ML lifecycle II

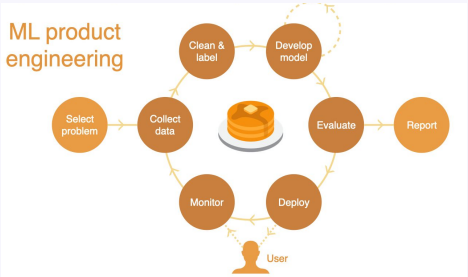
Big data management

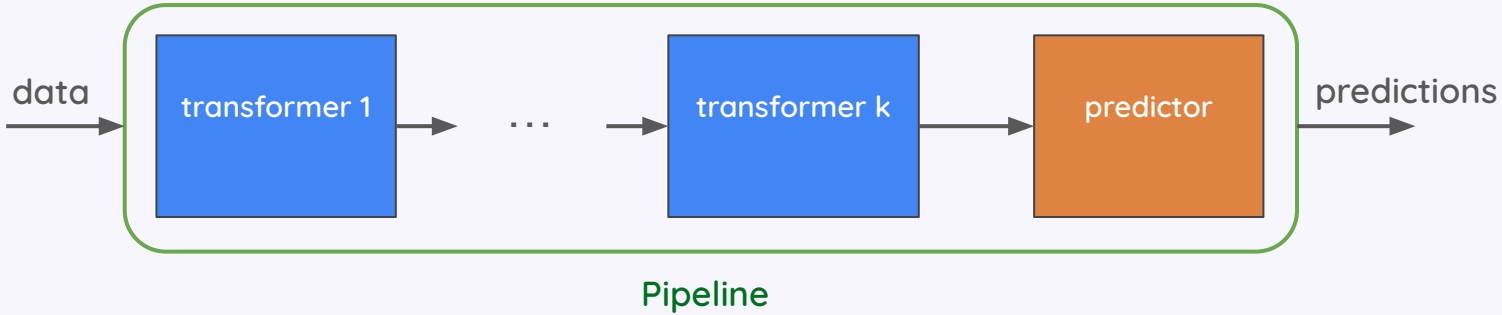
Last time

Building machine learning models = iterative process

Data exploration, cleaning, and preprocessing

Pipelines - ensure transformations are done in order, consistently, without data leakage





Purposes of using pipelines

Convenience and encapsulation

You only have to call fit and predict once on your data to fit a whole sequence of estimators.

Joint parameter selection

You can grid search over parameters of all estimators in the pipeline at once.

Safety

Pipelines help avoid leaking statistics from your test data into the trained model in cross-validation, by ensuring that the same samples are used to train the transformers and predictors.

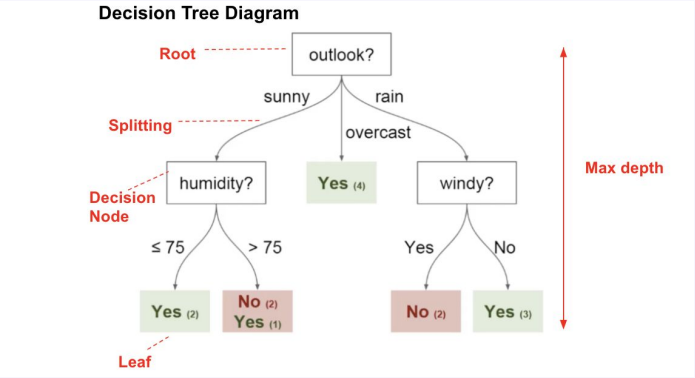
ML lifecycle II - Training models on big data

Decision trees

Decision trees: classification and regression

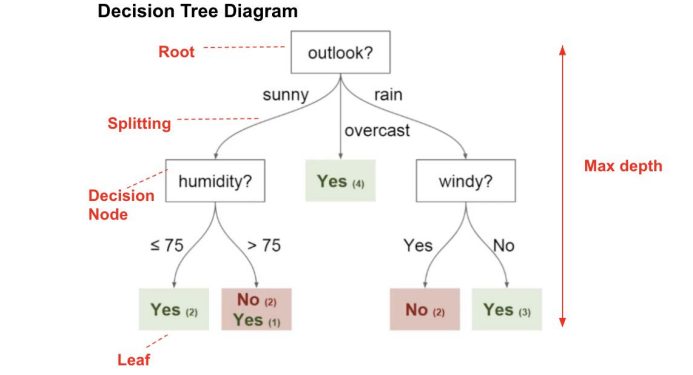
Versatile, easy to understand and visualize (white box), powerful

Day	Outlook	Temperature	Humidity	Wind	Play Golf
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



Building decision trees

Day	Outlook	Temperature	Humidity	Wind	Play Golf
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes



To find the order of building trees, compute either:

- gini index
- entropy

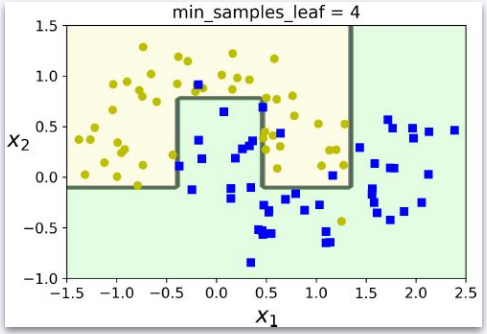
Each is 0 when all instances belong to the same class.

Intuitively, they show how informative a feature is.

Further info on basics of tree-based models

Read the chapter on Decision trees from the Hands-on ML book to find out more:

- how they are used for regression
- how gini index and entropy are calculated
- limitations



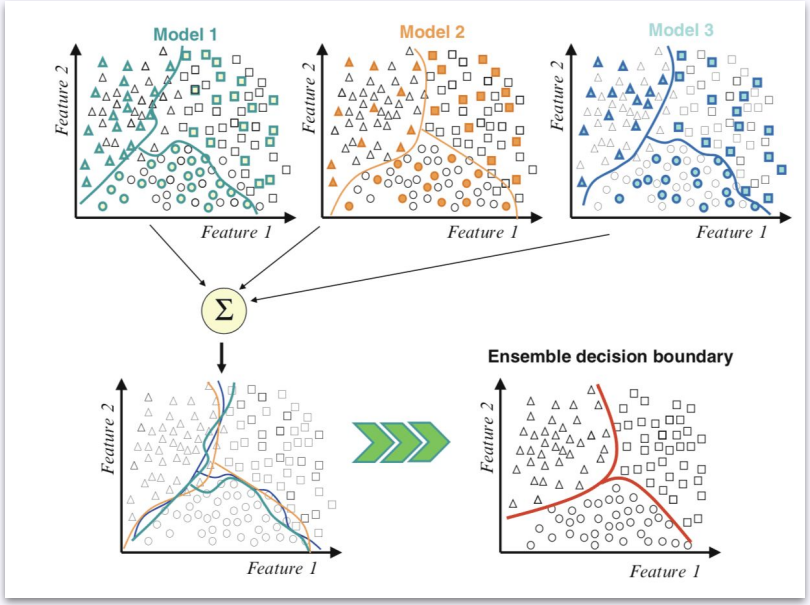
Also check this in-depth tutorial on using Decision trees in sklearn.

Decision trees limitations

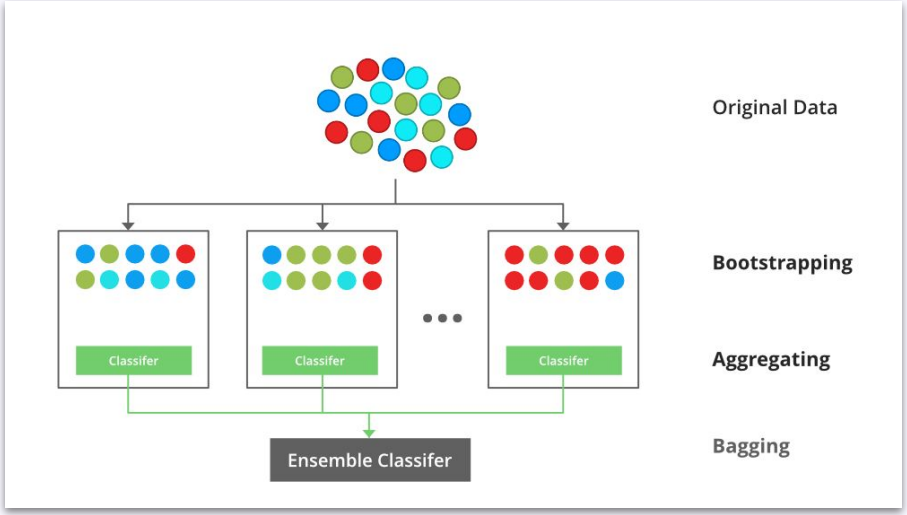
- Over-complex trees => **overfitting**
  - Solutions: pruning, minimum number of samples required at a leaf node, maximum depth of the tree
- If small variations in the data - > completely different tree being generated => **unstable**
  - Solution: ensembles
- Heuristic algorithms (greedy) => **not globally optimal** decision tree (NP-complete)
  - Solution: problem mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement
- If some classes dominate => **biased** trees
  - Solution: balance the dataset prior to fitting with the decision tree

Ensemble models in ML

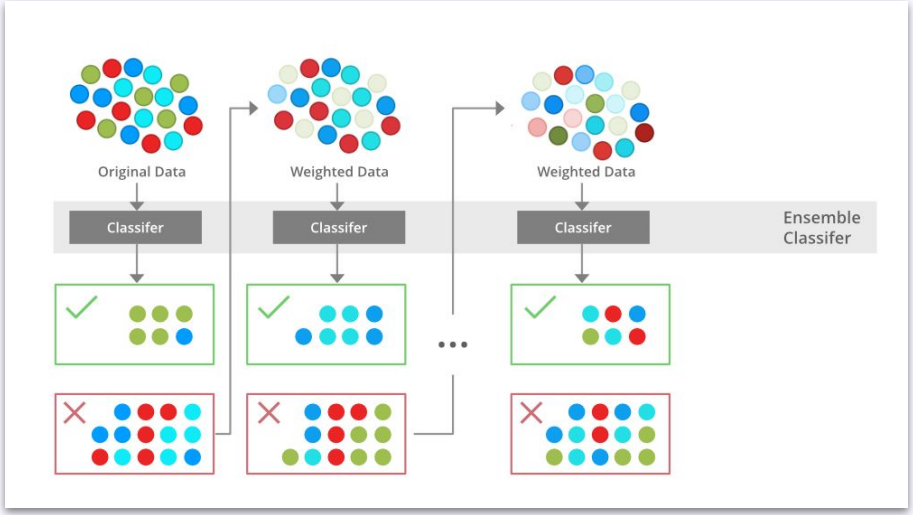
Ensemble learning - using multiple learning algorithms to obtain better predictive performance than could be obtained from any of the individual models.



Ensemble models in ML

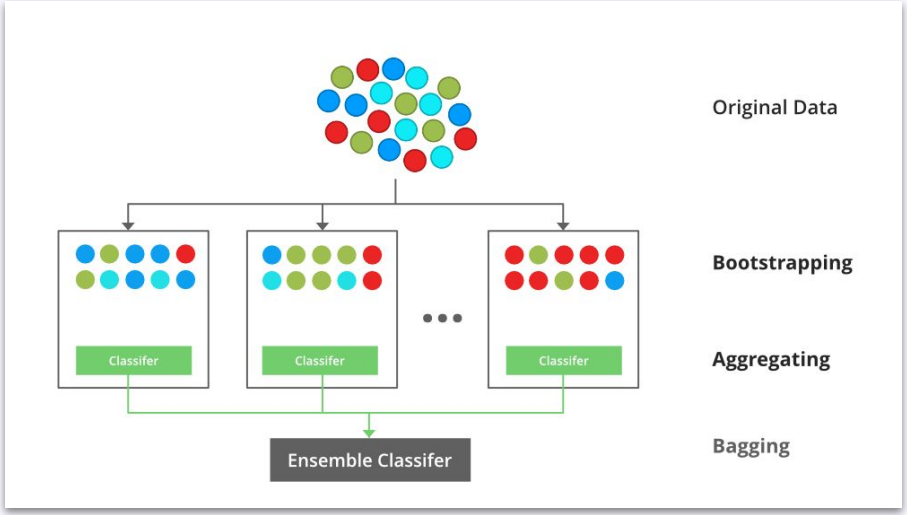


Bagging



Boosting

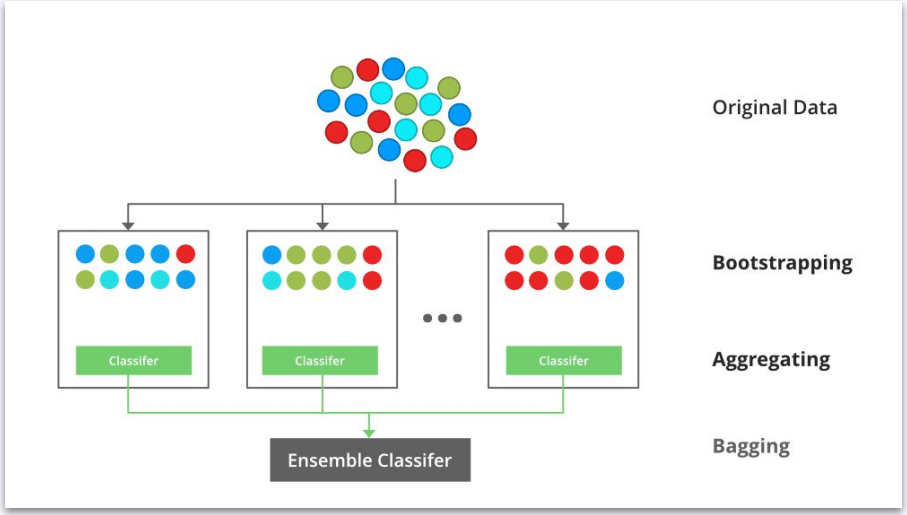
Ensemble models in ML



Bagging - Bootstrap Aggregation

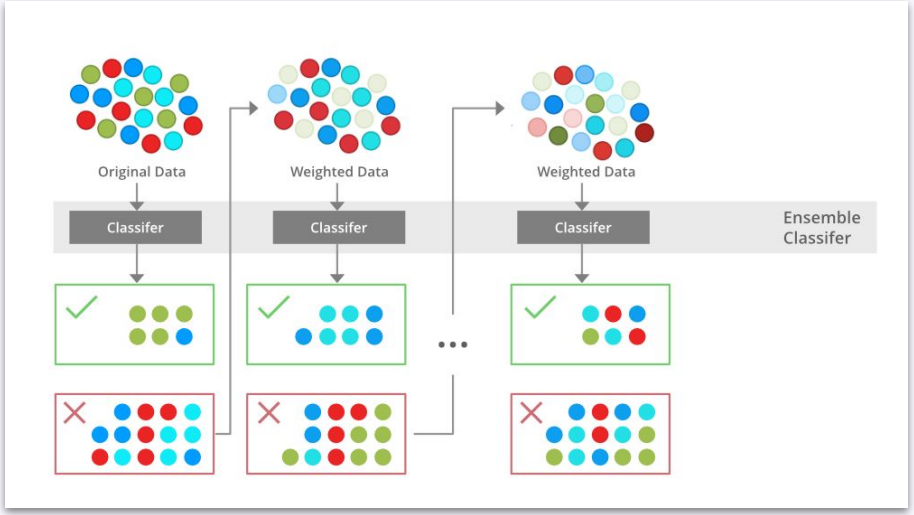
- create several subsets of data from training sample chosen randomly with replacement
- each collection of subset data is used to train a model
- take average of all the predictions
- random forest: sample features

Ensemble models in ML



Bagging - Bootstrap Aggregation

- create several subsets of data from training sample chosen randomly with replacement
- each collection of subset data is used to train a model
- take average of all the predictions
- random forest: sample features

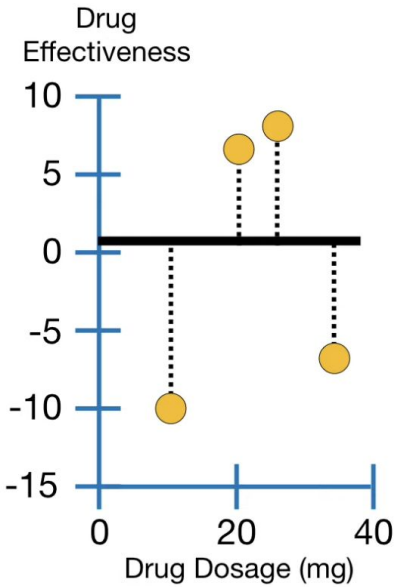


Boosting

- models are learnt sequentially, each one improving on the previous error
- improve performance on instances where the previous model performed poorly
- final prediction is a weighted sum of all of the predictions

