DEEP NEURAL NETWORK

Deep Learning for Computer Vision

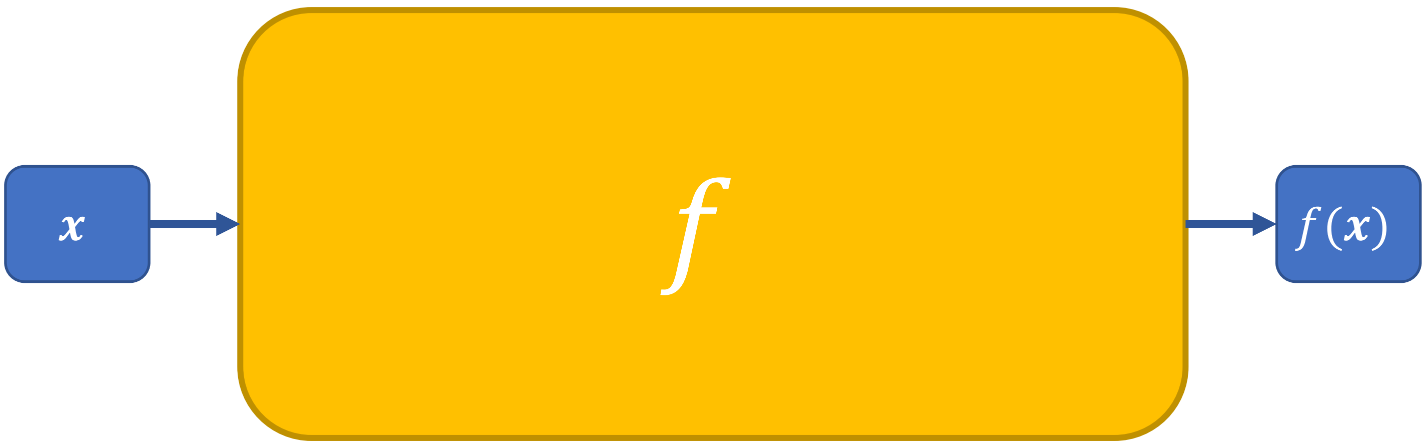
Arthur Douillard

https://arthurdouillard.com/deepcourse

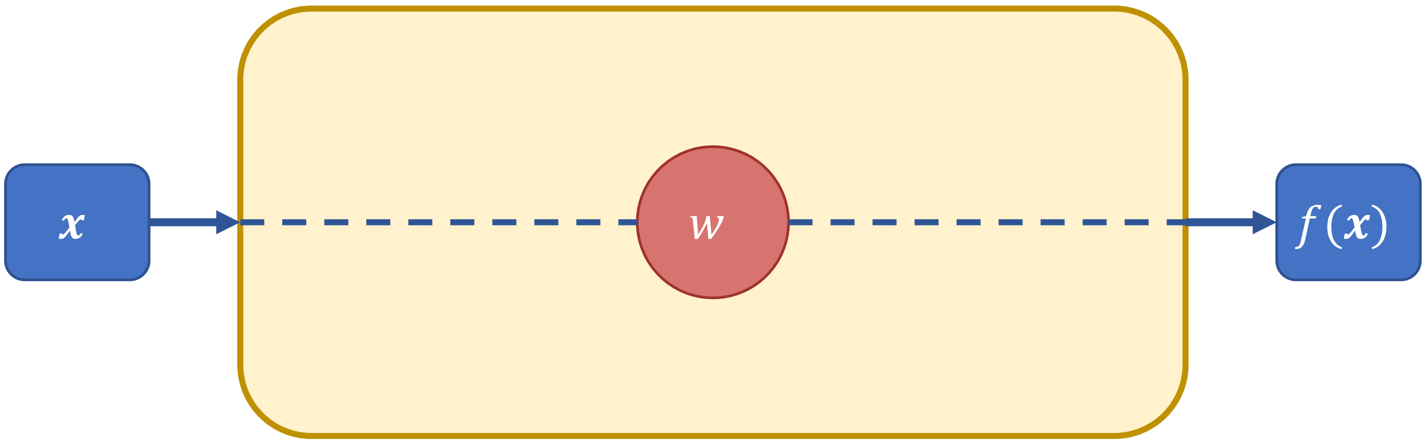
Deep Neural Networks

Function Approximator

A neural network <u>approximates</u> a function

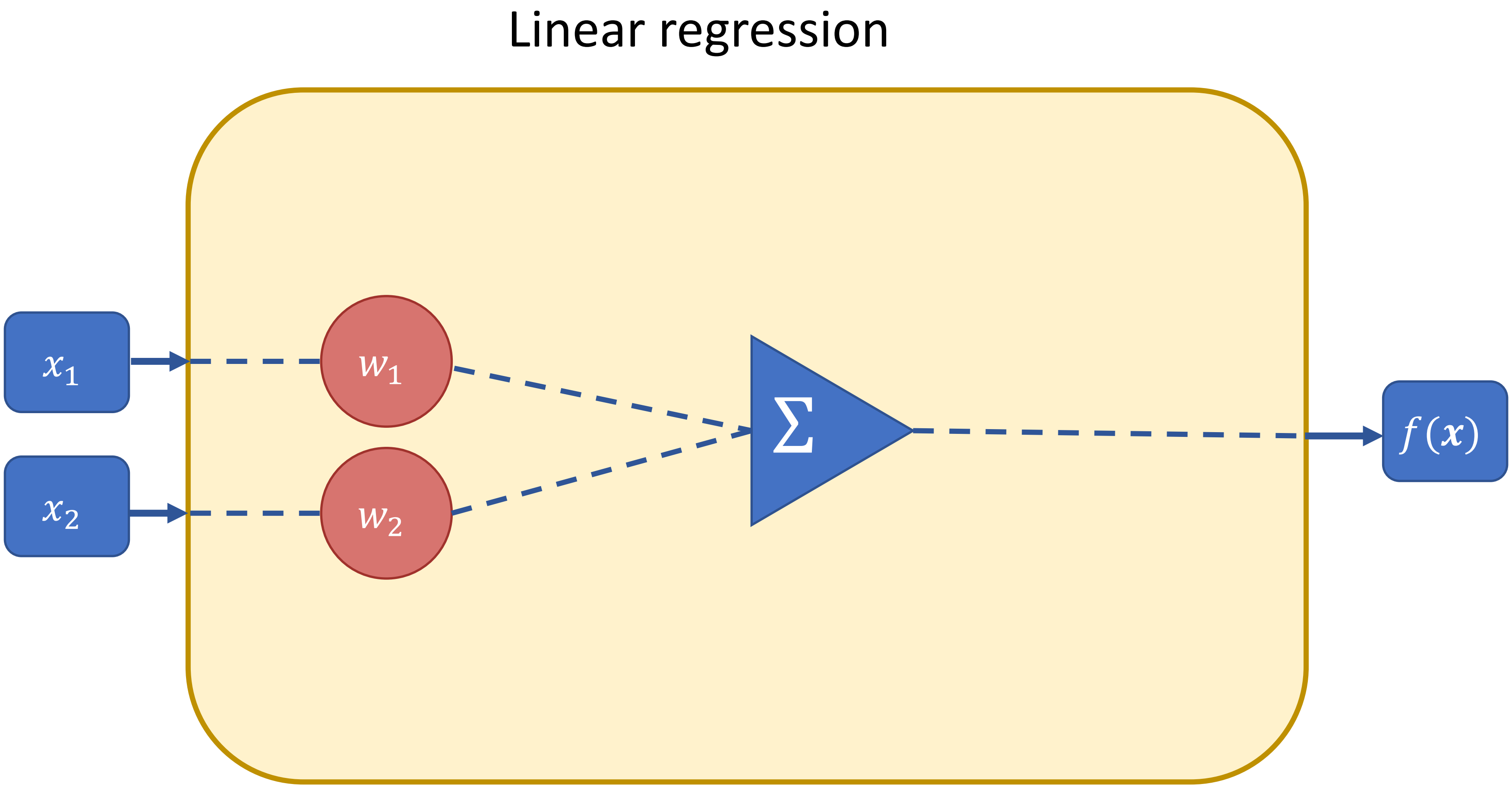


Linear regression



Useful to predict continuous variables

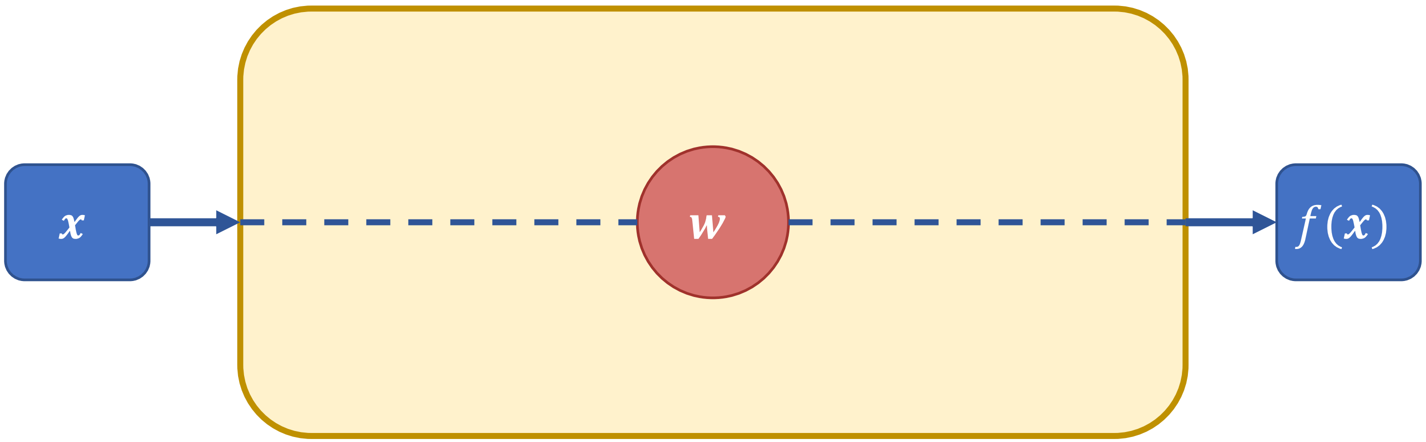
$$f(x) = wx$$



The "connection" often depicted are only multiplications and additions

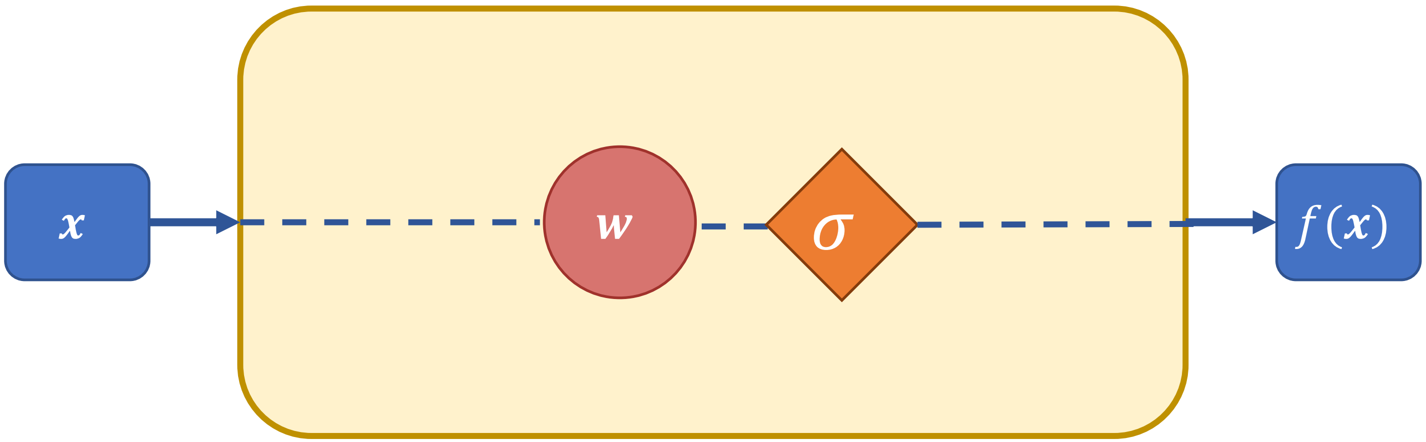
$$f(\boldsymbol{x}) = w_1 x_1 + w_2 x_2$$

Linear regression



$$f(x) = wx$$

Single Neuron

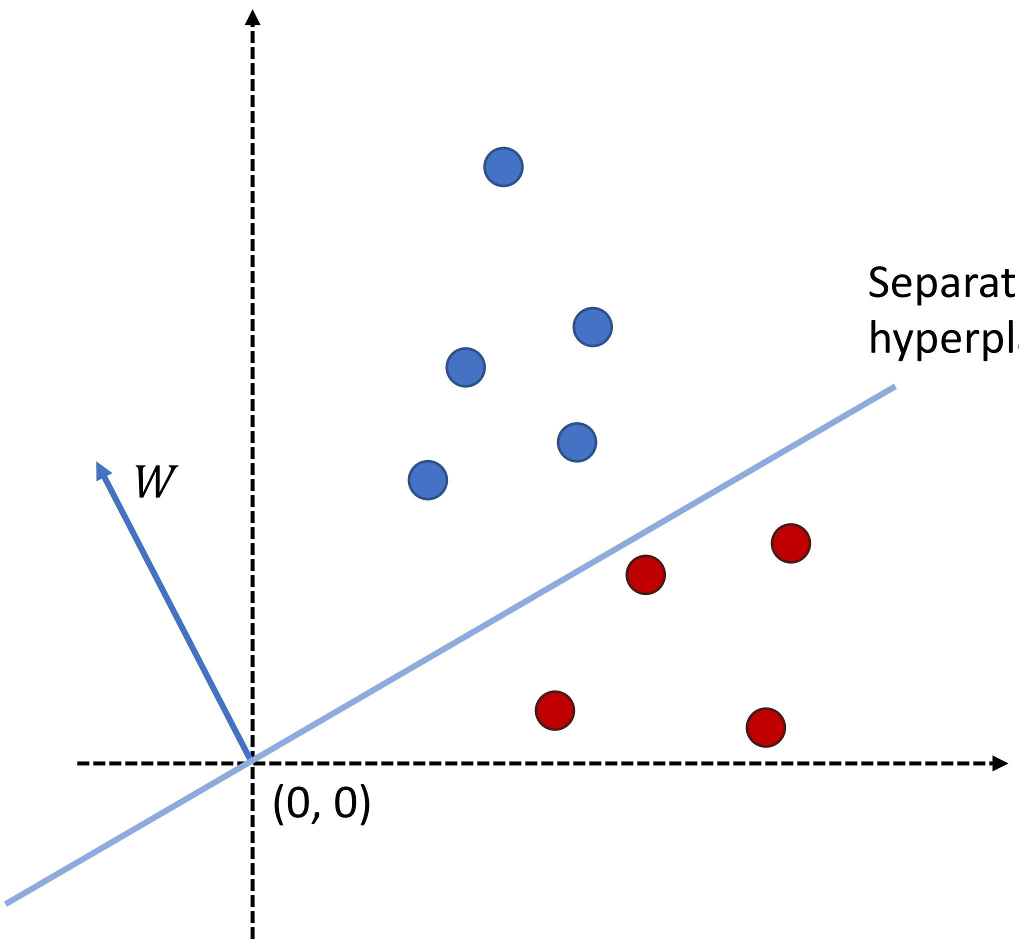


Non-linear activation to determine how much a neuron “fires”

$$f(x) = \boldsymbol{\sigma}(wx)$$

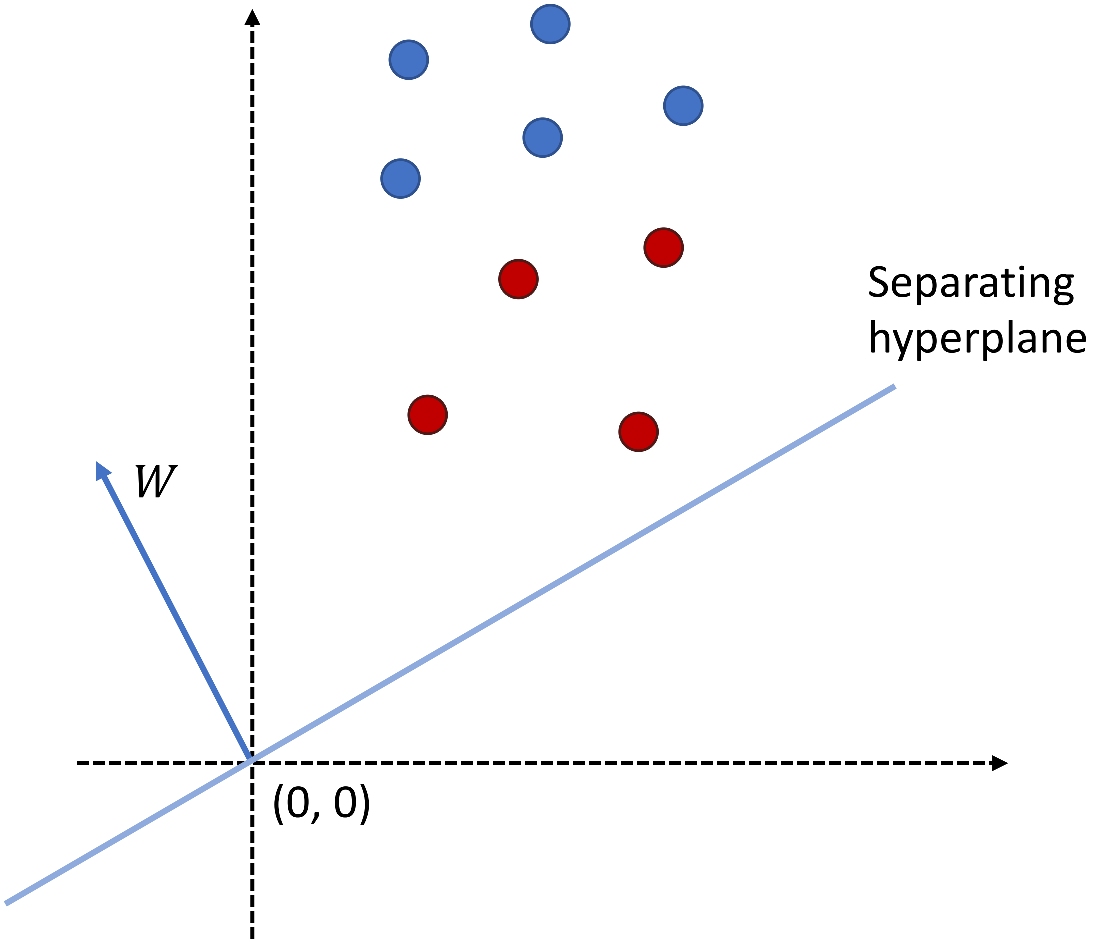
Separating Hyperplane

Optimize so that W
is orthogonal to
the separating hyperplane

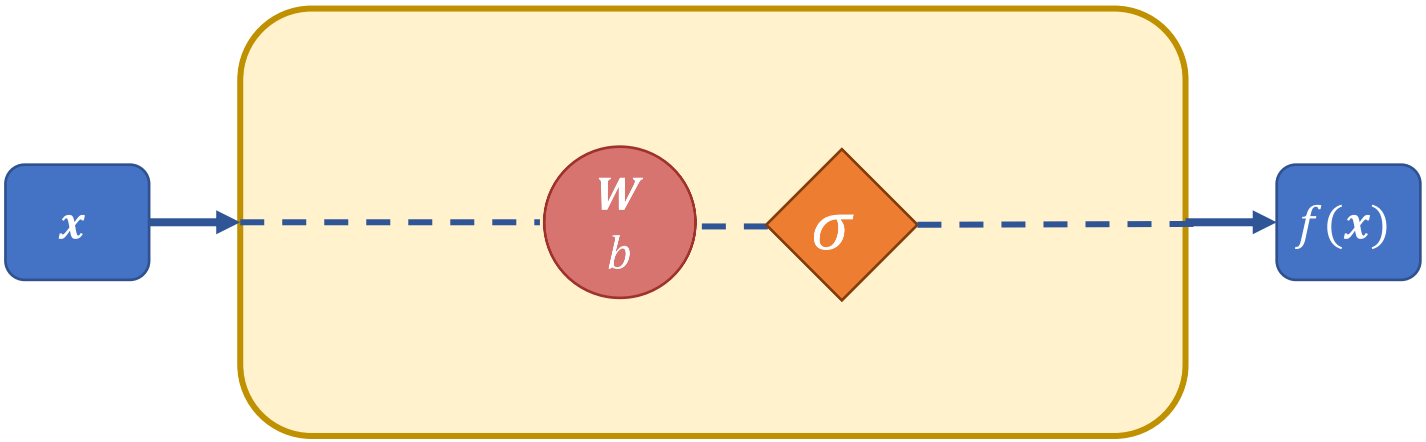


Without bias

Need bias!



