# Fully-Connected Neural Networks

# Reading

- Deep Learning: chapters 6.1-6.4
  - https://www.deeplearningbook.org/contents/mlp.html

- An overview of feedforward neural networks
  - Many, many other types nowadays…

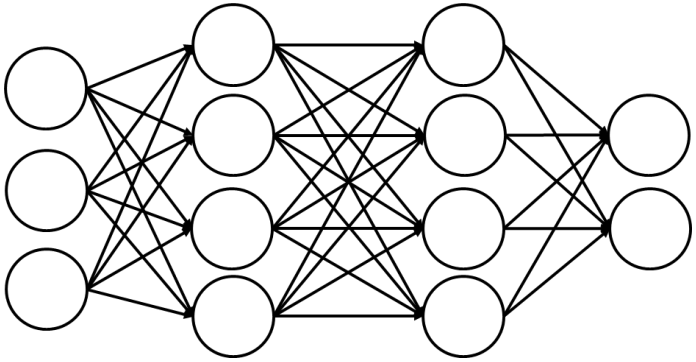
# Overview

- Neural networks have been around for a while
  - Initially developed in the 1940s
  - Earlier attempts suffered from insufficient computational power (for training purposes) and insufficient data (overfitting)
- Neural networks became popular (again) in the early 2010s
- In the early 2010s, Krizhevsky et al. noticed that one could use GPUs to train very large neural networks on large datasets
  - That sparked a decade of frantic improvements

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012).

