Companion to Machine Learning

Rohan Kumar

Contents

0 Notation **6**
0.1 Data . . . 6

1 Introduction **7**
1.1 What is Machine Learning . . . 7
1.2 Applications of Machine Learning . . . 7
1.3 Types of Machine Learning . . . 7
1.3.1 Supervised Learning . . . 7
1.3.2 Unsupervised Learning . . . 7
1.3.3 Semisupervised Learning . . . 8
1.3.4 Reinforcement Learning . . . 8

2 Data Analysis **9**
2.1 Limitations of Data . . . 9
2.1.1 Nonrepresentative Training Data . . . 9
2.1.2 Poor Quality Data . . . 9
2.1.3 Irrelevant Features . . . 9
2.2 Feature Engineering . . . 9
2.2.1 Feature Construction . . . 9
2.2.2 Feature Selection . . . 10
2.3 Overfitting . . . 11
2.4 Underfitting . . . 11
2.5 Bias Variance Decomposition . . . 11

3 Evaluation of Learning **12**
3.1 Performance Formulation . . . 12
3.2 Testing and Validation . . . 12
3.2.1 Cross Validation . . . 12
3.2.2 Bootstrapping . . . 12
3.3 Performance Evaluation of Classifiers . . . 13
3.3.1 Accuracy and Error . . . 13
3.3.2 Precision and Recall . . . 13
3.3.3 F-Measure . . . 13
3.3.4 Sensitivity and Specificity . . . 14

4 Activation Functions **14**
4.1 Linear/Identity . . . 14
4.2 Threshold . . . 14
4.3 Sigmoid Function . . . 15
4.4 Tanh . . . 15
4.5 Rectified Linear Units . . . 16
4.5.1 Maxout Units . . . 16

5 Convex Optimization **16**
5.1 Normal Solution 16
5.2 Gradient Descent 17
5.2.1 Batch Gradient Descent 17
5.2.2 Stochastic Gradient Descent 18
5.2.3 Mini-batch Gradient Descent 18
5.3 Gradient Descent Optimization 18
5.3.1 Adaptive Gradients 18
5.3.2 RMS Prop 19
5.3.3 Adaptive Moment Estimate 19

6 Instance-Based Learning **19**
6.1 Parametric vs Non-Parametric Methods 19
6.1.1 Approximation 19
6.1.2 Efficiency 19
6.2 K-Nearest Neighbors 20
6.2.1 Implementation 20
6.2.2 Distance Function 20
6.2.3 Decision Boundaries 20
6.2.4 Selection of K 20
6.2.5 Pre-Processing 20
6.2.6 Distance-Weighted Nearest Neighbor 21
6.2.7 High Dimensionality 21

7 Statistical Learning **21**
7.1 Bayesian Learning 21
7.2 Approximate Bayesian Learning 22
7.2.1 Maximum a Posteriori 22
7.2.2 Maximum Likelihood 22
7.3 Bayesian Linear Regression 23
7.3.1 Prediction 23
7.4 Noisy Linear Regression 23
7.4.1 Maximum Likelihood Solution 24
7.4.2 Maximum A Posteriori Solution 24
7.5 Mixture of Gaussians 25
7.5.1 Binary Classification 25
7.5.2 Multinomial Classification 26
7.5.3 Parameter Estimation 26

8 Linear Models **27**
8.1 Linear Regression 27
8.1.1 Formulation 27
8.1.2 Simple Regression 28
8.1.3 Multivariable Regression 28
8.1.4 Cost Function 28

8.1.5 Gradient Descent Solution . . . 28
8.1.6 Normal Equation Solution . . . 28
8.2 Logistic Regression . . . 29
8.2.1 Formulation . . . 29
8.2.2 Prediction . . . 29
8.2.3 Cost Function . . . 29
8.2.4 Solution . . . 30
8.2.5 Softmax Regression . . . 30
8.3 Generalized Linear Models . . . 31
8.4 Regularization . . . 31
8.4.1 Ridge Regression . . . 32
8.4.2 Lasso Regression . . . 32
8.4.3 Elastic Net . . . 32
8.4.4 Early Stopping . . . 32

9 Kernel Methods **32**
9.1 Kernel Trick . . . 32
9.1.1 Formulation . . . 32
9.1.2 Dual Problem . . . 33
9.1.3 Kernel Function . . . 33
9.1.4 Constructing Kernels . . . 33
9.1.5 Common Kernels . . . 33
9.2 Gaussian Processes . . . 34
9.2.1 Function Space . . . 34
9.2.2 Representation . . . 35
9.2.3 Gaussian Process Regression . . . 35
9.3 Support Vector Machines . . . 35
9.3.1 Max-Margin Classifier . . . 36
9.3.2 Soft Margin Classifier . . . 37
9.3.3 Multiclass SVM . . . 38

10 Artificial Neural Networks Primer **38**
10.1 Origins . . . 38
10.2 ANN Unit . . . 39
10.3 Perceptron . . . 39
10.3.1 Threshold Perceptron Learning . . . 39
10.3.2 Sigmoid Perceptron Learning . . . 40
10.4 Multi-Layer Neural Nets . . . 41
10.4.1 n-Layer Perceptron . . . 41
10.4.2 Backpropagation . . . 42

11 Deep Learning **43**
11.1 Training . . . 43
11.1.1 Vanishing/Exploding Gradients Problem . . . 43
11.1.2 Dropout . . . 43

Sources

Throughout this compendium, each piece of information will be formatted as such.

Name / Description of fact Source

Information about fact.

The location which currently contains "Source" could potentially be filled with a variety of sources. Here is how to find the source based off the shortened form.

- **Hands-On Machine Learning** refers to Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition by Aurélien Géron

0 Notation

0.1 Data

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_M \end{pmatrix} : \text{ data point corresponding to a column vector of } M \text{ features}$$

$$\overline{\boldsymbol{x}} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ ... \\ x_M \end{pmatrix} : \text{ concatenation of 1 with the vector}\boldsymbol{x}$$

$$\boldsymbol{X} = \begin{pmatrix} x_{1,1} & ... & x_{1,N} \\ ... & ... & ... \\ x_{M,1} & ... & x_{M,N} \end{pmatrix} : \text{ dataset consisting of } N \text{ data points and } M \text{ features}$$

$$\overline{\boldsymbol{X}} = \begin{pmatrix} 1 & ... & 1 \\ x_{1,1} & ... & x_{1,N} \\ ... & ... & ... \\ x_{M,1} & ... & x_{M,N} \end{pmatrix} : \text{ concatenation of a vector of 1's with the matrix } \boldsymbol{X}$$

$$y = : \text{ output target (regression) or label (classification)}$$

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_N \end{pmatrix} : \text{ vector of outputs for a dataset of } N \text{ points}$$

$$\boldsymbol{x}_* = : \text{ test input / unknown input}$$

$$\boldsymbol{y}_* = : \text{ predicted output}$$

$$N = : \text{ Number of data points in the dataset}$$

$$M = : \text{ Number of a features in a data point}$$

$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ w_2 \\ ... \\ w_M \end{pmatrix}$$

$$\boldsymbol{w}^T = (w_1, w_2, ..., w_M) \text{ or } (w_0, w_1, w_2, ..., w_M)\ w_0 \text{ multiplies the first entry of } \overline{\boldsymbol{x}} \text{ (bias)}$$

Note: bold symbols represents a vector

1 Introduction

1.1 What is Machine Learning

Machine Learning is the field of study that gives computers the ability to learn from data without being explicitly programmed. This is good for problems that require a lot of fine-tuning or for which using a traditional approach yields no good solution. Machine Learning's data dependency allows it to adapt to new data and gain insight for complex problems and large amounts of data.

1.2 Applications of Machine Learning

Machine Learning can be used for a range of tasks and can be seen used in:

- Analyzing images of products on a production line to automatically classify them (Convolutional Neural Net)
- Forecasting company revenue based on performance metrics (Regression or Neural Net)
- Automatically classifying news articles (NLP using Recurrent Neural Networks)
- Summarizing long documents automatically (Natural Language Processing)
- Building intelligent bot for a game (Reinforcement Learning)

1.3 Types of Machine Learning

1.3.1 Supervised Learning

In supervised learning, the training set you feed to the algorithm includes the desired solutions, called labels. (e.g determining if an email is spam would be trained a dataset of example emails labelled as spam or not spam.)

Some commonly used supervised learning algorithms are:

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural Networks

1.3.2 Unsupervised Learning

In unsupervised learning, the training data is unlabeled and the system tries to learn without guidance. The system will try and automatically draw inferences and conclusions about the data and group it as such. (e.g. having a lot of data about blog visitors. Using a clustering algorithm we can group and detect similar visitors).

Some important unsupervised learning algorithms are:

- Clustering
  - K-Means
  - DBSCAN
  - Hierarchical Cluster Analysis
- Anomaly detection and novelty detection
  - One-class SVM
  - Isolation Forest
- Visualization and dimensionality reduction
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally Linear Embedding (LLE)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Association rule learning
  - Apriori
  - Eclat

1.3.3 Semisupervised Learning

Labelling can be very time-consuming and costly, often there will be plenty of unlabelled and a few labelled instances. Algorithms that deal with data that is partially labeled is called semi-supervised learning. A good example of this is Google Photos. Google clusters and groups your photos based on facial recognition (unsupervised) and then you can label one photo and it will be able to label every picture like that (supervised). Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms.

1.3.4 Reinforcement Learning

Reinforcement Learning is a learning algorithm based on a reward system. The learning system, called an agent, can observe the environment, select and perform actions, and get rewards in return (or penalties in the form of negative rewards). It will then learn by itself what the best strategy, called a policy, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

2 Data Analysis

2.1 Limitations of Data

2.1.1 Nonrepresentative Training Data

One thing to look out for when using training data is whether the data is representative of the new cases you want to generalize to. For example if you are training linear regression life satisfaction vs GDP of countries, if some countries are missing from the dataset then the dataset is not fully representative of the problem.

2.1.2 Poor Quality Data

If your data is full of errors, outliers and noise, it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. In order to mitigate this we need to clean the training data.

- If some instances are outliers, it may help to discard them or try to fix the errors manually
- If some instances are missing features, you may decide to ignore that attribute, ignore the instance, fill in the missing values, or train one model with the feature and one without

2.1.3 Irrelevant Features

A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process is called *feature engineering*.

- *Feature selection*: Selecting the most useful features to train on among the existing features
- *Feature extraction*: Combining existing features to produce a more useful one (dimensionality reduction algorithms can help)
- *Ad hoc Features*: Creating new features by gathering new data.

2.2 Feature Engineering

2.2.1 Feature Construction

Features can be modified for various reasons, including to increase predictor performance and to reduce time or memory requirements. Below are common techniques for constructing features.

2.2.1.1 Transformation

Common feature transformations include:

- **Centering** each feature to be around the origin.
- **Scaling** each feature to be of the same scale. For example, scaling can be done to make sure each feature has the same variance or the same maximum absolute value.
- **Logarithmically** transforming each feature to reduce the skewness of feature distributions.

Note that feature transformation runs the risk of discarding useful information. For example, scaling to make each feature have the same variance should not be done if the differing variances of the features are actually relevant to the problem.

2.2.1.2 Feature Extraction

Feature expansion involves combining multiple features into new features when first order interactions are not good enough. For example, given features x_1 and x_2, $x_1 \cdot x_2$ is a new feature (i.e. meta-feature) formed by an expansion of x_1 and x_2.

2.2.1.3 "Ad hoc" Features

Constructing ad hoc features involves applying domain knowledge to introduce custom features.

