# BZAN 554:
# Deep Learning for Business Applications

## Introduction: Answers to Frequently Asked Questions

Michel Ballings

# Outline

# What is deep learning?

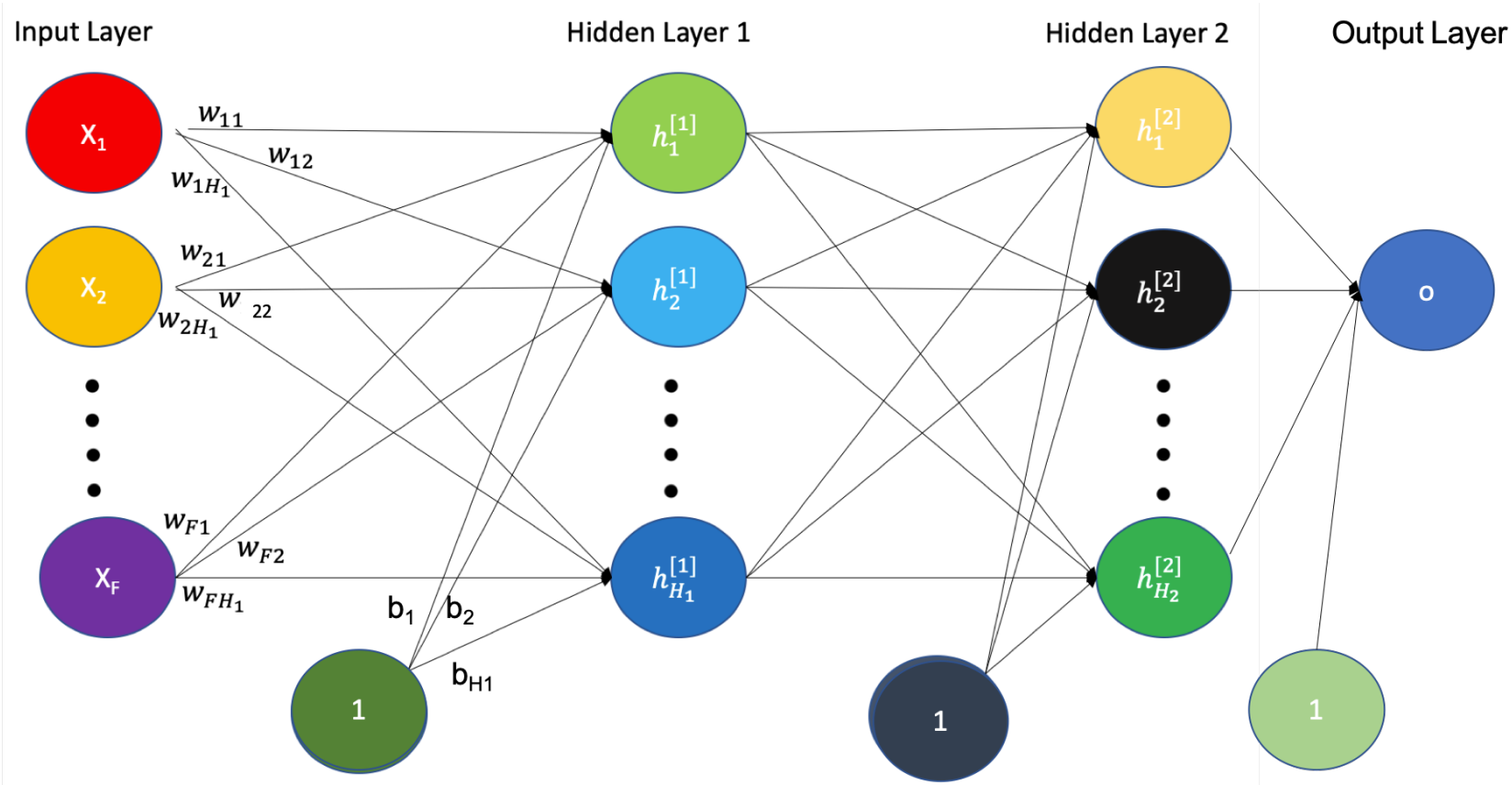
Deep Learning (DL) is a field within Machine Learning (ML) and is a set of algorithms inspired by the structure and function of the brain. DL algorithms are very powerful and can learn from the environment just like humans (i.e., from pixels and audio), sometimes resulting in superhuman performance. That is the reason why some people call DL models (DL algorithm + data) artificial intelligence.

Typical nomenclature by community:

| | Statistics community | ML community | DL community |
|---|---|---|---|
| Modeling | estimating | learning | deep learning |
| Algorithm | estimator | learner | network, graph |
| Model | model | hypothesis | network, graph, map, AI |
| Representation | specification | structure | architecture |

# Notation

Assuming a multi-layered neural network:



# Notation

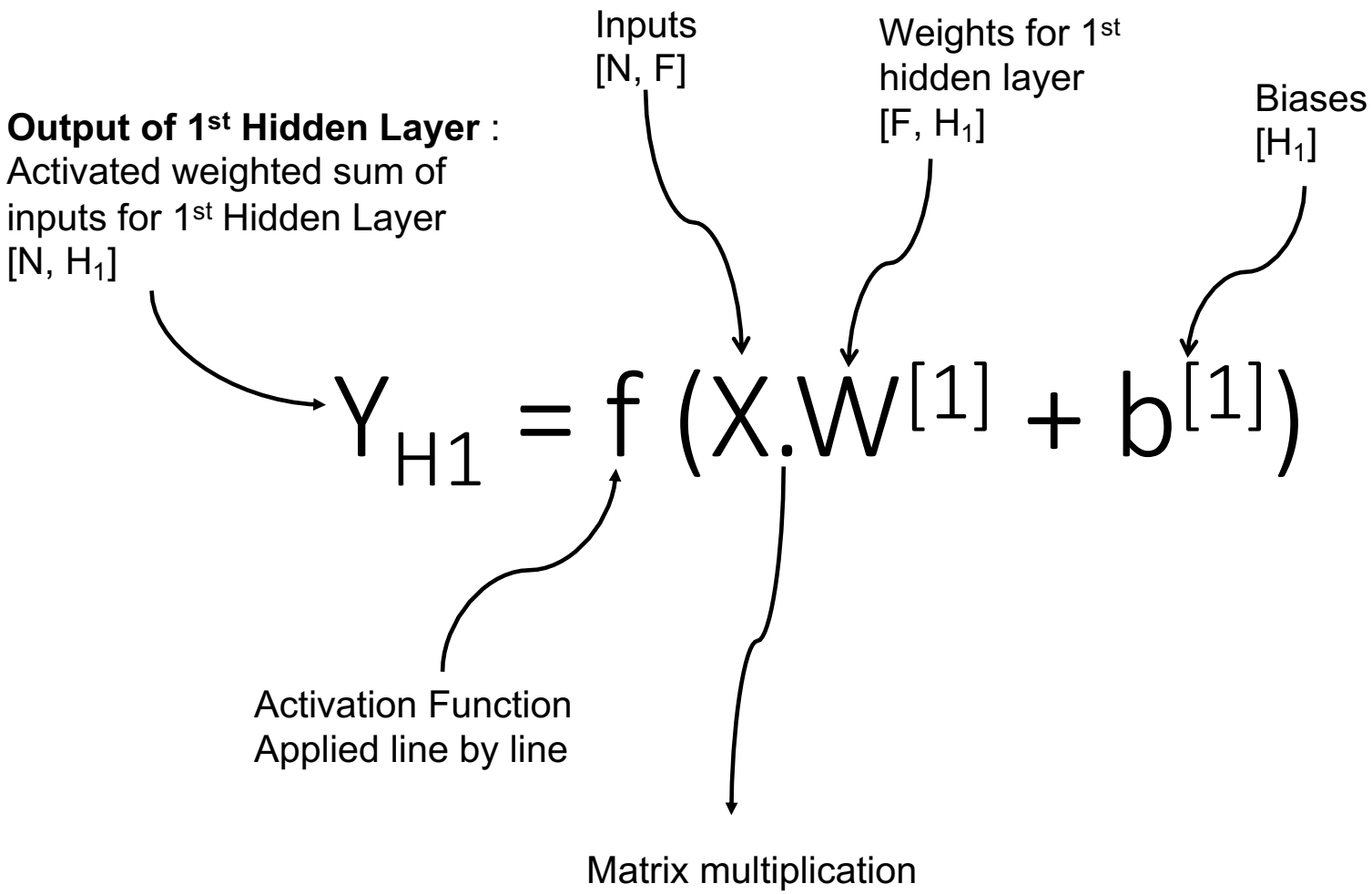
Assuming the multi-layered neural network in the previous slide:

- Circles are called neurons, units, or nodes. The units on the far left side are called inputs or input units. They are passthrough units because their function is for data to pass through them without applying any operation to the data. We can conceive of them as *variables* or *features*.
- The collection of $F$ inputs is called the input layer $X = \{x_1, x_2, ..., x_F\}$
- The first hidden layer has $H_1$ hidden units. The second hidden layer has $H_2$ hidden units (neurons).
- Superscript [.] identifies the receiving hidden or output layer
- $W^{[1]}$ contains the weights from the input layer to the first hidden layer, and has shape $[F, H_1]$. $W^{[2]}$ contains the weights from the first hidden layer to the second hidden layer, and has shape $[H_1, H_2]$. $W^{[3]}$ contains the weights from the second hidden layer to the output layer, and has shape $[H_2, 1]$.

