COMPSCI 389
Introduction to Machine Learning

Days: Tu/Th. **Time:** 2:30 – 3:45 **Building:** Morrill 2 **Room:** 222

Topic 12.2: Supervised Learning Review

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

What is machine learning (ML)?

- Subfield of *artificial intelligence* (AI)

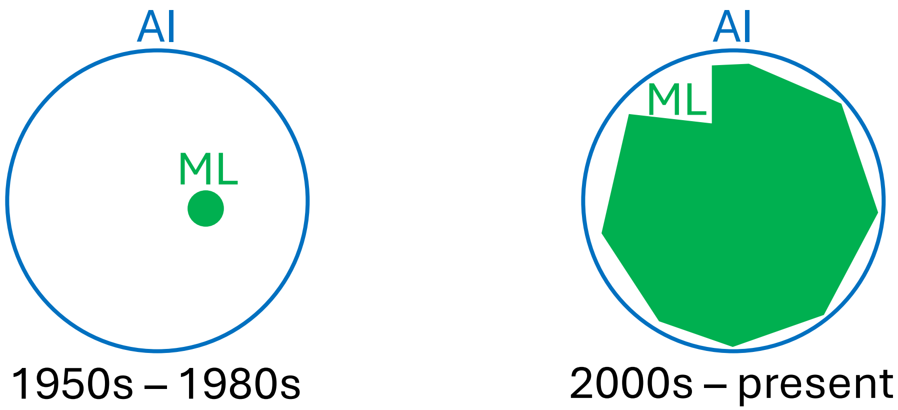
  “AI is a field concerned with intelligent behavior in ~~artifacts~~ agents*.”*

  – Nilsson 1998

Like math, physics or theology

- AI is not a thing/object.
- The thing/object using AI methods is called an agent.
  - Agent: Something that acts, from Latin *agere*, which means “to do.”
  - E.g., a robot or software program

ML is a subfield of AI



- ML is a subfield of AI “*concerned with the question of how to construct ~~computer~~ programs that automatically improve with experience*.” [Tom Mitchell, 1997]
- Improve = learn
- Experience = data
- Computer = unnecessary

Data & Supervised Learning

- Different subfields of ML assume access to different kinds of data.
- During the first part of the course, we will focus on **supervised learning** problems.
- These are problems where the data is a set of points, and so it is called a **data set** or **dataset.**
- Each point consists of a pair of **inputs** and **outputs**.
- Given a data set of such input-output pairs, a supervised learning algorithm learns to predict the output given the input, even for points not in the data set.

Data Set Notation

- X: **Input** (also called **features**, **attributes**, **covariates**, or **predictors**)
  - Typically, X is a vector, array, or list of numbers or strings.
- Y: **Output** (also called **labels** or **targets**)
  - Typically, Y is a single number or string.
- An input-output pair is (X, Y).
- Let n, called the **data set size** or **size of the data set**, be the number of input-output pairs in the data set.
- Let (X_i, Y_i) denote the i^{th} input output pair.
- The complete data set is
$$(X_i, Y_i)_{i=1}^{n} = \left((X_1, Y_1), (X_2, Y_2), \dots, (X_n, Y_n)\right).$$

Feature Types

- **Numerical**
    - **Continuous**: Features that can take any value in a range, like temperature or velocity.
    - **Discrete**: Features that take a countable number of distinct values, like the number of cats a person owns. (**Binary** features are a special case.)
- **Categorical** (discrete, but not numbers)
    - **Nominal**: Unordered categories like colors (red, green, blue) or genre (drama, comedy, science fiction, etc.).
    - **Ordinal**: Categories with a specific order like educational level (high school, bachelor’s, master’s) or military rank (private, specialist, corporal, etc.)
- **Text/String**
- **Image**
- **Other**

Feature Types

- Non-numerical features are often converted into numerical features to make them easier to work with.
    - Categorical features map to integers: “Sunday”→0, “Monday”→1, “Tuesday”→2, etc.
    - Images can be converted to sequences of (r,g,b) values describing each pixel.
    - Text can be converted to discrete or continuous features
        - Discrete: Each word (or part of a word) maps to a unique integer.
            - Each basic unit of text (word, character, or subword) is called a **token**.
        - Continuous: Each word can be mapped to a vector of real numbers. This is called a **word embedding**. Ideally, similar words are mapped to similar vectors of numbers. Word embeddings are themselves learned from data.

Regression and Classification

- Within supervised learning, recall that a data set is a set of input-output pairs (X, Y).
- **Regression**: Y is a continuous number.
  - Multivariate Regression: Y is a vector. That is, $Y \in \mathbb{R}^m$ and $m > 1$.
- **Classification**: Y is categorical (mapped to an integer).
  - Binary Classification: $Y \in \{0,1\}$ or $Y \in \{-1,1\}$.
  - Multi-Class Classification: $Y \in \{0,1, \ldots, k\}$.

Nearest Neighbor

- A particularly simple yet effective ML algorithm based on the core idea:

  When presented with a query, find the data point (row) that is most similar to the query and give the label associated with this most-similar point as the prediction.

- We can map this to fit/predict functions:
  - `fit`: Store the data
  - `predict`: For each query row do the following
    - Loop over each row in the training data, computing the Euclidean distance between the query and the row.
    - Create an array holding the labels from the rows with the smallest distance to the query feature vector (often just one element).
    - Return an arbitrary (e.g., random) element of the array.

Evaluation Metrics (Regression)

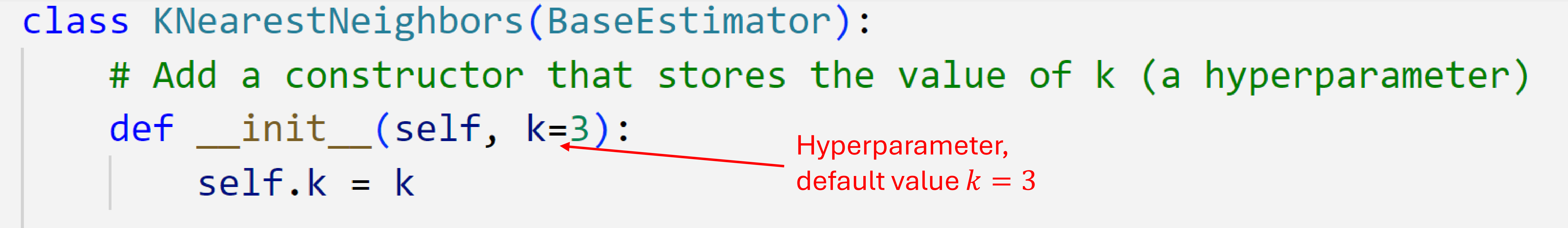
- Mean Error: $\frac{1}{n}\sum_{i=1}^{n} y_i - \hat{y}_i$
  - Rarely what you want.
  - Allows positive and negative errors to cancel each other out.
- Mean Squared Error (MSE): $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
  - Very common choice.
  - Gives a higher weight to larger errors, making it sensitive to outliers. It’s useful when large errors are particularly undesirable.
- Root Mean Squared Error (RMSE): $\sqrt{\mathrm{M}SE}$
  - Has the same units as the target variable (unlike MSE).

Evaluation Metrics (Regression, cont.)

- Mean Absolute Error (MAE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
  - Like MSE, but with less emphasis on outliers.
- R-squared (R^2): $1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$, where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.
  - Also called the *coefficient of determination*.
  - Indicates the proportion of the variance of the dependent variable (labels) that is predictable from the independent variables (predictions).
  - Larger is better (maximum possible is one).

k-Nearest Neighbors (k-NN)

- **Idea**: Average the labels of the k nearest points
- Pseudocode:
    - Find the k nearest neighbors to the query point.
        - Called the “nearest neighbors”
        - If you will run many queries, consider using a data structure like a KD-Tree to find the nearest neighbors
    - Set the prediction to be the average label of these k nearest neighbors.
- Code:



```python
class KNearestNeighbors(BaseEstimator):
    # Add a constructor that stores the value of k (a hyperparameter)
    def __init__(self, k=3):
        self.k = k
```

Hyperparameter, default value $k = 3$

Weighted k-Nearest Neighbor

- Let $\left(x_i^{NN}, y_i^{NN}\right)$ be the i^{th} nearest neighbor
- Let w_i be the weight associated with this point
    - We consider only non-negative weights: $w_i \geq 0$.
    - We describe how w_i can be computed on future slides.
- Weighted k-NN predicts the label:

$$\hat{y} = \frac{\sum_{i=1}^{k} w_i y_i^{NN}}{\sum_{j=1}^{k} w_j}$$

- This is equivalent to:

$$\hat{y} = \sum_{i=1}^{k} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN}$$

Why do we divide by the sum of the weights?

- So that the weights sum to one.
- This keeps the prediction at the same “scale” as the labels.
- Example: If $k = 2$, $w_1 = 1$ and $w_2 = 1$, and the division by the sum of weights is dropped.
    - The prediction is $2\times$ too big!
- Dividing by the sum of the weights makes this a *weighted average*.

Gaussian Kernel

- The re-scaled *probability density function* (PDF) of a normal distribution.
    - PDF of a normal distribution

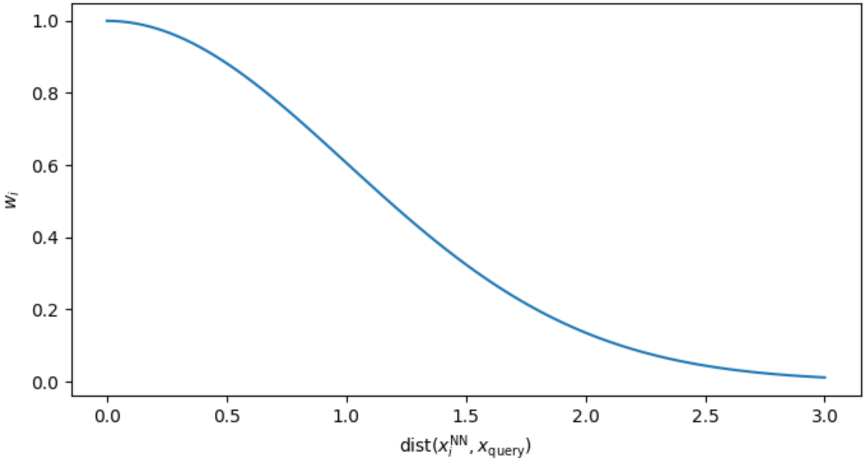
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

    - Mean $\mu = 0$
    - Standard deviation σ (a hyperparameter)
- Normalizing the weights makes the constant $\frac{1}{\sigma\sqrt{2\pi}}$ cancel out in each weight. Hence:

$$w_i = e^{-\frac{x^2}{2\sigma^2}}$$

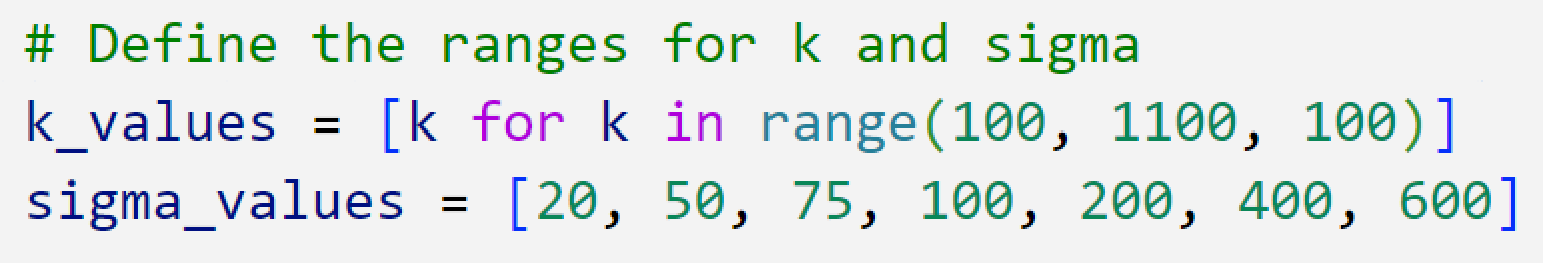
- We use $x = \mathrm{dist}\left(x_i^{NN}, x_{\mathrm{query}}\right)$ giving:

$$w_i = e^{-\frac{\mathrm{dist}\left(x_i^{NN}, x_{\mathrm{query}}\right)^2}{2\sigma^2}}$$



Tuning Hyperparameters

- How should we set k and σ?
- **Idea**: Enumerate a “grid” of possible values.



```
# Define the ranges for k and sigma
k_values = [k for k in range(100, 1100, 100)]
sigma_values = [20, 50, 75, 100, 200, 400, 600]
```

- Try all possible combinations of values of k `in k_values` and σ in `sigma_values`.
    - If plotted as points where the horizontal axis is k and the vertical is σ (or *vice versa*), the points would form a grid.
    - Hence, called “Grid Search”
- Select the values that result in the best evaluation

Tuning Hyperparameters

- Grid search is common due to its simplicity.
- Research suggests that randomized searches may be more principled.
  - Randomly sample each hyperparameter from some distribution
  - Typically run for some fixed number of hyperparameter settings

Train/Validation/Test Sets

- Validation sets are often used to automatically tune hyperparameters.
- The data is split into three sets: train, evaluation, and test. The following procedure is then used:
    - For each hyperparameter setting:
        - Train a model using the training data.
        - Evaluate the model using the validation data.
    - Select the hyperparameter settings that achieve the best evaluation on the validation set.
    - Train a model using all the training and validation data and the hyperparameters that achieved the best evaluation.
    - Evaluate the model using the testing set.

Classification with NN-Variants

- NN: No changes needed!
- k-NN: The predicted label comes from a majority vote of the k nearest neighbors.
- Weighted k-NN: Each neighbor’s vote is weighted in the vote.

Mean Squared Error (revisited)

- The MSE is:

$$\mathrm{MSE} = \mathbf{E}\left[\left(Y - \hat{Y}_i\right)^2\right].$$

  - This is a *parameter* or *population statistic*.
- The sample MSE is:

$$\widehat{\mathrm{MSE}}_n = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \quad \text{or} \quad \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i)^2\,.$$

  - This is a *statistic* or *sample statistic*.
  - The “hat” means “an estimate” and the n-subscript indicates it is computed from n samples.
- Our goal is typically to optimize a parameter.
  - We don’t know this parameter’s value.
- In an attempt to achieve this goal, we use sample statistics.
  - We can compute sample statistics from data!

Confidence Interval

- We will use the number of samples and their variance to construct a **confidence interval** for the parameter (e.g., MSE) based on the sample statistic (sample MSE).
- A confidence interval is an interval (range of numbers) that contains a parameter with a specified confidence, $1-\delta$.
- If $[L, U]$ is a $1-\delta$ confidence interval for the mean μ, then
$$\Pr(L \leq \mu \leq U) \geq 1-\delta.$$
- **Question**: What is random in this statement of probability?
- **Answer**: The *confidence interval* is random! It is typically computed from data. Different samples of data result in different lower and upper bounds.

Standard Error

- One common way to obtain a confidence interval is using **standard error.**
- Let $x_1, x_2, \ldots, x_n$ be a sequence of n numbers.
- Let σ be the sample **standard deviation** of this sequence (with Bessel's correction):

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n - 1}},$$

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- The **standard error** is then

$$\mathrm{SE} = \frac{\sigma}{\sqrt{n}}.$$

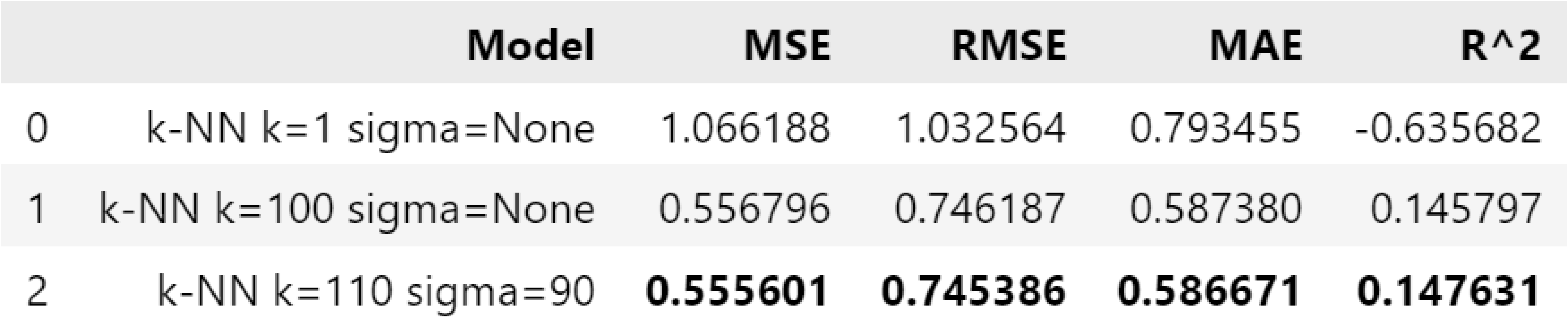
Using Standard Error

- If $X_1, X_2, \dots, X_n$ are n random variables and:
  - The random variables are i.i.d. with mean μ.
  - The random variables are each normally distributed.
  - $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean.
- Then $[\bar{X} - 1.96 \times \mathrm{SE},\ \bar{X} + 1.96 \times \mathrm{SE}]$ is a 95% confidence interval for μ.
- That is:

$$\Pr(\bar{X} - 1.96 \times \mathrm{SE} \leq \mu \leq \bar{X} + 1.96 \times \mathrm{SE}) \geq 0.95.$$

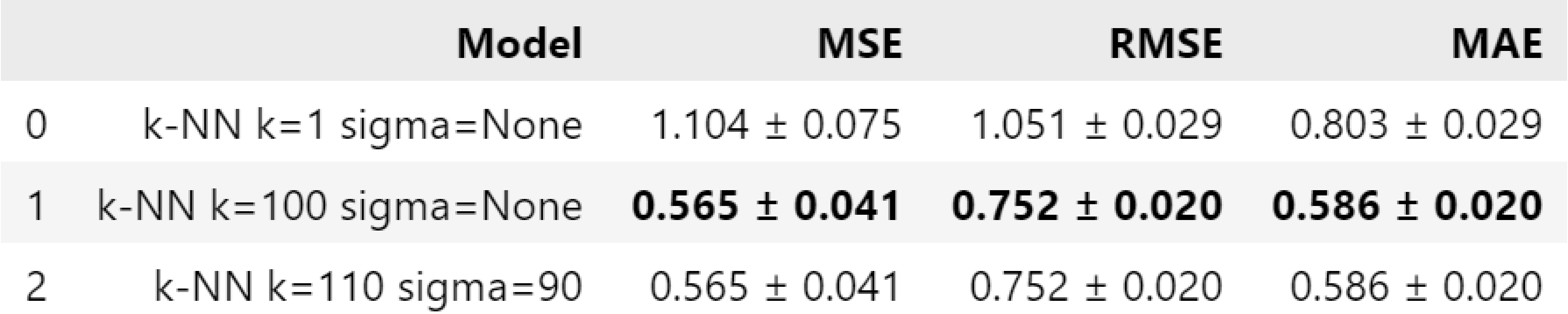
Mean Squared Error (re-revisited)

- MSE: $\mathrm{MSE} = \mathbf{E}\left[\left(Y - \hat{Y}_i\right)^2\right]$.
- Sample MSE: $\widehat{\mathrm{MSE}}_n = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$.
- Let $Z_i = \left(Y_i - \hat{Y}_i\right)^2$.
- Notice that $\mu = \mathbf{E}[Z_i] = \mathrm{MSE}$, and let SE be the standard error of $Z_1, Z_2, \dots, Z_n$.
- So, $\widehat{\mathrm{MSE}}_n \pm 1.96 \times SE$ is a 95% confidence interval for the actual MSE (under normality assumptions).
    - Although normality assumptions often false, this gives a rough idea of how much the sample MSE can be trusted.



	Model	MSE	RMSE	MAE	R^2
0	k-NN k=1 sigma=None	1.066188	1.032564	0.793455	-0.635682
1	k-NN k=100 sigma=None	0.556796	0.746187	0.587380	0.145797
2	k-NN k=110 sigma=90	**0.555601**	**0.745386**	**0.586671**	**0.147631**

±1.96 × SE



	Model	MSE	RMSE	MAE
0	k-NN k=1 sigma=None	1.104 ± 0.075	1.051 ± 0.029	0.803 ± 0.029
1	k-NN k=100 sigma=None	**0.565 ± 0.041**	**0.752 ± 0.020**	**0.586 ± 0.020**
2	k-NN k=110 sigma=90	0.565 ± 0.041	0.752 ± 0.020	0.586 ± 0.020

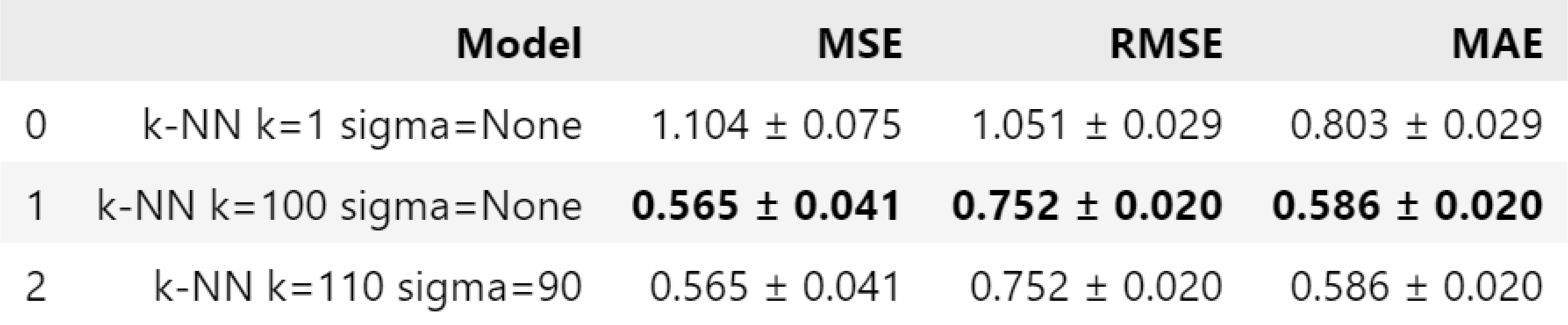
We can be *somewhat* confident that the model learned by NN is worse than the model learned by k-NN ($k = 100$) and weighted k-NN ($k = 110, \sigma = 90$).

We cannot be confidence about k-NN vs weighted k-NN.

Note: Always check for the *meaning* of the ± value! Standard error, standard deviation, and confidence intervals all have very different meanings!

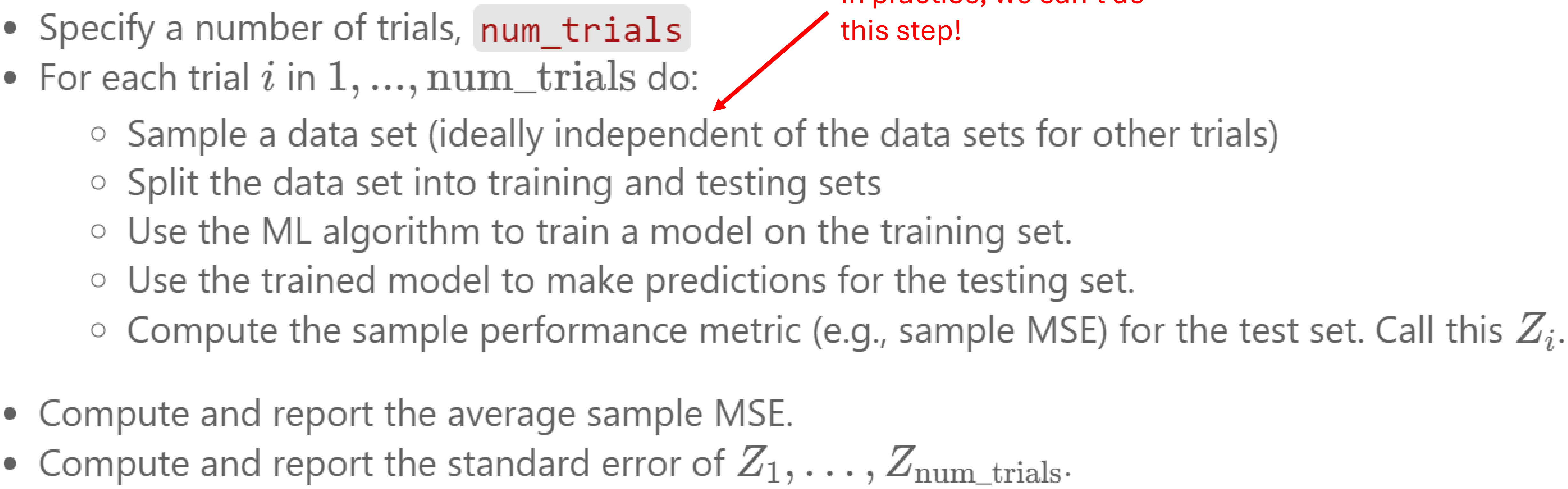
Model Evaluation (Review)

- Often ML texts evaluate models by doing the following:
  - Partition the data into train/test.
  - Train the model on the training data.
  - Evaluate the model on the testing data.
  - Report a performance metric and a number representing the *uncertainty* in this performance metric.
    - Format: performance ±uncertainty



	Model	MSE	RMSE	MAE
0	k-NN k=1 sigma=None	1.104 ± 0.075	1.051 ± 0.029	0.803 ± 0.029
1	k-NN k=100 sigma=None	**0.565 ± 0.041**	**0.752 ± 0.020**	**0.586 ± 0.020**
2	k-NN k=110 sigma=90	0.565 ± 0.041	0.752 ± 0.020	0.586 ± 0.020

Algorithm Evaluation (Ideal)

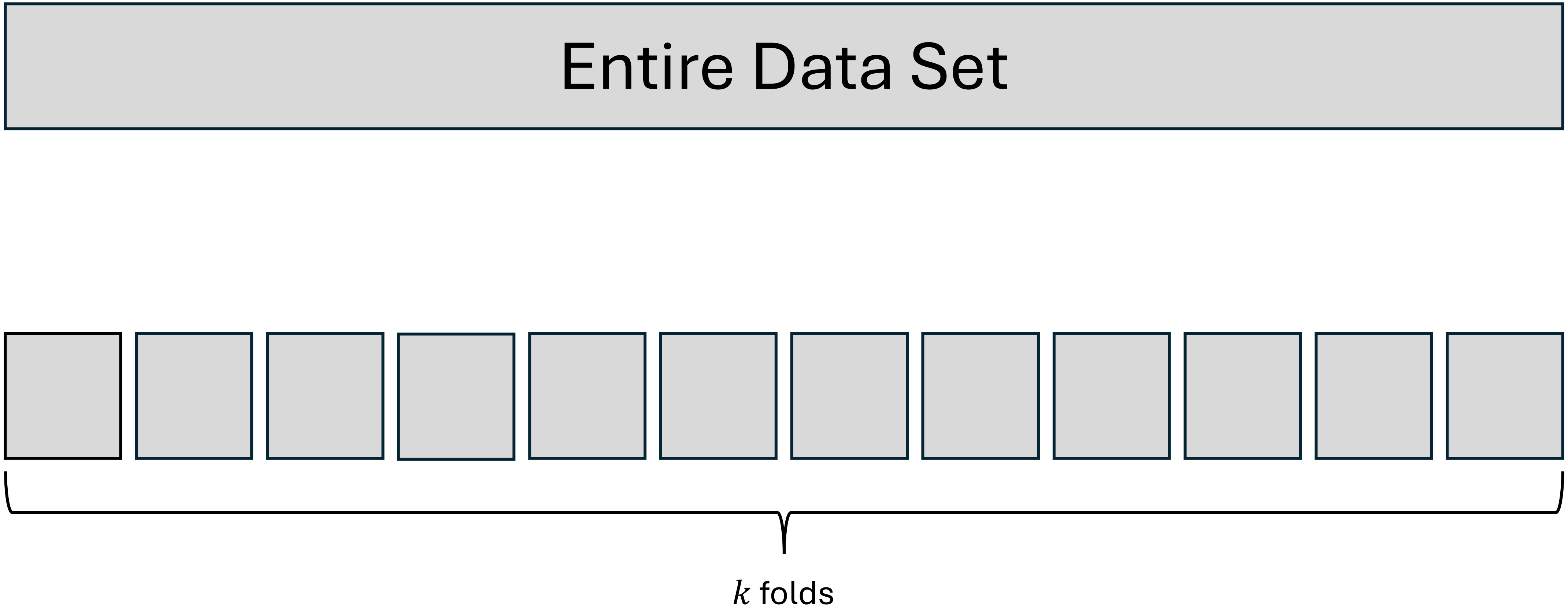


- Specify a number of trials, `num_trials`
- For each trial i in $1, ..., \text{num\_trials}$ do:
  - Sample a data set (ideally independent of the data sets for other trials)
  - Split the data set into training and testing sets
  - Use the ML algorithm to train a model on the training set.
  - Use the trained model to make predictions for the testing set.
  - Compute the sample performance metric (e.g., sample MSE) for the test set. Call this Z_i.
- Compute and report the average sample MSE.
- Compute and report the standard error of $Z_1, \ldots, Z_{\text{num\_trials}}$.

In practice, we can’t do this step!

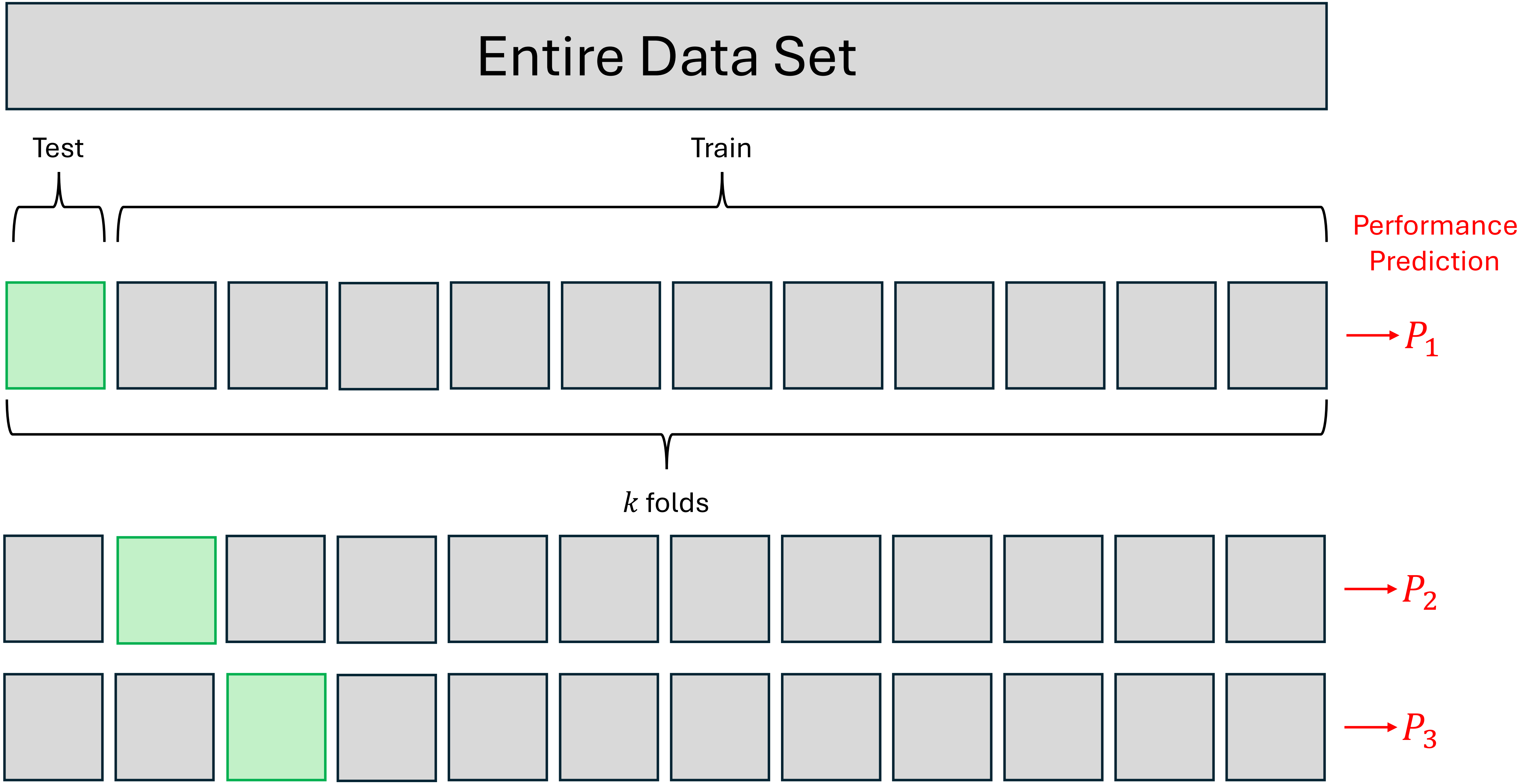
Cross-Validation

- **Idea**: Repeatedly define different parts of the data set to be training and testing data.
  - Different training sets result in different models.
  - The testing set for each model will always be independent of the data used to train the model.
- To do this, we will split the data D into k equally-sized subsets.
  - Each of these subsets is called a *fold*.
  - This k is not related to the k in nearest neighbor.
- We will train on all but one fold and test on the held-out fold.
  - These individual evaluations on test sets containing one fold have high variance!
  - We can average these high-variance evaluations to obtain a better estimate of performance.



Entire Data Set

k folds



Entire Data Set

Test

Train

Performance Prediction

$\longrightarrow P_1$

k folds

$\longrightarrow P_2$

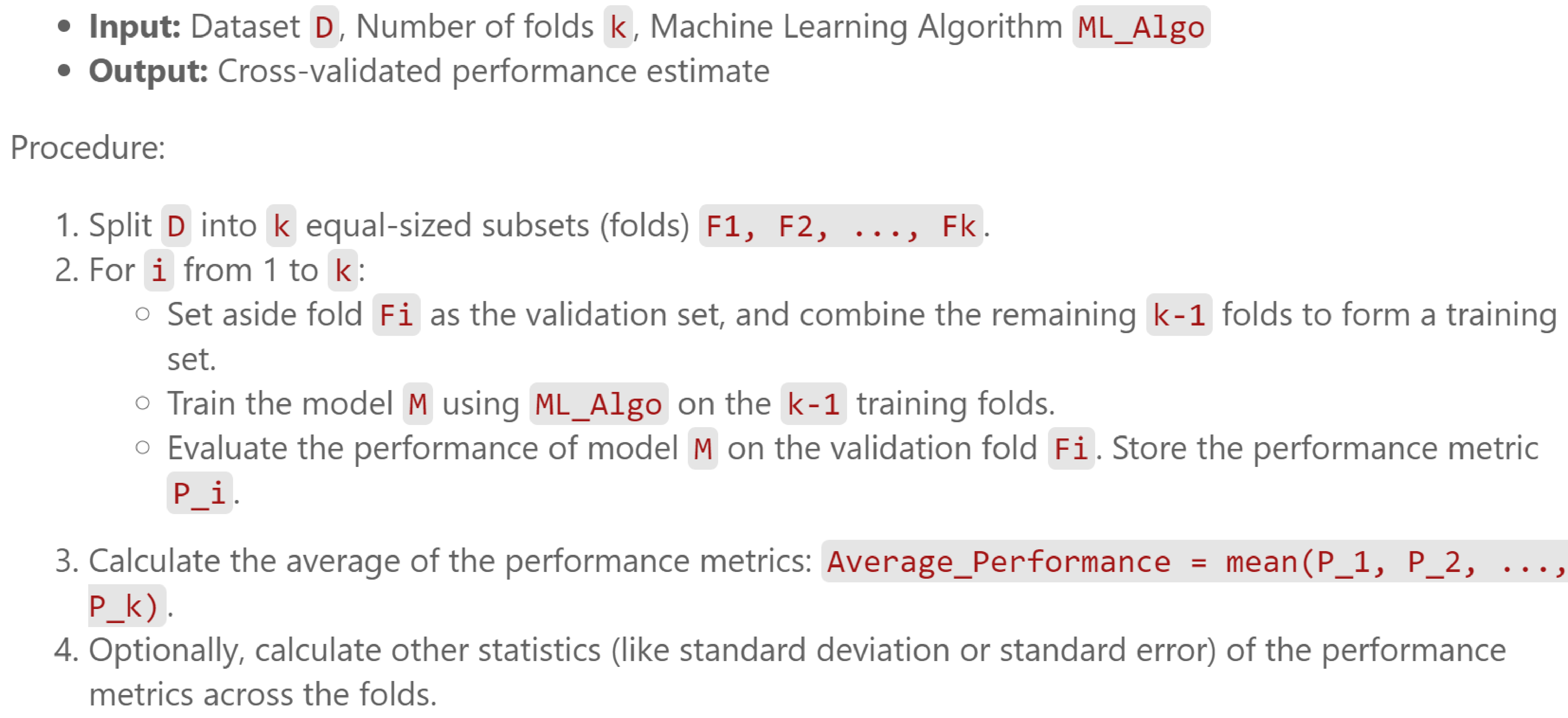
$\longrightarrow P_3$

Repeat for $P_1, \dots, P_k$

Performance Estimate = mean($P_1, \dots, P_k$)

Uncertainty quantification = SE($P_1, \dots, P_k$)

K-Fold Cross-Validation Pseudocode



- **Input:** Dataset `D`, Number of folds `k`, Machine Learning Algorithm `ML_Algo`
- **Output:** Cross-validated performance estimate

Procedure:

1. Split `D` into `k` equal-sized subsets (folds) `F1, F2, ..., Fk`.
2. For `i` from 1 to `k`:
   - Set aside fold `Fi` as the validation set, and combine the remaining `k-1` folds to form a training set.
   - Train the model `M` using `ML_Algo` on the `k-1` training folds.
   - Evaluate the performance of model `M` on the validation fold `Fi`. Store the performance metric `P_i`.
3. Calculate the average of the performance metrics: `Average_Performance = mean(P_1, P_2, ..., P_k)`.
4. Optionally, calculate other statistics (like standard deviation or standard error) of the performance metrics across the folds.

Leave-One-Out (LOO) Cross-Validation

- The number of folds equals the number of points in the data set.
- Each test set contains only a single point!
- Provides the best estimates of performance.
- Often too computationally intensive to perform.