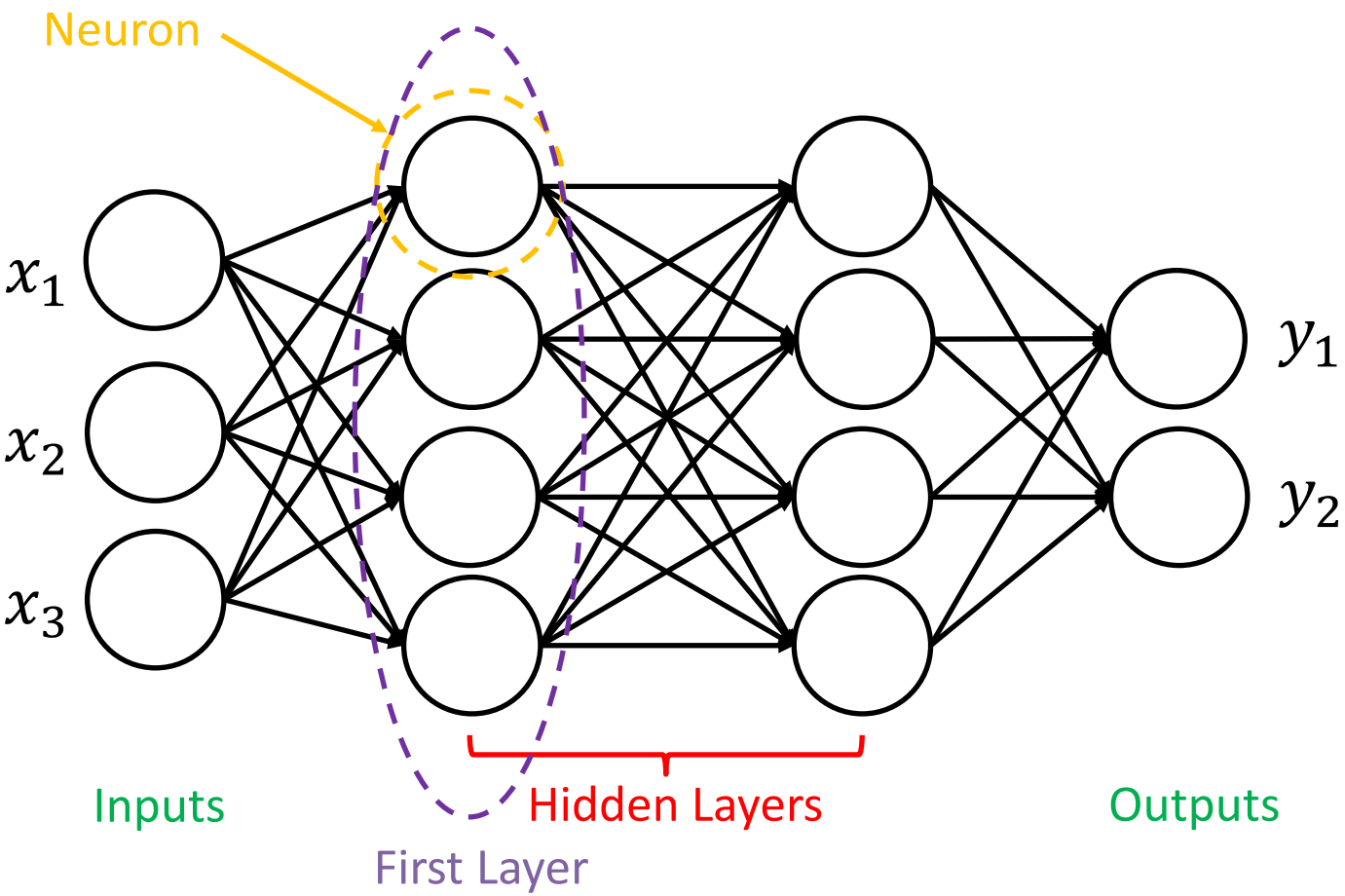
# Feedforward Neural Networks

- Also known as multi-layer perceptrons
  - Old name, at least from the 1960's



- The term "deep neural networks" is essentially rebranding
  - Modern networks are deeper than ever, however
  - Term "neural" is (very) loosely inspired by neuroscience
- The term "feedforward" means that computation happens from left to right in network, without any feedback

# NN terminology



## NNs as functions

- Standard ML model

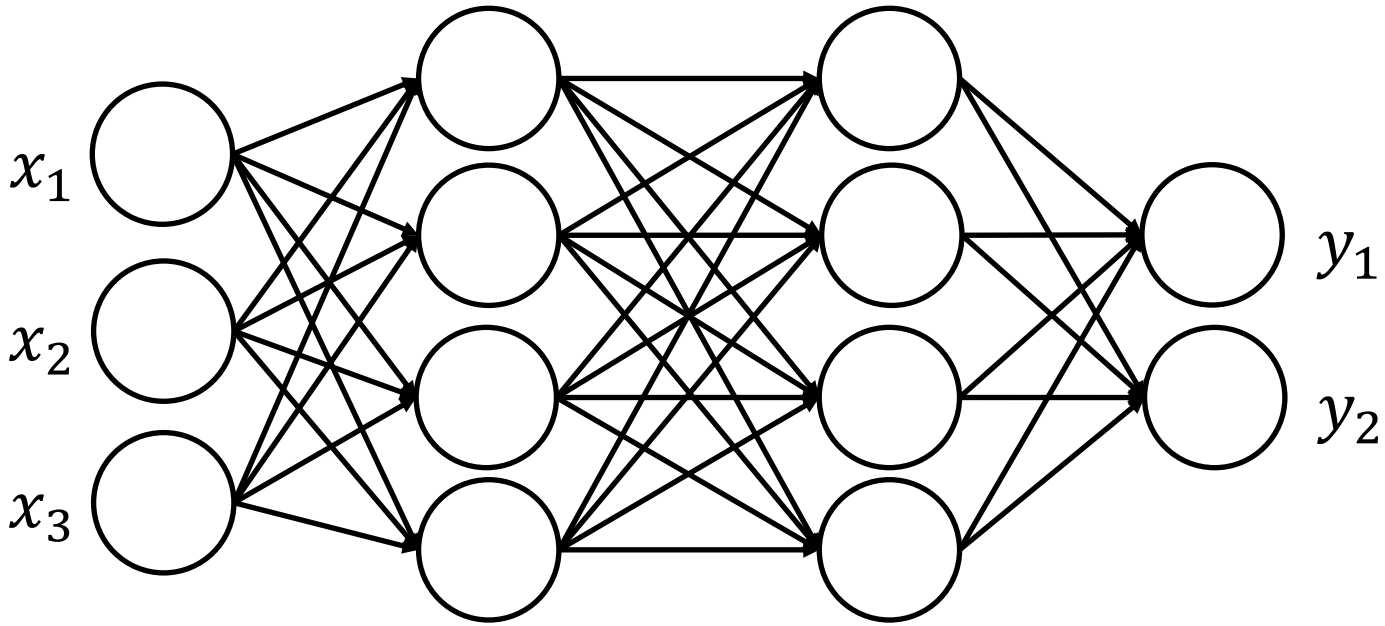
$$y = f(\boldsymbol{x}; \boldsymbol{\theta})$$

