Final Recap

Introduction to **Machine Learning** with **Python**

Faculty of Mathematics and Computer Science, University of Bucharest
and
Sparktech Software

Academic Year 2018/2019, 1st Semester

Introduction

Objectives of this course

- Provide you with an **intuitive understanding** of fundamental Machine Learning notions and algorithms.
  - Sometimes the *idea* behind an algorithm is more important than the algorithm itself.
- At the same time, provide you with a clear **mathematical foundation** for them.
  - Understanding the *inner workings* of an algorithm allows you to truly take advantage of what it can do.
- Allow you to **experiment hands-on** with the notions discussed in the lecture.
  - Using *Python* with *NumPy*, *matplotlib*, *scikit-learn* and *TensorFlow*.
  - Make *connections* between theoretical and practical aspects.

What is Machine Learning?

- Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed.

Arthur Samuel, 1959

- A computer program is said to learn from experience *E* with respect to some task *T* and some performance measure *P*, if its performance on *T*, as measured by *P*, improves with experience *E*.

Tom Mitchell, 1997

When to use Machine Learning?

- Problems for which traditional solutions require **lots of hand-tuning** or long **lists or rules**, which are *hard to maintain*:
  - E.g. Spam Detection, Machine Translation

- **“Unprogrammable”** tasks: Complex problems for which using a traditional programming approach is *virtually impossible*:
  - E.g. Object Detection, Speech Recognition

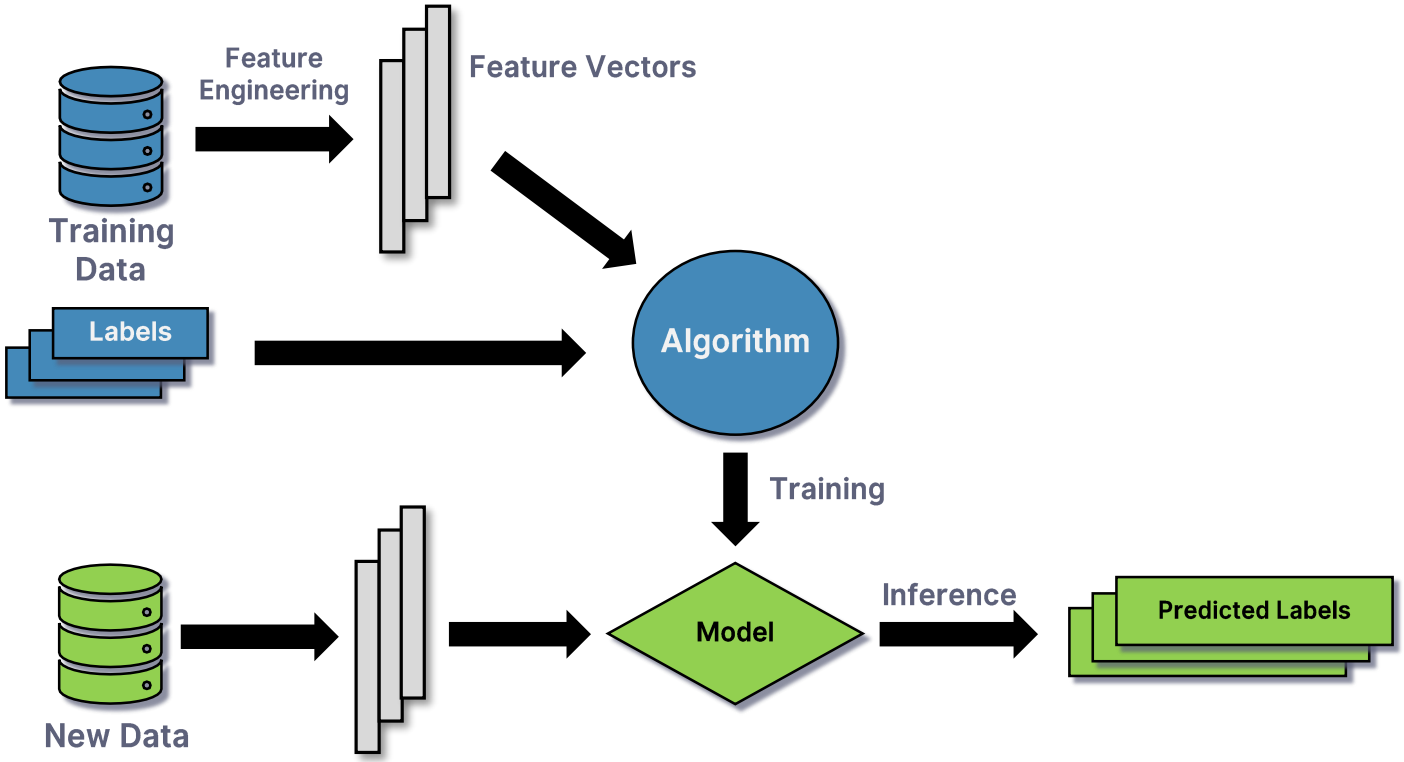
- Revealing insights and **unsuspected correlations** from large amounts of data.

Machine Learning Terminology

Machine Learning Terminology

- **Label** (or **target**) → What we are trying to *predict*.
- **Feature** (or **attribute**) → Measurable characteristic of a **sample** (data point).
  - All features form a **feature vector**.
- **Model** (or **hypothesis**) → Relationship between features and labels.
- **Training** (or **fitting**) → Establishing the relationship based on a set of data points.
- **Inference** → Making predictions on previously unseen points.
- **Algorithm** → Defines a concrete way of doing training.
  - Has constrains on the set of **allowed hypotheses**, some by design, some by the use of **hyperparameters**

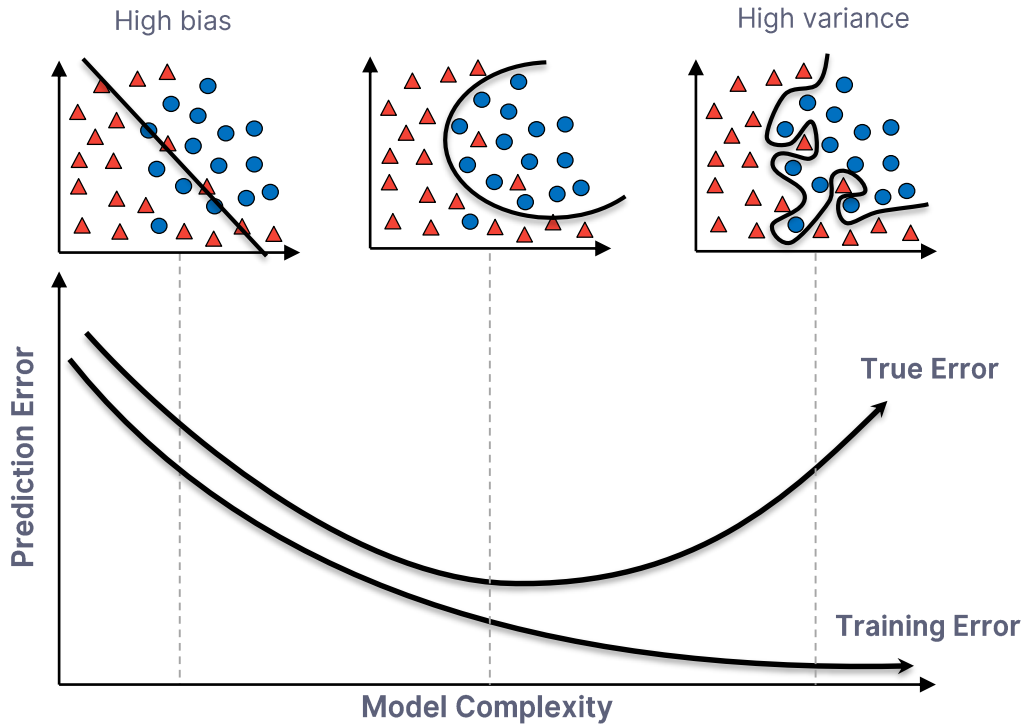
Typical Machine Learning flow



Types of Learning

- **Supervised Learning** → There is a label
  - Label is continuous → **Regression**
  - Label is discrete → **Classification**
- **Unsupervised Learning** → Discovering structure in the data.
  - Grouping similar items → **Clustering**
  - Reducing number of features → **Dimensionality Reduction**
  - Frequent patterns → **Association Rule Mining**
- **Reinforcement Learning** → There is no label, only rewards (or penalties) for taking actions
- *Semi-supervised Learning* → Some labeled, lots of unlabeled data
- *Transfer Learning* → Use model trained for one task to speed up learning for another task

Underfitting vs. Overfitting



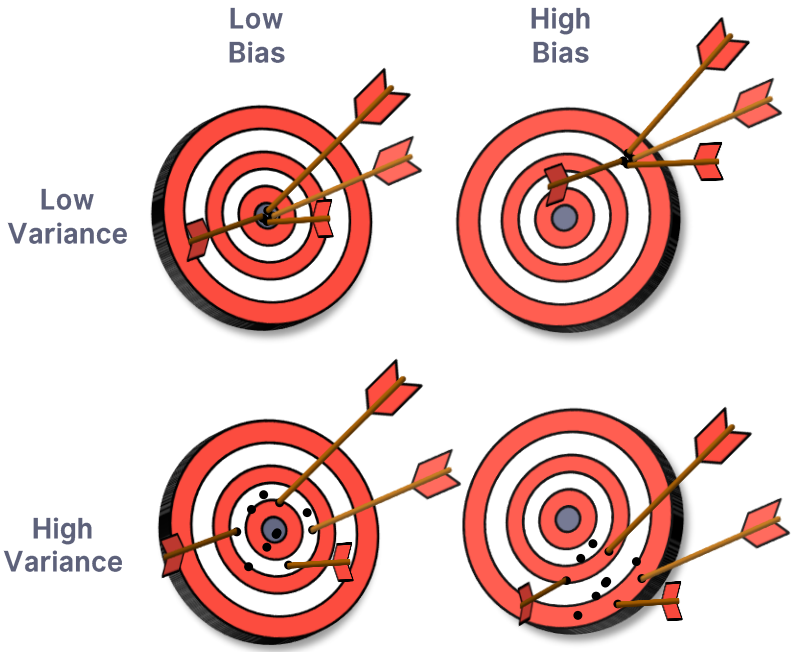
"Everything should be made simple as possible, but not simpler"
Albert Einstein

How do we avoid overfitting?

- By not computing empirical error on the same data which was used for training:
  - Hold out data for testing.
- By not choosing model hyperparameters to only fit the test set:
  - Hold out data for validation.
  - Use *cross-validation*
- By preferring simple models over complex models:
  - Occam's Razor principle.
  - Penalizing model complexity is called **regularization.**
- By making sure data points are i.i.d. *(independent and identically distributed*):
  - i.i.d. means there is no bias when selecting the training set
    → every point is selected independently and from the same distribution
  - This assumption is very often violated in practice!
- By getting more data. ☺

Bias vs. Variance

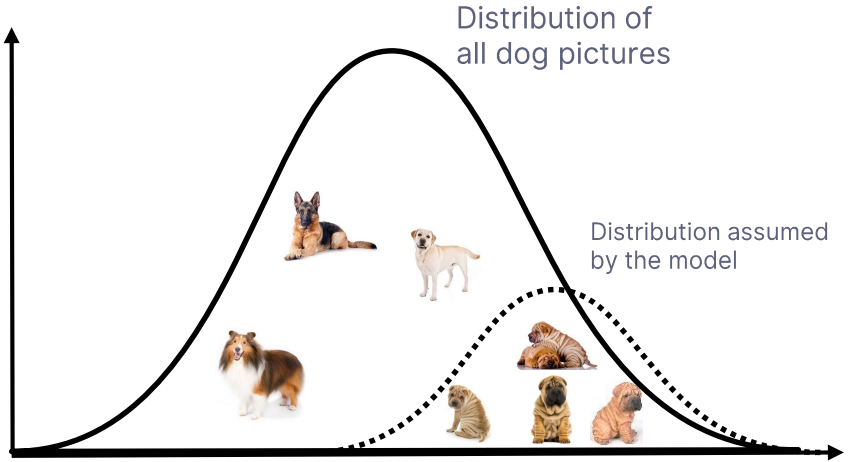
- There are two sources of error in ML models.
- **Bias** (or **systematic error**) comes from the inability of the algorithm to model the true relationship between features and label (*underfitting*).
  - Bias error cannot be corrected by adding more training samples, but by increasing the complexity of the model.
- **Variance** (or **random error**) comes from sensitivity to small fluctuations in the training data, causing the algorithm to model random noise. (*overfitting*)
  - Variance error can be corrected by adding more training samples or by decreasing the complexity of the model.
- There is usually a *tradeoff* between bias and variance.



I.I.D. Assumption

- Most algorithms assume data points are **independent and identically distributed**.
- Let's say that we want to train a model to recognize pictures of dogs:
  - We give it this training set: ( ,  ,  , )

- The examples are not i.i.d.
  - It likely that the model will make mistakes on pictures from different parts of the distribution



Learning a function

- We assume we have:
  - An instance space X, with a fixed (but unknown) distribution D_{X}
  - A target space Y and a function $f: X \to Y$
- We are given:
  - A set of labeled examples $\mathrm{E} \subseteq X \times Y = \{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \ldots, (\vec{x}^{(\mathrm{m})}, y^{(m)})\}$
    such that: $\forall e = (\vec{x}, y) \in E \Rightarrow f(\vec{x}) = y$
    $\vec{x} \sim D_x$ ($\vec{x}$ is drawn i.i.d. from D_{X})
  - A set of allowed hypothesis $\mathcal{H}$
- We need to find:
  - A hypothesis $h \in \mathcal{H}$ s.t. $error_{D_X}(h) \stackrel{\text{def}}{=} \mathbb{E}_{\mathrm{D_X}}[\mathcal{L}(f(\vec{x}), h(\vec{x}))]$ is minimal.
  - D_x is unknown, so we compute $error_S(h) \stackrel{\text{def}}{=} \frac{1}{|S|}\sum_{\vec{x} \in S} \mathcal{L}(f(\vec{x}), h(\vec{x}))$
    - where $S \subset X$ is a finite set (also i.i.d. from D_{X})

Sometimes there is no function f, only a distribution $\mathrm{D_{XY}}$ with

$(\vec{x}, \mathrm{y}) \sim \mathrm{D_{XY}}$

($\vec{x}$ has a probability of having label y)

$\mathcal{L}$ is the loss on a single example $\vec{x}$

$\mathrm{error}_{D_X}(h)$ is the expected error over D_X

(i.e. the **true error** of h)

$\mathrm{error}_S(h)$ is the average error on set S

Learning a function

- The **loss function** should reflect the nature of the problem:
  - Classification:
    - Y is finite (usually small, sometimes binary)
    - $\mathcal{L}\big(f(\vec{x}), h(\vec{x})\big) = \mathcal{L}(y, \hat{y}) = \begin{cases} 1 & \text{if } y \neq \hat{y} \\ 0 & \text{otherwise} \end{cases}$ 0-1 loss
    - $\text{error}_{D_x}(h) = P_{\vec{x} \sim D_X}\big(f(\vec{x}) \neq h(\vec{x})\big)$
  - Regression:
    - Y is continuous
    - $\mathcal{L}\big(f(\vec{x}), h(\vec{x})\big) = \mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$ Squared loss
  - These are very common loss functions, but many others are used as well.

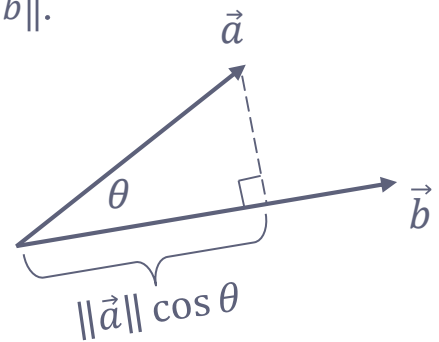
$\hat{y}$ ("y hat") is the notation we'll use for predicted label

Refresher – Dot Product

- An operation which takes two vectors and returns a **scalar** value.
- Also called the *scalar product* or *inner product*.

$$\vec{a} \cdot \vec{b} = \langle \vec{a}, \vec{b} \rangle = \|\vec{a}\| \|\vec{b}\| \cos\theta = \sum_i a_i b_i$$

- It can be interpreted as a **similarity measure** between vectors.
- If we write it as $(\|\vec{a}\| \cos\theta)\|\vec{b}\|$, it can be viewed as the **length of the projection** of $\vec{a}$ on $\vec{b}$, measured in units of length $\|\vec{b}\|$.



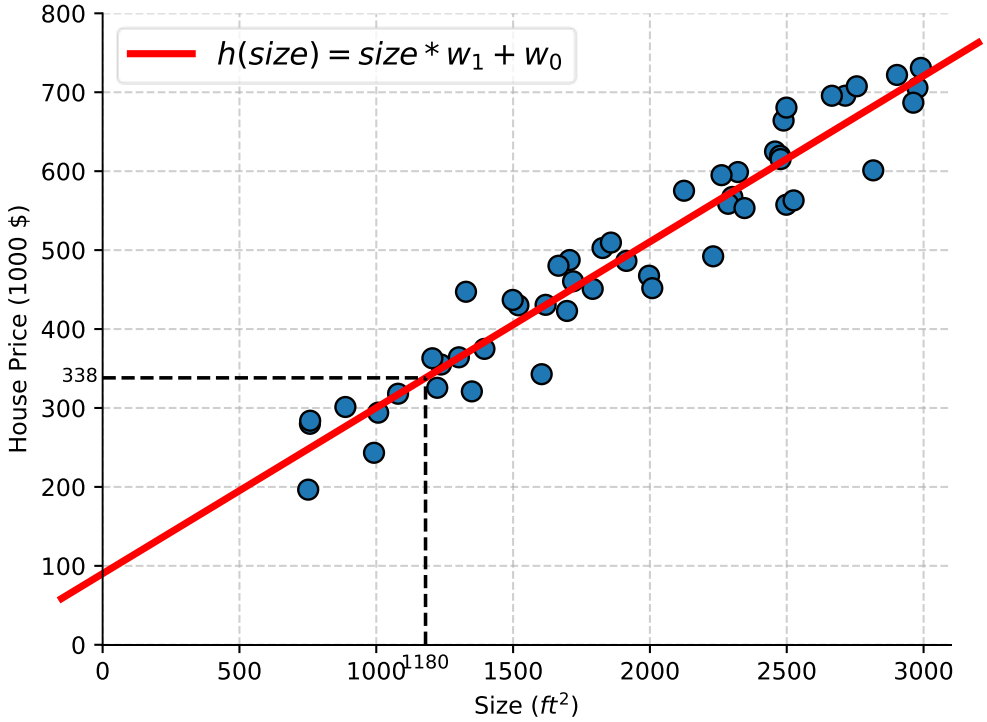
Linear Regression

Simple Linear Regression

Slope is positive →
Positive correlation between price and size (i.e. price tends to grow with size).

We can also determine how much of the **variance** in price is **explained** by size.

We can use the model for forecasting →
- **The model *predicts* that a $1220\ ft^2$ house will cost $347K.**



The relation ship is modeled with a **linear function**.

There are many linear functions to choose from, but we find the *line* which *best fits* the data.
- We need a way of choosing the *best* w_0 and w_1

Least Squares Method

- Given $E = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$, with $x^{(i)}, y^{(i)} \in \mathbb{R}$

$$\mathcal{L}_E = \sum_i \left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \sum_i \left(y^{(i)} - w_1 x^{(i)} - w_0\right)^2$$

Squared loss

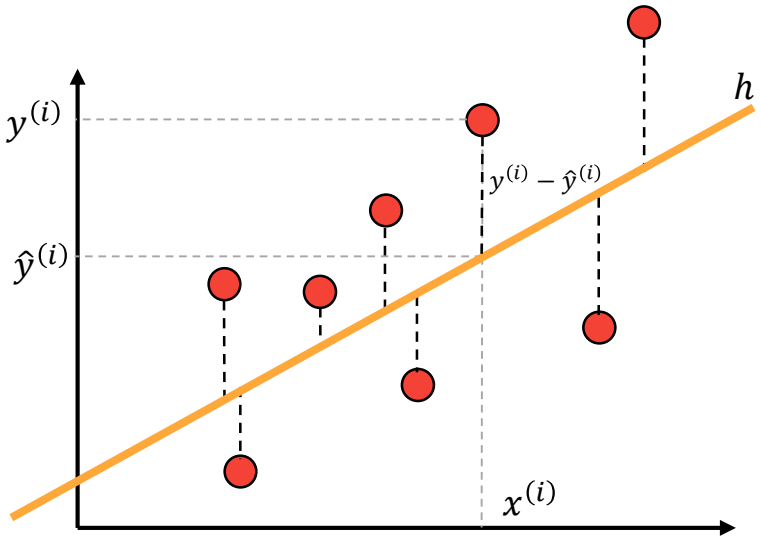
$$\mathcal{L}_E \overset{\text{not}}{=} \sum_{(x,y)\in E} \mathcal{L}(y, h(x))$$

- Minimize $\mathcal{L}_E$ w.r.t. w_0, w_1:

$$\frac{\partial \mathcal{L}_E}{\partial w_0} = 0, \qquad \frac{\partial \mathcal{L}_E}{\partial w_1} = 0$$

$$\Longrightarrow \quad w_1 = \frac{\sum_i \left[\left(x^{(i)} - \bar{x}\right)\left(y^{(i)} - \bar{y}\right)\right]}{\sum_i \left(x^{(i)} - \bar{x}\right)^2}$$

$$w_0 = \frac{1}{m}\left(\sum_i y^{(i)} - w_1 \sum_i x^{(i)}\right)$$



Multiple Linear Regression

- $\vec{x} \in \mathbb{R}^n = [x_1 \quad x_2 \quad \dots \quad x_n]$,
- $\vec{w} \in \mathbb{R}^n = [w_1 \quad w_2 \quad \dots \quad w_n], w_0 \in \mathbb{R}$

$$\hat{y} = \mathrm{w}_0 + w_1 x_1 + w_2 \mathrm{x}_2 + \cdots + \mathrm{w}_\mathrm{n} \mathrm{x}_\mathrm{n} = w_0 + \langle \vec{w}, \vec{x} \rangle$$

- Common mathematical trick is to get rid of the intercept by considering:

$\vec{x} \in \mathbb{R}^{n+1} = [1 \quad x_1 \quad x_2 \quad \dots \quad x_n]$
$\vec{w} \in \mathbb{R}^{n+1} = [w_0 \quad w_1 \quad w_2 \quad \dots \quad w_n]$

$$\Rightarrow$$

$$\hat{y} = \langle \vec{w}, \vec{x} \rangle$$

Multiple Linear Regression

- Given $E = \{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \dots, (\vec{x}^{(m)}, y^{(m)})\}, \vec{x}^{(i)} \in \mathbb{R}^n, \mathrm{y}^{(i)} \in \mathbb{R}$

$$\hat{y}^{(\mathrm{i})} = \langle \vec{w}, \vec{x}^{(i)} \rangle = \mathrm{w}_0 + w_1 x_1^{(\mathrm{i})} + w_2 x_2^{(\mathrm{i})} + \cdots + \mathrm{w}_\mathrm{n} x_\mathrm{n}^{(\mathrm{i})}$$

- We can use *matrix multiplication* to compute the predictions for all samples:

$$\begin{pmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{pmatrix} = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \quad \Rightarrow \quad \mathbf{\hat{Y} = Xw}$$

$$\mathcal{L}_E = \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \left(Y - \hat{Y}\right)^T \left(Y - \hat{Y}\right) \overset{\text{not}}{=} \mathbf{\left(Y - \hat{Y}\right)^2}$$

Multiple Linear Regression

$$\mathcal{L}_E = Y^T Y - 2w^T X^T Y + w^T X^T X w$$

Derivative w.r.t. to a vector ⟹ derivative w.r.t. each component.

$$\left[\frac{\partial \mathcal{L}_E}{\partial w_0} \quad \frac{\partial \mathcal{L}_E}{\partial w_1} \quad \dots\right]$$

- Minimize $\mathcal{L}_E$ with respect to w:

$$\frac{\partial \mathcal{L}_E}{\partial w} = 0 \Rightarrow -2X^T Y + 2X^T X w = 0 \Rightarrow X^T X w = X^T Y \Rightarrow$$

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

- Observations:
  - $X^T X$ needs to be *invertible*.
  - If $X \in \mathbb{R}^{m \times n}$ with $n > m$ (more features than examples), there is a high chance of $X^T X$ being singular.
    - The problem is *"ill-posed"*.

Well-posed vs. Ill-posed problems

- A problem is **well-posed** if:
  - A solution exists.
  - The solution is unique.
  - The solution's behavior changes continuously with the initial conditions.

Jacques Hadamard, 1902

- If a problem is not well-posed, it is said to be **ill-posed**.
  - This usually implies that additional assumptions need to be taken into consideration for numerical treatment of the problem.
  - This process is called **regularization**
  - Inverse problems are often ill-posed (determining the cause by observing the effects)

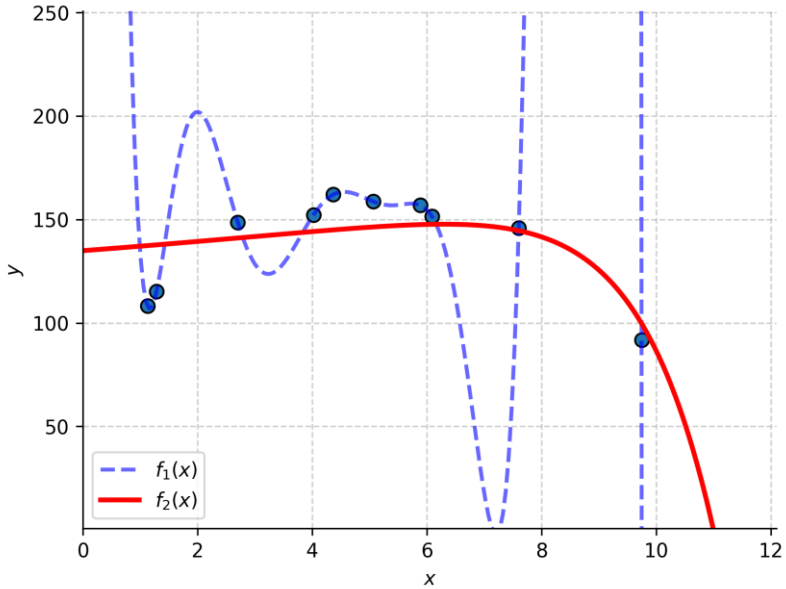
Ill-posed problems

- Lets consider a dataset $X \in \mathbb{R}^{10\times10}$ (10 samples with 10 features each)
  - For ease of plotting let's consider the features to be $[1 \quad x \quad x^2 \quad \dots \quad x^{10}]$

"Ill-posed" problem + unconstrained hypothesis space → data is overfitted.

$$f_1(x) = 5.28 * 10^3 - 1.50 * 10^4 x + 1.72 * 10^4 x^2 - 1.01 * 10^4 x^3 + \cdots$$

Large coefficients →
- Small changes in input cause large changes in output.