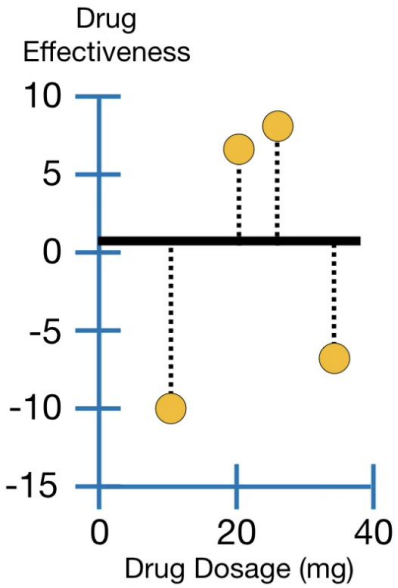
Extreme Gradient Boosting = a framework for distributed ML

XGBoost example - predicting drug effectiveness



initial prediction:
effectiveness = 0.5

XGBoost example - predicting drug effectiveness

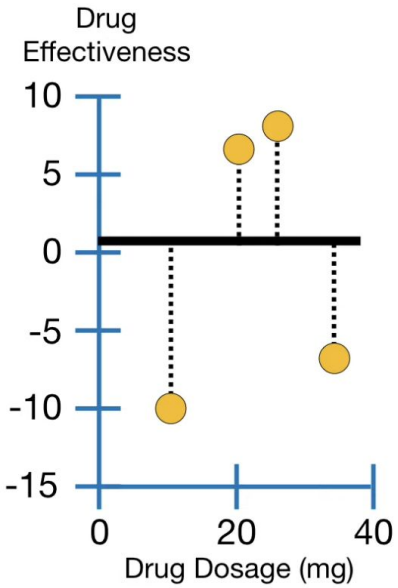


-10.5, 6.5, 7.5, -7.5

start building the tree with a single leaf that contains the residuals

initial prediction:
effectiveness = 0.5

XGBoost example - predicting drug effectiveness



initial prediction:
effectiveness = 0.5

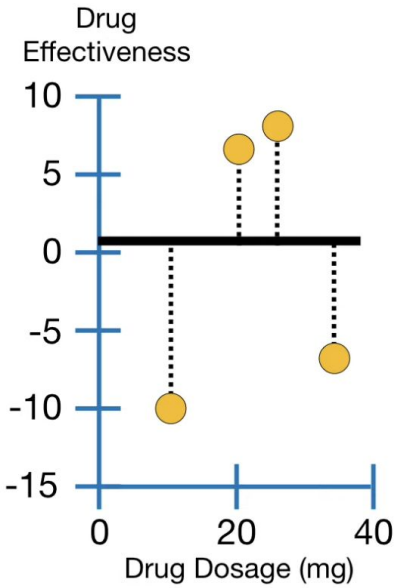
-10.5, 6.5, 7.5, -7.5

start building the tree with a single leaf that contains the residuals

$$\text{Similarity Score} = \frac{\text{Sum of Residuals, Squared}}{\text{Number of Residuals} + \lambda}$$

(lambda = regularization parameter; ignore for now)

XGBoost example - predicting drug effectiveness



initial prediction:
effectiveness = 0.5

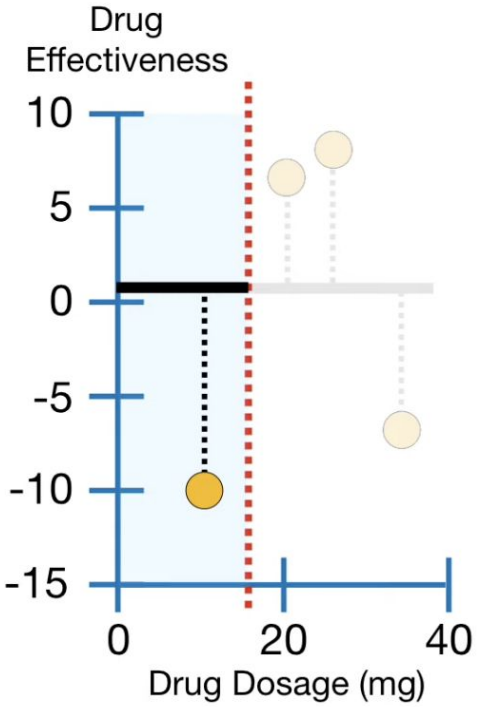
-10.5, 6.5, 7.5, -7.5 Similarity = 4

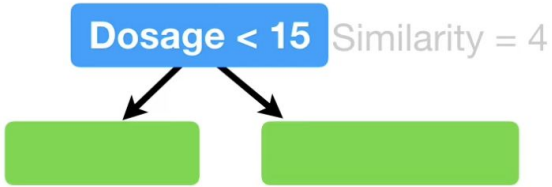
start building the tree with a single leaf that contains the residuals

$$\text{Similarity Score} = \frac{\text{Sum of Residuals, Squared}}{\text{Number of Residuals} + \lambda}$$

(lambda = regularization parameter; ignore for now)

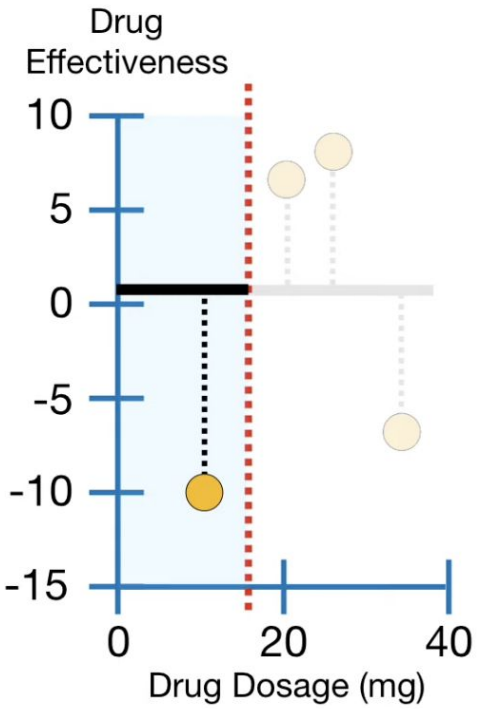
XGBoost example - predicting drug effectiveness

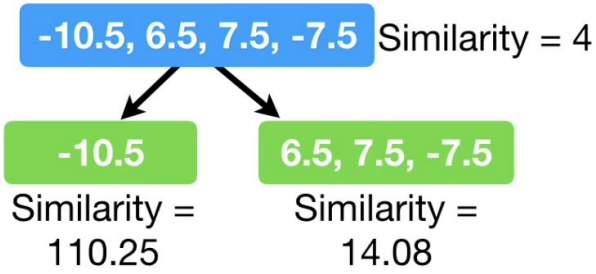




Can we do better? Try clustering residuals

XGBoost example - predicting drug effectiveness





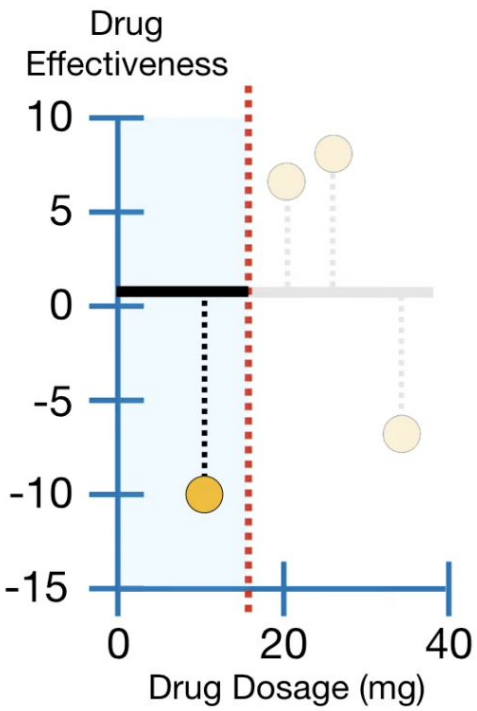
$$\text{Gain} = \text{Left}_{\text{Similarity}} + \text{Right}_{\text{Similarity}} - \text{Root}_{\text{Similarity}}$$

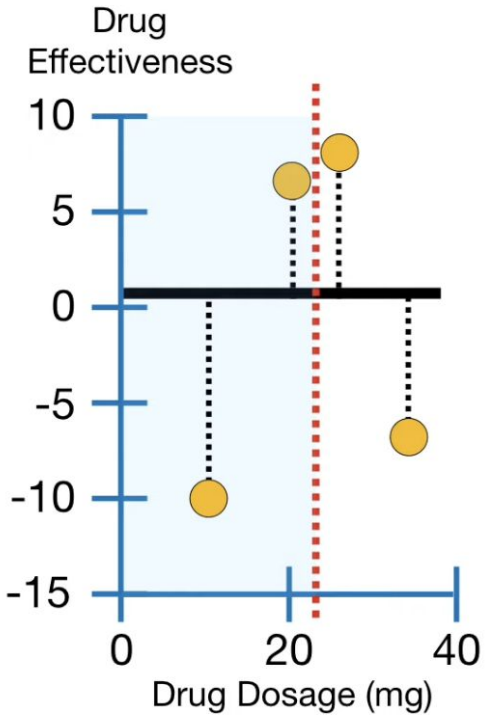
$$\text{Gain} = 110.25 + 14.08 - 4 = 120.33$$

Repeat the same process for different thresholds

Can we do better? Try clustering residuals

XGBoost example - predicting drug effectiveness



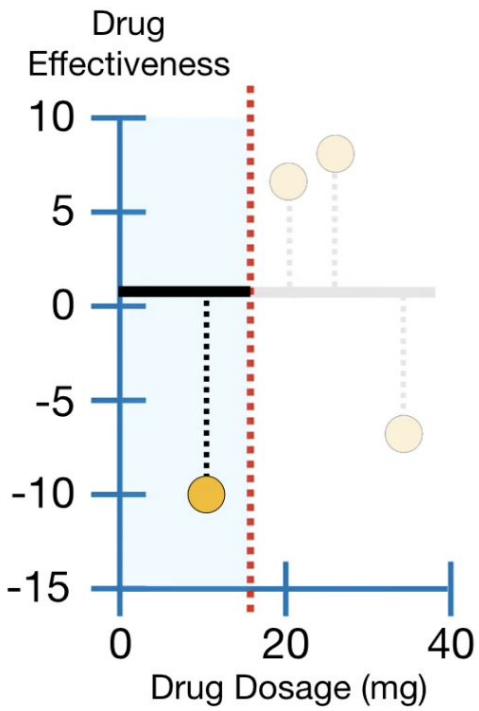


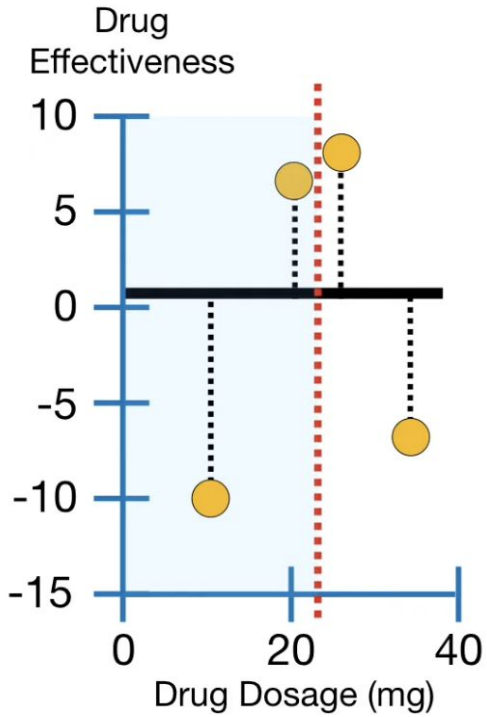
Consider a different threshold - shift it to the next value.

$$Gain = Left_{Similarity} + Right_{Similarity} - Root_{Similarity}$$

$$Gain = 8 + 0 - 4 = 4$$

XGBoost example - predicting drug effectiveness





Consider a different threshold - shift it to the next value.

$$Gain = Left_{Similarity} + Right_{Similarity} - Root_{Similarity}$$

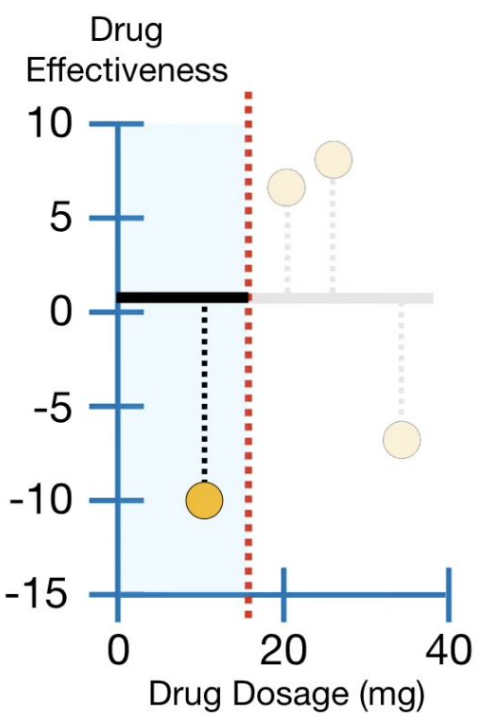
$$Gain = 8 + 0 - 4 = 4$$

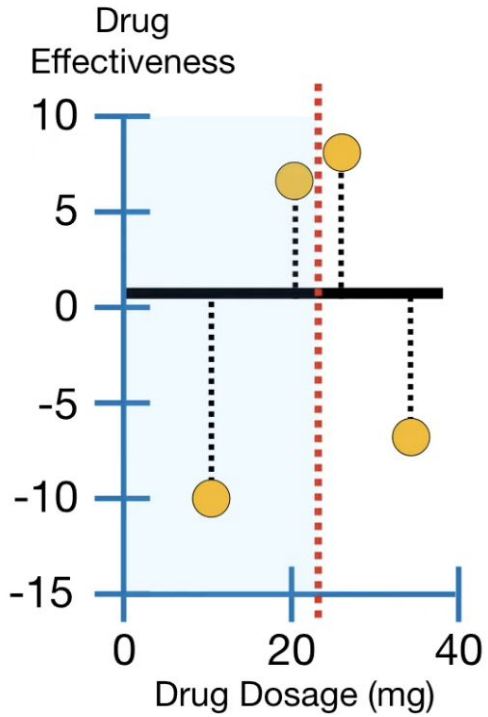
The gain for previous threshold Dosage<15 was:

