DEEP LEARNING FOR VISUAL RECOGNITION

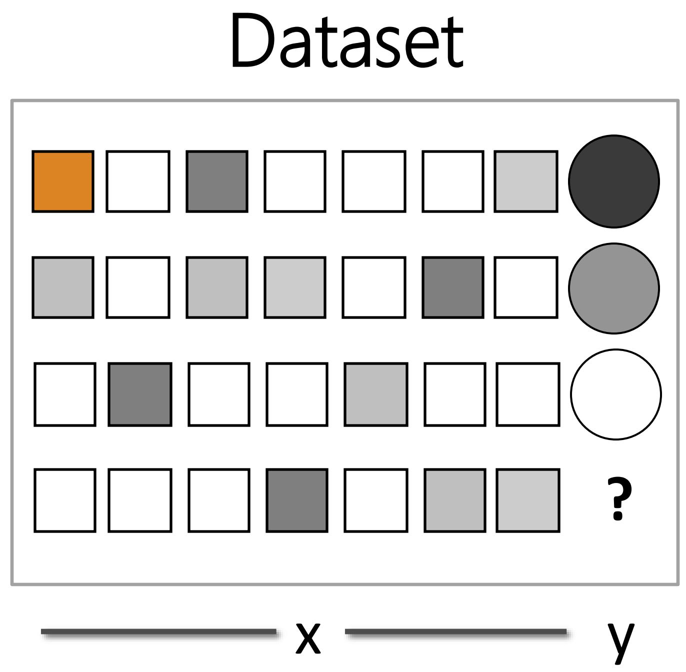
Lecture 2 – Machine learning fundamentals

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

Today’s agenda

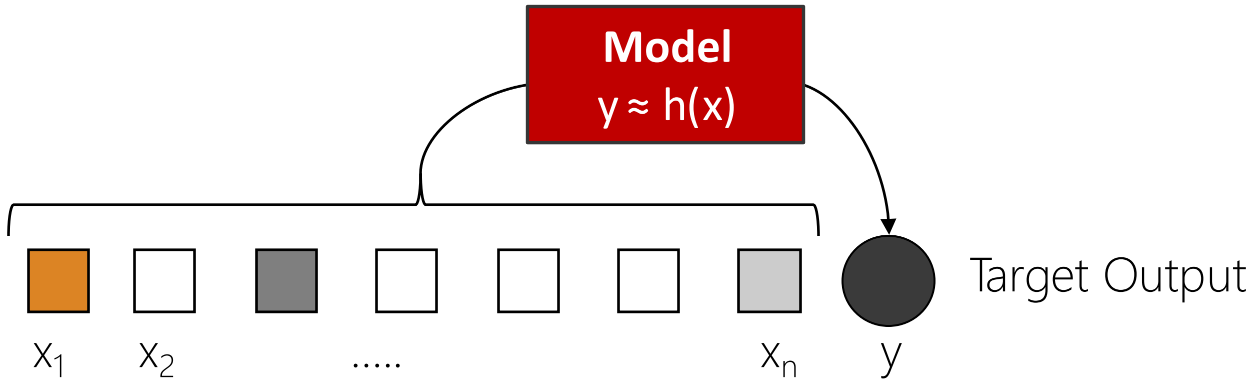
- You will learn about machine learning fundamentals.
- Topics
  - Linear regression
  - Gradient descent
  - Hyperparameters
  - Training set, validation set, test set
  - Logistic regression
  - Regularization with weight decay
  - Softmax regression
  - K-Nearest Neighbors
  - K-Means clustering

Recall: Learning principle



Goal:

- Learn a **model** that is good at predicting **y** from **x** across the entire training dataset
- A good/useful model generalizes to unseen data



Linear regression

INTRODUCTION TO LOSS FUNCTIONS AND OPTIMIZATION

Motivation

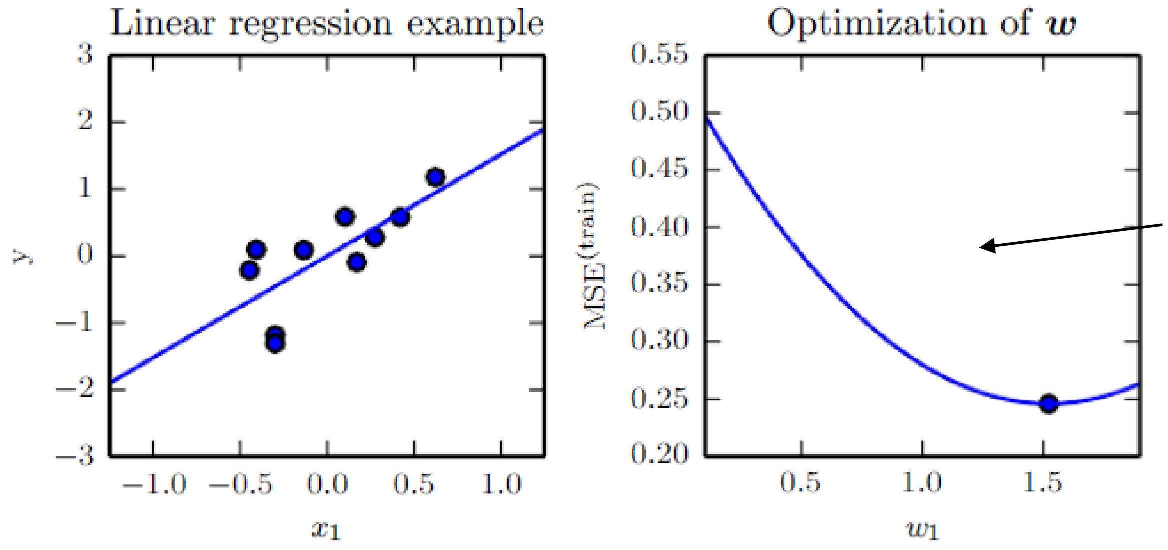
- We will start by learning how to implement linear regression.
- The main idea is to get familiar with loss functions, computing their gradients and optimizing the loss over a set of parameters.
- These basic tools will form the basis for more sophisticated algorithms later.

What is linear regression?

The goal is to build a **model** that takes a vector $\mathbf{x} \in \mathbb{R}^m$ as input and predicts the value of a scalar $y \in \mathbb{R}$ as its output. In the case of linear regression, the output is a linear function of the input.

Model

$y \approx h(x_1) = w_1 x_1$



This curve shows the loss for different choices of parameter (w_1).

The best choice of w_1 is the one that minimizes the loss.

Figure 5.1: A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)*Observe that linear regression learns to set w_1 such that the line $y = w_1 x$ comes as close as possible to passing through all the training points. *(Right)*The plotted point indicates the value of w_1 found by the normal equations, which we can see minimizes the mean squared error on the training set.

Goal in linear regression

- Our goal is to predict a target value $y \in \mathbb{R}$ from a vector $\mathbf{x} \in \mathbb{R}^m$ of input values.
- For example, we might want to make predictions about the price of a house so that **y** represents the price of the house in dollars and the elements of **x** represent "features" that describe the house (such as its size and the number of bedrooms).
- Suppose that we are given **n** examples of houses where the features for the i'th house are denoted $\mathbf{x}^{(i)}$ and the price is $\mathbf{y}^{(i)}$. For short, we will denote the training set

$$\{y^{(i)}, x^{(i)}\}_{i=1}^{n}$$

- Our goal is to find a function $\mathbf{y = h(x)}$ so that we have $\mathbf{y^{(i)} \approx h(x^{(i)})}$ for each training example.
- If we succeed in finding a function **h(x)** like this, and we have seen enough examples of houses and their prices, we hope that the function **h(x)** will also be a good predictor of the house price even when we are given the features for a new house where the price is not known.

Model and loss function

- To find a function **h(x),** where $\mathbf{y^{(i)} \approx h(x^{(i)})}$, we must first decide how to represent the function **h(x)**.
- To start out we will use a linear model:

inner product

$$h_w(x) = \sum_{j=1}^{m} w_j x_j = w^T x$$

- Here, $h_w(x)$ represents a large family of functions parametrized by the choice of **w**. Formally, this space of functions is called the "hypothesis class", but in practise we will refer to it as our **model**.
- With this choice of model, our task is to find a choice of **w** such that $\mathbf{h_w(x^{(i)})}$ is as close as possible to $\mathbf{y^{(i)}}$ for all samples, **i**. In particular, we will search for a choice of **w** that minimizes the L2 norm:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(w^T x^{(i)} - y^{(i)}\right)^2$$

Optimization

- We now want to find the choice of **w** that minimizes the loss **J(w)**.
- Landscape analogy: Find the bottom of a valley in the loss landscape.
- Important to keep in mind that the computer cannot just “eyeball” the landscape.
- We need some kind of heuristic that takes us to a valley.

Strategy #1: Random search

- Bad idea – we are just guessing…
- Could we search more intelligently?

```
bestloss = float("inf")
for num in range(1000):
  w_random = np.random.randn(m,1)
  loss = J(w_random,X_train,Y_train)
  if loss < bestloss:
    bestloss = loss
    bestw = w_random
```

Strategy #2: Follow the slope

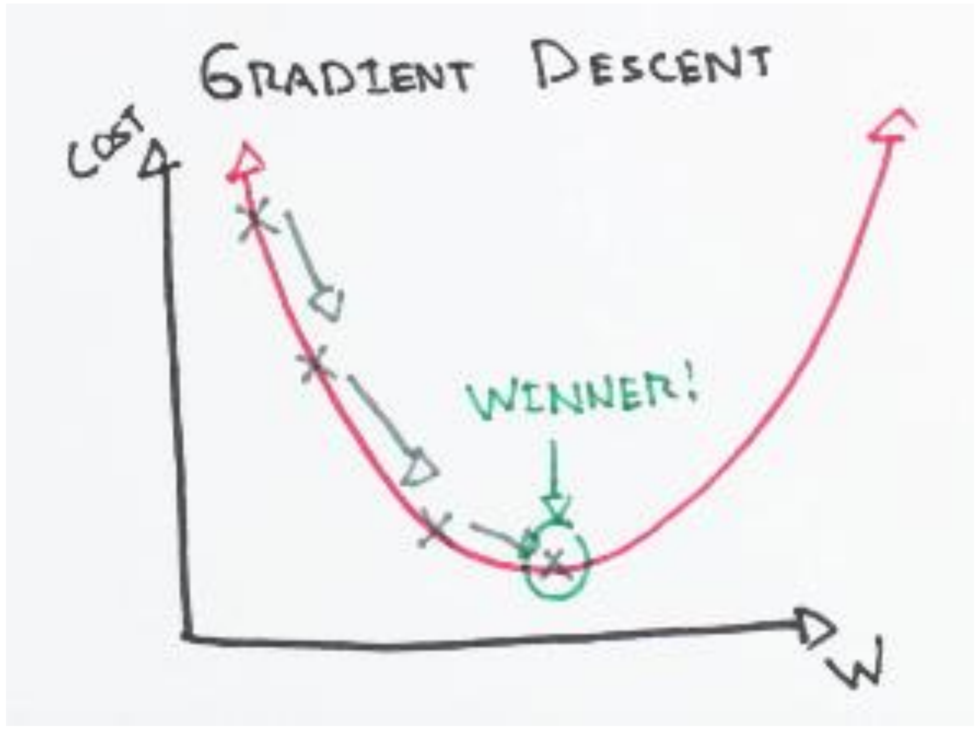
- Gradient descent (or steepest descent) is like a person walking down a hill.
- The person follows the steepest path downwards; progress is slow, but steady.

Gradient descent

- When optimizing a smooth function **J(w)**, we make a small step along the gradient*

$$w^{k+1} = w^k - \alpha \nabla J(w^k)$$

- w^k is our current guess, and w^{k+1} is our updated and improved guess.
- α is called the **learning rate** or **step size**.
- For a learning rate small enough, gradient descent makes a monotonic improvement at every iteration (i.e., **J(w)** becomes smaller).
- Gradient descent always converges, albeit to a local minimum.



* If you forgot what a gradient is, see the next slide ☺

Gradient recap

- The partial derivative of function f with respect to (w.r.t.) x_i is written

$$\frac{\partial}{\partial x_i} f(\mathbf{x})$$

- It measures how fast f changes as only the variable x_i increases at point $\mathbf{x}$.
- The gradient generalizes the notion of a derivative from scalar-valued functions to vector-valued functions, i.e., the gradient of f is the vector containing all of the partial derivatives, denoted

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_N} f(\mathbf{x}) \end{bmatrix}$$

Gradient descent in linear regression

- Start by initializing w^0 with zeros or random values.
- Then iteratively update according to

$$w^{k+1} = w^k - \alpha \nabla J\left(w^k\right)$$

- Recalling that

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(w^T x^{(i)} - y^{(i)}\right)^2$$

- The j'th element of ∇J is

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{n} x_j^{(i)}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right) = \sum_{i=1}^{n} x_j^{(i)}\left(w^T x^{(i)} - y^{(i)}\right)$$

Gradie ssion

Chain rule of differentiation:

If $\mathrm{p} = g(w)$ and $\mathrm{q} = f(p)$, then $\frac{dq}{dw} = \frac{dp}{dw}\frac{dq}{dp}$

Example:

$p = (wx - y)$ and $q = \frac{1}{2}p^2 = \frac{1}{2}(wx - y)^2$

$\frac{dp}{dw} = x$ and $\frac{dq}{dp} = p$

$$\frac{dq}{dw} = \frac{dp}{dw}\frac{dq}{dp} = xp = x(wx - y)$$

- Start by initial
- Then iterative
- Recalling that

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(w^T x^{(i)} - y^{(i)}\right)^2$$

How to get from here …

- The j’th element of ∇J is

… to here?

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{n} x_j^{(i)}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right) = \sum_{i=1}^{n} x_j^{(i)}\left(w^T x^{(i)} - y^{(i)}\right)$$

Limitations of gradient descent

- Gradient decent has many virtues, but speed is not one of them.
- Later in the course we will look at different ways to speed up gradient descent, such as using momentum.
- Also, gradient descent is only guaranteed to find a local minimum.

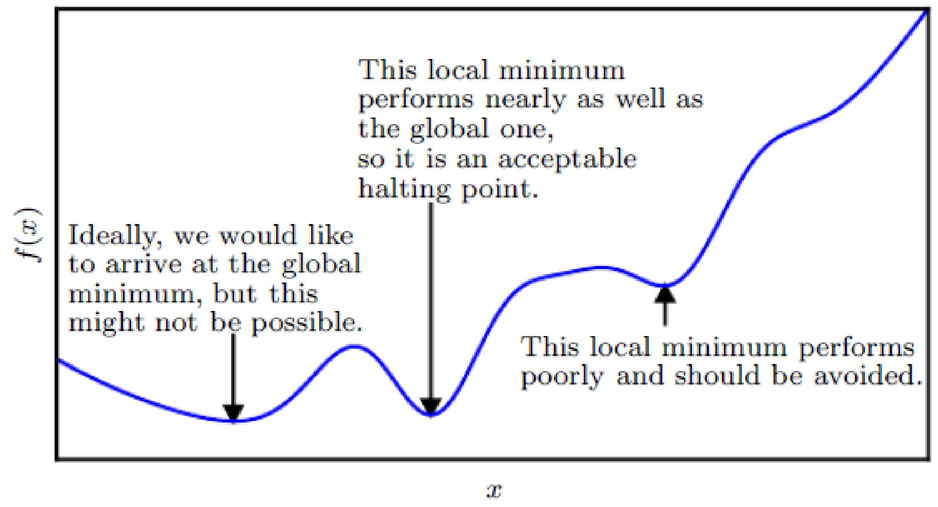
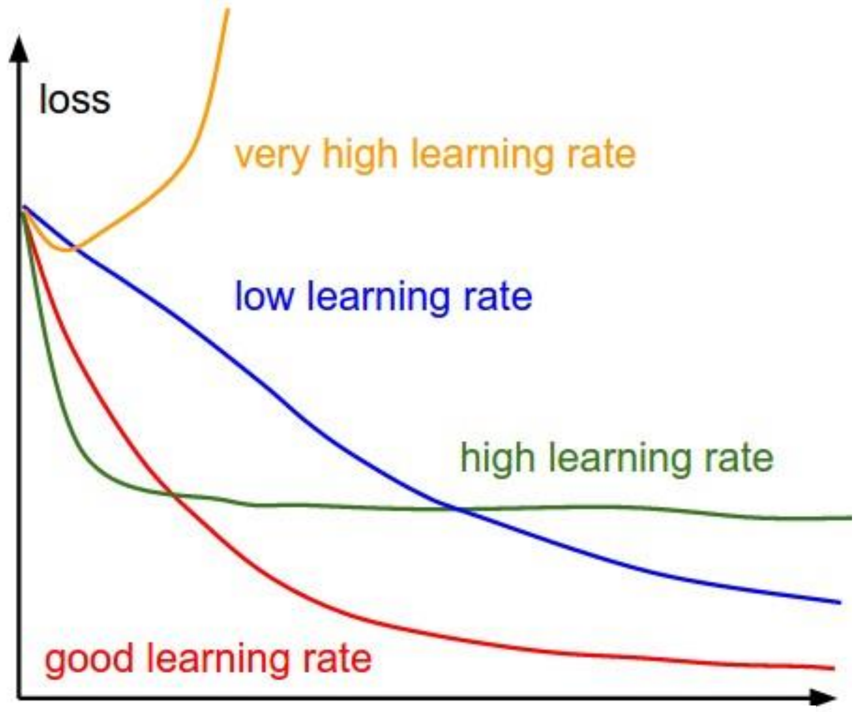


Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function.

Learning rate

- The learning rate is a **hyperparameter** that controls how much we are adjusting the weights of our model with respect the loss gradient $\nabla J(w)$.
- The diagram demonstrates the different scenarios one can fall into when configuring the learning rate (also see next slide)
- It is in general a good idea to decrease the learning rate over time to fine-tune the model parameters.

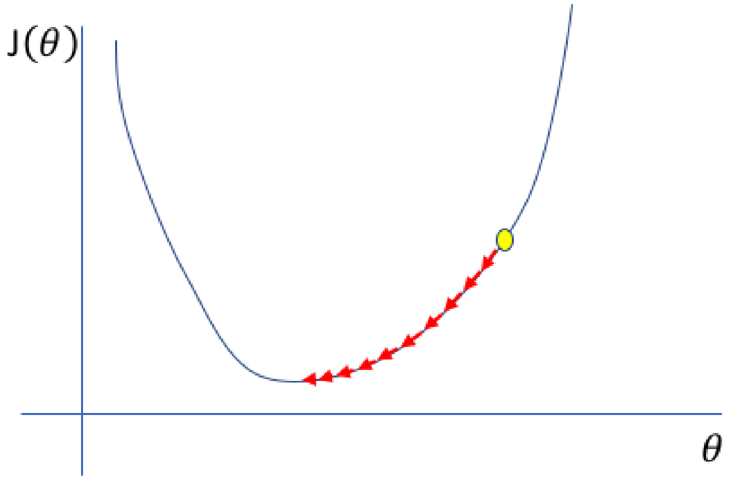


Training time

$$w^{k+1} = w^k - \alpha \nabla J\left(w^k\right)$$

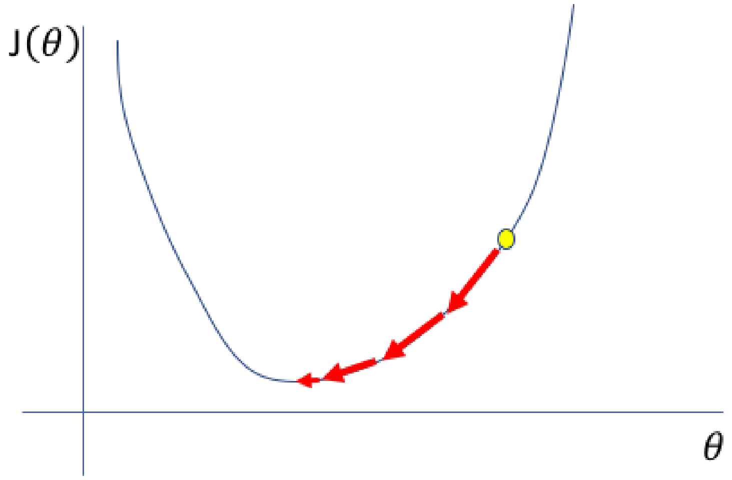
Learning rate

Too low



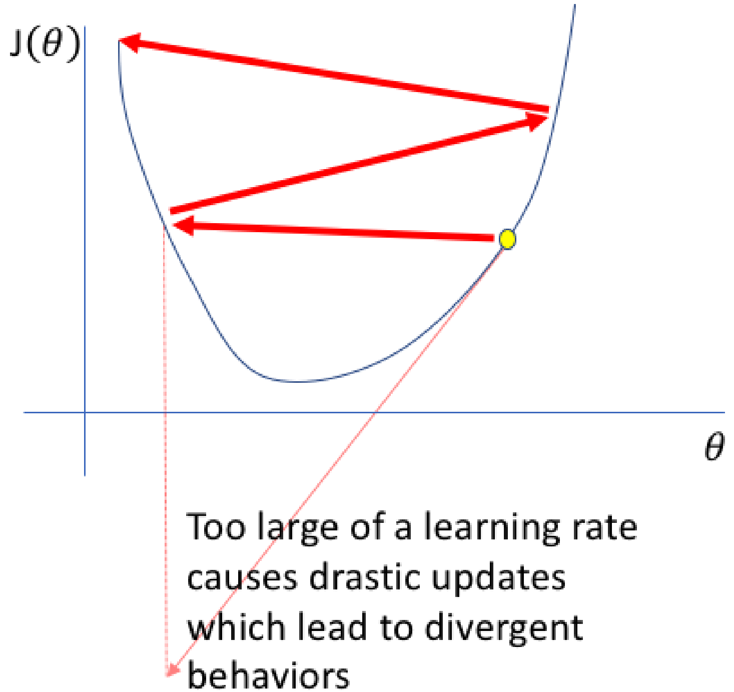
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high