What if we *force coefficients to be small*?

$$f_2(x) = 1.34 * 10^1 + 2.09 * 10^1 x + 7.11 * 10^{-2} x^2 - 6.24 * 10^{-4} x^3 + \cdots$$

Small coefficients →
- Function is much smoother.

Ridge Regression

- We want small coefficients, so we add the norm of the weight vector to the loss:

$$\mathcal{L}_E = \sum_i \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|\vec{w}\|_2^2$$

L_2 regularization

- In matrix format:

$$\mathcal{L}_E = \left(Y - \hat{Y}\right)^2 + \lambda w^T w$$

$$\frac{\partial \mathcal{L}}{\partial w} = 0 \Rightarrow$$

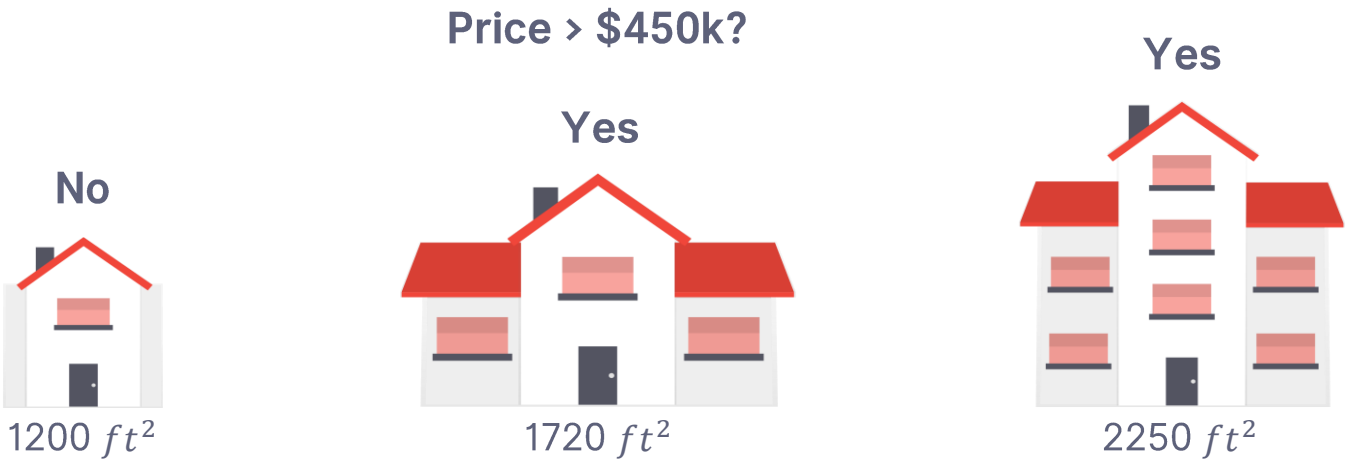
$$\boldsymbol{w = \left(X^T X + \lambda I\right)^{-1} X^T Y}$$

Always invertible.

Logistic Regression

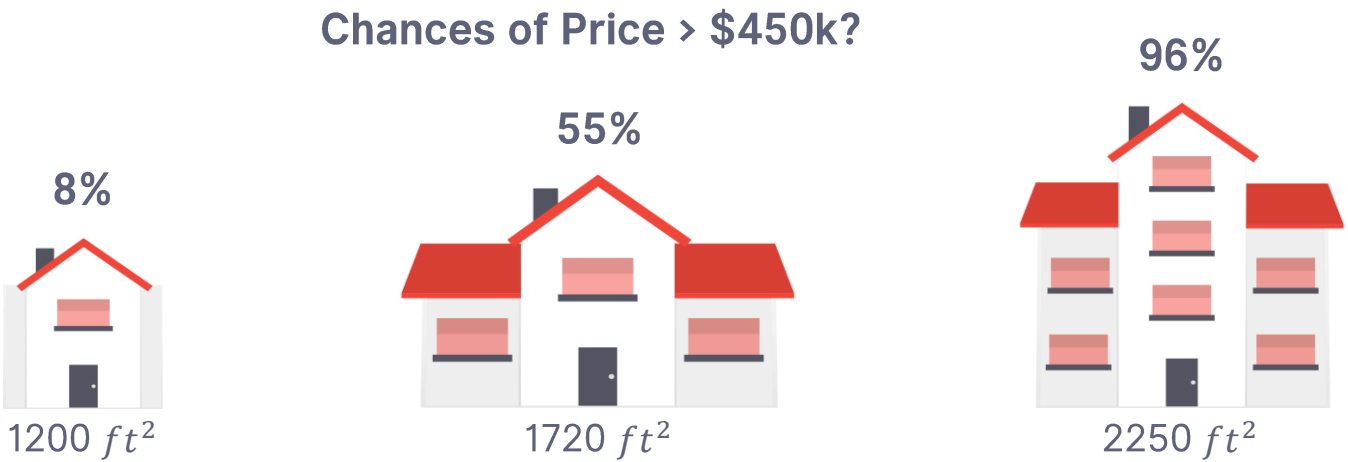
Classification vs. Regression

- Previously, we wanted to predict the price of a house, given its size.
- Now, we want to predict if a house costs more than $450k or not.
  - Not only that, but we only have access to a set of samples with this information.



Classification vs. Regression

- Previously, we wanted to predict the price of a house, given its size.
- Now, we want to predict if a house costs more than $450k or not.
  - Not only that, but we only have access to a set of samples with this information.
  - Even better, we want a model which gives us the **likelihood** of a house costing more than $450k



Classification vs. Regression

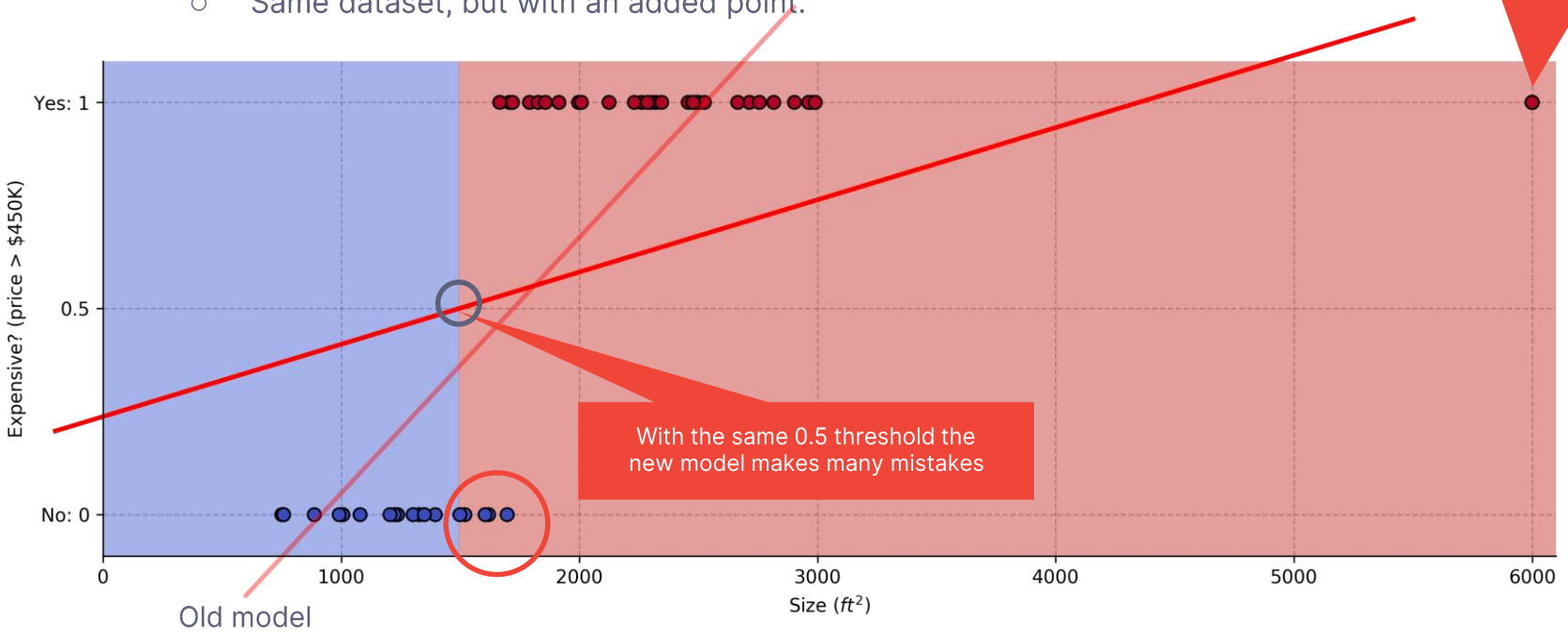
- Set of *expensive / non-expensive* house observations.
  - Is there anything wrong with this approach?

How do we set the threshold? Does 0.5 really make sense?

Some outputs are < 0 and some are > 1, so we can't interpret them as probabilities

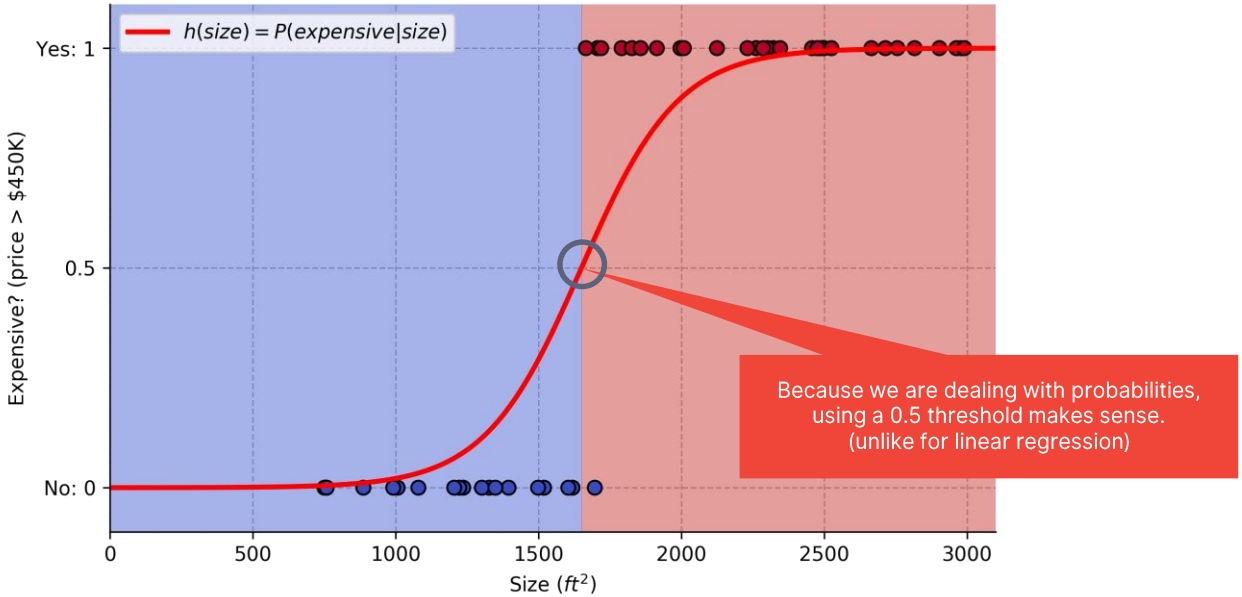
Classification vs. Regression

- Set of *expensive / non-expensive* house observations.
  - Same dataset, but with an added point.



Predicting probabilities

- Set of *expensive / non-expensive* house observations.
  - We want to learn a model *h*, which gives us the **probability** of a house being expensive, given its size.



Logistic Function

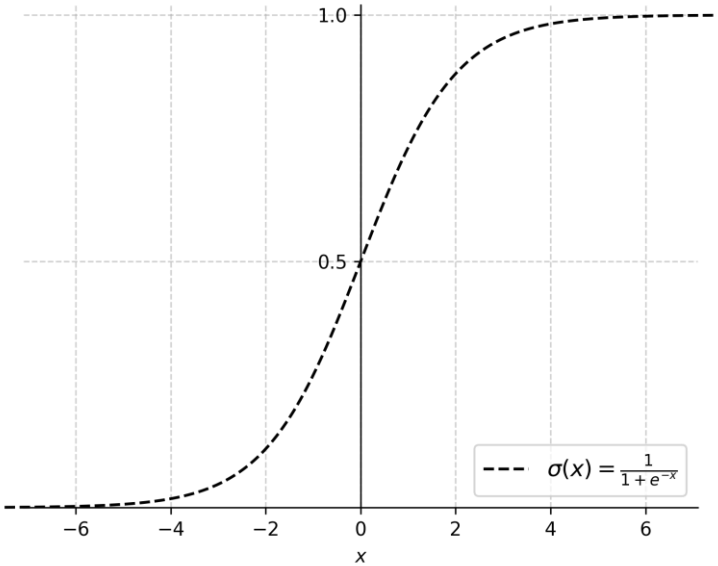
- **Sigmoid** function ("S-shaped" curve):
  - Family of functions which are bounded, differentiable, real and with a non-negative derivative at each point.
- Logistic function (special case of sigmoid):

$$\sigma(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

- **Standard logistic function** $(\mathrm{L} = 1, \mathrm{k} = 1, \mathrm{x}_0 = 0)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  - $\sigma(-x) = 1 - \sigma(x) = \frac{1}{1+e^{x}}$
  - $0 < \sigma(x) < 1 \Rightarrow$ Can be interpreted as **probability**



Logistic Model

- The relationship is modeled with a **logistic model**:
  - $\vec{x} \in \mathbb{R}^n$ represents the *independent variables*
  - $y \in \{0, 1\}$ is the *dependent variable*

$$\hat{y} = \sigma(w_0 + w_1 x_1 + \cdots + w_n x_n) = \frac{1}{1 + e^{-\langle \vec{w}, \vec{x} \rangle}}$$

Same trick as before: $x_0 = 1$

  - In other words, we apply a *logistic function on top of a linear model*.

- Prediction can be interpreted as **probability**:

$$\hat{y} = P(y = 1 | \vec{x}, \vec{w})$$

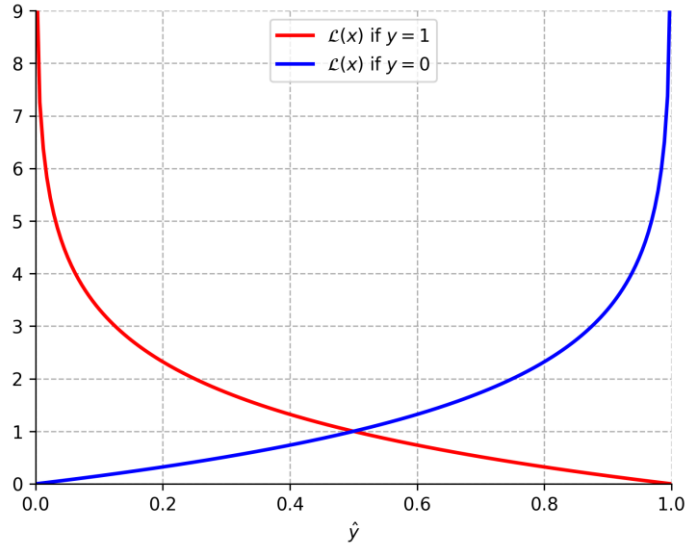
Loss function

- **Cross-entropy** for a single example:

$$\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) = \begin{cases} -\log \hat{y}^{(i)} & \text{if } y^{(i)} = 1 \\ -\log\left(1 - \hat{y}^{(i)}\right) & \text{if } y^{(i)} = 0 \end{cases}$$

  - $\mathcal{L} \to 0$ as $\hat{y}^{(i)} \to y^{(i)}$
  - $\mathcal{L} \to \infty$ as $\hat{y}^{(i)} \to 1 - y^{(i)}$

Loss grows exponentially if model is very confident in the wrong prediction.

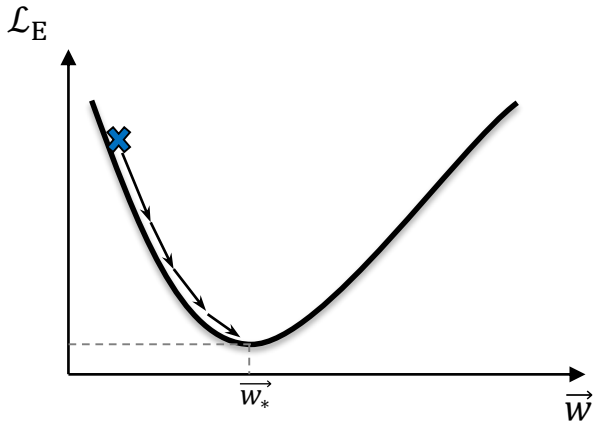


- Combining both branches and summing over all samples:

$$\mathcal{L}_{\mathrm{E}} = -\sum_{i} \left[y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right) \right]$$

Optimization

- We need to find $\overrightarrow{w_*} = \underset{\vec{w}}{\operatorname{argmin}}\, \mathcal{L}_{\mathrm{E}}(\vec{w})$
- There is no *closed-form solution* for finding the minimum of the log-loss.
  - i.e. We cannot compute it directly through mathematical operations, like for linear regression
  - We need to use an *optimization method.*
- Gradient Descent
  - Gradually go down the slope of the error function
- More complex methods:
  - Conjugate Gradient
  - BFGS
  - L-BFGS



What if we have multiple classes?

- We want to know *how likely it is* that a certain dog is of one of four breeds.

Weight = *25 kg*
Height = *46 cm*
Fur length = *0*
Ear length = *3 cm*
Color = *Cream*

Multinomial Logistic Regression

10%

German Sheppard

15%

Labrador Retriever

70%

Shar Pei

5%

Collie

Predictions should sum up to 100%.

The predicted class is the one with the highest probability.

What if we have multiple classes?

- If we have K classes, we predict a probability for each class:
  - The prediction $\hat{y}^{(i)}$ becomes array of probabilities.
  - ${\hat{y}_k}^{(i)}$ is the probability of $\vec{x}^{(i)}$ being class k (e.g. $\hat{y}^{(i)} = [0.1 \quad 0.15 \quad 0.7 \quad 0.05]$)
  - Training samples: $y_k^{(i)} = \begin{cases} 1 & \text{if } \vec{x}^{(i)} \text{ is class } k \\ 0 & \text{otherwise} \end{cases}$ (e.g. $y^{(i)} = [0 \quad 0 \quad 1 \quad 0]$)

The predicted class is the one with the highest probability.

- **Multinomial Cross-Entropy**:

$$\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) = \sum_{k=1}^{K} y_k^{(i)} \log {\hat{y}_k}^{(i)}$$

Only one term will be non-zero

Other strategies for multiclass classification

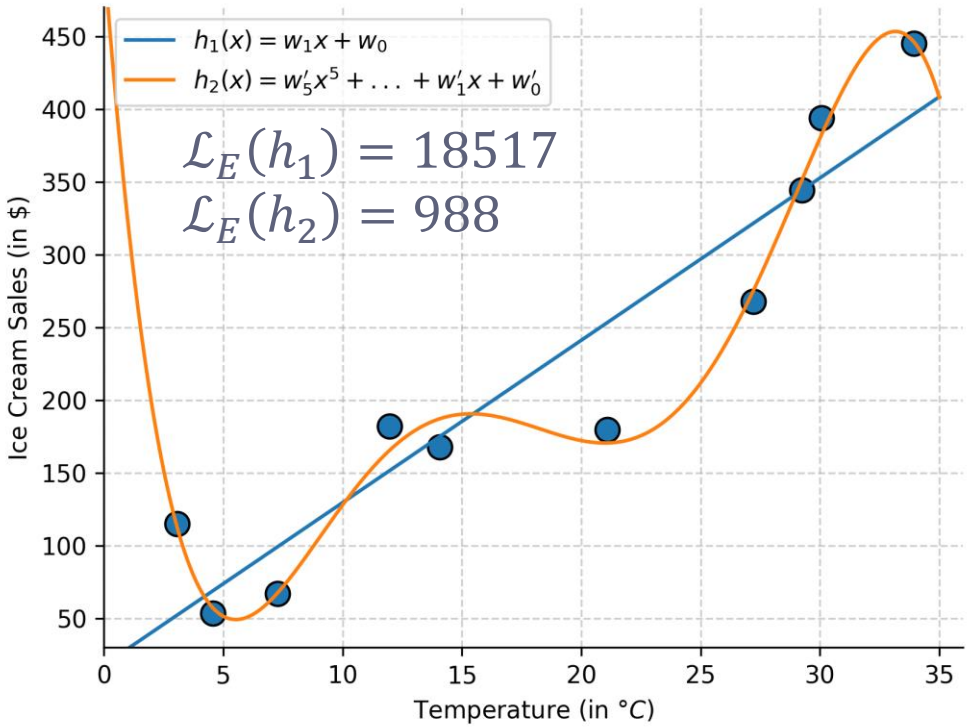
- Another method of having multiple classes is to fit multiple binary classifiers independently and combine their predictions.

- **One-versus-rest** (OVR), sometimes called (one-versus-all, OVA)
  - Train n classifiers, one for each class, where the negative examples are all the other classes.
  - At inference, run all classifiers and pick the class with the highest margin (most confident)

- **One-versus-one** (OVO)
  - Train $n(n-1)/2$ classifiers, one for each pair of classes.
  - At inference, run all classifiers and pick the class which was selected by most of them

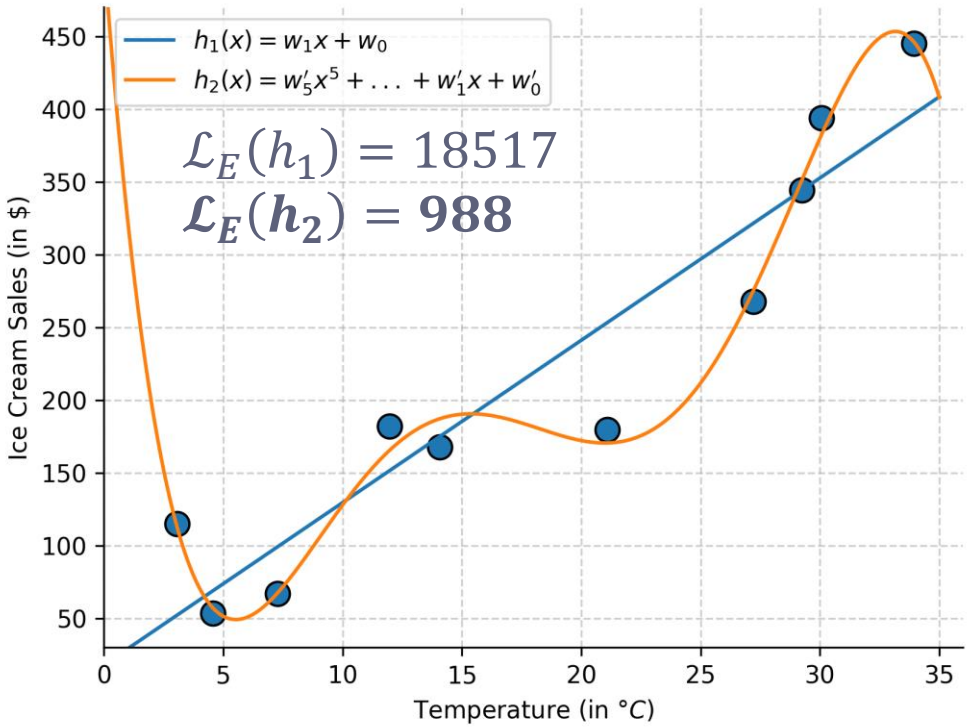
Model Evaluation

Which model is better?



Which model is better?

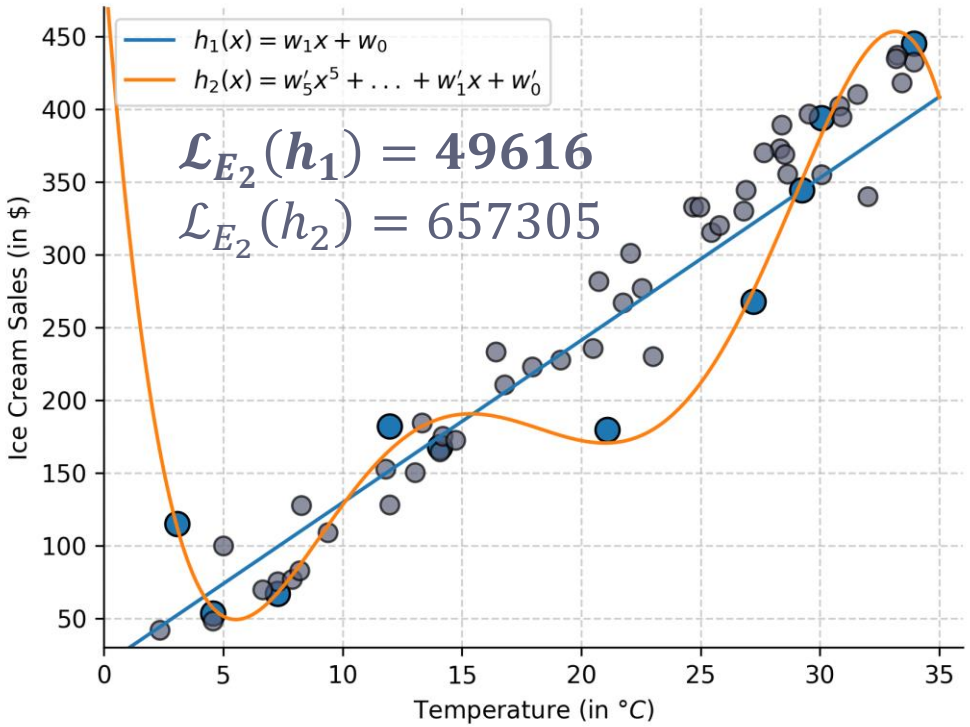
Which model is better?



Which model is better?

- h_2, according to training loss.

Which model is better?



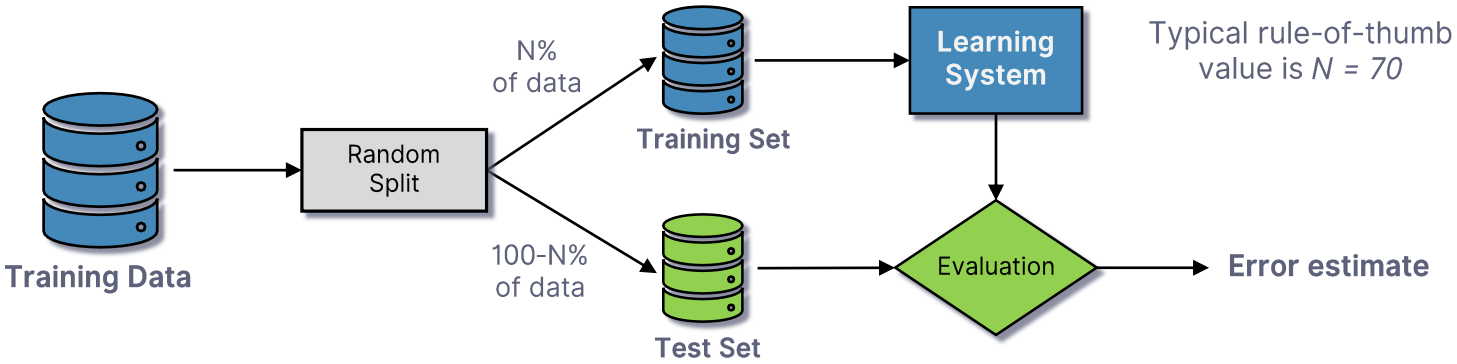
Which model is better?

