

Data Science & Machine Learning Lab

Introduction to Deep Learning

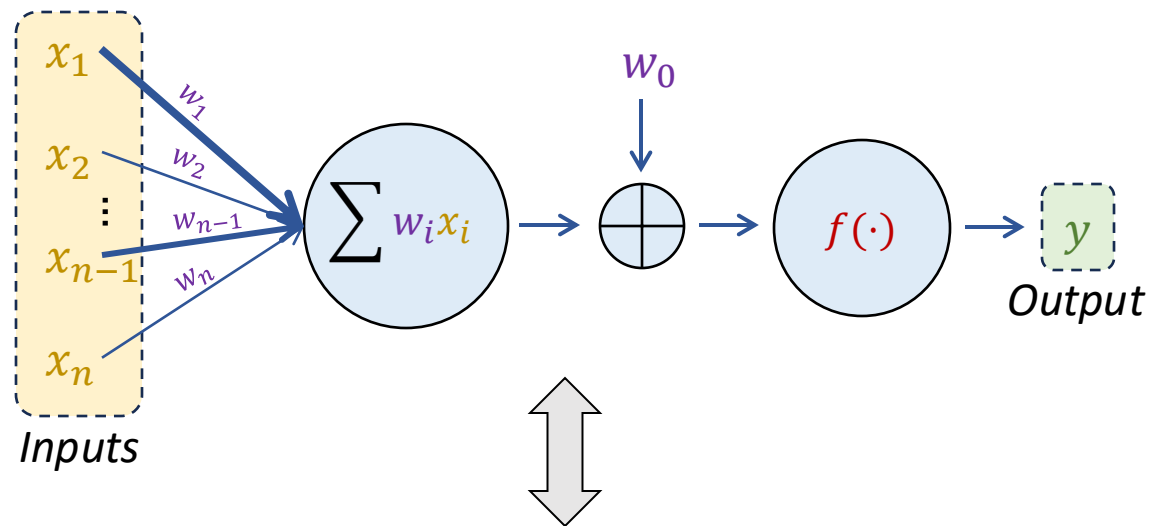
Flavio Giobergia



The perceptron

- The perceptron is the simplest unit of neural networks
- It takes an input with *multiple features*, and does the following:
 - It weights each input feature with a given *weight*,
 - It produces a weighted sum of the inputs, and
 - It applies a *function* to the output
- $y = f(w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n)$

The perceptron



Or, in other words, $y = f\left(\sum_{i=0}^n w_i x_i\right) = f(\mathbf{w}^T \mathbf{x})$ and $x_0 = 1$

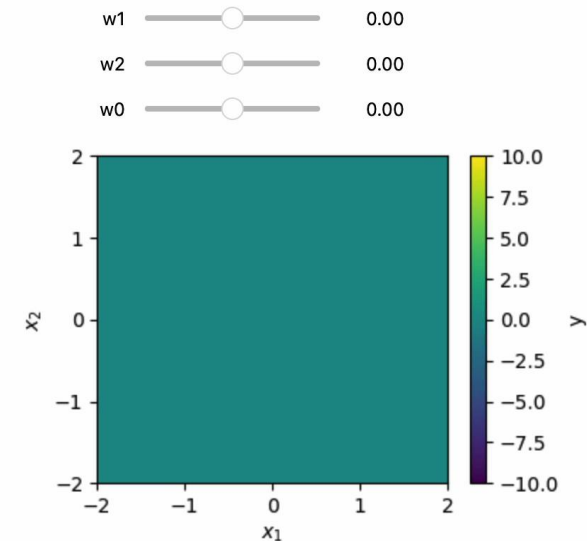
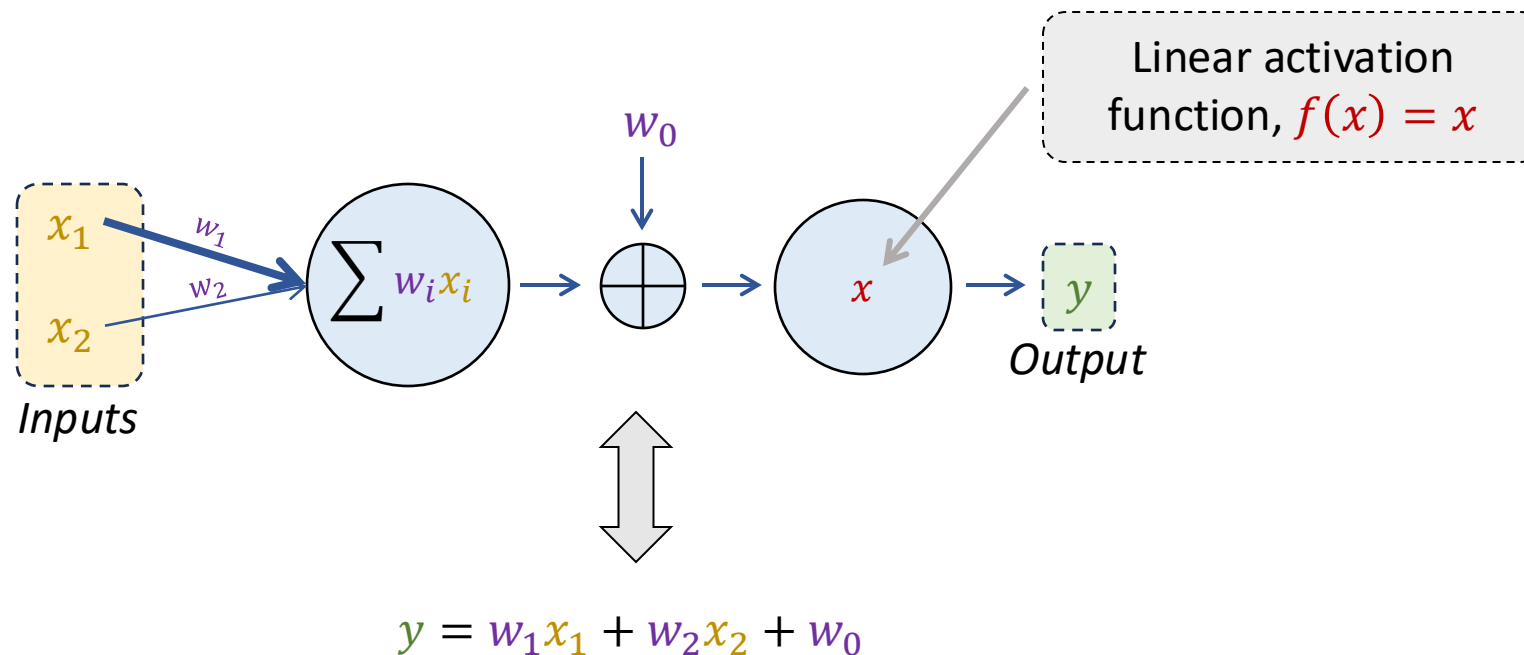
- $\mathbf{x} = (x_1, x_2, \dots, x_n)$ is the input sample
- y represents the output of the perceptron.
- $f(\cdot)$ represents a non-linear “activation” function
- w_i (and w_0) are weights (and bias), which are “learned”

Note

With the exception of $f(\cdot)$, this looks like the classic *linear regression*

And if $f(\cdot) = \sigma(\cdot)$ (sigmoid function), this looks like the (just as classic) *logistic regression*

The perceptron, in 2D



The perceptron can be used to represent a family of functions,
 $y = w_1 x_1 + w_2 x_2 + w_0$

Various values of w_0, w_1, w_2 define the different functions that can be learned by the perceptron.

