# Machine Learning

Matthew Chen

May 6, 2024

# Contents

**1 Preface** **3**
1.1 Supervised Learning . . . 3
1.2 Loss Functions . . . 5

**2 Linear Regression** **7**
2.1 The Ordinary Least Squares . . . 7
2.2 Ridge Regression . . . 9
2.3 Lasso Regression . . . 11

**3 Classification Methods** **11**
3.1 Logistic Regression . . . 11
3.2 Support Vector Machine (SVM) . . . 13
3.3 k-Nearest Neighbors (kNN) . . . 14

**4 Tree Based Methods** **14**
4.1 Decision Trees (CART) . . . 14
4.2 Random Forest . . . 18

**5 Ensemble Methods** **19**
5.1 Bagging . . . 19
5.2 Stacking . . . 21
5.3 Boosting . . . 21

**6 Neural Networks and Deep Learning** **24**
6.1 The Feedforward Neural Network . . . 24
6.2 Training Neural Networks . . . 26
6.3 Regularization . . . 30

**7 Sequential Neural Networks** **30**
7.1 Recurrent Neural Networks (RNN) . . . 30
7.2 Long Short Term Memory (LSTM) . . . 32
7.3 Sequence-to-sequence models . . . 34
7.4 Attention . . . 35
7.5 Transformer . . . 37

**8 Unsupervised Learning** **43**
8.1 Association Rules . . . 43
8.2 K-means Clustering . . . 44
8.3 Mixture Models and the EM Algorithm . . . 44
8.4 Principal Component Analysis (PCA) . . . 47

# 1 Preface

In this reader, I provide an overview of commonly used statistical learning techniques and their different uses. Focus in these notes will be put towards the understanding of key concepts in Machine Learning and not on theoretical results or implementations in computer code. The key reference for these notes is the second edition Elements of Statistical Learning by Trevor Hastie, Robert Tibshirani, and Rerome Friedman.

## 1.1 Supervised Learning

### 1.1.1 Empirical Risk

Statistical Risk and Empirical Risk are themes central to supervised statistical learning. Consider a regression task where we use the random vector $\vec{x}$ to predict the response variable $y$. Here, the elements of $\vec{x}$ represent different predictor variables or features, and $y$ represents the target variable being predicted. Supervised learning simply means that in our dataset of *observed* examples, we have paired information between the target variable and its corresponding feature variables. In any supervised machine learning task, we formulate the loss function, denoted as $\ell(\theta; \vec{x}, y)$ to denote how close the predicted value is to the true $y$. Here, $\theta$ represents the parameters of the model.

We define Statistical Risk as the expected loss over the random input $\vec{x}$ and $y$:

$$R(\theta) = \mathbb{E}\ell(\theta; \vec{x}, y)) = \int_{\mathbb{R}} p(\vec{x}, y)\ell(\theta; \vec{x}, y)\, dxdy$$

However, the true risk is unknown since the join distribution of the data, $p(\vec{x}, y)$is unknown. Instead, risk is estimated using the observed dataset, taking the average of $n$ individual losses corresponding to $n$ training observations or examples. Denoting the observed data as $\{\vec{x_i}, y_i\}_{i=1}^n$ we define Empirical Risk as the following

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(\theta; \vec{x_i}, y_i)$$

Note that by the Law of Large Numbers, the Empirical Risk asymptotically converges to the true Statistical Risk.

Using empirical risk, we can formulate the estimation of the model parameters $\theta$ as the following optimization problem. This class of estimators is known as the Empirical Risk Minimizers (ERM).

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\ R_n(\theta)$$

Unfortunately, in expectation, the empirical risk under the ERM is not equivalent to the parameter that minimizes the true risk.

$$\mathbb{E}R_n(\hat{\theta}) \neq \min_{\theta} R(\theta)$$

In practice, this problem is resolved by cross validation or the train/test split. In the train/test split framework, the data is split into two different components, a training set and a test set. The model parameters are then estimated using the training data and the model's true out-of-sample performance is evaluated on the test data. The intuition is that we estimate the true performance of the model on new unseen data after training.

In reality, the test error from a single train/test split has high variance - if we simply split the data differently the results would be different. A common why to reduce the variance is to average errors systematically from different train/test partitions of the data (recall that averaging acts to reduce the variance). In k-fold cross validation, the data is split equally into k partitions, and k models are trained. For each model, one of the data partitions is set aside and the rest are used to train the model. Finally, the errors from each of the k folds are averaged. Note that in the extreme case of k=n, we have what is called leave-one-out

cross validation which has the lowest variance but requires the training of n different models, which can be computationally expensive. In practice, k=5 or k=10 is commonly chosen.

A brief note about notation: often, the observed dataset $\{\vec{x_i}, y_i\}_{i=1}^n$ is represented by the design matrix $\mathbf{X} \in \mathbb{R}^{nxp}$ where its rows are the n observation vectors $\vec{x_i}$ and the response vector $\vec{y} \in \mathbb{R}$.

$$X = \begin{bmatrix} — & x_1 & — \\ \vdots & \vdots & \vdots \\ — & x_n & — \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

### 1.1.2 Bias-Variance Tradeoff

Consider a model $f(x)$ that represents the true data generating process

$$y = f(x) + \epsilon$$

where $\epsilon$ is some zero-mean error/noise term with constant variance $\sigma^2$. Our goal in supervised machine learning is to find an approximate function $\hat{f}$ that approximates the true function $f$ as well as possible, which we derive from our observed training data. This can be quantified by the MSPE (mean square prediction error) of $\hat{y}$, which is the expected square difference between predicted value of a never seen before data sample $x$ and the corresponding true $y$ . Note that we cannot compute MSPE, since the true $y$ is not known on our unseen sample. We can write the mean square prediction error as

$$MSPE := \mathbb{E}\left[(y - \hat{f}(x))^2\right]$$

Note that the MSPE can be decomposed into a variance term, a bias term, and an irreducible error term using simple algebra.

$$\begin{aligned} MSPE(\hat{f}) &= \mathbb{E}\left[(y - \hat{f}(x))^2\right] \\ &= \mathbb{E}[(\hat{f}(x) - \mathbb{E}\hat{f}(x))^2] + \left(\mathbb{E}[f(x) - \hat{f}(x)]\right)^2 + \sigma^2 \\ &= Variance(\hat{f}) + Bias(\hat{f})^2 + \sigma^2 \end{aligned}$$

Intuitively, models that are too large and complex for the given task tend to have greater variance. That is, for example, if we draw new data from the same underlying distribution as our original data and retrain the model, the estimates of the model parameters may vary from the first model and causing the predicted outputs to have high variance. The obvious consequence is the first model generalizes poorly to the new data. Intuitively, this is because the first model had enough complexity to learn the noise in the first dataset; so when new data is drawn with a new instance of observed noise, the model performs poorly. We call this regime as overfitting - when the training performance is excellent (since even the noise is learned) yet prediction performance suffers.

Conversely, a model with too little capacity or complexity may not be able to learn the true data generating process well and therefore may have high bias, i.e. on average (in expectation), the model does poorly. However, since the model is much smaller, the variance would be much lower, helping us avoid some of the negative effects of overfitting. This is called the underfitting regime. Therefore, the best models balance the first and the second scenario, and achieves the so-called bias-variance tradeoff. This concept is illustrated in Figure 1.

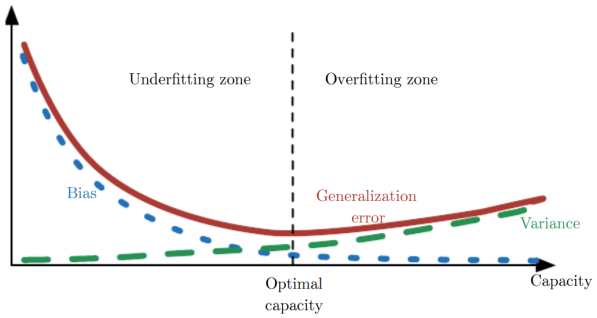


Figure 1: The Bias Variance Tradeoff. *Image Source: Deep Learning*

## 1.2 Loss Functions

A Loss Function defines Statistical Risk at the population level. That is, it is a function that describes how well a model is doing at a supervised learning task. In this section, I highlight popular loss functions, namely cross entropy loss for classification problems and square error loss for regression problems, and their connection to Maximum Likelihood Estimation (MLE) in statistics.

