# Tutorial Introduction to Neural Networks with an eye towards linguistic applications

Shane Steinert-Threlkeld

January 25, 2019

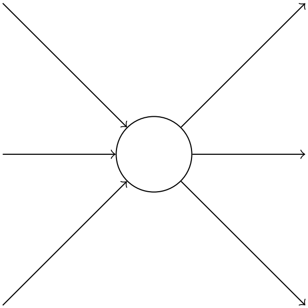
European Research Council
Established by the European Commission

# Today's Plan

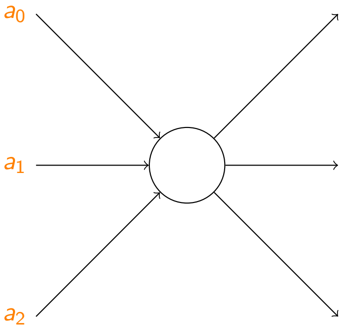
# Materials: Slides + Jupyter Notebook

https://github.com/shanest/nn-tutorial

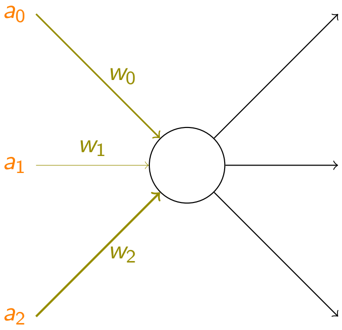
# Artificial Neuron



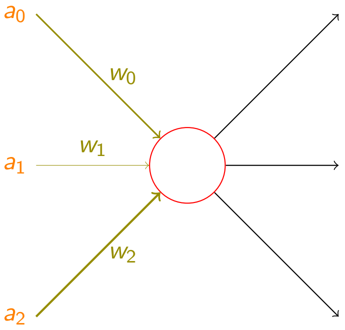
# Artificial Neuron



# Artificial Neuron

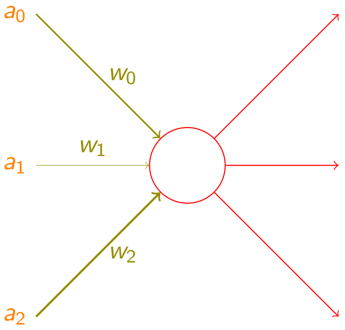


# Artificial Neuron



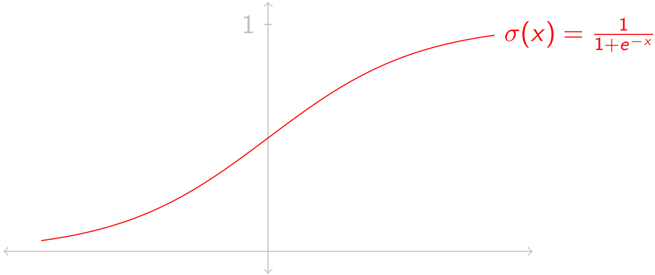

$$a = f(a_0 \cdot w_0 + a_1 \cdot w_1 + a_2 \cdot w_2)$$

# Artificial Neuron




$$a = f(a_0 \cdot w_0 + a_1 \cdot w_1 + a_2 \cdot w_2)$$

# Activation Function

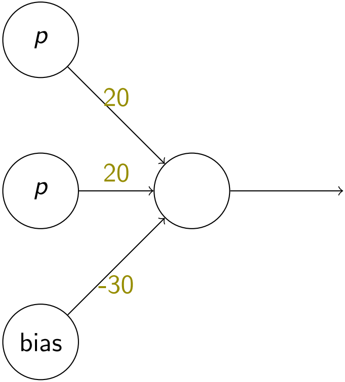



More on choosing activation functions later in the tutorial.

