

# Foundations of Deep Learning

## Backpropagation and Modularity



Stergios Christodoulidis  
MICS Laboratory  
CentraleSupélec  
Université Paris-Saclay  
<https://stergioc.github.io/>



# Last Lecture

# Motivation

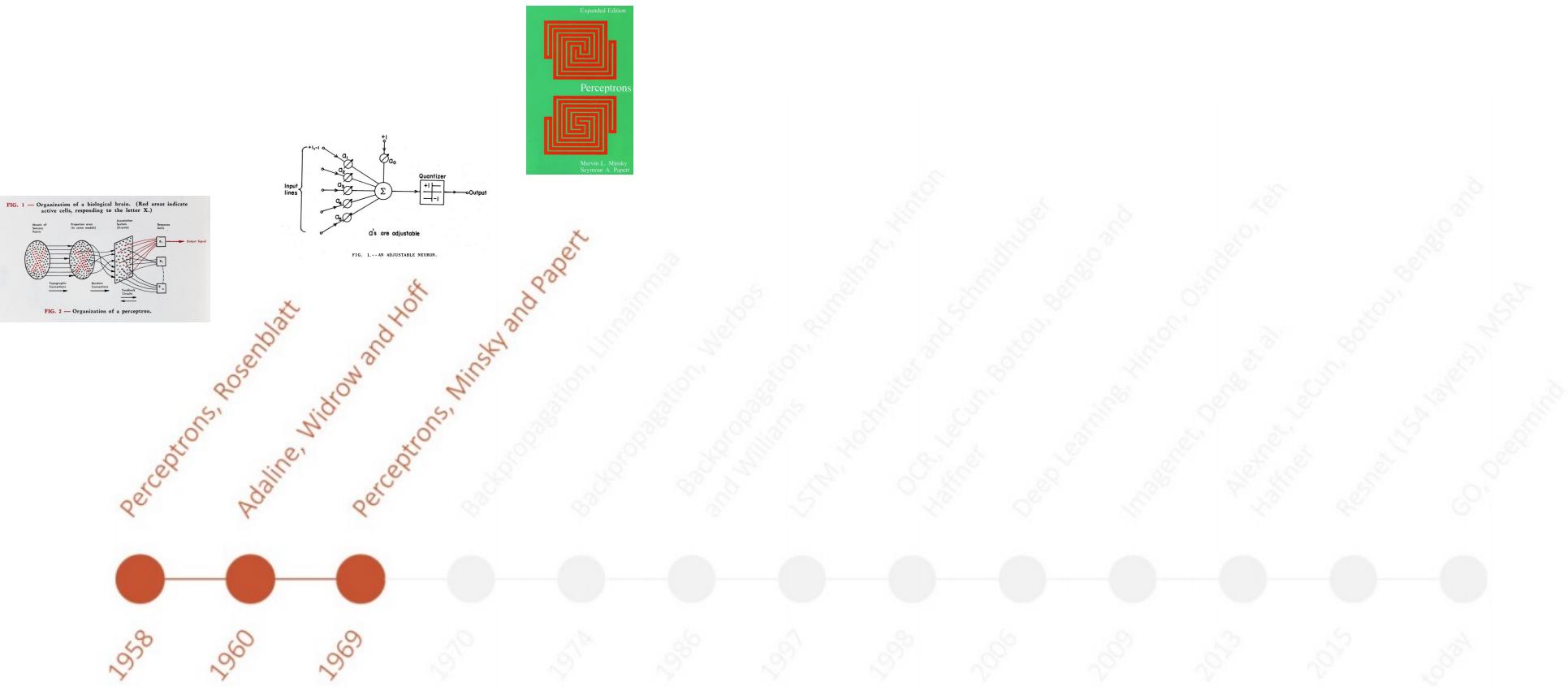


A person riding a motorcycle on a dirt road.

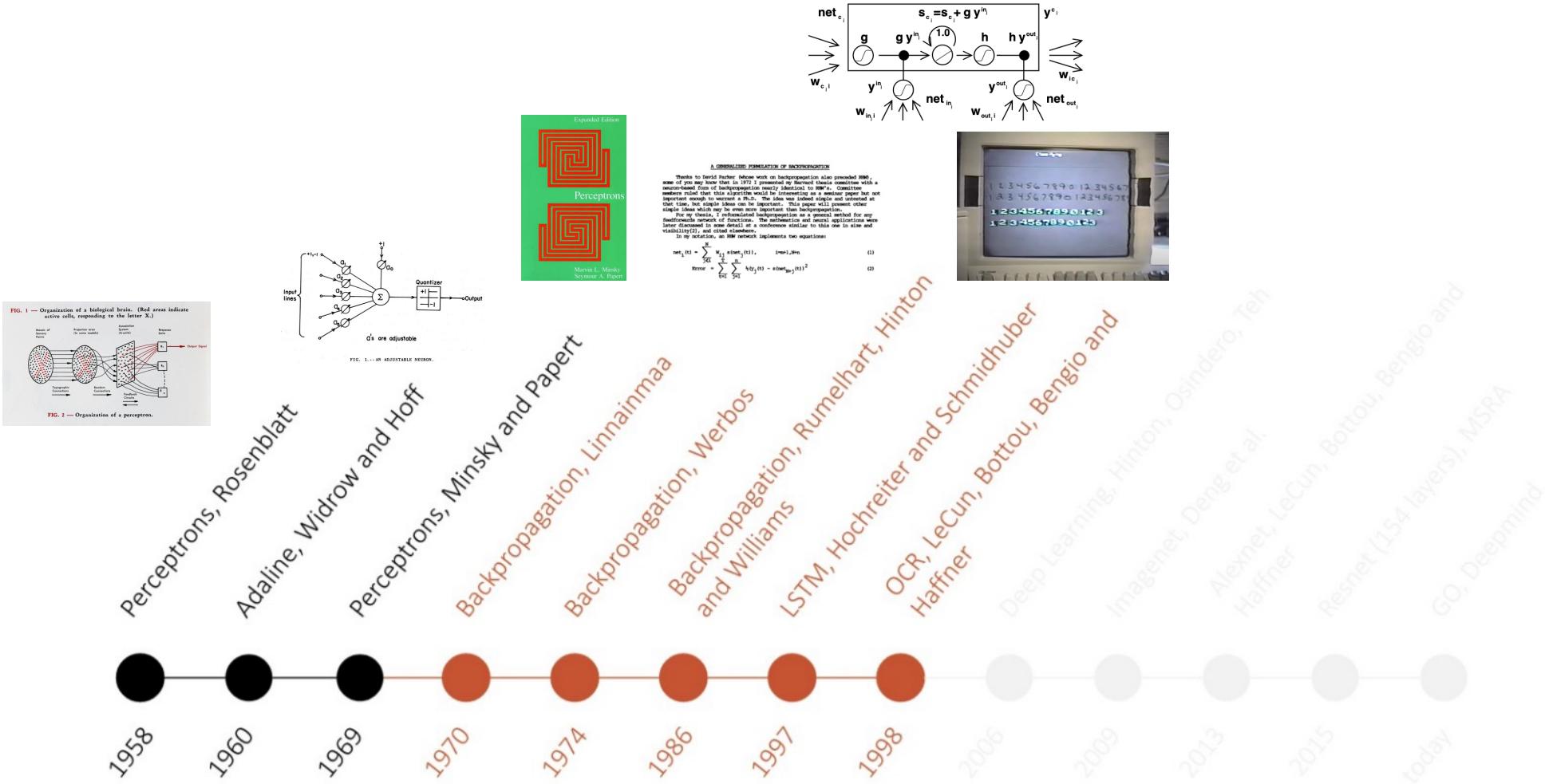
Two dogs play in the grass.

A skateboarder does a trick on a ramp.

