Data Science & Machine Learning Concept Overview

Omar Abdel Haq

Chapter 1

Basics

1.1 Types of Learning

Machine learning and data science are great for solving complex problems which either have no algorithmic solution or would require long lists of hand tuned rules, creating models that are relatively easier to build and which can adapt to fluctuating environments. There are multiple aspects using which we can classify machine learning algorithms:

Supervised vs. Unsupervised Learning

Supervised Learning: The training set you feed an algorithm includes the desired solutions, termed labels. Examples include regression and classification.

Unsupervised Learning: The training set is unlabeled, and so the algorithm tries to learn without a teacher/instruction. Examples include clustering and dimensionality reduction.

Instance vs. Model-Based Learning

Instance-Based Learning: Make predictions based on finding the most similar available data points, using some similarity measure, and making composite predictions based on these points.

Model-Based Learning: Based on a model tuned to all of the training data, where predictions on new data are made by using that model.

Batch vs. Incremental Learning

Batch Learning: A type of learning where the system is incapable of learning incrementally, needing to be trained at once using all of the available data. The problem with this type of learning is model rot or data drift.

Incremental Learning: An approach where the model learns from new data instances incrementally over time. Instead of training on the entire dataset in one go, the model is trained on small subsets or individual data points sequentially as they become available.

Regression vs. Classification Problems (*Within Supervised Learning*)

Regression: In regression problems, the goal is to predict a continuous or numerical output variable. The task involves finding a relationship or mapping between input features and a target variable that can take on any value within a specific range. The output in regression is typically a real-valued number or a set of numbers.

Classification: In classification problems, the goal is to assign input instances to predefined categories or classes. The task involves learning a decision boundary that separates different classes based on the input features. The output in classification is discrete and represents the predicted class membership of the input.

In addition to these categories of learning/problem types, one important approach in machine learning is **Reinforcement Learning**. With reinforcement learning, the learning dynamic is based on an agent, which, in a select environment can pick and perform different actions, and based on these actions receive rewards or penalties. It attempts to learn by finding the optimal strategy, called a policy.

1.2 Model Definitions

These are the main definitions associated with most machine learning models:

- *Parameter*: Values that determine the predictions a model makes on new data.
- *Hyperparameter*: Values that are integral to the learning algorithm itself.
- *Predictor Variable*: Feature used to predict a response variable. The set of predictor features, as a dataframe, is usually referred to as the *design matrix*.
- *Response Variable*: Feature algorithm intends to predict.

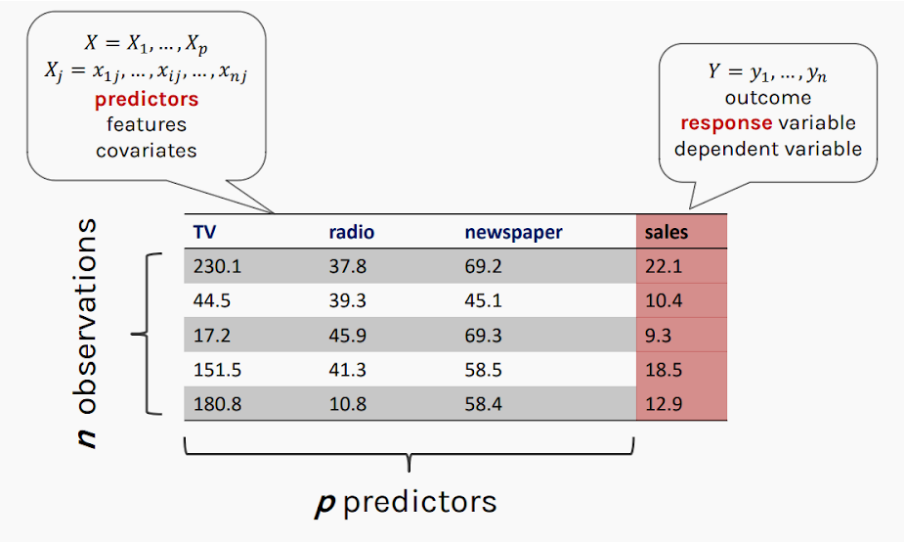


Figure 1.1: Predictor & Response Variables

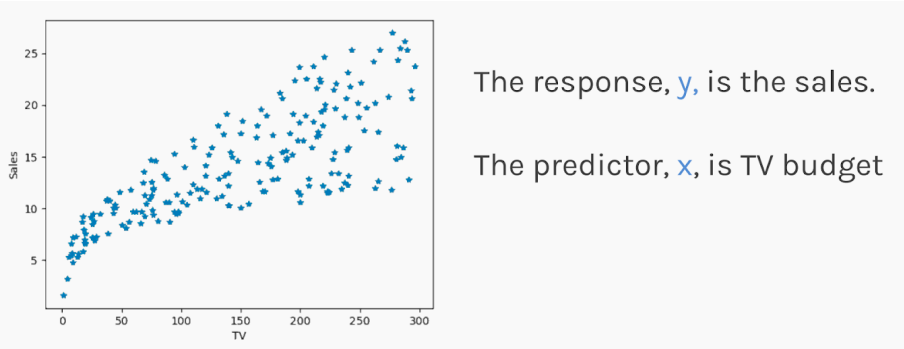


Figure 1.2: Example Predictor & Response Variables

1.3 Project Framework

1.3.1 Problem Framing

Before starting the main steps of any project, including feature engineering and model fitting, it is important to ask the following questions to figure out what you need to do:

- What is the question are you looking to answer?
- What is the current protocol/solution in place for this type of problem?
- What is the incentive behind the project?
- What type of training supervision will be needed?
- What type of problem are we dealing with; classification, regression, or something else?

Once you have an intuition on what the answers to these questions should be, we can move on to the next step, selecting a model performance measure.

1.3.2 Data Exploration & Cleaning

At this point, it is important to examine the data at hand, to determine if any transformations are necessary.

Exploration

- Look at which columns you have, which types they are, and how many null values you have and in which columns.
- Plot histograms of each value to find outliers, distributions, and irregularities.
- Plot location data (long. & lat. coordinate pairs).
- Plot the response vs. each numeric variable individually, looking for strong correlations.

Cleaning

Sometimes, data entry errors occur (or ones in units of measurement), which results in extreme outliers in some features. In this case, it is important to drop the outliers in question. There are three ways to go about doing so:

- Remove values known to be impossible (like a human body temperature of 150 degrees Fahrenheit.
- Mean/SD Approach: In this approach, the mean and standard deviation of a feature are computed. Observations that fall outside a certain range, typically defined as mean $\pm k \times$ SD (where k is a threshold value), are considered outliers and removed from the dataset. This approach is sensitive to extreme values because the standard deviation is influenced by outliers. This approach assumes that the data are normally distributed or at least approximately normally distributed. It works well when the data exhibit a symmetric bell-shaped distribution.
- Median/MAD Approach: In this approach, the median (a measure of central tendency) and the median absolute deviation (MAD) are calculated. MAD is the median of the absolute deviations from the median. Observations that deviate from the median by a certain threshold, typically defined as median $\pm k \times$ MAD, are considered outliers and removed. The median/MAD are more robust because they are less influenced by extreme values. The median/MAD approach is more suitable for datasets that deviate from normality or have skewed distributions.

Imputation

Sometimes (most times) our dataset will contain missing data. Depending on the situation, we may want to get rid of incomplete data, like when we have an abundance of data. At other times, we may not have the flexibility of deleting those data entries (when that results in a tiny dataset). There are three types of missingness in the context of data analysis:

1. **Missing Completely at Random (MCAR):** Data is considered to be Missing Completely at Random when the missingness occurs in a completely random and unpredictable manner, unrelated to any observed or unobserved variables.

2. **Missing at Random (MAR):** Missing at Random (MAR) occurs when the probability of missingness depends on observed variables but not on the missing values themselves. In other words, the missingness is related to observed information.

3. **Missing Not at Random (MNAR):** Missing Not at Random (MNAR) represents the situation where the missingness is related to unobserved or missing values. In this case, the missing data mechanism is driven by information that is not included in the dataset.

We can deal with missing data by imputing it, filling in values in blanks depending on the other predictors present (we here assume, that given a large number of predictors, our missingess would be at random – MAR). Here are the main types of imputation, and when to use each:

1. Mean Imputation: Mean imputation replaces missing values with the mean of the available values in the variable. Mean imputation is simple, but one drawback is that it could destroy the underlying distribution and the relationships between variables by creating an artificial spike at the group mean. This artificially lowers the estimated sampling variance of the final estimates.

2. Median Imputation: Median imputation replaces missing values with the median of the predictor's values. It is suitable for variables with outliers or highly skewed distributions. Median imputation is robust to extreme values, but doesn't preserve the mean or the shape of the distribution.

3. Mode Imputation: Mode imputation replaces missing values with the mode (most frequent value) of the variable. It is used for categorical or discrete variables. Although straightforward, it ignores any relationship between the variable and other variables in the dataset.

4. K-Nearest Neighbors (KNN) Imputation: KNN imputation uses the values of the k-nearest neighbors of a data point to impute missing values. It considers the overall patterns in the data and can be used for both numerical and categorical variables. KNN imputation works well when there are relationships between the missing variable and other variables in the dataset.

5. Iterative Imputation: Iterative imputation, also known as multiple imputation, is an advanced technique that imputes missing values by iteratively estimating them based on other variables in the dataset. It creates multiple imputed datasets to capture uncertainty.

Including a *missingness indicator variable* offers valuable benefits in handling missing data scenarios. This approach involves creating an additional binary variable that takes the value of 1 when a data point is missing and 0 when it's observed. Why do this? Because the group of individuals with a missing entry may be systematically different than those with that variable measured, and treating those as the same will result in underperformance in prediction.

Scaling

There are two ways to scale the data, which might be useful for some types of machine learning models, like neural networks.

1. Normalization (Min-Max Scaling): Normalization scales the values of a variable to a specific range, typically between 0 and 1. The process usually involves subtracting the minimum value of a feature from each observation, then dividing by the range of feature values. Normalization is primarily used when the scale of the variables varies widely, and you want to bring them to a similar range. It ensures that all variables contribute proportionately to the analysis, preventing dominant variables from overshadowing others. Normalization is commonly used in algorithms that rely on distance or similarity measures, such as k-nearest neighbors (KNN) or support vector machines (SVM). Normalization can also be beneficial when you want to restrict the range of variables between two values, especially if you're working with specific algorithms that are sensitive to the absolute values of the variables, such as neural networks with certain activation functions

2. Standardization: Standardization transforms the values of a variable to have a mean of 0 and a standard deviation of 1. The process involves subtracting the mean of a feature from each observation then dividing by the standard deviation. Standardization brings the variables onto a common scale, facilitating the interpretation and analysis of the data. It is commonly used in techniques that assume a normal distribution or when the algorithm or model requires variables to be on the same scale, such as linear regression, principal component analysis (PCA), or neural networks.

Both of these techniques squash values that have distributions with heavy tails, which isn't always ideal (as the normality assumption isn't always true). In these cases, it might be more beneficial to transform the features by first taking either the **logarithm** of all values, or replacing each value with its distribution **percentile** in the dataset.

Multimodal Features

When dealing with features that clearly seem to have multimodal distributions, it is useful to create features in accordance with that. After picking the number of modes you'd want to use, you can create new features, each representing a distance measure (using something like a Radial Basis Function) from each of the modes. This can give us insight on cluster affiliation, as different clusters within a distribution may have a different relationship with the response variable.

Image/Data Augmentation

An approach to increase the size and diversity of observations in the training set. It involves shifting an image by a small number of pixels to either the right, left, up, or down to create more observation instances. This helps the model distinguish between noise and signal by increasing the training set size (mainly by decreasing reliance on whether an image is centered or not to begin with).

Categorical Variables

When pre-processing categoriable features, it is best to first transform them into dummy variables using one-hot-encoding. One-Hot Encoding (OHE) is a technique used to convert categorical variables into a numerical representation that can be easily understood by machine learning algorithms. It creates binary variables (or "dummy variables") for each unique category in the original categorical variable (we can also get away with making 1 less dummy variable column than there are categorical variables). Each binary variable indicates the presence or absence of a specific category. OHE is useful in several scenarios:

- Machine learning algorithms typically work with numerical data, so categorical variables need to be transformed into a format that can be processed. OHE provides a way to represent categorical variables as binary vectors, enabling algorithms to handle them effectively.
- OHE maintains the distinctness of each category by creating separate binary variables for each unique value. This is important when the categories do not have an inherent order or hierarchy. It prevents the algorithm from assuming any ordinal relationship between categories.
- OHE is particularly useful for nominal variables, where the categories have no natural order. For example, if you have a categorical variable representing different colors (red, blue, green), OHE would create binary variables for each color, allowing algorithms to understand the presence or absence of each color for a given observation.

It is important, however, to make sure you cannot condense some variables via binning, as having a massive amount of one-hot-encoded features isn't ideal when it comes to fitting model.

Binning

Binning, also known as discretization, is the process of dividing continuous variables into a set of discrete bins or intervals. Each bin represents a range of values, and observations falling within that range are assigned the corresponding bin label. Binning can be useful in machine learning models in the following scenarios:

- Dealing with Non-Linear Relationships: Binning can help capture non-linear relationships between a continuous variable and the target variable. By dividing the variable into bins, the model can capture different patterns or trends within each bin separately. This can be particularly useful when the relationship between the continuous variable and the target is not linear and has complex patterns.
- Handling Outliers: Binning can be used to handle outliers by grouping extreme values into a separate bin. Outliers can have a disproportionate impact on the model's performance, and by placing them in their own bin, the model can treat them differently or mitigate their influence.
- Reducing Noise or Variability: Binning can help smooth out noise or reduce the impact of small variations in the data. By grouping similar values into bins, the model focuses on the general trend within each bin rather than individual values, which can lead to more stable predictions.
- Handling Computational Complexity: In some cases, binning can simplify the computational complexity of the model. Instead of dealing with a large number of unique values in a continuous variable, binning reduces the number of unique categories, making the model more computationally efficient.

Chapter 2

Metrics, Losses, & Distances

2.1 Definitions

A *loss function*, also known as a cost function or objective function, quantifies how well a model performs on the training data. It measures the discrepancy between the model's predictions and the actual target values (ground truth) during training. The primary goal of the model is to minimize this loss function, and achieving a low loss value indicates that the model is effectively learning to make accurate predictions.

An *error metric*, also known as an evaluation metric or performance metric, is a quantitative measure used to assess the performance of a trained model on unseen data. It provides valuable insights into the model's accuracy and how well it generalizes to real-world scenarios.

A distance or similarity measure quantifies the similarity between individual points in a dataset, which can be used in unsupervised learning settings.

2.2 Regression Metrics

1. Mean Squared Error (MSE): MSE is a popular error metric that measures the average squared difference between predicted and actual values. It is calculated by taking the mean of the squared residuals. MSE assigns larger penalties to larger errors due to the squaring operation. Its benefits also come from the mathematical properties of MSE (such as when using gradient-based optimization algorithms), since squaring is a differentiable function.

$$\text{Formula: } \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

2. Root Mean Squared Error (RMSE): RMSE is the square root of MSE, and it provides the measure of the average magnitude of the residuals in the same units as the dependent variable. RMSE is preferred when you want an error metric that is interpretable in the original units of the data. For example, if you are predicting housing prices, the RMSE would give you an estimate of the average dollar amount by which your predictions differ from the actual prices.

$$\text{Formula: } \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

3. Mean Absolute Error (MAE): MAE measures the average absolute difference between predicted and actual values. It is calculated by taking the mean of the absolute residuals. MAE is useful when you want an error metric that is not heavily influenced by outliers and penalizes errors linearly. If your dataset contains outliers and you don't want them to disproportionately affect your error measurement, MAE is a good choice.

$$\text{Formula: } \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

4. Max Absolute Error: Max Absolute Error is a metric commonly used in modeling to measure the maximum magnitude of the difference between predicted and actual values. It represents the largest vertical deviation between the predicted values and the true values. It is usually a good baseline metric, giving information on how poorly a model performs in the very worst case:

$$\text{Formula: } \max_i |y_i - \hat{y}_i|$$

5. R^2 (Coefficient of Determination): R^2 is a statistical measure that represents the proportion of variance in the dependent variable that can be explained by the independent variables in a regression model. It ranges from -1 to 1, where 1 indicates that the model explains all the variability in the data. R^2 is useful for understanding the goodness of fit of a regression model since it compares it to the baseline of predicting the mean all the time. However, it has limitations and should not be solely relied upon as it doesn't account for overfitting or the quality of predictions.

$$\text{Formula: } 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

R^2 can be negative, and this is possible when your model is worse than the average model (possible when evaluating on the test set).

6. Adjusted R^2: Adjusted R^2 is a modified version of R^2 that takes into account the number of predictors in the model and adjusts the value accordingly. It penalizes the addition of unnecessary predictors and helps to mitigate the issue of overfitting. Adjusted R^2 is preferable when comparing models with different numbers of predictors. It provides a more conservative evaluation of model performance compared to R^2.

$$\text{Formula: } 1 - \left(1 - R^2\right) \cdot \frac{n-1}{n-p-1}, \text{ where } p \text{ is the number of predictors}$$

The choice of which metric to use depends on the specific problem, the nature of the data, and the evaluation criteria. MSE and RMSE are commonly used when you want to penalize larger errors more strongly. MAE is preferred when you want a metric that is less sensitive to outliers. R^2 and adjusted R^2 are useful for understanding the overall fit and explanatory power of a regression model, but they don't capture all aspects of model performance. It is generally recommended to consider multiple metrics and their implications to gain a comprehensive understanding of your model's performance.

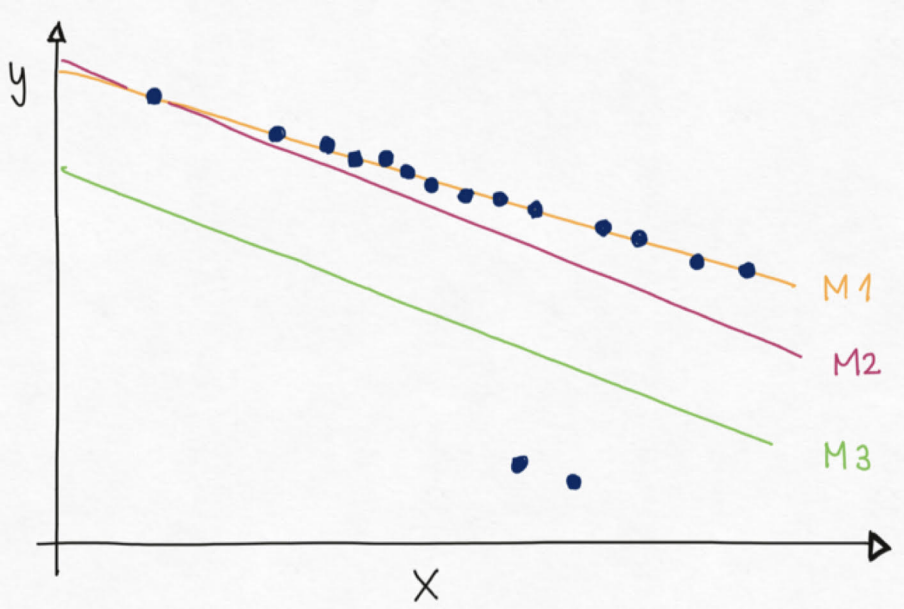


Figure 2.1: Three Different Best-Fit Lines. M1 (MAE), M2 (MSE), & M3 (Max AE).

2.3 Classification Metrics

1. **Accuracy**: The percentage of correct predictions made by the model out of the total number of predictions. In other words, accuracy measures how well the model correctly classified instances into their respective classes.

$$\text{Formula: } \frac{\text{Correct Predictions}}{\text{Total Number of Predictions}}$$

   One advantage of accuracy as a performance metric is its simplicity and ease of interpretation. It provides a straightforward measure of how well the model is performing in terms of correct classification. However, accuracy has certain limitations that should be considered:

   - Class Imbalance: Accuracy can be misleading when dealing with imbalanced datasets, where one class has significantly more instances than the others. In such cases, the model may achieve high accuracy by predicting the majority class correctly while performing poorly on minority classes.
   - Ignores Cost of Missclassification: Accuracy treats all misclassifications equally, regardless of the potential consequences. In many real-world scenarios, misclassifying certain classes may have more severe implications or costs than misclassifying others. For example, in medical diagnosis, misclassifying a serious illness as benign can have more significant consequences than misclassifying a benign condition as serious.

2. **Precision**: The proportion of correctly predicted positive instances out of all instances *predicted* as positive.

$$\text{Formula: } \frac{\text{True Positive}}{\text{True Positive } + \text{False Positive}}$$

   One advantage of precision is its focus on the accuracy of positive predictions. It provides valuable information about the model's ability to correctly identify positive instances and avoid false positives. Precision is particularly useful in scenarios where false positives have significant consequences or costs, such as fraud detection. However, precision has certain limitations that should be considered:

   One disadvantage is the fact that it ignores false negatives: precision solely focuses on the accuracy of positive predictions and does not account for false negatives. False negatives occur when positive instances are incorrectly classified as negative. Precision alone does not provide a comprehensive view of the model's performance regarding the correct identification of all positive instances.

3. **Recall**: The proportion of correctly predicted positive instances out of all actual positive instances.

$$\text{Formula: } \frac{\text{True Positive}}{\text{True Positive } + \text{False Negative}}$$

   One advantage of recall is its focus on capturing all positive instances. It provides valuable information about the model's ability to avoid false negatives and correctly identify all positive instances in the dataset. Recall is particularly useful in scenarios where missing positive instances can have severe consequences, such as in disease diagnosis.

   One disadvantage is the fact that it ignores false positives: recall solely focuses on the ability to identify positive instances and does not consider false positives.

4. **$\mathbf{F}_1$ Score**: Provides a balanced evaluation of the model's performance by taking into account both the ability to correctly identify positive instances (precision) and the ability to capture all positive instances (recall). The F1 score is the harmonic mean of precision and recall:

$$\text{Formula: } \frac{2 \times \text{Precision } \times \text{Recall}}{\text{Precision } + \text{Recall}}$$

   One advantage of the F1 score is that it considers both precision and recall, providing a balanced assessment of the model's performance. It is particularly useful when the trade-off between false positives and false negatives is important, as it takes into account the model's ability to correctly classify both positive and negative instances. This however, also counts as one of its drawbacks, since in some instances precisiona and recall may not be equally important in evaluating a model.

5. **Precision-Recall (PR) Curve**: The PR curve is a plot of precision against recall, with precision on the y-axis and recall on the x-axis. Each point on the curve represents a different classification threshold. Precision measures the accuracy of positive predictions, while recall measures the ability to identify all positive instances.

   The PR curve is particularly useful when dealing with imbalanced datasets or scenarios where the cost of false positives and false negatives differs significantly. It provides a comprehensive view of the model's performance across different levels of recall and can help in selecting an appropriate threshold that balances precision and recall based on the problem's specific requirements.

6. **Receiver-Operating-Characteristics (ROC) Curve**: The ROC curve is a plot of the true positive rate (TPR) or recall on the y-axis against the false positive rate (FPR) on the x-axis. Each point on the curve corresponds to a different threshold setting.

   ROC curve is suitable when the trade-off between true positive rate and false positive rate is important, particularly in scenarios where the cost of false positives and false negatives is roughly similar.

7. **ROC-AUC**: The area under the ROC curve, ranging from 0 to 1. It quantifies the overall performance of the model in distinguishing between positive and negative instances. A higher AUC value indicates better model performance, with 1 representing a perfect classifier and 0.5 indicating a random classifier (no discrimination). Its use comes from the fact that it aggregates performance across all thresholds.

8. **Confusion Matrix**: A confusion matrix, also known as an error matrix, is a table that summarizes the performance of a classification model by displaying the counts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions. It provides a detailed breakdown of the model's predictions and the actual class labels.

9. **Matthews Correlation Coefficient (MCC)**: Takes into account all four elements of the confusion matrix (true positives, true negatives, false positives, and false negatives) to provide a balanced measure of performance.

   The MCC ranges from -1 to 1, where 1 indicates a perfect classifier, 0 indicates a random classifier, and -1 indicates a perfectly inverted classifier. One advantage of the MCC is that it considers the imbalances in the dataset and provides a more balanced evaluation of the model's performance compared to metrics like accuracy, especially when dealing with imbalanced datasets. It takes into account true positives, true negatives, false positives, and false negatives simultaneously, providing a comprehensive measure of performance.

2.4 Regression Losses

1. **Mean Squared Error (MSE)**: MSE takes the mean of the squared residuals. MSE assigns larger penalties to larger errors due to the squaring operation. Its benefits also come from the mathematical properties of MSE (such as when using gradient-based optimization algorithms), since squaring is a differentiable function.

$$\text{Formula: } \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

2. **Mean Absolute Error (MAE)**: MAE measures the average absolute difference between predicted and actual values. It is calculated by taking the mean of the absolute residuals. MAE is useful when you want an error metric that is not heavily influenced by outliers and penalizes errors linearly. If your dataset contains outliers and you don't want them to disproportionately affect your error measurement, MAE is a good choice.

$$\text{Formula: } \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

3. **Huber Loss**: Huber loss is a robust error metric that strikes a balance between Mean Squared Error (MSE) and Mean Absolute Error (MAE). It is particularly useful when dealing with noisy data and outliers.

$$\text{Formula: } L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & \text{if } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

where y is the actual value, $\hat{y}$ is the predicted value, and δ is a threshold parameter. Huber loss provides a more balanced trade-off between smoothness and robustness, making it a reliable choice for regression tasks with noisy or outlier-prone data.

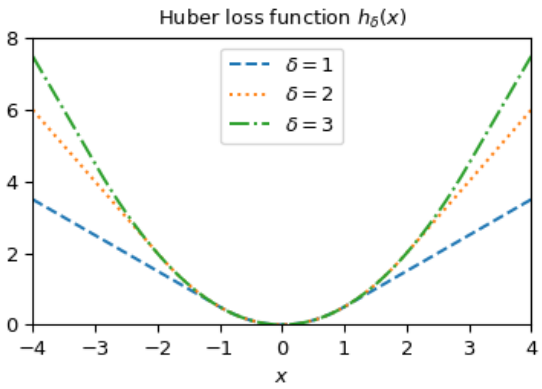


Figure 2.2: Huber Loss for Different Values of δ

2.5 Classification Losses

1. **Binary Cross-Entropy (Binary Entropy or Log Loss)**: Binary Cross-Entropy is a common loss function and performance metric used in binary classification tasks, where there are only two classes (0 and 1). It measures the difference between the predicted probabilities and the true binary labels. Encouraging the model to output high probabilities for the correct class and low probabilities for the incorrect class, Binary Cross-Entropy can be expressed as:

$$\text{Formula: } -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

   where N is the number of samples, y_i is the true binary label (0 or 1), and $\hat{y}_i$ is the predicted probability for the positive class.

2. **Categorical Cross-Entropy (Categorical Entropy or Softmax Cross-Entropy)**: Categorical Cross-Entropy is commonly used in multi-class classification tasks, where there are more than two classes. It measures the difference between the predicted class probabilities and the true one-hot encoded class labels. Encouraging the model to output high probabilities for the correct class and low probabilities for the other classes, Categorical Cross-Entropy can be expressed as:

$$\text{Formula: } -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

   where N is the number of samples, C is the number of classes, y_{ij} is the true one-hot encoded class label (1 if the sample belongs to class j, 0 otherwise), and $\hat{y}_{ij}$ is the predicted probability for class j for the ith sample.

3. **Hinge Loss**: Hinge Loss is often used in Support Vector Machine (SVM) classifiers for binary classification tasks. The loss function aims to maximize the margin between the classes, making it suitable for scenarios with a clear margin between positive and negative instances. For binary classification, Hinge Loss can be expressed as:

$$\text{Formula: } \max(0, 1 - y_i \cdot \hat{y}_i)$$

   where y_i is the true class label (1 or -1) and $\hat{y}_i$ is the predicted score. A penalty is only incurred if there is a difference in the sign of the two values; the true and the predicted.

2.6 Reconstruction Losses

Reconstruction loss in neural networks refers to the measure of dissimilarity between the original input data and the data reconstructed by a neural network. It is commonly used in autoencoders, where the objective is to minimize this loss during training. The lower the reconstruction loss, the better the model can reconstruct the input data, incentivizing it to retain the full depth of the dataset throughout its process. Commonly used loss functions for the reconstruction loss include Mean Squared Error (MSE) and Binary Cross-Entropy (BCE) for different types of data (e.g., continuous or binary).

1. **Mean Squared Error Reconstruction Loss**: Used for continuous data labels.

$$\text{MSE Reconstruction Loss} = \frac{1}{n}\sum_{i=1}^{n} ||x_i - \hat{x}_i||^2$$

   where n is the number of samples, x_i is the original input data, and $\hat{x}_i$ is the reconstructed output.

2. **Binary Cross Entropy Reconstruction Loss**: Used for categorical (0 or 1) data labels.

$$\text{BCE Reconstruction Loss} = -\frac{1}{n}\sum_{i=1}^{n}[x_i \log(\hat{x}_i) + (1 - x_i)\log(1 - \hat{x}_i)]$$

   where x_i and $\hat{x}_i$ represent binary values in the original input and reconstructed output, respectively.

2.7 Distance & Similarity Measures

1. **Euclidean Distance**: The Euclidean distance between two points $P = (x_1, y_1, \ldots, z_1)$ and $Q = (x_2, y_2, \ldots, z_2)$ in an n-dimensional space is given by:

$$\text{Euclidean Distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + \ldots + (z_2 - z_1)^2}$$

   Euclidean distance is best suited for scenarios where you want to measure the straight-line distance between two points in a continuous space.

2. **Manhattan Distance**: The Manhattan distance between two points $P = (x_1, y_1, \ldots, z_1)$ and $Q = (x_2, y_2, \ldots, z_2)$ in an n-dimensional space is given by:

$$\text{Manhattan Distance} = |x_2 - x_1| + |y_2 - y_1| + \ldots + |z_2 - z_1|$$

   Manhattan distance is more suitable for grid-like spaces where movement is restricted to horizontal and vertical directions. It is often used in path-finding algorithms and scenarios where only certain directions are allowed.

3. **Pearson Correlation Distance**: The Pearson correlation distance between two points $P_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ and $P_k = (x_{k1}, x_{k2}, \ldots, x_{kn})$ in an n-dimensional space is given by:

$$\text{Pearson Correlation Distance} = 1 - \frac{\sum_{j=1}^{n}(X_{ij} - \bar{x}_j)(X_{kj} - \bar{x}_j)}{\sqrt{\sum_{j=1}^{n}(X_{ij} - \bar{x}_j)^2}\sqrt{\sum_{j=1}^{n}(X_{kj} - \bar{x}_j)^2}}$$

   Using correlation distances is useful in micro-array analyses, where researchers can gain insights into the underlying patterns and relationships between genes and samples by working with similar groupings of data points.

4. **Spearman Rank Correlation Distance**: The Spearman rank correlation distance is a measure of the dissimilarity between two points X_i and X_k of in an n-dimensional space based on the Spearman rank correlation coefficient.

$$\text{Spearman Rank Correlation Distance} = \frac{\sum_{j=1}^{n}(R_{ij} - \bar{r}_j)(R_{kj} - \bar{r}_j)}{\sqrt{\sum_{j=1}^{n}(R_{ij} - \bar{r}_j)^2}\sqrt{\sum_{j=1}^{n}(R_{kj} - \bar{r}_j)^2}}$$

   Where R_{ij} is the rank for point R_i for reature j, and $\bar{r}_j$ is the average of the ranks for feature j. This type of distance measure is particularly useful concerning outliers in a dataset.