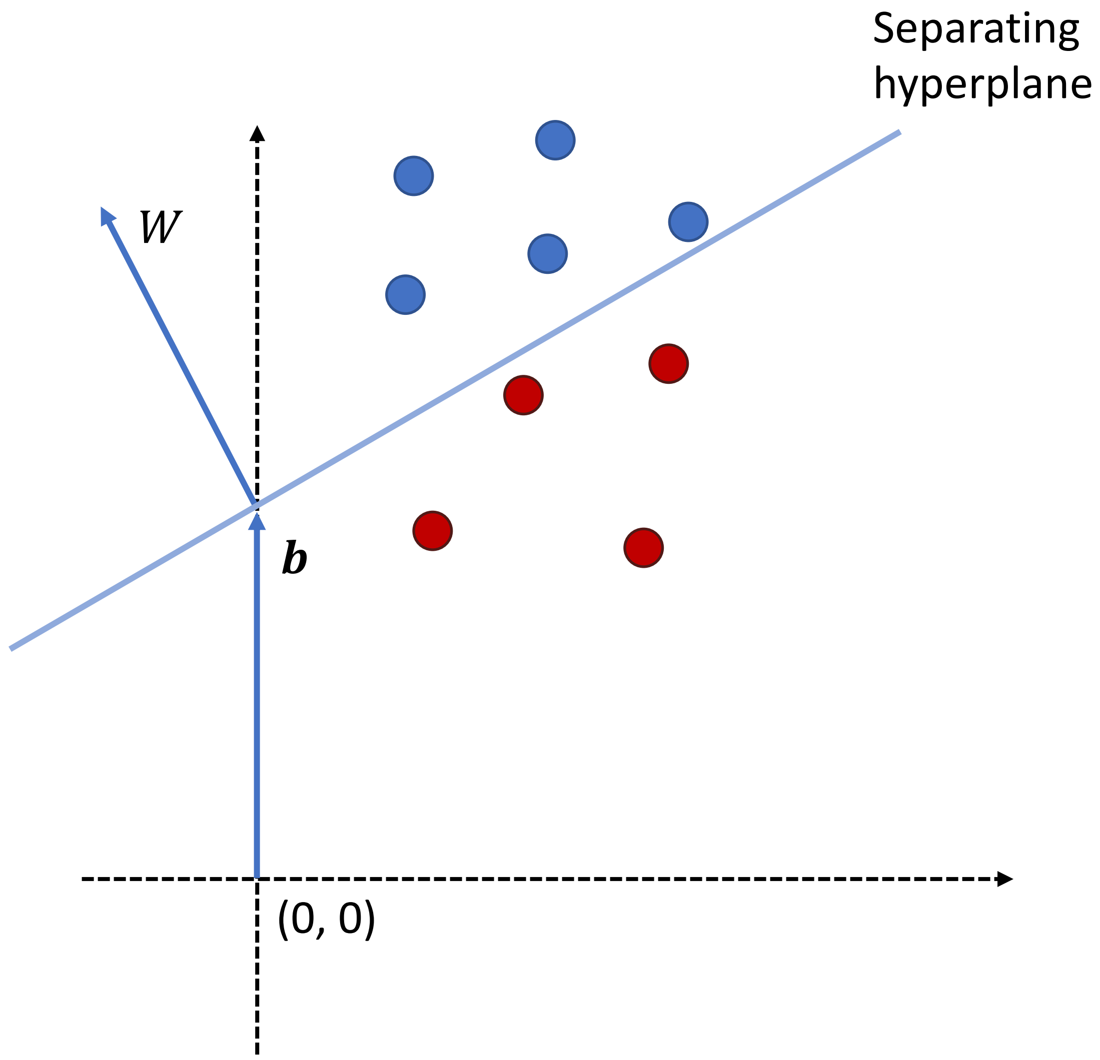
Single Neuron



With a **bias**

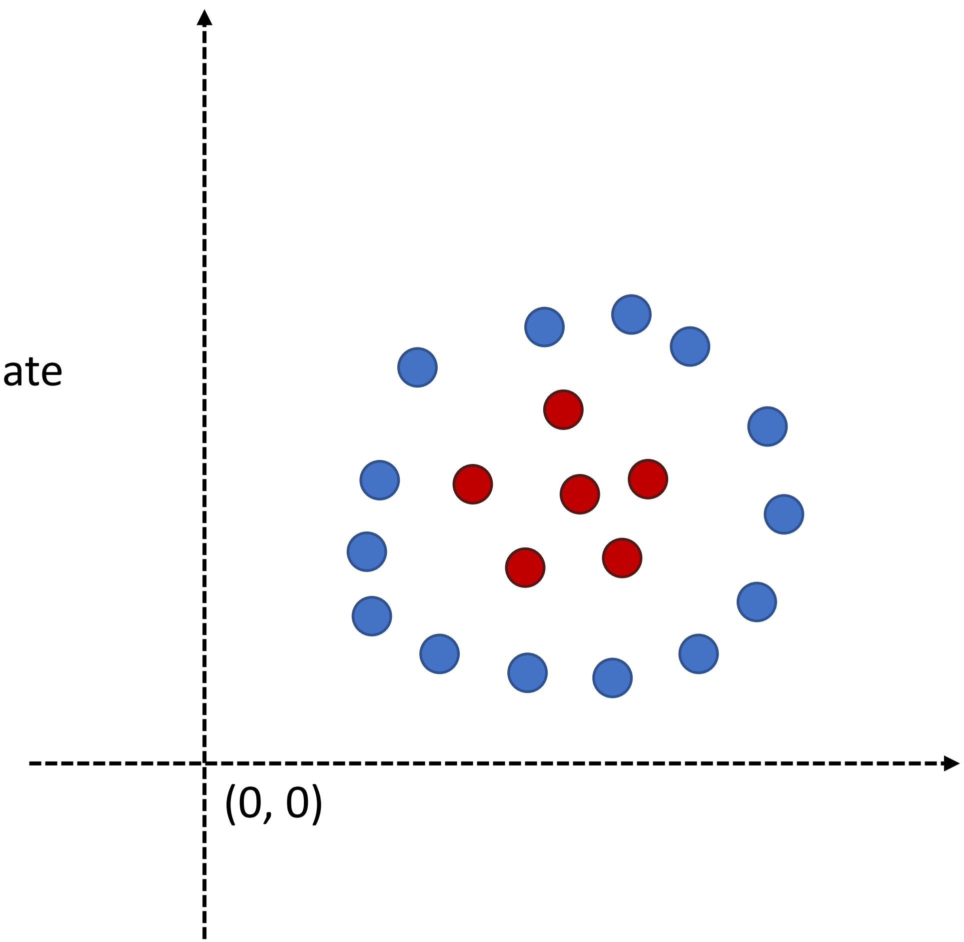
$$f(x) = \boldsymbol{\sigma}(\boldsymbol{w}\boldsymbol{x} + b)$$

With bias

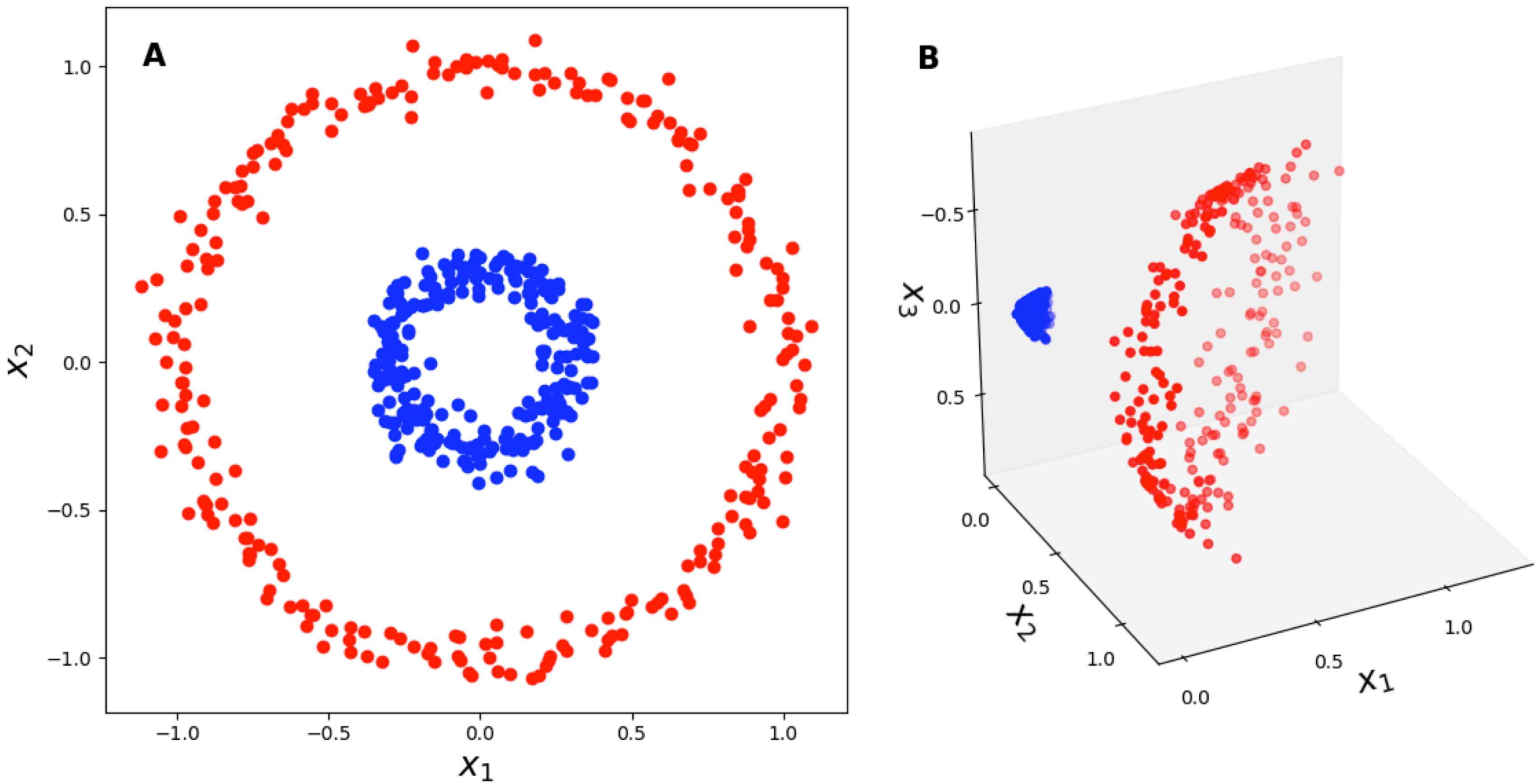


Non-Linear Data

A linear model cannot discriminate **blue** and **red** classes!

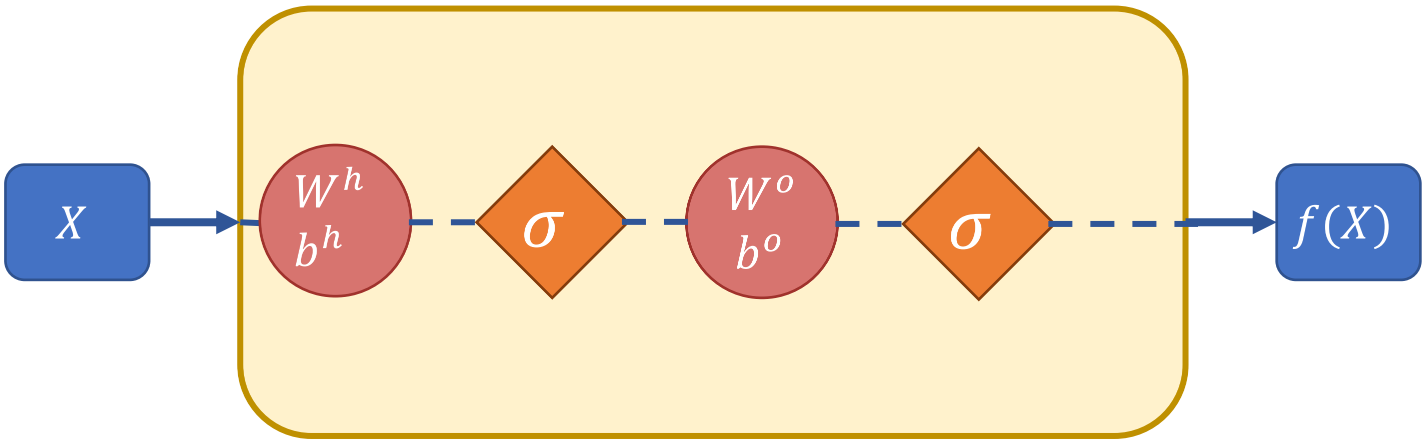


Before, you would have use the **kernel trick**



Excellent blog post on that [Gundersen]

Multi-Layer Perceptron

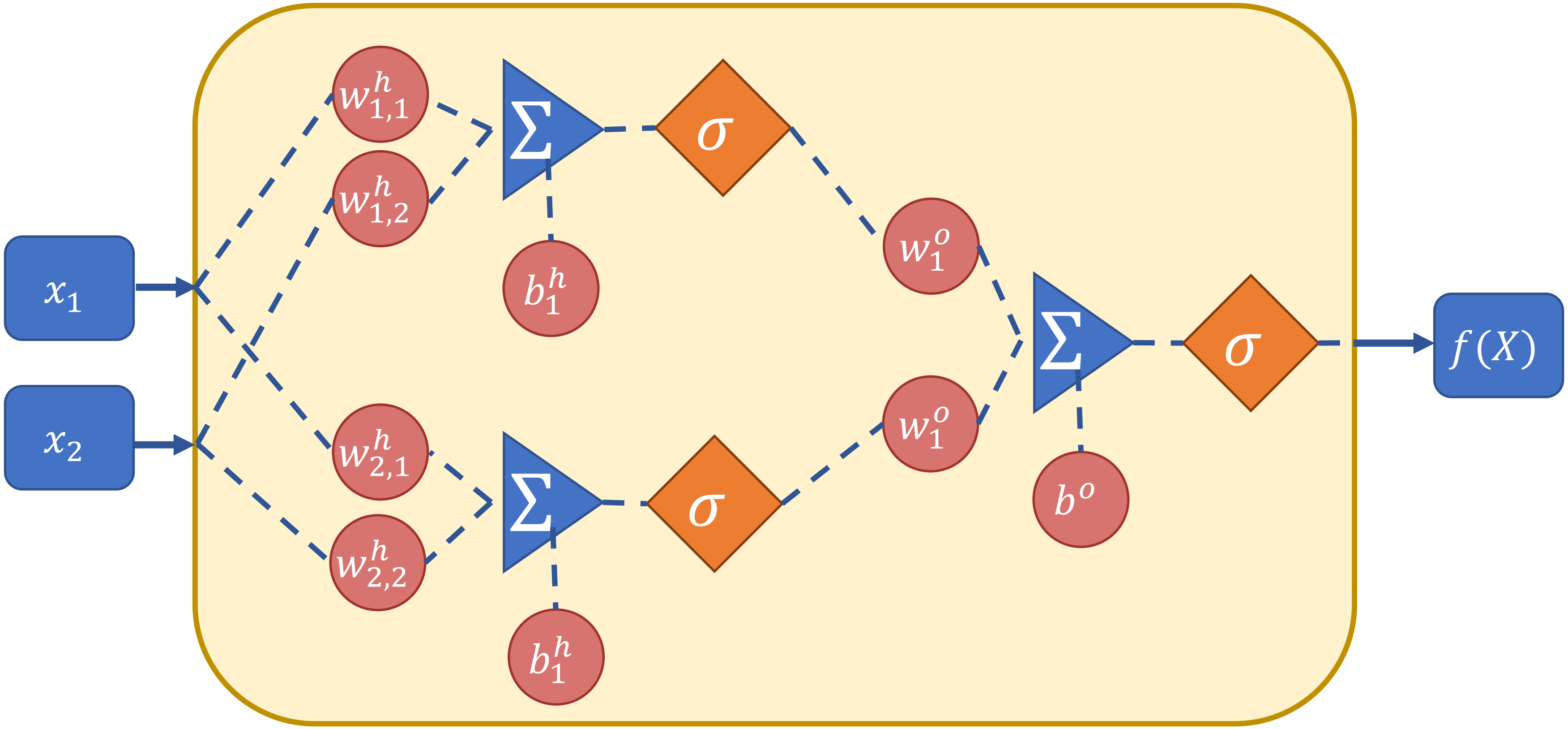


Stack multiple **linear transformations** and **non-linear activation**

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w^o}\sigma\left(\boldsymbol{W^h x} + \boldsymbol{b^h}\right) + \mathrm{b}^o)$$

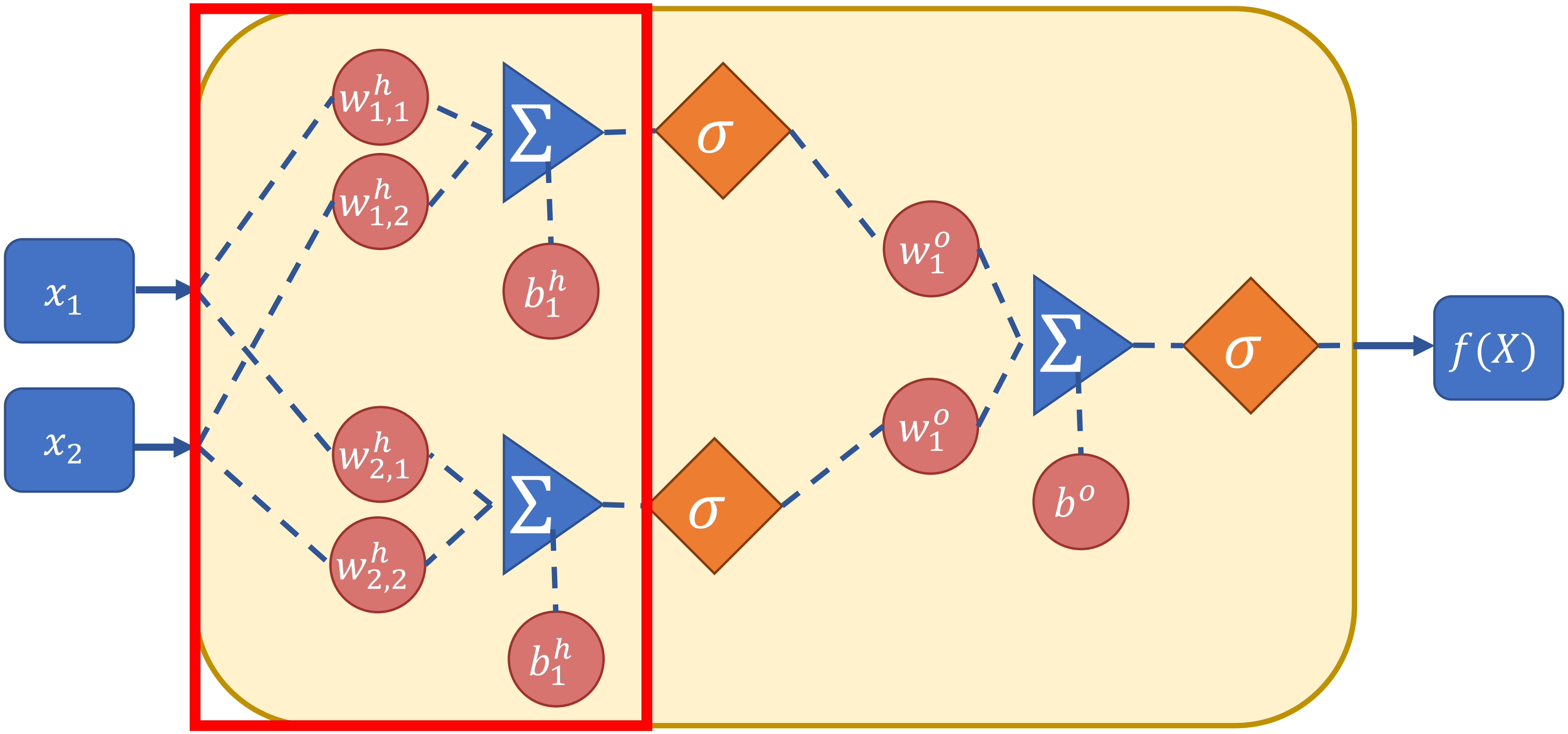
Multi-Layer Perceptron

$\boldsymbol{W}^h$ is a matrix because it has here 2 output dimensions



$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w^o}\sigma\left(\boldsymbol{W^h x} + \boldsymbol{b^h}\right) + \mathrm{b}^o)$$