Activation functions

- Activation functions are used for two *main* reasons:

1. Enforce *properties* on perceptron's output

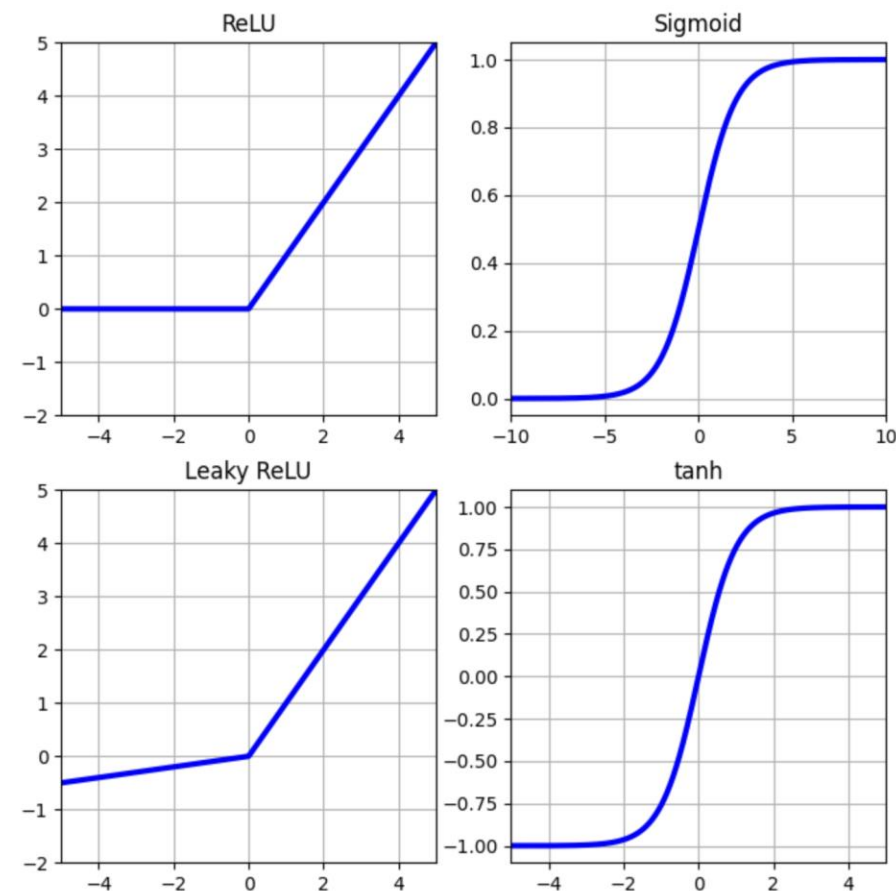
- E.g., sigmoid → binds output to $[0, 1]$ range

2. Introduce *non-linearities* in the model

+ some others (faster convergence, sparsity, ...)

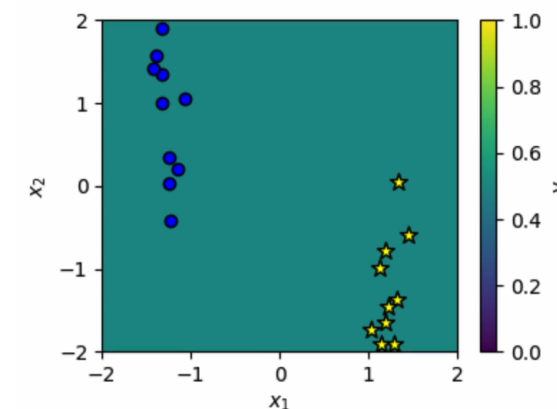
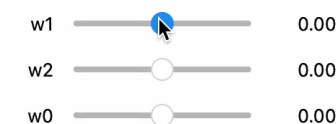
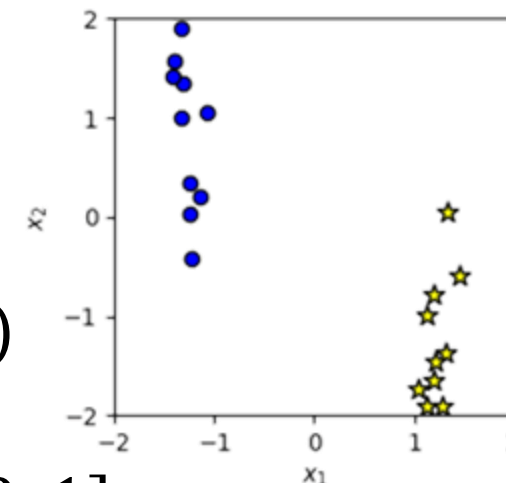
- Commonly adopted functions:

- ReLU
- Sigmoid
- Leaky ReLU
- Tanh
- Softmax
- Linear
- GeLU

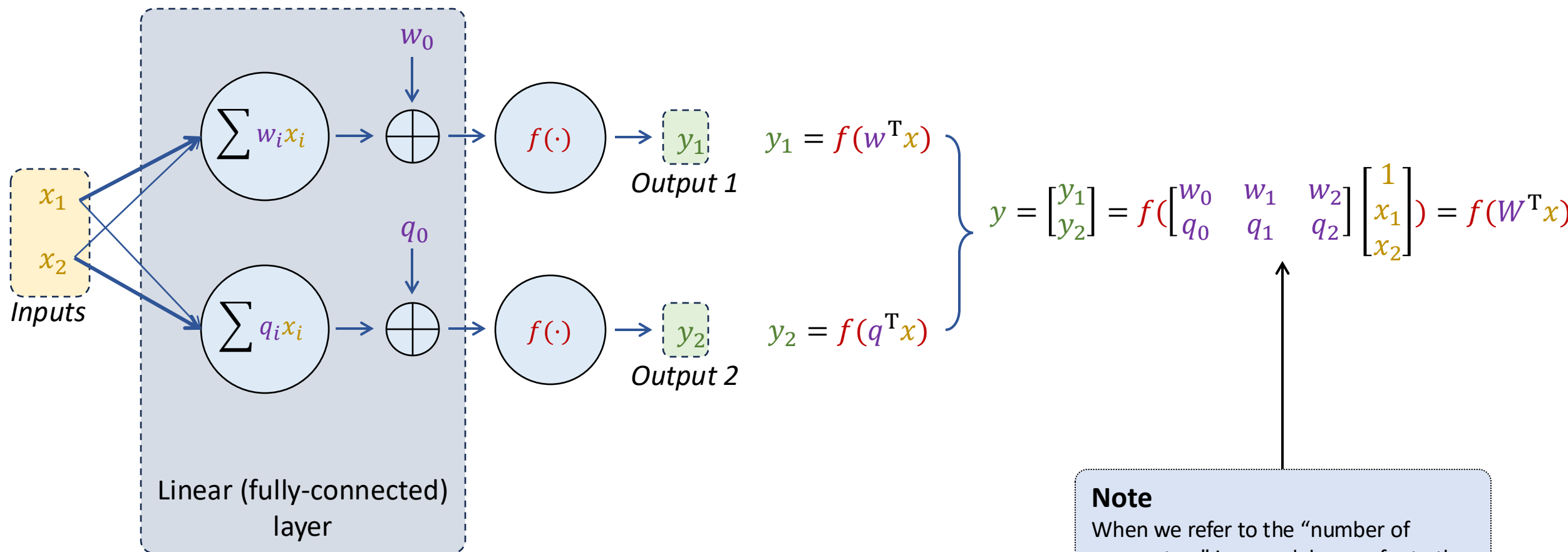


1. Enforce *properties* on perceptron's output

- Binary classification problem
 - Separate positive (★) and negative (●) samples
- For a point $x \in \mathbb{R}^2$, the perceptron can predict $p(\star | x)$
 - For the binary case, this implies $p(\bullet | x) = 1 - p(\star | x)$
- To get a valid probability, we must enforce $p(\star | x) \in [0, 1]$
 - We already have $p(\bullet | x) + p(\star | x) = 1$ by construction
- The Sigmoid maps any value in $\mathbb{R}$ to the range $[0, 1]$
 - i.e., the perceptron's output (in $\mathbb{R}$) is squashed to $[0, 1]$
 - $\sigma(x) = \frac{1}{1+e^{-x}}$



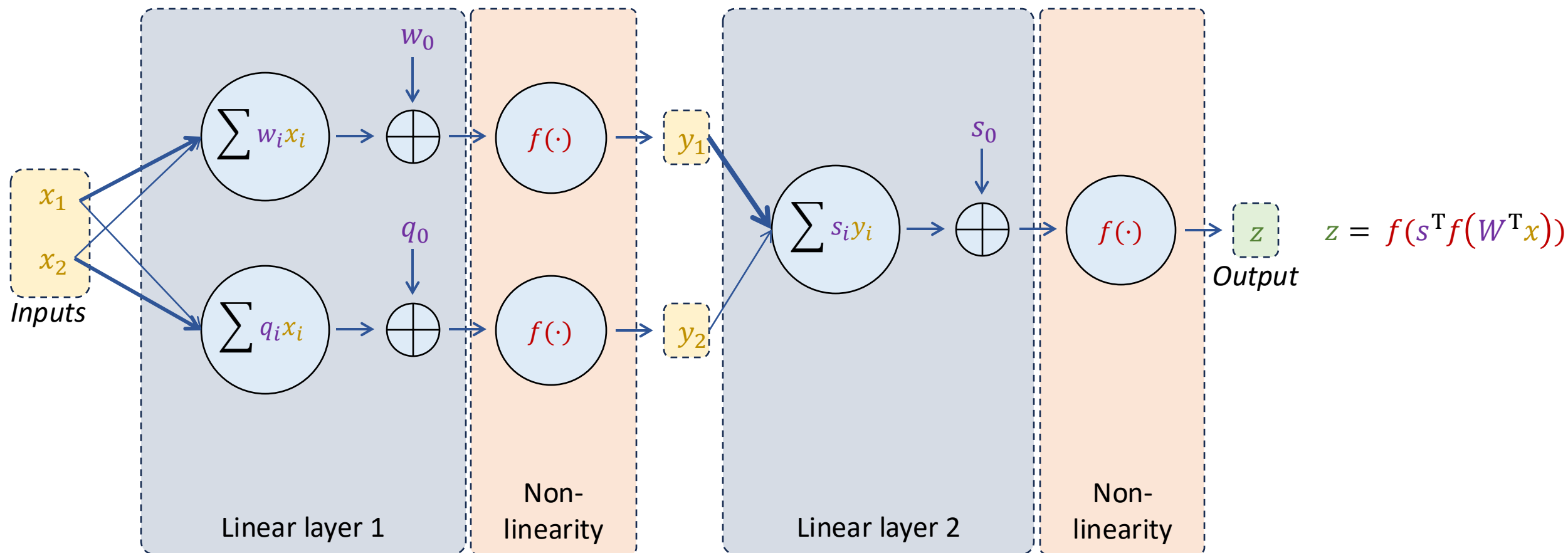
Adding some perceptrons



Note

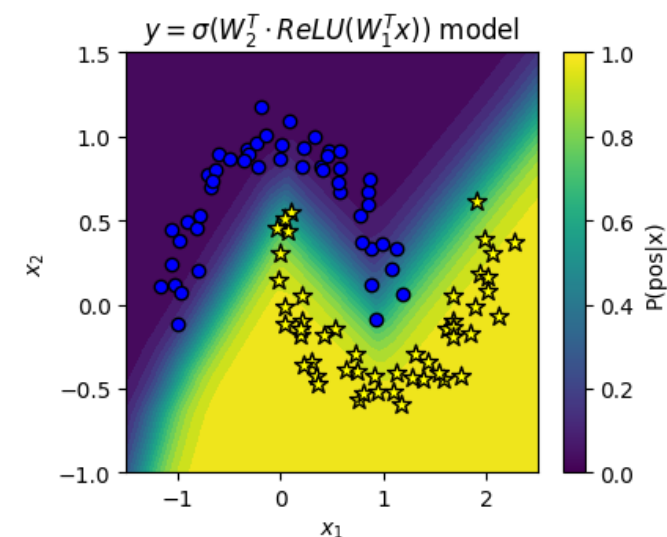
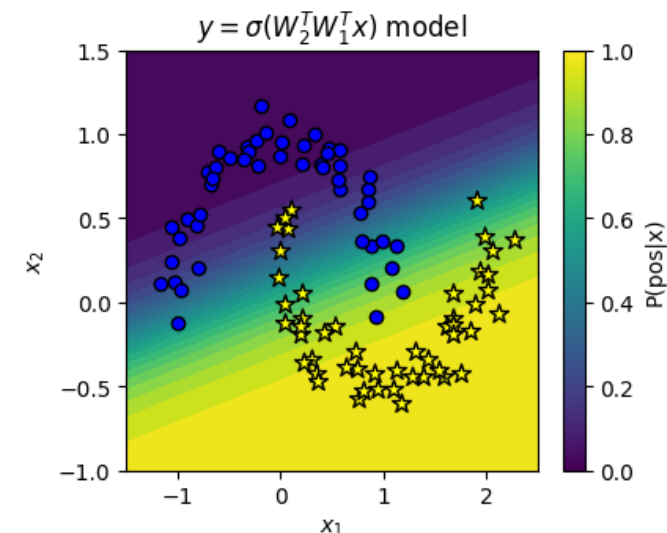
When we refer to the “number of parameters” in a model, we refer to the total number of weights the model has. This is a “6 parameters” model!