# Notation

- There can be any number of hidden layers and outputs
- $b^{[1]}$ is a row vector of biases of size $[H_1]$, each element is a bias for one hidden unit in the first hidden layer. $b^{[2]}$ are the biases of size $[H_2]$ going to the second hidden layer. $b^{[3]}$ is a single bias for the output. A more standard way of doing this is to create an additional input, with all 1s, and then treat is as a normal feature, but TensorFlow uses a process called broadcasting (creating a 1 for every observation that is passed through the input layer). Think of this as a unit emitting a 1 every time a new record is processed. This is very convenient to the user.

# Notation: From input layer to 1st Hidden layer



# Notation: From Input Layer to 1st Hidden layer

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1F} \\ x_{21} & x_{22} & \dots & x_{2F} \\ \cdot & \cdot & \dots & \cdot \\ x_{N1} & x_{N2} & \dots & x_{NF} \end{bmatrix}_{N \times F} \tag{1}$$

where each column of matrix X represents features and each row represents observations.

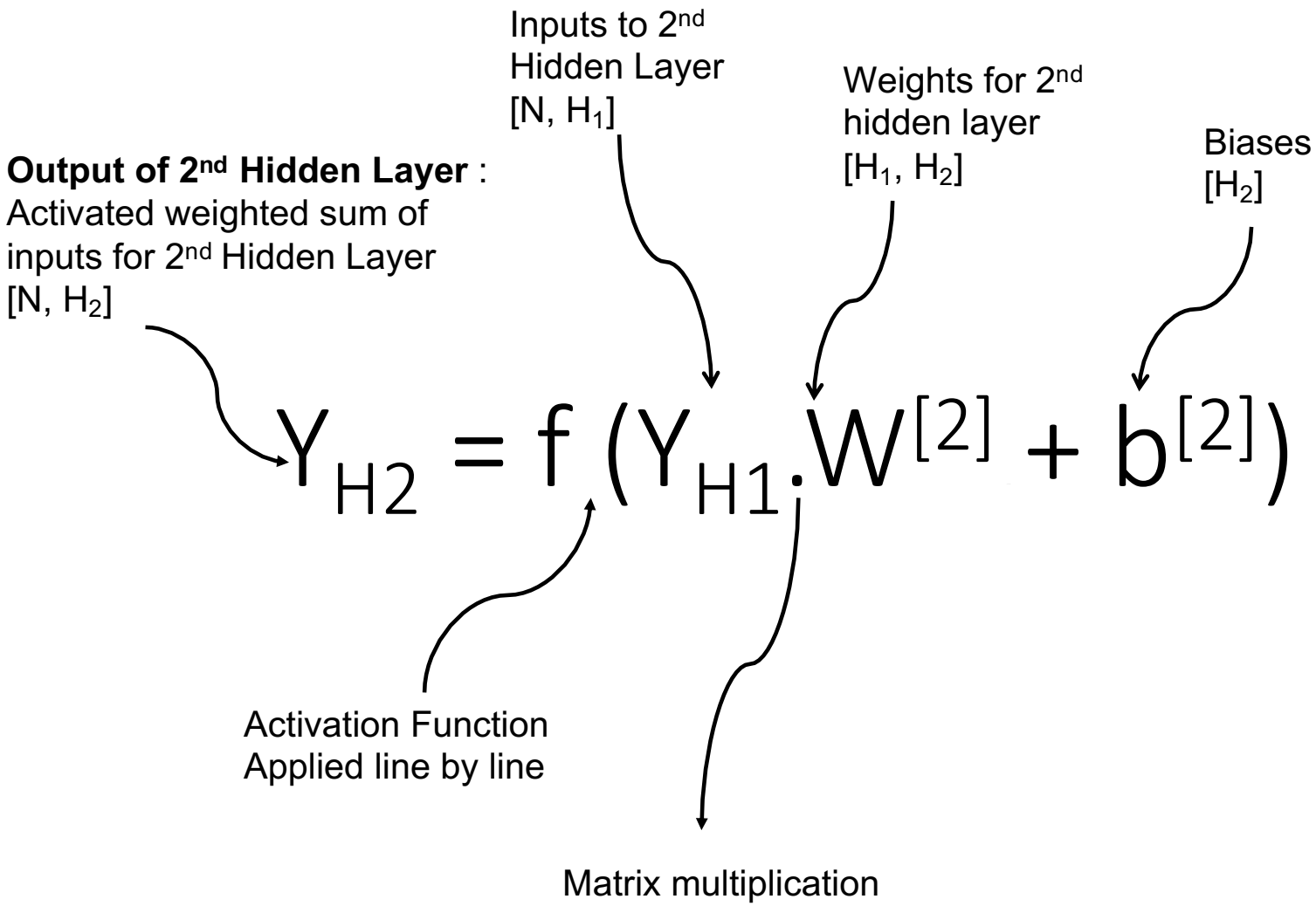
$$W^{[1]} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1H_1} \\ w_{21} & w_{22} & \dots & w_{2H_1} \\ \cdot & \cdot & \dots & \cdot \\ w_{F1} & w_{F2} & \dots & w_{FH_1} \end{bmatrix}_{F \times H_1} \tag{2}$$

where each column of matrix W can be thought of weights going from features in the input layer to a unit in the first hidden layer. In other words, each column represents all the arrows going into a hidden unit in the first hidden layer. Each row represents all the arrows going out of an input feature to the different hidden units.

# Notation: From 1st Hidden layer to 2nd Hidden layer

- For all other hidden layers, calculations are similar.
- Assume that the second hidden layer has $H_2$ neurons.
- $W^{[2]}$ carries the weights from the 1st hidden layer to the 2nd hidden layer, and has size $[H_1, H_2]$.
- $b^{[2]}$ are the biases, a vector of size $[H_2]$, in which each element is a bias for one hidden unit in the 2nd hidden layer.
- Inputs to the 2nd hidden layer will be outputs of the 1st hidden layer (i.e., activated weighted sum of inputs of 1st hidden layer).

# Notation: From 1st Hidden layer to 2nd Hidden layer



# Notation: From 2nd Hidden layer to output layer

The calculations are similar:

- Assume that the output hidden layer has one output neuron (there can be any number of output neurons).
- $W^{[3]}$ carries the weights from the 2nd hidden layer to the output layer, and has size $[H_2, 1]$.
- The activation function is sometimes linear (i.e., there is no function). This is often the case when the output is continuous. More about this later.

# A Useful Analogy

It is useful to think about a neural network as a water treatment plant. The data is the water. The units are basins, each treating the water with a different operation. The connections are pipelines or tubes.

# Outline

# Outline

# When should I use deep neural networks? Situation 1

**When we have**

- large amounts of data (and/or data is streaming in really fast) **and**
- complex nonlinear problems.

