OBSERVATIONAL COSMOLOGY:

INTRODUCTION TO MACHINE LEARNING

Credit: A.Asperti (Unibo); F. Villaescusa (Simon Foundation)

A.A. 2025/2026 | Matteo Costanzi - University of Trieste / INAF

Why Machine Learning?

There are problems that are difficult to address with traditional programming techniques:

- classify a document according to some criteria (e.g. spam, sentiment analysis, ...)
- compute the probability that a credit card transaction is fraudulent
- recognize an object in some image (possibly from an unusual point of view, in new lighting conditions, in a cluttered scene)
- ...

Typically the result depends on a non-linear combination of a large number of parameters, each one contributing to the solution in a small degree

The Machine Learning approach:

Suppose to have a set of input-output pairs (training set):

$$\{\boldsymbol{x}, \boldsymbol{y}\}$$

the problem consists in understanding the map between **x** and **y**

The M.L. approach:

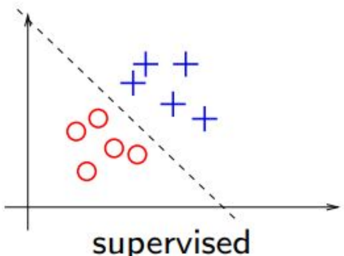
- describe the problem with a model depending on some parameters Θ (i.e. choose a parametric class of functions)
- define a loss function to compare the results of the model with the expected (experimental) values
- optimize (fit) the parameters Θ to reduce the loss to a minimum

The Machine Learning approach:

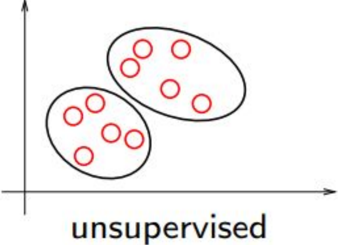
- Machine Learning problems are in fact optimization problems! So, why talking about learning?
- The point is that the solution to the optimization problem is not given in an analytical form (we don’t have a theoretical/analytical model to explain the data, and often there is no closed form solution).
- So, we use iterative techniques (typically, gradient descent) to progressively approximate the result.
- This form of iteration over data can be understood as a way of progressive learning of the objective function based on the experience of past observations.

Different types of learning tasks

- **supervised learning**:
  inputs + outputs (labels)
  - classification
  - regression



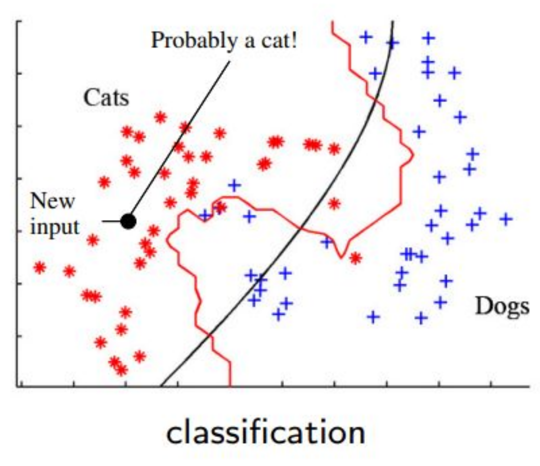
- **unsupervised learning**:
  just inputs
  - clustering
  - component analysis
  - autoencoding



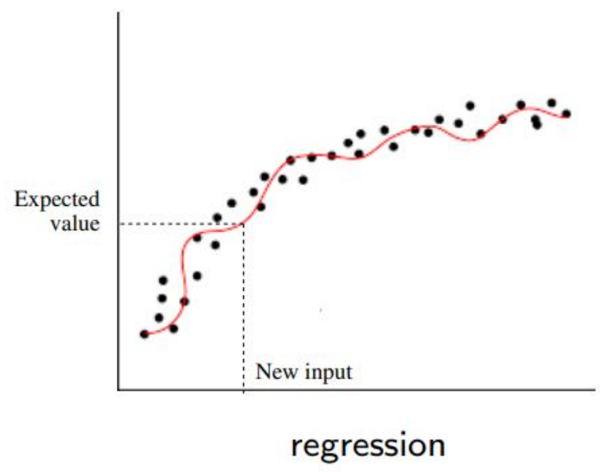
- **reinforcement learning**
  actions and rewards
  - learning long-term gains
  - planning

Classification vs. Regression

Two forms of supervised learning: $\{\langle x_i, y_i \rangle\}$



classification



regression

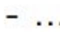
y is discete: $y \in \{\bullet, +\}$

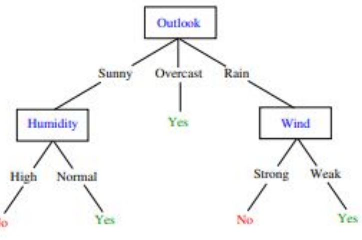
y is (conceptually) continuous

Many different techniques

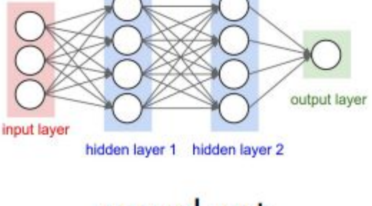
- **Different ways to define the models**:
  - decision trees
  - linear models
  - neural networks
  - ...

- **Different error (loss) functions**:
  - mean squared errors
  - logistic loss
  - cross entropy
  - cosine distance
  - maximum margin
  - ...