5. **Cosine Similarity**: Cosine similarity is a measure of similarity between two points, and can be measured by:

$$\text{Cosine Similarity} = \frac{\sum_{j=1}^{n}(X_{ij} \cdot X_{kj})}{\sqrt{\sum_{j=1}^{n}(X_{ij})^2} \cdot \sqrt{\sum_{j=1}^{n}(X_{kj})^2}}$$

Chapter 3

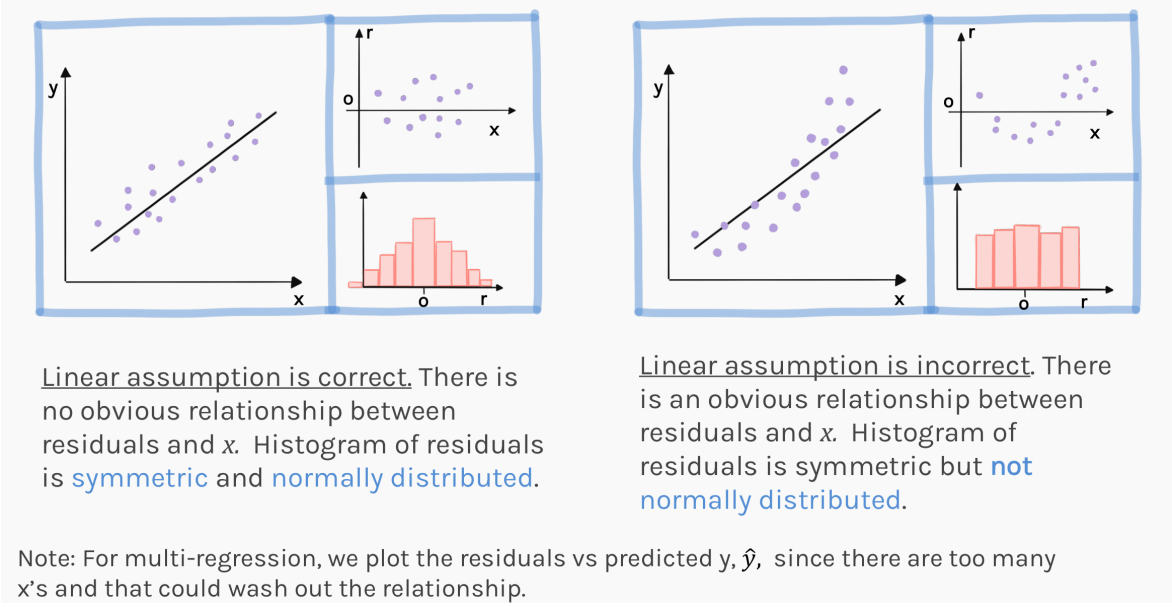
Linear Regression

Linear regression is at the heart of explaining all regression problems. Regression algorithms have improved over the past few decades, but as they did they went through a lot of transformations and increasing levels of complexity. The following outlines the main linear regression coefficient-finding (also known as model training approaches):

3.1 Simple Linear Regression Assumptions

For linear regression, we make four important assumptions:

- Linearity: Linear relationship between expected value and predictors.
- Normality: Residuals are normally distributed about expected value.
- Homoskedasticity: Residuals have constant variance σ^2.
- Independence: Observations are independent of one another.



Linear assumption is correct. There is no obvious relationship between residuals and *x*. Histogram of residuals is symmetric and normally distributed.

Linear assumption is incorrect. There is an obvious relationship between residuals and *x*. Histogram of residuals is symmetric but not normally distributed.

Note: For multi-regression, we plot the residuals vs predicted y, $\hat{y}$, since there are too many x's and that could wash out the relationship.

Figure 3.1: Linearity Assumption Validity

3.2 Closed Form Solution to Simple Linear Regression

Consider a simple regression problem with two features, one predictor and the other response, x and y. The goal is to find the coefficients $y = \beta_1 x + \beta_0$ that best fits the relationship between the two variables. The way go about finding this line of best fit goes as follows:

- Choose Measure of Overall Error: In this example, this will be Mean Squared Error (MSE).
- Define Loss Function: $L(\beta_1, \beta_0) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_1 x_i + \beta_0))^2$.
- Find Parameters that Minimize Loss: This involves trying to find the global minimum, though we usually try to do so by finding the stationary points. Stationary points are those where the gradient is zero, meaning:

  $$\nabla L = \left[\frac{\partial L}{\partial \beta_0}, \frac{\partial L}{\partial \beta_1}\right] = 0$$

  This point isn't guaranteed to be a minimum (nor be global) and as such we put extra checks in place to discover which type of stationary point we are dealing with.
- To check that a function L is convex (meaning the global minimum happens at the stationary point), we check whether its Hessian (a matrix of second partial derivatives) satisfies a property called positive semi-definiteness.
- Compute the gradient of the loss (can be done in closed form for a simple regression example), then solve for the stationary points such that $\nabla L = 0$. Doing so for this linear regression task results in the following estimates for the coefficients:

  $$\hat{\beta}_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

  ‘

  $$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

It was also possible to use a different loss function to optimize, such as mean absolute error, or maximum absolute error (explained in the Project Framework section).

3.3 Closed Form Solution to Multi-Linear Regression

Now we can expand the problem to one where we have multiple predictors for one response variable. Instead of a simple vector for the predictor information, $\mathbf{X}$ is a matrix where each row represents a set of input variables, and $\mathbf{Y}$ is a column vector representing the corresponding output values.

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

To get an intercept value during fitting, we first add a column of ones to $\mathbf{X}$:

$$\tilde{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1m} \\ 1 & x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}$$

To calculate the parameter vector (the equivalent of fitting the model), we pick the MSE loss function:

$$\text{MSE}(\beta) = \frac{1}{n}||Y - X\beta||^2$$

As in the simple, one predictor one response example, we need to find the gradient of the loss function with respect to each parameter, which, in matrix notation can be expressed as:

$$\frac{\partial L}{\partial \beta} = -2\mathbf{X}^T(Y - \mathbf{X}\beta)$$

Just as before, we set this equal to zero (drop the coefficient), and solve for the parameter vector β:

$$\mathbf{X}^T(Y - \mathbf{X}\beta) = 0$$

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

This exact solution is known as the *Normal Equation*.

3.4 Polynomial Regression

The strength of polynomial regression comes from the fact that it incorporates the synergy effect. The synergy or interaction effect is the case where the combined effect of two or more variables is greater or different from the sum of their individual effects. In other words, when the interaction between variables produces an effect that is not simply additive.

This difference between multi-linear and polynomial regression doesn't mean modeling should change. We can use the multi-linear regression approach for polynomial regression, where now, we consider the interaction terms of predictors, and the predictor values raised to different degrees between 1 and d, as new features for our model.
Polynomial regression also requires another important pre-processing step: scaling. Since we are dealing with predictor values raised to arbitrary exponents, extremely small or extremely large predictor values will be problematic when raised to these exponents. This results in a need to scale our data, which we can do by standardizing all of the features.

3.5 Singular Value Decomposition Approach

Using the Normal Equation isn't always the best approach to solving a regression problem. An issue with the Normal equation is that using it may not work if the matrix $\mathbf{X}$ is not invertible, such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined. This is where the SVD approach comes in, as it is more computationally efficient and handles edge cases nicely. The Singular Value Decomposition (SVD) is a powerful matrix factorization technique that can be used to solve linear regression problems. The SVD of a matrix X is defined as:

$$X = U\Sigma V^T$$

where X is the input matrix, U and V are orthogonal matrices, and Σ is a diagonal matrix containing the singular values of X.

To apply SVD to linear regression, we follow these steps:

1. Calculate the SVD of the input matrix X:

$$X = U\Sigma V^T$$

2. Compute the pseudoinverse of Σ, denoted as Σ^+, by taking the reciprocal of nonzero singular values and transposing the result. The nonzero singular values in Σ correspond to the diagonal elements of Σ^+.

3. Compute the pseudoinverse of X, denoted as X^+, using the formula:

$$X^+ = V\Sigma^+ U^T$$

4. Calculate the slopes β using the pseudoinverse X^+ and the output vector Y as:

$$\beta = X^+ Y$$

The pseudoinverse X^+ obtained through SVD provides a solution to the linear regression problem even when the matrix X is not invertible or when there are collinearities among the input variables.

3.6 Model Selection

In the realm of model selection, we usually try to find the model with the least error. However, not all error is created equally, and some we shouldn't try to reduce. There are two types of error:

- Irreducible (Aleatoric) Error: We can't do anything to decrease the error due to noise.
- Reducible (Epistemic) Error: We can decrease the error due to overfitting and underfitting by improving the model.

Model selection is centered on one principle: finding a model that hits a sweet spot in the Bias-Variance trade-off continuum. On one end, we overfit to the training data: the model is tuned too closely to the train data, capturing meaningless noise. In this case we have a model with low bias but high variance. On the other end, we underfit to the training data: the model is too simple to accurately capture the nuance within the data. This results in a model with high bias but low variance.

There are a multitude of different ways to find the best linear regression model, but not all are created equal, outlined below are some of the most common methods, their advantages and their limitations.

3.6.1 Exhaustive Search

We can create a separate model for each different combination of predictors, starting at 1 predictor and ending with all of them. This means that we will need to train, for p predictors, a time complexity of $O(2^p)$, which is really computationally expensive.

3.6.2 Stepwise Forward Variable Selection

The stepwise forward approach goes as follows:

1. Calculate a baseline model with 0 predictors (just the intercept).
2. Train p models, each with 1 different predictor each, and evaluate each using some error metric.
3. Select the best model from the previous step, and then create $p-1$ model, with the best predictor from the previous step alongside a new predictor. Then, evaluate each new model.
4. Repeat this step, now with 3 predictor, and then 4, all the way until we create a model with all predictors.
5. Finally, select the best model by comparing the best model of each generation of models.

This approach has time complexity $O(p^2)$, much less than the exhaustive approach.

3.6.3 Stepwise Backward Variable Selection

In the backward approach, we start with a model using all predictors, and then scale down one predictor at a time until we reach our baseline (intercept) model. The time complexity of this approach is also $O(p^2)$.

3.6.4 Cross-Validation

Cross-validation is a method used to assess how well a model generalizes to new, unseen data wuthout the need of looking at the test set. It involves partitioning the dataset into subsets and iteratively training and evaluating the model on different combinations of these subsets.

One common approach to cross-validation is k-fold cross-validation. Here, the dataset is divided into k equally sized folds. During each iteration, one fold is designated as the validation set, while the remaining $k-1$ folds are used as the training set. This process is repeated k times, allowing every fold to serve as the testing set once. For each iteration, the model is trained on the training set and then evaluated on the testing set. Performance metrics, such as MSE, are calculated based on the model's predictions. After all iterations are completed, the performance metrics are aggregated or averaged to provide an overall assessment of the model's performance; this gives us a way to compare different models without the need to access our test set.

3.6.5 Regularization

One way to deal with model overfitting issues is regularization. Regularization is based off the idea of adding a second objective during the training process: limiting the coefficient complexity that our best model finds. We do this using a complexity term, which is different based on the type of regularization used.

Once we have selected the type of regularization used, we scale it by a regularization parameter λ to control how much we want our training process should focus on the error-reduction objective and the complexity reduction objective. A λ value of 0 means the model we train is equivalent to normal regression, while a λ value that tends to ∞ results in a model with coefficients equal to 0. Our choice of λ is usually selected by cross-validation.

Ridge

Ridge regression employs "L2 regularization," which supplements the linear regression cost function with a complexity measure directly proportional to the squared magnitude of coefficients. The formula for the ridge regression cost function can be expressed as follows:

$$\text{Cost} = \text{Least Squares Cost} + \lambda \sum_i \beta_i^2$$

Due to the nature of the complexity measure, ridge regression tends to decrease coefficients in an exponential fashion. Since the complexity measure is squared, we can solve for an anlytical solution (since we can take the derivative of this combined cost function). This makes ridge regression less costly than its counterpart, LASSO regression.

LASSO

LASSO, or Least Absolute Shrinkage and Selection Operator, encompasses a form of regularization called "L1 regularization." This form augments the linear regression cost function with a complexity measure equal to the absolute values of coefficients. The formula for the LASSO cost function is as follows:

$$\text{Cost} = \text{Least Squares Cost} + \lambda \sum_i |\beta_i|$$

Due to the nature of the complexity measure, LASSO regression tends to decrease coefficients a lot quicker; zeroing out unimportant ones. Since the complexity measure uses the absolute value function, we can't take its derivative and so need to use a solver (using sub-gradients).

Elastic Net

Elastic Net regularization combines elements of both L1 and L2 regularization techniques. It introduces a complexity measure that blends the effects of L1 and L2 regularization. This measure aims to mitigate overfitting by promoting sparse coefficients while also controlling their magnitudes. The formula for the Elastic Net cost function can be expressed as:

$$\text{Cost} = \text{Least Squares Cost} + \lambda_1 \sum_i |\beta_i| + \lambda_2 \sum_i \beta_i^2$$

Here, λ_1 and λ_2 are regularization parameters that govern the strengths of the L1 and L2 regularization components, respectively. Elastic Net combines the benefits of both L1 and L2 regularization: L1 regularization (LASSO) encourages sparsity in feature selection by pushing some coefficients to exactly zero (feature selection), while L2 regularization (Ridge) finds coefficient values in a more stable fashion.

3.7 Model Significance

We need to assess whether a certain model we trained is statistically relevant or significant. This means that if a model predicts that there is a strong positive relationship between predictor and response, we need to be sure that this result is not by chance, but is actually a valid trend. We can do this using a number of different strategies, outlined below.

3.7.1 Coefficient Accuracy

We can create confidence intervals for our coefficients. We do this through bootstrapping: sampling data out of our dataset with replacement until we get a number of bootstrap samples. Then, to each bootstrap sample, we fit our model and find the associated predictor coefficients. We can then find the mean, and standard error of our coefficient estimates. If we add/subtract 2 standard errors from this mean we get a 95% confidence interval: the range of values such that the true value of our coefficient is contained in this range with 95% probability.

3.7.2 Coefficient Significance

We can also use these bootstrap estimates to find whether we have significant information to definitevely say the relationship between predictor and response is positive, for example. We do this by using the $\hat{t}$-test: we calculate how far away the mean estimate is from zero in units of standard error (by simply dividing the bootstrap mean by the standard error). We then use hypothesis testing (and find an associated p-value) by comparing the $\hat{t}$-test value to how likely it is to observe such a value purely due to chance (which we can do using the student-t distribution).

3.7.3 Model Confidence & Prediction Intervals

95% Confidence Interval using Bootstrapping

1. **Data Resampling:** Randomly select samples (with replacement) from the original dataset. The number of samples should match the size of the original dataset.
2. **Model Fitting:** Fit a linear regression model to each bootstrap sample. Calculate the regression coefficients for each sample.
3. **Coefficient Distribution:** Create a distribution of the regression coefficients obtained from the different bootstrap samples. This distribution represents the variability in the coefficient estimates.
4. **Percentile Method:** Calculate the 2.5th and 97.5th percentiles of the coefficient distribution. These percentiles define the boundaries of the 95% confidence interval.

95% Prediction Interval using Bootstrapping

1. **Data Resampling:** Similarly, randomly select samples (with replacement) from the original dataset to create bootstrap samples.
2. **Model Fitting:** Fit a linear regression model to each bootstrap sample.
3. **Prediction Distribution:** For a specific new observation (predictor values), make predictions using the fitted models for each bootstrap sample. Create a distribution of these predicted values.
4. **Percentile Method:** Calculate the 2.5th and 97.5th percentiles of the predicted value distribution. These percentiles define the boundaries of the 95% prediction interval for the new observation.

A confidence interval is a range within which we are confident that the true parameter (such as a coefficient) lies. It quantifies the uncertainty associated with estimating the population parameter based on a sample. A prediction interval, on the other hand, is a range within which we expect a future observation to fall with a certain level of confidence. It takes into account the variability of the data points around the regression line and the uncertainty in predicting a new observation.

Prediction intervals are usually wider than confidence intervals. In addition to the uncertainty in estimating the regression coefficients (captured by the confidence interval), prediction intervals also account for the uncertainty in predicting individual observations – this added uncertainty contributes to its wider spread.

Chapter 4

Generalized Linear Models

4.1 Model Flexibility

Generalized Linear Models (GLMs) are a class of statistical models that extend the concept of linear regression to handle a wider range of response variables and error distributions. While traditional linear regression assumes that the response variable follows a normal distribution and has constant variance, GLMs relax these assumptions and allow for different types of distributions and relationships. GLMs relax two of the assumptions of linear regression: the two assumptions are:

- Constant Variance (Homoskedasticity): GLMs allow the variance of the response variable to vary with the mean of the response variable. This means that the assumption of constant variance across all levels of the predictor variables is relaxed.
- Normality of Errors: GLMs do not require the errors to be normally distributed. They can handle a variety of response variable distributions beyond the normal distribution, which is a departure from the assumption of normality in traditional linear regression.

This flexibility comes as a result of two amendments to our modeling choices: we assume response variable comes from a family of distributions called the exponential dispersion family (EDF), and that the relationship between expected value and our predictors is expressed through a link function.

4.1.1 Exponential Dispersion Family

In the context of GLMs, the Exponential Dispersion Family (EDF) encompasses a family of probability distributions suitable for modeling diverse data types and relationships. The density function for a distribution within the EDF framework has the following general form:

$$f(y_i \mid \theta_i, \phi_i) = \exp\left(\frac{y_i\theta_i - b(\theta_i)}{\phi_i} + c(y_i, \phi_i)\right)$$

Where:

- $f(y_i \mid \theta_i, \phi)$ is the density function of the distribution for observation y_i, with parameters θ_i and ϕ_i.
- y_i is the observed response value for the ith observation.
- θ_i is the canonical parameter.
- ϕ_i is the dispersion parameter.
- $b(\theta_i)$ is the cumulant function.
- $c(y_i, \phi_i)$ is the normalization factor.

Rewriting distributions in this useful format gives rise to two important properties:

$$\mathrm{E}(y_i) = b'(\theta_i)$$
$$\mathrm{Var}(y_i) = \phi_i \cdot b''(\theta_i)$$

The Normal, Bernoulli/Binomial, Poisson, Negative Binomial, and Exponential/Gamma are all examples of distributions which belong to the EDF.

4.1.2 Link Function

A link function is a crucial component in Generalized Linear Models (GLMs) that relates the linear predictor to the expected value of the response variable. It serves as the bridge between the linear predictor $x_i\beta_i$ (referred to as η, and the expected value of the response variable within the chosen distribution family. It transforms the linear predictor into the appropriate scale for the response variable's distribution.
For example, for the binomial distribution, the link function is called the logit function:

$$\log\left(\frac{p_i}{1-p_i}\right) = x_i\beta_i$$

This gives rise to (and explains) the methodology of logistic regression.

4.2 Logistic Regression

Unlike linear regression, which is used for continuous dependent variables, logistic regression is specifically designed for predicting probabilities and outcomes in cases where the dependent variable is categorical and binary. This allows us to extend modeling to scenarios where the response isn't a continuous variable, but is instead discrete with a finite number of classes or labels; it also allows us to predict a categorical variable without the need to order the classes labels in some fashion (which is done by default under the assumptions of linear regression).

The logistic regression model aims to model the probability $P(y = 1 \mid x_1, x_2, \ldots, x_p)$ of the outcome being 1 as follows:

$$P(y = 1 \mid x_1, x_2, \ldots, x_p) = \frac{1}{1 + e^{-\beta_0 - \beta_1 x_1 - \beta_2 x_2 - \ldots - \beta_p x_p}}$$

4.2.1 Cross Entropy Loss

Binary Cross Entropy is the loss of choice for logistic regression. We can derive the cross-entropy loss from the perspective of maximum likelihood estimation (MLE). The likelihood function represents the probability of observing the given labels (y) given the features and the model parameters. Assuming independent and identically distributed samples, the likelihood for a single instance is given by:

$$L(y|x;\beta) = \hat{y}^y \cdot (1-\hat{y})^{1-y}$$

Where:

- $L(y|x;\theta)$ is the likelihood of observing label y given features x and parameters β.
- $\hat{y}$ is the predicted probability (output of the logistic function) that the instance belongs to the positive class.

We can then generalize to the join probability of all datapoints as our likelihood:

$$L(\mathbf{y}|\mathbf{X};\beta) = \prod_{i=1}^{N} \left(\hat{y}_i^{y_i} \cdot (1-\hat{y}_i)^{1-y_i}\right)$$

It's often more convenient to work with the log-likelihood to simplify calculations. Taking the logarithm of this likelihood, we get:

$$\log L(\mathbf{y}|\mathbf{X};\beta) = \sum_{i=1}^{N} (y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

In MLE, we seek the values of the parameters β that maximize the log-likelihood. Since maximizing the likelihood is equivalent to minimizing the negative log-likelihood, up to a constant value, we arrive it binary cross-entropy loss formula:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$$

Finding the analytical solution to this formula is intractable (unlike linear regression) so we utilize gradient descent to find the optimal parameters β.

4.2.2 OVR Multiclass Logistic Regression

OVR (One-vs-Rest) Multiclass Logistic Regression is a technique used for multiclass classification tasks. In multiclass classification, the goal is to assign a single input instance to one of several possible classes. OVR logistic regression is a simple way to extend binary logistic regression to handle multiple classes.

For a multiclass problem with k classes, OVR trains k separate logistic regression models. Model i is trained on instances of class i as positives and instances of other classes as negatives. To predict the class label for a new instance, each of the k models produces a probability score. The class with the highest probability is predicted as the output.

4.2.3 Multinomial Logistic Regression

Multiclass logistic regression, also referred to as softmax regression or multinomial logistic regression, is a different method used to predict labels for multiple categories, which, instead of using binary logistic regression, generalizes the approach.

To implement multiclass logistic regression, a linear model computes a "score" for each class k, related to the input features. This score is calculated as $z_k = \mathbf{w}_k \cdot \mathbf{x} + b_k$, where $\mathbf{w}_k$ represents the weight vector for class k, $\mathbf{x}$ is the input feature vector, and b_k is the bias term for class k. The softmax function is then employed to transform these scores into class probabilities. By exponentiating the scores and normalizing them, the probabilities for each class k are obtained. Specifically, the probability of class k is:

$$P(y = k|\mathbf{x}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

where e^{z_k} is the exponential of the score for class k, and the denominator computes the sum of exponentials across all classes.

For this approach, we need to generalize our loss too. Multiclass logistic regression utilizes the cross entropy loss. For a single instance, the cross entropy loss is expressed as:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \cdot \log(\hat{y}_k)$$

where y_k denotes the true binary label for class k (1 if the sample belongs to class k, 0 otherwise), and $\hat{y}_k$ signifies the predicted probability for class k after applying the softmax function. Similarly to logistic regression, gradient descent is used to find the weight vectors $\mathbf{w}$ and biases $\mathbf{b}$.

4.3 Poisson Regression

4.3.1 Assumptions

Poisson regression is a statistical technique used for modeling count data, where the response variable represents the number of occurrences or events within a fixed unit of time, space, or volume. It is suitable when you are interested in modeling the relationships between predictor variables and the rate of occurrence of events. It's an extension of the linear regression model, specifically designed for situations where the assumptions of the normal distribution and constant variance don't hold.
Poisson regression, a type of generalized linear model, upholds the following linear regression assumptions:

- **Independence:** Observations are assumed to be independent of each other.
- **Linearity:** The relationship between the predictor variables and the log of the expected counts is linear.

However, it also introduces its own set of assumptions to the mix:

- **Count Data:** The response variable represents counts of events in a fixed interval or area.
- **Non-Negativity:** The response variable must be non-negative (counts cannot be negative).
- **Equal Variance:** The variance of the response variable is equal to its mean: the variability in the count data is entirely captured by the mean count. If overdispersion is present (variance > mean), alternatives like negative binomial regression might be more appropriate.

4.3.2 Modeling

In Poisson regression, the response variable Y is assumed to follow a Poisson distribution with mean λ (lambda). The model can be represented as:

$$\log(\lambda) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p$$

Where:

- $\log(\lambda)$ is the natural logarithm of the expected count.
- $\beta_0, \beta_1, \ldots, \beta_p$ are the coefficients for the predictor variables $x_1, x_2, \ldots, x_p$.
- p is the number of predictor variables.

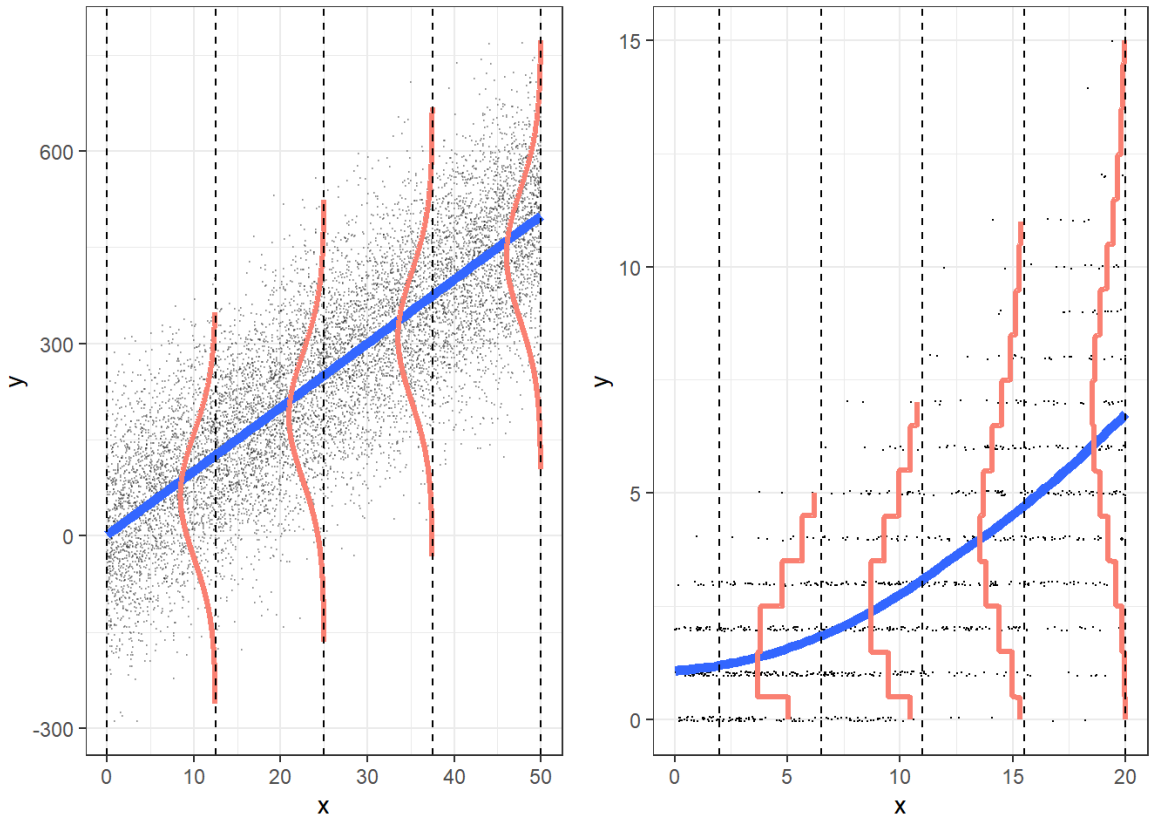


Figure 4.1: Linear vs. Poisson Regression

The link function used to relate the linear combination of predictors to the mean of the Poisson distribution is the natural logarithm. This is because the logarithm helps ensure that the predicted mean count remains positive.

For training purposes, we use the Poisson PDF to construct a negative log-likelihood for our model, which we then minimize using gradient descent to find the optimal parameters β.

4.3.3 Limitations

Despite having a large number of use cases, Poisson regression exhibits some limitations:

- **Overdispersion**: If the variance of the counts is greater than the mean, a better alternative to the Poisson might be Negative Binomial Regression, which is an extension of Poisson regression that allows for greater flexibility in modeling variance.
- **Zero-Inflation**: Zero-inflated models exhibit two separate data generation processes for observed counts: one for the zeros and one for positive counts. When there's an excessive number of zeros in the count data, zero-inflated Poisson or zero-inflated negative binomial models can be used to handle excess zeros in the data.

Chapter 5

Support Vector Machines

Support Vector Machines (SVMs) are powerful supervised machine learning models used for classification and regression tasks. SVMs are particularly useful when working with complex, high-dimensional datasets, though are very difficult to train in cases where the training set is in the tens of thousands of observations or more (the model scales poorly). SVMs aim to find the optimal hyperplane that separates data points of different classes with the maximum margin. The points closest to the hyperplane are called support vectors and play a crucial role in defining the decision boundary.

5.1 Mathematical Explanation

Support Vector Machines (SVMs) aim to find the optimal hyperplane that separates data points of different classes with the maximum margin. This optimization problem can be formulated as follows:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right)$$

Here, let's explain the meaning of each variable and term in the optimization problem:

- $\mathbf{w}$: The weight vector perpendicular to the decision boundary (hyperplane). It defines the orientation of the hyperplane.
- b: The bias term, also known as the intercept. It determines the position of the hyperplane along the y-axis.
- C: The regularization parameter, which controls the trade-off between maximizing the margin and minimizing the training errors. A higher value of C allows for fewer margin violations but may lead to overfitting, while a lower value of C emphasizes a wider margin with more margin violations.
- n: The number of training samples.
- y_i: The binary class label of the ith training sample. $y_i = 1$ for positive class and $y_i = -1$ for negative class.
- $\mathbf{x}_i$: The ith training sample, represented as a feature vector.

The term $\frac{1}{2}\|\mathbf{w}\|^2$ in the objective function represents the regularization term that encourages a smaller weight vector and a larger margin. The second term, $C\sum_{i=1}^{n} \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right)$, is the hinge loss, which penalizes the misclassification of training samples. It measures how far each sample is from being correctly classified by the decision boundary.

5.2 The Dual Problem

The optimization problem of SVMs is typically transformed into its dual form for computational efficiency and to take advantage of the kernel trick. The dual problem allows us to compute the solution using only the inner products between the training samples, avoiding the explicit calculation of the weight vector $\mathbf{w}$. The reason we need to rethink in terms of a dual problem is that we need a certain level of soft-margin classification in our models. To do so, we introduce a slack variable ζ, which indicates how much an instance is allowed to violate the margin. This results in two conflicting objectives:

- Make the slack variables as small as possible to reduce margin violations.
- Make the $\frac{1}{2}\|\mathbf{w}\|^2$ term as small as possible to increase the margin. Intuitively, a smaller weight vector implies that the decision boundary is less sensitive to individual data points. This allows for a wider margin between the decision boundary and the support vectors, which are the data points closest to the decision boundary.

The dual problem is obtained by introducing Lagrangian multipliers (α_i) to create a Lagrangian function and then maximizing it with respect to the Lagrange multipliers, subject to certain constraints. Solving the dual problem leads to the computation of the support vectors and their corresponding Lagrange multipliers. The dual problem reduces the dimensionality of the optimization problem and allows for efficient kernel-based computations. By solving the dual problem, we can obtain the support vectors and construct the decision boundary without explicitly calculating the weight vector $\mathbf{w}$.

5.3 Kernel Trick

The kernel trick is a technique used in SVMs to handle non-linear data. It involves mapping the original data into a higher-dimensional feature space using a kernel function. By applying the kernel function, SVMs can implicitly compute the dot products between feature vectors in the high-dimensional space without explicitly transforming the data. The kernel trick is based on the fact that many machine learning algorithms only depend on the dot products between data points. Common kernel functions include the linear kernel, polynomial kernel, Gaussian (RBF) kernel, and sigmoid kernel. The kernel trick allows SVMs to efficiently and effectively handle non-linear data, making them a versatile choice for various classification tasks.

5.4 Advantages & Limitations

SVMs offer several advantages:

- Effective in high-dimensional spaces: SVMs perform well in scenarios where the number of features is large compared to the number of samples.
- Robust against overfitting: SVMs are less prone to overfitting due to the margin maximization objective.
- Versatile with kernel functions: SVMs can handle non-linear data by using the kernel trick.

However, they may also not be the best choice in the following situations:

- Large datasets: SVMs can be computationally expensive and memory-intensive for large datasets.
- Interpretability: SVMs tend to be less interpretable compared to other models like decision trees or logistic regression.

Chapter 6

Decision Trees

The motivation behind using decision trees is creating models that are easy to interpret yet allow for complex decision boundaries. In a decision tree, every decision corresponds to a partition along one of the axis of the given predictors. To make decisions, we simply traverse the tree to arrive at a leaf node.

6.1 Splitting Criteria

1. The regions in the feature space should grow progressively purer with the number of splits. That is, we should see that each region 'specializes' towards a single class.
2. The fitness metric of a split should take a differentiable form (making optimization possible).
3. We shouldn't end up with empty regions – regions containing no training points.

As to how we quantify the purity of each created region, we use one of the following three metrics. In each, k is the number of child nodes, c is the number of classes, p_{ij} represents the proportion of instances belonging to class i in child node j, N_j is the number of instances in child node j, and N is the total number of instances in the node:

6.1.1 Weighted Average Classification Error

Used to measure the misclassification rate within a node. It is calculated by dividing the number of misclassified instances by the total number of instances in the node. The formula for classification error is as follows:

$$\text{Classification Error} = \sum_{j=1}^{k} (1 - \max(p_{ij})) \times \frac{N_j}{N}$$

The value of classification error ranges between 0 and 1, where 0 indicates a perfectly pure node with all instances belonging to the same class; the reason why the max function is used is that we classify a group with the label of the most represented class within it.