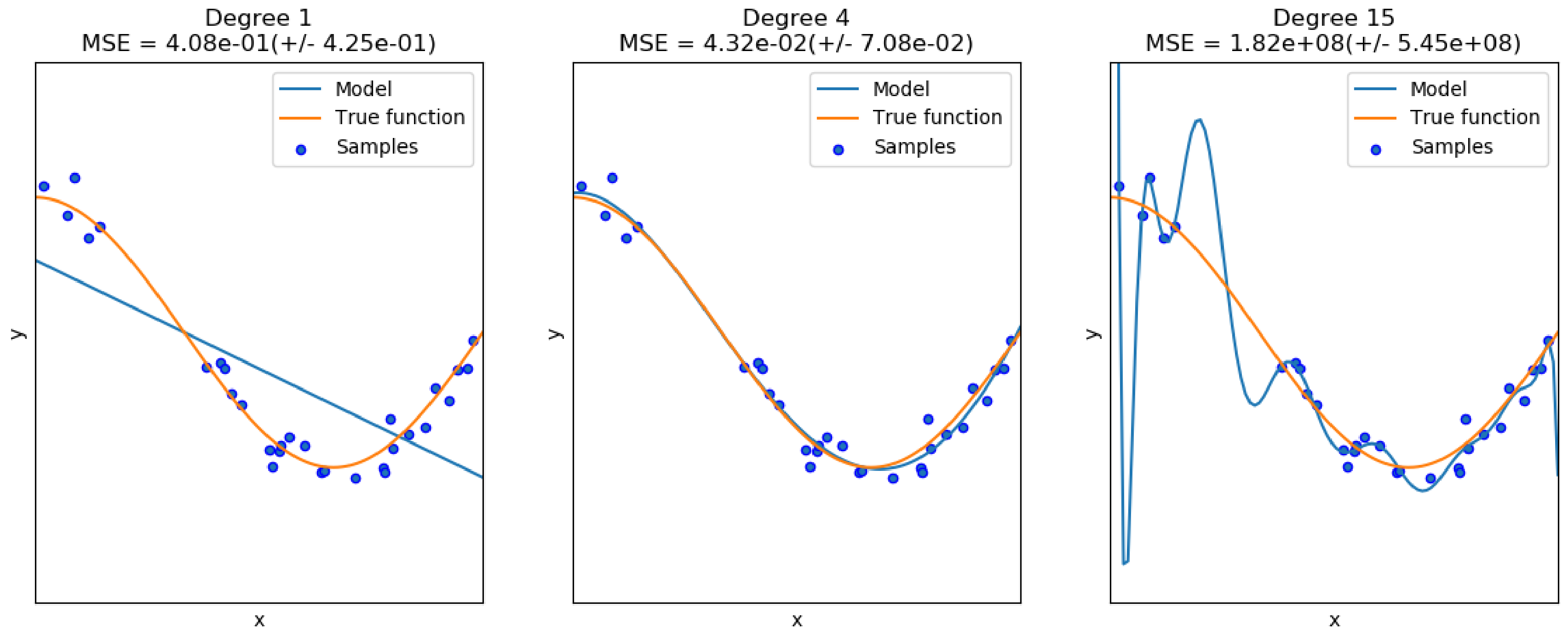


Too large of a learning rate causes drastic updates which lead to divergent behaviors

Linear regression example: polynomial fit

- This model, though nonlinear in the input **x**, is linear in the weights (**w**), and therefore we can write the model as a linear combination of monomials, like:

$$h_w(x) = \sum_{j=1}^{m} w_j x^{j-1}$$

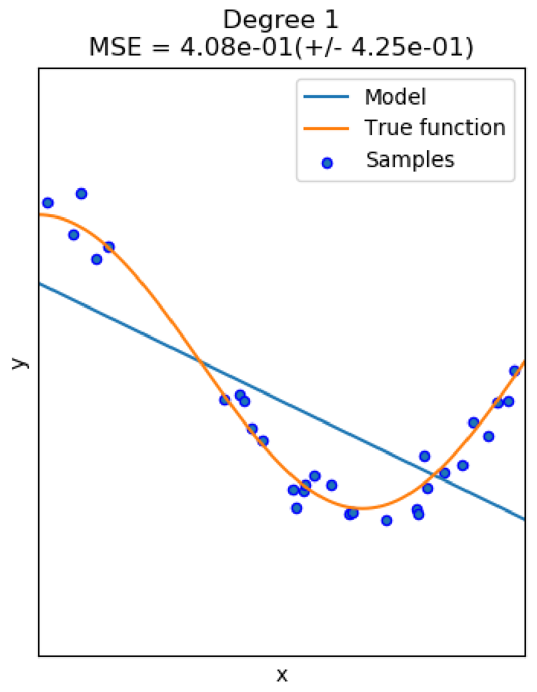


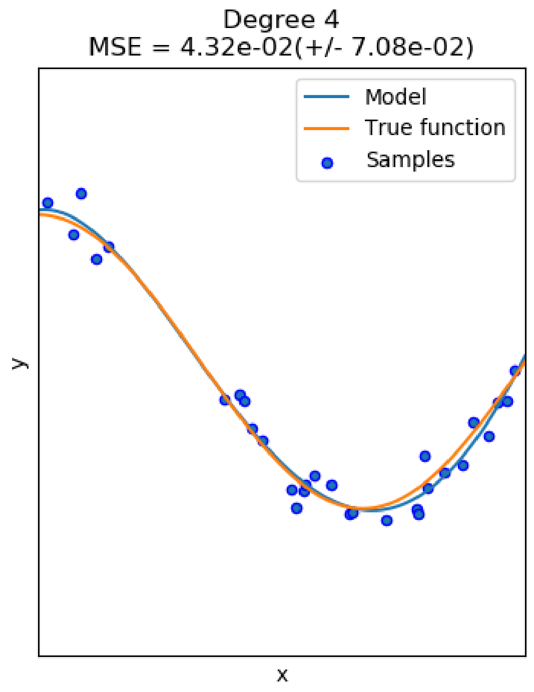
Underfitting and overfitting

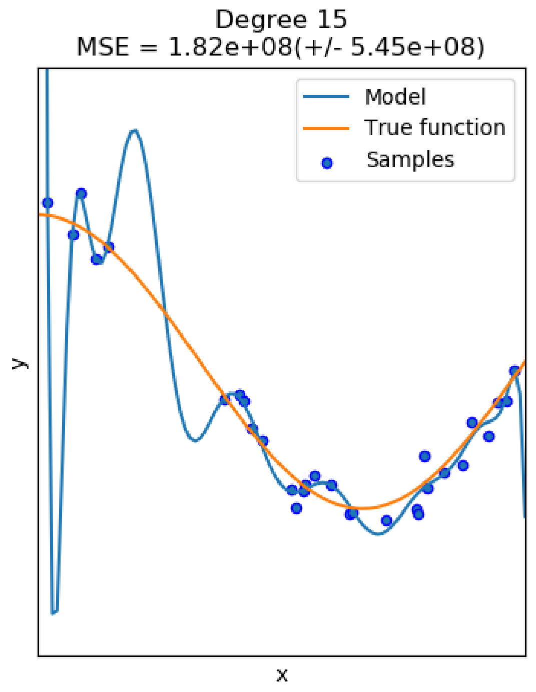
Underfitting: When the model has too low capacity to capture the variation that is present in the data.

Appropriate capacity: When the model fits to the data and generalizes well to unseen points.

Overfitting: When the model fits the training data well (almost exactly) but fails to generalize to unseen data.







Side-note: Why is it linear?

- This model, though nonlinear in the input **x**, is linear in the weights (**w**), and therefore we can write the model as a linear combination of monomials, like:

$$h_w(x) = \sum_{j=1}^{m} w_j x^{j-1}$$

- Reminder: The mapping T is linear if the following two conditions are met for all vectors **u** and **v** and all scalars α**:**

$$T(\boldsymbol{u}+\boldsymbol{v}) = T(\boldsymbol{u}) + T(\boldsymbol{v})$$
$$T(\alpha\boldsymbol{v}) = \alpha T(\boldsymbol{v})$$

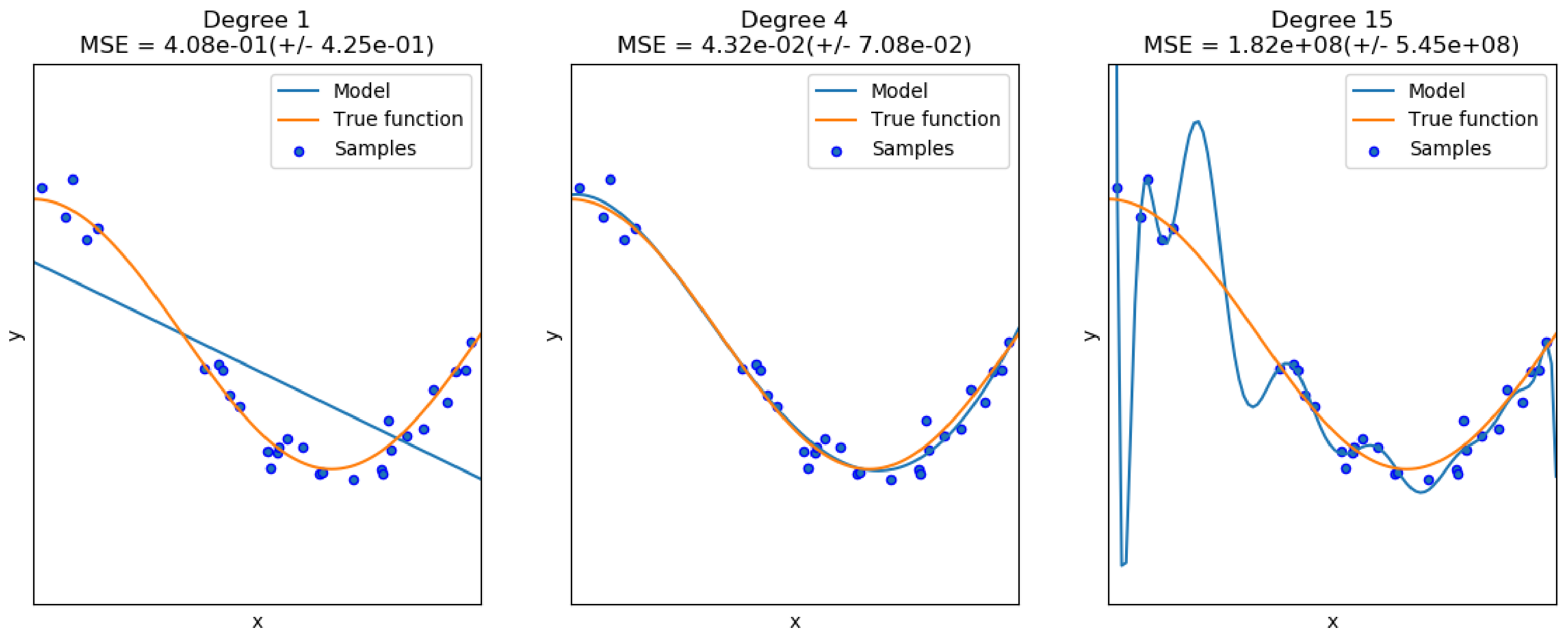
- You can verify for yourself that $h_w(x)$ is linear in the weights, i.e., that

$$h_{u+v}(x) = h_u(x) + h_v(x) \text{ and } h_{\alpha v}(x) = \alpha h_v(x)$$

Hyperparameters

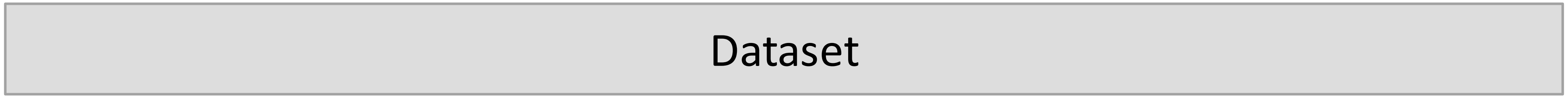
What are hyperparameters?

- In machine learning, a **hyperparameter** is a parameter whose value is set before the learning process begins. By contrast, the values of learnable parameters are derived via training.
- Example: In polynomial fitting, the degree of the polynomial is a hyperparameter, whereas the weight coefficient of each term in the polynomial is a trainable parameter.

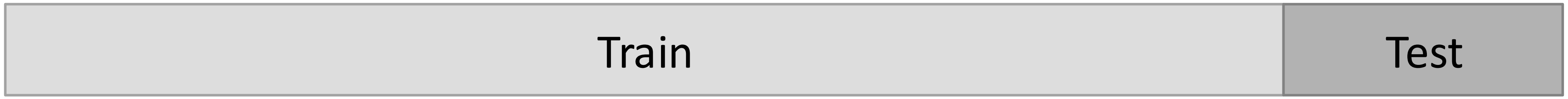


Hyperparameter search

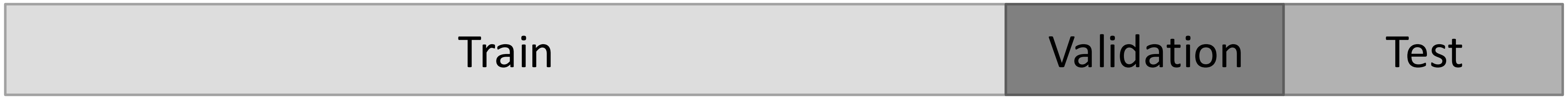
- **Idea #1:** Choose hyperparameters that work best on all data
- Bad: For a polynomial degree high enough, we could fit the training data perfectly (overfitting)



- **Idea #2:** Split data into **train** and **test**, choose hyperparameters that work best on test data
- Bad: No idea how algorithm will perform on new data

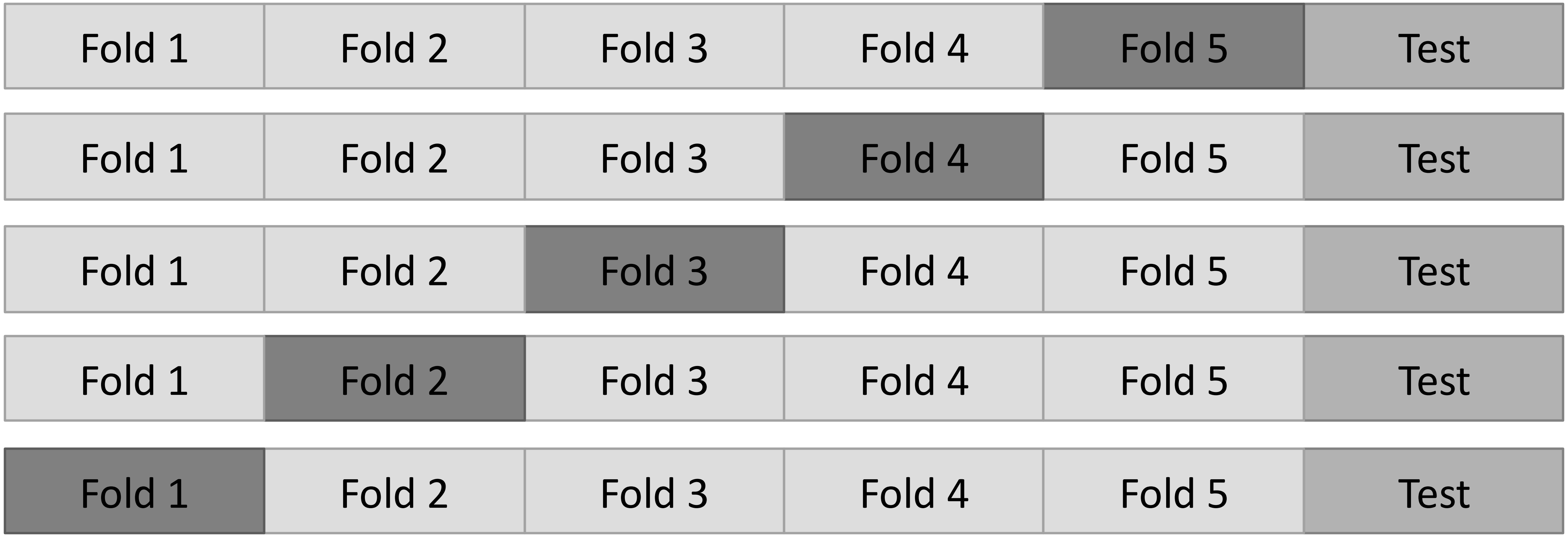


- **Idea #3:** Split data into **train**, **val**, and **test**; choose hyperparameters on val and evaluate on test
- Better!!!

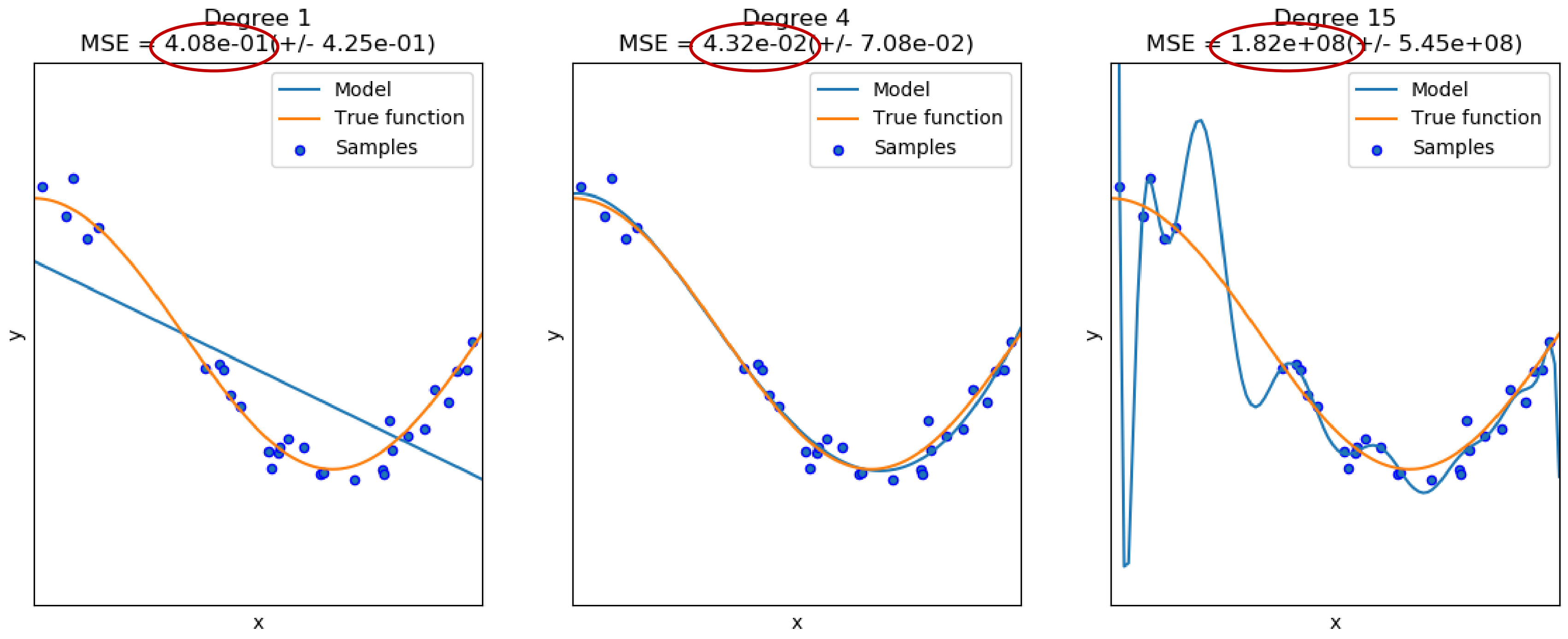


Hyperparameter search

- **Idea #4: Cross-validation:** Split data into **folds**, try each fold as validation and average the results
- Useful for small datasets, but not used too frequently in deep learning



Hyperparameter search – example



Example:

- Say we have created a validation set for our polynomial fitting problem (not shown).
- For each polynomial degree, we can calculate the mean squared error (MSE) on the validation set.
- The optimal polynomial degree is 4, because it results on the lowest MSE on the validation set.

