Introduction to Deep Learning: Lecture I

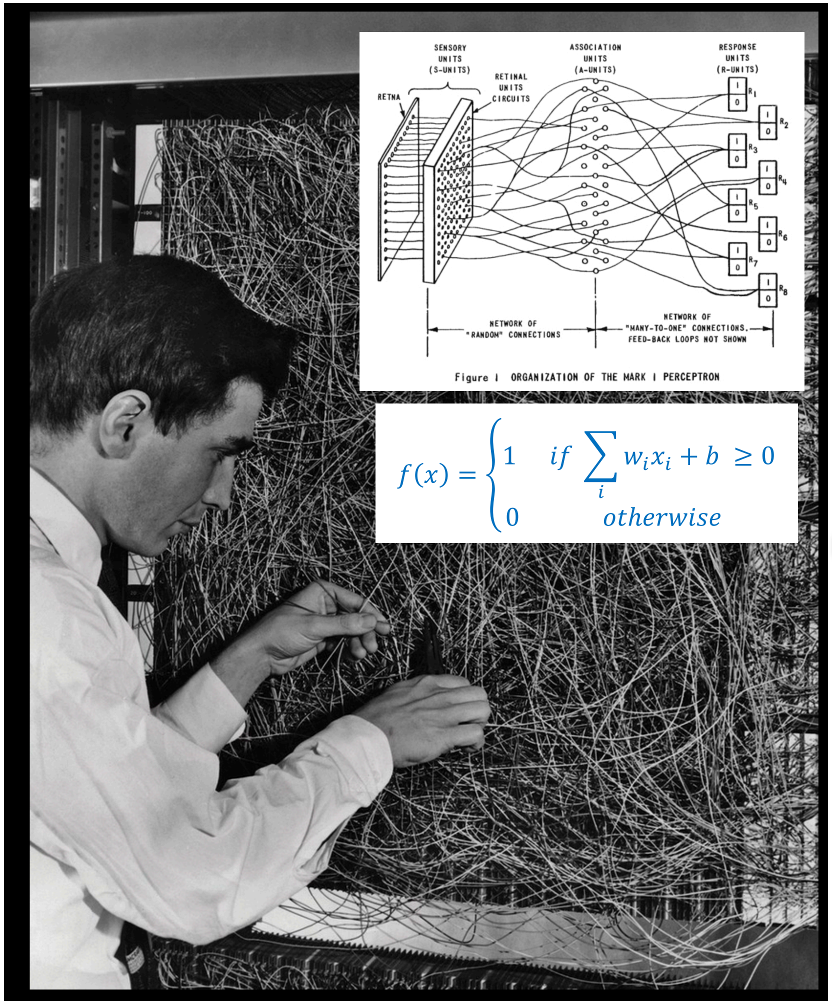
Michael Kagan

SLAC

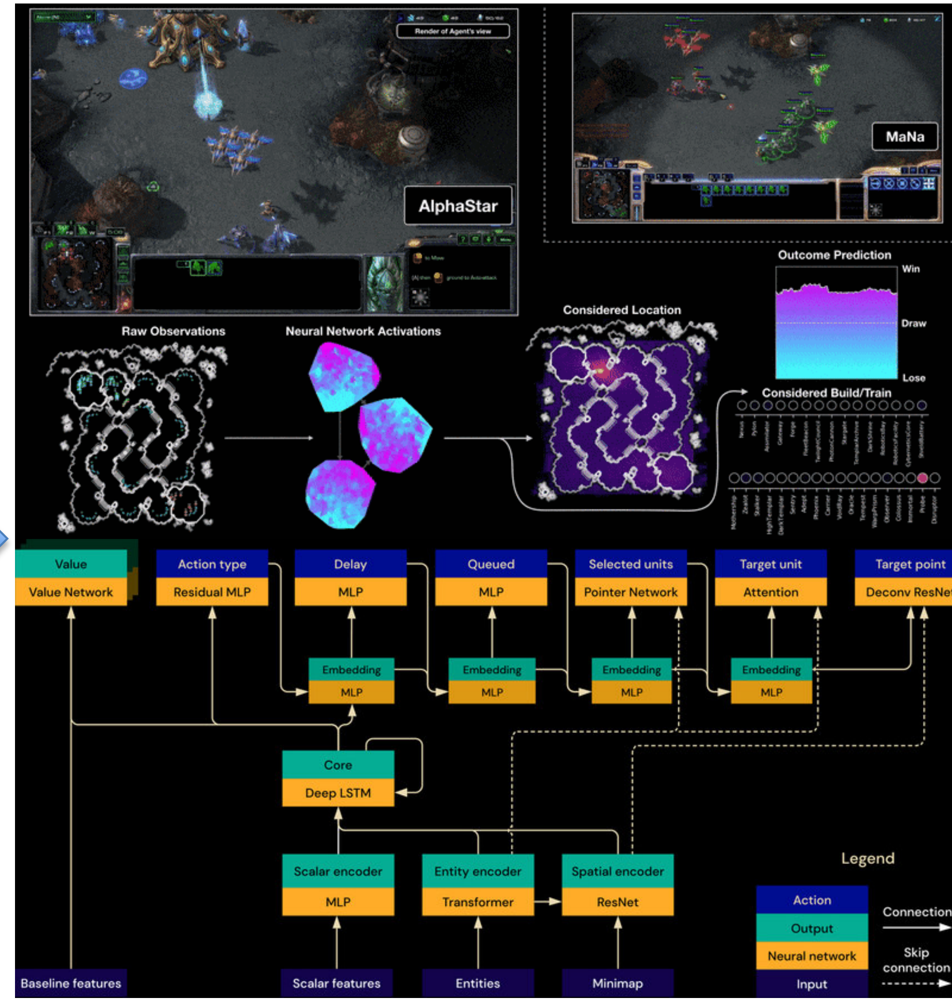
IN2P3 School of Statistics 2021

January 26, 2021

Long History of Neural Networks



Perceptron



AlphaStar

Rosenblatt 1958, 1960

Vinyals et. al. 2019

Modern Neural Networks

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**.

- Yann LeCun, 2018

Modern Neural Networks

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**.

- Yann LeCun, 2018

- Non-linear operations of data with parameters
- Layers (set of operations) designed to perform specific mathematical operations
- Chain together layers to perform desired computation
- Train system (with examples) for desired computation using gradient descent

Modern Neural Networks

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**.

- Yann LeCun, 2018

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**

- Yann LeCun, 2018

The Plan

- Deep Learning is a HUGE field
  - O(10,000) papers submitted to NeurIPS 2020 Conference

- I'm will condense *some* parts of what you would find in *some lectures* of a Deep Learning course

- Highly recommend taking the time to go more slowly through lectures from a class. Online-available Recommendations:
  - Francois Fleuret course at University of Geneva
  - Gilles Louppe course at University of Liege
  - Yann LeCun & Alfredo Canziani course at NYU

The Plan

- From Logistic Regression to Neural Networks
- Basics of Neural Networks
- Deep Neural Networks
- Convolutional Neural Networks

Lecture 1

- Recurrent Neural Networks
  - And a bit about Graph Neural Networks
- AutoEncoders and Generative Models

Lecture 2

Reminder: Empirical Risk Minimization

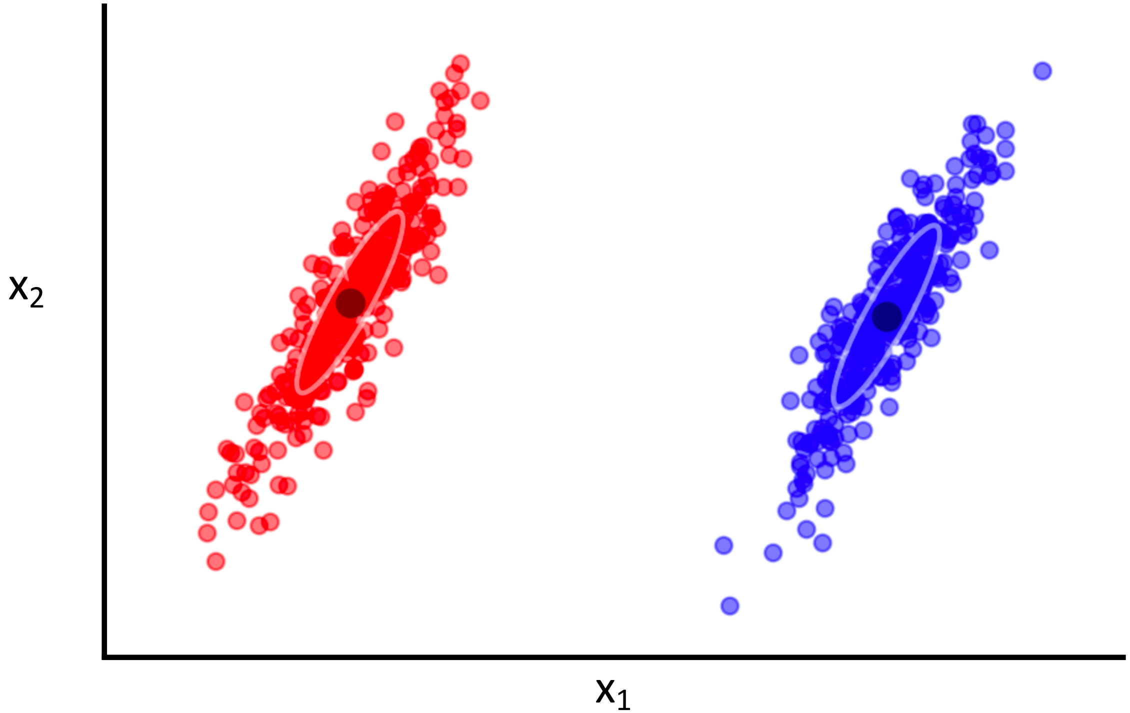
$$\underbrace{\arg\min_{\mathbf{w}} \frac{1}{N}\sum_{i=1}^{N} L(h(\mathbf{x}_i;\mathbf{w}), y_i)}_{\text{Average expected loss}} + \underbrace{\lambda\Omega(\mathbf{w})}_{\text{Model regularization}}$$

- Framework to design learning algorithms
  - L(·) is a loss function comparing prediction h(·) with target y
  - $\Omega(\mathbf{w})$ is a regularizer, penalizing certain values of $\mathbf{w}$
    - λ controls how much penalty… a hyperparameter we have to tune
- Learning is cast as an optimization problem

From Logistic Regression to Neural Networks

Linear Discriminant Analysis

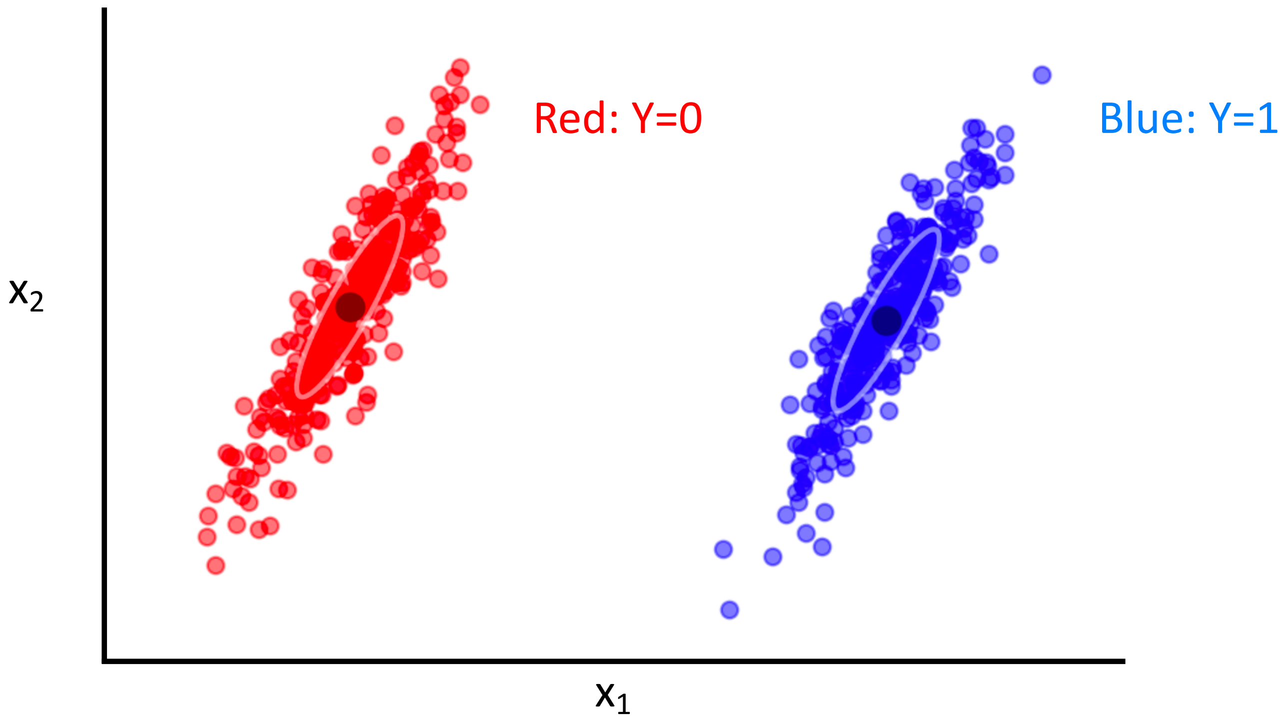
- Goal: Separate data from two classes / populations



Linear Discriminant Analysis

- Goal: Separate data from two classes / populations

- Data from joint distribution $(\mathbf{x}, y) \sim p(\mathbf{X}, Y)$
  - Features: $\mathbf{x} \in \mathbb{R}^m$
  - Labels: $y \in \{0,1\}$



Linear Discriminant Analysis

- Goal: Separate data from two classes / populations

- Data from joint distribution $(\mathbf{x}, y) \sim p(\mathbf{X}, Y)$
  - Features: $\mathbf{x} \in \mathbb{R}^m$
  - Labels: $y \in \{0,1\}$

- Breakdown the joint distribution:

$$p(x, y) = p(x|y)p(y)$$

Likelihood:
Distribution of features
for a given class

Prior:
Probability of each class

Linear Discriminant Analysis

- Goal: Separate data from two classes / populations

- Data from joint distribution ($\mathbf{x}$, y) ~ p($\mathbf{X}$, Y)
  - Features: $\mathbf{x} \in \mathbb{R}^m$
  - Labels: $y \in \{0,1\}$

- Breakdown the joint distribution:

$$p(x, y) = p(x|y)p(y)$$

- Assume likelihoods are Gaussian

$$p(x|y) = \frac{1}{\sqrt{(2\pi)^m|\Sigma|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_y)^T \Sigma^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_y)\right)$$

Predicting the Class

- Separating classes → Predict the class of a point $\mathbf{x}$

$$p(y = 1|\mathbf{x})$$

- Want to build a classifier to predict the label y given and input $\mathbf{x}$

Predicting the Class

- Separating classes → Predict the class of a point $\mathbf{x}$

$$p(y = 1|\mathbf{x}) = \frac{p(\mathbf{x}|y = 1)p(y = 1)}{p(\mathbf{x})}$$

Bayes Rule

Predicting the Class

- Separating classes → Predict the class of a point $\mathbf{x}$

$$p(y=1|\mathbf{x}) = \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x})}$$

Bayes Rule

$$= \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x}|y=0)p(y=0) + p(\mathbf{x}|y=1)p(y=1)}$$

Marginal definition

Predicting the Class

- Separating classes → Predict the class of a point $\mathbf{x}$

$$p(y=1|\mathbf{x}) = \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x})}$$

Bayes Rule

$$= \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x}|y=0)p(y=0) + p(\mathbf{x}|y=1)p(y=1)}$$

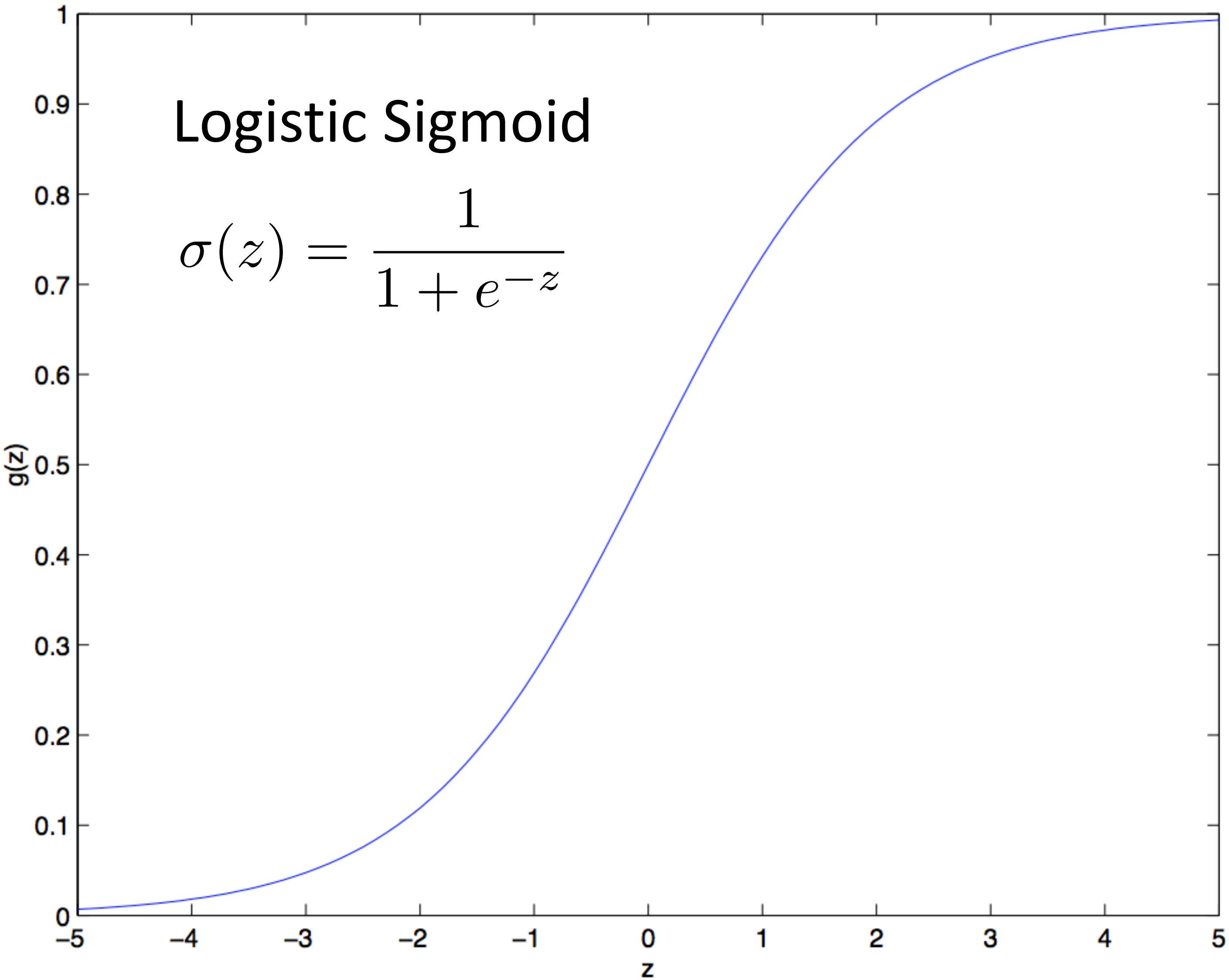
Marginal definition

$$= \frac{1}{1 + \frac{p(\mathbf{x}|y=0)p(y=0)}{p(\mathbf{x}|y=1)p(y=1)}}$$

$$= \frac{1}{1 + \exp\left(\log \frac{p(\mathbf{x}|y=0)p(y=0)}{p(\mathbf{x}|y=1)p(y=1)}\right)}$$

Why?

Logistic Sigmoid Function



Predicting Classes with Gaussian Likelihoods

$$p(y=1|\mathbf{x}) = \sigma\Big(\log\frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=0)} + \log\frac{p(y=1)}{p(y=0)}\Big)$$

Log-likelihood ratio

Constant w.r.t. **x**

Predicting Classes with Gaussian Likelihoods

$$p(y=1|\mathbf{x}) = \sigma\Big(\log\frac{p(\mathbf{x}|y=1)}{p(\mathbf{x}|y=0)} + \log\frac{p(y=1)}{p(y=0)}\Big)$$

- For our Gaussian data:

$$= \sigma\Big(\log p(\mathbf{x}|y=1) - \log p(\mathbf{x}|y=0) + const.\Big)$$

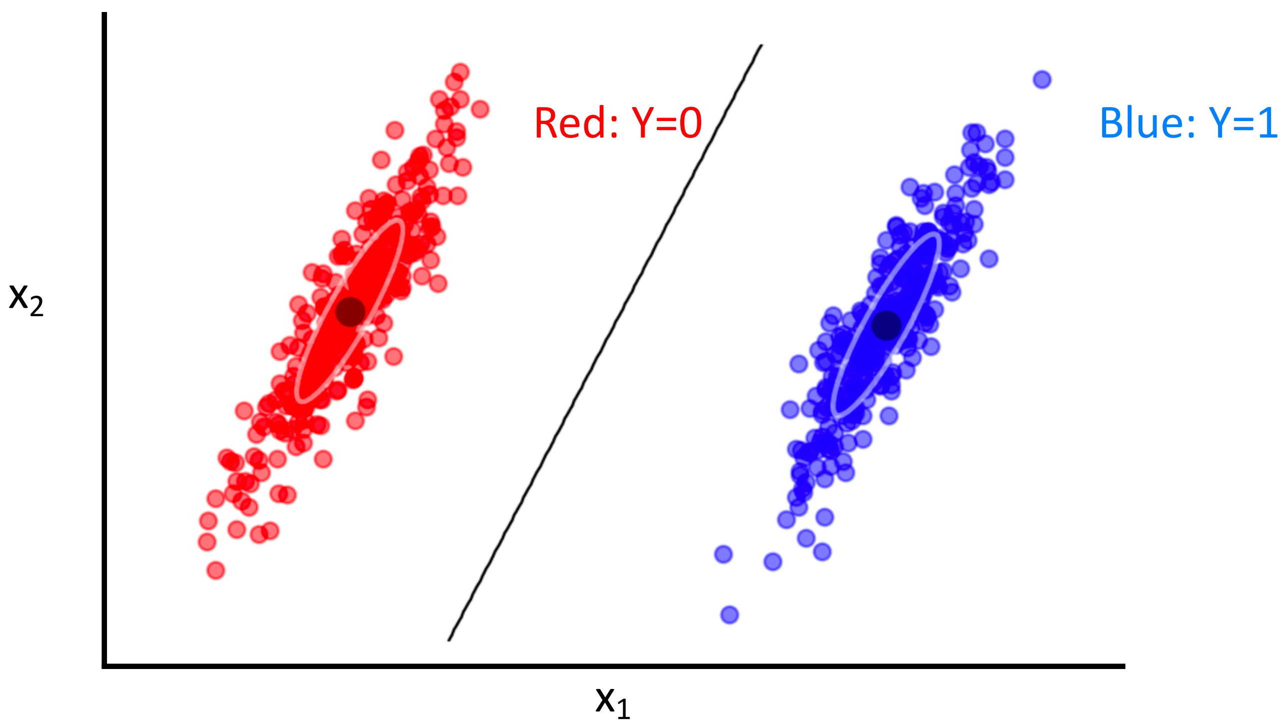
$$= \sigma\Big(-\frac{1}{2}(\mathbf{x}-\mu_1)^T\Sigma^{-1}(\mathbf{x}-\mu_1) + \frac{1}{2}(\mathbf{x}-\mu_0)^T\Sigma^{-1}(\mathbf{x}-\mu_0) + const.\Big)$$

$$= \sigma\Big(\mathbf{w}^T\mathbf{x} + b\Big)$$

Collect terms

What did we learn?