  - where $\boldsymbol{x}$ are the inputs (e.g., pixels), $y$ are the outputs (e.g., labels), $\boldsymbol{\theta}$ are the parameters to be optimized

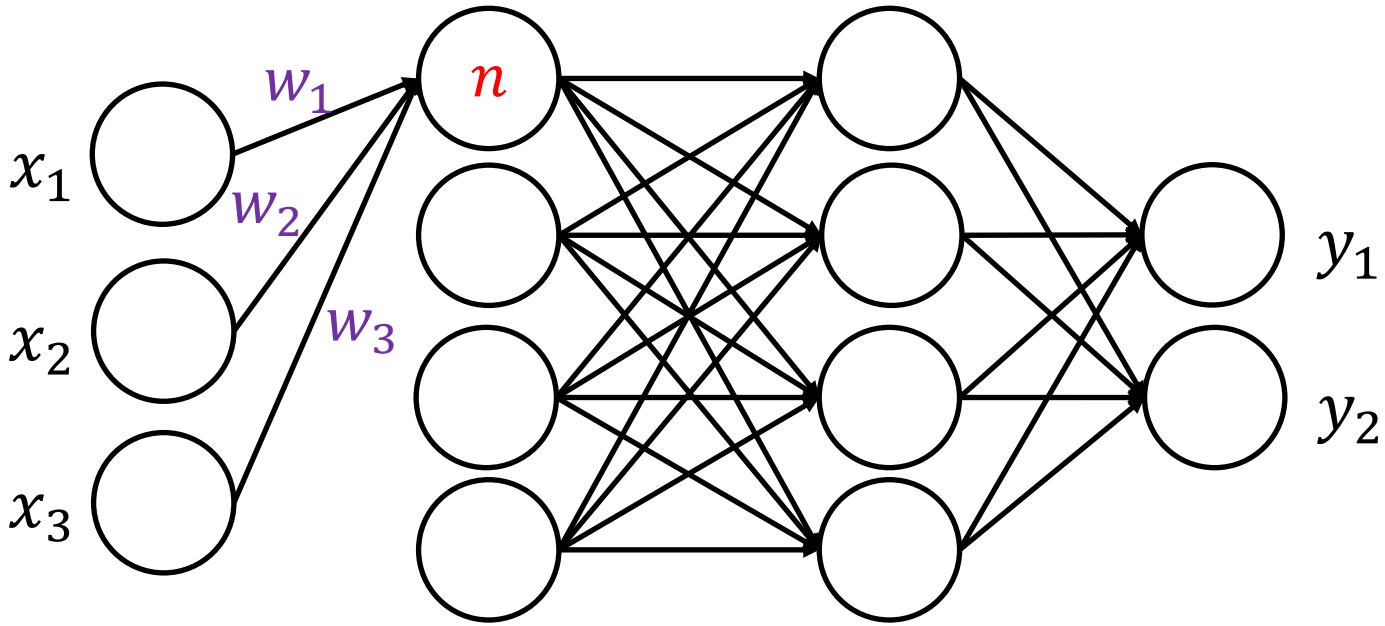
- Can be written as a composition of its $L$ hidden layers

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = f_L \circ f_{L-1} \circ \cdots \circ f_1(\boldsymbol{x})$$