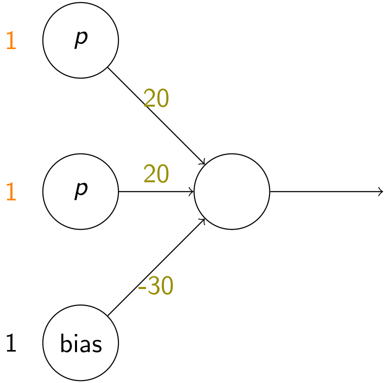
# Computing 'and'

| $p$ | $q$ | $p \wedge q$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

# Computing ‘and’

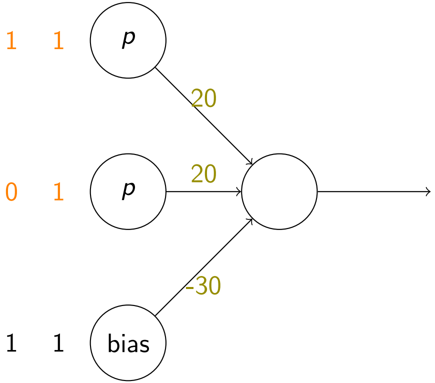


# Computing ‘and’




$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

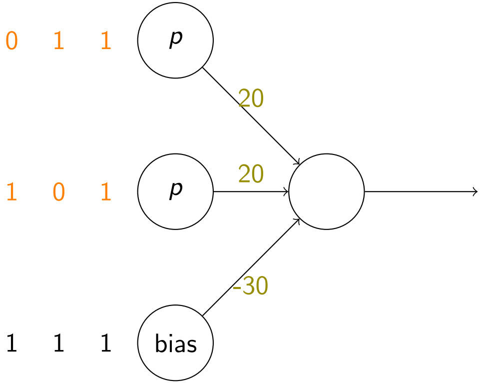
# Computing ‘and’




$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

$$a = \sigma(1 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

# Computing 'and'

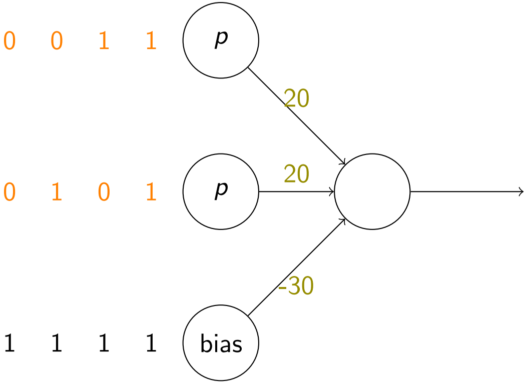



$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

$$a = \sigma(1 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

$$a = \sigma(0 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

# Computing 'and'



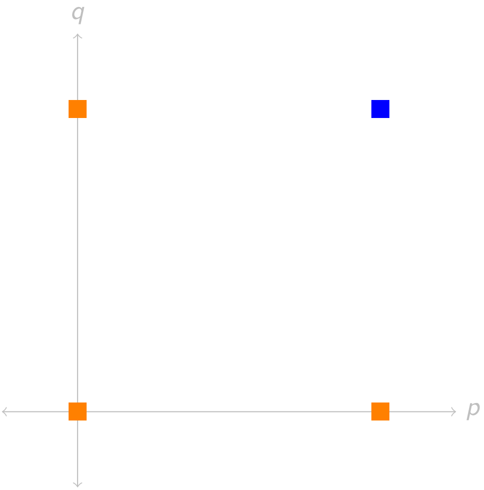

$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

$$a = \sigma(1 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

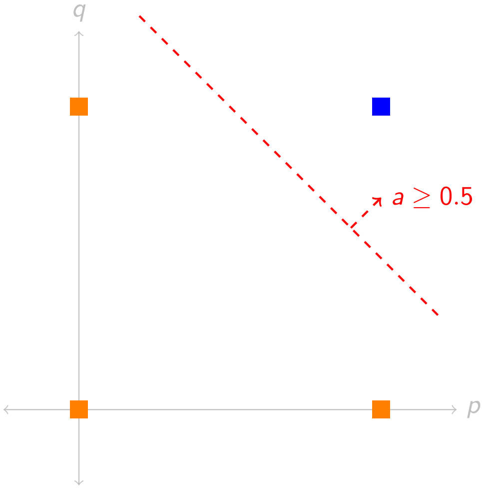
$$a = \sigma(0 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

$$a = \sigma(0 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-30) \approx 0$$