### 1.2.1 Cross Entropy Loss

Consider a classification problem with $k = 1, ..., K$ classes and the cross-entropy loss function.

$$\ell(y_i, \vec{x_i}; \theta) = -\sum_{k=1}^{K} y_{ik} log(f_k(\vec{x_i})),\ i = 1, ..., n$$

where $y_i$ is the true class for the $ith$ training observation, $x_i$ is the corresponding input vector, $\theta$ is the parameter set of the model. Further, we write $y_{ik}$ as shorthand for $\mathbf{1}\{Y_i = k\}$ and $f_k(x_i)$ represents the modeled probability of class $k$, $\mathbb{P}\{Y_i = k | X = x_i; \theta\}$.

Combining all $n$ losses, we have the empirical risk minimizer

$$\begin{aligned}
\hat{\theta} &= \underset{\theta}{\text{argmin}} R_n(\theta) \\
&= \underset{\theta}{\text{argmin}} -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(f_k(\vec{x_i})) \\
&= \underset{\theta}{\text{argmax}} \sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(f_k(\vec{x_i}))
\end{aligned}$$

Aside, let's write out the maximum likelihood estimator for $\theta$. First, we write the joint likelihood, which is the joint distribution evaluated using the observed data. Recall that the essence of maximum likelihood is to maximize the probability of realizing the data we have already observed. Thus, we want to model the data in such a way where the joint likelihood is maximized.

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \mathbb{P}\left(Y_1 = y_1, ...Y_n = y_n | X_1 = x_1, ..., X_n = x_n; \theta\right)$$

Assuming independence between observations, we have

$$\underset{\theta}{\text{argmax}} \prod_{i=1}^{n} \mathbb{P}(Y_i = y_i | X = x_i; \theta)$$

Rewriting using an indicator function, we have

$$\underset{\theta}{\text{argmax}} \prod_{i=1}^{n} \prod_{k=1}^{K} \mathbb{P}(Y_i = y_i | X = x_i; \theta)^{\mathbf{1}\{Y_i = k\}}$$

Rewriting the indicator using the notation defined earlier, we have

$$\underset{\theta}{\text{argmax}} \prod_{i=1}^{n} \prod_{k=1}^{K} \mathbb{P}(Y_i = y_i | X = x_i; \theta)^{y_{ik}}$$

Evaluating the modeled probability

$$\underset{\theta}{\text{argmax}} \prod_{i=1}^{n} \prod_{k=1}^{K} f_k(x_i; \theta)^{y_{ik}}$$

Finally, we see that the log-likelihood is

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} log(f_k(x_i))$$

Thus, minimizing the cross entropy loss gives the same result as maximizing the maximum likelihood when we use our model to estimate the class probabilities.

Cross-entropy originally comes from information theory, and measures the "distance" between two distributions $p$ and $q$. We can think of $p$ as the "true" distribution and $q$ like the computed distribution.

Formally, we write cross-entropy $H(p, q)$ as

$$H(p, q) = -\sum_{k \in \chi} p(k) log(q(k))$$

where $\chi$ is the support of $p$ and $q$.

Consider a "ground truth" distribution $p$ that has probability 1 at the true class and 0 elsewhere. For example, if $y_i = 3$, then $p = [0, 0, 1, 0, 0]$ for a support of $K = 5$ possible classes. In this scenario, $q$ is the computed probability, e.g. from a softmax. For example, $q = [0.01, 0.09, 0.8, 0.05, 0.05]$. The goal, then, is to have the computed probability be as close as possible to the ground truth. With this general framework, we see we arrive at the same cross entropy loss function described above.

### 1.2.2 Square Error Loss

Above, we saw the connection between cross entropy and the maximum likelihood estimator for regression problems. It turns out, square error loss has a very similar interpretation for Gaussian distributed data.

Recall the empirical risk with square error loss

$$R_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i; \theta))^2$$

where $y_i$ is the value of the $ith$ training example, and $f(x_i; \theta)$ is the corresponding predicted value from the model.

Consider modeling the response variable $Y$ with a Gaussian distribution, with mean $f(X)$ and constant variance $\sigma^2$.

$$Y|X \sim \mathcal{N}(f(X; \theta), \sigma^2)$$

Let $\phi(f(x_i), \sigma^2)$ denote the normal probability density function with mean $f(x_i)$ and variance $\sigma^2$. Writing out the joint likelihood of the data, assuming independence between observations, we have

$$\begin{aligned} L\left(y_1, ..., y_n | x_1, ..., x_n; \theta\right) &= \prod_{i=1}^{n} \phi(y_i | f(x_i; \theta), \sigma^2) \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{1}{2\sigma^2}(y_i - f(x_i;\theta)^2\right) \end{aligned}$$

Taking the log of the expression, we arrive at the log-likelihood

$$\ell(\theta) = \sum_{i=1}^{n} log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}(y_i - f(x_i;\theta))^2$$

writing out the maximum likelihood estimator

$$\hat{\theta}_{MLE} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}(y_i - f(x_i;\theta))^2$$

Removing terms that do not depend on $\theta$

$$\begin{aligned} \hat{\theta}_{MLE} &= \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} -(y_i - f(x_i;\theta))^2 \\ &= \underset{\theta}{\text{argmin}} \sum_{i=1}^{n} (y_i - f(x_i;\theta))^2 \end{aligned}$$

We see that minimizing the square error loss gives the MLE estimate.

# 2 Linear Regression

## 2.1 The Ordinary Least Squares

In this section I review the basics of linear regression, which by itself plays an important role in predictive modeling - it is perhaps the simplest and most interpretable model available. Further, its statistical properties are easily obtained given common Gaussian assumptions.

The model form is as follows. Here we consider a model with $p$ coefficients, so that $x_i \in \mathbb{R}^p$ and a response (target) variable $y_i$. An important nuance here is that the first element of $\vec{x_i}$ has $x_{i0} = 1$ , which corresponds to the intercept of the model. We denote the vector of model coefficients as $\beta$, and the random error terms as $\epsilon$

$$y_i = \beta_0 + \sum_{j=1}^{p-1} \beta_j x_{ij} + \epsilon_i$$

In matrix form, considering all $n$ data observations, we have

$$y = X\beta + \epsilon$$

To fit the model, we minimize the square error deviations between the predicted values, denoted as $\hat{y}$ and the ground truth $y$.

$$\hat{\beta}_{OLS} = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (y_i - \beta^T \vec{x_i})^2 = \underset{\beta}{\text{argmin}} ||y - X\beta||_2^2$$

An analytical solution to the OLS problem exists: by taking the gradient of the least-squares objective and setting it to zero we obtain the solution

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

with predicted values

$$\hat{y} = X\hat{\beta}_{OLS} = X(X^T X)^{-1} X^T Y = HY$$

$H = X(X^T X)^{-1} X^T$ is known as the *hat matrix*, which has the form of a projection matrix to the column space of the design matrix $X$. This geometric interpretation is very natural to linear regression: we are finding a member of $R(X)$ (R here denotes the range, or column space), i.e. a linear combination of the columns of X, such that the distance to the ground truth $y$ is as small as possible.

Linear regression has several classic assumptions:

- The relationship between $y$ and $x$ is linear (linearity assumption)
- The error terms are uncorrelated with each other
- The error terms have zero mean and equal variance (no heteroskedasticity)
- The error terms are normally distributed (normality assumption)

The last 3 assumptions can be summarized as $e_i \sim iid\ \mathcal{N}(0, \sigma^2)\ \forall i$ and are essential for statistical inference and testing of the model results. The Gaussian/normality assumption, however, is not needed if the goal is simply to build a predictive model and not use statistical inference. Note that any assumption on the error terms also applies to the fitted values as $\epsilon$ is the only random component of the model, taking the perspective that the observed data in $X$ is fixed.