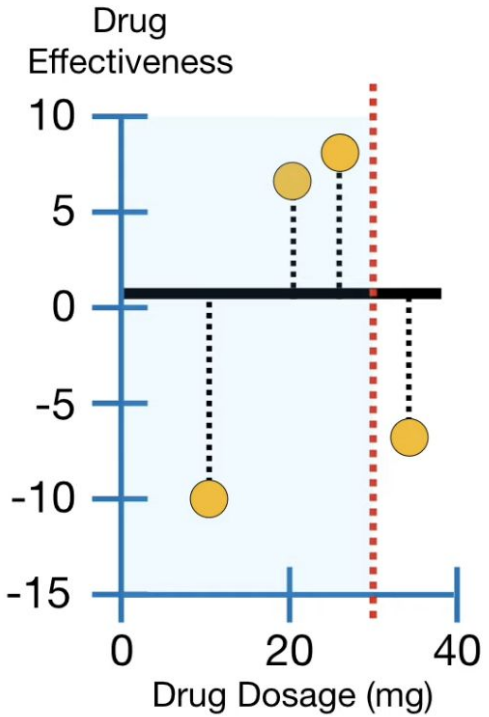
$$Gain = 110.25 + 14.08 - 4 = 120.33$$

Therefore the previous split has higher gain => gives better split

XGBoost example - predicting drug effectiveness

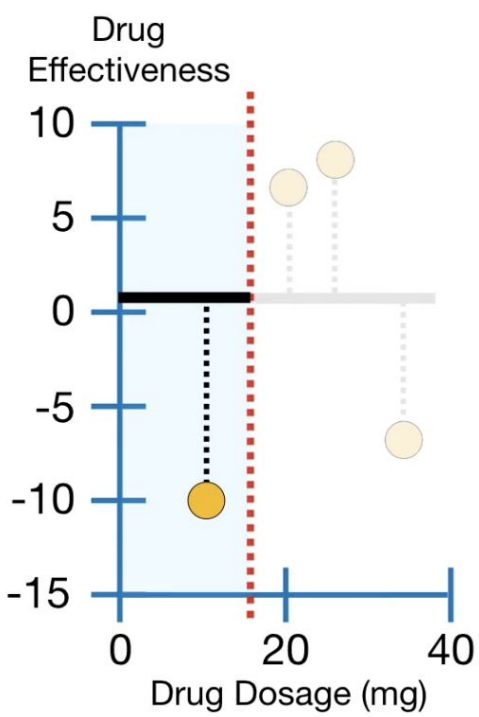


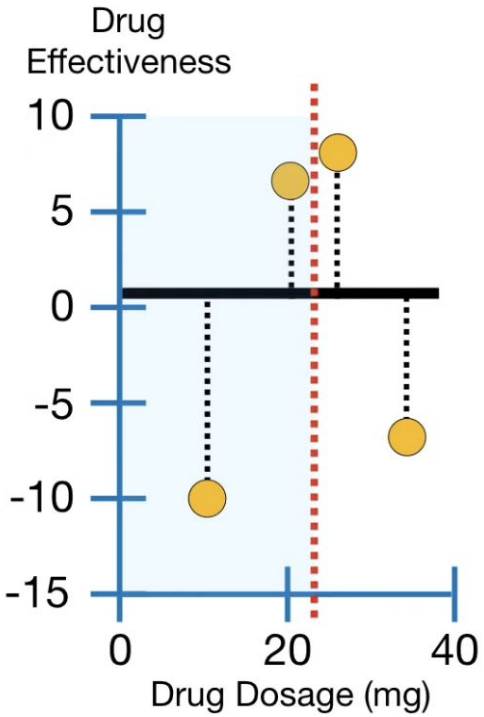


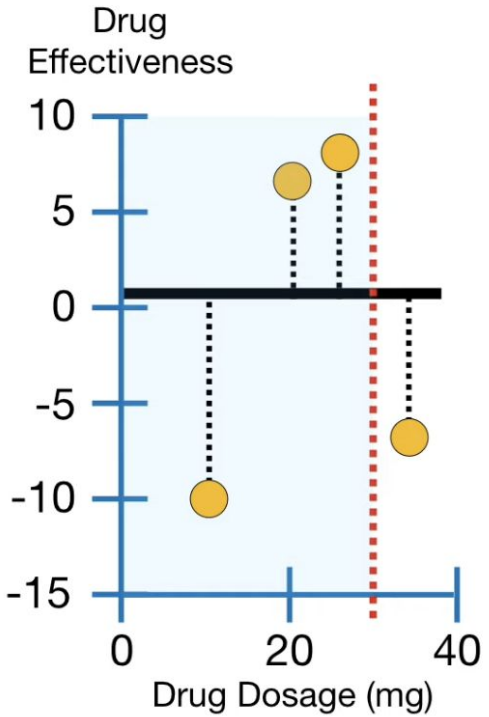


Consider a different threshold - shift it to the next value. Repeat

XGBoost example - predicting drug effectiveness





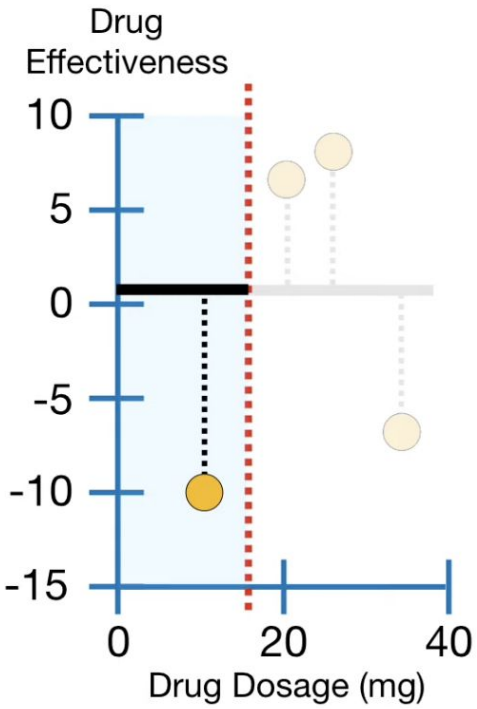


Gain = 110.25 + 14.08 - 4 = 120.33

Gain = 8 + 0 - 4 = 4

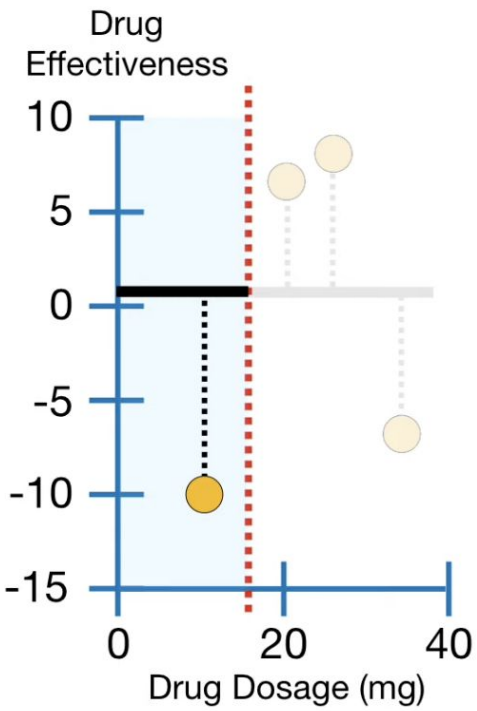
Gain = 4.08 + 56.25 - 4 = 56.33

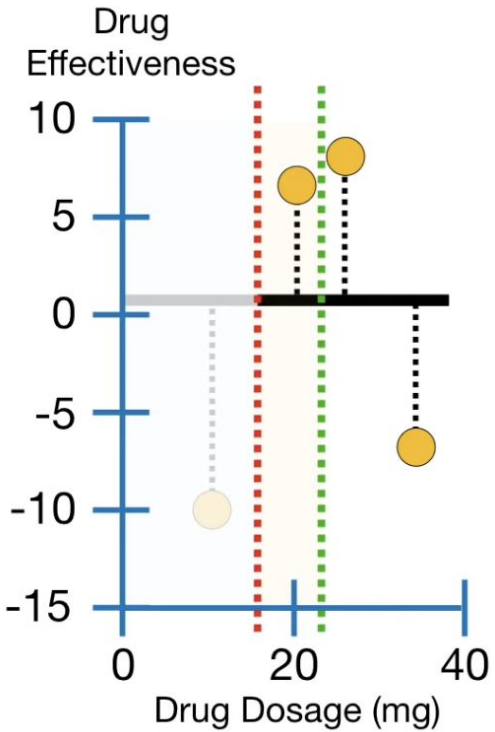
XGBoost example - predicting drug effectiveness



Fix the root based on the split that gives the highest gain.

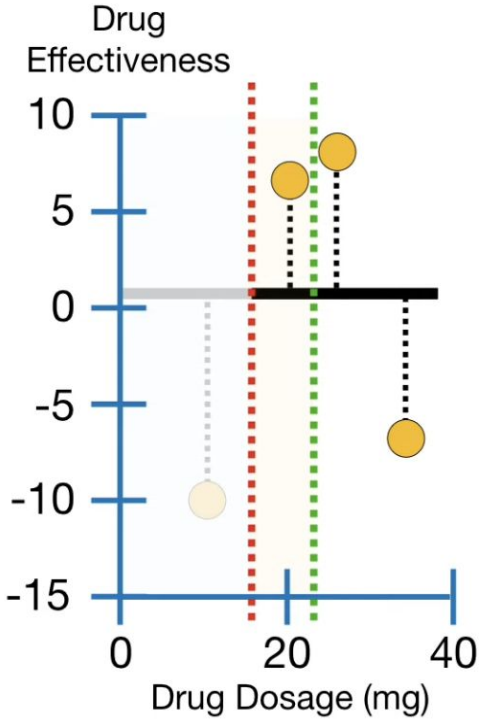
XGBoost example - predicting drug effectiveness

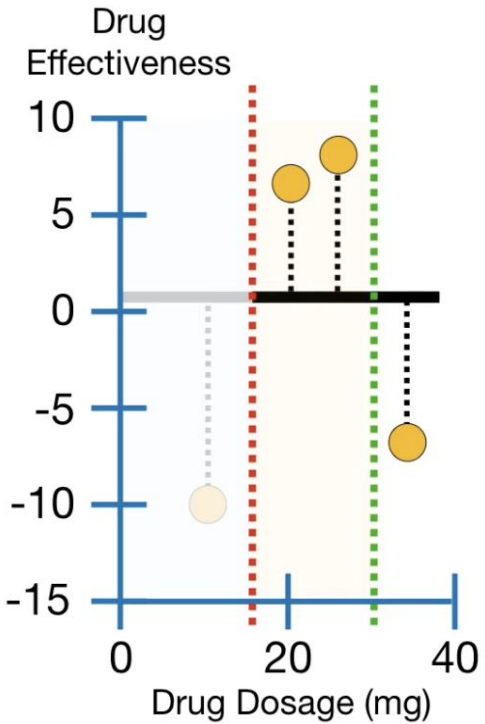


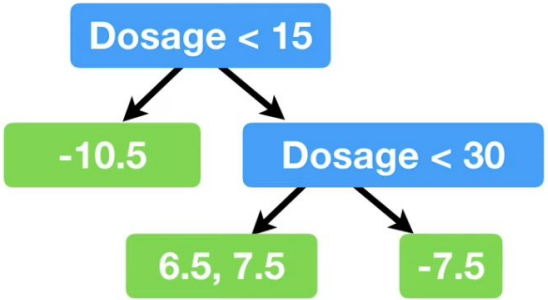


Repeat the process for the remaining residuals.

XGBoost example - predicting drug effectiveness







Repeat the process for the remaining residuals.

XGBoost example - predicting drug effectiveness

Predictions take into account the initial prediction and weighted subsequent trees.

output = average of values in the leaf

XGBoost example - predicting drug effectiveness

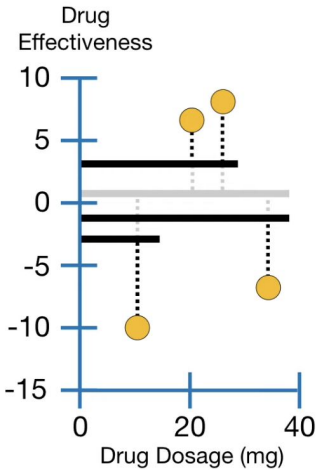
Predictions take into account the initial prediction and weighted subsequent trees.

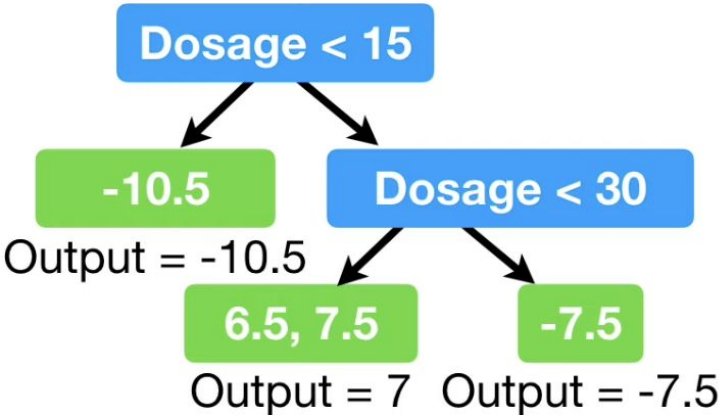
Predicted Drug Effectiveness

0.5

+

0.3 X

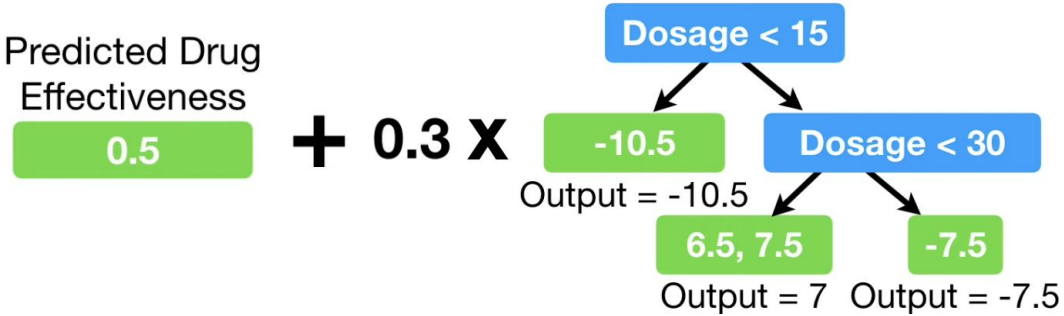


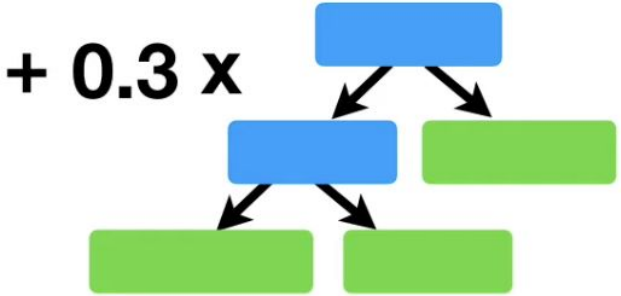


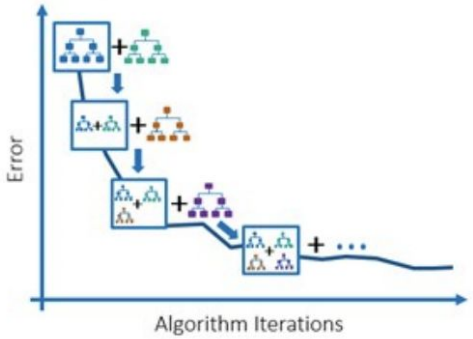
output = average of values in the leaf

XGBoost example - predicting drug effectiveness

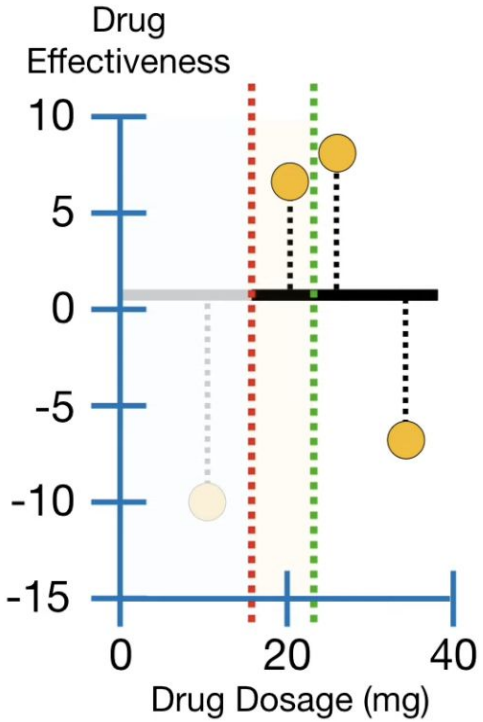
Continue building trees that lower the residuals.

