

Machine Learning for Physics

Supervised Learning

Lisbon School on Machine Learning for Physics 2025, LIP Lisboa, Portugal

Pietro Vischia
pietro.vischia@cern.ch
[@pietrovischia](https://twitter.com/pietrovischia)



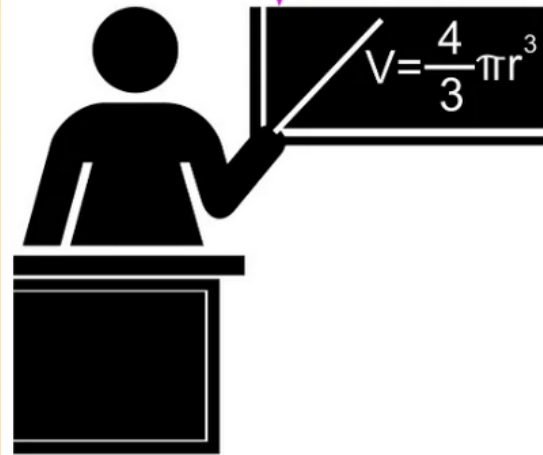
If you are reading this as a web page: have fun! If you are reading this as a PDF: please visit

https://www.hep.uniovi.es/vischia/persistent/2025-03-12_LisbonMLSchoolPhysics_SupervisedLearning.html

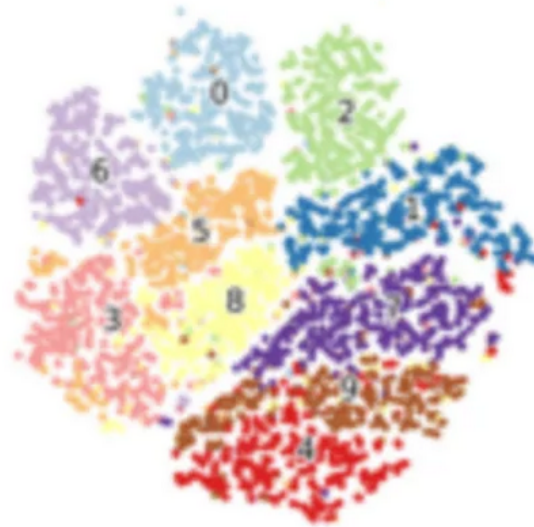
to get the version with working animations

Learn in different ways

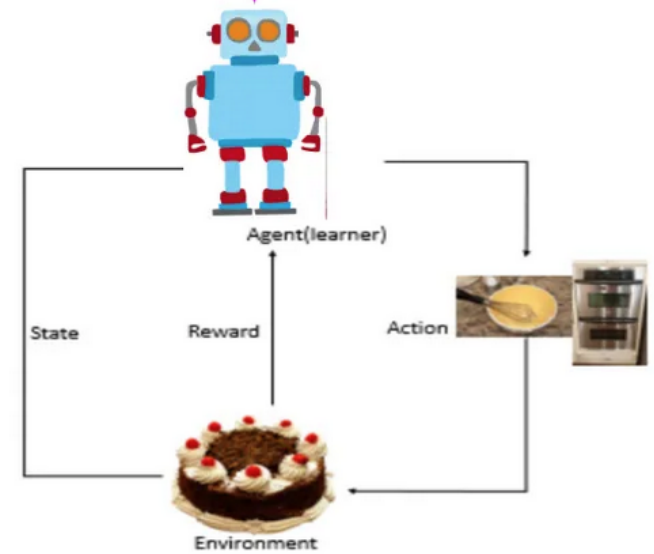
Machine Learning



Supervised Learning



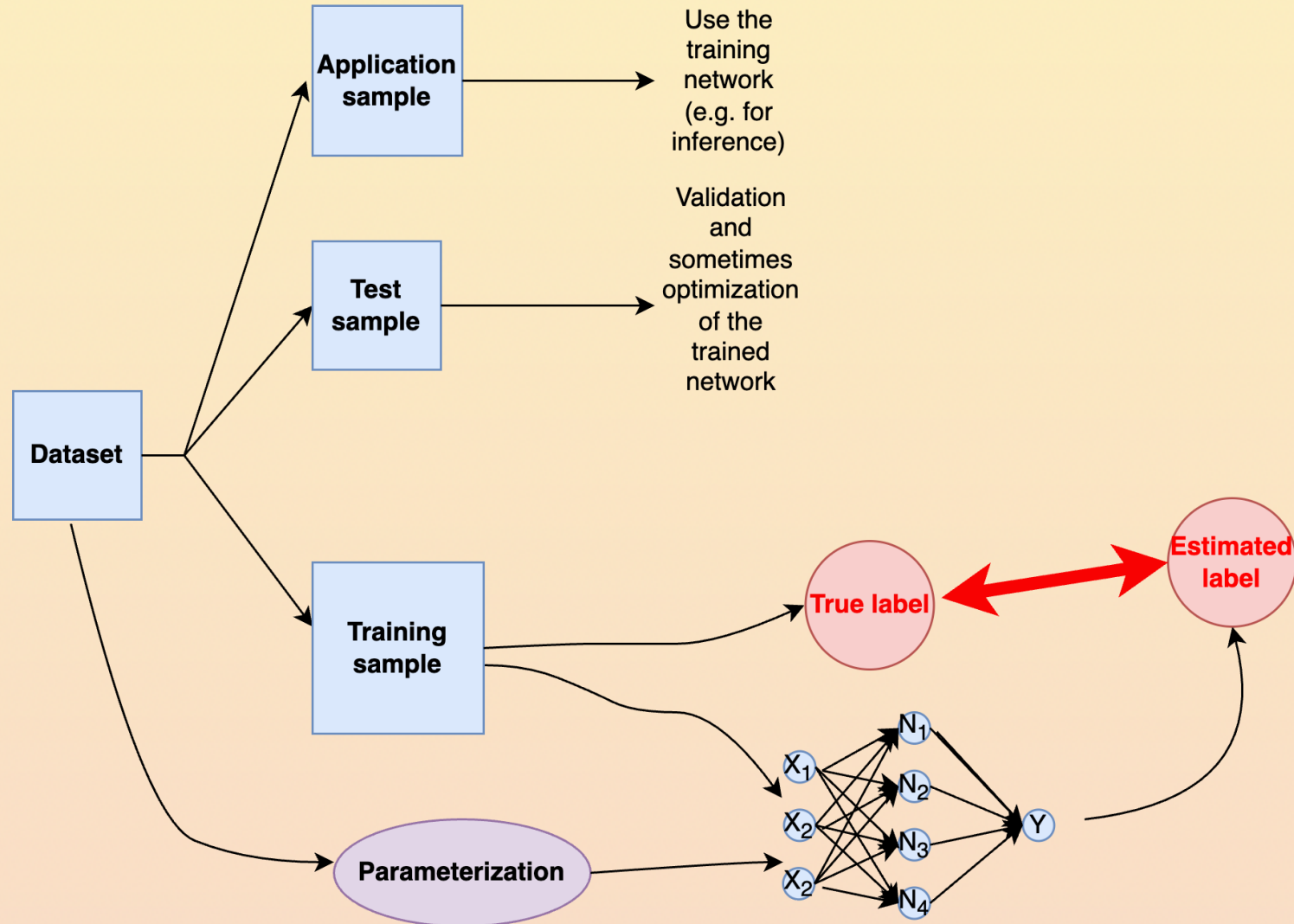
Unsupervised Learning



Reinforcement Learning

Supervised Learning

Training a model



Brain activity



INPUT



BRAIN

=

OUTPUT

Brain activity



INPUT



BRAIN



OUTPUT

Brain Activity



INPUT

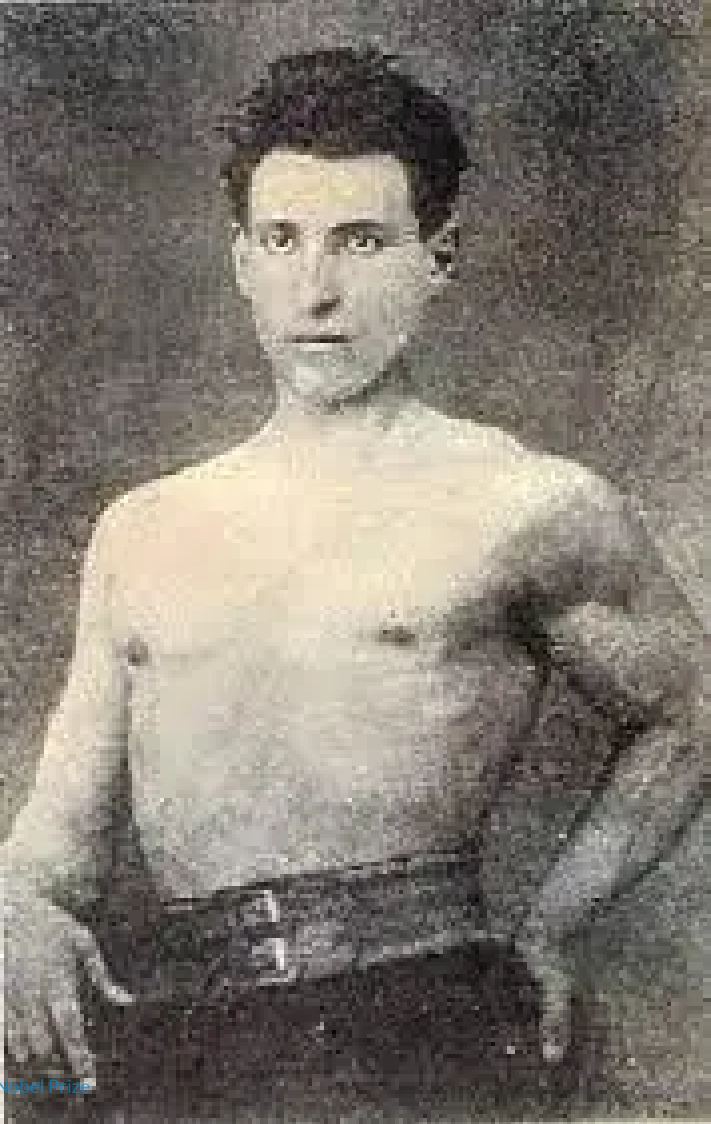


BRAIN

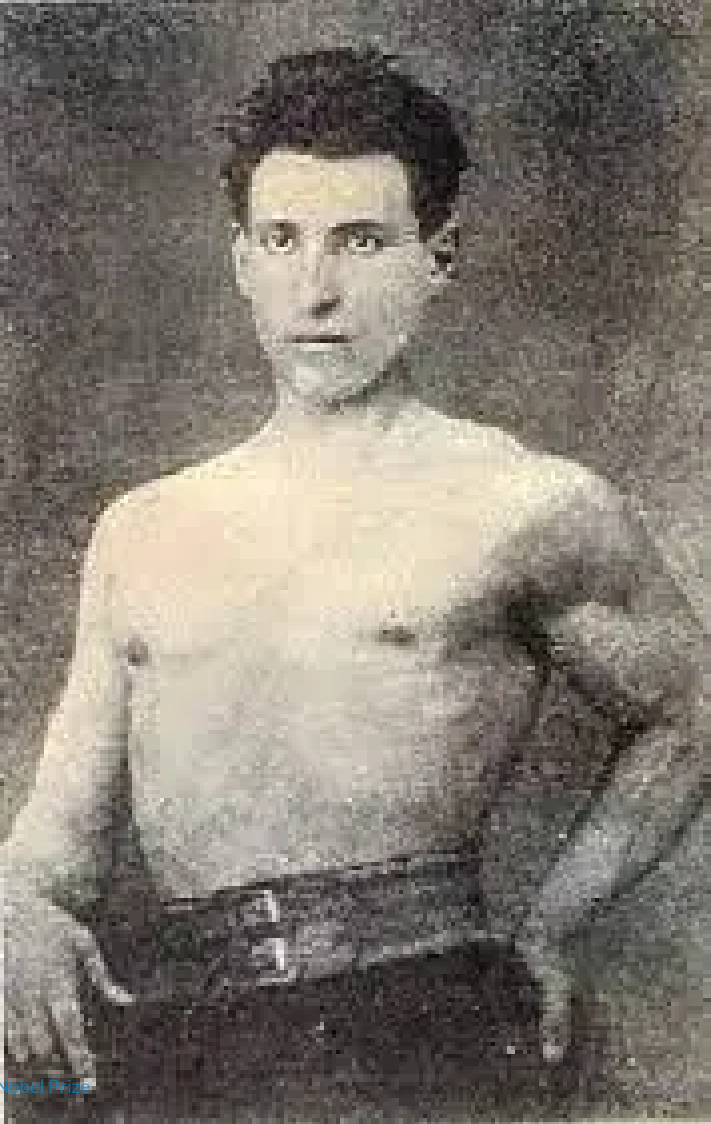
=

OUTPUT

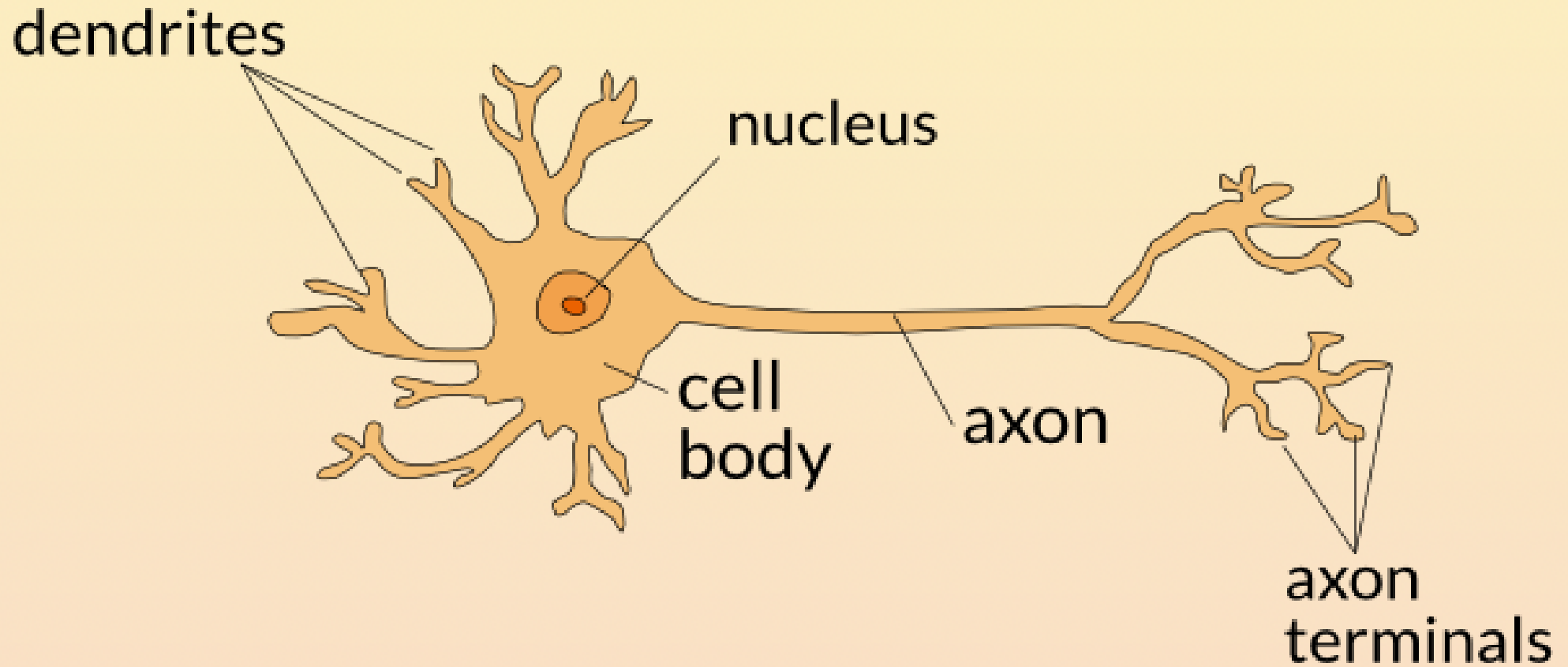
Santiago Ramón y Cajal



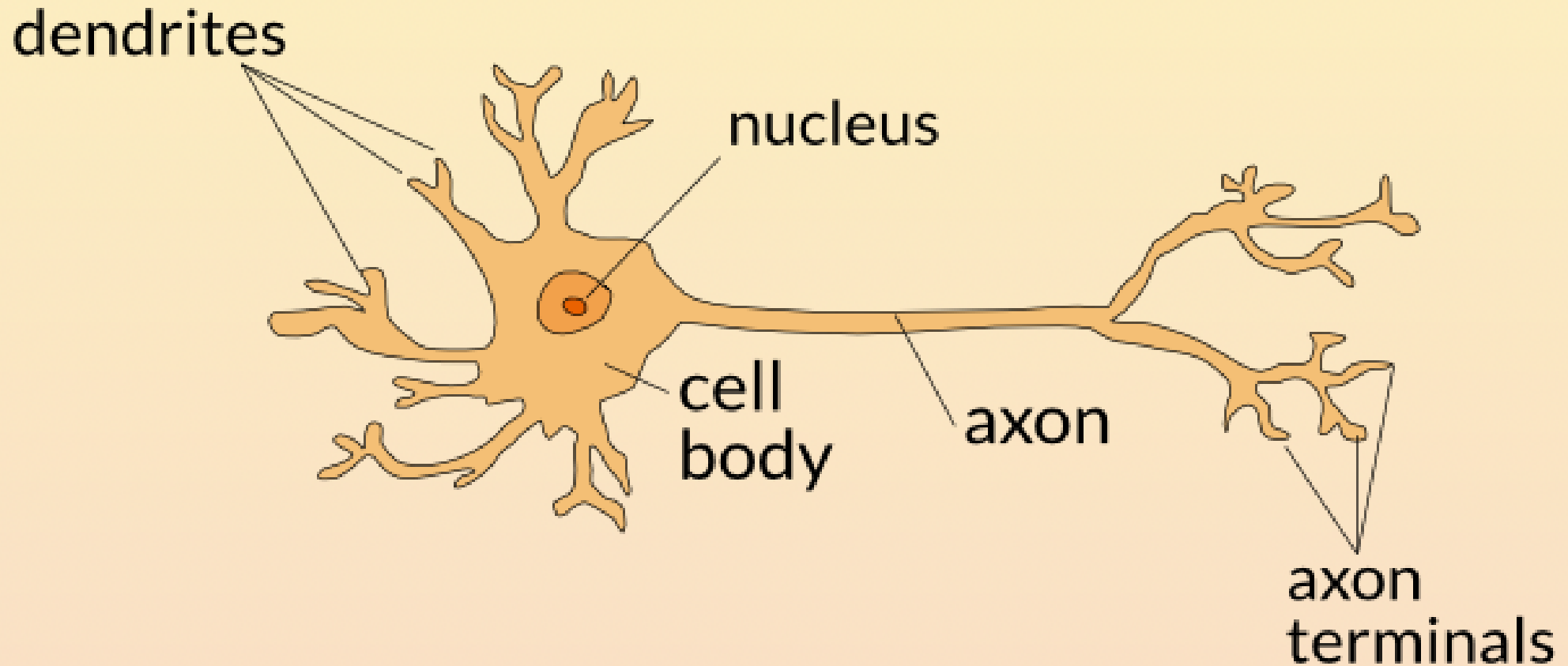
Santiago Ramón y Cajal



Real neurons



Real neurons



$$I = C \frac{dV}{dt} + G_{Na} m^3 h (V - V_{Na}) + G_K n^4 (V - V_K) + G_L (V - V_L)$$

Computationally heavy



Simplified Neurons

Bulletin of Mathematical Biology Vol. 52, No. 1/2, pp. 99–115, 1990.
Printed in Great Britain.

0092–8240/90\$3.00 + 0.00
Pergamon Press plc
Society for Mathematical Biology

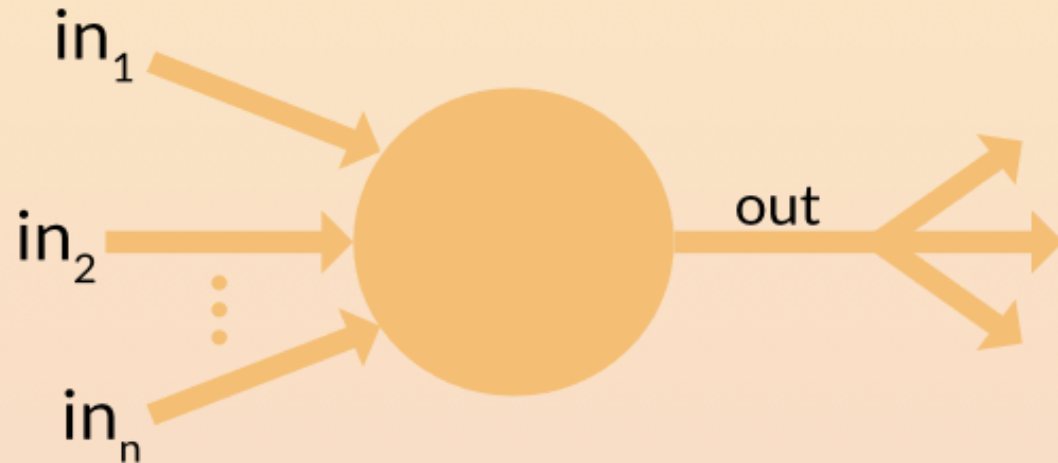
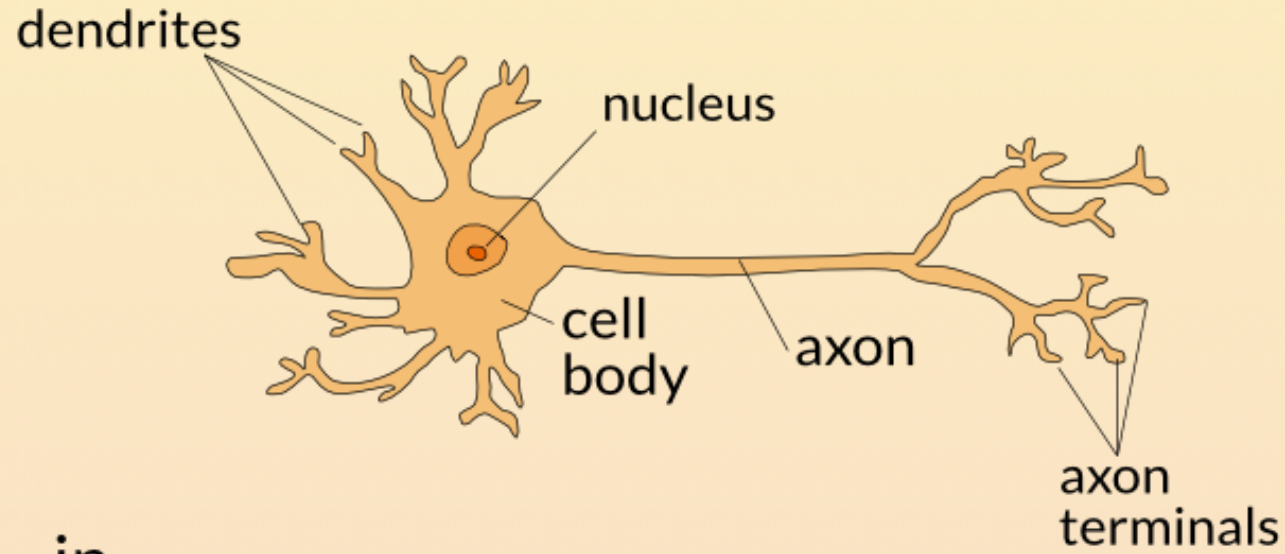
A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

- WARREN S. MCCULLOCH AND WALTER PITTS
University of Illinois, College of Medicine,
Department of Psychiatry at the Illinois Neuropsychiatric Institute,
University of Chicago, Chicago, U.S.A.

Because of the “all-or-none” character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

Perceptrons

$$y = g\left(w_0 + \sum w_i x_i\right)$$



Perceptron step by step

- Linear combination of the inputs

$$\sum_j w_j x_j$$

- Activation function (imitates the activation of real neurons, where a voltage peak, a **spike**, is generated when the injected potential passes a threshold)

$$g\left(w_0 + \sum_j w_j x_j\right)$$

- Bias term, an order-zero term in the inputs (translation)

$$\hat{y} = g\left(w_0 + \sum_j w_j x_j\right)$$

- In matrix form, a row-column product

$$\hat{y} = g\left(w_0 + \mathbf{X}^T \mathbf{W}\right)$$

Activation Function

- Originally, the activation function was linear

$$g(z) = 1 \text{ if } z = w_0 + \sum w_i x_i \geq 0$$
$$g(z) = -1 \text{ if } z = w_0 + \sum w_i x_i < 0$$

- Activation function is the only chance of estimating nonlinear functions
- Otherwise, a neural network would be just a fancier linear regression model

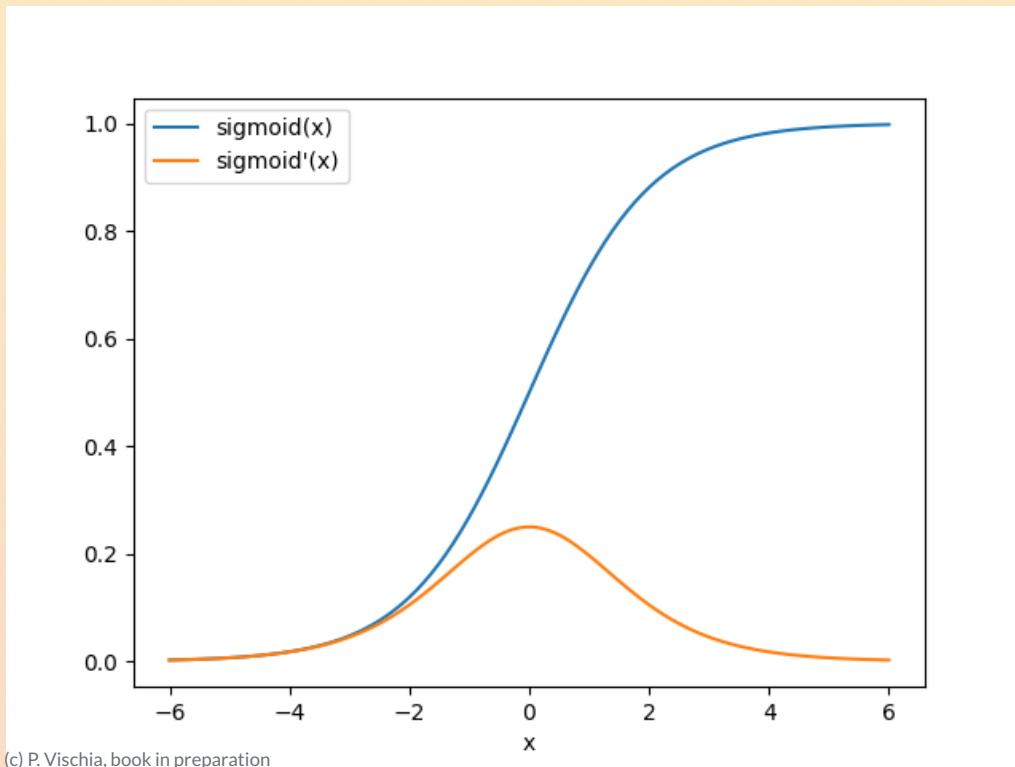
$$\hat{y} = g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ 2 \end{bmatrix}\right) = g(1 + 3x_1 + 2x_2)$$