Under the normality assumptions, we can test the statistical significance of the results by decomposing the total variance (total sum of squares, SSTO) into an explained variance part (regression of of squares, SSR) and an unexplained variance part (error sum of squares, SSE). This is called the sum of squares decompostion.

$$SSTO = SSE + SSR$$
$$\sum_{i=1}^{n}(y_i - \bar{y})^2 = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$

Note that the coefficient of determination, $R^2$, a common measure for goodness-of-fit is simply

$$R^2 = \frac{SSR}{SSTO} = 1 - \frac{SSE}{SSTO}$$

SSTO has degrees of freedom n-1, SSE has degrees of freedom n-p, and SSR has degrees of freedom p-1. Using these quantities we can define the F statistic in this context as

$$F^* = \frac{\frac{SSR}{p-1}}{\frac{SSE}{n-p}} = \frac{MSR}{MSE}$$

We can use the F-statistic to test the significance of the regression relationship, testing the hypothesis:

$$H_0 : \beta = 0$$
$$H_1 : \beta \neq 0$$

Under the null hypothesis, the F-statistic follows a F-distribution with degrees of freedom $p - 1$ and $n - p$, respectively. The p-value then measures how extreme our observed $F^*$ statistic is under the null distribution - if it is too extreme then we have strong evidence *against* the null hypothesis. Technically, the null distribution follows from the fact that the sum of squares follows $\chi^2$ distributions under the null, specifically: $SSE \sim \sigma^2\chi^2_{n-p}$ and $SSR \sim \sigma^2\chi^2_{p-1}$ and the F-statistic is simply the ratio of the two normalized by their respective degrees of freedom. Overall, this procedure known is known as an ANOVA F-test. In practice, ANOVA tests can be an important tool in determining the significance between predictors and can be useful

for variable selection early on in the modeling pipeline.

Note that a common issue that arises in practice is the issue of multicollinearity; i.e., when different X predictors are inter-correlated with each other. In such as scenario, the regression coefficients represent the incremental contribution of that variable *given the other variables already in the model*. Note that a new variable may not add much to the model if it is highly correlated with the existing predictor variables since most of its information is already in the model. Importantly, the presence of collinearity also inflates the variance of the model. This highlights the importance of variable/feature selection in the bias-variance tradeoff. Other methods that help the OLS achieve the bias-variance tradeoff include penalizing the size of the coefficients (such as in Ridge and Lasso regression), and as an added benefit these methods also provide solutions in the case of $p > n$ where the OLS solution does not exist.

The Variance Inflation Factor (VIF) is a measure of how much multicollinearity exists in a model. The VIF for the kth predictor $X_k$ is defined as

$$VIF_k := \frac{Var(\hat{\beta}_k)}{\sigma^2}$$

It can be shown that the VIF can also be written as the following, which is easier to compute

$$VIF_k = \frac{1}{1 - R_k^2}$$

where $R_k^2$ is the coefficient of determination when regressing all other predictors in the model except $X_k$ to $X_k$ (denote this regression as $X_k \sim X_{-k}$). Intuitively, when $R_k^2 > 0$, then $VIF_k > 1$ and the variance is inflated due to inter-correlation between the predictors. When there is perfect multicollinearity (i.e. $R_k^2 = 1$ and $VIF_k = \infty$), the least square estimators are not defined. In practice, $max_k VIF_k > 10$ is an indication of high multicollinearity in the model.

## 2.2 Ridge Regression

The ordinary least squares solution minimizes the square deviance between the predicted values and the observed data. In contrast, the ridge solution regularizes the OLS by solving a penalized version of the residual sum of squares, using the Euclidean (L2) norm of the parameter vector. Intuitively, the larger the parameters, the greater the penalty. Overall, this penalty shrinks the size of the coefficients (toward 0) and trades some bias for less variance.

Using $\lambda$ as a shrinkage parameter, we can formulate the problem as the following

$$\begin{aligned}\hat{\beta}_{ridge} &= \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \beta^T \vec{x_i})^2 + \lambda ||\beta||_2^2 \\ &= \underset{\beta}{\operatorname{argmin}} (\vec{y} - X\beta)^T (\vec{y} - X\beta) + \lambda \beta^T \beta\end{aligned}$$

Solving for $\hat{\beta}_{ridge}$ we obtain

$$\hat{\beta}_{ridge} = (X^T X + \lambda I)^{-1} X^T \vec{y}$$

The mathematical interpretation of ridge regression becomes clear if we apply a thin singular value decomposition to X. Recall that the singular value decomposition generalizes the eigenvalue/eigenvector decomposition to any rectangular matrix (rather than just square matrices with $n$ independent eigenvectors). In other words, we can decompose the data matrix as $X = UDV^T$.

Here we briefly review the singular value decomposition. Suppose $X \in \mathbb{R}^{nxp}$. Here, the columns of $V$ contain the $n$ eigenvectors of the symmetric and square matrix $X^T X$ which span $\mathbb{R}^n$ and are orthogonal to each other. In other words, the columns of $V$, known as the left singular vectors, form an orthogonal basis for

$\mathbb{R}^n$. Thus, $V \in \mathbb{R}^{nxn}$ .

Suppose $X$ is Rank $r$ and therefore has $r$ non-zero singular values. The columns of $U$ contains the normalized products of $X$ and the left singular vectors $\vec{v_i}$. Note that we will have $r$ non-zero vectors of such products, and the resulting vectors are orthogonal to each other and span the column space of $X$.

$$\{u_1, .., u_r\} = \{\frac{Xv_1}{||Xv_1||}, .., \frac{Xv_r}{||Xv_r||}\}$$

In standard SVD, we extend the nonzero vectors above $\{u_1, .., u_r\}$ to $\{u_1, .., u_p\}$ to be a orthonormal basis for $\mathbb{R}^m$. Then $U \in \mathbb{R}^{nxp}$. Finally, we write the block matrix $\Sigma \in \mathbb{R}^{pxn}$ where the top right block is the matrix $D = diag(\sigma_1, ..., \sigma_r)$ containing the $r$ non-zero singular values $\sigma$ of $X$ and all other elements are zero. Note that the singular values of X correspond to the square-root of the eigenvalues of $X^TX$.

With this setup, we can write

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

Thin SVD is very similar, but instead of extending the columns of $U$ and building a block matrix for $\Sigma$ we keep only the non-zero singular values and corresponding left singular vectors. Then we have $U \in \mathbb{R}^{nxr}$ , $D \in \mathbb{R}^{rxr}$ , and $V^T \in \mathbb{R}^{rxn}$ where $U$ and $V$ are semi-orthogonal matrices (since they are not square) satisfying $U^TU = V^TV = I$.

Then we can write

$$X = UDV^T$$

Going back to Ridge regression. If we evaluate $X = UDV^T$ then the fitted values are as follows:

$$\begin{aligned}
\hat{y} &= X\hat{\beta}_{ridge} \\
&= (X^TX + \lambda I)^{-1}X^T\vec{y} \\
&= UD(D^2 + \lambda I)^{-1}DU^T\vec{y} \\
&= \sum_{j=1}^{p} u_j \frac{\sigma_j^2}{\sigma_j^2 + \lambda} u_j^T \vec{y}
\end{aligned}$$

By convention, we arrange the singular values in decreasing order so that $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p$ .

Further, we can write the sample covariance matrix of X as

$$S_x = \frac{1}{n}X^TX$$

It turns out that the sample variance of the ith principal component (PCA is discussed later on in these notes) is

$$Var(PC_i) = \frac{1}{n}\sigma_i^2$$

By looking at the shrinkage term $\frac{\sigma_j^2}{\sigma_j^2+\lambda}$ we see that the ridge solution shrinks the directions (i.e. features, columns of $X$) corresponding to the *principal components with the least variance the most.*

Finally, as a sanity check, note that if $\lambda = 0$, then the result is the same as the ordinary least squares solution.

$$\hat{y} = X\hat{\beta}_{OLS} = X(X^TX)^{-1}X^T\vec{y} = UU^Ty = \sum_{j=1}^{p} u_j u_j^T \vec{y}$$

### 2.3 Lasso Regression

The lasso solution is similar to ridge, except it employs an L1 penalty on the coefficients instead of an L2 penalty.

$$\hat{\beta}_{lasso} = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (y_i - \beta^T \vec{x_i})^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

By the nature of this constraint, lasso sets coefficients exactly to zero, and can be seen as a type of continuous variable selection. That is, in LASSO we make an assumption of sparsity. Specifically, the ridge solution does a proportional shrinkage while the lasso shifts coefficients by $\lambda$ and then truncates at zero (see Figure 2 below). This is called soft-thresholding. An advantage of lasso is that it does variable selection and provides estimates simultaneously, although the estimates for large coefficients are biased by $\lambda$.