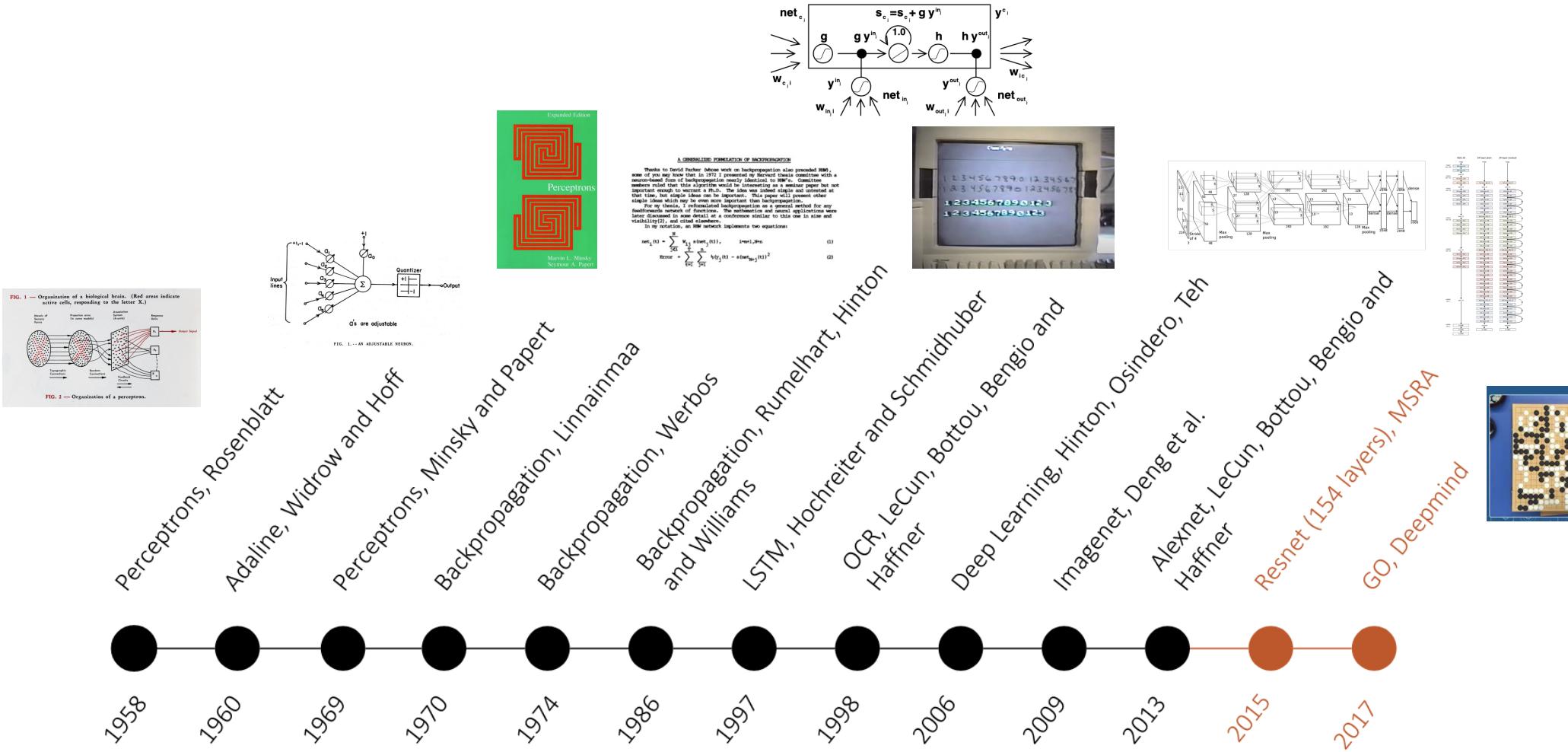
# Machine Learning Early Years



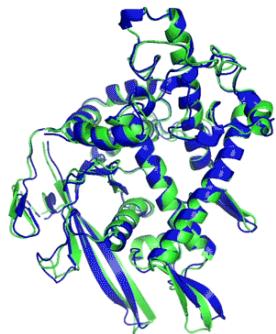
# The AI winter



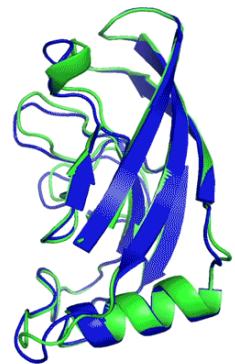
# Deep Learning Golden Era



# Latest Advances



T1037 / 6vr4  
90.7 GDT  
(RNA polymerase domain)



T1049 / 6y4f  
93.3 GDT  
(adhesin tip)

- Experimental result
- Computational prediction

(AlphaFold, DeepMind, 2020)



(DALL-E, OpenAI, 2021)

# Latest Advances



A bucket bag made of blue suede. The bag is decorated with intricate golden paisley patterns. The handle of the bag

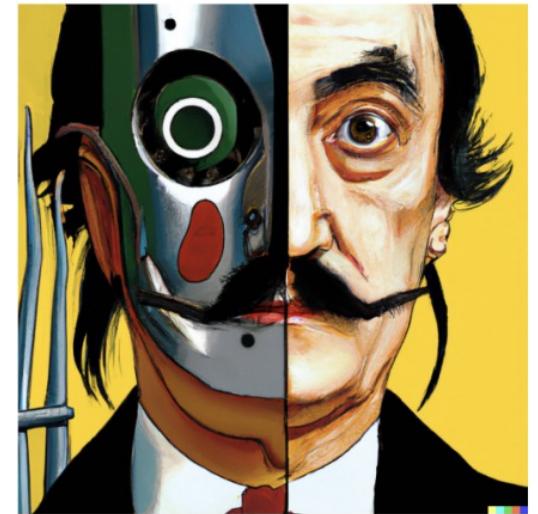


An alien octopus floats through a portal reading a newspaper.



A Pomeranian is sitting on the Kings throne wearing a crown. Two tiger soldiers are standing next to the throne.

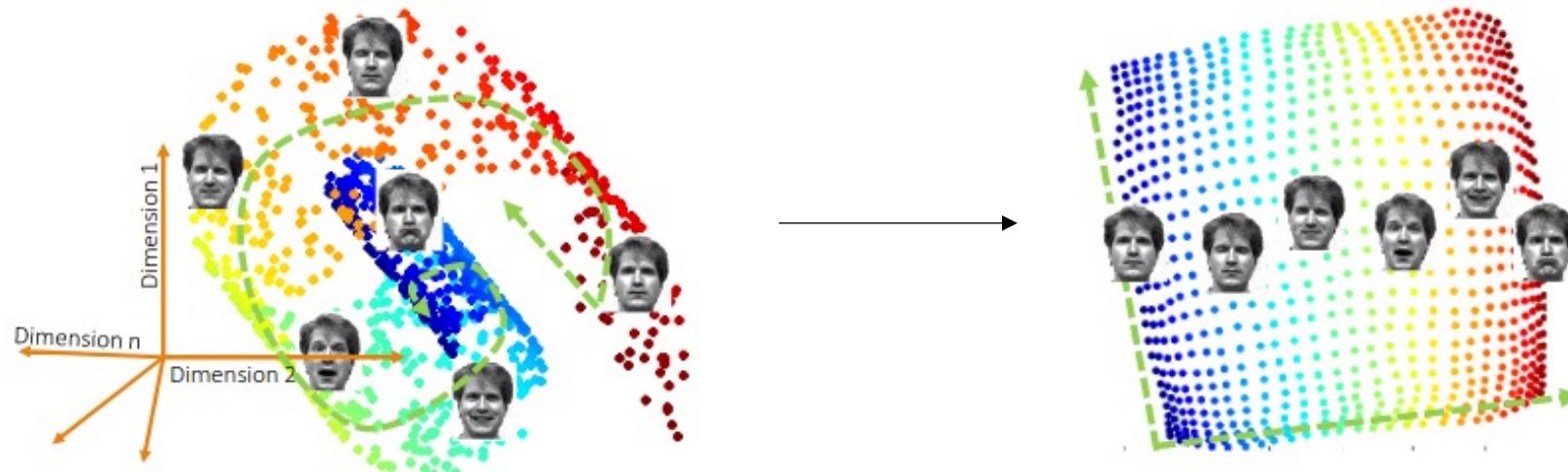
Imagen, Google, 2022



Dall-E 2, OpenAI 2022

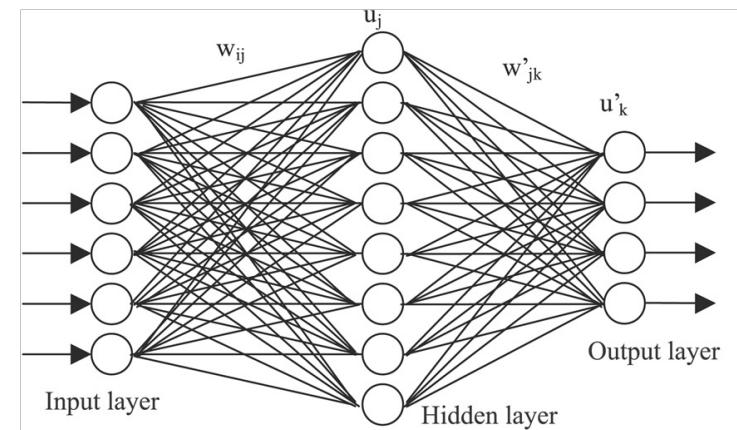
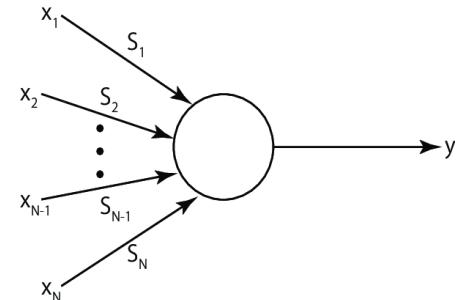
# Manifolds

- Raw data live in huge dimensionalities
- But effectively lie in lower dimensional manifolds → Euclidean properties in local regions
- Can we discover this manifold to embed our data on?



