MACHINE LEARNING MODELS

XGBOOST

Gradient Boosting

Definition

*Gradient boostin*g is one of the most powerful techniques for building predictive models. It is an automatic learning technique used for regression analysis and statistical classification problems, which produces a predictive model in the form of a set of weak predictive models, typically decision trees. It builds the model in a staggered manner as other boosting methods do, and generalizes them allowing arbitrary optimization of a differentiable loss function.

This algorithm is an efficient algorithm for converting relatively poor hypotheses into very good hypotheses. A **weak hypothesis** or weak learner is defined as one whose performance is at least slightly better than random chance. *Hypothesis boosting* was the idea of filtering observations, leaving those observations that the weak learner can handle and focusing on developing new weak learns to handle the remaining difficult observations. The idea is to use the weak learning method several times to get a succession of hypotheses, each one refocused on the examples that the previous ones found difficult and misclassified.

Gradient boosting involves three elements:

1. A **loss function** to be optimized.
2. A **weak learner** to make predictions.
3. An **additive model** to add weak learners to minimize the loss function.

Loss Function

A **loss function** or **cost function** is a function that maps an event or values of
one or more variables onto a real number intuitively representing some *"cost"* associated with the event. An optimization problem seeks to minimize a loss function.

The *loss function* used depends on the type of problem being solved.

It must be *differentiable* (a **differentiable function** of one real variable is a function whose derivative exists at each point in its domain), but many standard loss functions are supported and you can define your own. For example, regression may use a squared error and classification may use logarithmic loss.

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

Weak Learner

Decision trees are used as the weak learner in gradient boosting. Specifically regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and "correct" the residuals in the predictions.

Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss. Initially, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally with 4-to-8 levels. It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes. This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

Additive Model

Trees are added one at a time, and existing trees in the model are not changed. A gradient descent procedure is used to minimize the loss when adding trees. Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.
Instead of parameters, we have **weak learner sub-models** or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must **add a tree to the model** that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by reducing the residual loss.

Generally this approach is called **functional gradient descent** or **gradient descent with functions**. One way to produce a weighted combination of classifiers which optimizes the cost is by gradient descent in function space.

A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.

Improvements to Basic Gradient Boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting. In this this section we will look at 4 enhancements to basic gradient boosting:

1. *Tree Constraints*
2. *Shrinkage*
3. *Random sampling*
4. *Penalized Learning*

Tree Constraints

It is important that the weak learners have skill but remain weak. There are a number of ways that the trees can be constrained. A good general heuristic is that the more constrained tree creation is, the more trees you will need in the model, and the reverse, where less constrained individual trees, the fewer trees that will be required.
Below are some constraints that can be imposed on the construction of decision trees:

- **Number of trees**, generally adding more trees to the model can be very slow to overfit. The advice is to keep adding trees until no further improvement is observed.
- **Tree depth**, deeper trees are more complex trees and shorter trees are preferred. Generally, better results are seen with 4-8 levels.
- **Number of nodes or number of leaves**, like depth, this can constrain the size of the tree, but is not constrained to a symmetrical structure if other constraints are used.
- **Number of observations per split** imposes a minimum constraint on the amount of training data at a training node before a split can be considered
- **Minimum improvement to loss** is a constraint on the improvement of any split added to a tree.

Weighted Updates

The predictions of each tree are added together sequentially. The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a **shrinkage** or a **learning rate**. Each update is simply scaled by the value of the “learning rate parameter v”.

The effect is that learning is slowed down, in turn require more trees to be added to the model, in turn taking longer to train, providing a configuration trade-off between the number of trees and learning rate. Decreasing the value of v [the learning rate] increases the best value for M [the number of trees]. It is common to have small values in the range of 0.1 to 0.3, as well as values less than 0.1.

Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model.

