HANDS-ON AI I

Your First Neural Networks

Sohvi Luukkonen

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

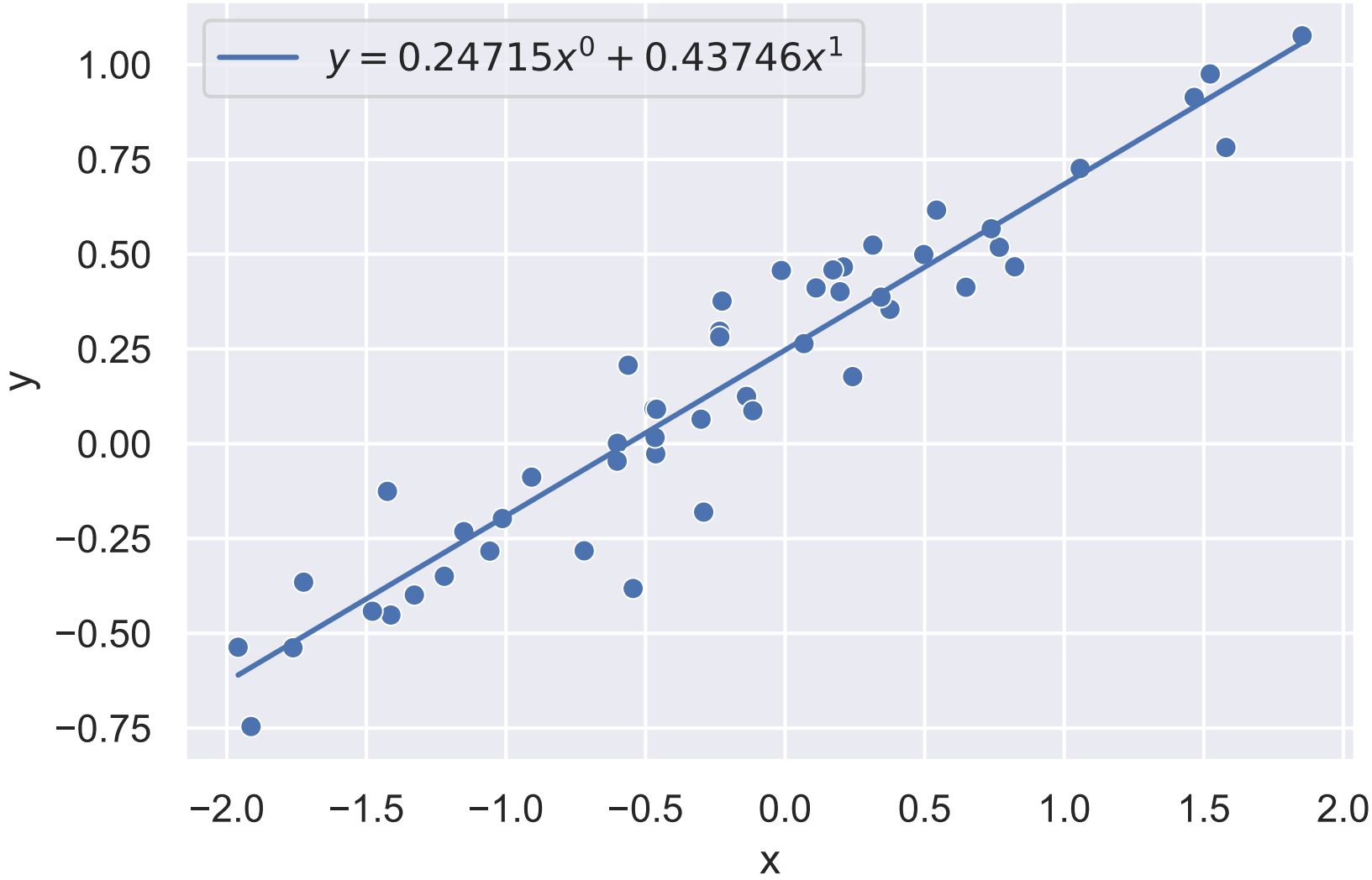
Institute for
Machine Learning

Copyright Statement

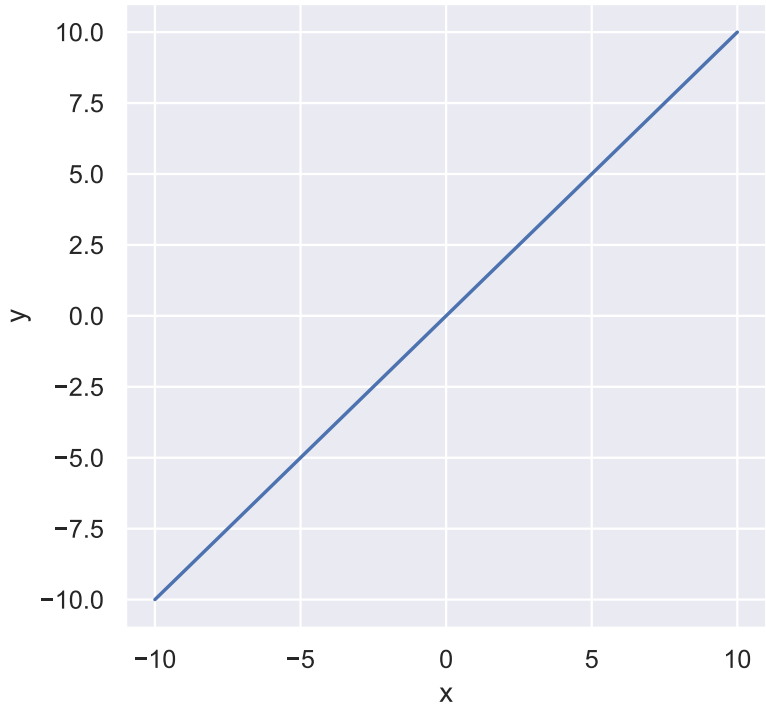
Content of Unit 5

- Let's put everything together and build our first neural networks.
- We elaborate a bit on:
  - Loss functions
  - Optimization of loss functions
  - From Logistic Regression towards Neural Networks
  - Logistic Regression and Neural Networks coded in PyTorch

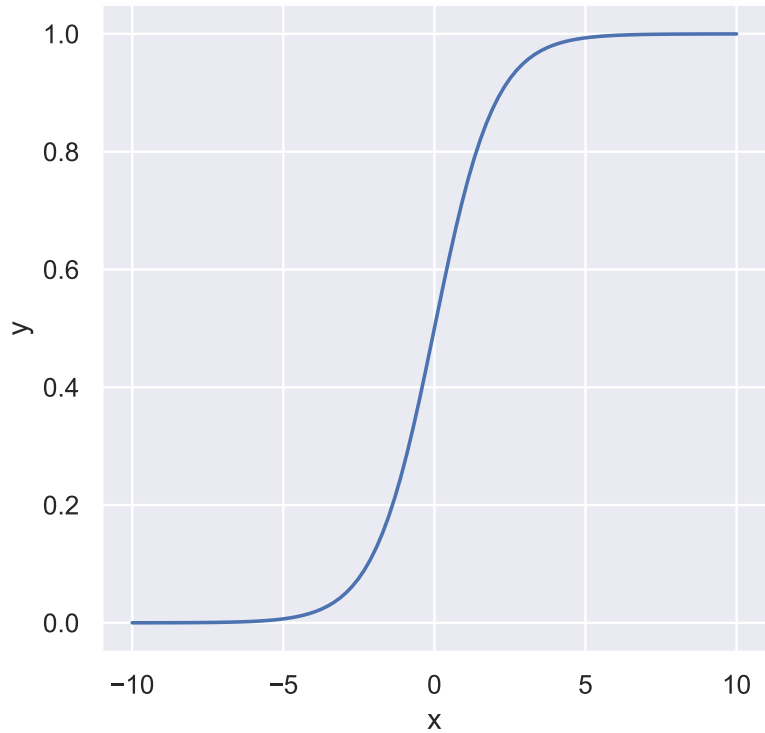
Recap: Linear Regression



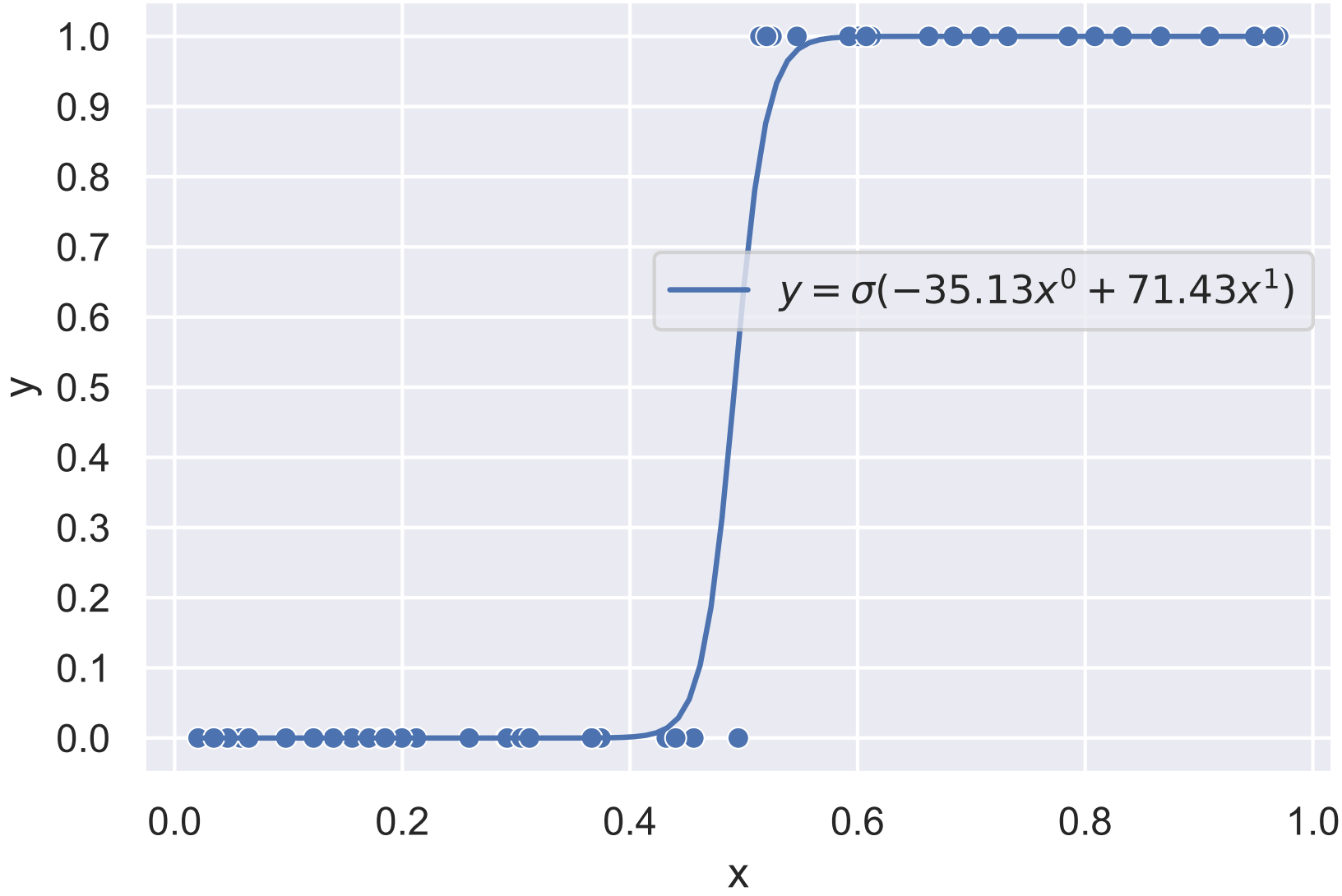
Recap: Linear Function $\Rightarrow$ Logistic Function