- For this data, the log-likelihood ratio is linear!
  - Line defines boundary to separate the classes
  - Sigmoid turns distance from boundary to probability



Logistic Regression

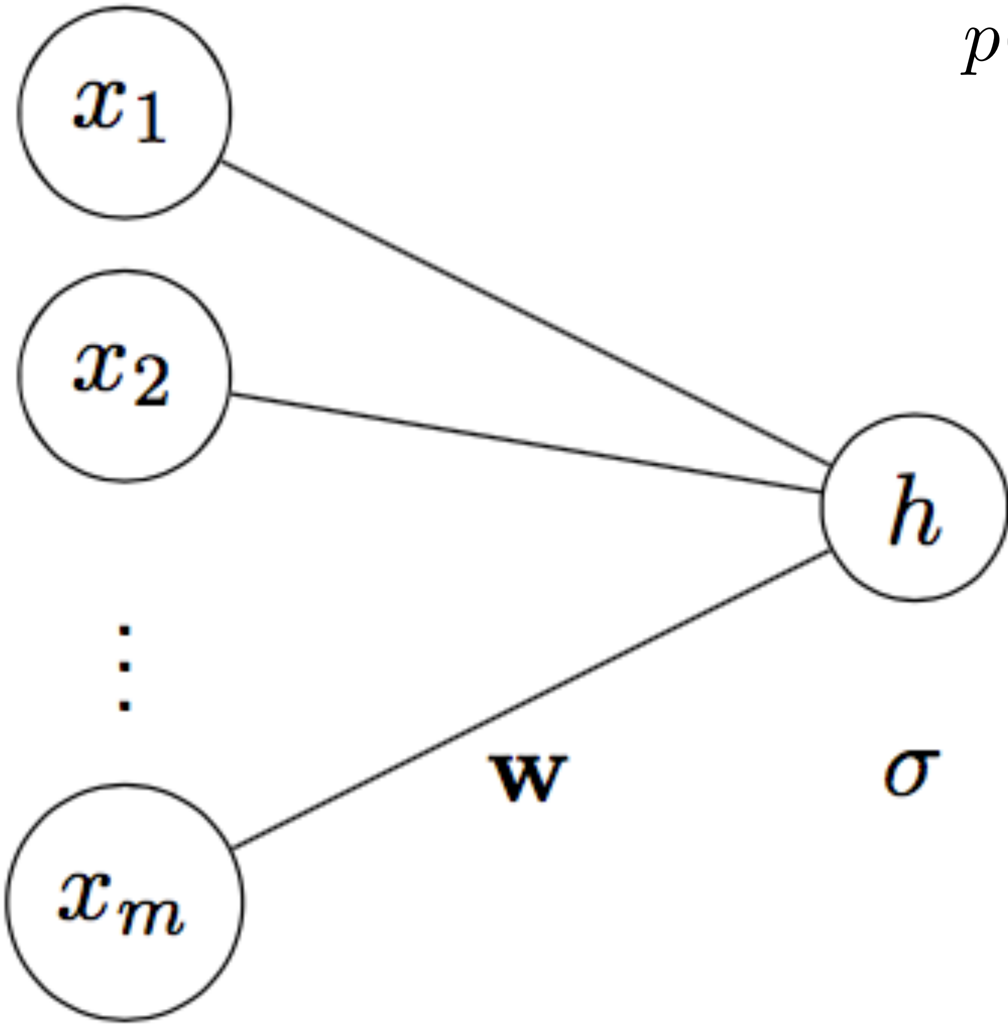
- What if we ignore Gaussian assumption on data?

Model: $$p(y=1|\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right) \equiv h(\mathbf{x};\mathbf{w})$$

- Farther from boundary $\mathbf{w}^{\mathrm{T}}\mathbf{x}+\mathrm{b}=0$, more certain about class

- Sigmoid converts distance to class probability

Logistic Regression



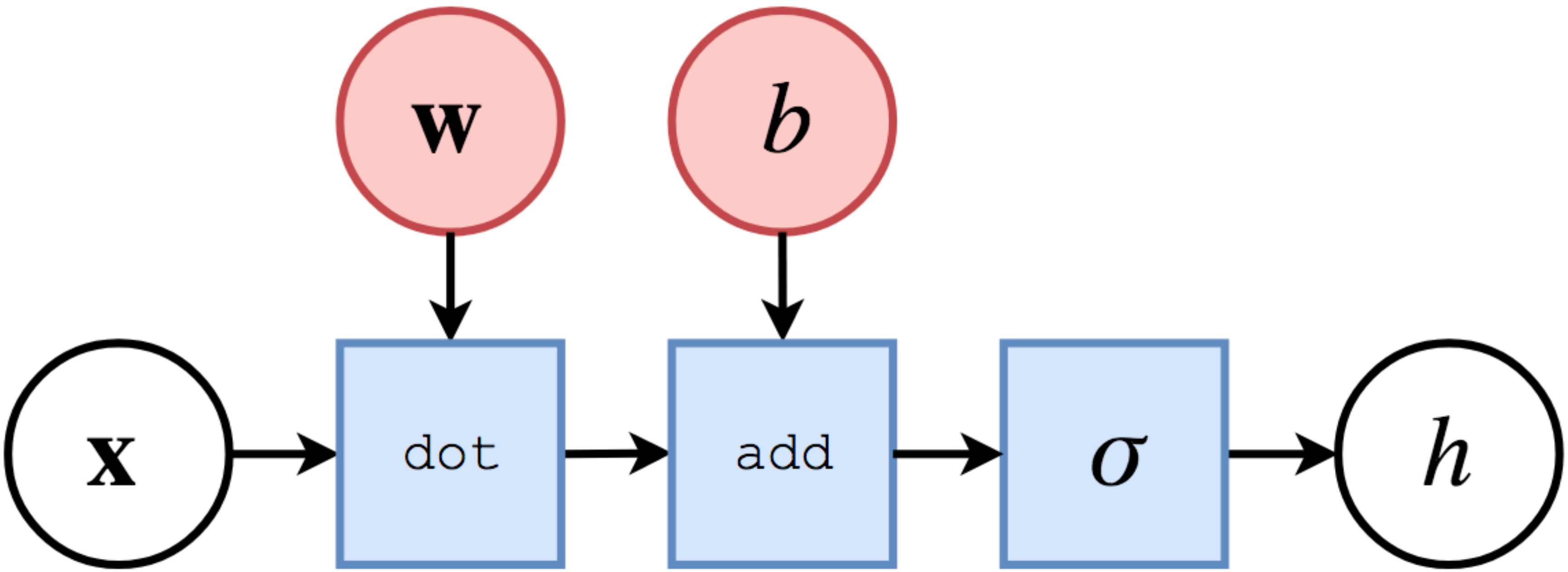
$$p(y=1|\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x}+b\right)$$
$$= \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}\text{-b}}}$$

This unit is the main building block of Neural Networks!

Logistic Regression

Slide credit: G. Louppe

- Computational Graph of function
  - White node = input
  - Red node = model parameter
  - Blue node = intermediate operations



This unit is the main building block of Neural Networks!

Logistic Regression

- What if we ignore Gaussian assumption on data?

$$\text{Model:} \qquad p(y=1|\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x}+b\right) \equiv h(\mathbf{x};\mathbf{w})$$

- With $p_i \equiv p(y_i = y|\boldsymbol{x}_i)$

$$P(y_i = y|x_i) = \text{Bernoulli}(p_i) = (p_i)^{y_i}(1-p_i)^{1-y_i} = \begin{cases} p_i & \text{if } y_i=1 \\ 1\text{-}p_i & \text{if } y_i=0 \end{cases}$$

- **Goal**:
  - Given i.i.d. dataset of pairs ($\mathbf{x}_i$, y_i) find **w** and b that maximize likelihood of data

Logistic Regression

- Negative log-likelihood

$$-\ln \mathcal{L} = -\ln \prod_i (p_i)^{y_i} (1 - p_i)^{1-y_i}$$

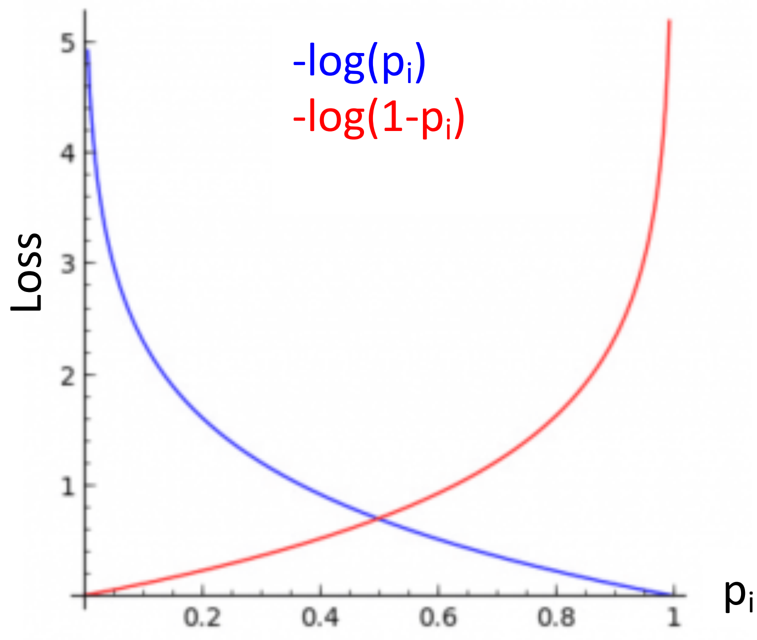
Logistic Regression

- Negative log-likelihood

$$-\ln \mathcal{L} = -\ln \prod_i (p_i)^{y_i}(1-p_i)^{1-y_i}$$

$$= -\sum_i y_i \ln(p_i) + (1-y_i)\ln(1-p_i)$$

binary cross entropy loss function!



Logistic Regression

- Negative log-likelihood

$$-\ln \mathcal{L} = -\ln \prod_i (p_i)^{y_i} (1-p_i)^{1-y_i}$$

$$= -\sum_i y_i \ln(p_i) + (1-y_i)\ln(1-p_i)$$

binary cross entropy loss function!

$$= \sum_i y_i \ln(1+e^{-\mathbf{w}^T\mathbf{x}}) + (1-y_i)\ln(1+e^{\mathbf{w}^T\mathbf{x}})$$

- No closed form solution to $w^* = \arg\min_w -\ln \mathcal{L}(w)$

- How to solve for $\mathbf{w}$?

How to Minimize Loss $\mathcal{L}(\theta)$? Gradient Descent

- **Gradient Descent**:

  Make a step $\theta \leftarrow \theta - \eta v$ in ***direction*** v with ***step size*** γ to reduce loss

- How does loss change in different directions?

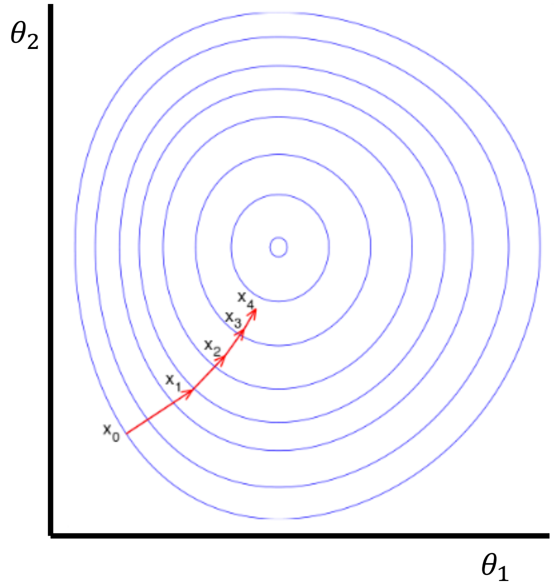
  Let λ be a perturbation along direction v

$$\frac{d}{d\lambda}\mathcal{L}(\theta + \lambda v)\bigg|_{\lambda=0} = v \cdot \nabla_\theta \mathcal{L}(\theta)$$

- Then Steepest Descent direction is: $v = -\nabla_\theta \mathcal{L}(\theta)$

Gradient Descent

- Minimize loss by repeated gradient steps
  - Compute gradient w.r.t. current parameters: $\nabla_{\theta_i}\mathcal{L}(\theta_i)$
  - Update parameters: $\theta_{i+1} \leftarrow \theta_i - \eta\nabla_{\theta_i}\mathcal{L}(\theta_i)$
  - η is the *learning rate*, controls how big of a step to take



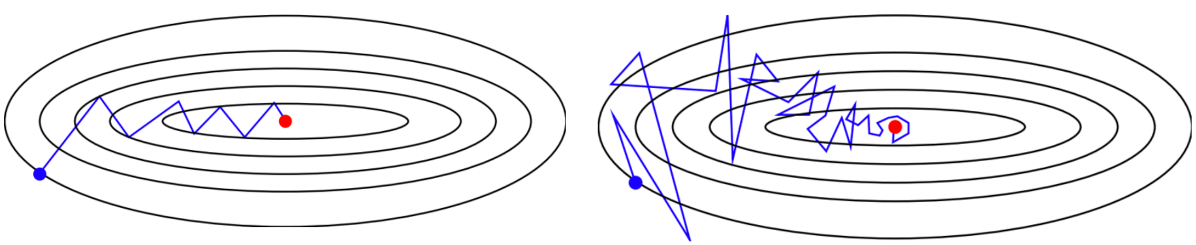
Stochastic Gradient Descent

- Loss is composed of a sum over samples:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \mathcal{L}\big(y_i, h(x_i;\theta)\big)$$

  - Computing gradient grows linearly with N!

- **(Mini-Batch) Stochastic Gradient Descent**
  - Compute gradient update using 1 random sample (small size batch)
  - Gradient is unbiased → on average it moves in correct direction
  - Tends to be much faster the full gradient descent



Batch gradient descent *Stochastic gradient descent*

Stochastic Gradient Descent

- Loss is composed of a sum over samples:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \mathcal{L}\big(y_i, h(x_i; \theta)\big)$$

  - Computing gradient grows linearly with N!

- **(Mini-Batch) Stochastic Gradient Descent**
  - Compute gradient update using 1 random sample (small size batch)
  - Gradient is unbiased → on average it moves in correct direction
  - Tends to be much faster the full gradient descent

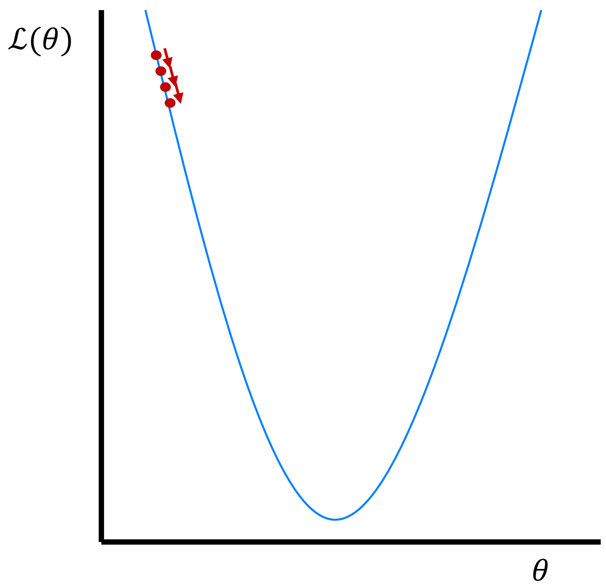
- Several updates to SGD, like momentum, ADAM, RMSprop to
  - Help to speed up optimization in flat regions of loss
  - Have adaptive learning rate
  - Learning rate adapted for each parameter
  - …

Step Sizes

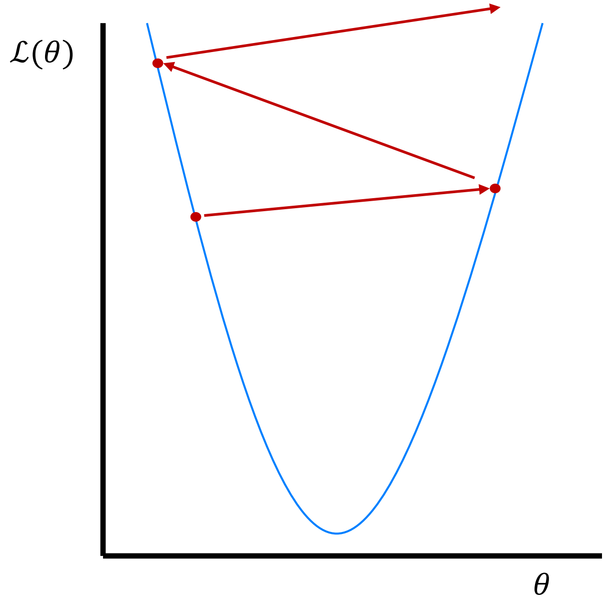
- Too small a learning rate, convergence very slow