Stochastic Gradient Boosting

A big insight into bagging ensembles and random forest was allowing trees to be greedily created from subsamples of the training dataset. This same benefit can be used to reduce the correlation between the trees in the sequence in gradient boosting models. This variation of boosting is called **stochastic gradient boosting**.

At each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. The randomly selected subsample is then used, instead of the full sample, to fit the base learner.

A few variants of stochastic boosting that can be used:

- Subsample rows before creating each tree.
- Subsample columns before creating each tree
- Subsample columns before considering each split.

Generally, aggressive sub-sampling such as selecting only 50% of the data has shown to be beneficial.

Penalized Gradient Boosting

Additional constraints can be imposed on the parameterized trees in addition to their structure. Classical decision trees like CART are not used as weak learners, instead a modified form called a regression tree is used that has numeric values in the leaf nodes (also called terminal nodes). The values in the leaves of the trees can be called weights in some literature.

As such, the leaf weight values of the trees can be regularized using popular regularization functions, such as:

- L1 regularization of weights.
- L2 regularization of weights.

(See functions here)

The additional regularization term helps to smooth the final learnt weights to avoid overfitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions.

Definition

XGBoost stands for e**X**treme **G**radient **B**oosting. XGBoost is an implementation of ***gradient boosted decision trees*** designed for speed and performance that is dominative competitive machine learning. The library is laser focused on computational speed and model performance, as such there are few frills. Nevertheless, it does offer a number of advanced features.

The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

- **Gradient Boosting:** algorithm also called gradient boosting machine including the learning rate.
- **Stochastic Gradient Boosting:** with sub-sampling at the row, column and column per split levels.
- **Regularized Gradient Boosting:** with both L1 and L2 regularization.

The library provides a system for use in a range of computing environments, not least:

- **Parallelization** of tree construction using all of your CPU cores during training.
- **Distributed Computing** for training very large models using a cluster of machines.
- **Out-of-Core Computing** for very large datasets that don't fit into memory.
- **Cache Optimization** of data structures and algorithm to make best use of hardware.

The implementation of the algorithm was engineered for efficiency of compute time and memory resources. A design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

- **Sparse Aware** implementation with automatic handling of missing data values.
- **Block Structure** to support the parallelization of tree construction.
- **Continued Training** so that you can further boost an already fitted model on new data.

Why Use XGBoost?

The two reasons to use XGBoost are also the two goals of the project:

1. *Execution Speed.*
2. *Model Performance.*

XGBoost Execution Speed

Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting.

XGBoost Model Performance

XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems.The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform.

What Algorithm Does XGBoost Use?

The XGBoost library implements the **gradient boosting decision tree algorithm**. This algorithm goes by lots of different names such as *gradient boosting*, *multiple additive regression trees*, *stochastic gradient boosting* or *gradient boosting machines*.

Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. **Gradient boosting** is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.
This approach supports both regression and classification predictive modeling problems.

LightGBM

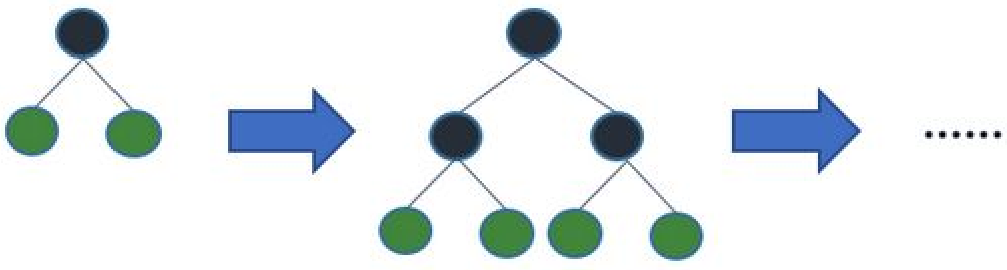
Light GBM is a gradient boosting framework that uses tree based learning algorithm; is a fast, distributed, high-performance gradient boosting framework based on decision tree algorithm, used for ranking, classification and many other machine learning tasks.