2.2.2 Feature Selection

Irrelevant features are features that are uncorrelated with a prediction task. Redundant features are features that are highly correlated with one another, so using multiple redundant features does not help with predictions much more than using a single such feature.

Different learning algorithms have differing levels of robustness to irrelevant or redundant features. For example, decision trees are robust to redundant features, since such features have low information gain, while KNN is not, since the set of redundant features will behave as one heavily weighted feature. When possible, these feature should not be selected in the first place. Below are common ways to avoid selecting such features.

2.2.2.1 Wrapper Methods

Wrapper methods involve building a model for feature subsets, and then selecting the best performing model. A "forward search" approach starts with no features and then adds the feature that best improves the model until a certain number of features are selected. A "backward" search approach starts with all features and removes the feature that improves the model the least until a certain number of features have been removed.

Computing all possible feature subsets would guarantee finding the optimal one. However, a problem with M features has 2^M possible feature subsets, so finding all possible subsets is infeasible for large values of M. The forward and backward search approaches approximate this but with a time complexity of $O(M^2)$.

2.2.2.2 Filter Methods

Filter methods, also known as variable ranking, involve assigning each feature a score measuring how informative it is in predictions. This score is determined by some “scoring function” S. Features are then ranked by score, and a number of top features are selected.

2.2.2.3 Embedded Methods

Embedded methods involve modifying the cost function to constrain the choice of model. A common example of this is regularization, which can be used to penalize complex models and encourage a sparse feature set.

2.3 Overfitting

Overfitting is when the model performs well on the training data, but does not generalize well. Complex models can detect subtle patterns in the data, but if the training set is noise, or if it is too small, then the model will likely detect patterns in the noise itself. These patterns will not generalize to new instances. Overfitting often happens when the training data has many features, which allows for an approximation of the target function with many degrees of freedom. We can use regularization to constrain a model to make it simpler to reduce the risk of overfitting.

2.4 Underfitting

Underfitting is the opposite of overfitting: it occurs when the model is too simple to learn the underlying structure of the data. Methods of fixing the problem include:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm
- Reduce the constraints on the model (e.g., reduce the regularization hyperparameter)

2.5 Bias Variance Decomposition

Many machine learning algorithms are based on building a formal model based on the training data (e.g. a decision tree). Models have parameters, which are characteristics that can help in classification (e.g. a node in a decision tree). Models may also have hyper-parameters, which in turn control other parameters in a model (e.g. max height of decision tree).

Generalization errors result from a combination of noise, variance, and bias. Bias concerns how well the type of model fits the data. Models with high bias pay little attention to training data and suffer from underfitting, while models with low bias may pay too much attention to training data and become overfitted. Bias and variance tend to be at odds with one another (high bias typically leads to low variance, and vice versa).

3 Evaluation of Learning

3.1 Performance Formulation

Let y_* be an output generated by a function f approximating some target function. Let y be the corresponding output of the target function. A loss function $l(y, y_*)$ can be used to measure the accuracy of the approximation function f. Some common loss functions include:

- Squared Loss: $l(y, y_*) = (y - y_*)^2$
- Absolute Loss: $l(y, y_*) = |y - y_*|$
- Zero/One Loss: $l(y, y_*) = 1_{y \neq y_*}$

We assume that the data coming from our target function comes from some probability distribution D, and that our training data is a random sample of (x, y) pairs from D. A Bayes Optimal Classifier is a classifier that for any input x, returns the y most likely to be generated by D.

Based on the available training data, the goal of supervised learning is to find a mapping f from x to y such that generalization error $\sum_{(x,y)} D(x, y) l(y, f(x))$ is minimized. However, since D is unknown, we instead estimate the error from the average error in our training or test data, which is $\frac{1}{N} \sum_{n=1}^{N} l(y_n, f(x_n))$.

3.2 Testing and Validation

Models are initially built based on a training dataset. Test sets (also known as holdout sets) are then used to estimate the generalization error. Validation sets are also used to measure the model's performance, but unlike test sets, validation sets can make changes to the model's parameters.

3.2.1 Cross Validation

Cross validation is a technique for measuring how well a model generalizes. The idea behind it is to break up a training data set into K equally sized partitions, and use $K - 1$ of the partitions as training data and the remaining partition for testing. This should be repeated K times, so that all points of data are at some point used for testing. Higher values of K lower the amount of variance of in the error estimation. To avoid training and testing data having a different probability distribution, the data should be shuffled before being split.

3.2.2 Bootstrapping

Bootstrapping is an alternative to cross validation where instead of dividing a training data set into partitions, a random sample of points (with possible duplicates) is used as training data. The remaining points are then used as testing data, with the goal being similar to that of cross validation.

3.3 Performance Evaluation of Classifiers

Consider the following terminology for classification problems:

- True positive (TP) - Examples of class 1 predicted as class 1
- False positive (FP) - Examples of class 0 predicted as class 1 (Type 1 Error)
- True negative (TN) - Examples of class 0 predicted as class 0
- False negative (FN) - Examples of class 1 predicted as class 0 (Type 2 Error)

3.3.1 Accuracy and Error

The following formulas can be used to measure accuracy and error:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$ErrorRate = \frac{FP + FN}{TP + TN + FP + FN}$$

3.3.2 Precision and Recall

Precision and recall can be measured as follows:

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

Precision measures the ratio of positive predictions that were correct, while recall measures the ratio of total positive instances that were predicted. Similarly to how variance and bias are often at odds with one another, so are precision and recall.

3.3.3 F-Measure

An F-measure (also known as a F1 score) measures a model's accuracy by taking into account both precision and recall as follows:

$$F = \frac{2PR}{P + R}$$

To adjust the relative importance of precision vs recall, a weighted F-measure can be used, which is defined as follows:

$$F = \frac{(1 + \beta^2)PR}{\beta^2 P + R}$$

In a standard F-measure, $\beta = 1$, $\beta < 1$ means that precision is valued over recall, while $\beta > 1$ means recall is valued over precision.

3.3.4 Sensitivity and Specificity

Sensitivity is the same measure as recall. Specificity is a measure of how well a classifier avoids false positives, and is measured as:

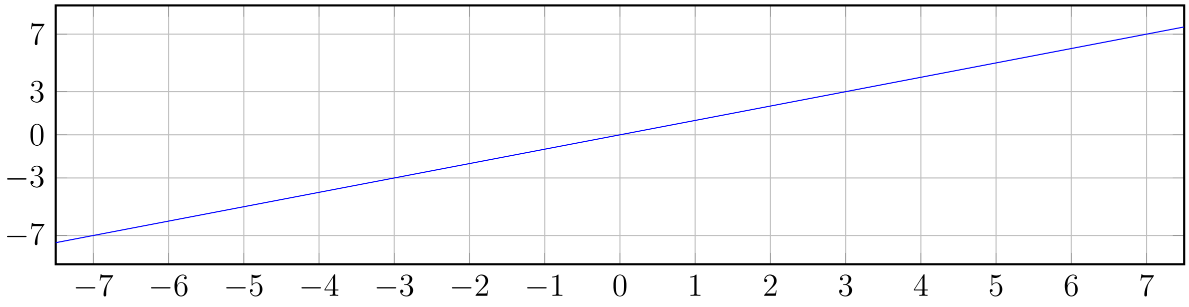
$$Specificity = \frac{TN}{TN + FP}$$

4 Activation Functions

4.1 Linear/Identity

The *linear function* is an activation function where the output is proportional to the input and the *identity function* is a subset of the linear function where $a = 1$.

$$h(x) = ax$$

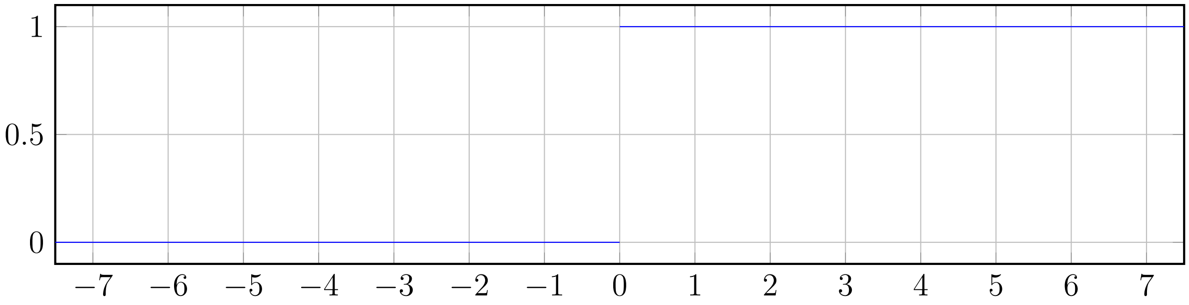


The linear function gives a range of activations so it is not a binary activation. The derivative is constant so the gradient has no relationship with $\boldsymbol{X}$ and therefore backpropagation and gradient descent would not work with this activation function.

4.2 Threshold

The *threshold function* denoted *heaviside*$(\cdot)$ outputs a number 0 or 1 based on its input z. It is defined as a piecewise function

$$heaviside(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

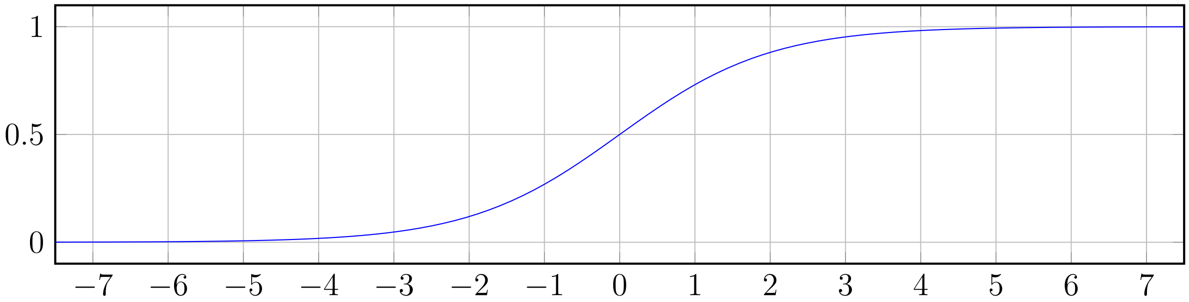


The key property of the threshold function is that it will predict 1 when z, which will generally be our prediction $\boldsymbol{w}^T\boldsymbol{x}$, is greater than 0 else it will predict 0.

4.3 Sigmoid Function

The *sigmoid function* denoted $\sigma(\cdot)$ outputs a number between 0 and 1. It is defined as

$$\sigma(t) = \frac{1}{1 + exp(-t)}$$

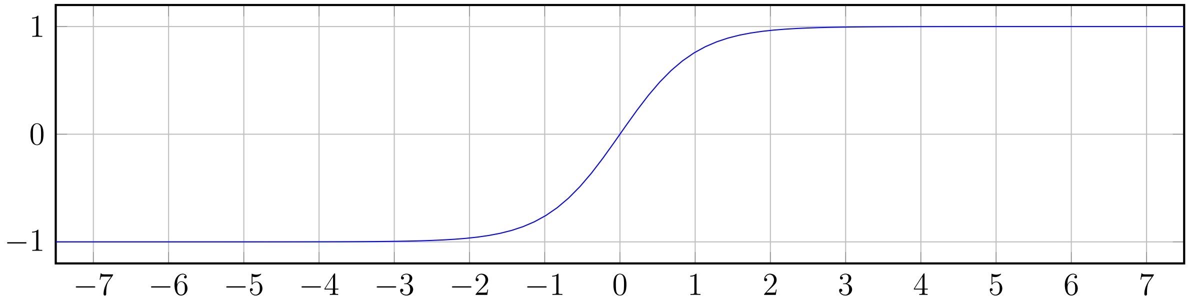


The key property of the sigmoid function is that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a sigmoid function is useful for classification since it can predict 1 when $\boldsymbol{w}^T\boldsymbol{x}$ is positive and 0 if it is negative.