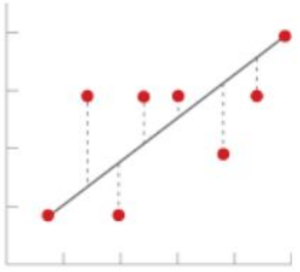




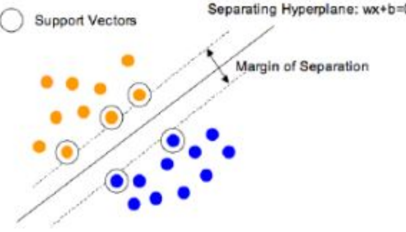
decision tree



neural net

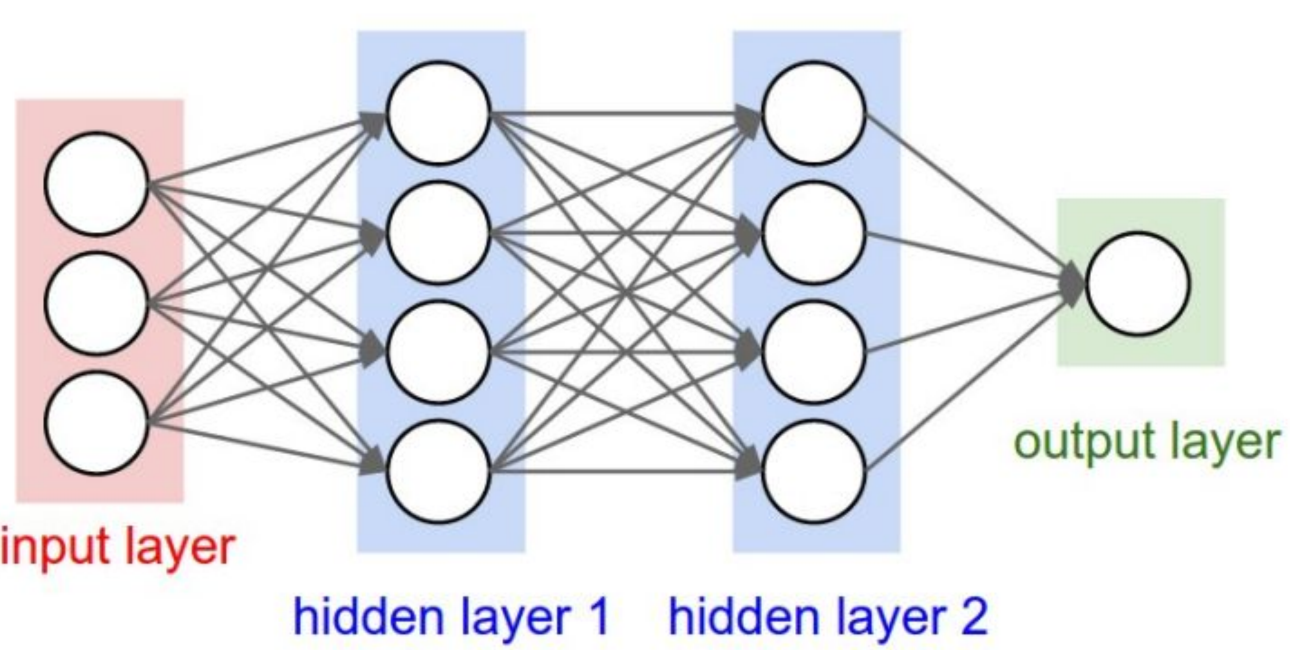


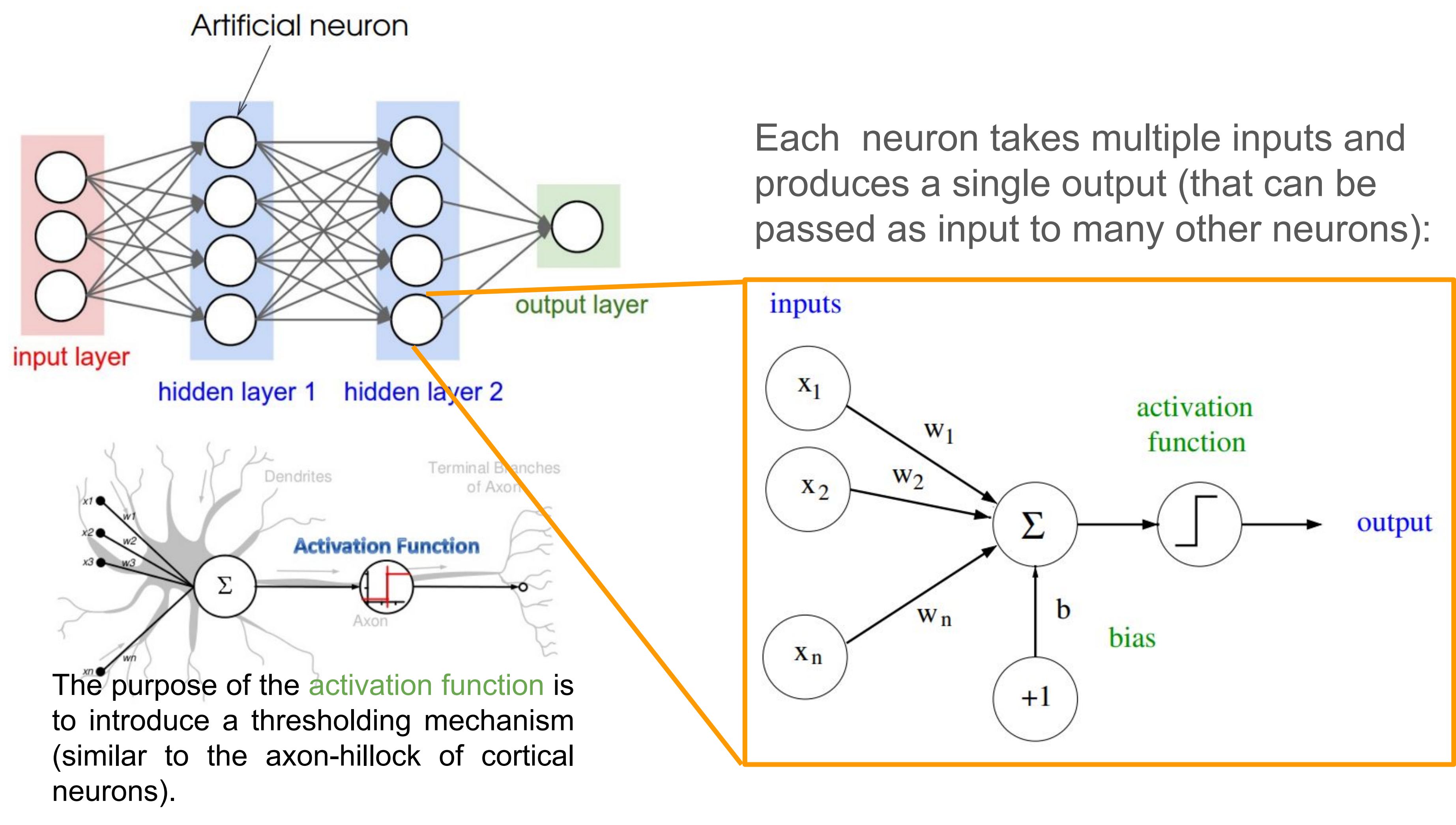
mean squared errors



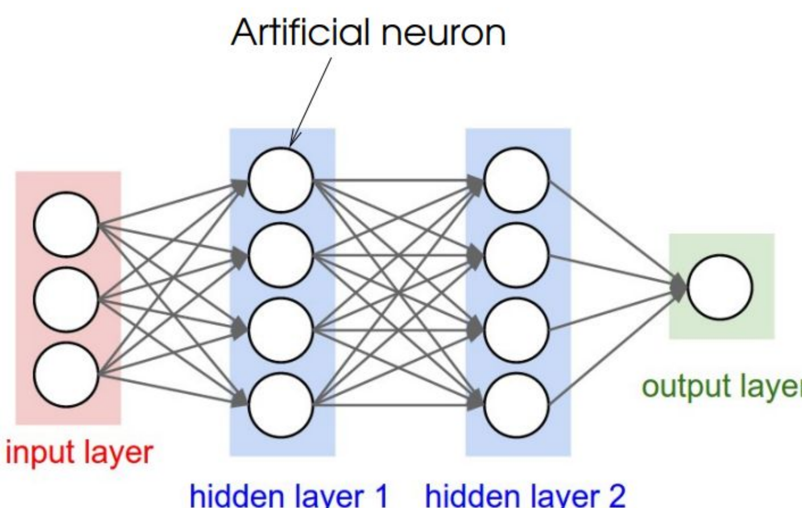
maximum margin

Neural Networks





Artificial neuron
input layer
hidden layer 1
hidden layer 2
output layer
Each neuron takes multiple inputs and produces a single output (that can be passed as input to many other neurons):
inputs
x1
x2
xn
w1
w2
wn
Σ
activation function
output
b
bias
+1
Dendrites
Terminal Branches of Axon
Activation Function
Axon
x1
x2
x3
xn
w1
w2
w3
wn
Σ
The purpose of the activation function is to introduce a thresholding mechanism (similar to the axon-hillock of cortical neurons).



The purpose of the activation function is to introduce a thresholding mechanism (similar to the axon-hillock of cortical neurons).

NOTE: Composing linear transformations does not increase the complexity of your model, since we still get a linear transformation!

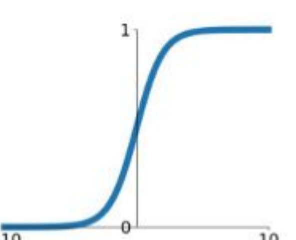
The activation function provides the source of NON LINEARTY in the neural networks

f(x)

Activation Functions

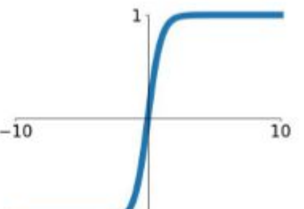
Sigmoid

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



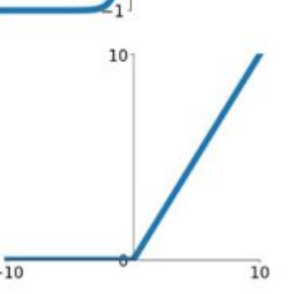
tanh

$\tanh(x)$

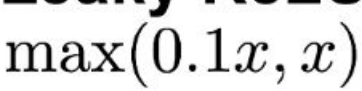


ReLU

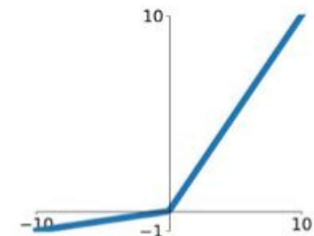
$\max(0, x)$



Leaky ReLU



$\max(0.1x, x)$

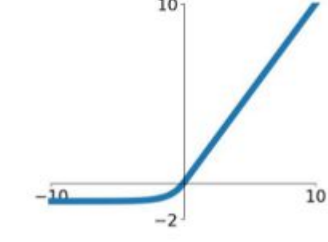


Maxout

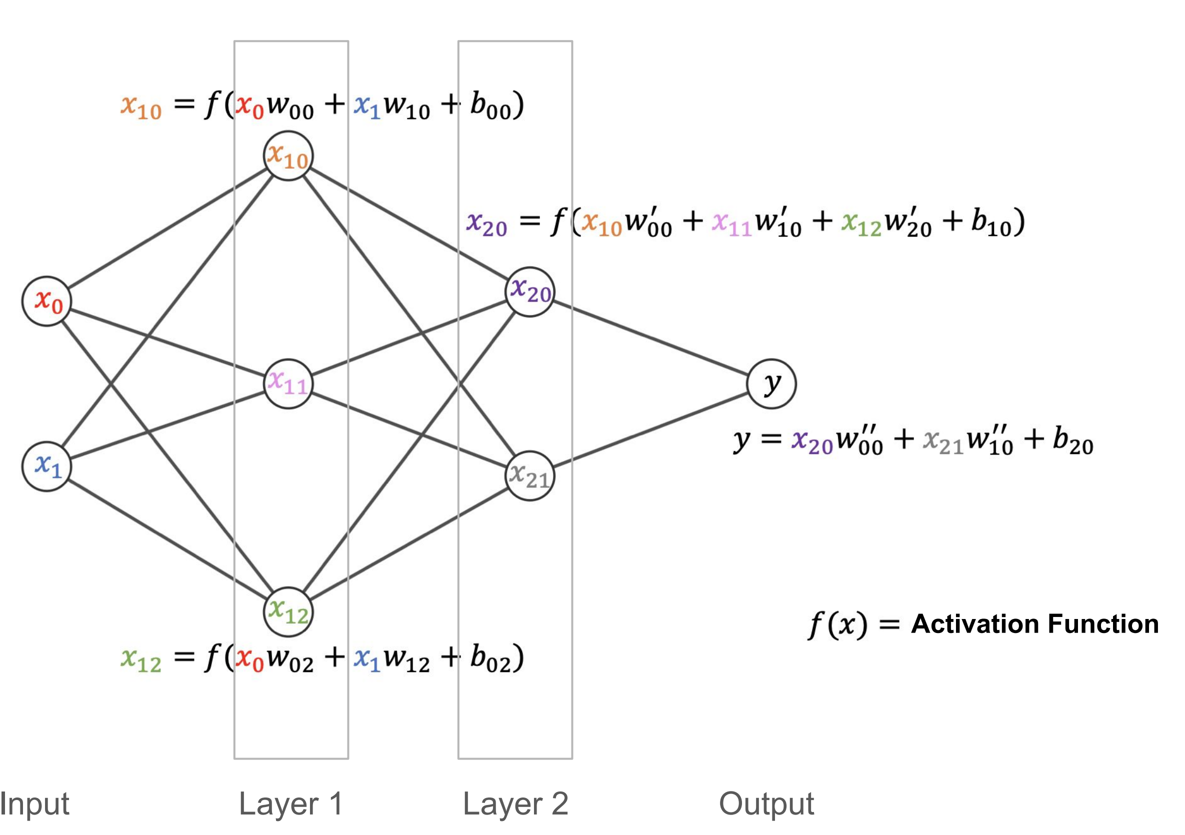
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Dense Feed-Forward NN