LightGBM is a great implementation that is similar to XGBoost but varies in a few specific ways, especially in how it creates the trees.**Light GBM grows tree vertically** while other algorithm grows trees horizontally meaning that Light GBM grows tree **leaf-wise** while other algorithm grows level-wise (like XGBOOST). That is, since it is based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise.

Compared with depth-wise growth, the leaf-wise algorithm can converge much faster. This is because, when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms. Also, it is surprisingly very fast, hence the word 'Light'.

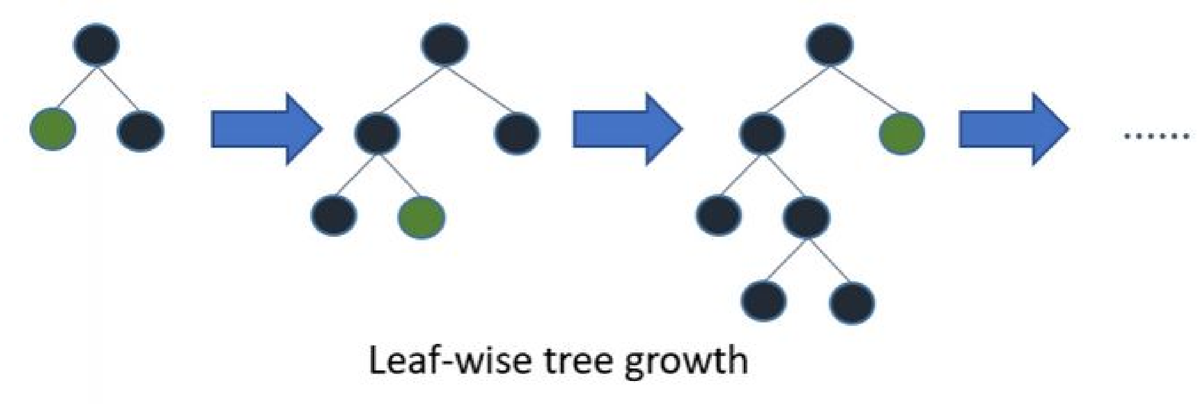
However, leaf-wise splits lead to increase in complexity and may lead to overfitting if not used with the appropriate parameters. It can be overcome by specifying another parameter *max-depth* which specifies the depth to which splitting will occur.

Before is a diagrammatic representation by the makers of the Light GBM to explain the difference clearly.



Level-wise tree growth

Level-wise tree growth in XGBOOST.



Leaf wise tree growth in Light GBM.

Advantages

1. **Faster training speed and higher efficiency**: Light GBM use histogram based algorithm i.e it buckets continuous feature values into discrete bins which fasten the training procedure.
2. **Lower memory usage:** Replaces continuous values to discrete bins which result in lower memory usage.
3. **Better accuracy than any other boosting algorithm:** It produces much more complex trees by following leaf wise split approach rather than a level-wise approach which is the main factor in achieving higher accuracy. However, it can sometimes lead to overfitting which can be avoided by setting the max_depth parameter.
4. **Compatibility with Large Datasets:** It is capable of performing equally good with large datasets with a significant reduction in training time as compared to XGBOOST.
5. **Parallel learning supported.**

Why Light GBM is gaining extreme popularity?

The size of data is increasing day by day and it is becoming difficult for traditional data science algorithms to give faster results. Light GBM is prefixed as ‘Light’ because of its **high speed.** Light GBM can **handle the large size** of data and **takes lower memory to run**. Another reason of why Light GBM is popular is because it **focuses on accuracy of results**. LGBM also **supports GPU learning** and thus data scientists are widely using LGBM for data science application development.

Why don't more people use it then?