- Too large a learning rate, algorithm diverges

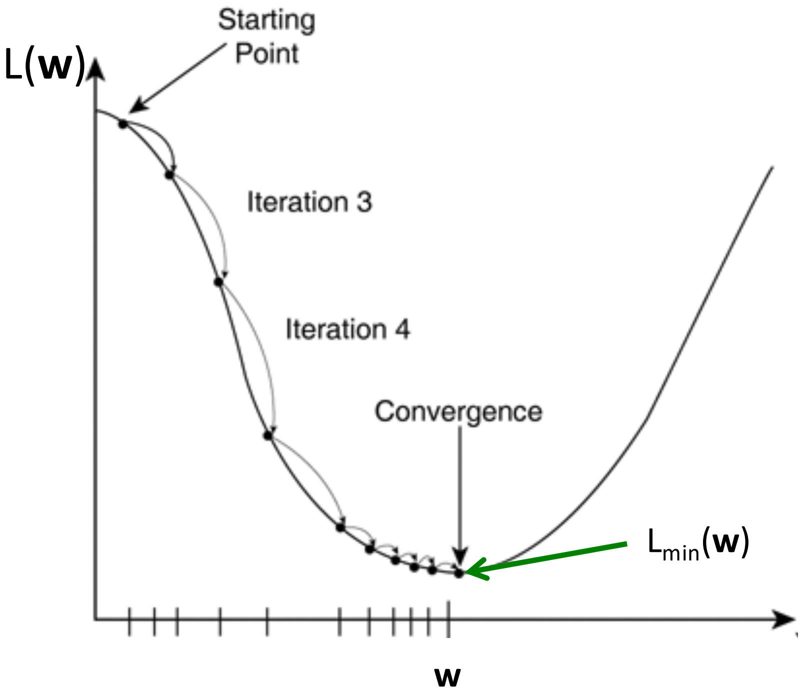
Small Learning rate

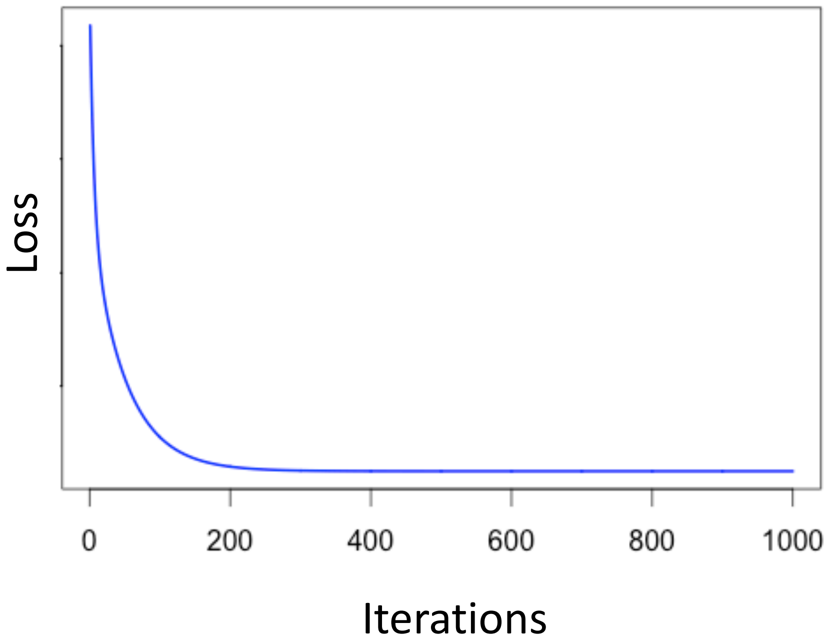


Large Learning rate



Gradient Descent





- Logistic Regression Loss is convex
  – Single global minimum

- Iterations lower loss and move toward minimum

Logistic Regression Example

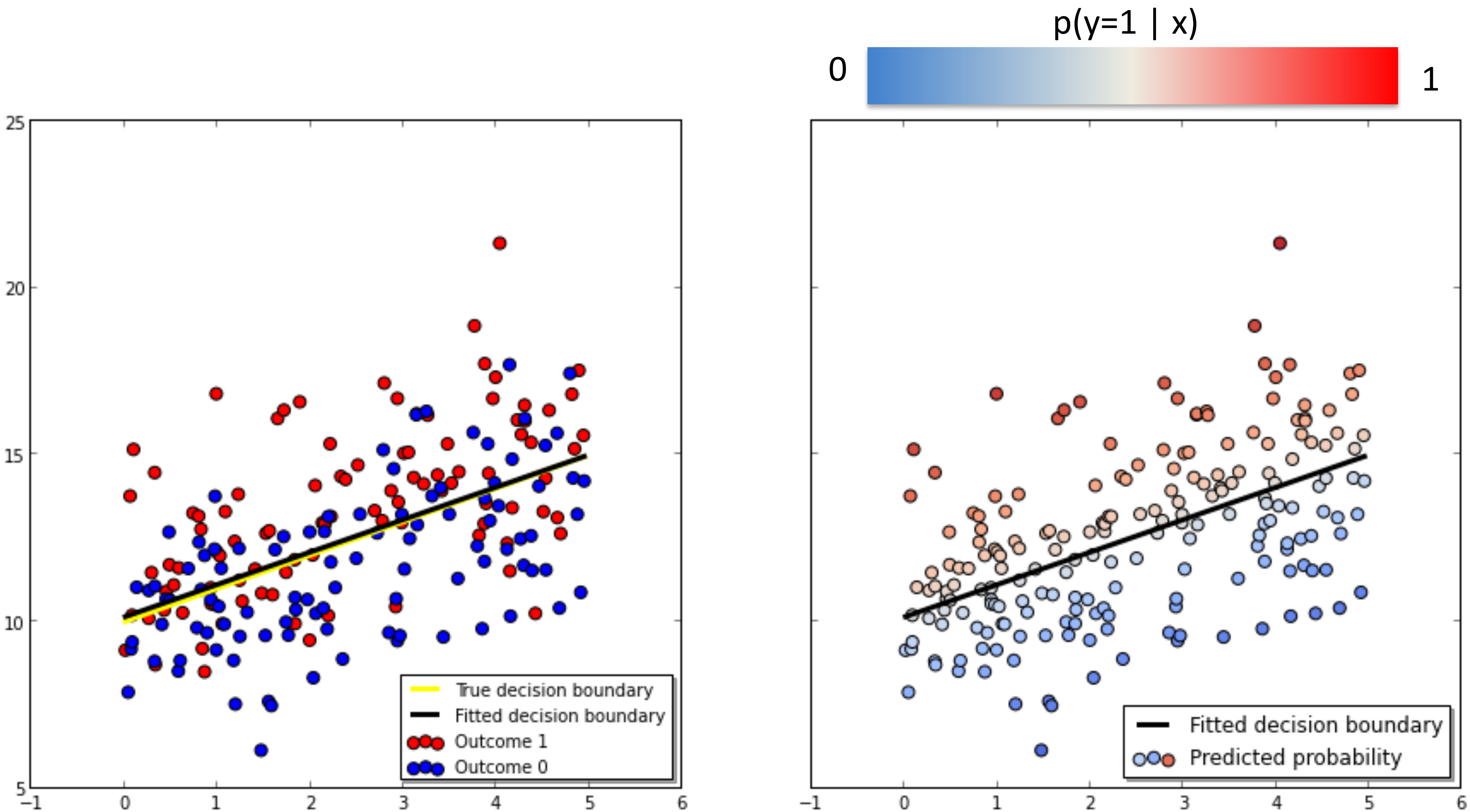


Image source

Adding non-linearity

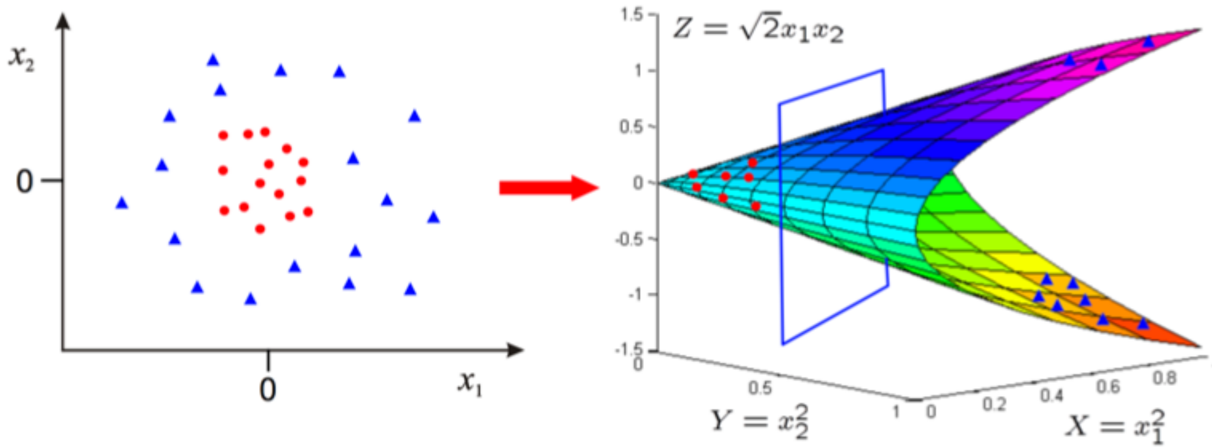
- What if we want a non-linear decision boundary?

Adding non-linearity

- What if we want a non-linear decision boundary?
  - Choose basis functions, e.g: $\phi(x) \sim \{x^2, \sin(x), \log(x), \ldots\}$

$$p(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\phi(\mathbf{x})}}$$

$$\Phi : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \qquad \mathbb{R}^2 \rightarrow \mathbb{R}^3$$



Adding non-linearity

- What if we want a non-linear decision boundary?
  - Choose basis functions, e.g: $\phi(\mathrm{x}) \sim \{\mathrm{x}^2, \sin(\mathrm{x}), \log(\mathrm{x}), \ldots\}$

$$p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \phi(\mathbf{x})}}$$

- What if we don't know what basis functions we want?

Adding non-linearity

- What if we want a non-linear decision boundary?
  - Choose basis functions, e.g: $\phi(\mathrm{x}) \sim \{\mathrm{x}^2, \sin(\mathrm{x}), \log(\mathrm{x}), \ldots\}$

$$p(y=1|\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\phi(\mathbf{x})}}$$

- What if we don't know what basis functions we want?

- Learn the basis functions directly from data

$$\phi(\mathbf{x}; \mathbf{u}) \qquad \mathbb{R}^{\mathrm{m}} \rightarrow \mathbb{R}^{\mathrm{d}}$$

  - Where **u** is a set of parameters for the transformation

Adding non-linearity

- What if we want a non-linear decision boundary?
  - Choose basis functions, e.g: $\phi(\mathrm{x}) \sim \{\mathrm{x}^2, \sin(\mathrm{x}), \log(\mathrm{x}), \ldots\}$

$$p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \phi(\mathbf{x})}}$$

- What if we don't know what basis functions we want?
- Learn the basis functions directly from data

$$\phi(\mathbf{x}; \mathbf{u}) \qquad \mathbb{R}^{\mathrm{m}} \longrightarrow \mathbb{R}^{\mathrm{d}}$$

  - Where **u** is a set of parameters for the transformation
  - Combines basis selection and learning
  - Several different approaches, focus here on neural networks
  - Complicates the optimization

Neural Networks

- Define the basis functions j = {1…d}

$$\phi_j(\mathbf{x}; \mathbf{u}) = \sigma(\mathbf{u}_j^T\mathbf{x})$$

Neural Networks

- Define the basis functions $j = \{1 \ldots d\}$

$$\phi_j(\mathbf{x};\, \mathbf{u}) = \sigma(\mathbf{u}_j{}^T \mathbf{x})$$

- Put all $\mathbf{u}_j \in \mathbb{R}^{1\text{xm}}$ vectors into matrix $\mathbf{U}$

$$\phi(\mathbf{x};\, \mathbf{U}) = \sigma(\mathbf{U}\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{u}_1{}^T\mathbf{x}) \\ \sigma(\mathbf{u}_2{}^T\mathbf{x}) \\ \ldots \\ \sigma(\mathbf{u}_d{}^T\mathbf{x}) \end{bmatrix} \in \mathbb{R}^d$$

  - σ is a point-wise non-linearity acting on each vector element

Neural Networks

- Define the basis functions $j = \{1 \ldots d\}$

$$\phi_j(\mathbf{x}; \mathbf{u}) = \sigma(\mathbf{u}_j^T \mathbf{x})$$

- Put all $\mathbf{u}_j \in \mathbb{R}^{1\text{x}m}$ vectors into matrix $\mathbf{U}$

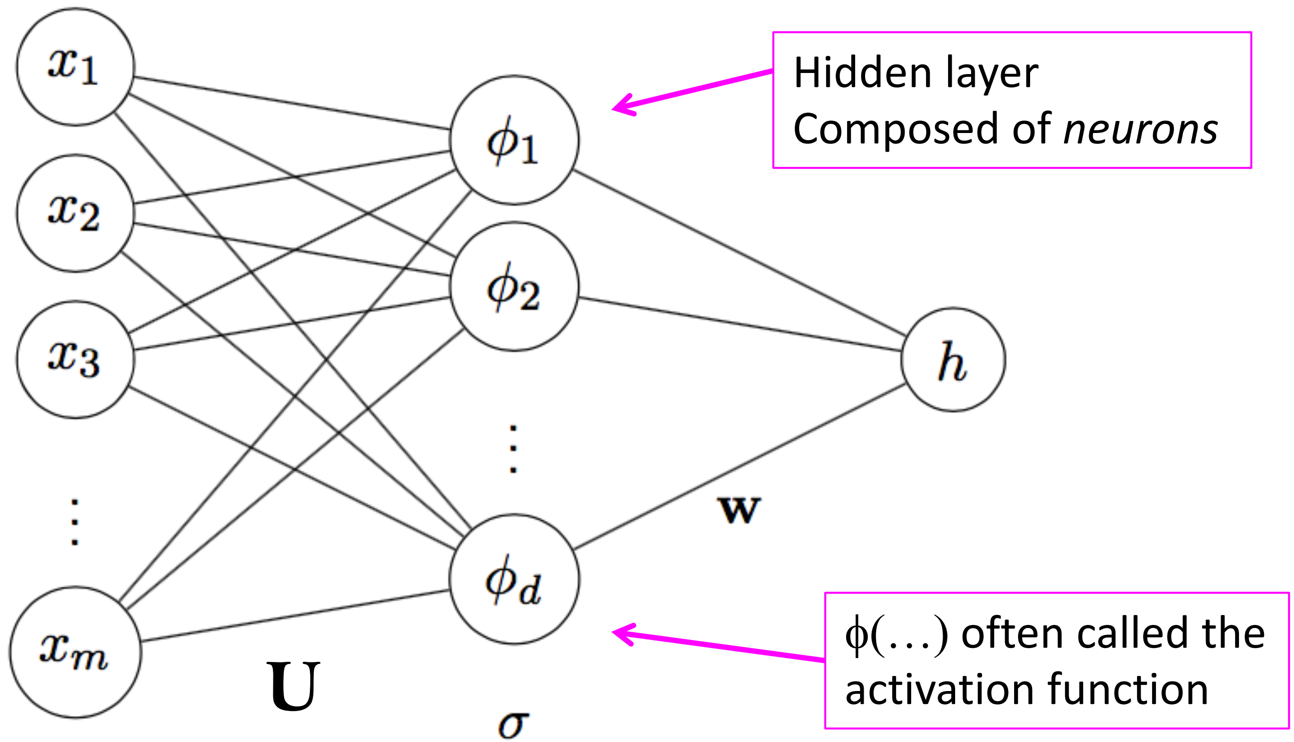
$$\phi(\mathbf{x}; \mathbf{U}) = \sigma(\mathbf{U}\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{u_1}^T\mathbf{x}) \\ \sigma(\mathbf{u_2}^T\mathbf{x}) \\ \ldots \\ \sigma(\mathbf{u_d}^T\mathbf{x}) \end{bmatrix} \in \mathbb{R}^d$$

  – σ is a point-wise non-linearity acting on each vector element

- Full model becomes

$$h(\mathbf{x}; \mathbf{w}, \mathbf{U}) = \mathbf{w}^T \phi(\mathbf{x}; \mathbf{U})$$

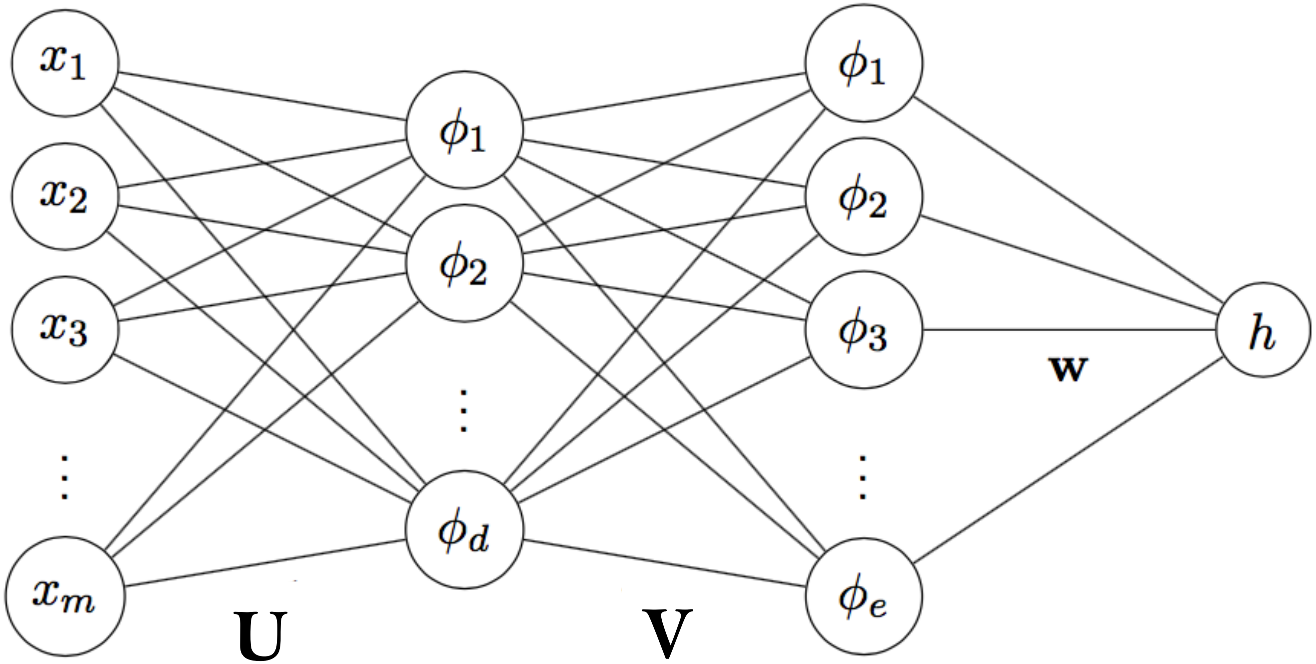
Feed Forward Neural Network



$$\phi(\mathbf{x}) = \sigma(\mathbf{U}\mathbf{x})$$

$$h(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x})$$

Multi-layer Neural Network



- Multilayer NN
  - Each layer adapts basis functions based on previous layer

Neural Network Optimization Problem

- Neural Network Model: $h(\mathbf{x}) = \mathbf{w}^T \sigma(\mathbf{U}\mathbf{x})$

- **Classification**: Cross-entropy loss function

$$p_i = p(y_i = 1 | \mathbf{x}_i) = \sigma(h(\mathbf{x}_i))$$

$$L(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)$$

Neural Network Optimization Problem

- Neural Network Model: $h(\mathbf{x}) = \mathbf{w}^T \sigma(\mathbf{U}\mathbf{x})$

- **Classification**: Cross-entropy loss function

$$p_i = p(y_i = 1|\mathbf{x}_i) = \sigma(h(\mathbf{x}_i))$$

$$L(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)$$

- **Regression**: Square error loss function

$$L(\mathbf{w}, \mathbf{U}) = \frac{1}{2} \sum_i (y_i - h(\mathbf{x}_i))^2$$

Neural Network Optimization Problem

- Neural Network Model: $h(\mathbf{x}) = \mathbf{w}^T \sigma(\mathbf{U}\mathbf{x})$

- **Classification**: Cross-entropy loss function

$$p_i = p(y_i = 1|\mathbf{x}_i) = \sigma(h(\mathbf{x}_i))$$

$$L(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)$$

- **Regression**: Square error loss function

$$L(\mathbf{w}, \mathbf{U}) = \frac{1}{2} \sum_i (y_i - h(\mathbf{x}_i))^2$$

- Minimize loss with respect to weights **w, U**

Minimizing loss with gradient descent:

- Parameter update:

$$w \leftarrow w - \eta \frac{\partial L(w, U)}{\partial w}$$

$$U \leftarrow U - \eta \frac{\partial L(w, U)}{\partial U}$$

- How to compute gradients?

Chain Rule – Symbolic Differentiation Painful!

$$L(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(\sigma(h(\mathbf{x}_i))) + (1 - y_i) \ln(1 - \sigma(h(\mathbf{x}_i)))$$

- Derivative of sigmoid: $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$
- Chain rule to compute gradient w.r.t. $\mathbf{w}$

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial \mathbf{w}} = \sum_i y_i (1 - \sigma(h(\mathbf{x}_i))) \sigma(\mathbf{U}\mathbf{x}) + (1 - y_i) \sigma(h(\mathbf{x})) \sigma(\mathbf{U}\mathbf{x}_i)$$

- Chain rule to compute gradient w.r.t. $\mathbf{u}_j$

$$\frac{\partial L}{\partial \mathbf{u}_j} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{u}_j} =$$

$$= \sum_i y_i (1 - \sigma(h(\mathbf{x}_i))) w_j \sigma(\mathbf{u}_j \mathbf{x}_i)(1 - \sigma(\mathbf{u}_j \mathbf{x}_i)) \mathbf{x}_i$$

$$+ (1 - y_i) \sigma(h(\mathbf{x}_i)) w_j \sigma(\mathbf{u}_j \mathbf{x}_i)(1 - \sigma(\mathbf{u}_j \mathbf{x}_i)) \mathbf{x}_i$$

Problem: Compute gradients of z with respect to inputs $\{x_1, x_2\}$

$$z = \sin(x_1) + x_1 x_2$$

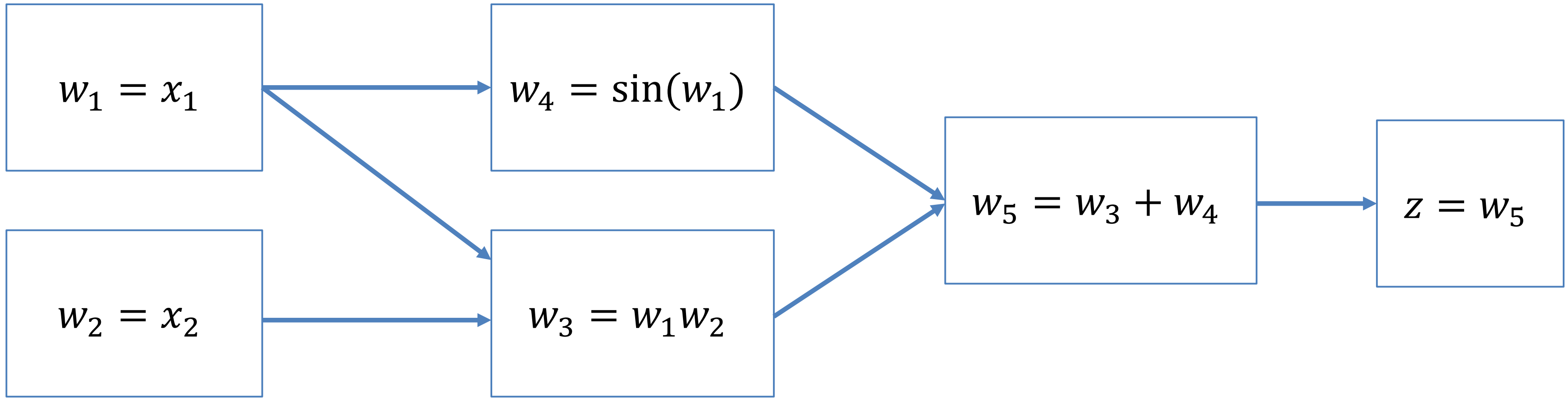
Automatic Differentiation

$$
\begin{aligned}
w_1 &= x_1 \\
w_2 &= x_2 \\
w_3 &= w_1 w_2 \\
w_4 &= \sin(w_1) \\
w_5 &= w_3 + w_4 \\
z &= w_5
\end{aligned}
$$

Problem: Compute gradients of z with respect to inputs $\{x_1, x_2\}$

$$z = \sin(x_1) + x_1 x_2$$

Organize as a computational Graph



Automatic Differentiation

$$
\begin{aligned}
w_1 &= x_1 \\
w_2 &= x_2 \\
w_3 &= w_1 w_2 \\
w_4 &= \sin(w_1) \\
w_5 &= w_3 + w_4 \\
z &= w_5
\end{aligned}
$$

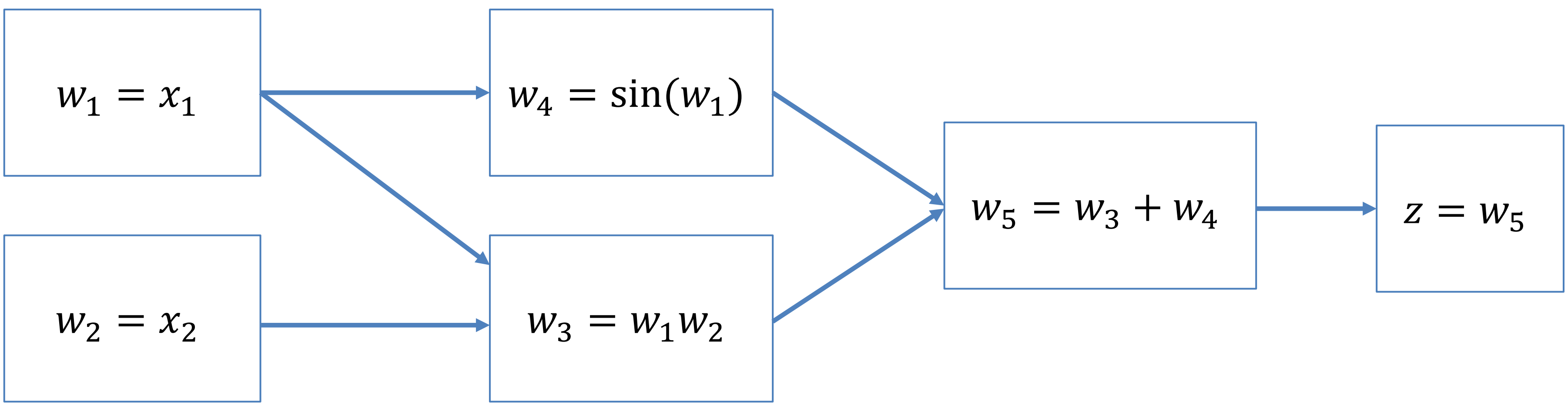
Problem: Compute gradients of z with respect to inputs $\{x_1, x_2\}$

We know the gradients of simple functions: $\sin(x)$, $x * y$, $x + y$ …

Chain rule:

$$\frac{dz}{dw_1} = \sum_{p \in parents} \frac{dz}{dw_p} \frac{dw_p}{dw_i}$$

$$
\begin{aligned}
&\frac{dw_1}{dx_1} = 1 \\
&\frac{dw_2}{dx_2} = 1 \\
&\frac{dw_3}{dw_1} = w_2 \quad \frac{dw_3}{dw_2} = w_1 \\
&\frac{dw_4}{dw_1} = \cos(w_1) \\
&\frac{dw_5}{dw_3} = 1 \quad \frac{dw_5}{dw_4} = 1
\end{aligned}
$$