- h_2, according to training loss.
- But when we evaluate on other points, h_1 performs much better.

In practice, we don't always get to do this.

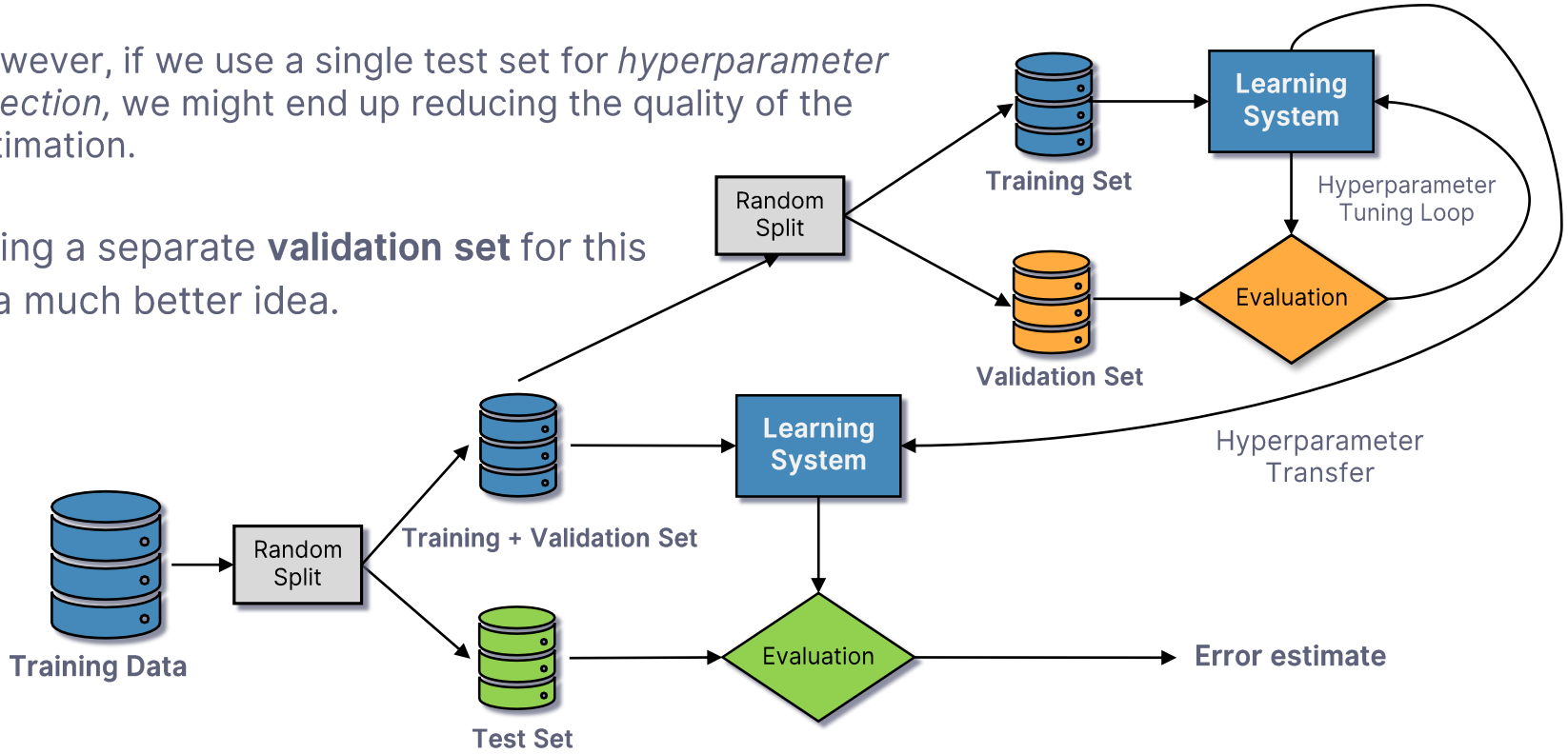
Training – Test split

- Using training error for performance evaluation is usually a bad idea.
  - It is a very optimistic estimate of true error and it favors models which overfit.
- **Holding out data for testing** gives a much better estimate.



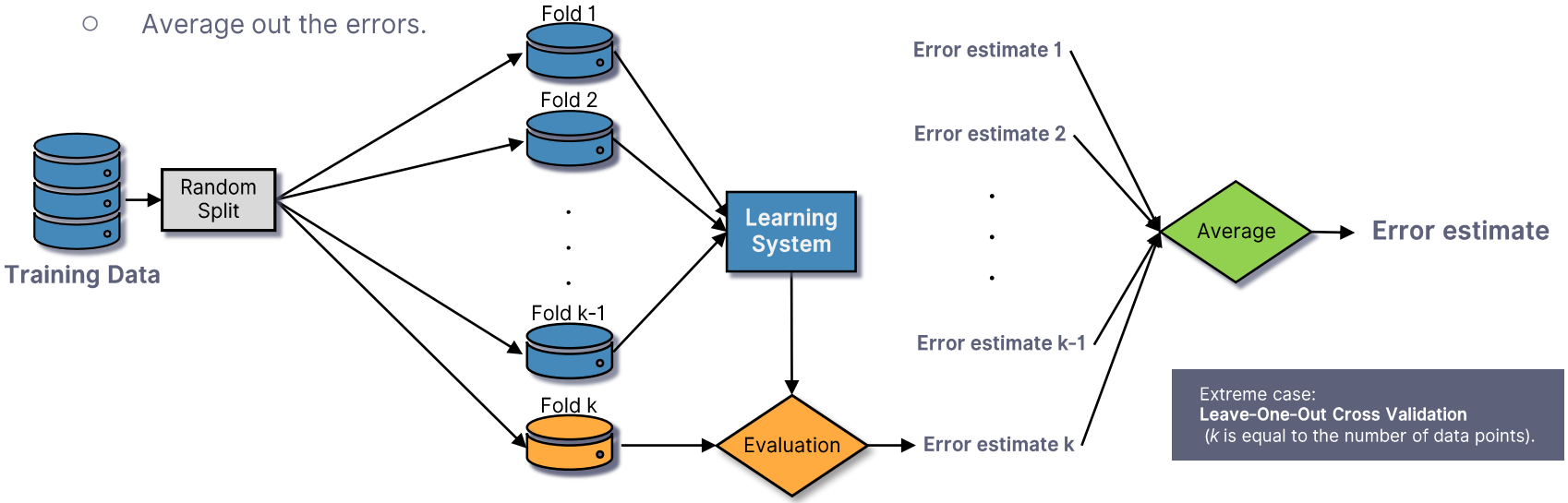
Training– Validation – Test split

- However, if we use a single test set for *hyperparameter selection*, we might end up reducing the quality of the estimation.
- Using a separate **validation set** for this is a much better idea.



K-fold Cross-Validation

- Sometimes we don't have enough data to "*afford*" a validation set.
  - The remaining training set would simply be too small to fit a model on.
- **K-fold Cross-Validation**:
  - Split the data into k equal parts (a.k.a. *folds*)
  - Repeat the train – test process k times, each time using one fold for testing and the rest for training.
  - Average out the errors.

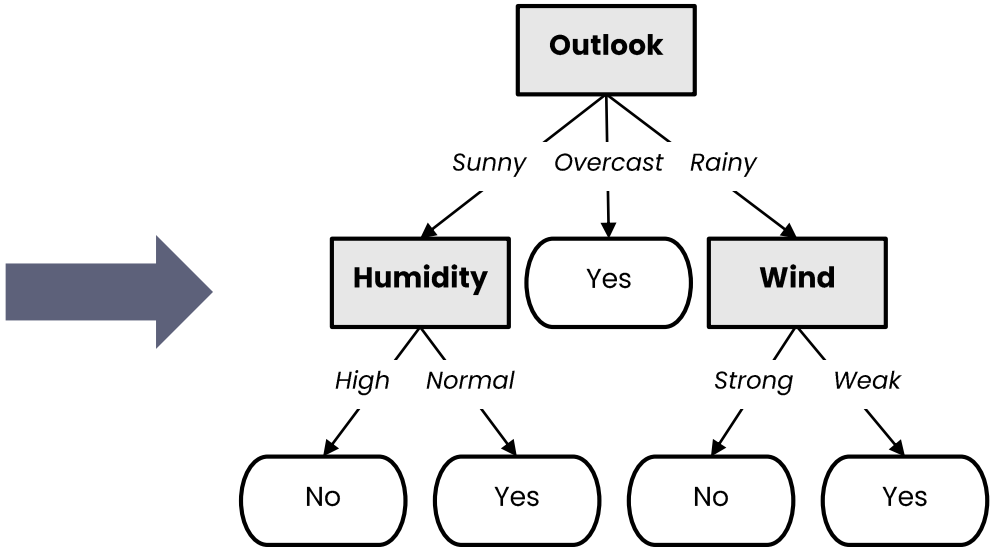


Decision Trees

Decision Tree Learning

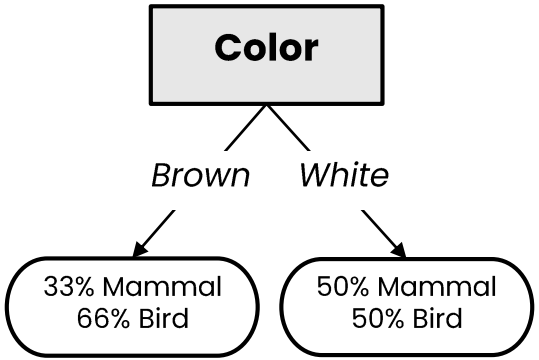
- In ML, we want to use labeled data to obtain a *suitable* decision tree for future predictions.
  - We want a tree which works well on unseen data, while asking as few questions as possible

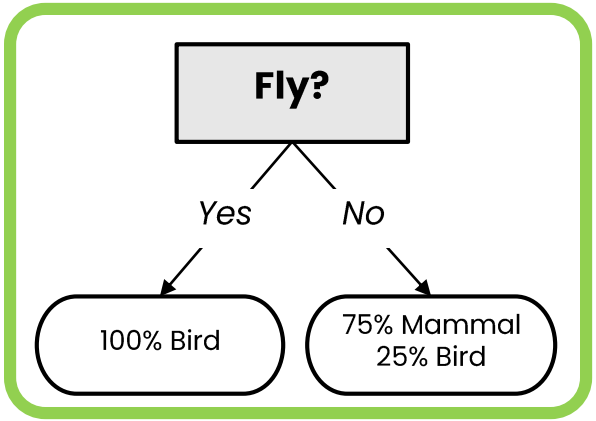
Outlook	Temperature	Humidity	Wind	Play Tennis?
Sunny	Hot	High	Weak	**No**
Sunny	Hot	High	Strong	**No**
Overcast	Hot	High	Weak	**Yes**
Rainy	Mild	High	Weak	**Yes**
Rainy	Cool	Normal	Weak	**Yes**
Rainy	Cool	Normal	Strong	**No**
Overcast	Cool	Normal	Strong	**Yes**
Sunny	Mild	High	Weak	**No**
Sunny	Cool	Normal	Weak	**Yes**
Rainy	Mild	Normal	Weak	**Yes**
Sunny	Mild	Normal	Strong	**Yes**
Overcast	Mild	High	Strong	**Yes**
Overcast	Hot	Normal	Weak	**Yes**
Rainy	Mild	High	Strong	**No**



What makes a "good" attribute?

Does it fly?	Color	Class
No	Brown	**Mammal**
No	White	**Mammal**
Yes	Brown	**Bird**
Yes	White	**Bird**
No	White	**Mammal**
No	Brown	**Bird**
Yes	White	**Bird**

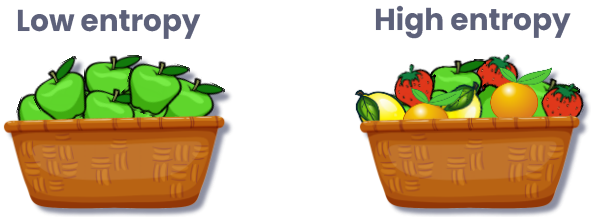




- Which one is a *better* attribute for splitting?
- Why?
  - Because the resulting subsets are more "*pure*".
  - Knowing the value of this attribute gives us "*more information*" about the label.
    - i.e. The **entropy** of the subsets is lower.

Entropy

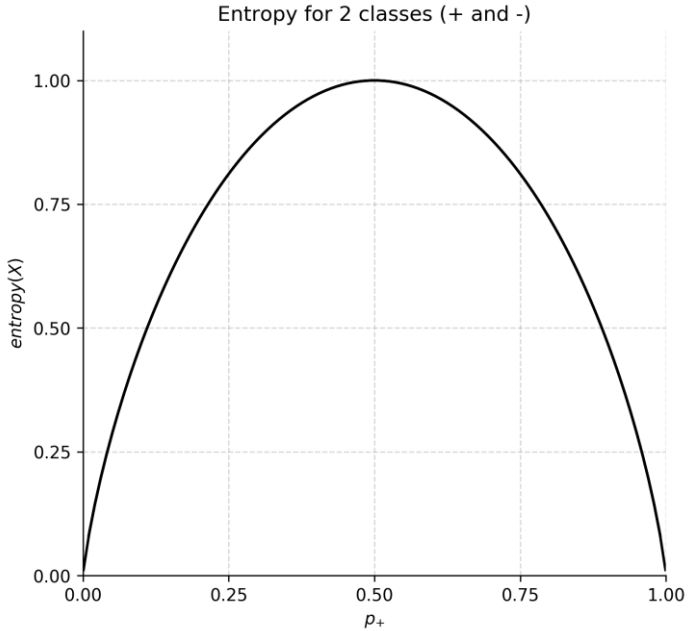
- **Entropy** measures the *degree of randomness* in a dataset, with lower entropy implying greater *predictability*.



- For a set of samples X with k classes:

$$\textbf{entropy}(X) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

where p_i is the proportion of elements with class i



Information Gain

- The **information gain** of an attribute a is the expected *reduction in entropy* due to splitting on values of a:

$$\mathbf{gain}(X, a) = \mathbf{entropy}(X) - \sum_{v \in \mathbf{values}(a)} \frac{|X_v|}{|X|} \mathbf{entropy}(X_v)$$

where X_v is the subset of X for which $a = v$.

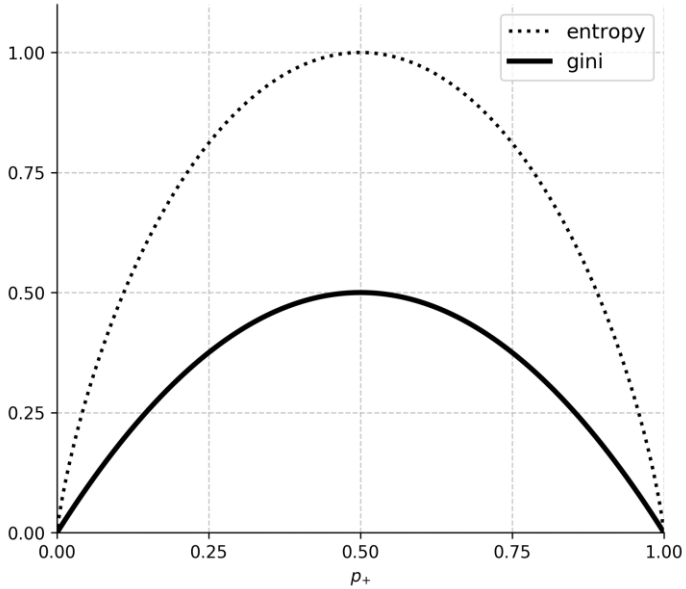
Gini impurity

- Measures how often a randomly chosen element would be incorrectly labeled if it was randomly labeled according to the distribution of labels.

Error of classifying randomly picked fruit with randomly picked label

$$\text{gini}(X) = 1 - \sum_{i=1}^{k} p_i^2$$

- Can be used as an alternative to entropy for selecting attributes.



Decision Tree Learning Algorithm

```
1   def extract_node(X):
2       node = TreeNode(X)
3       if should_be_leaf_node(X):
4           node.label = majority_label(X)
5       else:
6           a = select_best_splitting_attribute(X)
7           for v in values(a):
8               X_v = {x ∈ X | x[a] == v}
9               node.children.append(extract_node(X_v))
10      return node
```

C4.5 Algorithm

- **C4.5** is a an extension to the ID3 algorithm which brings several improvements:
  - Ability to handle *both discrete and* ***continuous attributes***. Continuous attributes are split by finding a best-splitting threshold.
  - Ability to handle ***missing attributes*** both at training and inference time. Training missing values are not used in information gain computation. Inference missing values are handled by considering all subsequent branches.
  - Ability to handle ***attributes with different costs***.
  - Post-**pruning** in a bottom-up manner for removing branches which decrease validation error.

Random Forests

- **Random Forests** basic idea:
  - Instead of *growing* a single decision tree and use it to make predictions, grow many slightly different trees and combine their predictions.
- We have one dataset, so how do we obtain *slightly different* trees?
  - **Bagging** (**B**ootstrap **Agg**regat**ing**)
    - Take random subsamples of data points from the large dataset to create *N* smaller datasets
    - Fit a different model on each of small datasets.
  - **Random Subspace Method** (a.k.a. Feature Bagging)
    - Fit *N* different models, but allow each one to see a random subsample of features from the original dataset.

Ensemble Learning

- **Ensemble Learning** is a generic term for methods which combine *multiple learning algorithms* to obtain better performance than could be obtained from any constituent algorithm alone.
- *Random Forests* are one of the most common instances of ensemble learning.
- Common ensemble techniques:
  - **Bagging** – multiple models on random subsamples of data
  - **Random Subspace Method** – multiple models on random subsamples of features
  - **Boosting** – iteratively train models, while making the current model focus more on the mistakes of previous ones by increasing the weight of misclassified samples.

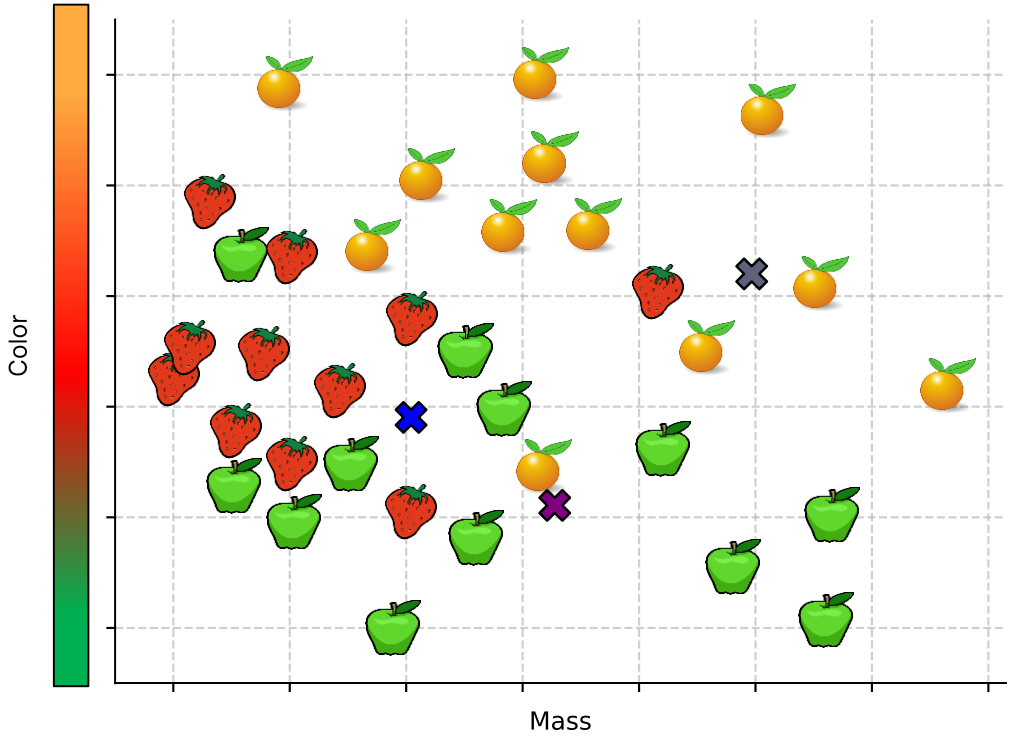
K-Nearest Neighbors

How it works?

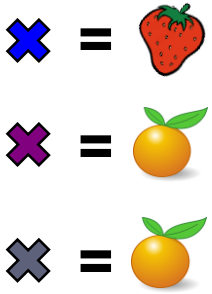
✖ = ?

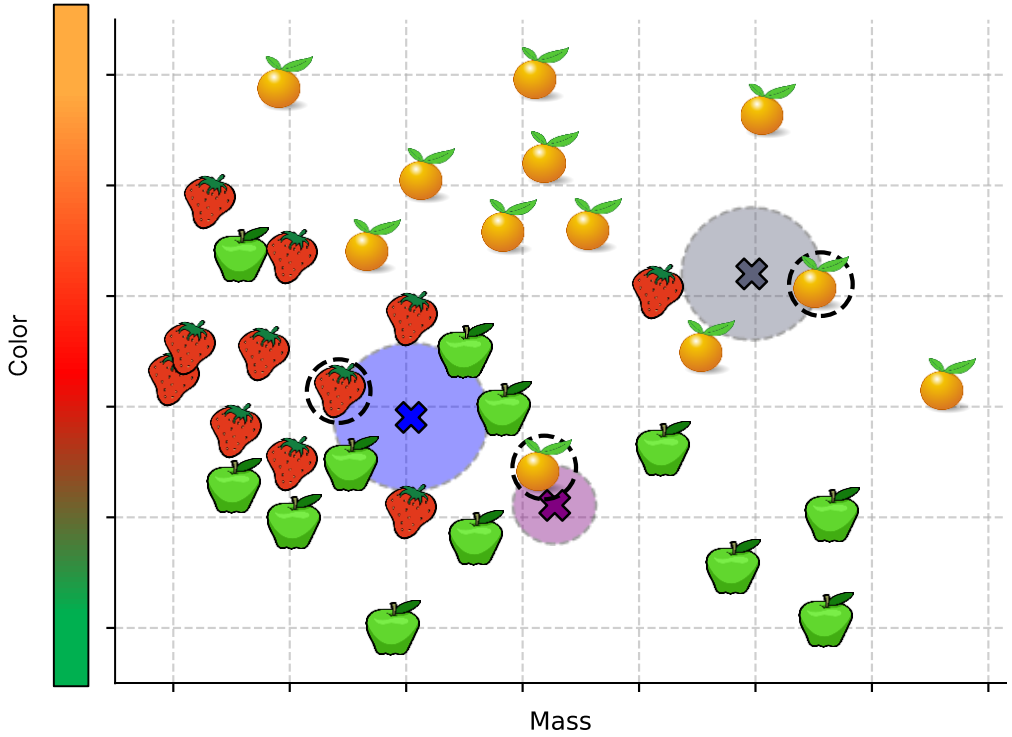
✖ = ?

✖ = ?



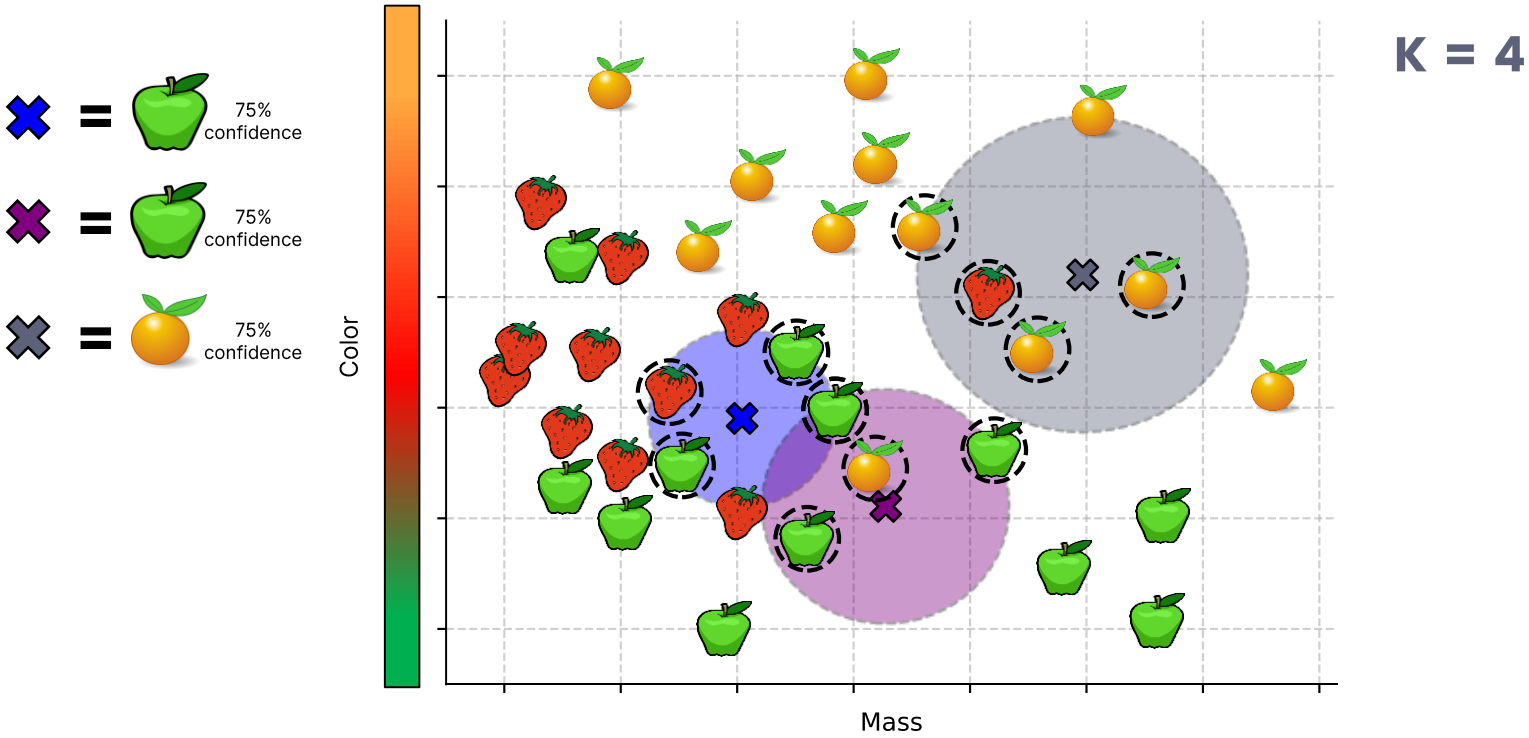
How it works?