**Why those two requirements?**

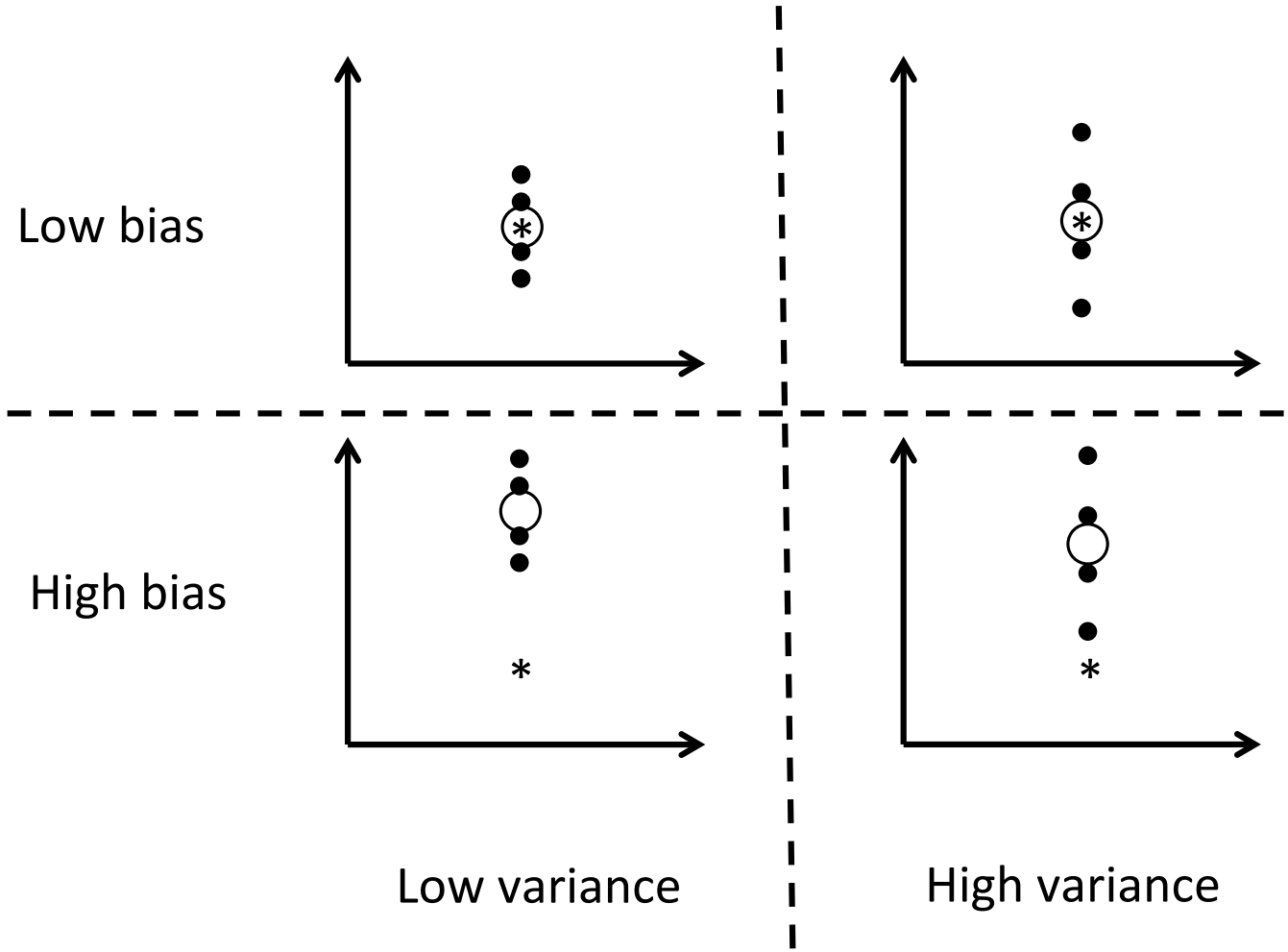
**Statistical reason.** Neural networks automatically learn complex functions, resulting in very accurate models, but making them prone to overfitting. Larger data sets reduce the risk of overfitting. Complexity of the model can be increased when data are increased. To understand this we need to know the bias-variance tradeoff and the PAC learning theorem.
**Computational reason.** Neural networks are trained incrementally, and have low constant RAM and constant CPU per instance.

# Bias[1]-Variance Tradeoff

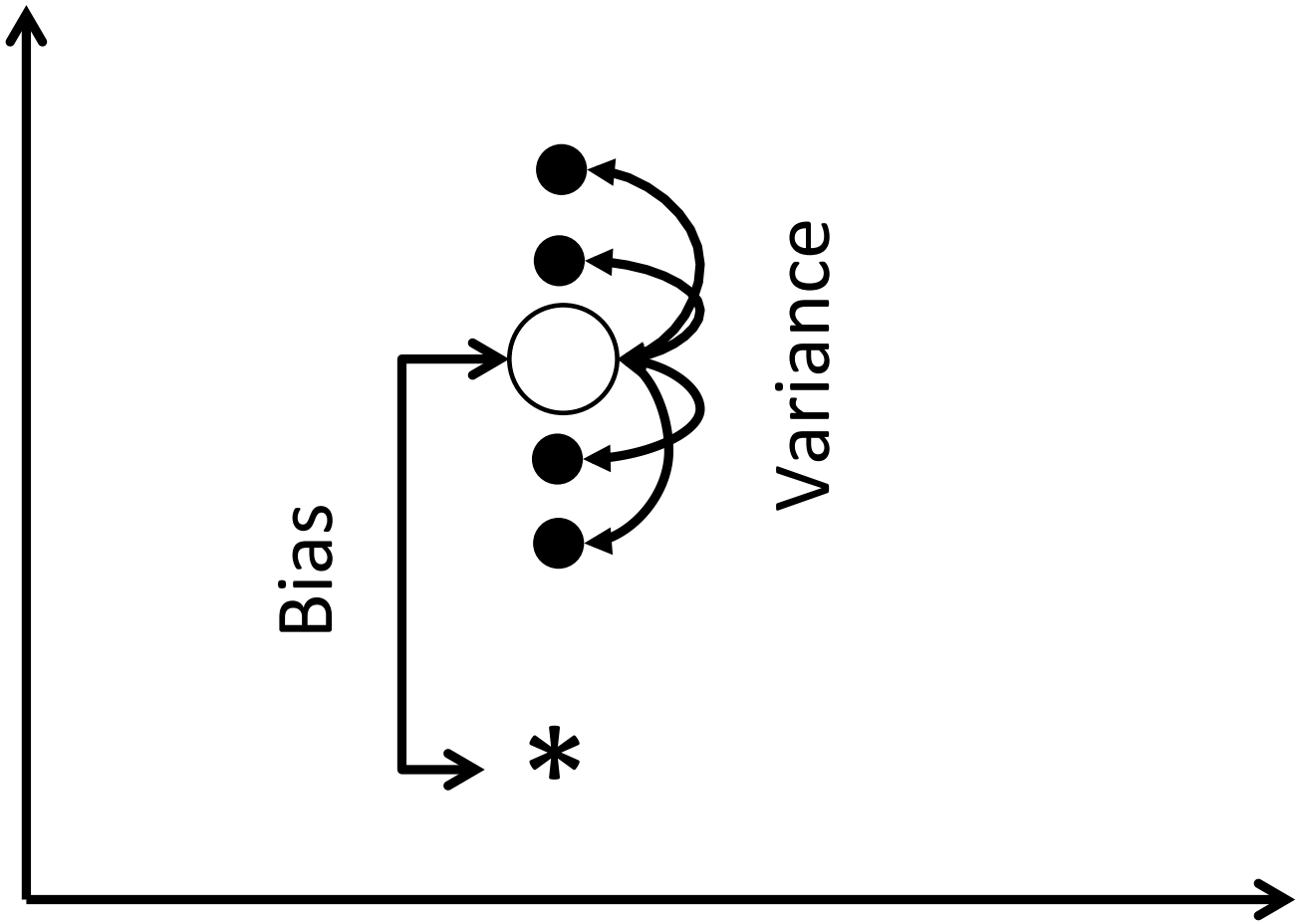
- Prediction error consists of (1) bias, (2) variance, and (3) irreducible error (noise). Noise is of lesser concern because we cannot change anything about it. Therefore we will focus on bias and variance.
- Consider:
  - a single data point, called the true point $\tilde{y}$, denoted by $*$,
  - multiple predictions by different models $\hat{y}_i$, with i=$\{$1,2,. . . , K models$\}$ each denoted by $\bullet$, and
  - the average of all the predictions $\hat{y}_i$, called $\bar{y}$, denoted by o.
- If we would repeat the model building process (e.g., using new data, random initialization),
  - the bias denotes how far off the average of the predictions of the different models is from the true value, and
  - the variance denotes how much predictions of the different models vary.

[1] The word *bias* in the discussion of the bias-variance tradeoff is unrelated to the *bias* used earlier. Here it denotes a form of deviance from the true point, whereas earlier it referred to an intercept in a model.