# Make each layer linear?



# Make each layer linear?



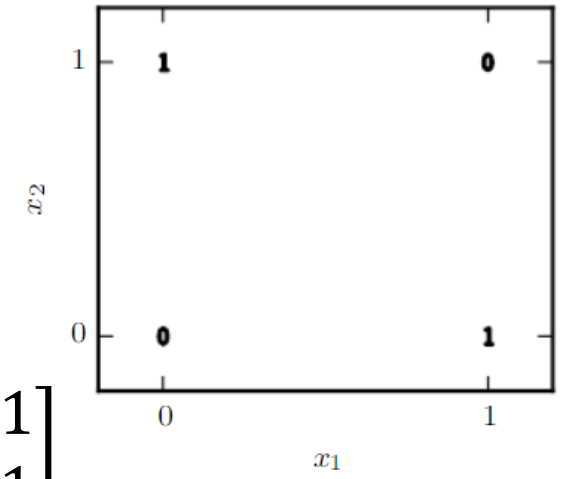

**What's wrong with this?**

# Limitations of linear models

- Learning XOR function with a linear classifier
  - Data is $\{((0,0),0),((0,1),1),((1,0),1),((1,1),0)\}$
- Learn $y = \boldsymbol{w}^T \boldsymbol{x}$, using least squares
- Recall that

$$\boldsymbol{w}^* = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

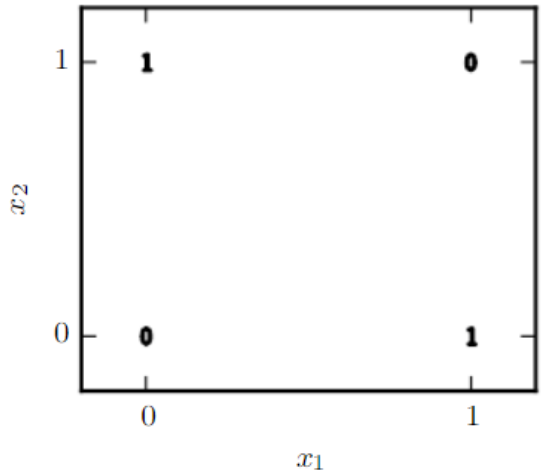
$$= \left( \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$



# Limitations of linear models, cont'd

- Learning XOR function with a linear classifier
  - Data is $\{((0,0),0),((0,1),1),((1,0),1),((1,1),0)\}$
- Learn $y = \boldsymbol{w}^T \boldsymbol{x}$, using least squares
- Recall that

$$\boldsymbol{w}^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$
$$= \left(\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}\right)^{-1}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$= \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \end{bmatrix}$$



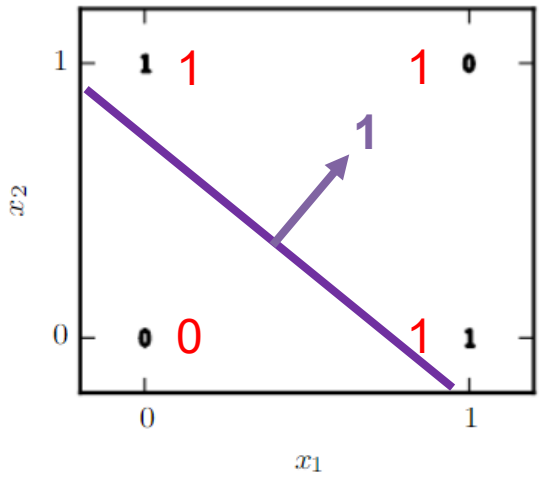

Output set is $\{0, 1/3, 1/3, 2/3\}$

# Limitations of linear models, cont'd

- Learning XOR function with a linear classifier
  - Data is $\{((0,0),0),((0,1),1),((1,0),1),((1,1),0)\}$
- Learn $y = \boldsymbol{w}^T \boldsymbol{x}$, using least squares
- Recall that

$$\boldsymbol{w}^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$
$$= \left(\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}\right)^{-1}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$= \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \end{bmatrix}$$