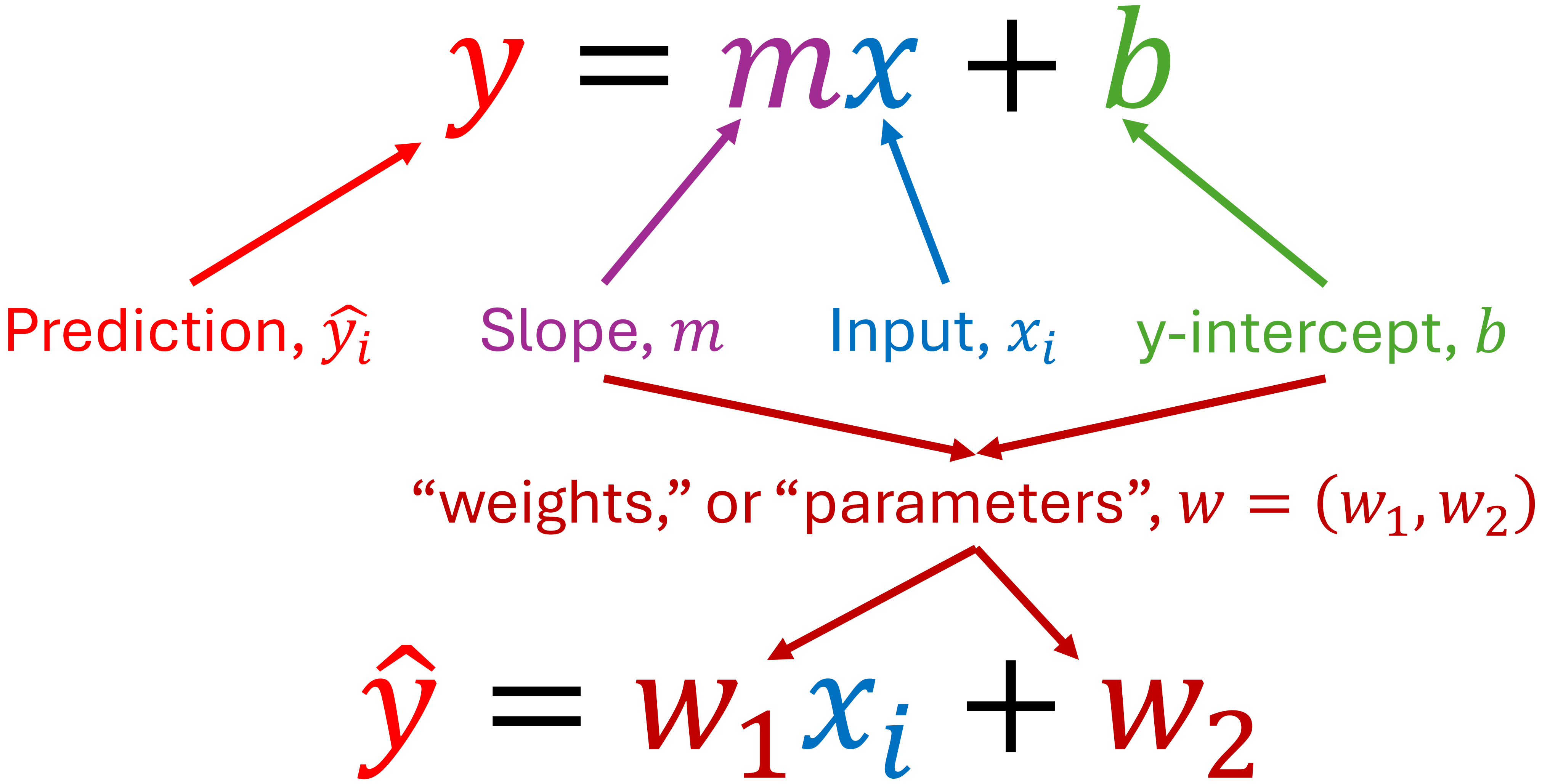
Linear Regression

- Search for the **line** that is a best fit to the data.
- Different performance measures correspond to different ways of measuring the quality of a fit.
- Sample mean squared error, or the sum of the squared errors is particularly common:

$$\widehat{\mathrm{MSE}}_n\text{: } \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \text{ and } \mathrm{SSE}\text{: } \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  - Although not identical, the line that minimizes one also minimizes the other.
- Using sample MSE, this method is called “least squares linear regression.”

Linear Regression: What is a line?



$$y = mx + b$$

Prediction, $\hat{y}_i$

Slope, m

Input, x_i

y-intercept, b

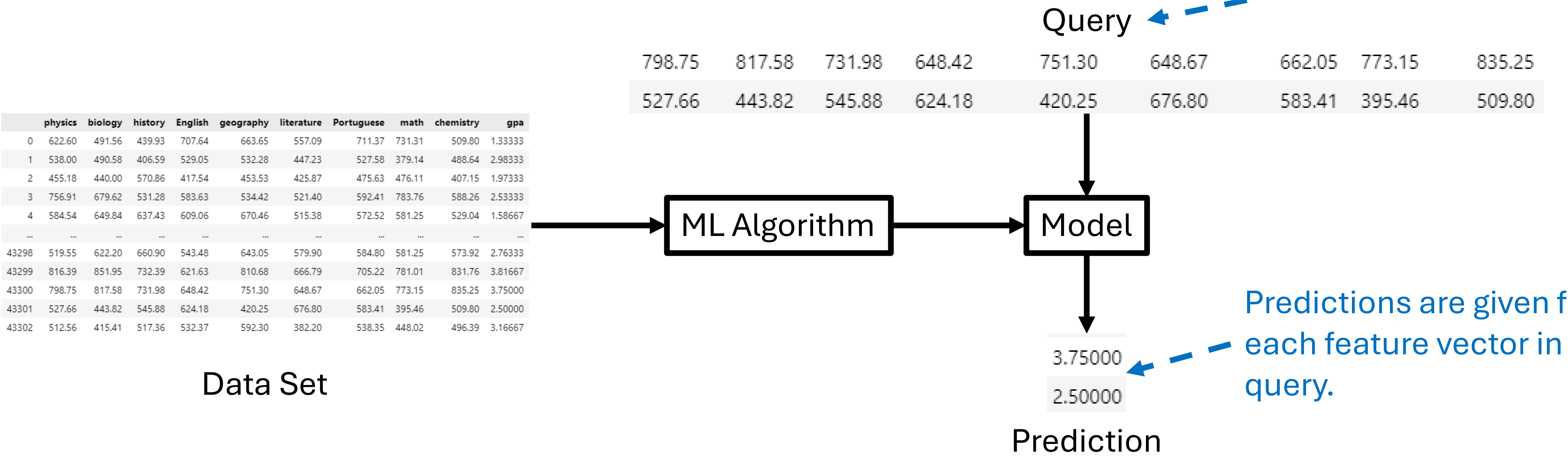
“weights,” or “parameters”, $w = (w_1, w_2)$

$$\hat{y} = w_1 x_i + w_2$$

Models (Review)

- A model is a mechanism that maps input data to predictions.
- ML algorithms take data sets as input and produce models as output.

A query can be one **or more** feature vectors.



Query

798.75	817.58	731.98	648.42	751.30	648.67	662.05	773.15	835.25
527.66	443.82	545.88	624.18	420.25	676.80	583.41	395.46	509.80

	physics	biology	history	English	geography	literature	Portuguese	math	chemistry	gpa
0	622.60	491.56	439.93	707.64	663.65	557.09	711.37	731.31	509.80	1.33333
1	538.00	490.58	406.59	529.05	532.28	447.23	527.58	379.14	488.64	2.98333
2	455.18	440.00	570.86	417.54	453.53	425.87	475.63	476.11	407.15	1.97333
3	756.91	679.62	531.28	583.63	534.42	521.40	592.41	783.76	588.26	2.53333
4	584.54	649.84	637.43	609.06	670.46	515.38	572.52	581.25	529.04	1.58667
...	...	...	...	...	...	...	...	...	...	...
43298	519.55	622.20	660.90	543.48	643.05	579.90	584.80	581.25	573.92	2.76333
43299	816.39	851.95	732.39	621.63	810.68	666.79	705.22	781.01	831.76	3.81667
43300	798.75	817.58	731.98	648.42	751.30	648.67	662.05	773.15	835.25	3.75000
43301	527.66	443.82	545.88	624.18	420.25	676.80	583.41	395.46	509.80	2.50000
43302	512.56	415.41	517.36	532.37	592.30	382.20	538.35	448.02	496.39	3.16667

Data Set

3.75000
2.50000

Prediction

Predictions are given for each feature vector in the query.

Parametric Model

- A model "parameterized" by a weight vector w.
- Different settings of w result in different predictions.
- Let $\hat{y} = f_w(x)$
    - 1-dimensional linear case:
    $$f_w(x) = w_1 x + w_2$$
    - d-dimensional linear case:
    $$f_w(x_i) = w_1 x_{i,1} + w_2 x_{i,2} + \dots + w_d x_{i,d}$$
    - We can write this as:
    $$f_w(x_i) = \sum_{j=1}^{d} w_j \, x_{i,j}.$$
    - This is called a **dot product** and can be written as $w \cdot x_i$ or $w^T x_i$.

Linear Regression: Optimization Perspective

- Given a parametric model f_w of any form how can we find the weights w that result in the "best fit"?
- Let L be a function called a **loss function**.
  - It takes as input a model (or model weights w)
  - It also takes as input data D
  - It produces as output a real-number describing how *bad* of a fit the model is to the provided data.
- The evaluation metrics we have discussed can be viewed as loss functions. For example, the sample MSE loss function is:

$$L(w, D) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f_w(x_i)\right)^2$$

For the sample MSE loss function, this can be *any* parametric model, not just a linear one!

- We phrase this as an **optimization problem**:

$$\operatorname{argmin}_w L(w, D)$$

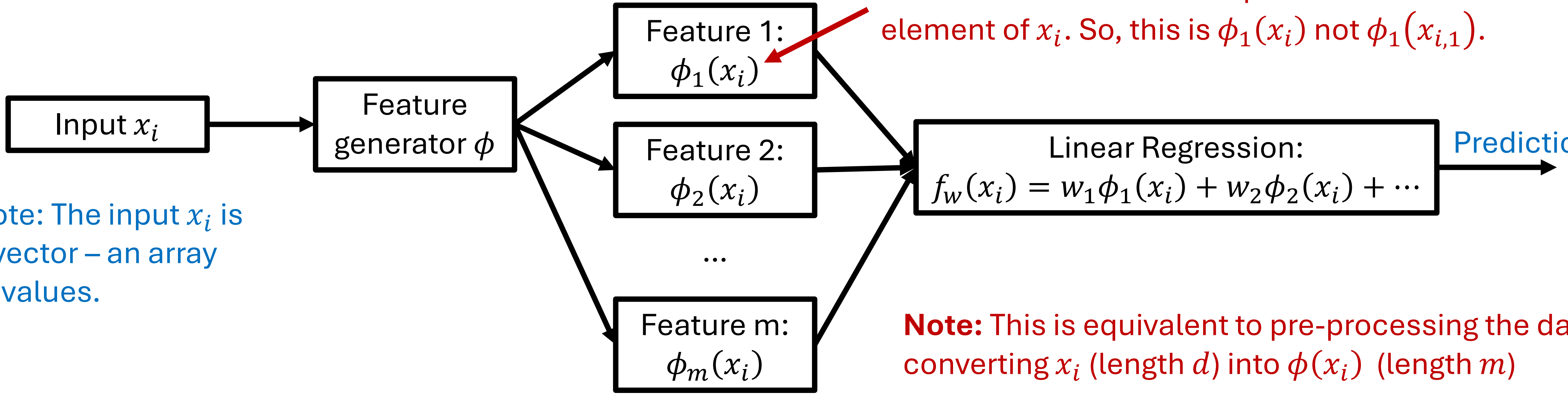
Linear Regression: Optimization Perspective

$$\operatorname{argmin}_w L(w, D)$$

- **Recall**: argmin returns the w that achieves the minimum value of $L(w, D)$, not the minimum value of $L(w, D)$ itself.
- This expression describes a *massive* range of ML methods.
  - Supervised, unsupervised, (batch/offline) RL
  - Deep neural networks
  - Large language models and generative AI
- Different problem settings and algorithms in ML correspond to:
  - Different loss functions
  - Different parametric models.
  - Different algorithms for approximating the best weight vector w.

Linear Parametric Model ≠Linear Functions

- **Linear parametric functions** are functions $f_w(x_i)$ that are **linear functions of the weights** $\boldsymbol{w}$.
- They need not be linear functions of the input x_i.



Each feature is a real number (not a vector or array)

Note: Each feature can depend on more than one element of x_i. So, this is $\phi_1(x_i)$ not $\phi_1(x_{i,1})$.

Note: The input x_i is a vector – an array of values.

Note: This is equivalent to pre-processing the data, converting x_i (length d) into $\phi(x_i)$ (length m)

Linear Parametric Model $\neq$Linear Functions

- **Linear parametric functions** are functions $f_w(x_i)$ that are **linear functions of the weights $\boldsymbol{w}$**.
- They need not be linear functions of the input x_i.
- That is, a linear parametric model has the form:

$$f_w(x_i) = \sum_{j=1}^{m} w_j \phi_j(x_i)\,,$$

where ϕ takes the input vector x_i as input and produces a vector of m features as output. That is, $\phi_j(x_i)$ is the j^{th} feature output by ϕ.

- ϕ is called the **basis function, feature generator,** or **feature mapping function**.

Multivariate Polynomial Basis

- How does the polynomial basis, ϕ, work if x is multidimensional (an array rather than a number?)
- Multivariate polynomial on inputs x, y:

$$a + bx + cy + dxy + ex^2 + fy^2 + gxy^2 + hx^2y + ix^3 + \cdots$$

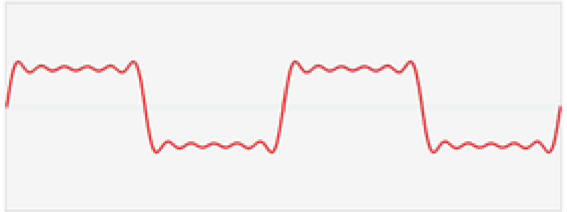
- Multivariate polynomial on input $x_{i,1}, x_{i,2}$:

$$w_1 + w_2 x_{i,1} + w_3 x_{i,2} + w_4 x_{i,1} x_{i,2} + w_5 x_{i,1}^2 + w_6 x_{i,2}^2 + w_7 x_{i,1} x_{i,2}^2 + w_8 x_{i,1}^2 x_{i,2}^2 + w_9 x_{i,1}^3 + \cdots$$

- The expression above is $f_w(x_i)$ for a linear parametric model using the multivariate polynomial basis.
- Notice that some $\phi_j(x_i)$ terms depend on more than one element of x_i!
  - This term is $w_8 \phi_8(x_i)$

Fourier Basis

- Each ϕ_j is a cosine function with a different period.
  - Can optionally include both sine and cosine functions.
- Univariate:
  - $\phi_j(x_i) = \cos(j\pi x)$
- Approximation of a step function (from Wikipedia "Fourier series" page)



Parametric vs Nonparametric

- ML algorithms are often categorized into **parametric** and **nonparametric**.
  - In general:
    - Parametric methods use parameterized functions with weights w.
    - Nonparametric methods store the training data or statistics of the training data.
  - More precisely
    - Parametric:
      - Have a fixed number of weights w.
      - Tend to make specific assumptions about the form of the function.
    - Nonparametric:
      - Do not make explicit assumptions about the form of the function.
      - Number of values stored tends to vary with the amount of training data (e.g., storing data).
  - There is some debate about whether some methods are parametric or nonparametric.
    - Linear regression and regression with linear parametric are canonical examples of parametric.
    - Nearest neighbor algorithms are canonical examples of nonparametric.

Optimization Perspective

- Recall:

$$\operatorname{argmin}_w L(w, D)$$

- Viewing $L(w, D)$ as a function, f, of just the weights (and a fixed data set):

$$\operatorname{argmin}_w f(w)$$

- Note that this is equivalent to maximizing a different function, where $g = -f$

$$\operatorname{argmax}_w g(w)$$

- We could also write x instead of w:

$$\operatorname{argmin}_x f(x)$$

- The function being optimized (minimized or maximized) is called the **objective function** (optimization terminology)**.**
  - In this case, our objective function is a **loss function** (machine learning terminology).
- **Question**: How do we find the input that minimizes a function?

Local Search Methods

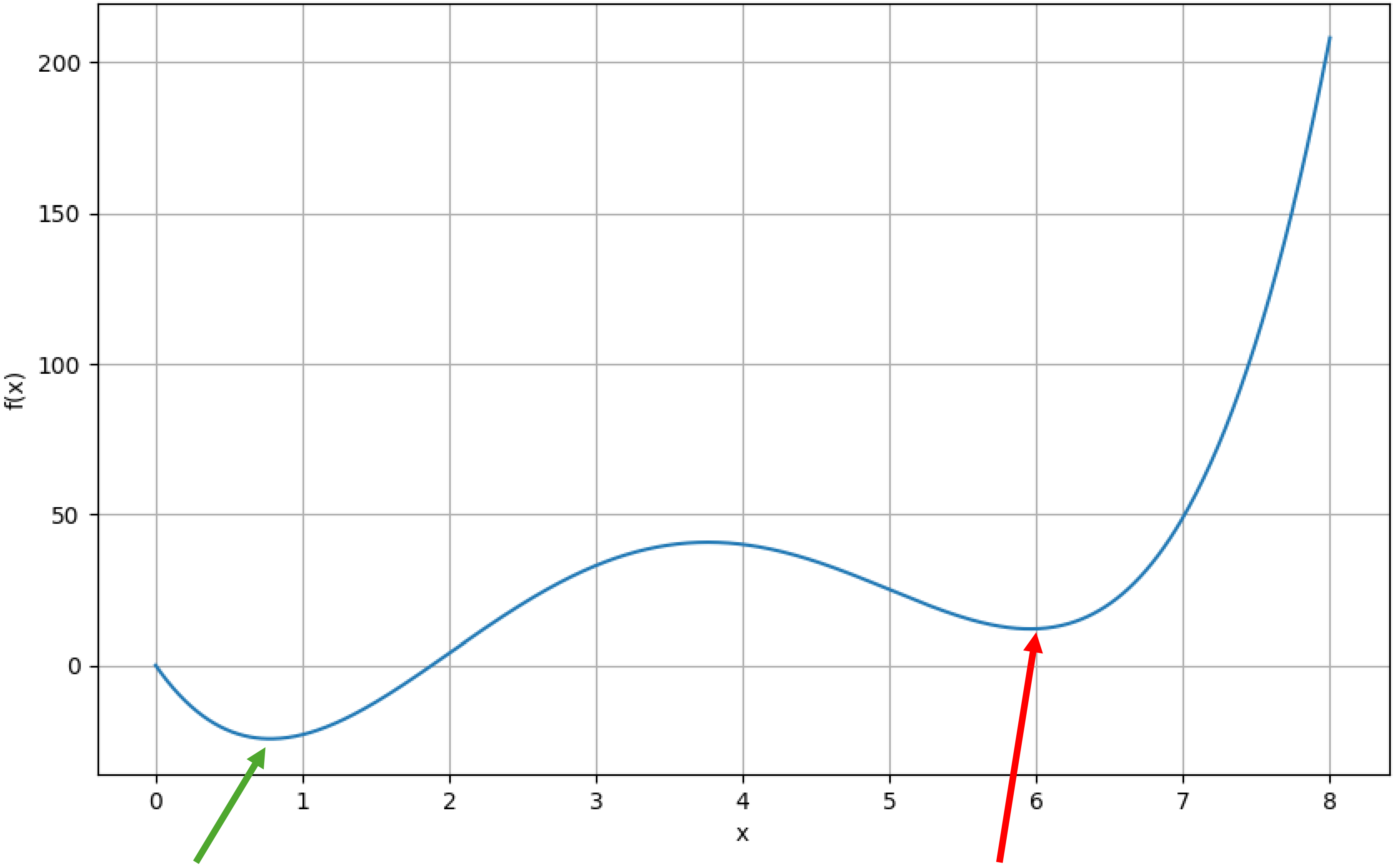
- Start with some initial input, x_0
- Search for a nearby input, x_1, that decreases f:
$$f(x_1) < f(x_0)$$
- Repeat, finding a nearby input x_{i+1} that decreases f (for each iteration i):
$$f(x_{i+1}) < f(x_i)$$
- Stop when:
    - You cannot find a new input that decreases f
    - The decrease in f becomes very small
    - The process runs for some predetermined amount of time
- Called “local search methods” because they search locally around some current point, x_i.

“Find a nearby point that decreases f”

- We will consider gradient-based optimizers.
- At any input/point x, we can query:
    - $f(x)$: The value of the objective function at the point
    - $\frac{df(x)}{dx}$: The derivative of the objective function at the point
        - This is the **gradient**, and is also written as $\nabla f(x)$

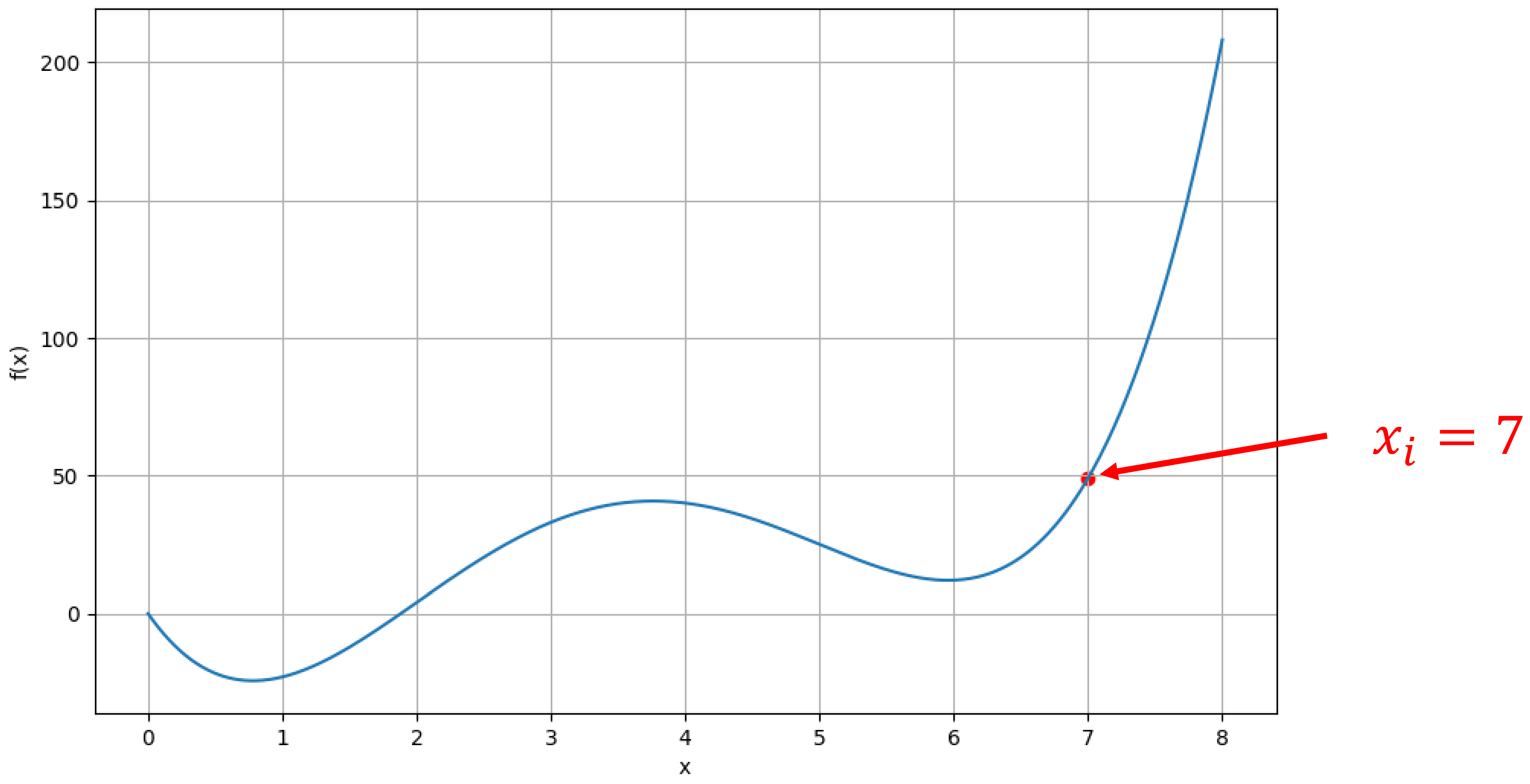
Question: Is a global minimum a local minimum?

Answer: Yes!

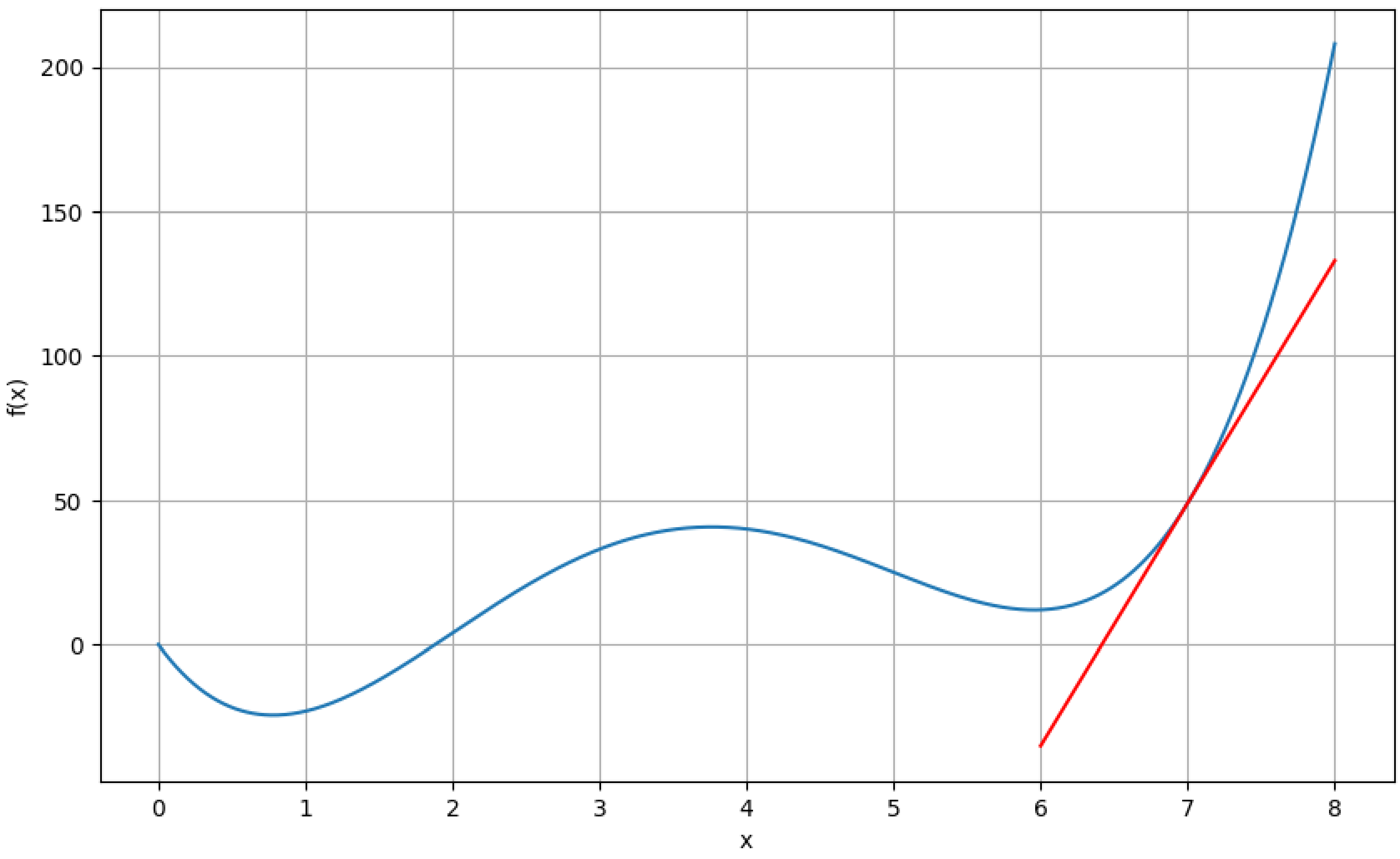


Global minimum: A location where the function achieves the lowest value (the argmin).

Local minimum: A location where all nearby (adjacent) points have higher values.

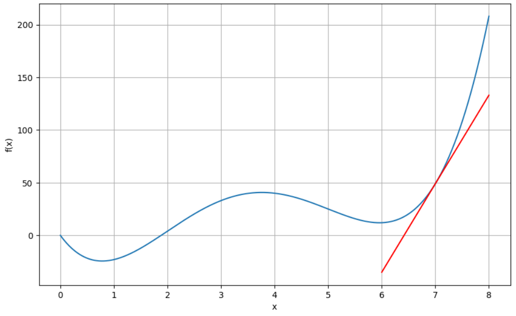


Question: How can we find a point x_{i+1} such that $f(x_{i+1}) < f(x_i)$? That is, a point that is “lower”?
Idea: Move a small amount “downhill”



Notice: The slope of the function tells us which direction is uphill / downhill.
Positive slope: Decrease x_i to get x_{i+1}. **Negative slope**: Increase x_i to get x_{i+1}.

Gradient Descent



- Take a step of length α (a small positive constant) in the opposite direction of the slope:

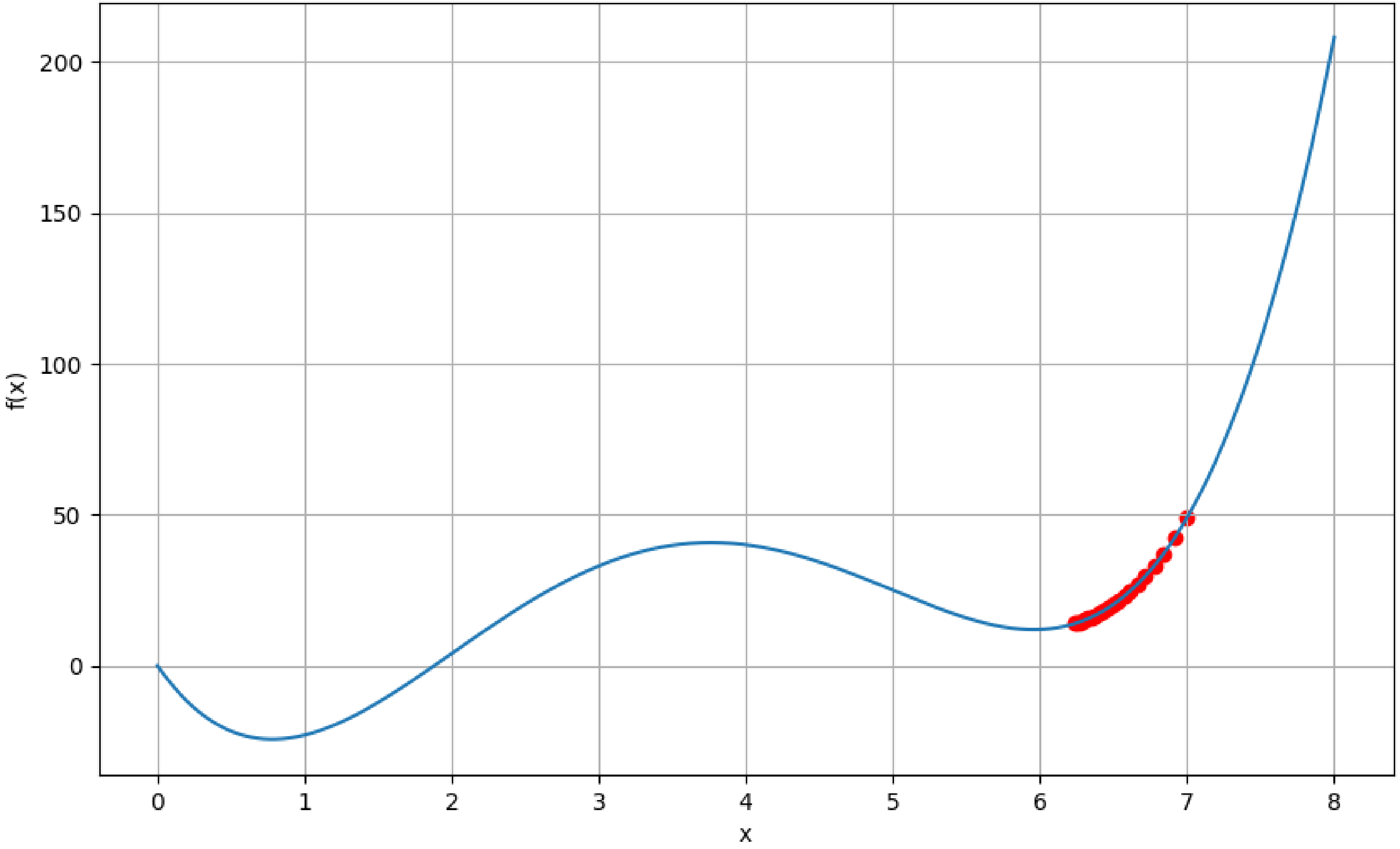
$$x_{i+1} = x_i - \alpha \times \text{slope}.$$

- **Note**: The slope is $\frac{df(x)}{dx}$, so we can write:

$$x_{i+1} = x_i - \alpha \frac{df(x)}{dx}.$$

- α is a hyperparameter called the **step size** or **learning rate**.

Gradient descent, $x_0 = 7, \alpha = 0.001$
$f(x) = x^4 - 14x^3 + 60x^2 - 70x$



Question: Why do the points get closer together when we use the same step size, α?

The Gradient (multi-dimensional setting)

Question: How can we find a new point that is “downhill”?

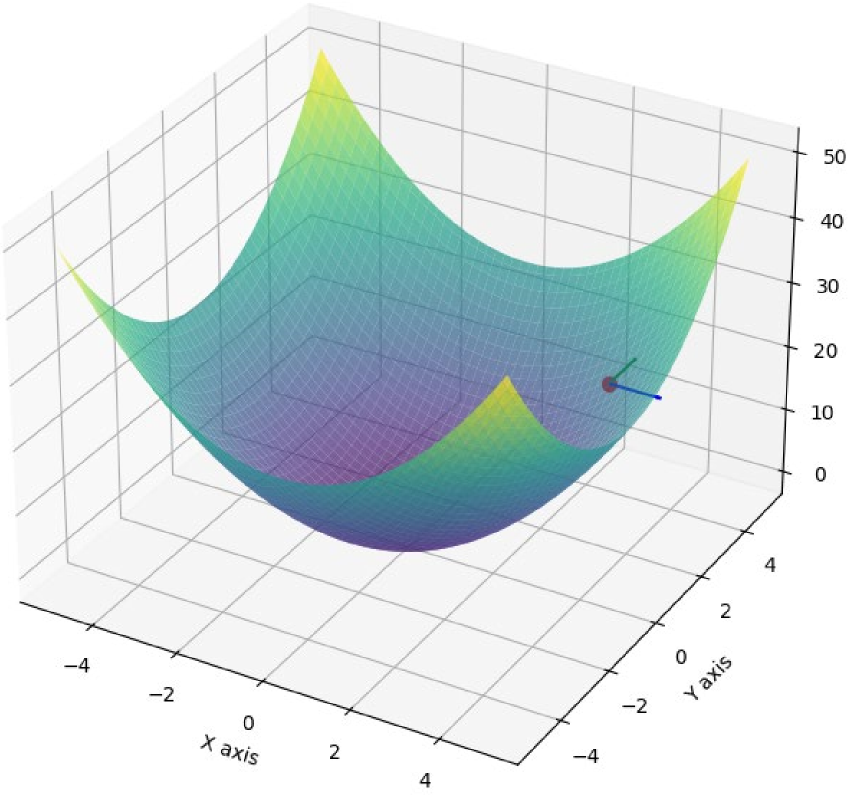
Idea: Compute the slope along each axis!

x-slope: $\frac{\partial f(x,y)}{\partial x}$

y-slope: $\frac{\partial f(x,y)}{\partial y}$

The **gradient** is the concatenation of the slopes along each dimension/axis:

$$\nabla f(x) = \left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right]$$



Note: The gradient is also called the “**direction of steepest ascent**”. It indicates how to change each input to go up-hill as quickly as possible.

Gradient Descent: Move both x and y in the negative direction of their slopes. That is, move in the opposite direction of the gradient:

$$x_{i+1} = x_i - \alpha \frac{\partial f(x_i, y_i)}{\partial x_i}$$

$$y_{i+1} = y_i - \alpha \frac{\partial f(x_i, y_i)}{\partial y_i}$$

OR

$$(x_{i+1}, y_{i+1}) = (x_i, y_i) - \alpha \nabla f(x_i, y_i)$$

Pseudocode: Gradient Descent on $f(x)$

- **Hyperparameter**: Step size α. Typically a small constant like $0.1, 0.01, 0.001, \ldots$
- **Assumption**: f is a function that takes a vector (or single real number) as input, and produces a single real number as output.
- **Assumption**: f is smooth (differentiable)
- **Method**:
    - Select an arbitrary initial point, x_0 (a vector).
    - For each iteration i, set $x_{i+1} = x_i - \alpha \nabla f(x_i)$. Equivalently, for each element of x_i (indexed by j):

$$x_{i+1,j} = x_{i,j} - \alpha \frac{\partial f(x_i)}{\partial x_{i,j}}$$

    - Stop when progress becomes slow or after some fixed amount of time.

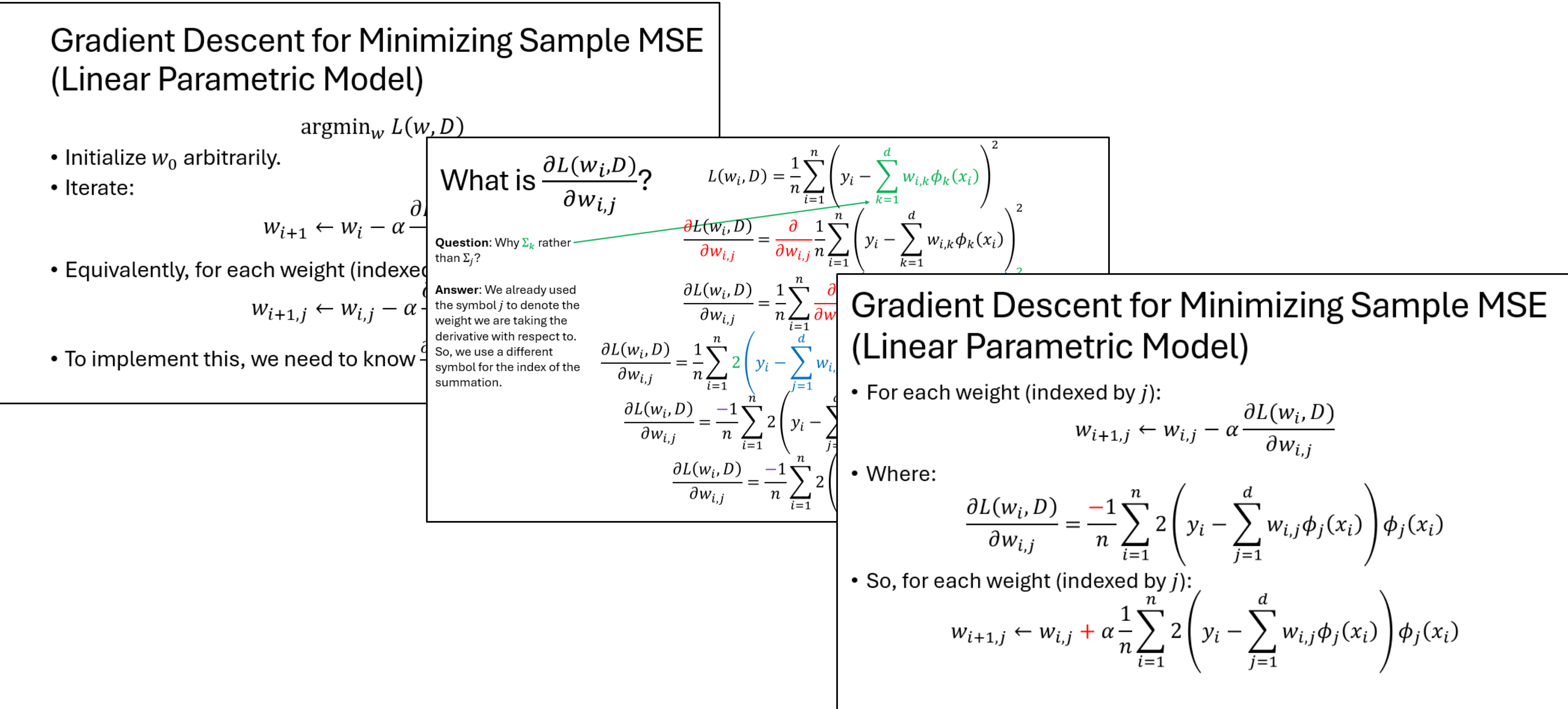
Pseudocode: Gradient Descent on $f(x)$

- **Hyperparameter**: Step size α. Typically a small constant like $0.1, 0.01, 0.001, \ldots$
- **Assumption**: f is a function that takes a vector (or single real number) as input, and produces a single real number as output.
- **Assumption**: f is smooth (differentiable)
- **Method**:
  - Select an arbitrary initial point, x_0 (a vector).
  - For each iteration i, set $x_{i+1} = x_i - \alpha \nabla f(x_i)$. Equivalently, for each element of x_i (indexed by j):

$$x_{i+1,j} = x_{i,j} - \alpha \frac{\partial f(x_i)}{\partial x_{i,j}}$$

  - Stop when progress becomes slow or after some fixed amount of time.

