Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 1

October 4th, 2022

Image Processing and Recognition – Course

- **Scope**
  - Learn the basics of image processing and recongition using deep learning
- **Contents**
  1. Introduction. Linear models
  2. Multilayer perceptrons
  3. Convolutional neural networks
  4. Recurrent neural networks
  5. Attention mechanisms
  6. Computer vision
- **Grading**
  - written exam (open book)
  - 0.2 points per presence
  - 50% of final grade

Image Processing and Recognition – Laboratory

- **Scope**
  - Implement the algorithms presented at the course in PyTorch, using Google Colaboratory
- **Contents**
  1. Introduction to Python
  2. Introduction to PyTorch
  3. Multilayer perceptrons
  4. Convolutional neural networks
  5. Recurrent neural networks
  6. Attention mechanisms
  7. Computer vision
- **Grading**
  - 2 practical tests (open book)
  - 50% of final grade

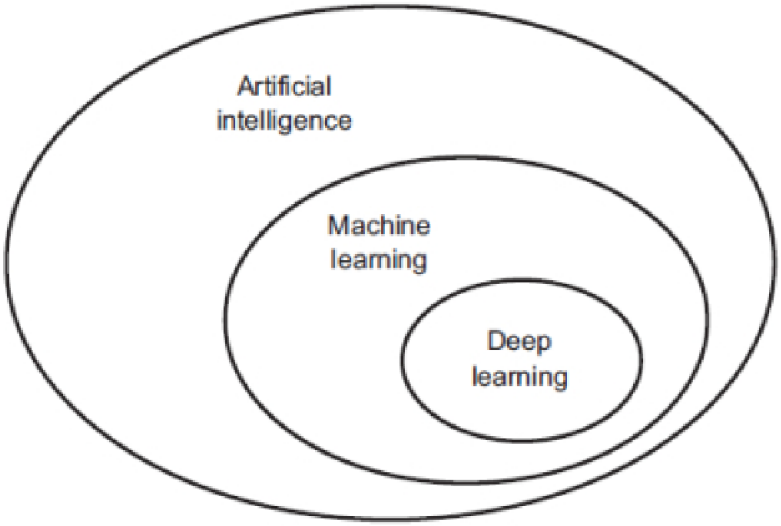

Figure 1: Artificial intelligence, machine learning, and deep learning.

1.1.1 Artificial intelligence

- *the effort to automate intellectual tasks normally performed by humans*
- *symbolic AI*: programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge
- peak popularity during the *expert systems* boom of the 1980s
- symbolic AI – suitable to solve well-defined, logical problems, but intractable to figure out explicit rules for solving more complex, fuzzy problems

- rather than programmers crafting data processing rules by hand, could a computer automatically learn these rules by looking at data?

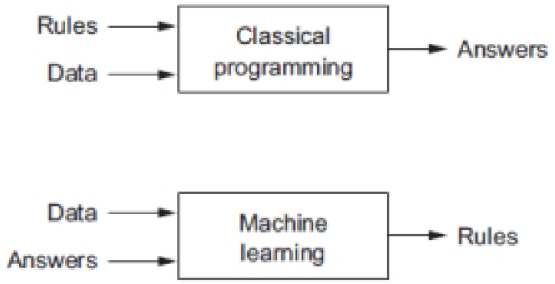

Figure 2: Machine learning: a new programming paradigm.

- a machine learning system is *trained* rather than explicitly programmed
- although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI

1.1.3 Learning representations from data

- machine learning discovers rules to execute a data processing task, given examples of what's expected
- so, to do machine learning, we need three things:
  - *Input data points*
  - *Examples of the expected output*
  - *A way to measure whether the algorithm is doing a good job*
- the central problem in machine learning and deep learning is to *meaningfully transform data*: in other words, to learn useful *representations* of the input data at hand – representations that get us closer to the expected output
- representation – a different way to look at data – to *represent* or *encode* data

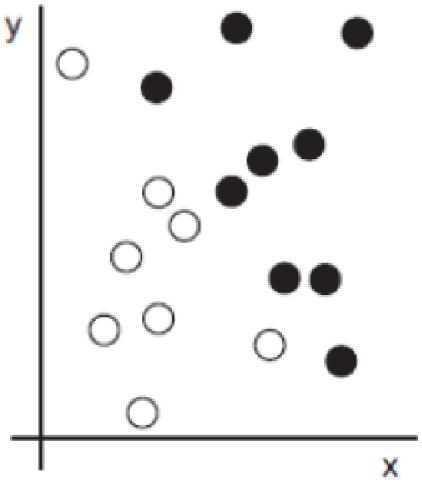

Figure 3: Some sample data.