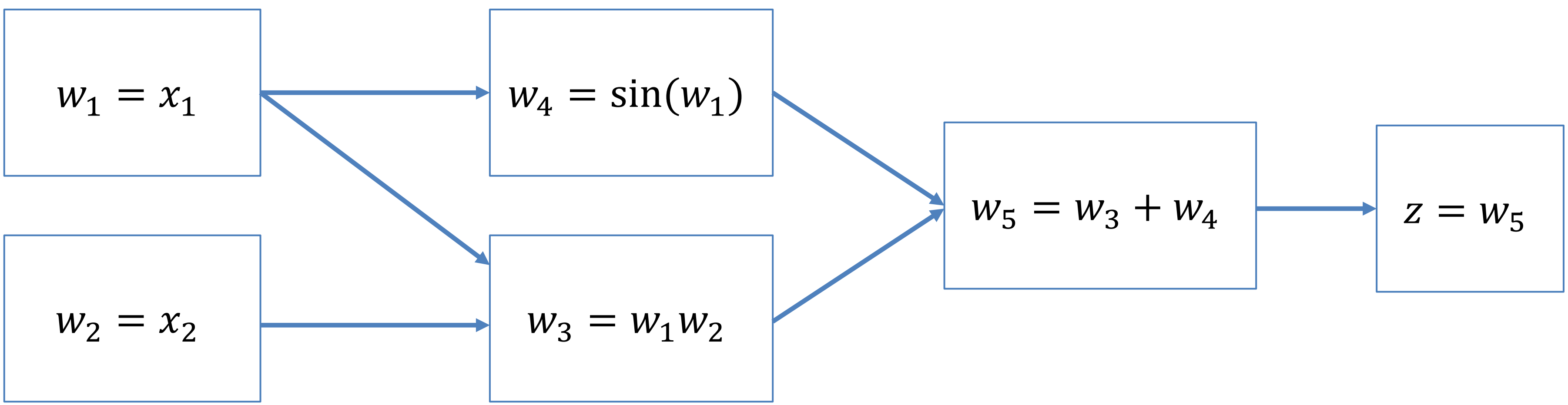
Automatic Differentiation

$$\begin{aligned} w_1 &= x_1 \\ w_2 &= x_2 \\ w_3 &= w_1 w_2 \\ w_4 &= \sin(w_1) \\ w_5 &= w_3 + w_4 \\ z &= w_5 \end{aligned}$$

Problem: Compute gradients of z with respect to inputs $\{x_1, x_2\}$

NOT going to find analytic derivative

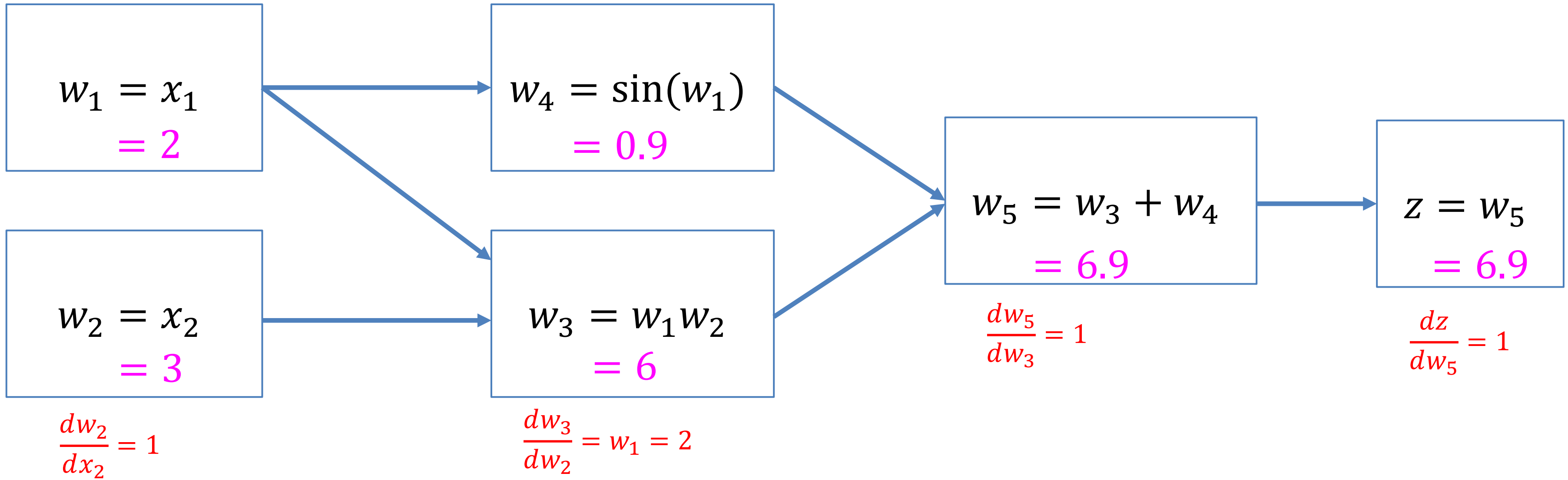
WILL find a way to compute value of gradient for a given input point



Forward Mode Automatic Differentiation

$w_1 = x_1 = 2$
$w_2 = x_2 = 3$
$w_3 = w_1 w_2 = 6$
$w_4 = \sin(w_1) = 0.9$
$w_5 = w_3 + w_4 = 6.9$
$z = w_5$

For each input, from input to output sequentially, evaluate graph and gradients and store values



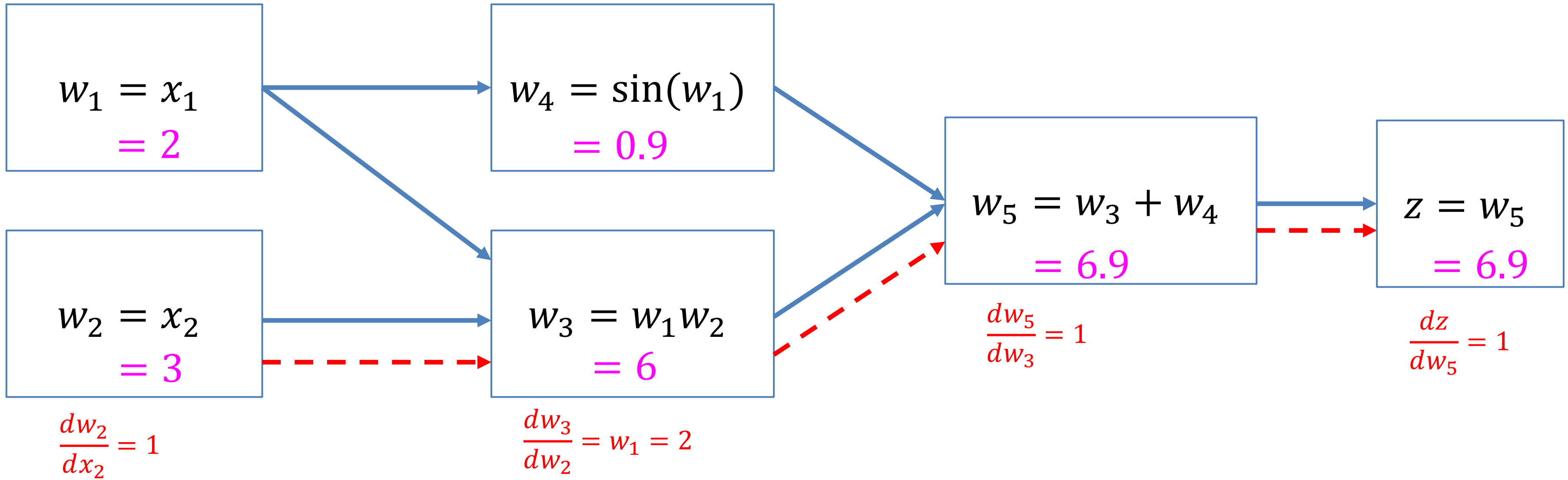
Forward Mode Automatic Differentiation

$$w_1 = x_1 = 2$$
$$w_2 = x_2 = 3$$
$$w_3 = w_1 w_2 = 6$$
$$w_4 = \sin(w_1) = 0.9$$
$$w_5 = w_3 + w_4 = 6.9$$
$$z = w_5$$

For each input, from input to output sequentially, evaluate graph and gradients and store values

Apply chain rule with multiplication

$$\frac{dz}{dx_2} = \frac{dw_2}{dx_2}\frac{dw_3}{dw_2}\frac{dw_5}{dw_3}\frac{dz}{dw_5} = 1 * 2 * 1 * 1 = 2$$



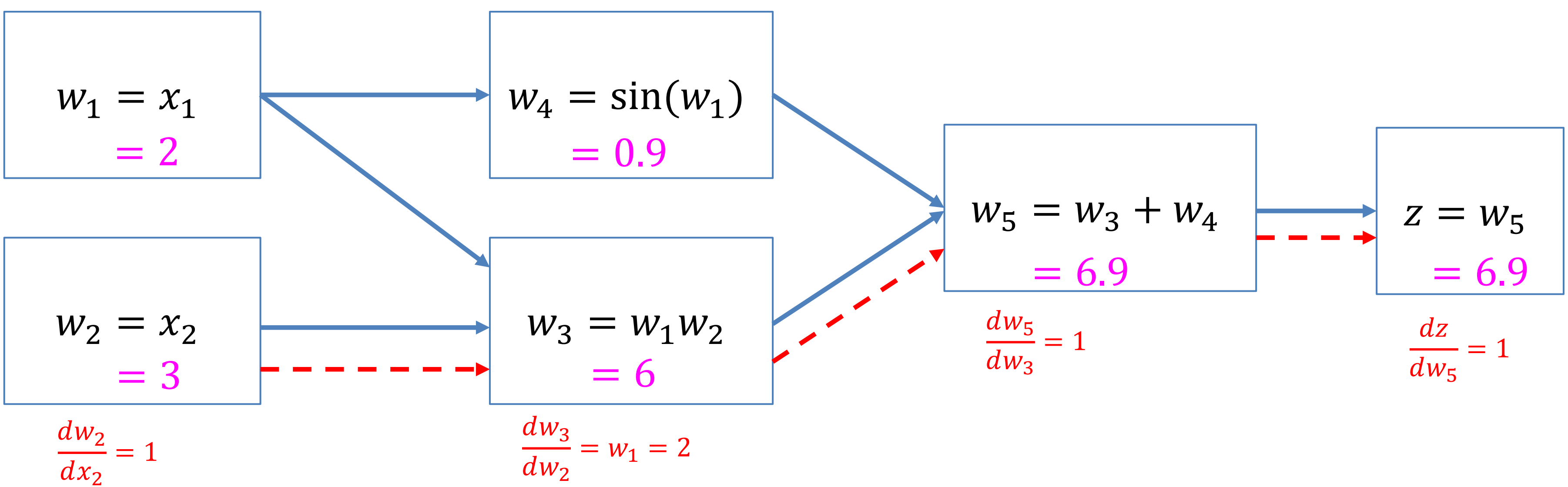
Forward Mode Automatic Differentiation

$$
\begin{aligned}
w_1 &= x_1 = 2\\
w_2 &= x_2 = 3\\
w_3 &= w_1 w_2 = 6\\
w_4 &= \sin(w_1) = 0.9\\
w_5 &= w_3 + w_4 = 6.9\\
z &= w_5
\end{aligned}
$$

Forward Mode allows us to compute the gradient of one input with respect to all the output

Jacobian $$\frac{d\boldsymbol{z}}{d\boldsymbol{x}} = \begin{pmatrix} \frac{dz_1}{dx_1} & \cdots & \frac{dz_M}{dx_1} \\ \vdots & \ddots & \vdots \\ \frac{dz_1}{dx_N} & \cdots & \frac{dz_M}{dx_N} \end{pmatrix}$$

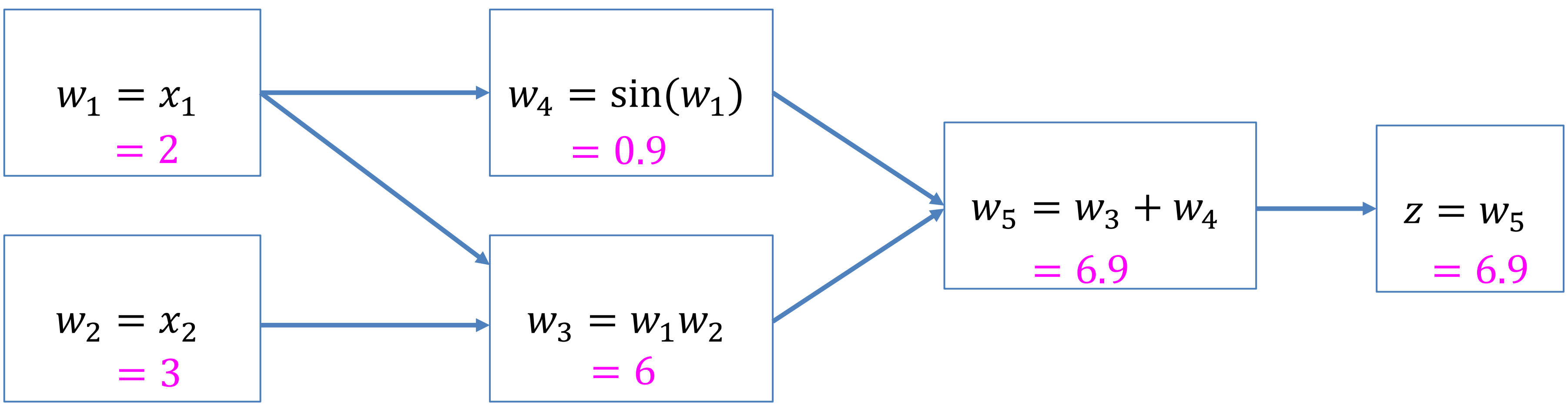
If we have 1 output (Loss) and many inputs → SLOW!



Reverse Mode Automatic Differentiation

$$
\begin{aligned}
w_1 &= x_1 = 2 \\
w_2 &= x_2 = 3 \\
w_3 &= w_1 w_2 = 6 \\
w_4 &= \sin(w_1) = 0.9 \\
w_5 &= w_3 + w_4 = 6.9 \\
z &= w_5
\end{aligned}
$$

Evaluate graph and store values

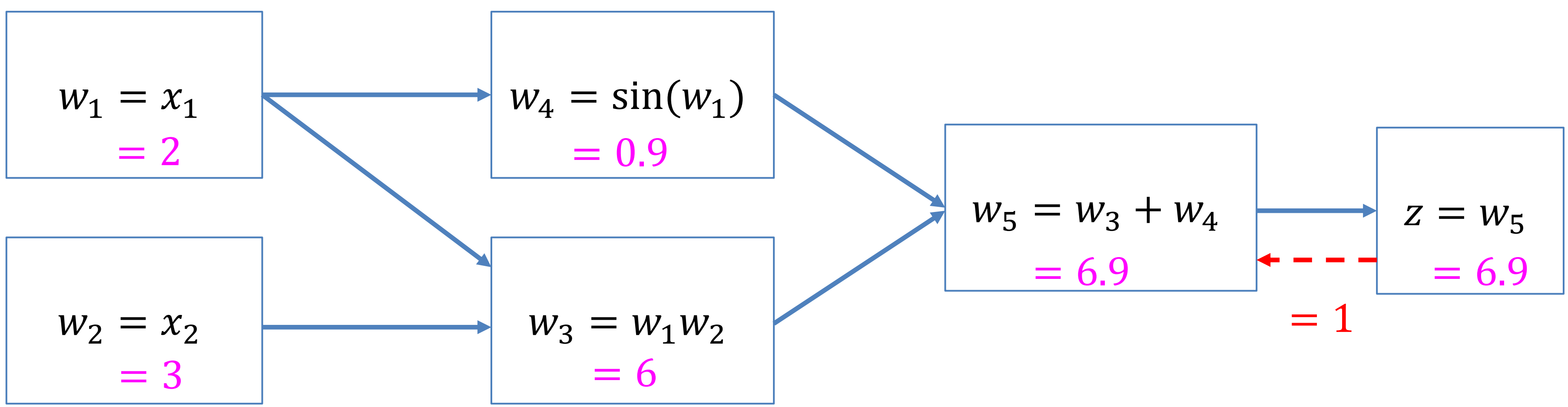


Reverse Mode Automatic Differentiation

$$w_1 = x_1 = 2$$
$$w_2 = x_2 = 3$$
$$w_3 = w_1 w_2 = 6$$
$$w_4 = \sin(w_1) = 0.9$$
$$w_5 = w_3 + w_4 = 6.9$$
$$z = w_5$$

Compute derivatives with chain rule from end to beginning:

$$\frac{dz}{dw_5} = 1$$

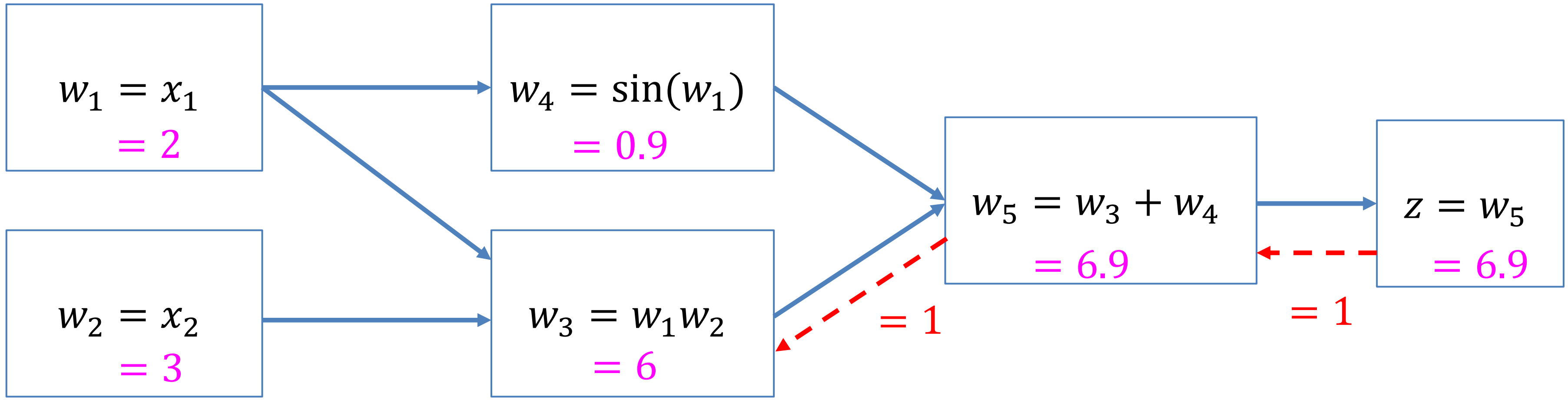


Reverse Mode Automatic Differentiation

$$w_1 = x_1 = 2$$
$$w_2 = x_2 = 3$$
$$w_3 = w_1 w_2 = 6$$
$$w_4 = \sin(w_1) = 0.9$$
$$w_5 = w_3 + w_4 = 6.9$$
$$z = w_5$$

Compute derivatives with chain rule from end to beginning:

$$\frac{dz}{dw_5} = 1$$
$$\frac{dz}{dw_3} = \frac{dz}{dw_5}\frac{dw_5}{dw_3} = 1 \times 1 = 1$$

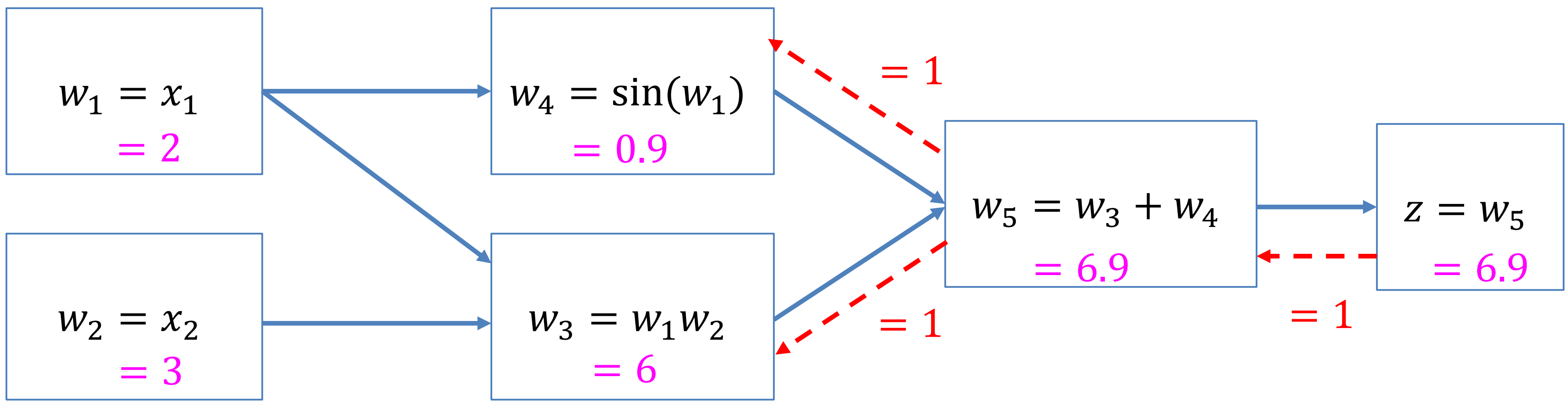


Reverse Mode Automatic Differentiation

$$w_1 = x_1 = 2$$
$$w_2 = x_2 = 3$$
$$w_3 = w_1 w_2 = 6$$
$$w_4 = \sin(w_1) = 0.9$$
$$w_5 = w_3 + w_4 = 6.9$$
$$z = w_5$$

Compute derivatives with chain rule from end to beginning:

$$\frac{dz}{dw_5} = 1$$
$$\frac{dz}{dw_3} = \frac{dz}{dw_5}\frac{dw_5}{dw_3} = 1\times 1 = 1$$
$$\frac{dz}{dw_4} = \frac{dz}{dw_5}\frac{dw_5}{dw_4} = 1\times 1 = 1$$



Reverse Mode Automatic Differentiation

$w_1 = x_1 = 2$
$w_2 = x_2 = 3$
$w_3 = w_1 w_2 = 6$
$w_4 = \sin(w_1) = 0.9$
$w_5 = w_3 + w_4 = 6.9$
$z = w_5$

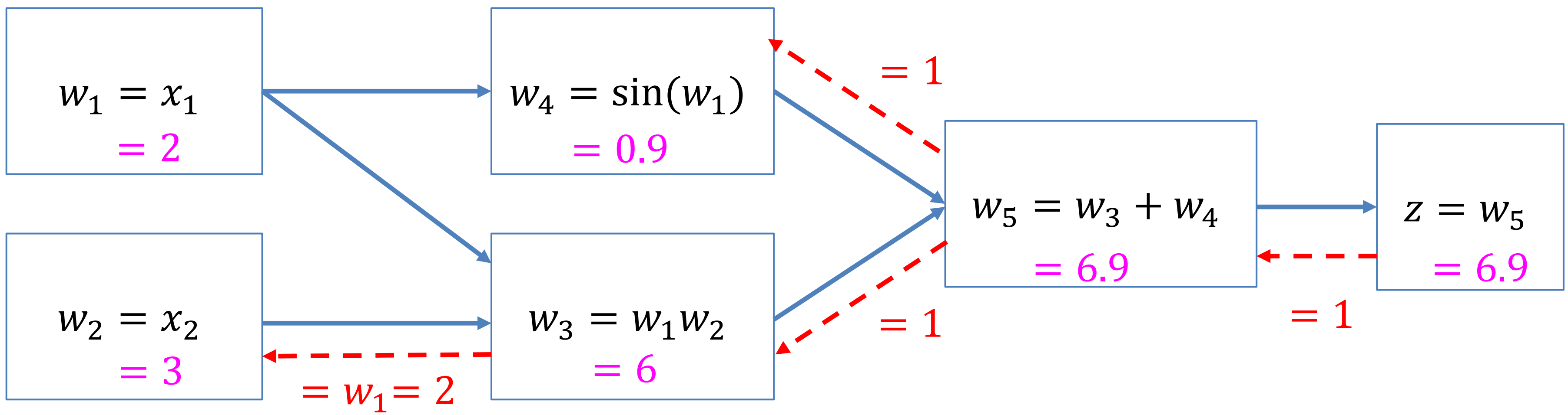
Compute derivatives with chain rule from end to beginning:

$$\frac{dz}{dw_5} = 1$$

$$\frac{dz}{dw_3} = \frac{dz}{dw_5}\frac{dw_5}{dw_3} = 1 \times 1 = 1$$

$$\frac{dz}{dw_4} = \frac{dz}{dw_5}\frac{dw_5}{dw_4} = 1 \times 1 = 1$$

$$\frac{dz}{dw_2} = \frac{dz}{dw_3}\frac{dw_3}{dw_2} = 1 \times w_1 = w_1 = 2$$



Reverse Mode Automatic Differentiation

$w_1 = x_1 = 2$
$w_2 = x_2 = 3$
$w_3 = w_1 w_2 = 6$
$w_4 = \sin(w_1) = 0.9$
$w_5 = w_3 + w_4 = 6.9$
$z = w_5$