XGBoost has been around longer and is already installed on many machines. LightGBM is rather new and didn't have a Python wrapper at first. The current version is easier to install and use so no obstacles here. Many of the more advanced users on Kaggle and similar sites already use LightGBM and for each new competition, it gets more and more coverage. Still, the starter scripts are often based around XGBoost as people just reuse their old code and adjust a few parameters. I'm sure this will increase once there are a few more tutorials and guides on how to use it (most of the non-ScikitLearn guides currently focus on XGBoost or neural networks).

Can we use Light GBM everywhere?

No, it is not advisable to use LGBM on small datasets. Light GBM is **sensitive to overfitting** and can easily overfit small data. There is no threshold on the number of rows but my experience suggests me to use it only for data with 10,000+ rows.

Parameters

Control Parameters

- **max_depth:** It describes the maximum depth of tree. This parameter is used to handle model overfitting. Any time you feel that your model is overfitted, my first advice will be to lower max_depth.
- **min_data_in_leaf:** It is the minimum number of the records a leaf may have. The default value is 20, optimum value. It is also used to deal overfitting
- **feature_fraction:** Used when your boosting is random forest. 0.8 feature fraction means LightGBM will select 80% of parameters randomly in each iteration for building trees.
- **bagging_fraction:** specifies the fraction of data to be used for each iteration and is generally used to speed up the training and avoid overfitting.
- **early_stopping_round:** This parameter can help you speed up your analysis. Model will stop training if one metric of one validation data doesn’t improve in last early_stopping_round rounds. This will reduce excessive iterations.
- **lambda:** lambda specifies regularization. Typical value ranges from 0 to 1.

- **min_gain_to_split:** This parameter will describe the minimum gain to make a split. It can used to control number of useful splits in tree.
- **max_cat_group:** When the number of category is large, finding the split point on it is easily overfitting. So LightGBM merges them into ‘max_cat_group’ groups, and finds the split points on the group boundaries. Default:64

Core Parameters

- **task:** It specifies the task you want to perform on data. It may be either train or predict.
- **application:** This is the most important parameter and specifies the application of your model, whether it is a regression problem or classification problem. LightGBM will by default consider model as a regression model.
    - *regression*: for regression
    - *binary*: for binary classification
    - *multiclass*: for multiclass classification problem
- **boosting:** defines the type of algorithm you want to run, default=gdbt
    - *gbdt*: traditional Gradient Boosting Decision Tree
    - *rf*: random forest
    - *dart*: Dropouts meet Multiple Additive Regression Trees
    - *goss*: Gradient-based One-Side Sampling
- **num_boost_round:** Number of boosting iterations, typically 100+
- **learning_rate:** This determines the impact of each tree on the final outcome. GBM works by starting with an initial estimate which is updated using the output of each tree. The learning parameter controls the magnitude of this change in the estimates. Typical values: 0.1, 0.001, 0.003…
- **num_leaves:** number of leaves in full tree, default: 31
- **device:** default: cpu, can also pass gpu

Metric parameter

- **metric:** again one of the important parameter as it specifies loss for model building. Below are few general losses for regression and classification.
    - mae: mean absolute error
    - mse: mean squared error
    - binary_logloss: loss for binary classification
    - multi_logloss: loss for multi classification

IO parameters

- **max_bin:** it denotes the maximum number of bin that feature value will bucket in.
- **categorical_feature:** It denotes the index of categorical features. If categorical_features=0,1,2 then column 0, column 1 and column 2 are categorical variables.
- **ignore_column:** same as categorical_features just instead of considering specific columns as categorical, it will completely ignore them.
- **save_binary:** If you are really dealing with the memory size of your data file then specify this parameter as ‘True’. Specifying parameter true will save the dataset to binary file, this binary file will speed your data reading time for the next time.

Knowing and using above parameters will definitely help you implement the model. Remember I said that implementation of LightGBM is easy but parameter tuning is difficult. So let’s first start with implementation and then I will give idea about the parameter tuning.

Multiple Layer Perception

Definition

The field of artificial neural networks is often just called **neural networks** or **multi-layer perceptrons** after perhaps the most useful type of neural network. A perceptron is a single neuron model that was a precursor to larger neural networks.

