Learning from Data

Lecture 3: Linear Models, Neural Networks and Word Embeddings

Rik van Noord - 18 September 2023

Previous weeks: Classification

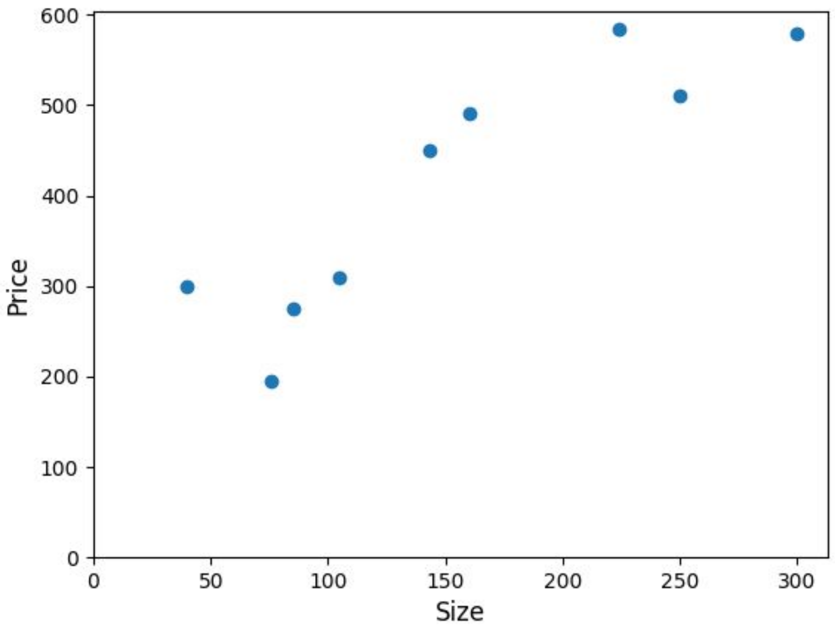
New type of problem: predicting continuous values

Regression

Example data - house prices

Size (m^2)	Price (x1000)
40	300
76	195
85	275
105	310
143	450
160	491
224	585
250	510
300	580

Example data - house prices



Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Why do we need b_0?

Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Why do we need b_0?

- Otherwise Y would be 0 when X=0, this is often not the case

Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Why do we need b_0?

- Otherwise Y would be 0 when X=0, this is often not the case

How do we set b_0 and b_1?

Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Why do we need b_0?

- Otherwise Y would be 0 when X=0, this is often not the case

How do we set b_0 and b_1?

- Depends on the algorithm we use

Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Why do we need b_0?

- Otherwise Y would be 0 when X=0, this is often not the case

How do we set b_0 and b_1?

- Depends on the algorithm we use

How do we use multiple features?

Linear model

- $Y = b_0 + b_1 * X_1$
- **Or:** Price = constant value + (weight * size)

Why do we need b_0?

- Otherwise Y would be 0 when X=0, this is often not the case

How do we set b_0 and b_1?

- Depends on the algorithm we use

How do we use multiple features?

- $Y = b_0 + b_1 * X_1 + b_2 * X_2 + \ldots + X_n * b_n$

Linear model

```
from sklearn import linear_model
```

Setup data set

```
size  = np.array([40, 76, 85, 105, 143, 160, 224, 250, 300]).reshape(-1, 1)
price = np.array([300, 195, 275, 310, 450, 491, 585, 510, 580]).reshape(-1, 1)
```

Linear model

```
from sklearn import linear_model
```

Setup data set

```
size  = np.array([40, 76, 85, 105, 143, 160, 224, 250, 300]).reshape(-1, 1)
price = np.array([300, 195, 275, 310, 450, 491, 585, 510, 580]).reshape(-1, 1)
```

Create model and make predictions on training set

```
regr = linear_model.LinearRegression()
regr.fit(size, price)
pred = regr.predict(size)
```

Linear model

```
from sklearn import linear_model
```

Setup data set

```
size  = np.array([40, 76, 85, 105, 143, 160, 224, 250, 300]).reshape(-1, 1)
price = np.array([300, 195, 275, 310, 450, 491, 585, 510, 580]).reshape(-1, 1)
```

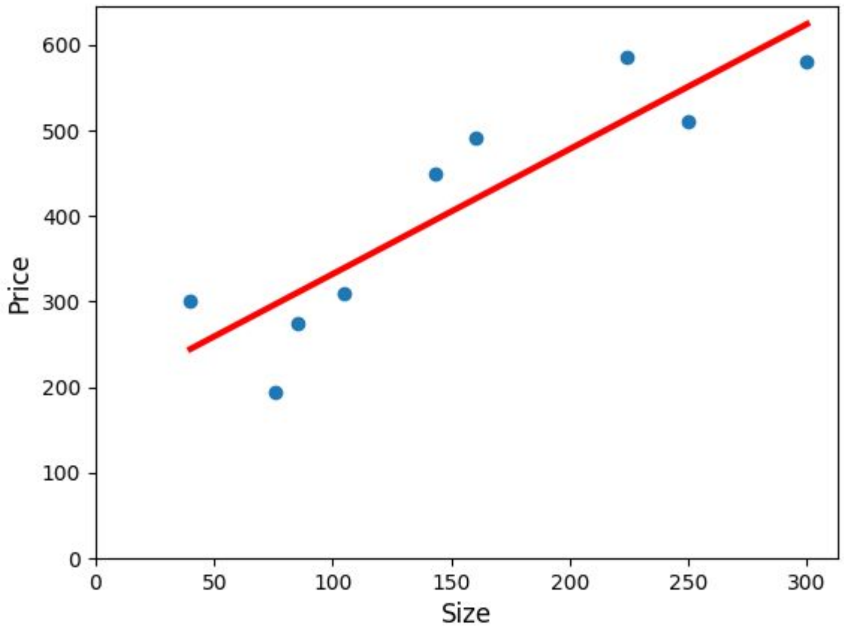
Create model and make predictions on training set

```
regr = linear_model.LinearRegression()
regr.fit(size, price)
pred = regr.predict(size)
```

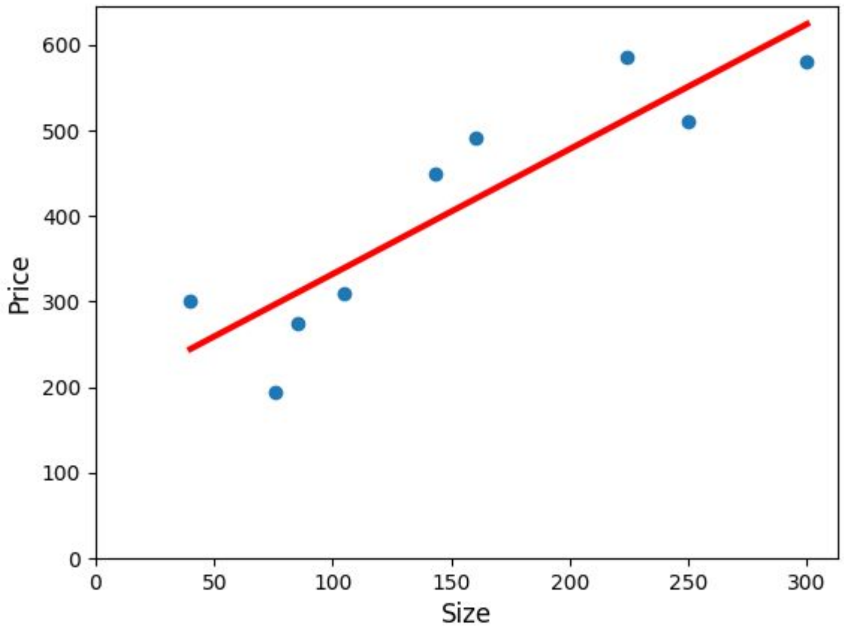
Show in plot

```
plt.scatter(size, price)
plt.plot(size, pred, color='red', linewidth=3)
plt.show()
```

Example data - house prices

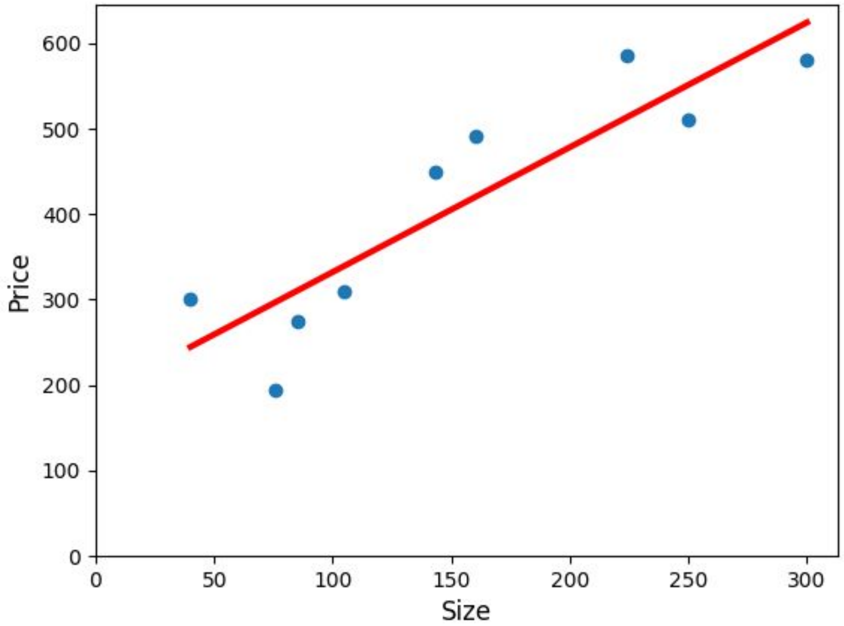


Example data - house prices



What about a new instance for size = 200?

Example data - house prices



What about a new instance for size = 200?

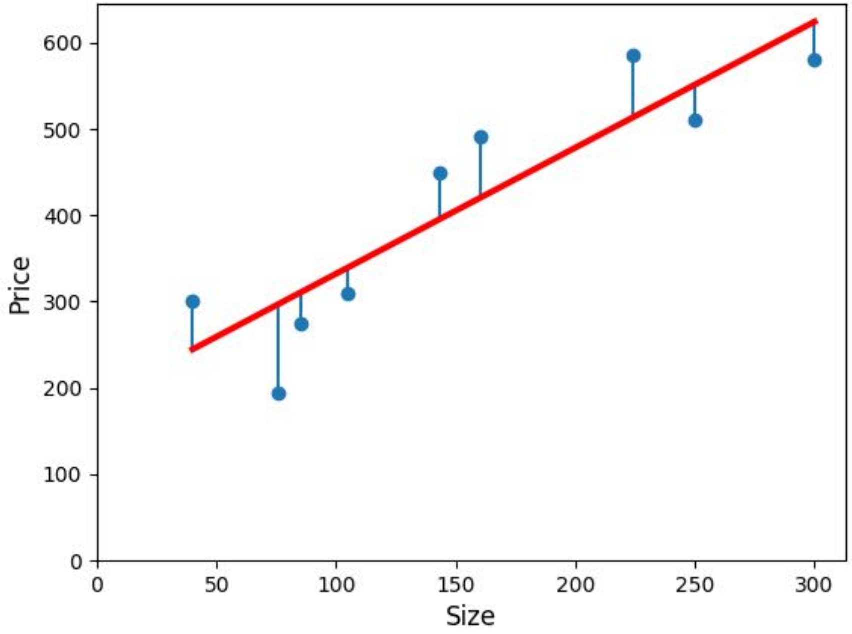
We actually have the exact formula:

$Y = b_0 + b_1 * X_1$

$Y = 186.4 + 1.46 * \text{size}$

$= 186.4 + 1.46 * 200 =$ **478**

Errors as vertical lines



Mean Squared Error (MSE)

How close is a regression line to a set of points?