4.4 Tanh

The *tanh function* denoted $tanh(\cdot)$ outputs a number between -1 and 1. It is zero-centered function making it easier to model inputs that have strongly negative, neutral, and strongly positive values, otherwise it is similar to the sigmoid function.

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



The gradient is stronger for tanh than sigmoid, however, tanh also has the vanishing gradient problem.

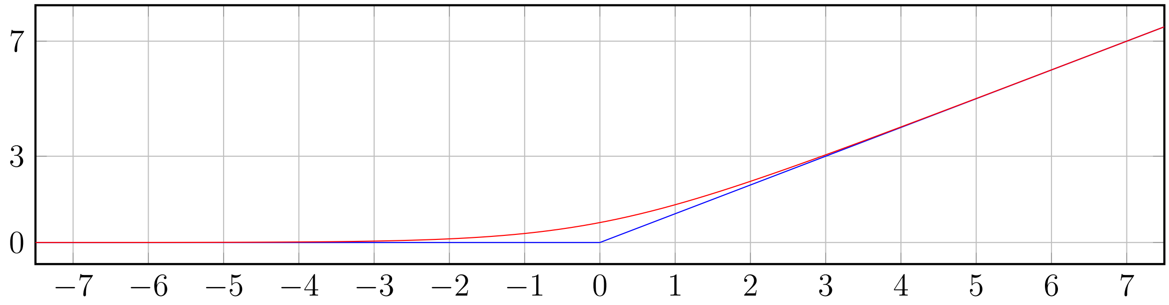
4.5 Rectified Linear Units

The *RelU function* plotted in blue is defined as

$$h(a) = max(0, a)$$

and the soft version (”Softplus”) plotted in red is defined as

$$h(a) = log(1 + e^a)$$



The benefit of using the RelU activation function is that the gradient is 0 or 1 so it helps mitigate the vanishing problem for deep neural networks. The softplus function does not prevent gradient vanishing.

4.5.1 Maxout Units

Generalization of rectified linear units where we can have several linear parts and having them be whatever we want rather than having a 0 part. They can be thought of as the aggregation of a hidden layer of identity units with a max unit.

$$max\{\sum_i w_i^{(1) x_i, \sum_i w_i^{(2)}} x_i, \sum_i w_i^{(3)} x_i, ...\}$$

5 Convex Optimization

In machine learning we can often turn problems into convex functions and simplify the problem into finding the global minima of the function, which in essence is minimizing the training error. One of the key theorem’s of a convex function is that the local minimum of a convex function is also a global minimum. Therefore we can apply many methods to find parameters that satisfy the global minima.

5.1 Normal Solution

To find the value of our parameter (generally $\boldsymbol{w}$) that minimizes the cost function, there is a closed-form solution. We can express this closed form solution for any convex loss function as follows.

$$\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = 0$$

The limitations of using the Normal Solution is that we usually have to compute the inverse of $\boldsymbol{X}^T\boldsymbol{X}$ which is a $(M+1)\times(M+1)$ matrix (where M is the number of features). The computational complexity of inverting such a matrix is typically about $O(M^{2.4})$ to $O(M^3)$, depending on the implementation. Some of the other approaches below are better suited for cases where there are a large number of features or too many training instances to fit in memory.

5.2 Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solution to a wide range of problems. The general ida is to tweak parameters iteratively in order to minimize a cost function. The main concept utilized in gradient descent is to measure the local gradient of the error with regard to the parameter vector and move in the direction of the descending gradient. Once the gradient is zero, you have reached the minimum.

You start with filling your parameter $\boldsymbol{w}$ with random values (random initialization. Then you improve it gradually, taking small steps at a time, each step attempting to decrease the cost function, until the algorithm converges to a minimum. One important parameter of Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, if the learning rate is too high, you might jump across the minimum possibly, higher than you were before and potentially make the algorithm diverge. One gradient descent technique is having a learning rate that changes as you approach the minimum to prevent overshoot, also called the learning schedule.

A limitation of Gradient Descent is when the cost function we are dealing with is not a convex function. In this case holes, ridges, irregular terrain will make the convergence to the minimum difficult.

5.2.1 Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter w_j - how much the cost function will change if you change w_j a little bit. This is equivalent to the partial derivative of the cost function with regard to the parameter w_j. For the entire parameter vector $\boldsymbol{w}$ we can denoted the gradient vector as $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$.

Once we have the gradient vector, which points uphill, we descend in the opposite direction (subtract $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$ from $\boldsymbol{w}$). This is where we use our learning rate α to determine the size of the downhill step.

$$\boldsymbol{w}^{(nextstep)} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$$

The limitation of Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.

5.2.2 Stochastic Gradient Descent

Stochastic Gradient Descent picks a random instance in the training set at every step and computes the gradients based on only that single instance. This makes the algorithm much faster and also makes it possible to train on huge training sets. However, due to its stochastic nature, this algorithm will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. Therefore, once the algorithm stops, the final parameter values are good, but not optimal.

This can actually help when the cost function is very irregular (not convex) as it can help the algorithm jump out of a local minima. One solution to the problem of being unable to settle at the minimum is gradually reducing the learning rate. The steps start out large (helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minima. The function that determines the learning rate is called the *learning schedule*.

5.2.3 Mini-batch Gradient Descent

Mini-batch GD is a combination of Batch GD and Stochastic GD. At each step, instead of computing the gradients based on the full training set or based on just one instance, Mini-batch GD computes the gradient on small random sets of instances called *mini-batches*. The main advantage of this over Stochastic GD is that you get a performance boost from hardware optimization of matrix operations. Mini-batch will perform better to get closer to the minimum than Stochastic GD but it may be harder for it to escape local minima.

5.3 Gradient Descent Optimization

When training neural networks there can be issues involving slow convergence, dimensionality and magnitude. So other methods were introduced to be able to quickly train neural networks with accuracy for large amounts of data.

5.3.1 Adaptive Gradients

Adagrad is an algorithm for gradient-based optimization that adapts to the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features and larger updates for parameters associated with infrequent features. It is well suited for dealing with sparse features.

$$r_t \leftarrow r_{t-1} + (\frac{\partial E_n}{\partial w_{ji}})^2$$

$$w_{ji} \leftarrow w_{ji} - \frac{\alpha}{\sqrt{r_t}} \frac{\partial E_n}{\partial w_{ji}}$$

The problem is that the learning rate $\frac{\alpha}{\sqrt{r_t}}$ decays too quickly.

5.3.2 RMS Prop

To combat the problem with AdaGrad we can instead divide by the root mean square of partial derivatives.

$$r_t \leftarrow \beta r_{t-1} + (1-\beta)(\frac{\partial E_n}{\partial w_{ji}})^2 \quad \text{where } 0 \leq \beta \leq 1$$

$$w_{ji} \leftarrow w_{ji} - \frac{\alpha}{\sqrt{r_t}} \frac{\partial E_n}{\partial w_{ji}}$$

The problem now is that the gradient lacks momentum.

5.3.3 Adaptive Moment Estimate

Now to induce momentum, Adam replaces the gradient by the moving average.

$$r_t \leftarrow \beta r_{t-1} + (1-\beta)(\frac{\partial E_n}{\partial w_{ji}})^2 \quad \text{where } 0 \leq \beta \leq 1$$

$$s_t \leftarrow \gamma s_{t-1} + (1-\gamma)(\frac{\partial E_n}{\partial w_{ji}}) \quad \text{where } 0 \leq \gamma \leq 1$$

$$w_{ji} \leftarrow w_{ji} - \frac{\alpha}{\sqrt{r_t}} s_t$$

6 Instance-Based Learning

6.1 Parametric vs Non-Parametric Methods

Datasets can be represented as a set of points in a high-dimensional space; a data point with n features $x_1, x_2, ..., x_n$ can be represented with the feature vector $(x_1, x_2, ..., x_n)$ in n-dimensional space. Parametric methods of supervised learning attempt to model the data using these features, while non-parametric (also known as instance-based) methods do not.

6.1.1 Approximation

Parametric methods use parameters to create global approximations. Non-parametric methods instead create approximations based on local data.

6.1.2 Efficiency

Parametric methods do most of their computation beforehand, and the summarize their results in a set of parameters. Non-parametric methods tend to have a shorter training time but a longer query answering time.

6.2 K-Nearest Neighbors

K-nearest neighbors (KNN) is a common non-parametric method. The idea is to predict the value of a new point based on the values of the K most similar (i.e. closest) points.

6.2.1 Implementation

A common implementation of KNN involves looping through all N points in a training set and computing their distance to some point x. Then the K nearest points are selected. This process can be sped up by storing the data points in a data structure that helps facilitate distance-based search (e.g. a k-d tree).

6.2.2 Distance Function

"Nearby" means of minimal distance, which is commonly defined by Euclidean distance. Other distance functions $d(x, x')$ can be used, though must meet the following conditions:

- $d(x, x') = d(x', x)$ (i.e. symmetric)
- $d(x, x) = 0$ (i.e. definite)
- $d(a, c) \leq d(a, b) + d(b, c)$ (i.e. triangle inequality holds)

6.2.3 Decision Boundaries

Decision boundaries define the borders of a single classification of input. These boundaries are formed of sections of straight lines that are equidistant to two points of different classes. A highly jagged line is an indicator of overfitting, while a simple line is an indicator of underfitting.

6.2.4 Selection of K

The selection of the value of K is a bias-variance tradeoff. Low values of K have high variance but low bias, while high values of K have low variance but high bias. High-values of K result in smoother decision boundaries, which can be a sign of underfitting, and vice versa.

K can be selected experimentally by evaluating the performance for different values of K through cross-validation or against a testing set. In theory, as the number of training examples approaches infinity, the error rate of a 1NN classifier is at worst twice that of the Bayes Optimal Classifier.

6.2.5 Pre-Processing

Some common forms of pre-processing for KNN include:

- Removing undesirable inputs. Common removal methods are:
    - Editing methods, which involve eliminating noisy points of data.

  - Condensation methods, which involve selecting a subset of data that produces the same or very similar classifications.

- Use custom weights for each feature (not all features may be equally relevant for the situation)

6.2.6 Distance-Weighted Nearest Neighbor

A common problem with KNN is that it can be sensitive to small changes in the training data. One way to mitigate with drawback is to compute a weight for each neighbor based on its distance (e.g. through a Gaussian distribution), and this weight determines how much of an influence that point's value has. This differs from standard KNN which weighs the values of the K nearest neighbors equally and ignores all other values.

6.2.7 High Dimensionality

In uniformly distributed high-dimensional spaces, distances between points tend to be roughly equal, since there are so many features that changing a few features results in only a small change in distance. However, KNN can still be applied in practice for high-dimensional spaces, since data in high-dimensional spaces tends to be concentrated around certain hubs rather than uniformly distributed.

7 Statistical Learning

Data is often incomplete, indirect, or noisy. Statistical learning lets us consider forms of uncertainty to help us build better models. If we have access to the underlying probability distribution of the data, then we can form an optimal regression or classifier. In practice we typically do not know the underlying probability distributions, so we have to estimate them from the available training data. It is generally best to choose a family of parametric distributions (e.g. Gaussian or Binomial) and then determine which parameters describe the available training data the best. This is known as a density estimate and we assume that each point of training data is independently selected from the same distribution.