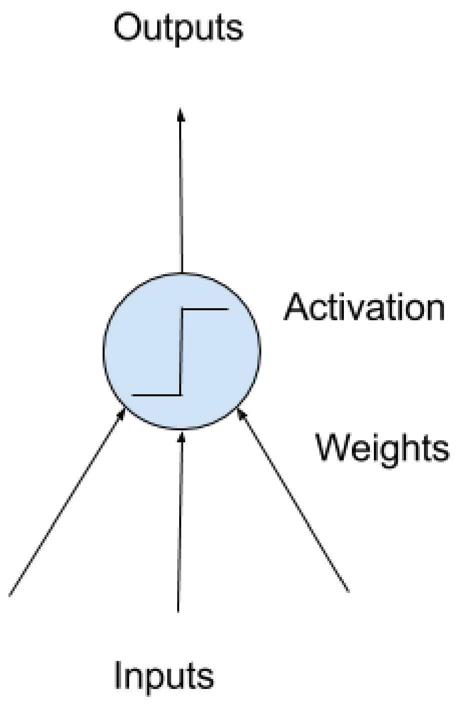
It is a field that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain, but instead to develop robust algorithms and data structures that we can use to model difficult problems.

The power of neural networks come from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm.

The predictive capability of neural networks comes from the hierarchical or multi-layered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example from lines, to collections of lines to shapes.

Neurons

The building block for neural networks are **artificial neurons**. These are simple computational units that have weighted input signals and produce an output signal using an *activation function*.



Neuron Weights

You may be familiar with linear regression, in which case the weights on the inputs are very much like the coefficients used in a regression equation. Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0 and it too must be weighted.

For example, a neuron may have two inputs in which case it requires three weights. One for each input and one for the bias. Weights are often initialized to small random values, such as values in the range 0 to 0.3, although more complex initialization schemes can be used.

Like linear regression, larger weights indicate increased complexity and fragility. It is desirable to keep weights in the network small and regularization techniques can be used.

Activation

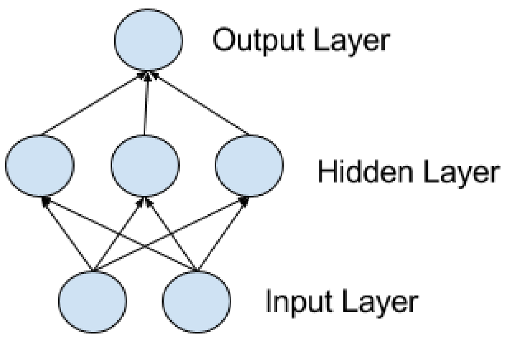
The weighted inputs are summed and passed through an activation function, sometimes called a **transfer function**. An *activation function* is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and strength of the output signal.

Historically simple step activation functions were used where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0.

Traditionally non-linear activation functions are used. This allows the network to combine the inputs in more complex ways and in turn provide a richer capability in the functions they can model. Non-linear functions like the logistic also called the *sigmoid function* were used that output a value between 0 and 1 with an s-shaped distribution, and the hyperbolic tangent function also called tanh that outputs the same distribution over the range -1 to +1. More recently the rectifier activation function has been shown to provide better results.

Networks of Neurons

Neurons are arranged into networks of neurons. A row of neurons is called a **layer** and one network can have multiple layers. The architecture of the neurons in the network is often called the **network topology**.



Input or Visible Layers

The bottom layer that takes input from your dataset is called the **visible layer**, because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset. These *are not neurons as described above*, but simply pass the input value though to the next layer.

Hidden Layers

Layers after the input layer are called **hidden layers** because that are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value.

Given increases in computing power and efficient libraries, very deep neural networks can be constructed. *Deep learning* can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically, but may take seconds or minutes to train using modern techniques and hardware.