Logistic Regression

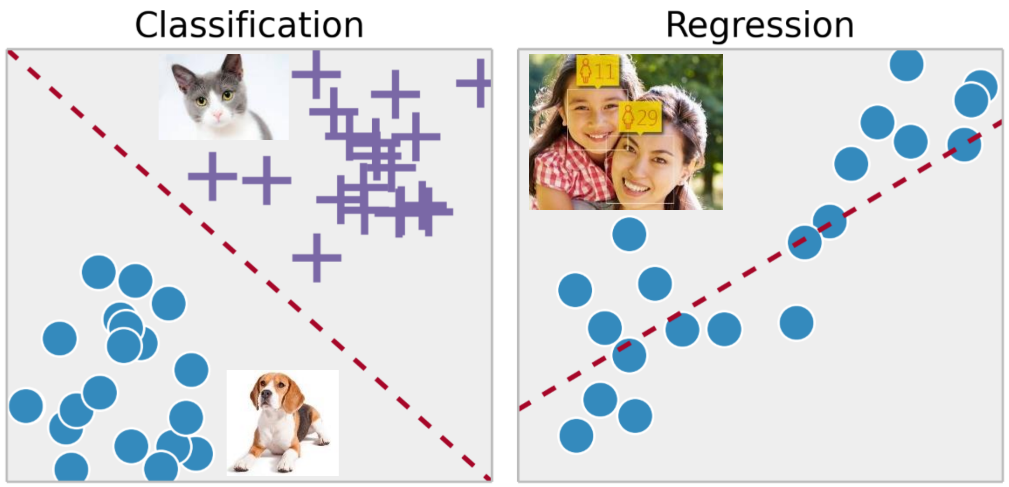
MOVING ON TO CLASSIFICATION

Regression vs. classification

- Above we learned how to predict continuous-valued quantities (e.g., age of a person) as a linear function of input values (e.g., a grid of pixel intensities).
- Sometimes we want to predict a discrete variable instead, such as predicting whether a grid of pixel intensities represents a "cat" or a "dog". This is a classification problem.
- Logistic regression is a simple classification algorithm for learning to make such decisions.

Classifier
Uses features to distinguish between two or more classes.

Output:
Discrete labels
("Dog" or "Cat")



Regressor
Uses features to predict some functional relationship.

Output:
Real numbers
("Age" of person in image)

Model

- In logistic regression our goal is to predict one of **two possible labels**, such that $y^{(i)} \in \{0,1\}$.
- Clearly, we need a different model than before, where we had $\mathrm{y} = h_w(x) = w^T x$.
- Candidate solution

$$h_w(x) = \begin{cases} 1 & \text{if } \; w^T x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Not going to work with gradient descent (why?).
- The trick is to pick a **differentiable function** that outputs **the probability** that a given example belongs to class “1” versus the probability that it belongs to class “0”:

$$h_w(x) = P(y = 1|x) \quad \rightarrow \quad P(y = 0|x) = 1 - P(y = 1|x) = 1 - h_w(x)$$

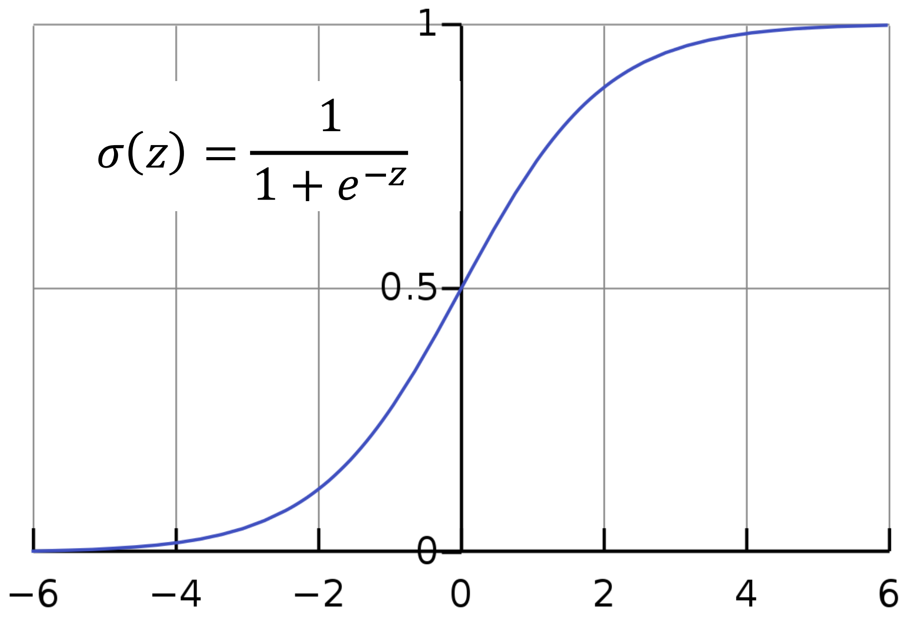
Model

- Specifically, we will try to learn a function of the form:

$$h_w(x) = P(y = 1|x) = \frac{1}{1 + e^{-w^T x}} \equiv \sigma(w^T x)$$

$$P(y = 0|x) = 1 - P(y = 1|x) = 1 - h_w(x)$$

- **Sigmoid function:**
  - The function $\sigma(z) = \frac{1}{1+e^{-z}}$ is often called the "sigmoid" or "logistic" function.
  - It "squashes" the value of $\mathrm{z} = w^T x$ into the range [0,1] so that we may interpret $h_w(x)$ as a **probability**.



Model

- Recall that $h_w(x) = P(y = 1|x)$
- Not quite what we want – we want our classifier to output class labels, $y \in \{0,1\}$
- We use this convention:

$$y = \begin{cases} 1 & \text{if } h_w(x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Loss function

- For a set of training examples with binary labels

$$\{y^{(i)}, x^{(i)}\}_{i=1}^{n}, y^{(i)} \in \{0,1\}$$

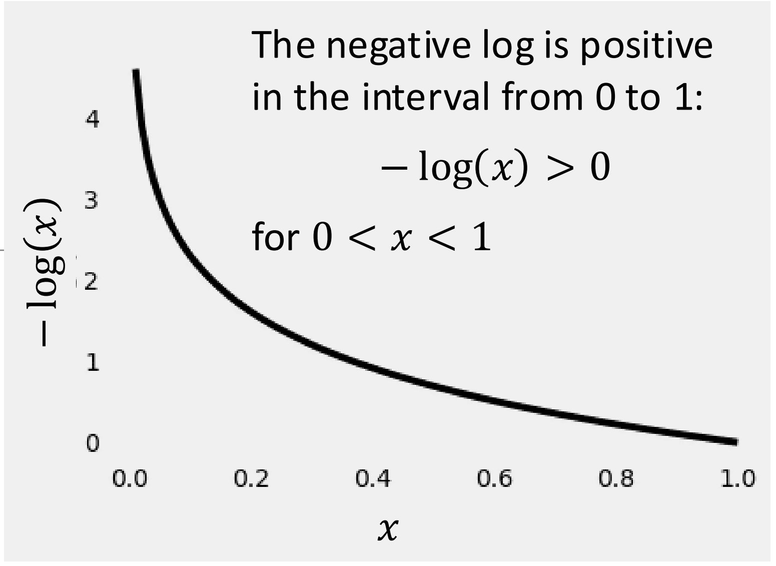
- the following cost function measures how well a given $h_w(x)$ fits the data:

$$J(w) = -\sum_{i=1}^{n} y^{(i)} \log\left(h_w\left(x^{(i)}\right)\right) + (1 - y^{(i)})\ \log\left(1 - h_w\left(x^{(i)}\right)\right)$$

- Note that only one of the two terms in the summation is non-zero for each training example (depending on whether the label $y^{(i)}$ is 0 or 1).
- When $y^{(i)} = 1$ we need $h_w\left(x^{(i)}\right) \approx 1$ to minimize the loss
- When $y^{(i)} = 0$ we need $1 - h_w\left(x^{(i)}\right) \approx 1$ or equivalently $h_w\left(x^{(i)}\right) \approx 0$ to minimize the loss.

Reminder: $0 \leq h_w(x) \leq 1$

Loss function



The negative log is positive in the interval from 0 to 1:

$$-\log(x) > 0$$

for $0 < x < 1$

Q: Why negative?
A: We want the loss to be positive and have a minimum value of 0. In our case, the inputs to the log lie in the range from 0 to 1, where the log is negative. So, we need the minus to make the loss positive…

ry labels

$(i)\}_{i=1}^{n}, y^{(i)} \in \{0,1\}$

w well a given $h_w(x)$ fits the data:

$$J(w) = -\sum_{i=1}^{n} y^{(i)} \log\left(h_w\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_w\left(x^{(i)}\right)\right)$$

- Note that only one of the two terms in the summation is non-zero for each training example (depending on whether the label $y^{(i)}$ is 0 or 1).
- When $y^{(i)} = 1$ we need $h_w\left(x^{(i)}\right) \approx 1$ to minimize the loss
- When $y^{(i)} = 0$ we need $1 - h_w\left(x^{(i)}\right) \approx 1$ or equivalently $h_w\left(x^{(i)}\right) \approx 0$ to minimize the loss.

Reminder: $0 \leq h_w(x) \leq 1$

Optimization

- To minimize $J(w)$ we can use the same tools as for linear regression. For logistic regression the derivate of $J(w)$ w.r.t w_j is

$$\frac{\partial J(w)}{\partial w_j} = \sum_{i=1}^{n} x_j^{(i)}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)$$

- This is essentially the same as the gradient for linear regression except that now $h_w(x) = \sigma(w^T x)$.
- See derivation here: http://cs229.stanford.edu/notes-spring2019/cs229-notes1.pdf

Side-note: Loss function derivation

- Recall that

$$P(y=1|x) = h_w(x) \qquad P(y=0|x) = 1 - P(y=1|x) = 1 - h_w(x)$$

- Recalling that label y is either 0 or 1 can rewrite this more compactly as

$$P(y|x) = \left(h_w(x)\right)^y \left(1 - h_w(x)\right)^{1-y}$$

- Assuming independence the probability (or likelihood) of observing $\left\{y^{(i)}, x^{(i)}\right\}_{i=1}^{n}$ is

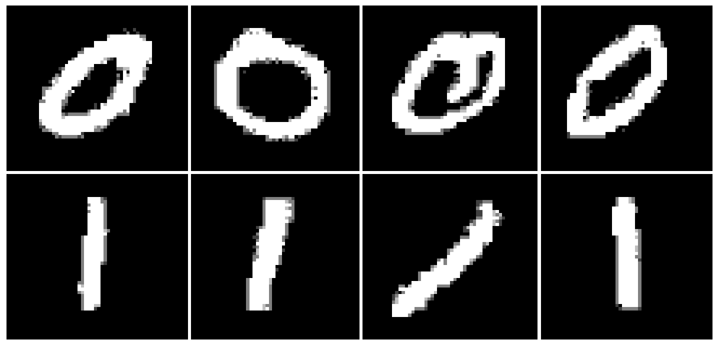
$$L = \prod_{i=1}^{n} P\left(y^{(i)} \middle| x^{(i)}\right) = \prod_{i=1}^{n} \left(h_w\left(x^{(i)}\right)\right)^{y^{(i)}} \left(1 - h_w\left(x^{(i)}\right)\right)^{1-y^{(i)}}$$

- Finding the **w** that maximizes the likelihood **L** is equivalent to minimizing the loss $J(w)$.
- Why? Taking the logarithm of **L** does not change the optimal choice of w, because the logarithm is monotonic. To derive the loss from L just apply the rules $\log(ab) = \log(a) + \log(b)$ and $\log(x^p) = p\log(x)$.

Our first image classifier

Logistic regression with image data

- You can use logistic regression to classify images.
- The “Hello world” dataset for image classification is the MNIST dataset of handwritten digits.
- Let’s consider the task of distinguishing between 0’s and 1’s.
- Some examples of these digits are shown below:



- Each of the digits is represented by a 28x28 grid of pixel intensities.
- The label is binary, so $y^{(i)} \in \{0,1\}$.

Images as vectors

- **Problem:** Our logistic regression model only operates on vectors.
- In order to make the math work, we need to turn the images into vectors.
- **Solution:** Just concatenate image rows into a single row vector.
- … or use numpy's reshape function:

train images test images

train labels test labels

```
Xtr, Ytr, Xte, Yte = load_MNIST('data/mnist/') # a magic function we provide

# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 28 * 28) # Xtr_rows becomes 50000 x 784
Xte_rows = Xte.reshape(Xte.shape[0], 28 * 28) # Xte_rows becomes 10000 x 784
```

Reminder – inner product

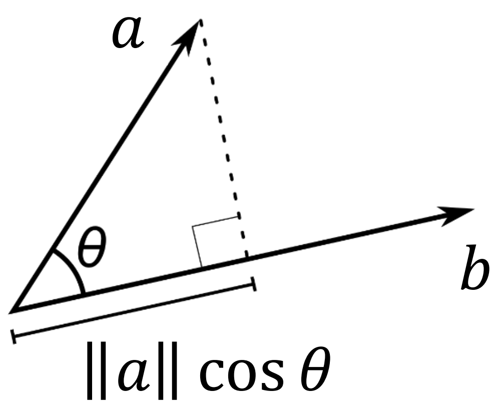
- Recall that the inner product or dot product between two (row) vectors **a** and **b** of length *n* is

$$a \cdot b = ab^T = \sum_{i=1}^{n} a_i b_i$$

- The inner product satisfies

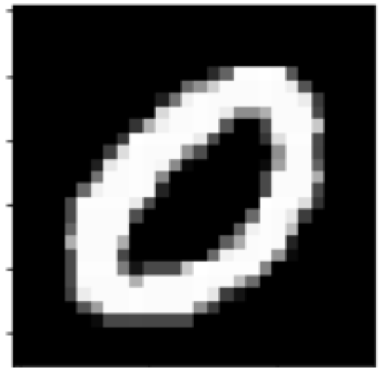
$$a \cdot b = \|a\| \|b\| \cos\theta$$

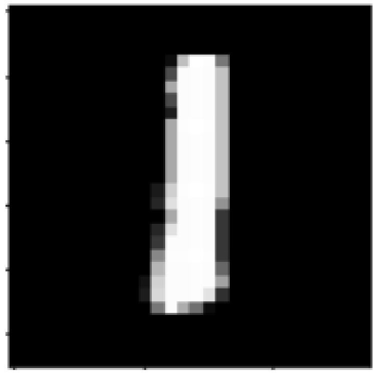
- We can think of the inner product as a **similarity measure**.
- Easy to see if we normalize the lengths such that $\|a\| = \|b\| = 1$.
- Then if **a** and **b** point in the same direction, $\theta = 0$ and $a \cdot b = \cos(0) = 1$.
- On the other hand, if two vectors point in opposite direction of each other, $\theta = \pi$ and $a \cdot b = \cos(\pi) = -1$.



Initial random weights

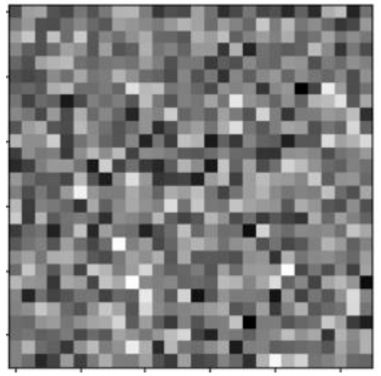
Input $x^{(i)}$





$y^{(i)} = 0$

Initial weights (w)



$y^{(i)} = 1$

$w^T x = 0.190$

$$P(y=1|x) = h_w(x) = \frac{1}{1+e^{-w^T x}} = 0.548 \approx 1$$

$$J(w) = -\left(y^{(i)} \log\left(h_w(x^{(i)})\right) - (1-y^{(i)}) \log\left(1 - h_w(x^{(i)})\right)\right)$$
$$= -\log\left(1 - h_w(x^{(i)})\right) = -\log(1-0.548) = 0.793$$

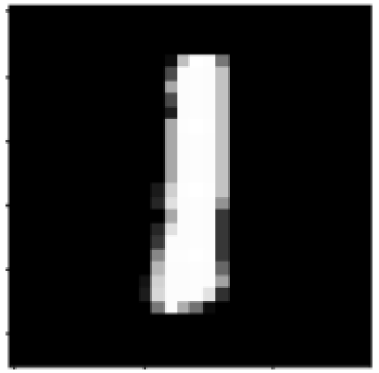
$w^T x = -0.229$

$$P(y=1|x) = h_w(x) = \frac{1}{1+e^{-w^T x}} = 0.443 \approx 0$$

$$J(w) = -\left(y^{(i)} \log\left(h_w(x^{(i)})\right) - (1-y^{(i)}) \log\left(1 - h_w(x^{(i)})\right)\right)$$
$$= -\log\left(h_w(x^{(i)})\right) = -\log(0.443) = 0.814$$

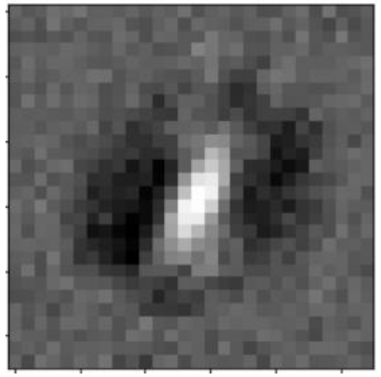
After training

Input $x^{(i)}$



$y^{(i)} = 0$

Learned weights (w)



$y^{(i)} = 1$

$w^T x = -10.08$

$$P(y=1|x) = h_w(x) = \frac{1}{1+e^{-w^T x}} = 4.17 \times 10^{-5} \approx 0$$

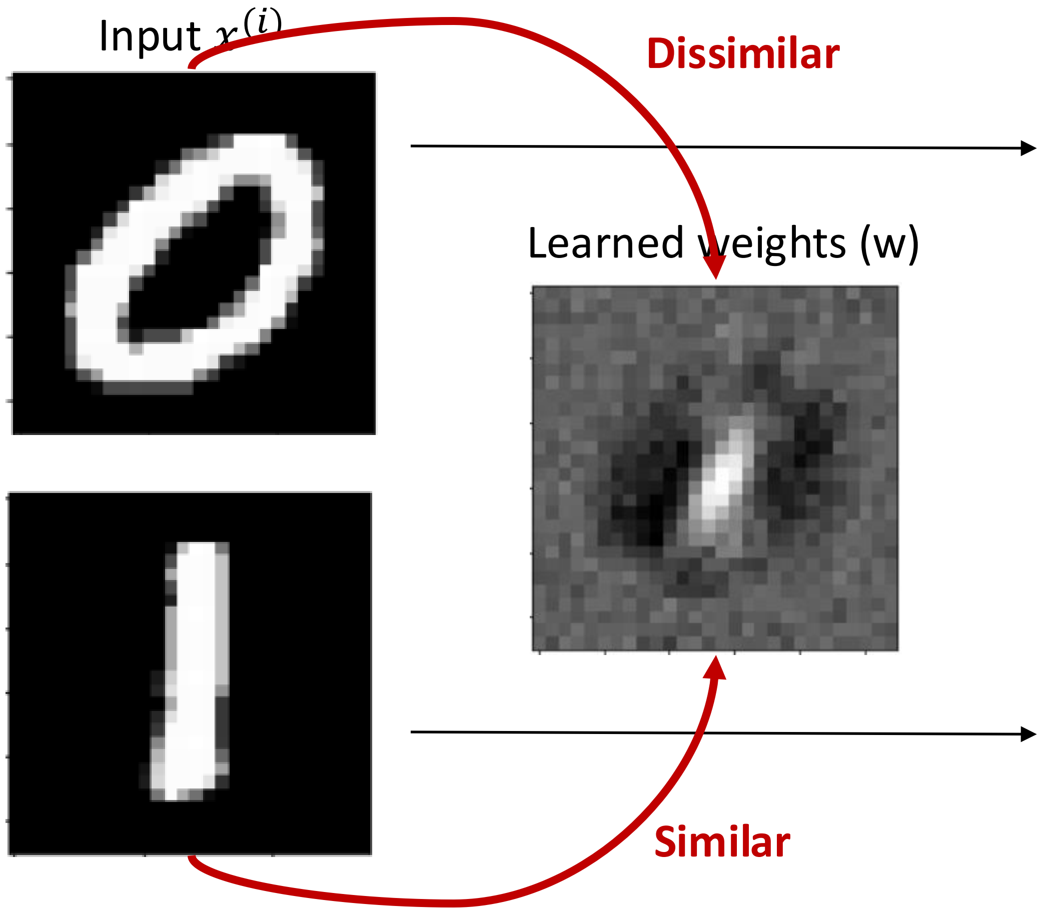
$$J(w) = -\left(y^{(i)} \log\left(h_w\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right) \log\left(1 - h_w\left(x^{(i)}\right)\right)\right) = -\log\left(1 - h_w\left(x^{(i)}\right)\right) = -\log(1 - 4.17 \times 10^{-5}) = 4.17 \times 10^{-5}$$

$w^T x = 6.264$

$$P(y=1|x) = h_w(x) = \frac{1}{1+e^{-w^T x}} = 0.998 \approx 1$$

$$J(w) = -\left(y^{(i)} \log\left(h_w\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right) \log\left(1 - h_w\left(x^{(i)}\right)\right)\right) = -\log\left(h_w\left(x^{(i)}\right)\right) = -\log(0.998) = 0.0019$$

What is going on?