7.1 Bayesian Learning

Bayes' theorem describes the probability of an event H given evidence e.

$$\begin{aligned} P(H|e) &= \frac{P(e|H)P(H)}{P(e)} && (1) \\ &= kP(e|H)P(H) && (2) \end{aligned}$$

where:

- $P(H|e)$: Posterior probability
- $P(e|P)$: Likelihood

- $P(H)$: Prior probability
- $P(e)/k$: Normalizing constant

Bayesian Learning consists of determining the posterior probability using Bayes' theorem.

Suppose we want to make a prediction about an unknown quantity $\boldsymbol{X}$ we can consider the hypothesis space which represents all possible models h_i to predict the scenario.

$$P(\boldsymbol{X}|\boldsymbol{e}) = \sum_i P(\boldsymbol{X}|e, h_i)P(h_i|e) \tag{3}$$

$$= \sum_i P(\boldsymbol{X}|h_i)P(h_i|e) \tag{4}$$

This prediction yields the weighted combination of all the hypothesis' in the hypothesis space based on it's likelihood from the evidence. The prior $P(h_i|e)$ is yields the weight for each hypothesis and $P(\boldsymbol{X}|h_i)$ yields the likelihood of the hypothesis for the unknown quantity $\boldsymbol{X}$.

Bayesian probability is:

- Optimal: give a prior probability, no prediction is correct more often than the Bayesian prediction.
- Overfitting-free: all hypothesis are weighted and considered, eliminating overfitting.

One of the constraints of bayesian learning is that it can be intractable when the hypothesis space grows very large, often as a result of approximating a continuous hypothesis space with many discrete hypothesis. This requires us to approximate Bayesian Learning.

7.2 Approximate Bayesian Learning

7.2.1 Maximum a Posteriori

Maximum a Posteriori (MAP) makes predictions based on only the most probable hypothesis $h_{MAP} = argmax_{h_i} P(h_i|e)$. This differs from Bayesian learning, which makes predictions for all hypothesis weighted by their probability. MAP and Bayesian learning predictions tend to converge as the amount of data increases, and overfitting can be mitigated by giving complex hypothesis a low prior probability. However, finding h_{MAP} may be difficult or intractable.

7.2.2 Maximum Likelihood

Maximum Likelihood (ML) simplifies MAP by assuming uniform prior probabilities and then makes a prediction based on the most probable hypothesis h_{ML}. ML tends to be less accurate than MAP and Bayesian predictions, it is also subject to overfitting due to the prior probabilities being uniform. Finding h_{ML} is easier than finding h_{MAP} since finding h_{ML} for $P(e|h)$ is equivalent to calculating it for $argmax_h \sum_n logP(e_n|h)$.

7.3 Bayesian Linear Regression

Instead of taking the hypothesis $\boldsymbol{w}$ that maximizes the posterior, we can compute the posterior and work with that directly as follows:

$$\begin{aligned} P(\boldsymbol{w}|\boldsymbol{y},\boldsymbol{X}) &= \frac{P(\boldsymbol{y}|\boldsymbol{w},\boldsymbol{X})P(\boldsymbol{w}|\boldsymbol{X})}{P(\boldsymbol{y}|\boldsymbol{x})} \\ &= ke^{-\frac{1}{2}(\boldsymbol{w}-\overline{\boldsymbol{w}})^T\boldsymbol{A}(\boldsymbol{w}-\overline{\boldsymbol{w}})} \\ &= N(\overline{\boldsymbol{w}},\boldsymbol{A}^{\boldsymbol{-1}}) \end{aligned}$$

where

$$\begin{aligned} \overline{w} &= \sigma^{-2}\boldsymbol{A}^{\boldsymbol{-1}}\overline{\boldsymbol{X}}\boldsymbol{y} \\ A &= \sigma^{-2}\overline{\boldsymbol{X}\boldsymbol{X}}^T + \boldsymbol{\Sigma}^{\boldsymbol{-1}} \end{aligned}$$

7.3.1 Prediction

Let us consider an input $\boldsymbol{x}_*$ for which we want a corresponding prediction y_*.

$$\begin{aligned} P(y_*|\overline{\boldsymbol{x}_*},\overline{\boldsymbol{X}},\boldsymbol{y}) &= \int_{\boldsymbol{w}} P(y_*|\overline{\boldsymbol{x}_*},\boldsymbol{w})P(\boldsymbol{w}|\overline{\boldsymbol{X}},\boldsymbol{y})d\boldsymbol{w} \\ &= k\int_{\boldsymbol{w}} e^{-\frac{(y_*-\overline{\boldsymbol{x}}^T\boldsymbol{w})^2}{2\sigma^2}} ke^{-\frac{1}{2}(\boldsymbol{w}-\overline{\boldsymbol{w}})^T\boldsymbol{A}(\boldsymbol{w}-\overline{\boldsymbol{w}})}d\boldsymbol{w} \\ &= N(\overline{\boldsymbol{x}_*}^T\boldsymbol{A}^{-1}\overline{\boldsymbol{X}}\boldsymbol{y},\overline{\boldsymbol{x}_*}^T\boldsymbol{A}^{-1}\overline{\boldsymbol{x}_*}) \end{aligned}$$

This gives us a gaussian distribution of the solution. Generally for the prediction we take the mean of the distribution.

7.4 Noisy Linear Regression

Linear Regression data is often noisy and isn't distributed in a perfectly straight line.

$$\boldsymbol{y} = f(\overline{\boldsymbol{X}}) + \varepsilon$$

Now assuming our noise ε is a Gaussian distribution (good in practice and mathematically) then we get the likelihood distribution:

$$\begin{aligned} P(\boldsymbol{y}|\overline{\boldsymbol{X}},\boldsymbol{w},\sigma) &= N(\boldsymbol{y}|\boldsymbol{w}^T\overline{\boldsymbol{X}},\sigma^2) \\ &= \prod_{n=1}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(y_n-\boldsymbol{w}^T\overline{\boldsymbol{x}}_n)^2}{2\sigma^2}} \end{aligned}$$

7.4.1 Maximum Likelihood Solution

We can apply maximum likelihood to this and find the best $\boldsymbol{w}^*$ by maximizing the likelihood of the data.

$$\begin{aligned}
\boldsymbol{w}^* &= argmax_{\boldsymbol{w}} P(\boldsymbol{y}|\overline{\boldsymbol{X}}, \boldsymbol{w}, \sigma) \\
&= argmax_{\boldsymbol{w}} \prod_n e^{-\frac{(y_n - \boldsymbol{w}^T]\overline{\boldsymbol{x}}_n)^2}{2\sigma^2}} \\
&= argmax_{\boldsymbol{w}} \sum_n -\frac{(y_n - \boldsymbol{w}^T\overline{\boldsymbol{x}}_n)^2}{2\sigma^2} \\
&= argmin_{\boldsymbol{w}} \sum_n (y_n - \boldsymbol{w}^T\overline{\boldsymbol{x}}_n)^2
\end{aligned}$$

This leads us to least square problem derived in the Linear Regression section using the Mean Squared Error.

7.4.2 Maximum A Posteriori Solution

Alternatively we can apply MAP to our noisy linear regression problem and find $\boldsymbol{w}^*$ with the highest posterior probability (most probable hypothesis).

Gaussian Prior:

$$P(\boldsymbol{w}) = N(0, \boldsymbol{\Sigma})$$

Posterior:

$$\begin{aligned}
P(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) &\propto P(\boldsymbol{w})P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) \\
&= ke^{-\frac{\boldsymbol{w}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{w}}{2}} e^{-\frac{\sum_n (y_n - \boldsymbol{w}^T\boldsymbol{x}_n)^2}{2\sigma^2}}
\end{aligned}$$

We can now simplify this to an optimization problem of finding

$$\begin{aligned}
\boldsymbol{w}^* &= argmax_{\boldsymbol{w}} P(\boldsymbol{w}|\overline{\boldsymbol{X}}, \boldsymbol{y}) \\
&= argmax_{\boldsymbol{w}} - \sum_n (y_n - \boldsymbol{w}^T\overline{\boldsymbol{x}}_n)^2 - \boldsymbol{w}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{w} \\
&= argmin_{\boldsymbol{w}} \sum_n (y_n - \boldsymbol{w}^T\overline{\boldsymbol{x}}_n)^2 + \boldsymbol{w}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{w}
\end{aligned}$$

Let $\boldsymbol{\Sigma}^{-1} = \lambda\boldsymbol{I}$ then

$$\boldsymbol{w}^* = argmin_{\boldsymbol{w}} \sum_n (y_n - \boldsymbol{w}^T\overline{\boldsymbol{x}}_n)^2 + \lambda\|\boldsymbol{w}\|^2$$

This is the ridge regularized least square problem that reduces overfitting.

7.5 Mixture of Gaussians

Now we consider the probabilistic generative model for classification. We can compute the posterior $P(C|\boldsymbol{x})$ according to Bayes' theorem to estimate the probability of the class for a given data point. Here we are using Bayes theorem for inference rather than for Bayesian learning (estimating parameters of a model).

$$\begin{aligned} P(C|\boldsymbol{x}) &= \frac{P(\boldsymbol{x}|C)P(C)}{\sum_C P(\boldsymbol{x}|C)P(C)} \\ &= kP(\boldsymbol{x}|C)P(C) \end{aligned}$$

where:

- $P(C)$: Prior probability of class C
- $P(\boldsymbol{x}|C)$: class conditional distribution of $\boldsymbol{x}$

with the following assumptions:

- In classification the number of classes is finite, so a natural prior $P(C)$ is the multinomial $P(C = c_k) = \pi_k$
- when $\boldsymbol{x} \in \mathbb{R}$ then it is often ok to assume that $P(\boldsymbol{x}|C)$ is Gaussian.
- Assume the same covariance matrix $\boldsymbol{\Sigma}$ is used for each class.

From our assumptions we get

$$P(\boldsymbol{x}|c_k) \propto e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)}$$

7.5.1 Binary Classification

Subbing our assumptions into Bayes theorem for binary classification and simplifying, we get the following posterior distribution for classes c_k, c_j.

$$P(c_k|\boldsymbol{x}) = \frac{1}{1 + e^{(-\boldsymbol{w}^T\boldsymbol{x}+w_0)}}$$

where:

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_k - \boldsymbol{\mu}_j) \\ w_0 &= \boldsymbol{\mu}_k^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k + \frac{1}{2}\boldsymbol{\mu}_j^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_j + log\frac{\pi_k}{\pi_j} \end{aligned}$$

We can observe that this equation is the logistic sigmoid and we can draw the class boundary/linear separator at $\sigma(\boldsymbol{w}^T\boldsymbol{x} + w_0) = 0.5$ which is equivalent to $\boldsymbol{w}_k^T\overline{\boldsymbol{x}} = 0$.

7.5.2 Multinomial Classification

Now similarly for a multi-class problem where all K classes are a gaussian distribution we get.

$$P(c_k|\boldsymbol{x}) = \frac{e^{\boldsymbol{w}_k^T\boldsymbol{x}}}{\sum_j e^{\boldsymbol{w}_j^T\boldsymbol{x}}}$$

where

$$\boldsymbol{w}_k^T = (-\frac{1}{2}\boldsymbol{\mu}_k^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k + log(\pi_k), \boldsymbol{\mu}_k^T\boldsymbol{\Sigma}^{-1})$$

This process can be extrapolated for classes that aren't all distributed with a gaussian distribution (e.g exponential, poisson, bernoulli etc ...). We can see that this is a specific case of the softmax distribution which is a generalization of the sigmoid and is discussed in further detail in the next section.