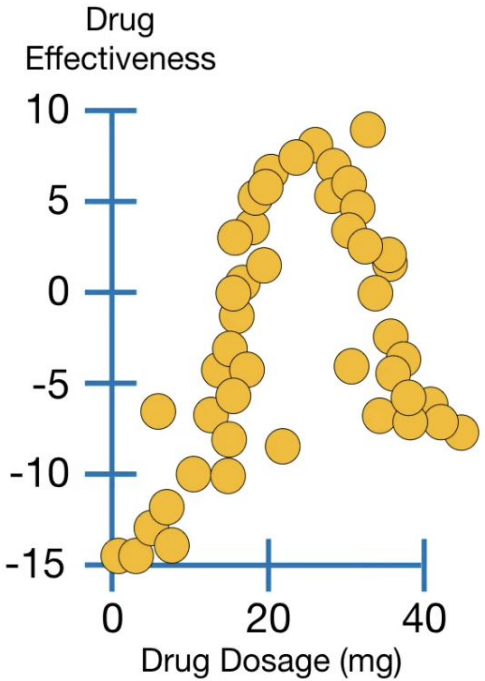




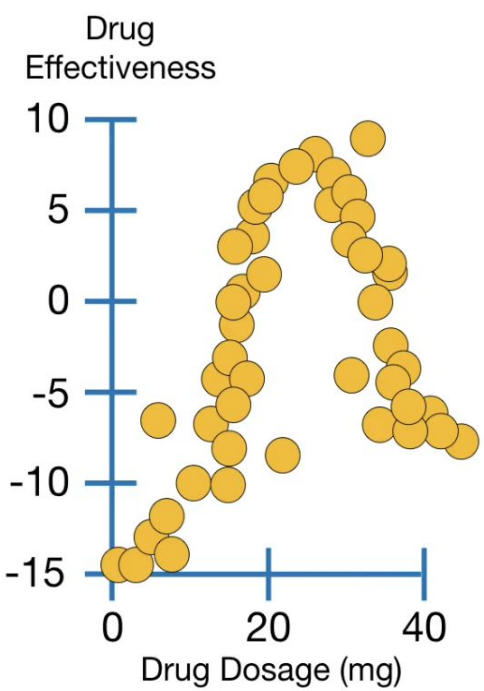


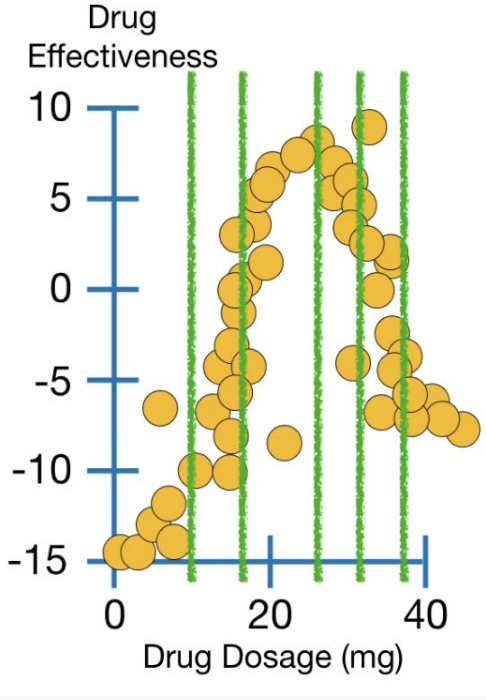
XGBoost - approximate greedy algorithm





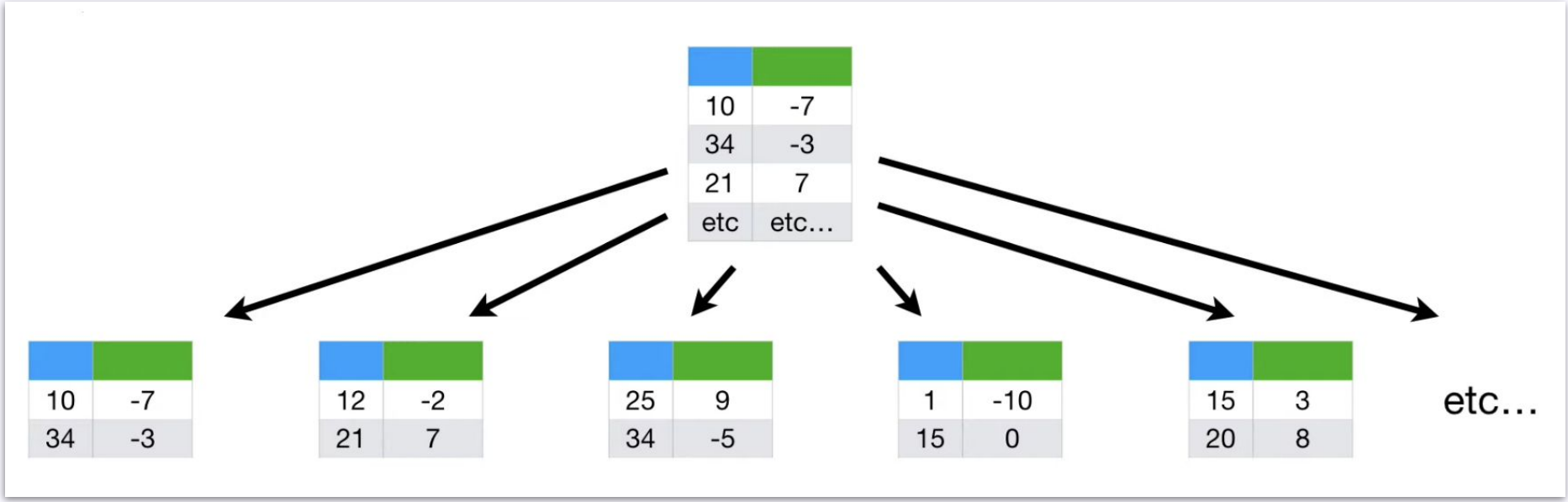
Learning on large datasets





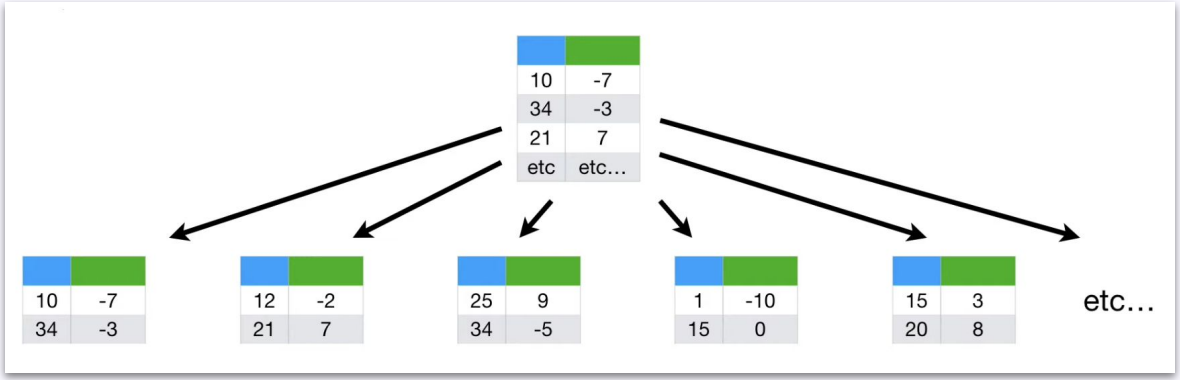
Use quantiles to fix thresholds.

Learning on large datasets



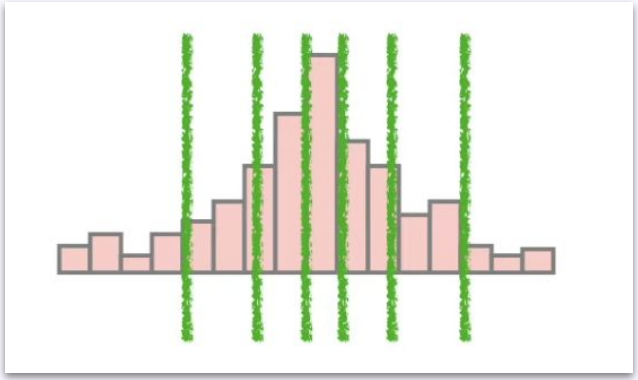
For very large datasets: Split the data into partitions.

XGBoost - approximate greedy algorithm



Combine the values from each partition to find an approximate histogram.

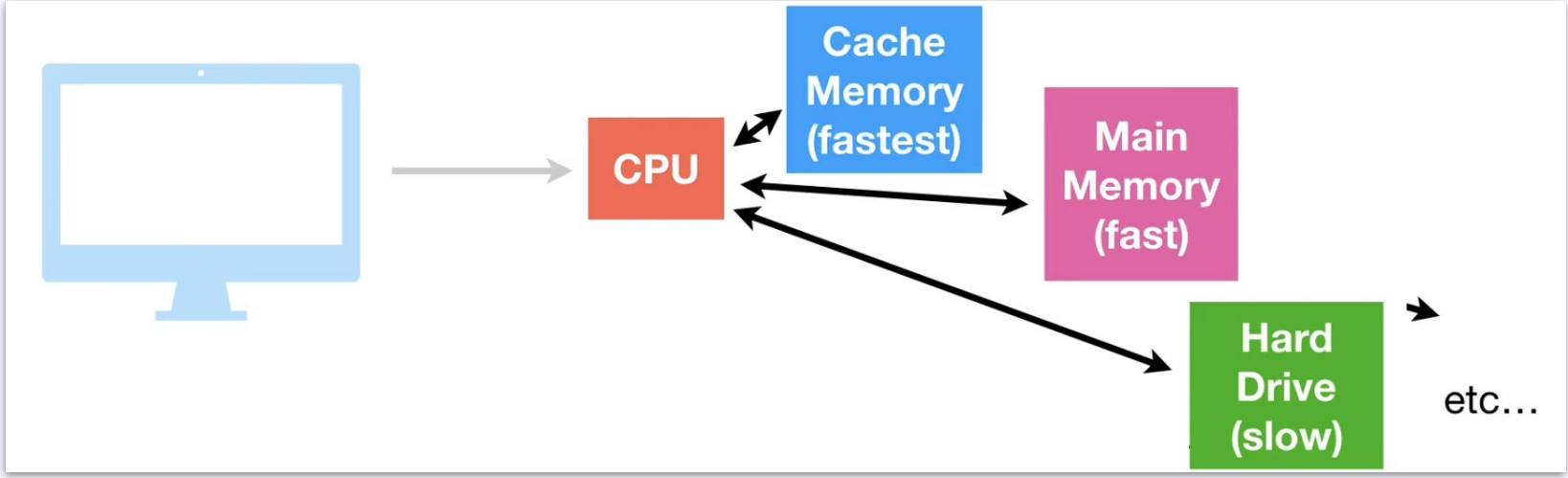
The approximate histogram is used to calculate approximate quantiles.



XGBoost optimizations

System optimizations:

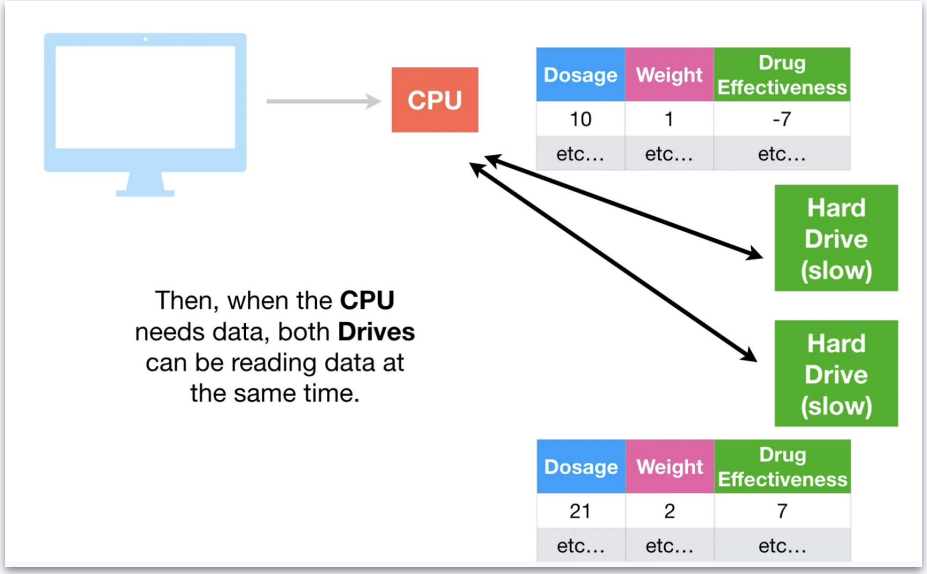
- Cache aware access



XGBoost optimizations

System optimizations:

- Cache aware access
- Blocks for out of core computation



XGBoost optimizations

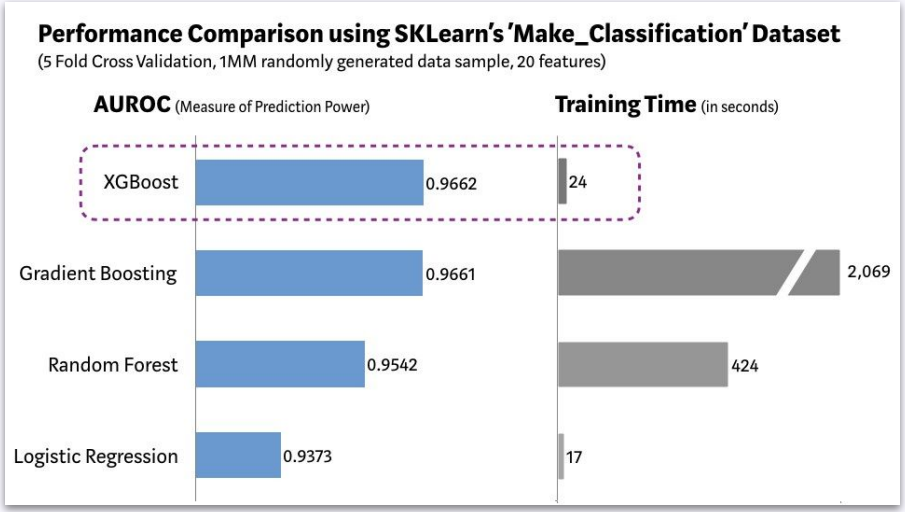
System optimizations:

- Cache aware access
- Blocks for out of core computation
- Distributed computing
  - build trees on subsets of the data/of the features

XGBoost optimizations

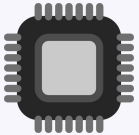
System optimizations:

- Cache aware access
- Blocks for out of core computation
- Distributed computing
  - build trees on subsets of the data/of the features



XGBoost

Cache awareness and out-of-core computing



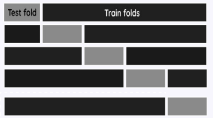
Regularization to avoid overfitting

Tree pruning using depth-first approach

Efficient handling of missing data

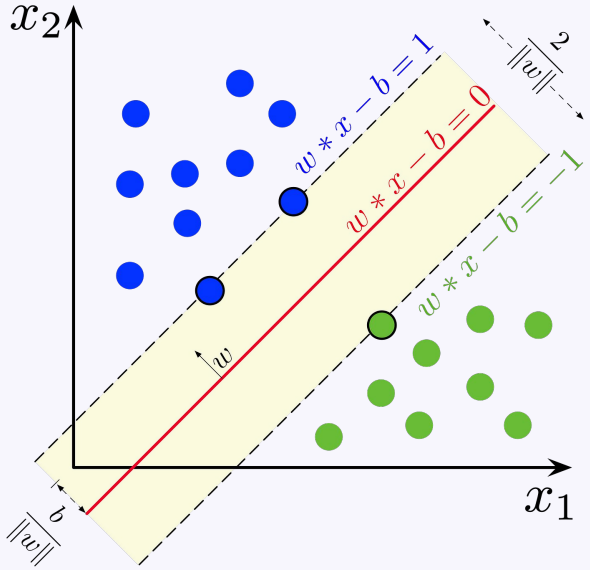
Parallelized tree building

In-built cross-validation capability



[Resources about gradient boosting, XGboost etc.]

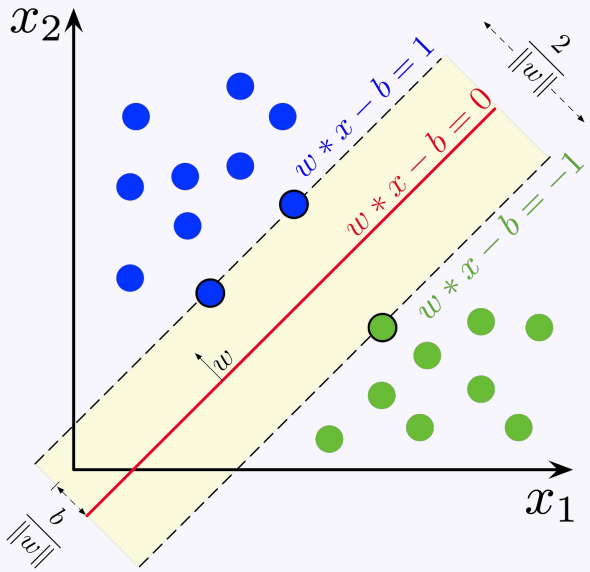
Support vector machines (SVM)

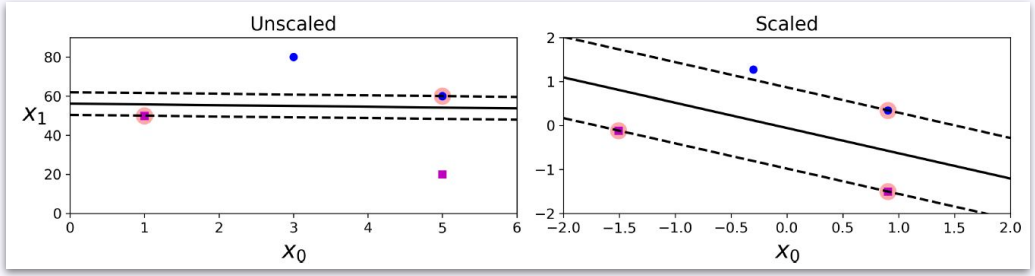


Powerful and versatile models used for: classification, regression, outlier detection.