Popular activation functions

Sigmoid

$$g(z) = \frac{1}{1 + e^{-z}}$$

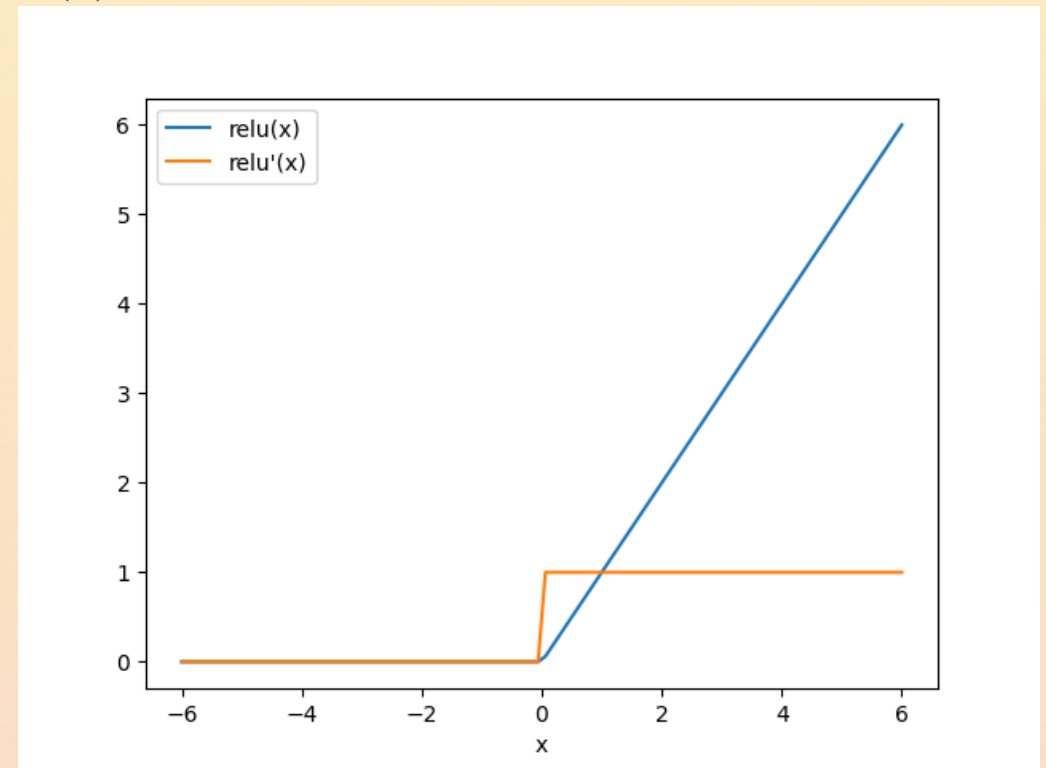
$$g'(z) = g(z)(1 - g(z))$$



Rectified Linear Unit (ReLU)

$$g(z) = \max(0, z)$$

$$g'(z) = 1 \text{ if } z > 0, 0 \text{ otherwise}$$

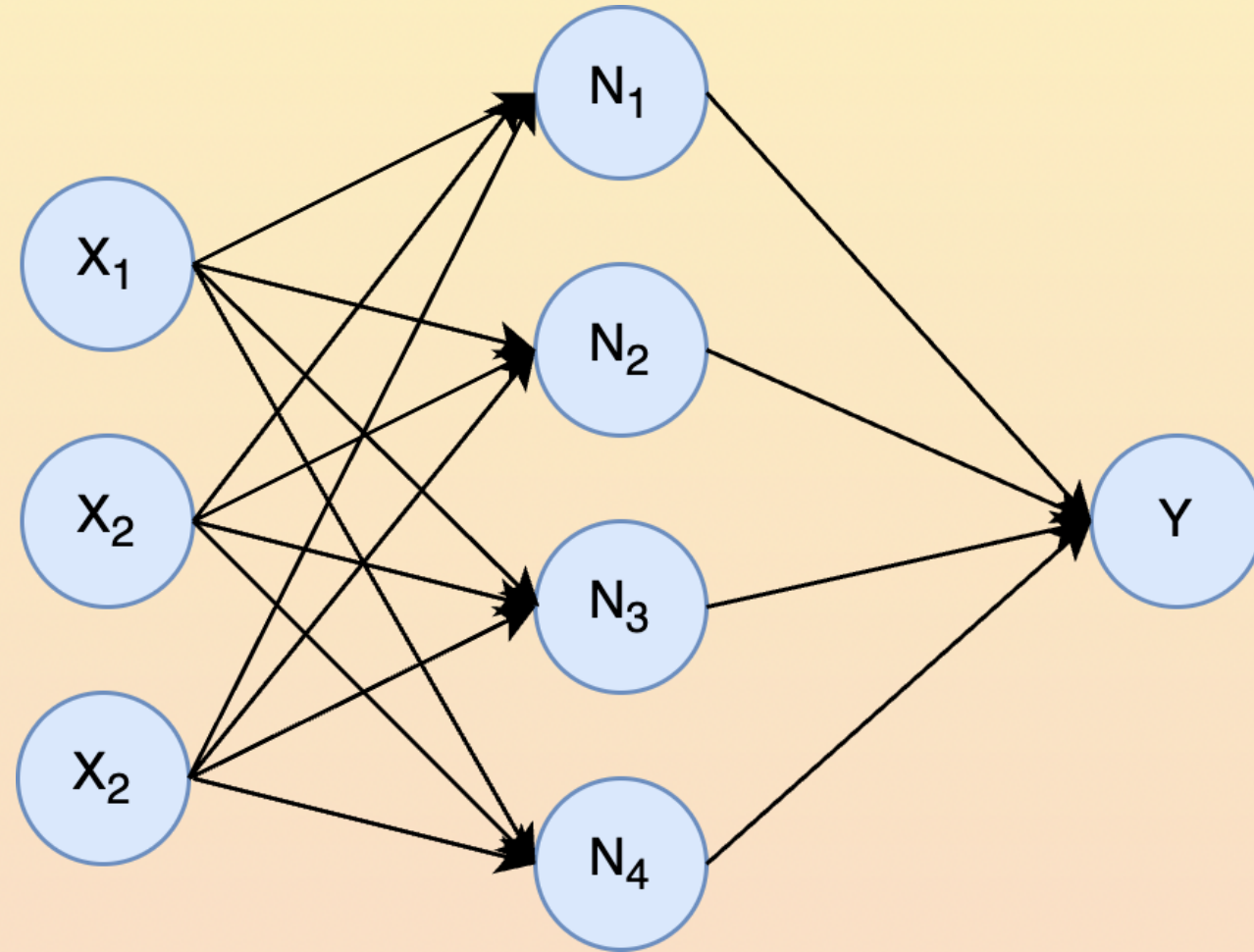


Multidimensional outputs

- \hat{y}_1, \hat{y}_2 , each one with the same formula as a single neuron \rightarrow just with an additional index

$$\hat{y}_i = g(w_{0,i} + \sum_j w_{j,i} x_j)$$

Artificial Neural Networks



Neural network with one internal layer

- Between input and hidden layer: $\mathbf{W}^{(1)}$
- Between hidden layer and output layer: $\mathbf{W}^{(2)}$
- Output of the hidden layer neurons:

$$z_i = g(w_{0,i}^{(1)} + \sum_j w_{j,i}^{(1)} x_j)$$

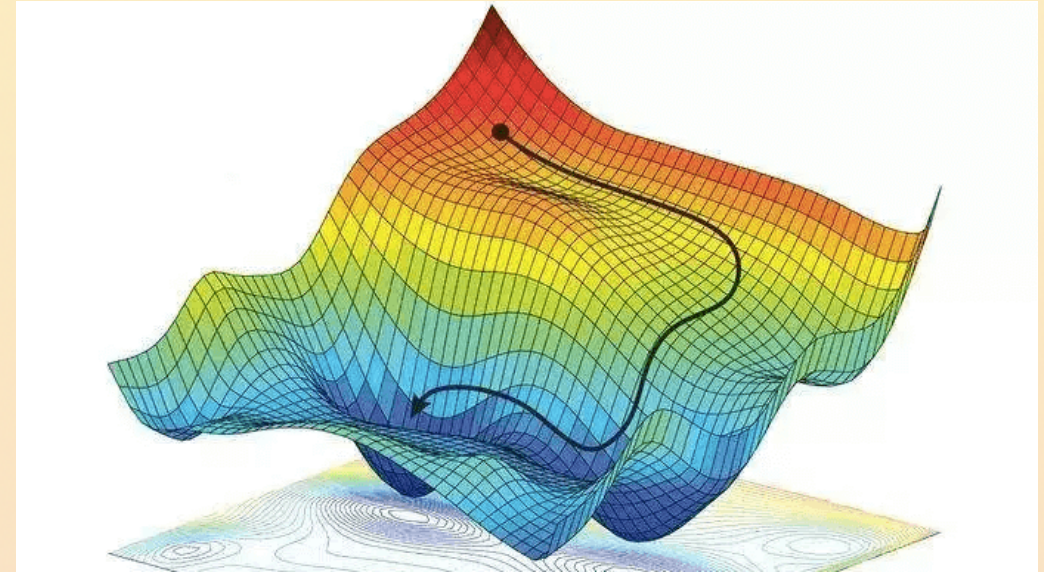
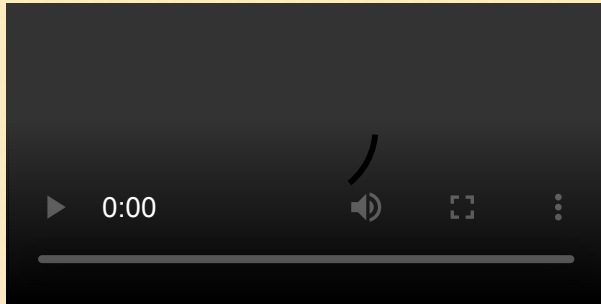
- Output of the network

$$\hat{y}_i = g(w_{0,i}^{(2)} + \sum_j w_{j,i}^{(2)} z_j)$$

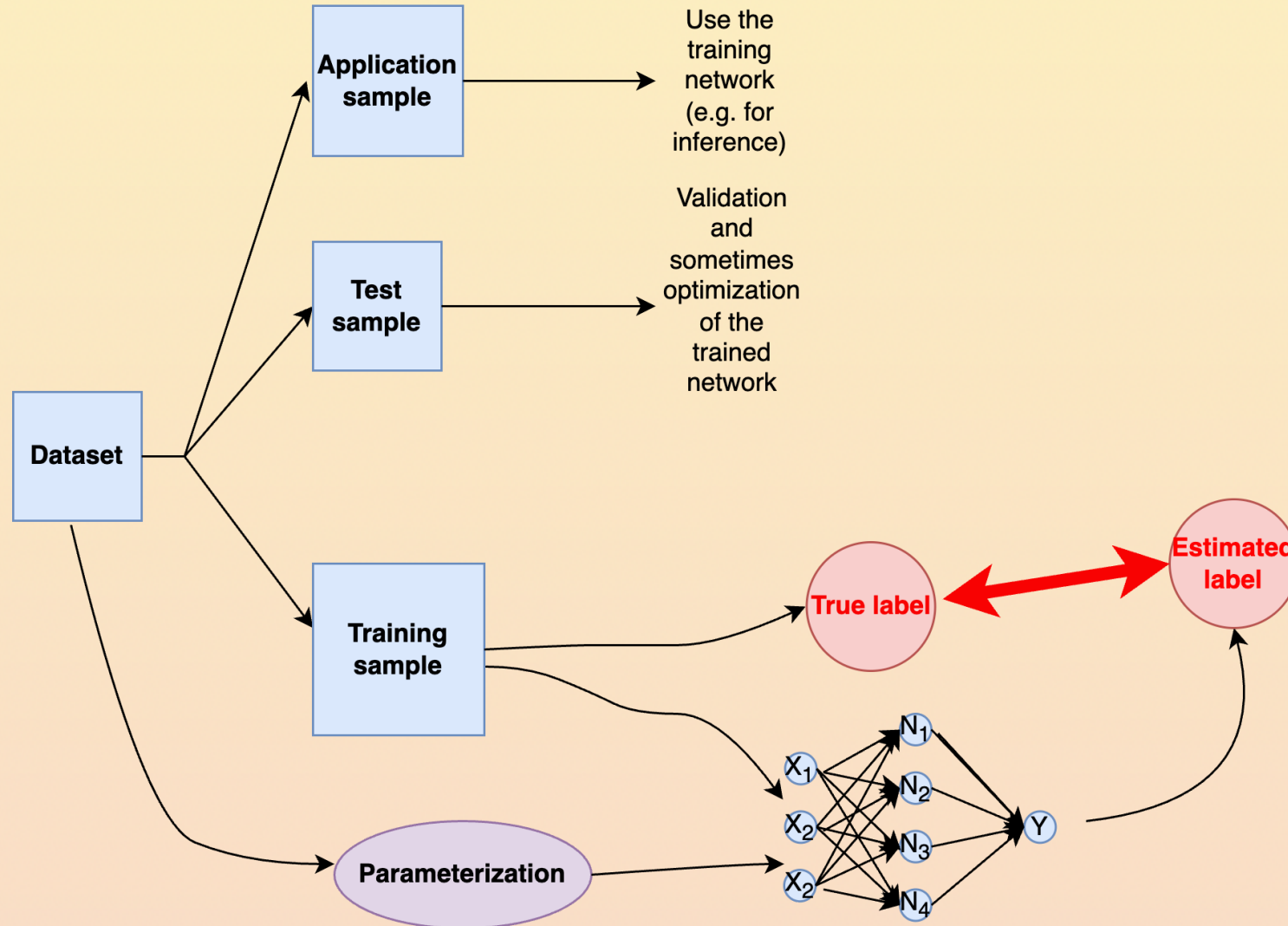
- The generalization to multiple outputs \hat{y}_i is also trivial

Gradient Descent

- Search for the "minimum error" as a function of the values of the training parameters (weights)

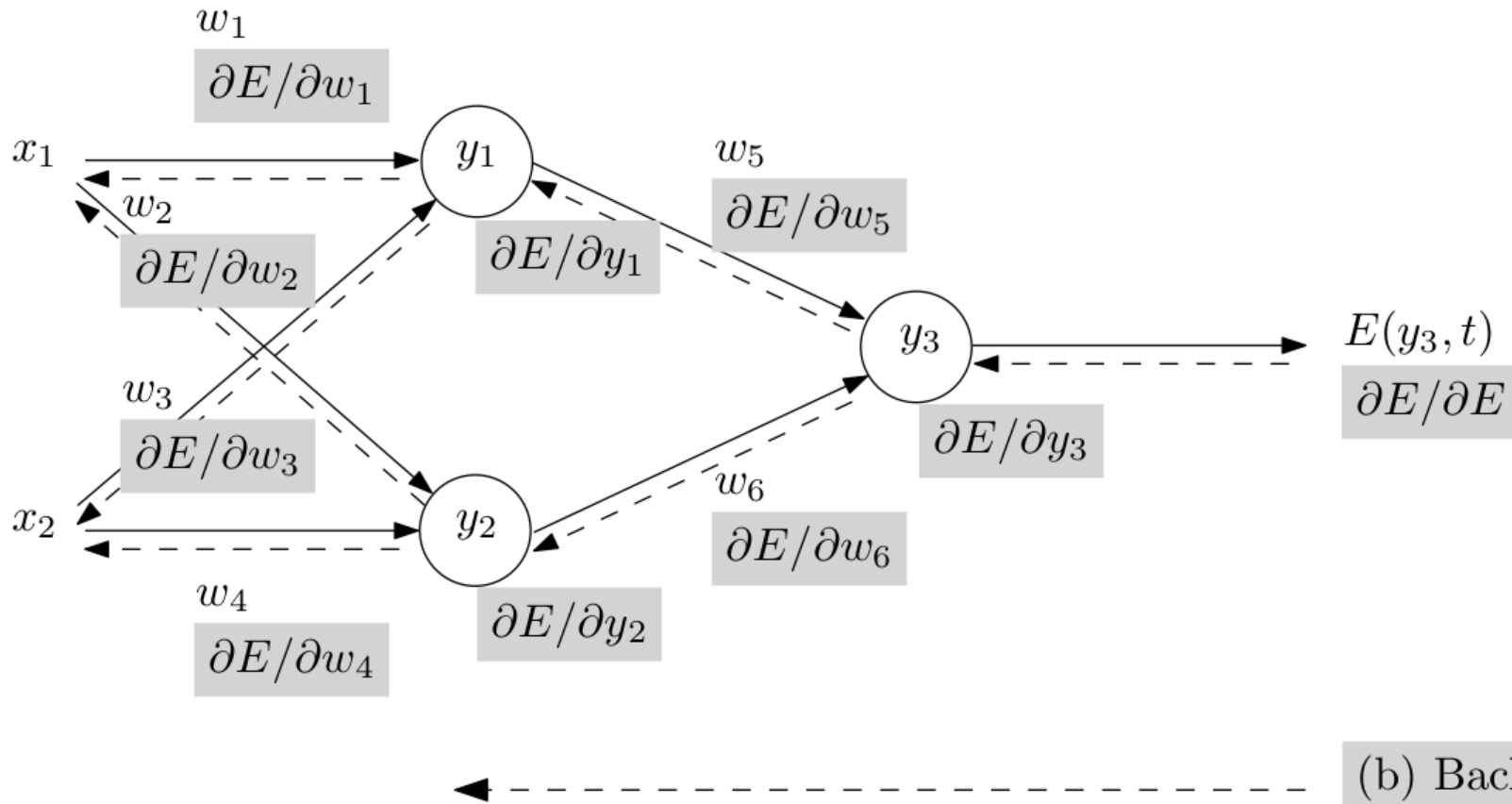


Learning



Backpropagation

(a) Forward pass



(b) Backward pass

Backpropagation

- Empirical Loss Function
- Cost function
- Empirical risk

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{*(i)})$$

- Minimized by:

$$\mathbf{W}^0 = \operatorname{argmin}_{\mathcal{W}} \mathbf{J}(\mathbf{W}) = \operatorname{argmin}_{\mathcal{W}} \mathbf{J}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{*(i)})$$

Loss function comes from inference

- Decision-theoretic approach (C.P. Robert, "The Bayesian Choice")
 - \mathcal{X} : observation space
 - Θ : parameter space
 - \mathcal{D} : decision (action) space
- **Statistical inference** take a decision $d \in \mathcal{D}$ related to parameter $\theta \in \Theta$ based on observation $x \in \mathcal{X}$, under $f(x|\theta)$
 - Typically, d consists in estimating $h(\theta)$ accurately

$$U(\theta, d) = \mathbb{E}_{\theta, d} [U(r)]$$

Loss function comes from inference

- Loss function: $L(\theta, d) = -U(\theta, d)$
 - Represents intuitively the loss or error in which you incur when you make a bad decision (a bad estimation of the target function)
 - Lower bound at 0: avoids "infinite utility" paradoxes (St. Petersburg paradox, martingale-based strategies)
- Generally impossible to uniformly minimize in d the loss for θ unknown
 - Need for a practical prescription to use the loss function as a comparison criterion in practice

Frequentist loss, Bayesian loss

- Frequentist loss (risk) is integrated (averaged) on \mathcal{X} : $R(\theta, \delta) = \mathbb{E}_{\theta} \left[L(\theta, \delta(x)) \right]$
 - $\delta(\cdot)$ is an `\textbf{estimator}` of θ (e.g. MLE)
 - Compare estimators, find the best estimator based on long-run performance for all values of unknown θ
 - Issues: based on long run performance (not optimal for x_{obs}); repeatability of the experiment; no total ordering on the set of estimators
- Bayesian loss: is integrated on Θ : $\rho(\pi, d|x) = \mathbb{E}^{\pi} \left[L(\theta, d)|x \right]$
 - π is the prior distribution
 - Posterior expected loss averages the error over the posterior distribution of θ conditional on x_{obs}
 - Can use the conditionality because x_{obs} is known!
 - Can also integrate the frequentist risk; integrated risk $r(\pi, \delta) = \mathbb{E}^{\pi} \left[R(\theta, \delta) \right]$ averaged over θ according to π (total ordering)

ANNs and Bayesian networks

- Standard ANN training essentially is a frequentist MLE
 - NN weights: true, unknown values
 - Data: random variable
- Bayesian networks treat weights ω as random (latent) variables, and condition on the observed data
 - Obtain $p(\omega|data)$ starting from prior belief $\pi(\omega)$ and likelihood $p(data|\omega)$
 - Predictions obtained as expectation values, $E_p[f] = \int f(\omega)p(\omega|data)d\omega$, averaging f weighting by the posterior
 - Marginalization leads to essentially learning the generative model (the pdfs), leading to interpretability

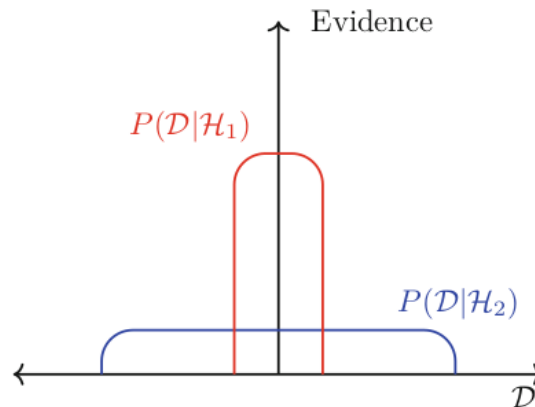


Fig. 3.4 Graphical illustration of how the evidence plays a role in investigating different model hypotheses. The simple model \mathcal{H}_1 is able to predict a small range of data with greater strength, while the more complex model \mathcal{H}_2 is able to represent a larger range of data, though with lower probability. Adapted from [45, 46]

Loss function: regression

- Regression is the prototype generalization of least squares regression
- Typically, mean square error is used as a loss function

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^m |\hat{y}_i - y_i|^2$$

- Measures the average spread the residuals
- Large errors will count more (will be suppressed first, in the minimization loop)
- Sometimes composite loss functions, see data challenge

Classification: Entropy vs cross entropy

- Cross entropy

$$\mathcal{L} = - (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

- where $y \in 0, 1$ is the true label, and $\hat{y} \in [0, 1]$ is the predicted probability for the positive class

- Entropy

$$S = -k_B \sum_i p_i \ln(p_i),$$

- where: S is the entropy, k_B is the Boltzmann constant, p_i is the probability of the i -th microstate

- Prediction error vs disorder, but both linked to information content (log probabilities)

Kullback-Leibler divergence

$$D_{\text{KL}}(P|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

- Additional information required to approximate P using Q .
 - $D_{\text{KL}}(P|Q) \geq 0$ (non-negative)
 - $D_{\text{KL}}(P|Q) = 0$ if and only if $P = Q$
- BCE is a special case of KL for binary classification, where $P = y$ and $Q = \hat{y}$

Negative log likelihood loss

- For m samples, the Negative log-likelihood (NLL) loss is:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \log P(y_i | \mathbf{x}_i)$$

- where $P(y_i | \mathbf{x}_i)$ is the predicted probability for the true class y_i for the i th sample,

Binary vs Multiclass classification

- The sigmoid activation function maps $[-\infty, +\infty] \rightarrow [0, 1]$
 - When doing binary classification, it can be loosely interpreted as probability p of belonging to one of the classes
 - Implicitly, the probability of belonging to the other one will be $1 - p$
 - When there is more than one class, the third Kolmogorov axiom $\sum_i p_i = 1$ is not enforced anymore
- Two alternative strategies
 - Tweak the activation function of the output layer: the [Softmax](#) activation function comes to the rescue
 - Tweak the loss function into one that forces outputs to sum up to 1

Softmax

- Converts raw output of the classifier (logits) into numbers in the interval $[0, 1]$ and that sum up to 1 , i.e. that are interpretable as probabilities
 - Logits are typically the output of a linear layer: $z_i = \mathbf{w}_i^\top \mathbf{x} + b_i$
- Given real numbers $\mathbf{z} = z_1, z_2, \dots, z_n$, softmax is:
 - Numerator expresses the relative importance of z_i , denominator is the normalization factor

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}, \quad \forall i \in 1, \dots, n.$$

Properties of softmax

- Softmax imposes the Kolmogorov axioms
 - Each element $0 \leq \text{Softmax}(z_i) \leq 1$ (first axiom)
 - Sum $\sum_{i=1}^n \text{Softmax}(z_i) = 1$ (second and third axioms)
- Interpret output for the predicted class y as:

$$P(y = i \mid \mathbf{x}) = \text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$

How to use for multiclass classification

- Neural network with e.g. linear output layer, and cross-entropy loss encouraging high probability for the correct class
 - y_{ij} is the one-hot encoded true label for the i th sample and j th class
 - $\hat{y}_{ij} = P(y = j \mid \mathbf{x}_i)$ is the softmax output

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^n y_{ij} \log(\hat{y}_{ij})$$

Numerical issues and translation invariance

- When the logits z_i have large magnitudes, the exponents in the softmax formula, e^{z_i} , can easily overflow.
- Shift the logits:

$$\text{Softmax}(z_i) = \frac{e^{z_i - \max(\mathbf{z})}}{\sum_{j=1}^n e^{z_j - \max(\mathbf{z})}}$$

- The output is invariant w.r.t. this operation

$$\frac{e^{z_i - c}}{\sum_{j=1}^n e^{z_j - c}} = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}, \quad \forall c$$

Lack of scaling invariance

- For the logits $\mathbf{z} = [1, 2, 3]$, the probabilities are

$$\text{Softmax}(\mathbf{z}) = \left[\frac{e^1}{e^1 + e^2 + e^3}, \frac{e^2}{e^1 + e^2 + e^3}, \frac{e^3}{e^1 + e^2 + e^3} \right] \approx [0.090, 0.245, 0.665]$$

- For the logits $\mathbf{z} = [2, 4, 6]$, the probabilities are

$$\text{Softmax}(\mathbf{z}) = \left[\frac{e^2}{e^2 + e^4 + e^6}, \frac{e^4}{e^2 + e^4 + e^6}, \frac{e^6}{e^2 + e^4 + e^6} \right] \approx [0.0159, 0.1173, 0.8670]$$

- Softmax amplifies differences, and is not scaling invariant
 - Used together with the loss function to penalize incorrect predictions more heavily when the model is confident but wrong (low predicted probability for the true class)

Which loss function to use?

- Softmaxed output (probabilities) requires a loss function that interprets its inputs as predicted probability distributions, and measures dissimilarity between that and the true labels.
 - For m samples, the NLL loss is:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \log P(y_i | \mathbf{x}_i)$$

- where $P(y_i | \mathbf{x}_i)$ is the predicted probability for the true class y_i for the i th sample,
- Substituting the softmax into the NLL loss

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \left(z_{i,c} - \log \sum_{k=1}^n e^{z_{i,k}} \right)$$

What if you don't want to use softmax?

- Alternatively, you can just take the model with logits, and use the (non-binary) cross entropy loss

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \left(z_{i,c} - \log \sum_{j=1}^n e^{z_{i,j}} \right)$$

- where:
 - $z_{i,j}$ is the logit for the i th sample and j th class
 - c is the index of the correct class for the i th sample
 - $\sum_{j=1}^n e^{z_{i,j}}$ is the partition function, ensuring normalization across classes
- It is the same expression as the previous slide!!!
- Essentially here the softmax is computed within the loss function, so it is completely equivalent to:
 - Apply a softmax activation function, then use the negative log likelihood loss on the predicted probabilities
 - Use the cross entropy loss on the logits

A few caveats

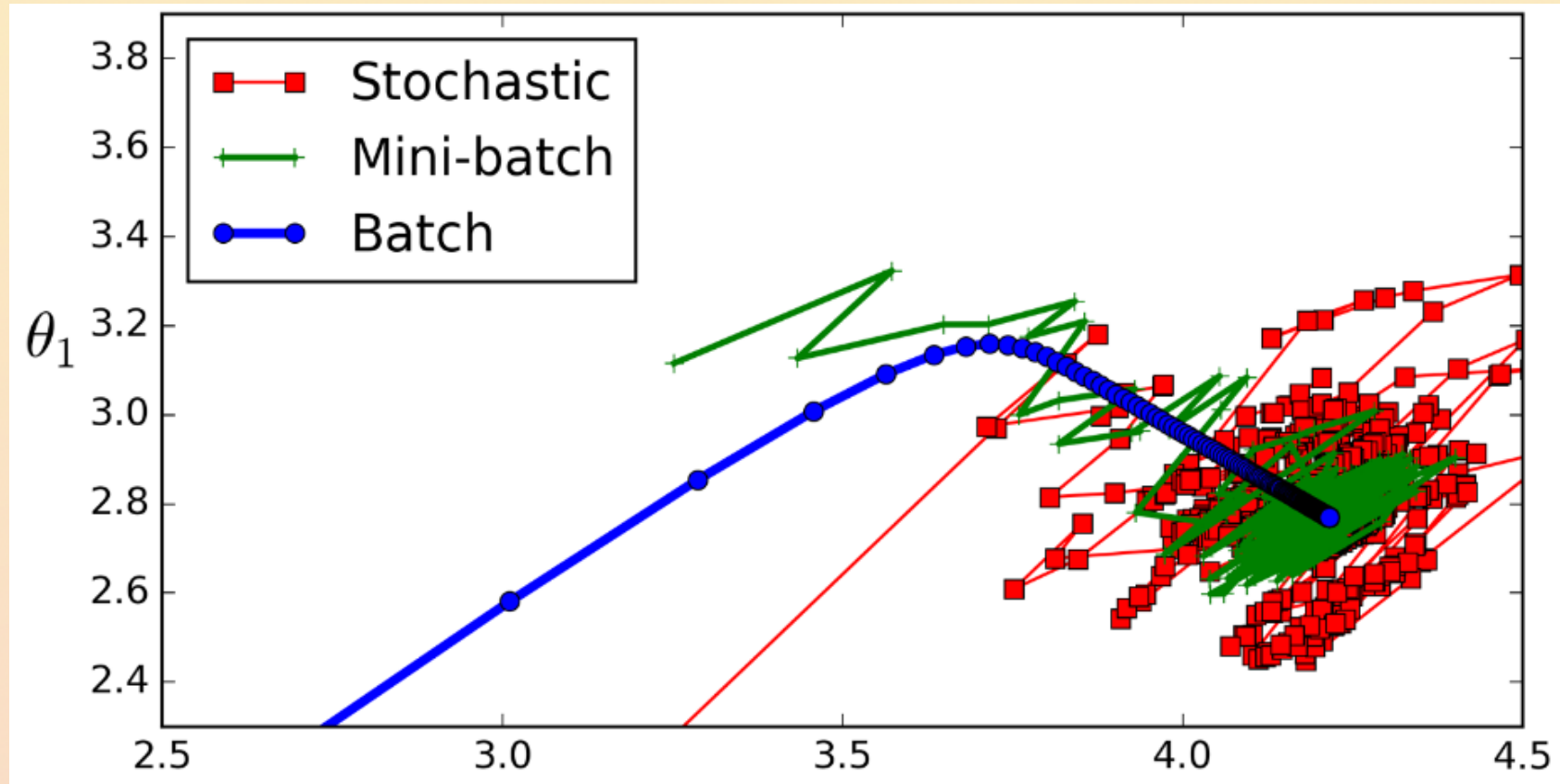
- This assumes each sample can belong to only one of the classes (multiclass problem)!!!
- If this is not true, the problem is a [multilabel problem](#)
 - Cannot use softmax, because now the probabilities of belonging to each class must be independent
 - Use other loss functions like [BCEWithLogitsLoss](#), that treat each label separately (i.e. makes an independent binary classification problem for each label)

Propagation Algorithm

- Initialise weights (for instance, $w \sim \text{Gaus}(0, \sigma^2)$)
- Loop until convergence
 - Compute output of the network with the current weights (**forward step**)
 - Compute gradient $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
 - Update weights (step defined by the learning rate) $\mathbf{W} \leftarrow \mathbf{W} + \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- How often to update the weights? How large a step?

