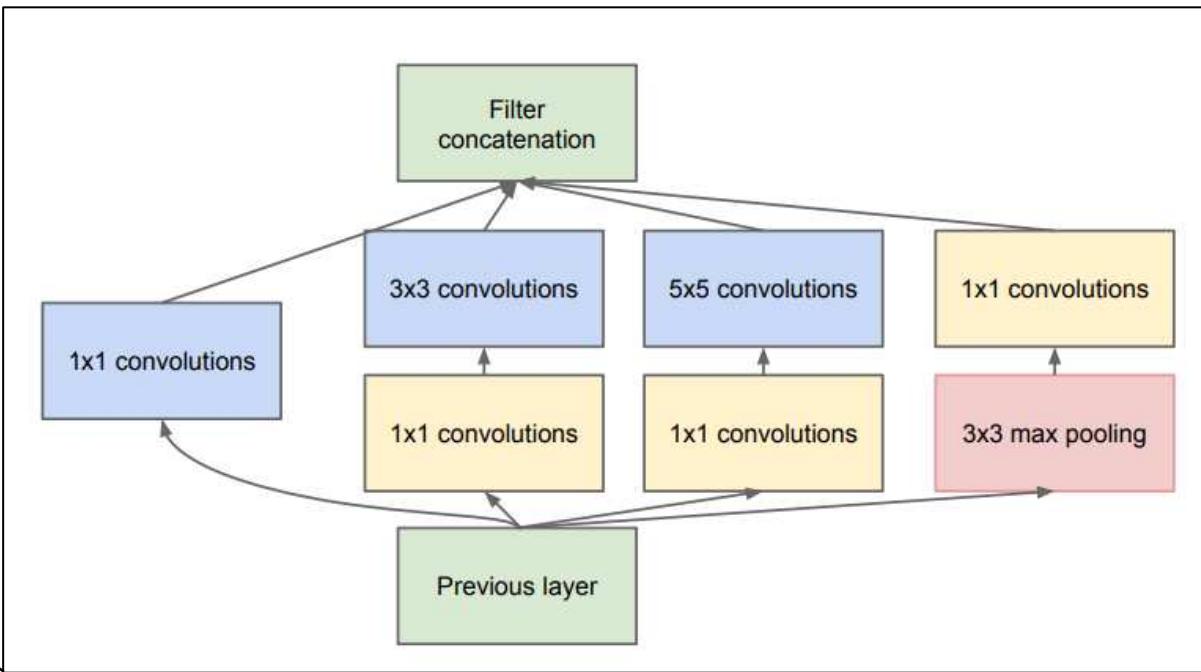
Pointwise convolution



This is known as **inception structure** used in **GoogLeNet**

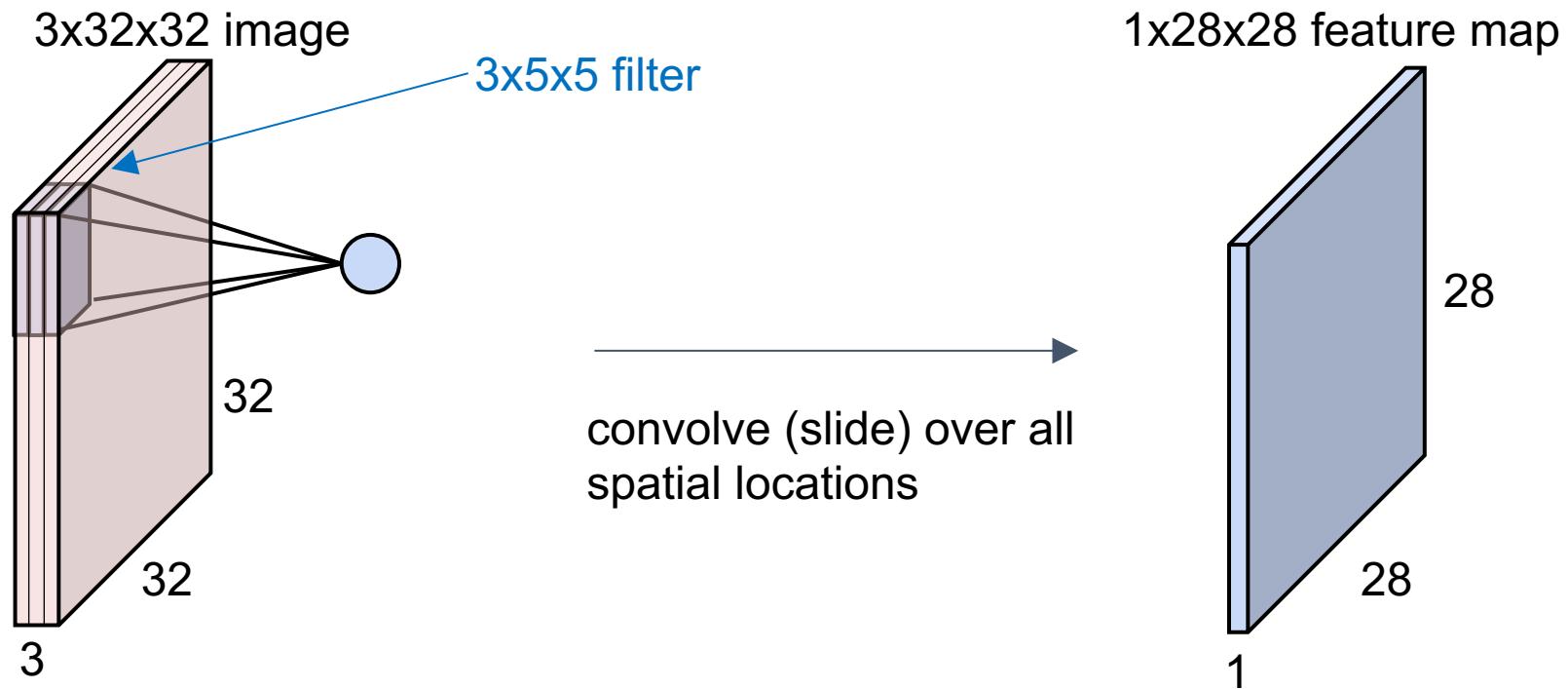
Proposed by Google in 2014, winning the ImageNet competition that year

4M parameters vs 60M parameters of AlexNet



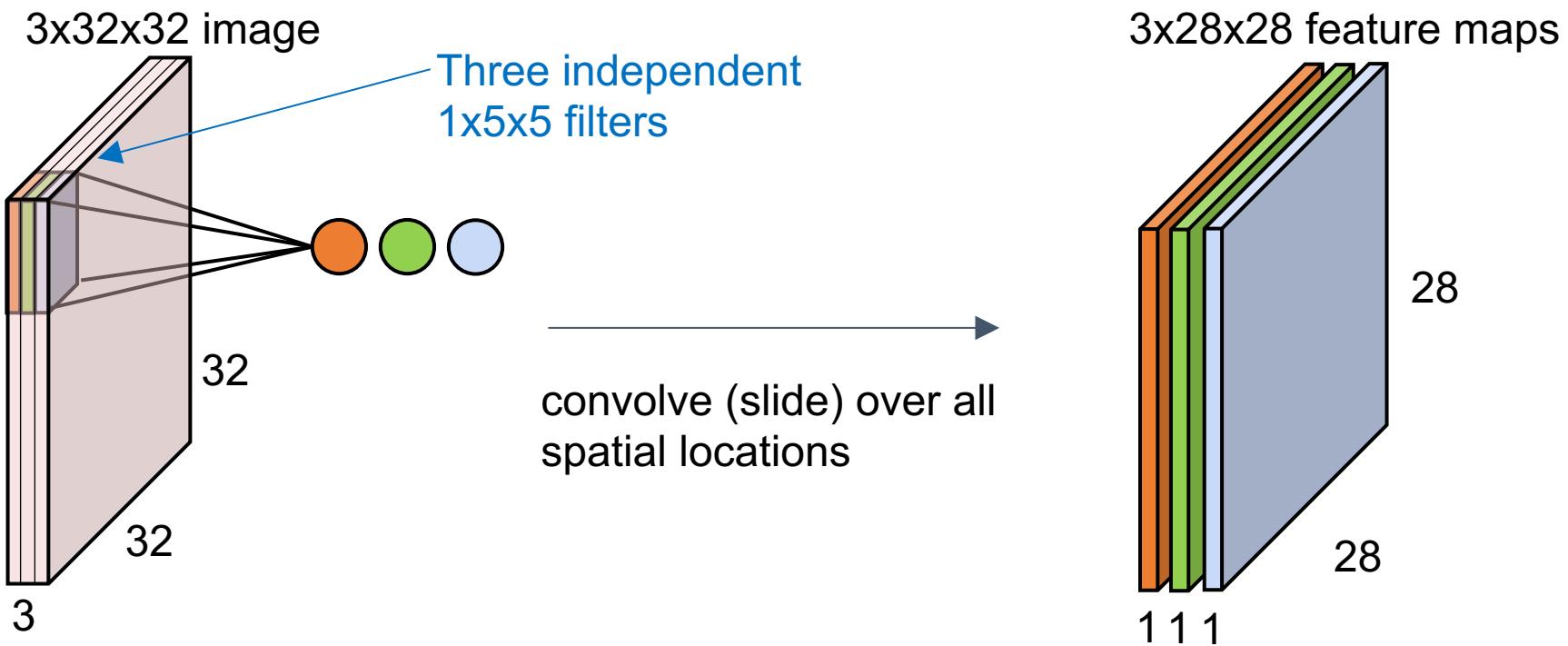
Standard convolution

- Standard convolution
 - The input and output are locally connected in spatial domain
 - In **channel** domain, they are **fully connected**

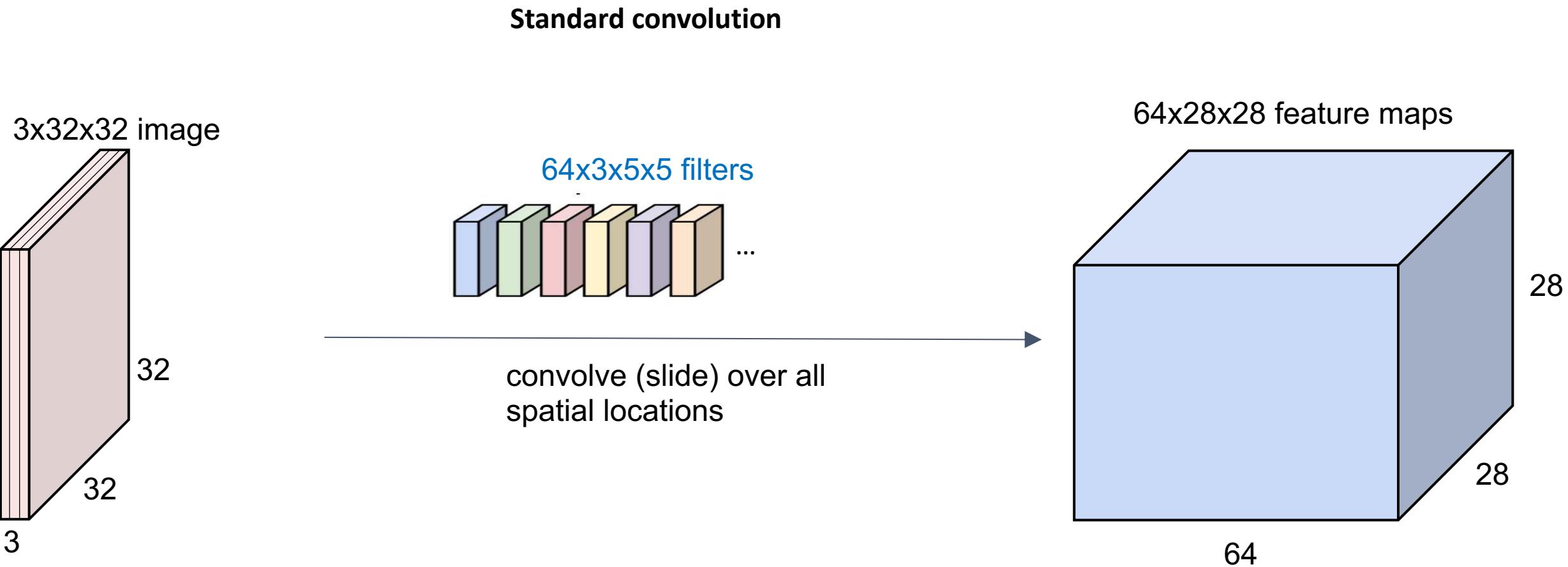


Depthwise convolution

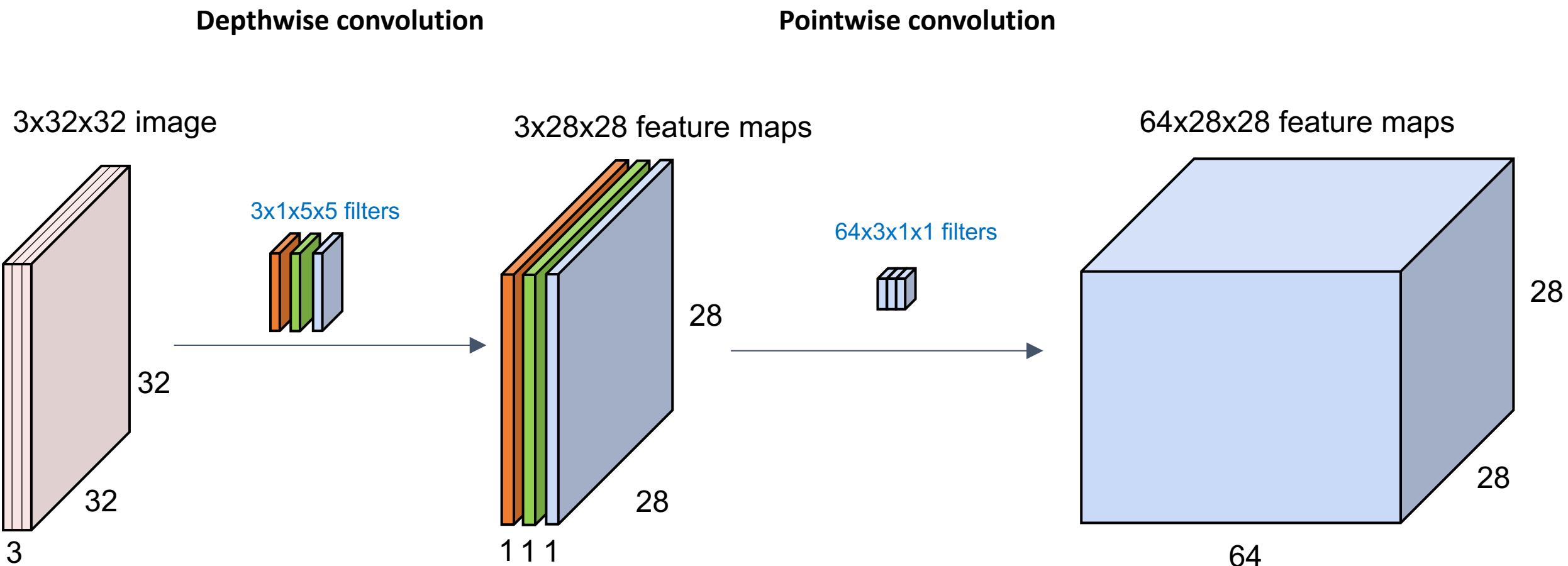
- Depthwise convolution
 - Convolution is performed **independently** for each of input channels



Standard convolution



Depthwise convolution + Pointwise convolution

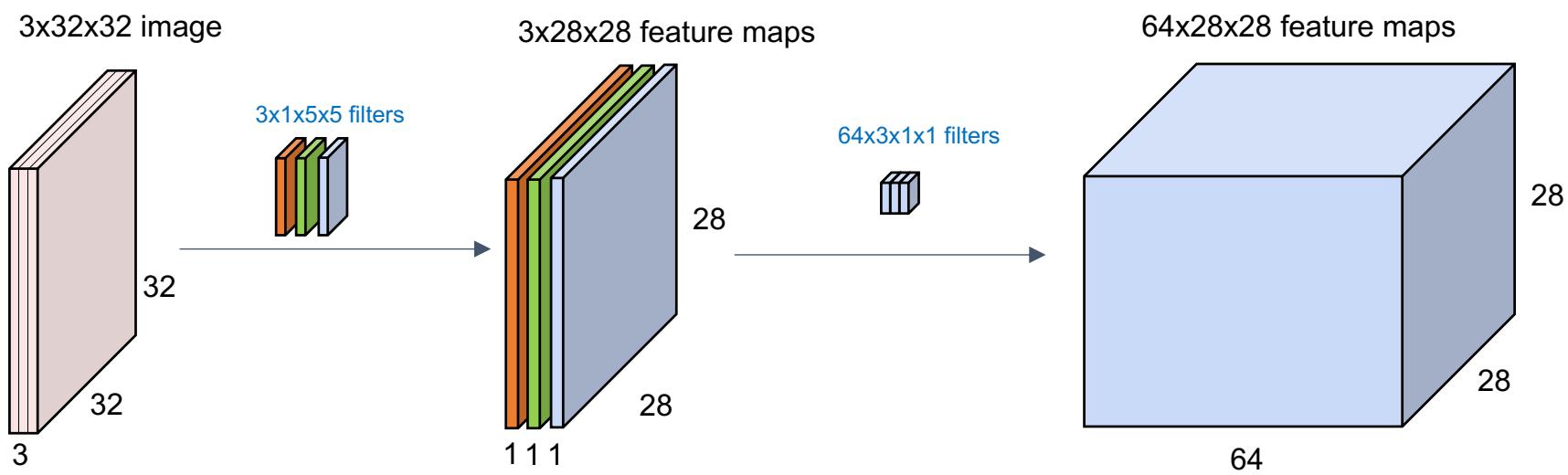
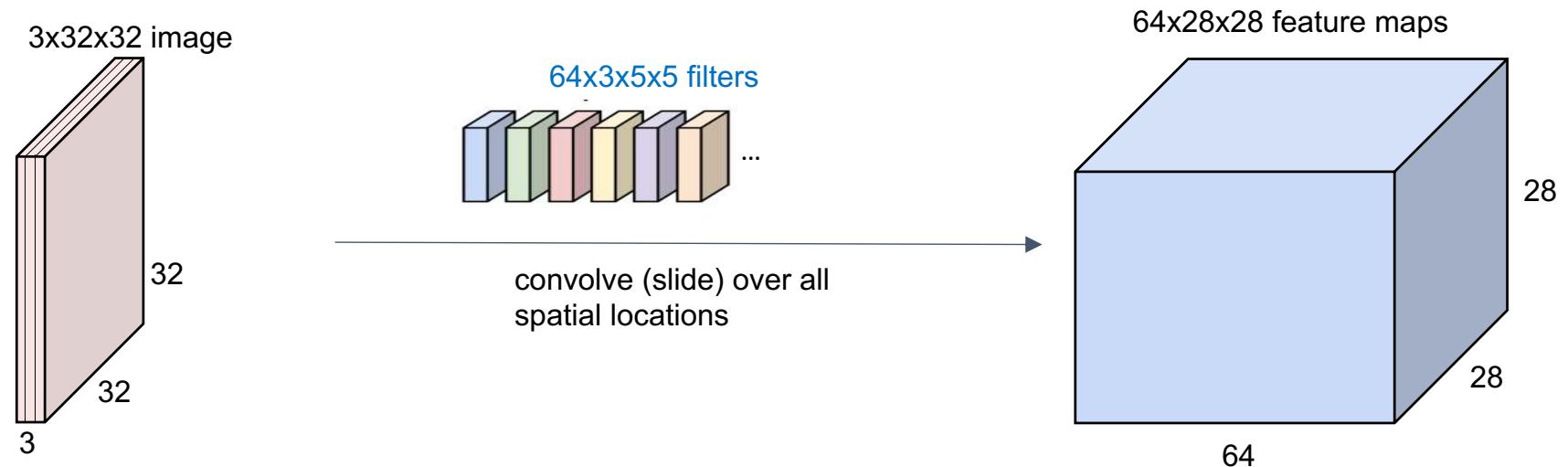


Depthwise convolution + Pointwise convolution

Replace standard convolution
with

Depthwise convolution +
pointwise convolution

And we still get the same
size of output volume!



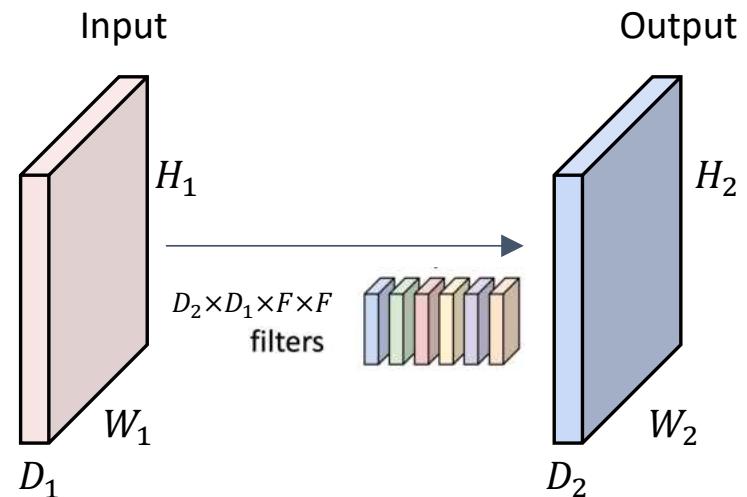
Depthwise convolution + Pointwise convolution

- Why replacing standard convolution with depthwise convolution + pointwise convolution?
 - The computational cost reduction rate is roughly $1/8\text{--}1/9$ at only a small reduction in accuracy
 - Good if you want to deploy small networks on devices with CPU
 - Used in network such as MobileNet

Depthwise convolution + Pointwise convolution

- Standard convolution cost

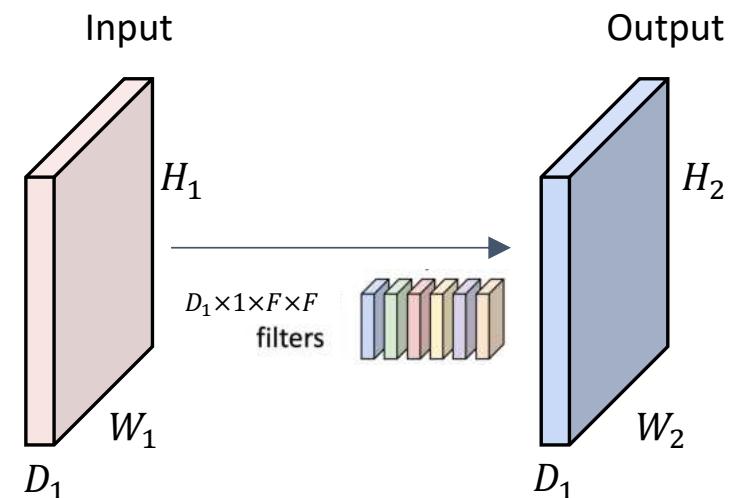
$$\text{FLOPs}_{\text{standard}} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$



- Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = [(1 \times F^2) + (1 \times F^2 - 1) + 1] \times D_1 \times H_2 \times W_2$$

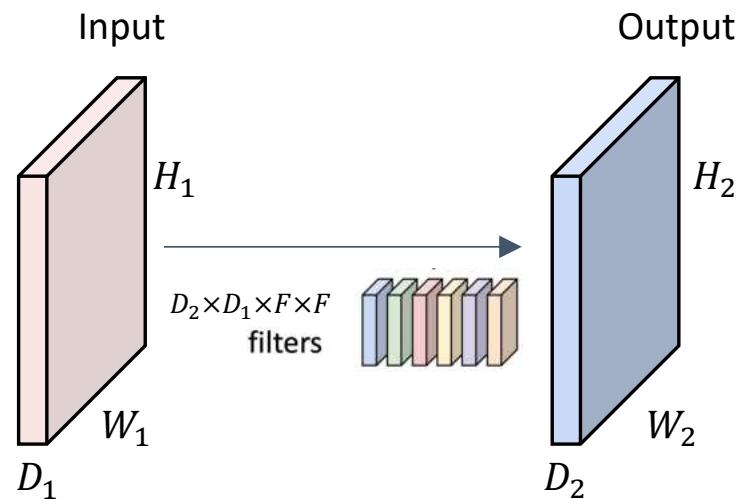
- Each filter is applied only to one channel
- The number of output channels equals to the number of input channels



Depthwise convolution + Pointwise convolution

- Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2$$



- Pointwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = [(D_1 \times 1) + (D_1 \times 1 - 1) + 1] \times D_2 \times H_2 \times W_2$$

- The filter spatial size is 1×1

Depthwise convolution + Pointwise convolution

- Standard convolution cost

$$\text{FLOPs}_{\text{standard}} = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

- Depthwise convolution cost

$$\text{FLOPs}_{\text{depthwise}} = (2 \times F^2) \times D_1 \times H_2 \times W_2$$

- Pointwise convolution cost

$$\text{FLOPs}_{\text{pointwise}} = (2 \times D_1) \times D_2 \times H_2 \times W_2$$

Depthwise convolution + Pointwise convolution

- How much computation do you save by replacing standard convolution with depthwise+pointwise?

$$\text{Reduction} = \frac{\text{Cost of depthwise convolution} + \text{pointwise convolution}}{\text{Cost of standard convolution}}$$

$$\text{Reduction} = \frac{(2 \times F^2) \times D_1 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2} + \frac{(2 \times D_1) \times D_2 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2}$$

$$\text{Reduction} = \frac{1}{D_2} + \frac{1}{F^2}$$

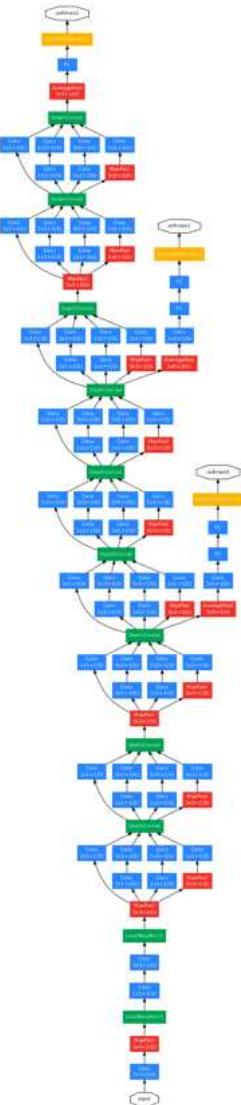
D_2 is usually large. Reduction rate is roughly 1/8–1/9 if 3×3 depthwise separable convolutions are used

Summary

- More on convolutions
 - You learn how to calculate computation complexity of convolutional layer
 - Pointwise convolution can be used to change the size of channels. This can be used to achieve channel reduction and thus saving computational cost
 - Depthwise convolution + Pointwise convolution yields lower computations than standard convolution

Batch Normalization

GoogLeNet



Proposed by Google in 2014, winning the ImageNet competition that year

Apart from the inception structure, there is another important technique called **batch normalization**

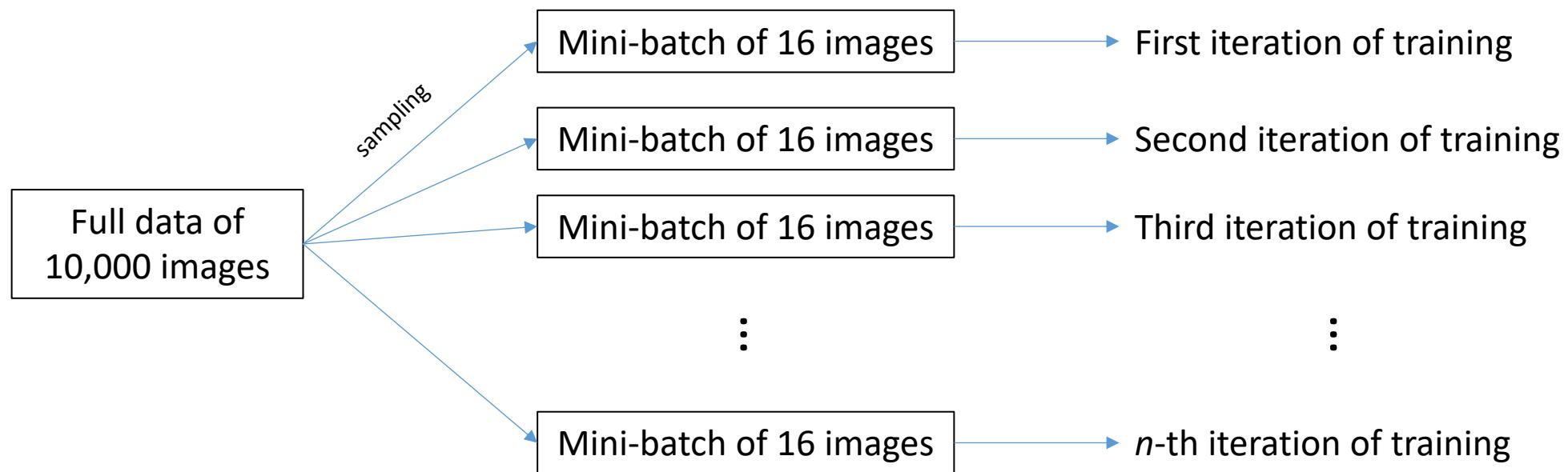
Problem: deep networks are very hard to train!

Main idea: “Normalize” the outputs of a layer so they have **zero mean and unit variance**

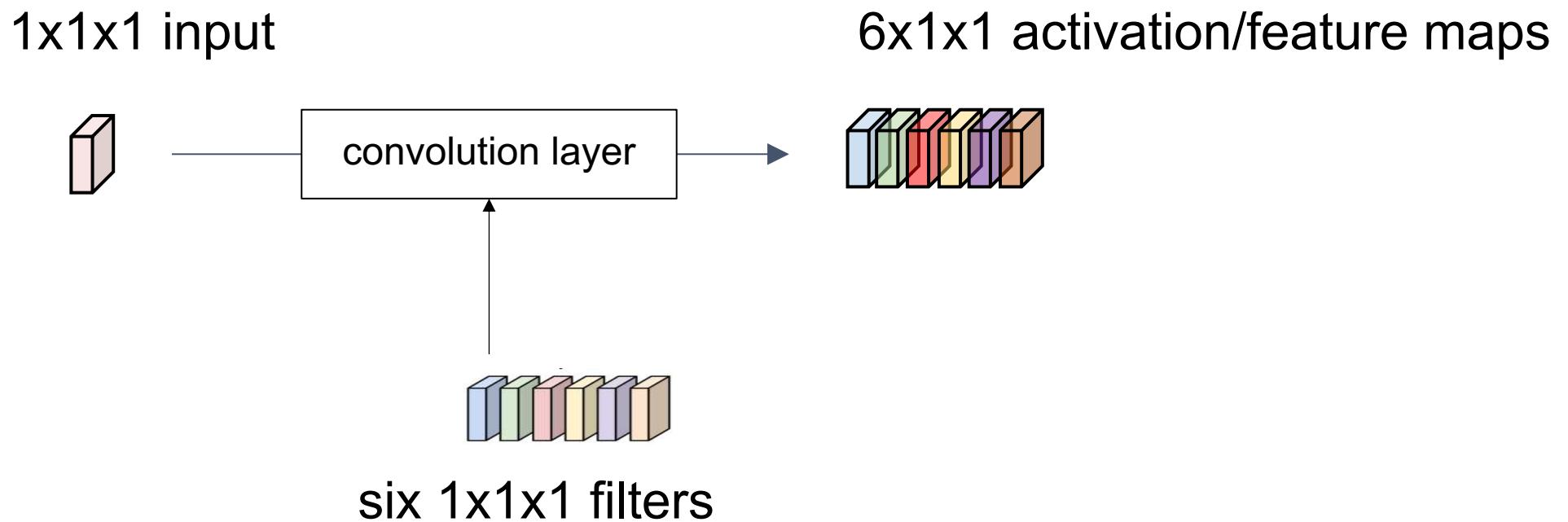
Effect: **allowing higher learning rates** and **reducing the strong dependence on initialization**

Mini-batch

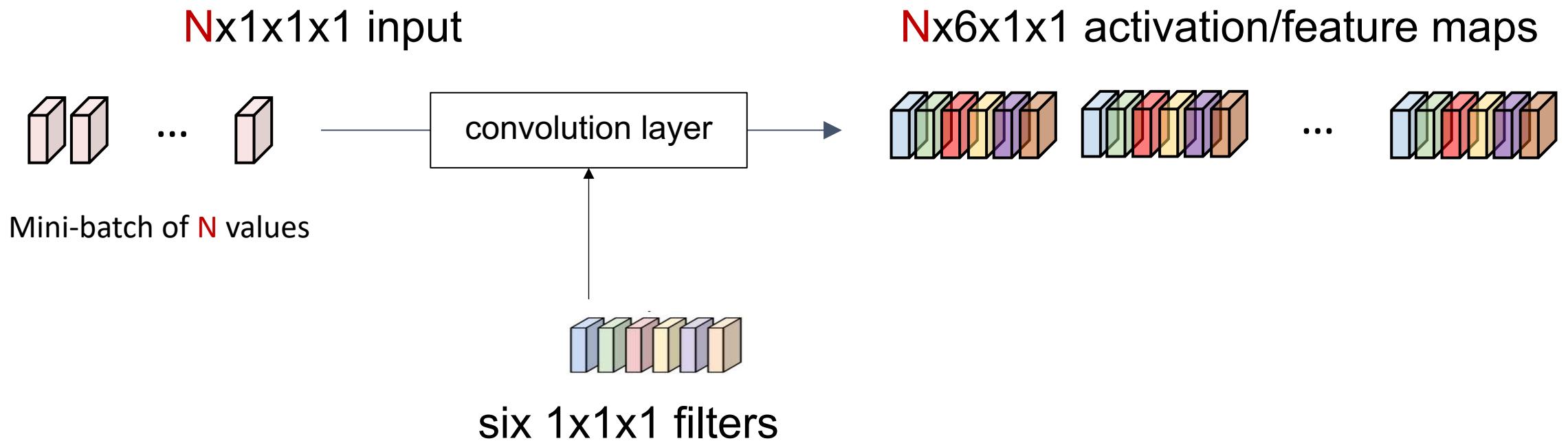
- What is mini-batch?
 - A **subset of all data** during one iteration to compute the gradient



Mini-batch



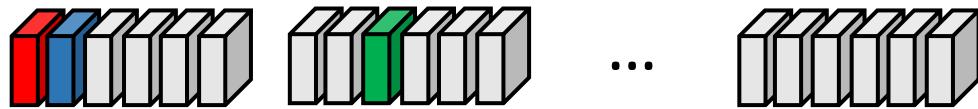
Mini-batch



Batch Normalization

Let's arrange the activations of the mini batch in a matrix of $N \times D$

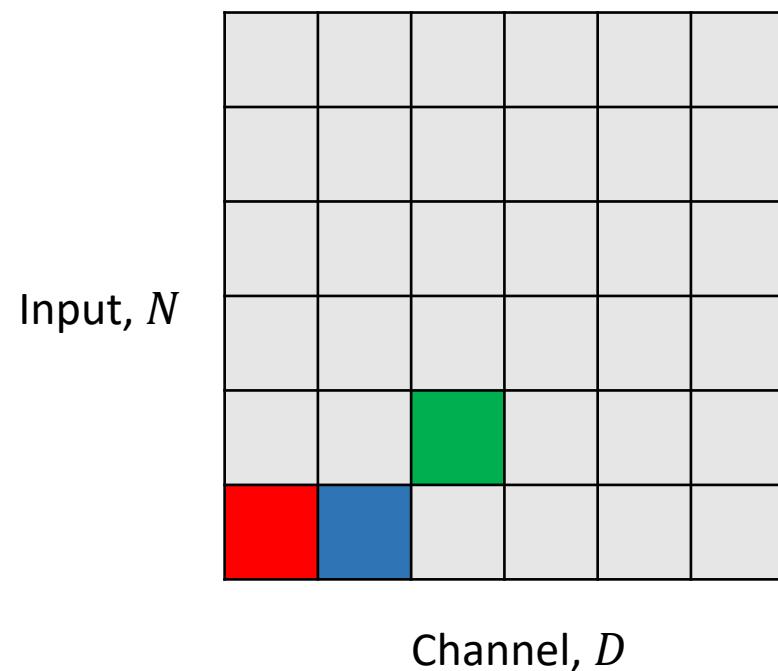
Nx6x1x1 activation/feature maps



Activation of the first input in the mini-batch

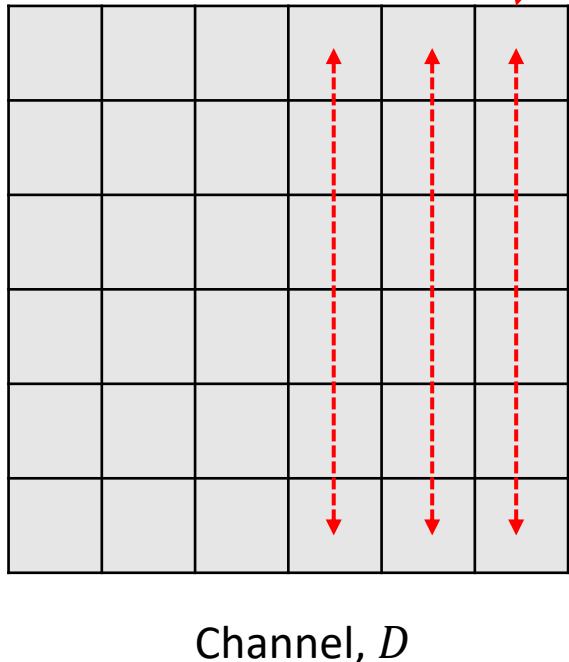
Activation of the second input in the mini-batch

Activation of the N-th input in the mini-batch



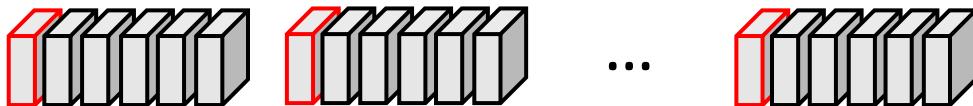
Batch Normalization

Input, N



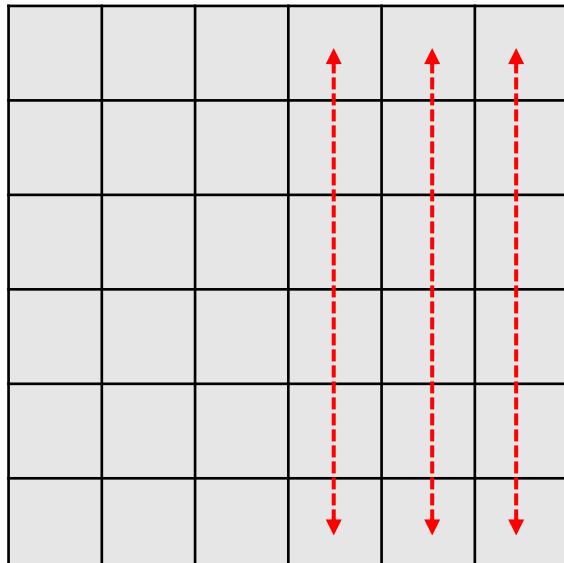
The goal of batch normalization is to normalize the values across each column so that the values of the column have **zero mean and unit variance**

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below



Batch Normalization - Training

Input, N



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization – Test Time

Input



Problem: Estimates depend on minibatch; can't do this at test-time

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization – Test Time

Input



Channel, D

Average of values seen
during training

Per-channel mean, shape is $1 \times D$

Average of values seen
during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add
flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization – Test Time

Input



Channel, D

During testing batchnorm
becomes a linear operator!
Can be fused with the previous
fully-connected or conv layer

Average of values seen
during training

Per-channel mean, shape is $1 \times D$

Average of values seen
during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add
flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization

Batch Normalization for
fully-connected networks

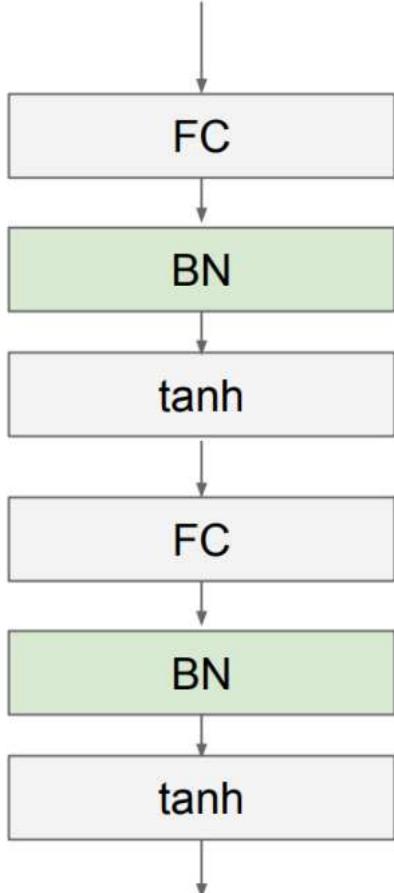
$$\begin{aligned} \mathbf{x} &: N \times D \\ \text{Normalize} &\quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} &: 1 \times D \\ \boldsymbol{\gamma}, \boldsymbol{\beta} &: 1 \times D \\ \mathbf{y} &= \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{aligned}$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

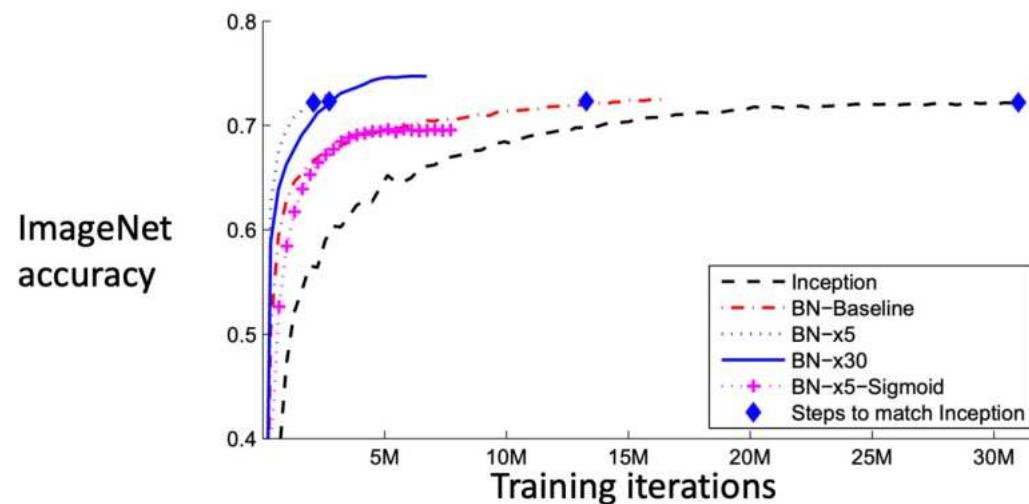
$$\begin{aligned} \mathbf{x} &: N \times C \times H \times W \\ \text{Normalize} &\quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} &: 1 \times C \times 1 \times 1 \\ \boldsymbol{\gamma}, \boldsymbol{\beta} &: 1 \times C \times 1 \times 1 \\ \mathbf{y} &= \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{aligned}$$

Normalize also on the spatial dimensions

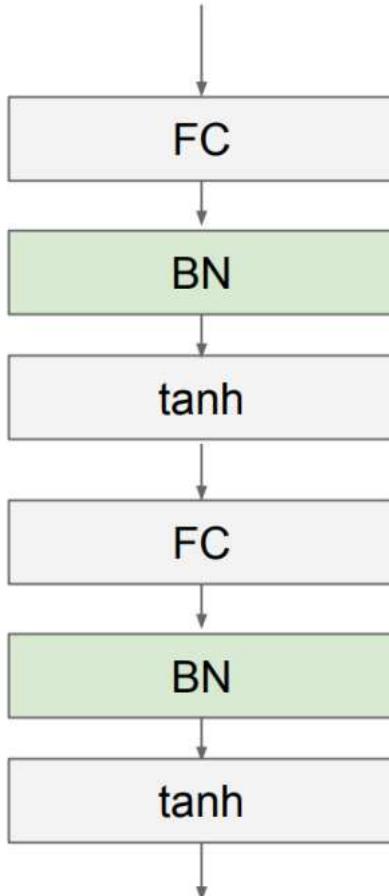
Batch Normalization



- It is usually done after a fully connected/convolutional layer and before a non-linearity layer
- Zero overhead at test-time: can be fused with conv
- Allows higher learning rates, faster convergence
- Networks becomes more robust to initialization



Batch Normalization



- It is usually done after a fully connected/convolutional layer and before a non-linearity layer
- Zero overhead at test-time: can be fused with conv
- Allows higher learning rates, faster convergence
- Networks becomes more robust to initialization
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a very common source of bugs!

Prevent Overfitting

I WAS WINNING
IMAGENET

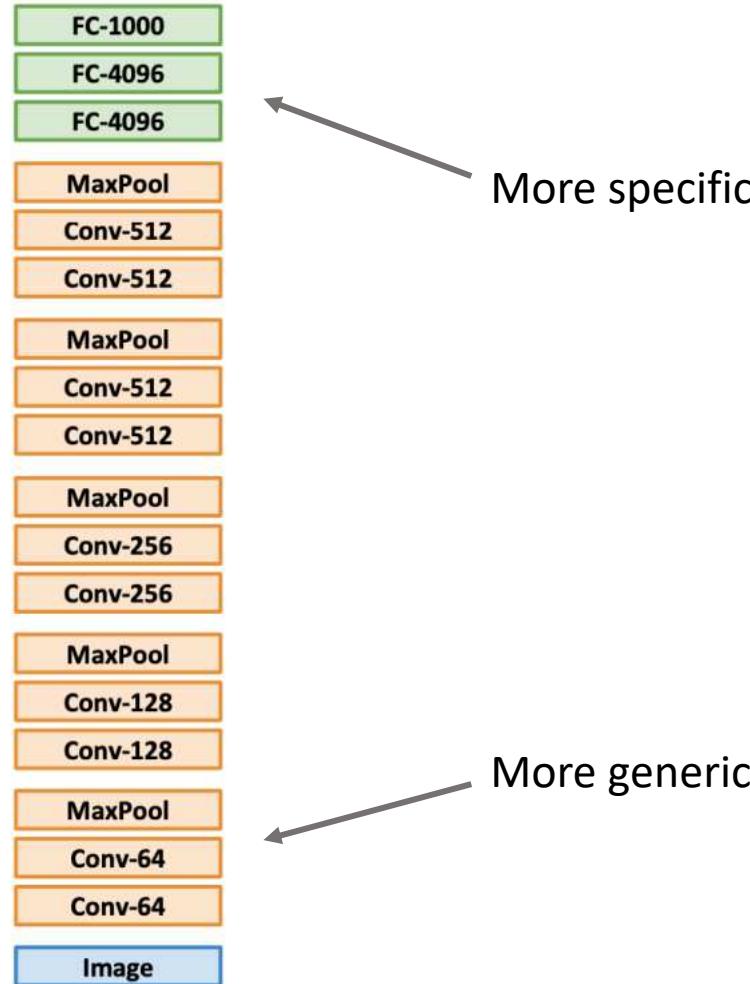


UNTIL A
DEEPER MODEL
CAME ALONG

Why overfitting?

- This happens when our model is too complex and too specialized on a small number of training data.
- Increase the size of the data, remove outliers in data, reduce the complexity of the model, reduce the feature dimension

Transfer learning

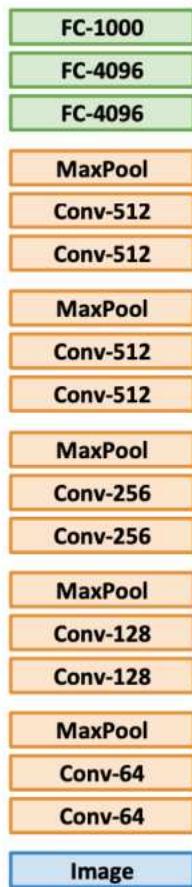


In a network with an N-dimensional softmax output layer that has been successfully trained toward a supervised classification objective, each output unit will be **specific** to a particular class

When trained on images, deep networks tend to learn first-layer features that resemble either Gabor filters or color blobs. These first-layer features are **general**.

Transfer learning

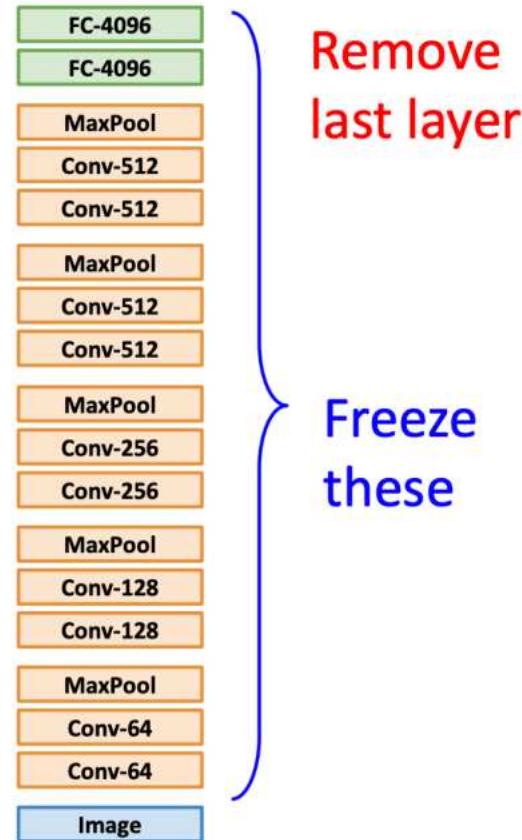
Step 1: Pre-training on large-scale dataset like ImageNet



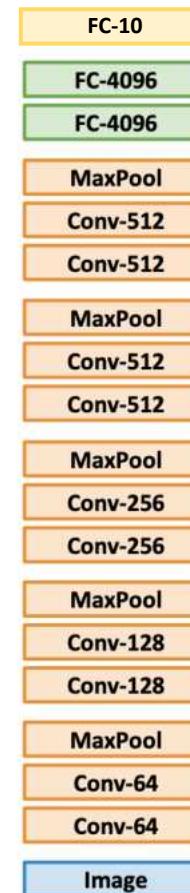
Transfer weights

Pre-training + Fine-tuning

Step 2: Use pre-trained network as initialization



Step 3: Fine-tuning

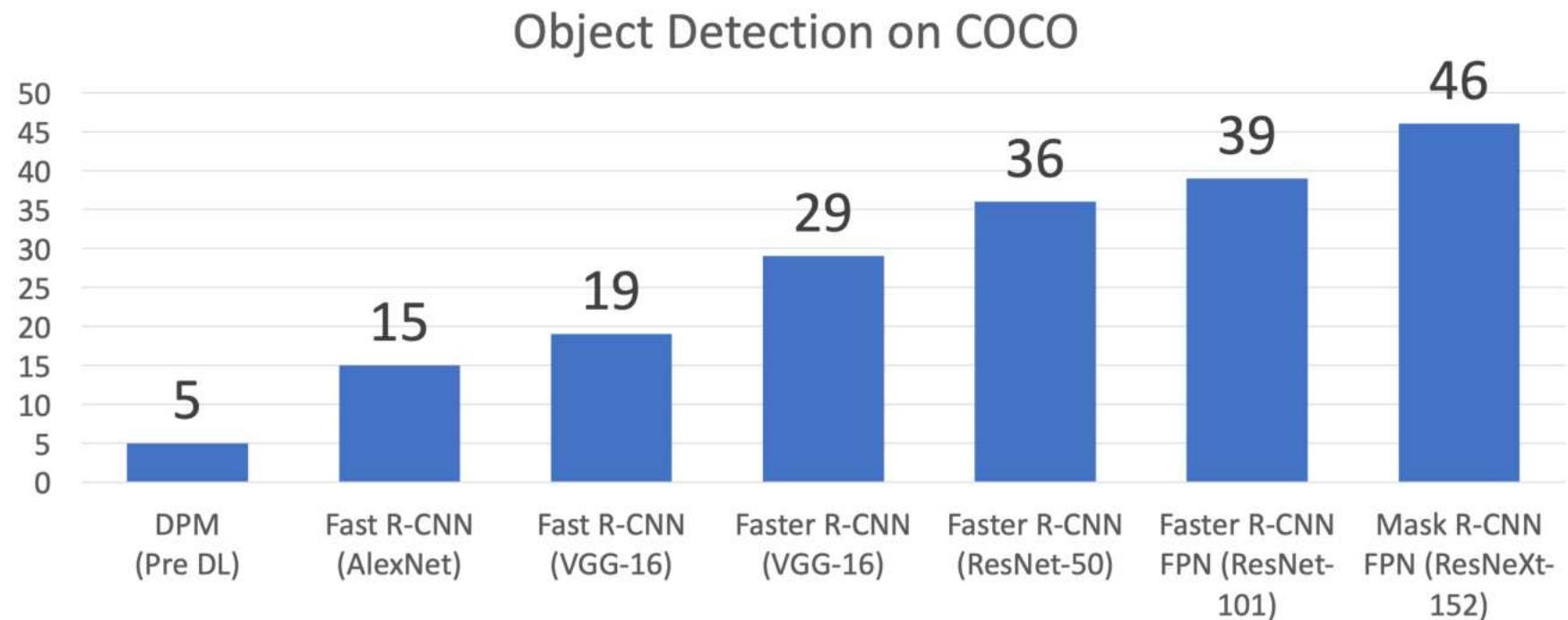


Add new layer
correspond to the
target class number

Train on new data

Transfer learning

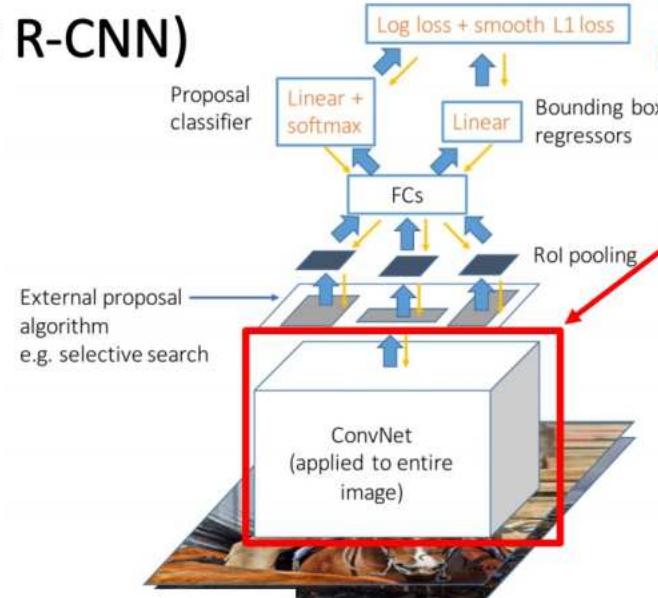
Architecture matters



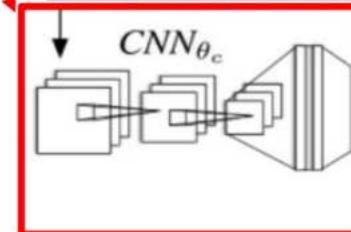
Transfer learning

Transfer learning is pervasive

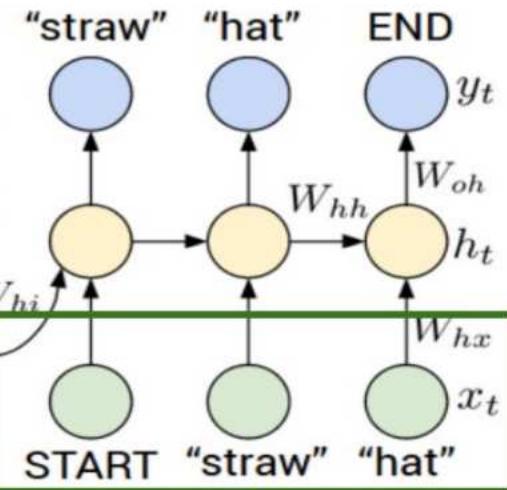
Object
Detection
(Fast R-CNN)



CNN pretrained
on ImageNet



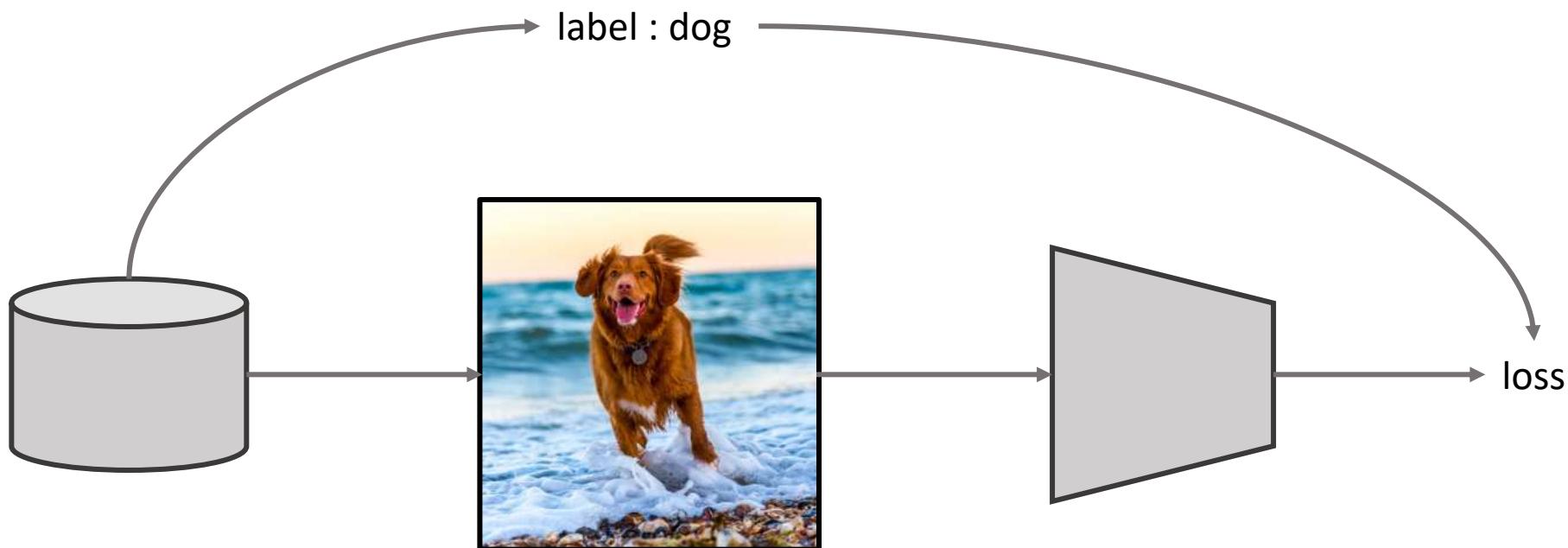
Word vectors pretrained
with word2vec



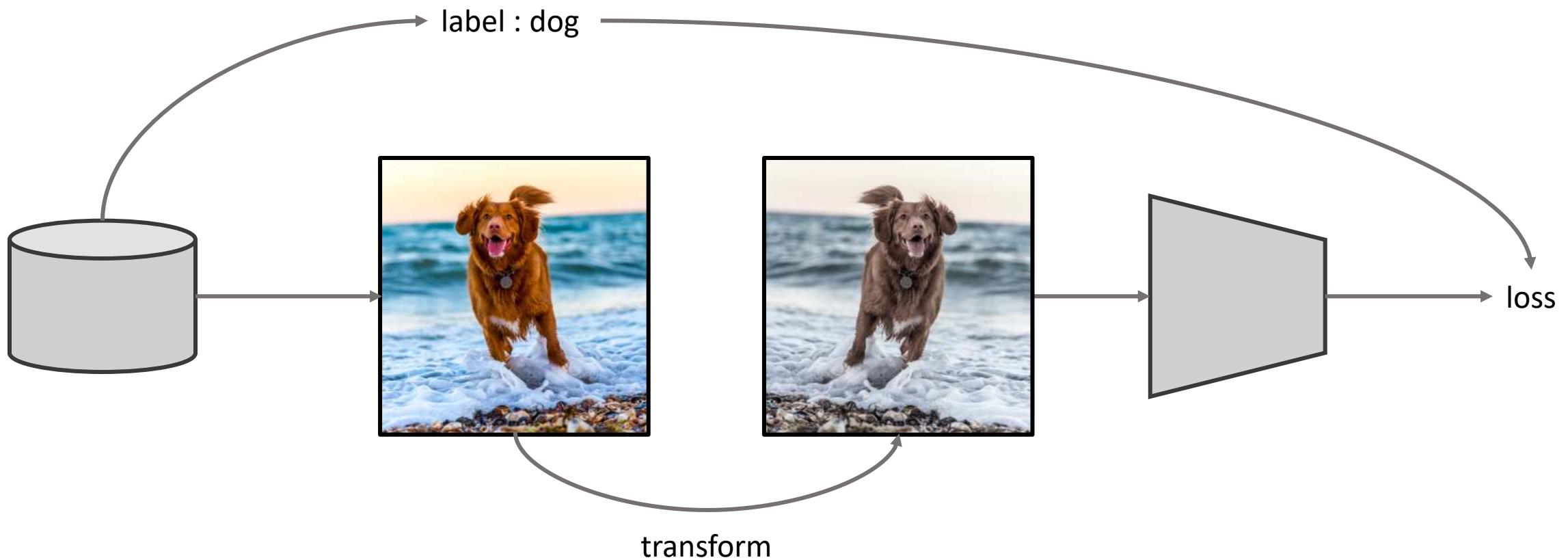
Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Data augmentation



Data augmentation



Data augmentation

Horizontal flip

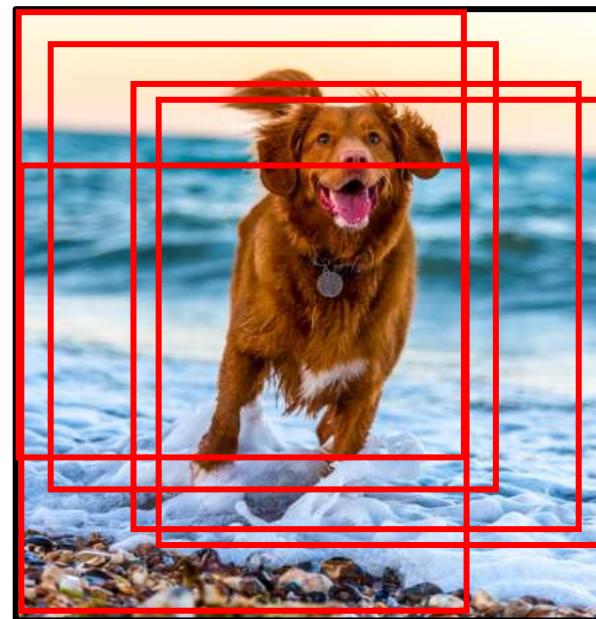


Data augmentation

Random crops and scales

Training:

1. Pick random L in range $[256, 480]$
2. Resize training image, short side = L
3. Sample random 224×224 patch



Data augmentation

Color jitter

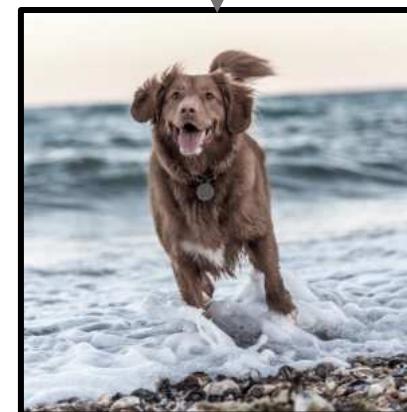
Simple :

1. Randomize contrast and brightness

Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(Used in AlexNet, ResNet, etc.)

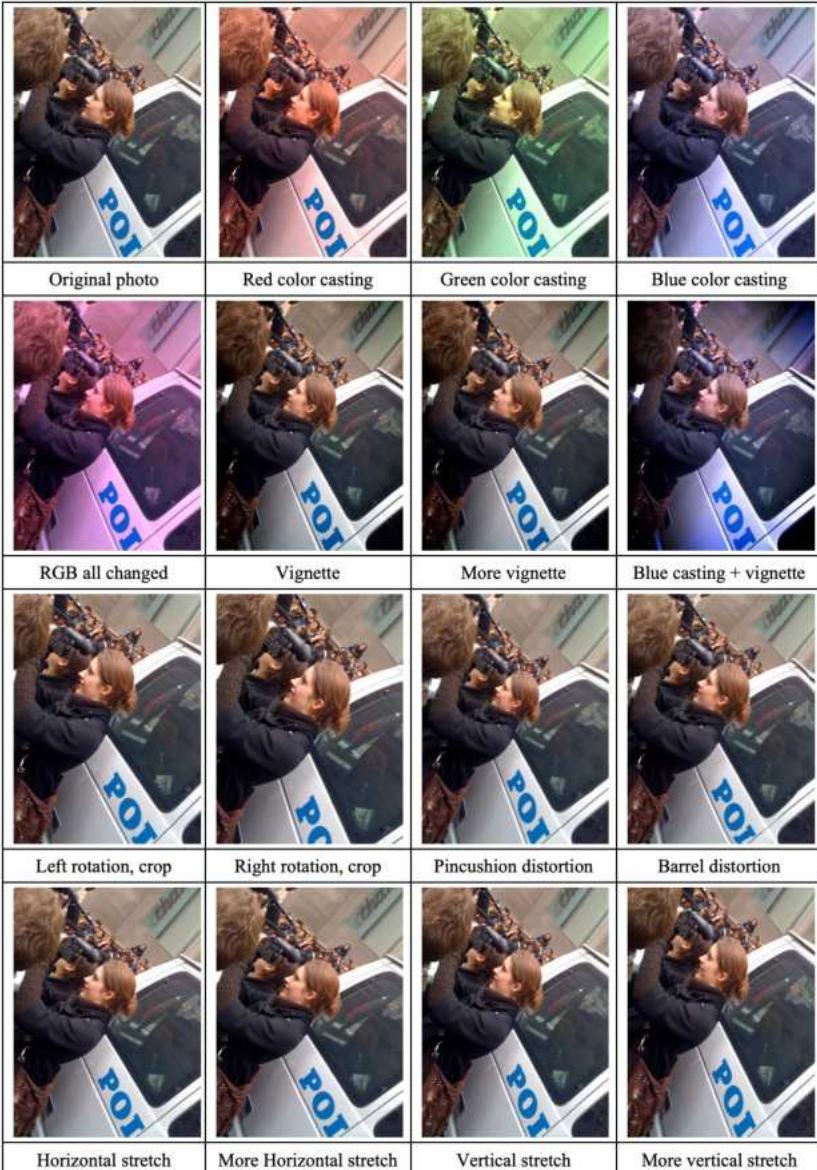


Data augmentation

There are many more data augmentation schemes

Random mix/combinations of:

translation - rotation - stretching - shearing, - lens distortions



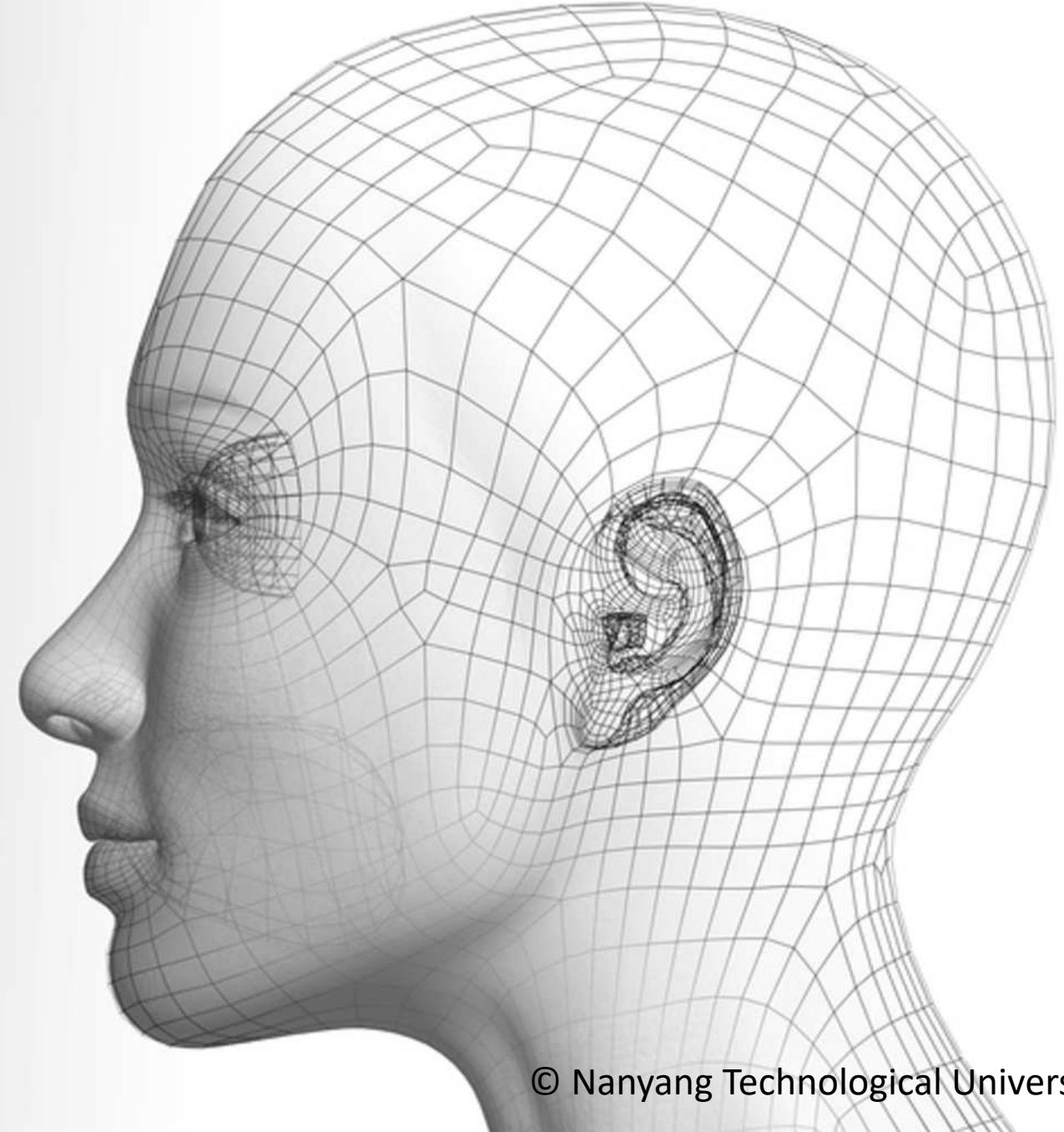
Recurrent Neural Networks (RNN)

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

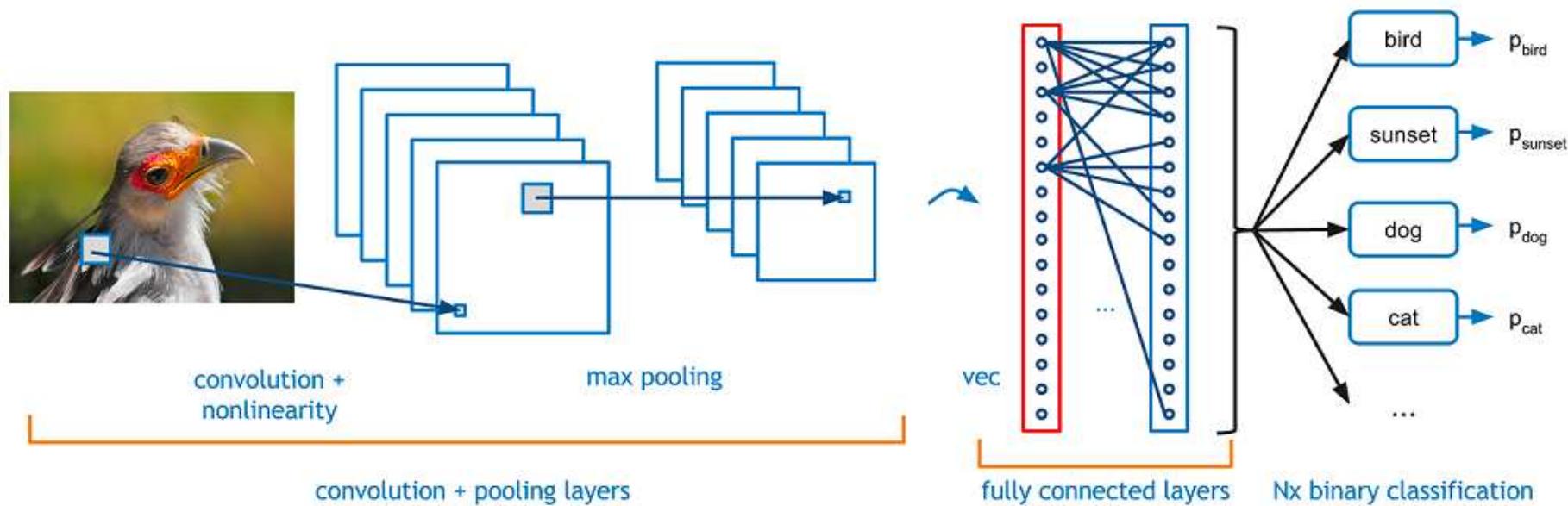
<https://twitter.com/XingangP>



Last week

- CNN Architectures
 - You learn some classic architectures
- More on convolution
 - How to calculate FLOPs
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution
 - You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network
- Batch normalization
 - You learn an important technique to improve the training of modern neural networks
- Prevent overfitting
 - Transfer learning
 - Data augmentation
 - You learn two important techniques to prevent overfitting in neural networks

Previous: Convolutional Neural Networks (CNN)



How about sequential information?

- Turn a sequence of sound pressures into a sequence of word identities?
- Speech recognition?
- Video prediction?

Outline

- Recurrent Neural Network (RNN)
 - Hidden recurrence
 - Top-down recurrence
- Long Short-Term Memory (LSTM)
 - Long-term dependency
 - Structure of LSTM
- Example Applications

Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNN)

Recurrent neural networks (RNN) are designed to process **sequential information**. That is, the data presented in a sequence.

The next data point in the sequence is usually **dependent** on the current data point.

Examples:

- Natural language processing (spoken words and written sentences). The next word in a sentence depends on the word which comes before it.
- Genomic sequences: a nucleotide in a DNA sequence is dependent on its neighbors.

RNN attempts to **capture dependency** among the data points in the sequence.

Text Generation with an RNN

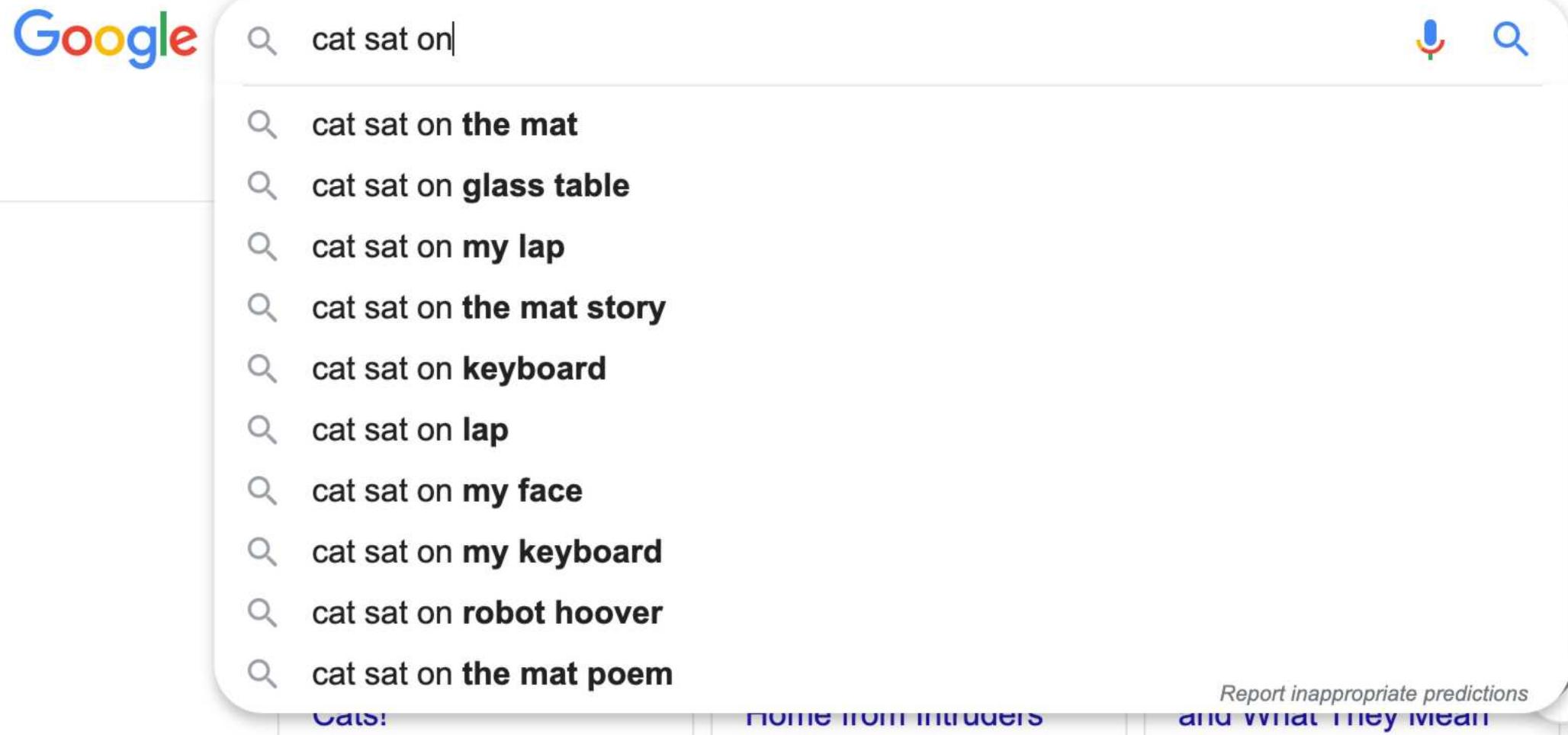
QUEENE:

*I had thought thou hadst a Roman; for the oracle,
Thus by All bids the man against the word,
Which are so weak of care, by old care done;
Your children were in your holy love,
And the precipitation through the bleeding throne.*

BISHOP OF ELY:

*Marry, and will, my lord, to weep in such a one were prettiest;
Yet now I was adopted heir
Of the world's lamentable day,
To watch the next way with his father with his face?*

Text Generation with an RNN



A screenshot of a Google search interface showing search suggestions for the query "cat sat on". The suggestions are listed below the search bar, each preceded by a magnifying glass icon.

- cat sat on the mat
- cat sat on glass table
- cat sat on my lap
- cat sat on the mat story
- cat sat on keyboard
- cat sat on lap
- cat sat on my face
- cat sat on my keyboard
- cat sat on robot hoover
- cat sat on the mat poem

The suggestions include several common nouns like "mat", "table", "lap", "face", and "keyboard", as well as some less common ones like "robot hoover" and "poem". The "the mat" suggestion appears twice, once with "the" and once without it.

Image Captioning



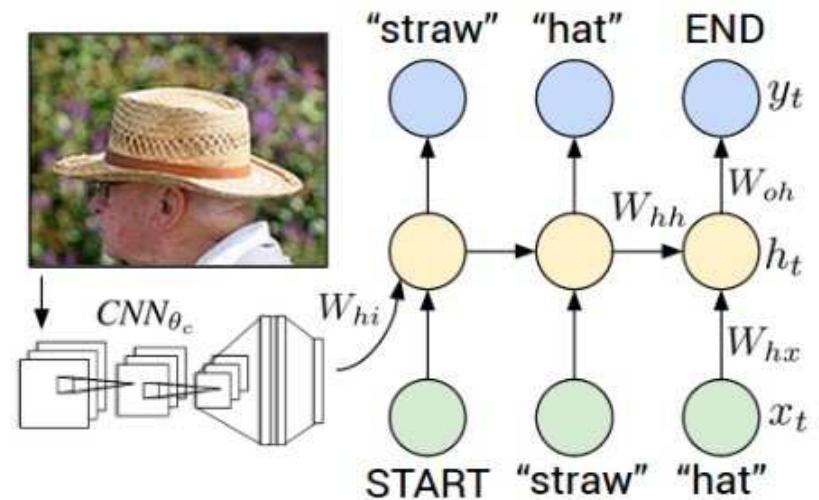
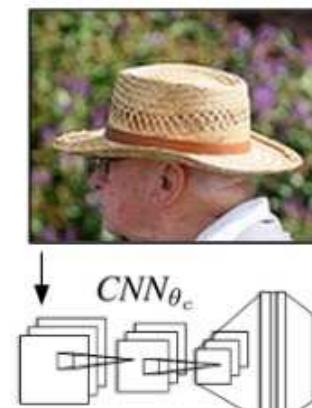
"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."

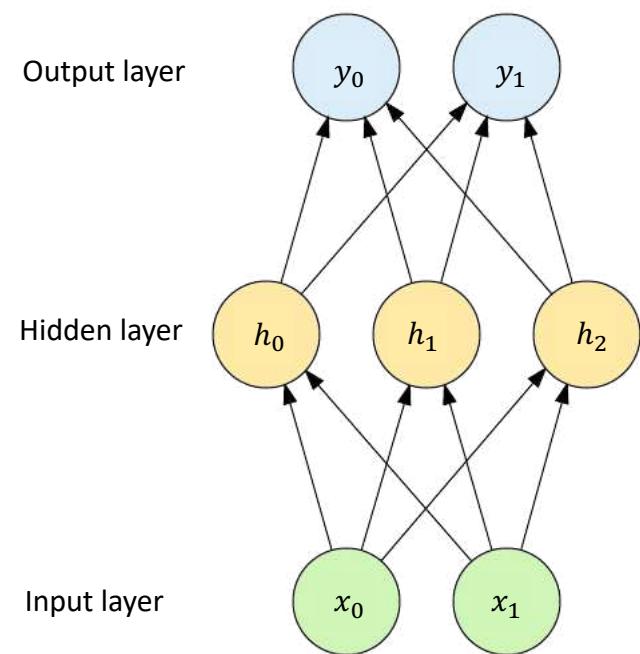


"two young girls are playing with lego toy."

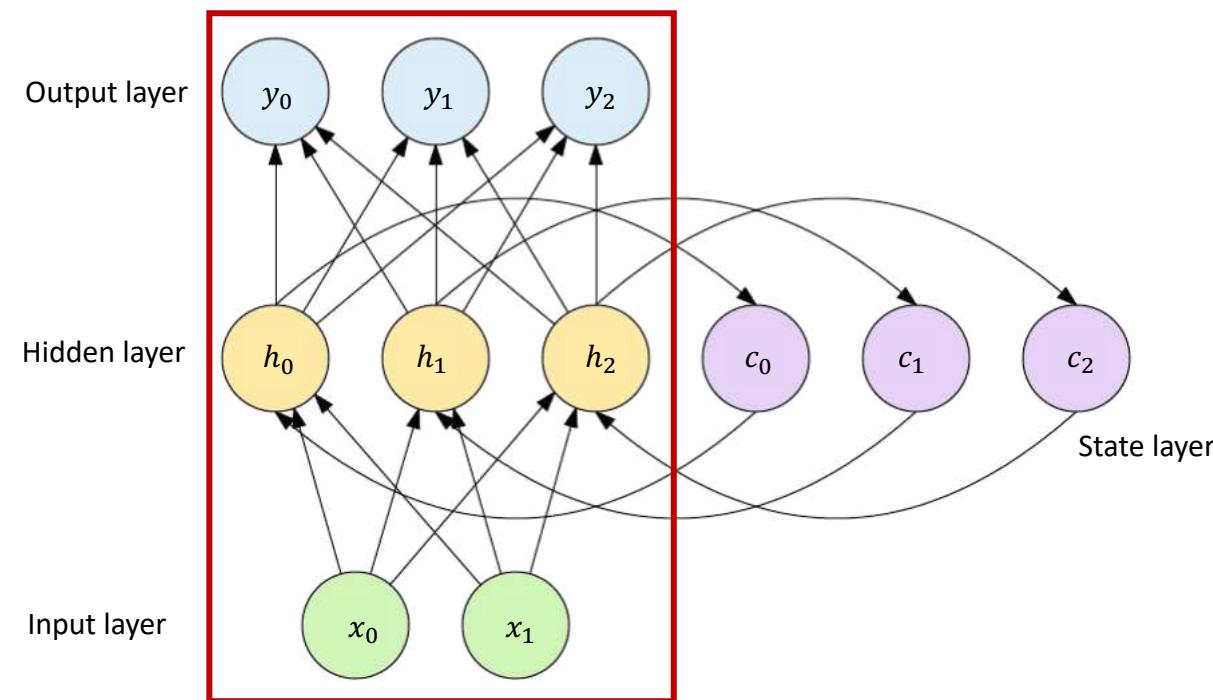


Recurrent Neural Networks (RNN)

Feedforward NN



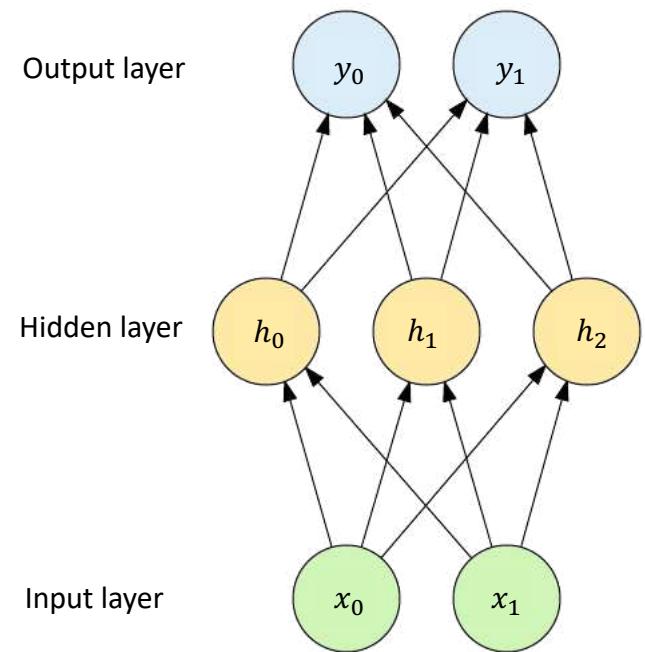
RNN with hidden recurrence



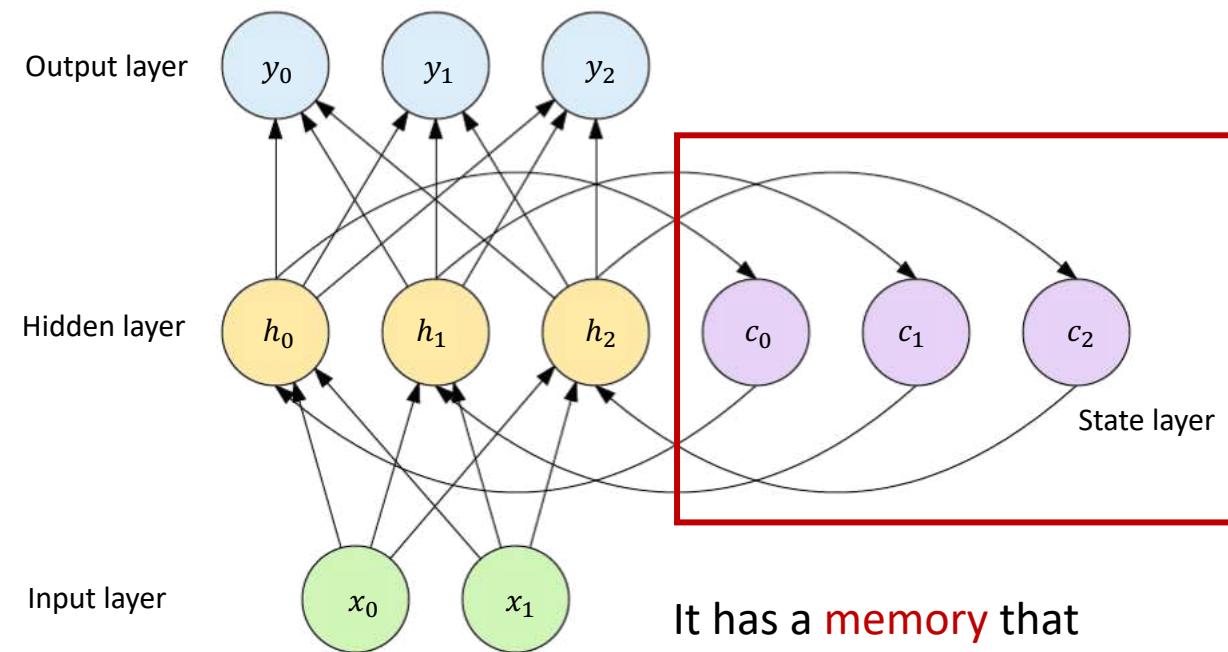
RNNs are called recurrent because they perform **the same task for every data element** (frame) of a sequence, with the output **depending on the previous computations**.

Recurrent Neural Networks (RNN)

Feedforward NN



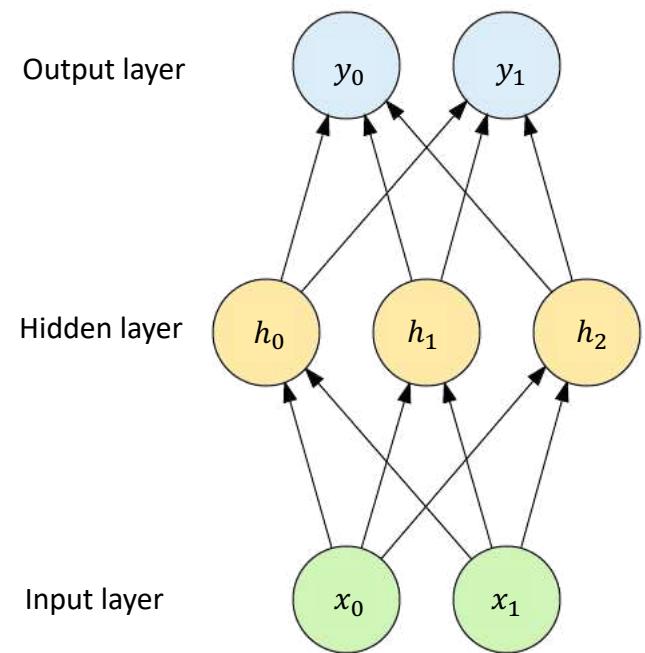
RNN with hidden recurrence



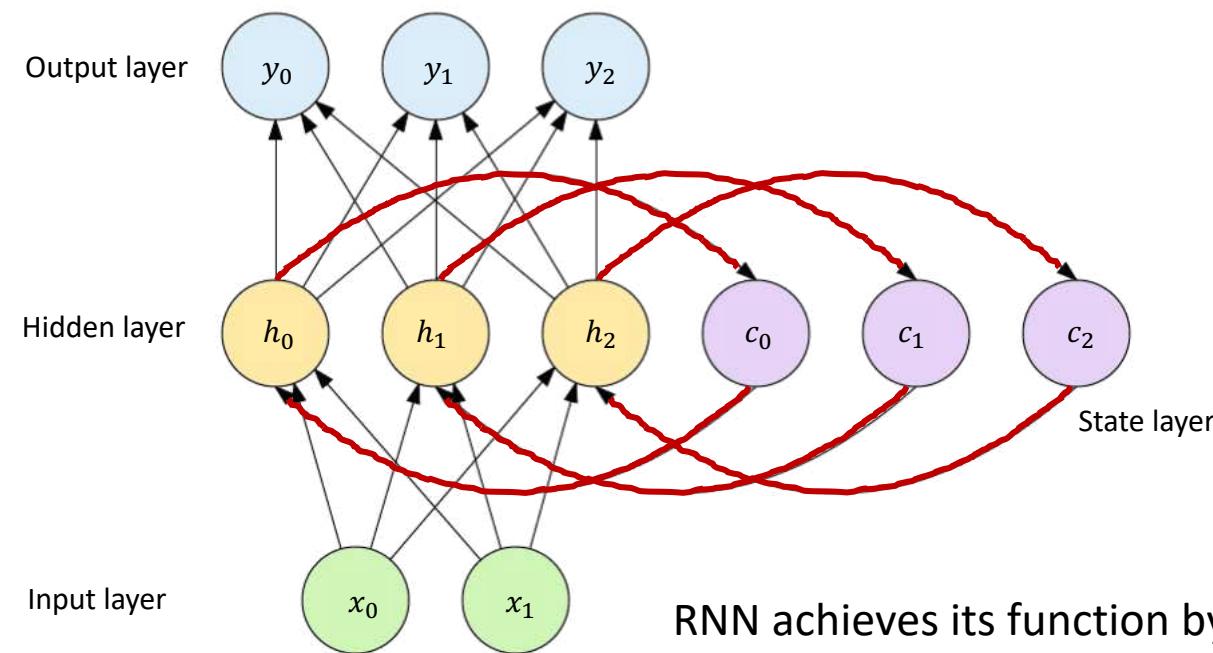
It has a **memory** that captures information about what has been processed so far.

Recurrent Neural Networks (RNN)

Feedforward NN



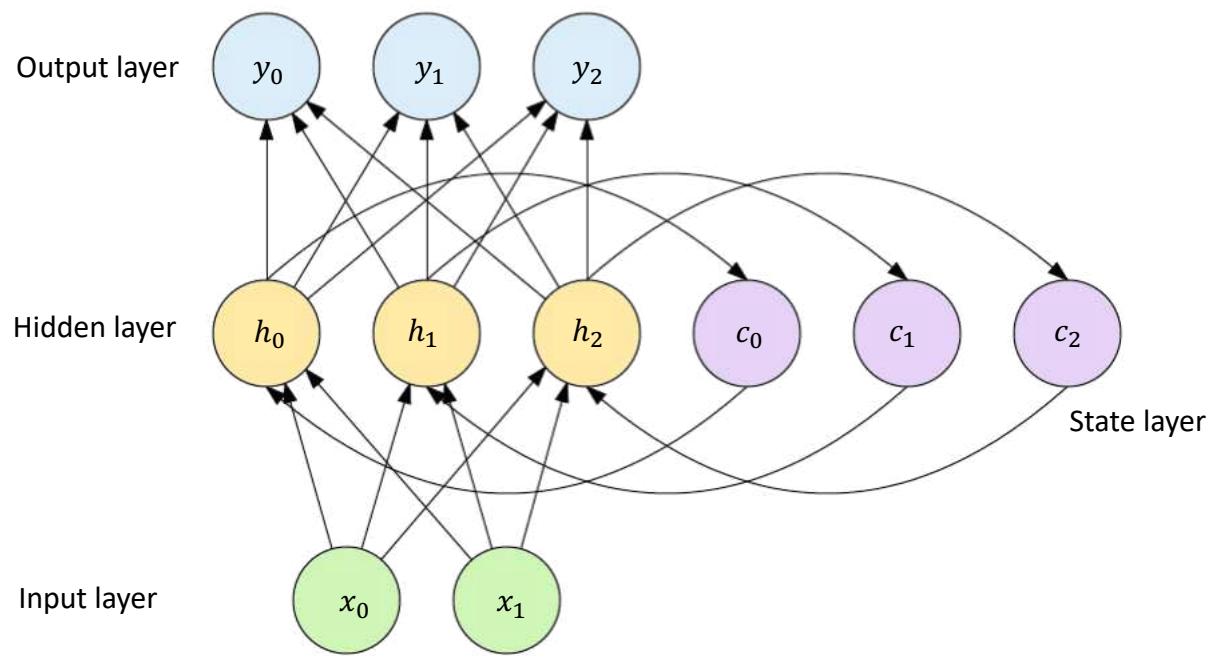
RNN with hidden recurrence



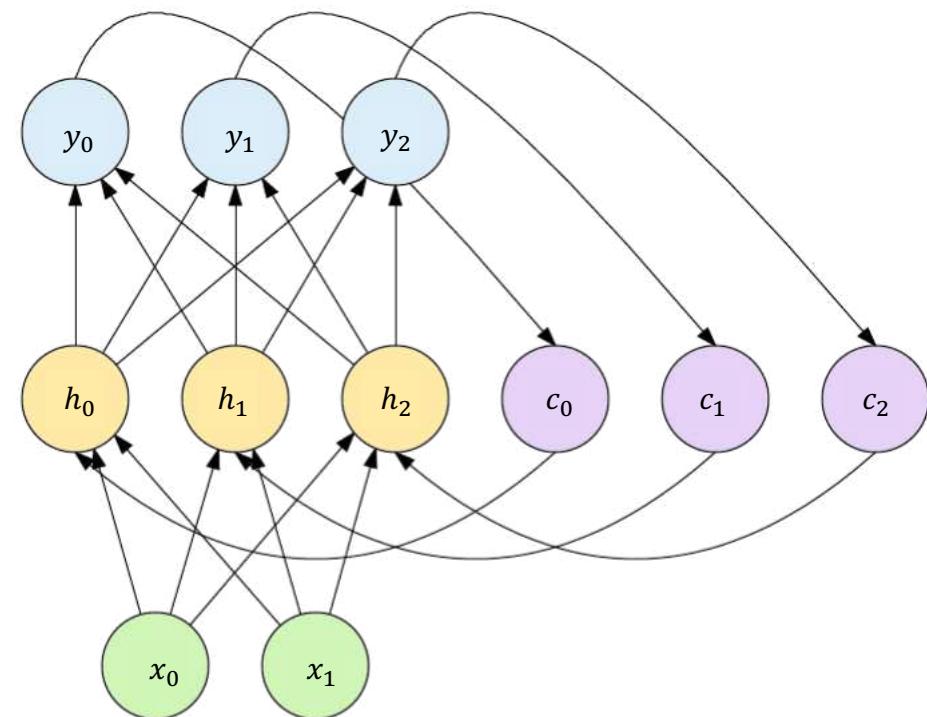
RNN achieves its function by using **feedback connections** that enables learning of sequential (temporal) information of sequences.

Types of RNN

RNN with hidden recurrence
(Elman-type)

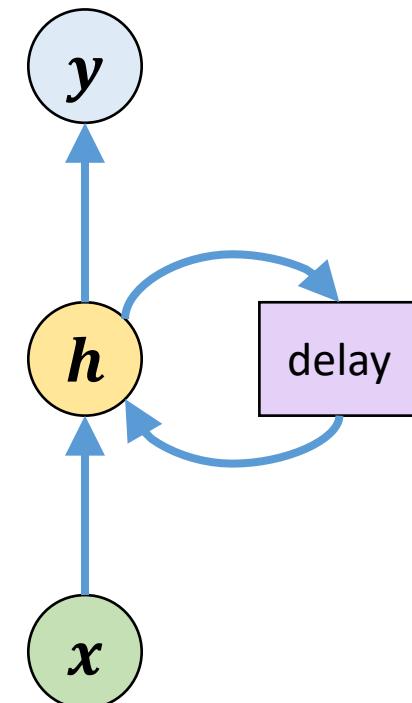
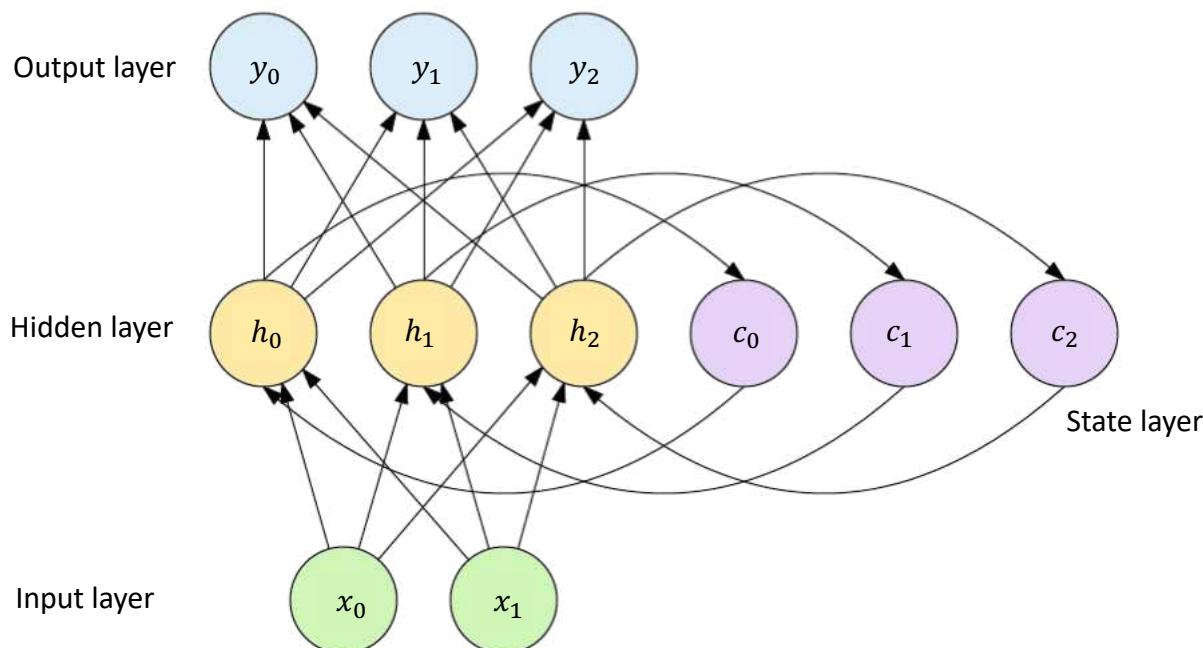


RNN with top-down recurrence
(Jordan-type)



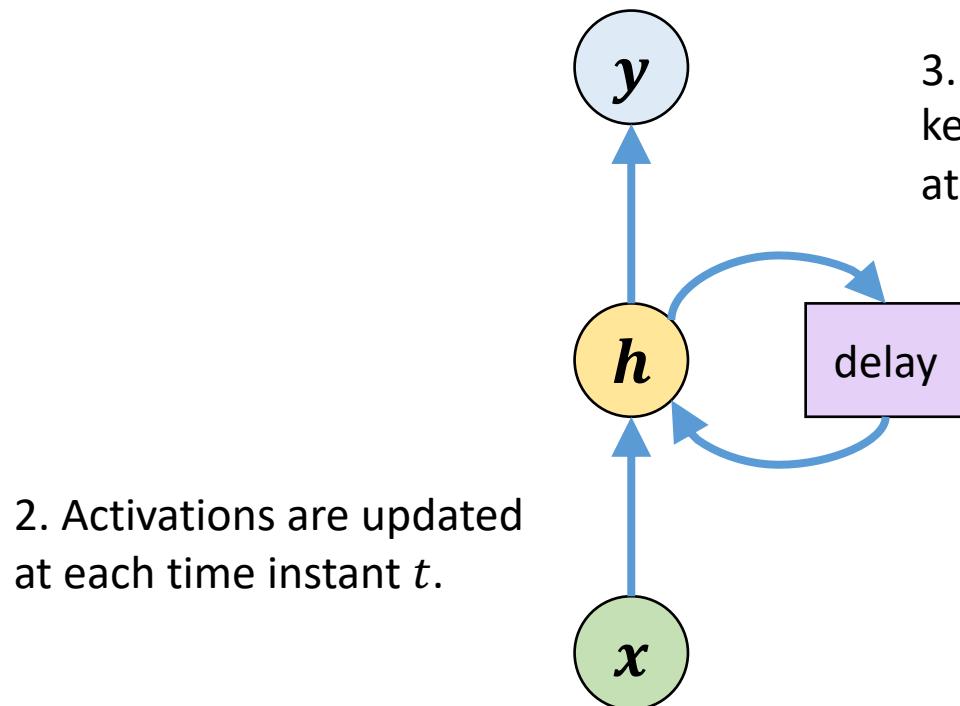
RNN with hidden recurrence (Elman type)

RNN with hidden recurrence
(Elman-type)



Elman type networks (sometimes called a “**Vanilla RNN**”) that produce an output at each time step and have recurrent connections between hidden units.

RNN with hidden recurrence (Elman type)



1. Data is presented as a sequence of instance t (time) that is discretized

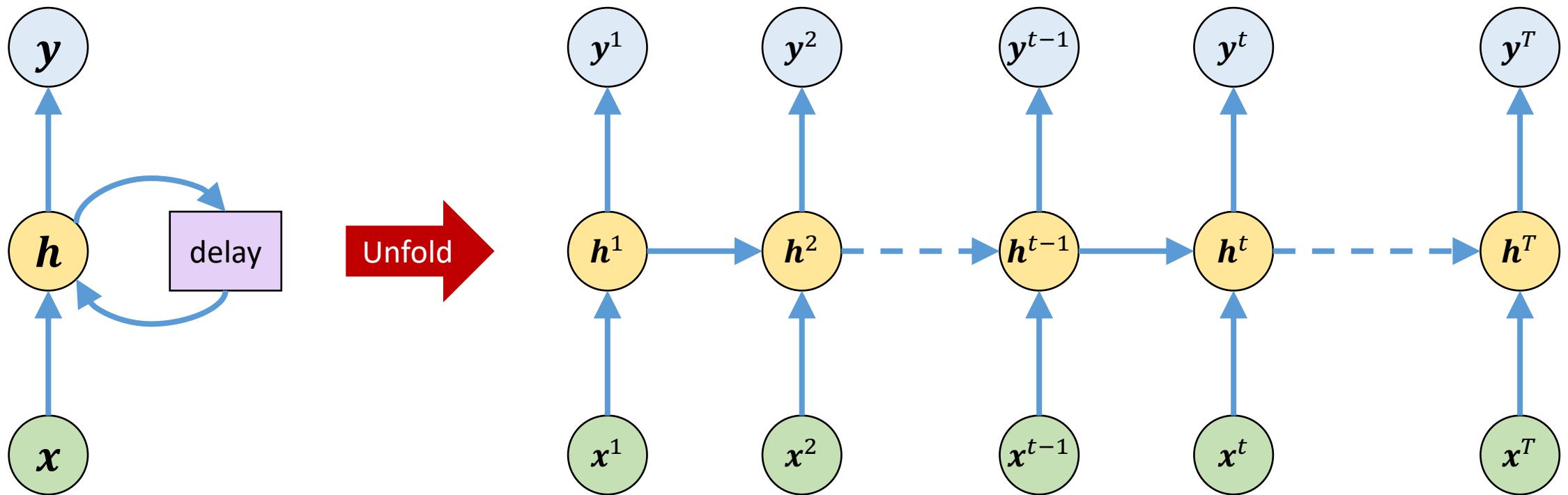
2. Activations are updated at each time instant t .

3. The hidden-layer activation at time $t - 1$ is kept by the **delay unit** and fed to the hidden layer at time t together with the raw input $x(t)$.

4. The **delay unit** represents that the activation is held for one time unit until the next time instance.

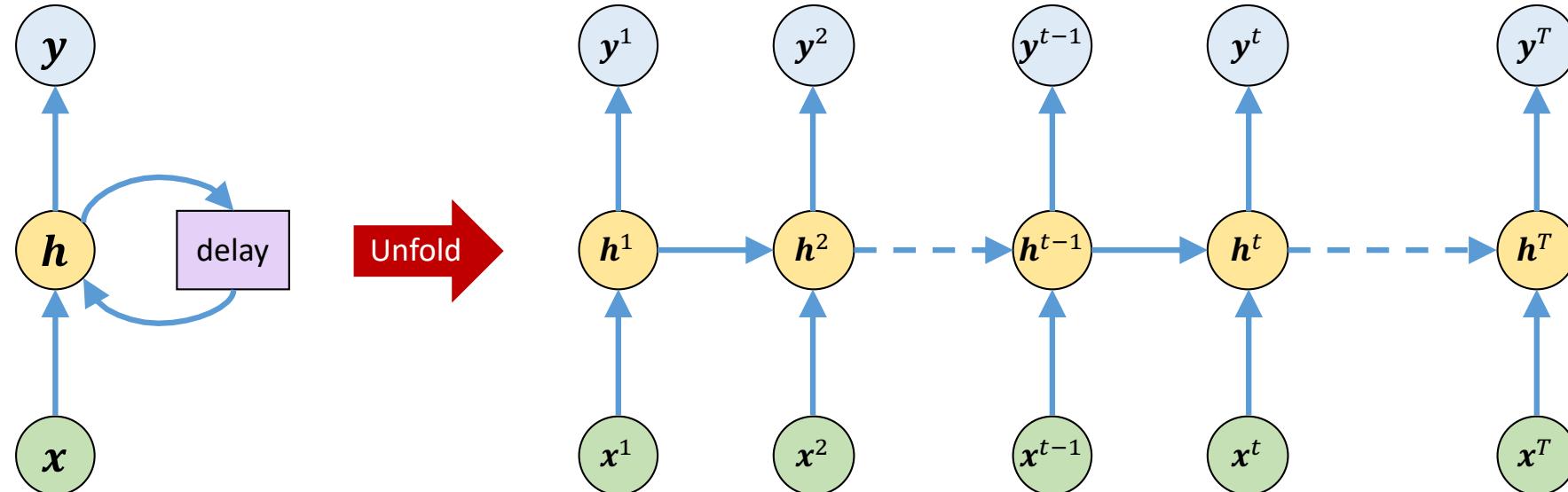
Here, one unit represents the time between two adjacent data points in the sequence .

RNN with hidden recurrence



The recurrent connections in the hidden-layer can be unfolded to process sequences of arbitrary length.

RNN with hidden recurrence



By considering the unfolded structure, $\mathbf{h}(t)$ is dependent on all the inputs at time t and before time t :

$$\mathbf{h}(t) = f^t(x(t), x(t-1), \dots, x(2), x(1))$$

The function f^t takes the whole past sequence $(x(t), x(t-1), \dots, x(2), x(1))$ as input and produce the hidden layer activation.

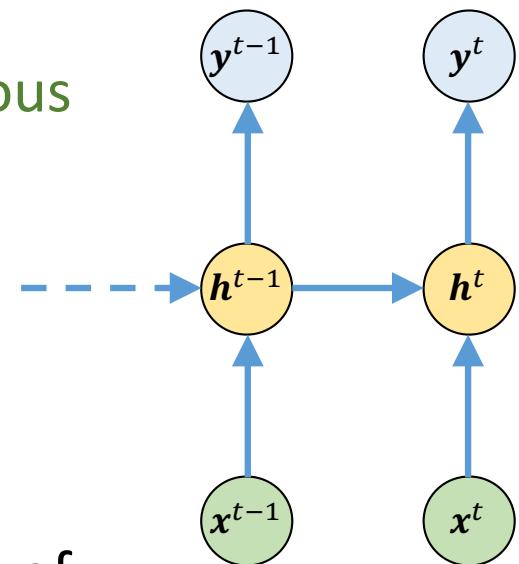
RNN with hidden recurrence

Not an efficient way! The function f^t is dependent to the sequence length.

Let's represent the past inputs by the hidden-layer activations in the previous instant.

$$h(t) = f^t(x(t), x(t-1), \dots, x(2), x(1))$$


$$h(t-1)$$



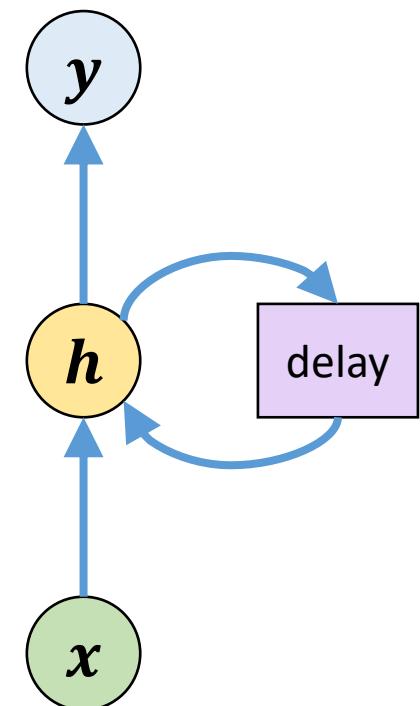
The recurrent structure allows us to factorize f^t into repeated applications of a function f . We can write:

new state $\mathbf{h}(t) = f(\mathbf{h}(t-1), \mathbf{x}(t))$
some function old state input vector at
some time step

RNN with hidden recurrence

The folded structure introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same size, rather than specified in terms of a variable-length history of states.
 2. It is possible to use same transition function f with the same parameters at every time step.



RNN with hidden recurrence

U : weight vector that transforms raw inputs to the hidden-layer

W : recurrent weight vector connecting previous hidden-layer output to hidden input

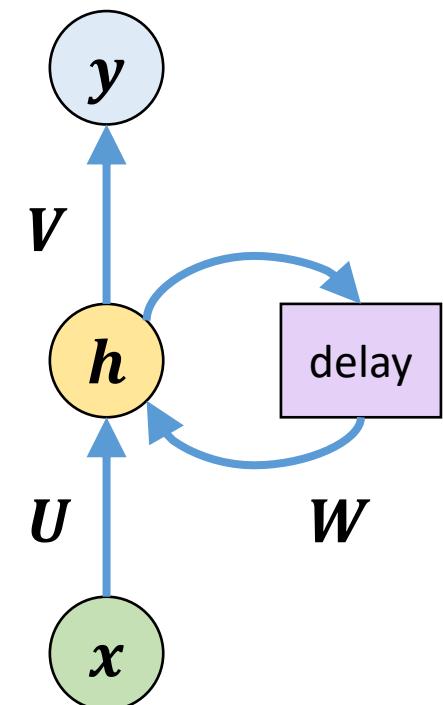
V : weight vector of the output layer

b : bias connected to hidden layer

c : bias connected to the output layer

ϕ : the tanh hidden-layer activation function

σ : the linear/softmax output-layer activation function



RNN with hidden recurrence

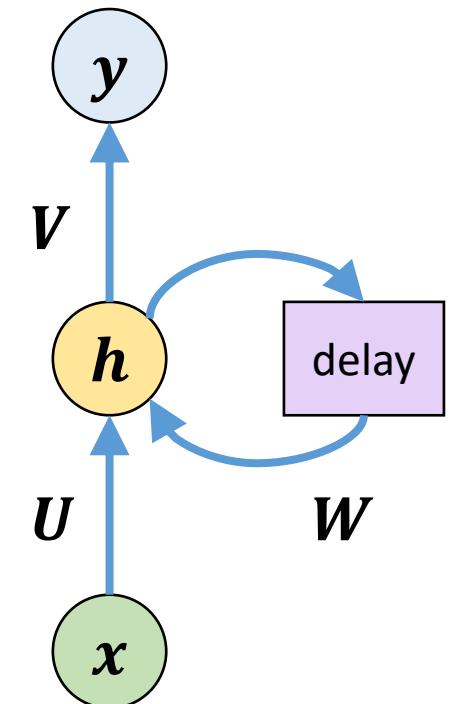
Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Elman-type RNN with one hidden-layer is given by:

$$h(t) = \phi(U^T x(t) + W^T h(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

σ is a *softmax* function for classification and a *linear* function for regression.



RNN with hidden recurrence: batch processing

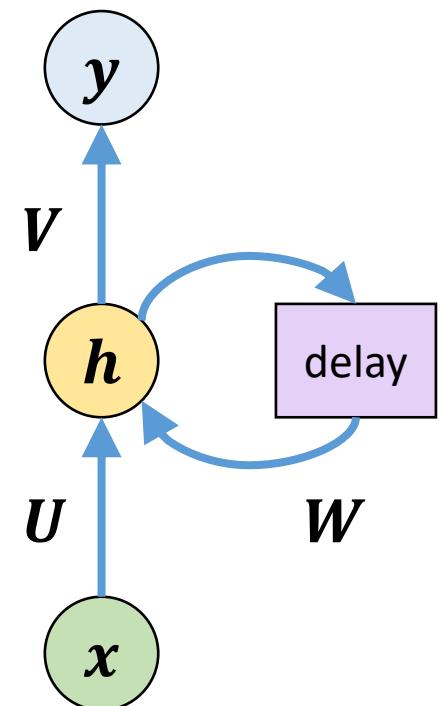
Given P patterns $\{\mathbf{x}_p\}_{p=1}^P$ where $\mathbf{x}_p = (\mathbf{x}_p(t))_{t=1}^T$,

$$\mathbf{X}(t) = \begin{pmatrix} \mathbf{x}_1(t)^T \\ \mathbf{x}_2(t)^T \\ \vdots \\ \mathbf{x}_P(t)^T \end{pmatrix}$$

Let $\mathbf{X}(t)$, $\mathbf{Y}(t)$, and $\mathbf{H}(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Elman-type RNN is given by:

$$\begin{aligned}\mathbf{H}(t) &= \phi(\mathbf{X}(t)\mathbf{U} + \mathbf{H}(t-1)\mathbf{W} + \mathbf{B}) \\ \mathbf{Y}(t) &= \sigma(\mathbf{H}(t)\mathbf{V} + \mathbf{C})\end{aligned}$$



Example 1

A recurrent neural network with hidden recurrence has two input neurons, three hidden neurons, and two output neurons. The parameters of the

network are initialized as $\mathbf{U} = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix}$, $\mathbf{W} = \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix}$ and $\mathbf{V} = \begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}$.

Bias to the hidden layer $\mathbf{b} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$ and to the output layer $\mathbf{c} = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$.

For a sequence of inputs $(\mathbf{x}(1), \mathbf{x}(2), \mathbf{x}(3), \mathbf{x}(4))$ where

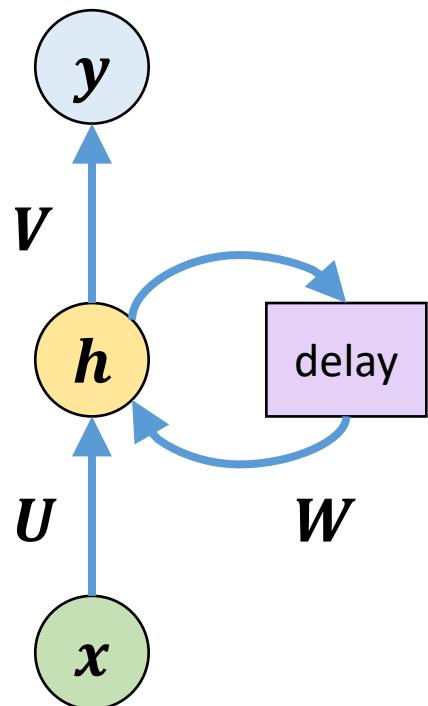
$$\mathbf{x}(1) = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \mathbf{x}(2) = \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \mathbf{x}(3) = \begin{pmatrix} 0 \\ 3 \end{pmatrix}, \text{ and } \mathbf{x}(4) = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

find the output of RNN.