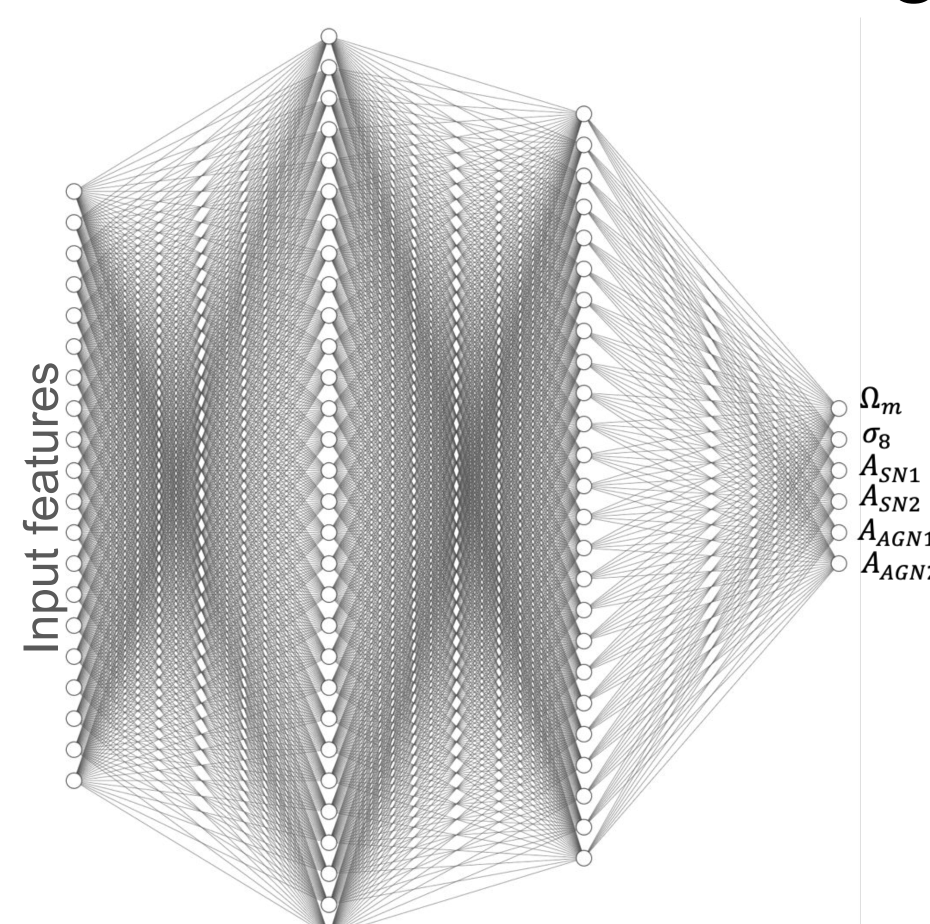


The most typical feed-forward network is a dense (i.e. w/ more than 1 hidden layer) network where each neuron at layer k − 1 is connected to each neuron at layer k.

The network is defined by a matrix of parameters (weights) w^k for each layer (+ biases). The matrix w^k has dimension $L_k \times L_{k+1}$ where L_k is the number of neurons at layer k.

The weights w^k and biases are the parameters of the model: they are learned during the training phase.

Training the NN



Goal: tune the value of the network parameters to get the most accurate predictions on the parameters.

Accuracy defined in the *loss function*

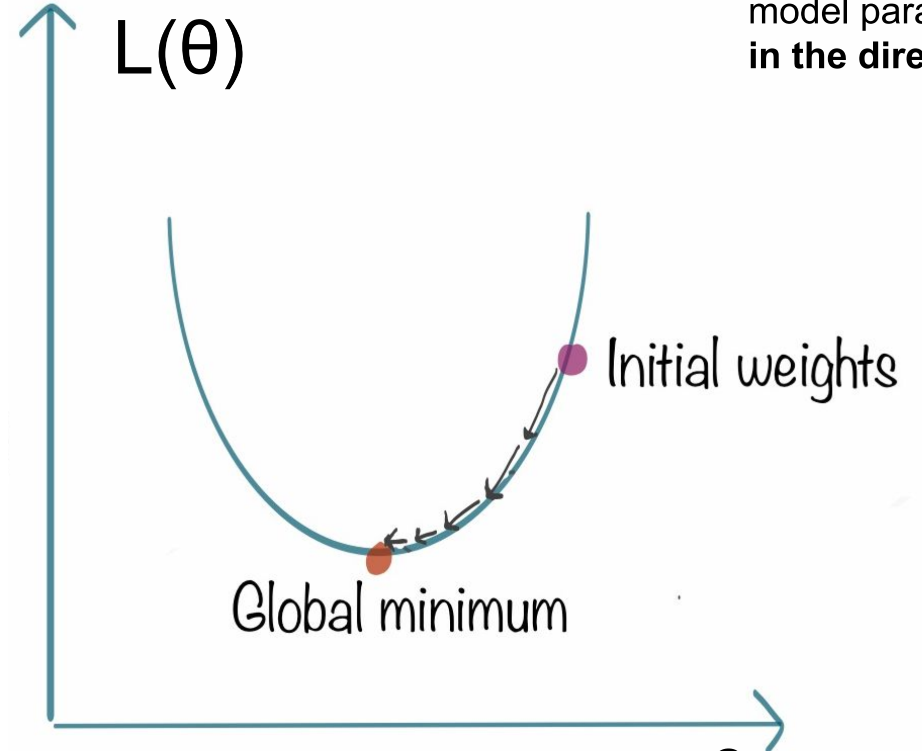
$$L = \frac{1}{N}\sum_{i=1}^{N}(y_{\mathrm{NN}}(\theta) - y_{\mathrm{true}})^2$$

In other words we want to “learn” the parameters which minimize the loss function (optimization problem!)

Gradient Descent

The objective is to minimize the loss function over (fixed) training samples by suitably adjusting the parameters ϑ_i .

To do so we compute the gradient of the loss function w.r.t. the model parameters ϑ_i , $\nabla_\vartheta L$. The **gradient is the vector pointing in the direction of steepest ascent**.

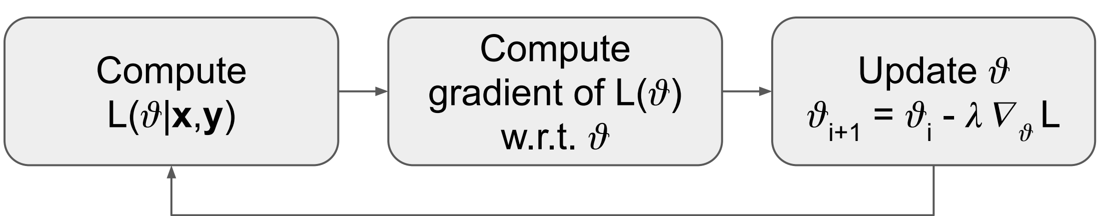


We can reach a minimal configuration for $L(\vartheta)$ by **iteratively taking small steps in the direction opposite to the gradient** (gradient descent).

$$\theta_{i+1} = \theta_i - \lambda \nabla_\theta L$$

learning rate parameter

Stochastic Gradient Descent



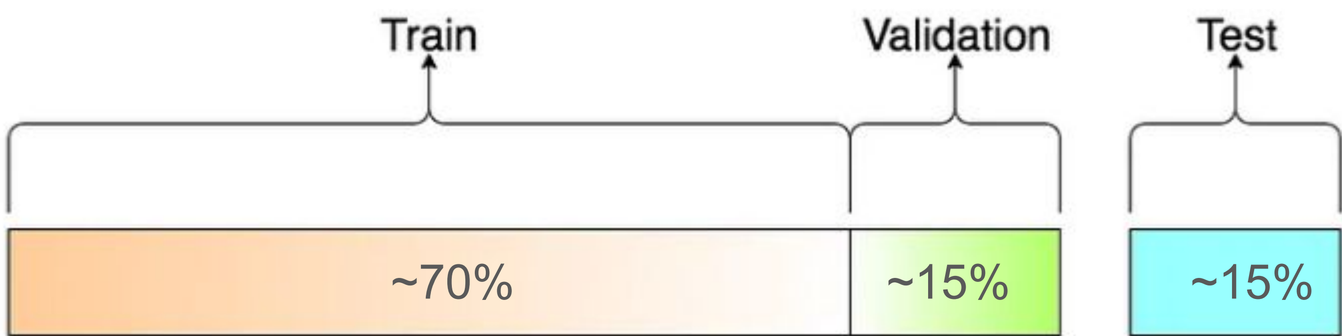
- Compute L and the derivative using all available data?
  Derivative will be smooth. Fast convergence but you may end up in a local minima

- Compute L and the derivative using a single data point?
  Derivative will be noisy. Will help escaping local minima, but hard to get convergence

- Compute L and the derivative using a random batch of point?
  Good trade between fast convergence and escape saddle points; also efficient for memory usage

