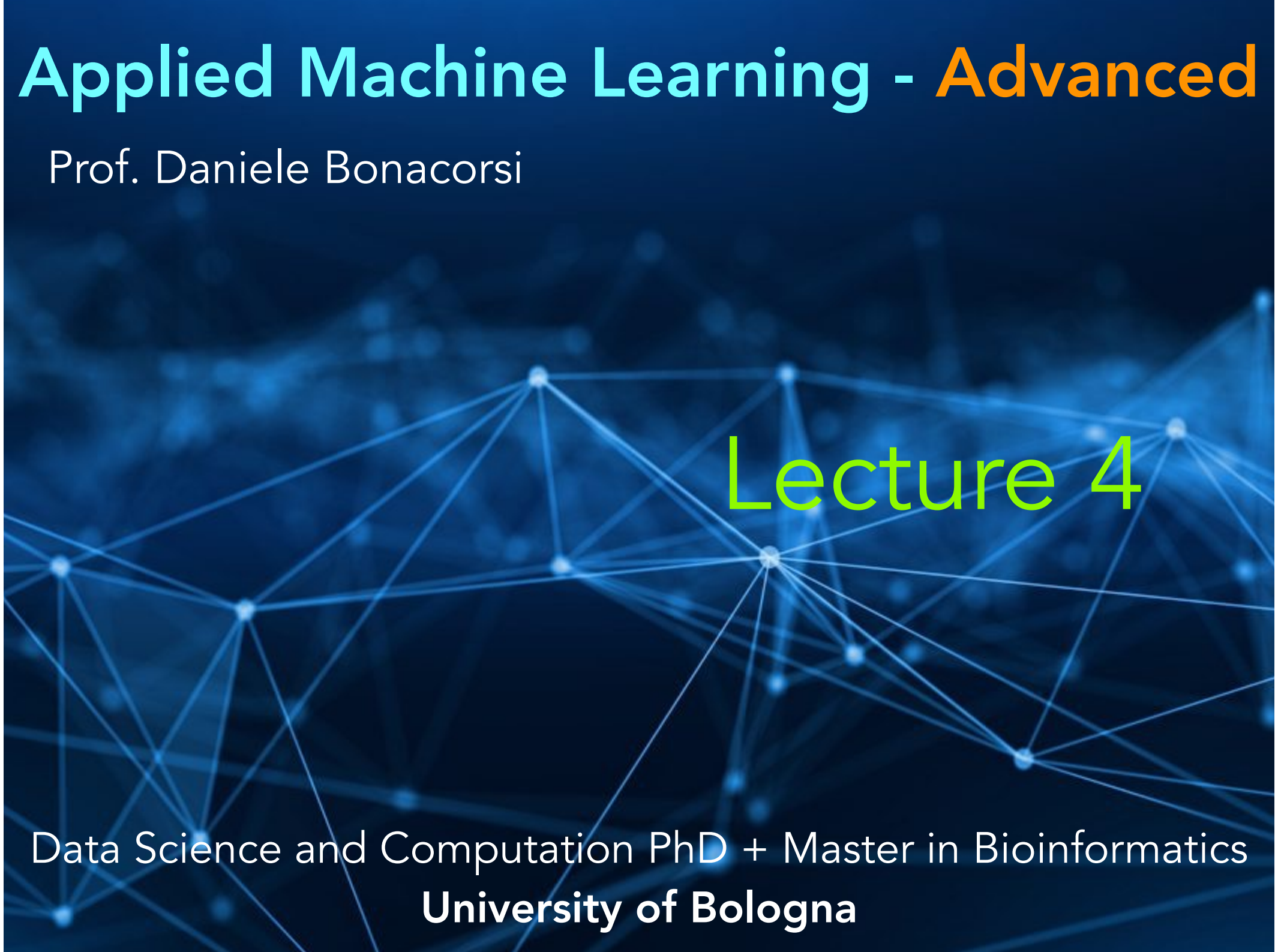
Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi



Lecture 4

Data Science and Computation PhD + Master in Bioinformatics

University of Bologna

Communications

Calendar: updates

AML - Advanced: 4 May - 16 June

- 12 lectures → 9 * 4hrs + 4 * 3 hrs = 48 hrs → **6 CFU**

thu May 4 14-18 → DONE
fri May 5 14-18 → DONE
thu May 11 14-18 → DONE
fri May 12 14-17 → today
thu May 18 14-18 → stay tuned for info!
fri May 19 14-17 → stay tuned for info!
thu May 25 14-18
fri May 26 14-17
thu Jun 1 14-18
thu Jun 8 14-18
thu Jun 9 14-18
Thu Jun 15 14-18
fri Jun 16 14-17

Dates are susceptible to change:
in case, you will be notified in advance.

NNs' building blocks

Tensors and tensor operations

(.. cont'd ..)

Gradient-based optimization

Trainable parameters

Let's restart from here:

```
my_network = keras.models.Sequential([
  keras.layers.Dense(512, activation='relu'),
  keras.layers.Dense(10, activation='softmax')
])
```

```
output = relu(dot(W, input) + b)
```

W and b are **trainable parameters** (or **weights**) of that layer, and they are **attributes** of the layer (namely, the **kernel** and **bias** attributes). They contain the information learned by the NN from exposure to training data

- from the Basic part of the course: the θ parameters..

They are **tensors**. What's in them?

- initially, they are filled with small random values (**random initialization**)
- of course, applying a function to random stuff does not yield any useful representation - it is just a starting point. Weights need to be adjusted, based on some feedback. This gradual adjustment is indeed what we call the **training** (namely, the "learning" part in "machine learning")
  - see next

Training

The training happen in loops. Let's zoom into one loop:

1. Draw a **batch** of training samples x and corresponding targets y
2. Run the NN, i.e. run a **forward pass** on x to obtain predictions y_pred
3. Compute the loss of the NN on this batch (i.e. a measure of the mismatch between y_pred and y)
4. Update all weights of the network in a way that slightly reduces the loss on this batch

When you eventually end up with a NN that has a very low loss on its training data, and you call it done, you judge that the NN has "learned" to map its inputs to correct targets.

Let's revisit steps 1-4 on the basis of what happens behind the scene, and how complex it is.

Training loop evaluation

One training loop:

1. Draw a **batch** of training samples x and corresponding targets y
   - easy - just I/O code
2. Run the NN, i.e. run a **forward pass** on x to obtain predictions y_pred
3. Compute the loss of the NN on this batch (i.e. a measure of the mismatch between y_pred and y)
   - not as easy as 1, but both 2 and 3 are tensors ops - one can implement everything from scratch or use existing (optimized, sophisticated) libraries
4. Update all weights of the network in a way that slightly reduces the loss on this batch
   - this is tough. You need to update the NN's weights, decide if to increase or decrease, and by how much. How?

How to perform weights updates?

I can naively think of freezing all weights in the NN, except the one scalar coefficient being considered → then, try different values for that coefficient

- e.g. assume that the initial value (of one single coefficient) is 0.3. I do a first fwd pass on a batch of data → loss of the NN on that batch is 0.9. If I change the coefficient to 0.35 (0.25) and rerun the fwd pass, the loss increases (decreases) to 1.0 (0.8) → you update it e.g. by -0.05 to contribute to minimising the loss. And you need to repeat this for all coefficients in the NN
- terribly inefficient → 2 (expensive) fwd passes needed per each coefficient (and they are many)

A much better approach is to take advantage of the fact that all ops used in the NN are **differentiable**

- hence compute the **gradient** of the loss with regard to the NN's coefficients → you can then move the coefficients in the opposite direction from the gradient, thus decreasing the loss

Gradient as the derivative of a tensor operation

- i.e. the generalisation of the concept of derivatives to functions of multidimensional inputs (i.e. that take tensors as inputs)

Normal equation vs GD

We discussed it in the Basic part of the course in quite some detail

- we know already that an iterative method is highly desirable!

The strategy is simple:

- we modify the parameters little by little based on the current loss value on a random batch of data
- as we are dealing with a differentiable function, we can compute its gradient
- the gradient gives us an efficient way to implement the difficult step 4 (see previous slides) as per our training loop → we just need to update the weights in the opposite direction from the gradient, and the loss will be a little less every time

Let's hence modify our initial training loop

- see next

SGD

The "new" training loop:

1. Draw a **batch** of training samples x and corresponding targets y
2. Run the NN, i.e. run a **forward pass** on x to obtain predictions y_pred
3. Compute the loss of the NN on this batch (i.e. a measure of the mismatch between y_pred and y)
4. Compute the gradient of the loss with regard to the network's parameters (a ***backward pass***).
5. Move the parameters a little in the opposite direction from the gradient thus reducing bit a the loss on the batch