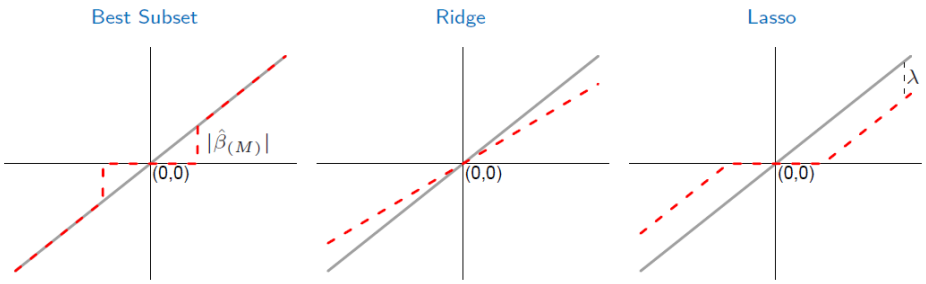


Figure 2: Proportional shrinking (ridge) and soft thresholding (lasso). Hard thresholding is represented by the best subset selection, which is much harder to solve in practice. The gray slopped line represents the unrestricted estimates. *Image Source: Elements of Statistical Learning*

The final question is how the shrinkage parameter $\lambda$ is chosen for lasso and ridge. In practice, this typically done using leave-one-out or k-fold cross validation. Considering a set of candidates $\{\lambda_1, ..., \lambda_m\}$ we choose

$$\hat{\lambda} = \underset{\lambda}{\text{argmin}} CVError(\lambda)$$

## 3 Classification Methods

### 3.1 Logistic Regression

Logistic regression is a linear supervised classification method, typically used a binary classifier. The idea is to model the log-odds ratio as a linear relationship with $\vec{x}$. Let $p(x;\beta) = \mathbb{P}(Y = 1 | X = \vec{x})$ be the probability that the outcome is $Y = 1$ given $X = \vec{x}$. In the case of binary classification, we model

$$logit(p) = log\left(\frac{p(x;\beta)}{1 - p(x;\beta)}\right) = \beta_0 + \beta^T x$$

Re-arranging the terms

$$\begin{aligned}
\frac{p(x;\beta)}{1 - p(x;\beta)} &= exp(\beta_0 + \beta^T x) \\
p(x;\beta) &= exp(\beta_0 + \beta^T x) - p(x;\beta) exp(\beta_0 + \beta^T x) \\
(1 + exp(\beta_0 + \beta^T x)) p(x;\beta) &= exp(\beta_0 + \beta^T x) \\
p(x;\beta) &= \frac{exp(\beta_0 + \beta^T x)}{1 + exp(\beta_0 + \beta^T x)}
\end{aligned}$$

From this we see that $p(x;\beta)$ ranges from $(0, 1)$, which makes it a valid probability.

The set where the log-odds is 0 (i.e. the probabilities of both classes are equal) is called the *decision boundary*. In set-builder notation, we can write the decision boundary as $\{\vec{x} : \beta_o + \beta^T x = 0\}$. On the decision boundary, the probability of predicting either $Y = 1$ or $Y = 0$ is both 0.5.

Note that in logistic regression, the decision boundary is linear, and thus has very similar assumptions to linear regression. Overall, we make the following assumptions

- (In the case of binary classification), we assume the target variable has binary outcomes
- We assume that the logit (log-odds) is linear in its relationship with the predictor variables $\vec{x}$
- We assume that observations are independent of each other
- We assume that there is no perfect or complete separation

In the case of complete separation, the logistic coefficients will tend to (positive or negative) infinity due to an unstable maximum likelihood. This is intuitive since if $p = 1$ or $p = 0$, the log-odds does not exist. This problem can be addressed with penalized (e.g. L2) logistic regression.

We can also generalize the binary classification case to the multiclass case (with $K$ classes instead of 2). The model has the following form, comparing every class $Y = 1, ..., K-1$ to class $K$. Here, $\beta_{k0}$ is the intercept term for class $k$ and $\vec{\beta_k}$ are the other parameters for class $k$.

$$\begin{aligned} log\frac{\mathbb{P}(Y=1|X=x)}{\mathbb{P}(Y=K|X=x)} &= \beta_{10} + \beta_1^T x \\ log\frac{\mathbb{P}(Y=2|X=x)}{\mathbb{P}(Y=K|X=x)} &= \beta_{20} + \beta_2^T x \\ &... \\ log\frac{\mathbb{P}(Y=K-1|X=x)}{\mathbb{P}(Y=K|X=x)} &= \beta_{(k-1)0} + \beta_{k-1}^T x \end{aligned}$$

Rearranging the terms, we can write

$$\begin{aligned} \mathbb{P}(Y=k|X=x) &= \frac{exp(\beta_{k0} + \beta_k^T x)}{1 + \sum_{\ell=1}^{K-1} exp(\beta_{\ell 0} + \beta_\ell^T x)} \; for \; k = 1, ..., K-1 \\ \mathbb{P}(Y=K|X=x) &= \frac{1}{1 + \sum_{\ell=1}^{K-1} exp(\beta_{\ell 0} + \beta_\ell^T x)} \end{aligned}$$

which clearly suns to 1. Thus, these are valid multinomial probabilities.

As shorthand, we can write the parameter set:

$$\theta = \{\beta_{10}, ..., \beta_{(K-1)0}, \beta_1^T, ..., \beta_{K-1}^T\}$$

and denote the probabilities

$$p_k(x; \theta) = \mathbb{P}(Y = k|X = x)$$

### 3.1.1 Fitting Logistic Regression

Logistic regression is typically fit using maximum likelihood estimation (recall in classification, this corresponds to the cross-entropy loss function). The log-likelihood for the multiclass classification case for $n$ observations is as follows

$$\ell(\theta) = \sum_{i=1}^{n} logp_{y_i}(x_i; \theta)$$

where $p_{y_i} = \mathbb{P}(Y = y_i|X = x_i; \theta)$.

In the case of binary outcomes, we can write the log-likelihood as

$$\ell(\beta) = \sum_{i=1}^{n} y_i log p(x_i; \beta) + (1 - y_i) log(1 - p(x_i; \beta))$$

where $p = \mathbb{P}(Y = 1|X = x_i; \beta)$.

## 3.2 Support Vector Machine (SVM)

Consider a binary classification problem with $y_i \in \{+1, -1\}$ and a perfectly separable dataset $\{x_i, y_i\}_{i=1}^n$. Now consider a classifier with weights/parameters $w$ such that predictions are induced by the rule $\hat{y}_i = \mathbf{1}\{w^T x_i > 0\}$. Now we need to choose a loss function, where a reasonable choice is the $0 - 1$ loss (0 if the prediction was correct, 1 otherwise). However, the $0 - 1$ loss is not convex and difficult to optimize. Instead, consider the hinge loss:

$$\ell(w; (x, y)) = \max(0, 1 - y_i w^T x_i)$$

That is, for positive $y = 1$, there is no penalty if $w^T x > 1$ and a penalty of $1 - w^T x$ if $w^T x < 1$. Similarly, for negative $y = -1$, there is no penalty if $w^T x < -1$ and a penalty of $1 + w^T x$ if $w^T x > -1$ . Note how the penalty is a function of the distance from the margin lines $w^T x = 1$ or $w^T x = -1$, depending on the sign of $y$. This is different than the $0 - 1$ loss, which does not take into account how far away from the decision boundary a missclassified point is.
Below in Figure 3, we visualize this setup