Manual derivation of gradient



Gradient Descent for Minimizing Sample MSE (Linear Parametric Model)

$$\text{argmin}_w\ L(w,D)$$

- Initialize w_0 arbitrarily.
- Iterate:

$$w_{i+1} \leftarrow w_i - \alpha \frac{\partial}{}$$

- Equivalently, for each weight (indexe

$$w_{i+1,j} \leftarrow w_{i,j} - \alpha$$

- To implement this, we need to know

What is $\frac{\partial L(w_i,D)}{\partial w_{i,j}}$?

$$L(w_i,D) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \sum_{k=1}^{d} w_{i,k}\phi_k(x_i)\right)^2$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \sum_{k=1}^{d} w_{i,k}\phi_k(x_i)\right)^2$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial w}$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=1}^{d} w_{i,}\right.$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{-1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=}\right.$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{-1}{n}\sum_{i=1}^{n} 2$$

Question: Why Σ_k rather than Σ_j?

Answer: We already used the symbol j to denote the weight we are taking the derivative with respect to. So, we use a different symbol for the index of the summation.

Gradient Descent for Minimizing Sample MSE (Linear Parametric Model)

- For each weight (indexed by j):

$$w_{i+1,j} \leftarrow w_{i,j} - \alpha\frac{\partial L(w_i,D)}{\partial w_{i,j}}$$

- Where:

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{-1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=1}^{d} w_{i,j}\phi_j(x_i)\right)\phi_j(x_i)$$

- So, for each weight (indexed by j):

$$w_{i+1,j} \leftarrow w_{i,j} + \alpha\frac{1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=1}^{d} w_{i,j}\phi_j(x_i)\right)\phi_j(x_i)$$

Missing Data

- **Question**: What can we do if some values are missing in the data set?
    - **Example**: Some students are missing exam scores.
- **Answer 1**: Remove rows with missing values.
    - This can add bias when there is a correlation between *when* points are missing and other features/labels.
    - This can be effective when only a few rows are missing values.
- **Answer 2**: Use **imputation techniques**.
    - Replace missing values with the mean or median feature value.
    - Replace missing values with the feature values from the nearest neighbor (or k nearest neighbors).
    - Use more sophisticated techniques to estimate the missing values.

One Hot Encoding

- **One hot encoding** is a common strategy to avoid assigning meaning to the encoding of categorical features.
- If the feature has m possible values, it is converted into m features.
    - One column is converted into m columns.
- The value of the j^{th} new feature is 1 if the original feature took its j^{th} value, and 0 otherwise.
- Example: Original feature: “red”, “green”, “blue”
    - Three new features, “is red”, “is green”, and “is blue”
    - If “red”, the three new features have values [1, 0, 0]
    - If “green”, the three new features have values [0, 1, 0]
    - If “blue”, the three new features have values [0, 0, 1]

Feature Scaling

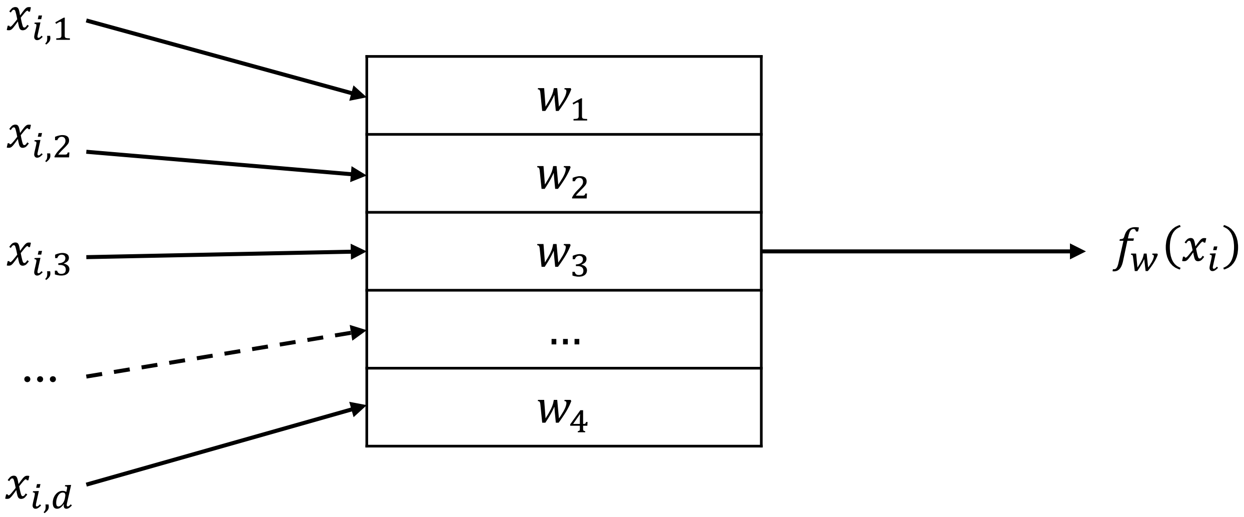
- When features have very different scales, it can cause problems for some ML algorithms.
  - **Question**: Consider a data set with income (range 0 to 1 million) and age (range 0 to 100). If we use nearest neighbor algorithms with Euclidean distance, what will happen?
  - **Answer**: Points with (relatively) slightly different incomes will be viewed as far apart relative to points with different ages.
  - **Note**: This is not unique to nearest neighbors algorithms. *Most* ML algorithms can struggle when features have very different scales.
- When all features have a very large or small scale, it can change the necessary hyperparameters in unintuitive ways.
  - **Example**: The step size for running gradient descent to fit a linear parametric model, using the second-degree polynomial basis, to the GPA data set (see `8.0 Data Cleaning Intro.ipynb`).

Feature Scaling

- **Idea**: Re-scale features.
- **Approach 1 (Min-Max Scaling)**: Normalize to the range [0,1]
    - $x_{\text{normalized}} = (x_{\text{unnormalized}} - \text{min})/(\text{max} - \text{min})$
    - Scikit-learn includes “Scalers” that perform common feature rescaling.
    - The `fit_transform` function “fits” the scaler to the data (e.g., calculating min and max values of features) and then “transforms” the data (applies the specified rescaling).
- **Approach 2 (Standardization)**:
    - Centers the feature (so the average is zero)
    - Rescales the feature so that the standard deviation is 1
    - $x_{\text{normalized}} = (x_{\text{unnormalized}} - \text{mean})/(\text{standard deviation})$
- Several others (robust scaling, normalization, etc.)

Question: How can we make this model non-linear w.r.t. the model parameters (weights w)?

$$f_w(x_i) = \sum_{j=1}^{d} w_j x_{i,j}$$

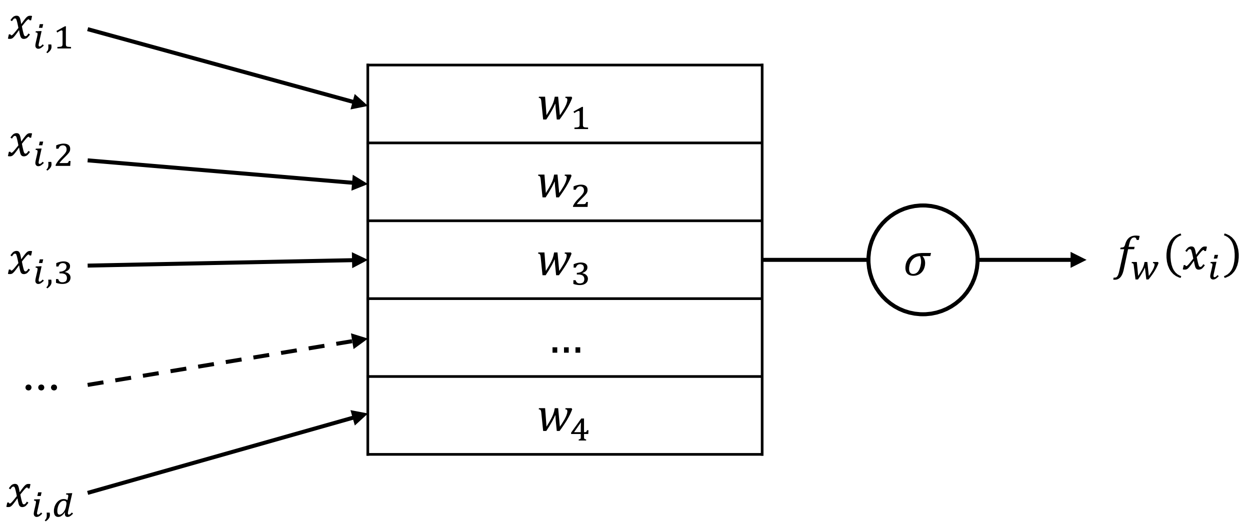


Answer: One way is to apply a non-linear function, σ, to the output.

Question: Would $\sigma(z) = 5z$ work?
Answer: No, this is a linear function. This would be equivalent to multiplying each weight by 5. It doesn't change the functions that can be represented.

Question: Would $\sigma(z) = z^2$ work?
Answer: Yes, this would result in a non-linear parametric model.

$$f_w(x_i) = \sigma\left(\sum_{j=1}^{d} w_j x_{i,j}\right) \quad f_w(x_i) = \left(\sum_{j=1}^{d} w_j x_{i,j}\right)^2$$



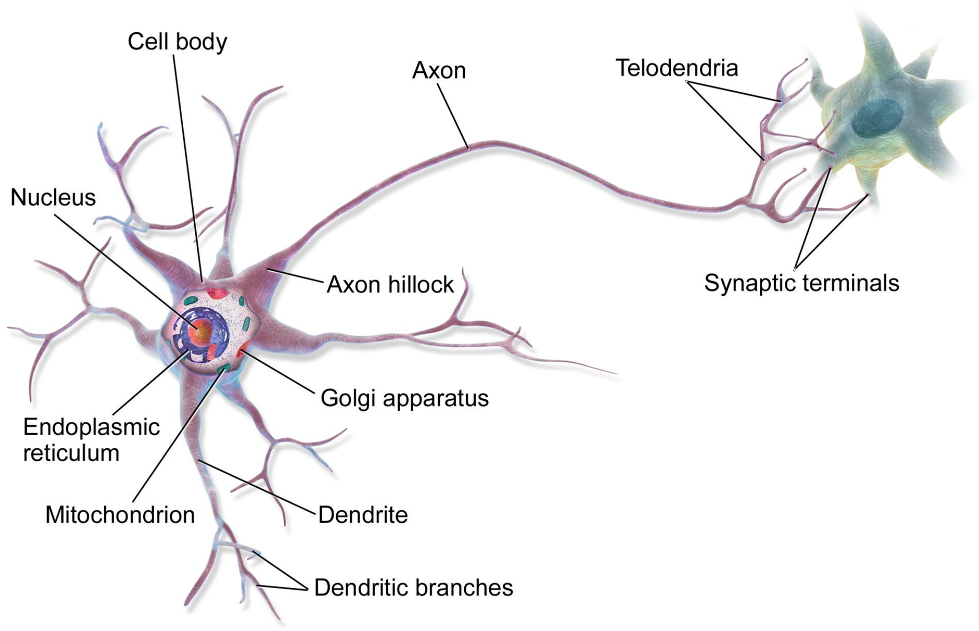
Note: The function σ is often called an **activation function, nonlinearity, threshold function**, or **squashing function**.

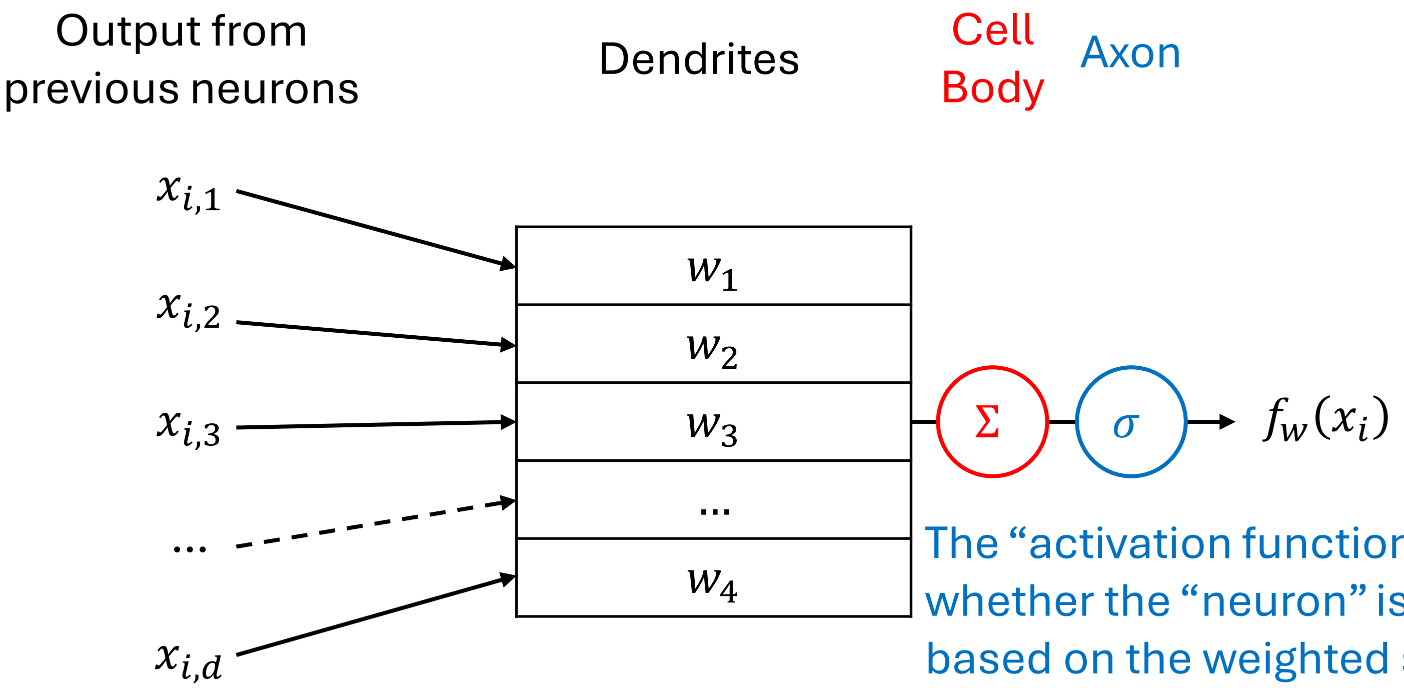
Note: This parametric model (with any nonlinear σ) is called a **perceptron**.

Perceptron

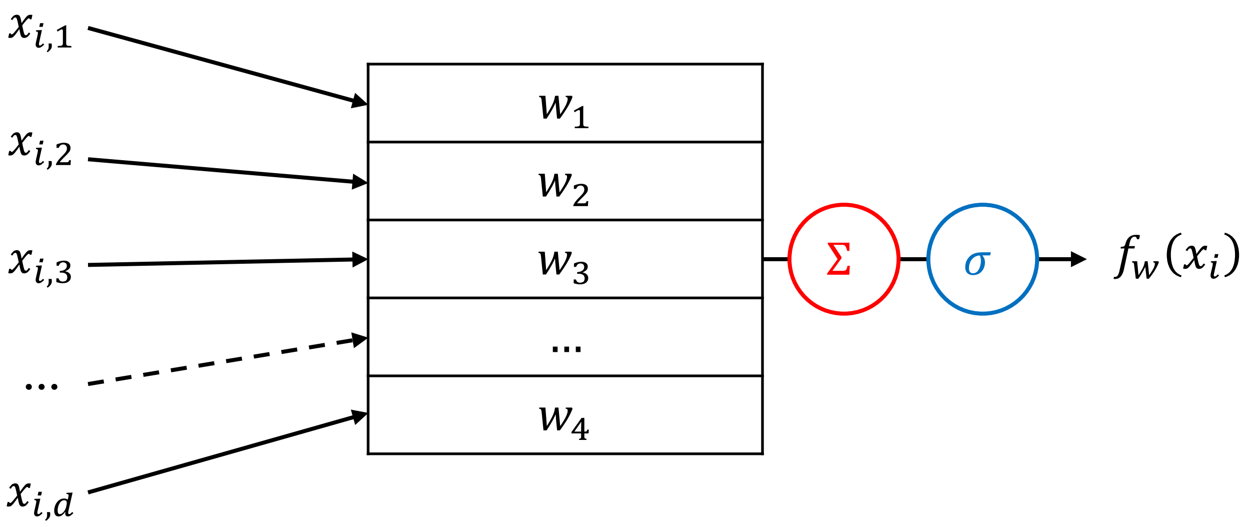
Perceptrons can be viewed as **extremely crude** simulations of neurons.

- Roughly speaking (ignoring important aspects of biology and neuroscience), when enough of the inputs to a neuron are activated, the neuron becomes sufficiently stimulated and "fires" (it becomes activated).
- We can select σ to be similar to a threshold function.
  - If the weighted sum is below some threshold for the neuron to be activated, σ outputs 0 (not firing).
  - If the weighted sum is above the threshold, σ outputs 1 (firing).



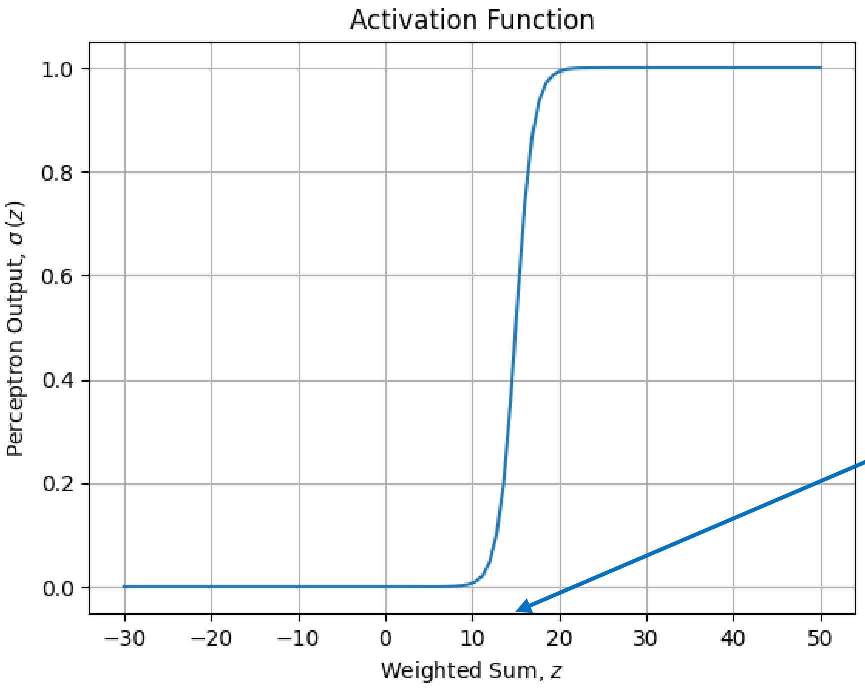


The "activation function" decides whether the "neuron" is firing based on the weighted sum.



Note: This model typically outputs 0 or 1, which may not be what we want for our parametric model. We will revisit this later.

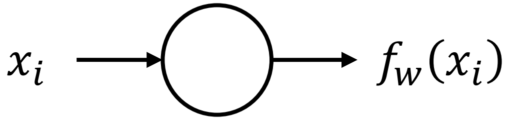
Note: σ *squashes* the output to the range [0,1], hence the name *squashing function*.



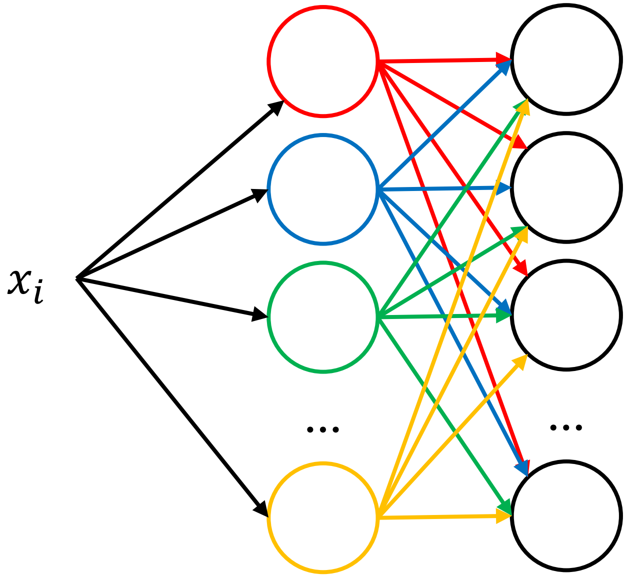
Threshold ≈ 15

Neural Networks: Parametric Models Comprised of Many Perceptrons

- Recall the graphical representation:

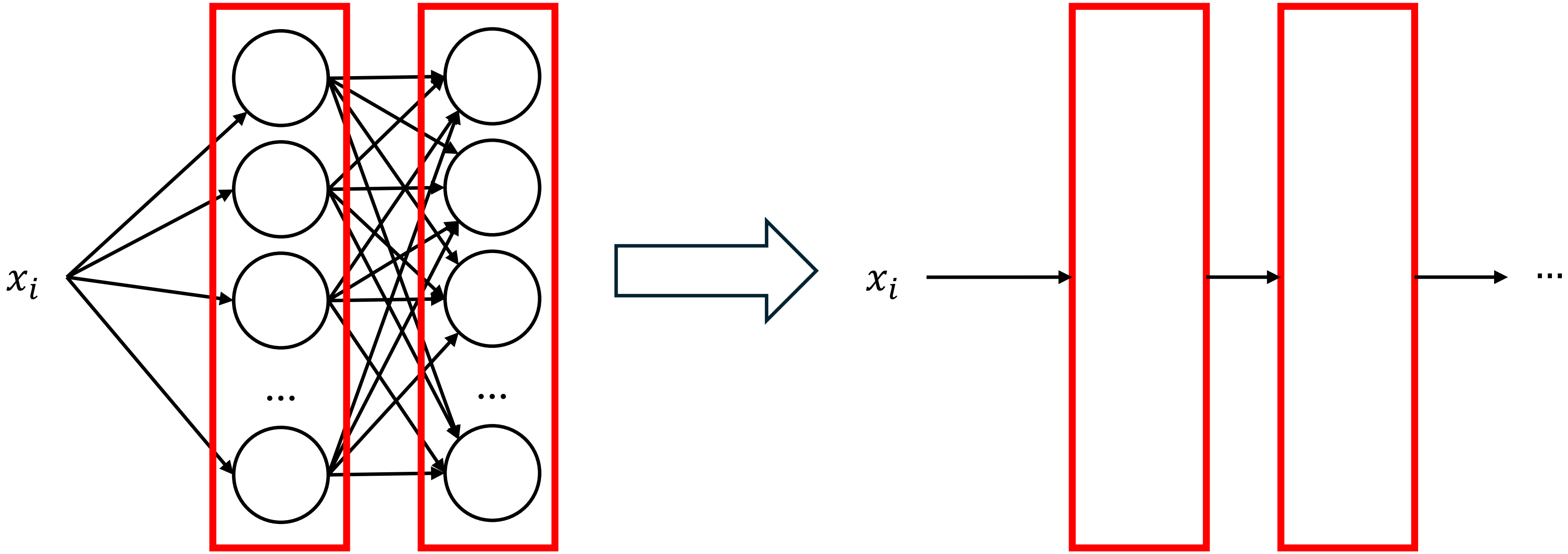


- **Idea**: Connect many perceptrons together.



This is tedious and too many arrows!

Neural Network Graphical Depiction

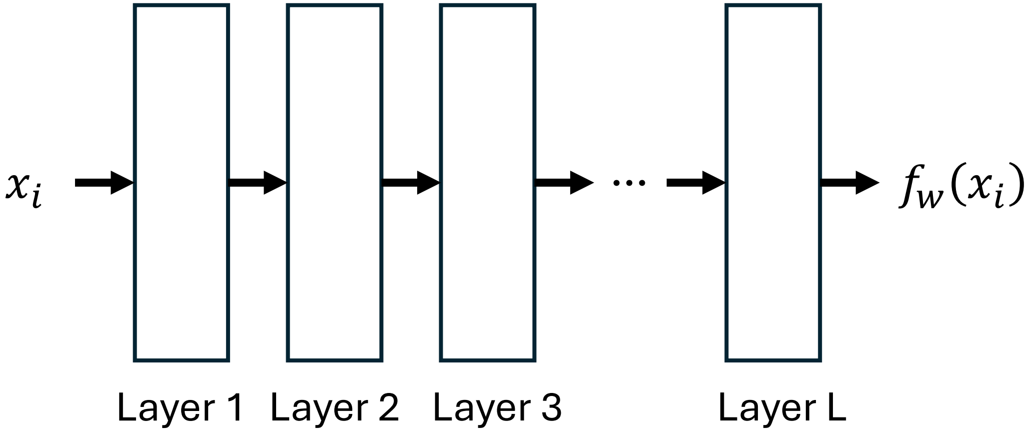


Idea: Use boxes to represent **layers** (columns) of perceptrons.

Here arrows between boxes denote **fully connected layers**.

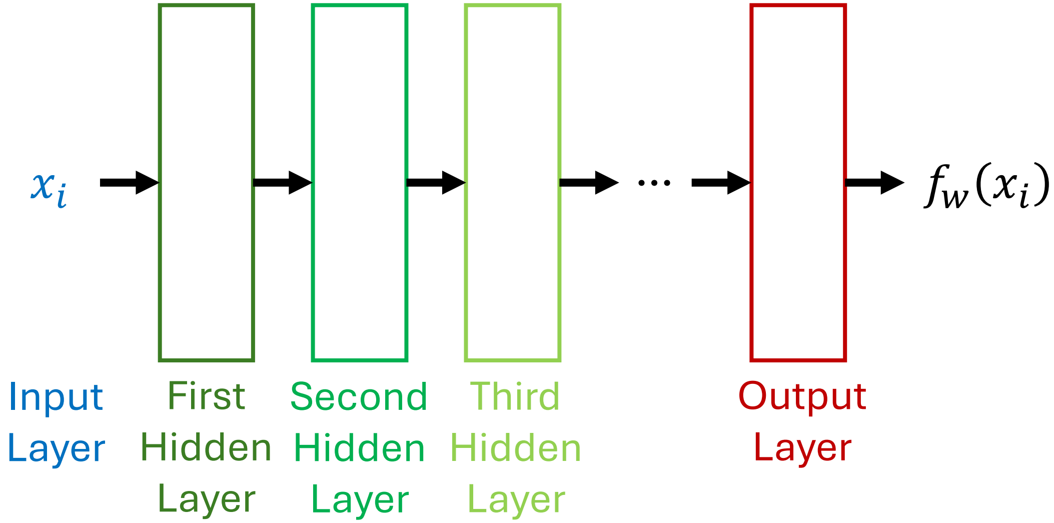
- Each perceptron in the right-layer takes the output of each perceptron in the left-layer as input.

Neural Network (Graphical Depiction)



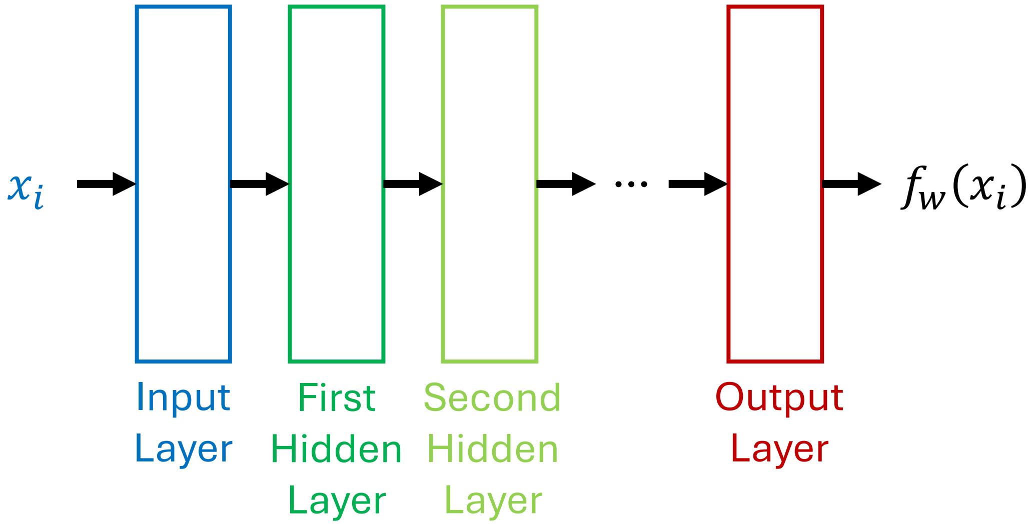
- In the context of neural networks, perceptrons are often called **units**.
- Each layer can have different numbers of units.
    - The number of units in a layer is often called the “size” of the layer.

Neural Network (Graphical Depiction)



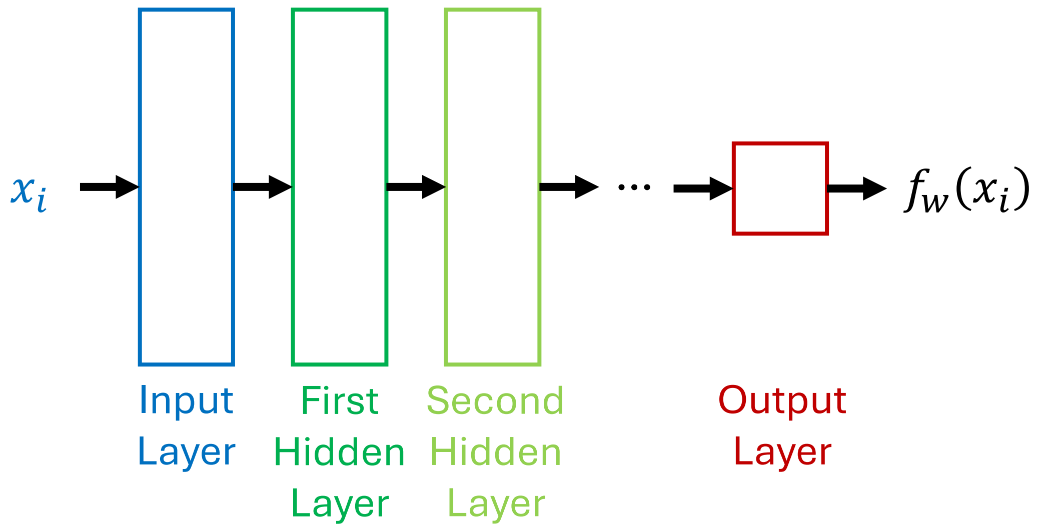
- The input, x_i is called the **input layer**.
- The last layer is called the **output layer**.
- All layers between the input and output layers are called **hidden layers**.

Neural Network (Graphical Depiction)



- Sometimes the input layer is represented by its own rectangle.
- This layer simply outputs x_i.

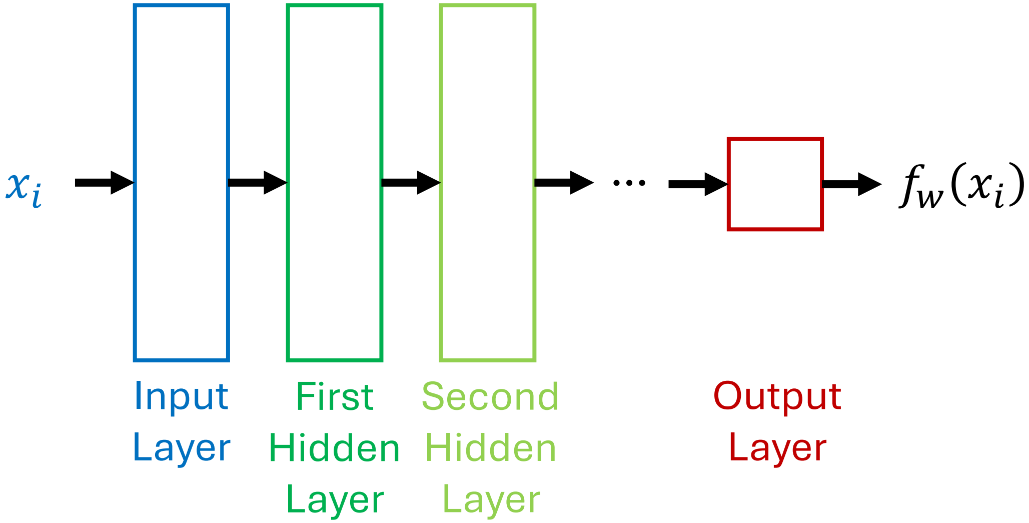
Neural Network (Graphical Depiction)



For a classification problem with 10 classes, how many outputs should the network have?

- The number of units in the output layer should equal the number of outputs of $f_w(x_i)$
  - For the GPA-prediction task, $x_i \in \mathbb{R}^9$ and $y_i \in \mathbb{R}$.
  - So, the output layer should have one unit.

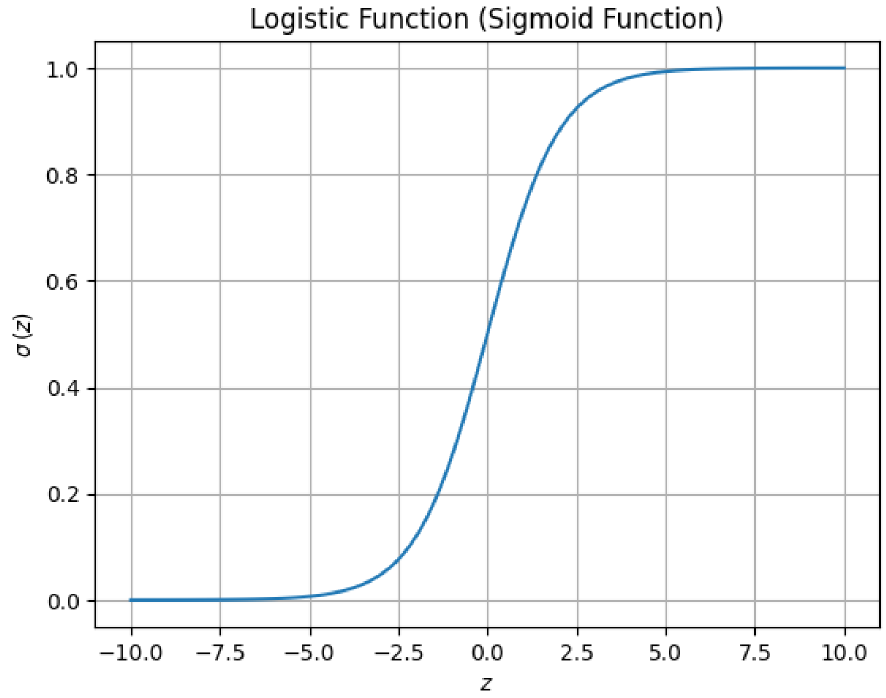
Neural Network (Graphical Depiction)



- If the output of the parametric model should not be “squashed” to [0,1], the squashing function (activation function) can be omitted from the output layer.