$w^T x = -10.08$

$w^T x = 6.264$

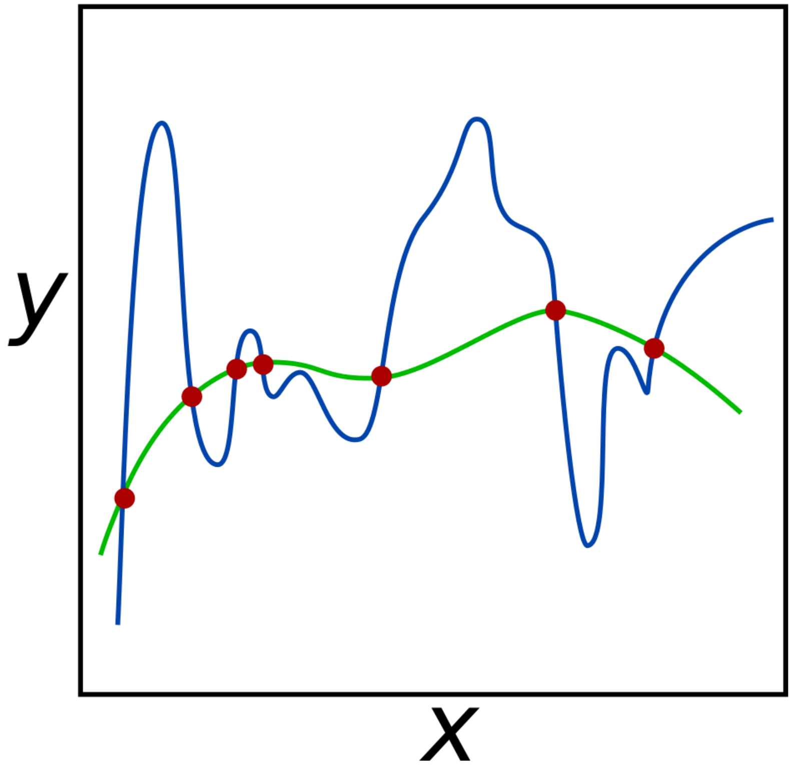
Intuition: The model learns to separate the two classes by adjusting the weights such that the inner product $w^T x$ is positive for 1’s and negative for 0’s.

Regularization

WEIGHT DECAY

General definition

- In machine learning and inverse problems, **regularization** is the process of adding information in order to solve an **ill-posed problem** or to **prevent overfitting**.
- Many different approaches that we will discuss in more detail later in the course:
  - Data augmentation
  - Early stopping
  - Dropout
  - Batch normalization
  - Weight decay
  - Sparsity
- Today, we will only look at weight decay.



Example of overfitting a polynomial

Weight decay – intuition

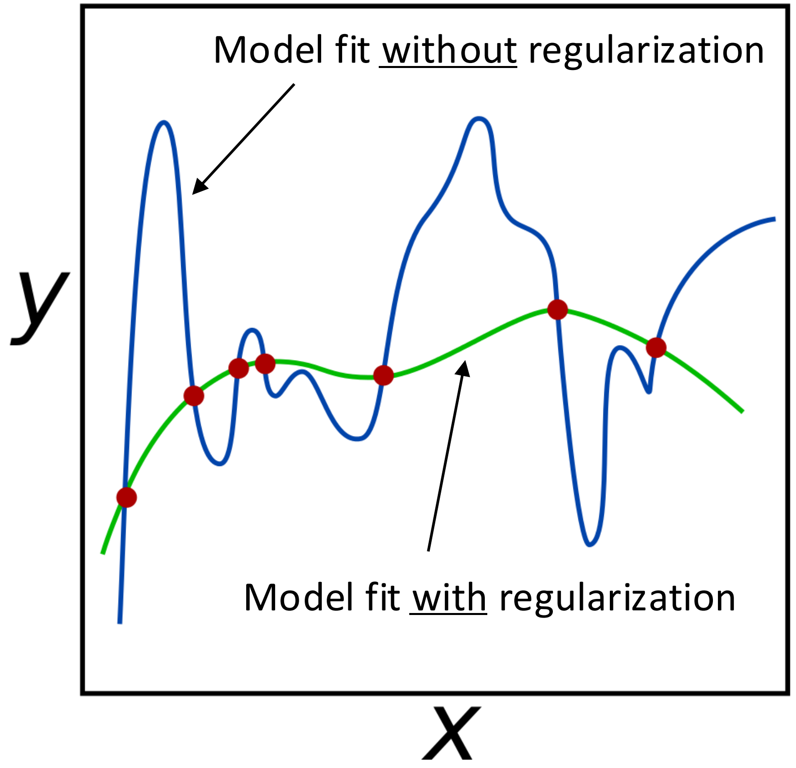
- Assume our model is a 15-degree polynomial

$$h_w(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_8 x^8 + \cdots + w_{15} x^{15}$$

- Overfitting can happen when the model is too complex and, hence, capable of fitting noise in the data.
- What weight decay does is to suppress unimportant terms in the polynomial

$$h_w(x) = w_0 + \cancel{w_1 x} + w_2 x^2 + \cdots + w_8 x^8 + \cdots + \cancel{w_{15} x^{15}}$$

- This is accomplished by keeping the weights small via an extra term in the loss function.



https://medium.com/datadriveninvestor/l1-l2-regularization-7f1b4fe948f2

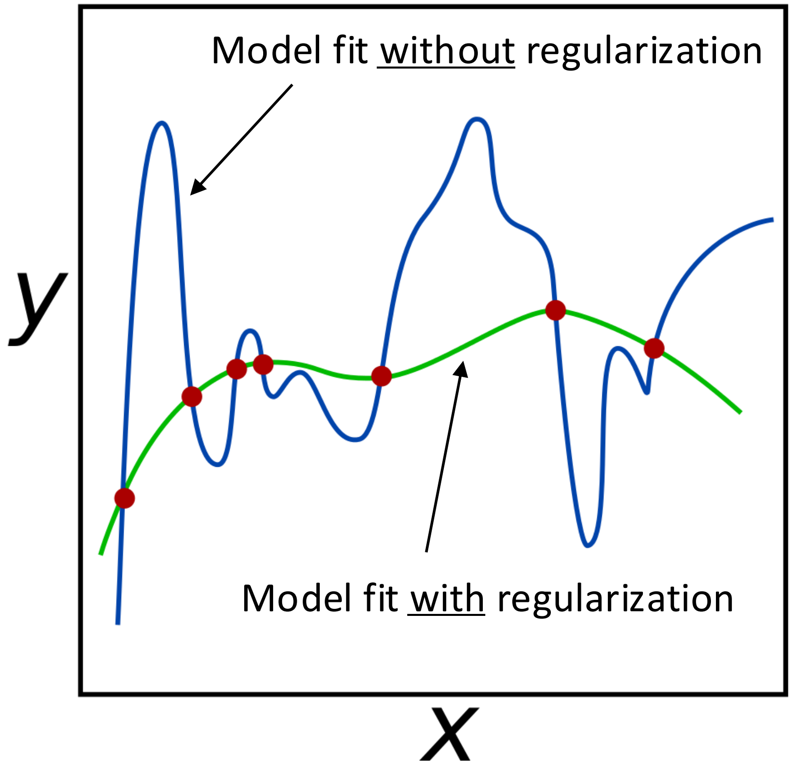
Weight decay

- The principle is simple: We extend our usual data loss $J(w)$ with an extra term

$$J_{reg}(w) = J(w) + \lambda R(w)$$

  where $R(w)$ is either the L1-norm or the L2-norm of w.

- The last term as called the *regularization term*, and λ is the regularization parameter.
- λ determines the trade-off between minimizing the data loss and minimizing the model parameters w.
- By keeping the weights small, the regularization term makes the model simpler, thereby **avoiding overfitting**.
- So, weight decay pushes against fitting the data *too* well so **we don't fit noise in the data**.


Model fit without regularization
y
Model fit with regularization
x

Example – simple linear equation

- Suppose our task is to estimate the parameters **w** of a straight line

$$y = h_w(x) = w_1 x + w_2$$

- You are given a set of training examples: $\{y^{(i)}, x^{(i)}\}_{i=1}^{n}$
- The usual **data loss** would be

$$J(w) = \sum_{i=1}^{n} \left(y^{(i)} - (w_1 x^{(i)} + w_2)\right)^2$$

- Thus, the **regularized loss** becomes

$$J_{reg}(w) = J(w) + \lambda R(w) = \sum_{i=1}^{n} \left(y^{(i)} - (w_1 x^{(i)} + w_2)\right)^2 + \lambda R(w)$$

$R(w)$ is either

$$L_2 = \sum_i {w_i}^2$$

or

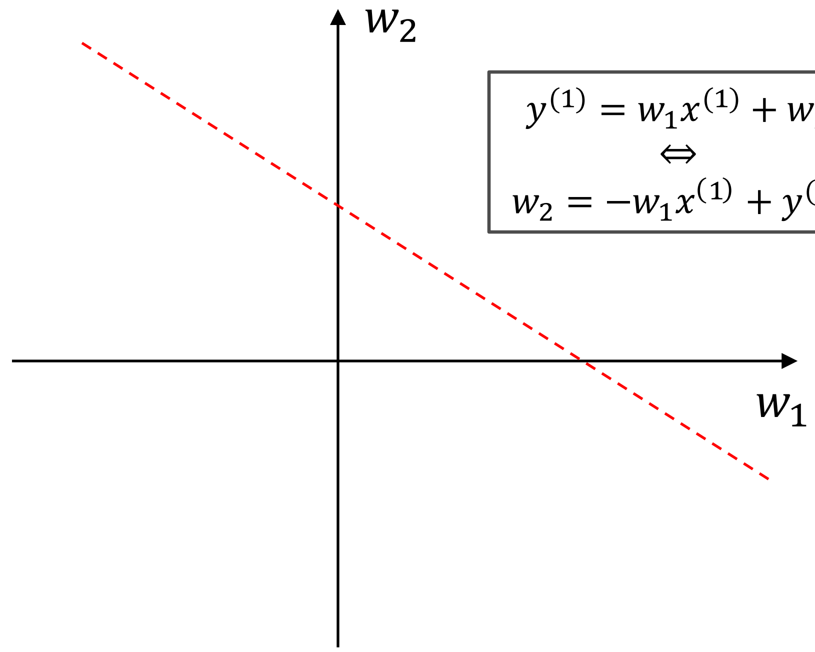
$$L_1 = \sum_i |w_i|$$

L1 vs L2

- The difference between L1 and L2 regularization can be explained by the problem of fitting a straight line

$$y = w_1 x + w_2$$

- To explain the difference between L1 and L2 regularization we will make use of a simple example, where we are just given one training pair: $\left(y^{(1)}, x^{(1)}\right)$
- With just one training pair, our problem is under-determined because we have two unknowns (w_1 and w_2), but only one equation: $y^{(1)} = w_1 x^{(1)} + w_2$
- There are infinitely exact solutions.
- The exact solutions lie on a straight line in the (w_1, w_2) plane, determined by $\left(y^{(1)}, x^{(1)}\right)$.



$$y^{(1)} = w_1 x^{(1)} + w_2 \\ \Leftrightarrow \\ w_2 = -w_1 x^{(1)} + y^{(1)}$$

For a single training pair $\left(y^{(1)}, x^{(1)}\right)$ the infinitely many exact solutions lie on a straight line in the (w_1, w_2)-plane.

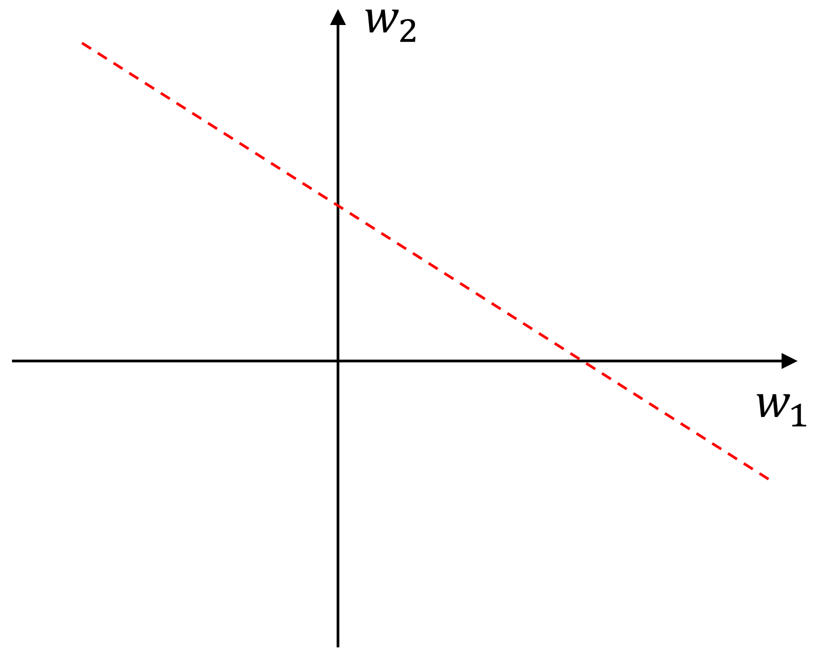
L1 vs L2

- So, there are infinitely many exact solutions.
- Now, we will search for an exact solution (w_1, w_2) that minimizes the regularized loss function

$$J_{reg}(w) = \underbrace{\left(y^{(1)} - (w_1 x^{(1)} + w_2)\right)^2}_{\text{Data loss}} + \underbrace{\lambda R(w)}_{\text{Regularization}}$$

where R(w) is either the L1-norm or the L2-norm of **w**.

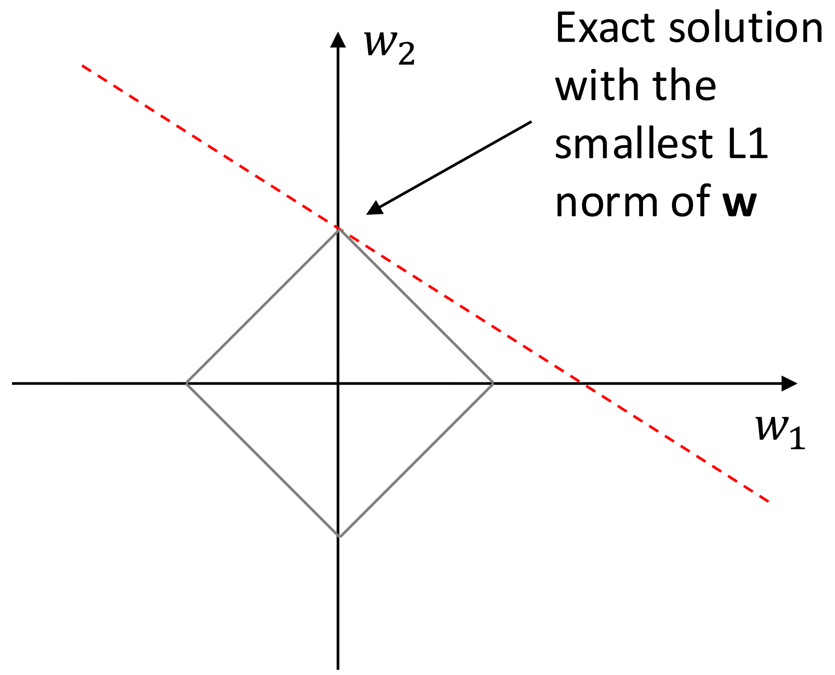
- In other words, we will search for a solution (w_1, w_2) that has a data loss of zero (exact solution) and a minimal norm in the L1- or L2-sense.
- The above loss function always has a unique solution for L2, and for L1 the solution is unique except in a few rare special cases.



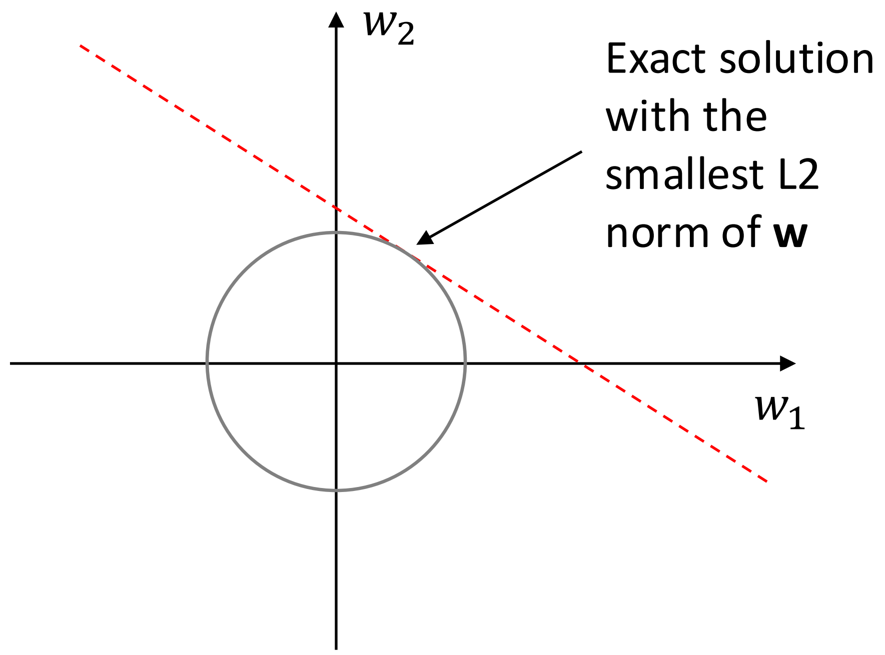
For a single training pair $(y^{(1)}, x^{(1)})$ the infinitely many exact solutions lie on a straight line in the (w_1, w_2)-plane.

L1 vs L2

L1 norm: $\sum |w_i|$



L2 norm: $\sum (w_i)^2$

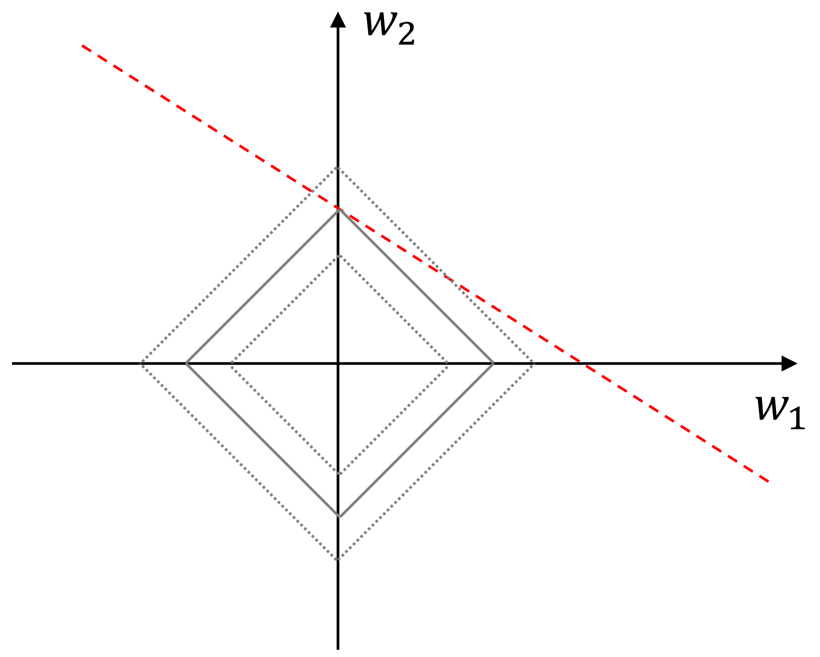


- Using L1 regularization tends to lead to **sparse solutions**, i.e., where many entries of **w** are zero.
- It's actually a kind of **feature selection**.
- Solution depends on choice of coordinate system.

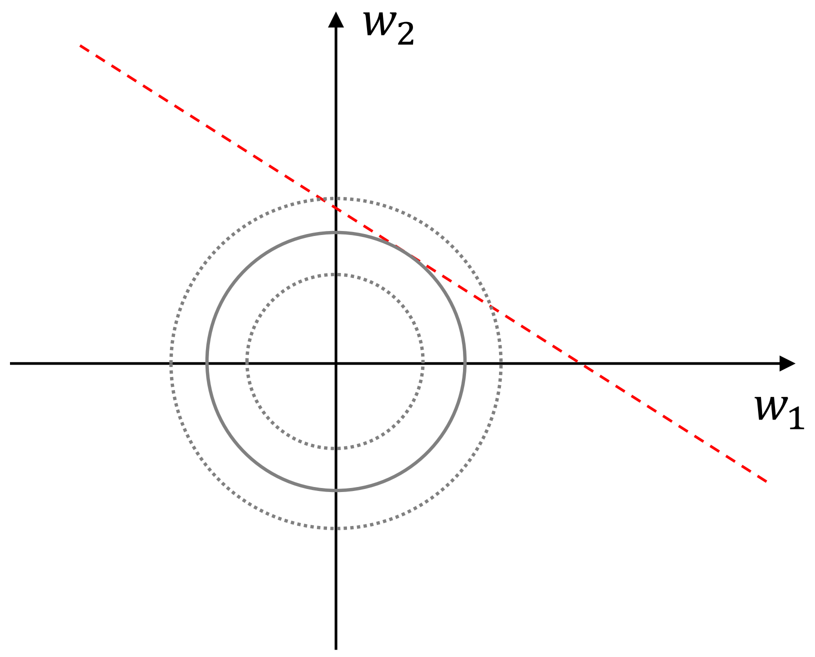
- L2 regularization has an analytical solution and is computationally faster than L1 regularization.
- L2 regularization is also called weight decay
- The solution will be non-sparse and does not depend on the choice of coordinate system.