6.1.2 Weighted Average Gini Index

This measure calculates the probability of a randomly selected element being misclassified when it is randomly labeled according to the distribution of labels in the node. The weighted average Gini index takes into account the impurity of child nodes weighted by the number of instances in each child node. The formula for the weighted average Gini index is as follows:

$$\text{Weighted Average Gini Index} = \sum_{j=1}^{k} \left(1 - \sum_{i=1}^{c} p_{ij}^2\right) \times \frac{N_j}{N}$$

The Gini index ranges between 0 and 1, where 0 indicates a pure node.

6.1.3 Weighted Average Entropy

Entropy calculates the amount of information required to describe the class distribution in the node. The weighted average entropy considers the entropy of child nodes weighted by the number of instances in each child node. The formula for the weighted average entropy is as follows:

$$\text{Weighted Average Entropy} = -\sum_{j=1}^{k}\sum_{i=1}^{c} p_{ij}\log_2(p_{ij}) \times \frac{N_j}{N}$$

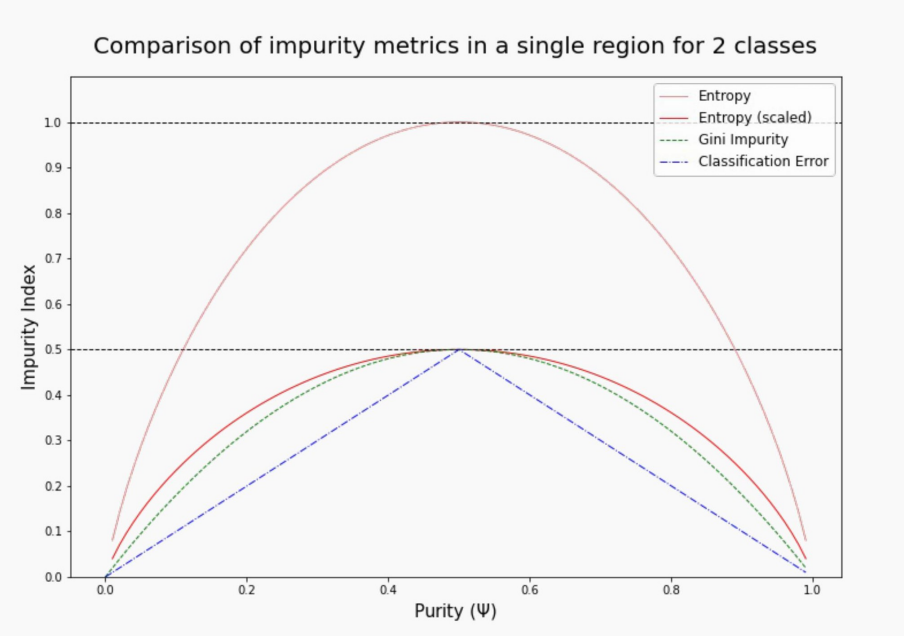


Figure 6.1: Compairson of Impurtiy Metrics

Of these, the entropy measure penalizes impurity the most, and so we see that it is held as the impurity metric of choice.

6.2 Stopping Conditions

If we let trees grow freely, we will end up with leaf nodes that have exactly one point each, and the leaf label being the class of that singular training point. To prevent that, we use numerous different stopping conditions, which, as always, we can tune the hyperparameters for using cross-validation:

1. Maximum Depth (*default strategy*)
2. Minimum Samples Per Leaf
3. Maximum Leaf Nodes
4. Minimum Impurity Decrease

There are two main approaches to growing a tree:

- *Best-First*: The algorithm evaluates and selects the best attribute to split the current node based on a predefined evaluation measure. The best-first strategy considers all possible attributes and evaluates their potential splits before selecting the one with the highest evaluation measure. It prioritizes exploring the most promising splits at each step, leading to potentially faster convergence to an optimal tree structure. Best-first strategies tend to prioritize exploration of the most promising splits, potentially leading to faster convergence to an optimal tree structure. Generally used in tandem with *maximum leaf nodes* as the stopping criterion. However, they can be computationally more expensive, especially for large datasets with a large number of attributes.
- *Depth-First*: The algorithm starts from the root node and chooses the best attribute to split the current node based on the evaluation measure. Once the split is made, the algorithm proceeds to expand the child nodes and repeats the process until a stopping criterion is met, which is usually *maximum depth*. Depth-first strategies are computationally efficient as they explore fewer splits at each node. They are suitable for large datasets or cases where memory or computational resources are limited. However, they may not guarantee finding the globally optimal tree structure and can be prone to overfitting if the tree depth is not properly controlled.

6.3 The CART Algorithm

The CART (Classification and Regression Trees) algorithm is the main method for building decision trees. It can be applied to both classification and regression tasks, making it versatile in handling different types of data. Here is a step-by-step explanation of how the CART algorithm works:

1. The algorithm starts by considering the entire dataset as the root node of the decision tree. It evaluates different combinations of features and thresholds and selects the best to split the data (based on a given impurity metric).

2. Notably, the algorithm is not exhaustive in evaluating all possible combinations of features and thresholds. Instead, it follows a heuristic approach to find a locally optimal split at each node of the decision tree.

3. The tree is grown out until a stopping condition is met.

4. The algorithm generates leaf nodes when a stopping criterion is met. Leaf nodes represent the final predictions or decisions made by the decision tree. In a classification problem, each leaf node is associated with a specific class label, while in regression, the leaf nodes contain numerical values representing the predicted outcome.

6.4 Pruning

Rather than preventing a complex tree from growing, we can obtain a simpler tree by 'pruning' a complex one. Pruning relies in cost complexity scoring: a metric that incorporates classification error as well as a complexity measure for the number of leaves in the tree – usually balanced by a hyperparameter α. The pruning algorithm completes the following steps:

- Start with a full tree, where each leaf node is pure, and with an arbitrarily fixed value α.
- Start by pruning one part of tree, such that the cut used is the one that maximizes the difference of the cost complexity score of the trees.
- Continue the pruning step iteratively until you get trees $T^0, T^1...$, with the last tree just a root node of T^0.
- Select the optimal one of these trees by cross validation.
- Again, using cross validation, select the best hyperparameter value for α.

6.5 Axis Orientation

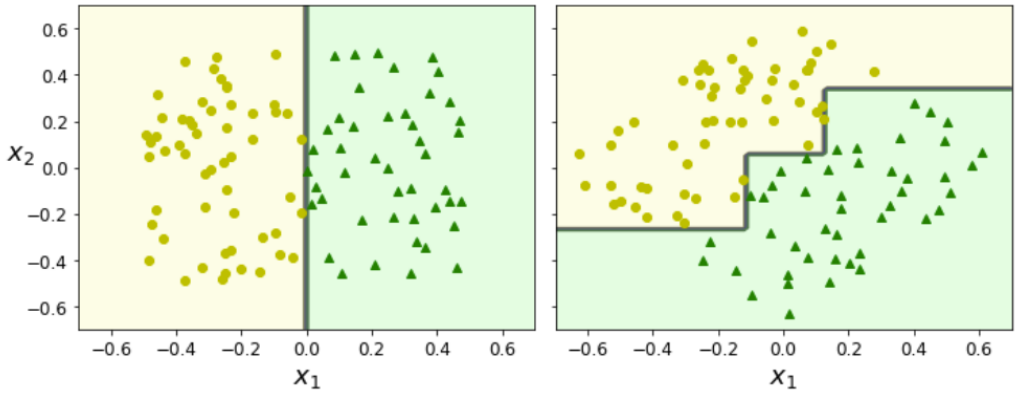


Figure 6.2: Data Axis Rotation

The figure above showcases a dataset that is linearly separable. On the left side, a decision tree can effortlessly split the data, while on the right side, where the dataset is rotated by 45 deg, the decision boundary appears convoluted unnecessarily. Despite both decision trees perfectly fitting the training set, it is highly probable that the model on the right will struggle to generalize well. To mitigate this issue, one approach is to scale the data and then apply a principal component analysis (PCA) transformation. PCA rotates the data to reduce the correlation between the features, which often simplifies matters for decision trees, though not always.

Chapter 7

Ensemble Learning

7.1 Reasoning

Provided we have a sufficient number of weak learners that are sufficiently different in their decision making methodologies, their aggregate, called an *ensemble*, will likely perform better than each of these learners on their own. These ensemble methods work best when the individual models used are as independent from each other as possible. An ensemble ends up having comparable bias to individual member models but less variance.

7.2 Soft vs. Hard Voting

Hard voting is a simple approach where each individual model in the ensemble makes a prediction, and the final prediction is determined by majority voting. In other words, the class label that receives the most votes from the individual models is selected as the final prediction. This approach treats all models equally and assigns equal weight to their predictions.

Soft voting takes into account the probability or confidence scores assigned by each individual model for each class label. Instead of considering only the majority vote, it calculates the average probability for each class label across all the models. The class label with the highest average probability is chosen as the final prediction.

7.3 Bagging & Pasting

You can create an ensemble of the same type of learner in two different ways. For one, called bagging (or bootstrap aggregating), you train each model on a different subset of the training dataset by sampling with replacement. Alternatively, for pasting, you train each model on a different subset of the training dataset by sampling without replacement. Both of these methods scale very well, since training can happen in parallel, using different CPUs.

Since bagging introduces more diversity to data subsets, bagging ensembles end up higher bias than pasting; individual member models end up less correlated to one another, and so produce an ensemble with lower variance.

Due to the properties of random sampling with replacement, each member model ends up being trained on only part of the available data, with the remaining datapoints (called Out-of-Bag instances) being available for validation set-esque evaluation. This can be aggregated across member models to get an accuracy score.

7.4 Random Forest

The ensemble of trees used in bagging tends to be highly correlated; we can create a better ensemble that produces less correlated trees. A random forest is an ensemble of decision trees trained using the bagging method. By default, each decision tree samples n features with replacement out of the n features in total, and trains the tree on that subset. A version of the random forest model, called an extra trees model, uses random thresholds for splitting criteria (and deciding whether to use it as opposed to a random forest can be decided by cross-validation).

It is possible to create feature importance measures through random forests by measuring a weighted average of the impurity reduction attained through splits that use a particular feature (also called the mean decrease in impurity). The biggest advantage of the MDI is speed of computation; all needed impurity values are computed during training. The drawback of the method is its tendency to prefer numerical features and categorical features with high cardinality.

Feature importance in random forests can also be assessed using permutation importance. Permutation importance measures the impact of each feature on the model's performance by randomly permuting the values of a specific feature and observing the change in the model's predictive accuracy. If a feature is crucial, its permutation is likely to significantly decrease the model's accuracy, indicating its importance. In contrast, less important features are expected to have a minimal effect on performance when permuted. By calculating the average decrease in accuracy across multiple permutations, permutation importance provides a reliable measure of feature importance in random forests.

There is generally one advantage and one disdvantage of using random forests:

- (Advantage) Increasing the number of trees in the forest generally does not increase the risk of overfitting. This is because of the element of randomness; this randomness and diversity help prevent overfitting by reducing the variance and capturing different aspects of the data. This is also due to aggregation; aggregating the predictions of multiple trees to make the final prediction tends to cancel out the individual biases and errors of single trees.
- (Disadvantage) When the number of predictors is large, but the number of relevant predictors is small, random forests can perform poorly. This is since in each split, the chances of selecting a relevant predictor will be low and hence most trees in the ensemble will be weak models.

7.5 Boosting

An ensemble method that uses multiple weak learners to arrive at a strong composite learner is called boosting. Adaptive Boosting, or AdaBoost combines several weak classifiers into a stronger classifier. A weak classifier is a model that performs slightly better than random guessing (like a decision stump – a one-level decision tree). The algorithm initially assigns equal weights to all training examples. It then trains a weak classifier on the training data and calculates the error rate of the classifier. The weak classifier's performance determines its contribution to the final prediction. AdaBoost gives higher weight to misclassified examples from previous models, which enables subsequent weak classifiers to focus on those examples. This process is repeated for a specified number of iterations, and then each weak classifier is assigned a weight based on its performance overall. AdaBoost tends to give more weight to the more accurate classifiers, effectively "boosting" their influence in the ensemble. Multiclass versions of AdaBoost exist, called SAMME which outputs multi-class predictions, and SAMME.R which outputs class probabilities.

Adaboost uses an exponential loss function to update weights and combine weak classifiers. The exponential loss function is given by:

$$\text{Exponential Loss} : \frac{1}{N}\sum_{n=1}^{N} e^{y_n \hat{y}_n}$$

Where y_n is one of $-1, 1$. Exponential loss is differentiable and is an upper bound on error. This allows us to analytically solve for the optimal learning rate hyperparamter.
A different boosting method, called Gradient Boosting, mainly used for regression problems (as oopposed to AdaBoost, which is primarily used for classification), fits a model on the dataset, and then continuously trains models on the residual errors made by the previous model in the chain. Predictions are then made simply by adding up the predictions of each model. This type of ensemble allows for a type of early stopping where we decide when to stop adding more trees when we see that a certain number of them are resulting in little to no improvement in validation accuracy.

Gradient boosting can be interpreted as a gradient descent algorithm by considering the optimization process it employs. The algorithm builds a predictive model by sequentially adding weak models (typically decision trees) to the ensemble. Its goal is to minimize a loss function by iteratively updating the ensemble model. Each weak model in the ensemble is trained to minimize the gradient of the loss function with respect to the predictions made by the current ensemble. By iteratively minimizing the loss function's gradient, gradient boosting aims to reach a point in the parameter space that corresponds to the optimal model.

7.6 XGBoost

XGBoost, short for Extreme Gradient Boosting, is a powerful and highly efficient gradient boosting algorithm. What sets XGBoost apart from other gradient boosters is its ability to handle large-scale datasets with high-dimensional features. It implements several key optimizations, such as parallel tree construction, approximate histogram-based splitting, and regularization techniques, to deliver superior performance and speed. XGBoost also supports custom loss functions, allowing users to optimize for specific objectives. Furthermore, it provides built-in capabilities for handling missing values and supports advanced features like early stopping and cross-validation.

7.7 Stacking

Stacking involves training multiple diverse base models on the same dataset. Cross validation is used to train models and then create model predictions for all train datapoints (in each fold, we predict values for the validation set). The predictions from these base models are then used as input features for a higher-level model called the meta-model. The meta-model learns to make predictions based on the outputs of the base models. Stacking can be used with various machine learning algorithms as base models, such as decision trees, support vector machines, or neural networks.

7.8 Blending

Blending is similar to stacking, but doesn't involve cross-validation training. From the start, we take out a holdout set from our training data, and then train our base models on the remainder of the training set. Then, we use the base models to create predictions for our holdout set, and combine these predictions as new features along with out holdout set. We then train our meta-model on this expanded holdout set and use it to create our final predictions.

Chapter 8

Dimensionality Reduction

8.1 The Curse of Dimensionality

Although it is generally recommended to train your models on the entirety of the design matrix, sometimes doing so is less than ideal. The *curse of dimensionality* refers to the trouble high-dimension datasets cause when it comes to training machine learning models. A great way to tackle this issue is reducing the dimensionality of the data, which will decrease the time necesssary to train models. Even though reducing dimensionality can filter out some of the noise in a dataset resulting in higher performance, usually this doesn't happen; the only tangible improvement is training speed.

High-dimensional datasets tend to be very sparse because as the number of dimensions increases, the available data points become more spread out across the different dimensions, leaving empty regions. The distances between points increase, making it harder to find meaningful patterns and extrapolate findings from training instances onto test ones.

One common way to reduce dimensionality involves projection: projecting multi-dimensional data onto a lower-dimensional space because the points as they are are very close to lying in a lower dimensional subspace of the true dimension of the data.

Another common way is manifold learning. Manifold learning is a technique used to uncover the underlying structure or geometry of high-dimensional data. It aims to represent the data in a lower-dimensional space called a manifold, where the essential features and relationships of the data are preserved.

The manifold assumption is a fundamental concept in manifold learning. It assumes that high-dimensional data lies on or near a lower-dimensional manifold embedded within the high-dimensional space. In simpler terms, it suggests that complex high-dimensional data can be described by a lower number of underlying variables or parameters. By exploiting this assumption, manifold learning algorithms aim to discover and model this lower-dimensional structure, enabling better data representation, visualization, and analysis.

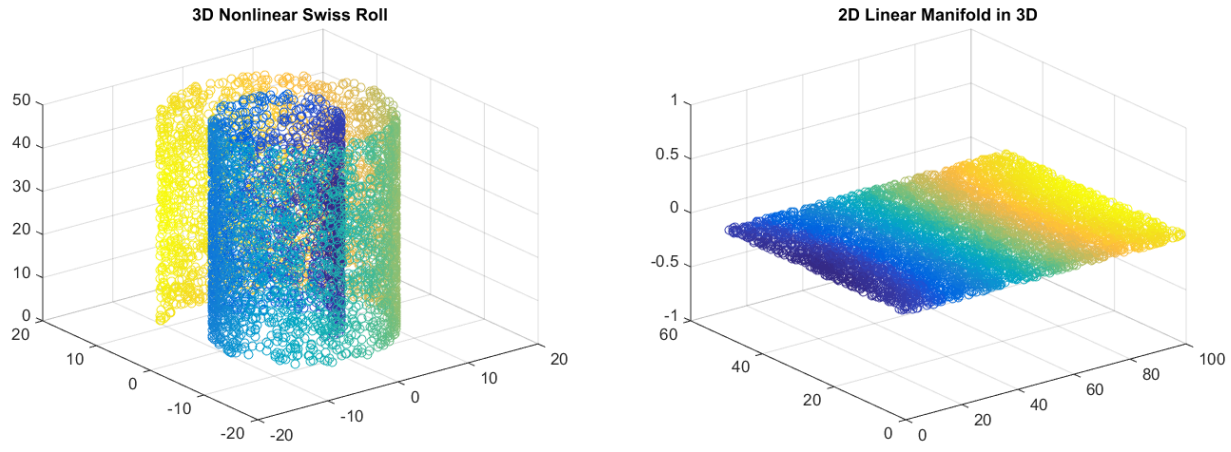


Figure 8.1: The Swiss Roll Dataset in 3D & Unrolled Forms

8.2 PCA: Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique used to identify the most important features or directions (principal components) in a dataset. It achieves this by projecting the original data onto a new coordinate system where the dimensions are ordered based on their variance. PCA utilizes Singular Value Decomposition (SVD) to perform the transformation. SVD decomposes a matrix into three matrices: U, Σ, and V.

To perform dimensionality reduction using PCA, we do the following:

1. Calculate the mean of each feature in the dataset and subtract it from the corresponding feature values. This centers the data around the origin.
2. Apply SVD to the design matrix X. This gives the decomposition $X = U\Sigma V^T$, where U contains the left singular vectors, Σ is a diagonal matrix with singular values, and V^T contains the right singular vectors. The singular values in Σ represent the variance explained by each principal component. They can also be used to determine the amount of information retained in the reduced space, allowing for dimensionality reduction with minimal loss of information.
3. Select the first d columns of V corresponding to the largest singular values. These represent the principal components (all of which are orthogonal to one another). Call this new projection matrix W_d
4. To project a new data point x from the original n-dimensional space to the reduced d-dimensional space, calculate the dot product of x and W_d, i.e., $x_{\text{new}} = x \cdot W_d$.

Performing PCA allows us to find the explained variance ratios of each component, and as such understand how much information the last principal components carry. This allows us to decide how much of the variance from the original dataset we want to preserve (and as a result how many components to keep).

8.3 Random Projection

Random projection is a dimensionality reduction technique that maps high-dimensional data to a lower-dimensional space using random linear projections. It leverages the Johnson-Lindenstrauss lemma, which guarantees that high-dimensional data can be mapped to a lower-dimensional space while maintaining the relative distances between the points reasonably well and within a controlled distortion limit. Random projection is a computationally efficient approach that can be faster than PCA for large datasets, and is particularly useful when the data has a high intrinsic dimensionality or exhibits nonlinear relationships that PCA may struggle to capture.

8.4 Locally Linear Embedding

Locally Linear Embedding (LLE) is a non-linear dimensionality reduction technique that aims to preserve the local relationships and geometry of the data. It works by finding a lower-dimensional representation of the data in which the local relationships between neighboring data points are preserved. LLE achieves this by constructing a weight matrix that encodes the linear relationships between each data point and its neighbors, and then finding the low-dimensional coordinates that best reconstruct the data based on these linear relationships.

Chapter 9

Neural Network Architectures

9.1 Universal Approximation Theorem

The Universal Approximation Theorem (UAT) is a fundamental result in the field of neural networks, which states that a feedforward neural network with a single hidden layer containing a sufficient number of neurons can approximate any continuous function to any desired degree of accuracy, provided the activation function is non-constant, bounded, and continuous. In other words, a neural network with just one hidden layer can, in theory, learn to approximate any complex function, making it a universal function approximator. This theorem is crucial because it demonstrates the expressive power of neural networks, showing that they are capable of representing a wide range of functions, and have the potential to learn highly complex mappings between inputs and outputs.

Although complex architectures are not strictly necessary, these architectures can drastically increase the speed and efficacy of a solution found by neural networks. In recent years, different architectures have risen for their own unique purposes.

9.2 Perceptron

9.2.1 Function

The perceptron functions as a binary classifier. It learns to classify input data into two classes based on the decision boundary it learns during training. By adjusting the weights using the Perceptron Learning Rule, it adapts to better classify new data points. The activation function allows the perceptron to model complex decision boundaries.

9.2.2 Structure

The perceptron consists of the following components:

- Inputs: $x_1, x_2, \ldots, x_n$, representing input features or attributes.
- Weights: $w_1, w_2, \ldots, w_n$, associated with each input, determining their importance.
- Weighted Sum: The perceptron computes the weighted sum as $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n$.
- Activation Function: An activation function $\phi(z)$ introduces non-linearity and maps the weighted sum to the perceptron's output.
- Output: The perceptron's output $y = \phi(z)$, indicating the binary classification result.

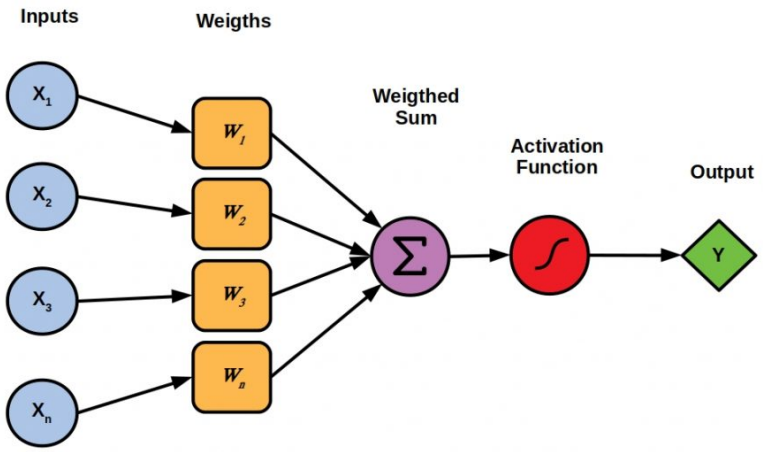


Figure 9.1: Perceptron Structure

9.2.3 Perceptron Learning Rule

The Perceptron Learning Rule is an iterative algorithm used to train a single-layer perceptron for binary classification tasks. Here's how it works:

1. Initialize Weights: Start with random or small initial values for the weights $(w_1, w_2, \ldots, w_n)$ and a bias term b associated with the perceptron.

2. Training Process: The perceptron is trained iteratively using a labeled training dataset with examples from both classes.

3. For each input sample $x = (x_1, x_2, \ldots, x_n)$:

   - Compute the weighted sum $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$.
   - Apply the activation function $\phi(z)$ to obtain the predicted output $y = \phi(z)$. Common activation functions include the step function (for binary output) or the sigmoid function (for continuous output).
   - If the predicted output matches the true label (y equals the target label t), no update is required.
   - If the prediction is incorrect (y differs from t), update the weights and bias using the learning rate γ:

$$w_i \leftarrow w_i + \gamma \times (t - y) \times x_i$$
$$b \leftarrow b + \gamma \times (t - y)$$

4. Repeat the above steps for all input samples in the training dataset for multiple epochs (complete passes through the dataset).

5. Convergence: The training process continues until the perceptron converges to a decision boundary that correctly classifies the training examples, or until a maximum number of epochs is reached.

9.2.4 Limitations

The perceptron has several limitations:

- Binary Classification: It is limited to binary classification tasks and requires extensions for multi-class classification.
- Linear Boundaries: The perceptron can only learn linear decision boundaries and struggles with non-linearly separable data.
- Convergence Issues: The Perceptron Learning Rule may not converge to the optimal solution for noisy or non-linearly separable data.
- Feature Dependency: The performance depends on the informative representation of data features.

Despite its limitations, the perceptron played a pivotal role in the development of neural networks and inspired more advanced architectures used in modern machine learning and deep learning.

9.3 Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) is a type of artificial neural network with multiple layers of interconnected neurons. MLPs are powerful models capable of learning complex patterns but require careful training and hyperparameter tuning. Advanced architectures like CNNs and RNNs have gained popularity, but MLPs remain foundational in neural networks. Here's a detailed summary of how an MLP operates and functions:

9.3.1 Structure & Components

- Input Layer: The first layer that receives input data features. Each neuron corresponds to a feature.
- Hidden Layers: Intermediate layers that process and transform input through weighted connections.
- Output Layer: The final layer that produces network predictions. The number of neurons depends on the task.

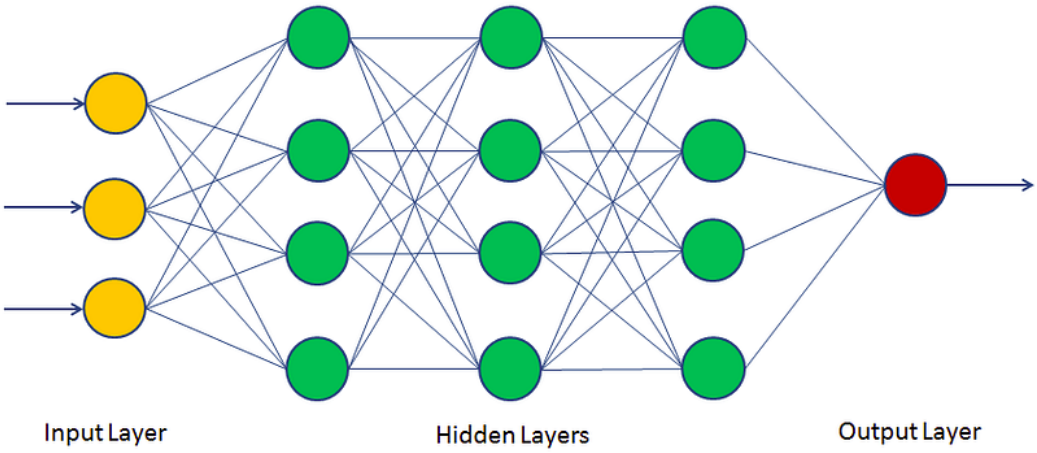


Figure 9.2: MLP Structure

9.3.2 Neurons & Connections

The MLP is structred as follows:

- Neurons in hidden and output layers receive inputs from all neurons in the previous layer, multiplied by corresponding weights.
- A bias term is added to introduce a shift in the activation function's output.
- The weighted sum with the bias is passed through an activation function to introduce non-linearity.

9.3.3 Forward Propagation

Forward propagation passes input data through the MLP to obtain predictions. The data then flows through hidden layers, and activations are calculated using weighted sums and activation functions. The process continues until the output layer is reached, producing final predictions.

9.3.4 Backpropagation

- MLP training involves adjusting weights and biases to minimize prediction error.
- Backpropagation propagates error backward from output to hidden layers, updating weights and biases using gradient descent.
- The goal is to minimize a chosen loss function, like mean squared error for regression or cross-entropy for classification.

9.3.5 Activation Functions

Activation functions introduce non-linearity to model complex relationships in the data. They are applied at the end of every layer before passing data onto a subsequent layer. Common activation functions include the sigmoid, ReLU, and tanh functions.

9.4 Wide & Deep Neural Network

A Wide & Deep Neural Network model is an extension of the standard Multi-Layer Perceptron (MLP) architecture. It combines a wide component for capturing feature interactions and a deep component for learning hierarchical representations. Here's a brief summary of how an MLP can be expanded to create a Wide & Deep Neural Network model:

1. Wide Component:
    - The wide component adds additional "cross-feature" connections to the network.
    - These connections enable the model to learn explicit feature interactions, capturing complex relationships between features.
    - The wide component typically includes a shallow, wide layer with a large number of neurons.
2. Deep Component:
    - The deep component remains similar to the standard MLP architecture, with multiple hidden layers.
    - The deep layers learn hierarchical representations of the data, capturing intricate patterns and higher-level features.
3. Integration:
    - The outputs from the wide and deep components are combined at the final layer of the model.
    - The combined outputs are used to make predictions for the task, such as classification or regression.

The Wide & Deep Neural Network model leverages the strengths of both components, making it particularly effective in recommendation systems and tasks requiring feature engineering and capturing feature interactions.

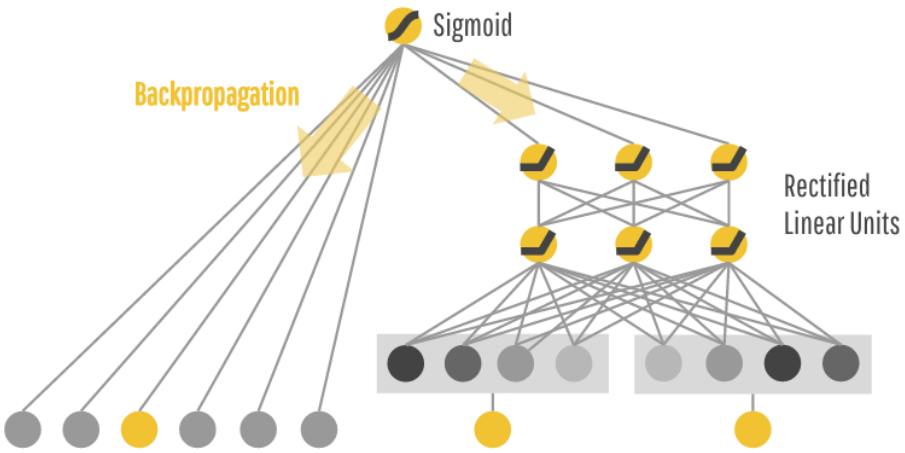


Figure 9.3: Example Wide & Deep Network Diagram

9.5 Convolutional Neural Networks

9.5.1 Usefulness

The usefulness of CNNs shines due to the three main limitations of regular MLPs:

1. **Limited Scalability for Images**: MLPs are well-suited for tabular data but may struggle to effectively process images. Treating each pixel as a separate input feature can lead to an explosion in the number of parameters as the image size increases. This scalability issue results in longer training times and increased risk of overfitting, making it challenging to train deep MLP architectures on image data.

2. **Ignoring Spatial Information & Pixel Correlation**: A significant drawback of MLPs on images is their inability to capture spatial relationships between pixels. Images often contain local patterns and structures that require models to understand the correlations between neighboring pixels. MLPs treat each pixel independently, missing out on valuable spatial information. This limitation undermines their performance on image-related tasks, such as image classification and object detection.

3. **Inability to Handle Translations**: MLPs lack the capacity to handle translations within images. They consider shifted versions of the same pattern as distinct inputs, making it harder to recognize objects that may appear in various positions.

9.5.2 Structure

A CNN comprises three fundamental components that contribute to its exceptional performance in image analysis tasks:

- **Convolutional Layers:** These layers apply filters through convolutions over input images.
- **Pooling Layers:** Pooling layers downsample feature maps; they retain essential information while discarding redundancy, and so enhancing computational efficiency.
- **Fully Connected Neural Network:** This component utilizes learned features from previous layers for classification and prediction.

The first two types of layers are strategically placed at different points in a CNN, while the FCNN is usually placed at the very end of the neural net's structure.

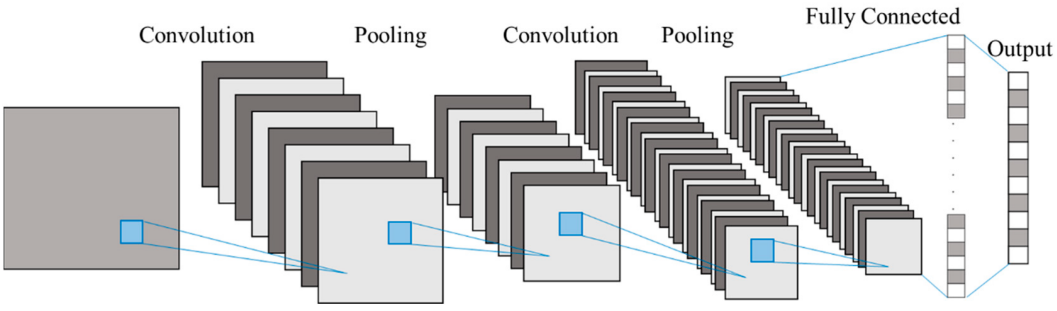


Figure 9.4: Example CNN Diagram

9.5.3 Convolutional Layer

The core operation of a convolutional layer is the convolution operation itself. It involves sliding a small filter (also known as a kernel) over the input image's pixel values. The filter is a matrix of learnable weights. At each step, an element-wise multiplication is performed between the filter and the overlapping region of the input image, and the results are summed up. This process produces a single value that represents a feature in that specific location.