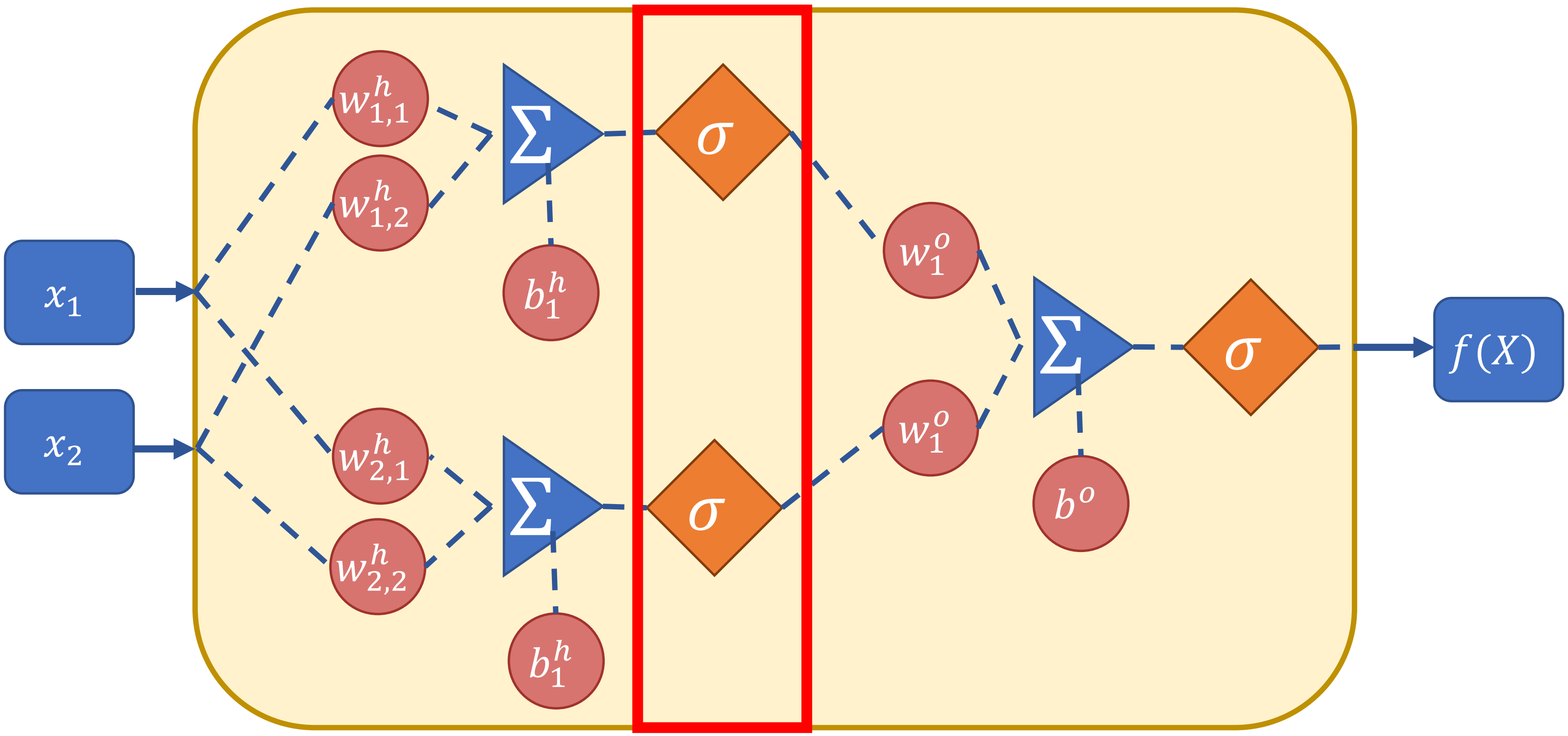
Multi-Layer Perceptron



$$\widetilde{\boldsymbol{h}} = \boldsymbol{W}^h \boldsymbol{x} + \boldsymbol{b}^h$$

Multi-Layer Perceptron

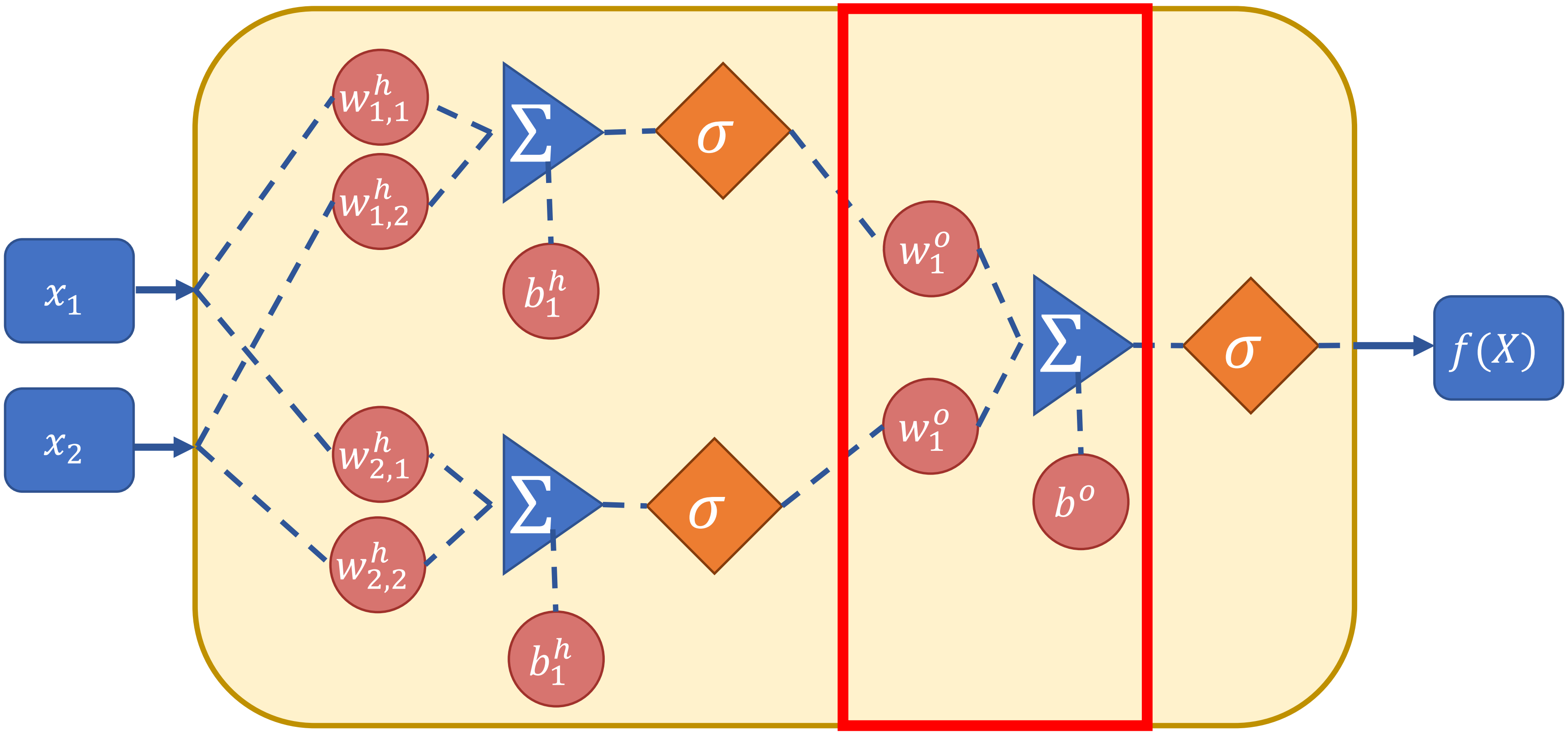
Embeddings/features



$$\boldsymbol{h} = \sigma(\widetilde{\boldsymbol{h}})$$

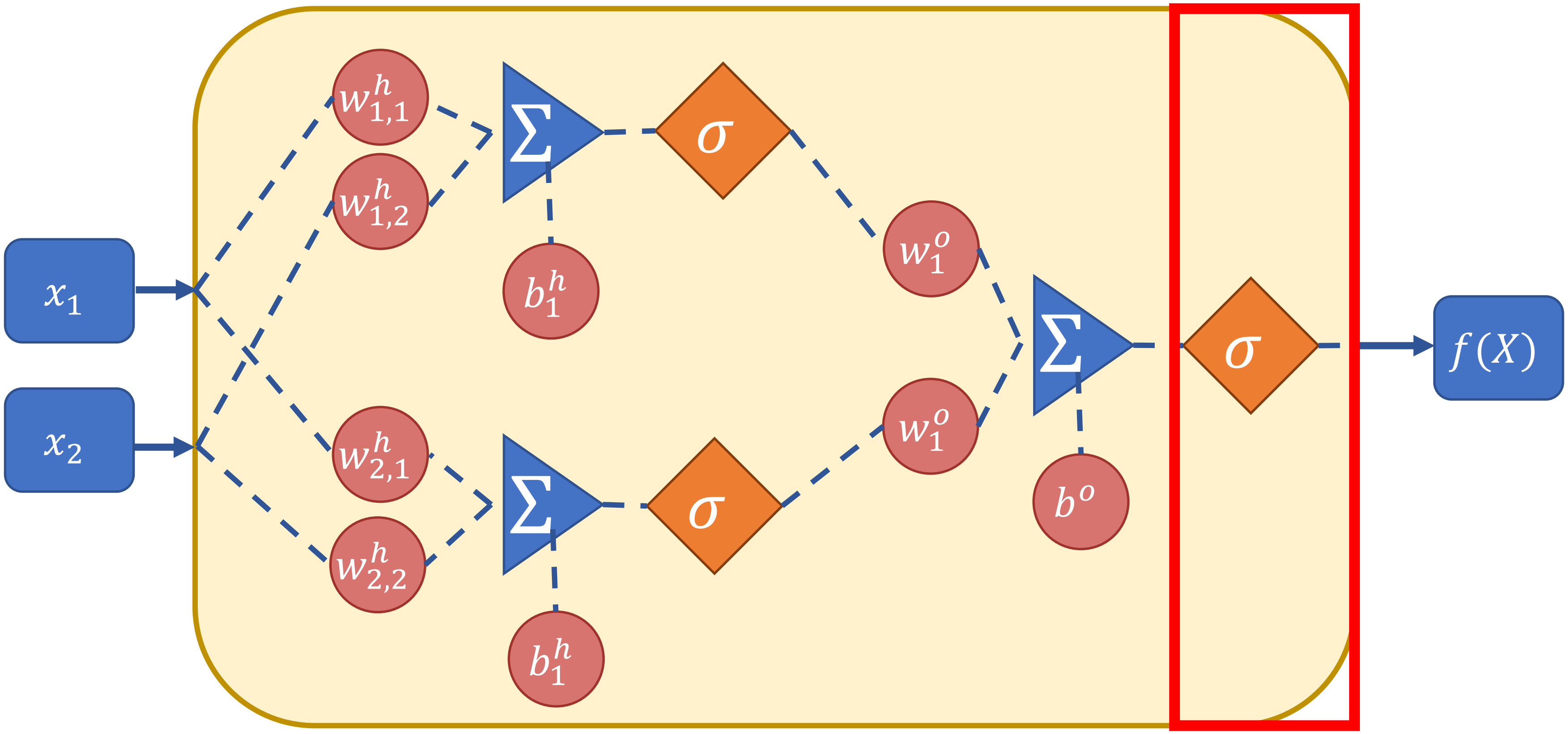
Multi-Layer Perceptron

Logits!



$$\tilde{\boldsymbol{y}} = \boldsymbol{w^o h} + b^o$$

Multi-Layer Perceptron

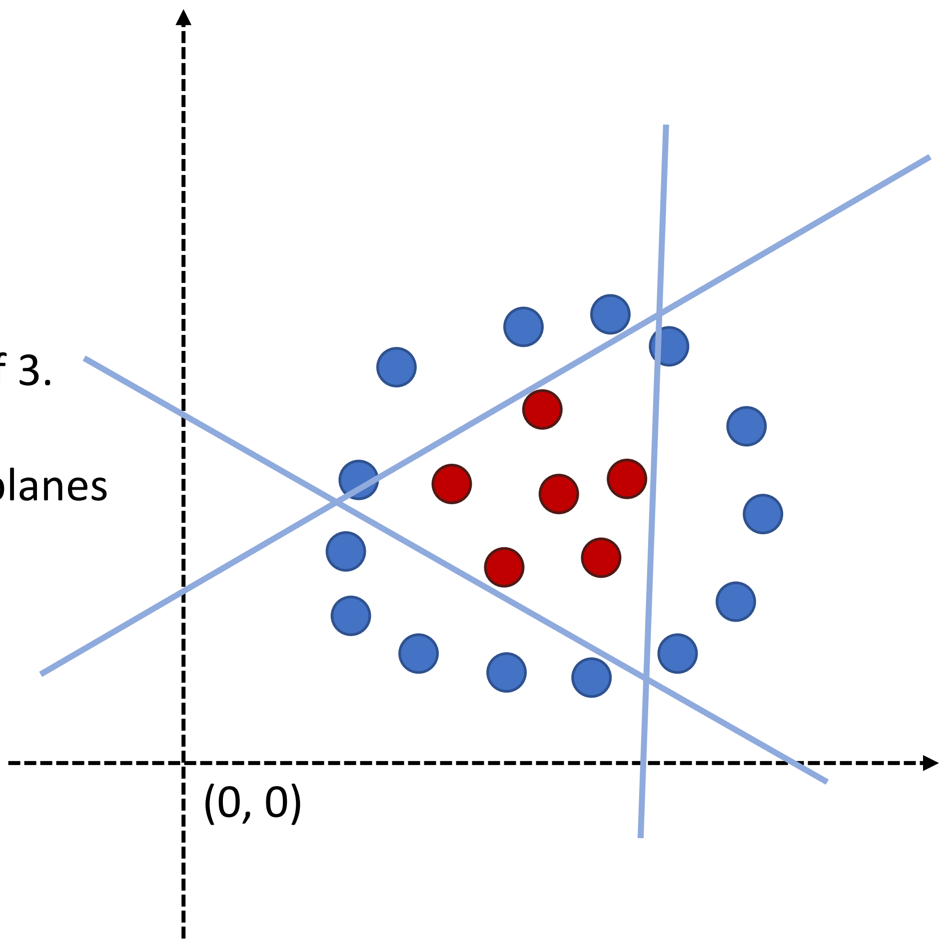


$$\hat{\boldsymbol{y}} = \sigma(\tilde{\boldsymbol{y}})$$

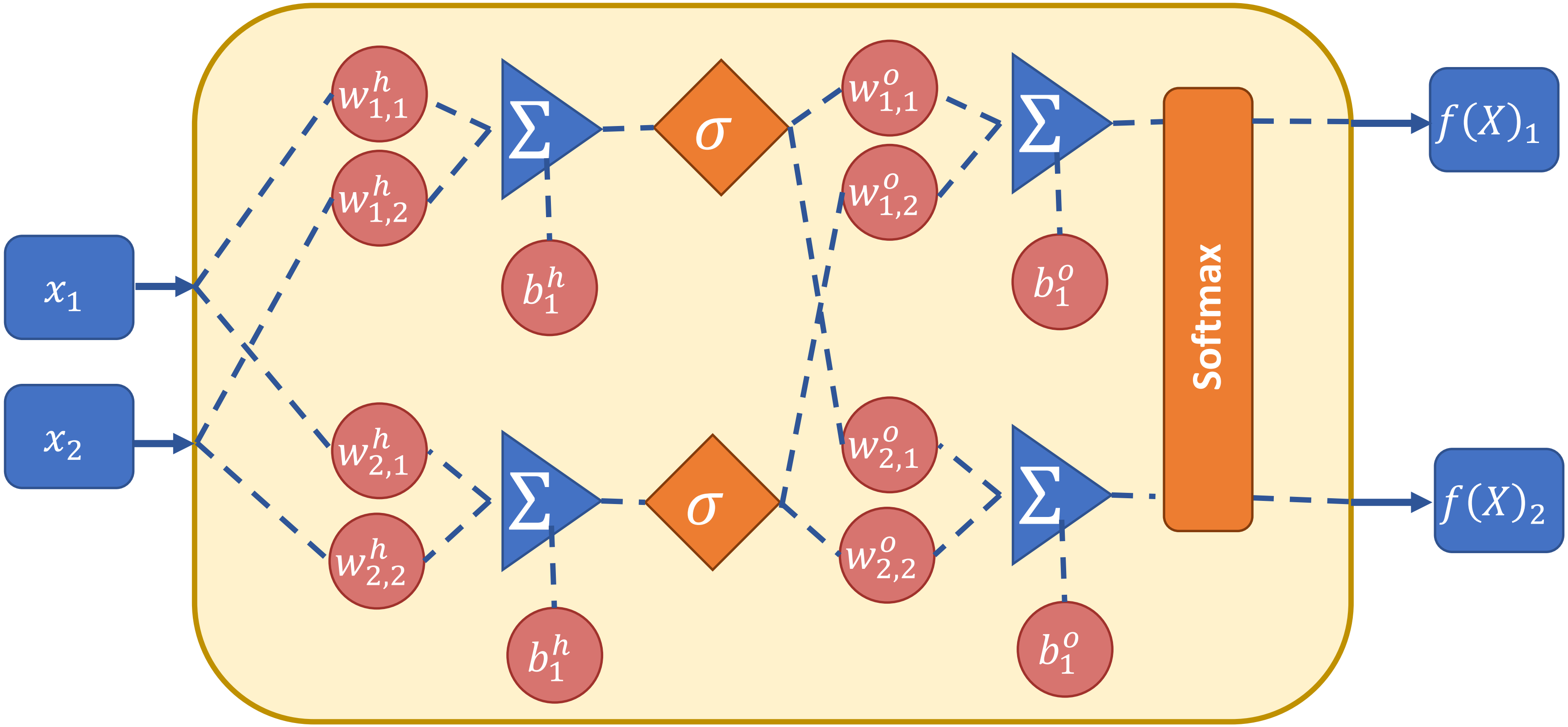
Non-Linear Data

Considering hidden dimension of 3.