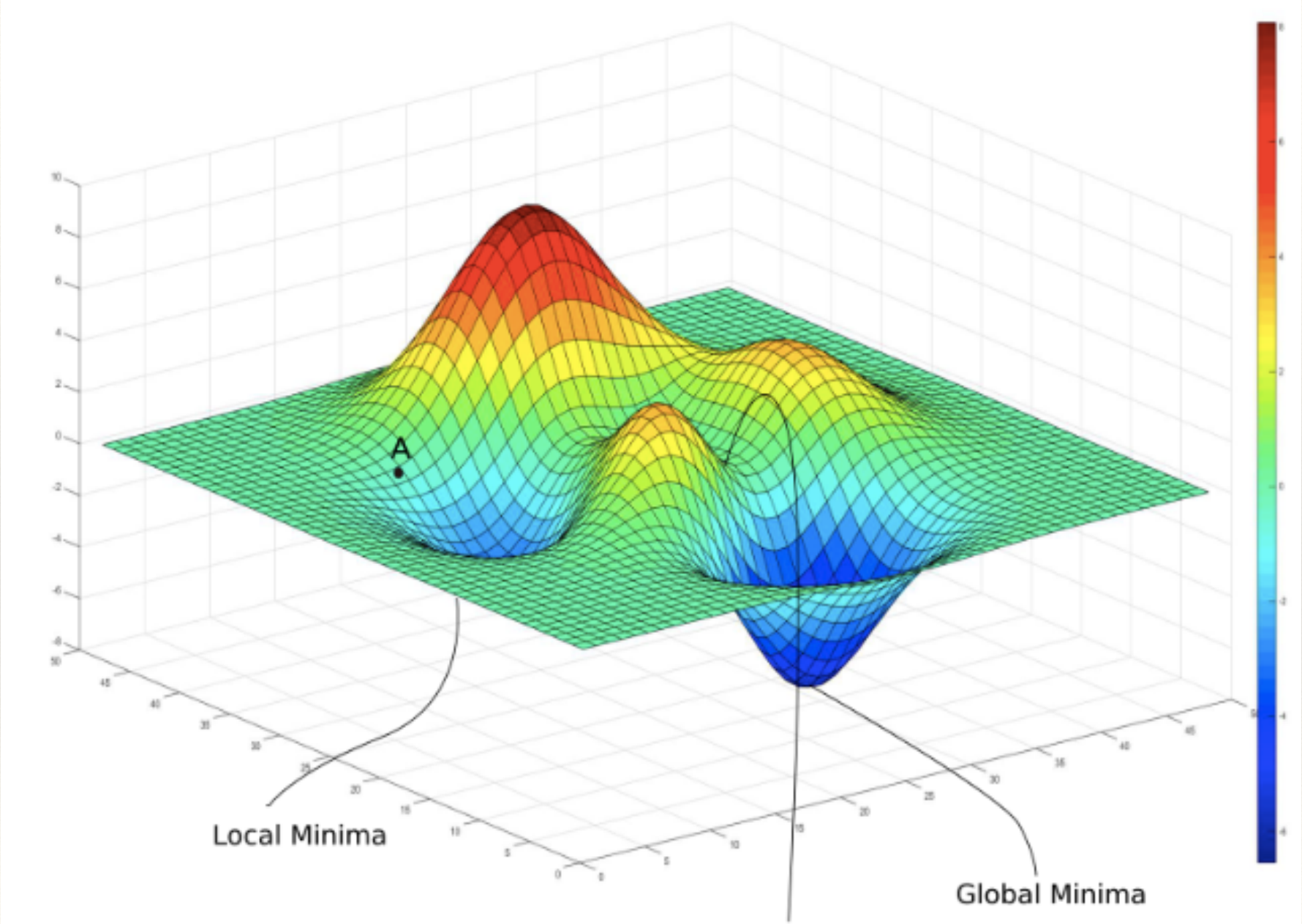
This is **mini-batch stochastic gradient descent** (**mini-batch SGD**)

- "stochastic" due to the random draw of samples to form each batch

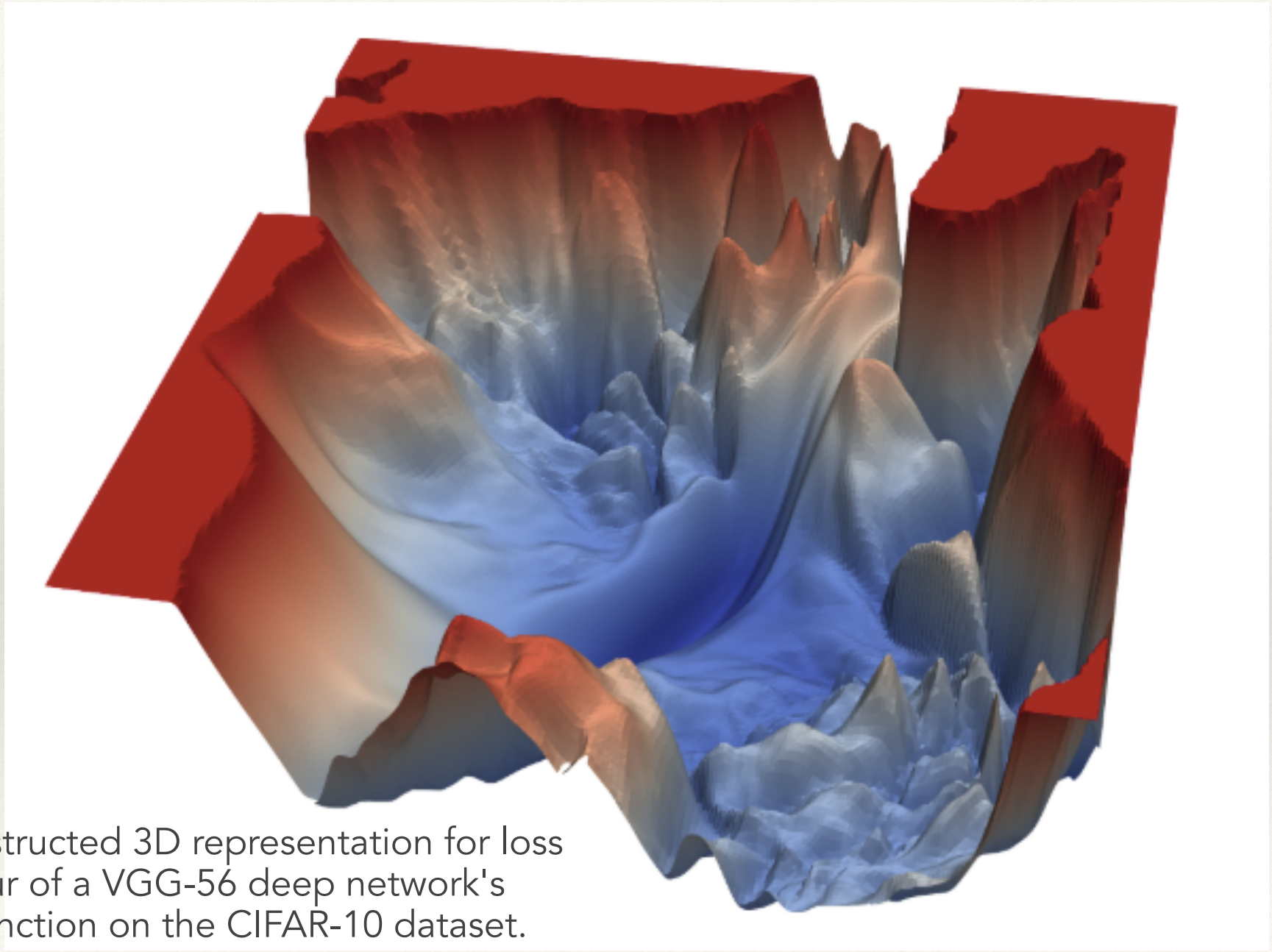
A good compromise among two extremes:

- "true SGD": no batch → perform training loop on every training instance
- "batch SGD": one single batch → perform training loop on all training set every time

Complexity of a loss curve? low here…



Complexity of a loss landscape? **high** here…



A constructed 3D representation for loss contour of a VGG-56 deep network's loss function on the CIFAR-10 dataset.

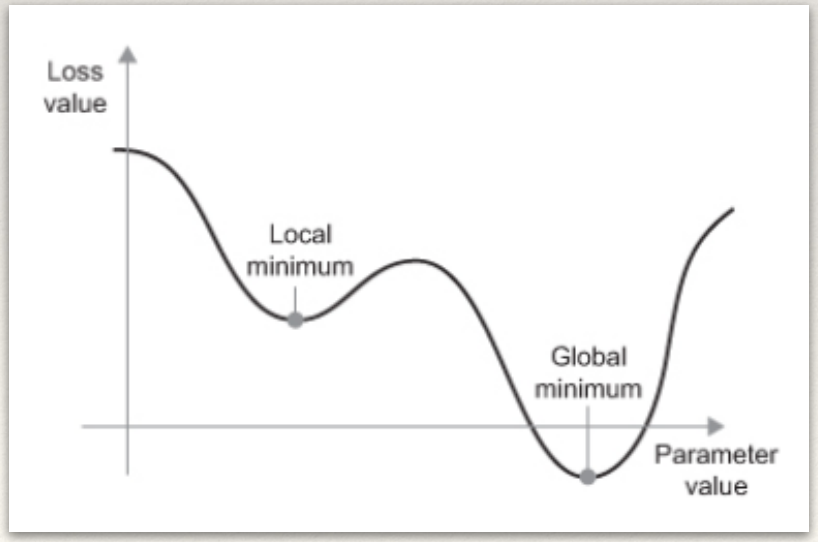
Optimisers

Multiple variants of SGD exist, and they are known as **optimisers** (or **optimisation methods**)

- e.g. **SGD with momentum**, as well as **Adagrad**, **RMSProp**, and others
- they differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients

The concept of momentum (**derived from physics**) and its application here is very interesting