L1/L2 plots explained

L1 norm: $\sum|w_i|$



L2 norm: $\sum(w_i)^2$



- The points on the grey L1/L2 contours are equidistant to the origin according to the chosen norm.
- To find the exact solution that minimizes the cost **J(w)**, simply expand or shrink the contour until it intersects only in one point with the red line of exact solutions.
- Dashed contours either don’t intersect with the red line, or they intersect with it in two places.

L1 and L2 have different preferences

- Example – suppose we want to find w that solves $w^T x = 1$, where $x = [1, 1, 1, 1]$.
- The regularized loss would be

$$J(w) = \left(1 - \sum_{i=1}^{4} w_i x_i\right)^2 + \lambda R(w)$$

- With L1 regularization, we get one of the following solutions:

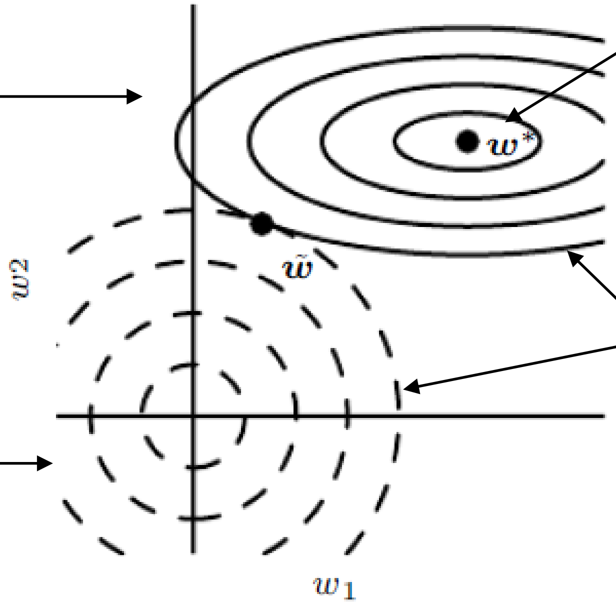
$$w = [1, 0, 0, 0], w = [0, 1, 0, 0] , w = [0, 0, 1, 0], \text{ or } w = [0, 0, 0, 1]$$

- Whereas for L2 regularization, we get a unique solution: $w = [0.25, 0.25, 0.25, 0.25]$
- Why? Because L1 prefers many zero-weights, whereas L2 prefers to “spread out” the weights.

General case

- In general, the plots we saw earlier look more like this:

$$J_{reg}(w) = \underbrace{\left(y^{(1)} - (w_1 x^{(1)} + w_2)\right)^2}_{\text{Data loss}} + \underbrace{\lambda R(w)}_{\text{Regularization}}$$



Solution with smallest possible data loss.

Contours represent candidate solutions with equal value of the data loss or L2 loss.

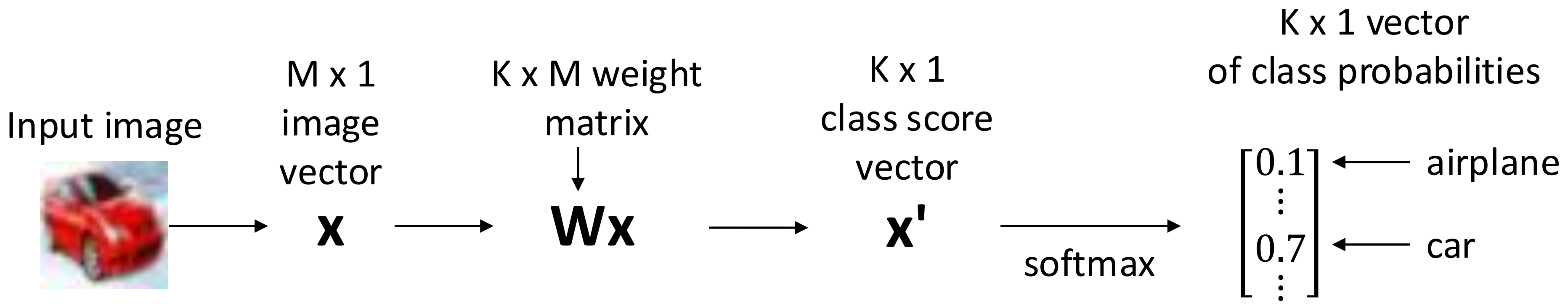
Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the

Softmax regression

MULTINOMIAL LOGISTIC REGRESSION

How do we handle multiple classes?

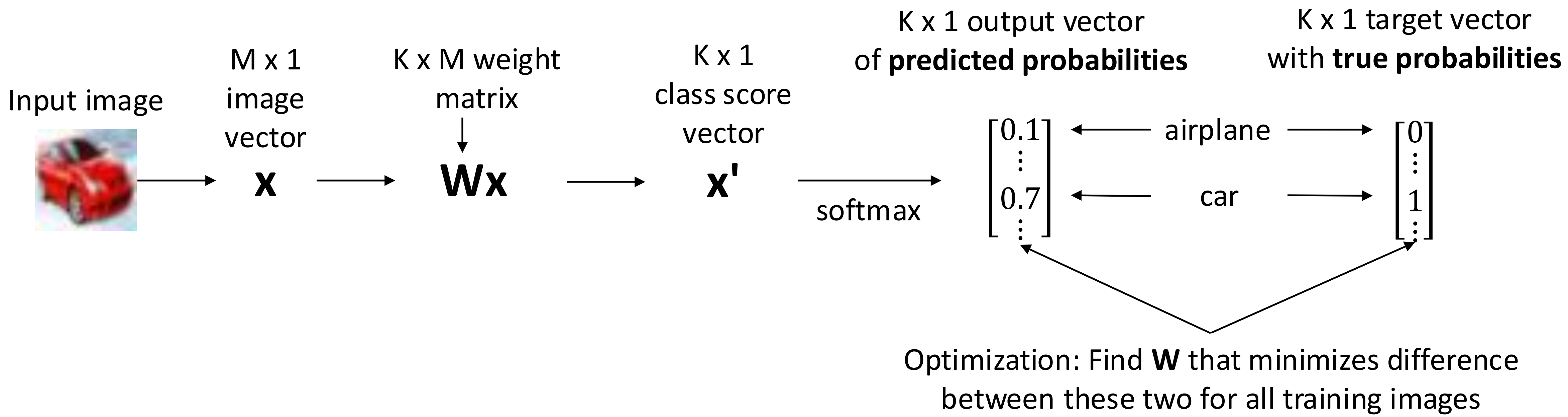
- What if we have more than two classes?
- Logistic regression would be possible via a one-vs-all approach (e.g., “car” vs “not car”).
- But training multiple binary classifiers seems impractical.
- **Idea:** Calculate a vector of **class probabilities**
- How? Softmax regression



Note that to use Softmax, K must equal the number of classes (i.e., out vector has one entry per class).

How do we handle multiple classes?

- Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes.
- Search for weights (**W**) that minimize the difference between:
  - 1) the output vector of **predicted probabilities**
  - 2) the target vector of **true probabilities** (called a one-hot vector)



Input image
M x 1 image vector
x
K x M weight matrix
Wx
K x 1 class score vector
x'
softmax
K x 1 output vector of **predicted probabilities**
0.1
0.7
airplane
car
K x 1 target vector with **true probabilities**
0
1
Optimization: Find **W** that minimizes difference between these two for all training images

From logistic regression to softmax

- In logistic regression we assumed that the labels were binary: $y^{(i)} \in \{0,1\}$.
- Softmax regression allows us to handle K classes. Thus, our training set is

$$\{y^{(i)}, x^{(i)}\}_{i=1}^{n}, \text{ where } y^{(i)} \in \{1, 2, \dots, K\}$$

- Given a test input $x \in \mathbb{R}^{m\times 1}$, we want our model to estimate the probability

$$P(y = k|x) \text{ for each } \mathrm{k} = 1, 2, \dots, K$$

- Thus, the output will be a K-dimensional vector of probabilities.

Hypothesis

- Our model $h_W(x)$ takes the form

$$h_W(x) = \begin{bmatrix} P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=K|x) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp\left(w_j^T x\right)} \begin{bmatrix} \exp(w_1^T x) \\ \exp(w_2^T x) \\ \vdots \\ \exp(w_K^T x) \end{bmatrix}$$

- where the matrix $W \in \mathbb{R}^{m \times K}$ holds the parameters of our model:

$$W = \begin{bmatrix} | & | & | & | \\ w_1 & w_2 & \cdots & w_K \\ | & | & | & | \end{bmatrix}$$

- Notice that we are using a separate “template” for each class: $w_1, w_2, \ldots, w_K$ (the reason for this will become clear when we try it on the CIFAR-10 dataset).

Hypothesis

- Our model $h_W(x)$ takes the form

$$h_W(x) = \begin{bmatrix} P(y = 1|x) \\ P(y = 2|x) \\ \vdots \\ P(y = K|x) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp\left(w_j^T x\right)} \begin{bmatrix} \exp(w_1^T x) \\ \exp(w_2^T x) \\ \vdots \\ \exp(w_K^T x) \end{bmatrix}$$

Q: Why use $\exp\left(w_j^T x\right)$ and not just $w_j^T x$?

Hypothesis

- Our model $h_W(x)$ takes the form

$$h_W(x) = \begin{bmatrix} P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=K|x) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp\left(w_j^T x\right)} \begin{bmatrix} \exp(w_1^T x) \\ \exp(w_2^T x) \\ \vdots \\ \exp(w_K^T x) \end{bmatrix}$$

Q: Why use $\exp\left(w_j^T x\right)$ and not just $w_j^T x$?

A: Probabilities must be positive numbers (or zero). Applying $\exp$ ensures that numbers are positive.

Hypothesis

- Our model $h_W(x)$ takes the form

$$h_W(x) = \begin{bmatrix} P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=K|x) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp\left(w_j^T x\right)} \begin{bmatrix} \exp(w_1^T x) \\ \exp(w_2^T x) \\ \vdots \\ \exp(w_K^T x) \end{bmatrix}$$

Q: What is the purpose of the term $\sum_{j=1}^{K} \exp\left(w_j^T x\right)$?

Hypothesis

- Our model $h_W(x)$ takes the form

$$h_W(x) = \begin{bmatrix} P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=K|x) \end{bmatrix} = \underbrace{\frac{1}{\sum_{j=1}^{K} \exp\left(w_j^T x\right)}}_{\uparrow} \begin{bmatrix} \exp(w_1^T x) \\ \exp(w_2^T x) \\ \vdots \\ \exp(w_K^T x) \end{bmatrix}$$

Q: What is the purpose of the term $\sum_{j=1}^{K} \exp\left(w_j^T x\right)$?

A: It normalizes the probability distribution, so that it sums to 1.

Loss function

- In the equation below, $\mathbf{1}\{\cdot\}$ is the "indicator function," so

$$\mathbf{1}\{\text{true statement}\} = 1$$
$$\mathbf{1}\{\text{false statement}\} = 0$$

- For example, $\mathbf{1}\{2 + 2 = 4\}$ evaluates to 1; whereas $\mathbf{1}\{1 + 1 = 5\}$ evaluates to 0.
- Our loss function will be:

$$J(W) = -\sum_{i=1}^{n}\sum_{k=1}^{K} \mathbf{1}\left\{y^{(i)} = k\right\} \log\left(\frac{\exp\left(w_k^T x^{(i)}\right)}{\sum_{j=1}^{K} \exp\left(w_j^T x^{(i)}\right)}\right)$$

- Notice that because of the indicator function only one term in the inner sum will be non-zero for each training example (outer sum).

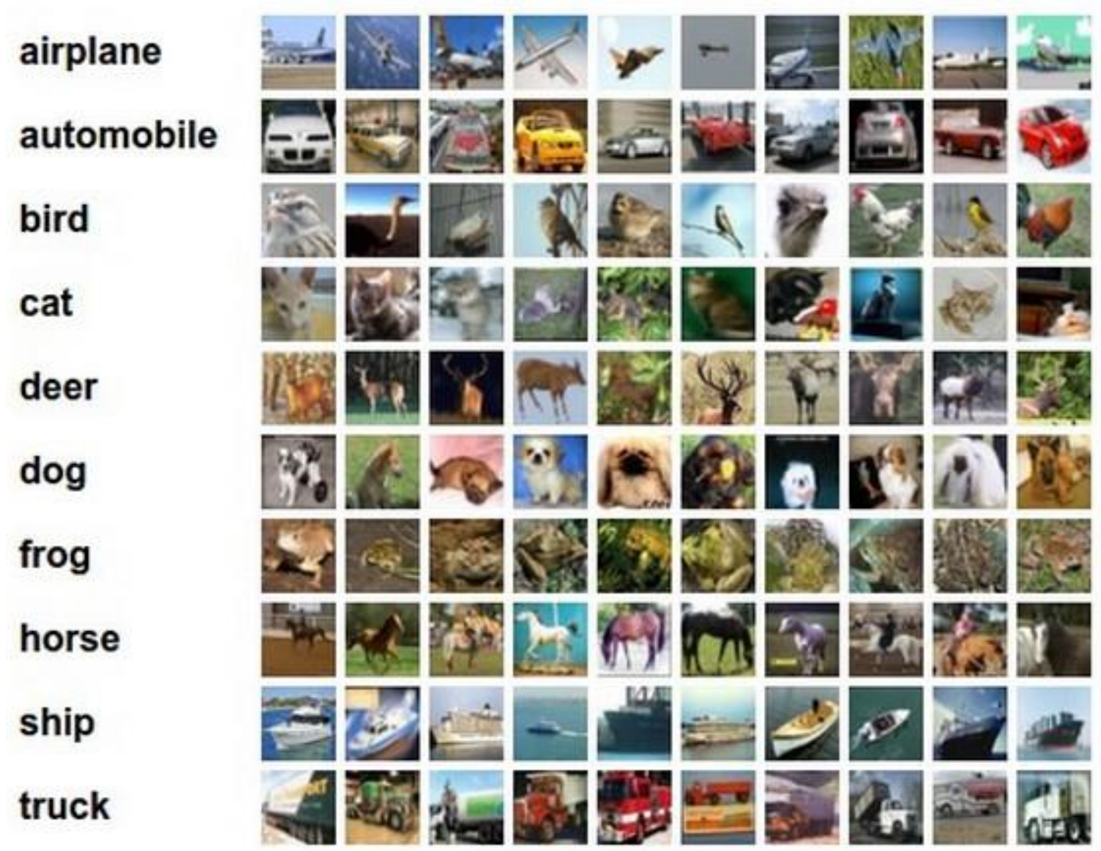
Optimization

- We cannot solve for the minimum of $J(w)$ analytically, and thus as usual we'll resort to an iterative optimization algorithm.
- Taking derivatives, one can show that the gradient of w_k is:

$$\nabla J(w_k) = -\sum_{i=1}^{n} \left[x^{(i)} \left(\mathbf{1}\left\{ y^{(i)} = k \right\} - P\left(y^{(i)} = k \middle| x^{(i)} \right) \right) \right] \text{ for each } \mathrm{k} = 1, 2, \dots, K$$

- $\nabla J(w_k)$ is itself a vector, and it has to be calculated for all k.
- Armed with this formula for the derivative, one can then plug it into a standard optimization package and have it minimize $J(W)$.

Example dataset: CIFAR10

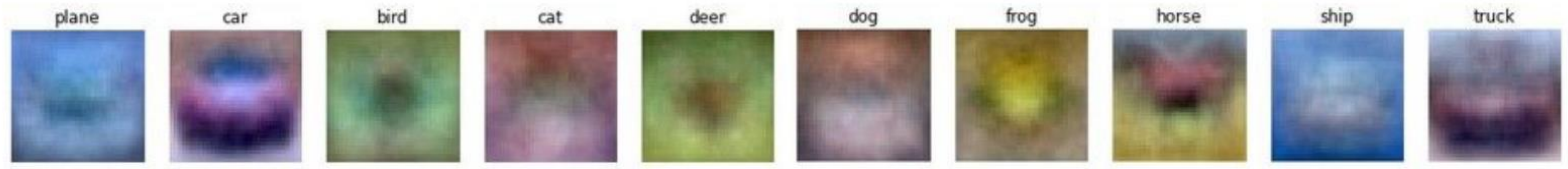


- 10 classes
- 50,000 training images (5,000 for each label)
- 10,000 test images
- Image size: 32 x 32 x 3

https://www.cs.toronto.edu/~kriz/cifar.html

Example dataset: CIFAR10

- We now have all the building blocks needed in order to train our linear (multi-class) classifier



Rows of **W**

Input image

M x 1 image vector

K x M weight matrix

K x 1 class score vector

K x 1 output vector of **predicted probabilities**

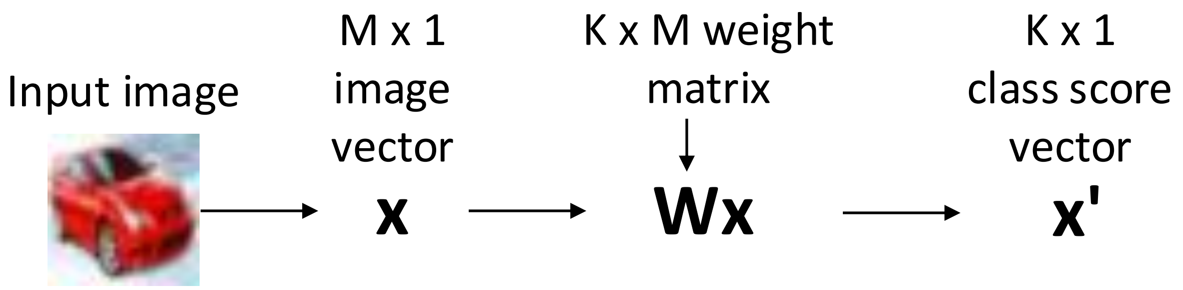
K x 1 target vector with **true probabilities**

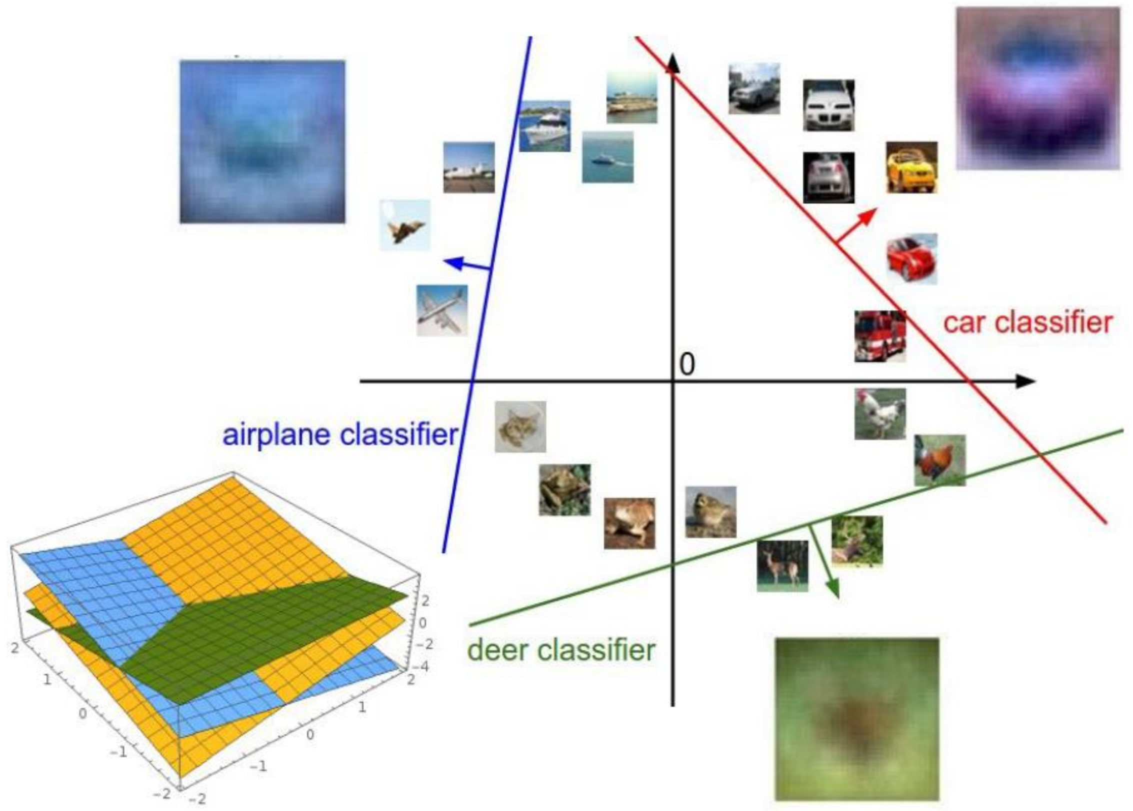
$$\begin{tikzcd} \text{Input image} \arrow[r] & \mathbf{x} \arrow[r] & \mathbf{Wx} \arrow[r] & \mathbf{x'} \arrow[r, "\text{softmax}"'] & \begin{bmatrix} 0.1 \\ \vdots \\ 0.7 \\ \vdots \end{bmatrix} & \begin{matrix} \text{airplane} \\ \\ \text{car} \end{matrix} \arrow[l] \arrow[r] & \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} \end{tikzcd}$$

Decision boundaries

Each row of **W** represents one class and gives rise to one linear decision boundary (e.g. car vs not car).