Hidden layer can define 3 hyperplanes



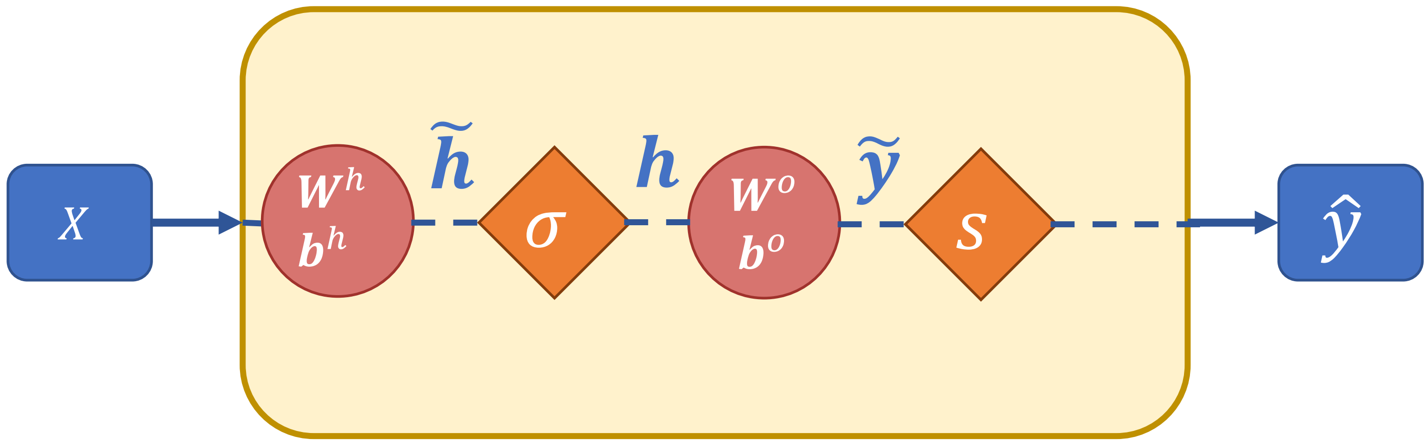
Multi-Class



W^o is a matrix because it has here 2 predicted classes.
Can be extended to 3, 4, …, 1000 classes.

Multi-Layer Perceptron

Multi-Layer Perceptron



$$\widetilde{\boldsymbol{h}} = \boldsymbol{W}^h \boldsymbol{x} + \boldsymbol{b}^h$$

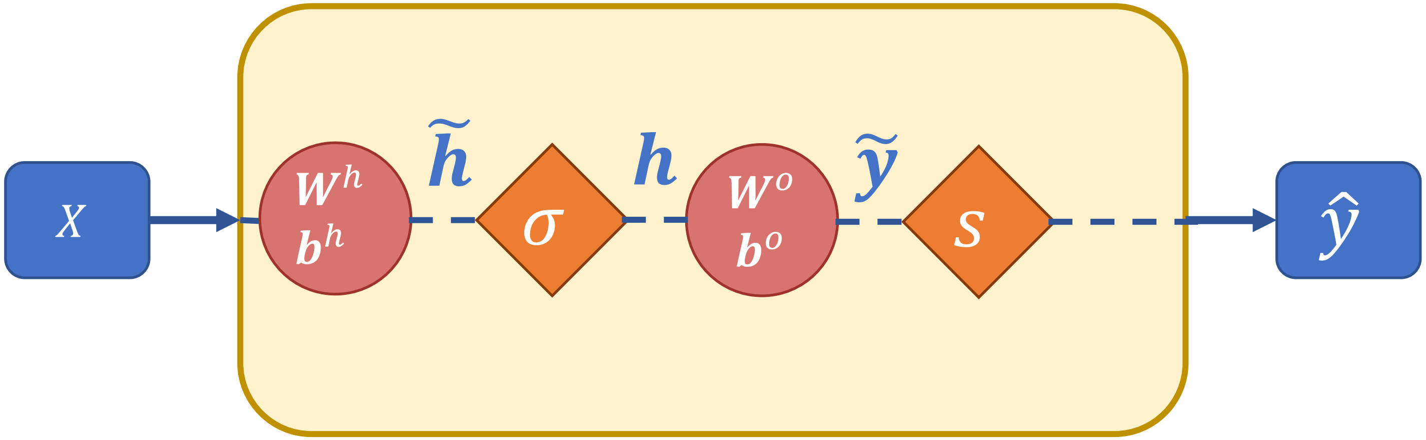
$$\boldsymbol{h} = \sigma(\widetilde{\boldsymbol{h}})$$

$$\widetilde{\boldsymbol{y}} = \boldsymbol{W}^o \boldsymbol{h} + \boldsymbol{b}^o$$

$$\widehat{\boldsymbol{y}} = \sigma(\widetilde{\boldsymbol{y}})$$

Multi-Layer Perceptron

Multi-Layer Perceptron



$$\tilde{\boldsymbol{h}} = \boldsymbol{W}^h \boldsymbol{x} + \boldsymbol{b}^h$$

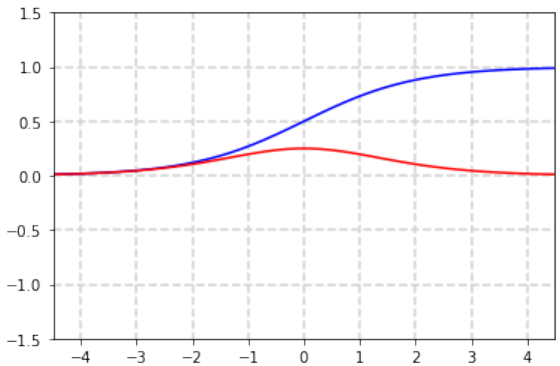
$$\boldsymbol{h} = \sigma(\tilde{\boldsymbol{h}})$$ → features / embeddings

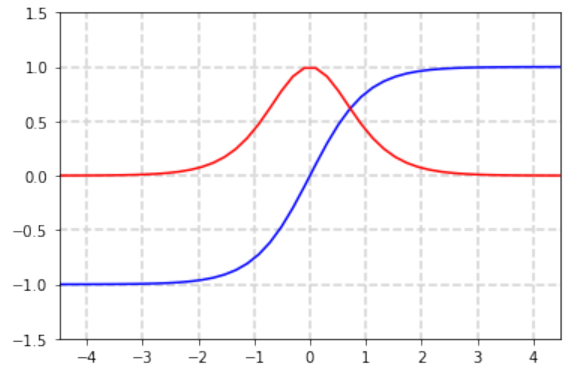
$$\tilde{\boldsymbol{y}} = \boldsymbol{W}^o \boldsymbol{h} + \boldsymbol{b}^o$$ → logits

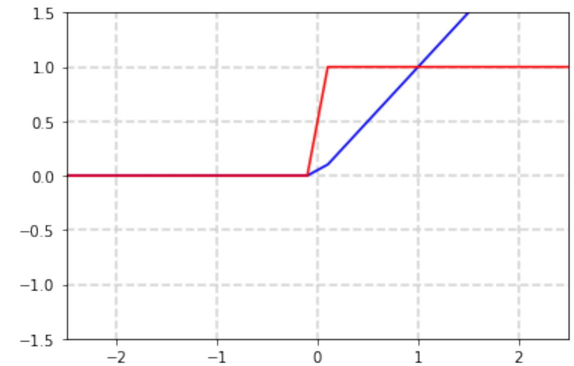
$$\hat{\boldsymbol{y}} = \sigma(\tilde{\boldsymbol{y}})$$ → Model predictions

Hidden Activations

Function and their **derivative**







Sigmoid

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

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

Hyperbolic tangent

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

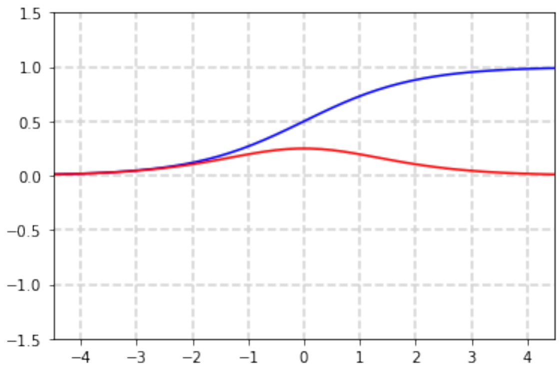
$$\frac{d}{dx}\tanh(x) = 1 - \tanh(x)\text{^}2$$

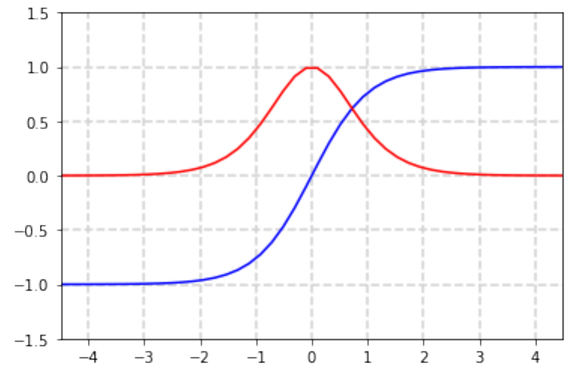
Rectified Linear Unit

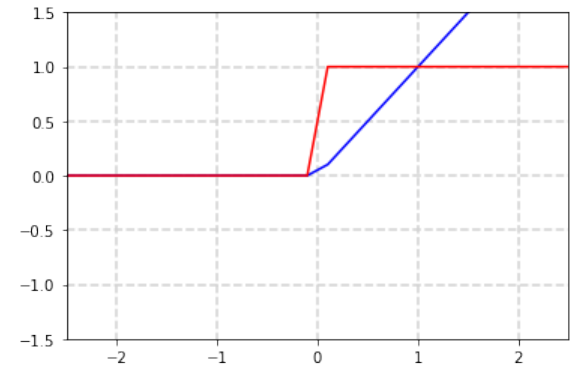
$$\mathrm{ReLU}(x) = \max(x, 0)$$

$$\frac{d}{dx} ReLU(x) = \begin{cases} 1 \; if \; x > 0 \\ 0 \; otherwise \end{cases}$$

Function and their derivative







Sigmoid

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

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

Hyperbolic tangent

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\frac{d}{dx}\tanh(x) = 1 - \tanh(x)\text{^}2$$

Rectified Linear Unit

$$\mathrm{ReLU}(x) = \max(x, 0)$$

$$\frac{d}{dx}ReLU(x) = \begin{cases} 1\ if\ x > 0 \\ 0\ otherwise \end{cases}$$

Without hidden activation, a MLP is equivalent to a single layer!

→ Composition of affine functions is an affine function

Output activations

Binary classification: **sigmoid**
→ Applied element-wise
→ Range [0, 1]

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

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

Multi-Class classification: **softmax**
→ Applied over a vector
→ Range [0, 1] per element
→ Sum to 1 for the vector
  → Probability distribution

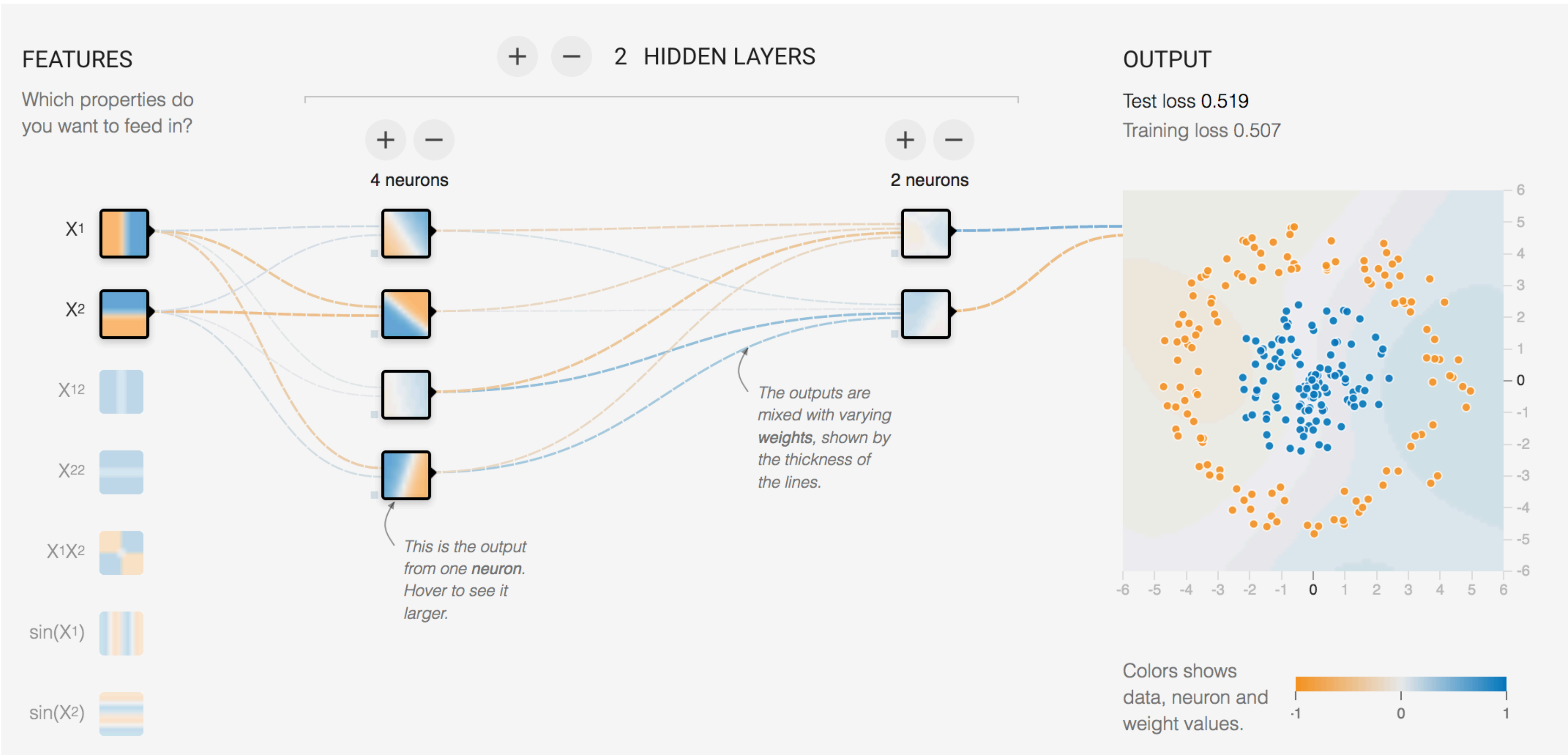
$$softmax(\boldsymbol{x}) = \frac{1}{\sum_j e^{\boldsymbol{x}_j}} \begin{bmatrix} e^{x_1} \\ \dots \\ e^{x_n} \end{bmatrix}$$

$$softmax(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}}$$

$$\frac{d}{dx_j} softmax(\boldsymbol{x})_i = \begin{cases} softmax(\boldsymbol{x})_i(1 - softmax(\boldsymbol{x})_i \ if \ i = j \\ -softmax(\boldsymbol{x})_i softmax(\boldsymbol{x})_j \ \ if \ i \neq j \end{cases}$$

Intuition

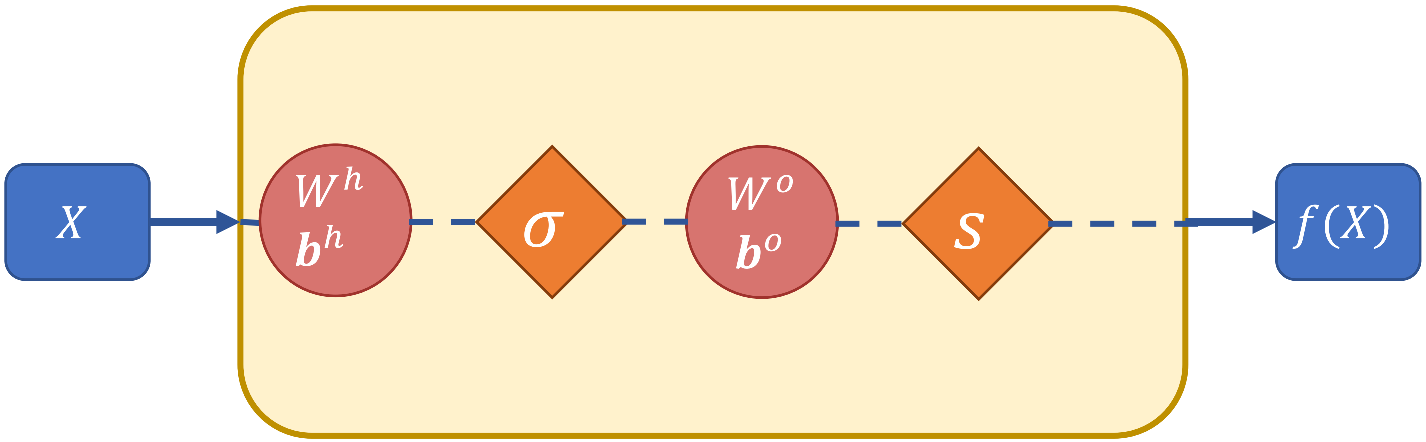
To get an intuition of the effect of hidden dimensions, number of layers, and activations:



playground.tensorflow.org

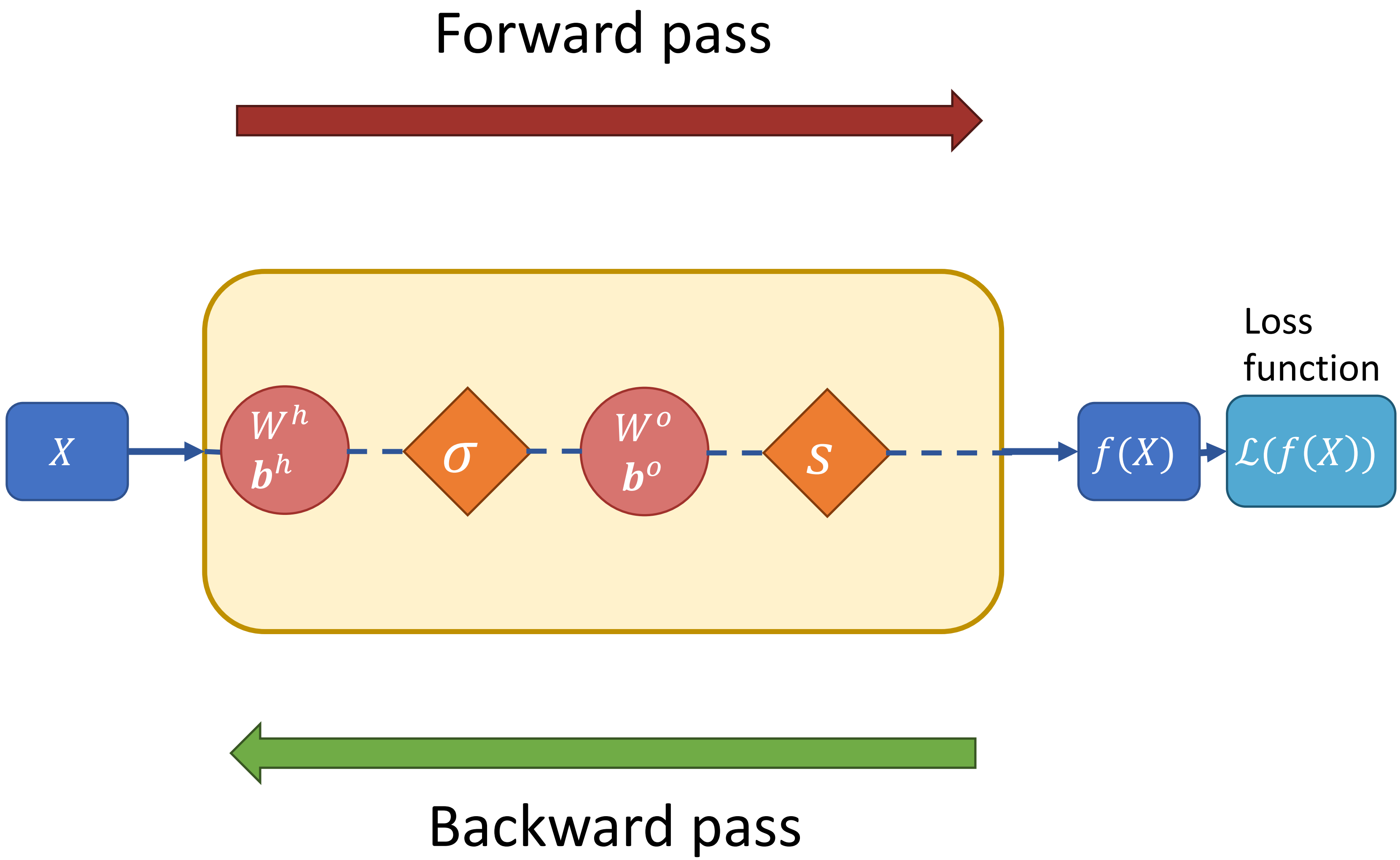
Learning DNNs

Multi-Layer Perceptron



1 hidden layer, H hidden dimensions, C output dimensions with a softmax

Forward & Backward passes

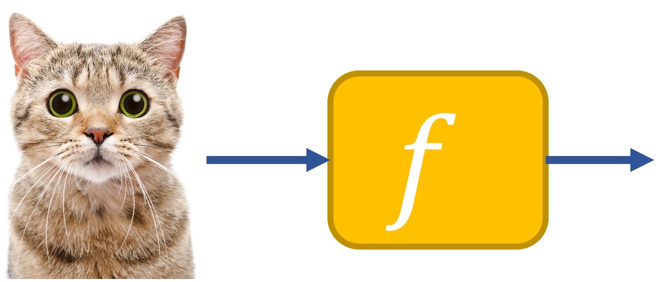


Loss Function

For classification with softmax as final activation:

Cross-entropy (also known as negative log-likelihood):

$$\mathcal{L}_{CE}(\hat{y},\ \underbrace{y}_{\text{One-hot target}}) = -\sum_{i} y_i \log \hat{y}_i$$



Dog: 0.2

Cat: 0.8

Cross-Entropy: $-\log 0.8 = 0.2231 \ldots$

One-Hot

Avoid doing *if* in GPUs, use one-hot in cross-entropy.