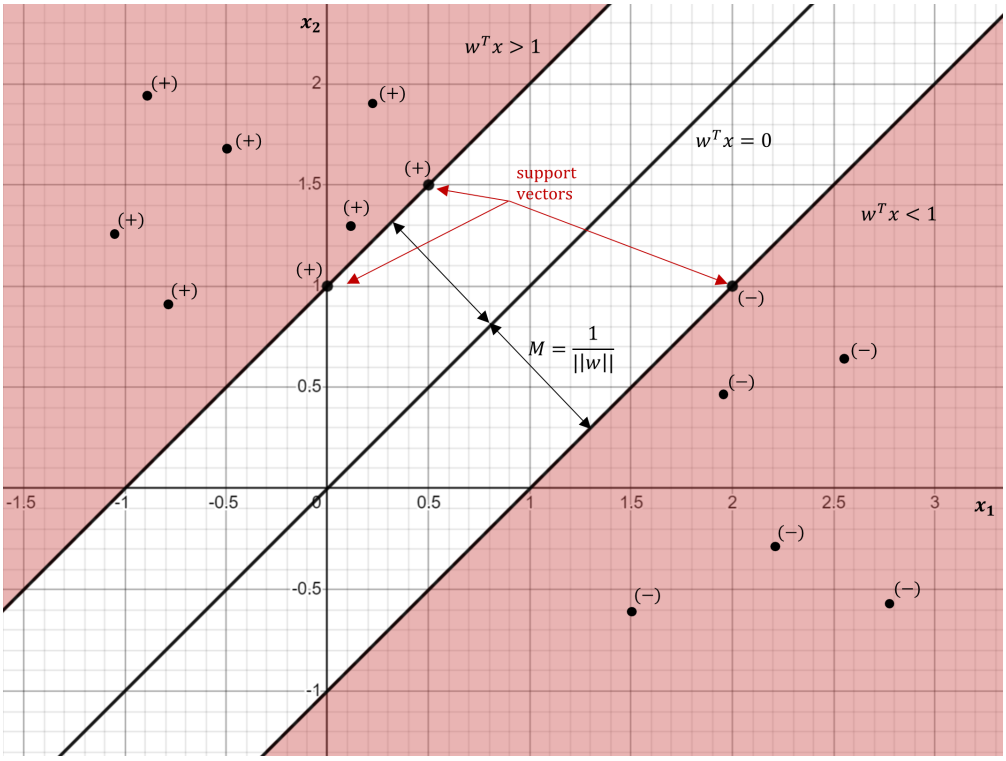


Figure 3: Support vector machine.

If we solve for

$$\begin{aligned}\hat{w} &= \underset{w}{\text{argmin}} \frac{1}{n}\sum_{i=1}^{n} \ell(w;(x_i,y_i)) \\ &= \underset{w}{\text{argmin}} \frac{1}{n}\sum_{i=1}^{n} \max(0, 1-y_i w^T x_i)\end{aligned}$$

on perfectly separable data, we will achieve a total loss of 0. That is, for every datapoint with positive $y_i$, we will have $\hat{w}^T x_i > 1$ and for every datapoint with negative $y_i$, we will have $\hat{w}^T x_i < -1$. In other words, we are solving for separating hyperplanes with maximum margin width $M$ so that every training datapoint is on the correct side of their respective separating hyperplane. The datapoints that end up touching these hyperplanes are called support vectors

To solve for the margin width, $M$, we need to first find the intersection between a vector in the direction of $w$ (which is orthogonal to the line $w^T x = 0$) and $w^T x = 1$. Setting this intersecting vector as $x_{in} = cw$ and evaluating, we have that $cw^T w = 1$ and $c = \frac{1}{||w||^2}$ . Then it easily follows that

$$M = ||x_{in}|| = c||w|| = \frac{1}{||w||}$$

By purely minimizing hinge loss to achieve $\ell = 0$, $M$ is as narrow as necessary so that all training data points are on the correct side of their respective separating hyperplanes. However, we should consider allowing for $\ell \neq 0$ (that is, training points that are on the wrong side of the margin) as to avoid overfitting the training data, and to allow situations were the data is not perfectly separable. Thus, we instead solve for the following, with a penalty $\lambda$ on margins that are too small.

$$\hat{w} = \underset{w}{\text{argmin}} \frac{1}{n}\sum_{i=1}^{n} \ell(w;(x_i,y_i)) + \frac{\lambda}{2}||w||^2$$

In practice, we also include a bias parameter, which shifts the picture up or down by $b$ . That is, we can write the decision boundary as

$$\hat{y_i} = \mathbf{1}\{w^T x_i + b > 0\}$$

and adjust for the hinge loss / separating hyperplanes accordingly.

### 3.3 k-Nearest Neighbors (kNN)

kNN is a simple classification algorithm based on the intuition that similar points are close to each other. The algorithm makes predictions by finding the k-closest points in $X$, and then taking either a majority vote for classification problems or taking the average of the neighbors for regression problems to predict the target $y$. Euclidean distance is the classic choice as the measure of distance. Note that in this model, the number of neighbors $k$ is a model hyperparameter that needs to be tuned.

## 4 Tree Based Methods

### 4.1 Decision Trees (CART)

Tree based methods essentially divide the input feature space into rectangles, and fit simple models (often times a constant) for each partition region. To simplify the description of each rectangular region, we consider partitions that can be created with recursive binary splits. Consider the following example in Figure 4.

Note that the partitioning in Figure 4 could be made using the following steps

1. Split $X_1$ at $t_1$

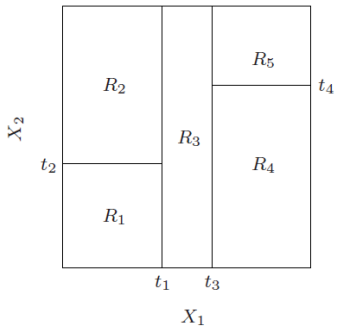


Figure 4: An example of a 2D input space partitioned using recusive binary splits *Image Source: Elements of Statistical Learning*

2. Split the regions $(X_1 \leq t_1)$ at $X_2 = t_2$
3. Split the region $(X_1 > t_1)$ at $X_1 = t_3$
4. Split the region $(X_1 > t_3)$ at $X_2 = t_4$

These recursive splits result in 5 different regions: $R_1, R_2, R_3, R_4, R_5$.

Equivalently, we can represent the splits using a binary tree, with all the data beginning at the top. Then, at each junction node, data satisfying the condition are assigned to the left branch, and the rest of the data are assigned to the right branch. The terminal nodes, which are called leaves, are what define the final resulting regions (see Figure 5).

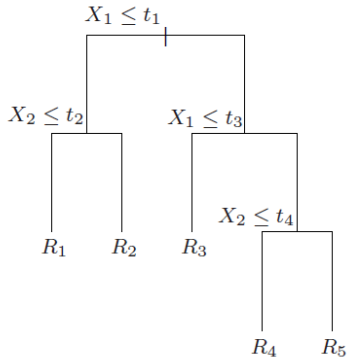


Figure 5: Rectangular partitions represented as a binary tree *Image Source: Elements of Statistical Learning*

If we model each region with a constant, the response surface may look like the following in Figure 6.
The main advantage of CART (classification and regression trees) are their interpretability. The partitions for an entire feature space can be described by a single tree.

#### 4.1.1 Trees in Regression

As usual, consider the data $\{x_i, y_i\}_{i=1}^n$ of $n$ data observations, with the features $\vec{x}$ consisting of p variables $(x_1, ..., x_p)$. Suppose we have M regions, and we model the response in each region with a constant $c_m$:

$$f(x) = \sum_{m=1}^{M} c_m \mathbf{1}\{x \in R_m\}$$

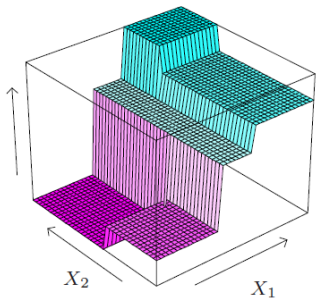


Figure 6: Response surface of a decision tree *Image Source: Elements of Statistical Learning*

$c_m$ can be found such that the sum of squares is minimized.

$$c_m = \underset{c_m}{argmin} \sum_{i \in R_m} (y_i - c_m)^2$$

Setting the derivative to zero we obtain

$$\begin{aligned}\left( \sum_{i \in R_m} (y_i - c_m)^2 \right)' &= \sum_{i \in R_m} \left( (y_i - c_m)^2 \right)' \\ &= \sum_{i \in R_m} 2(y_i - c_m)(-1) \\ &= 0 \\ \rightarrow c_m &= \frac{1}{|R_m|} \sum_{i \in R_m} y_i\end{aligned}$$

Thus, the optimal $c_m$ is simply the average of the points in that region.

$$\hat{c}_m = avg(y_i | x_i \in R_m)$$

Consider the following greedy algorithm. If we split variable $j$ at split point $s$, then the regions defined after the split are

$$\begin{aligned}R_1(j,s) &= \{x | x_j \leq s\} \\ R_2(j,s) &= \{x | x_j > s\}\end{aligned}$$

We seek to find the variable to split $j$ and the split point $s$ by minimizing an impurity criterion, such as the one below. In regression, this impurity criterion has to do with the sample variance in each region. We solve:

$$\underset{j,s}{min} \left[ \sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2 \right]$$

Recall that $\hat{c}_1$ and $\hat{c}_2$ are the sample averages of the points belong to that region. It turns out, for each splitting variable, the split point can be solved quickly. Thus, we scan though all potential variables and find the best pair $(j, s)$ that minimize the impurity measure the most. Then this process is repeated iteratively for both of the resulting regions to continue growing the tree.