- inputs?
- expected outputs?
- a way to measure whether the algorithm is doing a good job?

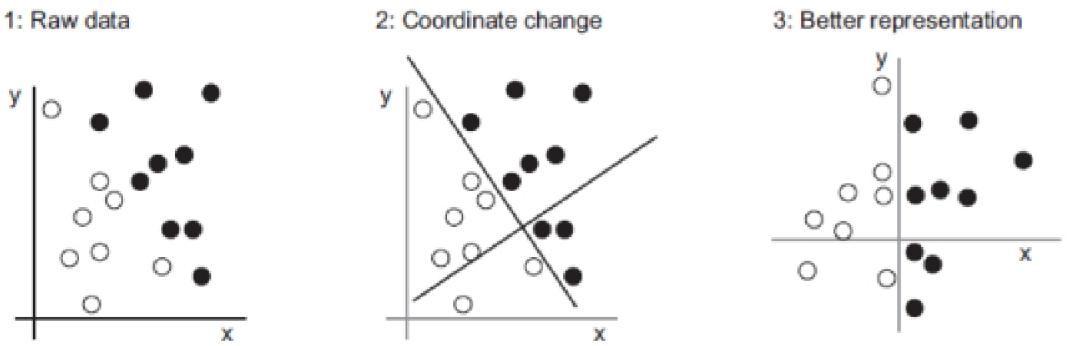

Figure 4: Coordinate change.

- with this representation, the black/white classification problem can be expressed as a simple rule: “Black points are such that $x > 0$”, or “White points are such that $x < 0$”

1.1.3 Learning representations from data

- if we tried systematically searching for different possible coordinate changes, and used as feedback the percentage of points being correctly classified – machine learning
- *learning*, in the context of machine learning, describes an automatic search process for better representations
- machine learning algorithms aren't creative in finding transformations; they're merely searching through a predefined set of operations, called a *hypothesis space*
- two main types of problems:
  - *classification*, where the goal is to predict a single discrete label for an input data point
  - *regression*, which consists of predicting a continuous value instead of a discrete label

1.1.4 Four branches of machine learning
Supervised learning

- it consists of learning to map input data to known targets (also called *annotations*), given a set of examples (often annotated by humans)
- generally, almost all applications of machine learning that are in the spotlight these days belong in this category, such as speech recognition, image classification, and language translation
- although supervised learning mostly consists of classification and regression, there are more exotic variants as well, including the following:
  - *Sequence generation*—Given a picture, predict a caption describing it.
  - *Syntax tree prediction*—Given a sentence, predict its decomposition into a syntax tree.
  - *Object detection*—Given a picture, draw a bounding box around certain objects inside the picture. This can also be expressed as a classification problem (given many candidate bounding boxes, classify the contents of each one) or as a joint classification and regression problem, where the bounding box coordinates are predicted via vector regression.
  - *Image segmentation*—Given a picture, draw a pixel-level mask on a specific object.

Unsupervised learning

- this branch of machine learning consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand
- unsupervised learning is at the core of data analytics, and it's often a necessary step in better understanding a dataset before attempting to solve a supervised learning problem
- *dimensionality reduction* and *clustering* are well-known categories of unsupervised learning

Self-supervised learning

- this is a specific instance of supervised learning, but it's different enough that it deserves its own category
- self-supervised learning is supervised learning without human-annotated labels
- there are still labels involved (because the learning has to be supervised by something), but they're generated from the input data, typically using a heuristic algorithm
- for instance, *autoencoders* are a well-known instance of self-supervised learning, where the generated targets are the input, unmodified
- in the same way, trying to predict the next frame in a video, given past frames, or the next word in a text, given previous words, are instances of self-supervised learning (*temporally supervised learning*, in this case: supervision comes from future input data)