Assume that hidden layer activations are initialized to zero and *tanh* and sigmoid functions for the hidden and output layer activation functions, respectively.

Example 1 (con't)

$$\mathbf{U} = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix}, \quad \mathbf{W} = \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix} \text{ and } \mathbf{V} = \begin{pmatrix} 2.0 & -1.0 \\ -1.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}$$



$$\mathbf{b} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix} \text{ and } \mathbf{c} = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$$

Three hidden neurons and two output neurons

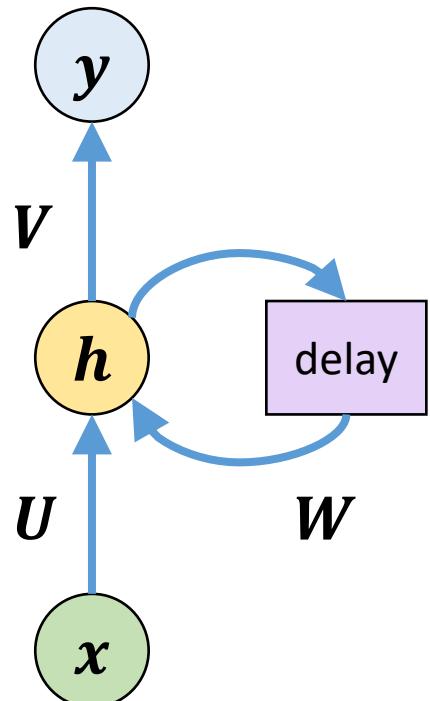
$$\begin{aligned}\mathbf{h}(t) &= \phi(\mathbf{U}^T \mathbf{x}(t) + \mathbf{W}^T \mathbf{h}(t-1) + \mathbf{b}) \\ \mathbf{y}(t) &= \sigma(\mathbf{V}^T \mathbf{h}(t) + \mathbf{c})\end{aligned}$$

$$\begin{aligned}\phi(u) &= \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}} \\ \sigma(u) &= \text{sigmoid}(u) = \frac{1}{1 + e^{-u}}.\end{aligned}$$

Assume $\mathbf{h}(0) = (0 \quad 0 \quad 0)^T$.

Example 1 (con't)

At $t=1$, $\mathbf{x}(1) = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$:

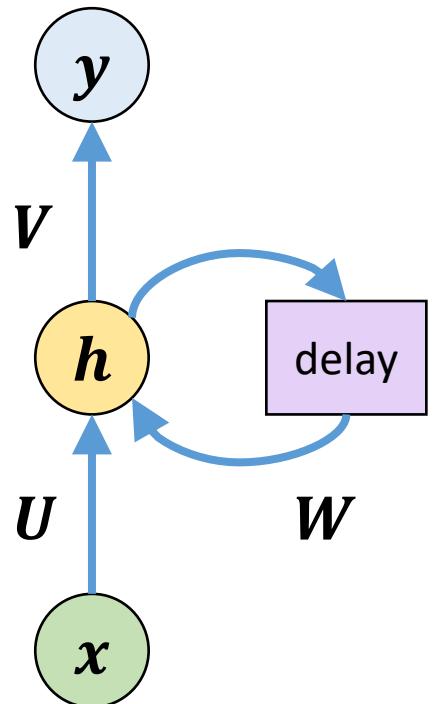


$$\begin{aligned}\mathbf{h}(1) &= \phi(\mathbf{U}^T \mathbf{x}(1) + \mathbf{W}^T \mathbf{h}(0) + \mathbf{b}) \\ &= \tanh \left(\begin{pmatrix} -1.0 & 0.5 \\ 0.5 & 0.1 \\ 0.2 & -2.0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 2.0 & 1.5 & -0.2 \\ 1.3 & 0.0 & 1.5 \\ -1.0 & -0.5 & 0.4 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.2 \\ 0.72 \\ -1.0 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{y}(1) &= \sigma(\mathbf{V}^T \mathbf{h}(1) + \mathbf{c}) \\ &= \text{sigmoid} \left(\begin{pmatrix} 2.0 & -1.5 & 0.2 \\ -1.0 & 0.5 & 0.8 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.72 \\ -1.0 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.41 \\ 0.37 \end{pmatrix}\end{aligned}$$

Example 1 (con't)

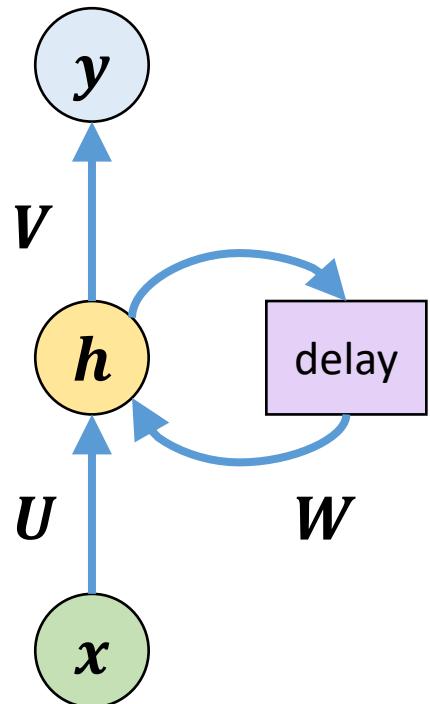
At $t=2$, $\mathbf{x}(2) = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$:



$$\begin{aligned}\mathbf{h}(2) &= \phi(\mathbf{U}^T \mathbf{x}(2) + \mathbf{W}^T \mathbf{h}(1) + \mathbf{b}) \\ &= \tanh \left(\begin{pmatrix} -1.0 & 0.5 \\ 0.5 & 0.1 \\ 0.2 & -2.0 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 2.0 & 1.5 & -0.2 \\ 1.3 & 0.0 & 1.5 \\ -1.0 & -0.5 & 0.4 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.72 \\ -1.0 \end{pmatrix} + \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix} \right) \\ &= \begin{pmatrix} 1.0 \\ -0.89 \\ -0.99 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{y}(2) &= \sigma(\mathbf{V}^T \mathbf{h}(2) + \mathbf{c}) \\ &= \text{sigmoid} \left(\begin{pmatrix} 2.0 & -1.5 & 0.2 \\ -1.0 & 0.5 & 0.8 \end{pmatrix} \begin{pmatrix} 1.0 \\ -0.89 \\ -0.99 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix} \right) = \begin{pmatrix} 0.97 \\ 0.11 \end{pmatrix}\end{aligned}$$

Example 1 (con't)



Similarly,

$$\text{at } t=3, \mathbf{x}(3) = \begin{pmatrix} 0 \\ 3 \end{pmatrix}; \mathbf{h}(3) \begin{pmatrix} 0.99 \\ 0.3 \\ -1.0 \end{pmatrix} \text{ and } \mathbf{y}(3) = \begin{pmatrix} 0.86 \\ 0.18 \end{pmatrix}$$

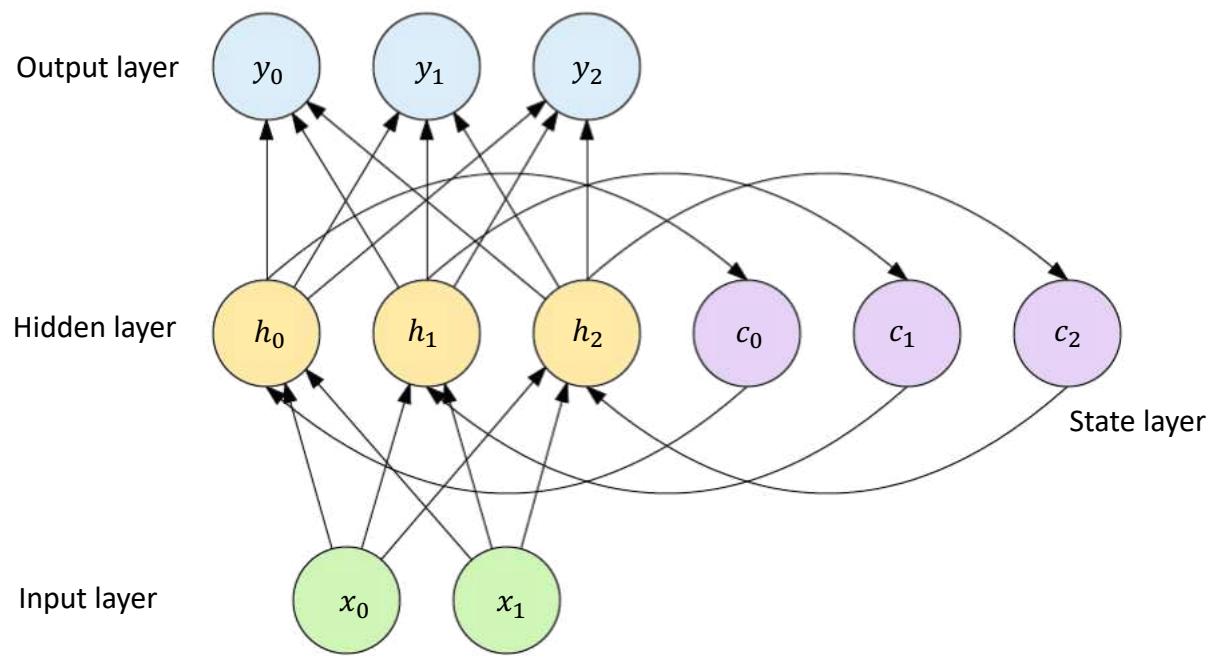
$$\text{at } t=4, \mathbf{x}(4) = \begin{pmatrix} 2 \\ -1 \end{pmatrix}; \mathbf{h}(4) \begin{pmatrix} 0.31 \\ 0.71 \\ 0.79 \end{pmatrix} \text{ and } \mathbf{y}(4) = \begin{pmatrix} 0.55 \\ 0.68 \end{pmatrix}$$

The output is $(\mathbf{y}(1), \mathbf{y}(2), \mathbf{y}(3), \mathbf{y}(4))$ where

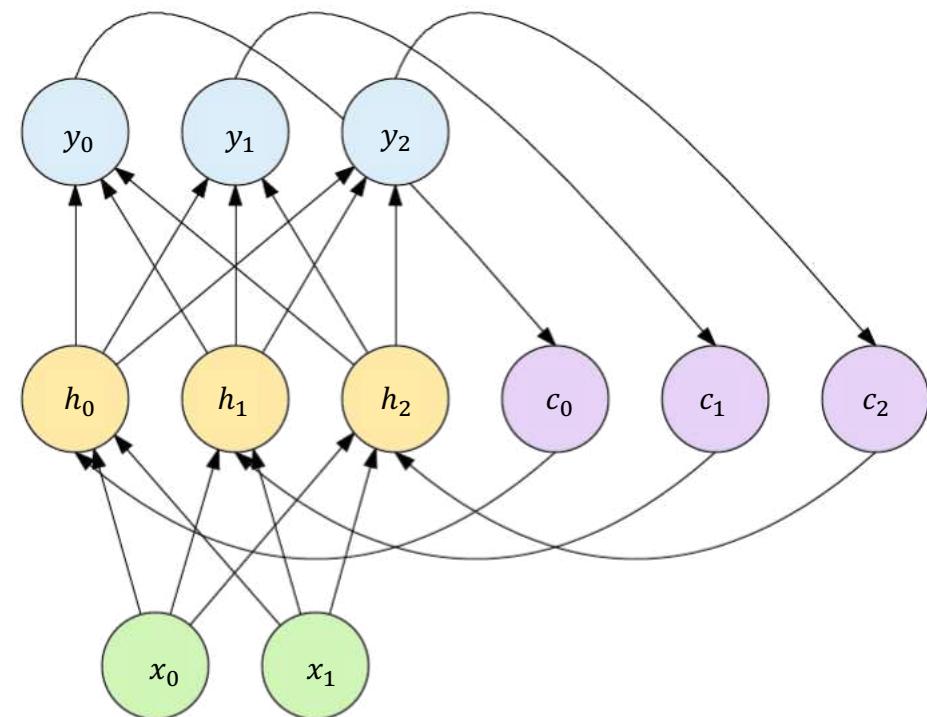
$$\mathbf{y}(1) = \begin{pmatrix} 0.41 \\ 0.37 \end{pmatrix}, \mathbf{y}(2) = \begin{pmatrix} 0.97 \\ 0.11 \end{pmatrix}, \mathbf{y}(3) = \begin{pmatrix} 0.86 \\ 0.18 \end{pmatrix}, \mathbf{y}(4) = \begin{pmatrix} 0.55 \\ 0.68 \end{pmatrix}$$

Types of RNN

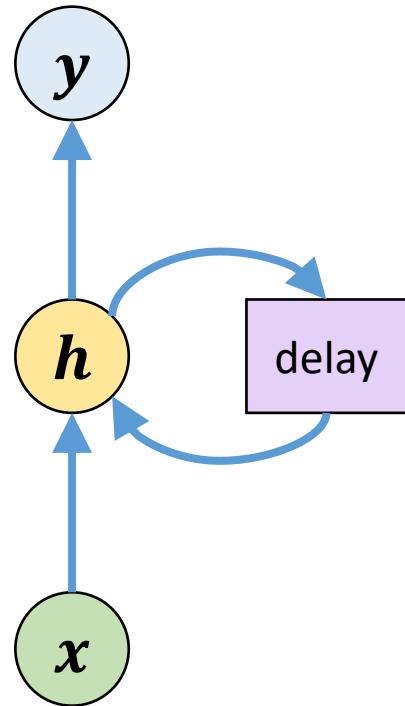
RNN with hidden recurrence
(Elman-type)



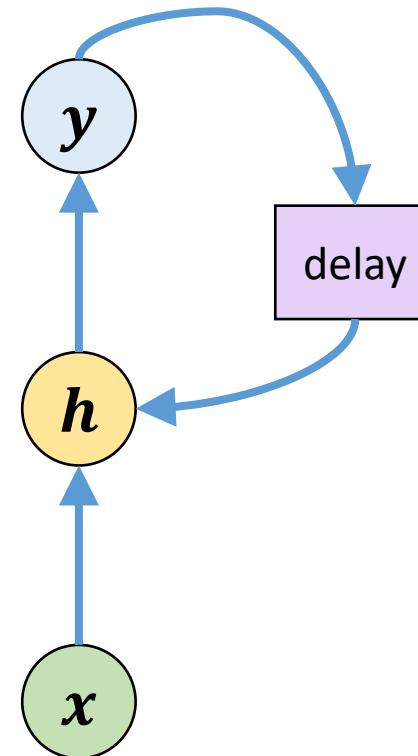
RNN with top-down recurrence
(Jordan-type)



RNN with top-down recurrence (Jordan type)



The **hidden-layer activation** at time $t - 1$ is kept by the **delay unit** and fed to the hidden layer at time t together with the raw input $x(t)$.



The **output of the output-layer** at time $t - 1$ is kept by the **delay unit** and fed to the hidden layer at time t together with the raw input $x(t)$ for time t .

- $\mathbf{h}(t) = \phi(\mathbf{U}^T \mathbf{x}(t) + \mathbf{W}^T \mathbf{h}(t-1) + \mathbf{b})$
- $\mathbf{y}(t) = \sigma(\mathbf{V}^T \mathbf{h}(t) + \mathbf{c})$

- $\mathbf{h}(t) = \phi(\mathbf{U}^T \mathbf{x}(t) + \mathbf{W}^T \mathbf{y}(t-1) + \mathbf{b})$
- $\mathbf{y}(t) = \sigma(\mathbf{V}^T \mathbf{h}(t) + \mathbf{c})$

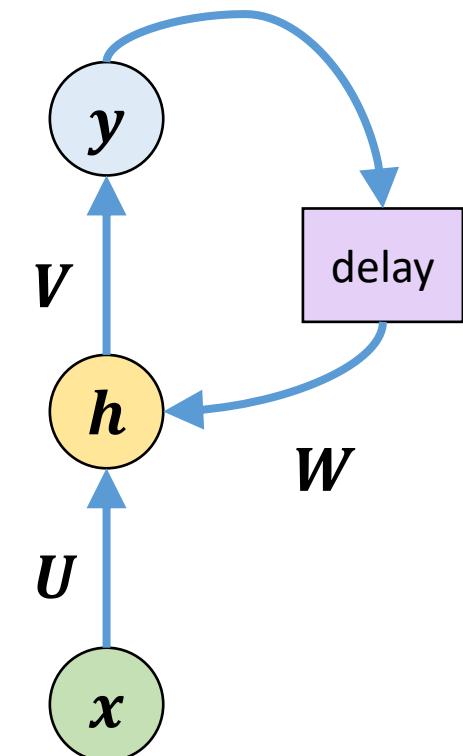
RNN with top-down recurrence

Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

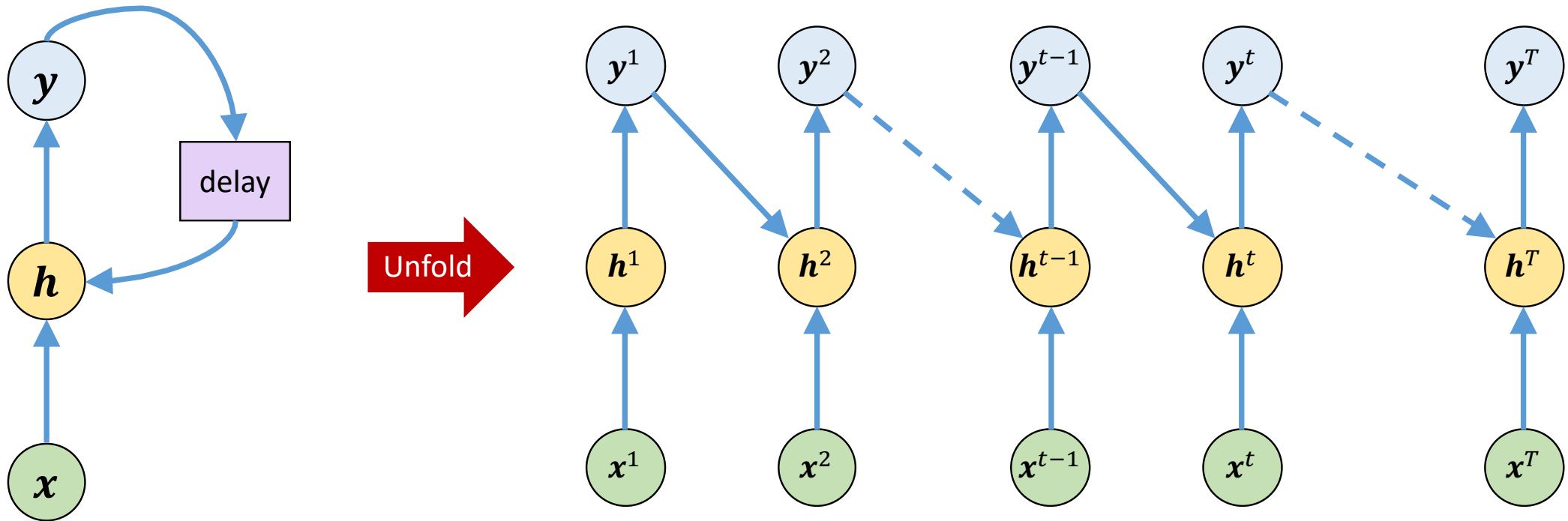
Activation of the Jordan-type RNN with one hidden-layer is given by:

$$\begin{aligned} h(t) &= \phi(U^T x(t) + W^T y(t-1) + b) \\ y(t) &= \sigma(V^T h(t) + c) \end{aligned}$$

Note that output of the previous time instant is fed back to the hidden layer and W represents the recurrent weight matrix connecting previous output to the current hidden input

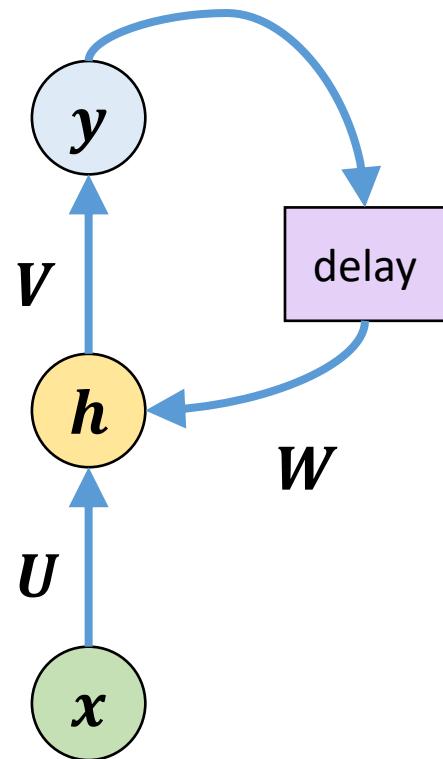


RNN with top-down recurrence



The recurrent connections in the hidden-layer is unfolded and represented in time for processing a time series $(x(t))_{t=1}^T$

RNN with top-down recurrence: batch processing



Given P patterns $\{x_p\}_{p=1}^P$ where $x_p = (x_p(t))_{t=1}^T$,

$$X(t) = \begin{pmatrix} x_1(t)^T \\ x_2(t)^T \\ \vdots \\ x_P(t)^T \end{pmatrix}$$

Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Jordan-type RNN is given by:

$$H(t) = \phi(X(t)U + Y(t-1)W + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Example 2

A recurrent neural network with top-down recurrence receives 2-dimensional input sequences and produce 1-dimensional output sequences. It has three hidden neurons and following weight matrices and biases:

Weight matrix connecting input to the hidden layer $\mathbf{U} = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix}$

Recurrence weight matrix connecting previous output to the hidden layer $\mathbf{W} = \begin{pmatrix} 2.0 & 1.3 & -1.0 \end{pmatrix}$

Weight matrix connecting hidden layer to the output $\mathbf{V} = \begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix}$

Bias to the hidden layer $\mathbf{b} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}$ and bias to the output layer $\mathbf{c} = 0.1$.

Given the following two input sequences:

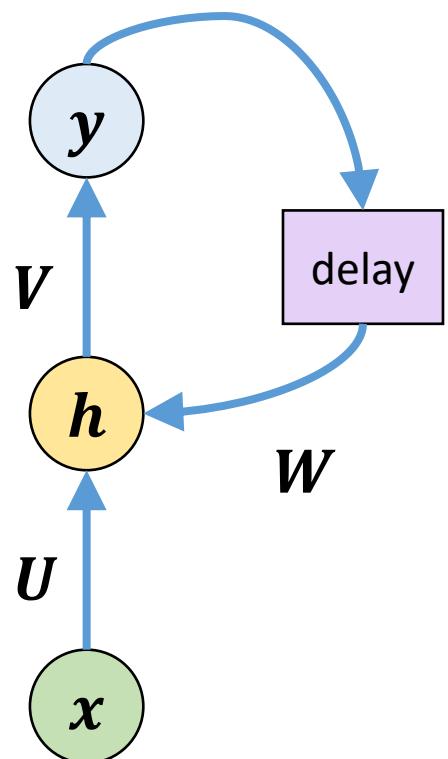
$$\mathbf{x}_1 = \begin{pmatrix} (1) & (-1) & (0) \\ (2) & (1) & (3) \end{pmatrix}$$

$$\mathbf{x}_2 = \begin{pmatrix} (-1) & (2) & (3) \\ (0) & (-1) & (-1) \end{pmatrix}$$

Using batch processing, find output sequences. Assume outputs are initialized to zero at the beginning. Assume tanh and sigmoid activations for hidden and output layer, respectively.

eg8.2.ipynb

Example 2 (con't)



$$\begin{aligned} \mathbf{x}_1 &= \begin{pmatrix} 1 \\ 2 \\ -1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} -1 \\ 1 \\ 3 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 3 \end{pmatrix} \\ \mathbf{x}_2 &= \begin{pmatrix} -1 \\ 2 \\ -1 \end{pmatrix} \quad \begin{pmatrix} 2 \\ -1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ -1 \end{pmatrix} \end{aligned}$$

Two sequences as batch of sequences:

$$\mathbf{x} = \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} & \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix} & \begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix} \end{pmatrix}$$

Three hidden neurons and one output neuron.

$$\begin{aligned} \mathbf{H}(t) &= \phi(\mathbf{X}(t)\mathbf{U} + \mathbf{Y}(t-1)\mathbf{W} + \mathbf{B}) \\ \mathbf{Y}(t) &= \sigma(\mathbf{H}(t)\mathbf{V} + \mathbf{C}) \end{aligned}$$

Initially,

$$\mathbf{Y}(0) = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

Example 2 (con't)

At $t = 1$: $\mathbf{X}(1) = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}$

$$\begin{aligned}\mathbf{H}(1) &= \tanh(\mathbf{X}(1)\mathbf{U} + \mathbf{Y}(0)\mathbf{W} + \mathbf{B}) \\ &= \tanh \left(\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} (2.0 \quad 1.3 \quad -1.0) + \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.2 & 0.72 & -1.0 \\ 0.83 & -0.29 & 0.0 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{Y}(1) &= \text{sigmoid}(\mathbf{H}(1)\mathbf{V} + \mathbf{C}) \\ &= \text{sigmoid} \left(\begin{pmatrix} 0.2 & 0.72 & -1.0 \\ 0.83 & -0.29 & 0.0 \end{pmatrix} \begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.31 \\ 0.90 \end{pmatrix}\end{aligned}$$

Example 2 (con't)

At $t = 2$: $\mathbf{X}(2) = \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}$

$$\begin{aligned}\mathbf{H}(2) &= \tanh(\mathbf{X}(2)\mathbf{U} + \mathbf{Y}(1)\mathbf{W} + \mathbf{B}) \\ &= \tanh \left(\begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix} \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.31 \\ 0.9 \end{pmatrix} \begin{pmatrix} 2.0 & 1.3 & -1.0 \end{pmatrix} + \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{Y}(2) &= \text{sigmoid}(\mathbf{H}(2)\mathbf{V} + \mathbf{C}) \\ &= \text{sigmoid} \left(\begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix} \begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix}\end{aligned}$$

Example 2 (con't)

At $t = 3$: $\mathbf{X}(3) = \begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}$

$$\begin{aligned}\mathbf{H}(3) &= \tanh(\mathbf{X}(3)\mathbf{U} + \mathbf{Y}(1)\mathbf{W} + \mathbf{B}) \\ &= \tanh \left(\begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix} \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix} (2.0 \quad 1.3 \quad -1.0) + \begin{pmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{pmatrix} \right) \\ &= \begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.00 & 0.94 & 0.99 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{Y}(3) &= \text{sigmoid}(\mathbf{H}(3)\mathbf{V} + \mathbf{C}) \\ &= \text{sigmoid} \left(\begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.0 & 0.94 & 0.99 \end{pmatrix} \begin{pmatrix} 2.0 \\ -1.5 \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.63 \\ 0.04 \end{pmatrix}\end{aligned}$$

Example 2 (con't)

Output (batch):

$$\begin{aligned}\mathbf{Y} &= (\mathbf{Y}(1) \quad \mathbf{Y}(2) \quad \mathbf{Y}(3)) \\ &= \left(\begin{pmatrix} 0.31 \\ 0.9 \end{pmatrix} \quad \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix} \quad \begin{pmatrix} 0.63 \\ 0.04 \end{pmatrix} \right)\end{aligned}$$

Outputs for inputs

$$\begin{aligned}x_1 &= \left(\begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 3 \end{pmatrix} \right) \\ x_2 &= \left(\begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 2 \\ -1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ -1 \end{pmatrix} \right)\end{aligned}$$

are

$$\begin{aligned}y_1 &= (0.31, 0.83, 0.63) \\ y_2 &= (0.9, 0.11, 0.04)\end{aligned}$$

Backpropagation through time (BPTT)

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

$$\begin{aligned}\mathbf{h}(t) &= \tanh(\mathbf{U}^T \mathbf{x}(t) + \mathbf{W}^T \mathbf{h}(t-1) + \mathbf{b}) \\ \mathbf{u}(t) &= \mathbf{V}^T \mathbf{h}(t) + \mathbf{c}\end{aligned}$$

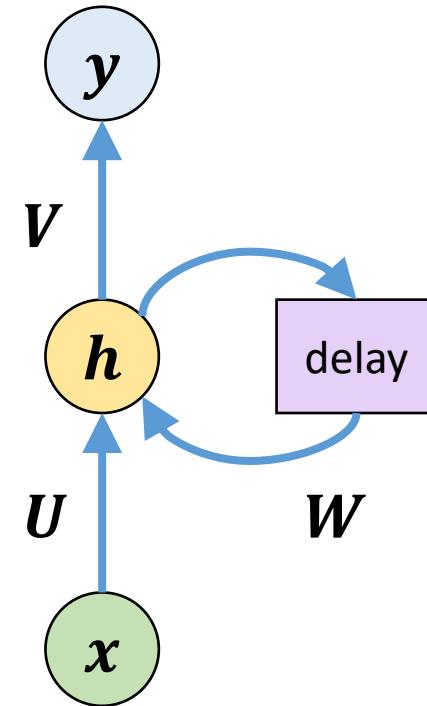
For classification,

$$\mathbf{y}(t) = \text{softmax}(\mathbf{u}(t))$$

For regression,

$$\mathbf{y}(t) = \mathbf{u}(t)$$

Learnable parameters



\mathbf{U} : weight vector that transforms raw inputs to the hidden-layer

\mathbf{W} : recurrent weight vector connecting previous hidden-layer output to hidden input

\mathbf{V} : weight vector of the output layer

\mathbf{b} : bias connected to hidden layer

\mathbf{c} : bias connected to the output layer

Backpropagation through time (BPTT)

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

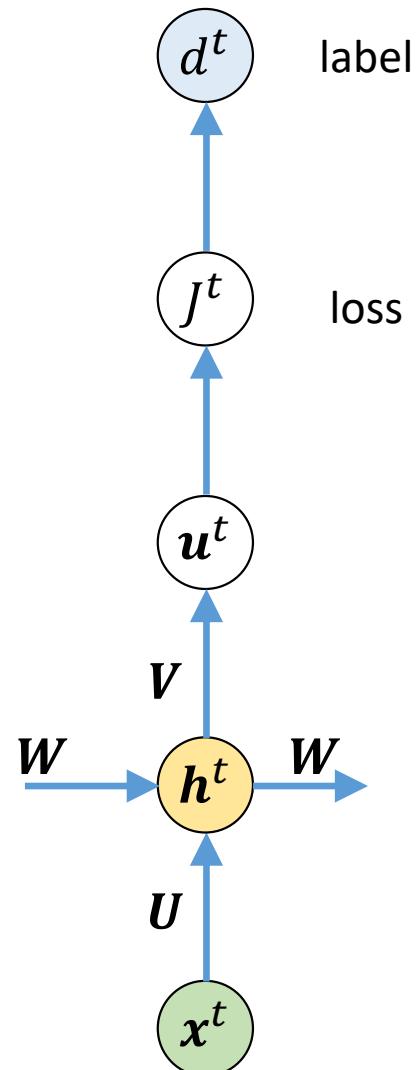
$$\begin{aligned}\mathbf{h}(t) &= \tanh(\mathbf{U}^\top \mathbf{x}(t) + \mathbf{W}^\top \mathbf{h}(t-1) + \mathbf{b}) \\ \mathbf{u}(t) &= \mathbf{V}^\top \mathbf{h}(t) + \mathbf{c}\end{aligned}$$

For classification,

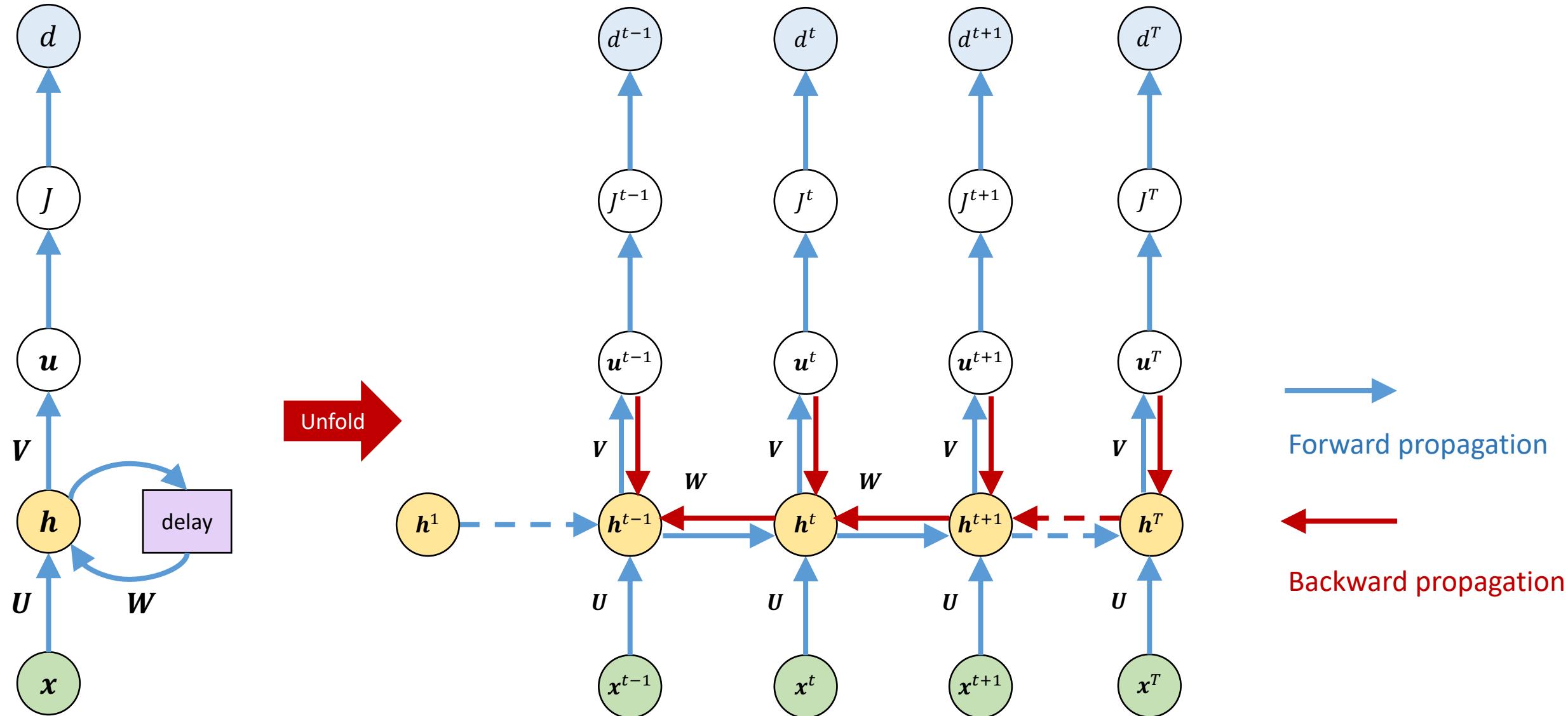
$$\mathbf{y}(t) = \text{softmax}(\mathbf{u}(t))$$

For regression,

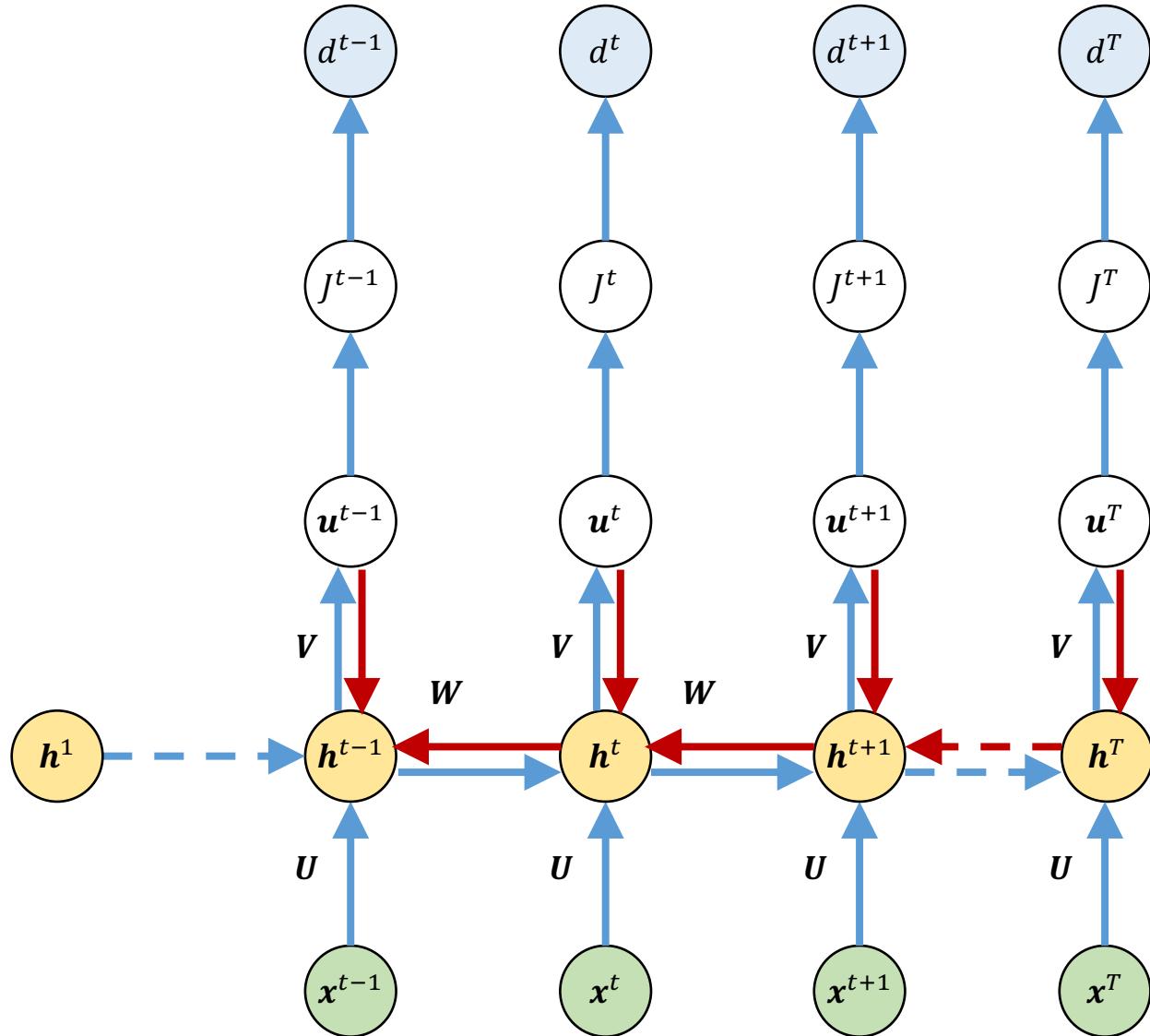
$$\mathbf{y}(t) = \mathbf{u}(t)$$



Backpropagation through time (BPTT)



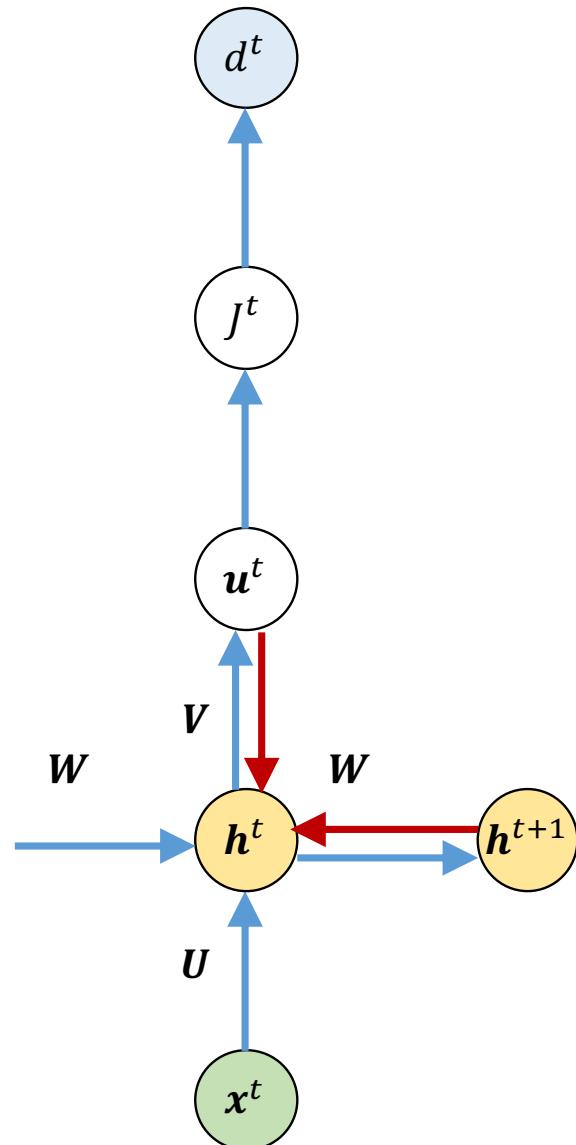
Backpropagation through time (BPTT)



The gradients propagate **backward**, starting from the end of the sequence from **two directions**:

- Top-down direction
- Reverse sequence direction

Backpropagation through time (BPTT)



The gradient computation involves performing a **forward propagation pass moving left to right** through the unfolded graph, followed by a **backpropagation pass moving right to left** through the graph. The gradients are propagated from the final time point to the initial time points .

The runtime is $O(T)$ where T is the length of input sequence and cannot be reduced by parallelization because the forward propagation is inherently **sequential**. The back-propagation algorithm applied to the unrolled graph with $O(T)$ cost is called *back-propagation through time (BPTT)*.

The network with recurrence between hidden units is thus very powerful but also **expensive** to train.

Example 3

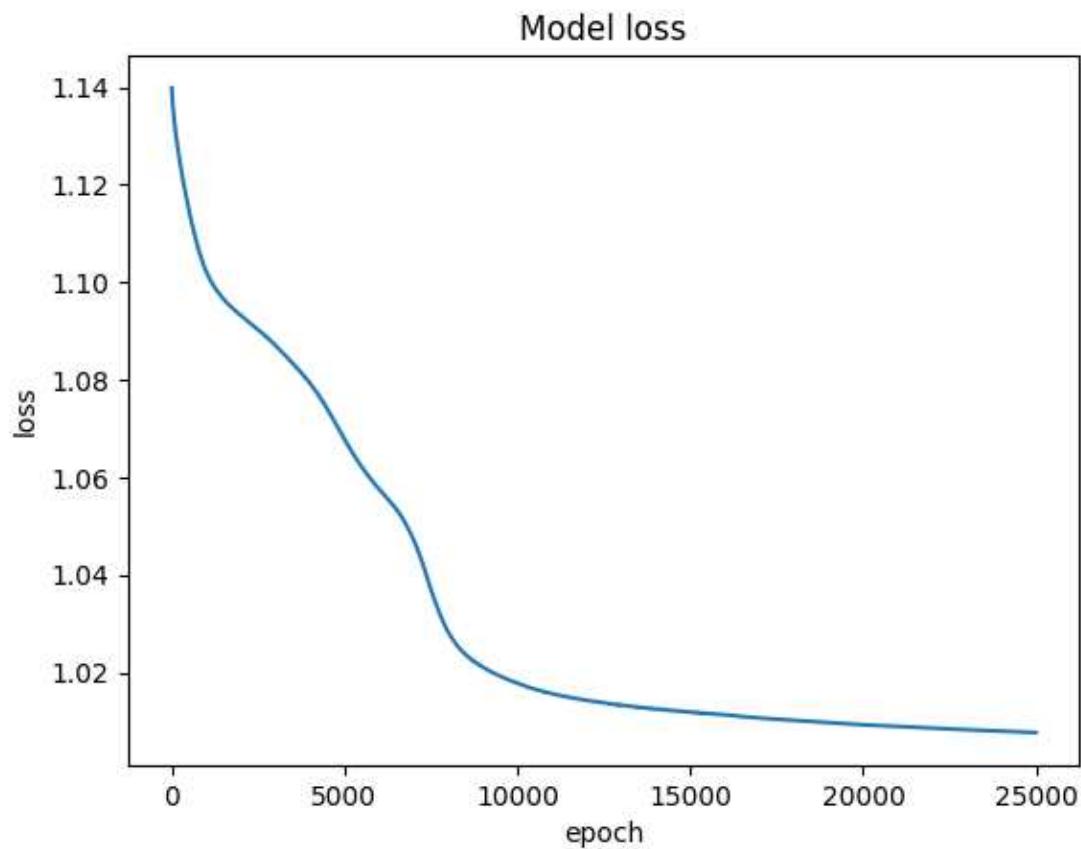
Generate 8-dimensional 16 input sequences of 64 time-steps with each input is a random number between [0.0, 1.0].

Generate the corresponding 1-dimensional labels by randomly generating a number from [0, 1, 2].

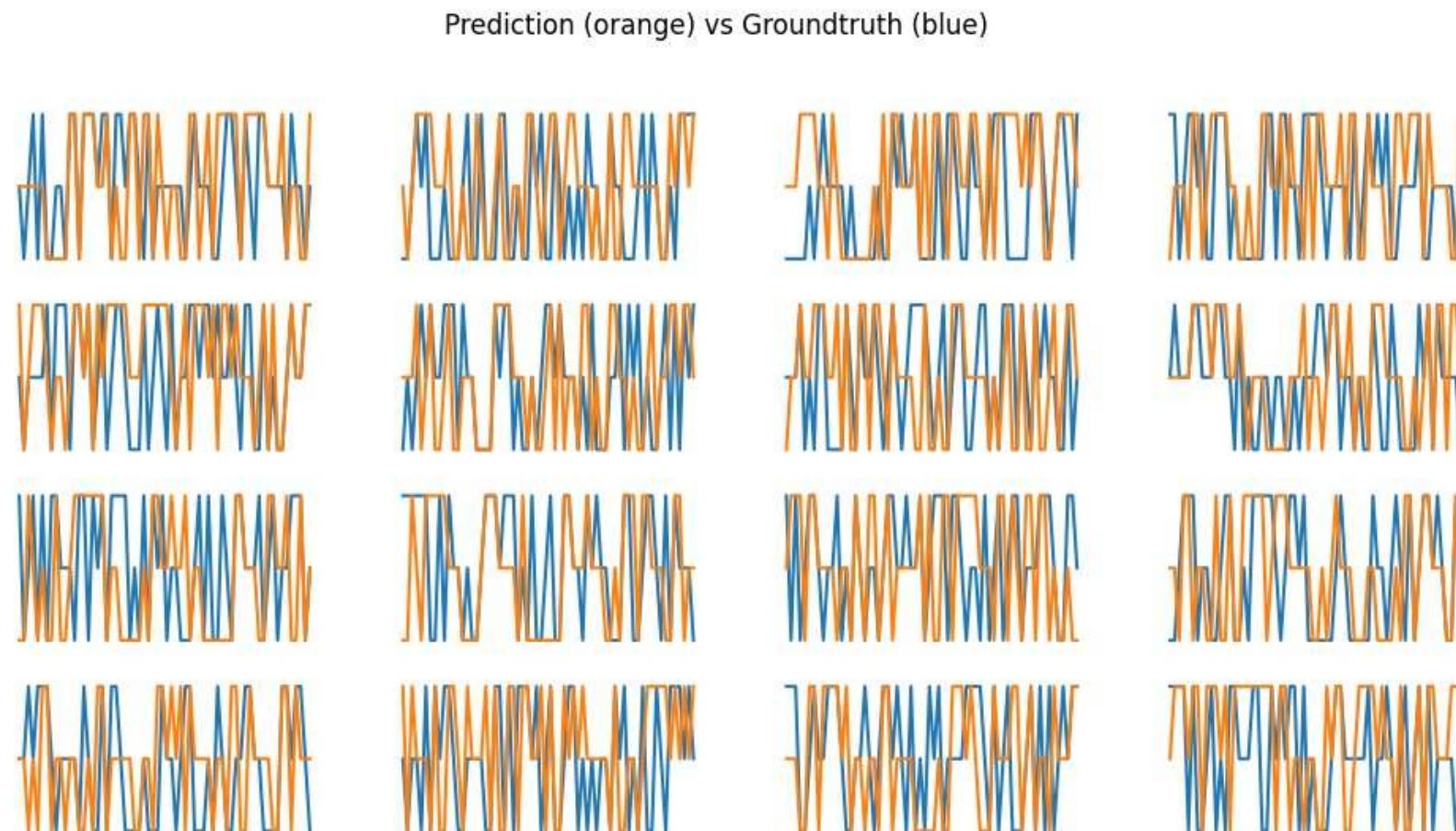
Create an RNN with one hidden layer of 5 neurons.

Plot the learning curves and predicted labels.

Example 3 (con't)



Example 3 (con't)



Long Short-Term Memory (LSTM)

Long-term dependency

“The man who ate my pizza has purple hair”

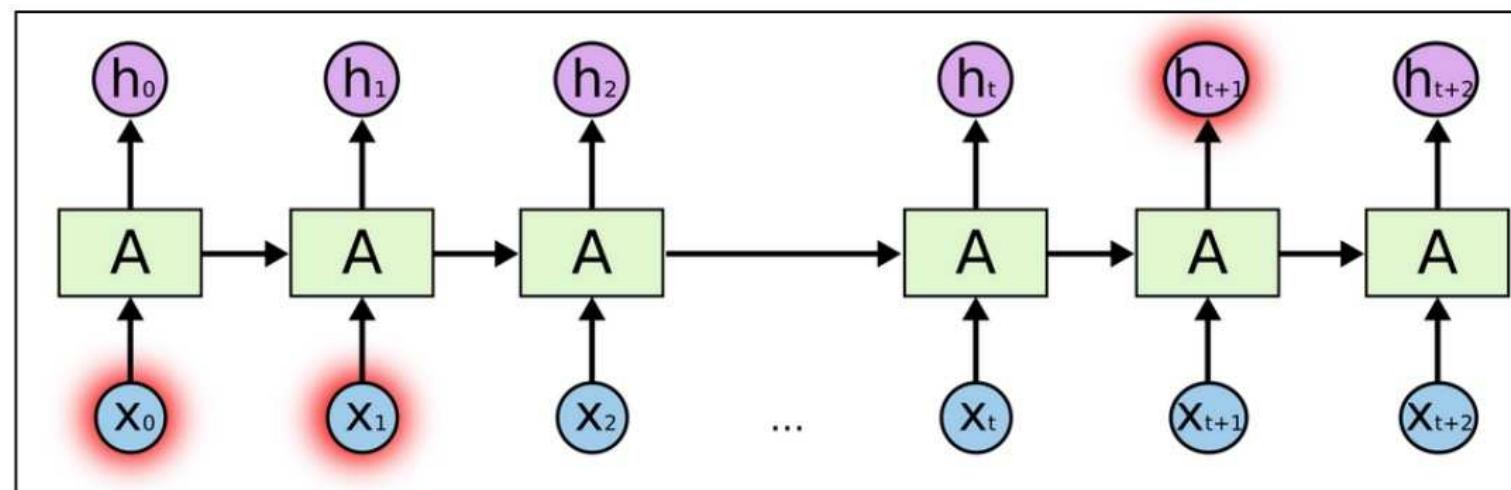
Purple hair is for the man and not the pizza!
This is a long term dependency.

There are cases where we need even more context

- To predict last word in “I grew up in France...<long paragraph>...I speak *French*”
- Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is *French*

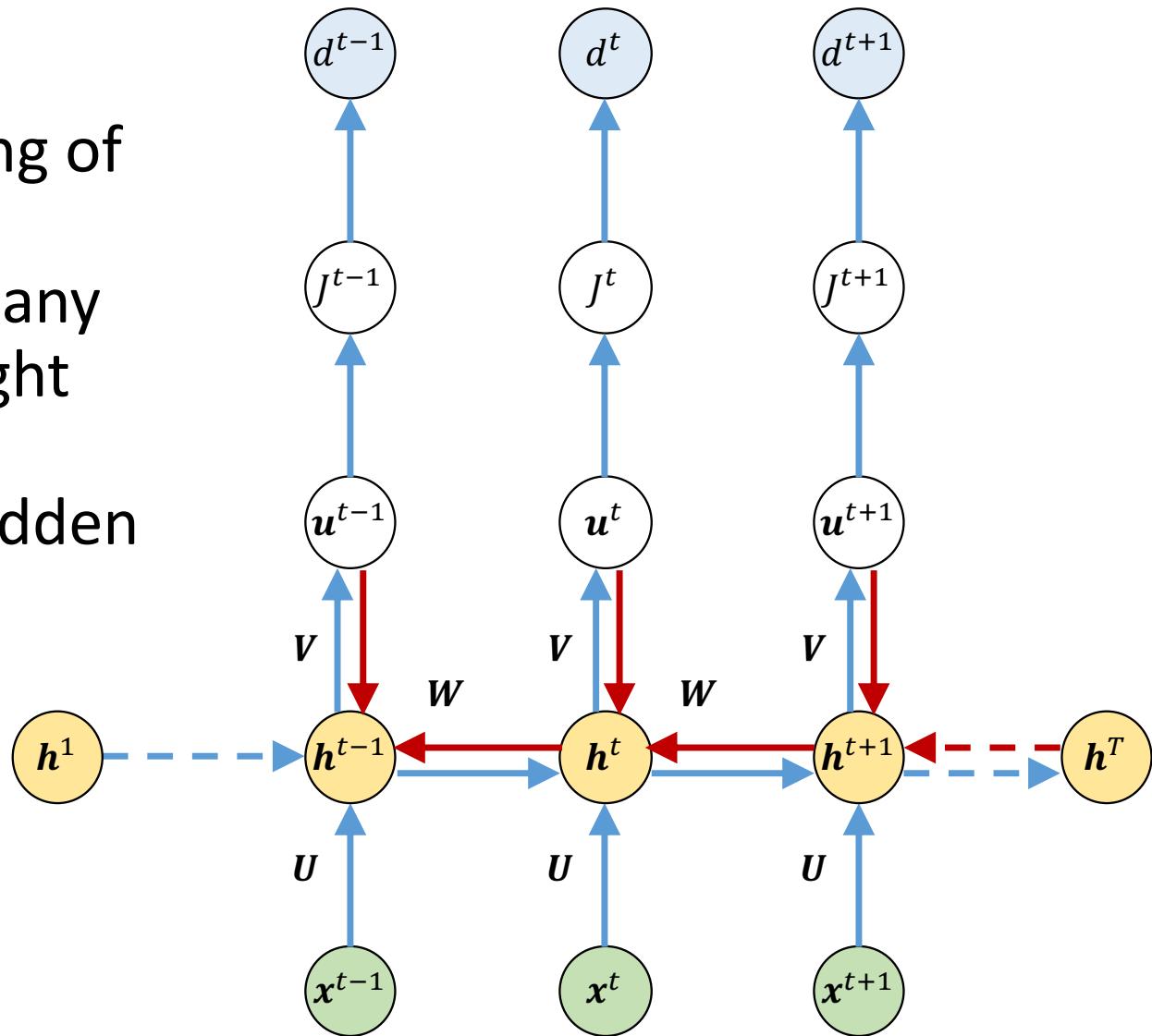
Long-term dependency

- RNNs work upon the fact that the result of an information is dependent on its previous state or previous n time steps
- RNNs have difficulty in learning **long range dependencies**
- Gap between relevant information and where it is needed is large



Exploding and vanishing gradients in RNN

During gradient back-propagation learning of RNN, the gradient can end up being **multiplied a large number of times** (as many as the number of time steps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer.

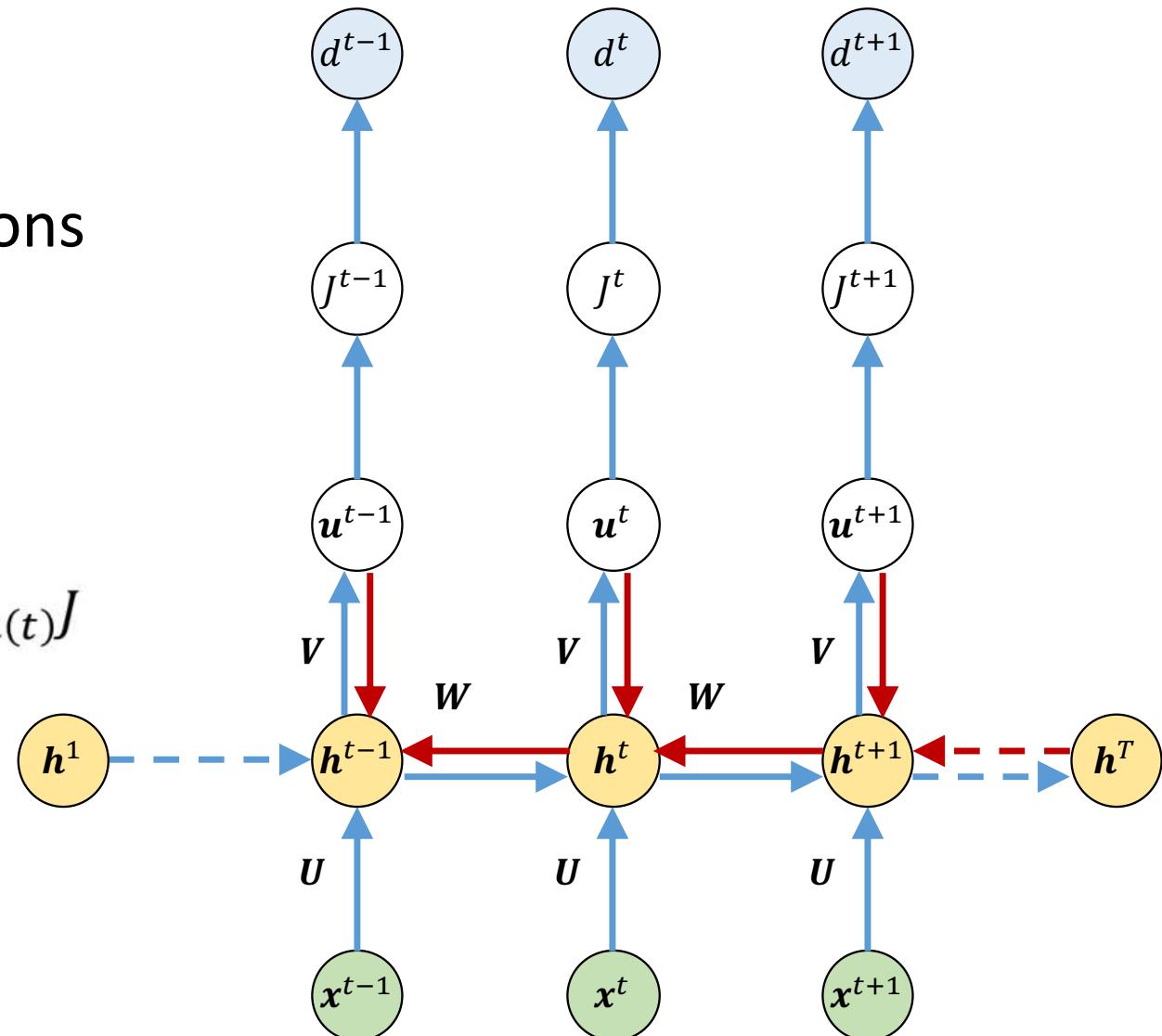


Exploding and vanishing gradients in RNN

Note that each time the activations are **forward propagated** in time, the activations are **multiplied by W** and each time the gradients are **back propagated**, the gradients are multiplied by W^T .

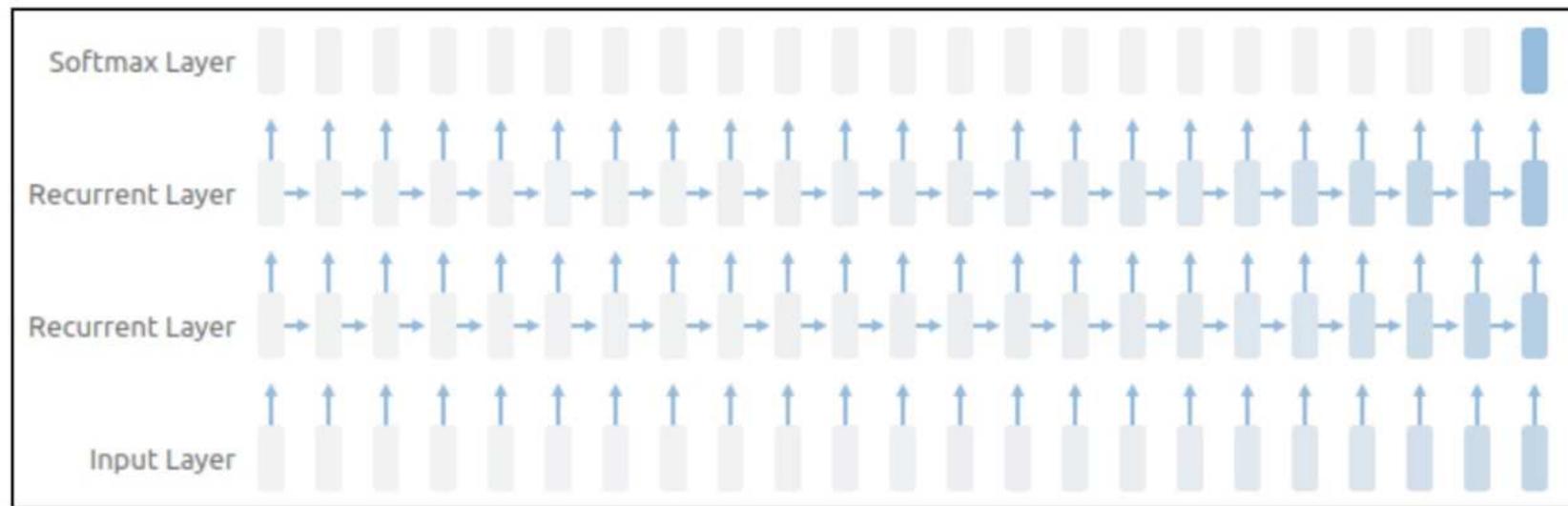
$$\nabla_{\mathbf{h}(t)} J = \mathbf{W} \text{diag}(1 - \mathbf{h}^2(t+1)) \nabla_{\mathbf{h}(t+1)} J + \mathbf{V} \nabla_{\mathbf{u}(t)} J$$

Gradient from reverse direction



Exploding and vanishing gradients in RNN

If the weights in this matrix are **small**, the recursive derivative can lead to a situation called **vanishing gradients** where the gradient signal gets so small that **learning either becomes very slow or stops working altogether**



Contribution from the earlier steps becomes insignificant in the gradient

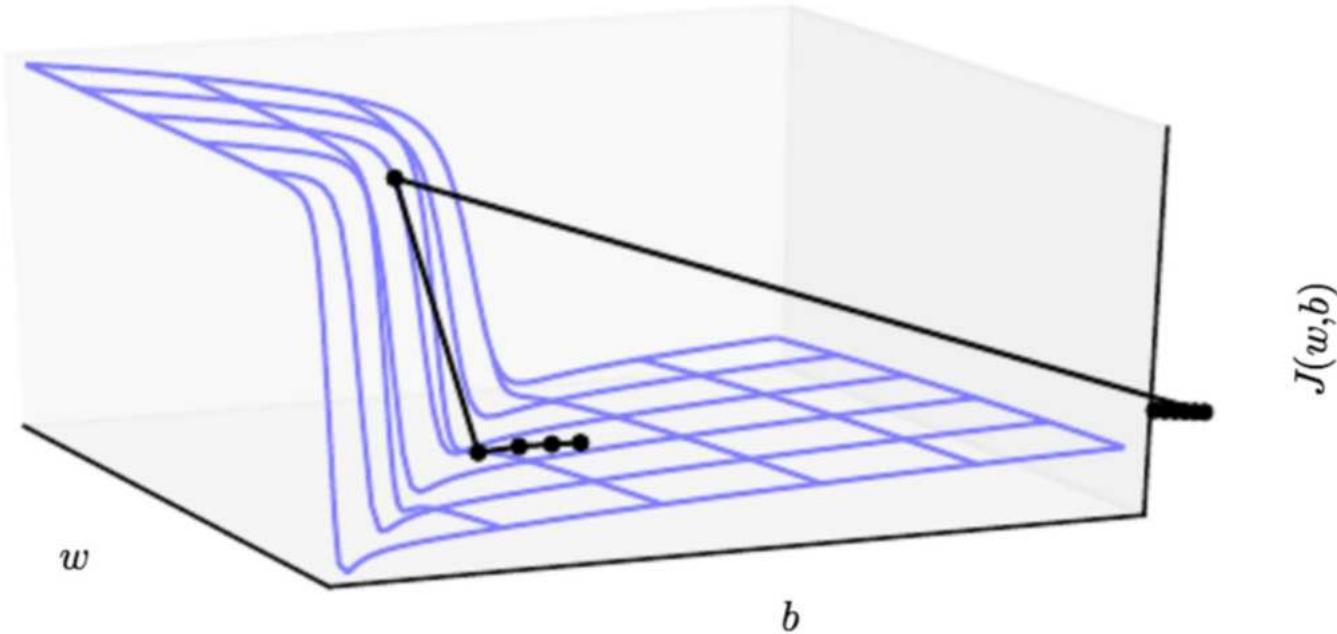
Exploding and vanishing gradients in RNN

Conversely, if the weights in this matrix are **large**, it can lead to a situation where the gradient signal is so large that it can **cause learning to diverge**. This is often referred to as **exploding gradients**.

Exploding gradient easily solved by clipping the gradients at a predefined threshold value.

Vanishing gradient is more concerning

Gradient Clipping



Due to long term dependencies, RNN tend to have gradients having very large or very small magnitudes. The large gradients resemble cliffs in the error landscape and when the gradient descent encounters the gradient updates can move the parameters away from true minimum.

A gradient clipping is employed to avoid the gradients to become too large.

Gradient Clipping

Commonly, two methods are used:

1. Clip the gradient g when it exceeds a threshold:

```
if ||g|| > ν:  
    ||g|| ← ν
```

2. Normalize the gradient when it exceeds a threshold:

```
if ||g|| > ν:  
    ||g|| ←  $\frac{g}{||g||} \nu$ 
```

Exploding and vanishing gradients in RNN

Vanishing and exploding gradients in gradient backpropagation learning makes it difficult to train RNN to learn long-term dependencies.

Solution: Gated RNNs

- Long short-term memory (LSTM), Gated Recurrent Units (GRU)

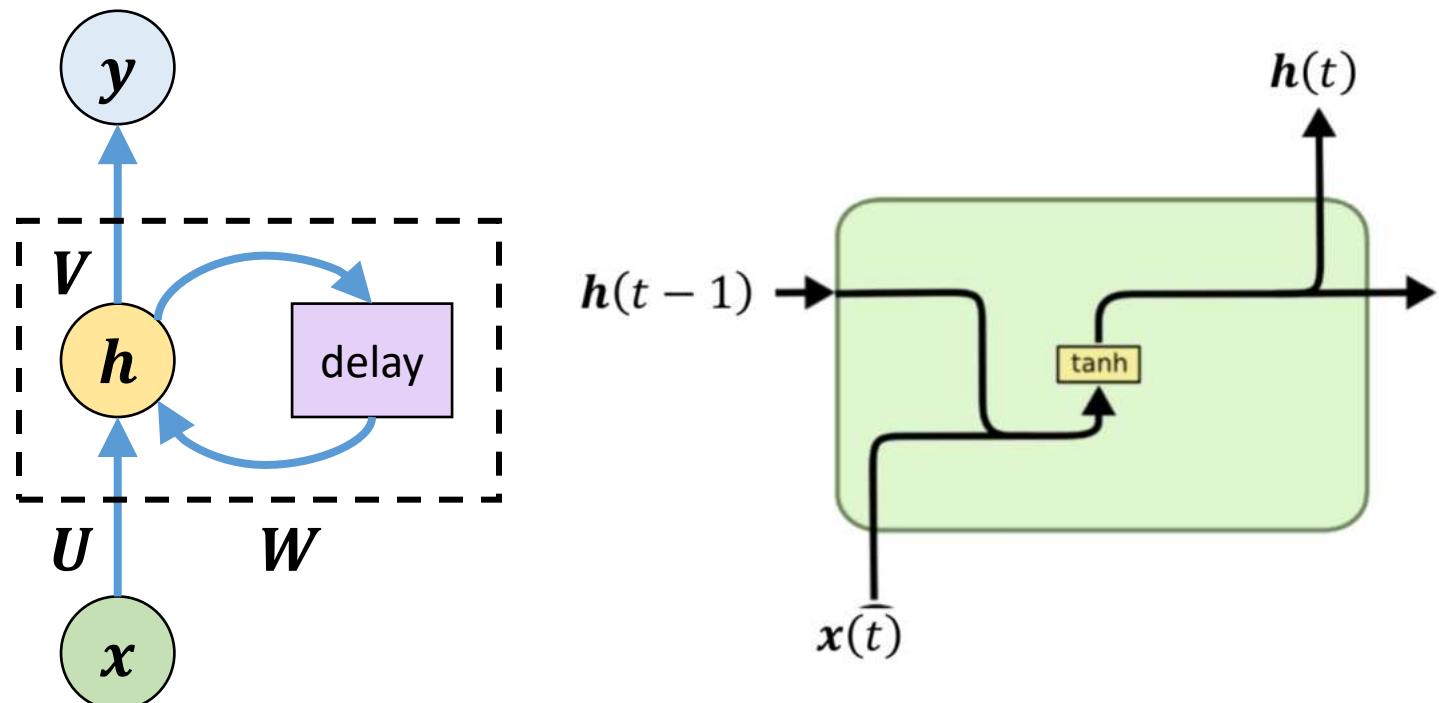
Basic RNN unit

The **RNN cell (memory unit)** is characterized by

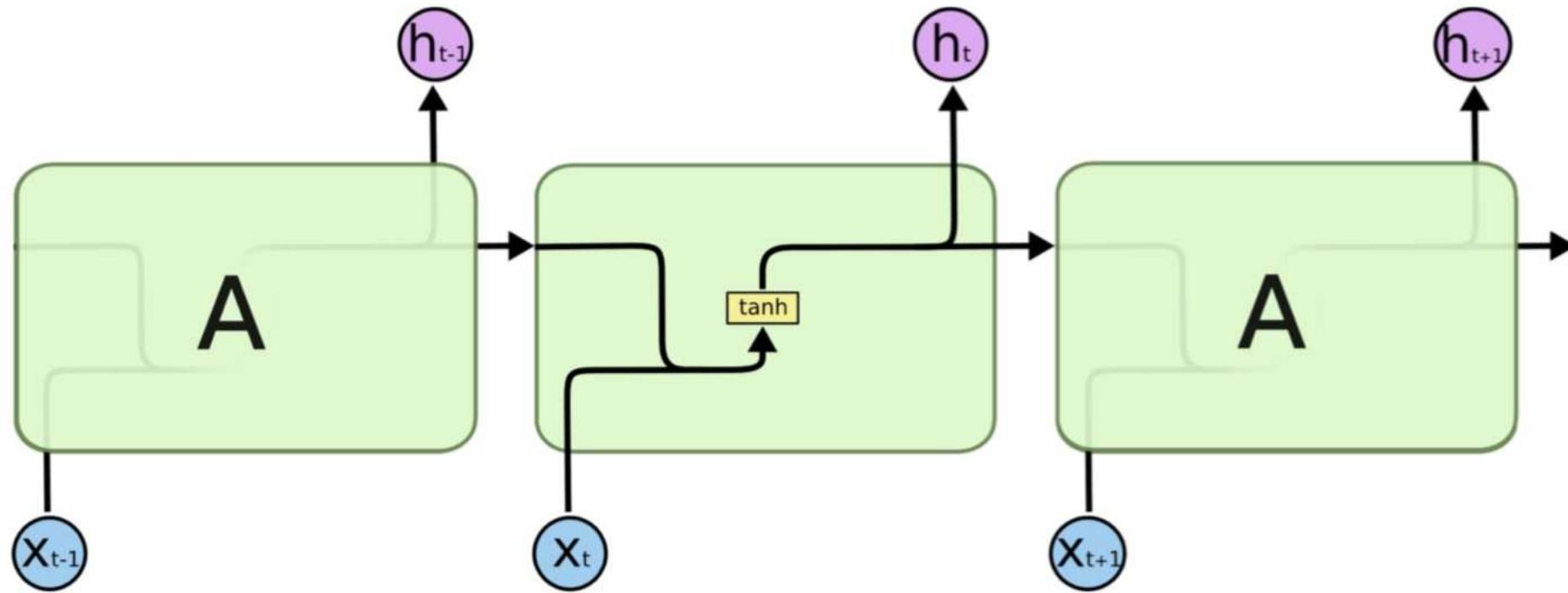
$$\mathbf{h}(t) = \phi(\mathbf{U}^T \mathbf{x}(t) + \mathbf{W}^T \mathbf{h}(t - 1) + \mathbf{b})$$

where ϕ is the *tanh* activation function and RNN cell is referred to as *tanh* units.

RNN build with simple *tanh* units are also referred to as **vanilla RNN**.



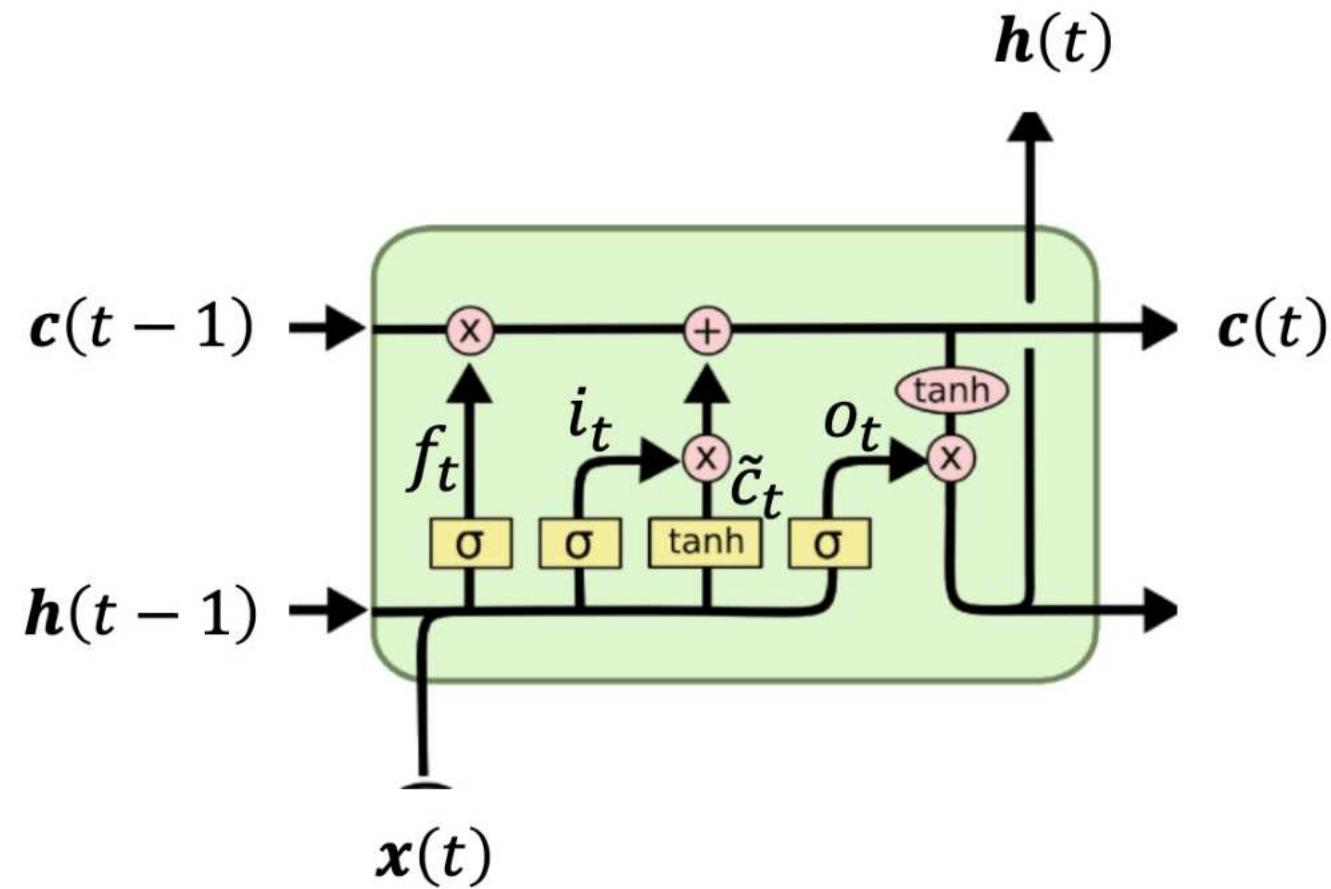
Basic RNN layer



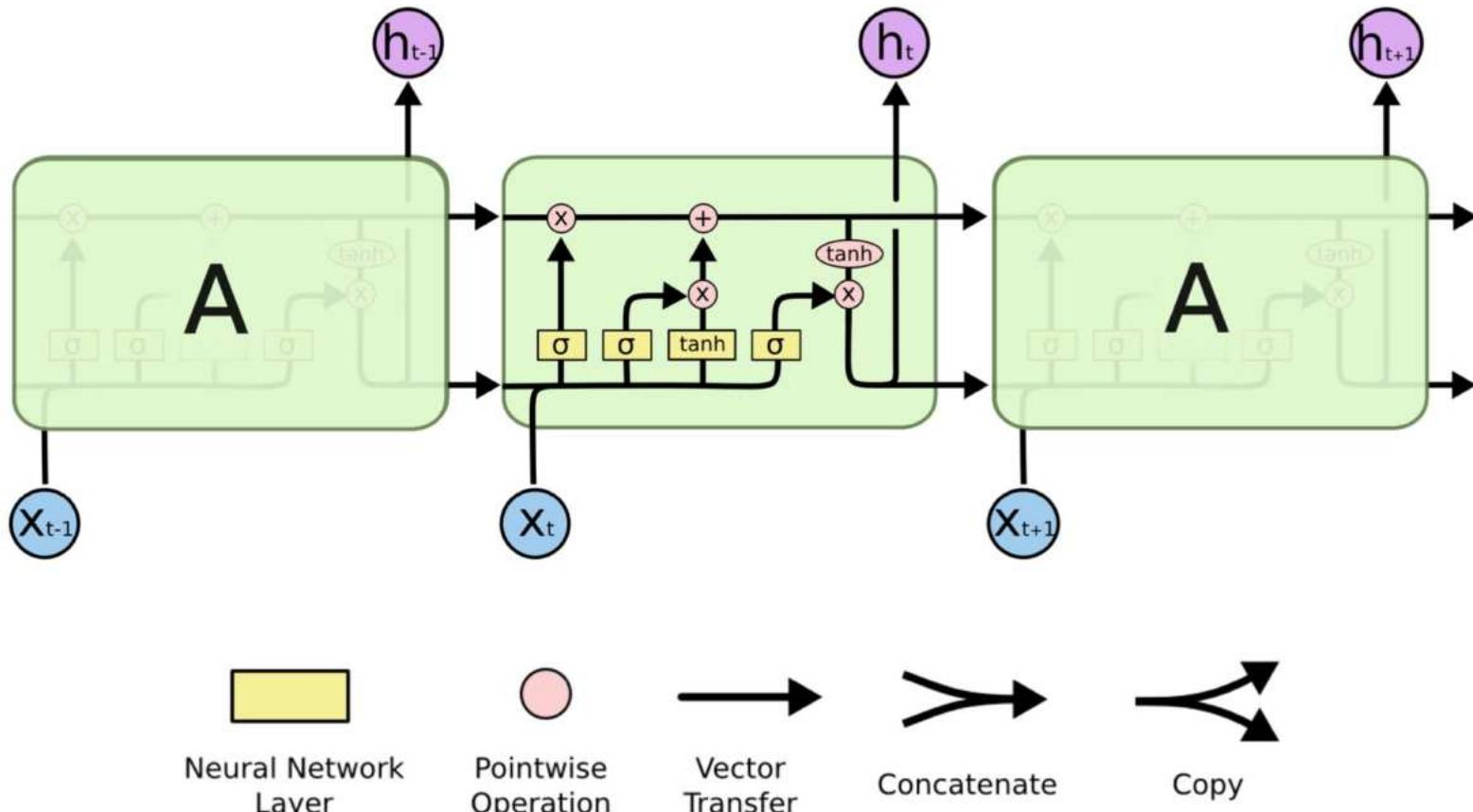
Long short-term memory (LSTM) unit

Key of LSTM is "state"

- A persistent module called the cell-state
- "State" is a representation of past history
- It comprises a common thread through time



Long short-term memory (LSTM) unit

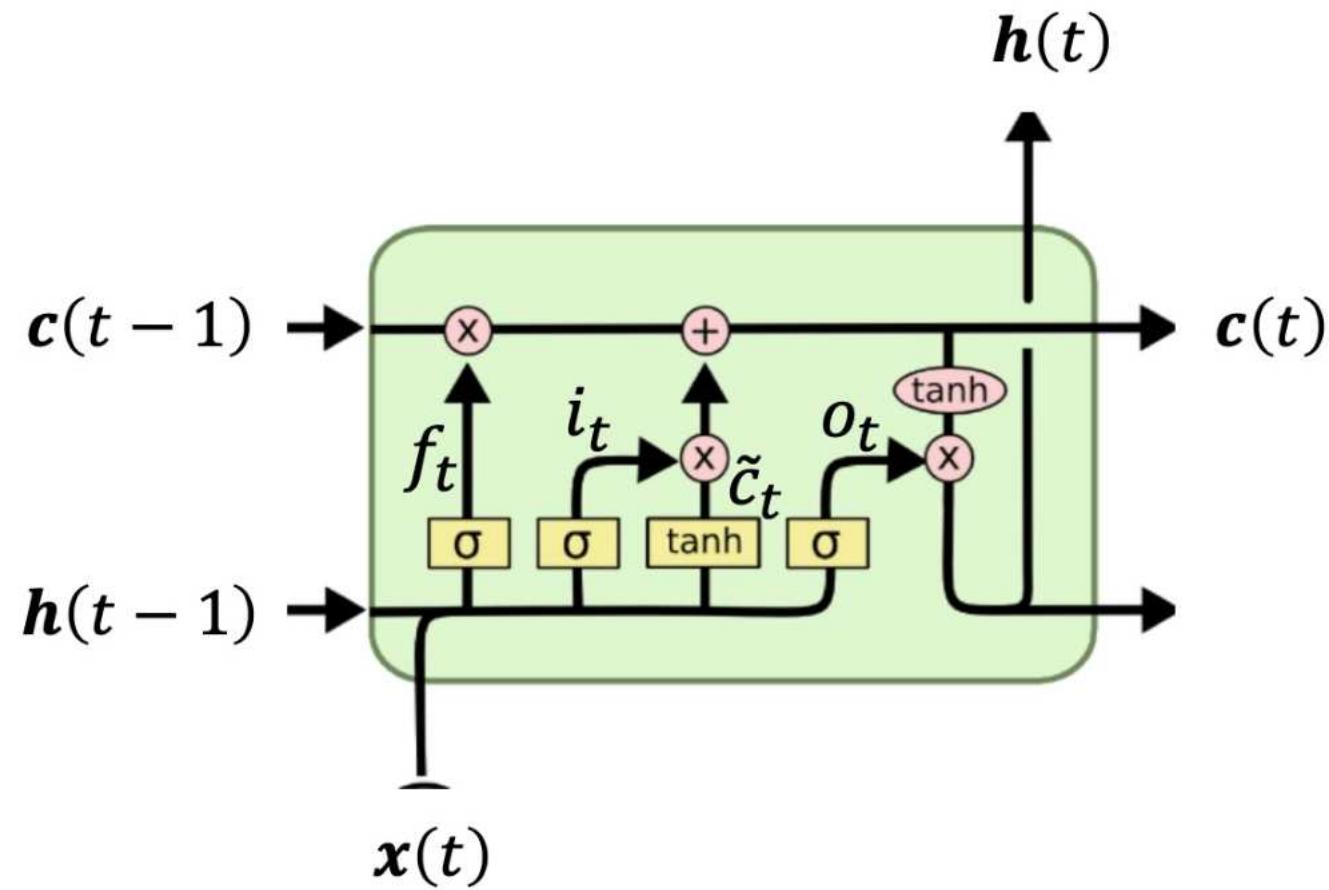


Cells are connected recurrently to each other - Replacing hidden units of ordinary recurrent networks

Long short-term memory (LSTM) unit

LSTMs provide a solution by incorporating memory units that allow the network to learn when to **forget previous hidden states** and when to **update hidden states** given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.

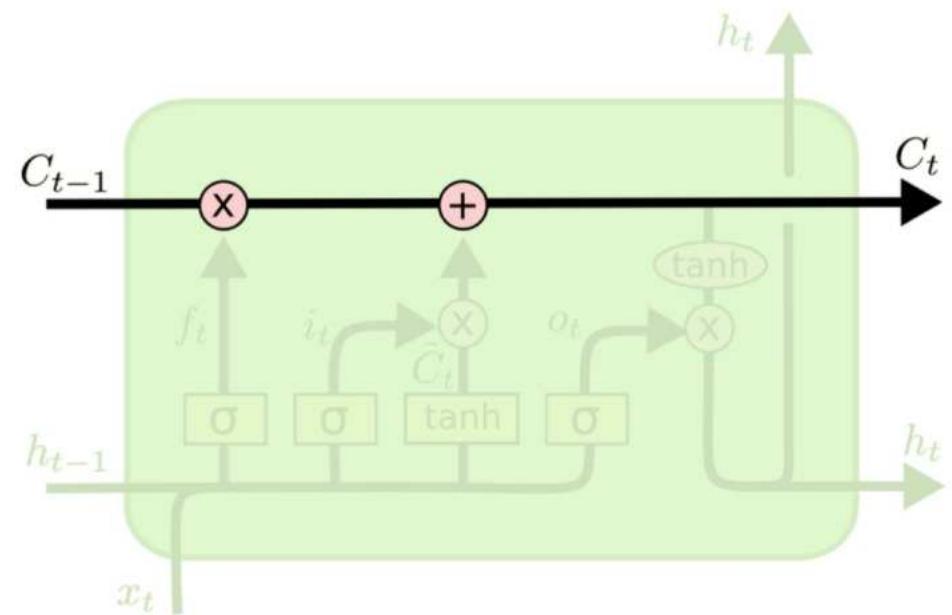


Long short-term memory (LSTM) unit

The key to LSTM is the **cell state $c(t)$** - the horizontal line through the top of the diagram.

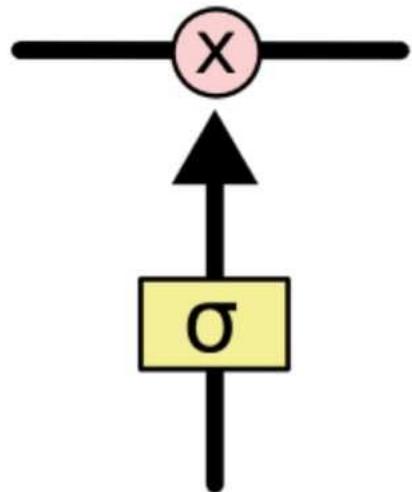
The long-term memory.

Like a conveyor belt that runs through entire chain with minor interactions



Gates

The LSTM has the ability to add and remove information to the cell states through gates.



Gates are a way to **optionally let information through**.

They are composed of sigmoid neural net layer and pointwise multiplication operations.

Gates

The sigmoid layer outputs numbers between **zero** and **one**, describing how much of each component should be let through.

A value of **zero** means “let nothing through,” while a value of **one** means “let everything through”



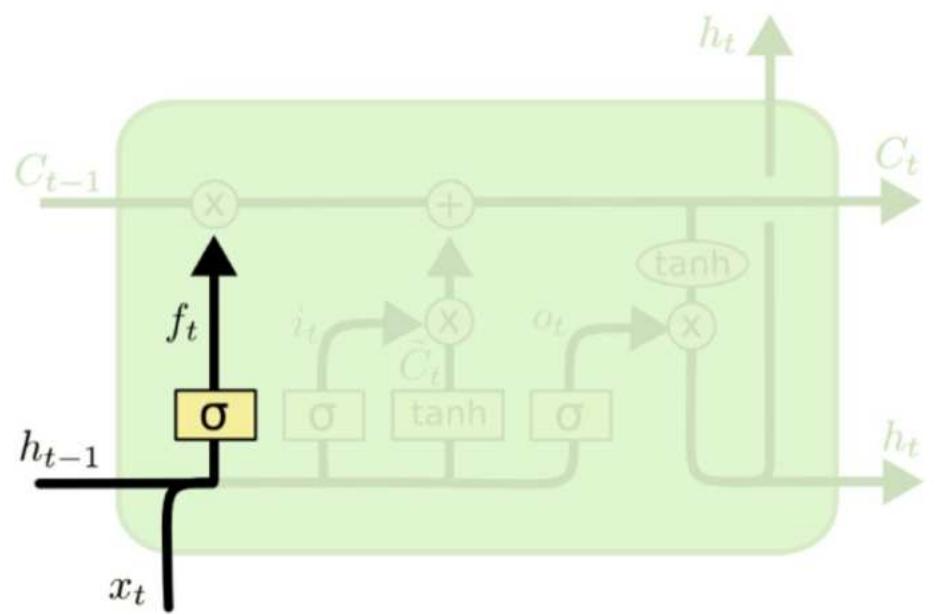
An LSTM has **three** of these gates to control cell state.

Forget gate

The forget gate can modulate the memory cell's self-recurrent connection, allowing the cell **to remember or forget its previous state**, as needed.

$$f(t) = \sigma \left(U_f^T x(t) + W_f^T h(t-1) + b_f \right)$$

Value of $f(t)$ determines if $c(t-1)$ is to be remembered or not.



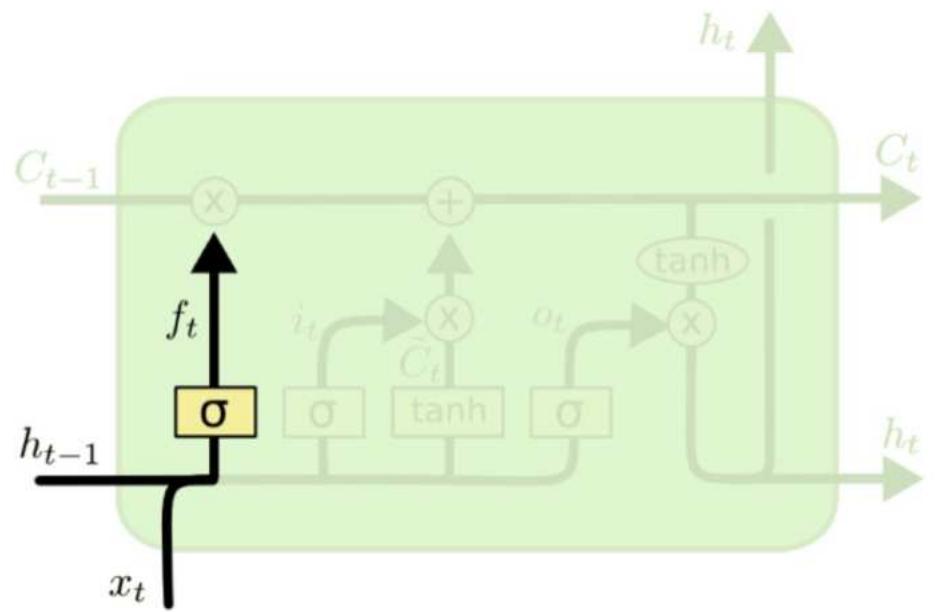
Forget gate

Example: Language modelling

Consider trying to predict the next word based on all previous ones

The cell state may include the gender of the present subject so that the proper pronouns can be used

When we see a new subject we want to forget old subject



Input gate

The input gate can allow incoming signal to **alter the state of the memory cell or block it**. It decides what new information to store in the cell stage.

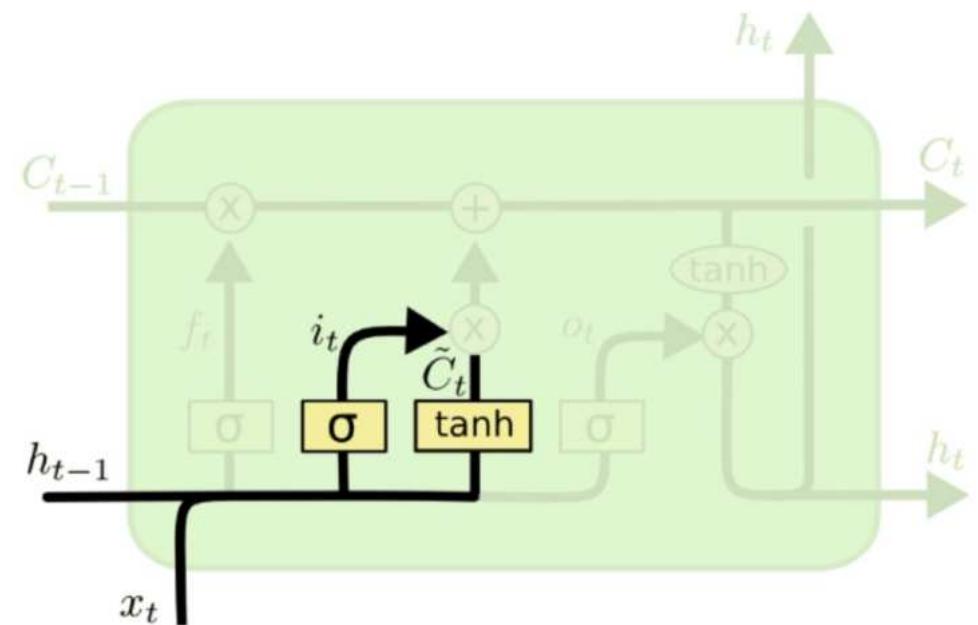
This has two parts:

A sigmoid input gate layer decides **which values to update**;

A tanh layer creates **a vector of new candidate values** $\tilde{c}(t)$ that could be added to the state.

$$i(t) = \sigma(U_i^T x(t) + W_i^T h(t-1) + b_i)$$

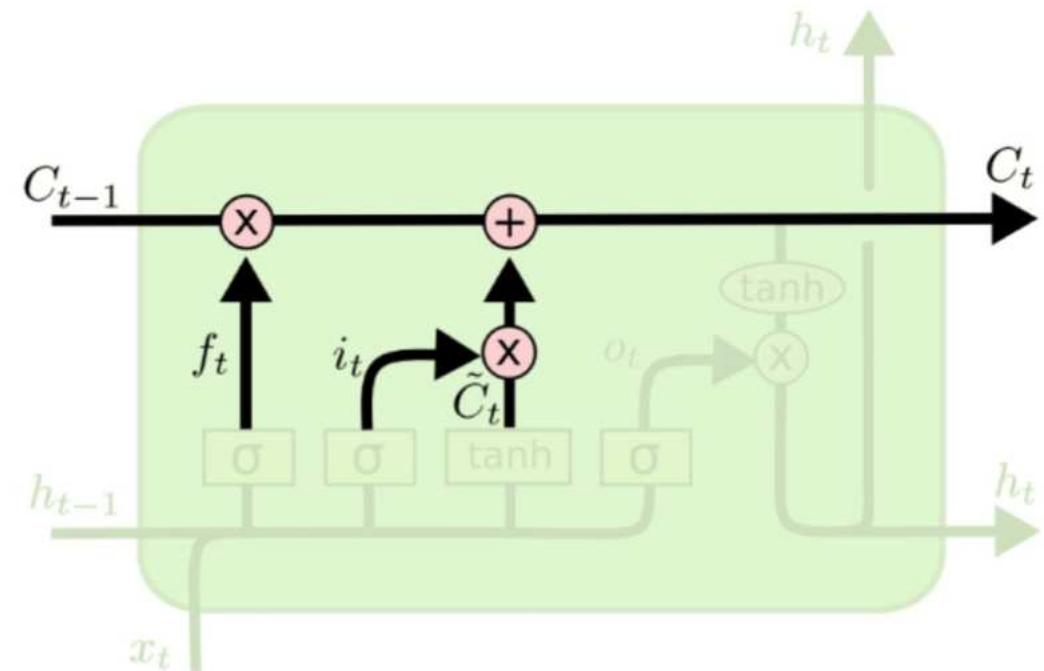
$$\tilde{c}(t) = \phi(U_c^T x(t) + W_c^T h(t-1) + b_c)$$



Example: Language modeling

We'd want to add the gender of the new subject to the cell state, to replace the old one we are forgetting

Cell state



Example: Language modeling

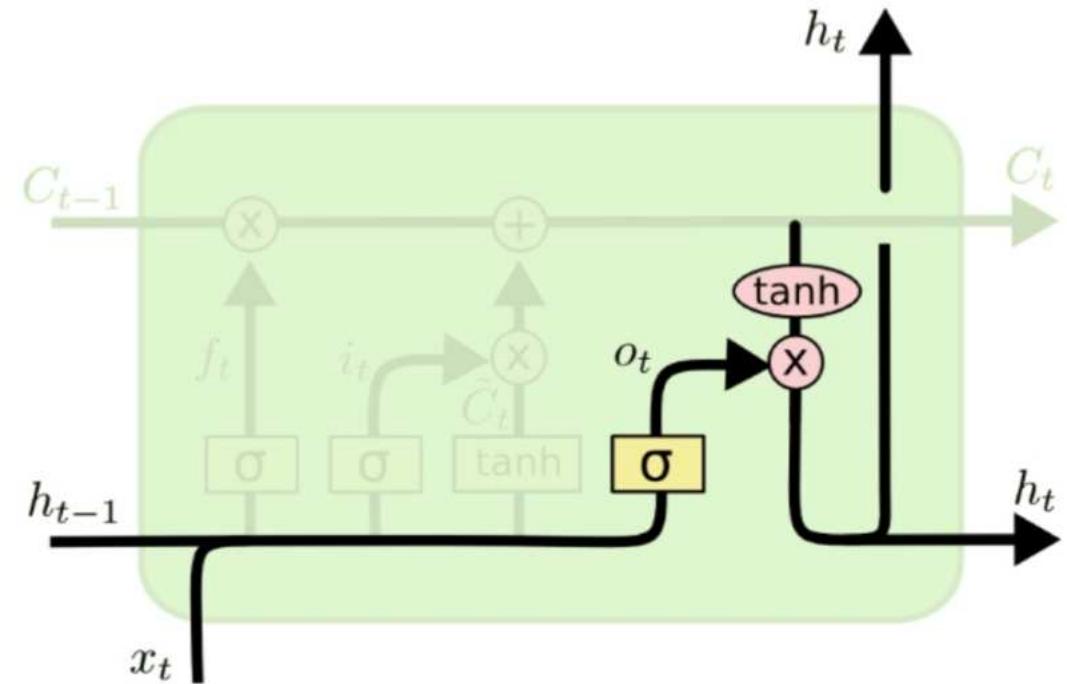
In the Language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in previous steps

Output gate

The output gate can allow the state of the memory cell to have an effect on other neurons or prevent it.

$$\mathbf{o}(t) = \sigma(\mathbf{U}_o^\top \mathbf{x}(t) + \mathbf{W}_o^\top \mathbf{h}(t-1) + \mathbf{b}_o)$$

$$\mathbf{h}(t) = \phi(\mathbf{c}(t)) \odot \mathbf{o}(t)$$



LSTM unit

$$i(t) = \sigma(U_i^\top x(t) + W_i^\top h(t-1) + b_i)$$

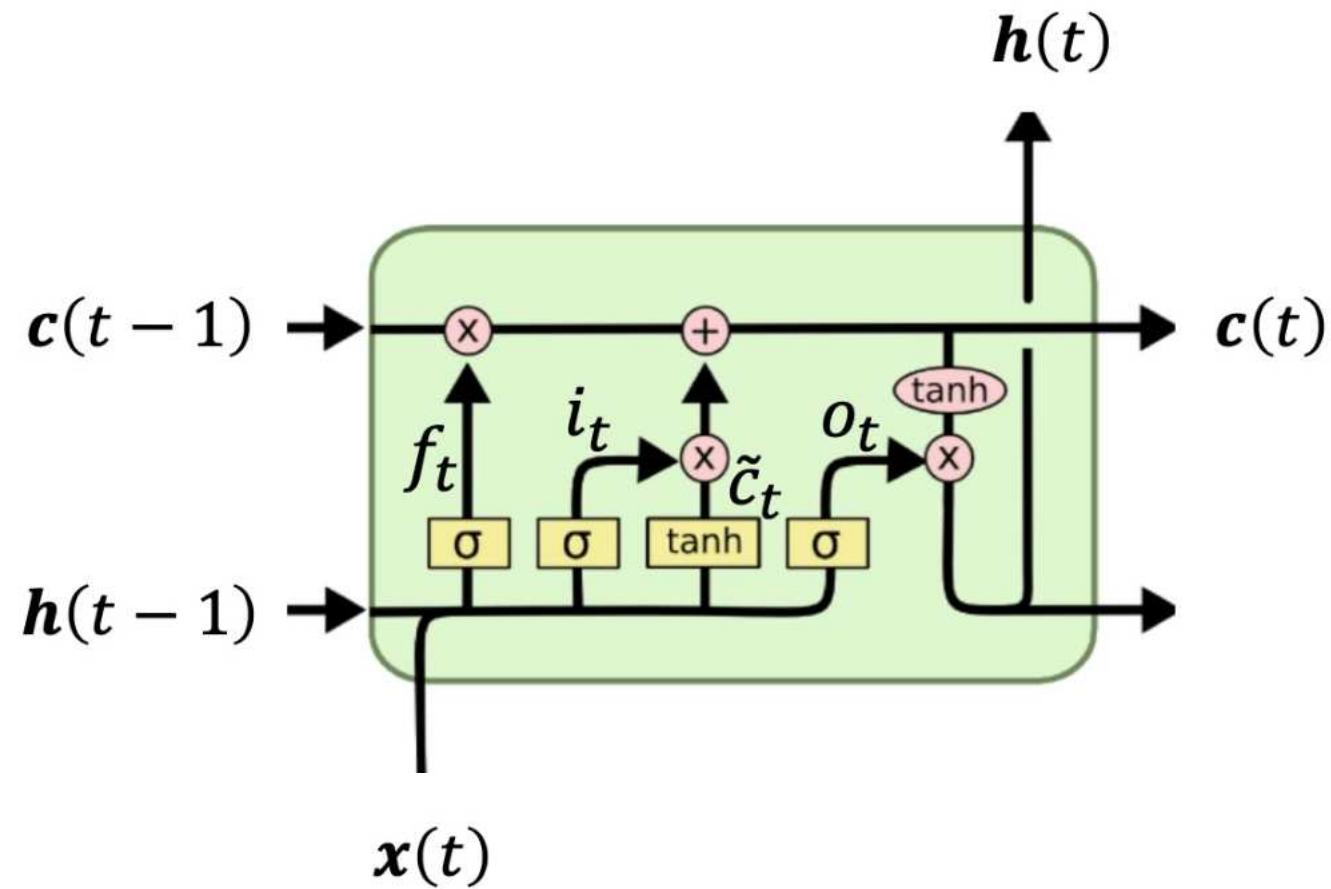
$$f(t) = \sigma(U_f^\top x(t) + W_f^\top h(t-1) + b_f)$$

$$o(t) = \sigma(U_o^\top x(t) + W_o^\top h(t-1) + b_o)$$

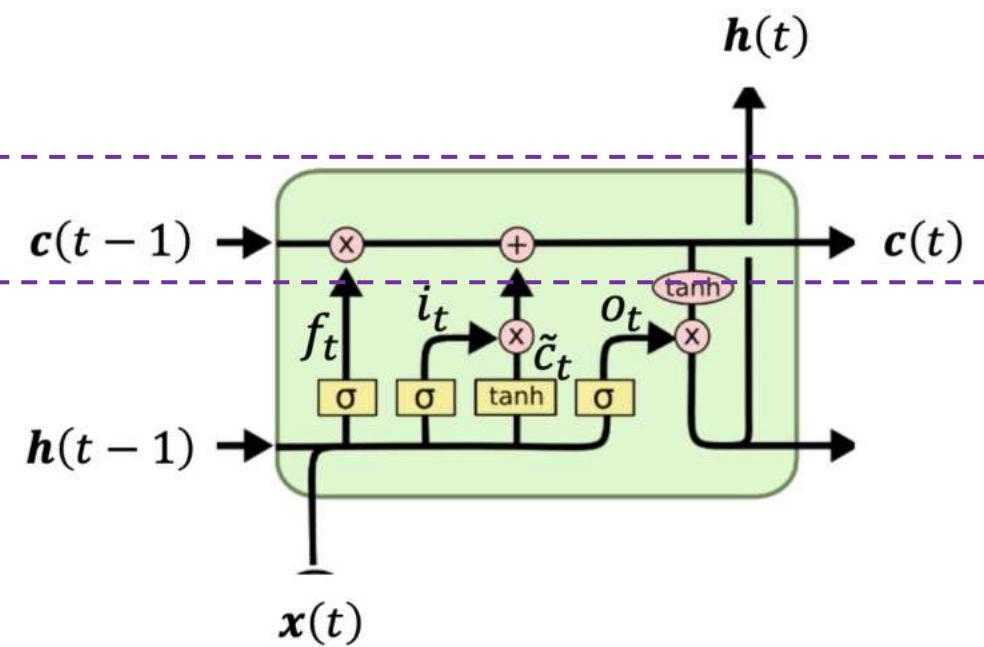
$$\tilde{c}(t) = \phi(U_c^\top x(t) + W_c^\top h(t-1) + b_c)$$

$$c(t) = \tilde{c}(t) \odot i(t) + c(t-1) \odot f(t)$$

$$h(t) = \phi(c(t)) \odot o(t)$$



LSTM unit

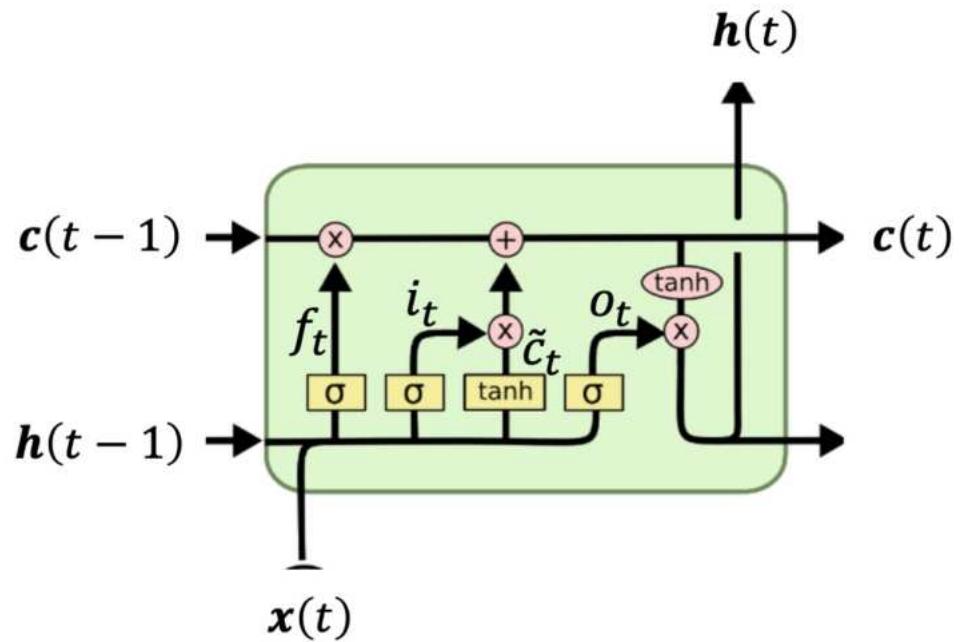


LSTM introduces a gated cell where the **information flow can be controlled**.

The most important component is the state unit $c_i(t)$ (for time step t and cell i) which has a **linear self-loop**.

The self-loop weight is controlled by a **forget gate** unit, which sets this weight to a value between 0 and 1 via a sigmoid unit.

LSTM unit



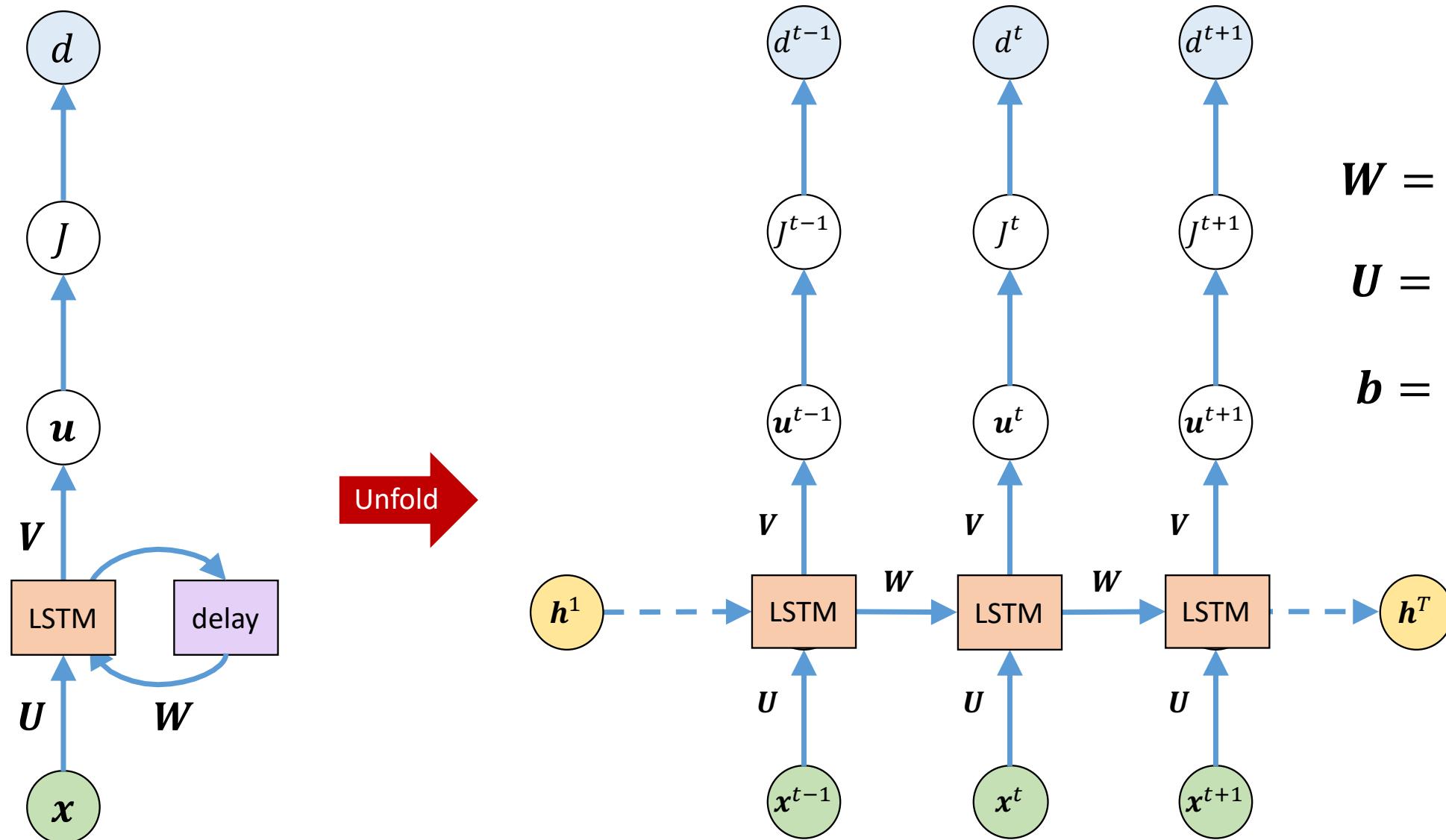
Clever idea!

The self-loop inside the cell is able to produce paths where **the gradient can flow for long duration** and to make the weight on this self-loop conditioned on the context rather than fixed.

By making the weight of this self-loop gated (controlled by another hidden unit), **the time scale of integration can be changed dynamically based on the input sequence**.

In this way, LSTM help **preserve error terms** that can be propagated through many layers and time steps.

Training LSTM networks



$$\mathbf{W} = \{W_i, W_f, W_o, W_c\}$$

$$\mathbf{U} = \{U_i, U_f, U_o, U_c\}$$

$$\mathbf{b} = \{b_i, b_f, b_o, b_c\}$$

Example 4

Generate 64 2-dimensional input sequences $(x(t))_{t=1}^{16}$ where $(x_1(t), x_2(t)) \in [0, 1]^2$ by randomly generating numbers uniformly.

Generate 1-dimensional output sequences $(y(t))_{t=1}^{16}$ where $y(t) \in \mathbf{R}$ by following the following recurrent relation:

$$y(t) = 5x_1(t-1)x_2(t-2) - 2x_1(t-7) + 3.5x_2^2(t-5) + 0.1\epsilon$$

where $\epsilon \sim N(0, 1)$.

Train a LSTM layer to learn the mapping between input and output sequences.