Threshold at 1/4

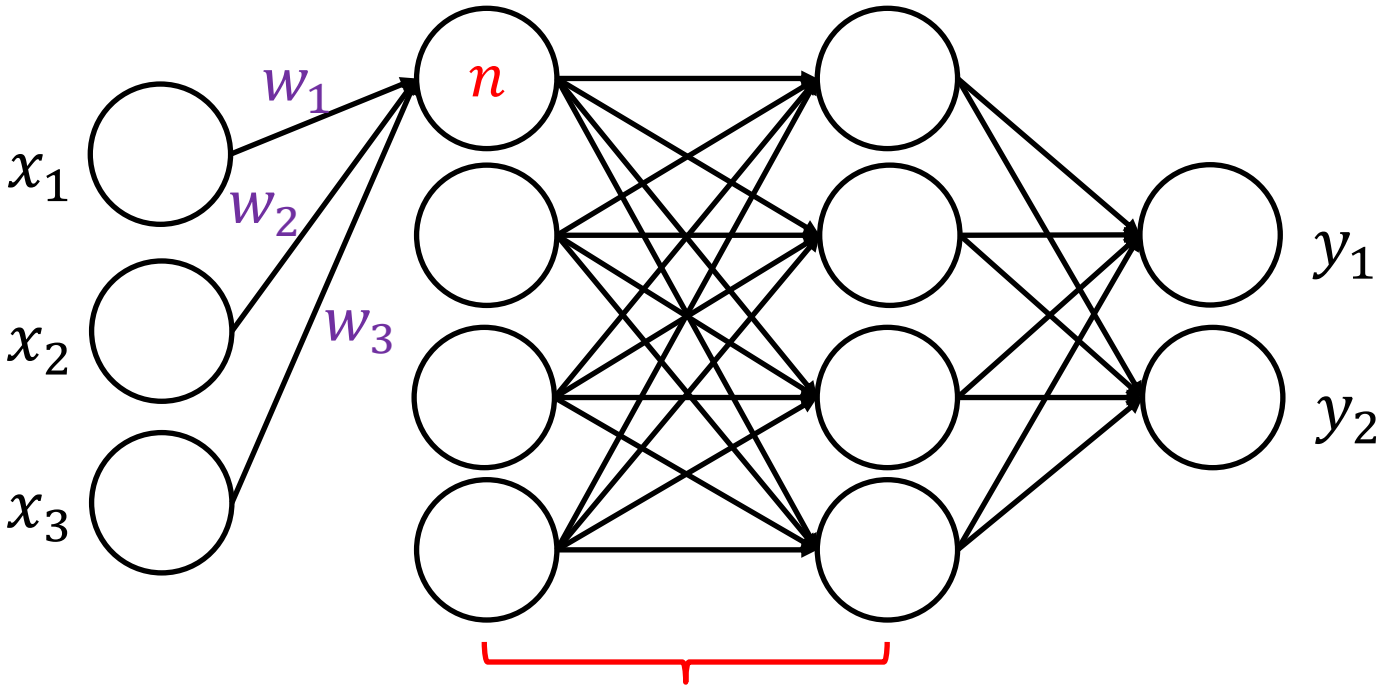



Output set is $\{0, 1/3, 1/3, 2/3\}$

Could output {0,1} by thresholding

For any threshold, at least one mistake

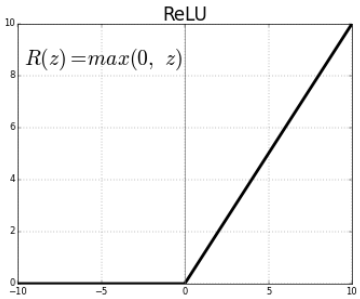
# Add small non-linearity

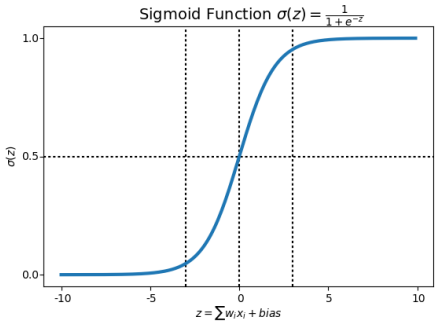



$$n = a(w_1x_1 + w_2x_2 + w_3x_3)$$

Common activations include:

- Relu: $a(x) = \max(0, x)$
- sigmoid: $a(x) = \frac{1}{1+e^{-x}}$

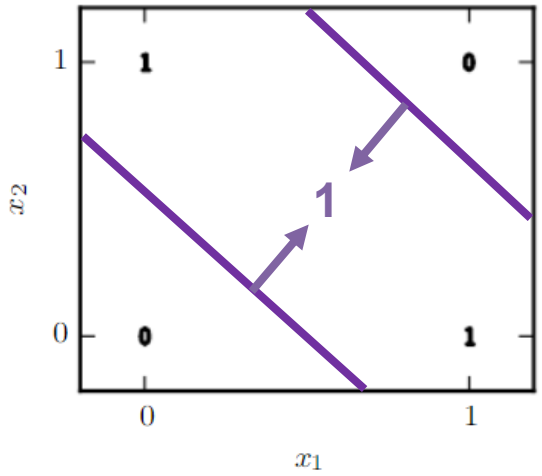


# NNs for XOR

- Consider the NN

$$f(\boldsymbol{x}) = [1 \quad -2] * ReLU\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\boldsymbol{x} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right)$$




With a threshold of 0.5

No linear model can learn this decision space

# Why neural networks?

- Universal function approximators[1]
  - Given enough neurons (even with a single layer), a NN can approximate any continuous function
  - Many function classes have this property, however
- Quick training
  - Computing derivates is very efficient on GPUs (more later)
- They work well in practice
  - Often, no setup is necessary (no need to design special features, losses)

[1]Hornik, Kurt; Tinchcombe, Maxwell; White, Halbert (1989). Multilayer Feedforward Networks are Universal Approximators (PDF). Neural Networks. 2. Pergamon Press. pp. 359–366.

# Neural Network Design: Architecture Choice

- "Architecture" refers to the overall number of layers, neurons, connections and activation functions
- So far, we've only seen fully-connected NNs
  - We'll also discuss convolutional NNs (CNNs)
  - Many, many other classes of NNs
- Most NN architectures are universal approximators
  - So why choose one over others?
- Some architectures more efficient for certain tasks
  - Convolution is good for detecting edges/obstacles in images
  - Recurrent architectures have state (e.g., good for language)

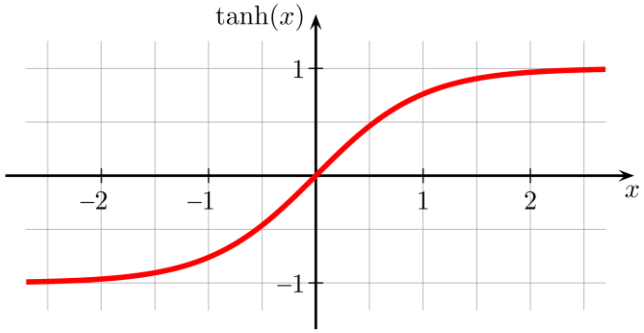
# Fully-Connected Architecture Choice

- Even if using a fully-connected NN, there's still a lot of choice
  - How many neurons? How many layers? How to distribute neurons across layers?
- If you're having trouble training the network, the issue is rarely the architecture
  - Maybe the features aren't sufficiently descriptive
  - Maybe the features need to be normalized
  - Maybe you need more data
  - **Always start with small and simple architectures!**
    - 2 layers of 100 neurons each will get you far in life
    - Once you understand the problem, you can make the architecture more complex
    - Don't expect gains from bigger architectures simply due to size

# NN Design: Activation Function

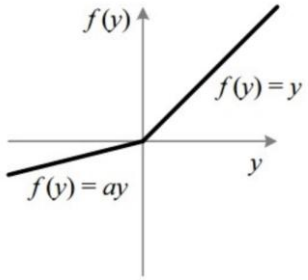
- No general consensus on choice of activation function since all are universal approximators
- ReLUs are most widely used due to their simplicity and efficient training
- Sigmoids are the original activation function
  - tanh is closely related