$\Rightarrow$



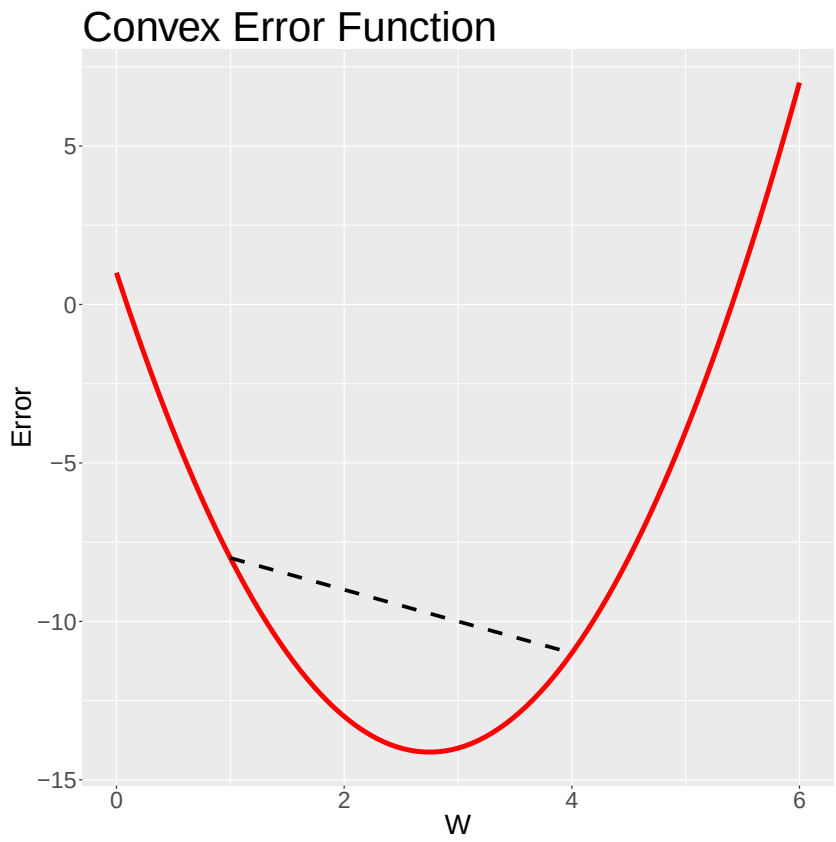
Recap: Logistic Regression

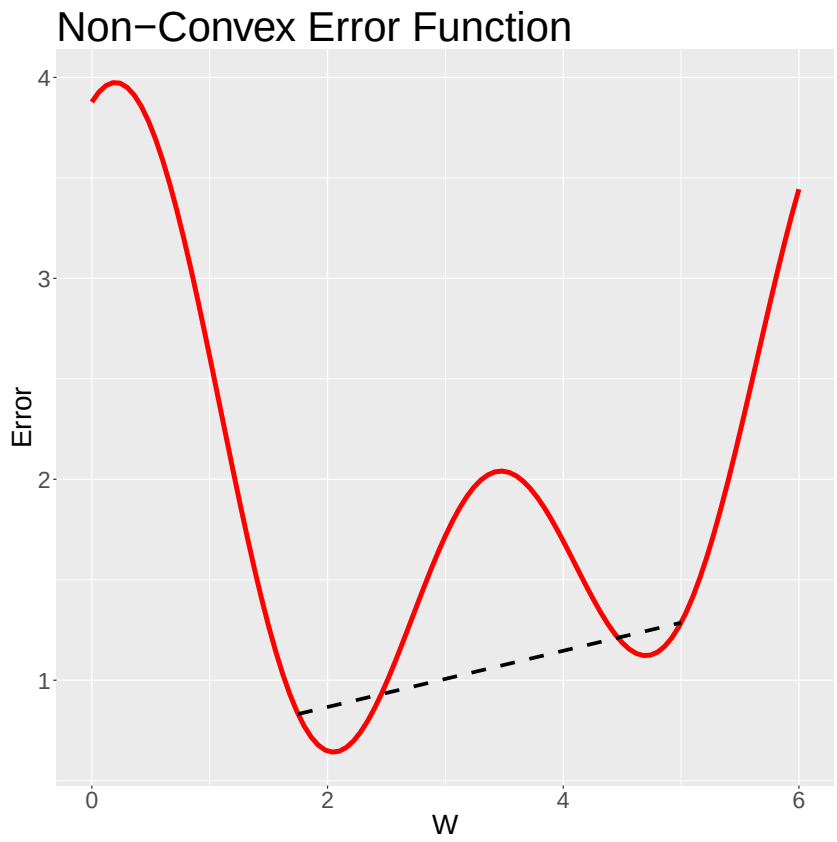


Recap: Loss Function

- The loss function tells **how good the prediction is**.
- The loss function gives the **error on the predictions**.
- We can optimize our machine learning algorithm knowing this error $\rightarrow$ we want to **minimize** the error/loss.
- Common loss functions:
  - Regression: **Mean-Squared Error** (MSE)
  - Classification: (Binary) **Cross Entropy** ((B)CE)

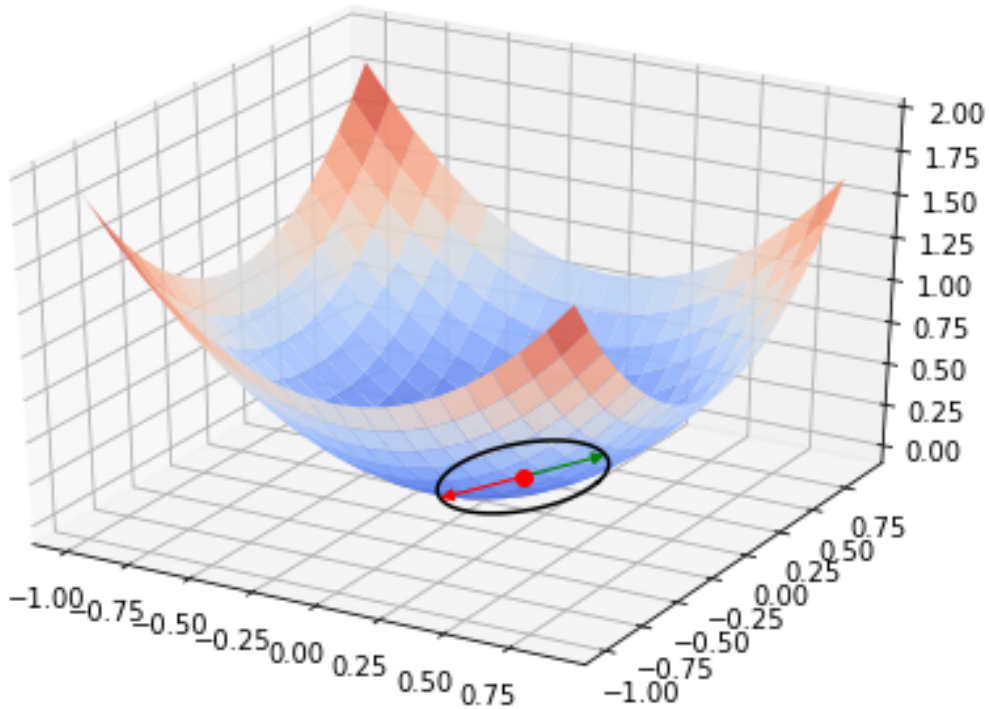
Recap: Convex vs. Non-convex Functions





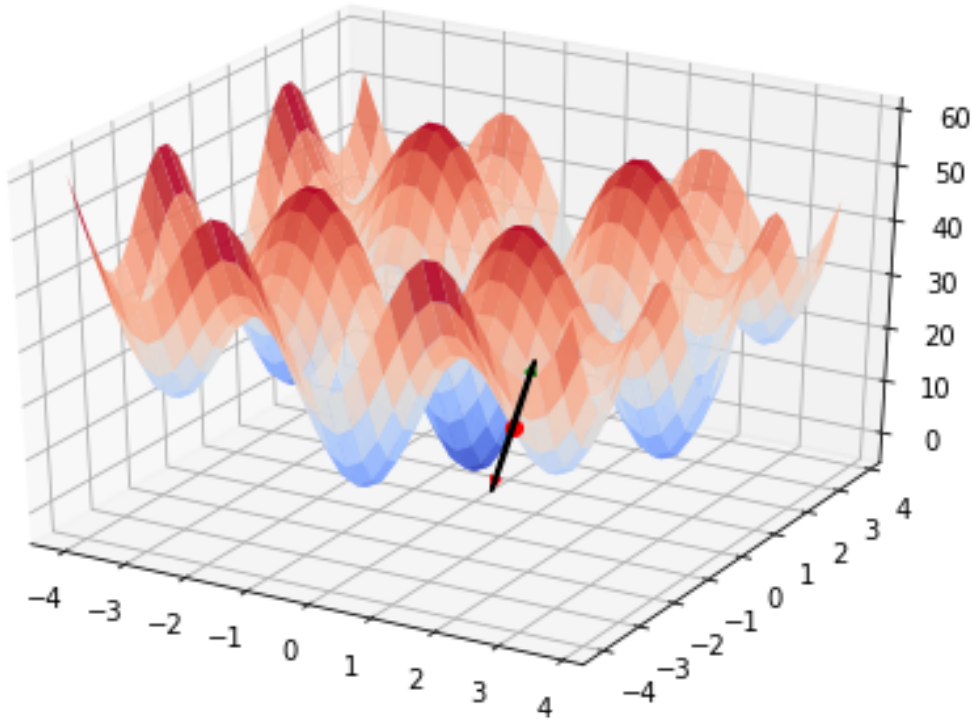
Convex Loss Functions

- Convex loss functions only have one **minimum** (global).
- Often (but not always!) easy to optimize (no iterative algorithms required).



Non-Convex Loss Functions

- Non-convex loss functions are harder to handle since they have different **local** and **global** minima.
- Difficult to optimize (in the majority of the cases, **iterative methods needed**).
- Pretty common in machine learning.



One Subtlety

- A non-convex loss function **often** has a non-closed-form solution, i.e., iterative methods are needed, there is no analytical solution.
- A convex loss function has **sometimes** a closed-form solution (linear regression), and sometimes not (logistic regression).
- While a loss function itself can be convex with respect to its own parameters (e.g., MSE), this can (and typically will) change when applied to a neural network with respect to its parameters.