# Perceptrons

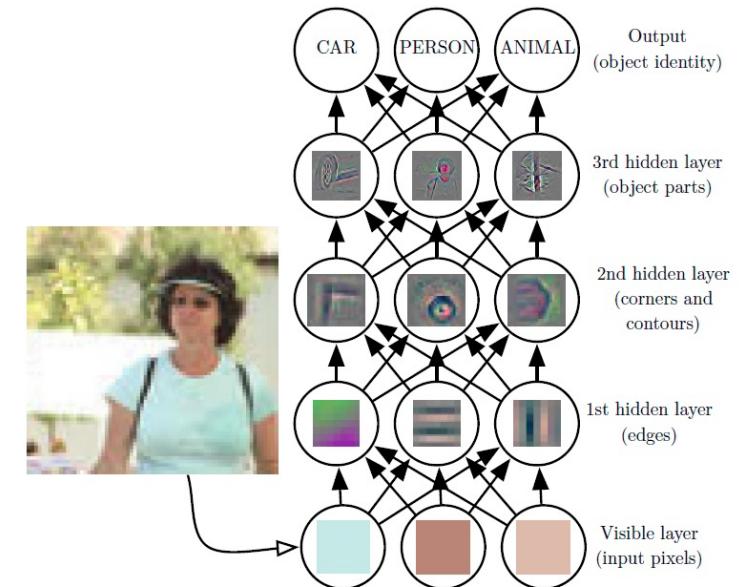
- 1 perceptron = 1 decision (Neuron)
- What about multiple decisions?
  - E.g. digit classification
- Stacks as many outputs as the possible outcomes into a layer
  - Neural Networks
- Use one layer as input to the next layer
  - Add nonlinearities between layers
  - Multi-layer perceptron (MLP)



# Deep Learning

- A set of parametric, nonlinear and hierarchical representation learning models, which are optimized with stochastic gradient descent to encode domain knowledge, i.e., domain invariances, stationarity.
- Given a training corpus the task is to find the optimal parameters for the model that fit to the data.
- The model is a set of nested parametric linear/non-linear functions:

$$f(X; \theta_{1,\dots,n}) = h_n(h_{n-1}(\dots h_1(X; \theta_1); \theta_{n-1}); \theta_n)$$



(Goodfellow et al., 2016)

# Today's Lecture

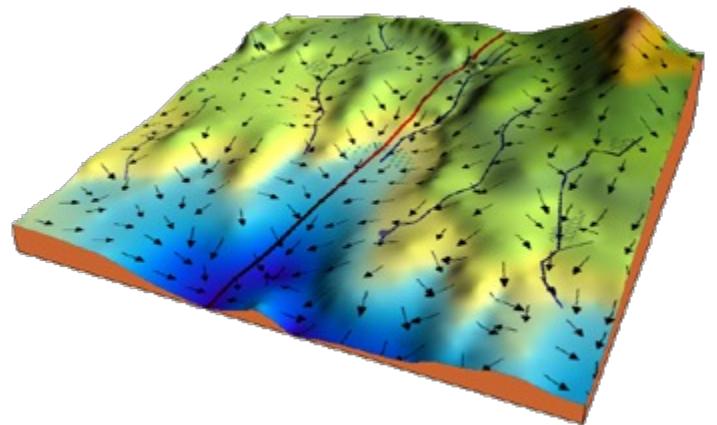
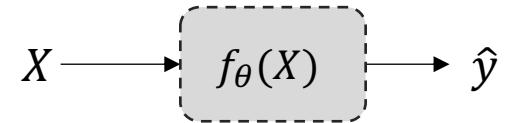
# Today's Lecture

- Gradient Descent
  - E.g., Simple Linear regression
  - E.g., Linear Neuron model
  - Chain Rule and Backpropagation
  - E.g., Multi-layer Perceptron
- Modularity in Deep Learning
  - Combining modules
  - Popular Deep Learning modules
- Training Deep Neural Networks in Practice
  - Learning rate
  - Schedules

# Gradient Descent

# Gradient Descent

- Optimization algorithm
  - Minimize a function by tuning the model parameters
  - Moving in the direction of the steepest negative gradient ( $\frac{d\mathcal{L}}{d\theta}$ )
- Loss/Cost Function ( $\mathcal{L}$ )
  - The function that is minimized in the context of ML
- Parameter optimization ( $\theta$ )
  - We start at the top of a hill
  - We iteratively take steps towards the negative gradient
  - We stop when we can no longer move downhill
- Learning Rate ( $\lambda$ )
  - How large will our steps be



$$\theta_{n+1} = \theta_n - \lambda \cdot \frac{d\mathcal{L}}{d\theta}$$

# Simple Linear Regression Model

- Model Function

$$f(x) = mx + b$$

- Data

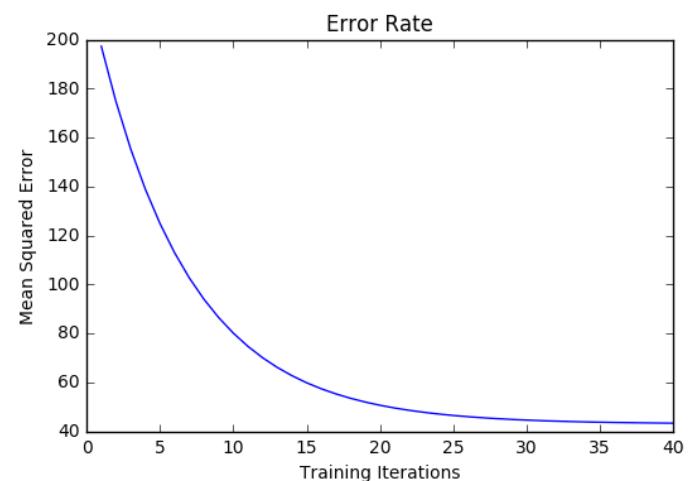
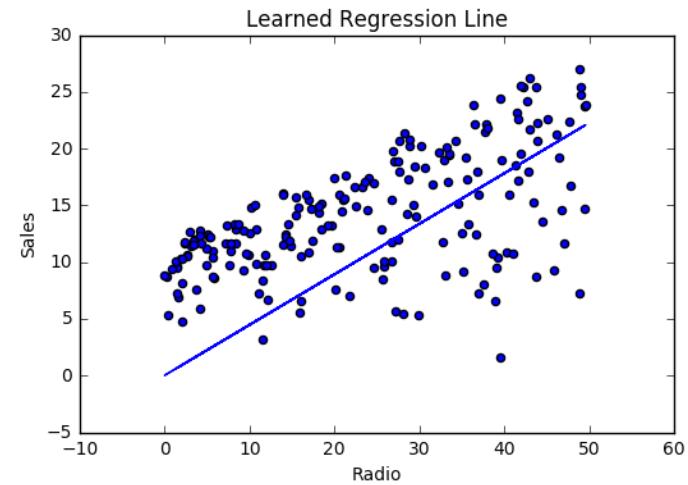
$$\{x^{(i)}, y^{(i)}\}_{i=1}^n: x \in \mathcal{R}, y \in \mathcal{R}$$

- Loss function (MSE)

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n (f(x^{(i)}) - y^{(i)})^2$$

- Gradient Descent

$$\theta_{n+1} = \theta_n - \lambda \frac{d\mathcal{L}}{d\theta}, \quad \theta = \{m, b\}$$



# Linear Neuron Model – Perceptron

- Model Function

$$f(x) = R(\mathbf{X}, \mathbf{S}) = x_1S_1 + x_2S_2 + \dots + x_NS_N = \mathbf{x}\mathbf{S}^T$$

- Data

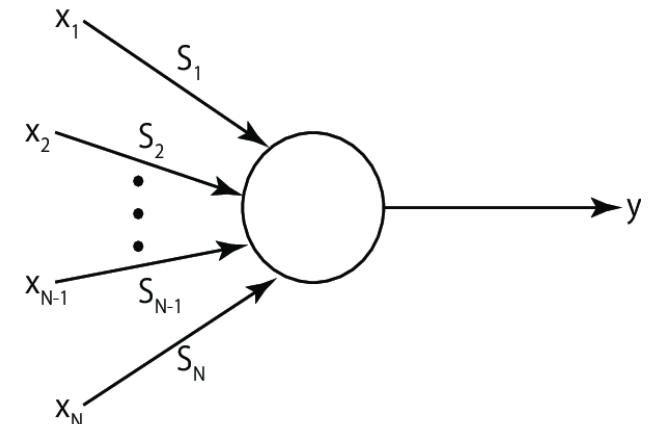
$$\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^N: x \in \mathcal{R}^d, y \in \mathcal{R}$$