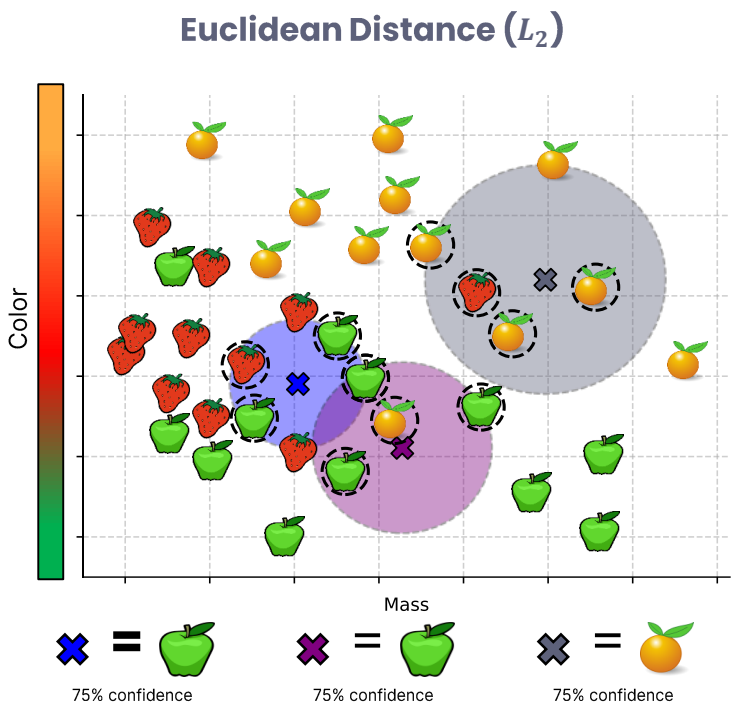


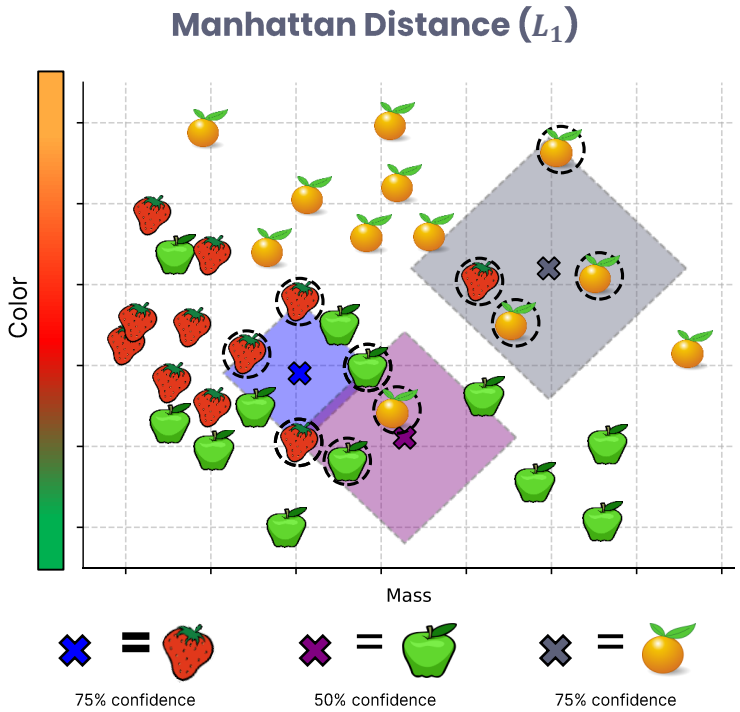
K = 1

How it works?



Distance metric matters

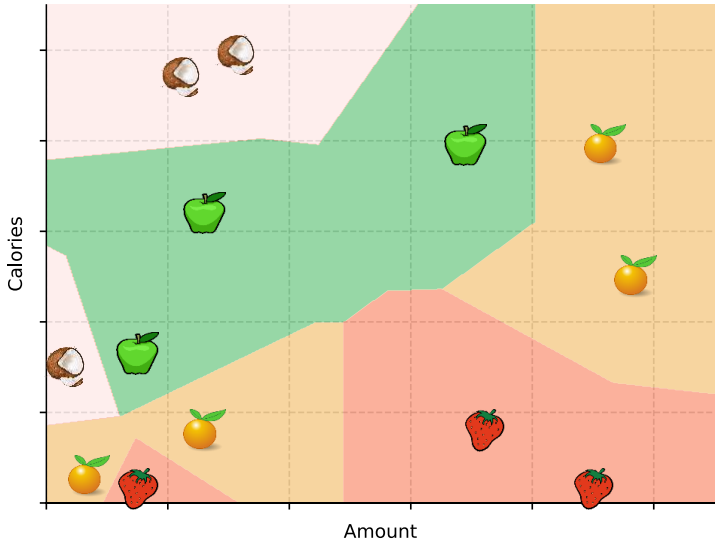




Distance metric matters

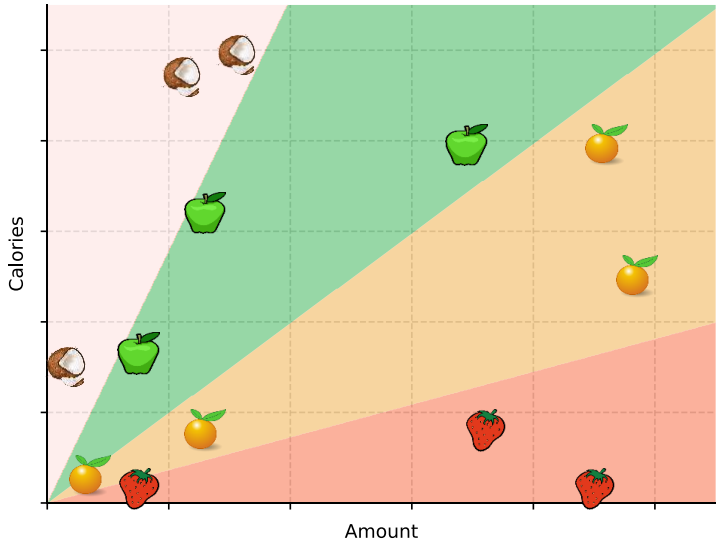
- Task: Predict the *type of fruit* based of a set of samples for which we know the *amount* and *calories* contained in it.

Euclidean Distance



vs.

Cosine Distance



Feature Normalization

- If features have different scales, some features might dominate others when computing distances.
- We must bring all feature values to the same numeric range:
  - **Min-max scaling** – bring all values to a fixed range, usually between 0 and 1:

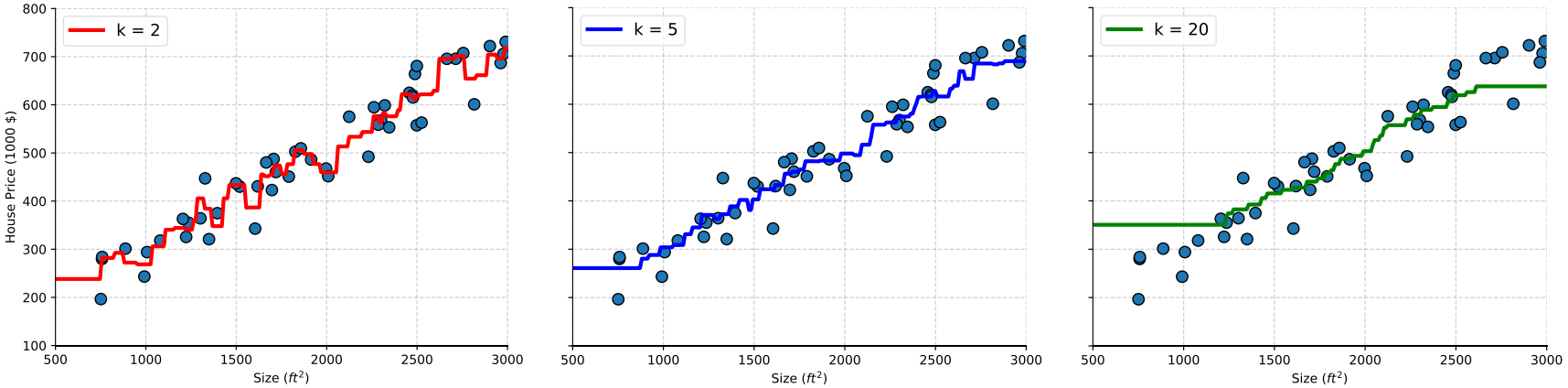
$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

  - **Standardization (Z-score normalization)** – scale features so that they will have the properties of a *standard normal distribution* (mean $\mu = 0$ and standard deviation $\sigma = 1$):

$$X_{norm} = \frac{X - \mu_X}{\sigma_X}$$

KNN Regression

- It is possible to use kNN for *regression* problems.
  - Predicted label is the **average label** of the input's *k* nearest neighbors

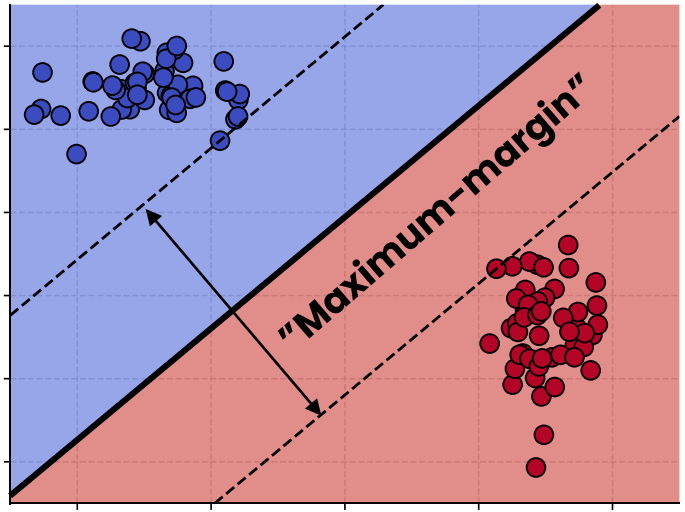


- Increasing *k* makes the function smoother.
  - But it also increases the area at the edges of the training data for which the prediction is constant.

Support Vector Machines

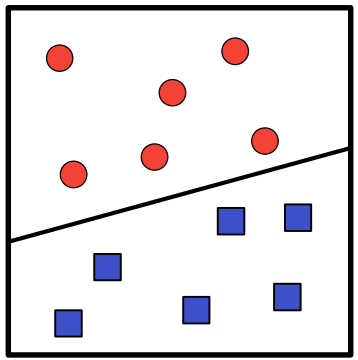
Decision Boundaries

- An SVM tries to find a *separating* hyperplane, which is as far away from all training points at the same time (a **maximum-margin hyperplane**)
  - A point is classified as “+” or “-”, depending on which part of the hyperplane it lies.

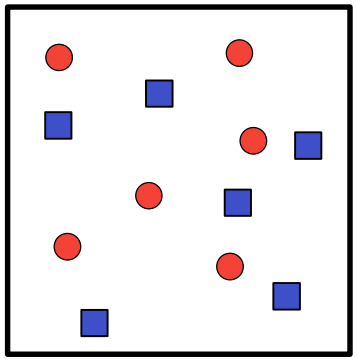


Linear Separability

- Two sets of points are **linearly separable**, if there is at least one **hyperplane** which completely separates them.
- An **n-dimensional hyperplane** is a flat *n-1* dimensional subset of the space.
  - A *1d* hyperplane is a **point.**
  - A *2d* hyperplane is a **line.**
  - A *3d* hyperplane is a **plane.**

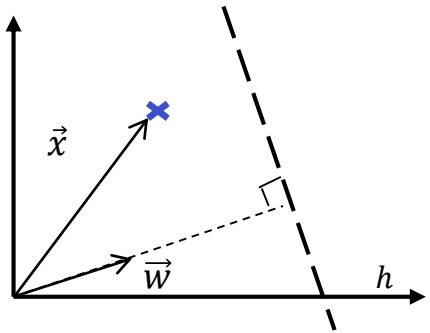


Separable



Non-separable

Decision Rule



$\langle \vec{x}, \vec{w} \rangle + b > 0 \Rightarrow \vec{x}$ is on the right side of h

$\langle \vec{x}, \vec{w} \rangle + b < 0 \Rightarrow \vec{x}$ is on the left side of h

$\langle \vec{x}, \vec{w} \rangle + b = 0 \Rightarrow \vec{x}$ is exactly on h

$|\langle \vec{x}, \vec{w} \rangle + b|$ is proportional to the distance from h

In fact, we can say that h is determined by $\vec{w}$ and b

- SVM decision rule:

$\vec{x}$ is a "+" sample if $\langle \vec{x}, \vec{w} \rangle + b \geq 0$

$\vec{x}$ is a "-" sample otherwise

- Which $\vec{w}$? Which b?

Learning a “good” separating hyperplane

- Training set: $E = \{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\},\ \vec{x}^{(i)} \in \mathbb{R}^n,\ y^{(i)} \in \{+1, -1\}$
- We want to separate the ”+” and ”-” training samples and have **some margin between the classes:**

$$\langle \overrightarrow{x_+}, \vec{w} \rangle + b \geq \mathbf{1} \qquad \forall \overrightarrow{x_+} \in \{\vec{x}^{(i)} | y^{(i)} = +1\}$$
$$\langle \overrightarrow{x_-}, \vec{w} \rangle + b \leq -\mathbf{1} \qquad \forall \overrightarrow{x_-} \in \{\vec{x}^{(i)} | y^{(i)} = -1\}$$

± 1 forces a gap between points.

- Combining the two inequations:

$$y^{(i)}\left(\langle \vec{x}^{(i)}, \vec{w} \rangle + b\right) - 1 \geq 0$$

- For examples $(\vec{x}^{(i)}, y^{(i)})$ which lie exactly on the edges of the gap:

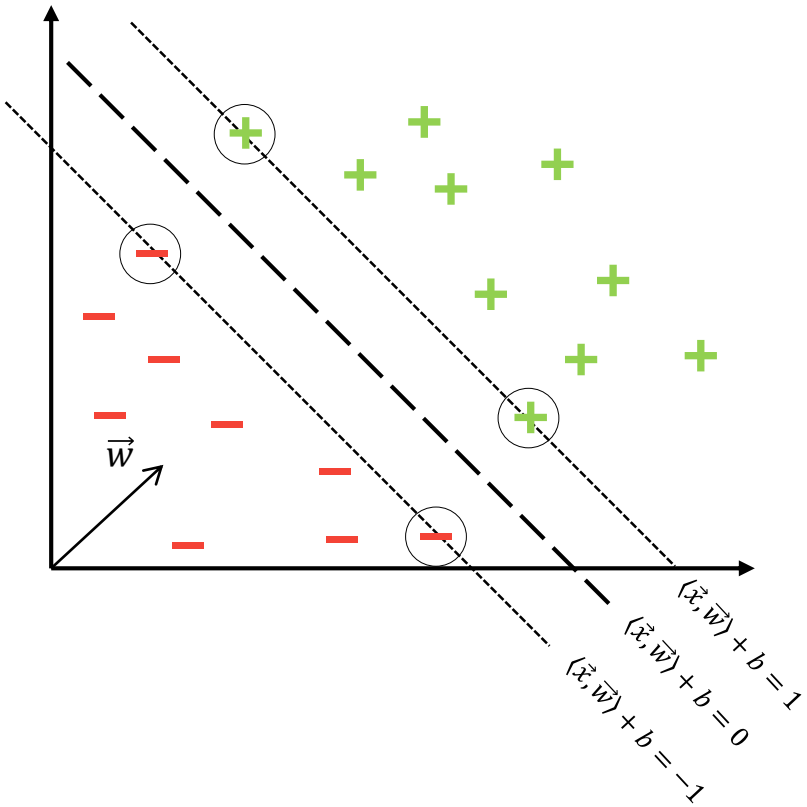
$$y^{(i)}\left(\langle \vec{x}^{(i)}, \vec{w} \rangle + b\right) - 1 = 0$$

Learning a "good" separating hyperplane

- For training examples $\left(\vec{x}^{(i)}, y^{(i)}\right)$, which lie exactly on the edges of the gap:

$$\boldsymbol{y^{(i)}\left(\left\langle \vec{x}^{(i)}, \vec{w} \right\rangle + b\right) - 1 = 0}$$

- We call these examples "**Support Vectors**"



Making the margin "wide"

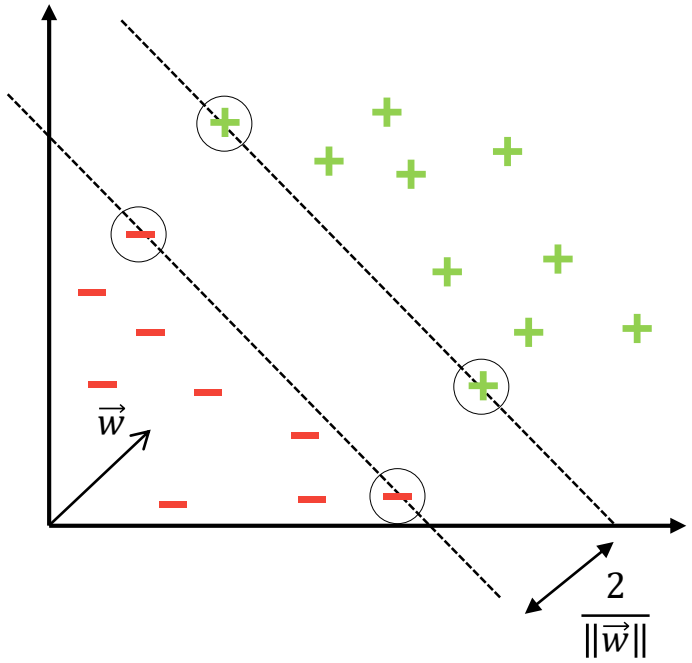
- How do we express the width of the gap?

$$g = \frac{2}{\|\vec{w}\|}$$

- We want to *maximize the gap:*

$$\text{maximize} \frac{2}{\|\vec{w}\|} \Rightarrow \text{minimize } \|\vec{w}\| \Rightarrow$$

$$\textbf{minimize} \frac{\|\vec{w}\|^2}{2}$$



SVM Primal Form

- The decision rule is:

$$\vec{x} \text{ is a “+” sample if } \langle \vec{x}, \vec{w} \rangle + \mathbf{b} \geq \mathbf{0}$$

- In order to obtain $\vec{w}$ and b we need to:

$$\text{minimize } \frac{\|\vec{w}\|^2}{2}$$

$$\text{subject to } y^{(i)}\left(\left\langle \vec{x}^{(i)}, \vec{w} \right\rangle + b\right) - 1 \geq 0$$

SVM Dual Form

- Using *Lagrange multipliers* on the primal form brings us to a different optimization method, called the **"dual form".**
- The decision rule is:

$$\vec{x} \text{ is a "+" sample if } \sum_i \alpha_i y^{(i)} \langle \vec{x}^{(i)}, \vec{x} \rangle + b \geq 0$$

Optimization and decision depend only on **dot product with training samples.**

- In order to obtain α_i and b we need to:

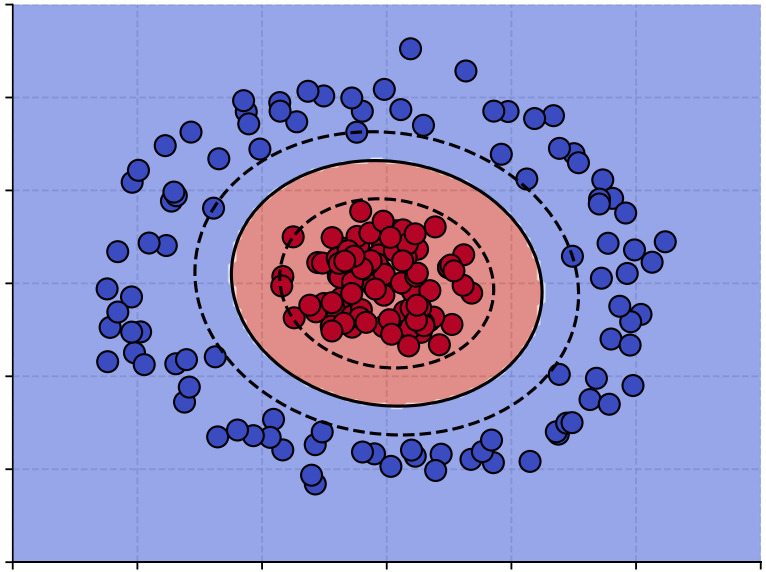
$$\text{minimize } \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y^{(i)} y^{(j)} \langle \vec{x}^{(i)}, \vec{x}^{(j)} \rangle$$

$$\text{subject to } \alpha_i \geq 0 \; \forall i$$

Kernel Trick

- The **kernel trick** means replacing the dot product with a *kernel function*, which creates an *implicit* high-dimensional feature space.
- Any algorithm which uses this approach is called a **kernel method**
- The kernel function can be viewed as a *similarity function* between data points.

The decision boundary is no longer linear in the original space.



Soft-margin SVM

- *Soft-margin SVM*: Allow some training samples to be misclassified, at a cost.
- We introduce "slack variables" $\xi_i \geq 0$ (How much example $\vec{x}^{(i)}$ is allowed to cross the edge).

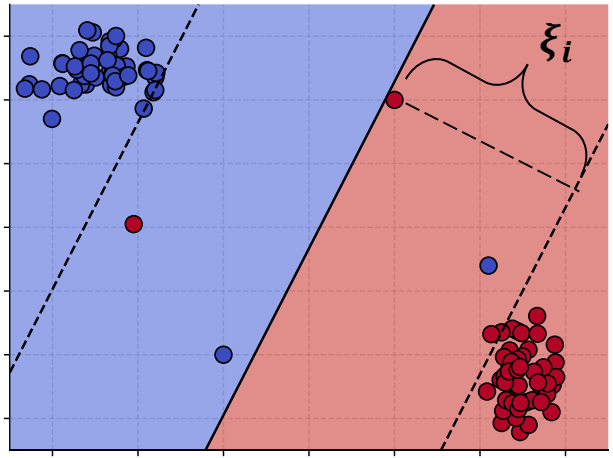
$y^{(i)}\left(\langle \vec{x}^{(i)}, \vec{w} \rangle + b\right) - 1 \geq 0$ becomes $y^{(i)}\left(\langle \vec{x}^{(i)}, \vec{w} \rangle + b\right) - 1 \geq -\xi_i$

- The optimization problem becomes:

$$\textbf{minimize} \frac{\|\vec{\mathbf{w}}\|^2}{2} + \mathbf{C}\sum_i \boldsymbol{\xi}_\mathbf{i}$$

$$\textbf{subject to } \boldsymbol{y^{(i)}\left(\langle \vec{x}^{(i)}, \vec{w} \rangle + b\right) \geq 1 - \xi_i}$$

- Tradeoff between making the margin wide and allowing training mistakes.
- *C* controls the weight of a mistake.



Hinge Loss

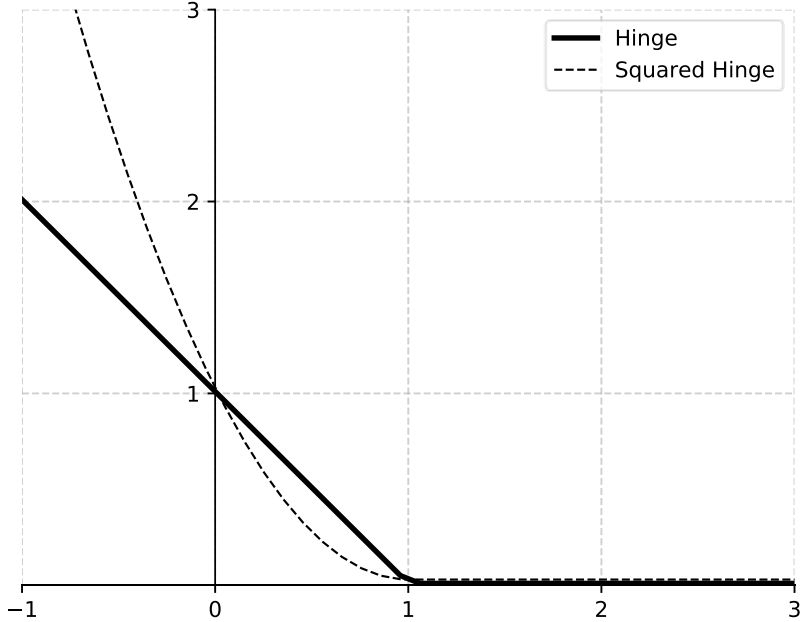
$$\text{minimize} \quad \left(\frac{\|\vec{w}\|^2}{2} + C \sum_i \max\left(0, 1 - y^{(i)}\left(\langle \vec{x}^{(i)}, \vec{w} \rangle + b\right)\right)\right)$$

$$\mathcal{L}(\vec{x}) \stackrel{\text{def}}{=} \max\left(0, 1 - y(\langle \vec{x}, \vec{w} \rangle + b)\right)$$

Hinge Loss

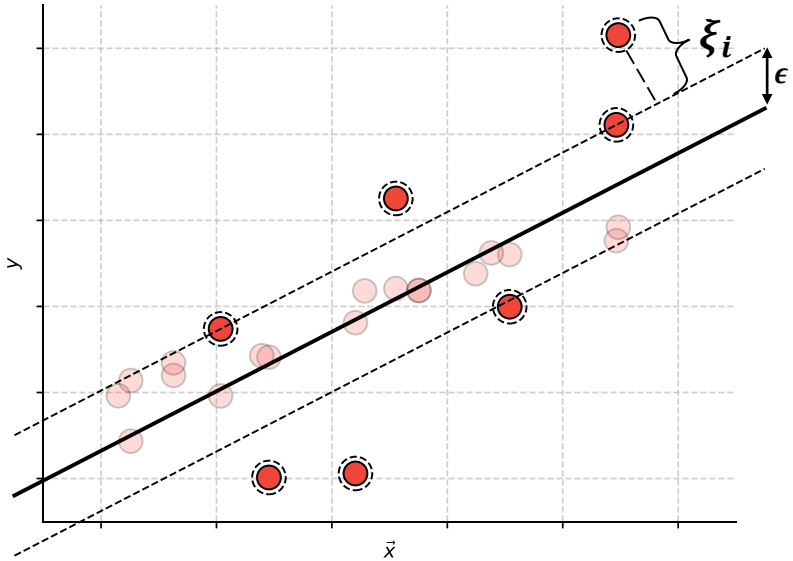
$$\Longrightarrow \text{minimize} \left(C \sum_i \mathcal{L}\left(\vec{x}^{(i)}\right) + \frac{\|\vec{w}\|^2}{2} \right)$$

What the soft-margin SVM is actually doing **is *minimizing the hinge loss*** with ***Tikhonov regularization***.