# Computing ‘and’



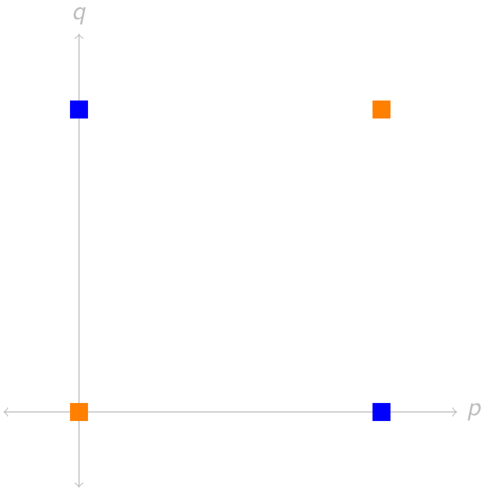
# Computing 'and'



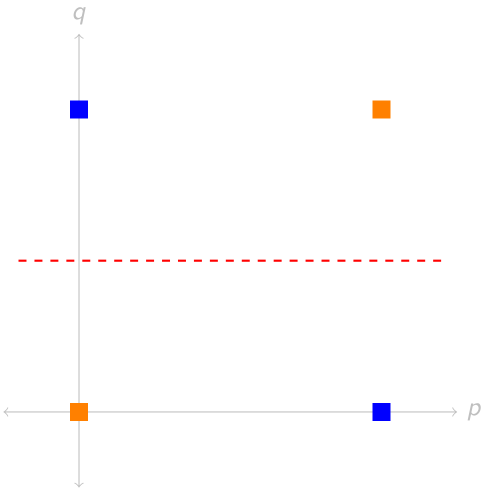
# Computing 'xor'

| $p$ | $q$ | $p$ xor $q$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

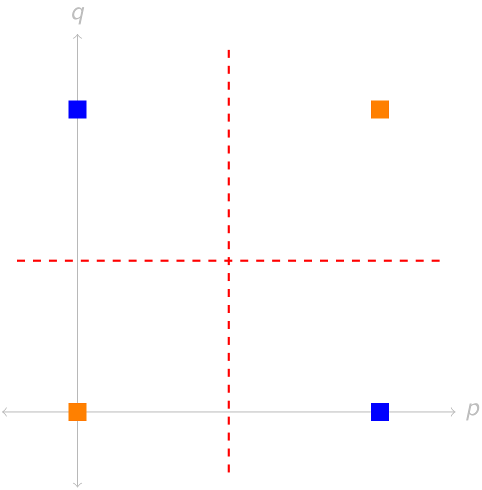
# Computing ‘xor’



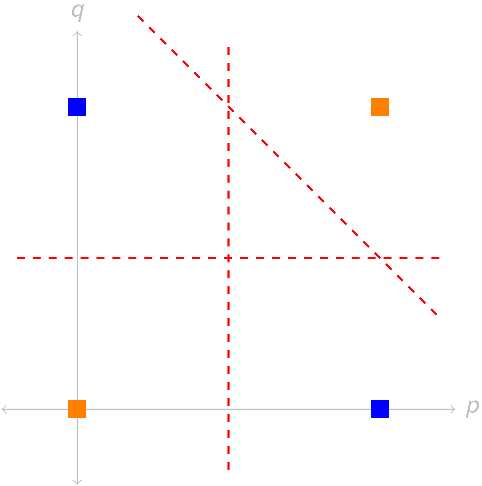
# Computing 'xor'