- Loss function (MSE)

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N (f(\mathbf{x}^{(i)}) - y^{(i)})^2$$

- Gradient Descent

$$\theta_{n+1} = \theta_n - \lambda \frac{d\mathcal{L}}{d\theta}, \quad \theta = \{S_1, S_2, \dots, S_n\}$$



# Multi-layer Perceptron Model

- Model Function

$$g_{W_h}(x) = h = \mathbf{x} W_h \text{ (linear activation function)}$$

$$f_{W_o}(h) = \hat{y} = h W_o \text{ (linear activation function)}$$

- Data

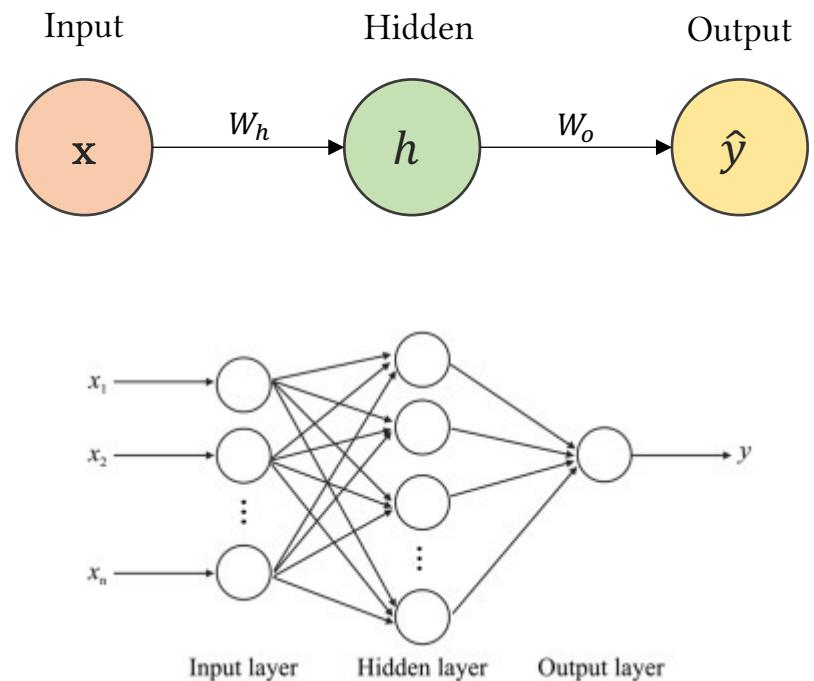
$$\mathbf{x} \in \mathcal{R}^d, y \in \mathcal{R}$$

- Loss function

$$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$$

- Gradient Descent

$$\theta_{n+1} = \theta_n - \lambda \frac{d\mathcal{L}}{d\theta}, \quad \theta = \{W_h, W_o\}$$



# Chain Rule Refresher

- Differentiation in the case of nested functions

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

- We can use the chain rule to calculate the gradient with respect to a variable

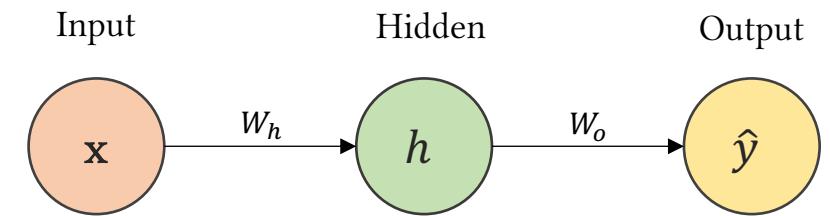
$$F'(x) = f'(g(x))g'(x) \text{ (Lagrange's notation)}$$

$$\frac{dF}{dx} = \frac{df}{dg} \frac{dg}{dx} \text{ (Leibniz's notation)}$$

# Applying the Chain Rule to Multi-layer Perceptron

- Multiple Layers of Computation → Nested functions

$$\min \mathcal{L}\left(f_{W_o}\left(g_{W_h}(x)\right)\right)$$



- We can use the chain rule to calculate the gradient:

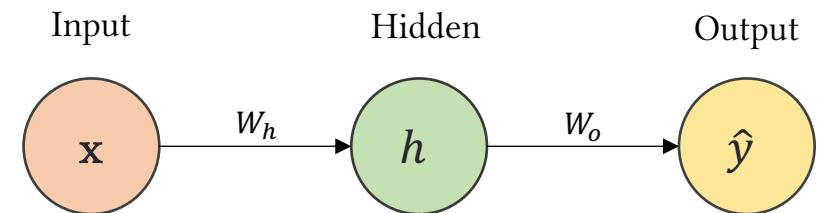
$$\frac{d\mathcal{L}}{dW_o} = \frac{d\mathcal{L}}{df} \cdot \frac{df}{dW_o} = \frac{d\mathcal{L}}{d\hat{y}} \cdot \frac{d\hat{y}}{dW_o}$$

$$\frac{d\mathcal{L}}{dW_h} = \frac{d\mathcal{L}}{df} \cdot \frac{df}{dg} \cdot \frac{dg}{dW_h} = \frac{d\mathcal{L}}{d\hat{y}} \cdot \frac{d\hat{y}}{dh} \cdot \frac{dh}{dW_h}$$

# Applying the Chain Rule to Multilayer Perceptron

- Let's calculate the gradients:

Function	Gradient
$\mathcal{L} = \frac{1}{2}[\hat{y} - y]^2$	$\frac{d\mathcal{L}}{d\hat{y}} = (\hat{y} - y)$
$\hat{y} = \mathbf{h}\mathbf{W}_o$	$\frac{d\hat{y}}{d\mathbf{W}_o} = \mathbf{h}, \quad \frac{d\hat{y}}{d\mathbf{h}} = \mathbf{W}_o$
$\mathbf{h} = \mathbf{X}\mathbf{W}_h$	$\frac{d\mathbf{h}}{d\mathbf{W}_h} = \mathbf{X}$

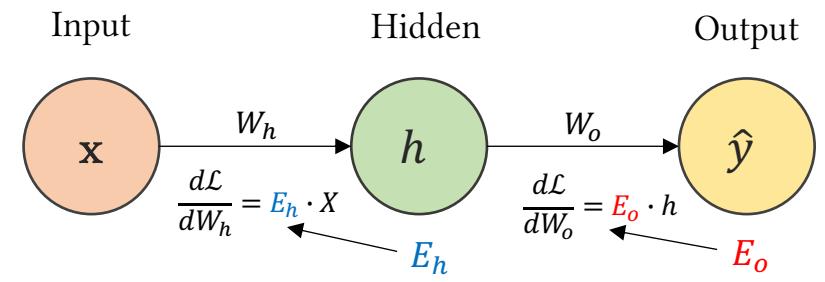


# Applying the Chain Rule to Multilayer Perceptron

- Let's calculate the gradients:

$$\frac{d\mathcal{L}}{dW_o} = \frac{d\mathcal{L}}{d\hat{y}} \cdot \frac{d\hat{y}}{dW_o} = (\hat{y} - y) \mathbf{h} = \mathbf{E}_0 \mathbf{h}$$

$$\frac{d\mathcal{L}}{dW_h} = \frac{d\mathcal{L}}{d\hat{y}} \cdot \frac{d\hat{y}}{dh} \cdot \frac{dh}{dW_h} = (\hat{y} - y) \mathbf{W}_o \mathbf{X} = \mathbf{E}_h \mathbf{X}$$