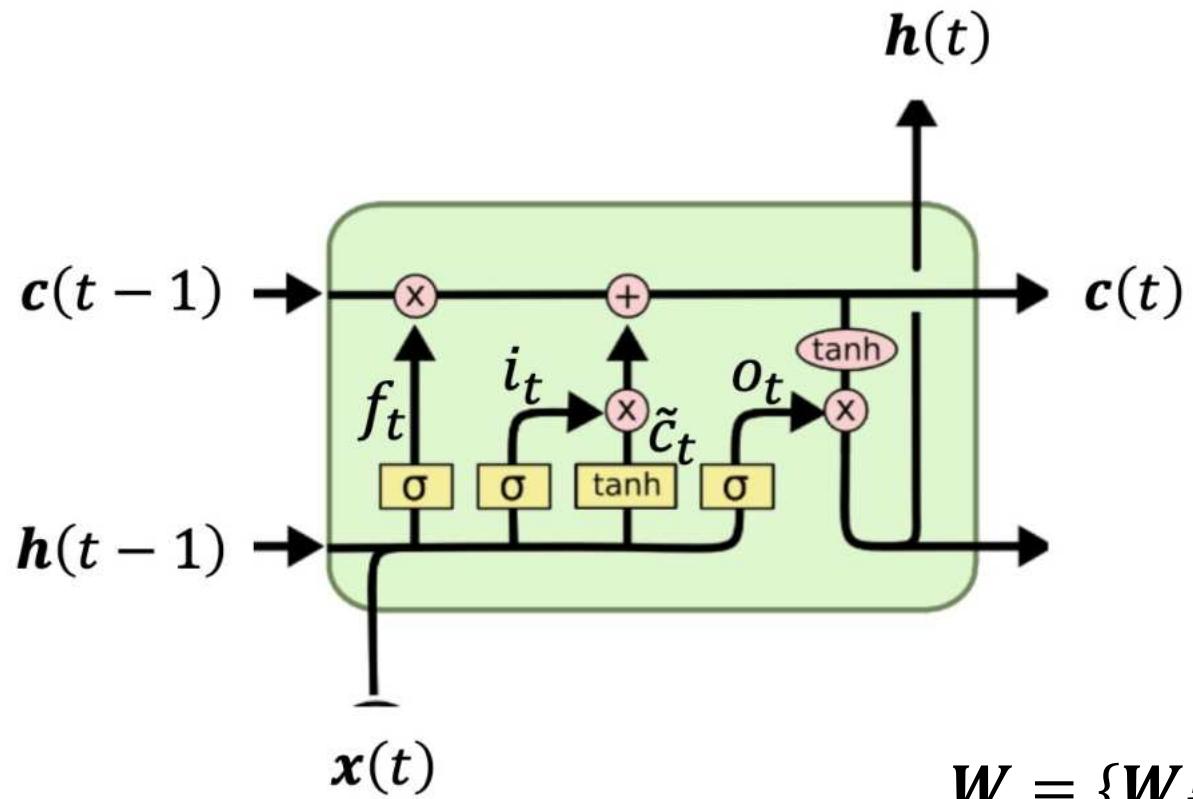
Use a learning factor $\alpha = 0.001$.

eg8.4a.ipynb

Repeat the above using RNN and GRU and compare the performances.

eg8.4b.ipynb

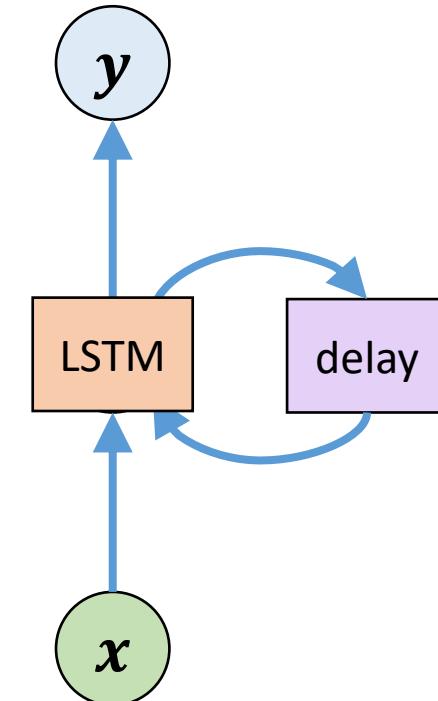
Example 4



$$\mathbf{W} = \{\mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_c\}$$

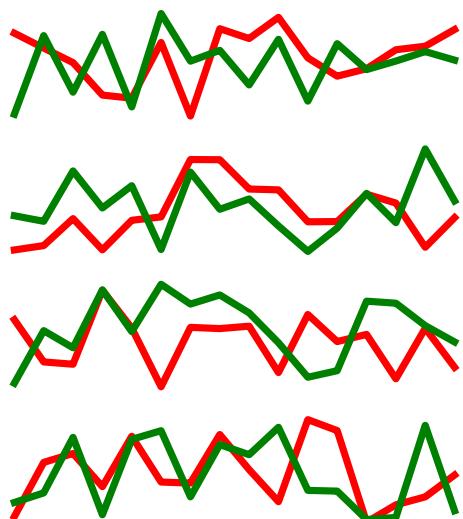
$$\mathbf{U} = \{\mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_o, \mathbf{U}_c\}$$

$$\mathbf{b} = \{\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o, \mathbf{b}_c\}$$

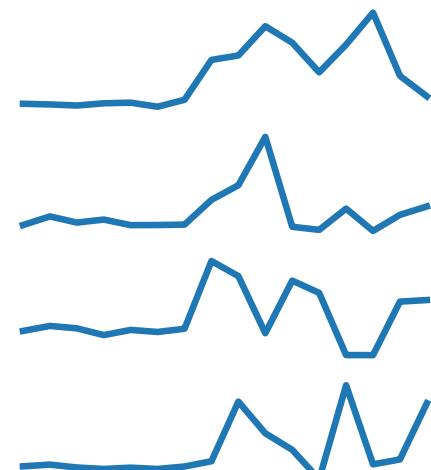
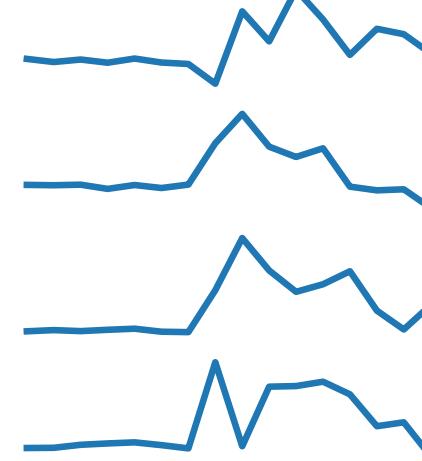
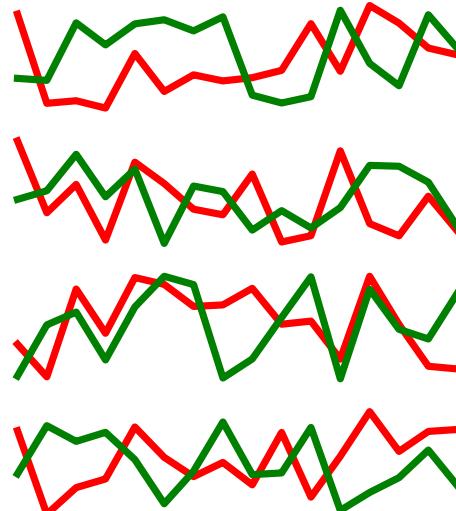


Example 4

$x_1(t)$ and $x_2(t)$



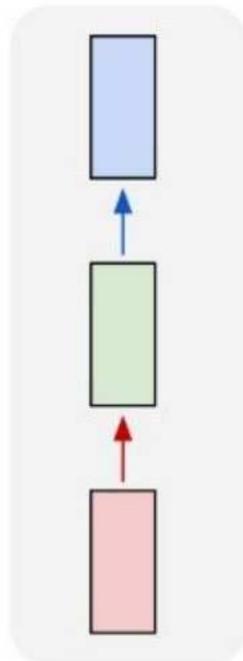
$$y(t) = 5x_1(t - 1)x_2(t - 1) - 2x_1(t - 7) + 3.5x_2^2(t - 5)$$



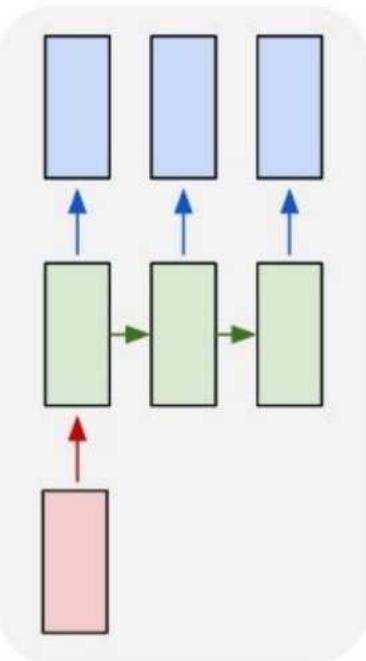
Example Applications

Input-output scenarios

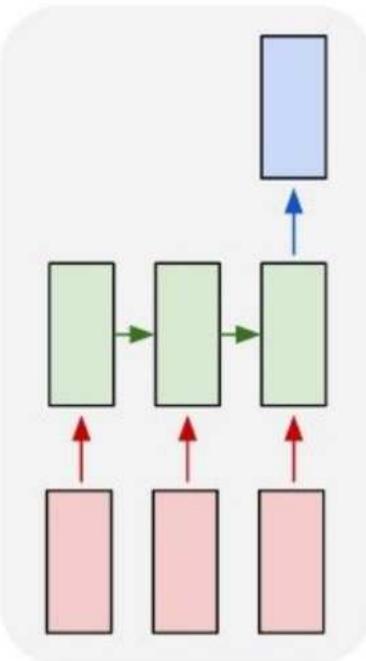
one to one



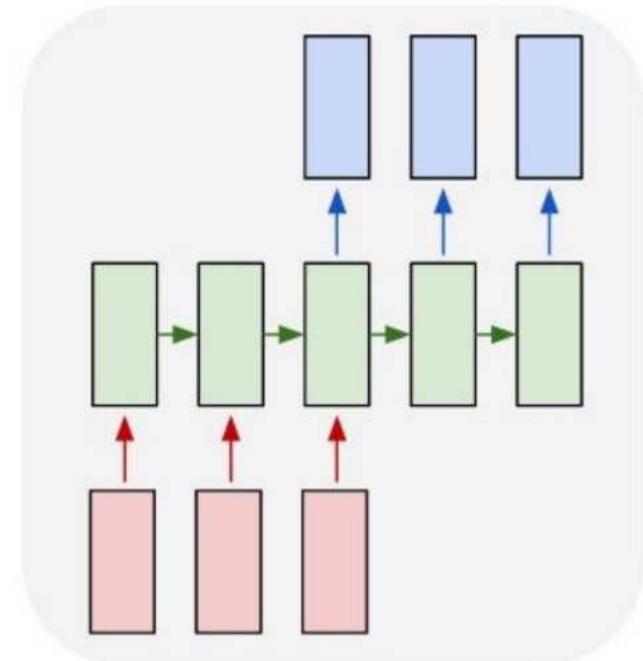
one to many



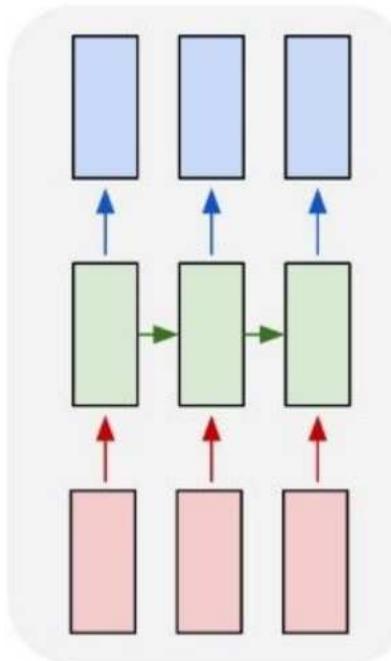
many to one



many to many



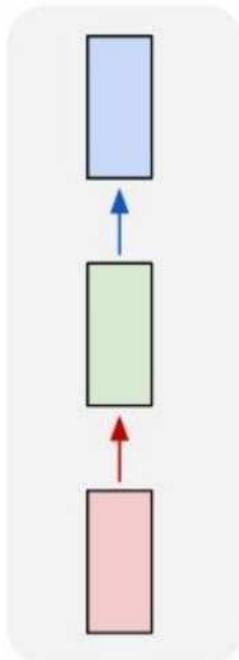
many to many



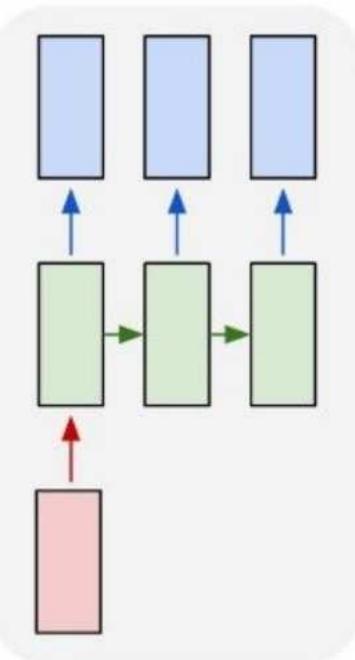
e.g. **Image Captioning**
image -> sequence of words

Input-output scenarios

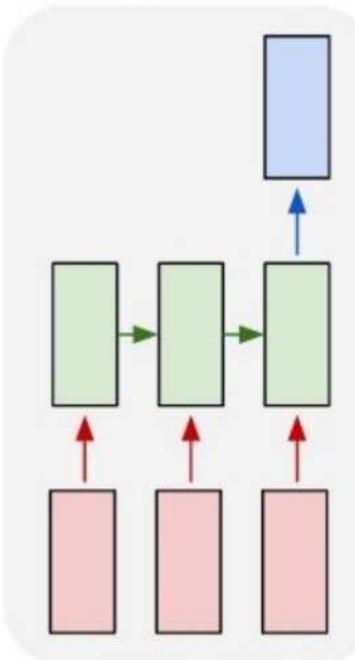
one to one



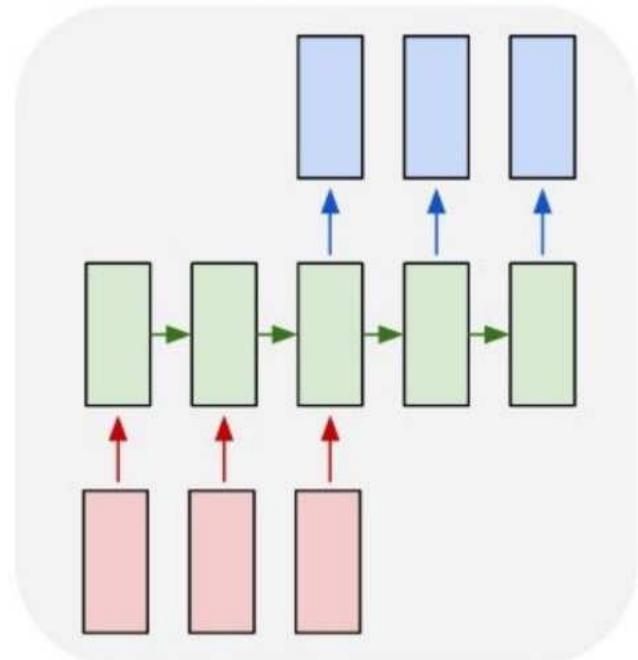
one to many



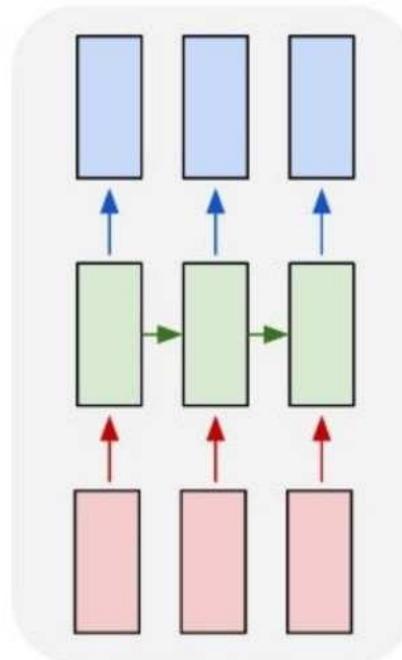
many to one



many to many



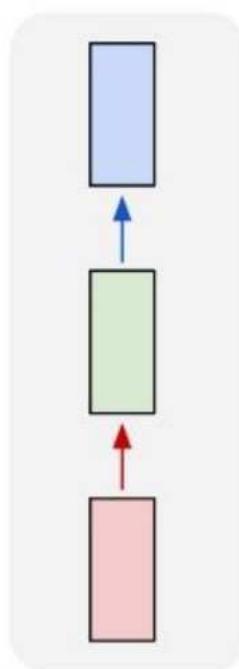
many to many



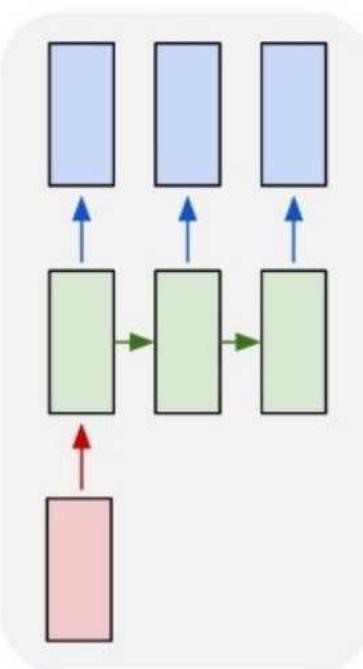
e.g. **Sentiment Classification**
sequence of words -> sentiment

Input-output scenarios

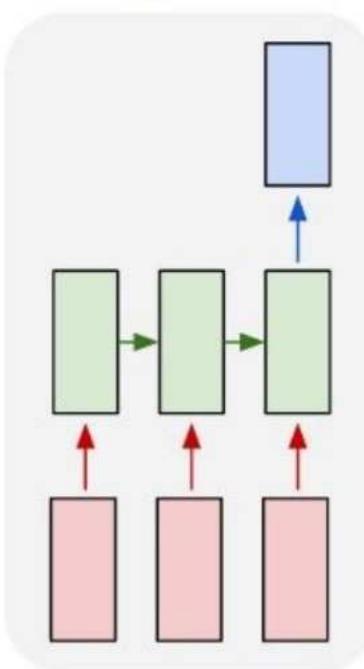
one to one



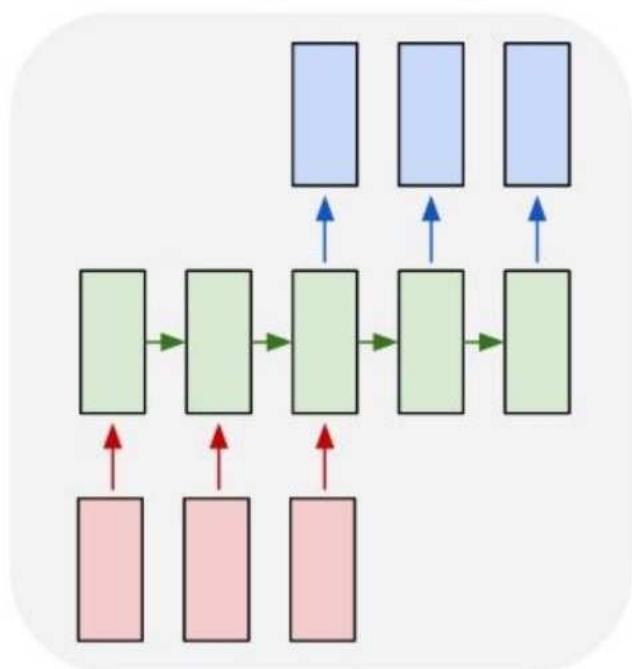
one to many



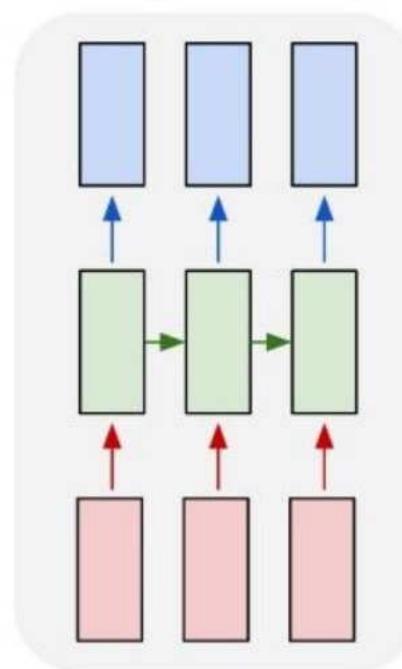
many to one



many to many



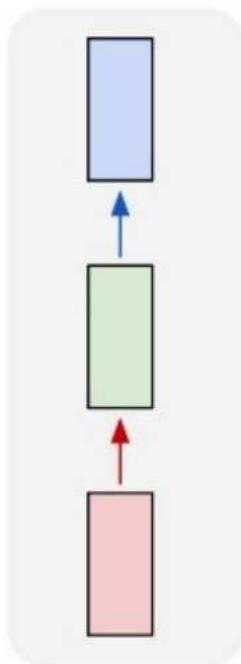
many to many



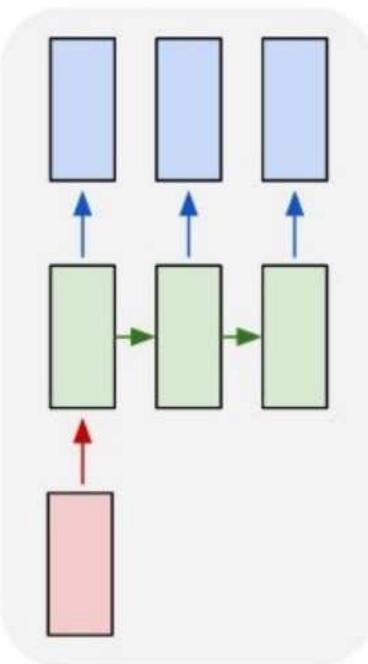
e.g. **Machine Translation**
seq of words -> seq of words

Input-output scenarios

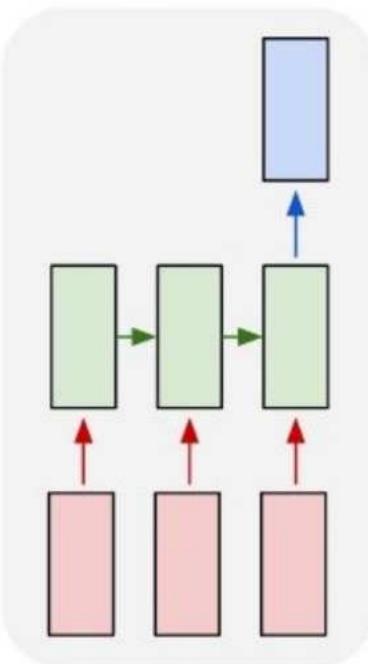
one to one



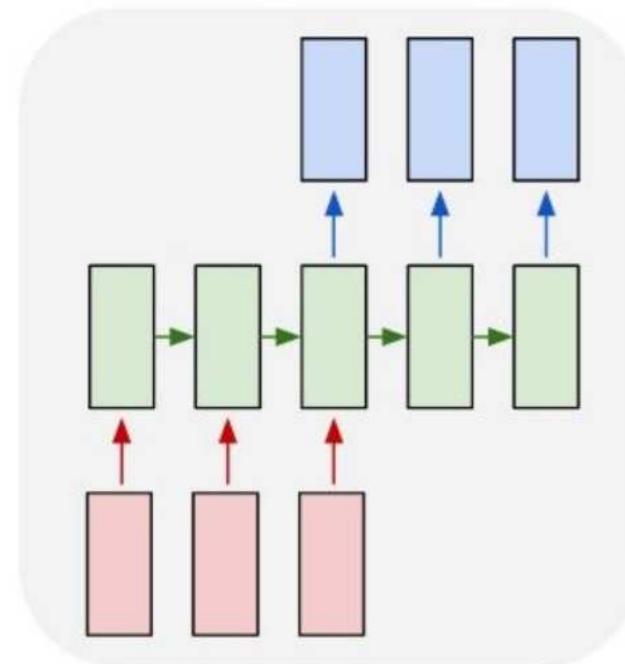
one to many



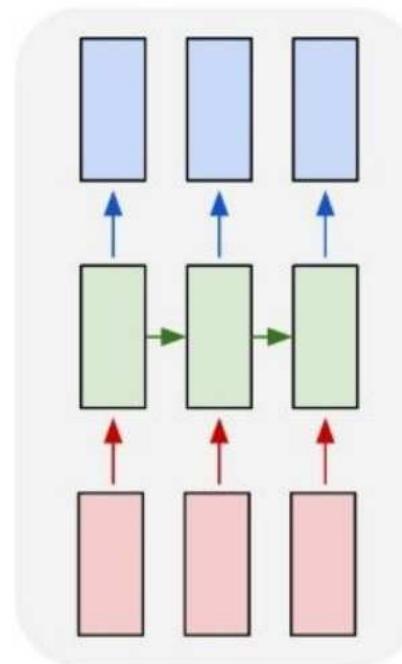
many to one



many to many



many to many



e.g. Video classification on frame level

Text classification: Dbpedia dataset

This dataset contains first paragraph of Wikipedia page of 56000 entities and label them as one of 15 categories like people, company, schools, etc.

• <small>DBpedia Resources</small>	
1	D. C. Thomson & Co.
1	Pyro Spectaculars
1	Admiral Insurance
1	Pax Softnica
1	The ACME Laboratories Ltd
2	Dubai Gem Private School & Nursery
2	Michael Wallace Elementary School
2	Alma Heights Christian Schools

DBpedia Resources was a research project in New York City which was founded in 2007 by Jennifer Heer

D. C. Thomson & Company Limited is a Scottish publishing company based in Dundee best known for
Pyro Spectaculars is headquartered in Rialto California USA and occupies a portion of a former World War II era aircraft factory
Admiral a trading name of EUI Limited is a car insurance specialist which launched in January 1993. Its
Pax Softnica (パックスソフトニカ) is a Japanese video game developer founded in 1983 under the name
The ACME Laboratories Ltd is a major pharmaceutical company based in Bangladesh. It is part of the ACME Group of companies
Dubai Gem Private School (DGPS) is a British school located in the Oud Metha area of Dubai United Arab Emirates
Michael Wallace Elementary School is a Canadian public school in Dartmouth Nova Scotia. It is operated by the Halifax Regional School Board
Alma Heights Christian Schools (AHC) formerly Alma Heights Christian Academy is a private elementary school in Mississauga Ontario Canada

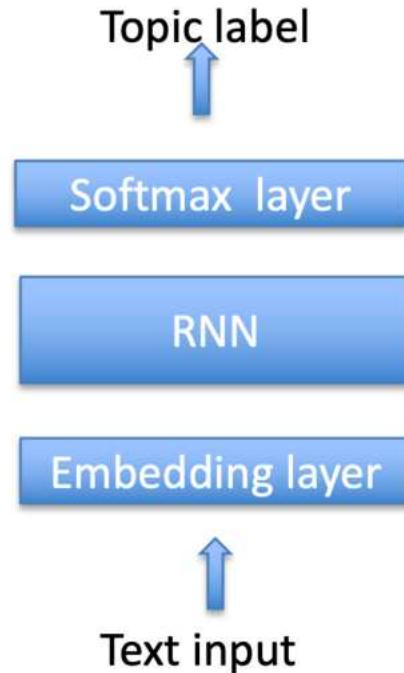
An input is a text

Requires to find all words in the text and remap them into word IDs, a number per each unit word (word-level inference)

my work is cool! → [23, 500, 5, 1402, 17]

We need to make sure that each sentence is of same length (`no_words`). The maximum document length is fixed and longer sentences will be truncated and shorter ones are padded with zeros.

Text classification



$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \end{bmatrix} [1] = [w_{11}] \\ \begin{bmatrix} w_{21} & w_{22} & w_{23} & w_{14} \end{bmatrix} [0] = [w_{21}] \\ \begin{bmatrix} w_{31} & w_{32} & w_{33} & w_{14} \end{bmatrix} [0] = [w_{31}] \\ [0] \end{bmatrix}$$

Matrix multiply with a one-hot vector just extracts a column from the weight matrix.

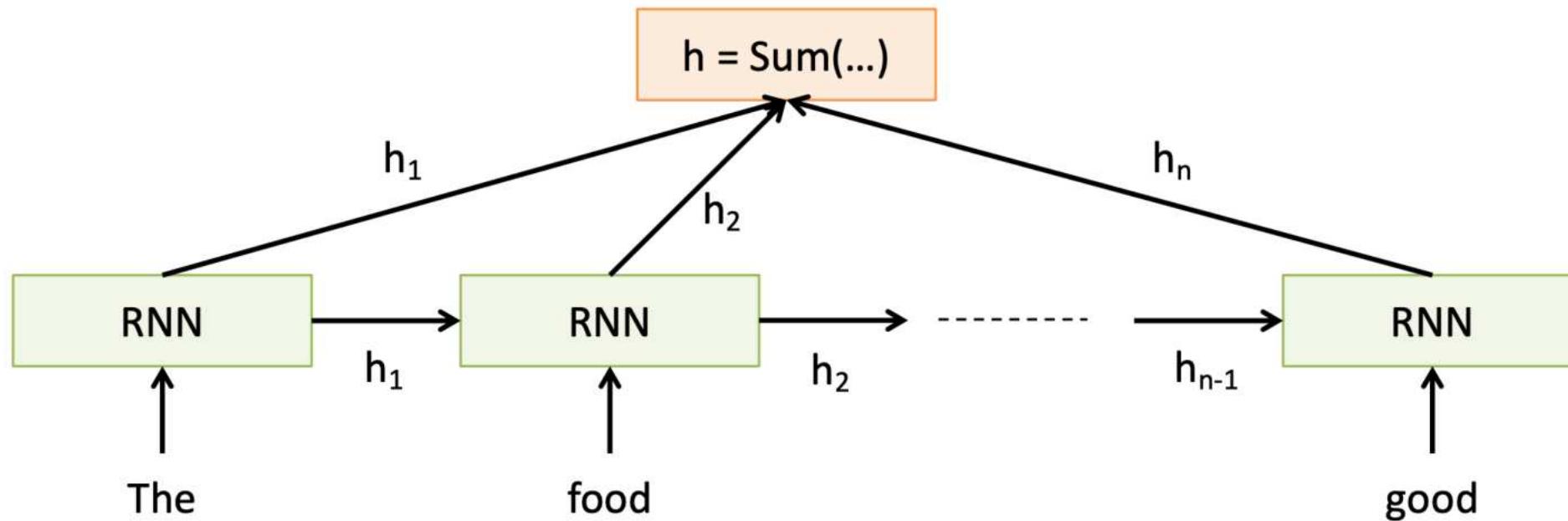
The embedding layer converts one-hot vector of word representations (of size of the vocabulary) to a vector of fixed length (embedding_size) vectors.

Embedding layer learns a weight matrix of [embedding size, vocab size]
And then maps word indexes of the sequences into
[batch_size, sequence_length, embedding_size] input to the RNN.

Sentiment classification

- “The food was really good”
“The chicken crossed the road because it was uncooked”
- Classify a
restaurant review from Yelp! OR
movie review from IMDB OR
...
as positive or negative
- Inputs: Multiple words, one or more sentences
- Outputs: Positive / Negative classification

Sentiment classification



Sentiment classification

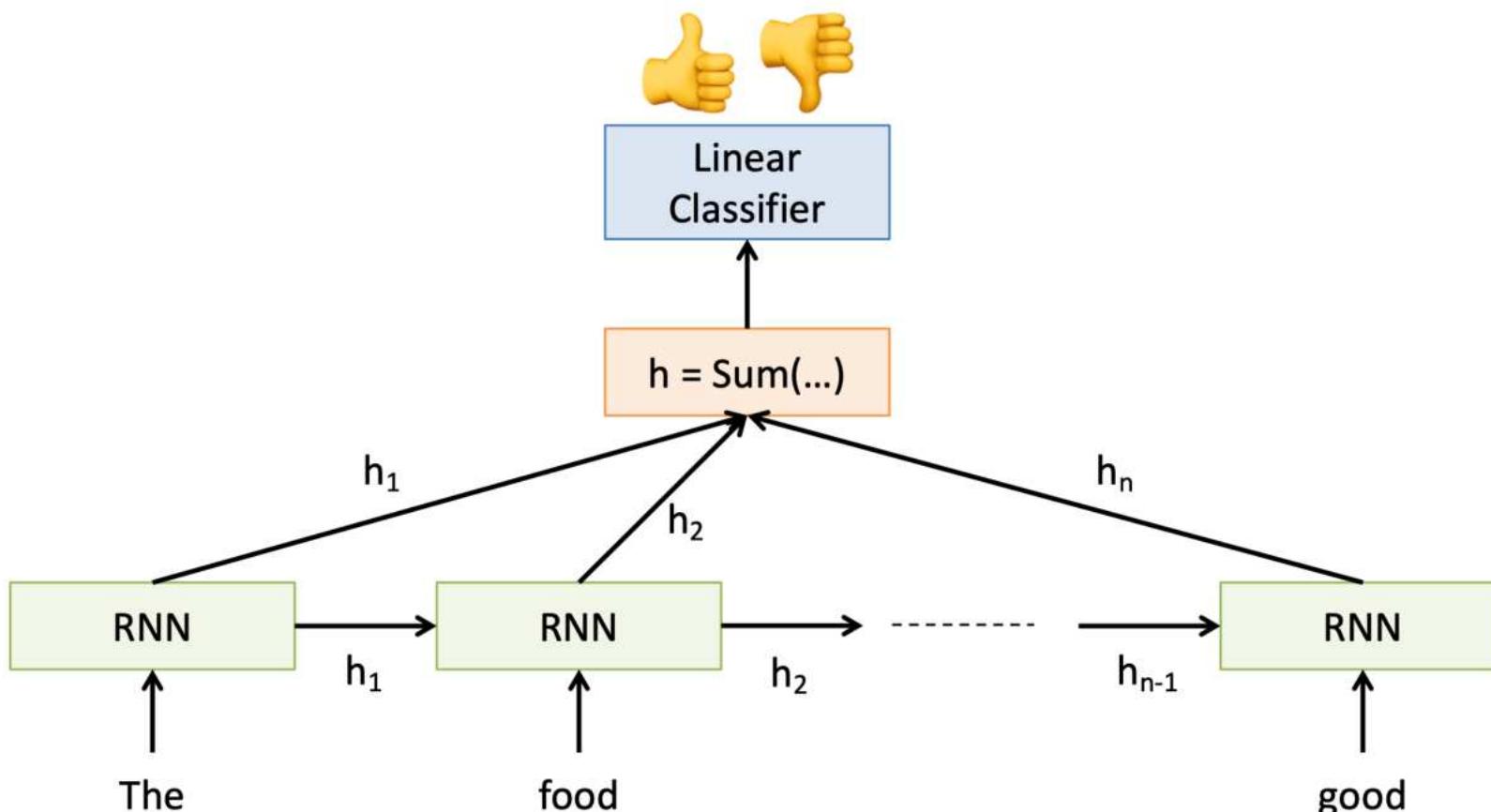
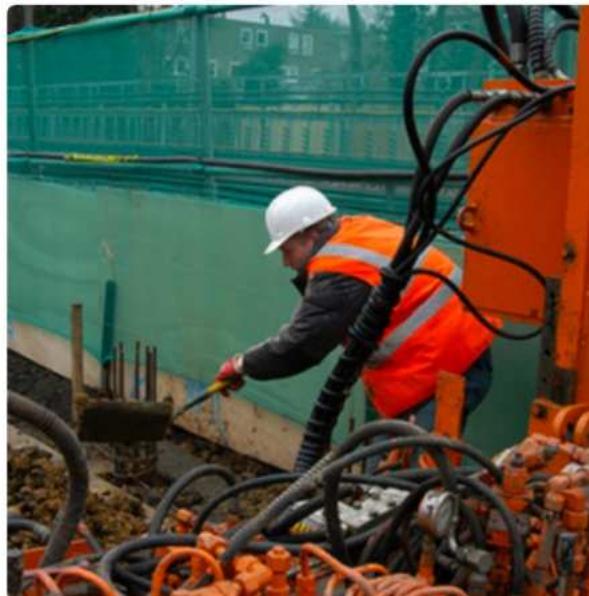


Image captioning

- Given an image, produce a sentence describing its contents
- Inputs: Image feature (from a CNN)
- Outputs: Multiple words (let's consider one sentence)



"man in black shirt is playing
guitar."



"construction worker in orange
safety vest is working on road."



"two young girls are playing with
lego toy."

Image captioning

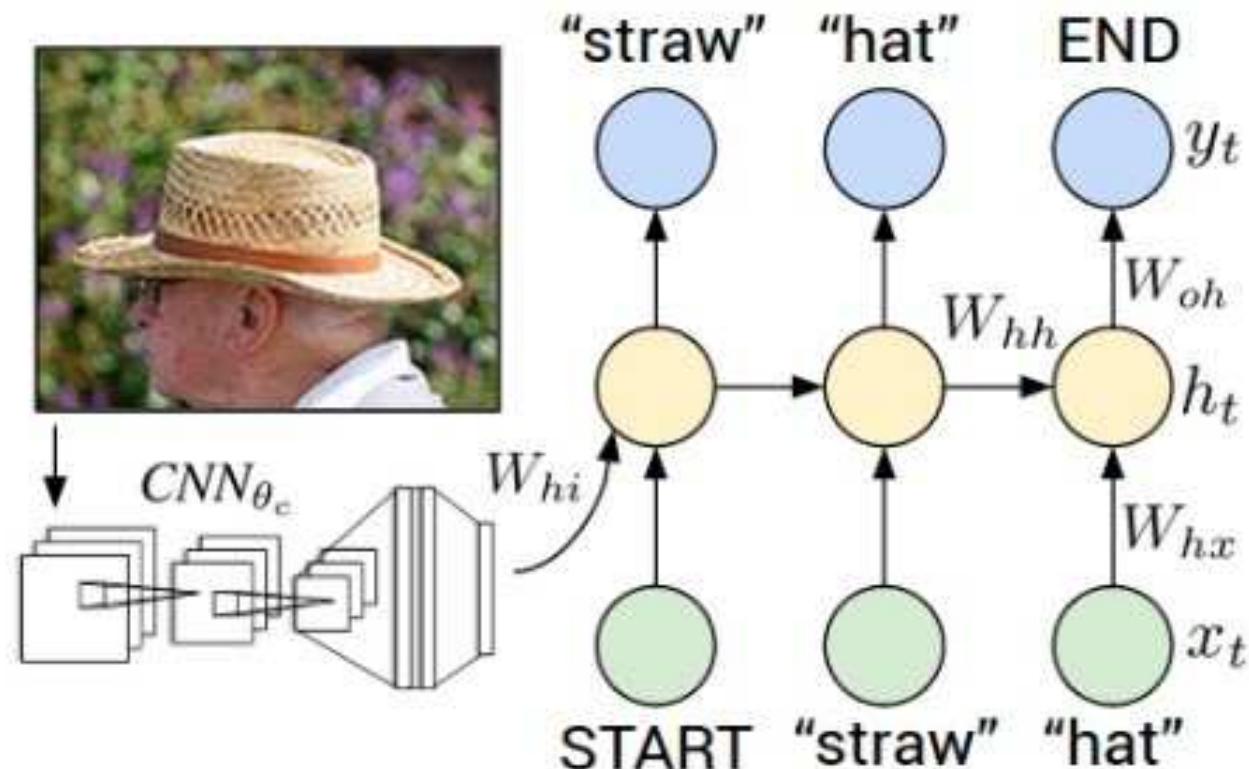


Image captioning

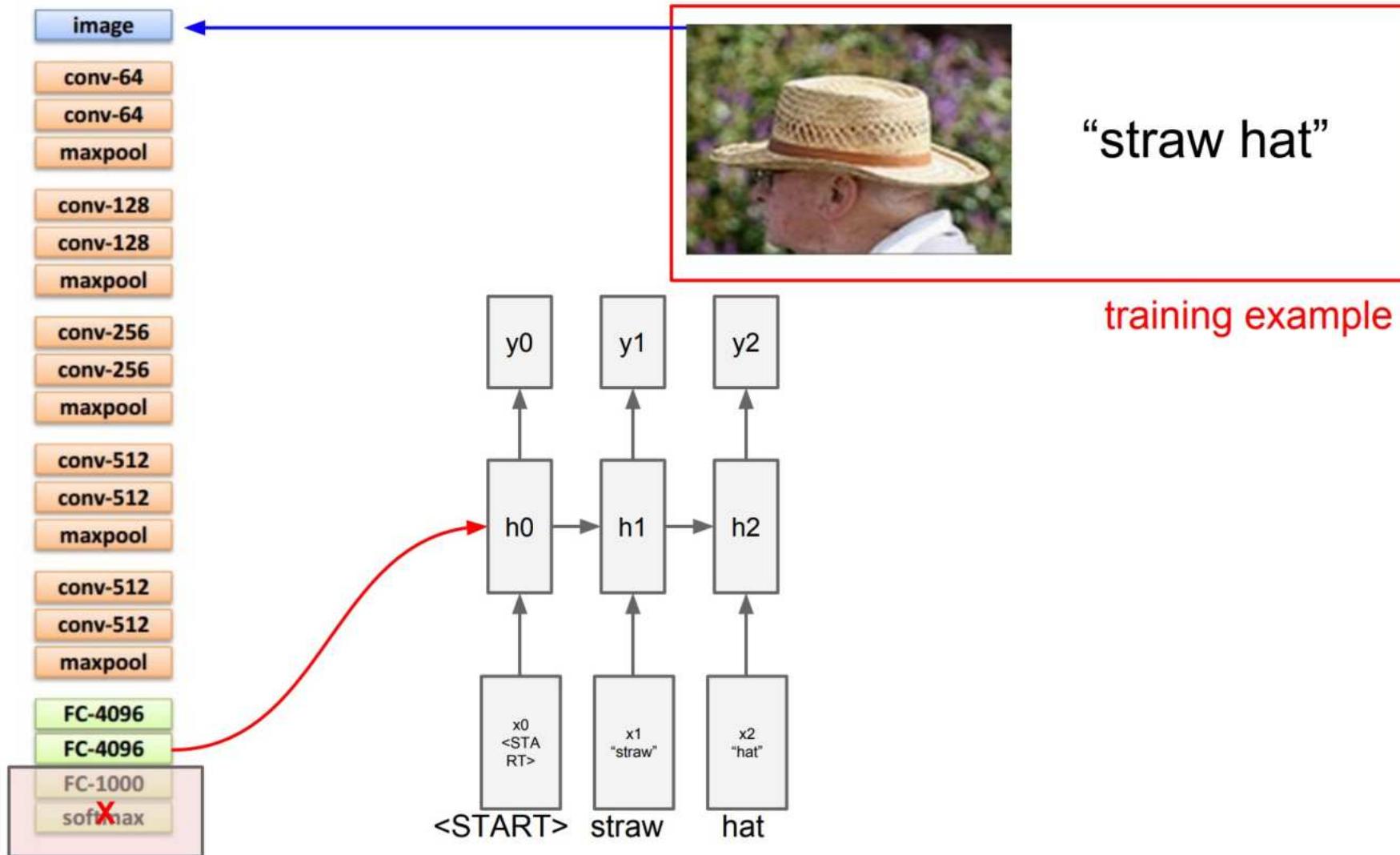


Image captioning

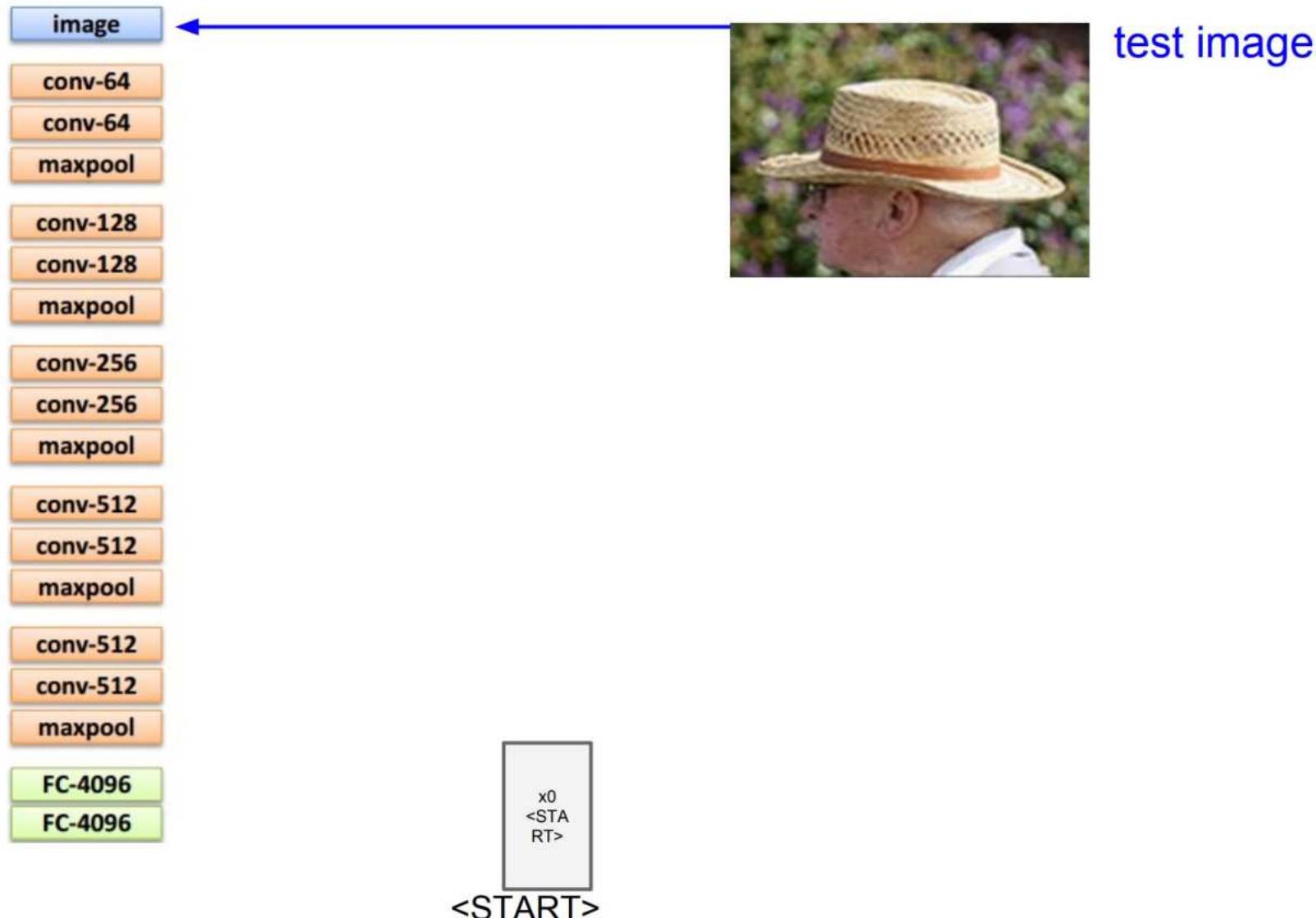


Image captioning

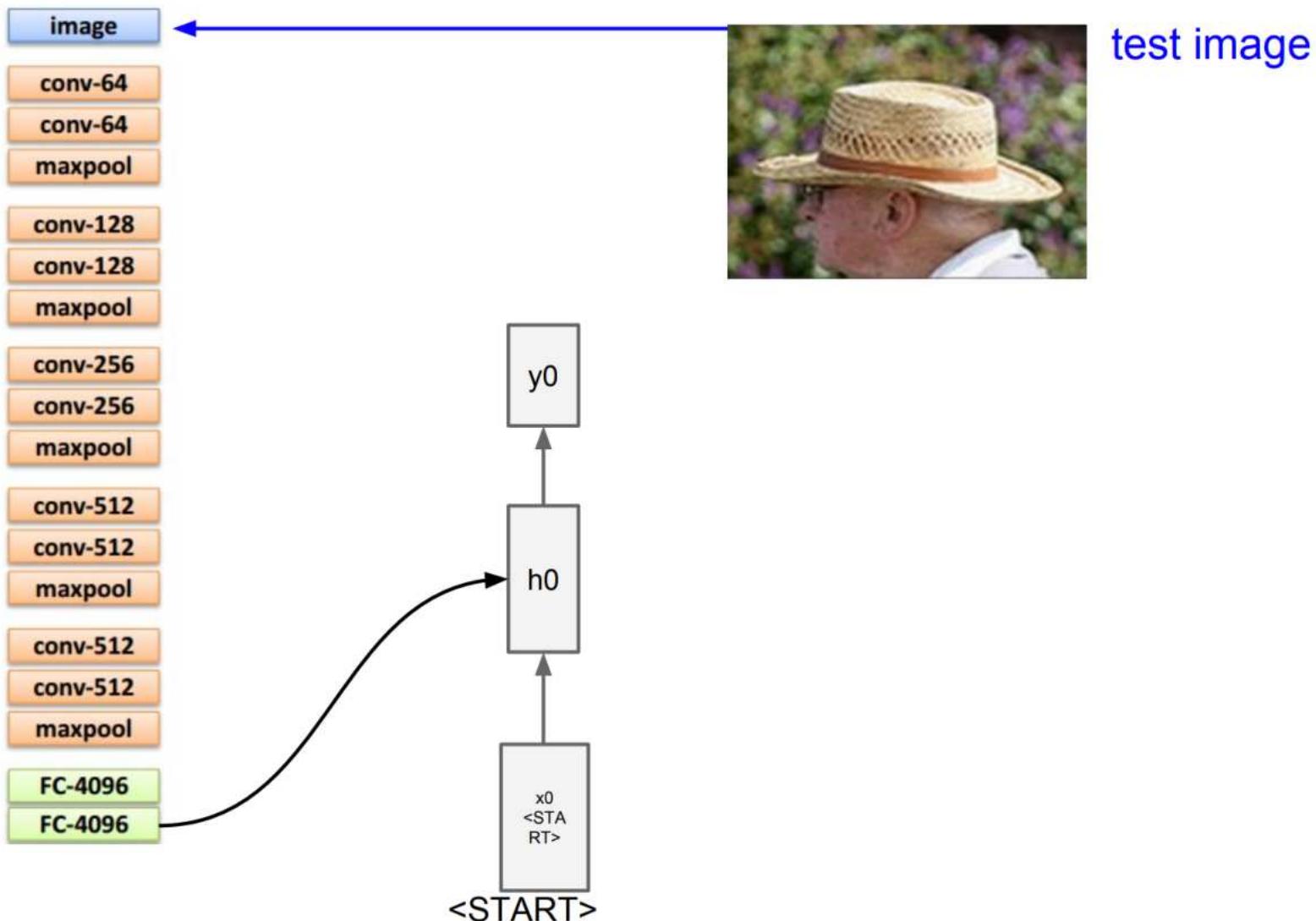


Image captioning

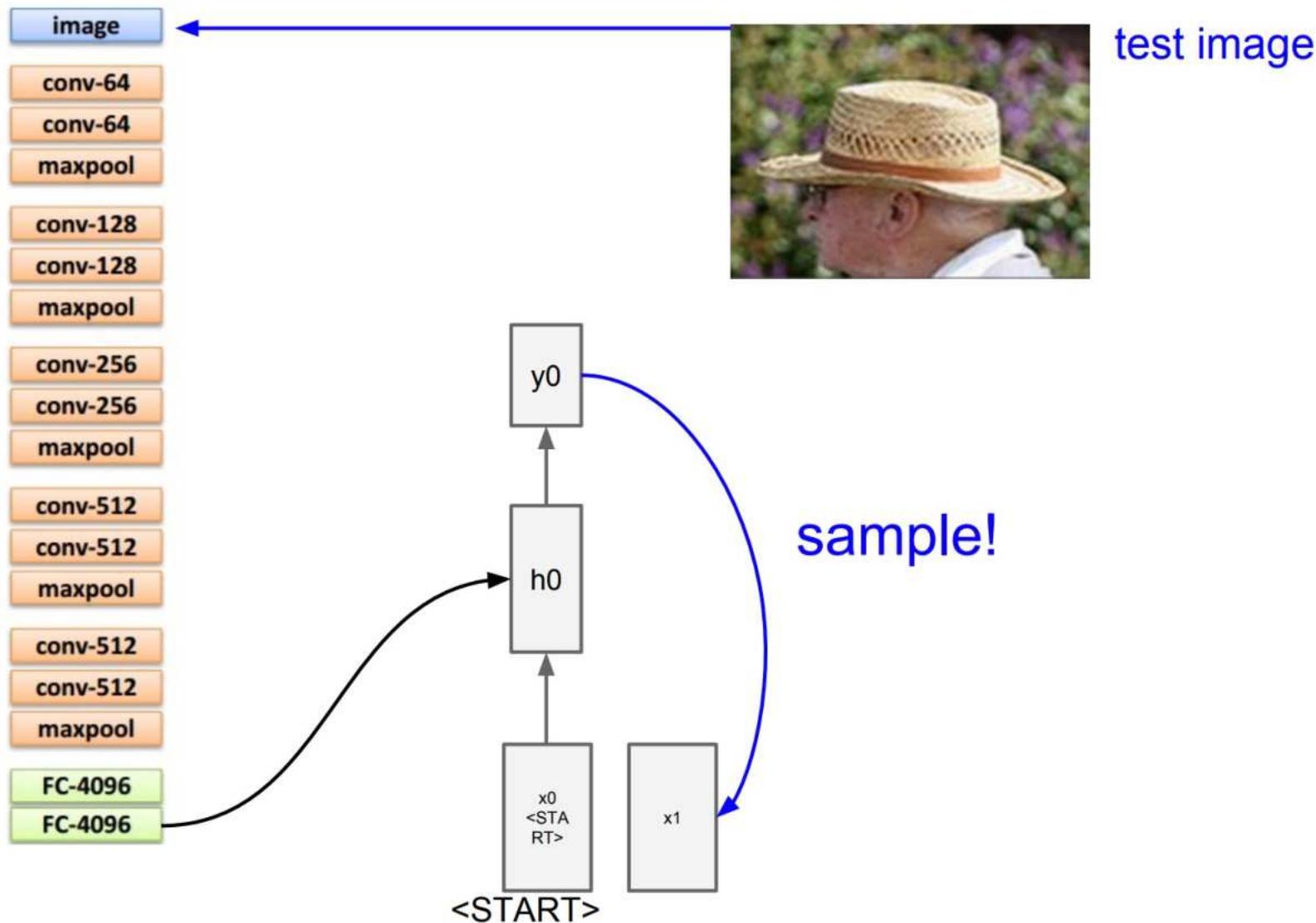


Image captioning

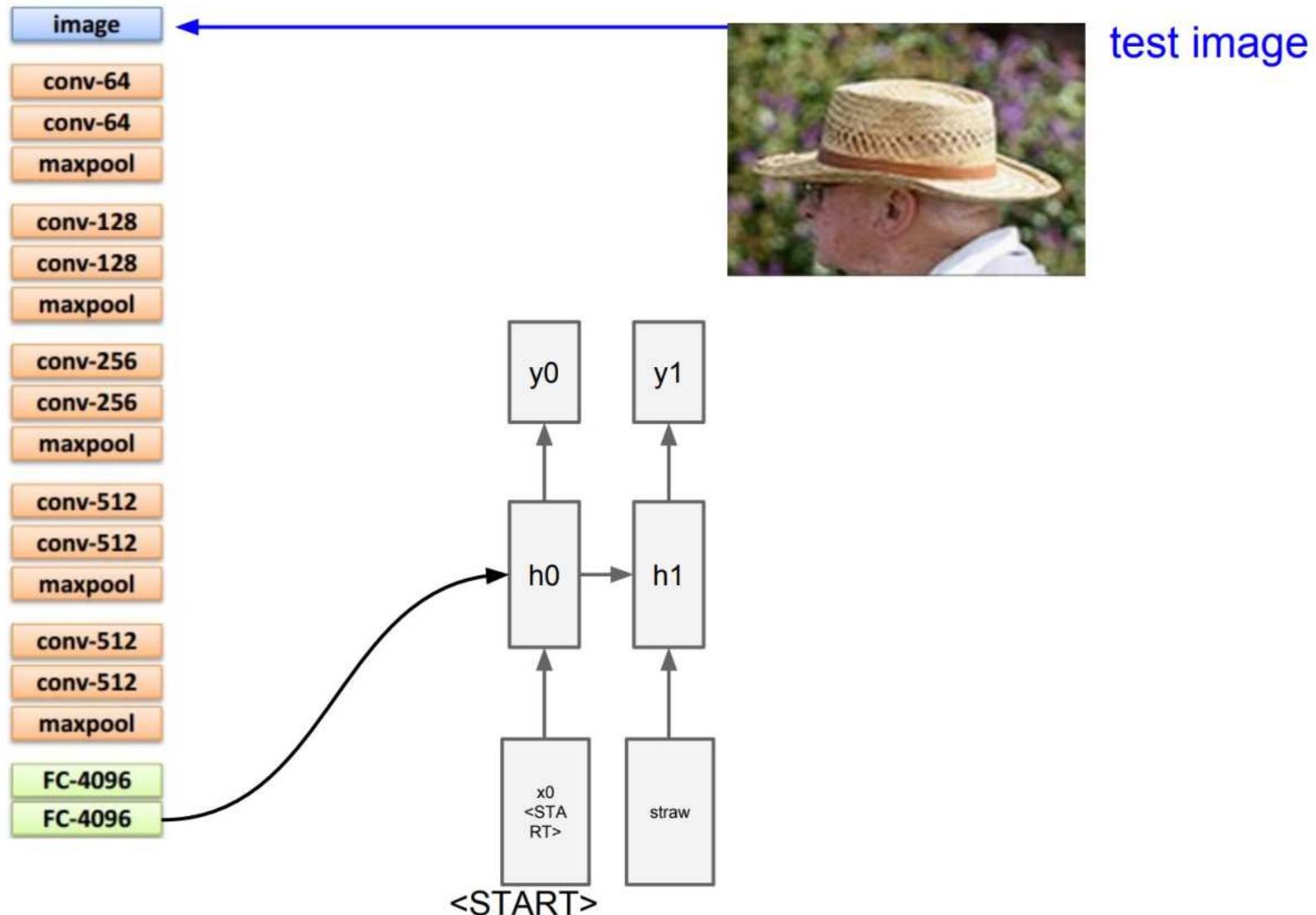


Image captioning

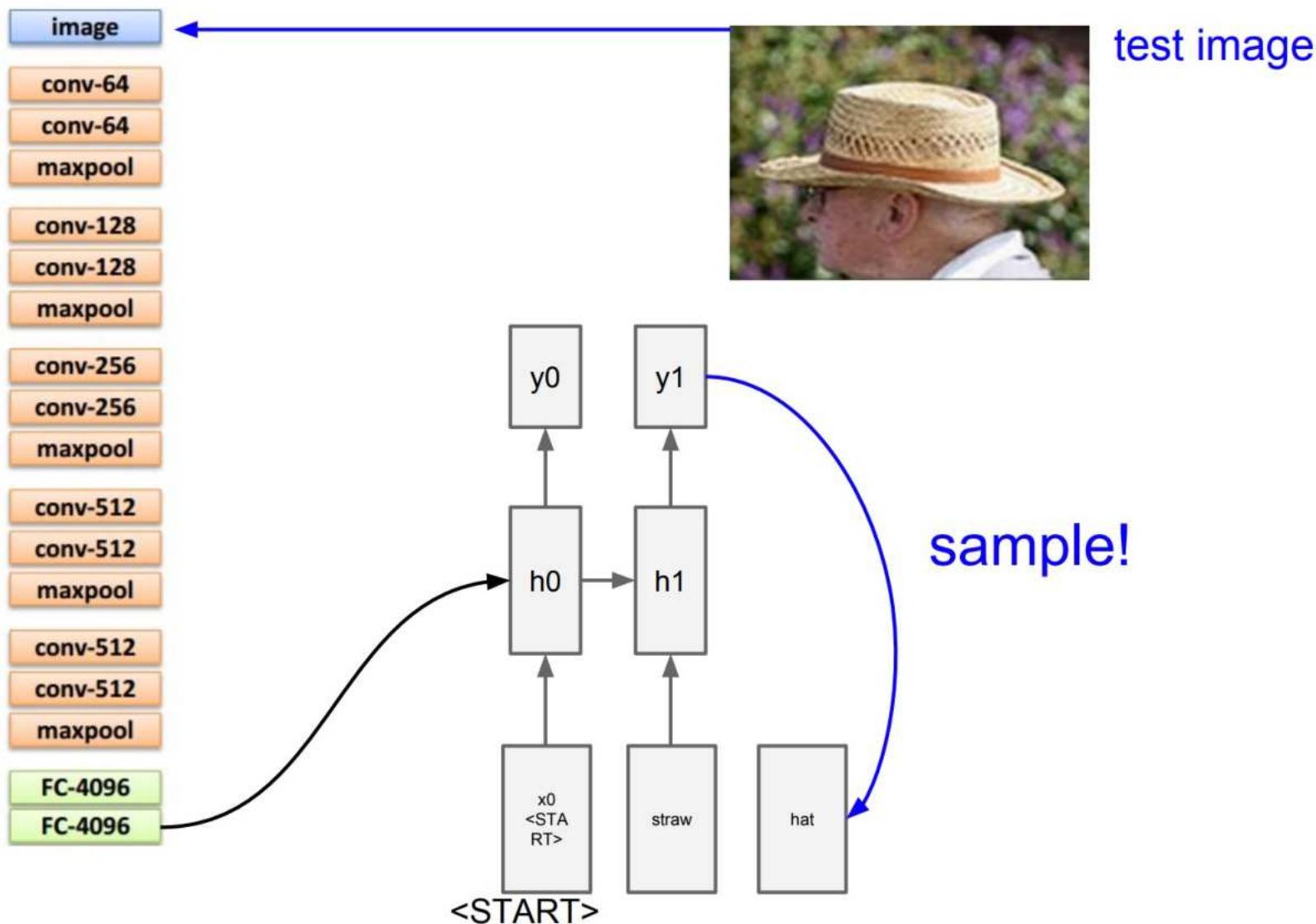


Image captioning

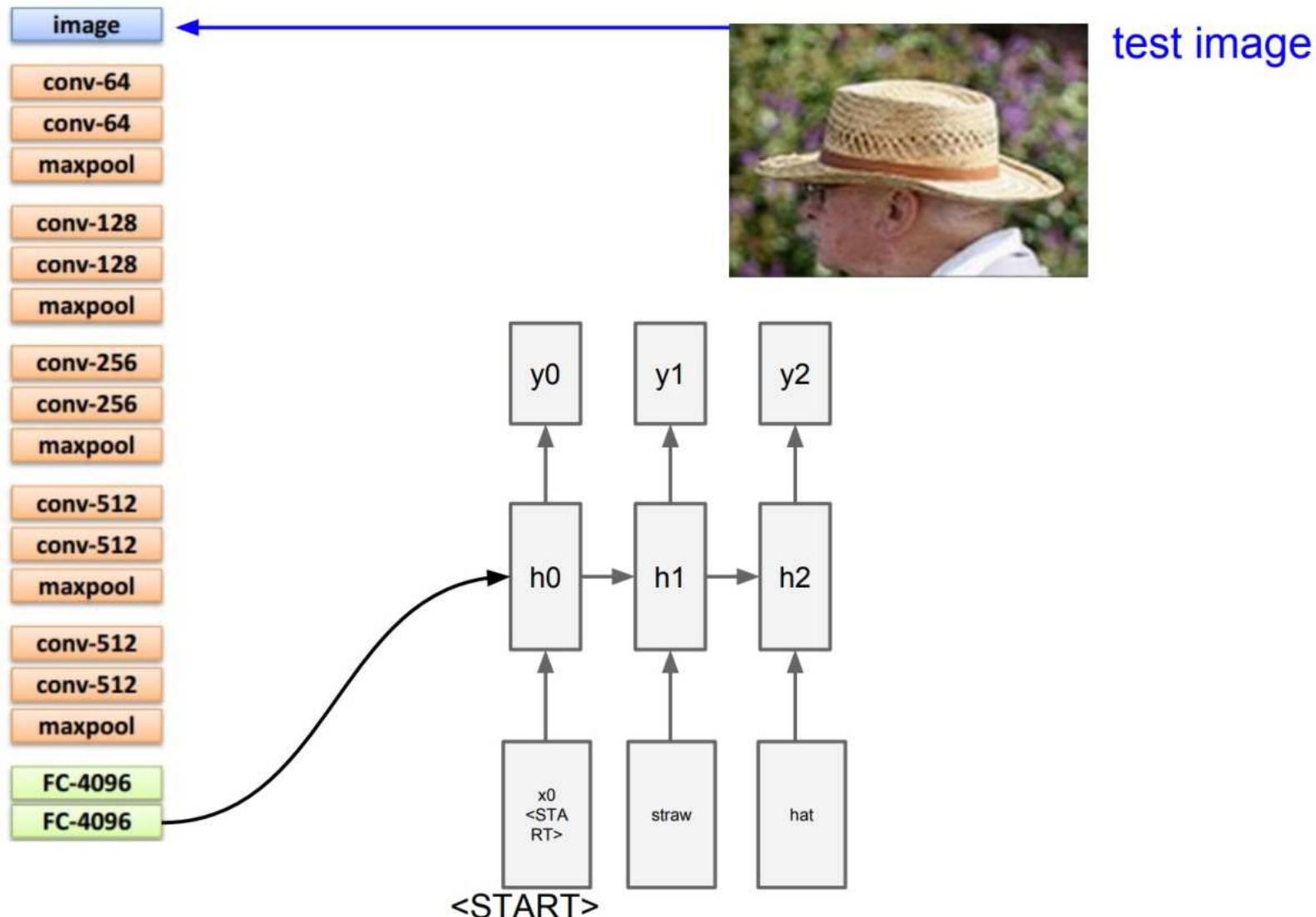


Image captioning

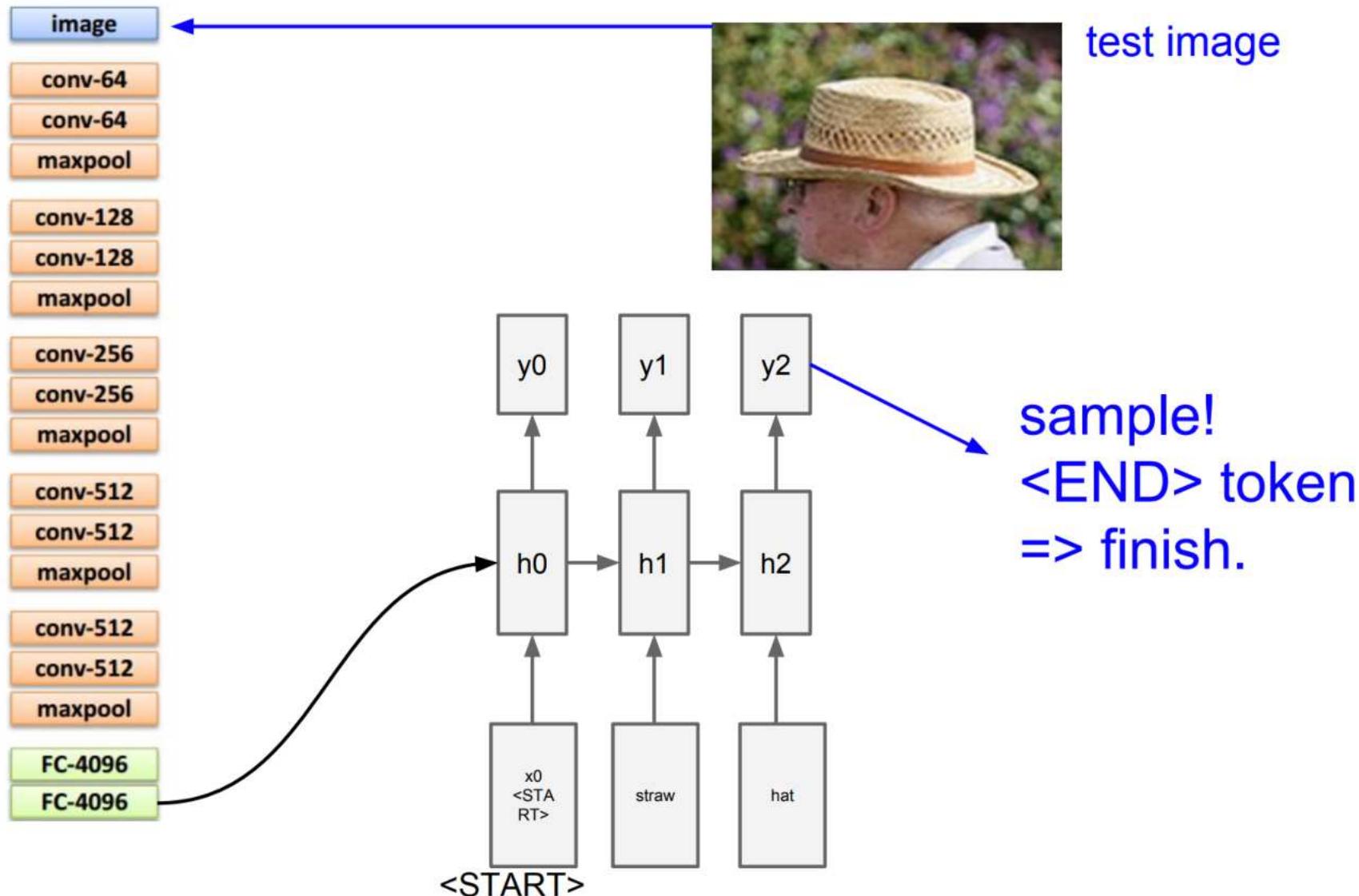


Image captioning



a young boy is holding a
baseball bat
logprob: -7.65



a baby laying on a bed with a stuffed bear
logprob: -8.66

Image Credits

- <https://cs.stanford.edu/people/karpathy/sfmltalk.pdf>
- <https://developer.ibm.com/articles/cc-cognitive-recurrent-neural-networks/>

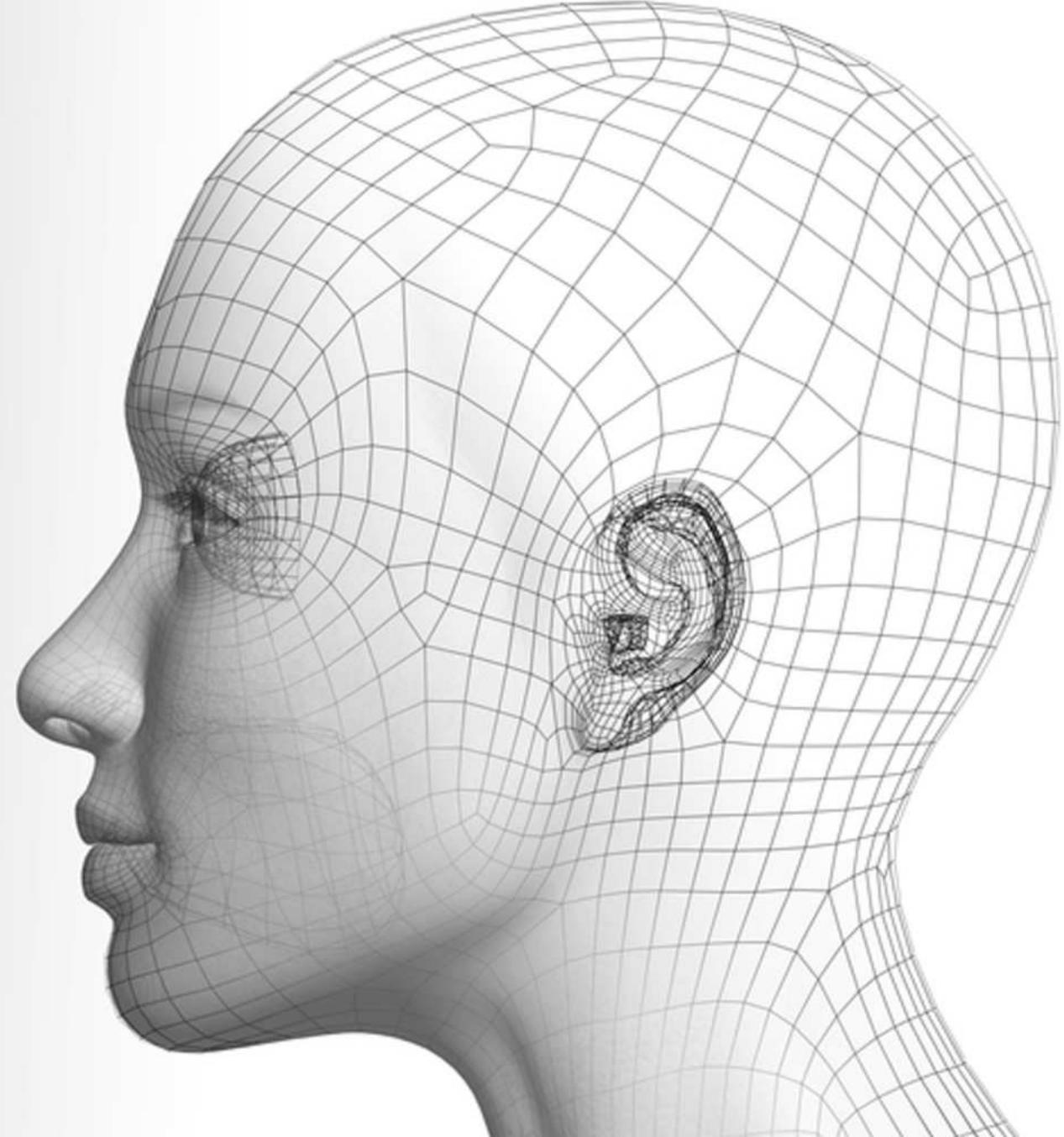
Attention

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

<https://twitter.com/XingangP>



Outline

- Attention
- Transformers
- Vision Transformers

Attention



"Where's Waldo?"

Attention

Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified by learned weights** and thus the output is usually formed as a **weighted average**.

Transformers

Background

- Convolutional neural networks (CNN) has been dominating
 - Greater scale
 - More extensive connections
 - More sophisticated forms of convolution
- Transformers
 - Competitive alternative to CNN
 - Generally found to perform best in settings with large amounts of training data
 - Enable multi-modality learning

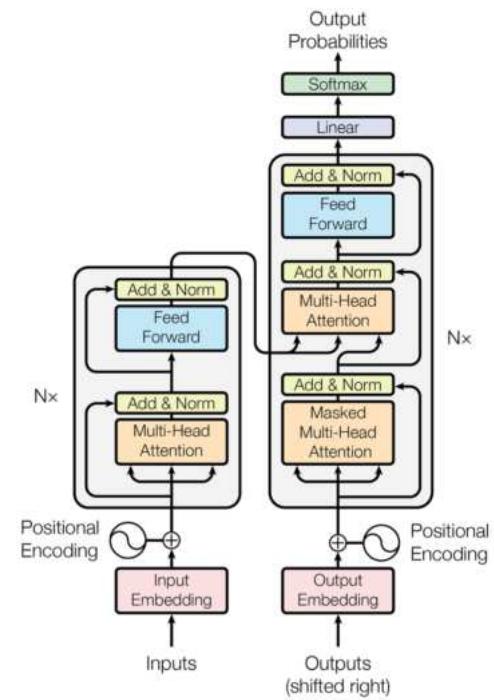
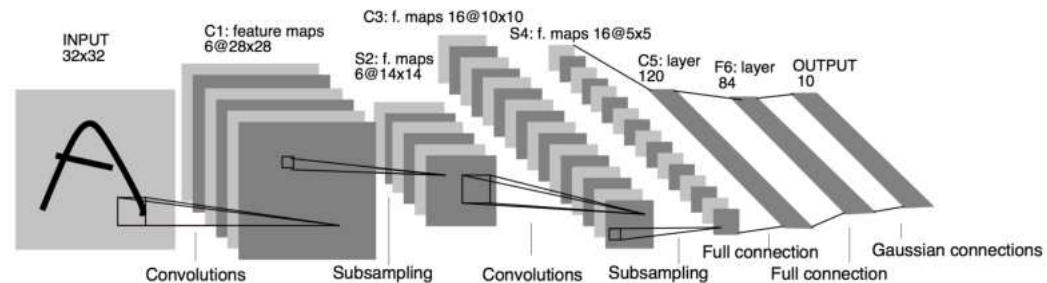


Figure 1: The Transformer - model architecture.

Background

◆ GPT-4



What is the best way to learn knowledge about Transformers (a machine learning model)?



Transformers are an important concept in modern machine learning, especially in the field of natural language processing. They are the key model behind the technology in GPT-3 and GPT-4, BERT, T5, and many other influential architectures.

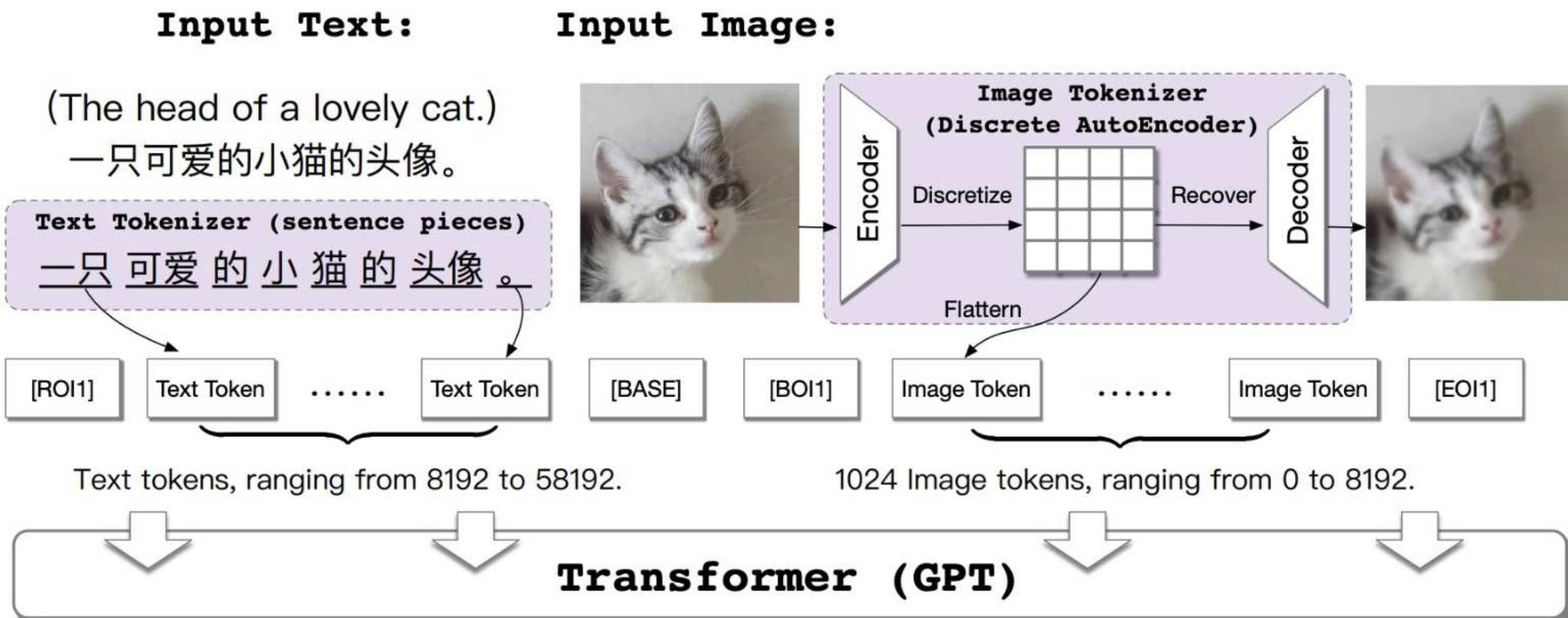


Here is a suggested path for learning about Transformers:

1. Understand the Basics of Machine Learning and Deep Learning

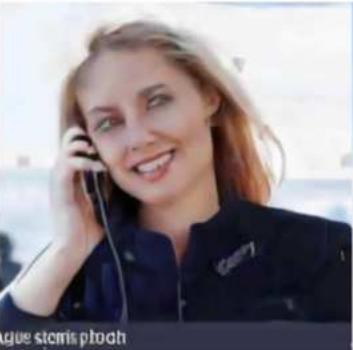
Before you get into Transformers, it's crucial to have a solid understanding of Machine Learning and Deep Learning. Knowing the fundamental concepts of Deep Learning such as neural networks, activation functions, loss functions, backpropagation, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Long Short Term Memory (LSTM) units is very helpful.

Background



Background

A beautiful young blond woman talking on a phone.



A Big Ben clock tower over the city of London.



A couple wearing leather biker garb rides a motorcycle.



A tiger is playing football.



A coffee cup printed with a cat. Sky background.



A man is flying to the moon on his bicycle.



Chinese traditional drawing. Statue of Liberty.



Oil painting. Lion.



Sketch. Houses.



Cartoon. A tiger is playing football.



Super-resolution: mid-lake pavilion



Transformers

Notable for its use of **attention** to model long-range dependencies in data

A sequence-to-sequence model

Model of choice in natural language processing (NLP)



Step-by-step guide to self-attention with illustrations and code <https://jalammar.github.io/illustrated-transformer/>

Ashish Vaswani et al., [Attention Is All You Need](#), NIPS 2017 (from Google)

Transformers

Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder)

But it differs from the existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).

- During training, layer outputs can be calculated in parallel, instead of a series like an RNN
- Attention-based models allow modeling of dependencies without regard to their distance in the input or output sequences

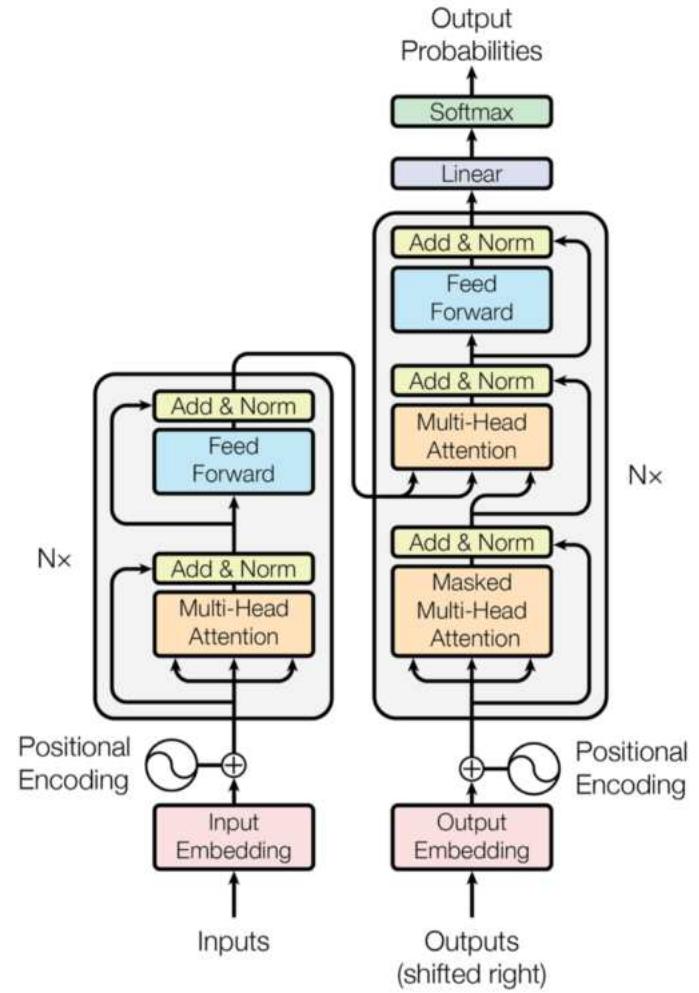


Figure 1: The Transformer - model architecture.

Transformers

Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified by learned weights** and thus the output is usually formed as a **weighted average**.

$$\text{Attention} = \mathbf{W}\mathbf{V}$$

Input Attention

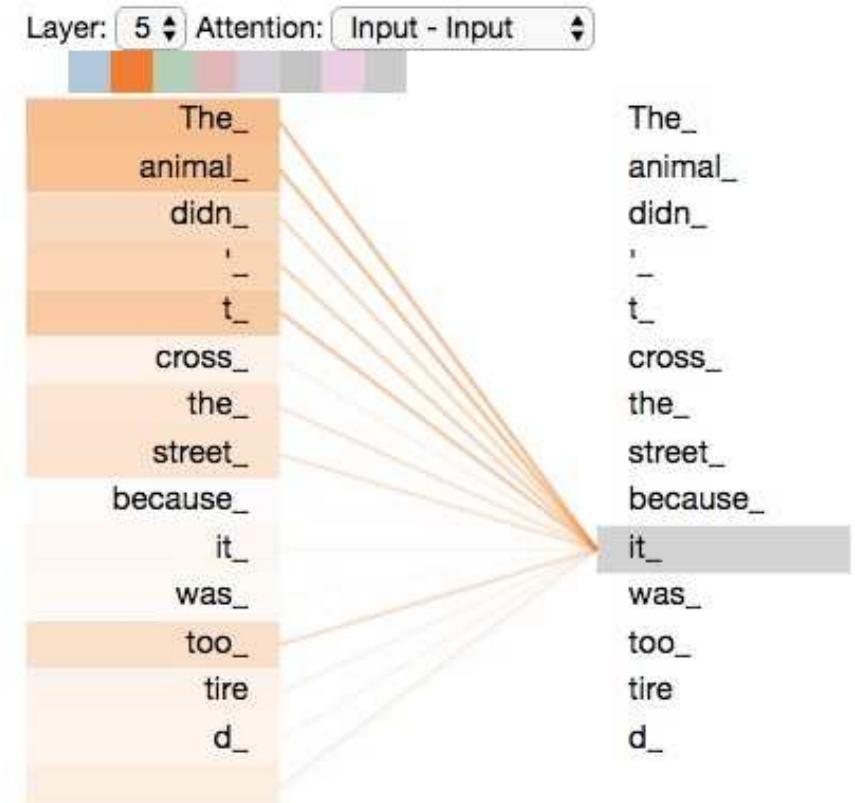


The model attends to image regions that are semantically relevant for classification

Transformers

It is entirely built on the **self-attention** mechanisms without using sequence-aligned recurrent architecture

Self-attention is a type of attention mechanism where the model **makes prediction for one part of a data sample using other parts of the observation about the same sample.**



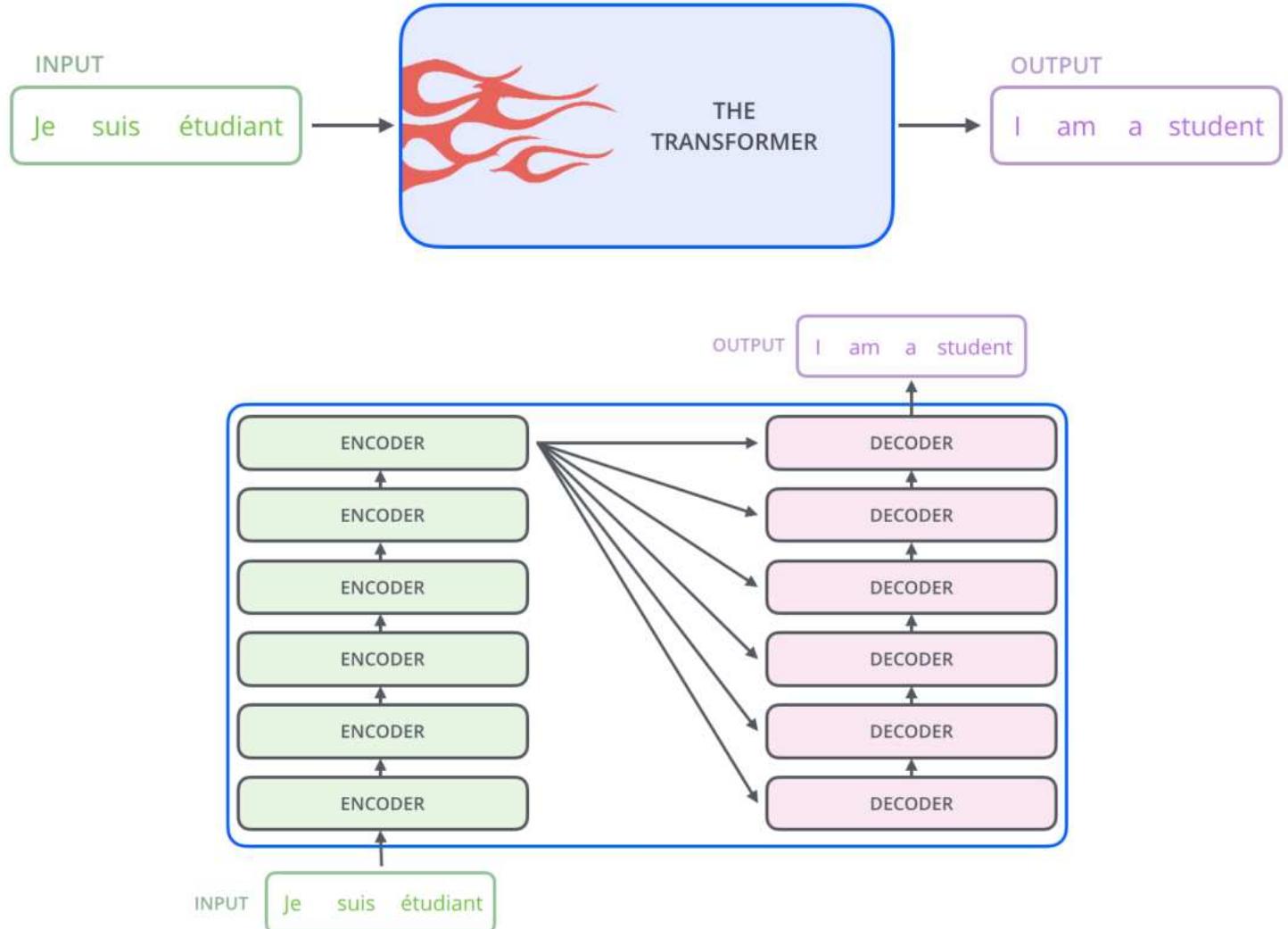
What does “it” in this sentence refer to? Is it referring to the street or to the animal?

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

Transformers

The encoding component is a stack of encoders

The decoding component is a stack of decoders of the same number

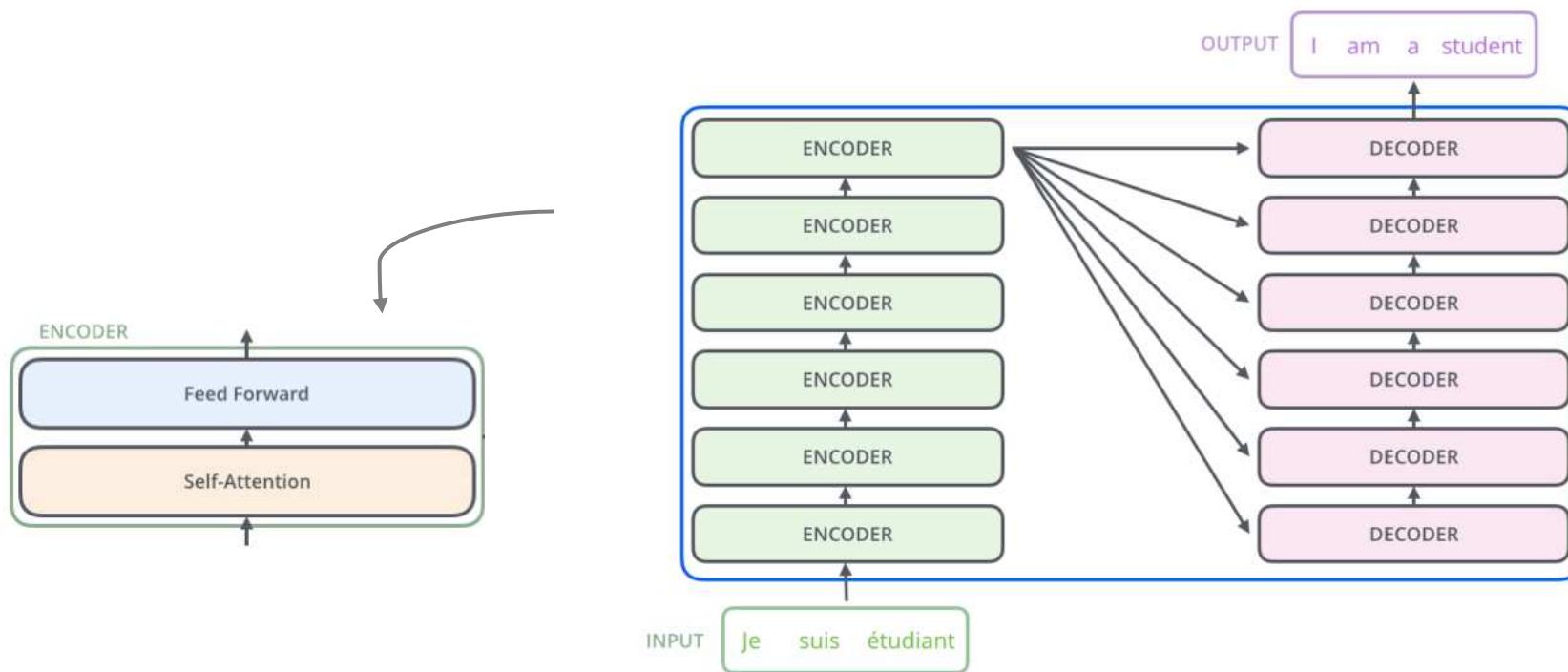


Transformers

The encoder's inputs first flow through a self-attention layer – a layer that **helps the encoder look at other words in the input sentence** as it encodes a specific word

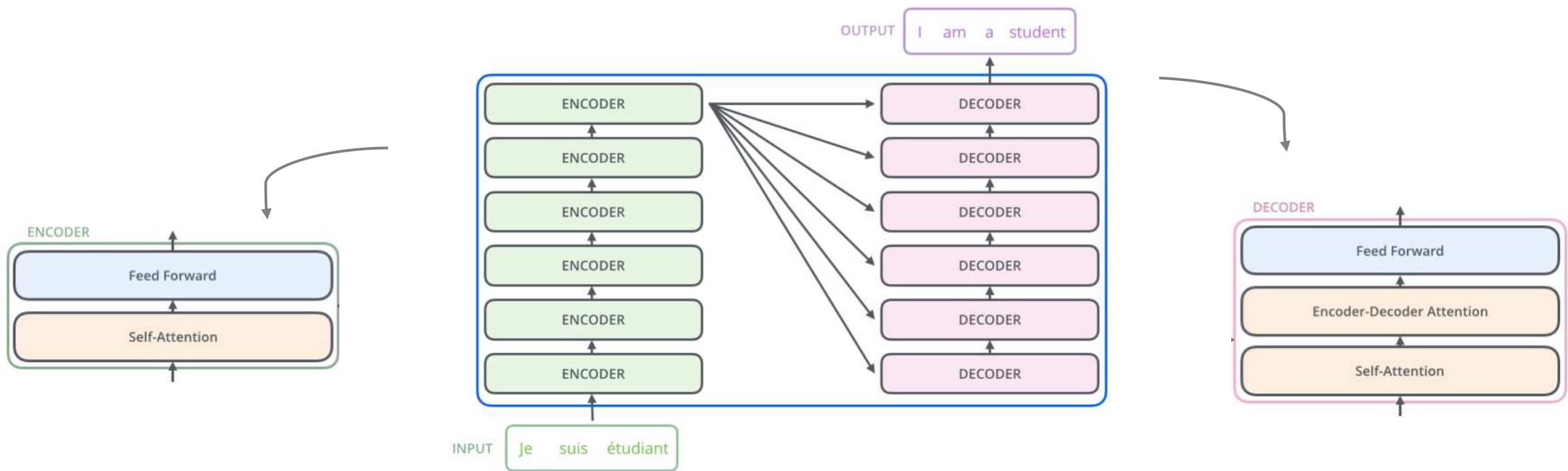
The outputs of the self-attention layer are fed to a **feed-forward neural network**.

The exact same feed-forward network is **independently applied** to each position (each word/token).

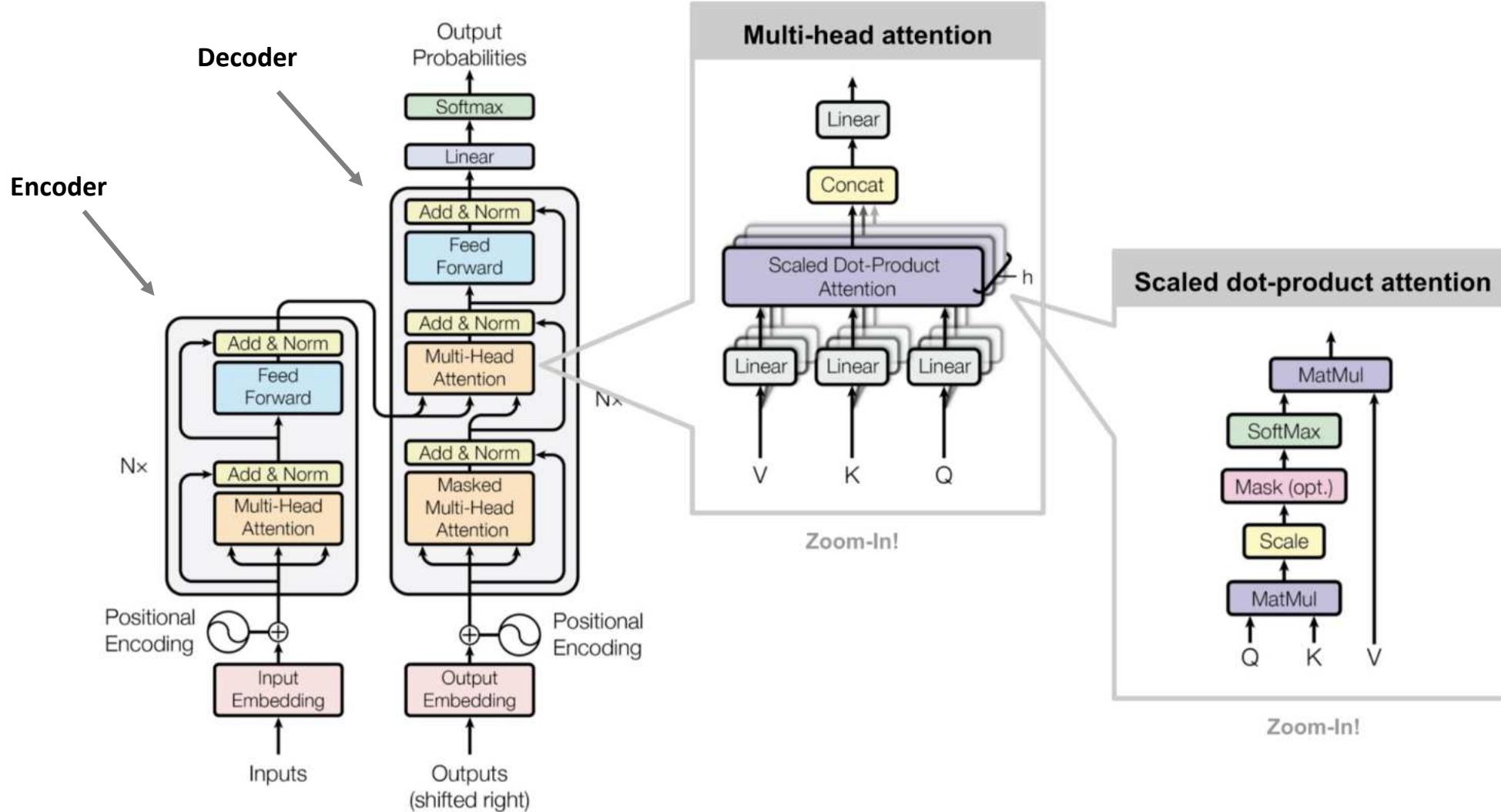


Transformers

The decoder has both those layers, but between them is an attention layer that **helps the decoder focus on relevant parts of the input sentence**



Transformers

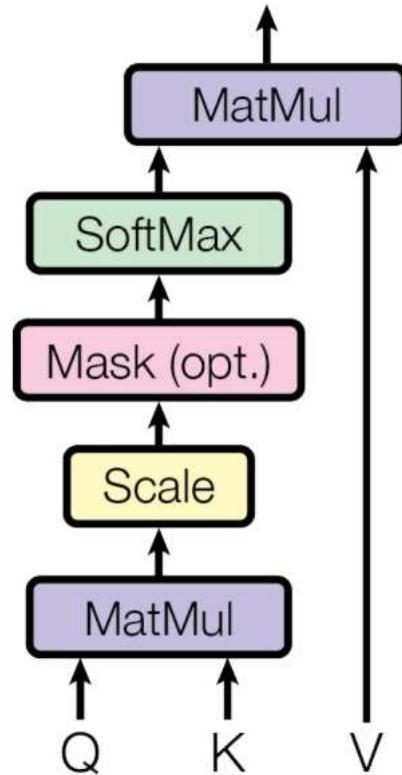


Transformers

Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$



Scaled Dot-Product Attention

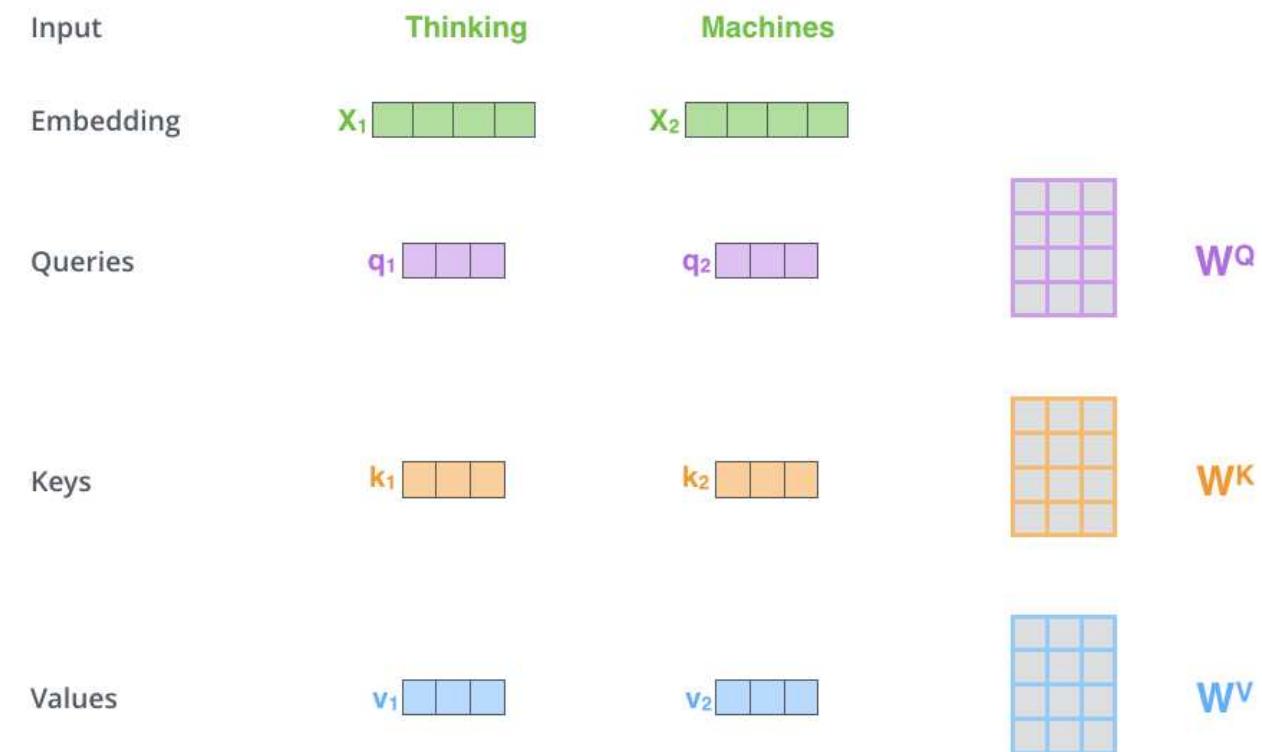
Self-Attention in Detail

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).

So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

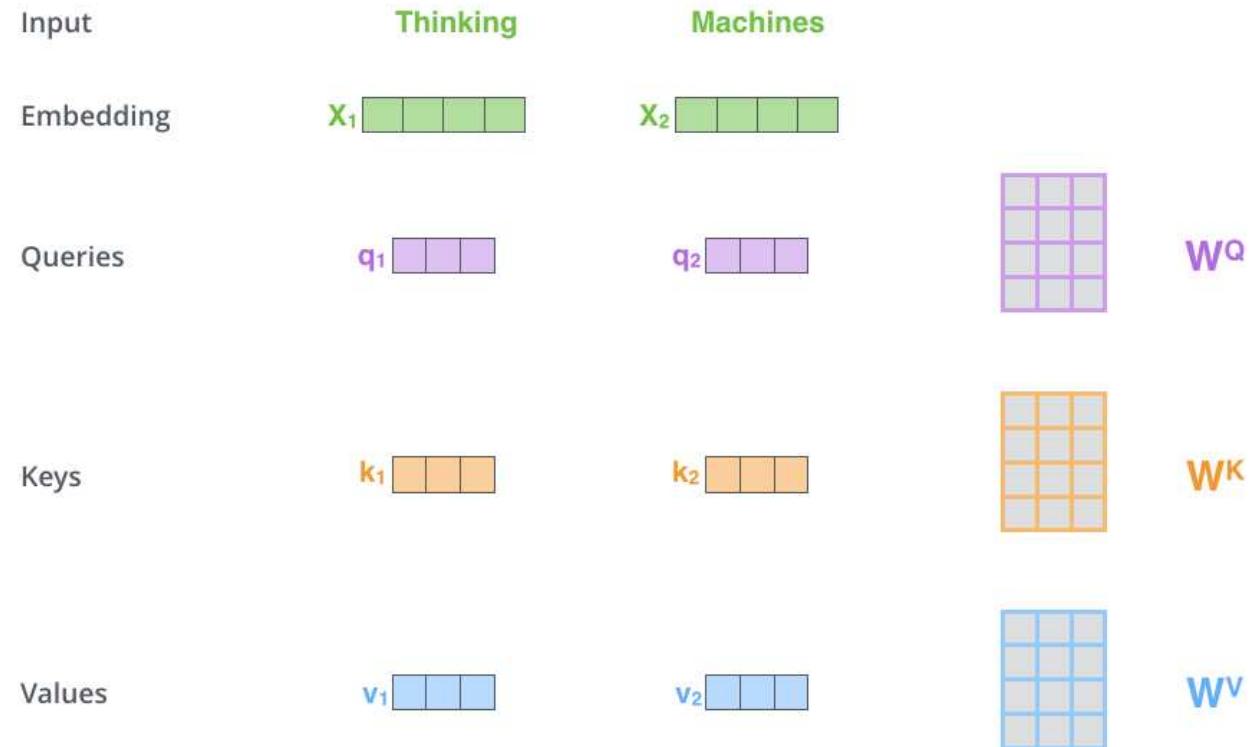
First Step

What are the “query”, “key”, and “value” vectors?

The names “query”, “key” are inherited from the field of **information retrieval**

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{QK^T}{\sqrt{d_k}}$ depend on the relative similarities between the n -th **query** and all of the **keys**

The softmax function means that the **key** vectors “compete” with one another to contribute to the final result.



$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

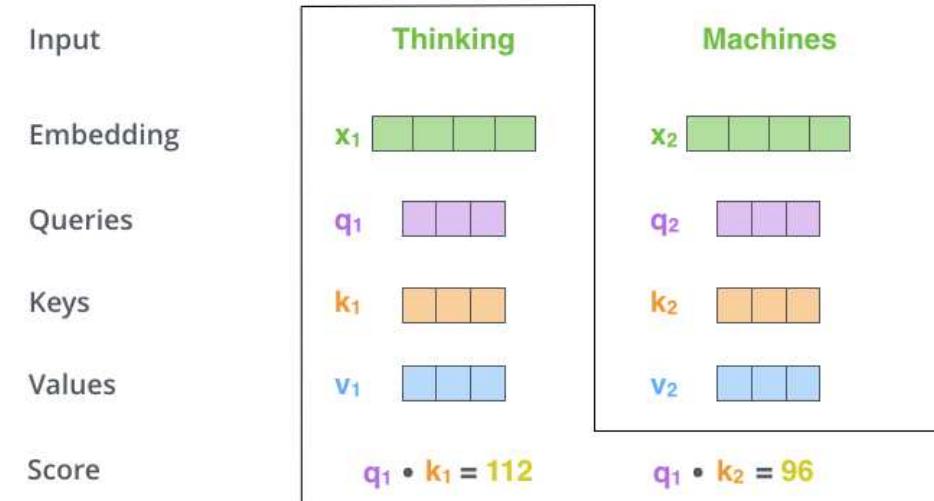
Self-Attention in Detail

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.



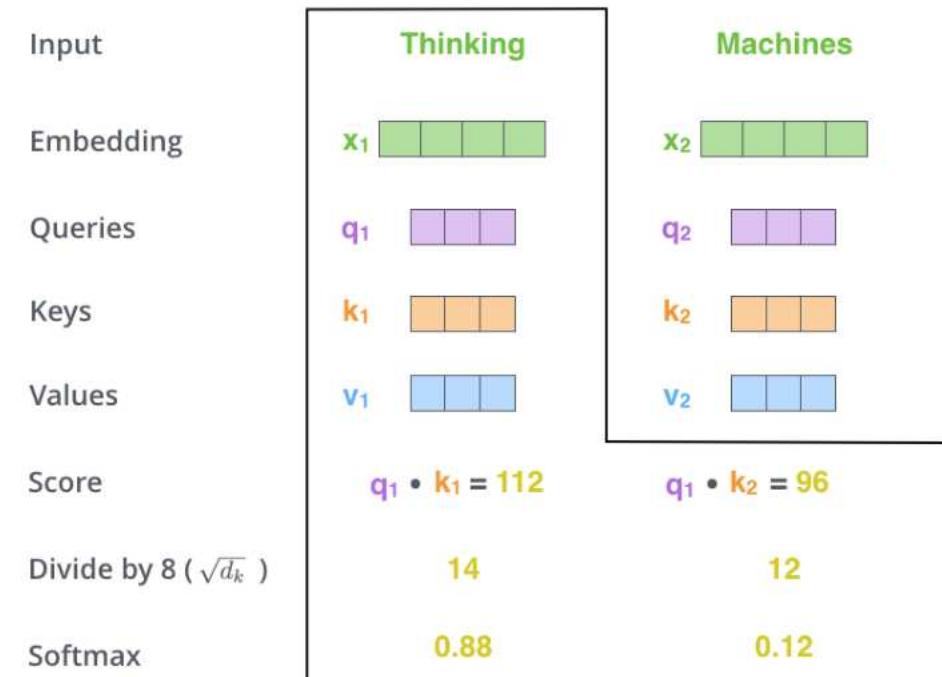
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)



Fourth Step

Softmax for normalization

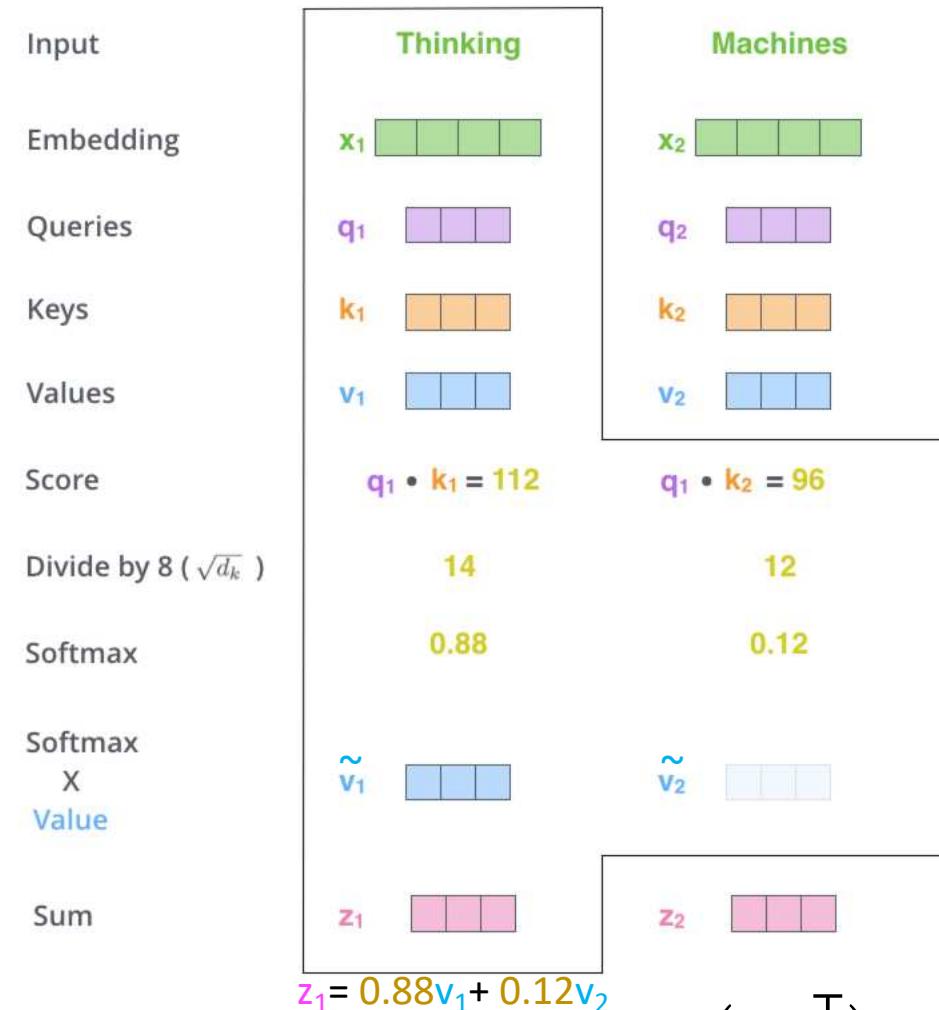
$$z_1 = q_1 \cdot k_1 / \sqrt{d_k} = 112 / \sqrt{64} = 112 / 8 = 14, z_2 = q_1 \cdot k_2 / \sqrt{d_k} = 96 / \sqrt{64} = 96 / 8 = 12$$
$$\text{softmax}(z_1) = \exp(z_1) / \sum_{i=1}^2 (\exp(z_i)) = 0.88, \text{softmax}(z_2) = \exp(z_2) / \sum_{i=1}^2 (\exp(z_i)) = 0.12$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

Fifth Step

Multiply each value vector by the softmax score



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Matrix Calculation of Self-Attention

First Step

Calculate the Query, Key, and Value matrices.

Pack our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q , W^K , W^V)

Every row in the X matrix corresponds to a word in the input sentence.

$$X \times W^Q = Q$$

$$X \times W^K = K$$

$$X \times W^V = V$$

Second Step

Calculate the outputs of the self-attention layer.

SoftMax is **row-wise**

$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) V = Z$$

Example

Assuming we have two sequences:

(1, 2, 3)
(4, 5, 6)

And the given \mathbf{W}^Q , \mathbf{W}^K , \mathbf{W}^V matrices are respectively given as

$$\begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix}, \begin{pmatrix} 0.05 & 0.05 \\ 0.06 & 0.05 \\ 0.07 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix}$$

Compute the output of the scaled-dot product attention, $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$.

Example

Step 1: Find $\mathbf{Q}, \mathbf{K}, \mathbf{V}$

$$\mathbf{Q} = \mathbf{XW}^Q = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix}$$

$$\mathbf{K} = \mathbf{XW}^K = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.5 & 0.5 \\ 0.6 & 0.5 \\ 0.7 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.38 & 0.30 \\ 0.92 & 0.75 \end{pmatrix}$$

$$\mathbf{V} = \mathbf{XW}^V = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix}$$

Example

Step 2: Find Attention($\mathbf{Q}, \mathbf{K}, \mathbf{V}$)

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\frac{e^{z_j}}{\sum_j e^{z_j}} = \frac{e^{0.0588}}{e^{0.0588} + e^{0.1441}} = 0.4787$$

Perform row-wise SoftMax

$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) = \text{Softmax}\left(\frac{\begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix} \begin{pmatrix} 0.38 & 0.92 \\ 0.30 & 0.75 \end{pmatrix}}{\sqrt{2}}\right) = \text{Softmax}\begin{pmatrix} 0.0588 & 0.1441 \\ 0.1454 & 0.3566 \end{pmatrix} = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix}$$

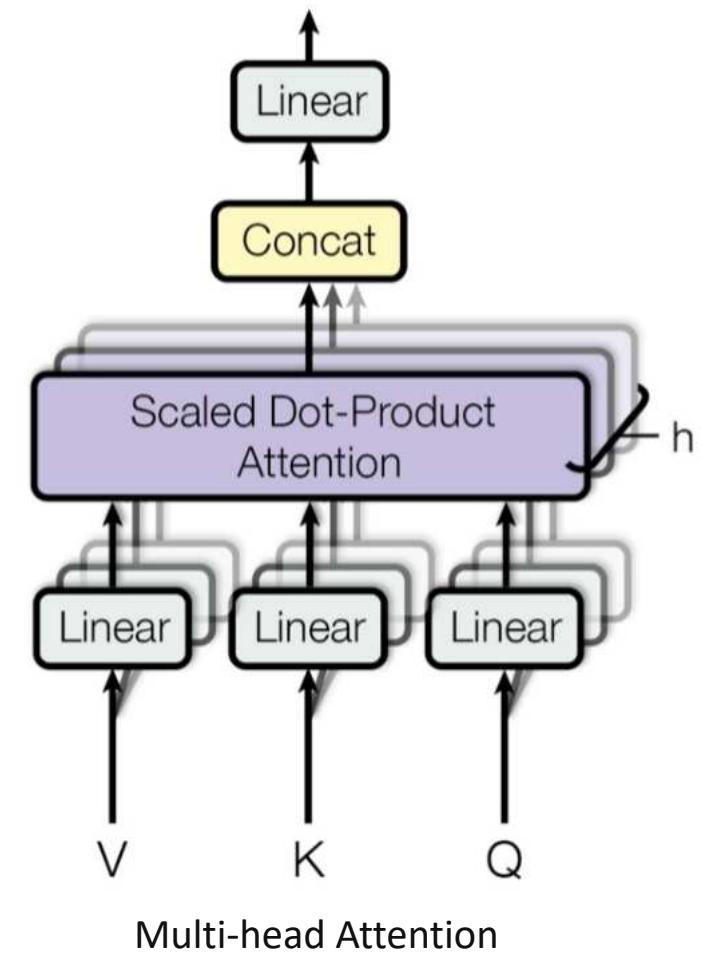
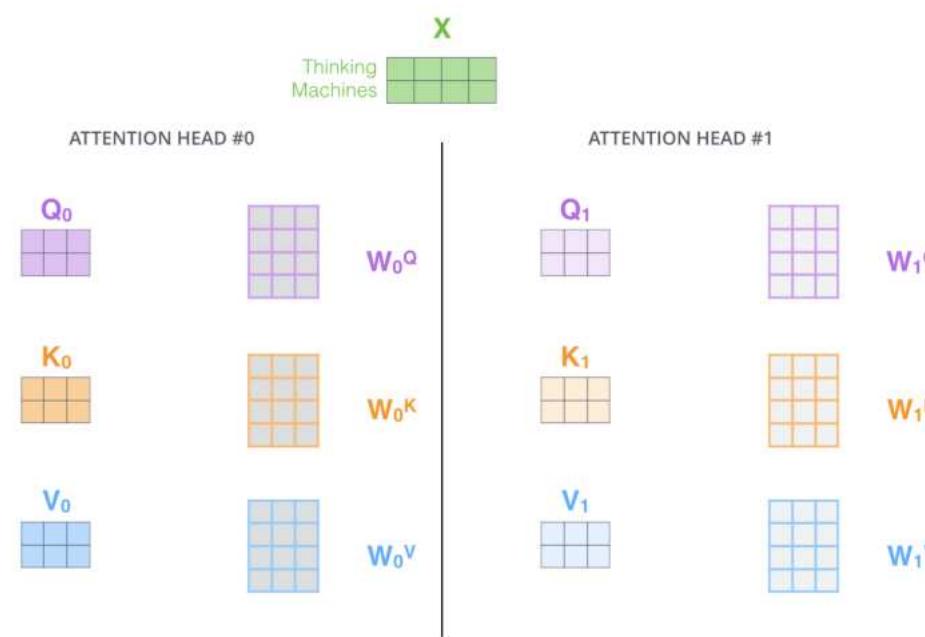
$$\text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix} \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix} = \begin{pmatrix} 0.1326 & 0.1682 \\ 0.1363 & 0.1729 \end{pmatrix}$$

Multi-Head Self-Attention

Multi-Head Self-Attention

Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention **multiple times in parallel**.

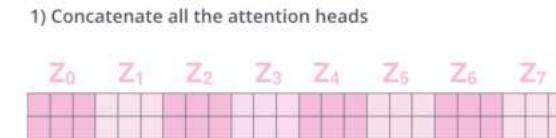
Due to different linear mappings, each head is presented with different versions of keys, queries, and values



Multi-Head Self-Attention

Multi-Head Self-Attention

The independent attention outputs are simply **concatenated and linearly transformed** into the expected dimensions.



2) Multiply with a weight matrix W^O that was trained jointly with the model

x

W^O

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

ATTENTION
HEAD #0

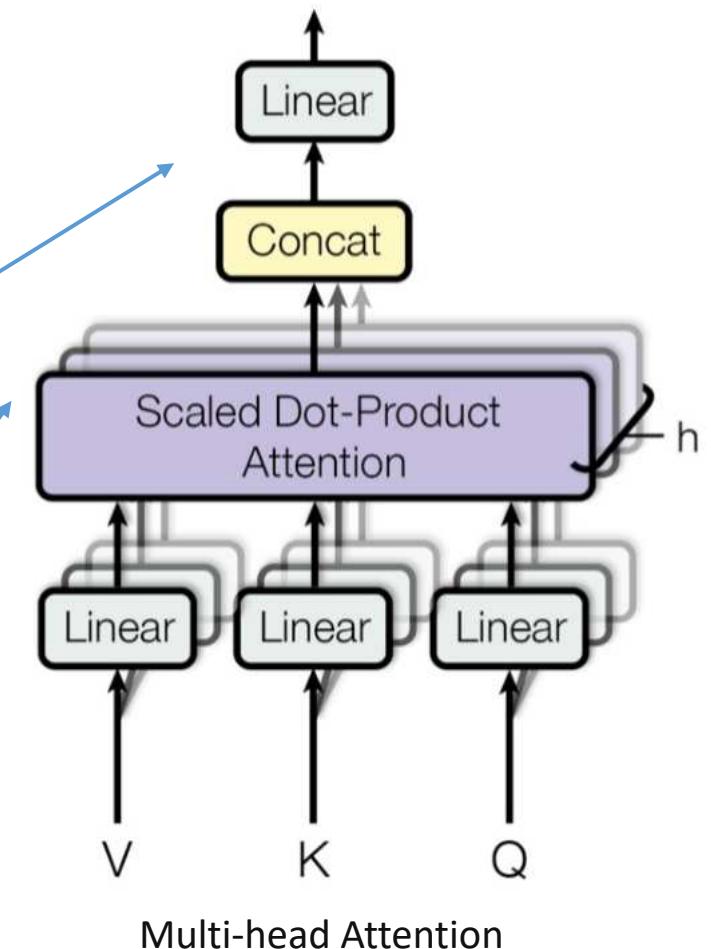


ATTENTION
HEAD #1



...