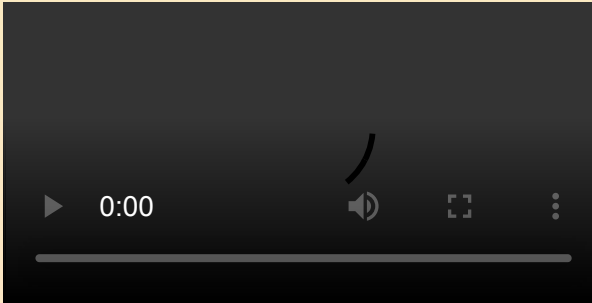
Sampling scheme

- Batch: compute on the whole training set (for large sets becomes too costly)
- Stochastic: compute on one sample (large noise, difficult to converge)
- Mini-batch: use a relatively small sample of data (tradeoff)



Descent strategies

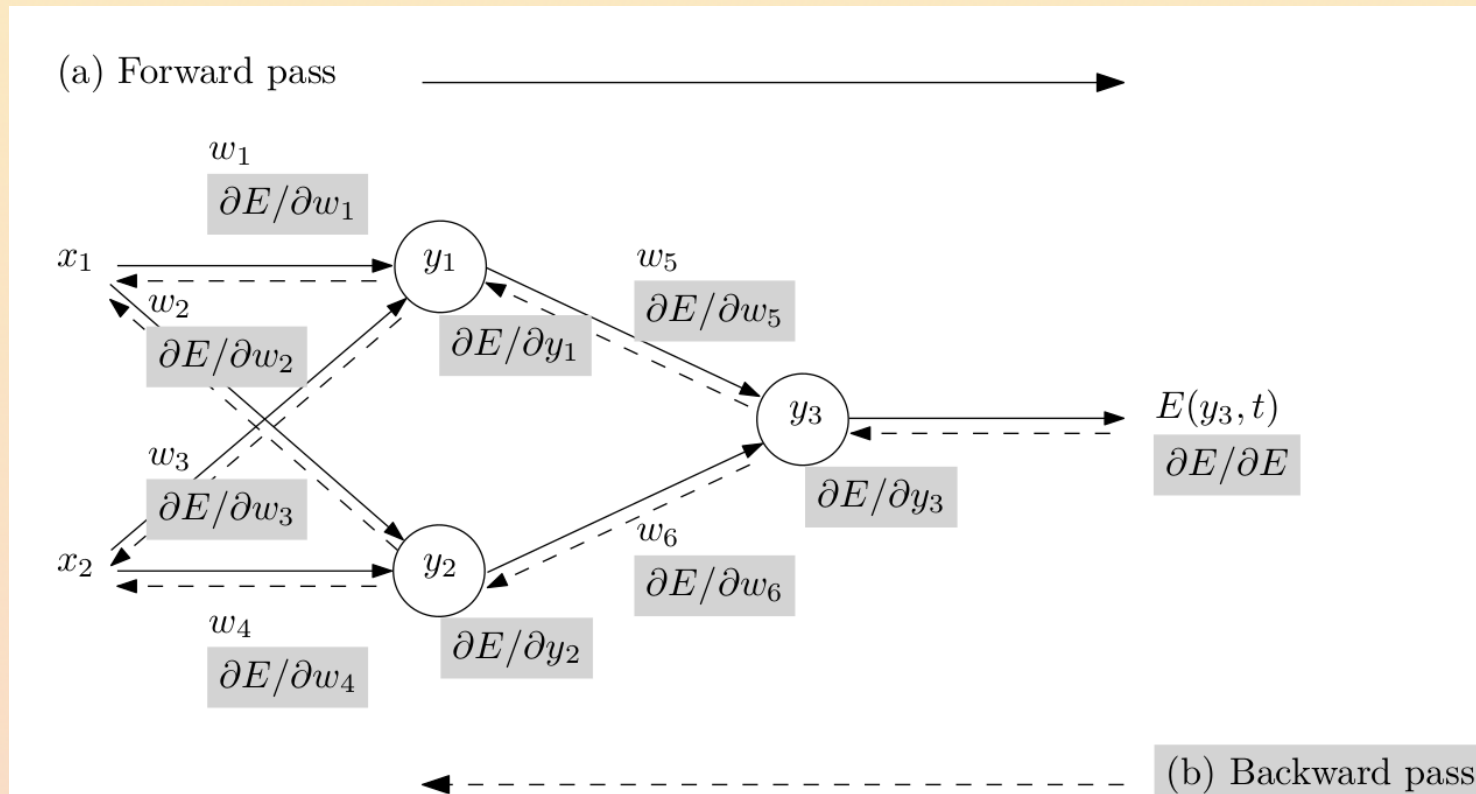
- Mostly nonconvex optimization: very complicated problem, convergence in general not guaranteed
- Nesterov momentum: big jumps followed by correction seem to help!
- Adaptive moments: gradient steps decrease when getting closer to the minimum (avoids overshooting)



Backpropagation summary

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{*(i)}), \quad \mathbf{W}^0 = \operatorname{argmin}_{\mathbf{W}} \mathbf{J}(\mathbf{W})$$

$$\mathbf{W} \leftarrow \mathbf{W} + \eta \frac{\partial \mathbf{J}(\mathbf{W})}{\partial \mathbf{W}}$$



Jacobian and Hessian

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$$

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

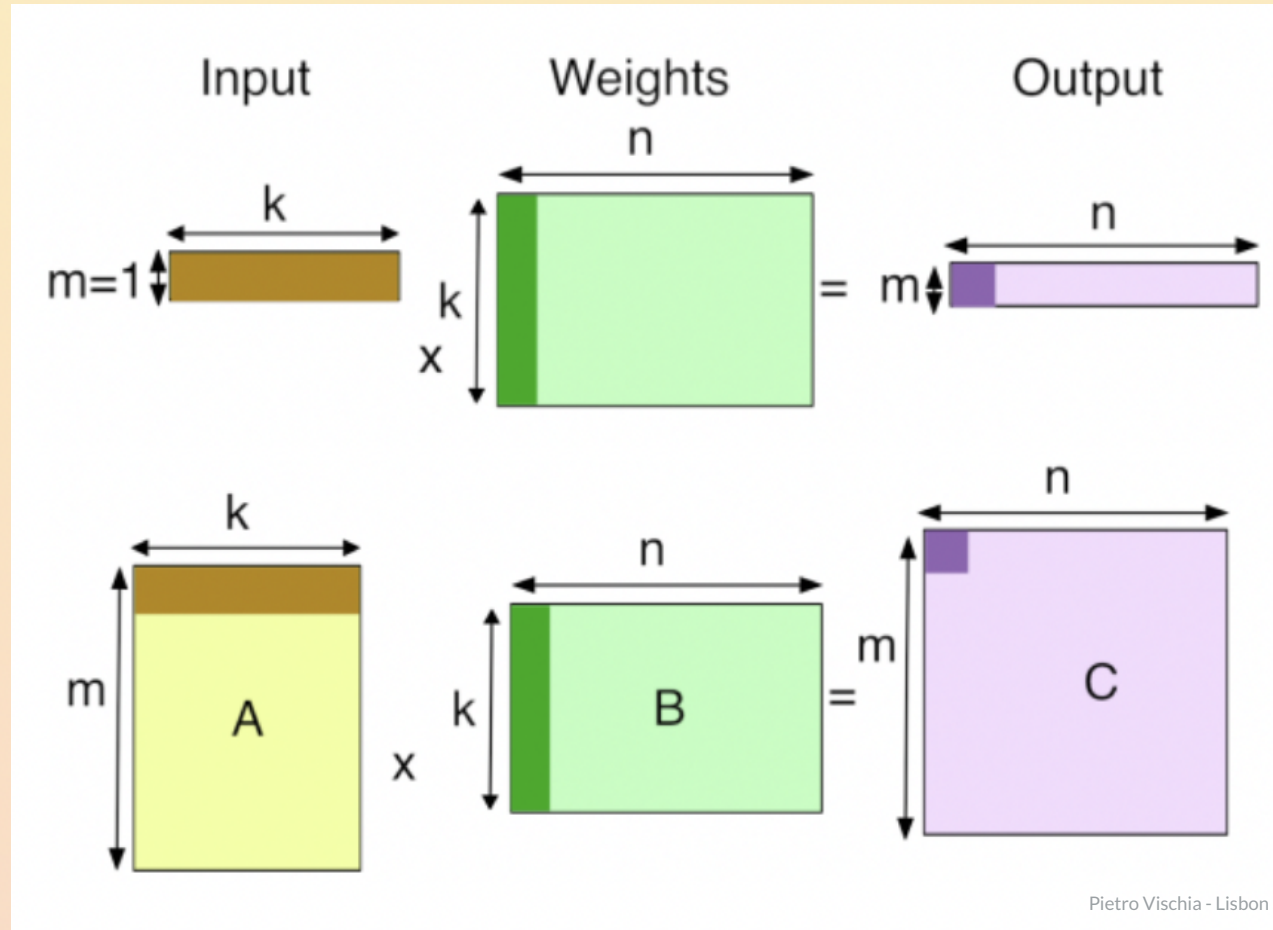
$$\mathbf{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2^2}(\mathbf{x}) & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_n \partial x_2}(\mathbf{x}) & \dots & \frac{\partial^2 f}{\partial x_n^2}(\mathbf{x}) \end{bmatrix}$$

- $\mathbf{H}(f(\mathbf{x})) = \mathbf{J}(\nabla f(\mathbf{x}))$
(describes local curvature)

Matrix multiplication

- Neural network weights expressible as matrices
- Generalize matrix calculus to tensors ([tensorflow](#))
- Optimize for efficient tensor calculus (e.g. GPU \rightarrow TPU, computational tricks)



Example: Google's TPUs

- Systolic flow
 - Hide four-stage process within the matrix multiplication operation
 - E.g. decoupled access/execution when reading weights
 - Trick flow control into thinking inputs are read and update results at once

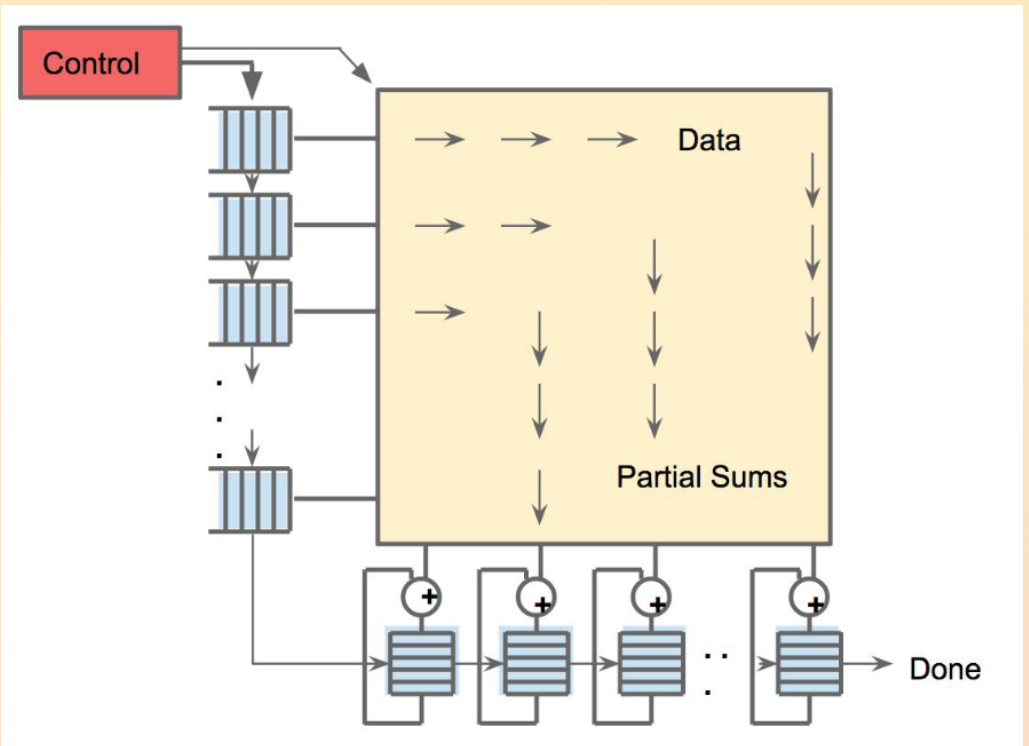
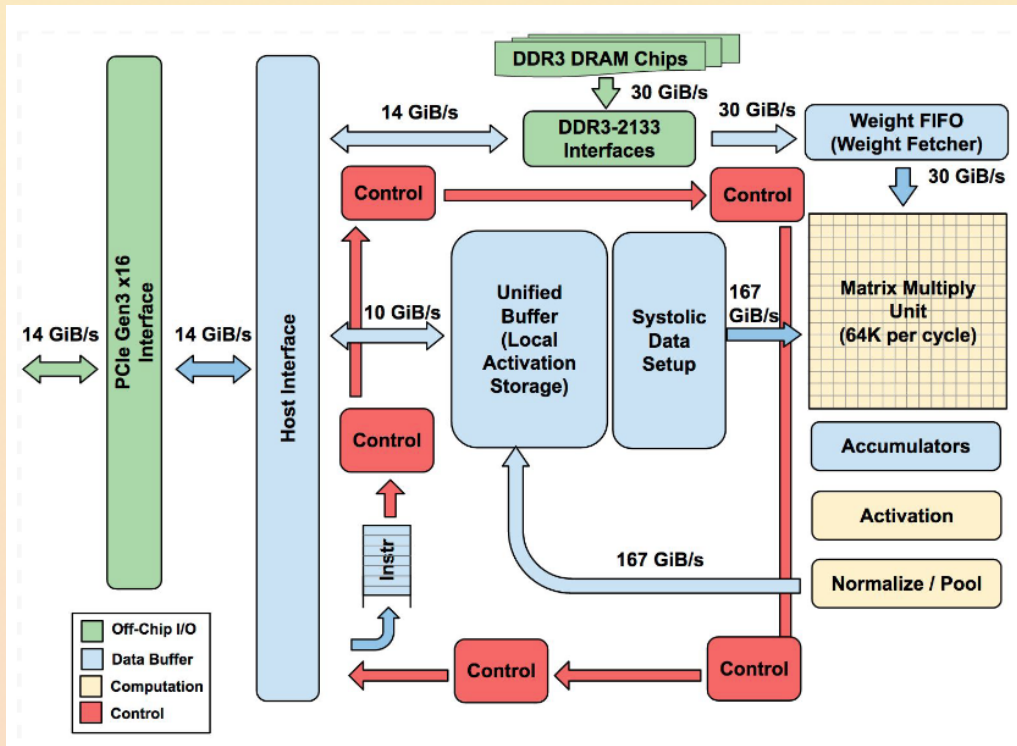
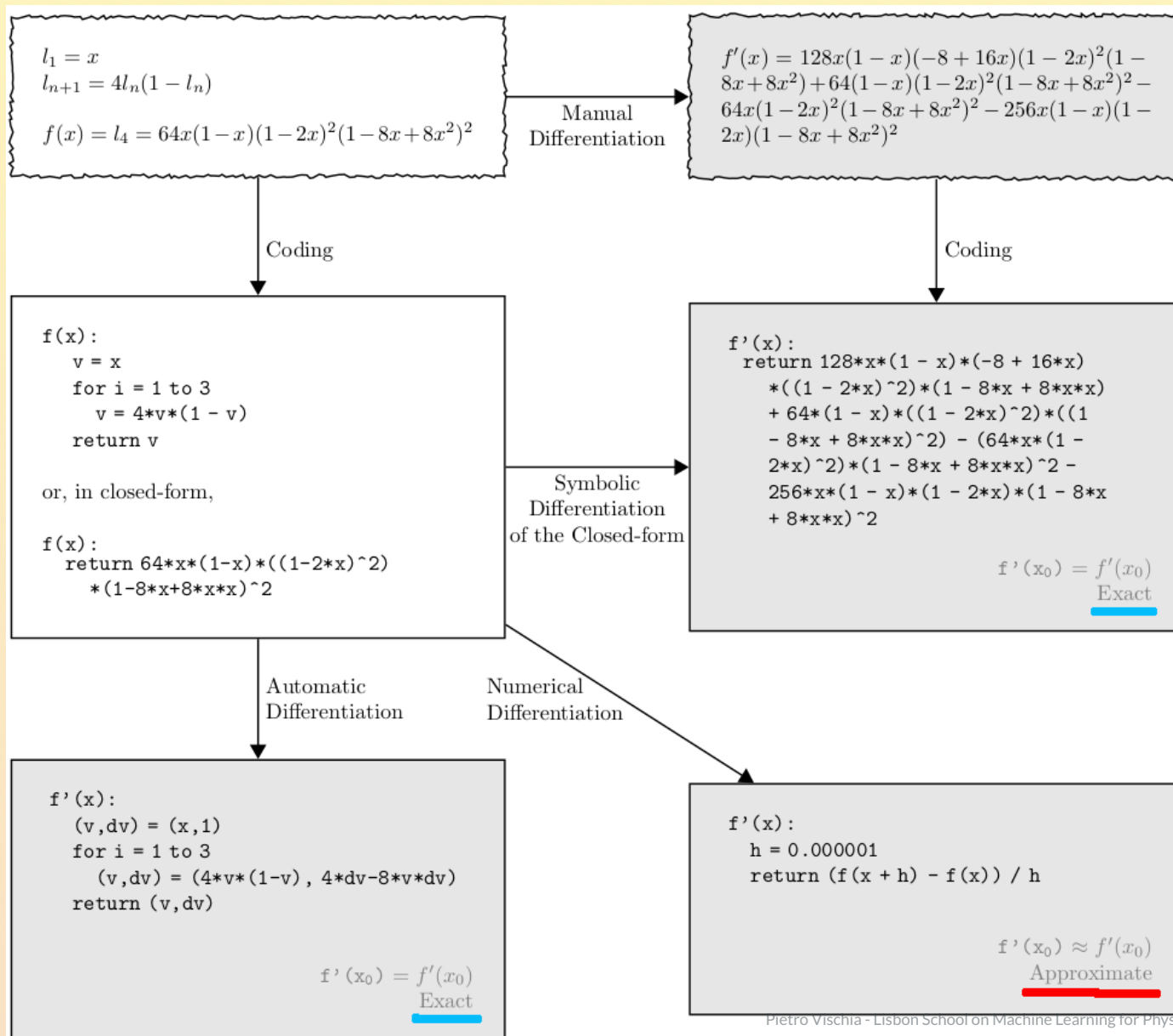


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Derive



Derivatives in machine learning

- Modern techniques rely heavily on full differentiability
 - Variational inference
 - Bijections (e.g. normalizing flows)
- Numerous software efforts
 - Pyro
 - ProbTorch
 - PyProb
 - Edward
 - TensorFlow Probability
 - Theano
 - Jax
 - Dex

Automatic Differentiation has many names

- Automatic differentiation
- Algorithmic differentiation
- AD
- Autodiff
- Algodiff
- Autograd

Derivatives

- Derivative: sensitivity of a function value to a change in its argument
 - instantaneous rate of change
 - can approximate with average rate

- $f : \mathbb{R} \rightarrow \mathbb{R}$

- $y = f(x)$

- y : dependent variable
- x : independent variable

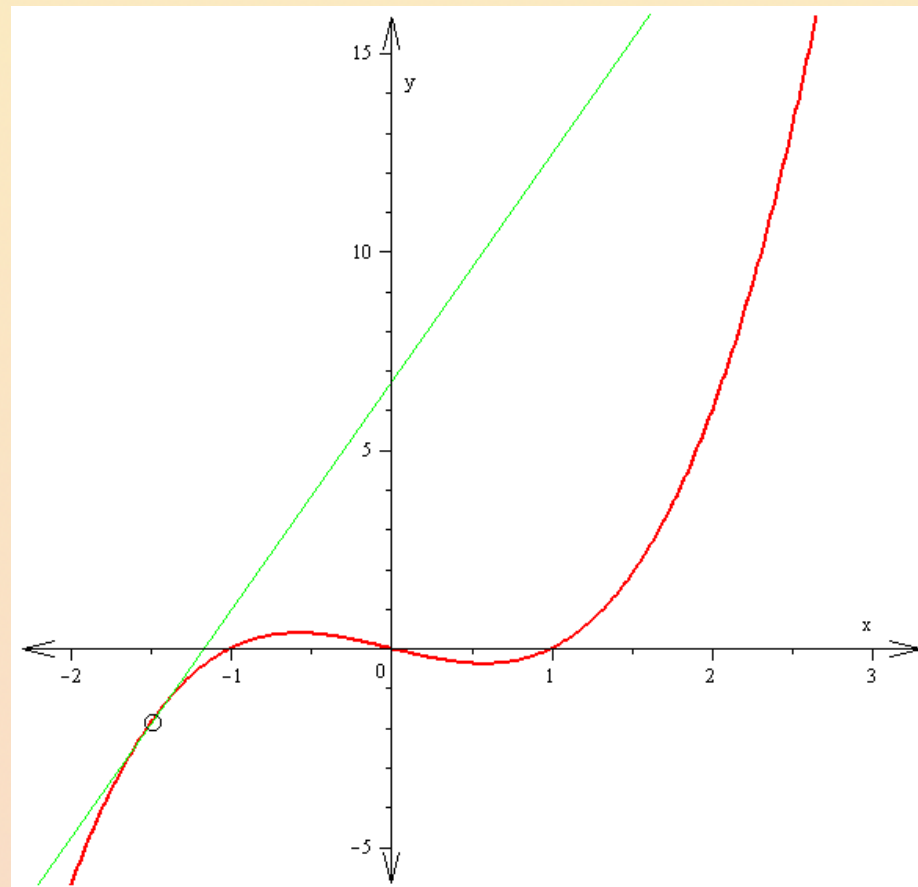
- Leibniz: $\frac{dy}{dx}$

- Lagrange: $f'(x)$

- Newton: \dot{y}

- Linear operator
(higher-order function)

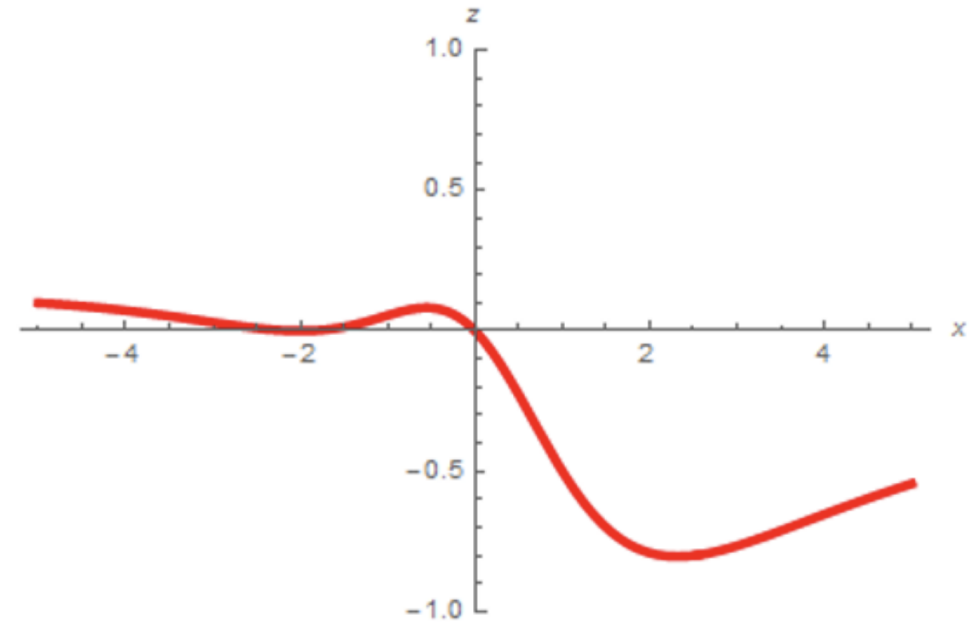
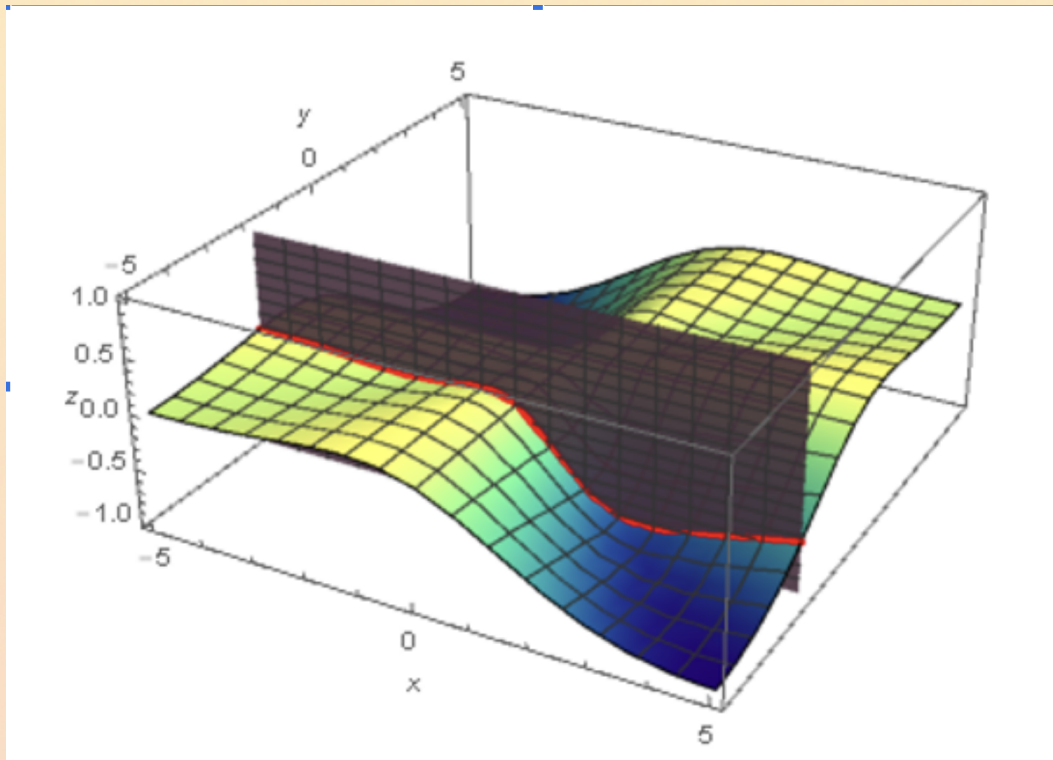
in programming languages: $(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$



Partial derivatives

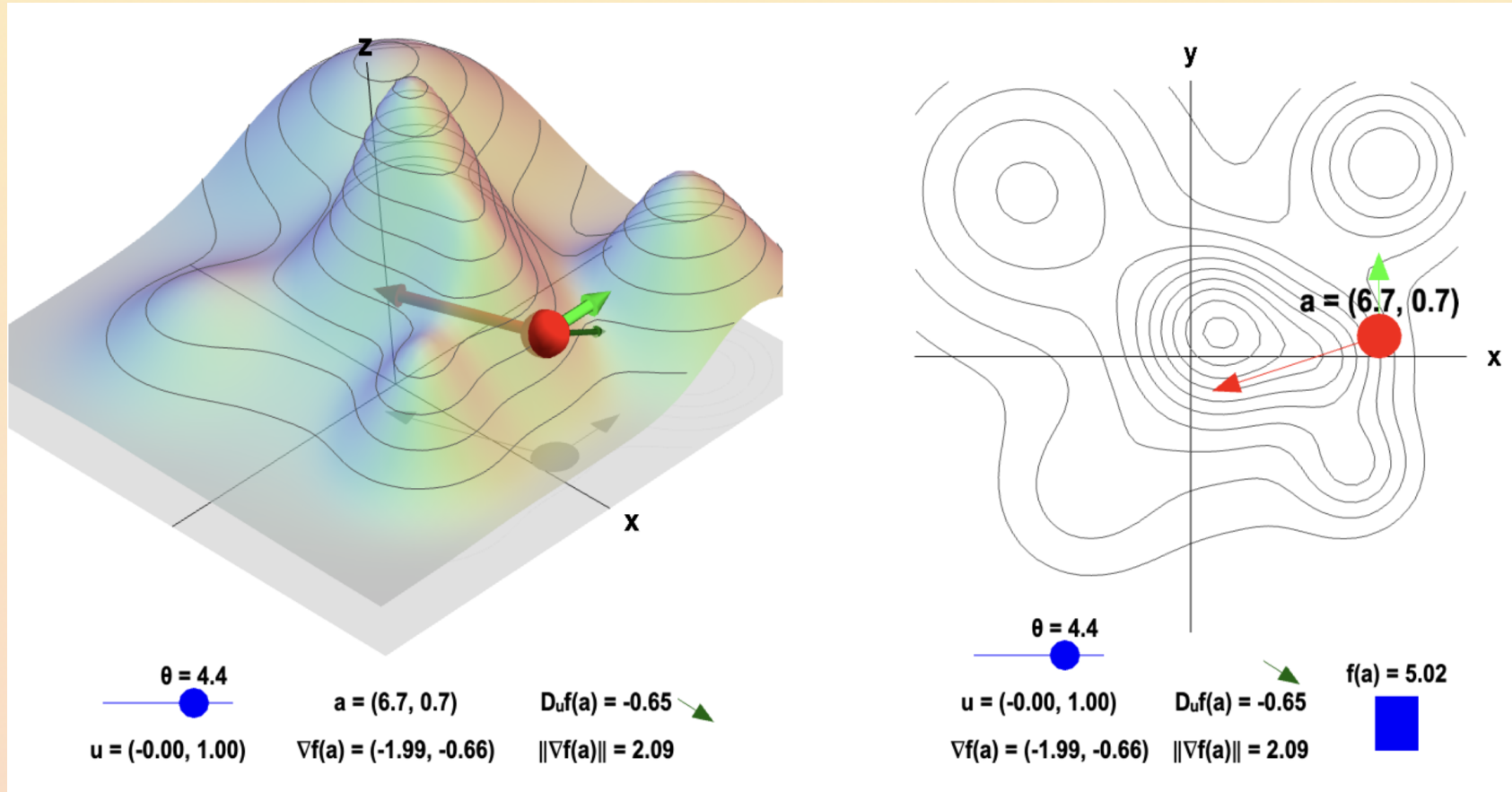
- $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Differentiate w.r.t. one independent variable (while keeping the others constant)

$$z(x, y) = 2x^2 + 3xy + y^3, \quad \frac{\partial z(x, y)}{\partial x} = 4x + 3y, \quad \frac{\partial z(x, y)}{\partial y} = 3x + 3y^2$$



Gradient

- $f : \mathbb{R}^n \rightarrow \mathbb{R}$ $\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$
- **Gradient**: the vector of all partial derivatives
(represents the direction with the largest rate of change)



Total Derivative

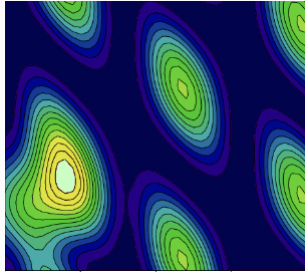
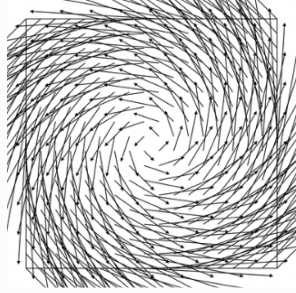
- Derivative w.r.t. all variables (independent and dependent)

$$f(t, x(t), y(t)) \quad \longrightarrow \quad \frac{df}{dt} = \frac{df}{dt} \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x} \frac{\partial x}{\partial t} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial t}$$

- Accumulate all direct and indirect contributions from the partial derivatives to the total derivative
 - Sum all the contributions that are responsible for the change in value of a variable
 - Crucial for backpropagation

Matrix calculus...