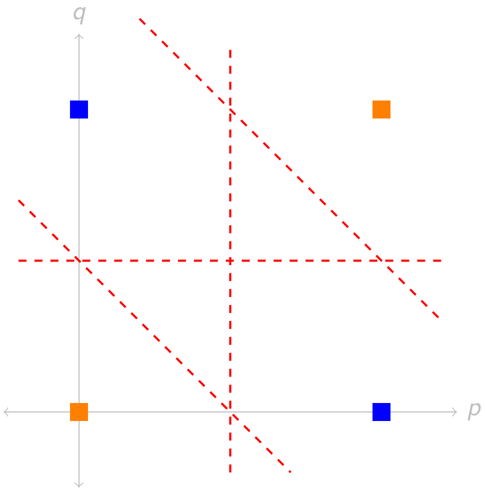
# Computing ‘xor’



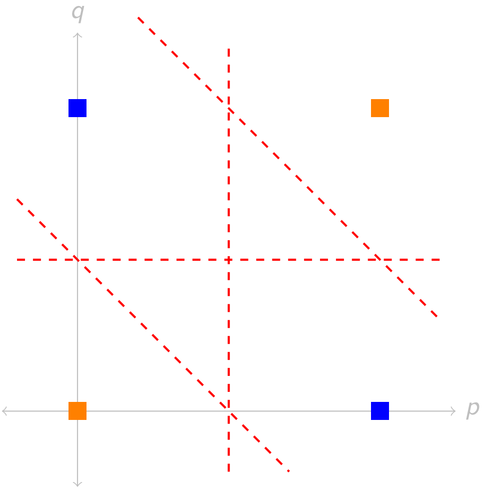
# Computing ‘xor’



# Computing 'xor'

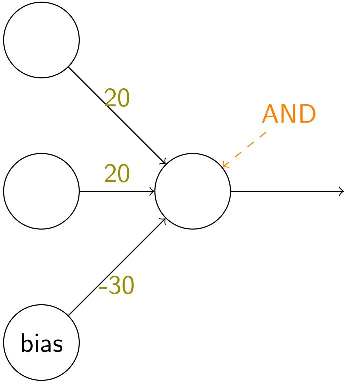


# Computing ‘xor’

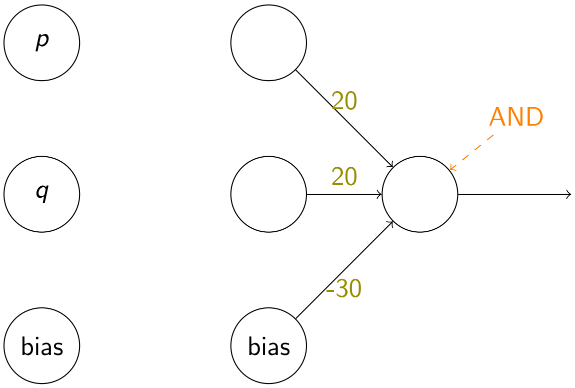



xor is not *linearly separable*

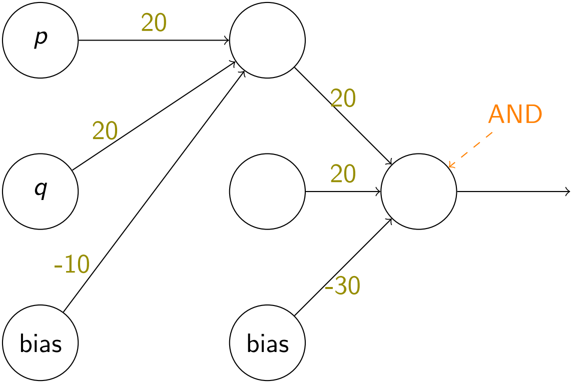
# Computing ‘xor’