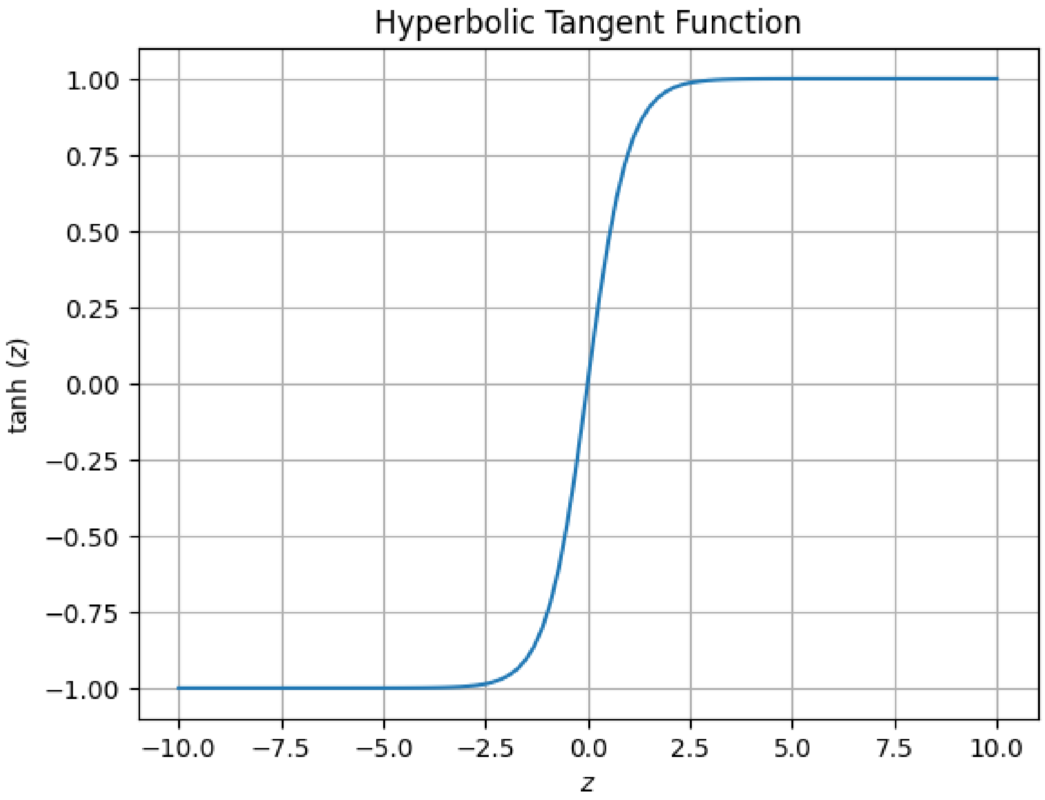
Activation Function: Sigmoid

- **Sigmoid functions** are a class of S-shaped functions.
- The most common one is called the **logistic function.**
  - It is so common that it is often called “the” sigmoid function.
- $\sigma(z) = \frac{1}{1+e^{-z}}$



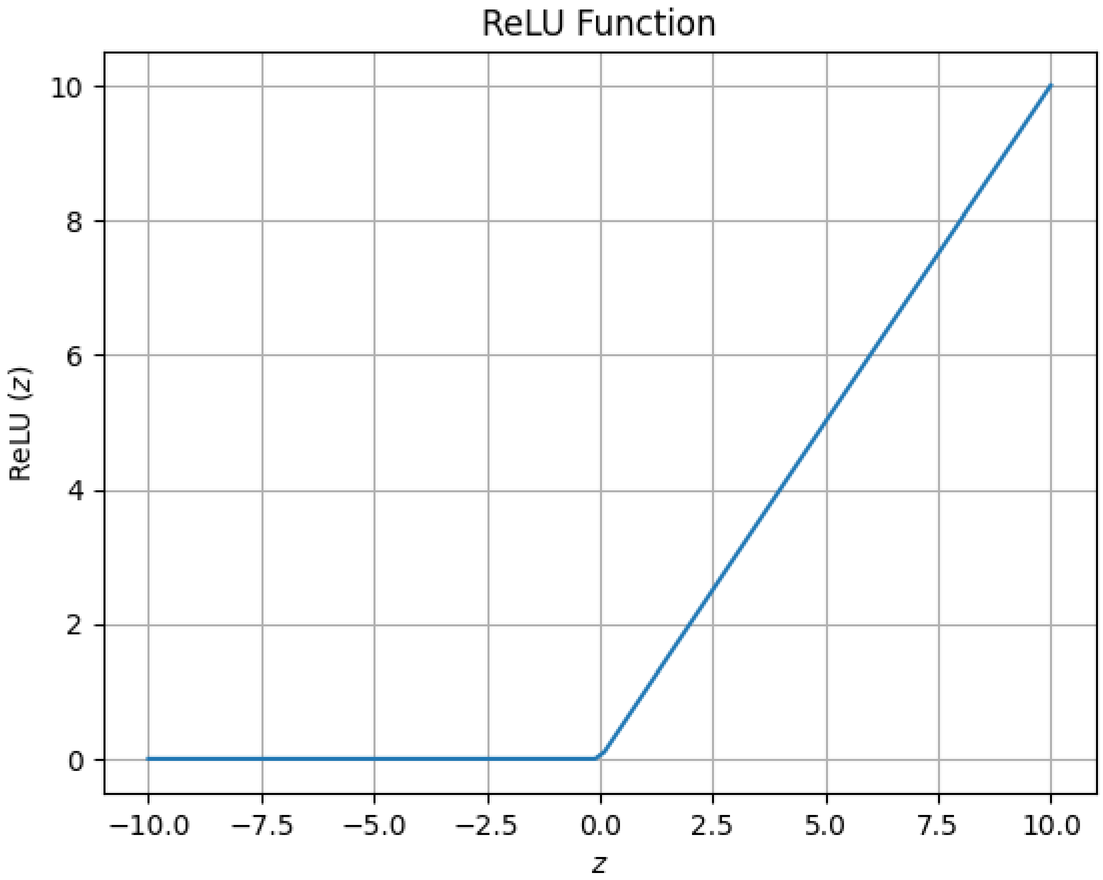
Activation Function: Hyperbolic Tangent Function (tanh)

- $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$



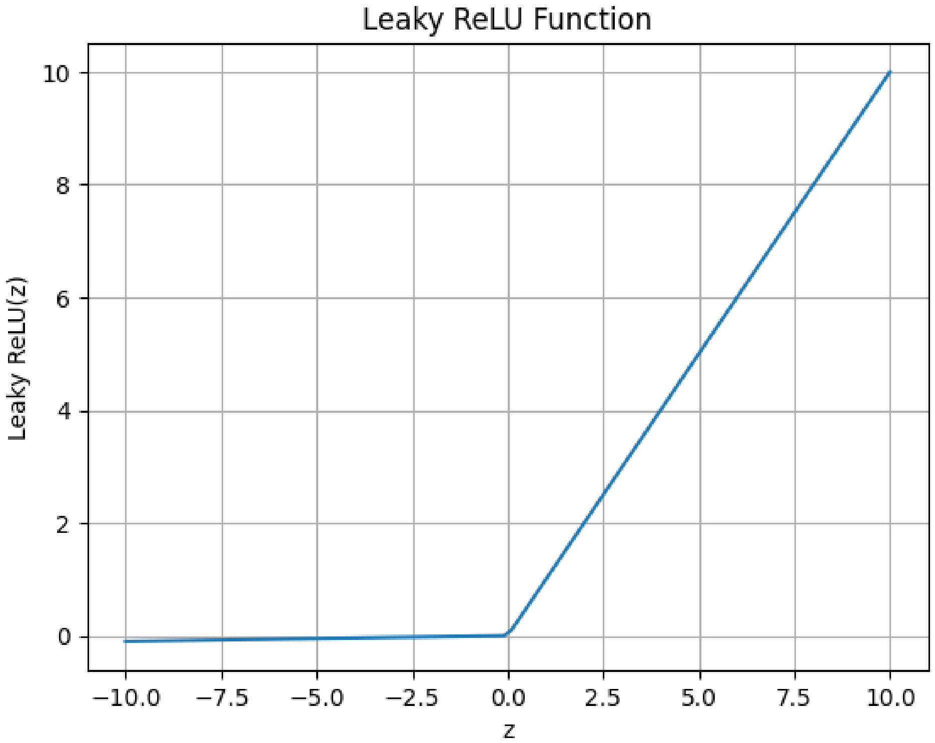
Activation Function: Rectified Linear Unit (ReLU)

- $\mathrm{ReLU}(z) = \max(0, z)$



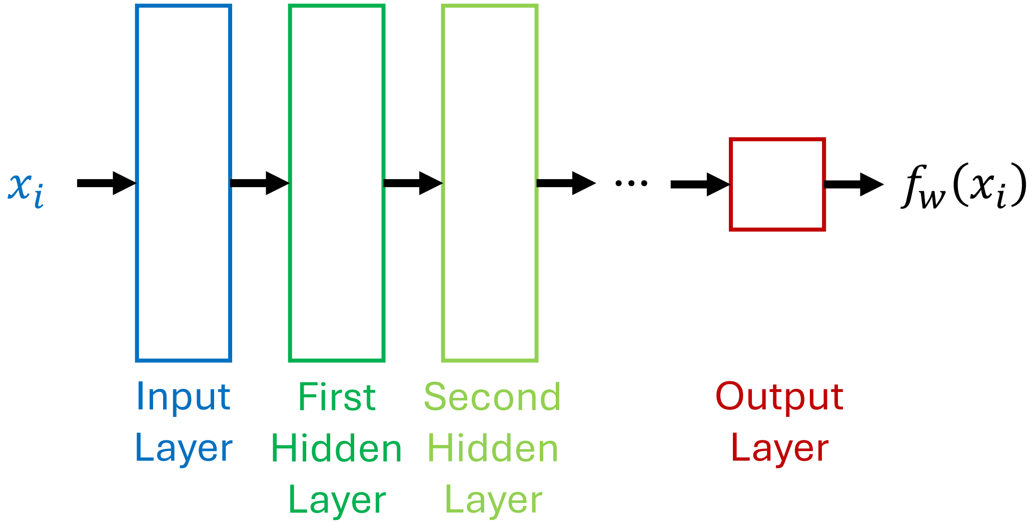
Activation Function: Leaky ReLU

- $\text{Leaky ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}$
  - Here α is a small constant, typically 0.01.



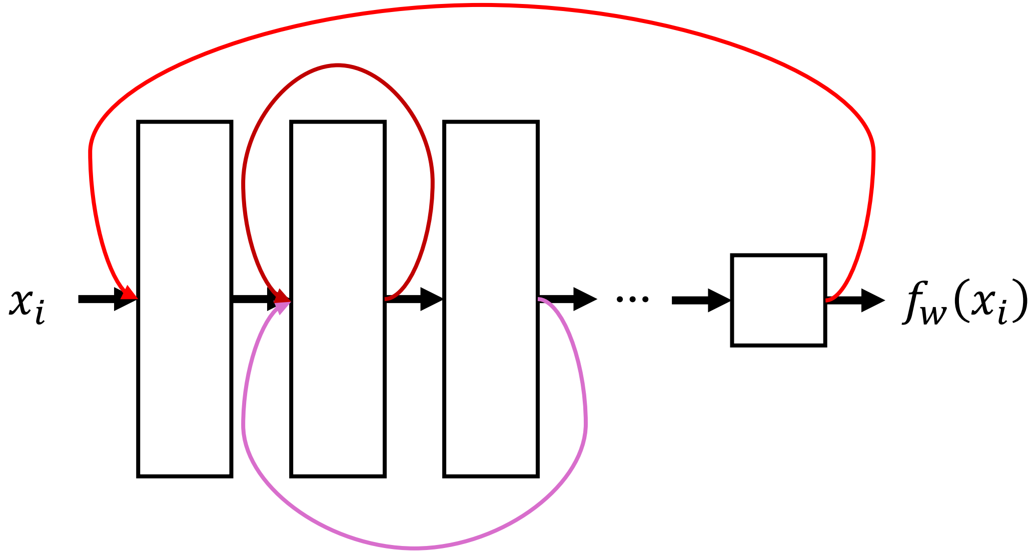
Fully-Connected Feed-Forward Networks

- A fully-connected feed-forward ANN is one where each unit in the i^{th} layer:
  - Takes the output of each unit in the $(i-1)^{\text{th}}$ layer as input.
  - Provides its output to each unit in the $(i+1)^{\text{th}}$ layer.



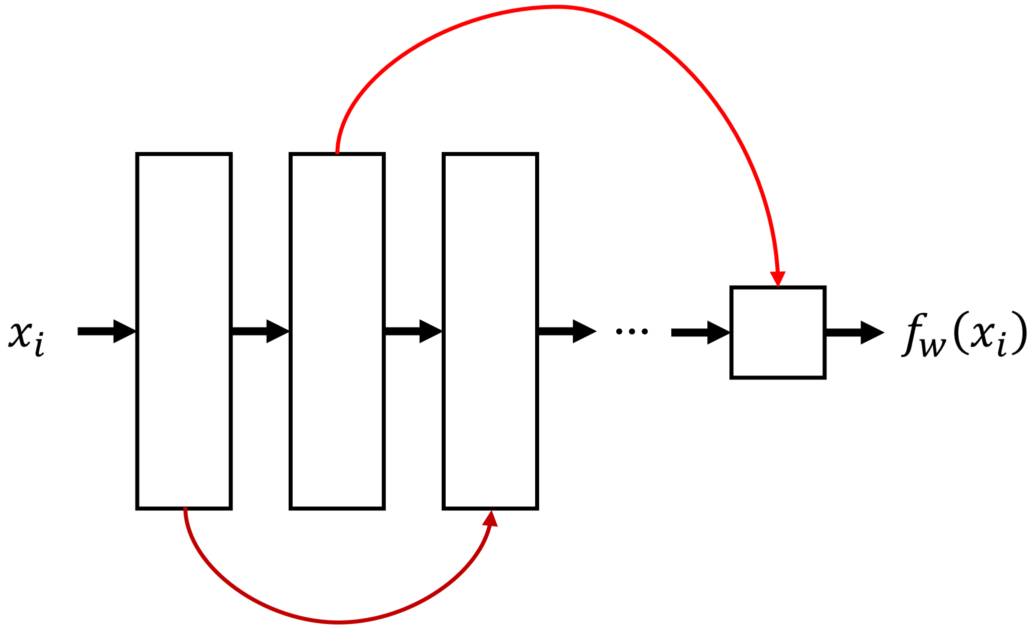
Recurrent Neural Network (RNN)

- Recurrent neural networks can have backwards connections between layers.
- These networks are typically run several times on the same input, and recurrent (backwards) edges provide values from the previous runs.
  - Recurrent connections provide a form of "memory"



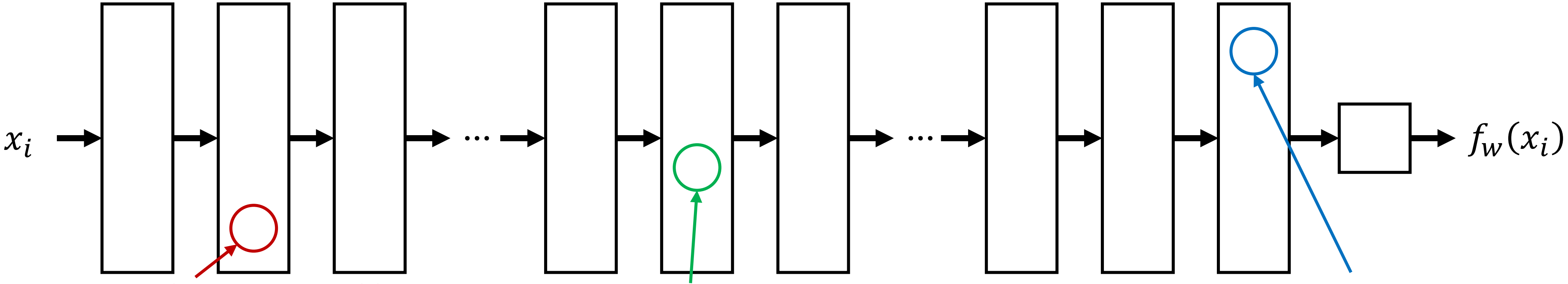
Skip Connections

- Skip connections are connections that skip over one or more layers.



What do different layers learn?

- Consider parametric models that take images as input.
- The layers closer to the input tend to learn low-level visual features.
- Later layers use these low-level features to learn about higher-level features and concepts.



Fires if there is an edge passing through position (372, 981) in the image, at an angle of 43 degrees.

Fires if there is a cow in the image

Fires if there is a cow jumping over the moon

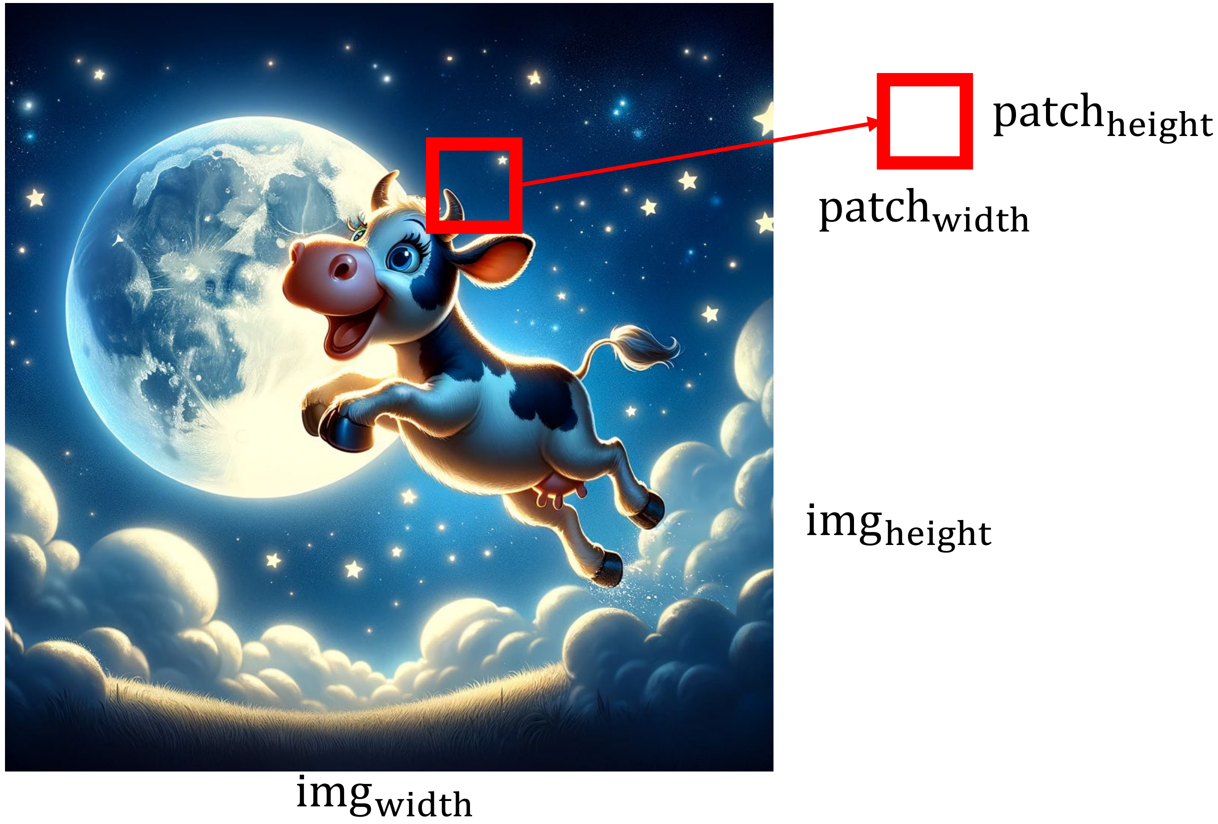
Learning Low-Level Features

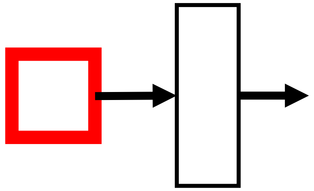
- An ANN might use early layers to detect low-level features of an image
  - One unit early in the network might “fire” when there is an edge at position (x,y) in the image, and the edge is vertical.
  - Another unit might fire when there is an edge at position (x,y) at an angle of 80 degrees (nearly vertical).
  - There may be different units for all of these features at each (x,y) coordinate in the image!
- Learning to separately detect the same feature at each location in the image is wasteful.
- **Idea**: Create a parametric model (layer for ANNs) that learns to find and represent features *anywhere* in the image.

Convolutional Layer

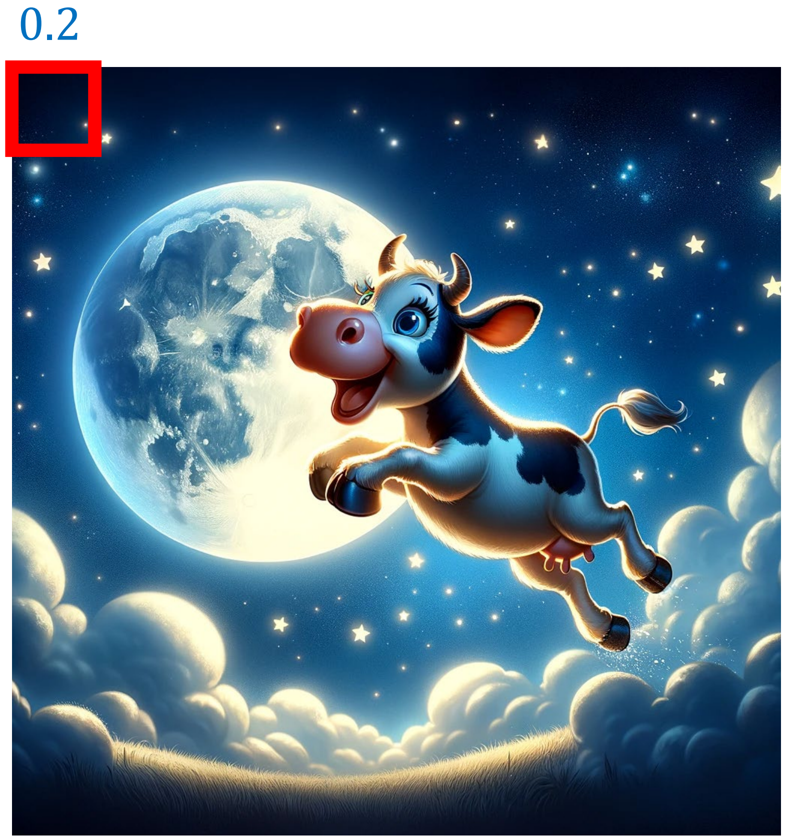
- If an image is of size $\text{img}_{\text{width}} \times \text{img}_{\text{height}}$, create a parametric model, called a **filter,** that takes as input a small subregion of the image, called a **patch**.

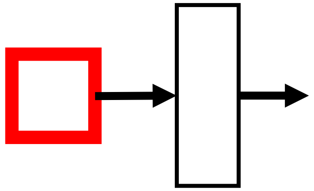
- This filter (small parametric model) is run on each patch in the image.
  - The patches can overlap.
  - Each patch is a fixed number of pixels over from the previous patch. This number is called the **stride**.


$\text{patch}_{\text{height}}$
$\text{patch}_{\text{width}}$
$\text{img}_{\text{height}}$
$\text{img}_{\text{width}}$

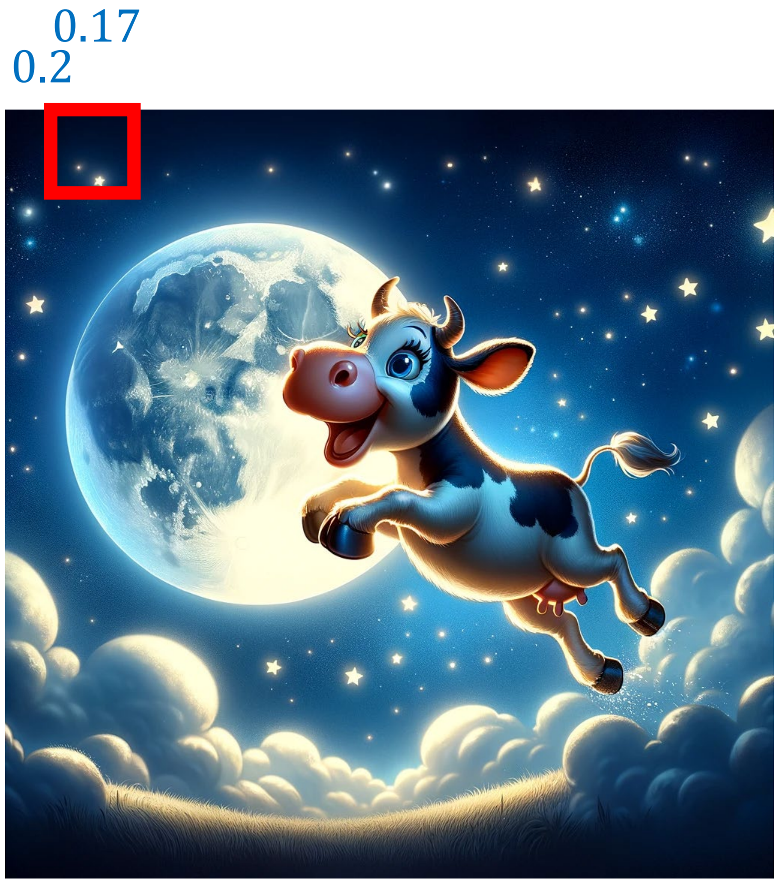


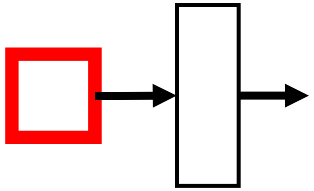
One number, the “feature” value for this patch.



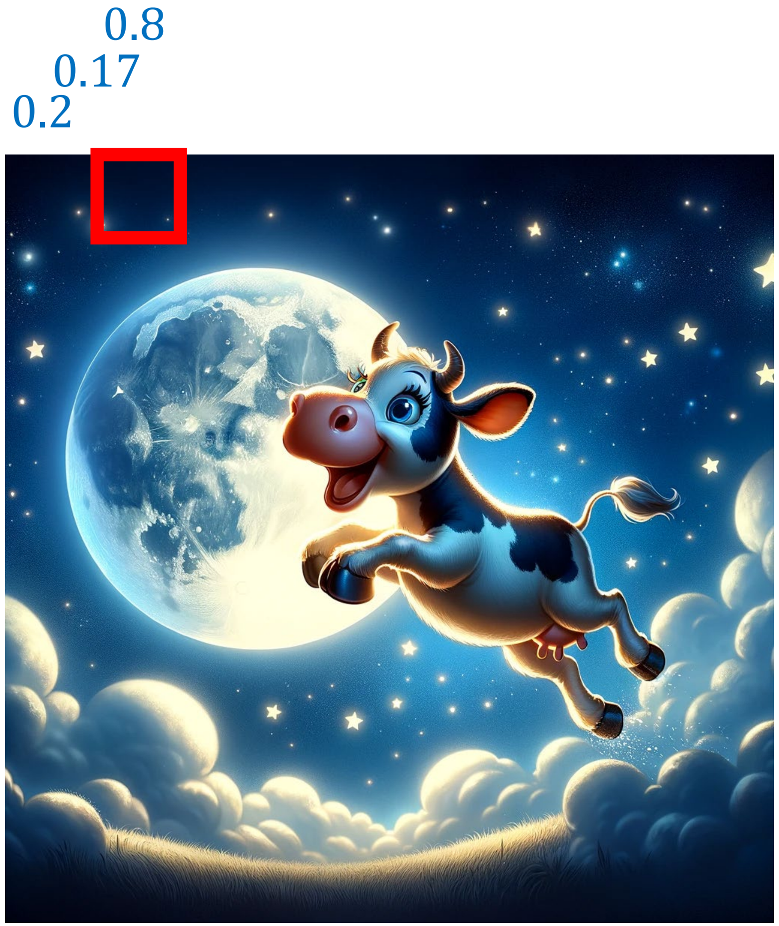


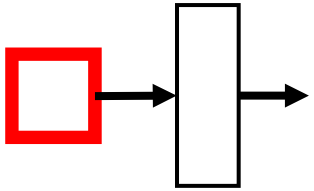
One number, the “feature” value for this patch.



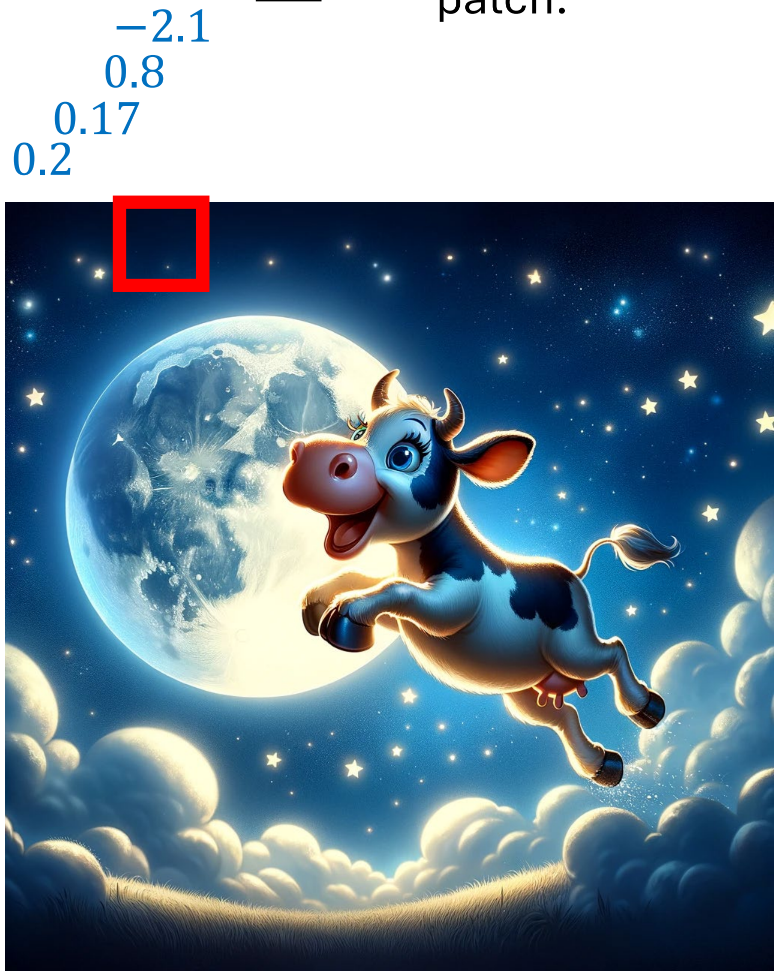


One number, the “feature” value for this patch.

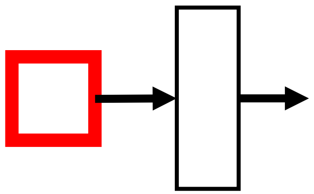




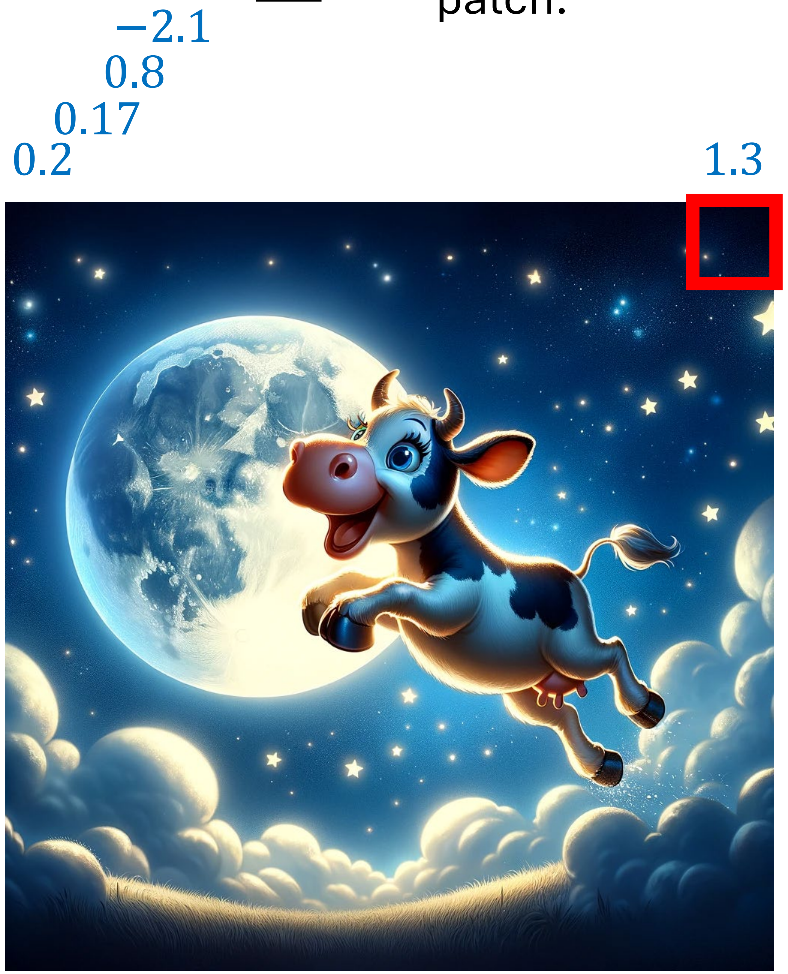
One number, the “feature” value for this patch.



The patch is shifted over by **stride** number of pixels each time.

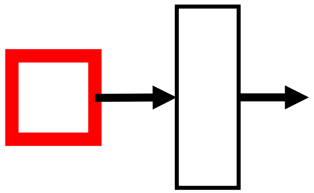


One number, the “feature” value for this patch.

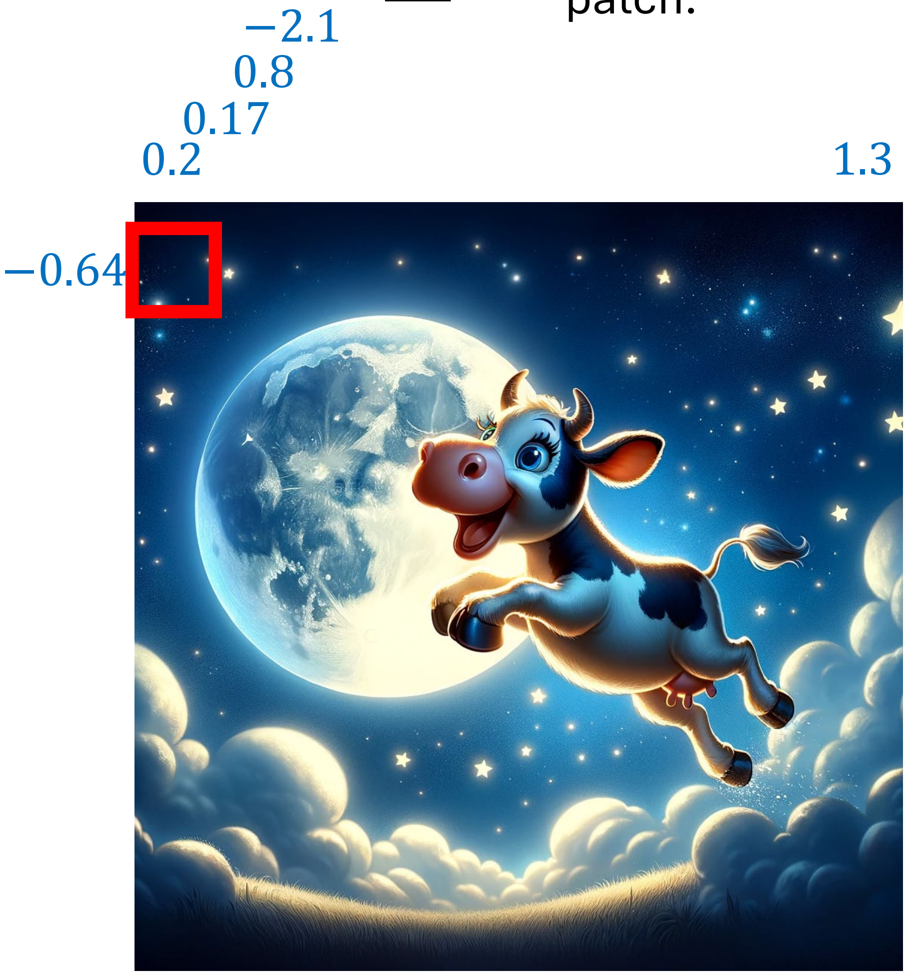


The patch is shifted over by **stride** number of pixels each time.

When the patch reaches the end, it shifts down by **stride** pixels and starts over.

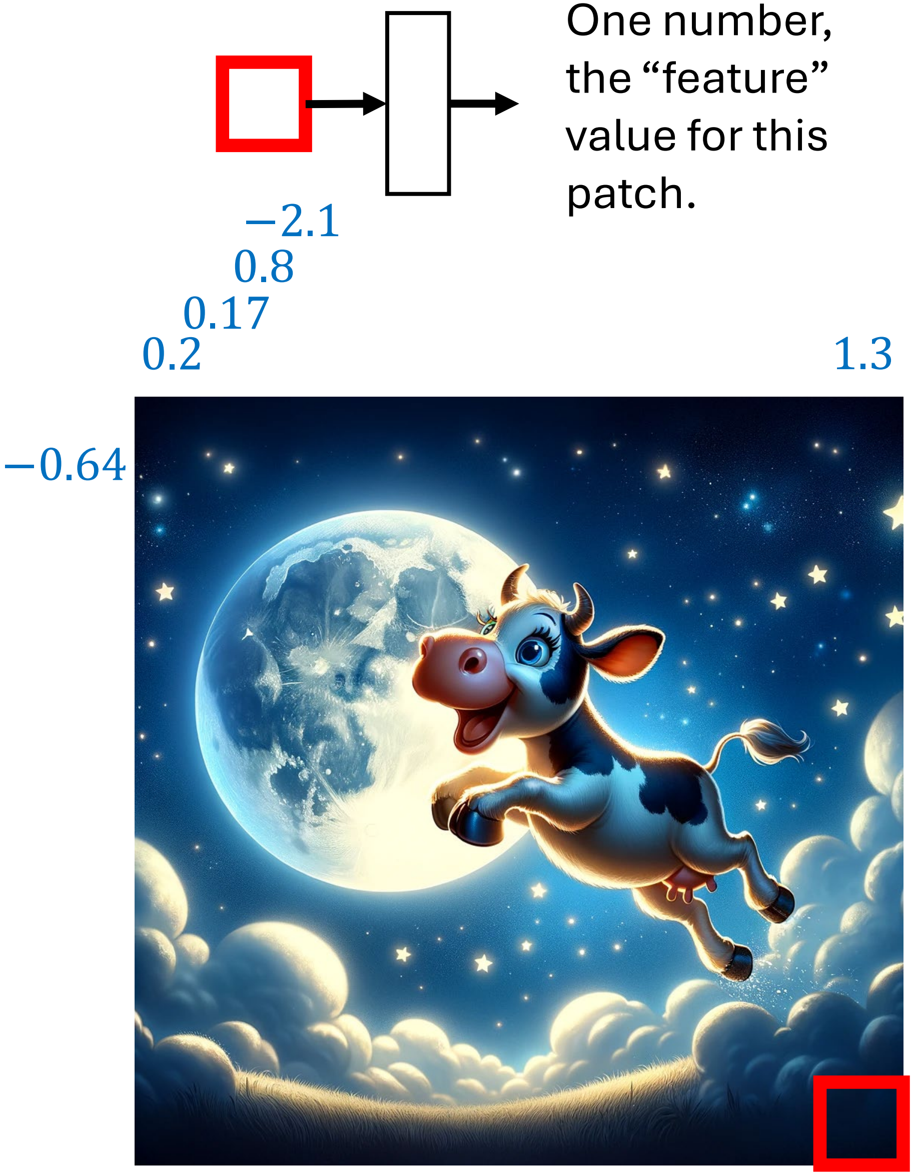


One number, the “feature” value for this patch.



The patch is shifted over by **stride** number of pixels each time.

When the patch reaches the end, it shifts down by **stride** pixels and starts over.

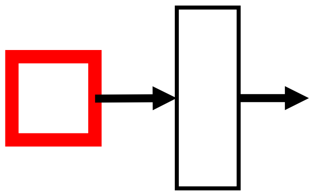


One number, the “feature” value for this patch.

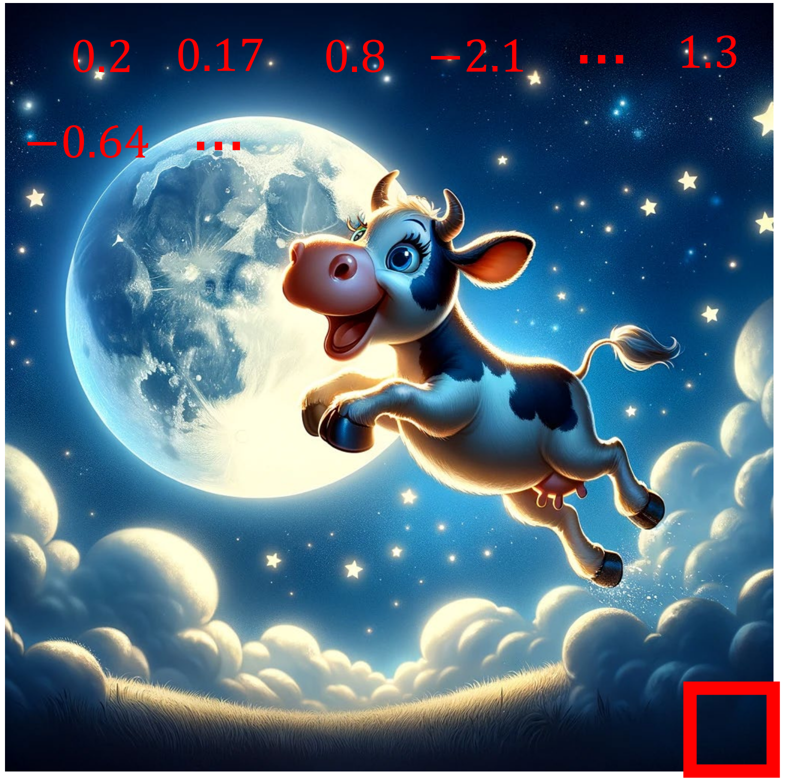
The patch is shifted over by **stride** number of pixels each time.

When the patch reaches the end, it shifts down by **stride** pixels and starts over.