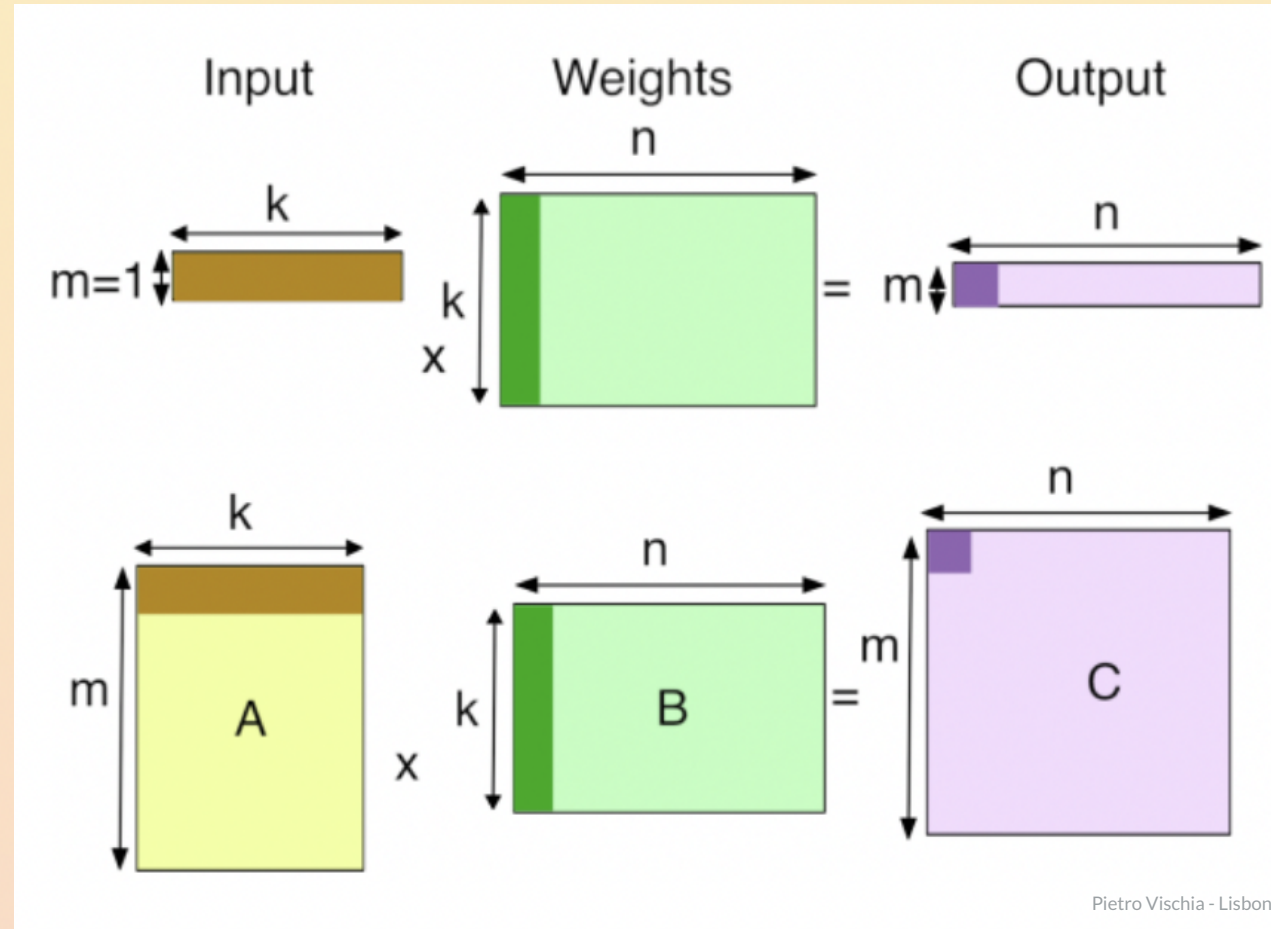
- Fundamental extension to multivariate functions

	Scalar output	Vector output
Scalar input	$\mathbb{R} \rightarrow \mathbb{R}$	$\mathbb{R} \rightarrow \mathbb{R}^m$
Vector input	$\mathbb{R}^n \rightarrow \mathbb{R}$ (scalar field) 	$\mathbb{R}^n \rightarrow \mathbb{R}^m$ (vector field) 

- Typical neural network: $\mathbb{R}^n \rightarrow \mathbb{R}^m$ (sometimes with $m = 1$)
- Loss function (e.g. KL divergence): $\mathbb{R}^n \rightarrow \mathbb{R}$

...in Machine Learning

- Neural network weights expressible as matrices
- Generalize matrix calculus to tensors ([tensorflow](#))
- Optimize for efficient tensor calculus (e.g. GPU \rightarrow TPU, computational tricks)



Jacobian and Hessian

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$$

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

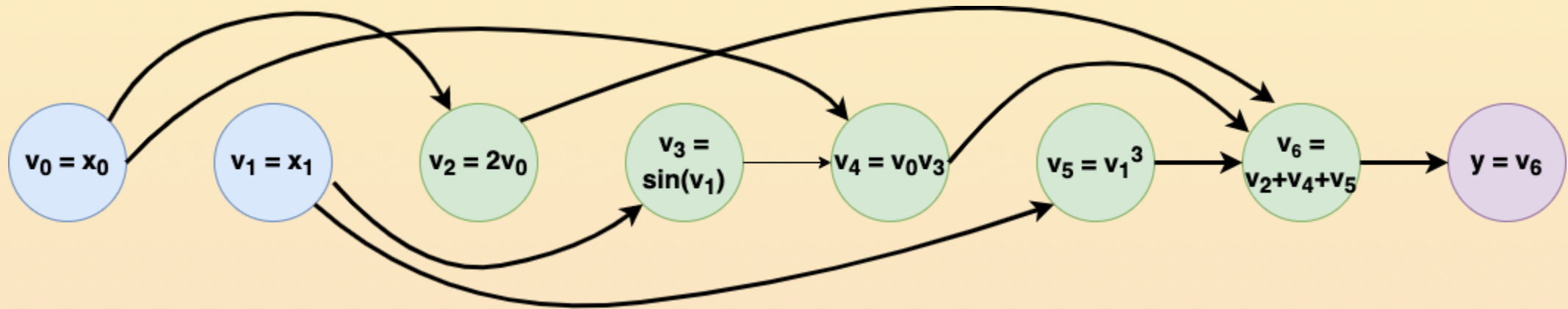
$$\mathbf{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2^2}(\mathbf{x}) & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_n \partial x_2}(\mathbf{x}) & \dots & \frac{\partial^2 f}{\partial x_n^2}(\mathbf{x}) \end{bmatrix}$$

- $\mathbf{H}(f(\mathbf{x})) = \mathbf{J}(\nabla f(\mathbf{x}))$
(describes local curvature)

Automatic differentiation

$$z(x, y) = 2x + x \sin(y) + y^3$$



Forward mode

- To the extreme, $f : \mathbb{R} \rightarrow \mathbb{R}^m$
- Evaluates $(\frac{\partial f_1}{\partial x}, \dots, \frac{\partial f_m}{\partial x})$
- Computational cost of calculating $\mathbf{J}_f(\mathbf{x})$ for $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ in $\mathbb{R}^n \times \mathbb{R}^m$

$\mathcal{O}(n \text{ time}(f))$

Reverse mode

- To the extreme, $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Evaluate $\nabla f(\mathbf{x})(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$

$\mathcal{O}(m \text{ time}(f))$

Forward and reverse (==backprop) modes

Primal: independent to dependent

Adjoint (derivatives): dependent to independent

$$y(\mathbf{x}) = 2x_0 + x_0 \sin(x_1) + x_1^3$$

Fwd Primal Trace Atomic operation	Value in (1, 2)	Fwd Tangent Trace (set $\dot{x}_0 = 1$ to compute $\frac{\partial y}{\partial x_0}$) Atomic operation	Value in (1, 2)
$v_0 = x_0$ $v_1 = x_1$	1 2	$\dot{v}_0 = \dot{x}_0$ $\dot{v}_1 = \dot{x}_1$	1 0
$v_2 = 2v_0$ $v_3 = \sin(v_1)$ $v_4 = v_0 v_3$ $v_5 = v_1^3$ $v_6 = v_2 + v_4 + v_5$	2 0.9093 0.9093 8 10.9093	$\dot{v}_2 = 2\dot{v}_0$ $\dot{v}_3 = \dot{v}_1 \cos(v_1)$ $\dot{v}_4 = \dot{v}_0 v_3 + v_0 \dot{v}_3$ $\dot{v}_5 = 3\dot{v}_1 v_1^2$ $\dot{v}_6 = \dot{v}_2 + \dot{v}_4 + \dot{v}_5$	2×1 0×-0.41 $1 \times 0.9093 + 1 \times 0$ $3 \times 0 \times 4$ $2 + 0.9093 + 0$
$y = v_6$	10.9093	$\dot{y} = \dot{v}_6$	2.9093

Fwd Primal Trace Atomic operation	Value in (1, 2)	Rev Adjoint Trace (set $\bar{y} = 1$ to compute $\frac{\partial v}{\partial y}$) Atomic operation	Value in (1, 2)
$v_0 = x_0$ $v_1 = x_1$	1 2	$\bar{x}_0 = \bar{v}_0$ $\bar{x}_1 = \bar{v}_1$	2.9093 11.5839
$v_2 = 2v_0$ $v_3 = \sin(v_1)$ $v_4 = v_0 v_3$ $v_5 = v_1^3$ $v_6 = v_2 + v_4 + v_5$	2 0.9093 0.9093 8 10.9093	$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \partial v_2 / \partial v_0$ $\bar{v}_0 = \bar{v}_4 \partial v_4 / \partial v_0$ $\bar{v}_1 = \bar{v}_1 + \bar{v}_3 \partial v_3 / \partial v_1$ $\bar{v}_1 = \bar{v}_5 \partial v_5 / \partial v_1$ $\bar{v}_2 = \bar{v}_6 \partial v_6 / \partial v_2$ $\bar{v}_3 = \bar{v}_4 \partial v_4 / \partial v_3$ $\bar{v}_4 = \bar{v}_6 \partial v_6 / \partial v_4$ $\bar{v}_5 = \bar{v}_6 \partial v_6 / \partial v_5$	$\bar{v}_0 + \bar{v}_2 \times 2 = 2.9093$ $\bar{v}_4 \times v_3 = 0.9093$ $\bar{v}_1 + \bar{v}_3 \times \cos(v_1) = 11.5839$ $\bar{v}_5 \times 3v_1^2 = 12$ $\bar{v}_6 \times 1 = 1$ $\bar{v}_4 \times v_0 = 1$ $\bar{v}_6 \times 1 = 1$ $\bar{v}_5 \times 1 = 1$
$y = v_6$	10.9093	$\bar{v}_6 = \bar{y}$	1

Designed to be simple in software

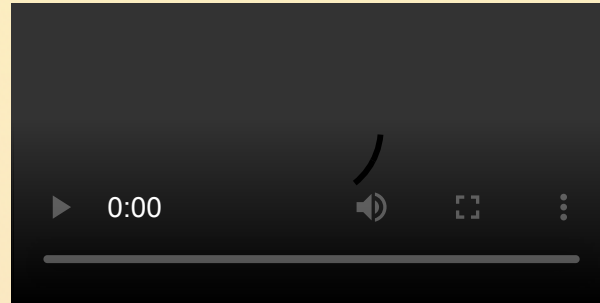
```
import torch, math
x0 = torch.tensor(1., requires_grad=True)
x1 = torch.tensor(2., requires_grad=True)
p = 2*x0 + x0*torch.sin(x1) + x1**3
print(p)
p.backward()
print(x0.grad, x1.grad)
```

yielding

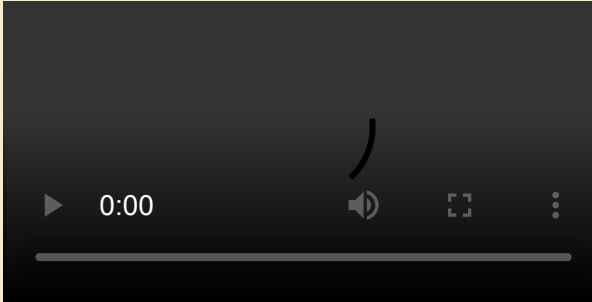
```
Primal: tensor(10.9093, grad_fn=<AddBackward0>)
Adjoint: tensor(2.9093) tensor(11.5839)
```

- Torch (and similar software) will correctly differentiate **only when the atomic operations are supported within it**
 - Common operations are overloaded (`__mul__` rewritten by `torch._mult_`)
 - Operations from libraries (`math.sin()`) must be replaced by their differentiation-aware equivalents (`torch.sin()`)

Impressive results



Impressive results



Differentiable Programming

Execute **differentiable functions (programs)** via **automatic differentiation**



Yann LeCun ✓

January 5, 2018 · 🌐

OK, Deep Learning has outlived its usefulness as a buzz-phrase.

Deep Learning est mort. Vive Differentiable Programming!

Yeah, Differentiable Programming is little more than a **rebranding** of the modern collection Deep Learning techniques, the same way Deep Learning was a rebranding of the modern incarnations of neural nets with more than two layers.

But the important point is that people are now **building a new kind of software** by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization.

An increasingly large number of people are defining the networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the **input data fed to** them. It's really very much like a regular program, except it's parameterized, **automatically differentiated**, and trainable/optimizable. Dynamic networks have **become** increasingly popular (particularly for NLP), thanks to deep learning frameworks that can handle them such as PyTorch and Chainer (note: our old deep learning framework Lush could handle a particular kind of dynamic nets called Graph Transformer Networks, back in 1994. It was needed for text recognition).

People are now actively working on compilers for **imperative differentiable programming languages**. This is a very exciting avenue for the development of learning-based AI.

Important note: this won't be sufficient to take us to "true" AI. Other concepts will be needed for that, such as what I used to call predictive learning and now decided to call Imputative Learning. More on this later...

👍 1.8K

186 Comments 464 Shares

How neural networks behave

Universal approximation theorem

Given a family of neural networks, for each function f from a certain function space, there exists a sequence of neural networks ϕ_1, ϕ_2, \dots from the family, such that $\phi_n \rightarrow f$ according to some criterion. That is, the family of neural networks is dense in the function space.

- No prescription on how to find the sequence
- No guarantee that any specific method can find the sequence at all
- No guarantee that any finite network size is enough (e.g. "10000 neurons is enough")

Universal approximation theorem: width

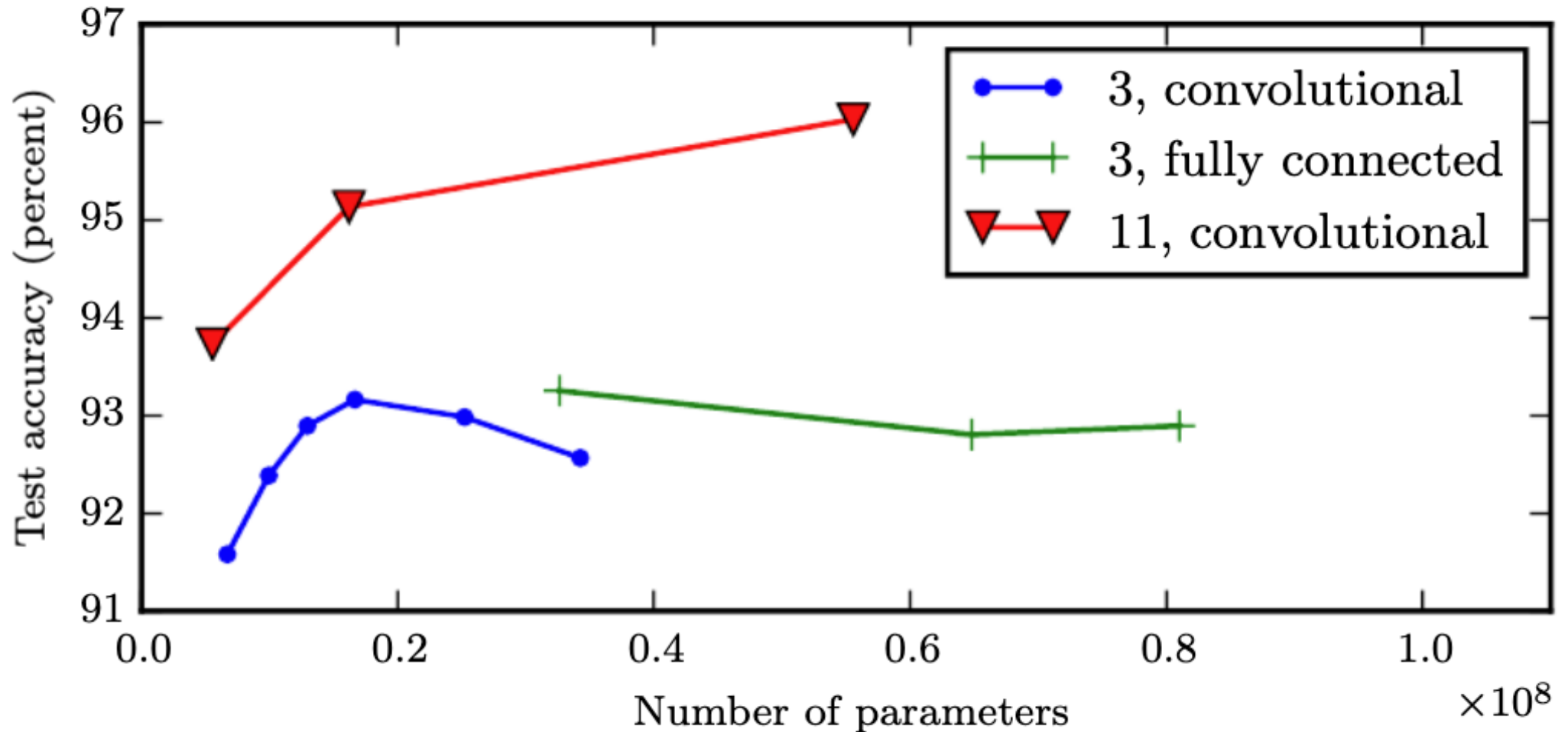
- A feed-forward network with sigmoid activation functions can approximate any continuous real-valued function
 - Cybenko, G. (1989)
- Any failure in mapping a function comes from inadequate choice of weights or insufficient number of neurons
 - Hornik et al (1989), Funahashi (1989)
- Derivatives can be approximated as well as the functions, even in case of non-differentiability (e.g. piecewise differentiable functions)
 - Hornik et al (1990)
- These results are valid even with other classes of activation functions
 - Light (1992), Stinchcombe and White (1989), Baldi (1991), Ito (1991), etc

Universal approximation theorem: depth

- Infinite number of finite-width layers approximate arbitrary functions, if activation function is twice-differentiable
 - Gripenberg (2016)
- Deep ReLU networks approximate smooth functions more efficiently than shallow networks
 - Yarotsky (2016), Lu et al. (2017)
- Hanin and Sellke (2017)
 - Minimal width to approximate continuous real-valued function to any precision: $d_{input} + 1$
 - Any continuous function can be approximated by a deep ReLU network of minimal width $d_{input} + d_{output}$
 - With skip connections, a network of width **1** and infinite layers is a universal approximator
- Kidger and Lyons (2019) extended these results to any activation function
 - Generalizes for more than one output neuron
 - The bound is now $d_{input} + d_{output} + 2$
 - For most commonly used activations function, the bound is actually $d_{input} + d_{output} + 2$

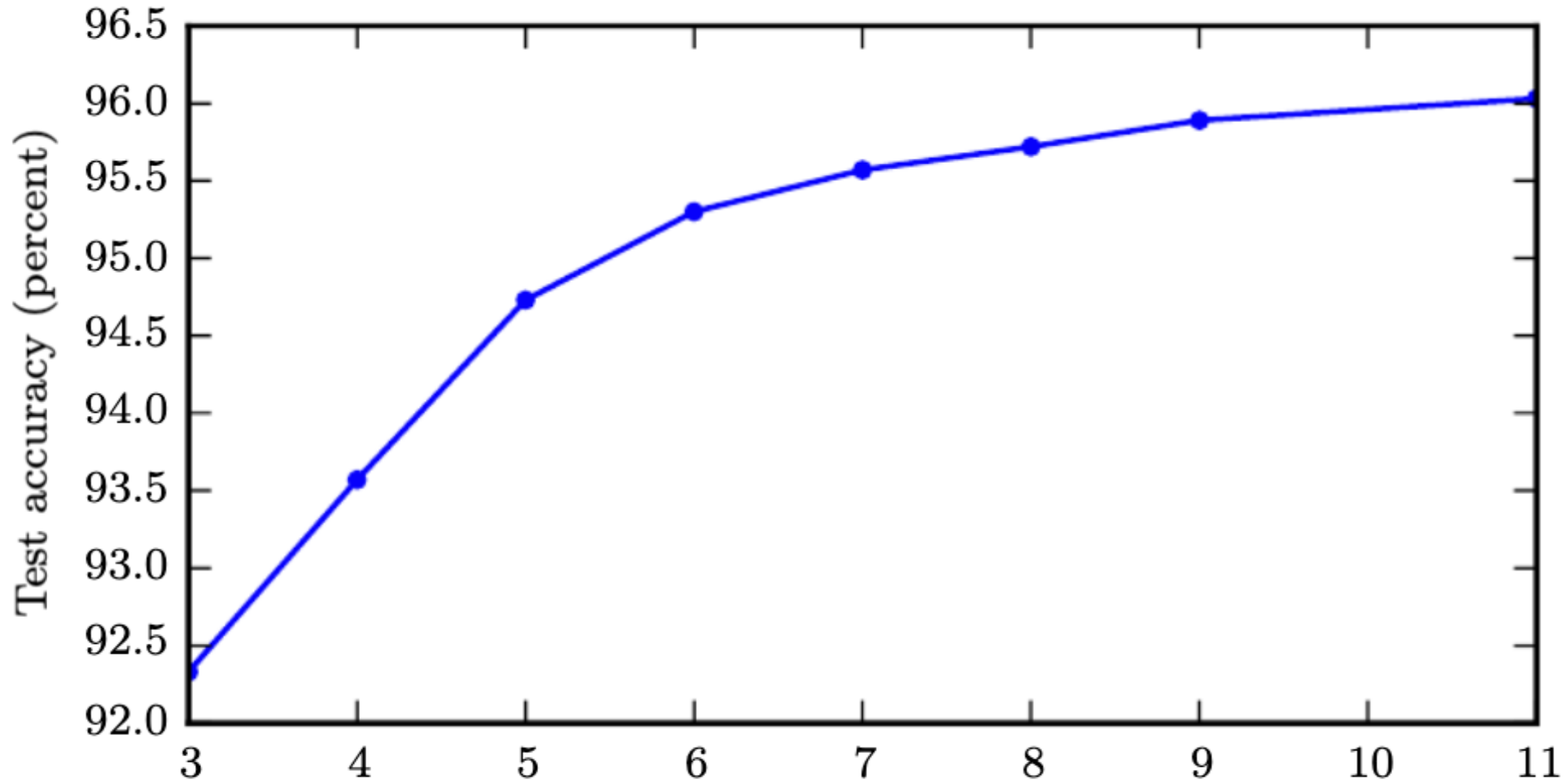
Number of parameters

- Empirical studies: increasing number of parameters doesn't help beyond a certain point



Depth

- Empirical studies: increasing depth tends to always result in some improvement



Regularization

- Regularization: any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.
 - Restriction to parameter values
 - Extra terms in loss function (indirect constraint on parameter values)
- Sometimes, constraints encode prior knowledge: [inductive bias](#)
 - For instance, CP symmetry enforced in neural networks: [2405.13524 accepted by PRD](#)
- **CAVEAT:** regularization works by bias-variance trade-off

Bias-Variance tradeoff of MSE

$$\mathbb{E}_{D,\epsilon} \left[(y - \hat{f}(x; D))^2 \right] = \left(\text{Bias}_D [\hat{f}(x; D)] \right)^2 + \text{Var}_D [\hat{f}(x; D)] + \sigma^2$$

where x is a data set unseen during training (test data set)

- Error caused by simplifying assumptions in the method:

$$\text{Bias}_D [\hat{f}(x; D)] := \mathbb{E}_D [\hat{f}(x; D) - f(x)] = \mathbb{E}_D [\hat{f}(x; D)] - \mathbb{E}_{y|x} [y(x)]$$

- Variance of the method (how much it moves around its mean):

$$\text{Var}_D [\hat{f}(x; D)] := \mathbb{E}_D \left[\left(\mathbb{E}_D [\hat{f}(x; D)] - \hat{f}(x; D) \right)^2 \right]$$

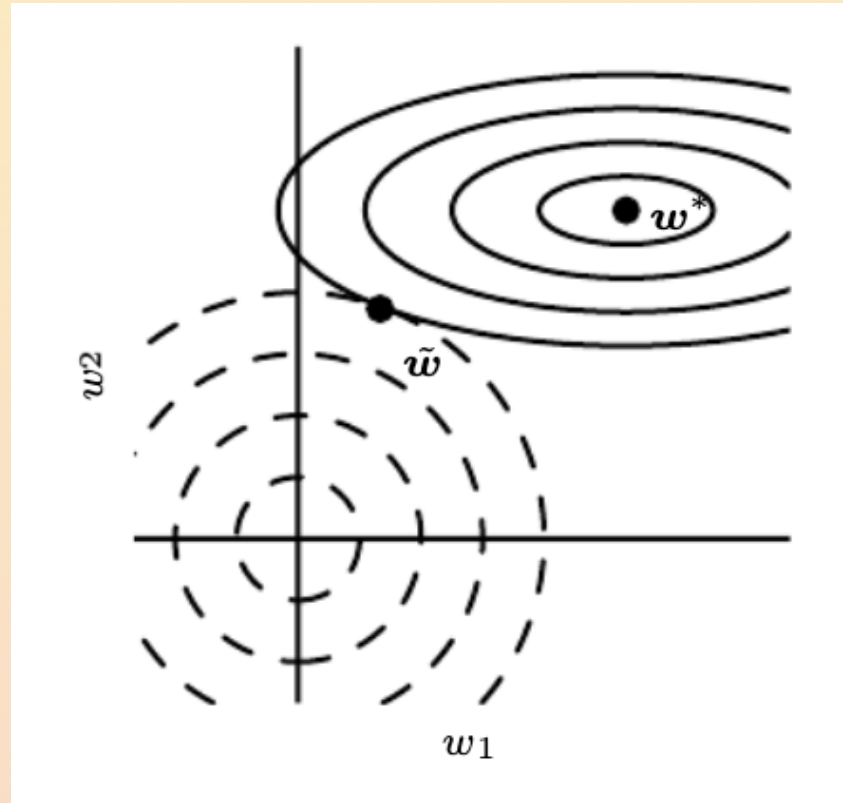
- Optimization error (irreducible):

$$\sigma^2 = \mathbb{E}_y \left[\left(y - \underbrace{f(x)}_{\mathbb{E}_{y|x}[y]} \right)^2 \right]$$