General idea:

- For each distance, measure distance (=error) to the regression line
- Square the distance
    - Deal with negative numbers, give more weight to large mistakes
- Take average over all errors

The **lower** the number, the better!

Regression models

- Most algorithms will also work with regression
  - KNeighborsRegressor
  - DecisionTreeRegressor
  - RandomForestRegressor
  - svm.SVR (Support Vector Regression)

Sometimes, regression might be a **good fit** even if the data looks categorical

Regression models

- Most algorithms will also work with regression
  - KNeighborsRegressor
  - DecisionTreeRegressor
  - RandomForestRegressor
  - svm.SVR (Support Vector Regression)

Sometimes, regression might be a **good fit** even if the data looks categorical

For example for certain sentiment analysis tasks:

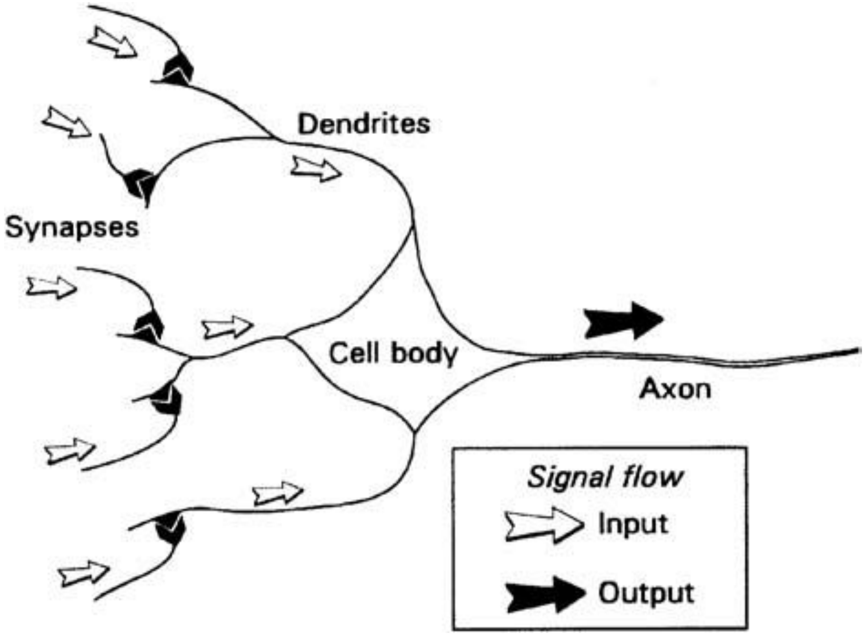
- Positive, neutral and negative can be represented as **1, 0 and -1**
- Classification models don’t know that **neutral** is closer to **positive** than **negative** is!

Neural Networks

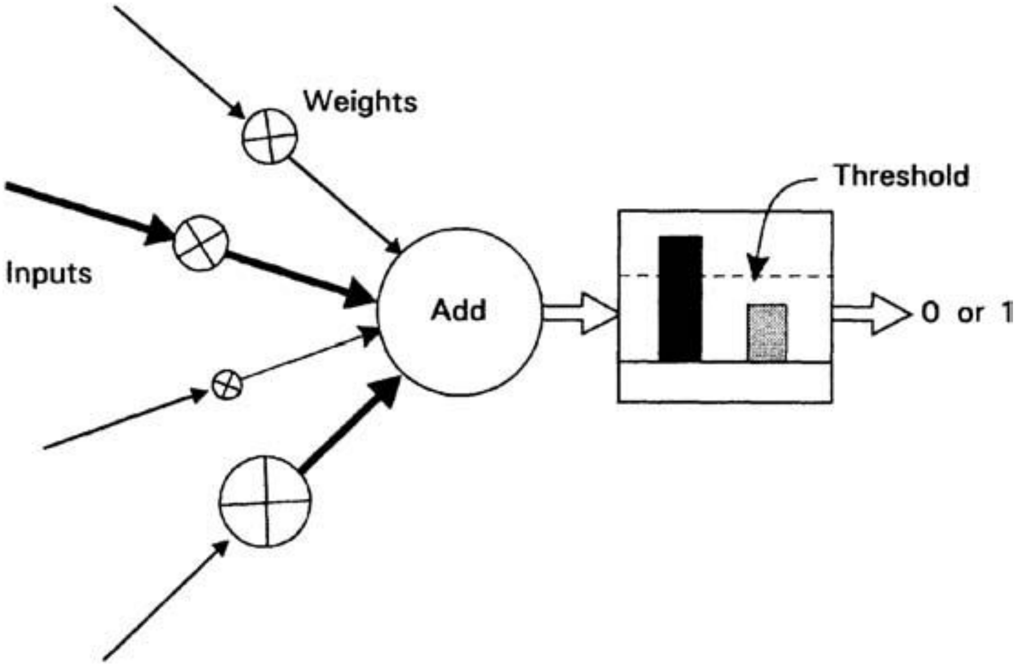
Neural Networks

- For years the dominant method in NLP
- Algorithm that works with vector spaces
- We will start with the most simple variant and gradually work our way up
- Word embeddings: treat words as vector themselves
- Next week: **deep learning**

Human Neuron



Artificial Neuron



Neural Networks

- **Intuition:** each **input** feature has a weight associated with it

Neural Networks

- **Intuition:** each **input** feature has a weight associated with it
- These weights are summed, after which we make a decision (**output**)

Neural Networks

- **Intuition:** each **input** feature has a weight associated with it
- These weights are summed, after which we make a decision (**output**)
- Remember the linear model:
  - $Y = w_0 + w_1 * X_1 + w_2 * X_2 + \ldots + X_n * w_n$

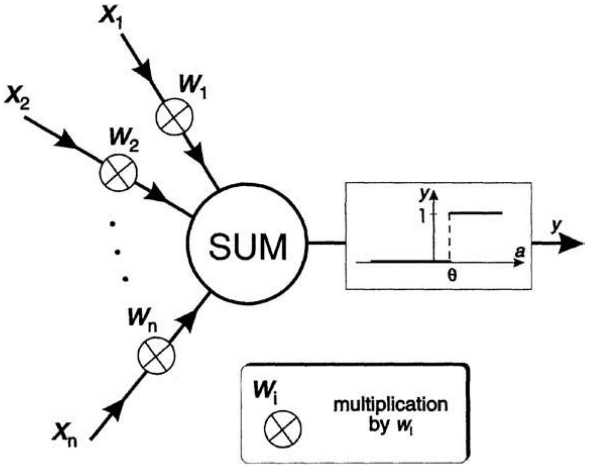
Neural Networks

- **Intuition:** each **input** feature has a weight associated with it
- These weights are summed, after which we make a decision (**output**)
- Remember the linear model:
    - $Y = w_0 + w_1 * X_1 + w_2 * X_2 + \ldots + X_n * w_n$

- The simplest neural network (perceptron) actually has the same definition
    - The difference is **how** we set the weights (w)
    - Setting the weights is what **learning** is!

Neural Networks

- **Intuition:** each **input** feature has a weight associated with it
- These weights are summed, after which we make a decision (**output**)
- Remember the linear model:
  - $Y = w_0 + w_1 * X_1 + w_2 * X_2 + \ldots + X_n * w_n$
- The simplest neural network (perceptron) actually has the same definition
  - The difference is **how** we set the weights (w)
  - Setting the weights is what **learning** is!

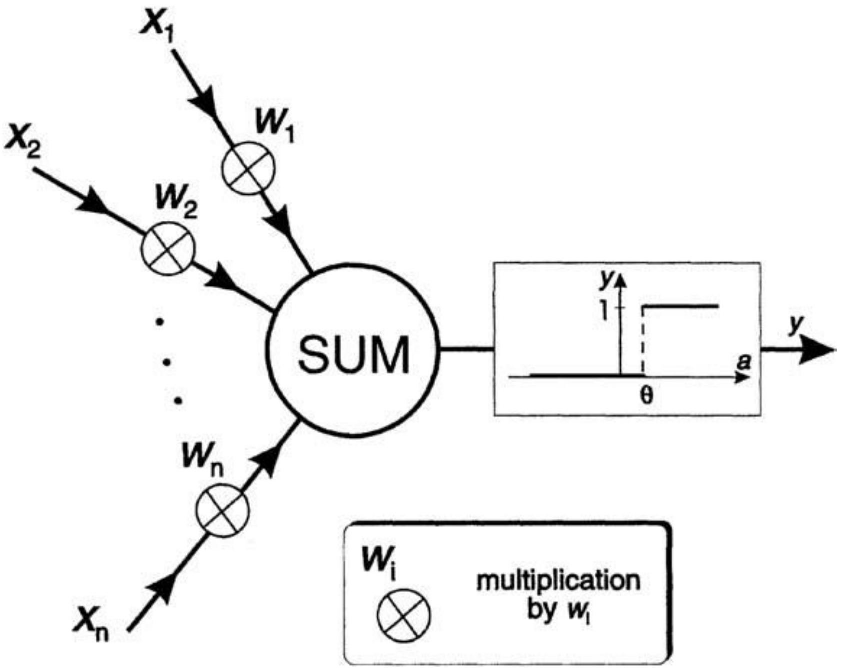


Some definitions

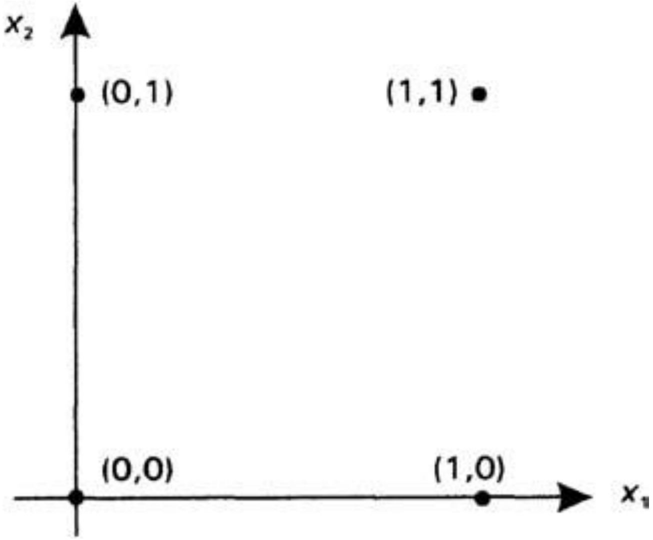
- **Input:** the features x_1, x_2, x_3
- **Weights:** w_1, w_2, w_3
- **Node:** process input (SUM)
- **Activation:** transform input
    - Decision rule

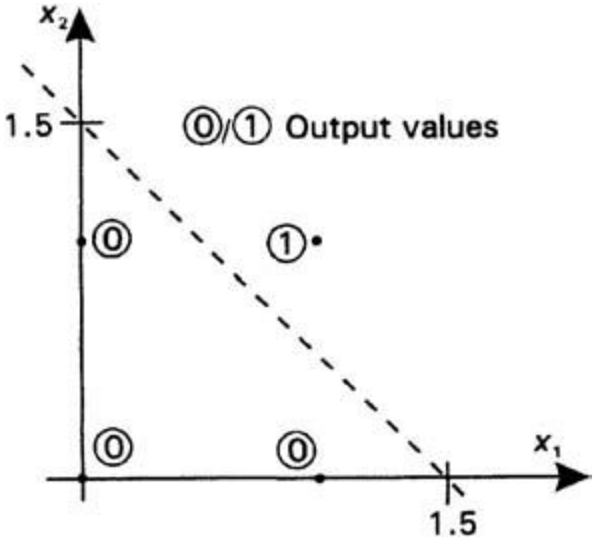
$y(x_1,x_2,x_3)$ = **1 if** $w_0 + w_1 * x_1 + w_2 * x_2 + x_3 * w_3$ **> 0**
else 0