# Difference between bias and variance, given a single data point



# Alternative perspective of difference between bias and variance, given a single data point



# Bias-variance decomposition

In what follows we'll see how to investigate bias and variance mathematically (called bias-variance decomposition). First, consider the following notation:

- $K$= number of predictions (i.e., number of realizations of the model)
- $N$= number of instances
- $\xi$= error of the model
- y= observed response
- y= $\tilde{y}$ + $\epsilon$, with $\tilde{y}$=true function, and $\epsilon$= error term (noise)
- $\hat{y}$= predicted response
- $\bar{y}$= average predicted response across different realizations of the model

The $\xi$ of a model is denoted by:

$$\xi = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{3}$$

We can decompose $\xi$ into bias, variance and noise. The first term is the squared bias, the second term is the variance and the third term is the squared noise.

$$\xi = \sum_{i=1}^{N} \left( (\bar{y}_i - \tilde{y}_i)^2 + \frac{1}{K-1} \sum_{j=1}^{K} (\hat{y}_{i,j} - \bar{y}_i)^2 + (y_i - \tilde{y}_i)^2 \right) \tag{4}$$

How do we use this equation in a practical way? First of all, we cannot do anything about noise, but we can do something about bias and variance. Hence we are only interested in the relative magnitudes of bias and variance and we simply set $\epsilon$ (noise) to 0. Recall that y= $\tilde{y}$ + $\epsilon$, so y now equals $\tilde{y}$. This helps us simplify our formula to:

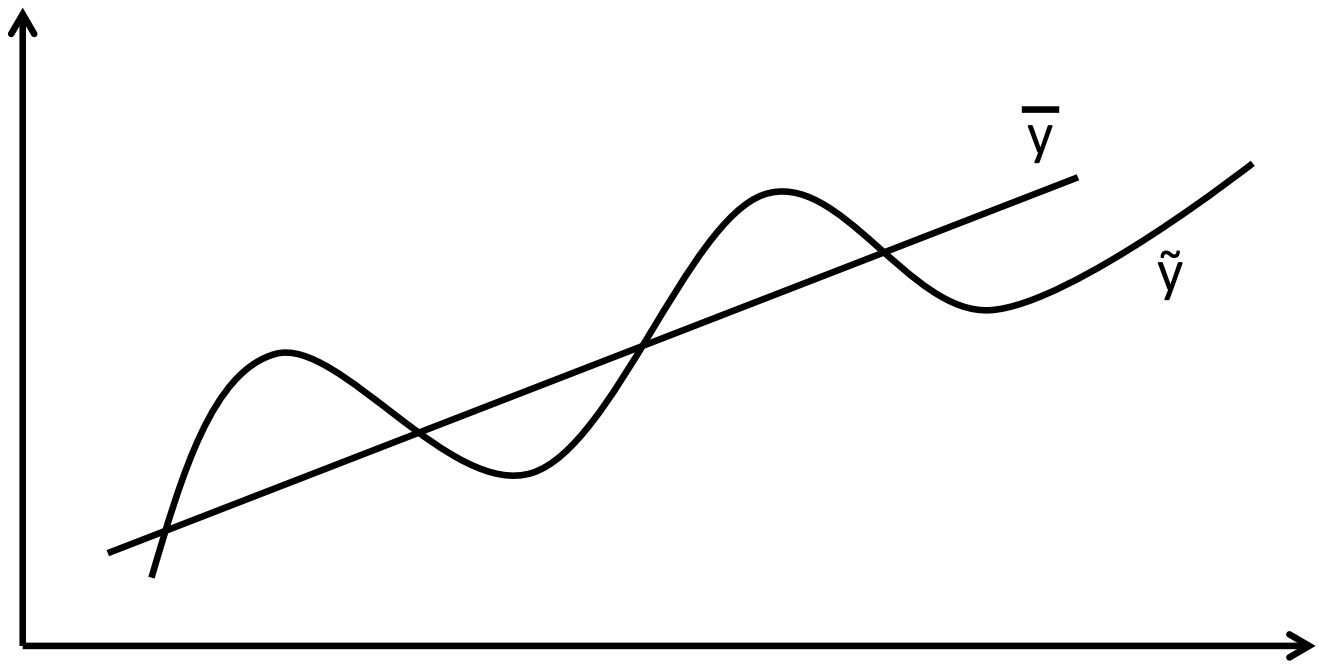
$$\xi = \sum_{i=1}^{N} \left( (\bar{y}_i - y_i)^2 + \frac{1}{K-1} \sum_{j=1}^{K} (\hat{y}_{i,j} - \bar{y}_i)^2 \right) \tag{5}$$

To compute that equation we need to have multiple models. However, in practice we only have one. The solution is simple: bootstrap your data multiple times, and build a model on each bootstrap sample. Here are the steps in computing the equation:

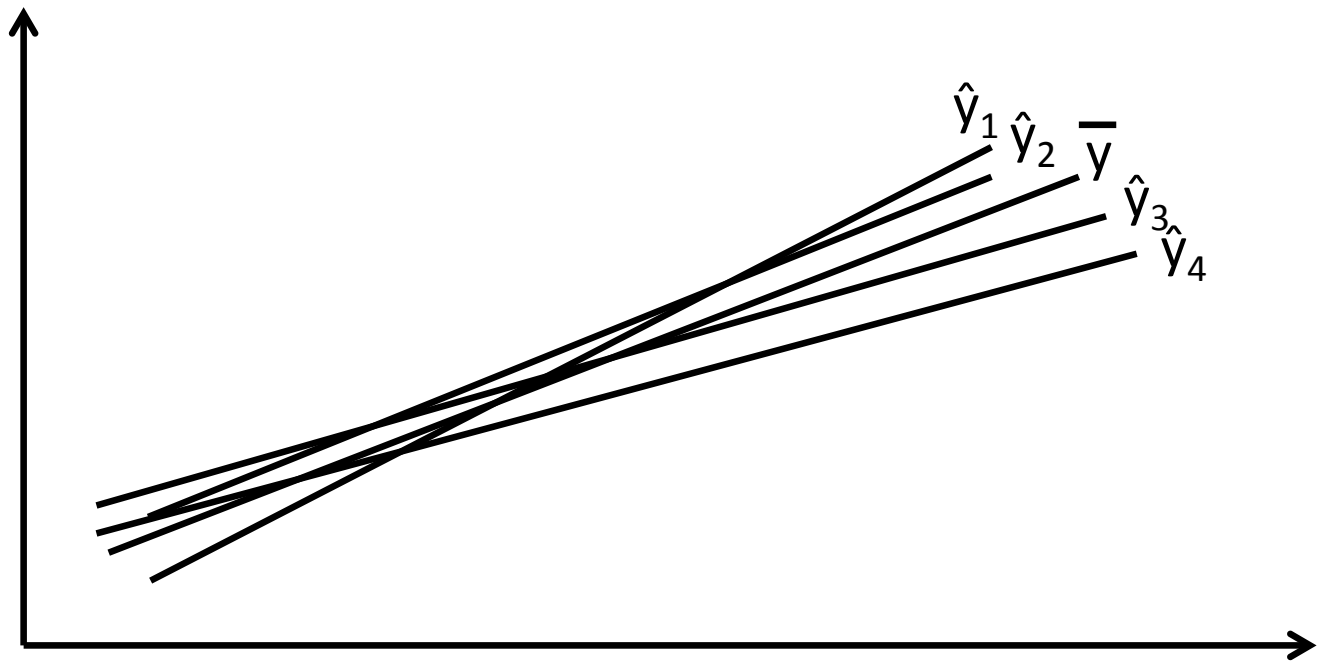
1. Make $K$ bootstrap samples (keep some data out-of-bag, i.e., set some data aside, e.g., 20% )
2. Learn a model on each bootstrap sample
3. For each model predict $\hat{y}$ on out-of-bag data. We now have, for each instance, one $y$ and $\hat{y}_1, \ldots, \hat{y}_K$.
4. Compute $\bar{y}$
5. Compute bias
6. Compute variance
7. Plot bias and variance. We will see how we can use this practically. For now we focus on understanding the nature of bias and variance.

We have set $\epsilon$ to 0. If we have multiple data points with the same $x$-values then we can estimate $\epsilon$. We could also estimate $\epsilon$ by pooling $y$ from nearby $x$-values. But as I mentioned before, we are not really interested in $\epsilon$ since we cannot do anything about it