ATTENTION
HEAD #7

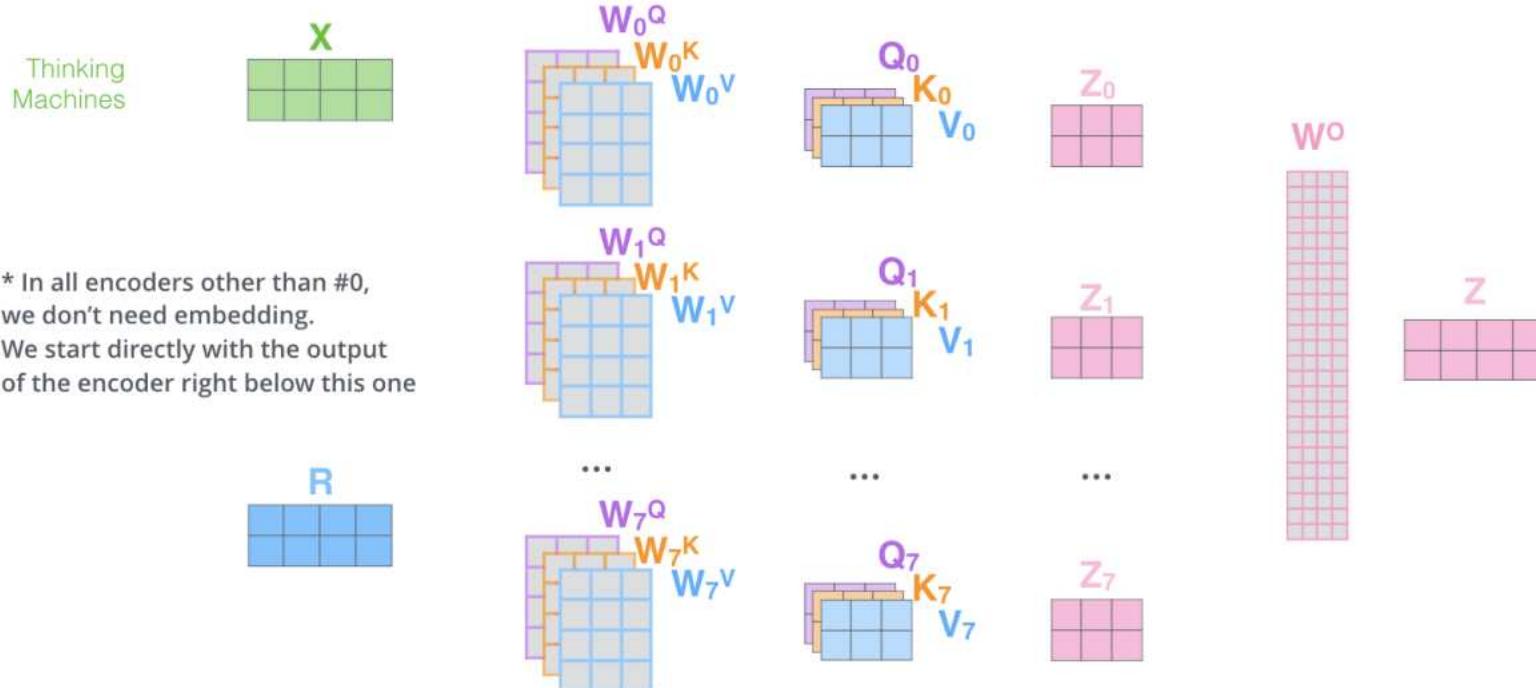


Multi-Head Self-Attention

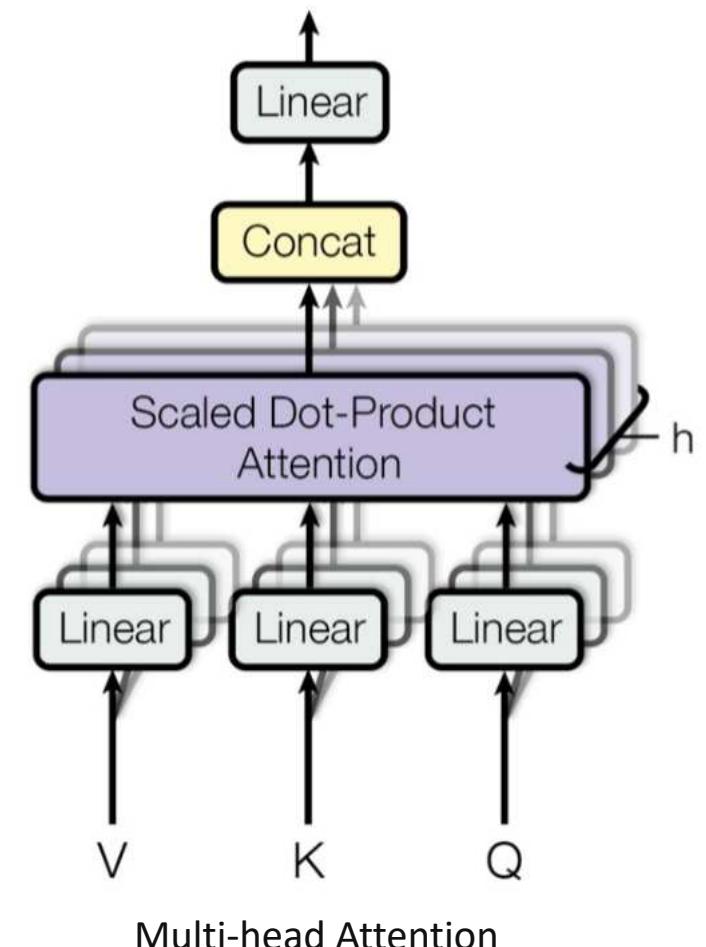
Multi-Head Self-Attention

The big picture. Note that after the split each head can have a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.

- 1) This is our input sentence*
- 2) We embed each word*
- 3) Split into 8 heads. We multiply X or R with weight matrices
- 4) Calculate attention using the resulting $Q/K/V$ matrices
- 5) Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer



* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



Multi-Head Self-Attention

Why?

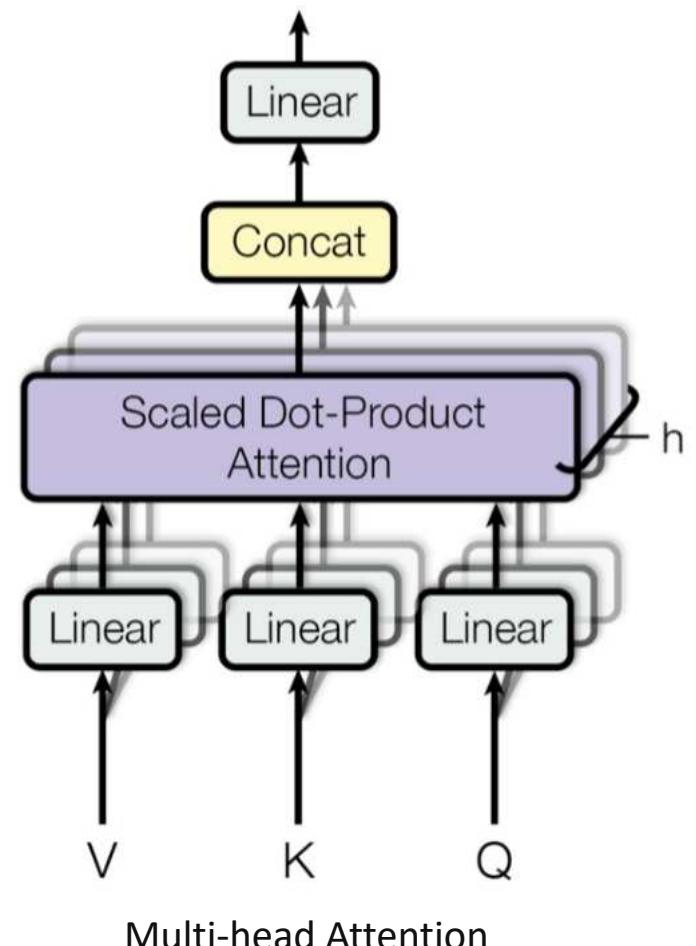
*“Multi-head attention allows the model to jointly attend to information from different representation **subspaces** at different positions.”*

An intuitive example

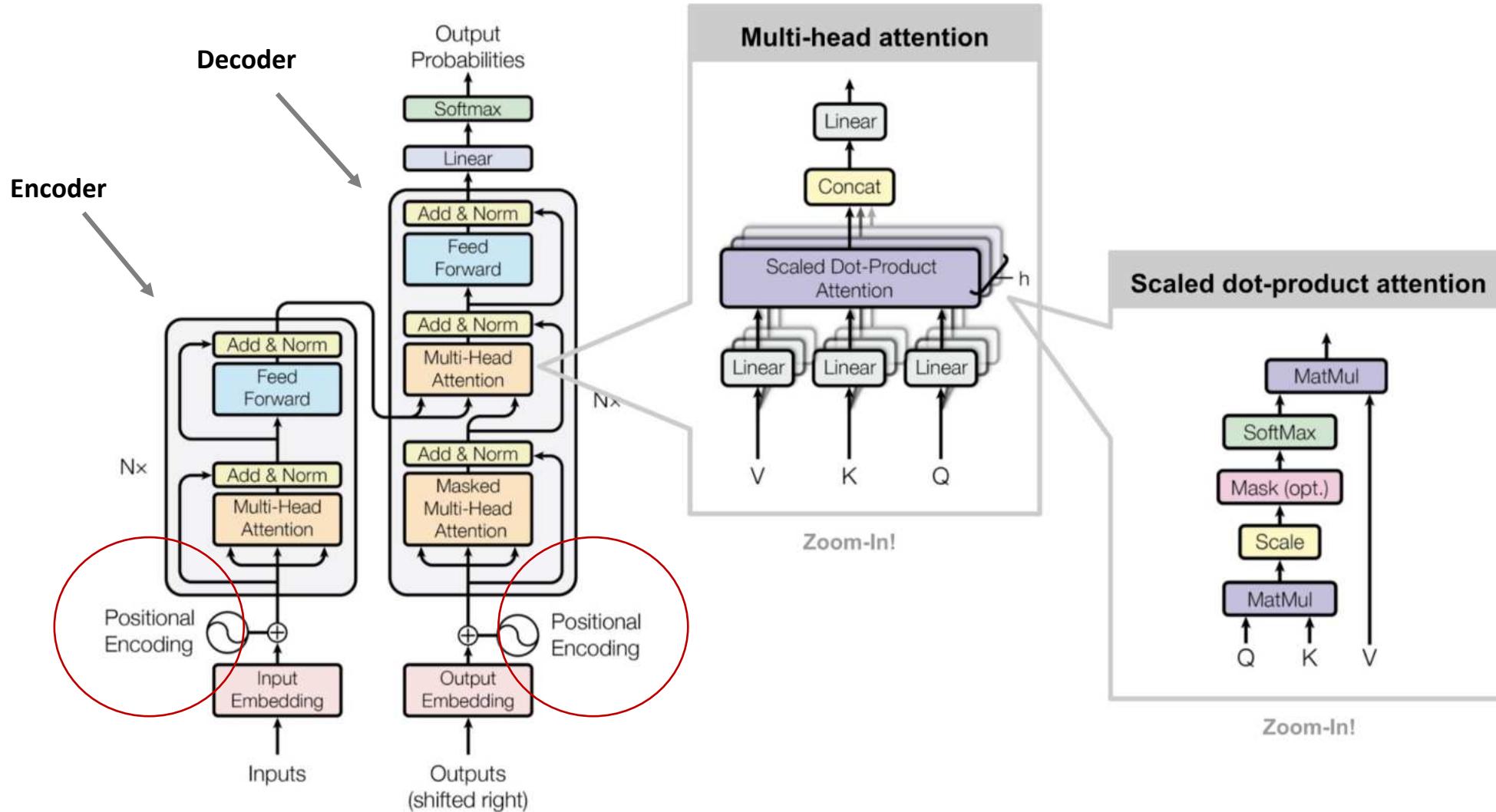
Given a sentence:

“Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning.”

Given “representation learning”, the first head attends to “Deep learning” while the second head attends to the more general term “machine learning methods”.



Positional Encoding



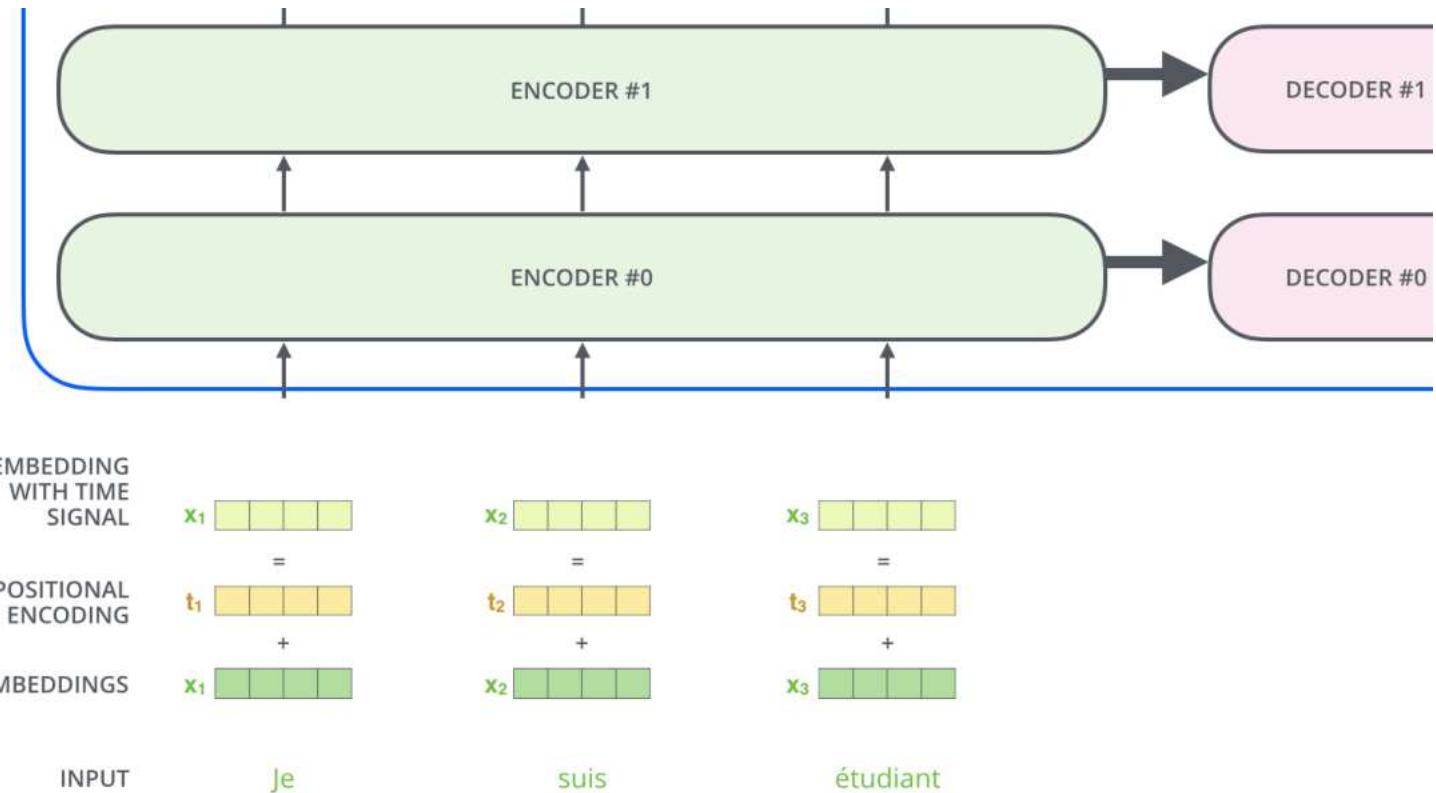
Positional Encoding

Self-attention layer works on sets of vectors and it **doesn't know the order** of the vectors it is processing

The positional encoding has the **same dimension** as the input embedding

Adds a vector to each input embedding to give information about the **relative or absolute position** of the tokens in the sequence

These vectors follow a specific pattern



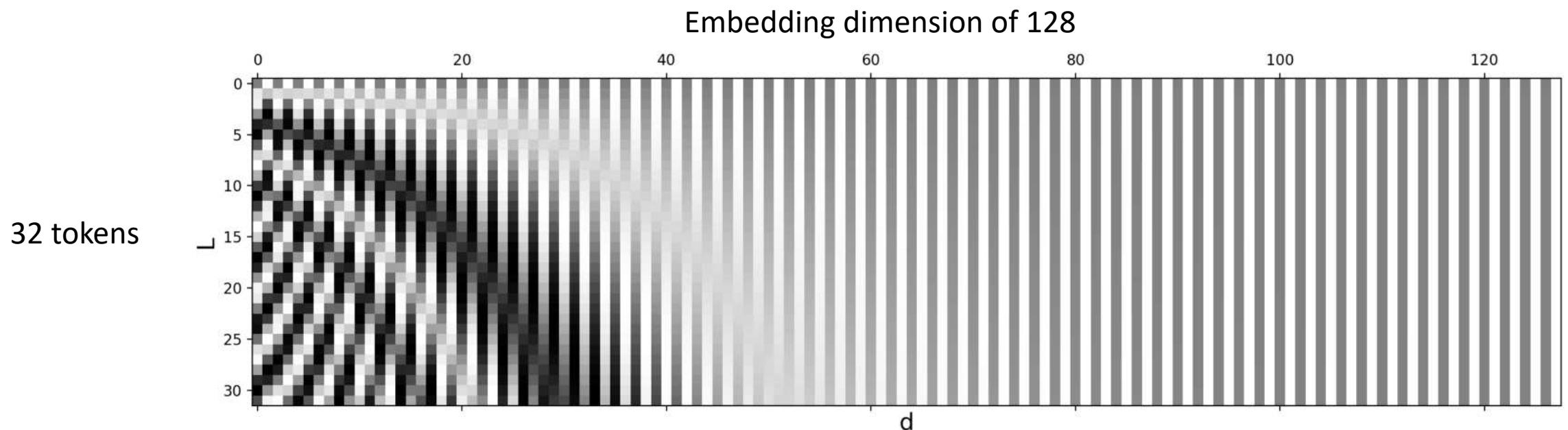
Positional Encoding

What might this pattern look like?

Each row corresponds to a positional encoding of a vector.

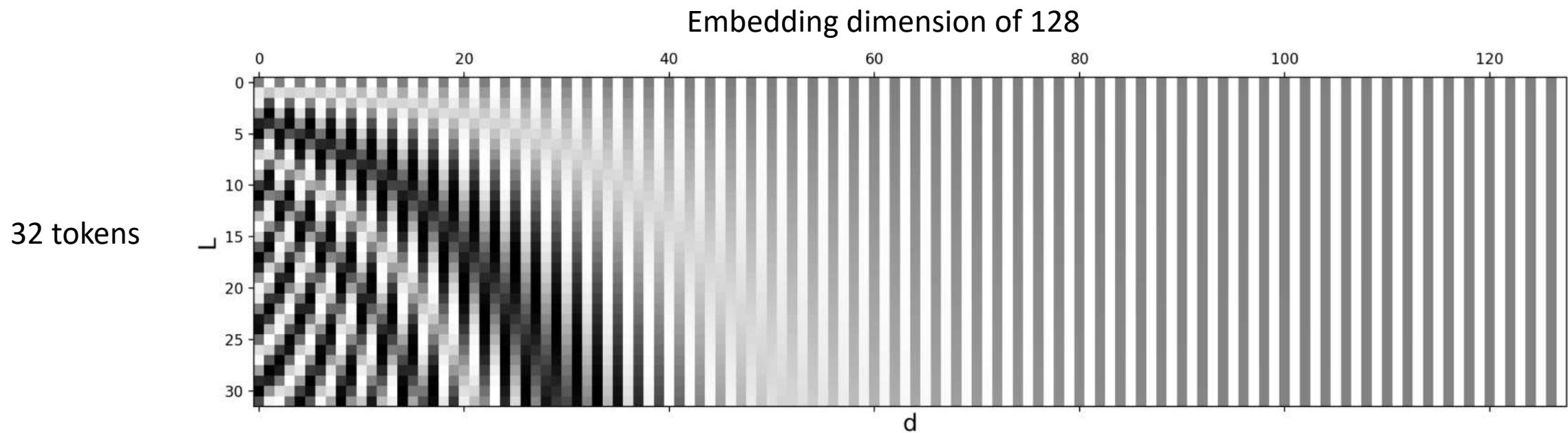
The first row would be the vector we'd add to the embedding of the first word in an input sequence.

Each position is uniquely encoded and the encoding can deal with sequences longer than any sequence seen in the training time.



Sinusoidal positional encoding with 32 tokens and embedding dimension of 128. The value is between -1 (black) and 1 (white) and the value 0 is in gray.

Positional Encoding



Sinusoidal positional encoding - interweaves the two signals (sine for even indices and cosine for odd indice)

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension, $[0, \dots, d_{\text{model}}/2]$

Example: Positional Encoding

Example:

Given the following *Sinusoidal positional encoding*, calculate the $PE(pos = 1)$ for the first five dimensions [0, 1, 2, 3, 4]. Assume $d_{model} = 512$

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Solution:

Given $pos = 1$ and $d_{model} = 512$

At dimension 0, $2i = 0$ thus $i = 0$, therefore $PE_{(pos,2i)} = PE_{(1,0)} = \sin(1/10000^{0/512})$

At dimension 1, $2i + 1 = 1$ thus $i = 0$, therefore $PE_{(pos,2i+1)} = PE_{(1,1)} = \cos(1/10000^{0/512})$

At dimension 2, $2i = 2$ thus $i = 1$, therefore $PE_{(pos,2i)} = PE_{(1,2)} = \sin(1/10000^{2/512})$

At dimension 3, $2i + 1 = 3$ thus $i = 1$, therefore $PE_{(pos,2i+1)} = PE_{(1,3)} = \cos(1/10000^{2/512})$

At dimension 4, $2i = 4$ thus $i = 2$, therefore $PE_{(pos,2i)} = PE_{(1,4)} = \sin(1/10000^{4/512})$

Implementation

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000): ←
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

The constructor initializes the module and sets up the dropout layer with the given dropout probability.

`d_model` is the dimension of the embeddings (or the depth of the model).

`max_len` is the maximum expected length of the sequences.

Implementation

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1) ←
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

position is a tensor containing integers from 0 to max_len-1, representing each position in the sequence.

Implementation

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

div_term is a scaling term used to adjust the rate of the sinusoidal functions.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

Implementation

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

The sinusoidal functions (sine for even indices and cosine for odd indices) are applied to the positions and the results are stored in `pe`. This creates a unique positional encoding for each position.

Implementation

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

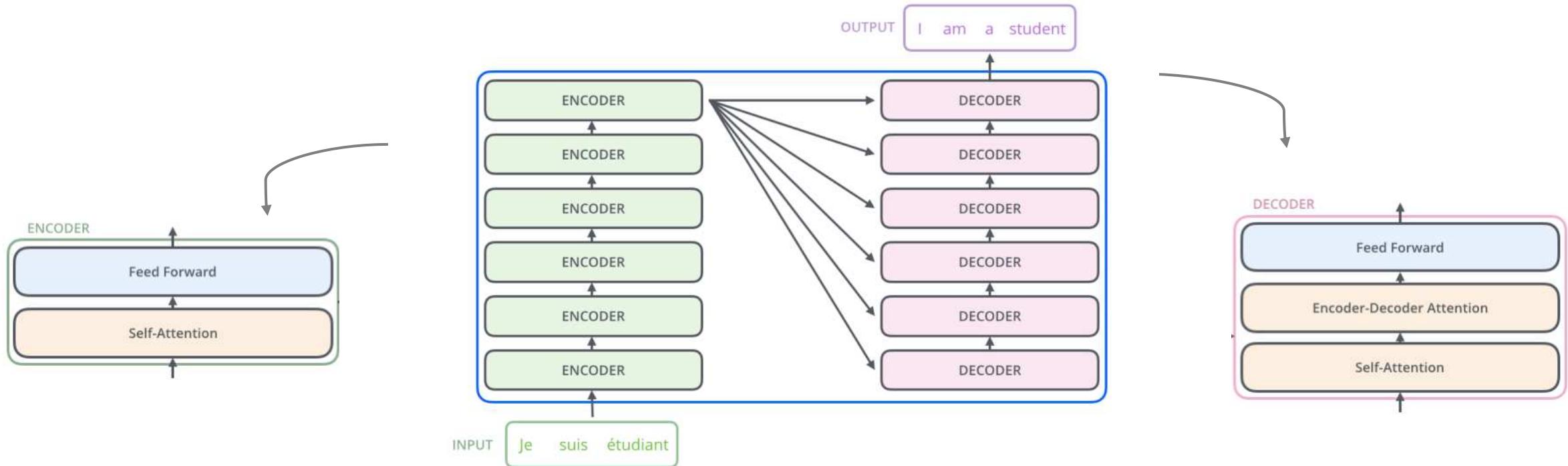
        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Arguments:
            x: Tensor, shape ``[seq_len, batch_size, embedding_dim]``
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

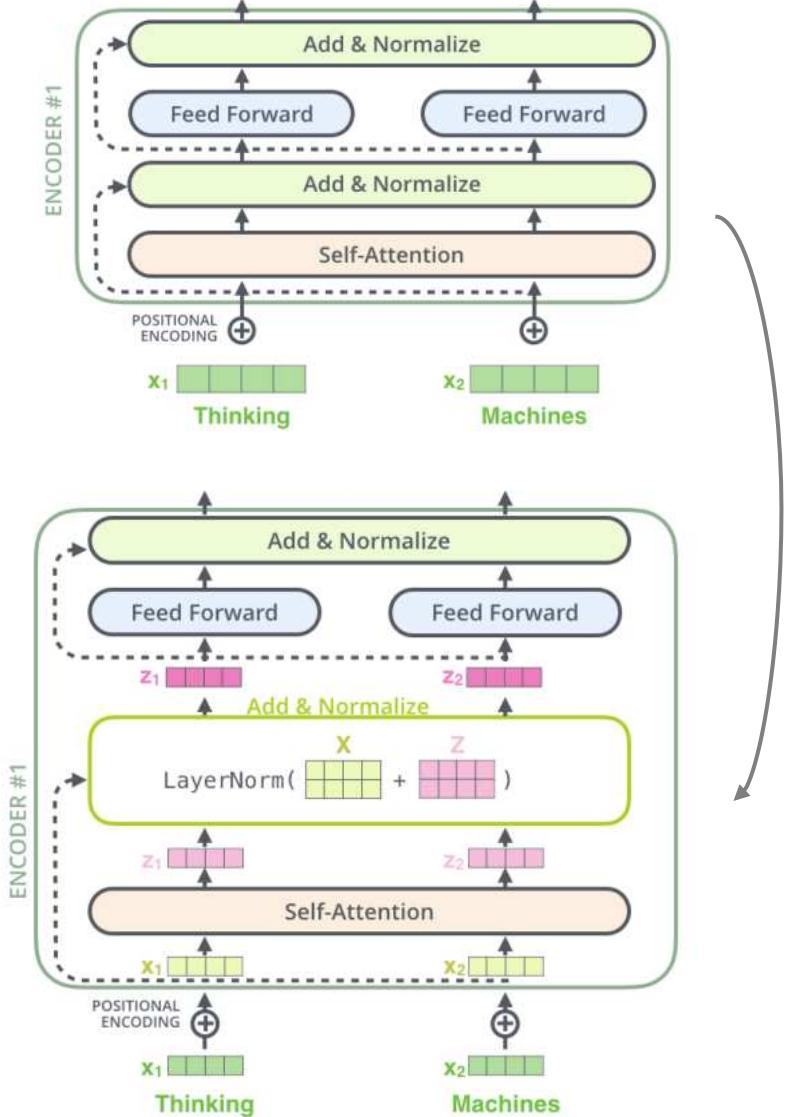
The positional encoding (`self.pe`) corresponding to the length of the input sequence is added to the input tensor `x`.

This addition operation effectively combines the positional information with the embeddings of the tokens.

Transformers



Transformer Encoder



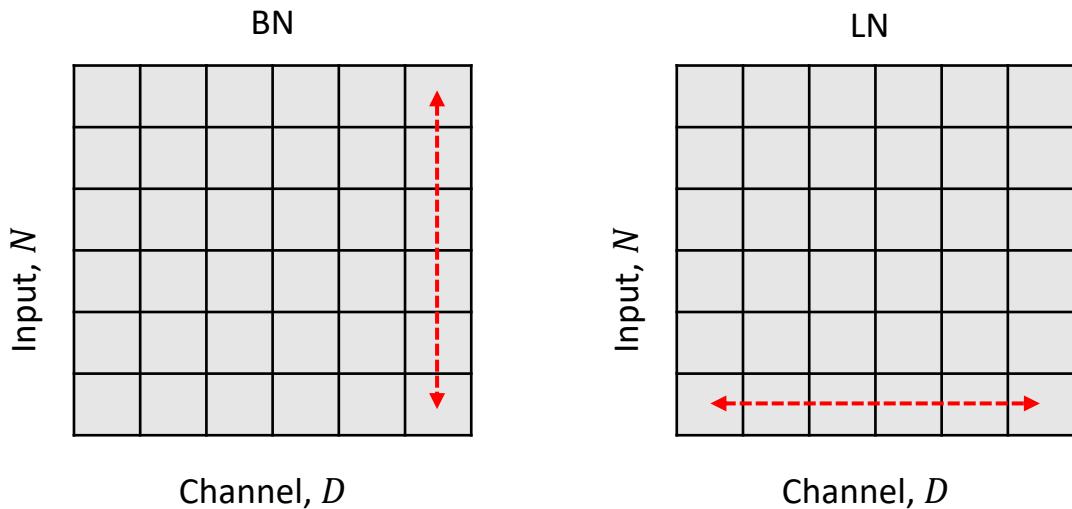
Encoder

- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

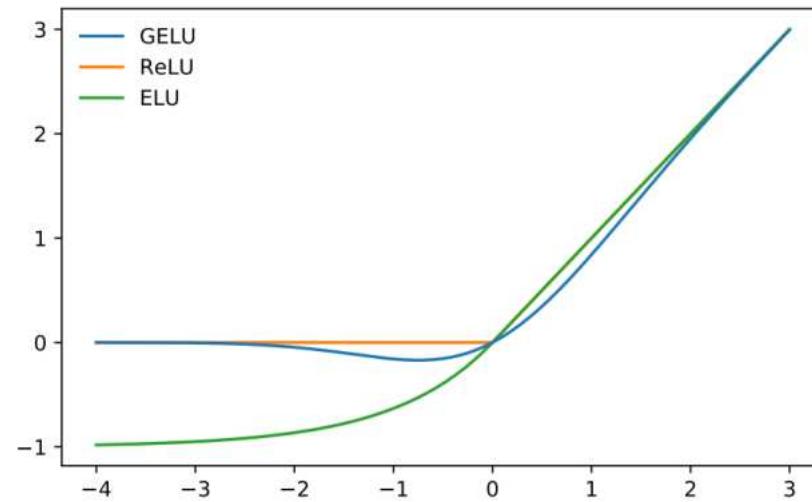
- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

Transformer Encoder



Layer Normalization (LN)¹

- The pixels along the red arrow are normalized by the same mean and variance, computed by aggregating the values of these pixels.
- BN is found unstable in Transformers²
- Works well with RNN and now being used in Transformers



Gaussian Error Linear Units (GELU)³

- Can be thought of as a smoother ReLU
- Used in GPT-3, BERT, and most other Transformers

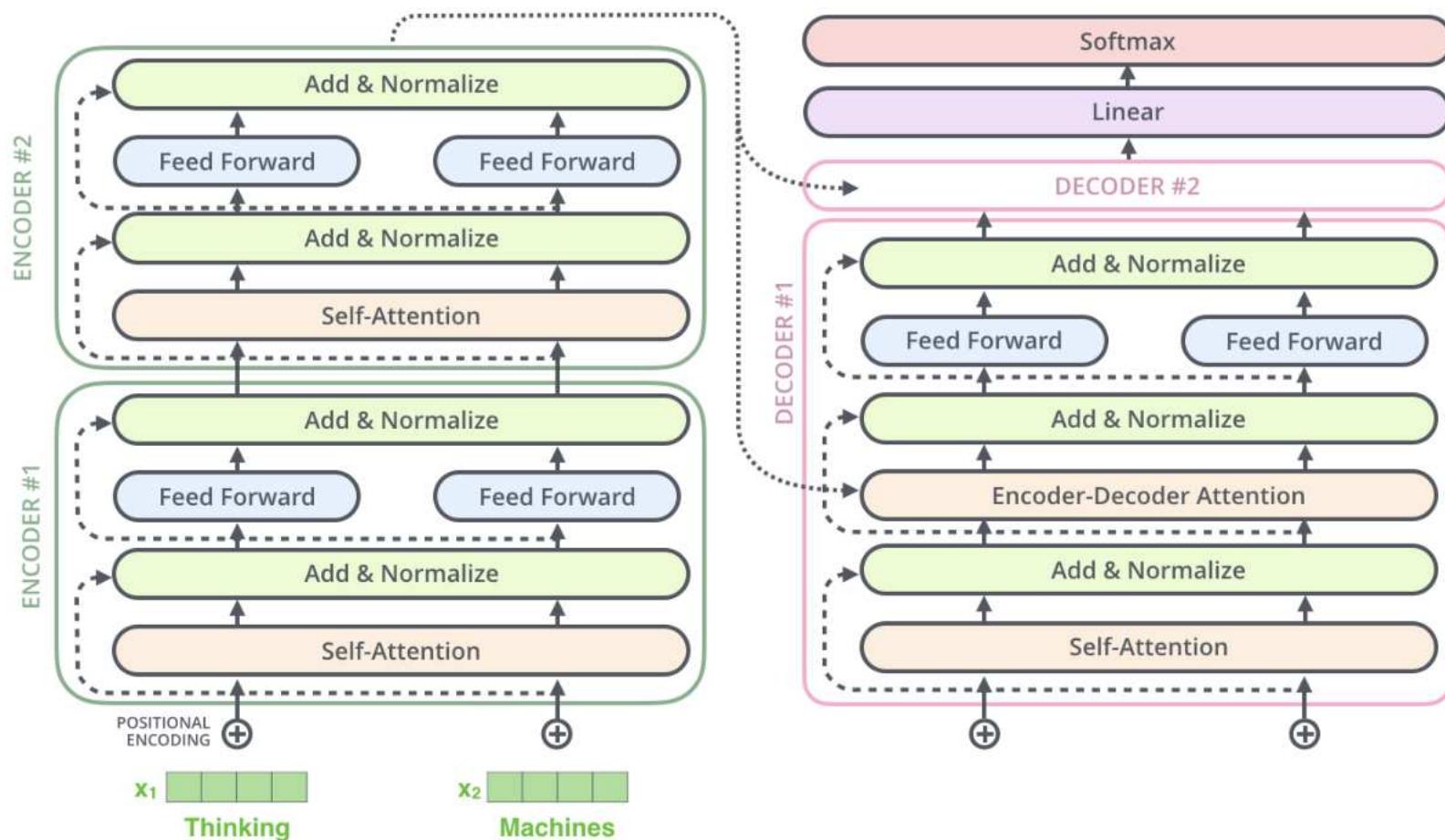
¹ Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton, [Layer Normalization](#), arXiv:1607.06450

² Sheng Shen, Zhewei Yao, Amir Gholami, Michael Mahoney, Kurt Keutzer, [Rethinking Batch Normalization in Transformers](#), ICML 2020

³ Dan Hendrycks, Kevin Gimpel, [Gaussian Error Linear Units \(GELUs\)](#), arXiv:1606.08415

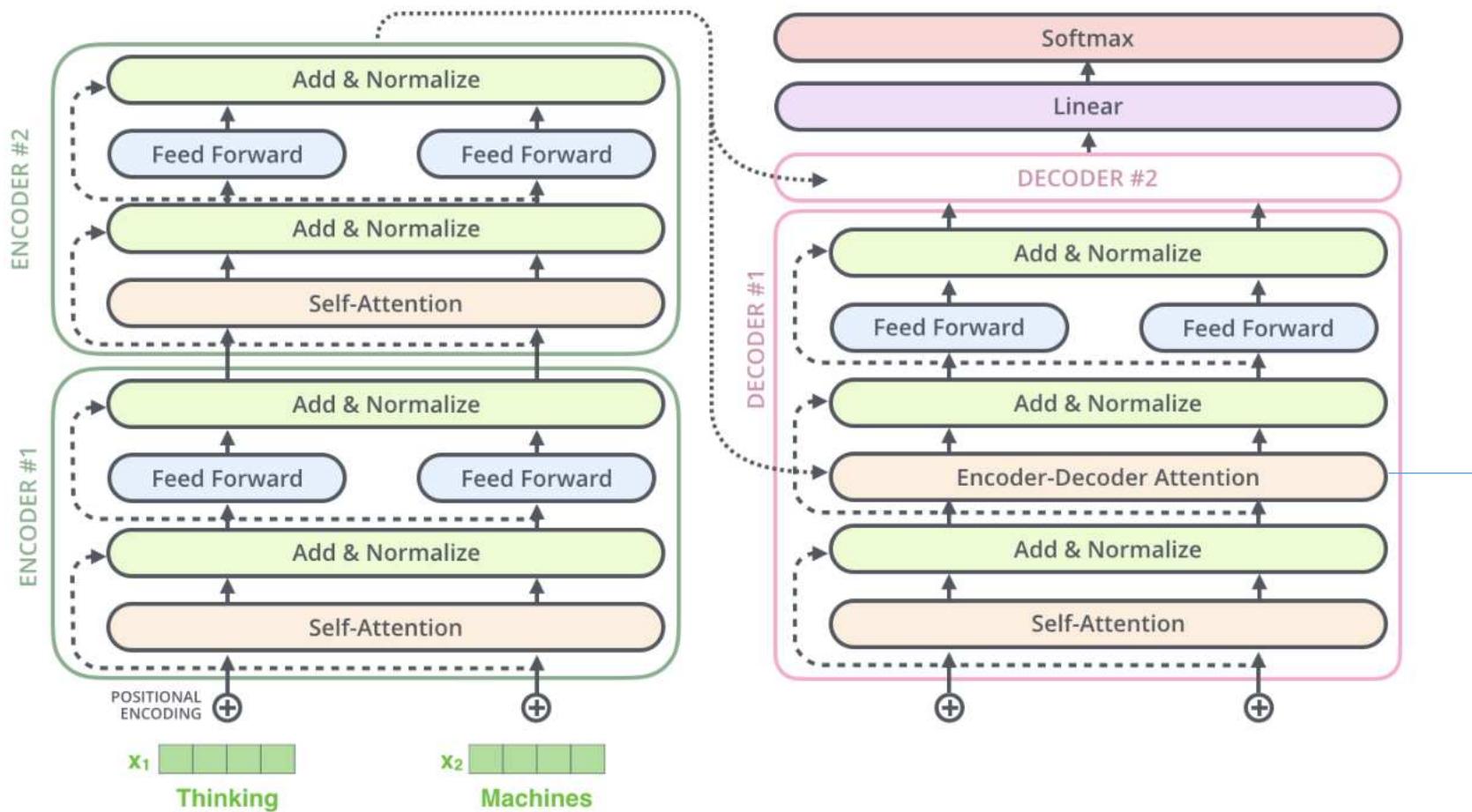
Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:

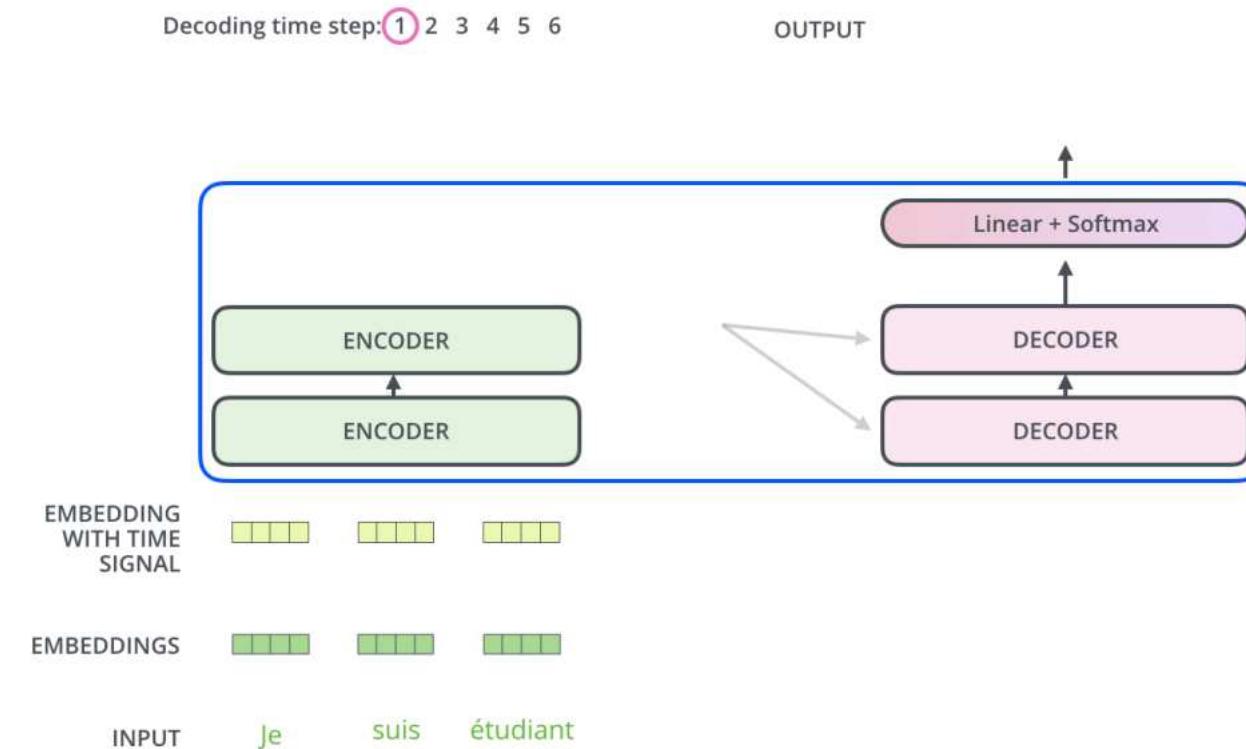


The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the **Keys** and **Values** matrix from the output of the encoder stack.

Transformer Decoder

How encoder and decoder work together

- The encoder starts by processing the input sequence
- The output of the top encoder is then transformed into a set of attention vectors **K** and **V**.
- These are to be used by each decoder in its “encoder-decoder attention” layer which helps the decoder focus on appropriate places in the input sequence

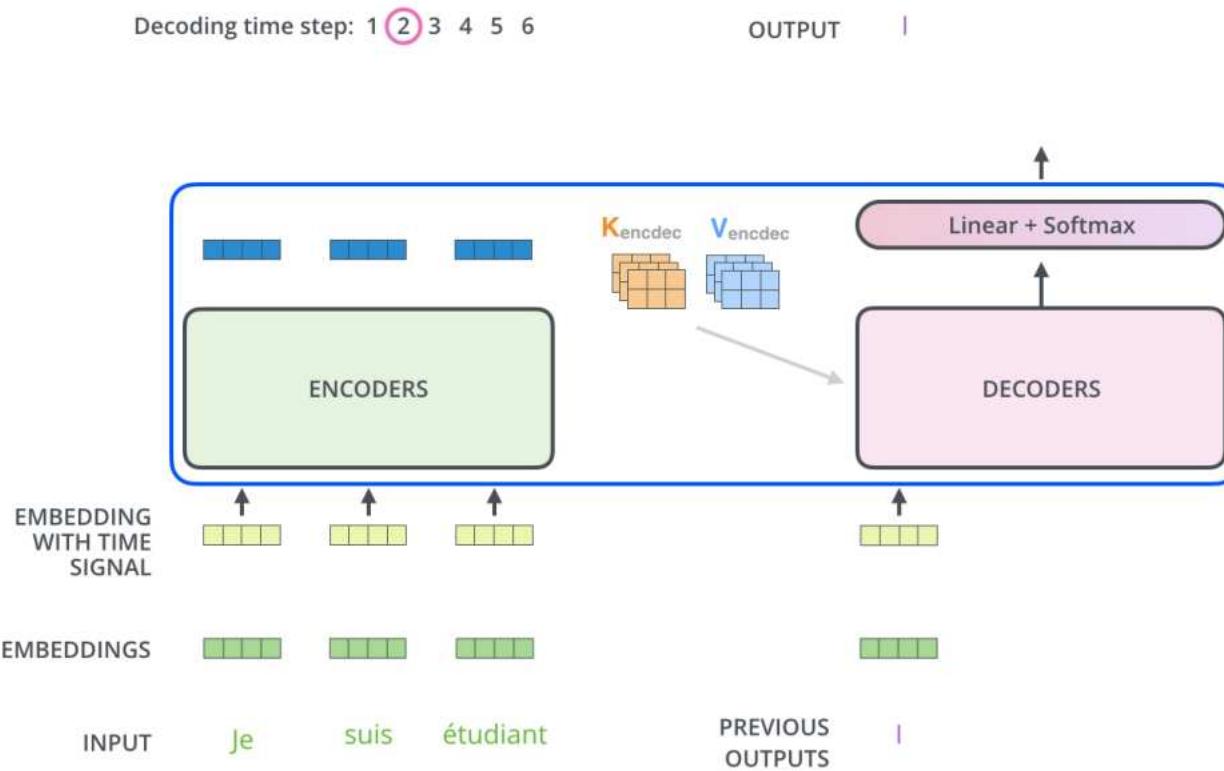


After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

Transformer Decoder

How encoder and decoder work together

- The output of each step is fed to the bottom decoder in the next time step
- Embed and add positional encoding to those decoder inputs. Process the inputs
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.



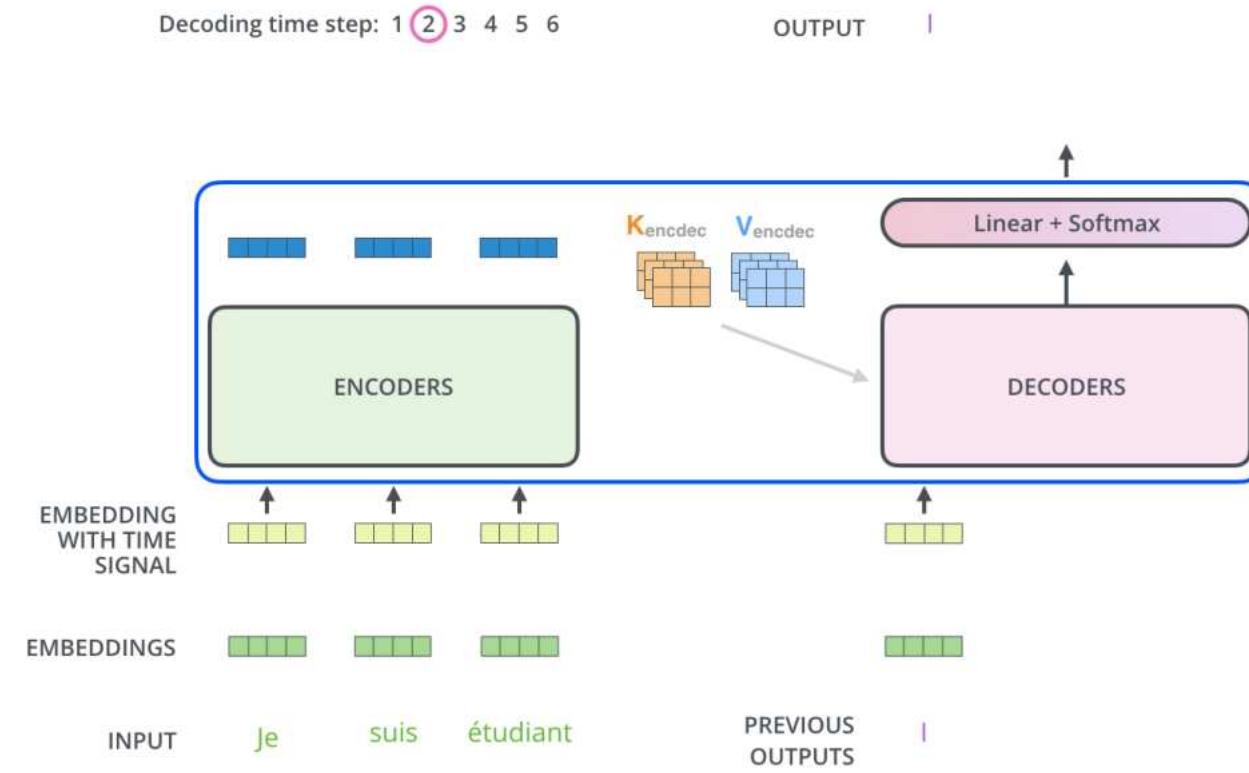
Transformer Decoder

How encoder and decoder work together

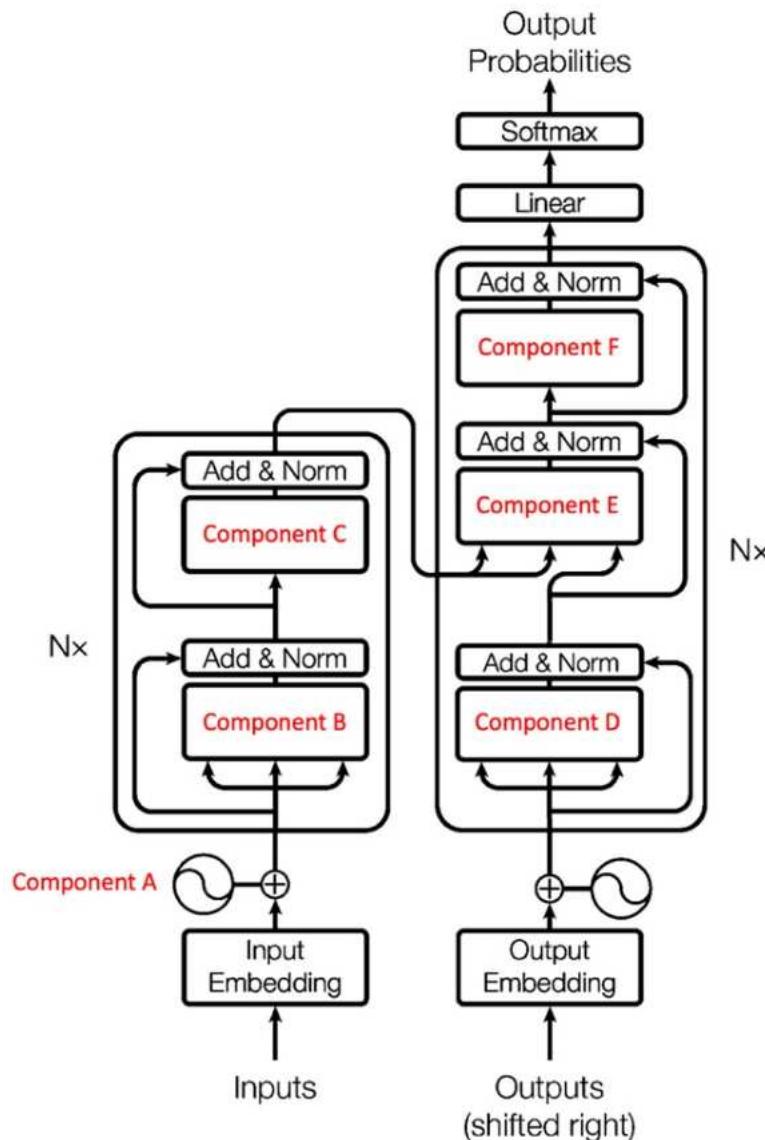
- The output of each step is fed to the bottom decoder in the next time step
- Embed and add positional encoding to those decoder inputs. Process the inputs
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.

Note:

In the decoder, *the self-attention layer is only allowed to attend to earlier positions in the output sequence*. This is done by **masking future positions** (setting them to -inf) before the softmax step in the self-attention calculation.



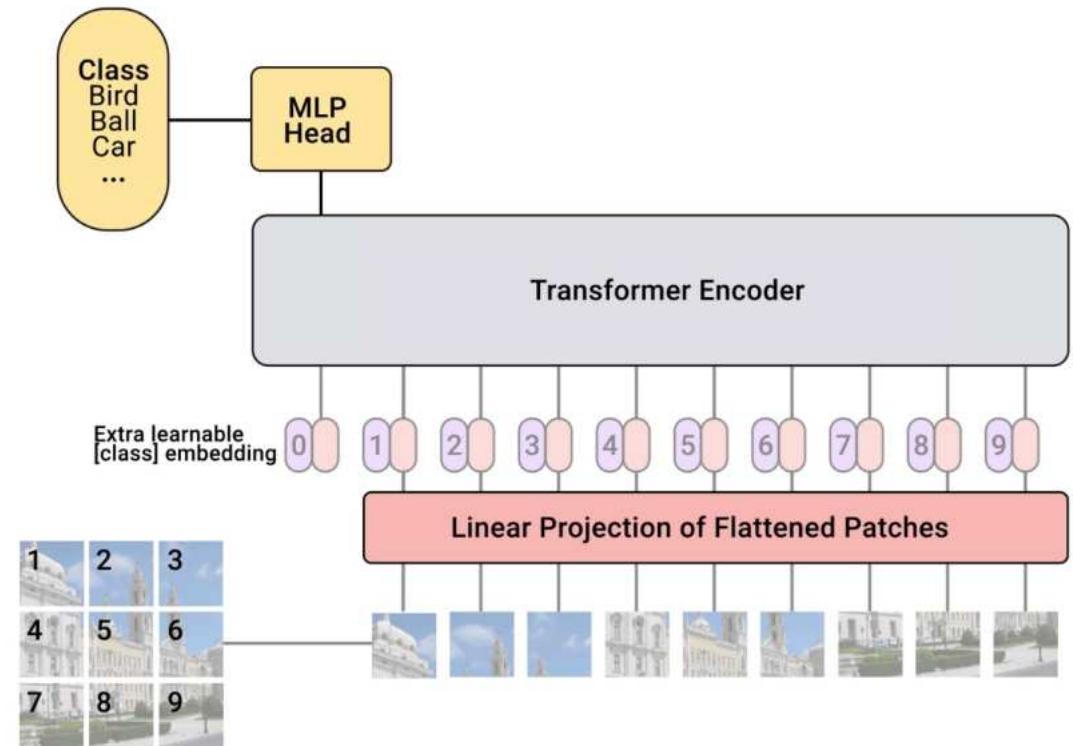
Transformer



Vision Transformer

Vision Transformer (ViT)

- Do not have decoder
- Reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
 - (H, W) is the resolution of the original image
 - C is the number of channels
 - (P, P) is the resolution of each image patch
 - $N = HW/P^2$ is the resulting number of patches



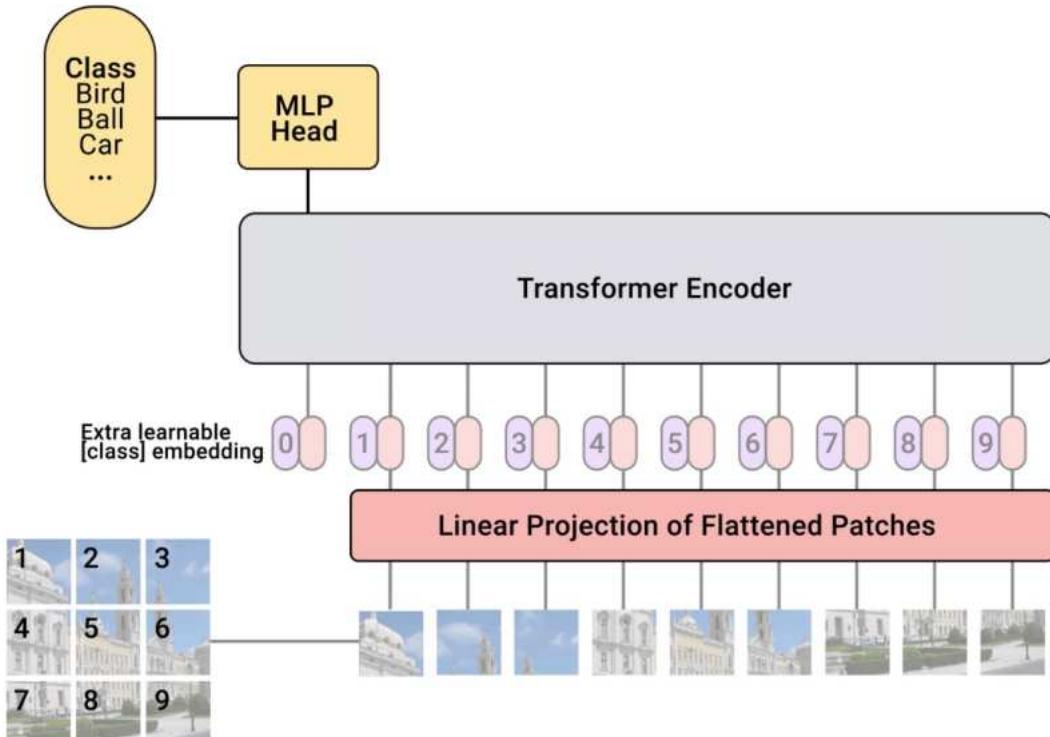
Vision Transformer (ViT)

Prepend a **learnable embedding** ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1 \mathbf{E}; \mathbf{x}_p^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{pos}$$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

Add **learnable position embeddings** $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information



Vision Transformer (ViT)

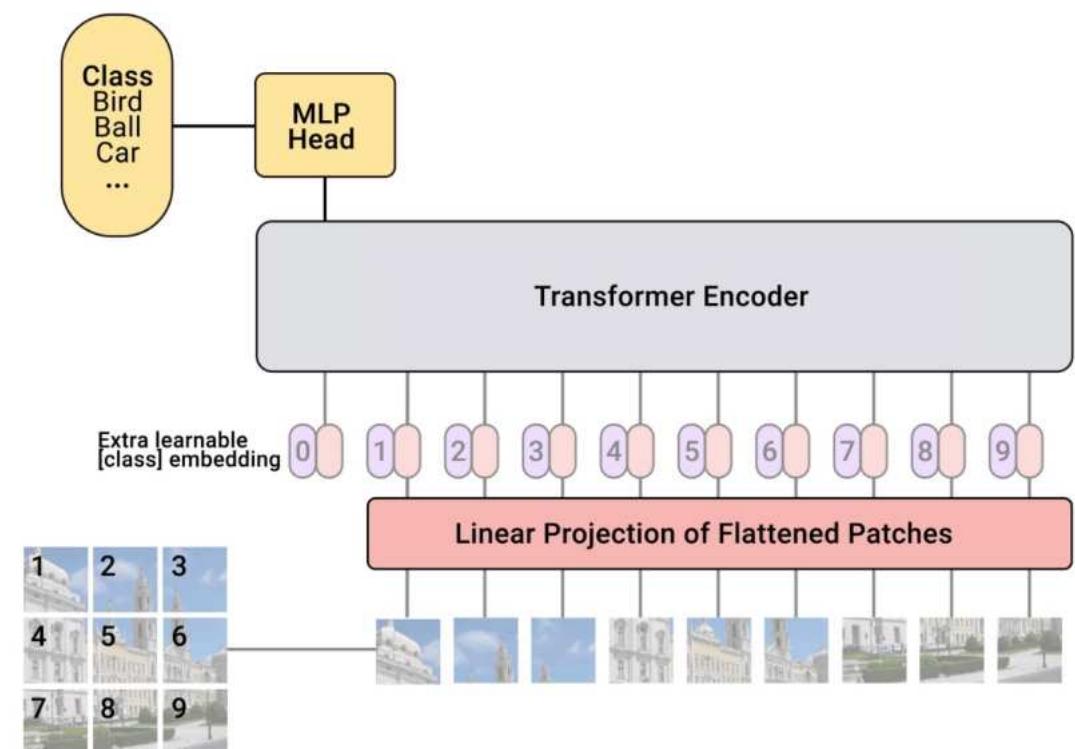
Feed the resulting sequence of vectors \mathbf{z}_0 to a standard Transformer encoder

Transformer Encoder $\left\{ \begin{array}{l} \mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1 \mathbf{E}; \mathbf{x}_p^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{\text{pos}}, \quad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D} \\ \mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \quad \ell = 1 \dots L \\ \mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \quad \ell = 1 \dots L \\ \mathbf{y} = \text{LN}(\mathbf{z}_L^0) \end{array} \right.$

Classification Head



The output of the additional [class] token is transformed into a class prediction via a small multi-layer perceptron (MLP) with tanh as non-linearity in the single hidden layer.



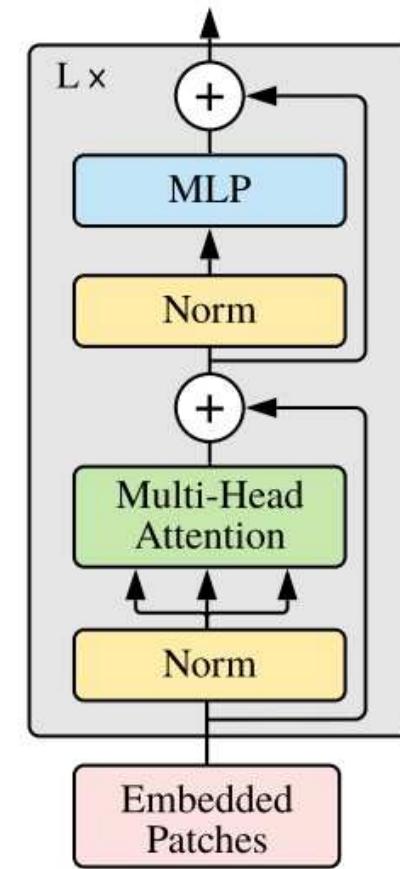
Vision Transformer (ViT)

Transformer Encoder

- Consists of a multi-head self-attention module (**MSA**), followed by a 2-layer MLP (with **GELU**)
- LayerNorm (LN) is applied before MSA module and MLP, and a residual connection is applied after each module.

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1},$$

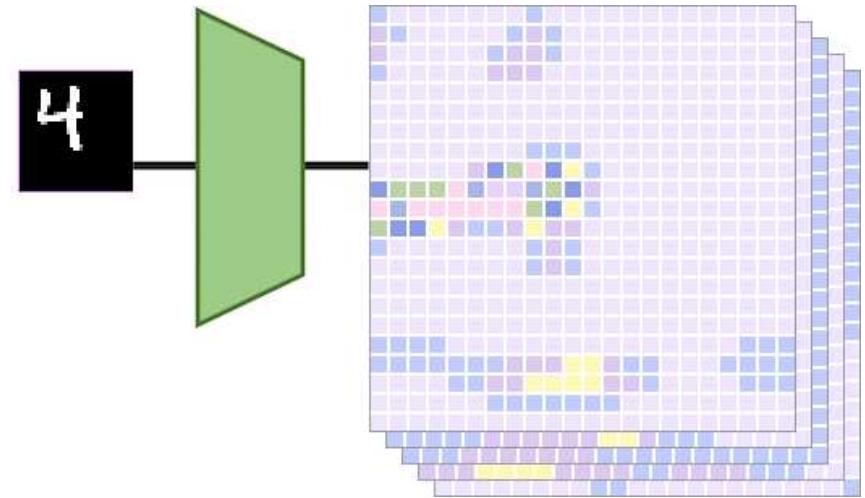
$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell,$$



Transformer Encoder

Inductive Bias

- ViT has much less image-specific inductive bias than CNNs
- Inductive bias in CNN
 - Locality
 - Two-dimensional neighborhood structure
 - Translation equivariance
- ViT
 - Only MLP layers are local and translationally equivariant. Self-attention layer is global
 - Two dimensional neighborhood is used sparingly – i) only at the beginning where we cut image into patches, ii) learnable position embedding (spatial relations have to be learned from scratch)



An equivariant mapping is a mapping which preserves the algebraic structure of a transformation.

A translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping

Vision Transformer (ViT)

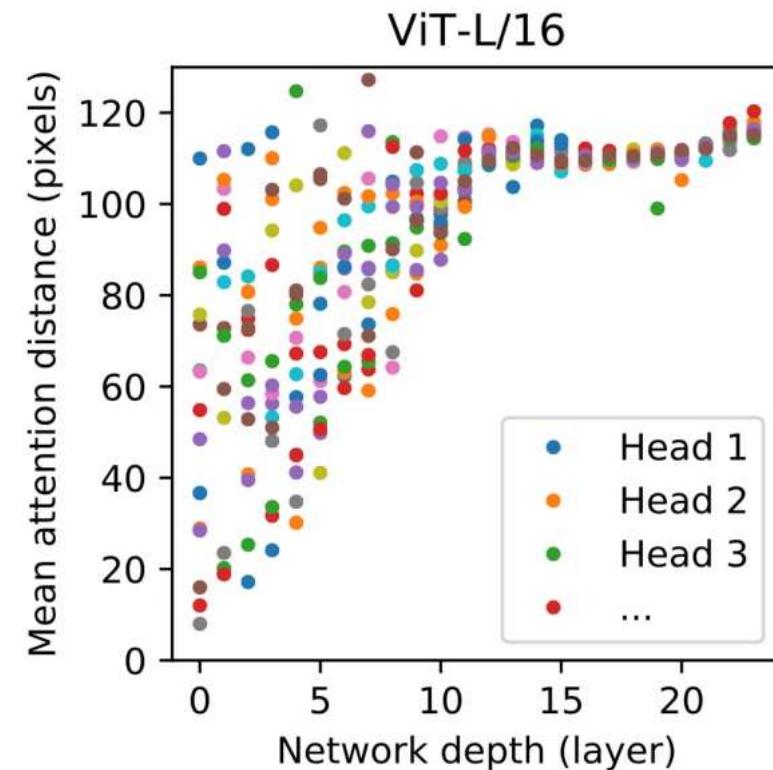
Examine the attention distance, analogous to receptive field in CNN

Compute the average distance in image space across which information is integrated, based on the attention weights.

Attention distance was computed for 128 example images by averaging the distance between the query pixel and all other pixels, weighted by the attention weight.

Each dot shows the mean attention distance across images for one of 16 heads at one layer. Image width is 224 pixels.

An example: if a pixel is 20 pixels away and the attention weight is 0.5 the distance is 10.

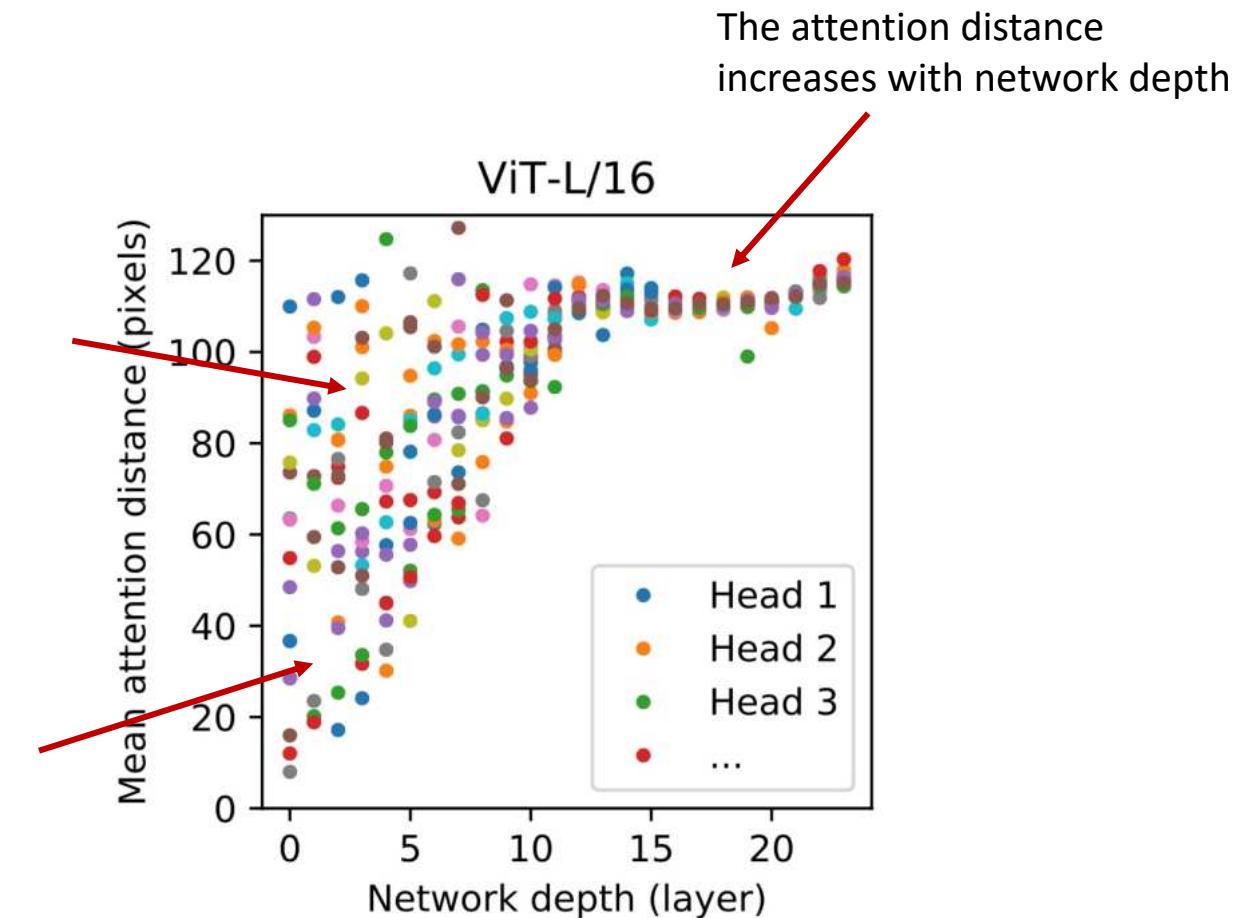


Vision Transformer (ViT)

Examine the attention distance, analogous to receptive field in CNN

Some heads attend to most of the image already in the lowest layers, showing the capability of ViT in integrating information globally

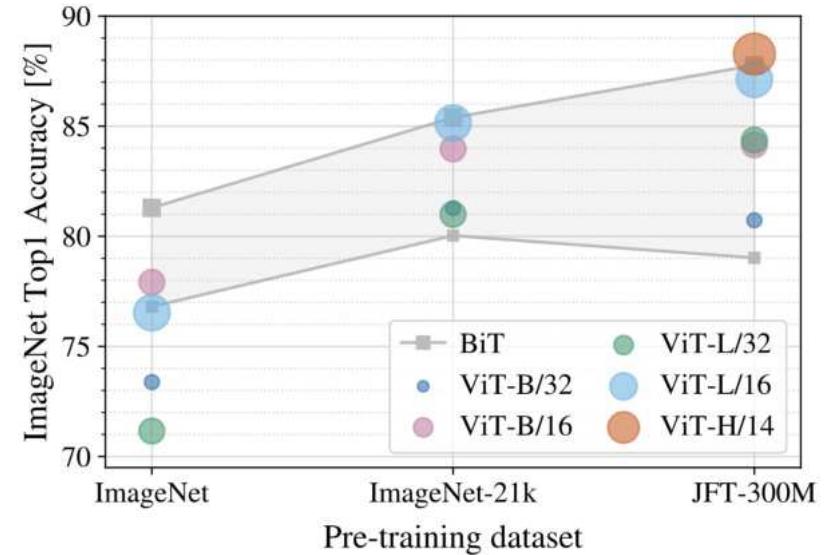
Other attention heads have consistently small attention distances in the low layers



Vision Transformer (ViT)

Performance of ViT

- ViT performs significantly worse than the CNN equivalent (BiT) when trained on ImageNet (1M images).
- However, on ImageNet-21k (14M images) performance is comparable, and on JFT (300M images), ViT outperforms BiT.
- ViT overfits the ImageNet task due to **its lack of inbuilt knowledge about images**



Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Vision Transformer (ViT)

ViT conducts global self-attention

- Relationships between a token and all other tokens are computed
- Quadratic complexity with respect to the number of tokens
- Not tractable for dense prediction or to represent a high-resolution image

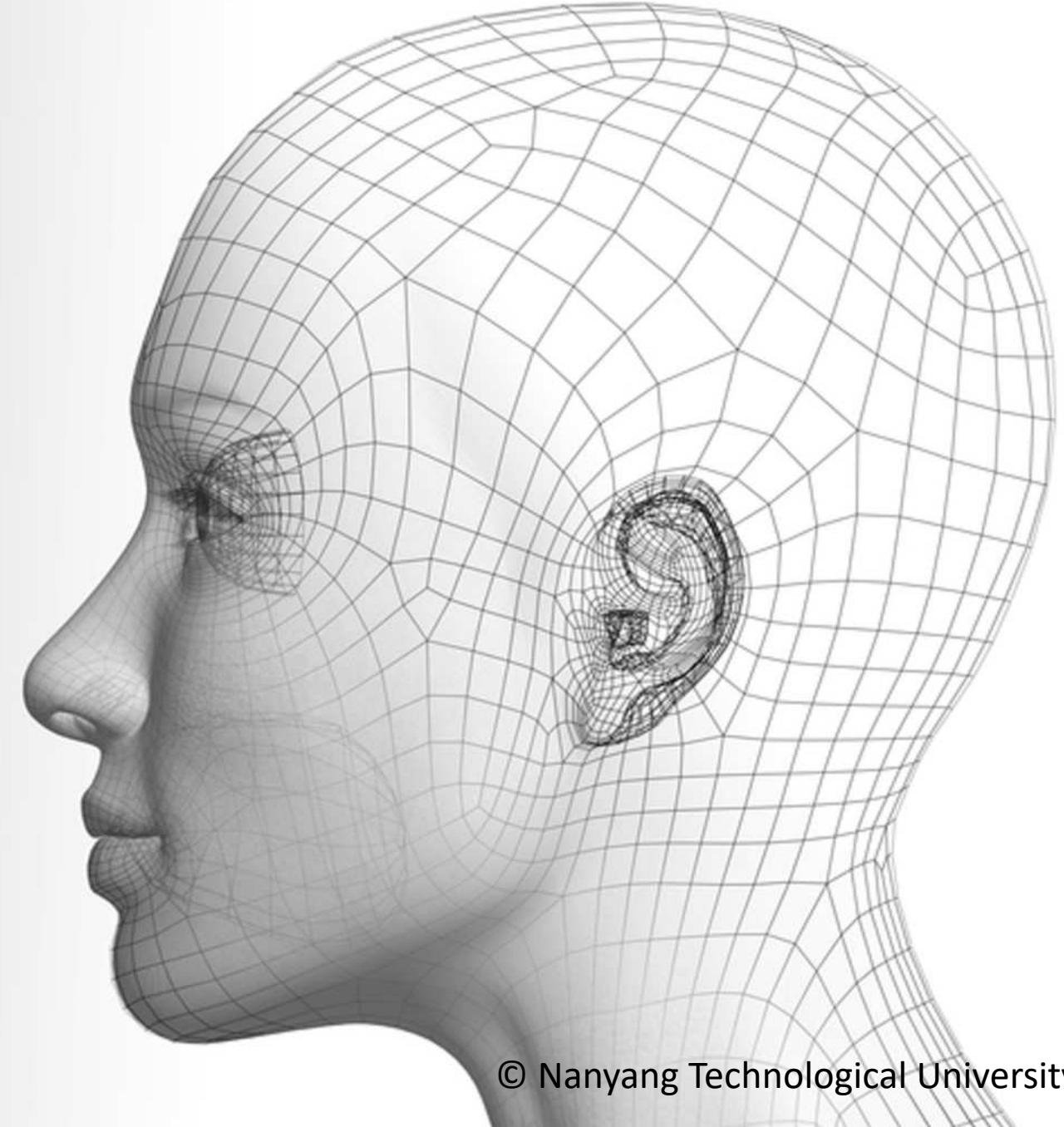
Autoencoders

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

<https://twitter.com/XingangP>



Outline

- Supervised vs unsupervised learning
- Autoencoders
 - Denoising autoencoders
 - Undercomplete and overcomplete autoencoders
 - Sparse autoencoders
 - Other encoders

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples: Classification, regression,
object detection, semantic
segmentation, image captioning, etc.

Classification



Cat

Supervised vs Unsupervised Learning

Supervised Learning

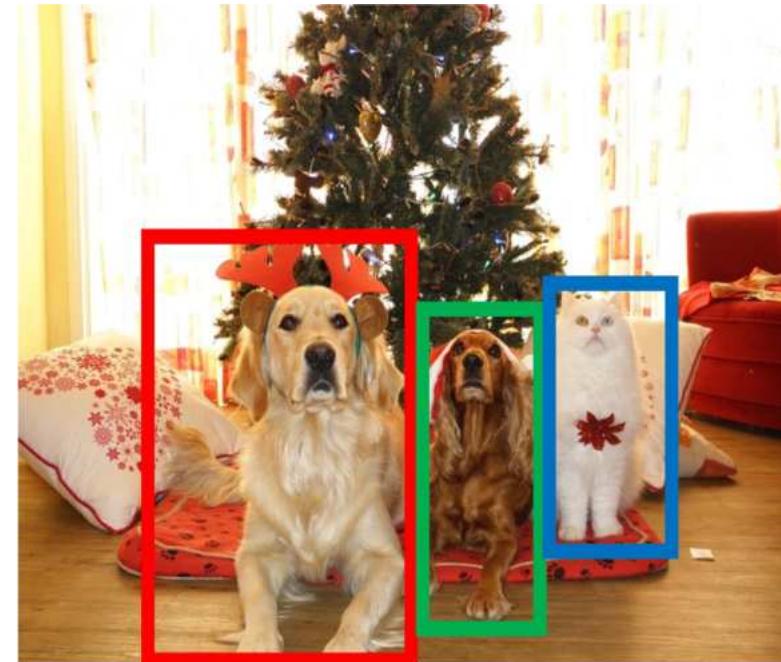
Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples: Classification, regression,
object detection, semantic
segmentation, image captioning, etc.

Object Detection



DOG, DOG, CAT

[This image is CC0 public domain](#)

Supervised vs Unsupervised Learning

Supervised Learning

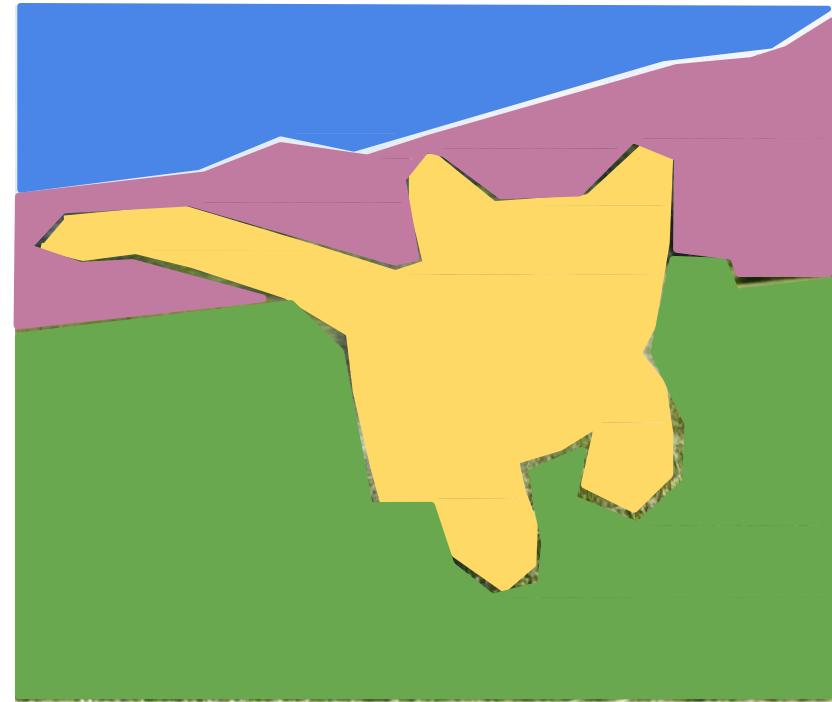
Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples: Classification, regression,
object detection, semantic
segmentation, image captioning, etc.

Semantic Segmentation



GRASS, CAT, TREE, SKY

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Image captioning



A cat sitting on a suitcase on the floor

Supervised vs Unsupervised Learning

Supervised Learning

Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples: Classification, regression,
object detection, semantic
segmentation, image captioning, etc.

Unsupervised Learning

Data: x

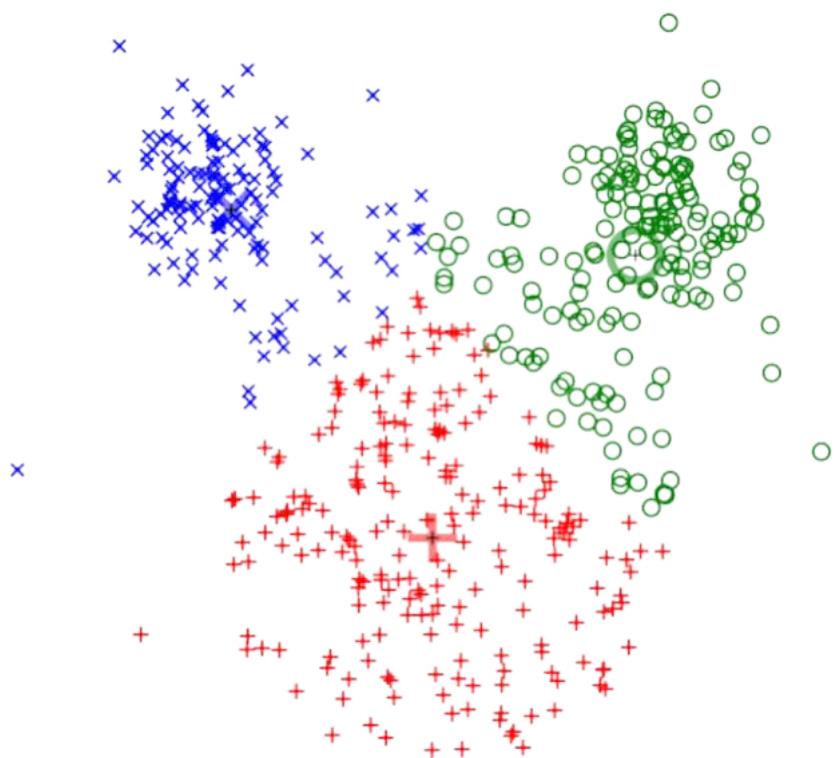
Just data, no labels!

Goal: Learn some underlying
hidden *structure* of the data

Examples: Clustering,
dimensionality reduction, feature
learning, density estimation, etc.

Supervised vs Unsupervised Learning

Clustering (e.g. K-Means)



Unsupervised Learning

Data: x

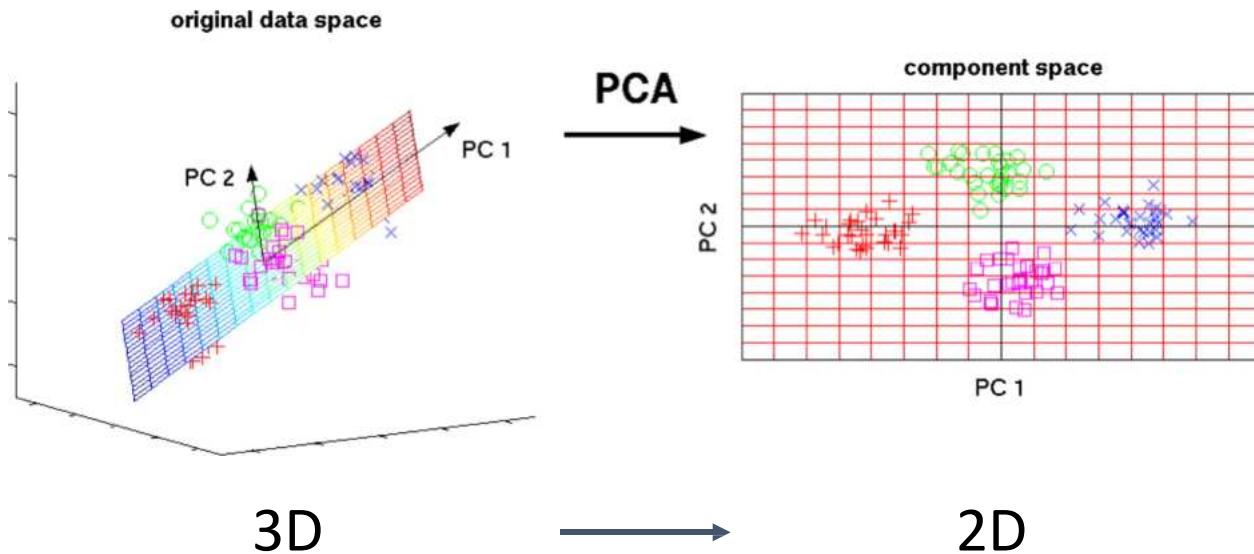
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Dimensionality Reduction
(e.g. Principal Components Analysis)



Unsupervised Learning

Data: x

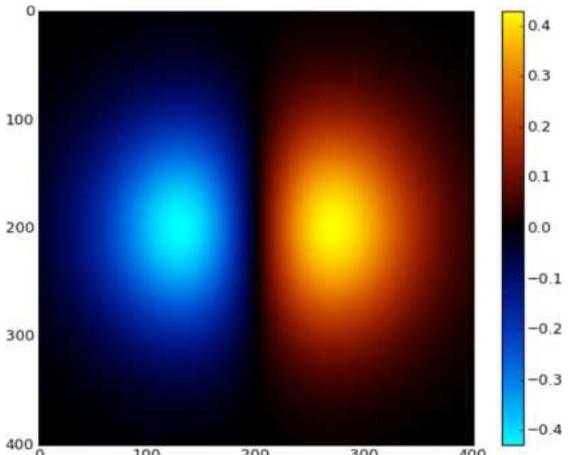
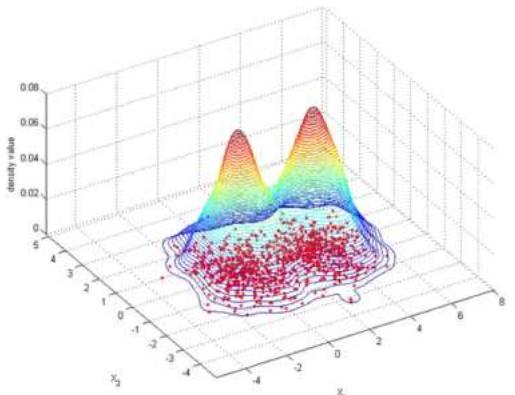
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Density Estimation



Unsupervised Learning

Data: x

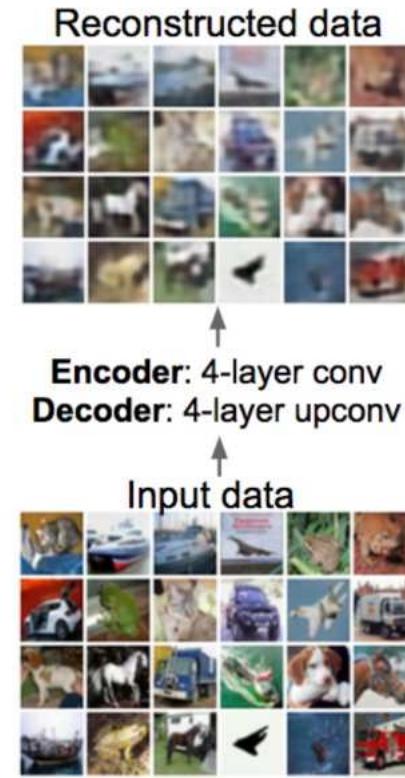
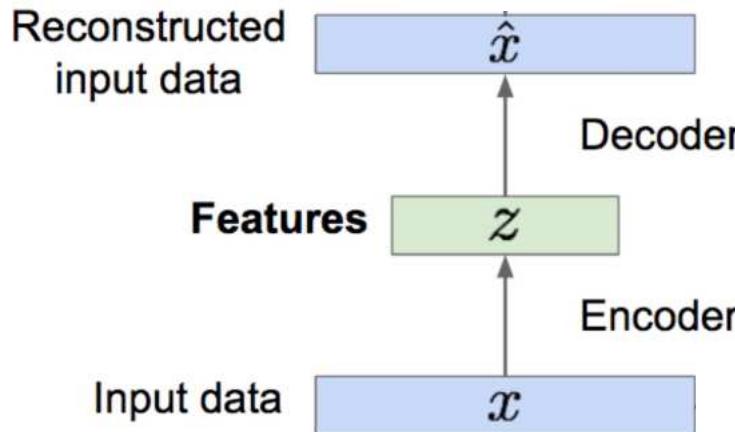
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Supervised vs Unsupervised Learning

Feature Learning (e.g. autoencoders)



Unsupervised Learning

Data: x

Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

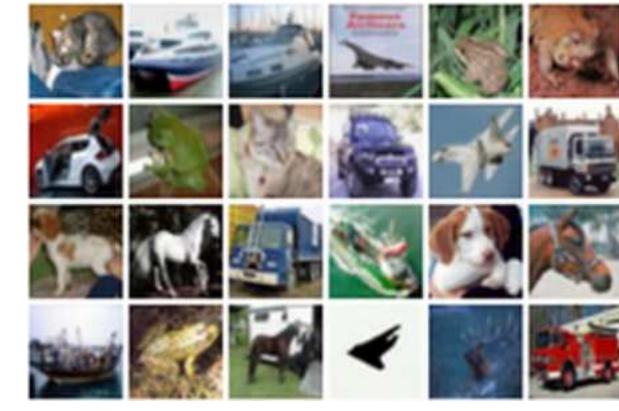
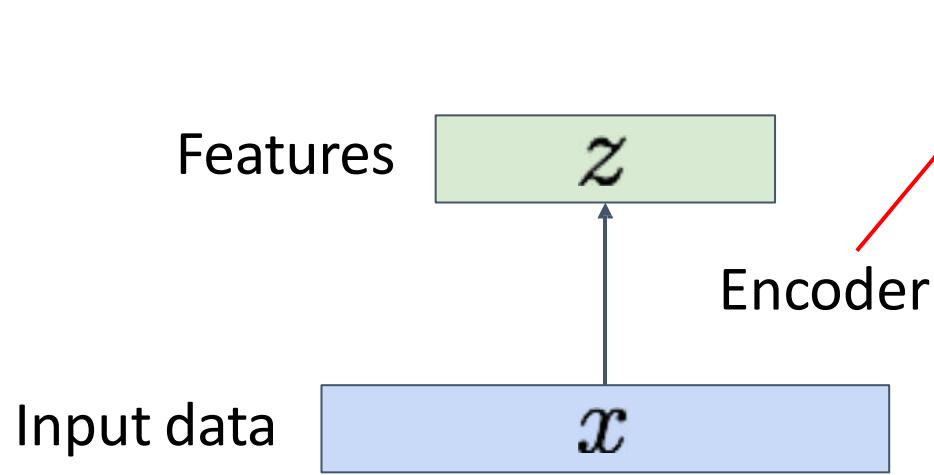
Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Autoencoders

Unsupervised method for learning feature vectors from raw data x , without any labels

Features should extract useful information (maybe object identities, properties, scene type, etc) that we can use for downstream tasks

Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN

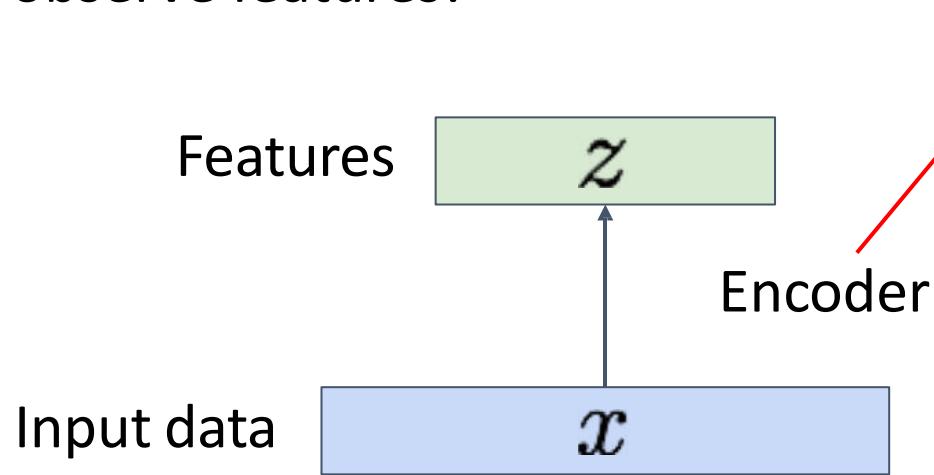


Autoencoders

Problem: How can we learn this feature transform from raw data?

Features should extract useful information (maybe object identities, properties, scene type, etc) that we can use for downstream tasks
But we can't observe features!

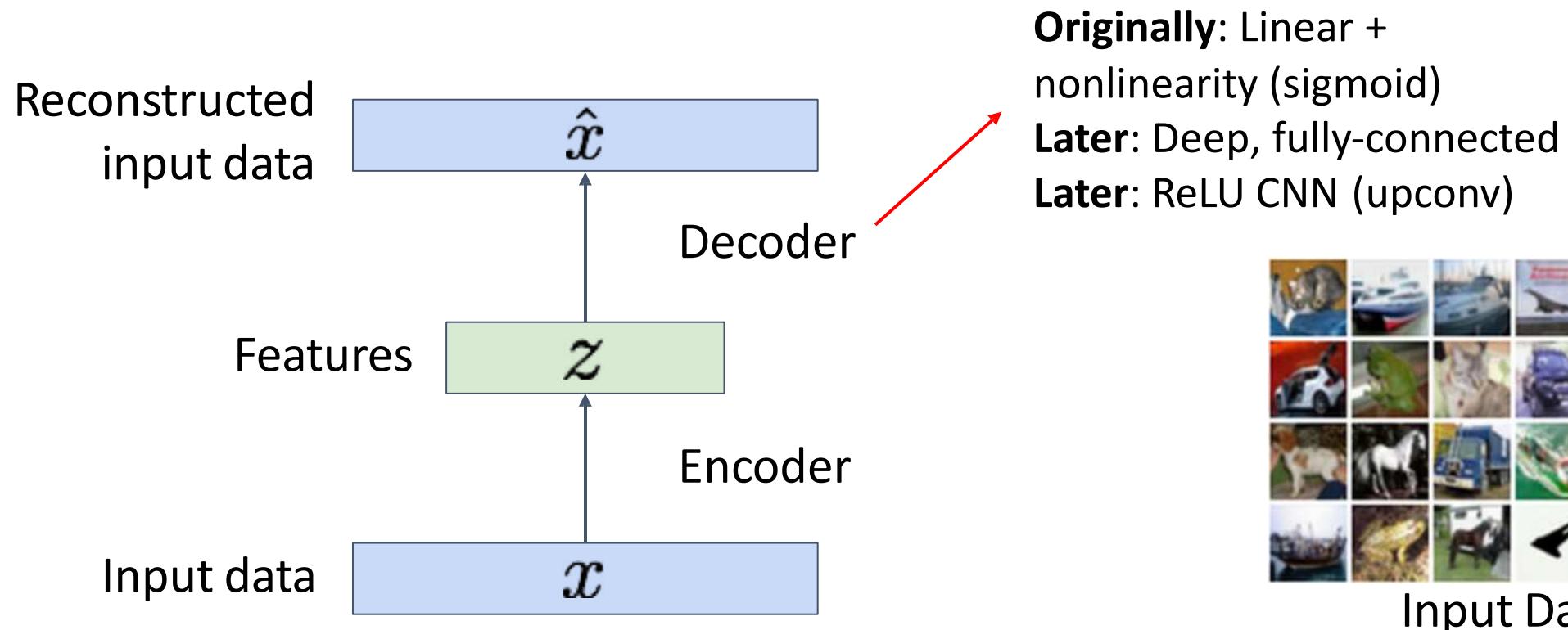
Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN



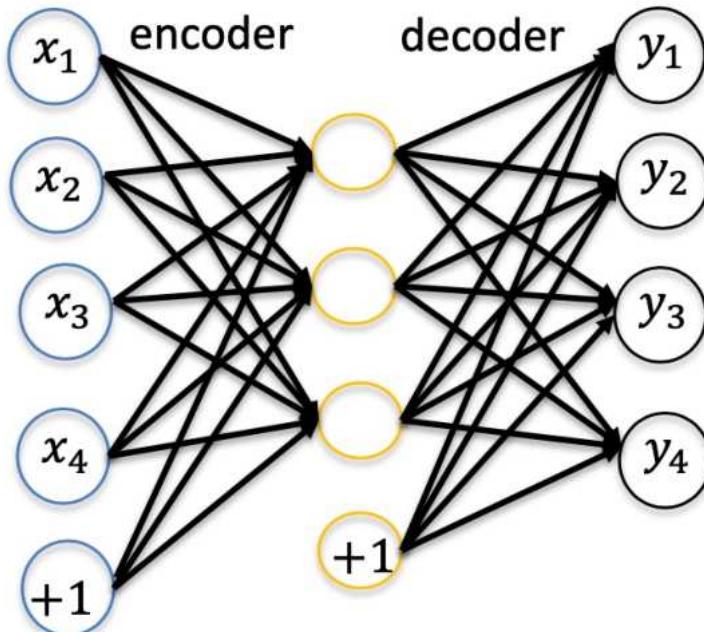
Autoencoders

- **Problem:** How can we learn this feature transform from raw data?

Idea: Use the features to reconstruct the input data with a **decoder**
“Autoencoding” = encoding itself



Autoencoders

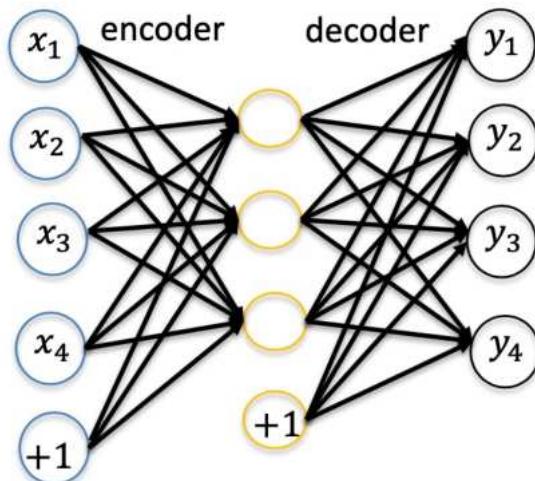


An autoencoder is a neural network that is trained to attempt to copy its input to its output. Its hidden layer describes a *code* that represents the input.

The network consists of two parts: an **encoder** and a **decoder**.

Autoencoders

Given an input x , the hidden-layer performs the encoding function $\mathbf{h} = \emptyset(x)$ and the decoder φ produces the reconstruction $y = \varphi(\mathbf{h})$.



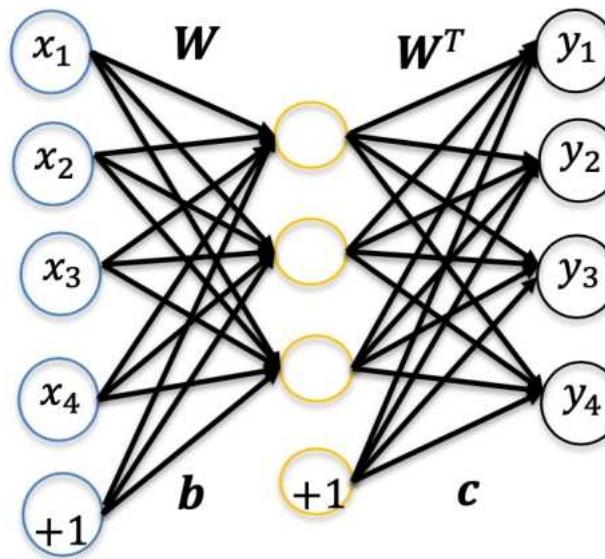
If the autoencoder succeeds:

$$\mathbf{y} = \varphi(\mathbf{h}) = \varphi(\emptyset(x)) = \mathbf{x}$$

In order to be useful, autoencoders are designed to be **unable to copy exactly** and enable to copy only inputs that resembles the training data.

Since the model is forced to prioritize which aspects of the input should be copied, the hidden-layer often **learns useful properties of the data**.

Autoencoders



Reverse mapping from the hidden layer to the output can be optionally constrained to be the same as the input to hidden-layer mapping. That is, if encoder weight matrix is \mathbf{W} , the decoder weigh matrix is \mathbf{W}^T .

Hidden layer and output layer activation can be then written as

$$\begin{aligned}\mathbf{h} &= f(\mathbf{W}^T \mathbf{x} + \mathbf{b}) \\ \mathbf{y} &= f(\mathbf{W}\mathbf{h} + \mathbf{c})\end{aligned}$$

f is usually a sigmoid.

Training autoencoders

The **cost function of reconstruction** can be measured by many ways, depending on the appropriate distributional assumptions on the inputs.

Learning of autoencoders is **unsupervised** as no specific targets are given.

Given a training set $\{\mathbf{x}_p\}_{p=1}^P$.

The mean-square-error cost is usually used if the data is assumed to be continuous and Gaussian distributed:

$$J_{mse} = \frac{1}{P} \sum_{p=1}^P \|\mathbf{y}_p - \mathbf{x}_p\|^2$$

where \mathbf{y}_p is the output for input \mathbf{x}_p and $\|\cdot\|$ denotes the magnitude of the vector.

Training autoencoders

If the inputs are interpreted as **bit vectors** or **vectors of bit probabilities**, **cross-entropy of the reconstruction** can be used:

$$J_{\text{cross-entropy}} = - \sum_{p=1}^P (x_p \log y_p + (1 - x_p) \log(1 - y_p))$$

Learning are done by using gradient descent:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J$$

$$\mathbf{c} \leftarrow \mathbf{c} - \alpha \nabla_{\mathbf{c}} J$$

Example 1

Given input patterns:

$(1, 0, 1, 0, 0)$, $(0, 0, 1, 1, 0)$, $(1, 1, 0, 1, 1)$ and $(0, 1, 1, 1, 0)$

Design an autoencoder with 3 hidden units.

Show the representations of inputs at the hidden layer upon convergence.

$$\mathbf{H} = f(\mathbf{XW} + \mathbf{B})$$

$$\mathbf{Y} = f(\mathbf{HW}^T + \mathbf{C})$$

$$\mathbf{O} = \mathbf{1}(\mathbf{Y} > 0.5)$$

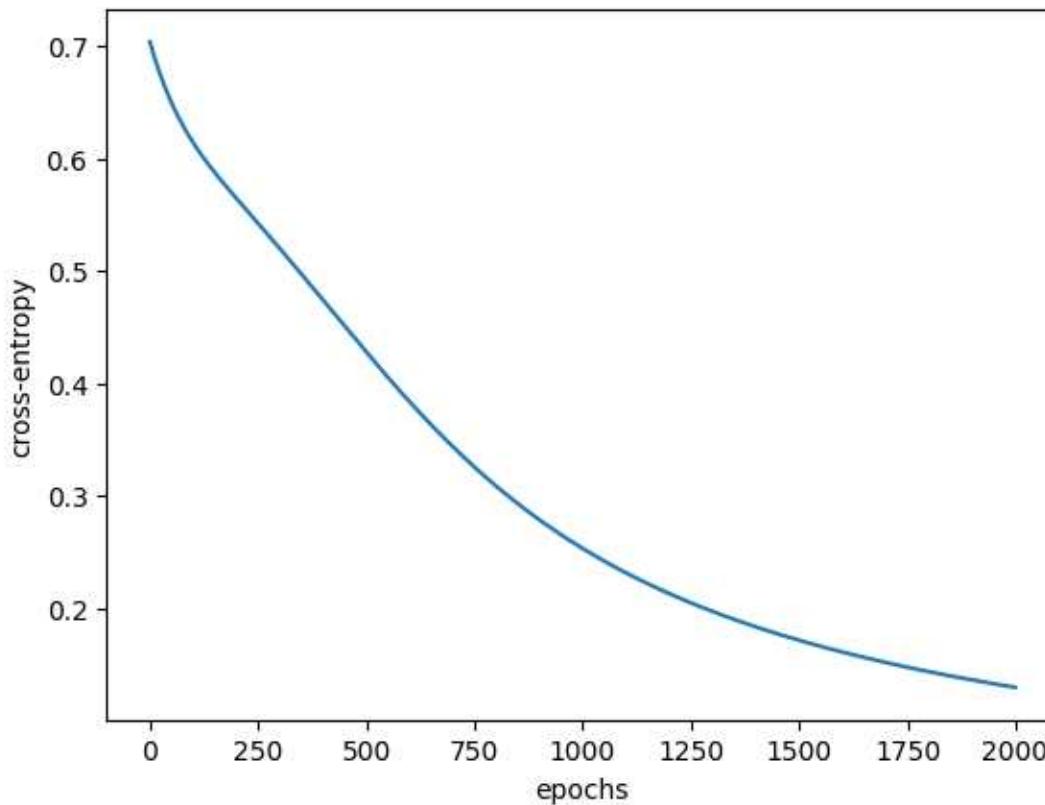
Gradient descent on entropy:

$$J = - \sum_{p=1}^4 (x_p \log y_p + (1 - x_p) \log(1 - y_p))$$

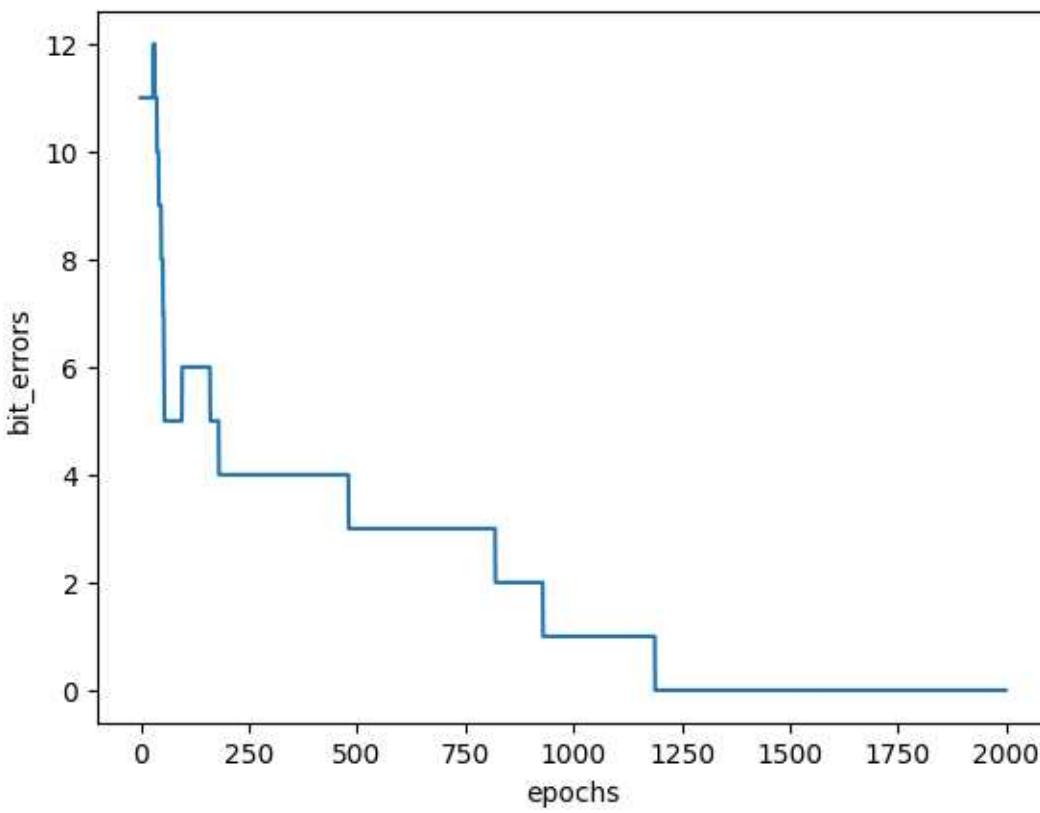
$$bit\ errors = \sum_{p=1}^4 \sum_{k=1}^5 1(x_{pk} \neq o_{pk})$$

eg10.1.ipynb

Example 1



Example 1



Example 1

```
▶ # Display weights and biases
print(f'W:\n {autoencoder.W.data}\n')
print(f'b:\n {autoencoder.b.data}\n')
print(f'b_prime:\n {autoencoder.b_prime.data}\n')

→ W:
    tensor([[-3.6867, -0.3808,  2.6520],
           [-0.2545, -3.6158, -1.8657],
           [ 1.7148,  3.0138,  0.9254],
           [ 1.9011, -0.9880, -3.5277],
           [-2.4830, -2.5980, -1.1112]])

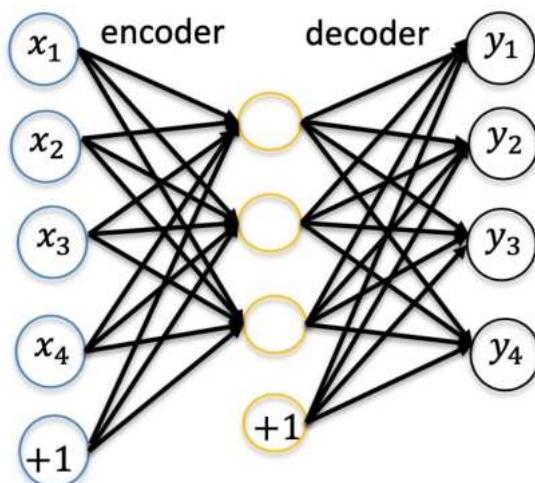
b:
    tensor([ 0.1496,  0.5881, -0.1604])

b_prime:
    tensor([ 1.2448,  2.3414, -0.7427,  2.1173,  0.9872])
```

Denoising Autoencoders (DAE)

A *denoising autoencoder* (DAE) receives corrupted data points as inputs.

It is trained to predict the original uncorrupted data points as its output.



The idea of DAE is that in order to force the hidden layer to **discover more robust features** and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to **undo the effect of corruption process** applied to the input of the autoencoder.

DAE

For training DAE:

- First, input data is corrupted to mimic the noise in the images: $x \rightarrow \tilde{x}$

\tilde{x} is the corrupted version of data. The corruption process simulates the distribution of data

- The network is trained to produce uncorrupted data: $y \rightarrow x$

DAE: Corrupting inputs

To obtain corrupted version of input data, each input x_i of input data is added with **additive or multiplicative noise**.

Additive noise:

$$\tilde{x}_i = x_i + \varepsilon$$

where noise ε is Gaussian distributed: $\varepsilon \sim N(0, \sigma^2)$

And σ is the standard deviation that determines the S/N ratio. Usually used for continuous data.

Multiplicative noise:

$$\tilde{x}_i = \varepsilon x_i$$

where noise ε could be Binomially distributed: $\varepsilon \sim Binomial(p)$

And p is the probability of ones and $1 - p$ is the probability of zeros (noise). Usually, used for binary data.

Example 2: Denoising autoencoders

The MNIST database of gray level images of handwritten digits:
<http://yann.lecun.com/exdb/mnist/>



Each image is 28x28 size.

Intensities are in the range [0, 255] and normalized to [0, 1].

Training set: 60,000, Test set: 10,000

To build a DAE with 500 hidden units.

Example 2a: DAE (multiplicative noise)

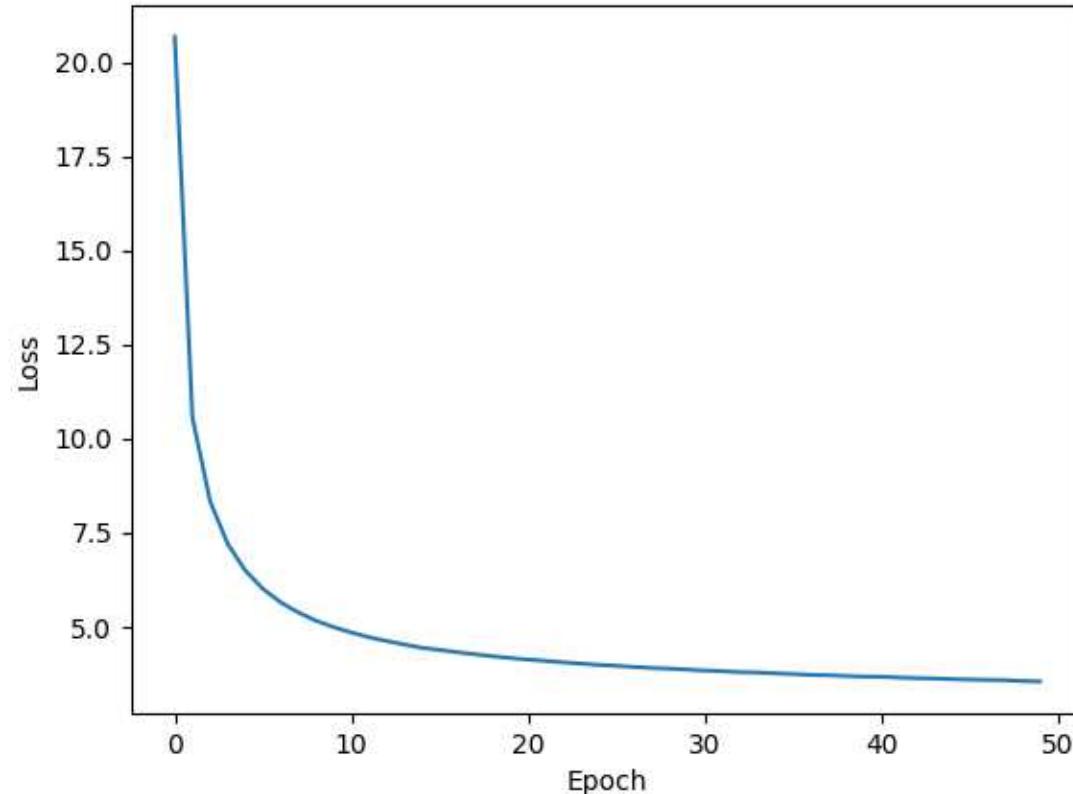
Original data						
7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Corrupted data						
7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

DAE attempts to predict the input data from corrupted input data with multiplicative noise with binomial distribution.
Corruption level $1 - p = 10\%$

eg10.2a.ipynb

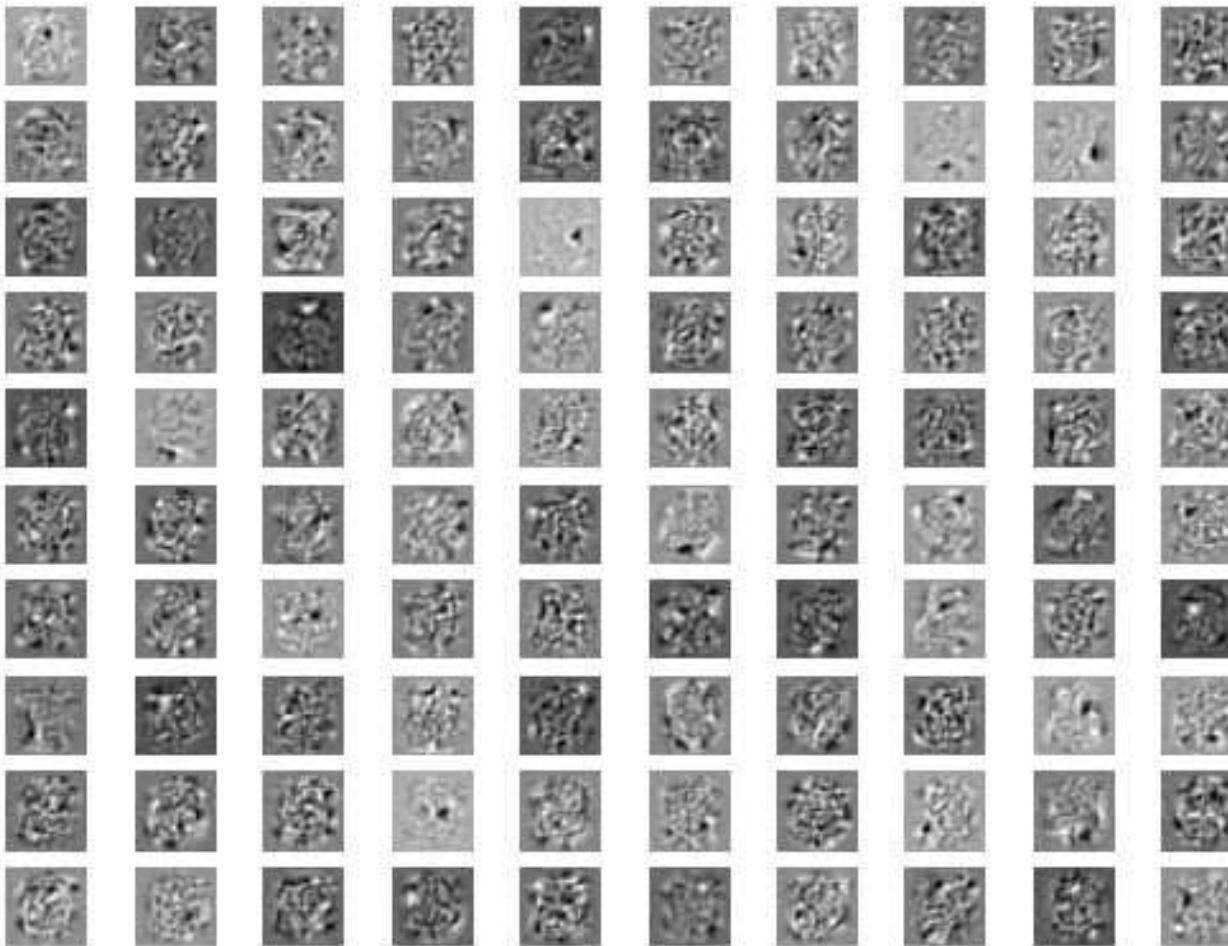
Example 2a: DAE (multiplicative noise)



500 hidden units
 $\alpha = 0.1$

Example 2a: DAE (multiplicative noise)

Weights learned by the hidden-layer



Example 2a: DAE (multiplicative noise)

Sample of reconstructed test images

Input data

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Reconstructed (noise-filtered) data

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Example 2b: DAE (additive noise)

Original data

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Corrupted data

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

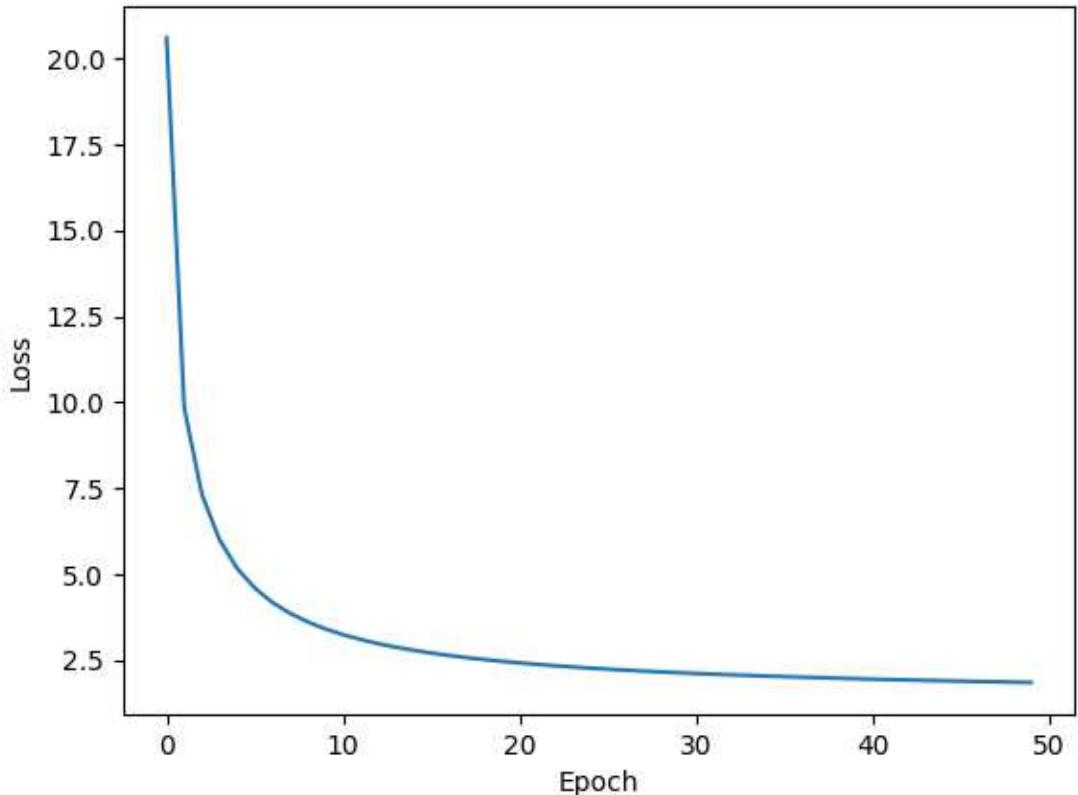


Additive noise with gaussian distribution.