- **Output:** final decision (y)



Classification example





Classification example

Feature 1 (X_1)	**Feature 2 (X_2)**	**Label**
0	0	0
0	1	0
1	0	0
1	1	1

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
	0		0				

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
    - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2			

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0
	0		1				

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0
0.1	0	-0.4	1	0.2			

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0
0.1	0	-0.4	1	0.2	0.3	1	0

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0
0.1	0	-0.4	1	0.2	0.3	1	0
0.1	1	-0.4	0	0.2	-0.3	0	0

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0
0.1	0	-0.4	1	0.2	0.3	1	0
0.1	1	-0.4	0	0.2	-0.3	0	0
0.1	1	-0.4	1	0.2	-0.1	0	1

Perceptron example

- **First:** weights are randomly initialized (w_0, w_1, w_2)
  - Say $w_0 = 0.1$, $w_1 = -0.4$ and $w_2 = 0.2$

Let's look at the performance of our model with these weights:

Remember: $a = w_0 + w_1 * x_1 + w_2 * x_2$ and $y = 1$ for $a > 0$ and 0 otherwise

w_0	x_1	w_1	x_2	w_2	a	y	t
0.1	0	-0.4	0	0.2	0.1	1	0
0.1	0	-0.4	1	0.2	0.3	1	0
0.1	1	-0.4	0	0.2	-0.3	0	0
0.1	1	-0.4	1	0.2	-0.1	0	1

Learning the weights

- We want to iteratively change the weights to improve our model

Learning the weights

- We want to iteratively change the weights to improve our model
- Each time we process a training instance, we want to make changes

Learning the weights

- We want to iteratively change the weights to improve our model
- Each time we process a training instance, we want to make changes
- **Intuition:** figure out who (what weights) are to blame for mistakes
  - Learning a task == learning the weights

Learning the weights

- We want to iteratively change the weights to improve our model
- Each time we process a training instance, we want to make changes
- **Intuition:** figure out who (what weights) are to blame for mistakes
    - Learning a task == learning the weights
- Compare predicted and gold values and change weights accordingly
    - Bigger contribution to error means bigger change in weight

Learning the weights

Rule: $\Delta w_i = r * (t - y) * x_i$ with: **Δw**: change in weight

r: learning rate (rate of change)

t: target (gold label)

y: output (predicted label)

x: input (feature value)

Learning the weights

Rule: $\Delta w_i = r * (t - y) * x_i$ with: **Δw**: change in weight
r: learning rate (rate of change)
t: target (gold label)
y: output (predicted label)
x: input (feature value)

- If the prediction is correct, (t-y) == 0. Which means Δw_i is zero for any i
  - No changes to any weights, the model works for this instance

Learning the weights

Rule: $\Delta w_i = r * (t - y) * x_i$ with: **Δw**: change in weight

r: learning rate (rate of change)

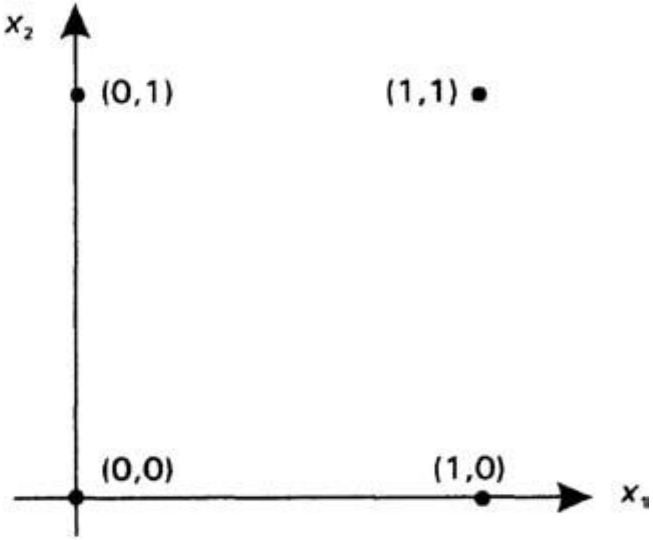
t: target (gold label)

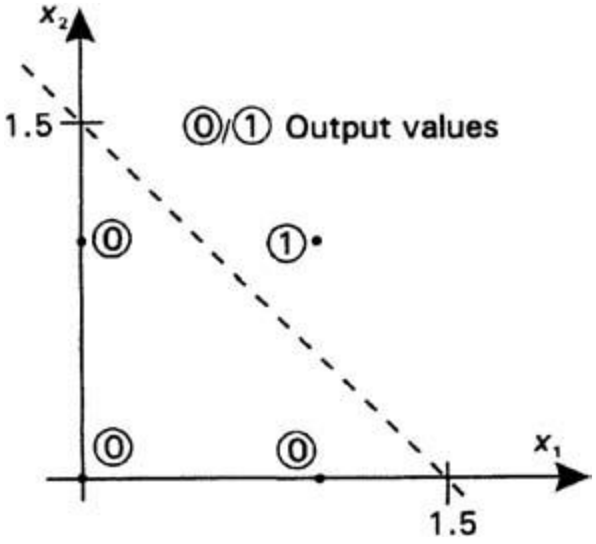
y: output (predicted label)

x: input (feature value)

- If the prediction is correct, (t-y) == 0. Which means Δw_i is zero for any i
  - No changes to any weights, the model works for this instance
- If the feature value x_i is zero, Δw_i is zero
  - No change for this weight, as it was not to blame for the wrong decision

Classification example





Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1		-0.4		0.2							

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1		-0.4		0.2							

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2							

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0				

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25			

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
	0		1								

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2							

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25			

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05							

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05							

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25

The rule: $\Delta w_i = r * (t - y) * x_i$

Learning rate r: 0.25

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
	0		0								

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2							

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2	-0.15	0	0				

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2	-0.15	0	0				

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2	-0.15	0	0	0	0	0	0

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2	-0.15	0	0	0	0	0	0
-0.15	0	-0.15	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25

We can loop over the data multiple times!

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2	-0.15	0	0	0	0	0	0
-0.15	0	-0.15	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.15	0	-0.05	-0.55	0	0	0	0	0	0

Perceptron example

w_0	x_1	w_1	x_2	w_2	a	y	t	r * (t - y)	Δw_0	Δw_1	Δw_2
0.1	0	-0.4	0	0.2	0.1	1	0	-0.25	-0.25	0	0
-0.15	0	-0.4	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.4	0	-0.05	-0.8	0	0	0	0	0	0
-0.4	1	-0.4	1	-0.05	-0.85	0	1	0.25	0.25	0.25	0.25
-0.15	0	-0.15	0	0.2	-0.15	0	0	0	0	0	0
-0.15	0	-0.15	1	0.2	0.05	1	0	-0.25	-0.25	0	-0.25
-0.4	1	-0.15	0	-0.05	-0.55	0	0	0	0	0	0
-0.4	1	-0.15	0	-0.05	-0.55	0	1	0.25	0.25	0.25	0.25

Perceptron example

- Sure enough, this converges after 7 loops over the data!
  - $w_0 = -0.65$, $w_1 = 0.6$ and $w_2 = 0.2$

Perceptron example

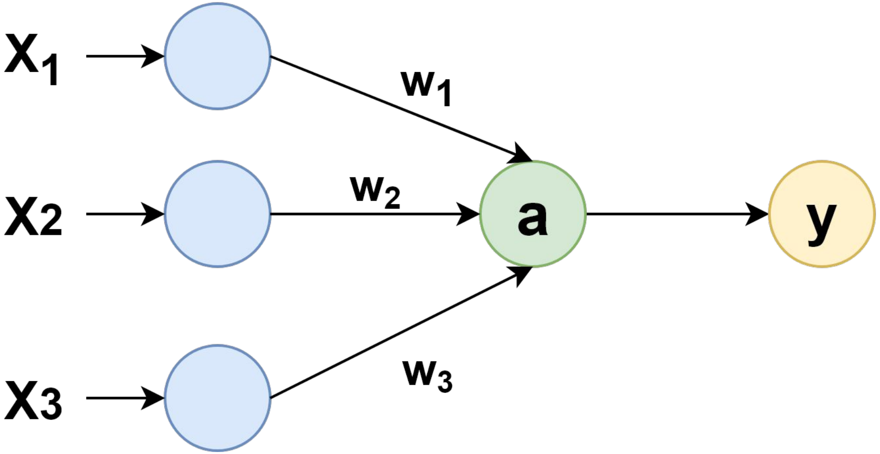
- Sure enough, this converges after 7 loops over the data!
  - $w_0 = -0.65$, $w_1 = 0.6$ and $w_2 = 0.2$

w_0	x_1	w_1	x_2	w_2	a	y	t
-0.65	0	0.6	0	0.2	-0.65	0	0
-0.65	0	0.6	1	0.2	-0.05	0	0
-0.65	1	0.6	0	0.2	-0.45	0	0
-0.65	1	0.6	1	0.2	0.15	1	1

Check for yourself

```
data = [[0,0], [0,1], [1,0], [1,1]]
gold = [0, 0, 0, 1]
w0, w1, w2 = 0.1, -0.4, 0.2
lr = 0.25
for epoch in range(7):
    for idx, (x1, x2) in enumerate(data):
        a = w0 + (x1 * w1) + (x2 * w2)
        y = 1 if a > 0 else 0
        change = lr * (gold[idx] - y)
        w0 += change
        w1 += (change * x1)
        w2 += (change * x2)
```

Perceptron schematically



Perceptron formula

Features:

Represented as a vector: $x = [x_1, x_2, \ldots, x_n]$

Perceptron formula

Features:

Represented as a vector: $x = [x_1, x_2, \ldots, x_n]$

Weights:

Represented as a vector: $w = [w_1, w_2, \ldots, w_n]$

Perceptron formula

Features:

Represented as a vector: $x = [x_1, x_2, \dots, x_n]$

Weights:

Represented as a vector: $w = [w_1, w_2, \dots, w_n]$

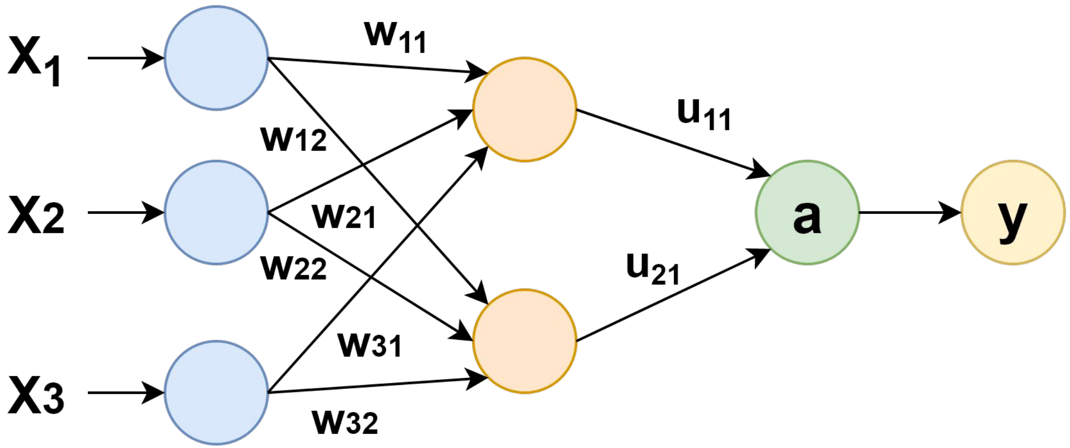
Formula for calculating output a:

$a = w_1x_1 + w_2x_2 + \dots + w_nx_n$

$a = x \cdot v$ = a scalar (single value)

(Actually +b for bias, but we leave this out for simplicity)

Multi-layer perceptron



Formulas

Single-layer perceptron:

$a = x \cdot v$

Formulas

Single-layer perceptron:

$a = x \cdot v$

Multi-layer perceptron:

We have to multiply the same vector with multiple weight vectors (= a matrix):

$h = x \cdot W$ (output is a vector!)