SVMs work particularly well on small and medium sized datasets.

Support vector machines (SVM)



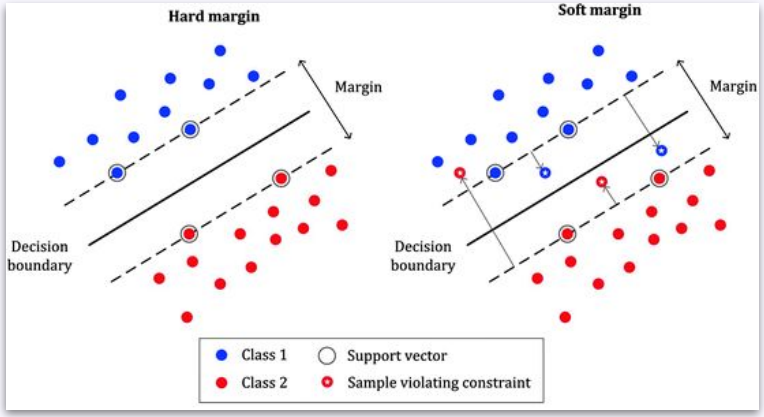


Powerful and versatile models used for: classification, regression, outlier detection.

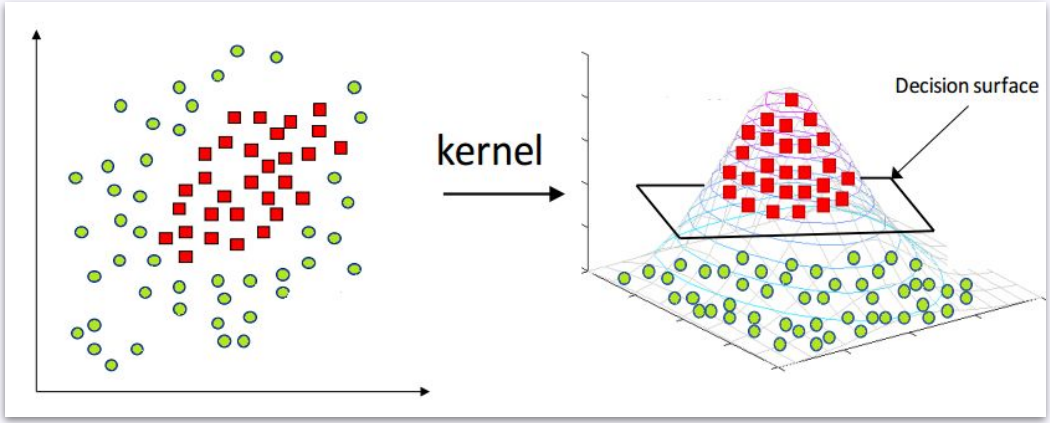
SVMs work particularly well on small and medium sized datasets.

Sensitive to the scale of the data!

SVM in practice



Soft margin classification - allows a few instances to intersect the boundary band



Use the kernel trick for data that is not linearly separable - data might be separable in a higher dimensional space

SVM computational complexity

Separating the support vectors from the other samples - heavy computation

- training complexity between $O(n^2 \times \#\text{ features})$ to $O(n^3)$
- ok when # of dimensions is large, inefficient when *n* is large

(impractical beyond tens of thousands of samples).

SVMs are useful:

- Small to medium datasets (< 10,000 samples)
- High-dimensional, low-sample problems
- When you have good kernel knowledge for your domain
- When you need strong theoretical guarantees

Computational efficiency

If you are working with large datasets and want to apply SVM:

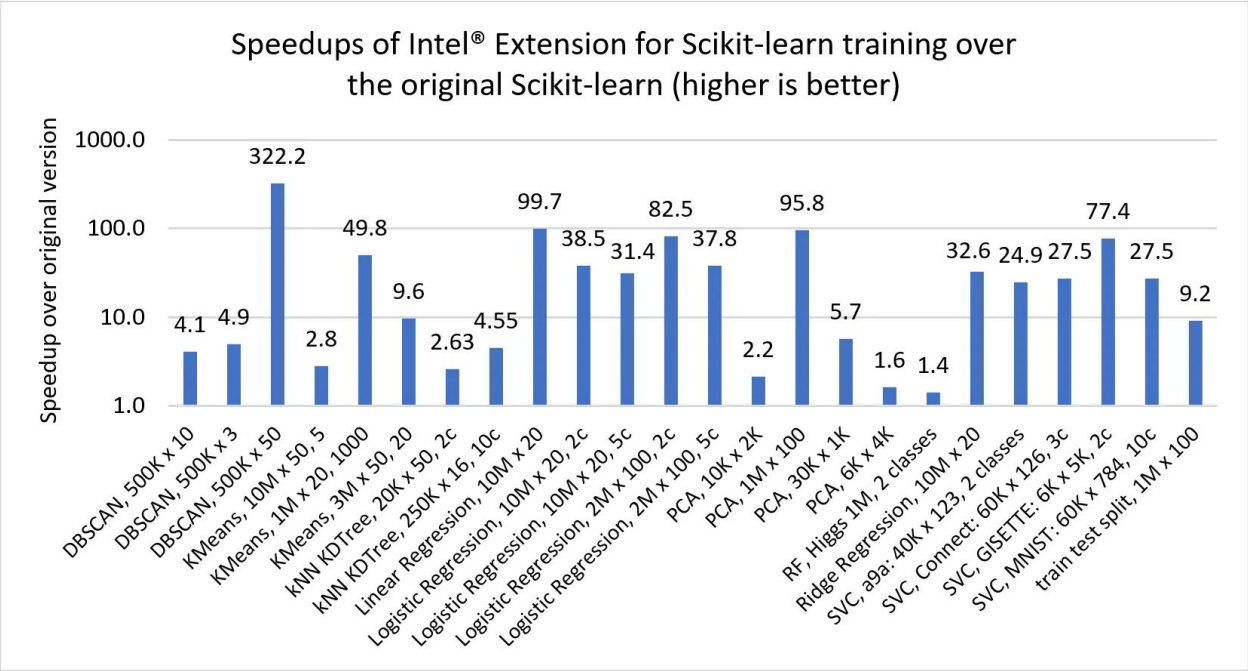
Linear case -> use LinearSVC in sklearn (can scale almost linearly to millions of samples and/or features)

Bagging -> train SVMs on samples of the dataset

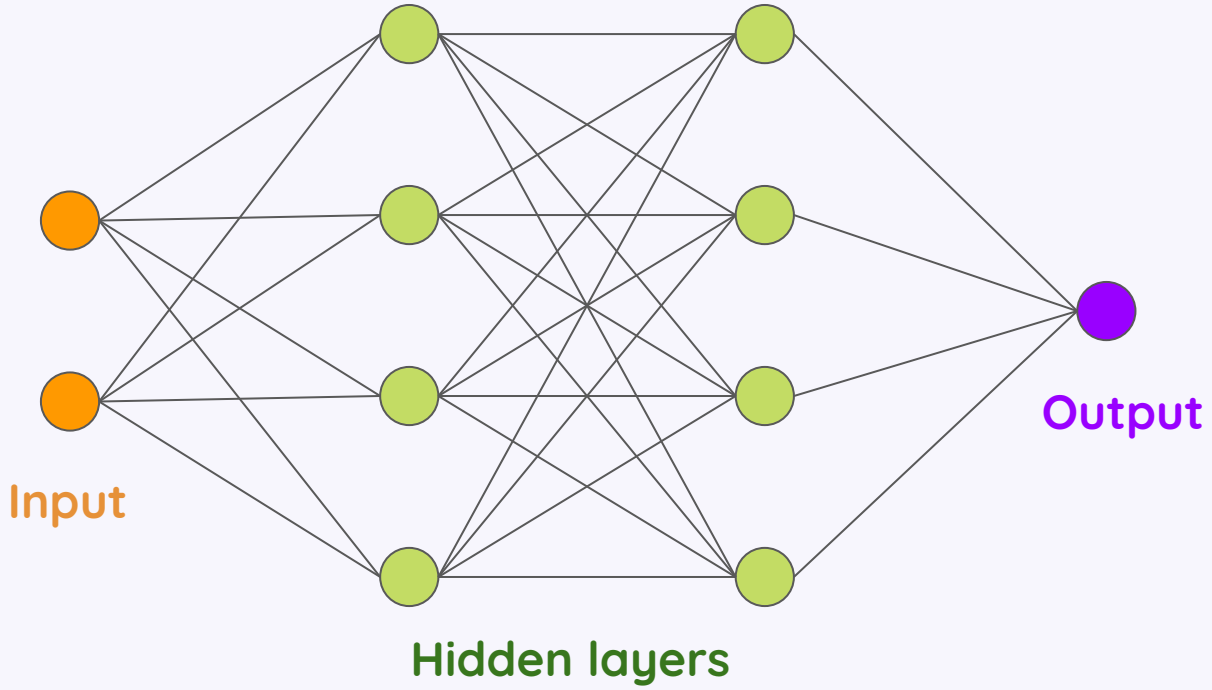
Kernel approximation -> approximate the feature mappings that correspond to certain kernels

Intel(R) Extension for Scikit-learn

Speed up your scikit-learn applications across single- and multi-node configurations.

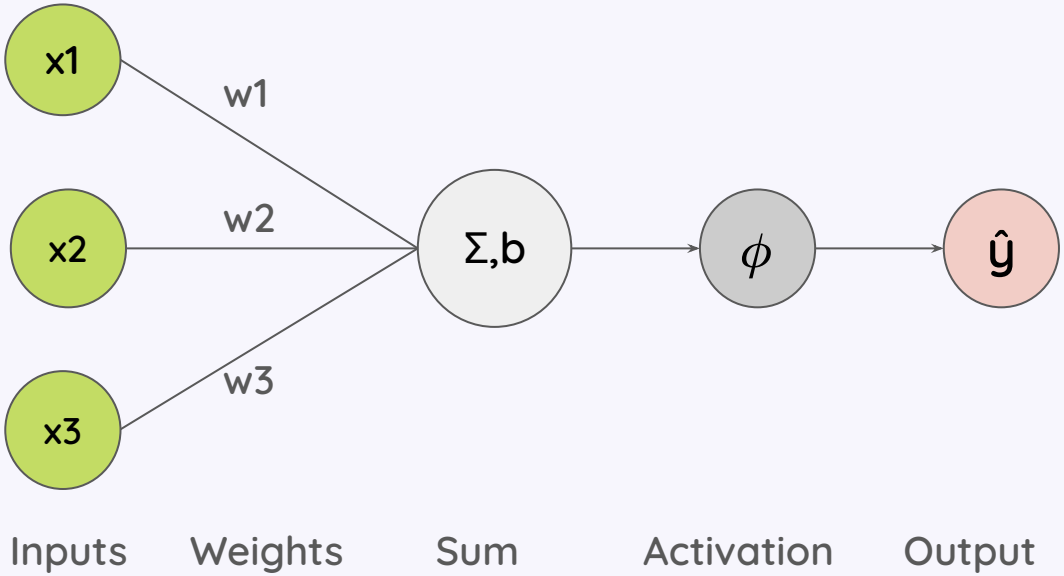


Neural networks



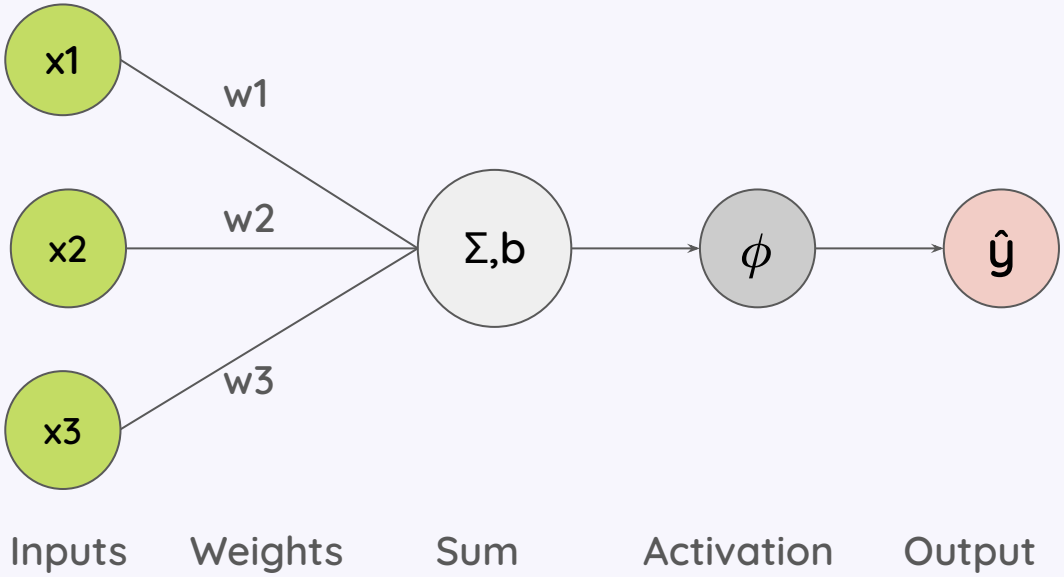
A fully connected neural network

Inside a neuron



Operations inside a neuron: multiply each input (x_i) with the corresponding weight (w_i), add them up, add bias, and apply an activation function

Activation functions



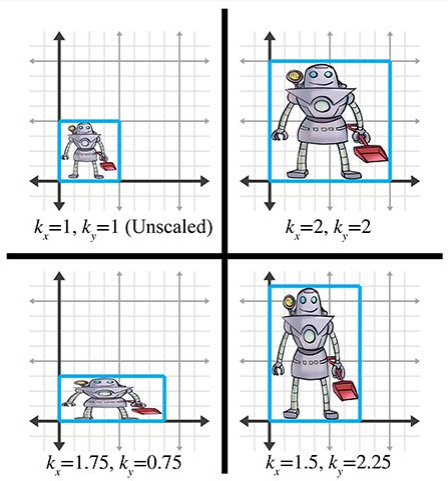
The purpose of having activations functions is to add **non-linearity**.

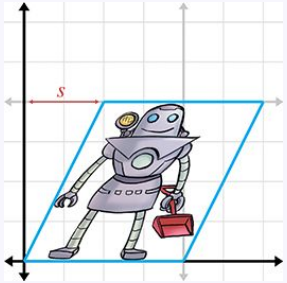
Without non-linearity all we could represent with neuron networks would be linear functions.