Given 5 classes, if the ground-truth class is 3:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Zero-indexed!

Loss Function

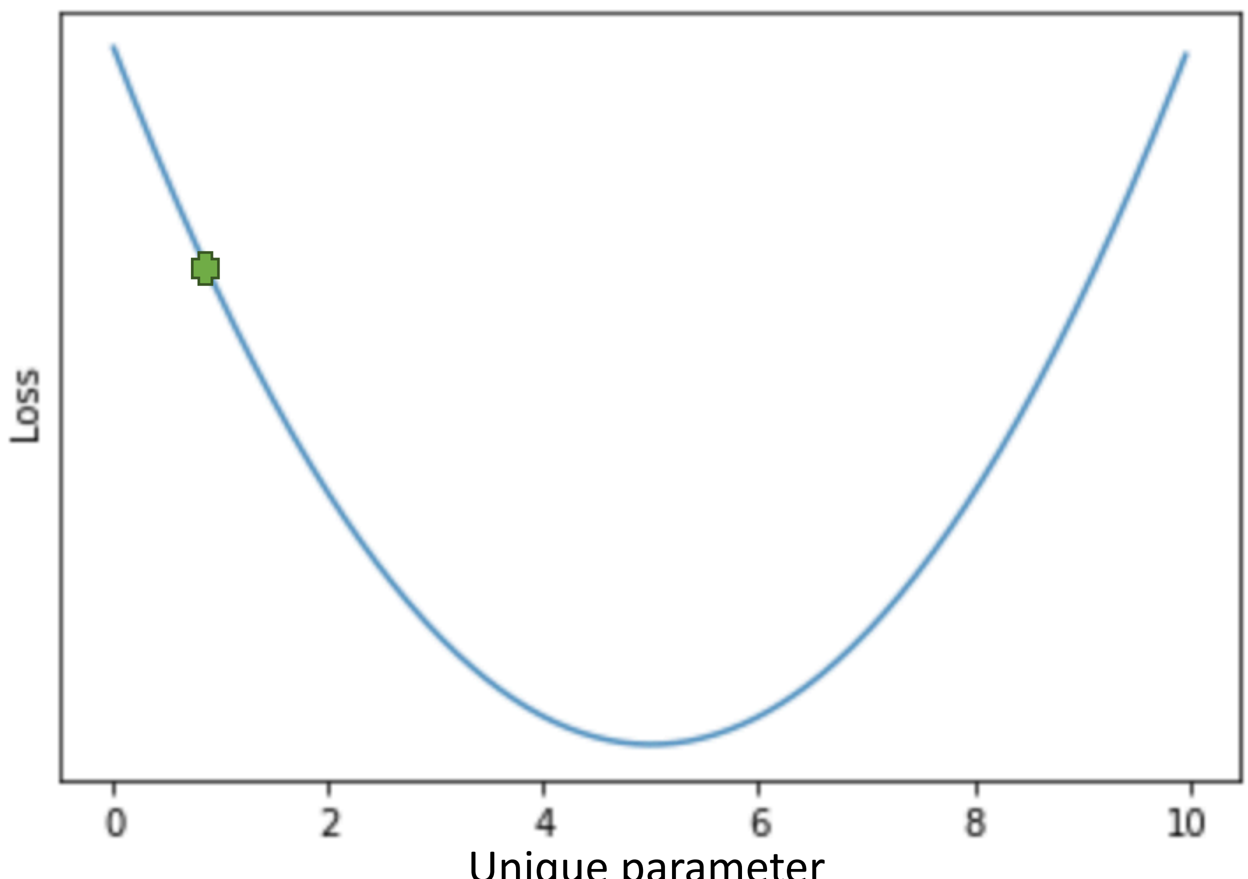
Optimize the network to minimize the loss with respect to all neurons:

$$\mathcal{L}_{CE}(\hat{y},\ y) = -\sum_i y_i \log \hat{y}_i$$

Minimization of a function

Optimize the network to minimize the loss with respect to all neurons:

$$\mathcal{L}_{CE}(\hat{y}, y) = -\sum y_i \log \hat{y}_i$$

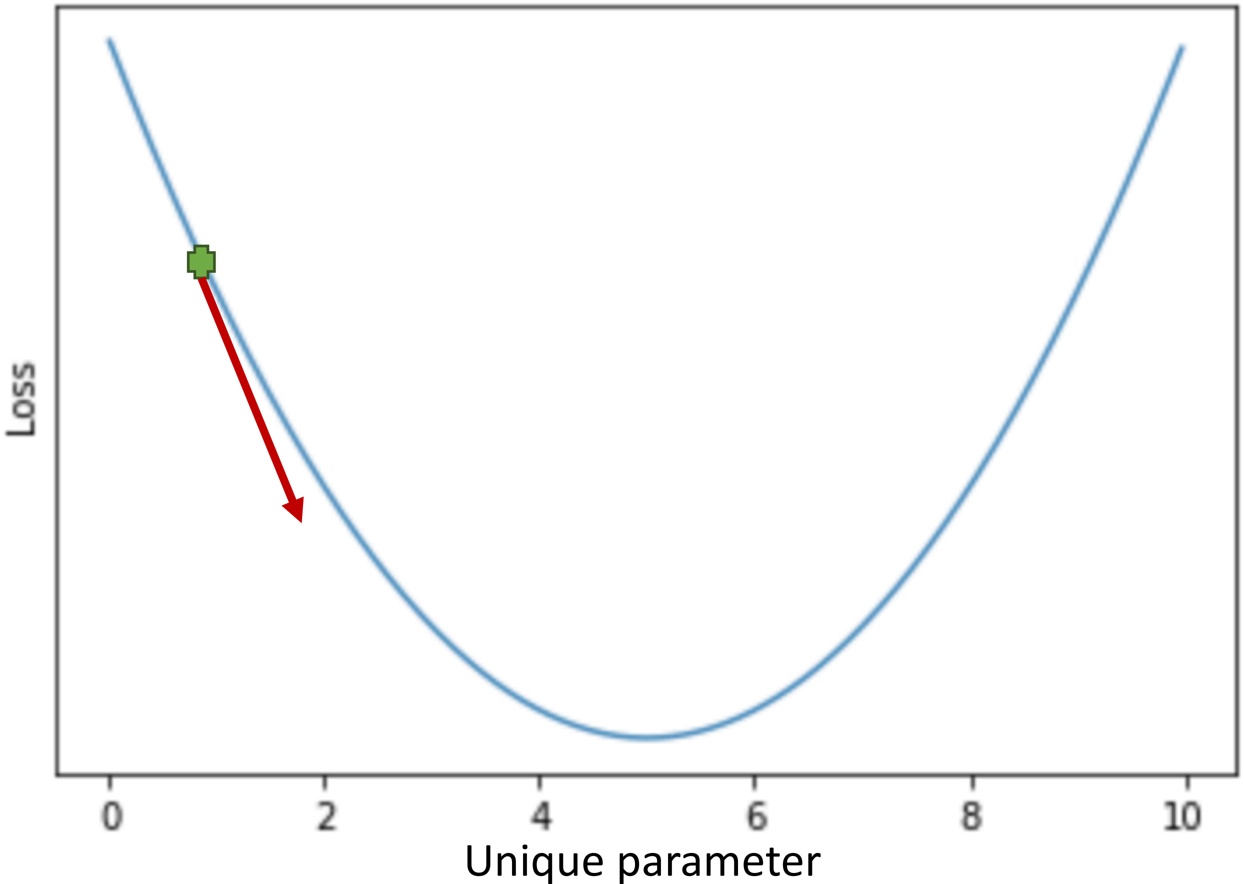


Minimization of a function

Optimize the network to minimize the loss with respect to all neurons θ:

$$\mathcal{L}_{CE}(\hat{y},\, y)$$

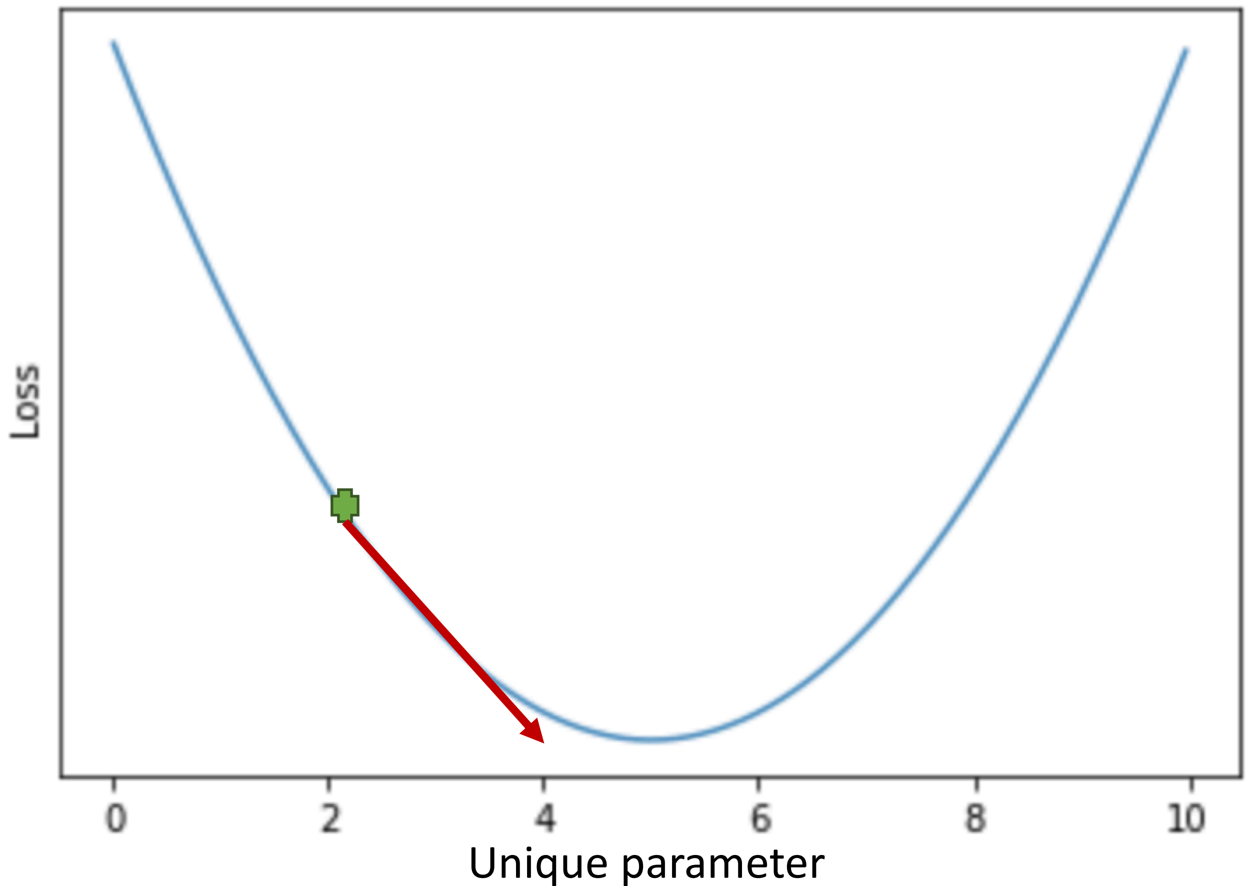
$$-\nabla_{\theta}\mathcal{L}_{CE}(\hat{y},\, y)$$



Minimization of a function

Optimize the network to minimize the loss with respect to all neurons θ:

$$\mathcal{L}_{CE}(\hat{y}, y)$$
$$-\nabla_\theta \mathcal{L}_{CE}(\hat{y}, y)$$

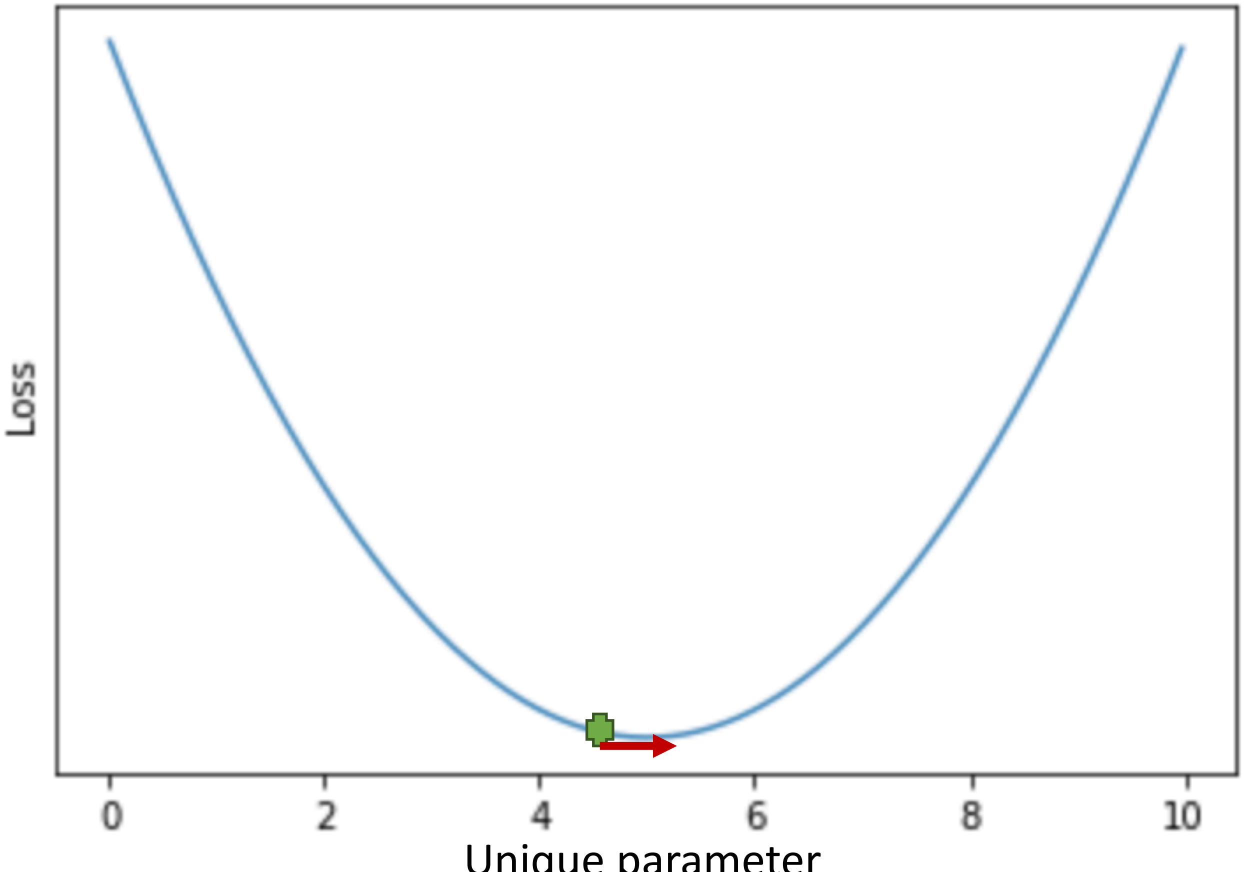


Minimization of a function

Optimize the network to minimize the loss with respect to all neurons θ:

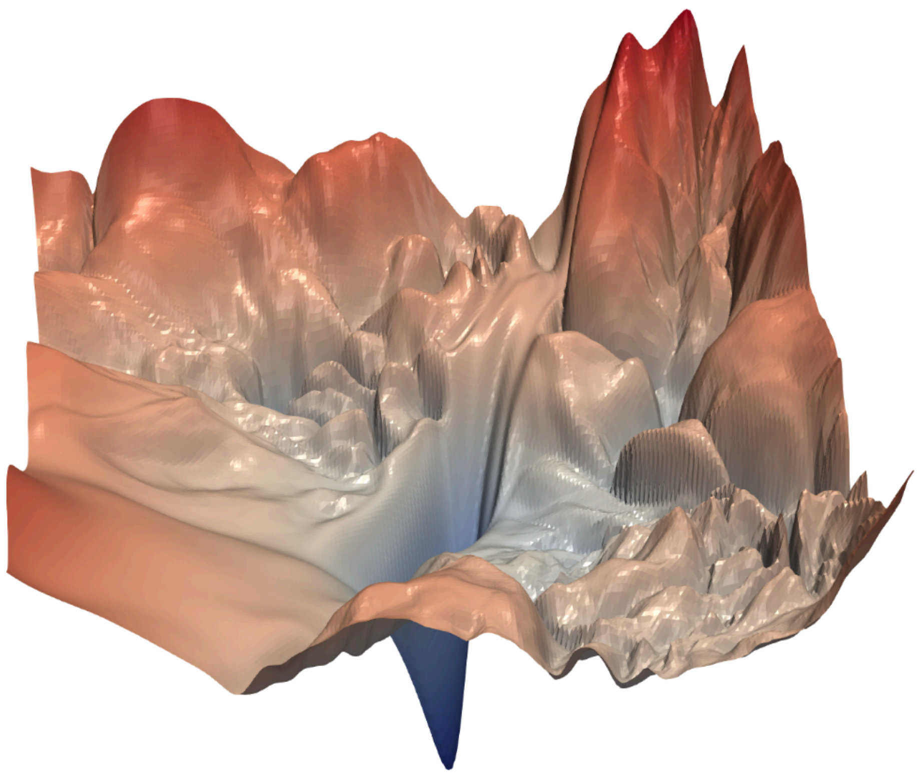
$$\mathcal{L}_{CE}(\hat{y}, y)$$

$$-\nabla_\theta \mathcal{L}_{CE}(\hat{y}, y)$$



Minimization of a function

Million of parameters to optimize together!
Highly non-convex!