Mean-Squared Error for Linear Regression

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - g(x_i; d, k)\right)^2 \qquad \text{with} \quad g(x_i; d, k) = d + k \cdot x_i$$

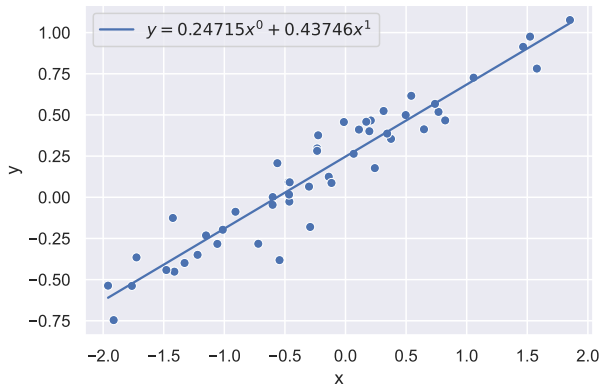
$$= \frac{1}{n}\sum_{i=1}^{n}\left(y_i - d - k \cdot x_i\right)^2$$

$$= \frac{1}{n}\sum_{i=1}^{n} y_i^2 - 2y_i d - 2y_i k x_i - d^2 - 2dkx_i - k^2 x_i^2$$

$$= -1 \cdot d^2 - \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) \cdot k^2 - \left(\frac{1}{n}\sum_{i=1}^{n} 2x_i\right) \cdot d \cdot k - \left(\frac{1}{n}\sum_{i=1}^{n} 2y_i\right) \cdot d$$

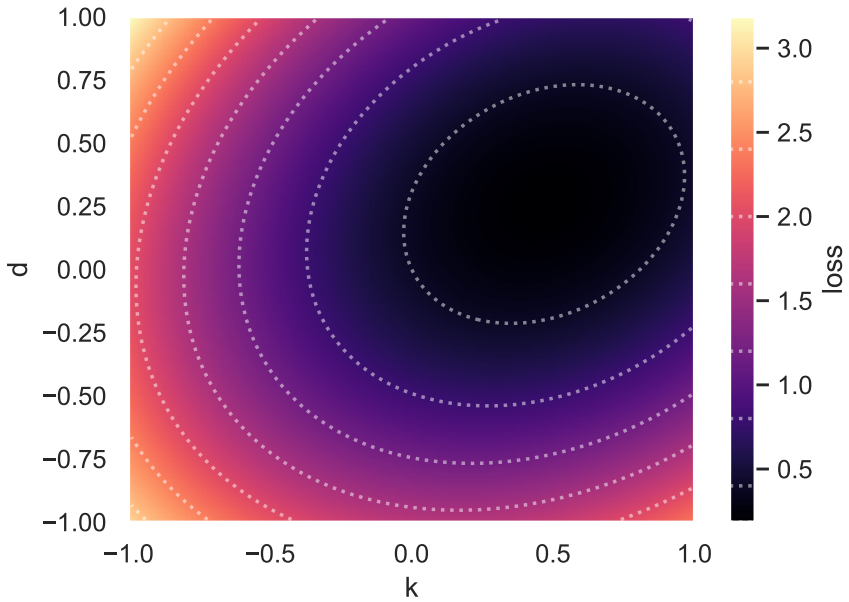
$$- \left(\frac{1}{n}\sum_{i=1}^{n} 2y_i x_i\right) \cdot k - \left(\frac{1}{n}\sum_{i=1}^{n} y_i^2\right)$$

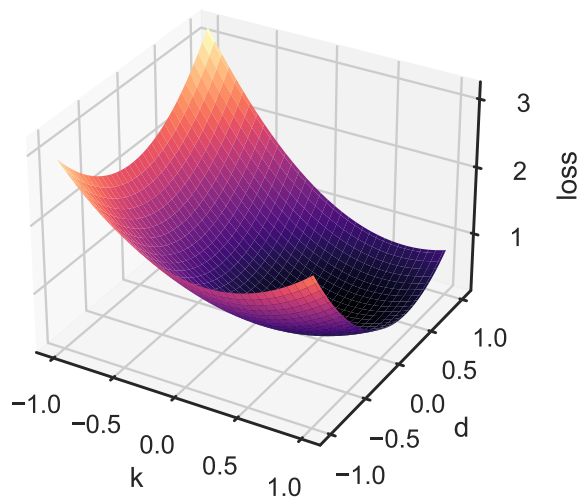
$$= a_1 \cdot d^2 + a_2 \cdot k^2 + a_3 \cdot d \cdot k + a_4 \cdot d + a_5 \cdot k + a_6$$



Closed-Form Solution

- The coefficients (a_i) of the function can **directly be calculated from the data set**.
- For quadratic functions like the MSE in combination with linear regression, the **minimum can be computed analytically**, i.e., there are equations we can plug in our coefficients to obtain the coordinates of the minimum.



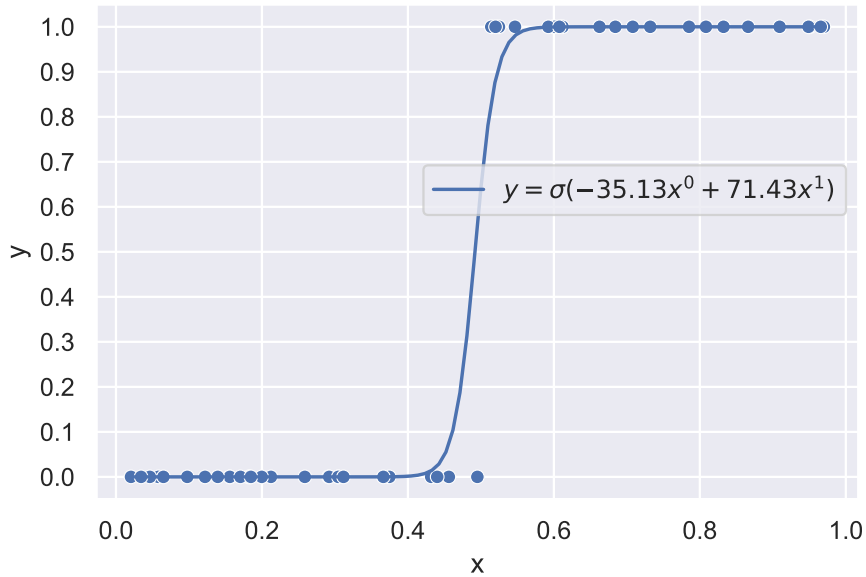


Binary Cross Entropy for Logistic Regression

$$\text{BCE} = \frac{1}{n}\sum_{i=1}^{n}\left(-y_i \log(g(x_i; d, k)) - (1 - y_i)\log(1 - g(x_i; d, k))\right)$$

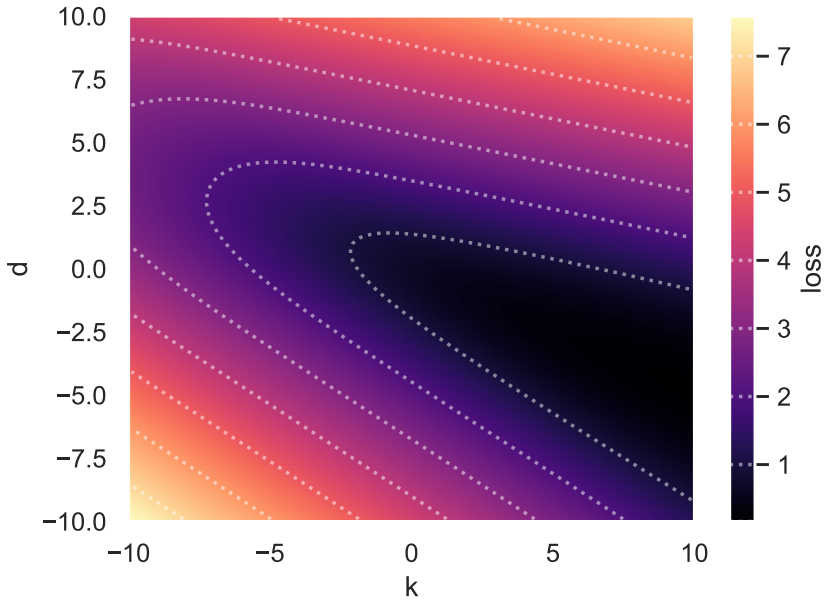
$$\text{with} \quad g(x_i; d, k) = \sigma\left(d + k \cdot x_i\right)$$

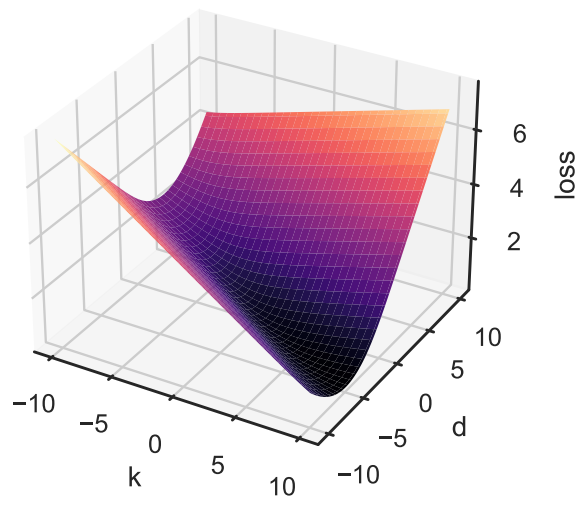
- Again, BCE can be regarded as a function that depends on the data set and the model parameters d, k.



No Closed-Form Solution

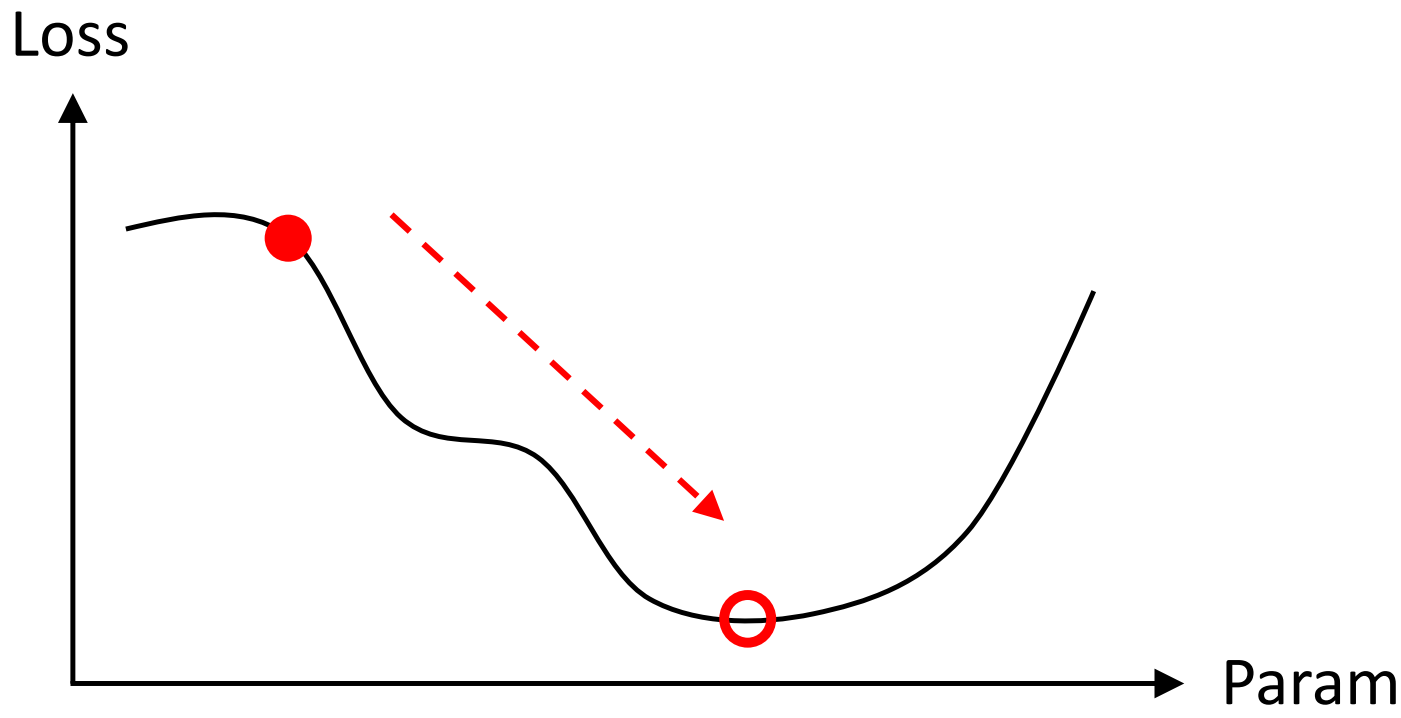
- There is **no closed-form solution** for the minimum that we can compute from the data set.
- We must use **iterative methods** such as **gradient descent** to solve this problem.