- it addresses two SGD issues: convergence speed and local minima
- momentum is implemented by moving at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration)




```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

(naive implementation of SGD with momentum)

Chaining derivatives: the **backpropagation** algo

In the training loop discussed so far, we can casually think of a "NN function" that is differentiable → we can compute its derivatives

In practice, a NN function consists of *many* tensor operations chained together, each of which has a simple, known derivative

- e.g. f(w1, w2, w3) = a(w1, b(w2, c(w3))) is a NN function f composed by 3 tensor ops a, b and c acting on weight tensors w1, w2 and w2
- to derive f above, I would apply the chain rule (f(g(x)) = f'(g(x)) * g'(x))

Applying the chain rule to the computation of the gradient values of a NN gives rise to an algo called **backpropagation** (or **reverse-mode differentiation**).

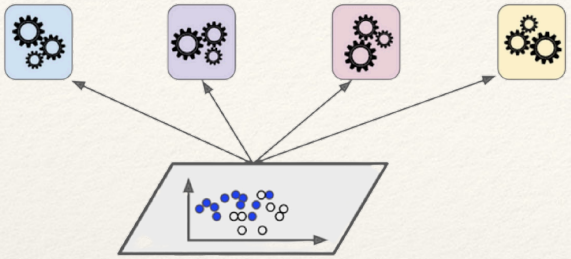
- it starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

A note

Nowadays, and in the foreseeable future, ML researchers will implement NNs in modern frameworks that are capable of ***symbolic differentiation*** (such as TensorFlow)

- given a chain of operations with a known derivative, they can compute a gradient *function* for the chain (by applying the chain rule) that maps network parameter values to gradient values
- when you have access to such a function, the backward pass is reduced to a call to this gradient function - no need to re-implement the backpropagation algo by hand

No deep dive on backprop in this Applied ML course, as most important is just a good intuition of how gradient-based optimization works.



Ensemble Learning

"Wisdom of the crowd" (more later..)

Aggregate and average over answers from a pool of experts usually is wiser than taking the answer by just one expert

This idea in ML gives this tactic:

- run a group of predictors (such as classifiers or regressors)
- aggregate their predictions

A group of predictors is called an "**ensemble**" → these techniques are referred to as "**Ensemble Learning**" (Ensemble algos/methods)

We will discuss the most popular ensemble methods, including:

- **voting**
- **bagging**
- **boosting**
- **stacking**

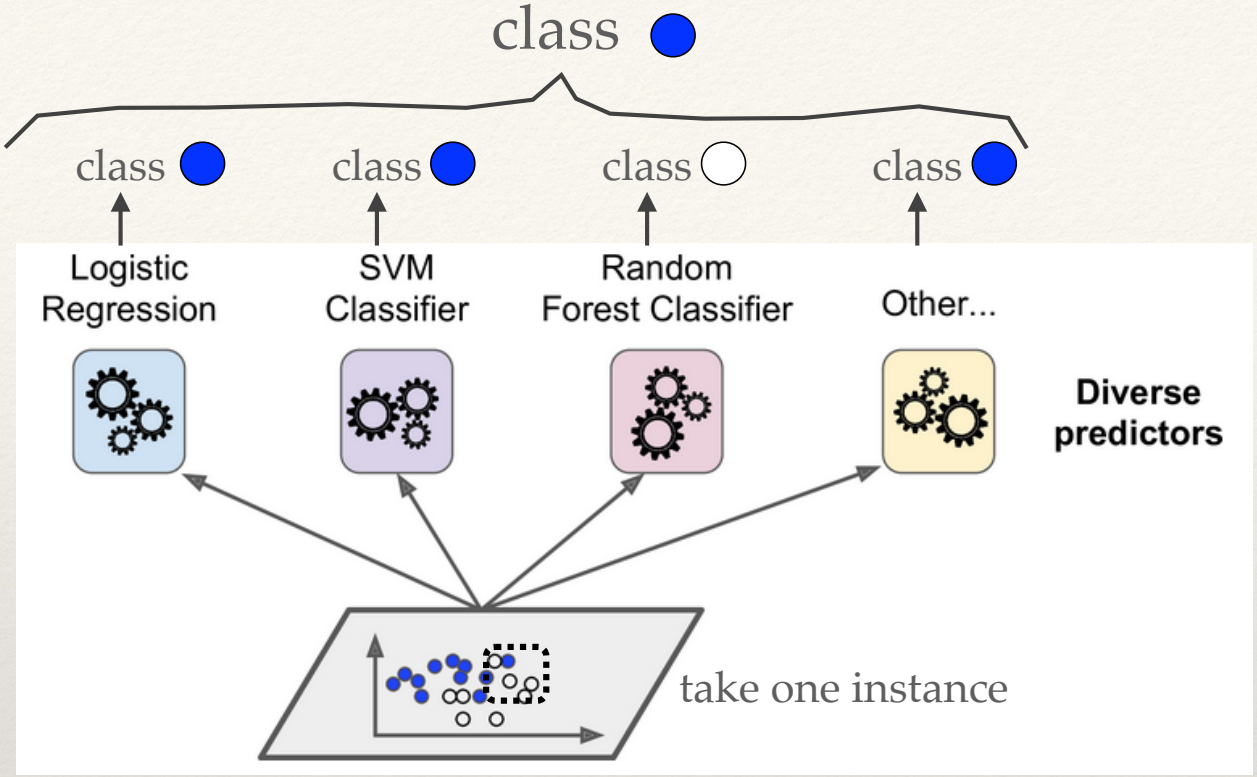
Ensemble :: Voting

Ensemble :: Bagging

Ensemble :: Boosting

Ensemble :: Stacking

(Hard) **Voting** classifiers



I trained various classifiers. Each one achieving its own (goodish) accuracy.

An even better classifier can be built by aggregating the predictions from each classifier and then predict, *per each instance*, the class that gets the most votes out of the ensemble of classifiers

- This "majority-vote" classifier is called a "**hard voting classifier**"

Wisdom of the crowd: wiser than the wisest?!

A voting classifier often achieves higher accuracy than the best classifier in the ensemble (!)

- even if each classifier is a weak learner (i.e. only slightly better than random guessing), the ensemble of classifiers can still be a strong learner (i.e. achieving high accuracy), provided 1) there are a sufficient number of weak learners and 2) they are sufficiently diverse.

Wait, what?! How is that possible?!

- Basically, law of large numbers.
- Suppose I build an ensemble containing 10^3 classifiers that are relatively poor, i.e. individually correct only 51% of the time (barely better than random guessing). If I predict the majority voted class, I can hope for something up to 75% accuracy!
- However, this is true if all classifiers are perfectly independent, make uncorrelated errors, etc (impossible, they are trained on the same data..). Best is to put together **diverse classifiers trained using very different algos**, i.e. higher chances they will make **very different types of errors**, thus **improving the ensemble's accuracy**.

An implementation in sklearn

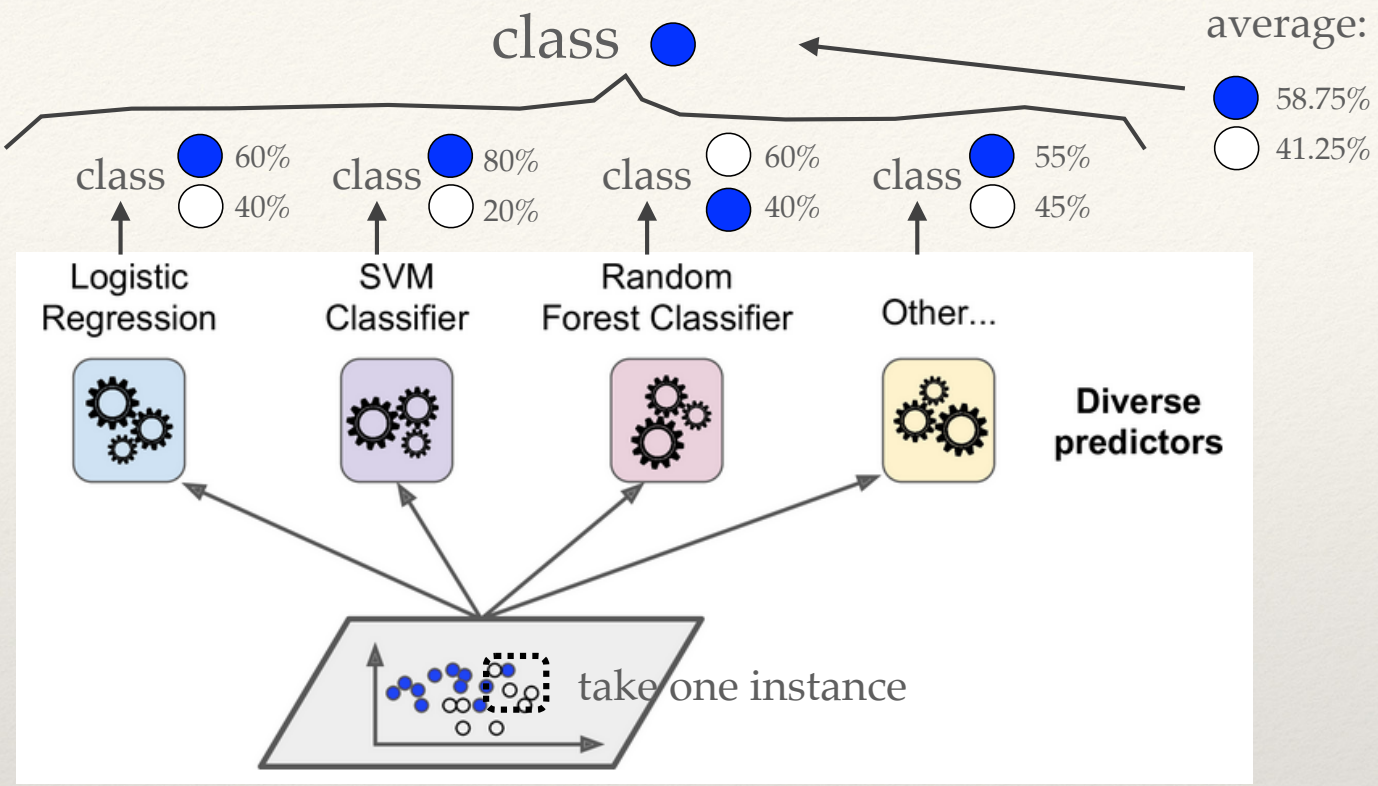
Check the code carefully..

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

The "hard" voting classifier slightly outperforms all the individual classifiers.

(Soft) **Voting** classifiers



If (and only if) all classifiers you select for your ensemble are able to estimate class probabilities (i.e. in sklearn they have a predict_proba() method), then you can avoid a hard voting, and instead use sklearn to **predict the class with the highest class probability, averaged over all the individual classifiers**.