Linear vs. non-linear

Linear transformations:

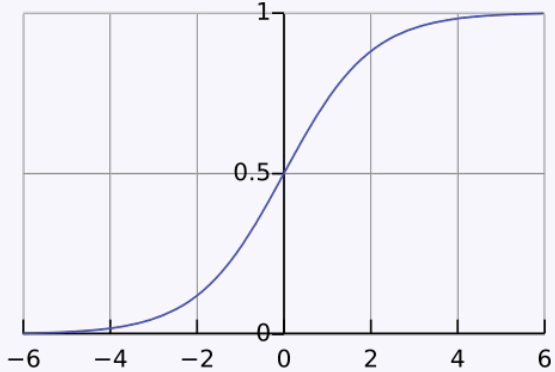
- 2*x1 + 4
- 5*x2 - 2
- …



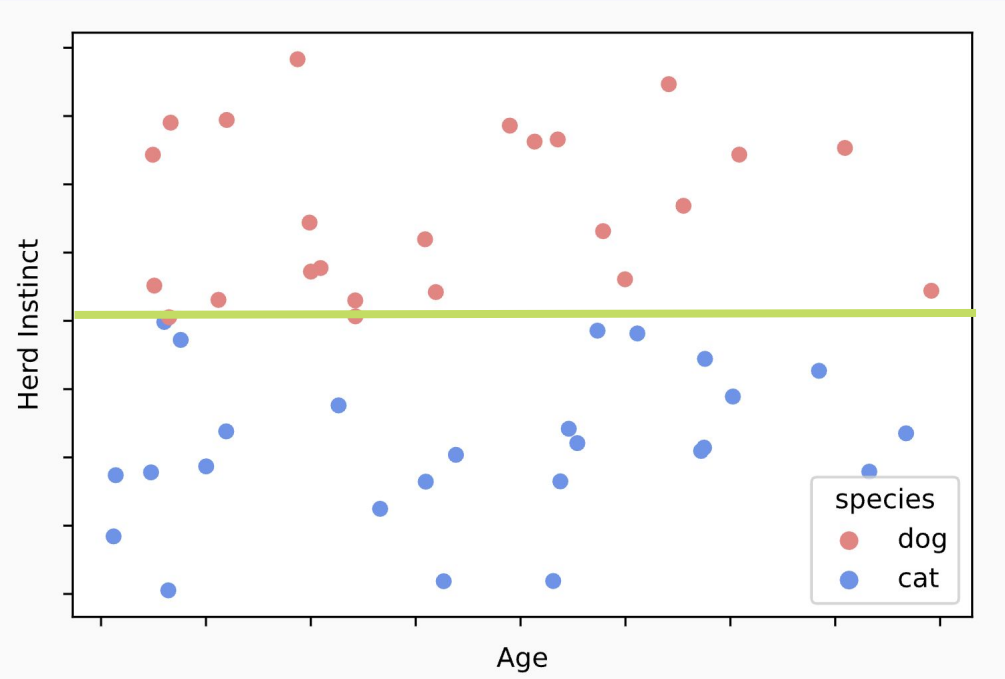


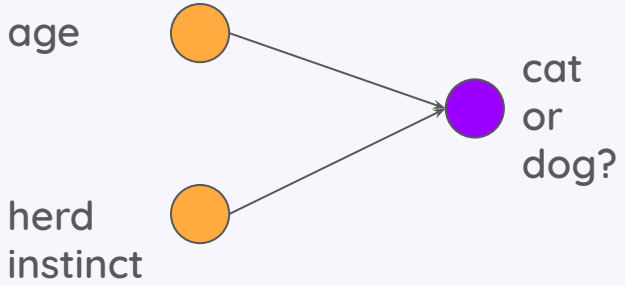
Non-linear transformations:

- max(2*x1,0)
- 1/(1+exp(-x2))
- …



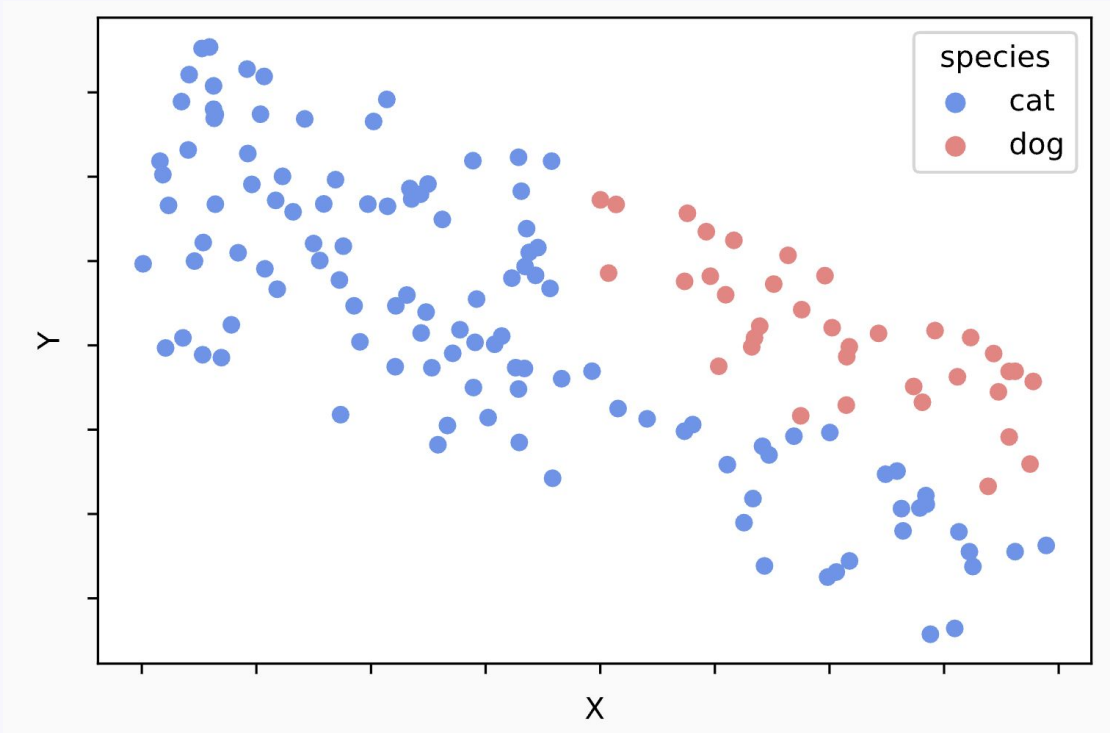
Learning a function

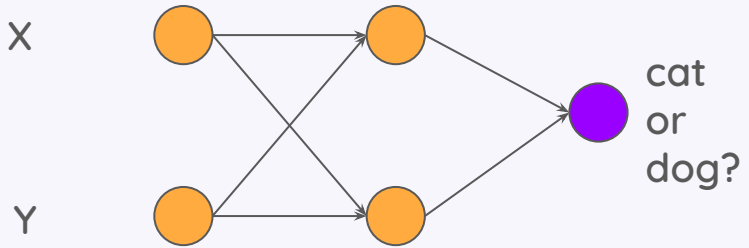




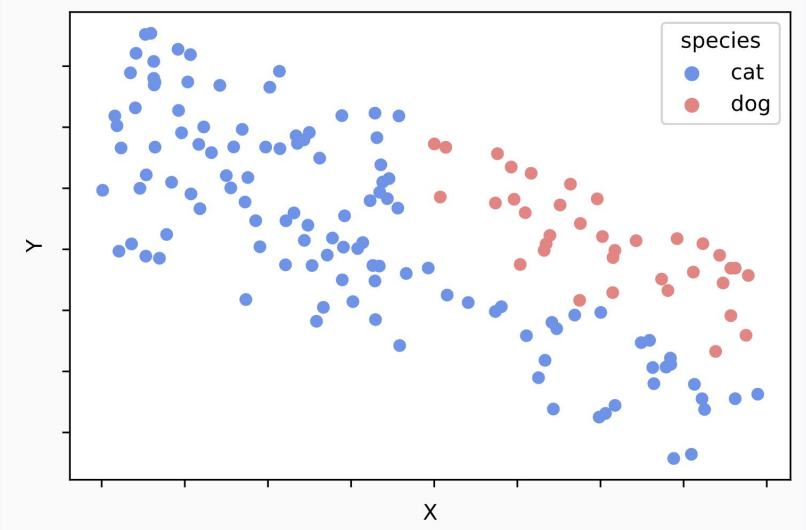
Adding non-linearity: everything under the decision boundary is classified as a cat and every data point above is of class dog.

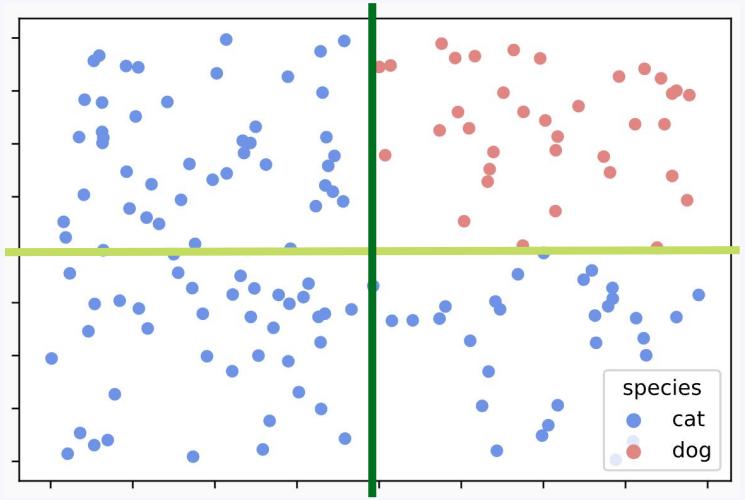
Deep neural networks

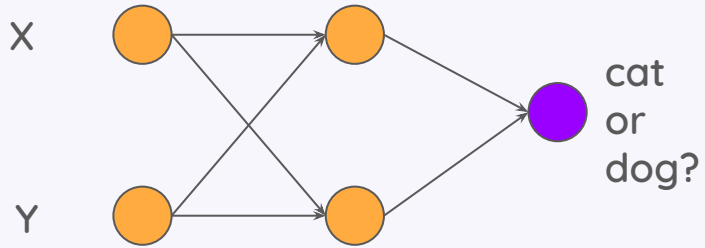




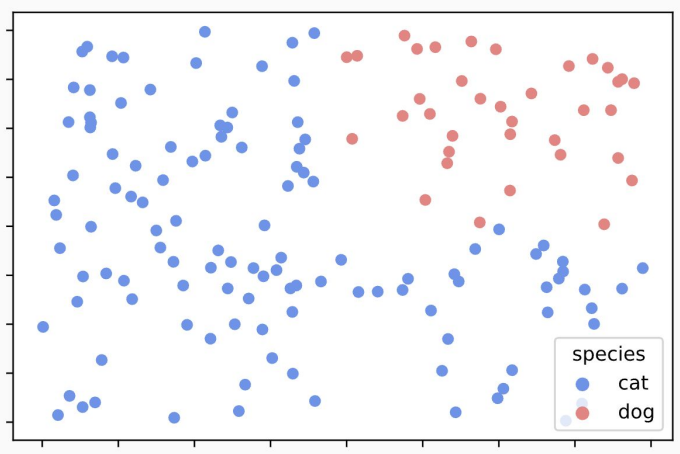
Deep neural networks

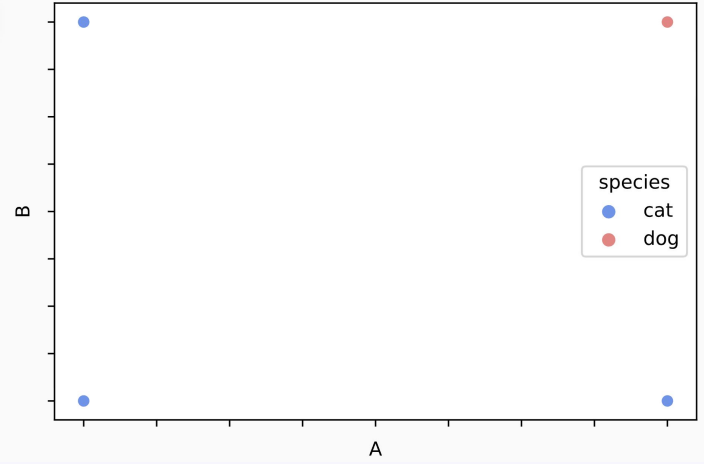


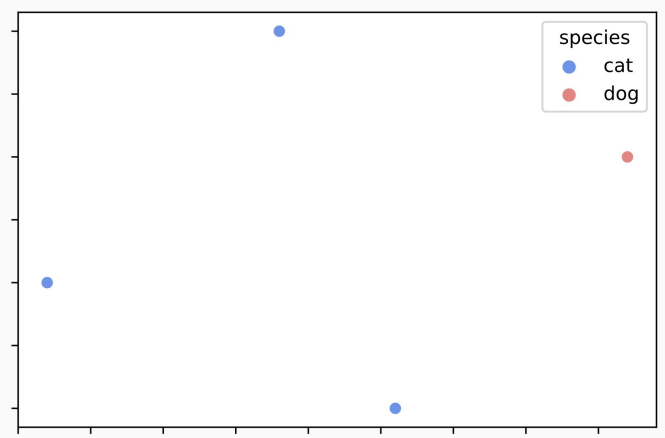


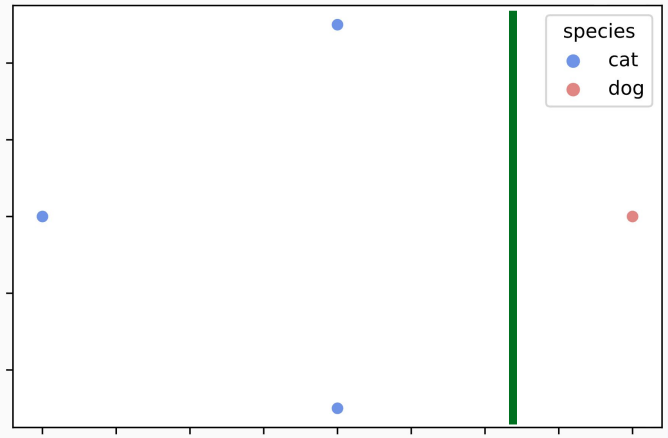


Rotation and non-linear transformation









Universal approximators

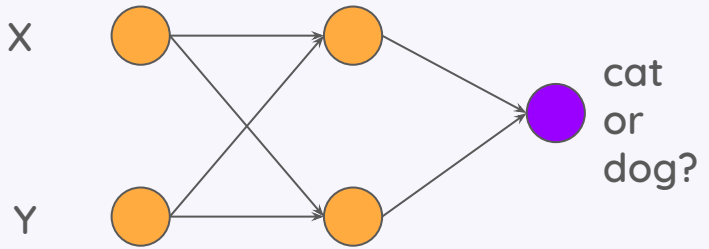
Universal Approximation Theorem

An arbitrarily wide Neural Network can approximate any continuous function on a specified range of inputs.

- Initialize the parameters (eg. weights and bias)
- Apply the transformations
- Calculate how close is the output to the desired output
- Based on the prediction error, tweak the parameters, so that the error decreases and repeat

Backpropagation in neural networks

Backpropagation (introduced in 1986) is the algorithm that finds out how much each output connection contributes to the error and uses an optimizer (such as Gradient Descent) to tweak each weight and bias in order to decrease the order.



error = number of instances misclassified

1. for each data point make a prediction based on current weights and biases

2. calculate error

3. calculate how much each output connection contributed to error

4. adjust parameters (weights and biases)

Distributed model training

Distributed training involves spreading training workload across multiple **processors** or **worker nodes**.

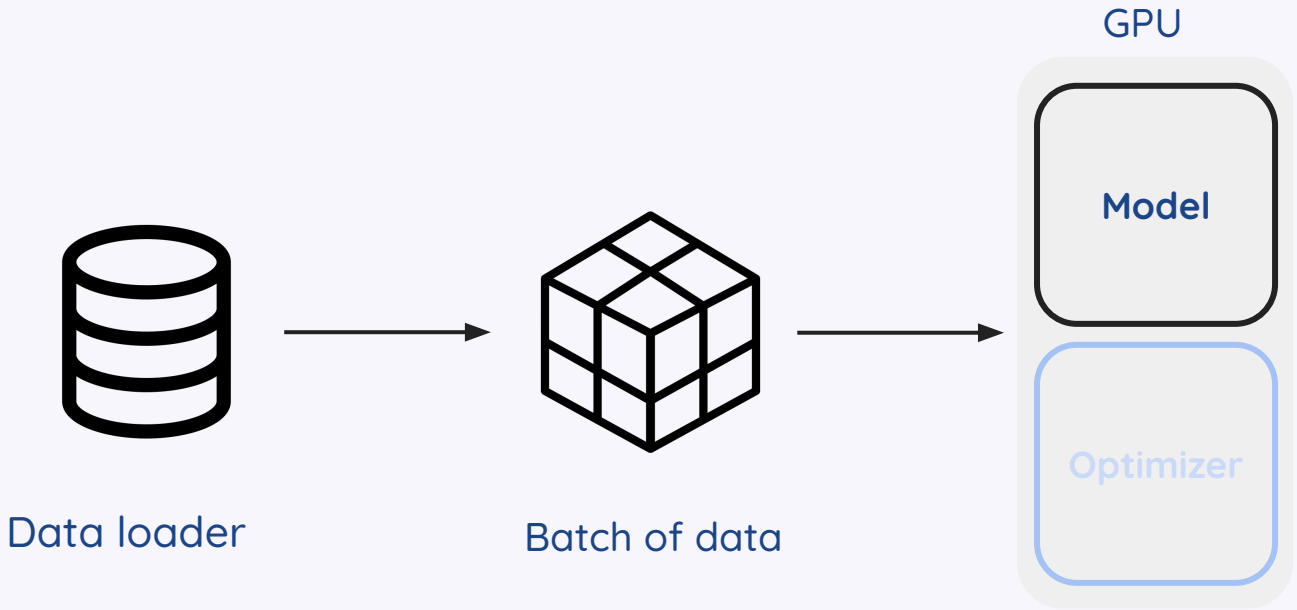
⇒ can improve training speed and accuracy

A worker node may be a virtual or physical machine.