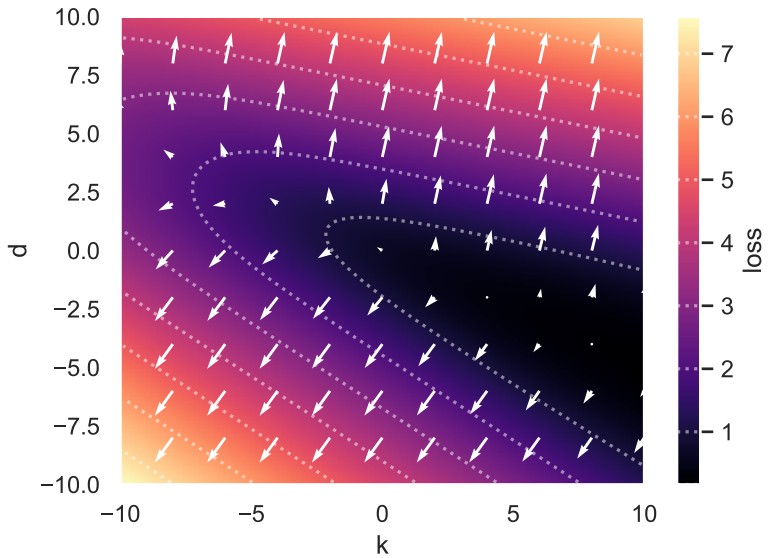
Recap: Gradient Descent

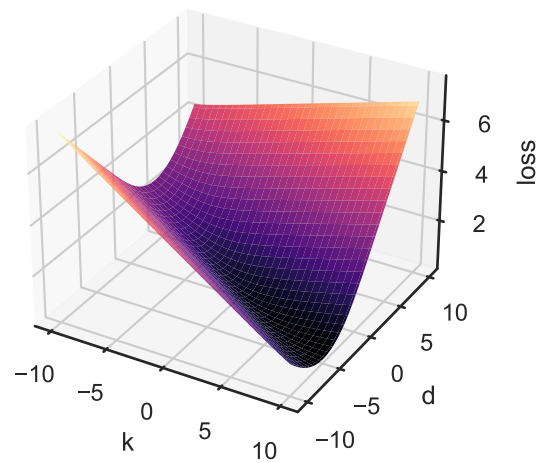
- Given: a function $f(x)$
- Task: find x that maximizes (or minimizes) $f(x)$
- Idea: start at some value x_0, and take a small step η in the direction in which the function decreases strongest



Gradient Descent for Logistic Regression

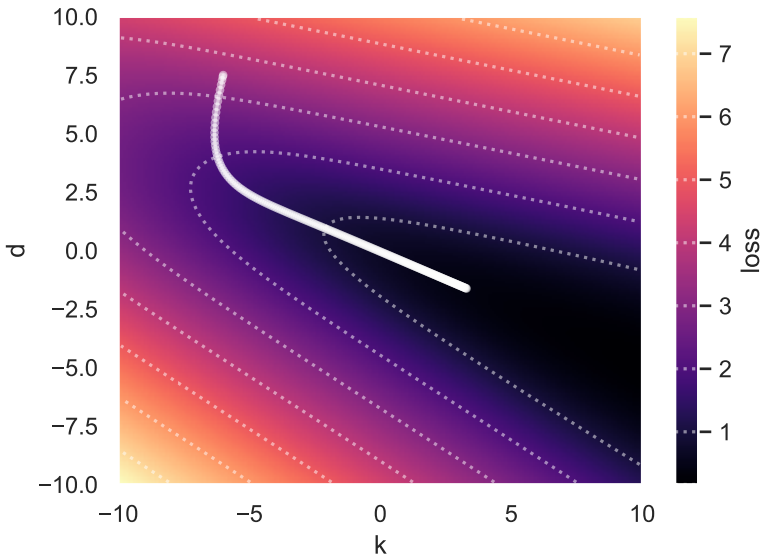
- For iterative optimization, we need the **gradient of the loss function**.
- This gradient is another function of d and k which tells us the **slope of the loss function** at a particular position.
- At each position (d, k), the gradient gives the **direction of steepest ascent**.

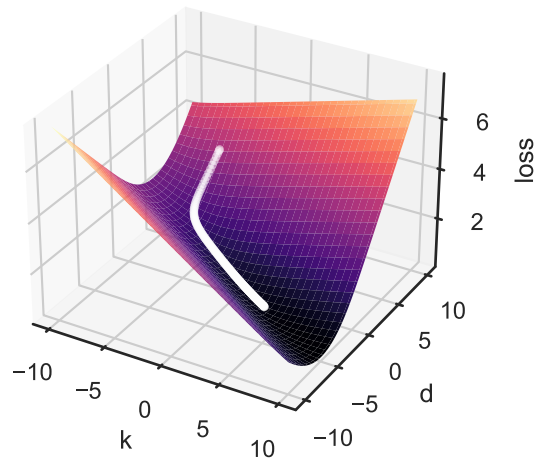




Gradient Descent for Logistic Regression

- **Gradient Descent** exploits exactly this principle: It starts at a random position and repeatedly takes **small steps in the direction opposing the gradient**.

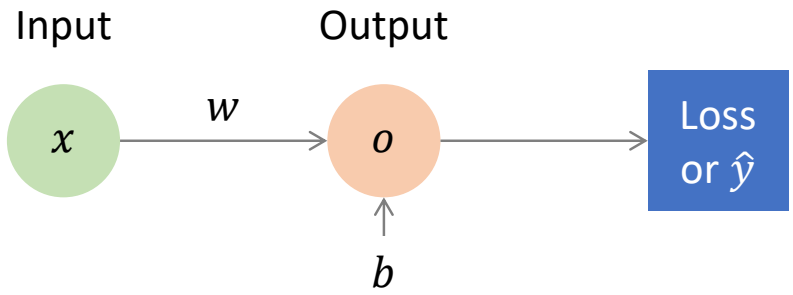




A Simple Logistic Regression Example

- Given binary labeled 1D data $((x_1, y_1), \ldots (x_n, y_n))$, where $x \in \mathbb{R}^1$ and $y \in \{0, 1\}$ (i.e., two classes 0 and 1).
- A matching logistic regression model would look like $o = g(x; \{d, k\}) = \sigma(z) = \sigma(d + k \cdot x)$, where z is the "raw" model output and o the final model output (a probability) for some given input x and the model parameters d and k.
- As preparation for the following steps, let's substitute the model parameters as follows: $d \to b, k \to w$, so our model now looks like $g(x; \{b, w\}) = \sigma(b + w \cdot x)$. Also, for brevity, we can optionally write just $g(x)$, i.e., we do not explicitly include the parameters for a more compact notation.
- We will call b the **bias** and w the **weight**.

Visualization and Output Calculation



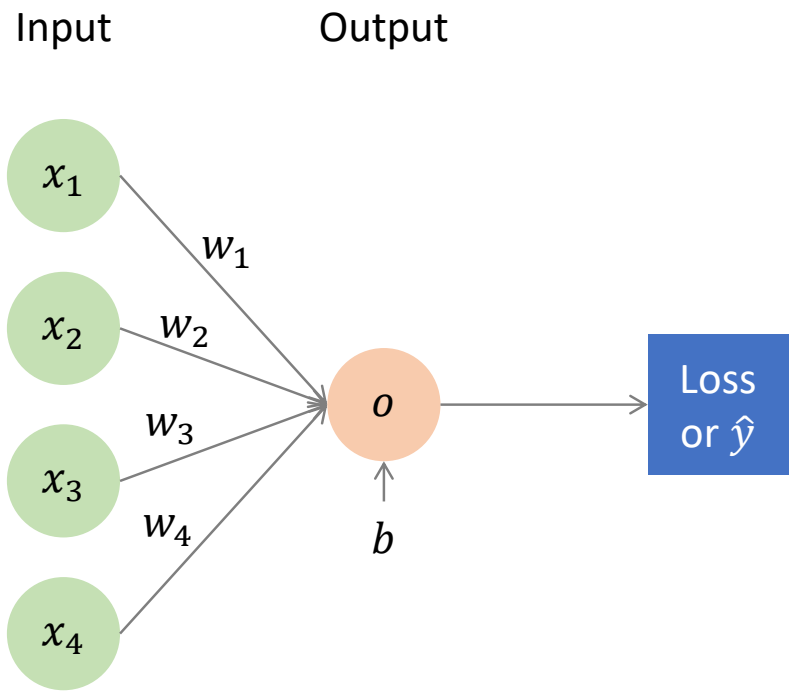
- The above is a simplified visualization of the data flow through our model: $o = \sigma(z) = \sigma(b + w \cdot x)$.
- The output o can then be used for two things:
  - Calculating the **loss** during gradient descent to repeatedly update our model parameters (bias b and weight w) to decrease the loss.
  - Determining the class **prediction** $\hat{y}$ (recall: we have to apply some decision rule, e.g., via a threshold).

Multi-Dimensional Example

- Given binary labeled M-dimensional data $((\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n))$, where $\boldsymbol{x} \in \mathbb{R}^M$ and $y \in \{0, 1\}$ (i.e., two classes 0 and 1).
- A matching logistic regression model would look like $o = g(\boldsymbol{x}; \{b, \boldsymbol{w}\}) = \sigma(z) = \sigma(b + \boldsymbol{w} \cdot \boldsymbol{x})$, where z is the "raw" model output and o the final model output (a probability) for some given input $\boldsymbol{x}$ and the model parameters b and $\boldsymbol{w}$.
- Note that the parameter $\boldsymbol{w}$ is now a vector,[1] and $\boldsymbol{w} \cdot \boldsymbol{x}$ is the scalar/dot product, i.e., $\sum_{i=1}^{M} w_i x_i$.