This is called "**soft voting**".

- It gives more weight to highly confident votes

A (modified) implementation in sklearn

Ensure that all classifiers can estimate class probabilities

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

'soft'

SVC class can't by default, one needs to set its probability hyperparameter to 'True' (which will add a predict_proba() method at the cost of making SVC use cross-validation to estimate class probabilities, slowing down training..)

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

this becomes roughly 91%

The "soft" voting classifier would achieve over 91% accuracy

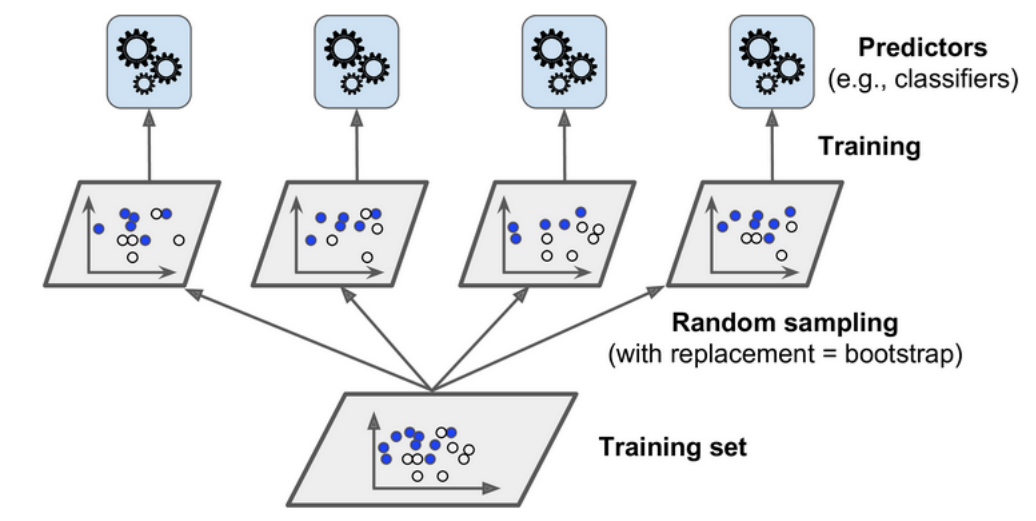
Ensemble :: Voting

Ensemble :: Bagging

Ensemble :: Boosting

Ensemble :: Stacking

Bagging (and **Pasting**)



Instead of using different training algos to get different predictors in the ensemble, just use the same training algo for every predictor, but **train them on different random subsets of the training set**.

- random sampling performed w replacement (use instances more than once) → **Bagging** (**B**ootstrap **AGG**regation → *"bootstrap=True" in the sklearn implementation*)
- random sampling performed w/o replacement (use instances only once) → **Pasting**

Aggregation of predictions from all predictors is typically the most frequent (hard voting) for classification, or the average for regression

Note: they can be trained in parallel (multiple CPUs or different servers, even), one of the reasons why they are popular: ***they scale up well****.*

Example in sklearn

BaggingClassifier automatically performs soft voting if the base classifier can estimate class probabilities

`BaggingRegressor` for regression

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

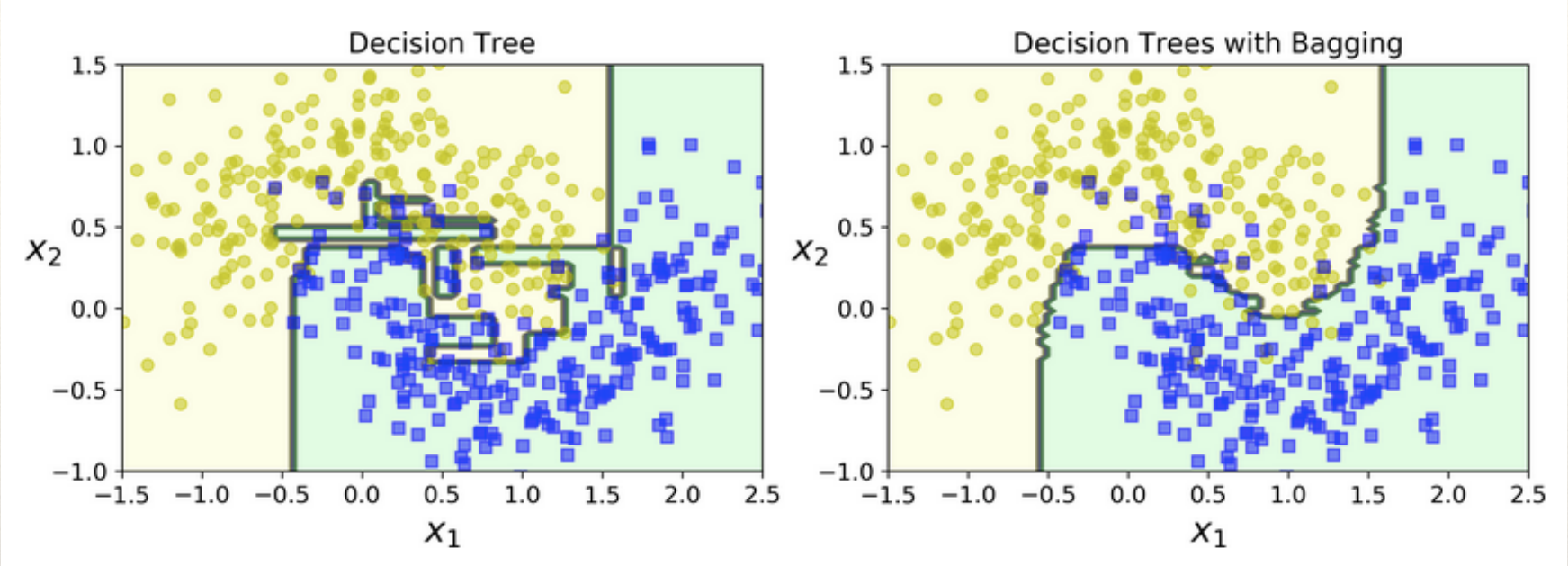
This is **Bagging**. For **Pasting**, put 'False'

This trains an ensemble of 500 DT classifiers, each trained on 100 training instances randomly sampled from the original training set with replacement

- n_jobs tells sklearn the nb CPU cores to use for training and predictions (–1 = use all available cores)

1 DT vs a bagging ensemble of 500 DT

[A]



[B]

[B]'s predictions will likely **generalize much better** than [A]

- i.e. comparable bias (roughly same nb errors on the training set)..
- .. but smaller variance (decision boundary less irregular)

This is quite general: the net result is that the ensemble has a **similar bias** but a **lower variance** than a single predictor trained on the original training set

Out-of-bag (**oob**) evaluation in bagging

With bagging, for any given predictor, some instances may be sampled several times, while others may not be sampled at all

So, **only ~63% of the m training instances are sampled on average for each predictor**

- On m training instances, bagging (i.e. w/ replacement) gives a chance for an instance of not being selected in any of draws as $(1-1/n)^n$, which for large n yields $1/e \approx 0.37$
- these **remaining 37%** are called "**out-of-bag**" (**oob**) training
  - Note that they are not the same 37% for all predictors

As a predictor never sees the oob instances during training, **it can be evaluated on these instances, without the need for a separate validation set**.

- e.g. by adding oob_score=True in my BaggingClassifier in sklearn, I can perform an oob evaluation of the ensemble's accuracy on that test set (averages out the oob evaluations of each predictor)

Random Forests

A **Random Forest** is an ensemble of DTs, generally trained via the bagging (sometimes pasting) method

- in sklearn, does this mean to build a BaggingClassifier and pass it a DecisionTreeClassifier? Yes but..
- .. one can instead use the RandomForestClassifier class (similarly: RandomForestRegressor), which is more convenient and optimised for DTs

→ Random Forest algo introduces **extra randomness when growing trees**

- instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features
- → greater tree diversity, which trades a higher bias for a lower variance, generally yielding an overall better model

For even additional randomness → Extremely Randomised Trees ensembles (**Extra Trees**)

- add randomness by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular DTs do)