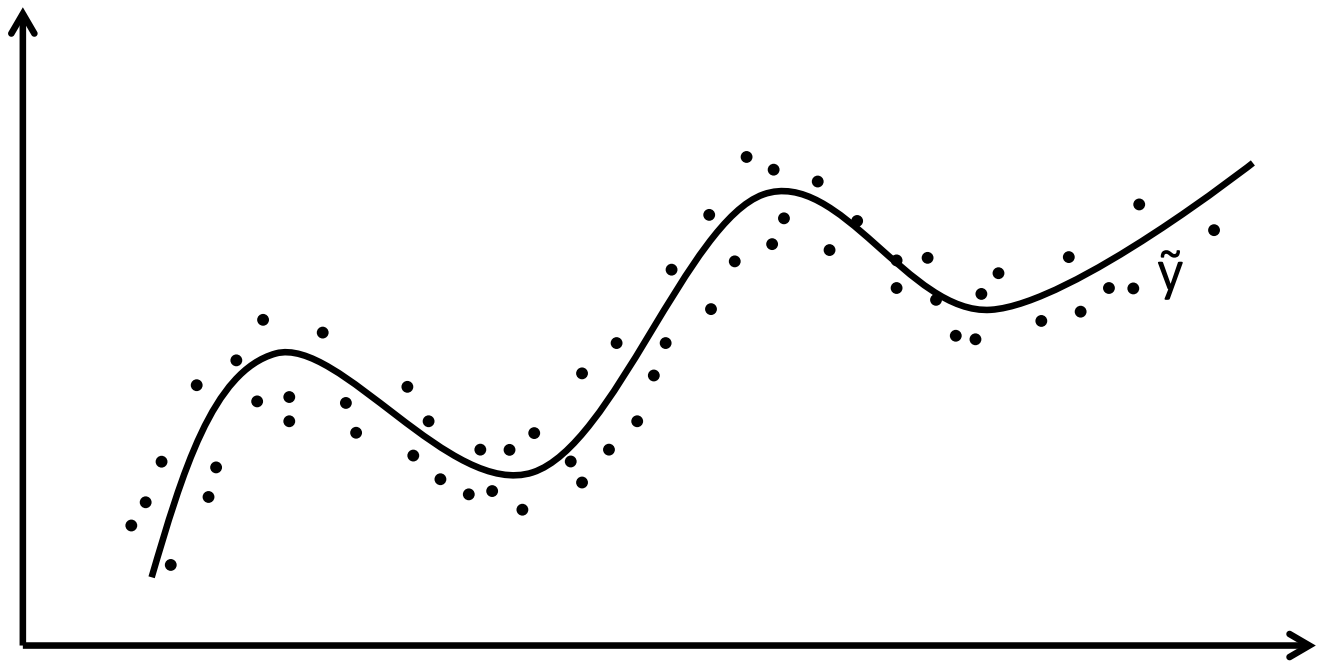
# Bias, given multiple data points



# Variance, given multiple data points

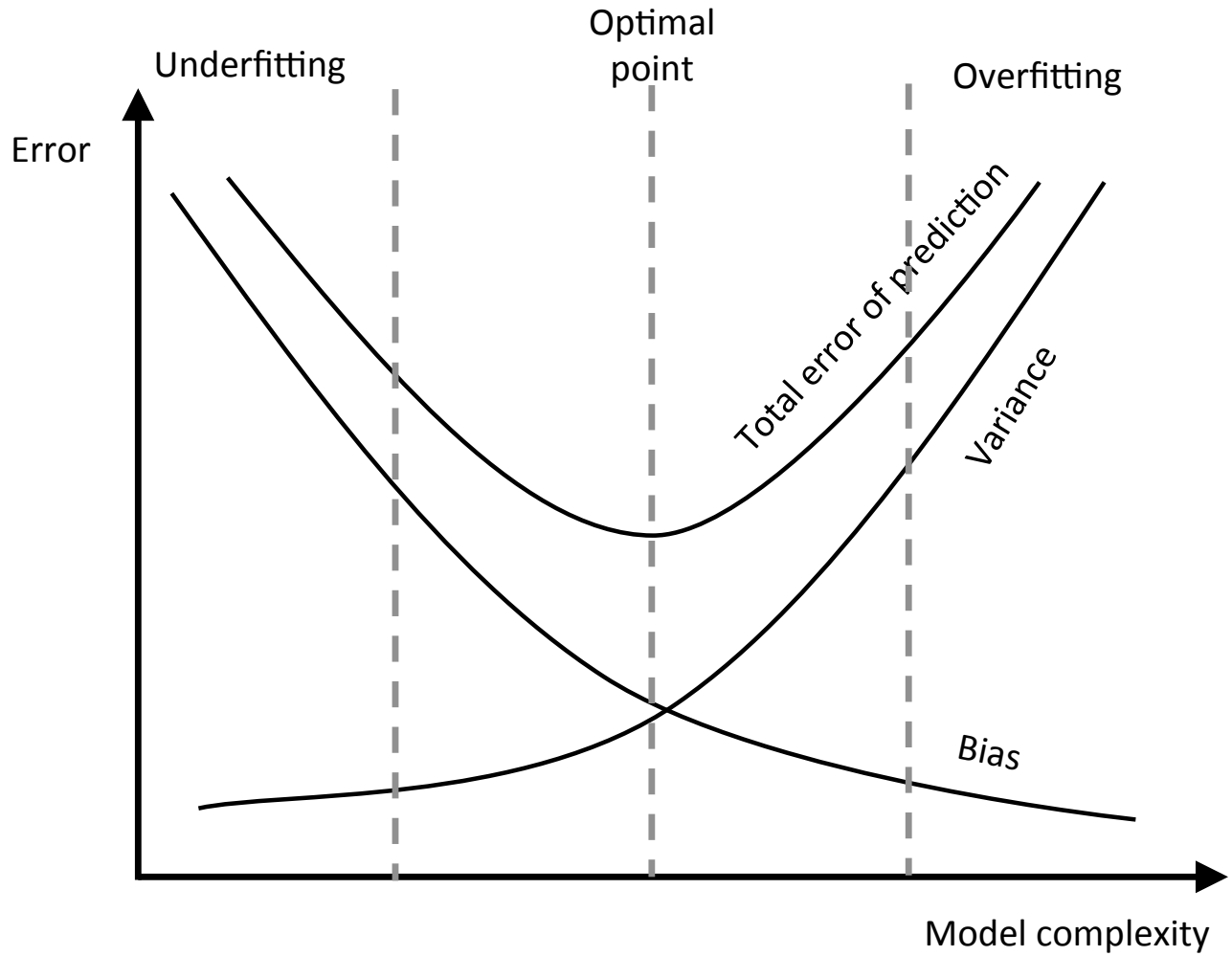


# Noise, given multiple points



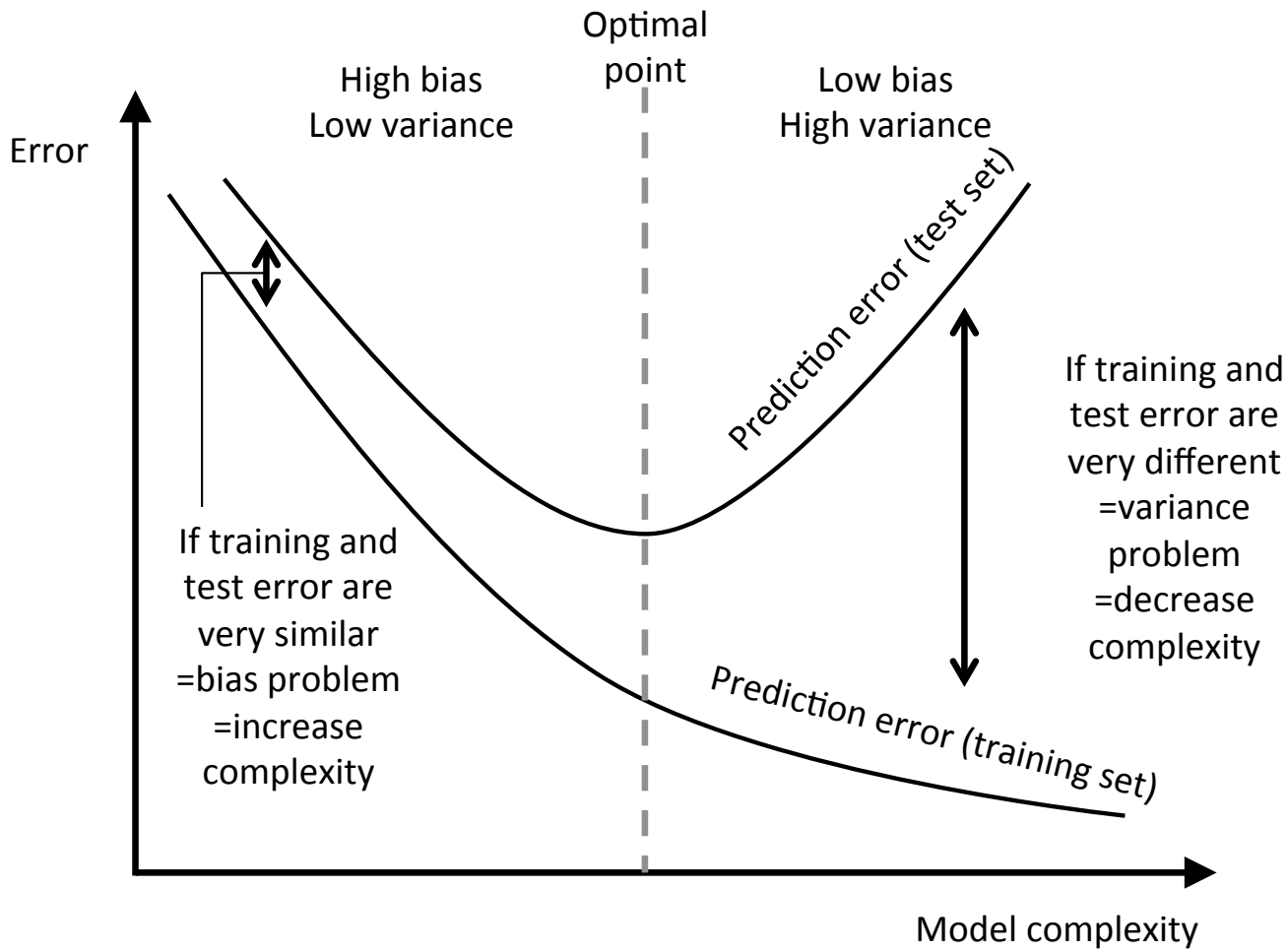
# How to use the bias-variance practically?