Problem: Unlikely that the classes can be separated by linear decision boundaries.

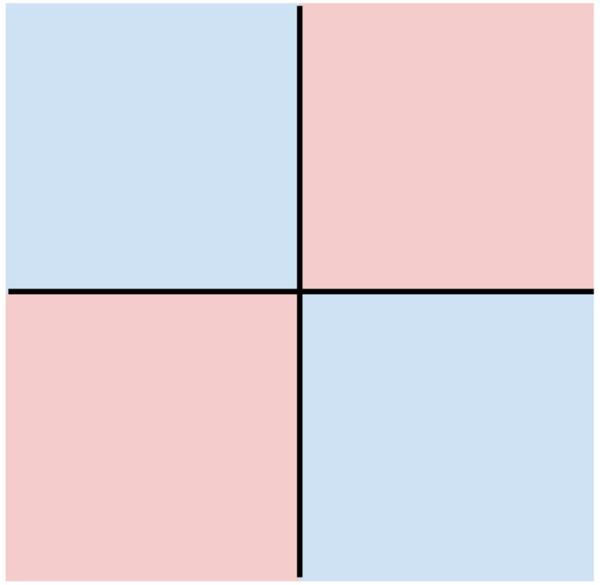




Hard cases for a linear classifier

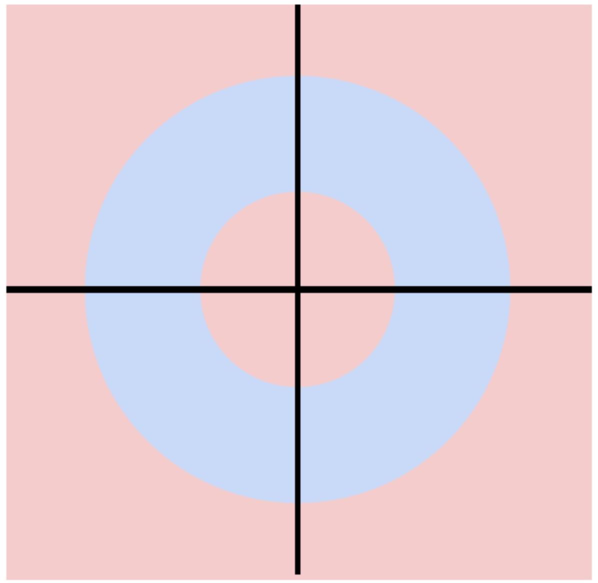
Class 1:
First and third quadrants

Class 2:
Second and fourth quadrants



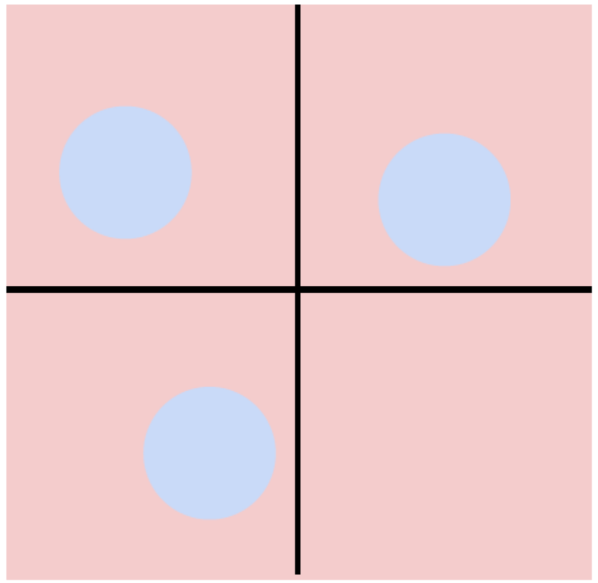
Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



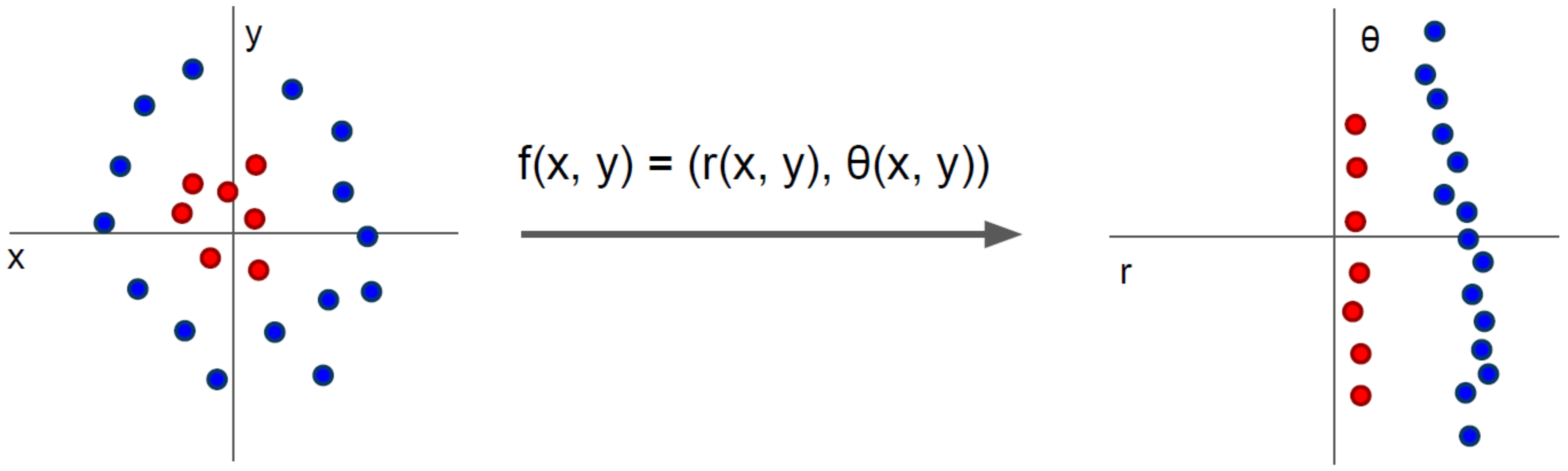
Class 1:
Three modes

Class 2:
Everything else



Solution: Non-linear feature transformation

Example – Convert Cartesian to polar coordinates

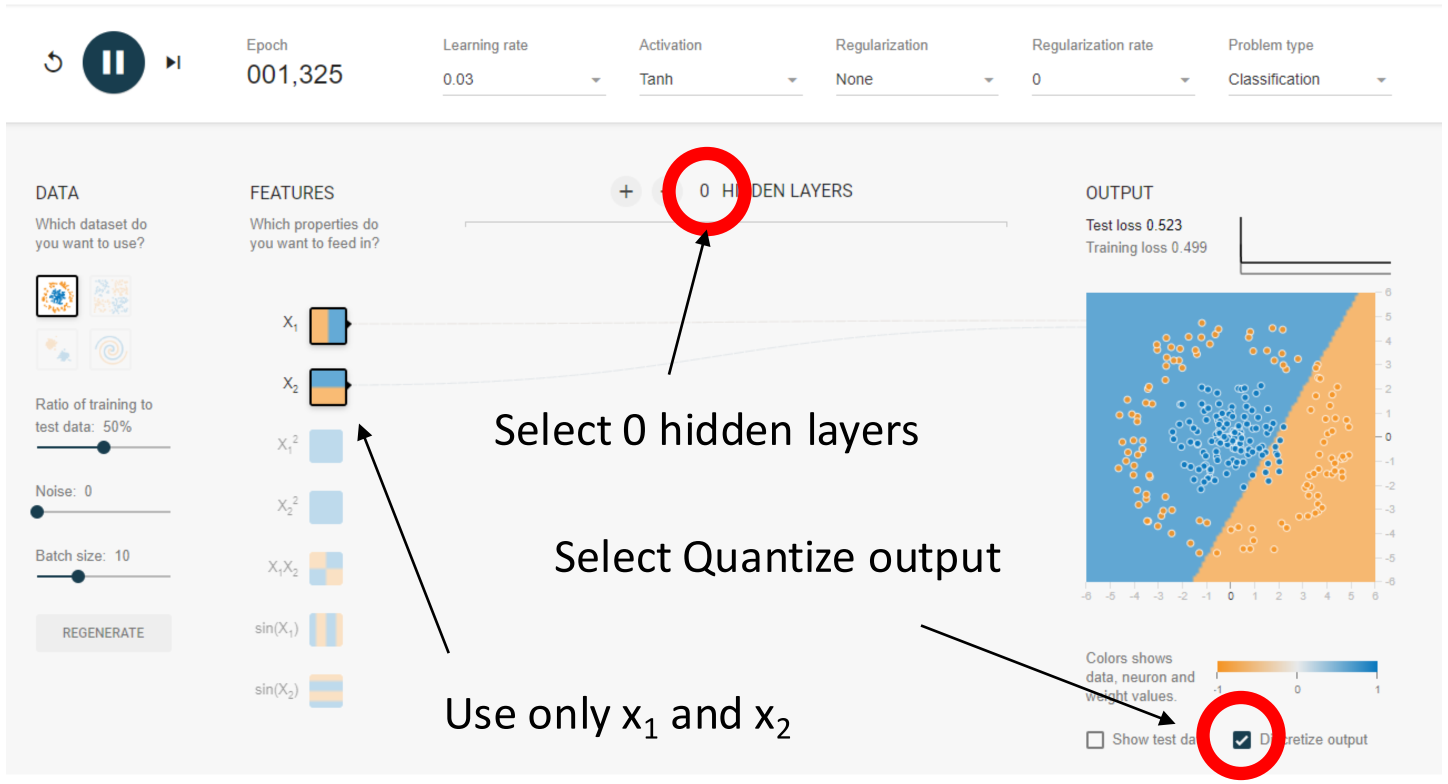


Cannot separate red and blue points with linear classifier

After applying feature transform, points can be separated by linear classifier

Solution: Neural networks

- **Problem:** We can’t solve the hard cases with softmax regression, because the underlying transformation is still linear.
- If you don’t believe me, try it out yourself here: https://playground.tensorflow.org/
- **Solution:** Neural networks



Epoch 001,325

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1, X_2, X_1^2, X_2^2, X_1X_2, $\sin(X_1)$, $\sin(X_2)$

0 HI DEN LAYERS

OUTPUT

Test loss 0.523

Training loss 0.499

Colors shows data, neuron and weight values.

Show test da D retize output

Select 0 hidden layers

Select Quantize output

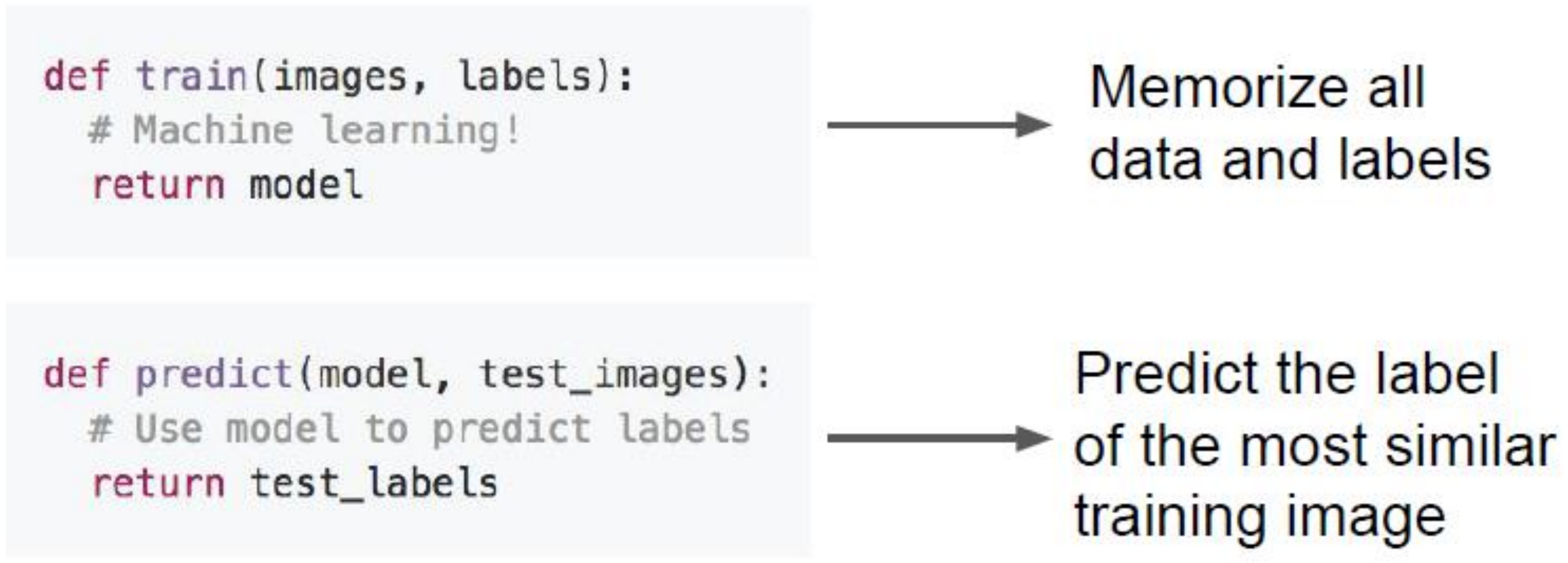
Use only x_1 and x_2

Preparation for Lab 2

Image classification pipeline

- **Task:** Take an array of pixels that represents a single image and assign a label to it.
- **Input:** Our input consists of a set of *N* images, each labelled with one of *M* different classes. We refer to this data as the **training set**.
- **Learning:** Our task is to use the training set to learn what every one of the classes looks like. We refer to this step as **training a classifier** or *learning a model*.
- **Evaluation:** In the end, we evaluate the quality of the classifier by asking it to predict labels for a new set of images that it has never seen before. We will then compare the true labels of these images to the ones predicted by the classifier. Intuitively, we're hoping that a lot of the predictions match up with the true answers (which we call the **ground truth**).

Basic nearest neighbour classifier



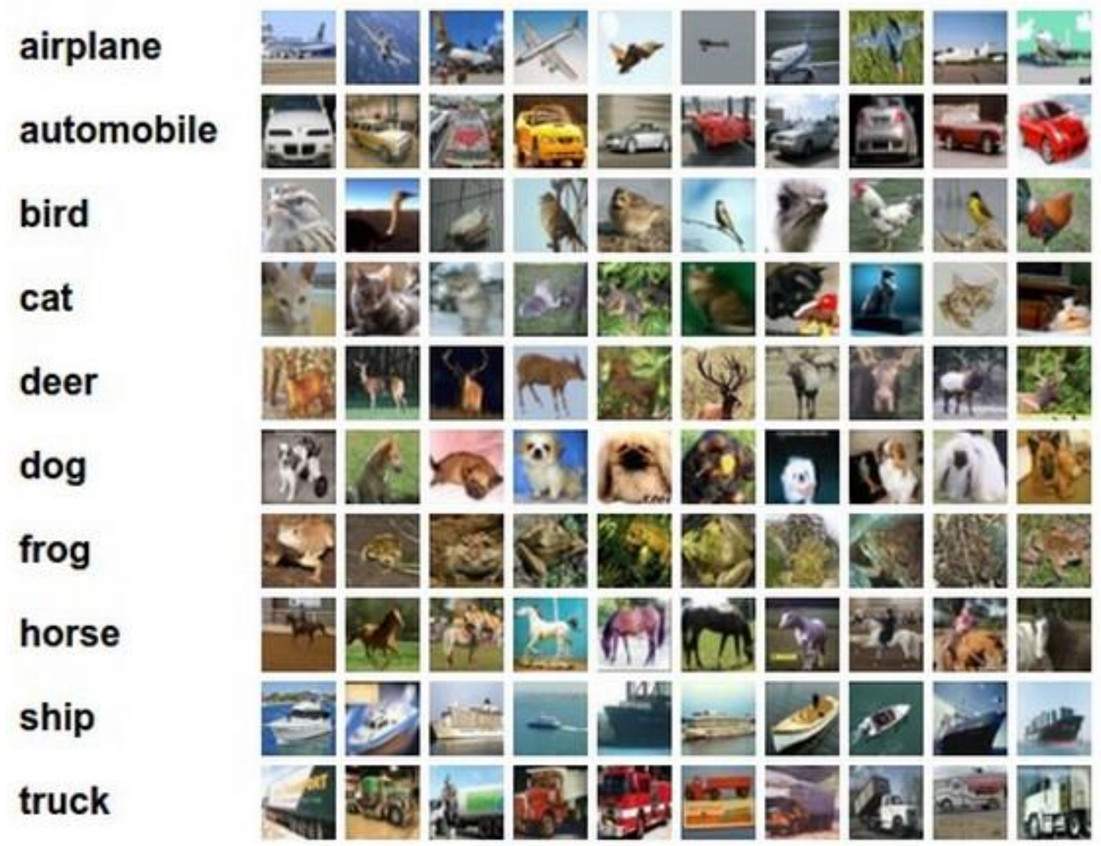
```python
def train(images, labels):
  # Machine learning!
  return model
```

