Università
degli Studi
di Padova

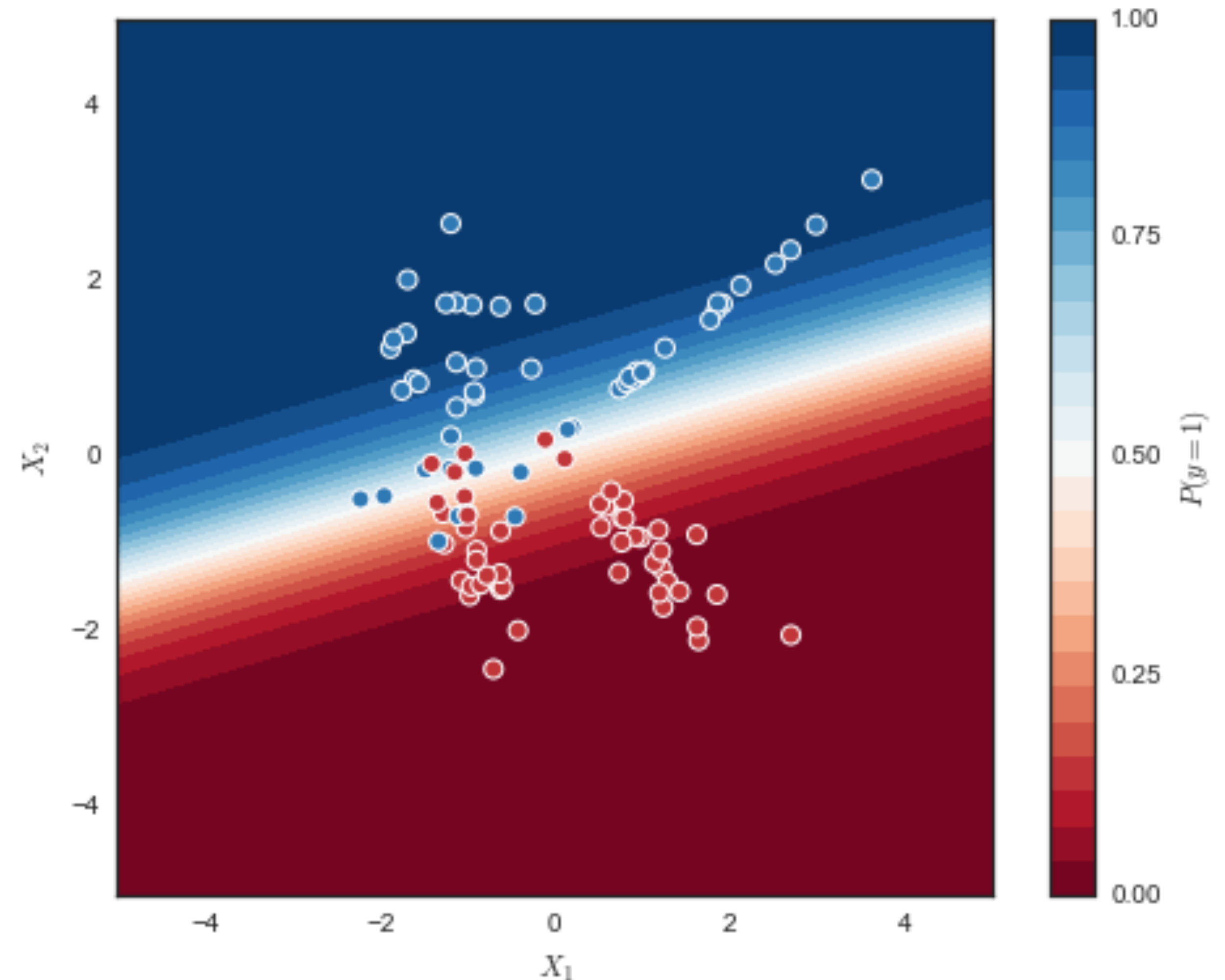
Machine Learning
2024/2025

Lecture #24
Neural Networks Training

Gian Antonio Susto

Before Starting 1/2: clarifications on ROC curve

In many approaches (logistic regression, SVM, ...) we can derive a distance from the decision boundary, a probability of being classified to one class or another...



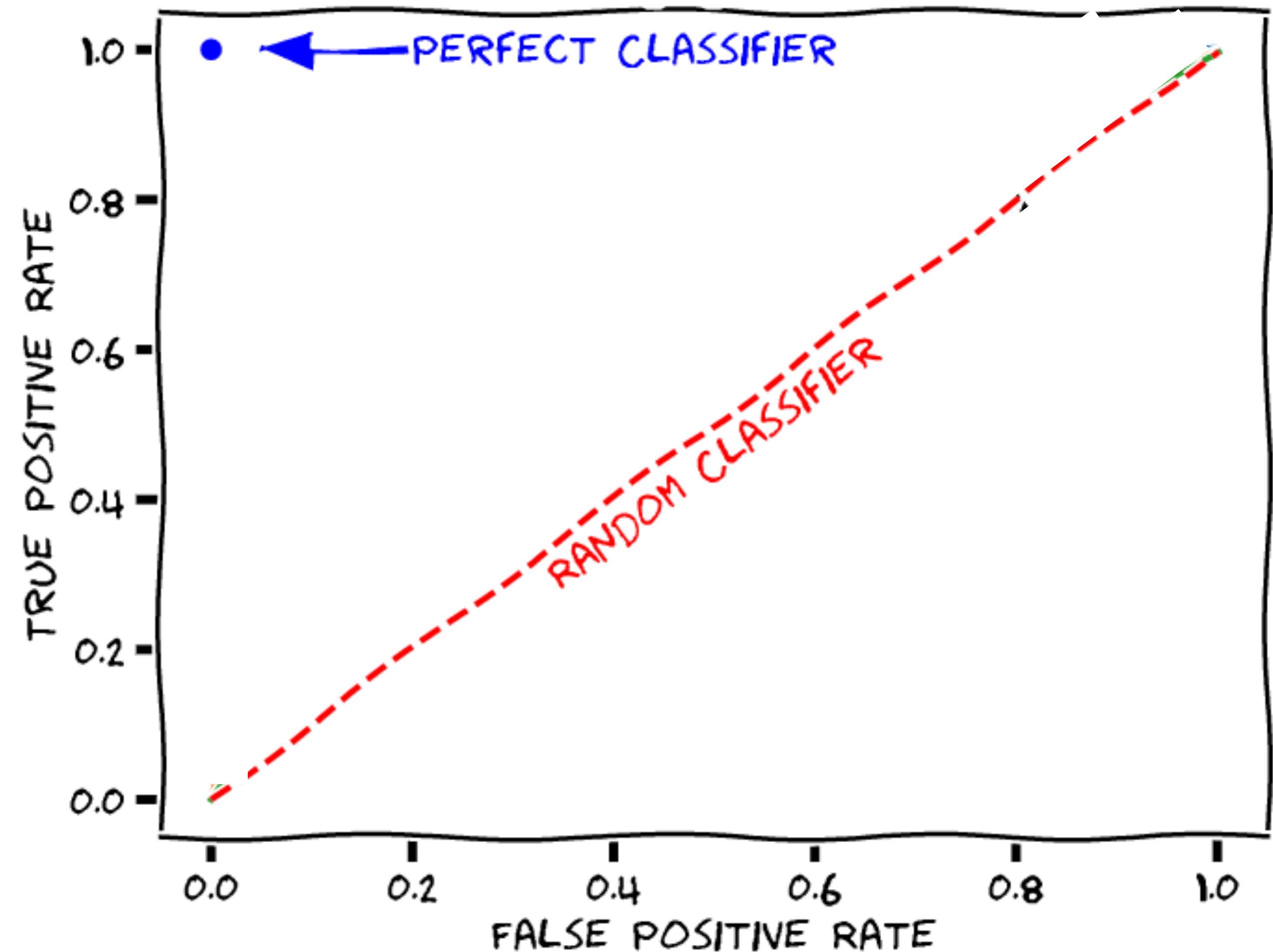
Before Starting 1/2: clarifications on ROC curve

- True Positive Rate

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

- False Positive Rate

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$



Before Starting 1/2: clarifications on ROC curve

- True Positive Rate

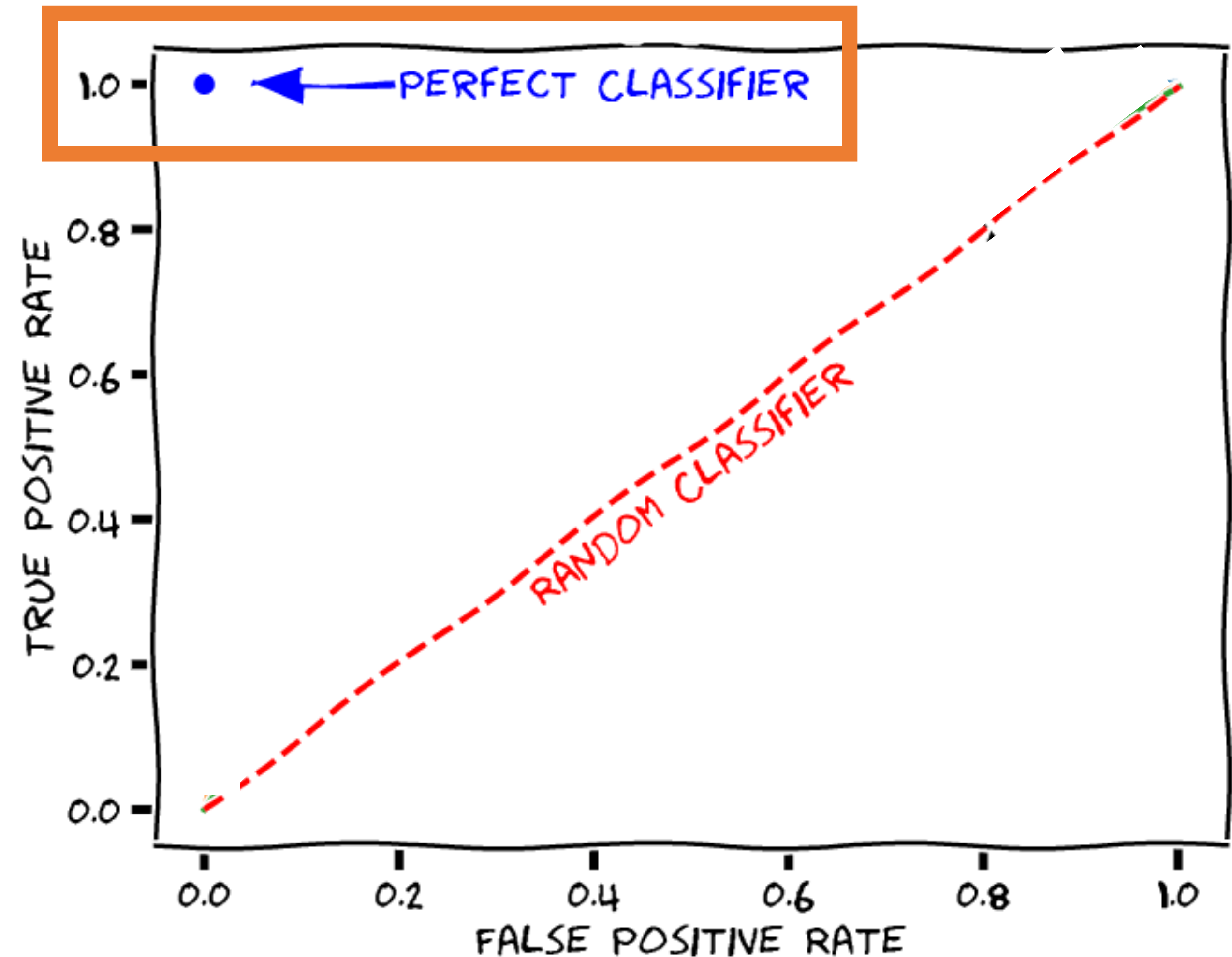
$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

- False Positive Rate

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

In this case:

- TPR = 1 (0 false negatives, ie all TP or TN)!
- FPR = 0 (0 false positives)



Before Starting 1/2: clarifications on ROC curve

- True Positive Rate

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

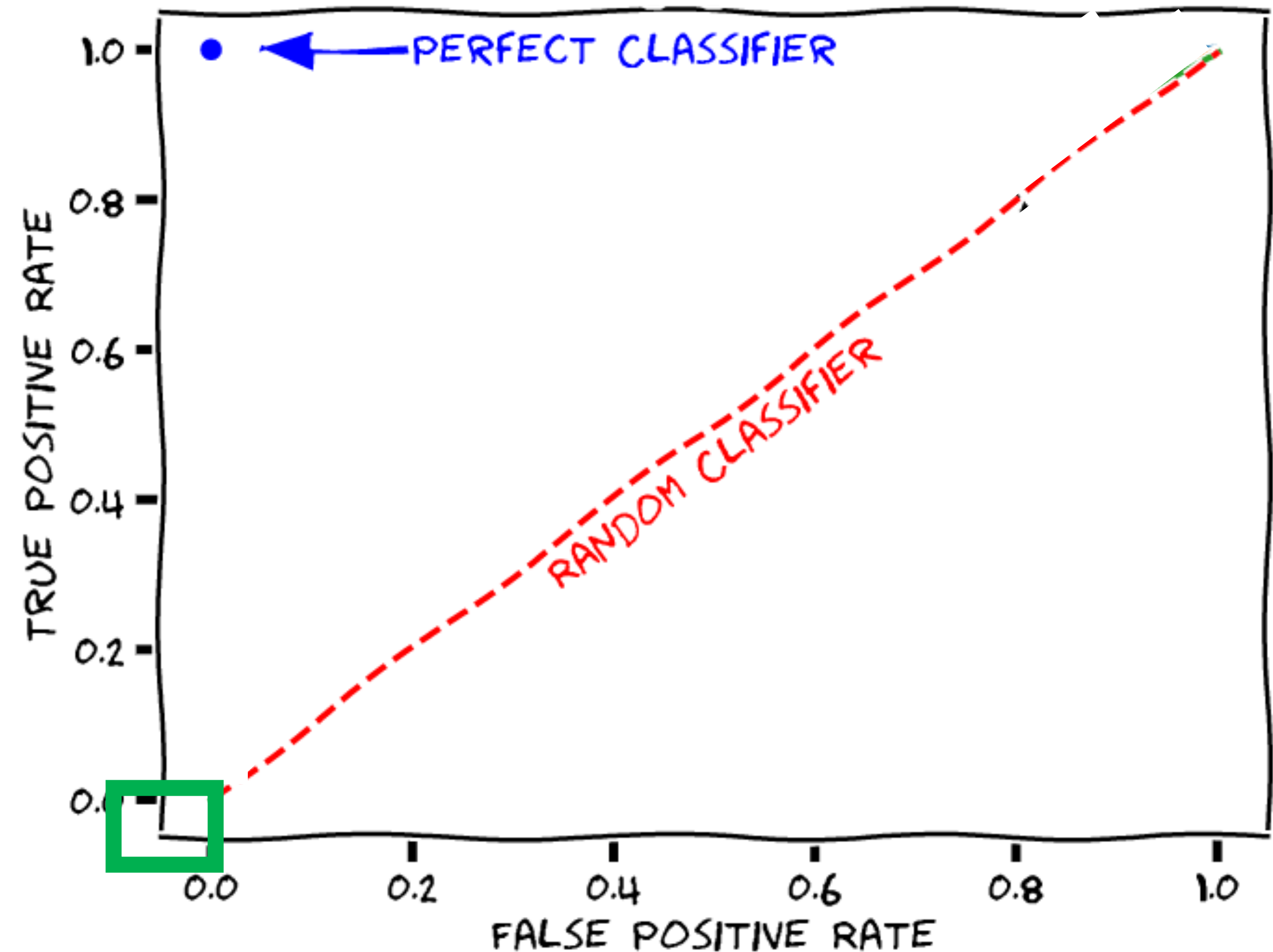
- False Positive Rate

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

If a classifier always classifies data as negatives:

- TPR = 0 (as TP = 0)

- FPR = 0 (as FP = 0)



Before Starting 1/2: clarifications on ROC curve

- True Positive Rate

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

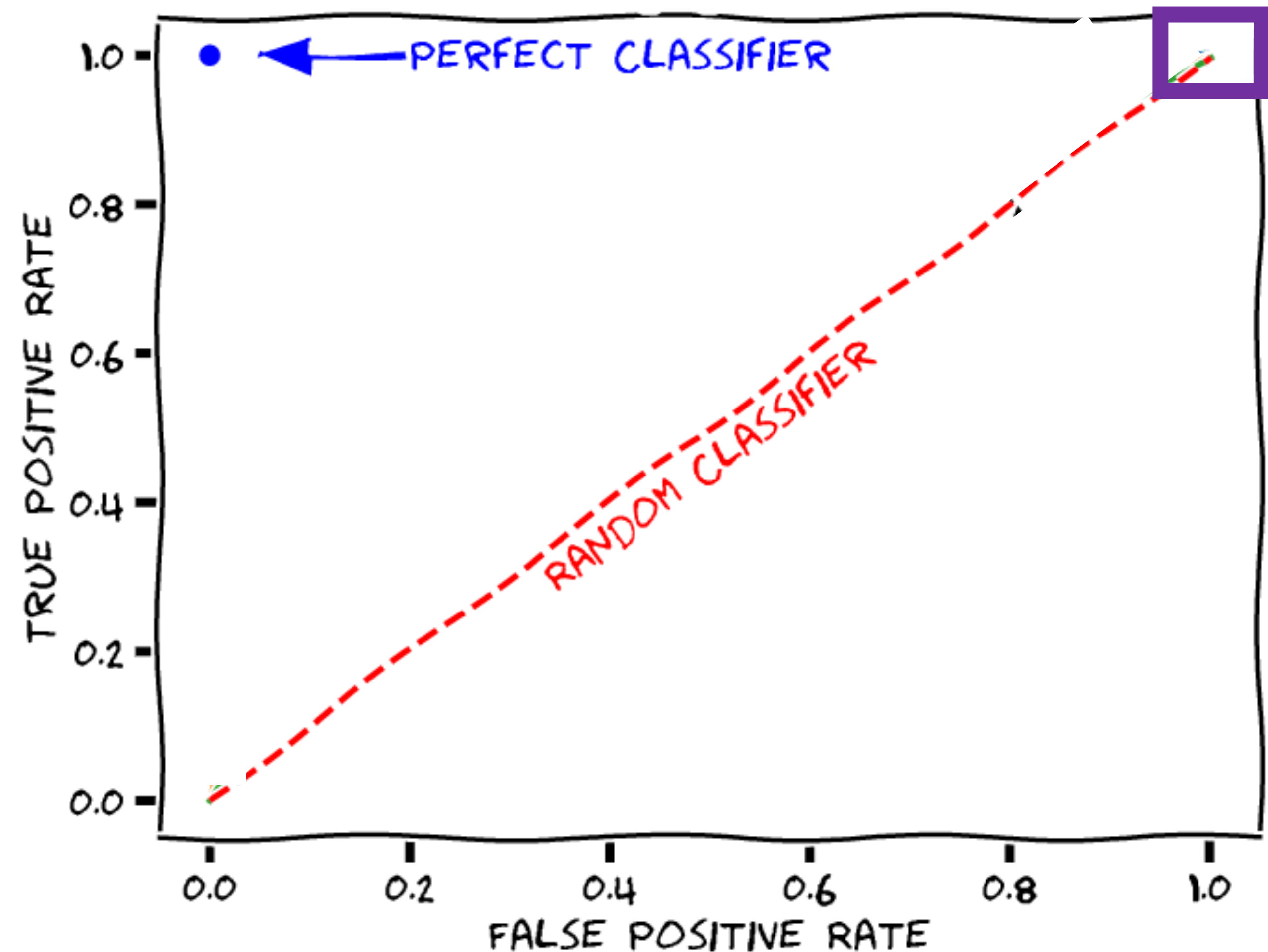
- False Positive Rate

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

If a classifier always classifies data as positives:

- TPR = 1 (as FN = 0)

- FPR = 1 (as TN = 0)