The problem with bias and variance is that if you decrease the former, you will increase the latter and vice versa. Hence the name 'bias-variance tradeoff'.



# Diagnosing bias and variance problems

We could plot the total error for a range of complexities, just as in the previous figure. The problem with this is that, depending on how many points you want on your curve, this is very inefficient. A more efficient way:



# Probably Approximately Correct

- With our refreshed understanding of the bias-variance tradeoff, we want to get back to our requirement that we need complex functions (requiring complex models) and large quantities of data if we are to use neural networks. Neural networks are great at fitting complex relationships. If we are dealing with a linear relationship we might just as well use a linear model.
- Assuming we have a complex relationship, and thus need a complex model, we will show here that we need large volumes of data. Intuitively, we say that generalization error (the error on the test test) will be high if we have a complex model trained on small data, because the model will have large variance (i.e., overfit).
- We could show this by computing the bias and variance of two models, one trained with a lot of data, and one with very few data points, and plot the four lines. The variance of the model trained with few data points will be higher than the variance of the model trained with many data points.
- Alternatively, there is much theoretical work trying to bound the generalization error using the training error, and it is interesting to look at some of this work to understand how the variance is impacted by training set size. On the next slide we will look at one bound based on Probably Approximately Correct learning theory.

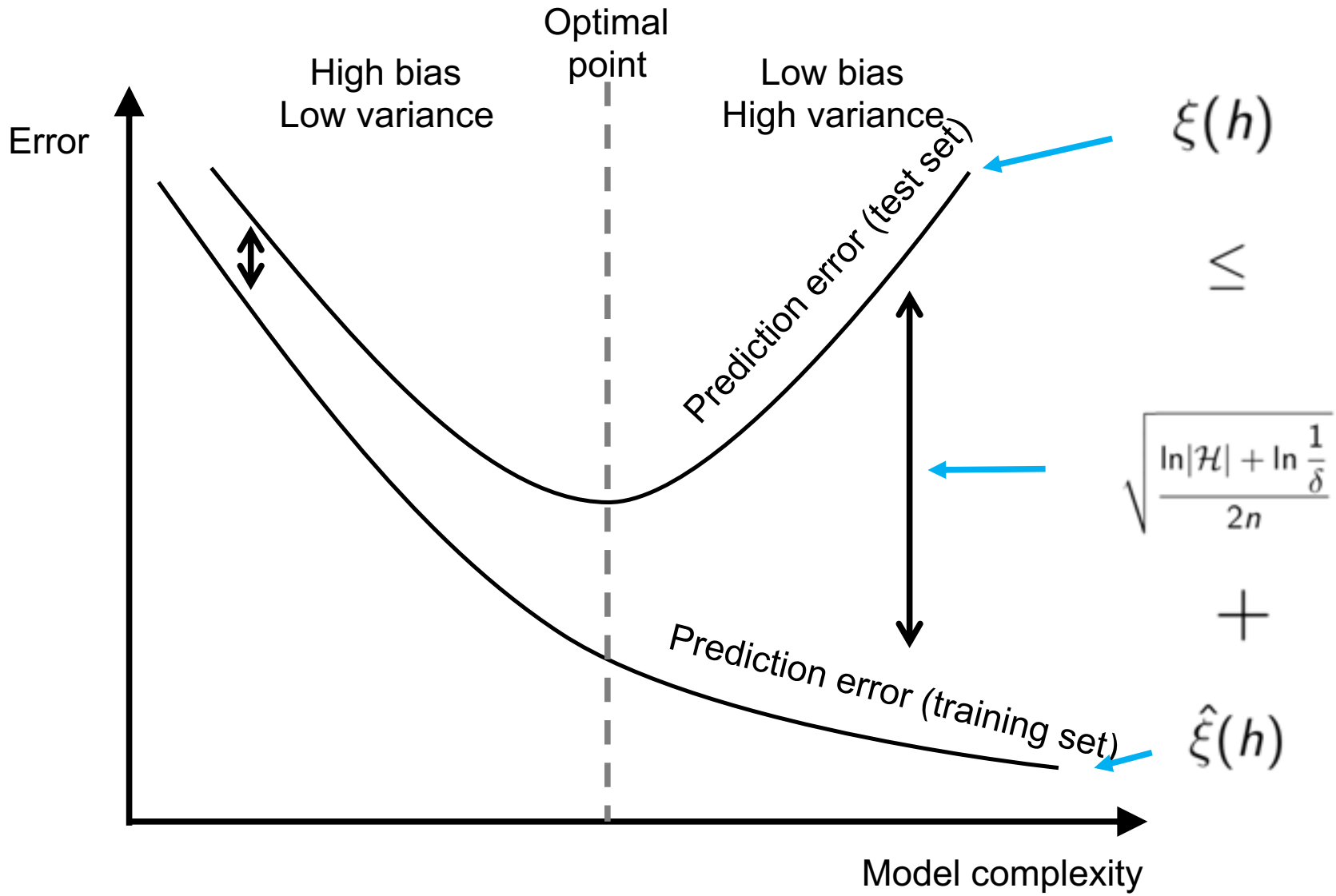
# Probably Approximately Correct

Based on PAC (Probably Approximately Correct) learning theory we know that for a given confidence level (probability) $0 < (1 - \delta) < 1$, larger volumes of data ($n$) will result in lower variance of a hypothesis (model) $h \in \mathcal{H}$ and therefore more data tightens the bound on the generalization error (approximate correctness) of the model as follows.

$$\xi(h) \leq \hat{\xi}(h) + \sqrt{\frac{\ln|\mathcal{H}| + \ln\frac{1}{\delta}}{2n}}, \tag{6}$$

where $\xi(h)$ is the generalization error (test error), $\hat{\xi}(h)$ is the training error, $\sqrt{\frac{\ln|\mathcal{H}| + \ln\frac{1}{\delta}}{2n}}$ is the variance, and $|\mathcal{H}|$ is the complexity of the model. The inequality tells us that the variance goes up when complexity increases, and that the variance goes down when $n$ increases. Let's gain some more understanding of this bound in the next slide.

# Probably Approximately Correct



# Computational reason

When we have large amounts of data and complex nonlinear problems, there is also a computational reason to use deep learning.

- Neural networks are **trained incrementally**, meaning that they **only need parts of the dataset in memory** (e.g., one record).
- We can think of the training procedure as a **loop construct in which we loop over the data one record at a time, each time updating the parameters of the neural network**.
- This means that we can train neural networks on **relatively small hardware** compared to what would be required for methods such as OLS, random forest, boosting machines, and support vector machines.

Let's compare incremental learning with batch learning in the following slide.

# Computational reason

We cannot drink (learn) directly from the stream because if the stream stops we will not be able to continue drinking. Therefore we set up a lake (data buffer), which is limited by our disk size.

**Algorithm 1** Creating a FIFO (first in, first out) data buffer. A data buffer is a table in a database stored on disk.

```
1: for every new instance that arrives do
2:     Append instance to buffer
3:     if maximum buffer size has been reached then
4:         Remove oldest instance from buffer
5:     else
6:         Do nothing
7:     end if
8: end for
```

# Computational reason

**Algorithm 2** Learning from buffer in batch mode

```
1: for every week do
2:     Load all instances into RAM
3:     Compute model accuracy on all instances
4:     if model accuracy is below threshold then
5:         Train model from scratch on all instances
6:     else
7:         Do nothing
8:     end if
9: end for
```

Algorithm 2 has the following features:

- Learning happens when data streams in slowly or quickly.
- Intra-week accuracy may drop below acceptable levels. Shortening the model training cadence will increase computational costs.
- Computational costs (RAM and CPU) spike when the model is trained. Powerful hardware is needed to load the records in memory and learn the model. Reducing the buffer's size will result in worse models.