Output Layer

The final hidden layer is called the **output layer** and it is responsible for outputting a value or vector of values that correspond to the format required for the problem.

The choice of activation function in he output layer is strongly constrained by the type of problem that you are modeling. For example:

- A regression problem may have a single output neuron and the neuron may have no activation function.
- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the class 1. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0 otherwise to 1.
- A multi-class classification problem may have multiple neurons in the output layer, one for each class (e.g. three neurons for the three classes in the famous iris flowers classification problem). In this case a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp class classification value.

Training Networks

Once configured, the neural network needs to be trained on your dataset.

Data Preparation

You must first prepare your data for training on a neural network. Data must be numerical, for example real values. If you have categorical data, such as a sex attribute with the values “male” and “female”, you can convert it to a real-valued representation called a one hot encoding. This is where one new column is added for each class value (two columns in the case of sex of male and female) and a 0 or 1 is added for each row depending on the class value for that row.
This same one hot encoding can be used on the output variable in classification problems with more than one class. This would create a binary vector from a single column that would be easy to directly compare to the output of the neuron in the network’s output layer, that as described above, would output one value for each class.

Neural networks require the input to be scaled in a consistent way. You can rescale it to the range between 0 and 1 called **normalization**. Another popular technique is to standardize it so that the distribution of each column has the mean of zero and the standard deviation of 1.

Scaling also applies to image pixel data. Data such as words can be converted to integers, such as the popularity rank of the word in the dataset and other encoding techniques.

Stochastic Gradient Descent

The classical and still preferred training algorithm for neural networks is called **stochastic gradient descent**.This is where one row of data is exposed to the network at a time as input. The network processes the input upward activating neurons as it goes to finally produce an output value. This is called a *forward pass* on the network. It is the type of pass that is also used after the network is trained in order to make predictions on new data.

The output of the network is compared to the expected output and an error is calculated. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount that they contributed to the error. This clever bit of math is called the backpropagation algorithm.

The process is repeated for all of the examples in your training data. One of updating the network for the entire training dataset is called an *epoch*. A network may be trained for tens, hundreds or many thousands of epochs.

Weight Updates

The weights in the network can be updated from the errors calculated for each training example and this is called **online learning**. It can result in fast but also chaotic changes to the network. Alternatively, the errors can be saved up across all of the training examples and the network can be updated at the end. This is called **batch learning** and is often more stable.

Typically, because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update is often reduced to a small number, such as tens or hundreds of examples. The amount that weights are updated is controlled by a configuration parameters called the **learning rate**. It is also called the *step size* and controls the step or change made to network weight for a given error. Often small weight sizes are used such as 0.1 or 0.01 or smaller.

The update equation can be complemented with additional configuration terms that you can set.

- *Momentum* is a term that incorporates the properties from the previous weight update to allow the weights to continue to change in the same direction even when there is less error being calculated.
- *Learning Rate Decay* is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine tuning changes later in the training schedule.

Prediction

Once a neural network has been trained it can be used to make predictions. You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously.

The network topology and the final set of weights is all that you need to save from the model. Predictions are made by providing the input to the network and performing a forward-pass allowing it to generate an output that you can use as a prediction.

Disadvantages

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

Bibliography

- https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/ (A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning)
- https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/ (A Gentle Introduction to XGBoost for Applied Machine Learning)
- https://machinelearningmastery.com/develop-first-xgboost-model-python-scikit-learn/ (How to Develop Your First XGBoost Model in Python with scikit-learn)
- https://lightgbm.readthedocs.io/en/latest/ (LightGBM)
- https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc (LightGBM)
- https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/ (Comparison between XGBoost and LightGBM)
- https://machinelearningmastery.com/neural-networks-crash-course/ (Multi-Layer Perceptrons)
- https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/ (Neural networks in Python)
- https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/
- http://benalexkeen.com/feature-scaling-with-scikit-learn/ -> Feature Scaling with scikit-learn