Data is scaled between [0, 1], noise s.d = 0.1

eg10.2b.ipynb

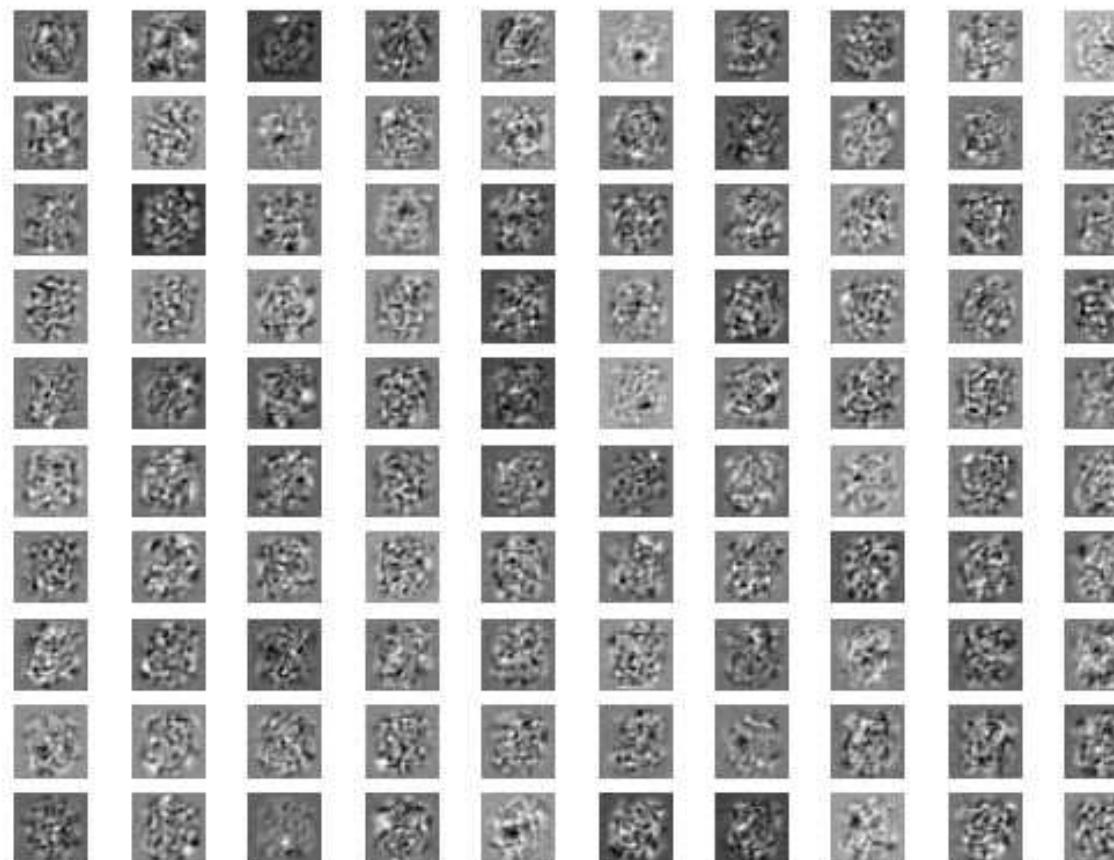
Example 2b: DAE (additive noise)



500 hidden units
 $\alpha = 0.1$

Example 2b: DAE (additive noise)

Features (weights) learned by the hidden-layer



Example 2b: DAE (additive noise)

Sample of reconstructed images

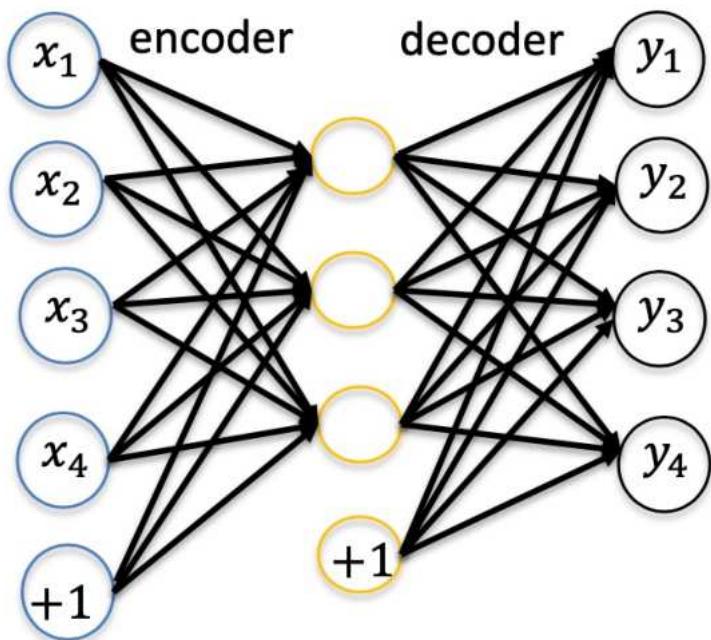
Input data

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	4	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Reconstructed (noise-filtered) data

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	4	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Autoencoders

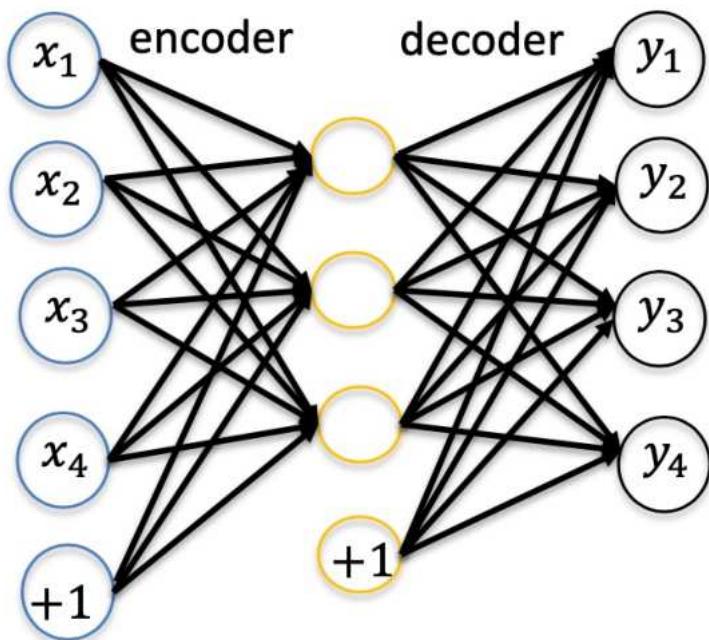


Input dimension n and hidden dimension M :

If $M < n$, *undercomplete* autoencoders

If $M > n$, *overcomplete* autoencoders

Undercomplete autoencoders



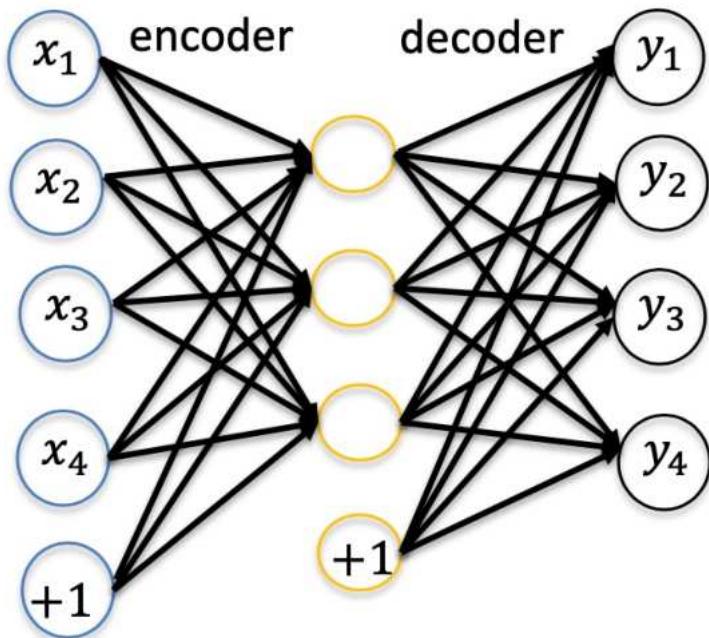
In undercomplete autoencoders, the hidden-layer has a **lower dimension** than the input layer.

By learning to approximate an n -dimensional inputs with M ($< n$) number of hidden units, we obtain a **lower dimensional representation** of the input signals. The network reconstructs the input signals from the reduced-dimensional hidden representation.

Learning an undercomplete representation **forces the autoencoder to capture the most salient features**.

By limiting the number of hidden neurons, **hidden structures** of input data can be inferred from autoencoders. For example, correlations among input variables, learning principal components of data, etc.

Overcomplete autoencoders

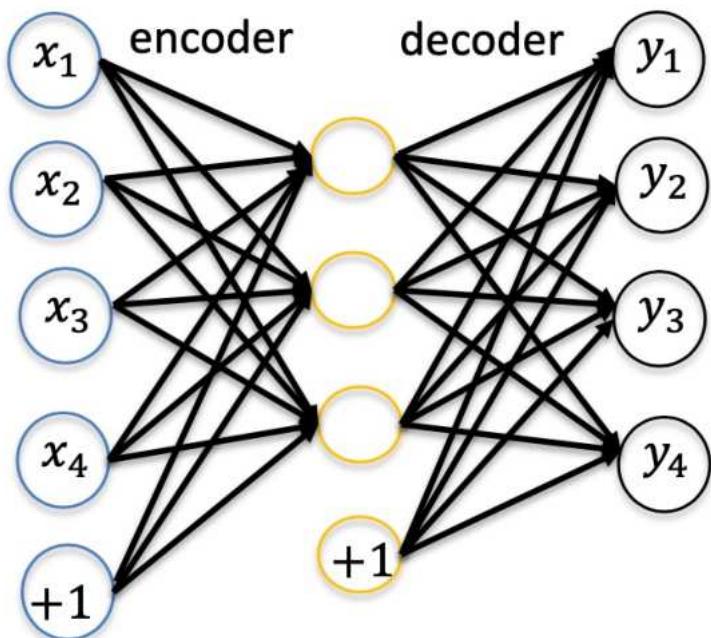


In overcomplete autoencoders, the hidden-layer has a **higher dimension** than the dimension of the input.

In order to learn useful information from overcomplete autoencoders, it is **necessary use some constraints** on its characteristics.

Even when the hidden dimensions are large, one can still explore interesting structures of inputs by introducing other constraints such as '**sparsity**' of input data.

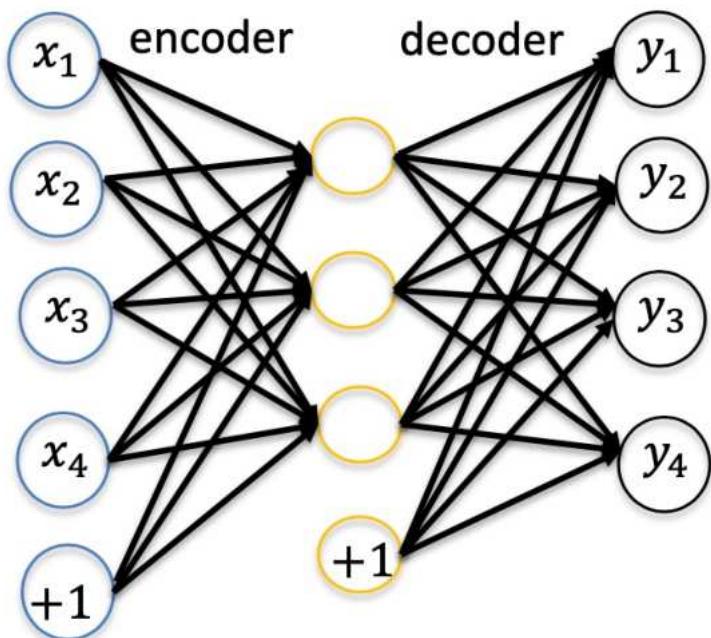
Regularizing autoencoders



Both undercomplete and overcomplete autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. Some form of constraints are needed in order to make them useful.

Deep autoencoders are only possible when some constraints are imposed on the cost function.

Regularizing autoencoders



Regularized autoencoders incorporate a penalty to the cost function to learn interesting features from the input.

Regularized autoencoders provide the ability to train any autoencoder architecture successfully by choosing suitable code dimension and the capacity of the encoder and decoder.

With regularized autoencoders, one can use **larger model capacity** (for example, deeper autoencoders) and large code size.

Regularizing autoencoders

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

$$J_1 = J + \beta \Omega(h)$$

where β is the penalty or regularization parameter. The penalty is usually **imposed on the hidden activations**.

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn trivial identity function.

The regularized loss function encourages the model to have other properties besides the ability to copy its input to its output.

Sparse Autoencoders (SAE)

A **sparse autoencoder** (SAE) is simply an autoencoder whose training criterion involves the **sparsity penalty** Ω_{sparsity} at the hidden layer:

$$J_1 = J + \beta \Omega_{\text{sparsity}}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be **sparse**.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be **inactive for most of the time**.

We say that the neuron is “active” when its output is close to 1 and the neuron is “inactive” when its output is close to 0.

Sparsity constraint

For a set $\{x_p\}_{p=1}^P$ of input patterns, the **average activation** ρ_j of neuron j at the hidden-layer is given by:

$$\rho_j = \frac{1}{P} \sum_{p=1}^P h_{pj} = \frac{1}{P} \sum_{p=1}^P f(\mathbf{x}_p^T \mathbf{w}_j + b_j)$$

where h_{pj} is the activation of hidden neuron j for p th pattern, and \mathbf{w}_j and b_j are the weights and biases of the hidden neuron j .

We would like to enforce the constraint: $\rho_j = \rho$ such that the **sparsity parameter** ρ is set to a **small value** close to zero (say 0.05).

That is, most of the time, hidden neuron activations are maintained at ρ on average. By choosing a smaller value for ρ , the neurons are **activated selectively to patterns and thereby learn sparse features**.

Sparsity constraint

To achieve sparse activations at the hidden-layer, the **Kullback-Leibler (KL) divergence** is used as the sparsity constraint:

$$D(\mathbf{h}) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the **deviation of the distribution** $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ .

The KL divergence is **minimum** when $\rho_j = \rho$ for all j .

That is, when the average activations are uniform and equal to very low value ρ .

Sparse Autoencoder (SAE)

The cost function for the sparse autoencoder (SAE) is given by

$$J_1 = J + \beta D(\mathbf{h})$$

where $D(h)$ is the KL divergence of hidden-layer activations:

$$D(\mathbf{h}) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

For gradient descent,

$$\nabla_{\mathbf{W}} J_1 = \nabla_{\mathbf{W}} J + \beta \nabla_{\mathbf{W}} D(\mathbf{h})$$

By chain rule:

$$\frac{\partial D(\mathbf{h})}{\partial \mathbf{w}_j} = \frac{\partial D(\mathbf{h})}{\partial \rho_j} \frac{\partial \rho_j}{\partial \mathbf{w}_j} \quad (\text{A})$$

where

$$D(\mathbf{h}) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

and

$$\begin{aligned} \frac{\partial D(\mathbf{h})}{\partial \rho_j} &= \rho \frac{1}{\rho} \left(-\frac{\rho}{\rho_j^2} \right) + (1 - \rho) \frac{1}{\left(\frac{1 - \rho}{1 - \rho_j} \right)} \left(\frac{-(1 - \rho)}{\left(\frac{1 - \rho}{1 - \rho_j} \right)^2} \right) \\ &= -\frac{\rho}{\rho_j} + \frac{1 - \rho}{1 - \rho_j} \end{aligned} \quad (\text{B})$$

SAE

Note:

$$\rho_j = \frac{1}{P} \sum_{p=1}^P f(u_{pj})$$

where $u_{pj} = \mathbf{x}_p^T \mathbf{w}_j + b_j$ is the synaptic input of the j th neuron due to p th pattern.

$$\frac{\partial \rho_j}{\partial \mathbf{w}_j} = \frac{1}{P} \sum_p f'((u_{pj})) \frac{\partial (u_{pj})}{\partial \mathbf{w}_j} = \frac{1}{P} \sum_p f'((u_{pj})) \mathbf{x}_p \quad (c)$$

Substituting (B) and (C) in (A):

$$\frac{\partial D(\mathbf{h})}{\partial \mathbf{w}_j} = \frac{1}{P} \left(\frac{\rho}{\rho_j} + \frac{1 - \rho}{1 - \rho_j} \right) \sum_p f'((u_{pj})) \mathbf{x}_p$$

Substituting in (A) , we can find:

$$\nabla_{\mathbf{W}} D(\mathbf{h}) = (\nabla_{w_1} D(\mathbf{h}) \quad \nabla_{w_2} D(\mathbf{h}) \quad \cdots \quad \nabla_{w_M} D(\mathbf{h}))$$

The gradient of the constrained cost function:

$$\nabla_{\mathbf{W}} J_1 = \nabla_{\mathbf{W}} J + \beta \nabla_{\mathbf{W}} D(\mathbf{h})$$

Similarly, $\nabla_{\mathbf{b}} J_1$ and $\nabla_{\mathbf{c}} J_1$ can be derived.

Learning can be done as

$$\begin{aligned}\mathbf{W} &\leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} J_1 \\ \mathbf{b} &\leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} J_1 \\ \mathbf{c} &\leftarrow \mathbf{c} - \alpha \nabla_{\mathbf{c}} J_1\end{aligned}$$

Example 3

Design the following autoencoder to process MNIST images:

1. Undercomplete autoencoder with 100 hidden units
2. Overcomplete autoencoder with 900 hidden units
3. Sparse autoencoder with 900 hidden units. Use sparsity parameter = 0.5

The MNIST database of gray level images of handwritten digits:

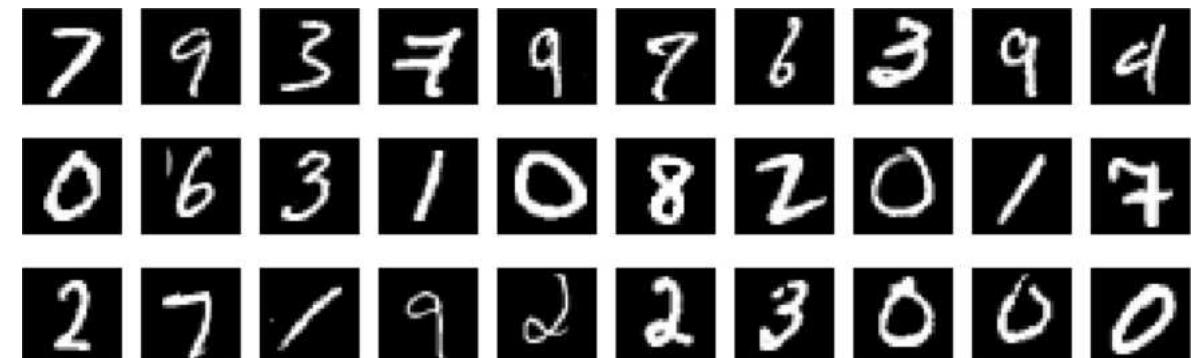
<http://yann.lecun.com/exdb/mnist/>

Each image is 28x28 size.

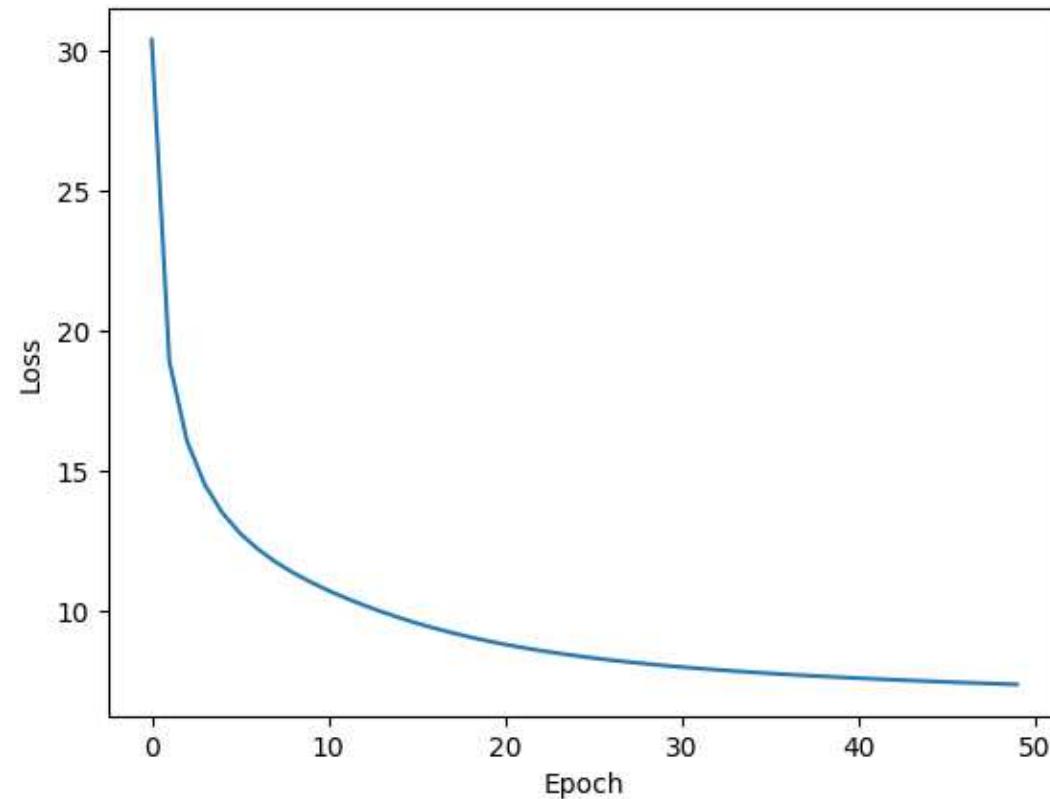
Intensities are in the range [0, 255].

Training set: 60,000

Test set: 10,000

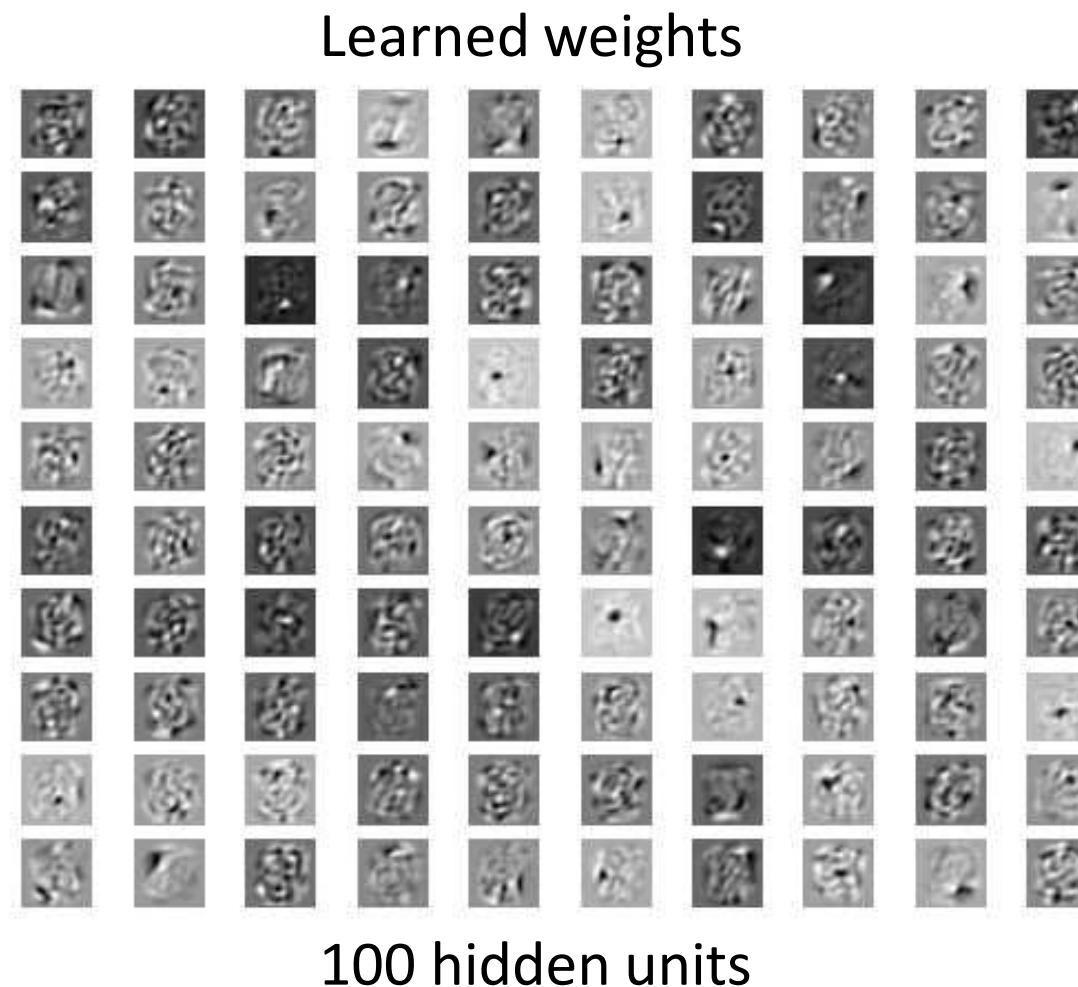


Example 3a: Undercomplete autoencoder



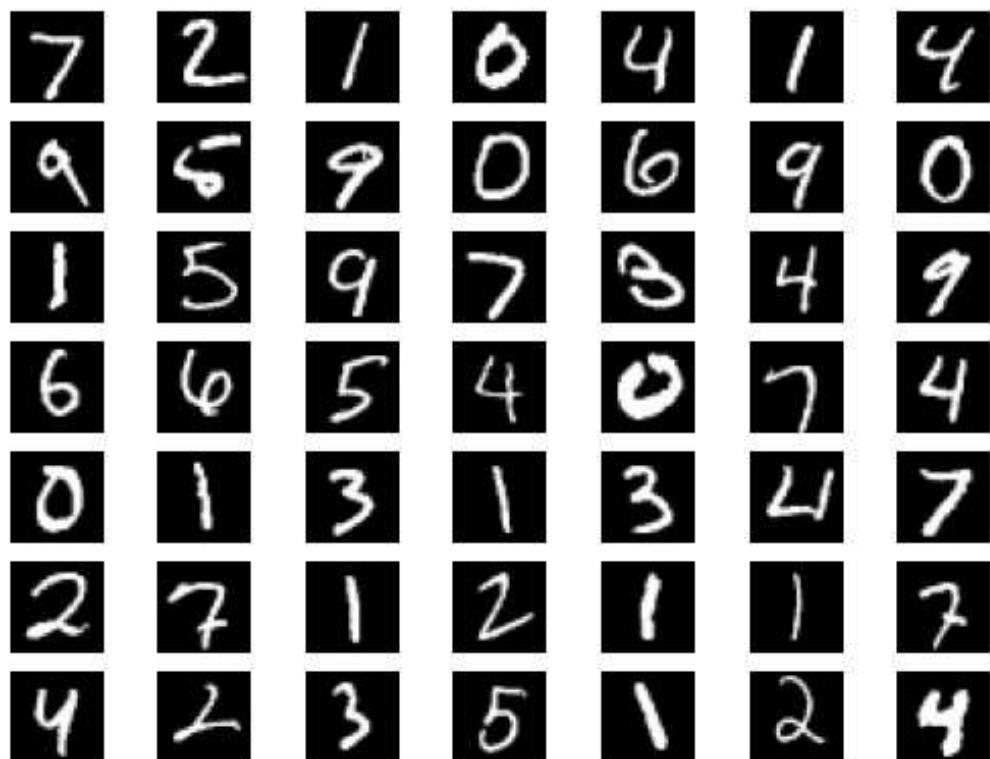
eg10.3a.ipynb

Example 3a: Undercomplete autoencoder

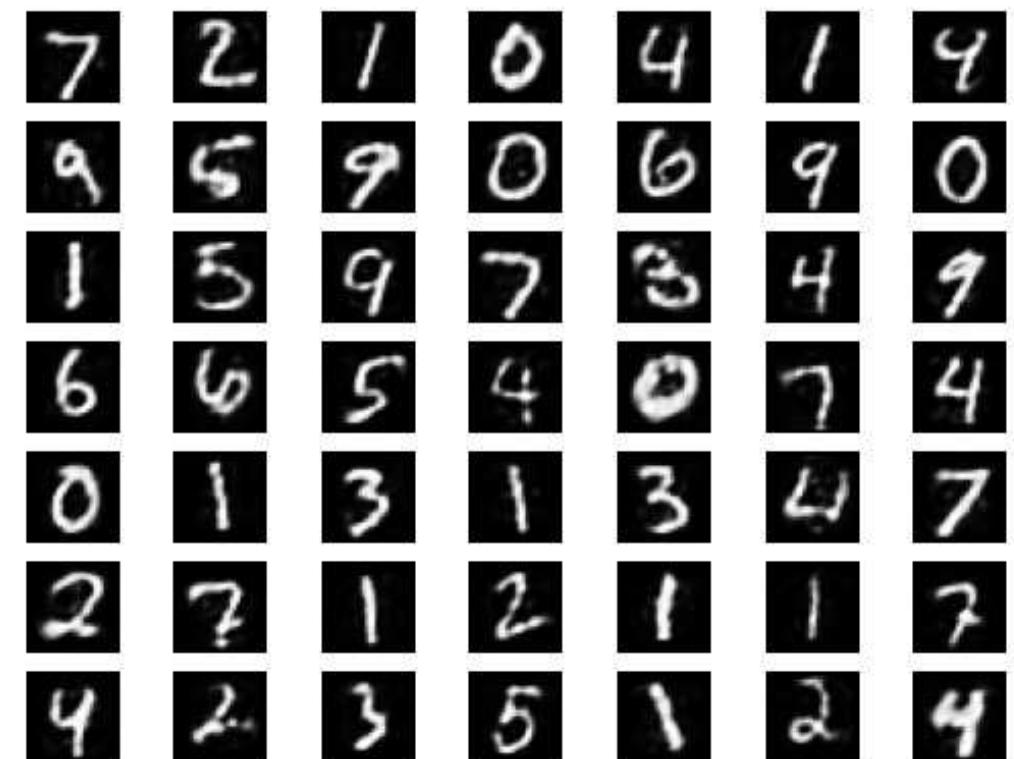


Example 3a: Undercomplete autoencoder

Test inputs

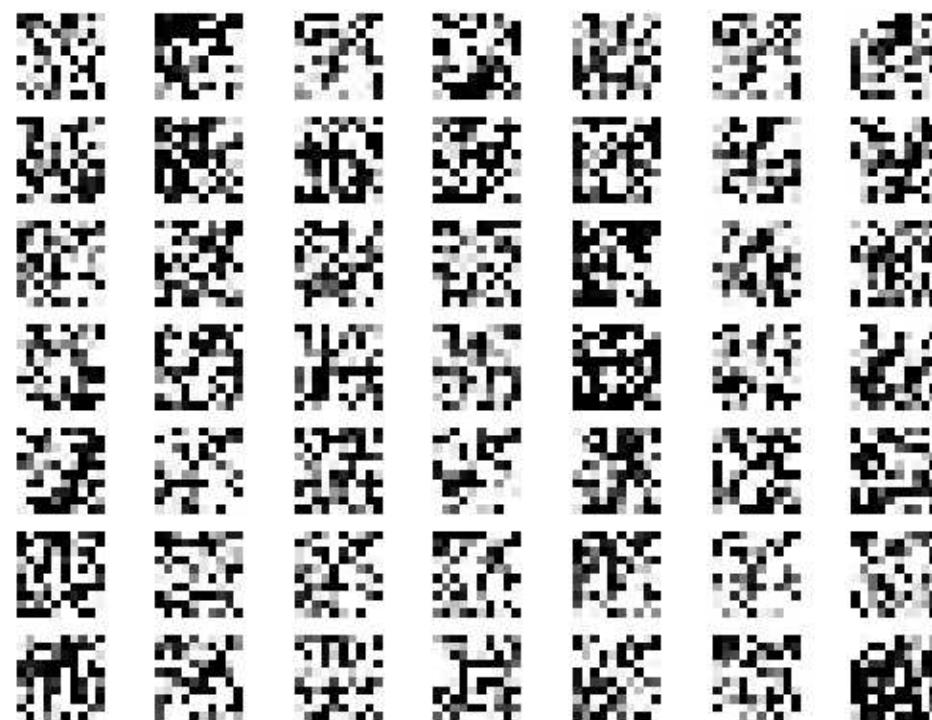


Reconstructed inputs



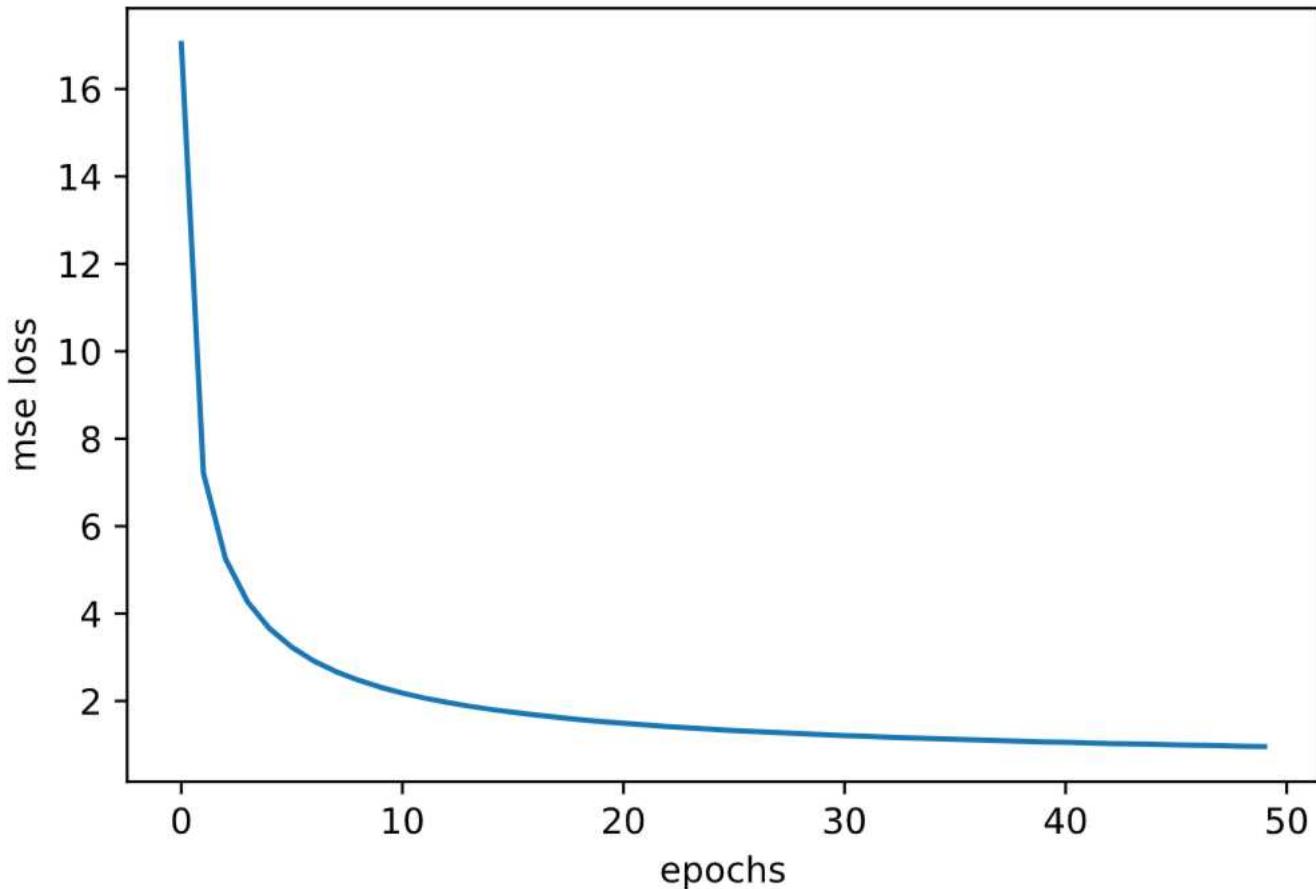
Example 3a: Undercomplete autoencoder

Hidden layer activations of 49 test patterns



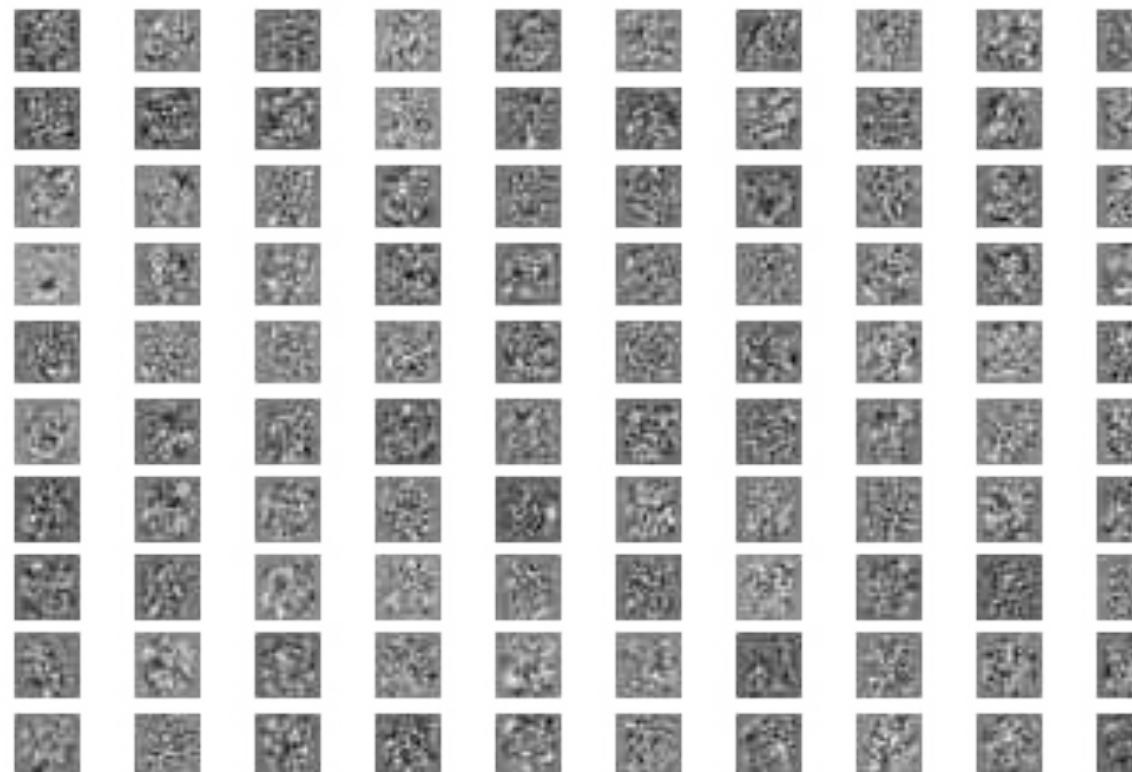
10x10 neurons

Example 3b: Overcomplete autoencoder



Example 3b: Overcomplete autoencoder

Learned weights



900 units

Example 3b: Overcomplete autoencoder

Test inputs

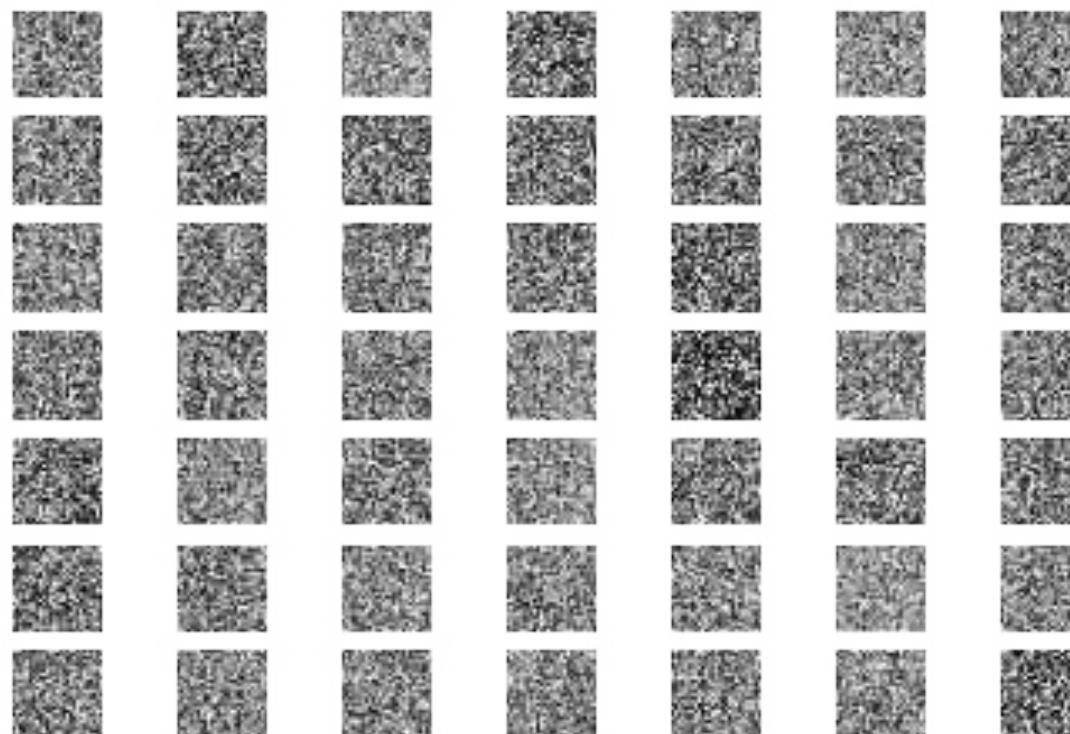
7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Reconstructed inputs

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

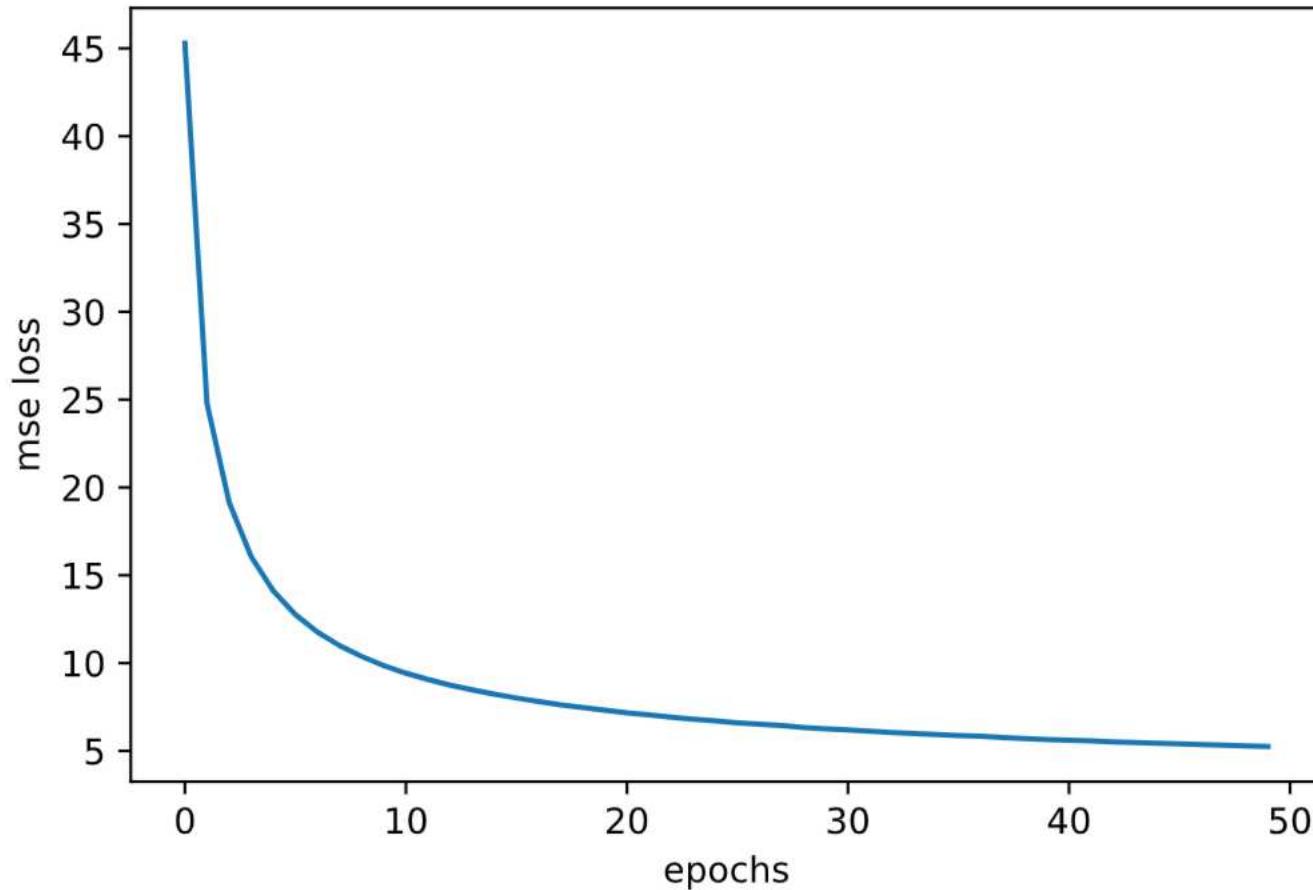
Example 3b: Overcomplete autoencoder

Hidden layer activations



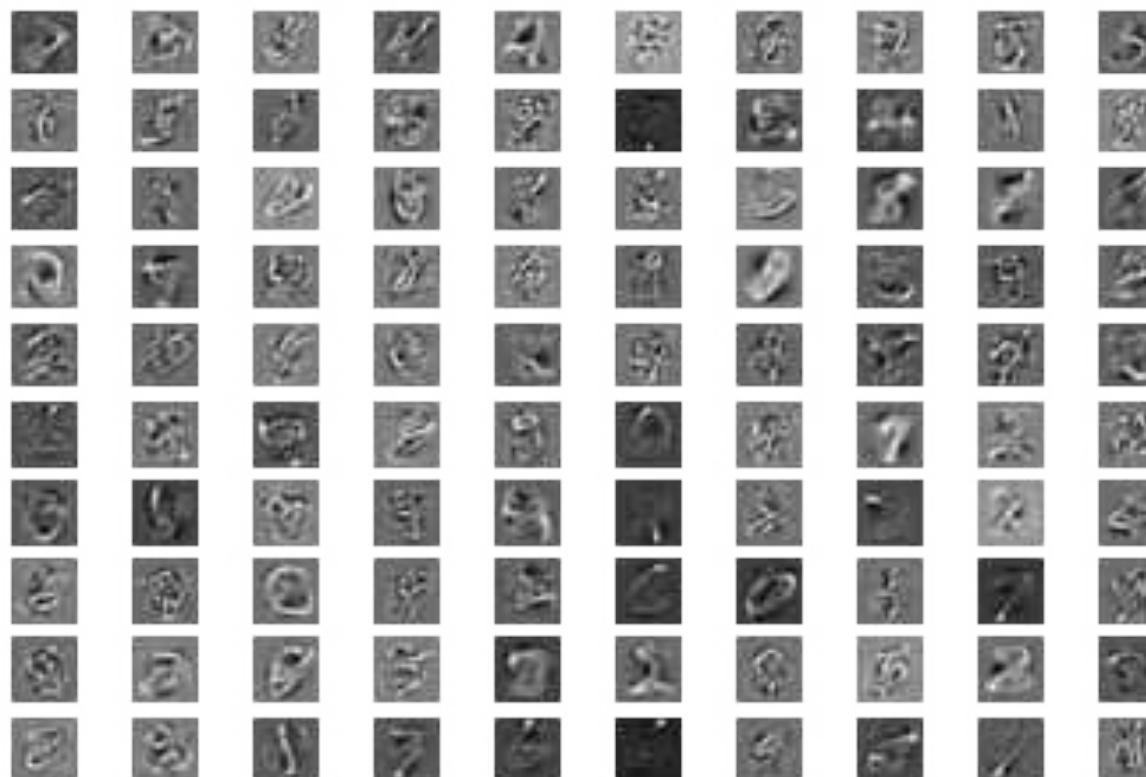
30x30 neurons

Example 3c: Sparse autoencoder



Example 3c: Sparse autoencoder

Learned weights



Example 3c: Sparse autoencoder

Test inputs

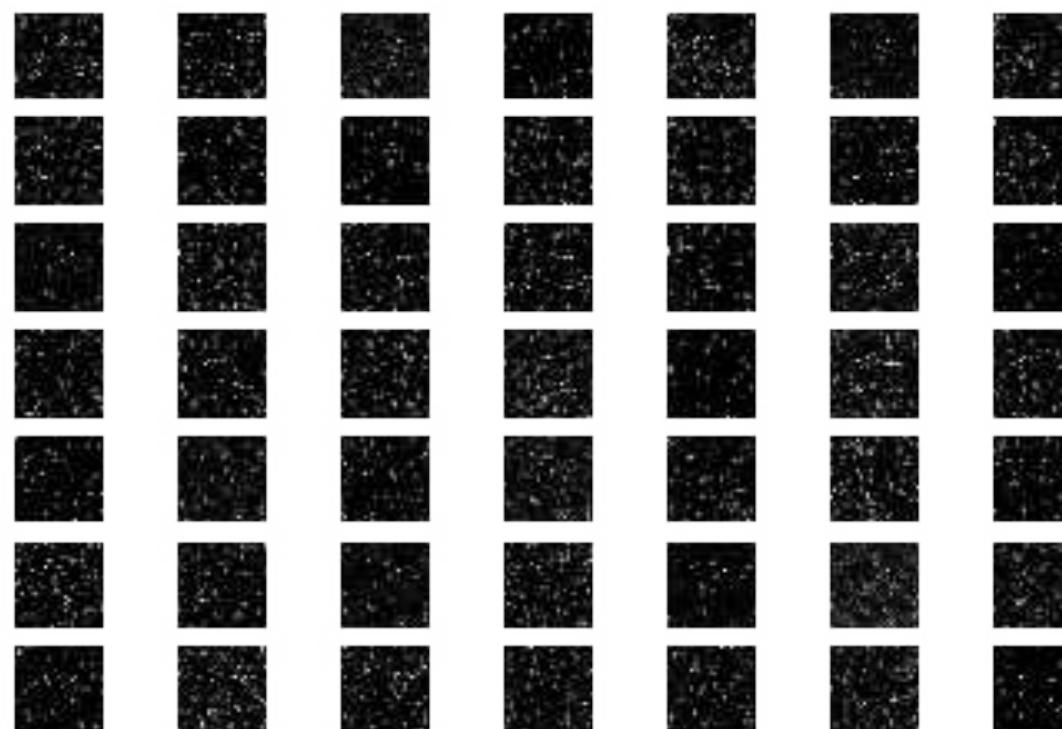
7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Reconstructed inputs

7	2	1	0	4	1	4
9	5	9	0	6	9	0
1	5	9	7	3	4	9
6	6	5	4	0	7	4
0	1	3	1	3	4	7
2	7	1	2	1	1	7
4	2	3	5	1	2	4

Example 3c: Sparse autoencoder

Hidden layer activations



30x30 neurons

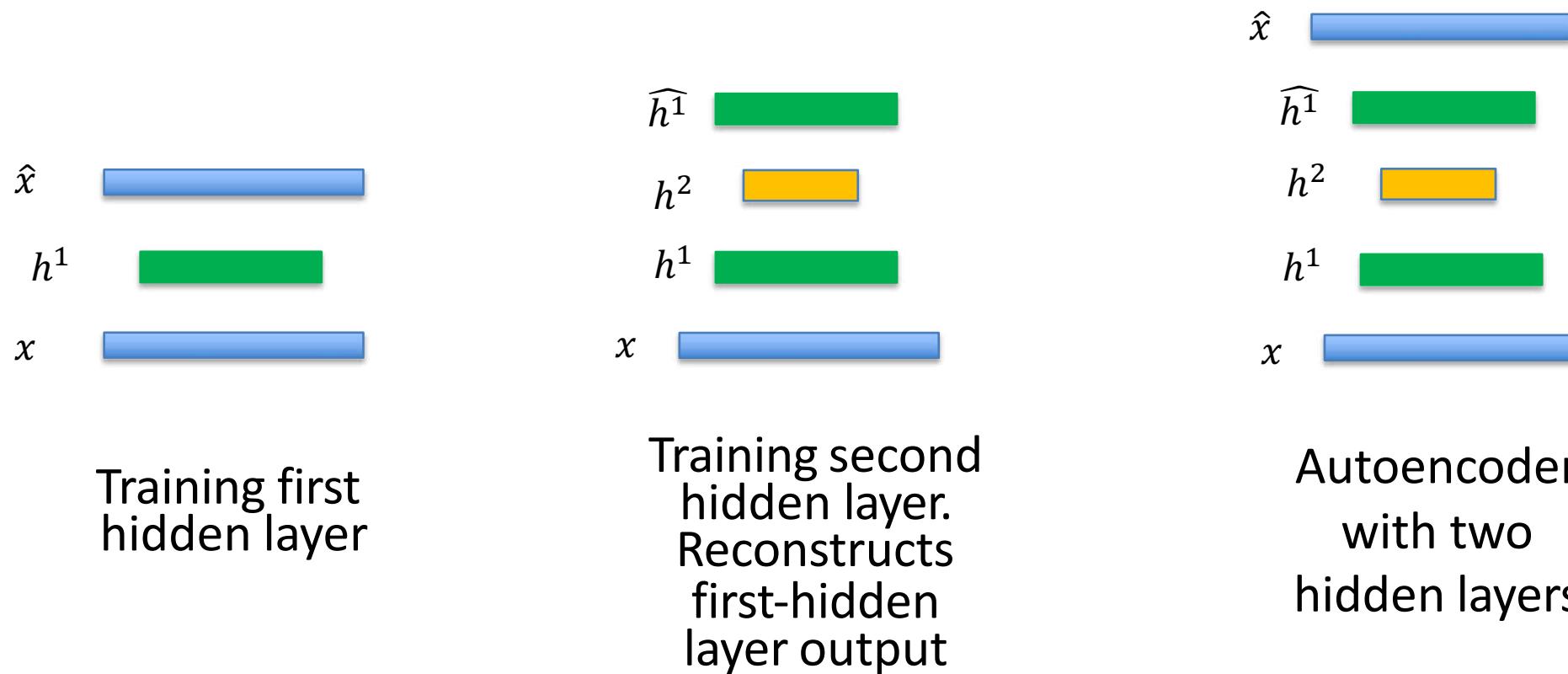
Deep stacked autoencoders

Deep autoencoders can be built by **stacking autoencoders** one after the other.

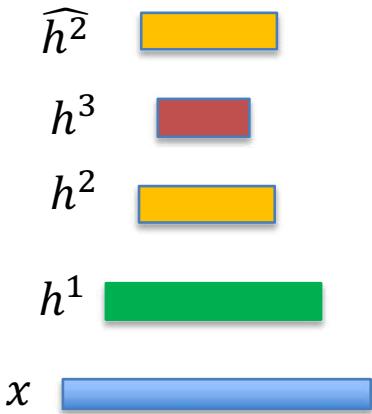
Training of deep autoencoders is done in a step-by-step fashion **one layer at a time**:

- After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder.
- The second level hidden representation can be used to train the third level of the encoders.
- This process is repeated and deep stacked autoencoder can be realized.

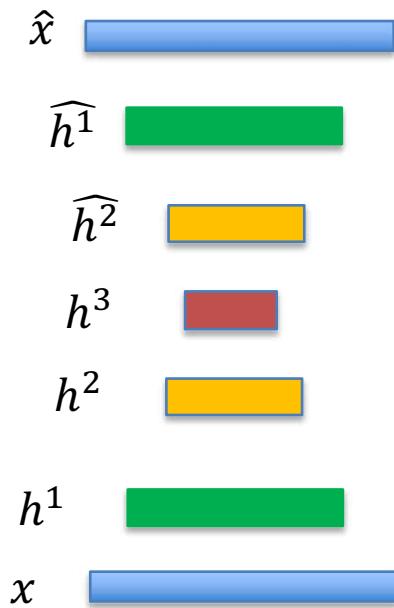
Deep stacked autoencoders



Deep stacked autoencoders



Training the third hidden layer.
Reconstructs second hidden
layer output

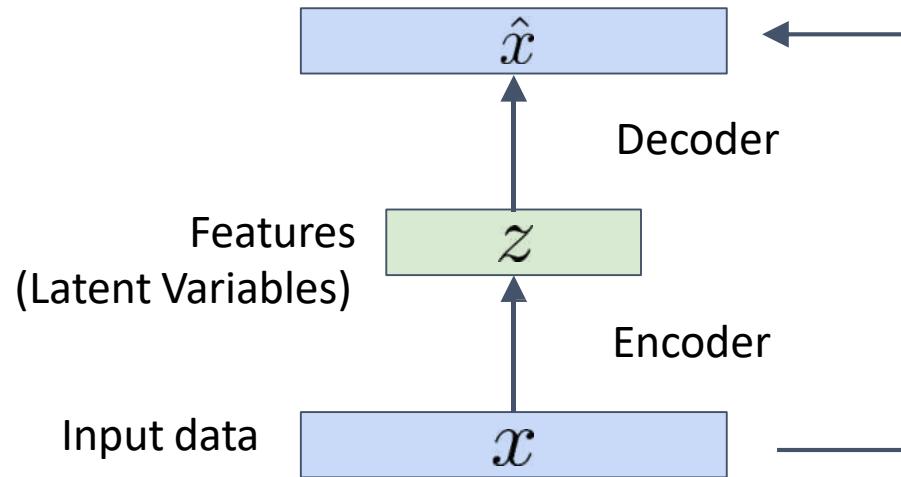


Autoencoder with
three hidden layers

Semi-Supervised Classification

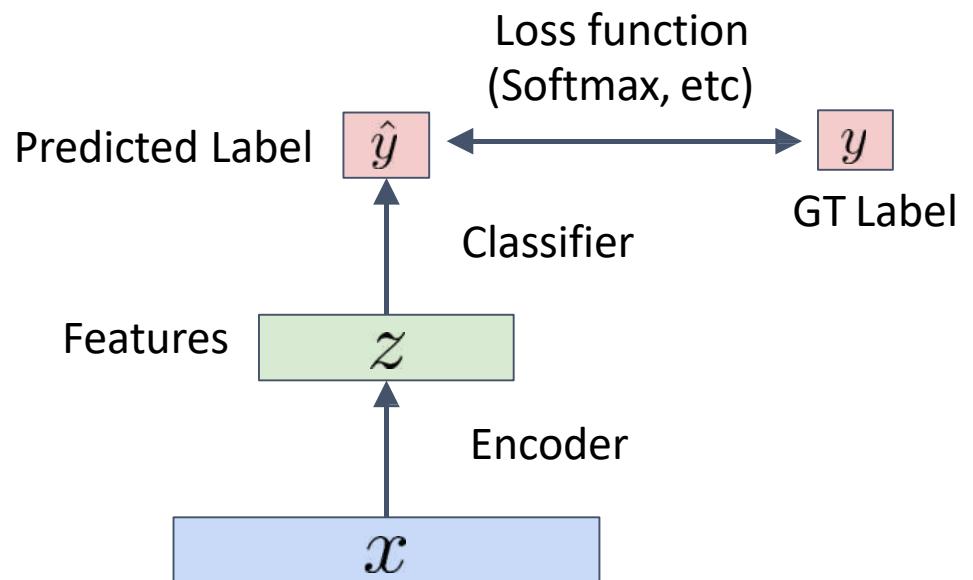
- Many images, but few ground truth labels

start unsupervised
train autoencoder on many images



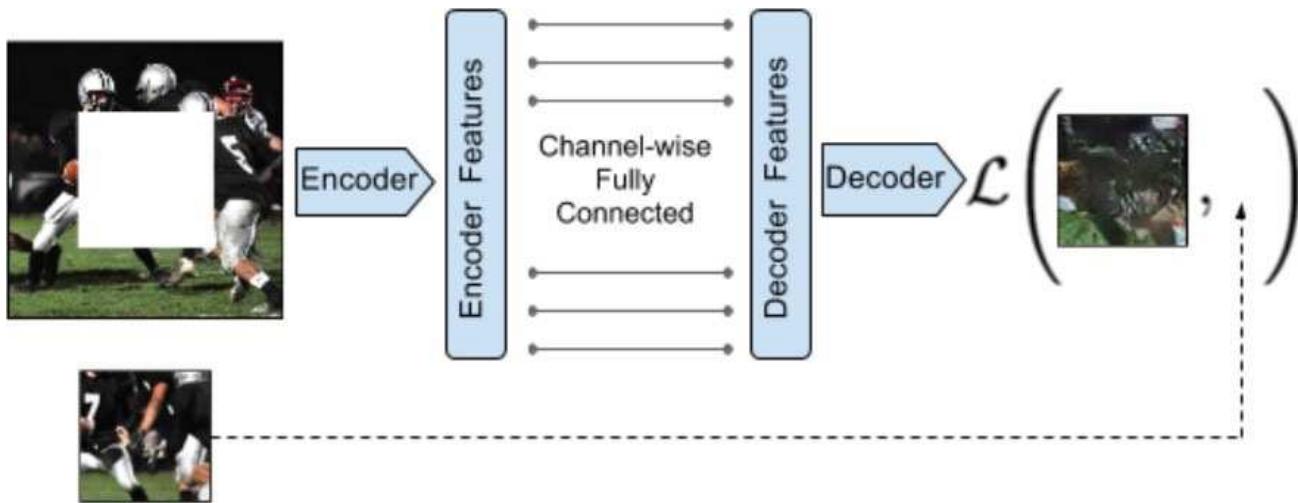
L2 Loss function:
 $\|x - \hat{x}\|^2$

supervised fine-tuning
train classification network on labeled images



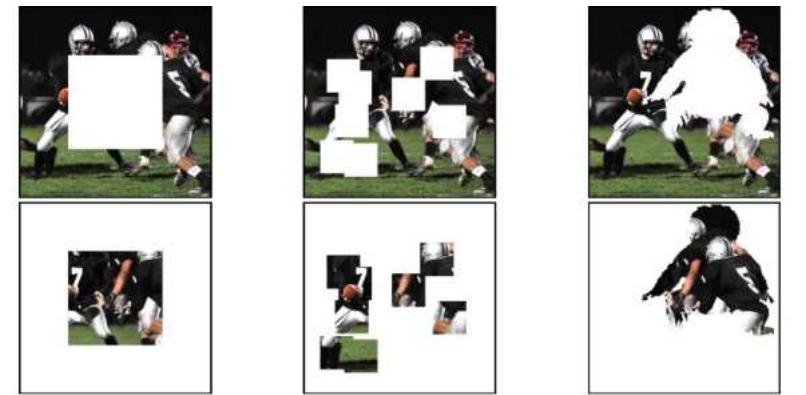
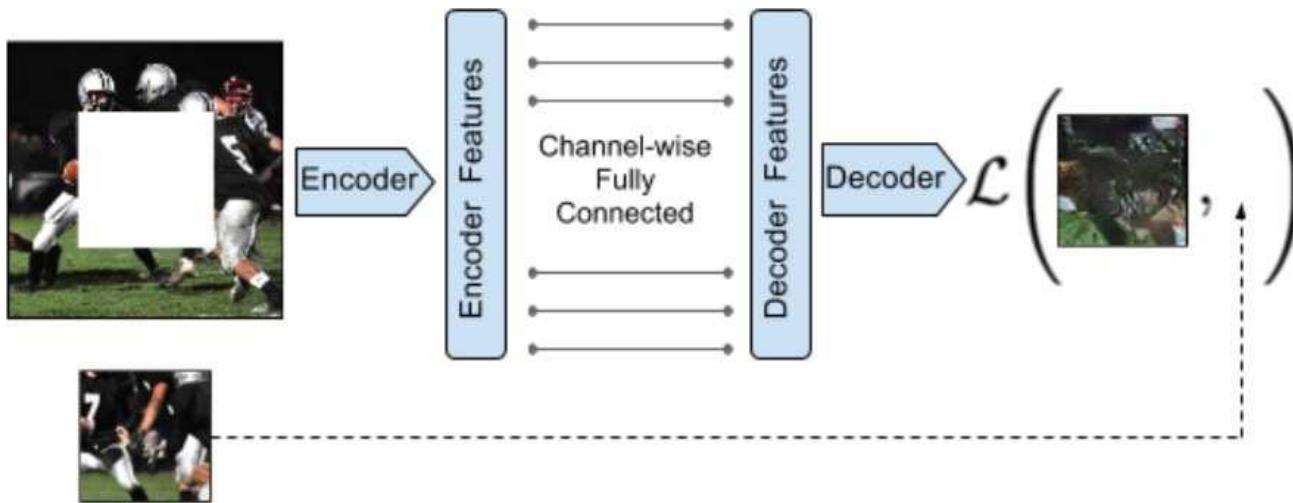
Context Encoders

[Pathak et al., 2016]



Context Encoders

[Pathak et al., 2016]



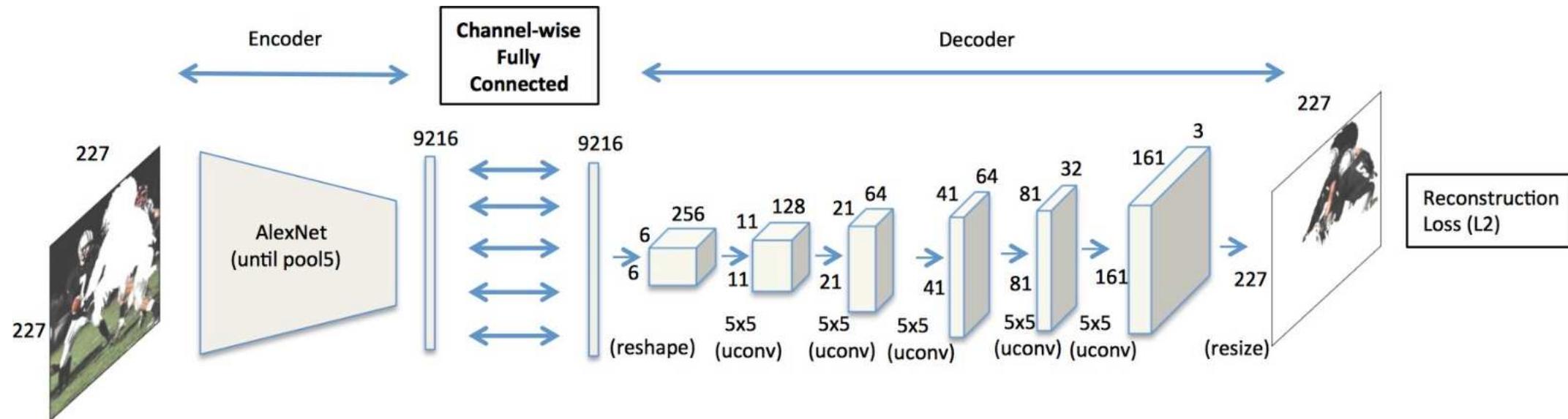
(a) Central region

(b) Random block

(c) Random region

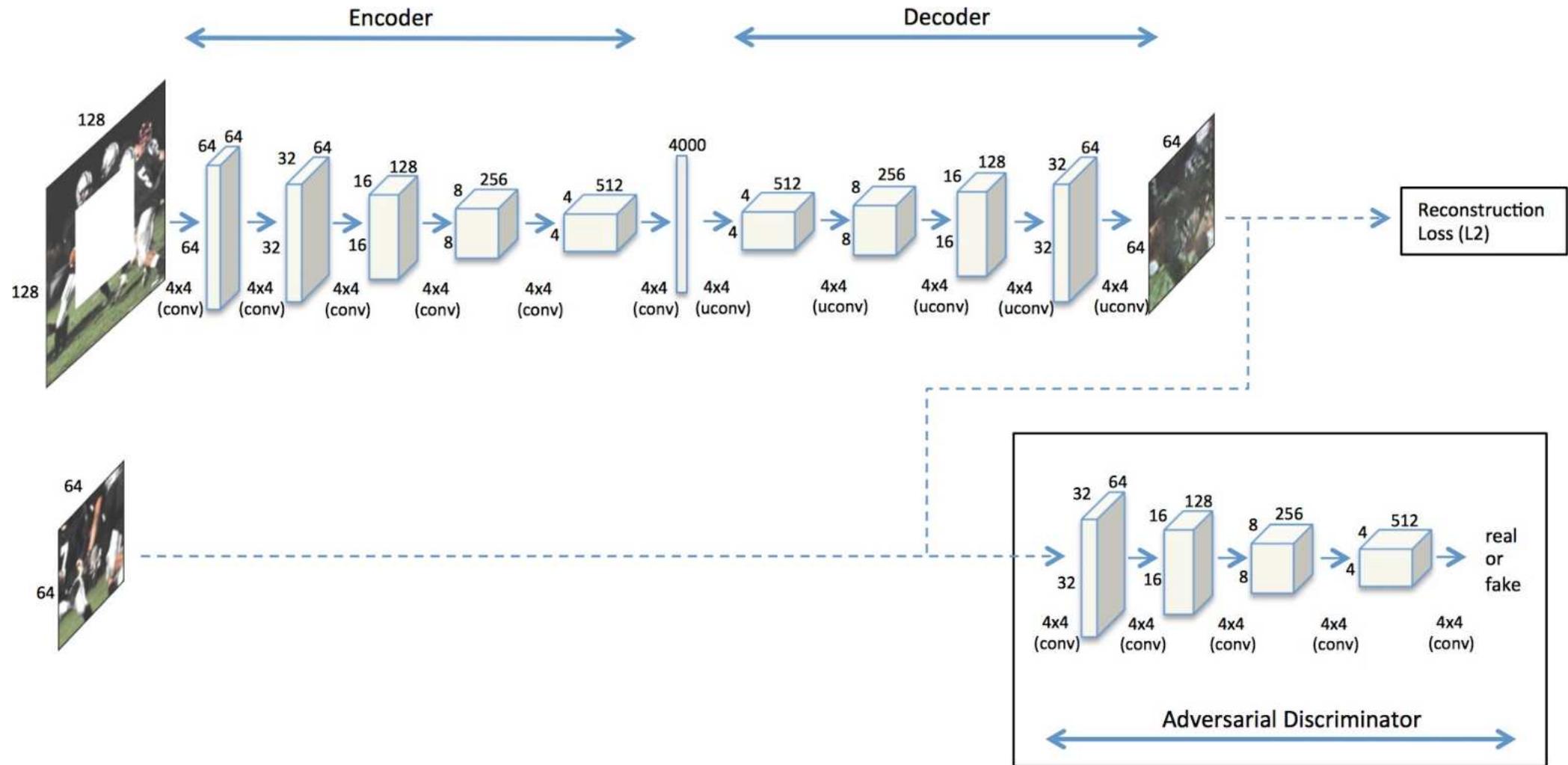
Context Encoders

[Pathak et al., 2016]



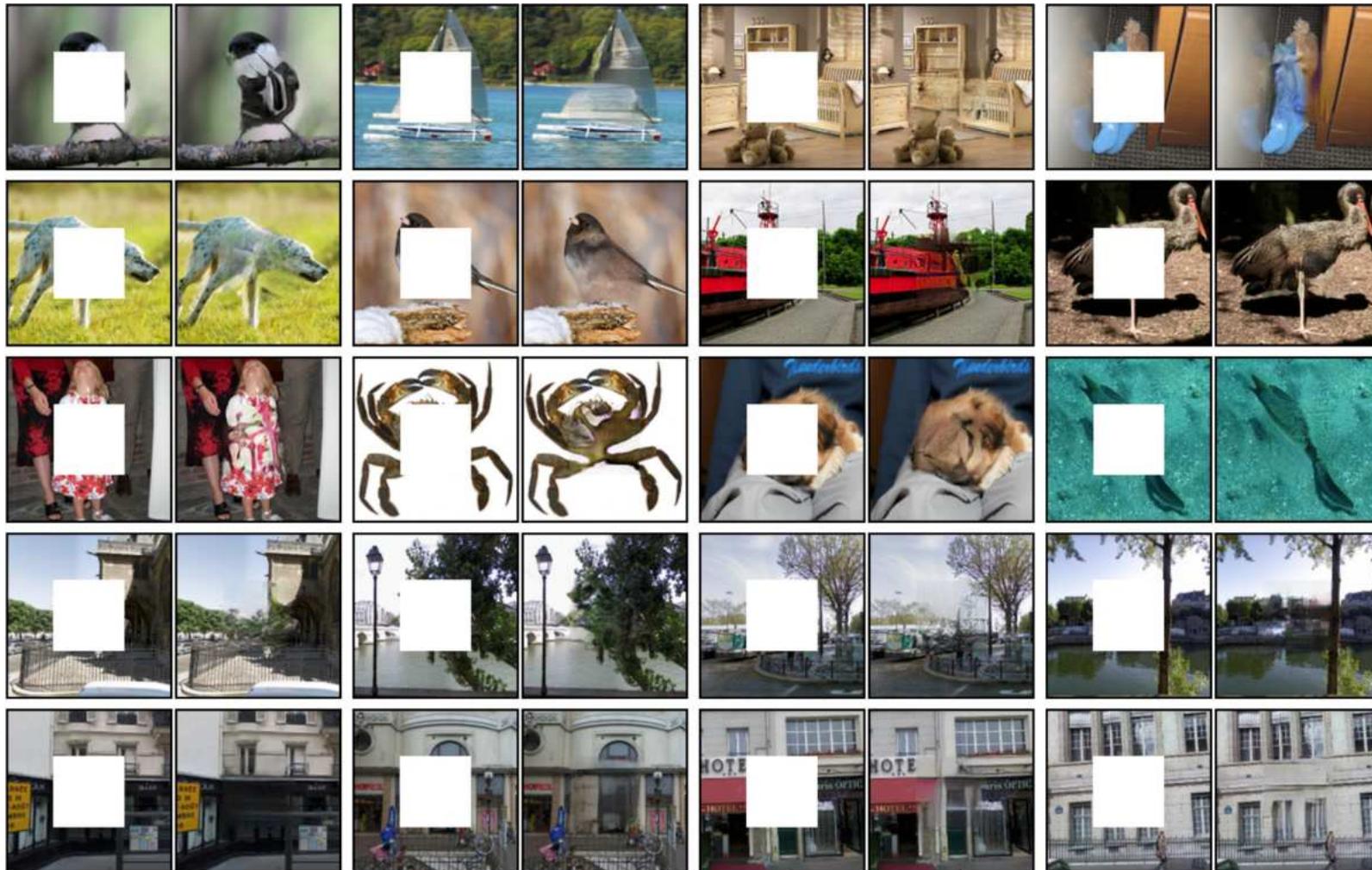
Context Encoders

[Pathak et al., 2016]



Context Encoders

[Pathak et al., 2016]



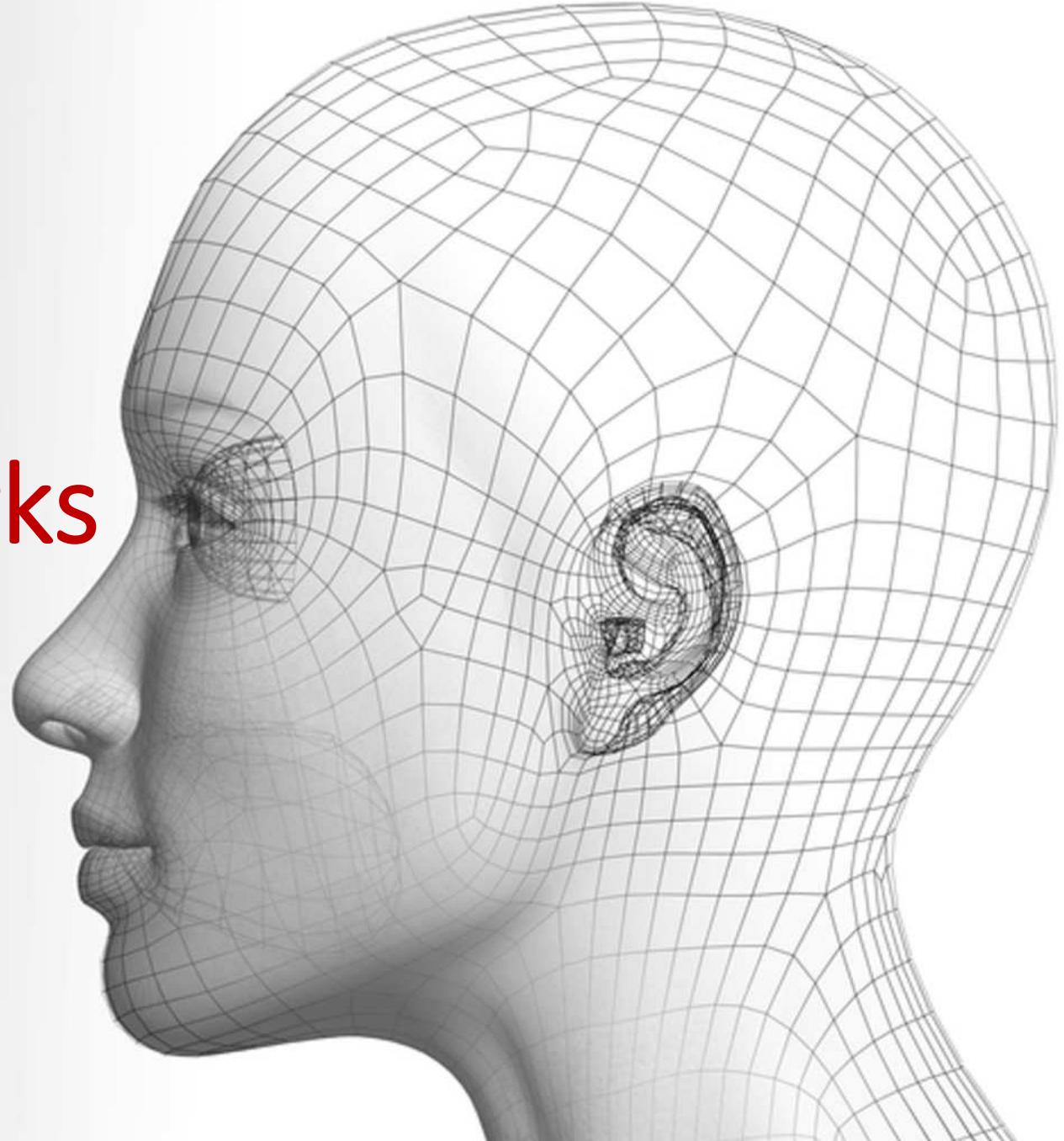
Generative Adversarial Networks

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

<https://twitter.com/XingangP>



Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Supervised learning

Data: (x, y)

x is data, y is label

Goal: Learn a *function* to map $x \rightarrow y$

Examples:

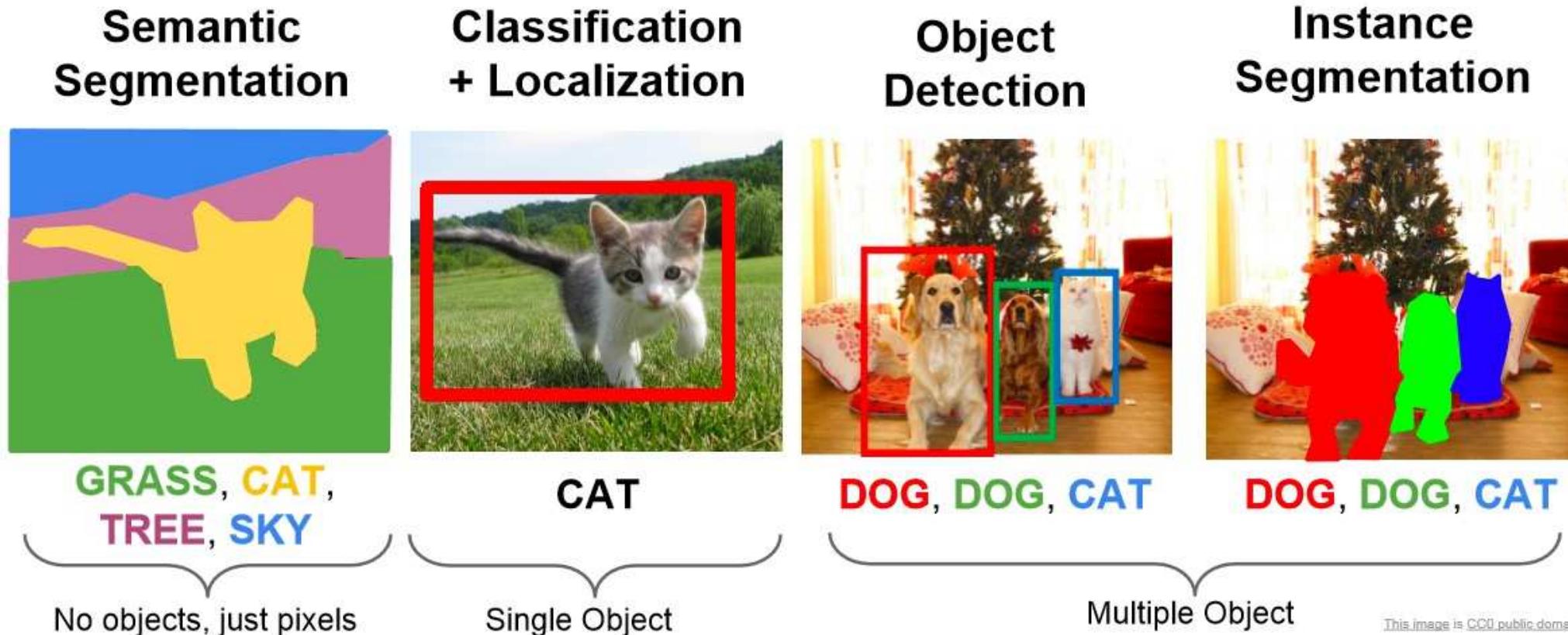
- Classification,
- regression
- object detection
- semantic segmentation,
- image captioning, etc.



→ **Cat**

Classification

Applications of supervised learning



Why generative models?

- We've only seen *discriminative models* so far
 - Given an data x , predict a label y
 - Estimates $p(y|x)$
- Discriminative models have several key limitations
 - Can't model $p(x)$, i.e., the probability of seeing a certain data
 - Thus, can't sample from $p(x)$, i.e., **can't generate new data**
- *Generative models* (in general) cope with all of above
 - Can model $p(x)$
 - Can generate new data or images

Generative modeling

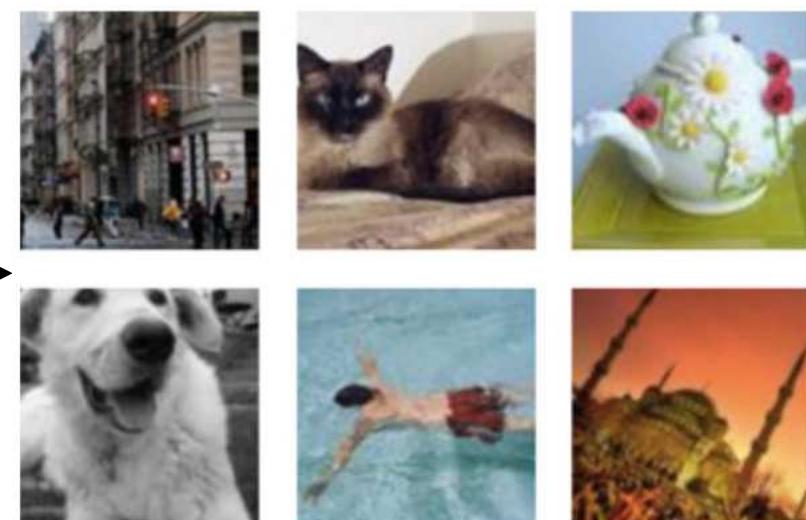
Density estimation – a core problem in unsupervised learning



Sample generation



Training samples – $p_{data}(x)$



Model samples – $p_{model}(x)$
(ideally $p_{model}(x) = p_{data}(x)$)

Image generated - ImageNet

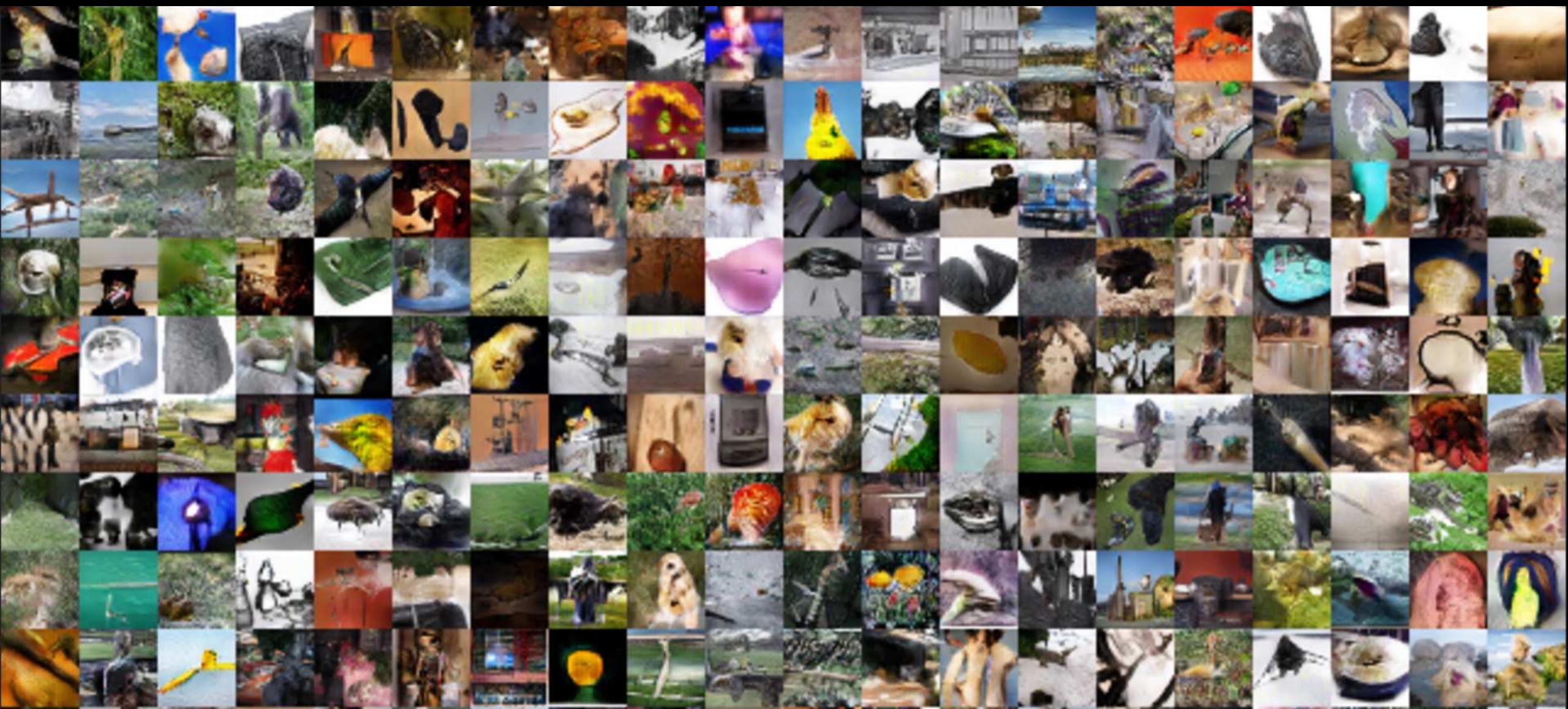
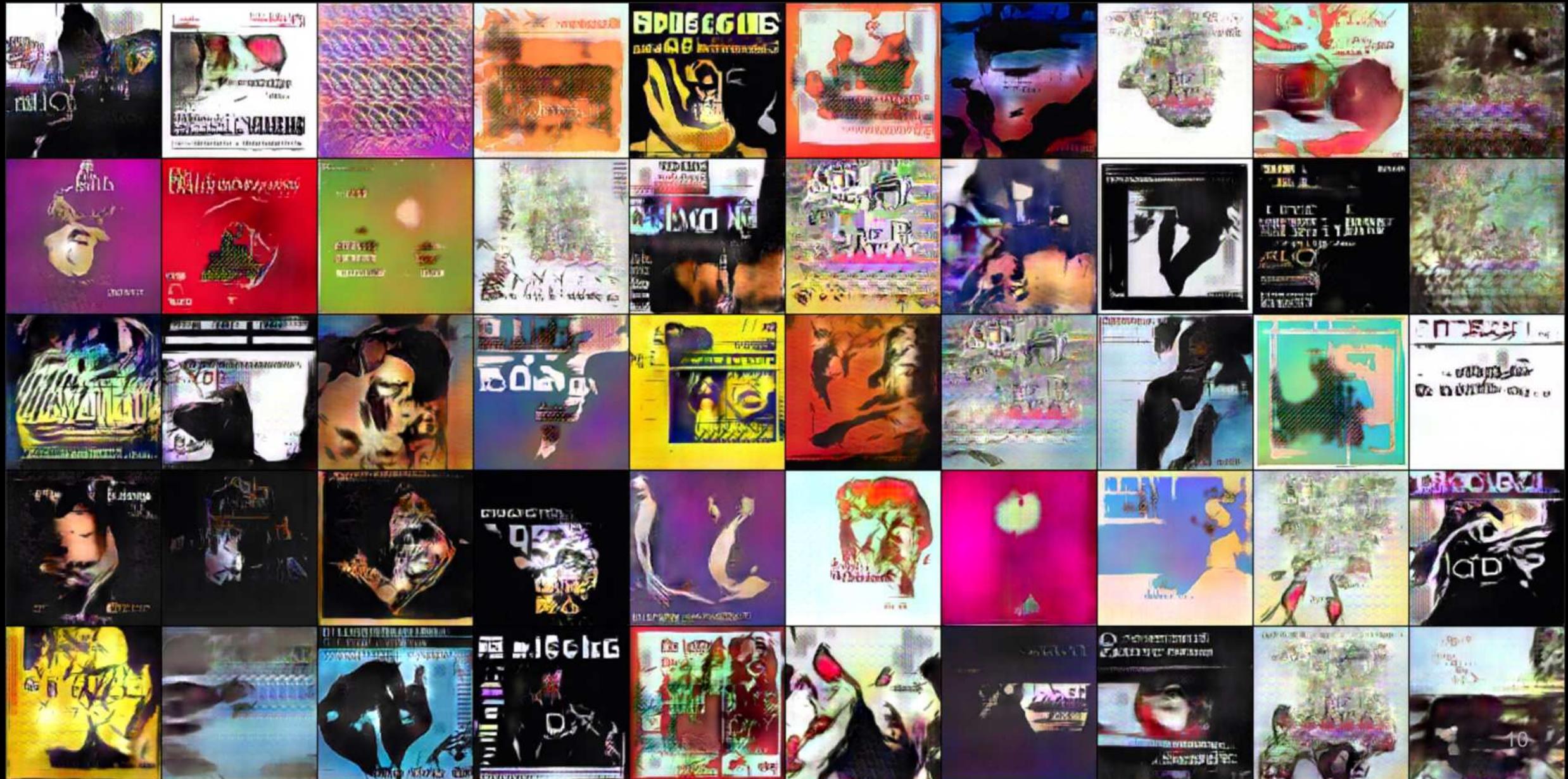


Image generated - Bedrooms



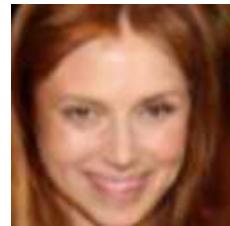
Image generated - Albums



The GAN Revolution 2014-2017



Conditional GAN



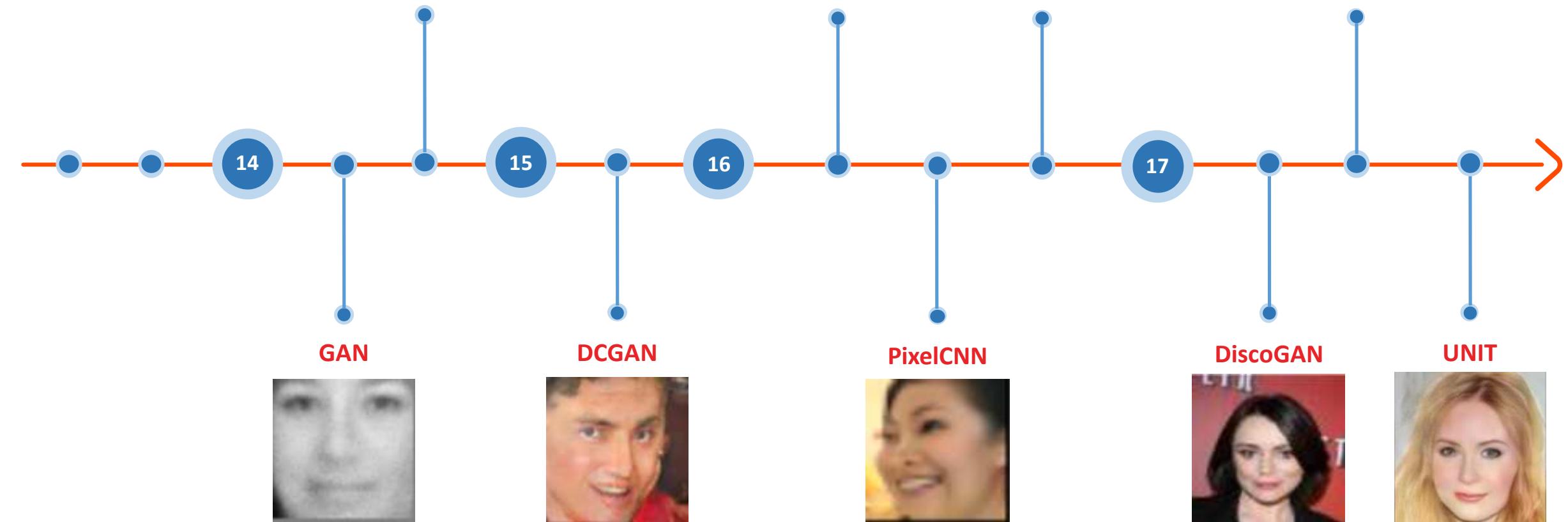
VAE/GAN



CoGAN



CycleGAN



The GAN Revolution 2018 - Present



StarGAN



ELEGANT



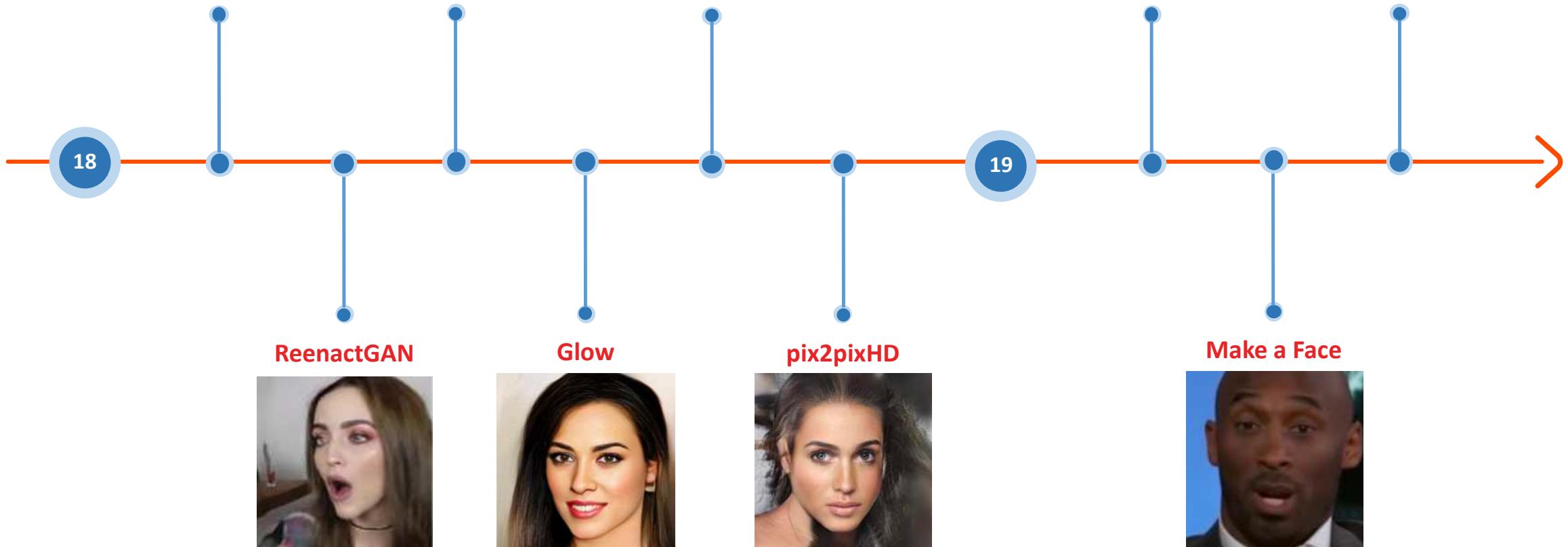
PGGAN



BeautyGlow



StyleGAN



Results of StyleGAN



Credit: Tero Karras et al., A Style-Based Generator Architecture for Generative Adversarial Networks, CVPR 2019

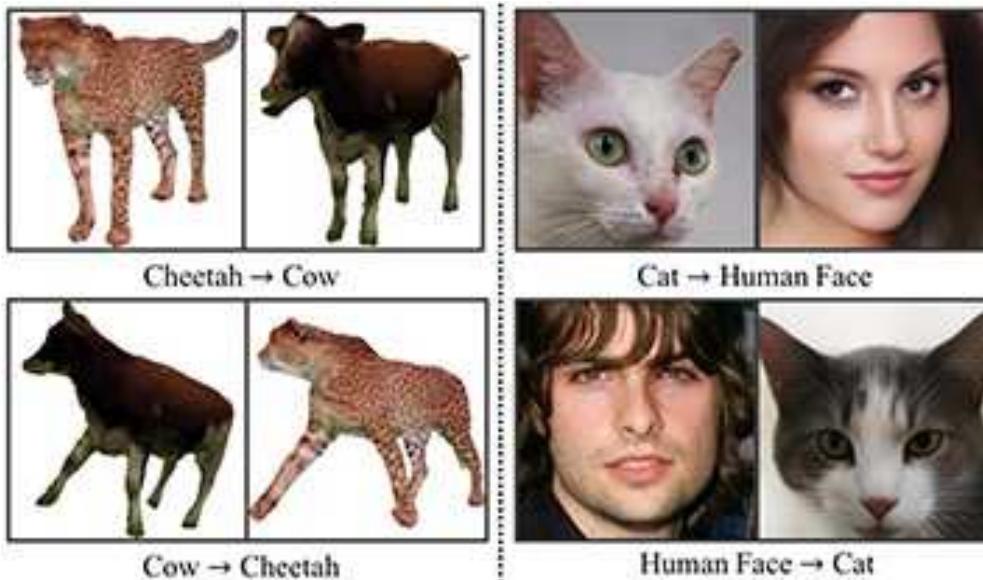
Almost Everything can be Forged!



CycleGAN
[Zhu et al., ICCV 2017]

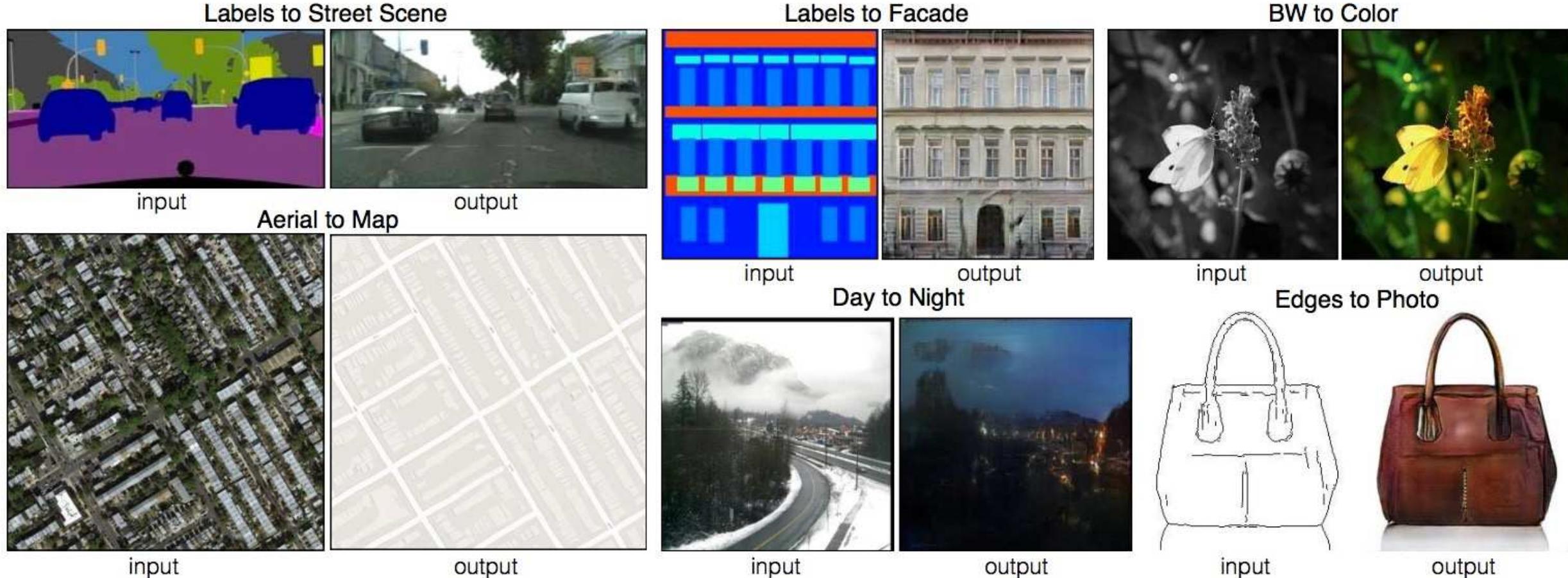


StarGAN
[Choi et al., CVPR 2018]



TransGaGa
[Wu et al., CVPR 2019]

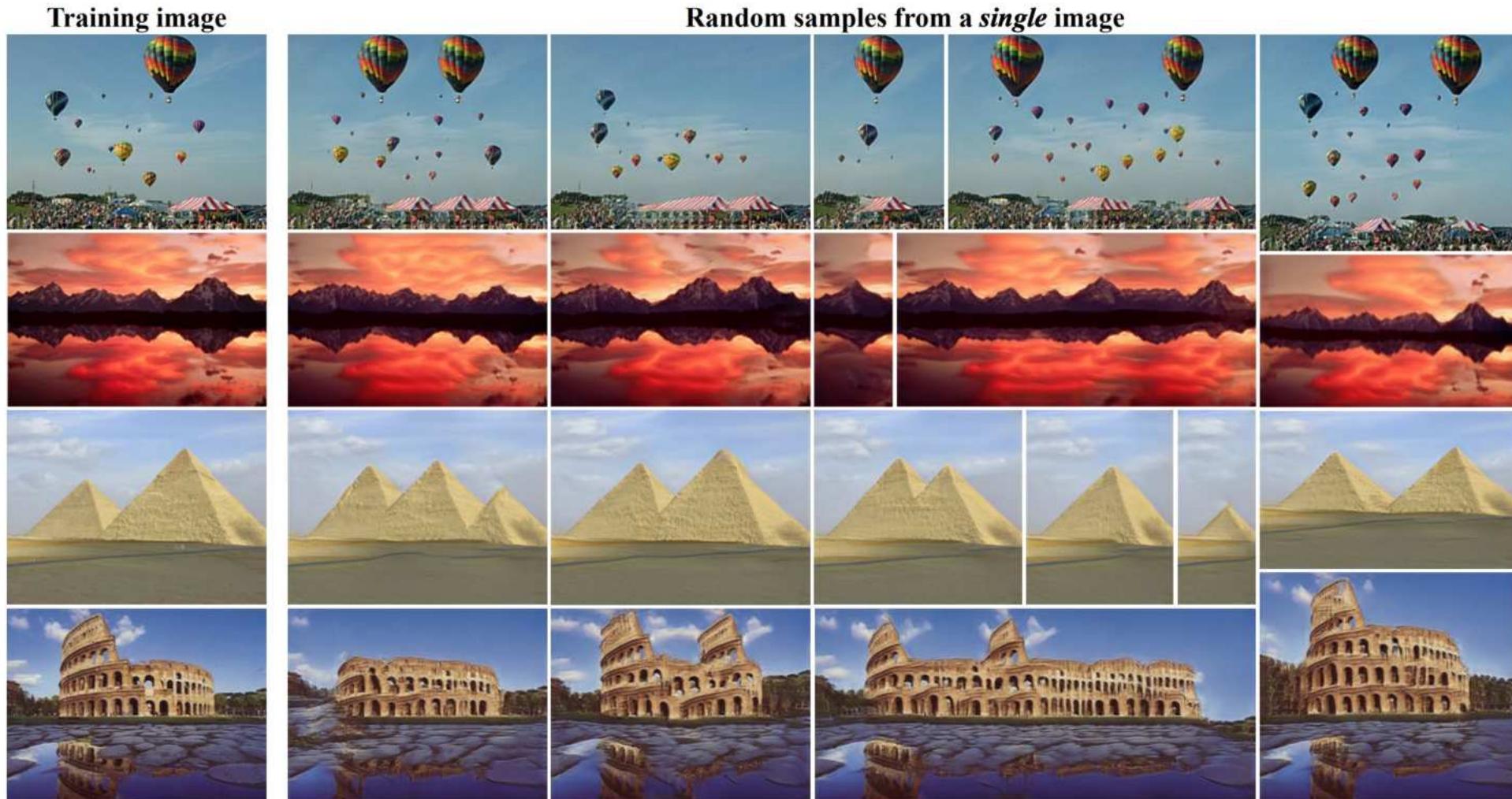
Image-to-Image Translation



Isola et al. "Image-to-Image Translation with Conditional Adversarial Networks", arXiv:1611.07004

<https://phillipi.github.io/pix2pix/>

SinGAN: Learning a Generative Model from a Single Natural Image



Audio-driven Emotional Face Manipulation



Angry



Contempt



Fear



Input Video



Reference Style



Our Toonification Result

Edit upper length (StyleSpace)



Edit bottom length (StyleSpace)



Edit upper length (InterFaceGAN)



Edit bottom length (InterFaceGAN)

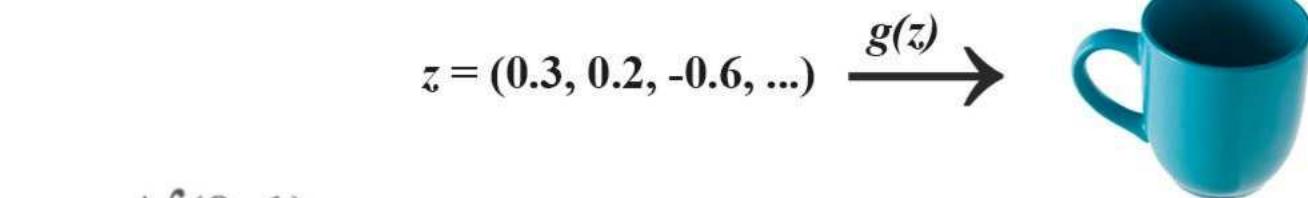


Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

GAN Basics

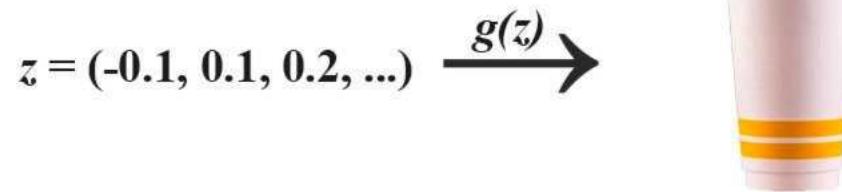
Generative Adversarial Networks (GAN)



$z \sim \mathcal{N}(0, 1)$

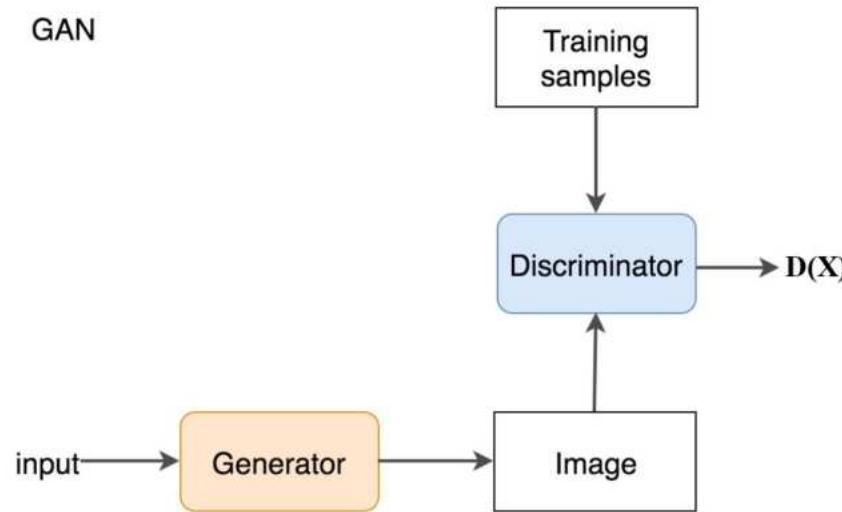
or

$z \sim U(-1, 1)$



GAN samples noise z using normal or uniform distribution and utilizes a deep network generator G to create an image x ($x=G(z)$)

Generative Adversarial Networks (GAN)

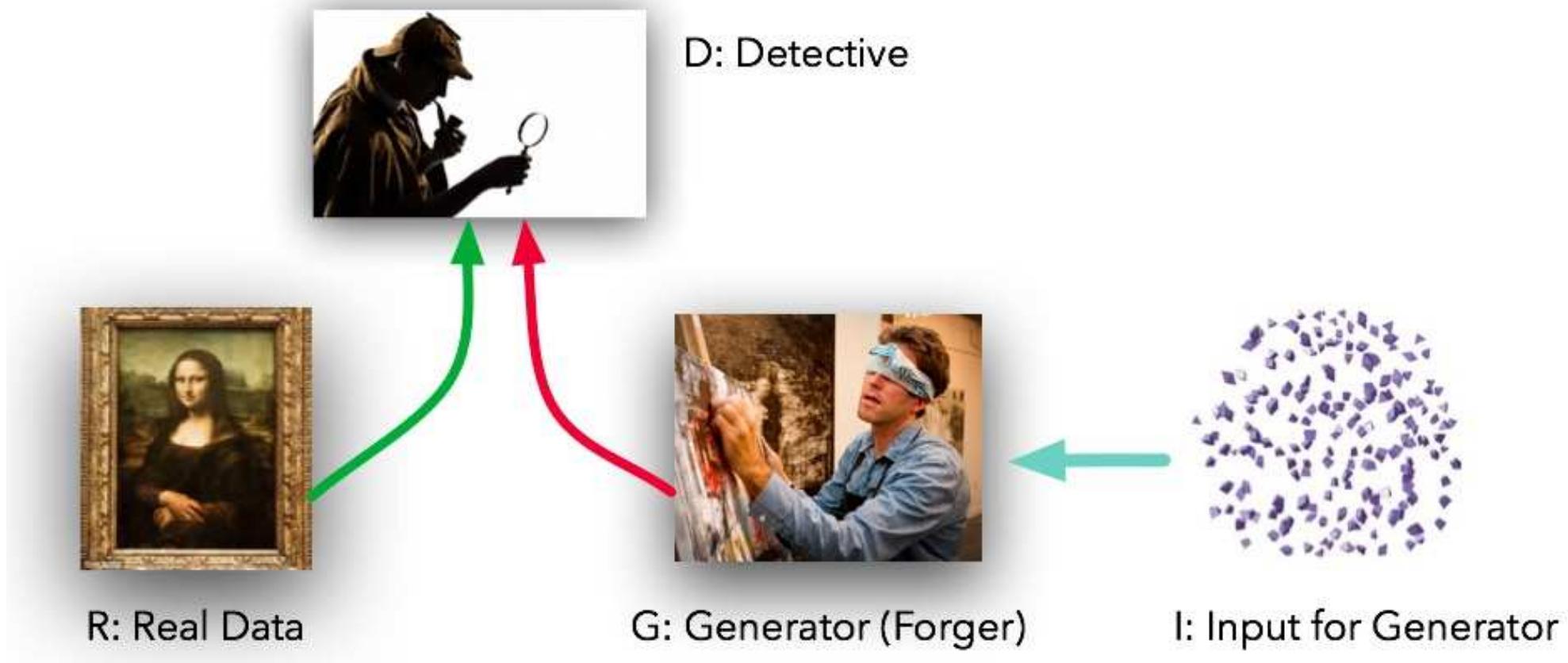


In GAN, we add a discriminator to distinguish whether the discriminator input is real or generated. It outputs a value $D(x)$ to estimate the chance that the input is real.

$D(x) = 1$ suggests x is real

$D(x) = 0$ suggests x is fake

Generative Adversarial Networks (GAN)



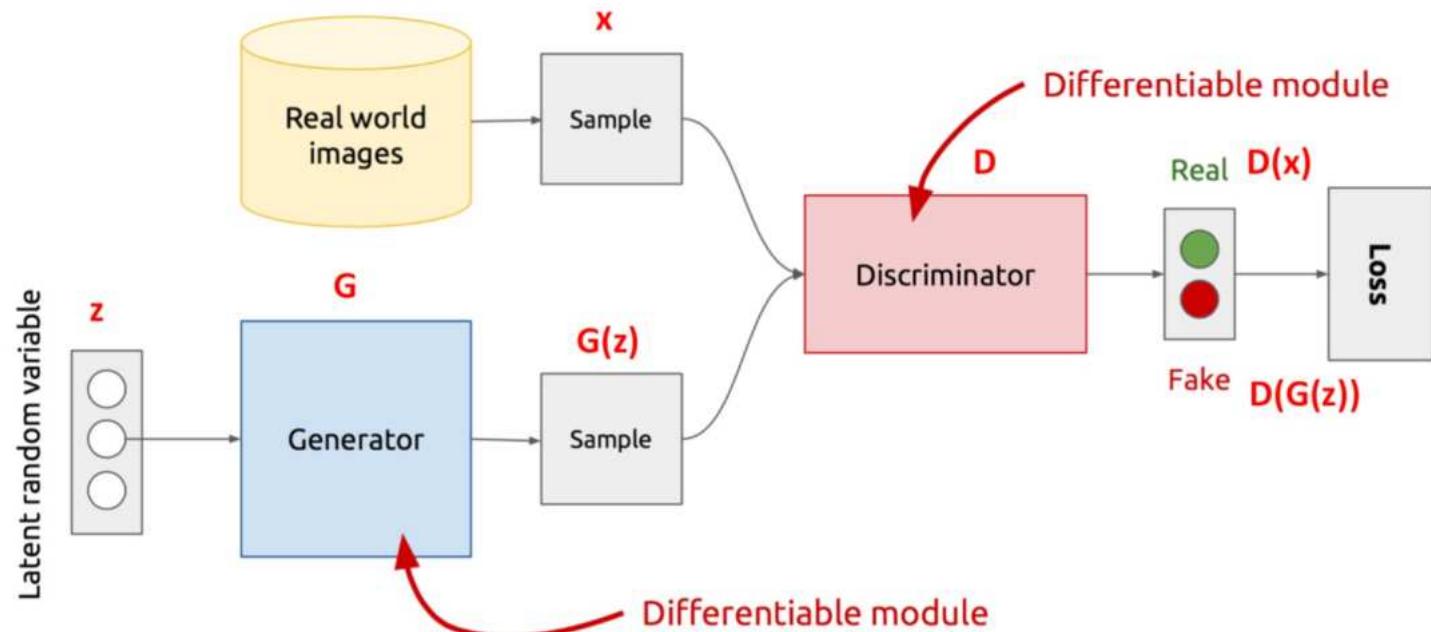
Generative adversarial networks (GAN)

- **Generative model G :**

- Captures data distribution
- Fool $D(G(z))$
- Generate an image $G(z)$ such that $D(G(z))$ is wrong (i.e. $D(G(z)) = 1$)

- **Discriminative model D :**

- Distinguishes between real and fake samples
- $D(x) = 1$ when x is a real image, and otherwise

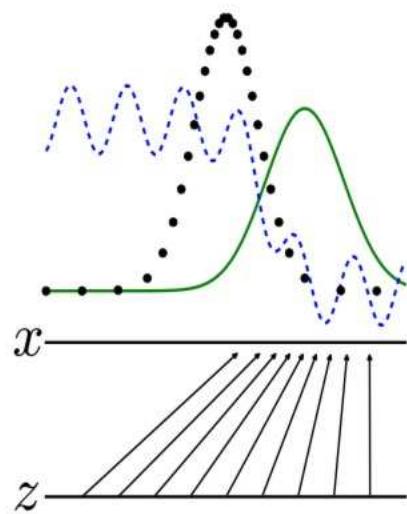


z is some random noise (Gaussian/Uniform).

z can be thought as the latent representation of the data.