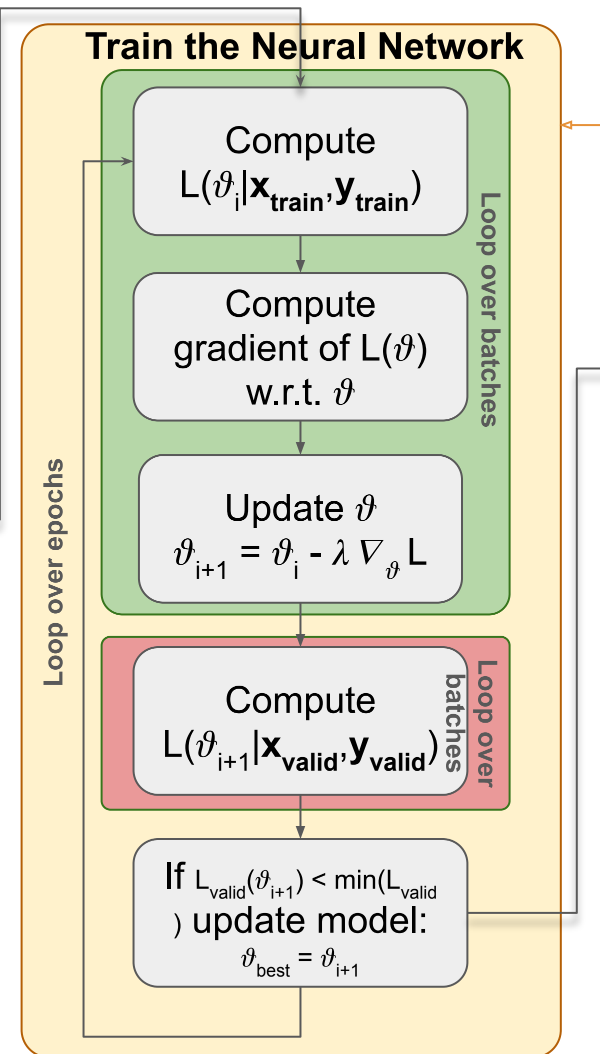
Training, validation and test data:

- **Training Dataset:** The actual dataset that we use to train the model (weights and biases in the case of a Neural Network). The model sees and learns from this data.
- **Validation Dataset:** The sample of data used to provide an unbiased evaluation of a model fit on the training dataset. The model see this data but doesn't learn from it.
- **Test Dataset:** The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset. The model doesn't see or learn from this data.



NN FLOW CHART:

Create data sets: Split your data ($\mathbf{x}$,$\mathbf{y}$) in
($\mathbf{x}_{train}$,$\mathbf{y}_{train}$)
($\mathbf{x}_{valid}$,$\mathbf{y}_{valid}$)
($\mathbf{x}_{test}$,$\mathbf{y}_{test}$)

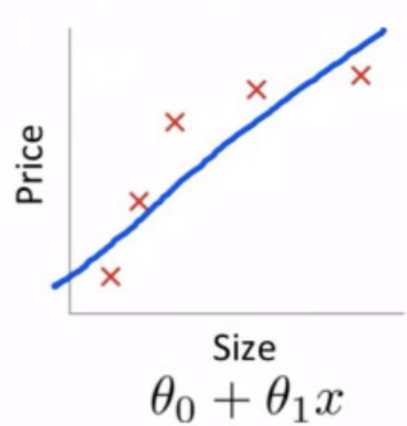


Neural Network components:

- Model: $\mathbf{y}(\mathbf{x},\boldsymbol{\vartheta})$
- Loss Function
- Optimizer

Test the network: Compute $y(\vartheta_{best}|\mathbf{x}_{test})$ and compare it with $\mathbf{y}_{test}$

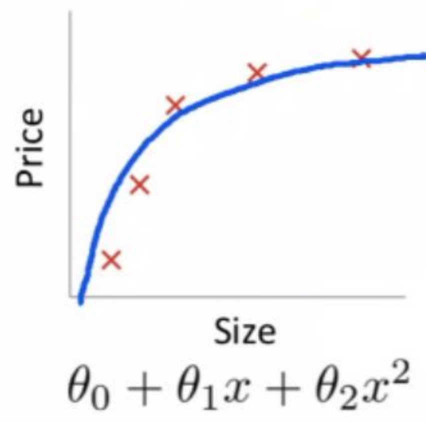
How many parameters?



$\theta_0 + \theta_1 x$

High bias
(underfit)

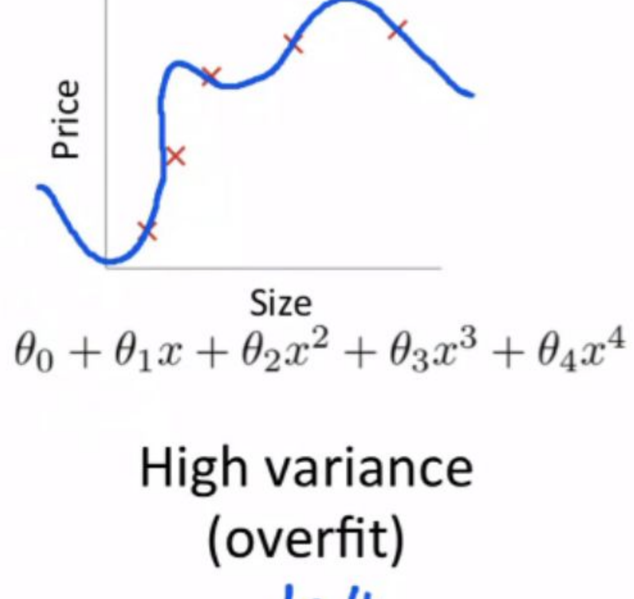
$d=1$



$\theta_0 + \theta_1 x + \theta_2 x^2$

"Just right"

$d=2$



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

High variance
(overfit)

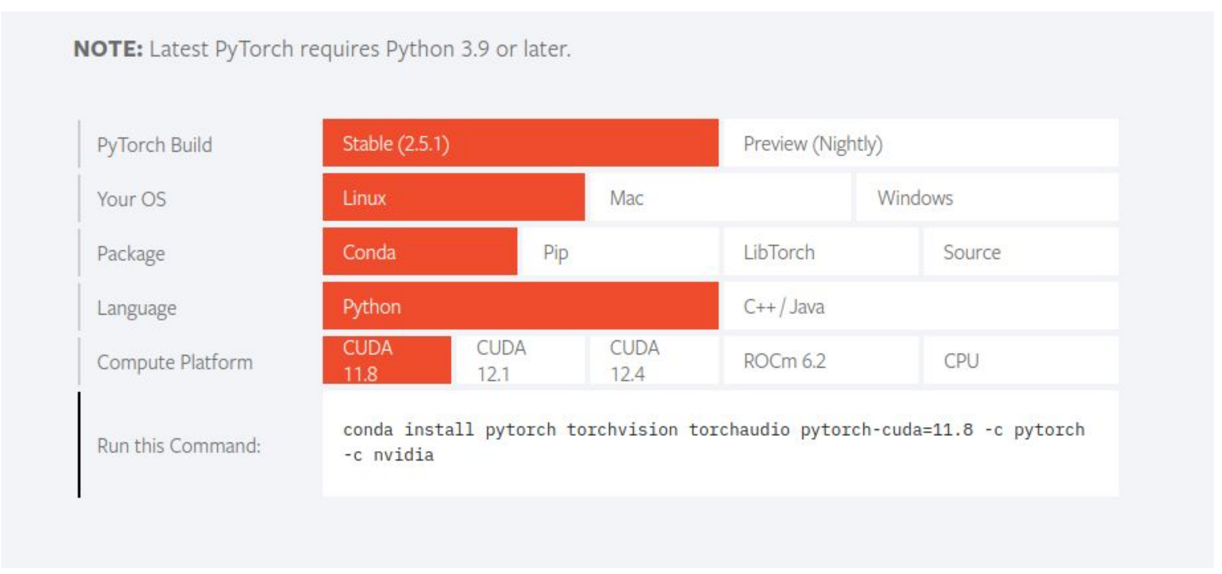
$d=4$

Regularization

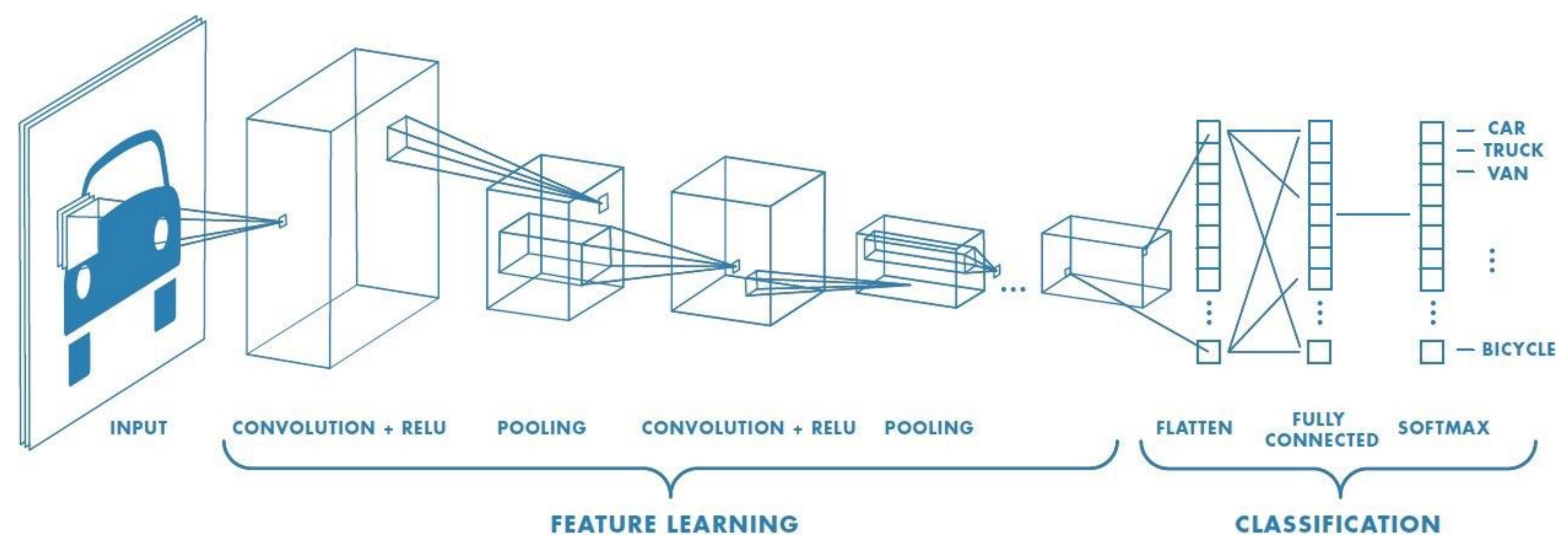
Weight decay	Dropout
$$L = \frac{1}{N}\sum_{i=1}^{N}(\theta_{NN} - \theta_{True})^2 + \eta\sum w_i^2$$	