- even faster than Random Forests

Random Forests and feature importance

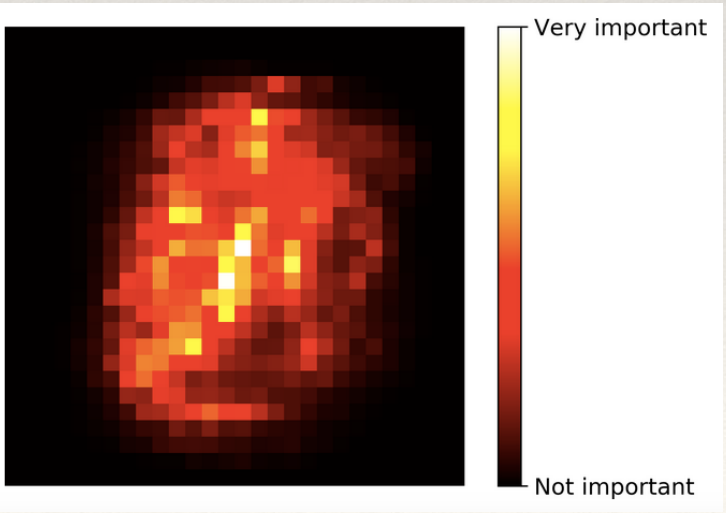
Another quality of Random Forests is that they make it easy to measure **the relative importance of each feature**

- e.g. sklearn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest)
- very useful in feature selection!

Feature importance in the IRIS dataset

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Pixel importance in the MNIST dataset



Ensemble :: Voting

Ensemble :: Bagging

Ensemble :: Boosting

Ensemble :: Stacking

Boosting

Boosting (originally called "hypothesis boosting") refers to any ensemble method that can combine several weak learners into a strong learner, with the general idea to **train predictors sequentially, each trying to correct its predecessor**.

There are many boosting methods available, with the by-far most popular being:

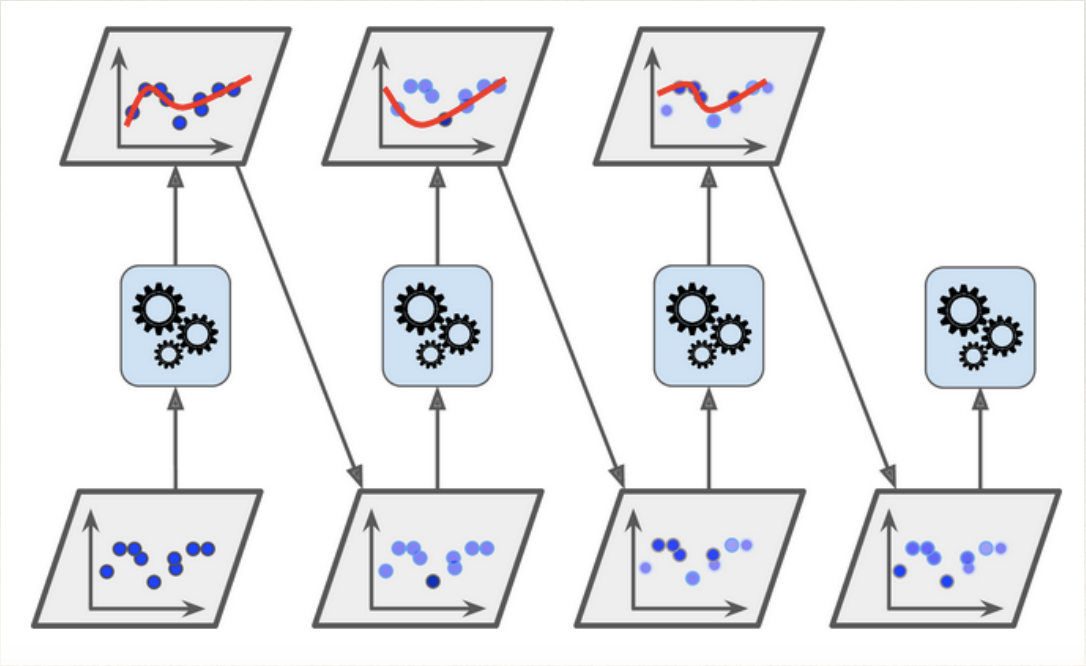
- **AdaBoost** (short for Adaptive Boosting)
- **Gradient Boosting**

AdaBoost

keyword: "adaptive"

AdaBoost is a predictor that corrects/improves its predecessor by paying a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases.

- E.g. a first base classifier (such as a DT) is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, then weights are updated, and so on



Gradient Boosting

keyword: “gradient”

Popular.

Like AdaBoost, **Gradient Boosting** works by sequentially adding predictors to an ensemble, each one correcting its predecessor..

.. but ..

while AdaBoost tweaks the instance weights at every iteration - Gradient Boosting tries to **fit the new predictor to the residual errors made by the previous predictor**

- also called Gradient Tree Boosting, or Gradient Boosted Regression Trees (GBRT)

Boosting: pros and cons

Push to **high perf and high generalisation levels**, but, in general, one drawback: it is a sequential learning technique, so **it cannot be parallelised** (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, **it does not scale as well as bagging or pasting**.

Gradient boosting logic in sklearn (by code)

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

First, fit a DecisionTreeRegressor to the training set (e.g., a noisy quadratic)..

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

.. then train a second regressor on the residual errors made by the first predictor..

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

.. then train a third regressor on the residual errors made by the second predictor..

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

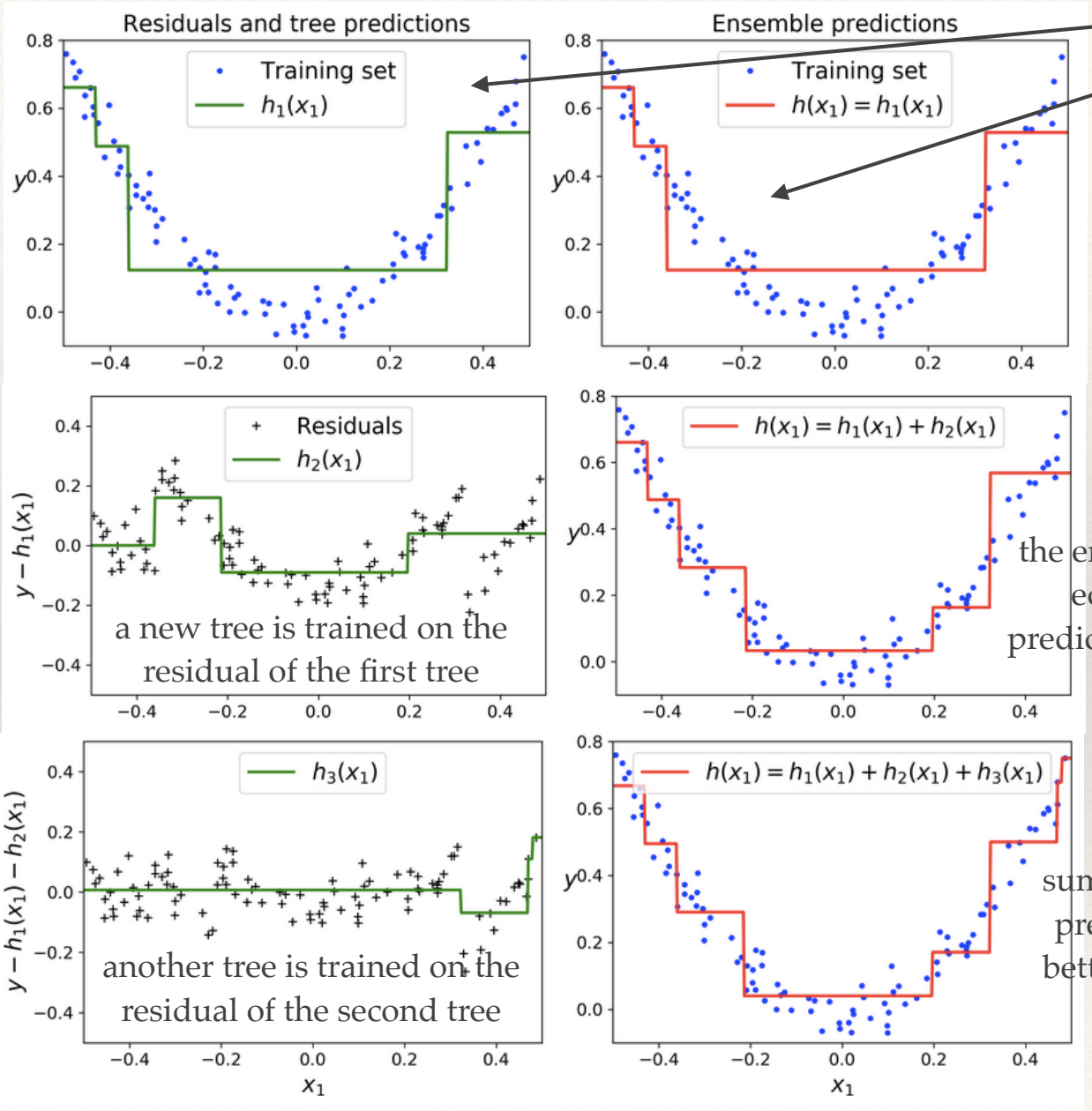
Now we have an ensemble containing 3 trees. It can make predictions on a new instance simply by adding up the predictions of all the trees

Gradient Boosting (by visualisation)

Prediction of tree 1

Prediction of tree 2

Prediction of tree 3



ensemble has still just 1 tree, so predictions are the same

Ensemble predictions over the sequence

a new tree is trained on the residual of the first tree

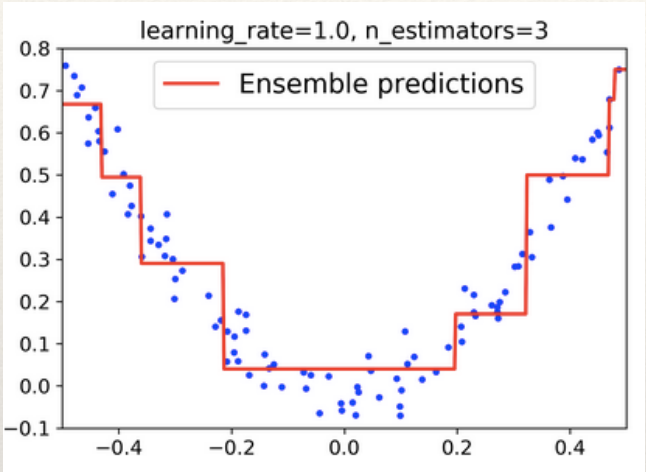
the ensemble's predictions are equal to the sum of the predictions of the first two trees

another tree is trained on the residual of the second tree

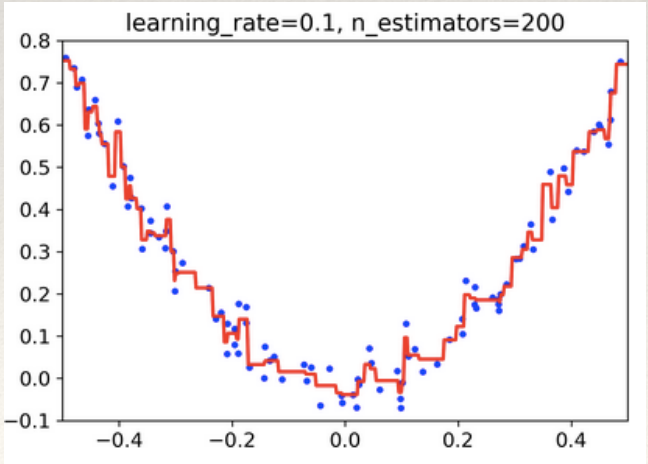
sum again. The ensemble's predictions gradually get better as trees are added to the ensemble.

Gradient Boosting tuning

Play with the learning_rate hyperparameter: it scales the contribution of each tree. If you set it to a low value (such as 0.1 in this example), you will need more trees (n_estimators) in the ensemble to fit the training set, but the predictions will usually generalise better.

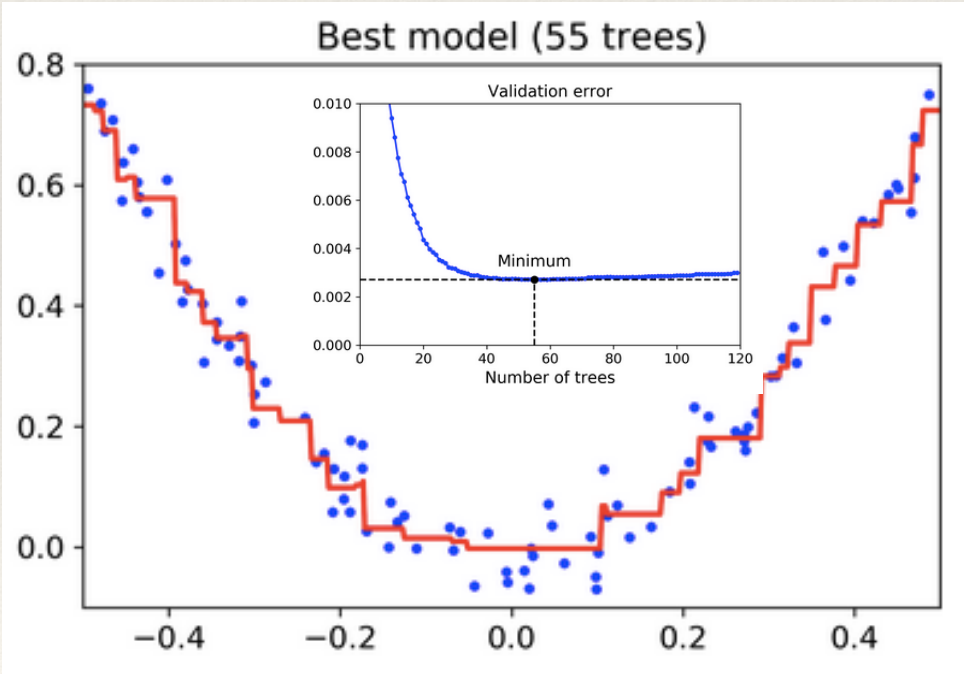


Best so far: not enough trees to fit the training set



Reduced learning rate: too many trees and overfits the training set

I can use early-stopping in sklearn: I train, while measuring the validation error at each stage of training in order to find the optimal nb trees, and once found I train another Gradient Boosting ensemble using the optimal nb trees



Extreme Gradient Boosting (**XGBoost**)

Very popular.

An optimized implementation of Gradient Boosting is available in the popular python library **XGBoost**, i.e. **Extreme Gradient Boosting**

- designed to be fast, scalable and portable

XGBoost's API is quite similar to sklearn's:

```
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

Some details in next slide

- keep an eye on it, as it often a key player in winning entries in ML competitions.. but full coverage would take a LOT of time!

Extreme Gradient Boosting (**XGBoost**)

XGBoost is one of the fastest implementations of gradient boosted trees. How does it work?

It directly attacks one of their major inefficiencies. It considers the potential loss for all possible splits to create a new branch (especially useful in case of thousands of features, therefore thousands of possible splits), and tackles this inefficiency by looking at the distribution of features across all data points in a leaf and using this information to reduce the search space of possible feature splits.

Yes, it uses a few regularisation tricks, but the achievable speed up is by far the most useful aspect of the XGBoost library: it allows many hyperparameter settings to be investigated quickly

- very helpful, as they are so many.. and nearly all of them are designed to limit overfitting (no matter how simple your base models are, if you put thousands of them together they will overfit..)

The list of hyperparameters is intimidating.. Most important ones:

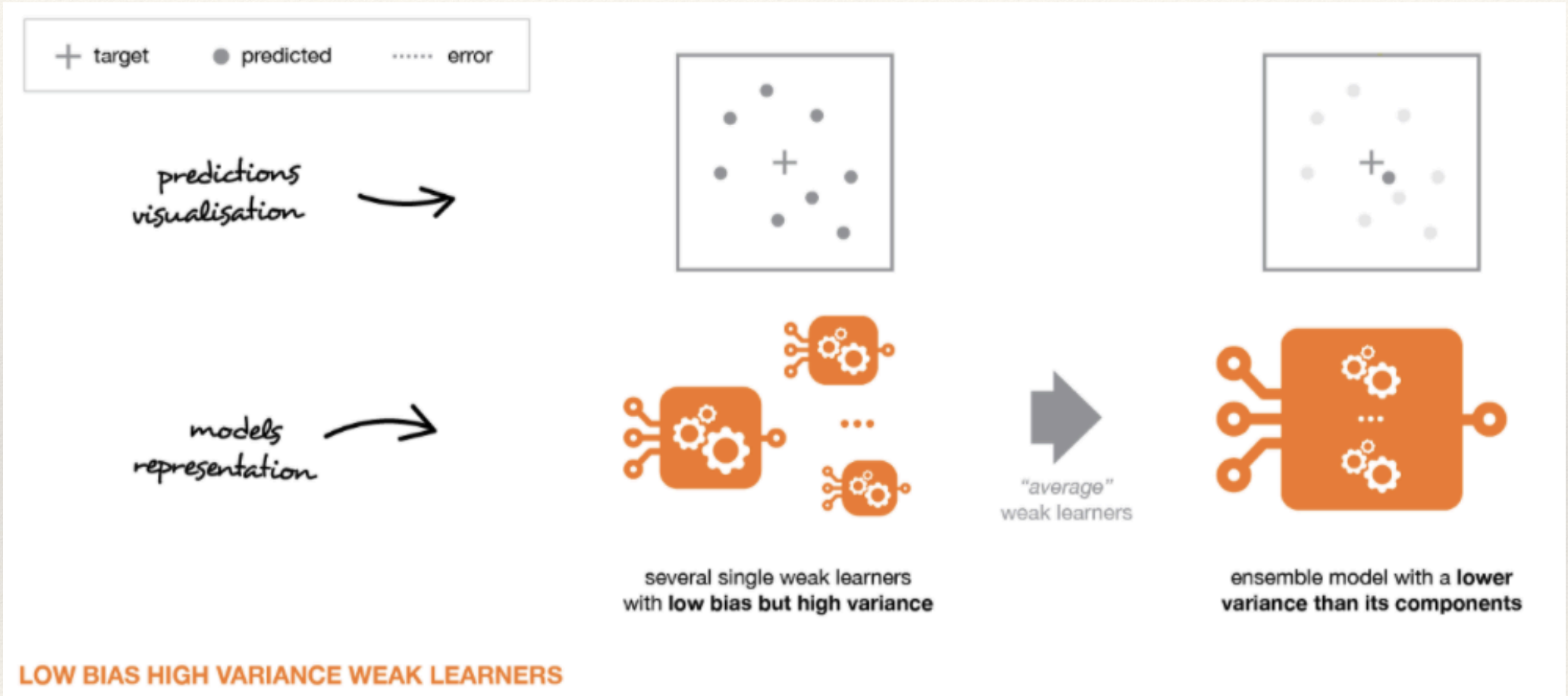
- n_estimators (and early stopping) - i.e. how many subtrees will be trained
- max_depth - i.e. the maximum tree depth each individual tree can grow to
- learning rate
- reg_alpha and reg_lambda - i.e. control the L1 and L2 regularisation terms

Limit them, otherwise any grid / random search with all of the hyperparameters XGBoost allows you to tune would quickly make the search space explode. Start with few, and then dive into the others only if you still have trouble with overfitting.