[1] Strictly speaking, b is internally also a vector since $b + \boldsymbol{w} \cdot \boldsymbol{x}$ is actually $\boldsymbol{b} \cdot \boldsymbol{x}^0 + \boldsymbol{w} \cdot \boldsymbol{x}^1$. However, because the computation does not depend on the input $\boldsymbol{x}$, i.e., $\boldsymbol{b} \cdot \boldsymbol{x}^0 = \boldsymbol{b} \cdot \boldsymbol{1} = \sum_{i=1}^{M} b_i \cdot 1 = \sum_{i=1}^{M} b_i$, we simplify it to just b.

Visualization and Output Calculation

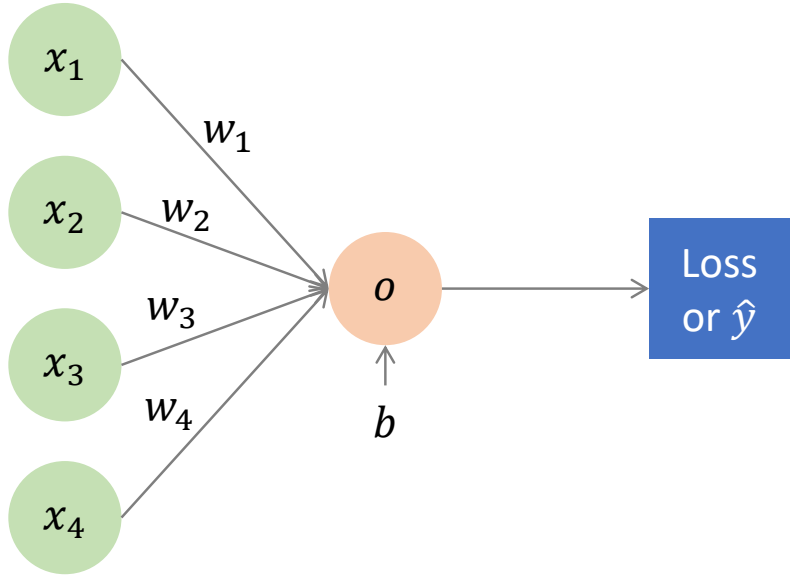


- The above is a simplified visualization of the data flow through our model with an $M = 4$-dimensional input size. See the next slide for how the final model output is calculated.

Output Calculation

- Model parameters b and $\boldsymbol{w}$:

$$b, \boldsymbol{w} = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$



- "Raw"/linear model output z given $M = 4$-dimensional input $\boldsymbol{x}$ (with $\boldsymbol{w} \cdot \boldsymbol{x}$ being the scalar product):

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, z = b + \boldsymbol{w} \cdot \boldsymbol{x} = b + \sum_{i=1}^{4} w_i x_i$$
$$= b + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$$

- Final model output o (class probability):

$$o = \sigma(z)$$

Multi-Class Example

- Given K-multi-class labeled 1D data $((x_1, y_1), \ldots (x_n, y_n))$, where $x \in \mathbb{R}^1$ and $y \in \{1, \ldots, K\}$.
- Recall that the softmax function is suitable for multi-class classification, so we replace our logistic function $\sigma(z)$ with the softmax function $\sigma(\boldsymbol{z})_i$, where i means the i-th class and $\boldsymbol{z} = (z_1, \ldots, z_K)^\top$ is the vector of "raw" model outputs.

Recap: Softmax

- Generalization of the sigmoid function.
- Suitable for **multi-class** classification.
- For K classes with $y \in \{1, \ldots, K\}$ the probability of $\boldsymbol{x}$ belonging to class i is:

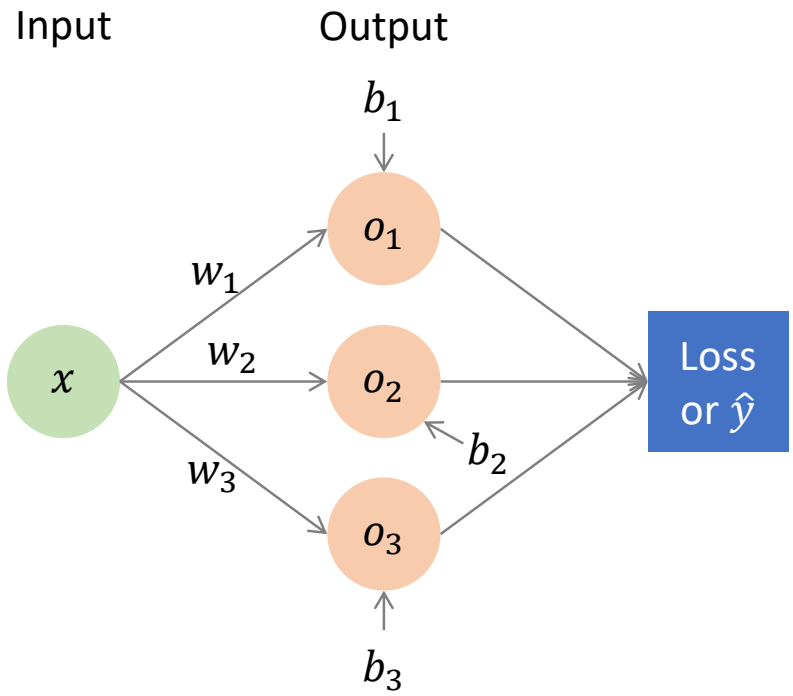
$$p(y = i \mid \boldsymbol{x}) = \sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

  where $\boldsymbol{z} = (z_1, \ldots, z_K)$ is the vector of "raw" model outputs, i.e., there is an output for each class (see example later).
- While not necessary, as a regular sigmoid function would suffice, this also works for binary classification, i.e., $K = 2$.

Multi-Class Example

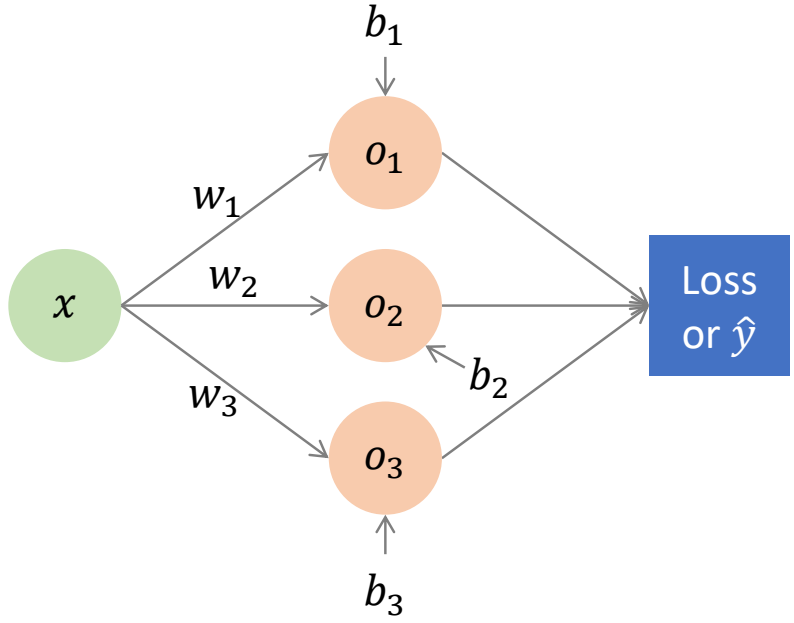
- Given K-multi-class labeled 1D data $((x_1, y_1), \ldots (x_n, y_n))$, where $x \in \mathbb{R}^1$ and $y \in \{1, \ldots, K\}$.
- Recall that the softmax function is suitable for multi-class classification, so we replace our logistic function $\sigma(z)$ with the softmax function $\sigma(\boldsymbol{z})_i$, where i means the i-th class and $\boldsymbol{z} = (z_1, \ldots, z_K)^\top$ is the vector of "raw" model outputs.
- A matching logistic regression model would look like $\boldsymbol{o} = (o_1, \ldots, o_K)^\top = g(x; \{\boldsymbol{b}, \boldsymbol{w}\}) = (\sigma(\boldsymbol{z})_1, \ldots, \sigma(\boldsymbol{z})_K)^\top$, where $\boldsymbol{o}$ is the final vector of K class probabilities for some given input x and the model parameters $\boldsymbol{b}$ and $\boldsymbol{w}$.
- Note that the parameters $\boldsymbol{b}$ and $\boldsymbol{w}$ are now vectors.

Visualization



- The above is a simplified visualization of the data flow through our model with 1D input size and $K = 3$ classes. See the next slide for how the final model output is calculated.

Output Calculation



- Model parameters $\boldsymbol{b}$ and $\boldsymbol{w}$:

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

- "Raw"/linear model output $\boldsymbol{z}$ given input x:

$$x, \boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \boldsymbol{b} + \boldsymbol{w} \cdot x = \begin{bmatrix} b_1 + w_1 x \\ b_2 + w_2 x \\ b_3 + w_3 x \end{bmatrix}$$

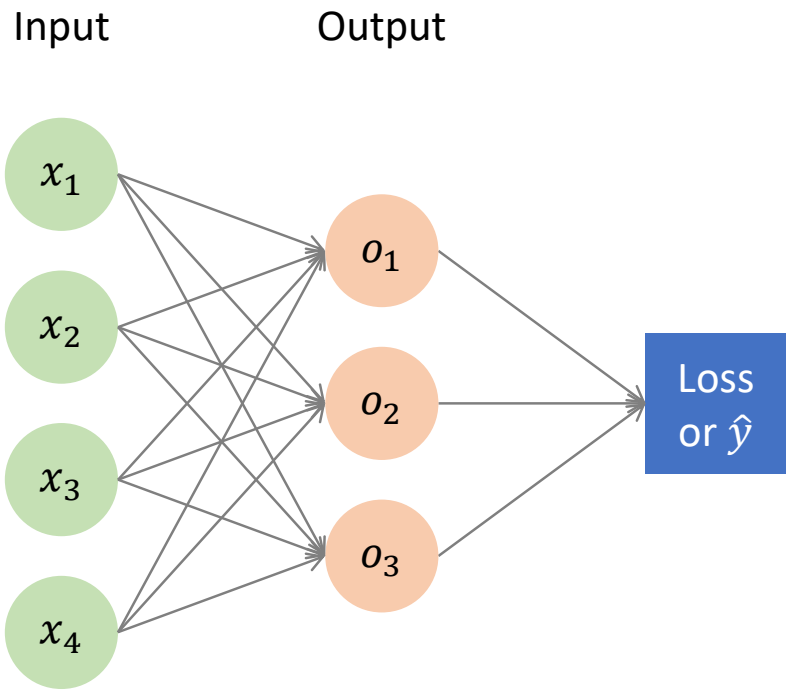
- Final model output $\boldsymbol{o}$ (class probability vector):