and adding other layers!



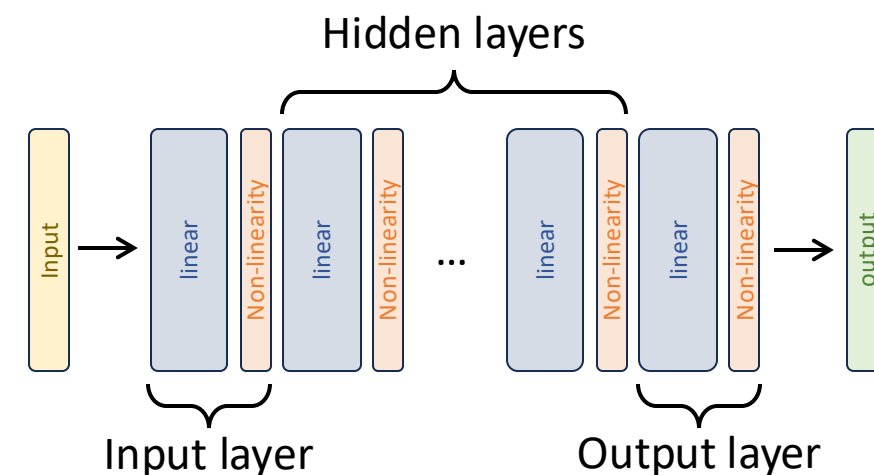
2. Introduce *non-linearities* in the model

- if $f(x) = x$ (i.e., no non-linearity is added), we get
 $z = s^T W^T x$
- This implies:
 1. We could have used $W' = Ws$ and get the same output
 2. We wouldn't have needed a second layer!
 3. But our model is still linear
- So, we use non-linear activation functions to model more complex functions



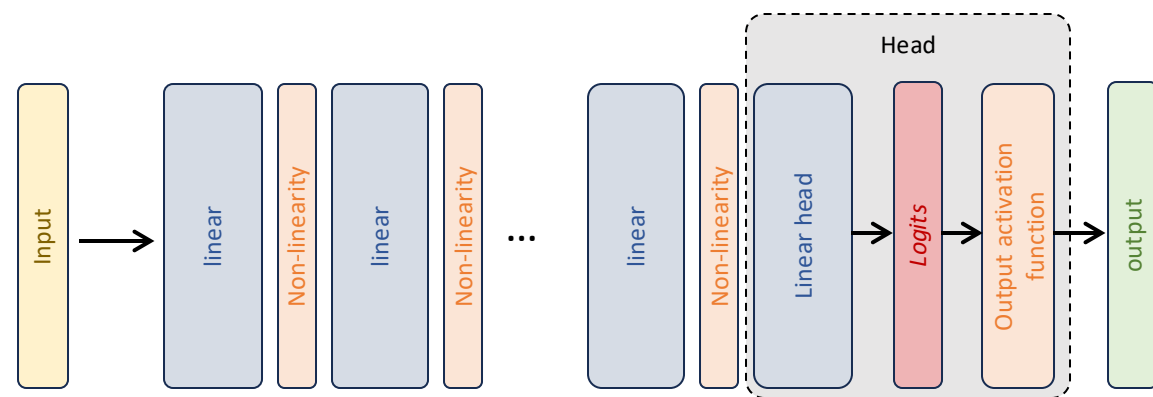
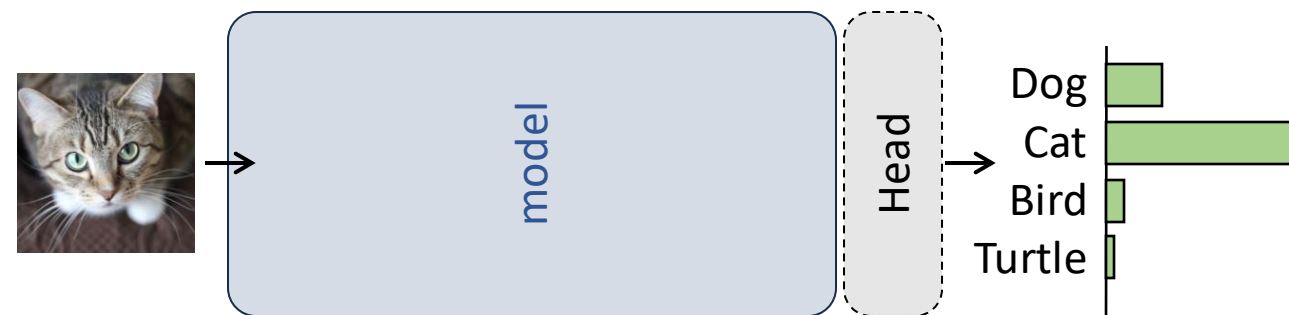
Multi-layer perceptron models

- We can stack additional *layers*
 - separated by *non-linearities* (activation functions) to prevent collapses
- *Universal Approximation Theorem* tells us that we can approximate “any” function with MLPs
 - “For any continuous function g defined on a compact subset of $\mathbb{R}^n$ and for any $\epsilon > 0$, there exists a feedforward neural network with a single hidden layer and a finite number of neurons that can approximate g to within an arbitrary degree of accuracy ϵ ”
 - A single-layer MLP works ... but no information on the number of neurons, or the weights’ values!
 - *Deeper, narrower* networks are generally used