SVM Regression

- Inference – Use the SVM's decision function as prediction $\hat{y} = \langle \vec{x}, \vec{w} \rangle + b$
- Training – Find the *flattest* function which can fit the training data inside a tube of radius ϵ



- Sometimes it is impossible to find a function to find all the training inside a $\pm\epsilon$ range.
- We introduce "slack variables":

$$\textbf{minimize} \quad \frac{\|\vec{w}\|^2}{2} + C\sum_i \xi_i$$
$$\textbf{subject to } \left|y^{(i)} - \langle \vec{x}^{(i)}, \vec{w} \rangle - b\right| \leq \epsilon + \xi_i$$

or

$$\mathcal{L}(\vec{x}) \stackrel{\text{def}}{=} \max(0, |y - \langle \vec{x}, \vec{w} \rangle - b| - \epsilon)$$

ϵ-insensitive Loss

$$\Rightarrow \textbf{minimize} \left(C\sum_i \mathcal{L}(\vec{x}^{(i)}) + \frac{\|\vec{w}\|^2}{2} \right)$$

K-Means Clustering

Clustering

- **Clustering** of **Cluster Analysis** is the task of grouping a set of objects in such a way that objects in the same group (called a **cluster**) are *more similar* to each other than to objects in other groups.
- There are several types of models:
  - *Centroid-based* – Each cluster is represented by a prototype point (a “center”) (e.g. K-means)
  - *Density-based* – Clusters a considered denser regions of space. (e.g. DBSCAN)
  - *Hierarchical models* – There is a hierarchical relationship between clusters.
  - *Distribution models* – Clusters are modeled using statistical distributions. (e.g. GMM)
  - *Graph-based* – Clusters are considered cliques in a graph (e.g. HCS)
  - *Others*
- Based on the relation between objects and clusters:
  - *Hard-clustering* (or *Partitioning*) – One point can only be in one cluster.
  - *Soft-clustering* (*or Fuzzy-clustering*) – A point can have a degree of membership in multiple clusters.

K-means

- **K-means** is a clustering algorithm which aims to *partition* the data points into a *fixed* number of clusters *k*.
- It is a *centroid-based* method, which means that each cluster is represented by a *prototype point* (called a **centroid**) and every point is assigned to the cluster with the *closest* centroid.
- The goal is to find a clustering which minimizes the **within-cluster sum of squares** (i.e. *variance* of clusters).
- K-means uses an iterative method and it converges to a *local optimum*.
    - Finding the global optimum is *NP-hard*

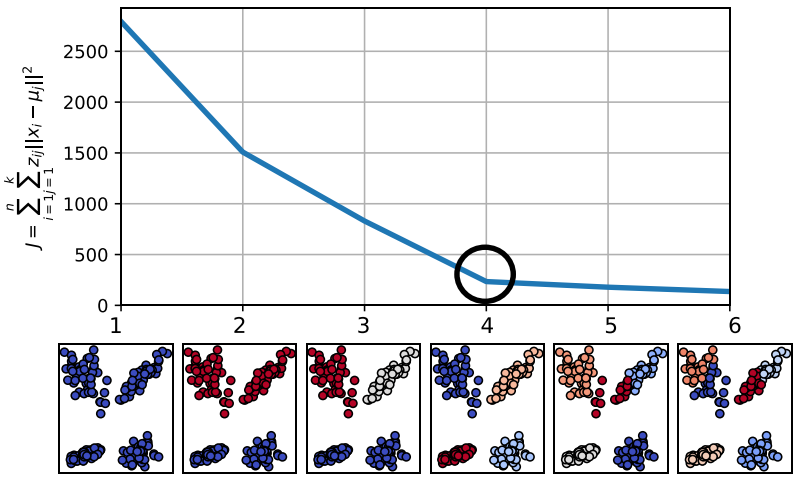
K-means Algorithm

```
initialization:
    - select k random cluster centers
repeat:
    “expectation” step:
        - assign each point to the cluster of the nearest center
    “maximization” step:
        - move the cluster centers to the mean point of each cluster
```

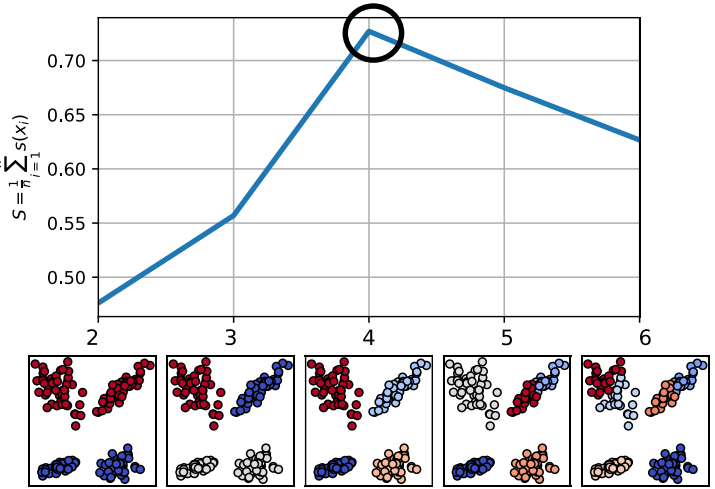
K-means is an instance of the *Expectation-Maximization (EM)* algorithm

How to choose k?

Elbow Method

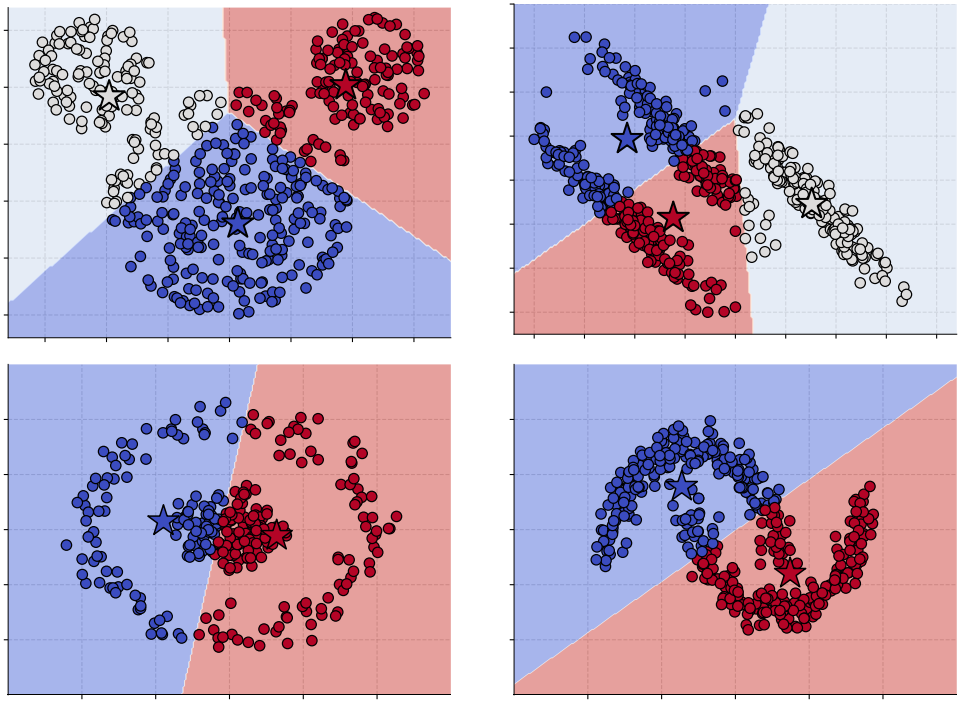


Silhouette Coefficient



K-means Limitations

- How will K-means handle these datasets?
- Not so good…
  - K-means only produces *convex clusters*.
  - It doesn't handle *non-spherical clusters* very well.
  - It tends to produce *clusters of equals sizes*.



DBSCAN

DBSCAN

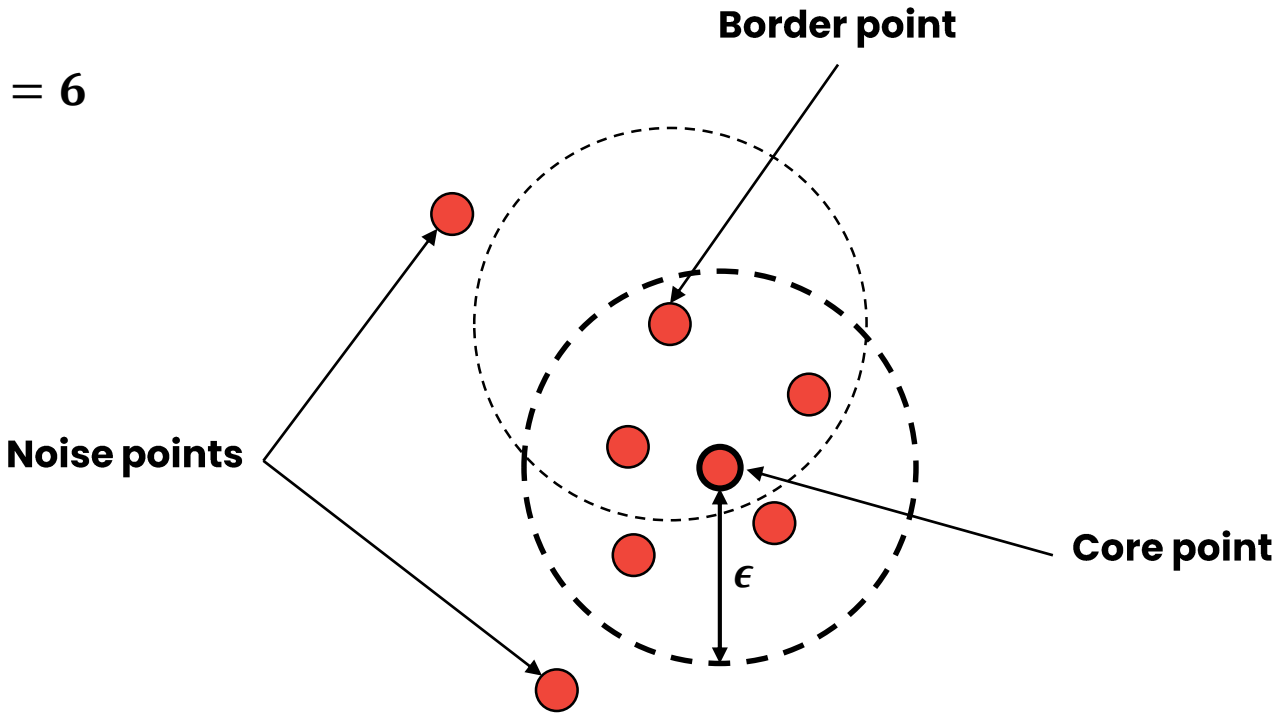
- **DBSCAN** is a *density-based* clustering algorithm which groups points that are *closely packed* together in feature space.
  - **“A density-based algorithm for discovering clusters in large spatial databases with noise”**

    Martin Ester , Hans-Peter Kriegel , Jörg Sander , Xiaowei Xu, 1996

- Unlike *K-means*, it does not require the *number of clusters* to be known in advance, but it has other *hyperparameters* which define the density of the clusters it looks for.

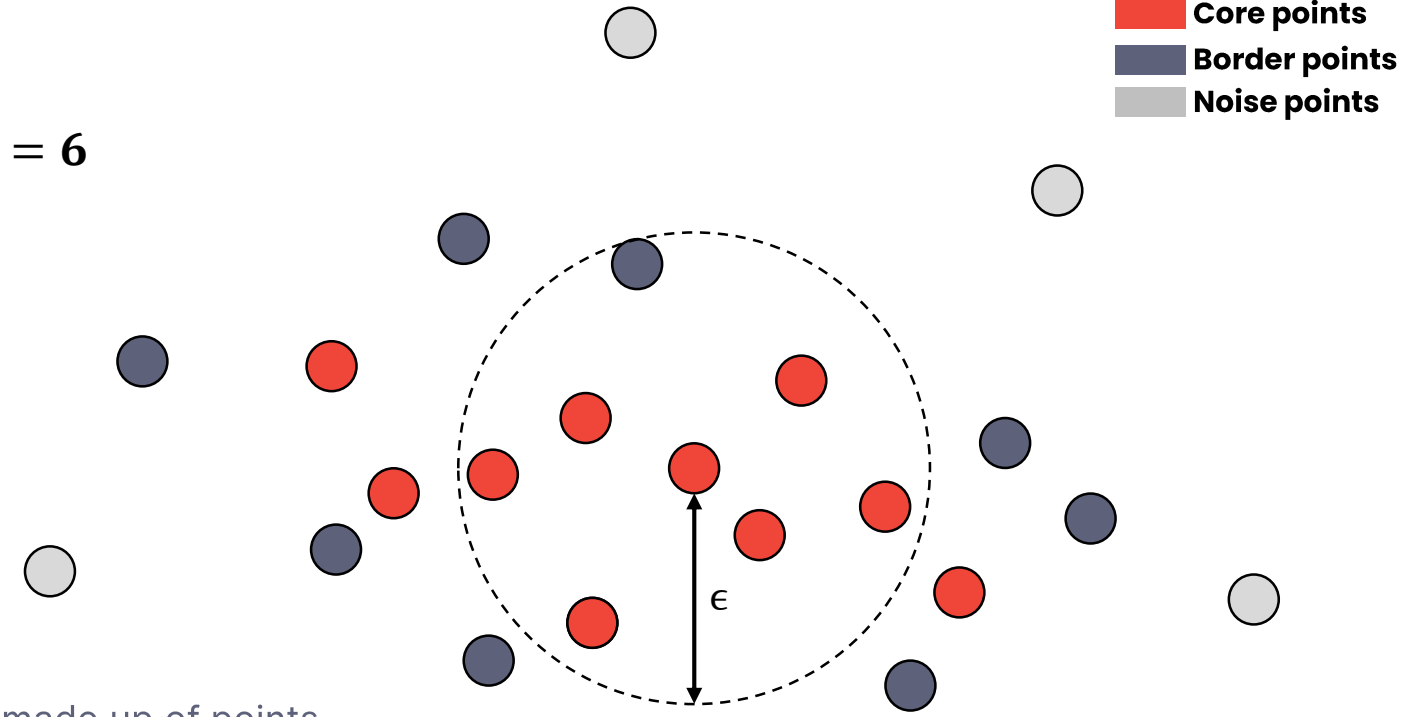
Terminology

$MinPts = 6$



Terminology

$MinPts = 6$



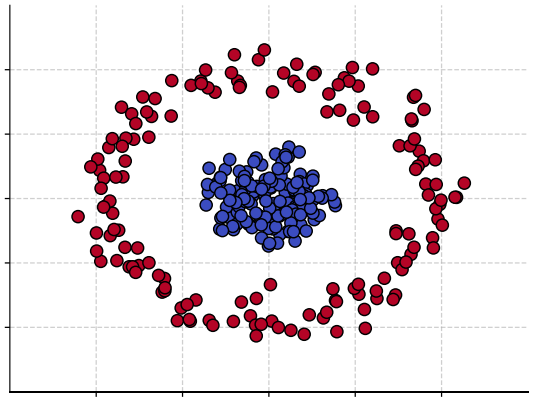
A **cluster** is made up of points which are density-reachable from a core point.

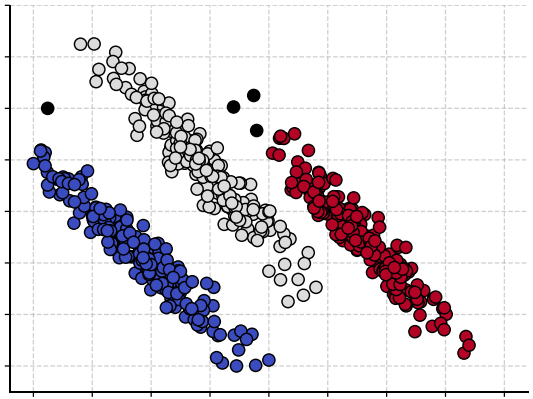
DBSCAN Algorithm

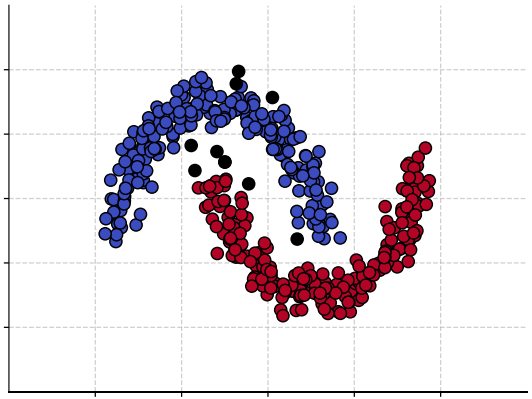
```
for every point p ∈ X:
    if p is a core point:
        label p with a unique cluster id
        for every point q ∈ X which is density-reachable from p:
            label q with the same cluster id as p
    else if p has no label:
        label p as “noise” # might get relabeled later
```

DBSCAN Results

- DBSCAN can handle *non-convex* cluster of various shapes.
- It doesn't require the *number of clusters* to be known in advance.
- The *distance metric* can also be considered a *hyperparameter*.
    - Euclidean distance is the most common.







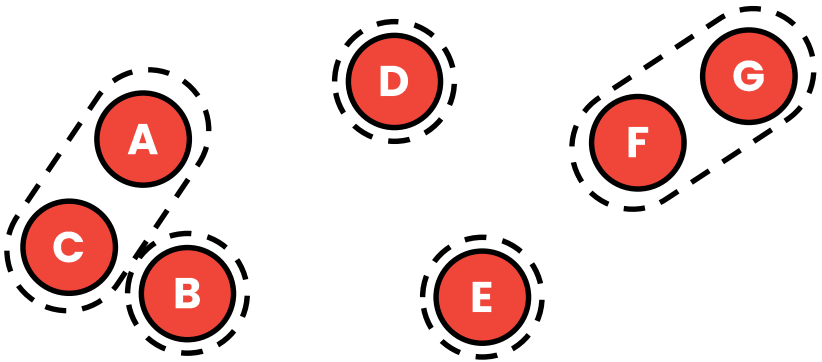
Hierarchical Clustering

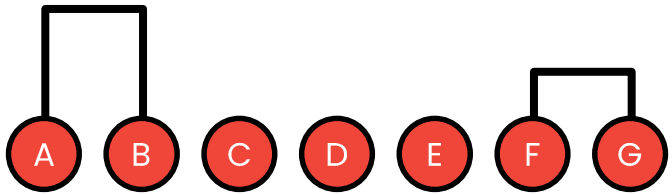
Hierarchical Clustering

- **Hierarchical Clustering** is a clustering method which seeks to build a hierarchy of clusters.
  - i.e. Each cluster is made up of smaller clusters.
- There are two strategies:
  - **Agglomerative** (or *"bottom-up"*) – each point starts in its own cluster and pairs of clusters are merged until only one cluster remains.
  - **Divisive** (or *"top-down")* – There is a single cluster for a the whole dataset and it is recursively split until each point is in its own cluster. (very rarely used in practice).

Agglomerative Clustering

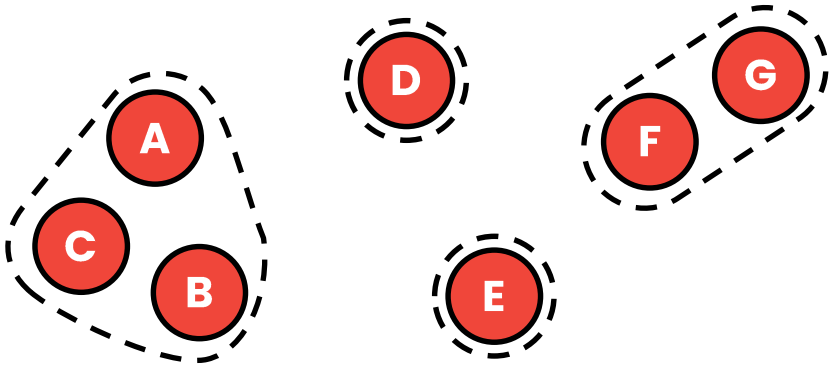
- Each point starts as its own cluster.
- **At every step, the two most similar clusters are merged.**

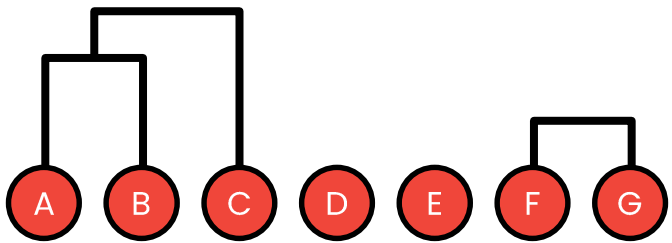




Agglomerative Clustering

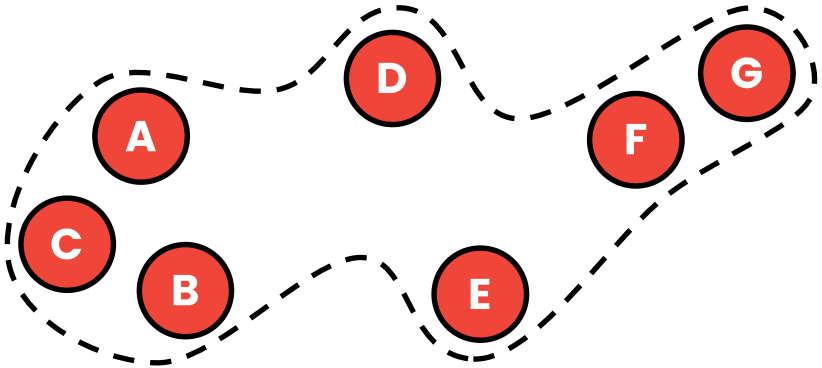
- Each point starts as its own cluster.
- **At every step, the two most similar clusters are merged.**

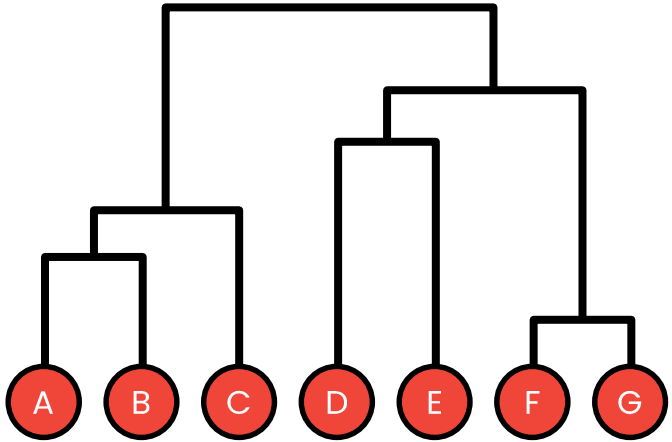




Agglomerative Clustering

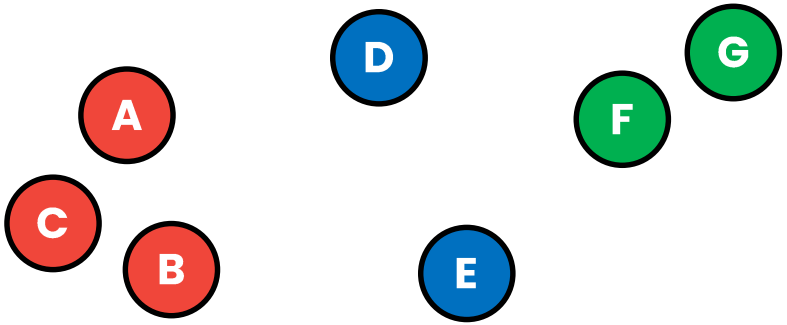
- Each point starts as its own cluster.
- **At every step, the two most similar clusters are merged.**



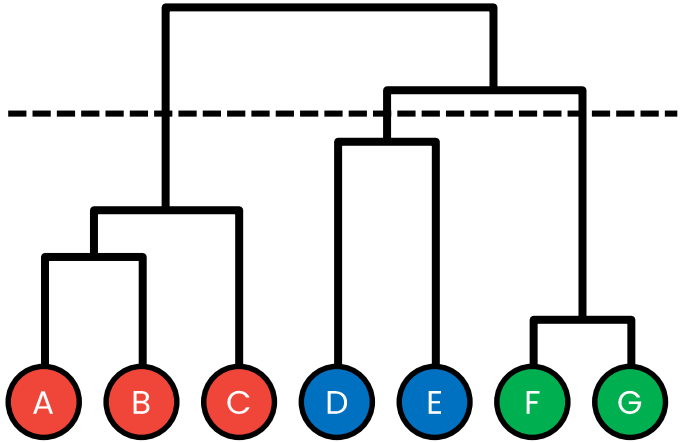


The diagram which shows the tree of clusters is called a **dendrogram.**

Agglomerative Clustering



- Each point starts as its own cluster.
- At every step, the two most similar clusters are merged.
- **We can cut at any level to get a clustering.**

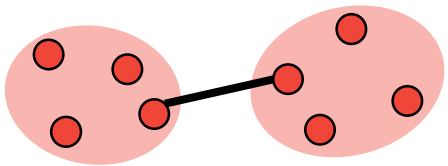


The diagram which shows the tree of clusters is called a **dendrogram.**

Cluster distance

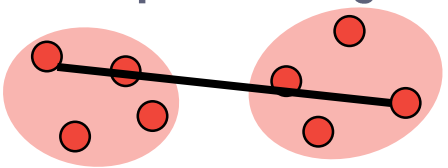
- We need to define a way to measure distance between clusters (called **linkage criterion**).