Formulas

Single-layer perceptron:

$a = x \cdot v$

Multi-layer perceptron:

We have to multiply the same vector with multiple weight vectors (= a matrix):

$h = x \cdot W$ (output is a vector!)

Then we are in a similar state as for the single-layer perceptron:

$a = h \cdot u$

Formulas

Single-layer perceptron:

$a = x \cdot v$

Multi-layer perceptron:

We have to multiply the same vector with multiple weight vectors (= a matrix):

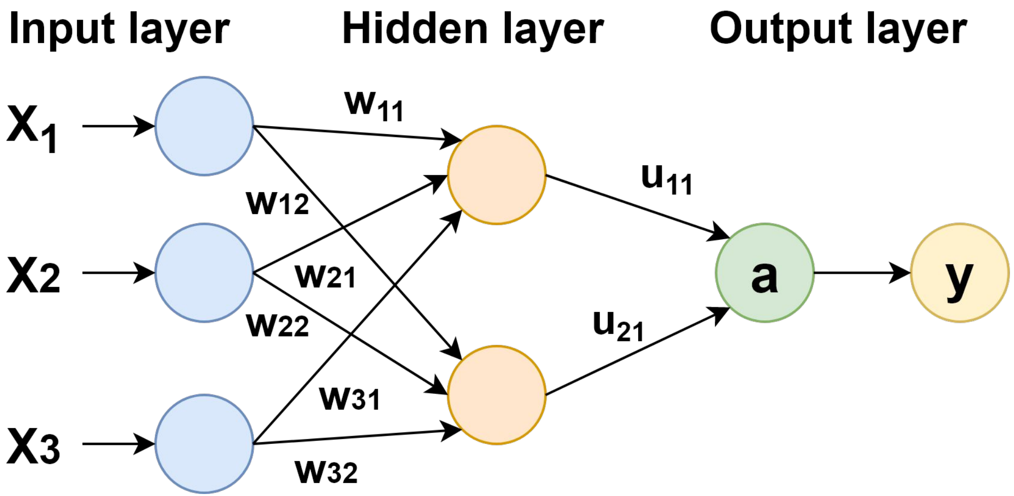
$h = x \cdot W$ (output is a vector!)

Then we are in a similar state as for the single-layer perceptron:

$a = h \cdot u$

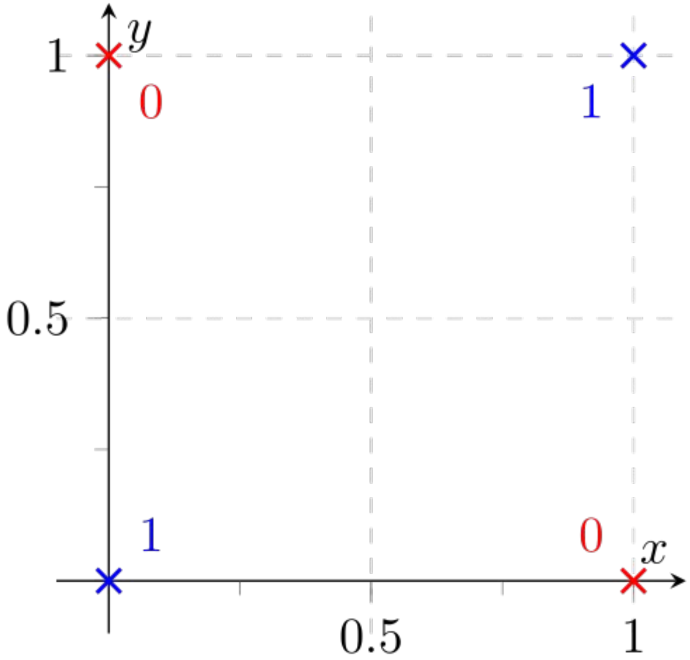
Giving us the formula:

$a = u \cdot (x \cdot W) = u(xW)$, but usually written as: $a = (xW_1)W_2$


Input layer
Hidden layer
Output layer
X1
X2
X3
w11
w12
w21
w22
w31
w32
u11
u21
a
y

But this is still a linear model?

XOR-problem



Non-linearity

Modelling non-linear problems:

- We apply a non-linear function to our hidden vector

Non-linearity

Modelling non-linear problems:

- We apply a non-linear function to our hidden vector
- This function is not learnable (it does not have weights), it's just a calculation we apply to our vector

Non-linearity

Modelling non-linear problems:

- We apply a non-linear function to our hidden vector
- This function is not learnable (it does not have weights), it’s just a calculation we apply to our vector
- This **transforms** the shape of our separator
- Remember the kernel trick for SVMs

Non-linearity

Modelling non-linear problems:

- We apply a non-linear function to our hidden vector
- This function is not learnable (it does not have weights), it’s just a calculation we apply to our vector
- This **transforms** the shape of our separator
- Remember the kernel trick for SVMs

Let’s apply it to our hidden vector and call the function **g:**

$a = g(xW_1)W_2$

Non-linear functions

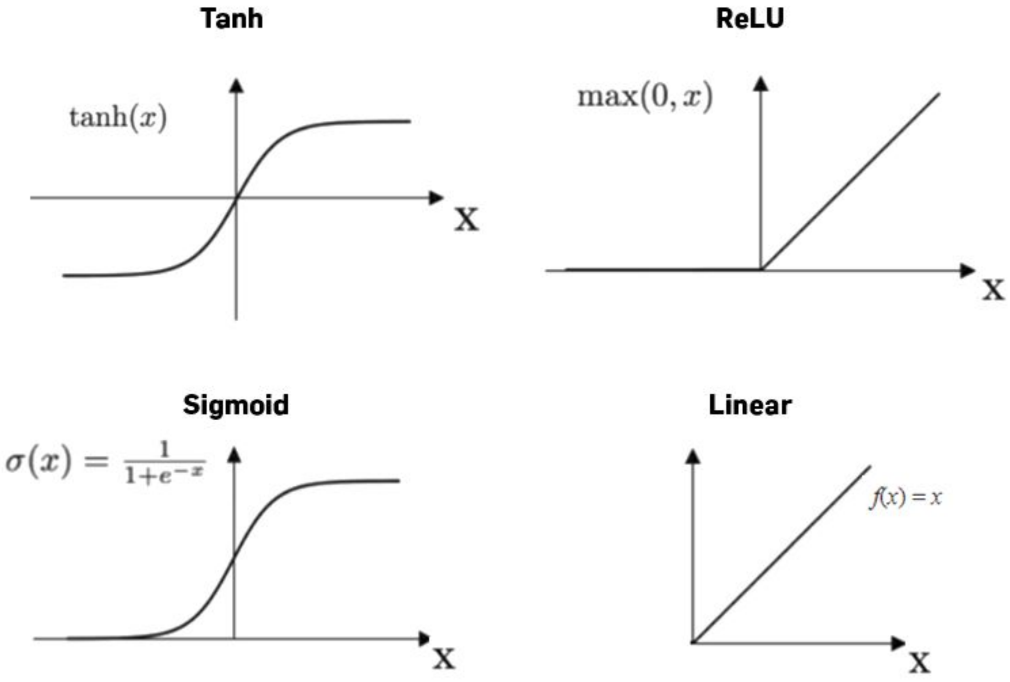
Let's do an example:

$x = [-0.4, 0.8, 1.9, -1.1]$

Non-linear functions

Let's do an example:

x = [-0.4, 0.8, 1.9, -1.1]



Non-linear functions

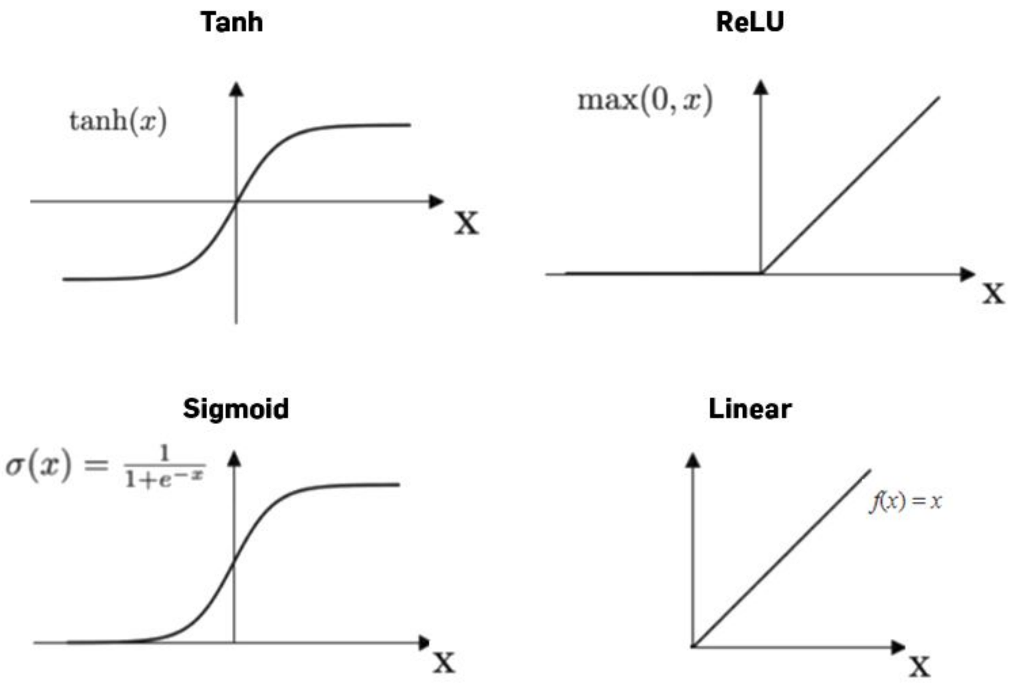
Let's do an example:

x = [-0.4, 0.8, 1.9, -1.1]

tanh(x)   = [-0.38, 0.66, 0.96, -0.80]

σ(x)   = [0.4, 0.69, 0.87, 0.25]

ReLu (x) = [0, 0.8, 1.9, 0]



These functions are called activation functions

Multi-layer perceptrons

1 hidden layer:

$a = g(xW_1)W_2$

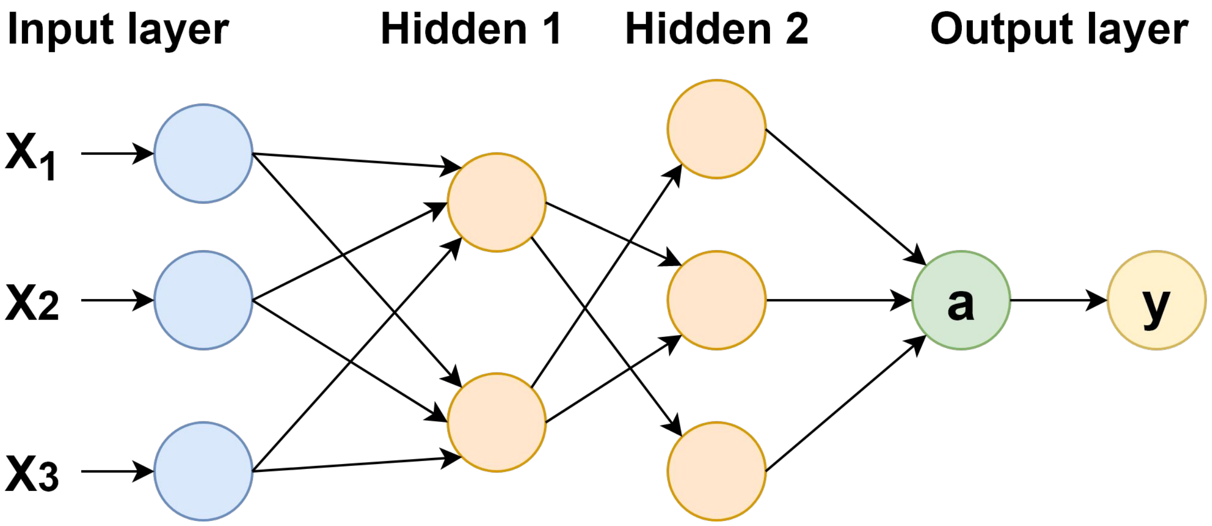
2 hidden layers:

$a = (g_2(g_1(xW_1)W_2)W_3$

2 hidden layers using intermediary variables:

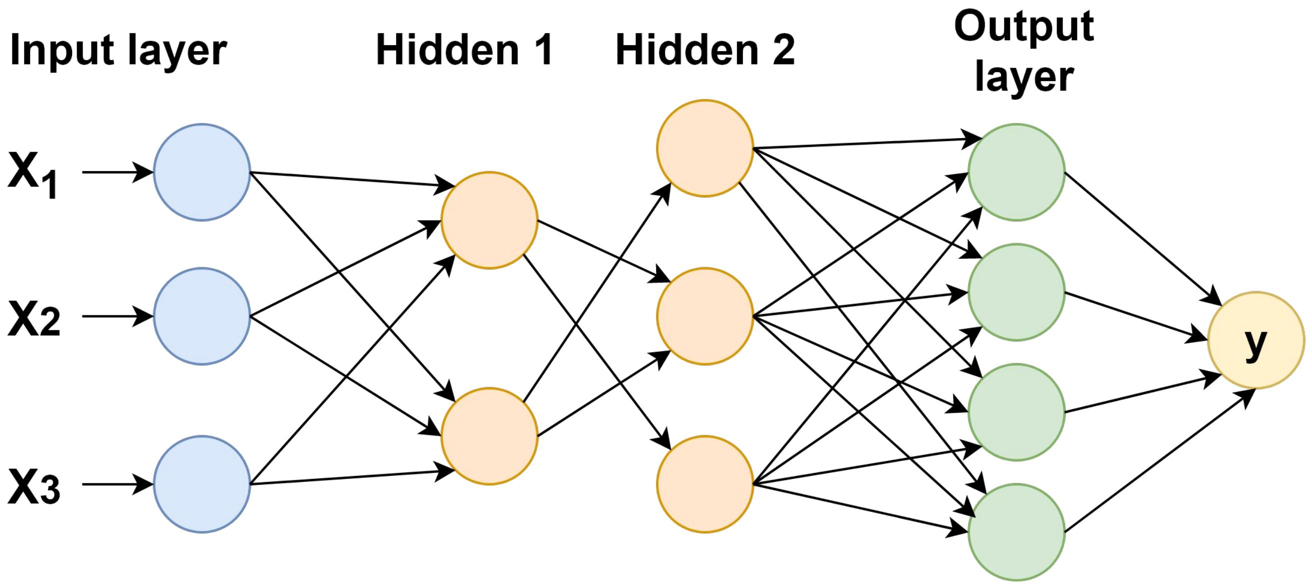
$h_1 = g_1(xW_1)$
$h_2 = g_2(h_1W_2)$
$a = h_2W_3$

2 hidden layers

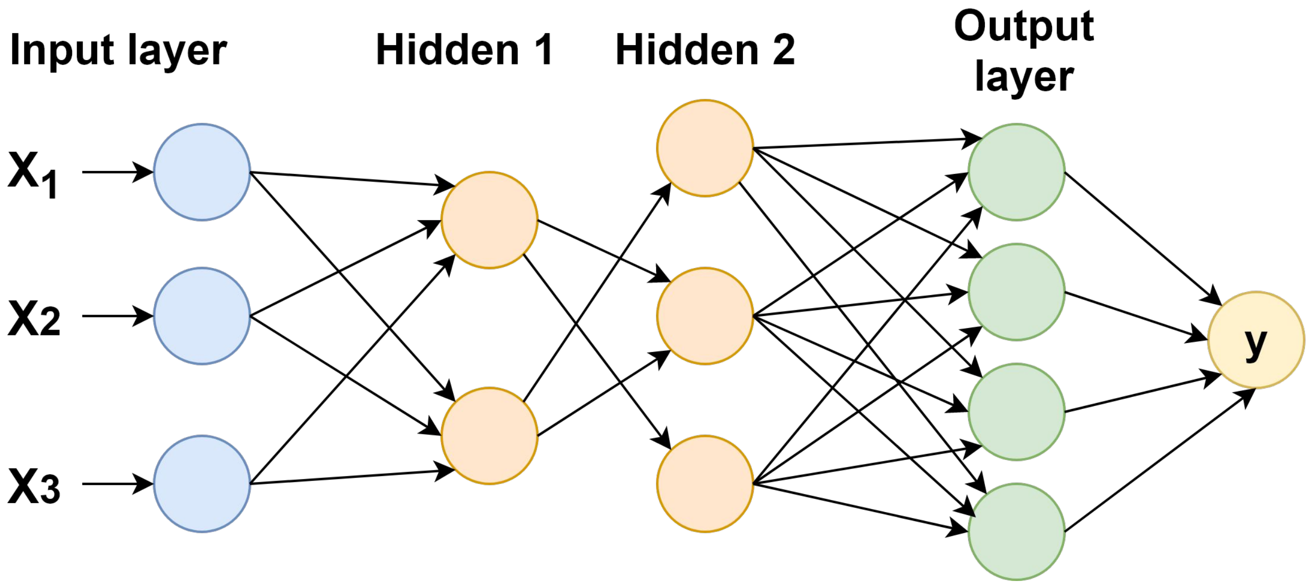


What about multi-class classification?

Multi-class classification

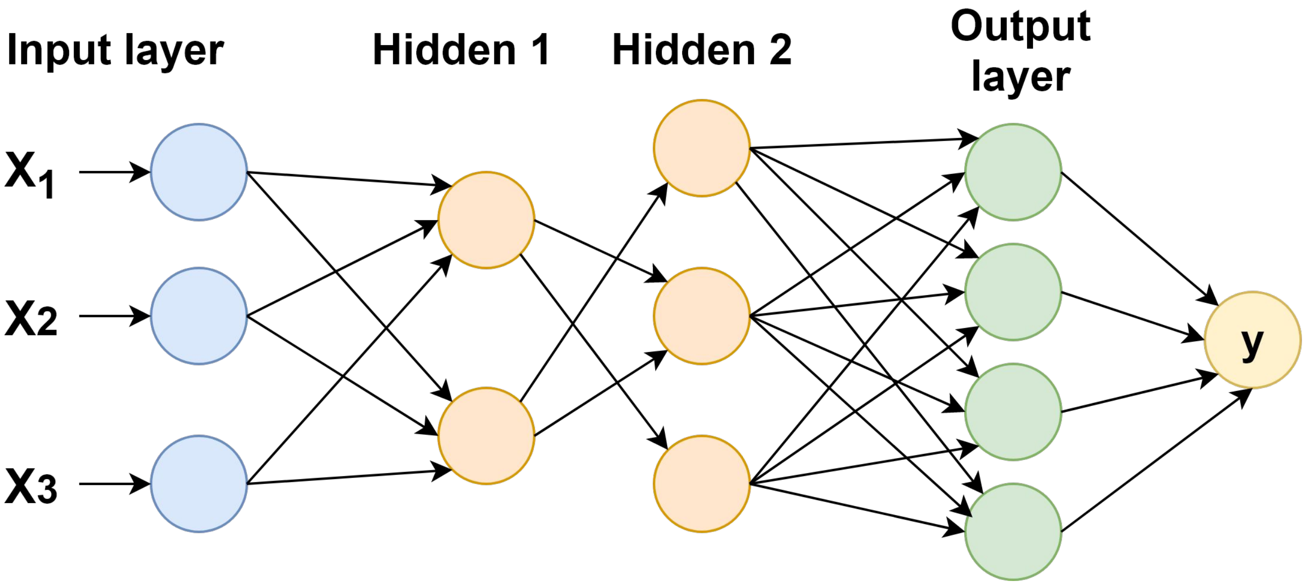


Multi-class classification



Each output class is represented by a node (green). We then simply take the node with the highest value as our final prediction (y).

Multi-class classification



For probabilities per class, put the output layer values through the **softmax function**

This normalizes the output vector to sum to 1 (i.e. perfect for probabilities)

Softmax

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

softmax([0.1, 0.05, 1.2, 0.4]) = [0.16, 0.15, 0.48, 0.21]

softmax([-0.5, -0.1, 0.3, 5]) = [0.004, 0.006, 0.009, 0.98]

softmax([1.5, 1.5, 1.5, 1.5]) = [0.25, 0.25, 0.25, 0.25]

Learning the weights

- Changing the weights is how **learning** takes place

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron
  - Pass instance through the network

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron
    - Pass instance through the network
    - Measure how far off we are (**loss function**)

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron
    - Pass instance through the network
    - Measure how far off we are (**loss function**)
    - Update weights so that we move in the right direction

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron
    - Pass instance through the network
    - Measure how far off we are (**loss function**)
    - Update weights so that we move in the right direction
    - Loop over instances multiple times until we are happy

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron
  - Pass instance through the network
  - Measure how far off we are (**loss function**)
  - Update weights so that we move in the right direction
  - Loop over instances multiple times until we are happy
  - Save weights as final model so we can predict on unseen data

Learning the weights

- Changing the weights is how **learning** takes place
- **Intuition:** the process from the perceptron
  - Pass instance through the network
  - Measure how far off we are (**loss function**)
  - Update weights so that we move in the right direction
  - Loop over instances multiple times until we are happy
  - Save weights as final model so we can predict on unseen data

We will go over these items one by one to see what happens in each step

Pass instance through the network

- This is called the **forward pass**
- Apply all the vector-matrix multiplications given the input
- No changes to the weights are made yet, they are used as is

Pass instance through the network

- This is called the **forward pass**
- Apply all the vector-matrix multiplications given the input
- No changes to the weights are made yet, they are used as is

In other words, we calculate **y**, given the current input vector **x** (say [0.2, 0.3, -0.1]) and the weight matrices $\mathbf{W_1}$, $\mathbf{W_2}$ and $\mathbf{W_3}$

$$y = (g_2(g_1(xW_1)W_2)W_3$$

Remember that **g** is a non-linear function such as tanh or sigmoid

Measure how far off we are

- This is called the **loss function**

Measure how far off we are

- This is called the **loss function**
- Compare what we should have predicted (gold standard label) with what we actually predicted (value in the forward pass)

Measure how far off we are

- This is called the **loss function**
- Compare what we should have predicted (gold standard label) with what we actually predicted (value in the forward pass)
- Possible loss functions:
  - Mean Squared Error: square errors and average
    - Works well for regression
  - Cross Entropy: loss = $-y_{gold} * \log (y_{pred})$
    - Works well with softmax!

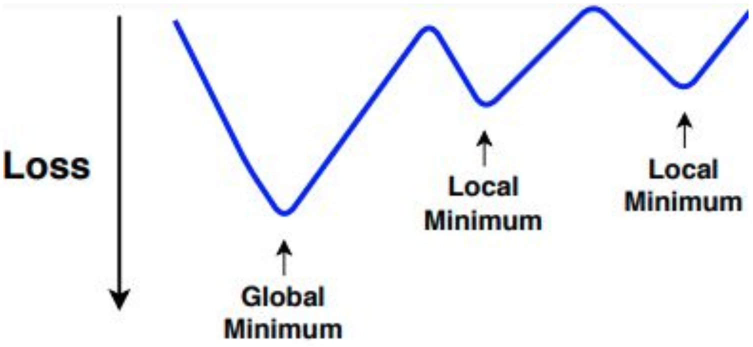
Measure how far off we are

- This is called the **loss function**
- Compare what we should have predicted (gold standard label) with what we actually predicted (value in the forward pass)
- Possible loss functions:
  - Mean Squared Error: square errors and average
    - Works well for regression
  - Cross Entropy: loss = $-y_{gold} * \log (y_{pred})$
    - Works well with softmax!