Compute derivatives with chain rule from end to beginning:

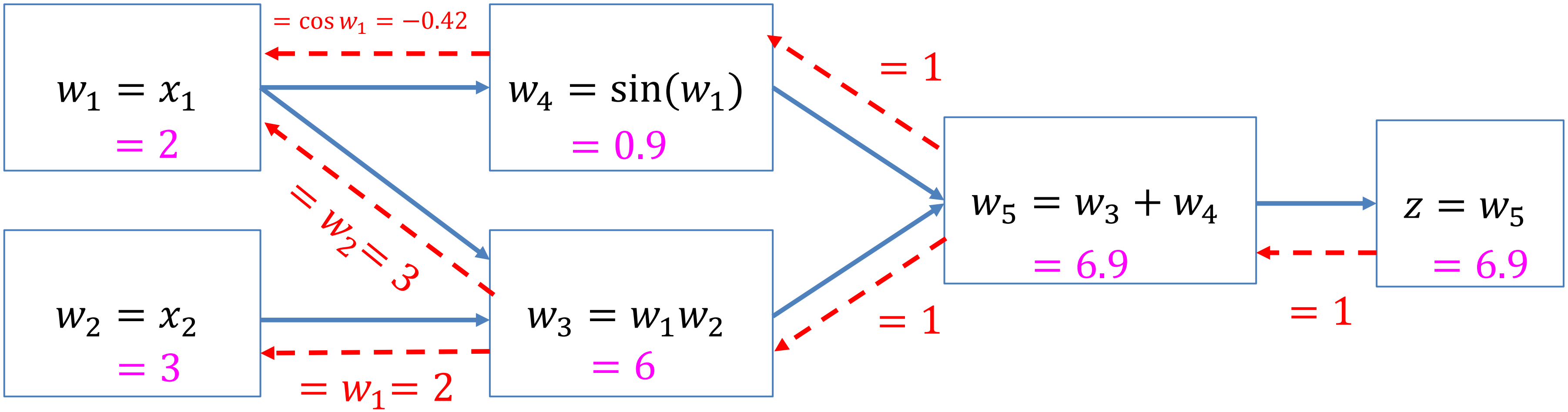
$$\frac{dz}{dw_5} = 1$$

$$\frac{dz}{dw_3} = \frac{dz}{dw_5}\frac{dw_5}{dw_3} = 1 \times 1 = 1$$

$$\frac{dz}{dw_4} = \frac{dz}{dw_5}\frac{dw_5}{dw_4} = 1 \times 1 = 1$$

$$\frac{dz}{dw_2} = \frac{dz}{dw_3}\frac{dw_3}{dw_2} = 1 \times w_1 = w_1 = 2$$

$$\frac{dz}{dw_1} = \frac{dz}{dw_4}\frac{dw_4}{dw_1} + \frac{dz}{dw_3}\frac{dw_3}{dw_1}$$
$$= \cos(w_1) + w_2 = \cos(2) + 3$$
$$= 2.58$$

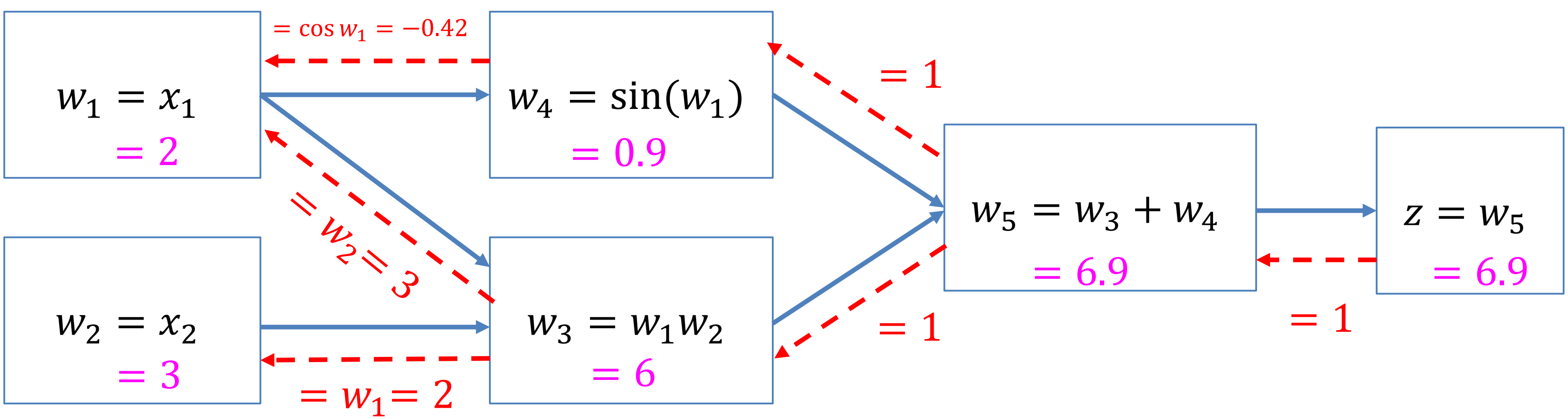


Reverse Mode Automatic Differentiation

$$
\begin{aligned}
w_1 &= x_1 = 2 \\
w_2 &= x_2 = 3 \\
w_3 &= w_1 w_2 = 6 \\
w_4 &= \sin(w_1) = 0.9 \\
w_5 &= w_3 + w_4 = 6.9 \\
z &= w_5
\end{aligned}
$$

For each output, can compute the gradient w.r.t. all inputs in one pass!

Jacobian $\frac{d\boldsymbol{z}}{d\boldsymbol{x}} = \begin{pmatrix} \frac{dz_1}{dx_1} & \cdots & \frac{dz_M}{dx_1} \\ \vdots & \ddots & \vdots \\ \frac{dz_1}{dx_N} & \cdots & \frac{dz_M}{dx_N} \end{pmatrix}$



Backpropagation

- Loss function composed of layers of nonlinearity

$$L\left(\phi^N\left(\ldots\phi^1(x)\right)\right)$$

Backpropagation

- Loss function composed of layers of nonlinearity

$$L\left(\phi^N\left(\ldots\phi^1(x)\right)\right)$$

- Forward step (f-prop)
  - Compute and save intermediate computations

$$\phi^N\left(\ldots\phi^1(x)\right)$$

Backpropagation

- Loss function composed of layers of nonlinearity

$$L\left(\phi^N\left(\ldots\phi^1(x)\right)\right)$$

- Forward step (f-prop)
  - Compute and save intermediate computations

$$\phi^N\left(\ldots\phi^1(x)\right)$$

- Backward step (b-prop) $\dfrac{\partial L}{\partial \phi^a} = \sum_j \dfrac{\partial \phi_j^{(a+1)}}{\partial \phi_j^a} \dfrac{\partial L}{\partial \phi_j^{(a+1)}}$

Backpropagation

- Loss function composed of layers of nonlinearity

$$L\left(\phi^N\left(\ldots\phi^1(x)\right)\right)$$

- Forward step (f-prop)
  - Compute and save intermediate computations

$$\phi^N\left(\ldots\phi^1(x)\right)$$

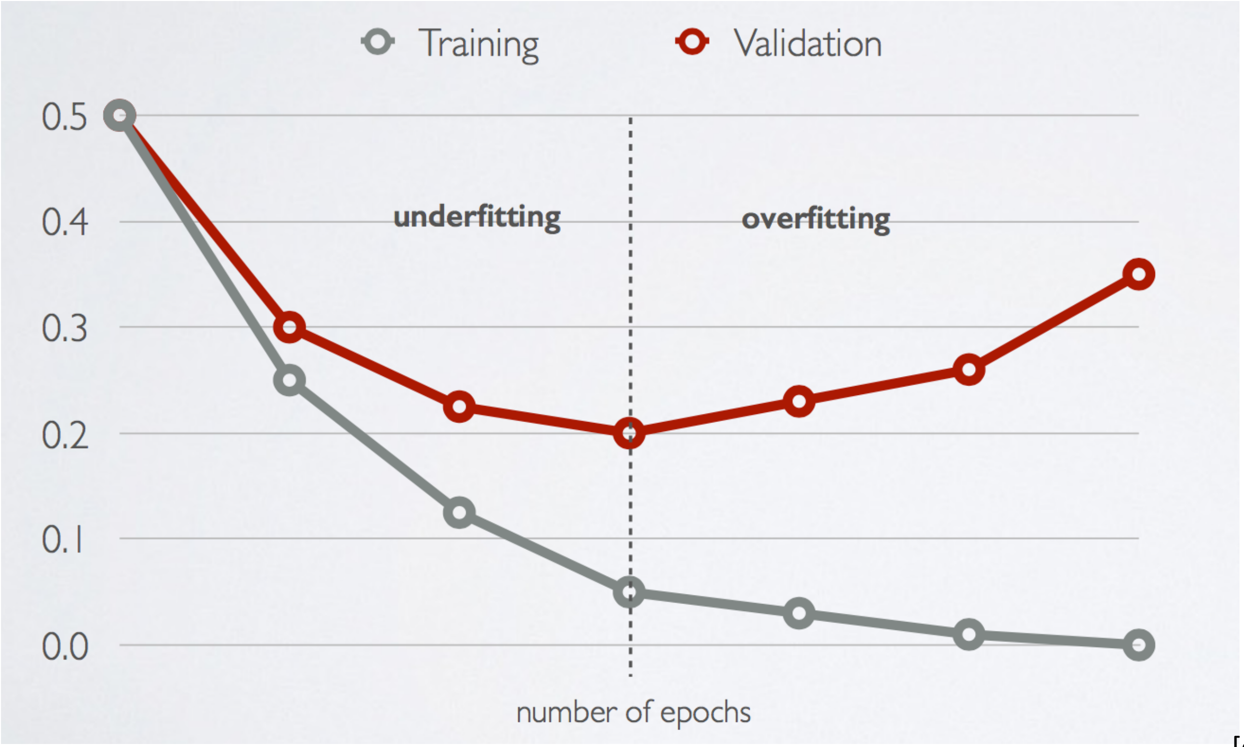
- Backward step (b-prop) $$\frac{\partial L}{\partial \phi^a} = \sum_j \frac{\partial \phi_j^{(a+1)}}{\partial \phi_j^a} \frac{\partial L}{\partial \phi_j^{(a+1)}}$$

- Compute parameter gradients $$\frac{\partial L}{\partial \mathbf{w}^a} = \sum_j \frac{\partial \phi_j^a}{\partial \mathbf{w}^a} \frac{\partial L}{\partial \phi_j^a}$$

Training

- Repeat gradient update of weights to reduce loss
  - Each iteration through dataset is called an epoch

- Use validation set to examine for overtraining, and determine when to stop training

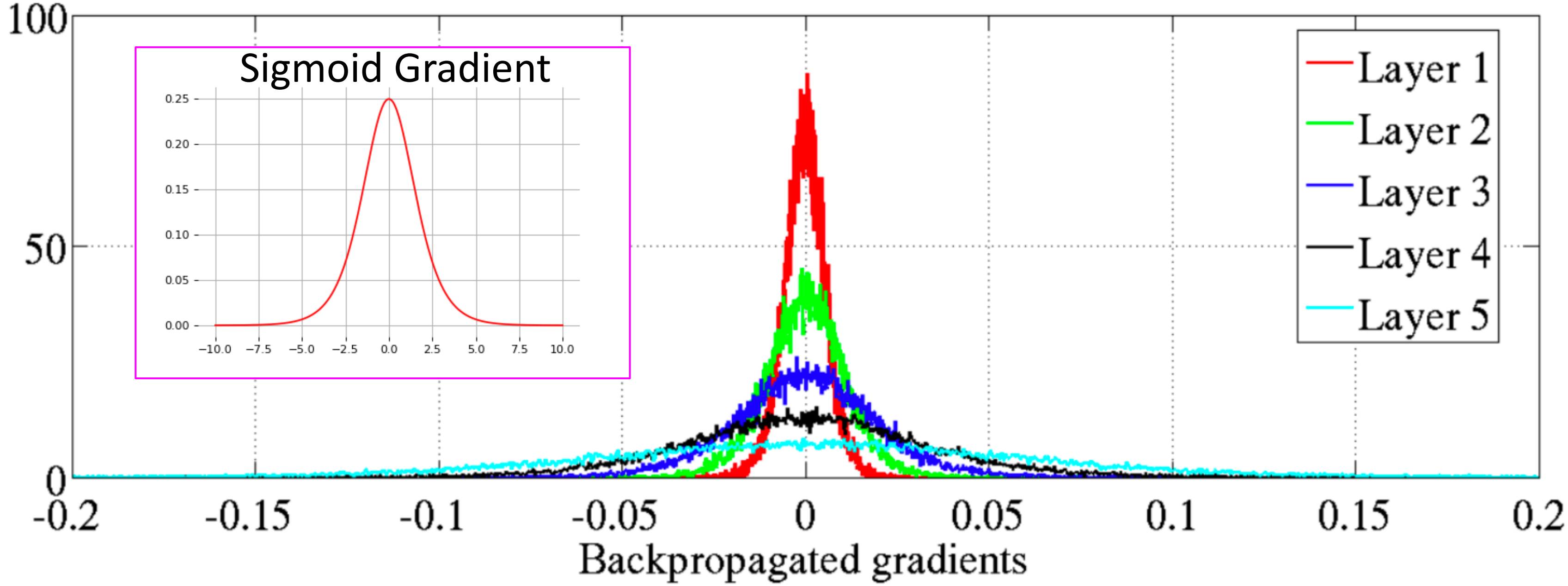


[graphic from H. Larochelle]

Vanishing Gradients

- Major challenge in DL: Vanishing Gradients

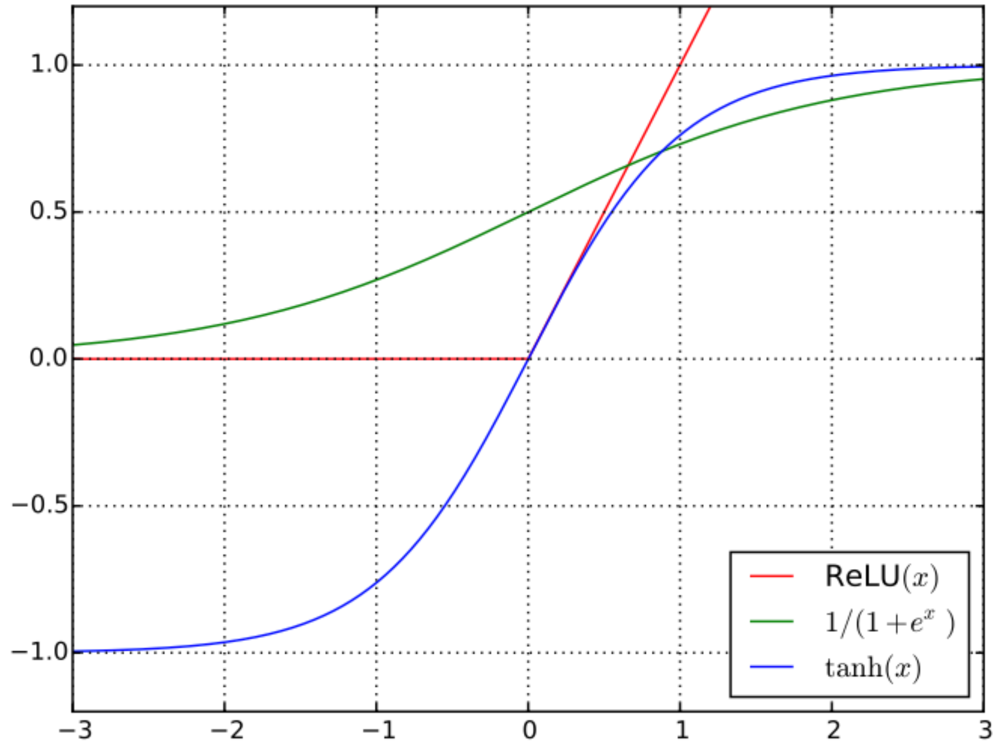
- Small gradients slow down / block, stochastic gradient descent → Limits ability to learn!



Backpropagated gradients normalized histograms (Glorot and Bengio, 2010). Gradients for layers far from the output vanish to zero.

Slide credit: [G. Louppe]

Activation Functions



- **Vanishing gradient problem**
  - Derivative of sigmoid:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

  - Nearly 0 when x is far from 0!
  - Can make gradient descent hard!

- **Rectified Linear Unit (ReLU)**
  - ReLU(x) = max{0, x}
  - Derivative is constant!

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 1 & when\ x > 0 \\ 0 & otherwise \end{cases}$$

  - ReLU gradient doesn't vanish

Neural Network Decision Boundaries

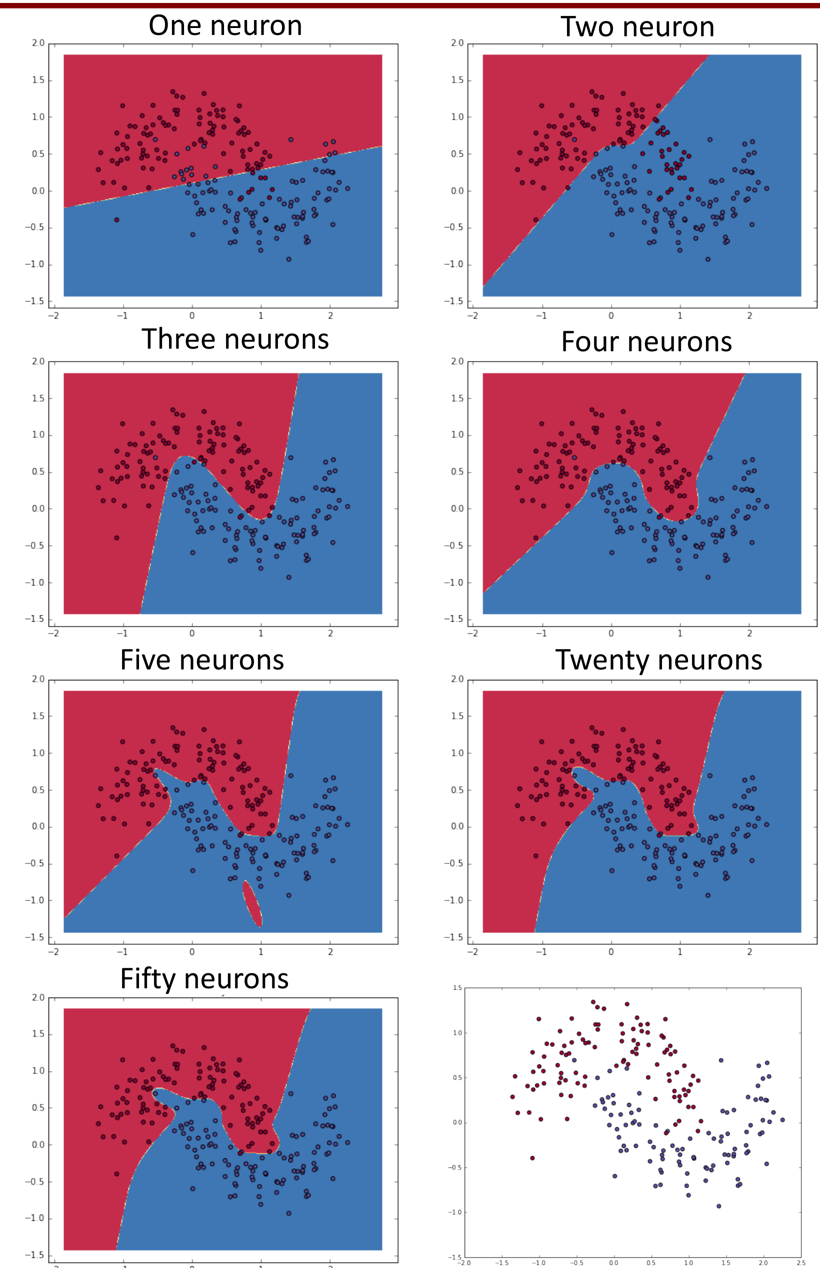
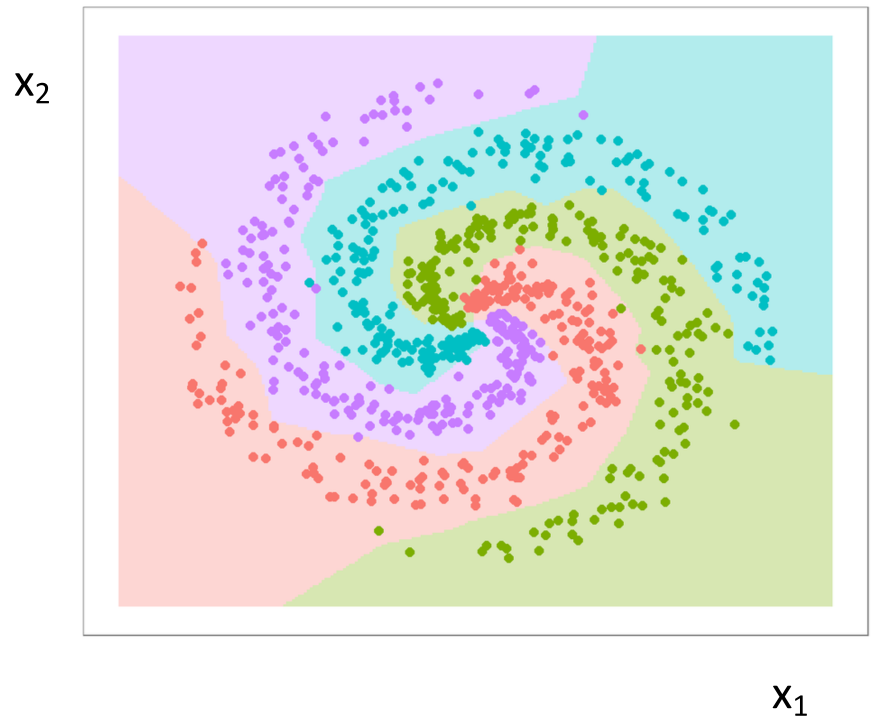