Single-linkage



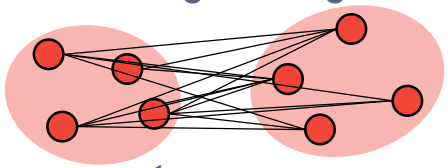
$$\text{dist}(C_1, C_2) = \min_{\vec{a} \in C_1, \vec{b} \in C_2} \left[\text{dist}(\vec{a}, \vec{b})\right]$$

Complete-linkage



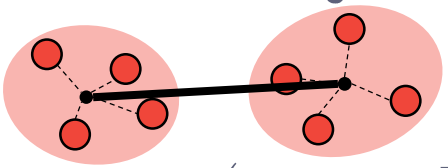
$$\text{dist}(C_1, C_2) = \max_{\vec{a} \in C_1, \vec{b} \in C_2} \left[\text{dist}(\vec{a}, \vec{b})\right]$$

Average-linkage



$$\text{dist}(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{\vec{a} \in C_1} \sum_{\vec{b} \in C_2} \text{dist}(\vec{a}, \vec{b})$$

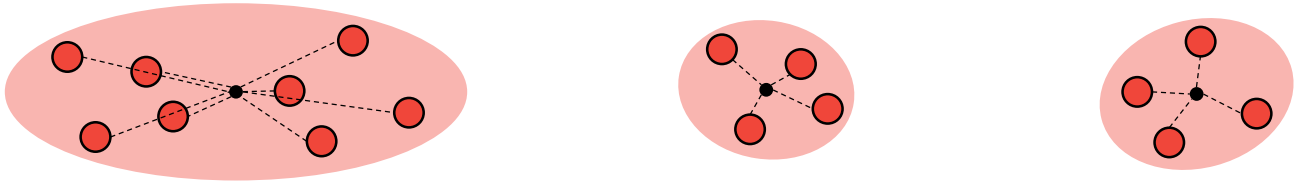
Centroid-linkage



$$\text{dist}(C_1, C_2) = \text{dist}\left(\frac{\sum_{\vec{a} \in C_1} \vec{a}}{|C_1|}, \frac{\sum_{\vec{b} \in C_2} \vec{b}}{|C_2|}\right)$$

Cluster distance

- **Ward's criterion** defines the distance between clusters as the *increase in variance* due to merging the two clusters.



$$\text{dist}(C_1, C_2) = \frac{1}{|C_1 \cup C_2|}\sum_{\vec{x}\in C_1\cup C_2}\text{dist}(\vec{x}, \vec{\mu}_{C_1\cup C_2}) - \frac{1}{|C_1|}\sum_{\vec{a}\in C_1}\text{dist}(\vec{a}, \vec{\mu}_{C_1}) - \frac{1}{|C_2|}\sum_{\vec{b}\in C_2}\text{dist}(\vec{b}, \vec{\mu}_{C_2})$$

$$= \quad Var(C_1 \cup C_2) \quad - \quad Var(C_1) \quad - \quad Var(C_2)$$

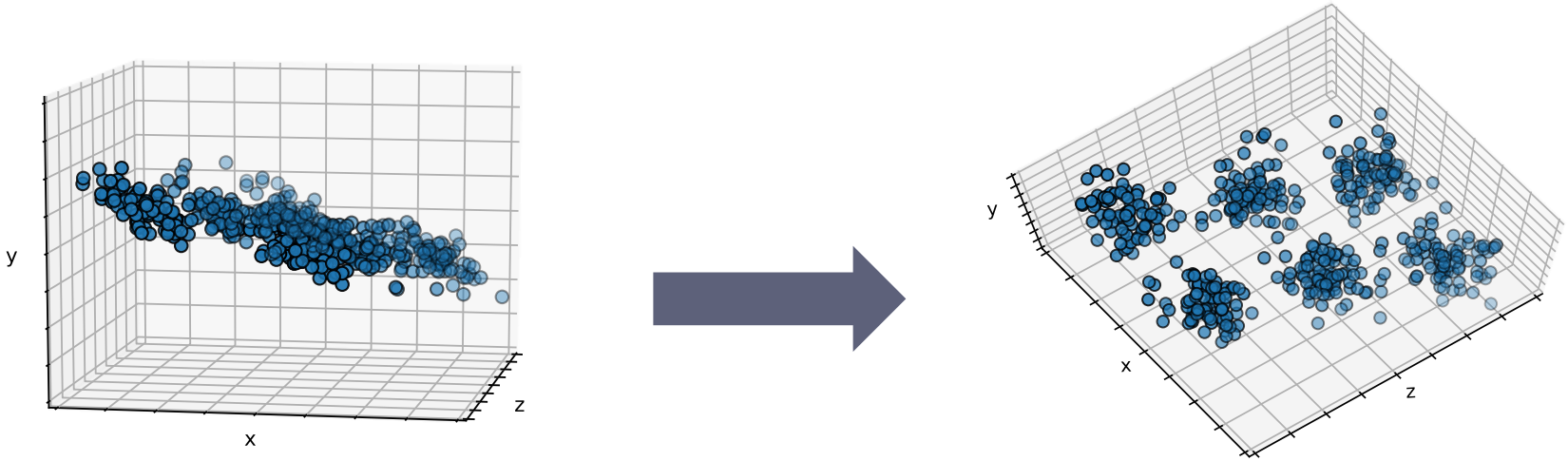
Principal Component Analysis

Principal Component Analysis

- **Principal Component Analysis** (**PCA**) transforms a dataset into a new *orthogonal coordinate system* in which the data is *centered* and the features are completely *uncorrelated*.
  - The *mean* of the new dataset is 0.
  - The *covariance* of any pair of distinct features is 0.
- The features of the transformed dataset (called **principal components**) are sorted by *variance*, such that the first feature has the greatest variance.
- Component with low variance can be discarded, making *PCA* a method of **dimensionality reduction**.

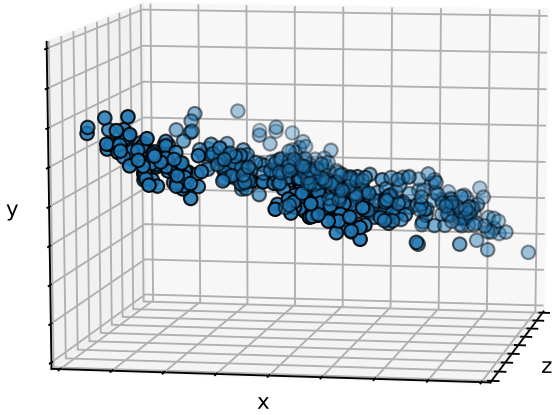
PCA

- We can rotate the axes until we find the *best angle* to look at the data from
- Rotation of the axes is equivalent to changing the *coordinate system* (or rotating the data).

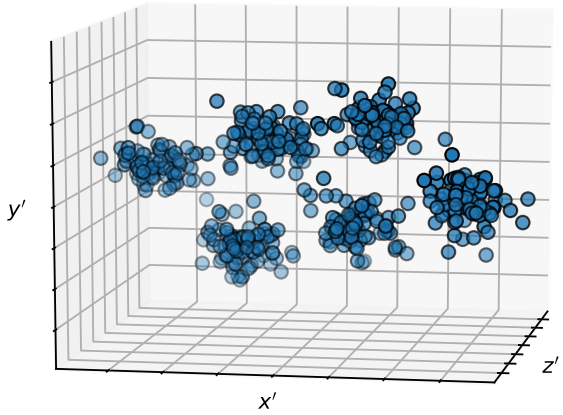


PCA

- We can rotate the axes until we find the *best angle* to look at the data from
- Rotation of the axes is equivalent to changing the *coordinate system* (or rotating the data).

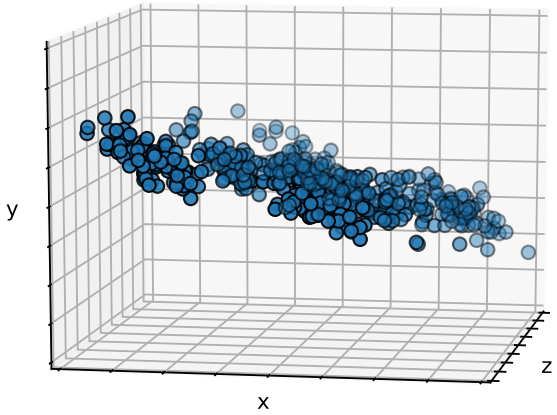


$$x' = 0.56x - 0.25y - 0.79z$$
$$y' = -0.75x + 0.25y - 0.61z$$
$$z' = 0.35x + 0.94y - 0.04z$$

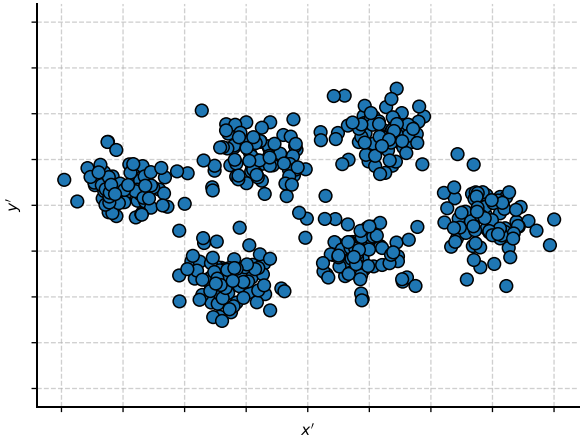


PCA

- We can rotate the axes until we find the *best angle* to look at the data from
- Rotation of the axes is equivalent to changing the *coordinate system* (or rotating the data).



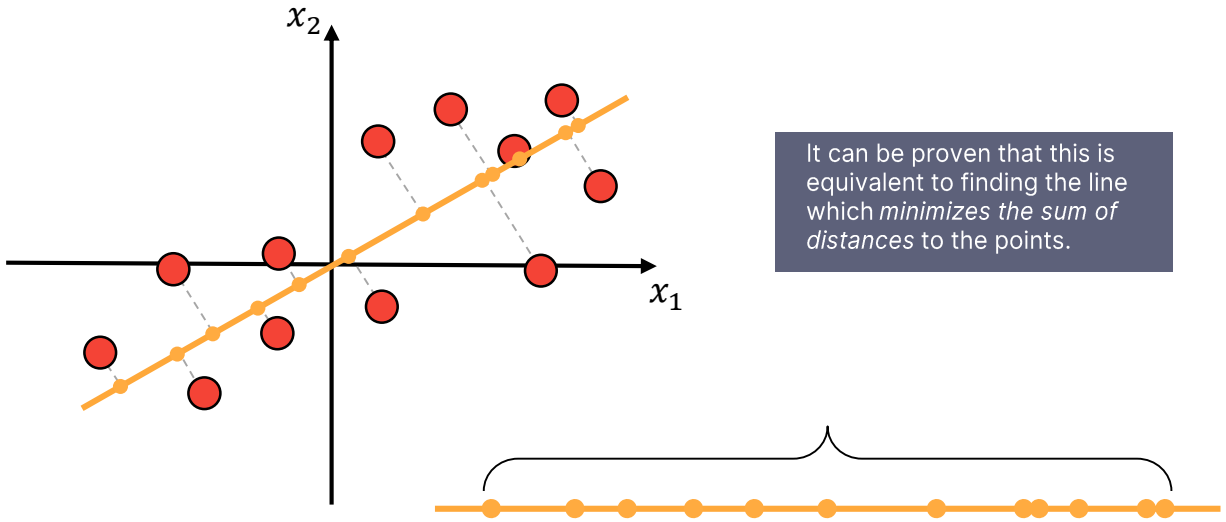
$$x' = 0.56x - 0.25y - 0.79z$$
$$y' = -0.75x + 0.25y - 0.61z$$
$$z' = 0.35x + 0.94y - 0.04z$$



- The axes which gives the *best view* of the data are called **principal components**.
- We also compute *how much variance is explained* by each component
  - x' explains ~76%, y' explains ~26% and z' explains < 1%
  - We can discard components which explain too little variance.

Finding the first PC

- The first step is *centering* the data (subtracting the *mean* from all data points).
- The first *principal component* is the direction on which the data has the *largest variance*.
  - We are looking for a line on which the projection of data points are as spread out as possible.

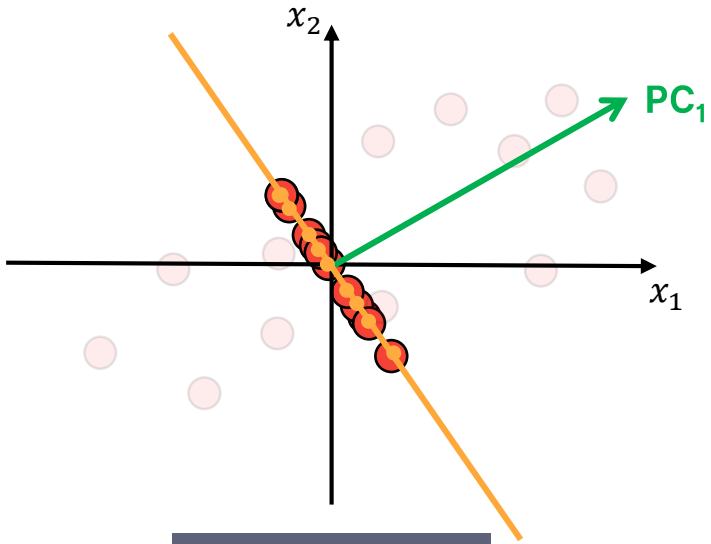


It can be proven that this is equivalent to finding the line which *minimizes the sum of distances* to the points.

Finding further PCs

- If we subtract the projection of points on the first PC from the points themselves, we get a dataset which has 0 variance on that direction.
- By applying the same method on the new dataset, we get the *second principal component* of the original dataset.

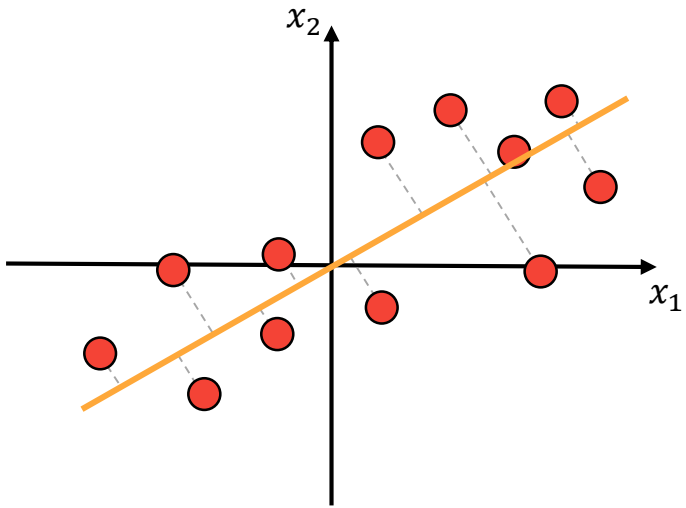
The second PC is the direction in which the data varies the most, after eliminating the variance on the first PC.



Trivial in 2 dimensions.

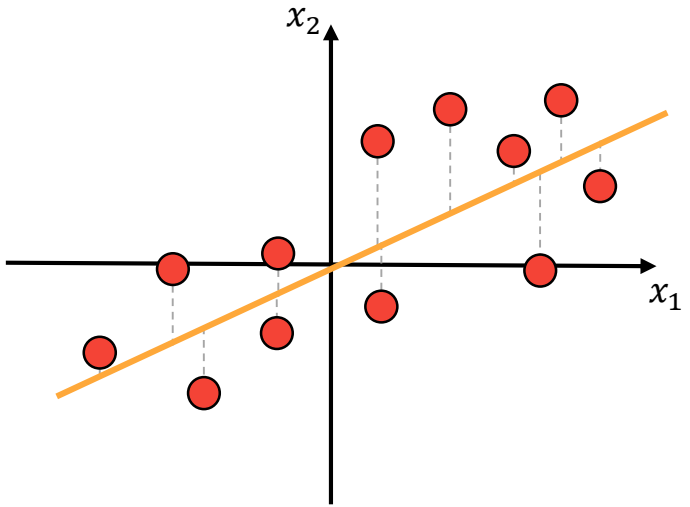
PCA is not Linear Regression!

PCA finds the line which minimizes the *sum of distances to the data points.*



x_1 and x_2 are both features
(a.k.a. independent variable)

Linear Regression finds the line which minimizes the *sum of squared distances to the predictions* given by the line.



x_2 is a label
(a.k.a. dependent variable)

Random Variable

- A **random variable** is a variable whose possible values are outcomes of a *random phenomenon* (e.g. rolling a die).
  - The source of uncertainty in a random variable can either be "objective" (the result of a *random process*) or "subjective" (the result of *incomplete knowledge*).
  - A random variable has a corresponding *probability distribution* which specifies the probability that its value falls in any given interval.
- A **random variate** is a particular outcome of a *random variable*
  - The result of sampling its probability distribution.
- The **expected value** E (or **mean** μ) of a *random variable* is the long-run average of its *random variates*.
  - For the discrete case, it is the *probability-weighted average* of all possible outcomes.

$$E[\text{🎲}] = 3.5$$

- The **variance** σ^2 is a measure of the dispersion of *random variates* around the *expected value*.
  - σ is called the **standard deviation**.

Random Variable

- $\mathrm{X} = \{x_1, x_2, \ldots, x_n\},\ Y = \{y_1, y_2, \ldots, y_n\}$ are two sets of random variates of two random variables which are sampled together (x_i at the same time with y_i).
- X and Y are just subsets of the whole (potentially infinite) population.
  - Then the **sample mean** and **sample variance** are just estimates of the true mean and variance values.

$$Mean(X) = E[X] = \mu_X = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$Var(X) = \sigma_X^2 = \frac{1}{n-1}\sum_{i=1}^{n} (x_i - \mu_X)^2$$

$n-1$ instead of n is **Bessel's correction** which accounts for biased estimation of the mean.

- **Covariance** measures the joint variability of two random variables (e.g. if larger values of one corresponds to larger values of the other, or vice-versa).

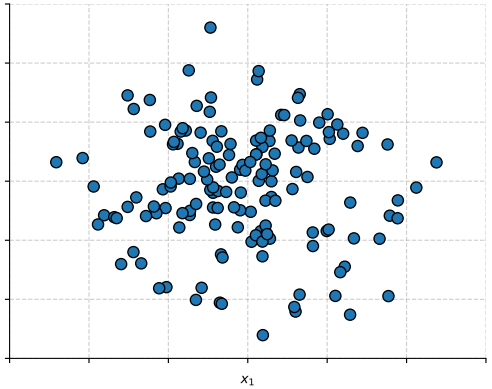
$$Cov(X, Y) = \sigma_{XY}^2 = \frac{1}{n-1}\sum_{i=1}^{n} (x_i - \mu_X)(y_i - \mu_Y)$$

Features as Random Variables

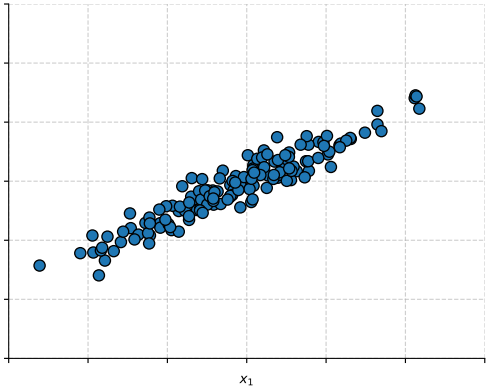
- Dataset $X \in \mathbb{R}^{m \times n} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$ (m examples with n features).
  - Each row represents all features of an examples.
  - Each column represent the same feature of all examples.
- We can regard each feature (each column) as a *random variable*.
  - This makes the value $x_j^{(i)}$ (feature j of example i) a *random variate*.
  - Example $\vec{x}^{(i)}$ is made up of n *random variables* sampled together.
- This makes dataset X a set of m sampling events of n random variables.
  - $\Longrightarrow$ We can compute *mean*, *variance*, *covariance*.

Covariance of a dataset

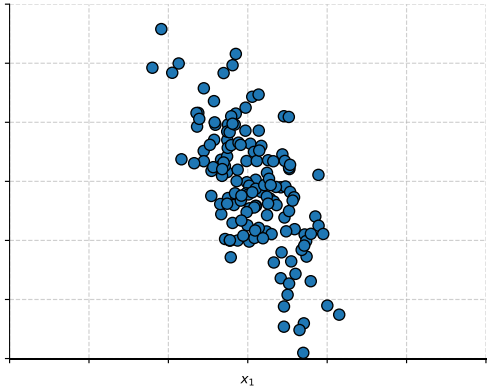
$$\Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$



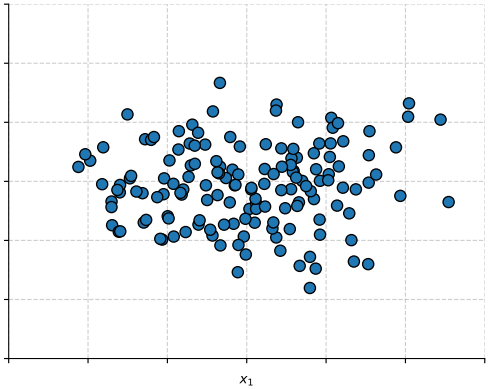
$$\Sigma = \begin{pmatrix} 1 & 0.6 \\ 0.6 & 0.4 \end{pmatrix}$$



$$\Sigma = \begin{pmatrix} 0.2 & -0.3 \\ -0.3 & 1 \end{pmatrix}$$



$$\Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 0.5 \end{pmatrix}$$



Eigenvectors and eigenvalues

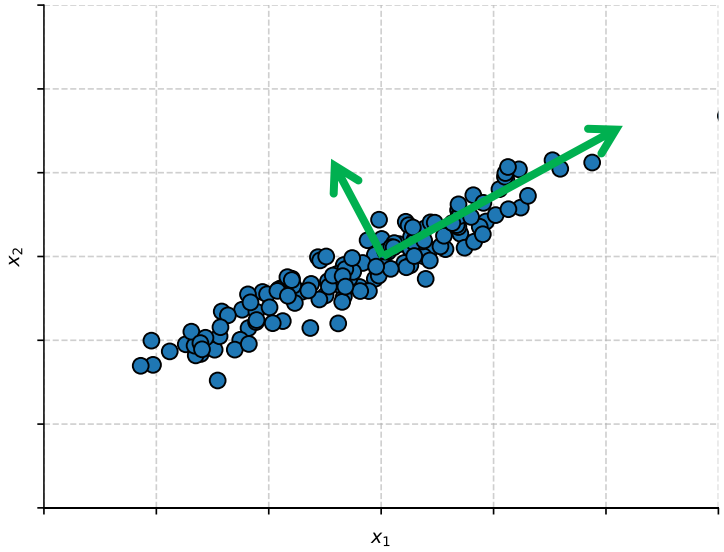
- An **eigenvector** $v \in \mathbb{R}^n$ and its corresponding **eigenvalue** $\lambda \in \mathbb{R}$ of a square matrix $A \in \mathbb{R}^{n \times n}$ are the solutions to the following equation:

$$Av = \lambda v$$

- In other words, they are the vectors which *only get scaled* when multiplied by A, but *don't change direction*. The corresponding eigenvalue λ is the scaling factor.

PCA as Eigen Decomposition

- $X \in \mathbb{R}^{m \times n}$ with covariance Σ_X.
- How do the eigenvectors of Σ_X look like?
  - Intuitively, consider Σ_X "responsible" for X's shape.
  - Remember than the *eigenvectors* are the vectors which do not change direction when multiplied by Σ_X.
- The **eigenvectors** of the *covariance matrix* are, in fact, the **principal components** of matrix X.
  - The corresponding **eigenvalues** are equal to the *amount of explained variance* of each component.



- Eigen Decomposition:

$$\Sigma_X = V \Lambda V^{-1}$$

where

$$V \in \mathbb{R}^{n \times n} = (v_1 \quad \cdots \quad v_n), \quad v_i \in \mathbb{R}^{n \times 1} \text{ - column eigenvectors}$$

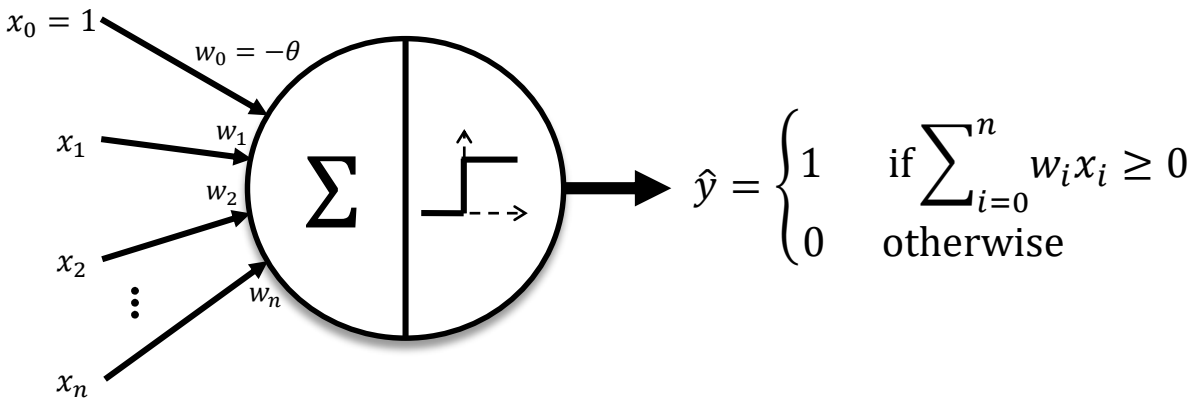
$$\Lambda \in \mathbb{R}^{n \times n} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix} \text{ - diagonal eigenvalues}$$

The Perceptron

The Perceptron

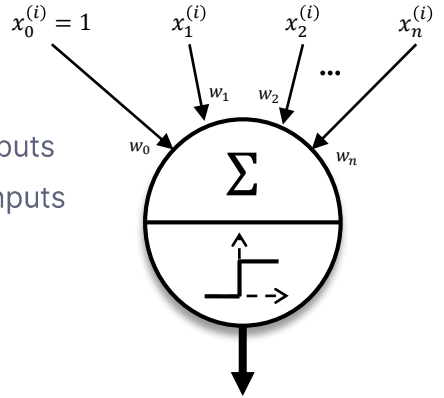
- Based on the *MuCulloch-Pitts* model and with Hebb's ideas in mind, **Frank Rosenblatt** invented, in 1957, a machine and an associated learning algorithm, which he called **"The Perceptron",** designed for image recognition.
  - It was intended to be a physical machine (with photocells as inputs, potentiometers as weights and electric motors which performed weight updates), although the first implementation was in software.

$$x_0 = 1, \qquad x_1, x_2, \dots, x_n \in \{0,1\}, \qquad w_0, w_1, w_2, \dots, w_n \in \mathbb{R}, \qquad \hat{y} \in \{0,1\}$$



Perceptron Learning Algorithm

- Training Set: $E = \{(\vec{x}^{(1)}, y^{(1)}), \ldots, (\vec{x}^{(m)}, y^{(m)})\}$, $\vec{x}^{(i)} \in \{0,1\}^{n+1} \left(x_0^{(i)} = 1, \forall i\right)$, $y^{(i)} \in \{0,1\}$
- Weights w_j start off as 0.
- For every example $(\vec{x}^{(i)}, y^{(i)})$ in the dataset:
  - If $\hat{y}^{(i)} == y^{(i)}$ we don't change anything
  - If $\hat{y}^{(i)} == 0$ and $y^{(i)} == 1$ we need to increase the strength of all active inputs
  - If $\hat{y}^{(i)} == 1$ and $y^{(i)} == 0$ we need to decrease the strength of all active inputs
- Perceptron update rule:



$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j^{(i)} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right) \cdot x_j^{(i)}$$

Amount by which the strength is changed ("**learning rate**").

Sets the update direction.

Selects active inputs.

Perceptron Limitations

- It turns out that the *Perceptron* had no trouble learning AND/OR and recognizing digits because they are not very hard problems.
- Let's consider the **XOR** (exclusive OR) function:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

For a Perceptron to learn XOR, it needs:

$$w_0 + 0 \cdot w_1 + 0 \cdot w_2 < 0 \Rightarrow \boldsymbol{w_0 < 0}$$

$$\left.\begin{array}{l} w_0 + 0 \cdot w_1 + 1 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_2 \\ w_0 + 1 \cdot w_1 + 0 \cdot w_2 \geq 0 \Rightarrow w_0 \geq -w_1 \end{array}\right\} 2w_0 \geq -w_1 - w_2$$

$$w_0 + 1 \cdot w_1 + 1 \cdot w_2 < 0 \Rightarrow w_0 < -w_1 - w_2$$

Contradiction

$$2w_0 > w_0 \Rightarrow \mathbf{w_0 > 0}$$

$\Rightarrow$ The Perceptron cannot possibly learn this function, no matter how many training steps it takes.