At the end, the **convolutional layer** outputs all the computed values: $(0.2, 0.17, 0.8, -2.1, \ldots, 1.3, -0.64, \ldots)$



One number, the “feature” value for this patch.



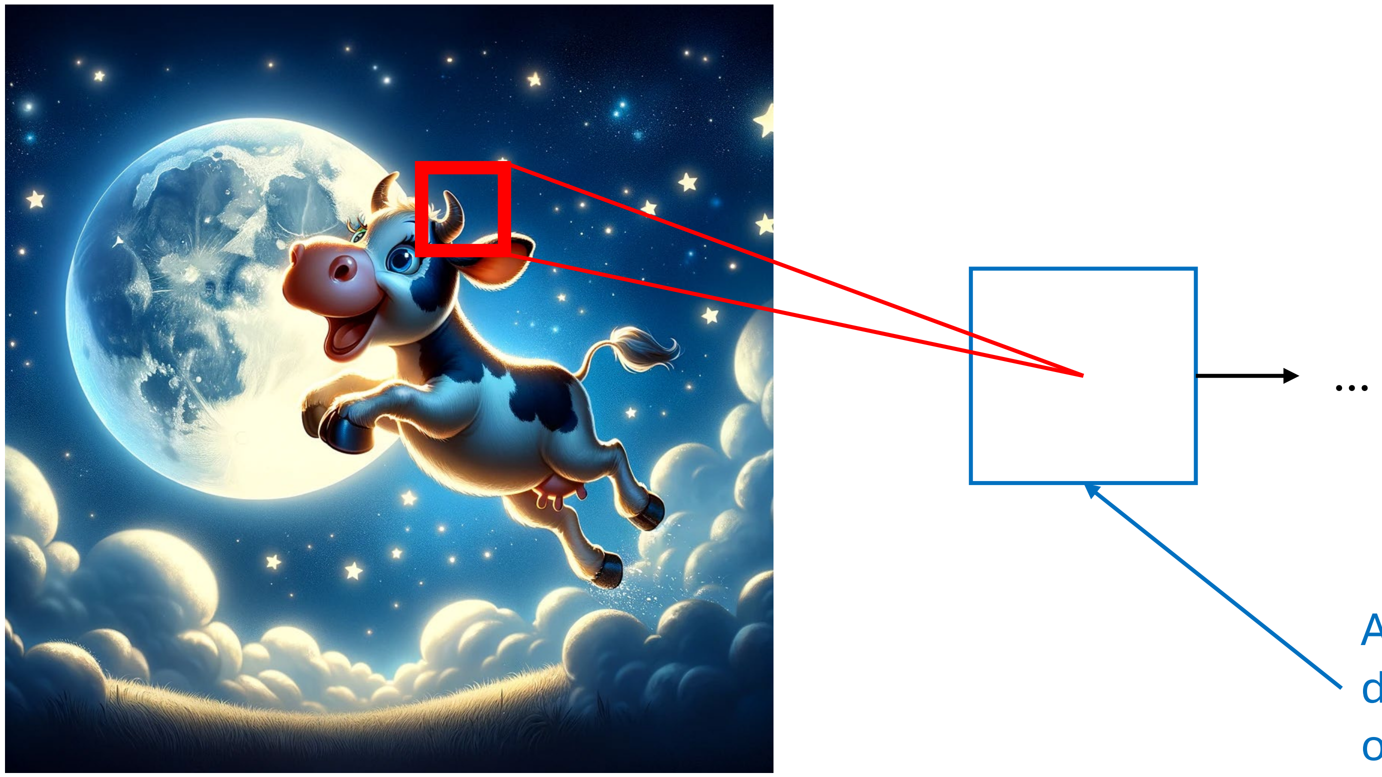
The patch is shifted over by **stride** number of pixels each time.

When the patch reaches the end, it shifts down by **stride** pixels and starts over.

At the end, the **convolutional layer** outputs all the computed values: $(0.2, 0.17, 0.8, -2.1, \dots, 1.3, -0.64, \dots)$

These values are usually represented as a matrix to track the position of the patch they were computed from.

Convolutional Layer (Graphical Depiction)

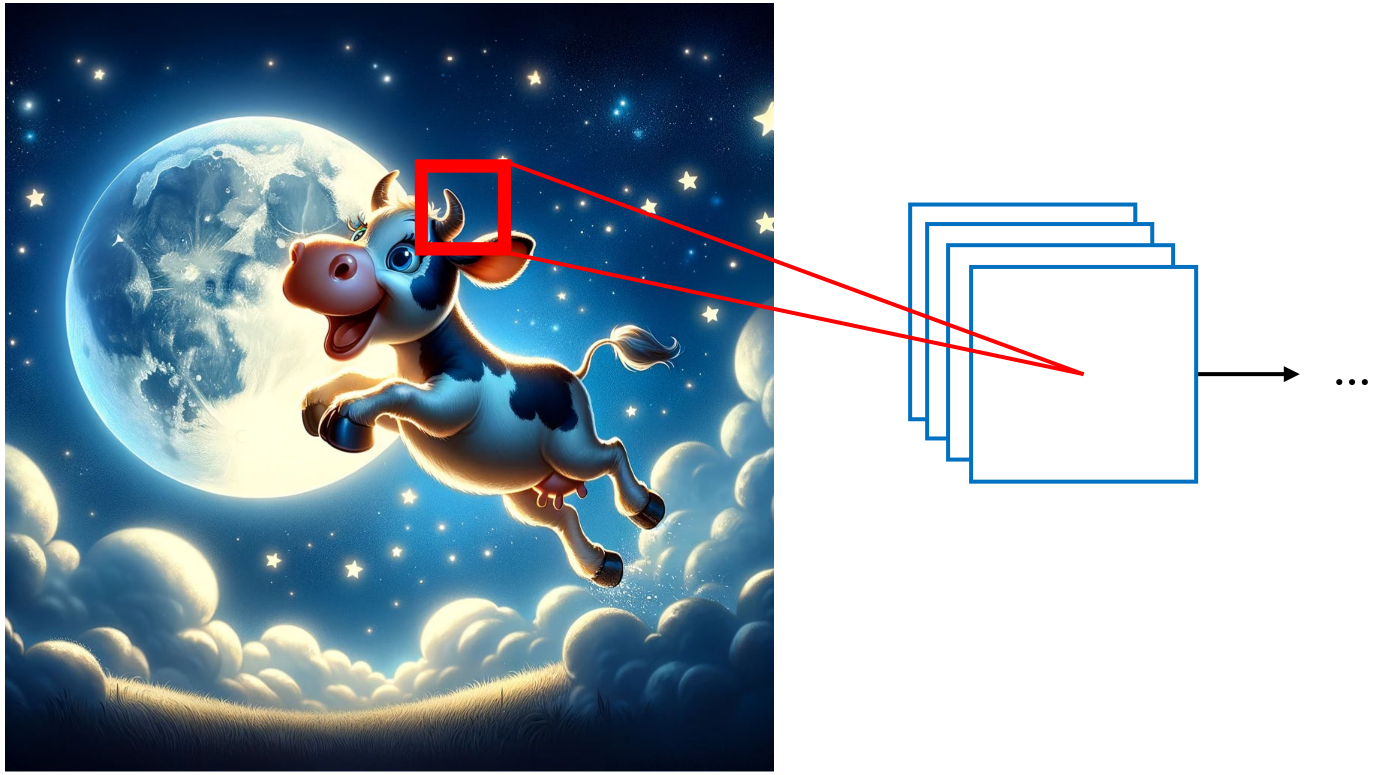


A wider rectangle to denote that this is a matrix of numbers, not a vector.

This represents a convolutional layer (blue) applied to an image.

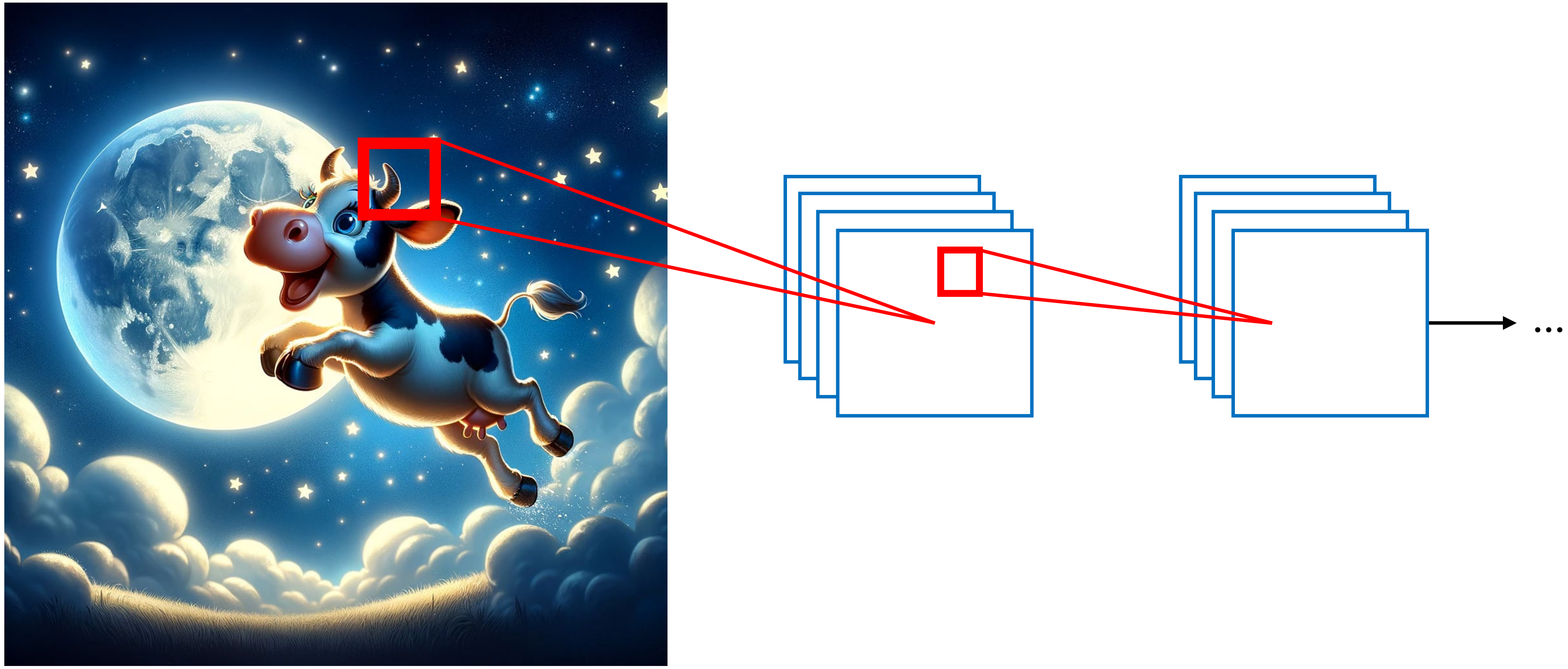
Convolutional Layer

- A convolutional layer with multiple filters is represented using many stacked boxes:



Convolutional Layer

- Convolutional layers can be applied in a sequence!



Max Pooling Layers

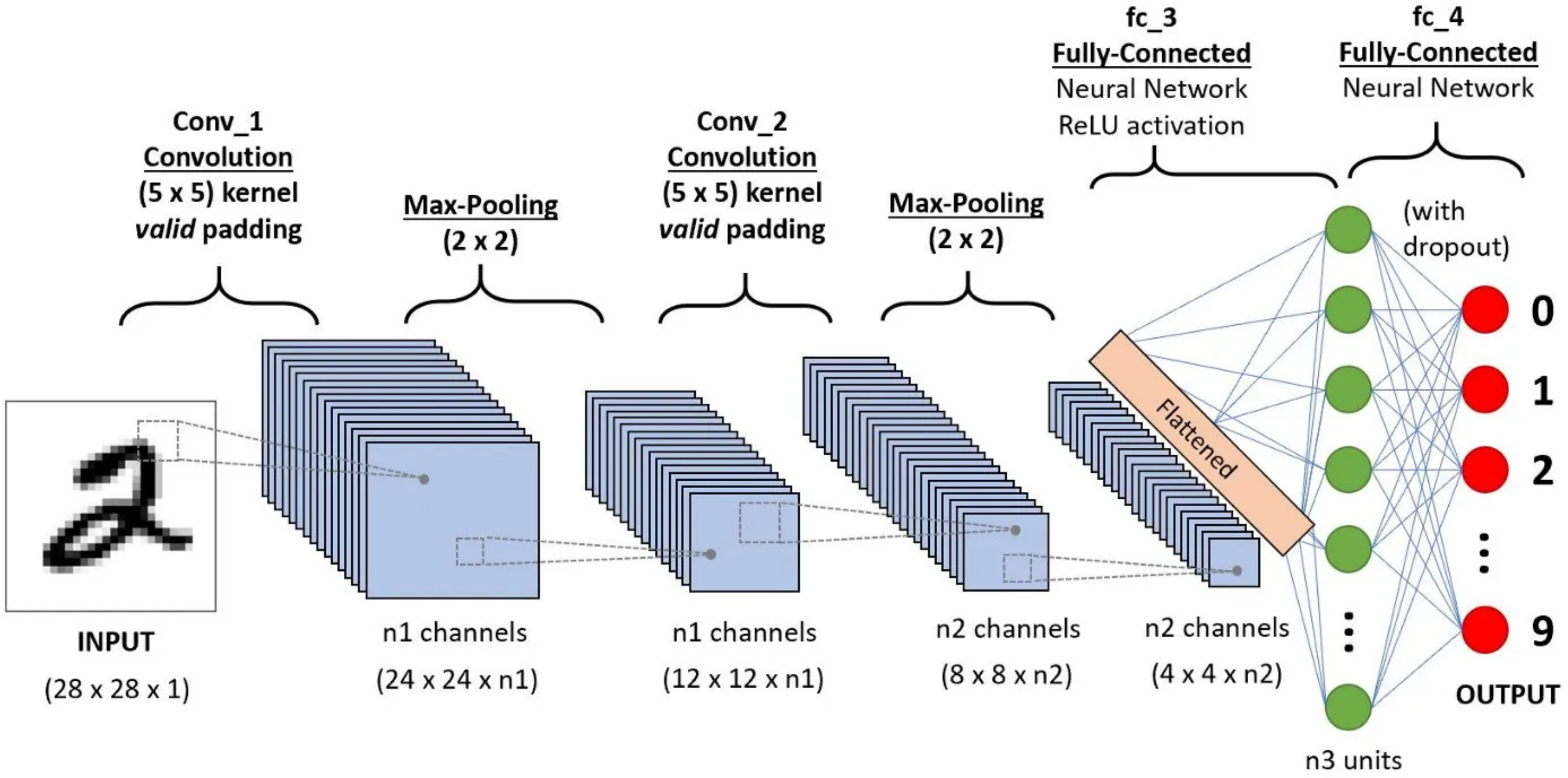
- When using convolutional layers with many filters, you can end up with more outputs from the convolutional layer than there were pixels in the original image!
- To make the number of values more manageable, a **max pooling** layer can be used to downsample (reduce) the number of features.
- A max pooling layer acts like a convolutional layer, but without any parameters.
  - For each patch, it returns the maximum value within the patch.
  - Other pooling layers (e.g., average pooling layers) compute other fixed functions of a patch (e.g., the average value in the patch)
  - A max pooling layer typically has a relatively wide stride and/or patch.
    - For example, a 2x2 patch with no overlap between patches quarters the number of values.

Flattening Layers

- Convolutional layers output values in a matrix.
  - One matrix per filter
- Typical feed-forward layers expect values as a vector.
- **Flattening layers** convert the output of convolutional layers into one long vector (rather than a set of matrices).
  - Flattening layers have no tunable parameters, w.

Example from Online:

https://medium.com/@draj0718/convolutional-neural-networks-cnn-architectures-explained-716fb197b243

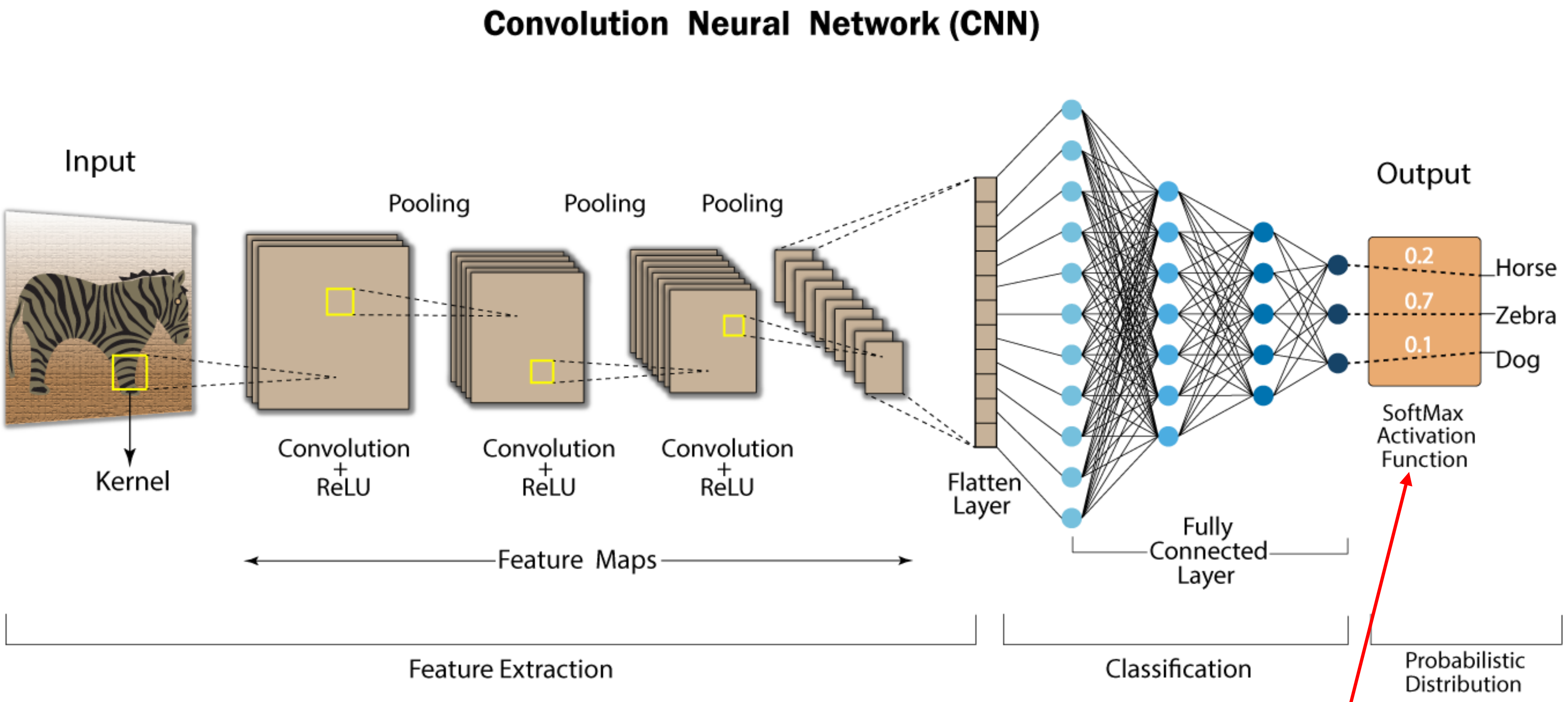


- Number of channels = number of filters
- Some concepts beyond the scope of this class (e.g., padding)
- This model has 10 outputs, one per digit (more on this when we discuss classification)

Why 10 outputs?

Example from Online:

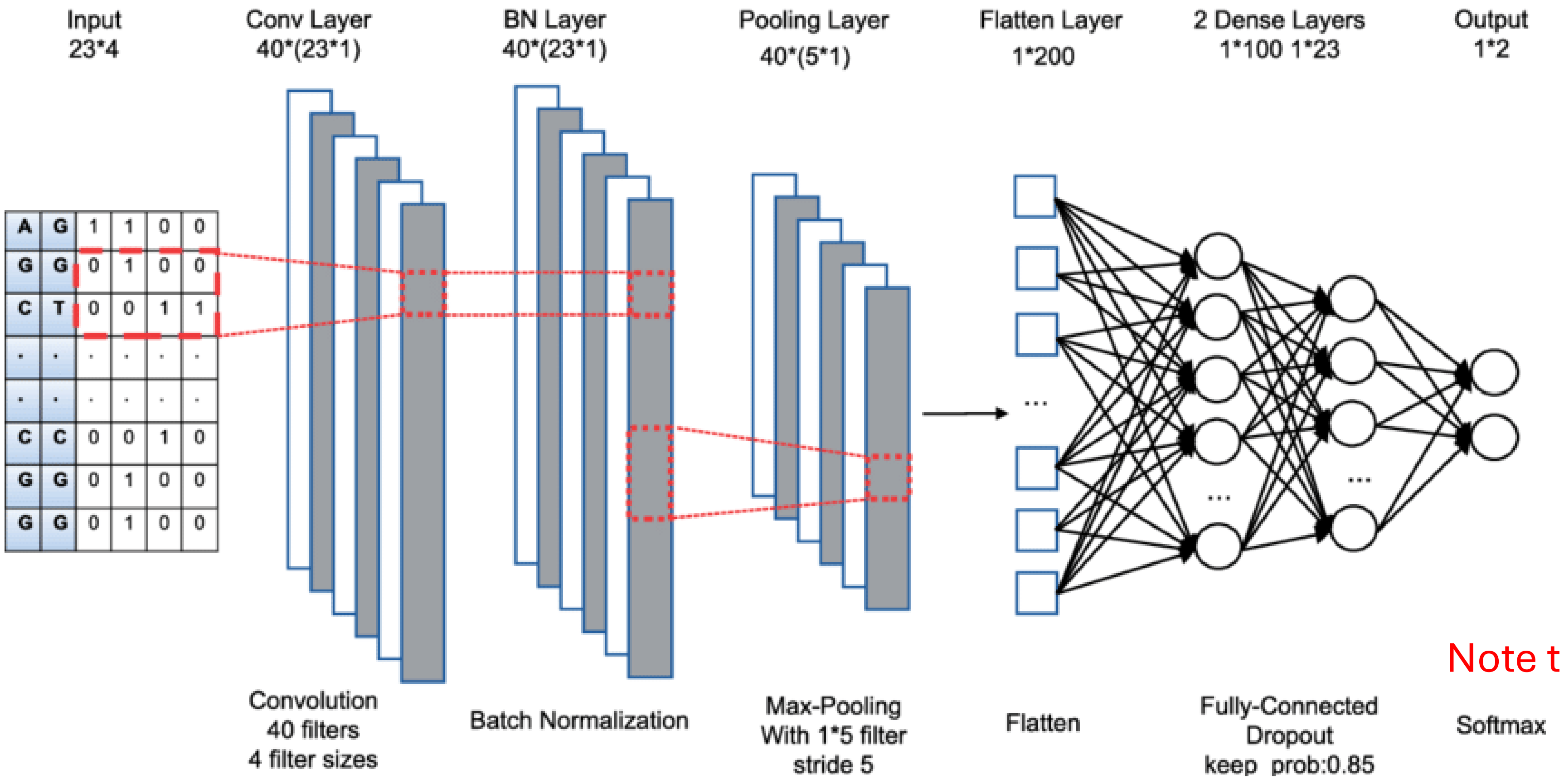
https://developersbreach.com/convolution-neural-network-deep-learning/



What is “softmax” doing here?

Example from Online:

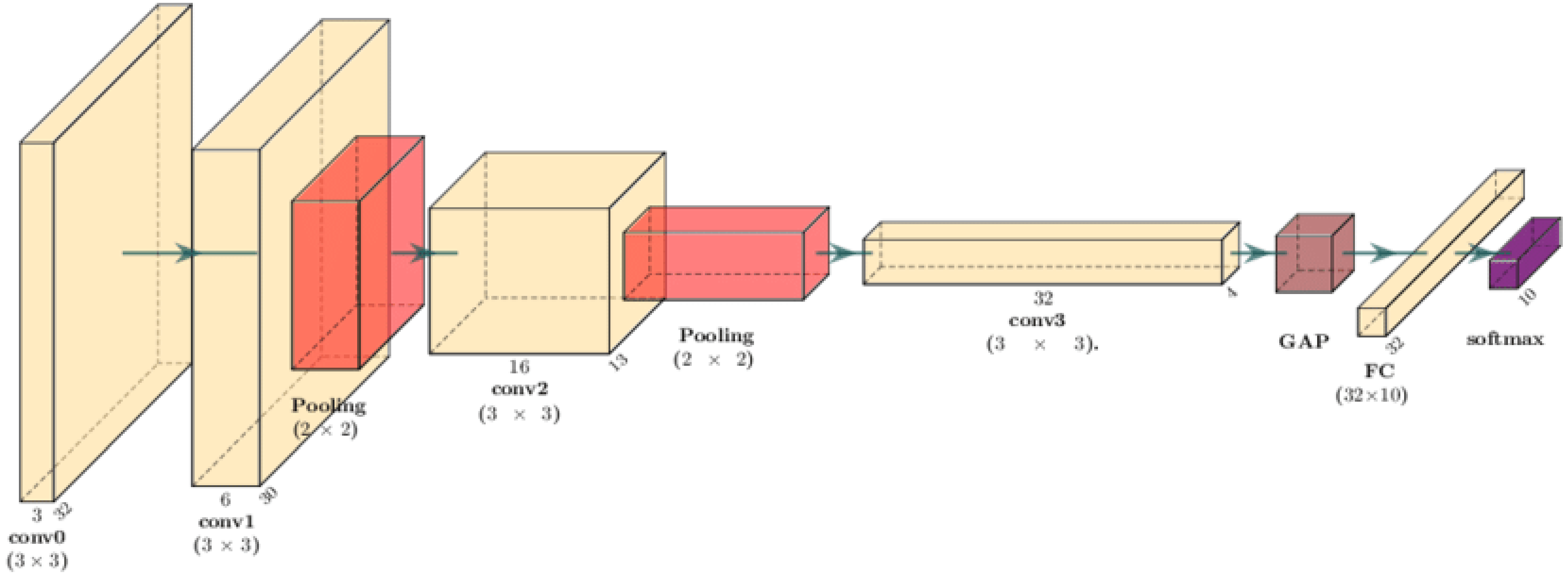
https://www.researchgate.net/figure/The-architecture-of-standard-deep-CNN-CNN-std-for-off-target-prediction-The-input-of_fig2_327641553



Note the softmax again!

Example from Online:

https://www.researchgate.net/figure/CNN-architecture-for-CIFAR-10-SVHN-The-network-consists-of-three-convolution-layers-with_fig3_353568132

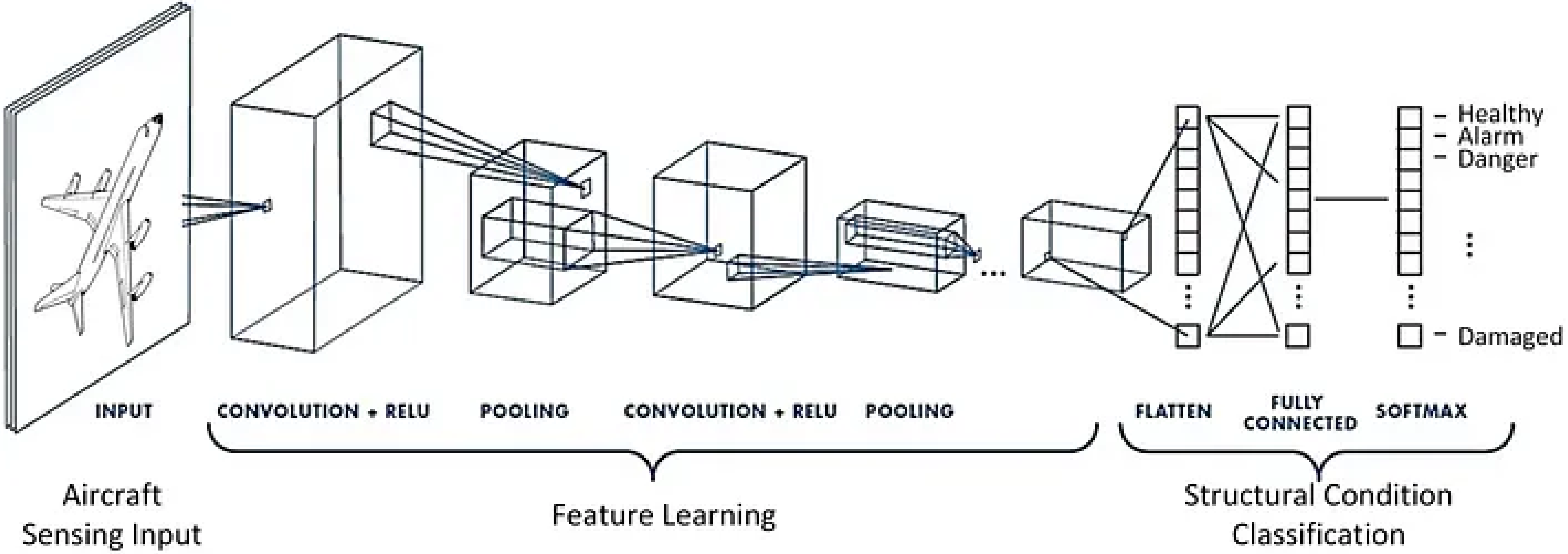


This refers to the size of each patch

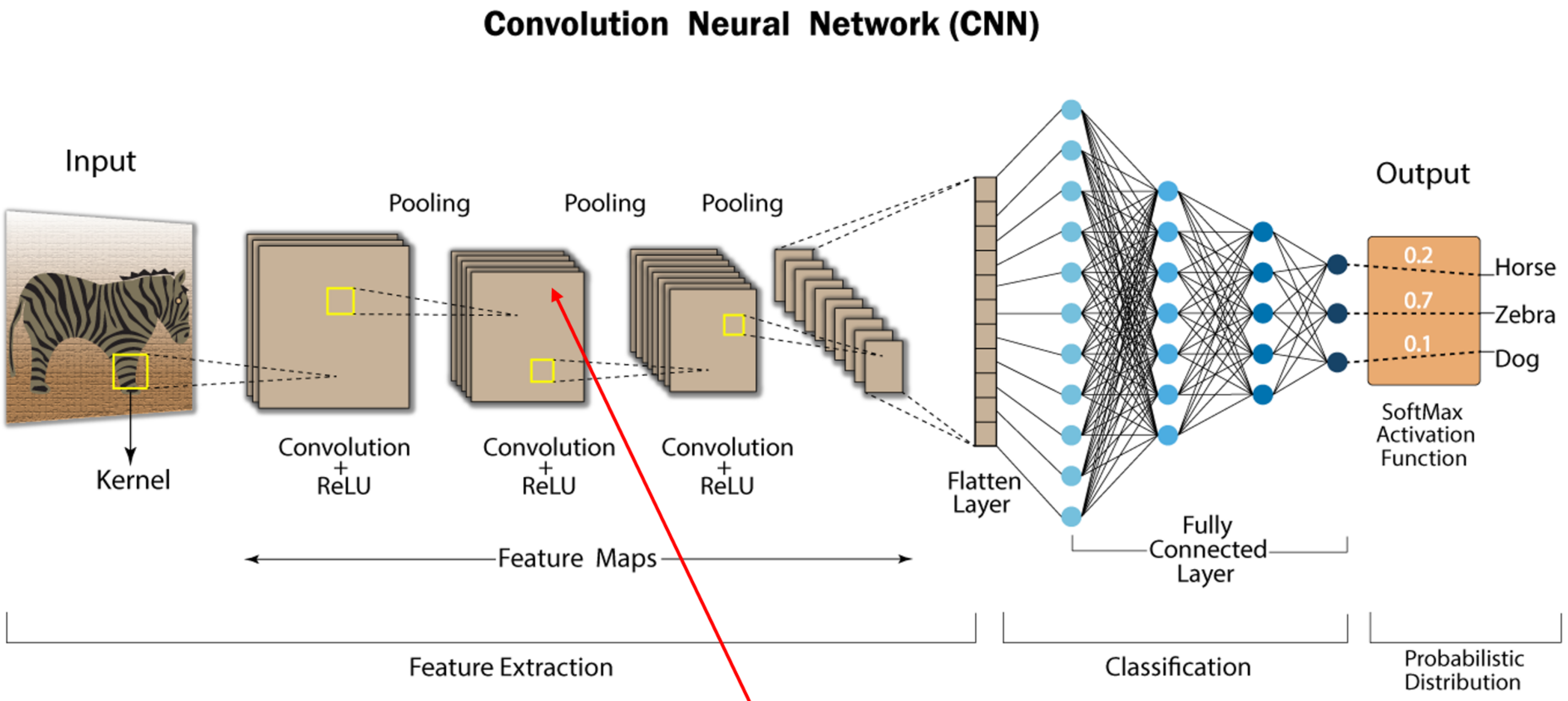
CNN architecture for CIFAR-10/SVHN: The network consists of three convolution layers with 3 × 3 filters, 0 padding and stride 1. The convolution layers are followed by a ReLU non-linearity. We use max pooling in this work with a filter size of 2 × 2, no padding and stride 2 which results in a downsampling of the features by a factor of 2. The three convolution layers have 6, 16 and 32 filters respectively. Finally, a Global Average Pooling (GAP) is applied and a fully connected (fc) outputs logits over the number of classes.

Example from Online:

https://medium.com/analytics-vidhya/convolutional-neuronal-network-with-keras-tuner-on-cifar-10-b4271ca4643d



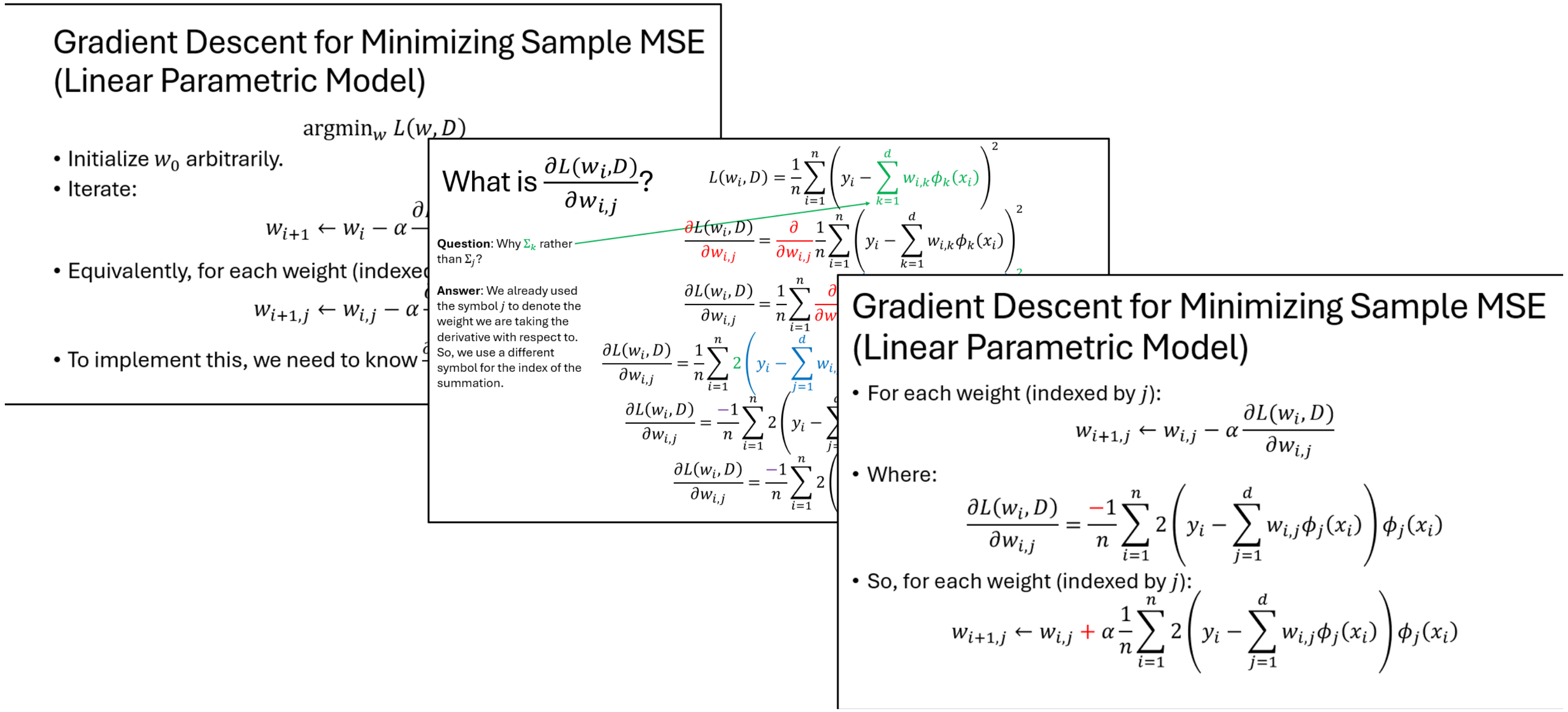
Coming up…



To train the model, we need the derivative of the loss function with respect to each weight.
How can we compute the derivative with respect to this weight in the model?

Our old approach (manual derivation) is error prone and can be specific to a network architecture

Manual derivation of gradient



Gradient Descent for Minimizing Sample MSE (Linear Parametric Model)

$$\operatorname{argmin}_w L(w,D)$$

- Initialize w_0 arbitrarily.
- Iterate:

$$w_{i+1} \leftarrow w_i - \alpha \frac{\partial}{}$$

- Equivalently, for each weight (indexe

$$w_{i+1,j} \leftarrow w_{i,j} - \alpha$$

- To implement this, we need to know

What is $\frac{\partial L(w_i,D)}{\partial w_{i,j}}$?

Question: Why Σ_k rather than Σ_j?