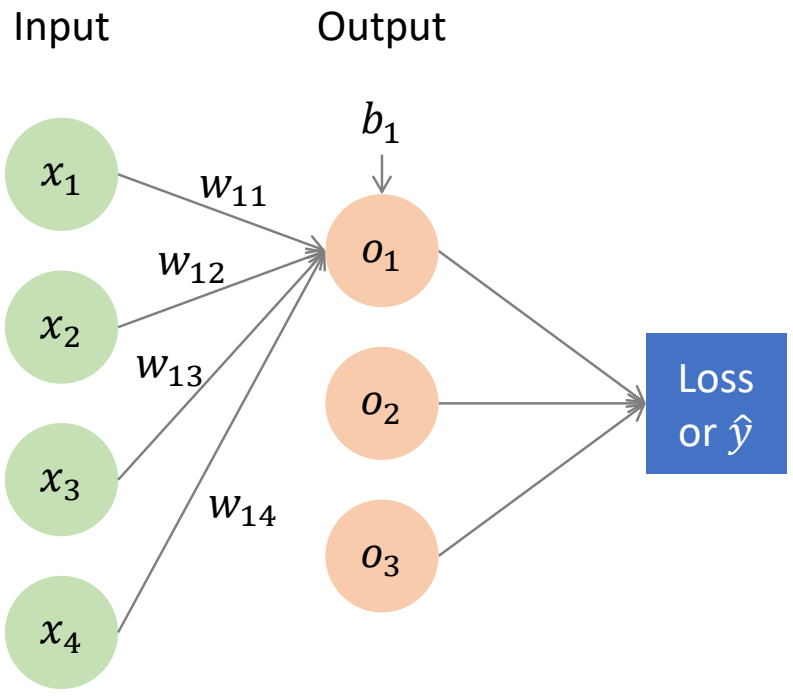
$$\boldsymbol{o} = \begin{bmatrix} o_1 \\ o_2 \\ o_3 \end{bmatrix} = \begin{bmatrix} \sigma(\boldsymbol{z})_1 \\ \sigma(\boldsymbol{z})_2 \\ \sigma(\boldsymbol{z})_3 \end{bmatrix}$$

Multi-Dimensional and Multi-Class Example

- Given K-multi-class labeled M-dimensional data $((\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n))$, where $\boldsymbol{x} \in \mathbb{R}^M$ and $y \in \{1, \ldots, K\}$.
- A matching logistic regression model would look like $\boldsymbol{o} = (o_1, \ldots, o_K)^\top = g(\boldsymbol{x}; \{\boldsymbol{b}, \boldsymbol{W}\}) = (\sigma(\boldsymbol{z})_1, \ldots, \sigma(\boldsymbol{z})_K)^\top$, where $\boldsymbol{o}$ is the final vector of K class probabilities for some given input $\boldsymbol{x}$ and the model parameters $\boldsymbol{b}$ and $\boldsymbol{W}$.
- Note that the parameter $\boldsymbol{b}$ is now a vector and the parameter $\boldsymbol{W}$ a matrix.

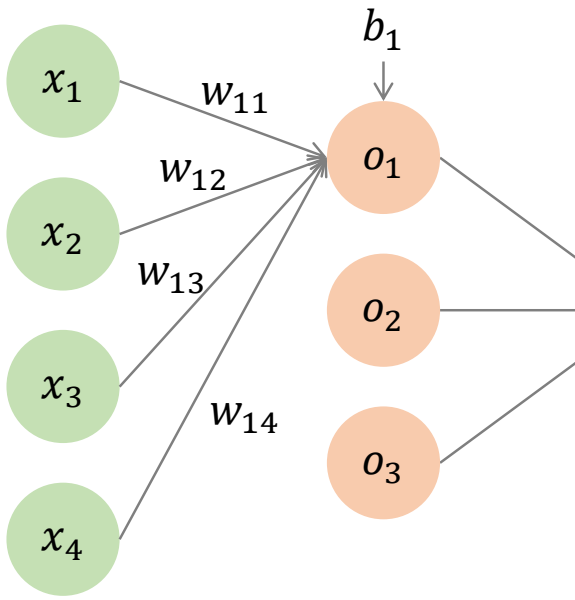
Visualization





- The above is a simplified visualization of the data flow through our model with $M = 4$-dimensional input size and $K = 3$ classes. See the next slide for how the final model output is calculated.

Output Calculation



- Model parameters $\boldsymbol{b}$ and $\boldsymbol{W}$:

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

- "Raw"/linear model output $\boldsymbol{z}$ given $M = 4$-dimensional input $\boldsymbol{x}$ (with $\boldsymbol{W}\boldsymbol{x}$ being the matrix product):

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x} = \begin{bmatrix} b_1 + \sum_{i=1}^{4} w_{1i}x_i \\ b_2 + \sum_{i=1}^{4} w_{2i}x_i \\ b_3 + \sum_{i=1}^{4} w_{3i}x_i \end{bmatrix}$$

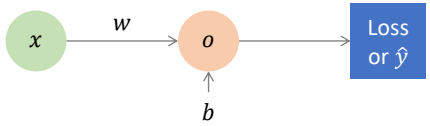
- Final model output $\boldsymbol{o}$ (class probability vector):

$$\boldsymbol{o} = \begin{bmatrix} o_1 \\ o_2 \\ o_3 \end{bmatrix} = \begin{bmatrix} \sigma(\boldsymbol{z})_1 \\ \sigma(\boldsymbol{z})_2 \\ \sigma(\boldsymbol{z})_3 \end{bmatrix}$$

Recap - Logistic Regression Calculation

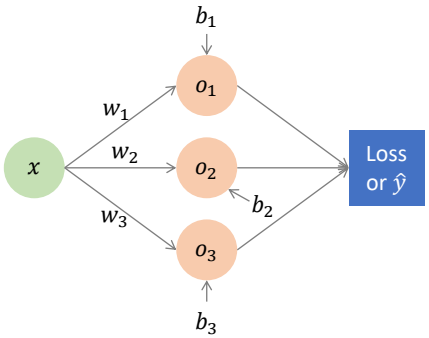
$D = 1, K = 1$

Input Output



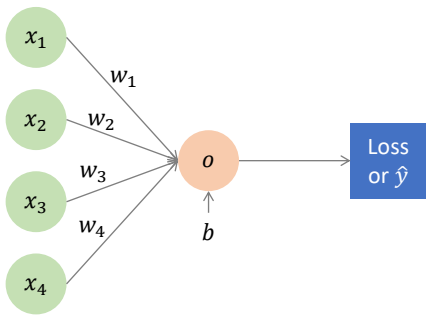
$D = 1, K = 3$

Input Output



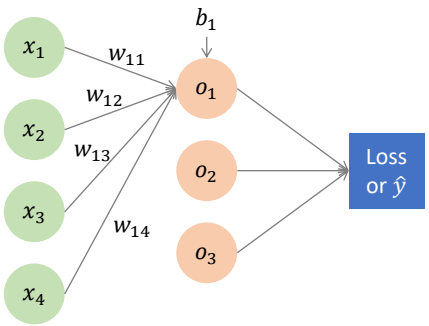
$D = 4, K = 1$

Input Output



$D = 4, K = 3$

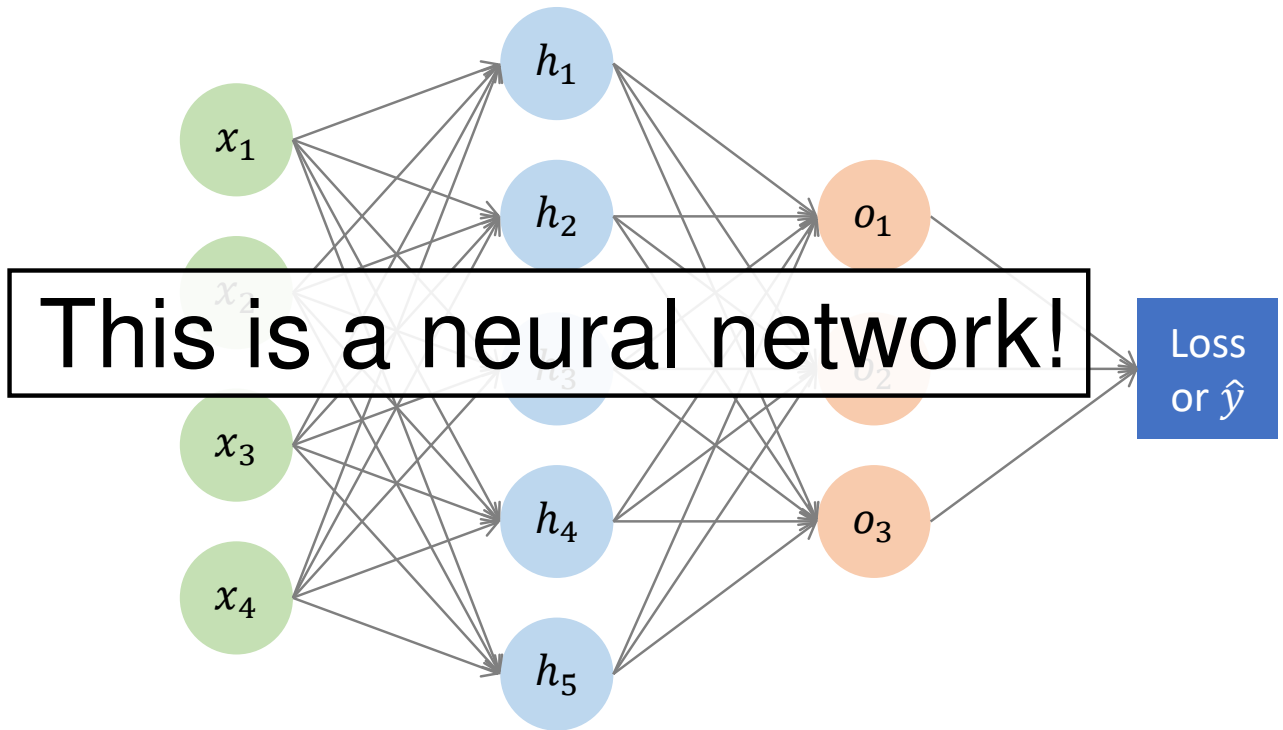
Input Output



$$\boldsymbol{o}_K^\top = \sigma\left(\boldsymbol{b}_K^\top + \boldsymbol{W}_{K\times D}\boldsymbol{x}_D\right)$$