When should we stop growing a tree? A tree that is too large will overfit the data. To answer this question, consider cost-complexity pruning. Essentially, the strategy is to grow a large tree ($T_0$) and prune afterwards to achieve the bias-variance tradeoff.

Let $T$ be a subtree of $T_0$, $T \subset T_0$. Denote $|T|$ as the number of terminal leaf nodes in $T$, indexed by $m$. Let $N_m$ be the number of data points in regions $R_m$, and let $\hat{c}_m$ be the average of all data points in $R_m$, as before. Define the square error impurity for the $mth$ region, which we can interpret as a measure of unexplained variance in the model.

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$$

Then we can define the cost-complexity criterion:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|$$

The first term is a measure of cost, and the second term penalizes model complexity - by balance the two, we control the trade-off between goodness of fit and tree size. $\alpha$ is a hyperparameter that needs to be tuned. Overall, out of all candidate subtrees, we want to choose the subtree with the lowest $C_\alpha(T)$.

Let $T_\alpha$ be the optimal subtree. To find $T_\alpha$, the strategy is to start from the full tree $T_0$ and successively collapse the internal node (i.e. non-leaf node) that produces the smallest per-node increase in $\sum_m N_m Q_m(T)$, the total impurity, until we are only left with one node. This obtains a finite sequence of subtrees which contains $T_\alpha$. This strategy is called weakest-link pruning.

To select the value of $\hat{\alpha}$, we can employ k-fold cross-validation and choose $\alpha$ that minimzes the cross-validated sum of squares.

#### 4.1.2 Trees in Classification

Classification trees are very similar to regression trees except the square error impurity does not make sense. We need a new measure of $Q_m(T)$ to grow and prune trees.

Consider a classification problem with $k = 1, ..., K$ different outcomes. Let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} \mathbf{1}\{y_i = k\}$$

That is, the proportion of observations belonging to class $k$ in leaf node/region $m$. We classify observations in node $m$ as $k(m) = \underset{k}{argmax}\hat{p}_{mk}$, or in other words, classifications for each region are made based on the majority class.

Different choices for node impurity $Q_m(T)$ are delineated below.

Missclassification error:

$$\frac{1}{N_m} \sum_{x_i \in R_m} \mathbf{1}\{y_i \neq k(m)\} = 1 - \hat{p}_{mk(m)}$$

Gini index:

$$\sum_{k \neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Cross-entropy/deviance:

$$-\sum_{k=1}^{K} \hat{p}_{mk} log \hat{p}_{mk}$$

It is common to grow a tree using cross-entropy or the Gini index, and then prune using the missclassification error.

To gain intuition on the Gini index, consider classifying an observation to class $k$ with probability with probability $\hat{p}_{mk}$. Recall that $\hat{p}_{mk}$ represents the proportion of class k in node m. Then the probability of missclassification is, by the Law of Total Probability, is the Gini index.

$$\begin{aligned}\mathbb{P}\{missclassification\} &= \sum_{k=1}^{K} \mathbb{P}\{missclassify \mid class\ k\}\mathbb{P}\{class\ k\} \\ &= \sum_{k=1}^{K} (1-\hat{p}_{mk})(\hat{p}_{mk})\end{aligned}$$

Below I delineate several drawbacks and advantages of decision trees, starting with the drawbacks.

Drawbacks:

1. *Instability / high variance:* A small change in the data can result in a very different series of splits as errors from higher splits are propagated down the entire tree. Consider bagging or a random forrest to reduce the varaince.
2. *Lack of smoothness:* Performance may suffer particularly for regression problems with smooth surfaces.
3. *Difficulty modeling an additive structure:* Suppose we have the model $y = c_1\mathbf{1}\{x_1 < t_1\} + c_2\mathbf{1}\{x_2 < t_2\} + \epsilon$. A decision tree might make a split at $x_1 = t_1$, but it will have difficulty splitting *both* the resulting nodes at $x_2 = t_2$

Advantages:

1. *Highly interpretable:* the entire input space is clearly mapped. Note this level of interpretability is not true for methods built on top of CART such as random forest or gradient boosting - these methods change the structure of the model.
2. *Fast to construct*
3. *Handles mixed data*: CART supports both categorical and numerical variables
4. *Invariant to transformations of individual predictors*
5. *Performs internal feature selection:* features are selected automatically in the splitting process.
6. *Robust to outliers*

## 4.2 Random Forest

A random forest is a modified bagging technique so that many different decision trees can be averaged to reduce the overall variance of the model. Bagging is a strategy that averages many high-variance, low-bias models. The variance of the sample mean of $B$ i.i.d random variables is $\sigma^2/B$. However, for $B$ such random variables that are not independent and have positive pairwise correlation $\rho$, then the variance of the sample mean is

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

As $B \to \infty$, we are only left with the first term. The idea of a random forest is to average many *decorrelated* trees by selecting $m \leq p$ input features at each split, decreasing $\rho$ between trees. The algorithm is as follows To make a prediction on a new observation $x$, we return the average of the forest for regression and the majority vote for classification problems. Specifically, in regression, a prediction is made using

$$\hat{f}_{rf}^{B} = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$$

**Algorithm 1** Random Forest

**for** $b = 1, ..., B$ **do**

- Draw a bootstrap sample $Z^*$ of size n from the training data
- Grow a random-forest tree $T_b$ using $Z^*$ by recursively repeating the following steps for each terminal leaf node of the tree, until the minimum node size $n_{min}$ is reached.
  - Select $m$ variables at random from the $p$ variables
  - Identify the best variable and split-point from the $m$ variables
  - Split the node into two daughter nodes

**end for**
**return ensemble of trees** $\{T_b\}_{b=1}^B$

In classification, we use the following. Let $\hat{C}_b(x)$ be the class prediction of the $b$th random-forest tree. Then the predicted class for $x$ is

$$\hat{C}_{rf}^B = majority\ vote\ \{\hat{C}_b(x)\}_{b=1}^B$$

The bias-variance tradeoff in a random forest is controlled by $m$. Choosing a smaller $m$ intuively reduces the correlation between trees, and thus reduces the variance. However, choosing a $m$ to be too small introduces bias into the model. Overall, random forests perform very well, are easy to train, not prone to overfit, and have decent interpretability.

#### 4.2.1 Out-of-Bag (OOB) Samples

For each observation $z_i = (x_i, y_i)$, we can construct its RF predictor by averaging trees trained on bootstrap samples *that did not include* $z_i$. Note that this is very similar to the idea behind k-fold corss-validation. Hyperparameters, such as the number of forest trees, can be chosen when OOB error stabilizes.

#### 4.2.2 Feature Importance

It is possible to create feature importance plots with random forests. One strategy is to keep track of how much a variable decreases impurity (Gini index, sample variance, etc...) at its respective splits and accumulate over all the trees in the forest for each variable separately. The advantage is this is very computationally easy, but tends to over-inflate the importance of continuous variables or high cardinality categorical variables. Additionally, for two variables that are highly co-linear, the random forest can only give one of the variables a higher split, and for subsequent splits the other variable will be considered less important as information is already contained in the first variable - where it is very much possible that the two variables are equally important.

Another strategy is to use OOB samples in what is known as permutation feature importance. After growing the $b$th tree, we can calculate the prediction accuracy for its respective OOB samples. Then, for the $j$th variable, we can randomly permutate its values among the OOB samples and measure the resulting decrease in prediction accuracy. Intuitively, the larger the decrease in prediction accuracy, the more important variable $j$ was. We then average this result for all trees in the forest.

Overall, feature importance in random forests take advantage of its self-feature selecting process to inform further downstream feature selection and for overall interpretability.

# 5 Ensemble Methods

## 5.1 Bagging

In bootstrap aggregation or bagging, we fit several models based on bootstrap samples from the data and average their results. By doing this, bagging can dramatically reduce the variance of many high variance

models, like decision trees for example. Roughly speaking, these methods train models in parallel to reduce the variance whiling leaving bias largely unchanged. For example, if we were to bag decision trees in regression, we could simply average the results of all the individual trees for each $x$. For classification, we could either average the estimates of class probabilities, or simply take the majority-vote for the predictions of each tree.