Before Starting 1/2: clarifications on ROC curve

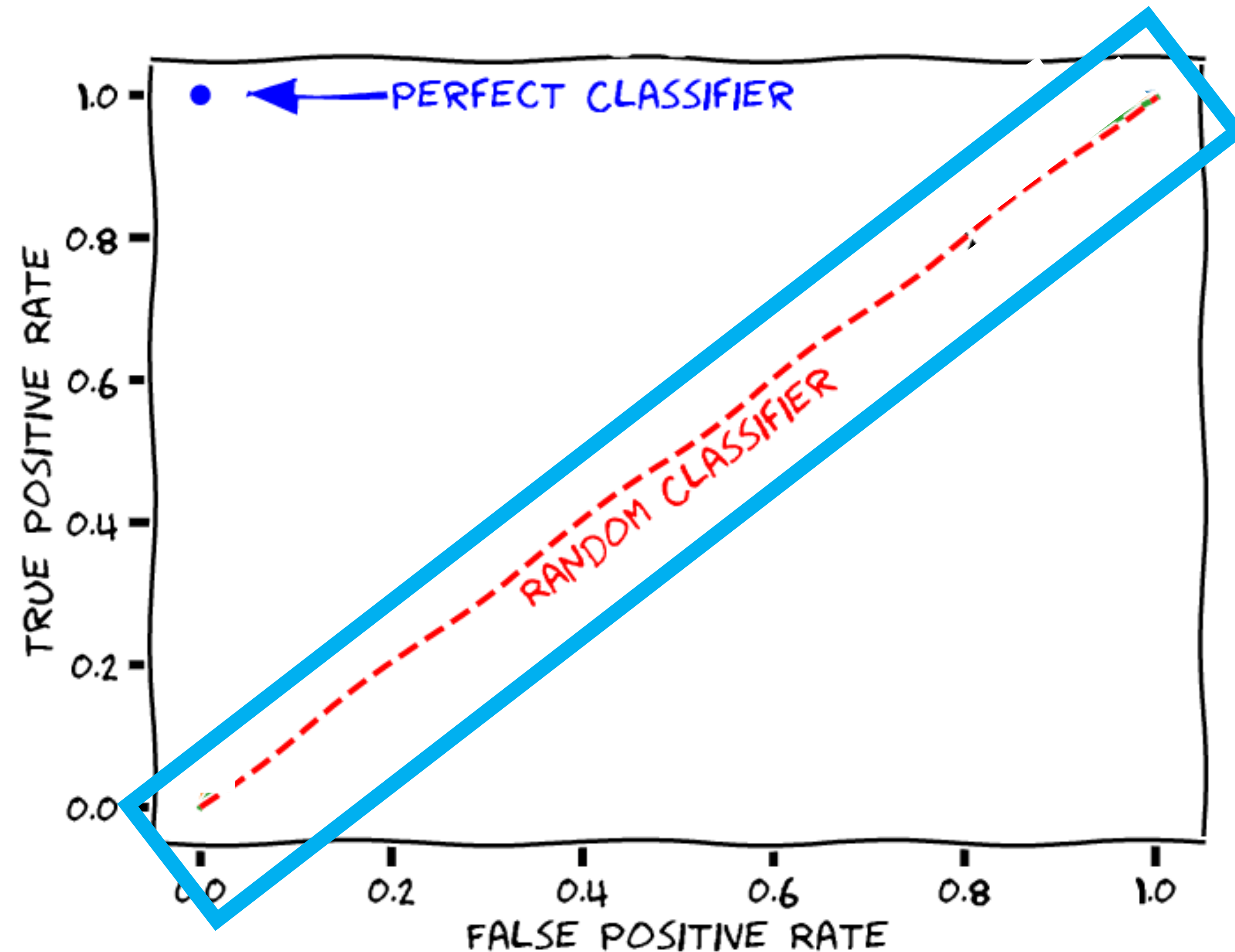
- True Positive Rate

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- False Positive Rate

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

A random classifier will have TPR = FPR since positives and negatives are assigned randomly



Before Starting 1/2: clarifications on ROC curve

- True Positive Rate

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

- False Positive Rate

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

Scenario: A dataset with 100 people (50 positives, 50 negatives)

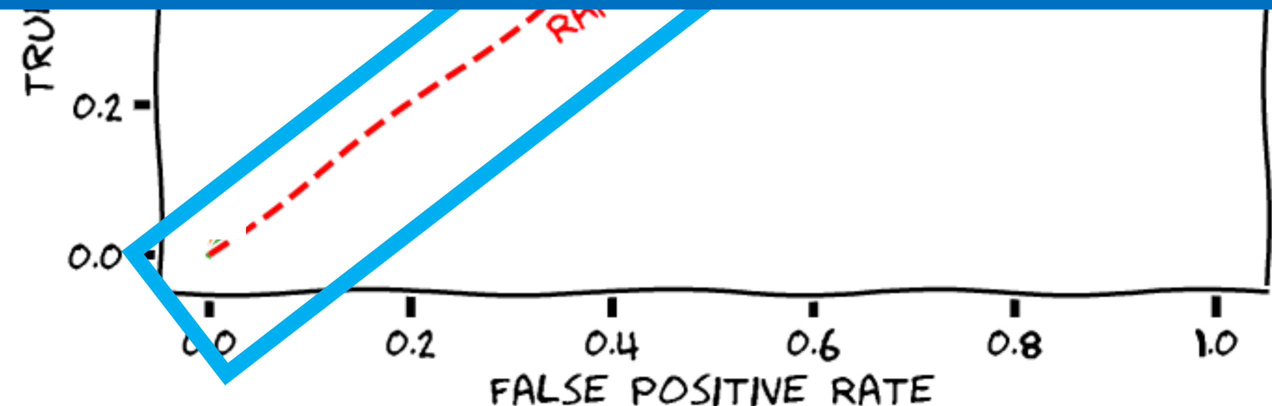
- CASE #01 (Threshold = 0.8, ie 20% are predicted as P, TP = 10, FP = 10, TN = 40, FN = 40)

TPR = 10/(10+40) = 0,2 & FPR = 10/(10+40) = 0,2

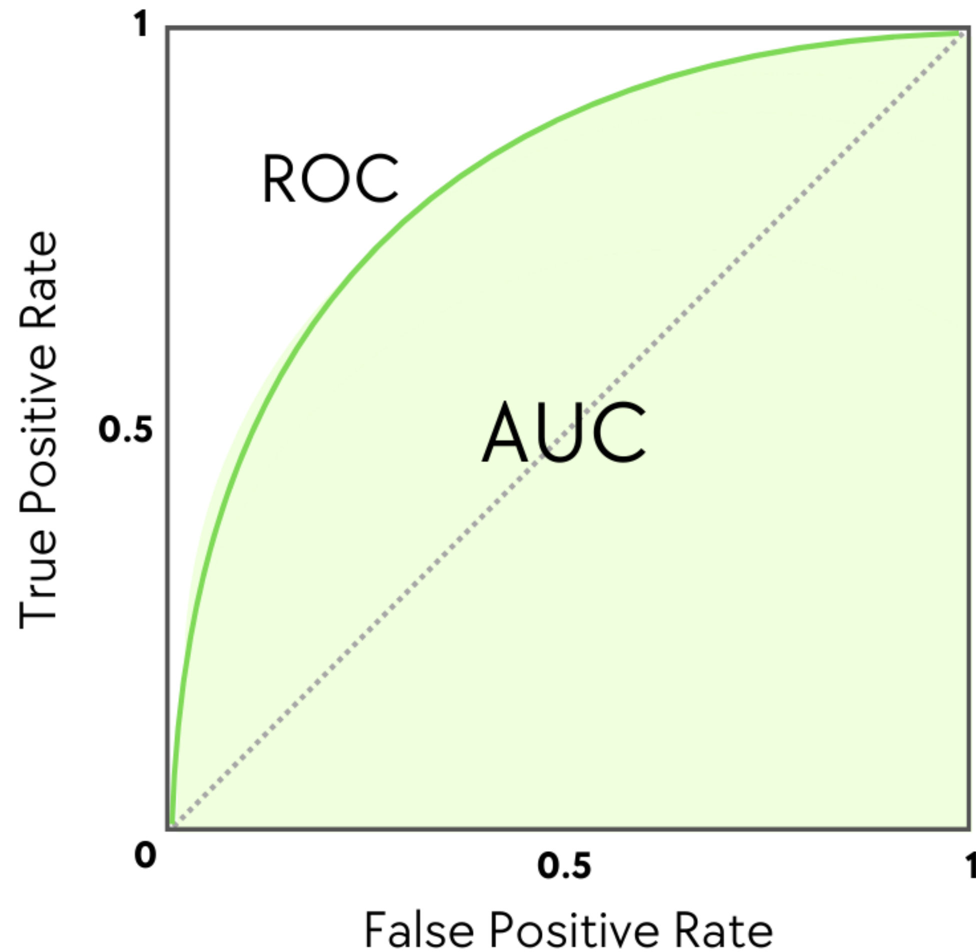
- CASE #02 (Threshold = 0.5, ie 50% are predicted as P, TP = 25, FP = 25, TN = 25, FN = 25)

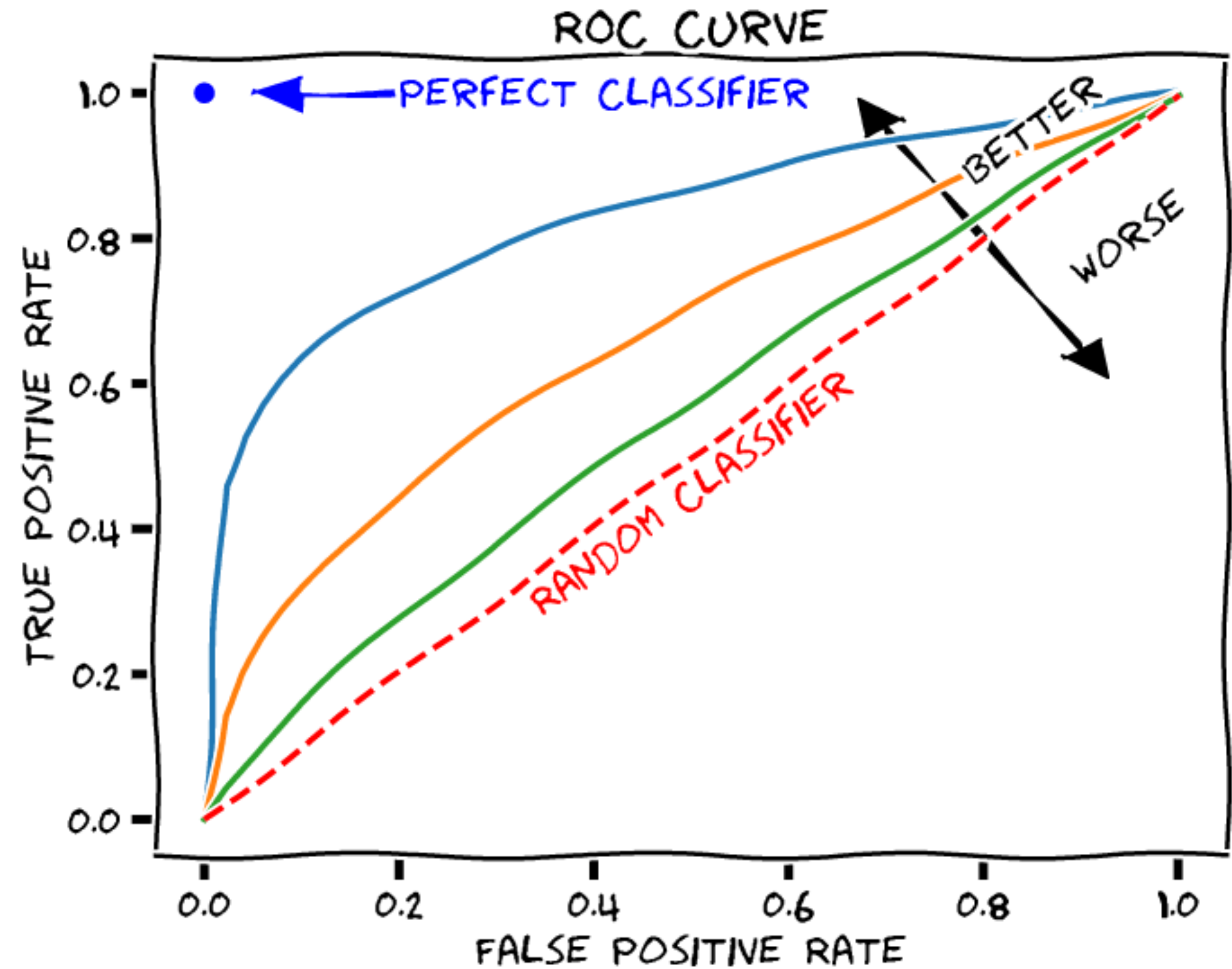
TPR = 25/(25+25) = 0,5 & FPR = 25/(25+25) = 0,5

A random classifier will have TPR = FPR since positives and negatives are assigned randomly

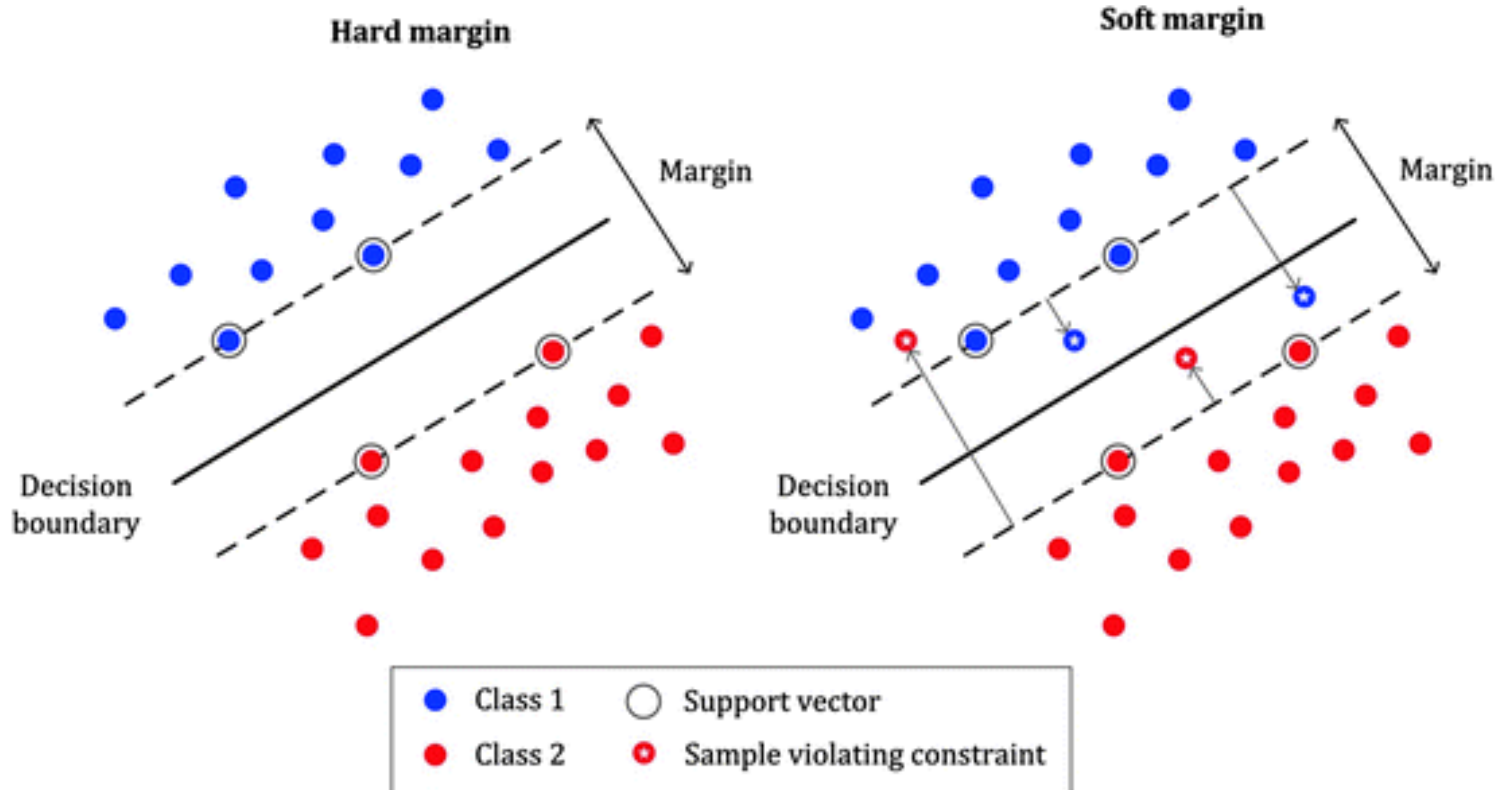


Before Starting 1/2: clarifications on ROC curve





Before Starting 2/2: on the role of C in SVM



Before Starting 2/2: on the role of C in SVM

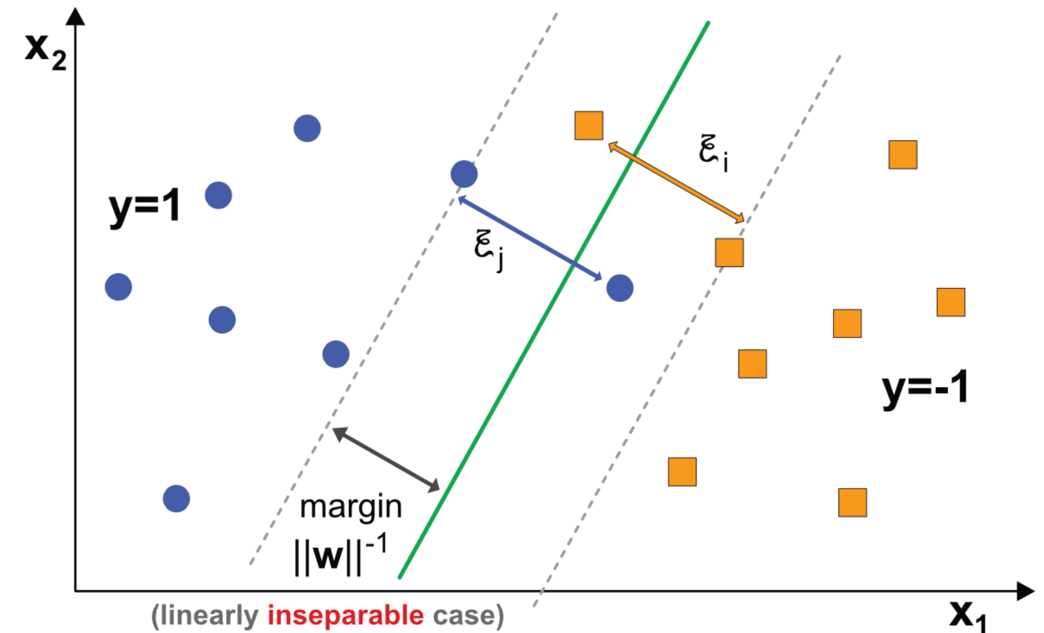
SVM solves a convex optimization problem (linearly separable case):

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \qquad \text{Subject to} \qquad y_i(w \cdot x_i + b) \geq 1 \quad \forall i$$

If it is not separable, we allow some "slack" (errors) by introducing variables ξ:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum \xi_i$$

Subject to $y_i(w \cdot x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$



Before Starting 2/2: on the role of C in SVM

Blobs (linear + noisy) - C=0.01

Blobs (linear + noisy) - C=1

Blobs (linear + noisy) - C=100

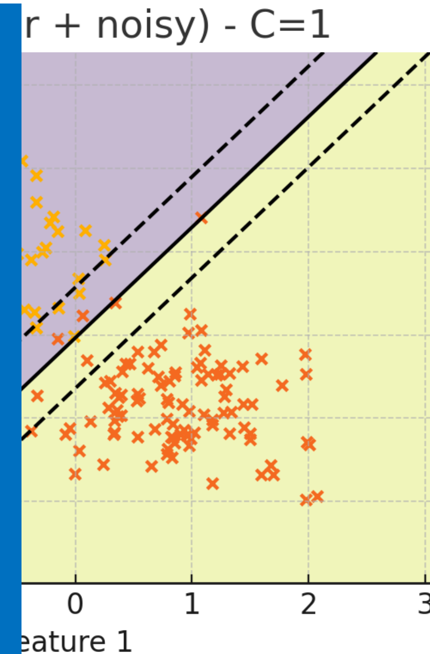
Class 0
Class 1

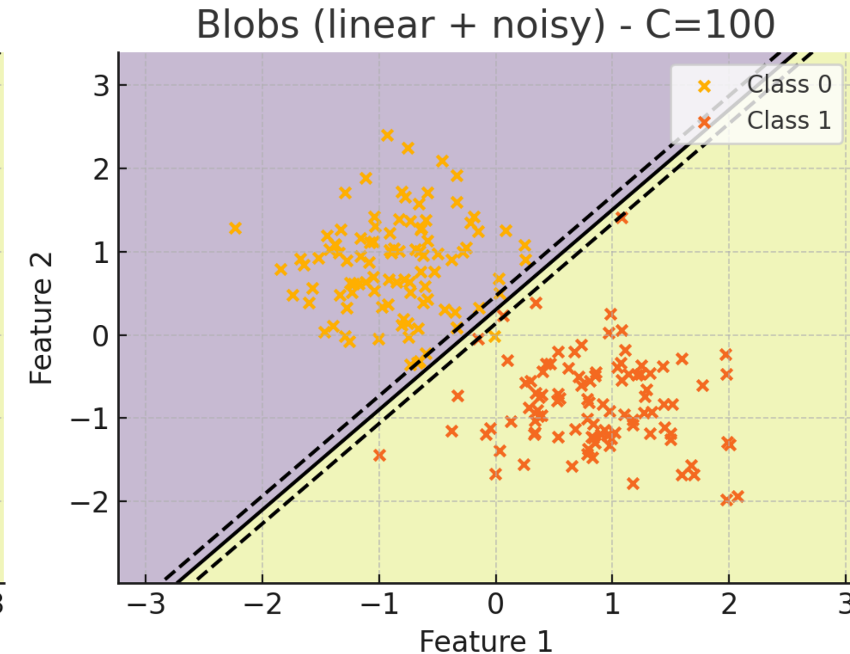
Feature 1
Feature 2

Here we allow the classifier to have some misclassification

Here we try to have all historical data point correctly classified

Before Starting 2/2: on the role of C in SVM





<u>C high</u>

- Low regularization

- The model penalizes misclassifications heavily

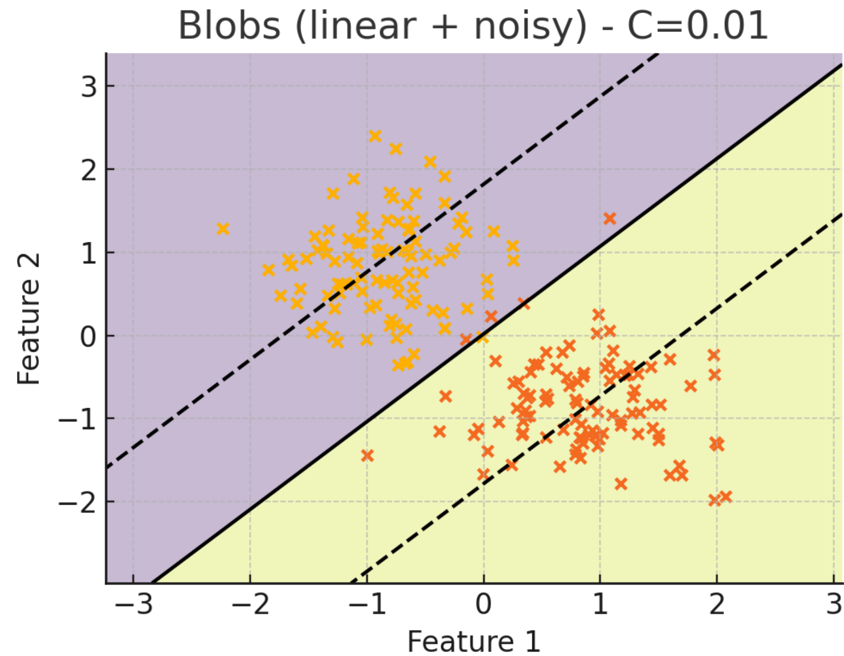
- Tries to classify every point correctly (even noisy points)