Pytorch library

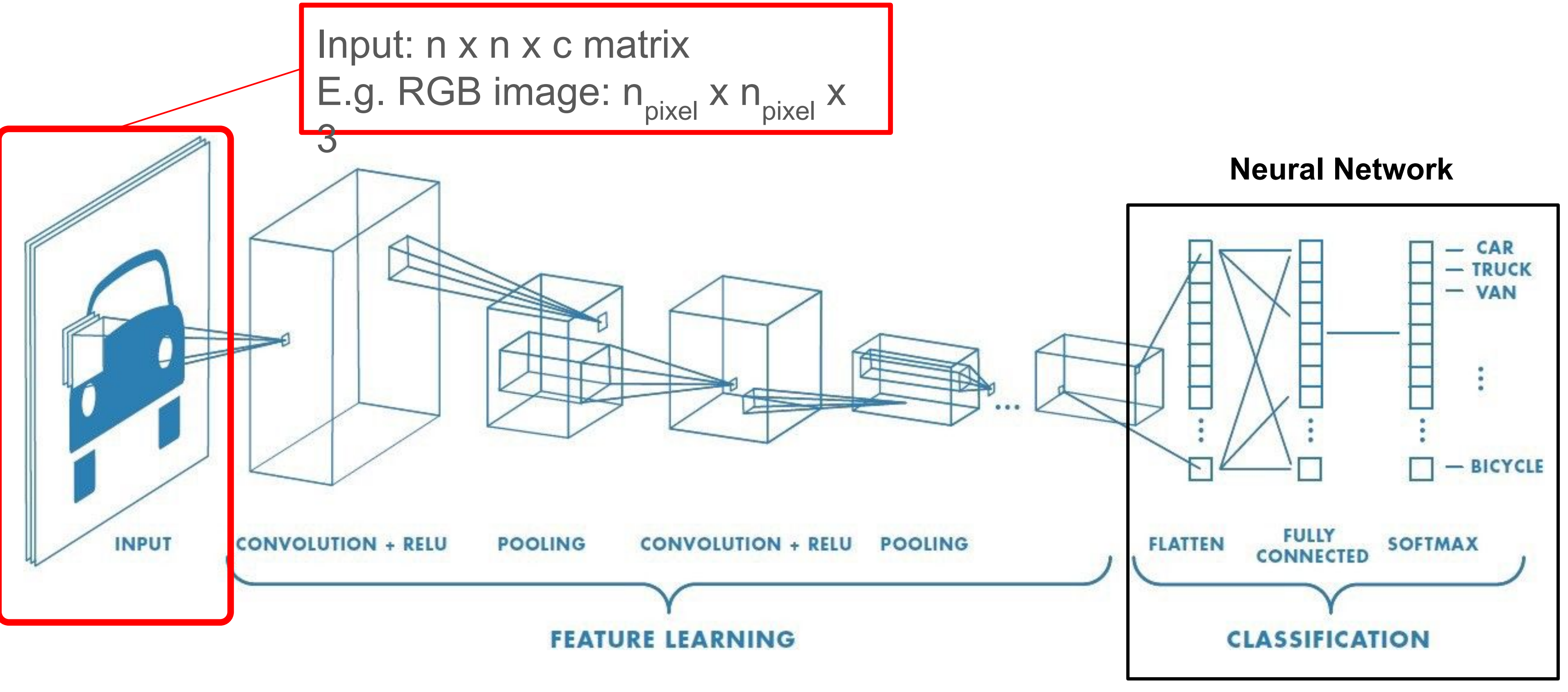
To install follow the instruction here



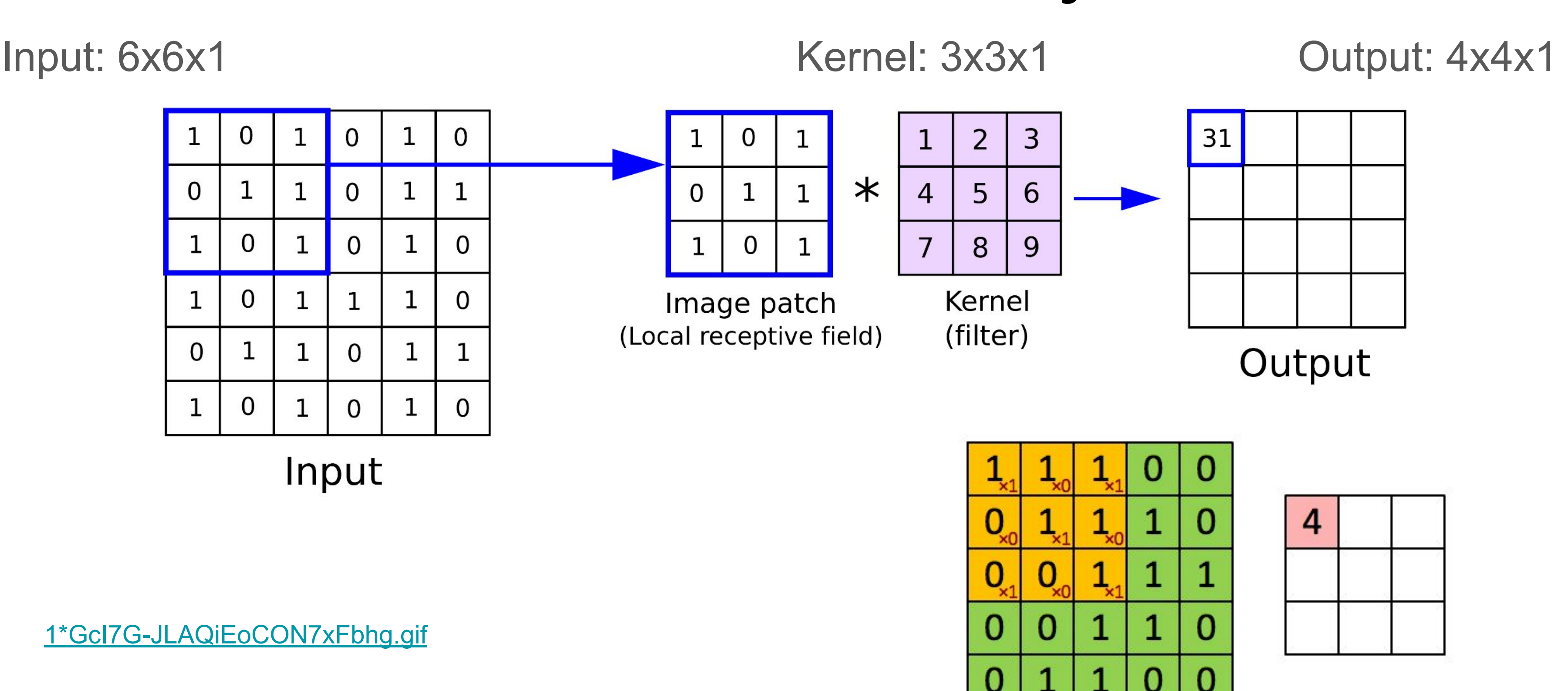
Convolutional Neural Networks (CNN)



Convolutional Neural Networks (CNN)