# Computational reason

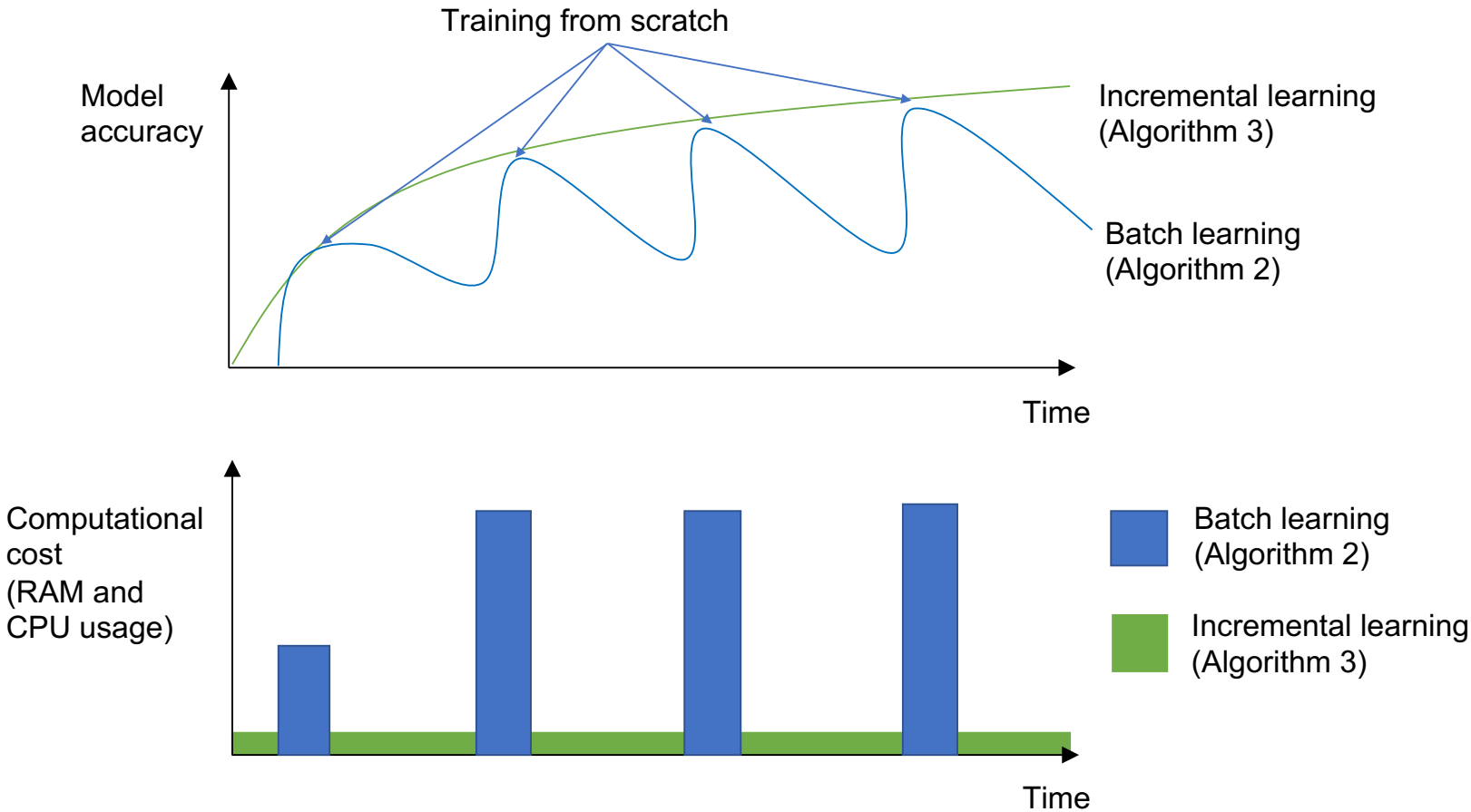
**Algorithm 3** Learning from buffer in incremental mode

```
1: while TRUE do
2:     Load one instance from buffer into RAM
3:     Update parameters of model
4: end while
```

Algorithm 3 has the following features:

- Learning happens when data streams in slowly or quickly.
- Accuracy remains high all the time. New instances are prioritized.
- Computational costs (RAM and CPU) are constant and low across time. Inexpensive hardware is sufficient.

# Computational reason



# Outline

# When should I use deep neural networks? Situation 2

**When we have**

- large amounts of data (and/or data is streaming in really fast) **and**
- images/videos and/or text as inputs (potentially in addition to numerical features).

**Why those two requirements?**

Specialized neural network architectures exist for text (e.g., sequence models such as recurrent neural network and attention) and images (e.g., convolutional neural networks). No other algorithm can beat neural networks in these two application domains in terms of accuracy and efficiency. We will discuss this in detail later in the course.

# Outline

# How can I look inside the deep neural network black box?

- Is there such as thing as a black box method?
- The effects of features on the predictions of any learning method can be described with simulation methods such as
  - variable importance measures, and
  - partial dependence plots.

# Variable Importance Measures

1. Learn a model
2. Make a prediction for all instances and compute how well the model performs on an arbitrary metric. Call this $performance_{before}$.
3. For a specific predictor $x_p$, compute the importance as follows:
   - 3.1 Permute all values of $x_p$
   - 3.2 Make a prediction for all instances and compute how well the model performs on the metric used before. Call this $performance_{after}$.
   - 3.3 Importance = $performance_{before}$ - $performance_{after}$
4. Return to step 2 and repeat for other predictors of interest

# Partial dependence plots

Partial dependence plots are created by the following procedure:

1. Learn a model
2. For a specific predictor $x_p$, create a partial dependence plot as follows:
   - 2.1 Let $v$ be the distinct values of a predictor $x_p$
   - 2.2 For each value of the $v$ values
     - 2.2.1 Create a novel data set where $x_p$ only takes on that value and leave all other variables untouched
     - 2.2.2 Predict the response for all instances in that novel data set using the model
     - 2.2.3 Calculate the mean of the predictions yielding one single value for all instances
   - 2.3 Plot the means of the predictions against the $v$ values
3. Return to step 2 and repeat for other predictors of interest

It is important to note that partial dependence plots represent the effect of a predictor or a subset of predictors after accounting for the effects of the other predictors. Analogously to a multiple linear regression, the coefficient of predictor $x_1$, obtained from regressing $y$ on all $x_j$, measures the effect of $x_1$ accounting for the effects of the other variables.

# Outline

# Why should I use deep neural networks?

Neural networks **optimize** parameters with **incremental learning**, meaning they

- analyze data in constant time per instance; the processing time per instance is the same, if we are analyzing 100 instances or 1 million instances; this means we know a-priori how long it will take for the algorithm to get through all the data,
- are anytime algorithms, in that parameters can be queried at any time, and one does not have to wait until all instances have been processed; the code will leave a trail of models and we will have a first model quickly,
- have constant memory; the RAM requirements do not grow as more data is analyzed; memory usage is known immediately when learning starts, and it will remain the same while training progresses; no risk of running out of memory.

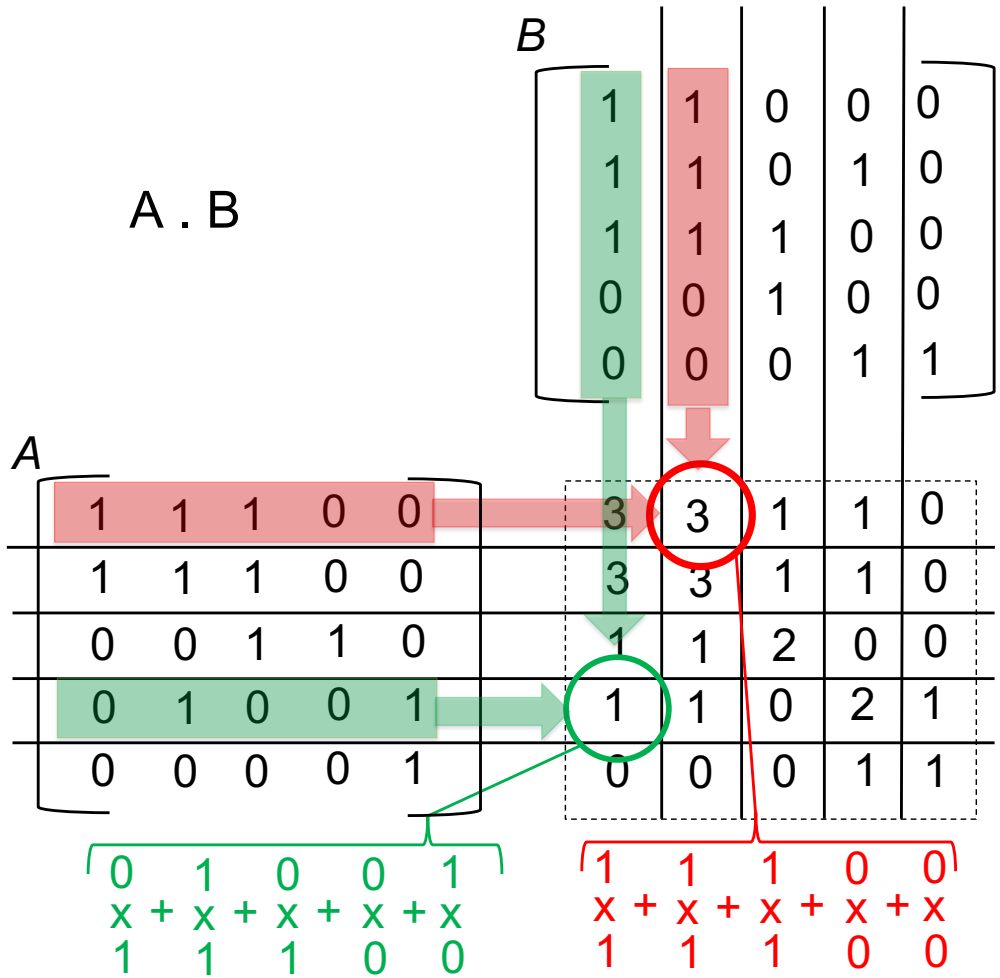
We therefore say that they are **scalable, resource-bounded, and resource-predictable**: neural networks have a fixed throughput and a fixed footprint.