Consider the training data $Z = \{x_i, y_i\}_{i=1}^n$. Let $\{Z^{*b}\}_{b=1}^B$ represent B different bootstrap samples of the data, with corresponding predictions on the bootstrap samples $\hat{f}^{*b}(x)$ for a given $x$. Then in regression, we could write the bagged estimate as

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

This expression is an estimate of the true bagging estimate, which we can express as $\mathbb{E}_{\hat{\mathcal{P}}}\hat{f}^*(x)$, where $\hat{\mathcal{P}}$ is an empirical distribution weighting each of the data samples $(x_i, y_i)$ with equal probability $\frac{1}{n}$. Then we can write the bootstrap sample that $\hat{f}^*(x)$ is trained on as $Z^* = \{x_i^*, y_i^*\}_{i=1}^n$ where each $(x_i^*, y_i^*)$ is drawn from the data $Z$ based on $\hat{\mathcal{P}}$. As $B \to \infty$, the estimate approaches the true expectation by the law of large numbers.

In classification, we can consider a majority vote.

$$\hat{f}_{bag}(x) = \underset{k}{\text{argmax}}\{p_1(x), ..., p_k(x)\}$$

where $\{p_1(x), ..., p_k(x)\}$ represents the proportion of trees predicting each of the $1, ..., k$ classes. Note that these proportions are *not the class probability estimates*. For example, in a 0-1 binary classification where the class probability of predicting 1 is 0.75. Then if we had two trees that we bag, and both trees correctly predict 1, then $p_1(x) = 1$ which is different than the true class probability.

To give intuition on the benefits of bagging in regression, we can consider true population aggregation. Instead of drawing from the empirical distribution $\hat{\mathcal{P}}$, consider drawing from the true distribution $\mathcal{P}$. Then the ideal estimator is $f_{ag}(x) = \mathbb{E}_{\mathcal{P}}\hat{f}^*(x)$ where $\hat{f}^*(x)$ is trained on the bootstrap dataset $Z^*$ drawn from the true distribution $\mathcal{P}$. Consider:

$$\begin{aligned}
\mathbb{E}_{\mathcal{P}}[(y - \hat{f}^*(x))^2] &= \mathbb{E}_{\mathcal{P}}[(y - f_{ag}(x) + f_{ag}(x) - \hat{f}^*(x))^2] \\
&= \mathbb{E}_{\mathcal{P}}[(y - f_{ag}(x))^2] + \mathbb{E}_{\mathcal{P}}[(\hat{f}^*(x) - f_{ag}(x))^2] \\
&\geq \mathbb{E}_{\mathcal{P}}[(y - f_{ag}(x))^2]
\end{aligned}$$

Thus, true population aggregation *never increases mean squared error*. Specifically, this suggests that bagging will often times decrease the overall mean squared error. This argument, however, does not hold under classification.

In classification, bagging a poor classifier will create a worse classifier, but bagging a good classifier can make it better. To understand bagging under classification, we consider what is called as the Wisdom of Crowds, or the idea that the consensus of independent weak learners is stronger than any of the models alone. Consider a binary classification case, for example. Let $\hat{f}_b^*(x)$ be one of the independent $b = 1, ..., B$ bagged models, and assume we have an error rate less than 0.5 for any individual model, $e < 0.5$. Consider $y = 1$ at the given $x$. The consensus vote for class 1 can be written

$$S_1(x) = \sum_{b=1}^{B} 1\{\hat{f}_b^*(x) = 1\}$$

Note that since we assumed the models are independent, we have

$$S_1(x) \sim Binomial(B, 1 - e)$$

Finally, note that the probability of getting the correct answer of $y = 1$ (in other words, $S_1 > B/2$) goes to 1 as $B$ grows. In contrast, the upper-bound for the error of a single predictor is 0.5. Through this

demonstration, we can see that if we started with a poor model with $e > 0.5$, the probability of getting the desired answer does not increase but actually decreases. Another important assumption made here is the independence assumption between models - which is not true for bagged trees. However, the Random Forest improves on this issue by building decorrelated trees. That is, not only are individual trees trained on a bootstrap sample of *data*, but also a new set of *features* are drawn for each tree.

## 5.2 Stacking

Stacking refers to the process of combining the strengths of several different models. For example, after training several classifiers on the training data (called level 0 predictors), we can further combine the predictions using a logistic regression model (called a level 1 predictor). The final model can then be together evaluated using k-fold cross validation.

If we take a Bayesian interpretation of model stacking, we can view the posterior mean as a weighted average of level 0 predictions. Suppose that the prediction $f(x)$ is our quantity of interest, considering some fixed feature variable $x$. Let $Z$ represent out training data set $Z$. Let the level 0 models be $\mathcal{M}_m$ indexed by $m = 1, ..., M$. Then the posterior distribution of $f(x)$ is

$$P(f(x)|Z) = \sum_{m=1}^{M} P(f(x)|\mathcal{M}_m, Z)P(\mathcal{M}_m|Z)$$

taking the posterior mean, we have

$$\mathbb{E}(f(x)|Z) = \sum_{m=1}^{M} \mathbb{E}(f(x)|\mathcal{M}_m, Z)P(\mathcal{M}_m|Z)$$

Thus, the posterior mean (which presents our final prediction), is a weighted average of each individual prediction.

In the Frequentist viewpoint, given the predictions $\hat{f}_1(x), ..., \hat{f}_M(x)$ again considering a fixed $x$, and let the dataset $Z$ be distributed according to $\mathcal{P}$. Under square-error loss, we are searching for weights such that

$$\hat{w} = \underset{w}{argmin}\mathbb{E}_{\mathcal{P}}\left[\left(Y - \sum_{m=1}^{M} w_m \hat{f}_m(x)\right)^2\right]$$

At the population level, this is simply a linear regression of $Y$ on $\{\hat{f}_1(x), ..., \hat{f}_M(x)\}$. Note that while we discuss linear regression here, any other learning algorithm can be used to combine the level 0 models. In practice, stacking models typically perform better than any one single model.

## 5.3 Boosting

### 5.3.1 Gradient Boosting

In gradient boosting, we want to sequentially fit models (so called "weak learners") so that each model improves on the previous model's mistakes. This motivation aligns with the general framework of boosting. Here, I outline a simple example to highlight the intuition of gradient boosting. For example, consider $f_m(x_i)$, the current model at the mth iteration. To improve on this model, we could add another model $h_m(x_i)$ as follows to obtain the m+1 model iteration:

$$f_{m+1}(x_i) = f_m(x_i) + h_m(x_i) = y_i$$

Equivalently, we can write $h_m(x_i) = y_i - f_m(x_i)$. In other words, we want to fit $h_m(x_i)$ to the *residuals* of the current model in order to calculate the next update.

Further, it has been shown that this sequential process has a gradient descent interpretation. For some easy intuition, consider the MSE loss function and its derivative w.r.t the function value $f_m(x_i)$. Here, we see that the gradient of the MSE loss is *proportional to the residual*.

$$L(y_i, f_m(x_i)) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_m(x_i))^2$$
$$-\frac{\partial L}{\partial f_m(x_i)} = \frac{2}{n}(y_i - f_m(x_i)) = constant * (y_i - f_m(x_i))$$

Additionally, a "gradient descent update" to $f$ that minimizes MSE on the training data could look like the following:

$$f_{m+1}(x_i) = f_m(x_i) - constant * \frac{\partial L}{\partial f_m(x_i)}$$

The problem with this example, however, is that we are ultimately interested in fitting a model or function which we can use to predict $y$ using a never-seen-before $x$, and not just minimize training error. Yet the loss and the gradient above is only defined for observed values in the training data $\{x_i, y_i\}_{i=1}^n$ and is unknown for unseen data. In other words, the gradient above depends on $y$, yet for unseen data this is not observed. A solution to this problem is to train a weak learner using the training data, denoted by $h_m(x)$, that models the (negative) gradient as well as possible. Then we could write the "gradient descent update" in a general function form:

$$f_{m+1}(x) = f_m(x) + constant * h_m(x_i)$$

Formalizing the above example, we note that in gradient boosting, we are seeking a sequence of $m = 1, ...., M$ weak learners so that the final approximating function is a weighted sum of the sequence. That is,

$$\hat{f}(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

This can be solved in a "forward stage-wise" manner. That is, to solve for iteration m, we optimize for the next additive function and fix the previous m-1 function values. This process is initialized by fitting a constant (denoted $\gamma$ below). Consider a class of weak learners $\mathcal{H}$ and some arbitrary loss function $L$. Then we can write for $m = 1, ..., M$:

$$f_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$$
$$f_m(x) = f_{m-1}(x) + \left( \underset{h_m \in \mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} L\left(y_i, f_{m-1}(x_i) + h_m(x_i)\right) \right)(x)$$