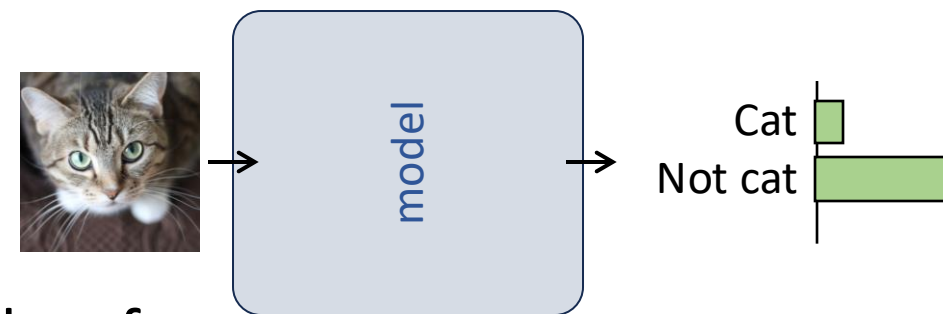


Activation functions for classification models

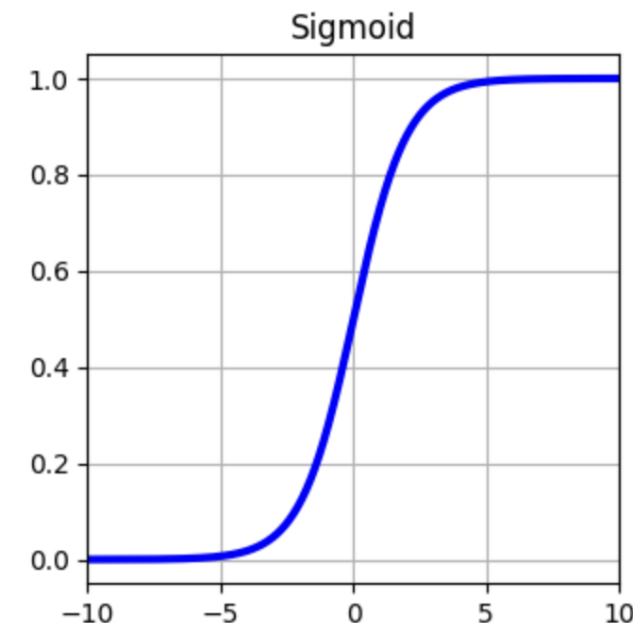
- As argued, activation functions can be used to enforce properties on the model's output
- In classification problems, the output *before* the final activation is treated as *unnormalized probabilities (logits)*
- We still need a step to convert *logits* into *valid* probabilities
 - i.e., all probabilities should sum to 1, and be in $[0, 1]$



Binary classification

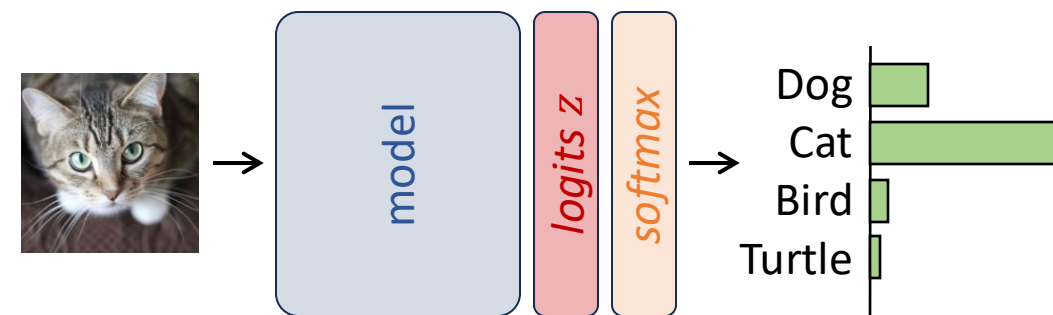


- The model predicts the probability of a single class for point x
 - As a convention, the *positive* one $P(pos|x)$ (e.g., $P(cat|x)$)
- The model produces a logit $z = model(x)$
- We use the **sigmoid function** on the output logit z
 - $\sigma(z) = \frac{1}{1+e^{-z}}$
 - This guarantees $P(pos|x) \in [0, 1]$ ✓
- We work out the probability of the negative class (e.g., “not cat”)
 - $P(neg|x) = 1 - P(pos|x)$
 - We can easily show that $P(neg|x) \in [0,1]$ ✓
- By construction, $P(pos|x) + P(neg|x) = 1$ ✓



Multi-class classification

- The output class is one of many $(c_1, c_2, \dots, c_n)$
 - E.g., (dog, cat, bird, turtle)
- The model produces a vector of n logits for a point x
 - (i.e., the last layer will have n perceptrons)
 - $z = (z_1, z_2, \dots, z_n) = \text{model}(x)$
- We need to obtain, from the logits, valid probabilities
 - $P(c_1|x), P(c_2|x), \dots, P(c_n|x)$
- The **softmax** function is applied:
 - $P(c_i|x) = \frac{e^{z_i}}{\sum_j e^{z_j}}$
- It can be easily shown that:
 - $P(c_i|x) \in [0, 1]$ ✓
 - $\sum_i P(c_i|x) = 1$ ✓



Activation functions for regression models

- In regression, models generally predict real numbers
- Typically, there is no need to enforce properties
- Output activation function can be the identity function
 - $f(x) = x$
 - Generally the only situation where it makes sense to use it!

Defining weights (parameters)

- So far, we assumed all weights and biases (let's call them θ) to be known
 - *But, we still need to figure out how we find them!*
- We pick a function (objective, or loss), $\mathcal{L}(\theta)$, that we want to minimize
 - e.g., in Linear Regression we minimize the Mean Squared Error
 - $\mathcal{L}(\theta) = MSE(\theta) = \frac{1}{n} \sum (y_i - \theta^T x_i)^2$
 - Then, we pick θ that minimizes it

Note

$\mathcal{L}$ also depends on the training points x_i, y_i , so we should refer to it as $\mathcal{L}(\theta, X, y)$.

However, the training set X, y is generally fixed. Thus, we only have control over θ , so we use the notation $\mathcal{L}(\theta)$.