Image source

4-class classification
2-hidden layer NN
ReLU activations
L2 norm regularization



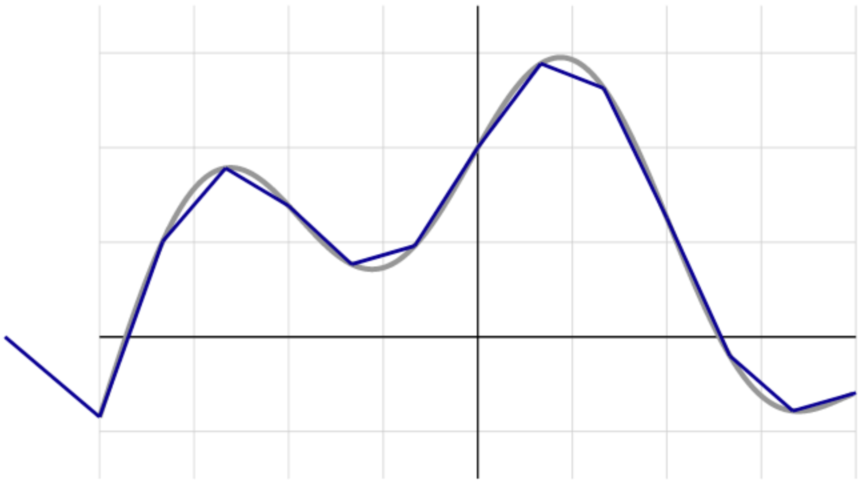
2-class classification
1-hidden layer NN
L2 norm regularization

Image source

Universal approximation theorem

- Feed-forward neural network with a single hidden layer containing a finite number of non-linear neurons (ReLU, Sigmoid, and others) can approximate continuous functions arbitrarily well on a compact space of $\mathbb{R}^n$

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$



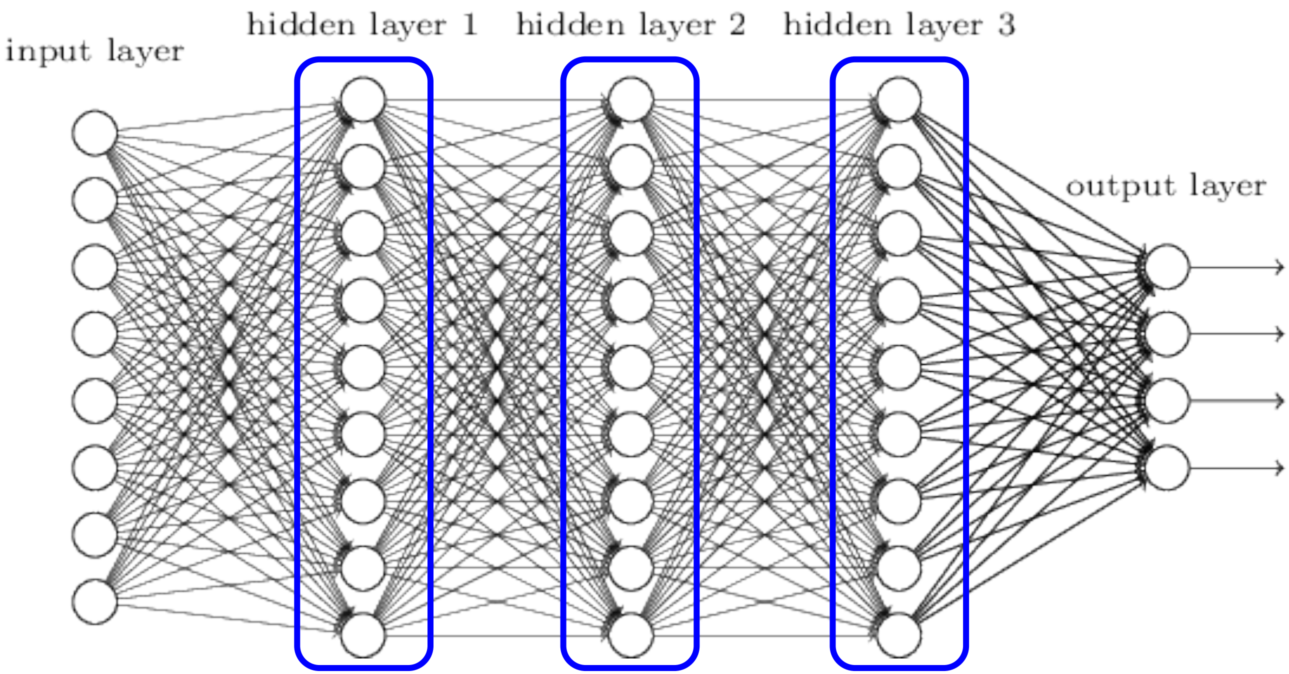
Fleuret, [Deep Learning Course]

Universal approximation theorem

- Feed-forward neural network with a single hidden layer containing a finite number of non-linear neurons (ReLU, Sigmoid, and others) can approximate continuous functions arbitrarily well on a compact space of $\mathbb{R}^n$

- NOTE!
  - A better approximation requires a larger hidden layer and this theorem says nothing about the relation between the two.
  - We can make training error as low as we want by using a larger hidden layer. Result states nothing about test error
  - Doesn't say how to find the parameters for this approximation

Deep Neural Networks



- As data complexity grows, need exponentially large number of neurons in a single-hidden-layer network to capture all structure in data
- Deep neural networks *factorize the learning* of structure in data across many layers
- Difficult to train, only recently possible with large datasets, fast computing (GPU / TPU) and new training procedures / network structures

Neural Network Zoo

- Structure of the networks, and the node connectivity can be adapted for problem at hand

- Moving inductive bias from feature engineering to model design

  - *Inductive bias*:
    Knowledge about the problem

  - *Feature engineering*:
    Hand crafted variables

  - *Model design*:
    The data representation and the structure of the machine learning model / network

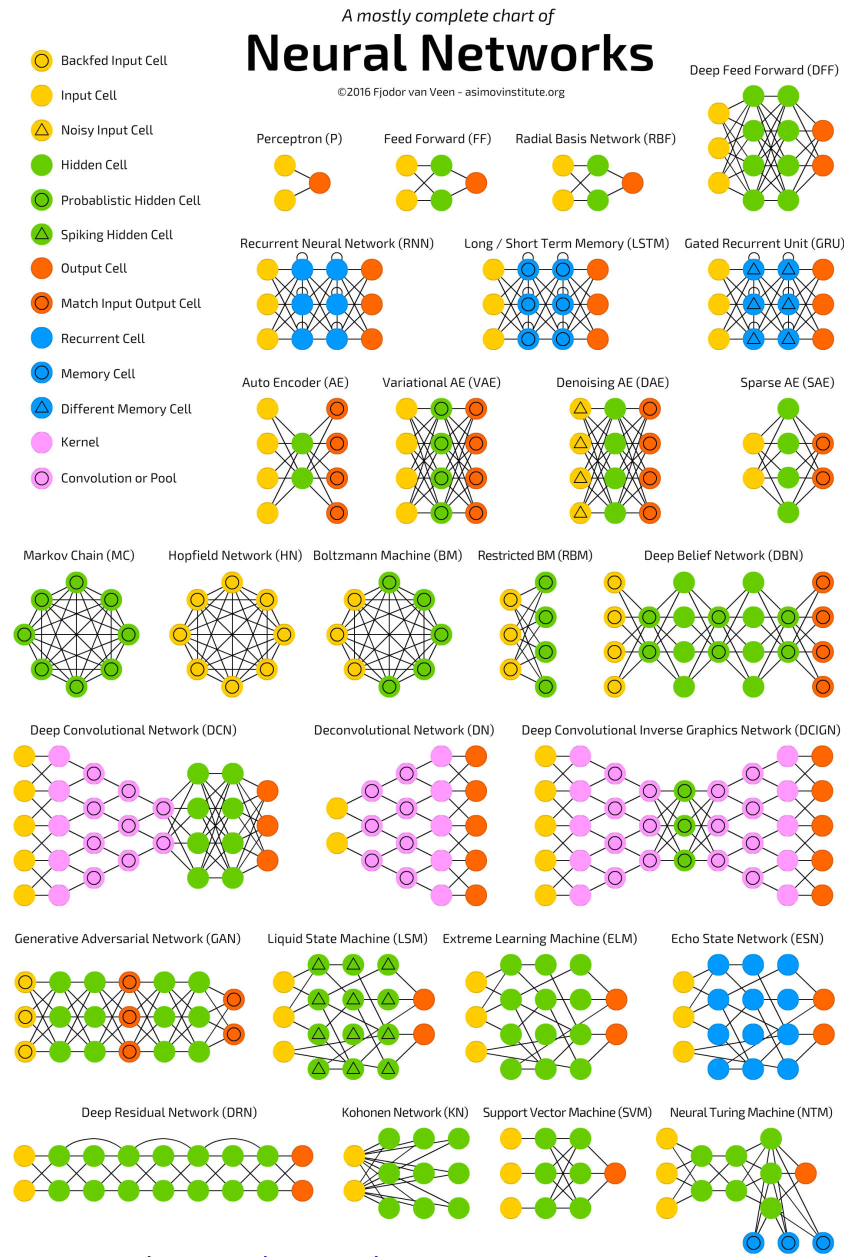


Image credit: [neural-network-zoo]

Neural Network Zoo – “Optimization” Perspective

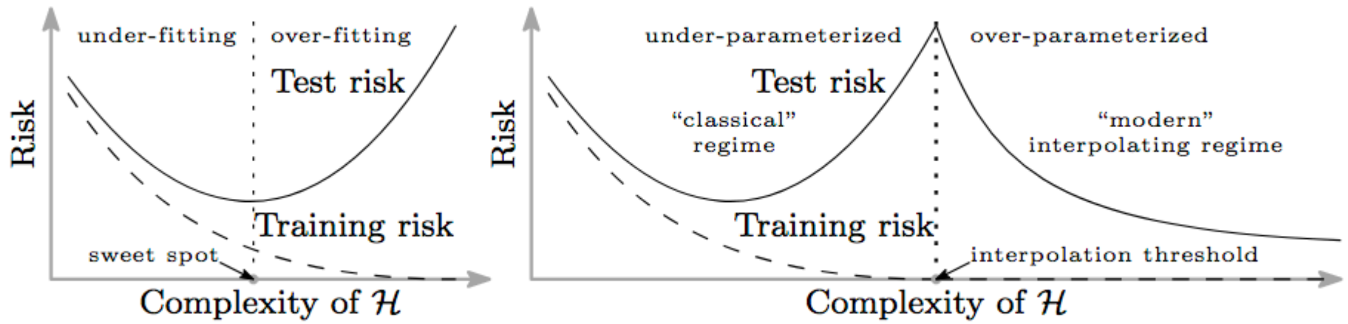
- A single layer network may need a width exponential in D to approximate a depth-D network’s output
  - Simplified version of Telgarsky (2015, 2016)

Neural Network Zoo – "Optimization" Perspective

- A single layer network may need a width exponential in D to approximate a depth-D network's output
  - Simplified version of Telgarsky (2015, 2016)

- Over-parametrizing a deep model often improves test performance, contrary to bias-variance tradeoff prediction

Belkin et. al. 2018



(a) U-shaped "bias-variance" risk curve (b) "double descent" risk curve

Figure 1: Curves for training risk (dashed line) and test risk (solid line). (a) The classical *U-shaped risk curve* arising from the bias-variance trade-off. (b) The *double descent risk curve*, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high complexity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

Neural Network Zoo – "Optimization" Perspective

- A single layer network may need a width exponential in D to approximate a depth-D network's output
  - Simplified version of Telgarsky (2015, 2016)

- Over-parametrizing a deep model often improves test performance, contrary to bias-variance tradeoff prediction

  - But we must control that:
    - Gradients don't vanish
    - Gradient amplitude is homogeneous across network
    - Gradients are under control when weights change

Neural Network Zoo – "Optimization" Perspective

- A single layer network may need a width exponential in D to approximate a depth-D network's output
  - Simplified version of Telgarsky (2015, 2016)

- Over-parametrizing a deep model often improves test performance, contrary to bias-variance tradeoff prediction

- Major part of deep learning is **trying to choose the right function…**

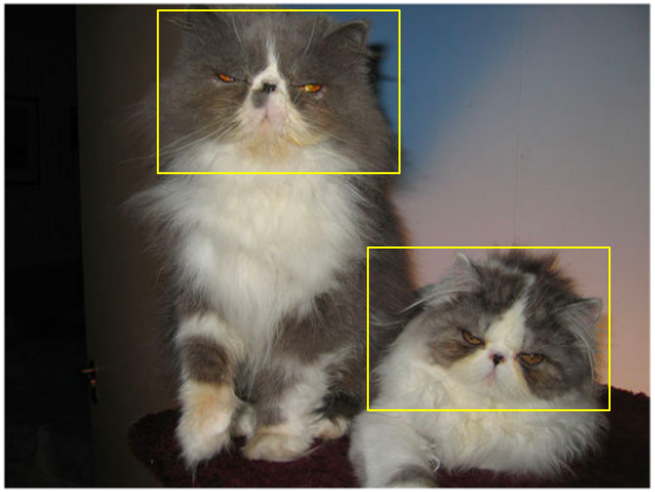
  … instead of trying to improve training with regularization and new optimizers

  - Need to **make gradient descent work**, even at the cost of a substantially engineering the model

Convolutional Neural Networks

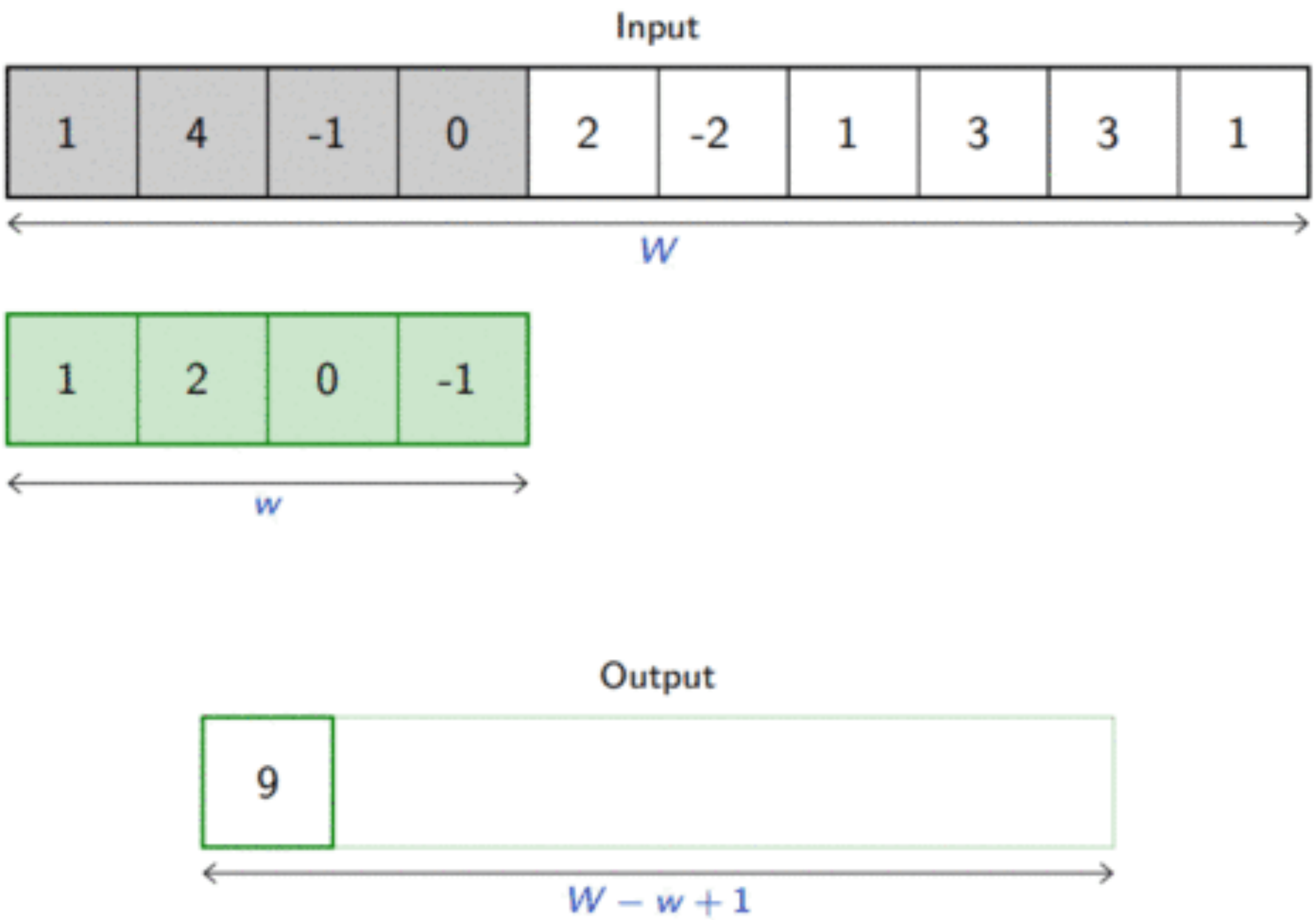
Convolutional Neural Networks

- When the structure of data includes "invariance to translation", a representation meaningful at a certain location can / should be used everywhere



- Covolutional layers build on this idea, that the same "local" transformation is applied everywhere and preserves the signal structure

1D Convolutional Layer Example



1D Convolutional Layers

- **Data:** $x \in \mathbb{R}^M$
- **Convolutional kernel of width k:** $u \in \mathbb{R}^k$

- Convolution $x \circledast u$ is vector of size M-k+1

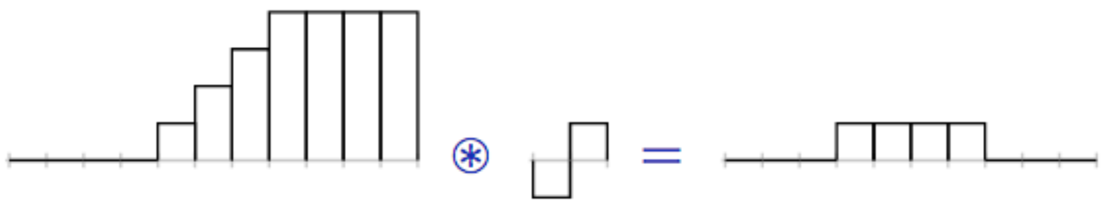
$$(x \circledast \mathrm{u})_i = \sum_{b=0}^{k-1} x_{i+b} u_b$$

- Scan across data and multiply by kernel elements

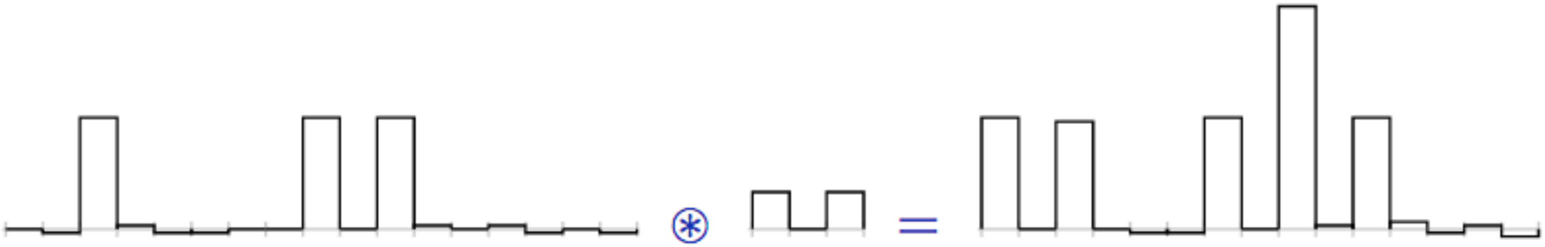
Convolutional Filters

Convolution can implement in particular differential operators, *e.g.*

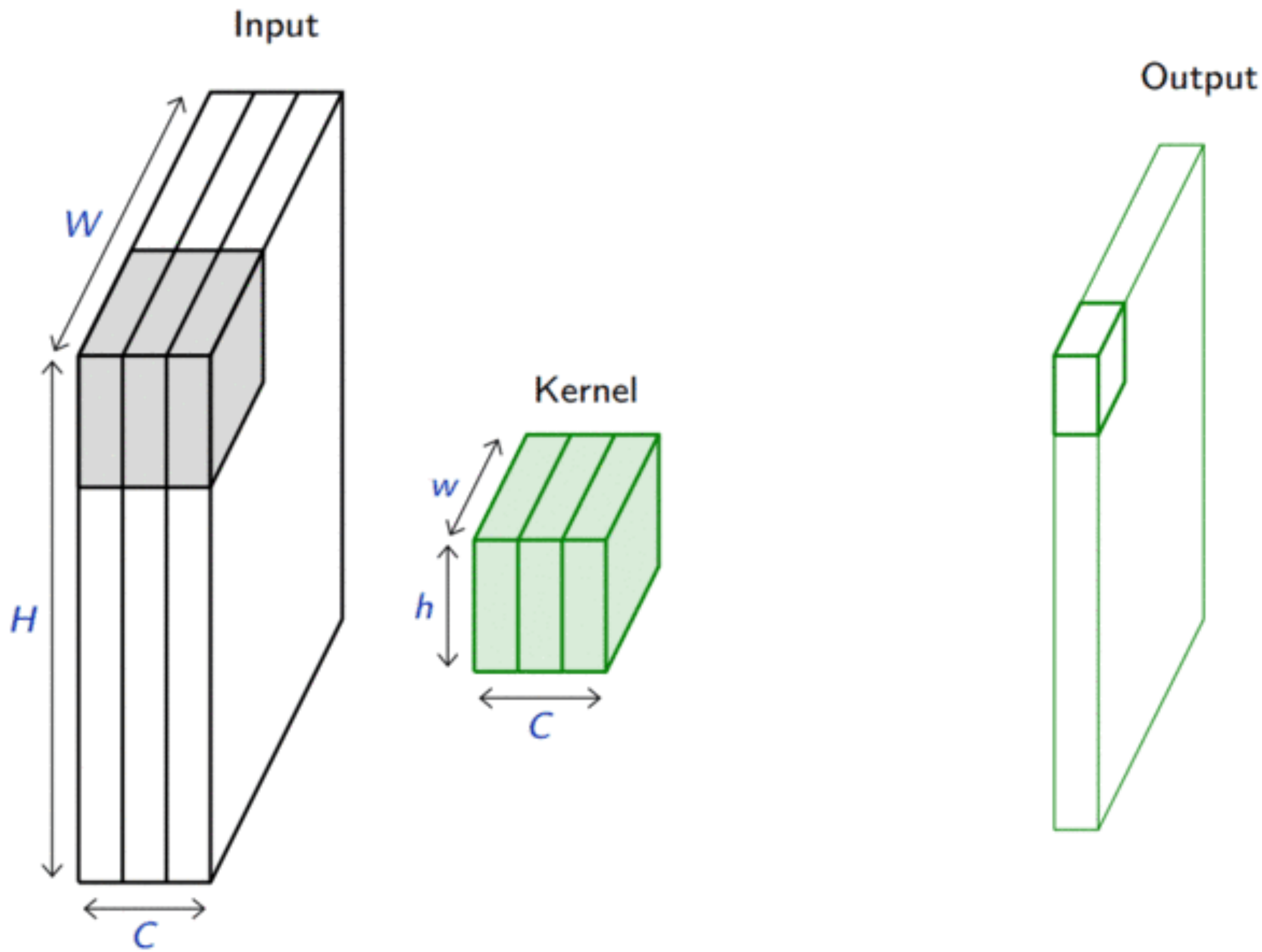
$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