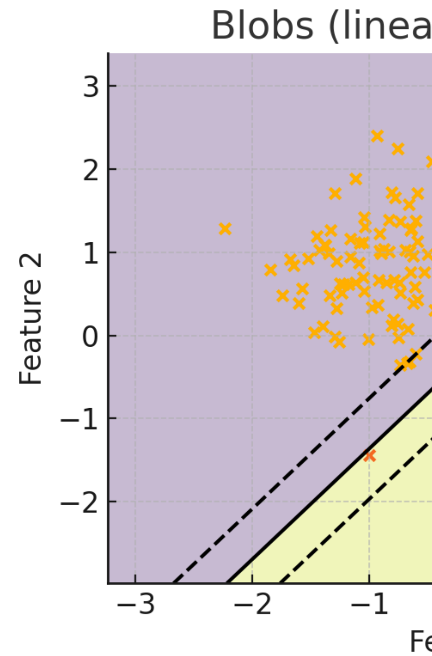
- Margin becomes narrower to avoid errors

- Risk of overfitting

Here we try to have all historical data point correctly classified

Before Starting 2/2: on the role of C in SVM





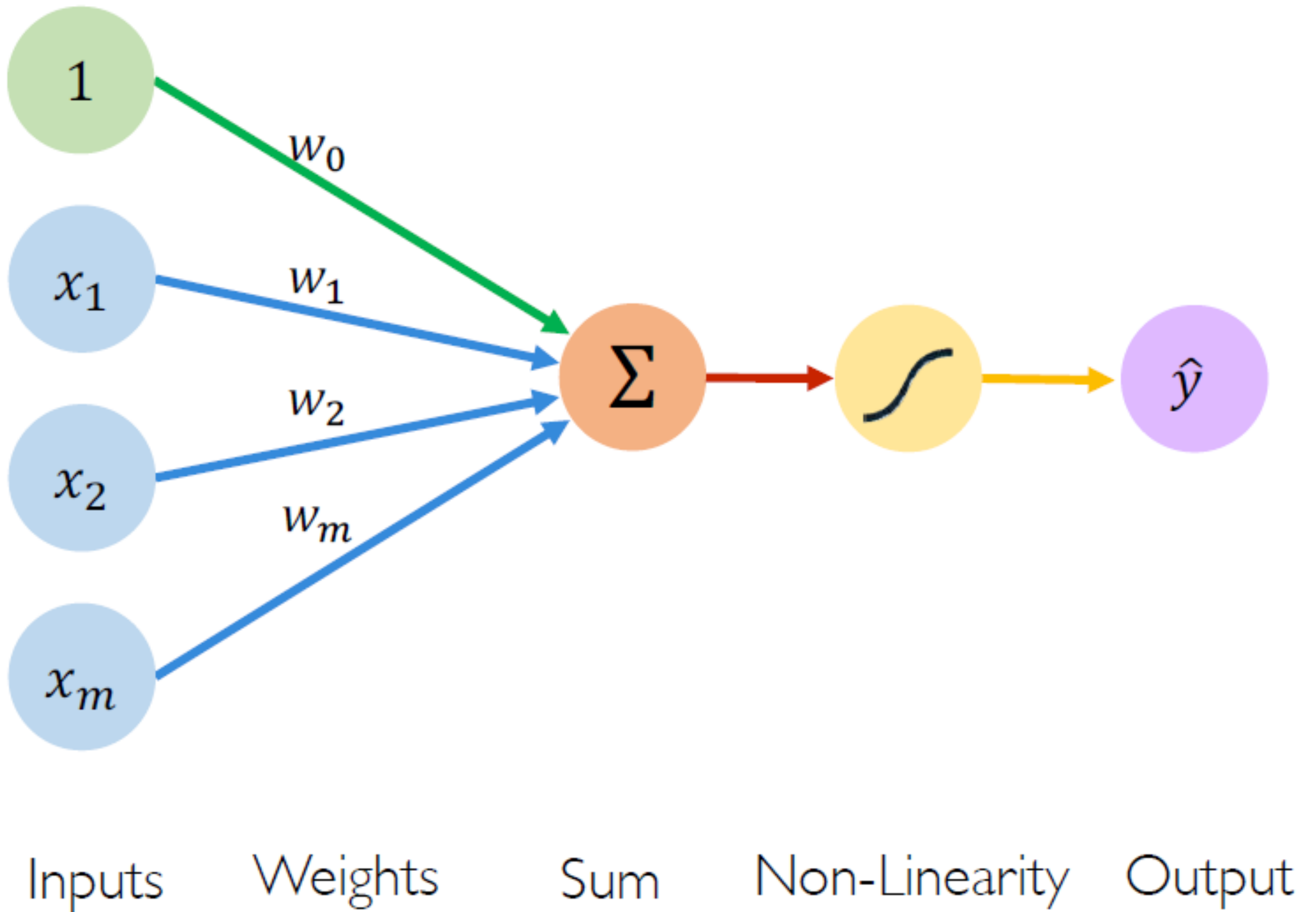
Here we allow the classifier to have some misclassification

C low

- High regularization
- Allows some classification errors to achieve a wider margin
- Focuses more on simplicity and generalization
- Risk of underfitting, but better performance on noisy data

Basics of Deep Learning (with some recap)

The building block of Neural Networks (NN): the neuron



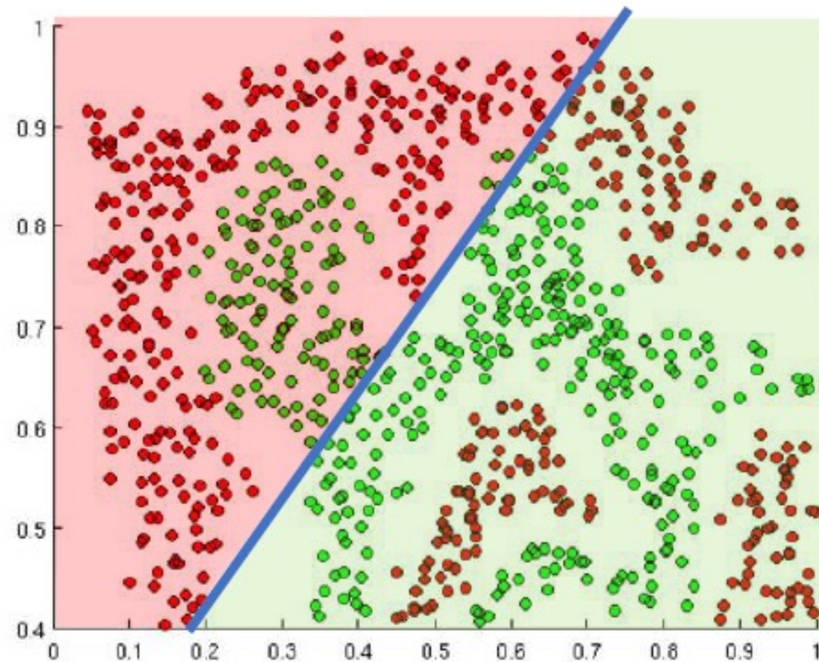
Activation function

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

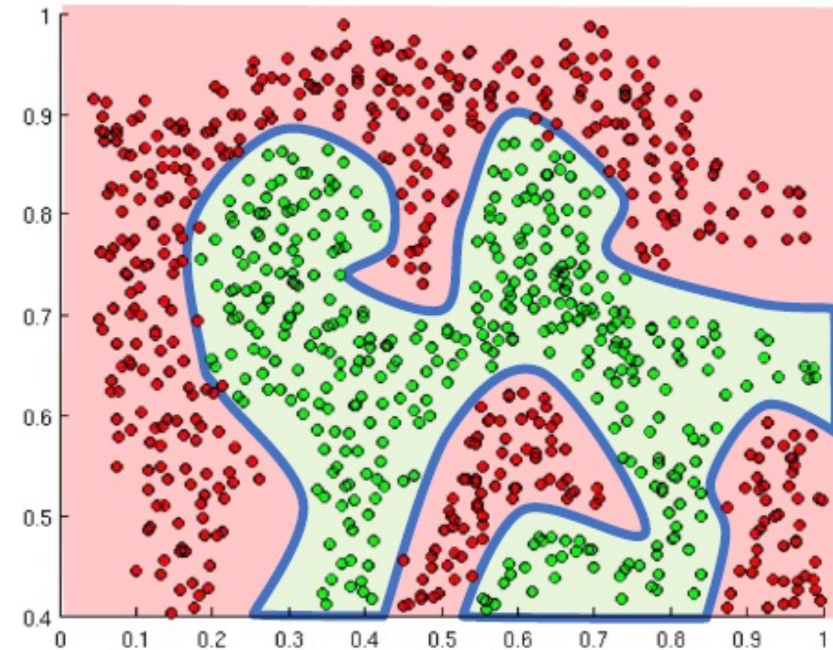
where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

Activation functions



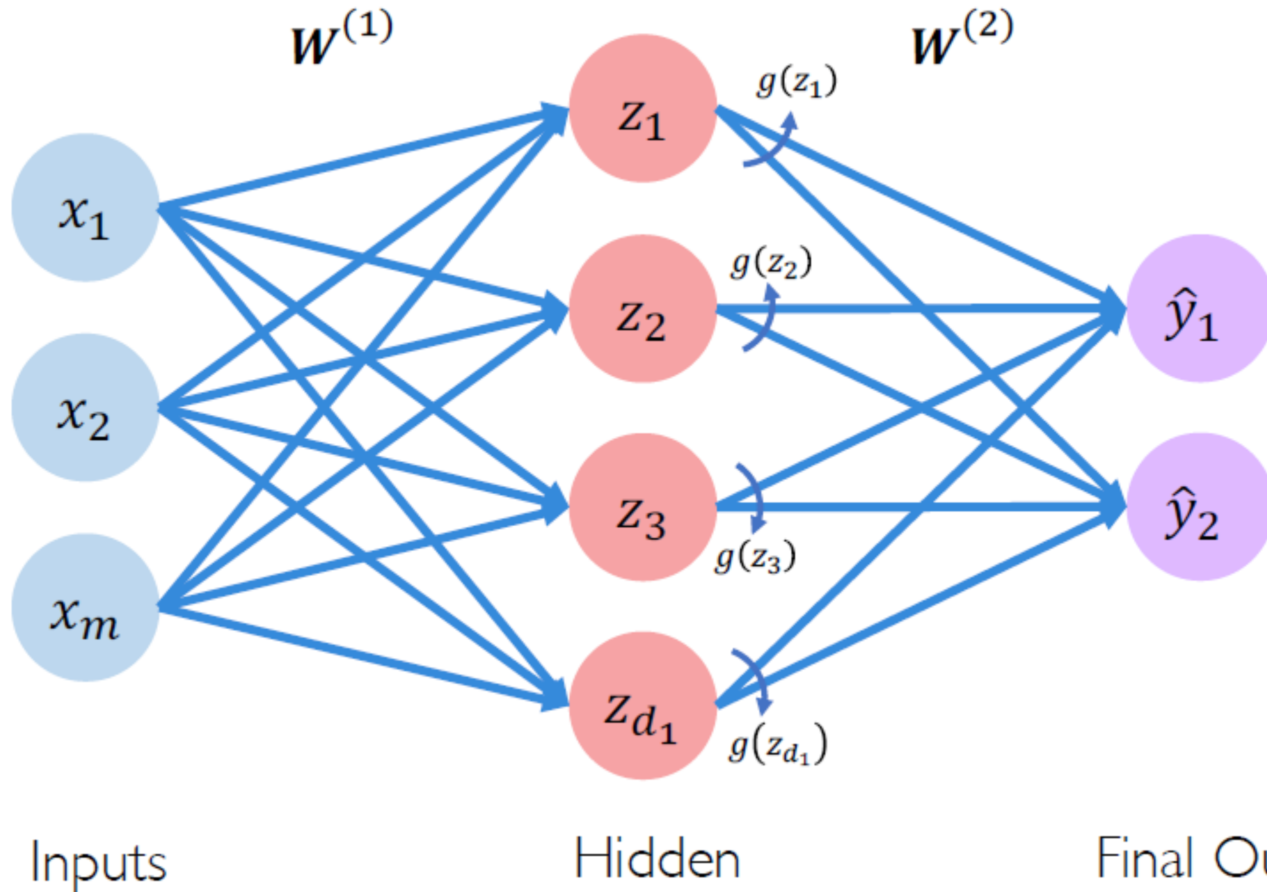
Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

Feedforward (Single Layer) Neural Network



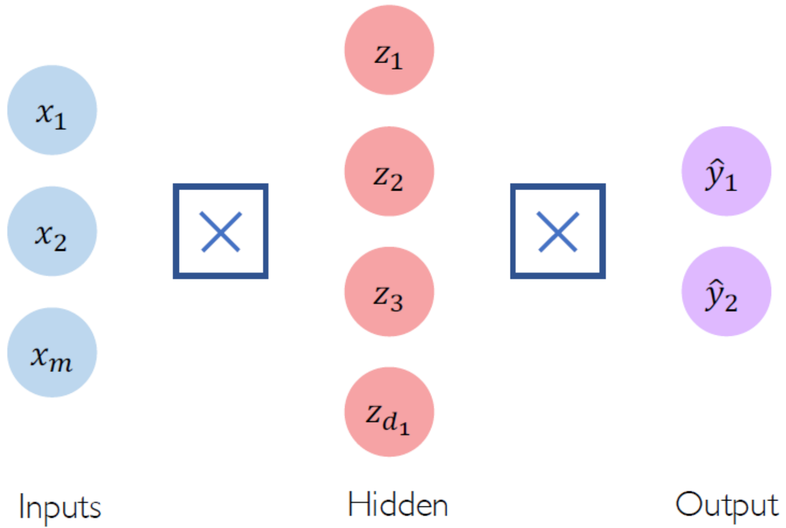
So-called *Vanilla Neural Network*

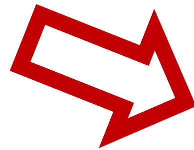
T. Hastie, R. Tibshirani, J. Friedman *The Elements of Statistical Learning*

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

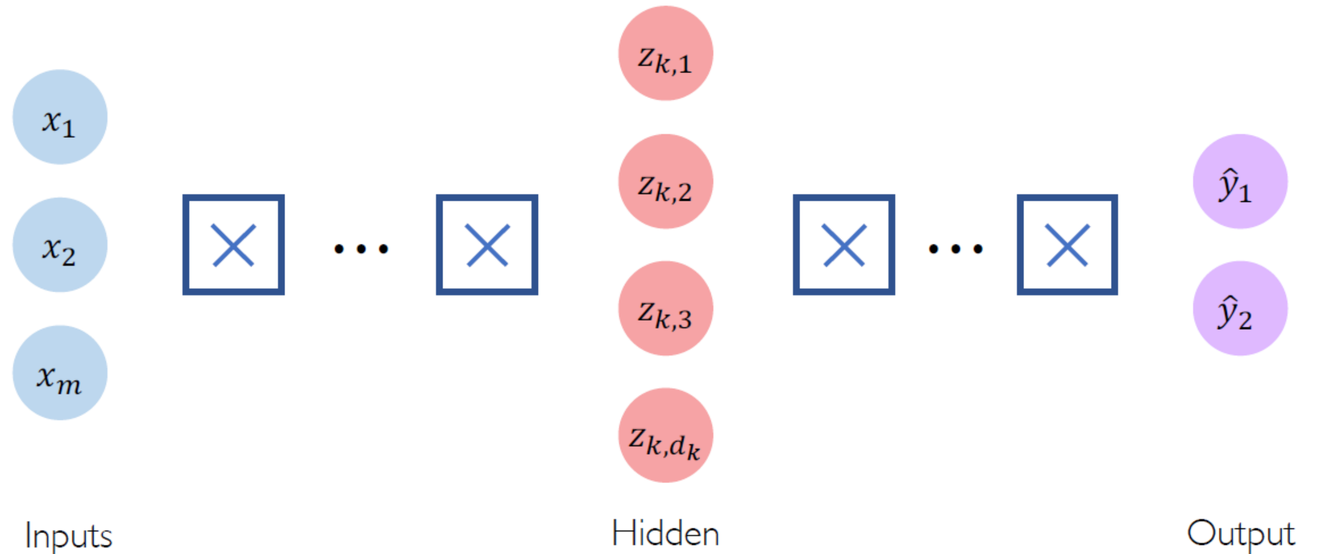
MIT *Introduction to Deep Learning* http://introtodeeplearning.com