Feature Maps

The result of the convolution operation is a feature map, which represents the activations of the filter as it slides across the input image. The feature map highlights where certain features or patterns exist in the input image. Multiple filters can be applied to the same input, resulting in multiple feature maps, each detecting a different pattern.

Padding

There are two primary types of padding used in CNNs: zero-padding and full padding:

- Zero-Padding: Involves adding zeros (or a specified constant value) to the edges of the input data. This type of padding is often used to ensure that the output feature map after convolution has the same spatial dimensions as the input. Zero-padding is particularly useful when you want to maintain the size of the input data throughout the convolutional layers. The amount of padding to be added depends on the size of the filter (kernel) and the desired output size. If the filter has a size of $(F \times F)$ and you want to maintain the original input size, then you would add $(F - 1)/2$ pixels of padding to each side of the input.
- Full-Padding: Also known as "same padding," involves adding padding to the input data in such a way that all pixels are visited the same number of times by the filter. For a filter of size $(F \times F)$, the amount of padding to be added to each side of the input is $F - 1$, which effectively increases the size of the output when compared to the input.

Stride

Stride determines how much the filter moves (slides) across the input image. A stride of 1 means the filter moves one pixel at a time. Larger strides reduce the spatial dimensions of the output feature maps.

Activation Function

After the convolution operation, an activation function, most likely ReLU, is applied element-wise to the resulting feature map. The activation function introduces non-linearity, enabling the network to learn complex relationships.

Mathematical Explanation

The mathematical formula for a single element of the convolution operation involves element-wise multiplication between the filter and window/input matrices. The exact formulation is as follows:

$$\text{Output}[i, j] = \sum(\text{Filter} * \text{Input}[i : i + H, j : j + W]) + \text{Bias}$$

Where:

- $\text{Output}[i, j]$ is the value at position (i, j) in the output feature map.
- Filter is the filter (kernel) matrix.
- $\text{Input}[i : i + H, j : j + W]$ is the region of the input image being convolved.
- Bias is a bias term added to each element of the feature map.

The filters' weights are learned during the training process through backpropagation and gradient descent, allowing the network to adapt and recognize meaningful patterns in the input data.

9.5.4 Pooling Layer

In CNNs, a pooling layer is a vital component designed to downsample the spatial dimensions of feature maps. Pooling helps reduce computation, extract key features, and improve pattern detection across varying positions. Pooling involves dividing the input feature map into non-overlapping regions, called pooling windows. Each window undergoes a pooling operation to aggregate information, resulting in a smaller representation while retaining crucial features. Here are the most common types of pooling:

- **Max Pooling**: In max pooling, within each pooling window the maximum value from the input feature map is chosen and transferred to the corresponding position in the pooled output. Max pooling effectively detects features, even if they're slightly shifted or distorted within the window.
- **Average Pooling**: Average pooling computes the average of elements within each window and assigns that average to the corresponding position in the pooled output. It provides a smoother representation and aids in noise reduction and trend preservation.
- **Global Average Pooling**: Global Average Pooling is distinct, as it reduces the entire feature map to a single value by computing the average of all elements. It's often used for transitioning from convolutional to fully connected layers.

9.5.5 Fully Connected Segment

CNNs typically have a fully connected neural network attached to its end, after all convolutional and pooling layers have been completed. This allows the network to process simple and complex features constructed from the combinations of different patterns that appear in images.

9.5.6 Dropout on CNNs

Dropout doesn't work on CNNs quite the same way it does on FCNNs. The reason this is true is a little nuanced, but can be explained by representing a element-wise filter-input multiplication operation in matrix-vector multiplication form:

Applying dropout in this scenario would result in the "zeroing-out" of one of the columns of the matrix in question. Though, if looking at the w_4 weight value alone, it is easy to see that dropout in one of the columns would not necessarily prevent an update of the w_4 weight. This is the exact issue with the use of regular dropout on CNNs.

9.5.7 Occlusion on CNNs

Occlusion analysis is a technique employed to evaluate CNNs and understand the significance of various regions or features within an image for the network's predictions. It involves systematically obscuring parts of an input image and observing the resulting changes in the network's output, shedding light on feature importance and generalization. The occlusion analysis technique is executed as follows:

1. **Loss Calculation:** Get the classification predictions of the input (not occluded) image. Then calculate the loss based on those probabilities ($L_{\text{no occlusion}}$.

2. **Select Region of Interest:** A specific portion of the image is chosen for occlusion, often in the form of a small square or other shapes.

3. **Occlude the Region:** The chosen region is covered or occluded with a neutral color (usually grey), effectively erasing the visual information within that area.

4. **Predictive Assessment:** The occluded image is fed into the CNN, and the network predicts the label or class for the occluded image.

5. **Analyze Prediction Change:** Calculate the loss of the occluded image $L_{\text{occluded } i}$, then calculate the difference in the loss: $\delta L = L_{\text{no occlusion}} - L_{\text{occluded } i}$. Minimal change suggests the region's insignificance, while substantial change indicates its relevance.

6. **Iterative Process:** This process is iterated across various regions in the image, offering a comprehensive understanding of the network's sensitivity to different features.

9.5.8 Class Model Visualization

Class model visualization for CNNs involves generating representative images per class by identifying the values per pixel that maximize the accuracy score. The process aims to find the image that would receive the highest accuracy score for a specific class; this is achieved by iteratively optimizing the image to enhance its resemblance to the class, guided by the network's weights (maximizing the score of a class is usually achieved by minimizing the scores of all other classes).

9.5.9 Saliency Maps

A saliency map is a visualization method used to gain insights into which specific regions of an input image contribute significantly to the network's predictions. It is a backwards-pass approach that allows us to pinpoint the most influential parts of the image that drive the network's decision-making process.

The process involves calculating gradients or sensitivity scores for each pixel in the image with respect to the network's output. Pixels that exhibit higher gradients or sensitivity scores are deemed more influential, indicating their substantial contribution to the network's predictions. These scores effectively highlight the regions that the network "pays attention to." Saliency maps can assist in debugging models and refining their performance by identifying regions that might lead to incorrect or biased predictions. They can also help verify whether the network is focusing on relevant visual cues.

9.5.10 Common Architectures

LeNet-5

LeNet-5 was primarily developed for handwritten digit recognition and played a pivotal role in popularizing CNNs for image classification tasks. LeNet-5's success demonstrated the efficacy of hierarchical feature learning in deep neural networks, laying the foundation for modern CNN architectures and their applications in computer vision.

Layer Type	Number of Maps	Dimension	Kernel Size
Input	1	32x32x1	–
Convolution	6	28x28x6	5x5
Average Pooling	6	14x14x6	2x2
Convolution	16	10x10x16	5x5
Average Pooling	16	5x5x16	2x2
Convolution	120	1x1x120	5x5
Fully-Connected	–	84	–
Fully-Connected	–	10	–

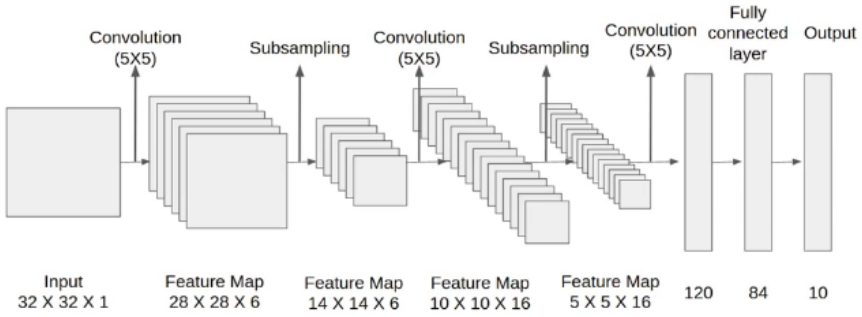


Figure 9.5: LeNet-5 Structure

AlexNet

AlexNet is a pioneering convolutional neural network (CNN) architecture that played a pivotal role in advancing deep learning for image recognition tasks. AlexNet introduced several groundbreaking concepts, such as the use of Rectified Linear Units (ReLU) for activation, back-to-back convolutional layers, dropout for regularization, and GPU acceleration for efficient training. Its success demonstrated the potential of deep CNNs in achieving exceptional performance on large-scale image classification tasks.

Layer Type	**Number of Maps**	**Dimension**	**Kernel Size**
Input	3	227x227x3	–
Convolution	96	55x55x96	11x11
Max Pooling	96	27x27x96	3x3
Convolution	256	27x27x256	5x5
Max Pooling	256	13x13x256	3x3
Convolution	384	13x13x384	3x3
Convolution	384	13x13x384	3x3
Convolution	256	13x13x256	3x3
Max Pooling	256	6x6x256	3x3
Fully-Connected	–	4096	–
Fully-Connected	–	4096	–
Output	–	1000	–

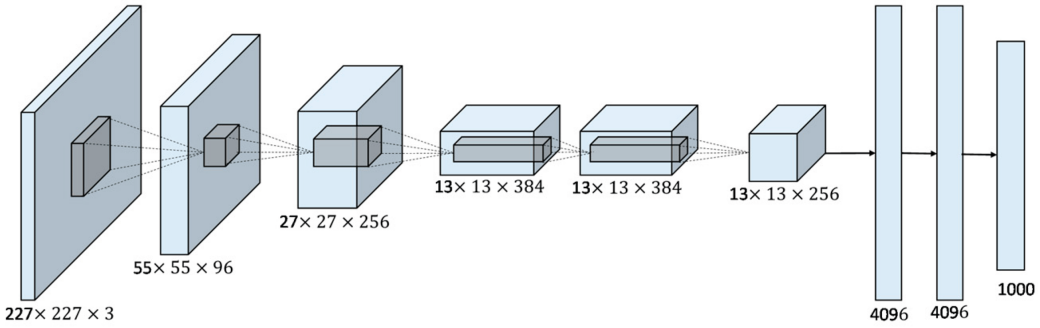
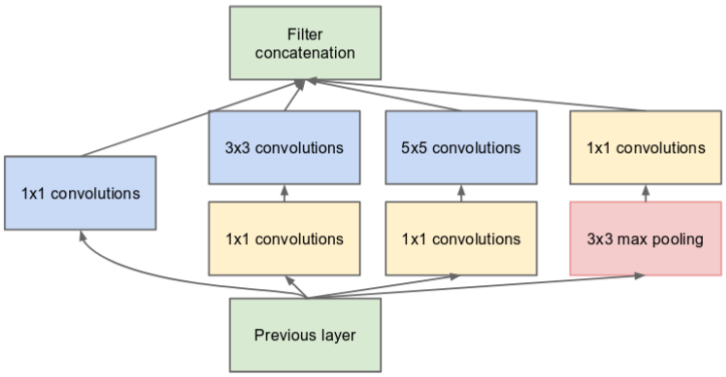


Figure 9.6: AlexNet Structure

GoogLeNet

GoogleNet is an advanced CNN architecture popularized due to the use of "inception blocks," which are modules that perform multiple parallel convolutions of different kernel sizes and then concatenate their outputs. This approach allows the network to capture features at different spatial scales and learn representations with varying levels of abstraction. Inception blocks significantly increase the depth and width of the network without dramatically increasing the computational cost.



A notable feature of inception blocks is their use of 1×1 convolutions, which have the following benefit:

1. **Capturing Patterns Across Channels:** A 1x1 convolution is used to capture patterns and relationships along the depth dimension of the input feature maps.
2. **Outputting Fewer Maps than Inputs:** 1x1 convolutions also enable dimensionality reduction by outputting a smaller number of feature maps than their inputs. This reduction in the number of channels helps control the computational complexity of the network.

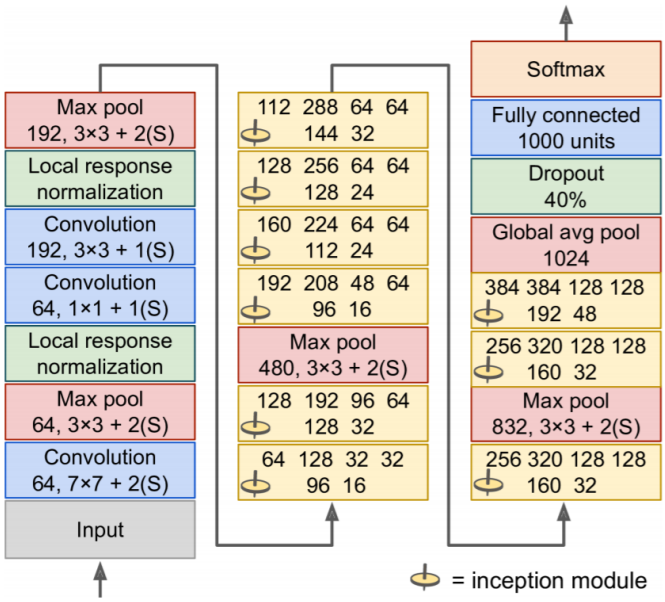


Figure 9.7: GoogLeNet Structure

ResNet

Another variation of a CNN architecture is ResNet, which improves on previous models by using residual blocks. A residual block is a component designed to overcome challenges like the vanishing gradient problem and enable the training of extremely deep neural networks. Residual blocks allow CNNs to learn residual functions, which capture the difference between input and desired output.

Each residual block consists of two main parts: the identity path and the residual path. The identity path directly passes the input data through the block unchanged. The residual path comprises a sequence of convolutional and activation layers, aimed at learning the residual function that represents desired changes to the input. Mathematically, if X represents the input to the residual block, the output Y is obtained by adding the output of the residual path (representing the learned residual) to the input X:

$$Y = X + F(X)$$

Here, $F(X)$ represents the output of the residual path, which captures the residual function. By adding the residual, the network learns to correct discrepancies or deviations of the input from the desired output.

The inclusion of residual blocks in ResNet efficiently trains deeper networks by addressing the vanishing gradient problem. The skip connections provided by these blocks enable gradients to flow through the identity path even during training, facilitating the learning of intricate mappings.

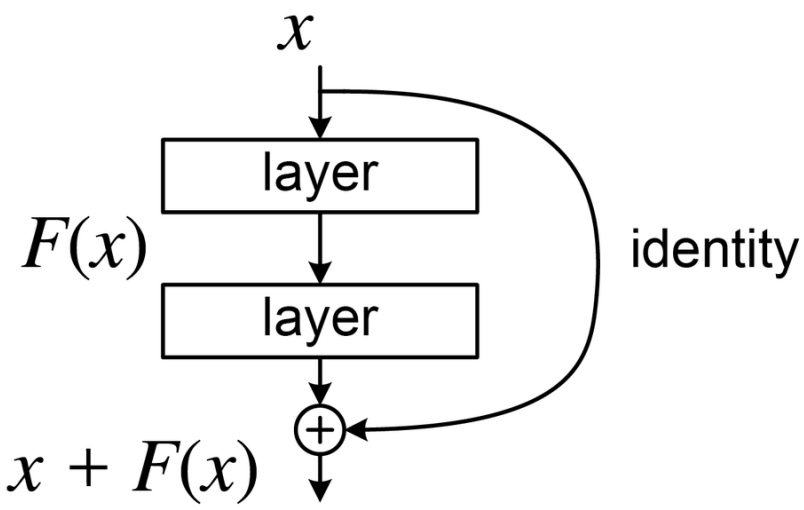


Figure 9.8: Residual Block Structure

Chapter 10

Tuning Neural Networks

10.1 Limitations

Neural networks are prone to two main limitations, both due to the nature of their dependence on gradient descent during training. These two limitations are outlined below.

10.1.1 Vanishing Gradients Problem

The vanishing gradients problem arises when the gradients of the loss function with respect to the network's weights become extremely small as they are backpropagated through the layers during training. Consequently, the network's early layers receive very small updates to their weights, slowing down learning significantly. This can cause these layers to effectively stop learning, leading to a lack of progress in the training process. The vanishing gradients problem comes about due to the nature of certain activation functions (e.g., sigmoid and tanh) that squash their input values to a limited range (e.g., (0, 1) or (-1, 1)). When the input values are close to the ends of these ranges, the gradients of these functions become very close to zero. Since gradients are multiplied during backpropagation, the multiplication of multiple small gradients in deep networks results in exponentially smaller gradients for earlier layers, causing the vanishing gradients problem.

10.1.2 Exploding Gradients Problem

The exploding gradients problem occurs when the gradients become extremely large during backpropagation. As a result, the weights of the network receive very large updates, causing instability and making the training process diverge. This can lead to erratic behavior during training and render the network's weight values too large to be practically useful. The exploding gradients problem is most commonly observed when using activation functions with unbounded output ranges, such as the ReLU (Rectified Linear Unit) function. The ReLU function outputs the input value itself for positive inputs, and this can cause the gradients to grow exponentially as they are multiplied during backpropagation. When gradients are excessively large, weight updates can result in unstable and oscillatory behavior, making the network difficult to train.

10.2 Weight Initialization

One way to deal with the vanishing/exploding gradients problem is properly initializing the weight of neural network nodes. The following table outlines the base ways initialization occurs in practice, with the means and variances of each initialization technique, and which activations it is most appropriate with:

Initialization	**Activation Function**	**Mean**	**Variance**
Glorot/Xavier	Sigmoid, Tanh, Softmax	0	$\frac{2}{\text{input\_size}+\text{output\_size}}$
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	0	$\frac{2}{\text{input\_size}}$
LeCun	SELU	0	$\frac{1}{\text{input\_size}}$

Table 10.1: Weight Initializations and Activation Functions

10.3 Activation Functions

1. Sigmoid: The sigmoid function, also known as the logistic function, maps any input value to a range between 0 and 1, making it suitable for binary classification tasks and introducing non-linearity to a network. The sigmoid function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Advantages:

- **Squashing Effect**: The sigmoid function maps a wide range of input values to a compact output range of (0, 1). This "squashing" effect helps normalize the neuron's output and ensures a consistent response from the neuron for a wide range of inputs.
- **Differentiability**: The sigmoid function is smooth and differentiable everywhere, which is essential for gradient-based optimization algorithms used in training neural networks, such as backpropagation.

Disadvantages:

- **Vanishing Gradients**: The sigmoid function has a limited output range between 0 and 1. When the input is very large or very small, the gradients of the sigmoid function become close to zero, causing vanishing gradients. This hinders the training of deep networks, particularly in the early layers.
- **Not Zero-Centered**: The sigmoid function is not zero-centered, leading to a shift in the neuron's output distribution. This can cause difficulties in training as the neurons in subsequent layers might receive only positive or negative inputs, impacting the optimization process.

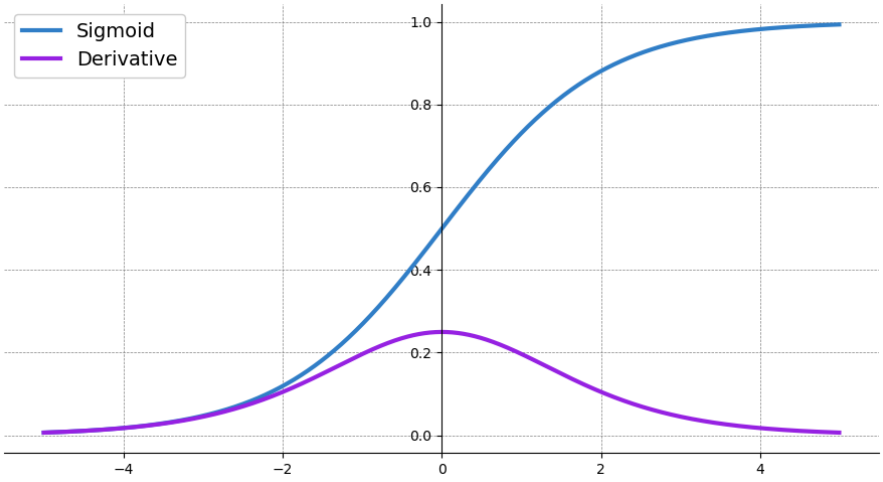


Figure 10.1: Sigmoid Function

2. ReLU: The Rectified Linear Unit (ReLU) function maps any input value to itself if it is positive, and to zero otherwise. The ReLU function is defined as:

$$f(x) = \max(0, x)$$

Advantages:

- **Non-linearity**: ReLU introduces non-linearity to the network, allowing it to learn complex patterns and relationships in the data, making it well-suited for deep neural networks.
- **Computationally Efficient**: The ReLU function is computationally efficient, as it involves simple thresholding operations and does not require exponentials or other costly mathematical operations.

Disadvantages:

- **Dying ReLU Problem**: ReLU neurons can sometimes become "dead" during training, where they always output zero for any input. This happens when the weights are adjusted in such a way that the neuron never activates.
- **Not Suitable for Negative Inputs**: The ReLU function outputs zero for all negative inputs. This can lead to the vanishing gradients problem in some cases, as the gradient for negative inputs becomes zero, and the weights are not updated during backpropagation.

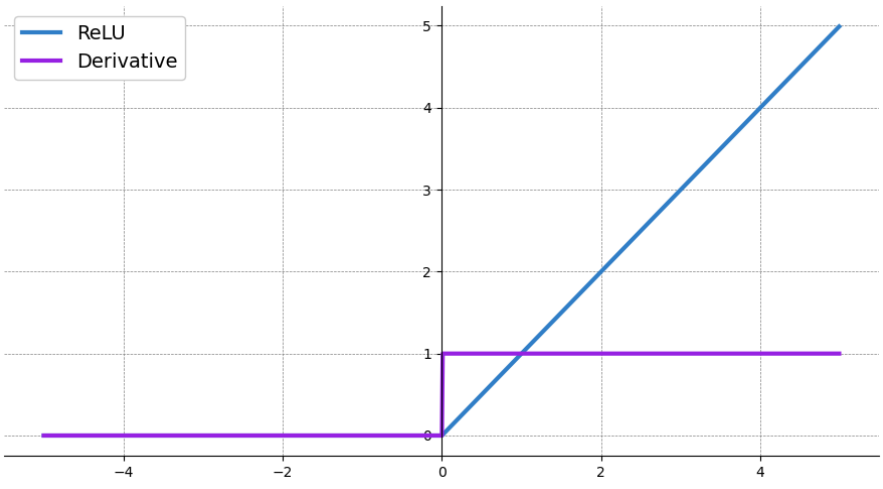


Figure 10.2: ReLU Function

3. Tanh: The hyperbolic tangent function, tanh, maps any input value to a range between -1 and 1, introducing non-linearity to the network. The tanh function is defined as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Advantages:

- **Zero-Centered**: The tanh function is zero-centered, which helps mitigate the shift in the neuron's output distribution and makes the optimization process more stable compared to the sigmoid function.
- **Stronger Non-linearity**: The tanh function provides stronger non-linearity than the sigmoid function, allowing the network to learn more complex relationships and patterns in the data.

Disadvantages:

- **Vanishing Gradients**: Similar to the sigmoid function, the tanh function also suffers from the vanishing gradients problem when the input is very large or very small. This can impede training in deep networks, especially in the early layers.

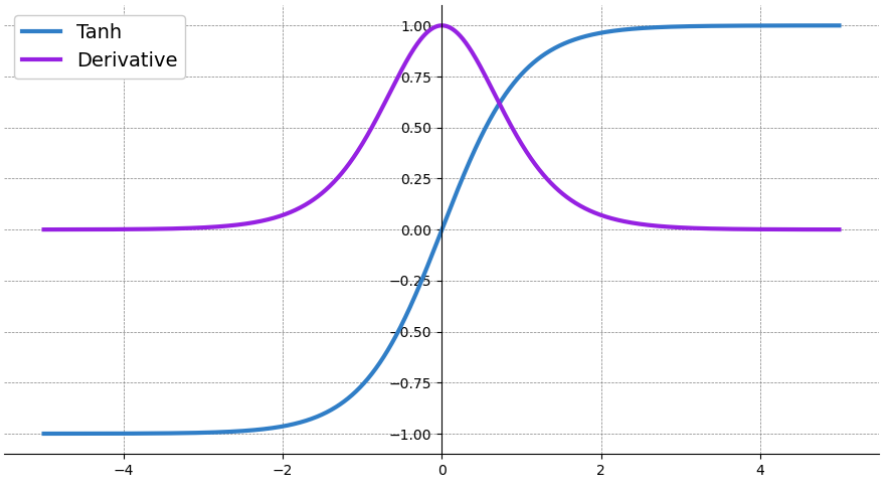


Figure 10.3: Tanh Function

4. Leaky ReLU: The Leaky Rectified Linear Unit (Leaky ReLU) is an alternative activation function that addresses some of the limitations of the ReLU function. It allows a small, non-zero gradient for negative inputs, which helps prevent the "dying ReLU" problem. The function is defined as:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha \cdot x, & \text{otherwise} \end{cases}$$

where α is a small positive constant (usually around 0.01).

Advantages:

- **Preventing "Dying ReLU" Problem**: The Leaky ReLU function allows a small gradient for negative inputs, preventing neurons from becoming "dead" during training. This makes the optimization process more robust and can lead to better learning in deep networks.
- **Computationally Efficient**: Like the ReLU function, the Leaky ReLU function involves simple thresholding operations and is computationally efficient.

Disadvantages:

- **Not Zero-Centered**: The Leaky ReLU function is not zero-centered, so it may still exhibit a shift in the neuron's output distribution, similar to the ReLU function. However, this is less problematic compared to the standard ReLU.
- **Choice of α**: The choice of the small positive constant α is a hyperparameter that needs to be tuned. An inappropriate value of α may still lead to vanishing gradients or not fully prevent the "dying ReLU" problem.

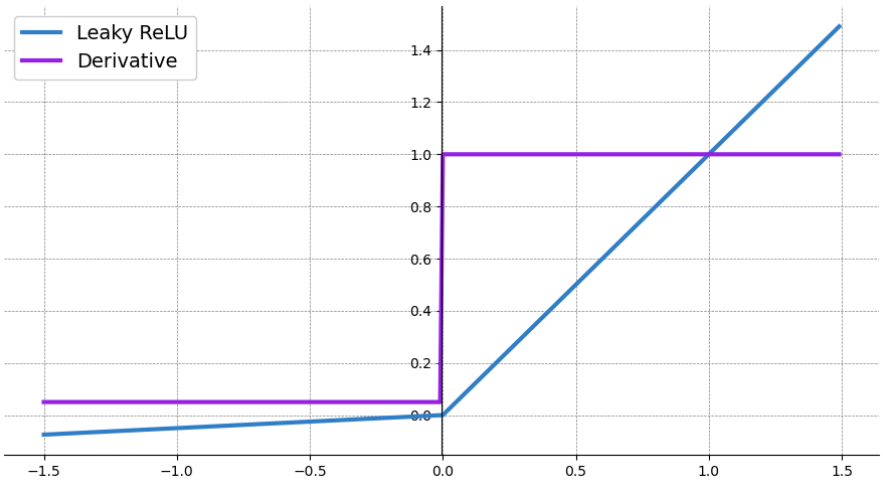


Figure 10.4: Leaky ReLU Function

5. Softplus: The Softplus activation function is a smooth and continuously differentiable function that introduces non-linearity to the network. It is commonly used as an alternative to the ReLU function, especially when a smoother activation is desired. Its functional form is:

$$f(x) = \log(1 + e^x)$$

Advantages:

- **Smoothness and Continuity**: Softplus is a smooth and continuous function, which ensures smooth gradient propagation during backpropagation, contributing to better training stability.
- **Monotonic Behavior**: Softplus is a monotonically increasing function, which avoids the "dying ReLU" problem that can occur with the ReLU activation.

Disadvantages:

- **Vanishing Gradients**: While Softplus avoids the "dying ReLU" problem, it can still suffer from vanishing gradients for very large positive inputs, hindering the training of deep networks.
- **Not Zero-Centered**: Like the ReLU function, Softplus is not zero-centered, which can lead to a shift in the neuron's output distribution and affect optimization in some cases.
- **Limited to Positive Inputs**: Softplus is mainly suitable for activations with positive input values, as it maps all input values to positive outputs.

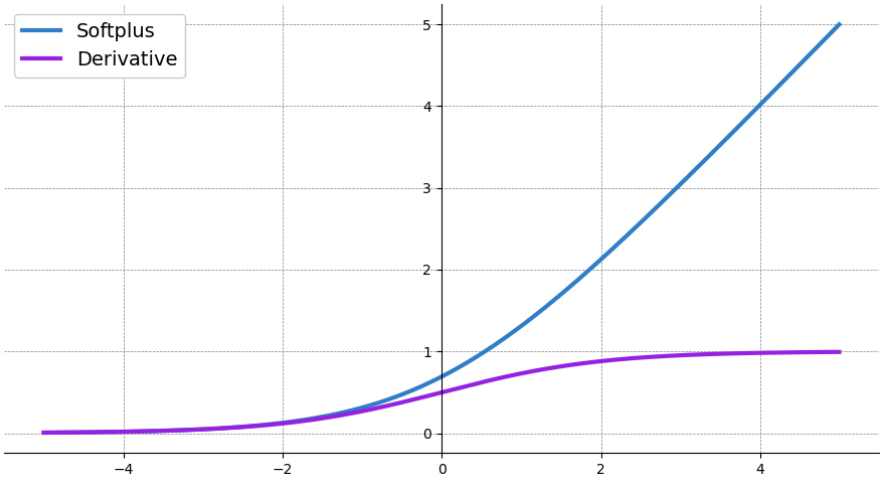


Figure 10.5: Softplus Function

6. ELU: The Exponential Linear Unit (ELU) is a variant of the Rectified Linear Unit (ReLU) that addresses some of its limitations. It introduces a smooth exponential curve for negative inputs, allowing a small, non-zero gradient.

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha \cdot (e^x - 1), & \text{otherwise} \end{cases}$$

where α is a positive constant (usually around 1).

Advantages:

- **Smooth Non-Linearity**: The ELU function provides a smooth non-linearity for negative inputs, avoiding the "dying ReLU" problem and ensuring that neurons remain active even for negative values of x.
- **Approximation of Identity Function**: For positive inputs, the ELU function is identical to the input x, preserving some of the linearity that the ReLU function exhibits for positive values.
- **Zero-Centered for Negative Inputs**: The ELU function is zero-centered for negative inputs, which helps mitigate the shift in the neuron's output distribution, making it more suitable for optimization.

Disadvantages:

- **Computational Cost**: The ELU function involves exponential operations, which can be computationally more expensive than other activation functions like ReLU.
- **Choice of α**: The choice of the positive constant α is a hyperparameter that needs to be tuned. Different values of α can significantly affect the behavior of the ELU function.

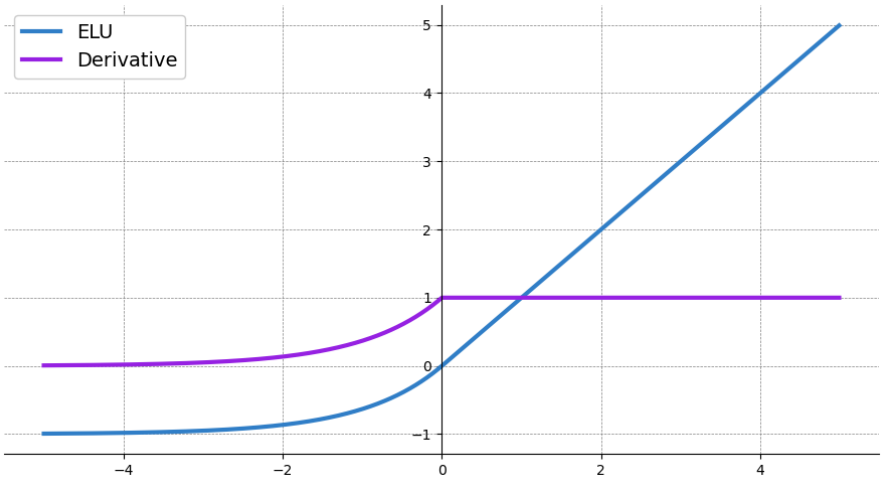


Figure 10.6: ELU Function

7. SELU: The Scaled Exponential Linear Unit (SELU) is a self-normalizing variant of the Exponential Linear Unit (ELU) that helps preserve zero mean and unit variance of input values during training, enabling deep networks to be more stable and reliable.

$$f(x) = \lambda \cdot \begin{cases} x, & \text{if } x > 0 \\ \alpha \cdot (e^x - 1), & \text{otherwise} \end{cases}$$

where λ and α are positive constants (usually set to 1.0507 and 1.67326, respectively).