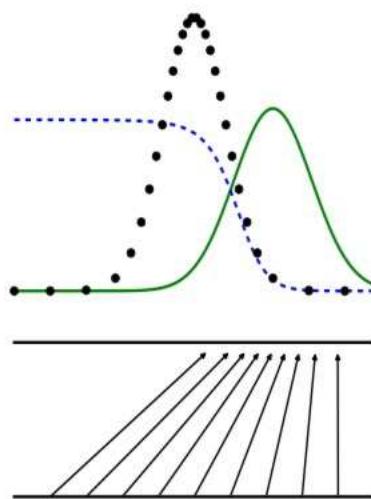
Generative adversarial networks (GAN)

----- data generating distribution
——— generative distribution
- - - discriminative distribution



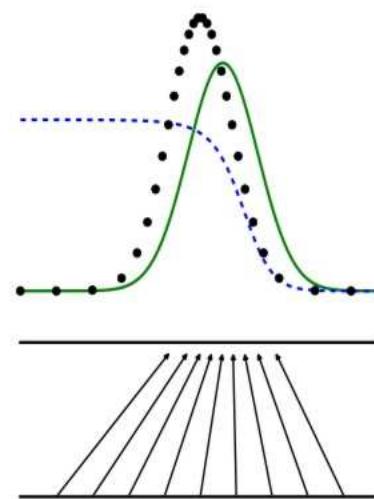
(a)

An adversarial pair near convergence



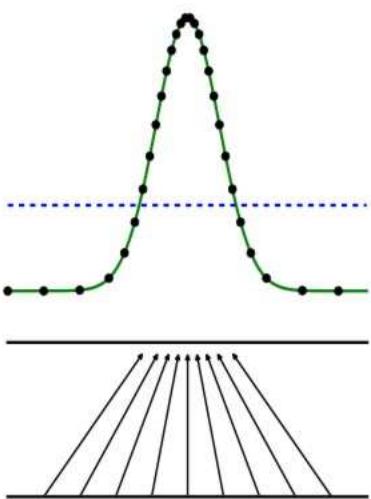
(b)

D is trained to discriminate samples from data



(c)

After an update to G , gradient of D has guided $G(z)$ to flow to regions that are more likely to be classified as data

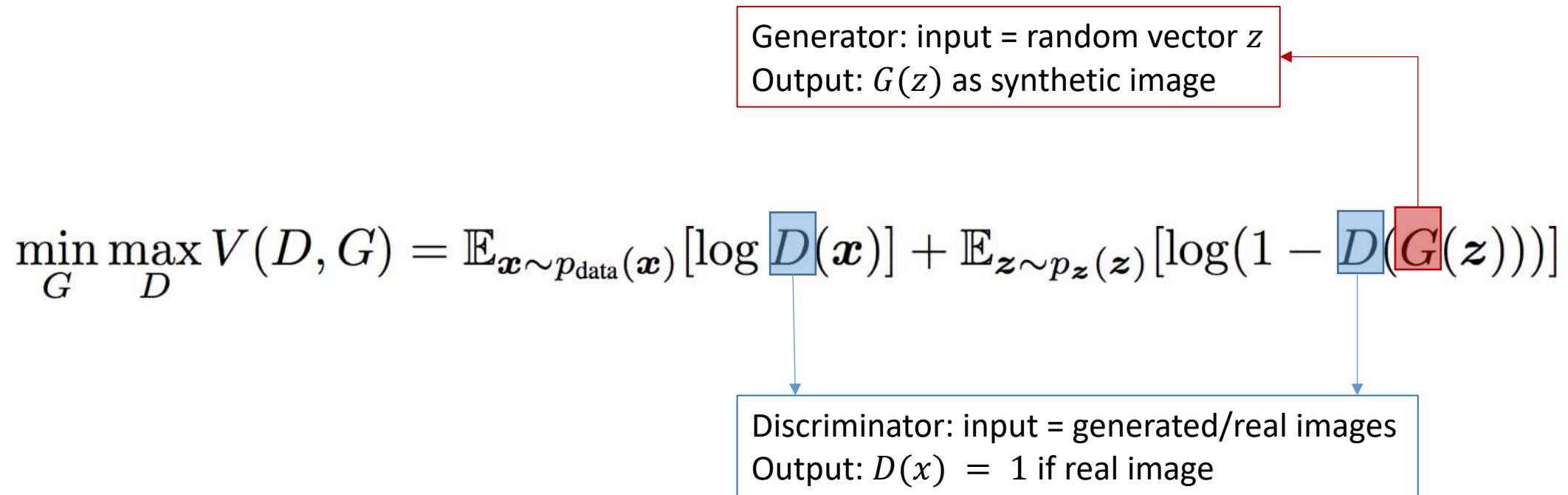


(d)

After several steps of training, G and D will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(x) = 0.5$.

Generative adversarial networks (GAN)

- D and G play the following two-player minimax game with value function $V(D, G)$



Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

GAN Training

Training procedure

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

maximize

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

minimize

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**
for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

Uniform noise vector (random numbers)

maximize

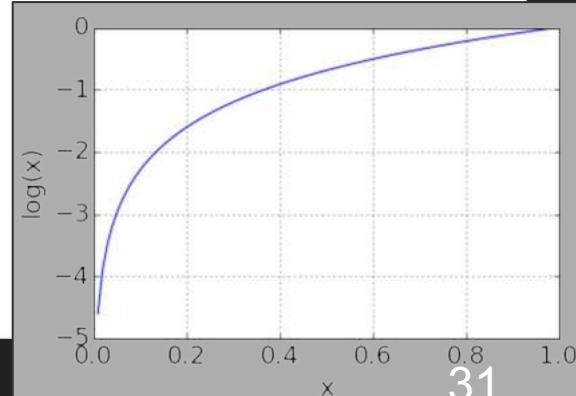
Real images

Generator:
 input=*random numbers*,
 output=*synthetic image*

Discriminator: (input=*generated/real image*,
 output=*prediction of real image*)

Let's do an example

Note showing $\ln(x)$ not $\log(x)$



end for

$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

$$D(x) = 0.8 \\ \log(0.8) = -0.2$$

Imagine for a real image $D(x)$ scores 0.8 that it is a real image (good)

Uniform noise vector (random numbers)

maximize

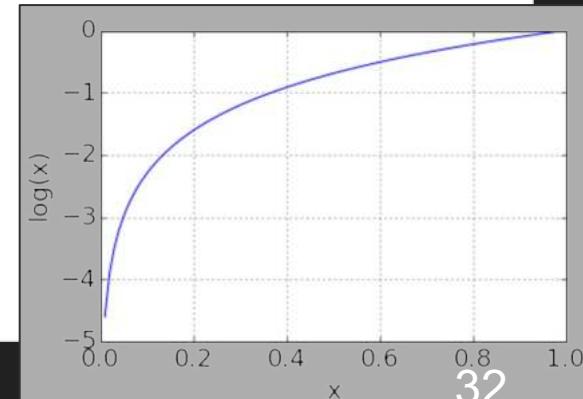
Real images

Generator:

input=random numbers,
output=synthetic image

end for

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

Discriminator: (input=generated/real image, output=prediction of real image)

$$D(x) = 0.8 \\ \log(0.8) = -0.2$$

Then for a **generated** image, $D(x)$ scores 0.2 that it is a real image (good)

$$D(G(z)) = 0.2 \\ \log(1-0.2) = \log(0.8) = -0.2$$

end for

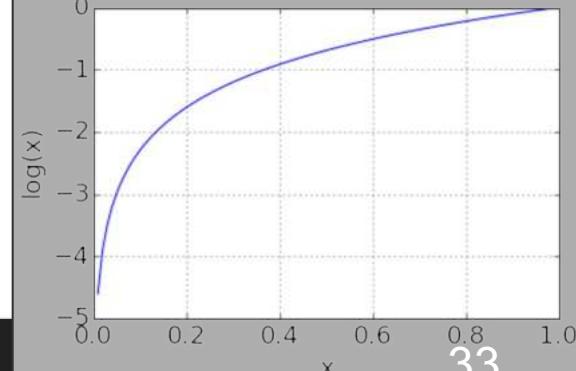
Uniform noise vector (random numbers)

maximize

Real images

Generator:

input=random numbers, output=synthetic image



Note showing $\ln(x)$ not $\log(x)$

$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

Uniform noise vector (random numbers)

maximize

Real images

Generator:
 input=random numbers,
 output=synthetic image

end for

$$D(x) = 0.8$$

$$\log(0.8) = -0.2$$

$$D(G(z)) = 0.2$$

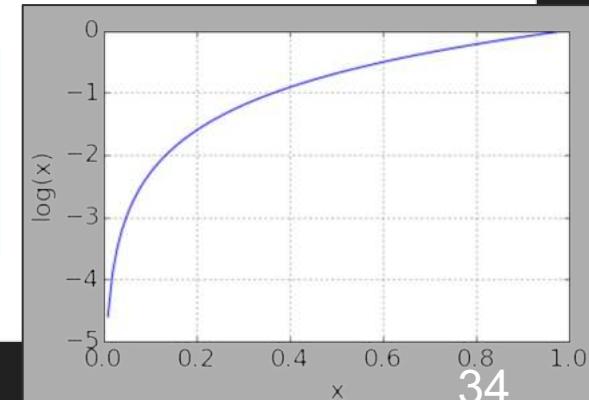
$$\log(1-0.2) = \log(0.8) = -0.2$$

$$-0.2 + -0.2 = -0.4$$

We add them together and this gives us a fairly high (-0.4) loss (note we ascend so we want to maximize this).

Also note that we are adding two negative numbers so 0 is the upper bound

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**
for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

Discriminator: (input=generated/real image, output=prediction of real image)

Uniform noise vector (random numbers)

Real images

Generator:
 input=random numbers,
 output=synthetic image

$$D(x) = 0.8$$

$$\log(0.8) = -0.2$$

Let's do another example

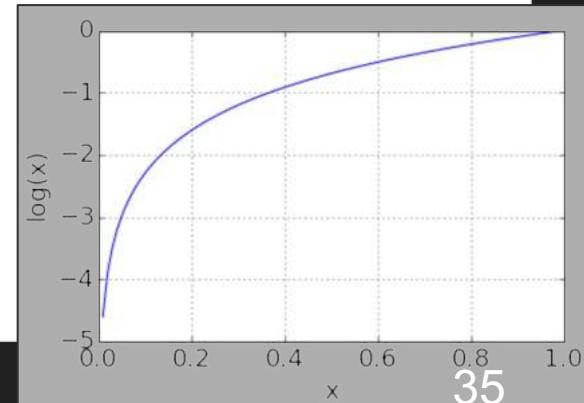
$$D(G(z)) = 0.2$$

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$-0.2 + -0.2 = -0.4$$

end for

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

Discriminator: (input=generated/real image, output=prediction of real image)

$$D(x) = 0.8$$

$$\log(0.8) = -0.2$$

$$D(G(z)) = 0.2$$

$$\log(1-0.2) = \log(0.8) = -0.2$$

end for

$$-0.2 + -0.2 = -0.4$$

$$D(x) = 0.2$$

$$\log(0.2) = -1.6$$

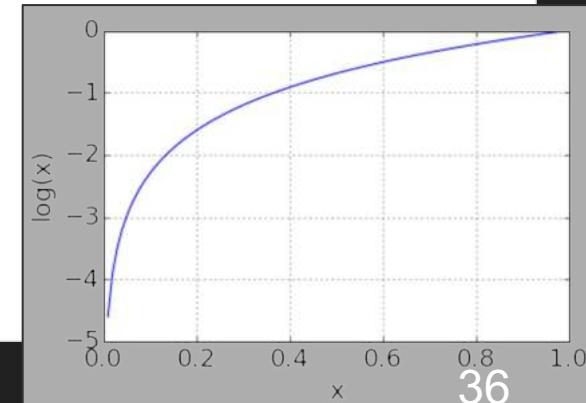
$D(x)$ scores 0.2 that a real image is a real image (bad)

Uniform noise vector (random numbers)

Real images

Generator:
input=random numbers,
output=synthetic image

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

$$D(x) = 0.8$$

$$\log(0.8) = -0.2$$

$D(x)$ scores 0.8 that the generated image is a real image (bad)

$$-0.2 + -0.2 = -0.4$$

maximize

Uniform noise vector (random numbers)

Real images

Generator:
 input=random numbers,
 output=synthetic image

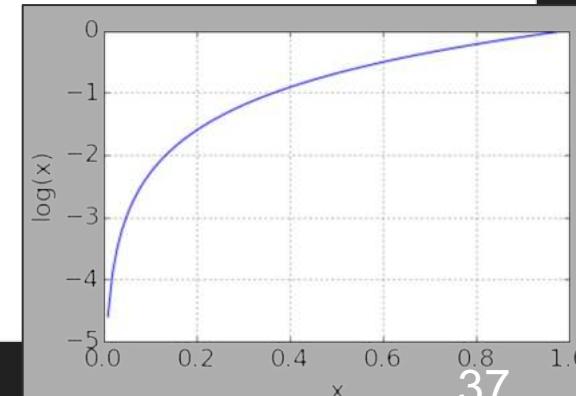
Discriminator: (input=generated/real image,
 output=prediction of real image)

$$D(x) = 0.2$$

$$\log(0.2) = -1.6$$

$D(G(z)) = 0.8$
 $\log(1-0.8) = \log(0.2) = -1.6$

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

$$D(x) = 0.8$$

$$\log(0.8) = -0.2$$

$$D(G(z)) = 0.2$$

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$-0.2 + -0.2 = -0.4$$

end for

Uniform noise vector (random numbers)

maximize

Real images

Generator:

input=random numbers,
output=synthetic image

Discriminator: (input=generated/real image,
output=prediction of real image)

$$D(x) = 0.2$$

$$\log(0.2) = -1.6$$

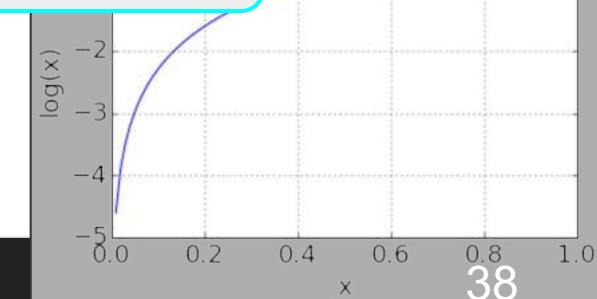
$$D(G(z)) = 0.8$$

$$\log(1-0.8) = \log(0.2) = -1.6$$

$$-1.6 + -1.6 = -3.2$$

Note showing $\ln(x)$ not $\log(x)$

These "bad" predictions combined gives a loss of -3.2



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

Average over m samples

Uniform noise vector (random numbers)

maximize

Real images

Generator:
 input=random numbers,
 output=synthetic image

$$D(x) = 0.8$$

$$\log(0.8) = -0.2$$

$$D(G(z)) = 0.2$$

$$\log(1-0.2) = \log(0.8) = -0.2$$

end for

$$-0.2 + -0.2 = -0.4$$

$$D(x) = 0.2$$

$$\log(0.2) = -1.6$$

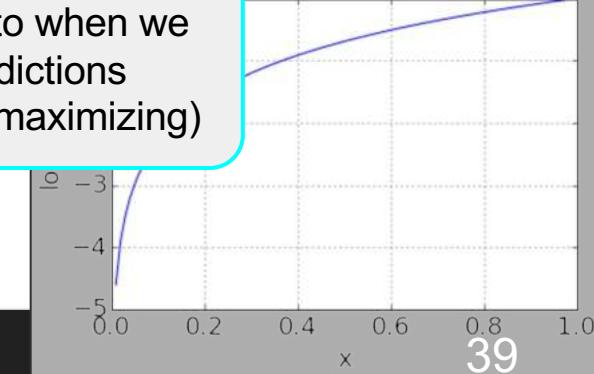
$$D(G(z)) = 0.8$$

$$\log(1-0.8) = \log(0.2) = -1.6$$

$$-1.6 + -1.6 = -3.2$$

Note showing $\ln(x)$ not $\log(x)$

Compare the loss to when we had "good" predictions (remember we are maximizing)



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**
for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

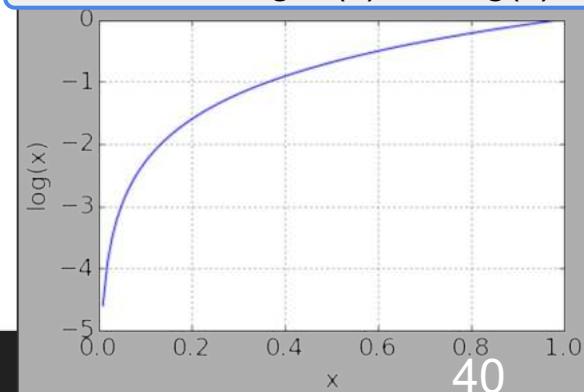
Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

Note showing $\ln(x)$ not $\log(x)$



end for

$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for

Uniform noise vector (random numbers)

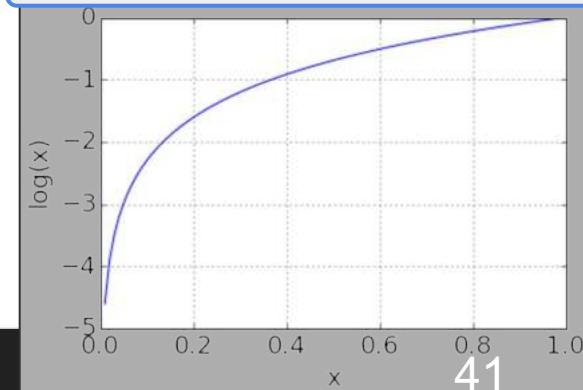
maximize

Real images

Generator:

input=random numbers,
output=synthetic image

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**
for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

$D(G(z)) = 0.2$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

$D(G(z))$ scores 0.2 that a generated image is a real image = bad :(We didn't fool D(.)

end for

Uniform noise vector (random numbers)

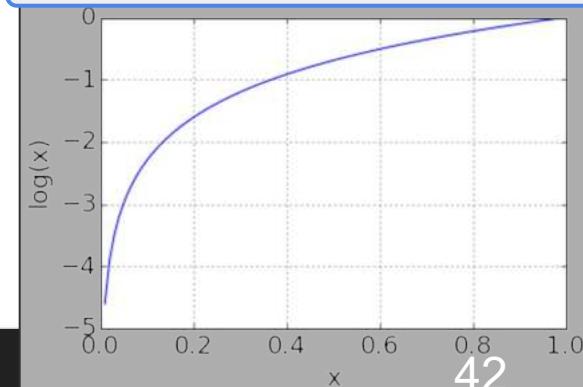
maximize

Real images

Generator:

input=random numbers,
output=synthetic image

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

$$D(G(z)) = 0.2$$

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

Gets assigned a loss of -0.2

end for

Uniform noise vector (random numbers)

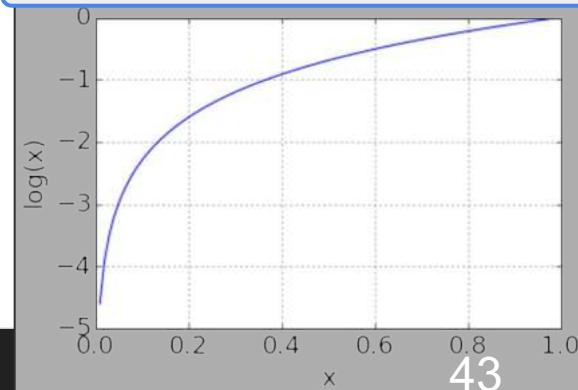
maximize

Real images

Generator:

input=random numbers,
output=synthetic image

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$D(G(z)) = 0.2$$

minimize

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

Notice here we want to minimize this loss function (not maximize like before)

end for

Uniform noise vector (random numbers)

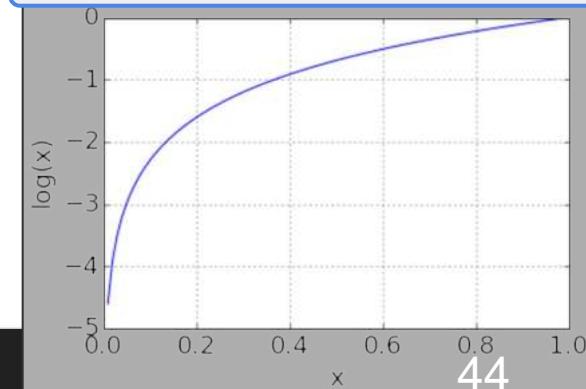
maximize

Real images

Generator:

input=random numbers,
output=synthetic image

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$D(G(z)) = 0.2$$

$$\log(1-0.2) = \log(0.8) = -0.2$$

minimize

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for

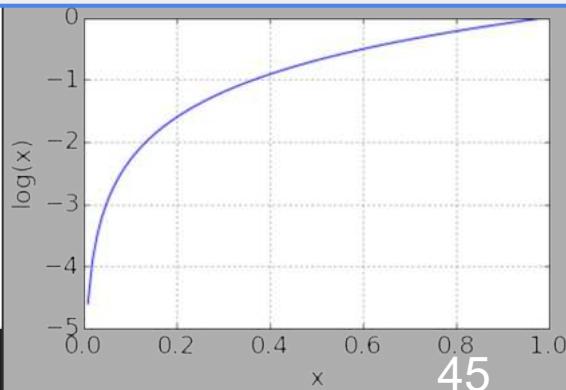
One more example ...

Uniform noise vector (random numbers)

Real images

Generator:
 input=random numbers,
 output=synthetic image

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **ascending** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$D(G(z)) = 0.2$$

minimize

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$D(G(z)) = 0.8$$

end for

Uniform noise vector (random numbers)

maximize

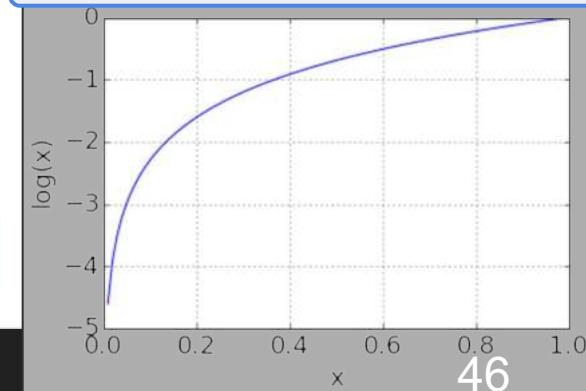
Real images

Generator:
input=random numbers,
output=synthetic image

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

$D(G(z))$ scores 0.8 that a generated image is a real image = good :) We fooled $D(\cdot)$!

Note showing $\ln(x)$ not $\log(x)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **maximize** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$D(G(z)) = 0.2$$

minimize

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$D(G(z)) = 0.8$$

$$\log(1-0.8) = \log(0.2) = -1.6$$

end for

Uniform noise vector (random numbers)

maximize

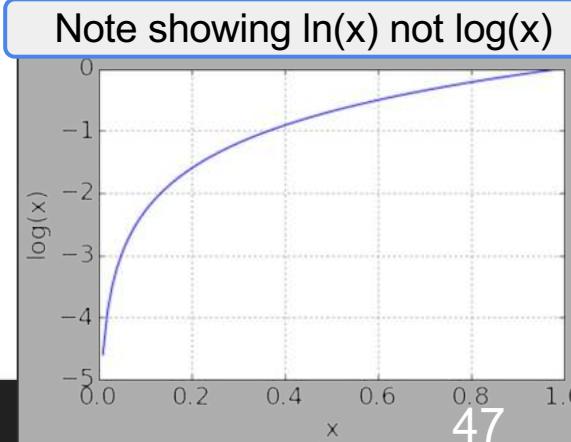
Real images

Generator:

input=random numbers,
output=synthetic image

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

This gives loss of -1.6



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **ascending** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$D(G(z)) = 0.2$$

minimize

$$\log(1-0.2) = \log(0.8) = -0.2$$

$$D(G(z)) = 0.8$$

$$\log(1-0.8) = \log(0.2) = -1.6$$

end for

Uniform noise vector (random numbers)

maximize

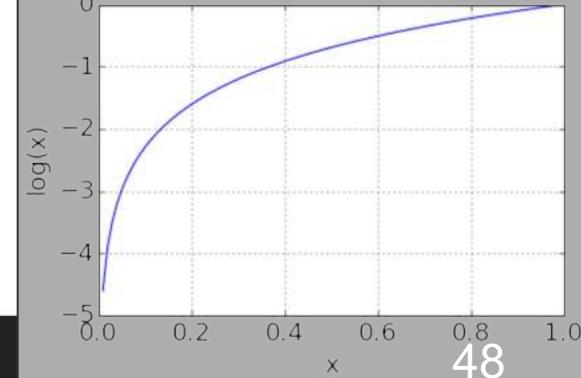
Real images

Generator:

input=random numbers,
output=synthetic image

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

So minimizing this loss means to generate images that fool $D(\cdot)$



$D(x)$ = probability that x is a real image
 0 = generated image; 1 = real image

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **ascending** its stochastic gradient:

Gradient w.r.t the parameters of the Discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

Discriminator: (input=generated/real image, output=prediction of real image)

end for

Average over m samples

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

minimize

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

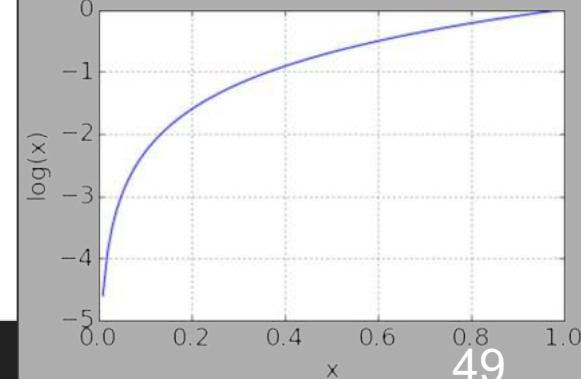
end for

Gradient w.r.t the parameters of the Generator

Uniform noise vector (random numbers)

Real images

Generator:
input=random numbers,
output=synthetic image



Results from GAN



a)



b)



c)



d)

Training a system for rendering arbitrary images (arbitrary within the bounds of the subject matter that we trained the system on)

Example 1

Design a GAN to learn Gaussian distributed data with mean = 1.0 and standard deviation = 1.0. The generator receives uniformly distributed 1-dimensional inputs in the range [-1, +1].

Discriminator is a 4-layer DNN:

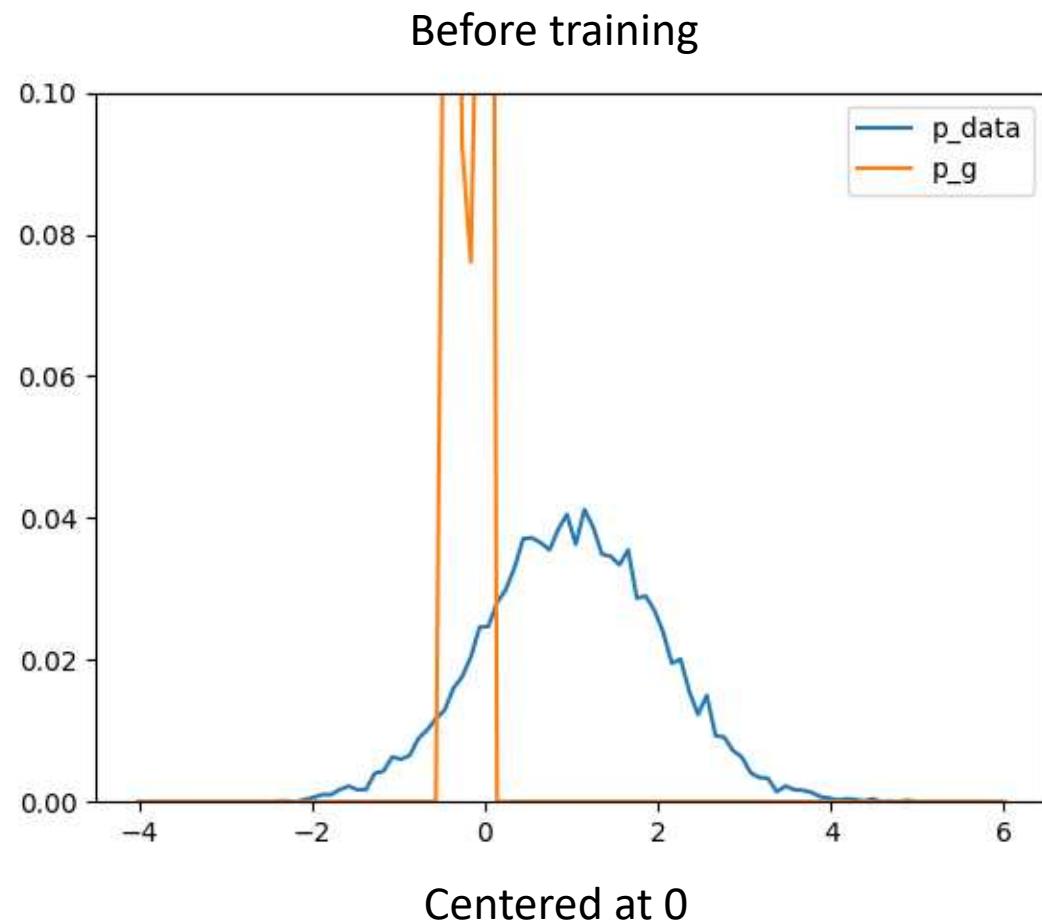
- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 10 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 10 (activation: Tanh)
- Output dimension = 1 (binary classification)

Generator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 5 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 5 (activation: Tanh)
- Output dimension = 1 (activation: None)

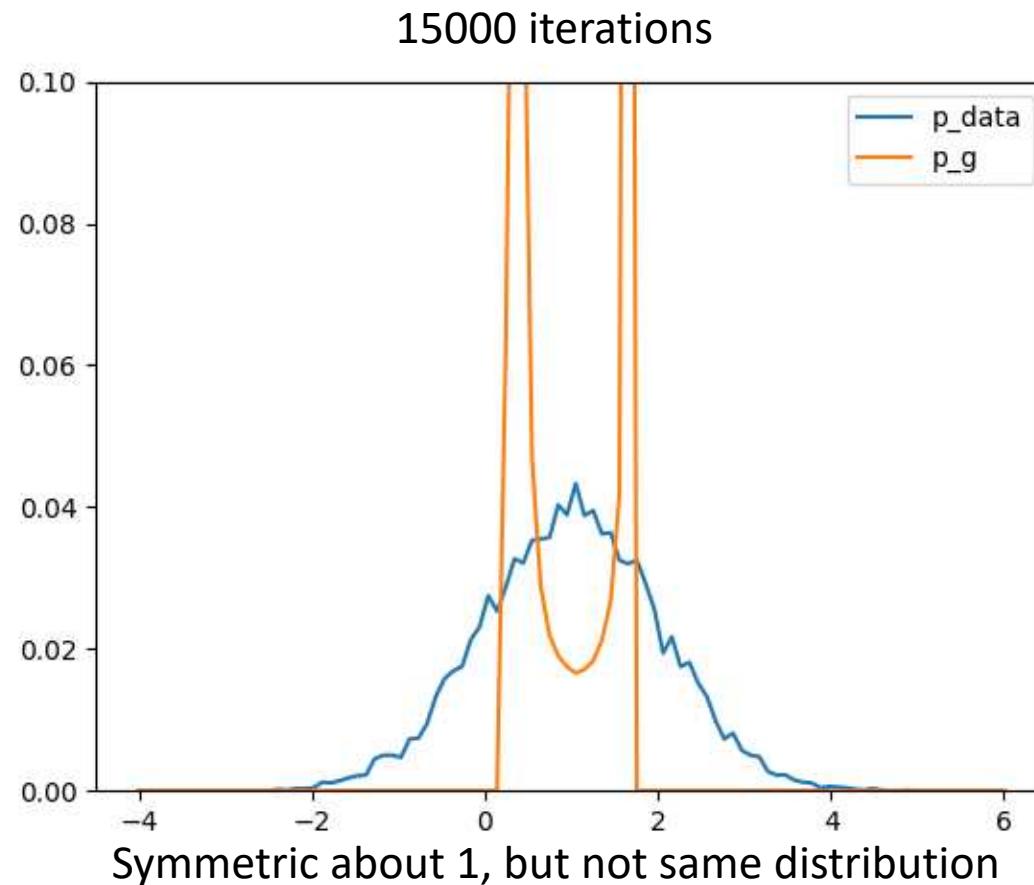
Example 1

Histogram of 10000 randomly drawn points from $U[-1, 1]$



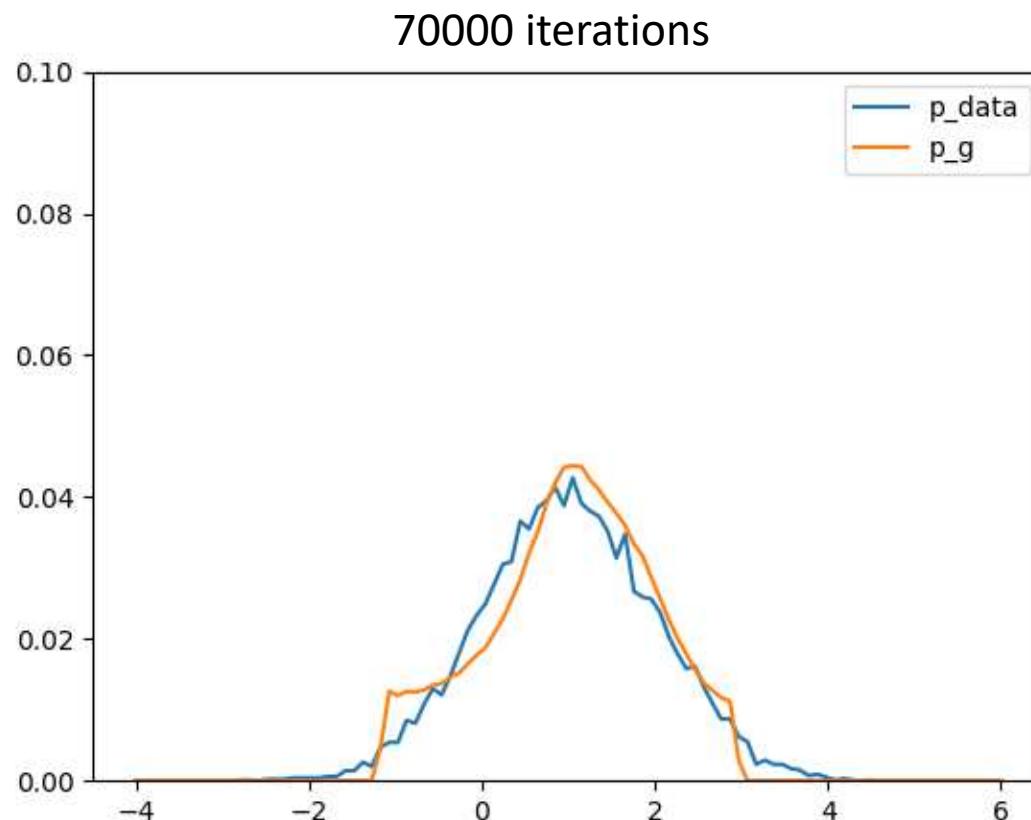
Example 1

Histogram of 10000 randomly drawn points from $U[-1, 1]$



Example 1

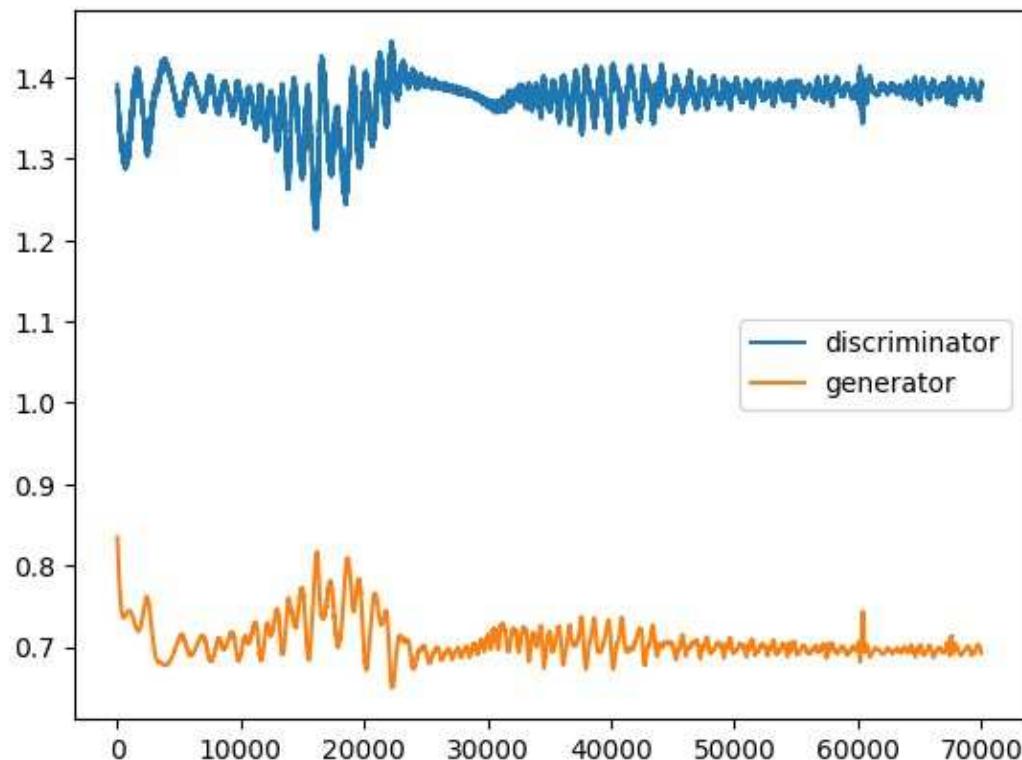
Histogram of 10000 randomly drawn points from $U[-1, 1]$



Not perfect, but somehow similar to the data distribution

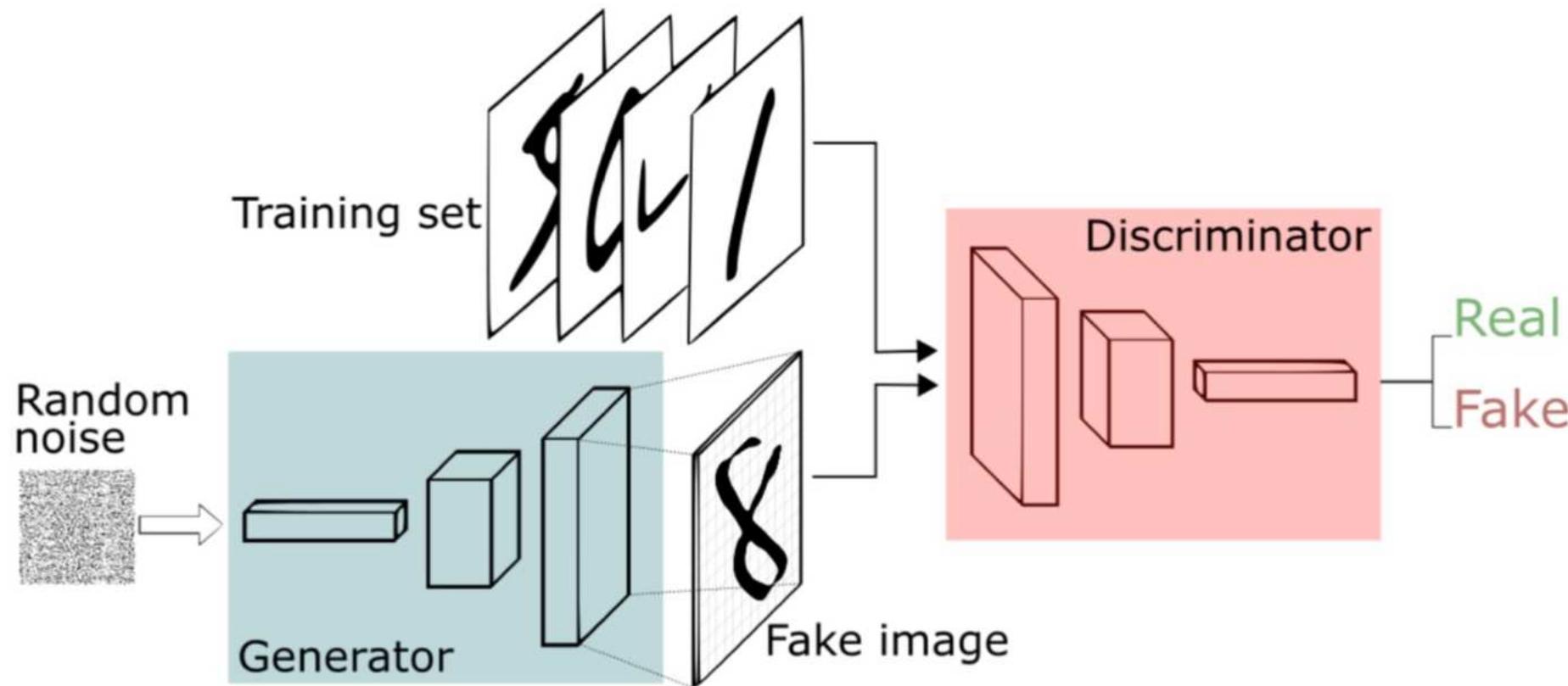
Example 1

Training Curves



Example 2

Design a GAN to generate MNIST images from a uniformly distributed noise vector of 100 dimensions.



Example 2

Generator:

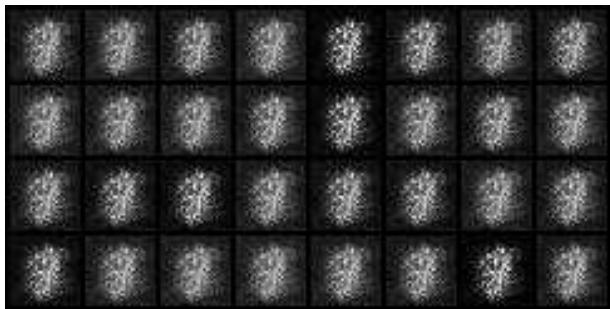
- Input dimension = 100, drawn from a uniform distribution $U(-1, 1)$
- Hidden layers
 - Number of hidden neurons = 256 (activation: ReLU)
 - Number of hidden neurons = 512 (activation: ReLU)
 - Number of hidden neurons = 1024 (activation: ReLU)
- Output dimension = 784 (activation: Tanh)

Discriminator:

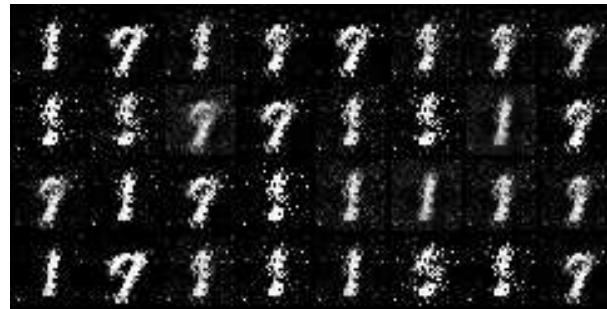
- Input dimension = 784
- Hidden layers
 - Number of hidden neurons = 1024 (activation: ReLU)
 - Number of hidden neurons = 512 (activation: ReLU)
 - Number of hidden neurons = 256 (activation: ReLU)
- Output dimension = 1 (activation: Sigmoid, binary classification)

Example 2

Start of training



10th epoch



20th epoch



30th epoch



40th epoch



50th epoch

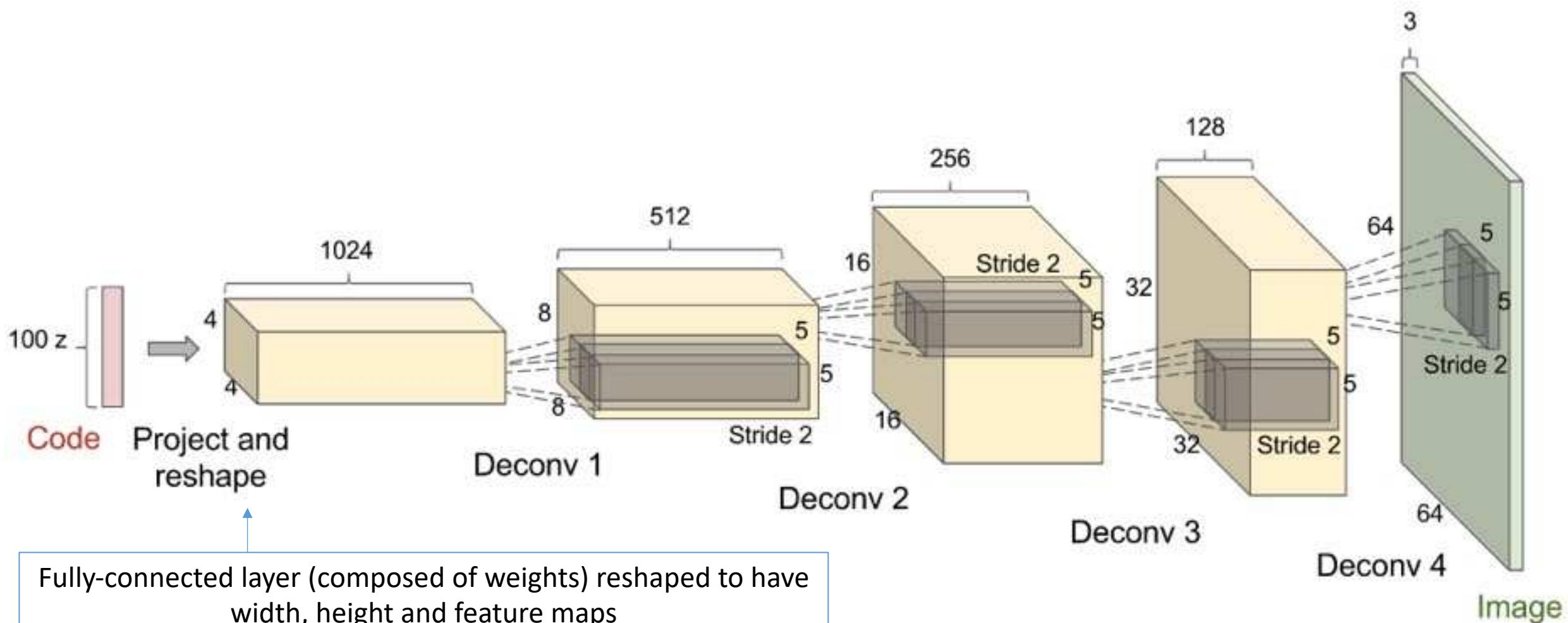


Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

DCGAN

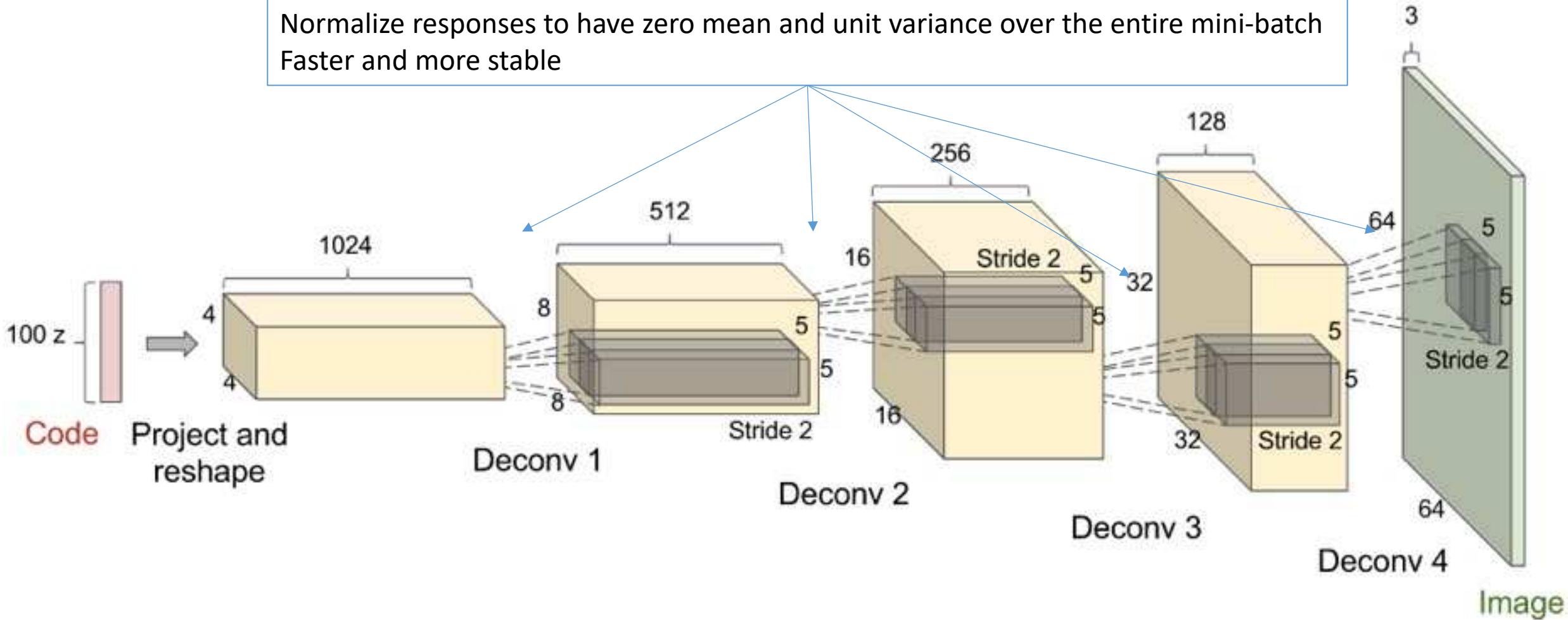
Deep Convolutional GAN (DCGAN)



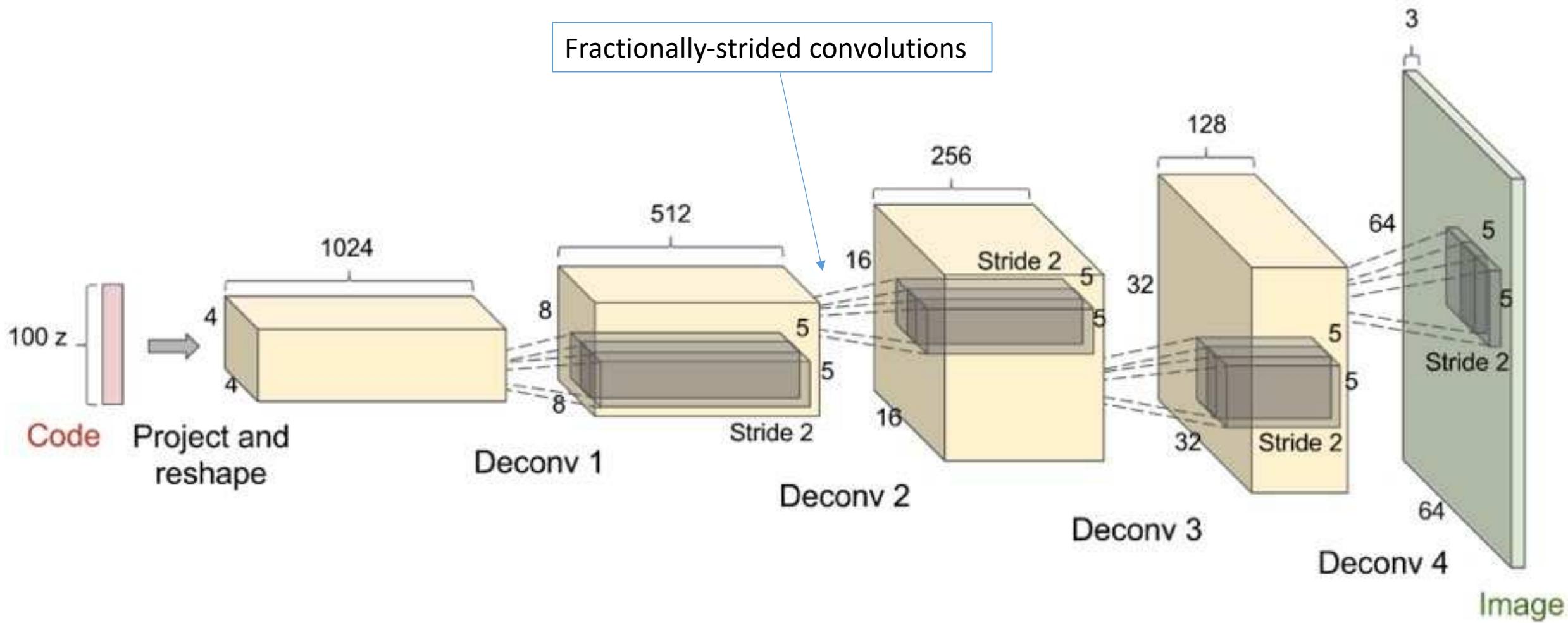
We will go through the code in the tutorial

Deep Convolutional GAN (DCGAN)

Most “deconv”s are batch normalized:
Normalize responses to have zero mean and unit variance over the entire mini-batch
Faster and more stable

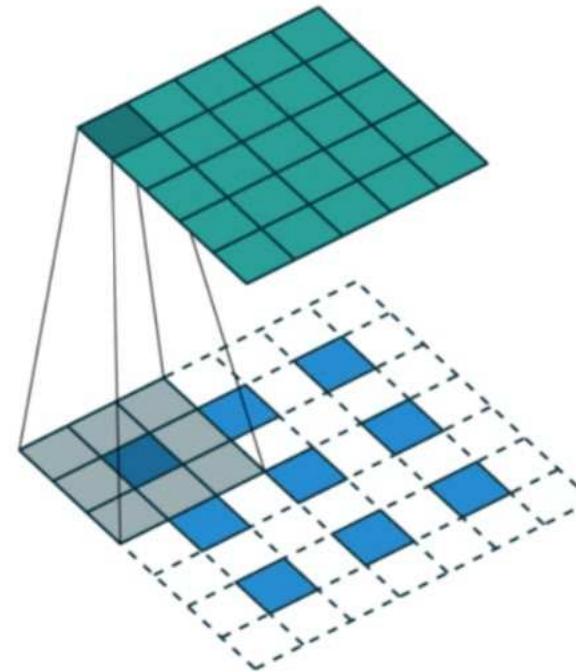


Deep Convolutional GAN (DCGAN)



Fractionally-strided convolutions

Fractional strides involve: zeros are inserted *between* input units, which makes the kernel move around at a slower pace than with unit strides



<https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html>

Up-sampling by fractional striding

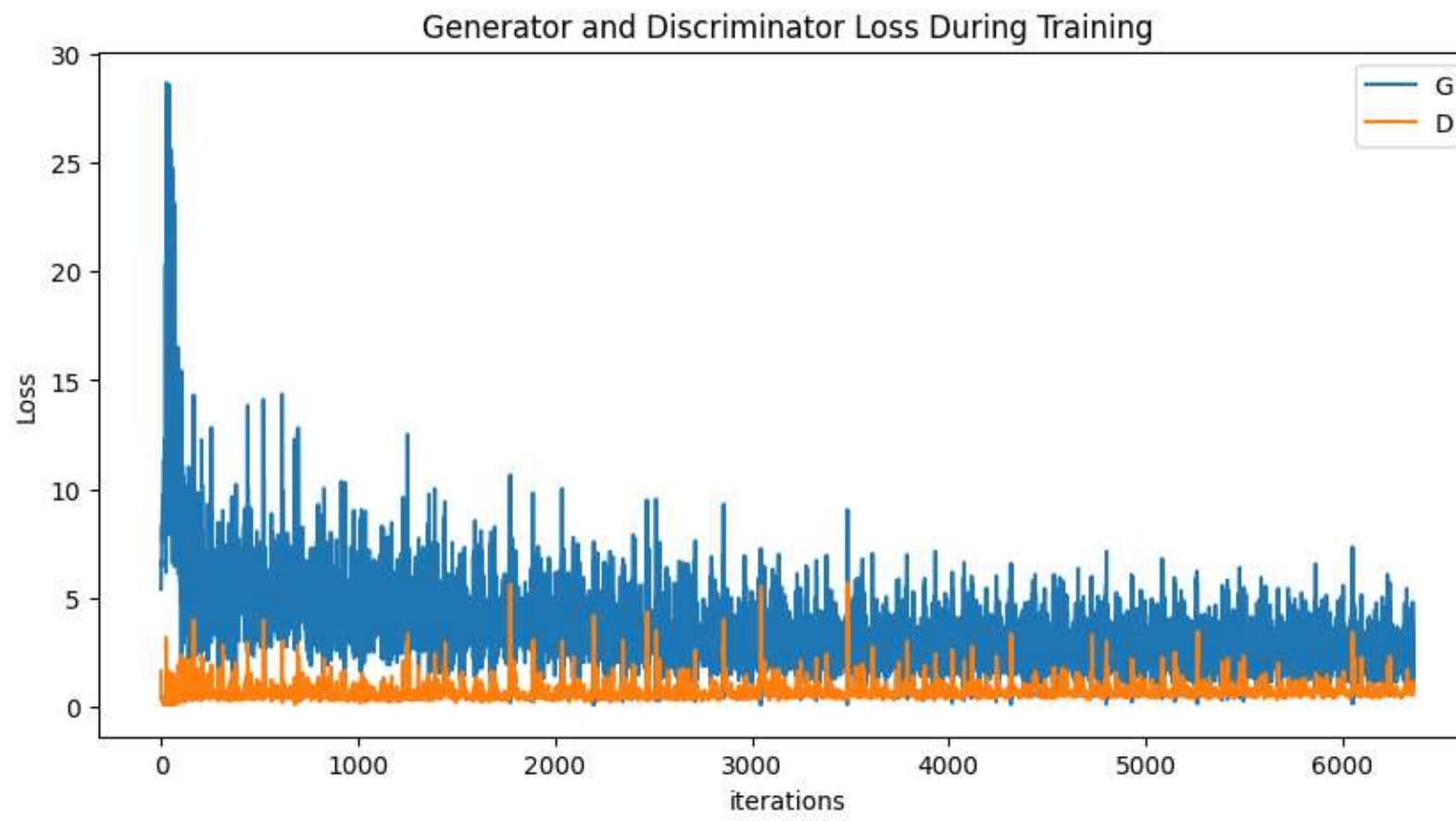
Input 3x3, output = 5x5

stride = 1 convolution window = 3 x 3 (with respect to output)

Other tricks proposed by DCGAN

- Adam optimizer (adaptive moment estimation) = similar to SGD but with less parameter tuning
- Momentum = 0.5 (usually is 0.9 but training oscillated and was unstable)
- Low learning rate = 0.0002

DCGAN

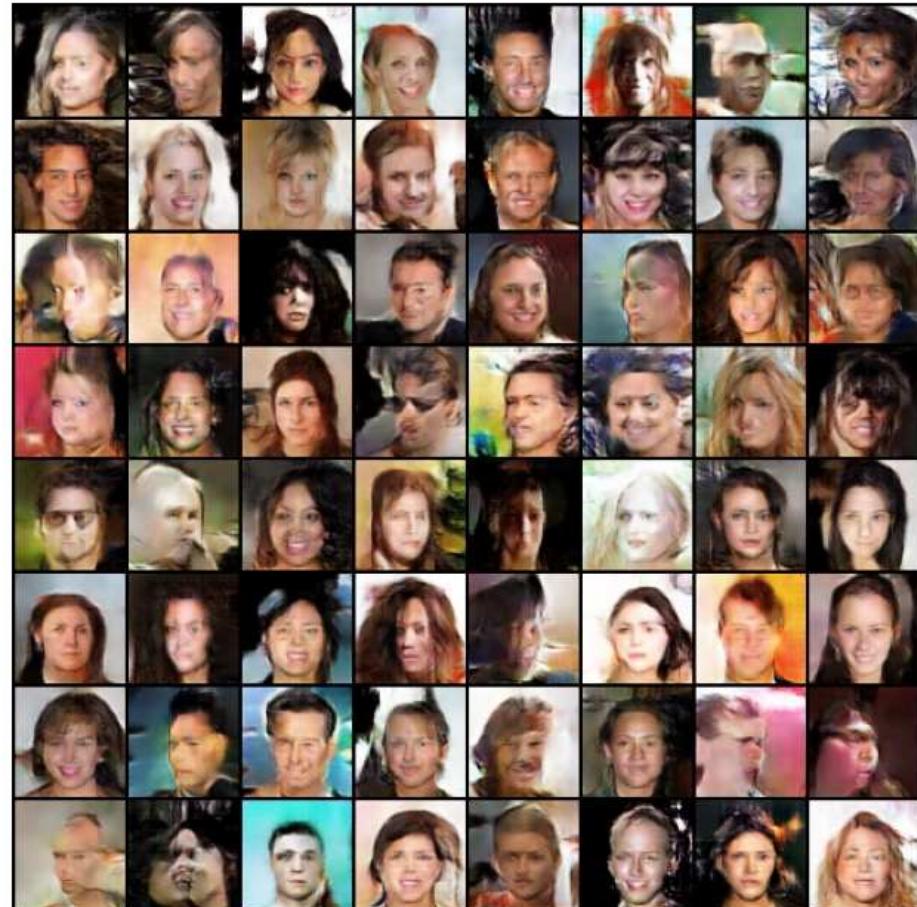


DCGAN

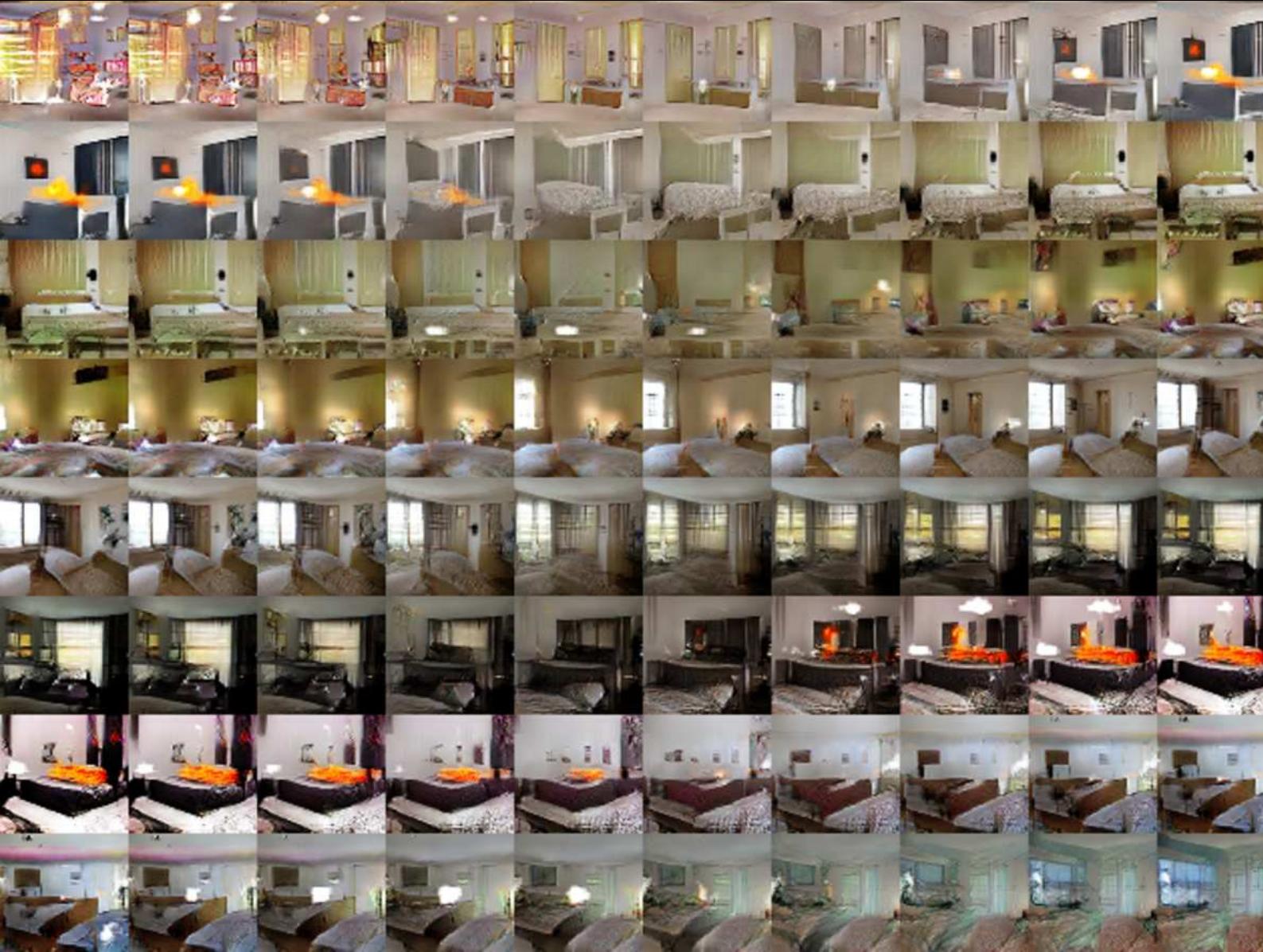
Real Images



Fake Images



Sanity check by walking on the manifold



Interpolating between a series of random points in \mathbf{z}

(\mathbf{z} = the 100-d random numbers)

e.g.,

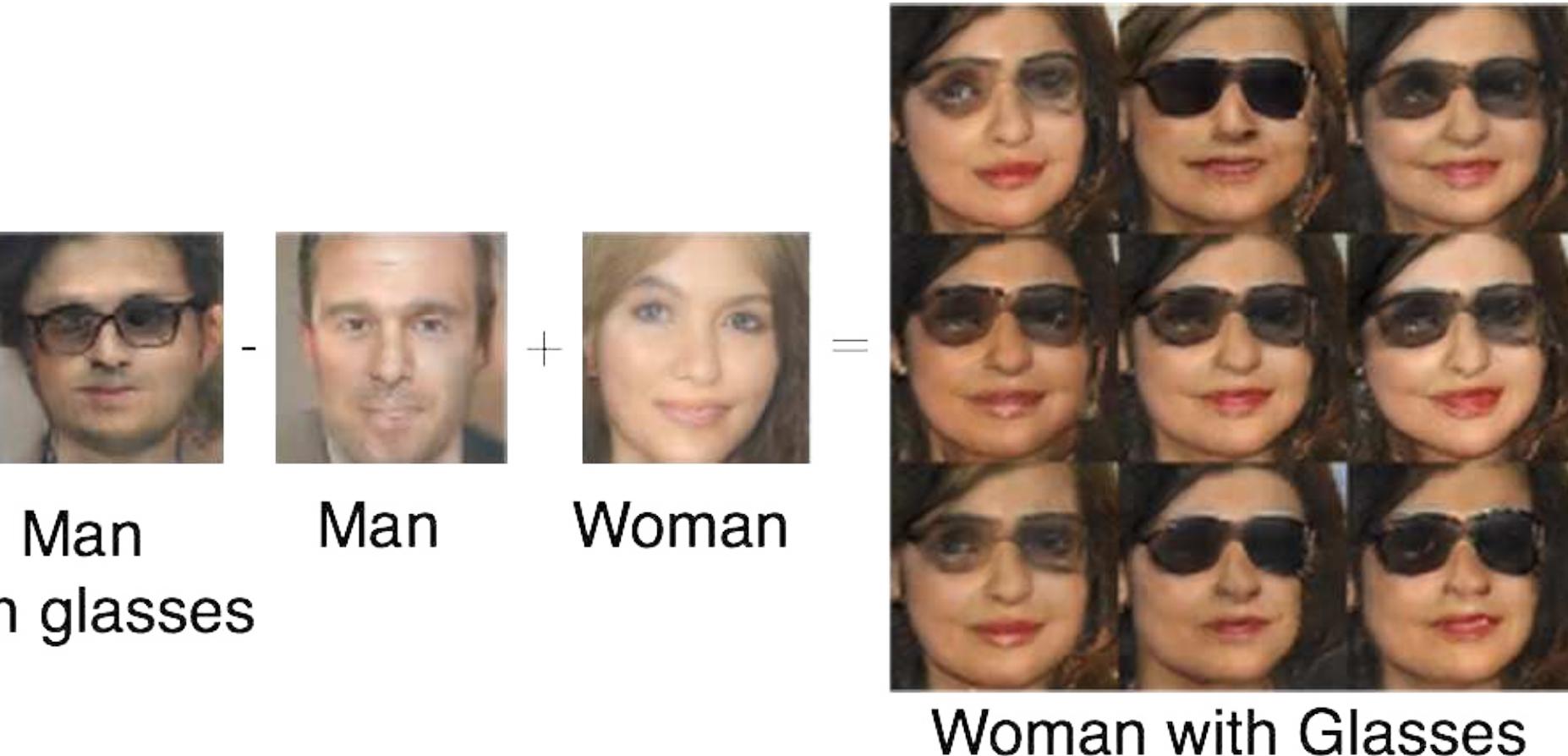
$$\mathbf{z}_{new} = m \mathbf{z}_1 + (1 - m) \mathbf{z}_2$$

Note the smooth transition between each scene

←the window slowly appearing

This indicates that the space learned has smooth transitions!
(and is not simply memorizing)

Sanity check by vector space arithmetic

$$\text{Man with glasses} - \text{Man} + \text{Woman} = \text{Woman with Glasses}$$


The diagram illustrates vector space arithmetic using facial images. On the left, three images are shown: a man with glasses, a man, and a woman. Below them are their respective labels: "Man with glasses", "Man", and "Woman". To the right of the images is an equals sign. To the right of the equals sign is a collage of six images of a woman wearing sunglasses, representing the result of the arithmetic operation.

Outline

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Mode Collapse

Advantages of GANs

- **Plenty of existing work on Deep Generative Models**
 - Boltzmann Machine
 - Deep Belief Nets
 - Variational AutoEncoders (VAE)
- **Why GANs?**
 - Sampling (or generation) is straightforward.
 - Training doesn't involve Maximum Likelihood estimation (i.e. finding the best model parameters that fit the training data the most), believed to cause poorer quality of images (blurry images)
 - Robust to overfitting since the generator never sees the training data.

Problems with GAN

- **Probability Distribution is Implicit**
 - Not straightforward to compute $p(x)$
 - Thus **Vanilla GANs** are only good for Sampling/Generation.
- **Training is Hard**
 - Non-Convergence
 - Mode-Collapse

Non-convergence

- Deep Learning models (in general) involve a single player
 - The player tries to maximize its reward (minimize its loss).
 - Use SGD (with Backpropagation) to find the optimal parameters.
 - SGD has convergence guarantees (under certain conditions).
 - **Problem:** With non-convexity, we might converge to local optima.

$$\min_G L(G)$$

- GANs instead involve two (or more) players
 - Discriminator is trying to maximize its reward.
 - Generator is trying to minimize Discriminator's reward.

$$\min_G \max_D V(D, G)$$

- SGD was not designed to find the Nash equilibrium of a game.
- **Problem:** We might not converge to the Nash equilibrium at all.

Non-convergence example

$$\min_x \max_y V(x, y)$$

Let $V(x, y) = xy$

- State 1:

x > 0	y > 0	V > 0
-------	-------	-------

Increase y	Decrease x
------------	------------

- State 2:

x < 0	y > 0	V < 0
-------	-------	-------

Decrease y	Decrease x
------------	------------

- State 3:

x < 0	y < 0	V > 0
-------	-------	-------

Decrease y	Increase x
------------	------------

- State 4 :

x > 0	y < 0	V < 0
-------	-------	-------

Increase y	Increase x
------------	------------

- State 5:

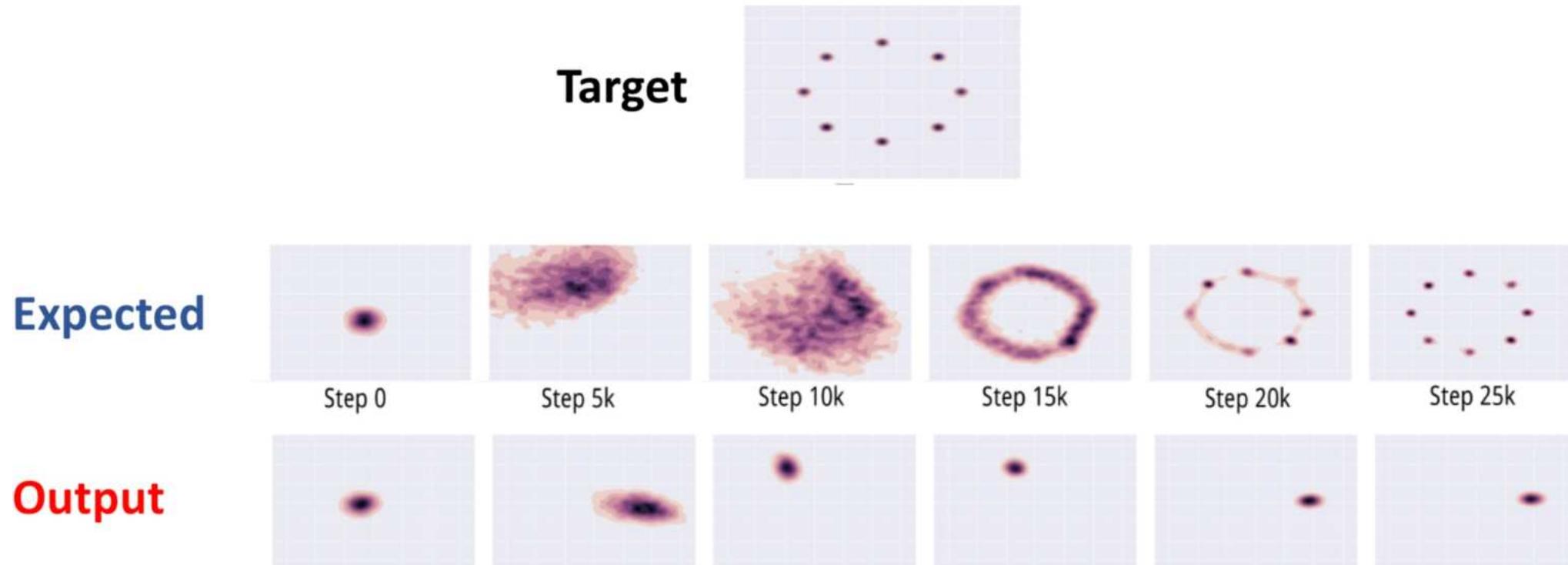
x > 0	y > 0	V > 0
-------	-------	-------

== State 1

Increase y	Decrease x
------------	------------

Mode Collapse

Generator fails to output diverse samples



How to reward sample diversity

- **At Mode Collapse,**

- Generator produces good samples, but a very few of them.
- Thus, Discriminator can't tag them as fake.

- **To address this problem,**

- Let the Discriminator know about this edge-case.

- **More formally,**

- Let the Discriminator look at the entire batch instead of single examples
- If there is lack of diversity, it will mark the examples as fake

- **Thus,**

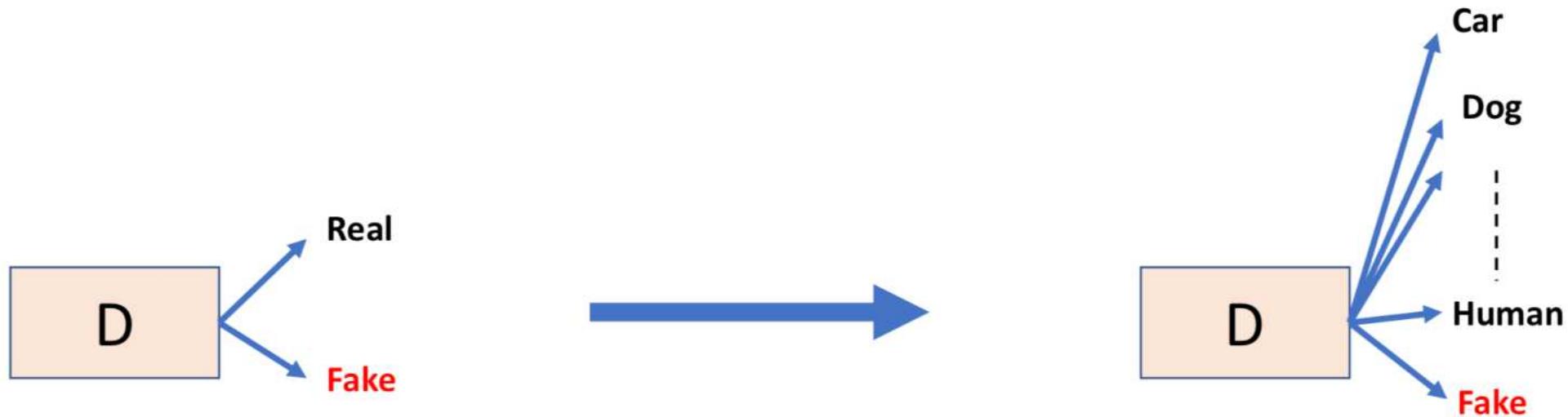
- Generator will be forced to produce diverse samples.

Mini-Batch GANs

- **Extract features that capture diversity in the mini-batch**
 - For e.g. L2 norm of the difference between all pairs from the batch
 - That is, the errors at an intermediate layer for real and fake samples are minimized.
- **Feed those features to the discriminator along with the image**
- **Feature values will differ b/w diverse and non-diverse batches**
 - Thus, Discriminator will rely on those features for classification
- **This in turn,**
 - Will force the Generator to match those feature values with the real data
 - Will generate diverse batches

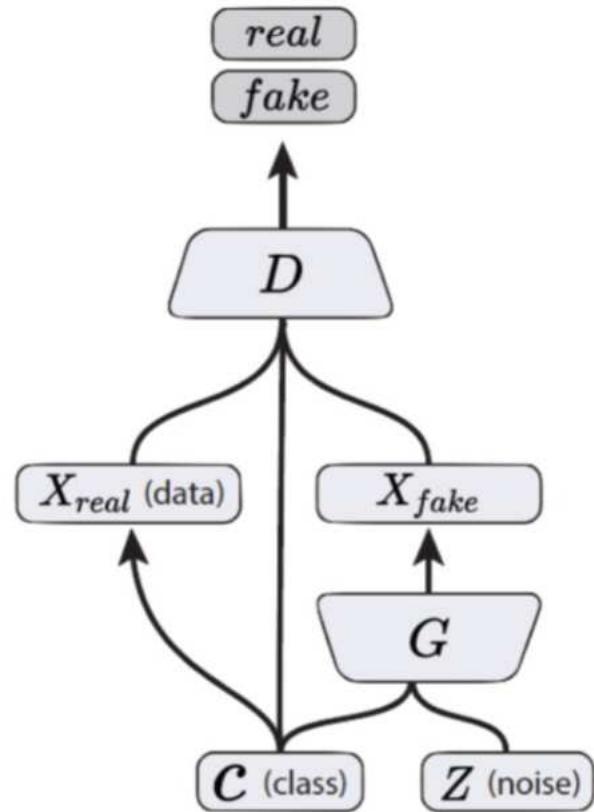
Supervision with Labels

- Label information of the real data might help



- Empirically generates much better samples

Conditional GANs



Conditions the inputs to the generator and discriminator with the labels

Conditional GAN
(Mirza & Osindero, 2014)

Conditional GANs

MNIST digits generated conditioned on their class label.

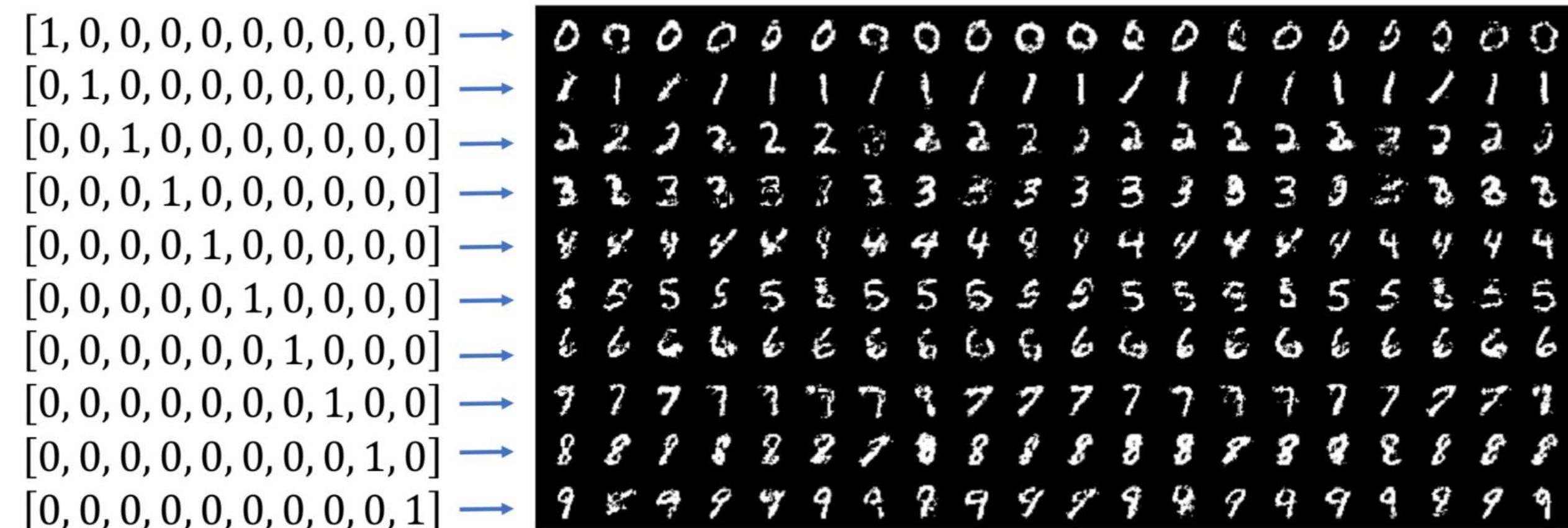
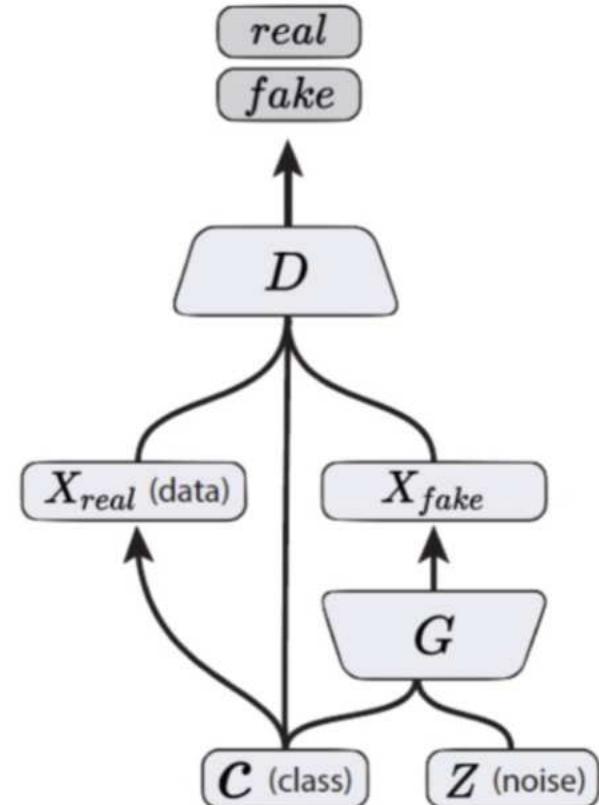


Figure 2 in the original paper.

Conditional GANs

- Simple modification to the original GAN framework that conditions the model on *additional information* for better multi-modal learning.
- Lends to many practical applications of GANs when we have explicit *supervision* available.



Conditional GAN
(Mirza & Osindero, 2014)

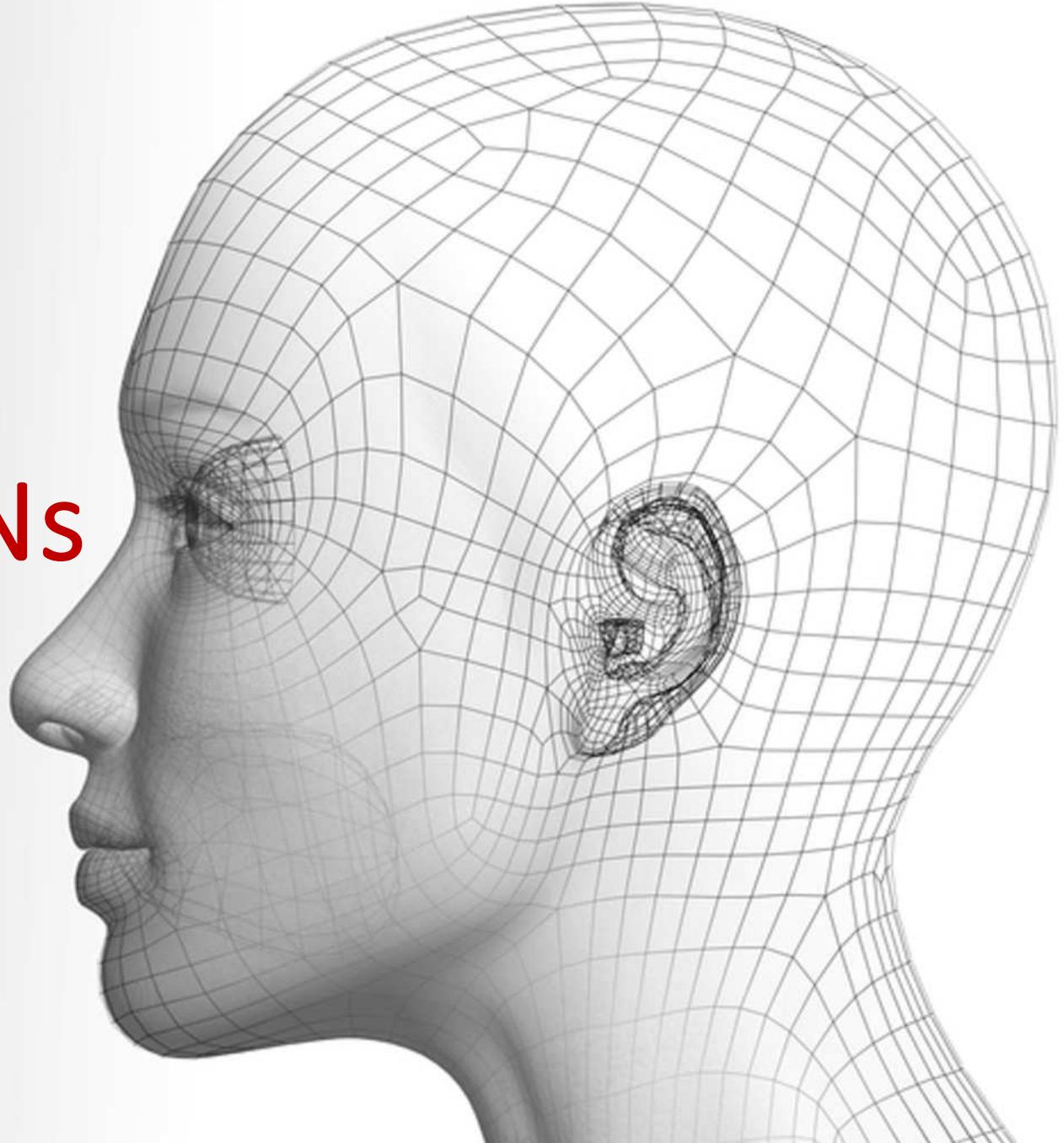
Revision & Applications of GANs

Xingang Pan

潘新钢

<https://xingangpan.github.io/>

<https://twitter.com/XingangP>



Revision

- **Week 7** – Convolutional Neural Network (CNN) I
- **Recess Week**
- **Week 8** – Convolutional Neural Network (CNN) II
- **Week 9** – Recurrent Neural Networks (RNN)
- **Week 10** – Attention
- **Week 11** – Autoencoders
- **Week 12** – Generative Adversarial Networks (GAN)
- **Week 13** – Revision and Selected Topics

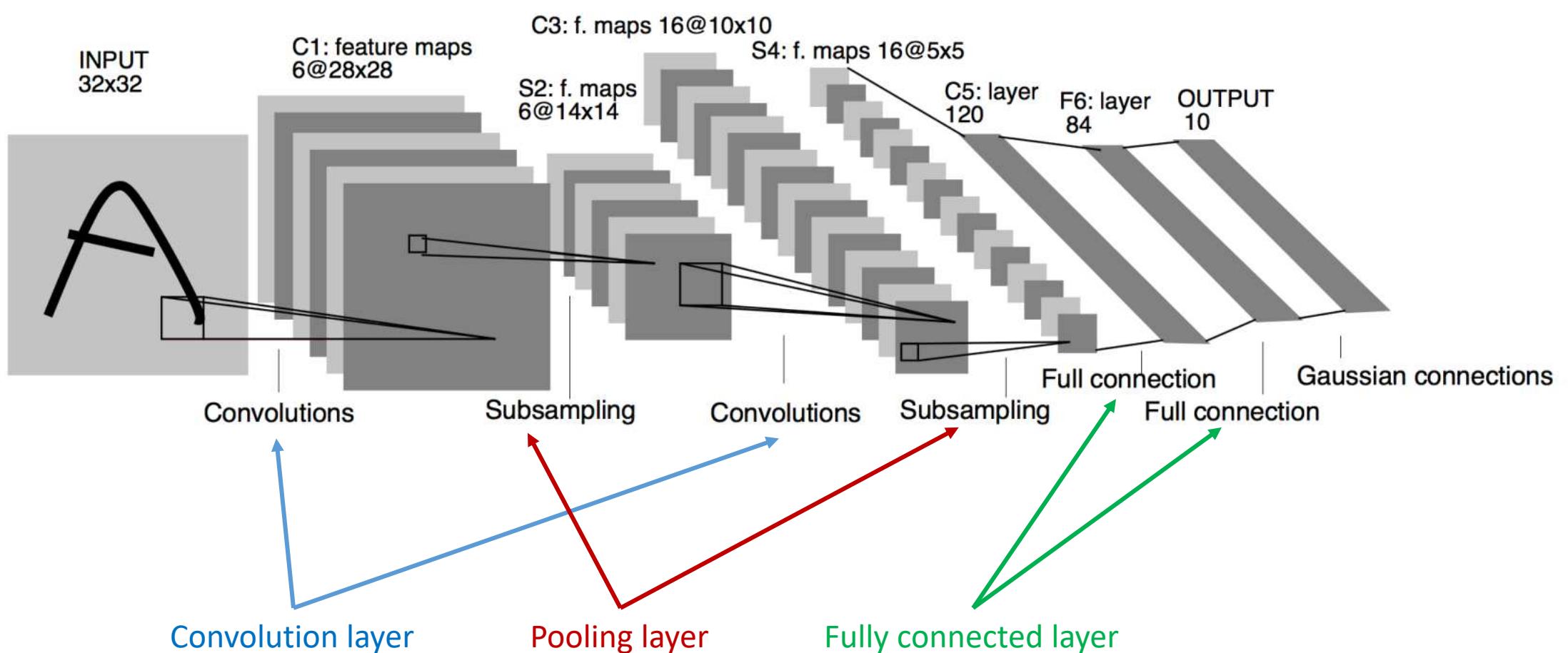
Final Exam

- Time: 9:00 AM – 11:00 AM, 7 May 2024
- Venue: Hall C
- The exam will be open book

Outline – CNN I

- Basic components in CNN
- Training a classifier
- Optimizers

An example of convolutional network: LeNet 5

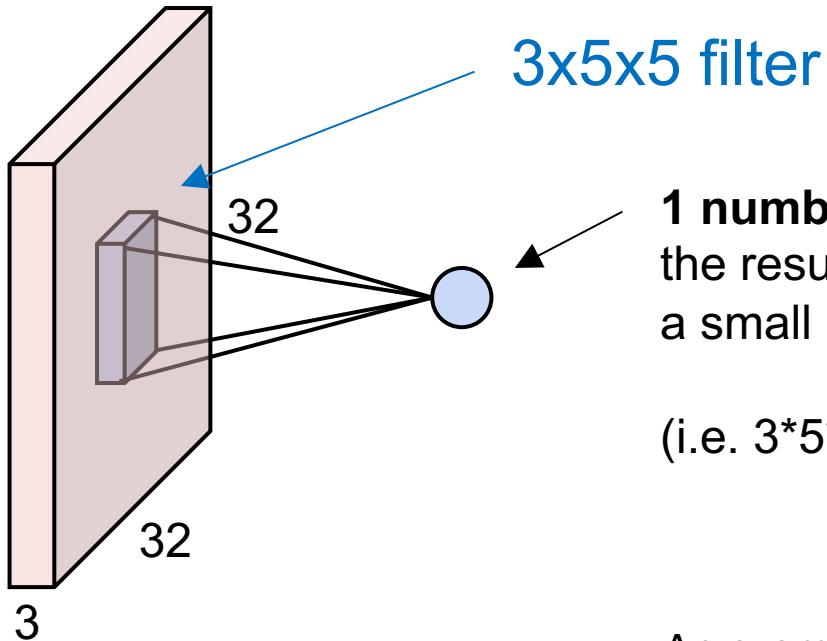


As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

Common layer arrangement: Conv → pool → Conv → pool → fully connected → fully connected → output

Convolution layer

3x32x32 image



1 number:

the result of taking a dot product between the filter and a small 3x5x5 chunk of the image

(i.e. $3 \times 5 \times 5 = 75$ -dimensional dot product + bias)

An example of convolving
a 1x5x5 image with a
1x3x3 filter

1	0	1
0	1	0
1	0	1

Filter (weights or kernel)

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

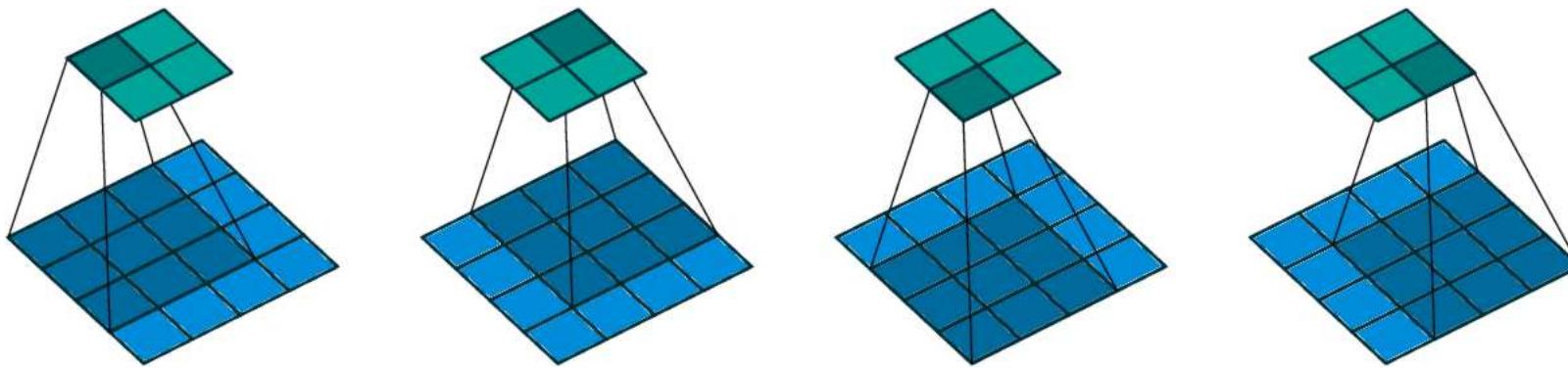
Image

4		

Convolved
Feature

Convolution layer

Convolution by doing a sliding window

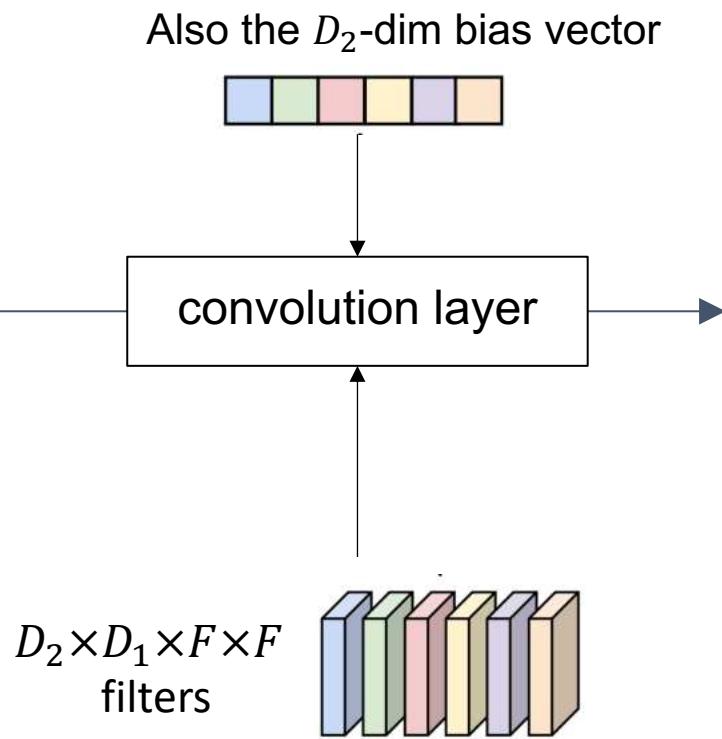
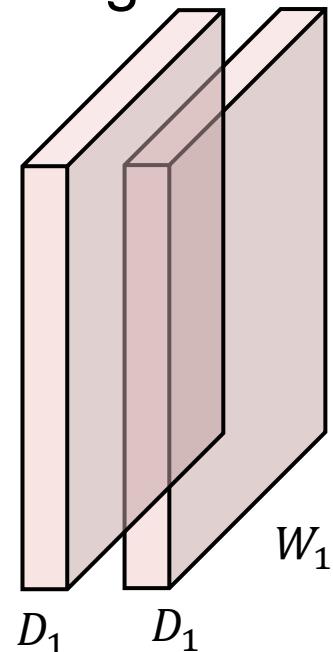


As a guiding example, let us consider the convolution of single-channel tensors $\mathbf{x} \in \mathbb{R}^{4 \times 4}$ and $\mathbf{w} \in \mathbb{R}^{3 \times 3}$:

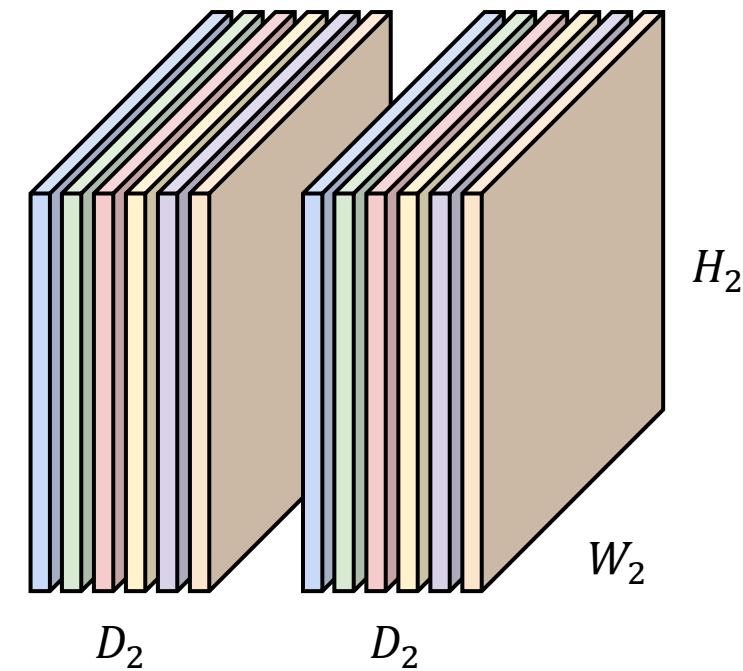
$$\mathbf{w} * \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

Convolution layer

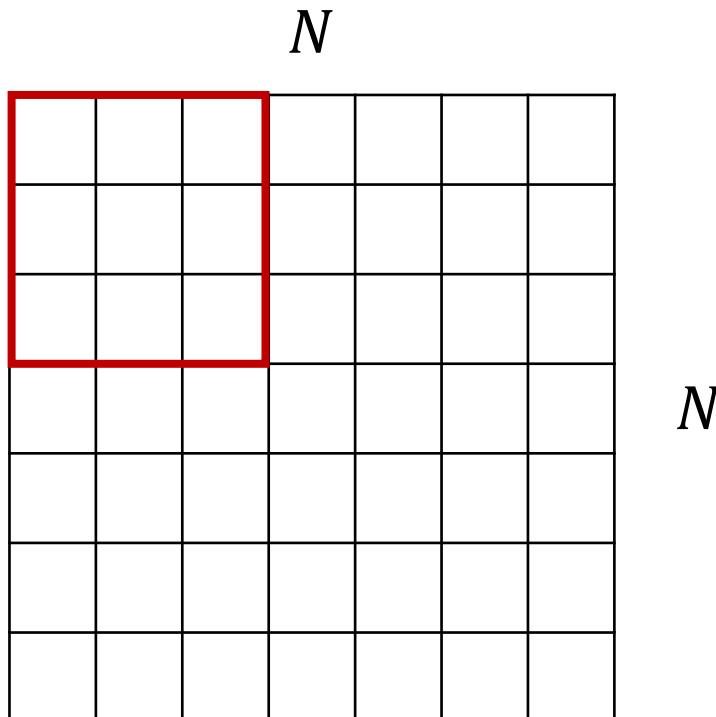
$N \times D_1 \times H_1 \times W_1$ batch of images



$N \times D_2 \times H_2 \times W_2$ batch of activation/feature maps



Convolution layer – spatial dimensions



$N \times N$ input (spatially), assume $F \times F$ filter, and S stride

$$\text{Output size} = \frac{N-F}{S} + 1$$

e.g.

$$N = 7, F = 3$$

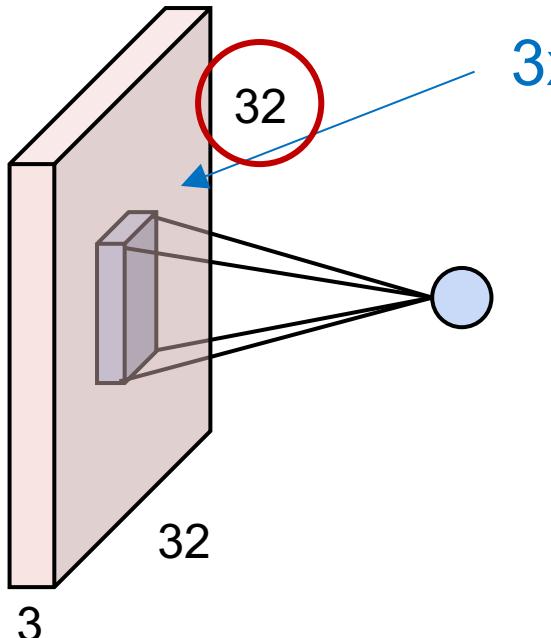
$$\text{stride } 1 \Rightarrow (7 - 3)/1 + 1 = 5$$

$$\text{stride } 2 \Rightarrow (7 - 3)/2 + 1 = 3$$

$$\text{stride } 3 \Rightarrow (7 - 3)/3 + 1 = 2.33 : \backslash$$

Convolution layer – spatial dimensions

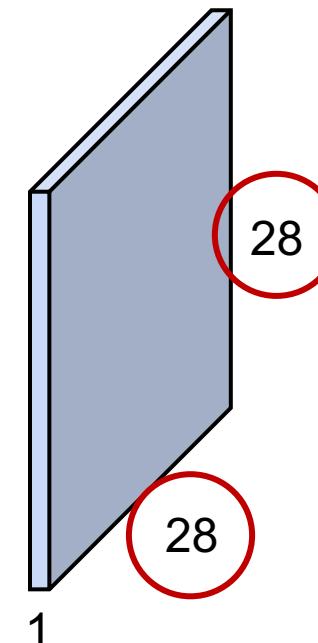
3x32x32 image



3x5x5 filter

convolve (slide) over all
spatial locations

1x28x28
activation/feature map



Why does the feature map
has a size of 28x28?

$$\text{Output size} = \frac{N-F}{S} + 1$$

$$N = 32, F = 5$$

$$\text{stride } 1 \Rightarrow (32 - 5)/1 + 1 = 28$$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In **PyTorch**, by default, we pad top, bottom, left, right with zeros (this can be customized, e.g., 'constant', 'reflect', and 'replicate').

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output shape?

Recall that without padding, output size = $\frac{N-F}{S} + 1$

With padding, output size = $\frac{N-F+2P}{S} + 1$

e.g.

$$N = 7, F = 3$$

$$\text{stride 1} \Rightarrow (7 - 3 + 2(1))/1 + 1 = 7$$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In **PyTorch**, by default, we pad top, bottom, left, right with zeros (this can be customized).

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output shape?

7×7 output!

In general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F - 1)/2$. (**will preserve size spatially**)

e.g. $F = 3 \Rightarrow$ zero pad with 1

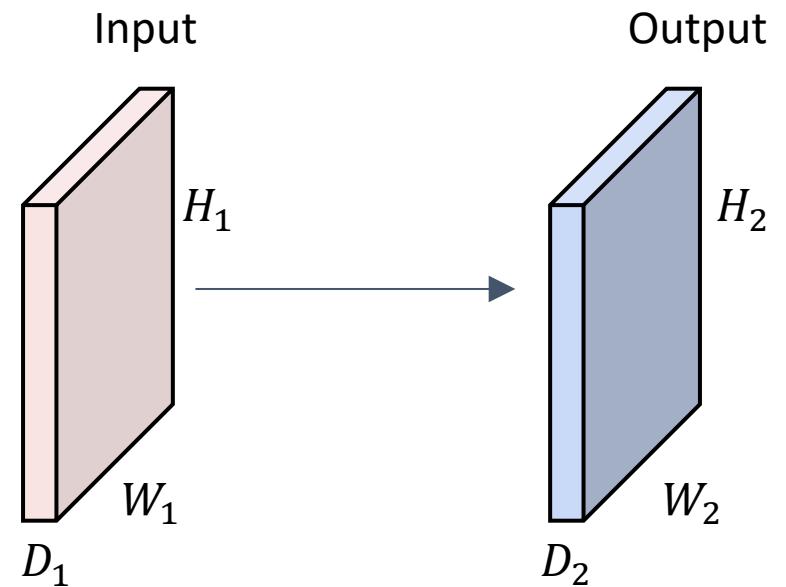
$F = 5 \Rightarrow$ zero pad with 2

$F = 7 \Rightarrow$ zero pad with 3

Convolution layer - summary

A convolution layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
 - Number of filters K
 - Their spatial extent F
 - The stride S
 - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $W_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d -th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias



Pooling layer - summary

A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
 - Their spatial extent F
 - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $W_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

Outline – CNN I

- Basic components in CNN
- Training a classifier
- Optimizers

Outline – CNN II

- CNN Architectures
 - You learn some classic architectures
- More on convolution
 - How to calculate FLOPs
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution
 - You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network
- Batch normalization
 - You learn an important technique to improve the training of modern neural networks
- Prevent overfitting
 - Transfer learning
 - Data augmentation
 - You learn two important techniques to prevent overfitting in neural networks

How to calculate the computations of Convolution?

FLOPs (floating point operations)

Not FLOPS (floating point operations per second)

Assume:

- Filter size F
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Produces a output volume of size $D_2 \times H_2 \times W_2$

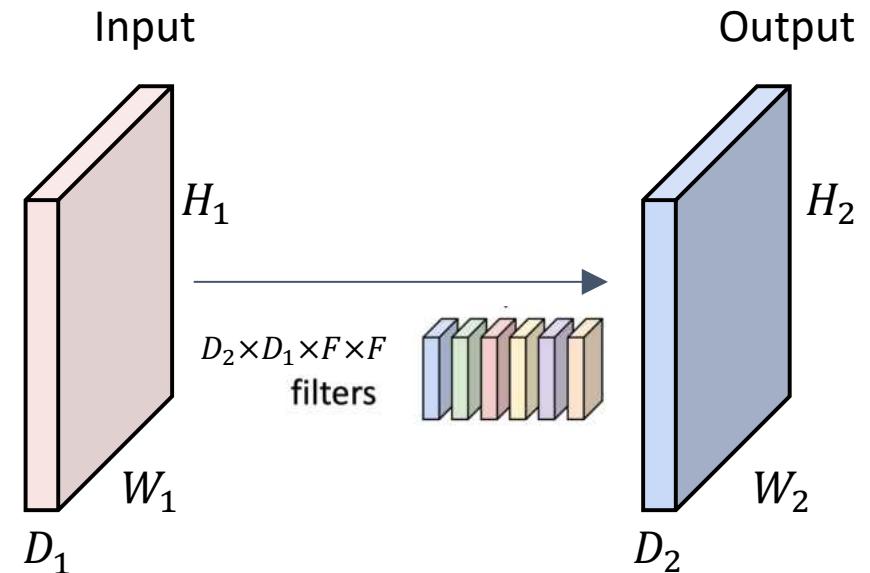
The FLOPs of the convolution layer is given by

$$\text{FLOPs} = [(D_1 \times F^2) + (D_1 \times F^2 - 1) + 1] \times D_2 \times H_2 \times W_2 = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

Elementwise multiplication of each filter on a spatial location

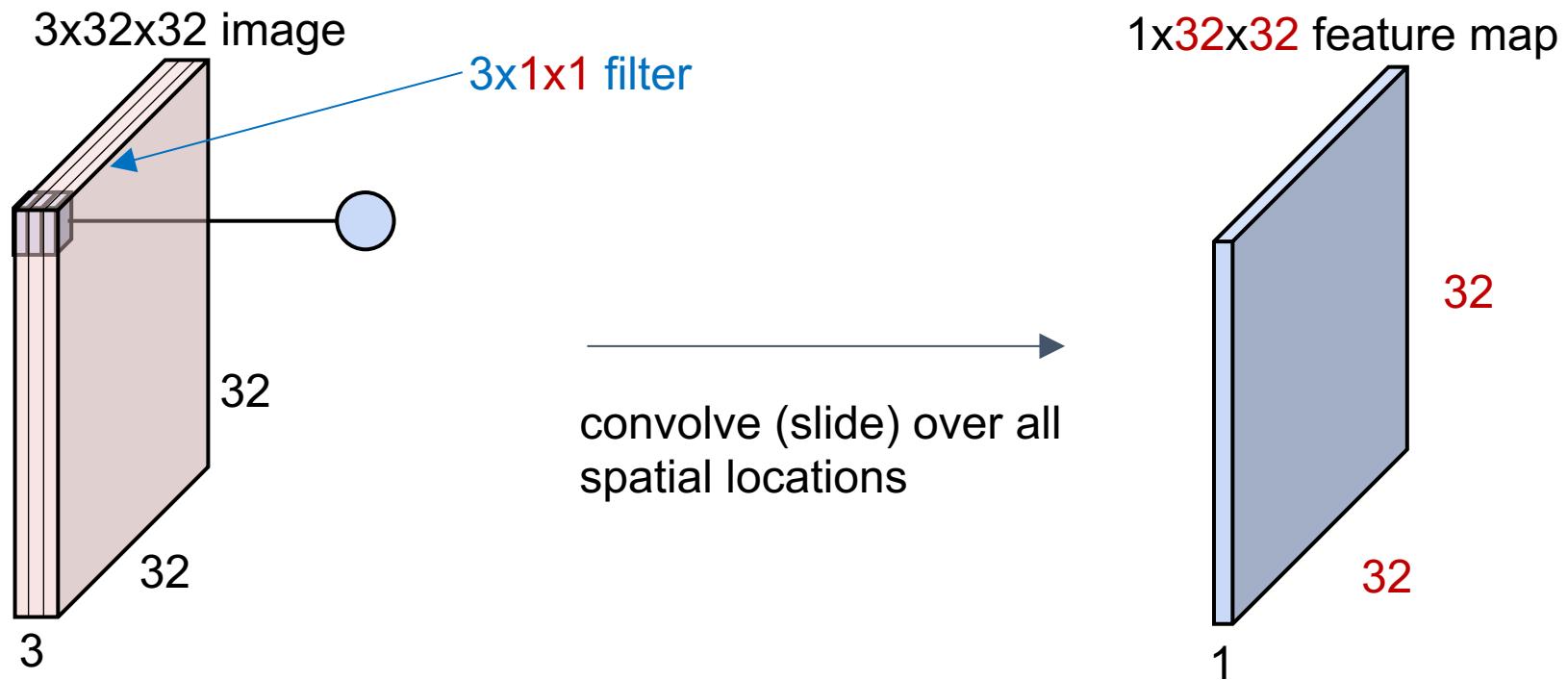
Adding the elements after multiplication, we need $n - 1$ adding operations for n elements

Add the bias



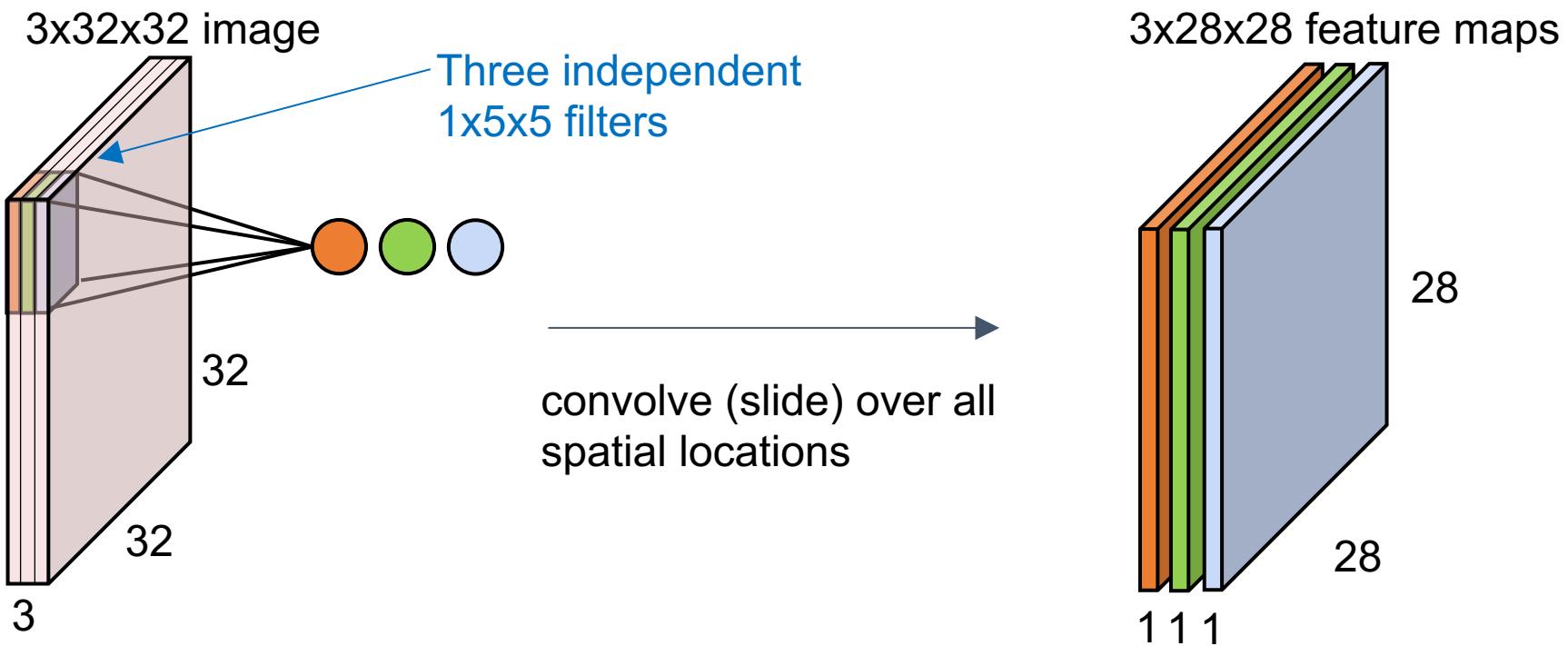
Pointwise convolution

- Why having filter of spatial size 1×1 ?
 - Change the size of channels
 - “Blend” information among channels by linear combination



Depthwise convolution

- Depthwise convolution
 - Convolution is performed **independently** for each of input channels

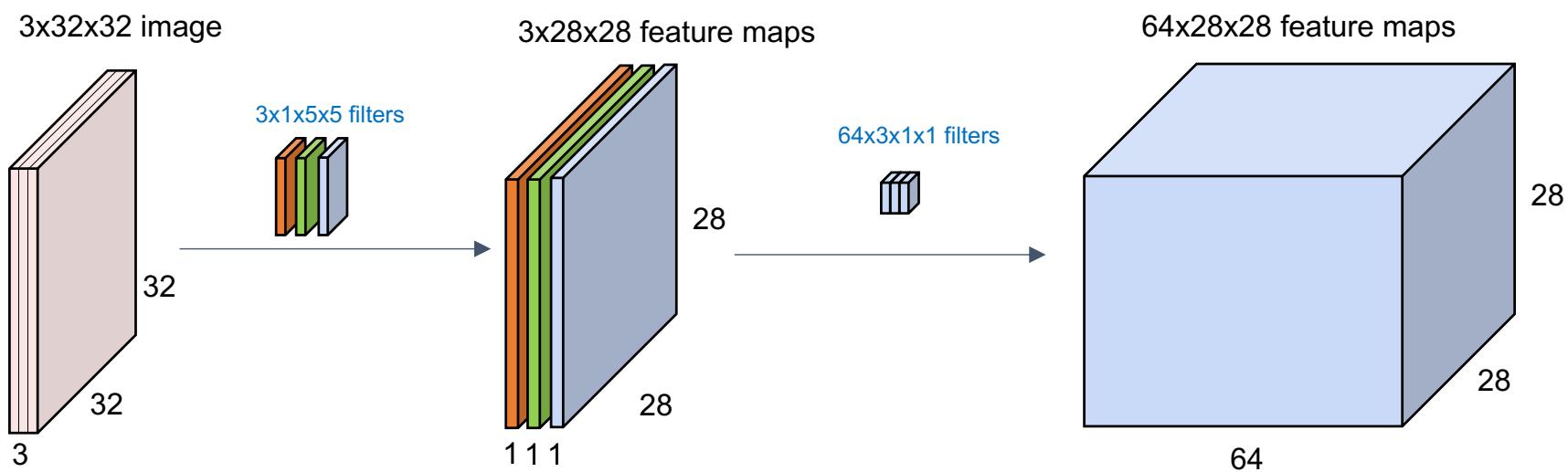
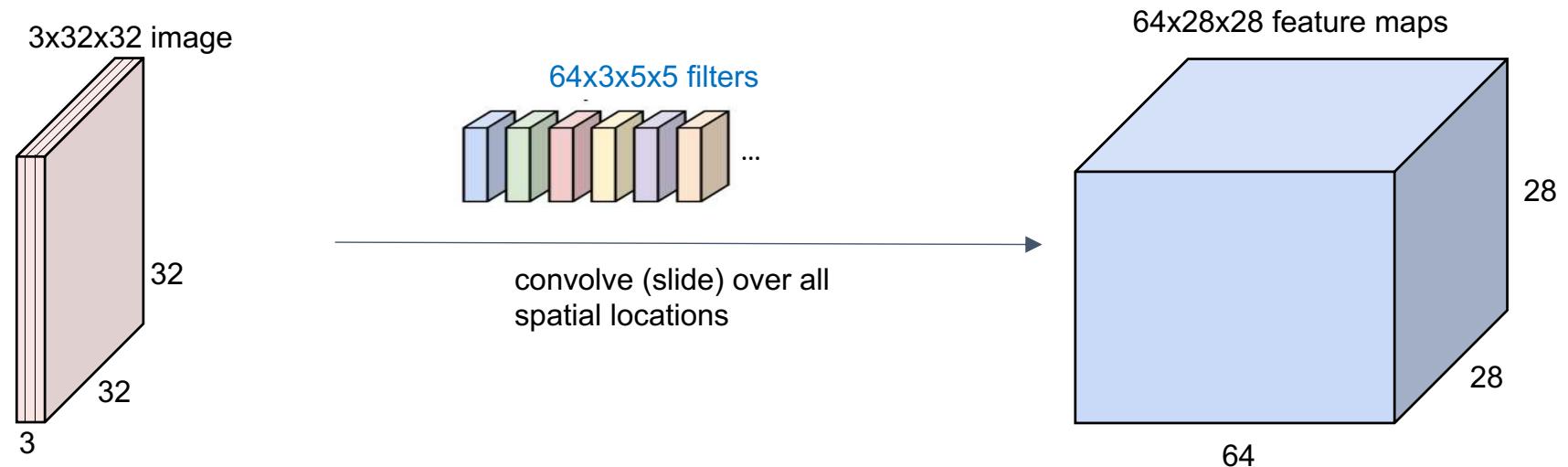


Depthwise convolution + Pointwise convolution

Replace standard convolution
with

Depthwise convolution +
pointwise convolution

And we still get the same
size of output volume!