The **lower** the number, the better!

How do we know how to update the weights?

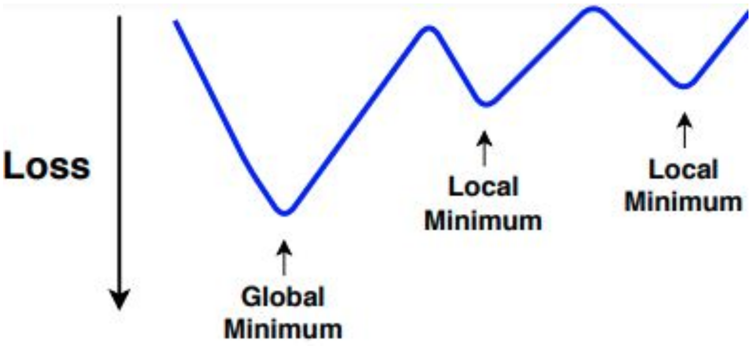
Intuition: since the loss is a function (with parameters/weights), we can minimize it by looking at its **derivative**



How do we know how to update the weights?

Intuition: since the loss is a function (with parameters/weights), we can minimize it by looking at its **derivative**

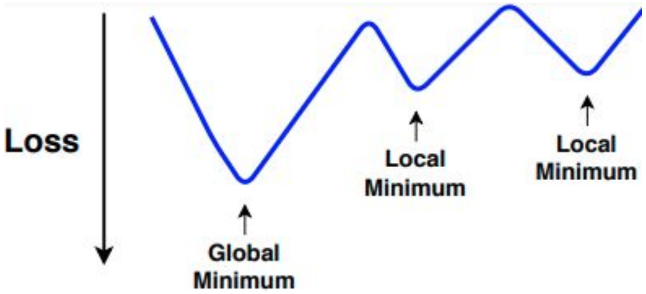
By using the derivative, we know in what direction we have to go to go down - or in other words, how to update the weights: in **opposite direction** of the gradient



How do we know how to update the weights?

Intuition: since the loss is a function (with parameters/weights), we can minimize it by looking at its **derivative**

By using the derivative, we know in what direction we have to go to go down - or in other words, how to update the weights: in **opposite direction** of the gradient

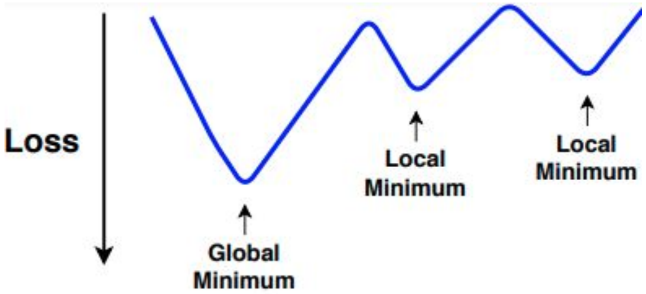


Though this gets complicated: the loss is dependent on the weights, which are dependent on previous weights, etc. How do we calculate the gradient?

How do we know how to update the weights?

Intuition: since the loss is a function (with parameters/weights), we can minimize it by looking at its **derivative**

By using the derivative, we know in what direction we have to go to go down - or in other words, how to update the weights: in **opposite direction** of the gradient

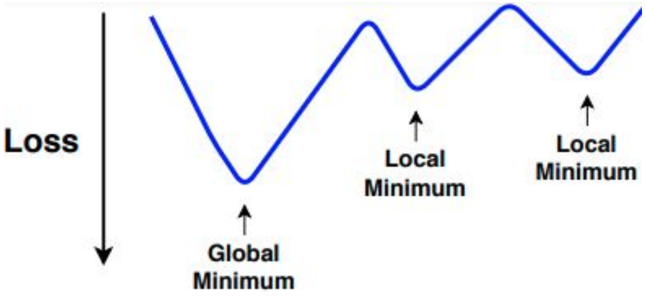


Though this gets complicated: the loss is dependent on the weights, which are dependent on previous weights, etc. How do we calculate the gradient?

Solution: **backpropagation**. Calculate gradients by recursively calculating partial derivatives by using the chain rule.

Gradient Descent

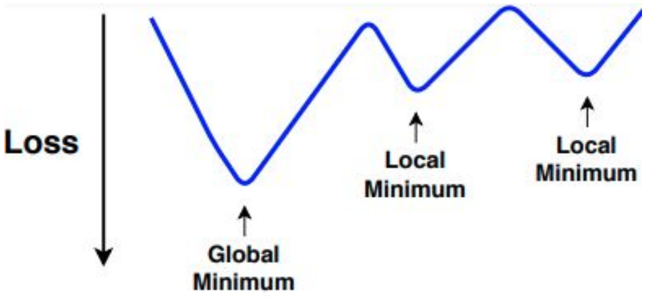
This method of changing the weights is known as **gradient descent**



Gradient Descent

This method of changing the weights is known as **gradient descent**

How much we change the weights at each step is known as the **learning rate.**
We usually update the weights based on a number of instances (**batch size**).

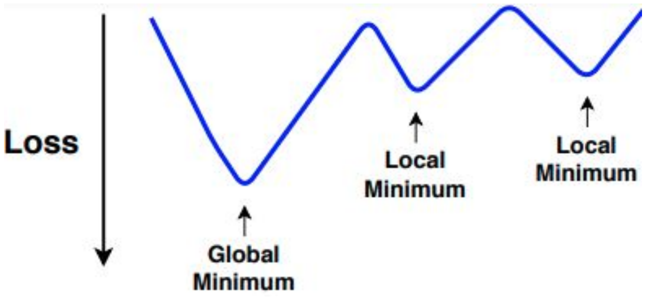


Gradient Descent

This method of changing the weights is known as **gradient descent**

How much we change the weights at each step is known as the **learning rate.**
We usually update the weights based on a number of instances (**batch size**).

These two settings you have to set yourself and are very important!



Optimizers

Types of gradient descent algorithms are called **optimizers**.

Optimizers

Types of gradient descent algorithms are called **optimizers**.

- Stochastic Gradient Descent (**SGD**):
  - Most straightforward: take a step in the right direction based on the current gradient
  - $w = w - lr * \nabla loss(w\ ;\ x_{i\,:\,i+n}\ ;\ y_{i\,:\,i+n})$

Optimizers

Types of gradient descent algorithms are called **optimizers**.

- Stochastic Gradient Descent (**SGD**):
  - Most straightforward: take a step in the right direction based on the current gradient
  - $w = w - lr * \nabla loss(w \,;\, x_{i\,:\,i+n} \,;\, y_{i\,:\,i+n})$
- **Momentum:** previous updates influence current update
- **Adagrad:** smaller updates for parameters associated with frequent features
- **Adadelta:** less aggressive version of AdaGrad
- **Adam:** combination of advantages of Adadelta and Momentum

Usually, you want to try multiple and see how they perform!

Note this is a highly active research area. For more: https://ruder.io/optimizing-gradient-descent/

How often do we perform these updates?

Training

- There is no clear stopping criterion: we can loop over the data and keep updating the weights as long as we want. Note that a single loop over the data is called an **epoch**

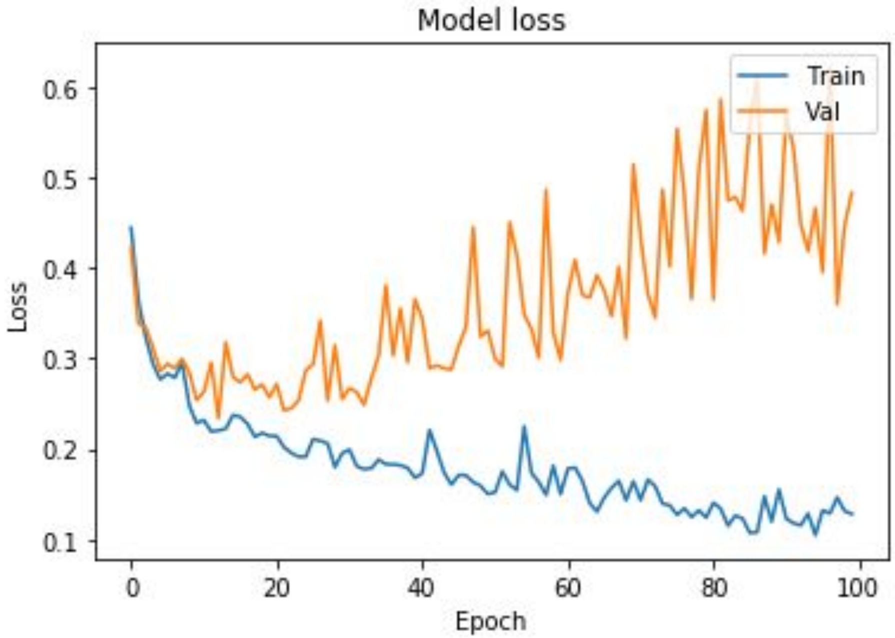
Training

- There is no clear stopping criterion: we can loop over the data and keep updating the weights as long as we want. Note that a single loop over the data is called an **epoch**
- Of course, this could lead to **overfitting**

Training

- There is no clear stopping criterion: we can loop over the data and keep updating the weights as long as we want. Note that a single loop over the data is called an **epoch**
- Of course, this could lead to **overfitting**

Common solution: check loss or performance on development set after each epoch. If we stop improving, we stop training. Note that no weight updates are performed by looking at the development set (only forward pass).


Model loss
Train
Val
0.6
0.5
0.4
0.3
0.2
0.1
Loss
0
20
40
60
80
100
Epoch

Dropout

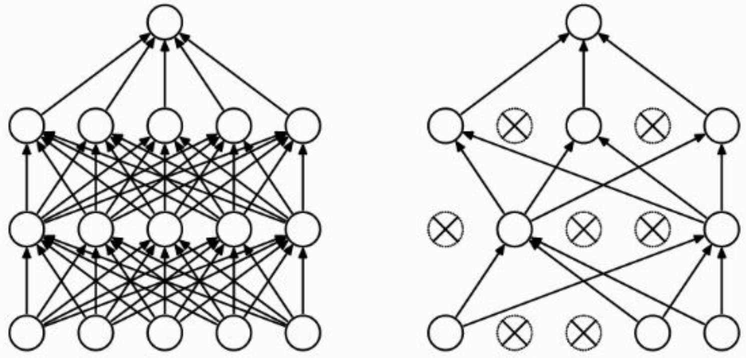
- Prevent overfitting by randomly setting weights to 0 during training
  - Hyperparameter: the percentage of weights that are set to 0

Dropout

- Prevent overfitting by randomly setting weights to 0 during training
  - Hyperparameter: the percentage of weights that are set to 0
- Teaches the model to not be too reliant on single weights

Dropout

- Prevent overfitting by randomly setting weights to 0 during training
  - Hyperparameter: the percentage of weights that are set to 0
- Teaches the model to not be too reliant on single weights
- Add this after a hidden layer + activation



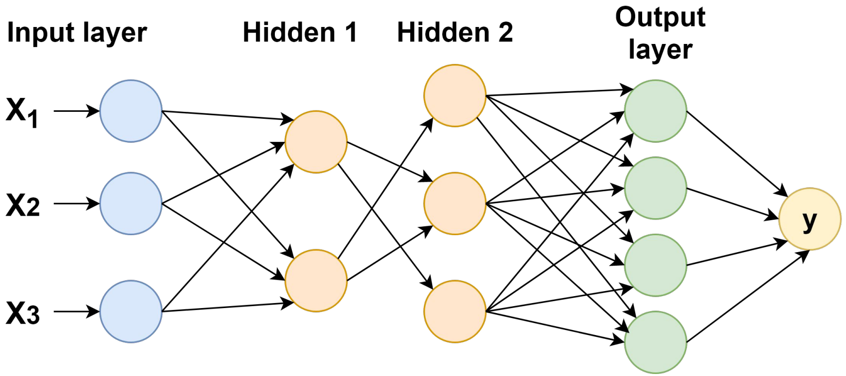
(a) Standard Neural Net

(b) After applying dropout.