Reinforcement learning

- long overlooked, this branch of machine learning recently started to get a lot of attention after Google DeepMind successfully applied it to learning to play Atari games (and, later, learning to play Go at the highest level)
- in reinforcement learning, an *agent* receives information about its environment and learns to choose actions that will maximize some reward
- for instance, a neural network that "looks" at a video game screen and outputs game actions in order to maximize its score can be trained via reinforcement learning
- currently, reinforcement learning is mostly a research area and hasn't yet had significant practical successes beyond games
- in time, however, we expect to see reinforcement learning take over an increasingly large range of real-world applications: self-driving cars, robotics, resource management, education, and so on

1.2 Linear regression

- *linear regression* is a very widely used method for predicting a real-valued output (dependent variable, also called the *label* or *target*) $y \in \mathbb{R}$, given a set of real-valued inputs (independent variables, also called *features* or *covariates*) $x_1, x_2, \ldots, x_d$
- we assume that the relationship between the independent variables $x_1, x_2, \ldots, x_d$ and the dependent variable y is *linear*, i.e., that y can be expressed as a weighted sum of the features $x_1, x_2, \ldots, x_d$, given some noise on the observations
- also, we assume that the noise follows a normal distribution
- to develop a model for predicting $y \in \mathbb{R}$, we need to obtain a *dataset* consisting of known outputs for corresponding inputs
- the dataset is called a *training dataset* or *training set*, and each row is called an *example* or *data point* or *sample*

- we will use n to denote the number of examples in our dataset
- we index the data examples by i, denoting each input as $x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}$ and the corresponding label as $y^{(i)}$
- the linear regression model is expressed as:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d, \tag{1}$$

where

- $\hat{y}$ is the *predicted* value
- x_j is the jth *feature* value
- d is the number of features
- w_j is the jth model *weight* (including the *bias* or *intercept* term w_0 and the *feature weights* $w_1, w_2, \ldots, w_d$)

- strictly speaking, (1) is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via weighted sum, combined with a *translation* via the added bias
- the weights determine the influence of each feature on our prediction and the bias just says what the predicted value should be when all of the features take value 0
- given a dataset, our goal is to choose the weights $w_1, w_2, \ldots, w_d$ and the bias w_0 such that, on average, the predictions $\hat{y}$ made by our model *best fit* the true labels y observed in the data
- models whose output prediction is determined by the affine transformation of input features are *linear models*, where the affine transformation is specified by the chosen weights and bias

- the linear regression model can be written more concisely in the vectorized form:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}, \tag{2}$$

where

  - $\boldsymbol{w}$ is the model's *weight vector*, containing the bias or intercept term w_0 and the feature weights $w_1, w_2, \ldots, w_d$
  - $\boldsymbol{x}$ is the instance's *feature vector*, containing x_0 to x_d, where x_0 is a "dummy feature" always equal to 1, which allows including the bias or intercept term into the weight vector

- in machine learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column
- $\boldsymbol{w}^\top$ is the *transpose* of $\boldsymbol{w}$ (a row vector instead of a column vector) and $\boldsymbol{w}^\top \boldsymbol{x}$ is the matrix multiplication of $\boldsymbol{w}^\top$ and $\boldsymbol{x}$, which represents the scalar product of $\boldsymbol{w}$ and $\boldsymbol{x}$
- because the scalar product is symmetric, (2) can also be written as:

$$\hat{y} = \boldsymbol{x}^\top \boldsymbol{w}. \tag{3}$$

- in (3), the vector $\boldsymbol{x}$ corresponds to features of a single data example
- we will often find it convenient to refer to features of our entire dataset of n examples via the *design matrix* $\boldsymbol{X} \in \mathbb{R}^{n \times (d+1)}$
- here, $\boldsymbol{X}$ contains one row for every example and one column for every feature:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \dots & x_d^{(n)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}^{(1)\top} \\ \boldsymbol{x}^{(2)\top} \\ \vdots \\ \boldsymbol{x}^{(n)\top} \end{bmatrix}.$$

- for a collection of features $\boldsymbol{X}$, the predictions $\hat{\boldsymbol{y}} \in \mathbb{R}^n$ can be expressed via the matrix-vector product:

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}, \tag{4}$$

where $\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(n)} \end{bmatrix}$

- given features of a training dataset $\boldsymbol{X}$ and corresponding (known) labels $\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}$, the goal of linear regression is to find the weight vector $\boldsymbol{w}$ that given features of a new data example sampled from the same distribution as $\boldsymbol{X}$, the new example's label will (in expectation) be predicted with the lowest error
- even if we believe that the best model for predicting y given $\boldsymbol{x}$ is linear, we would not expect to find a real-world dataset of n examples where $y^{(i)}$ exactly equals $\boldsymbol{w}^\top \boldsymbol{x}^{(i)}$, for all $1 \leq i \leq n$
- for example, whatever instruments we use to observe the features $\boldsymbol{X}$ and labels $\boldsymbol{y}$ might suffer small amount of measurement error
- thus, even when we are confident that the underlying relationship is linear, we will incorporate a *noise term* to account for such errors

- to refresh our memory, the probability density of a *normal distribution* $\mathcal{N}(\mu, \sigma^2)$ with *mean* μ and *variance* σ^2 (standard deviation σ) is given as:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

- we assume that observations arise from noisy observations, where the noise is normally distributed as follows:

$$y = \boldsymbol{w}^\top \boldsymbol{x} + \epsilon,$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$

- thus, we can now write the *likelihood* of seeing a particular $y^{(i)}$ for a given $\boldsymbol{x}^{(i)}$ as:

$$p(y^{(i)}|\boldsymbol{x}^{(i)}, \boldsymbol{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2\right).$$

- now, according to the principle of *maximum likelihood*, the best values of weights $\boldsymbol{w}$ are those that maximize the likelihood of the entire dataset:

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) = \prod_{i=1}^{n} p(y^{(i)}|\boldsymbol{x}^{(i)}, \boldsymbol{w}). \tag{5}$$

- estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*
- while maximizing the product of many exponential functions might look difficult, we can simplify things significantly, without changing the objective, by maximizing the log of the likelihood instead
- for historical reasons, optimizations are more often expressed as minimization rather than maximization
- so, without changing anything, we can minimize the *negative log-likelihood* $-\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})$; from (5), we have that:

$$-\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) = \sum_{i=1}^{n} \left[\frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2 \right].$$

- now we just need one more assumption that σ is some fixed constant; thus we can ignore the first term because it does not depend on $\boldsymbol{w}$
- because the solution does not depend on σ, we can also ignore the multiplicative constant $\frac{1}{\sigma^2}$
- as a consequence, negative log-likelihood is given as:

$$\sum_{i=1}^{n} \frac{1}{2}(y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2.$$

- because $\hat{y}^{(i)} = \boldsymbol{w}^\top \boldsymbol{x}^{(i)}$ are the predictions of the linear regression model, according to (4), the negative log-likelihood finally becomes:

$$\sum_{i=1}^{n} \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2.$$

- on the other hand, before we start searching for the best weights (or model weights) $\boldsymbol{w}$, we will need two more things:
  - a quality measure for some given model
  - a procedure for updating the model to improve its quality
- in general, the quality measure for a certain model is given by the *loss function*, which quantifies the distance between the *real* and *predicted* value of the target
- the loss will usually be a non-negative number, where smaller values are better and perfect predictions have a loss of 0
- the most popular loss function in regression problems is the *squared error*
- when the model prediction for an example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the squared error is given by:

$$l^{(i)}(\boldsymbol{w}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2.$$

- the constant $\frac{1}{2}$ makes no real difference, but will prove notationally convenient, canceling out when we take the derivative of the loss

- since the training dataset is given to us, and thus out of our control, the empirical error is only a function of the model weights
- to make things more concrete, consider the example below, where we plot a regression problem for a one-dimensional case, as shown in Figure 5

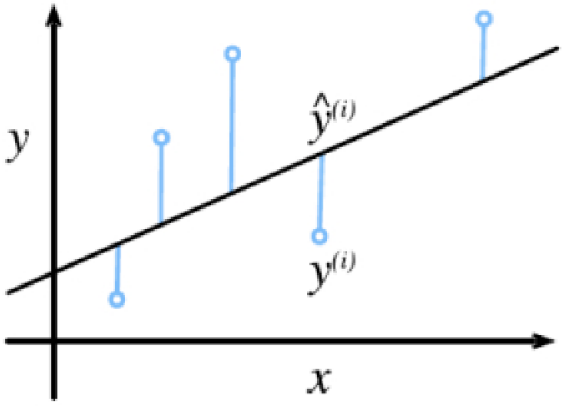

Figure 5: Linear regression with $d = 1$.