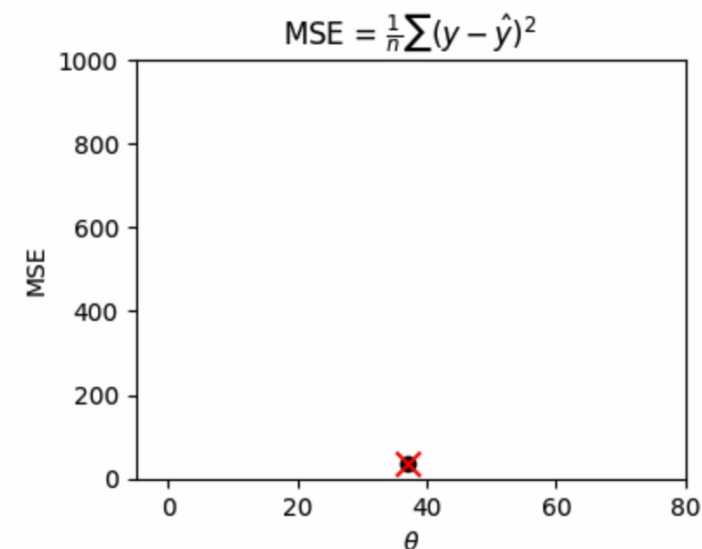
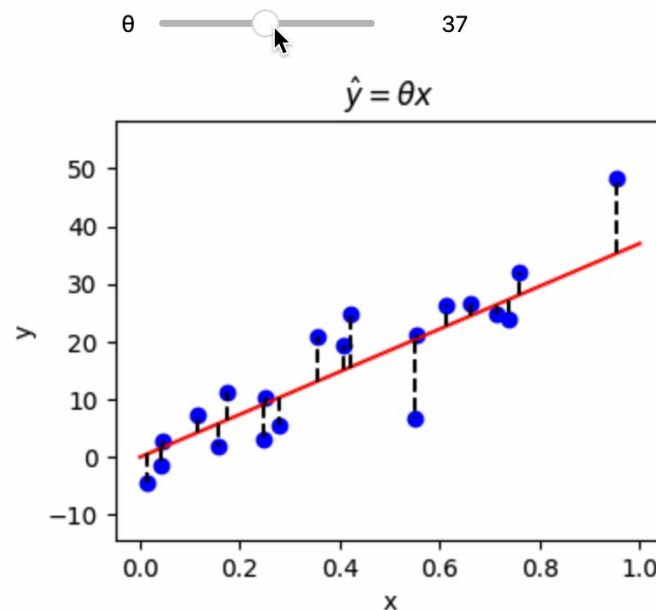
Linear regression

- For *simple* models, we can find the optimal weights in closed form
 - $\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \frac{\partial \text{MSE}(\theta)}{\partial \theta} = 0$
 - Quadratic in θ , can be solved easily!
- Or, we can evaluate the loss function for a bunch of θ 's, and find the "best" one

Note

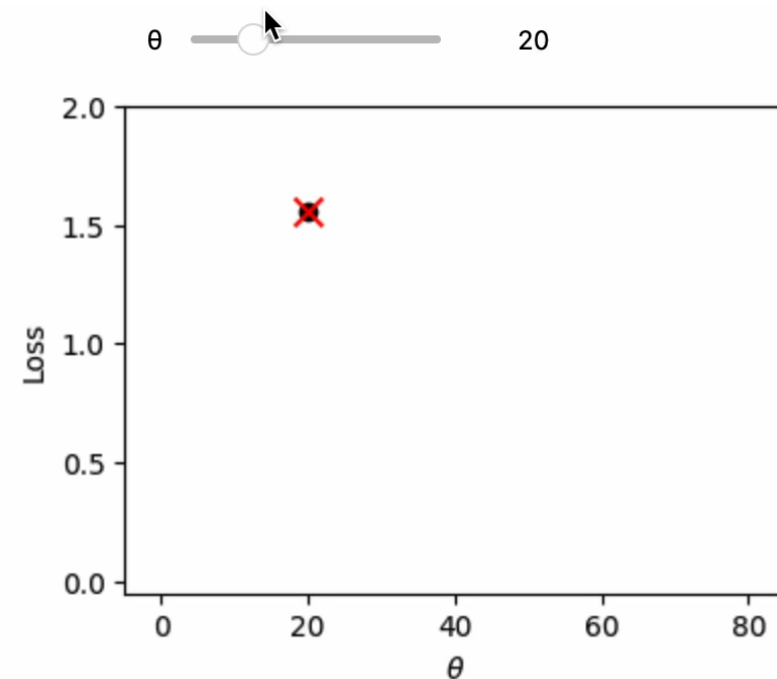
For linear regression, we don't try a bunch of θ since we can easily find the best value in closed form.

However, this provides the intuition for what we will do next with more complex loss functions/models.



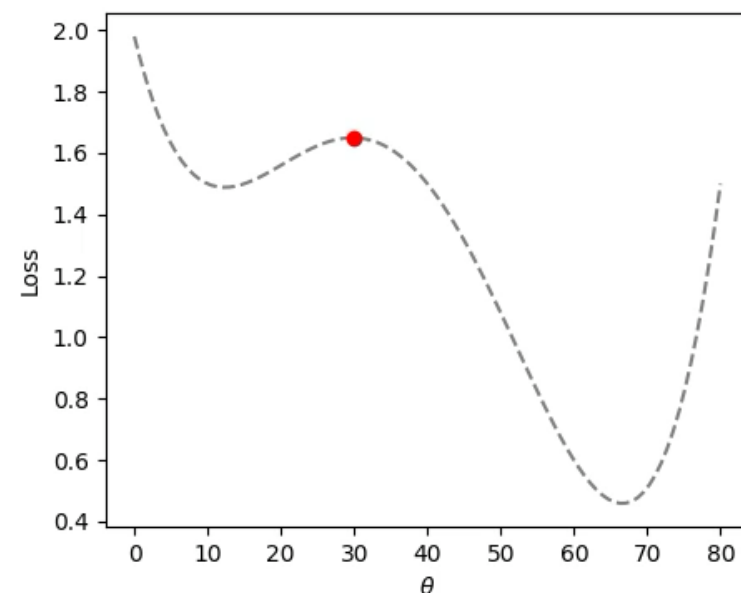
More complex losses/models

- For *more complex* loss functions/models, we may not be able to solve the problem in closed form
 - But we can evaluate $\mathcal{L}(\theta)$ for various values of θ
- We can iteratively update θ to reach a local minimum:
 - We start from a random value θ , then
 - we “move around” according to “some policy”



We “move around” according to “some policy”

- **Move around** = update θ incrementally, based on its current value
 - The new value of θ at any step depends on the previous step's value
 - $\theta_{t+1} := \theta_t + \text{update}$
- **Some policy** = we take a **small** step in the direction where the function decreases locally
 - i.e. in the **opposite** direction of the **gradient**
 - $\theta_{t+1} := \theta_t - \alpha \nabla_{\theta} \mathcal{L}(\theta_t)$
 - for 1-dimensional θ , we have $\theta_{t+1} = \theta_t - \alpha \frac{\partial \mathcal{L}(\theta)}{\partial \theta}$
 - α : learning rate, controls the “size” of the step
- Gradient Descent!