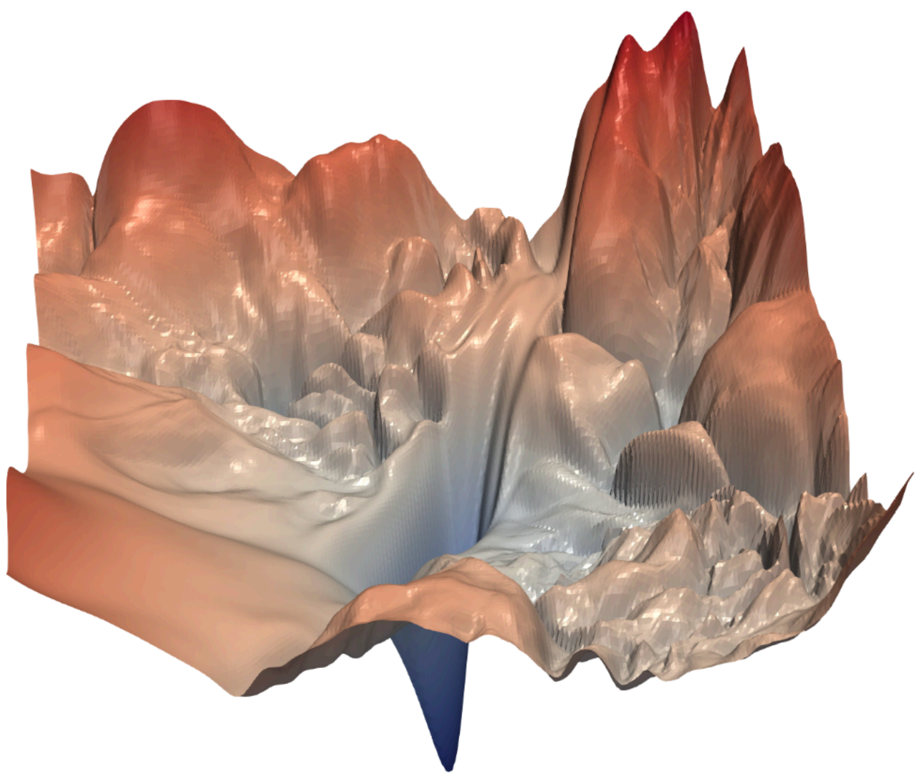


Multiple dimensional derivatives are called **gradients**

Visualizing the Loss Landscape of Neural Nets, [Li et al. NeurIPS2018]

Minimization of a function

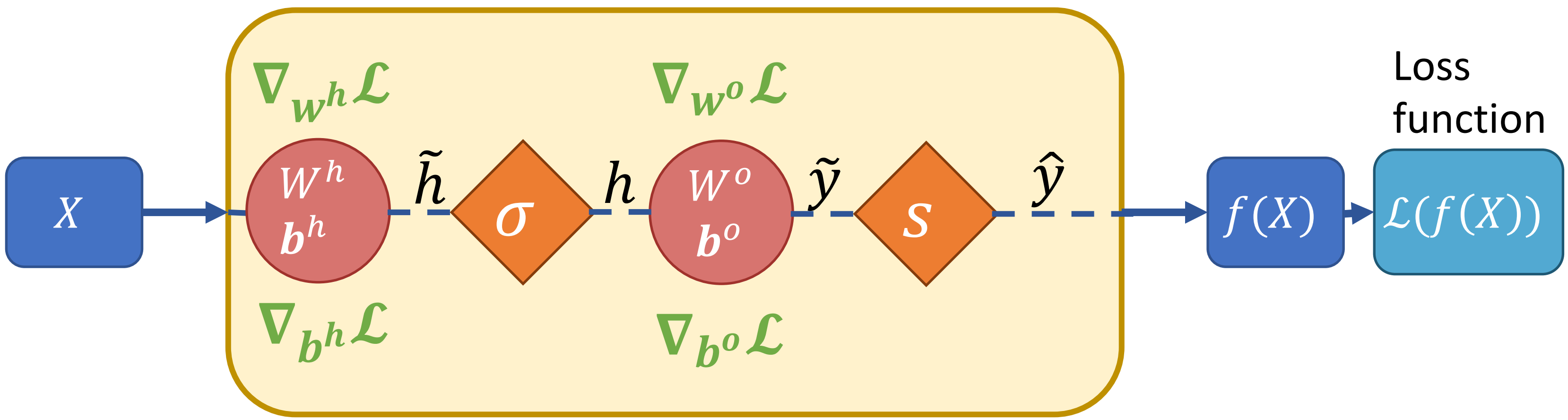
In theory a lot of bad local minima, but in practice a DNN usually find a good local minima close to optimum [LeCun et al. Nature 2015].



Although, this is debated [Goldblum et al. ICLR 2020]

Needed Gradients

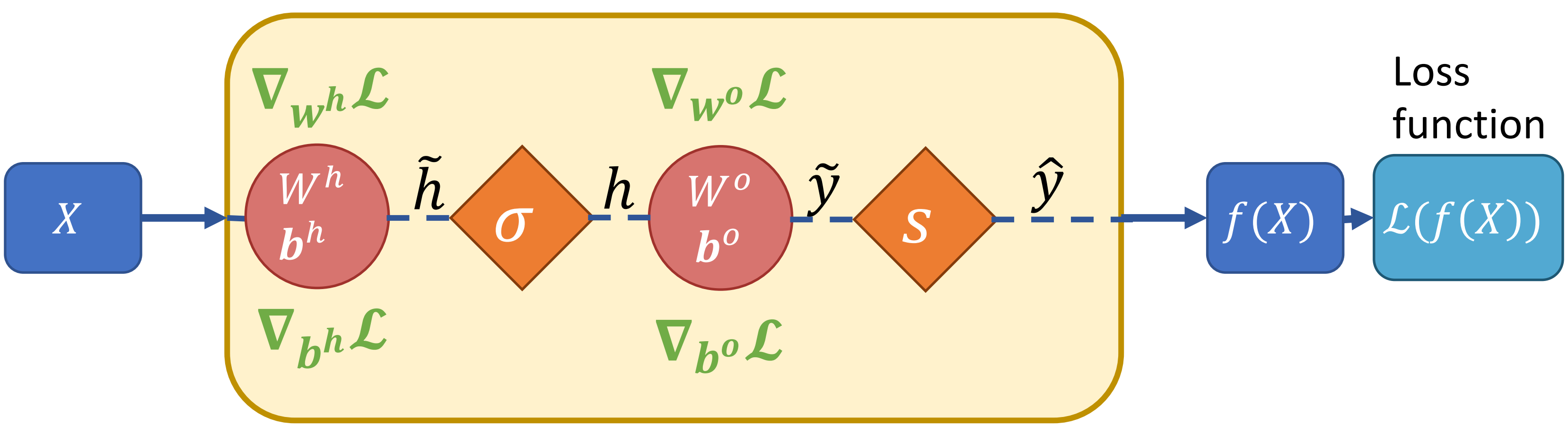
We need all parameters gradients in **green**



Stochastic Gradient Descent

1. Initialize randomly the parameters θ
2. For each epoch do
   1. Select a random sample of the data
   2. Forward
3. Compute gradients $\nabla_\theta \mathcal{L}$ for each parameter θ
4. Update parameters $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$

η is the **learning rate**.



Chain Rule

$$f \circ g(a) = f(b) = c$$

Given scalars:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a}$$

Given vectors, element-wise:

$$\frac{\partial c_i}{\partial a_k} = \sum_i \frac{\partial c_i}{\partial b_j}\frac{\partial b_j}{\partial a_k}$$

Given vectors, vector-wise:

$$\boldsymbol{\nabla_a c} = \boldsymbol{\nabla_b c}\ \boldsymbol{\nabla_a b^T}$$

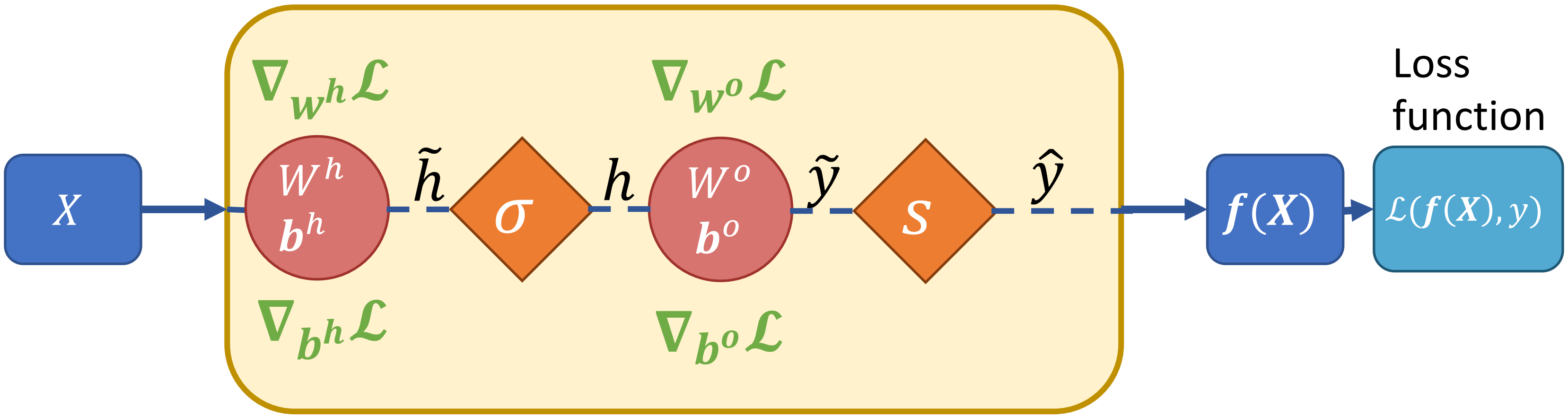
In denominator layout (≠ numerator layout):

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_{n_a} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_{n_b} \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_{n_c} \end{bmatrix} \qquad \boldsymbol{\nabla_b c} = \begin{bmatrix} \frac{\partial c_1}{\partial b_1} & \cdots & \frac{\partial c_{n_c}}{\partial b_1} \\ \vdots & & \vdots \\ \frac{\partial c_1}{\partial b_{n_b}} & \cdots & \frac{\partial c_{n_c}}{\partial b_{n_b}} \end{bmatrix}$$

Backpropagation

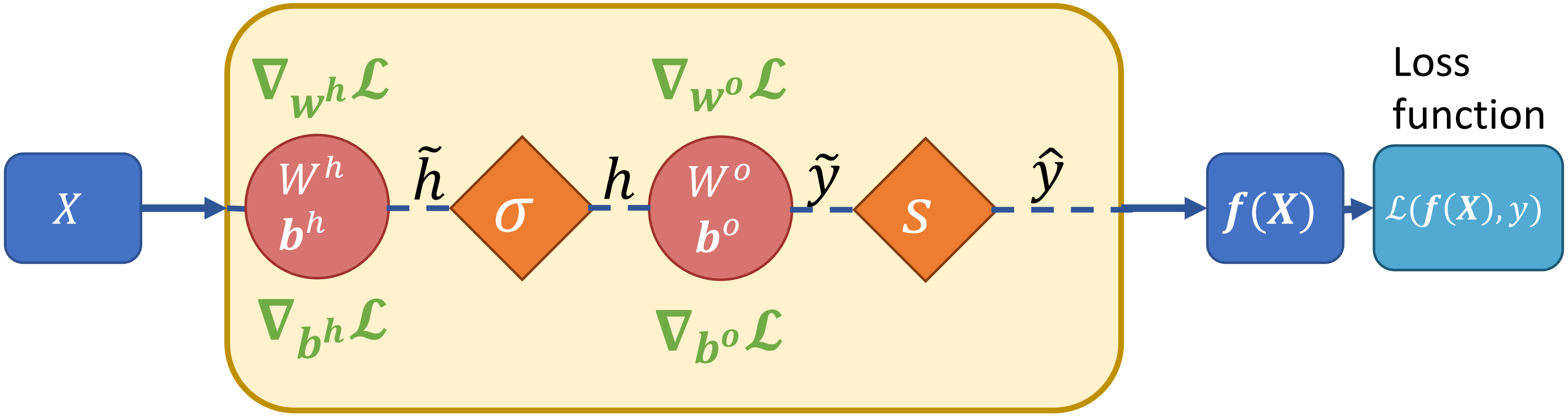
Partial derivative of the cross-entropy loss with respect to (w.r.t.) the probabilities:

$$\frac{\partial \mathcal{L}(\boldsymbol{f}(\boldsymbol{x}),y)}{\partial \boldsymbol{f}(\boldsymbol{x})_i} = \frac{\partial - \log \boldsymbol{f}(\boldsymbol{x})_y}{\partial \boldsymbol{f}(\boldsymbol{x})_i} = \frac{-1_{y=i}}{\boldsymbol{f}(\boldsymbol{x})_y}, \text{ for simplicity } \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}(\boldsymbol{x})_i}$$



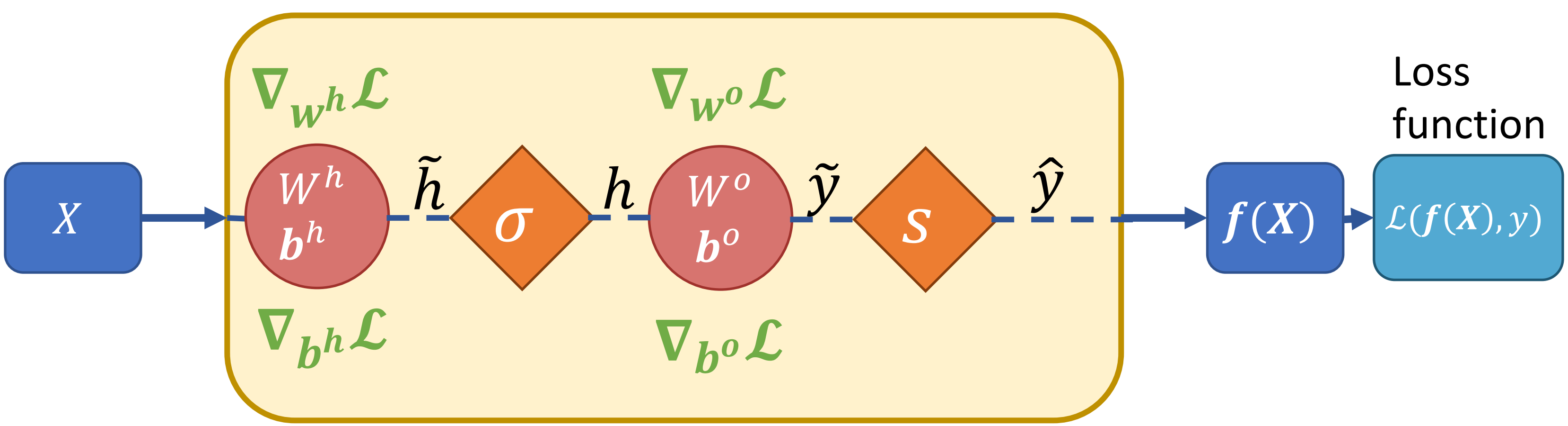
Backpropagation

$$\frac{\partial \mathcal{L}}{\partial \tilde{y}_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}(\boldsymbol{x})_j} \frac{\partial \boldsymbol{f}(\boldsymbol{x})_j}{\partial \tilde{y}_i}$$



Backpropagation

$$\frac{\partial \mathcal{L}}{\partial \tilde{y}_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}(\boldsymbol{x})_j} \frac{\partial \boldsymbol{f}(\boldsymbol{x})_j}{\partial \tilde{y}_i}$$
$$= \sum_j \frac{-1_{y=j}}{\partial \boldsymbol{f}(\boldsymbol{x})_y} \frac{\partial softmax(\tilde{y})_j}{\partial \tilde{y}_i}$$

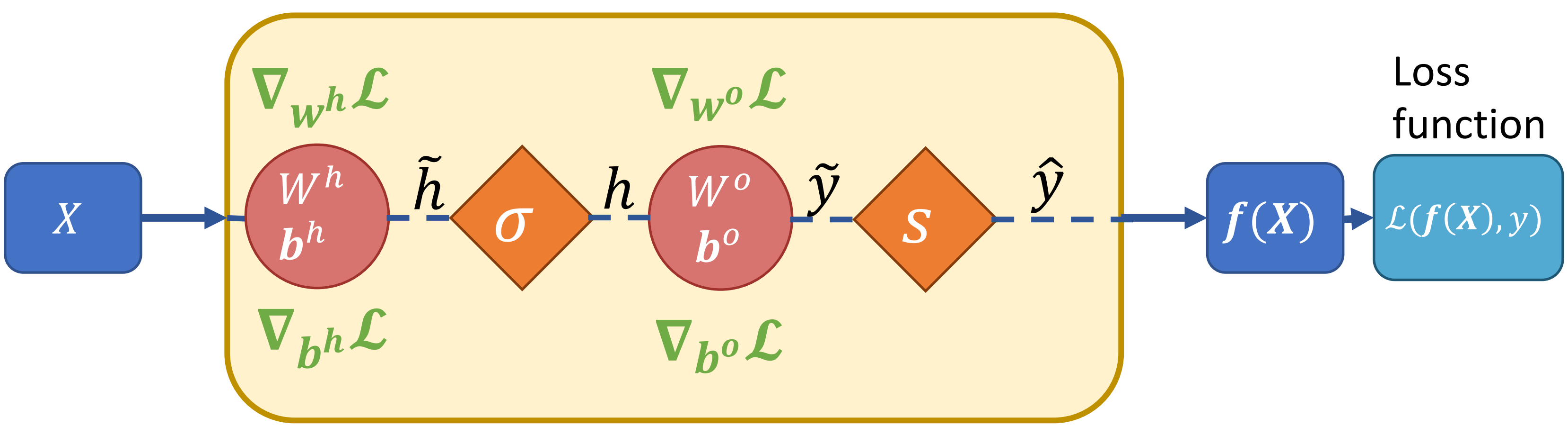


Backpropagation

$$\frac{\partial \mathcal{L}}{\partial \tilde{y}_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}(\boldsymbol{x})_j} \frac{\partial \boldsymbol{f}(\boldsymbol{x})_j}{\partial \tilde{y}_i}$$

$$= \sum_j \frac{-1_{y=j}}{\partial \boldsymbol{f}(\boldsymbol{x})_y} \frac{\partial softmax(\tilde{y})_j}{\partial \tilde{y}_i}$$

$$= \begin{cases} \frac{-1}{\boldsymbol{f}(\boldsymbol{x})_y} softmax(\tilde{y})_y \left(1 - softmax(\tilde{y})_y\right) & if\ i = y \\ \frac{1}{\boldsymbol{f}(\boldsymbol{x})_y} softmax(\tilde{y})_y softmax(\tilde{y})_i & if\ i \neq y \end{cases}$$



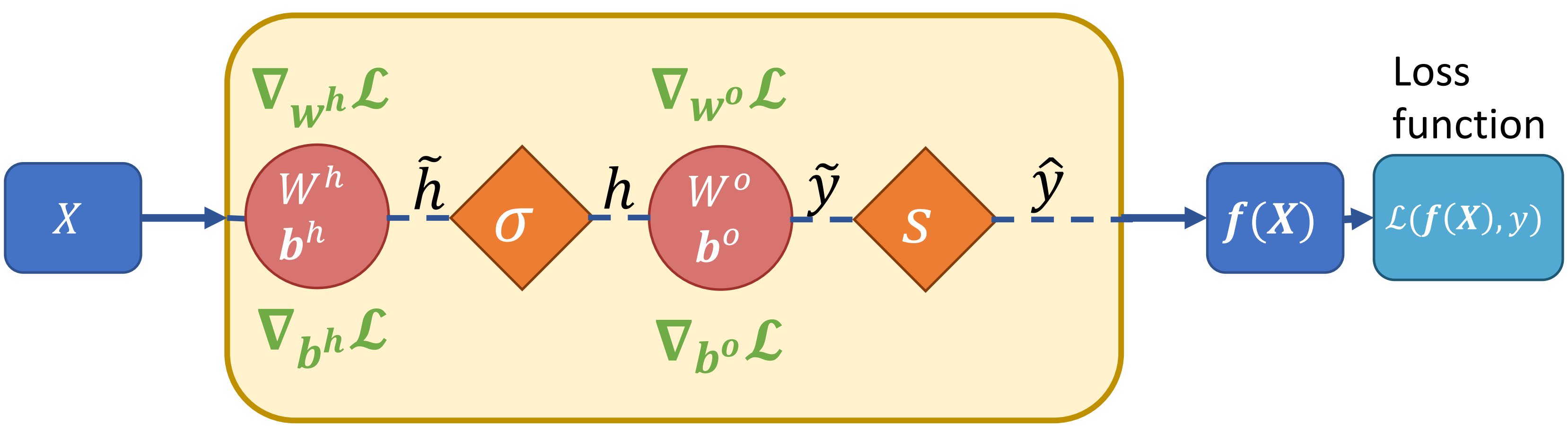
Backpropagation

$$\frac{\partial \mathcal{L}}{\partial \tilde{y}_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}(\boldsymbol{x})_j} \frac{\partial \boldsymbol{f}(\boldsymbol{x})_j}{\partial \tilde{y}_i}$$

$$= \sum_j \frac{-1_{y=j}}{\partial \boldsymbol{f}(\boldsymbol{x})_y} \frac{\partial softmax(\tilde{y})_j}{\partial \tilde{y}_i}$$

$$= \begin{cases} \dfrac{-1}{\boldsymbol{f}(\boldsymbol{x})_y} softmax(\tilde{y})_y \left(1 - softmax(\tilde{y})_y\right) & if\ i = y \\ \dfrac{1}{\boldsymbol{f}(\boldsymbol{x})_y} softmax(\tilde{y})_y softmax(\tilde{y})_i & if\ i \neq y \end{cases}$$

$$= \begin{cases} -1 + \boldsymbol{f}(\boldsymbol{x})_y & if\ i = y \\ \boldsymbol{f}(\boldsymbol{x})_i & if\ i \neq y \end{cases}$$