Advantages:

- **Self-Normalizing Property**: The SELU function helps maintain the mean and variance of activations close to 0 and 1, respectively, throughout the layers of a deep neural network, which aids in better training convergence and stability. This also helps alleviate the vanishing and exploding gradients problem often encountered in deep networks. *This, however, can only happen when inputs are standardized, LeCun is the weight initialization strategy, and the net structure is a pure MLP.*

Disadvantages:

- **Limited to Specific Architectures**: While SELU has shown promising results in some deep architectures, it may not be universally suitable for all types of neural networks, and its performance can be sensitive to hyperparameter tuning.
- **Non-Zero-Centered for Positive Inputs**: Unlike ELU, SELU is not exactly zero-centered for positive inputs, which can still introduce a shift in the output distribution, although the self-normalization property helps mitigate this effect.

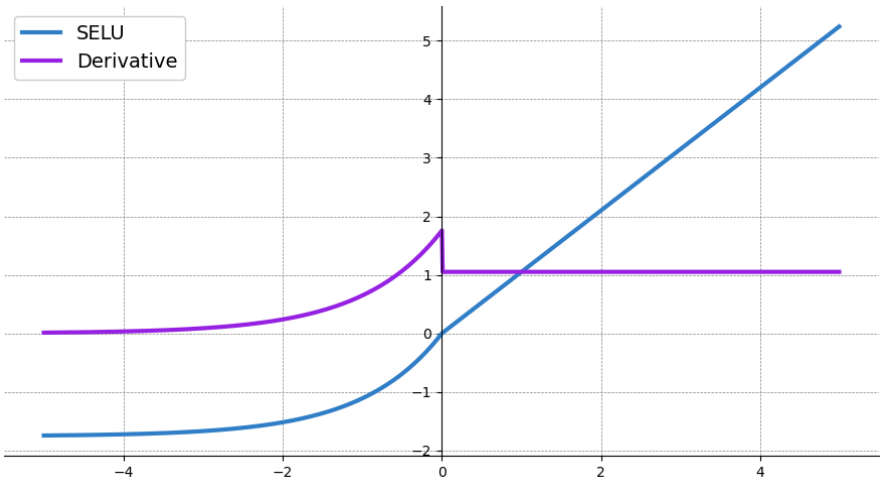


Figure 10.7: SELU Function

8. GELU: The Gaussian Error Linear Unit (GELU) is an activation function that approximates the CDF of the standard normal distribution. It was introduced as a smooth alternative to the ReLU function with a continuous and differentiable form. It can be approximated by $x\sigma(1.702x)$. Its functional form is:

$$f(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution.

Advantages:

- **Smoothness and Continuity**: GELU is a smooth and continuous function that allows for smooth gradient propagation during backpropagation, making it suitable for deep neural networks.
- **Approximation of Identity for Positive Inputs**: For positive inputs, GELU behaves similarly to the identity function, which helps retain some linearity for positive values and allows for better learning in certain architectures.
- **Curvature**: The function is neither convex nor monotonic, which can potentially help a net learn more complex decision boundaries.

Disadvantages:

- **Computational Cost**: GELU involves computing the cumulative distribution function of the standard normal distribution, which can be computationally more expensive compared to simpler activation functions like ReLU.
- **Not Zero-Centered**: Like ReLU, GELU is not zero-centered, which can lead to a shift in the neuron's output distribution and affect optimization in some cases.

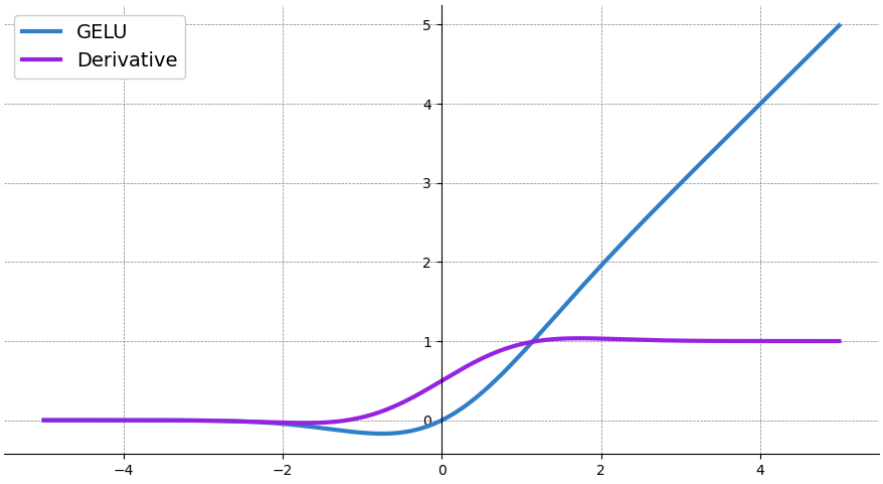


Figure 10.8: GELU Function

9. Swish: The Swish activation function is a smooth and non-monotonic function that was proposed as an alternative to traditional activation functions like ReLU and sigmoid. It introduces a non-linearity to the network while offering a smoother gradient for training.

$$f(x) = x \cdot \sigma(\beta \cdot x)$$

where σ is the sigmoid function, and β is a positive constant (usually set to 1 in practice).

Advantages:

- **Smoothness and Continuity**: Swish is a smooth and continuous function, which ensures smooth gradient propagation during backpropagation, contributing to better training stability.
- **Non-Monotonic Behavior**: Swish has a non-monotonic behavior, which allows it to adapt to different types of data and learning scenarios better.
- **Learnable Beta Parameter**: The β parameter in Swish can be learned during training, offering some flexibility in adapting the activation function to specific data distributions.

Disadvantages:

- **Computational Cost**: Like the GELU function, Swish involves computing the sigmoid function, which can be computationally more expensive compared to simpler activation functions like ReLU.
- **Limited Evidence of Superiority**: While Swish has shown promising results in some cases, it does not consistently outperform other activation functions in all scenarios. Its performance can vary depending on the specific architecture and dataset.

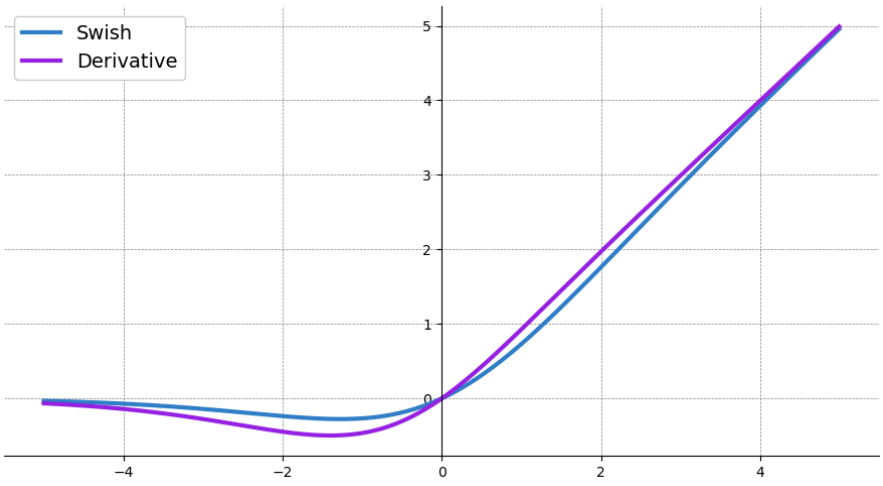


Figure 10.9: Swish Function

10.4 Batch Normalization

Batch normalization (BN) is a technique used to improve the training of deep neural networks. It normalizes the activations of intermediate layers during training, addressing the internal covariate shift. This normalization helps stabilize training, accelerate convergence, and improve generalization.

10.4.1 Parameters

- γ: Scaling parameter. It scales the normalized activations to an appropriate range.
- β: Shifting parameter. It shifts the scaled activations to have the desired mean and variance.
- μ: Population mean. During training, the mini-batch mean μ is computed for each mini-batch. The parameter μ is accumulated over multiple mini-batches and represents the overall mean for the entire training dataset.
- σ: Population standard deviation. Similar to μ, the mini-batch variance σ^2 is accumulated over multiple mini-batches to compute the overall standard deviation for the entire training dataset.

10.4.2 Process

Given a mini-batch of activations $\mathbf{x} = \{x_1, x_2, \ldots, x_m\}$ for a specific layer:

1. Compute the mean μ and variance σ^2 of the mini-batch:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$$

2. Normalize the activations using the mean and variance:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

   where ϵ is a small positive constant added for numerical stability.

3. Scale and shift the normalized activations using learnable parameters γ and β:

$$y_i = \gamma\hat{x}_i + \beta$$

   The parameters γ and β allow the network to learn the optimal scale and shift for each feature, preserving the representational power of the network.

10.4.3 Parameter Training

During the training process, the model updates the learnable parameters γ and β through backpropagation and gradient descent.

1. Gradient Computation: To compute the gradients of γ and β during backpropagation, we need to calculate the gradients of the loss function with respect to the normalized activations $\hat{x}_i$:

$$\frac{\partial \mathcal{L}}{\partial \hat{x}_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \gamma$$

   where $\mathcal{L}$ is the overall loss of the network.

2. Gradient Updates: After computing the gradients, we can update γ and β using gradient descent or any optimization algorithm, similar to the previous explanation.

10.4.4 Inference

During inference (testing or prediction), the batch normalization layer uses the accumulated population statistics (μ and σ) instead of the mini-batch statistics. This ensures consistent behavior during inference and helps the model generalize well to new data.

Batch normalization is a powerful technique that significantly improves the training of deep neural networks. By normalizing activations and introducing the parameters μ and σ, BN addresses the internal covariate shift and provides a stable and efficient training process. The update formulas for the new parameters γ and β involve backpropagation and gradient descent, while the population statistics μ and σ are learned through the accumulation of mini-batch statistics during training.

10.5 Gradient Clipping

Gradient clipping is a technique used in deep learning to prevent exploding gradients during the training of neural networks. Exploding gradients occur when the gradients become extremely large, leading to unstable learning and difficulties in converging to an optimal solution. Gradient clipping helps stabilize the training process and ensures more reliable model updates. The idea behind it is straightforward: if the gradient surpasses a certain threshold, it is scaled down to limit its magnitude. There are two main methods for gradient clipping:

1. **Clipping by Norm:** In this approach, the overall L2 norm (Euclidean norm) of the gradient vector is computed. If the norm exceeds a predefined threshold (e.g., a maximum allowable norm), the entire gradient vector is scaled down proportionally to ensure that it stays within the limit. This means that the direction of the gradient remains the same, but its magnitude is controlled.

2. **Clipping by Value:** Here, each individual element of the gradient is examined. If any element's absolute value exceeds a specified threshold, that particular element is scaled down to fit within the range. This method is more fine-grained and can handle cases where only certain gradients are causing the instability.

10.5.1 Benefits

Gradient clipping provides several benefits:

1. **Stable Training:** By constraining the gradients, it prevents large updates that can destabilize the learning process, making training more reliable.

2. **Faster Convergence:** It allows for more significant learning rates without the risk of overshooting the optimal solution, potentially leading to faster convergence.

3. **Better Generalization:** By avoiding extreme weight updates, gradient clipping can improve the generalization performance of the model on unseen data.

Gradient clipping is commonly used in recurrent neural networks (RNNs) and their variants, as they are prone to suffer from vanishing/exploding gradients due to the recurrent nature of their architecture, and the fact that Batch Normalization is difficult to implement on them. However, it can be applied to other types of neural networks as well.

10.6 Multi-Step Training Methods

10.6.1 Transfer Learning

Transfer learning is a machine learning technique where knowledge gained from solving one task is leveraged to improve the performance of a different but related task. Instead of training a model from scratch for a specific problem, transfer learning uses pre-trained models, typically trained on large datasets for general tasks, as a starting point. By fine-tuning or reusing parts of these pre-trained models, the new model can quickly adapt and achieve better results, especially when the target task has limited labeled data. Transfer learning is widely used in various domains, such as computer vision, natural language processing, and audio processing, to expedite the development of effective and accurate machine learning models.

10.6.2 Unsupervised Pretraining

Unsupervised pretraining is a machine learning technique used to initialize the parameters of a model by training it on an unlabeled dataset. Unlike supervised learning, where models learn from labeled data (input-output pairs), unsupervised pretraining focuses on learning meaningful representations of the input data without explicit target labels. These learned representations capture underlying patterns and structures in the data, enabling the model to have a better starting point before fine-tuning on a smaller labeled dataset using supervised learning. Unsupervised pretraining has proven particularly useful in scenarios where labeled data is scarce, helping to improve the overall performance and convergence of deep learning models.

10.7 Optimizers

Momentum-Based Approaches

10.7.1 Momentum

Momentum is a technique used to accelerate the convergence of gradient-based optimization algorithms. It helps the optimizer build up velocity in directions with persistent gradients, allowing it to overcome obstacles and reach the optimal solution more efficiently. The idea behind momentum is inspired by physics, where objects in motion tend to keep moving in the same direction with a certain velocity.

10.7.2 How Momentum Works

Momentum operates by introducing a moving average of the gradients calculated during training. This moving average is treated as a "velocity" vector, and it influences the update of the model's parameters at each iteration. When the gradients consistently point in the same direction over time, the momentum term accumulates and speeds up the updates along that direction. On the other hand, when the gradients change direction frequently or fluctuate, the accumulated momentum diminishes the effect of those noisy updates. The formula for updating the model's parameters using momentum is as follows:

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot \nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha \cdot v_t$$

Where:

- v_t is the velocity (moving average) at time step t.
- β is the momentum hyperparameter (usually set between 0 and 1).
- $\nabla J(\theta_t)$ is the gradient of the loss J with respect to the model's parameters θ_t at time step t.
- α is the learning rate hyperparameter, controlling the step size of the parameter updates.

10.7.3 Hyperparameters

Momentum introduces two hyperparameters:

- **Momentum (β):** This hyperparameter controls the contribution of the previous velocity to the current velocity. A value close to 1 means that the past gradients have a stronger influence, and the velocity accumulates more information over time. A lower value dampens the effect of past gradients, giving more importance to the current gradient.
- **Learning Rate (α):** This hyperparameter determines the step size for updating the model's parameters. A larger learning rate allows for more significant updates, which can lead to faster convergence but may cause oscillations or overshooting. A smaller learning rate ensures more cautious updates but may slow down convergence.

Momentum is especially effective in escaping local minima or narrow ravines in the optimization landscape. It helps the optimizer gain inertia and move past flat regions more quickly. However, setting the momentum hyperparameter too high can cause the optimizer to overshoot the optimal solution, potentially leading to unstable training.

10.7.4 Nesterov Accelerated Gradient

Nesterov Accelerated Gradient (NAG) is a variant of the momentum optimizer, designed to improve convergence and handling of high curvature regions in the optimization landscape. It addresses some of the limitations of traditional momentum and helps accelerate the optimization process.

10.7.5 How Nesterov Accelerated Gradient Works

NAG operates by introducing a "look-ahead" term to the standard momentum update. Instead of evaluating the gradient at the current position, NAG evaluates the gradient at a point slightly ahead in the direction of the current momentum. This allows the optimizer to anticipate the upcoming position and adjust the momentum accordingly. This optimizer uses the same hyperparameters as momentum.
The formula for updating the model's parameters using Nesterov Accelerated Gradient is as follows:

$$v_t = \beta \cdot v_{t-1} - \alpha \cdot \nabla J(\theta_t + \beta \cdot v_{t-1})$$

$$\theta_{t+1} = \theta_t + v_t$$

Where:

- v_t is the velocity (moving average) at time step t.
- β is the momentum hyperparameter (usually set between 0 and 1).
- $\nabla J(\theta_t + \beta \cdot v_{t-1})$ is the gradient of the cost function J with respect to the model's parameters evaluated at the look-ahead point.
- α is the learning rate hyperparameter, controlling the step size of the parameter updates.

Nesterov Accelerated Gradient is particularly effective in handling high curvature regions and has shown improved convergence compared to standard momentum. By using the look-ahead approach, it better aligns the momentum direction with the upcoming position, reducing the oscillations that can occur with traditional momentum.

10.7.6 Adaptive Gradient Algorithm (AdaGrad)

AdaGrad (Adaptive Gradient Algorithm) is an adaptive learning rate optimization algorithm that adjusts the learning rate for each parameter in the neural network based on its historical gradient information. It is designed to effectively handle sparse data and alleviate the problem of choosing a global learning rate.

10.7.7 How AdaGrad Works

AdaGrad maintains a separate learning rate for each parameter in the model, which is inversely proportional to the cumulative sum of the squared gradients for that parameter. This means that frequently updated parameters will have a smaller learning rate, while parameters with infrequent updates will have a larger learning rate.

The formula for updating the learning rate and the model's parameters using AdaGrad is as follows:

$$g_{t,i} = g_{t-1,i} + (\nabla J(\theta_{t,i}))^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{g_{t,i} + \epsilon}} \cdot \nabla J(\theta_{t,i})$$

Where:

- $g_{t,i}$ is the cumulative sum of squared gradients for parameter i up to time step t.
- $\nabla J(\theta_{t,i})$ is the gradient of the cost function J with respect to parameter $\theta_{t,i}$ at time step t.
- α is the learning rate hyperparameter, controlling the overall step size for updates.
- ϵ is a small constant (usually around 10^{-8}) added to the denominator to prevent division by zero.

AdaGrad's adaptive learning rate approach can be beneficial in situations where the data has varying feature distributions and sparse gradients. It automatically reduces the learning rate for frequently updated parameters, preventing them from overshooting and achieving more stable convergence. However, AdaGrad's learning rate diminishes rapidly, making it less suitable for long training sessions as the learning rate may become too small. To address this, more advanced adaptive optimizers like RMSprop and Adam have been developed.

10.7.8 RMSProp

RMSProp (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm that addresses the diminishing learning rate issue of AdaGrad. It adjusts the learning rate for each parameter in the neural network based on the moving average of squared gradients, offering more stable and efficient convergence.

10.7.9 How RMSProp Works

RMSProp computes an exponentially decaying average of the squared gradients for each parameter. This moving average acts as a measure of the historical gradient information for that parameter. By using this squared gradient average, RMSProp shrinks the learning rate for parameters with large and frequent updates, while it increases the learning rate for parameters with small or infrequent updates.

The formula for updating the moving average, the learning rate, and the model's parameters using RMSProp is as follows:

$$g_{t,i} = \beta \cdot g_{t-1,i} + (1 - \beta) \cdot (\nabla J(\theta_{t,i}))^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{g_{t,i} + \epsilon}} \cdot \nabla J(\theta_{t,i})$$

Where:

- $g_{t,i}$ is the exponentially decaying average of squared gradients for parameter i up to time step t.
- β is the decay rate hyperparameter, typically set close to 1 (e.g., 0.9).
- $\nabla J(\theta_{t,i})$ is the gradient of the cost function J with respect to parameter $\theta_{t,i}$ at time step t.
- α is the learning rate hyperparameter, controlling the overall step size for updates.
- ϵ is a small constant (usually around 10^{-8}) added to the denominator to prevent division by zero.

RMSProp is widely used in training neural networks due to its ability to handle varying feature distributions and sparse gradients while mitigating the diminishing learning rate issue of AdaGrad. Its adaptive learning rate approach makes it more robust in practice and facilitates faster convergence.

10.7.10 Adam

Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm that combines the benefits of both momentum and AdaGrad. It is widely used in training neural networks due to its efficiency and robustness across various tasks.

10.7.11 How Adam Works

Adam maintains two moving average vectors: the first moment estimate (m_t) and the second moment estimate (v_t). These vectors are initialized to zero at the beginning and are updated during each iteration using the gradients of the model's parameters.

The update equations for m_t and v_t are as follows:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla J(\theta_t)$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla J(\theta_t))^2$$

Where:

- m_t is the first moment estimate (mean) of the gradients at time step t.
- v_t is the second moment estimate (uncentered variance) of the gradients at time step t.
- β_1 and β_2 are hyperparameters (usually set to 0.9 and 0.999, respectively) that control the decay rates of the moving averages.
- $\nabla J(\theta_t)$ is the gradient of the cost function J with respect to the model's parameters θ_t at time step t.

The algorithm then uses these moving average vectors to update the model's parameters as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

Where:

- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the first and second moment vectors.
- α is the learning rate hyperparameter, controlling the overall step size for updates.
- ϵ is a small constant (usually around 10^{-8}) added to the denominator to prevent division by zero.

10.7.12 Hyperparameters

Adam introduces the following hyperparameters:

- **Decay Rates (β_1 and β_2):** These hyperparameters control the decay rates of the moving averages m_t and v_t. They are typically set to 0.9 and 0.999, respectively.
- **Epsilon (ϵ):** The epsilon value is a small constant added to the denominator in the update equation to prevent division by zero. A typical value for ϵ is 10^{-8}.

Now, let's consider different scenarios based on the values of m_t and v_t:

- **Large m_t and Small v_t:** If m_t is large and v_t is small, it means that the optimizer is in a stable and low-curvature region. In this scenario, the momentum term m_t dominates over the second moment term v_t in the update equation. Since m_t is large, the model's parameters θ_t will experience a significant update due to the momentum effect. The larger updates allow the optimizer to navigate through stable, low-curvature regions more efficiently.
- **Small m_t and Large v_t:** When m_t is small and v_t is large, it indicates that the optimizer is in a region with a consistent direction but varying curvature. Here, the second moment effect dominates over the momentum effect in the update equation. As a result, the step size for updating the model's parameters θ_t is reduced, and the model's parameters will be updated by a little. This cautious update helps prevent overshooting and oscillations in regions with varying curvature.

Adam's adaptive learning rate and momentum approach make it well-suited for a wide range of optimization problems, including those with sparse gradients. It often exhibits faster convergence and better generalization compared to traditional optimization algorithms. However, like other adaptive methods, it may still require some tuning of the learning rate and other hyperparameters to achieve the best performance on specific tasks.

10.8 Learning Rate & Learning Rate Schedules

10.8.1 Adjusting the Learning Rate

Using different learning rates can adjust the pace and efficacy of training. Values that are too large may cause the model to diverge, and ones that are too small might make a model halt on a non-optimal point.

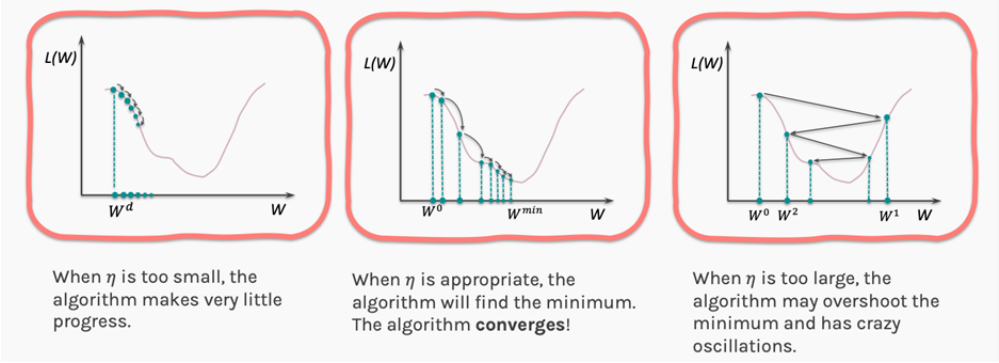


Figure 10.10: Effects of Different Learning Rates

We can also visulaize the loss function at each step of gradient descent using a trace plot.

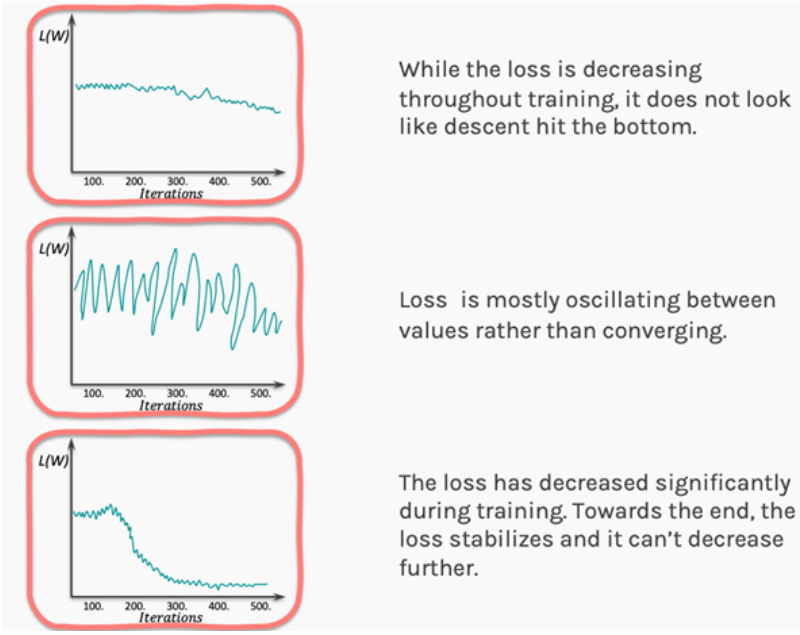


Figure 10.11: Trace Plots of Different Learning Rate Instances

10.8.2 Power Scheduling

Power scheduling, also known as polynomial decay, involves reducing the learning rate using a power function of the epoch or training step. The formula for updating the learning rate is:

$$\text{new\_learning\_rate} = \text{initial\_learning\_rate} \times \left(\frac{\text{epoch}}{\text{total\_epochs}}\right)^{\text{power}}$$

The "power" hyperparameter controls the rate of decay. A value of 1 results in linear decay, while smaller values slow down the decay, allowing the model to converge more smoothly.

10.8.3 Exponential Scheduling

Exponential scheduling reduces the learning rate exponentially over time based on a decay factor. The formula for updating the learning rate is:

$$\text{new\_learning\_rate} = \text{initial\_learning\_rate} \times \text{decay\_factor}^{\text{epoch}}$$

The "decay factor" determines the rate of decrease, and a value close to 1 leads to a slow decrease in the learning rate.

10.8.4 Piecewise Scheduling

Piecewise scheduling involves defining fixed learning rates for different epochs or training steps. The learning rate remains constant within each defined interval and changes abruptly when the next interval starts. Piecewise scheduling allows more fine-grained control over the learning rate, enabling adjustments at critical points in training.

10.8.5 Performance Scheduling

Performance scheduling adjusts the learning rate based on the model's performance on a validation set during training. If the model's performance plateaus or deteriorates, the learning rate is reduced to ensure more precise updates and potentially escape local minima. Performance scheduling helps maintain good convergence rates and adapt the learning rate to the model's learning progress.

10.8.6 1Cycle Scheduling

1Cycle scheduling is a technique that involves a cyclical learning rate. The learning rate starts at a minimum value η_0 and gradually increases to a maximum value η_1, and then decreases back to the minimum η_0 over one cycle. The maximum learning rate η_1 is chosen using the same approach used to find the optimal learning rate, and the initial learning rate η_0 is usually 10 times lower. 1Cycle scheduling has been shown to speed up training, stabilize optimization, and potentially improve the final accuracy of the model.

10.9 Regularization

10.9.1 Dropout

Dropout is a popular regularization technique used to prevent overfitting in neural networks. In a neural network, overfitting can occur when certain neurons become overly reliant on specific features in the training data. These neurons can become highly sensitive to small changes in the input data, leading to reduced generalization on unseen examples. Dropout aims to mitigate this issue by introducing noise and diversity during training.

10.9.2 The Dropout Mechanism

During each forward pass during training, Dropout stochastically *"drops out"* a fraction p (dropout rate) of the neurons in a specific layer. The dropout process randomly sets the activations of these neurons to zero, effectively removing them temporarily from the network. The remaining neurons are then scaled by a factor of $\frac{1}{1-p}$ to compensate for the reduced activations.

10.9.3 Why Scale by $\frac{1}{1-p}$?

The scaling is a crucial aspect of Dropout. The reason behind dividing the activations by $\frac{1}{1-p}$ is to maintain the expected output of the layer during training. Mathematically, for a single neuron, let x be the input and w be the weight associated with that input. During training, when a neuron is active (not dropped out), its output is scaled by $\frac{1}{1-p}$ to compensate for the dropout effect. This ensures that the expected output of the layer is the same during both training and inference (testing).

10.9.4 How Dropout Prevents Overfitting

By randomly dropping out neurons during training, Dropout introduces noise and diversity in the activations of the network. This process prevents specific neurons from becoming too specialized and ensures that different subsets of neurons are activated on each forward pass. Consequently, the network learns more robust and general features, leading to better generalization on unseen data.

10.9.5 Inference/Testing

During the inference phase (testing), Dropout is not applied, and all neurons are active. To compensate for the scaling applied during training, the weights of the neurons remain unchanged (i.e., they are not divided by $\frac{1}{1-p}$) anymore.

10.9.6 Monte Carlo Dropout

Monte Carlo Dropout is an extension of the standard Dropout regularization technique used in neural networks. Monte Carlo Dropout allows the model to estimate uncertainty in predictions, making it a valuable tool for tasks that require quantifying uncertainty, such as Bayesian deep learning and uncertainty-aware decision-making. In regular Dropout, during each forward pass during training, a fraction p of neurons in a layer are randomly dropped out, and the remaining neurons are scaled to compensate for the dropout effect. The intuition behind Monte Carlo Dropout is to extend this process of introducing noise and randomness from training to the testing phase as well. By running multiple forward passes during inference with Dropout still applied, Monte Carlo Dropout generates a distribution of predictions for each input, allowing the model to capture uncertainty in its predictions. In other words, Monte Carlo Dropout turns the Dropout mechanism into a Bayesian approximation.

10.9.7 Estimating Uncertainty with Monte Carlo Dropout

The variability in predictions obtained from multiple forward passes with Dropout is an indicator of the model's uncertainty. When the model is confident about a prediction, the predictions from different forward passes should be relatively consistent. On the other hand, when the model is uncertain or when the input is ambiguous, the predictions will be more spread out. The variance of predictions obtained from Monte Carlo Dropout can be used to quantify uncertainty in the model's predictions.

10.9.8 Max-Norm Regularization

Max Norm Regularization is a technique used in neural networks to prevent overfitting by constraining the magnitudes of weights. It limits the L2 norm of each weight vector to a maximum value c. If the norm exceeds c, the weights are rescaled as follows:

$$w \leftarrow \frac{c}{\|w\|_2} \cdot w$$

By doing this, Max Norm Regularization ensures that the weights do not grow excessively large during training, promoting stability and improved generalization. The hyperparameter c controls the strength of regularization, with smaller values providing stronger regularization effects.

Chapter 11

Time Series Analysis

Time series data consists of a sequence of data points that are indexed in chronological order. This type of data is found across various domains and applications. In business and accounting, it can include metrics like corporate quarterly earnings and ticket sales for airlines. In the realm of finance and investment, time series data encompasses variables such as stock prices, interest rates, and foreign exchange rates. Additionally, time series data is crucial in environmental sciences for studying phenomena like global warming and tracking population changes among wild animals. These time-ordered datasets provide insights into how values change over time, enabling us to analyze trends, patterns, and fluctuations within specific contexts.

11.1 Stationarity

Stationarity refers to a fundamental property of a time series where its statistical properties remain consistent over time. A stationary time series simplifies analysis and modeling, as its statistical properties remain predictable. For a time series to be considered stationary, it needs to satisfy three key criteria:

1. **Constant Mean:** The mean of the time series remains unchanged across different time periods.
2. **Constant Variance:** The variance (or standard deviation) of the time series remains stable over time.
3. **Constant Autocorrelation:** The autocorrelation, which measures the relationship between observations at different time lags, remains consistent across time (no seasonality).