Convolution Layer



1*GcI7G-JLAQiEoCON7xFbhg.gif

Image

Convolved
Feature

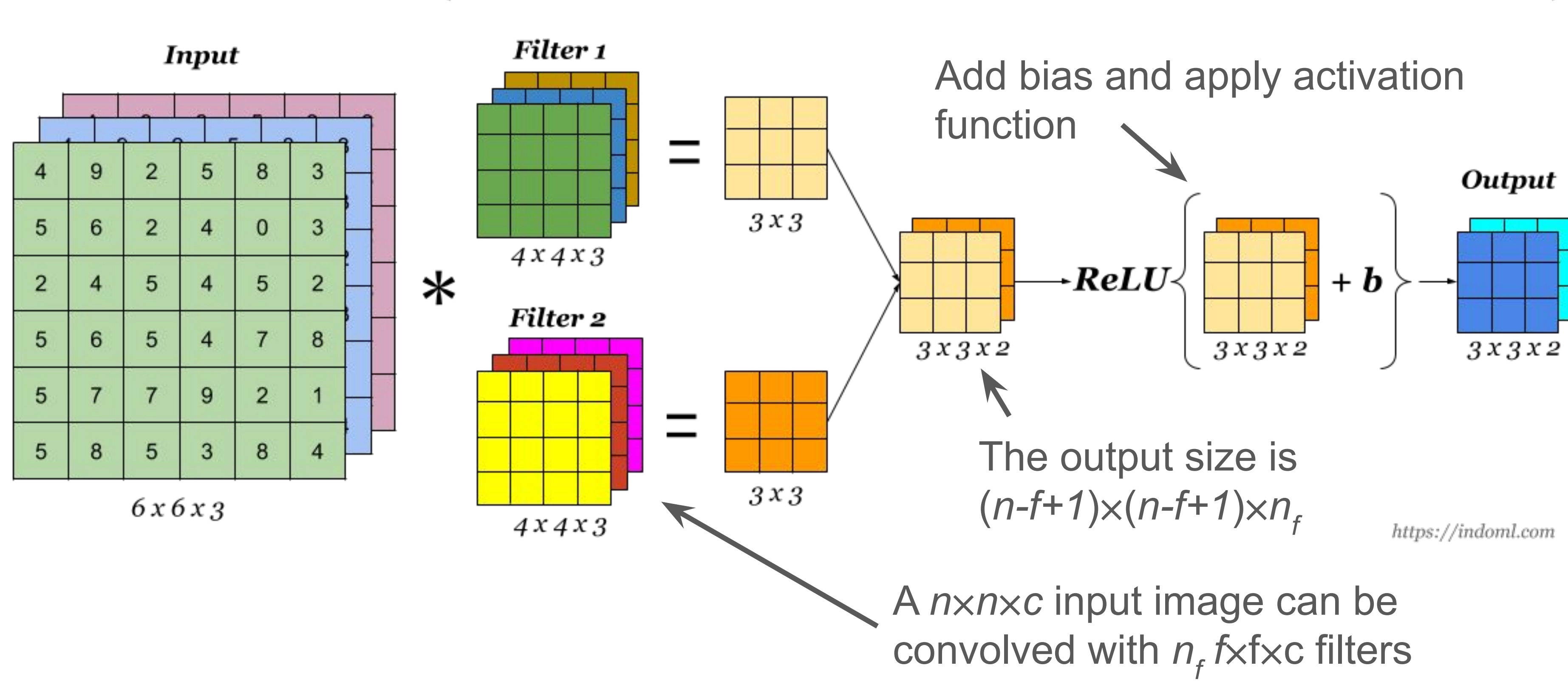
Convolution Layer

Feature extraction by filtering the image:

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Convolution Layer: 2



Padding

0	0	0	0	0	0	0	0
0	3	3	4	4	7	0	0
0	9	7	6	5	8	2	0
0	6	5	5	6	9	2	0
0	7	1	3	2	7	8	0
0	0	3	7	1	8	3	0
0	4	0	4	3	2	2	0
0	0	0	0	0	0	0	0

$6 \times 6 \rightarrow 8 \times 8$

*

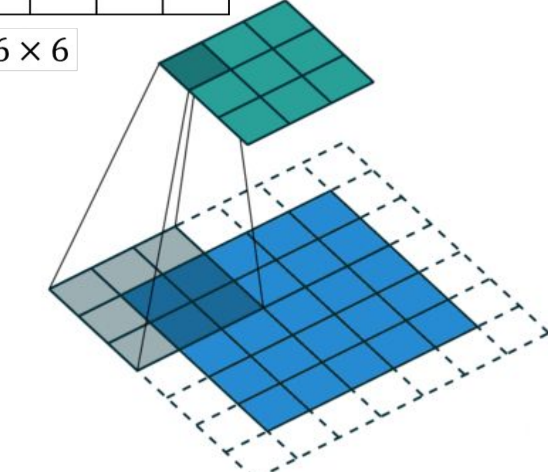
1	0	-1
1	0	-1
1	0	-1

3×3

=

-10	-13	1			
-9	3	0			

6×6

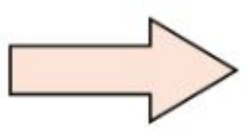


1*1VJDP6qDY9-ExTuQVEOlVg.gif

Strides

1	2	3	4	5	6	7
11	12	13	14	15	16	17
21	22	23	24	25	26	27
31	32	33	34	35	36	37
41	42	43	44	45	46	47
51	52	53	54	55	56	57
61	62	63	64	65	66	67
71	72	73	74	75	76	77

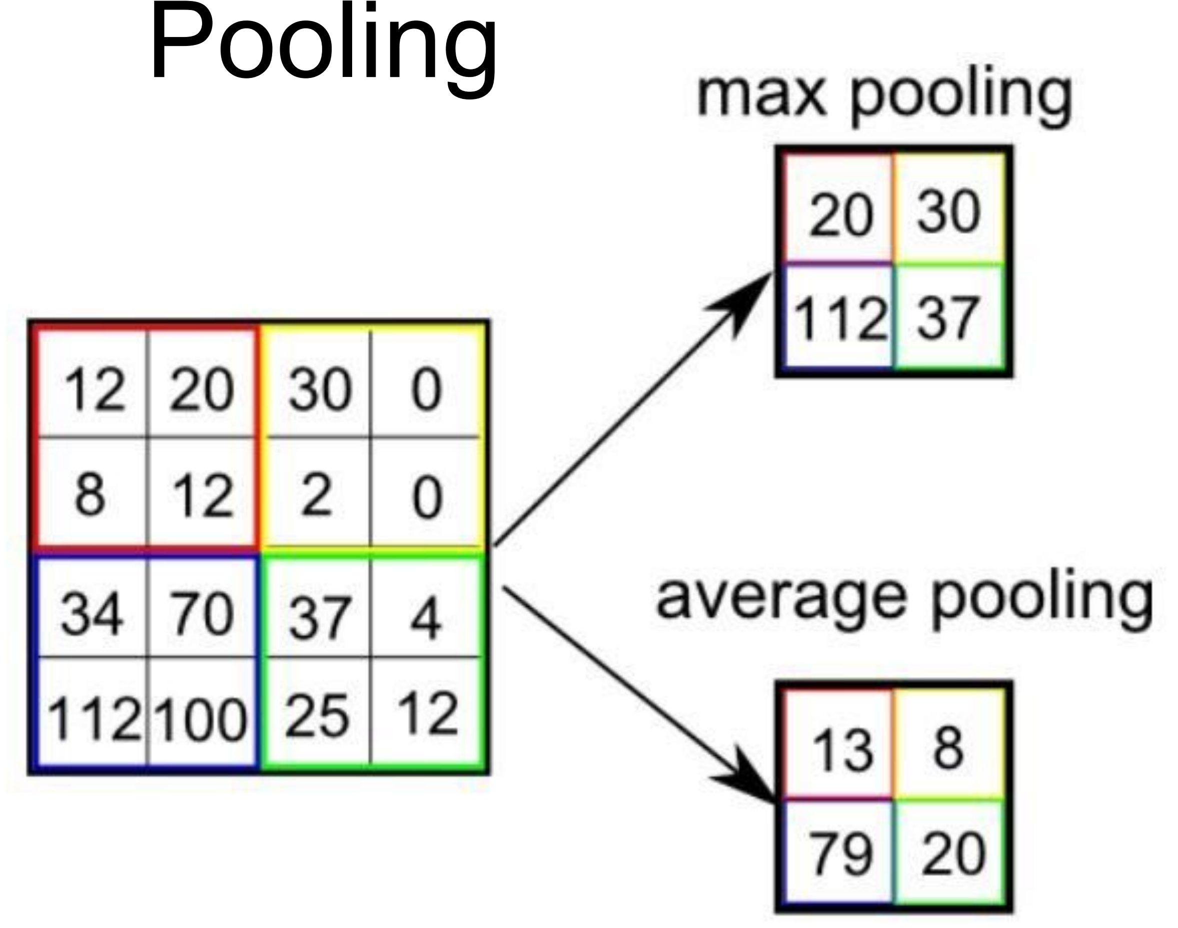
Convolve with 3x3
filters filled with ones



108	126	
288	306	

$$S_{\text{out}} = \frac{S_{\text{in}} + 2\text{Padding} \; - \; \text{Kernel_size} \; - 2}{\text{Stride}} + 1$$