From Neural Network to Deep Learning architecture



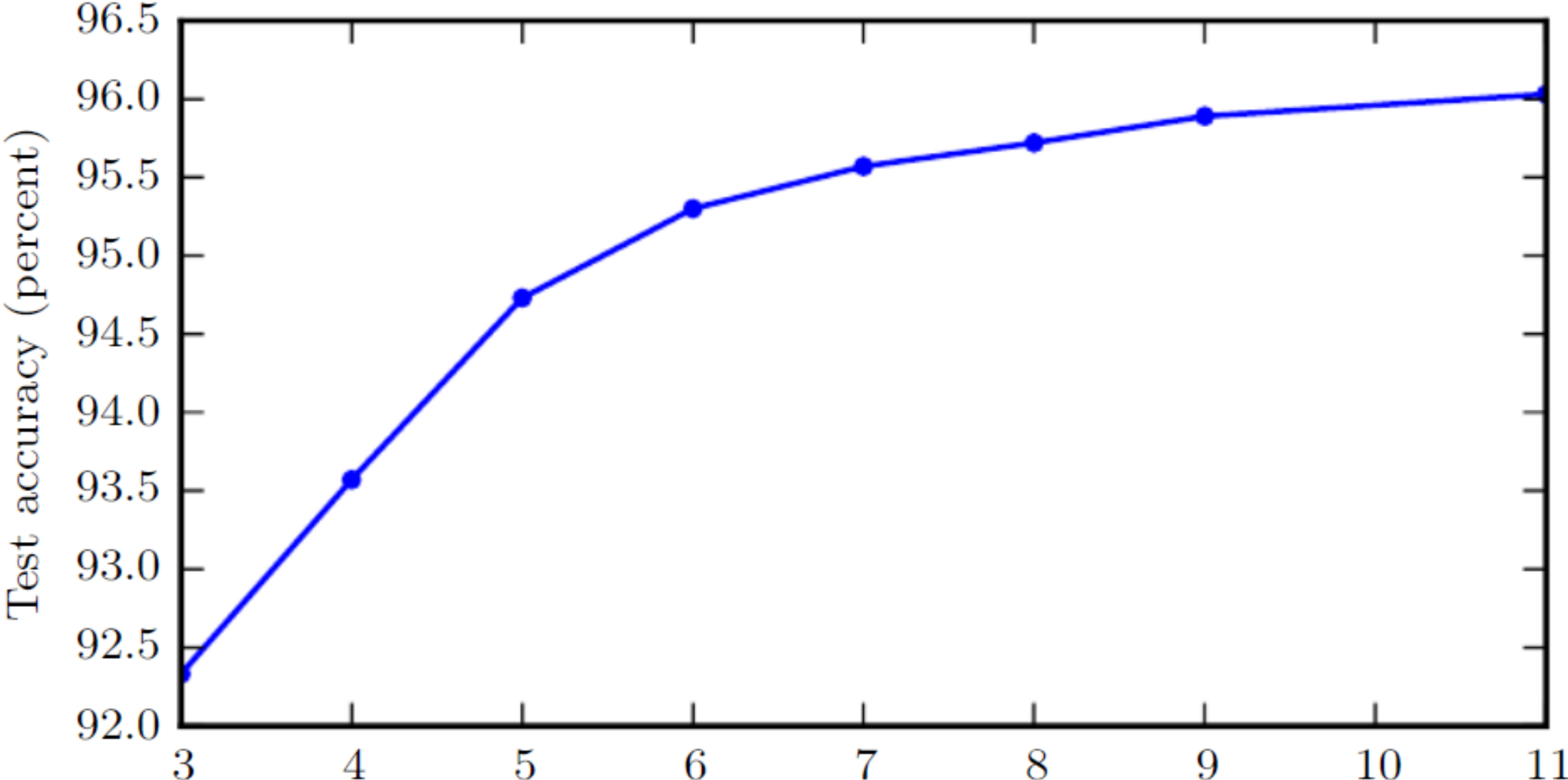


From 1 Layer to ‘many’ layers



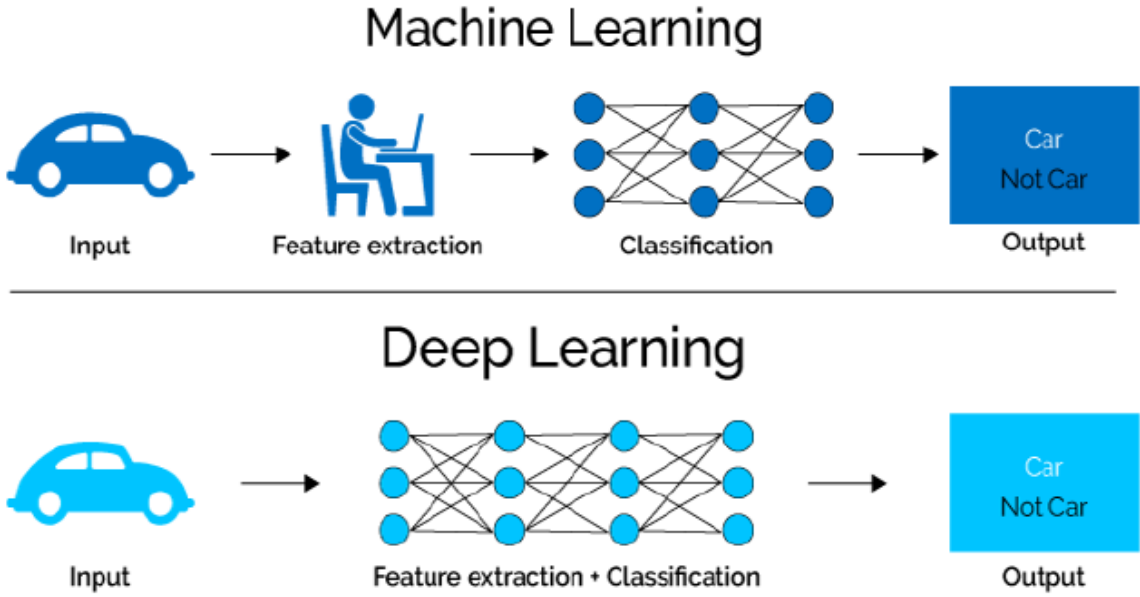
$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

From Neural Network to Deep Learning architecture



Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

One of the advantages of DL is to do ‘embedded’ both the feature extraction and the modeling part

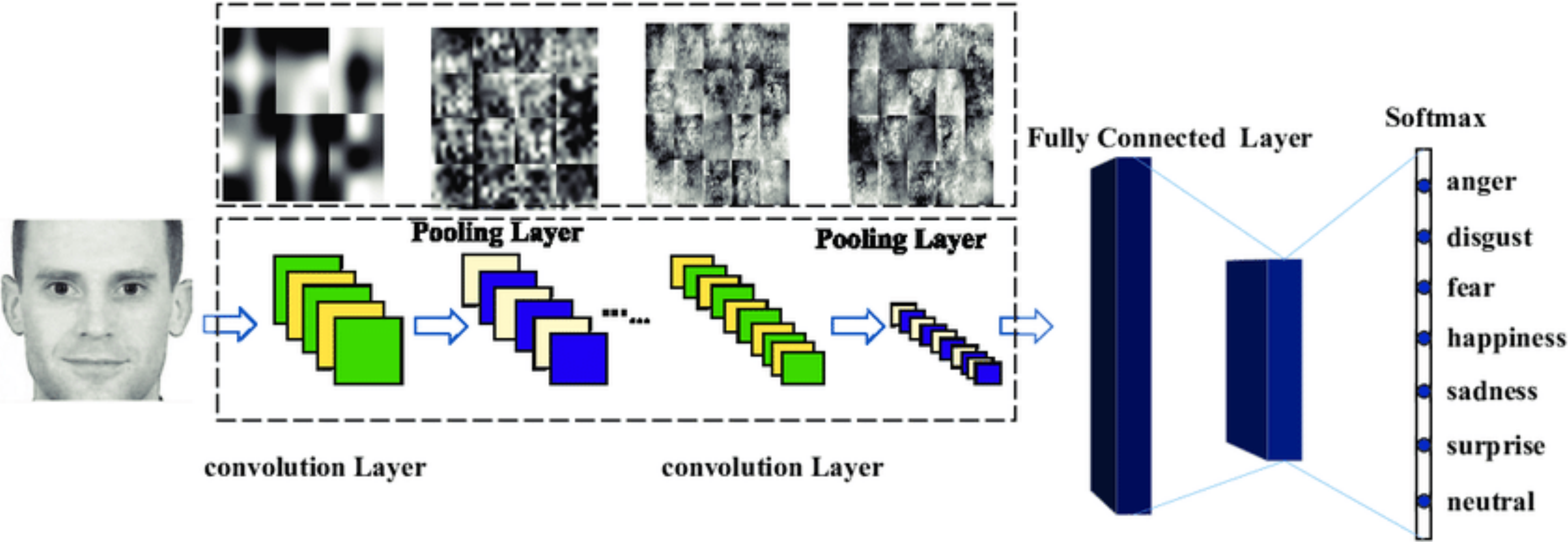


L. Fridman MIT Deep Learning https://deeplearning.mit.edu/

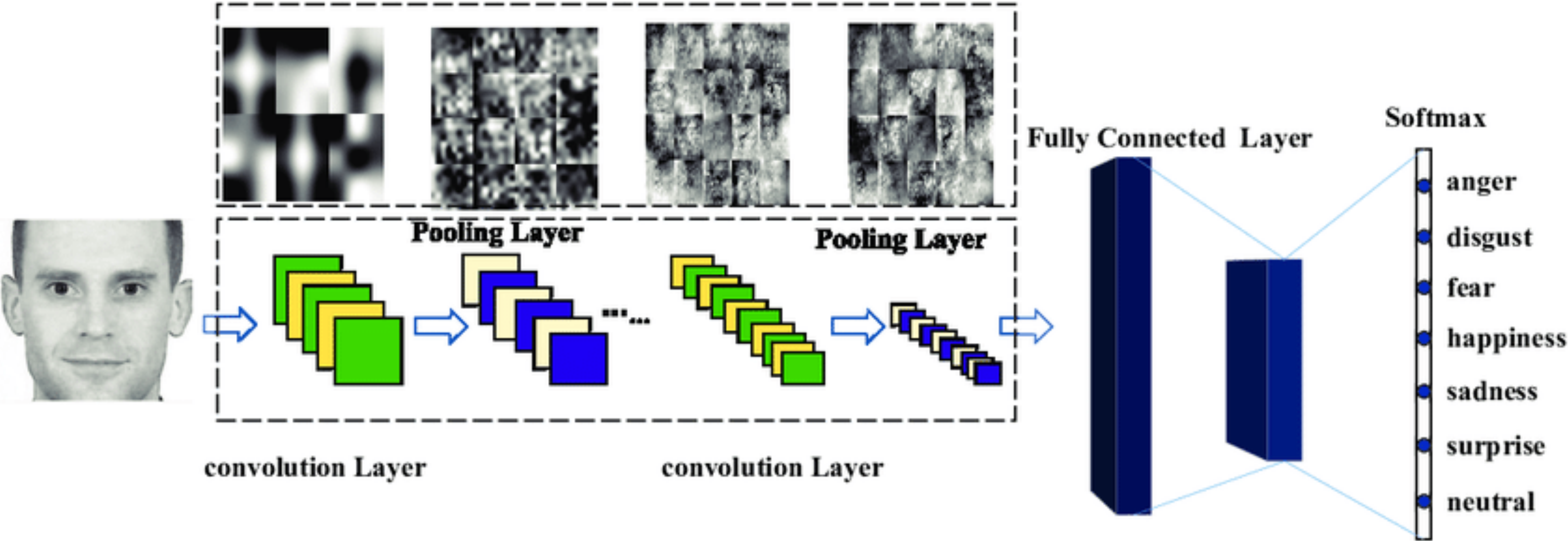
Output
Output
Output
Mapping from features
Output
Mapping from features
Mapping from features
Additional layers of more abstract features
Hand-designed program
Hand-designed features
Features
Simple features
Input
Input
Input
Input
Rule-based systems
Classic machine learning
Deep learning
Representation learning

I. Goodfellow, Y. Bengio, A. Courville. Deep learning. MIT press, 2016.

One of the advantages of DL is to do ‘embedded’ both the feature extraction and the modeling part

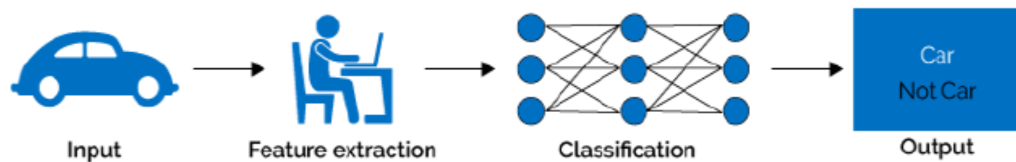


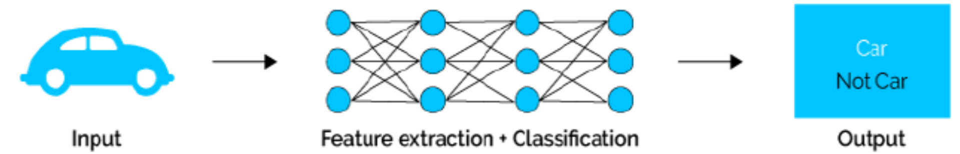
One of the advantages of DL is to do ‘embedded’ both the feature extraction and the modeling part



We are calling these, ‘learned features’

Handcrafted Features vs Learned Features





Handcrafted features are manually designed by humans based on domain knowledge and intuition.

Pros:

- Simple, interpretable
- Work well when you have strong domain expertise

Cons:

- Limited expressiveness — can't capture complex patterns
- Often task-specific — poor generalization
- Require manual effort, which doesn't scale

Learned features are automatically extracted by a neural network during training — the network learns how to represent the data in a way that is useful for the task.

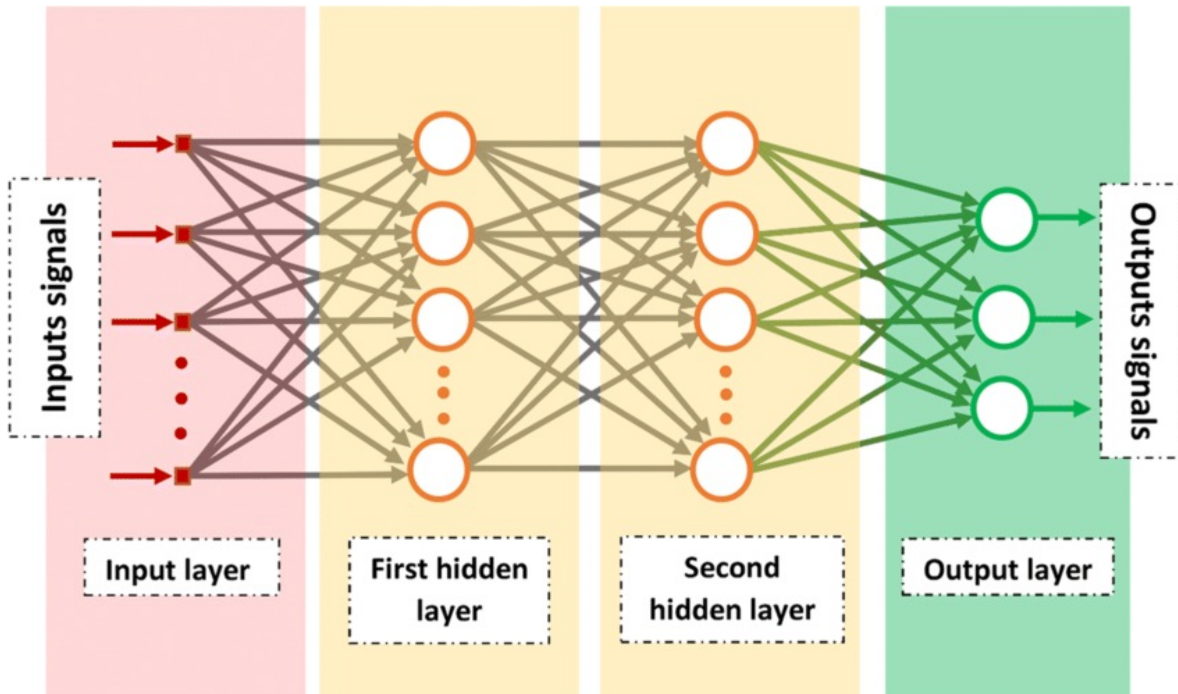
Pros:

- Can learn very complex patterns
- Adapt to the specific data and task
- Scale well to large datasets and varied problems

Cons:

- Less interpretable (but explainability tools help)
- Require more data and compute
- Can overfit if not properly regularized

Multi-output DL are particularly relevant in some cases

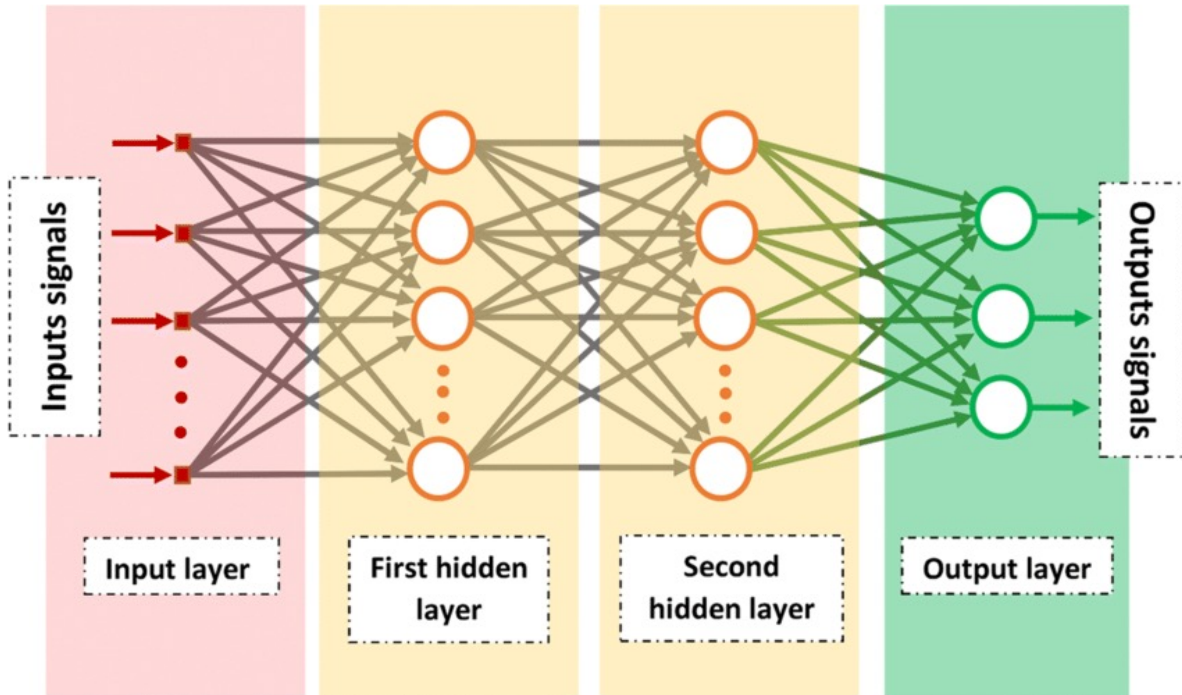


The learned features may be relevant for multiple objectives!

Some benefits:

- Efficiency (one model, one training process)
- Shared knowledge (task can benefit from each other's learned representations)
- Regularization (learning multiple tasks may prevent overfitting)

Multi-output DL are particularly relevant in some cases



For example, with O outputs, a regression loss could simply be

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{o=1}^{O} \left(\hat{y}_o^{(i)} - y_o^{(i)} \right)^2$$

The learned features may be relevant for multiple objectives!

Some benefits:

- Efficiency (one model, one training process)
- Shared knowledge (task can benefit from each other's learned representations)
- Regularization (learning multiple tasks may prevent overfitting)

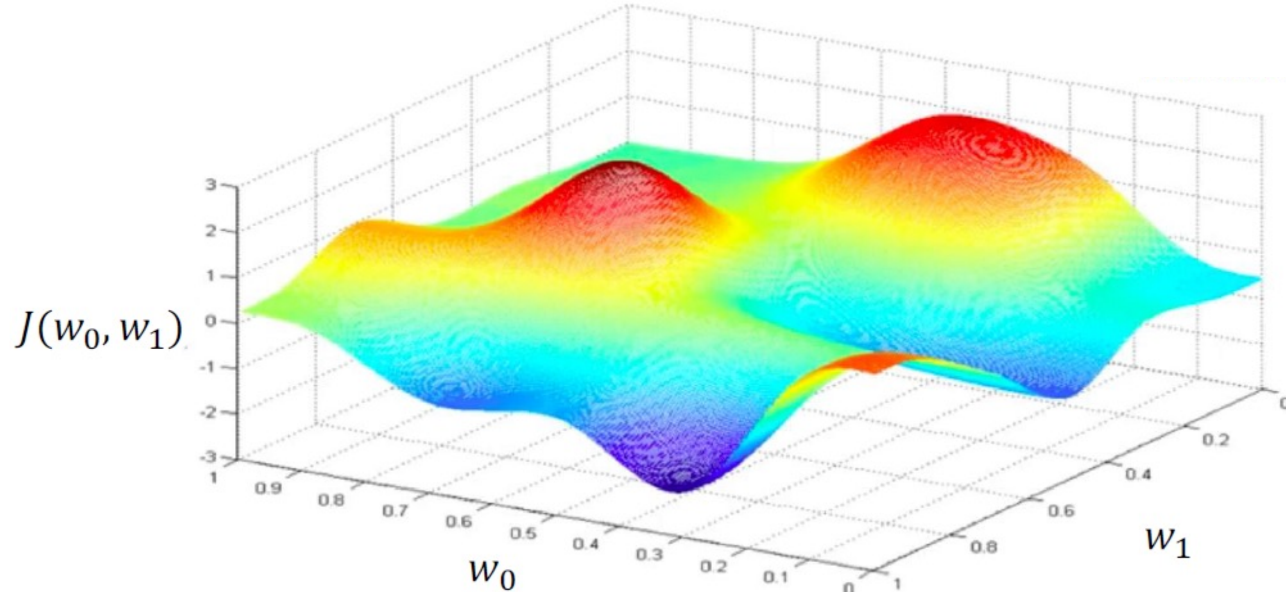
Deep Learning Training

Training a Neural Network = Minimizing a Loss

We seek for a set of weights that achieve minimal loss:

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \, J(\boldsymbol{W})$$

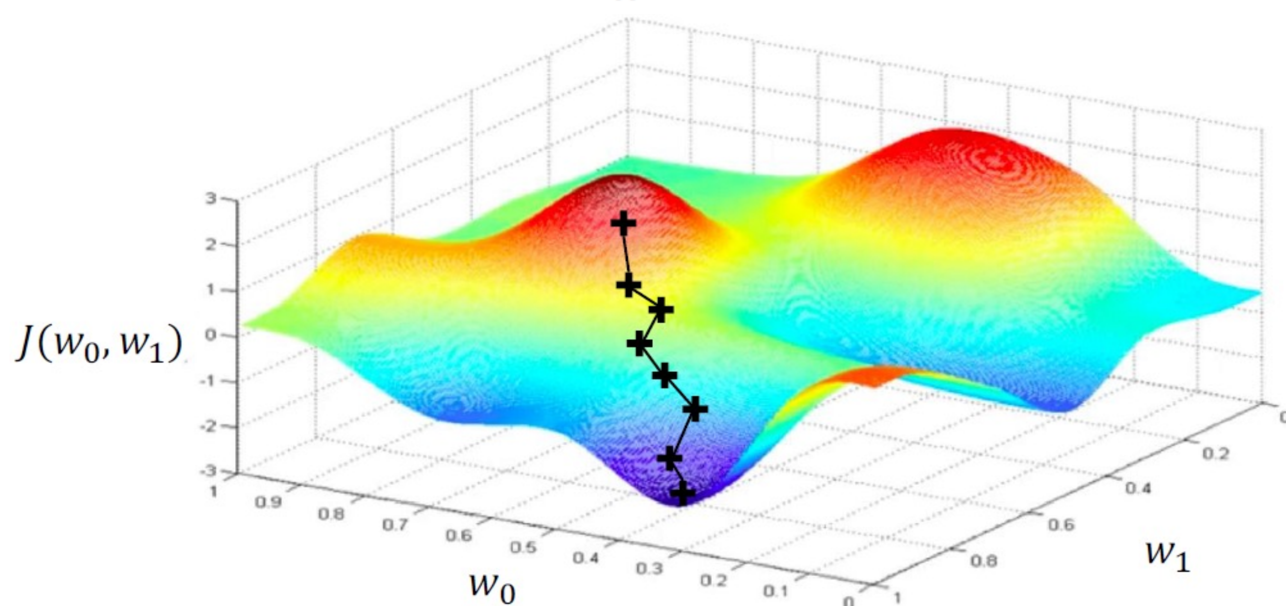


Training a Neural Network = Minimizing a Loss

We seek for a set of weights that achieve minimal loss:

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

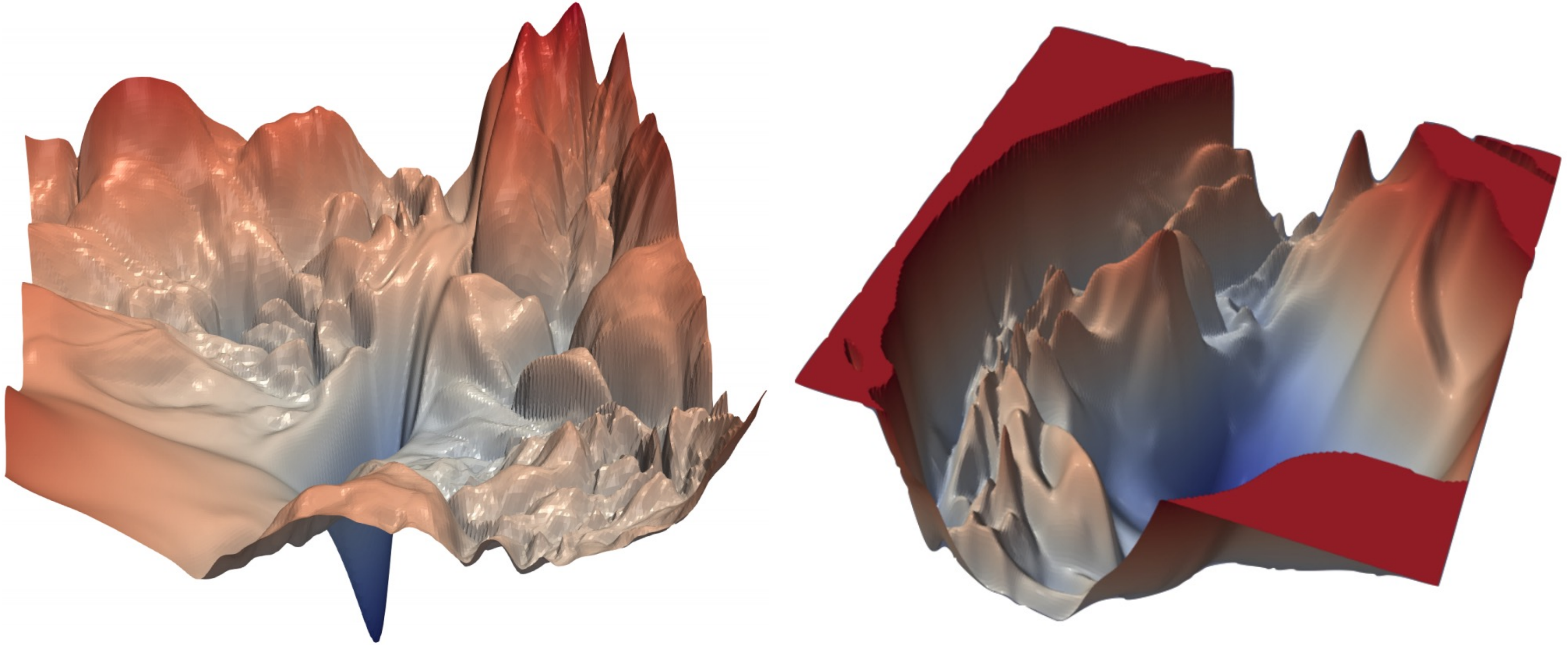


Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

Loss functions may be quite complex...



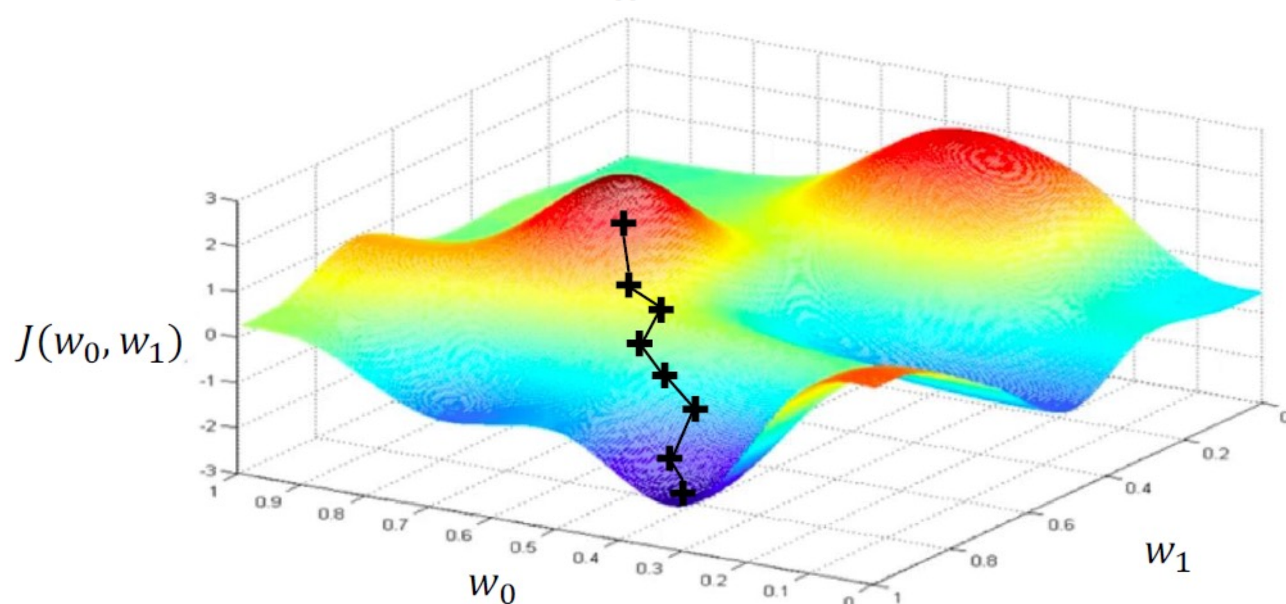
H. Li et al. *Visualizing the Loss Landscape of Neural Nets* NIPS 2018

Training a Neural Network = Minimizing a Loss

We seek for a set of weights that achieve minimal loss:

$$W^* = \underset{W}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; W\right), y^{(i)}\right)$$

$$W^* = \underset{W}{\operatorname{argmin}}\, J(W)$$



Algorithm (Gradient Descent)

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

Learning Rate

To cope with complex loss functions and to optimize training the following algorithm is generally sophisticated

Algorithm (Gradient Descent)

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

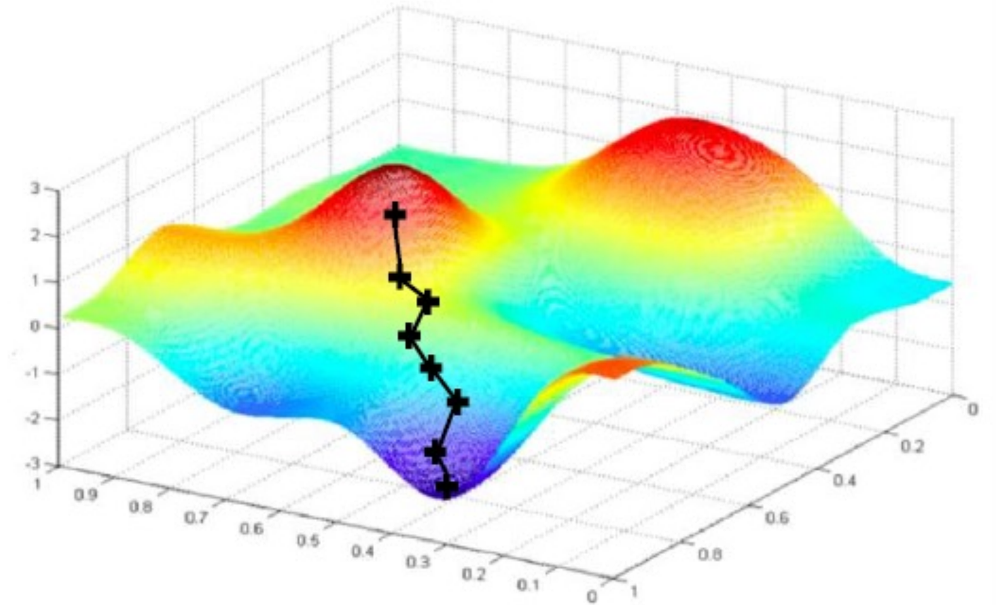
Learning Rate

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Can be very burdensome to compute…
Average over all samples in the dataset!



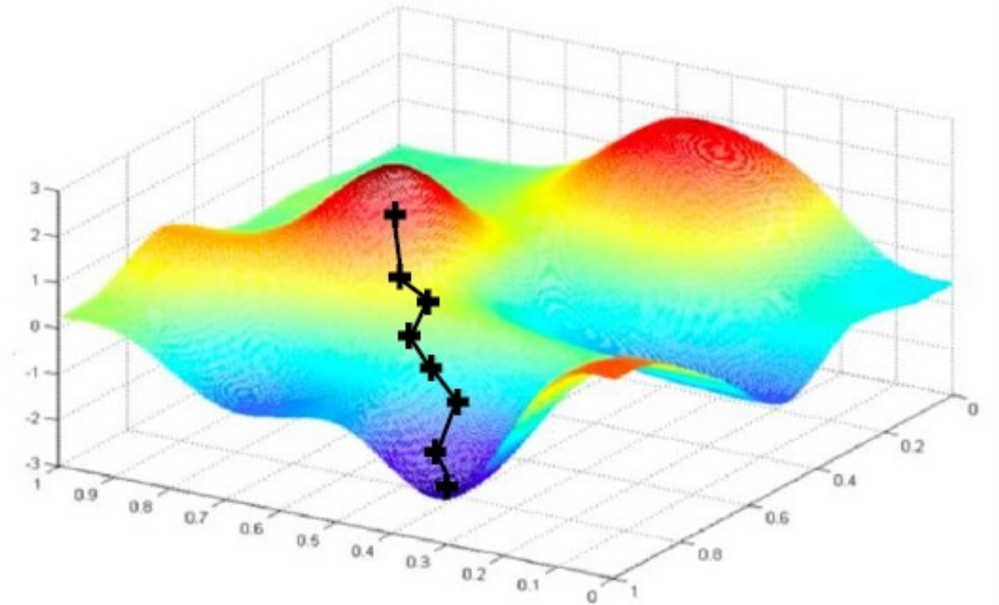
MIT *Introduction to Deep Learning* http://introtodeeplearning.com

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

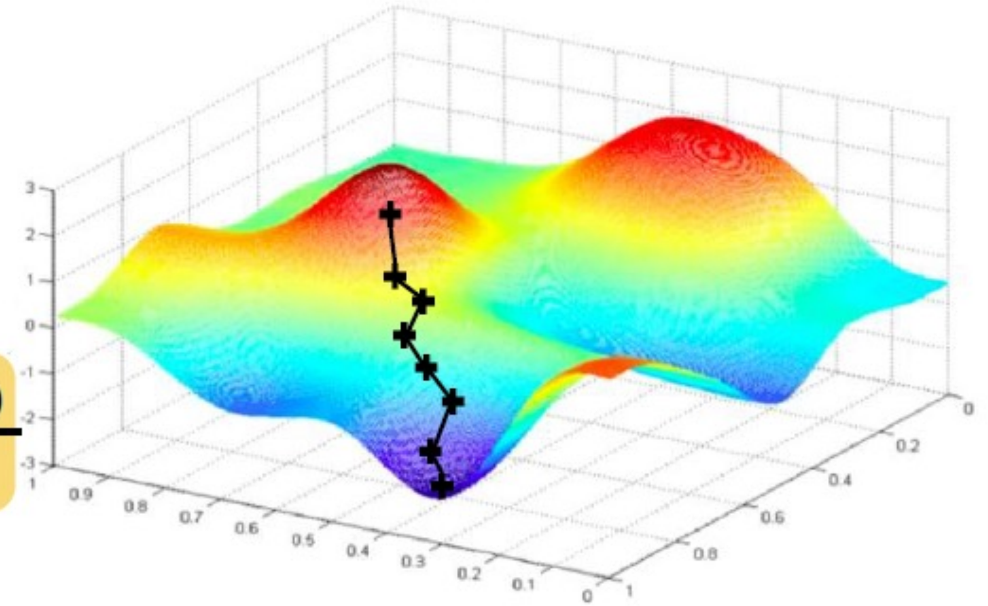
Easy to compute but **very noisy** (stochastic)!



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

Stochastic Gradient Descent: Mini-batches

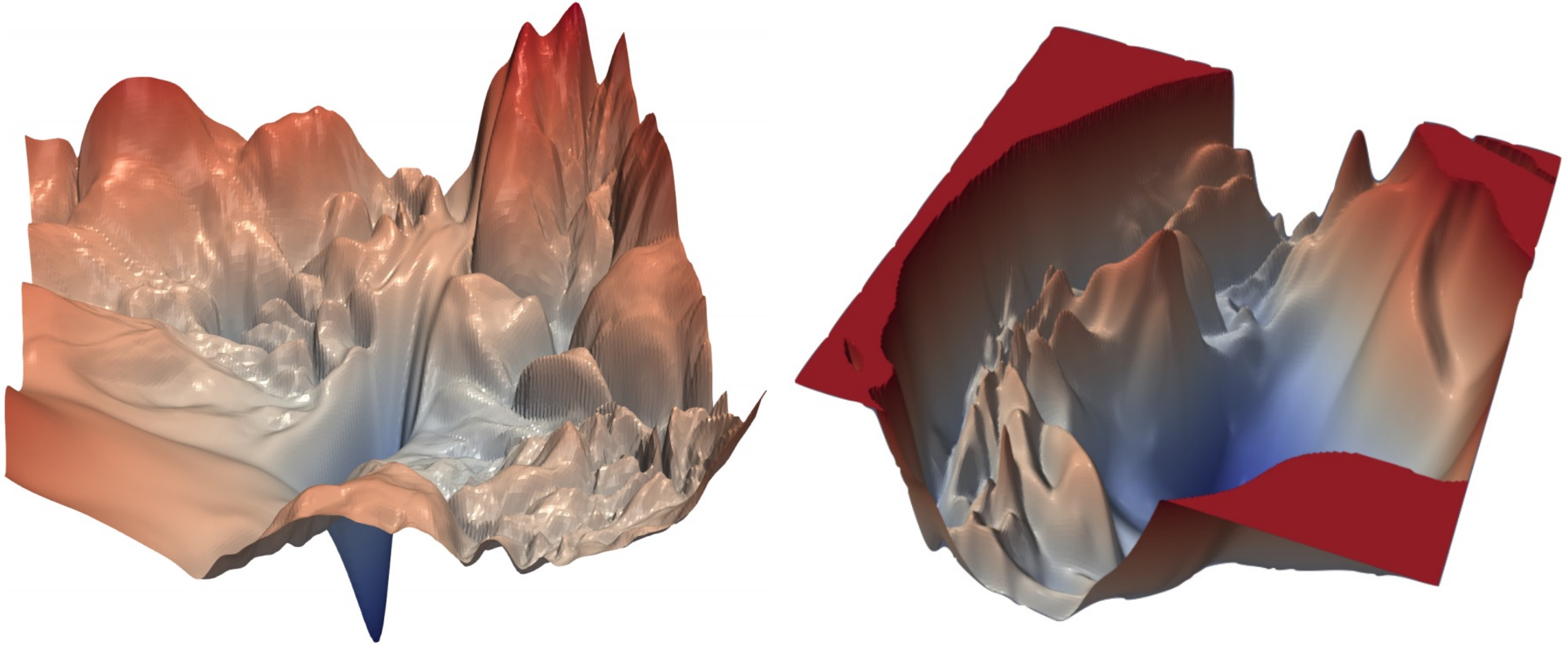
The set of B data points is called *mini-batch*

Mini-batches allow to accurated estimation of gradient, smoother convergence, larger learning rates

Mini-batches lead to fast training: computation can be parallelized and significant speed increases can be obtained on GPUs

In large architectures, smart choices related to ‘simple’ things can have a huge impact

Loss functions may be quite complex...



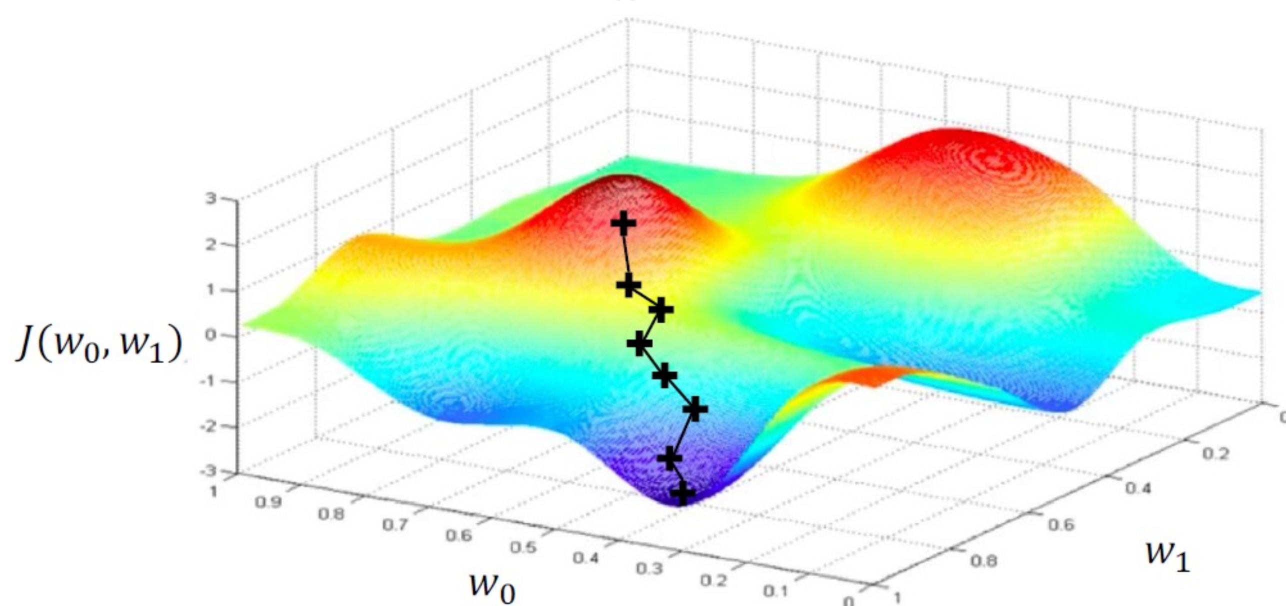
H. Li et al. *Visualizing the Loss Landscape of Neural Nets* NIPS 2018

Training a Neural Network = Minimizing a Loss

We seek for a set of weights that achieve minimal loss:

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$



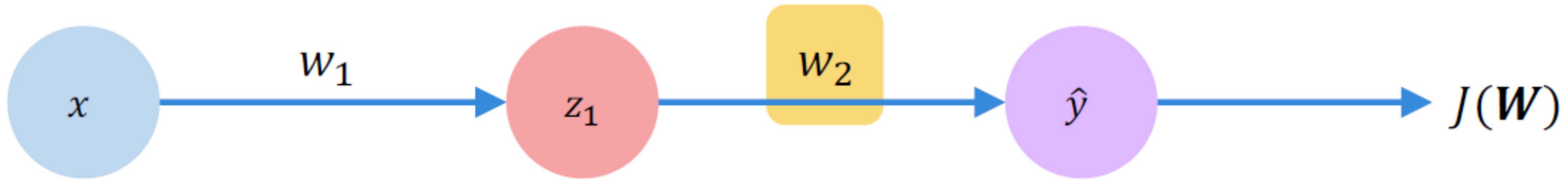
Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

How to compute the gradient: backpropagation

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function.