Regularization: weight decay

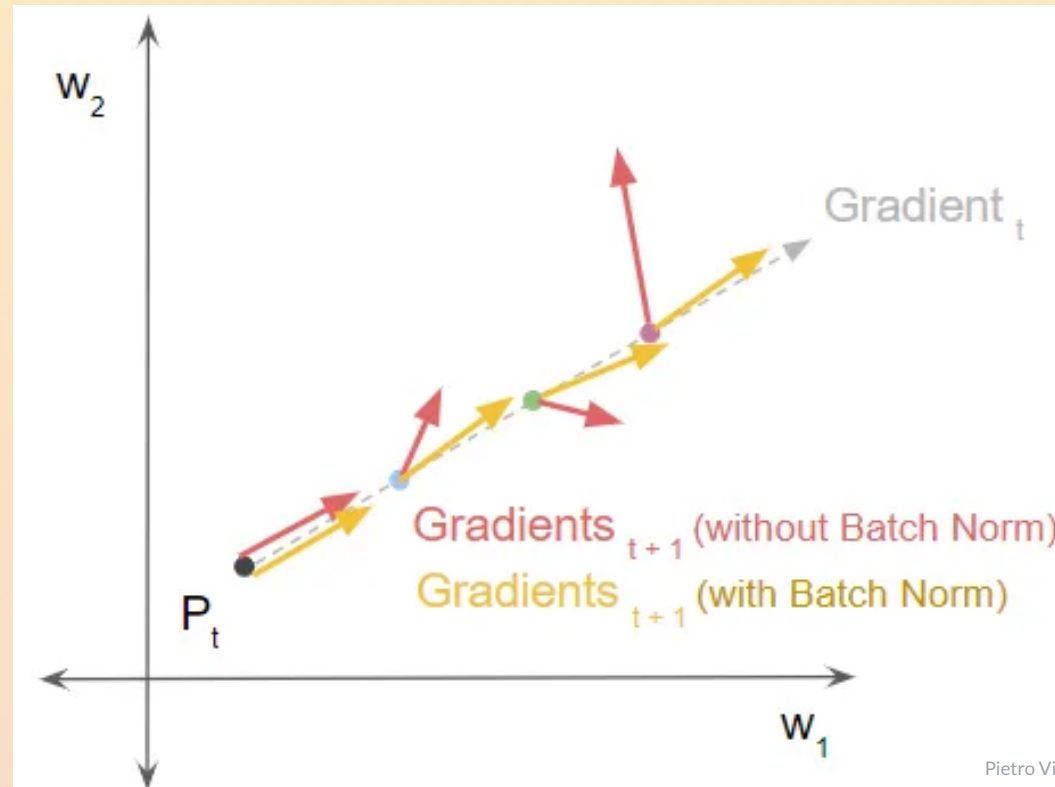
- Tradeoff between good fitting (small MSE) and small norm (smaller slope, or fewer features with large weights)
- In Mathematics, L^2 regularization; in statistics, "ridge regression", "Tikhonov regularization"

$$J(\mathbf{w}) = MSE_{train} + \lambda \mathbf{w}^T \mathbf{w}$$



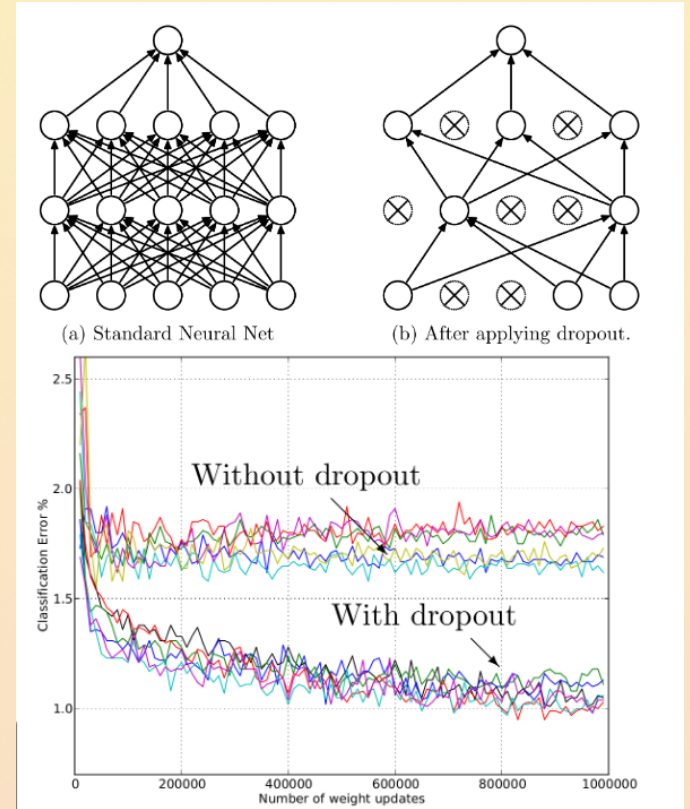
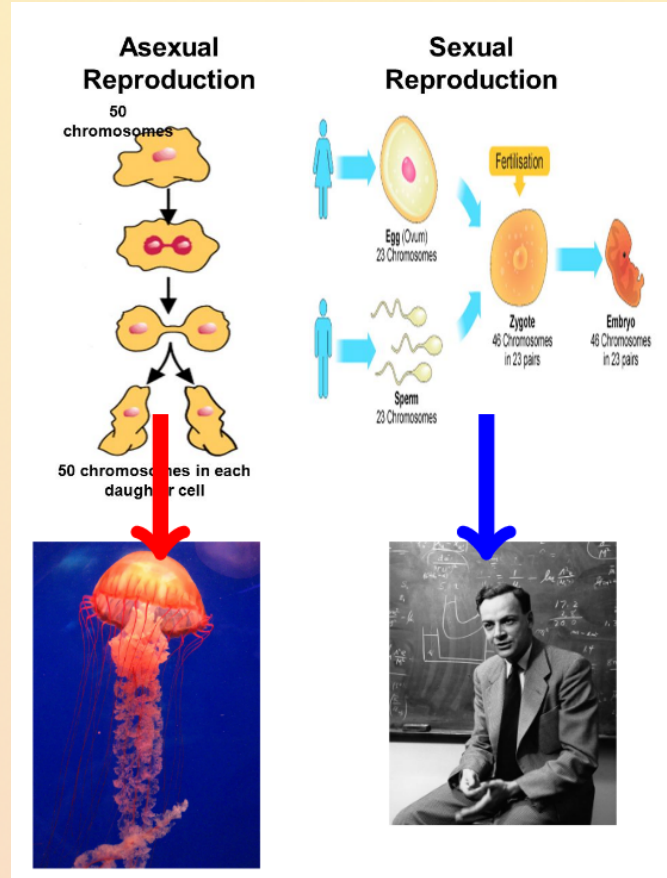
Regularization: batch normalization

- Standardize (transform by $(x - \bar{x})/\text{var}(x)$) each input coming from previous layer over the minibatch
 - Done for mini-batch, for batch training it would be too costly
 - Stabilizes response and reduces dependence among layers
 - Reduces also dependence on initial weight values
 - Works badly for small batch sizes (too much noise)
 - Cannot be used for recurrent networks (distributions at each timestep are different)



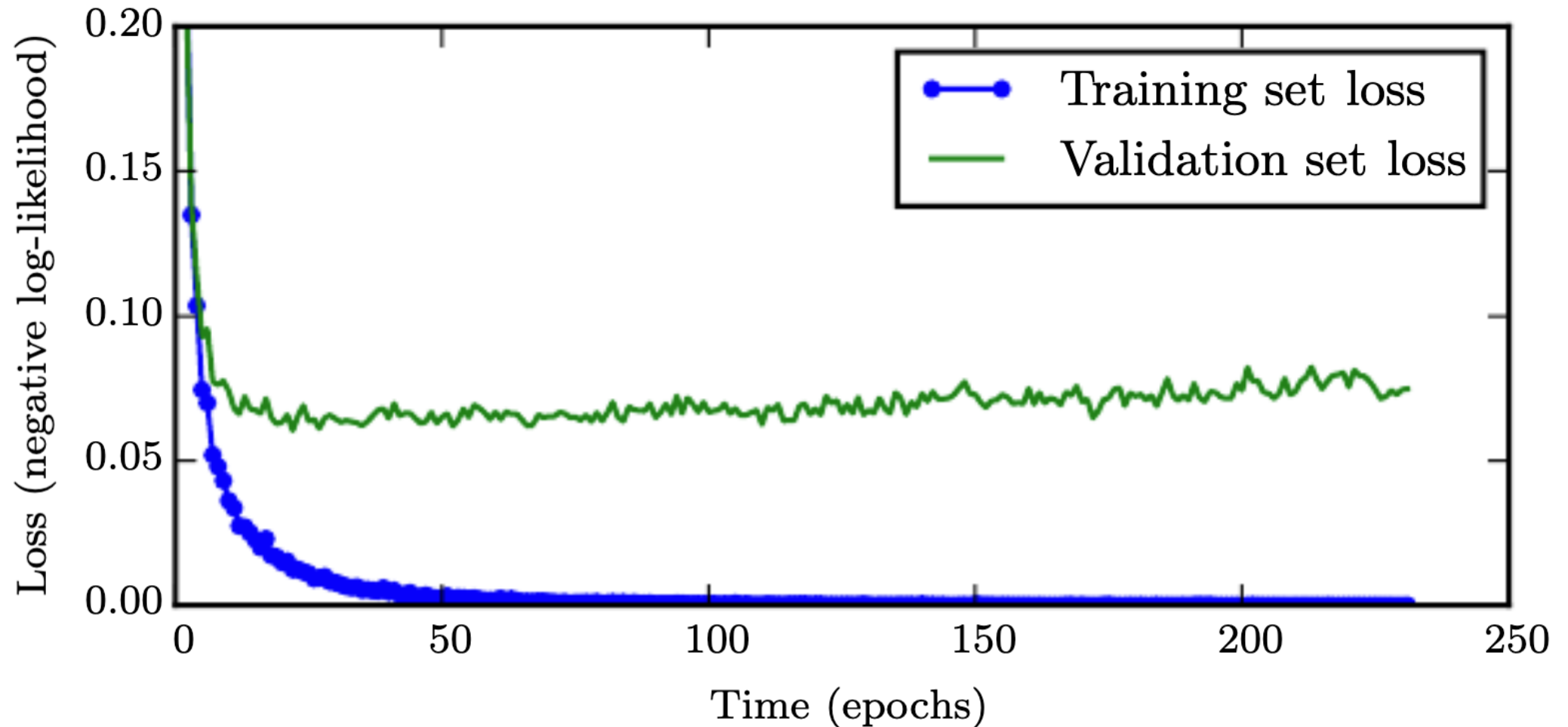
Regularization: dropout

- Randomly shut down nodes in training
 - Avoids a weight to acquire too much importance
 - Inspired in genetics



Early stopping...

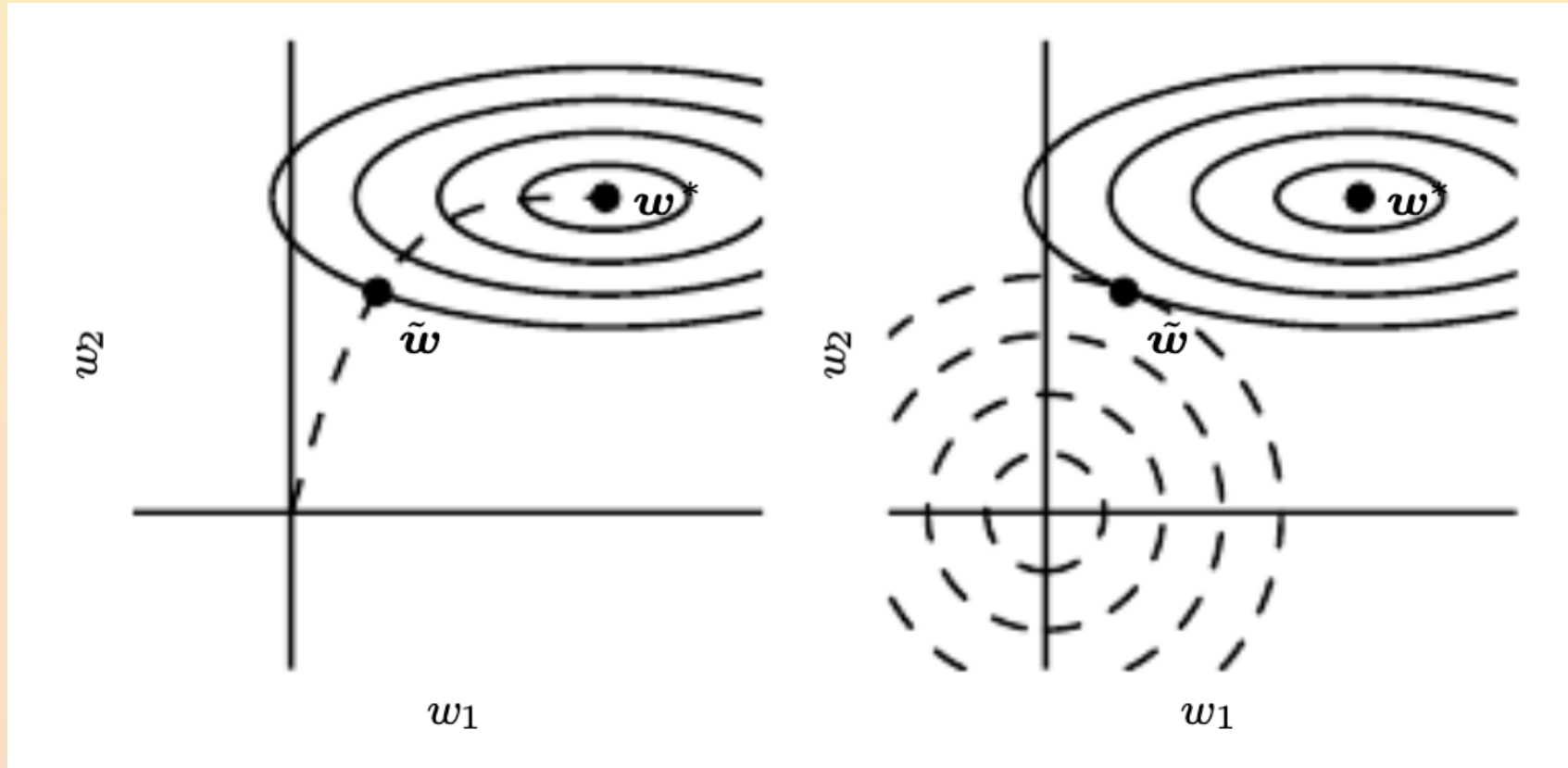
- Train until the validation set loss starts increasing, and pick the model corresponding to the minimum validation loss



...is a form of regularisation

- Early stopping limits the reachable phase space, and is therefore analogous to L2 regularization (weight decay)
 - Bishop (1995a) and Sjöberg and Ljung (1995)

$$J(\mathbf{w}) = MSE_{train} + \lambda \mathbf{w}^T \mathbf{w}$$



Network structure

and inductive bias

Convolutional Neural Networks (CNNs)

- They target spatial data (e.g. images)
 - Whenever the elements of the data vector can be seen as spatially structured
 - They account for strong correlations in the elements of the data vector (e.g. if pixel (24, 24) is white then all pixels around likely have bright colour)
- Image classification, object detection, segmentation, etc.
- Activation and loss functions mostly the same as dense networks
 - Except for more specialized tasks (e.g. style mixing)
- Special layer structure
 - Convolutional Layer
 - Pooling Layer
 - Fully Connected (FC) Layer



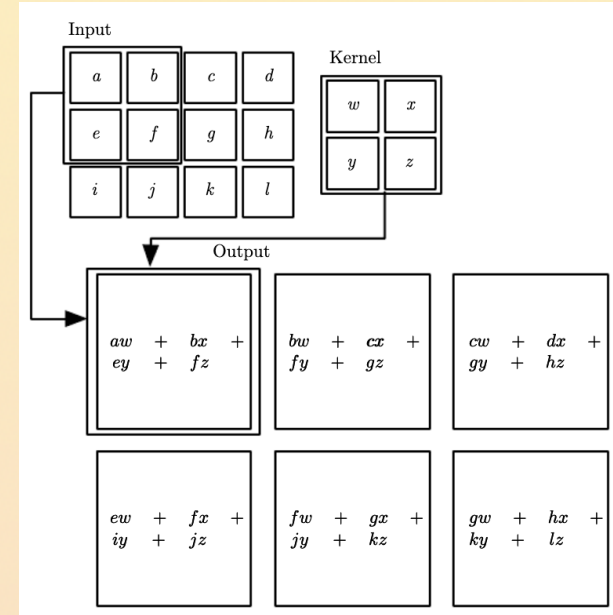
Convolutional Layer

- Convolution is a form of averaging
 - Detect local patterns (edges, textures, shapes...)

$$s(t) = \int x(a)w(t - a)da$$

- When discretized, integral becomes a sum
 - x input
 - w kernel: specifies how far does the averaging goes
 - s feature map
- For images, e.g., with $k_H \times k_W$ pixels and C colour channels,

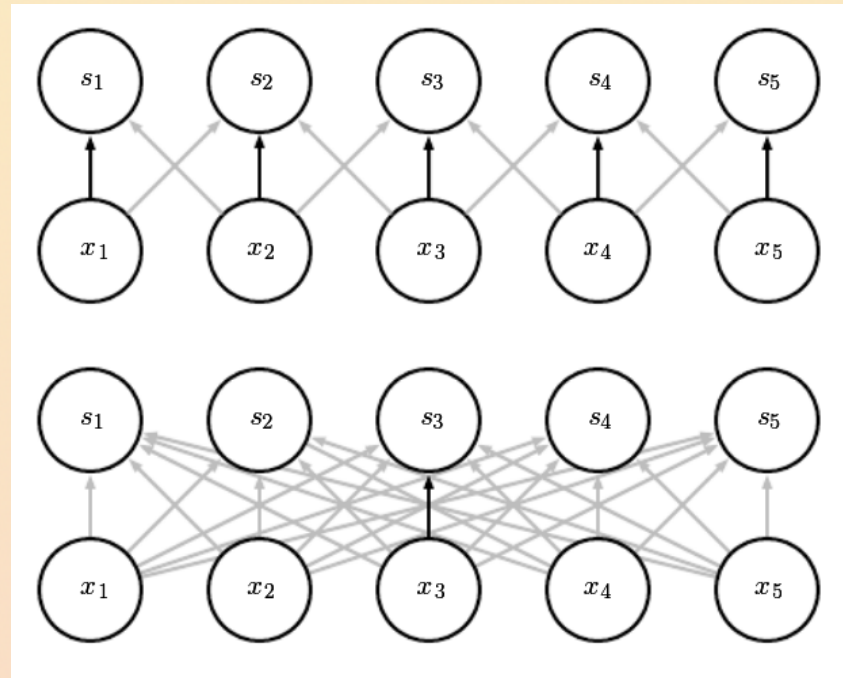
$$Y_{ij} = \sum_{m=0}^{k_H-1} \sum_{n=0}^{k_W-1} \sum_{c=0}^{C-1} X_{(i+m)(j+n)c} \cdot K_{mnc}$$



- The output has size $\frac{H+2P-k_H}{S} + 1 \times \frac{W+2P-k_W}{S} + 1$
 - Stride (S): step size for shifting the kernel around
 - Padding (P): add a border (or fold the image on itself) to maintain spatial dimensions

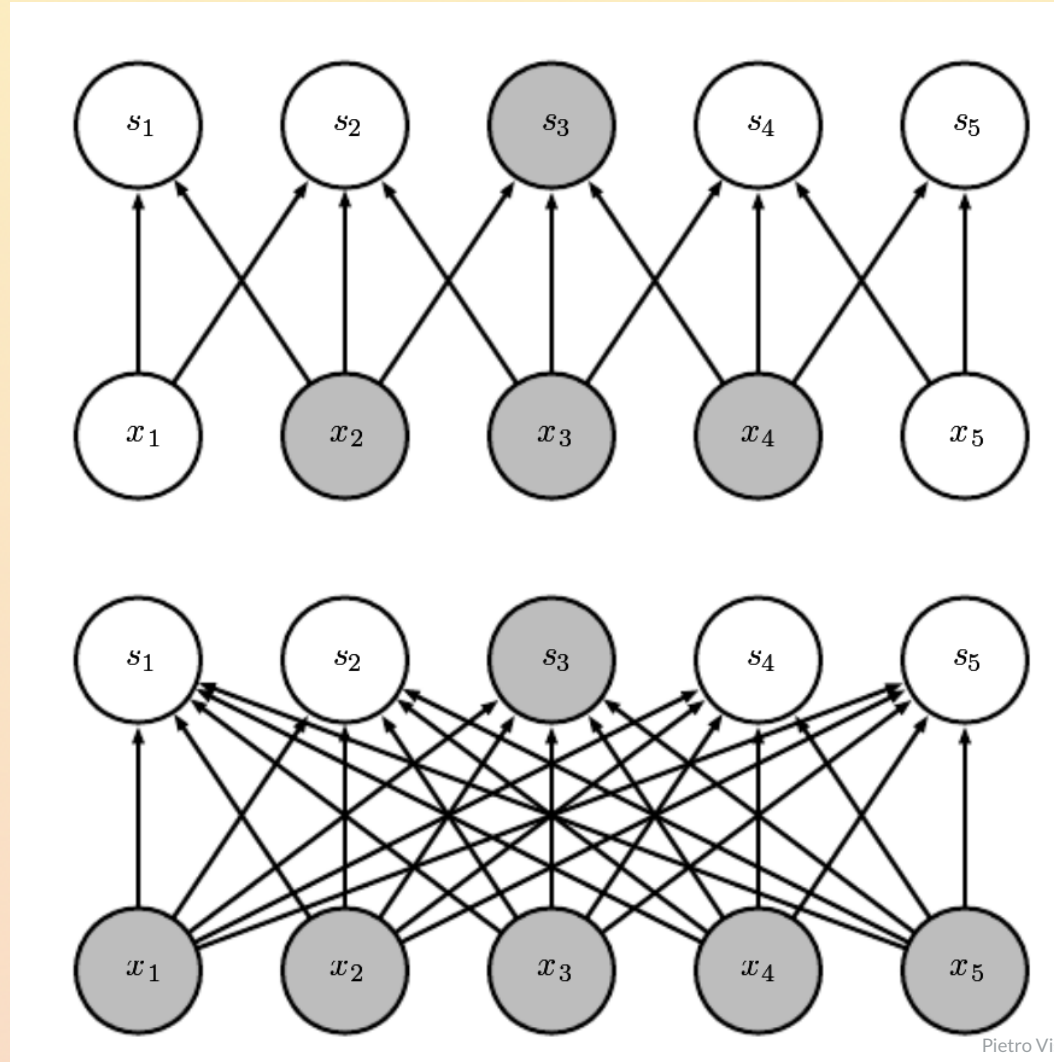
Parameter sharing

- Kernel smaller than the input \rightarrow sparse connectivity (sparse weights)
 - Not simple matrix multiplication anymore
 - Detect small features over a small number of pixels
- Simply require parameters are **equal**
 - If they are equal, you can store only one number in memory (sometimes dramatic memory footprint reduction)



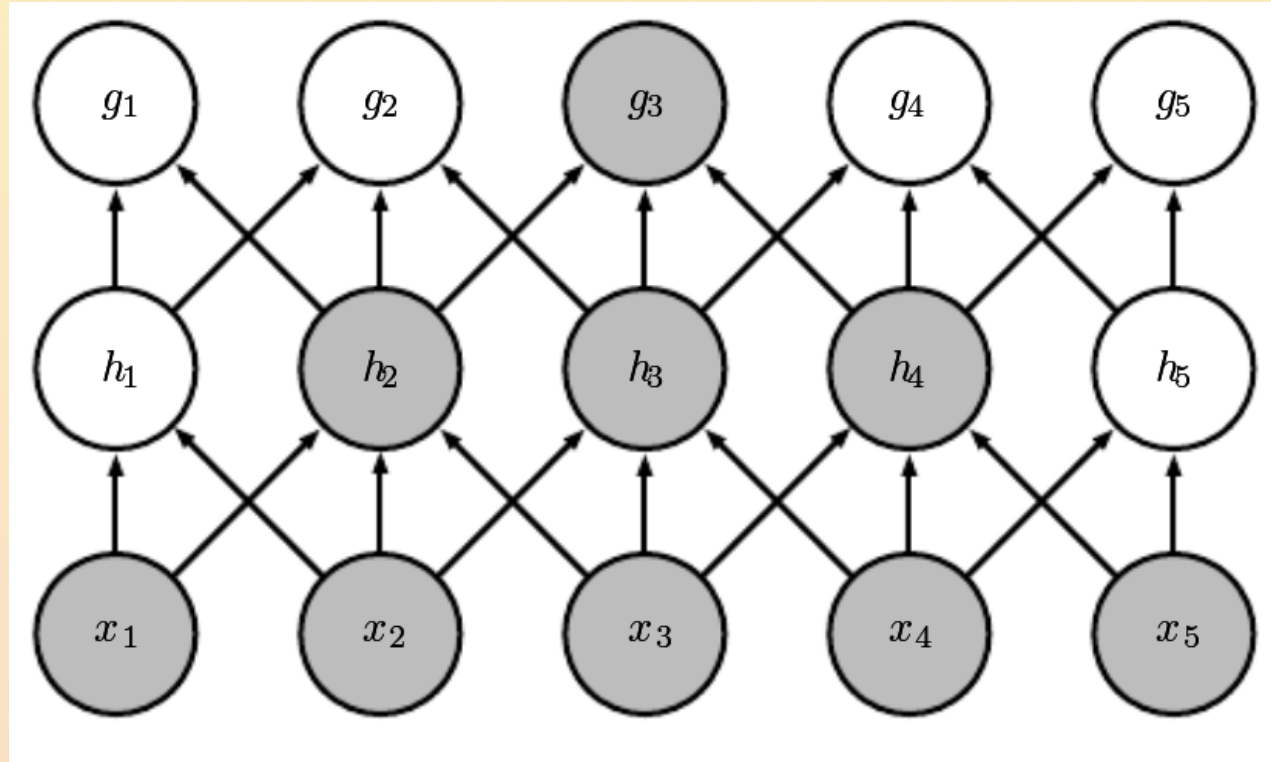
Sparse connectivity and receptive field

- Describe complicated interactions constructing them from simpler building blocks



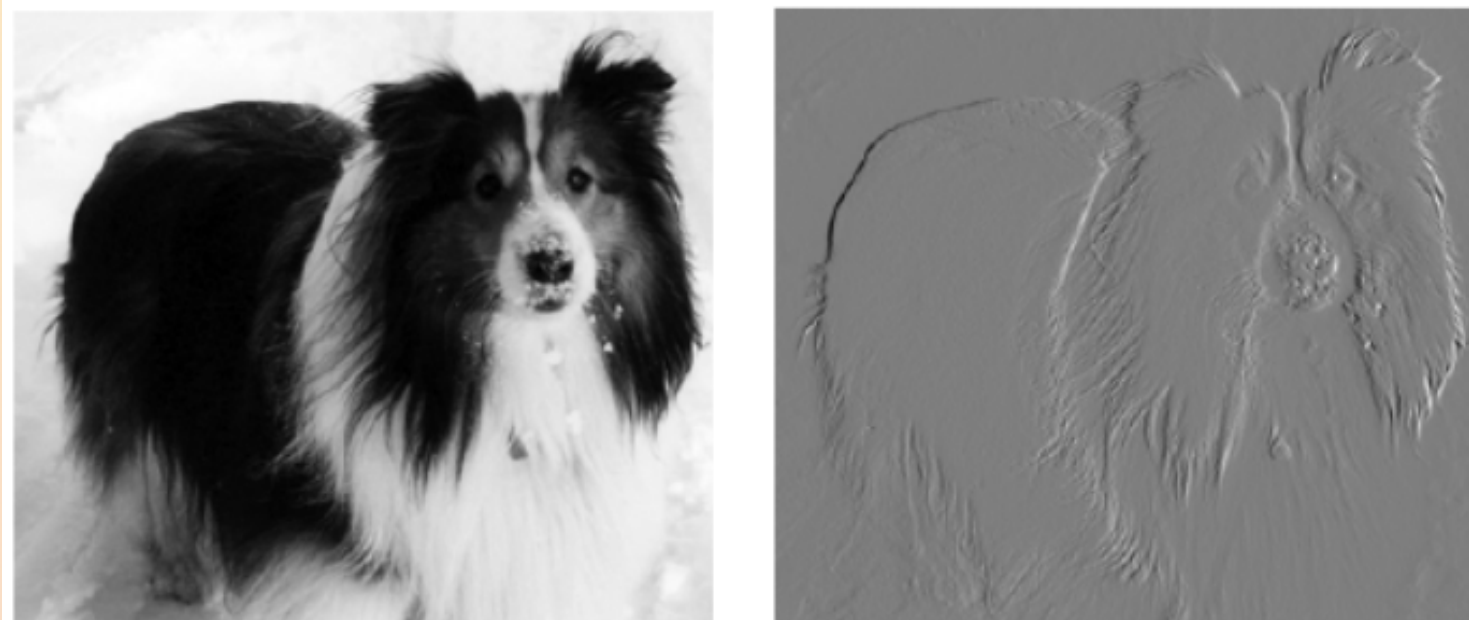
Receptive field: deeper = larger

- Direct connections are sparse, but indirect connections can extend to all (or most of) the input