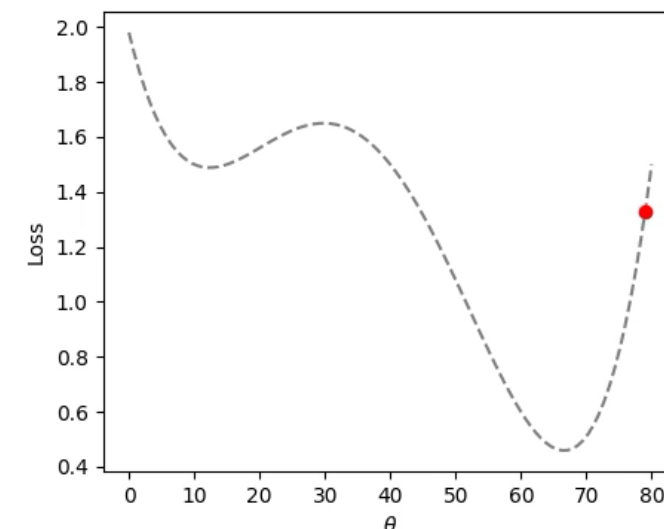
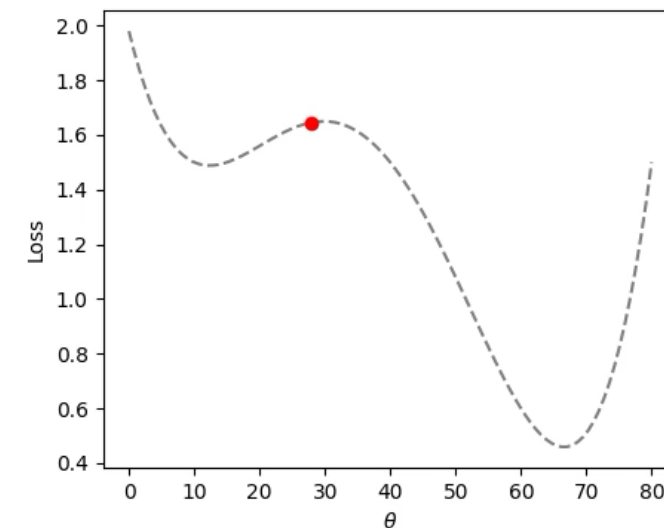


Some limitations of GD

Note

Different initializations will lead to the global minimum for convex loss functions. However, that represents a trivial situation we typically do not encounter.

- GD is sensitive to weight initialization
 - Different initializations can lead to different solutions!
 - GD can get stuck in local minima
- Various solutions to help prevent local minima:
 - Adding momentum
 - Adaptive learning rates
 - Learning rate schedules



Backpropagation

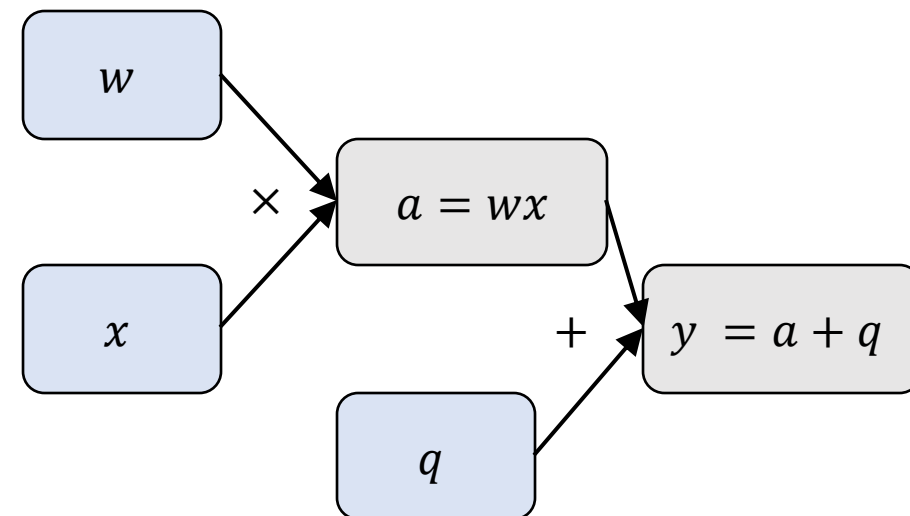
- So far, we assumed we were able to compute $\nabla_{\theta} \mathcal{L}(\theta)$
- However, any loss/model combination would need a different gradient computation!
- We can use backpropagation to compute the gradient of the loss w.r.t. any weight!
 - Backpropagation is just a fancy word for “using the chain rule”

Using the chain rule

- We use the chain rule from calculus, $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$
 - Sometimes known as $f(g(x))' = f'(g(x)) \cdot g'(x)$
- And apply it from the end of the computational graph, backwards
 - (hence the name, backpropagation)

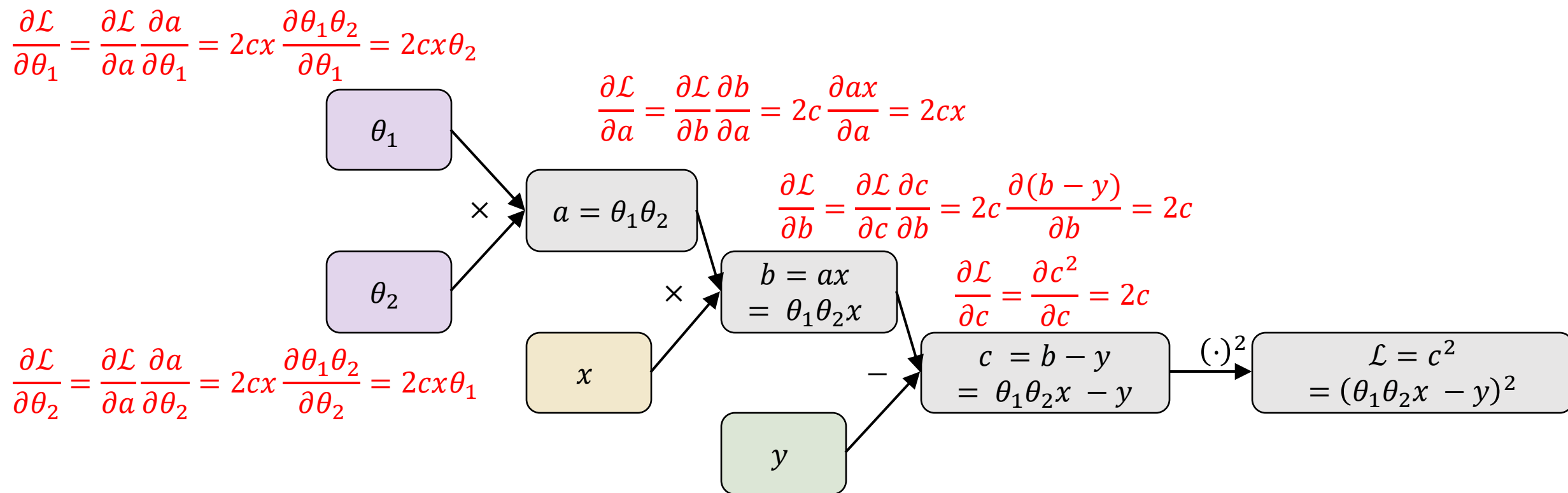
Computational graph

- A computational graph is a directed graph
 - Each node corresponds to an operation
 - Each edge represents the flow of data between nodes
- For instance, we may want to compute $y = wx + q$
 - We start from three variables, w , x and y
 - The computational graph performs one operation at a time
 - First, compute the intermediate variable $a = wx$
 - Then, compute the output variable $z = a + y = wx + y$