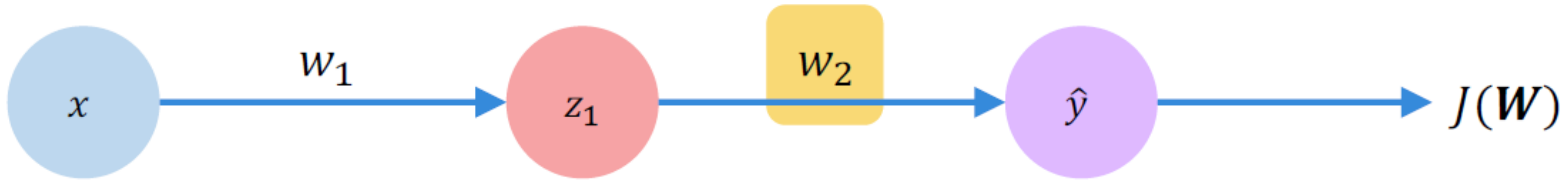


Let's consider a simple NN with one node: how the final loss is affected by changes in w_2?

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

How to compute the gradient: backpropagation

Let's consider a simple NN with one node: how the final loss is affected by changes in w_2

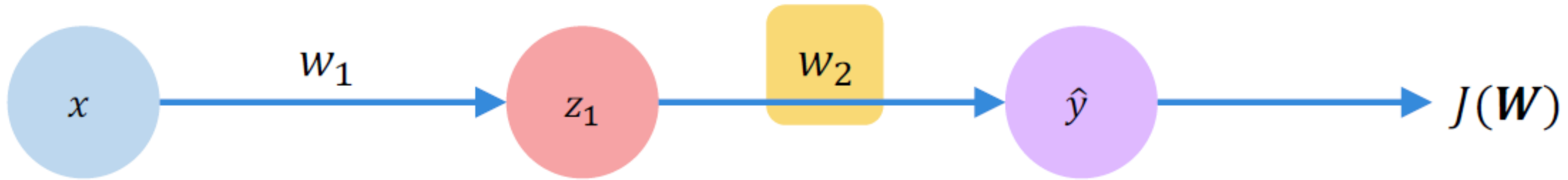


$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = ?$$

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

How to compute the gradient: backpropagation

Let's consider a simple NN with one node: how the final loss is affected by changes in w_2



We can apply the chain rule:

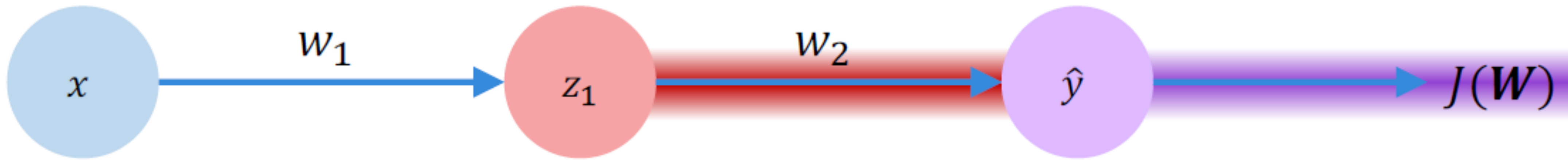
$$y = f(u)$$
$$u = g(x)$$

$$\frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = ?$$

How to compute the gradient: backpropagation

Let's consider a simple NN with one node: how the final loss is affected by changes in w_2



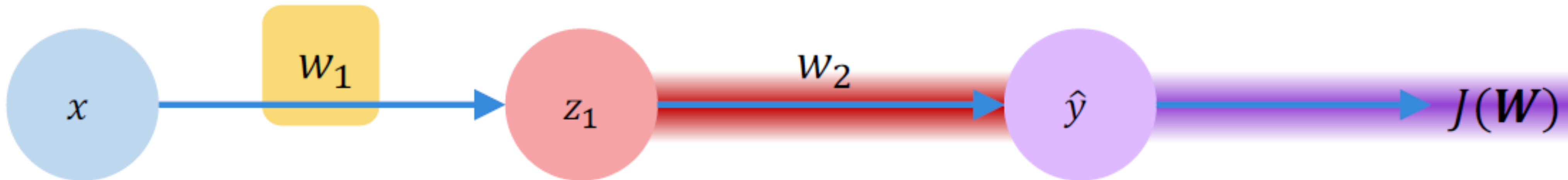
We can apply the chain rule:

$$y = f(u) \qquad u = g(x) \quad \Rightarrow \quad \frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

How to compute the gradient: backpropagation

This simple network is characterized also by the weigth w_1



We can apply the chain rule:

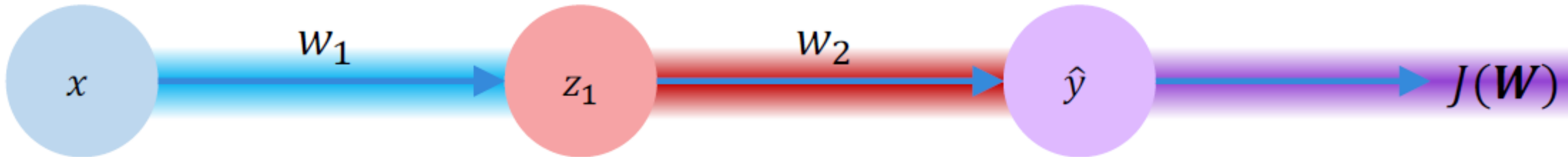
$$y = f(u) \quad u = g(x) \rightarrow \frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule! Apply chain rule!

How to compute the gradient: backpropagation

This simple network is characterized also by the weigth w_1



We can apply the chain rule:

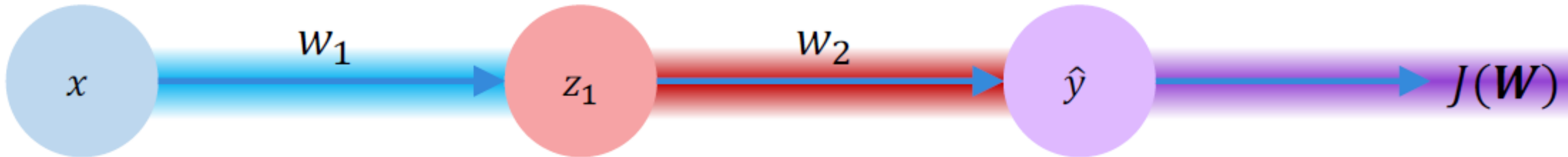
$y = f(u)$
$u = g(x)$

$$\frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

How to compute the gradient: backpropagation

This simple network is characterized also by the weigth w_1



We can apply the chain rule:

$y = f(u)$
$u = g(x)$

$$\frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

$$z_1 = \sigma(x \cdot w_1) \quad \text{(activation)}$$

We also have to keep into account the non-linear transformations (activations)! In that case, we need to apply again the chain rule!

How to compute the gradient: backpropagation

We can apply the chain rule:

$$y = f(u) \quad u = g(x) \quad \Rightarrow \quad \frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

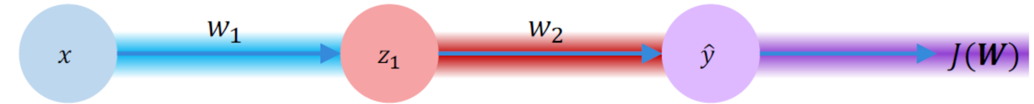
Let's consider an example with sigmoid activation

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

$$z_1 = \sigma(x \cdot w_1) \quad \text{(activation)}$$

$$\hat{y} = z_1 \cdot w_2 \quad \text{(linear output)}$$

$$L = \frac{1}{2}(\hat{y} - y)^2 \quad \text{(loss)}$$



How to compute the gradient: backpropagation

We can apply the chain rule:

$$y = f(u) \qquad u = g(x) \quad \Rightarrow \quad \frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

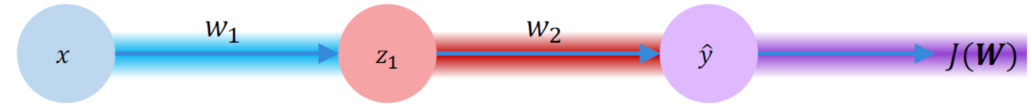
Let's consider an example with sigmoid activation

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

$$z_1 = \sigma(x \cdot w_1) \quad \text{(activation)}$$

$$\hat{y} = z_1 \cdot w_2 \quad \text{(linear output)}$$

$$L = \frac{1}{2}(\hat{y} - y)^2 \quad \text{(loss)}$$



With x=1, w1 = 0.5, w2=-1, y=1:

$$z_1 = \sigma(x \cdot w_1) = \sigma(1.0 \cdot 0.5) = \sigma(0.5) \approx 0.622$$

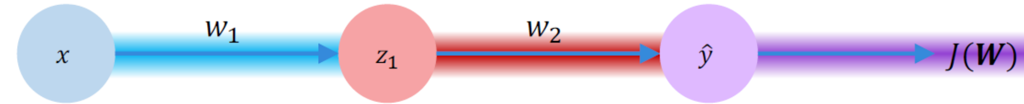
$$\hat{y} = z_1 \cdot w_2 = 0.622 \cdot (-1.0) = -0.622$$

$$L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(-0.622 - 1)^2 = \frac{1}{2}(-1.622)^2 \approx 1.315$$

How to compute the gradient: backpropagation

We can apply the chain rule:

$$y = f(u) \quad u = g(x) \quad \Rightarrow \quad \frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$



$$\frac{dL}{dw_2} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dw_2}$$

$$\frac{dL}{d\hat{y}} = \hat{y} - y = -0.622 - 1 = -1.622$$

$$\frac{d\hat{y}}{dw_2} = z_1 = 0.622$$

$$\frac{dL}{dw_2} = -1.622 \cdot 0.622 \approx -1.009$$

How to compute the gradient: backpropagation

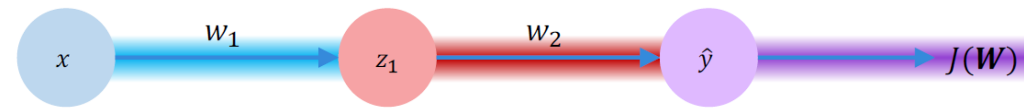
We can apply the chain rule:

$y = f(u)$

$u = g(x)$

$\frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$

$$\frac{dL}{dw_1} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dz_1} \cdot \frac{dz_1}{dw_1}$$



Breaking it down:

- $\frac{dL}{d\hat{y}} = -1.622$
- $\frac{d\hat{y}}{dz_1} = w_2 = -1.0$
- $\frac{dz_1}{dw_1} = \sigma'(x \cdot w_1) \cdot x = z_1(1 - z_1) \cdot x$

Using:

- $z_1 = 0.622$
- $\sigma'(z) = \sigma(z)(1 - \sigma(z)) \approx 0.622 \cdot (1 - 0.622) \approx 0.235$

Then:

$$\frac{dz_1}{dw_1} = 0.235 \cdot 1.0 = 0.235$$

$$\frac{dL}{dw_1} = (-1.622) \cdot (-1.0) \cdot 0.235 \approx 0.381$$

$$\frac{dL}{dw_2} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dw_2}$$

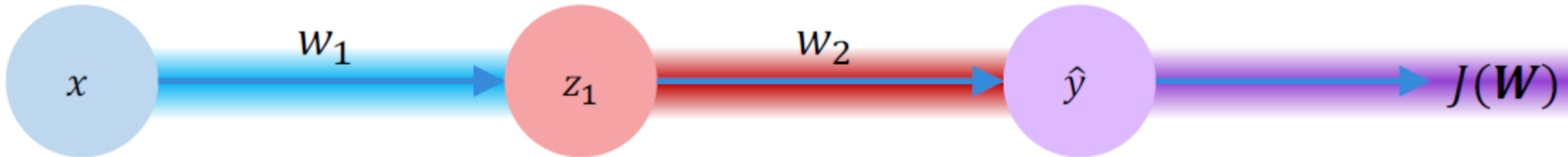
$$\frac{dL}{d\hat{y}} = \hat{y} - y = -0.622 - 1 = -1.622$$

$$\frac{d\hat{y}}{dw_2} = z_1 = 0.622$$

$$\frac{dL}{dw_2} = -1.622 \cdot 0.622 \approx -1.009$$

How to compute the gradient: backpropagation

This simple network is characterized also by the weigth w_1



We can apply the chain rule:

$$y = f(u), \quad u = g(x) \quad \rightarrow \quad \frac{dy}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

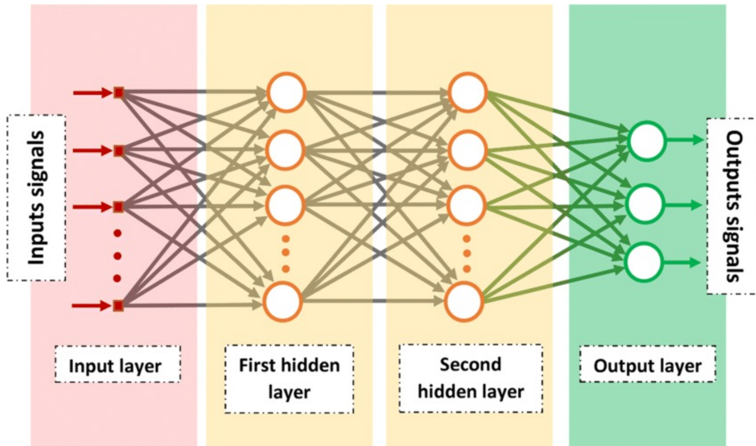
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

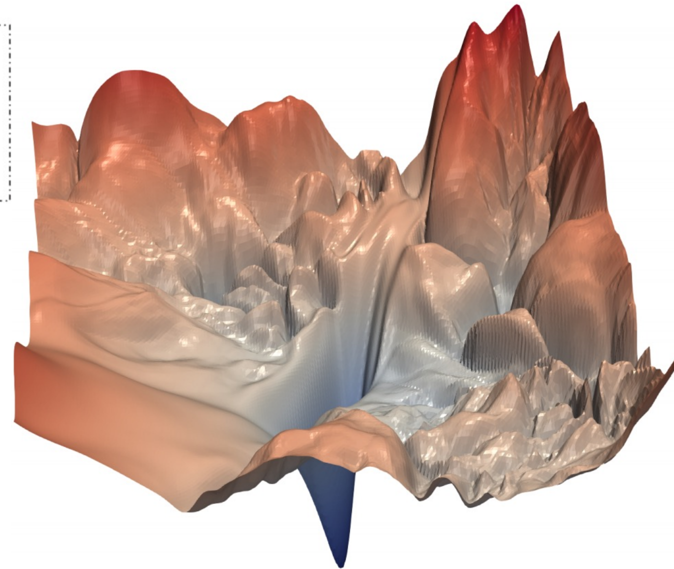
You can have a look here for the math: http://neuralnetworksanddeeplearning.com/chap2.html

MIT *Introduction to Deep Learning* http://introtodeeplearning.com

When dealing with NN training, we are dealing with a ‘beast’

Stochastic Gradient Descent and Back-propagation have been there for decades, but we weren’t able to properly train such architectures!

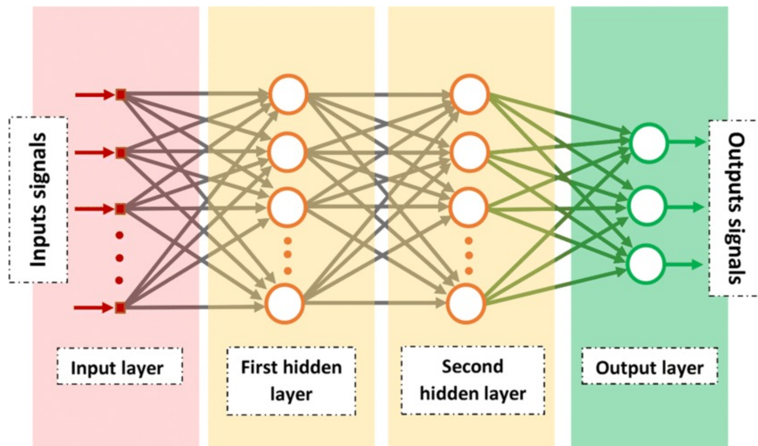


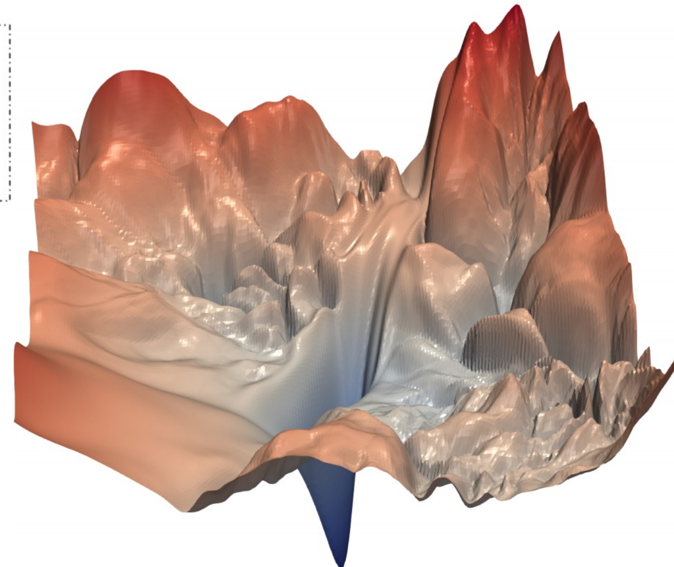


When dealing with NN training, we are dealing with a ‘beast’

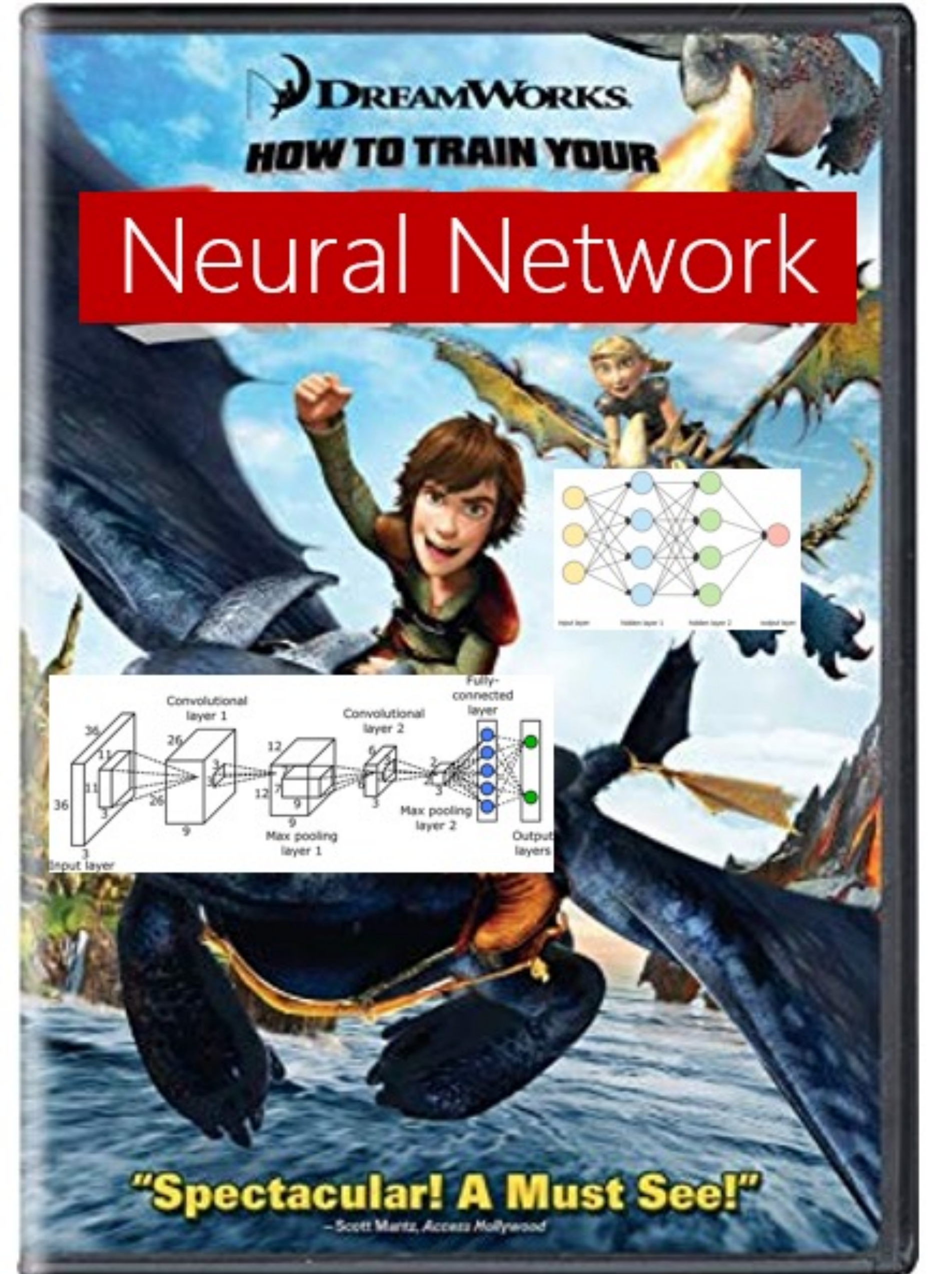
Stochastic Gradient Descent and Back-propagation have been there for decades, but we weren’t able to properly train such architectures!