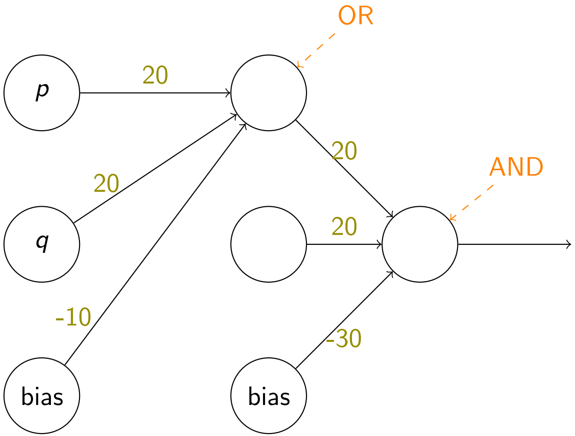
# Computing ‘xor’



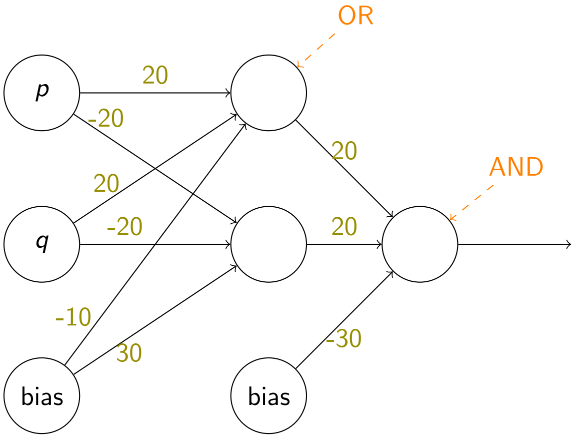
# Computing 'xor'



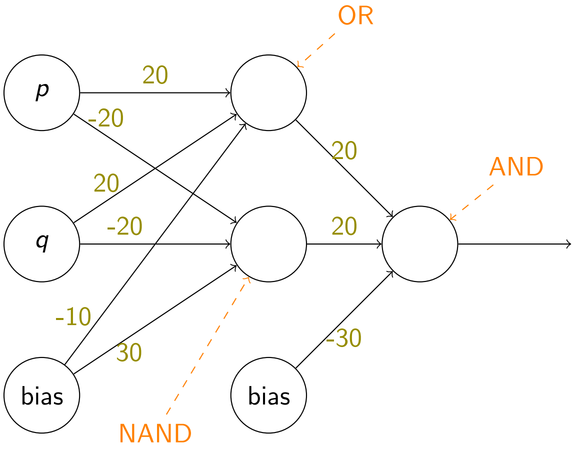
# Computing 'xor'



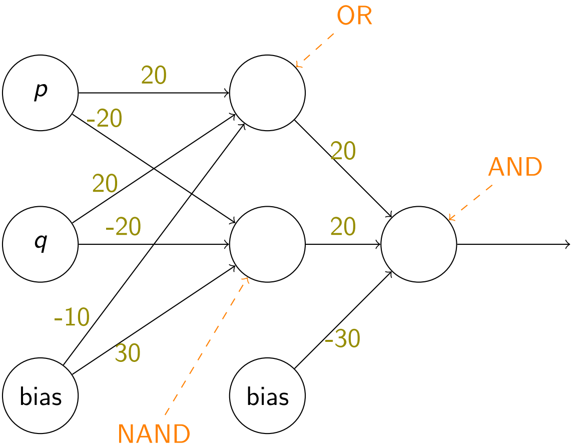
# Computing 'xor'