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



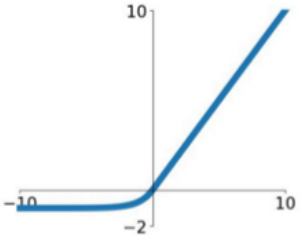
# More Activation Functions

- leaky ReLU

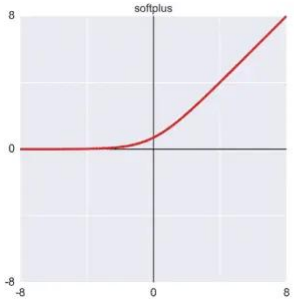



- ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$




- Softplus




$$f(x) = ln(1 + e^x)$$

## Choosing the right activation function

- ReLUs are usually the default choice since computing gradients is very efficient
  - However, more prone to vanishing gradients sometimes
- Leaky ReLUs, ELUs and others try to solve ReLU's vanishing gradient problem, but are not as widely used
- Sigmoid/tanh have gone slightly out of fashion for very deep neural networks
  - Mostly due to slow training, but also slightly worse performance

## NN Design: Output layer

- Output layer depends on the learning task and loss
- If task is regression, a linear last layer may be OK
  - Last layer similar to linear regression
  - Hidden layers transform features into linearly separable features
- If task is classification, typically one has as many output neurons as there are classes
  - How do we use such an output layer for classification?
  - Pick the neuron with highest value
- Given an input $\boldsymbol{x}$, let $F(\boldsymbol{x}) \in \mathbb{R}^L$ be the output layer
  - The NN's output is then

$$f(\boldsymbol{x}) = argmax_i \, F(\boldsymbol{x})$$

## Softmax output layer

- Often, we not only want to predict a label but we also want to predict the probabilities of each class, given an input $\boldsymbol{x}$
- With a pure linear layer, it is hard to enforce this property
- How can one do it in the case of 2 labels?
  - Logistic regression, i.e., one output neuron with a sigmoid activation
- How about multiple labels?
- Softmax!
  - Generalization of sigmoid to multi-label classification
- For an input $\boldsymbol{x}$, let $z_i = F_i(\boldsymbol{x})$ be the $i^{th}$ output neuron. Then

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

## Softmax output layer, cont'd

- Softmax normalizes last layer such that
  - all values are between 0 and 1
  - all values sum up to 1
  - essentially outputs are probabilities for each label
    - Though probabilities are often miscalibrated
- Softmax also makes training easier since it's a smooth function
- Will talk more about training later

# Training NNs: the Loss Function

- For any classifier type, we want to find the specific function that best fits the training data
- The loss function formalizes this goal during training
- There are a few popular loss functions
  - Least squares (more common for regression tasks)

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} (y_i - f(\boldsymbol{x}_i; \boldsymbol{\theta}))^2$$

  - Negative log likelihood (NLL):

$$\min_{\boldsymbol{\theta}} - \sum_{i=1}^{N} \log(\mathbb{P}_{model}[y_i | \boldsymbol{x}_i]; \boldsymbol{\theta})$$

# Maximizing data likelihood

- Suppose we want the NN to learn the true $\mathbb{P}[Y = y|\boldsymbol{X} = \boldsymbol{x}]$
  - i.e., $F_y(\boldsymbol{x}) \approx \mathbb{P}[Y = y|\boldsymbol{X} = \boldsymbol{x}]$
  - where $F_y$ is the $y^{th}$ output (softmax) neuron
- Given a dataset, the true conditional likelihood decomposes as

$$\mathbb{P}[y_1, \dots, y_N|\boldsymbol{x}_1, \dots, \boldsymbol{x}_N] = \prod_{i=1}^{N} \mathbb{P}[y_i|\boldsymbol{x}_i]$$

  - Data is IID
- Pick the NN that maximizes the conditional likelihood of the data (predicted by the model):

$$argmax_{\boldsymbol{\theta}} \prod_{i=1}^{N} \mathbb{P}_{model}[y_i|\boldsymbol{x}_i; \boldsymbol{\theta}]$$