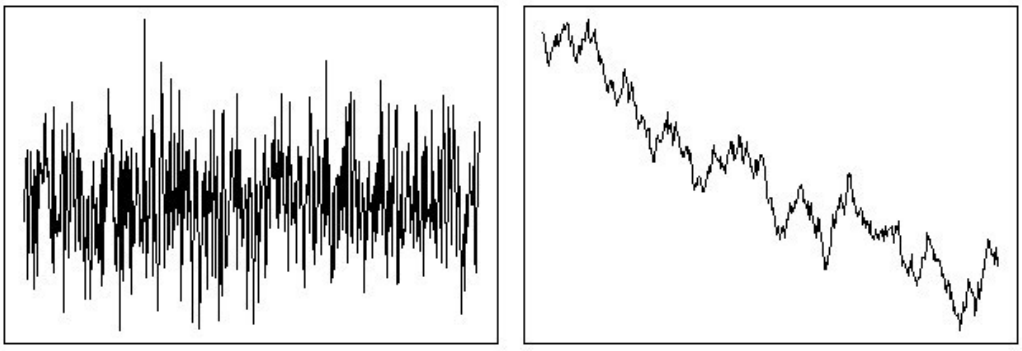


Figure 11.1: Left – Stationary Series, Right – Non-Stationary Series

11.1.1 Rolling Mean & Standard Deviation

One method to assess stationarity involves plotting the rolling mean and rolling standard deviation of the time series. The rolling mean is calculated by taking the average of data points within a sliding window, and the rolling standard deviation measures the variability within the same window. This approach uses the following formulas:

$$\mu_t = \frac{1}{k}\sum_{i=1}^{k} x_i, \text{ where } k \text{ is the window size.}$$

$$\sigma_t = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(x_i - \mu_t)^2}$$

11.1.2 Unit Roots & ADF

Another way to check for non-stationarity is to check for the existence of a unit root. To understand what a unit root is, imagine a financial time series, such as stock prices, that frequently fluctuate. A unit root suggests that the price changes tend to persist over time, and even after a significant change, the prices don't naturally revert to a consistent value. This is in contrast to a stationary series, where data points tend to cluster around a fixed mean, and any deviations from that mean are temporary. The presence of a unit root implies that past values heavily influence the current value, causing the series to exhibit a high degree of persistence. It also indicates that shocks or changes to the data have a long-lasting impact.

The Augmented Dickey-Fuller (ADF) test is a statistical method used to check for the presence of a unit root, which is indicative of non-stationarity. The ADF test involves regressing the differenced time series on its lagged values to assess whether the coefficient of the lagged values is significantly different from zero. The null hypothesis of the ADF test is that the time series has a unit root (non-stationary), while the alternative hypothesis suggests stationarity. The calculated p-value, when small, suggests stronger evidence for rejecting the null hypothesis and concluding that the series is stationary.

11.1.3 Transformations

The following tools may be used to transform a non-stationary series into a stationary one (these transformations can be undone to create model forecasts/predictions):

- **First Difference:** Subtract each data point from its previous point to remove mean trends.

$$\Delta y_t = y_t - y_{t-1}$$

- **Higher Order Differencing:** Apply consecutive differencing to address complex trends.

$$\Delta^2 y_t = \Delta y_t - \Delta y_{t-1}$$

- **Log Transformation:** Take the natural logarithm to stabilize variance in exponential data.

$$z_t = \ln(y_t)$$

11.2 Autocorrelation

Autocorrelation, also known as serial correlation, is a statistical concept that measures the correlation between a time series and its lagged versions. In other words, it quantifies the relationship between a data point and its historical values at different time lags.

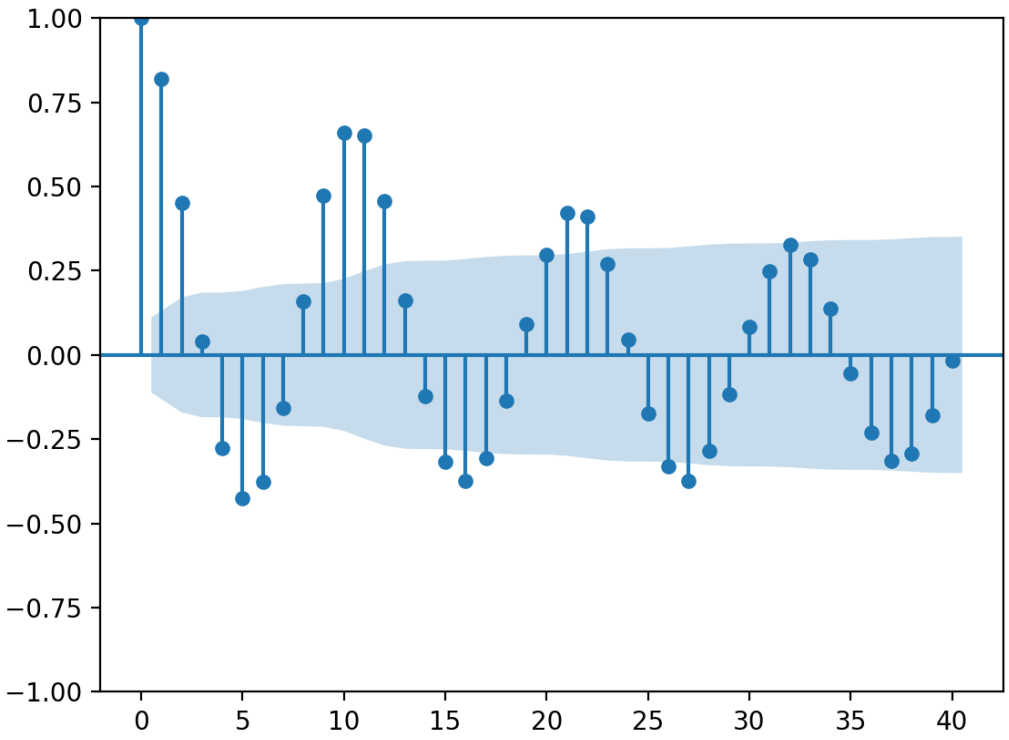


Figure 11.2: Autocorrelation Plot, with ACF/PACF Probability (Y) & Time Lag (X)

11.2.1 AutoCorrelation Function

The ACF measures the correlation between a time series and its lagged values at various time lags. It helps identify the presence of any significant correlation or pattern in the data at different lags. A positive ACF value at lag k indicates that the current value is correlated with the value at lag k, while a negative ACF value implies an inverse correlation.

11.2.2 Partial AutoCorrelation Function

The PACF, on the other hand, aims to capture the direct relationship between a data point and its lagged value while removing the influence of intermediate lags. It quantifies the correlation between the current value and its lag after accounting for the correlation contributed by previous lags. In essence, the PACF helps determine the optimal lag order for autoregressive components.

11.3 Models

11.3.1 Naive Forecasting

Naive forecasting based on seasonality is a simple time series forecasting approach that relies on the assumption that future values will be similar to the corresponding values from the previous season. In this method, the forecast for the next season is set to the observed value from the same season in the previous year (or corresponding period). While this technique doesn't account for trends, patterns, or external factors, it can be surprisingly accurate for time series with strong seasonal components.

11.3.2 AutoRegressive (AR)

An autoregressive (AR) model is a time series forecasting method that predicts a future data point based on its past values. The model assumes that the value at a particular time step is linearly dependent on its previous values, extending up to a certain number of lagged time steps. Mathematically, an autoregressive model of order p, denoted as AR(p), is defined as:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \varepsilon_t$$

Where:

- y_t is the value at time t that we want to predict.
- c is a constant term.
- $\phi_1, \phi_2, \ldots, \phi_p$ are the coefficients corresponding to the p previous time steps.
- $y_{t-1}, y_{t-2}, \ldots, y_{t-p}$ are the lagged values up to order p.
- ε_t represents the error term at time t, accounting for unpredictable variations, and acting as a regularization term.

The AR model captures autocorrelation in the data, modeling the dependency of a data point on its own previous values. Its accuracy relies on choosing an appropriate order p and estimating coefficients $\phi_1, \phi_2, \ldots, \phi_p$.

11.3.3 Moving Average (MA)

A Moving Average (MA) model is a time series forecasting method that predicts a future data point based on a linear combination of its past forecast errors. Unlike the autoregressive model, the MA model doesn't consider the previous values of the time series. Mathematically, a Moving Average model of order q, denoted as MA(q), is defined as:

$$y_t = \mu + \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2} + \ldots + \theta_q\varepsilon_{t-q}$$

Where:

- y_t is the value at time t that we want to predict.
- μ is the mean or constant term.
- ε_t represents the error term at time t, accounting for unpredictable variations.
- $\theta_1, \theta_2, \ldots, \theta_q$ are the coefficients corresponding to the q previous forecast errors.
- $\varepsilon_{t-1}, \varepsilon_{t-2}, \ldots, \varepsilon_{t-q}$ are the past forecast errors up to order q.

The MA model captures short-term dependencies between observations, focusing on the relationship between current and past forecast errors. It's useful for removing noise and irregular fluctuations from a time series, improving forecasting accuracy.

11.3.4 AutoRegressive Moving Average (ARMA)

An Autoregressive Moving Average (ARMA) model is a time series forecasting method that combines the autoregressive (AR) and moving average (MA) models. It captures both the linear relationship between past values and the short-term dependencies of forecast errors. Mathematically, an ARMA model of order (p, q), denoted as ARMA(p, q), is defined as:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2} + \ldots + \theta_q\varepsilon_{t-q}$$

Where:

- y_t is the value at time t that we want to predict.
- c is a constant term.
- $\phi_1, \phi_2, \ldots, \phi_p$ are the coefficients corresponding to the p previous time steps.
- ε_t represents the error term at time t, accounting for unpredictable variations.
- $\theta_1, \theta_2, \ldots, \theta_q$ are the coefficients corresponding to the q previous forecast errors.
- $\varepsilon_{t-1}, \varepsilon_{t-2}, \ldots, \varepsilon_{t-q}$ are the past forecast errors up to order q.

The ARMA model combines the strengths of both autoregressive and moving average models, capturing both long-term patterns and short-term dependencies in time series data.

11.3.5 AutoRegressive Integrated Moving Average (ARIMA)

An ARIMA model of order (p, d, q), denoted as ARIMA(p, d, q), extends ARMA by introducing an integration (I) step. Here's the breakdown:

- p is the order of the autoregressive component.
- d is the order of differencing required to achieve stationarity.
- q is the order of the moving average component.

The integration step involves differencing the time series by order d, resulting in a differenced series y'_t. This step removes trends and non-stationary patterns from the original time series y_t, making it suitable for modeling with the AR and MA components.

11.3.6 Seasonal AutoRegressive Integrated Moving Average (SARIMA)

The Seasonal Autoregressive Moving Average (SARMA) model extends the ARMA model to capture both temporal patterns and seasonal variations present in time series data. SARMA models are particularly useful for datasets with recurring patterns over specific time intervals. A SARMA model of order $(p, d, q) \times (P, D, Q)_s$, is expressed as a combination of autoregressive (AR), moving average (MA), seasonal autoregressive (SAR), and seasonal moving average (SMA) components:

$$\begin{aligned} y_t = &\phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} \\ &+ \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q} \\ &+ \Phi_1 y_{t-s} + \ldots + \Phi_P y_{t-Ps} \\ &+ \Theta_1 \varepsilon_{t-s} + \ldots + \Theta_Q \varepsilon_{t-Qs} + \varepsilon_t + c \end{aligned}$$

Where:

- p, d, and q are the orders of the autoregressive, differencing, and moving average components, respectively.
- P, D, and Q are the orders of the seasonal autoregressive, seasonal differencing, and seasonal moving average components, respectively.
- s is the seasonal period, indicating the length of the seasonal cycle.
- c is a constant term.
- $\phi_1, \ldots, \phi_p$ and $\theta_1, \ldots, \theta_q$ are coefficients for the AR and MA terms.
- $\Phi_1, \ldots, \Phi_P$ and $\Theta_1, \ldots, \Theta_Q$ are coefficients for the seasonal AR and MA terms.
- ε_t represents the forecast error at time t.

11.3.7 Choosing ARMA Model Family Hypeparameters

Several methods can help determine the appropriate values for p and q:

1. **Autocorrelation Plot:** When the ACF drops off significantly after a certain lag, it indicates that the correlation between the current observation and its earlier lags is not significantly influenced by the intervening lags. This suggests a potential moving average relationship of order q where q is the lag at which the ACF drops off.

2. **Partial Autocorrelation Plot:** When the PACF drops off significantly after a certain lag, it suggests that the correlation between the current observation and its earlier lags is not significantly influenced by the intervening lags. This suggests a potential autoregressive relationship of order p where p is the lag at which the PACF drops off.

3. **Information Criteria:** Metrics like AIC, BIC, and AICc quantify model fit and complexity to aid in selecting p and q. The following include the most useful information criteria in this regard:

   - **Akaike Information Criterion (AIC):** Balances model complexity and goodness of fit. Lower AIC values indicate better models.

   $$\text{AIC} = -2 \cdot \ln(L) + 2 \cdot (p + q + 1)$$

   - **Bayesian Information Criterion (BIC):** Similar to AIC but applies a stronger penalty for complexity. Lower BIC values are preferred.

   $$\text{BIC} = -2 \cdot \ln(L) + (p + q + 1) \cdot \ln(T)$$

   - **AICc (AIC with Correction):** Adjusts AIC for small sample sizes to prevent overfitting.

   $$\text{AICc} = \text{AIC} + \frac{2 \cdot (p + q + 1) \cdot (p + q + 2)}{T - p - q - 2}$$

   Where L is the likelihood value, T is the number of observations, p is the AR order, and q is the MA order.

11.3.8 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a specialized class of neural networks tailored for handling sequential and time-dependent. RNNs excel at capturing temporal dependencies in data by maintaining a hidden state that evolves over time with the processing of new inputs. This hidden state acts as a memory, retaining information from previous time steps, and so enabling the network to learn intricate patterns and relationships within the sequential data. RNNs employ different architectures like simple recurrent layers, LSTMs, and GRUs.

Recurrent Layer

A recurrent layer is a crucial component of Recurrent Neural Networks (RNNs) designed to process sequential data while retaining memory of past inputs. It is structured as follows:

1. **Input and Previous Hidden State:** At each time step t, the recurrent layer receives an input vector x_t representing the current element of the sequence. It also takes the previous hidden state h_{t-1} from the preceding time step $t-1$, encapsulating information from earlier steps.

2. **Combining Inputs:** The input vector x_t and previous hidden state h_{t-1} are combined to generate an intermediate state s_t. This amalgamation involves matrix multiplication of x_t and h_{t-1}, along with the addition of bias terms.

3. **Activation Function:** The intermediate state s_t undergoes a non-linear activation function.

4. **Hidden State Update:** The activated intermediate state s_t becomes the updated hidden state h_t for the current time step. This revised h_t now encapsulates information from both the current input x_t and previous time steps via h_{t-1}.

5. **Recurrent Connection:** Importantly, the hidden state h_t is propagated as input to the subsequent time step, $t+1$, to serve as the previous hidden state h_{t-1}. This recurrent connection allows the network to retain memory of past inputs across the sequence.

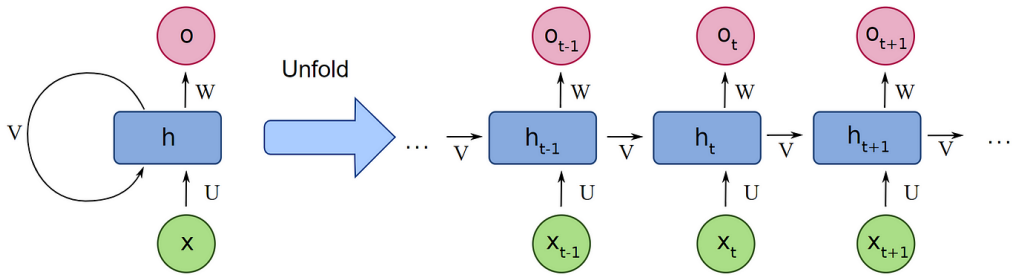


Figure 11.3: Recurrent Layer Structure

Nowadays, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are used in place of traditional recurrent layers in because they suffer from the vanishing gradient problem, which makes it difficult for the network to learn long-range dependencies in sequential data. LSTMs and GRUs alleviate this issue by incorporating gating mechanisms that allow them to selectively store and retrieve information over time. This enables them to capture long-term patterns and dependencies, making them better at modeling complex relationships within sequential data.

LSTM

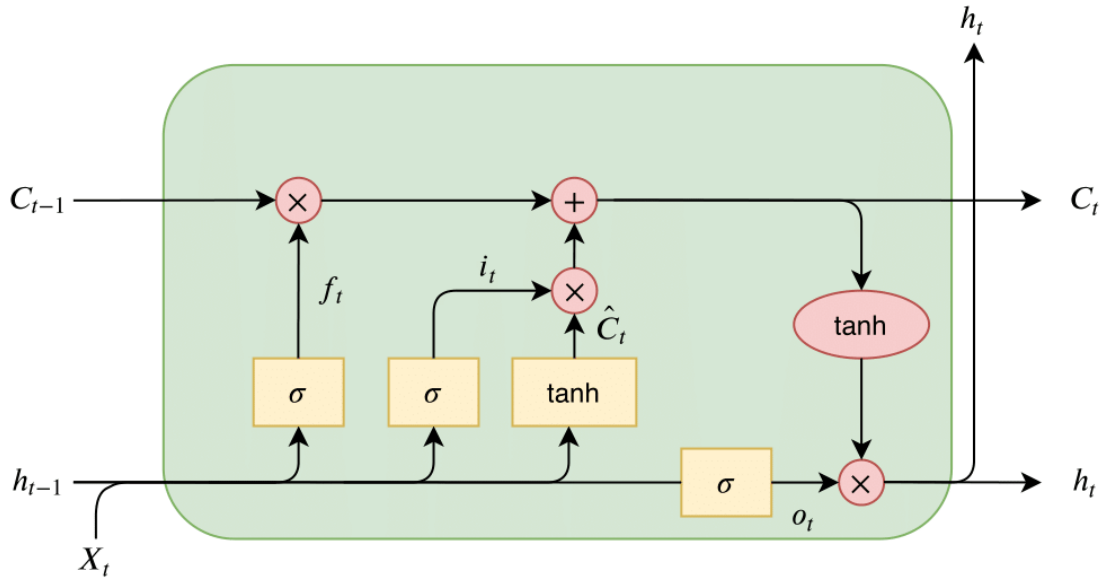


Figure 11.4: LSTM Structure

The LSTM (Long Short-Term Memory) cell comprises several essential steps that govern the flow of information and memory within the cell. It has the following components:

- **Forget Gate:** The first step in an LSTM is the *forget gate layer*. This layer decides what information to discard from the cell state. It does so by considering the previous hidden state h_{t-1} and the current input x_t. The output of the forget gate layer is a sigmoid activation applied to each value in the cell state C_{t-1}. The resulting values range between 0 and 1, where 0 signifies "completely discard" and 1 signifies "retain completely."

- **Input Gate:** The next step involves determining what new information to store in the cell state. The input gate layer employs a sigmoid activation to decide which values to update. Simultaneously, the *tanh* activation generates a vector of new candidate values C_t that could be added to the state.

- **Cell State Update:** With the input gate layer and candidate values prepared, the LSTM cell updates the cell state C_{t-1} to the new cell state C_t. This update involves a combination of the old state C_{t-1}, the forget gate values f_t, and the scaled candidate values $i_t \cdot C_t$.

- **Output Gate:** The output is derived from the new cell state C_t. The output gate layer involves two steps: first, a sigmoid activation determines which segments of the cell state to output; then, a *tanh* activation scales the cell state values between -1 and 1. The output is obtained by element-wise multiplication of the

GRU

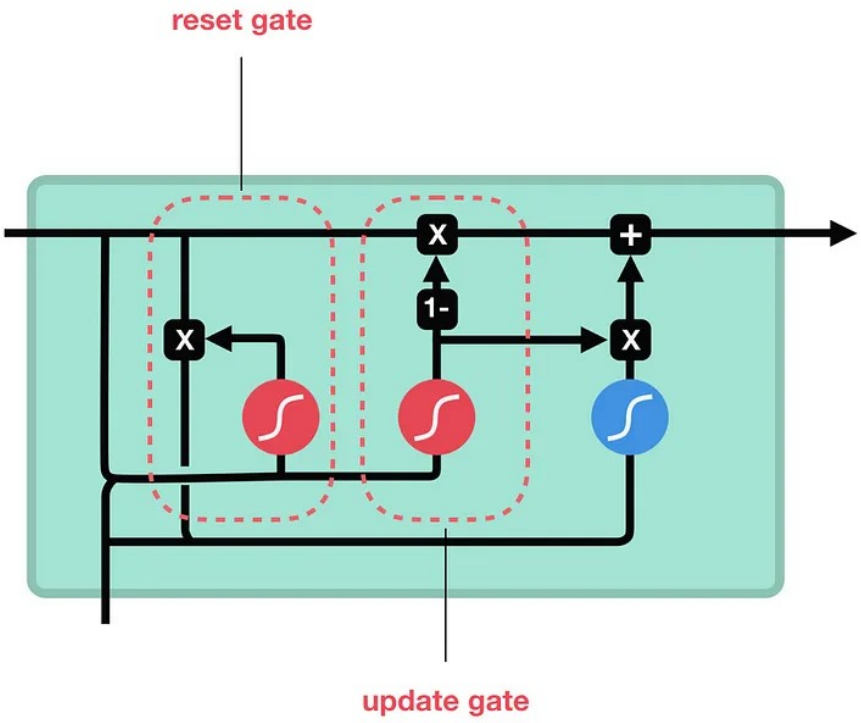


Figure 11.5: GRU Structure

The GRU (Gated Recurrent Unit) is another type of recurrent neural network cell that shares similarities with the LSTM. A GRU has the following components:

- **Reset Gate:** This gate determines what information from the previous state h_{t-1} should be ignored when calculating the new candidate state $\tilde{h}_t$. The candidate state $\tilde{h}_t$ is calculated based on the current input x_t and the previous hidden state h_{t-1}. It represents a candidate for the new hidden state h_t, with the information selected by the reset gate.
- **Update Gate:** This gate decides how much of the previous state h_{t-1} should be mixed with the new candidate state $\tilde{h}_t$. The update gate controls the weighting of these two components, allowing the model to decide how much of the past state to retain and how much of the candidate state to integrate.

11.4 MAPE Evaluation

The Mean Absolute Percentage Error (MAPE) serves as a crucial evaluation metric for assessing the accuracy of predictions within time series analysis. MAPE quantifies the average percentage difference between forecasted values and the actual observed values in a given time series dataset. The MAPE calculation formula is as follows:

$$\text{MAPE} = \frac{100}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right|$$

Where:

- n signifies the total number of observations in the time series.
- A_t represents the actual observed value at time t.
- F_t indicates the forecasted value at time t.

11.5 Windowing

Windowing is a method used to reshape data for effective training and assessment of machine learning models on sequential data. Time series information often follows a chronological sequence, where the order of observations carries significance. Windowing involves breaking down the time series into discrete, fixed-size windows. Each window constitutes a contiguous subset of consecutive time steps, creating coherent sequences that enable models to grasp patterns and facilitate predictions.

For example, a time series representing daily temperature readings needs windowing before being passed into a model. Windowing forms sequences consisting of, for instance, the temperature values of the last 7 days as input, with the temperature on the 8th day as the prediction target. By sliding the fixed-size window across the time series, numerous overlapping or non-overlapping sequences emerge.

Chapter 12

Natural Language Processing

12.1 Text Preprocessing

Text preprocessing is crucial in natural language processing (NLP) as it ensures that text data is consistent, noise-free, and compatible with NLP algorithms. By removing noise like special characters and punctuation, correcting typos, and transforming text into standardized feature representations, preprocessing reduces data complexity and dimensionality. This results in improved model performance, efficient storage, and better interpretability. Here are the most common text preprocessing steps employed:

- **Tokenization**: The process of breaking down a text or a sequence of characters into smaller units called tokens. Tokens are the basic units of text, and they can be words, subwords, or even individual characters.

  For example, "I love dogs and cats." can be tokenized as ["I", "love", "dogs", "and", "cats", "."]

  Tokenization can be done based on spaces (making each word its own token), or more abstractly. Here are two variations on how to improve the quality of tokenization:

  - **Subword Regularization:** Subword regularization introduces randomness in tokenization during training; instead of consistently splitting words into the same subword units, subword regularization involves introducing variations in the tokenization process. This variation aids the model in learning to handle diverse forms of input.
  - **Byte-Pair Encoding:** BPE begins with a vocabulary containing characters (initially treated as separate subword units). Then, iteratively,the most frequent pairs of subword units are merged into a new subword unit. This process is repeated for a specified vocabulary size or a fixed number of iterations. It enables a model to have information on word roots, helping it understand words it never saw but whose roots it is familiar with.

- **Lowercasing**: Converting text to lowercase ensures models do not differentiate between the same words in different capitalization levels, effectively reducing the size of the working vocabulary.

- **Stop Word Removal**: Removing common words like "and", "the", "is", etc., reduces noise and pushes focus on more meaningful words.

- **Text Correction & Standardization**: Some text corpuses may be in need of some text correction, including spell-checking to correct typos, special character replacement, contraction expansion, and punctuation removal for some tasks.

- **Stemming & Lemmatization**: Stemming is a text normalization technique that involves removing prefixes and suffixes from words to obtain their core form, or stem. It's a rule-based process that aims to reduce words to a common base, even if the result is not a valid word. For example:
  - Original words: jumps, jumping, jumped
  - Stemmed words: jump, jump, jump

  Lemmatization is a more linguistically sophisticated approach that maps words to their base or dictionary form, called the lemma. Unlike stemming, lemmatization considers language rules to ensure that the resulting word is valid and retains its proper meaning. For example:
  - Original words: better, best, good
  - Lemmatized words: good, best, good
- **Padding Tokens & Masking:** To make training points of uniform length for processing, shorter sequences are padded with a special token known as the padding token. Masking is then used in models to ignore padding tokens during backpropogation.

12.2 Sentiment Analysis

12.2.1 Naive Bayes Classifier

A Naive Bayes classifier is a simple machine learning algorithm which involves determining the sentiment (positive, negative, or neutral) expressed in a piece of text. After being fed a tokenized, labeled training dataset, the classifier does the following:

- The model assumes that the tokens are independent of each other given the sentiment label. This simplifying assumption allows the calculation of probabilities based on each word's occurrence without considering their relationships (and is why the model is called naive). So, for a sentence of n words:

$$P(w_1, w_2, ..., w_n|\text{ sentiment}) = P(w_1|\text{ sentiment}) \cdot P(w_2|\text{ sentiment}) \cdots P(w_n|\text{ sentiment})$$

- The following calculations are done for each sentiment, using Bayes' rule:

$$P(\text{sentiment}|w_1, w_2, ..., w_n) = \frac{P(w_1, w_2, ..., w_n|\text{ sentiment}) \cdot P(\text{sentiment})}{P(w_1, w_2, ..., w_n)}$$

  These probability values can be simply found using the word-counts for different sentiment labels.

- We will then have a probability for each sentiment, the maximum of which will be our predicted sentiment (and as a result we can actually ignore the denominator of the previous step).

12.2.2 TF-IDF

In the context of text analysis, TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical representation of words in a document corpus that captures the importance of words within individual documents while considering their significance across the entire corpus. TF, DF, and IDF are the three components of the TF-IDF formula:

1. **Term Frequency (TF):** Term Frequency represents how often a specific word appears in a document. It calculates the ratio of the frequency of a word t to the total number of words in the document. It is defined as:

$$\text{TF}(t, d) = \frac{\text{number of times word } t \text{ appears in document } d}{\text{total number of words in document } d}$$

2. **Document Frequency (DF):** Document Frequency counts the number of documents in which a word t appears. It is defined as:

$$\text{DF}(t) = \text{number of documents containing word } t$$

3. **Inverse Document Frequency (IDF):** Inverse Document Frequency quantifies the rarity of a word across the entire corpus. It is calculated as the logarithm of the ratio of the total number of documents to the document frequency of the word. The formula for IDF is:

$$\text{IDF}(t) = 1 + \log\left(\frac{\text{total number of documents} + 1}{\text{DF}(t) + 1}\right)$$

We add 1 to prevent words which appear in all documents from being given an IDF of 0. Additionally, we add a pseudo-document containing all words to our corpus, which results in adding 1 in both the numerator and denominator (and prevents division by zero issues for rare words).

The TF-IDF value of a word t in a document d is the product of its Term Frequency (TF) and Inverse Document Frequency (IDF):

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

We can use TF-IDF in a Naive Bayes Classifier. In our training data, we replace the frequency of each word in each document with its associated TF-IDF score. This, as a result, changes the definition of $P(w_1|\text{ sentiment})$ to be the following:

$$P(w_1|\text{ sentiment}) = \frac{\text{sum of TF-IDF scores for } w_1 \text{ in documents belonging to given sentiment}}{\text{sum of all TF-IDF scores belonging to given sentiment}}$$

The Naive Bayes classifier assumes that tokens are independent of each other given the class label (sentiment). While this assumption simplifies calculations, it's not entirely true. TF-IDF mitigates this limitation by providing a way to consider the relative importance of words despite the independence assumption, driving the accuracy of such a model up.

12.3 Embeddings

Word embedding is a technique used to represent words or phrases as dense, continuous-valued vectors in a high-dimensional space. These vectors capture semantic relationships between words, allowing NLP models to understand and analyze language more effectively. Word embeddings convert words from their textual form into a numerical format that machine learning algorithms can process.

12.3.1 Continuous Bag of Words (CBOW)

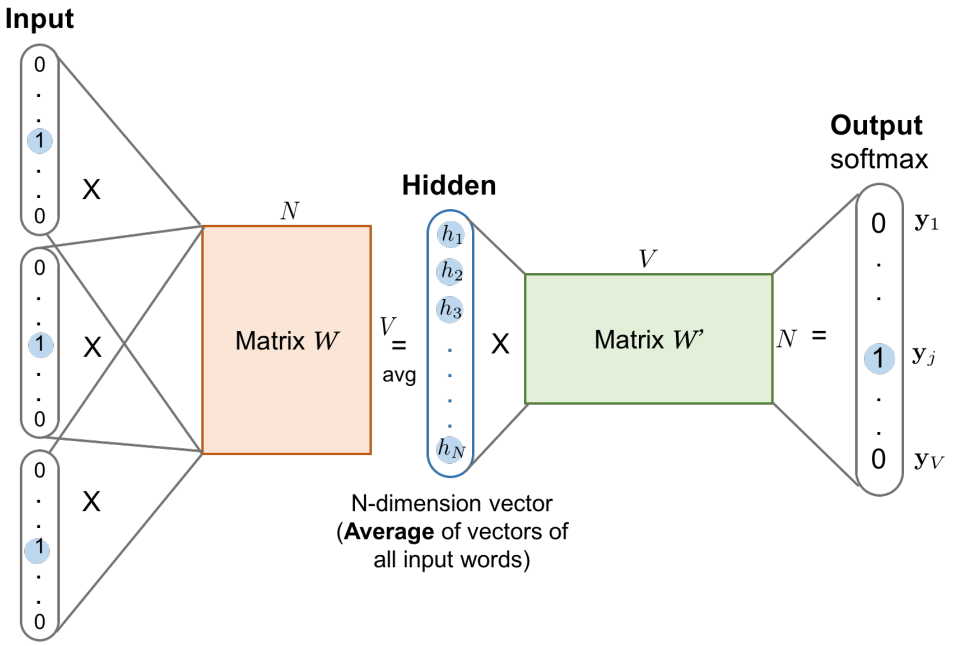


Figure 12.1: CBOW Model Structure

The Continuous Bag of Words (CBOW) model is a type of word embedding model used in natural language processing (NLP) to create distributed representations of words that capture their semantic relationships. CBOW aims to predict a target word based on the words in its preceding context. The CBOW model architecture consists of several layers:

1. **Input Layer**: The input layer receives a sequence of context words. These context words are the words that surround the target word we want to predict (training data is created using a sliding window as is the norm for language models).

2. **Embedding Layer**: The input words are transformed into their corresponding word embeddings in the embedding layer. Each word is represented by a dense vector of fixed dimensions, encoding the semantic information of the words.

3. **Hidden Layer**: The embedding vectors of the context words are averaged to create a single context representation vector.

4. **Output Layer**: The output layer predicts the target word based on the context representation from the hidden layer. Each neuron in the output layer corresponds to a word in the vocabulary. A softmax activation function converts the output values into probabilities.

Training

The CBOW model is trained on a dataset containing pairs of context words and their corresponding target words. During training, the model adjusts its weights (embedding vectors) to minimize the difference between predicted probabilities and the actual target word's representation. After training, the embedding vectors in the embedding layer are retained. Additionally, the model itself can be used for text-prediction algorithms!

12.3.2 Skip-Gram

The main-stream way to create word embeddings is through a look-up table structured neural network. The training goal is, given 2 words, to determine the probability of co-occurrence (with words co-occurring having a target of 1 and 0 otherwise. The network functions as follows:

Data Preparation

To create a suitable dataset, we take a large training corpus and slide a window of arbitrary size over its entirety to create our training data. Traditionally, the word is chosen from the middle of each windowed fragment, and all other words in that window are considered target words. As such, from a text-bit of 9 words, we create 8 data points: the centroid word as the first column in the predictor data, and the other 8 words as the second column, with the target value being 1, signaling that the words do co-occur.

Since this model could result in a model that always outputs 1 since our training data only has positive samples, we need to introduce negative sampling into the mix. Instead of computing the softmax over the entire vocabulary, negative sampling focuses on a small subset of words (negative samples) during each training step. The model learns to distinguish positive target words from these randomly chosen negatives. So for every positive sample, we add a few negative samples pairing random words that generally do not co-occur.