- In fact, the *Perceptron* can only learn the class of problems known as **linearly separable**.

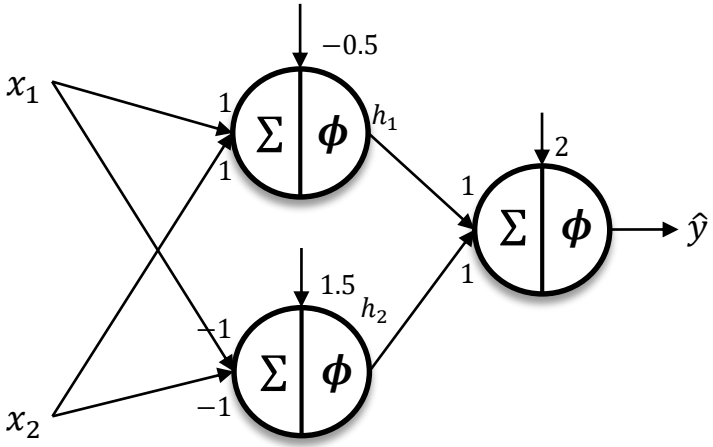
Multilayer Perceptron

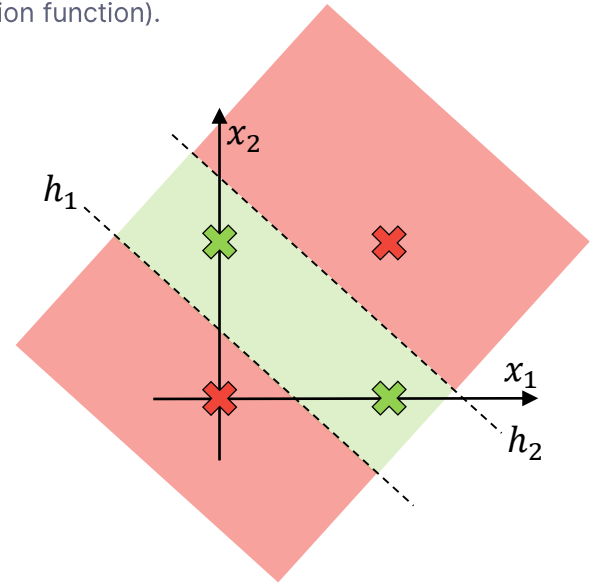
XOR with Perceptrons

- A single Perceptron cannot learn the XOR function because it is not linearly separable.
- This limitation can be overcome by “combining” the outputs of multiple Perceptrons.
  - We can combine them by using another Perceptron (weighed sum + activation function).

XOR

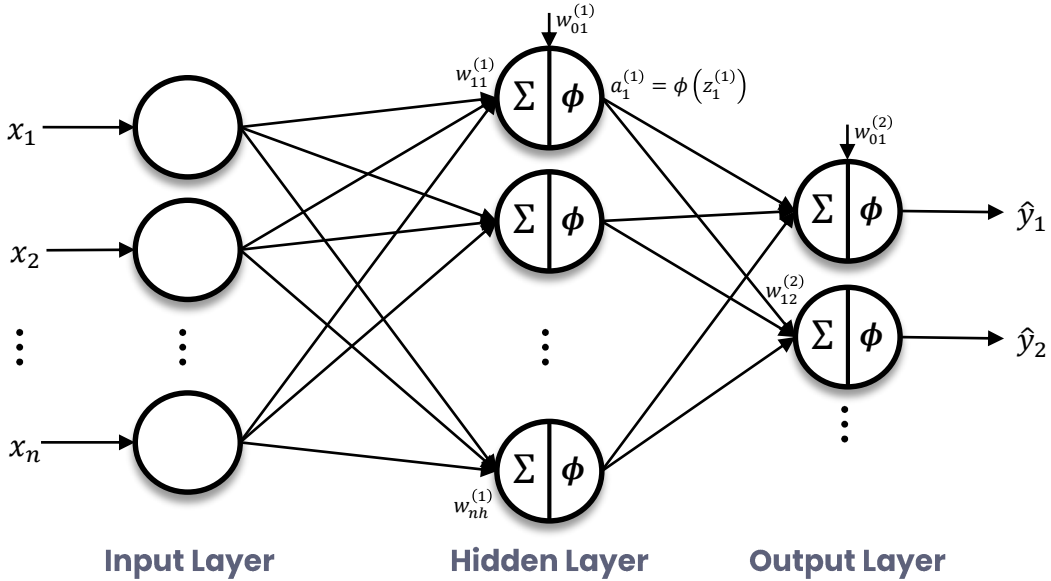
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0





The Multilayer Perceptron

- A **Multilayer Perceptron (MLP)** is a *feedforward artificial neural network* which has an *input layer*, one or more *hidden layers* and an *output layer*.
  - All neurons, excepts those from the input layer, apply an activation function to the weighted sum of inputs.
  - Each pair of neurons from consecutive layers has an associated weight.



- $w_{ij}^{(l)}$ is the weight from neuron i on layer $l-1$ to neuron j on layer l (input is layer 0).
- $w_{0j}^{(l)}$ is the bias of neuron j on layer l.
- $a_i^{(l)}$ is the output of neuron i on layer l.
- $z_i^{(l)}$ is the output before activation.

Training an MLP

- A multilayer perceptron is trained with **stochastic gradient descent**.
  - "*Stochastic*" because the gradient is computed only with respect to a single training example or a batch, not the entire dataset.
  - We need to compute the gradient of the error function with respect to each weight of the network and update the weights correspondingly.

$$E(\vec{x}) = \mathcal{L}(y, \hat{y})$$

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial E}{\partial w_{ij}^{(l)}}$$

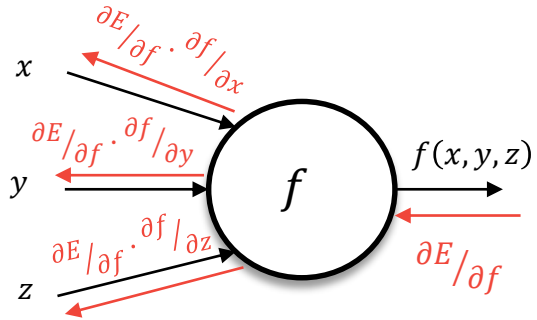
$\mathcal{L}$ is the loss (a function which should be small if $\hat{y}$ is close to y and larger otherwise).

  - Or in matrix format:

$$\Delta W^{(l)} = -\eta \frac{\partial E}{\partial W^{(l)}}$$

Backpropagation of error

- Computing the derivative *formula* (symbolic differentiation) for every weight in the network is both inefficient and very complex for larger networks.
- We are only interested in the numerical evaluation of the derivatives, we can focus on one gate at a time and we can used previously computed values.



$\partial E/\partial f$ is already computed numerically.

- This method is known as "**backpropagation**".
  - **„Learning representations by back-propagating errors"**

David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, 1986

Choosing an activation function

- In order to do backpropagation we need to compute the *derivative of the activation function*.
  - The *unit step function* of the perceptron is not differentiable.
  - The *linear activation* used for *Adaline* does not benefit from multiple layers:

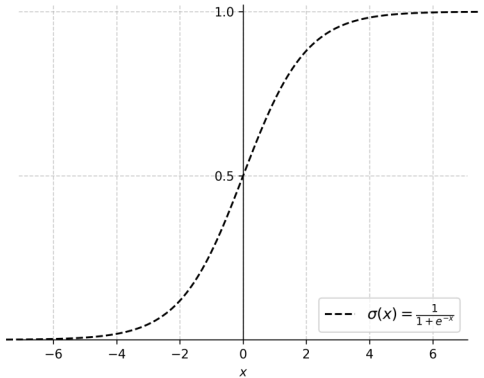
$$\hat{y} = \phi_{\text{linear}}\left(W^{(2)}\phi_{\text{linear}}\left(W^{(1)}x\right)\right) = W^{(2)}W^{(1)}x = W'x$$

- We are looking for a step-like differentiable function.
  - **Standard Logistic Function**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \sigma(x)\left(1-\sigma(x)\right)$$

Usually referred to as "the" **sigmoid**.



Choosing a loss function

- Adaline used a **least squares loss** function:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \qquad \mathcal{L}'(y, \hat{y}) = -(y - \hat{y})$$

- The **Cross-entropy loss** is much more common in practice.

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \qquad \mathcal{L}'(y, \hat{y}) = -\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}$$

- Typically, the output layer uses **Softmax** activation, instead of a *sigmoid* for each neuron, to make it suitable for probabilistic interpretation.

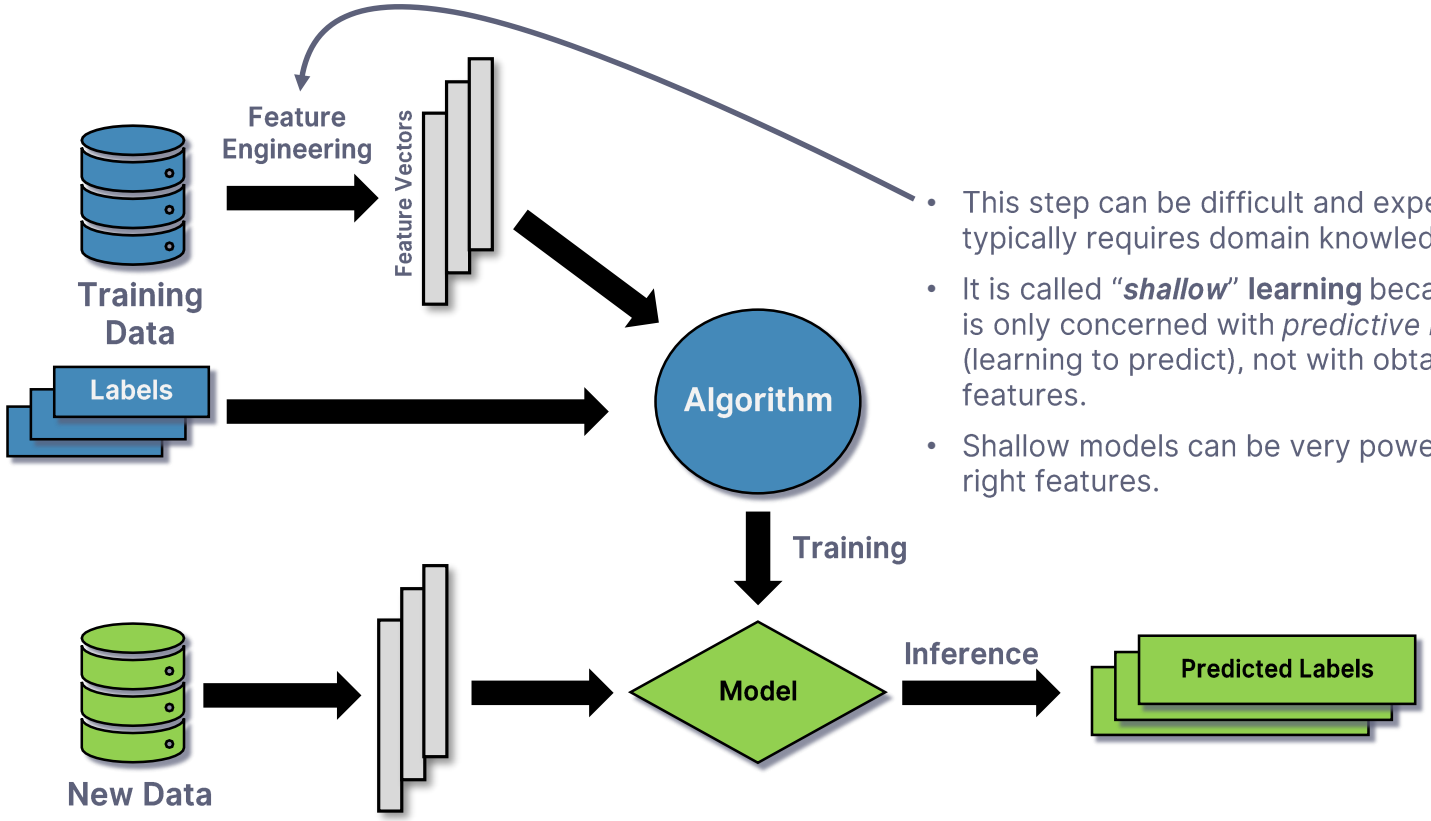
$$\hat{y}_j = \frac{e^{z_j^{(2)}}}{\sum_{k=1}^{o} e^{z_k^{(2)}}} \qquad \text{instead of} \qquad \hat{y}_j = a_j^{(2)} = \sigma\left(z_j^{(2)}\right)$$

Universal approximation theorem

- The **universal approximation theorem** states that a multilayer perceptron with a single hidden layer containing a finite number of neurons is sufficient to represent **any function** given appropriate parameters.
  - However, it does not say anything about the learnability of those parameters.
- This implies that the model which the MLP learns can be *arbitrarily complex*, which can rapidly lead to **overfitting**.

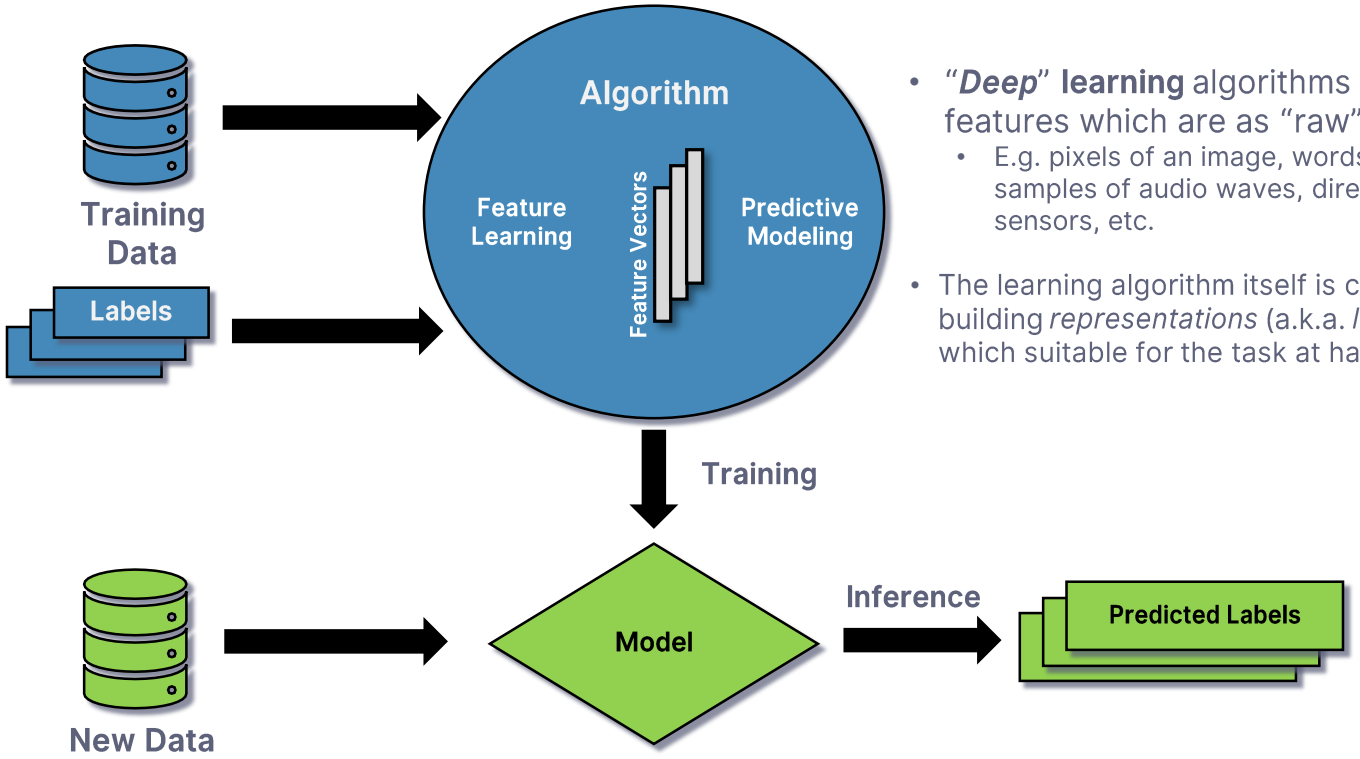
Deep Learning

Typical "Shallow" Learning Flow



- This step can be difficult and expensive and typically requires domain knowledge.
- It is called "***shallow***" **learning** because the algorithm is only concerned with *predictive modeling* (learning to predict), not with obtaining good features.
- Shallow models can be very powerful, given the right features.

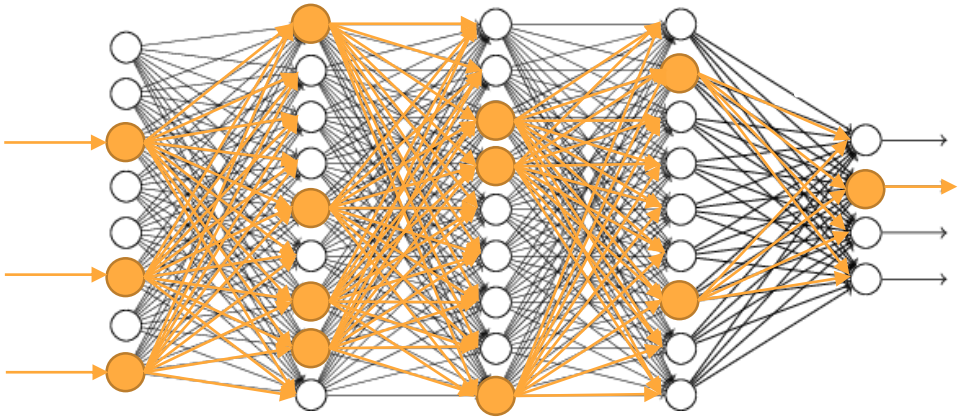
Typical "Deep" Learning Flow



- "***Deep***" **learning** algorithms usually receive features which are as "raw" as possible:
  - E.g. pixels of an image, words or characters of text, samples of audio waves, direct measurements of sensors, etc.
- The learning algorithm itself is concerned with building *representations* (a.k.a. *learning features*) which suitable for the task at hand.

Deep Neural Networks

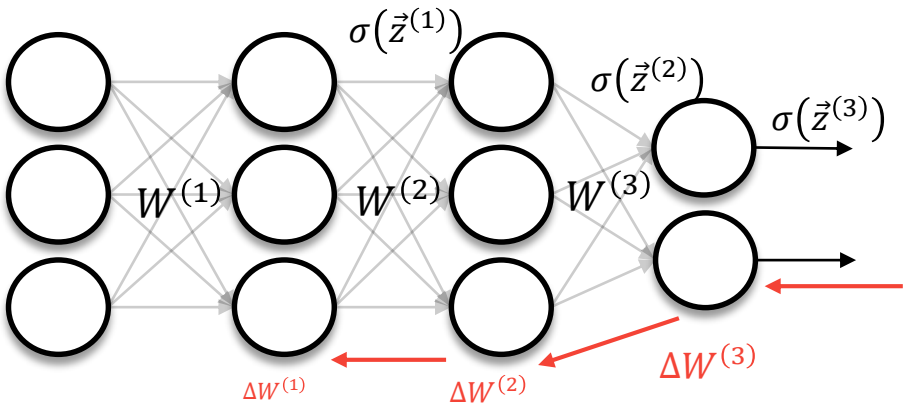
- A neural network is considered ”**deep**” if it has more than one hidden layer.
  - An input causes an *neuron activation pattern* in the first hidden layer.
  - The first layer causes another *activation pattern* in the second layer and so on.
  - The last layer makes a *prediction*, based the on *activation pattern* of the layer behind it
- In other words, with each layer, the network **gradually builds more and more abstract representations** of the input and the final layer acts like a **predictive model**.
- The *layered structure* of a **deep neural network** makes it the perfect candidate for a **deep learning algorithm**.

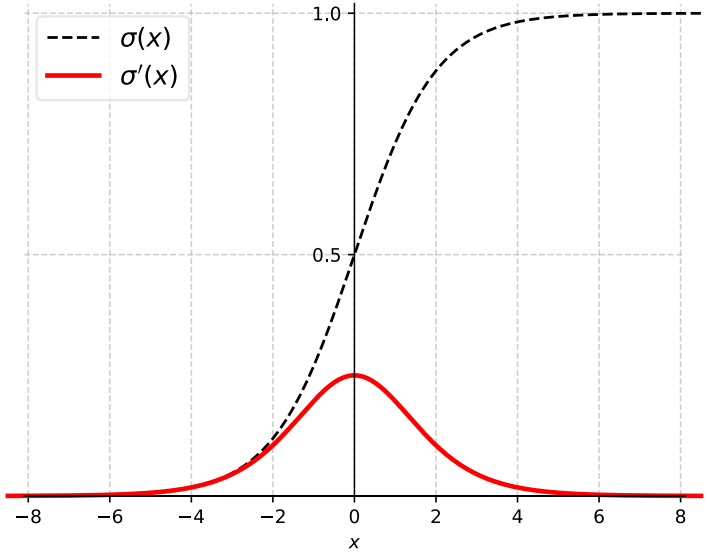


What has changed?

- In the late 2000s and early 2010s, *Deep Neural Networks* and *Deep Learning* have seen a major increase in popularity and performance.
- Aside from the major increase in **data availability** and **computing power**, an important reason for the recent breakthrough in *Deep Learning* are the advances made by researchers on the **algorithmic** side.
  - Better **activation** functions (e.g. ReLU)
  - Better **optimization** algorithms (e.g. Momentum, RMSProp, Adam)
  - Better network layer **structure** (e.g. Convolutions, LSTMs, Residual Networks).
  - Better **regularization** techniques (e.g. Dropout, BatchNorm).
  - Deep Learning **frameworks** (e.g. TensorFlow, Theano, PyTorch, Caffe).

The Trouble with Sigmoids





$$\Delta W^{(3)} = (\ldots) \cdot \sigma'(\vec{z}^{(3)})$$

$$\Delta W^{(2)} = (\ldots) \cdot \sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})$$

$$\Delta W^{(1)} = (\ldots) \cdot \underbrace{\sigma'(\vec{z}^{(1)})\sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})}_{\text{Very small number.}}$$

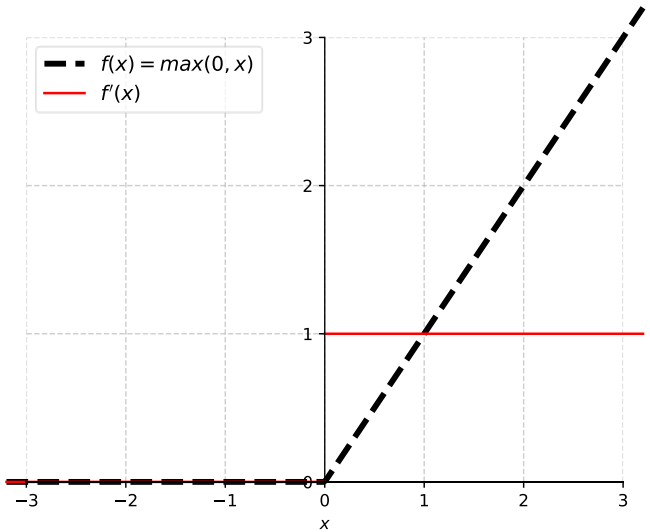
$\sigma'(x)$ has a maximum value of 0.25 for $x = 0$ and it drops very quickly (e.g. $\sigma'(10) \approx 0.00004$).

Rectified Linear Units

- The linear activation $f(x) = x$ does not suffer from *vanishing gradient problem*, but it also does not benefit from multiple layers.
- The **Rectified Linear Unit** (**ReLU**) is a *non-linear activation* function which takes only the positive part of the linear activation.

$$f(x) = \max(0, x) \qquad f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

- *ReLU* neurons cannot *saturate* on the positive side so they don't suffer from the *vanishing gradient* problem
- The *ReLU* activation is much easier to compute.
- *ReLUs* output 0 in the negative range which encourages activation sparsity.



f is not differentiable in 0, but this not a problem in practice

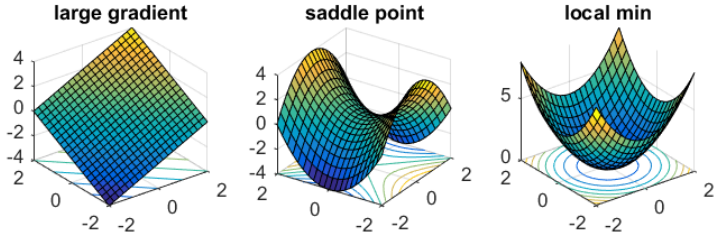
Stochastic, Batch and Mini-Batch

- **"Stochastic" Gradient Descent** is used for the case in which the algorithm computes the gradient and makes *one weight update for one training sample* at a time.
  - The can lead slow convergence because of *zig-zagging* of the weight vector instead of always taking steps towards the minimum.
- Ideally, we want to make a *single weight update for the average gradient of all training samples*.
  - This is known as **Batch Gradient Descent**, or sometimes **"Vanilla" Gradient Descent**.
  - This method computes the *optimal direction for every update*, but every step involves *lots of computations*.
- The most common method is to make *one weight update based on the average gradient for a (small) subset of data*.
  - This is called **Mini-Batch Gradient Descent**.
  - It usually *converges faster* the *Batch*, even if it takes more steps, because every step is *computed much faster*.

Mini-Batch Gradient Descent is almost always used in practice, but sometimes it is referred to as **SGD**.

Problems with Gradient Descent

- Even the optimal, but slow, **Batch Gradient Descent** does not guarantee *good convergence*.
- The *error surface* of a *deep neural network* is **non-convex** which means that GD can easily get stuck in **local minima**.
- However, it turns out that **saddle points**, not local minima, are a worse problem for GD convergence.
  - *Saddle points* are locations where some dimensions slope up and others slope down, which makes the gradient ~0 in all directions.



- Some of these issues can be alleviated by choosing a proper **learning rate**, but this is not trivial.
  - Small *learning rate* means slow convergence, *large learning* rate means fluctuations around the minimum or divergence.
  - Another issue is that the *same learning rate* applies to all weights. We might want more active neurons to have a smaller learning rate than neurons which are rarely active.

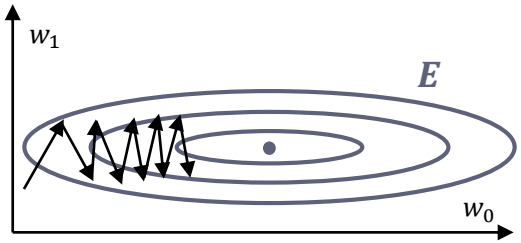
Momentum

- One case in which SGD is slow is when the error surface is more steep in one direction then in another.

$$\Delta w^{t+1} = -\eta \frac{\partial E}{\partial w^t}$$

w^t is the weight at time t.