Backpropagation

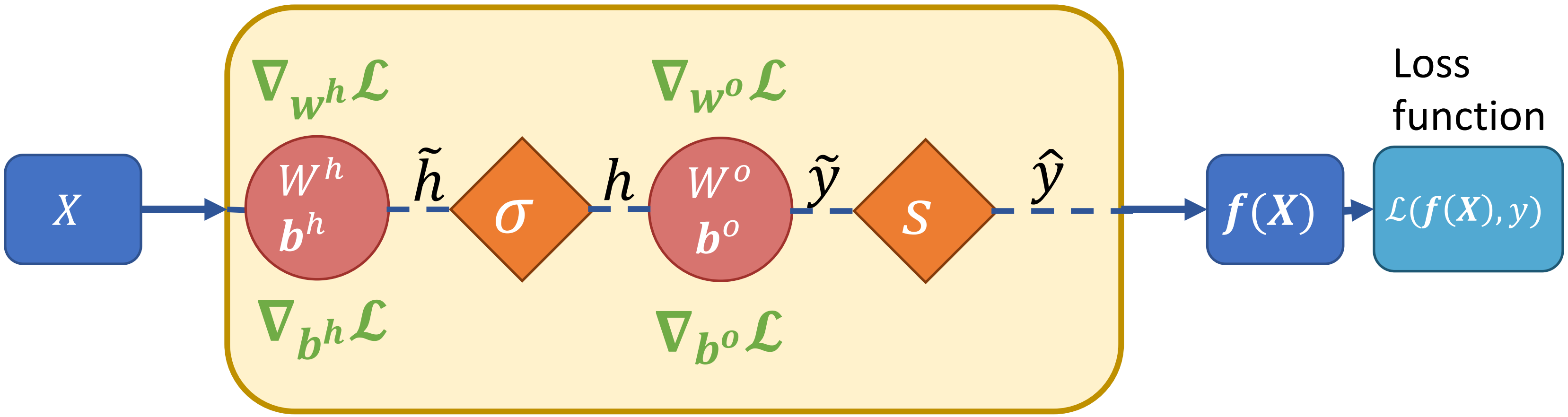
$$\frac{\partial \mathcal{L}}{\partial \tilde{y}_i} = \sum_j \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}(\boldsymbol{x})_j} \frac{\partial \boldsymbol{f}(\boldsymbol{x})_j}{\partial \tilde{y}_i}$$

$$= \sum_j \frac{-1_{y=j}}{\partial \boldsymbol{f}(\boldsymbol{x})_y} \frac{\partial softmax(\tilde{y})_j}{\partial \tilde{y}_i}$$

$$= \begin{cases} \dfrac{-1}{\boldsymbol{f}(\boldsymbol{x})_y} softmax(\tilde{y})_y \left(1 - softmax(\tilde{y})_y\right) & if\ i = y \\ \dfrac{1}{\boldsymbol{f}(\boldsymbol{x})_y} softmax(\tilde{y})_y softmax(\tilde{y})_i & if\ i \neq y \end{cases}$$

$$= \begin{cases} -1 + \boldsymbol{f}(\boldsymbol{x})_y & if\ i = y \\ \boldsymbol{f}(\boldsymbol{x})_i & if\ i \neq y \end{cases}$$

$\nabla_{\tilde{y}} \mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$ with one-hot encoding of the target

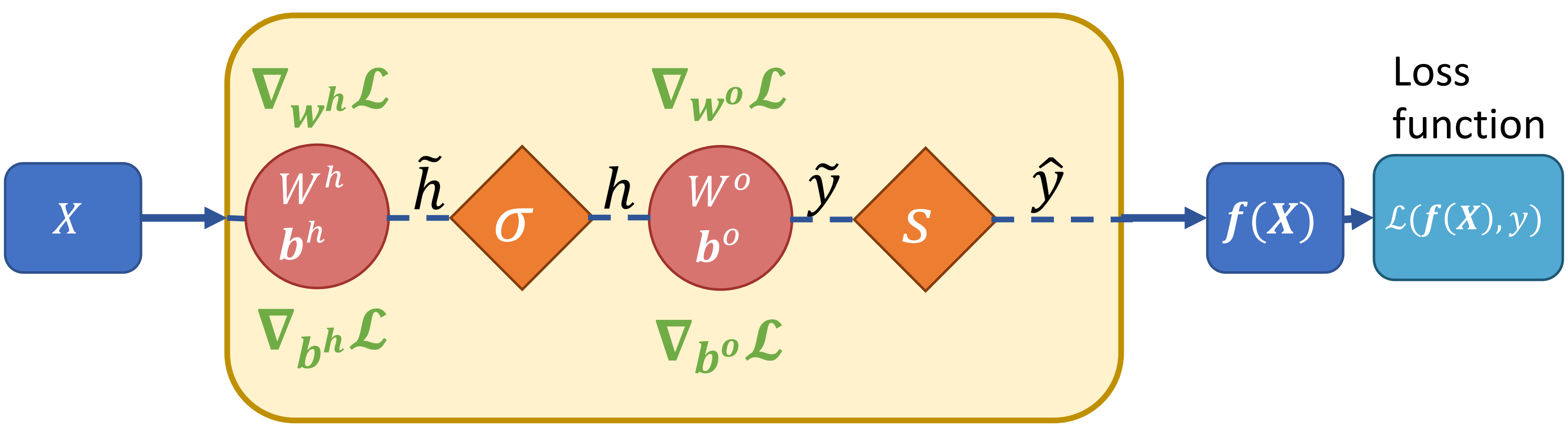


$$\nabla_{\tilde{y}}\mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$$

$$\nabla_{\boldsymbol{b}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L} \cdot \boldsymbol{h}^T$$

$$\tilde{\boldsymbol{y}} = \boldsymbol{W}^o\boldsymbol{h} + \boldsymbol{b}^o$$



Backpropagation

$$\nabla_{\tilde{y}}\mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$$

$$\nabla_{\boldsymbol{b}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

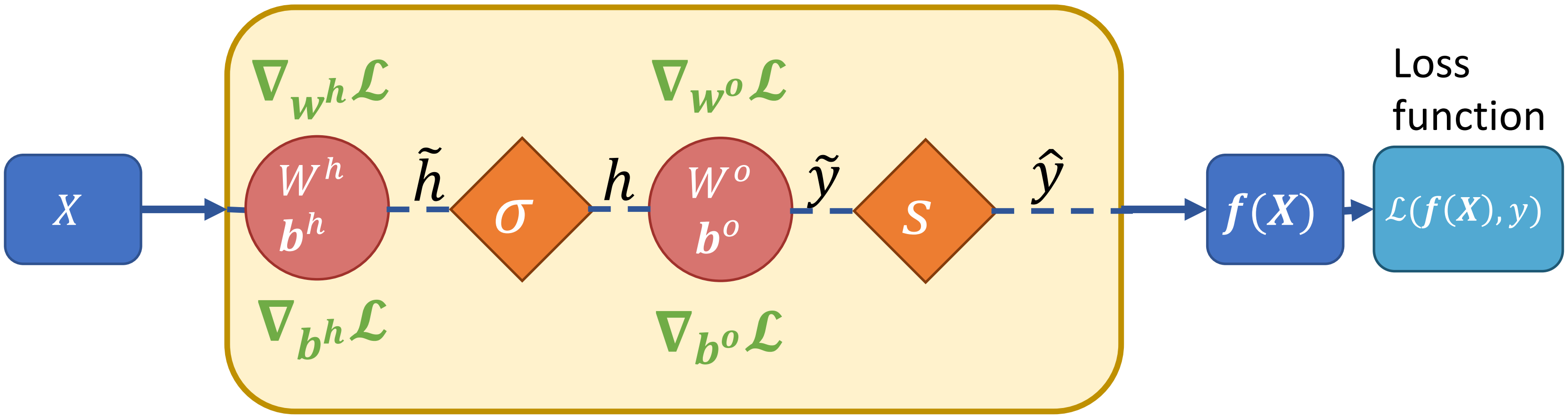
$$\nabla_{\boldsymbol{W}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L} \cdot \boldsymbol{h}^T$$

$$\nabla_{\boldsymbol{h}}\mathcal{L} = \boldsymbol{W^o}^{\boldsymbol{T}} \cdot \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

$$\nabla_{\tilde{h}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L} \odot \sigma'(\tilde{\boldsymbol{h}})$$

$\odot$ is the element-wise multiplication (Hadamard product).

Sigmoid is applied element-wise, thus the gradient also.



Backpropagation

$$\nabla_{\tilde{y}}\mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$$

$$\nabla_{\boldsymbol{b}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

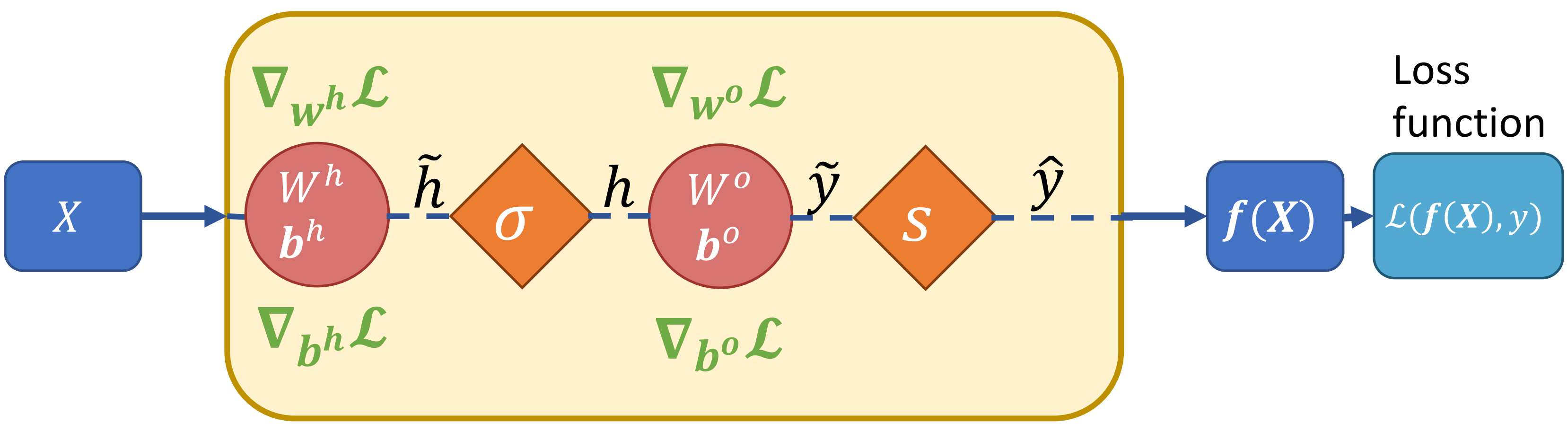
$$\nabla_{\boldsymbol{W}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L} \cdot \boldsymbol{h}^T$$

$$\nabla_{\boldsymbol{h}}\mathcal{L} = \boldsymbol{W}^{o\boldsymbol{T}} \cdot \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

$$\nabla_{\tilde{h}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L} \odot \sigma'(\tilde{\boldsymbol{h}})$$

$$\nabla_{\boldsymbol{b}^h}\mathcal{L} = ?$$

$$\nabla_{\boldsymbol{W}^h}\mathcal{L} = ?$$



Backpropagation

$$\nabla_{\tilde{y}}\mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$$

$$\nabla_{\boldsymbol{b}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

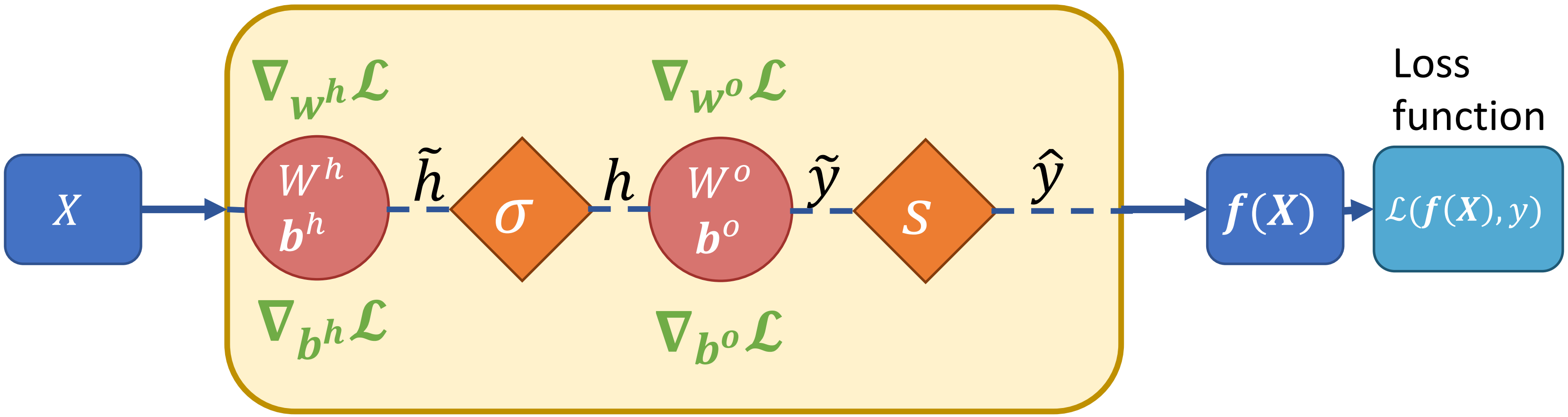
$$\nabla_{\boldsymbol{W}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L} \cdot \boldsymbol{h}^T$$

$$\nabla_{\boldsymbol{h}}\mathcal{L} = \boldsymbol{W}^{o\boldsymbol{T}} \cdot \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

$$\nabla_{\tilde{h}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L} \odot \sigma'(\tilde{\boldsymbol{h}})$$

$$\nabla_{\boldsymbol{b}^h}\mathcal{L} = \nabla_{\tilde{h}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}^h}\mathcal{L} = \nabla_{\tilde{h}}\mathcal{L} \cdot \boldsymbol{x}^T$$



Backpropagation

$$\nabla_{\tilde{y}}\mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$$

$$\nabla_{\boldsymbol{b}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L} \cdot \boldsymbol{h}^T$$

$$\nabla_{\boldsymbol{h}}\mathcal{L} = \boldsymbol{W^o}^{\boldsymbol{T}} \cdot \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

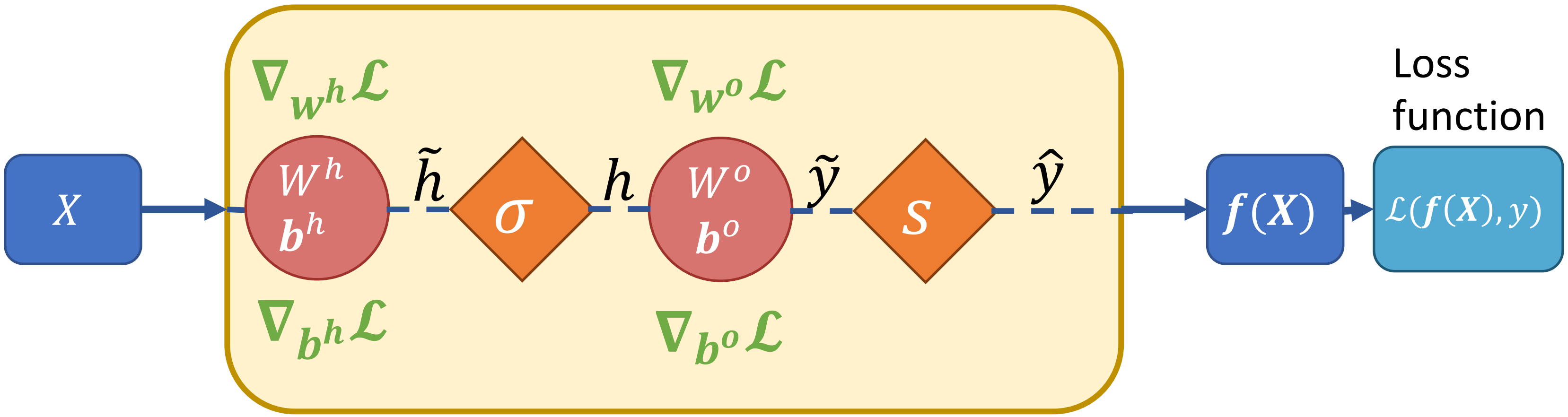
$$\nabla_{\tilde{h}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L} \odot \sigma'(\tilde{\boldsymbol{h}})$$

$$\nabla_{\boldsymbol{b}^h}\mathcal{L} = \nabla_{\tilde{h}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}^h}\mathcal{L} = \nabla_{\tilde{h}}\mathcal{L} \cdot \boldsymbol{x}^T$$

When in doubt, look at the shape.