7.5.3 Parameter Estimation

Let $\pi = P(y = C_1)$ and $1 - \pi = P(y = C_2)$ where $P(\boldsymbol{x}|C_1) = N(\boldsymbol{x}|\boldsymbol{\mu}_1, \boldsymbol{\Sigma})$ and $P(\boldsymbol{x}|C_2) = N(\boldsymbol{x}|\boldsymbol{\mu}_2, \boldsymbol{\Sigma})$. In order to actually use bayesian inference to get the classification probability of our input data, we need to learn the parameters π, $\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_2$ and $\boldsymbol{\Sigma}$. We can estimate the parameters by maximum likelihood, maximum a posteriori or bayesian learning. This example will demonstrate using maximum likelihood to learn these parameters.

We can express the Likelihood of our training set as $L(\boldsymbol{X}, \boldsymbol{y}) = P(\boldsymbol{X}, \boldsymbol{y}|\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma})$. We want to maximize the likelihood in order to use Bayes inference.

$$L(\boldsymbol{X}, \boldsymbol{y}) = \prod_n [\pi|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}]^{y_n}[(1-\pi)|N(\boldsymbol{x}_n|\boldsymbol{\mu}_2, \boldsymbol{\Sigma})]^{1-y_n}$$

Taking the log we can turn this into an optimization problem of finding

$$argmax_{\pi,\boldsymbol{\mu}_1,\boldsymbol{\mu}_2,\boldsymbol{\Sigma}} \sum_n y_n[log(\pi) - \frac{1}{2}(\boldsymbol{x}_n - \boldsymbol{\mu}_1)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}_1)] \\ + (1-y_n)[log(1-\pi) - \frac{1}{2}(\boldsymbol{x}_n - \boldsymbol{\mu}_2)^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}_2)]$$

7.5.3.1 Estimate π (probability of class)

$$0 = \frac{\partial log(L(\boldsymbol{X}, \boldsymbol{y}))}{\partial \pi}$$

$$\pi = \frac{\sum_n y_n}{N}$$

7.5.3.2 Estimate $\boldsymbol{\mu}$ (mean of classes)

$$\begin{aligned} 0 &= \frac{\partial log(L(\boldsymbol{X}, \boldsymbol{y}))}{\partial \boldsymbol{\mu}_1} \\ \boldsymbol{\mu}_1 &= \frac{\sum_n y_n \boldsymbol{x}_n}{N_1} \end{aligned}$$

and

$$\begin{aligned} 0 &= \frac{\partial log(L(\boldsymbol{X}, \boldsymbol{y}))}{\partial \boldsymbol{\mu}_2} \\ \boldsymbol{\mu}_2 &= \frac{\sum_n (1 - y_n) \boldsymbol{x}_n}{N_2} \end{aligned}$$

7.5.3.3 Estimate $\boldsymbol{\Sigma}$ (covariance matrix)

$$\begin{aligned} 0 &= \frac{\partial log(L(\boldsymbol{X}, \boldsymbol{y}))}{\partial \boldsymbol{\Sigma}} \\ \boldsymbol{\Sigma} &= \frac{N_1}{N} \boldsymbol{S}_1 + \frac{N_2}{N} \boldsymbol{S}_2 \end{aligned}$$

where S_k are the empirical covariance matrices of the class k

$$\begin{aligned} \boldsymbol{S}_1 &= \frac{1}{N_1} \sum_{n \in C_1} (\boldsymbol{x}_n - \boldsymbol{\mu}_1)(\boldsymbol{x}_n - \boldsymbol{\mu}_1)^T \\ \boldsymbol{S}_2 &= \frac{1}{N_1} \sum_{n \in C_2} (\boldsymbol{x}_n - \boldsymbol{\mu}_2)(\boldsymbol{x}_n - \boldsymbol{\mu}_2)^T \end{aligned}$$

8 Linear Models

8.1 Linear Regression

8.1.1 Formulation

Linear Regression is a supervised machine learning algorithm where the predicted output is continuous and has a constant slope. Our main objective is to generate a line that minimizes the distance from the line to all of data points. This is essentially minimizing the error and maximizing our prediction accuracy.

8.1.2 Simple Regression

A simple two variable linear regression uses the slope-intercept form, where m and b are the variables our algorithm will try to "learn". $\boldsymbol{x}$ represents our input data and y represents the prediction.

$$y = m\boldsymbol{x} + b$$

8.1.3 Multivariable Regression

Often times there are more than one feature in the data and we need a more complex multi-variable linear equation as our hypothesis. We can represent our hypothesis with the follow multi-variable linear equation, where $\boldsymbol{w}$ are the weights and $\boldsymbol{x}$ is the input data.

$$\begin{aligned} h_{\boldsymbol{w}}(\boldsymbol{x}) &= w_0x_0 + w_1x_1w_2x_2 + ... + w_nx_n \\ &= \boldsymbol{w}^T\boldsymbol{x} \end{aligned}$$

8.1.4 Cost Function

To predict based on a dataset we first need to learn the weights that minimize the mean squared error (euclidean loss) of our hypothesis. We can define the following to be our cost function to minimize with N being the number of data points and n being the n^{th} training example. This can be proven with statistical linear regression.

$$J(\boldsymbol{w}) = \frac{1}{2N}\sum_{n=1}^{N}(h_{\boldsymbol{w}}(\overline{\boldsymbol{x}}_n) - y_n)^2$$

8.1.5 Gradient Descent Solution

Now to solve for $\boldsymbol{w}$ we can use Gradient Descent and iteratively update $\boldsymbol{w}$ until it converges. We get the slope of the cost function to be:

$$\frac{\partial J\boldsymbol{w}}{\partial \boldsymbol{w}_j} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{w}^T\overline{\boldsymbol{x}}_n - y_n)x_{j,n}$$

now applying a step α we can iteratively change $\boldsymbol{w}$ until it reaches the global minima.

$$\boldsymbol{w}_j := \boldsymbol{w}_j - \alpha\frac{1}{N}\sum_{n=1}^{N}(h_{\boldsymbol{w}}(\overline{\boldsymbol{x}}_n) - y_n)$$

8.1.6 Normal Equation Solution

The closed form solution to the linear system in $\boldsymbol{w}$

$$\frac{\partial J\boldsymbol{w}}{\partial \boldsymbol{w}_j} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{w}^T\overline{\boldsymbol{x}}_n - y_n)x_{j,n}$$

writing this as a linear system in w we get $A\boldsymbol{w} = b$ where

$$A = \sum_{n=1}^{N} (\boldsymbol{x}_n \boldsymbol{x}_n^T) \text{ and } b = \sum_{n=1}^{N} (\boldsymbol{x}_n y_n)$$

so we can solve for $\boldsymbol{w} = \boldsymbol{A}^{-1}\boldsymbol{b}$ and get the following vectorized solution.

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

8.2 Logistic Regression

8.2.1 Formulation

Logistic regression is an algorithm used for classification. It is used to estimate the probability that an instance belongs to a particular class. If the estimated probability is greater than 50%, then the model predicts the instance belongs to that class, and otherwise it predicts it does not. Logistic Regression is form of discriminative learning as it attempts to model $P(c_k|\boldsymbol{x})$ directly, this is unlike the generative model where $P(c_k)$ and $P(\boldsymbol{x}|c_k)$ are found by max likelihood and $P(c_k|\boldsymbol{x})$ by Bayesian Inference.

8.2.2 Prediction

Logistic Regression computers the weighted sum of the input features (plus a bias term) and outputs the logistic (Sigmoid Function) of the result. The hypothesis for class k is given by

$$\pi_k = h_{\boldsymbol{w}}(\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\overline{\boldsymbol{x}})$$

Once the Logistic Regression model has estimated the probability that an instance $\boldsymbol{x}$ belongs to the positive class, it can make its prediction y easily.

$$y = \begin{cases} 0 & \pi_k < 0.5 \\ 1 & \pi_k \geq 0.5 \end{cases}$$

8.2.3 Cost Function

The objective of training the model is such that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This concept is captured through the cost function shown below.

$$J(\boldsymbol{w}) = \begin{cases} -log(\pi_k) & y = 1 \\ -log(1 - \pi_k) & y = 0 \end{cases}$$

This makes intuitive sense because $-log(t)$ grows very large when t approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance. The cost will also be very large if the model estimates a probability close to 1 for a negative instance.

On the other hand $-log(t)$ is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance.

We can express the cost as a single expression called the *log loss*.

$$J(\boldsymbol{w}) = -\frac{1}{N}\sum_{n=1}^{N}[y_n log(h_{\boldsymbol{w}}(\overline{\boldsymbol{x}}_n)) + ((1-y_n)log(1-h_{\boldsymbol{w}}(\overline{\boldsymbol{x}}_n))]$$

8.2.4 Solution

Unfortunately, there is no known closed-form solution to compute the value of $\boldsymbol{w}$ that minimizes the cost function. The cost function however, is convex, so Gradient Descent or any other convex optimization algorithm is guaranteed to find the global minimum. The gradient can be expressed as:

$$\frac{\partial J\boldsymbol{w}}{\partial \boldsymbol{w}_j} = \frac{1}{N}\sum_{n=1}^{N}(\sigma(\boldsymbol{w}^T\overline{\boldsymbol{x}}) - y_n)x_{j,n})$$

Some faster more sophisticated methods are

- Conjugate Gradient
- BFGS
- L-BFGS

8.2.5 Softmax Regression

The Logistic Regression model can be generalized to support multiple classes. When given an instance $\boldsymbol{x}$, the Softmax Regression model computes a score $f_k(\boldsymbol{x})$ for each class k, then estimates the probability of each class by applying the *softmax function* to the scores.

$$f_k(\boldsymbol{x}) = \boldsymbol{w}_k^T\overline{\boldsymbol{x}}$$

Once the score of every class for the instance $\boldsymbol{x}$ is computed, you can estimate the probability π_k that the instance belongs to class k. The function computes the exponential of each score, the normalizes them.

$$\pi_k = P(y_n = k|\boldsymbol{x}_n, \boldsymbol{w}) = \frac{e^{f_k(\boldsymbol{x})}}{\sum_{j=1}^{K} e^{f_j(\boldsymbol{x})}}$$

The Softmax Regression classifier predicts the class with the highest estimated probability.

The cost function associated with the Softmax Regression Classifier is the Cross Entropy cost function; it penalizes the model when it estimates a low probability for a target class. Cross entropy is used to measure how well a set of estimates class probabilities matches the target class. The cost function is represented as such

$$J(\boldsymbol{w}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}(y_n == k)log(\pi_k^{(n)})$$

where $y_n == k$ is the target probability the n^{th} instance belongs to class k and $\pi_k^{(n)}$ is the estimated probability that instance $\boldsymbol{x}_n$ belongs to class k.

The gradient vector is

$$\nabla_{\boldsymbol{w}_k} J(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} (\pi_k^{(n)} - (y_n == k))\boldsymbol{x}_n$$

that can be paired with an optimization algorithm to solve.

8.3 Generalized Linear Models

Often times our data won't be linear and it could be of a higher degree polynomial or a completely different distribution altogether. We can turn this non-linear problem into a linear regression problem by mapping the data to a different vector space using a basis function.