# Maximizing log likelihood

- Instead of maximizing the likelihood, we are actually going to maximize the logarithm of likelihood

$$LL = \log\left(\prod_{i=1}^{N} \mathbb{P}_{model}[y_i|\boldsymbol{x}_i;\boldsymbol{\theta}]\right)$$

- Claim: the $\boldsymbol{\theta}$ that maximizes the likelihood also maximizes the log-likelihood
  - Why?
  - Logarithm is monotonic
  - So maximizing the log-likelihood is the same as maximizing the likelihood

# Minimizing the negative log-likelihood

$$LL = \log\left(\prod_{i=1}^{N} \mathbb{P}_{model}[y_i|\boldsymbol{x}_i;\boldsymbol{\theta}]\right) = \sum_{i=1}^{N} \log(\mathbb{P}_{model}[y_i|\boldsymbol{x}_i;\boldsymbol{\theta}])$$

- Finally, ML people like to minimize stuff, so we negate the LL

$$NLL = -\sum_{i=1}^{N} \log(\mathbb{P}_{model}[y_i|\boldsymbol{x}_i;\boldsymbol{\theta}])$$

$$= -\sum_{i=1}^{N} \log\left(F_{y_i}(\boldsymbol{x}_i);\boldsymbol{\theta}\right)$$

- This is the negative log-likelihood loss

# Other Loss Functions: Cross-entropy

- Cross-entropy between "training data" distribution and predicted NN distribution
- The "training data" distribution is just a uniform distribution over the training data, i.e.,

$$\mathbb{P}_{data}[(\boldsymbol{X} = \boldsymbol{x}_i, Y = y_i)] = \frac{1}{N}, \forall i$$

- Similarly, the conditional "data" distribution is

$$\mathbb{P}_{data}[Y = y_i | \boldsymbol{X} = \boldsymbol{x}_i] = 1$$

  - And $\mathbb{P}_{data}[Y = j | \boldsymbol{X} = \boldsymbol{x}_i] = 0$ for $j \neq y_i$

- The predicted NN (conditional) distribution is the output (softmax) layer

$$\mathbb{P}_{model}[Y = y_i | \boldsymbol{X} = \boldsymbol{x}_i] = F_{y_i}(\boldsymbol{x}_i)$$

## Cross entropy, cont'd

- The cross-entropy loss is defined as

$$H(\mathbb{P}_{data}, \mathbb{P}_{model}) = -\sum_{(\boldsymbol{x}_i, y_i)} \mathbb{P}_{data}[y_i|\boldsymbol{x}_i] \log\left[\mathbb{P}_{model}[y_i|\boldsymbol{x}_i]\right]$$

- Correspondingly, the minimization problem is

$$\min_{\boldsymbol{\theta}} -\sum_{(\boldsymbol{x}_i, y_i)} \mathbb{P}_{data}[y_i|\boldsymbol{x}_i] \log\left[F_{y_i}(\boldsymbol{x}_i; \boldsymbol{\theta})\right]$$

- Note that this is the same as NLL
  - First note that $\mathbb{P}_{data}(y_i|\boldsymbol{x}_i) = 1$ by definition
    - Since $\mathbb{P}_{data}[Y \neq y_i|\boldsymbol{X} = \boldsymbol{x}_i] = 0$
  - Thus the loss becomes

$$-\sum_{(\boldsymbol{x}_i, y_i)} \log\left[F_{y_i}(\boldsymbol{x}_i; \boldsymbol{\theta})\right] = -\sum_{(\boldsymbol{x}_i, y_i)} \log(\mathbb{P}_{model}[y_i|\boldsymbol{x}_i]; \boldsymbol{\theta})$$

## Putting it all together

- In a supervised classification task, you are given a labelled dataset $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$
- To train a NN classifier, perform the following tasks:

1. Pick a NN architecture
2. Pick a training loss
3. Pick a training algorithm (more on this next)
4. Iterate on the above depending on where improvements are necessary

- Most of these details are handled by the deep learning libraries, but it's important to understand what happens under the hood