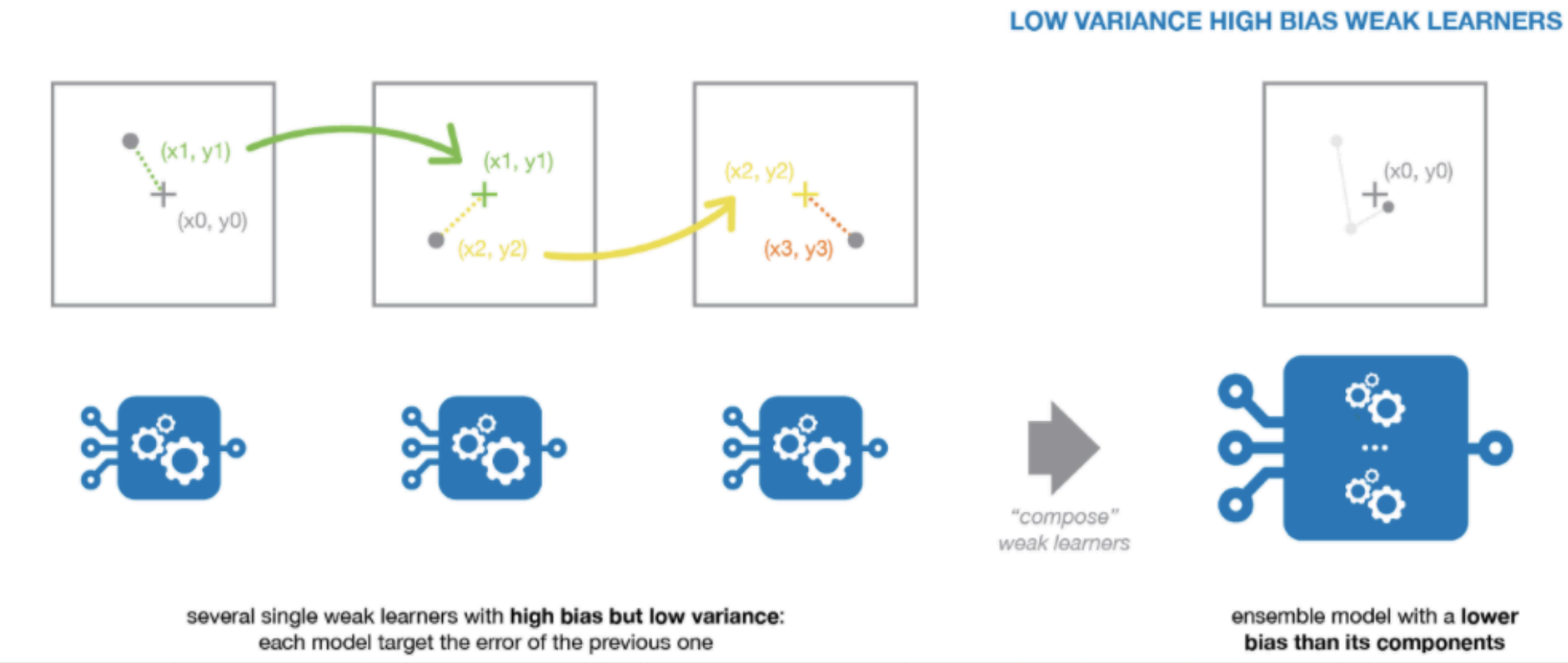
A visual summary (bagging vs boosting)

How many base, weak learners?

How to deal with weak learners? **Low-bias, high-variance**

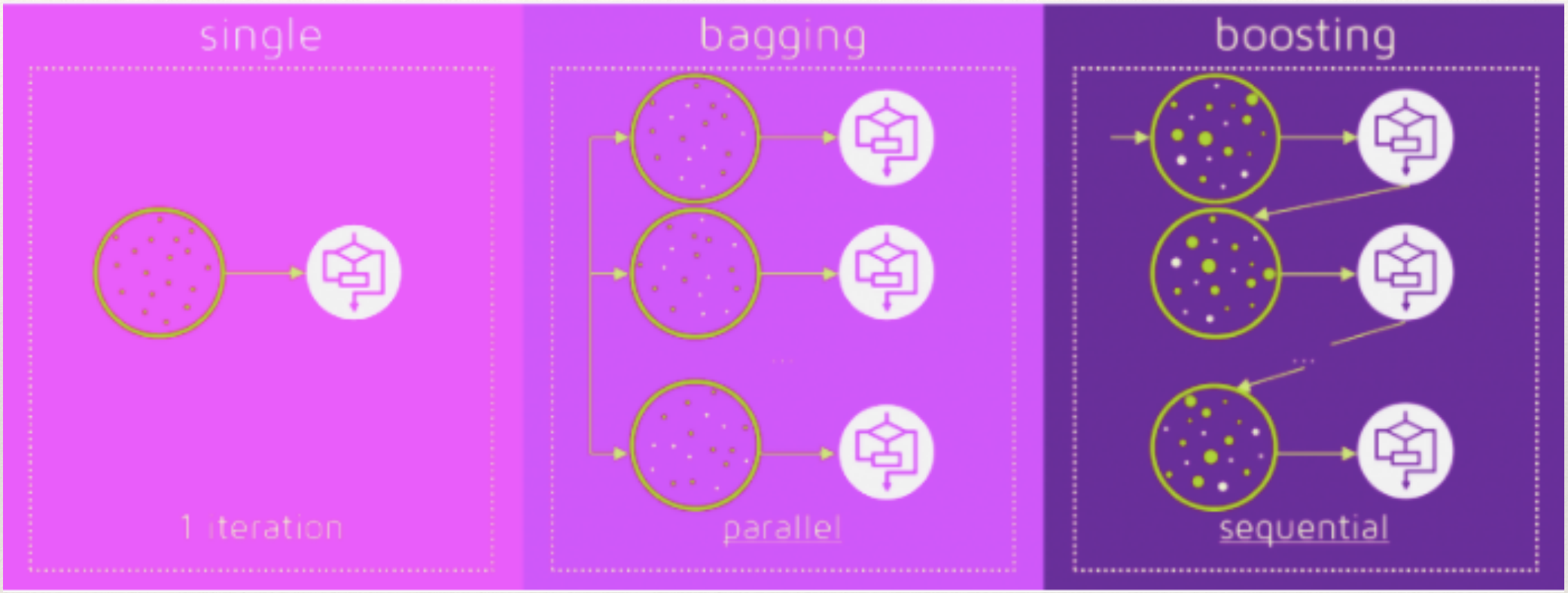


How to deal with weak learners? **Low-variance, high-bias**

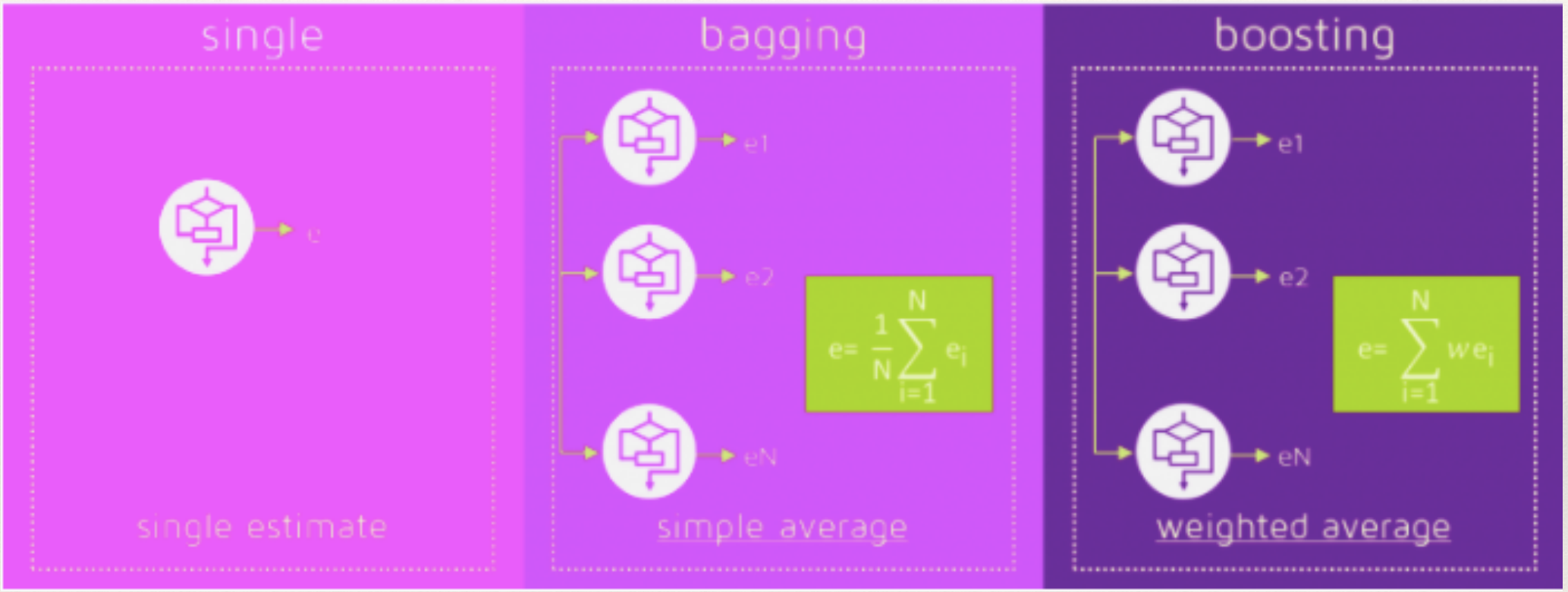


How do you use the training dataset?