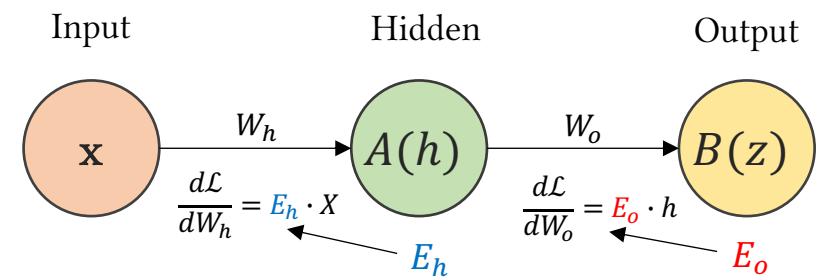
- Gradient Updates:

$$W_o^{(i+1)} = W_o^{(i)} - \lambda(\hat{y} - y) \mathbf{h}$$

$$W_h^{(i+1)} = W_h^{(i)} - \lambda(\hat{y} - y) \mathbf{W}_o \mathbf{X}$$

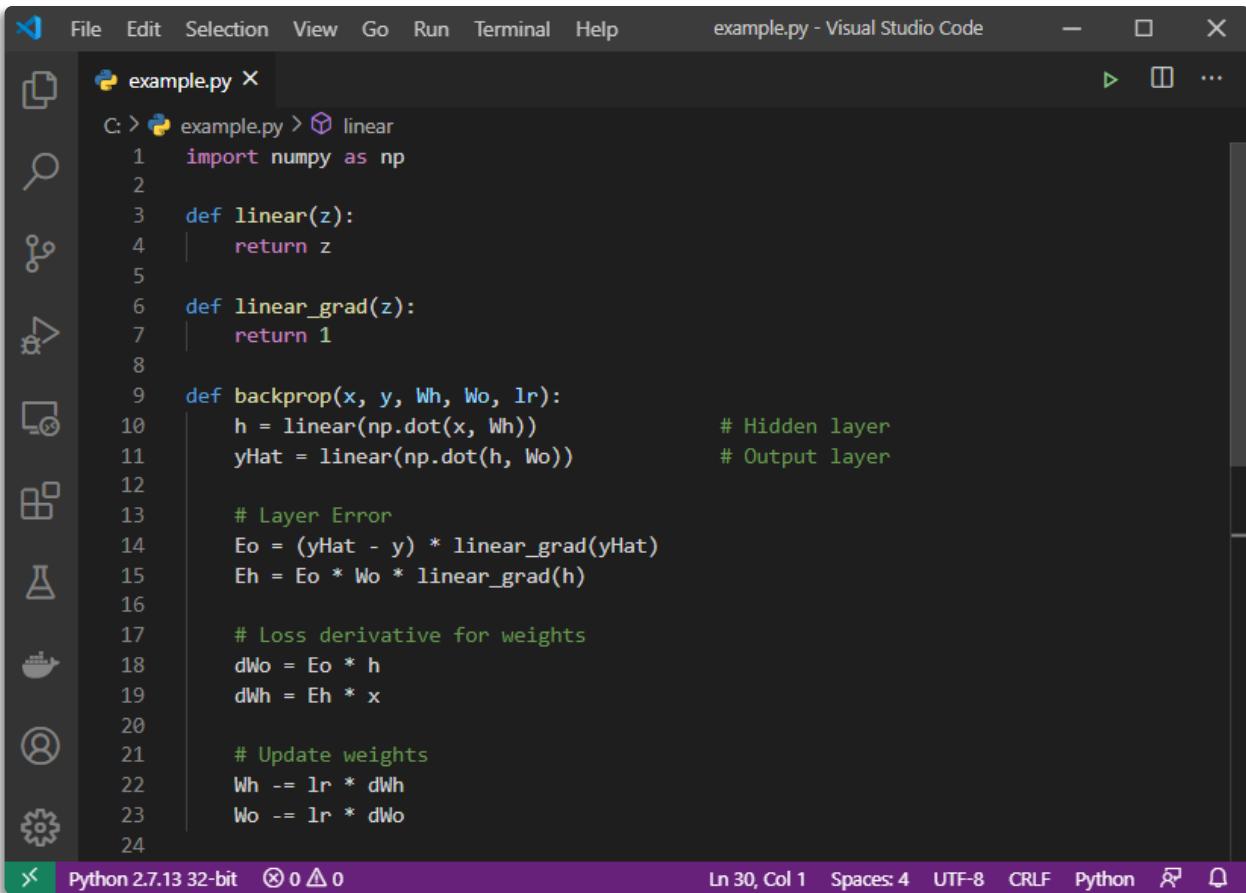
# General Case

- Different activation functions are applied in each output
- Optimization of parameters is performed again using the chain rule
- The gradient of each layer is the backpropagated error times it's input.



Function	Gradient
Output Layer Error	$E_o = \frac{d\mathcal{L}}{dB} \frac{dB}{dz} = (B(z) - y) \frac{dB}{dz}$
Hidden Layer Error	$E_h = \frac{d\mathcal{L}}{dB} \frac{dB}{dz} \frac{dz}{dA} \frac{dA}{dh} = E_o \cdot W_o \cdot \frac{dA}{dh}$
Gradient	<i>Backpropagated Error * Layer Input</i>

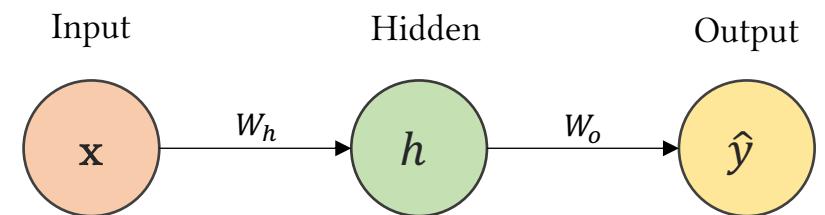
# Python Code Example



The screenshot shows a Visual Studio Code window with the file "example.py" open. The code implements a simple neural network layer. It defines three functions: "linear", "linear\_grad", and "backprop". The "linear" function returns its input. The "linear\_grad" function returns 1. The "backprop" function performs backpropagation through two linear layers. It calculates hidden layer output  $h$  and output layer prediction  $\hat{y}$ . It then computes error derivatives  $E_o$  and  $E_h$ , and loss derivatives  $dW_o$  and  $dW_h$ . Finally, it updates weights  $W_h$  and  $W_o$  using a learning rate  $lr$ .

```
C:\> example.py > linear
1 import numpy as np
2
3 def linear(z):
4     return z
5
6 def linear_grad(z):
7     return 1
8
9 def backprop(x, y, Wh, Wo, lr):
10    h = linear(np.dot(x, Wh))           # Hidden layer
11    yHat = linear(np.dot(h, Wo))         # Output layer
12
13    # Layer Error
14    Eo = (yHat - y) * linear_grad(yHat)
15    Eh = Eo * Wo * linear_grad(h)
16
17    # Loss derivative for weights
18    dWo = Eo * h
19    dWh = Eh * x
20
21    # Update weights
22    Wh -= lr * dWh
23    Wo -= lr * dWo
24
```

Python 2.7.13 32-bit ⊗ 0 Δ 0 Ln 30, Col 1 Spaces: 4 UTF-8 CRLF Python ⚙️ 🔍

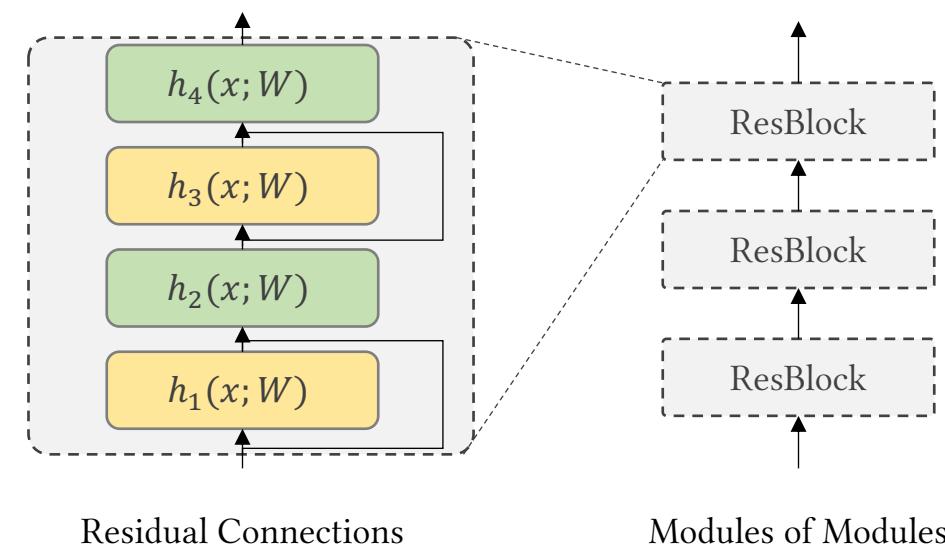
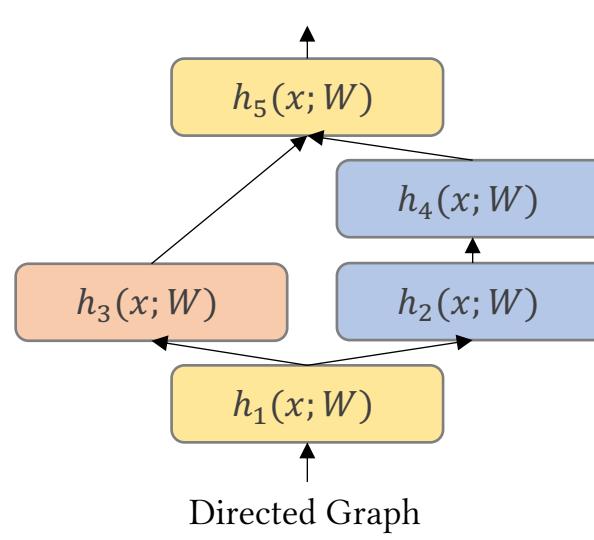
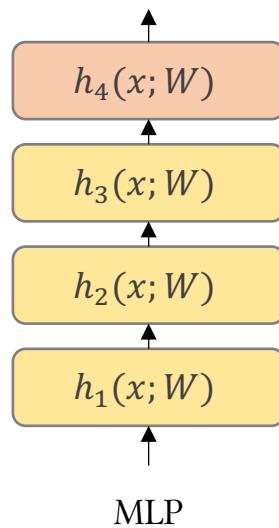