Creating a neural network

To create a neural net, you have to specify:

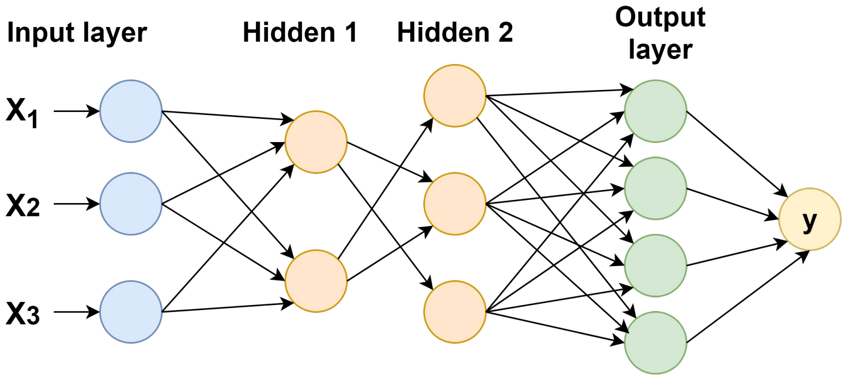
- Input features and output categories



Creating a neural network

To create a neural net, you have to specify:

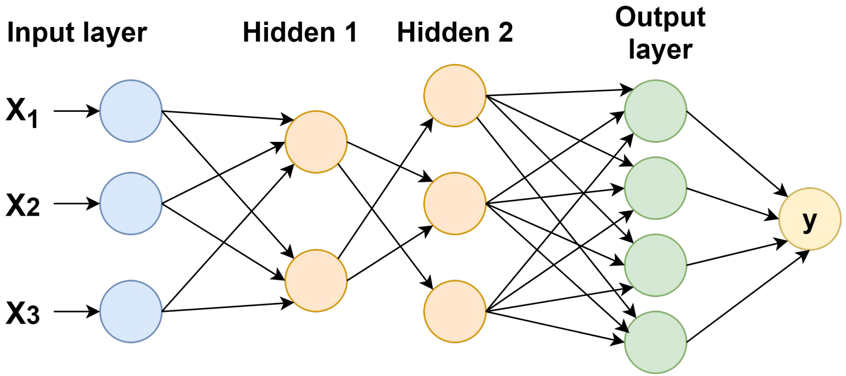
- Input features and output categories
- Number of hidden layers



Creating a neural network

To create a neural net, you have to specify:

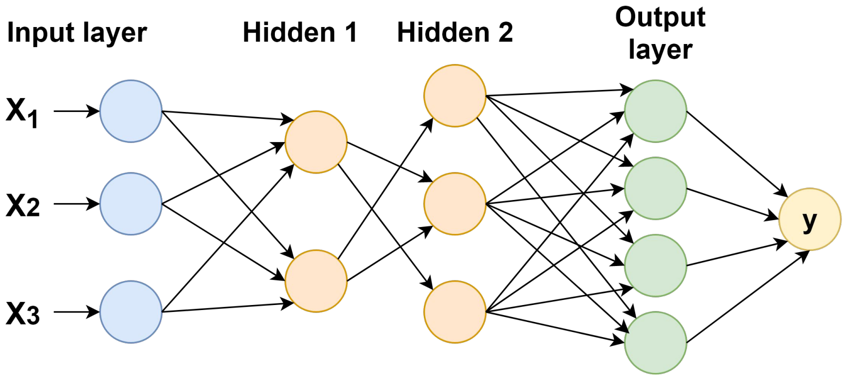
- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)



Creating a neural network

To create a neural net, you have to specify:

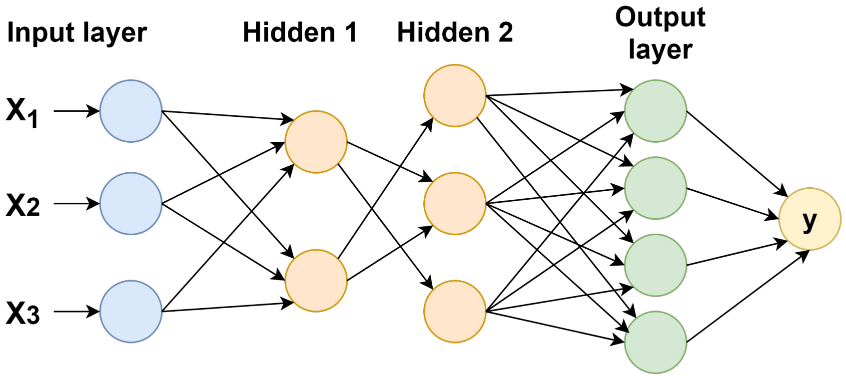
- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)
- Loss function (MSE, Cross entropy)



Creating a neural network

To create a neural net, you have to specify:

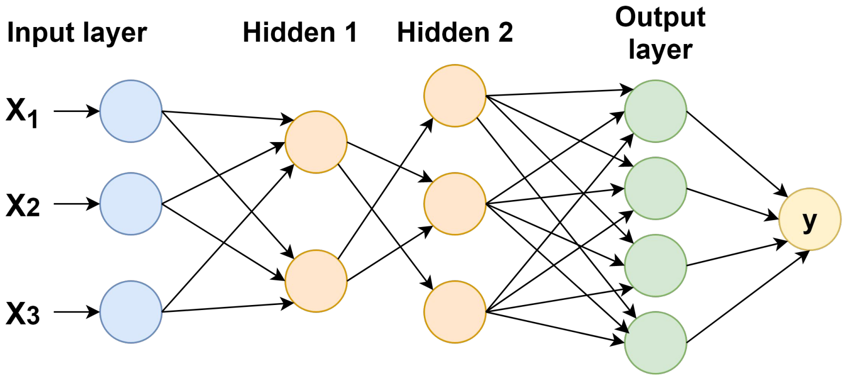
- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)
- Loss function (MSE, Cross entropy)
- Optimizer (SGD, Adadelta, Adam)



Creating a neural network

To create a neural net, you have to specify:

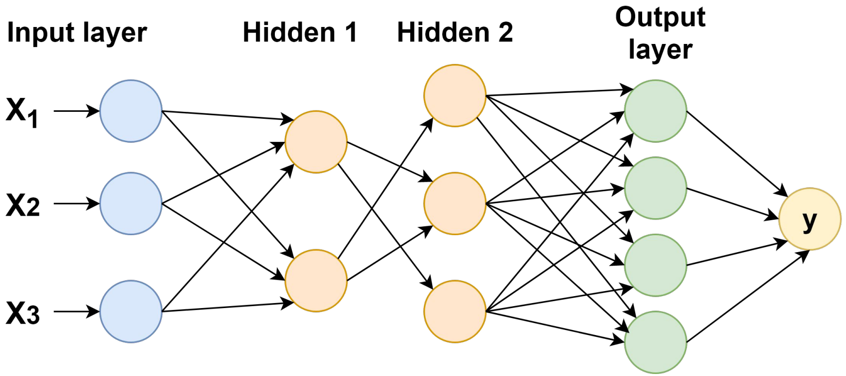
- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)
- Loss function (MSE, Cross entropy)
- Optimizer (SGD, Adadelta, Adam)
- Where and if to add dropout



Creating a neural network

To create a neural net, you have to specify:

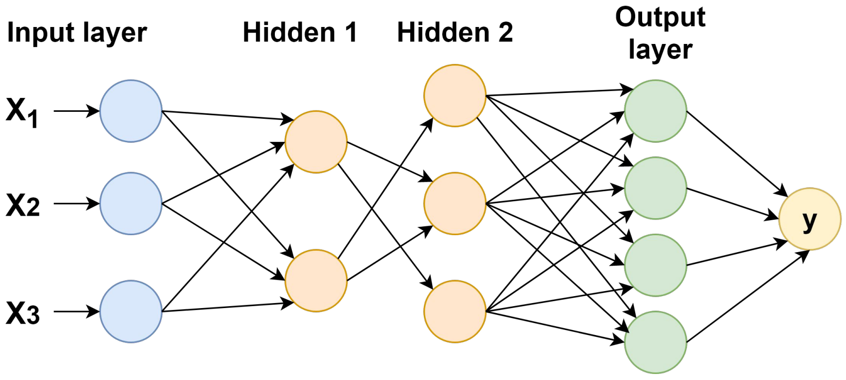
- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)
- Loss function (MSE, Cross entropy)
- Optimizer (SGD, Adadelta, Adam)
- Where and if to add dropout
- When to stop training



Creating a neural network

To create a neural net, you have to specify:

- Input features and output categories
- Number of hidden layers
- Activation function (sigmoid, tanh)
- Loss function (MSE, Cross entropy)
- Optimizer (SGD, Adadelta, Adam)
- Where and if to add dropout
- When to stop training



Of course, you don't always know what works in advance. **Experiment!**

Finding the best settings

Once you have neural net set up, you want to push the performance. You can do this by trying to optimize:

- Number of epochs
- Optimizer
- Activation function
- Learning rate
- Number of nodes in hidden layers
- Batch size
- Dropout percentage

(And of course by changing things from the previous slide)

Training a neural net

- Loop over data set in **batches**
- For each batch, update the weights of the network:
  - Calculate what we would have predicted (**forward pass**)
  - Calculate how far off we are (**loss function**)
  - Change the weights given the optimizer and learning rate (**backpropagation**)
- Measure loss throughout the process, stop training if we stop improving
- Save the weights as our final model
- Predict on new data by only doing a **forward pass** (no backpropagation)
  - Remember that a forward pass is nothing more than vector-matrix multiplications
- Given accuracy, perhaps make some changes and start again!

In the lab, we do a small competition!

Wait, how do we get the features?

Features

- Bag-of-words is too sparse and inefficient for neural networks
  - Though in principle neural networks can work with any input vector
- Preferably, we want to represent our text as a **dense vector**

Features

- Bag-of-words is too sparse and inefficient for neural networks
  - Though in principle neural networks can work with any input vector
- Preferably, we want to represent our text as a **dense vector**

We will now look at the concept of **pretrained word embeddings**, or in other words, have an associated **dense vector per word** in the vocabulary.

Word embeddings intuition

“You shall know a word by the company it keeps”

J.R. Firth, 1957

Words as vectors

- **Intuition:** documents have features, but words can also have features!