Answer: We already used the symbol j to denote the weight we are taking the derivative with respect to. So, we use a different symbol for the index of the summation.

$$L(w_i,D) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \sum_{k=1}^{d} w_{i,k}\phi_k(x_i)\right)^2$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \sum_{k=1}^{d} w_{i,k}\phi_k(x_i)\right)^2$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial w}$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=1}^{d} w_{i,}\right.$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{-1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum\right.$$

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{-1}{n}\sum_{i=1}^{n} 2\left(\right.$$

Gradient Descent for Minimizing Sample MSE (Linear Parametric Model)

- For each weight (indexed by j):

$$w_{i+1,j} \leftarrow w_{i,j} - \alpha \frac{\partial L(w_i,D)}{\partial w_{i,j}}$$

- Where:

$$\frac{\partial L(w_i,D)}{\partial w_{i,j}} = \frac{-1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=1}^{d} w_{i,j}\phi_j(x_i)\right)\phi_j(x_i)$$

- So, for each weight (indexed by j):

$$w_{i+1,j} \leftarrow w_{i,j} + \alpha \frac{1}{n}\sum_{i=1}^{n} 2\left(y_i - \sum_{j=1}^{d} w_{i,j}\phi_j(x_i)\right)\phi_j(x_i)$$

Chain Rule (Review)

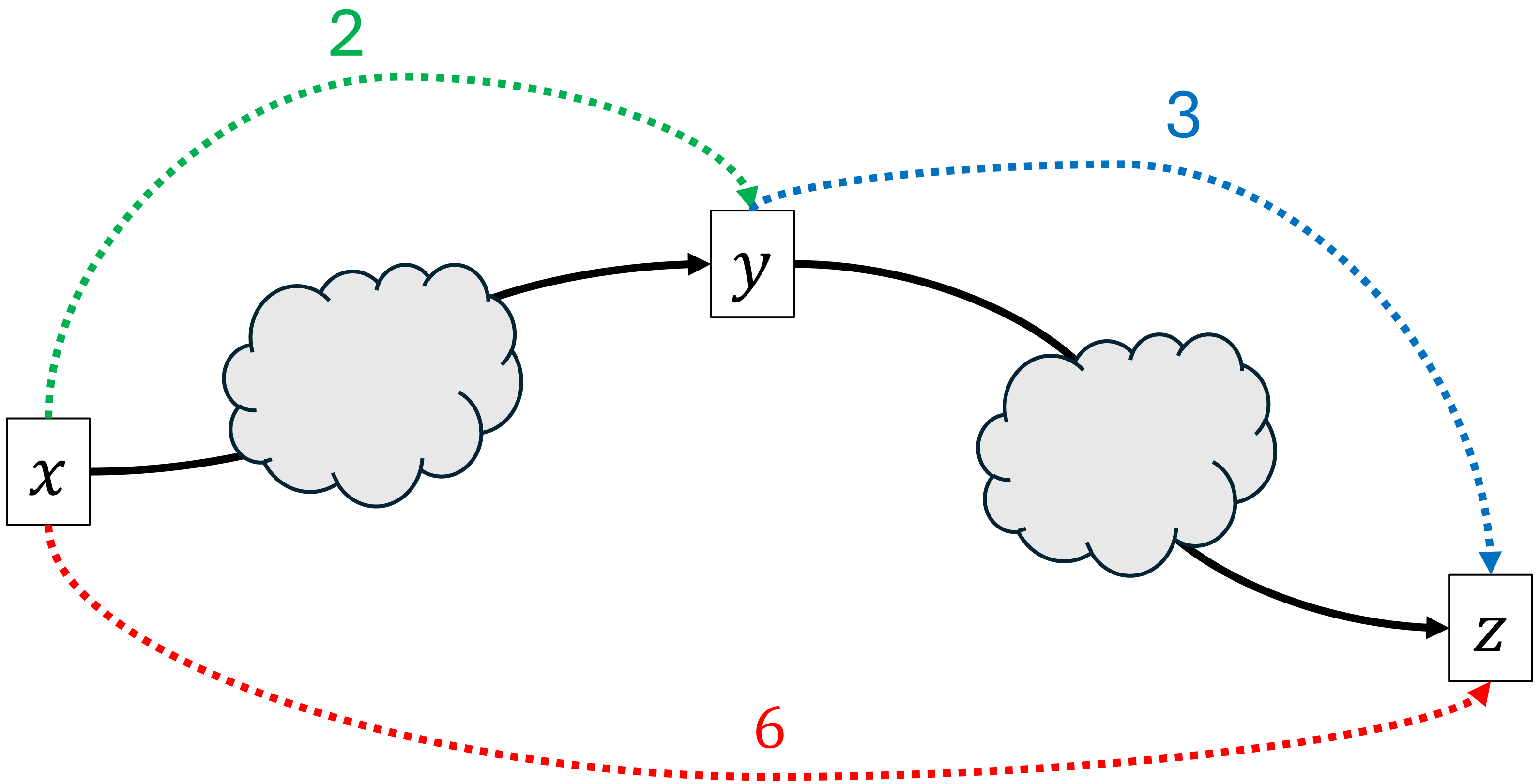
$$\frac{df\left(g(x)\right)}{dx} = \frac{df(x)}{dg(x)}\frac{dg(x)}{dx}$$

or

$$\frac{dz}{dx} = \frac{dz}{dy}\ \frac{dy}{dx}$$

Chain Rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



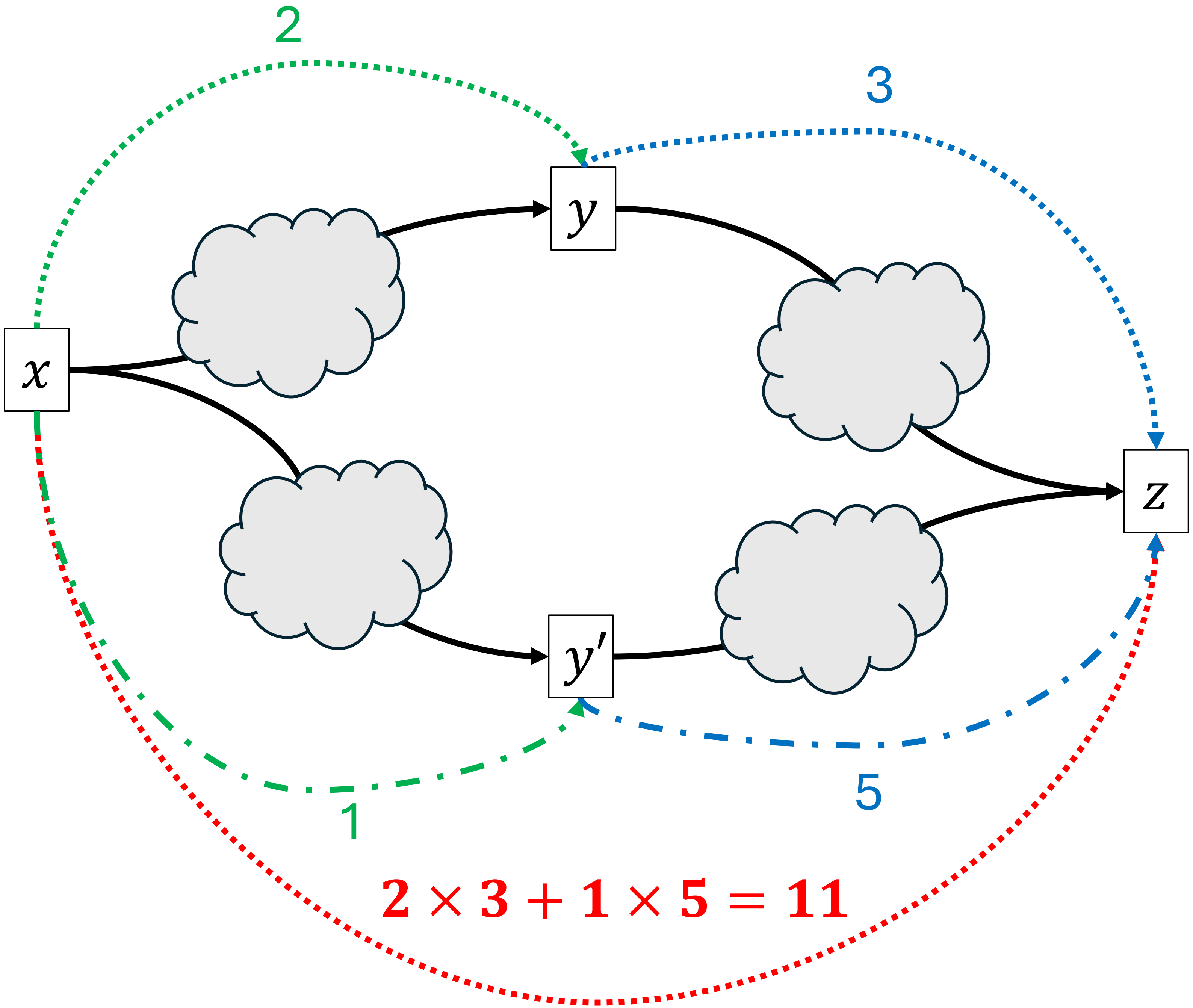
$\frac{dz}{dx}$ – How does changing x change z? =6 (adding ϵ to x increases z by 6ϵ)

$\frac{dy}{dx}$ – How does changing x change y? =2 (adding ϵ to x increases y by 2ϵ)

$\frac{dz}{dy}$ – How does changing y change z? =3 (adding ϵ to y increases z by 3ϵ)

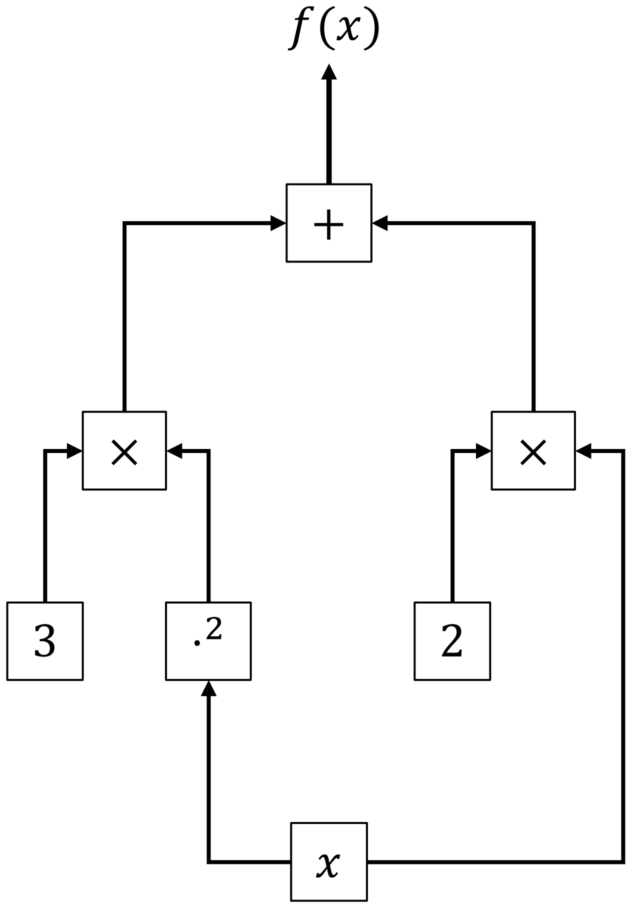
Chain Rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} + \frac{dz}{dy'}\frac{dy'}{dx}$$



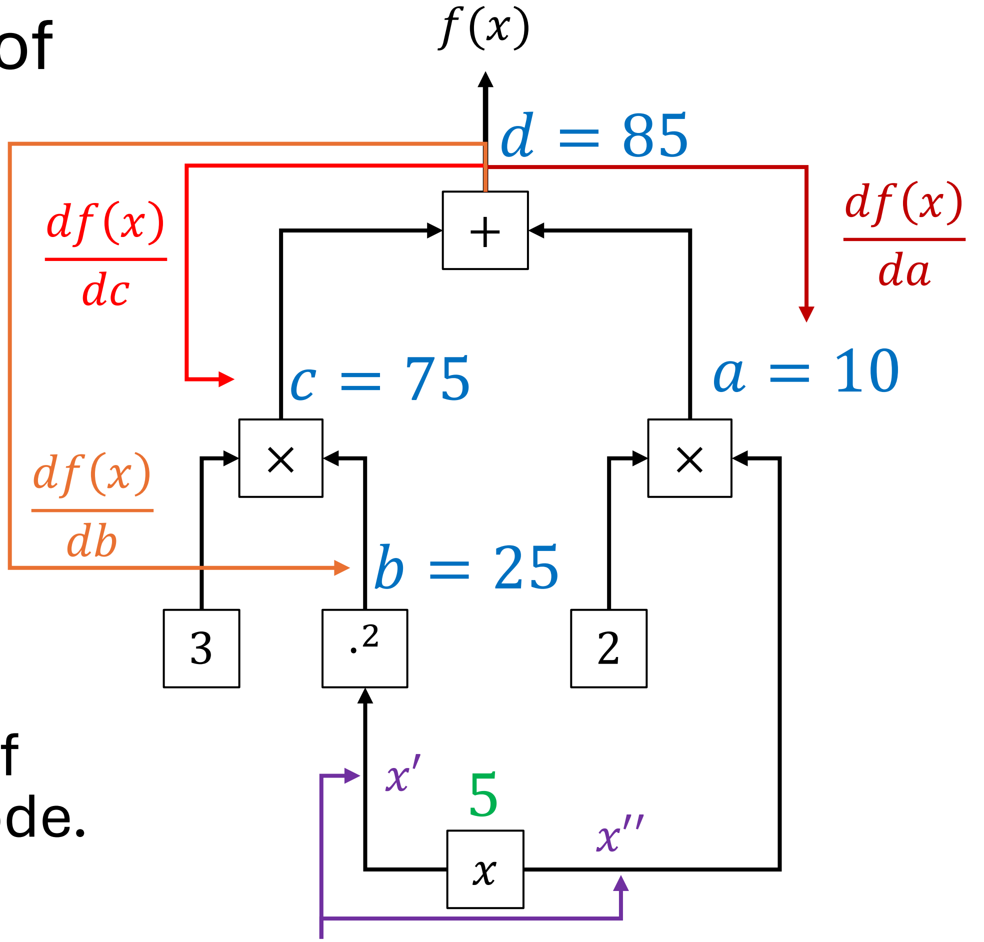
Expression Trees

- Math expressions like function definitions can be converted into *expression trees*.
  - Each internal node is a math operator.
  - Each leaf node is a constant or variable.
- Example: $f(x) = 3x^2 + 2x$



Automatic Differentiation

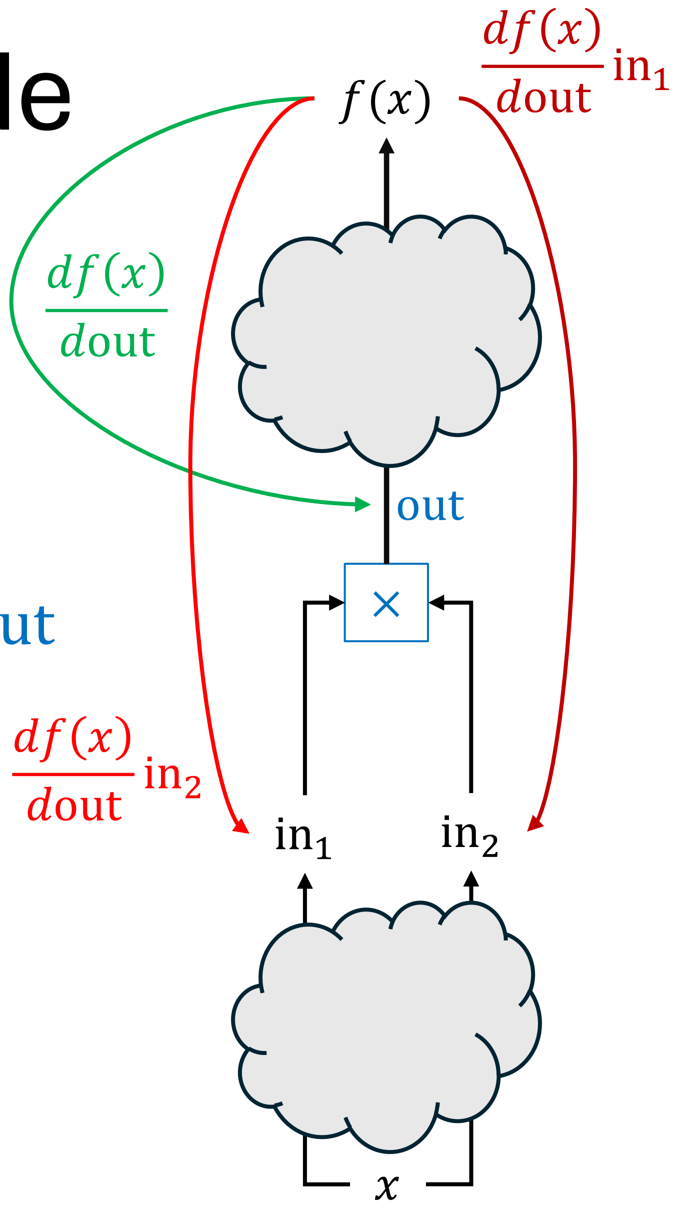
- **Goal**: Compute $\frac{df(x)}{dx}$, for some value of x
  - Example: $x = 5$
- Step 1: Run a “forwards pass”
  - Evaluate the expression tree, computing values from the bottom to the top.
- Step 2: Run a “backwards pass”
  - Loop over nodes from the top to the bottom.
  - For each node, compute the derivative of $f(x)$ with respect to each *input* of the node.



We write x' and x'' so that we can talk about the two paths, $\frac{df(x)}{dx'}$ and $\frac{df(x)}{dx''}$

Backwards Pass: Multiplication Node

- We want to compute $\partial f(x)/\partial \mathrm{in}_1$ and $\partial f(x)/\partial \mathrm{in}_2$
- Assume that we know:
  - The value of the inputs: in_1 and in_2
    - These were computed during the forwards pass
  - The derivative of $f(x)$ *with respect to* (w.r.t.) the output out of the multiplication function, $\times$.
    - This is $\frac{df(x)}{d\mathrm{out}}$
    - This was computed earlier in the backwards pass by the node “above” the multiplication node.
- $\frac{df(x)}{d\mathrm{in}_1} = \frac{df(x)}{d\mathrm{out}} \frac{d\mathrm{out}}{d\mathrm{in}_1} = \frac{df(x)}{d\mathrm{out}} \mathrm{in}_2$
- $\frac{df(x)}{d\mathrm{in}_1} = \frac{df(x)}{d\mathrm{out}} \frac{d\mathrm{out}}{d\mathrm{in}_2} = \frac{df(x)}{d\mathrm{out}} \mathrm{in}_1$

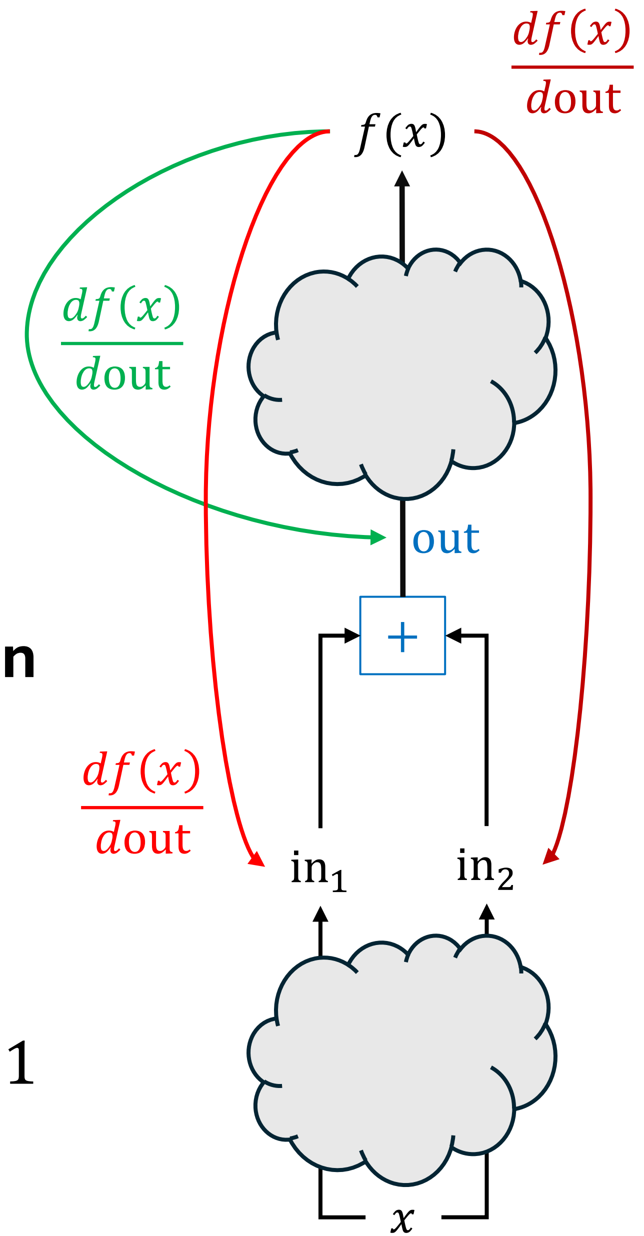


$\frac{df(x)}{d\mathrm{out}}\mathrm{in}_1$
$f(x)$
$\frac{df(x)}{d\mathrm{out}}$
out
$\times$
$\frac{df(x)}{d\mathrm{out}}\mathrm{in}_2$
in_1
in_2
x

Backwards Pass: **Addition** Node

- We want to compute $\partial f(x)/\partial \text{in}_1$ and $\partial f(x)/\partial \text{in}_2$
- Assume that we know:
  - The value of the inputs: in_1 and in_2
    - These were computed during the forwards pass
  - The derivative of $f(x)$ w.r.t. the output out of the **addition** function, $+$.
    - This is $\frac{df(x)}{d\text{out}}$
    - This was computed earlier in the backwards pass by the node "above" the multiplication node.
- $\frac{df(x)}{d\text{in}_1} = \frac{df(x)}{d\text{out}} \frac{d\text{out}}{d\text{in}_1} = \frac{df(x)}{d\text{out}}$
- $\frac{df(x)}{d\text{in}_1} = \frac{df(x)}{d\text{out}} \frac{d\text{out}}{d\text{in}_2} = \frac{df(x)}{d\text{out}}$

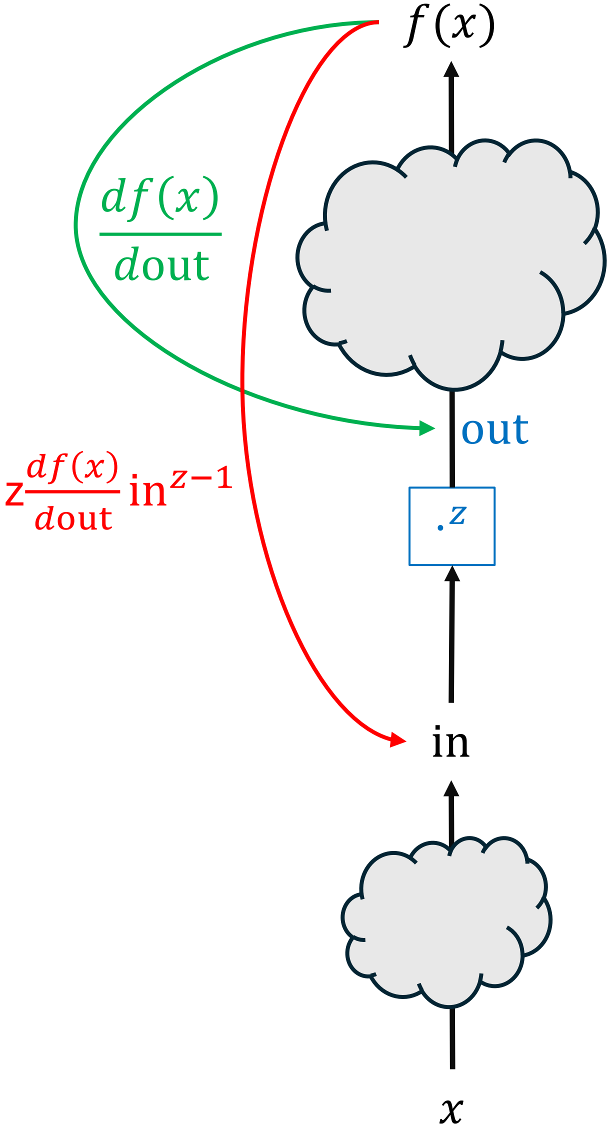
$$\frac{d\text{out}}{d\text{in}_1} = 1$$


$\frac{df(x)}{d\text{out}}$
$f(x)$
$\frac{df(x)}{d\text{out}}$
out
+
$\frac{df(x)}{d\text{out}}$
in_1
in_2
x

Backwards Pass: **Exponent** Node

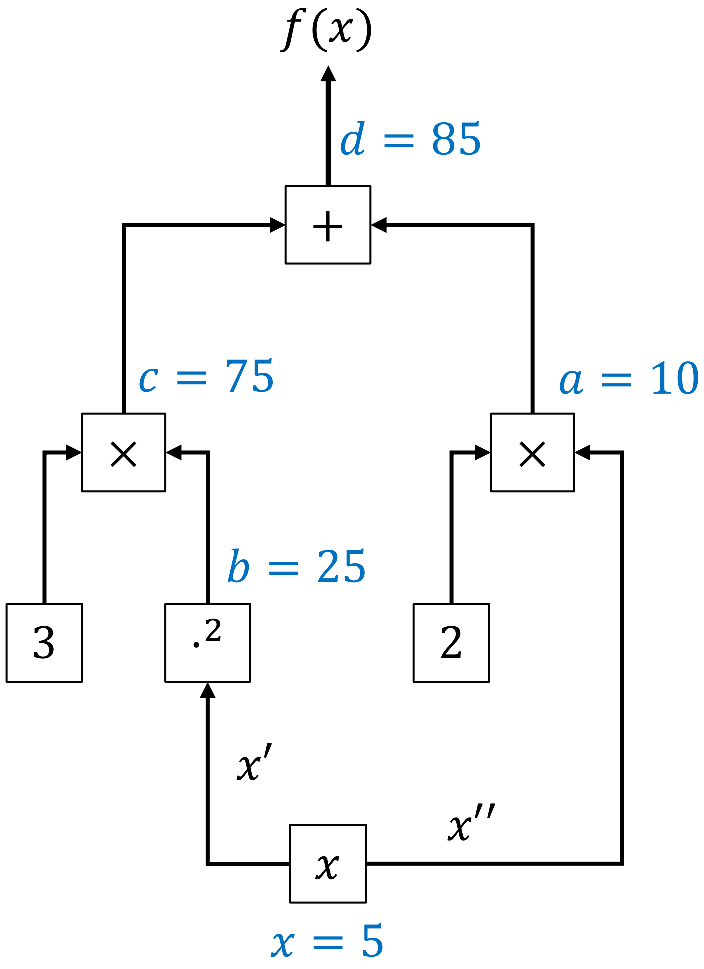
- We want to compute $\partial f(x)/\partial \mathrm{in}$.
- Assume z is a constant.
- Assume that we know:
  - The value of the input in from the forwards pass
  - The derivative of $f(x)$ w.r.t. the output out of the **exponentiation** function, $(\cdot)^z$.
    - This is $\frac{df(x)}{d\mathrm{out}}$, as was computed previously in the backwards pass

- $\frac{df(x)}{d\mathrm{in}} = \frac{df(x)}{d\mathrm{out}} \frac{d\mathrm{out}}{d\mathrm{in}} = \frac{df(x)}{d\mathrm{out}} \times z \times \mathrm{in}^{z-1}$


$f(x)$
$\frac{df(x)}{d\mathrm{out}}$
out
$z\frac{df(x)}{d\mathrm{out}}\mathrm{in}^{z-1}$
$\cdot^z$
in
x

Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

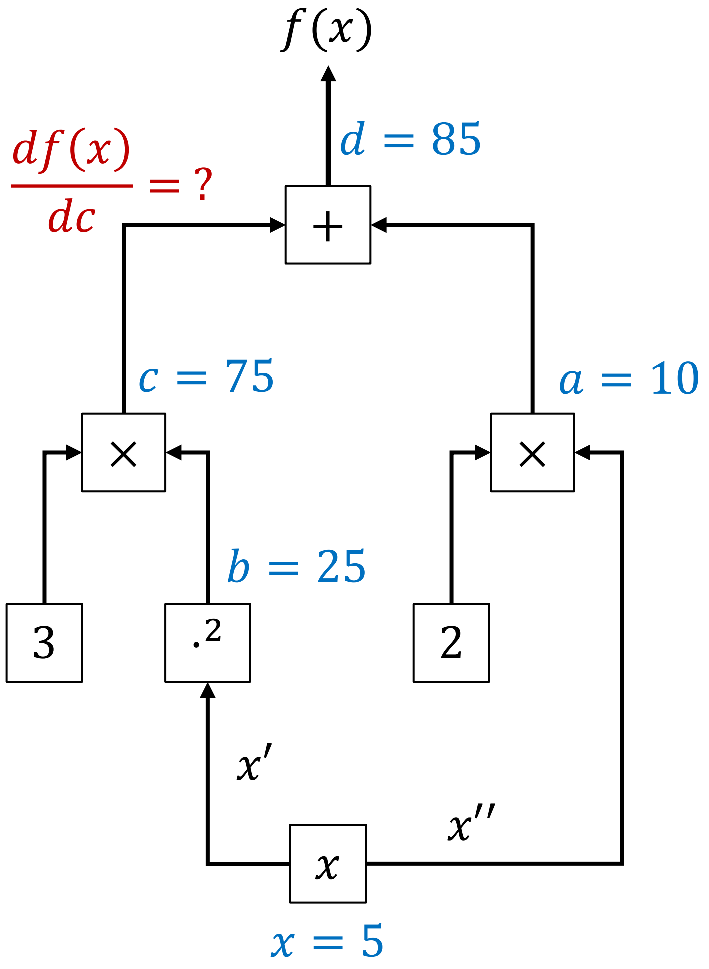
Forwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

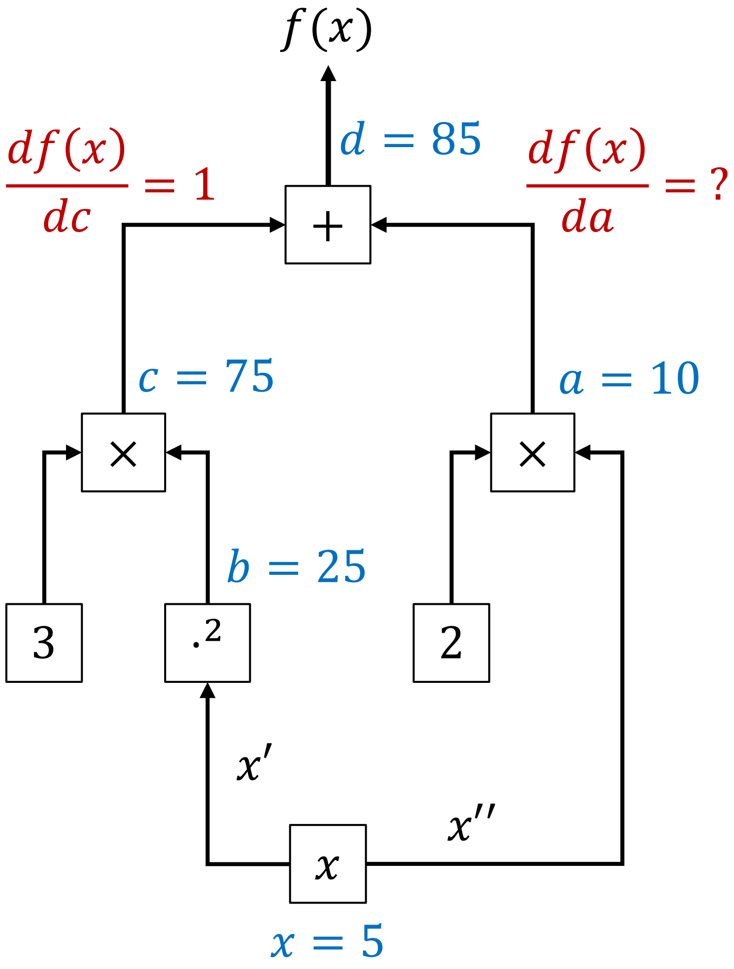
Backwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

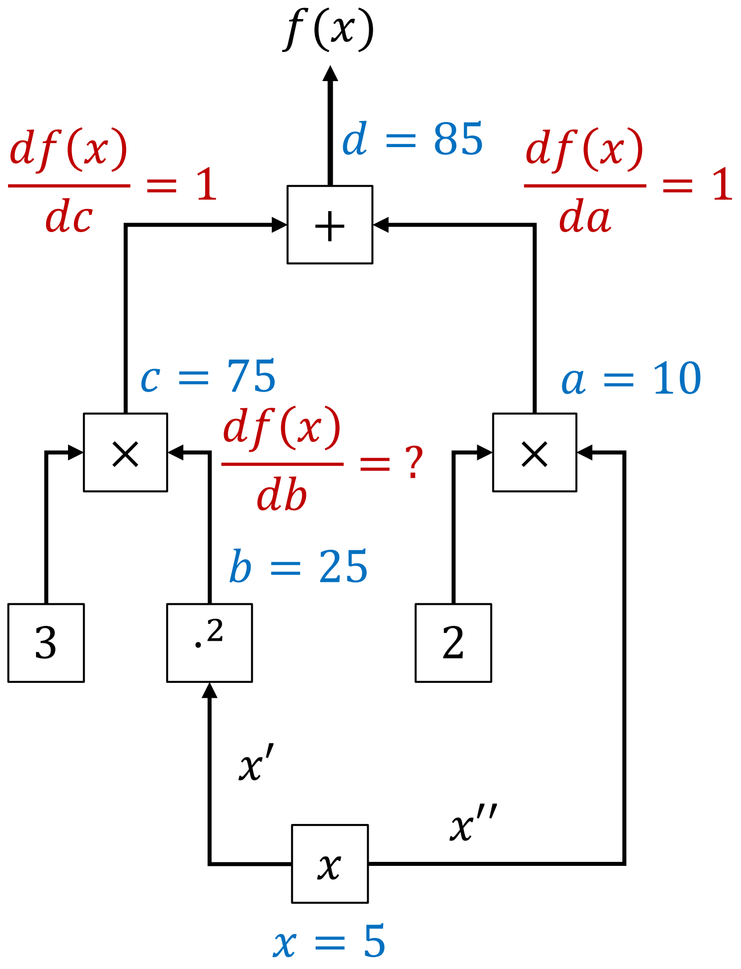
Backwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

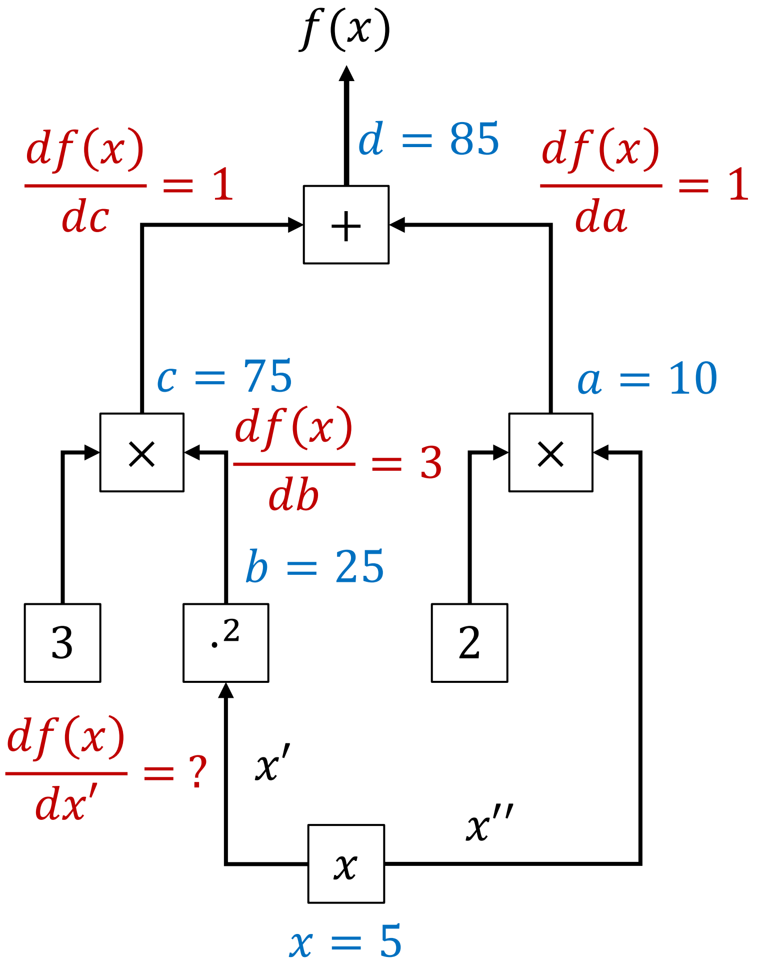
Backwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

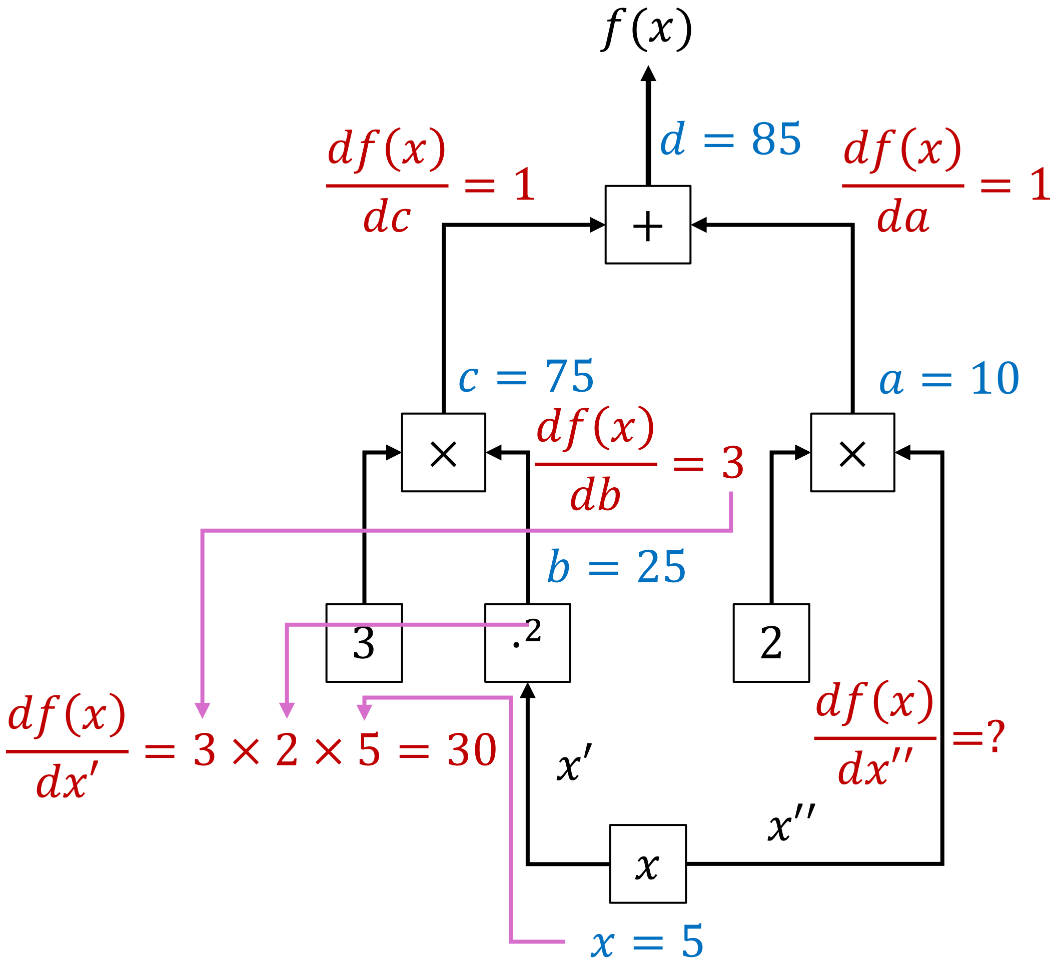
Backwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