# Computing 'xor'

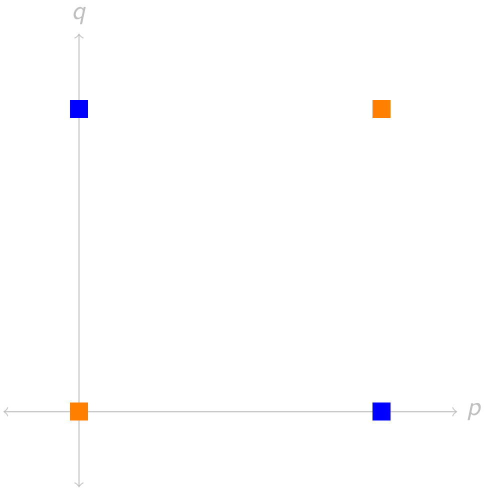


# Computing 'xor'

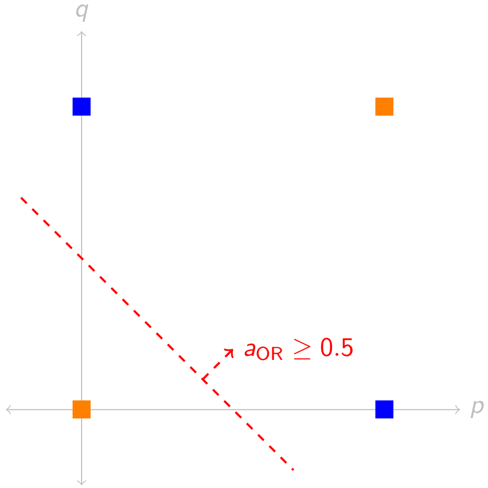



Exercise: show that the hidden units behave as labeled.

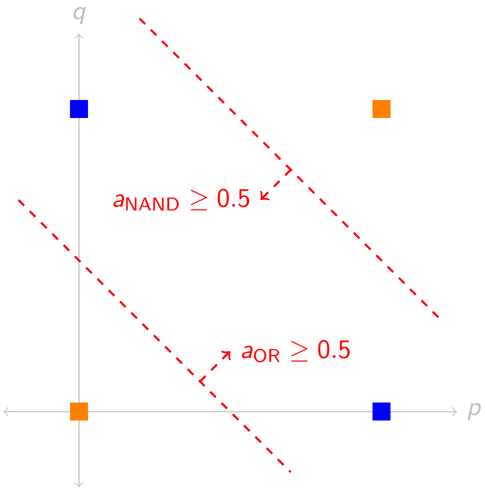
# Computing 'xor'



# Computing ‘xor’



# Computing 'xor'



# Computing Many Examples

It's often very useful to compute over a 'batch' of inputs at once. The linear combination then turns into a *matrix multiplication*:

$$\vec{a} = f\left(\begin{bmatrix} x_0^0 & x_1^0 & \cdots & x_n^0 \\ x_0^1 & x_1^1 & \cdots & x_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} w_0^0 & w_0^1 & \cdots & w_0^l \\ w_1^0 & w_1^1 & \cdots & w_1^l \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^1 & \cdots & w_n^l \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_l \\ b_0 & b_1 & \cdots & b_l \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_l \end{bmatrix}\right)$$

$$= f(xW + b)$$

# Computing Many Examples

It's often very useful to compute over a 'batch' of inputs at once. The linear combination then turns into a *matrix multiplication*:

$$\vec{a} = f\left(\begin{bmatrix} x_0^0 & x_1^0 & \cdots & x_n^0 \\ x_0^1 & x_1^1 & \cdots & x_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} w_0^0 & w_0^1 & \cdots & w_0^l \\ w_1^0 & w_1^1 & \cdots & w_1^l \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^1 & \cdots & w_n^l \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_l \\ b_0 & b_1 & \cdots & b_l \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_l \end{bmatrix}\right)$$
$$= f(xW + b)$$

- $x_j^i$: $j$th feature of input $i$
- $w_k^l$: weight from neuron $k$ to neuron $l$ in next layer
- $b_m$: bias to neuron $m$ in next layer

# Computing Many Examples

It's often very useful to compute over a 'batch' of inputs at once. The linear combination then turns into a *matrix multiplication*:

$$\vec{a} = f\left(\begin{bmatrix} x_0^0 & x_1^0 & \cdots & x_n^0 \\ x_0^1 & x_1^1 & \cdots & x_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} w_0^0 & w_0^1 & \cdots & w_0^l \\ w_1^0 & w_1^1 & \cdots & w_1^l \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^1 & \cdots & w_n^l \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_l \\ b_0 & b_1 & \cdots & b_l \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_l \end{bmatrix}\right)$$
$$= f(xW + b)$$

- $x_j^i$: $j$th feature of input $i$
- $w_k^l$: weight from neuron $k$ to neuron $l$ in next layer
- $b_m$: bias to neuron $m$ in next layer

Exercises:

- write down $W^1$ and $W^2$ for the xor network.
- re-write the above as $f(xW)$ by adding a column of 1s to $x$ and a new row to $W$.