To demonstrate, let us consider Linear Regression on a nonlinear M x 1 (feature) dataset. Let ϕ denote the polynomial basis function where $\phi_j(\boldsymbol{x}) = x^j$. Then we can express our hypothesis as:

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = w_0\phi(x) + w_1\phi_1(x) + w_2\phi_2(x) + ... + w_m\phi_m(x)$$

A dataset with 3 features with a polynomial basis would have a hypothesis as such

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1^2x_2 + w_6x_1x_2^2 + w_7x_1^2x_2^2 + w_8x_1^3 + w_9x_2^3$$

This can then be extrapolated to logisitic regression and m-features. Some commonly used basis functions are:

- Polynomial: $\phi_j(\boldsymbol{x}) = x^j$
- Gaussian: $\phi_j(\boldsymbol{x}) = e^{(\frac{x-\mu_j}{2s^2})}$
- Sigmoid: $\phi_j(\boldsymbol{x}) = \sigma(\frac{x-\mu_j}{s})$
- Fourier Basis, Wavelets, etc ...

8.4 Regularization

Small outliers can drastically change our values of $\boldsymbol{w}$ so rely on regularization to reduce overfitting. Polynomial models can be easily regularized by reducing the number of polynomial degrees. For a linear model, regularization is typically achieved by constraining the weights of the model. The regularization term should only be added to the cost function during training. Once the model is trained, the non-regularized cost should be used to measure the model's performance. The bias term w_0 is not regularized.

8.4.1 Ridge Regression

Ridge Regression (Tikhonov Regularization) is a regularized version of Linear regression with a regularization term of $\frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$ (l_2-norm) added to the cost function. This forces the learning algorithm to fit the data but also keep the model weights as small as possible. The hyperparameter λ controls how much you want to regularize the model.

$$J(\boldsymbol{w}) = ERROR(\boldsymbol{w}) + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$$

8.4.2 Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression is another regularized version of Linear Regression, it adds a regularization term to the cost function but uses the l_1 norm of the weight vector instead of half the square of the l_2 norm.

$$J(\boldsymbol{w}) = ERROR(\boldsymbol{w}) + \lambda \sum_{i=1}^{n} |\boldsymbol{w}_i|$$

An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e, set them to zero). Lasso Regression automatically performs feature selection and outputs a *sparse model*.

8.4.3 Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms and you can control the mix ratio r. When $r = 0$, Elastic Net is equivalent to Ridge Regression, and when $r = 1$, it is equivalent to Lasso Regression.

$$J(\boldsymbol{w}) = ERROR(\boldsymbol{w}) + \frac{(1-r)\lambda}{2}\|\boldsymbol{w}\|_2^2 + r\lambda \sum_{i=1}^{n} |\boldsymbol{w}_i|$$

8.4.4 Early Stopping

Early Stopping is a different way to regularize iterative learning algorithms such as Gradient Descent. This method aims to stop training as soon as the validation error reaches a minimum. For all convex optimization problems there will be a global minima, once that global minima is reached the curve will start going up. This proposes to stop as soong as we reach the minimum.

9 Kernel Methods

9.1 Kernel Trick

9.1.1 Formulation

When we consider generalized linear models we have to come up with some basis functions, our hypothesis space is limited because we have fixed basis functions. To have a non-limited hypothesis space we need to be able to consider an infinite number of basis functions. Using kernel trick we can change the complexity of the problem to depend on the number of data rather than the number of basis functions.

Examples:

- Gaussian Processes
- Support Vector Machine

9.1.2 Dual Problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*. The dual problem is generally achieved by constraint based optimization (e.g taking the lagrangian and representing the problem as an optimization in other simpler variables). The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can have the same solution as the primal problem. The complexity of the primal solution depends on the number of basis functions while the complexity of the dual problem depends on the number of data points.

9.1.3 Kernel Function

Let $\phi(\boldsymbol{x})$ be a set of basis functions that map inputs x to a feature space. In many cases this feature space only appears in the dot product $\phi(\boldsymbol{x})^T\phi(\boldsymbol{x}')$ of input pairs $\boldsymbol{x}, \boldsymbol{x}'$, therefore we can define the kernel function

$$k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T\phi(\boldsymbol{x}')$$

to be the dot product of any pair $\boldsymbol{x}, \boldsymbol{x}'$ in feature space. Now we only need to know $k(\boldsymbol{x}, \boldsymbol{x}')$, not $\phi(\boldsymbol{x})$. If we know $k(\boldsymbol{x}, \boldsymbol{x}')$ then we know the output to any input $\boldsymbol{x}, \boldsymbol{x}'$ without having to compute $\phi(\boldsymbol{x})$. Intuitively a kernel is a measure of the similarity of the input.

9.1.4 Constructing Kernels

Two main methods:

- Find mapping $\boldsymbol{\phi}$ to feature space and let $\boldsymbol{K} = \boldsymbol{\phi^T\phi}$
- Directly specify K

A valid kernel must be a positive semi-definite. This means that k must factor into the product of a transposed matrix by itself (e.g., $\boldsymbol{K} = \boldsymbol{\phi^T\phi}$) or, all eigenvalues must be greater than or equal to 0.

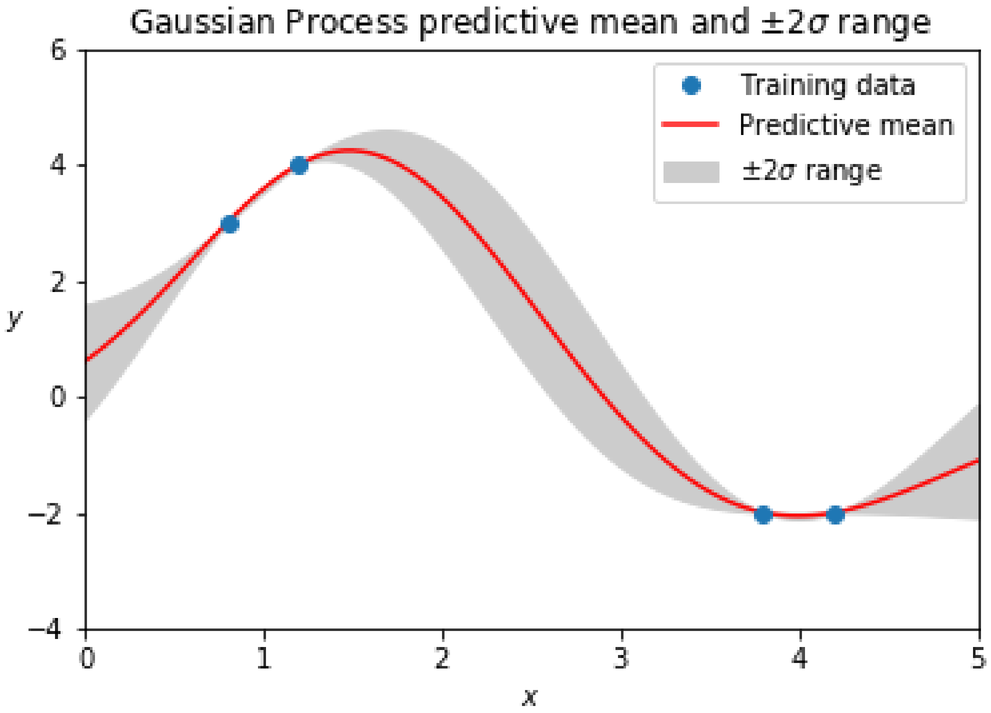
9.1.5 Common Kernels

- Linear/Identity: $k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T\boldsymbol{x}'$
- Polynomial: $k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}' + 1)^d$

- Gaussian RBF: $k(\boldsymbol{x}, \boldsymbol{x}') = exp(-\gamma\|\boldsymbol{x} - \boldsymbol{x}'\|^2)$
- Sigmoid Kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = tanh(\alpha\boldsymbol{x}^T\boldsymbol{x}' + c)$

9.2 Gaussian Processes

The idea behind Gaussian processes is that given a dataset it is able to capture any function that happens to be going through those points. It is does not assume any parametric form for the underlying data. The gaussian process essentially learns a distribution over what is the outcome of that function at any point. Between two points in the dataset it does not know what the function should look like, so it models the uncertainty of the prediction with a distribution between the data points, the bounds being + and - some multiple of standard deviation.



9.2.1 Function Space

In Bayesian Linear Regression we saw that it was a parametric form of learning the weight $\boldsymbol{w}$. Now instead we can consider the function space view were we can instead directly learn the function $f(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x})$. We can express this function space view as:

- Prior: $P(f(\boldsymbol{x}_*)) = \int_{\boldsymbol{w}} P(f|\boldsymbol{w}, \boldsymbol{x}_*)P(\boldsymbol{w}))d\boldsymbol{w}$
- Posterior: $P(f(\boldsymbol{x}_*)|\boldsymbol{X}, \boldsymbol{y}) = \int_{\boldsymbol{w}} P(f|\boldsymbol{w}, \boldsymbol{x}_*)P(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})d\boldsymbol{w}$

According to every function view, there is a Gaussian at $f(\boldsymbol{x}_*)$ for every $\boldsymbol{x}_*$. Those Gaussians are correlated through $\boldsymbol{w}$.

9.2.2 Representation

We can represent a Gaussian Process, a distribution over functions as:

$$f(\boldsymbol{x}) \sim GP(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}'))\forall \boldsymbol{x}, \boldsymbol{x}'$$

where $m(\boldsymbol{x}) = E(f(\boldsymbol{x}))$ is the mean and $k(\boldsymbol{x}, \boldsymbol{x}') = E((f(\boldsymbol{x} - m(\boldsymbol{x})))(f(\boldsymbol{x}') - m(\boldsymbol{x}')))$ is the kernel covariance matrix.

9.2.3 Gaussian Process Regression

The gaussian process regression is the kernel version of Bayesian Linear Regression where we learn based on the function space view, computing the posterior over f rather than over w. This allows for the complexity to be cubic in the number of training points rather than the number of features. We can perform regression using the following formulae.

Prior: $P(f(\cdot)) = N(m(\cdot), k(\cdot, \cdot))$

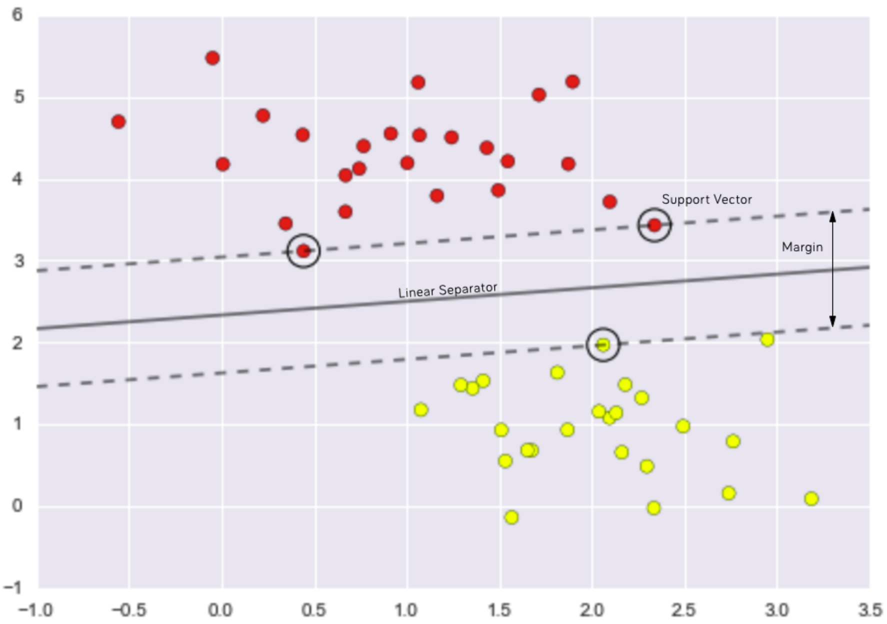
Likelihood: $P(\boldsymbol{y}|\boldsymbol{X}, f) = N(f(\cdot), \sigma^2 \boldsymbol{I})$

Posterior: $P(f(\cdot)|\boldsymbol{X}, \boldsymbol{y}) = N(\overline{f}(\cdot), k'(\cdot, \cdot))$ where
$\overline{f}(\cdot) = k(\cdot, \boldsymbol{X})(\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1}\boldsymbol{y}$ and
$k'(\cdot, \cdot) = k(\cdot, \cdot) - k(\cdot, \boldsymbol{X})(\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} k(\boldsymbol{X}, \cdot)$

Prediction: $P(\boldsymbol{y}_*|\boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}) = N(\overline{f}(\boldsymbol{x}_*), k'(\boldsymbol{x}_*, \boldsymbol{x}_*))$

9.3 Support Vector Machines

Support Vector Machines is a kernel based method that can be used for classification and regression. It performs extremely well for small amounts of data and can even beat out Neural Networks.