Depthwise convolution + Pointwise convolution

- How much computation do you save by replacing standard convolution with depthwise+pointwise?

$$\text{Ratio} = \frac{\text{Cost of depthwise convolution} + \text{pointwise convolution}}{\text{Cost of standard convolution}}$$

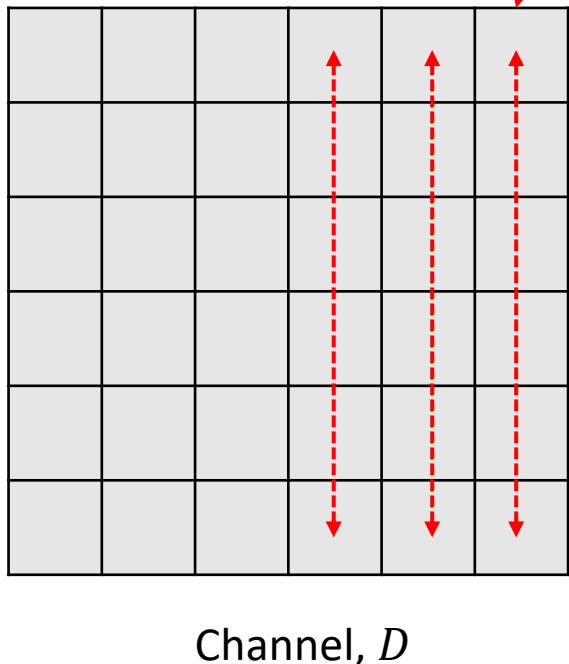
$$\text{Ratio} = \frac{(2 \times F^2) \times D_1 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2} + \frac{(2 \times D_1) \times D_2 \times H_2 \times W_2}{(2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2}$$

$$\text{Ratio} = \frac{1}{D_2} + \frac{1}{F^2}$$

D_2 is usually large. Reduction rate is roughly 1/8–1/9 if 3×3 depthwise separable convolutions are used

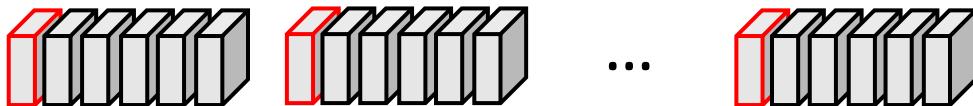
Batch Normalization

Input, N



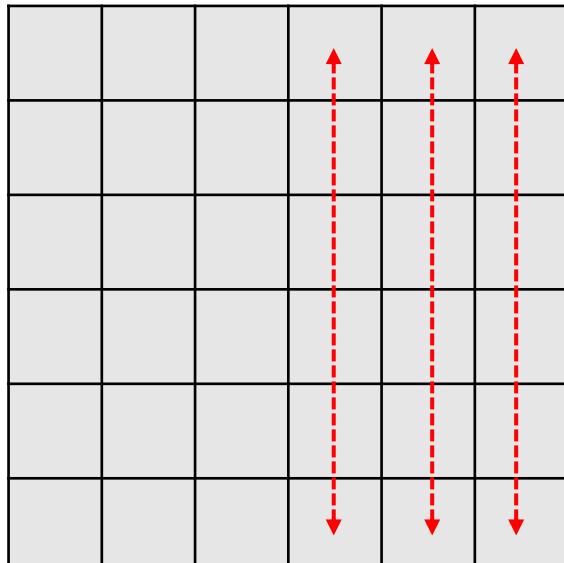
The goal of batch normalization is to normalize the values across each column so that the values of the column have **zero mean and unit variance**

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below



Batch Normalization - Training

Input, N



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization – Test Time

Input



Problem: Estimates depend on minibatch; can't do this at test-time

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Batch Normalization – Test Time

Input



Channel, D

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

Average of values seen during training

Per-channel mean, shape is $1 \times D$

Average of values seen during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β , shape is $1 \times D$

Outline – CNN II

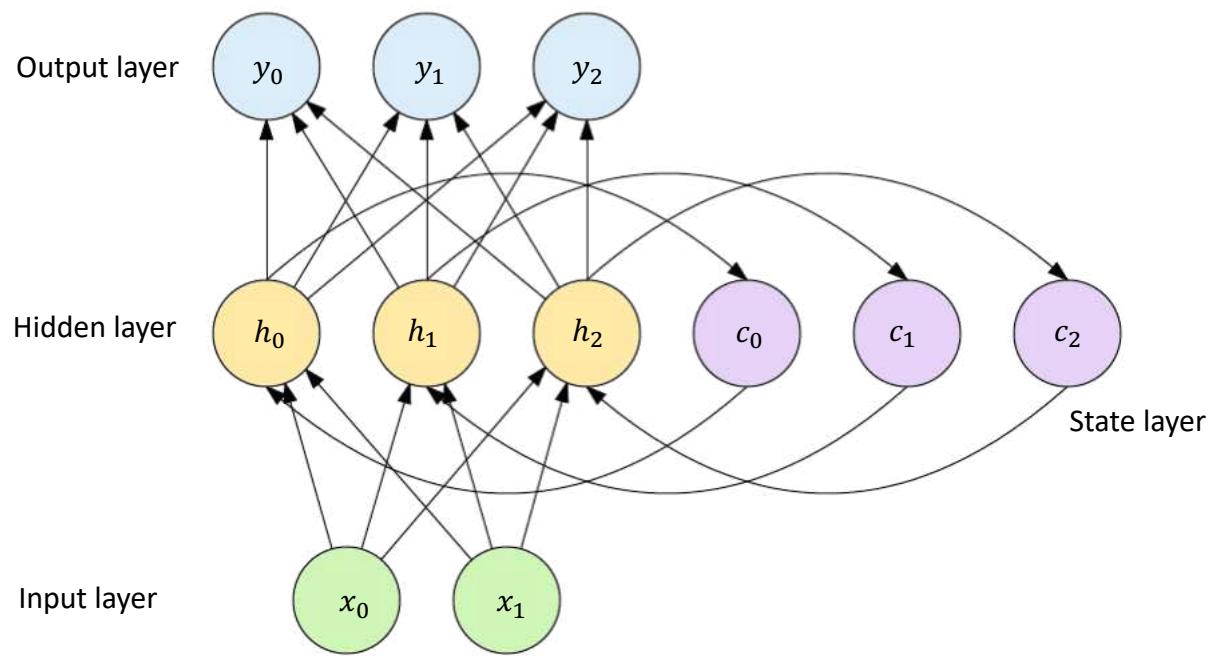
- CNN Architectures
 - You learn some classic architectures
- More on convolution
 - How to calculate FLOPs
 - Pointwise convolution
 - Depthwise convolution
 - Depthwise convolution + Pointwise convolution
 - You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network
- Batch normalization
 - You learn an important technique to improve the training of modern neural networks
- Prevent overfitting
 - Transfer learning
 - Data augmentation
 - You learn two important techniques to prevent overfitting in neural networks

Outline – RNN

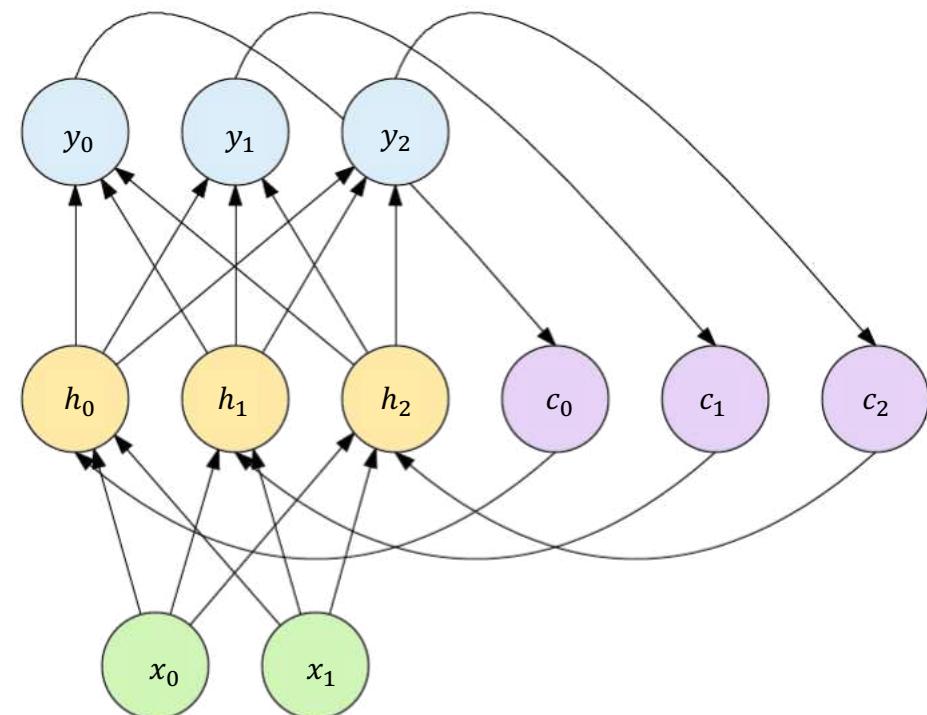
- Recurrent Neural Network (RNN)
 - Hidden recurrence
 - Top-down recurrence
- Long Short-Term Memory (LSTM)
 - Long-term dependency
 - Structure of LSTM
- Example Applications

Types of RNN

RNN with hidden recurrence
(Elman-type)



RNN with top-down recurrence
(Jordan-type)



RNN with hidden recurrence

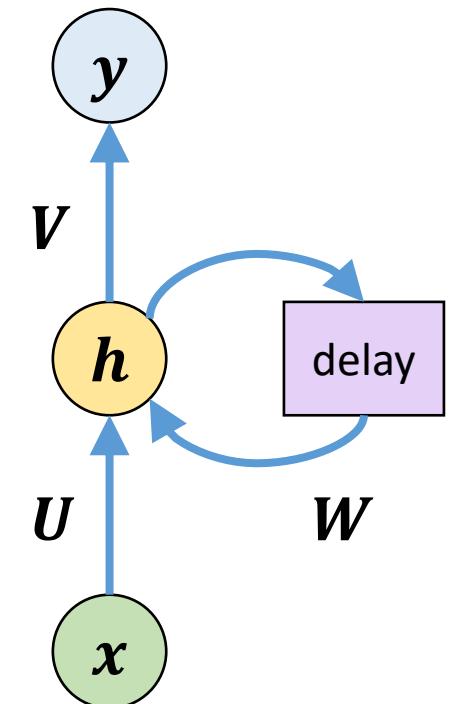
Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Elman-type RNN with one hidden-layer is given by:

$$h(t) = \phi(U^T x(t) + W^T h(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

σ is a *softmax* function for classification and a *linear* function for regression.



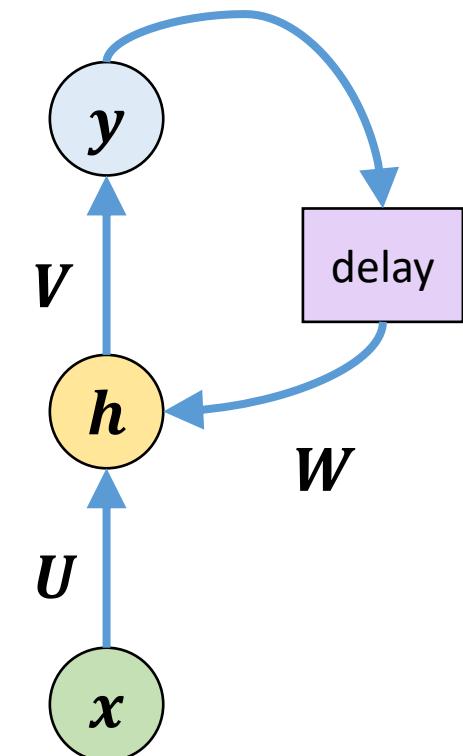
RNN with top-down recurrence

Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Jordan-type RNN with one hidden-layer is given by:

$$\begin{aligned} h(t) &= \phi(U^T x(t) + W^T y(t-1) + b) \\ y(t) &= \sigma(V^T h(t) + c) \end{aligned}$$

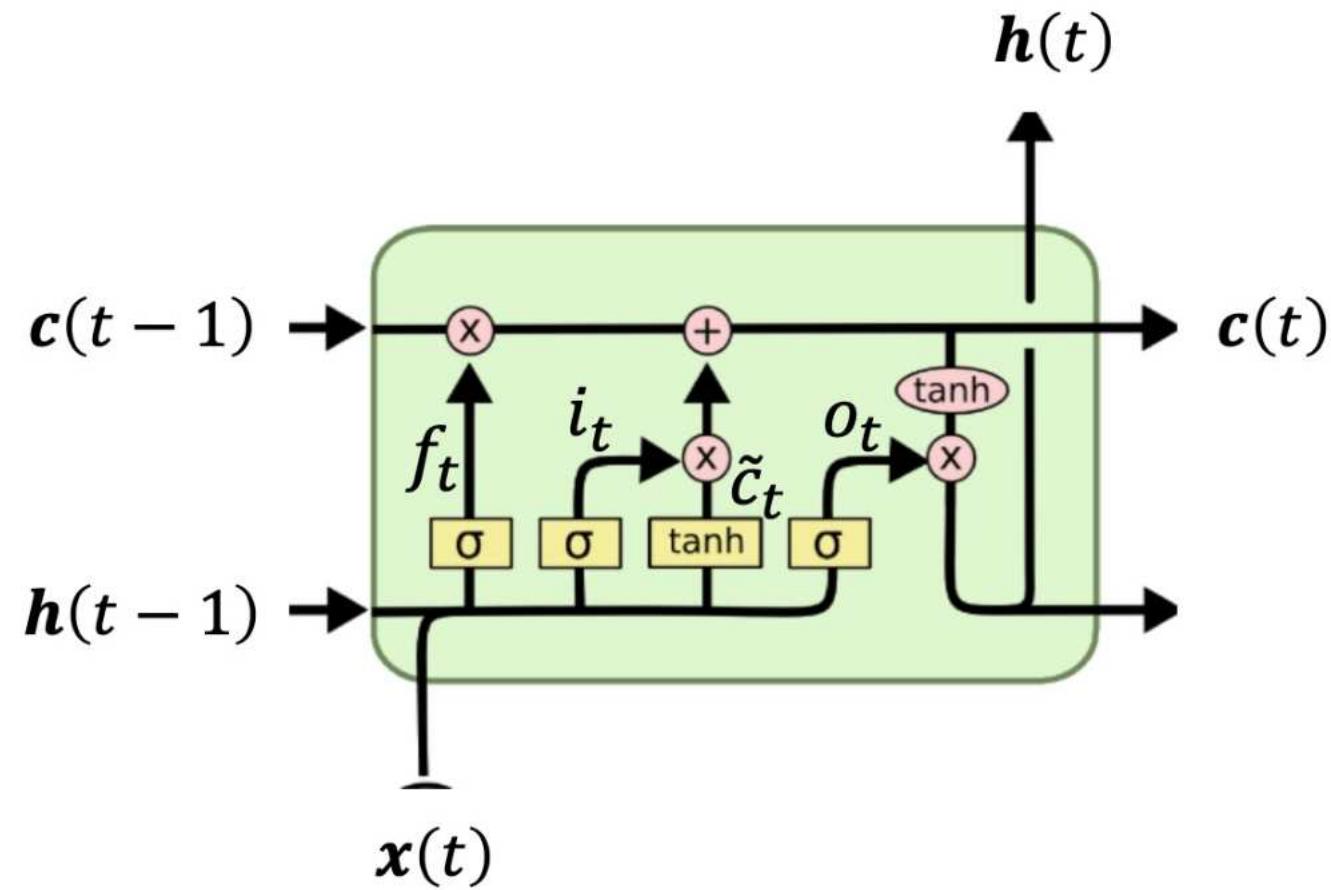
Note that output of the previous time instant is fed back to the hidden layer and W represents the recurrent weight matrix connecting previous output to the current hidden input



Long short-term memory (LSTM) unit

LSTMs provide a solution by incorporating memory units that allow the network to learn when to **forget previous hidden states** and when to **update hidden states** given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.



LSTM unit

$$i(t) = \sigma(U_i^\top x(t) + W_i^\top h(t-1) + b_i)$$

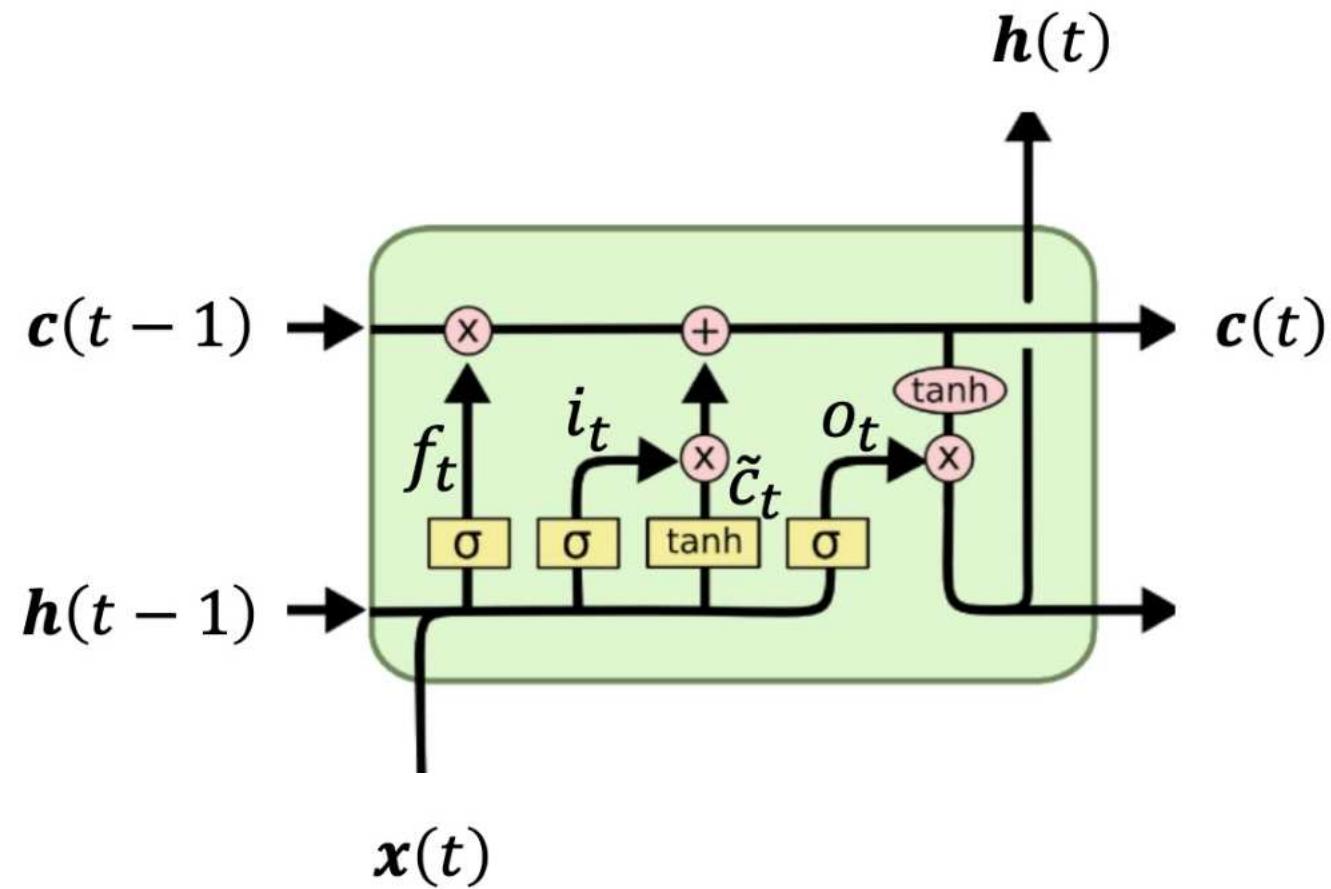
$$f(t) = \sigma(U_f^\top x(t) + W_f^\top h(t-1) + b_f)$$

$$o(t) = \sigma(U_o^\top x(t) + W_o^\top h(t-1) + b_o)$$

$$\tilde{c}(t) = \phi(U_c^\top x(t) + W_c^\top h(t-1) + b_c)$$

$$c(t) = \tilde{c}(t) \odot i(t) + c(t-1) \odot f(t)$$

$$h(t) = \phi(c(t)) \odot o(t)$$



Outline – Attention

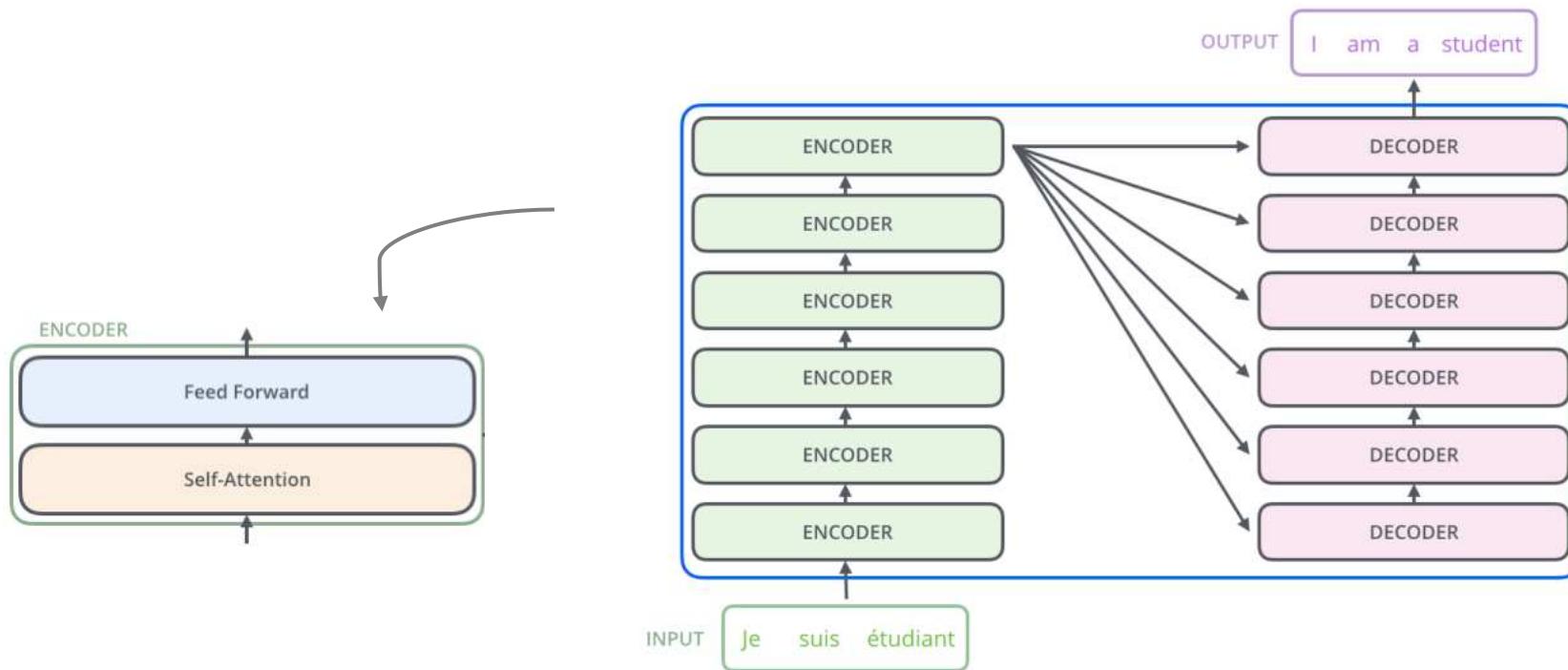
- Attention
- Transformers
- Vision Transformers

Transformers

The encoder's inputs first flow through a self-attention layer – a layer that **helps the encoder look at other words in the input sentence** as it encodes a specific word

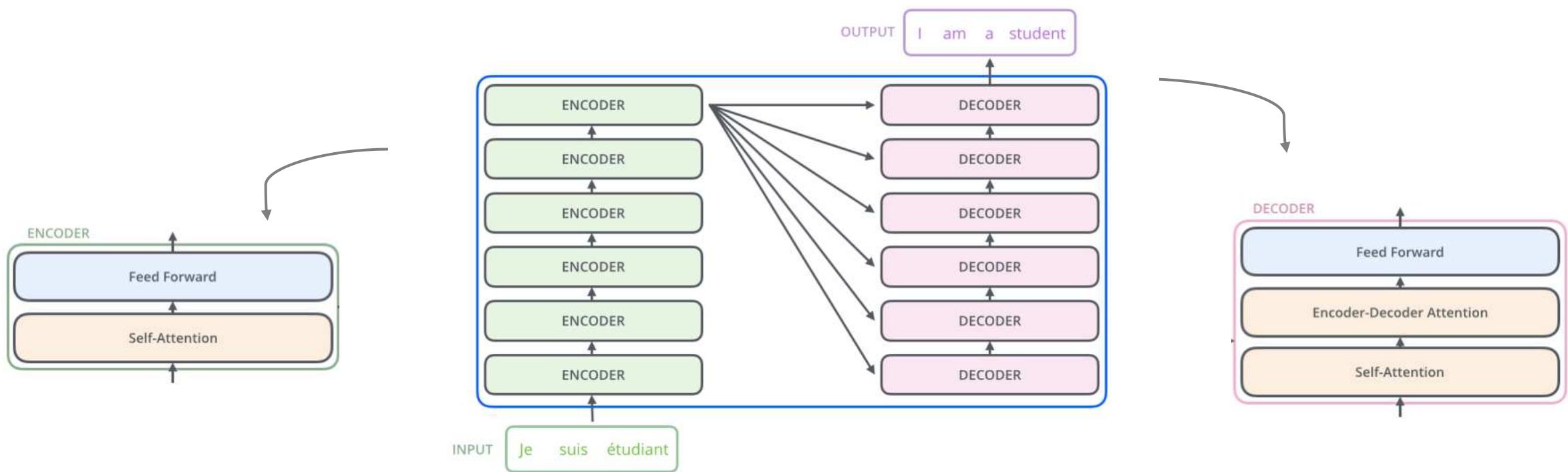
The outputs of the self-attention layer are fed to a **feed-forward neural network**.

The exact same feed-forward network is **independently applied** to each position (each word/token).

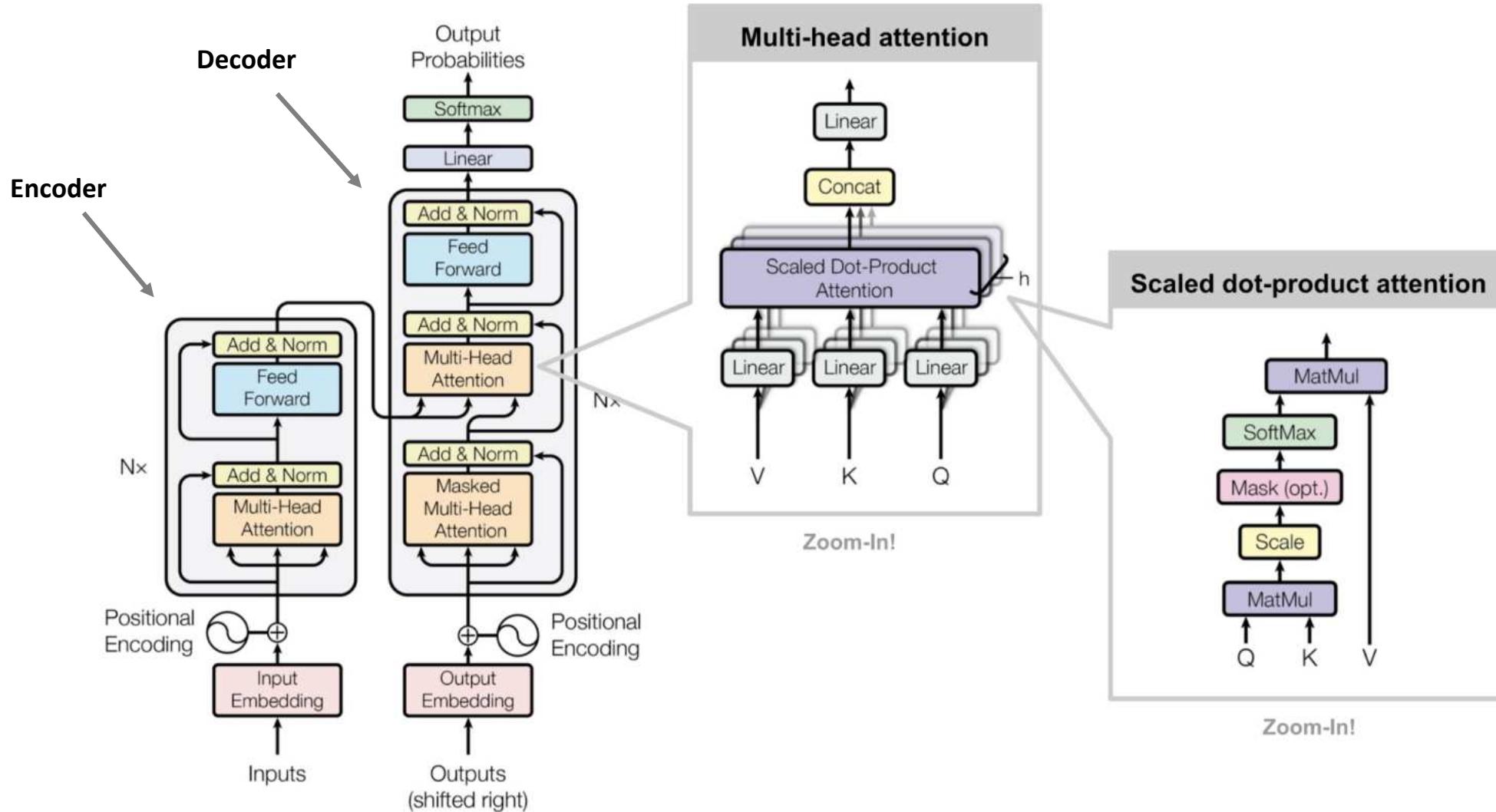


Transformers

The decoder has both those layers, but between them is an attention layer that **helps the decoder focus on relevant parts of the input sentence**



Transformers

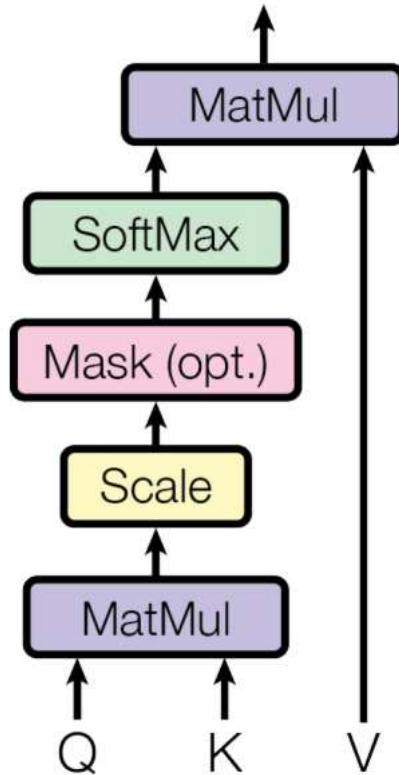


Transformers

Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$



Scaled Dot-Product Attention

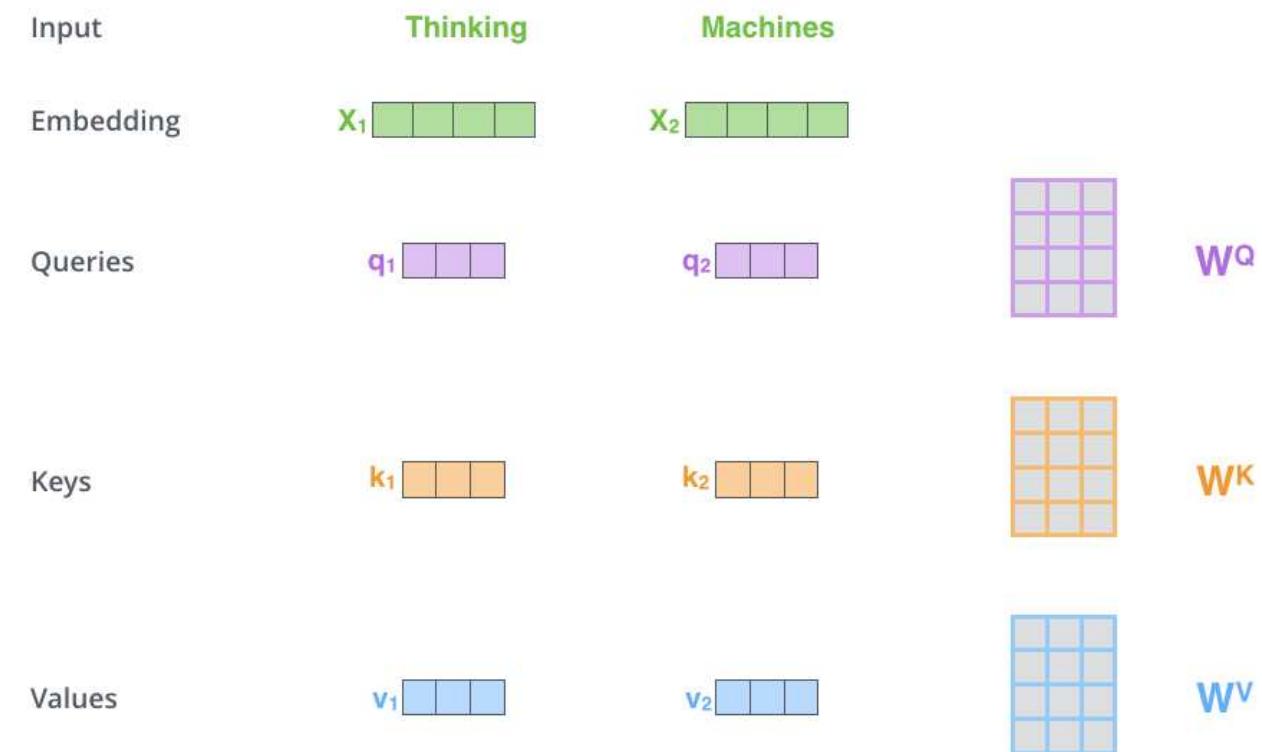
Self-Attention in Detail

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).

So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

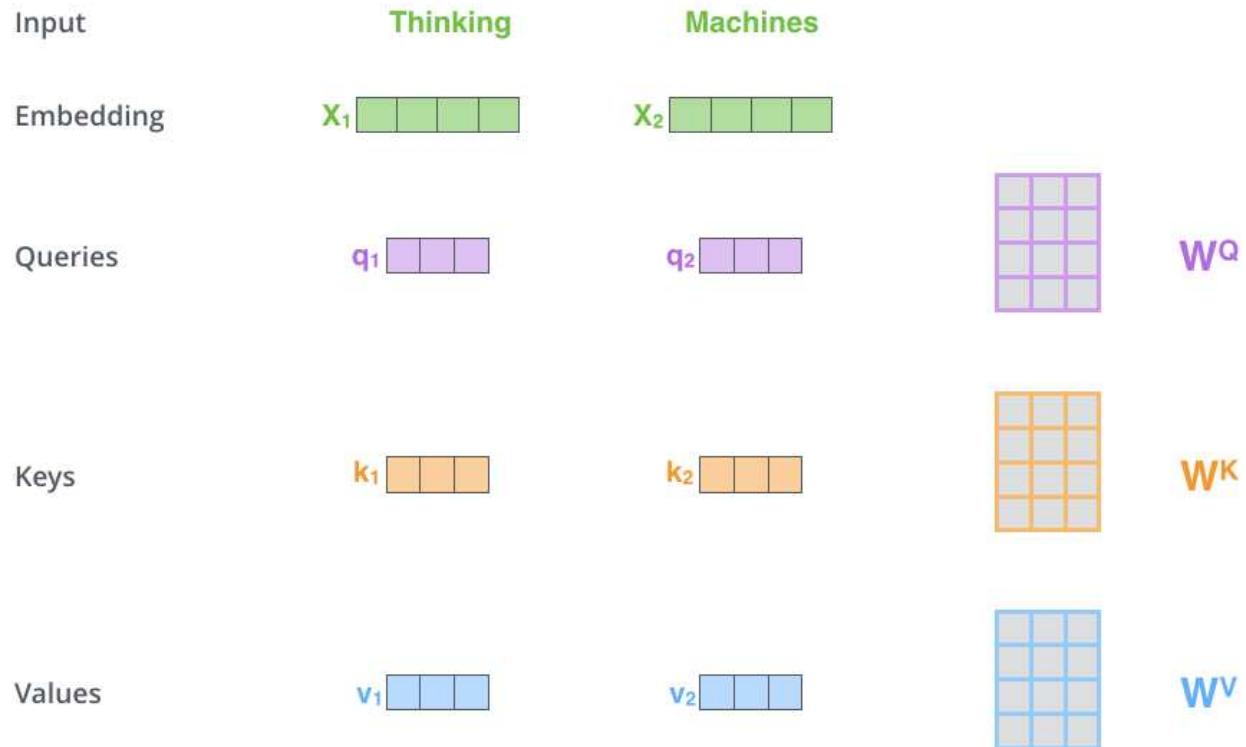
First Step

What are the “query”, “key”, and “value” vectors?

The names “query”, “key” are inherited from the field of **information retrieval**

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{QK^T}{\sqrt{d_k}}$ depend on the relative similarities between the n -th **query** and all of the **keys**

The softmax function means that the **key** vectors “compete” with one another to contribute to the final result.



$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

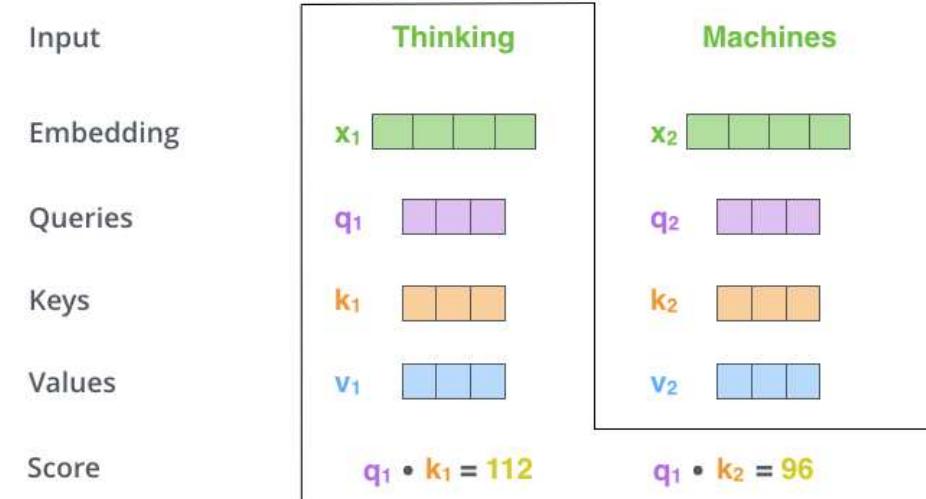
Self-Attention in Detail

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.



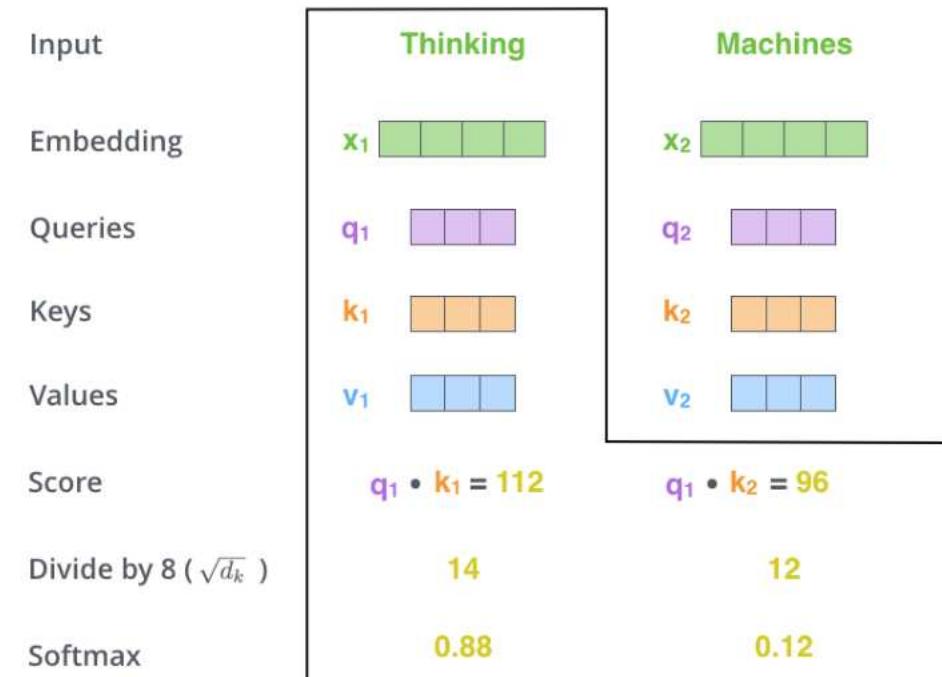
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)



Fourth Step

Softmax for normalization

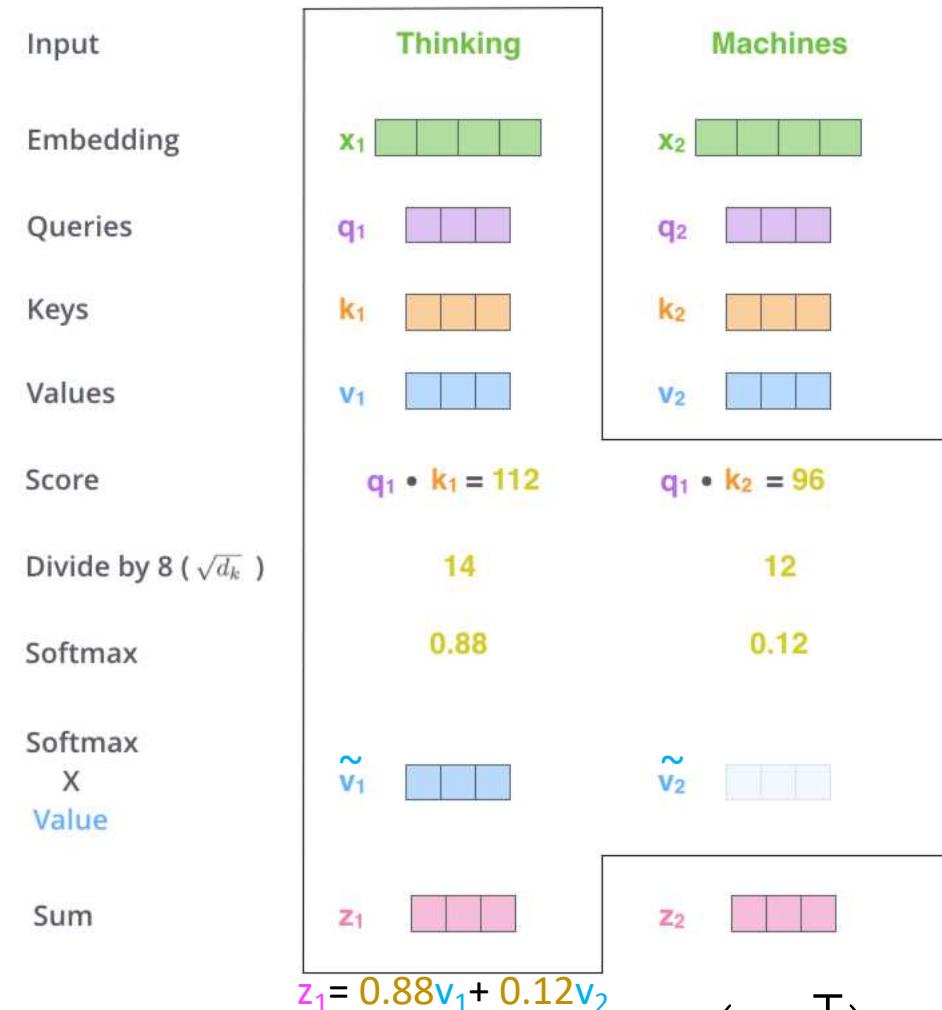
$$z_1 = q_1 \cdot k_1 / \sqrt{d_k} = 112 / \sqrt{64} = 112 / 8 = 14, z_2 = q_1 \cdot k_2 / \sqrt{d_k} = 96 / \sqrt{64} = 96 / 8 = 12$$
$$\text{softmax}(z_1) = \exp(z_1) / \sum_{i=1}^2 (\exp(z_i)) = 0.88, \text{softmax}(z_2) = \exp(z_2) / \sum_{i=1}^2 (\exp(z_i)) = 0.12$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

Fifth Step

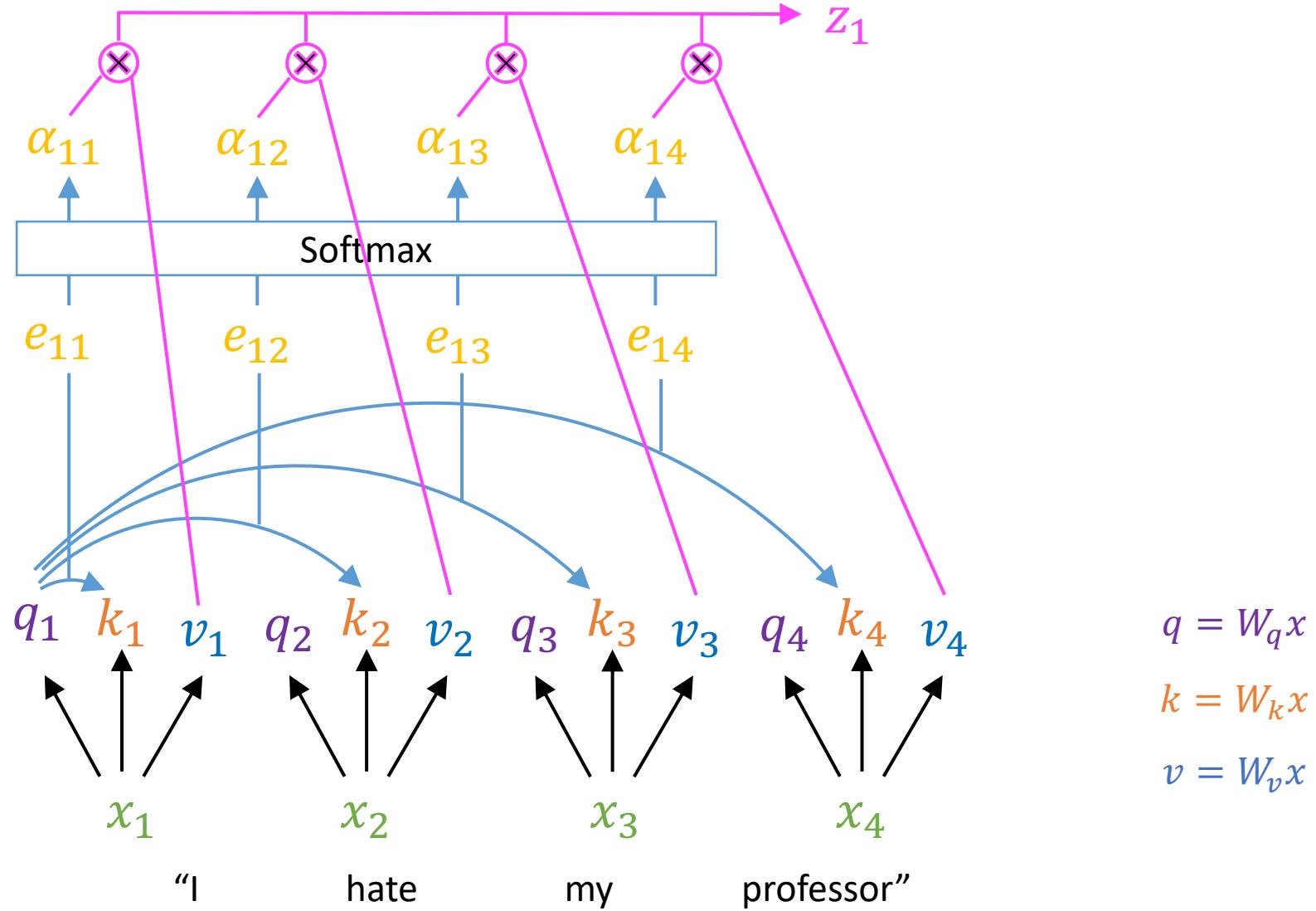
Multiply each value vector by the softmax score



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

$$e = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}$$

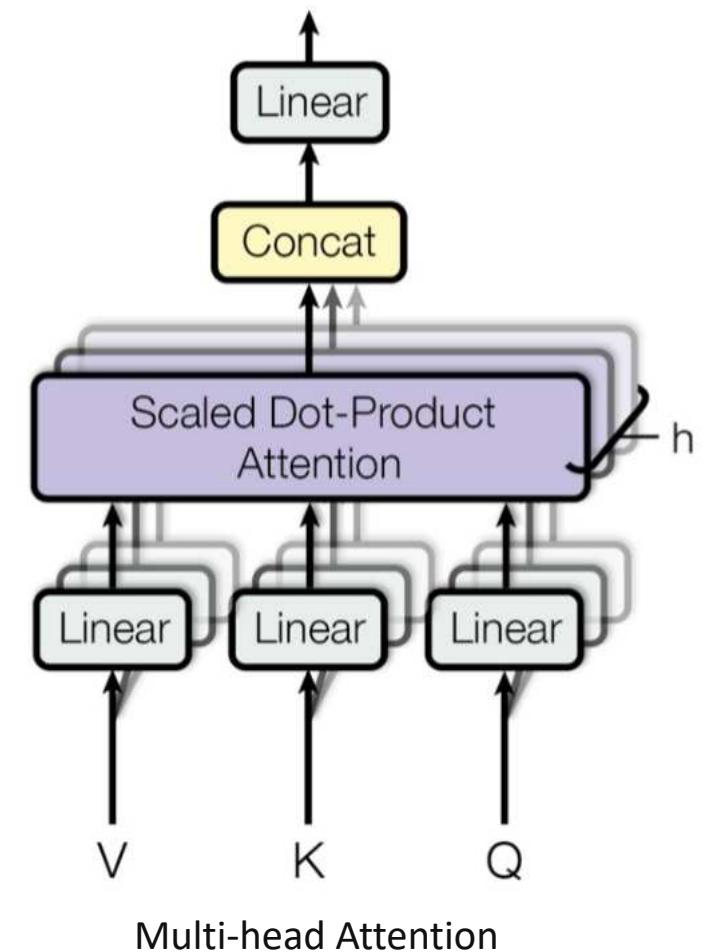
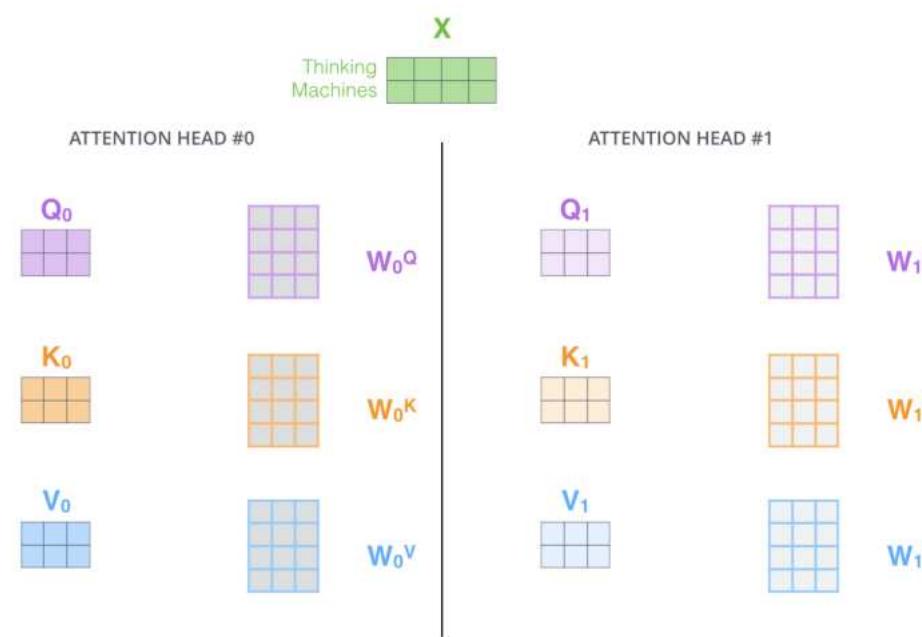


Multi-Head Self-Attention

Multi-Head Self-Attention

Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention **multiple times in parallel**.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values



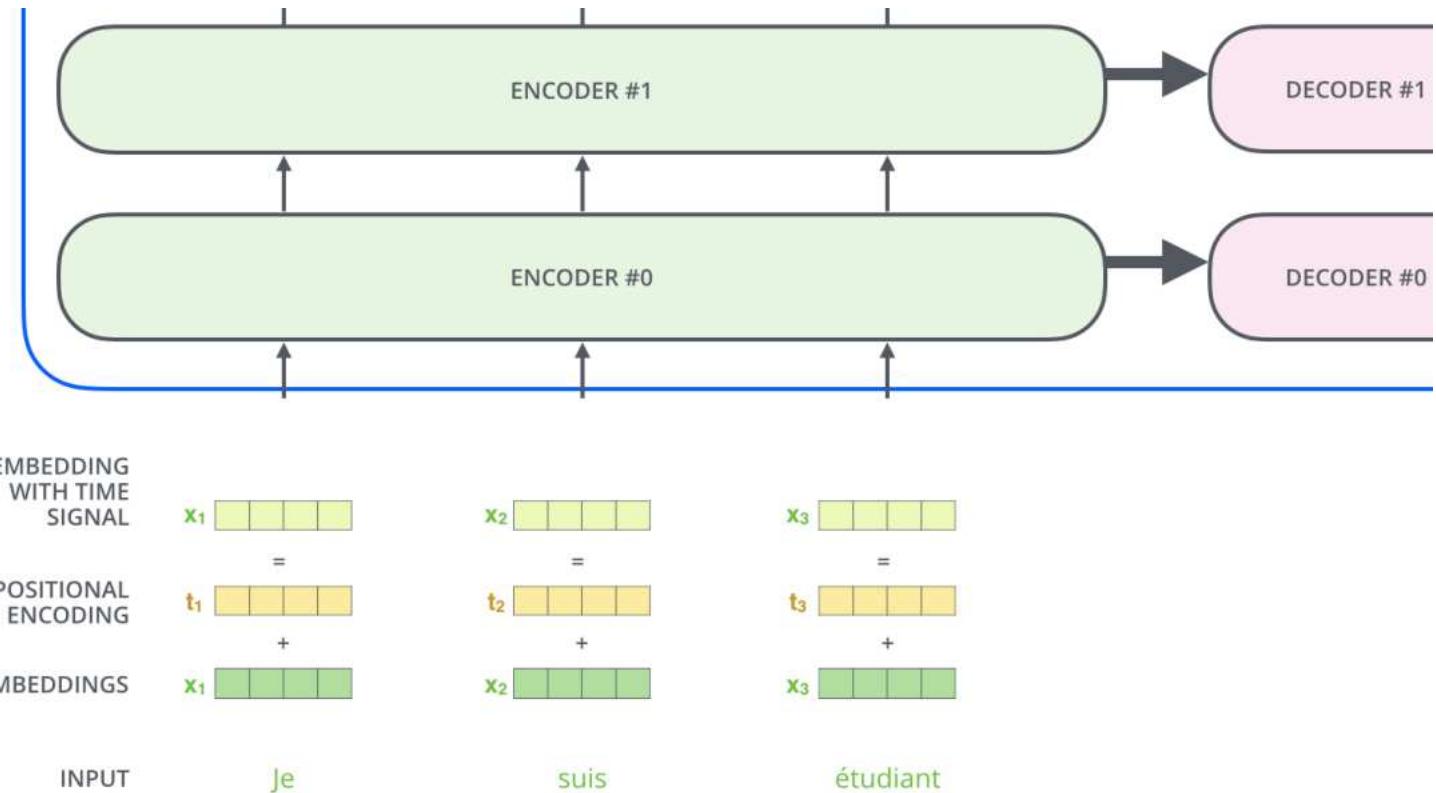
Positional Encoding

Self-attention layer works on sets of vectors and it **doesn't know the order** of the vectors it is processing

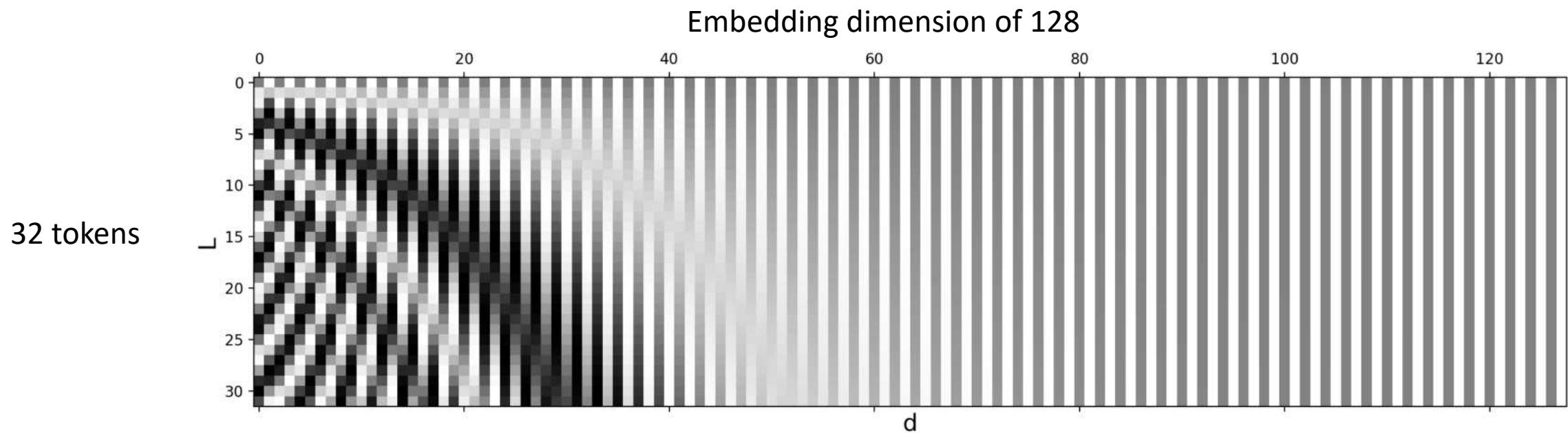
The positional encoding has the **same dimension** as the input embedding

Adds a vector to each input embedding to give information about the **relative or absolute position** of the tokens in the sequence

These vectors follow a specific pattern



Positional Encoding



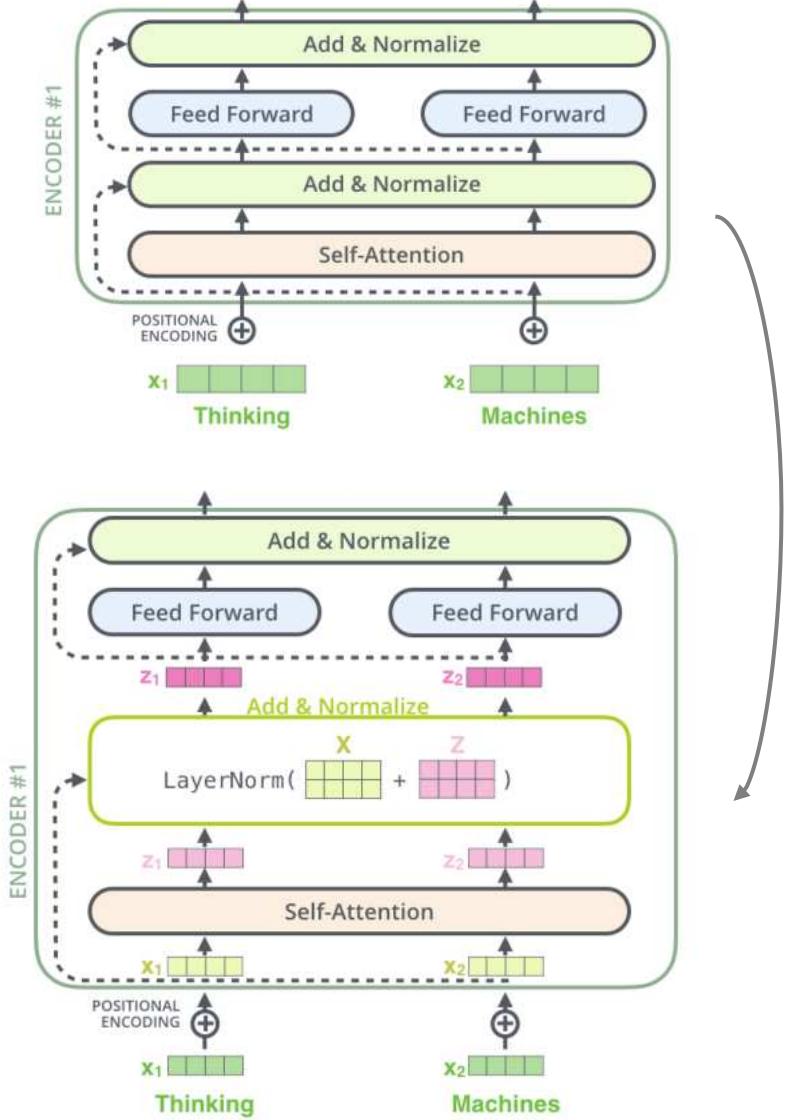
Sinusoidal positional encoding - interweaves the two signals (sine for even indices and cosine for odd indice)

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension, $[0, \dots, d_{\text{model}}/2]$

Transformer Encoder



Encoder

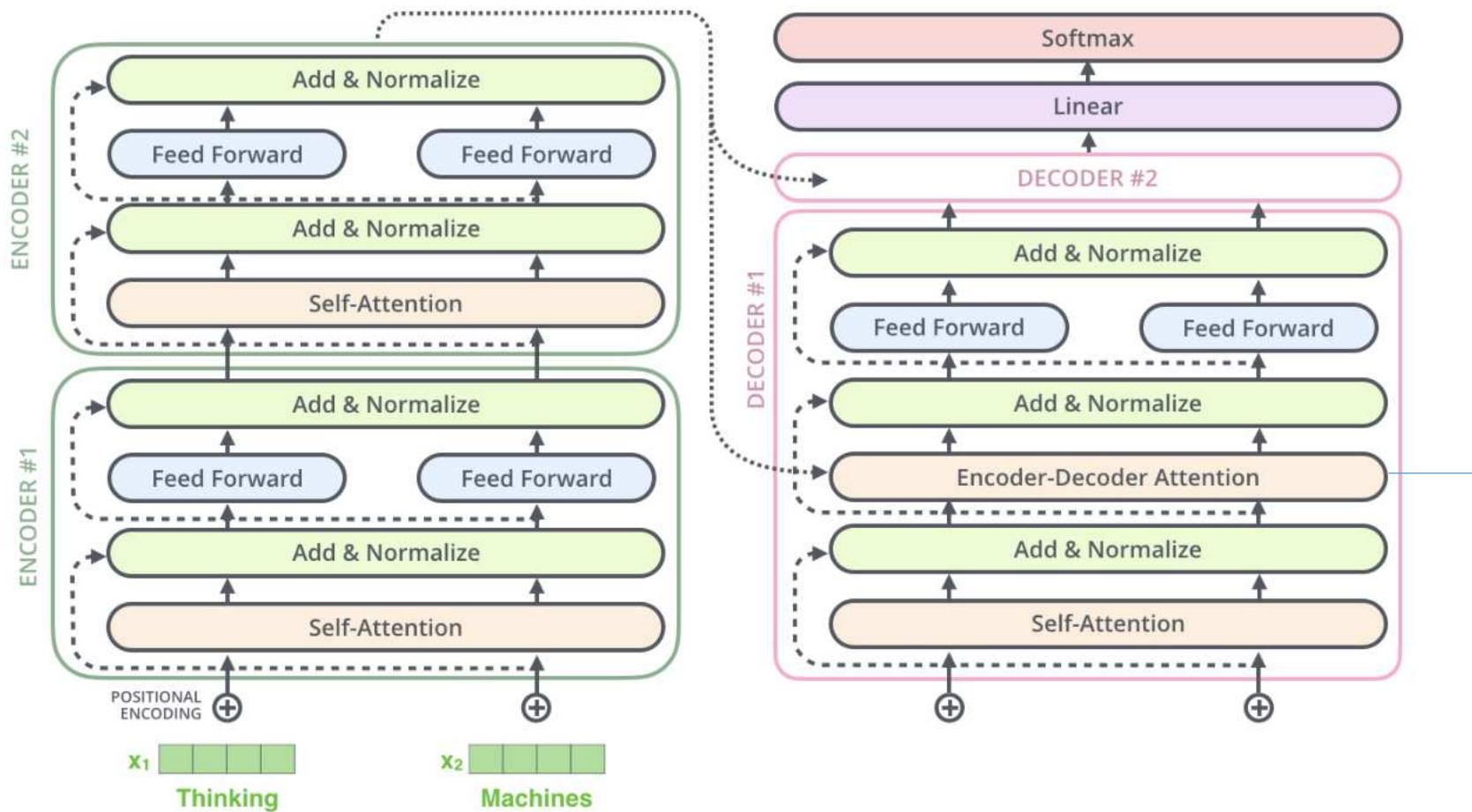
- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

Transformer Encoder

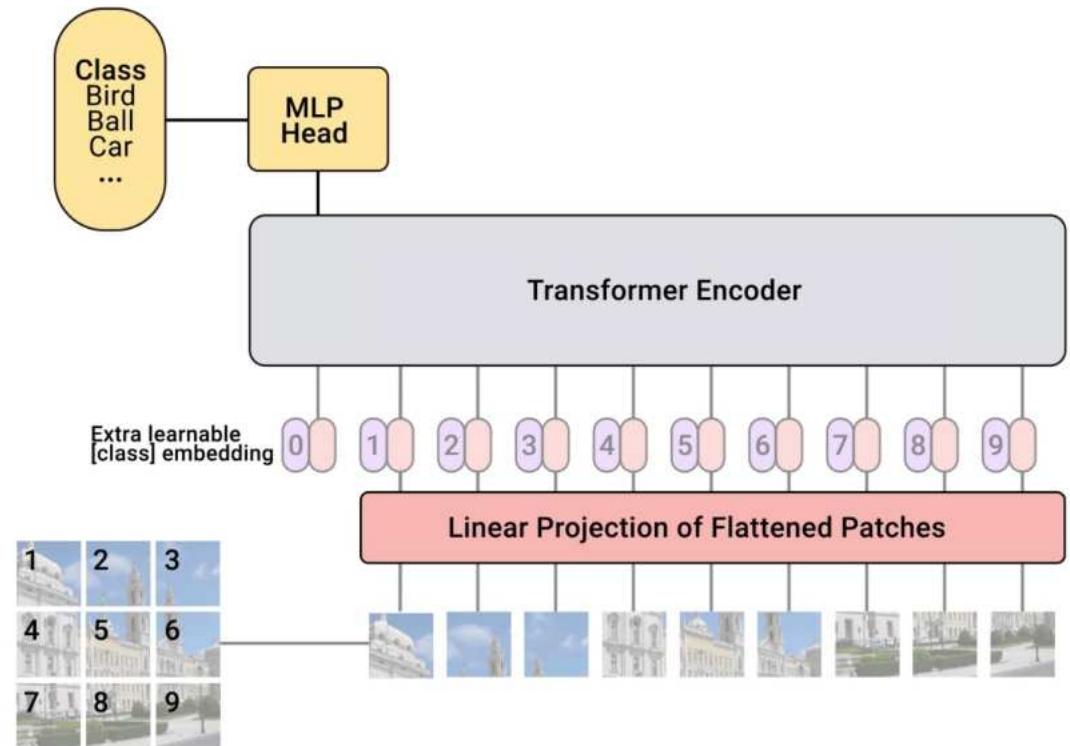
The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the **Keys** and **Values** matrix from the output of the encoder stack.

Vision Transformer (ViT)

- Do not have decoder
- Reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
 - (H, W) is the resolution of the original image
 - C is the number of channels
 - (P, P) is the resolution of each image patch
 - $N = HW/P^2$ is the resulting number of patches



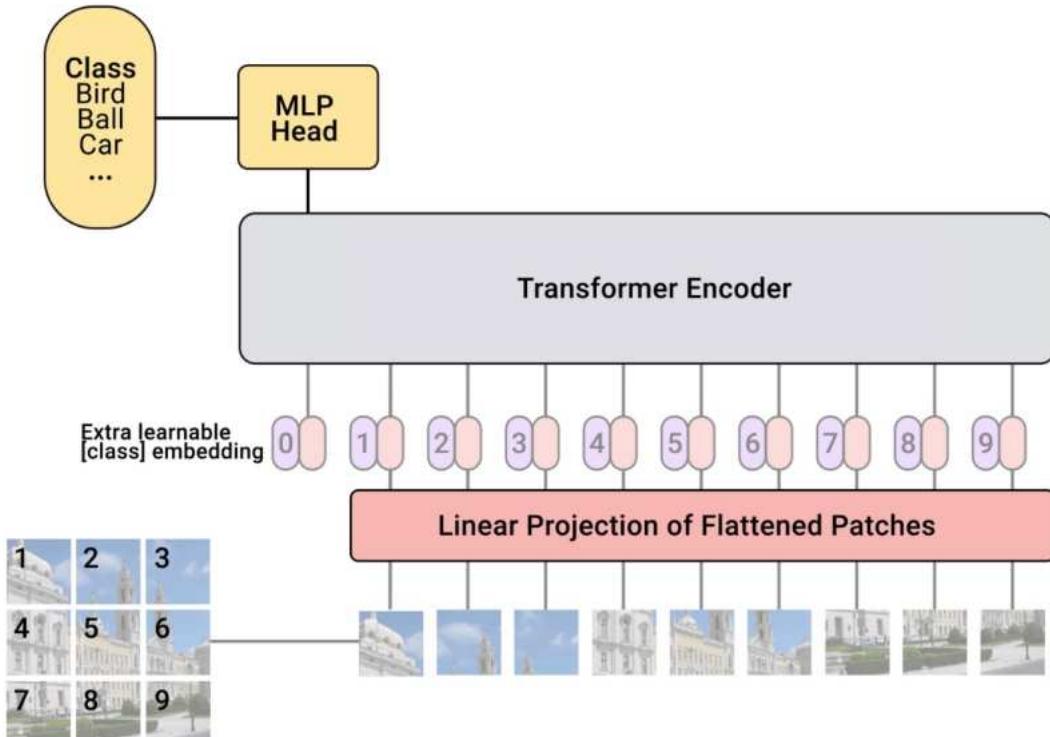
Vision Transformer (ViT)

Prepend a **learnable embedding** ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1 \mathbf{E}; \mathbf{x}_p^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{pos}$$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

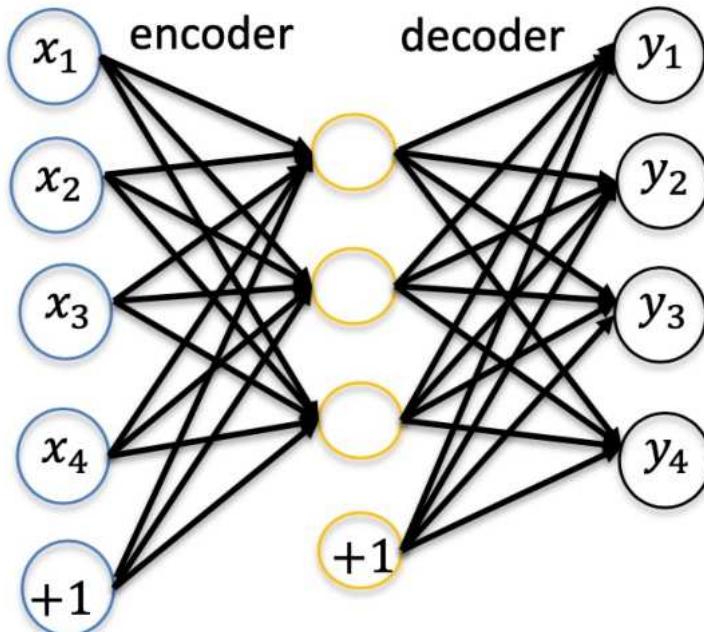
Add **learnable position embeddings** $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information



Outline – Autoencoders

- Supervised vs unsupervised learning
- Autoencoders
 - Denoising autoencoders
 - Undercomplete and overcomplete autoencoders
 - Sparse autoencoders
 - Other encoders

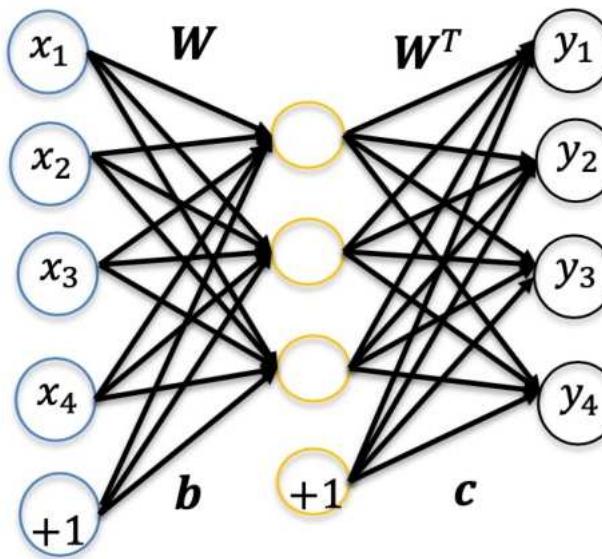
Autoencoders



An autoencoder is a neural network that is trained to attempt to copy its input to its output. Its hidden layer describes a *code* that represents the input.

The network consists of two parts: an **encoder** and a **decoder**.

Autoencoders



Reverse mapping from the hidden layer to the output can be **optionally** constrained to be the same as the input to hidden-layer mapping (**tied weights**). That is, if encoder weight matrix is \mathbf{W} , the decoder weigh matrix is \mathbf{W}^T .

Hidden layer and output layer activation can be then written as

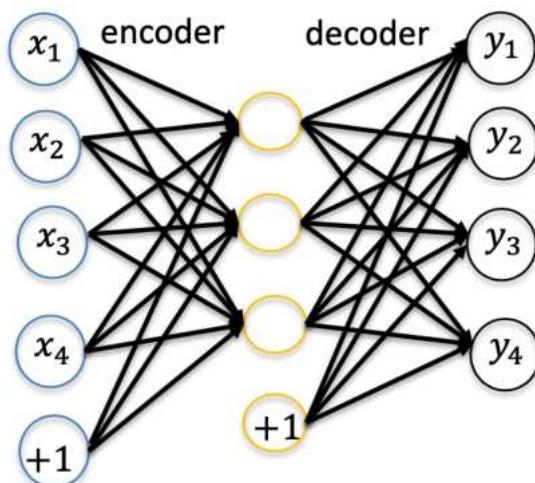
$$\begin{aligned}\mathbf{h} &= f(\mathbf{W}^T \mathbf{x} + \mathbf{b}) \\ \mathbf{y} &= f(\mathbf{W}\mathbf{h} + \mathbf{c})\end{aligned}$$

f is usually a sigmoid.

Denoising Autoencoders (DAE)

A *denoising autoencoder* (DAE) receives corrupted data points as inputs.

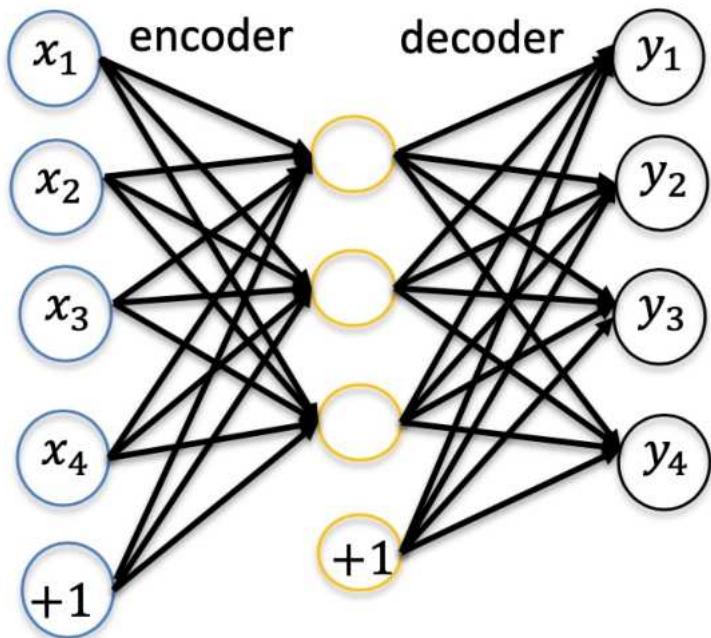
It is trained to predict the original uncorrupted data points as its output.



The idea of DAE is that in order to force the hidden layer to **discover more robust features** and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to **undo the effect of corruption process** applied to the input of the autoencoder.

Autoencoders



Input dimension n and hidden dimension M :

If $M < n$, *undercomplete* autoencoders

If $M > n$, *overcomplete* autoencoders

Sparse Autoencoders (SAE)

A **sparse autoencoder** (SAE) is simply an autoencoder whose training criterion involves the **sparsity penalty** Ω_{sparsity} at the hidden layer:

$$J_1 = J + \beta \Omega_{\text{sparsity}}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be **sparse**.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be **inactive for most of the time**.

We say that the neuron is “active” when its output is close to 1 and the neuron is “inactive” when its output is close to 0.

Sparsity constraint

To achieve sparse activations at the hidden-layer, the **Kullback-Leibler (KL) divergence** is used as the sparsity constraint:

$$D(\mathbf{h}) = \sum_{j=1}^M \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the **deviation of the distribution** $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ .

The KL divergence is **minimum** when $\rho_j = \rho$ for all j .

That is, when the average activations are uniform and equal to very low value ρ .

Outline – GAN

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

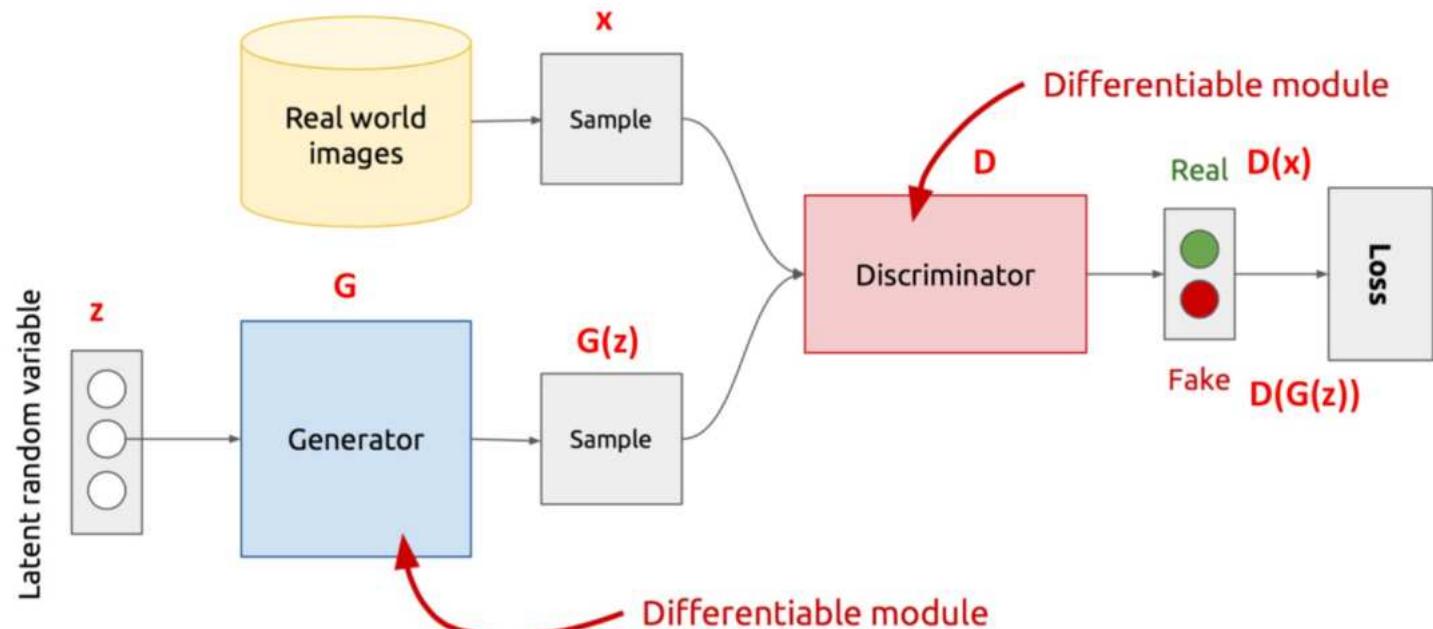
Generative adversarial networks (GAN)

- **Generative model G :**

- Captures data distribution
- Fool $D(G(z))$
- Generate an image $G(z)$ such that $D(G(z))$ is wrong (i.e. $D(G(z)) = 1$)

- **Discriminative model D :**

- Distinguishes between real and fake samples
- $D(x) = 1$ when x is a real image, and otherwise $D(x) = 0$

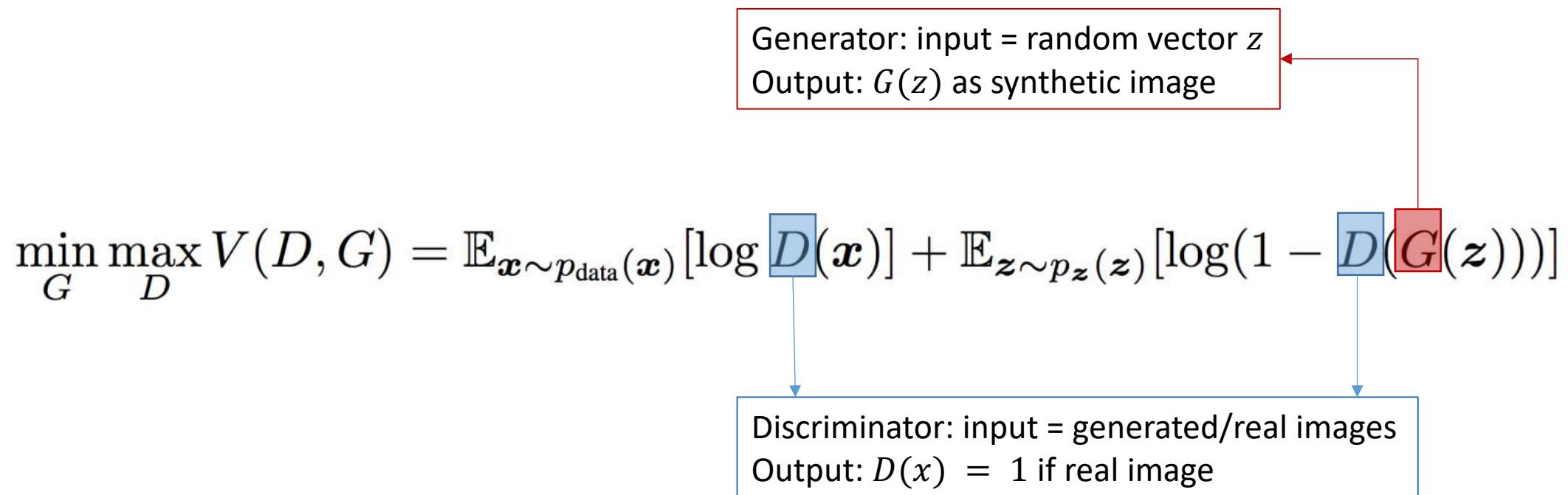


z is some random noise (Gaussian/Uniform).

z can be thought as the latent representation of the data.

Generative adversarial networks (GAN)

- D and G play the following two-player minimax game with value function $V(D, G)$



Training procedure

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

maximize

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)}))) \right].$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.

- Update the generator by descending its stochastic gradient:

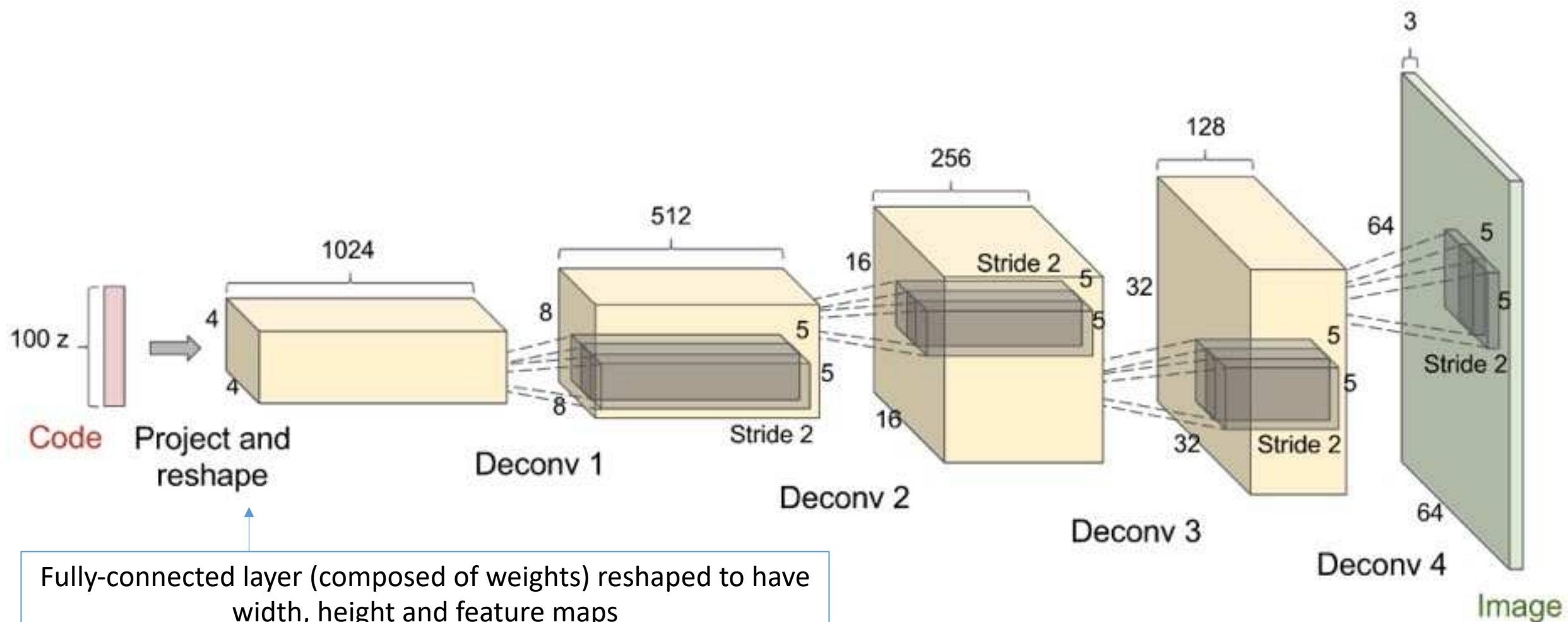
minimize

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

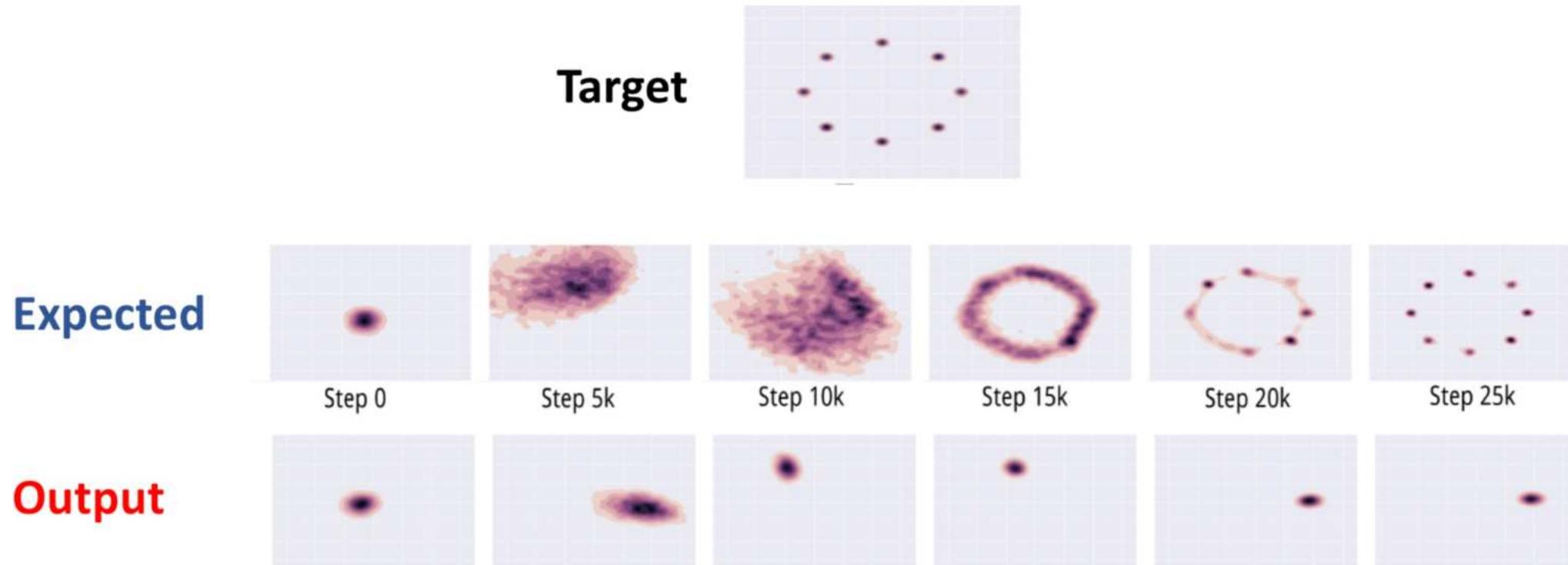
Deep Convolutional GAN (DCGAN)



We will go through the code in the tutorial

Mode Collapse

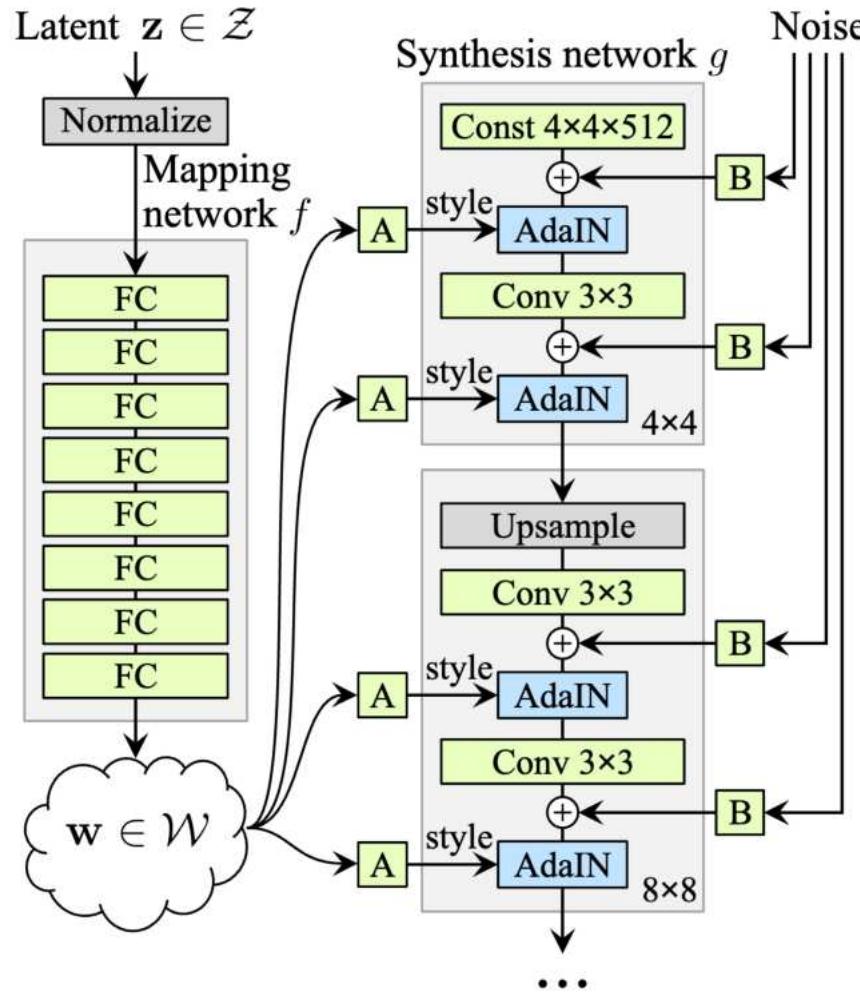
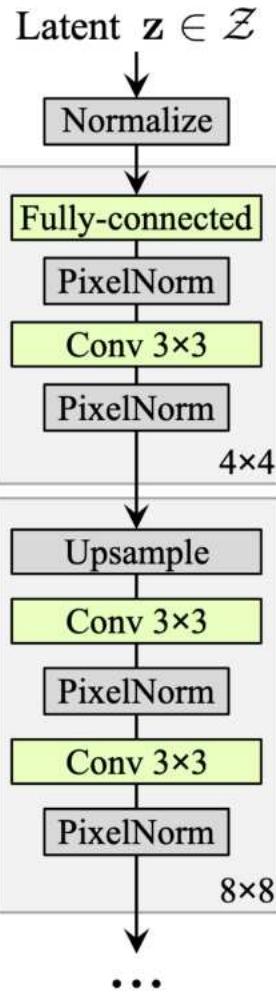
Generator fails to output diverse samples



Applications of GANs

(not included in the final exam)

StyleGAN



A = learned affine transformation block for AdaIN (predicts y)

Adaptive Instance Normalization (very effective in controlling styles)

$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

B = learned per-channel scaling factor for noise input.

Coarse styles
 $(4^2 - 8^2)$



BigGAN



BigGAN

Batch	Ch.	Param (M)	Shared	Hier.	Ortho.	Itr $\times 10^3$	FID	IS
256	64	81.5	SA-GAN Baseline			1000	18.65	52.52
512	64	81.5	✗	✗	✗	1000	15.30	58.77(± 1.18)
1024	64	81.5	✗	✗	✗	1000	14.88	63.03(± 1.42)
2048	64	81.5	✗	✗	✗	732	12.39	76.85(± 3.83)
2048	96	173.5	✗	✗	✗	295(± 18)	9.54(± 0.62)	92.98(± 4.27)
2048	96	160.6	✓	✗	✗	185(± 11)	9.18(± 0.13)	94.94(± 1.32)
2048	96	158.3	✓	✓	✗	152(± 7)	8.73(± 0.45)	98.76(± 2.84)
2048	96	158.3	✓	✓	✓	165(± 13)	8.51(± 0.32)	99.31(± 2.10)
2048	64	71.3	✓	✓	✓	371(± 7)	10.48(± 0.10)	86.90(± 0.61)

Table 1: Fréchet Inception Distance (FID, lower is better) and Inception Score (IS, higher is better) for ablations of our proposed modifications. *Batch* is batch size, *Param* is total number of parameters, *Ch.* is the channel multiplier representing the number of units in each layer, *Shared* is using shared embeddings, *Hier.* is using a hierarchical latent space, *Ortho.* is Orthogonal Regularization, and *Itr* either indicates that the setting is stable to 10^6 iterations, or that it collapses at the given iteration. Other than rows 1-4, results are computed across 8 different random initializations.

BigGAN – Interpolations between c , z pairs

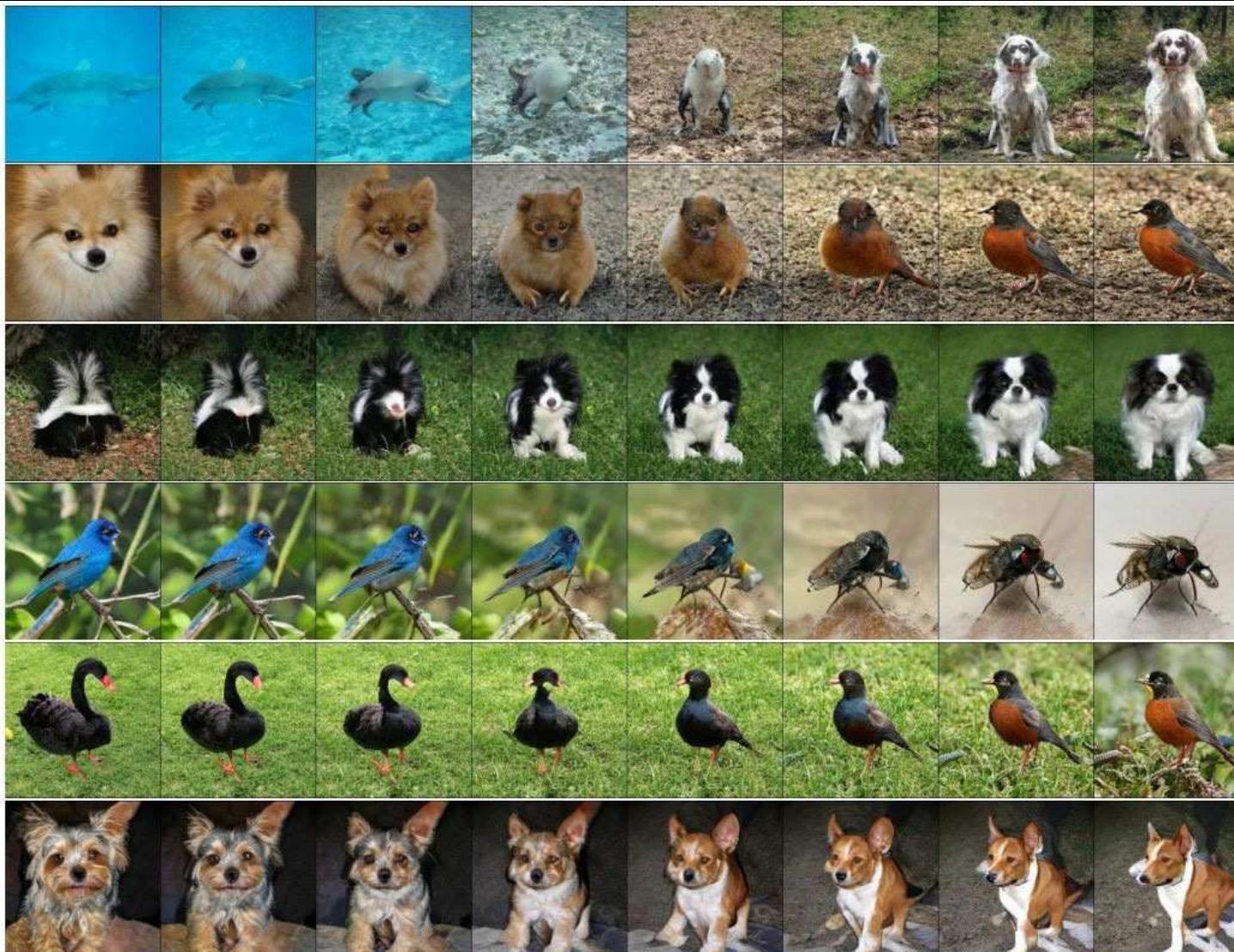
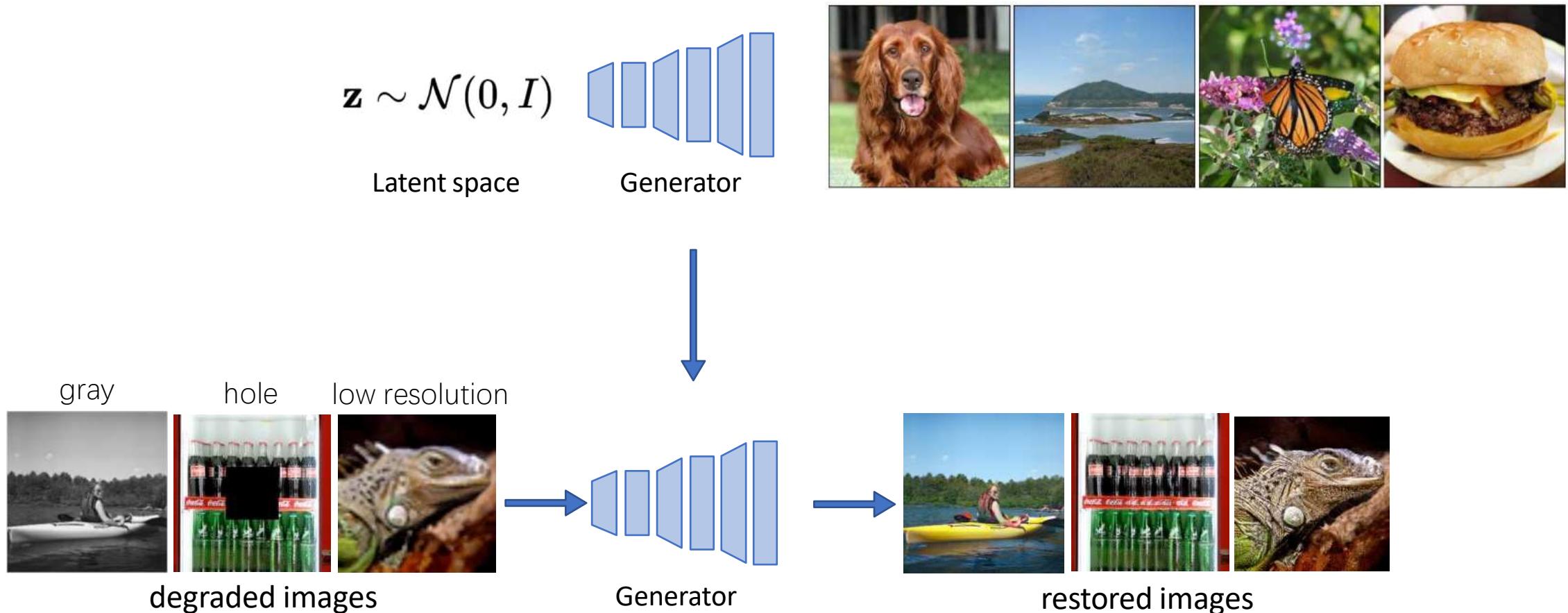


Figure 8: Interpolations between z, c pairs.