Words as vectors

- **Intuition:** documents have features, but words can also have features!
- Say we represent 4 adjectives with 2 dimensions (features)

Word	Intensity	Positivity
great		
terrible		
good		
bad		

Words as vectors

- **Intuition:** documents have features, but words can also have features!
- Say we represent 4 adjectives with 2 dimensions (features)

Word	Intensity	Positivity
great	0.8	0.98
terrible	0.9	0.01
good	0.5	0.96
bad	0.5	0.03

Words as vectors

- **Intuition:** documents have features, but words can also have features!
- Say we represent 4 adjectives with 2 dimensions (features)

Word	Intensity	Positivity
great	0.8	0.98
terrible	0.9	0.01
good	0.5	0.96
bad	0.5	0.03

Words as vectors

- **Intuition:** documents have features, but words can also have features!
- Say we represent 4 adjectives with 2 dimensions (features)

Word	**Intensity**	**Positivity**	**Adjectivity**
great	0.8	0.98	
terrible	0.9	0.01	
good	0.5	0.96	
bad	0.5	0.03	

Words as vectors

- **Intuition:** documents have features, but words can also have features!
- Say we represent 4 adjectives with 2 dimensions (features)

Word	Intensity	Positivity	Adjectivity
great	0.8	0.98	0.97
terrible	0.9	0.01	0.99
good	0.5	0.96	0.98
bad	0.5	0.03	0.96

Words as vectors

- **Intuition:** documents have features, but words can also have features!
- Say we represent 4 adjectives with 2 dimensions (features)

Word	Intensity	Positivity	Adjectivity
great	0.8	0.98	0.97
terrible	0.9	0.01	0.99
good	0.5	0.96	0.98
bad	0.5	0.03	0.96
table	0.2	0.25	0.03

Words as vectors

- More dimensions ⇨ more information

Words as vectors

- More dimensions ⇒ more information
- Not just *intensity* and *positivity*, but also *simplicity*, *rudeness*, *colour-related*, etc
    - Usually, words are represented by a vector size of 50 to 500
    - All words have the same size vector though!

Words as vectors

- More dimensions ⇨ more information
- Not just *intensity* and *positivity*, but also *simplicity*, *rudeness*, *colour-related*, etc
  - Usually, words are represented by a vector size of 50 to 500
  - All words have the same size vector though!

Not just adjectives: we want to represent **all words** in this **vector space**

How do we determine these features?
And how do we set them?

Training word embeddings

- We don’t manually set any features, we let a model figure it out
  - Only have to specify a predefined vector size (say 300)

Training word embeddings

- We don’t manually set any features, we let a model figure it out
  - Only have to specify a predefined vector size (say 300)
- We have a model find the “optimal” semantic representation
  - Afterwards, the dimensions are impossible to interpret

Training word embeddings

- We don’t manually set any features, we let a model figure it out
  - Only have to specify a predefined vector size (say 300)
- We have a model find the “optimal” semantic representation
  - Afterwards, the dimensions are impossible to interpret

Goal:

- Words that are similar in meaning should have similar vectors
- Words that are different in meaning should have different vectors

Training word embeddings

- We don’t manually set any features, we let a model figure it out
  - Only have to specify a predefined vector size (say 300)
- We have a model find the “optimal” semantic representation
  - Afterwards, the dimensions are impossible to interpret

Goal:

- Words that occur in similar contexts should have similar vectors
- Words that occur in different context should have different vectors

Don’t count, predict!

Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)

Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)
- Advantage: basically infinite training data available

Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)
- Advantage: basically infinite training data available
- Say we use a simple **feed-forward neural network**
    - Assign each word in the vocabulary a random vector (of size 300): matrix **E**

Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)
- Advantage: basically infinite training data available
- Say we use a simple **feed-forward neural network**
  - Assign each word in the vocabulary a random vector (of size 300): matrix **E**
  - Input: the vector for the previous word w_{i-1} taken from **E** $\Rightarrow$ $\mathbf{x} = \mathbf{E}_{w_{i-1}}$

Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)
- Advantage: basically infinite training data available
- Say we use a simple **feed-forward neural network**
  - Assign each word in the vocabulary a random vector (of size 300): matrix $\mathbf{E}$
  - Input: the vector for the previous word w_{i-1} taken from $\mathbf{E} \Rightarrow \mathbf{x} = \mathbf{E}_{w_{i-1}}$
  - Feed $\mathbf{x}$ into 1 or more hidden layer with weights $\mathbf{W}_1$, $\mathbf{W}_2$, etc (of say size 300 again)

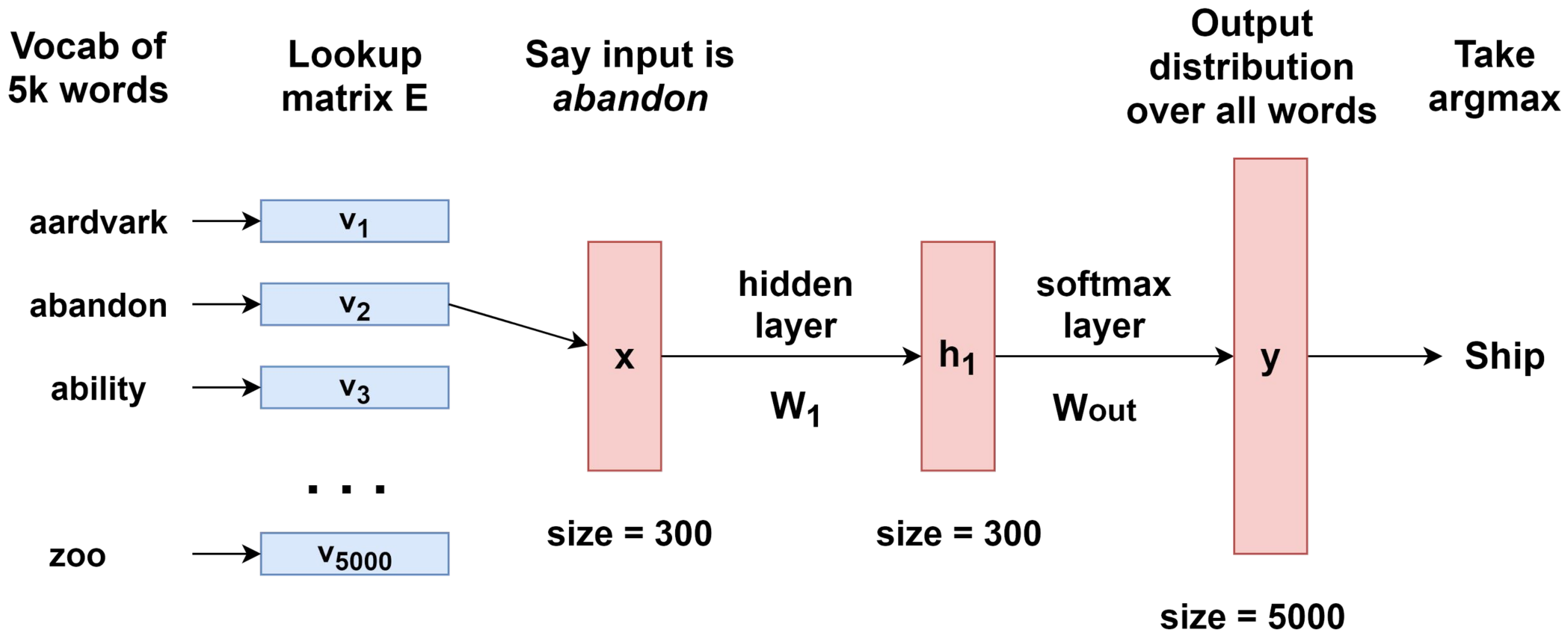
Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)
- Advantage: basically infinite training data available
- Say we use a simple **feed-forward neural network**
  - Assign each word in the vocabulary a random vector (of size 300): matrix **E**
  - Input: the vector for the previous word w_{i-1} taken from **E** ⇨ $\mathbf{x} = \mathbf{E}_{w_{i-1}}$
  - Feed **x** into 1 or more hidden layer with weights $\mathbf{W_1}$, $\mathbf{W_2}$, etc (of say size 300 again)
  - Feed this to output layer of the size of the vocab with **softmax** activation and weights $\mathbf{W_{out}}$
    - This given us an output probability distribution over all words

Neural Language Modelling

- Train a neural network to predict the next word given the previous word(s)
- Advantage: basically infinite training data available
- Say we use a simple **feed-forward neural network**
  - Assign each word in the vocabulary a random vector (of size 300): matrix **E**
  - Input: the vector for the previous word w_{i-1} taken from **E** $\Rightarrow$ $\mathbf{x} = \mathbf{E}_{w_{i-1}}$
  - Feed **x** into 1 or more hidden layer with weights $\mathbf{W}_1$, $\mathbf{W}_2$, etc (of say size 300 again)
  - Feed this to output layer of the size of the vocab with **softmax** activation and weights $\mathbf{W}_{out}$
    - This given us an output probability distribution over all words
  - Train the system: backpropagate errors through $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{W}_{out}$ and also **E**!

Neural Language Model



Neural Language Modelling

- Each word has an associated vector in $\mathbf{E}$ and $\mathbf{W}_{\mathbf{out}}$

Neural Language Modelling

- Each word has an associated vector in $\mathbf{E}$ and $\mathbf{W}_{\mathbf{out}}$
- These vectors were changed to be useful for language modelling

Neural Language Modelling

- Each word has an associated vector in **E** and $\mathbf{W}_{out}$
- These vectors were changed to be useful for language modelling
  - In other words, predicting words from a given context
  - Similar contexts should result in similar output words
  - It should follow that similar words get similar vectors!
  - We can extract **E** and $\mathbf{W}_{out}$ and use them for something else

Neural Language Modelling

- Each word has an associated vector in **E** and $\mathbf{W}_{out}$
- These vectors were changed to be useful for language modelling
  - In other words, predicting words from a given context
  - Similar contexts should result in similar output words
  - It should follow that similar words get similar vectors!
  - We can extract **E** and $\mathbf{W}_{out}$ and use them for something else

Intuition: we trained word vectors on a task were we have lots of data. Hopefully they learned some general things about language. We can then simply use them on a different task with not as much data!

Training procedures

- The actual best systems use a more complicated procedure
  - Though the intuition remains the same
- Some of the most popular procedures:
  - **CBOW:** use surrounding context to predict a word (**word2vec, FastText**)
  - **Skipgram:** use input word to predict context (**word2vec**)
  - Predicting co-occurrence counts (**GloVe**)
  - Exploiting syntactic structure (dependency triples)
  - And many more!

What they have in common: a release of words with associated vectors

How to use them?

We can assign a pretrained vector to each word in our input, but how do we then deal with a set of input vectors of variable length?

How to use them?

We can assign a pretrained vector to each word in our input, but how do we then deal with a set of input vectors of variable length?

- Average them, or apply max pooling, or some other method
- Next week we will look at models that can deal with them directly!

This week: input is a single word, so just take the vector as input features

How to evaluate word embeddings?

Extrinsic vs Intrinsic Evaluation

Extrinsic vs Intrinsic Evaluation

Extrinsic:

- Use the embeddings in a downstream application
- Higher performance $\Rightarrow$ high quality embeddings
- **Pro:** useful, realistic, meaningful
- **Con:** selection of tasks, datasets, systems

Extrinsic vs Intrinsic Evaluation

Extrinsic:

- Use the embeddings in a downstream application
- Higher performance ⇨ high quality embeddings
- **Pro:** useful, realistic, meaningful
- **Con:** selection of tasks, datasets, systems

Intrinsic:

- Use embeddings to measure similarity and analogies
- **Pro:** cheap, fewer confounding factors
- **Con:** too specific, human judgments are imperfect, different datasets

Let’s look at the power of word embeddings

Similarities

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Calculated by using the **cosine similarity** between two vectors.

Similarity

Similarity does not mean the same thing across different settings!

- Example: “good” vs “bad”
  - Sentiment analysis: good and bad are very different
  - Parsing: good and bad are treated the same
- Ambiguity: word meaning differs depending on context
  - He won the first **set**
  - I accidentally **set** myself on fire

Word embeddings only look at co-occurrence in contexts.

Common problem: antonyms get similar vectors

Assignment 2

- Individual. Have to answer questions only (not like research report!)
  - No need to submit code this week
- In the lab: **live competition** to train the best neural net using embeddings!
  - Optimizer hyperparameters, architecture, training time, etc
  - Best performing students get a bonus
  - See assignment for details

Check **Brightspace** for a list of resources/papers that might help

Deadline: Monday September 25th 10:59 AM

Next week: deep learning