(<https://youtu.be/VMj-3S1tku0>)

# Modularity in Deep Learning

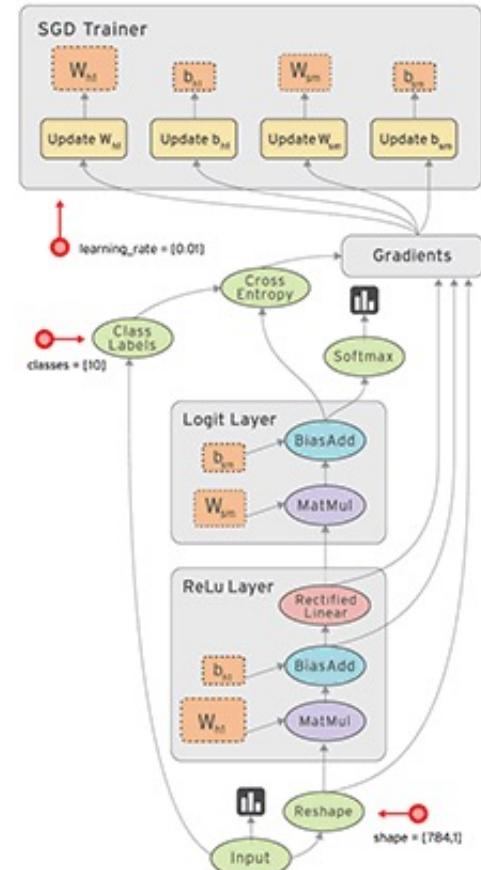
# Modularity

- Deep neural networks can be defined as a series of hierarchically connected functions.
- In order to be optimized with gradient descent each of these functions should be differentiable.
- These hierarchies can get very complex!



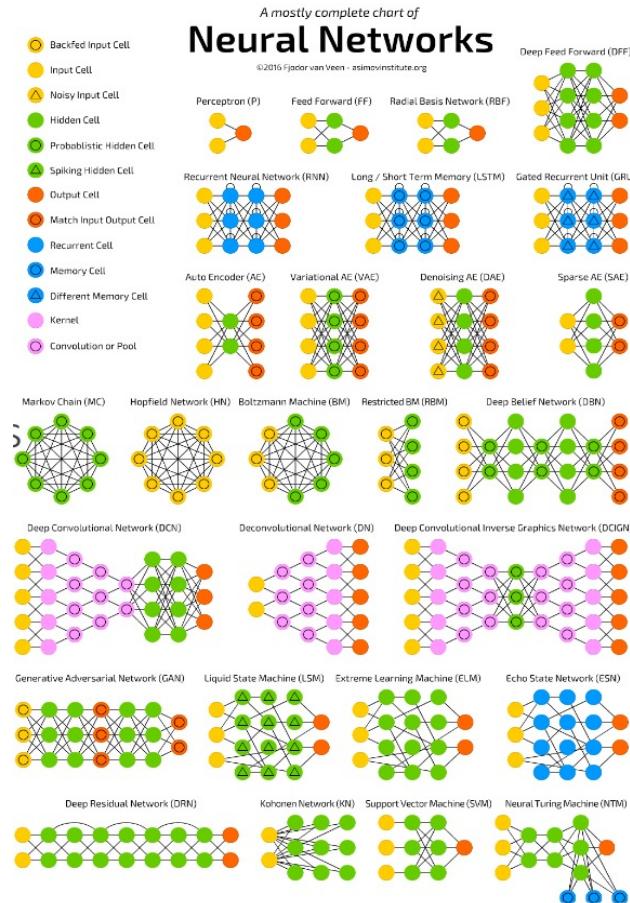
# Modules

- A module is a building block for our network
- Each module is a function  $a = h_w(x)$  that
  - Contains trainable parameters w
  - Receives as an argument an input x
  - Returns an output a based on the activation function  $h(\dots)$
- The activation function should be (at least) first order differentiable (almost) everywhere
- For easier/more efficient backpropagation → store module input
  - Easy to get module output fast
  - Easy to compute derivatives



# List of Different Modules/Functions

- Activation Functions
- Loss Functions
- Normalization Layers
- Down-sampling/Up-sampling Layers
- Convolutional Layers
- Concatenation Layers
- Noise Layers
- (Many More)



# Activation Functions

# Linear Activation Function

- Activation:

$$R(z) = z$$

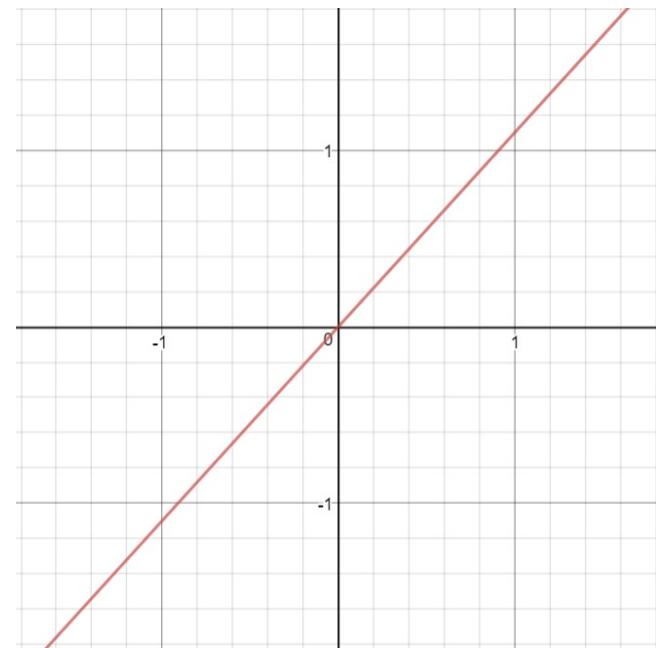
- Gradient:

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

- No activation saturation

- Hence, strong & stable gradients

- Reliable learning with linear modules



# Rectified Linear Unit (ReLU)

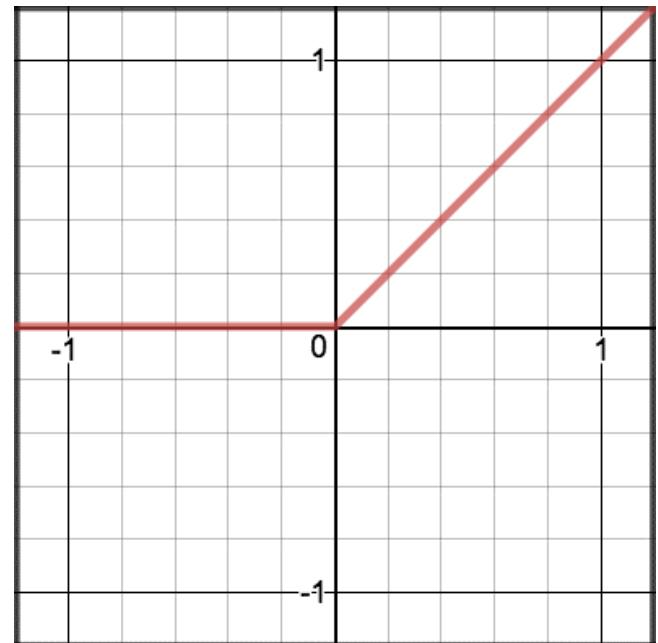
- Activation:

$$R(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

- Gradient:

$$\frac{dR}{dz} = \begin{cases} 1, & z > 0 \\ 0, & z < 0 \end{cases}$$

- Strong gradients: either 0 or 1
- Fast gradients: just a binary comparison
- It is not differentiable at 0, but not a big problem
  - An activation of precisely 0 rarely happens with non-zero weights, and if it happens, we choose a convention
- Nowadays ReLU is the default non-linearity