9.3.1 Max-Margin Classifier

Max-Margin classifier is generally only used for binary classification where the data is linearly separable. The intuition behind the max-margin classifier is that we want to find a linear separator for the data that maximizes the distance to the closest data points. We formulate this method as we need to consider noise, a larger margin will allow room for noise, otherwise if it is too narrow then there is room for possible misclassification.

We can turn the problem into an optimization problem of finding the max margin as follows.

Linear Separator: $\boldsymbol{w}^T\phi(\boldsymbol{x}) = 0$.

Distance to Linear Separator:

$$\frac{y\boldsymbol{w}^T\phi(x)}{\|\boldsymbol{w}\|}, y \in \{-1, 1\}$$

Maximum Margin:

$$max_{\boldsymbol{w}}\frac{1}{\|\boldsymbol{w}\|}\{min_n y_n \boldsymbol{w}^T\phi(\boldsymbol{x}_n)\}$$

We can transform this expression by fixing the minimal distance to 1 and minimizing our scale $\|\boldsymbol{w}\|$ to be:

$$min_w \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ s.t. } y_n\boldsymbol{w}^T\phi(\boldsymbol{x}_n) \geq 1 \; \forall n$$

This now becomes a convex quadratic optimization problem with linear constraints, which is a form of that is quite easy. The points where $y_n\boldsymbol{w}^T\phi(\boldsymbol{x}_n) = 1$ define the active constraints, called the support vectors.

9.3.1.1 Dual Representation

To compute this problem we want to reformulate such that $\phi(\boldsymbol{x})$ only appears in a kernel. From the kernel trick we see we can achieve this by finding the dual of the optimization. This will result in a sparse kernel since only the points on the margin matter.

We want to transform the constrained optimization problem

$$min_w \frac{1}{2}\|\boldsymbol{w}\|^2 \text{ s.t. } y_n\boldsymbol{w}^T\phi(\boldsymbol{x}_n) \geq 1 \; \forall n$$

into an unconstrained optimization problem. We can use the lagrangian to obtain the following.

$$max_{a\geq 0} \; min_{\boldsymbol{w}} L(\boldsymbol{w}, \boldsymbol{a})$$

where

$$L(\boldsymbol{w}, \boldsymbol{a}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_n a_n[y_n\boldsymbol{w}^T\phi(\boldsymbol{x}_n) - 1]$$

where the second term is the penalty for violating the n^th constraint.

We can then solve the inner minimization $min_{\boldsymbol{w}} L(\boldsymbol{w}, \boldsymbol{a})$ to obtain:

$$L(\boldsymbol{a}) = \sum_n a_n - \frac{1}{2} \sum_n \sum_n a_n a_n, y_n y_n, k(\boldsymbol{x}_n, \boldsymbol{x}_n)$$

We are then left with the optimization problem in $\boldsymbol{a}$ only known as the dual problem.

$$max_{\boldsymbol{a}} L(\boldsymbol{a}) \text{ s.t. } a_n \geq 0$$

This is sparse optimization since many of the a_n's are 0 when the data point is already ≥ 1, so the penalty is 0 for theses since they already satisfy the constraint.

9.3.1.2 Classification

Primal Problem:

$$y_* = sgn(\boldsymbol{w}^T \phi(\boldsymbol{x}_*))$$

Dual Problem:

$$y_* = sgn(\sum_n a_n y_n \phi(\boldsymbol{x}_n)^T \phi(\boldsymbol{x}_*))$$
$$y_* = sgn(\sum_n a_n y_n k(\boldsymbol{x}_n, \boldsymbol{x}_*))$$

Intuitively what is happening here is we are taking the sum of the degree of similarity between each query point and every point in the training set that is a support vector.

9.3.2 Soft Margin Classifier

Often times the data is not linearly separable and we have overlapping class distributions. We want a method such that we can relax the constraints yet keep the maximum margin as maximizing the margin is equivalent to minimizing an upper bound on the worst case loss. To achieve this we introduce Soft Margins which formulates that we can relax the constraints by introducing a slack variable $\xi \geq 0$. We can now impose the following constraint.

$$y_n \boldsymbol{w}^T \phi(\boldsymbol{x}_n) \geq 1 - \xi_n \ \forall n$$

This introduces a new optimization problem

$$min_{\boldsymbol{w},\boldsymbol{\xi}} \ C \sum_{n=1}^{N} \xi_n + \frac{1}{2} \|\boldsymbol{w}\|^2 \quad \text{s.t. } y_n \boldsymbol{w}^T \phi(\boldsymbol{x}_n) \geq 1 - \xi_n, \ \xi_n \geq 0 \ \forall n$$

$C > 0$ controls the tradeoff between the slack variable penalty and the margin.

Some intuition behind this is further explored.

- Since $\sum_n \xi_n$ is an upper bound on the number of misclassifications, C can also be thought as a regularized coefficient that controls the tradeoff between error minimization and model complexity
- We can see that if we let $C \to \infty$ then we recover the original hard margin problem.
- Soft margins can only handle minor misclassifications and are still sensitive to outliers.

9.3.3 Multiclass SVM

There are many approaches to train multi class SVM's the best one being the continuous ranking approach. The idea behind this, is that instead of computing the sign of a linear separator, we compare the values of linear functions for each class k. The SVM returns a continuous value to rank all classes.

9.3.3.1 Constraint

Now for each class $k \neq y$ we define a linear constraint that guarantees a margin of at least 1 between classes.

$$\boldsymbol{w}_y^T \phi(\boldsymbol{x}) - \boldsymbol{w}_k^T \phi(\boldsymbol{x}) \geq 1 \ \forall k \neq y$$

9.3.3.2 Classification

With the constraint we can achieve the optimization problem used for classification. For multiclass dataset that is linearly separable we get:

$$min_{\boldsymbol{w}} \frac{1}{2} \sum_k \|\boldsymbol{w}_k\|^2 \quad \text{s.t. } \boldsymbol{w}_{y_n}^T \phi(\boldsymbol{x_n}) - \boldsymbol{w}_k^T \phi(\boldsymbol{x}_n) \geq 1 \ \forall n, k \neq y_n$$

For overlapping classes we can add the slack variable ξ

$$min_{\boldsymbol{w},\boldsymbol{\xi}} \ C \sum_n \xi_n + \frac{1}{2} \sum_k \|\boldsymbol{w}_k\|^2 \quad \text{s.t. } \boldsymbol{w}_{y_n}^T \phi(\boldsymbol{x_n}) - \boldsymbol{w}_k^T \phi(\boldsymbol{x}_n) \geq 1 - \xi_n \ \forall n, k \neq y_n$$

10 Artificial Neural Networks Primer

10.1 Origins

The concept of a Artificial Neural Network (ANN) stems from the anatomy of the brain. They are modelled directly after biological neurons, a neural cell, found in the brain. Biological neurons produce short electrical impulses called *action potentials* which travel along the neurons and make synapses release chemical signals called *neurotransmitters*. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses. These, individual neurons behave in a simple way but they are organized in a vast network of billions, with each neuron typically connect to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons.

10.2 ANN Unit

Now the idea behind Artificial Neural Networks is to mimic the brain by making Artificial Neurons. The Perceptron is one of the simplest ANN architectures and is based on a slightly different artificial neuron called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The TLU computes a weighted sum of its inputs ($a = w_1x_1 + w_2x_2 + ... + w_nx_n = \boldsymbol{w}^T\boldsymbol{x}$), then applies an activation function to that sum and outputs the result: $h_{\boldsymbol{w}}(a)$. When picking an activation function, it should be nonlinear, otherwise the network is just a linear function. The activation function should be chosen such that it mimics firing in neurons: the unit should be "active" (output near 1) when fed with the "right" inputs and the unit should be "inactive" (output near 0) when fed with the wrong inputs.

10.3 Perceptron

A perceptron is a type of single layer feed-forward network. It is simply composed of a single layer of threshold logic units, with each TLU connected to all the inputs. When all the neurons in a layer are connected to every neuron in the previous later, the layer is called a *fully connected layer*, or a *dense layer*. The inputs of the perceptron are fed to special pass through neurons called input neurons: they output whatever input they are fed. All the input neurons form the *input layer*. Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a *bias neuron*, which outputs 1 all the time.

10.3.1 Threshold Perceptron Learning

10.3.1.1 Threshold Perceptron Algorithm

For threshold perceptron algorithm, learning is done separately for each unit j since units do not share weights. The learning algorithm is as follows, for each unit j, for each $(\boldsymbol{x}, \boldsymbol{y})$ pair do until the output is correct for all training instances:

- Case 1: Correct output produced then $\forall i W_{ji} \leftarrow W_{ji}$
- Case 2: Output produced 0 instead of 1 then $\forall i W_{ji} + x_i$
- Case 3: Output produced is 1 instead of 0 then $\forall i W_{ji} - x_i$

Now we will demonstrate the intuition behind this. If we consider using a threshold activation function then the perceptron computes:

- 1 when $\boldsymbol{w}^T\overline{\boldsymbol{x}} = \sum_i x_iw_i + w_0 > 0$
- 0 when $\boldsymbol{w}^T\overline{\boldsymbol{x}} = \sum_i x_iw_i + w_0 < 0$

Now leveraging the fact that $\overline{\boldsymbol{x}}^T\overline{\boldsymbol{x}} \geq 0$ and $\overline{\boldsymbol{x}}^T\overline{\boldsymbol{x}} \leq 0$ then we can we can come up with the following statements.

- If the output should be 1 instead of 0 then $\boldsymbol{w} \leftarrow \boldsymbol{w} + \overline{\boldsymbol{x}}$ since $(\boldsymbol{w} + \overline{\boldsymbol{x}})^T\overline{\boldsymbol{x}} \geq \boldsymbol{w}^T\overline{\boldsymbol{x}}$
- If the output should be 0 instead of 1 then $\boldsymbol{w} \leftarrow \boldsymbol{w} - \overline{\boldsymbol{x}}$ since $(\boldsymbol{w} - \overline{\boldsymbol{x}})^T\overline{\boldsymbol{x}} \leq \boldsymbol{w}^T\overline{\boldsymbol{x}}$

10.3.1.2 Sequential Gradient Descent Algorithm

Alternatively we can use gradient descent to minimize misclassification error to train the perceptron.