Memorize all data and labels

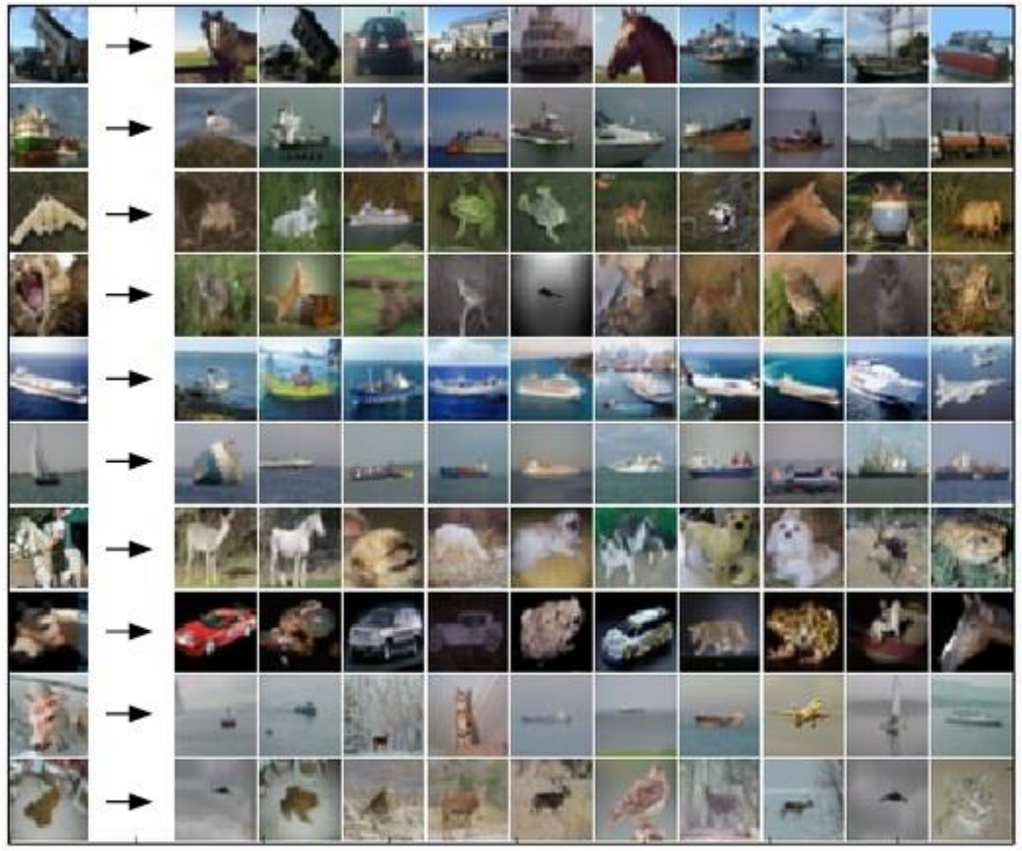
```python
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

Predict the label of the most similar training image

Example dataset: CIFAR10



Examples of test images and 10 nearest neighbours



Distance metrics to compare images

L2 norm

$$L_2 = \sum_p (I_1^p - I_2^p)^2$$

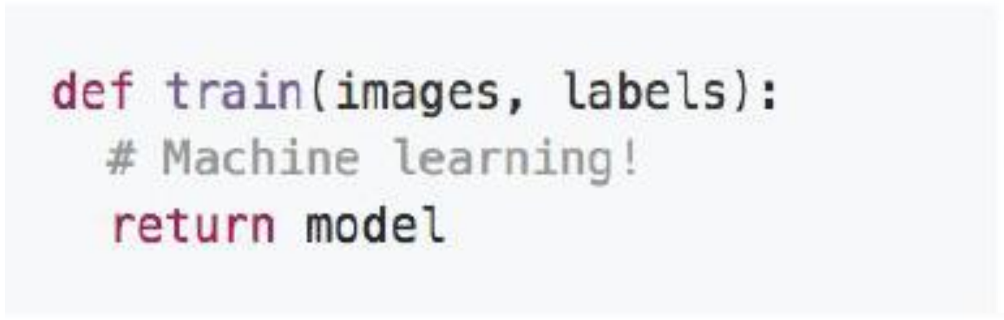
$$\begin{array}{c}\text{Input image}\\ \begin{array}{|c|c|c|c|}\hline 56 & 32 & 10 & 18\\ \hline 90 & 23 & 128 & 133\\ \hline 24 & 26 & 178 & 200\\ \hline 2 & 0 & 255 & 220\\ \hline\end{array}\end{array} - \begin{array}{c}\text{Training image}\\ \begin{array}{|c|c|c|c|}\hline 10 & 20 & 24 & 17\\ \hline 8 & 10 & 89 & 100\\ \hline 12 & 16 & 178 & 170\\ \hline 4 & 32 & 233 & 112\\ \hline\end{array}\end{array} = \begin{array}{|c|c|c|c|}\hline 46 & 12 & -14 & -1\\ \hline 82 & 13 & 39 & 33\\ \hline 12 & 10 & 0 & 30\\ \hline -2 & 32 & 22 & 108\\ \hline\end{array} \xrightarrow{\text{square and add}} 26{,}280$$

L1 norm

$$L_1 = \sum_p \left| I_1^p - I_2^p \right|$$

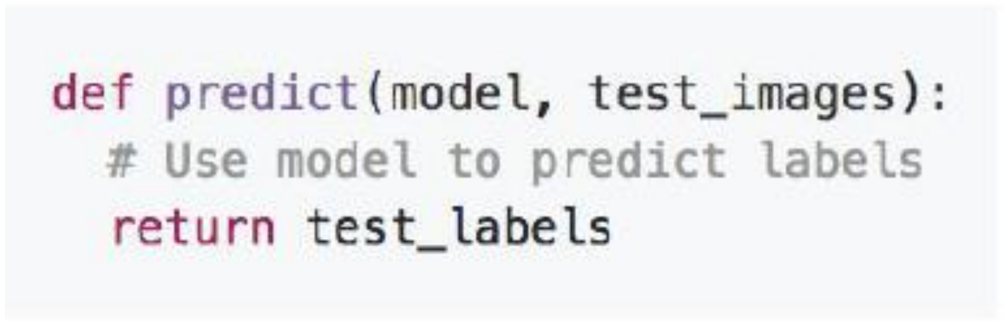
$$\left| \begin{array}{c}\text{Input image}\\ \begin{array}{|c|c|c|c|}\hline 56 & 32 & 10 & 18\\ \hline 90 & 23 & 128 & 133\\ \hline 24 & 26 & 178 & 200\\ \hline 2 & 0 & 255 & 220\\ \hline\end{array}\end{array} - \begin{array}{c}\text{Training image}\\ \begin{array}{|c|c|c|c|}\hline 10 & 20 & 24 & 17\\ \hline 8 & 10 & 89 & 100\\ \hline 12 & 16 & 178 & 170\\ \hline 4 & 32 & 233 & 112\\ \hline\end{array}\end{array} \right| = \begin{array}{|c|c|c|c|}\hline 46 & 12 & 14 & 1\\ \hline 82 & 13 & 39 & 33\\ \hline 12 & 10 & 0 & 30\\ \hline 2 & 32 & 22 & 108\\ \hline\end{array} \xrightarrow{\text{add}} 456$$

Basic nearest neighbour classifier



```python
def train(images, labels):
  # Machine learning!
  return model
```

→ Store training images and corresponding labels

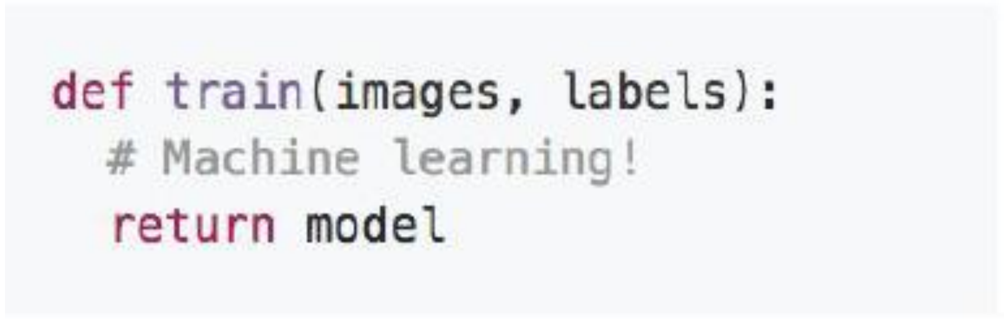


```python
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

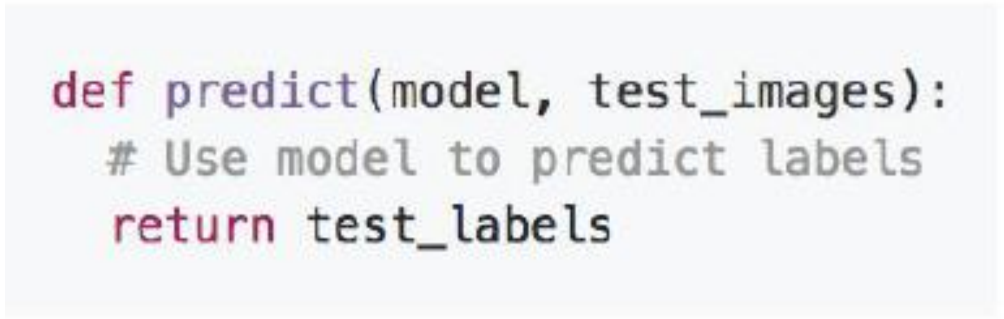
→ Given a test image:

1. Find nearest training image
2. Assign label of nearest image

Basic nearest neighbour classifier



```
def train(images, labels):
  # Machine learning!
  return model
```



```
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

Q: With N examples, how fast are training and prediction?

A:
train O(1)
predict O(N)

This is bad: we want classifiers that are **fast** at prediction; **slow** for training is ok.

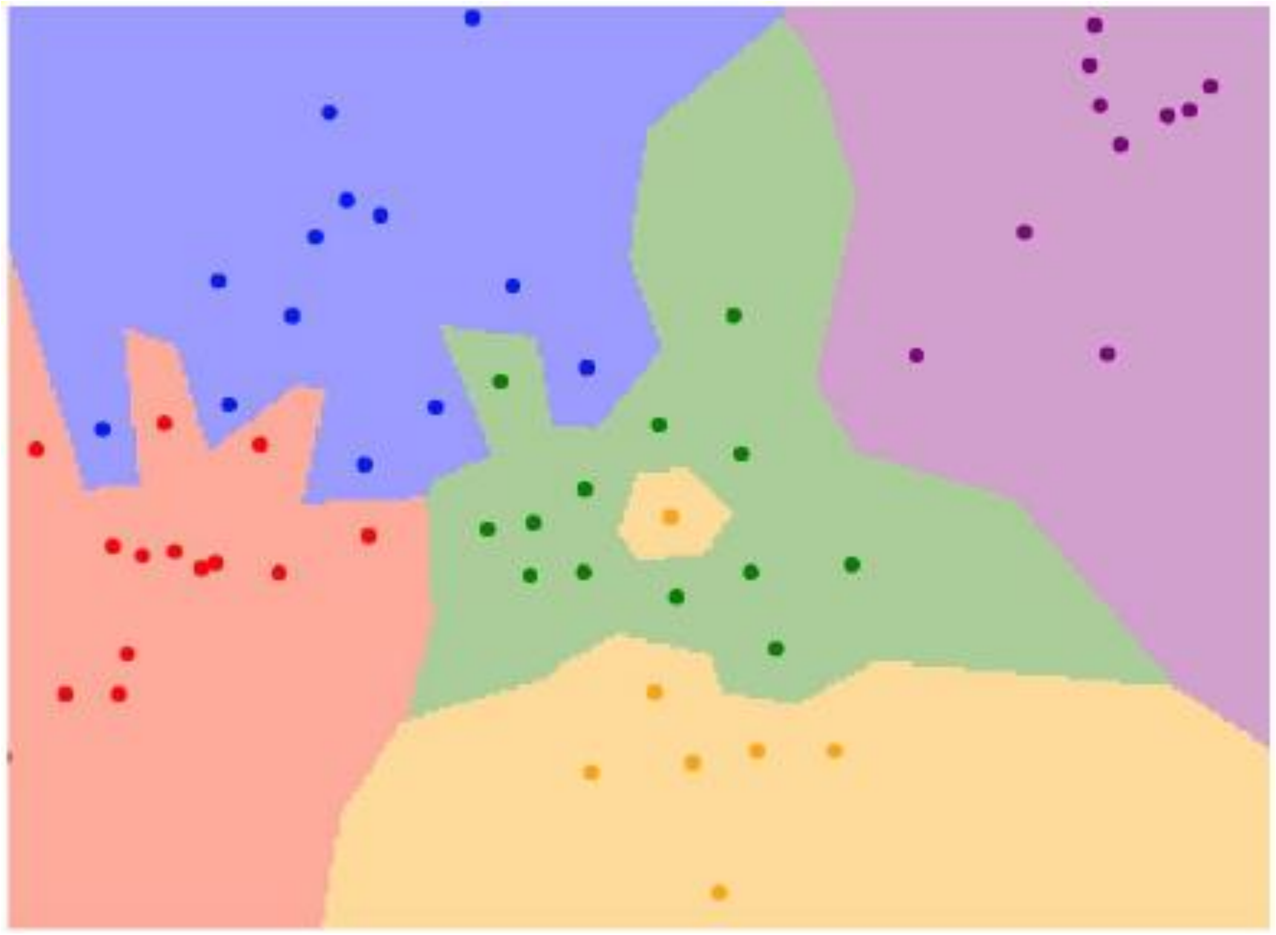
Note: Search can be made faster (look up KD-trees, for instance)

What does it look like?

Dots correspond to data points (observations)

Colors correspond to classes

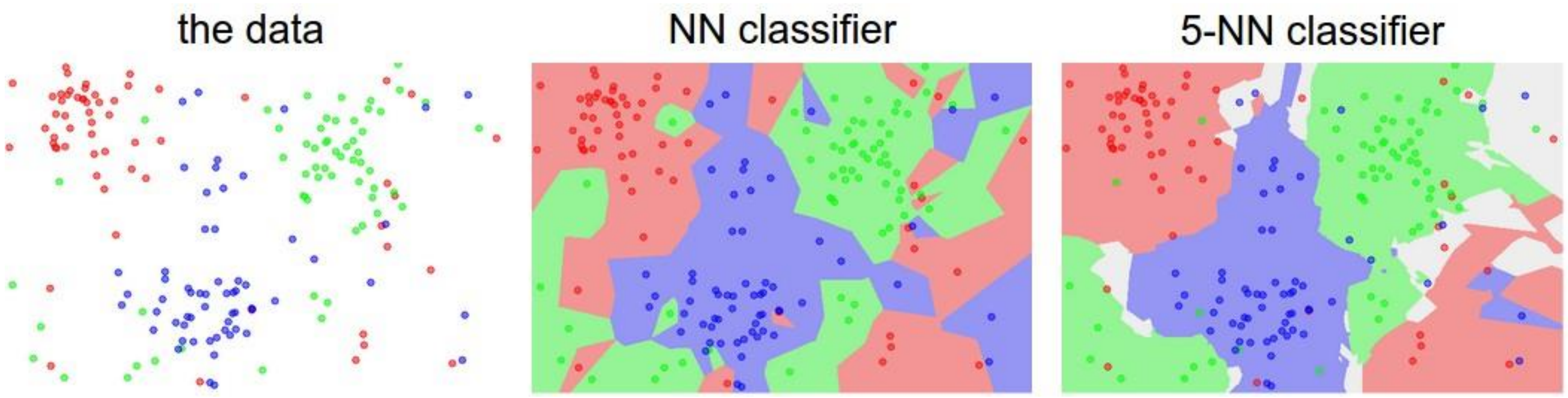
Colored regions mark decision boundaries



http://vision.stanford.edu/teaching/cs231n-demos/knn/

K-Nearest Neighbours

- Instead of copying label from nearest neighbour, take **majority vote** from K closest points
- Why?
- What are the white regions?



What are the hyperparameters of K-NN?

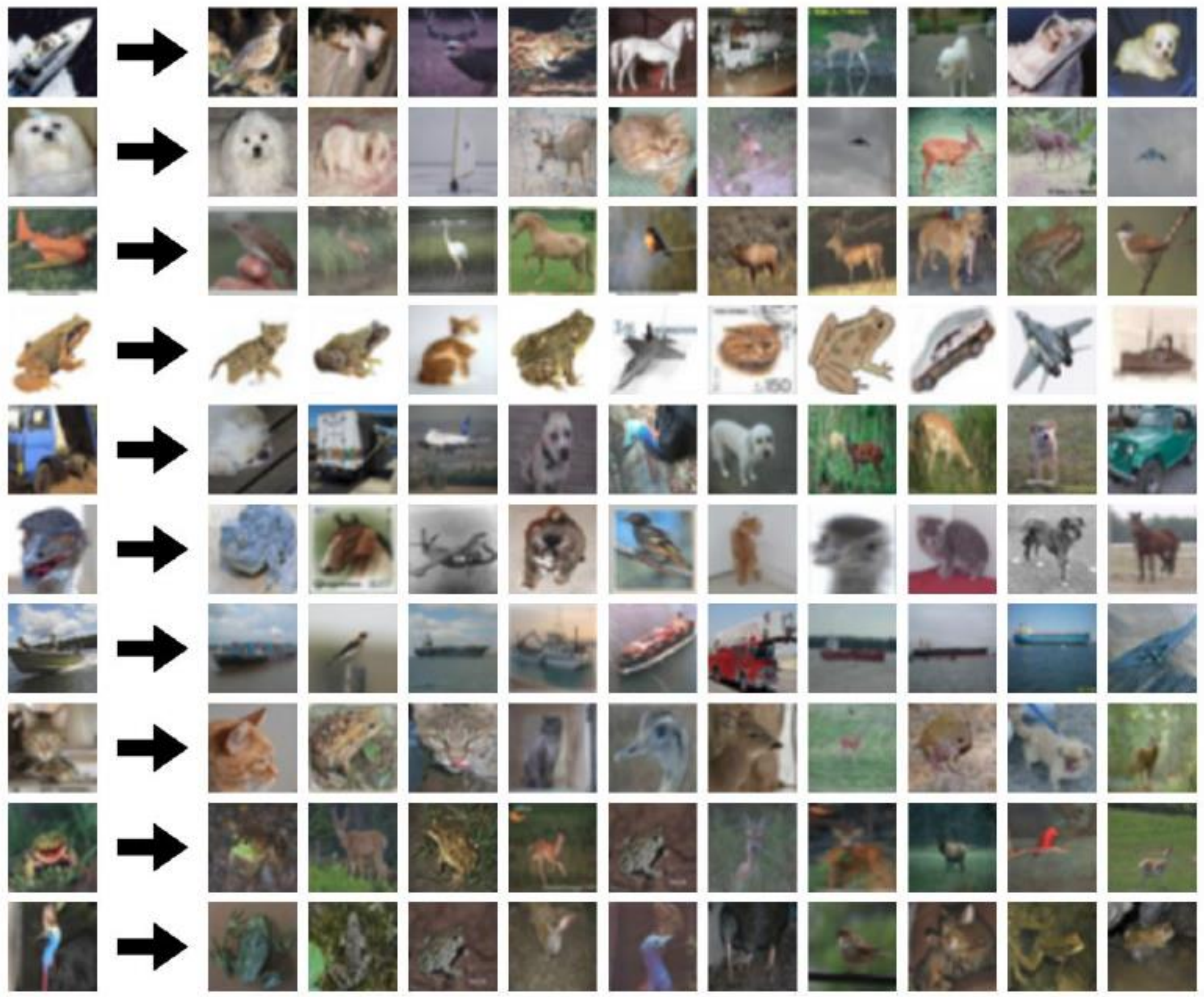
- What is the best value of **k**?
- What is the best **distance metric** to use?

- Very problem-dependant
- Must try them all out to see what works best

What does it look like?

For each input image (leftmost column), we find the 10 nearest neighbors.

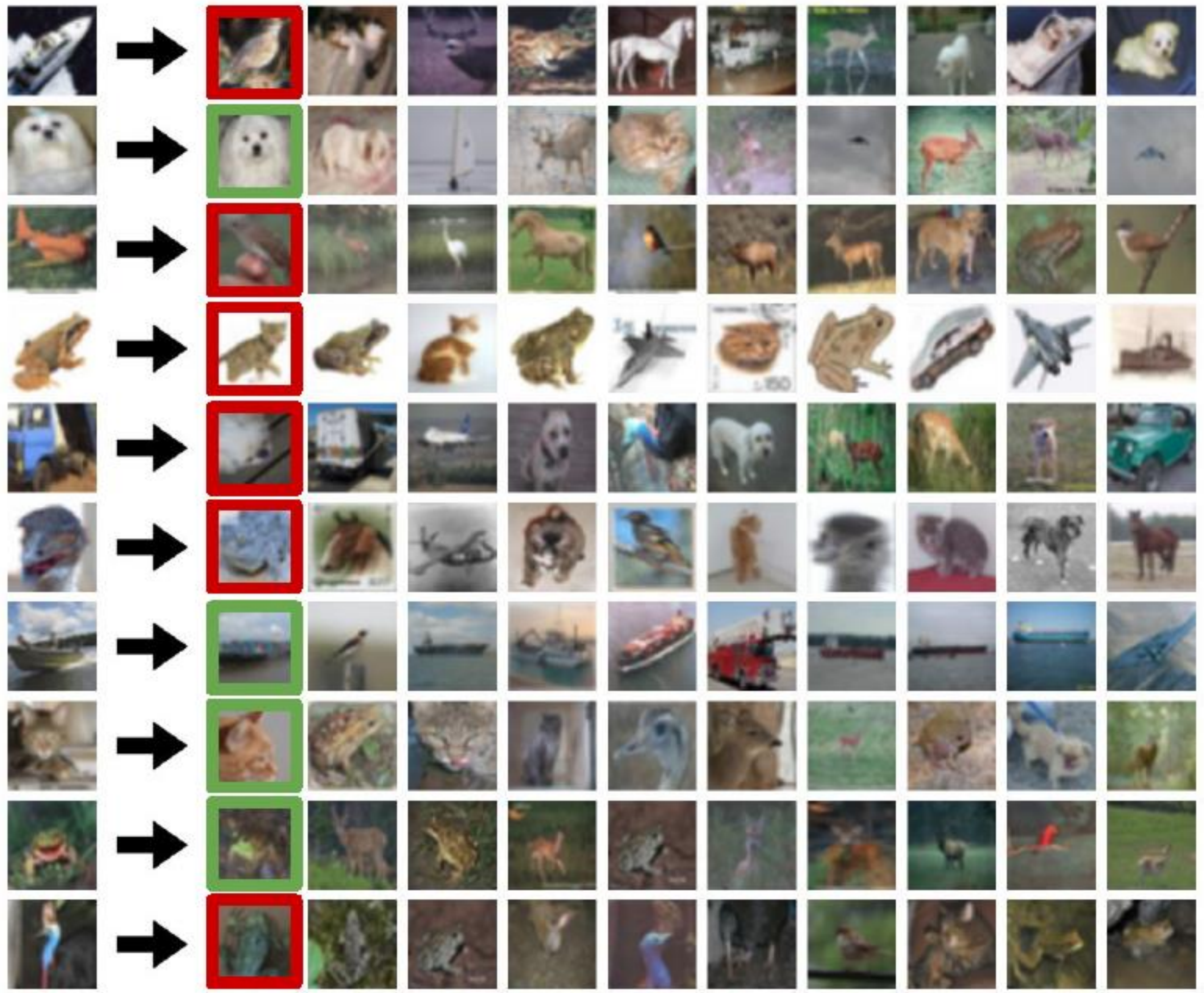
Images are compared using the L2 norm between the raw pixel values.



What does it look like?

For each input image (leftmost column), we find the 10 nearest neighbors.

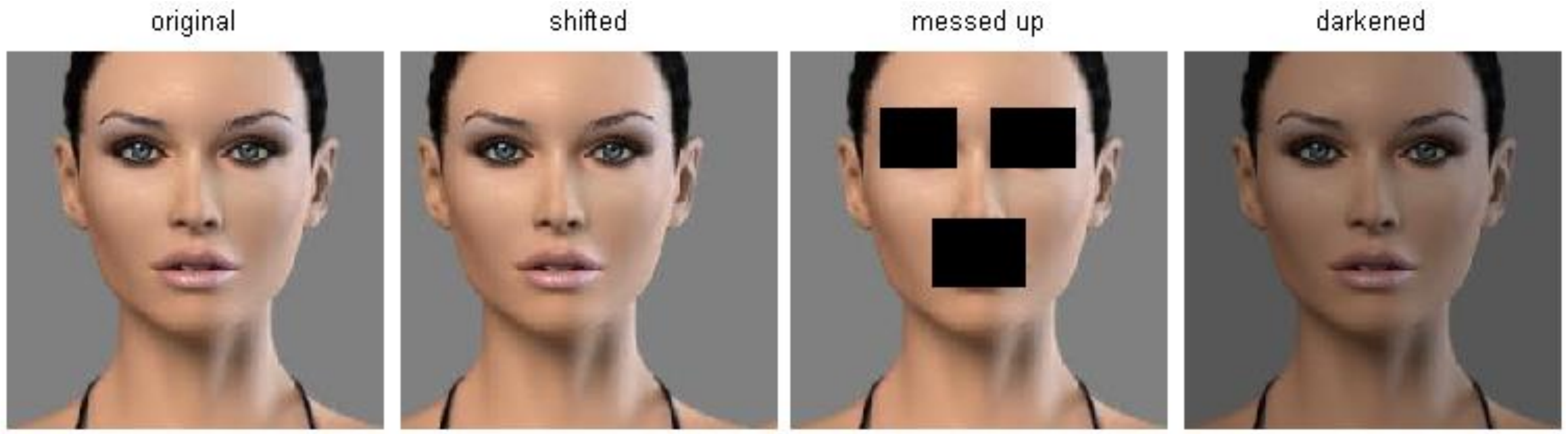
Images are compared using the L2 norm between the raw pixel values.



We see that using the raw pixels as our input doesn't work. The nearest neighbors are similar in terms of pixel values/colors, but they are not semantically similar.

Poor choice of features

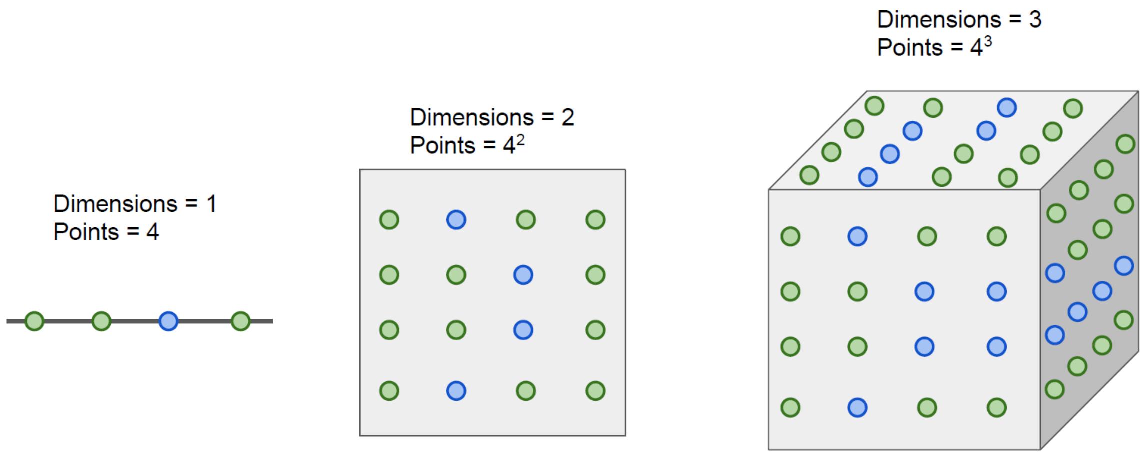
- K-NN is almost never used on raw pixel intensities
- Very slow at test time
- Distance metrics on pixels are not informative:



Pixel-based distances on high-dimensional data (and images especially) can be very unintuitive. An original image (left) and three other images next to it that are all equally far away from it based on L2 pixel distance. Clearly, the pixel-wise distance does not correspond at all to perceptual or semantic similarity.

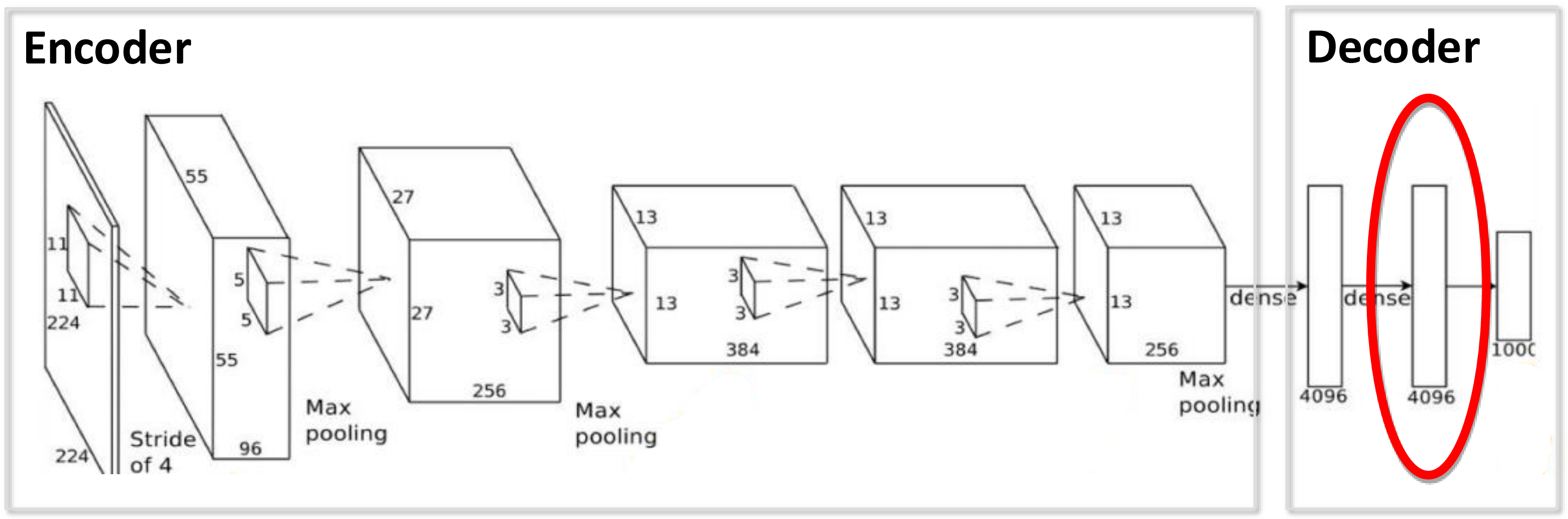
Curse of dimensionality

- Image classification based on K-NN relies on detecting areas in feature space where objects form groups with similar properties.
- In high dimensional data, however, all objects appear to be sparse and dissimilar in many ways, which is why image classification based on K-NN performs poorly when the features are the raw pixels.
- For CIFAR10 the feature space is 3072-dimensional
- Also, search becomes slower as dimensionality increases.

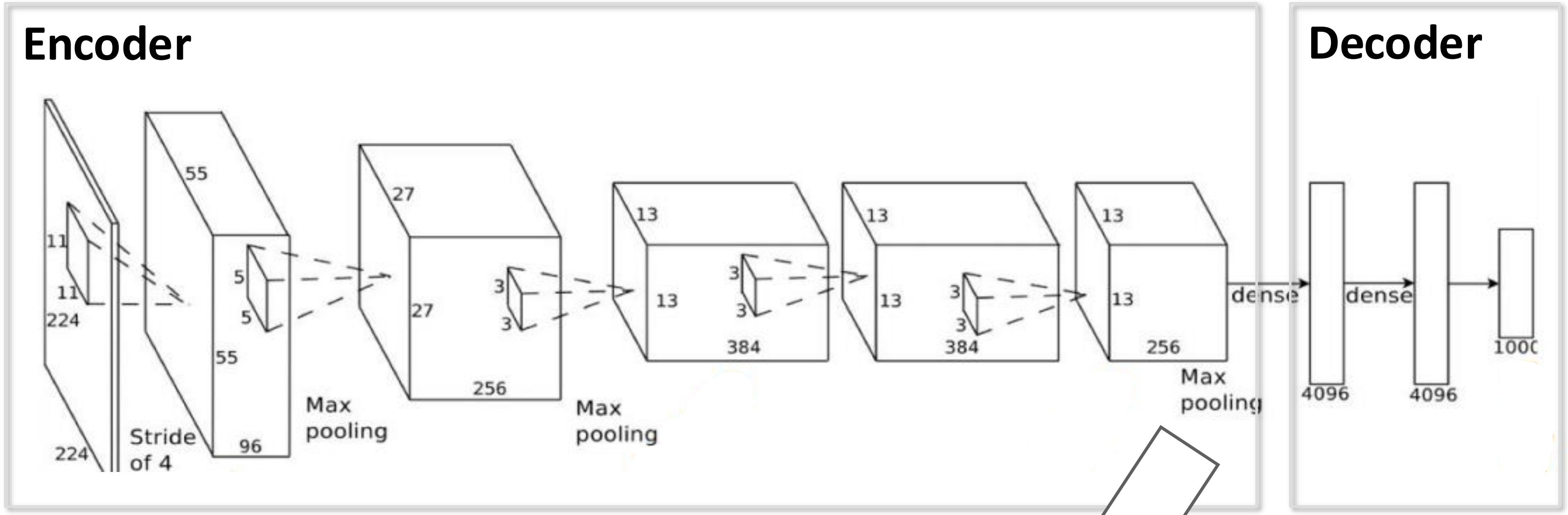


Preview of CNNs

- **Idea:** Use a pre-trained CNN to obtain a better feature representation.



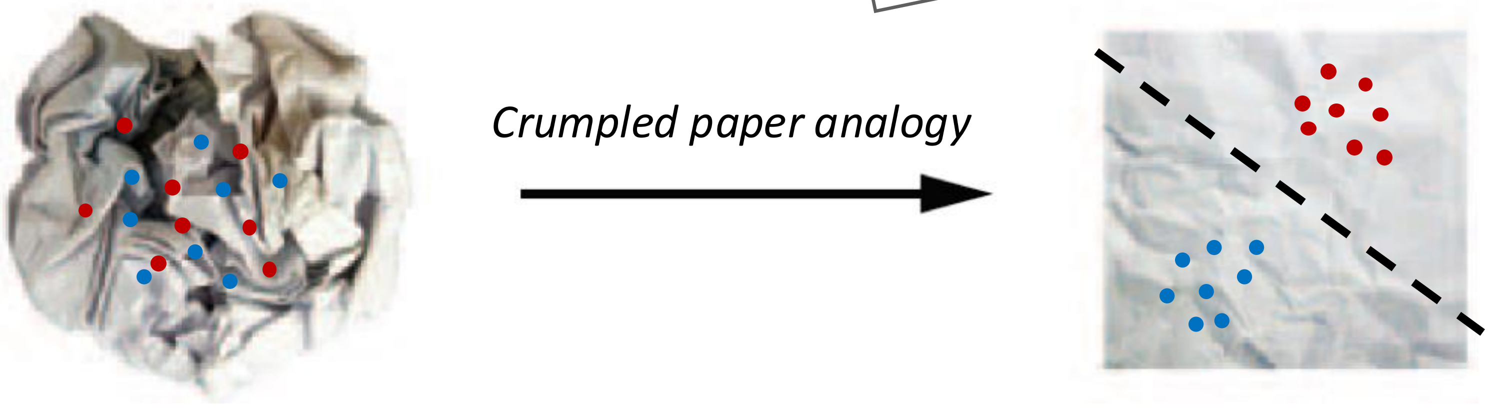
Why would that work?



Answer: The CNN has been trained to classify images. It performs some sort of non-linear transformation of the pixel values into a representation that is optimal for classification.

High-dimensional representation in input space (pixels) is all messed up!

Also, it is harder for the data to cluster due to curse of dimensionality

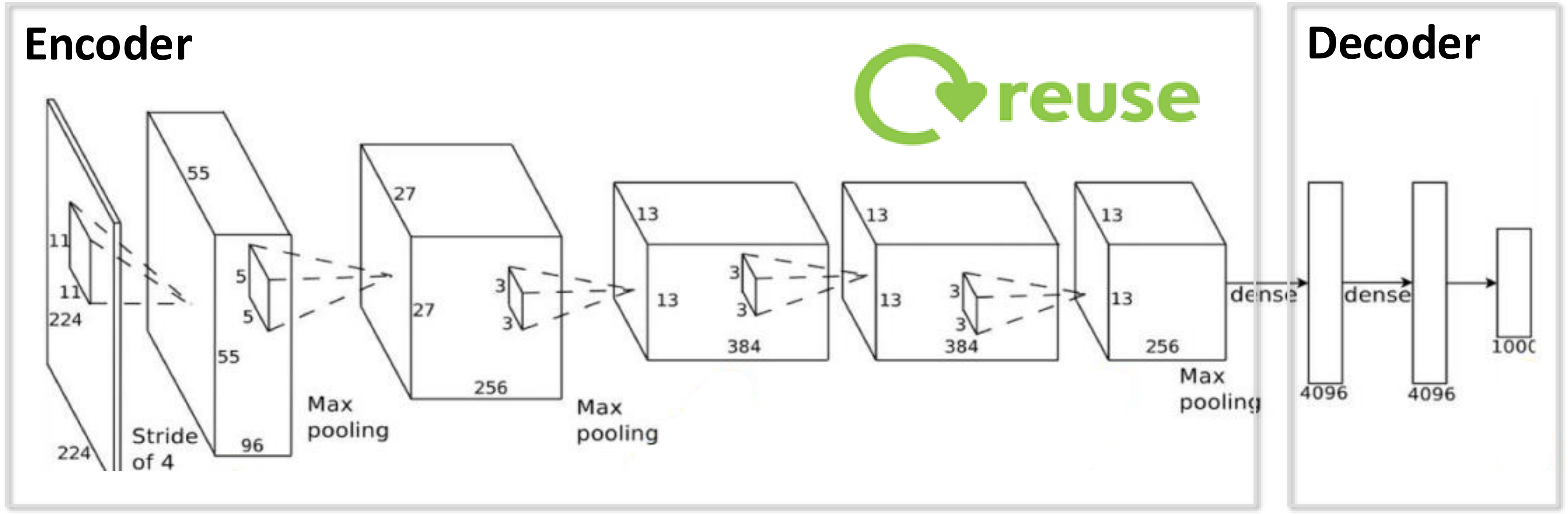


Low-dimensional representation learned by the CNN

● Cats

● Dogs

Transfer Learning



- Training a ConvNet from scratch can take days
- Use pre-trained encoder trained on ImageNet
- Fine-tune weights of the decoder, which is just a linear classifier performing softmax regression (like we have learned about in this lecture)
- Training time: Typically less than 1 hour
- More details: How transferable are features in deep neural networks?

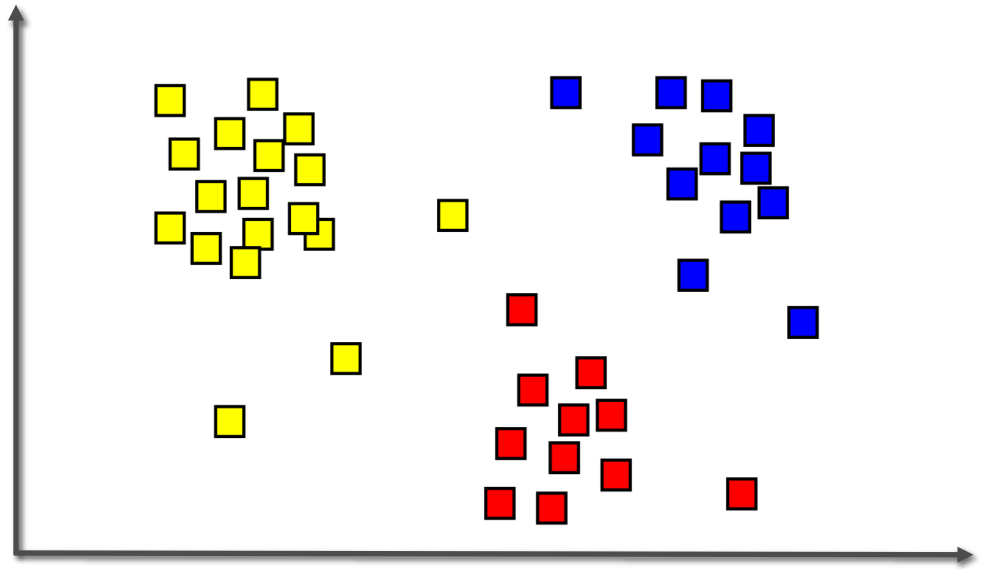
http://cs231n.github.io/transfer-learning/

Feature learning

- In machine learning, **feature learning** or **representation learning** is a set of techniques that allows a system to **automatically discover the representations needed** for feature detection or classification from raw data.
- This replaces manual feature engineering and allows a machine to learn the features that are optimal to solve a specific task.
- Two approaches to learning features:
  - **Supervised (from labelled data):** Includes neural networks and support vector machines (SVM) if inputs are the raw image pixels.
  - **Unsupervised (from unlabelled data):** Includes K-means clustering, principal component analyses (PCA), and autoencoders.

Cluster analysis

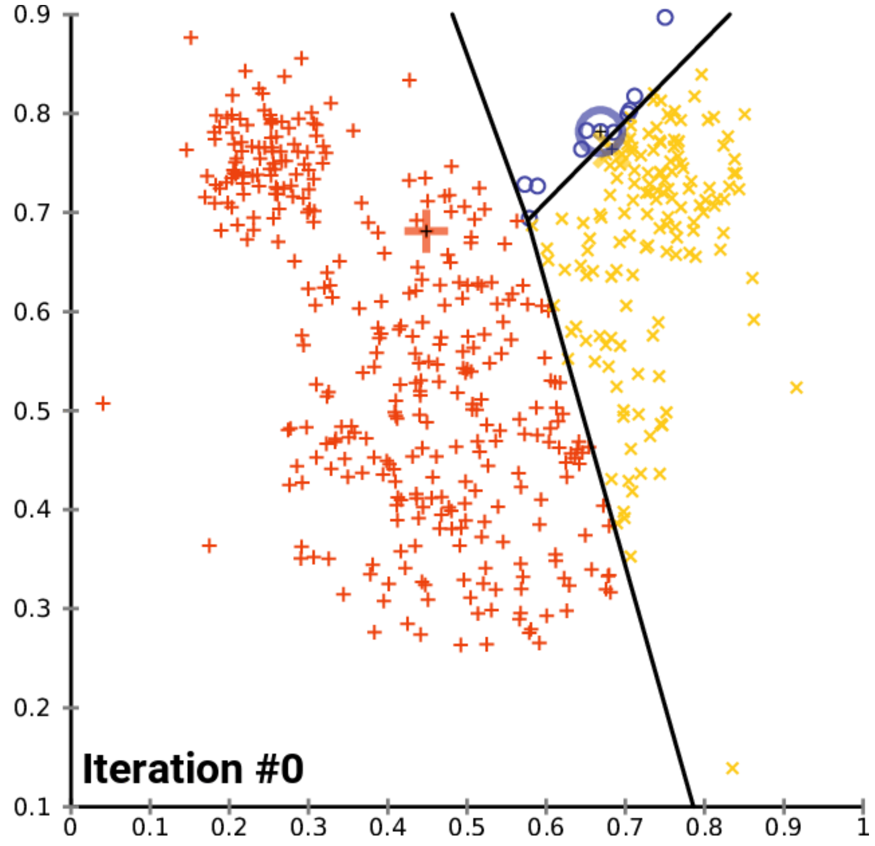
- **Cluster analysis** or clustering is the task of grouping a set of vectors in such a way that vectors in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).
- Example: Group the data points in the plot below, not knowing anything about possible labels (i.e., unsupervised).



How would you assign colors to these datapoints?

Basic K-Means clustering

- ***K*-Means** is a popular and simple clustering method.
- It partitions *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean, called a **centroid**.
- Standard algorithm
  - Initialization: Generate *k* random centroids.
  - Assignment step: Assign each observation to the nearest centroid (e.g., using L1 or L2 norm).
  - Update step: Update the centroids by taking the mean of all observations that were assigned to it.
  - Repeat Assignment + Update until convergence.



https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/K-means_convergence.gif/220px-K-means_convergence.gif

Standard algorithm

```
1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: {
        For every $i$, set
            $c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$
        For each $j$, set
            $\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$
    }
```

Initial centroids could also be a random subset of the observations

Further reading + online videos/demos

- Unsupervised Feature Learning and Deep Learning Tutorial, Andrew Ng
  - http://deeplearning.stanford.edu/tutorial/
- Machine Learning, Andrew Ng (Stanford), full course:
  - https://www.youtube.com/playlist?list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN
- Neural network playground:
  - https://playground.tensorflow.org/
- A Short Introduction to Entropy, Cross-Entropy and KL-Divergence, Aurélien Géron:
  - https://www.youtube.com/watch?v=ErfnhcEV1O8
- L1 and L2 regularization (advanced), Alexander Ihler:
  - https://www.youtube.com/watch?v=sO4ZirJh9ds
- Sneak peak at ConvNet for MNIST:
  - https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html