$\nabla_W\mathcal{L}$ must have the shape of W because of the update rule W $\leftarrow W - \nabla_W\mathcal{L}$



Backpropagation

$$\nabla_{\tilde{y}}\mathcal{L} = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{e}(y)$$

$$\nabla_{\boldsymbol{b}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}^o}\mathcal{L} = \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L} \cdot \boldsymbol{h}^T$$

$$\nabla_{\boldsymbol{h}}\mathcal{L} = \boldsymbol{W}^{o\boldsymbol{T}} \cdot \boldsymbol{\nabla}_{\tilde{y}}\mathcal{L}$$

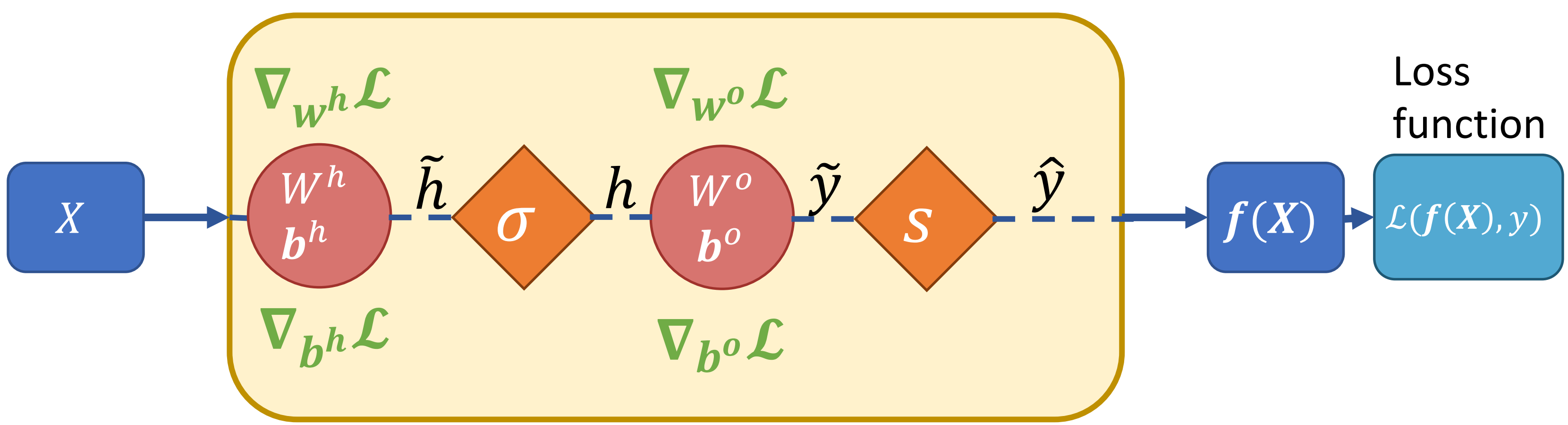
$$\nabla_{\tilde{h}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L} \odot \sigma'(\tilde{\boldsymbol{h}})$$

$$\nabla_{\boldsymbol{b}^h}\mathcal{L} = \nabla_{\tilde{h}}\mathcal{L}$$

$$\nabla_{\boldsymbol{W}^h}\mathcal{L} = \nabla_{\tilde{h}}\mathcal{L} \cdot \boldsymbol{x}^{\boldsymbol{T}}$$

To have huge speed-up:

1. Re-use previous gradients
2. Save tensors during forward

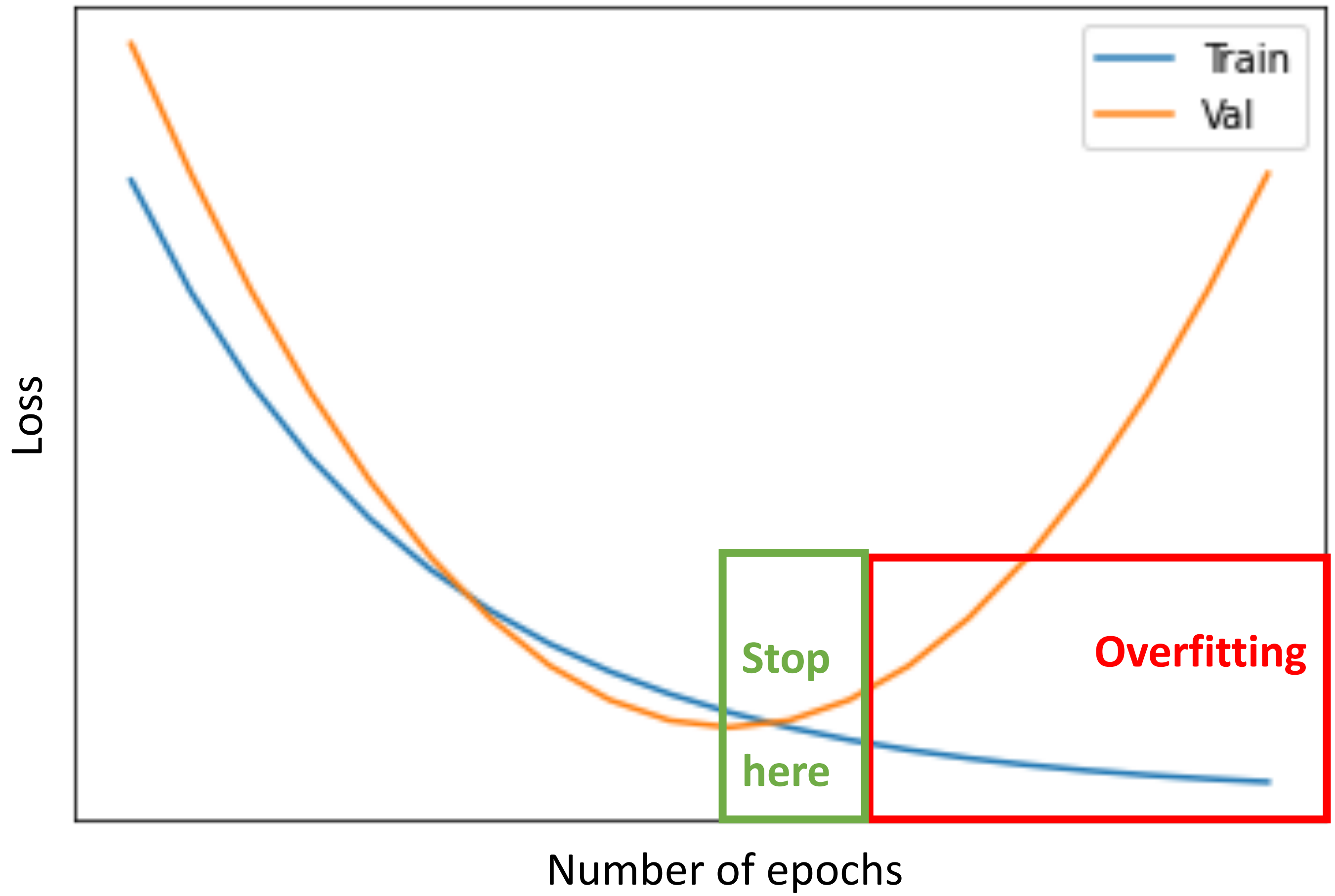


Tips & Tricks

When to Stop?

Split data in **train / val / test**

Stop when a criterion (loss, accuracy, f1, etc.) stop improving on **validation set**



(Mini-)Batch Size

Batch Gradient Descent: one forward & backward on the whole dataset
→ Better gradient estimation
→ GPU parallelism
→ Impracticable to fit large dataset in VRAM

Stochastic Gradient Descent: one forward & backward per sample
→ Easy to fit in VRAM
→ Add noise that may improve generalization
→ Add too much noise
→ Slow

Mini-Batch Gradient Descent: one forward & backward per group of samples
→ Trade-off between both
→ Learning rate should be proportional to batch size, e.g. batch size 32->64, lr 0.1->0.2

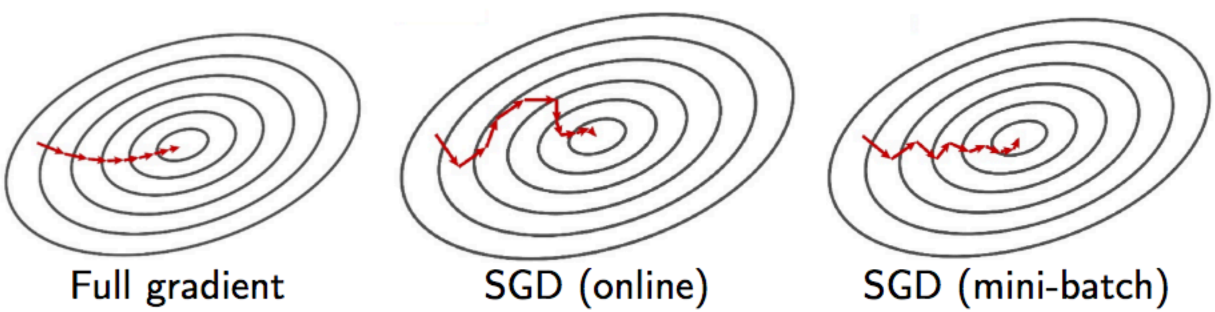


Image from [Matthieu Cord]

(Mini-)Batch Size

A fully connected layer in forward pass was:

$$\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

Considering that $\boldsymbol{x} \in \mathbb{R}^{n}$ was the features of a unique sample.

But with a batch size $\boldsymbol{X} \in \mathbb{R}^{b \times n}$, thus:

$$\boldsymbol{h} = \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{T}} + \boldsymbol{b}$$

With $\boldsymbol{W}$ defined as before.

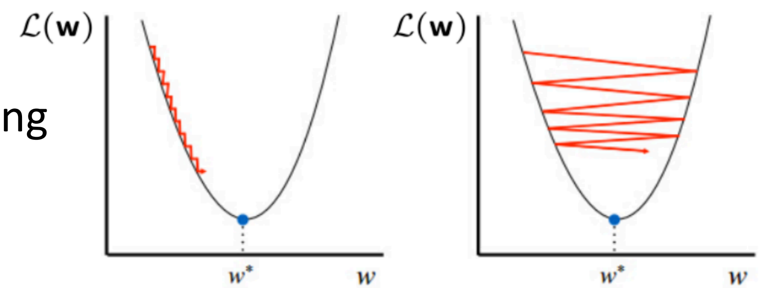
Learning Rate

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$$

Controls the rate of change.

Too high:
→ Cannot converge, but diverge, reduce overfitting

Too low:
→ Super slow, stuck in bad local minima



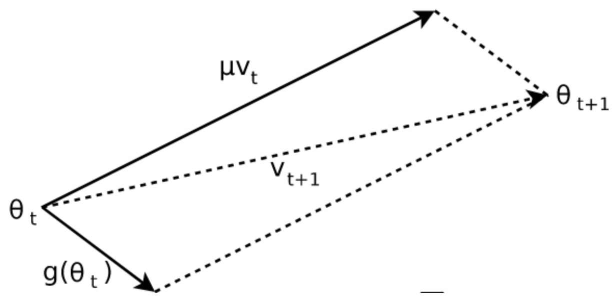
Too small: converge very slowly

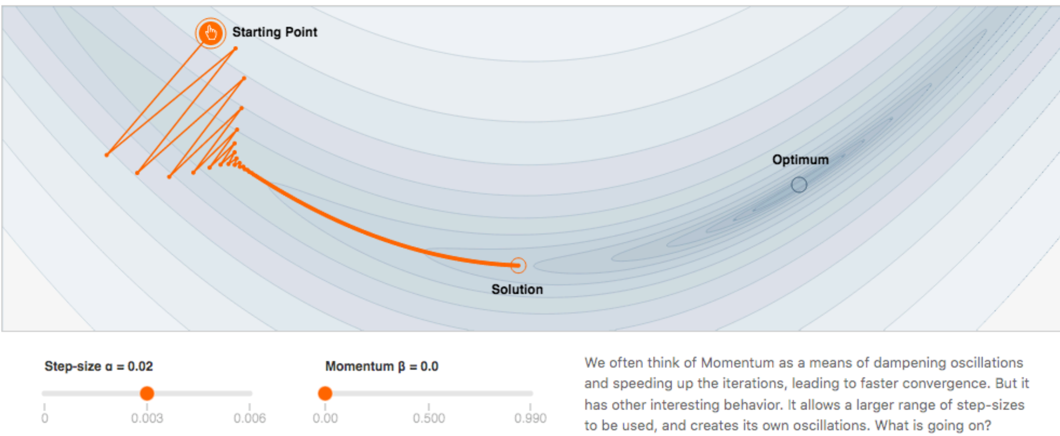
Too big: overshoot and even diverge

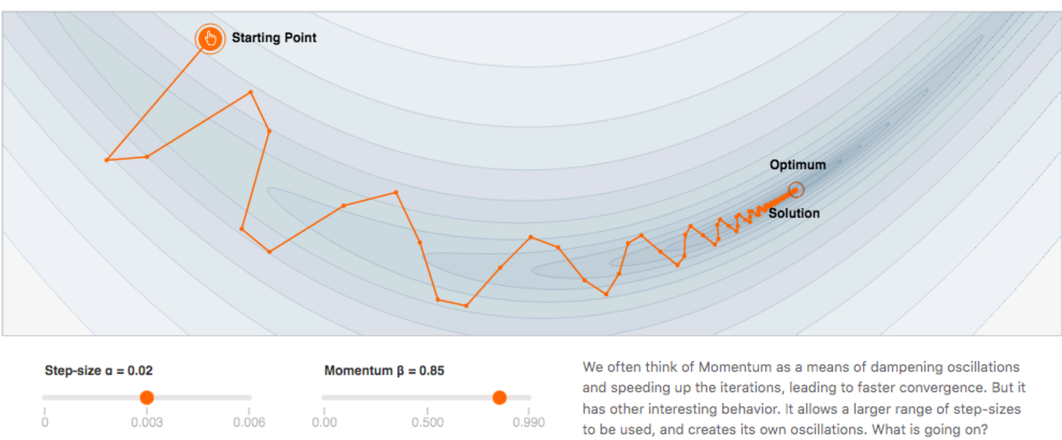
Tip: Start with high learning rate, and decreases it through time

Image from Matthieu Cord

SGD with Momentum







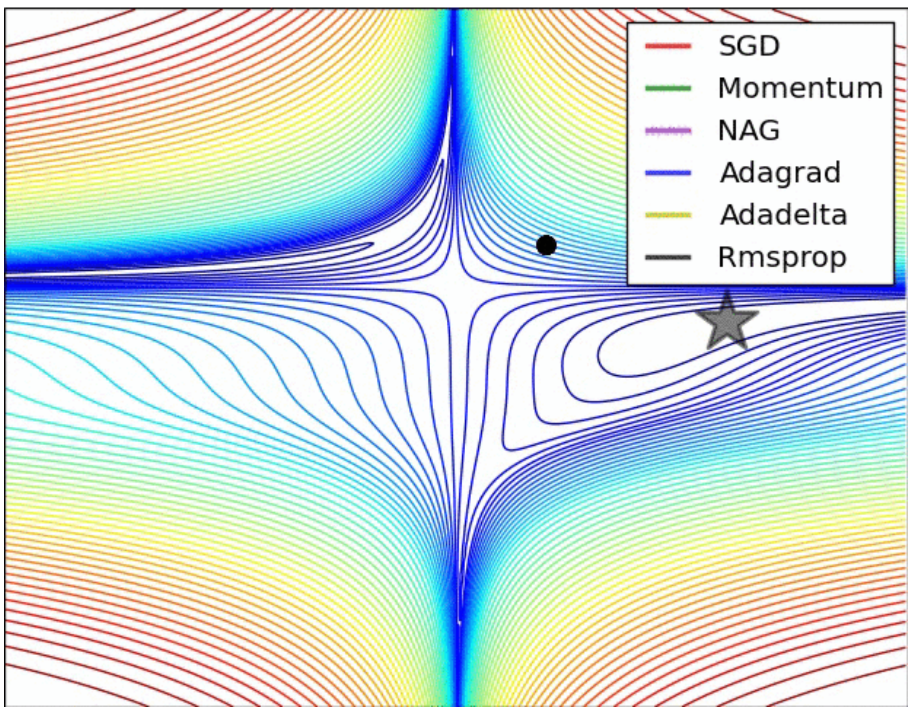
$$v \leftarrow \alpha v - (1 - \alpha)\nabla_\theta \mathcal{L}$$
$$\theta \leftarrow \theta + v$$

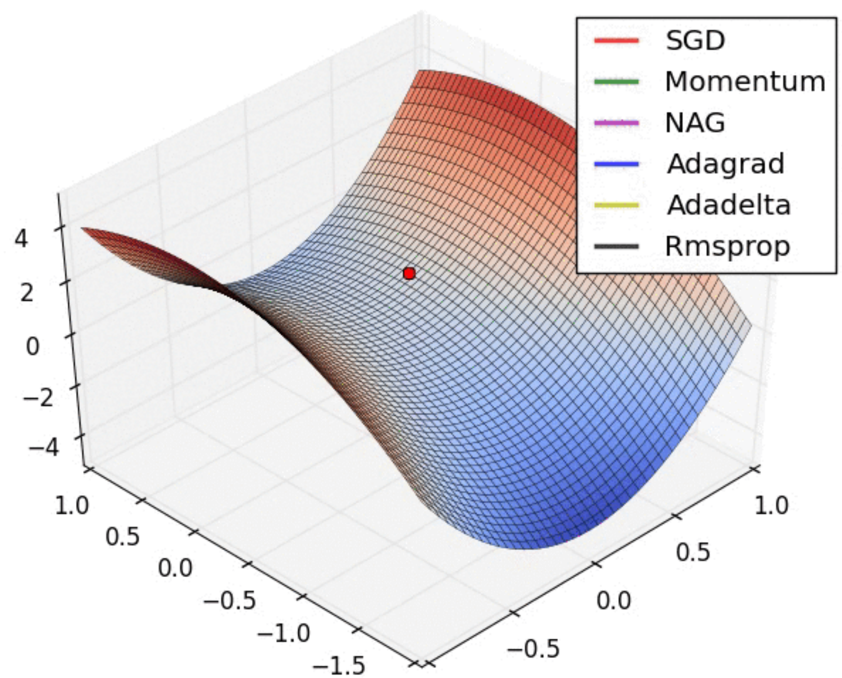
Why Momentum Really works, on distill.pub

Optimizers

Modern optimizers have an **adaptive learning rate** per parameter based on gradient statistics.
→ Especially useful on saddle point
→ The most famous is Adam

But a well-tuned SGD with momentum can be the best [Wilson et al. 2017].

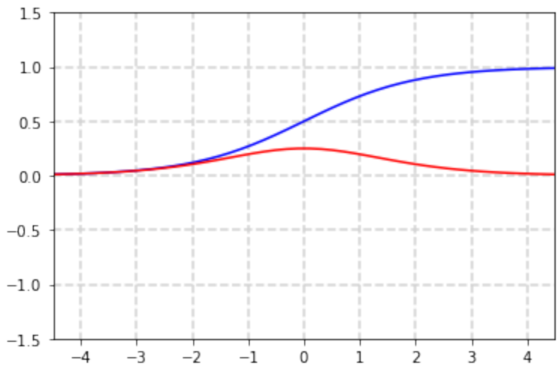




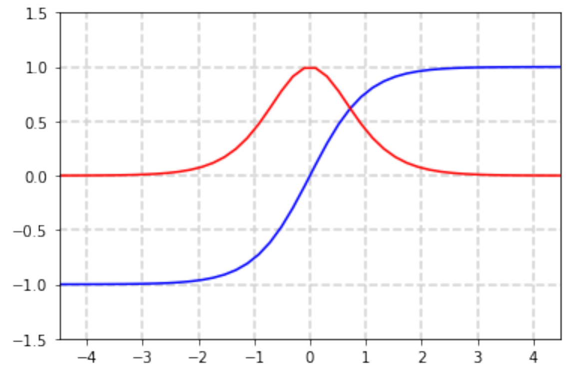
Great overview of gradient-descent based optimizers by Ruder.

Pitfalls of hidden activations

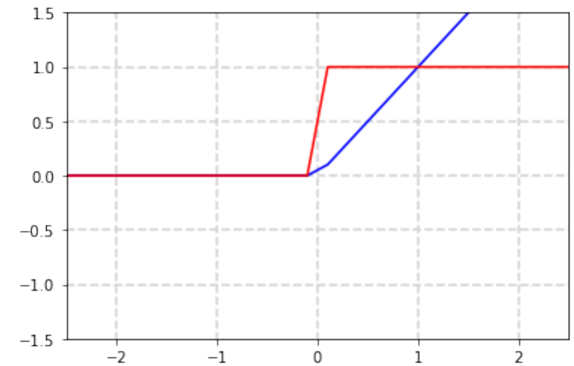
Function and their **derivative**



Sigmoid



Tanh



ReLU

Sigmoid and tanh **saturates** at small and large values.
→ Gradient is zero, no learning
→ Avoid these old-school activations

ReLU is zero if $x \leq 0$:
→ **Dying neurons** with zero-output and thus zero-gradient
→ If it happens, use a **Leaky ReLU** $LReLU(x) = \begin{cases} x & if\ x > 0 \\ \epsilon\, x & otherwise \end{cases}$

Initialization

Initializing the biases $\boldsymbol{b}^h$ and $\boldsymbol{b}^o$ to very small values.
→ Helpful to avoid dying neurons with ReLU

Initializing the weights $\boldsymbol{W}^h$ and $\boldsymbol{W}^o$ to:

- **Zero-weights**
  → No learning because gradient w.r.t input is also zero
- **Constant weights**
  → Symmetry where two hidden neurons are connected to the same inputs, they learn the same pattern!
- **Large values**
  → Risk gradient explosion
- **He / Glorot initialization**
  → Normalize weights to avoid explosion with large number of outgoing connections

Multiple Losses & Regularizations

Main loss Regularization

$$\mathcal{L} = \mathcal{L}_1(x, y) + \lambda \mathcal{L}_2(x, y) + \beta \mathcal{L}(\theta)$$

Auxiliary loss

You can combine multiple losses, each has been batch averaged.

Main loss (usually the classification loss) has a factor of 1.

Auxiliary losses have a factor as hyperparameter
- Need to find optimal through cross-validation
- Should often ensure that losses are more or less in the same values

Regularization losses usually only takes the parameters as argument
- Most common is **weight decay** $\beta \sum_i ||\theta_i||^2$
- Often useful to put a prior on the parameters

Small break,
then coding session!