or crude "template matcher", *e.g.*

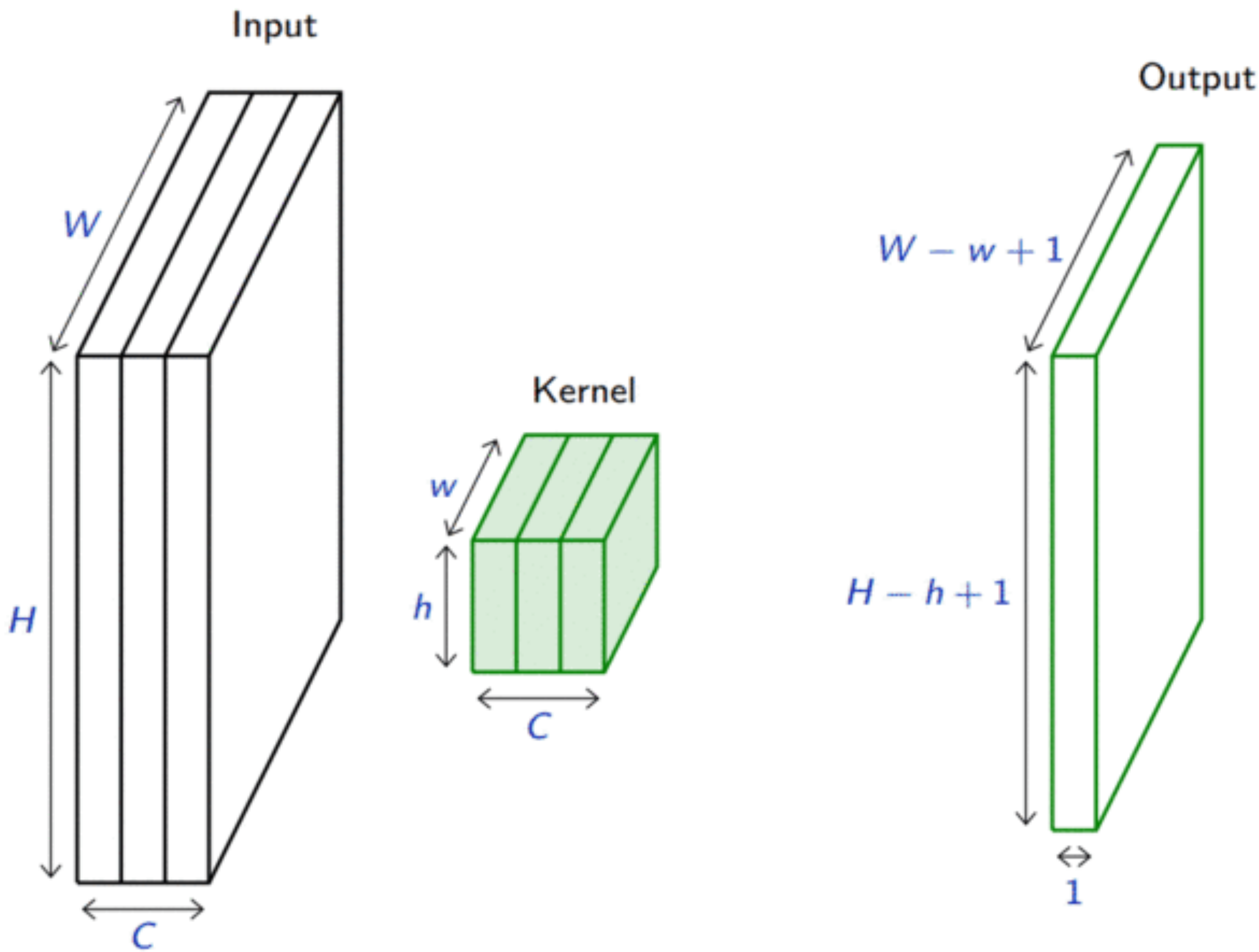


2D Convolution Over Multiple Channels



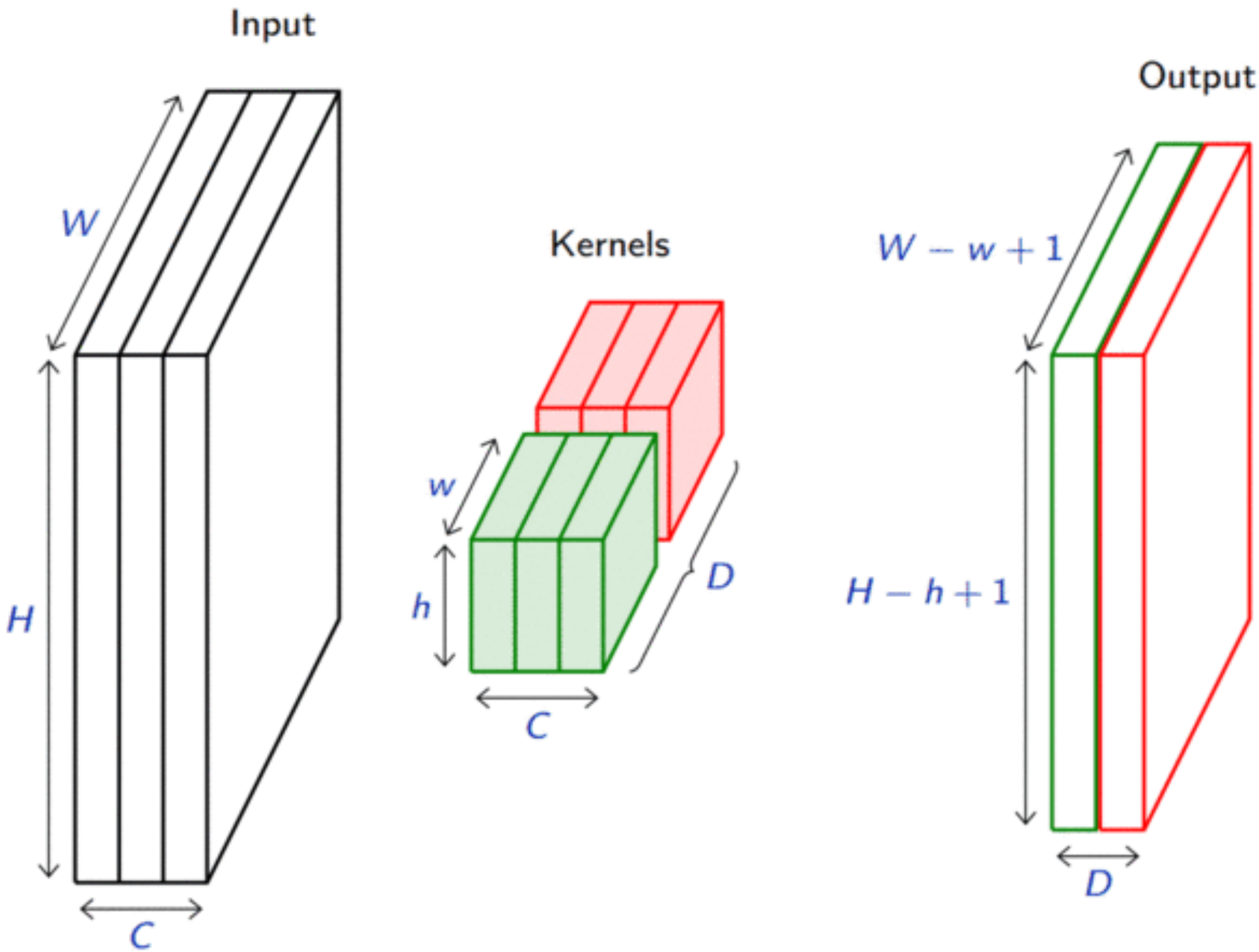
Fleuret, Deep Learning Course

2D Convolution Over Multiple Channels



Fleuret, [Deep Learning Course]()

2D Convolution Over Multiple Channels



Fleuret, [Deep Learning Course]

2D Convolutional Layer

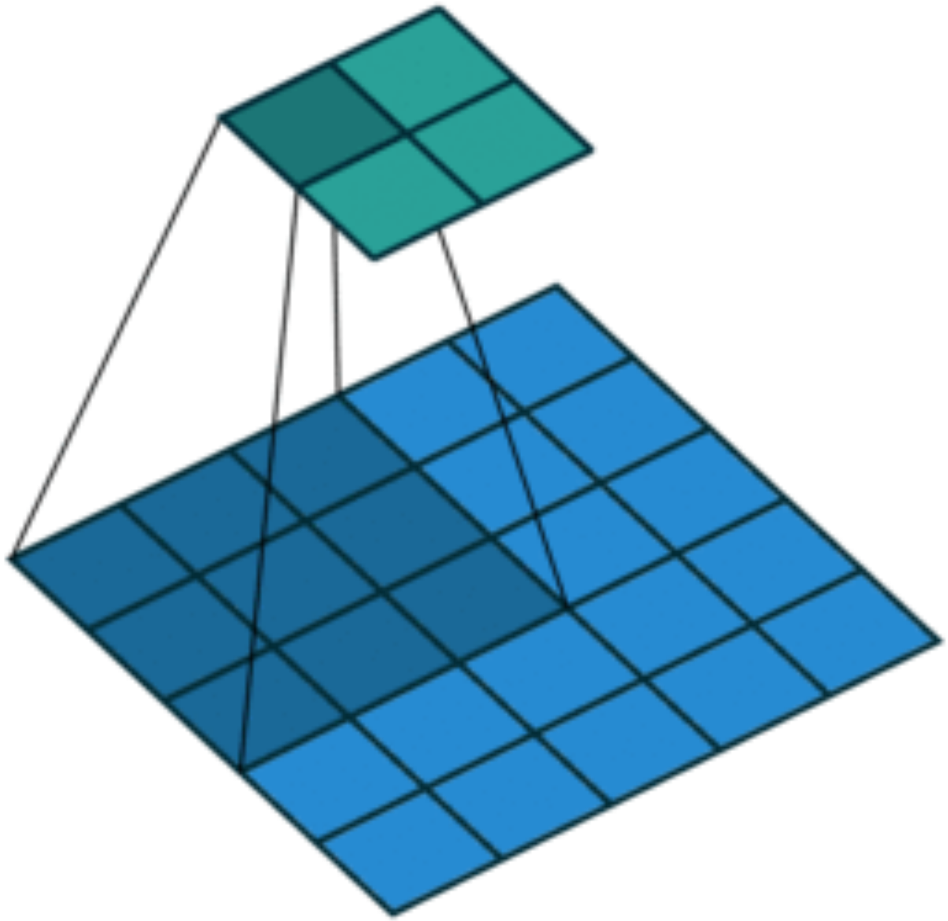
- Input data (tensor) **x** of size $C \times H \times W$
  - C channels (e.g. RGB in images)

- Learnable Kernel **u** of size $C \times h \times w$
  - The size $h \times w$ is the *receptive field*

$$(\boldsymbol{x} \circledast \boldsymbol{u})_{i,j} = \sum_{c=0}^{C-1} (\boldsymbol{x}_c \circledast \boldsymbol{u}_c)_{i,j} = \sum_{c=0}^{C-1} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \boldsymbol{x}_{c,n+i,m+j} \boldsymbol{u}_{c,n,m}$$

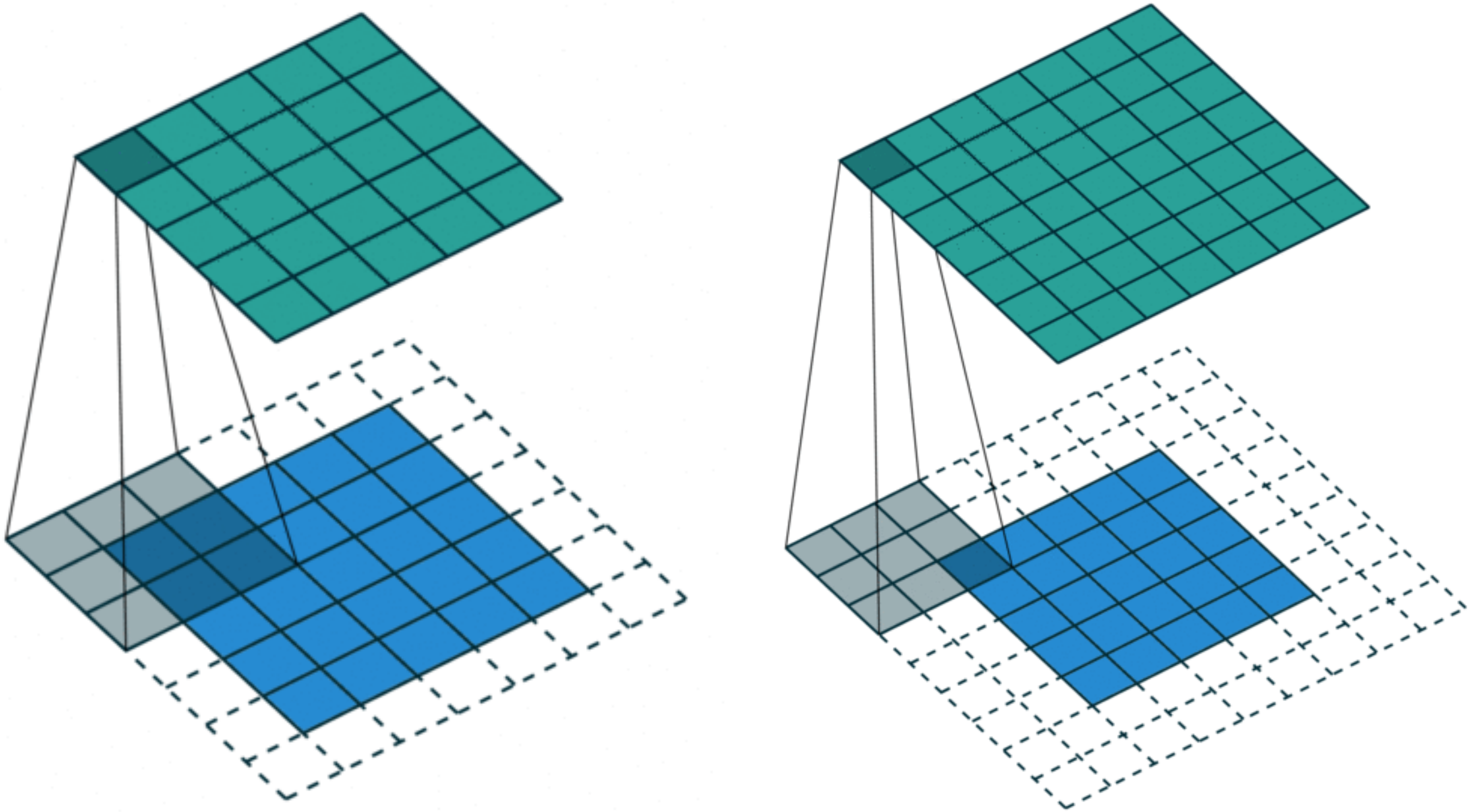
- Output size $(H - h + 1) \times (W - w + 1)$ for each kernel
  - Often called *Activation Map* or *Output Feature Map*

Stride – Step Size When Moving Kernel Across Input



Fleuret, Deep Learning Course

Padding – Size of Zero Frame Around Input



Fleuret, [Deep Learning Course]

Shared Weights: Economic and Equivariant

- Parameters are *shared* by each neuron producing an output in the activation map

- Dramatically reduces number of weights needed to produce an activation map
  - Data: $256 \times 256 \times 3$ RGB image
  - Kernel: $3 \times 3 \times 3 \rightarrow 27$ weights
  - Fully connected layer:
    - $256 \times 256 \times 3$ inputs → $256 \times 256 \times 3$ outputs → $O(10^{10})$ weights

Shared Weights: Economic and Equivariant

- Parameters are *shared* by each neuron producing an output in the activation map

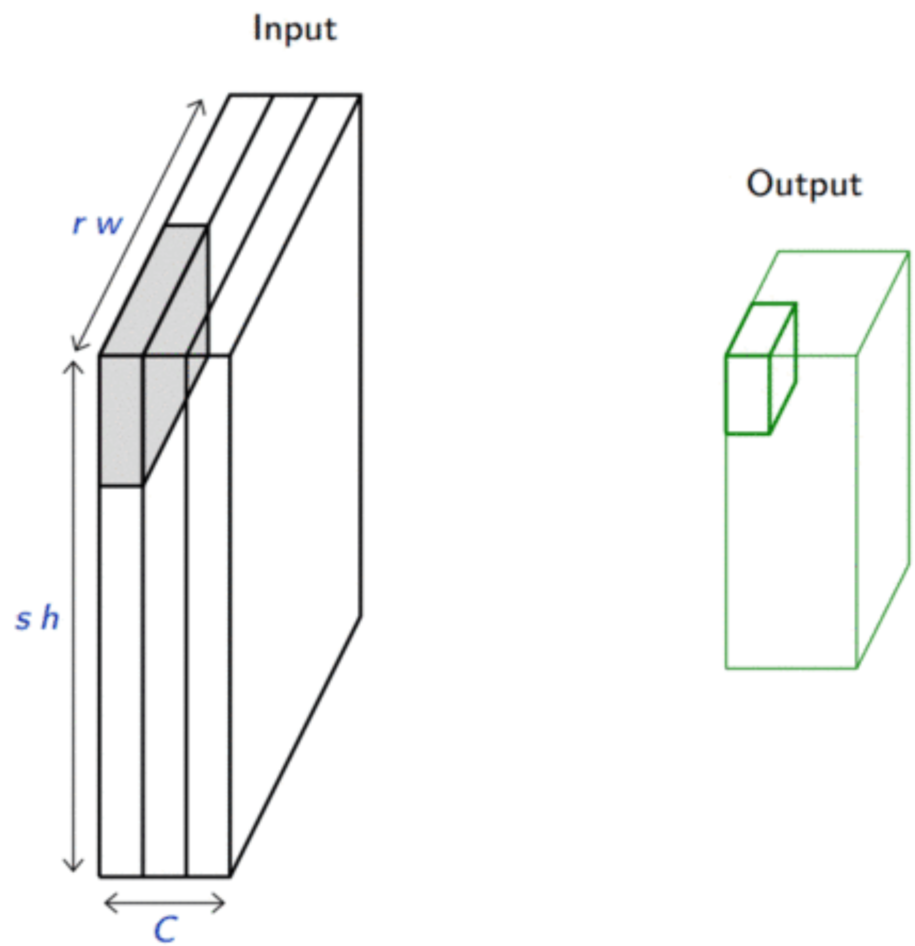
- Dramatically reduces number of weights needed to produce an activation map

- Convolutional layer does pattern matching at any location → Equivariant to translation

Y. LeCun et. al. [1998]

Pooling

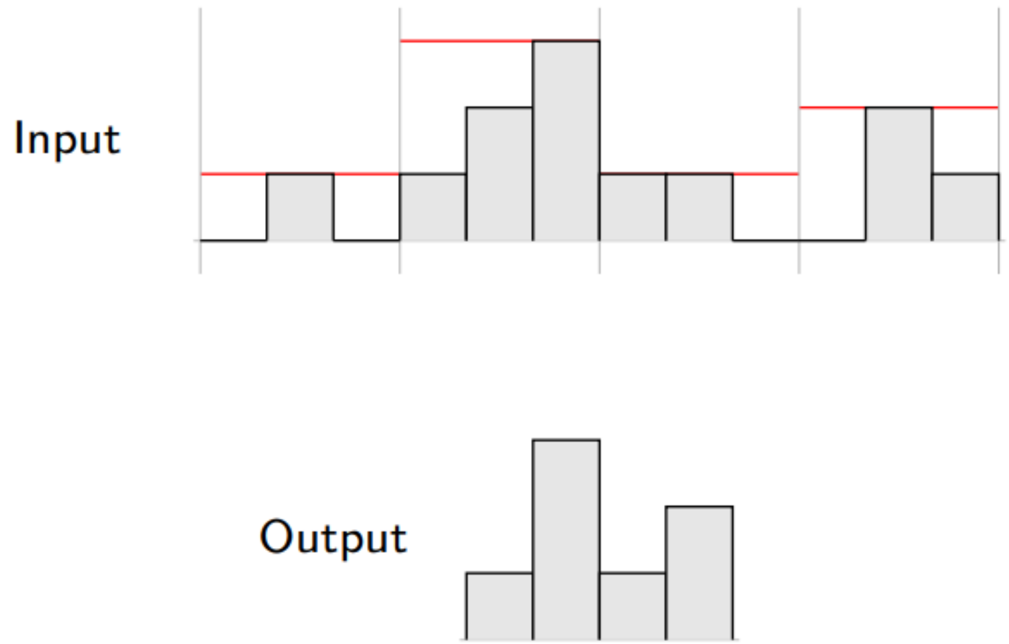
- In each channel, find *max* or *average* value of pixels in a pooling area of size $h \times w$