# (Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

# (Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.
Supervised learning (will talk about others later):

- Initialize the network randomly.

# (Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.

# (Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.
Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.

# (Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.
Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.
- Update the weights and biases to move the network's outputs closer to the true outputs.

# (Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.
Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

# Gradient Descent: Example

Task: predict a true value $y = 2$.

# Gradient Descent: Example

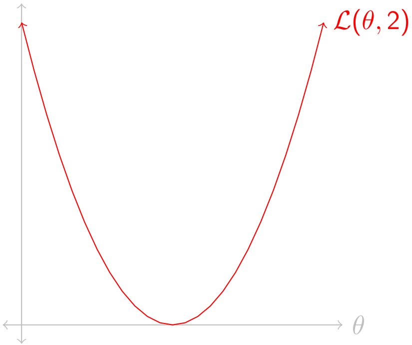
Task: predict a true value $y = 2$.
"Model": one parameter $\theta$, outputs $\hat{y} = \theta$.

# Gradient Descent: Example

Task: predict a true value $y = 2$.
"Model": one parameter $\theta$, outputs $\hat{y} = \theta$.
Loss function:

$$\mathcal{L}(\theta, y) = (\hat{y}(\theta) - y)^2$$

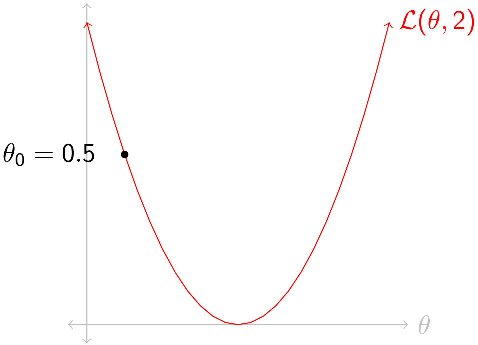
# Gradient Descent: Example




$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

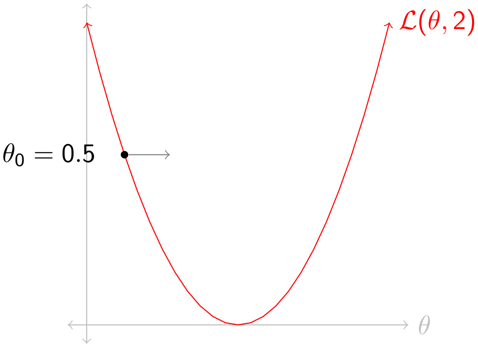
# Gradient Descent: Example




$$\frac{\partial}{\partial\theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial\theta}\mathcal{L}(\theta, y)$$