- note that large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger contributions to the loss, due to the quadratic dependence
- to measure the quality of a model on the entire dataset of n examples, we simply average (or equivalently, sum) the losses on the training set:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} - y^{(i)})^2. \tag{6}$$

- this is called the *mean squared error loss function*; using (4), the mean squared error can be written as:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}).$$

- we can now see that, in the case of linear regression, the negative log-likelihood is *equal* (up to multiplicative constants) to the mean squared error loss function
- when training the model, we want to find parameters $\boldsymbol{w}$ that minimize the total loss across all training examples:

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{w}).$$

- recall from first year calculus that, in order to find the minimum of the function $\mathcal{L}(\boldsymbol{w})$, we must compute its gradient, equal it with $\mathbf{0}^\top$, and extract the value of the minimum:

$$\begin{aligned}\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}) &= \nabla_{\boldsymbol{w}}\left(\frac{1}{2n}(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})\right)\\ &= \frac{1}{2n}\nabla_{\boldsymbol{w}}\left(\boldsymbol{w}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w}-2\boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{w}+\boldsymbol{y}^\top\boldsymbol{y}\right)\\ &= \frac{1}{n}(\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w}-\boldsymbol{X}^\top\boldsymbol{y}). \qquad (7)\end{aligned}$$

- from $\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})=\mathbf{0}^\top$, we obtain:

$$\begin{aligned}\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w})=\mathbf{0}^\top &\Leftrightarrow \boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w}-\boldsymbol{X}^\top\boldsymbol{y}=\mathbf{0}^\top\\ &\Leftrightarrow \boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w}=\boldsymbol{X}^\top\boldsymbol{y}\\ &\Leftrightarrow \boldsymbol{w}=(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y}.\end{aligned}$$

- thus, for linear regression, we have the analytic (closed-form) solution:

$$\boldsymbol{w}^*=(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y}.$$

- this is called the *normal equation*
- the matrix $\boldsymbol{X}^{+} := (\boldsymbol{X}^{\top}\boldsymbol{X})^{-1}\boldsymbol{X}^{\top}$ is called the *pseudoinverse* or *Moore-Penrose inverse* of matrix $\boldsymbol{X}$
- the pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix $\boldsymbol{X}$ into the matrix multiplication of three matrices: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^{\top}$
- the pseudoinverse is computed as $\boldsymbol{X}^{+} = \boldsymbol{V}\boldsymbol{S}^{+}\boldsymbol{U}^{\top}$; to compute the matrix $\boldsymbol{S}^{+}$, the algorithm takes $\boldsymbol{S}$ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix
- this approach is more efficient than computing the normal equation, plus it handles edge cases nicely: indeed, the normal equation may not work if the matrix $\boldsymbol{X}^{\top}\boldsymbol{X}$ is not invertible (i.e., singular), such as if $n < d$ or if some features are redundant, but the pseudoinverse is always defined

- the normal equation computes the inverse of $\boldsymbol{X}^\top \boldsymbol{X}$, which is an $(d+1) \times (d+1)$ matrix
- the computational complexity of inverting such a matrix is typically about $\mathcal{O}(d^{2.4})$ to $\mathcal{O}(d^3)$, depending on the implementation
- the SVD approach is about $\mathcal{O}(d^2)$
- both the normal equation and the SVD approach get very slow when the number of features grows large (e.g., $100,000$)
- on the positive side, both are linear with regard to the number of instances in the training set (they are $\mathcal{O}(n)$), so they handle large training sets efficiently, provided they can fit in memory

- also, once we have trained our linear regression model (using the normal equation or any other algorithm), predictions are very fast: the computational complexity is *linear* with regard to both the number of instances we want to make predictions on and the number of features
- in other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time
- next, we will look at a very different way to train a linear regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory

Thank you!