In past recent years we have developed many tricks to ‘tame the beast’

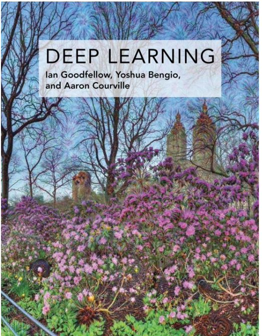




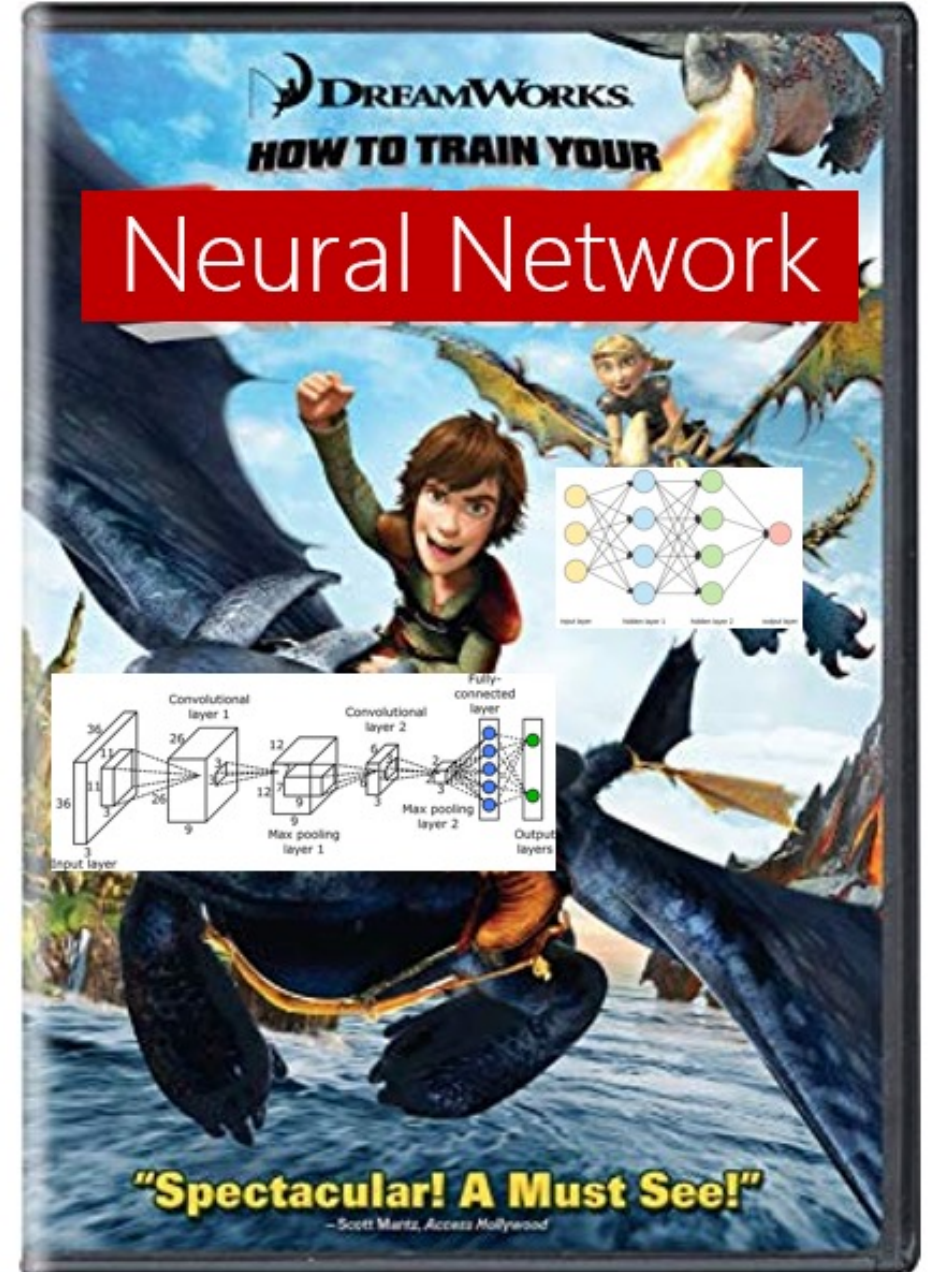
1. Activation functions
2. Weight Initialization
3. Batch Normalization
4. Optimization
5. Learning Rate
6. Regularization



1. Activation functions
2. Weight Initialization
3. Batch Normalization
4. Optimization
5. Learning Rate
6. Regularization



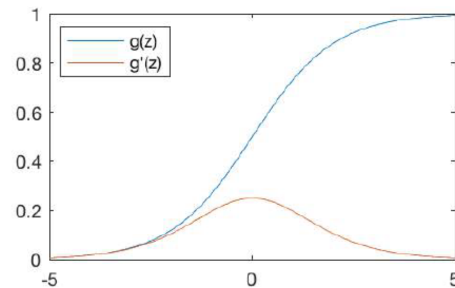
Chapter 6.3 Hidden Units



Activation functions

Beside introducing non-linearities, it is important for Activation functions to have an easy way to compute the gradients (we need this in backpropagation)

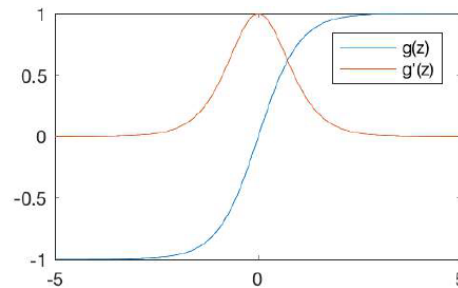
Sigmoid Function



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

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

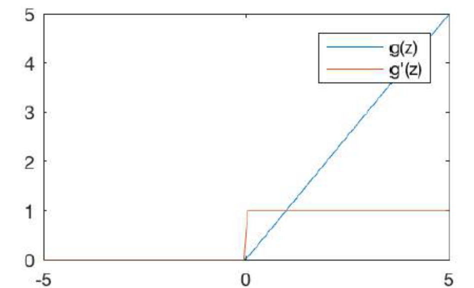
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

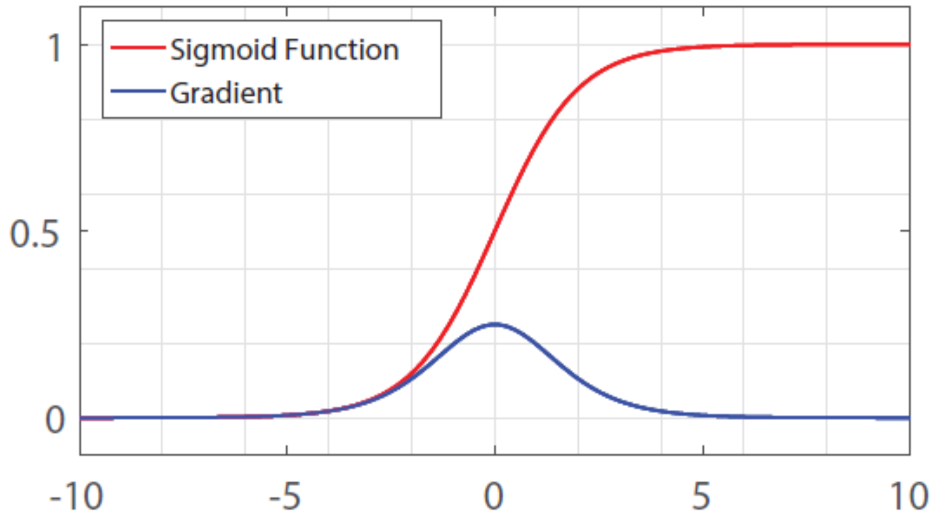
Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Sigmoid



G. Roffo *Ranking to Learn and Learning to Rank: On the Role of Ranking in Pattern Recognition Applications*

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

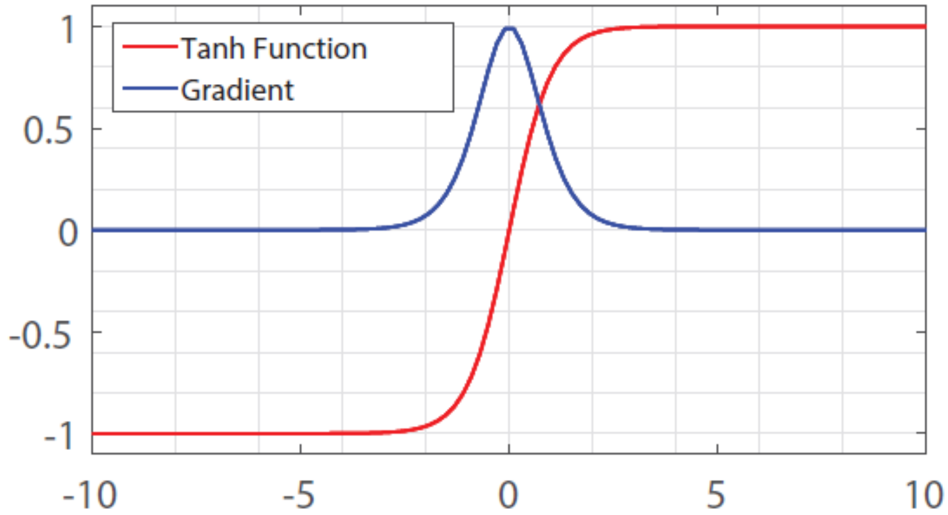
$$g'(z) = g(z)(1 - g(z))$$

Simple interpretation (probability)

Issues:

- Vanishing gradient problem: gradient becomes increasily small as the absolute value of the input increases (it is a problem in backpropagation)
- No zero-centered output (zig-zagging dynamics in the gradient updates)
- Exp function can be expensive to compute

Tanh



G. Roffo *Ranking to Learn and Learning to Rank: On the Role of Ranking in Pattern Recognition Applications*

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

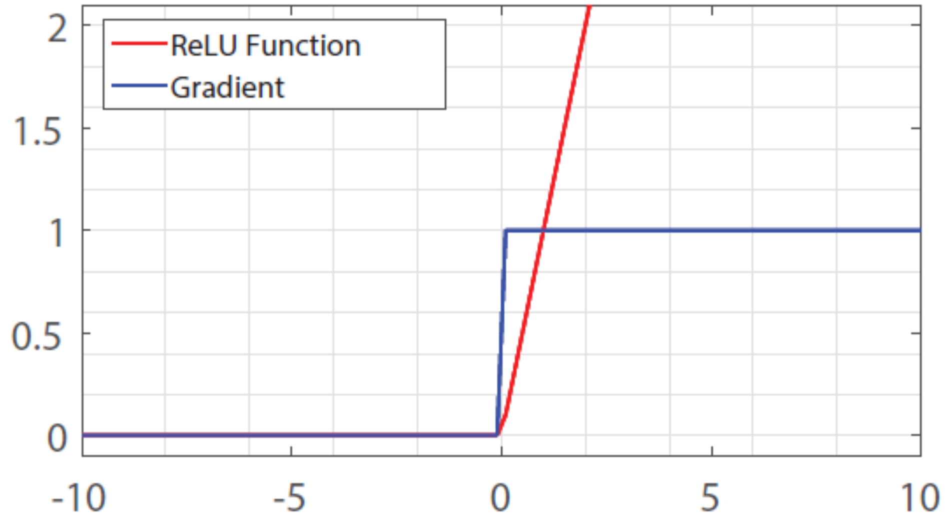
$$g'(z) = 1 - g(z)^2$$

Zero-centered

Issues:

- Vanishing gradient problem
- Exp function can be expensive to compute

ReLU



G. Roffo *Ranking to Learn and Learning to Rank: On the Role of Ranking in Pattern Recognition Applications*

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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Simple implementation

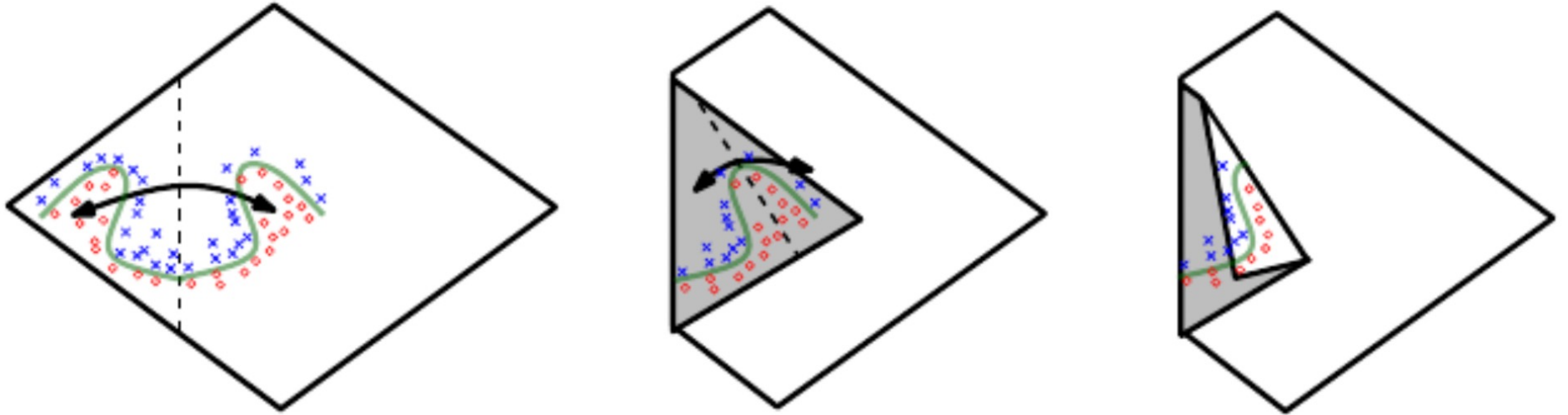
Does not saturate (in + region we have no vanishing gradient)

Faster convergence of SGD than sigmoid/tanh

Issues:

- Not zero-centered
- Gradient is zero for negative values: dead ReLU (as much as 40% of the network never activate if the learning rate is too high)

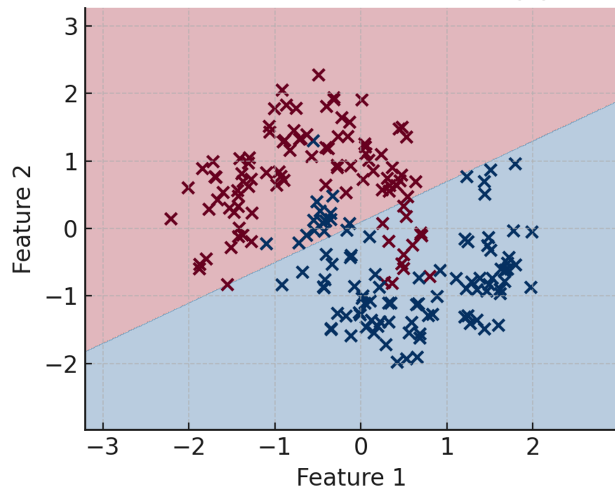
ReLU: Intuition

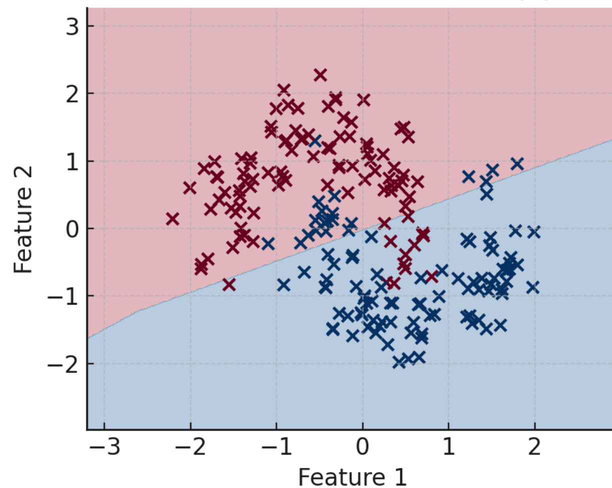


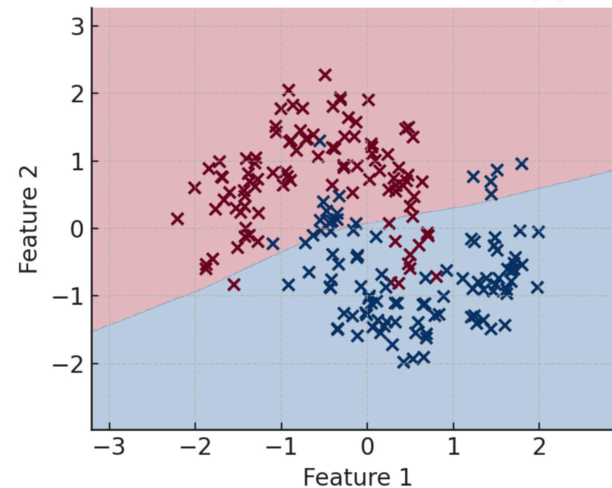
ReLU gives neural networks the power to ‘fold’ the input space.

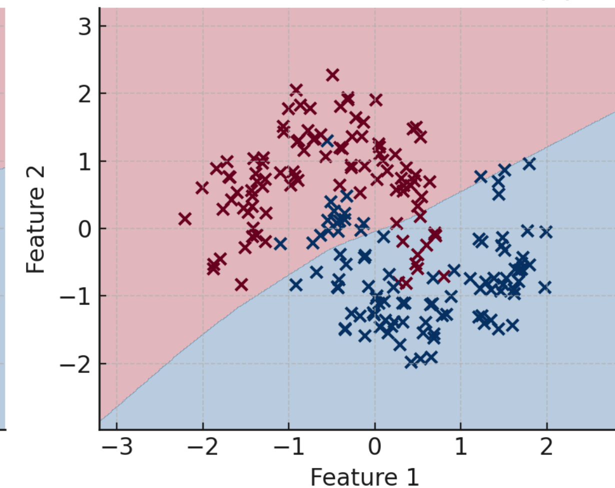
This lets them transform complex, tangled data into something that even a straight line can separate.