Fleuret, Deep Learning Course

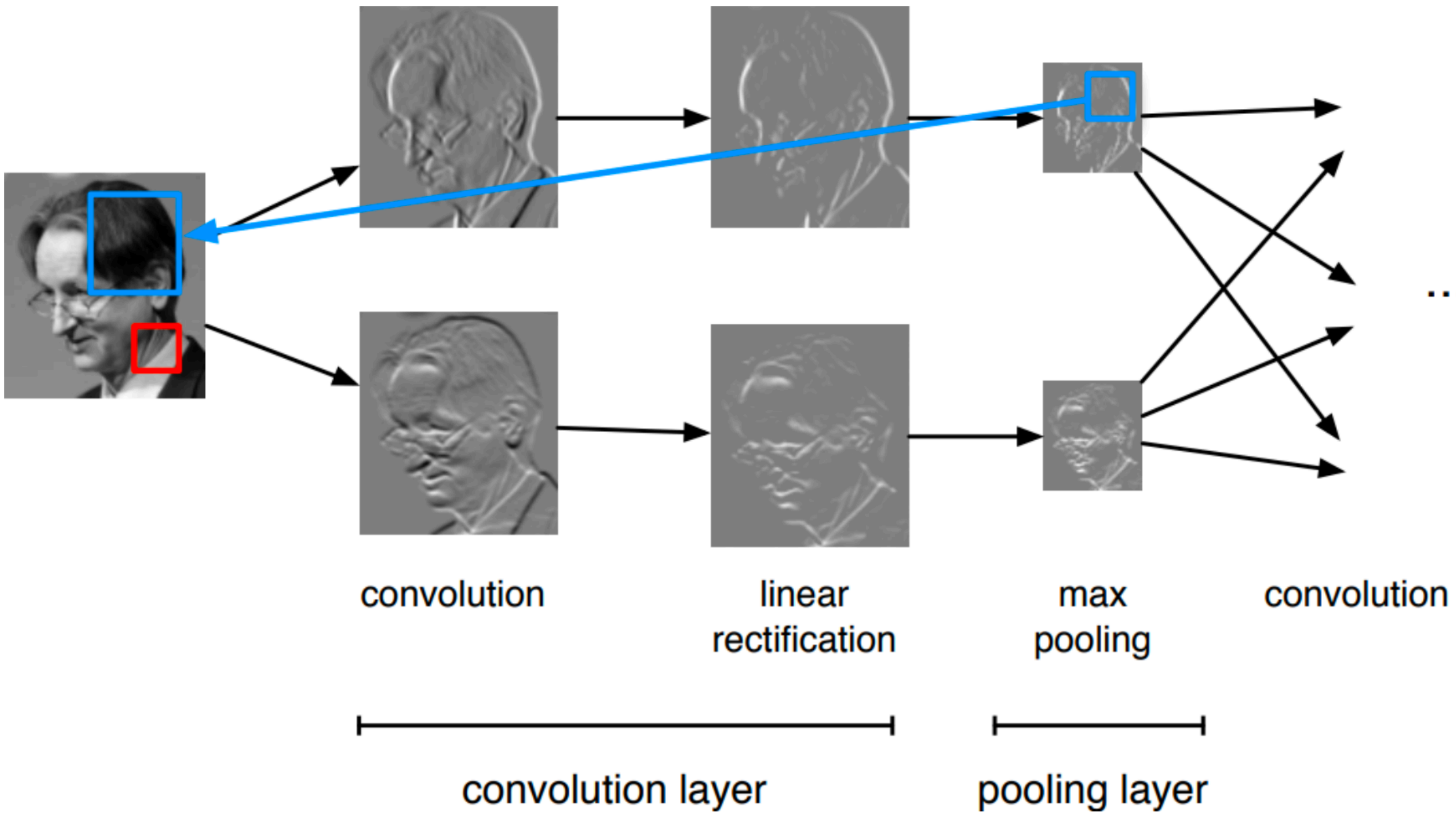
Pooling

- In each channel, find *max* or *average* value of pixels in a pooling area of size $h \times w$
- Invariance to permutation within pooling area
- Invariance to local perturbations

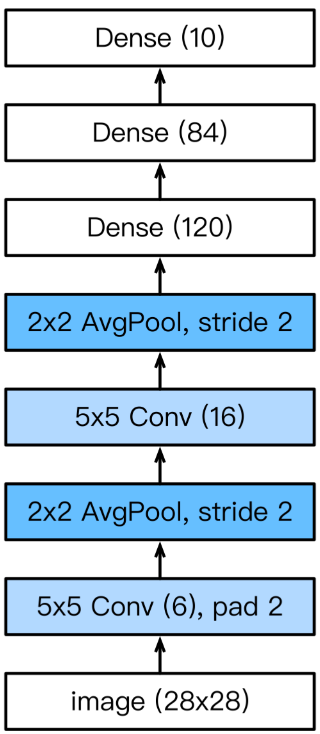


Convolutional Network

- A combination of convolution, pooling, ReLU, and fully connected layers

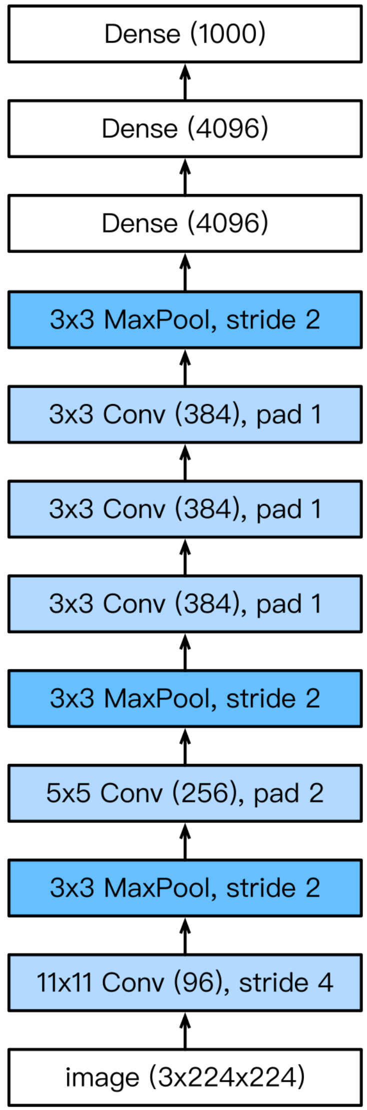


Convolutional Networks



LeNet

(LeCun et al, 1998)

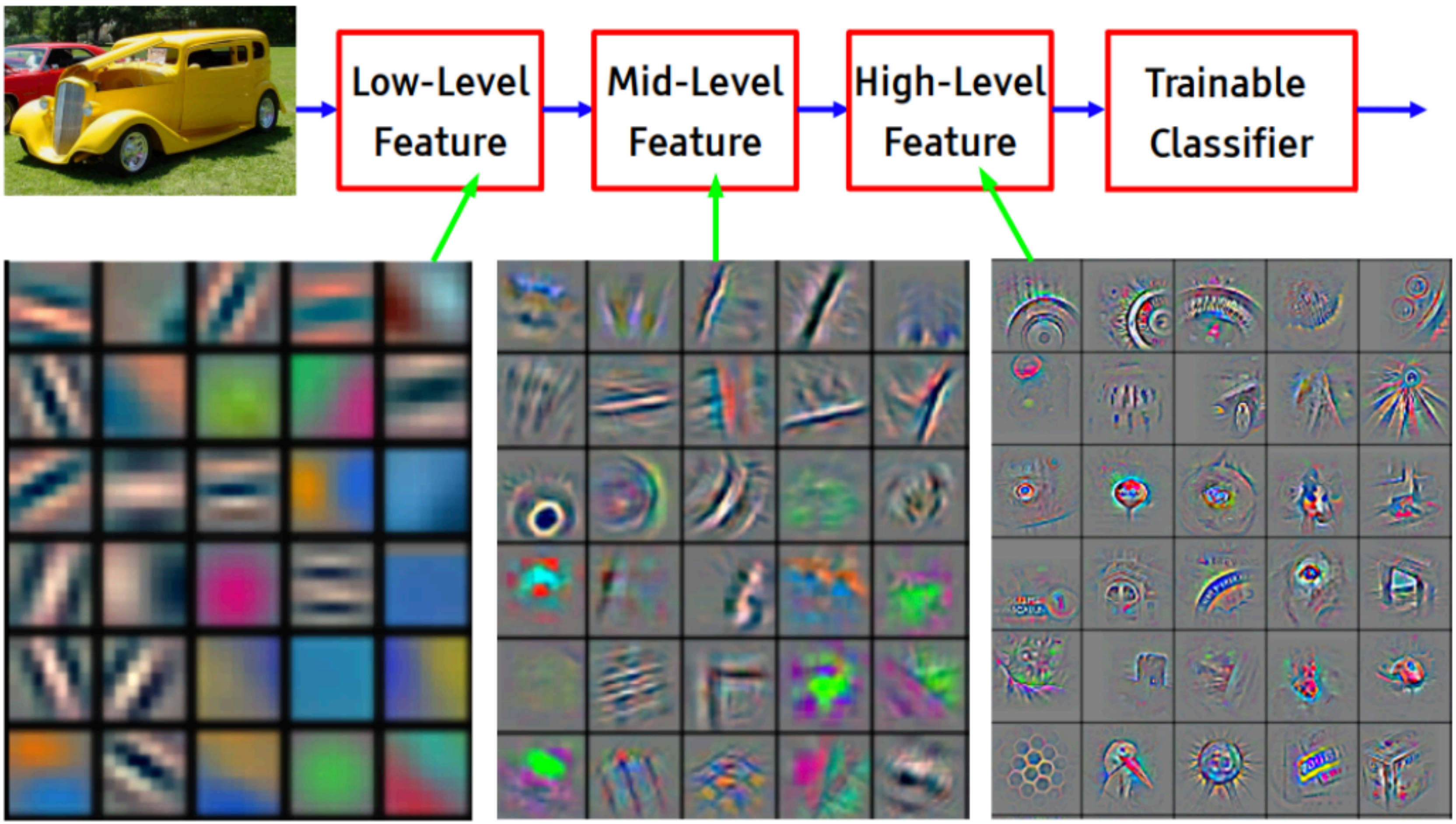


AlexNet

(Krizhevsky et al, 2012)

ImageNet Classification

Hierarchical Composition of Features

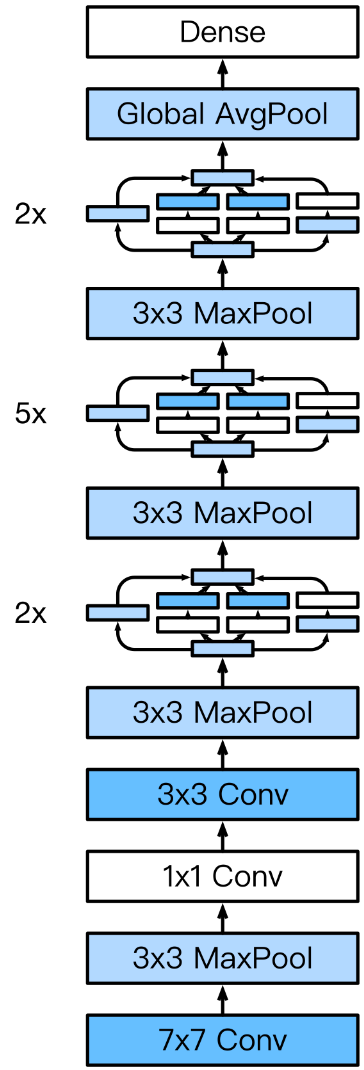


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

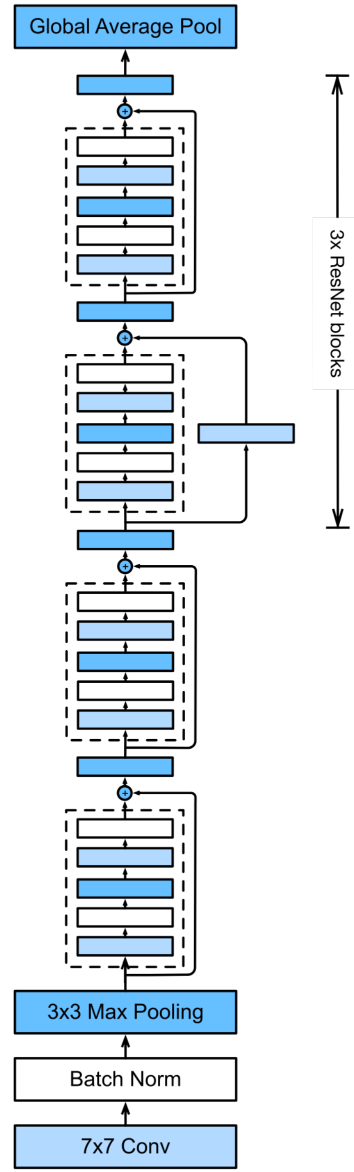
Very Deep CNNs

- To go deeper, architectures become much more complex
  - Multiple convolutions in parallel and recombined
  - Skip connections

- Recent ResNet-152 has 152 layers!



GoogLeNet
(Szegedy et al, 2014)

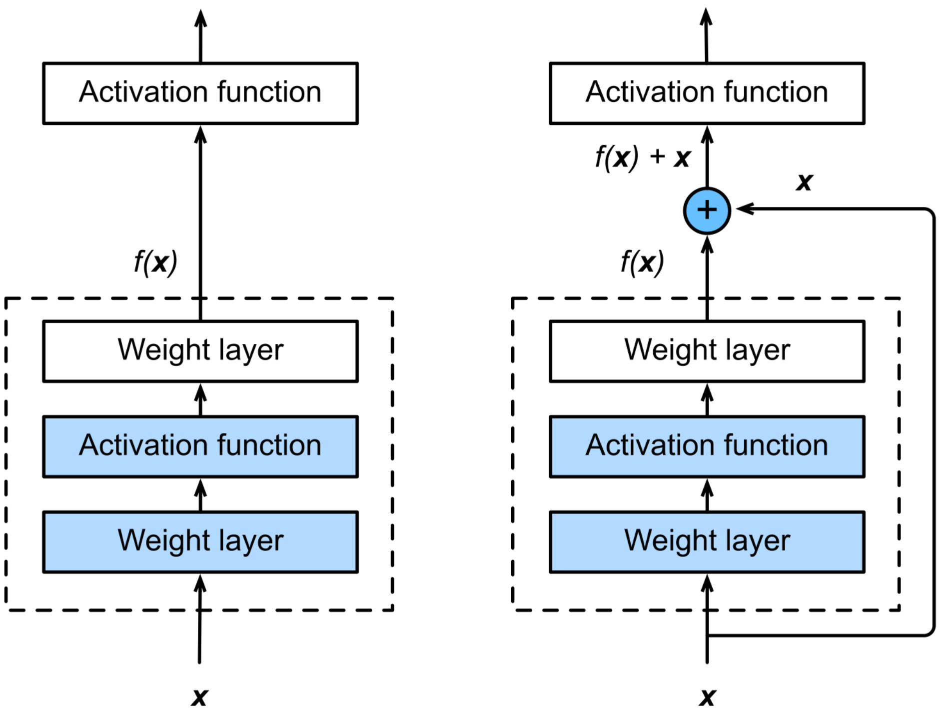


ResNet
(He et al, 2015)

Credits: [Deep Dive in Deep Learning]

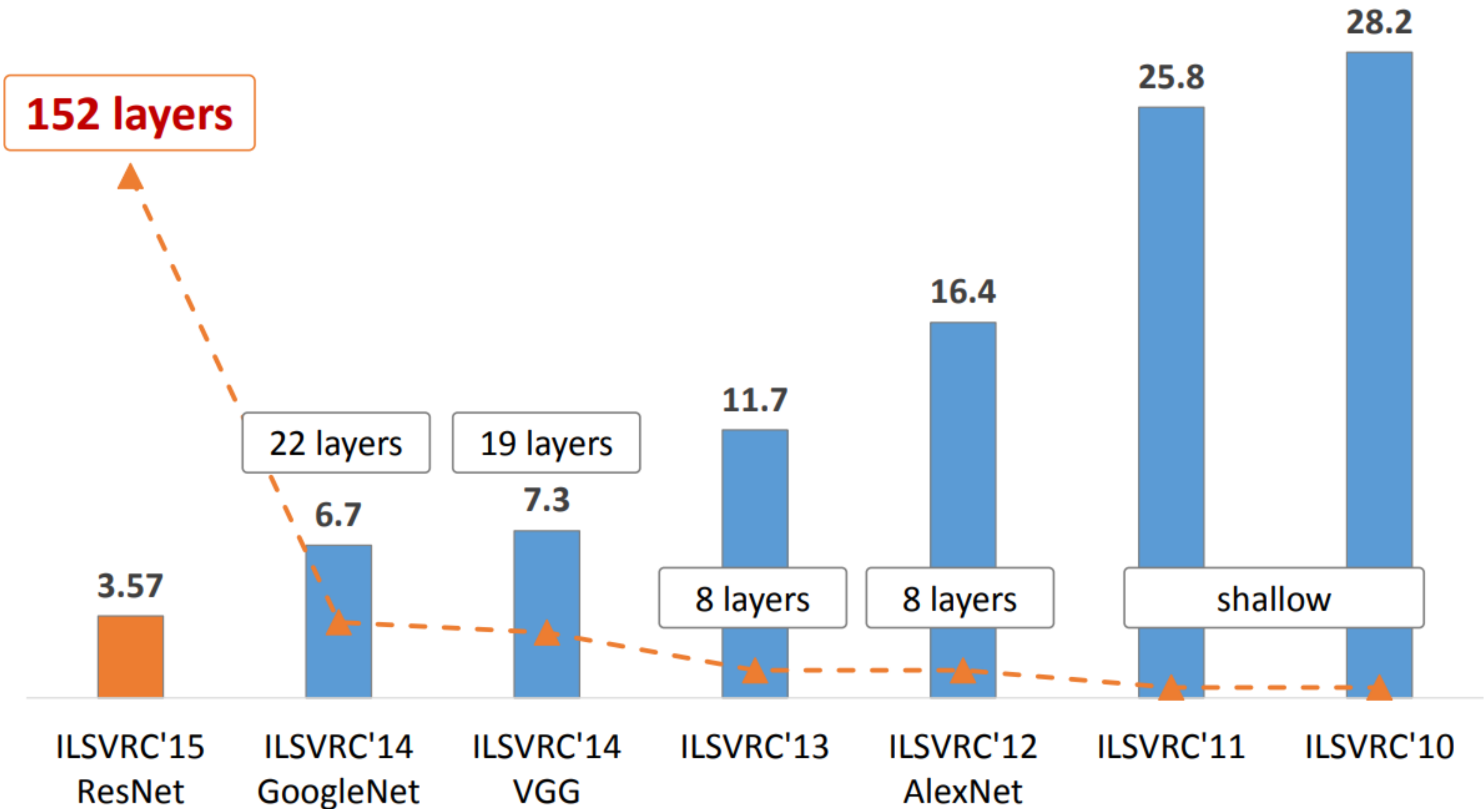
Residual Connections

- Training very deep networks is made possible because of the **skip connections** in the residual blocks. Gradients can shortcut the layers and pass through without vanishing.



Credits: Deep Dive in Deep Learning , and G. Louppe

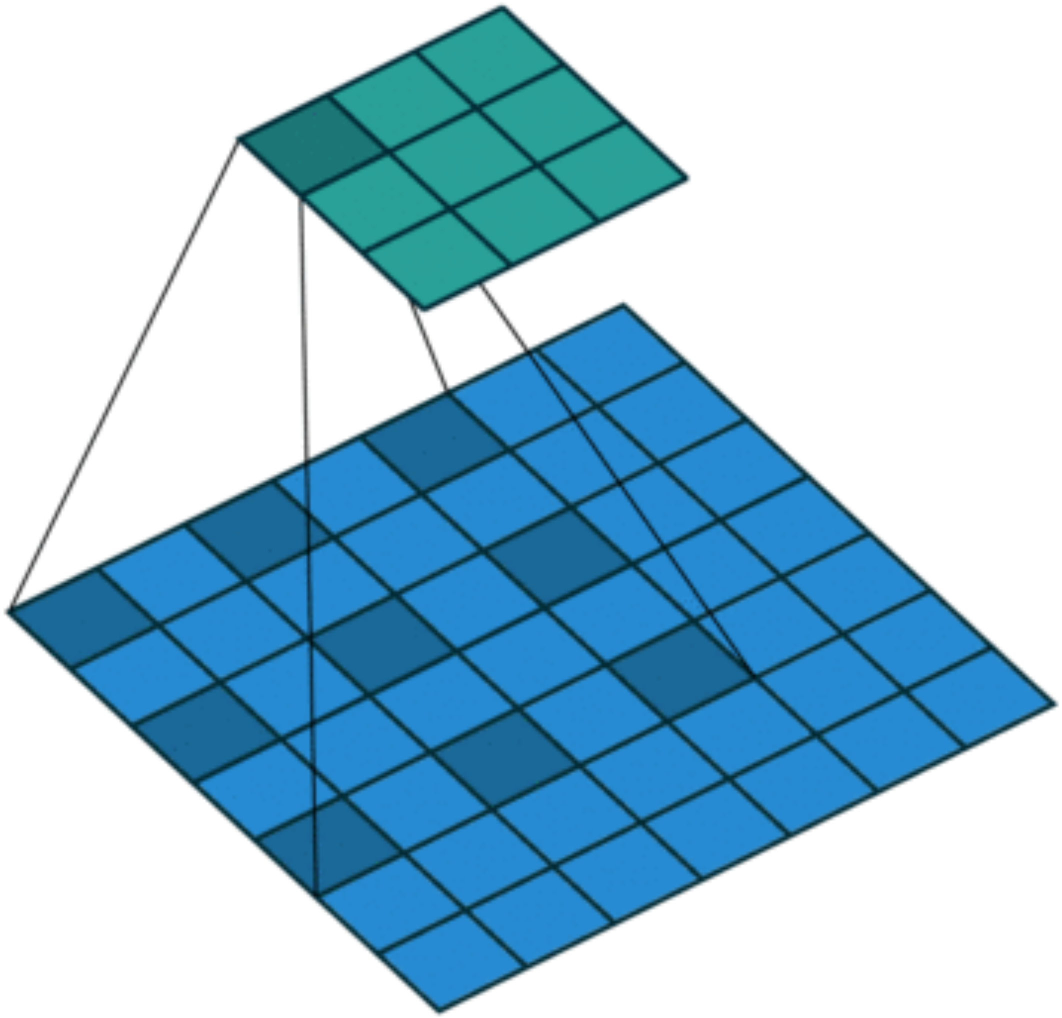
Benefits of Depth



End of Lecture I

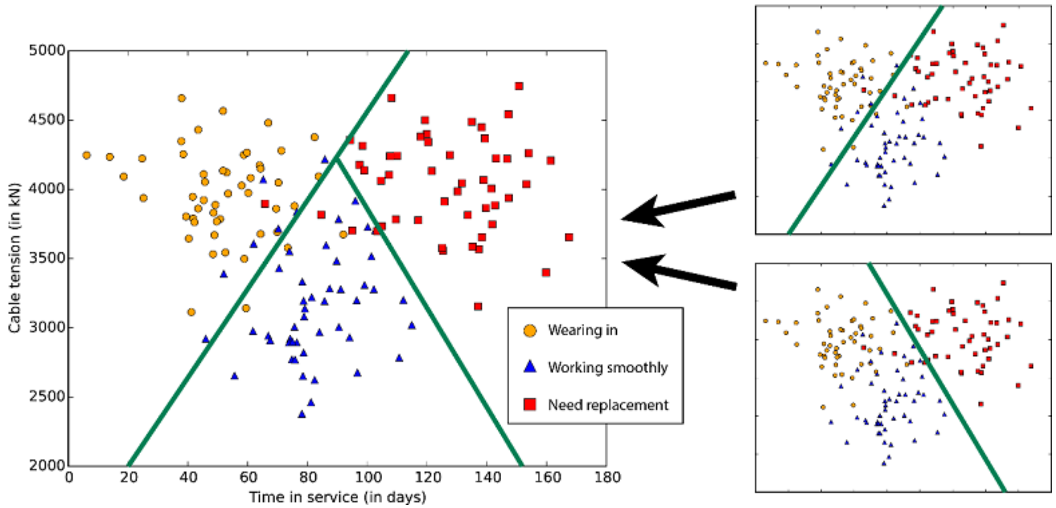
Backup

Dilation



Multiclass Classification?

- What if there is more than two classes?



- Softmax → multi-class generalization of logistic loss
  - Have N classes $\{c_1, \ldots, c_N\}$
  - Model target $\mathbf{y}_k = (0, \ldots, 1, \ldots 0)$

k^{th} element in vector

$$p(c_k|x) = \frac{\exp(\mathbf{w}_k x)}{\sum_j \exp(\mathbf{w}_j x)}$$

  - Gradient descent for each of the weights $\mathbf{w}_k$