  - This causes SGD to oscillate from one side to the other, while making little progress towards the minimum.

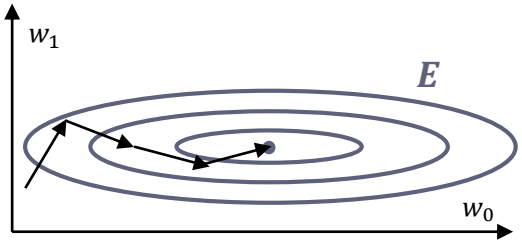


- **Momentum** is a method which accelerates SGD in a certain direction if several consecutive updates push it towards it.
- It works by keeping an *exponential moving average of gradients* (the "**velocity**") and using it to update the weights, instead of using the gradient directly.

$$v^t = \gamma v^{t-1} + (1-\gamma)\frac{\partial E}{\partial w^t} \qquad \Delta w^{t+1} = -\eta \Delta v^t$$

γ is usually ~0.9

  - The velocities in relevant directions grow over time.
  - The velocities in oscillating directions cancel out.



- Momentum can also help to avoid getting stuck in certain local minima.

Adaptive Learning Rate

- **Adagrad**, **Adadelta** and **RMSProp** are optimization techniques which use an **adaptive learning rate** (i.e. the learning rate adapts to each weight at each step).
- **RMSProp** (**R**oot **M**ean **S**quare **Prop**) tries to make learning rate smaller for weights which get updated frequently. It does this by keeping a *moving average of squared gradients* for each weight and scales the corresponding learning rate by the *root* of this value.

$$\text{MeanSquare}(w^t) = \gamma \text{MeanSquare}(w^{t-1}) + (1-\gamma)\left(\frac{\partial E}{\partial w^t}\right)^2$$

Exponential moving average of squared gradients.

$$\Delta w^{t+1} = -\frac{\eta}{\sqrt{\text{MeanSquare}(w^t)}}\frac{\partial E}{\partial w^t}$$

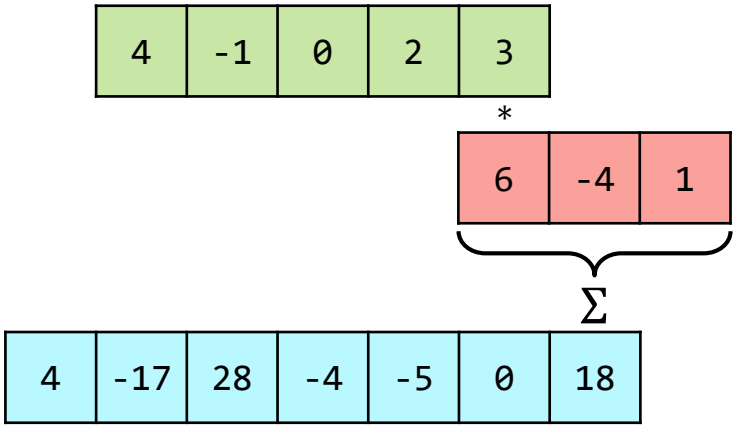
- **ADAM** (**Ada**ptive **M**oment Estimation) is a combination of *Momentum* and *RMSProp* and it is very common in practice.

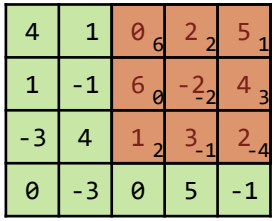
$$v^t = \gamma_1 v^{t-1} + (1-\gamma_1)\frac{\partial E}{\partial w^t} \qquad \text{MeanSquare}(w^t) = \gamma_2 \text{MeanSquare}(w^{t-1}) + (1-\gamma_2)\left(\frac{\partial E}{\partial w^t}\right)^2$$

$$\Delta w^{t+1} = -\frac{\eta}{\sqrt{\text{MeanSquare}(w^t)}} v^t$$

In practice, the moving averages have bias-correction.

Discrete Convolution





-16	-8	7	-6	-22	-1	10
8	-2	-23	6	9	-18	8
19	-9	32	-15	**16**		

1D: $(f \circledast g)(x) = \sum_{i=-\infty}^{\infty} f(i)g(x-i)$

2D: $(f \circledast g)(x_1, x_2) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i,j)g(x_1 - i, x_2 - j)$

Convolutions in Image Processing

- Even though convolution is *commutative* ($f \circledast g = g \circledast f$), in practice we usually have a larger "input" matrix (typically an *image*) and a smaller matrix, called a **kernel,** which we "*convolve*" over the image.
- In the field of image processing, the **kernel** is also known as a **filter** or **mask** and it is used for *blurring*, *sharpening*, *edge detection* and other transformations of the input image.

$\circledast$

0	-2	0
-2	9	-2
0	-2	0

Sharpen

$\circledast$ $\frac{1}{256}$

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

Blur

$\circledast$

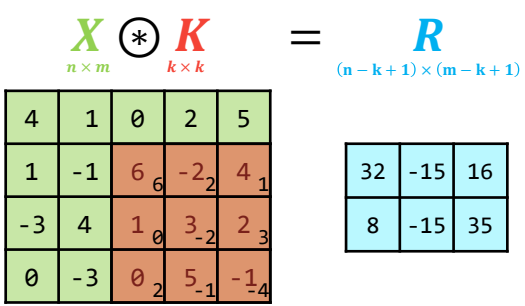
-1	-1	-1
-1	8	-1
-1	-1	-1

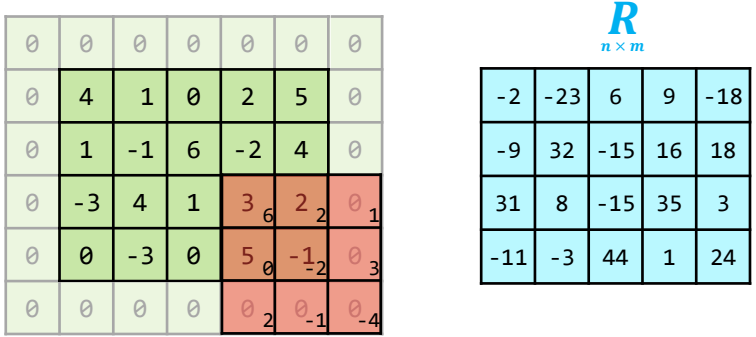
Edge Detection

Padding

Kernel matrix can be non-square, but it rarely happens in practice.

- Given an image $X \in \mathbb{R}^{n \times m}$ and a kernel $K \in \mathbb{R}^{k \times k}$, the mathematical definition of convolution will produce a result which is larger in size than X (it will be $(n + k - 1) \times (m + k - 1)$)
- The *convolution* used in image processing is only applied to the area in which the two matrices **overlap completely**.
  - This will result in a matrix $R \in \mathbb{R}^{(n-k+1) \times (m-k+1)}$.
- In practice, we sometimes want the output size to the be same as the input size ($R \in \mathbb{R}^{n \times m}$).
  - This means we need to **pad** the original image with a $^{k-1}/_{2}$ border of zeros.





"Valid" Padding

- Basically, "no" padding.
- Sometimes used in practice.

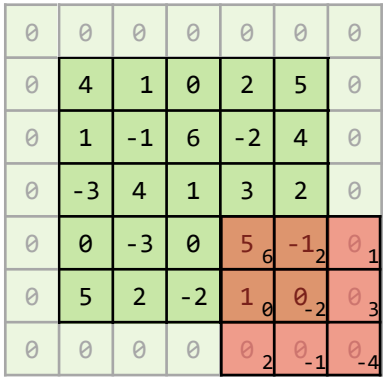
Kernel size *k* is usually odd. Otherwise we need unsymmetrical padding.

"Same" Padding

- Pad such that the output is the "same" size as the input
- Very common in practice.

Stride

- **Stride** is the amount by which the filter moves when it is convolved over the input image.



Stride $s = 2$

-2	6	-18
31	-15	3
-7	-14	28

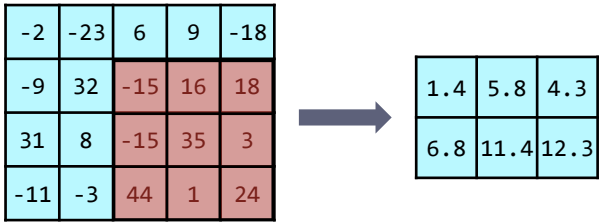
- If we have an image of size $n \times m$, a kernel of size $k \times k$ and we have padding p and stride s, the output will have size:

$$o = \left(\left\lfloor \frac{n - k + 2p}{s} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{m - k + 2p}{s} \right\rfloor + 1\right)$$

Pooling

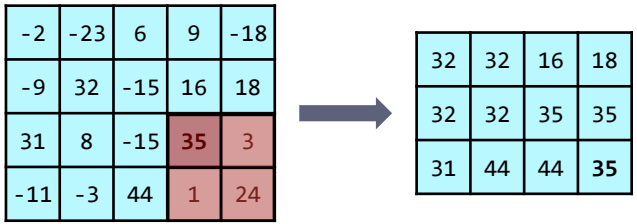
- **Pooling** is an operation which *down-samples* an input image and it is typically applied after a filter convolution.
  - The reasoning is that a filter usually tries *to capture a specific aspect* of the input image (e.g. edge detection) and it does not take as many pixels (features) to "describe" this particular aspect as for the original image.
  - Similarly to a convolution operation, pooling has *size*, *padding* and *stride*.

$\text{size} = 3x3, \text{stride} = 1$



Average Pooling

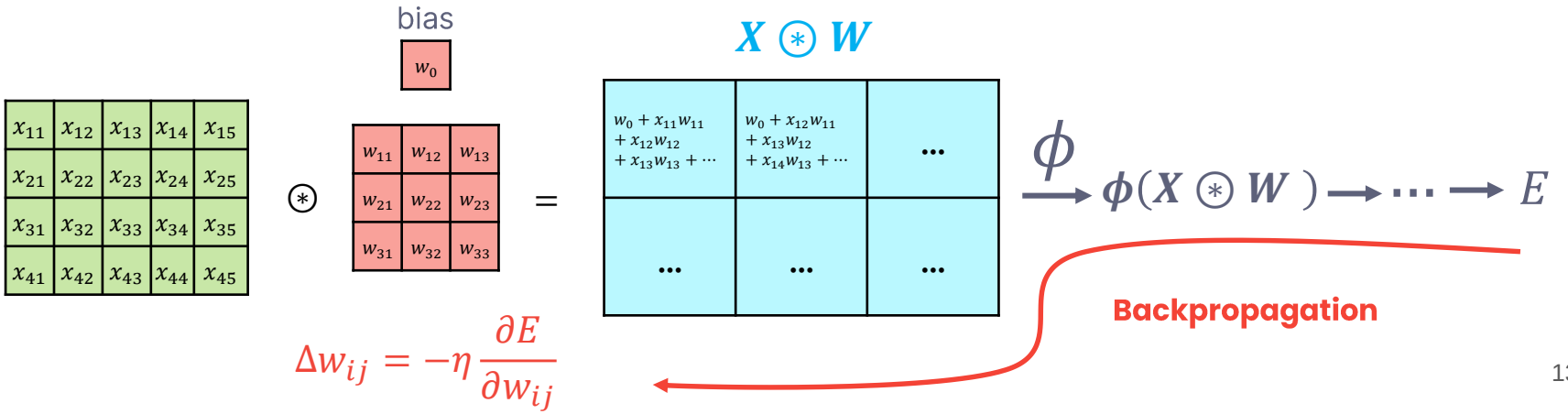
$\text{size} = 2x2, \text{stride} = 1$



Max Pooling
Most common in practice

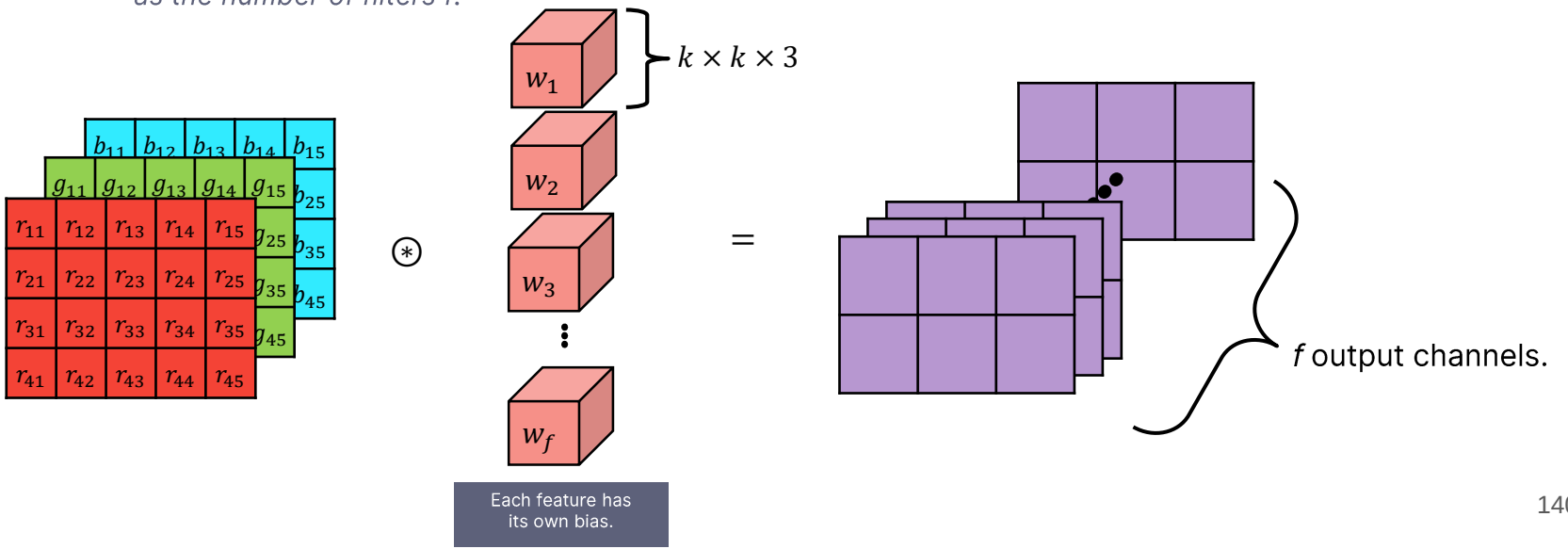
Convolutional Layer

- By choosing proper convolution kernels, we could substantially improve the performance of the predictive algorithm.
  - But in **Deep Learning**, *"choosing the proper convolutions"* should be the task of the learning algorithm itself.
- A **convolutional layer** is a filter (or a set of filters) whose *weights are learned* through *gradient descent* in the *optimization process*.



Multiple Channels

- One filter usually captures one feature of the input image, but we want to obtain a *representation with many features*.
  - A **convolutional layer** is usually made up of **multiple filters**.
  - Each filter has the *depth* (number of channels) of the input image and the output has *as many channels as the number of filters f*.



Motivation

- The problems we have seen so far are “one-to-one” (i.e. one input mapped to one output).
  - Better said, all inputs and all outputs always have the same fixed size.
    - e.g. In the digit recognition problem, all inputs had 28x28 = 724 pixels and all outputs had 10 values (one hot encoding of a digit).
- Many real problems involve sequences of data:

Speech Recognition

“What is the weather going to be like tomorrow?”

Image Captioning

“Black and white dog jumps over bar.”

Machine Translation

“I am going to the cinema.”

“Ich gehe ins Kino.”

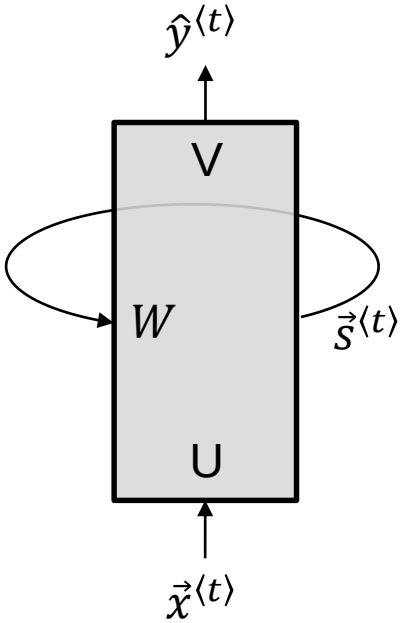
Notation

- One training example is now a **sequence of values**: $\vec{x} \in \mathbb{R}^{m \times n \times T}$
  - $\vec{x}_j^{(i)\langle t \rangle}$ - component *j* of training sample *i* at timestep *t*

$\vec{x}^{\langle t \rangle} \in \mathbb{R}^n$
input at "timestep" *t*

$\hat{y}^{\langle t \rangle} \in \mathbb{R}^o$
output at timestep *t*

$\vec{s}^{\langle t \rangle} \in \mathbb{R}^h$
hidden state at "timestep" *t*

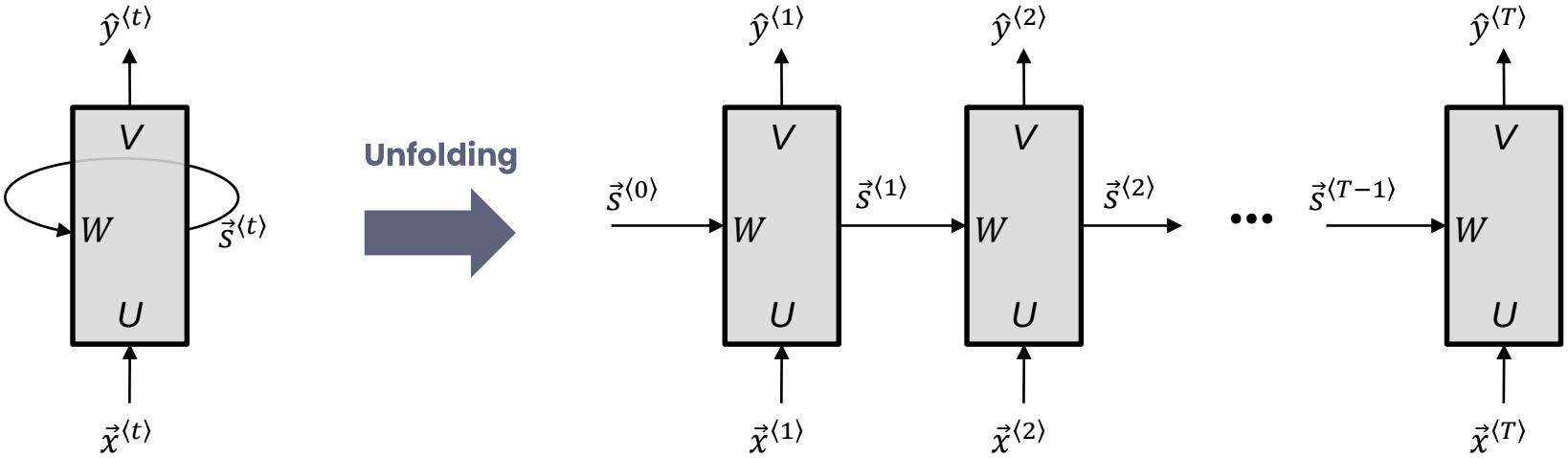


Three weight matrices:
- $U \in \mathbb{R}^{h \times n}$
- $W \in \mathbb{R}^{h \times h}$
- $V \in \mathbb{R}^{o \times h}$

$$\vec{s}^{\langle t \rangle} = \phi_1\left(W\vec{s}^{\langle t-1 \rangle} + U\vec{x}^{\langle t \rangle}\right)$$
$$\hat{y}^{\langle t \rangle} = \phi_2\left(V\vec{s}^{\langle t \rangle}\right)$$

ϕ_1 and ϕ_2 are different activation functions.
Usually ϕ_1 is a tanh and ϕ_2 is a sigmoid (or softmax)

Forward propagation



$$\vec{s}^{\langle 0 \rangle} = \vec{0}$$
$$s^{\langle t \rangle} = \phi_1\left(W\vec{s}^{\langle t-1 \rangle} + U\vec{x}^{\langle t \rangle}\right)$$
$$\hat{y}^{\langle t \rangle} = \phi_2\left(V\vec{s}^{\langle t \rangle}\right)$$

An RNN is a network, not just a neuron, even though it is sometimes called a "cell".

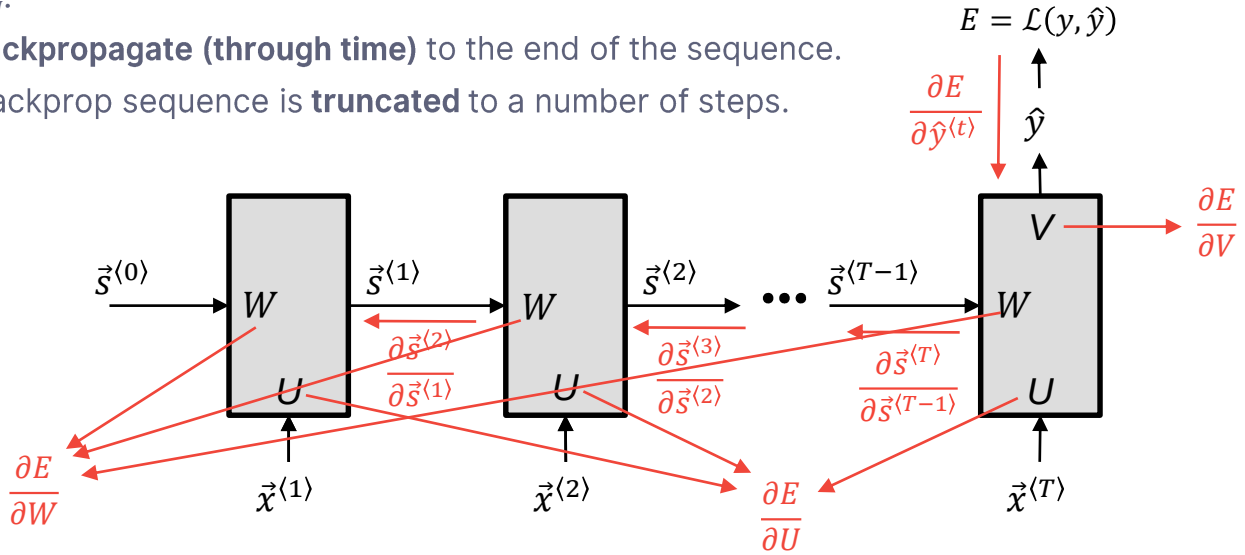
Backpropagation through time (BPTT)

- Like before, weights are updated through **backpropagation**:

$$\Delta V = -\eta \frac{\partial E}{\partial V} \qquad \Delta W = -\eta \frac{\partial E}{\partial W} \qquad \Delta U = -\eta \frac{\partial E}{\partial U}$$

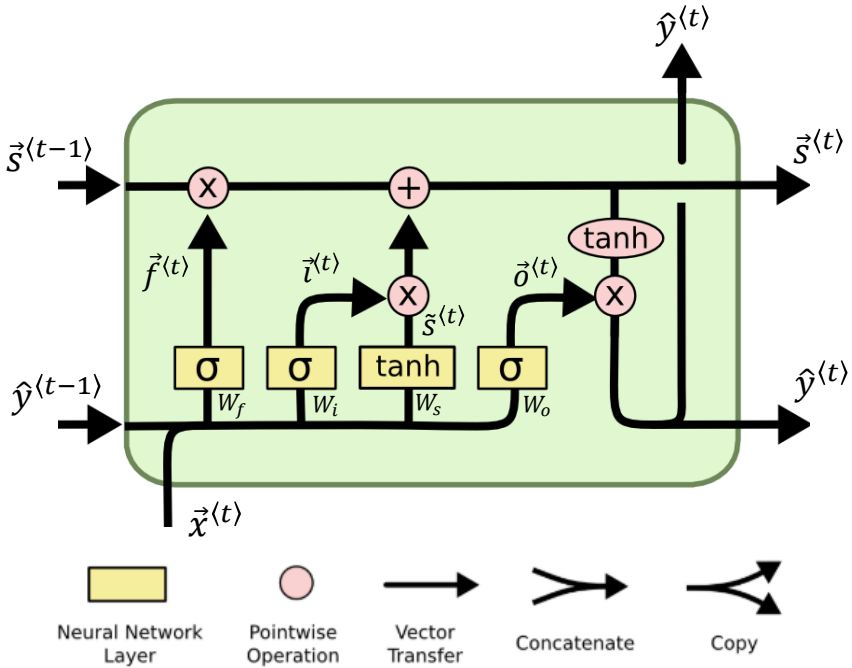
- $\partial E / \partial W$ depends on $\partial \vec{s}^{\langle T \rangle} / \partial W$, but $\vec{s}^{\langle T \rangle}$ depends on $\vec{s}^{\langle T-1 \rangle}$, which itself depends on W, an so on.
  - Same for $\partial E / \partial U$.
  - We need to **backpropagate (through time)** to the end of the sequence.
  - Usually, the backprop sequence is **truncated** to a number of steps.

This is a simpler "many-to-one" case but it can be generalized to "many-to-many".



Long Short-Term Memory

- Because of repeated application of the tanh activation, RNNs suffer from the **vanishing gradient** problem.
- ReLUs can be used to ameliorate this, but a much better way is to use a different inner structure, called an **LSTM** cell.



$$\vec{f}^{\langle t \rangle} = \sigma\left(W_f \cdot \left[\hat{y}^{\langle t-1 \rangle}, \vec{x}^{\langle t \rangle}\right] + b_f\right)$$ **"Forget" gate**

$$\vec{i}^{\langle t \rangle} = \sigma\left(W_i \cdot \left[\hat{y}^{\langle t-1 \rangle}, \vec{x}^{\langle t \rangle}\right] + b_i\right)$$ **"Input" gate**

$$\tilde{s}^{\langle t \rangle} = \tanh\left(W_s \cdot \left[\hat{y}^{\langle t-1 \rangle}, \vec{x}^{\langle t \rangle}\right] + b_s\right)$$ **State candidates**

$$\vec{s}^{\langle t \rangle} = \vec{f}^{\langle t \rangle} * \vec{s}^{\langle t-1 \rangle} + \vec{i}^{\langle t \rangle} * \tilde{s}^{\langle t \rangle}$$ **Hidden state (Memory)**

$$\vec{o}^{\langle t \rangle} = \sigma\left(W_o \cdot \left[\hat{y}^{\langle t-1 \rangle}, \vec{x}^{\langle t \rangle}\right] + b_o\right)$$ **"Output" gate**

$$\hat{y}^{\langle t \rangle} = \vec{o}^{\langle t \rangle} * \tanh\left(\vec{s}^{\langle t \rangle}\right)$$ **Output**

**Visuals are from http://colah.github.io/posts/2015-08-Understanding-LSTMs, but with different notation.*