Let $y \in \{-1, 1\} \forall y$ and let $M = \{(\boldsymbol{x}_n, y_n)_{\forall n}\}$ be the set of misclassified examples (i.e., $y_n \boldsymbol{w}^T \overline{\boldsymbol{x}}_n < 0$).

We need to find a $\boldsymbol{w}$ that minimizes the misclassification error

$$E(\boldsymbol{w}) = - \sum_{(\boldsymbol{x}_n, y_n) \in M} y_n \boldsymbol{w}^T \overline{\boldsymbol{x}}_n$$

and the gradient is:

$$\boldsymbol{\nabla E} = - \sum_{(\boldsymbol{x}_n, y_n) \in M} y_n \overline{\boldsymbol{x}}_n$$

Now applying this to the gradient descent algorithm we have:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha y \overline{\boldsymbol{x}}$$

We adjust $\boldsymbol{w}$ one sample at a time. Here α is the learning rate and we see that if we let $\alpha = 1$ then we recover the threshold perceptron algorithm.

10.3.1.3 Limitations

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex However, if the training instances are linearly separable then this algorithm will converge to a solution.

10.3.2 Sigmoid Perceptron Learning

Similar to Threshold perceptron learning, Sigmoid Perceptron Learning hinges on the same concepts. We can set our objective to minimizing the minimum squared error or maximum likelihood which will yield the same algorithm as for logistic regression.

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_n E_n(\boldsymbol{w})^2 = \frac{1}{2} \sum_n (y_n - \sigma(\boldsymbol{w}^T \overline{\boldsymbol{x}}_n))^2$$

We can compute the gradient to be

$$\boldsymbol{\nabla E} = - \sum_n E_n(\boldsymbol{w}) \sigma(\boldsymbol{w}^T \overline{\boldsymbol{x}}_n)(1 - \sigma(\boldsymbol{w}^T \overline{\boldsymbol{x}}_n)) x_i$$

Now using sequential gradient descent we repeat the following for each $(\boldsymbol{x}_n, y_n)$ until some stopping criterion is satisfied.

$$\epsilon_n \leftarrow y_n - \sigma(\boldsymbol{w}^T \overline{\boldsymbol{x}}_n)$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \epsilon_n \sigma(\boldsymbol{w}^T \overline{\boldsymbol{x}}_n)(1 - \sigma(\boldsymbol{w}^T \overline{\boldsymbol{x}}_n)) \overline{\boldsymbol{x}}_n$$

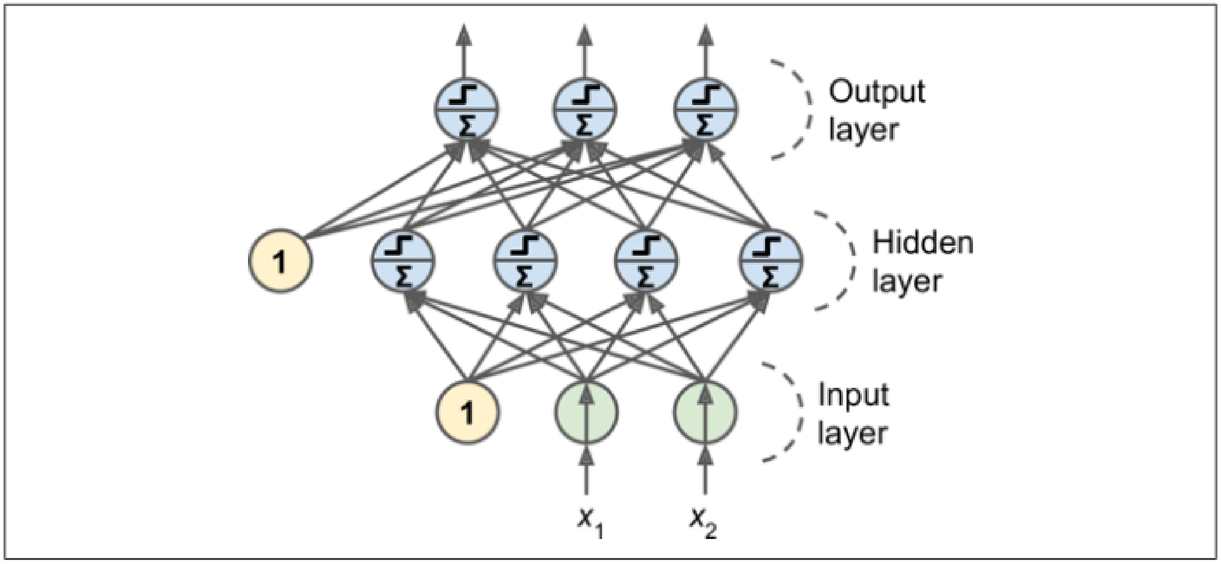
It possesses the same limitations as threshold perceptron learning.

10.4 Multi-Layer Neural Nets

We previously saw the limitations of the perceptron as it was only able to learn on linearly separable data. Unfortunately a lot of data is not linear and this led to the birth of Multi-layer Neural Networks: which are able to learn non-linear basis functions.

10.4.1 n-Layer Perceptron

The follow diagram depicts a 2 layer perceptron. Multilayer perceptron are composed of one (passthrough) *input layer*, one or more layers of ANN's units called *hidden layers*, and one final layer of ANN units called the *output layer*. The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.



Let us denote the hidden units with $\boldsymbol{z}$, the output units with $\boldsymbol{y}$ and the weights between the layers with $\boldsymbol{w}^(layer_num)$.

Hidden Units: $z_j = h_1(\boldsymbol{w}_j^{(1)}\overline{\boldsymbol{x}})$

Output Units: $y_k = h_2(\boldsymbol{w}_k^{(2)}\overline{\boldsymbol{z}})$

Overall: $y_k = h_2(\sum_j w_{kj}^{(2)} h_1(\sum_i w_{ji}^{(1)} x_i))$

10.4.1.1 Non-linear Regression

We can use a multi-layer neural network to equivalently represent regression with the following expression.

$$y_k = \sum_j w_{kj}^{(2)} \sigma(\sum_i w_{ji} x_i)$$

We can interpret the sigmoid as a non linear basis function and the outer summation to be the linear combination.

10.4.1.2 Non-linear Classification

We can use a multi-layer neural network to equivalently represent binary classification with the following expression.

$$P(c_k|\boldsymbol{x}) = \sigma(\sum_j w_{kj}^{(2)} \sigma(\sum_i w_{ji} x_i))$$

We can interpret the sigmoid as a non-linear basis function and the outer summation to be the linear combination. The outer sigmoid is the normalization to return the probability.

What is happening in both regression and classification is we are allowing the basis functions to adapt and vary and we are no longer restricted to fixed basis functions.

10.4.2 Backpropagation

One of the most common forms of weight training for multi-layer neural nets is error minimization using backpropagation. This allows us to compare the errors at the output and backpropagate the error back through the error through the network to train the weights.

We can then use gradient descent to to adjust the weights. Generally based on the size of the model and the data it is more favorable to use faster gradient descent algorithms and we can consider gradient descent optimizations for training.

10.4.2.1 Algorithm

Forward Phase: Propagate units forward to compute the output of each unit. We want to obtain the output z_j for each unit j.

$$z_j = h(a_j) \quad \text{where} \quad a_j = \sum_i w_{ji} z_i$$

Backward Phase: compute delta δ_j at each unit j. We can use chain rule to recursively compute the gradient and follow the following process.

For each weight w_{ji}

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$$

Let $\delta_j = \frac{\partial E_n}{\partial a_j}$ then

$$\delta_j = \begin{cases} h'(a_j)(z_j - y_j) & \text{base case: } j \text{ is an output unit} \\ h'(a_j) \sum_k w_k \delta_k & \text{recursion: } j \text{ is a hidden unit} \end{cases}$$

Since $a_j = \sum_i w_{ji} z_i$ then $\frac{\partial a_j}{\partial w_{ji}} = z_i$.

11 Deep Learning

A Deep Neural Network is a neural network with many hidden layers. The main advantage of a DNN is its high expressivity, it is able to learn very complex underlying functions. As we increase the number of layers, the number of ANN units needed may decrease (with the number of layers). The power and basis of deep learning is that instead of having to have domain knowledge about what features to apply machine learning to, deep neural networks are able to learn hierarchical feature representations. For example for facial classification the first hidden layer detects certain strokes, the next is able to detect facial features such as eyes and noses and so on until it is able to reconstruct and detect the face.

11.1 Training

11.1.1 Vanishing/Exploding Gradients Problem

One of the problems of backpropagation in deep neural networks consisting of sigmoid and hyperbolic units is that they often suffer from *vanishing gradients*. The problem here is that the computed gradients get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. This is because when moving in the forward phase the variance keeps increasing after each layer until the activation saturates at the top layer when using the sigmoid or tanh activation function. When the function saturates at 0 or 1, with a derivative extremely close to 0, then when backpropagation kicks in it has virtually no gradient to propagate back through the network. The little gradient that exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

11.1.1.1 Popular Solutions

- Pre-training
- Recfitified Linear Units and Maxout Units
- Skip Connections
- Batch Normalization

11.1.2 Dropout

Since deep neural networks are so highly expressive this increases the risk of overfitting. Often times the number of parameters is larger than the amount of data. We can use a technique called dropout to help mitigate overfitting.

The idea behind dropout is to randomly "drop" some units from the network when training, this effectively is the same as saying to reduce their values to 0. Now we train the model with missing nodes (that could represent features) making the network robust if it is able to still perform well and making it impervious to overfitting since it can perform well with some features removes. In each training iteration, a different subnetwork is trained. At test time, these subnetworks are merged by averaging their weights.

At training during each iteration of gradient descent:

- Each input unit is dropped with probability p_1
- Each hidden unit is dropped with probability p_2.

For prediction, since we scaled down the inputs by probability p_1 and the hidden units by p_2, so we need to multiply by their complementary probability to increase their magnitude accordingly.

- Multiply each input unit by $1 - p_1$
- Multiply each hidden unit by $1 - p_2$

11.1.2.1 Algorithm

Let $\odot$ denote elementwise multiplication. For each training example $(\boldsymbol{x}_n, y_n)$ do

```
    Sample $\boldsymbol{z}_n^{(1)}$ from Bernoulli$(1 - p_i)^{k_i}$ for $1 \leq l \leq L$
    Neural Network with dropout applied:
    $f_n(\boldsymbol{x}_n, \boldsymbol{z}_n; \boldsymbol{W}) = h_i(\boldsymbol{W}^{(L)}[...h_2(\boldsymbol{W}^{(2)}[h_1(\boldsymbol{W}^{(1)}[\overline{\boldsymbol{x}}_n \odot \boldsymbol{z}_n^{(1)}]) \odot \boldsymbol{z}_{\boldsymbol{n}}{}^{(2)}])... \odot \boldsymbol{z}_n^{(L)}])$
    Loss: $Err(y_n, f_n(\boldsymbol{x}_n, \boldsymbol{z}_n; \boldsymbol{W}))$
    Update: $w_{kj} \leftarrow w_{kj} - \alpha \frac{\partial Err}{\partial w_{kj}}$
```

Until convergence.

Prediction:

$$f(\boldsymbol{x}_n; \boldsymbol{W}) = h_i(\boldsymbol{W}^{(L)}[...h_2(\boldsymbol{W}^{(2)}[h_1(\boldsymbol{W}^{(1)}[\overline{\boldsymbol{x}}_n(1 - p_1)])(1 - p_2)])...(1 - p_L)])$$