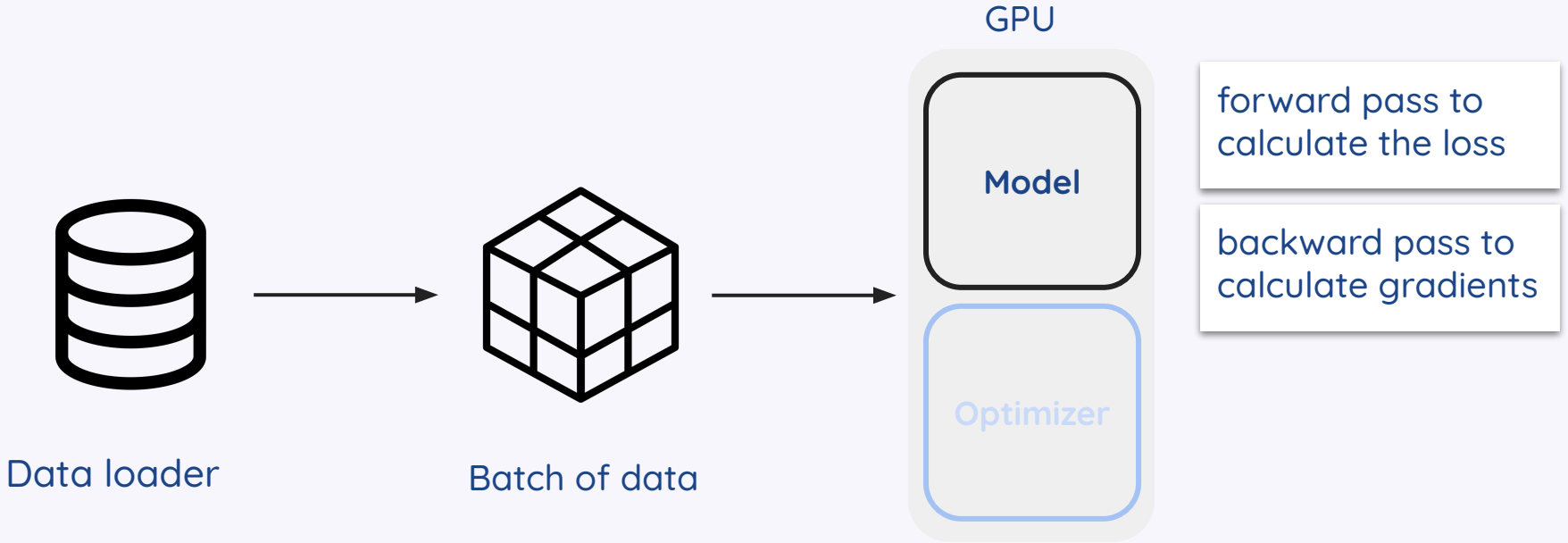
- Data parallelism
- Pipeline parallelism
- Parameter server training

(+ others: tensor parallelism, device mesh …)

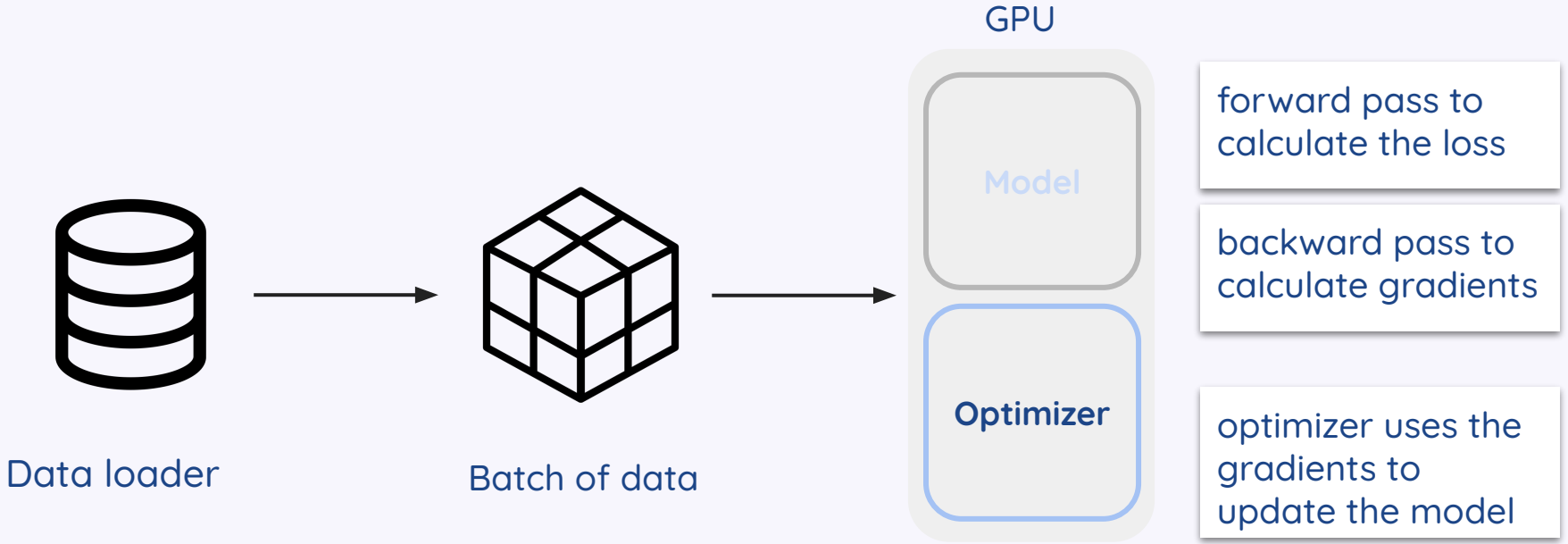
Deep learning on one machine, one GPU



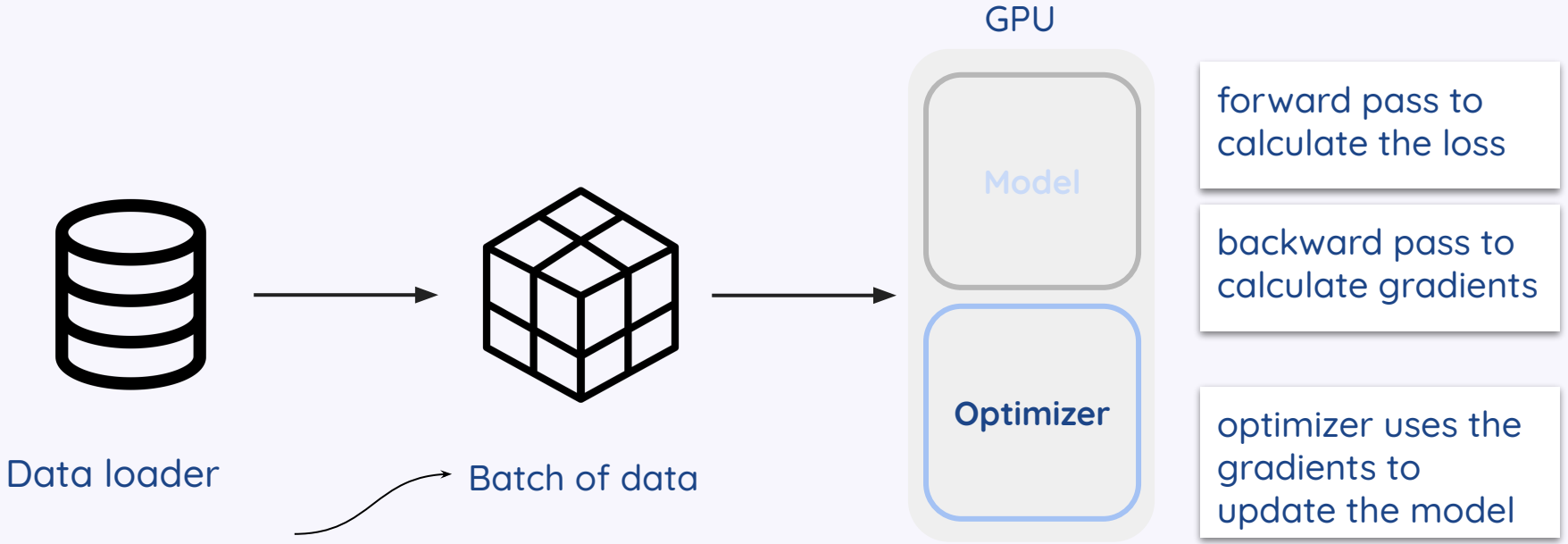
Deep learning on one machine, one GPU



Deep learning on one machine, one GPU



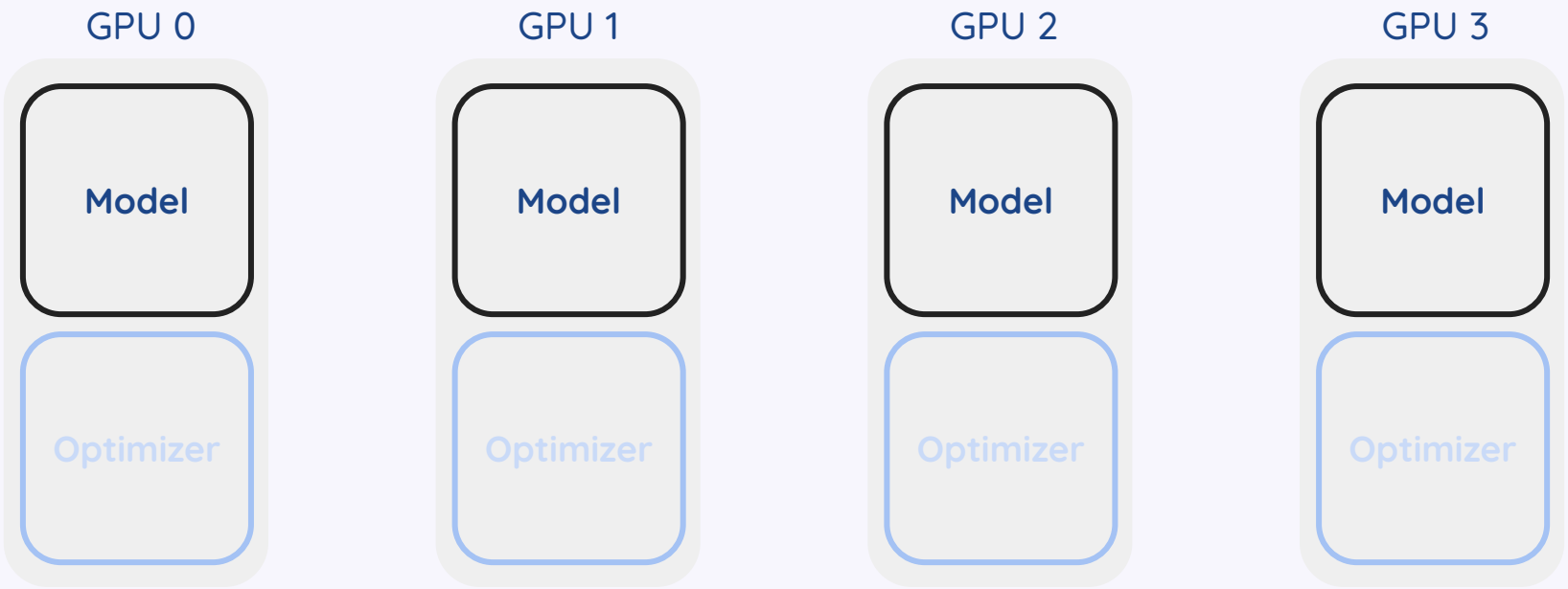
Deep learning on one machine, one GPU



Global batch size: number of records selected from the training dataset in each iteration to send to the GPUs/workers in a cluster.

Iteration: single forward and backward pass performed using a global batch sized batch
Epoch: one training cycle through the entire dataset

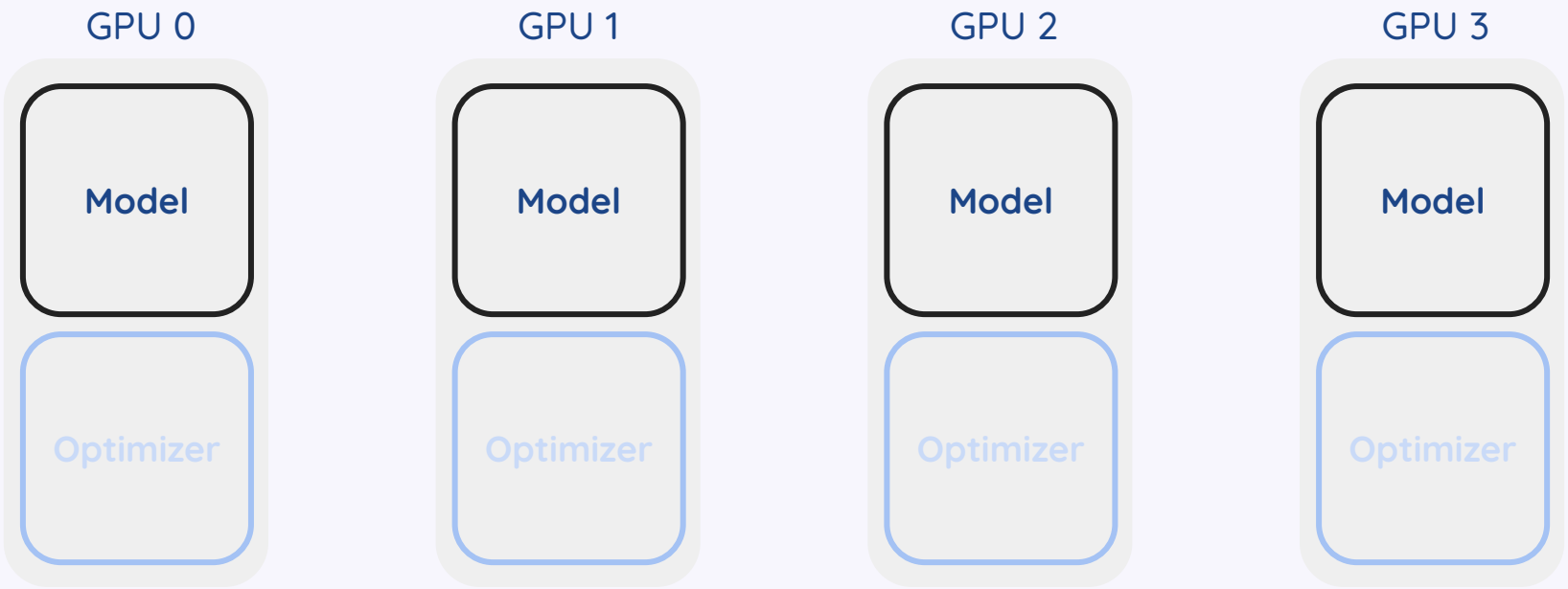
Training on multiple GPUs



Each GPU has a local copy of the model: same model parameters, optimizer parameters, random seed etc.