How do you do the training?



How do you perform the classification?



Ensemble :: Voting

Ensemble :: Bagging

Ensemble :: Boosting

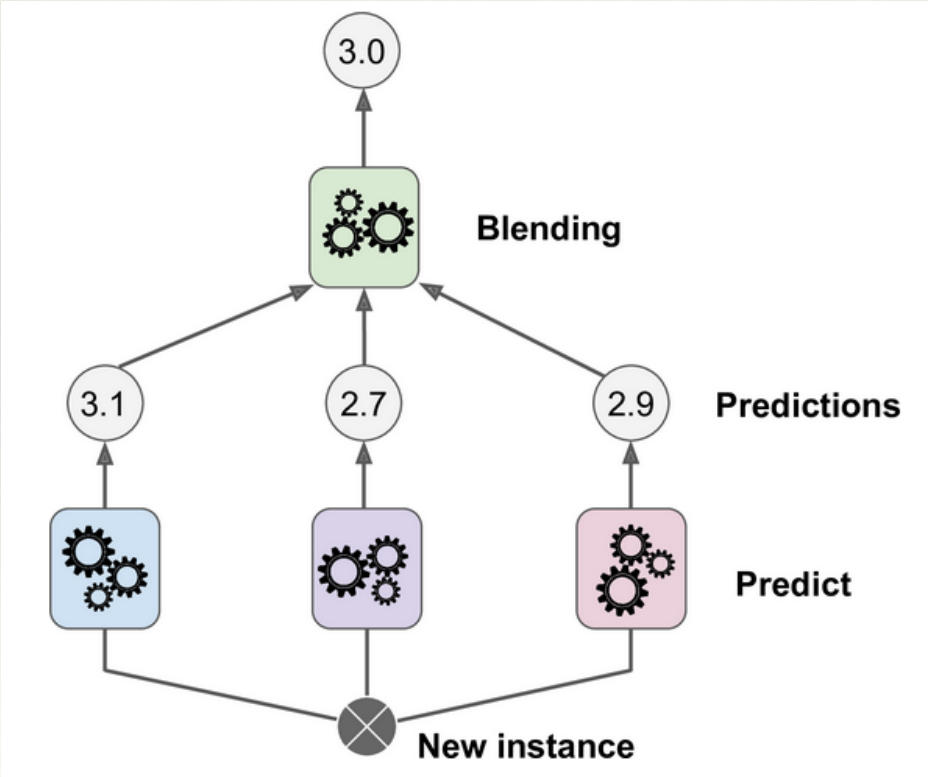
Ensemble :: Stacking

Stacking

Stacking (short for "stacked generalisation") is another Ensemble method

Idea: aggregate predictions from predictors in the ensemble not using trivial functions (such as hard voting) but **by training a model to perform such aggregation**.

- The additional intelligence lies in a final predictor (**blender**, or **meta-learner**) whose task is to take intermediate predictions as inputs and to make the final prediction



Example: **one layer**
(but you can have more)

Stacking (in words)

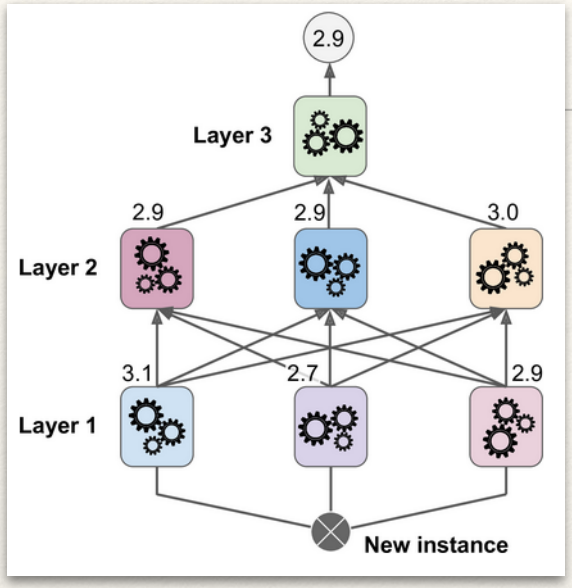
The trick is to split the training set into subsets:

- the first one is used to train the first layer
- the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer)
- the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer)

Once this is done, we can make a prediction for a new instance by going through each layer sequentially.

Clear? No? Let's go visual!

Stacking (in pictures) → start from the bottom..

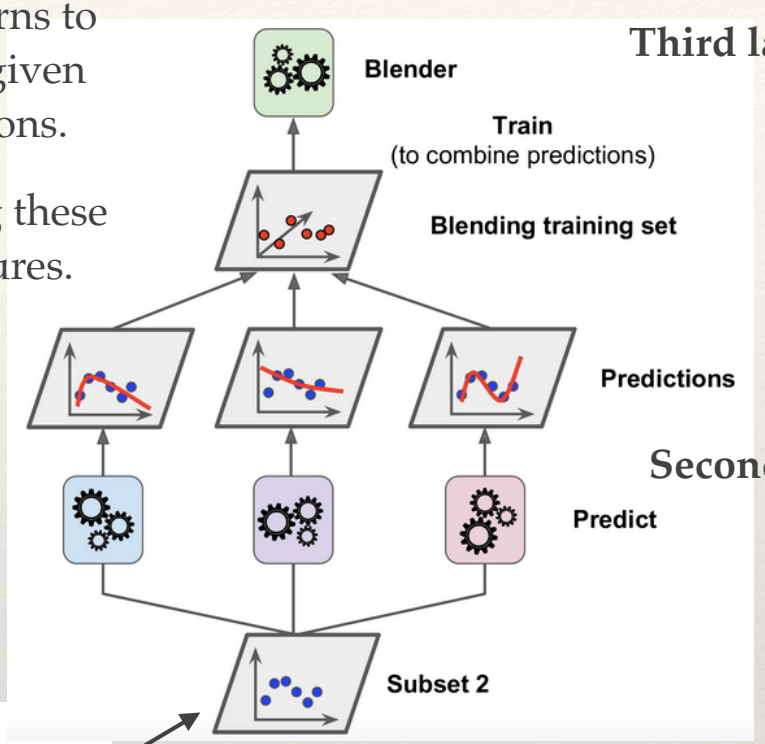


The **blender** is trained on this new training set, so it learns to predict the target value given the first layer's predictions.

I create **a new training set** using these predicted values as input features.

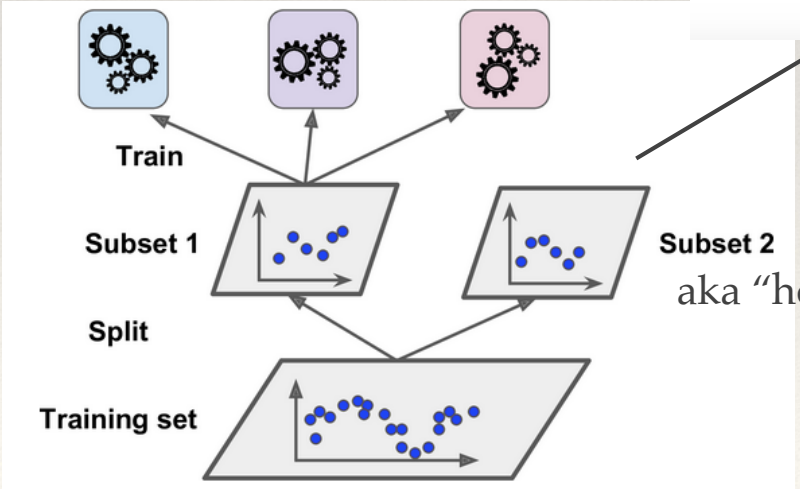
For each instance in Subset 2, I get 3 predicted values

First layer predictors are used to make predictions on Subset 2 (held-out, never seen, so predictions are "clean")



First layer

Subset 1 is used to **train** the predictors in the first layer



aka "hold-out set"

Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Hands-on: Ensemble learning

Data Science and Computation PhD + Master in Bioinformatics
University of Bologna

Hands-on:

Ensemble learning

[credits to: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

Coding material

AML2223Adv_Ensemble.ipynb

→ gym on the notebook