Embedding & Content Matrices

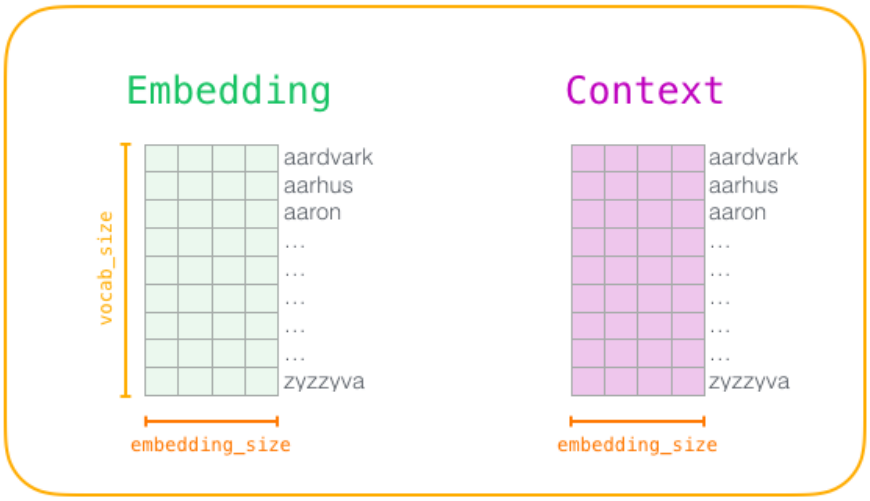


Figure 12.2: Embedding & Context Matrices

The embedding matrix contains rows of vectors representing words. Each vector encodes the semantic meaning of a word. During training, the model selects the vector of a word from the first column of our training data.

The associated context words are then looked-up in the context matrix. This table, once multipled with embeddings, transforms the embedding of a target word into a context representation. The resulting

context representation aims to capture the relationships between the target word and its context words.

Training & Backpropagation

During training, pairs of words (e.g., target word and context words) are provided as input, and their embedding/context multiplication values are passed through the sigmoid function. These values are then compared with the co-occurrence targets (0 or 1), and embeddings are updated based on prediction errors. This iterative process occurs over multiple training iterations on a text corpus. As training progresses, embeddings converge to values that encode semantic properties, allowing similar words to have similar vectors.

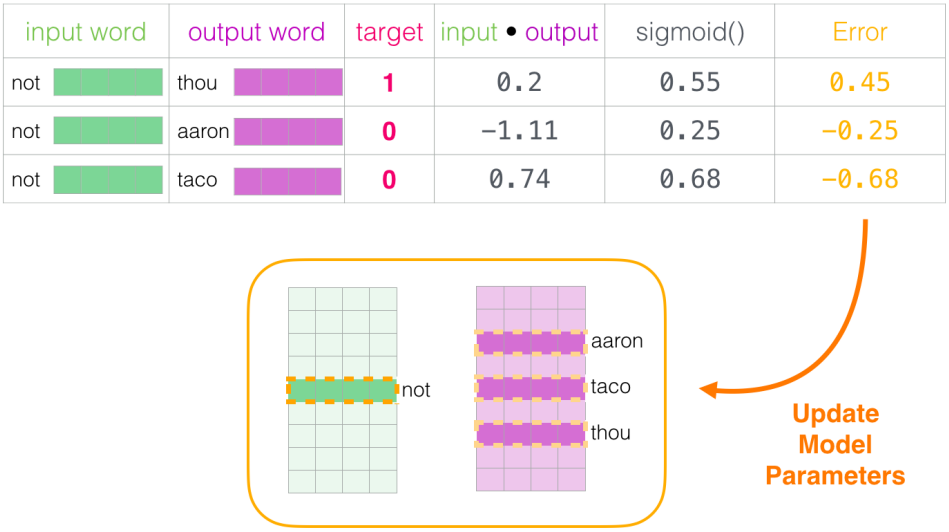


Figure 12.3: Step Update

Post-Processing

After training, we expect the embedding matrix to have learned properties of the words in our vocabulary, including co-occurrence and linguistic similarity. As a result, we can discard the context matrix and only keep our final embeddings.

One rule of thumb is that using smaller window sizes (between 2 and 15) results in embeddings where a high similarity score between two embeddings signifies that the words can be swapped without altering the context significantly (complements or substitutes). On the other hand, employing larger window sizes (ranging from 15 to 50 or more) generates embeddings where similarity is more indicative of the relationship or association between words.

12.3.3 ELMo

Character Component

The Char-CNN (Character Convolutional Neural Network) model is employed in ELMo to produce character-level embeddings for words. These embeddings capture subword information, which is valuable for handling out-of-vocabulary words and understanding word morphology. Here's a breakdown of the Char-CNN model:

1. **Architecture**: The Char-CNN model is a CNN-like model tailored to process word characters and yield a concise character-level representation. Its operation is akin to traditional CNNs that process images, except it works on word characters as input.

2. **Input**: For every word, its characters are first converted into character embeddings. These embeddings can be either initialized randomly or pre-trained (similar to word embeddings). Each character embedding serves as a channel in the CNN.

3. **Convolutional Layers**: The Char-CNN model applies multiple convolutional layers to the character embeddings. Each convolutional layer consists of several filters, which traverse the character embeddings, performing element-wise multiplications followed by summation to generate a feature map. These filters capture diverse n-gram patterns of characters (e.g. unigrams, bigrams, trigrams) to capture varying character-level details.

4. **Max-Pooling**: After convolutions, max-pooling is applied to the feature maps generated by each filter. This operation selects the maximum value within a window of values, effectively extracting the most prominent features from each feature map.

5. **Concatenation and Flattening**: The max-pooled values from all filters are concatenated into a single vector. This vector constitutes the character-level embedding for the word. The concatenated vector is then flattened to be fed into subsequent layers of the ELMo architecture.

High Way Network

A highway network is a type of neural network layer that combines the input with a transformation gate and a carry gate. These gates allow the network to determine whether the input should be transformed or passed through unchanged. The transformation gate and carry gate are both functions of the input, and they control the transformation process based on learned weights.

In the context of ELMo, the character-level embeddings produced by the Char-CNN model are fed into a highway network. This allows the model to decide which parts of the character-level embeddings should be modified and which should remain unchanged, based on the task's requirements. The transformation gate determines the extent to which the input should be transformed, while the carry gate controls the information flow from the input to the output without modification.

The transformation gate and carry gate are computed based on the input to the highway network. These gates are often sigmoid functions, usually claculated as follows:

$$T = \sigma(W_T \cdot x + b_T)$$
$$C = 1 - T$$

Here, W_T and b_T are learned weight parameters, and σ is the sigmoid activation function.

Contextual Component

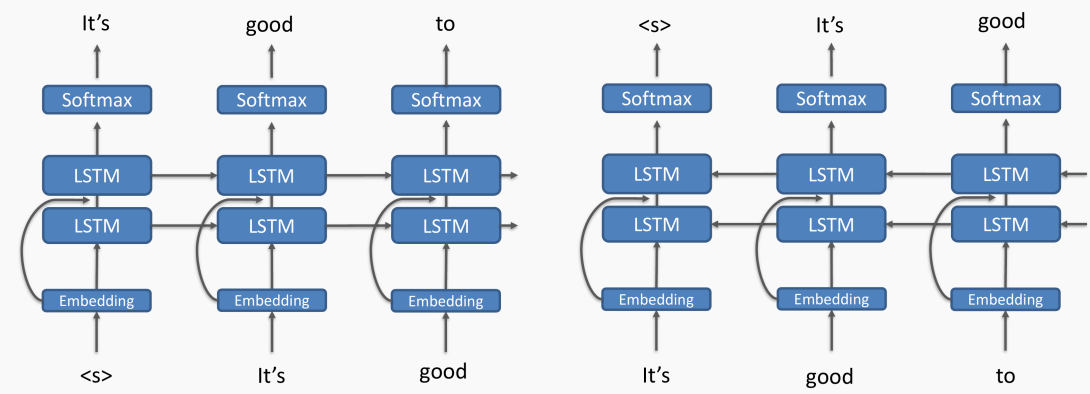


Figure 12.4: Bi-Directional LSTM ELMo Component

The contextual encoding LSTM captures the contextual information of each word by considering the surrounding words in both the forward and backward directions. Here's a breakdown of how it operates:

1. **Input**: The input to the model is the character-level embedding from the Char-CNN.

2. **Bidirectional Nature**: The LSTM used for contextual encoding is bidirectional, meaning it processes the input sequence in both the forward and backward directions simultaneously. This bidirectional processing captures the complete context of each word, as it considers both preceding and following words.

3. **Forward Pass**: The forward LSTM processes the input sequence from left to right. At each time step, the LSTM takes the current input and the hidden state from the previous time step to compute the current hidden state. This hidden state encodes information from the preceding words.

4. **Backward Pass**: Concurrently, the backward LSTM processes the same input sequence in reverse order, from right to left. Similar to the forward pass, it calculates the hidden state at each time step using the current input and the hidden state from the previous time step.

5. **Hidden State Concatenation**: At each time step, the hidden states from both the forward and backward LSTMs are concatenated. This concatenated hidden state effectively encodes the context of the current word concerning its preceding and following words in the sentence.

6. **Layer Stacking**: ELMo employs a stack of two bidirectional LSTM layers for contextual encoding. The output hidden states from the first layer are used as inputs for the second layer. This layer stacking enables the model to capture increasingly complex contextual relationships, with the lower layer capturing surface-level information and the higher layer capturing deeper semantic nuances.

7. **Residual Connections**: The output hidden states from each LSTM layer are added to the concatenated input representation using residual connections. This addition mitigates the vanishing gradient problem and facilitates the flow of information from the original input through the LSTM layers. The residual connections ensure that contextualized representations build upon existing information, enhancing the model's ability to capture context.

Training

The training objective of the ELMo model is to output porbabilities for the following words in our input sentence fragments. As infromation is passed, weights are learned, and then everything is frozen. The architecture is then used to find the character-based and context-based embeddings of each word (given the results of the Char-CNN and the hidden states of the BiLSTM portion).

In embedding applications, training weights are placed on the character-based and context-based embedding portions, and then these embeddings are concatenated to any other embedding system. This allows models to both retain root-based and context-based meanings of words, while also deciding which to ignore (and by how much).

12.4 Sequence-to-Vector Modeling

In the NLP model pipeline, model building depends on the type of task the model will be deployed for. In most cases, like text-prediction, the task is predicting the next word in the a particular expression. This type of modeling is called Sequence-to-Vector because that's exactly what it does: takes in a sequence and outputs a word. Here is how most Sequence-to-Vector models are built:

1. **Data Preprocessing**
2. **Vocabulary Building:** Create a vocabulary containing unique tokens from the training data.
3. **Embedding**
4. **Feature Extraction:** Pass embedded tokens through an RNN (LSTM, GRU) to capture sequential context, converting the series of embeddings (embedded expression) into a new aggregated form.
5. **Prediction:** The RNN architecture usually employs a softmax output layer, meaning the result of the model will be a vector of probabilities, with the highest probability word being the output of the model (next predicted word).

12.5 Sequence-to-Sequence Modeling

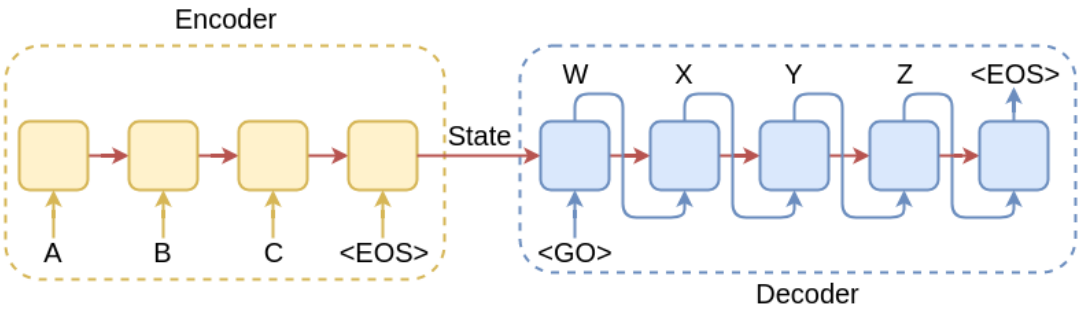


Figure 12.5: Encoder-Decoder Architecture (Inference Phase)

1. **Data Preprocessing:** In addition to regular data pre-processing steps, start and stop tokens are added to the target expressions.
2. **Vocabulary Building:** Create separate vocabularies for source and target sequences containing unique tokens.
3. **Embedding**
4. **Encoder:** Pass embedded source tokens through an encoder model (usually some form of RNN).
5. **Decoder:** Initializing the decoder's hidden state or context vector using the encoder's final state, pass through the decoder, trying to predict each word of the target sequence.
6. **Prediction:** Pass a source sequence to the encoder, then pass the final hidden state and the start token as the initial hidden state and input to the decoder, respectively. Then take the next-word predictions of the decoder as the next iteration's input. This is repeated until the model predicts a stop token, indicating the end of the predicted sequence.

12.5.1 Attention

The attention mechanism is a key component in sequence-to-sequence models. Its role is to enhance relationships within input sequences, focusing on specific elements as required. The idea is that importance weights are assigned to different input elements' hidden states to provide contextual relevance and capture long-range dependencies. The attention mechanism functions as follows (when coupled with a regular sequence-to-sequence model):

1. During the encoding operation, the hidden states are saved to be retrieved and used later.
2. A fully-connected neural network is then used, taking as input the various hidden states of the encoder model, one at a time, and the first (input) hidden state that is going to be fed into the decoder. This neural network outputs attention scores for each pair of encoder-decoder hidden state.
3. A softmax layer is used, such that for all encoder hidden states, the attention scores are converted to tangible weights between 0 and 1, indicating how much attention should be given to each encoder hidden state.
4. Based on the weights outputed, a context vector is created by summing the weighted hidden states of the encoder, and this context vector is then concatenated with the decoder input (initially a start token) to create a new context-rich input.
5. The decoder, now with a hidden state, context vector, and input, outputs a prediction as well as a new hidden state.
6. The new hidden state is used to repeat steps 2-4 to create a new context/attention vector, and so on.

12.6 Beam Search

Beam search is a search algorithm commonly used in tasks like machine translation, text generation, and other sequence-to-sequence tasks. Its purpose is to find the most likely sequence of tokens given a model or language model. It works as follows:

1. **Top-k Candidates:** Beam search maintains a set of top-k candidates or hypotheses at each step of sequence generation. These candidates represent potential sequences the model could generate.
2. **Initialization:** At the start of sequence generation, the model receives only the start token as input. It generates probabilities for the next token.
3. **Expanding Hypotheses:** For each top-k candidate, the model generates probabilities for the next token(s) based on the current sequence. This creates an expanded set of candidates.
4. **Selecting Top Candidates:** Among the expanded candidates, the top-k candidates with the highest multiplicative probabilities proceed to the next step. Other candidates are pruned.
5. **Termination:** The process continues until all candidates reach an end token or the sequence length limit. The candidate with the highest overall probability becomes the final output sequence. This process aids in avoiding local optima and exploring different potential paths.

12.7 Transformers

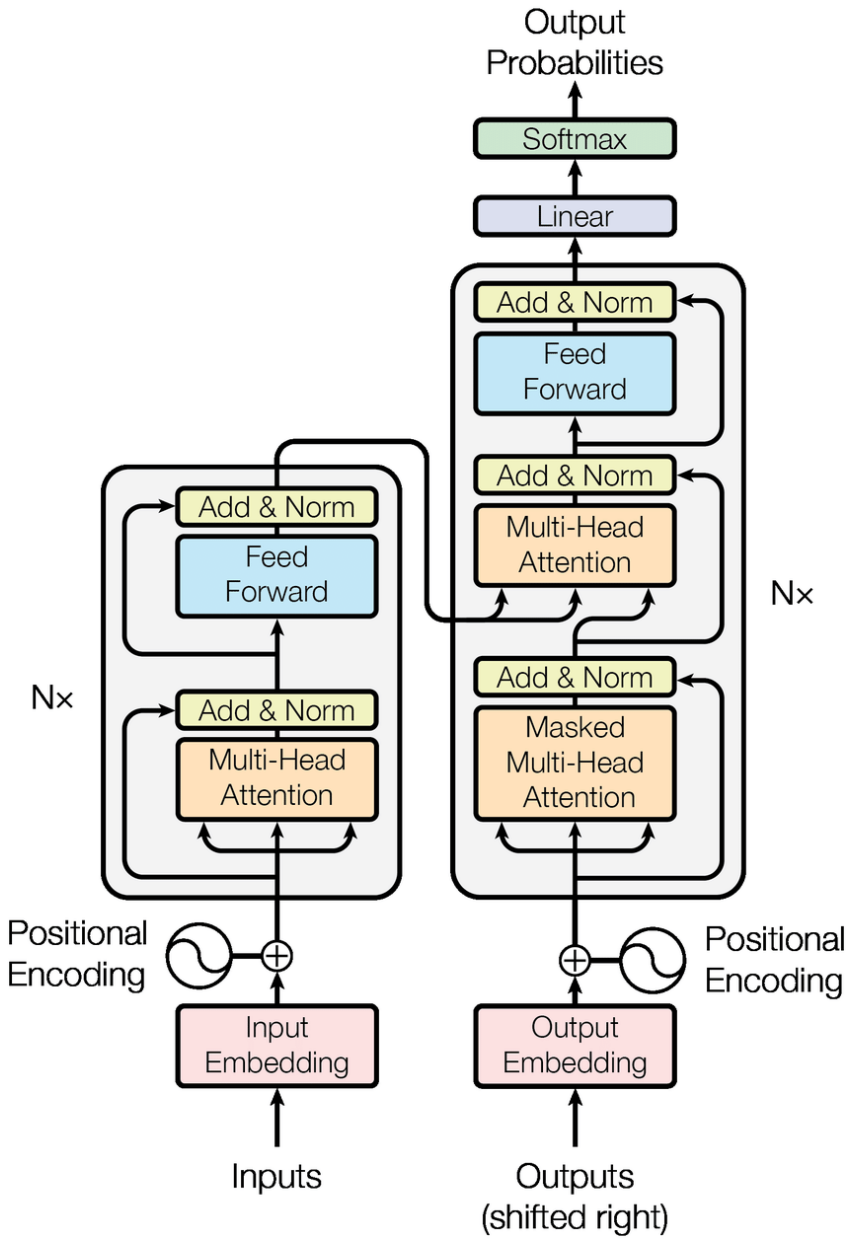


Figure 12.6: Transformer Architecture

The transformer is a revolutionary architecture which, quite frankly, is very confusing to understand. To understand how it operates, it is useful to go through each of its steps, one by one:

12.7.1 Embedding

For transformers, the same embeddings can be used as different language models, like Word2Vec and the such. These embeddings are then combined with positional encodings, to create a position-aware context encoding for each token in our corpus.

12.7.2 Positional Encoding

Positional encoding is a critical component of the Transformer architecture, providing information about the order or position of tokens in a sequence. It compensates for the lack of inherent order information in Transformers by incorporating positional information using sine and cosine functions. The formula for sine/cosine based positional embeddings are as follows:

For each each token position t and each dimension i of the embedding, the positional encoding is calculated as follows:

$$\mathrm{PE}_{(t,2i)} = \sin\left(\frac{t}{10,000^{2i/d_{\text{model}}}}\right), \quad \mathrm{PE}_{(t,2i+1)} = \cos\left(\frac{t}{10,000^{2i/d_{\text{model}}}}\right)$$

where t is the position, i is the dimension, and d_{model} is the dimensionality of the model's embedding. The exponent in the formula determines the frequency of the sine and cosine functions for each position and dimension. Higher dimensions capture longer-range relationships.

Positional encodings are simply added to the base encoding to create a encoding that is both semantically and positionally aware.

12.7.3 Self-Attention Mechanism

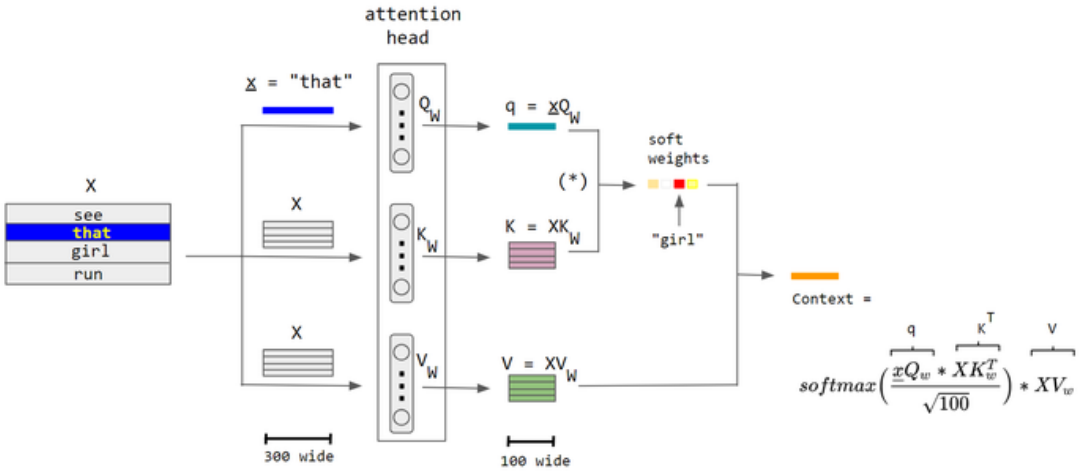


Figure 12.7: Self-Attention Architecture

A self-attention unit computes the attention relationship between every position in a sequence of input elements (words in a sentence) to capture contextual dependencies, and as a result create a contextual embedding. The mechanism has the following components:

1. **Queries (Q), Keys (K), and Values (V):** Each input element generates three representations: a query, a key, and a value. These representations are learned during training.

2. **Processing:** A set of learnable weights Q_w are applied to the queries (word embeddings) via a multiplicative step. The same is done to the keys (the same word embeddings) but now with a different set of learnable weights K_w. These produce convoluted representtions of the original embeddings, which are then transformed using cosine similarity and the softmax function to generate attention scores.

3. **Attention Scores:** The attention score measures the similarity between the query of one position and the key of another position (a pair of words within an expression). It quantifies how much focus one position should give to another.

4. **Value Scaling:** Separately, we train another set of learnable weights V_w, which linearly scale the values (the same input word embeddings). The attention weights, derived from the Query-Key relationships, determine how much each value (V) should contribute to the final context-aware representation of the token. The weighted sum of the values, with attention weights as coefficients, generates the context-aware representation.

12.7.4 Multi-Head Attention

While standard self-attention is powerful, it may not capture all nuances of complex relationships in the data. Multi-head attention addresses this limitation by performing self-attention multiple times in parallel, each with different sets of learned parameters. Instead of a single set of queries, keys, and values, multi-head attention involves splitting them into multiple sets for each attention head. In each head, the self-attention mechanism computes attention scores, attention weights, and context vectors independently. This parallel computation allows the model to focus on different aspects simultaneously.

12.7.5 Encoder

The encoder portion of the transformer goes through multiple steps, each outlined above: embedding, positional embedding, and a multi-head attention mechanism. After this point, the various contextual embeddings for each token are passed to a dense layer, and this layer, along with normalization and skip connections added to counter the issue of vanishing gradients, create one hidden state that thoroughly addresses the contextual, semantic, and positional aspects of each token.

12.7.6 Decoder

The decoder has a very similar start to the encoder: it takes in the desired output, finds the associated token embeddings, then finds positional embeddings and adds them on to the base embeddings. It then also includes a multi-head attention layer, though in this case a masked one. A masked multi-head attention layer is similar to its regular counterpart, with the amendment that a mask is applied to the attention scores of tokens and their successor tokens, which is done to prevent tokens from attending to positions that come after them in the sequence.

Although normalization layers and skip connections are the same for both decoder and encoder, the decoder has a second main difference: the encoder-decoder attention mechanism. In encoder-decoder attention, the queries are generated by the decoder, while the keys and values are derived from the encoder's output. This setup enables the decoder to focus on relevant parts of the source sequence. Similar to the regular multi-head attention mechanism, the encoder-decoder attention generates a weighted sum of the encoder's values using the attention weights. This weighted sum, encapsulates relevant information from the source sequence based on the decoder's queries.

Finally, the only other addition to the decoder that sets it apart from the encoder are the linear and softmax layers applied at the end, which create output token probabilities for the entire used vocabulary.

12.7.7 Advantages

Transformers have emerged as a significant advancement in NLP and sequence-based tasks due to their unique architecture. Here are the reasons why transformers are often considered superior to their RNN model counterparts:

- **Long-Range Dependencies**: Transformers capture long-range dependencies effectively due to the self-attention mechanism. Unlike RNNs that struggle with vanishing gradients, transformers can consider relationships across the entire input sequence.
- **Parallelization**: The self-attention mechanism enables parallel processing of tokens, leading to faster training and inference times. This contrasts with sequential RNN processing.

12.8 Perplexity

Perplexity is a commonly used evaluation metric used to assess the quality of probabilistic language models. It quantifies how perplexed a model is against an unseen textual example. Lower perplexity values indicate better model performance in predicting the next word in a sequence.

Perplexity is derived from the concept of information theory and can be defined as follows: for a language model M, given a sequence of words $w_1, w_2, \ldots, w_n$, the perplexity (PP) is calculated as:

$$\text{PP}(w_1, w_2, \ldots, w_n) = P(w_1, w_2, \ldots, w_n)^{-\frac{1}{n}}$$

Where:

- n is the number of words in the sequence.
- $P(w_1, w_2, \ldots, w_n)$ is the probability assigned by the model M to the sequence.

During the training of language models, the model learns the probabilities of word sequences based on the training data. The goal is to minimize perplexity, which means the model becomes better at predicting the training data. After training, perplexity is calculated on unseen validation or test data. A lower perplexity value indicates that the model assigns higher probabilities to the true sequences in the validation or test set, suggesting better predictive performance.

As a benchmark, we always know the perplexity measures produced by a random language model. Let n be the number of words in the sentence and V be the vocabulary size (number of unique words). In a random language model, the probability of each word in the vocabulary is $1/V$. Therefore, the probability of the entire sentence composed of n words is $\left(\frac{1}{V}\right)^n$. Using this probability in the perplexity formula:

$$\text{Perplexity} = \left(\left(\frac{1}{V}\right)^n \right)^{-\frac{1}{n}} = V$$

In other words, the perplexity of any sentence in a random language model is equal to the vocabulary size V, which is our benchmark for other *trained* LMs.

Chapter 13

Generative Modeling

13.1 Autoencoders

13.1.1 Architecture

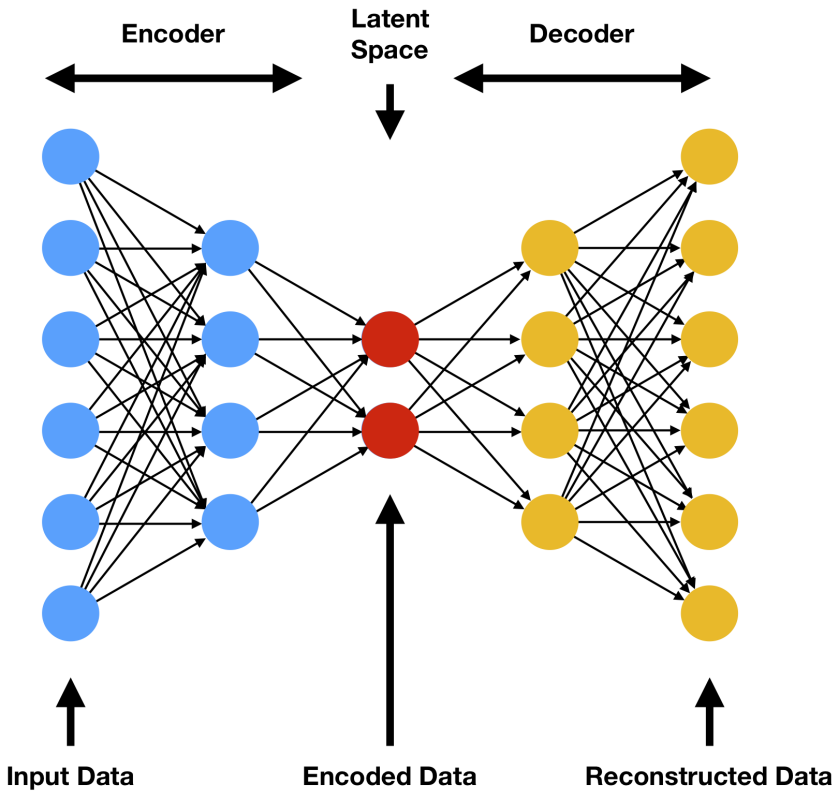


Figure 13.1: Autoencoder Architecture

An autoencoder is a type of ANN architecture primarily used for unsupervised learning tasks such as dimensionality reduction and feature learning. It consists of an encoder and a decoder, both of which are neural networks. The goal of an autoencoder is to learn a compact representation of input data in an encoded form and then reconstruct the original data as closely as possible from this encoded representation. Here's a breakdown of the architecture of an autoencoder:

1. **Encoder:** The encoder network takes in the input data and progressively reduces its dimensionality, extracting essential features in the process. The final layer of the encoder produces the encoded representation, which is a compressed version of the input data. The encoder typically consists of one or more hidden layers with decreasing neuron counts, which helps capture hierarchical features.

2. **Encoded Representation:** This is the output of the encoder and is a lower-dimensional representation of the input data. It is essentially a compressed version of the original data, capturing its essential features.

3. **Decoder:** The decoder network takes the encoded representation and aims to reconstruct the original data from it. Similar to the encoder, the decoder usually consists of one or more hidden layers with increasing neuron counts. The final layer of the decoder outputs data in the same format as the input data.

4. **Reconstructed Output:** This is the output of the decoder, which is an attempt to recreate the original input data based on the encoded representation. The autoencoder is trained to minimize the difference between the reconstructed output and the actual input data.

The training of an autoencoder involves minimizing a loss function that measures the difference between the input data and the reconstructed output. Common loss functions include mean squared error (MSE) or binary cross-entropy, depending on the nature of the data (continuous or binary).

13.1.2 Uses

Dimensionality Reduction & Visualization

Autoencoders learn to encode high-dimensional input data into a lower-dimensional representation (encoded space) and then decode it back to the original space. During training, the encoder learns to extract the most relevant features from the data, and the decoder learns to reconstruct the data from the encoded representation. This is exactly like PCA!

We can use the encoded representation of data for visualization purposes, to see relationships between high-dimensional datasets represented in lower dimensions.

Unsupervised Pretraining

By training an autoencoder on a dataset without labels, the encoder component learns to create a compact representation of the input data in a lower-dimensional space. This learned representation can capture inherent structures and essential characteristics present in the data. After training the autoencoder, the encoder part can be detached and repurposed as a feature extractor. This encoder has essentially learned to encode relevant information from the data into a compressed format.

When you have a specific supervised task to perform (classification or regression) but limited labeled data for that task, you can utilize the pre-trained encoder's learned features. Instead of initializing the supervised task's neural network from scratch, you start with the encoder's weights. This initialization helps the network to begin with a set of already informative features rather than random weights. With this initialized network, you then fine-tune it on the task-specific labeled data. Fine-tuning adjusts the network's weights to align with the task requirements.

The main benefits of this approach are twofold: it leverages the learned features from unsupervised pretraining, which can be especially valuable when labeled data is limited, and it provides a regularization effect, potentially preventing overfitting on the smaller labeled dataset.

13.1.3 Variations

Weight Tying

Weight tying is a design choice where the weights of certain layers in the encoder are constrained to be the transpose of the weights in the corresponding layers of the decoder. This means that the weights used to encode data into the latent space are also used to decode it back to the original space. This imposes a symmetry between encoding and decoding, which can have a number of benefits:

1. **Reduced Parameter Count:** Weight tying reduces the number of learnable parameters in the model, making it more memory-efficient and potentially reducing the risk of overfitting, especially when dealing with limited data.

2. **Stronger Regularization:** The smaller number of parameters acts as a form of regularization, guiding the model to learn more stable, meaningful features. This can prevent the autoencoder from simply memorizing the training data.

Convolutional Autoencoders

A convolutional autoencoder combines the principles of a regular autoencoder with the power of convolutional layers, creating an architecture tailored for processing image or grid-like data. A convolutional autoencoder takes advantage of the spatial relationships present in such data, which are crucial for tasks like image recognition, denoising, and generation.

In the encoding phase, convolutional layers are used to analyze the input image's local patterns, edges, and textures. These layers progressively reduce the spatial dimensions while retaining important features, resulting in a compressed representation of the input. During decoding, the convolutional autoencoder employs transposed convolutional layers (sometimes called deconvolutional layers) to reconstruct the original image from the latent space.