Backpropagation example

- Let's say:
 - Our dataset has one point, (x, y)
 - Our (weird) model has two parameters, θ_1 and θ_2 , and predicts $\theta_1\theta_2x$
 - Our loss function will be $\mathcal{L} = (\theta_1\theta_2x - y)^2$
- We build a computational graph with all operations and intermediate variables
 - $a = \theta_1\theta_2$
 - $b = ax = \theta_1\theta_2x$
 - $c = b - y = ax - y = \theta_1\theta_2x - y$
 - $\mathcal{L} = c^2 = (b - y)^2 = (ax - y)^2 = (\theta_1\theta_2x - y)^2$



Forward step

- The loss $\mathcal{L}$ is computed starting from the “inputs” θ_1, θ_2, x, y

Backward step (backpropagation)

- The loss $\mathcal{L}$ is used to compute the derivative w.r.t. $c \rightarrow \frac{\partial \mathcal{L}}{\partial c}$
- The derivative $\frac{\partial \mathcal{L}}{\partial c}$ is used to compute the derivative w.r.t. $b \rightarrow \frac{\partial \mathcal{L}}{\partial b}$
- The derivative $\frac{\partial \mathcal{L}}{\partial b}$ is used to compute the derivative w.r.t. $a \rightarrow \frac{\partial \mathcal{L}}{\partial a}$
- The derivative $\frac{\partial \mathcal{L}}{\partial a}$ is used to compute the derivative w.r.t. $\theta_1, \theta_2 \rightarrow \frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = 2(\theta_1 \theta_2 x - y)x\theta_2$$

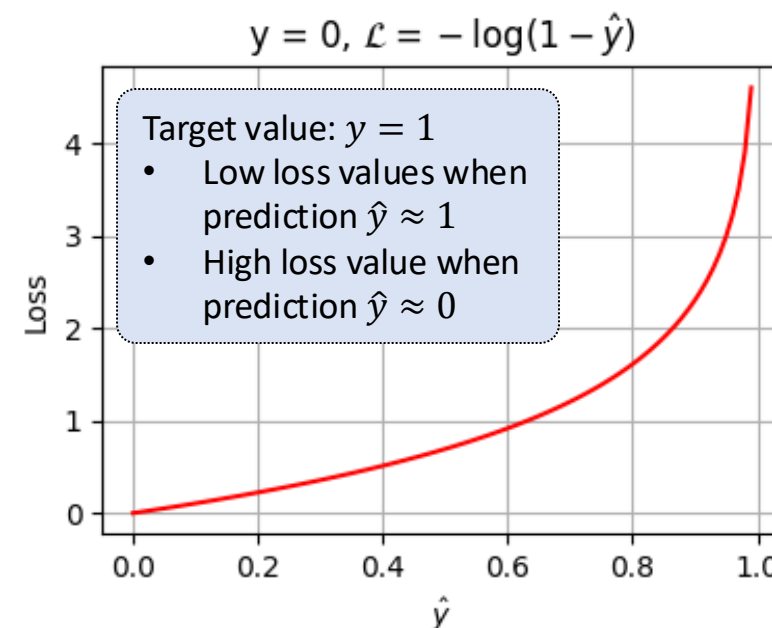
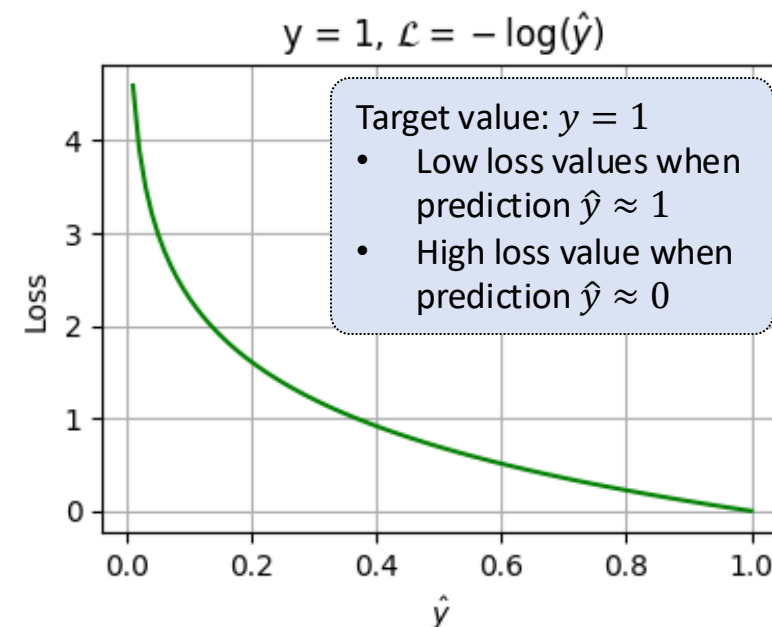
$$\frac{\partial \mathcal{L}}{\partial \theta_2} = 2(\theta_1 \theta_2 x - y)x\theta_1$$

Loss functions

- Regression
 - *Mean Squared Error, Mean Absolute Error*
- Binary
 - *Binary Cross-Entropy (BCE)*
 - $y = \{0, 1\} \rightarrow$ ground truth
 - $\hat{y} = \text{model}(x) \in [0, 1] \rightarrow$ predicted value

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

- y (ground truth) acts as a “selector” of the loss term to be applied
 - $y = 1 \rightarrow \mathcal{L} = -\log(\hat{y})$
 - $y = 0 \rightarrow \mathcal{L} = -\log(1 - \hat{y})$



Loss functions

- Multi-class classification
 - Cross-Entropy
 - Generalization to multiple classes of BCE
 - $y_i = 1$ when ground truth is the i th class, 0 otherwise
 - y_i plays the same “selector” mechanism as in BCE

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