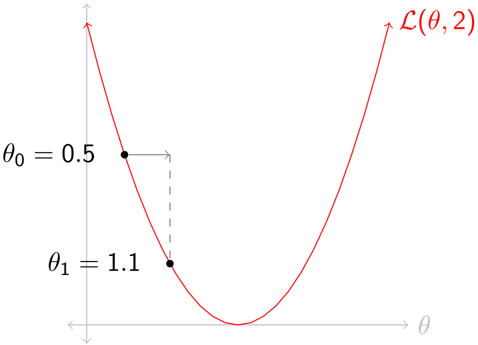
# Gradient Descent: Example




$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

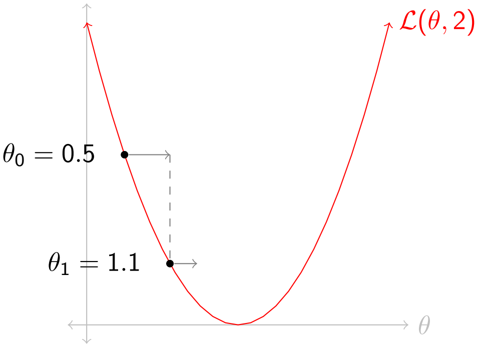
# Gradient Descent: Example




$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

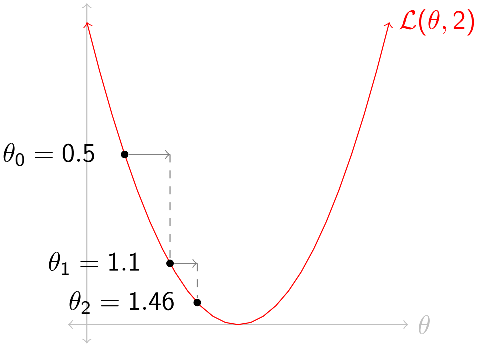
# Gradient Descent: Example




$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

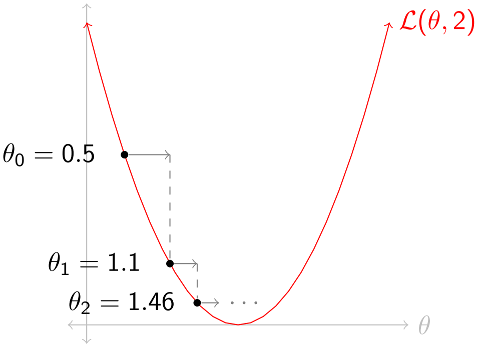
# Gradient Descent: Example




$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

# Gradient Descent: Example




$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

# Gradient Descent for NNs

A neural network computes a complex function of its input. For an $L$-layer feed-forward network:

$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

# Gradient Descent for NNs

A neural network computes a complex function of its input. For an $L$-layer feed-forward network:

$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

All of the weights and biases form a long vector of parameters $\theta$. So instead of a partial derivative, we take a *gradient*:

$$\nabla_\theta \mathcal{L}(\hat{y}(\theta), y) = \left\langle \frac{\partial}{\partial \theta_1}\mathcal{L}, \ldots, \frac{\partial}{\partial \theta_N}\mathcal{L} \right\rangle$$

# Gradient Descent for NNs

A neural network computes a complex function of its input. For an $L$-layer feed-forward network:

$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

All of the weights and biases form a long vector of parameters $\theta$. So instead of a partial derivative, we take a *gradient*:

$$\nabla_\theta \mathcal{L}(\hat{y}(\theta), y) = \left\langle \frac{\partial}{\partial\theta_1}\mathcal{L}, \ldots, \frac{\partial}{\partial\theta_N}\mathcal{L} \right\rangle$$

The (negative) gradient tells us *which direction in 'parameter space'* to walk in order to make the loss ($\mathcal{L}$) smaller, i.e. to make the network's output closer to the true output.

# Anatomy of a DL Experiment

1. Specify parameters

# Anatomy of a DL Experiment

1. Specify parameters
2. Build data input/generation pipeline

# Anatomy of a DL Experiment

1. Specify parameters
2. Build data input/generation pipeline
   - Train/test split [dev as well; more later]
3. Build model

# Anatomy of a DL Experiment

1. Specify parameters
2. Build data input/generation pipeline
   - Train/test split [dev as well; more later]
3. Build model
4. Train the model!

# Anatomy of a DL Experiment

1. Specify parameters
2. Build data input/generation pipeline
   - Train/test split [dev as well; more later]
3. Build model
4. Train the model!
   1. Evaluate at regular intervals
   2. Monitor train/test loss
   3. Early stopping [later in tutorial]

# Anatomy of a DL Experiment

1. Specify parameters
2. Build data input/generation pipeline
   - Train/test split [dev as well; more later]
3. Build model
4. Train the model!
   1. Evaluate at regular intervals
   2. Monitor train/test loss
   3. Early stopping [later in tutorial]

NOTE: keep detailed records about what you're doing!

# To the code!

https://github.com/shanest/nn-tutorial/blob/master/tutorial.ipynb

# References I