Pooling



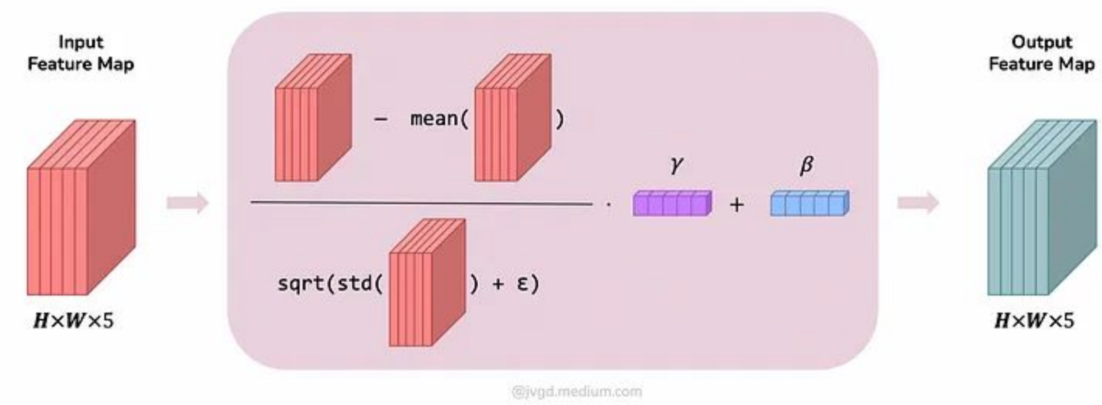
1*uoWYsCV5vBU8SHFPAPao-w.gif

BatchNorm

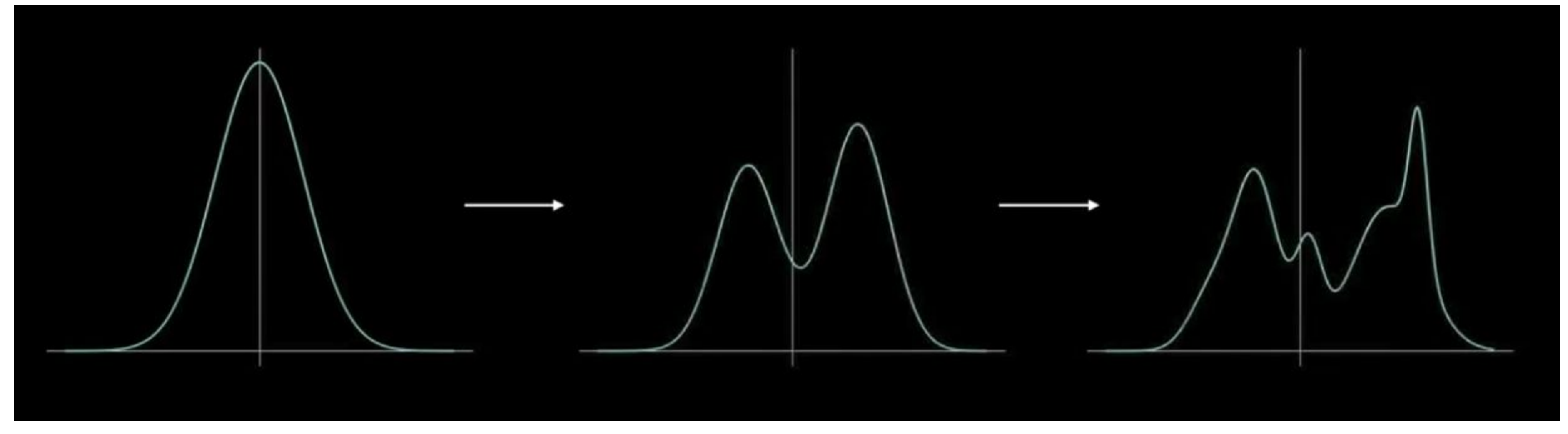
Purpose:

- **Normalize Inputs**: BatchNorm normalizes the inputs to each layer to have zero mean and unit variance within a mini-batch.
- **Mitigate Internal Covariate Shift**: It reduces the changes in the distribution of layer inputs during training, allowing the network to learn more efficiently.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$



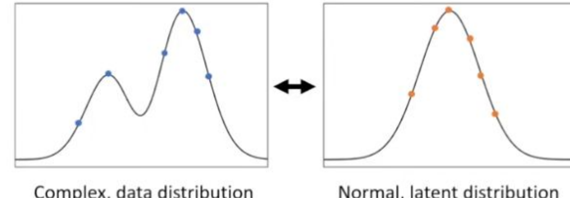
Normalizing Flow (NF)



Normalizing Flow (NF)

Normalizing Flows are a family of methods for constructing flexible learnable probability distributions, often with neural networks, which allow us to surpass the limitations of simple parametric forms.

Key ideas:

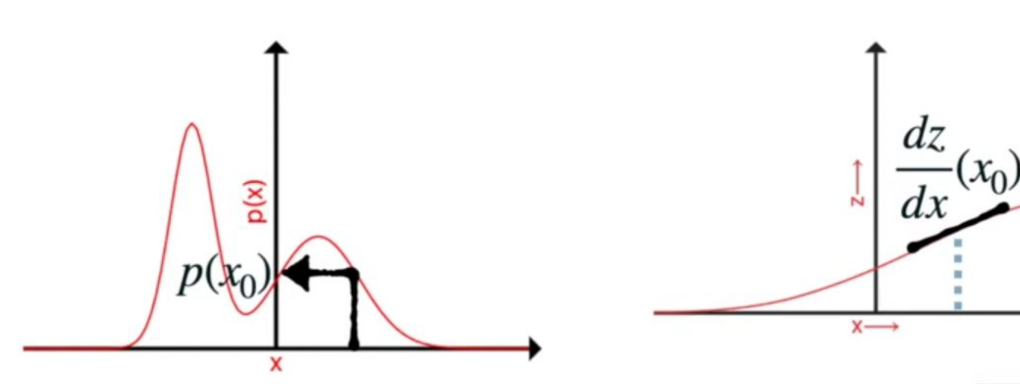


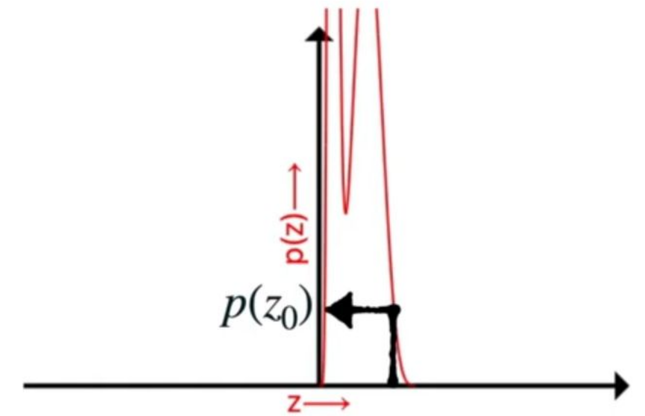
- Learn a **bijective mapping** function f: $R^D \rightarrow R^D$
- Use **change of variables** formula to compute densities $p(x)\,dx = p(z)\,dz \rightarrow p(z) = p(x)\,|dx/dz|$

Applications:

- **Density Estimation:** Exact likelihood modeling
- **Bayesian Inference:** Posterior sampling in simulation-based inference
- **Image/Audio Generation:** Realistic, high-resolution samples
- **Anomaly Detection**

Change of variable formula

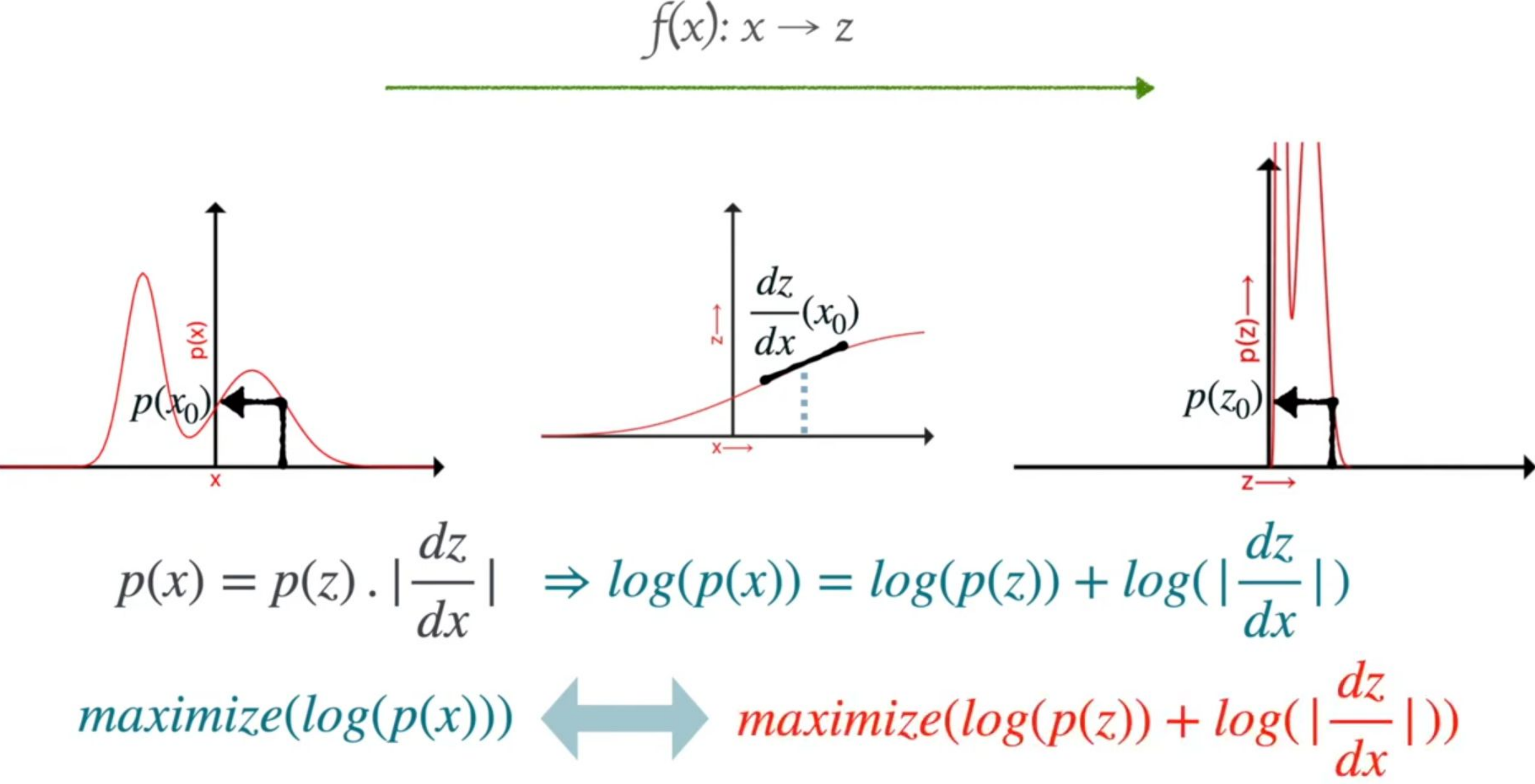




Let x = f(z), and z~p(z):

$$p(x) = p(z) \cdot \left|\frac{dz}{dx}\right|$$

Probability density of x = Probability density of z * Amount by which space around x is stretched or shrunk