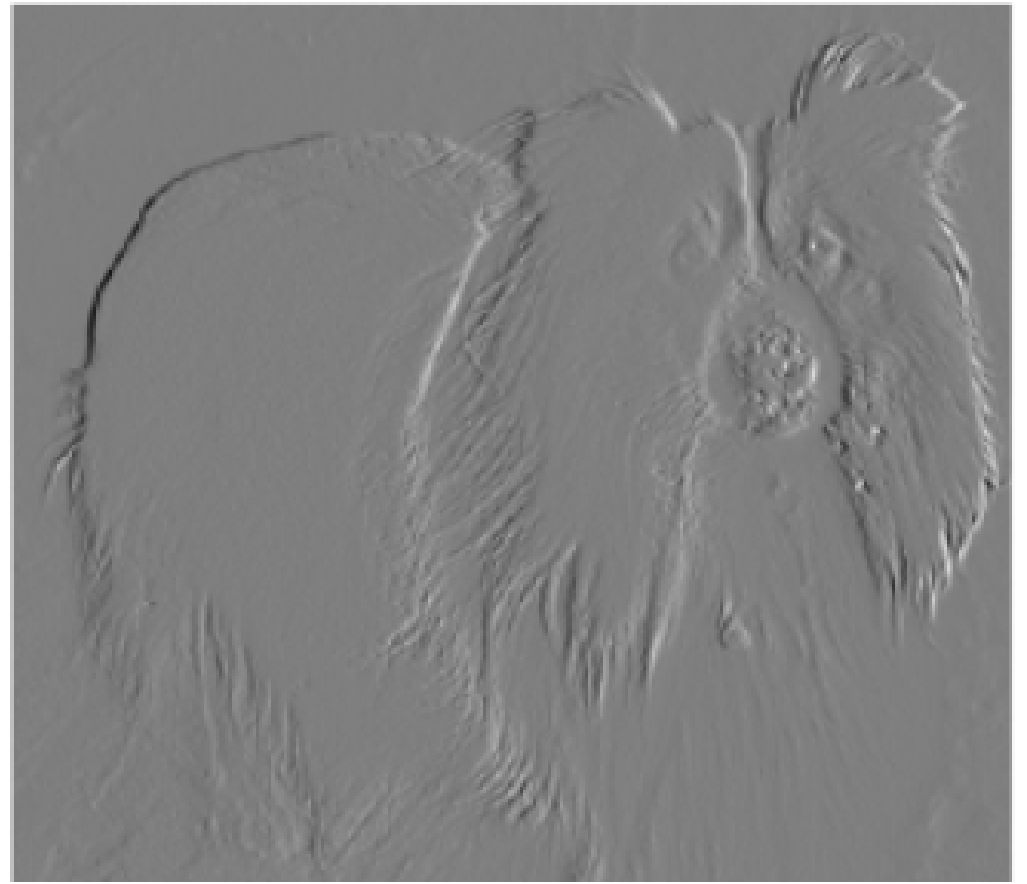
Computational advantage

- Take each pixel and subtract from it the value of the pixel to the left
 - Input: 320×280
 - Output: 319×280
- Implementation as convolution: $319 \times 280 \times 3 = 267960$ floating point operations
 - Two multiplications and one addition per pixel
- Implementation as matrix product: $320 \times 280 \times 319 \times 280 > 77998592000$ operations ($N_{weights} = N_{input} \times N_{output}$)



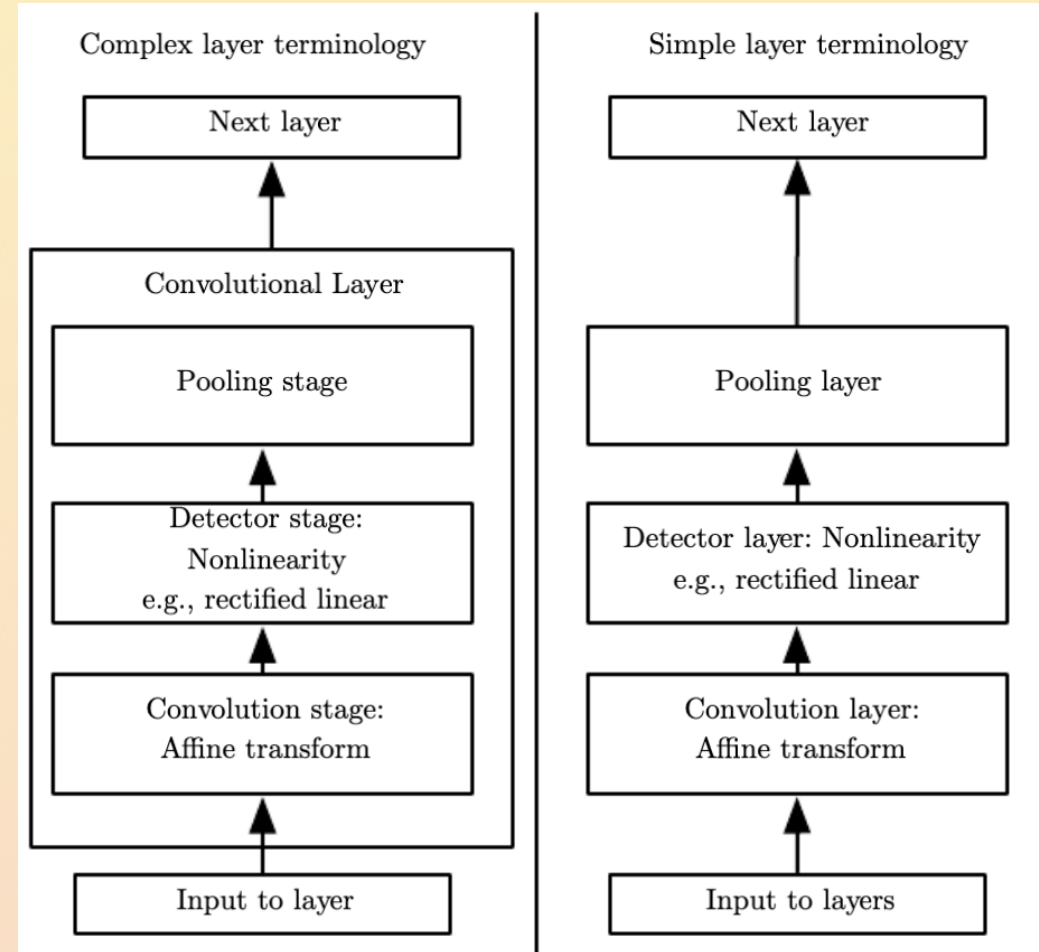
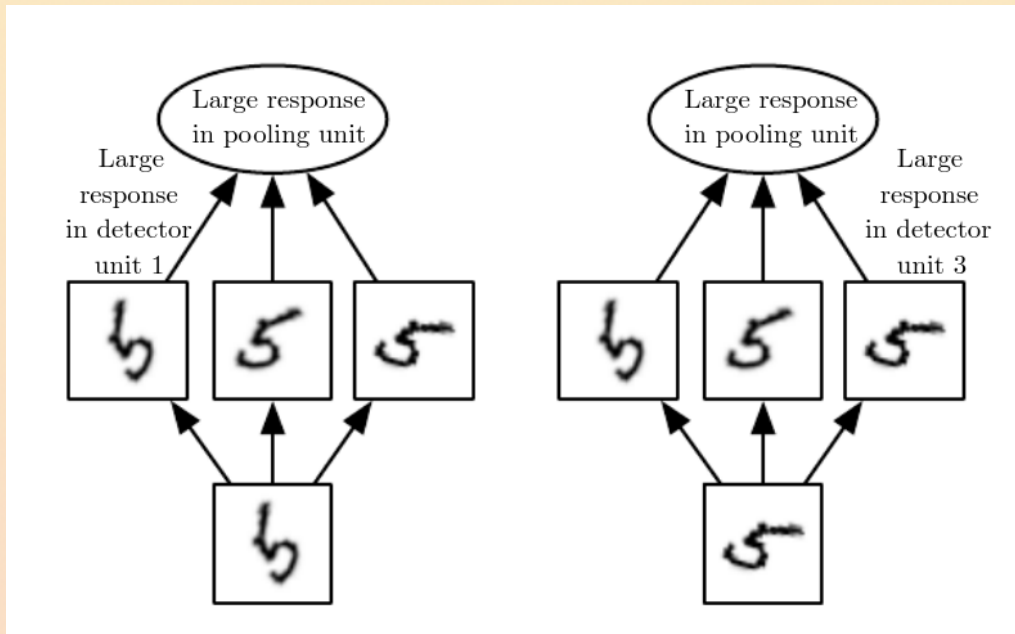
Equivariance

- This specific transformation, e.g., exhibits equivariance by translations
 - "The same object" moving to a different part of the input implies its representation moves by a same amount in the output (e.g. edges)



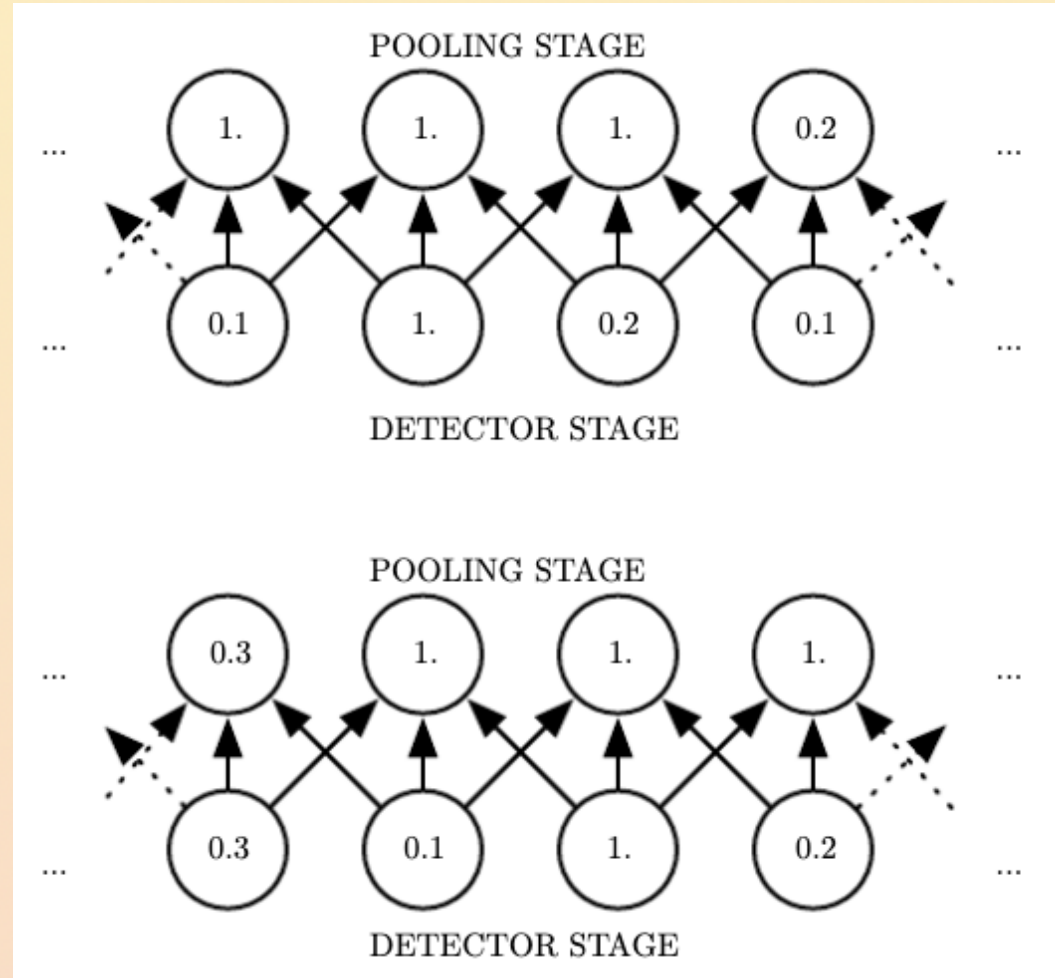
Pooling

- Convolution done in parallel → nonlinear activation → pooling
- Pooling: replace output at a location with a summary statistic
 - e.g., max pooling = report the maximum output in a neighbourhood
 - Helps with invariance for small variations (translations)

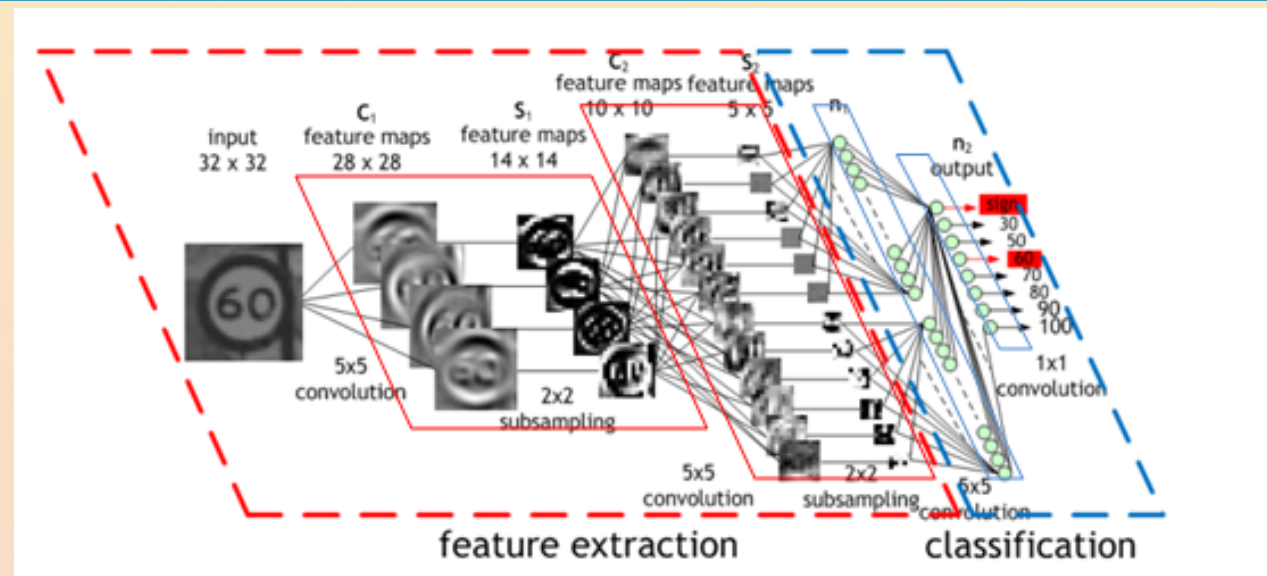


Pooling

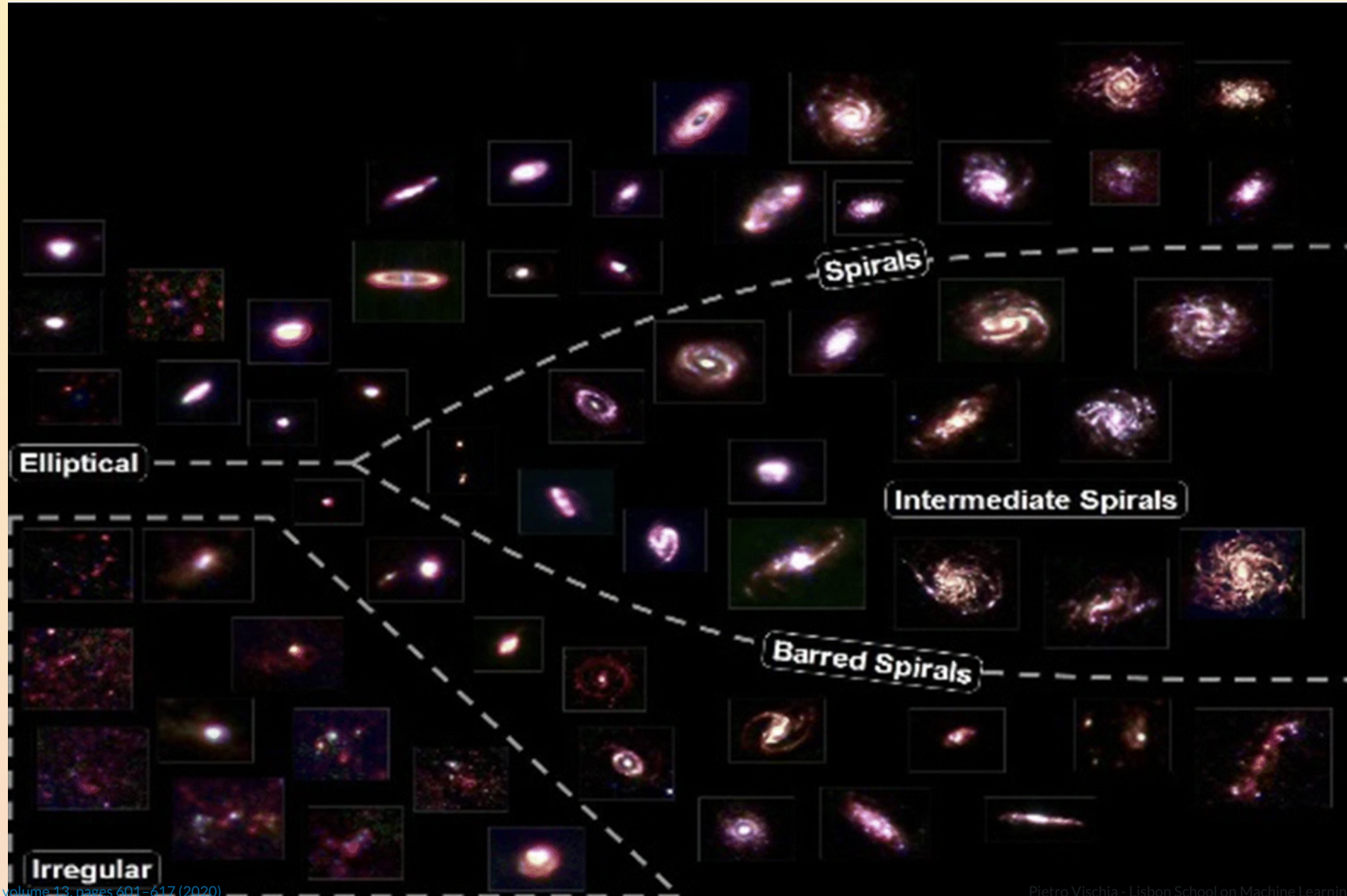
- Small translations of the input leave the output almost unchanged



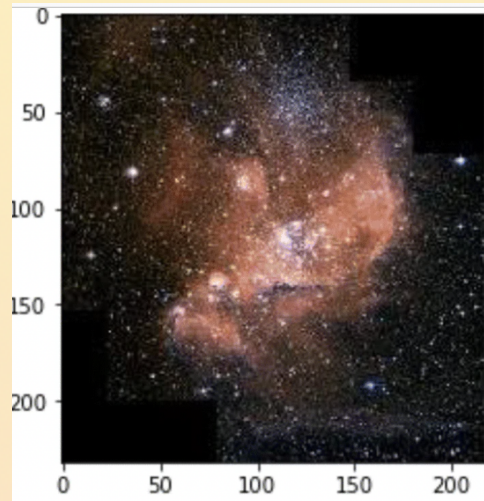
Convolutional networks



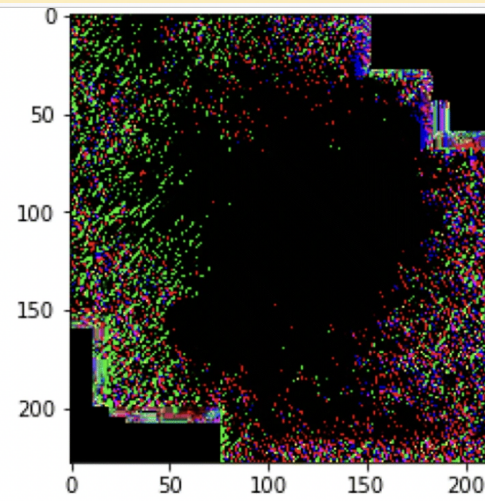
Morphology of galaxies



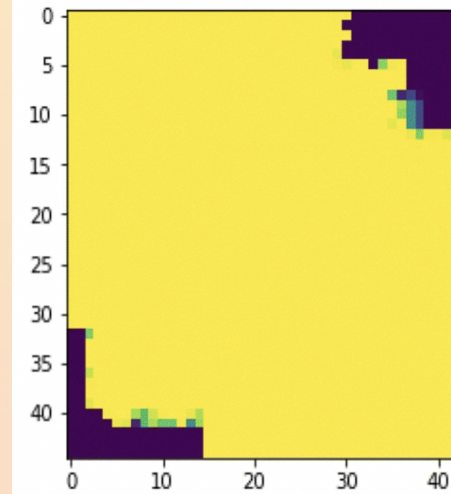
Representations of galaxies...



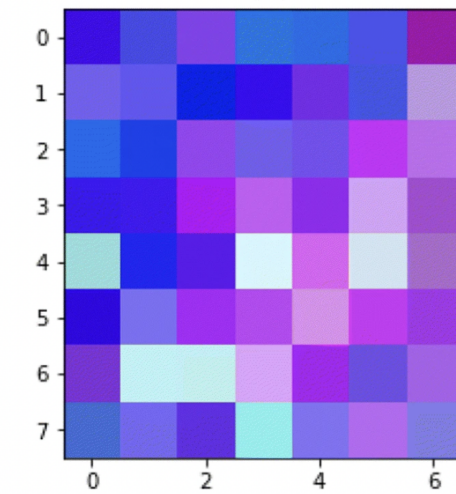
(a)



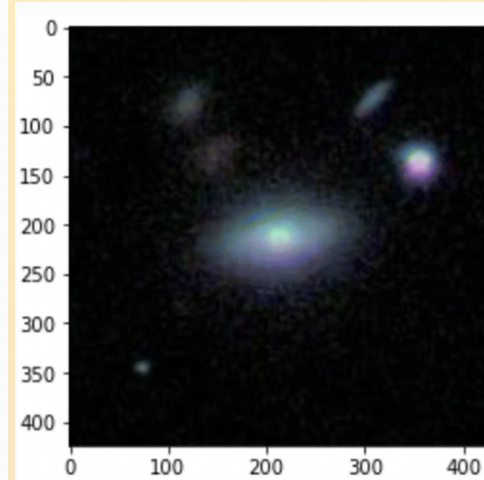
(b)



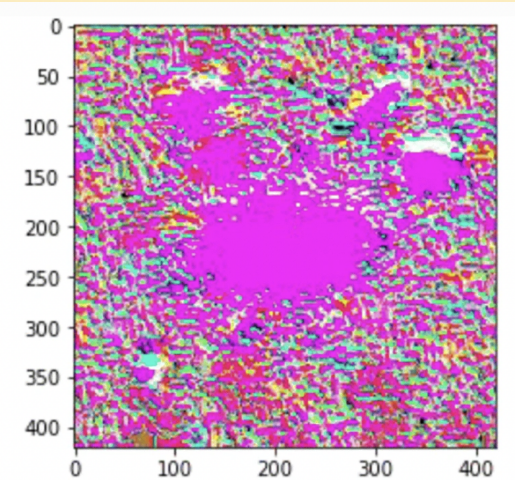
(c)



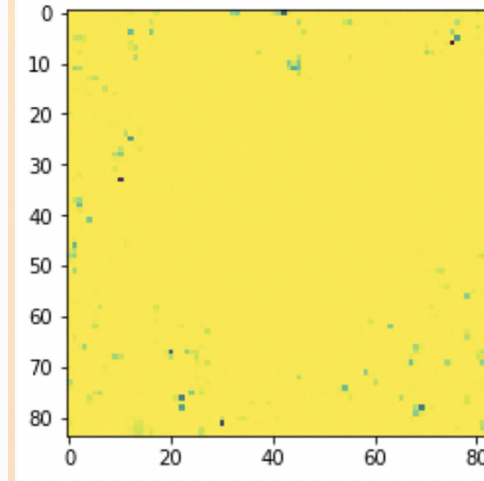
(d)



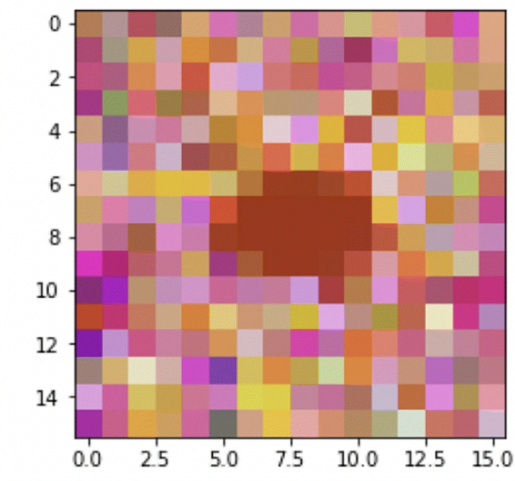
(a)



(b)

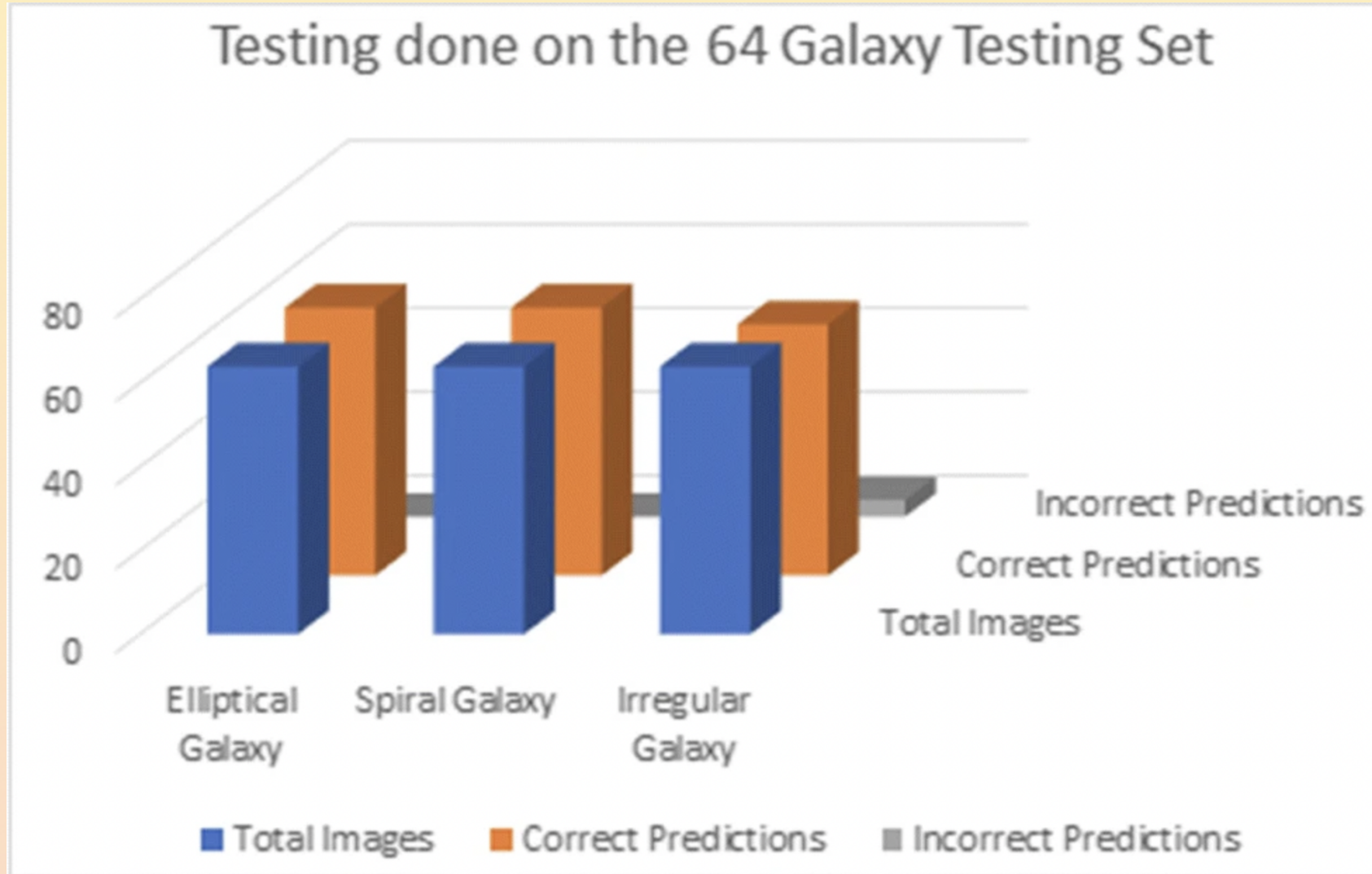


(c)



(d)

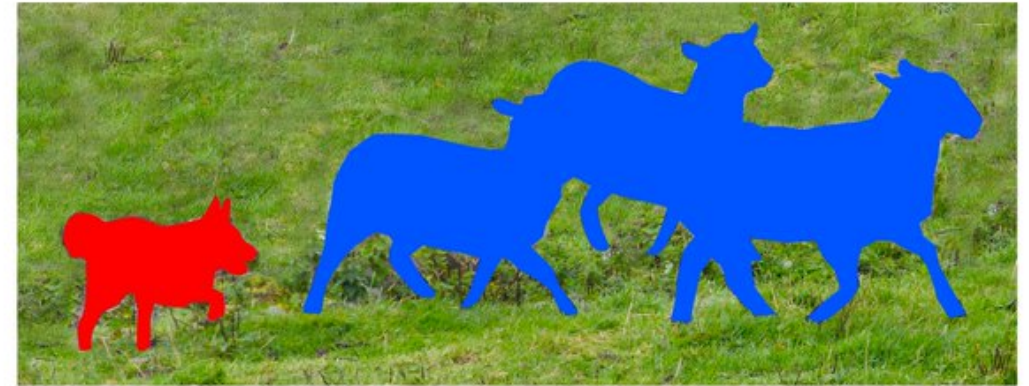
...work pretty well



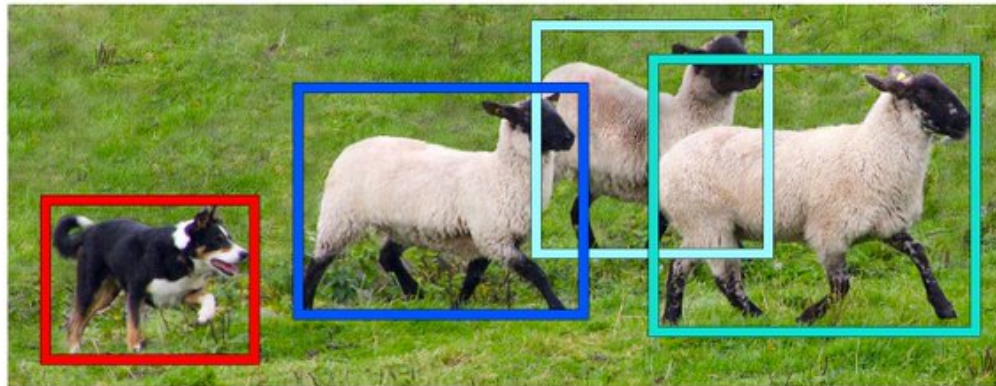
Semantic representations



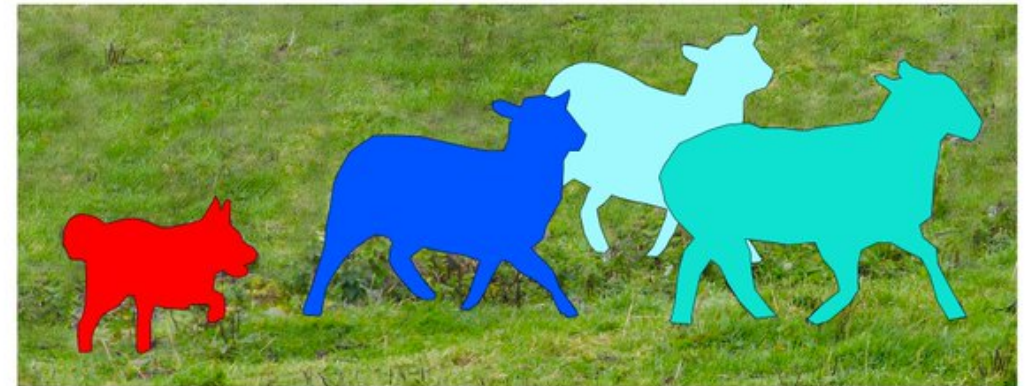
Image Recognition



Semantic Segmentation



Object Detection



Instance Segmentation

What about time?