Denoising Autoencoders

Denoising autoencoders are a variation of autoencoders designed to reconstruct clean data from noisy or corrupted input data. Denoising autoencoders take noisy input data and aim to reconstruct the original, clean data from it. During training, the model is exposed to pairs of noisy data and corresponding clean data. The autoencoder learns to capture meaningful features and patterns in the noisy data and uses these features to produce more accurate, denoised reconstructions. During inference, then, it is supplied noisy data for which we might not have clean versions, to improve readability or quality and reduce noise.

Sparse Autoencoders

Sparse autoencoders are a variation of autoencoders that impose a constraint on the activations of the hidden layer, promoting the activation of only a few neurons at a time. Sparse autoencoders focus on learning representations where only a small subset of neurons in the hidden layer are active for a given input. This encourages the model to capture the most salient and distinctive features of the data while ignoring less relevant information.

Variational Autoencoders

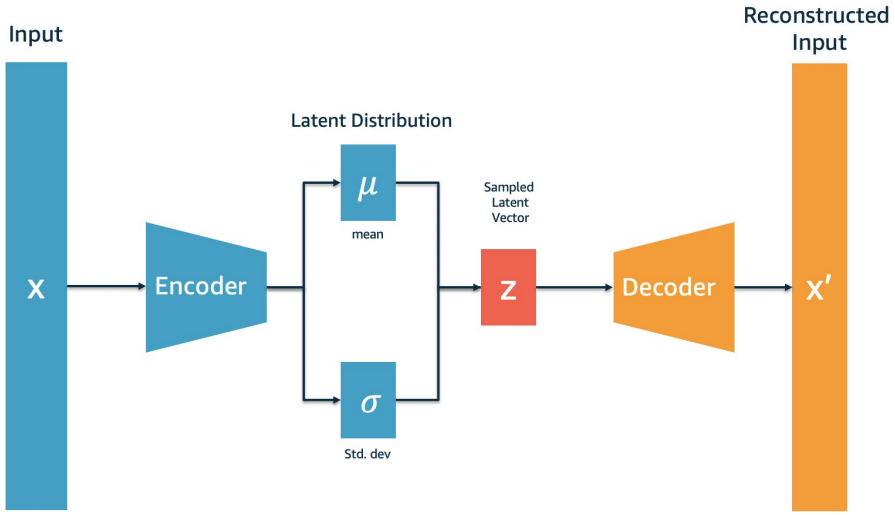


Figure 13.2: Variational Autoencoder Architecture

Variational autoencoders are a sophisticated type of traditional autoencoders that add a probabilistic twist by incorporating techniques from Bayesian inference. Variational Autoencoders (VAEs) represent a sophisticated class of neural network architectures that seamlessly blend the capabilities of autoencoders with probabilistic modeling techniques.

Analogous to conventional autoencoders, VAEs use the encoder to map input data, such as images, onto a probabilistic distribution within the latent space. This distribution is often modeled as a Gaussian distribution. The counterpart to the encoder, the decoder, uses samples extracted from the latent space to reconstruct the original data.

The encoder computes two critical parameters: the mean (μ) and variance (σ^2) of the latent distribution. A pivotal step in this process involves the stochastic sampling of the latent variable z, accomplished through $z = \mu + \epsilon \cdot \sigma$, where ϵ derives from a standard normal distribution. The latent variable z, once sampled, serves as an input for the decoder, which then generates a reconstructed version of the original data. The reconstruction loss serves as the guiding force prompting the decoder to produce accurate and faithful reconstructions.

Besides the reconstruction loss and to ensure the coherence of the latent space distribution and its proximity to a standard normal distribution, the VAE integrates the KL divergence loss. This loss, assessing the disparity between the learned latent distribution and the theoretical standard normal distribution, effectively steers the encoder towards producing latent distributions that closely align with the desired distribution, fostering structure and interpretability.

Post-training, the VAE transitions into a generative mode, capably creating novel data samples. This process involves initially sampling from the standard normal distribution in the latent space, which is then channeled through the decoder to produce synthetic data points. Through this mechanism, the VAE fosters the creation of diverse and original data samples by exploring distinct regions of the latent space.

13.2 General Adversarial Networks

13.2.1 Architecture

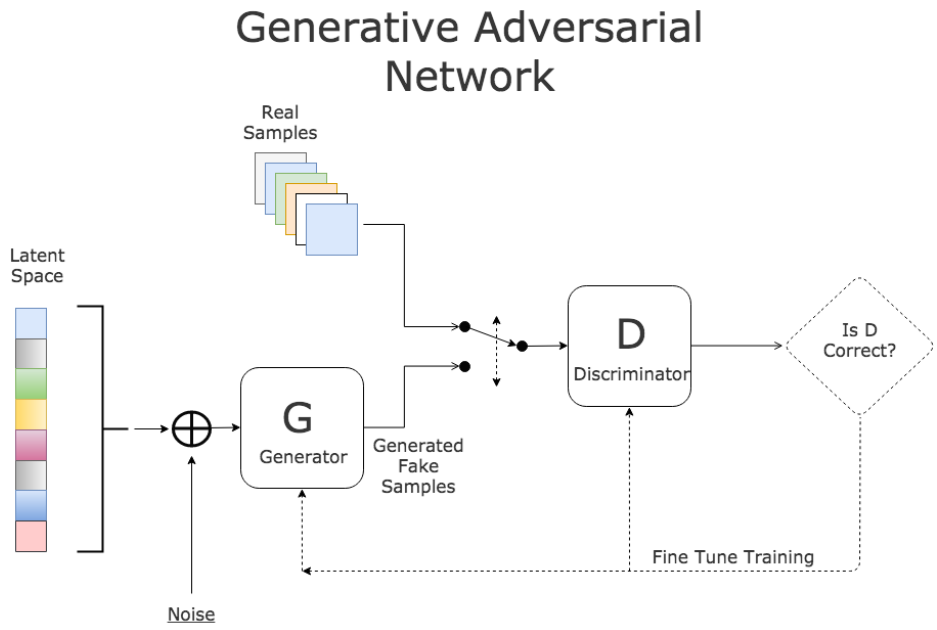


Figure 13.3: GAN Architecture

A Generative Adversarial Network (GAN) is a distinctive architecture that is based on the interaction of two neural networks: the generator and the discriminator. These networks engage in a dynamic process to collectively create synthetic data that mirrors real data distributions. The generator is at the core of this framework, tasked with crafting synthetic data samples. It initiates its process with a random noise vector, which is progressively transformed through a series of hidden layers. The generator's ultimate goal is to learn a transformation that is capable of generating data that is, ideally, indistinguishable from actual data.

On the opposite side of this adversarial interplay stands the discriminator. Functioning as a classifier, the discriminator's purpose is to differentiate between real and synthetic data samples. By assessing the inputs it receives, it computes a probability score, reflecting the likelihood that the input is genuine.

The training mechanism of a GAN revolves around an adversarial dynamic. It goes as follows:

1. **Generator Training:**
    - In the first step, the generator is trained while keeping the discriminator fixed.
    - The generator aims to create synthetic data that is so convincing that it "fools" the discriminator into misclassifying it as real data.
    - To achieve this, the generator takes random noise vectors as input and generates synthetic data samples.
    - The generated data samples are then fed into the discriminator, and the generator's loss is computed based on how well the discriminator is deceived.

- The generator's objective is to minimize this loss, effectively refining its ability to create data that appears authentic.

2. **Discriminator Training:**

   - In the second step, the discriminator is trained while the generator is kept fixed.
   - The discriminator's role is to distinguish between real and synthetic data.
   - During this step, both real data samples from the dataset and synthetic data samples created by the generator are used as input to the discriminator.
   - The discriminator's loss is calculated by comparing its accuracy in classifying real and synthetic data.
   - The discriminator seeks to maximize this loss by improving its ability to correctly classify the two types of data.

13.2.2 Limitations

The main limitation of GANs is its instability: GANs can sometimes suffer from mode collapse, where the generator produces a limited set of data samples, failing to capture the full diversity of the real data distribution. This occurs when the generator manages to fool the discriminator with a subset of data patterns, ignoring other modes present in the data. The generator and discriminator get stuck in a suboptimal state, leading to slow convergence or even divergence.

13.2.3 Deep Convolutional GANs

Deep Convolutional Generative Adversarial Networks (DCGANs) stand apart from regular GANs due to utilization of convolutional neural networks (CNNs) within both the generator and discriminator components. In a regular GAN, the reliance on fully connected layers in both the generator and discriminator can limit the network's capacity to capture intricate features present in images. In contrast, the defining characteristic of DCGANs lies in their adoption of convolutional layers throughout the architecture in the generator. Additionally, DCGANs employ transposed convolutional layers (deconvolutional layers) for image upsampling. This innovation enables the generator to craft higher-resolution images with finer details and textures.

13.2.4 StyleGANs

StyleGANs, or Style Generative Adversarial Networks, are an evolution of regular GANs with enhanced control over generated images. They offer fine-grained control of features like details, textures, and styles, resulting in highly realistic and diverse images. They are known for generating high-resolution images with progressive growing techniques, and offer customizability, allowing a user to control elements of the generated output.

13.3 Diffusion Models

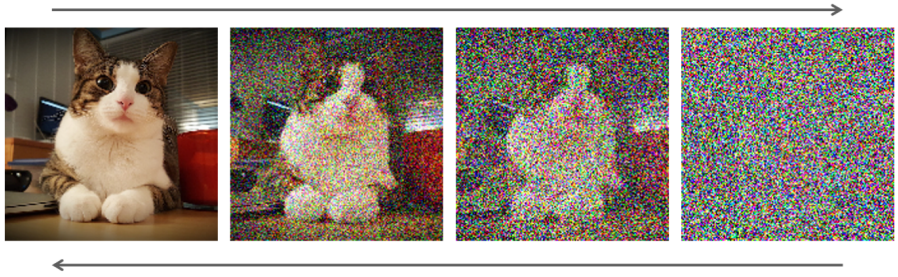


Figure 13.4: Diffusion Process Example

13.3.1 Architecture

The architecture of a diffusion model lies around a distinctive framework that specializes in synthesizing images of high quality from initial noise. The fundamental concept underpinning diffusion models is the gradual refinement of a noisy image across a sequence of steps, progressively molding it into a representation that closely mirrors a genuine image.

Diffusion models operate through a structured series of steps, often referred to as time steps. Starting with a random noise image as input, these steps orchestrate a sequence of transformations. These transformations serve a dual purpose: introducing controlled increments of noise to the image and iteratively updating it to reduce noise levels.

At the heart of a diffusion model is a neural network. This network, frequently featuring convolutional layers, analyzes the image and calculates the adjustments necessary to reduce noise at each time step. By learning the intricate features of actual images, the neural network becomes adept at steering the diffusion process toward generating images that look both real and detailed.

13.3.2 Limitations

The main limitation of diffusion models stems from the intensive nature of their denoising process during both training and image generation. The iterative steps involved in gradually reducing noise levels require substantial computational resources and time. This complexity can hinder real-time applications and scalability, making diffusion models less suitable for scenarios demanding quick image generation.

Chapter 14

Regression Algorithms

14.1 *k*-Nearest Neighbors Regressor

Type: Non-parametric.
Hyperparameter: k.

1. **Initialize Training Set**: Let the training set be denoted by T, where $T = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. Each data point x_i in T represents the features, and y_i represents the corresponding response variable.
2. **Select a Value for** k: Choose a positive integer k, which represents the number of nearest neighbors to consider for regression.
3. **Input a Test Value** x: Given a new test instance x, for which we want to predicted the associated response value y, calculate its distance to each training instance x_i using a distance metric such as Euclidean distance.
4. **Find Nearest Neighbors**: Identify the k training instances in T that are closest to the test instance x based on the calculated distances.
5. **Average the Response of Nearest Neighbors**: Sum over all of the response values of the k nearest neighbors, and then divide by the number of neighbors k.

14.2 Linear, Multi-Linear, & Polynomial Regressors

Type: Parametric.
Parameters: Coefficient Values $\beta_0, \beta_1, ...$, one per predictor.
Hyperparameter: d (degree – only for polynomial regression).

1. **Initialize Training Set**: Reformat the input data as the design matrix $\mathbf{X}$ and response vector $\mathbf{Y}$.
2. **Add Constant Column**: Add a column of ones such that fitting the model offers and intercept value.
3. In the case of *polynomial regression*, create columns for predictor variables raised to all degrees from 2 to d (predictor polynomial products are also create to include interaction information).
4. **Find Coefficients**: Use whichever linear regression approach is best to find the parameter values β.
5. **Prediction**: For new input data, add a column of ones, then multiply by the coefficient vector to get the model's predictions.

14.3 Decision Tree Regressor

Type: Non-parametric.
Hyperparameters: Impurity metric, stopping criterion, & pruning hyperparameter α.

1. **Initialize Training Set**: Let the training set be denoted by T, where $T = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Each data point x_i in T represents the features, and y_i represents the corresponding response variable.
2. **Define Stopping Criteria**: Specify the stopping criteria for tree growth, such as a maximum tree depth or a minimum number of samples per leaf, as well as the impurity metric used.
3. **Recursive Splitting**: Starting from the root node, recursively split the data based on feature values to create child nodes (where optimal splits are the ones that reduce MSE the most).
4. **Stop Splitting**: If the stopping criteria are met, stop further splitting and create a leaf node. The leaf node represents a predicted response value, which is typically the average or median of the response values in that node.
5. **(Optional) Pruning**: After the tree is constructed, apply pruning techniques to improve generalization and reduce overfitting.
6. **Prediction**: Given a new test instance, traverse the decision tree by comparing its feature values at each node. Follow the appropriate path based on the feature thresholds until reaching a leaf node. The predicted response value for the test instance is the value associated with the reached leaf node.

14.4 Random Forest Regressor

Type: Ensemble, Non-parametric.
Hyperparameters: Number of trees, maximum depth, minimum samples for a leaf node, feature subset size.

1. **Data Preprocessing**: Split the dataset into a training set and a test set for model evaluation.
2. **Initialize Ensemble**: Define the number of trees to include in the random forest ensemble. Set hyperparameters such as the maximum depth of each tree, the minimum number of samples required to split an internal node, and the size of the feature subset considered for each split.
3. **Tree Construction**: Create the specified number of decision trees, training each on a dataset sampled with replacement from the training set, where only a subset of the features is used to train the tree.
4. **Ensemble Prediction**: For a new test instance, obtain predictions from each tree in the ensemble. Aggregate the predictions (by averaging for regression) to obtain the final prediction.

14.5 Gradient Boosting Regressor

Type: Ensemble, Non-parametric.
Hyperparameters: Number of boosting stages, learning rate, maximum depth of individual trees, loss function.

1. **Data Preprocessing**: Split the dataset into a training set and a test set for model evaluation.
2. **Initialize Ensemble**: Define the number of boosting stages (number of trees) to include in the gradient boosting ensemble. Set hyperparameters such as the learning rate, the maximum depth of individual trees, and the loss function to optimize.
3. **Initialize Target Values**: Set the initial target values for the ensemble as the labels of the training set.
4. **Iterative Training**:
   (a) For each boosting stage:
   (b) **Tree Construction**: Create a decision tree, training it on the current target values. The tree is typically constructed to minimize the specified loss function.
   (c) **Predictions**: Obtain predictions from the newly created tree for the training set.
   (d) **Update Target Values**: Update the target values for the next boosting stage by adjusting them based on the residuals (the differences between the current predictions and the true labels).
5. **Ensemble Prediction**: For a new test instance, obtain predictions from each tree in the ensemble and combine them to obtain the final prediction. The predictions are usually aggregated by summing them (for regression) and applying the learning rate to control the contribution of each tree.

Chapter 15

Classification Algorithms

15.1 *k*-Nearest Neighbors Classifier

Type: Non-parametric.
Hyperparameter: k.

1. **Initialize Training Set**: Let the training set be denoted by T, where $T = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. Each data point x_i in T represents the features, and y_i represents the corresponding class label.
2. **Select a Value for** k: Choose a positive integer k, which represents the number of nearest neighbors to consider for regression.
3. **Input a Test Value** x: Given a new test instance x, for which we want to predicted the associated class label y, calculate its distance to each training instance x_i using a distance metric such as Euclidean distance.
4. **Find Nearest Neighbors**: Identify the k training instances in T that are closest to the test instance x based on the calculated distances.
5. **Determine Majority Class**: Count the number of instances belonging to each class label among the k nearest neighbors. Assign the test instance x the class label that is most frequent among the k neighbors.

15.2 Support Vector Machine (SVM) Classifier

Type: Parametric.
Parameters: w, the normal direction of the plane, b, a form of threshold.
Hyperparameters: C, a regularization parameter, kernel function choice (Linear, RBF), γ, the radius of influence of each support vector.

1. **Data Preprocessing**: Normalize or standardize the input features to ensure they have the same scale. Split the dataset into a training set and a test set for model evaluation.
2. **Model Training**: Define the SVM classifier and set the desired hyperparameters (e.g., the kernel type, regularization parameter C, etc.). Train the SVM classifier using the training set.
3. **Model Evaluation**: Predict the class labels for the test set using the trained SVM classifier. Calculate evaluation metrics to assess the model's performance.
4. **Hyperparameter Tuning**: Perform hyperparameter tuning using techniques like grid search or randomized search to find the optimal hyperparameters. Evaluate the performance of different hyperparameter combinations and select the best-performing one.

15.3 Decision Tree Classifier

Type: Non-parametric.
Hyperparameters: Impurity metric, stopping criterion, & pruning hyperparameter α.

1. **Initialize Training Set**: Let the training set be denoted by T, where $T = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Each data point x_i in T represents the features, and y_i represents the corresponding class label.
2. **Select a Splitting Criterion**: Choose a splitting criterion, as well as the stopping condition.
3. **Construct the Tree**: Recursively build the decision tree by selecting the best attribute to split the data at each internal node based on the chosen splitting criterion.
4. **Stopping Growth**: If any of the stopping criteria are met, stop growing the tree and proceed to the next step.
5. **Assign Class Labels to Leaf Nodes**: Once the tree is constructed, assign class labels to the leaf nodes based on the majority class of the instances in each leaf node.
6. **(Optional) Pruning**: After the tree is constructed, apply pruning techniques to improve generalization and reduce overfitting.
7. **Prediction**: Given a new test instance, traverse the decision tree from the root node to a leaf node by following the attribute conditions. The predicted class label is determined based on the class of the corresponding leaf node.

15.4 Random Forest Classifier

Type: Ensemble, Non-parametric.
Hyperparameters: Number of trees, maximum depth, minimum samples for a leaf node, feature subset size.

1. **Data Preprocessing**: Split the dataset into a training set and a test set for model evaluation.
2. **Initialize Ensemble**: Define the number of trees to include in the random forest ensemble. Set hyperparameters such as the maximum depth of each tree, the minimum number of samples required to split an internal node, and the size of the feature subset considered for each split.
3. **Tree Construction**: Create the specified number of decision trees, training each on a dataset sampled with replacement from the training set, where only a subset of the features is used to train the tree.
4. **Ensemble Prediction**: For a new test instance, obtain predictions from each tree in the ensemble. Aggregate the predictions (by majority voting for classification) to obtain the final prediction.

15.5 AdaBoost Classifier

Type: Ensemble, Non-parametric.
Hyperparameters: Number of estimators (base classifiers), learning rate.

1. **Data Preprocessing**: Split the dataset into a training set and a test set for model evaluation.
2. **Initialize Weights**: Assign equal weights to each sample in the training set.
3. **Progressive Model Creation**: Continuously:
   (a) **Train Base Classifier**: Train a base classifier (like a decision stump) on the current weighted training set.
   (b) **Compute Error**: Calculate the weighted error rate of the base classifier by summing the weights of misclassified samples.
   (c) **Compute Classifier Weight**: Compute the weight of the current base classifier based on its error rate.
   (d) **Update Sample Weights**: Update the weights of the training samples. Increase the weights of misclassified samples to emphasize their importance in the next iteration.
   (e) **Normalize Weights**: Normalize the sample weights so that they sum to 1.
4. **Ensemble Prediction**: For a new test instance, obtain predictions from each base classifier in the ensemble. Weight the predictions by the classifier weights obtained in the previous steps. Aggregate the weighted predictions to obtain the final prediction.

Chapter 16

Clustering Algorithms

16.1 k-Means

16.1.1 Algorithm

The k-means clustering algorithm identifies a prespecified k number of clusters given a dataset. It achieves this objective by doing the following:

1. Choose the desired number of clusters, k.
2. Initialize k cluster centroids randomly or by using a specific initialization method.
3. Assign each data point to the nearest centroid based on Euclidean distance.
4. Update the centroids by computing the mean of all data points assigned to each cluster.
5. Repeat steps 3 and 4 until convergence or until a maximum number of iterations is reached.

16.1.2 Finding the Optimal Clusters

To determine the best way to split data into k clusters, the idea of inertia is used. Inertia refers to the sum of squared distances between each data point and its nearest centroid. It quantifies the compactness or coherence of the clusters. A lower inertia value indicates that the data points within each cluster are closer to their respective centroid, suggesting a better clustering solution.
The inertia, denoted as I, in k-means clustering is calculated as:

$$I = \sum_{i=1}^{n} \min_{\mu_j \in C} \|x_i - \mu_j\|^2$$

where n is the number of data points, x_i represents each data point, μ_j represents the centroid of the cluster C to which x_i is assigned, and $\|\cdot\|^2$ denotes the squared Euclidean distance.

16.1.3 Finding the Optimal k

As for choosing the appropriate number of clusters k, inertia isn't of much help. Inertia naturally decreases with increasing k: as k increases, the inertia tends to decrease since each data point can be assigned to a nearby centroid, resulting in smaller distances. However, adding more clusters may not necessarily yield meaningful or distinct partitions of the data. Instead, we use something called a silhouette score to determine an appropriate k.

The silhouette score is a measure of how well-defined and separated clusters are in a clustering solution. It combines both the compactness within clusters and the separation between clusters. It is useful for determining the appropriate number of clusters (k) as it provides a quantitative evaluation of different clustering solutions.

The silhouette score for a single data point is calculated as follows:

1. Compute the average distance between the data point and all other points within the same cluster. Denote this as a (intra-cluster distance).

2. Calculate the average distance between the data point and all points in the nearest neighboring cluster. Denote this as b (inter-cluster distance).

3. The silhouette score (s) for the data point is then calculated as:

$$s = \frac{b - a}{\max(a, b)}$$

This process is repeated for all points, and the overall silhouette score is the average of the silhouette scores of all data points. A higher silhouette score close to 1 indicates well-defined and separated clusters, while scores close to 0 suggest overlapping or ambiguous clusters. Negative scores indicate potential misassignments. To determine the appropriate number of clusters (k), evaluate the silhouette scores for different k values. The optimal k corresponds to the highest average silhouette score, indicating a better-defined and more separated clustering solution.

The silhouette scores can be used in a silhouette diagram to better understand which k to use. A silhouette diagram is a visual representation of the silhouette scores for each data point in a clustering solution. The diagram typically consists of bars representing individual data points and a dashed line indicating the average silhouette score. In a silhouette diagram:

- Each knife represents a cluster; its length corresponds to sorted silhouette scores of the points, and its height corresponds to the number of points in the cluster.
- The dashed line represents the average silhouette score across all data points. It serves as a reference for comparing the individual knives.
- Good clustering solutions tend to have greater mean silhouette scores, and have their knives slashing through the mean silhouette score, indicating well-separated and internally coherent clusters. Good solutions (when logically appropriate) also produce clusters of similar sizes.

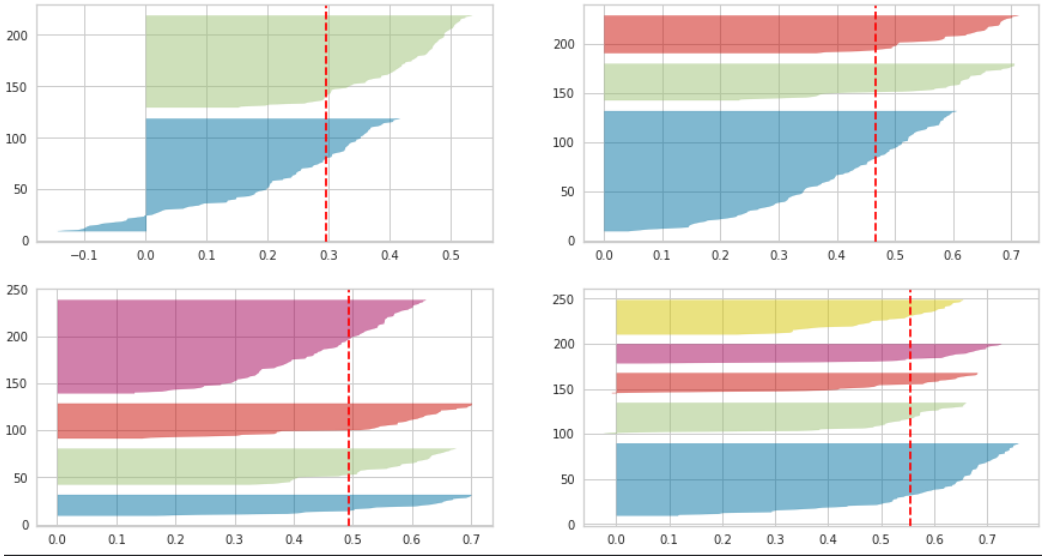


Figure 16.1: Example Silhouette Diagram

16.2 DBSCAN

DBSCAN, or Density-Based Spatial Clustering of Applications with Noise, is a density-based clustering algorithm that groups data points based on their density and proximity. Here's how it works:

1. Select an arbitrary data point and determine its neighborhood within a specified radius (ϵ).
2. If the number of data points within the neighborhood exceeds a predefined threshold ($min_samples$), mark the data point as a core point and expand its cluster.
3. Expand the cluster by recursively adding all directly reachable data points within the specified radius.
4. Repeat steps 1-3 for unvisited data points until all points have been processed.
5. Any data points that are not assigned to a cluster are considered anomalies or outliers.

DBSCAN operates based on two parameters: the radius (ϵ) and the minimum number of points ($min_samples$) required to form a dense region. It can discover clusters of arbitrary shapes and is robust to noise. Although useful, it has some drawbacks, too. DBSCAN struggles with datasets that have varying density across different regions. It may fail to properly cluster data points in regions with significantly different densities, resulting in under- or over-segmentation. Additionally, in high-dimensional spaces, the concept of density becomes less reliable due to the curse of dimensionality. DBSCAN's performance degrades as the number of dimensions increases, as the density-based nature of the algorithm becomes less effective.

16.3 GMM

16.3.1 Algorithm

A Gaussian Mixture Model (GMM) is a probabilistic model used for clustering and density estimation. It assumes that the data is generated from a mixture of Gaussian distributions. The GMM aims to find the parameters that best represent the underlying mixture components, including their means (μ), covariances (Σ), and mixing proportions (π). This enables the GMM to capture complex data distributions, identify clusters, and estimate the likelihood of new data points belonging to each component. Although general useful, using GMMs doesn't work that well with clusters of non-elliptical shapes or ones spread across many dimensions. The algorithm works as follows:

1. Initialize the parameters of the GMM, including the number of components (K), the means (μ_k), covariances (Σ_k), and mixing proportions (π_k) for each component.

2. Expectation Step (E-step):

   (a) Calculate the responsibilities (γ_{ik}) of each data point x_i for each component k using the current parameter estimates:

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)}$$

3. Maximization Step (M-step):

   (a) Update the means (μ_k), covariances (Σ_k), and mixing proportions (π_k) using the responsibility weights:

$$\mu_k = \frac{\sum_{i=1}^{N} \gamma_{ik} x_i}{\sum_{i=1}^{N} \gamma_{ik}}$$

$$\Sigma_k = \frac{\sum_{i=1}^{N} \gamma_{ik} (x_i - \mu_k)(x_i - \mu_k)^T}{\sum_{i=1}^{N} \gamma_{ik}}$$

$$\pi_k = \frac{1}{N} \sum_{i=1}^{N} \gamma_{ik}$$

4. Repeat steps 2 and 3 until convergence, which can be determined by a change in the log-likelihood or a maximum number of iterations.

GMMs can be used for outlier detection by calculating the likelihood of each data point under the learned GMM. Points with low likelihoods are considered outliers. By setting a threshold on the likelihood scores, data points can be classified as outliers or inliers. However, note that GMMs try to fit all the data, including outliers; if there are too many of them this will bias the model's view of "normality", and some outliers may wrongly be considered normal. If this happens, it's good to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset.

16.3.2 Finding the Optimal k

The Bayesian Information Criterion (BIC) and Akaike Information Criterion (AIC) are statistical measures used to determine the optimal number of Gaussian components (k) in a GMM. BIC and AIC provide a balance between model complexity and goodness of fit. The lower the BIC or AIC value, the better the model fit. To determine the optimal k using BIC or AIC, one typically evaluates the values for different k and selects the model with the lowest BIC or AIC score. This choice represents a trade-off between model complexity and the model's ability to explain the data.

An extension of GMMs, called Bayesian GMMs, allows for cluster matching without the explicit need to set the number of clusters; instead, an upper bound is needed, and the model zeros out the weights of clusters deemed unnecessary.

Chapter 17

Implementation Details

17.1 Numeric Instability

Numeric instability in machine learning refers to issues that arise due to the limitations of representing real numbers in finite precision. When computations involve very large or very small numbers, standard numerical methods may lose accuracy, leading to unreliable or erroneous results. These instabilities can negatively impact the training and performance of machine learning models.
The sigmoid function is a common activation function where instability can be an issue, causing two variants of the function to arise:

1. **Version 1:** The standard sigmoid function is $\frac{e^x}{1+e^x}$.

   Numeric Instability with Version 1: The exponential function e^x can lead to numeric overflow when the input x is very large (positive), resulting in undefined behavior.

2. **Version 2:** The alternative form is $\frac{1}{1+e^{-x}}$.

   Numeric Instability with Version 2: The exponential function e^{-x} can lead to numeric overflow when the input x is very negative, causing undefined behavior.

17.2 Sci-Kit Learn

17.3 TensorFlow

17.4 Keras

17.5 HuggingFace

Chapter 18

UNINCORPORATED

18.1 Optimization

We optimize a loss function to arrive at a the parameters of the model that result in the lowest loss (depending on which metric we use). We pick a loss function and try to find where it is minimized (a minimum point). To do so we can employ three strategies:

- Brute Force: Try every possible combination of parameter values.
- Exact Calculation: Find exact parameters $\beta_0, \beta_1, ...$ such that the gradient $\nabla L = \left[\frac{\partial L}{\partial \beta_0}, \frac{\partial L}{\partial \beta_1}, ...\right] = 0$.
- Greedy Algorithm: Finding the analytical solution is only possible some of the time (and only in rare cases). The rest of the time, we resort to algorithms like Gradient Descent.

18.2 Sample Two-Image Layout

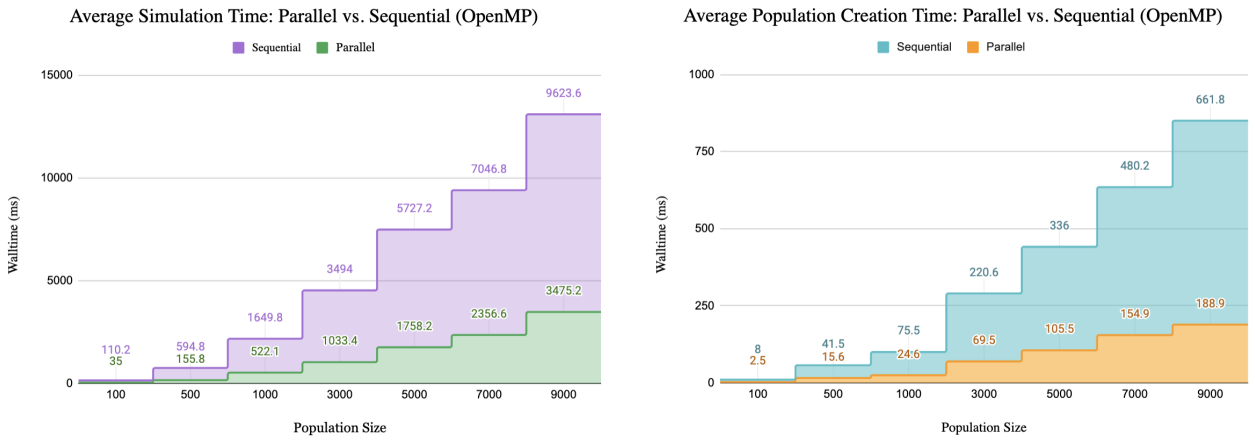


Figure 18.1: Linear Increase in Average Simulation and Population Creation Times