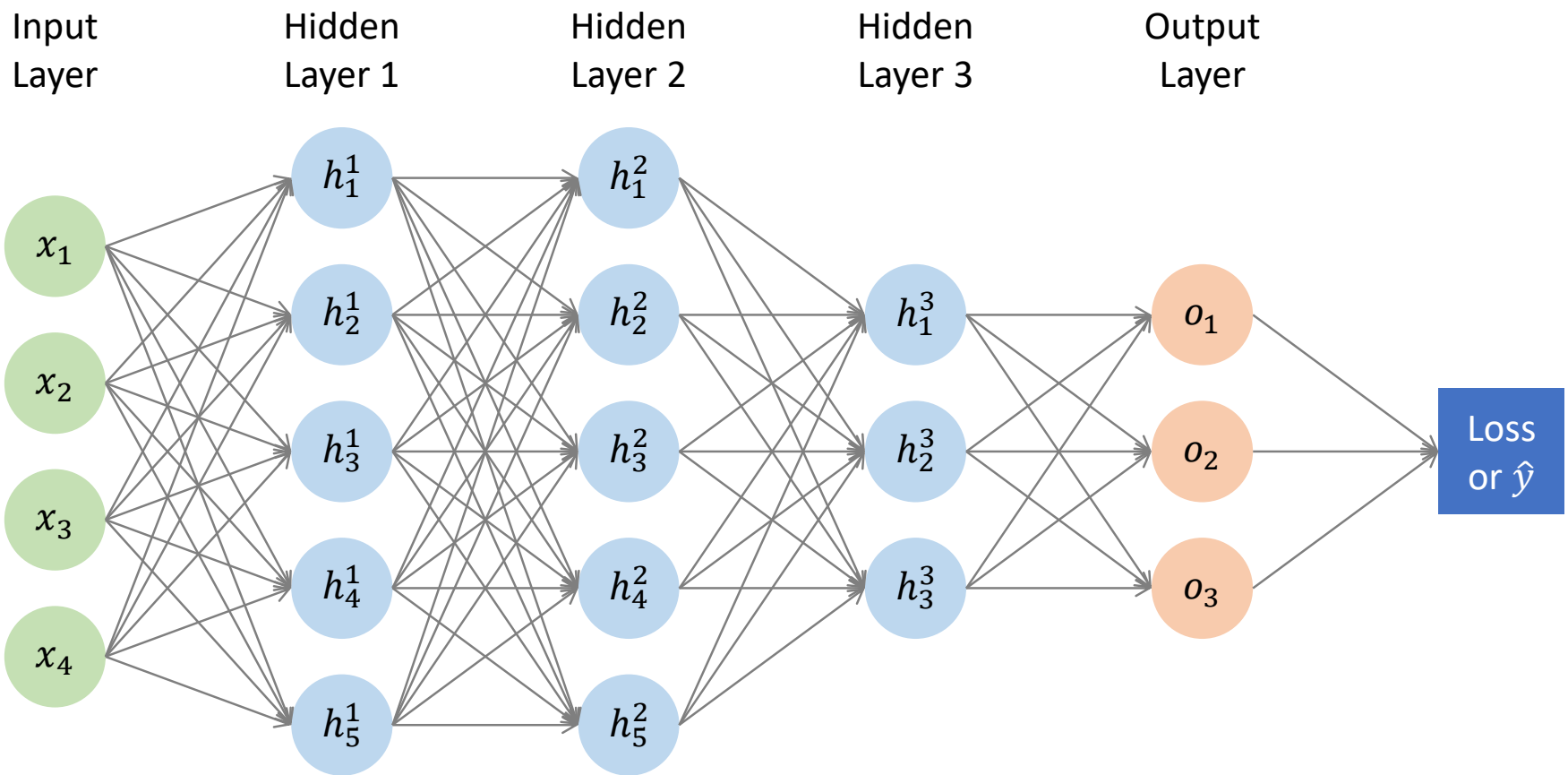
Logistic Regression $\Rightarrow$ Neural Networks

- Again as preparation, let's rename *Input* and *Output*:
- What if we made our model g **more complex/powerful**?
- We could just reuse the idea of our bias-weight calculation by **adding** an intermediate/hidden **layer** h with output $\boldsymbol{h} = h(\boldsymbol{x}; \{\boldsymbol{b}, \dots$ $\dots(\boldsymbol{x}))$:



Neural Networks

- We can add even more such layers, and the number of **nodes** (a.k.a. **units** or **neurons**) can also vary.
- E.g., 3 layers with a final output of $\boldsymbol{o} = g(h^3(h^2(h^1(\boldsymbol{x}))))$:



Stacking Linear Layers

- There is a **caveat** when naively stacking such linear layers.
- If we perform **multiple linear transformations** in a row, they can be **collapsed into** a **single** linear transformation. For n hidden layers h^i with output $\boldsymbol{h}^i = h^i(\boldsymbol{x}; \{\boldsymbol{b}^i, \boldsymbol{W}^i\}) = \boldsymbol{b}^i + \boldsymbol{W}^i\boldsymbol{x}$ or simply $h^i(\boldsymbol{x})$ for short, we get:

$$
\begin{aligned}
\boldsymbol{h}^n &= h^n(h^{n-1}(\ldots h^1(\boldsymbol{x}))) \\
&= \boldsymbol{b}^n + \boldsymbol{W}^n(\boldsymbol{b}^{n-1} + \boldsymbol{W}^{n-1}(\ldots \boldsymbol{b}^1 + \boldsymbol{W}^1\boldsymbol{x})) \\
&= \boldsymbol{b}^n + \boldsymbol{W}^n\boldsymbol{b}^{n-1} + \boldsymbol{W}^n\boldsymbol{W}^{n-1}\boldsymbol{b}^{n-2} + \ldots + \boldsymbol{W}^n\boldsymbol{W}^{n-1}\ldots\boldsymbol{W}^1\boldsymbol{x} \\
&= \boldsymbol{b}' + \boldsymbol{W}'\boldsymbol{x}
\end{aligned}
$$

- This means that simply stacking such layers **does not increase the complexity/power of our model** (it is just a single linear transformation again).

Non-linearity

- To get rid of the linear transformation problem, we thus add a **non-linear function** f after each layer.

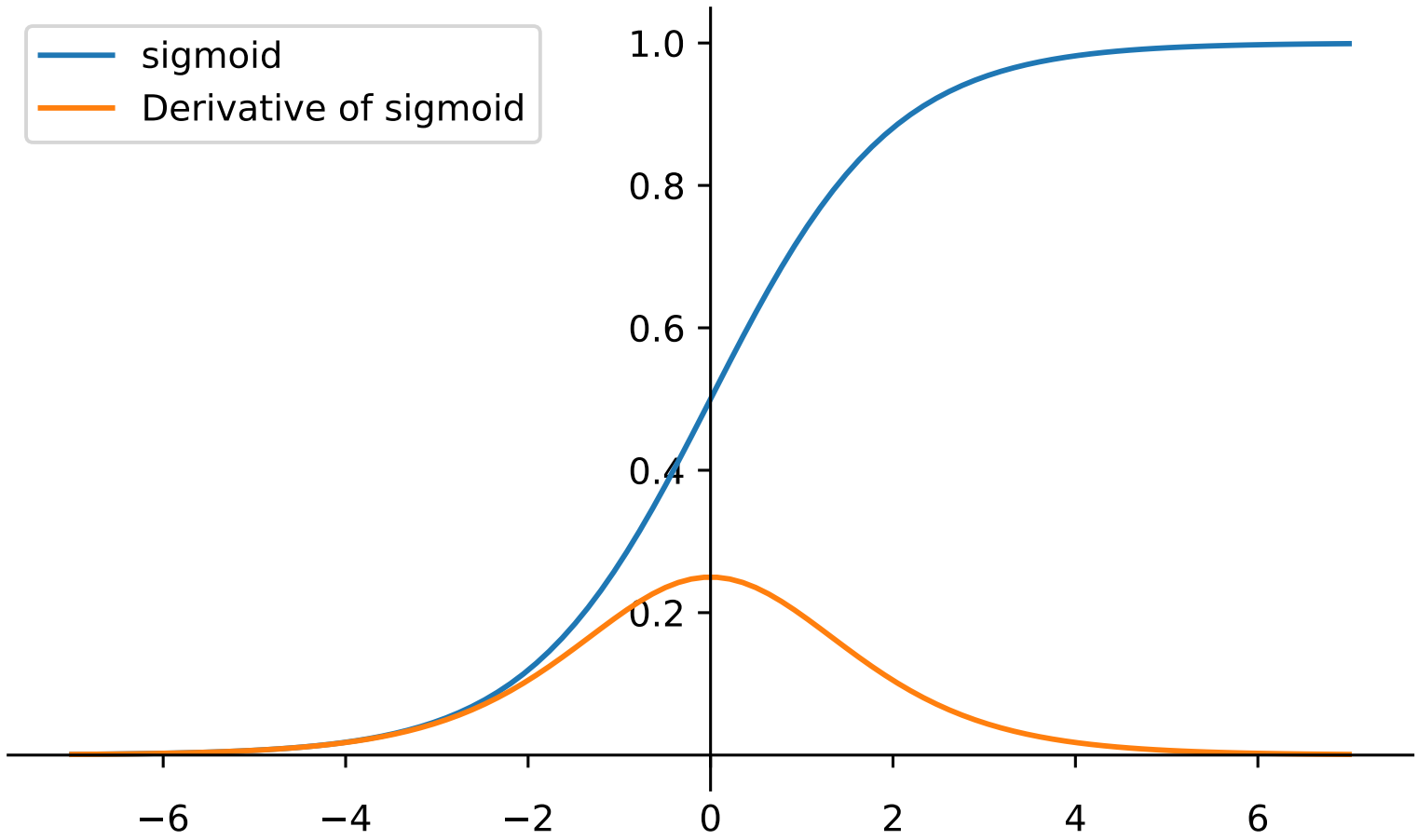
$$\text{Before: } h(\boldsymbol{x}) = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}$$

$$\text{After: } h(\boldsymbol{x}) = f(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x})$$

- Looks familiar? Indeed it does, this is the same idea we had when going from linear regression to logistic regression with $f = \sigma$.
- The logistic/sigmoid function is actually already such a function that we can use to introduce non-linearity.
- These functions are also called **activation functions**.