Neural networks can **automatically represent** a wide variety of functions. This means that

- a user does not have to do any detective work and manually specify additional terms in the model (e.g., quadratic terms)

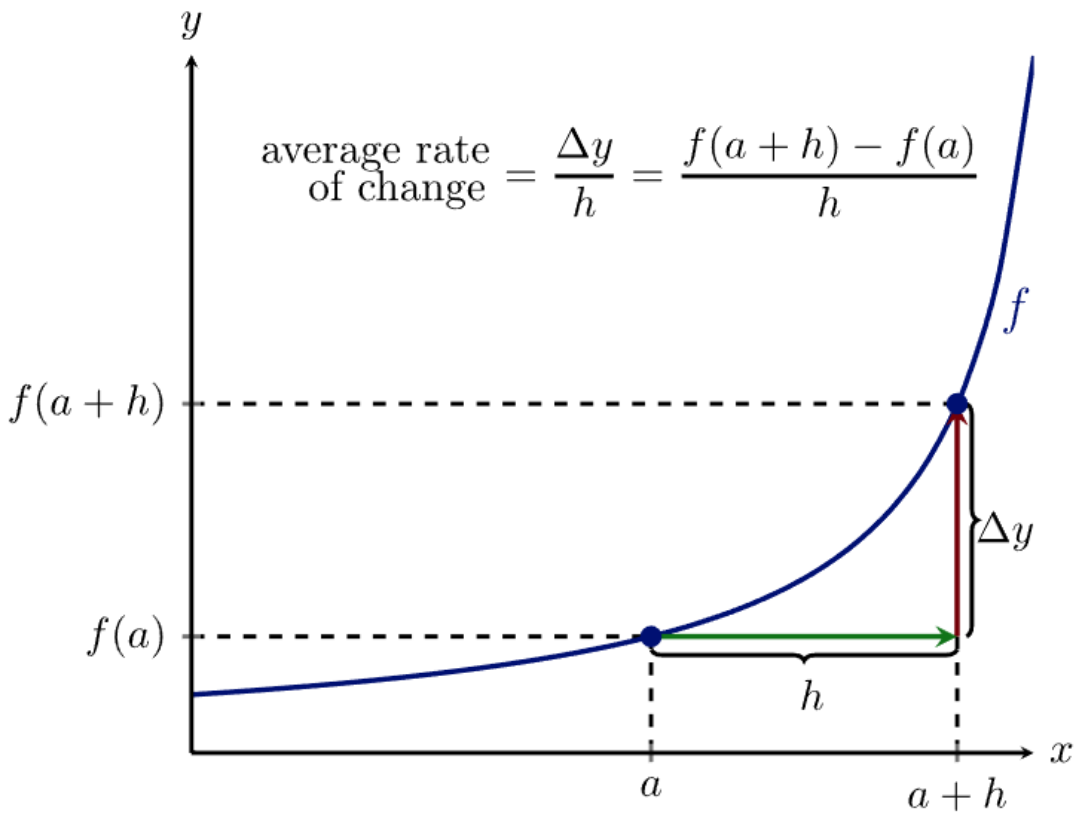
# Outline

# Matrix multiplication, refresher



# Numerical derivatives, refresher

How do we measure the 'impact' of something (e.g., changing a weight) on something else (e.g., the error of a model)? With impact I mean, change in something for one unit change in something else. Derivatives.




`https://ximera.osu.edu/mooculus/calculus1/derivativeAsAFunction/digInTheDerivativeAsAFunction`

# Numerical derivatives, refresher

$$impact = \frac{\Delta f(x)}{\Delta x} = \frac{f(a+h) - f(a)}{(a+h) - a} = \frac{f(a+h) - f(a)}{h} \tag{7}$$

But our optimization routine wants the instantaneous impact (i.e., h goes to 0), because otherwise it might jump over the optimal point :

$$instantaneous\ impact = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \tag{8}$$

There are two ways: **analytical derivatives** and **numerical derivatives**.

# Numerical derivative

$$\frac{Error(w + 0.00000000001) - Error(w)}{0.00000000001} \tag{9}$$

or better:

$$\frac{Error(w + 0.00000000001) - Error(w - 0.00000000001)}{2 * 0.00000000001} \tag{10}$$

Advantages

- Easy to do, no math required

Disadvantages

- Not really instantaneous impact, as 0.00000000001 is not h $\rightarrow$ 0. We can't make it smaller, because this will lead to inaccuracies due to rounding errors: the change in the numerator might be too small to be observable.
- Slow. Every derivative requires two evaluations of the objective function

# Analytical derivative

Advantages

- More accurate, as $h \to 0$
- Fast, requires one evaluation. Especially faster if the analytical derivatives are simpler than the objective function itself.

Disadvantages

- Need to know the math

We will learn several analytical derivatives during the course, because it is important to understand how they impact training neural networks. Luckily, TensorFlow automatically does analytical derivatives for us with something that is called *autodiff*.

# Learning rule

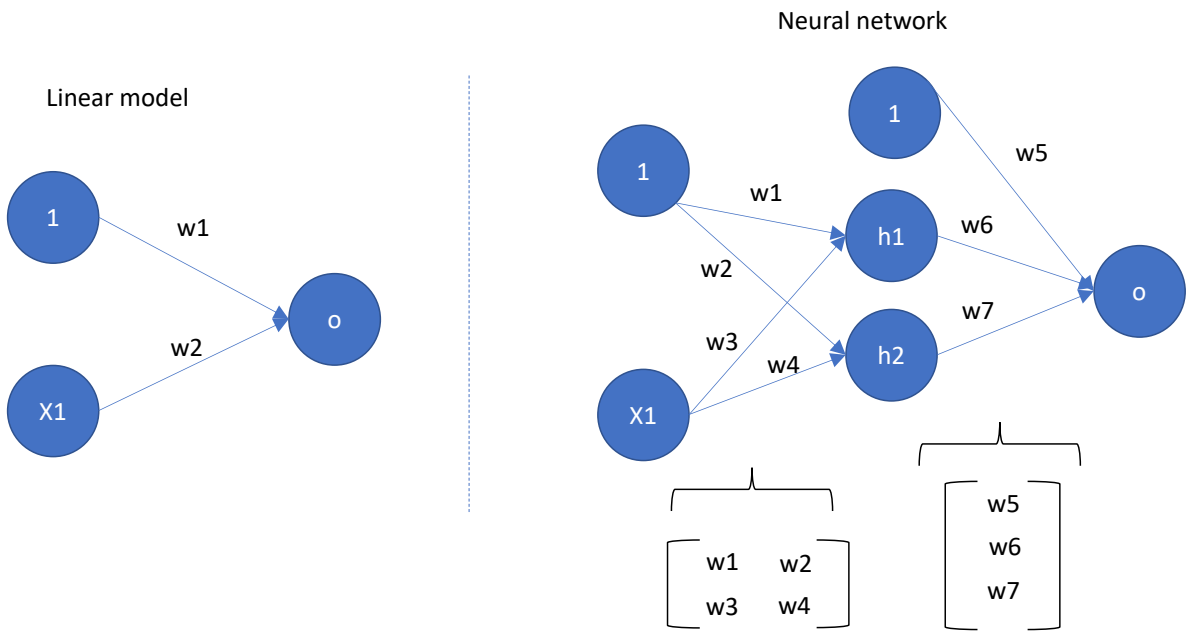
$$w_{new} = w_{old} - \eta \frac{\Delta error}{\Delta w} \tag{11}$$

- If we increase $w$ by a tiny bit (i.e., $\Delta w > 0$),
  - and the error becomes bigger (i.e., $\Delta error > 0$),
  - for example, $\frac{\Delta error}{\Delta w} = \frac{Error(w+0.00000000001) - Error(w)}{0.00000000001} = \frac{0.02-0.01}{0.00000000001} > 0$,
  - then we need to decrease $w$,
  - and that is exactly what Equation 11 does.
- If we increase $w$ by a tiny bit (i.e., $\Delta w > 0$),
  - and the error becomes smaller (i.e., $\Delta error < 0$),
  - for example, $\frac{\Delta error}{\Delta w} = \frac{Error(w+0.00000000001) - Error(w)}{0.00000000001} = \frac{0.005-0.01}{0.00000000001} < 0$,
  - then we need to increase $w$,
  - and that is exactly what Equation 11 does.

# Code neural network from scratch

We will code a neural network from scratch, with minimal math. This constitutes using nothing more than

- numerical derivatives,
- the learning rule, and
- matrix multiplication.




See *NeuralNetFromScratch.py*.