This enables **exact likelihood computation** and **efficient training** via maximum likelihood

Flow

A flow is a sequence of Invertible Transformation:

$$x = f_K \circ \cdots \circ f_1(z)$$

From the chain rule it follows that:

$$\log p_X(x) = \log p_Z(z) - \sum_{k=1}^{K} \log \left| \det \left(\frac{\partial f_k}{\partial h_{k-1}} \right) \right|$$

Exact likelihood evaluation!
Exact posterior inference (via $z = f^{-1}(x)$)

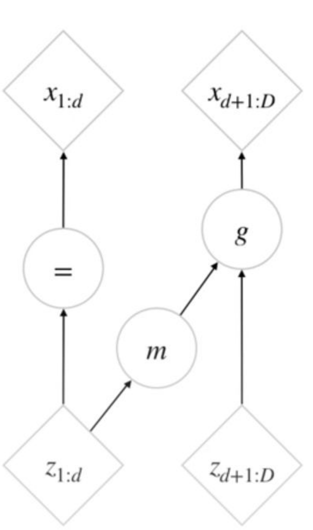
The goal is to define f_k which are invertible and has a tractable Jacobian determinant

Coupling Layer

Basic Idea:

- Split the input array **z**, in two part $\mathbf{z}_1$ and $\mathbf{z}_2$
- Transform only one part of the array so that the Jacobian matrix is triangular! → Invertible and simple to compute the determinant:

$$\left|\det\frac{dx}{dz}\right| = \Pi_i \frac{dx_i}{dz_i}$$

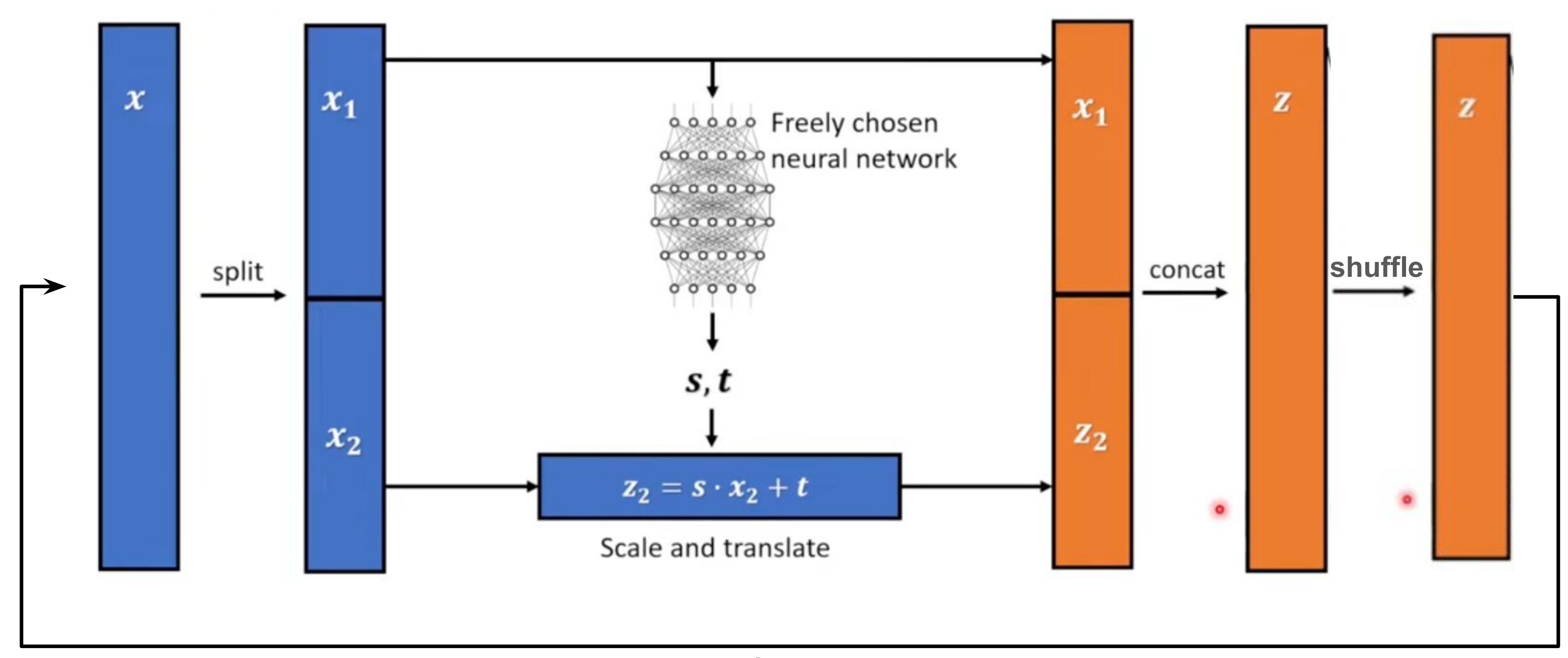


$$x_{1:d} = z_{1:d}$$

$$x_{d+1:D} = g(z_{d+1:D}; m(z_{1:d}))$$

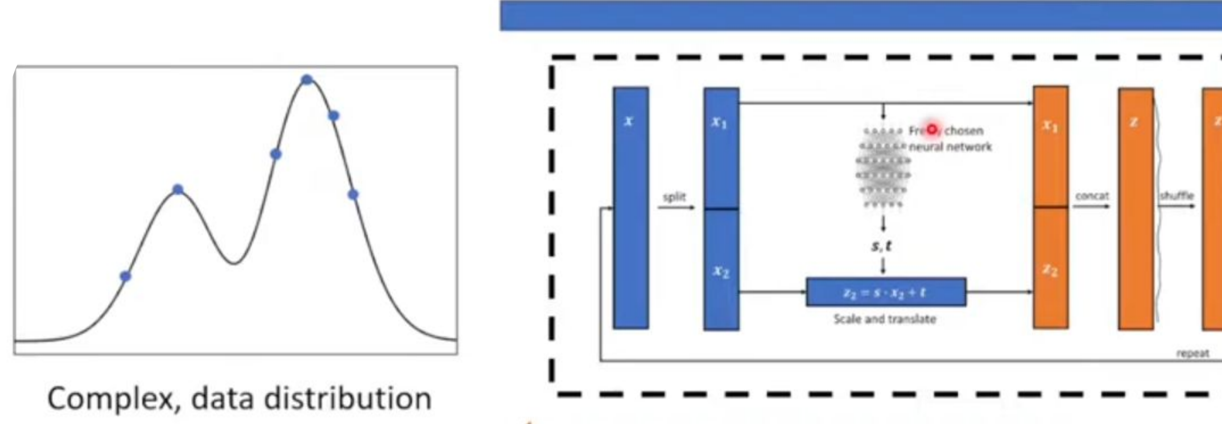
$$\frac{\partial x}{\partial z} = \begin{bmatrix} I_d & 0 \\ \frac{\partial x_{d+1:D}}{\partial z_{1:d}} & \frac{\partial x_{d+1:D}}{\partial z_{d+1:D}} \end{bmatrix}$$

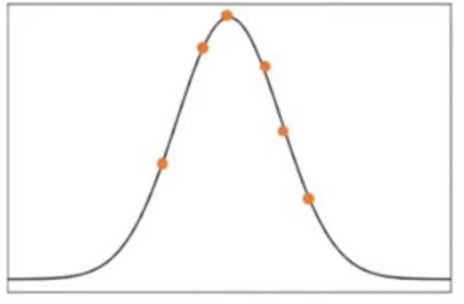
Affine Coupling Layer



Affine Coupling Layer

Flow Direction $f_\theta(x)$





Normal, latent distribution

Generative Direction $g_\theta(z) = f_\theta^{-1}(z)$

Masked Autoregressive Flow

Autoregressive Transformation:

- Each output dimension depends only on the previous input:

$$x_i = \mu_i(z_{<i}) + \sigma_i(z_{<i}) \cdot z_i$$

  where μ_i and σ_i are outputs of a neural network
- The Jacobian of the transformation is triangular and the likelihood can be easily computed as:

$$\log p_X(x) = \sum_i \left[\log p_Z(z_i) - \log|\sigma_i|\right]$$

```
Input Data (x)
|
├── [Masked Autoregressive Network]
|     |
|     ├── Layer 1: Mask M1 → Output (μ1, σ1)
|     |              |
|     |              └── Transform: x1' = (x1 - μ1) / σ1
|     |
|     ├── Layer 2: Mask M2 → Output (μ2, σ2)
|     |              |
|     |              └── Transform: x2' = (x2 - μ2) / σ2
|     |
|     └── ... (repeat for all dimensions)
|
└── Output: Latent Variable (z) ~ 𝒩(0, I)
```

Ensures the autoregressive property by masking connections

Each layer outputs location (μ) and scale (σ) parameters for the affine transformation

NF Architectures

Model	Transformation Type	Key Feature
RealNVP	Affine Coupling	Fast inverse, tractable Jacobian
MAF (Masked Autoregressive Flow)	Autoregressive	Easy to train, slow to sample
IAF (Inverse Autoregressive Flow)	Autoregressive	Fast sampling, slow likelihood
Glow	Invertible 1x1 conv + ActNorm	Scalable to images
Neural Spline Flows	Monotonic splines	More flexible transforms