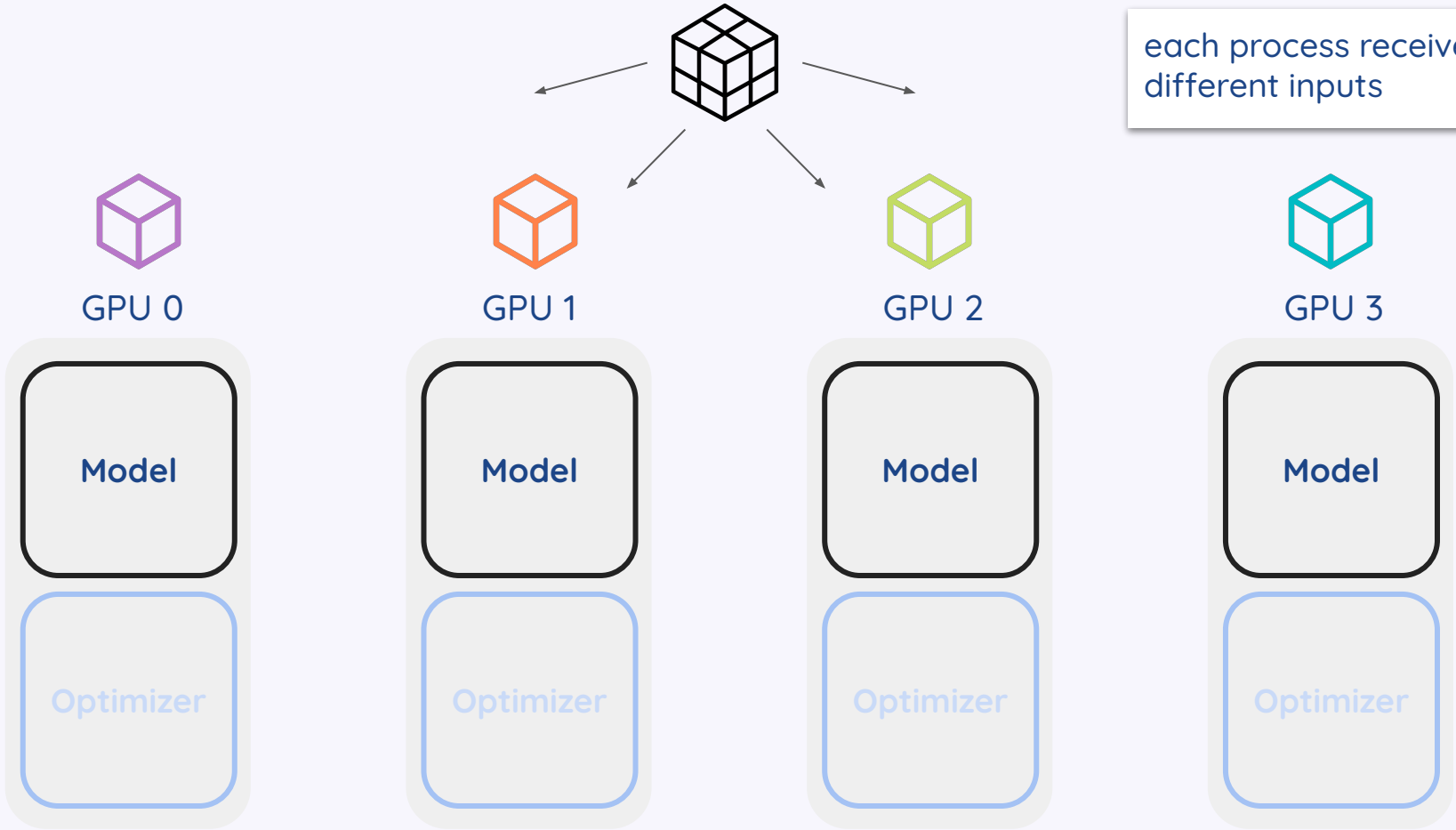
Training on multiple GPUs

Data parallelism: each processor will get a batch of the data

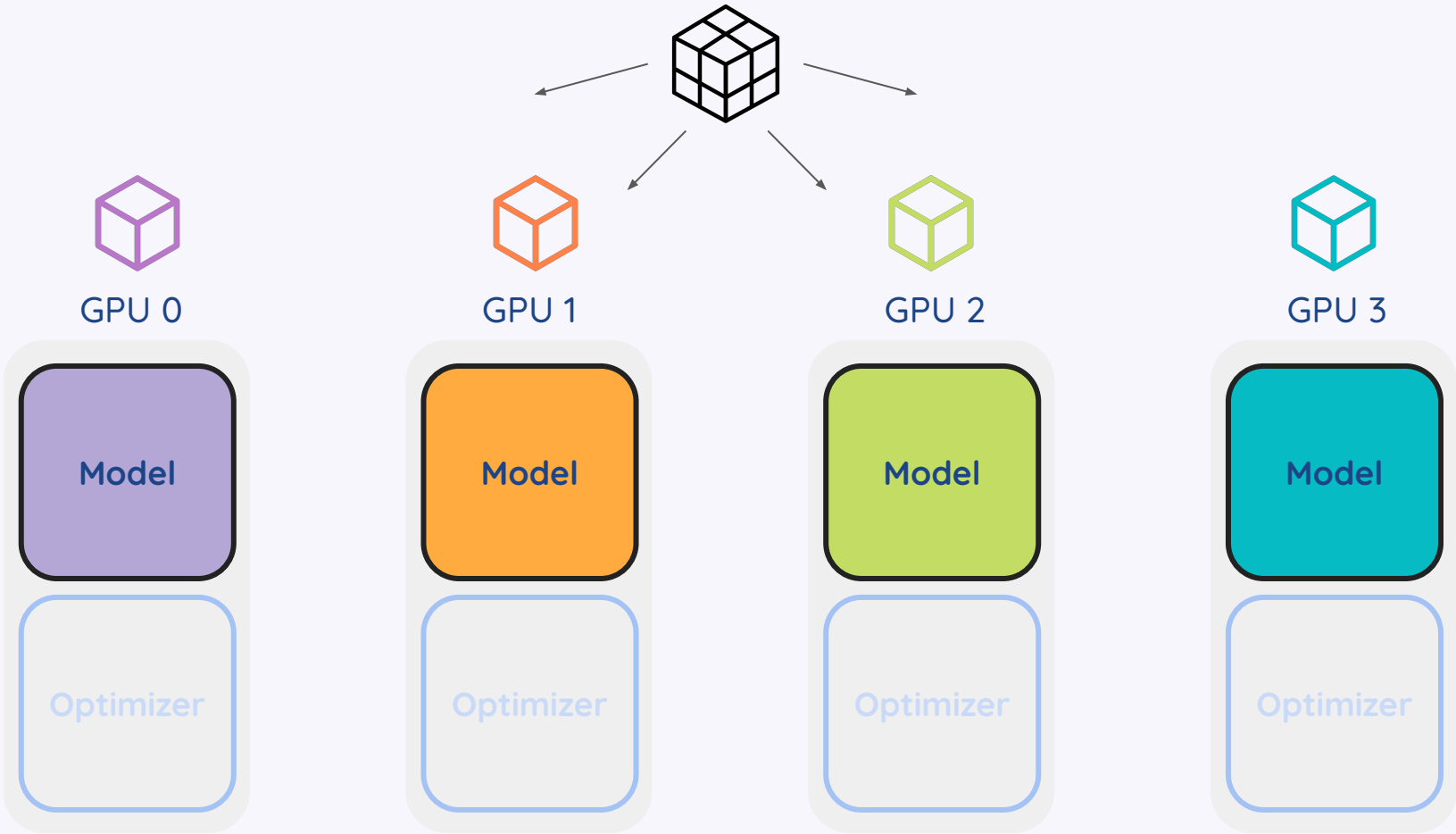


Each GPU has a local copy of the model: same model parameters, optimizer parameters, random seed etc.

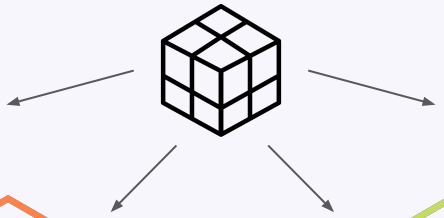
Training on multiple GPUs - data parallelism



Training on multiple GPUs - data parallelism



Training on multiple GPUs - data parallelism



running an optimizer
→ 4 different models

GPU 0

Model

Optimizer

GPU 1

Model

Optimizer

GPU 2

Model

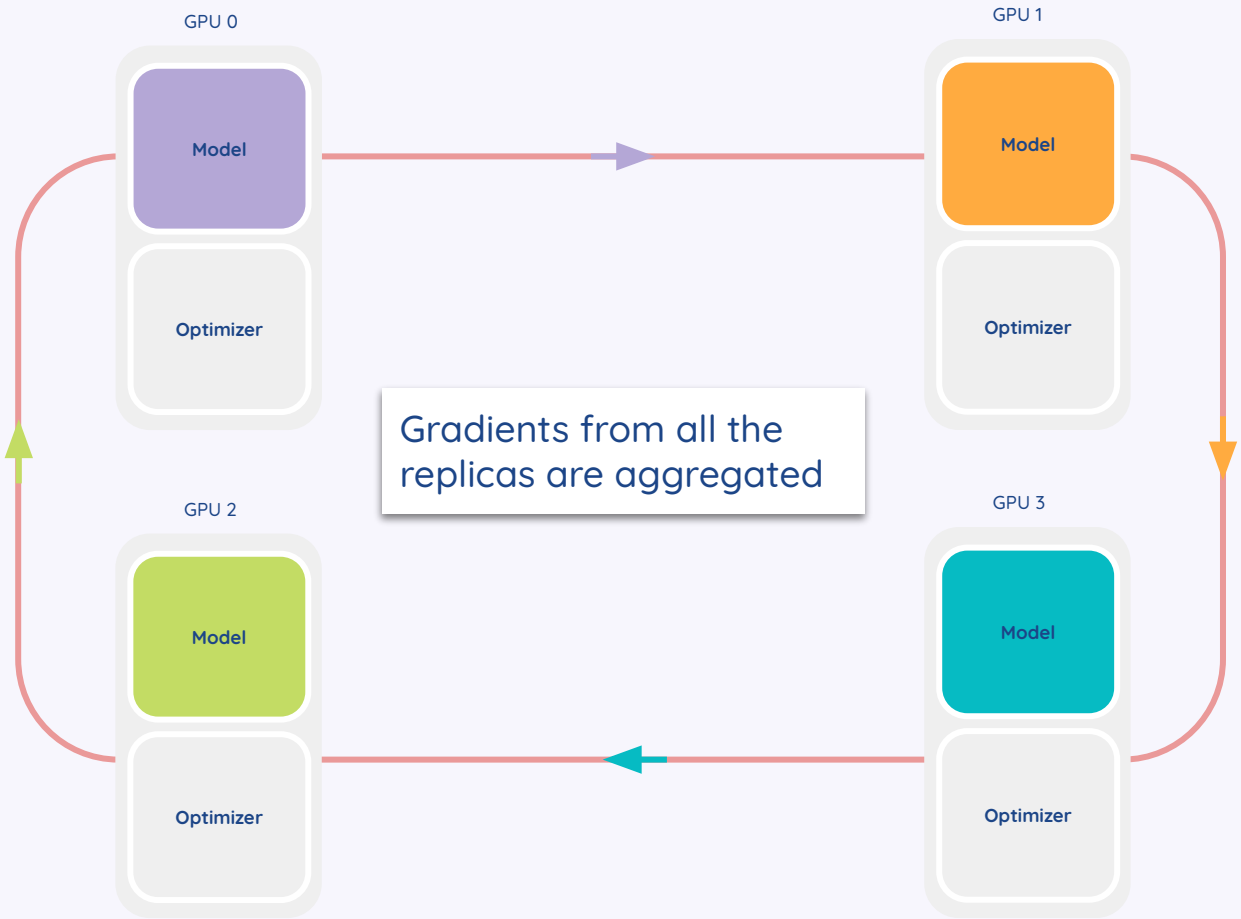
Optimizer

GPU 3

Model

Optimizer

Gradients aggregation



Synchronized replicas

Data parallelism for deep learning

Pytorch Data Parallel (DP): distribute the data across multiple GPUs on a single machine

Pytorch Distributed Data Parallel (DDP): training models across multiple processes or machines. Github repo for tutorial

Tensorflow MirroredStrategy: single machine with multiple GPUs

Tensorflow MultiWorkerMirroredStrategy: extends MirroredStrategy to distribute training across multiple machines. Each worker has access to one or more GPUs.

Tensorflow TPUStrategy: designed specifically for training models on Google's Tensor Processing Units (TPUs).

Fully sharded data parallelism

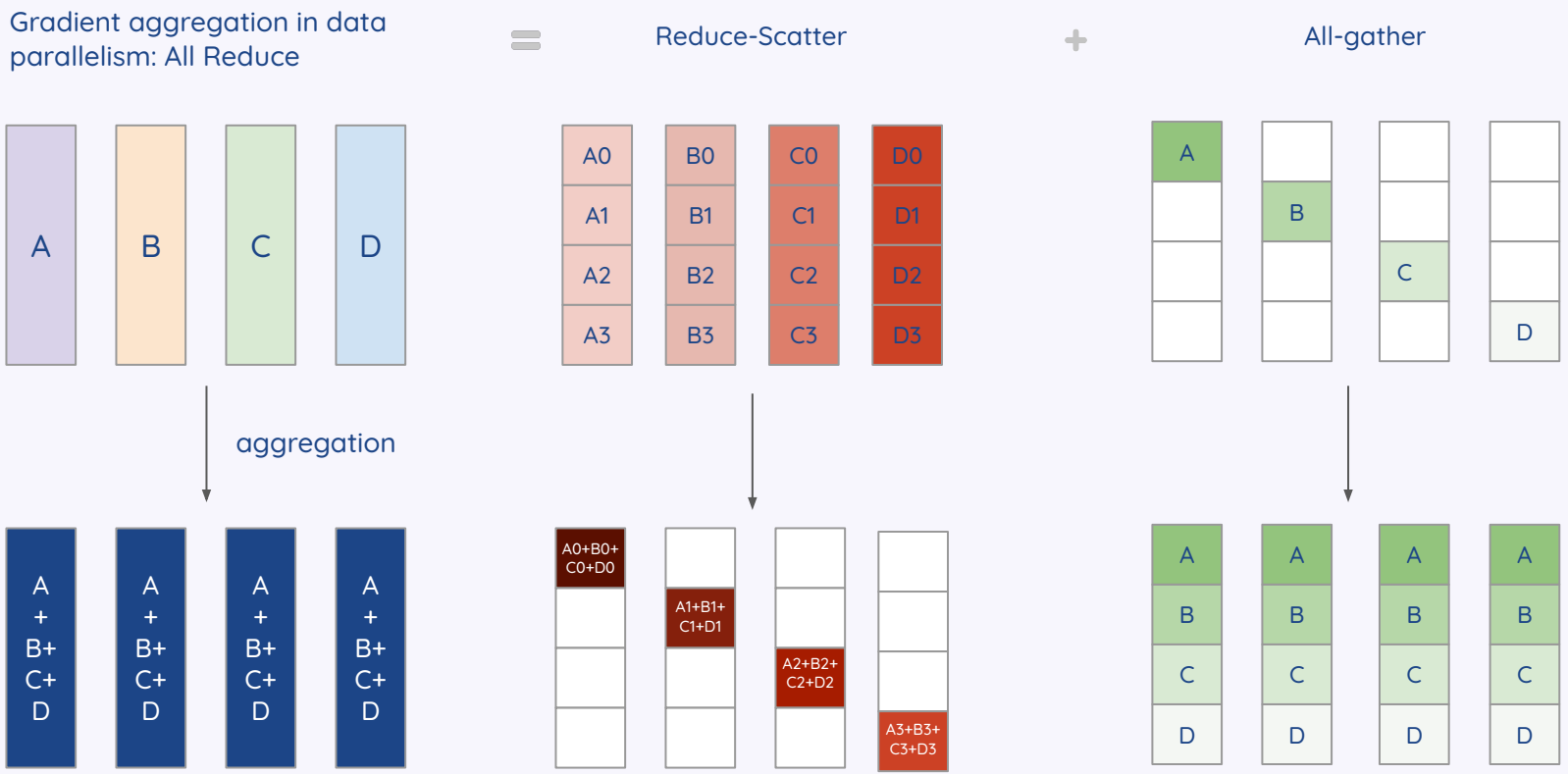
Data parallelism:

- each processor/worker had a replica of the model (model parameters, gradients, and optimizer states)

Fully sharded data parallelism:

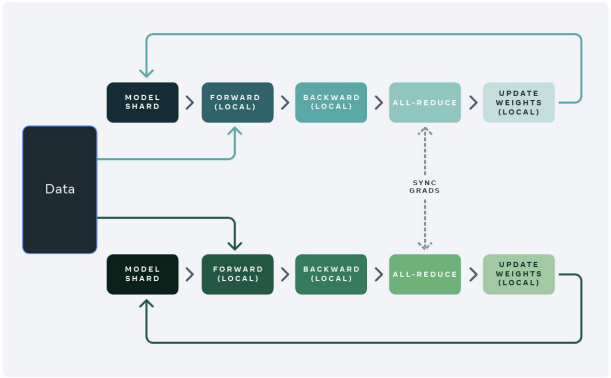
- model parameters, optimizer states and gradients across are distributed across workers
- ⇒ smaller GPU memory footprint, possible to train very large models
- but increased communication costs

Fully sharded data parallelism

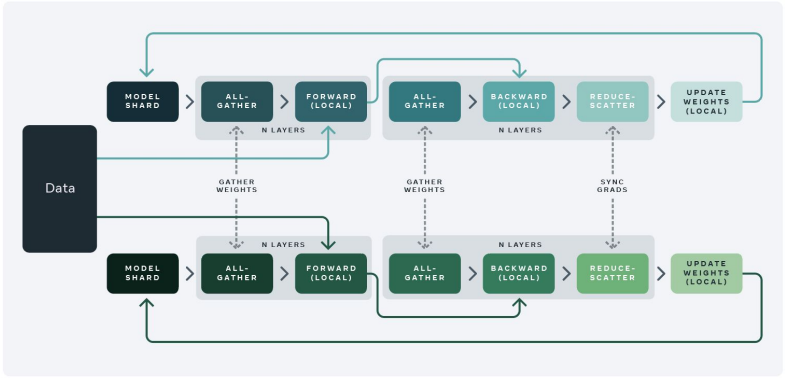


Fully sharded data parallelism

Standard data parallel training



Fully sharded data parallel training



Check this article for more details.

Next lecture

Model tracking, monitoring, and deployment