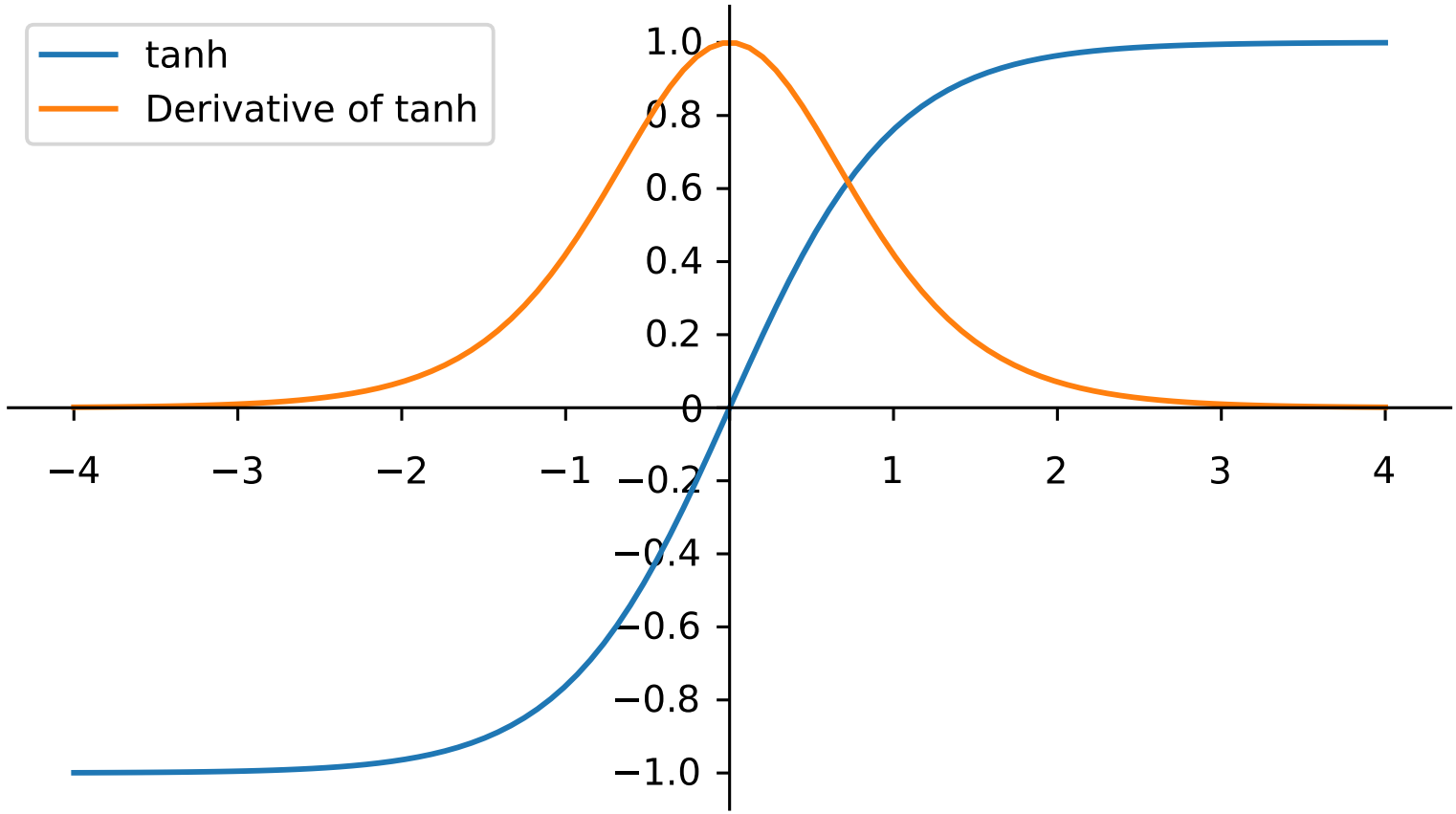
Common Activation Functions

- There are various activation functions, each with their own benefits and disadvantages.
- Here are some often used activation functions:



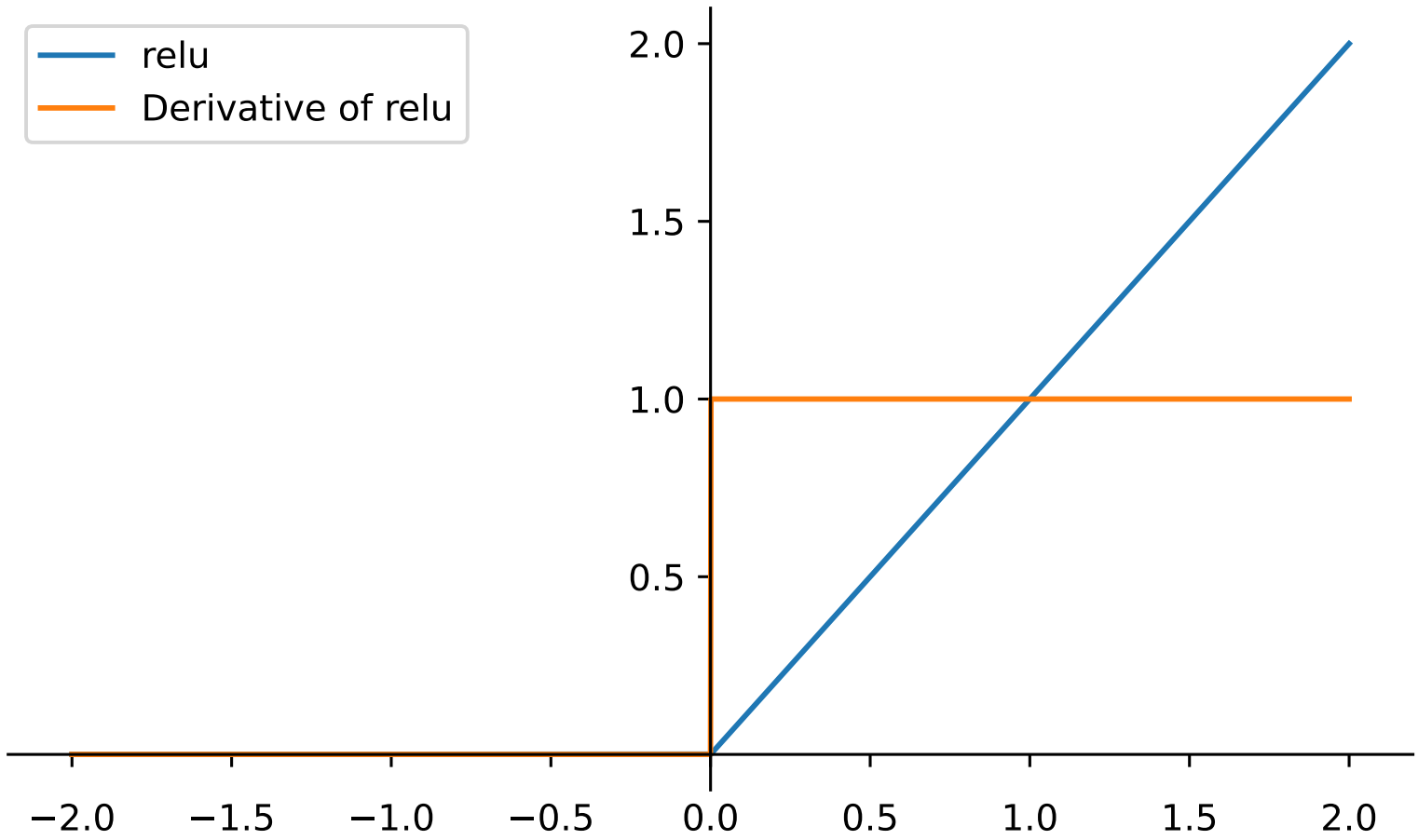
Common Activation Functions

- There are various activation functions, each with their own benefits and disadvantages.
- Here are some often used activation functions:



Common Activation Functions

- There are various activation functions, each with their own benefits and disadvantages.
- Here are some often used activation functions:

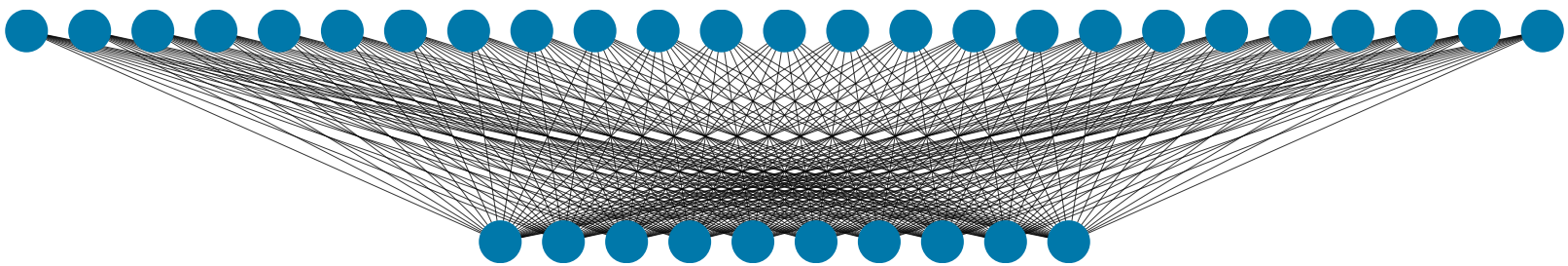


Logistic Regression in PyTorch

- The following is just a brief summary. For more details, see the exercise and the accompanying materials.
- Define the model, e.g, for $M = 25$-dimensional input and $K = 10$ classes, and define the loss:

```
import torch

model = torch.nn.Sequential(
    torch.nn.Flatten(), # Ensure flat vector
    torch.nn.Linear(25, 10)
    #torch.nn.Softmax() # Not needed (incl. in loss)
)
loss = torch.nn.functional.cross_entropy
```

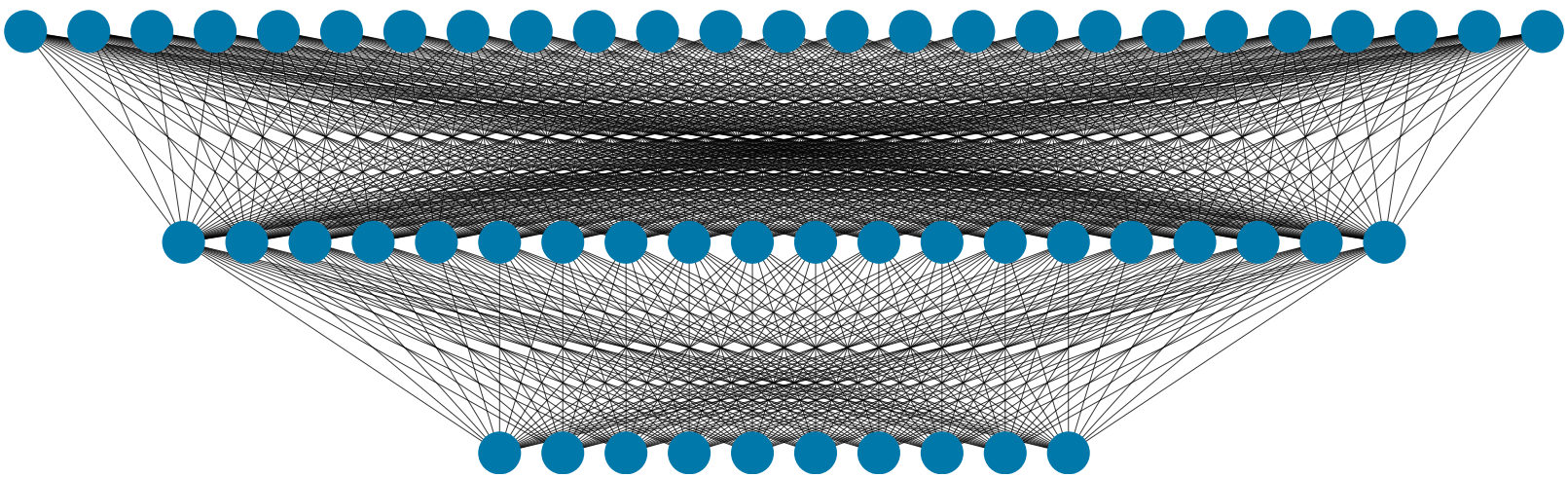


Adding Another Layer

- Neural network model with one hidden layer and 20 nodes in PyTorch:

```
model = torch.nn.Sequential(
    torch.nn.Flatten(),
    torch.nn.Linear(25, 20),

    torch.nn.Linear(20, 10)
)
```



Adding Non-linearity

- Neural network model with one hidden layer and 20 nodes in PyTorch including appropriate non-linearity:

```
model = torch.nn.Sequential(
    torch.nn.Flatten(),
    torch.nn.Linear(25, 20),
    torch.nn.Sigmoid(), # or: Tanh(), ReLU()
    torch.nn.Linear(20, 10)
)
```