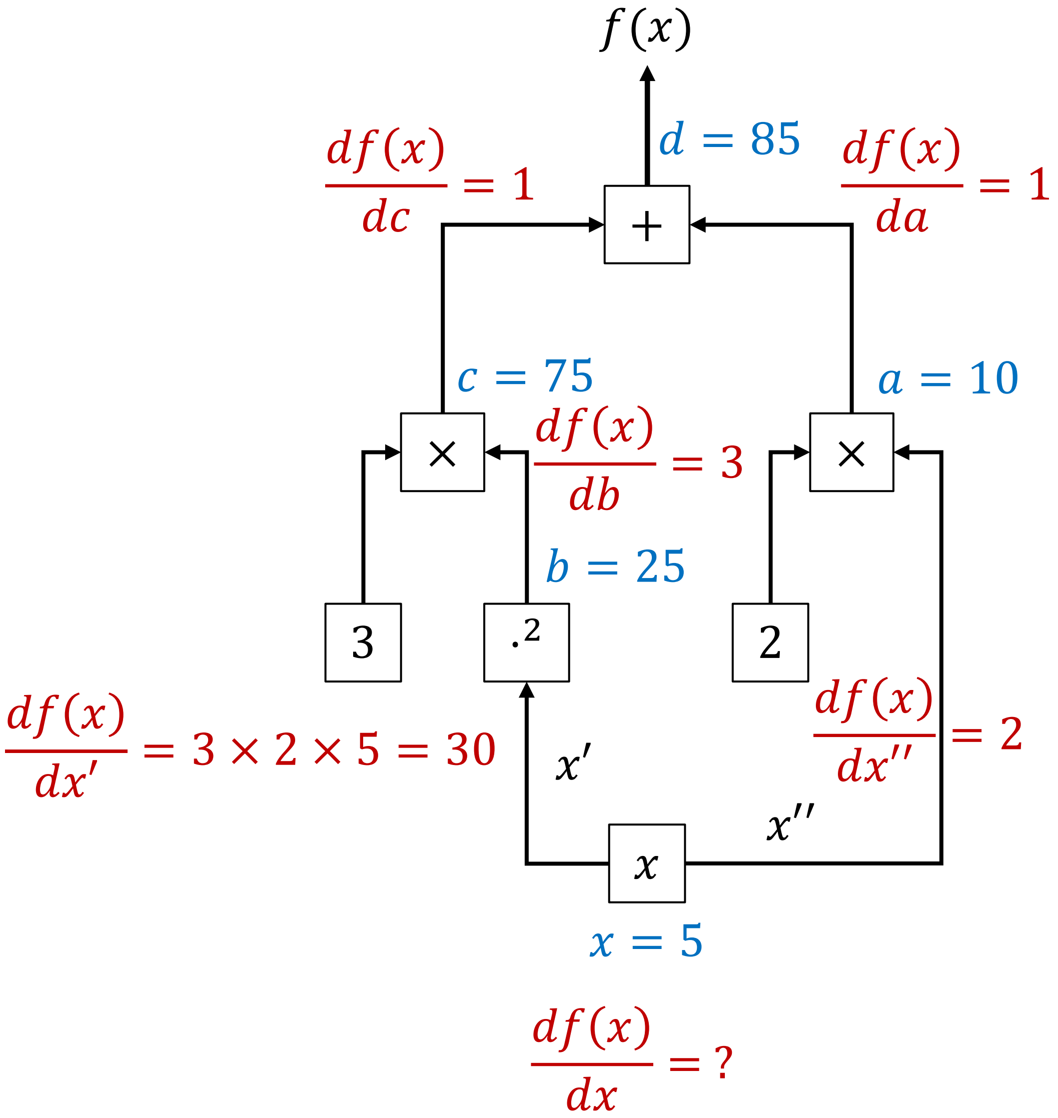
Backwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

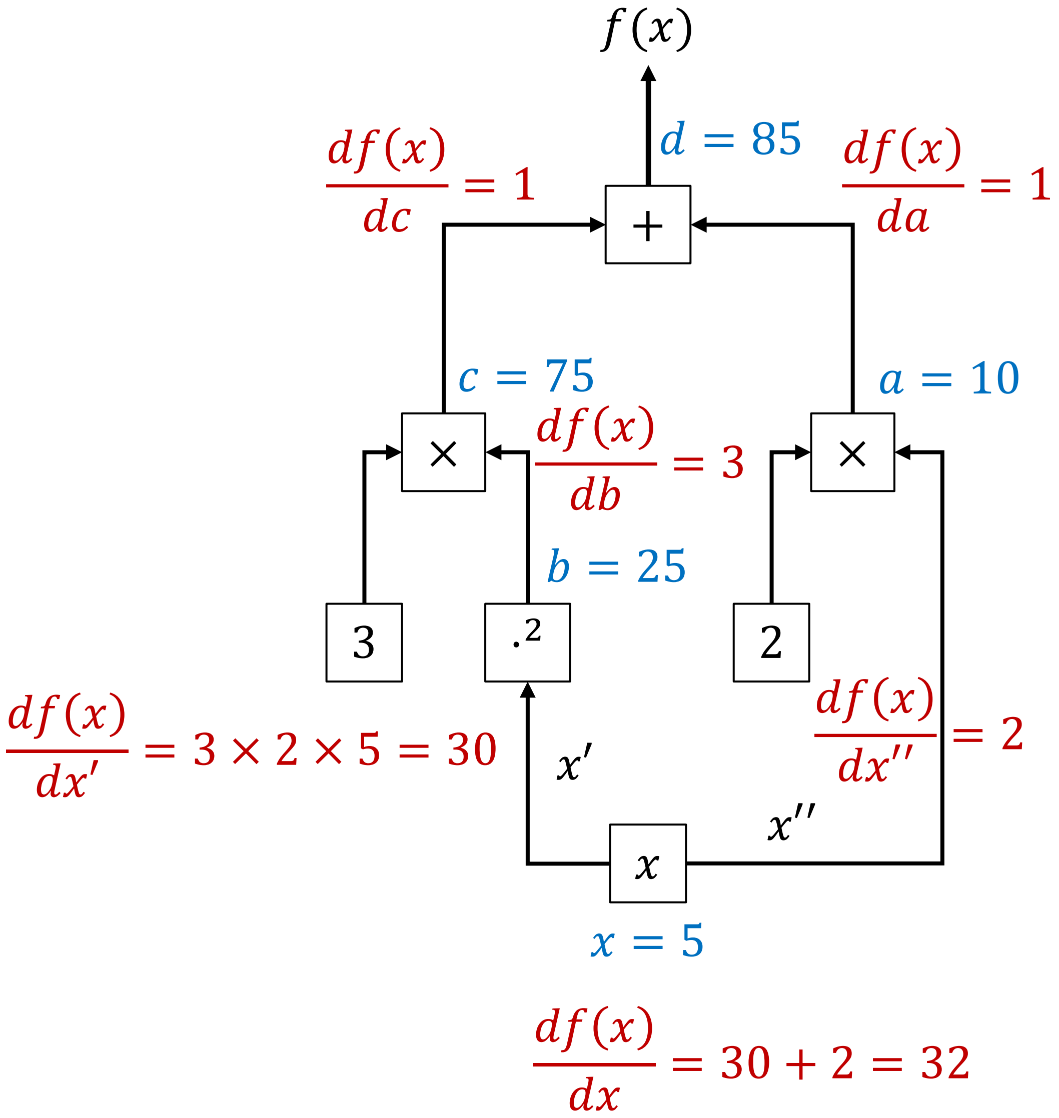
Backwards Pass



Compute $\frac{\mathrm{d}f}{\mathrm{d}x}$ for $f(x) = 3x^2 + 2x$ at $x = 5$

Forwards Pass

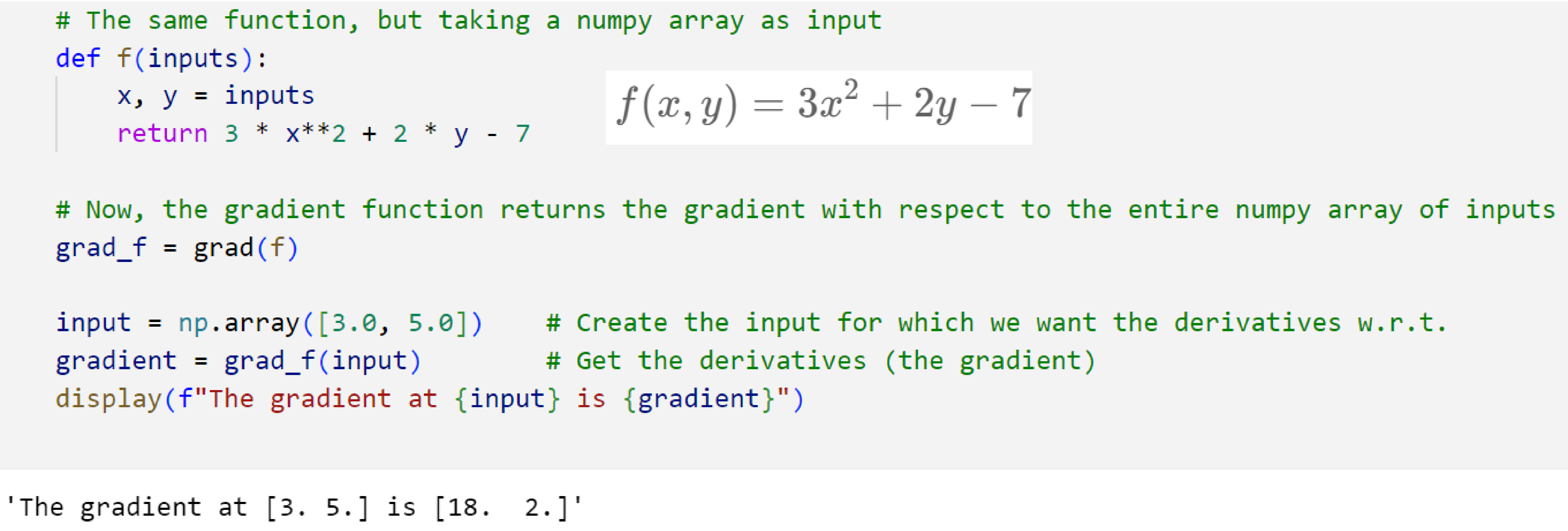
Backwards Pass



Automatic Differentiation

- **Automatic differentiation** tools take functions as input
  - Typically these functions are implemented as code, e.g., *python functions*.
- They can then be used to take the derivative of the function with respect to the arguments (inputs).
- There are several methods for automatic differentiation, with different pros and cons.
  - **Forwards Mode Automatic Differentiation**: Runs one forwards pass (no backwards pass!). Computes the derivative of the output w.r.t. a *single* scalar input.
  - **Reverse Mode Automatic Differentiation**: The strategy we have described.
    - Requires a forward and backwards pass.
    - Can compute the derivative with respect to all inputs with one forwards+backwards pass.
    - This is most common for automatically differentiating ML models and loss functions.
  - Others include **symbolic differentiation** (manipulating the mathematical expressions to calculate expressions for the derivative) and **finite difference methods** (beyond the scope of this course).

Autograd



```
# The same function, but taking a numpy array as input
def f(inputs):
    x, y = inputs
    return 3 * x**2 + 2 * y - 7

# Now, the gradient function returns the gradient with respect to the entire numpy array of inputs
grad_f = grad(f)

input = np.array([3.0, 5.0])    # Create the input for which we want the derivatives w.r.t.
gradient = grad_f(input)        # Get the derivatives (the gradient)
display(f"The gradient at {input} is {gradient}")
```

$$f(x, y) = 3x^2 + 2y - 7$$

```
'The gradient at [3. 5.] is [18.  2.]'
```

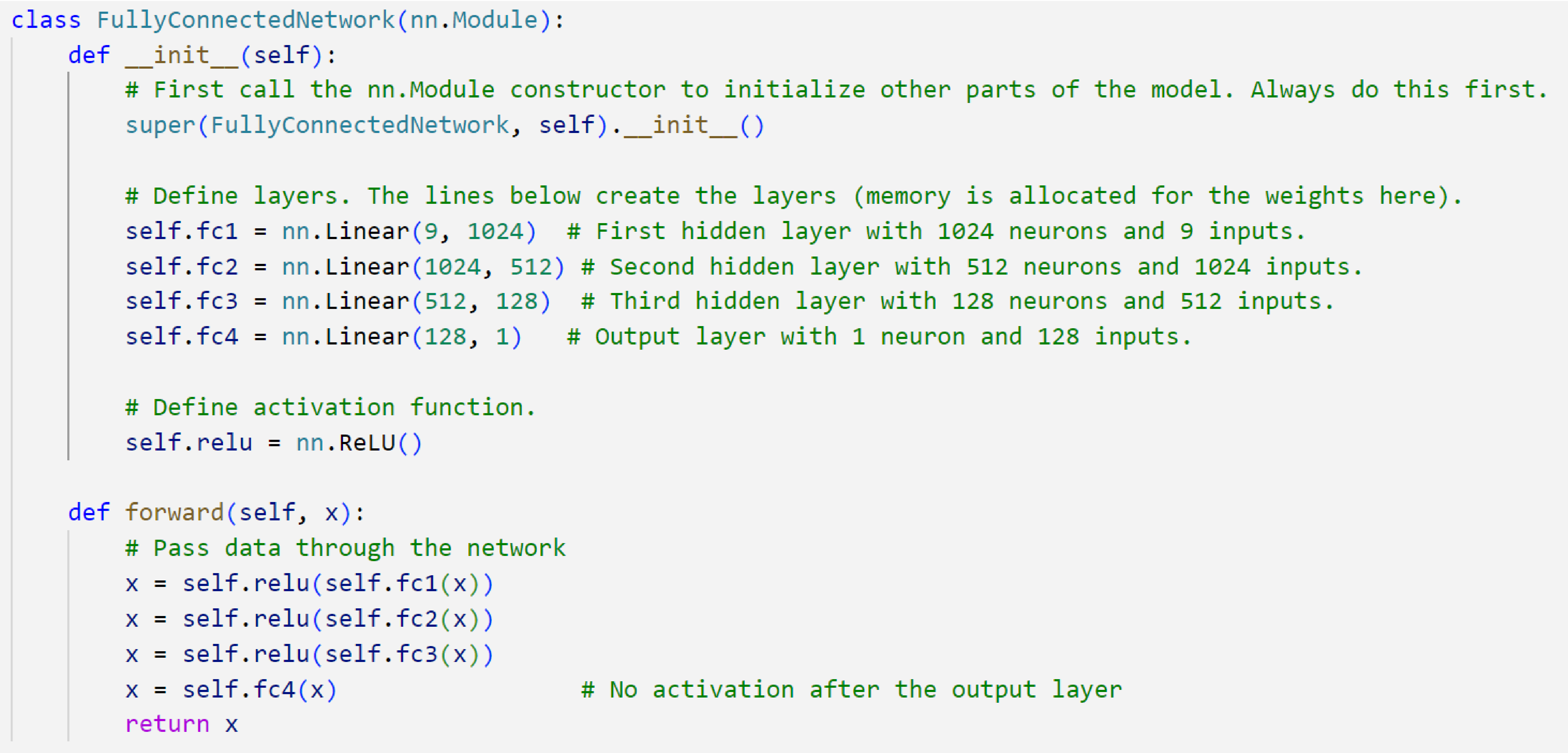
Deep Learning Libraries

- There are many deep learning libraries that extend autograd to:
  - Leverage low-level compiled code for faster runtimes.
  - Enable forward and backwards passes on the GPU rather than CPU (more on this later).
  - Have built-in implementations of
    - Common loss functions
    - Common activation functions
    - Common network layers
      - Fully connected feed-forward
      - Convolutional layers
      - Pooling layers
      - Etc.

Defining a Neural Network Architecture
Defining a Parametric Model

- Extend the `nn.Module` base class
  - The base class provides functionality for tracking trainable parameters (and their gradients), moving parameters to the GPU, saving and loading models, etc.
- Implement two functions:
  - `__init__(self)`: Define the different layers (number of units, number of inputs) and different activation functions that will be used.
  - `forward(self, x)`: Perform a forward pass on input x.
- You do *not* need to implement any gradients or the backwards pass!
  - PyTorch uses reverse mode automatic differentiation to automatically compute gradients.

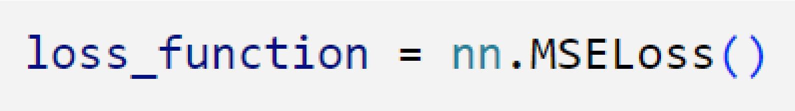
Note: This model is bigger than needed for the GPA prediction problem. This allows us to more easily compare runtimes later, and to show a phenomenon called “overfitting”.



```python
class FullyConnectedNetwork(nn.Module):
    def __init__(self):
        # First call the nn.Module constructor to initialize other parts of the model. Always do this first.
        super(FullyConnectedNetwork, self).__init__()

        # Define layers. The lines below create the layers (memory is allocated for the weights here).
        self.fc1 = nn.Linear(9, 1024)  # First hidden layer with 1024 neurons and 9 inputs.
        self.fc2 = nn.Linear(1024, 512) # Second hidden layer with 512 neurons and 1024 inputs.
        self.fc3 = nn.Linear(512, 128)  # Third hidden layer with 128 neurons and 512 inputs.
        self.fc4 = nn.Linear(128, 1)   # Output layer with 1 neuron and 128 inputs.

        # Define activation function.
        self.relu = nn.ReLU()

    def forward(self, x):
        # Pass data through the network
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)                 # No activation after the output layer
        return x
```

Loss Function

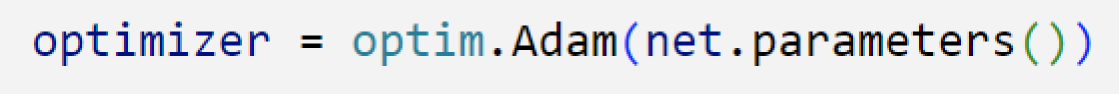
- PyTorch has many built-in loss functions, including MSE:



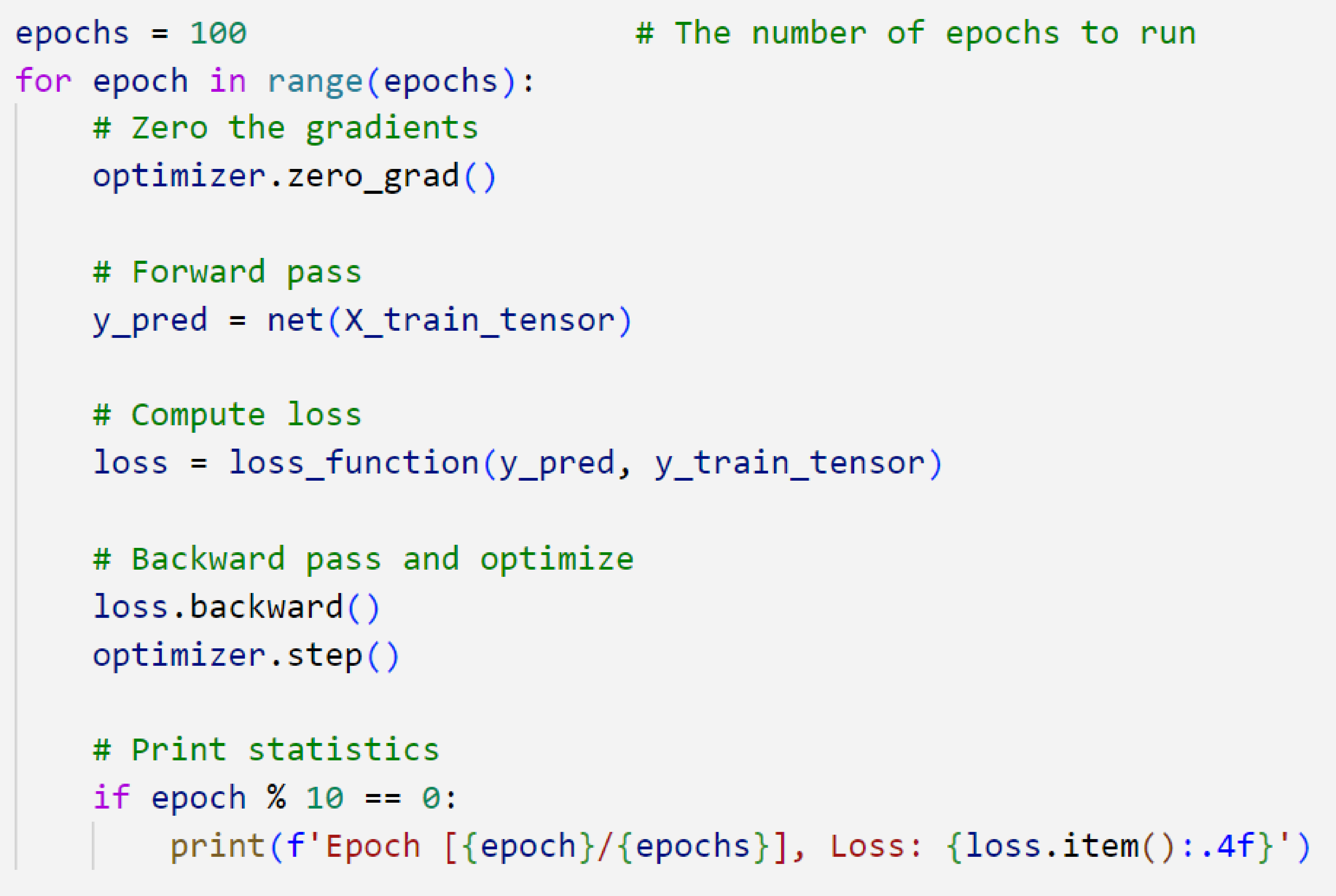
```
loss_function = nn.MSELoss()
```

Optimizer

- PyTorch has many built-in loss optimizers, including gradient descent (SGD), and Adam (SGD with a specific adaptive step size method).
  - Several optimizers are discussed in the Jupyter notebook.
  - Adam is the most common, and what we will use.



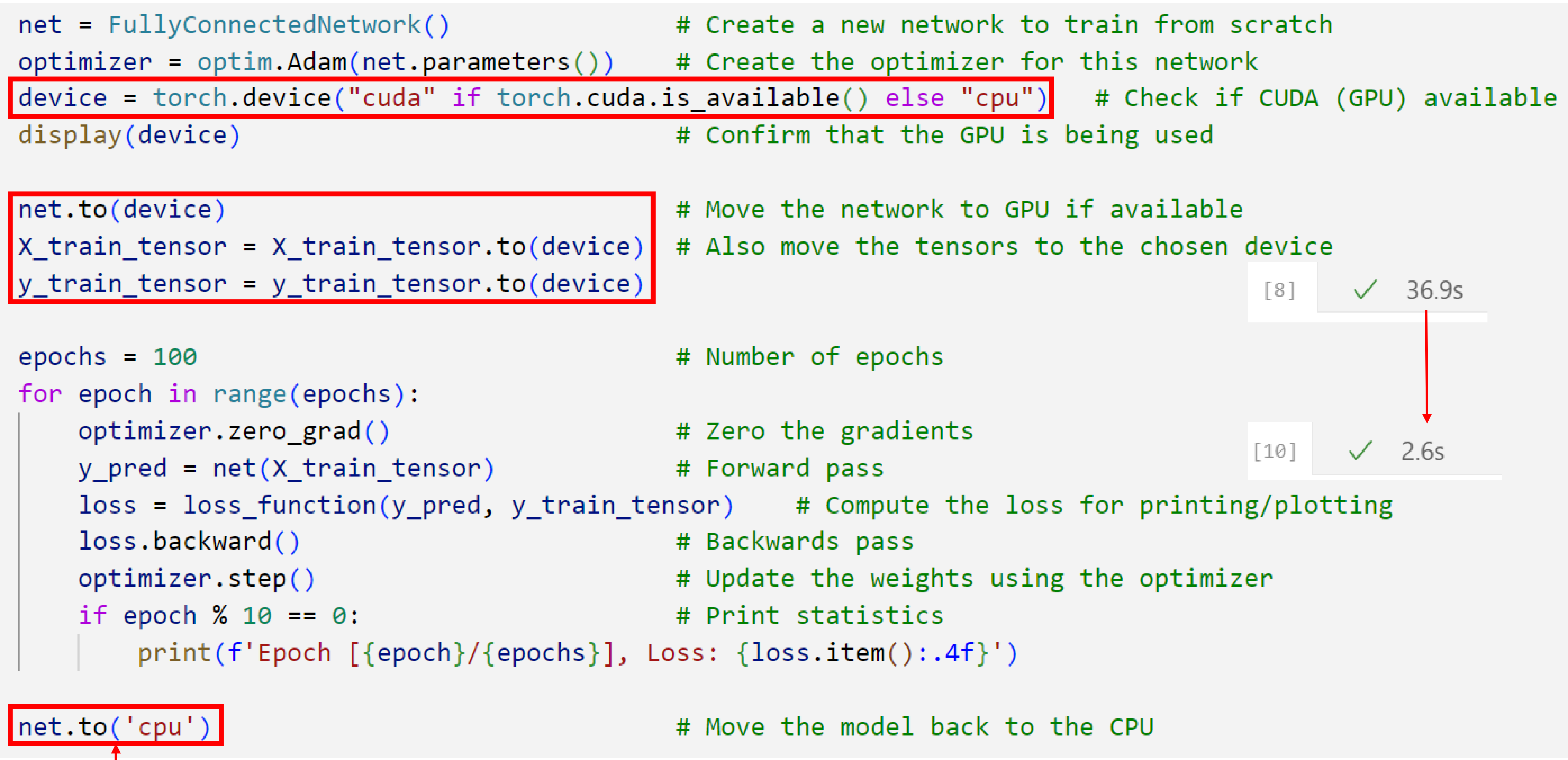
```
optimizer = optim.Adam(net.parameters())
```



```
epochs = 100                    # The number of epochs to run
for epoch in range(epochs):
    # Zero the gradients
    optimizer.zero_grad()

    # Forward pass
    y_pred = net(X_train_tensor)

    # Compute loss
    loss = loss_function(y_pred, y_train_tensor)

    # Backward pass and optimize
    loss.backward()
    optimizer.step()

    # Print statistics
    if epoch % 10 == 0:
        print(f'Epoch [{epoch}/{epochs}], Loss: {loss.item():.4f}')
```

Runtime

- My work desktop has an Intel i9-9900k with 16 cores (CPU).
- It also has an RTX 2070 GPU
    - This has 2304 cores! (An RTX 4090 has 18,432 CUDA cores and 512 special “Tensor” cores)
- These GPU cores are limited in comparison to CPU cores.
    - No branch prediction
    - Limited cache
    - Shorter pipeline (typically)
        - Slower clock (1.605 GHz vs 5 MHz)
    - Not designed for parallel processing (many processes running at once)
- Designed to perform many simple operations like dot products efficiently and in parallel
    - These operations are useful for displaying graphics (e.g., applying simple functions to each pixel on the screen between every frame, changing things like lighting)
    - They are also useful for ML! Running an ANN means computing a *lot* of dot products (and some non-linearities).



```python
net = FullyConnectedNetwork()                # Create a new network to train from scratch
optimizer = optim.Adam(net.parameters())     # Create the optimizer for this network
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")   # Check if CUDA (GPU) available
display(device)                              # Confirm that the GPU is being used

net.to(device)                               # Move the network to GPU if available
X_train_tensor = X_train_tensor.to(device)   # Also move the tensors to the chosen device
y_train_tensor = y_train_tensor.to(device)

epochs = 100                                 # Number of epochs
for epoch in range(epochs):
    optimizer.zero_grad()                    # Zero the gradients
    y_pred = net(X_train_tensor)             # Forward pass
    loss = loss_function(y_pred, y_train_tensor)    # Compute the loss for printing/plotting
    loss.backward()                          # Backwards pass
    optimizer.step()                         # Update the weights using the optimizer
    if epoch % 10 == 0:                      # Print statistics
        print(f'Epoch [{epoch}/{epochs}], Loss: {loss.item():.4f}')

net.to('cpu')                                # Move the model back to the CPU
```

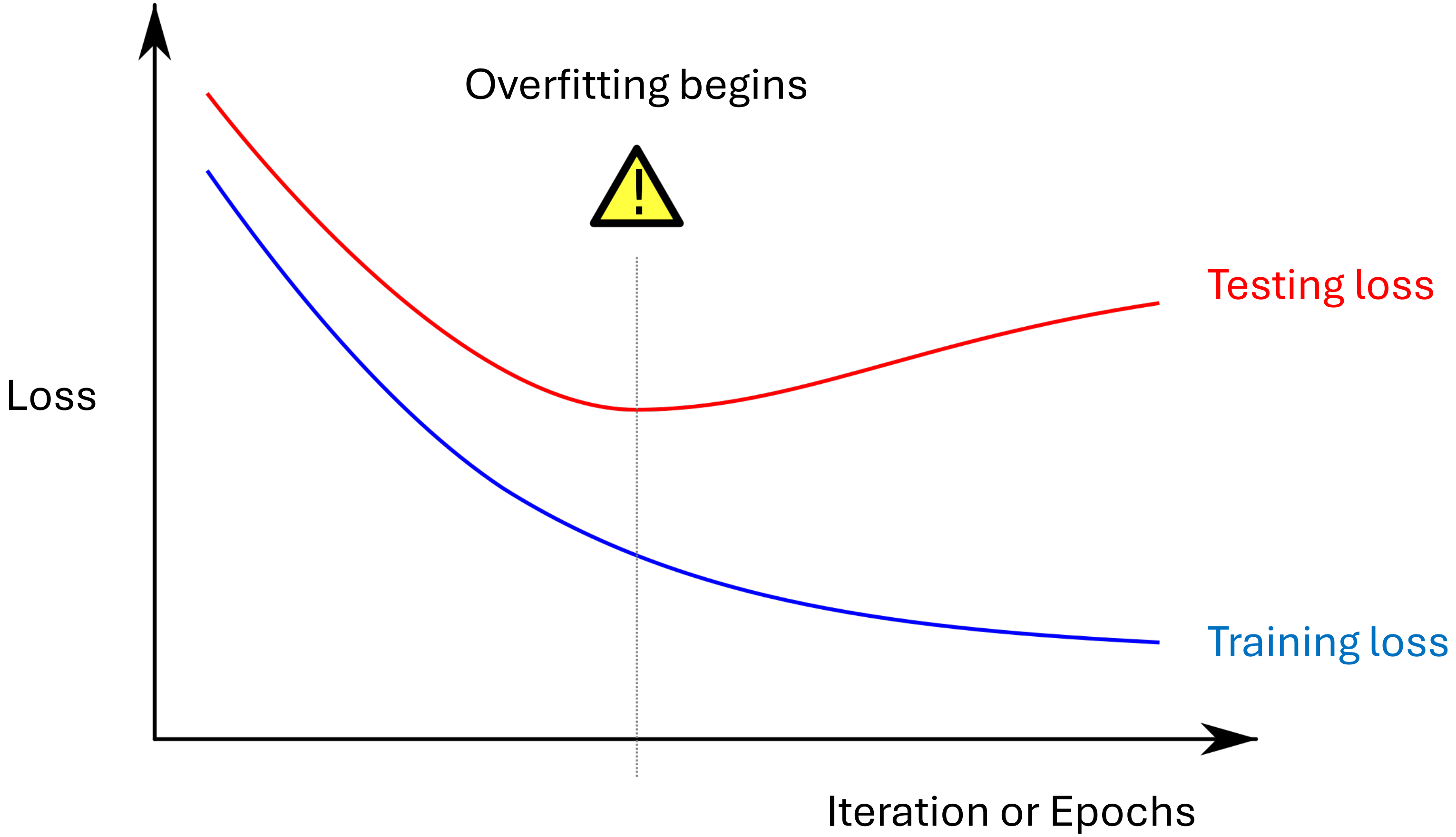
Move the model back to the CPU if you will run it or manipulate it on the CPU (e.g., saving the model/weights to a file). Leave on the GPU if you will only run it on the GPU.

Overfitting

- Recall that the *training* error for *nearest neighbor* (NN) was zero, but the testing error was large.
  - NN essentially "memorized" the training data, and gave good predictions for the training data.
  - The model did not **generalize** to new inputs: it had high errors for points not in the training data.
- When this happens using parametric models, it is called **overfitting**.

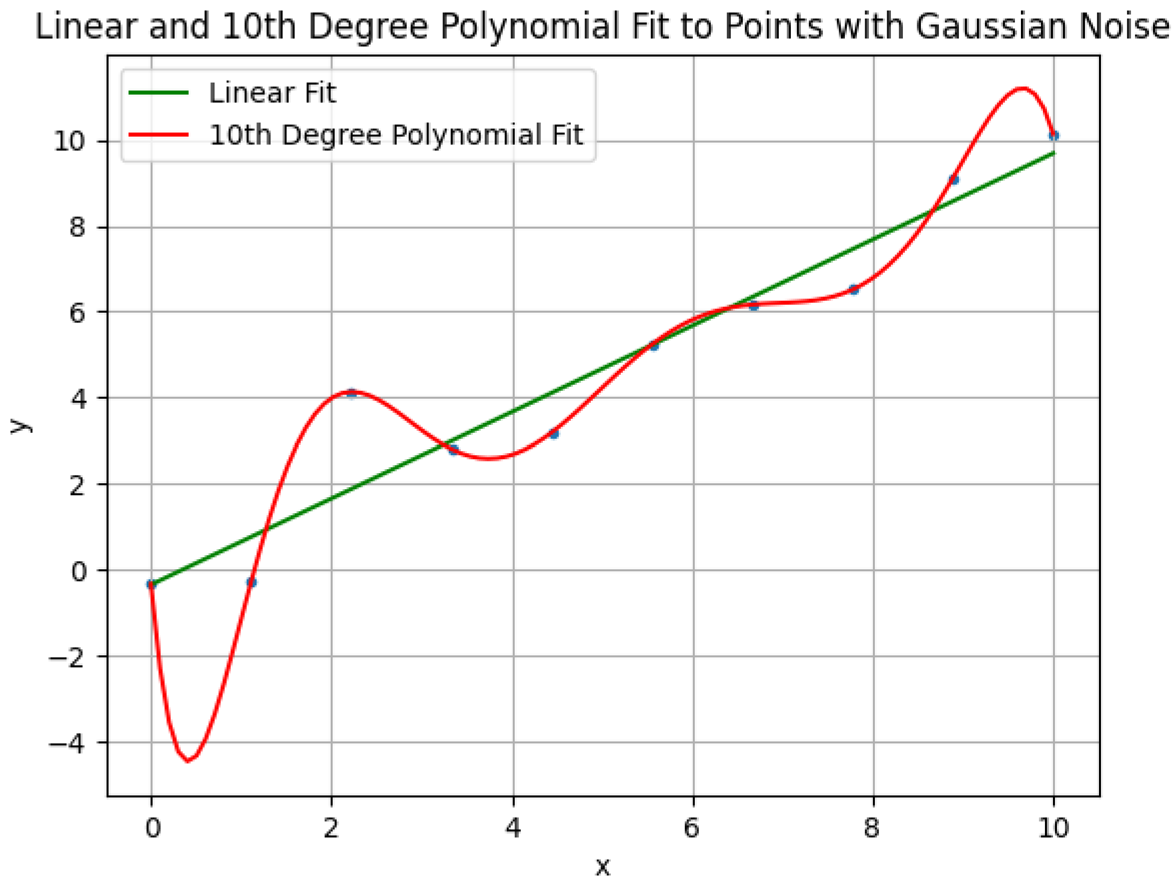
Plotting Training vs Testing Loss (General Case)

Idea: Stop training when the testing loss starts increasing.



Overfitting and Model Complexity/Capacity

- Notice that we can’t overfit this data using a line!
- The **model complexity** or **model capacity** refers to a parametric model’s ability to represent general functions.
    - Models with higher complexity/capacity can represent more functions.
    - Models with higher complexity/capacity are more prone to over-fitting.



Avoiding Over-Fitting (Overview of Strategies)

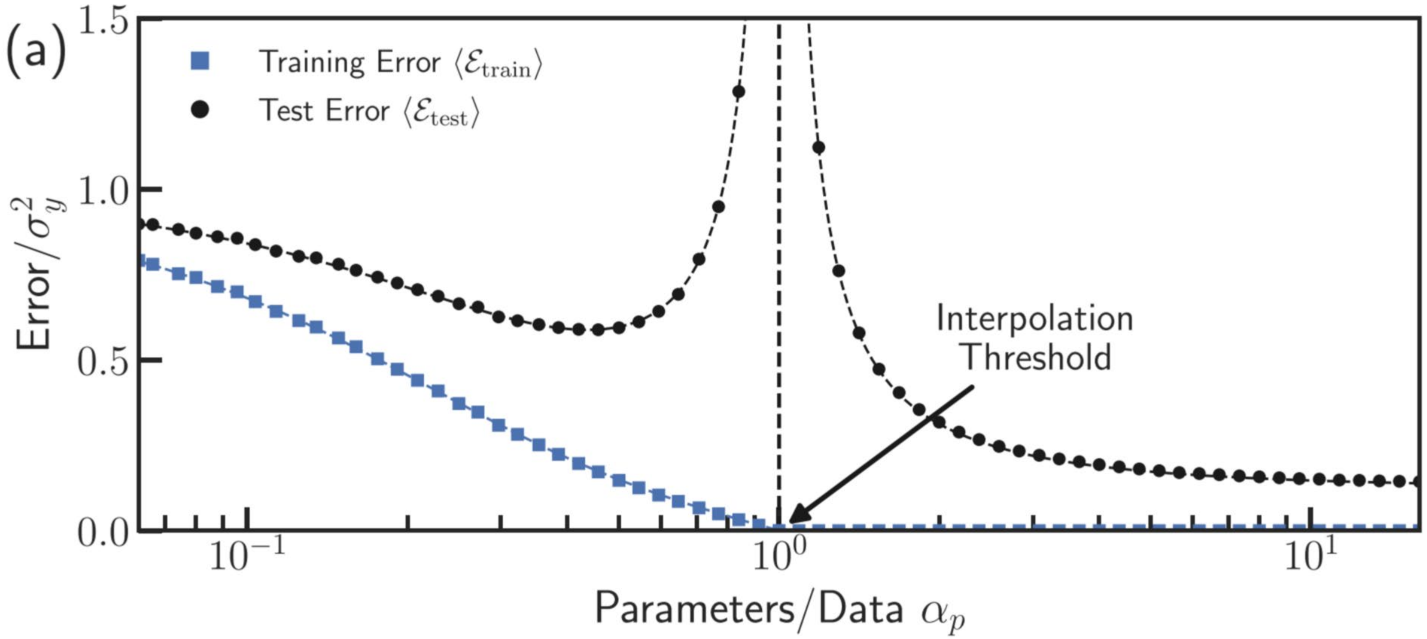
1. Early stopping: Stop training when testing error increases.
   - Typically split data into training, validation, and testing
   - Stop training when the error on the *validation* set begins to increase
   - This ensures that the training process never looks at the testing data
2. Include a "regularization" term in the loss function
   - Complete details are beyond the scope of this course.
   - Regularization terms increase the loss the farther the weight vector is from zero: $L_{\text{new}}(w, D) = L(w, D) + \lambda \|w\|$
     - Often using the L1 norm, $\|w\| = \sum_j |w_j|$ or the L2 norm $\|w\| = \sqrt{\sum_j w_j^2}$.
3. Other strategies (e.g., dropout)
4. Use a large network!

$\|\cdot\|$ denotes a **norm** (a notion of "length")

“Use a large network”: Double Descent

- Large networks seem like they should be particularly prone to overfitting.
- When trained sufficiently on large amounts of data, empirical evidence suggests that deep (large) networks tend not to over-fit!

This phenomenon, called *double descent*, is an active research topic!