1 hidden layer

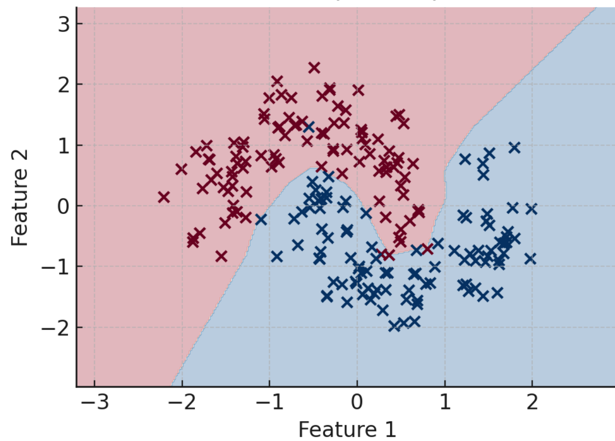


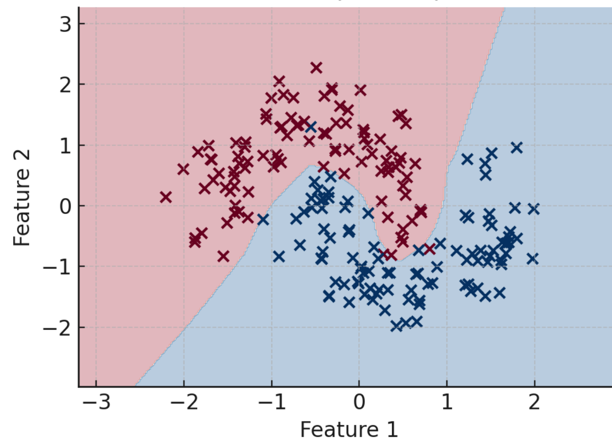


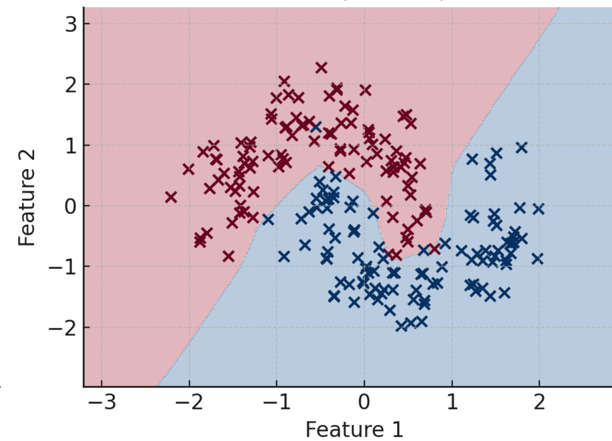


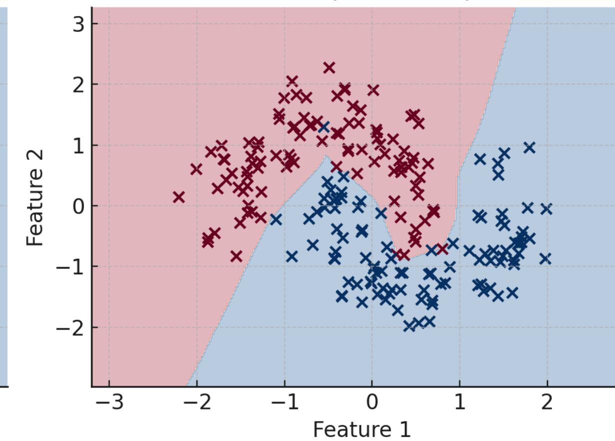


2 hidden layers

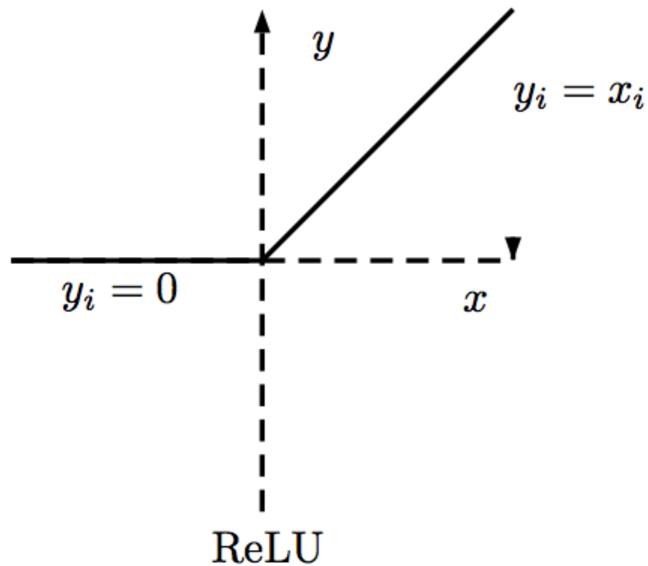


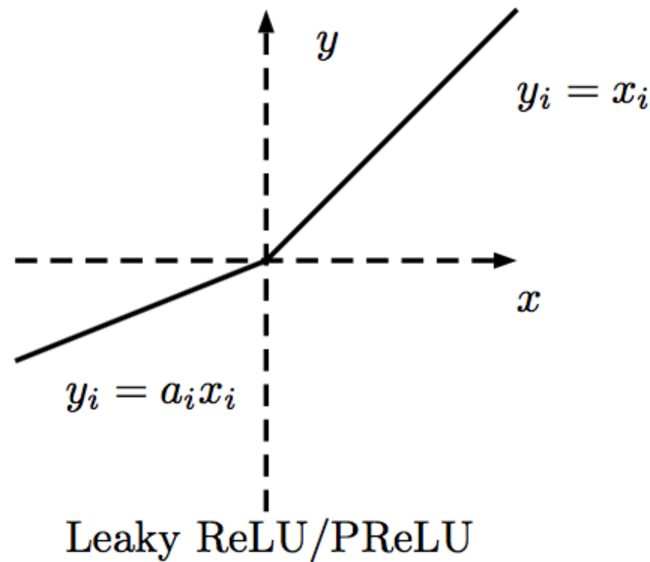






Leaky ReLU, Parametric ReLU, Randomized Leaky ReLU





$$g(z) = 0.01z, (z < 0)$$
$$g(z) = z, (z \geq 0)$$

$$g(z) = \alpha z, (z < 0)$$
$$g(z) = z, (z \geq 0)$$

Very easy to compute

Does not saturate (in + region)

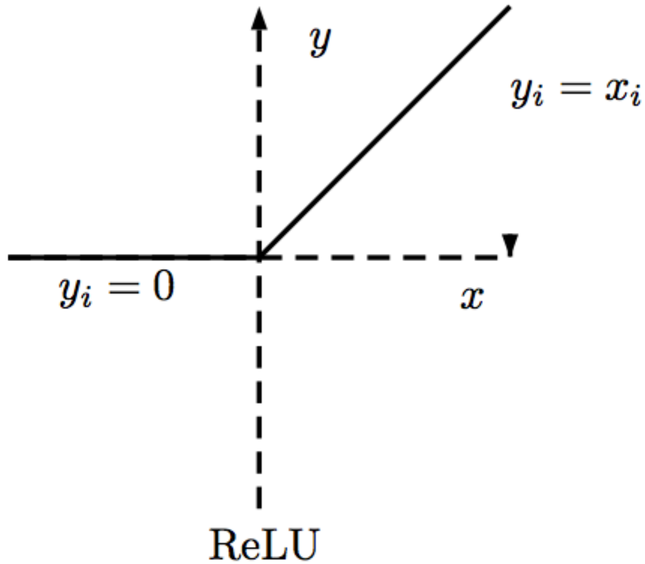
Faster convergence than sigmoid/tanh

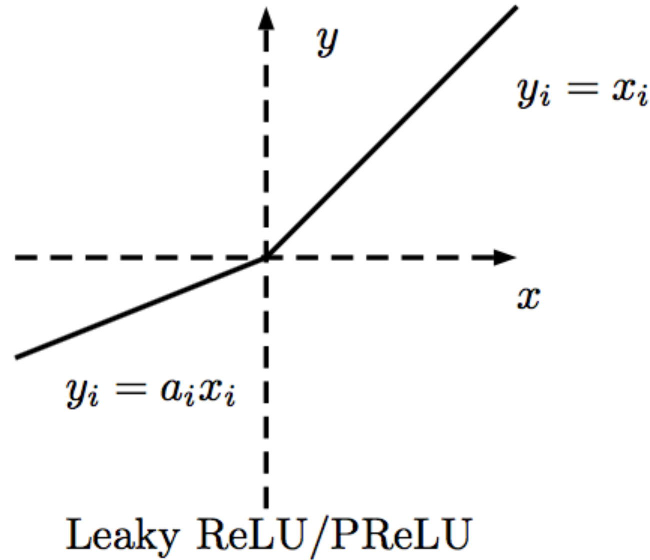
Does not die!

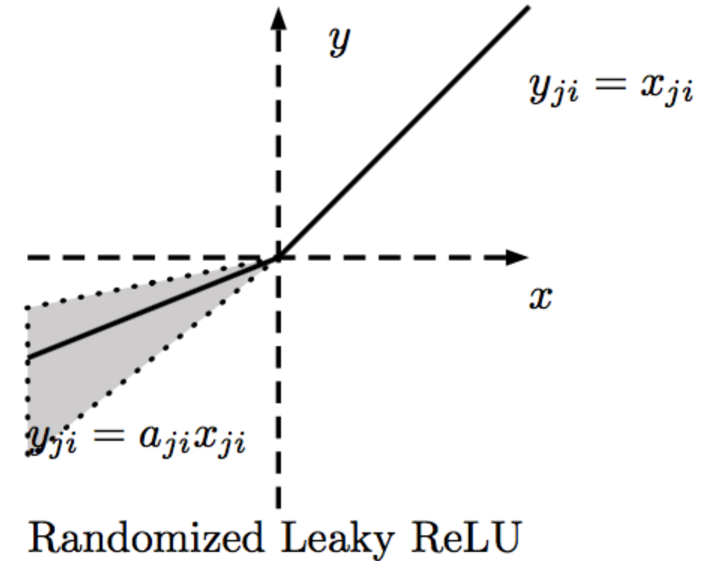
In the Parametric ReLU (PReLU) the parameter alpha is learned along with the other network parameters

https://isaacchanghau.github.io/post/activation_functions/

Leaky ReLU, Parametric ReLU, Randomized Leaky ReLU







$$g(z) = 0.01z, (z < 0)$$
$$g(z) = z, (z \geq 0)$$

$$g(z) = \alpha z, (z < 0)$$
$$g(z) = z, (z \geq 0)$$

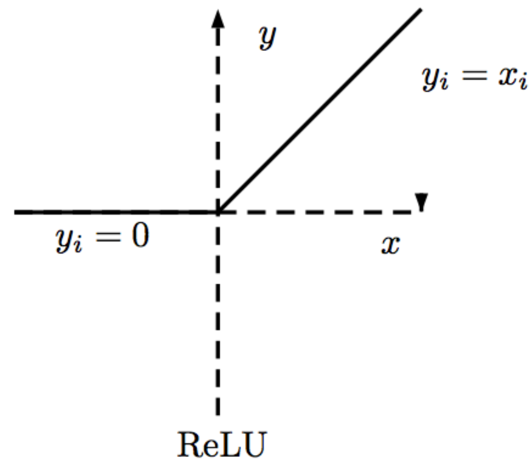
Alpha is chosen at random in a given range in training and it is then fixed in test

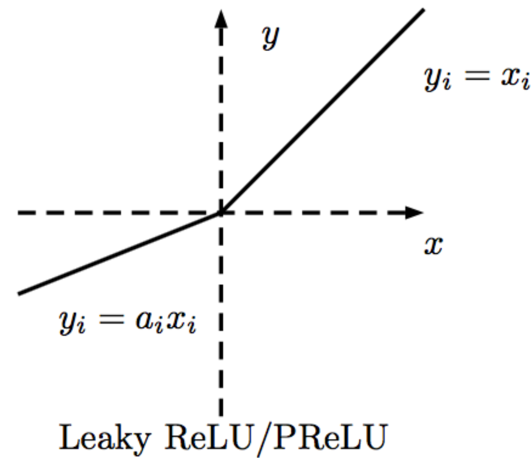
https://isaacchanghau.github.io/post/activation_functions/

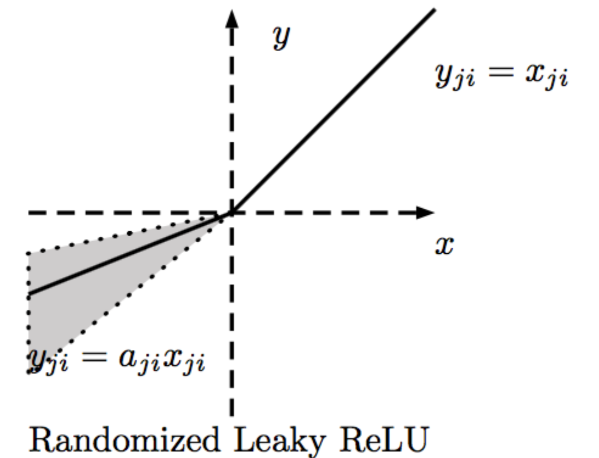
TLDR: Activations

Main issues with activation functions:

- Vanishing gradients
- Non-centered on zero outputs
- Costly computations







https://isaacchanghau.github.io/post/activation_functions/

Name	Plot	Equation	Derivative (with respect to x)	Range	Order of continuity	Monotonic	Monotonic derivative	Approximates identity near the origin
Identity		$f(x)=x$	$f'(x)=1$	$(-\infty,\infty)$	C^∞	Yes	Yes	Yes
Binary step		$f(x)=\begin{cases}0 & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for } x\neq 0\\ ? & \text{for } x=0\end{cases}$	$\{0,1\}$	C^{-1}	Yes	No	No
Logistic (a.k.a. Sigmoid or Soft step)		$f(x)=\sigma(x)=\frac{1}{1+e^{-x}}$ [1]	$f'(x)=f(x)(1-f(x))$	$(0,1)$	C^∞	Yes	No	No
TanH		$f(x)=\tanh(x)=\frac{(e^x-e^{-x})}{(e^x+e^{-x})}$	$f'(x)=1-f(x)^2$	$(-1,1)$	C^∞	Yes	No	Yes
ArcTan		$f(x)=\tan^{-1}(x)$	$f'(x)=\frac{1}{x^2+1}$	$\left(-\frac{\pi}{2},\frac{\pi}{2}\right)$	C^∞	Yes	No	Yes
ArSinH		$f(x)=\sinh^{-1}(x)=\ln\left(x+\sqrt{x^2+1}\right)$	$f'(x)=\frac{1}{\sqrt{x^2+1}}$	$(-\infty,\infty)$	C^∞	Yes	No	Yes
ElliotSig[8][9][10] Softsign[11][12]		$f(x)=\frac{x}{1+\lvert x\rvert}$	$f'(x)=\frac{1}{(1+\lvert x\rvert)^2}$	$(-1,1)$	C^1	Yes	No	Yes
Inverse square root unit (ISRU)[13]		$f(x)=\frac{x}{\sqrt{1+\alpha x^2}}$	$f'(x)=\left(\frac{1}{\sqrt{1+\alpha x^2}}\right)^3$	$\left(-\frac{1}{\sqrt{\alpha}},\frac{1}{\sqrt{\alpha}}\right)$	C^∞	Yes	No	Yes
Inverse square root linear unit (ISRLU)[13]		$f(x)=\begin{cases}\frac{x}{\sqrt{1+\alpha x^2}} & \text{for } x<0\\ x & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}\left(\frac{1}{\sqrt{1+\alpha x^2}}\right)^3 & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$\left(-\frac{1}{\sqrt{\alpha}},\infty\right)$	C^2	Yes	Yes	Yes
Square Nonlinearity (SQNL)[10]		$f(x)=\begin{cases}1 & : x>2.0\\ x-\frac{x^2}{4} & : 0\leq x\leq 2.0\\ x+\frac{x^2}{4} & : -2.0\leq x<0\\ -1 & : x<-2.0\end{cases}$	$f'(x)=1\mp\frac{x}{2}$	$(-1,1)$	C^2	Yes	No	Yes
Rectified linear unit (ReLU)[14]		$f(x)=\begin{cases}0 & \text{for } x<0\\ x & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$[0,\infty)$	C^0	Yes	Yes	No
Bipolar rectified linear unit (BReLU)[15]		$f(x_i)=\begin{cases}ReLU(x_i) & \text{if } i \bmod 2=0\\ -ReLU(-x_i) & \text{if } i \bmod 2\neq 0\end{cases}$	$f'(x_i)=\begin{cases}ReLU'(x_i) & \text{if } i \bmod 2=0\\ -ReLU'(-x_i) & \text{if } i \bmod 2\neq 0\end{cases}$	$(-\infty,\infty)$	C^0	Yes	Yes	No
Leaky rectified linear unit (Leaky ReLU)[16]		$f(x)=\begin{cases}0.01x & \text{for } x<0\\ x & \text{for } x\geq 0\end{cases}$	$f'(x)=\begin{cases}0.01 & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$(-\infty,\infty)$	C^0	Yes	Yes	No
Parameteric rectified linear unit (PReLU)[17]		$f(\alpha,x)=\begin{cases}\alpha x & \text{for } x<0\\ x & \text{for } x\geq 0\end{cases}$	$f'(\alpha,x)=\begin{cases}\alpha & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$(-\infty,\infty)$[2]	C^0	Yes iff $\alpha\geq 0$	Yes	Yes iff $\alpha=1$
Randomized leaky rectified linear unit (RReLU)[18]		$f(\alpha,x)=\begin{cases}\alpha x & \text{for } x<0\\ x & \text{for } x\geq 0\end{cases}$ [3]	$f'(\alpha,x)=\begin{cases}\alpha & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$(-\infty,\infty)$	C^0	Yes	Yes	No
Exponential linear unit (ELU)[19]		$f(\alpha,x)=\begin{cases}\alpha(e^x-1) & \text{for } x\leq 0\\ x & \text{for } x>0\end{cases}$	$f'(\alpha,x)=\begin{cases}f(\alpha,x)+\alpha & \text{for } x\leq 0\\ 1 & \text{for } x>0\end{cases}$	$(-\alpha,\infty)$	$\begin{cases}C_1 & \text{when } \alpha=1\\ C_0 & \text{otherwise}\end{cases}$	Yes iff $\alpha\geq 0$	Yes iff $0\leq\alpha\leq 1$	Yes iff $\alpha=1$
Scaled exponential linear unit (SELU)[20]		$f(\alpha,x)=\lambda\begin{cases}\alpha(e^x-1) & \text{for } x<0\\ x & \text{for } x\geq 0\end{cases}$ with $\lambda=1.0507$ and $\alpha=1.67326$	$f'(\alpha,x)=\lambda\begin{cases}\alpha(e^x) & \text{for } x<0\\ 1 & \text{for } x\geq 0\end{cases}$	$(-\lambda\alpha,\infty)$	C^0	Yes	No	No
S-shaped rectified linear activation unit (SReLU)[21]		$f_{t_l,a_l,t_r,a_r}(x)=\begin{cases}t_l+a_l(x-t_l) & \text{for } x\leq t_l\\ x & \text{for } t_l<x<t_r\\ t_r+a_r(x-t_r) & \text{for } x\geq t_r\end{cases}$ t_l,a_l,t_r,a_r are parameters.	$f'_{t_l,a_l,t_r,a_r}(x)=\begin{cases}a_l & \text{for } x\leq t_l\\ 1 & \text{for } t_l<x<t_r\\ a_r & \text{for } x\geq t_r\end{cases}$	$(-\infty,\infty)$	C^0	No	No	No
Adaptive piecewise linear (APL)[22]		$f(x)=\max(0,x)+\sum_{s=1}^{S}a_i^s\max(0,-x+b_i^s)$	$f'(x)=H(x)-\sum_{s=1}^{S}a_i^sH(-x+b_i^s)$[4]	$(-\infty,\infty)$	C^0	No	No	No
SoftPlus[23]		$f(x)=\ln(1+e^x)$	$f'(x)=\frac{1}{1+e^{-x}}$	$(0,\infty)$	C^∞	Yes	Yes	No
Bent identity		$f(x)=\frac{\sqrt{x^2+1}-1}{2}+x$	$f'(x)=\frac{x}{2\sqrt{x^2+1}}+1$	$(-\infty,\infty)$	C^∞	Yes	Yes	Yes
Sigmoid Linear Unit (SiLU)[24] (AKA SiL[25] and Swish-1[26])		$f(x)=x\cdot\sigma(x)$[5]	$f'(x)=f(x)+\sigma(x)(1-f(x))$[6]	$[\approx -0.28,\infty)$	C^∞	No	No	Approximates identity/2
SoftExponential[27]		$f(\alpha,x)=\begin{cases}-\frac{\ln(1-\alpha(x+\alpha))}{\alpha} & \text{for } \alpha<0\\ x & \text{for } \alpha=0\\ \frac{e^{\alpha x}-1}{\alpha}+\alpha & \text{for } \alpha>0\end{cases}$	$f'(\alpha,x)=\begin{cases}\frac{1}{1-\alpha(\alpha+x)} & \text{for } \alpha<0\\ e^{\alpha x} & \text{for } \alpha\geq 0\end{cases}$	$(-\infty,\infty)$	C^∞	Yes	Yes	Yes iff $\alpha=0$
Soft Clipping[28]		$f(\alpha,x)=\frac{1}{\alpha}\log\frac{1+e^{\alpha x}}{1+e^{\alpha(x-1)}}$	$f'(\alpha,x)=\frac{1}{2}\sinh\left(\frac{p}{2}\right)\operatorname{sech}\left(\frac{px}{2}\right)\operatorname{sech}\left(\frac{p}{2}(1-x)\right)$	$(0,1)$	C^∞	Yes	No	No
Sinusoid[29]		$f(x)=\sin(x)$	$f'(x)=\cos(x)$	$[-1,1]$	C^∞	No	No	Yes
Sinc		$f(x)=\begin{cases}1 & \text{for } x=0\\ \frac{\sin(x)}{x} & \text{for } x\neq 0\end{cases}$	$f'(x)=\begin{cases}0 & \text{for } x=0\\ \frac{\cos(x)}{x}-\frac{\sin(x)}{x^2} & \text{for } x\neq 0\end{cases}$	$[\approx -.217234,1]$	C^∞	No	No	No
Gaussian		$f(x)=e^{-x^2}$	$f'(x)=-2xe^{-x^2}$	$(0,1]$	C^∞	No	No	No

https://en.wikipedia.org/wiki/Activation_function

Which activation?

In practice:

- prefer ReLU. Use slightly positive initial bias to avoid dead Relu issue.
- Try out Leaky ReLU/PRelu
- No sigmoid!

Thank you!

Gian Antonio Susto