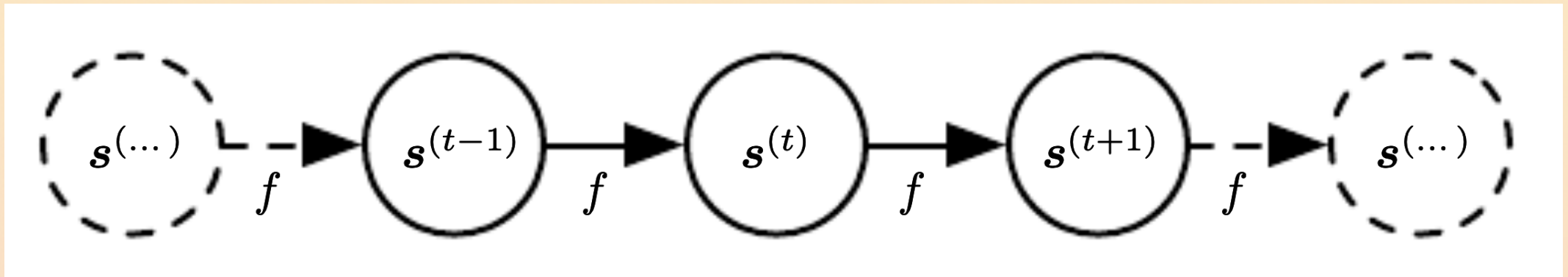
- Convolutional network: process grid of values (e.g. images)
- Recurrent networks: process a sequence of values indexed by a "time" component
 - Language is a sequence
- Parameter sharing crucial to generalize:
 - lengths unseen in training
 - different positions in the sentences
- Without parameter sharing, a network would have to learn all the language rules at each step of the sequence
 - Very impractical
- Both scale very well (thanks to parameter sharing)

Convolutional networks for sequences?

- Could "link" the steps of the sequence via the convolution
- Use the same kernel at each time step
- Shallow: it links only neighbouring time steps

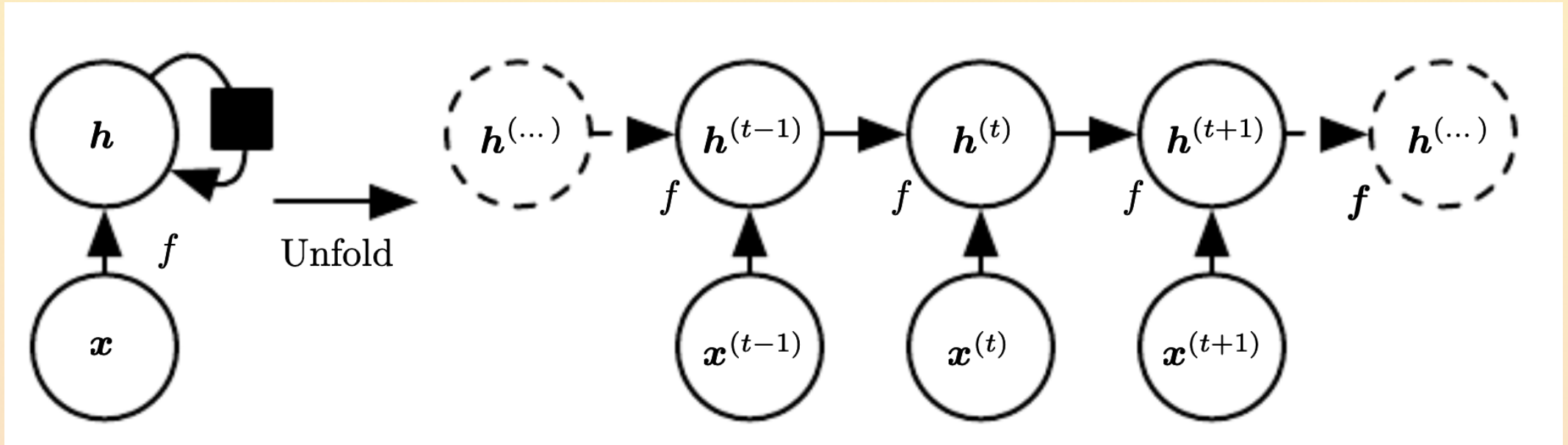
Recurrent network

- Use the same parameter at the same step, $s^{(t)} = f(s^{(t-1)}, \theta)$
 - Very deep structure



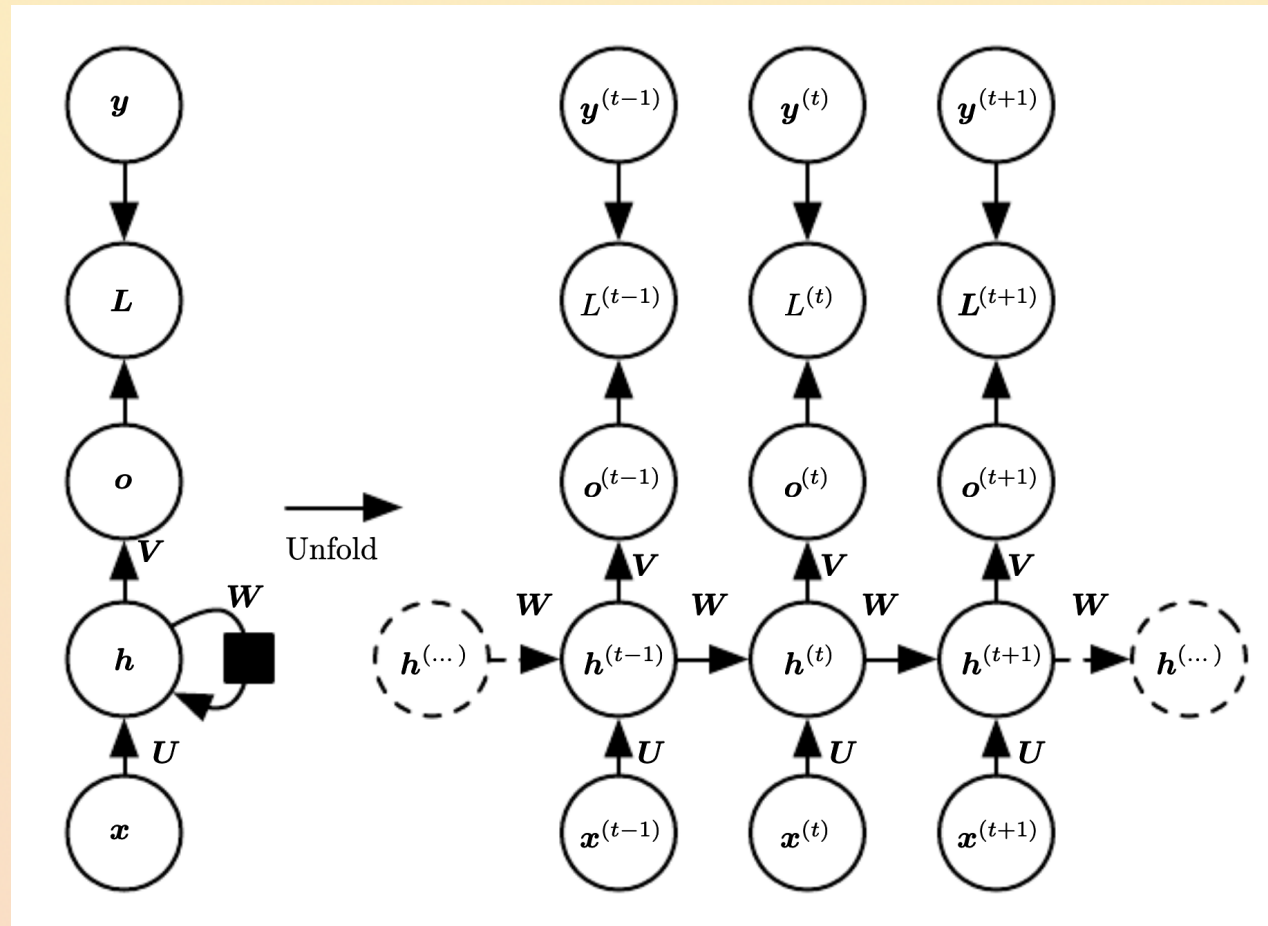
Unfold the graph

$$s^{(3)} = f(s^{(2)}, \theta) = f(f(s^{(1)}, \theta), \theta)$$



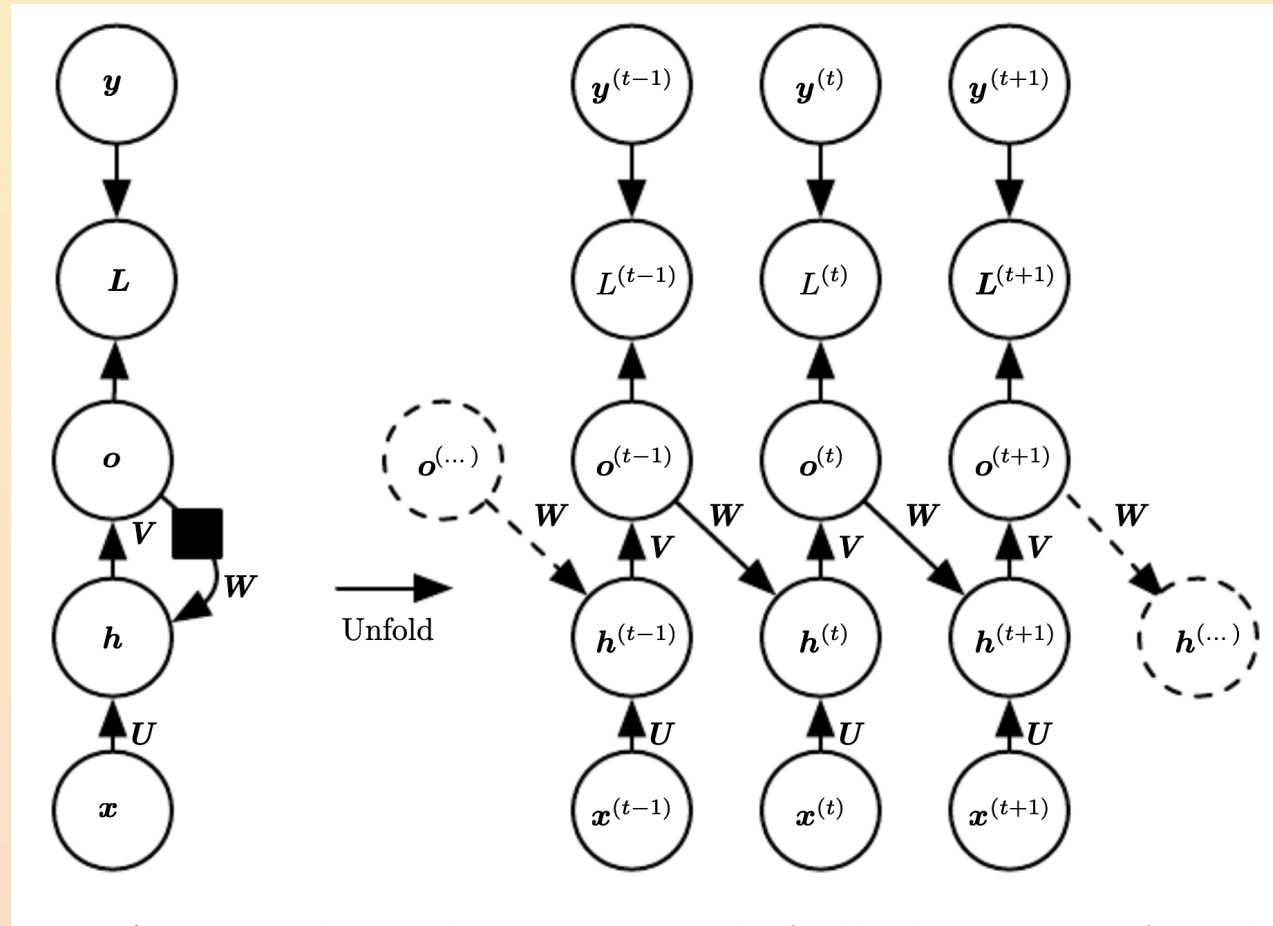
Vast zoology

- An output at each time step, recurrent connections between hidden units



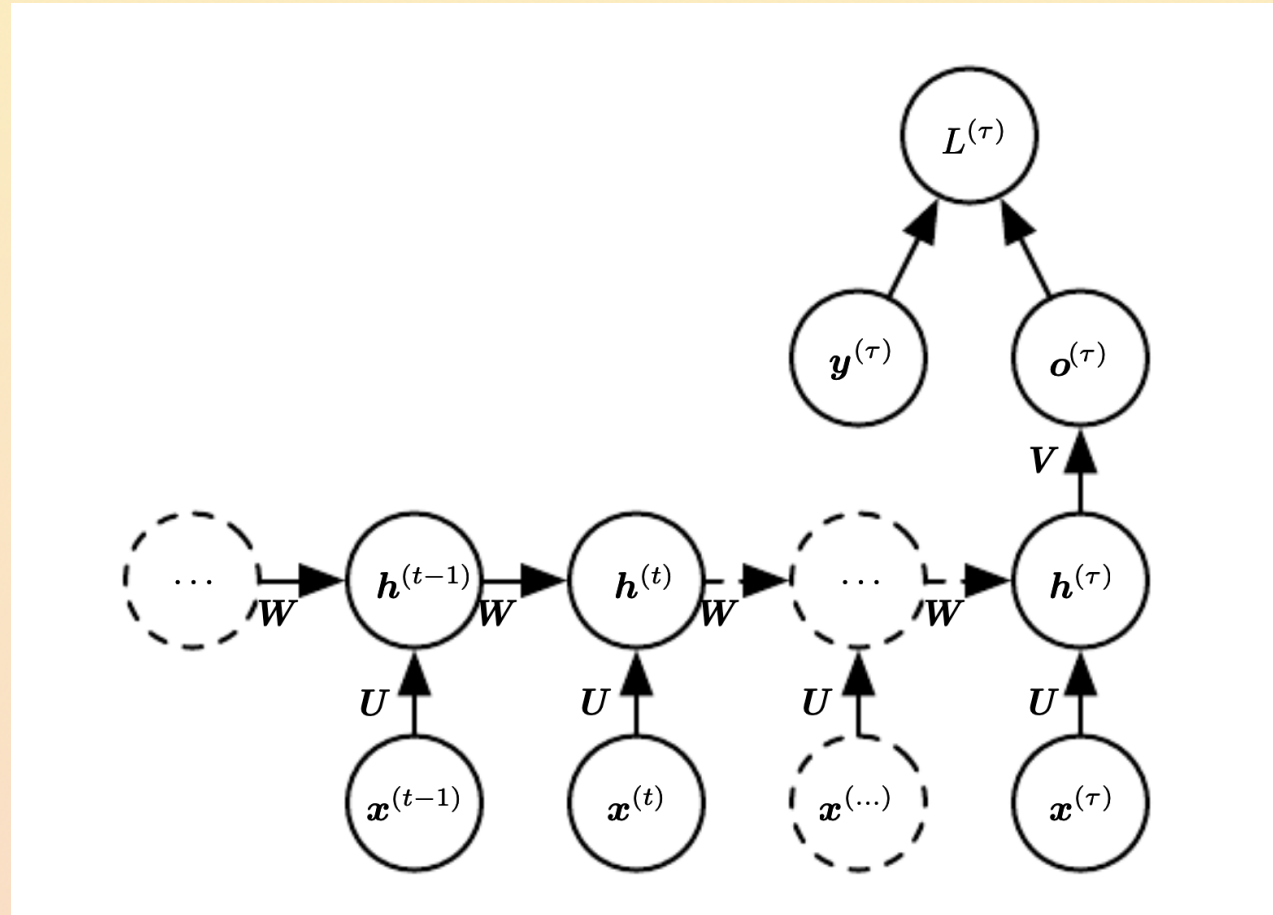
Vast zoology

- An output at each time step, recurrent connections only from the output at one time step to the hidden units at the next time step

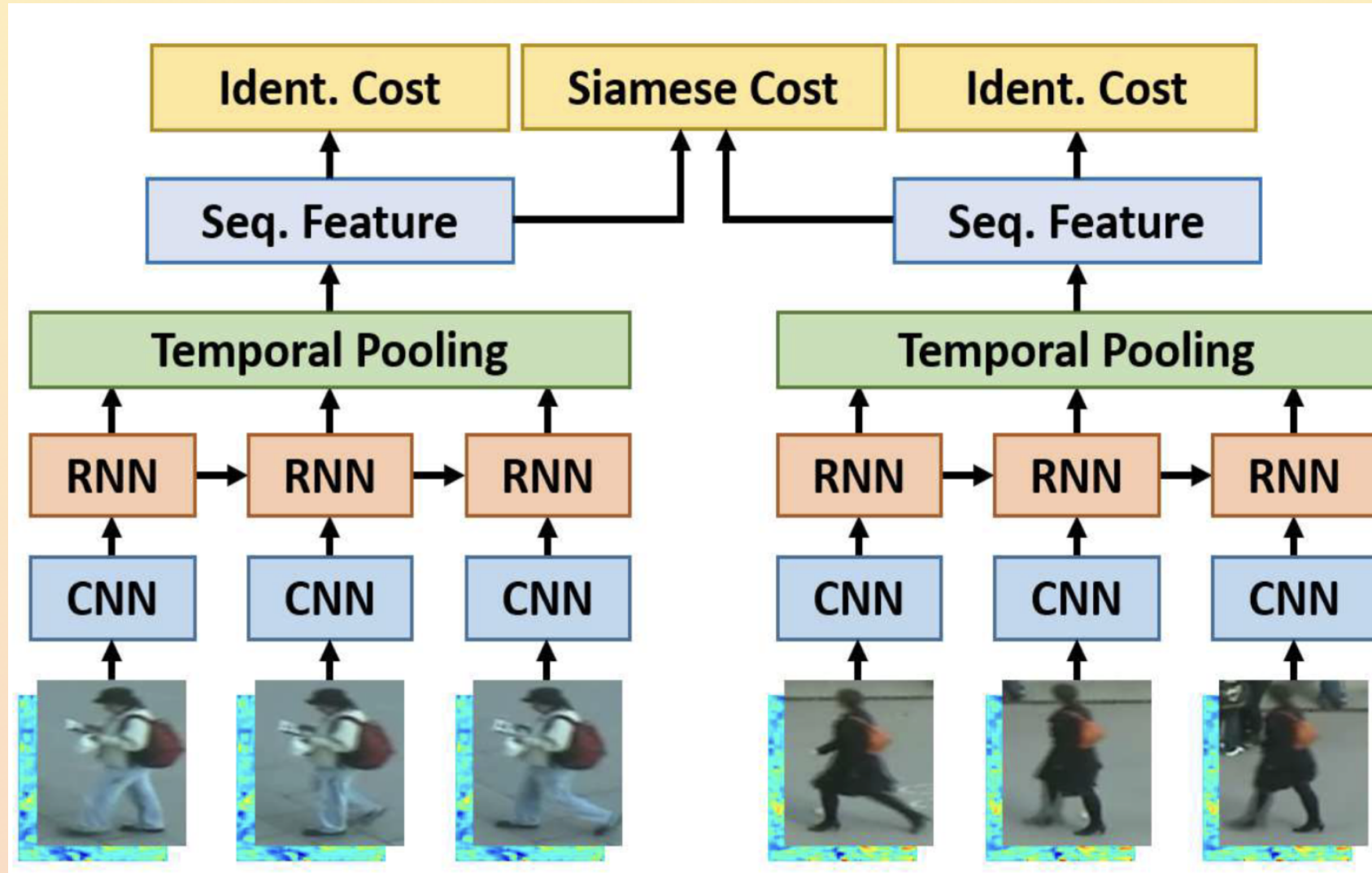


Vast zoology

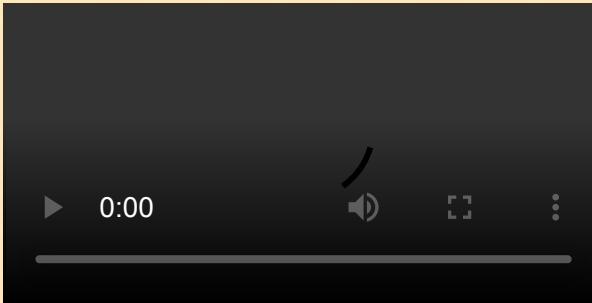
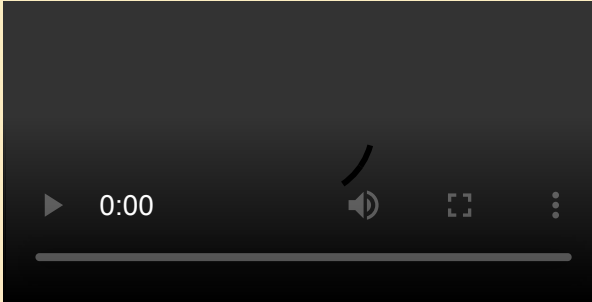
- Recurrent connections between hidden units, that read an entire sequence and then produce a single output



Sequences of images

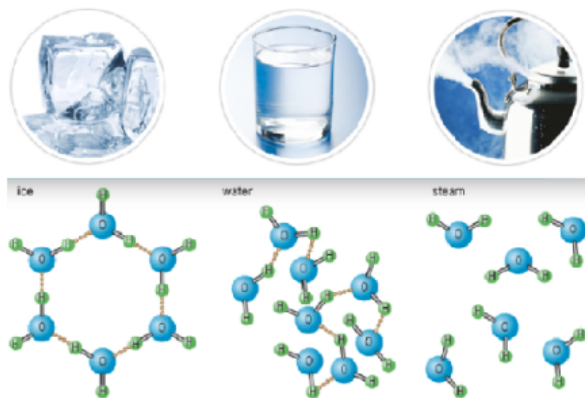


Real-time segmentation

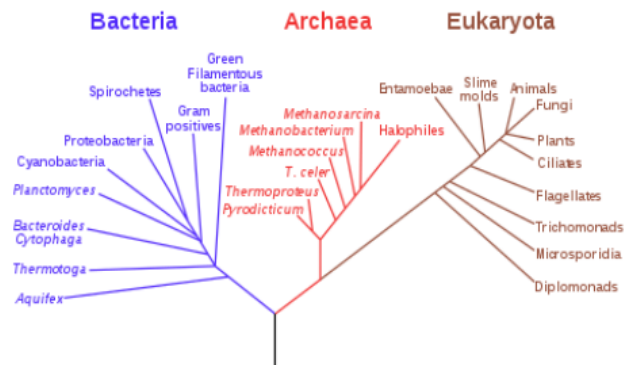


Graphs Represent Structure

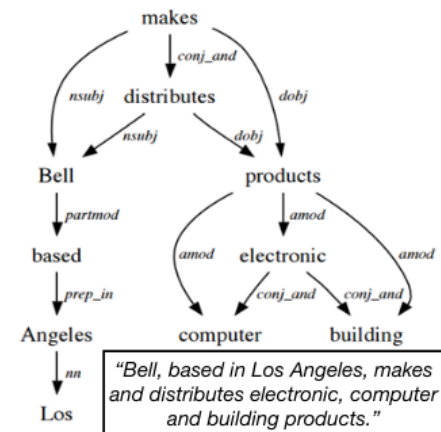
Molecules



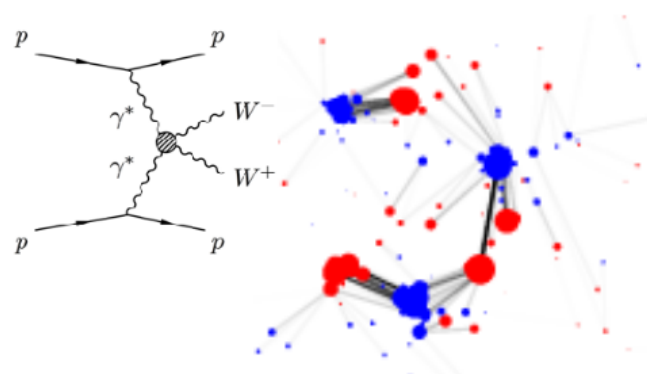
Biological species



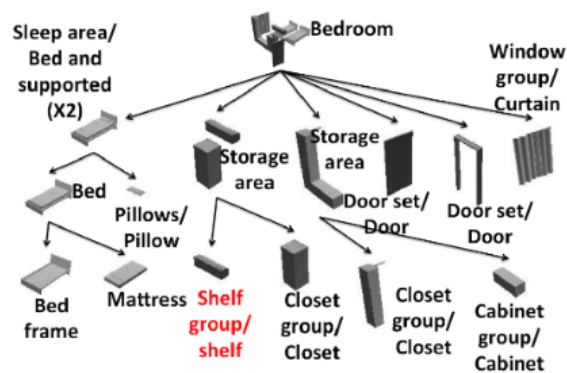
Natural language



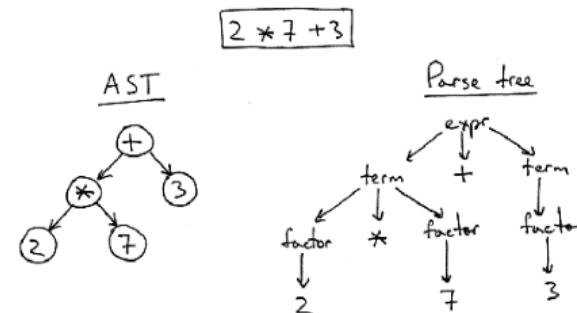
Sub-atomic particles



Everyday scenes

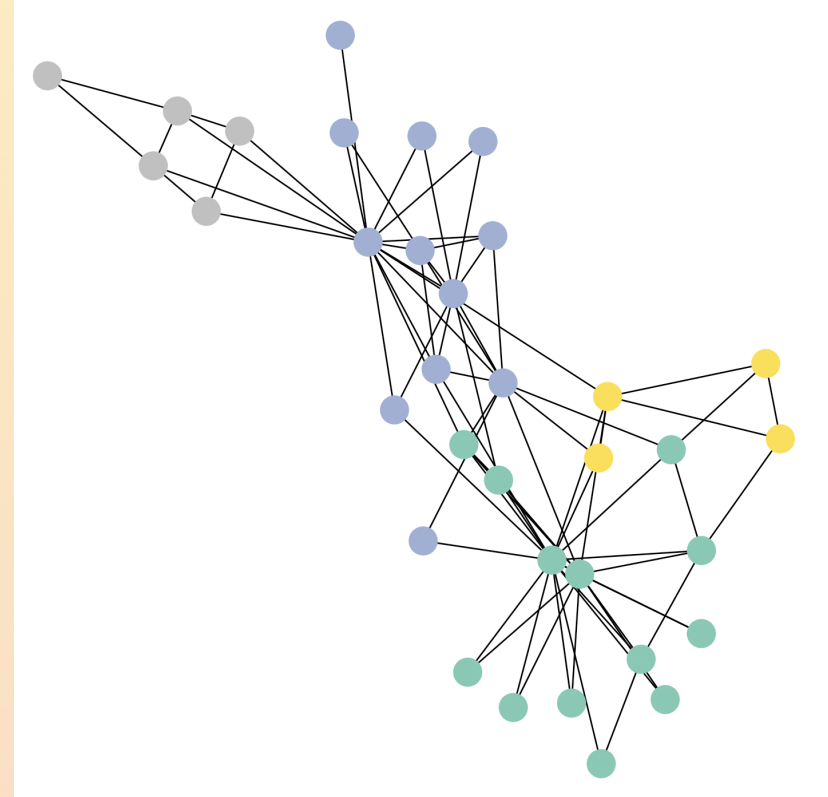


Code



Graph networks

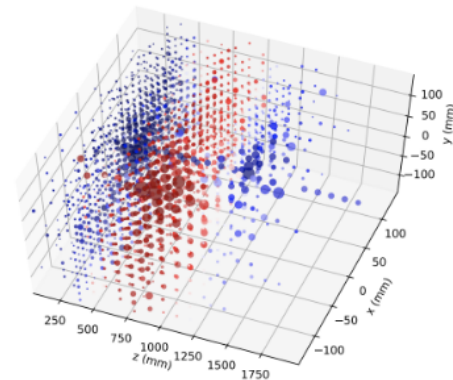
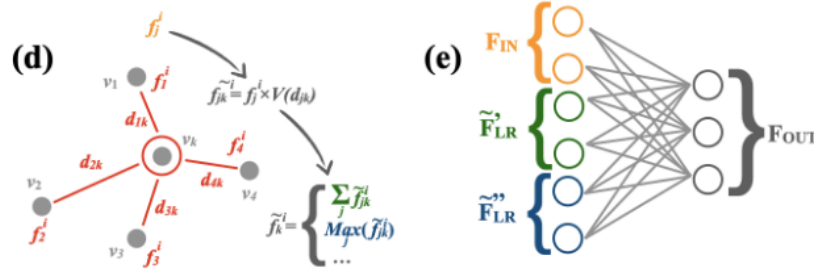
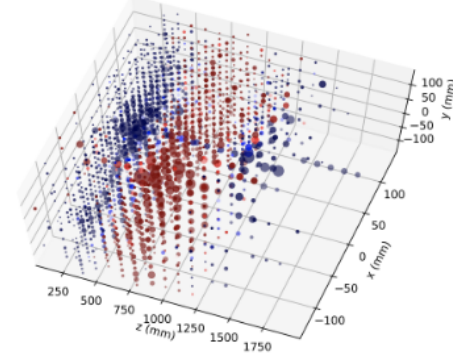
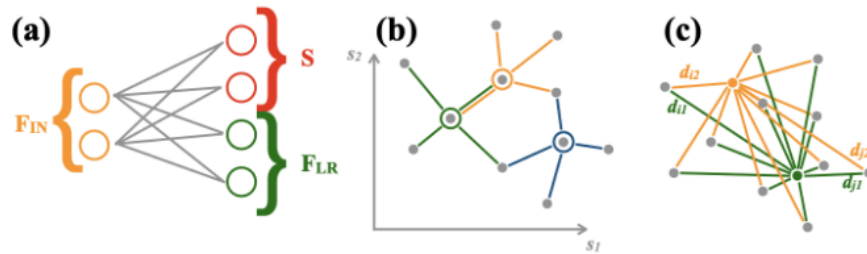
- Represent data as **point clouds**
- Connect data points with weight-dependent connections
- Train the network to find which weights are strongest
 - Learning the connectivity structure of the data



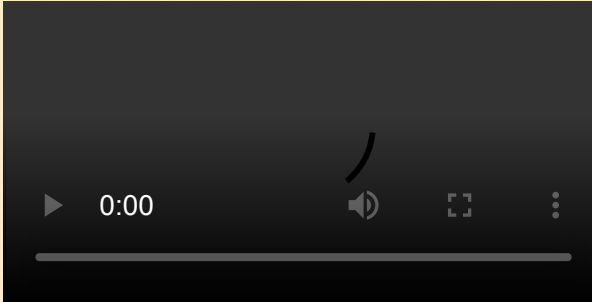
CMS High-granularity calorimeter

- 6 million cells with $\sim 3mm$ spatial resolution, over $600m^2$ of sensors
- Non-projective geometry

Learning representations of irregular particle-detector geometry with distance-weighted graph networks



Graphs for water simulation



Plug the Physics into the AI

How are symmetries implemented?