Regression → Classification

- Two changes for parametric methods:
  1. Change the parametric model so that it outputs a discrete label as a prediction rather than a number
  2. Select a loss function that is appropriate for classification tasks
- Note: Techniques differ for non-parametric methods
  - E.g., we discussed nearest neighbor (and variants) for classification
  - E.g., there are other custom non-parametric methods for classification like *decision trees*, which are beyond the scope of this course.
- Terminology: Each possible value of the label is called a **class**

Parametric models for classification

- Assume m classes (possible values of the label)
- Change parametric model to have m outputs rather than one.
- **Deterministic**:
  - Class with the highest output is the predicted class.
  - Simple and effective
  - Gradient of the loss function is typically zero, making this impractical for training.
- **Stochastic**:
  - The m outputs are converted to a probability distribution over the classes, and the label is sampled from this distribution.
  - The larger the output, the higher the probability of the class being selected

Stochastic Models: Softmax

- The **softmax** function converts the m outputs to a distribution over the m class values.
- Let $\mathrm{out}_1, \dots, \mathrm{out}_m$ be the model outputs.
- Probabilities cannot be negative, so convert each output to a positive value:

$$\mathrm{out}_1, \dots, \mathrm{out}_m \rightarrow \mathrm{e}^{\mathrm{out}_1}, \dots, e^{\mathrm{out}_m}$$

- A probability distribution must sum to one, so divide each by the sum:

$$\frac{e^{\mathrm{out}_1}}{\sum_{k=1}^{m} e^{\mathrm{out}_k}}, \frac{e^{\mathrm{out}_2}}{\sum_{k=1}^{m} e^{\mathrm{out}_k}}, \dots, \frac{e^{\mathrm{out}_m}}{\sum_{k=1}^{m} e^{\mathrm{out}_k}}.$$

$$\boxed{\Pr(\hat{Y}_i = \hat{y}) = \frac{e^{\mathrm{out}_{\hat{y}}}}{\sum_{k=1}^{m} e^{\mathrm{out}_k}}.}$$

Binary Classification

- Special case where $Y_i \in \{0,1\}$ or $Y_i \in \{-1,1\}$
  - Typically 1 is called the "positive class"
- Parametric models need only have one output, not $m = 2$
  - This output encodes the probability of the positive class.
  - The probability of the negative class is $1 - \Pr(\text{positive class})$.
- The output of the model must be scaled to $[0,1]$.
  - This can be done using the logistic function (sigmoid):

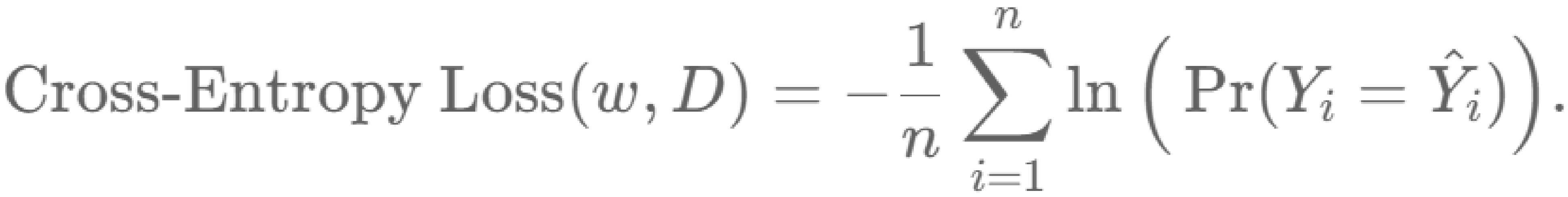
$$\Pr(\hat{Y}_i = 1) = \sigma(\text{out}_1),$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$, and

$$\Pr(\hat{Y}_i = 0) = 1 - \Pr(\hat{Y}_i = 1).$$

Loss Functions for Classification

- There are many loss functions for classification.
  - You can make your own that is tailored to your problem!
- Cross-Entropy Loss (log loss) is the most common.



$$\text{Cross-Entropy Loss}(w, D) = -\frac{1}{n}\sum_{i=1}^{n} \ln\Big(\Pr(Y_i = \hat{Y}_i)\Big).$$

- The $\frac{1}{n}$ is sometimes omitted (it makes no difference).

Logistic Regression

- **Logistic regression** uses the **logistic model** or **logit model**
  - Essentially a linear parametric model for classification

$\sigma(w \cdot \phi(X_i))$

$$\Pr(\hat{Y}_i = 1|X_i) = \frac{1}{1 + e^{-w\cdot\phi(X_i)}}.$$

- Use cross-entropy loss
  - Equivalent to maximizing the “likelihood” of the data given the model.

$$\text{Cross-Entropy Loss}(w, D) = -\frac{1}{n}\sum_{i=1}^{n} \ln\Big(\Pr(Y_i = \hat{Y}_i)\Big).$$

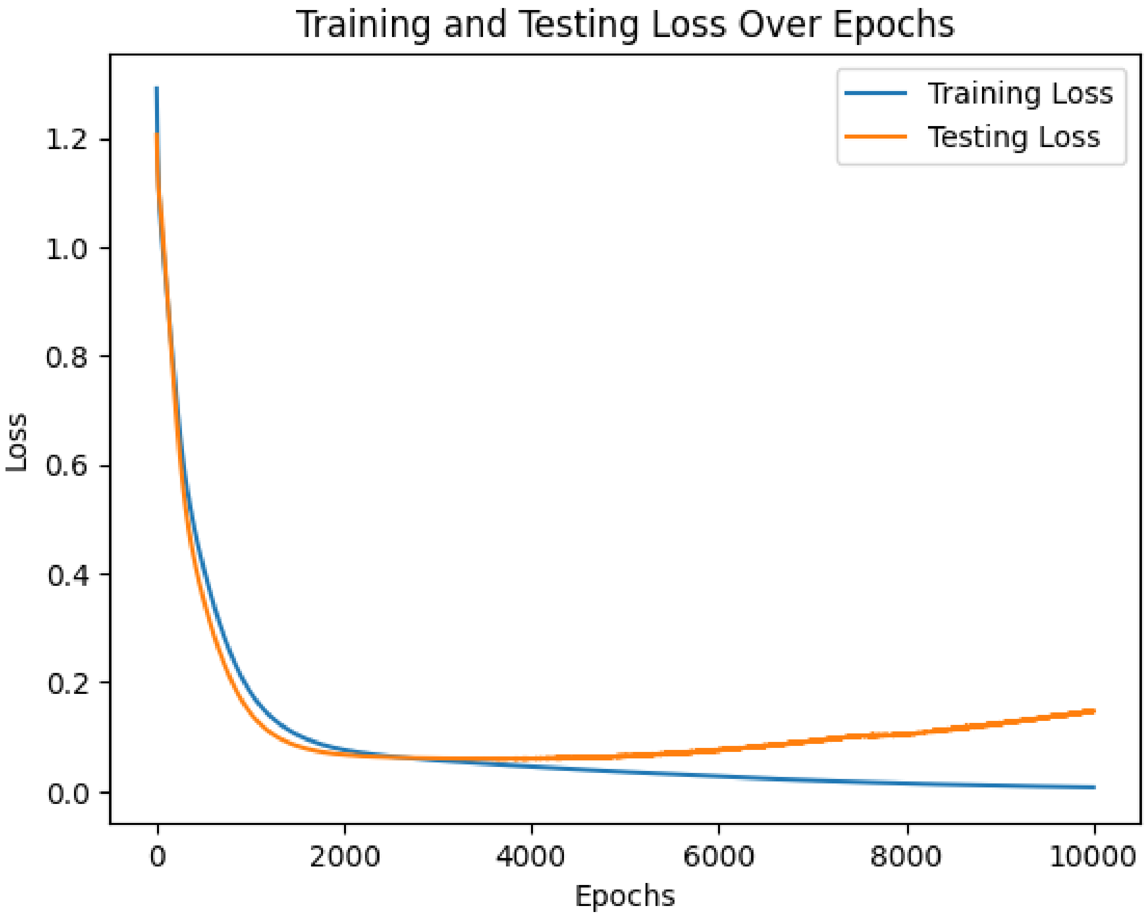
Stochastic → Deterministic Models

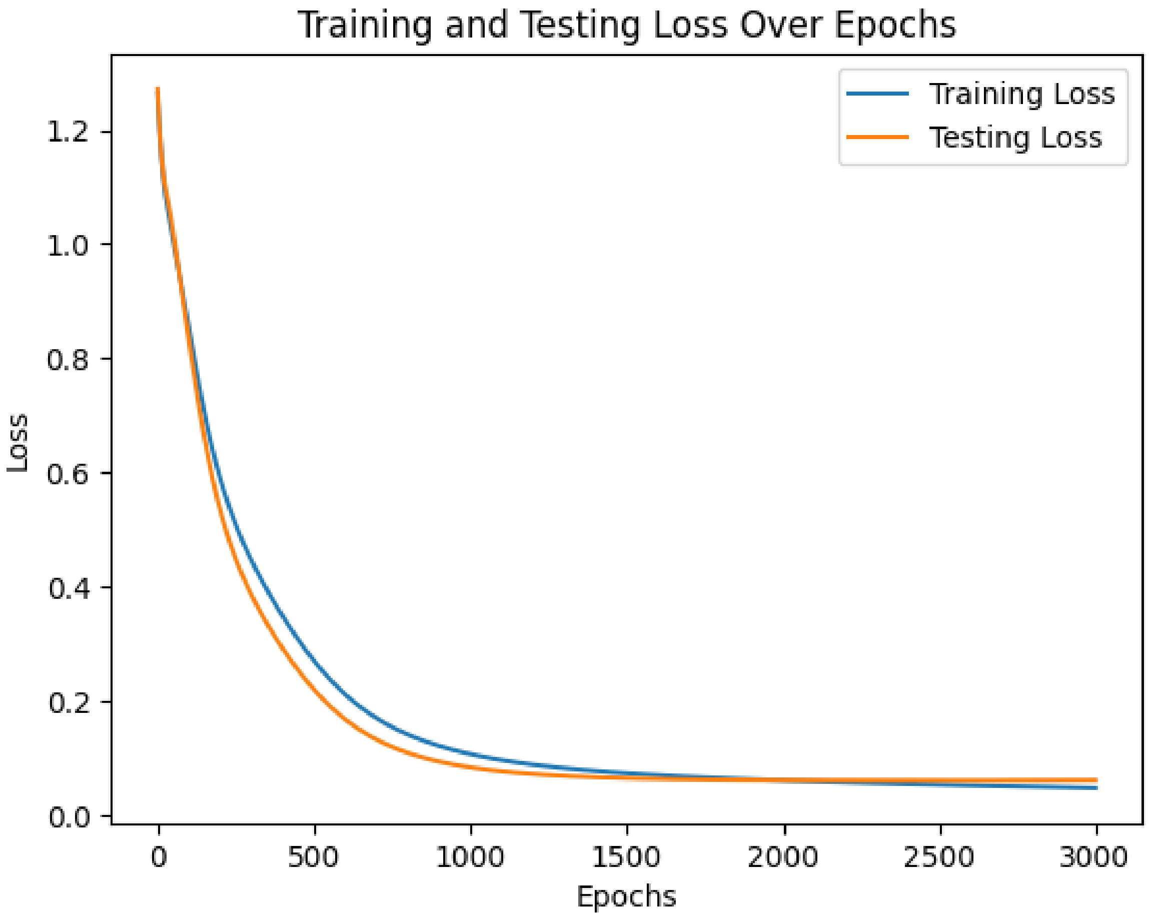
- During training often models are viewed as stochastic (minimizing cross-entropy loss).
- If the model is highly confident of the class for an input, the output for that class will be come large
  - No matter how large it is, the resulting probability of the label will not be 1

$$\Pr(\hat{Y}_i = 1 | X_i) = \frac{1}{1 + e^{-w \cdot \phi(X_i)}}.$$

- To enable models to make deterministic predictions, often models are *evaluated* (and then deployed to make predictions for new data) as deterministic models, even if they are trained as stochastic models.

We saw another example of over-fitting, and used early stopping to prevent it:



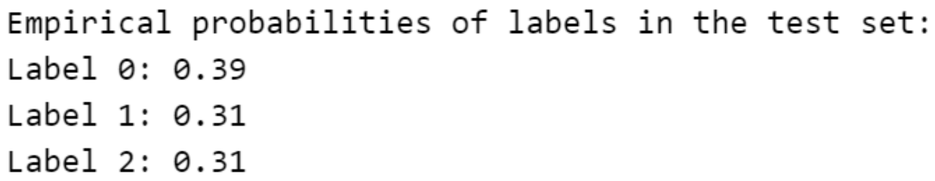


Evaluation Metric: Accuracy

The accuracy is the proportion of correct predictions to the total number of predictions:

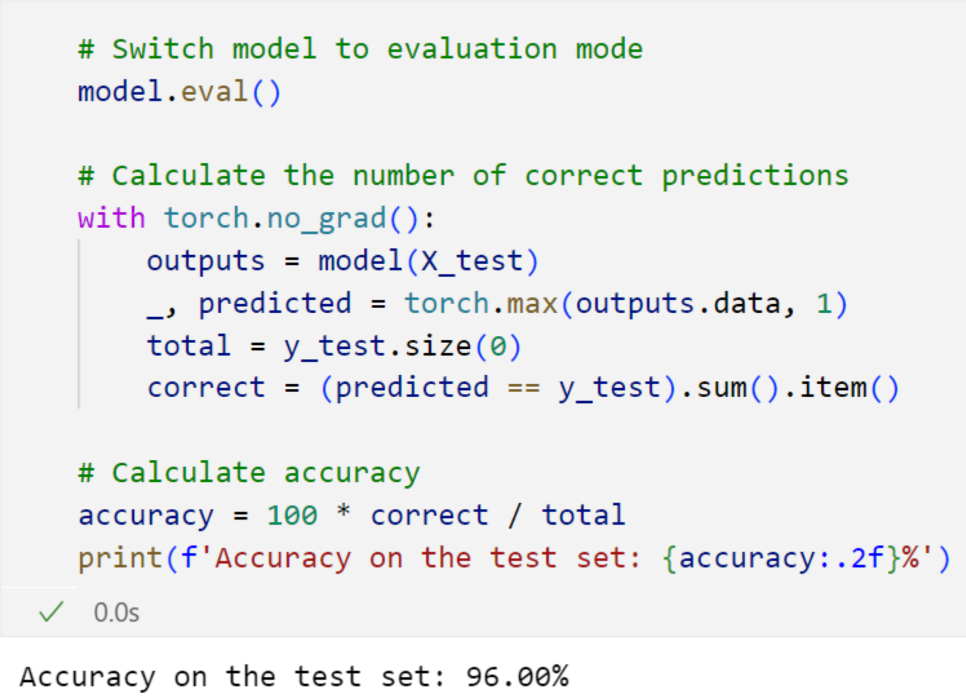
$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions}}.$$

- While relatively simple, accuracy can be misleading if the class distribution is imbalanced.



```
Empirical probabilities of labels in the test set:
Label 0: 0.39
Label 1: 0.31
Label 2: 0.31
```

- In this case, 96% accuracy is decent!



```python
# Switch model to evaluation mode
model.eval()

# Calculate the number of correct predictions
with torch.no_grad():
    outputs = model(X_test)
    _, predicted = torch.max(outputs.data, 1)
    total = y_test.size(0)
    correct = (predicted == y_test).sum().item()

# Calculate accuracy
accuracy = 100 * correct / total
print(f'Accuracy on the test set: {accuracy:.2f}%')
```

```
Accuracy on the test set: 96.00%
```

Evaluation Metric: Confusion Matrix

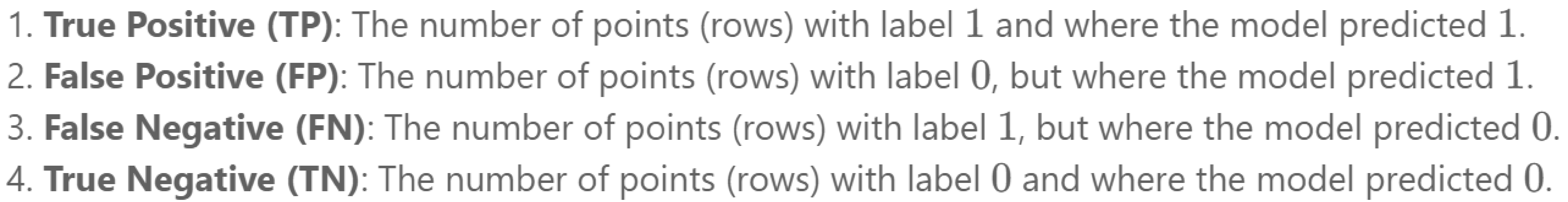
- Accuracy doesn’t provide information about what kinds of errors are common
  - Which classes are often confused?
- The **confusion matrix** provides this information. It is a matrix with one row per class and one column per class
  - The $(i, j)^{\mathrm{th}}$ entry holds the probability that a row with actual class i is classified as class j.
  - In some cases the matrix reports the number of errors of each type, rather than the estimated probability.

Evaluation Metric: Confusion Matrix

- Accuracy doesn’t provide information about what kinds of errors are common
  - Which classes are often confused?
- The **confusion matrix** provides this information. It is a matrix with one row per class and one column per class
  - The $(i, j)^{\mathrm{th}}$ entry holds the probability that a row with actual class i is classified as class j.
  - In some cases the matrix reports the number of errors of each type, rather than the estimated probability.

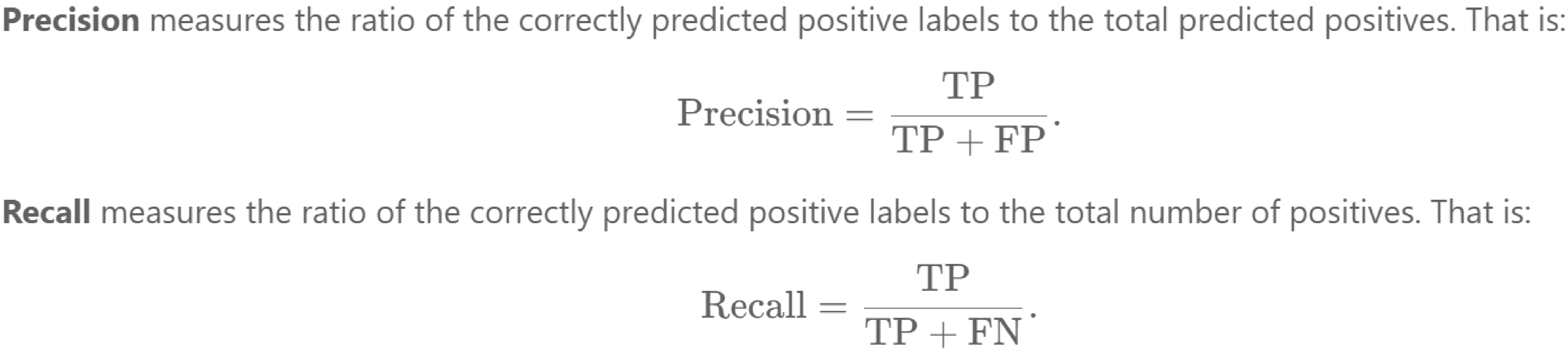
Evaluation Metric: Precision, Recall, and F1 Score

- For *binary classification* tasks, statistics like **precision, recall,** and the **F1 score** are often used to evaluate models.
    - Note: These are often used even when the loss function used in training measures something else, like cross-entropy loss.
- These metrics are expressed in terms of the following statistics:



1. **True Positive (TP)**: The number of points (rows) with label 1 and where the model predicted 1.
2. **False Positive (FP)**: The number of points (rows) with label 0, but where the model predicted 1.
3. **False Negative (FN)**: The number of points (rows) with label 1, but where the model predicted 0.
4. **True Negative (TN)**: The number of points (rows) with label 0 and where the model predicted 0.

Deterministic Classifiers



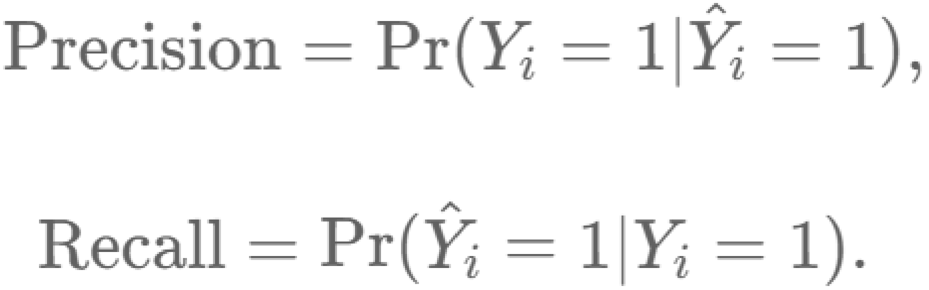
Precision measures the ratio of the correctly predicted positive labels to the total predicted positives. That is:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Recall measures the ratio of the correctly predicted positive labels to the total number of positives. That is:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Stochastic Classifiers



$$\text{Precision} = \Pr(Y_i = 1 | \hat{Y}_i = 1),$$

$$\text{Recall} = \Pr(\hat{Y}_i = 1 | Y_i = 1).$$

F1 Score

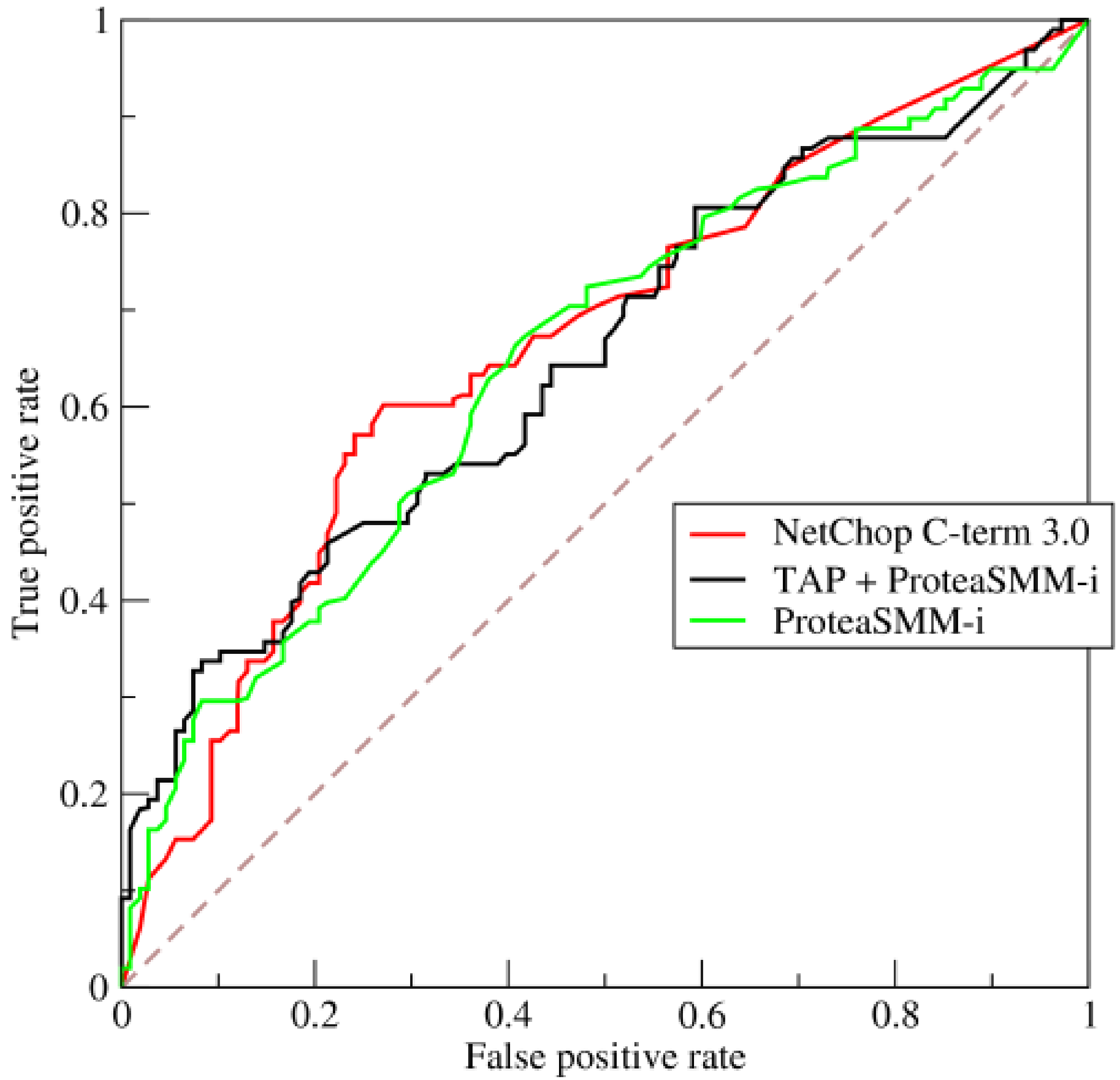
- The F_1 score (often written “F1 score”) combines precision and recall:

$$\mathrm{F}_1 \text{ Score} = 2\,\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

- This is the *harmonic mean* of the precision and recall
    - Places more weight on low values relative to the arithmetic mean
- F1 score ranges from 0 to 1, where 1 denotes perfect precision and recall, and 0 means that either precision or recall is zero.

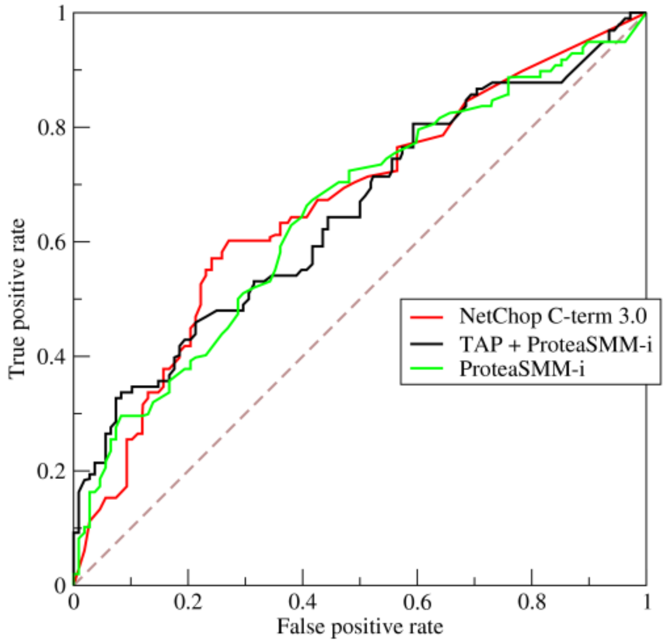
Example ROC Curve

- Curves closer to the top left corner correspond to better models.
- A classifier that ignores the inputs and outputs a uniform random number in $[0,1]$ results in a diagonal line from $(0,0)$ to $(1,1)$



Evaluation Metric: Area Under the ROC Curve (AUC)

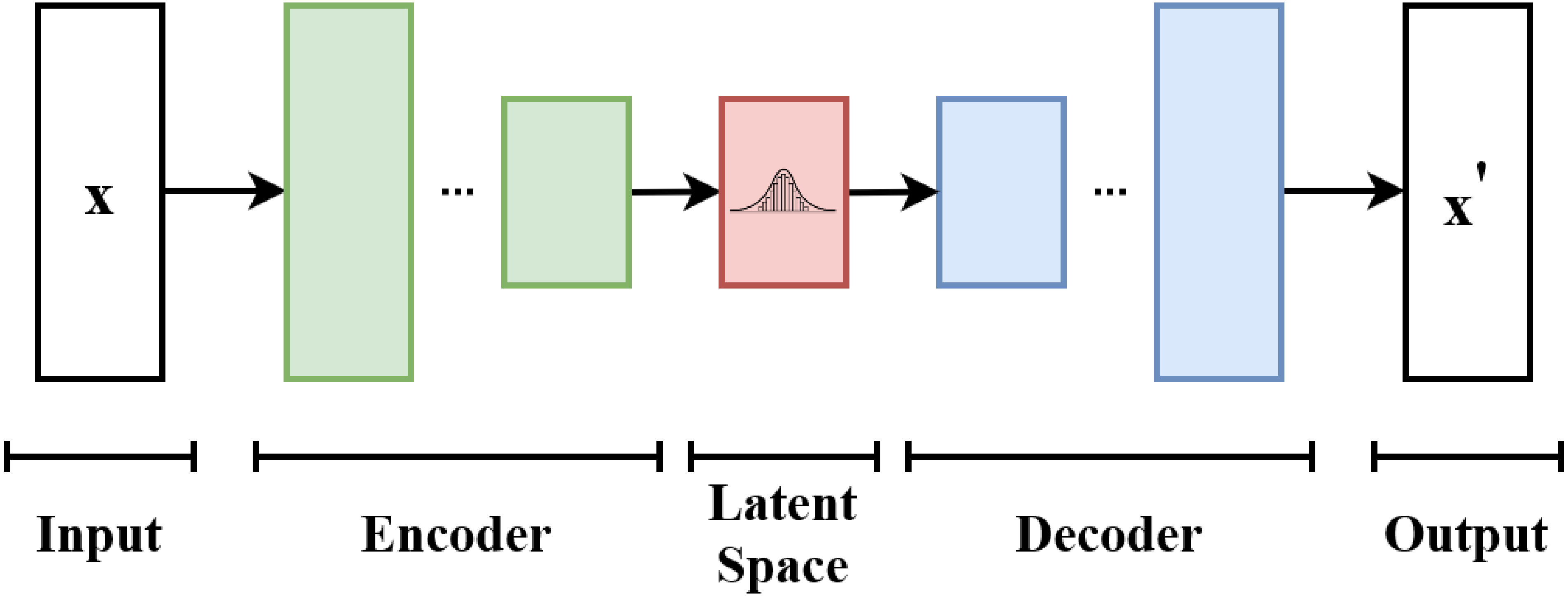
- The AUC summarizes the ROC curve with a single number: The area under the ROC curve.
- The best possible value is 1.
- A pessimal model (one that always gets the prediction wrong) would have an AUC of zero.
- The random classifier achieves an AUC of 0.5



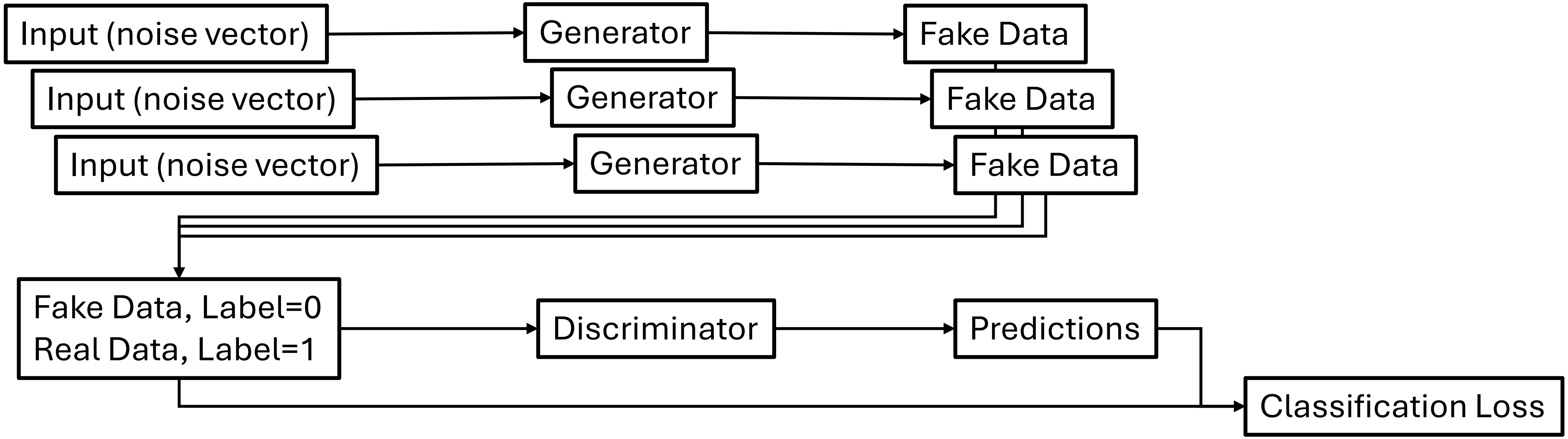
Generative AI

- **Generative AI** methods create new content like text, images, music, or other data, often mimicking some aspects of human creativity.
- Generative AI is often (not always!) a form of unsupervised learning (learning from data with no labels).
    - When presented with a data set $D = (X_i)_{i=1}^{n}$, the agent's goal is to create new data points that are indistinguishable from the data in D.
- Two core methods in generative AI are **variational autoencoders** (VAEs) and **generative adversarial networks** (GANs).

Variational Autoencoders (VAEs)



Generative Adversarial Networks (GANs)



Conditioning on Text

- VAEs and GANs can be conditioned on text.
- In a VAE, the text is first converted into its own embedding (numerical vector representation)
- The text (represented as a vector of numbers) is then appended to the input to the decoder.
  - The encoder does not see the text – it just learns a representation for the image.
  - The decoder is given the latent representation of the image *and* the text description.
- To be effective, the distribution of the latent representation *conditioned on the text* must still be normally distributed.
  - Otherwise, when generating a new image, the latent representation of the image that is sampled may not be compatible with the provided text query.
  - Mechanisms for ensuring this are beyond the scope of this course.

Conditioning on Text

- To condition a GAN on text, the generator receives both the noise and text embedding as input.
  - Its goal is to generate an image that corresponds to the text embedding that is indistinguishable from images and their corresponding text embeddings in the training data.
- The discriminator also takes the text embedding into account.
  - Its goal is to determine whether the image provided for the text embedding corresponds to an image from the real data set or the fake data set.
- **Note:** Both training VAEs and GANs that can be conditioned on text requires training data containing both images and corresponding text descriptions!

Large Language Models (LLMs)

- Large parametric models applied to text (or audio) generation.
- **Input**: A sequence of words, split into **tokens**
  - A token is a sequence of letters/punctuation
  - Often a token is a word or a part of a word
- **Output**: The next token
- **Training**: This is a standard classification problem!
  - Generate input-output pairs from human-written text

Foundation Models

- Modern parametric ML models are expensive to train
- Instead of everyone training new models, large models can be trained once and shared.
- These are called **foundation models**.
- Examples: GPT (OpenAI), BERT (Google), Llama (Meta), and many others.
  - Some can be found at https://huggingface.co/

Finetuning Models

- When using foundation models, often there is a need to change the model in some way.
  - Provide it with additional training data on a specific topic
  - Change the tone of its responses
  - Change it so that responses are more conversational
  - Change it so that it excels at summarizing reviews
  - …
- When a foundation model is further trained (often using a different data set and loss function!), it is called **fine-tuning**.

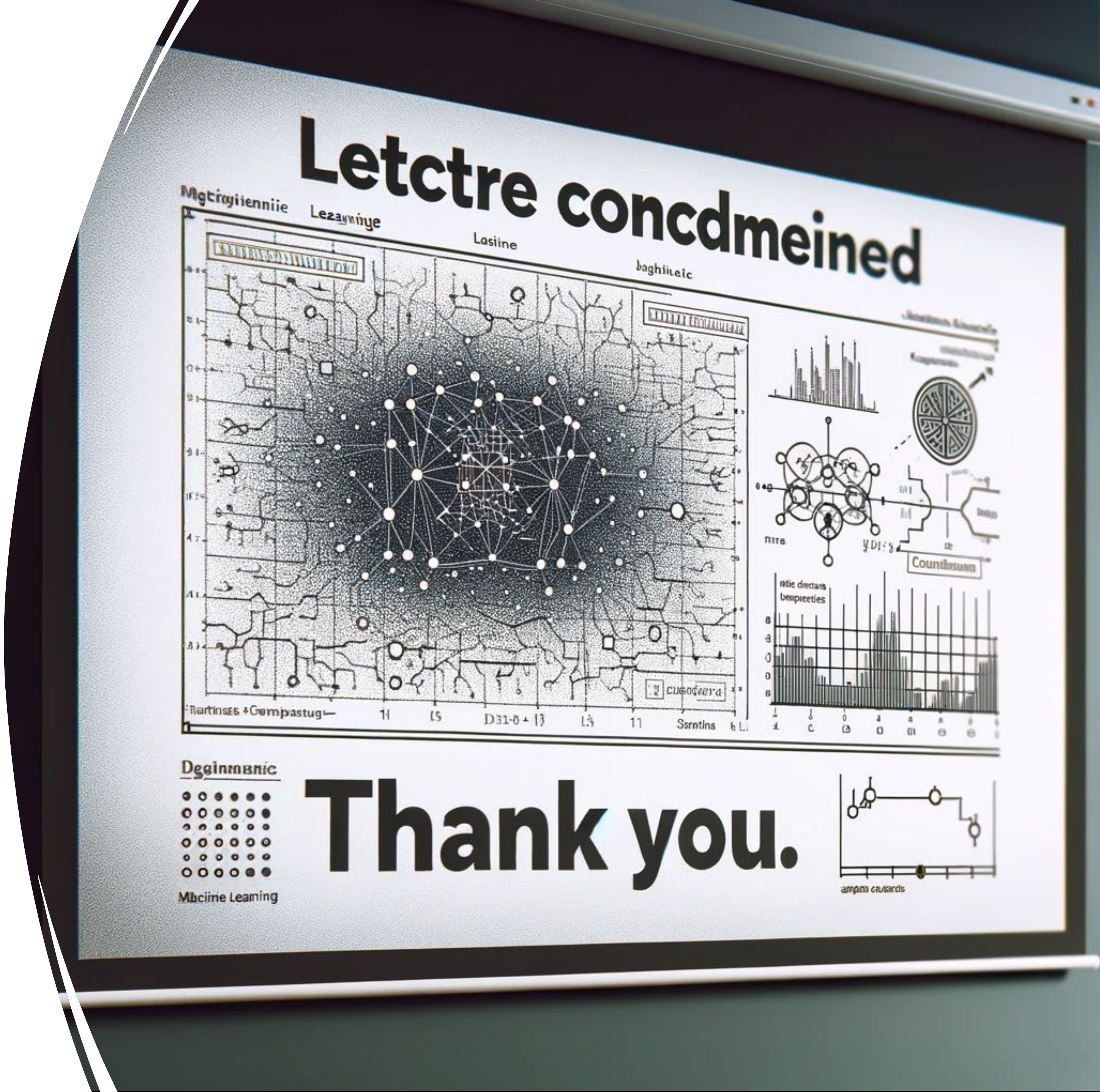
Finetuning Models Efficiently

- Even finetuning a large model can be infeasible without significant hardware and funding.
- One area of research involves finding more efficient ways to finetune models.
- Example: **Low Rank Adaptation** (LoRA)
    - Focusses on changing weights in a section of the network (attention and feed-forward parts of a transformer).
    - Uses low-rank matrices to represent the change to the weights.
        - This is a way of using a small number of weights to tune a larger number of weights
        - If there are $m \times n$ weights W, we tune two matrices A and B of sizes $m \times k$ and $k \times n$, where k is relatively small. The change to weights W is then AB.

Executing Models Efficiently

- Running (not just training!) large parametric models can also be expensive.
- Another area of research focusses on making the execution of large models more efficient
- Examples:
  - **Model pruning**: Finding unimportant weights and parameters that can be removed.
  - **Quantization**: Reducing weights from 32 bits to 8 bits.
  - **Knowledge Distillation**: Train a smaller model to mimic the outputs of a larger pre-trained model.

End


Letctre concdmeined
Mgtrqyiennie
Lezaynnye
Lasiine
baghiteic
Countlsuuno
Dgginmbnic
Thank you.
Mbcime Leaming
amapm cauards