# Sigmoid Function

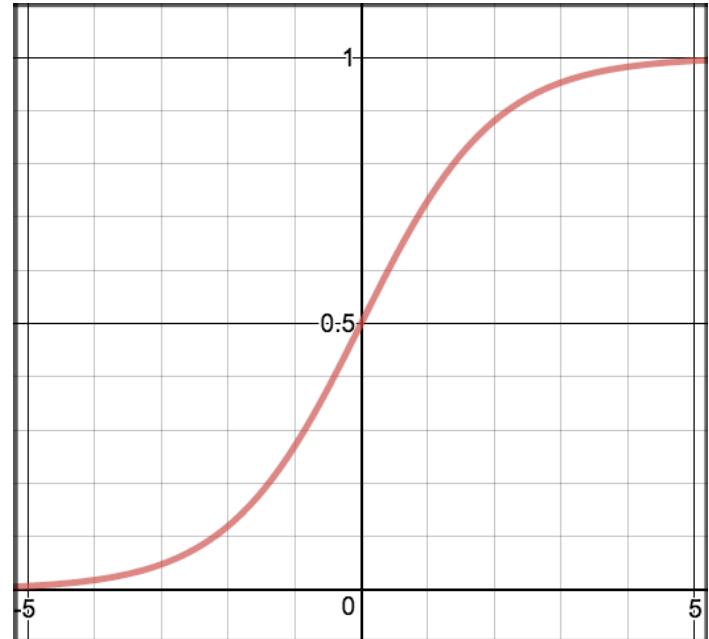
- Activation:

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

- Gradient:

$$\frac{dS}{dz} = S(z) \cdot (1 - S(z))$$

- Smooth gradients
- Outputs in the range [0,1]



# Tanh Function

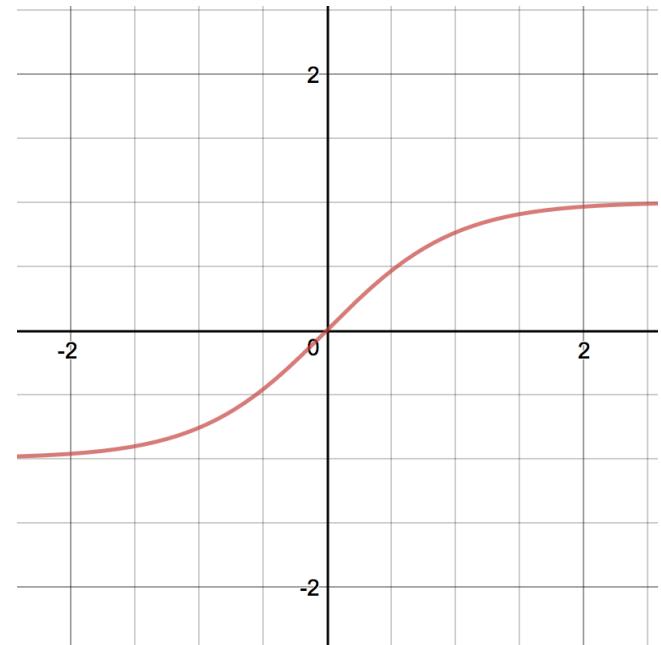
- Activation:

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

- Gradient:

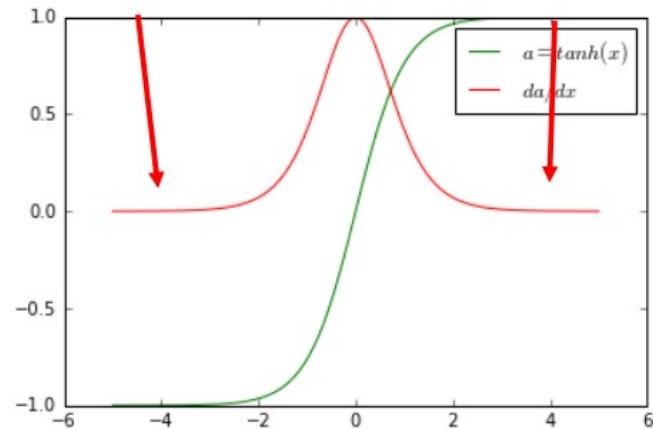
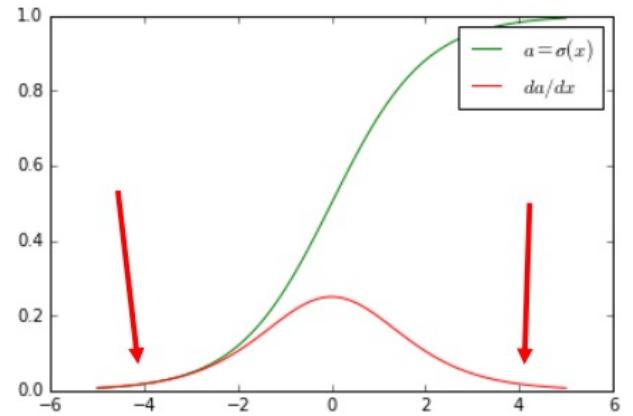
$$\frac{dT}{dz} = 1 - T(z)^2$$

- Smooth gradients
- output in the range [-1,1]



# Sigmoid vs Tanh

- Functional form is very similar
- $\tanh(x)$  due to the output range  $[-1, +1]$ :
  - Larger gradient ranges,  $[0, 1]$  instead of  $[0, 0.5]$
- Both saturate at the extreme values  $\rightarrow 0$  gradients
  - “Overconfident”, without necessarily being correct
  - Especially bad when in the middle layers: why should a neuron be overconfident, when it represents a latent variable
- The gradients for both are  $< 1$  so we can have small gradient values if propagating errors in multiple layers
- From the two,  $\tanh(x)$  enables better learning
  - But still, not a great choice



# Softmax

- Activation:

$$S(z_i) = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}}$$

- Outputs probability distribution for N classes

$$\sum_{j=1}^N S(z_j) = 1$$

- Avoid exponentiating too large/small numbers → better stability

# Loss Functions

# Loss functions for different Tasks

- Regression Tasks

- Mean Absolute Error (MAE, L1)

$$\mathcal{L}_{MAE} = \frac{1}{n} \sum_{i=1}^n \|f(x_i) - y_i\|$$

- Mean Squared Error (MSE, L2)

$$\mathcal{L}_{MSE} = \frac{1}{n} \sum_{i=1}^n (f(x_i) - y_i)^2$$

- Classification Tasks

- Categorical Cross Entropy Loss (CCE)

$$\mathcal{L}_{CCE} = - \sum_{j=1}^k y_j \cdot \log(\hat{y}_j)$$

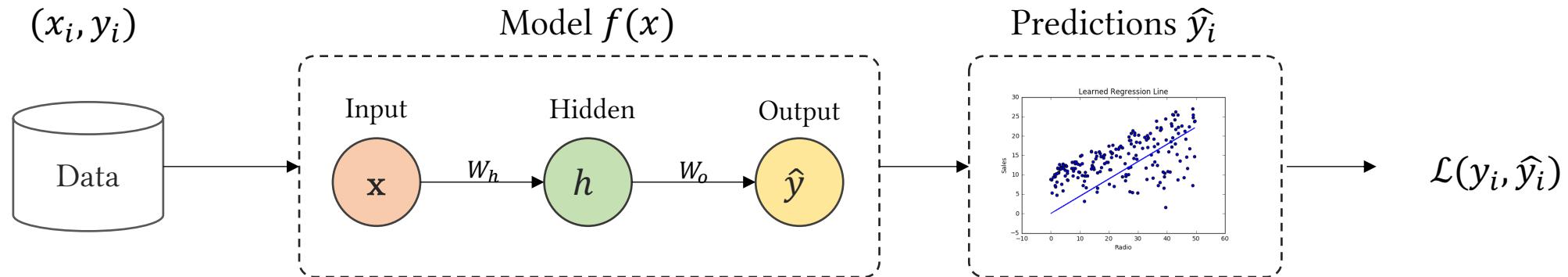
# Training Deep Neural Networks

# Task specific modules

- Regression:
  - Target output is a single or multiple continuous values
  - Activation functions for the output layer must be appropriate (e.g., Linear, tanh, sigmoid)
  - Take care of the target range e.g., sigmoid cannot be used for a target variable with range  $>1$  (!!!)
  - Typically used loss functions are MSE (L2), MAE (L1)
- Classification:
  - Target output a single or multiple categorical values.
  - Typically, the different target classes are encoded in one-hot vectors (e.g. [0 0 1 0] -> class3)
  - Activation functions should encode probabilities of the input to belong to a class (e.g. [0.01, 0.002, 0.9, 0.08])
  - Output should sum up to 1 (Softmax)
  - Typically used loss functions are Categorical Cross Entropy (CCE), Kullback-Leibler Divergence (KL)