- Data augmentation
Li, Dobriban '20

- Loss function penalties
- conserved quantities

- Architectural design

- Approximate symmetries (CNN)

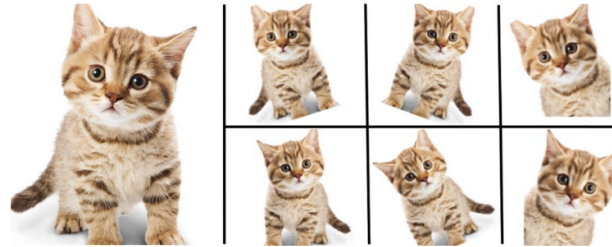
- Exact symmetries

- Weight sharing (message passing)
- Parameterization of symmetry preserving functions (group convolutions)

- Symmetries as constraints Finzi et al '21

- Irreducible representations Kondor, Thomas '18, Fuchs '20
Smidt

- Steerable CNNs Cohen '17, Welling...



Enlarge your Dataset

Credit: Bharath Raj

Cohen, Welling '19 Ravanbakhsh
Rose YU '21, '20. Weiler '21

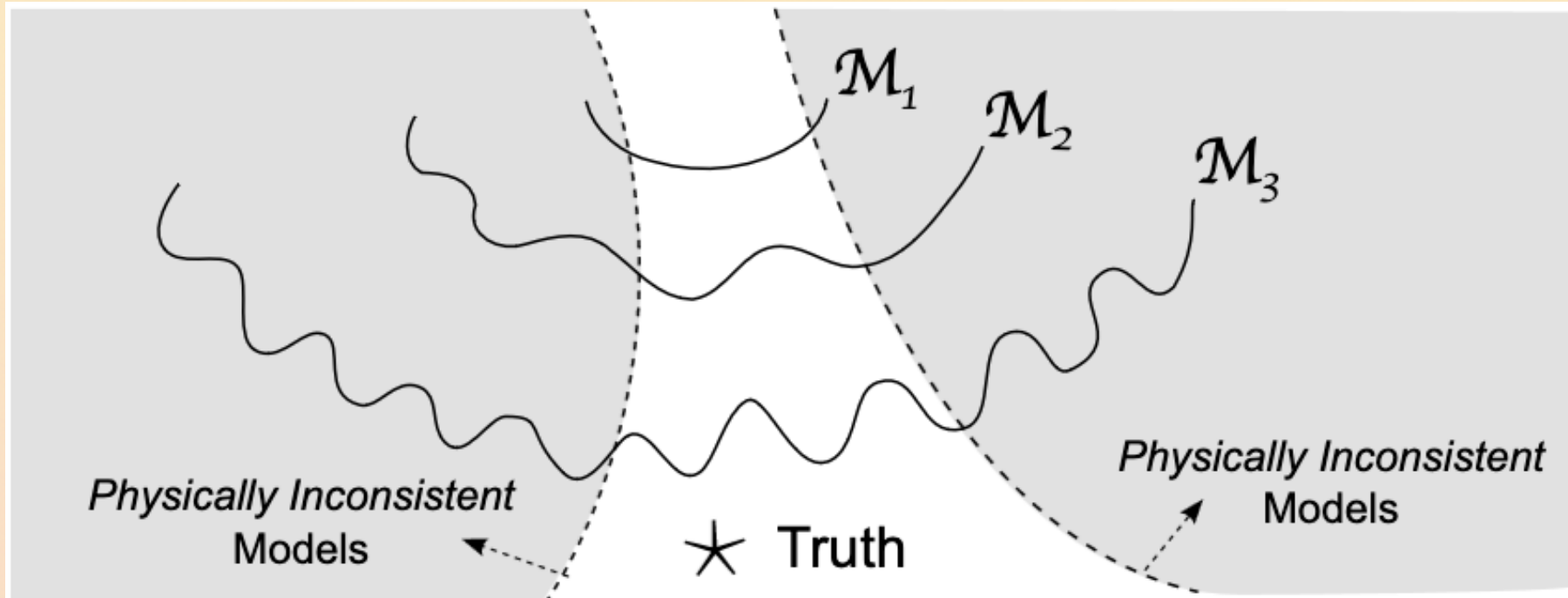
Kondor '18
Maron '18
Cohen '18

Plug the physics into the AI: constraints

$$\hat{y} = f(\mathbf{x}, \theta)$$

- Encode physics knowledge (e.g. inconsistency of models) inside the loss function as a penalty term

$$\mathbf{J}(\mathbf{w}) = \text{Loss}(y, \hat{y}) + \lambda \|\mathbf{w}\|_2^2 + \gamma \Omega(\hat{y}, \Phi)$$



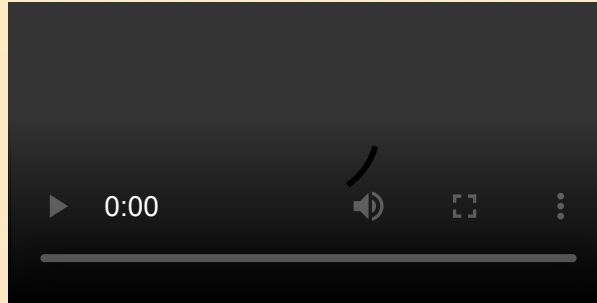
Plug the physics into the AI: network structure

- Equivariance under group transformation can e.g. enforced by convolutional layers
- Some implementations [available in pytorch](#)

$$\begin{array}{ccc} L_{V_1}(\mathcal{X}_1) & \xrightarrow{\mathbb{T}_g} & L_{V_1}(\mathcal{X}_1) \\ \downarrow \phi & & \downarrow \phi \\ L_{V_2}(\mathcal{X}_2) & \xrightarrow{\mathbb{T}'_g} & L_{V_2}(\mathcal{X}_2) \end{array}$$

Plug the Physics into the AI

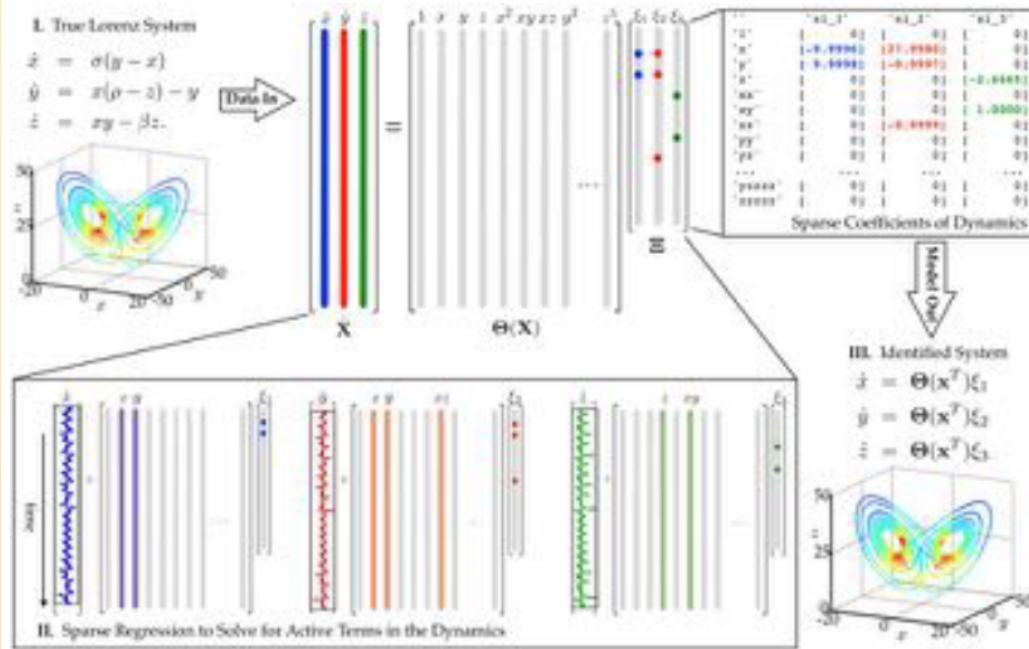
- Physics-aware differential equations solving



Plug the Physics into the AI

- Several ODE problems now solvable via neural networks

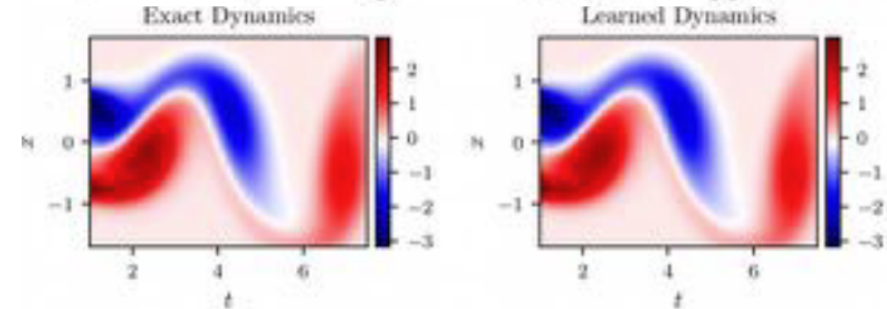
● Who needs Lorenz?



“Discovering governing equations from data by sparse identification of nonlinear dynamical systems” Brunton, Proctor, Kutz, PNAS 2016

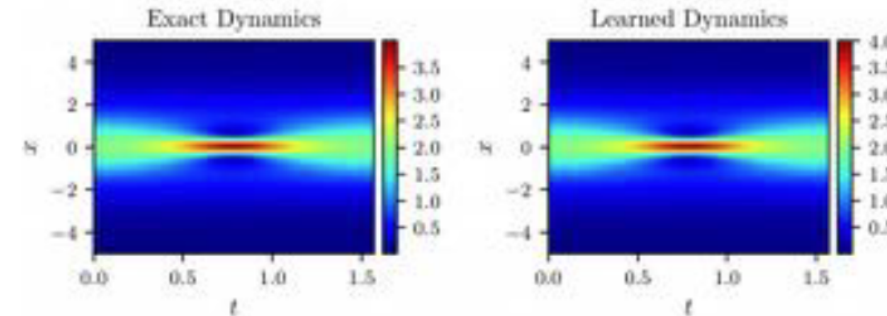
● Who needs Navier Stokes?

$$w_t = -uw_x - vw_y + 0.01(w_{xx} + w_{yy})$$



● Who needs Schrödinger?

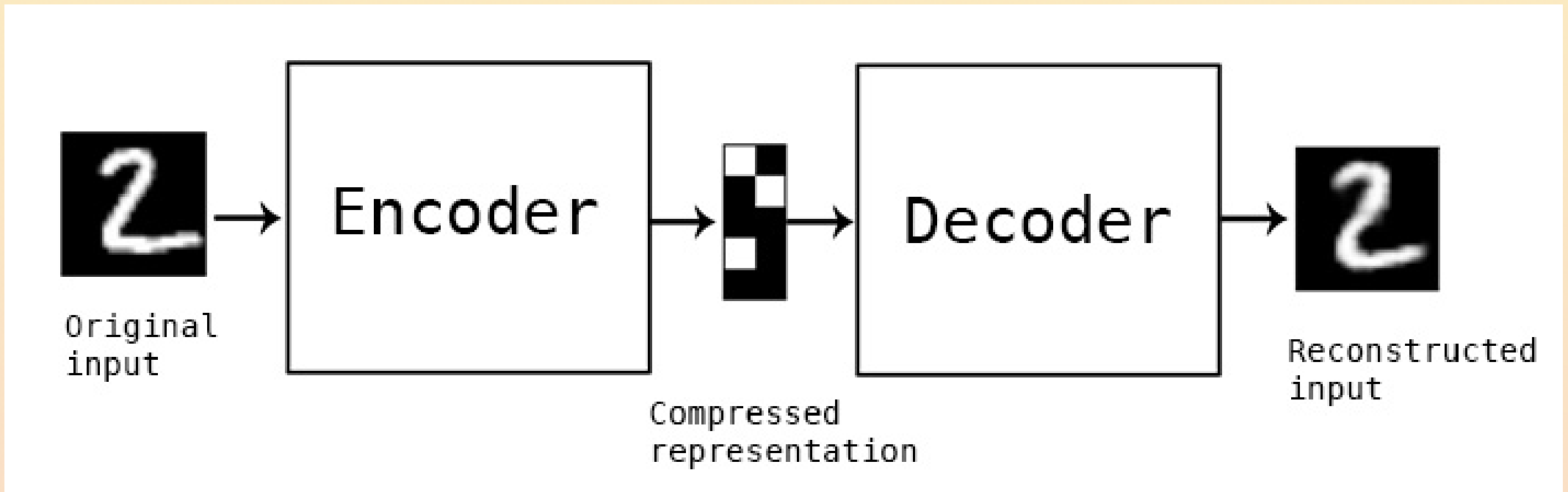
$$\psi_t = 0.5i\psi_{xx} + i|\psi|^2\psi$$



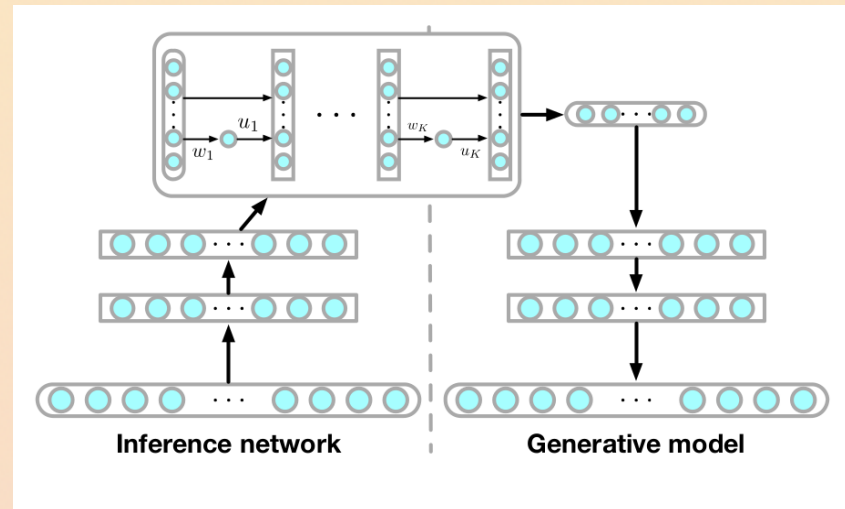
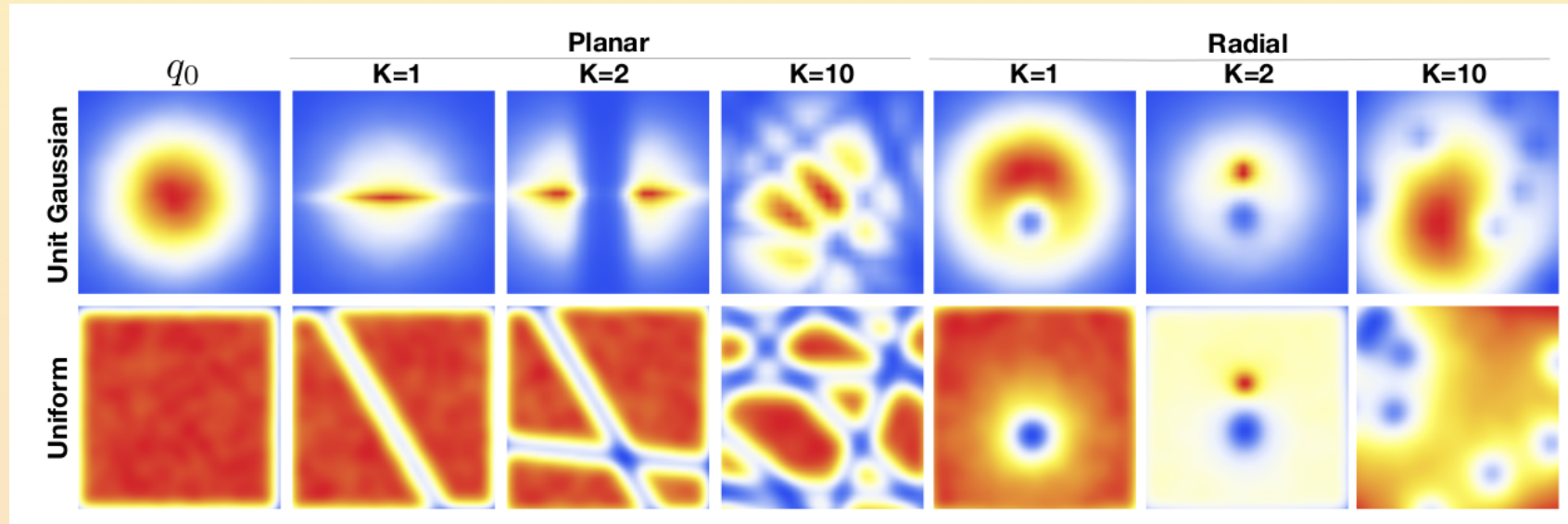
“Deep Hidden Physics Models: Deep Learning of Nonlinear Partial Differential Equations” Raissi, JMLR 2018

Autoencoders

- Learn the data itself passing by a lower-dimensional intermediate representations
 - Capture data generation features into a lower-dimensional space
- Can use for anomaly detection
- Can sample from the latent space to obtain random samples (generative AI)



Invertible networks



Solve inverse problems ("unfolding")

- Correct detector observation noise to recover source distribution

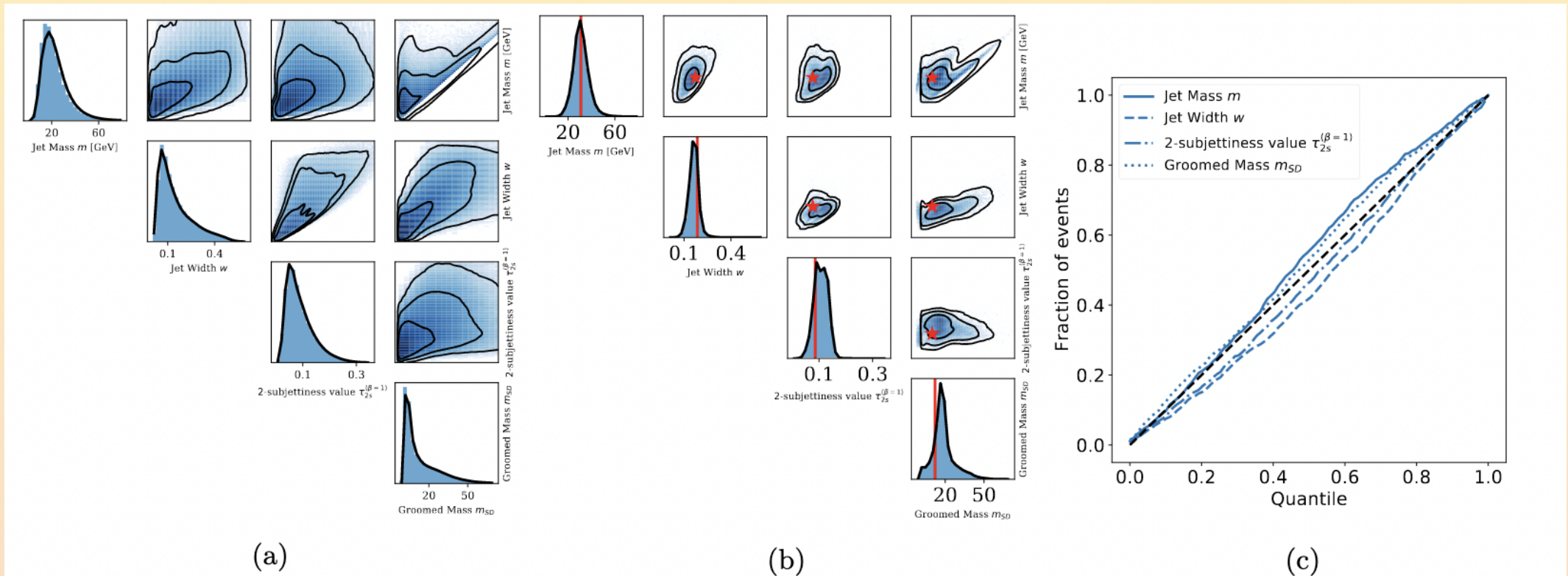
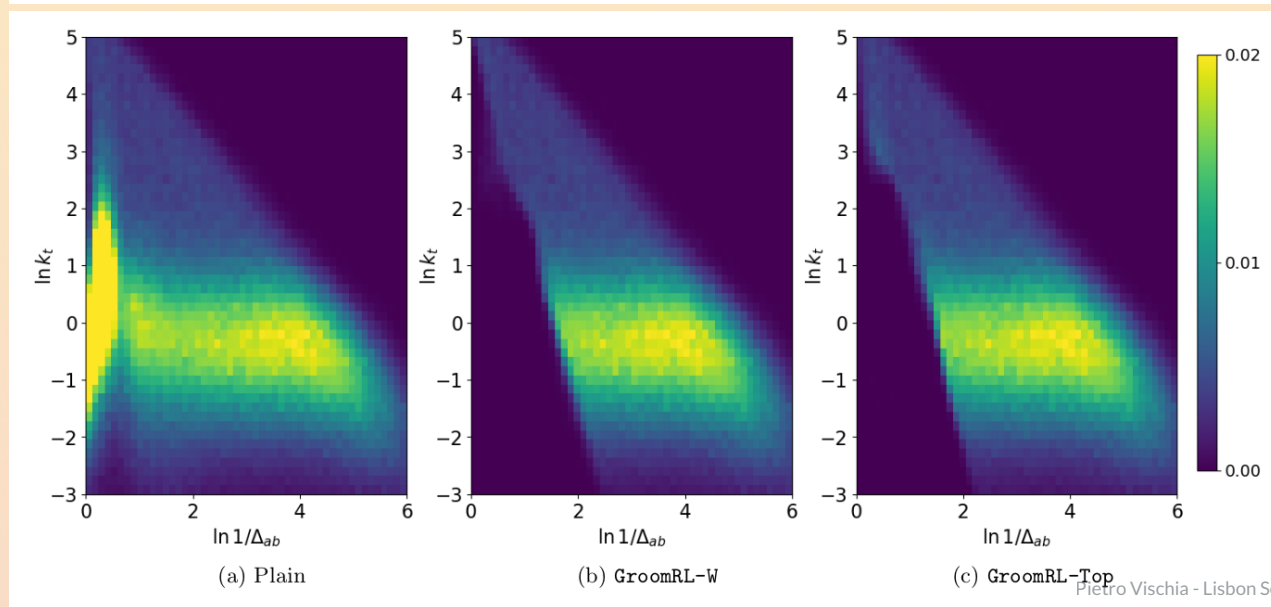
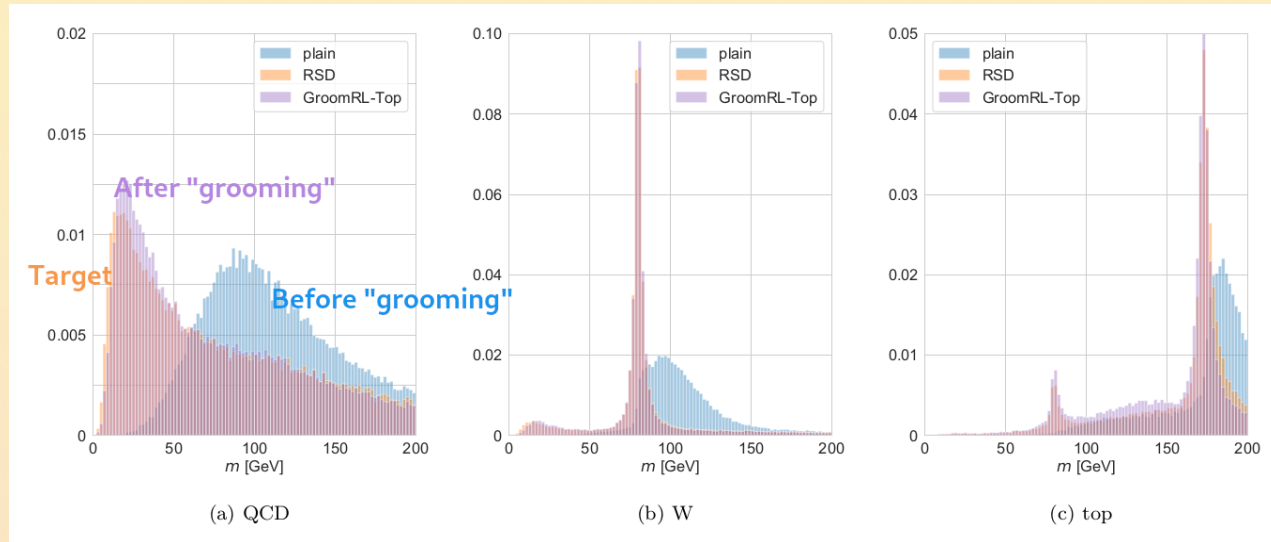


Figure 5: Neural Empirical Bayes for detector correction in collider physics. (a) The source distribution $p(\mathbf{x})$ is shown in blue against the estimated source distribution $q_\theta(\mathbf{x})$ in black. (b) Posterior distribution obtained with rejection sampling, with generating source sample \mathbf{x} indicated in red. (c) Calibration curves for each jet property obtained with rejection sampling on 10000 observations. In (a) and (b), contours represent the 68-95-99.7% levels.

Interpretability



That's all for this morning

This afternoon, your first dense and convolutional networks, and autoencoders