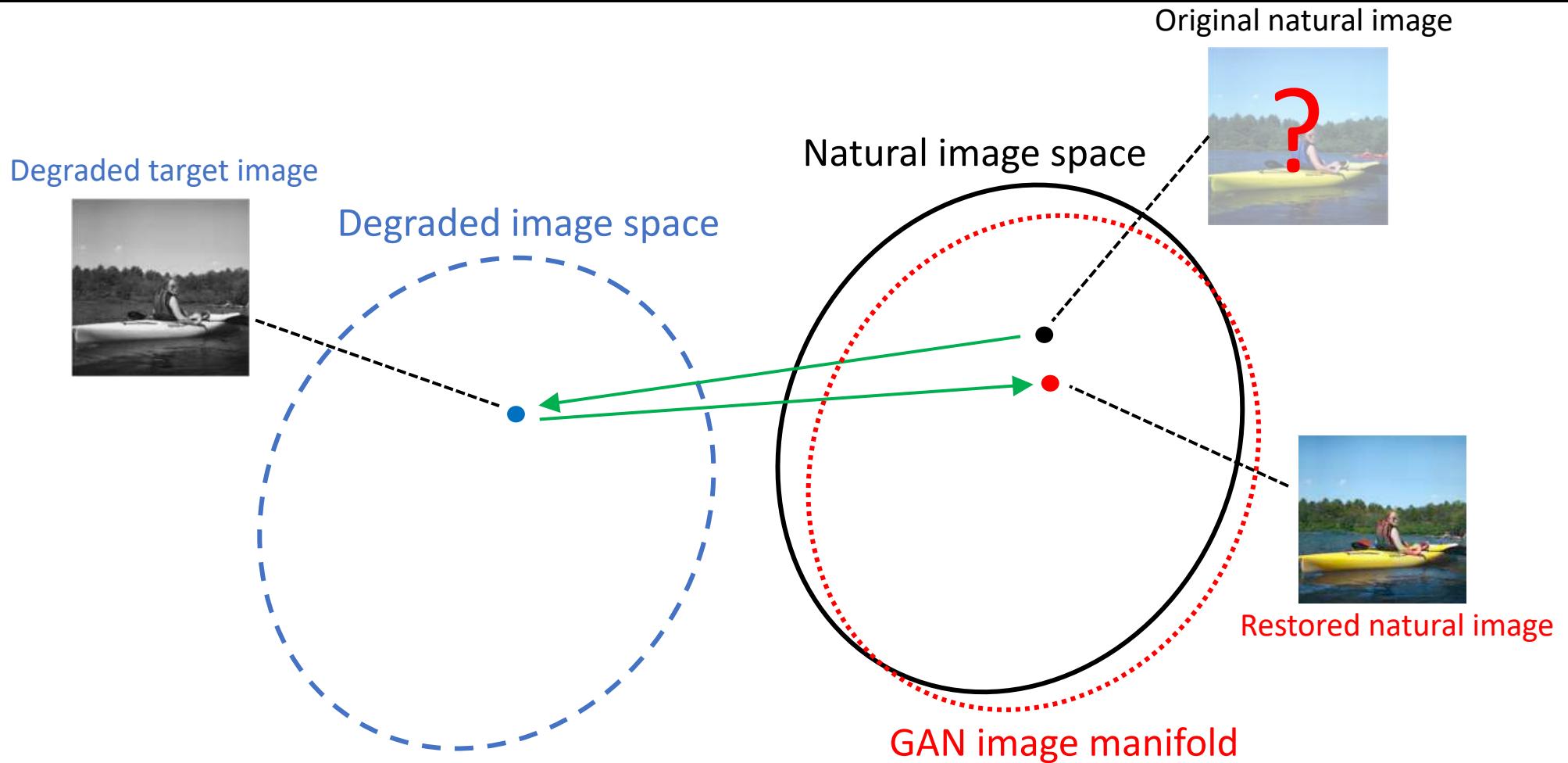
GAN as a Prior for Image Restoration and Manipulation

Motivation

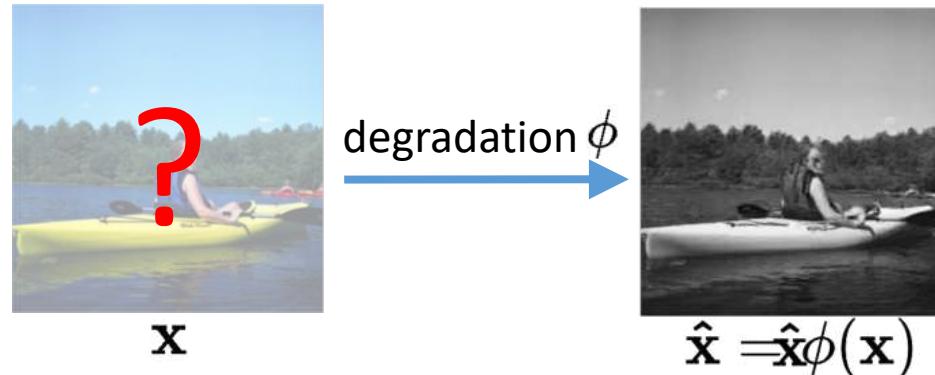
Our goal: exploit generic image prior of pretrained GAN



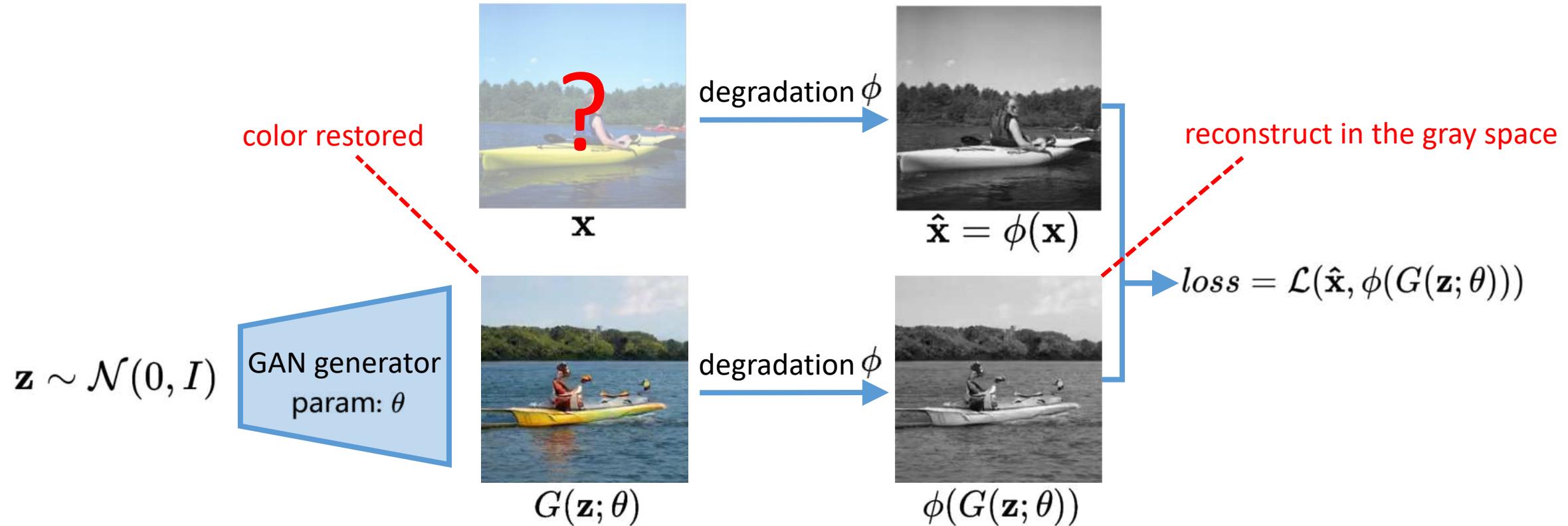
Deep Generative Prior



Deep Generative Prior

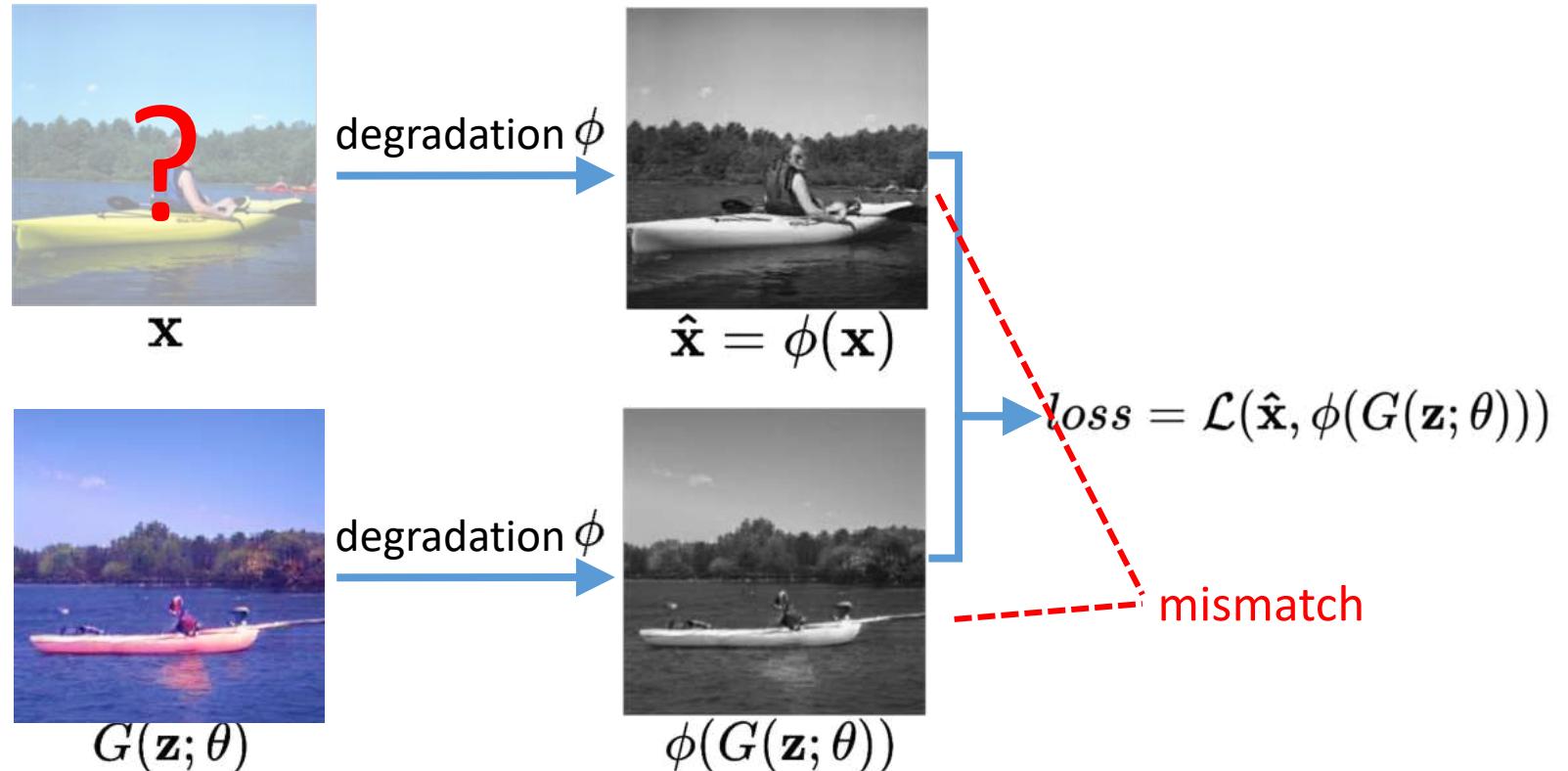
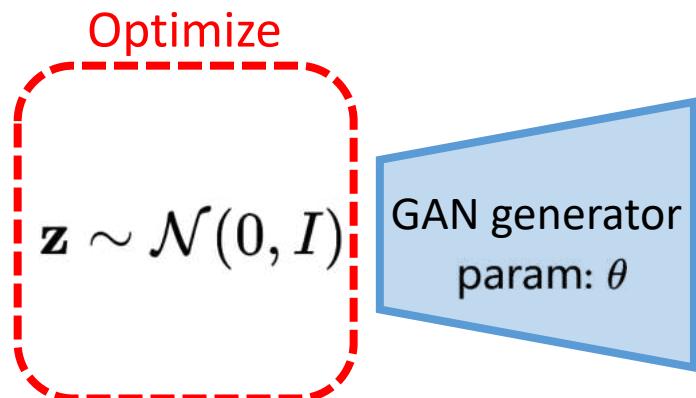


Deep Generative Prior



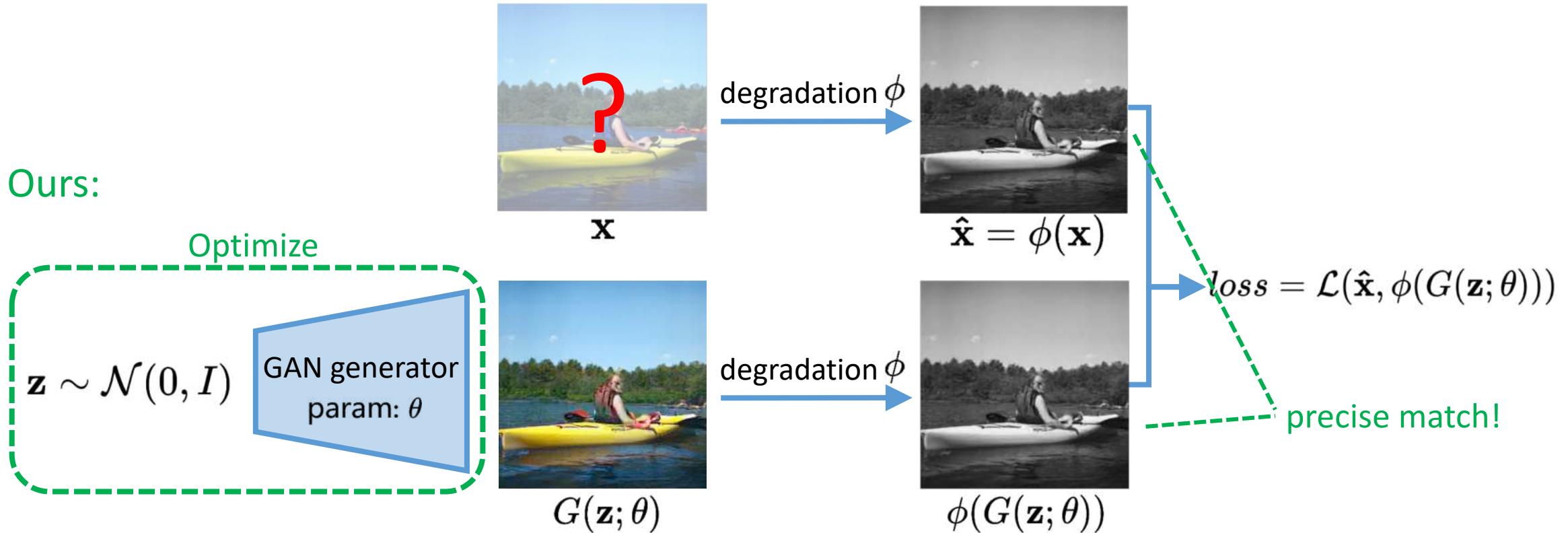
Deep Generative Prior

Conventional GAN-Inversion:



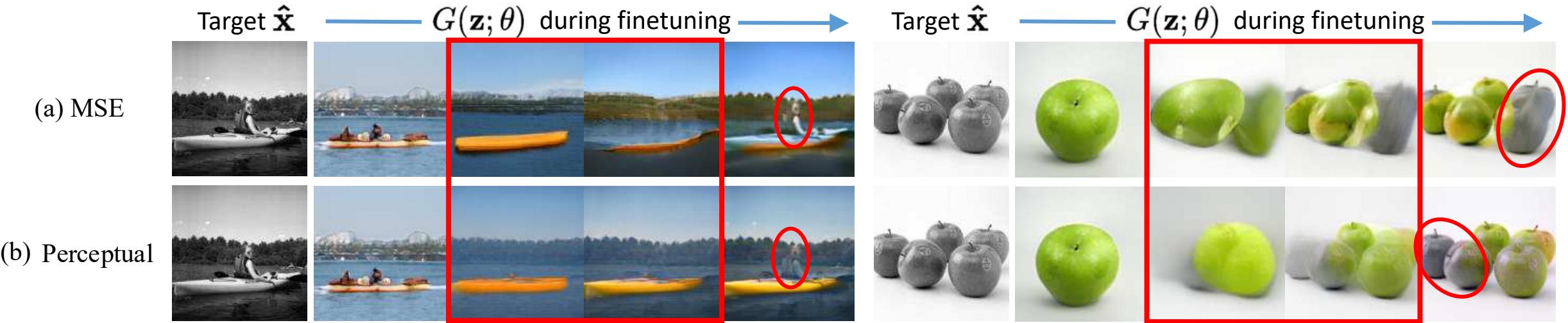
$$\mathbf{z}^* = \operatorname{argmin}_{\mathbf{z} \in R^d} \mathcal{L}(\hat{\mathbf{x}}, \phi(G(\mathbf{z}; \theta)))$$

Deep Generative Prior



$$\theta^*, \mathbf{z}^* = \operatorname{argmin}_{\theta, \mathbf{z}} \mathcal{L}(\hat{\mathbf{x}}, \phi(G(\mathbf{z}; \theta))) \quad (\text{Relaxed GAN-inversion})$$

Conventional Loss



Discriminator Feature Matching Loss

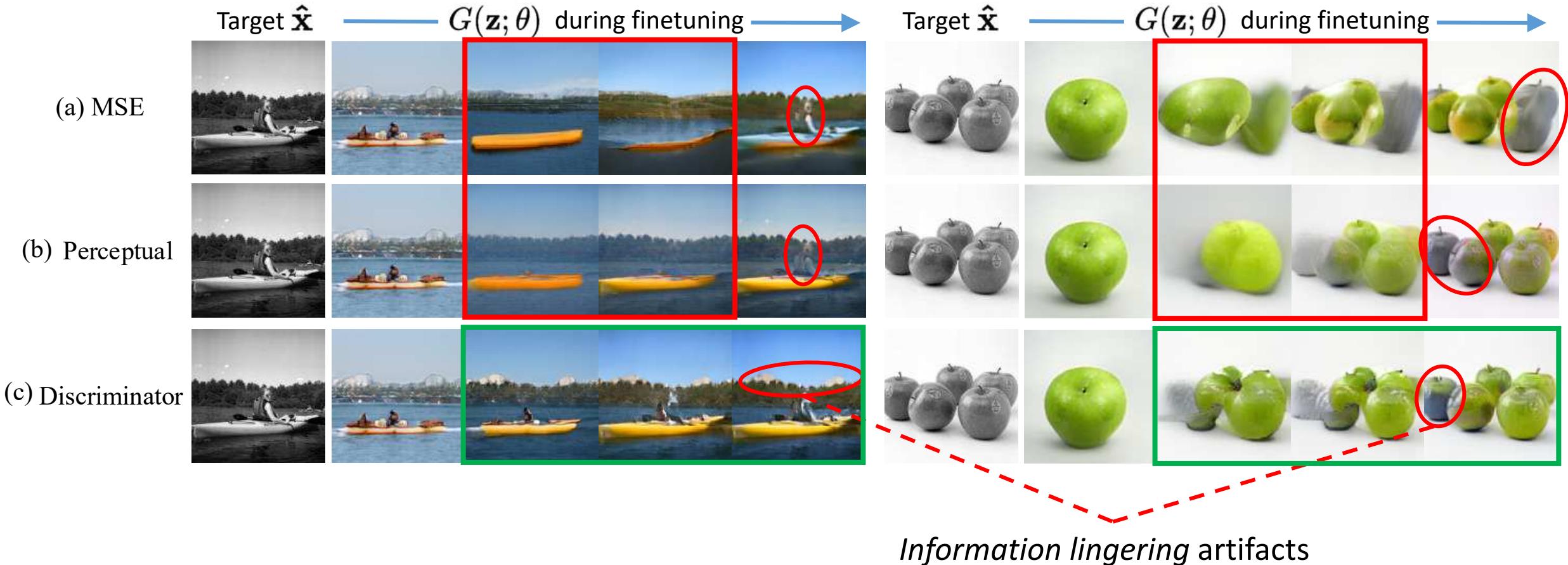


L1 distance in the discriminator feature space:

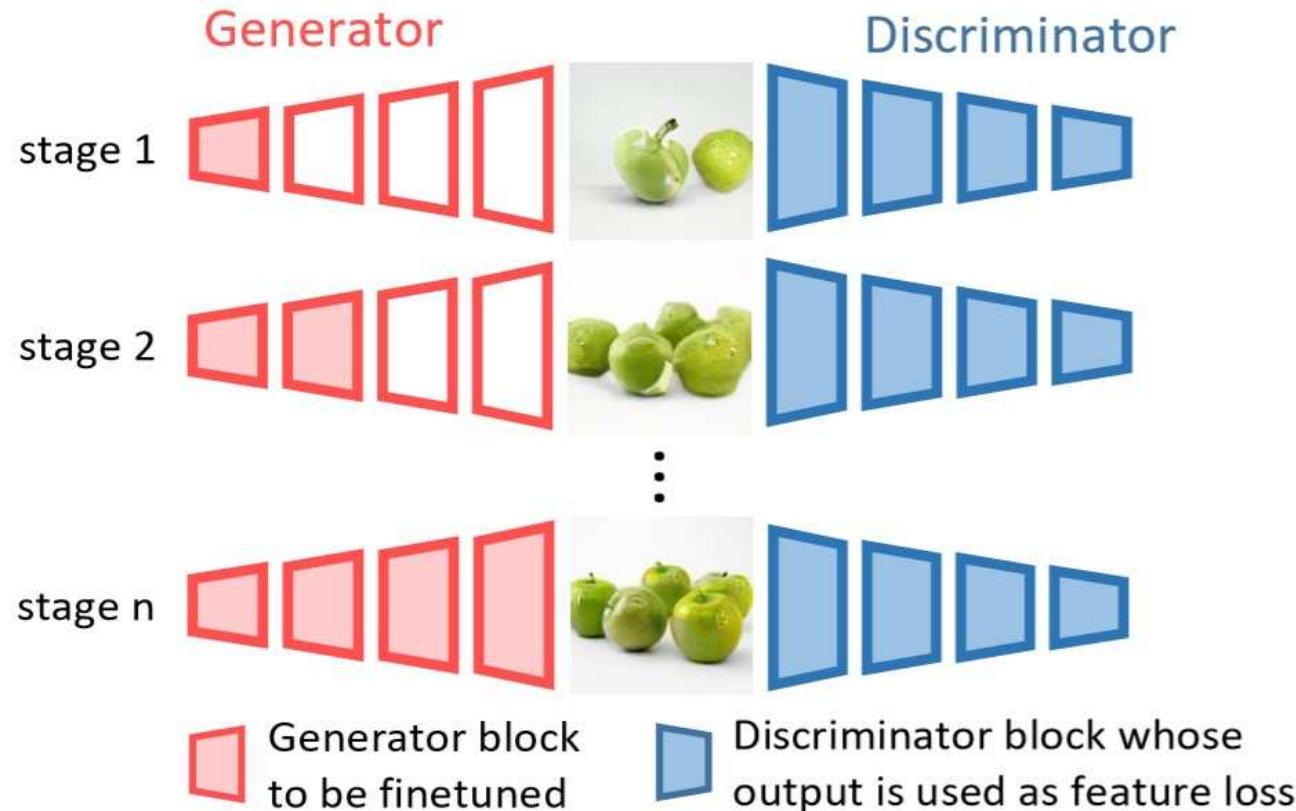
$$\mathcal{L}(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i \in \mathcal{I}} \|D(\mathbf{x}_1, i), D(\mathbf{x}_2, i)\|_1$$

$D(\mathbf{x}, i)$ returns the feature of \mathbf{x} at the i 'th block of the discriminator.

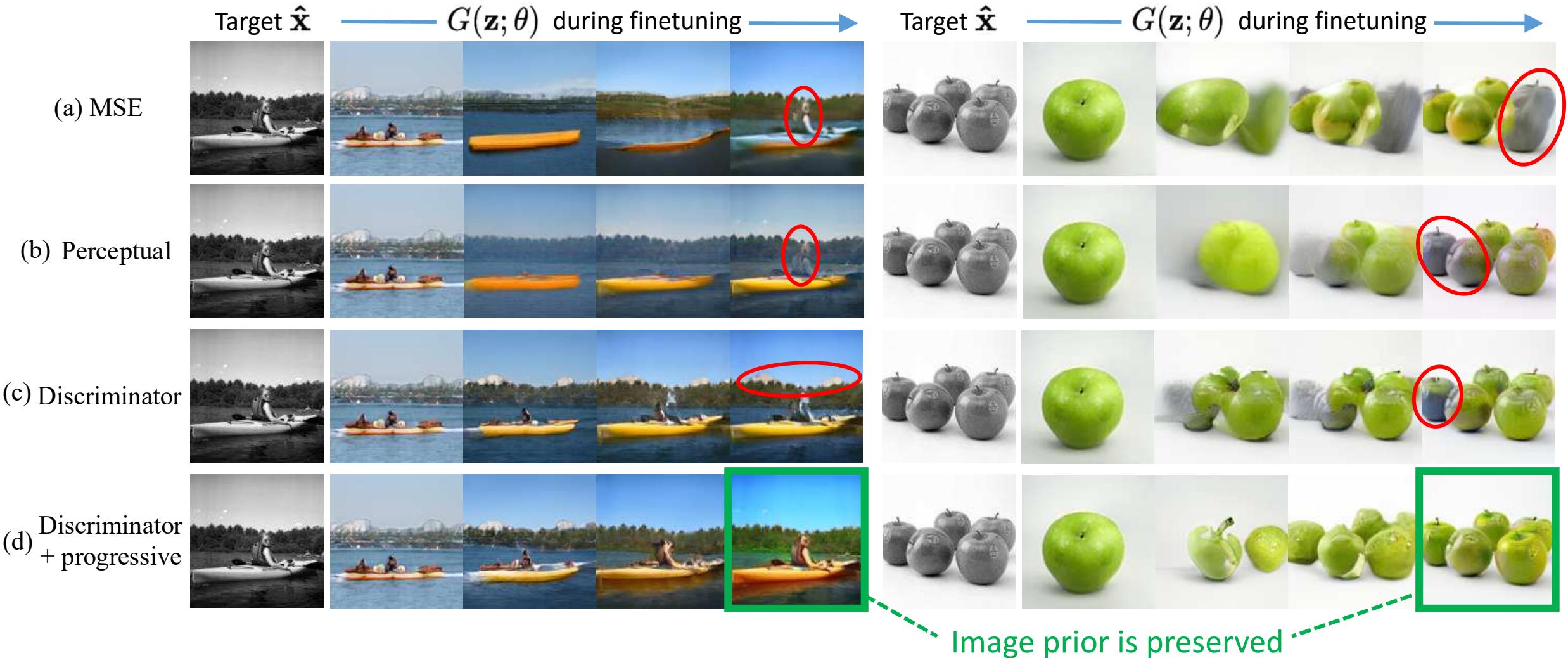
Discriminator Feature Matching Loss



Progressive Reconstruction



Comparison of Different Losses



Applications

Degradation transform $\phi(\mathbf{x})$ for different tasks:

Colorization: $\phi(\mathbf{x}) = 0.2989\mathbf{x}_r + 0.5870\mathbf{x}_g + 0.1140\mathbf{x}_b$

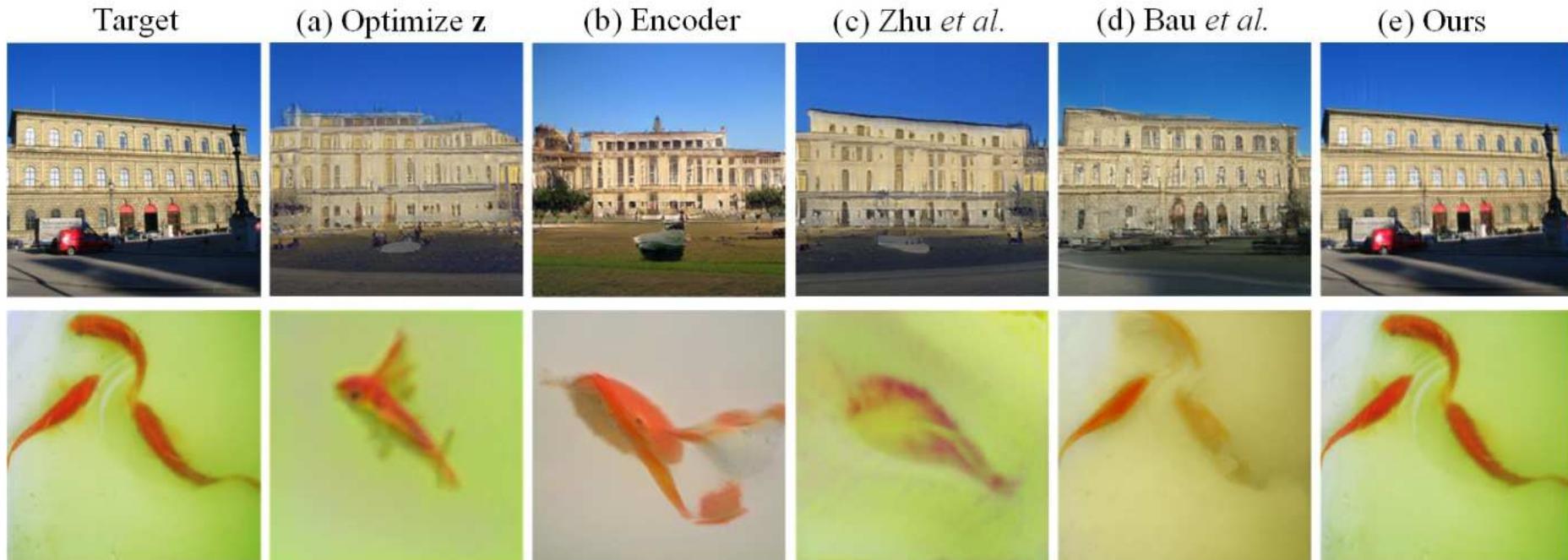
Inpainting: $\phi(\mathbf{x}) = \mathbf{x} \odot \mathbf{m}$ \mathbf{m} is inpainting mask

Super-resolution: $\phi(\mathbf{x})$ is Lanczos downsampling operator

Adversarial defense: $\phi(\mathbf{x}) = \mathbf{x} + \Delta\mathbf{x}$ $\Delta\mathbf{x}$ is the adversarial perturbation

Model: BigGAN trained on ImageNet training set (Brock et al. ICLR2018)

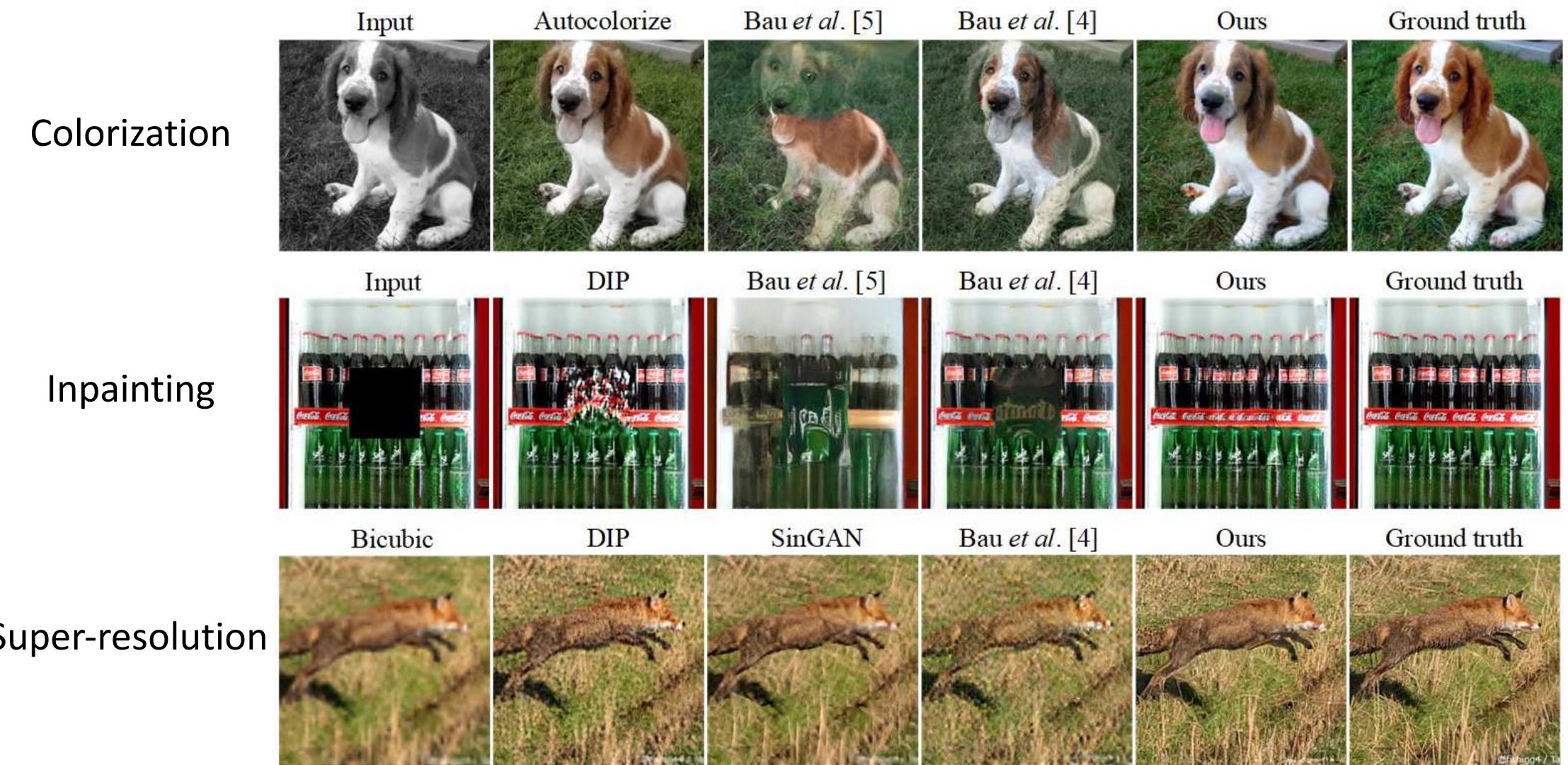
Application: GAN Inversion



	(a)	(b)	(c)	(d)	Ours
PSNR↑	15.97	11.39	16.46	22.49	32.89
SSIM↑	46.84	32.08	47.78	73.17	95.95
MSE↓ ($\times 10^{-3}$)	29.61	85.04	28.32	6.91	1.26

- (a) Optimize z (Creswell et al. 2018)
(b) Encoder (Zhu et al. ECCV2016)
(c) Encoder + Optimize z (Zhu et al. ECCV2016)
(d) Encoder + Optimize z + Perturbation on features
(Bau et al. ICCV2019)

Application: Image Restoration



Demo: Image Restoration

Colorization



Inpainting



Super-resolution



Application: Image Restoration

Colorization



Inpainting



Super-
Resolution



Generalization to non-ImageNet Images



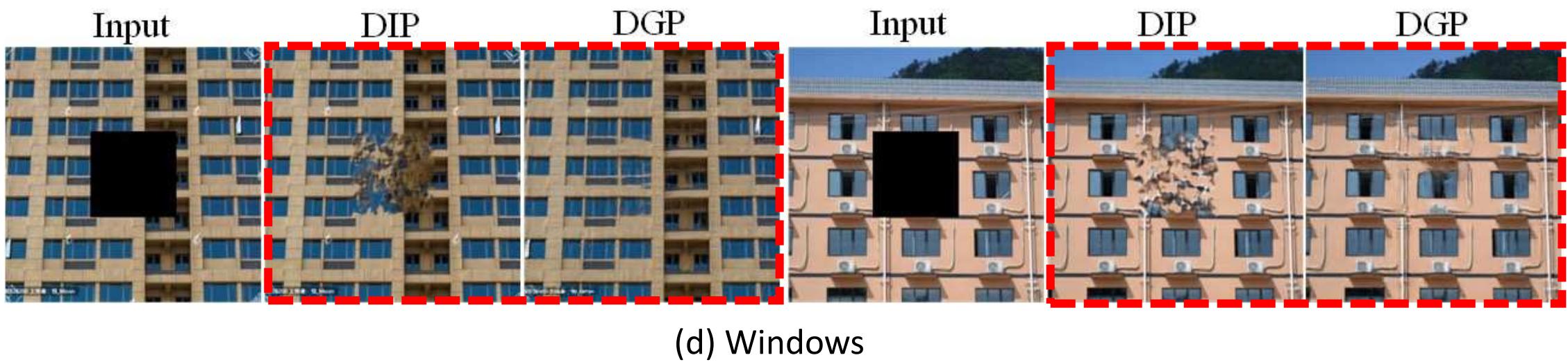
(a) Raccoon



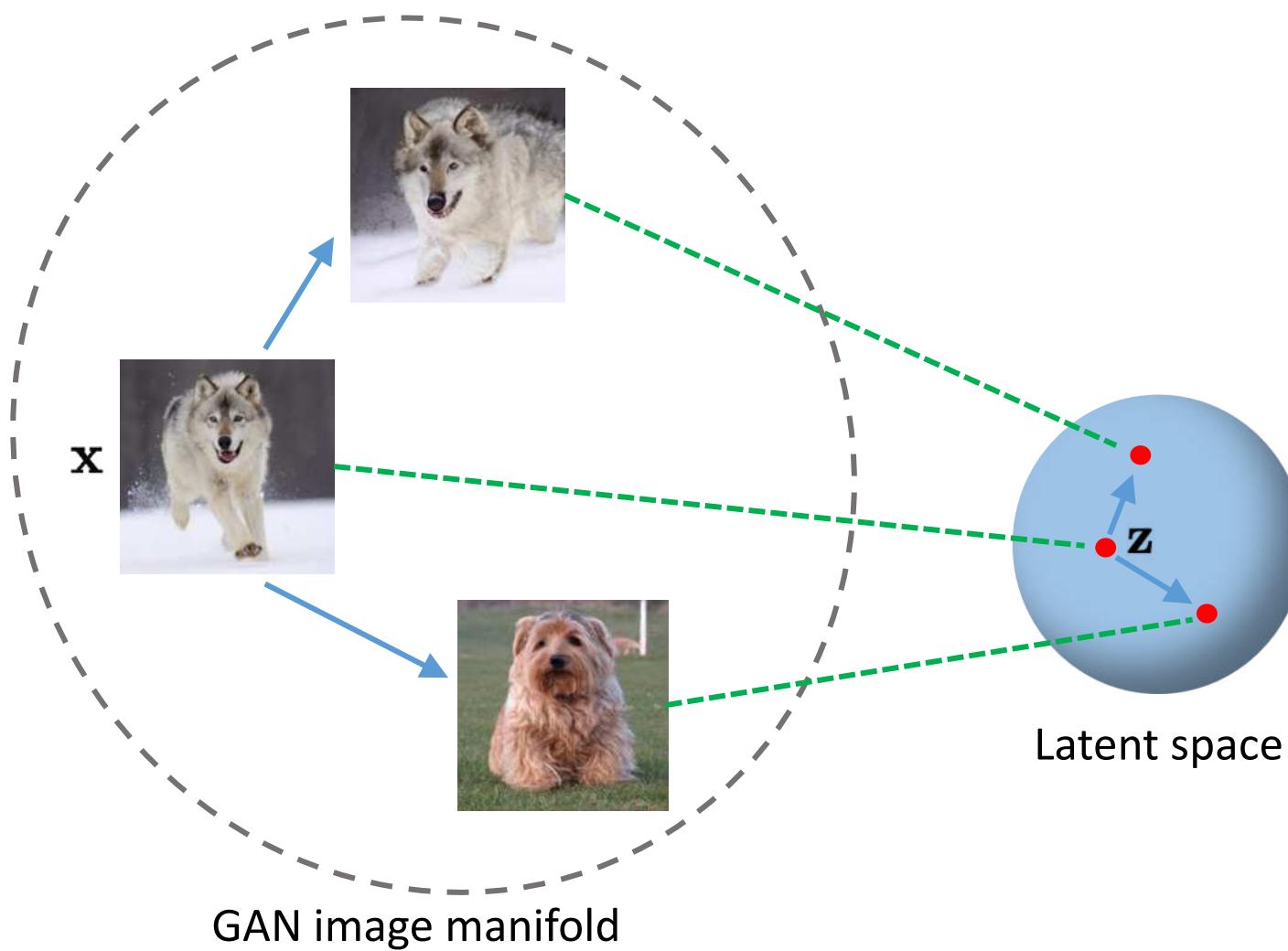
(b) Places



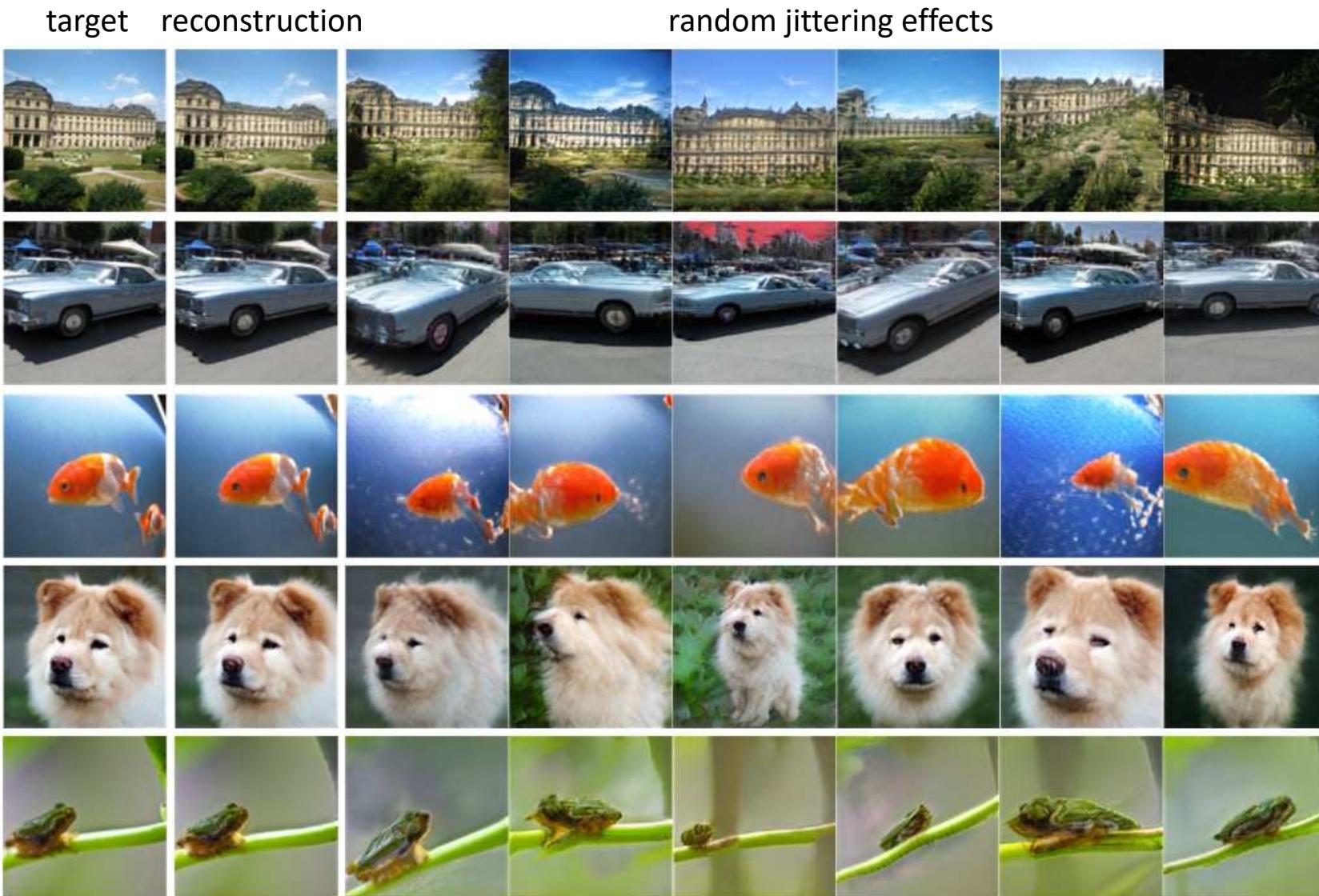
(c) No foreground



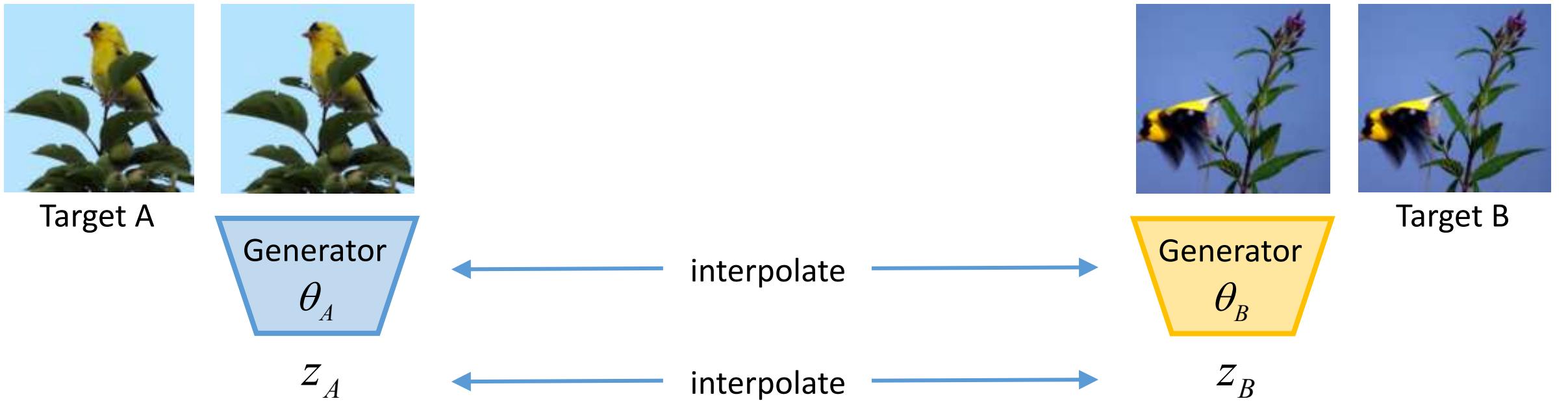
Application: Image Manipulation



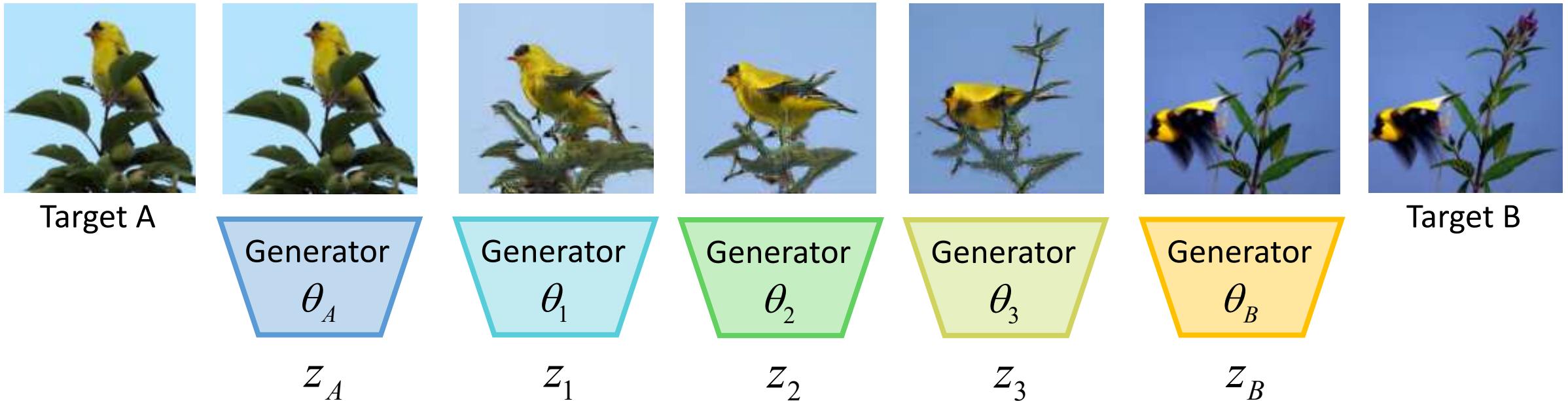
Application: Random Jittering



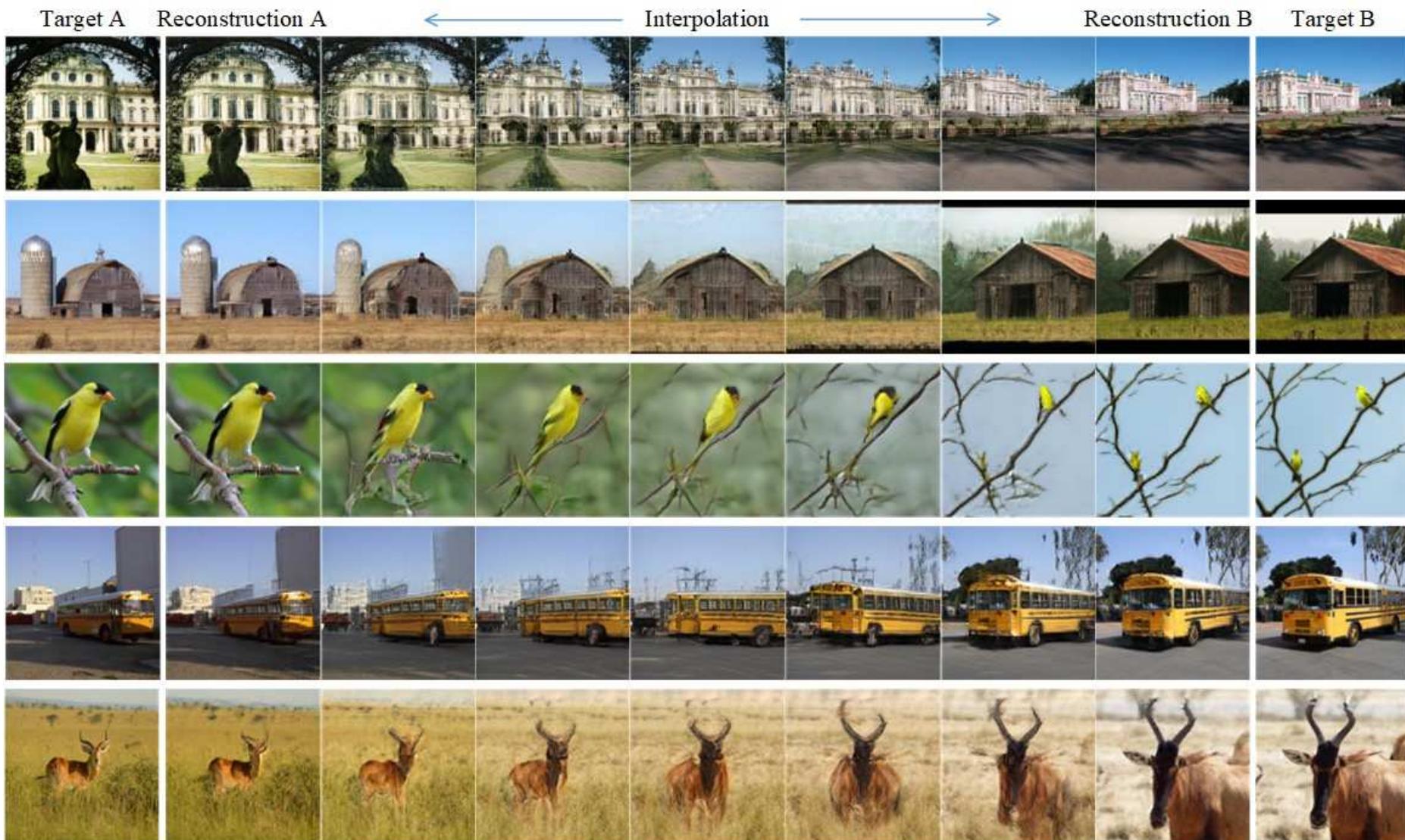
Application: Image Morphing



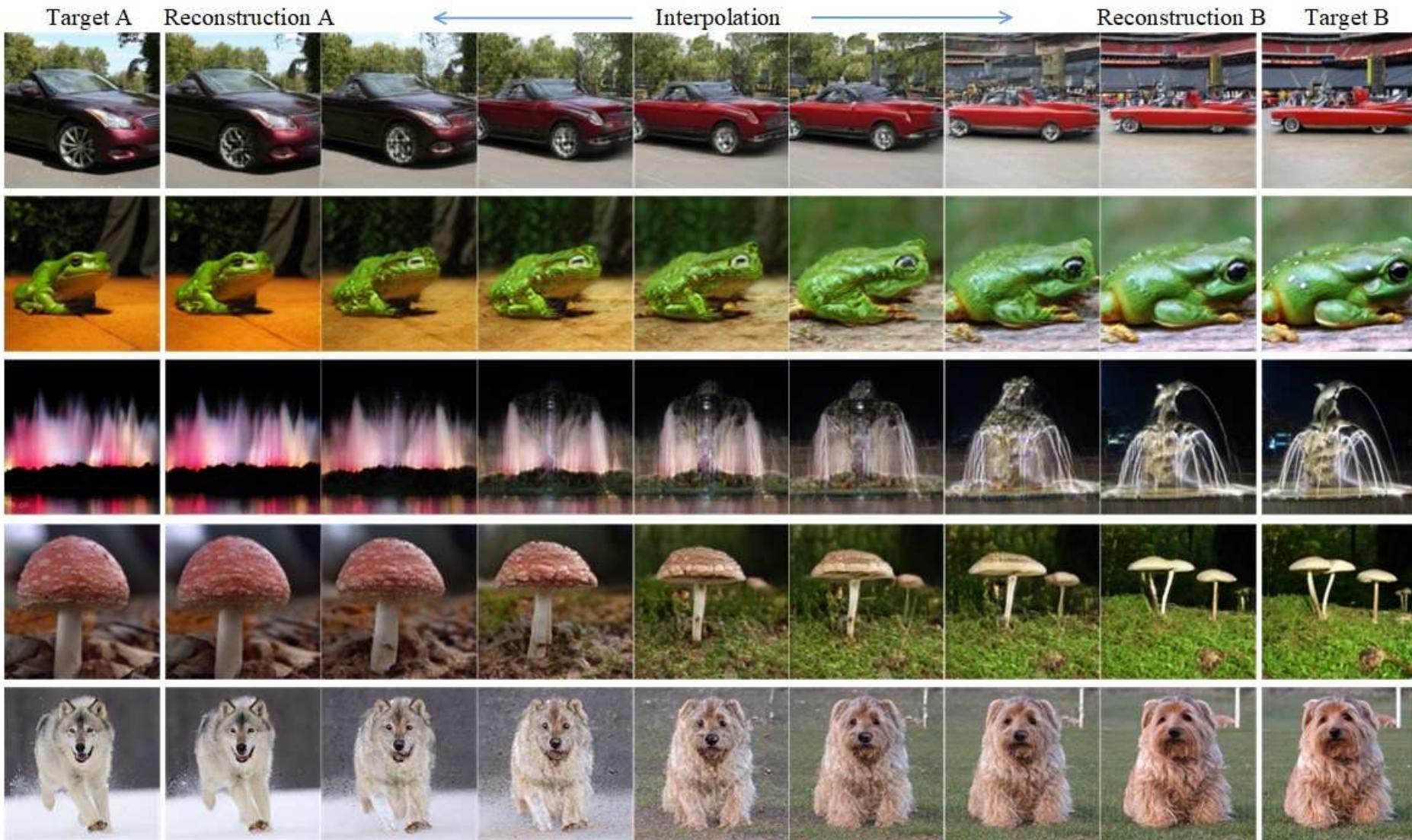
Application: Image Morphing



Application: Image Morphing



Application: Image Morphing



Application: Category Transfer

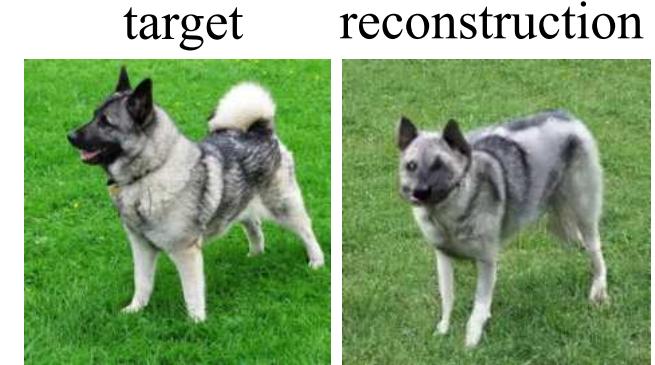
target reconstruct



transfer to other categories



Demo: Image Manipulation



Random jittering



Image morphing



Category transfer



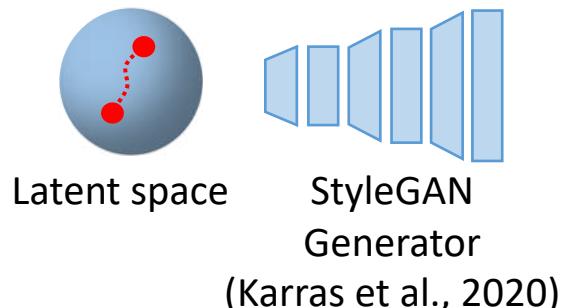
Exploiting 2D GAN Prior for 3D Generation

Do 2D GANs Model 3D Geometry?

Natural images are projections of 3D objects on a 2D image plane.

An ideal 2D image manifold (e.g., GAN) should capture 3D geometric properties.

The following example shows that there is a direction in the GAN image manifold that corresponds to viewpoint variation.



Can we Make Use of such Variations?

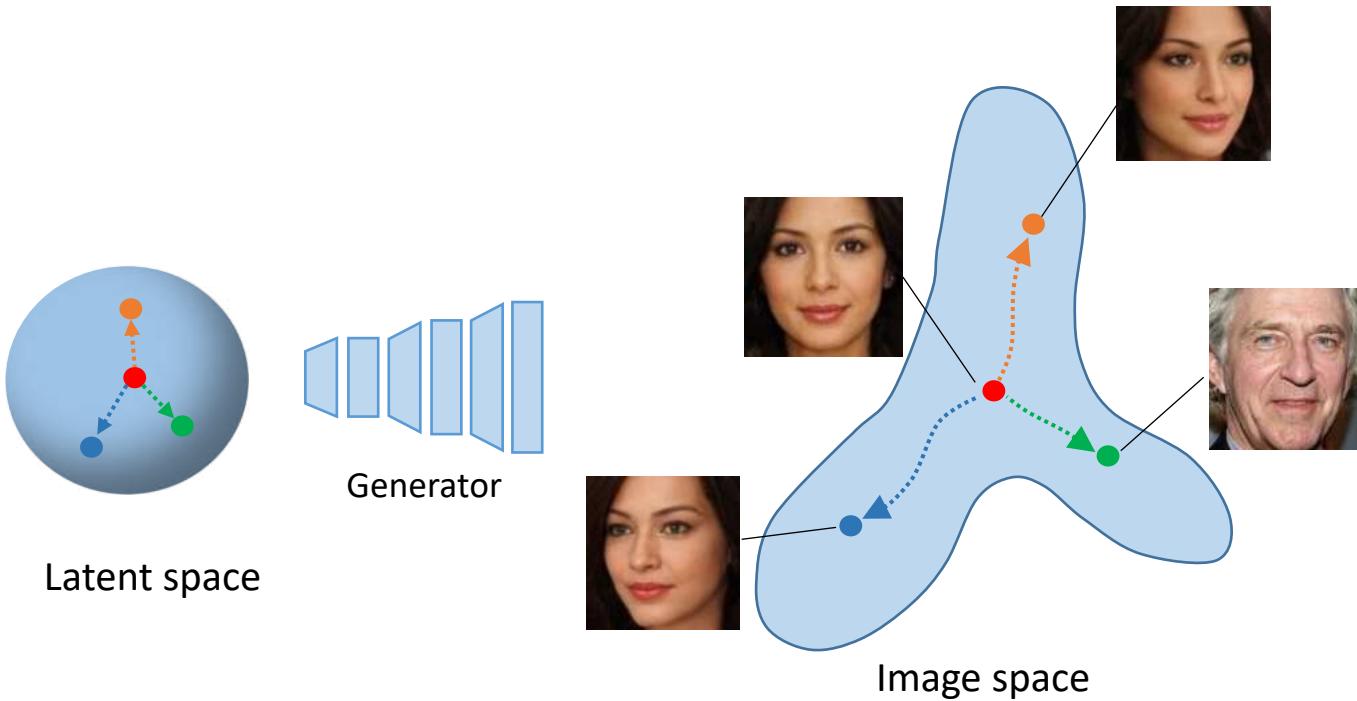
Can we make use of such variations for 3D reconstruction?

If we have multiple **viewpoint** and **lighting** variations of the same instance, we can infer its 3D structure.

Let's create these variations by exploiting the image manifold captured by 2D GANs!



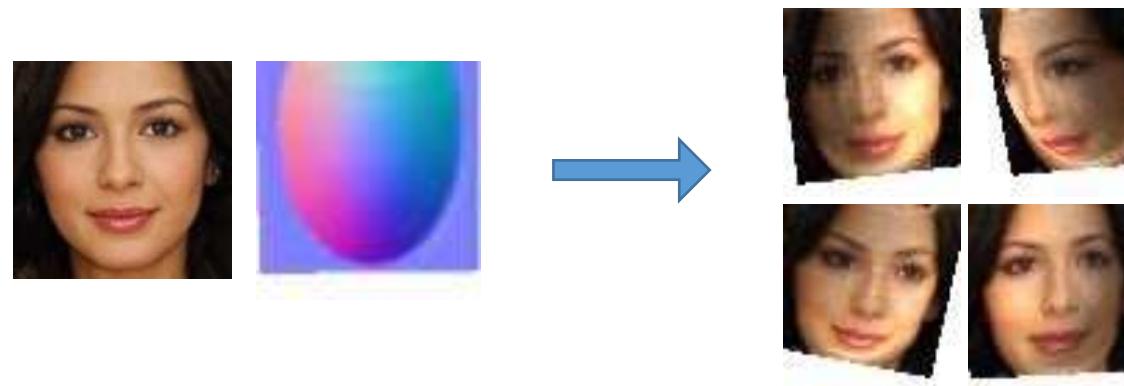
Challenge



It is non-trivial to find **well-disentangled latent directions** that control *viewpoint* and *lighting* variations in an unsupervised manner.

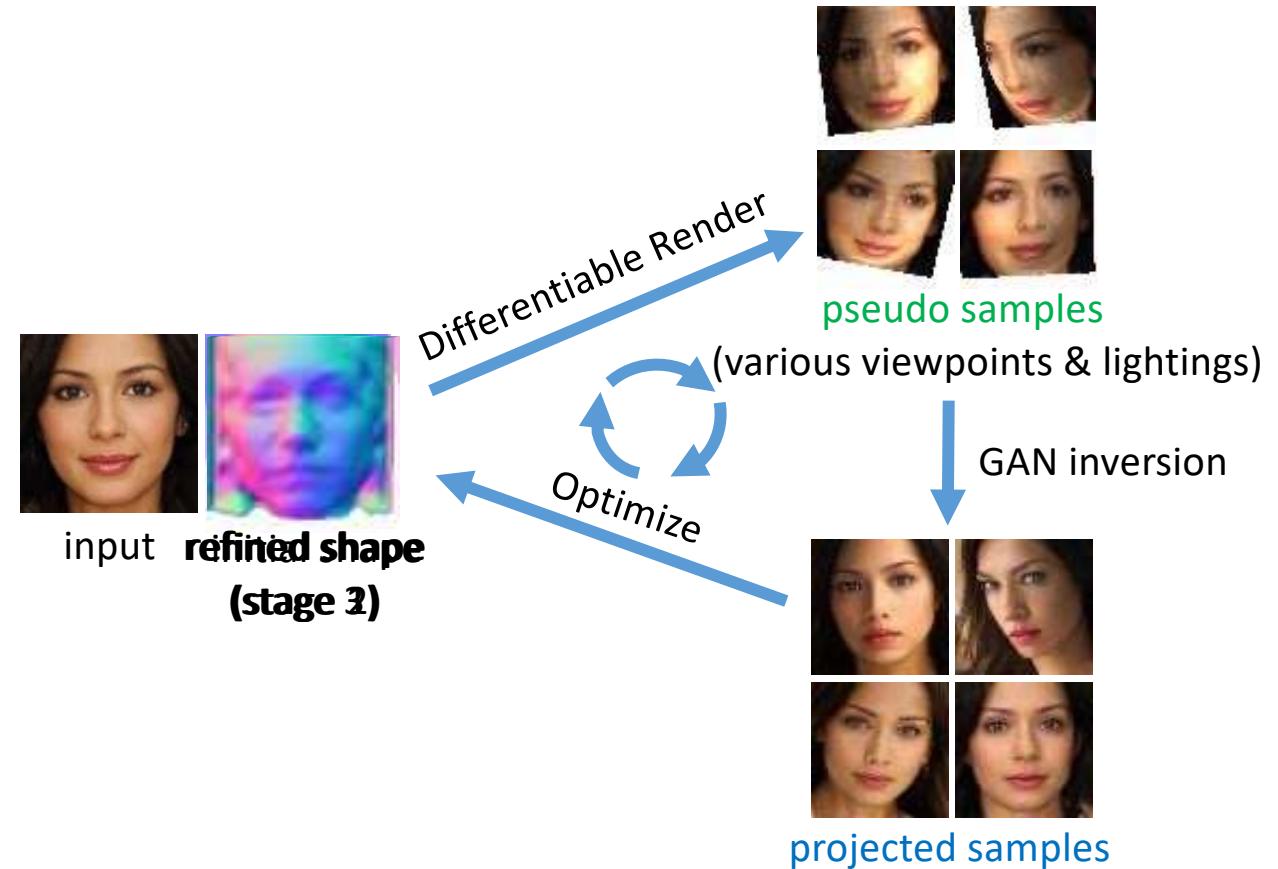
Our Solution

For many objects such as faces and cars, a **convex shape prior like ellipsoid** could provide a hint on the change of their viewpoints and lighting conditions.



GAN2Shape - Overview

- Initialize the shape with ellipsoid.
- Render '*pseudo samples*' with different viewpoints and lighting conditions.
- GAN inversion is applied to these samples to obtain the '*projected samples*'.
- '*Projected samples*' are used as the ground truth of the rendering process to optimize the 3D shape.
- Iterative training to progressively refine the shape.

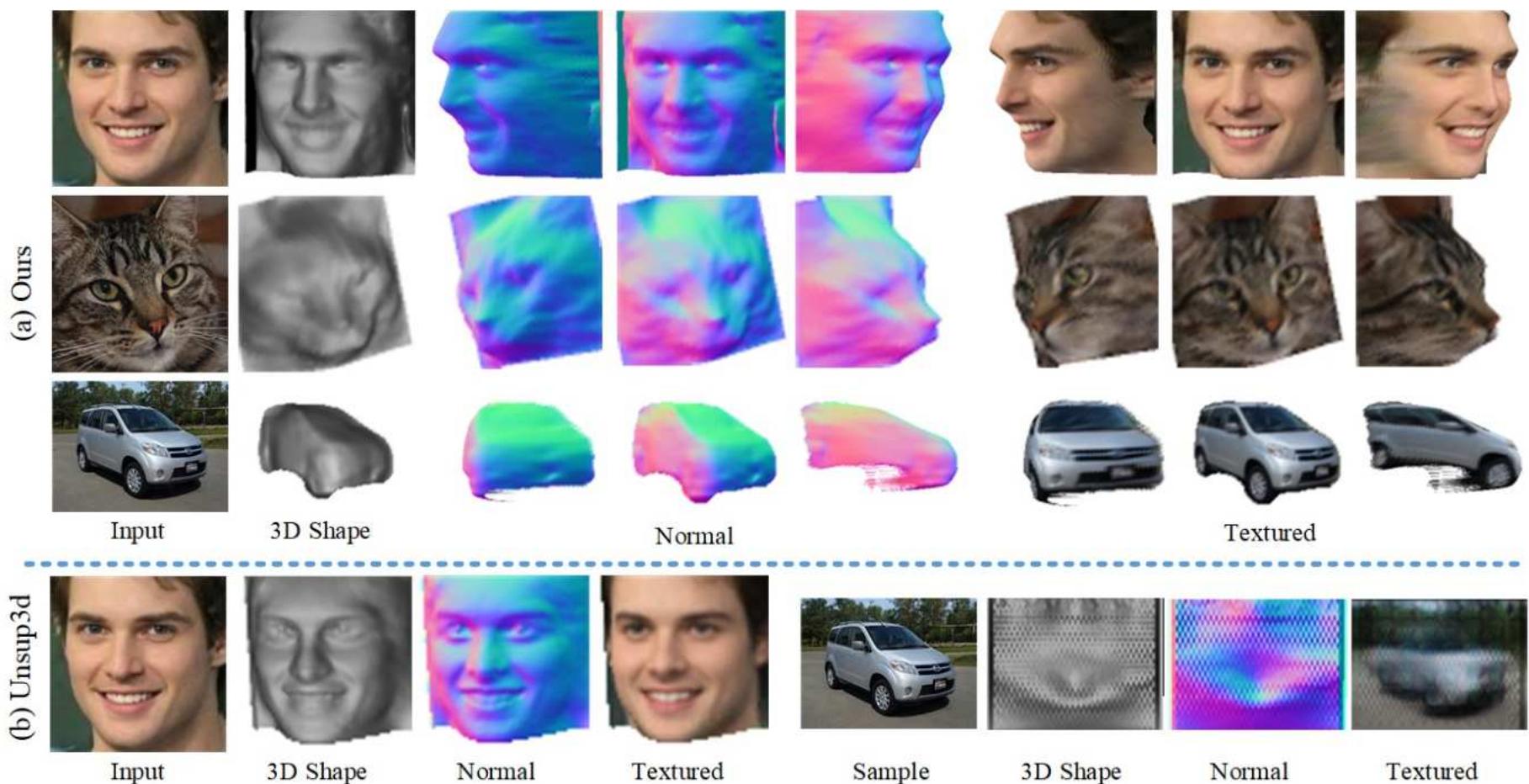


3D Generation Results

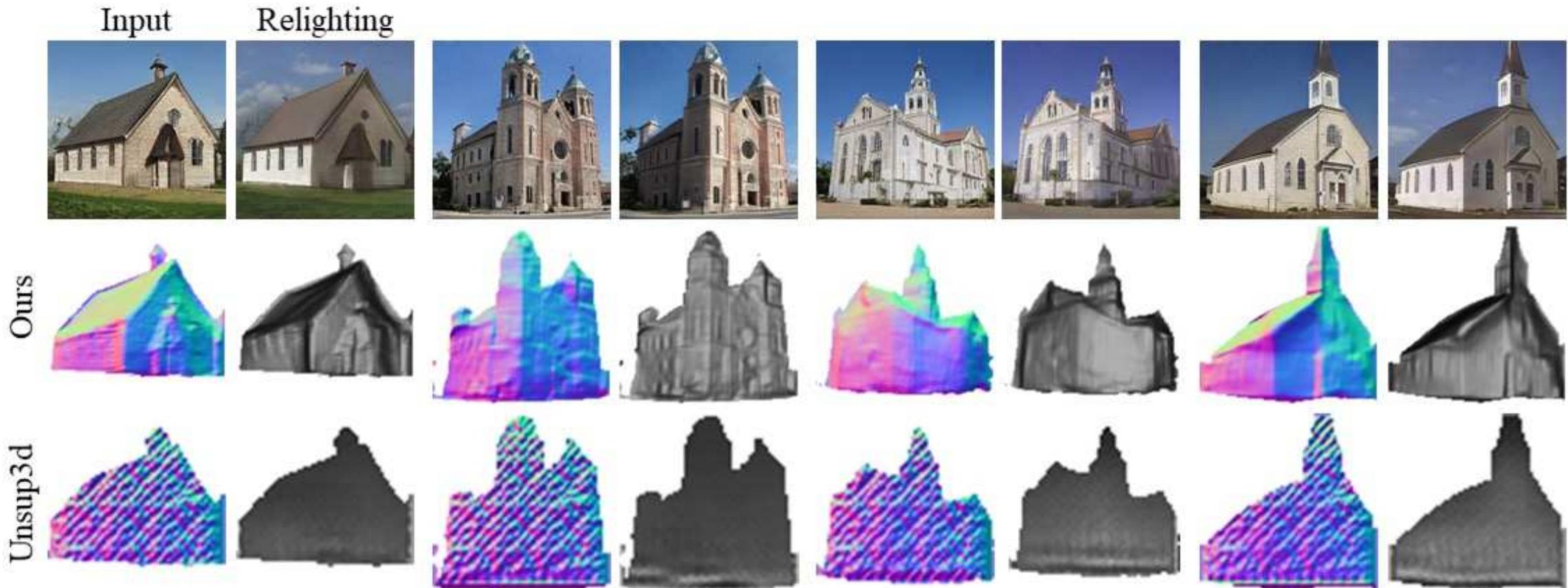
Without any 2D keypoint or 3D annotations

Unsupervised 3D shape reconstruction from unconstrained 2D images

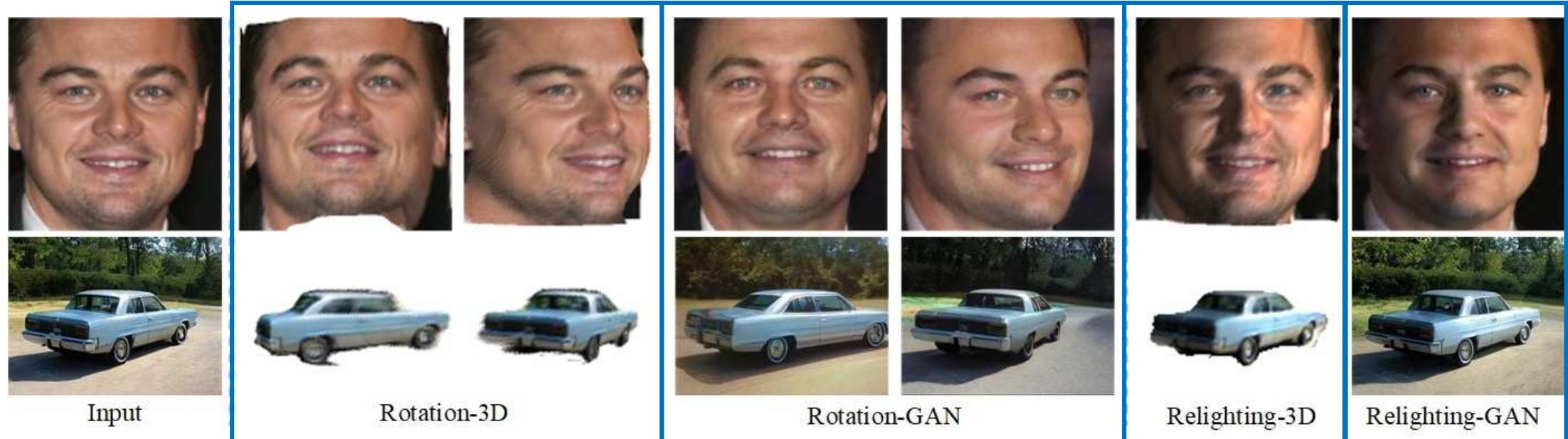
Without symmetry assumption
Work on many object categories such as human faces, cars, buildings, etc.



3D Generation Results

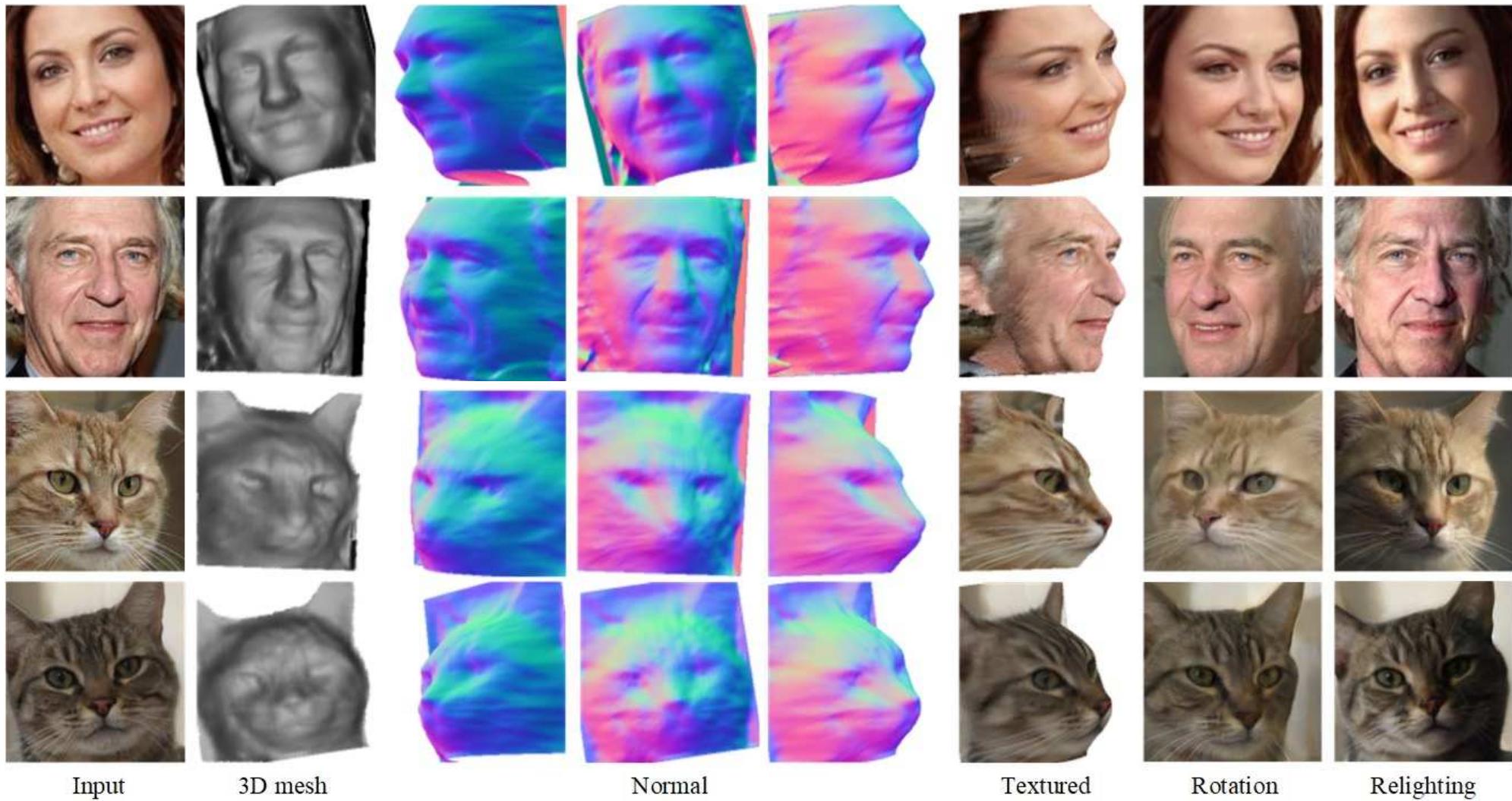


3D-aware Image Manipulation

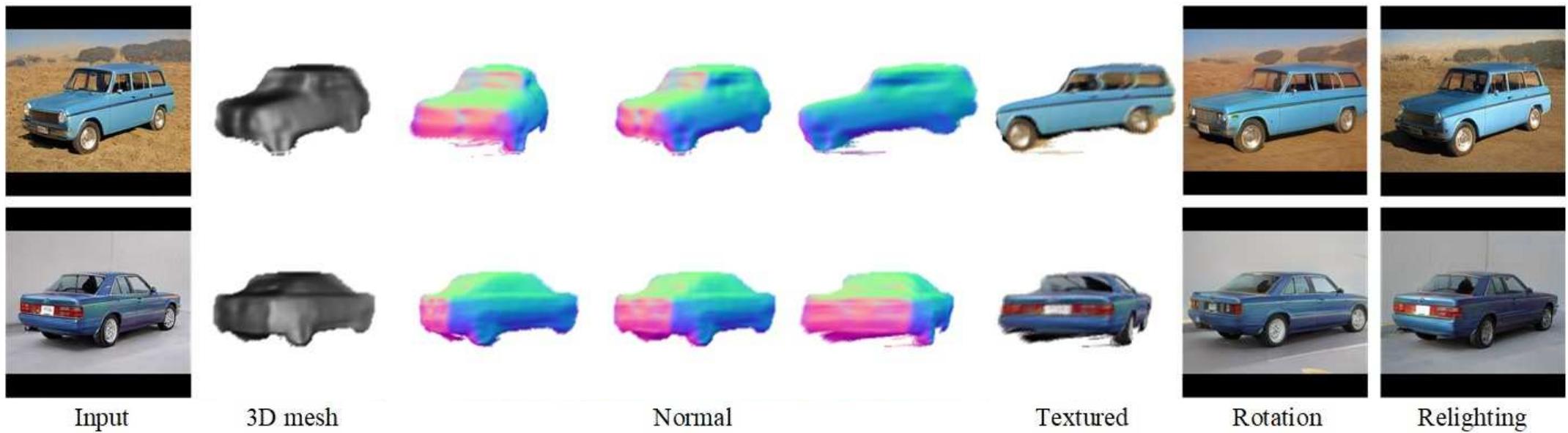


- Effect-3D: Rendered using the reconstructed 3D shape and albedo.
- Effect-GAN: project Effect-3D on the GAN image manifold using the trained encoder E .

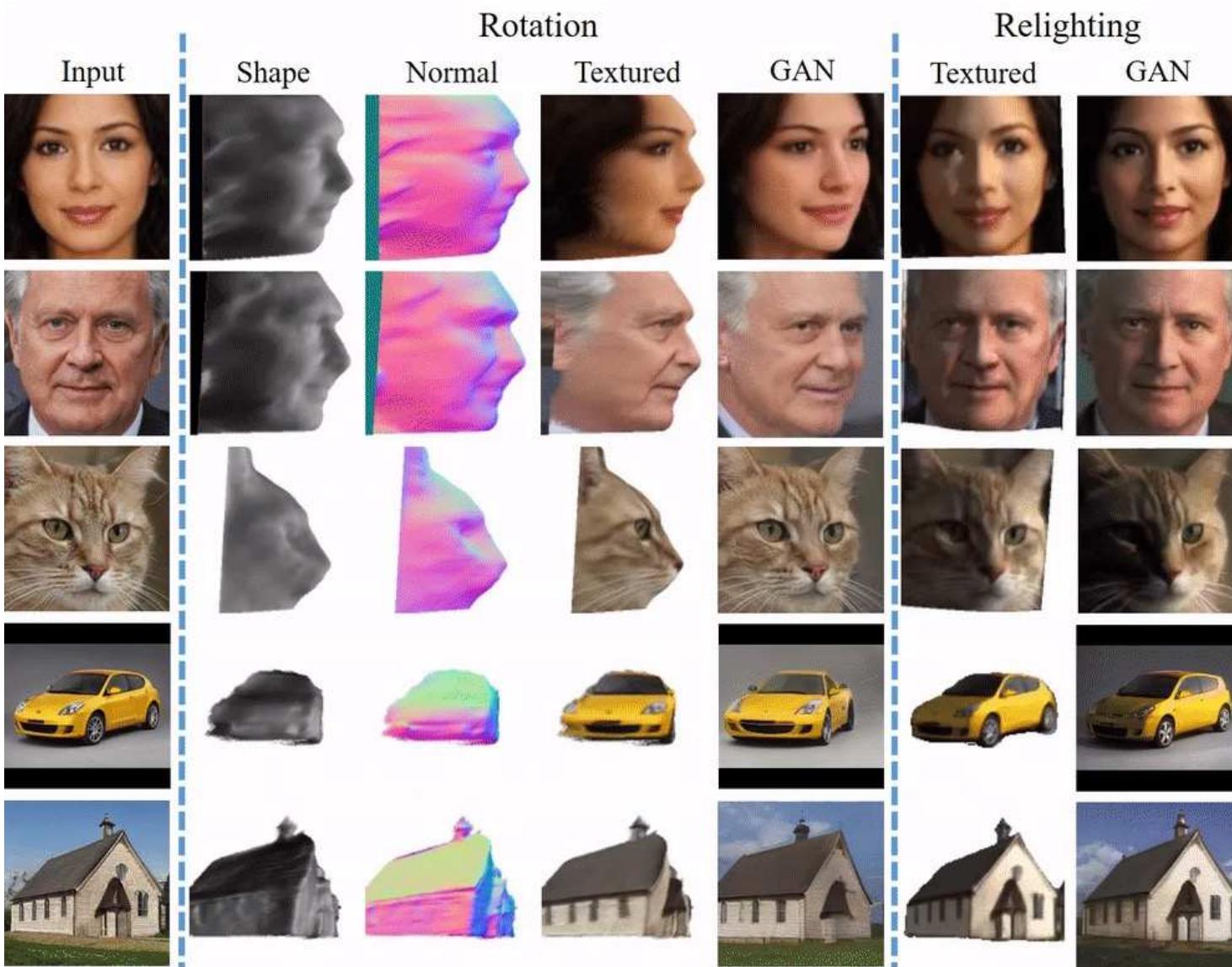
More Results



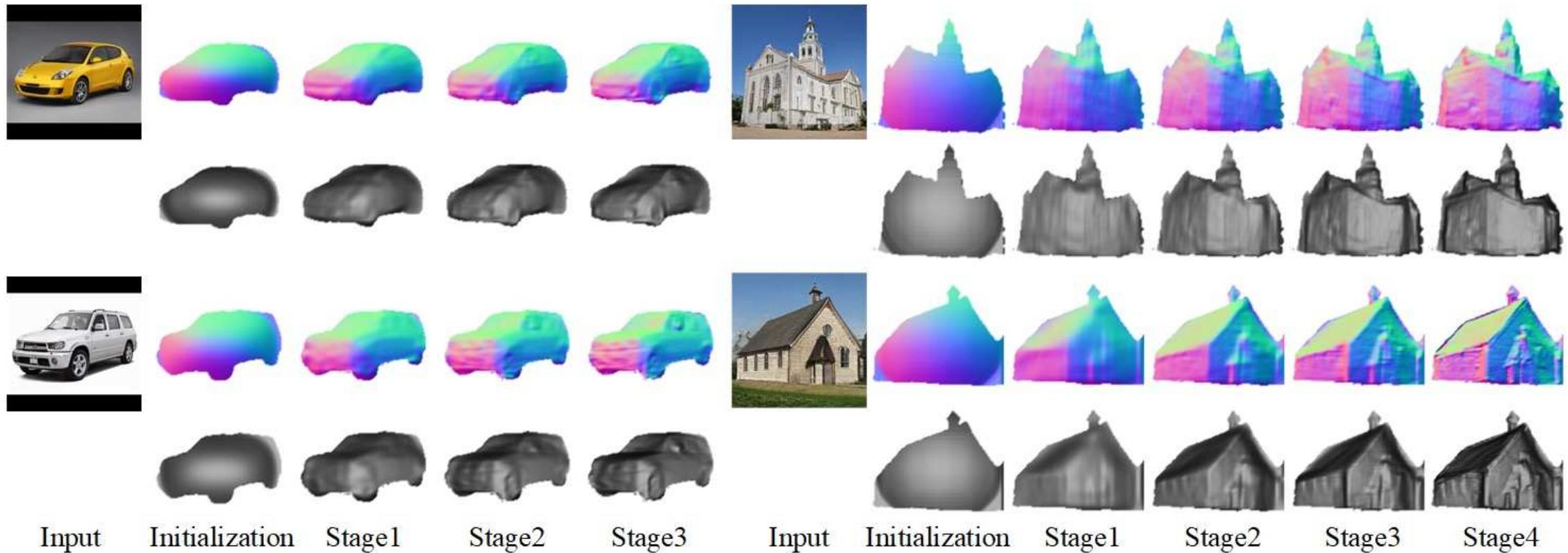
More Results



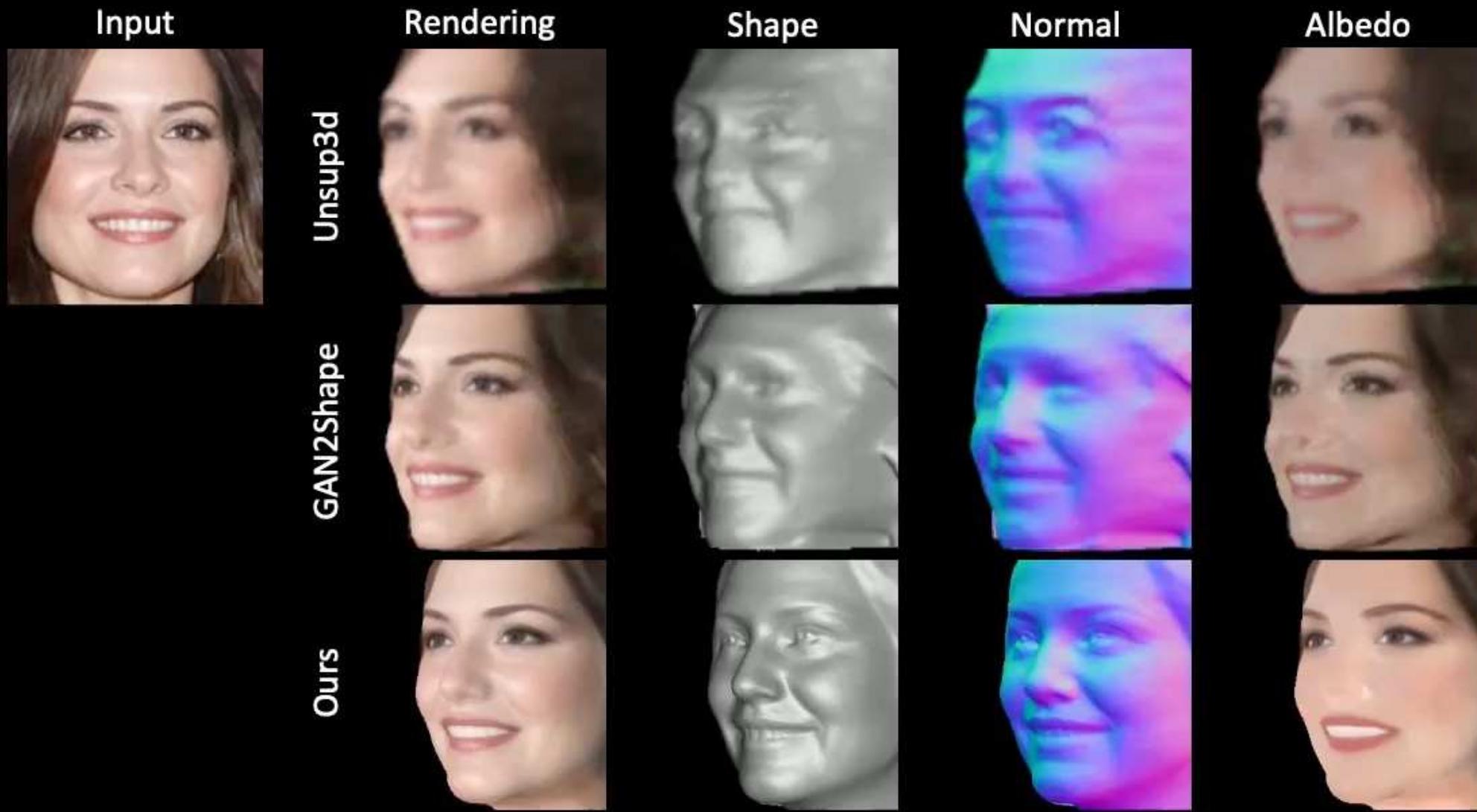
Demo



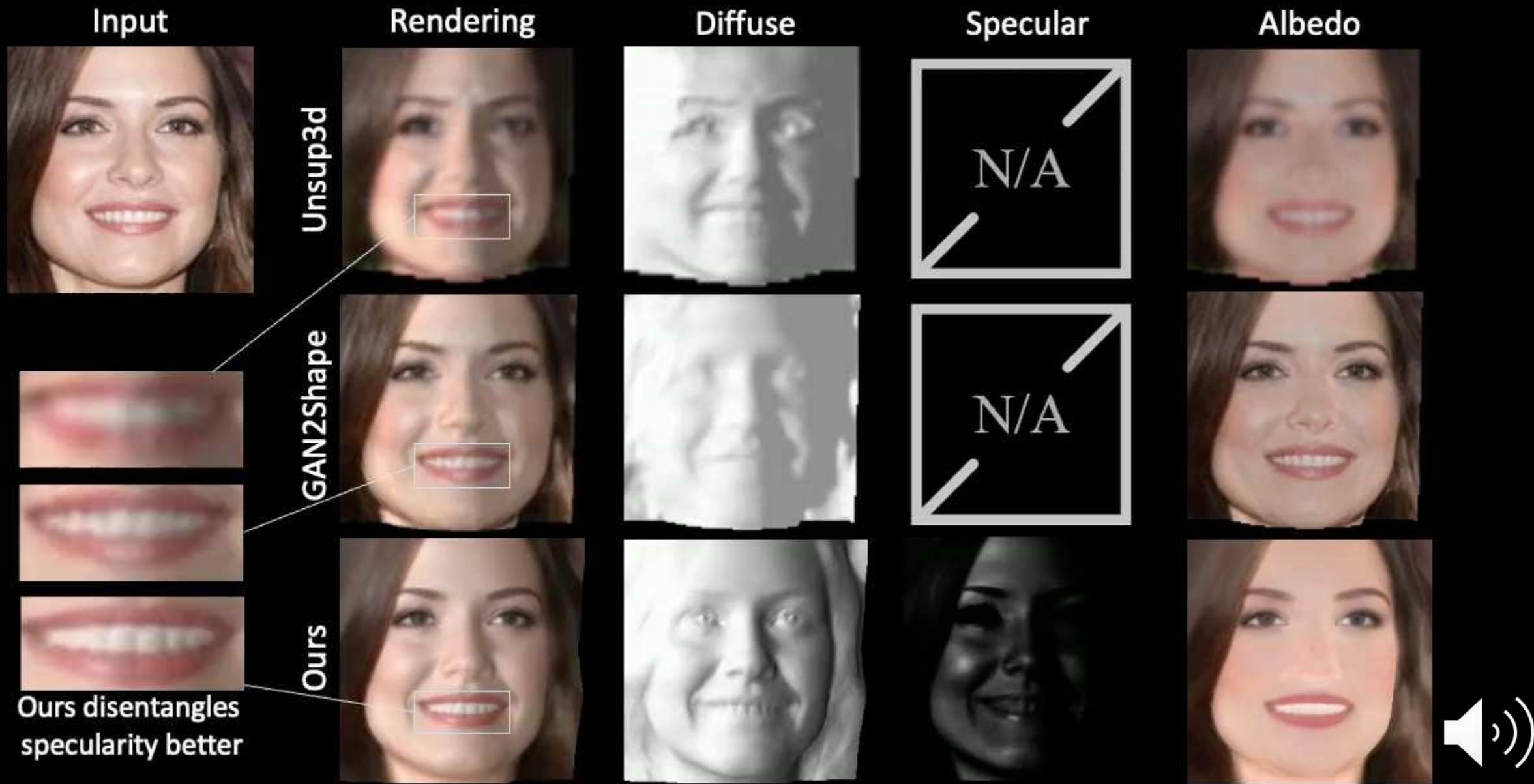
Effects of Iterative Training



Qualitative Comparison on CelebA: Rotation



Qualitative Comparison on CelebA: Relighting



Input



Rendering



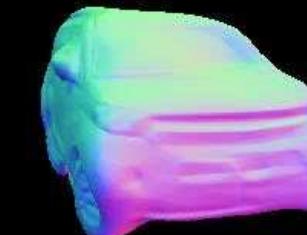
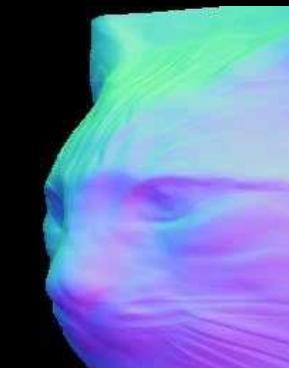
Shape



Normal



Albedo



Input



Rendering



Diffuse



Specular

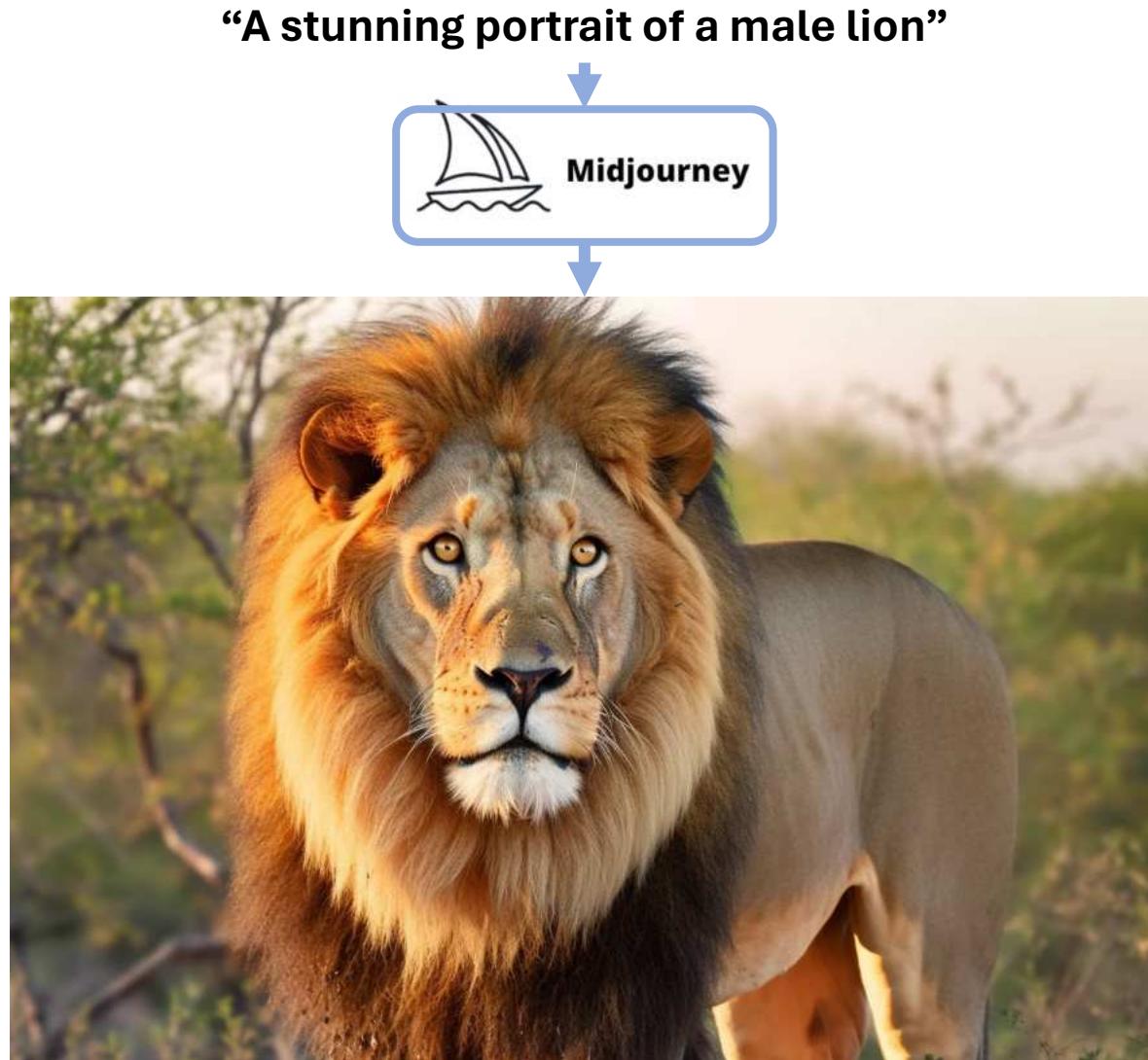


Albedo



Drag Your GAN

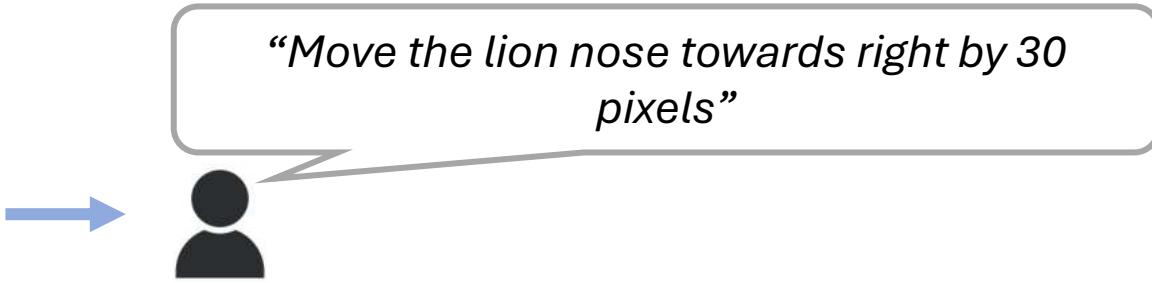
What's Missing in Image Synthesis



What if the user wants to ...

- Change the pose (e.g., rotate head)?
- Increase/reduce the animal size?
- Move the animal?
- Change the expression?
-

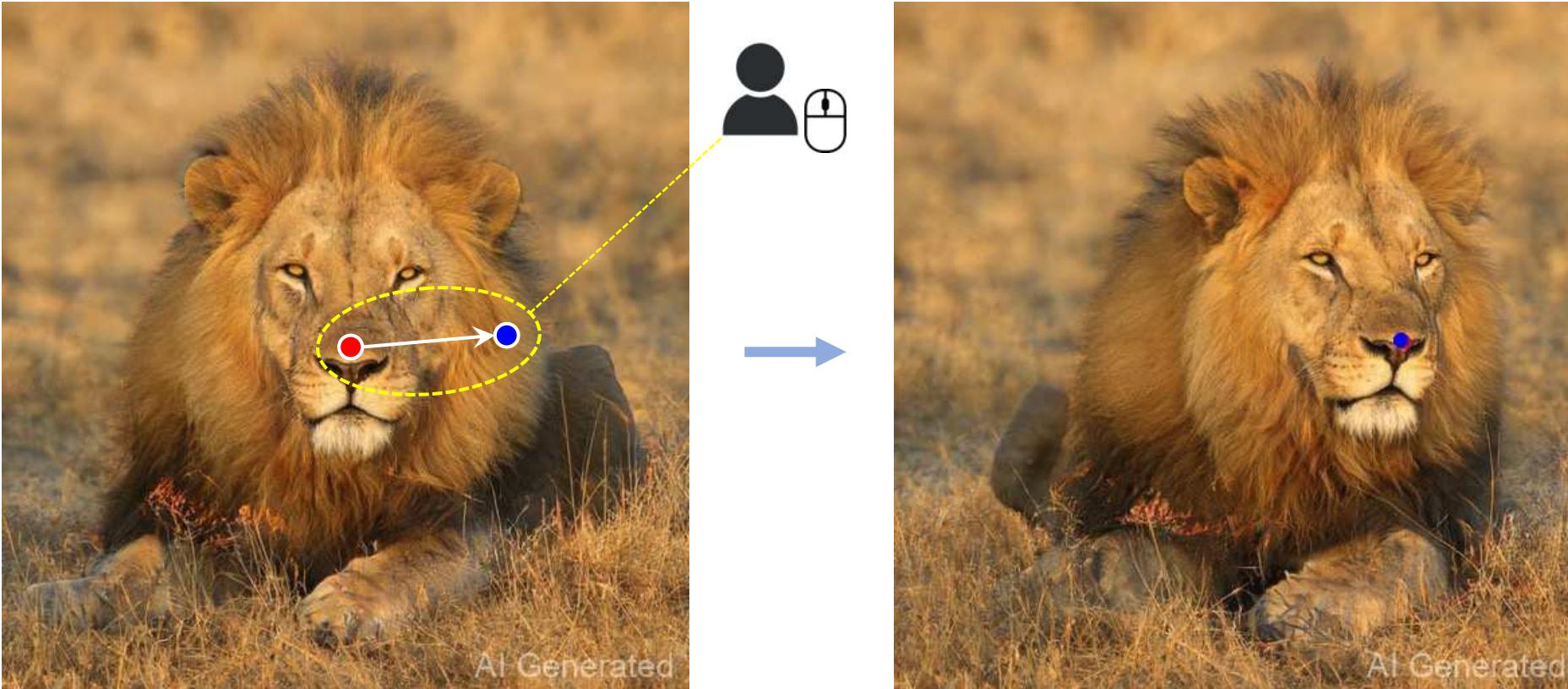
How Should We Control Spatial Attributes?



Limitations

- The language model need to understand all possible spatial editing commands
- It's hard for language model to understand the precise distance (30 pixels) in the image

Interactive Point Dragging



Advantages

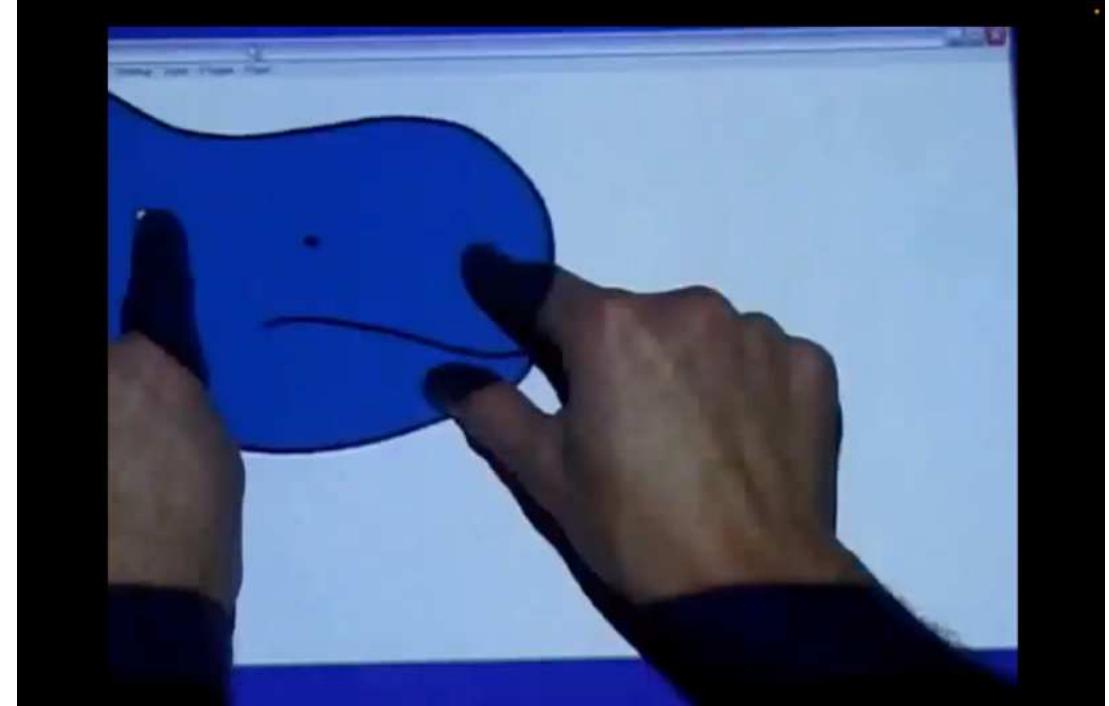
- Simple
- Precise
- Flexible
- Generic

Interactive Point Dragging



Conventional Shape Deformation

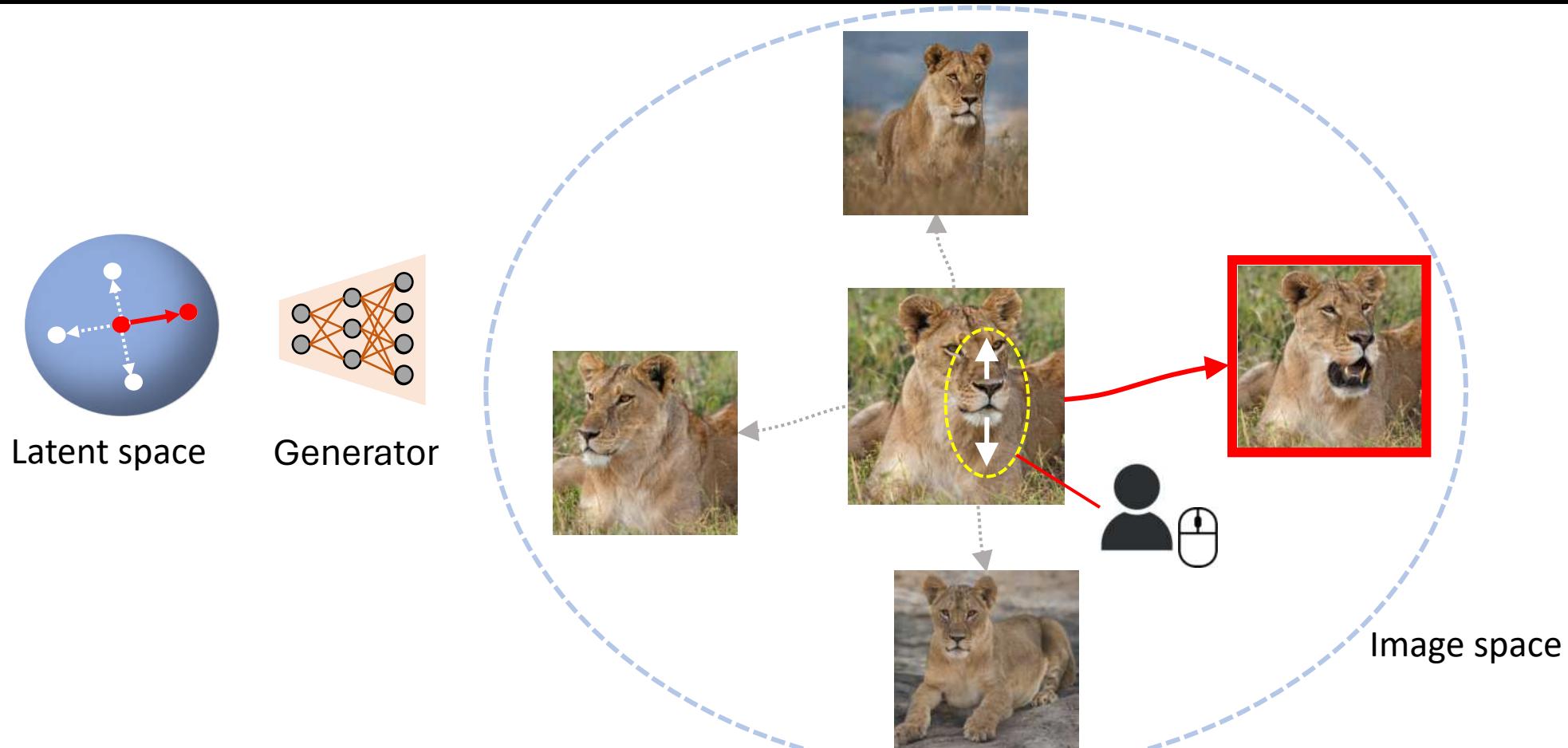
As-Rigit-As-Possible Shape Manipulation



Limitations:

- Assumes uniform rigidity
- Cannot hallucinate new content

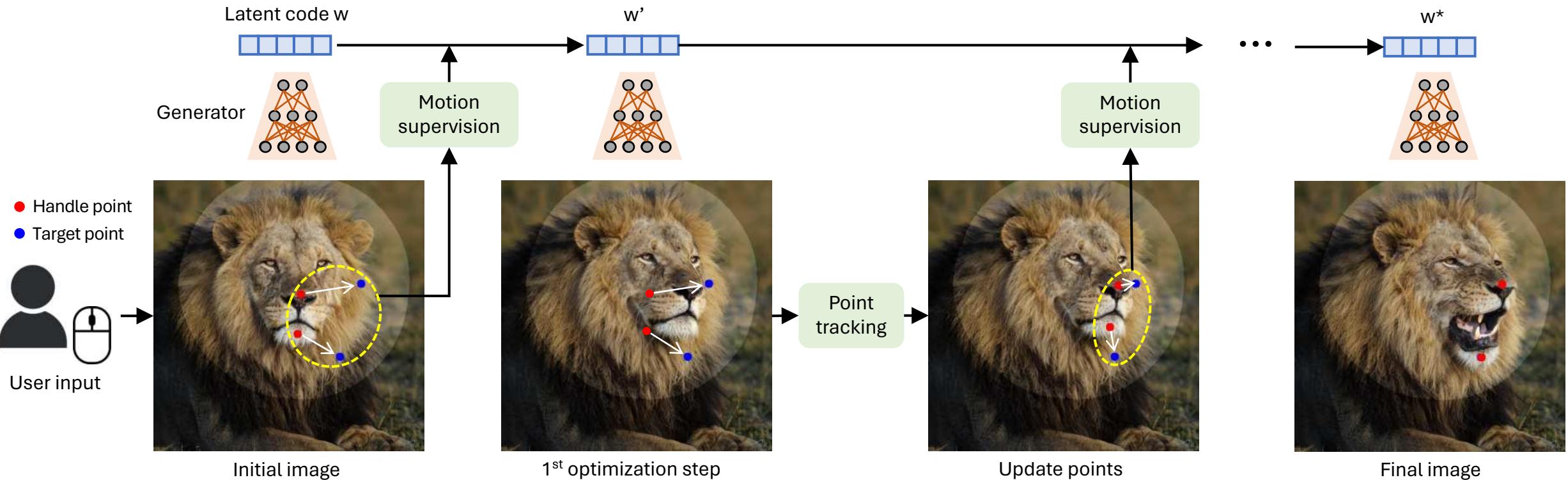
Motivation



GAN captures different variations of an object (pose, size, expression, etc)

We aims to traverse along the path that follows the dragging operation

Method Overview



Discriminative GAN Features

Few shot segmentation



RepurposeGAN [Rewatbowornwong et al, CVPR2021]

Few shot segmentation



DatasetGAN [Zhang et al, CVPR2021]

Unsupervised segmentation



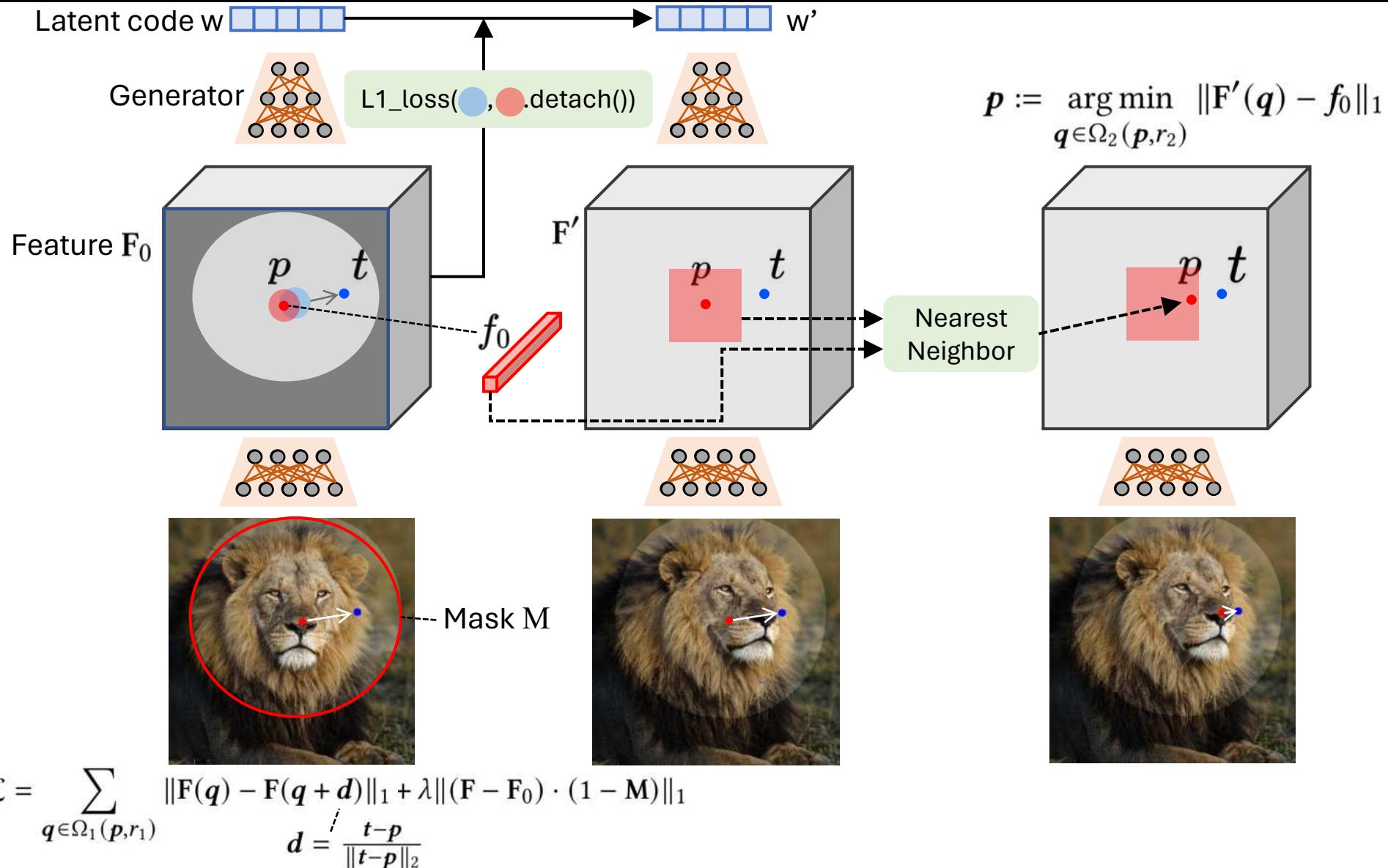
Labels4Free [Abdal et al, ICCV2021]

Dense Correspondence



Dual Deformation Field [Lan et al, arXiv, 2022]

Motion Sup. and Point Tracking in GAN Features

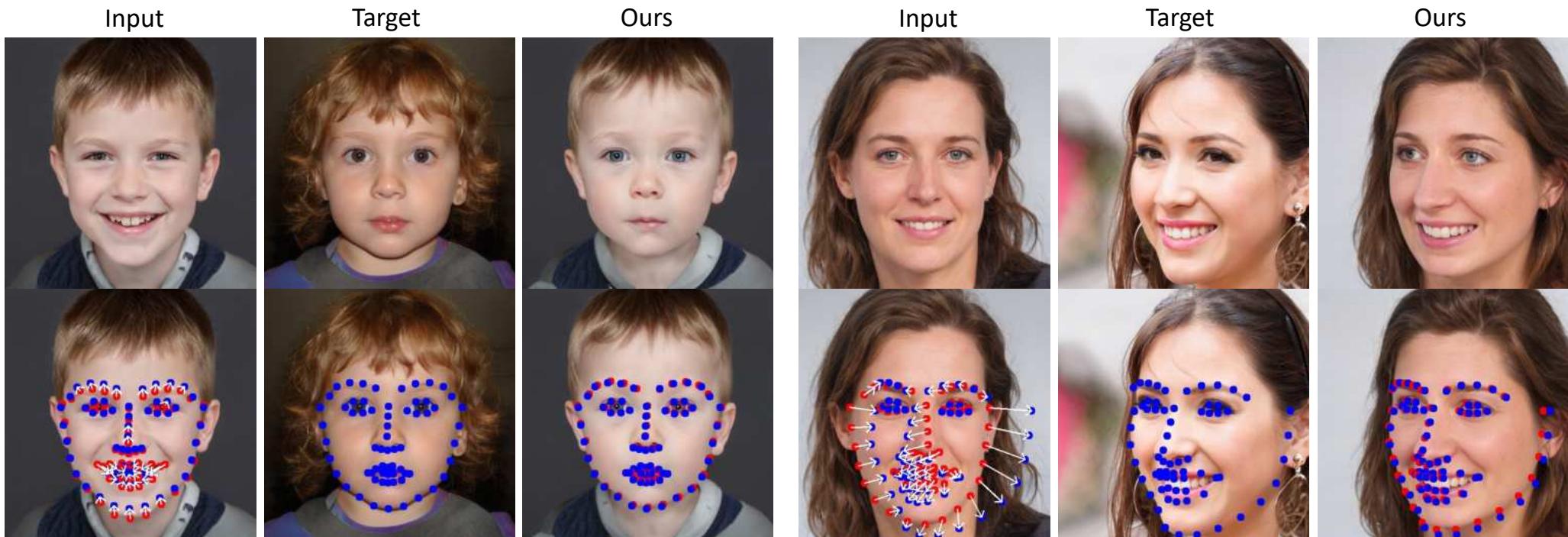




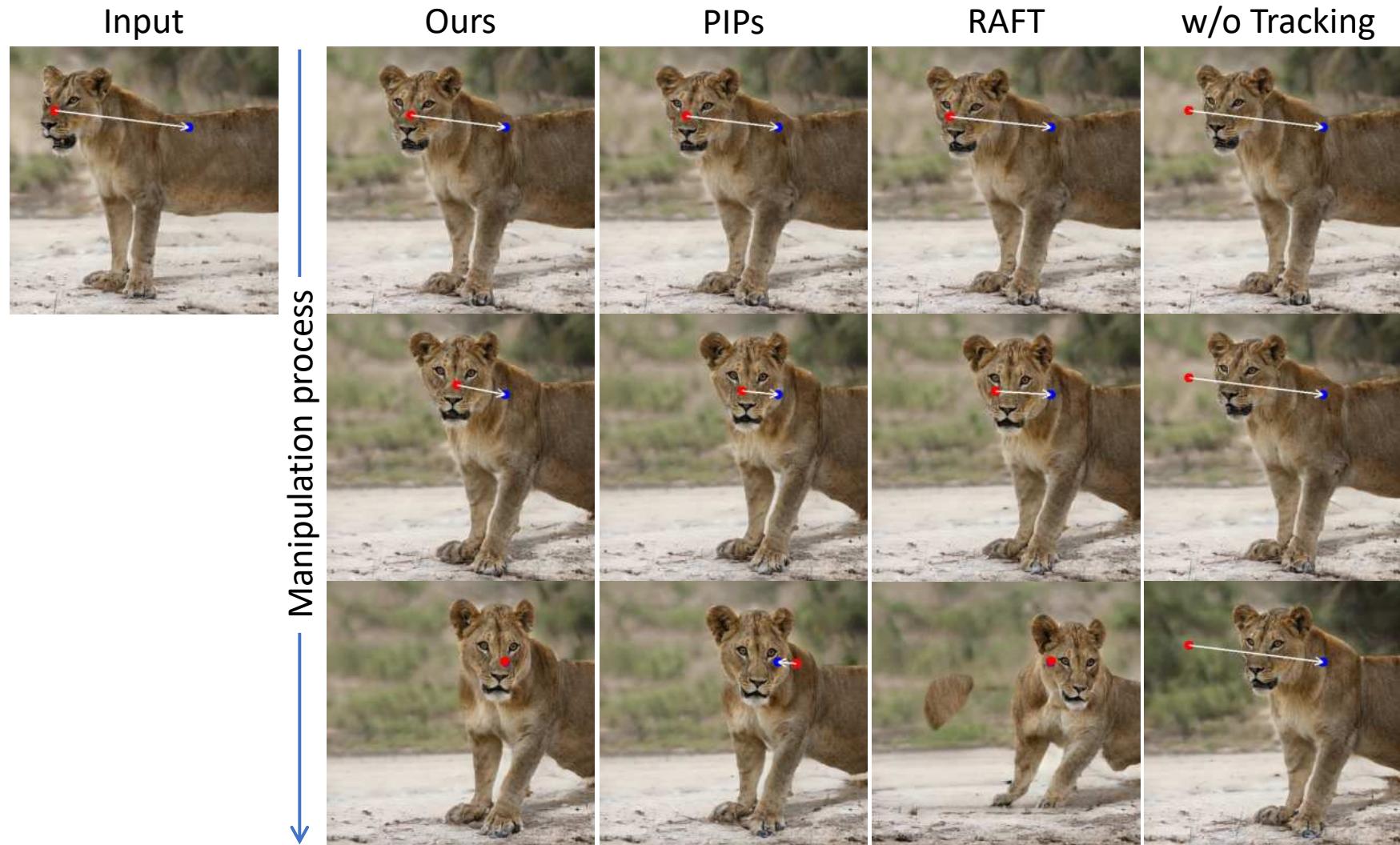
Qualitative Comparison



Face Landmark Manipulation

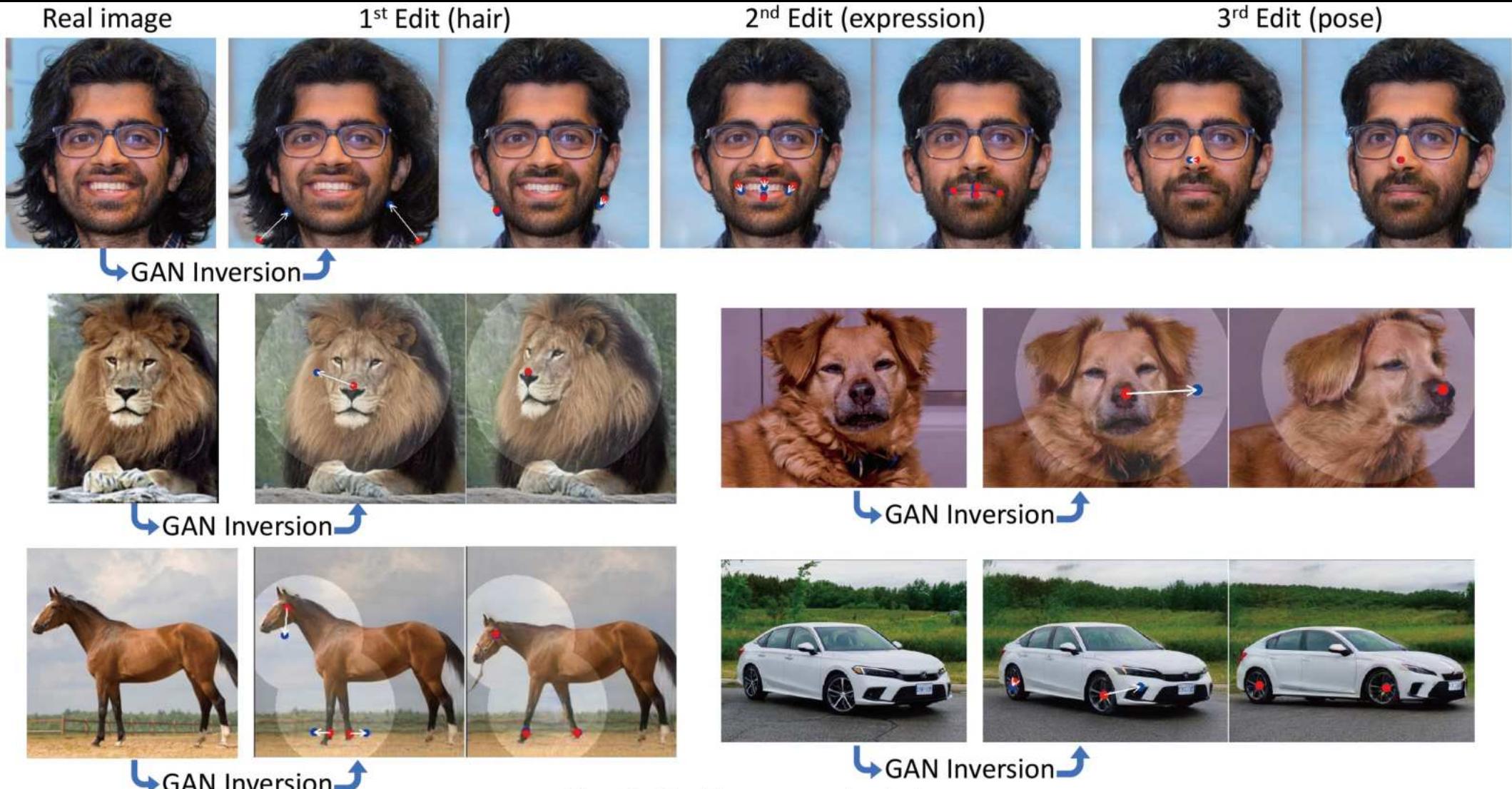


Point Tracking



Harley et al, Particle Video Revisited: Tracking Through Occlusions Using Point Trajectories, ECCV2022
Zachary et al, RAFT: Recurrent All-Pairs Field Transforms for Optical Flow, ECCV2020

Real Image Manipulation



Discussions

Effects of Mask

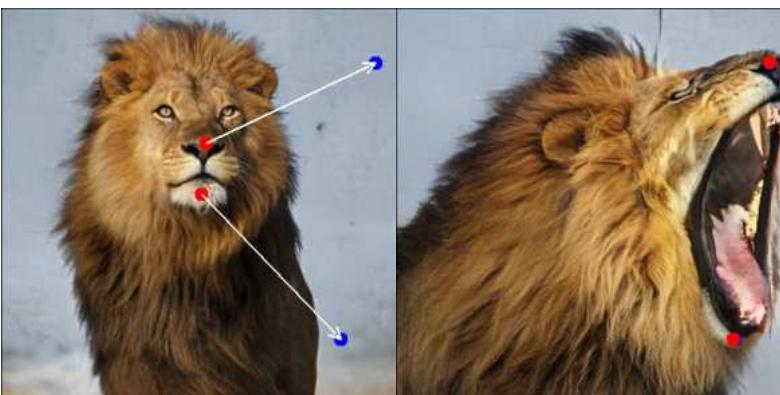
w/o mask



w/ mask



Out-of-distribution Manipulations



Thank you!