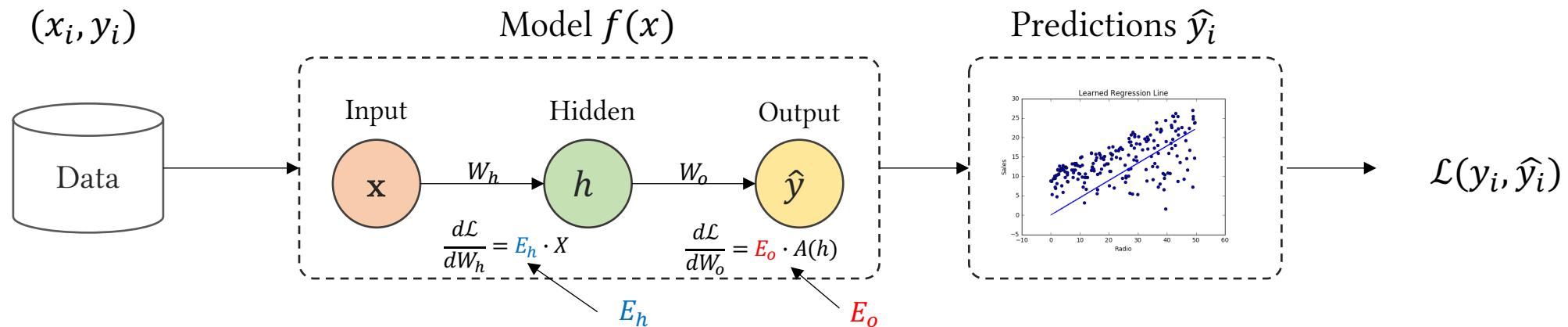
# Forward Computations

- Collect Annotated Data
- Define Model and initialize randomly
- Make a prediction using the current model
  - Typically, referred to as forward propagation or inference
  - Store all intermediate outputs
- Evaluate predictions



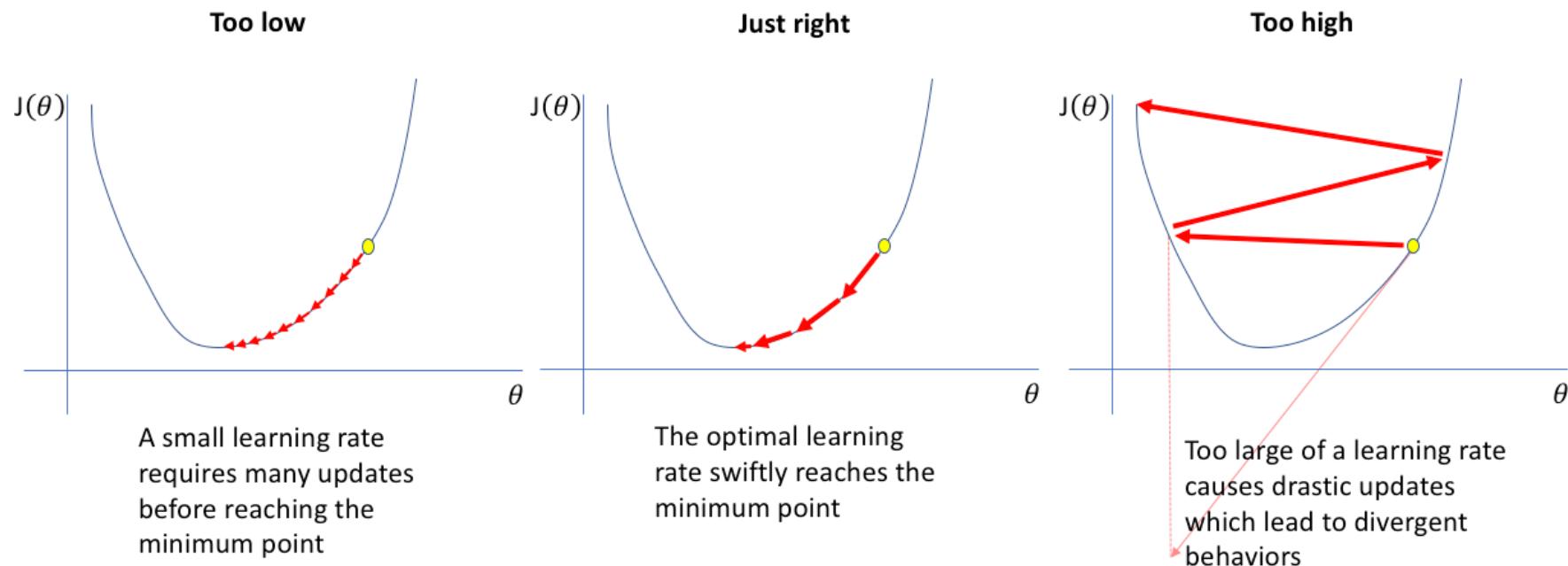
# Backward Computations

- Backpropagate the errors
  - Use the stored intermediate outputs and the errors using the chain rule to get the gradient
- Update the trainable parameters using the direction of the negative gradient
- Iterate until convergence or using early stopping criteria



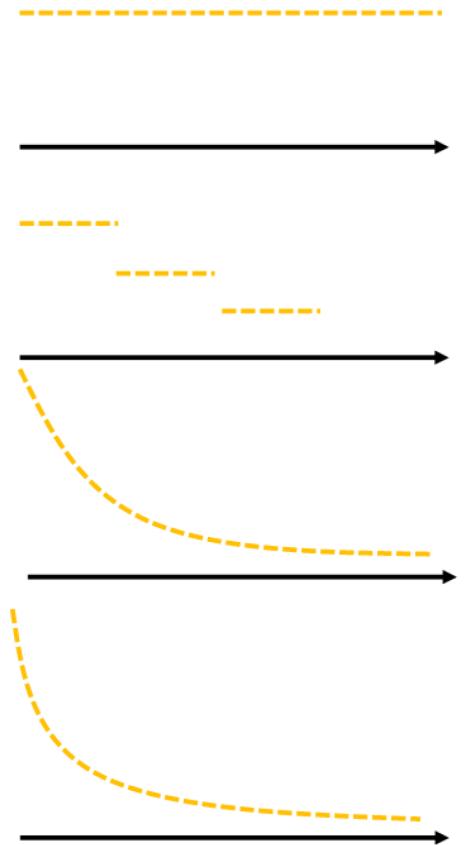
# Learning Rate

- The right learning rate  $\lambda$  very important for fast convergence
  - Too high -> strong gradients that overshoot and bounce
  - Too little -> slow training



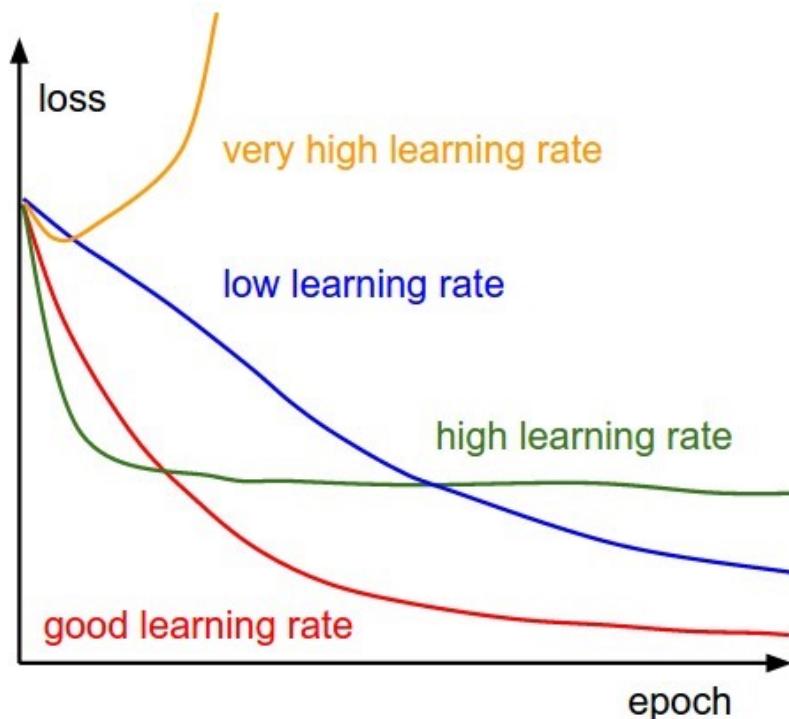
# Learning Rate Schedules

- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease (half) learning rate every T epochs or when validation loss reaches a plateau
- Inverce decay
  - $\lambda_t = \frac{\lambda_0}{1+\varepsilon t}$
- Exponential decay
  - $\lambda_t = \lambda_0 e^{-\varepsilon t}$
- Often step decay is preferred
  - Simple intuitive, work well



# In Practice

- Try several log-spaced values  $10^{-1}, 10^{-2}, 10^{-3}, \dots$  on a smaller set
- Then you can narrow it down from there around where you get the lowest errors



# Take Home Messages

- Deep learning and neural networks are based on simple mathematical functions.
- Different modules can be used and combined depending on the data and the problems we need to address.
- Depending on the task at hand the appropriate modules should be selected!
  - Regression: Linear, Sigmoid, L1, L2
  - Classification: Softmax, CCE
- Training is happening with gradient descend (gradient should be available almost everywhere).
- The parameters are optimized by backpropagating errors.
- Backpropagating errors is the heart of optimizing parametrical models like Deep Neural Networks.