Unfortunately, this problem as stated above is computationally infeasible. Instead, we can consider the "steepest descent" approach with a step size of $\gamma_m$ (not to be confused with the initial constant function $\gamma$).

$$f_m(x) = f_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_{f_{m-1}(x_i)} L(y_i, f_{m-1}(x_i))$$

where $\gamma_m$ can be found using a line search

$$\gamma_m = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x) - \gamma \nabla_{f_{m-1}} L(y_i, f_{m-1}(x_i)))$$

As alluded to, the problem with this approach is that the gradients are only feasible to compute on the observed training data, while our goal is to generalize to unseen data. That is, for unseen $X, Y$, we cannot evaluate the function $f_m(x)$. To resolve this issue, we fit a weak learner $h_m(x)$, a decision tree for example, to the easily computed (negative) derivatives over the training data, or the so called "pseudo-residuals." Finally,

we use the learned function $h_m(x)$ to compute the next model update, which can be evaluated for unseen data.

The pseudo residuals $r_{im}$ for the ith training data observation and the mth iteration can be written:

$$r_{im} = -\left(\frac{\partial L(y_i, f_{m-1}(x_i)}{\partial f_{m-1}(x_i)}\right)$$

Intuitively, these pseudo residuals represent the error of the model at the current iteration, and fitting a weak learner to them allows us to estimate the error at any input $x$ beyond the training data.

We can write a general gradient boosting algorithm as follows.

**Algorithm 2** General gradient boosting

$f_0(x) \leftarrow \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$

**for** $m = 1, ..., M$ **do**

- Compute the pseudo-residuals:

$$r_{im} = -\left(\frac{\partial L(y_i, f_{m-1}(x_i)}{\partial f_{m-1}(x_i)}\right)$$

  for all $i = 1, ..., n$
- Fit a weak learner $h_m(x)$ to the pseudo-residuals using the training set $\{x_i, r_{im}\}_{i=1}^{n}$
- Compute $\gamma_m$ using line search:

$$\gamma_m = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i))$$

- Perform the update step:

$$f_m(x) = f_{m-1}(x) + \gamma_m h_m(x)$$

**end for**
**return** $f_M(x)$

#### 5.3.2 Gradient Boosted Trees

The most common class of "weak learners" used in gradient boosting is the Decision Tree, hence the name. For the mth step, suppose that the decision tree $h_m(x)$ has $J_m$ leaf nodes. That is, $h_m(x)$ divides the input space into $J_m$ different regions, which we can denote as $R_{1m}, ..., R_{Jm}$, and models each region with a constant. We can write the decision tree as

$$h_m(x) = \sum_{j=1}^{J_m} b_{jm} \mathbf{1}\{x \in R_{jm}\}$$

Then the update equations directly follow as

$$f_m(x) = f_{m-1}(x) + \gamma_m h_m(x)$$
$$\gamma_m = \underset{\gamma}{\text{argmin}} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \gamma h_m(x_i))$$

Alternatively, instead of using $\gamma_m$ for the entire tree, we can optimize for a constant $\gamma_{mj}$ for *each tree region*, and $b_{jm}$ can be discarded. In other words, solving the line search for $\gamma_{jm}$ could look like

$$\gamma_{jm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

with the update equation

$$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{mj} \mathbf{1}\{x \in R_{jm}\}$$

#### 5.3.3 Regularization

The two obvious hyperparameters that control the bias variance tradeoff are the size of the trees $J$ and the number of boosting rounds $M$. If we set $J = 2$ (i.e. only one split), we do not allow for the interaction between terms. $J = 3$ allows for two-variable interactions and so on and so forth. In practice, choosing $J$ between 4 and 8 yields good results, and $J > 10$ is seldom needed. Note that setting $J$ to be too large will incur high variance. The best strategy is to try several different $J$ and choose the one with the highest performance on a hold-out validation set.

For $M$, note that each boosting iteration reduces the loss on the training set. If $M$ is too large, we can overfit the training data and do poorly on the test data. A convenient way to choose an optimal $M$ is to monitor the performance on a validation set and to choose $M$ where the performance is best, similar to early stopping in neural networks.

Another strategy is called shrinkage, where we control the "learning rate" of the boosting step. For example, let $\nu$ be a constant between 0 and 1. Then the update equation can be written as

$$f_m(x) = f_{m-1}(x) + \nu \gamma_m h_m(x)$$

Note that there is a tradeoff between $\nu$ and $M$, that is, for the same training error smaller $\nu$ corresponds to larger $M$. However, empirically, it appears that the best strategy is to set a small $\nu < 0.1$ and then control $M$ using early stopping.

Finally, similar to the idea of bootstrap averaging to reduce variance, we can sample a fraction $\eta$ of the training data without replacement. This is known as sub-sampling, and is related to *stochastic gradient boosting*. Sub-sampling improves the performance of the model and improves the computational time required for training. A typical choice is $\eta = 0.5$ but even smaller choices are possible for large $n$.

# 6 Neural Networks and Deep Learning

## 6.1 The Feedforward Neural Network

Neural networks are nonlinear statistical models used in both regression and classification. Below we discuss a model with one "hidden layer," and then extend the result to obtain a deep feedforward network with mutiple hidden layers, known as the multi-layer perceptron (MLP).
Consider the model in Figure 7. The intuition for the hidden layer is that the model learns to represent the data in a higher-dimensional space, where predictions could potentially be made easier. This is accomplished through one or multiple non-linear transformations of the input features.

In a neural network, data is processed sequentially.

1. A nonlinear function is applied to linear combinations of the input data, $x \in \mathbb{R}^p$. This nonlinear function, denoted $f$, is known as the activation function. The activation function is responsible for introducing non-linearity into the model; historically a common choice was the sigmoid function ($f(v) = \frac{1}{1+e^{-v}}$) but in modern settings the ReLU function, or rectified-linear unit ($f(v) = max\{0, v\}$), is the default option.

   $$z_m = f(\alpha_{0m} + \alpha_m^T x),\ m = 1, ..., M$$

   Here, $z_m$ for $m = 1, ..., M$ are the hidden units. We can think of them as derived features that serve as a basis expansion of the original $x_1, ..., x_p$ input features. In matrix notation, we can collect all $M$

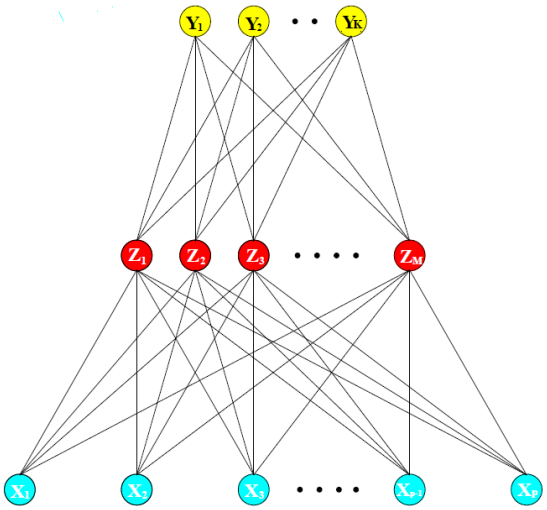


Figure 7: A schematic of a single layer neural network *Image Source: Elements of Statistical Learning*

hidden units as $z = (z_1, ..., z_M) \in \mathbb{R}^M$, and defining the weight matrix $A \in \mathbb{R}^{M,p}$ having $\alpha_1^T, ..., \alpha_M^T$ as its rows and the intercept term $a_0 = (\alpha_{01}, ..., \alpha_{0M}) \in \mathbb{R}^M$ we have

$$z = f(Ax + a_0)$$

Sometimes, the intercept term $a_0$ is called the *bias* term.

2. We continue to apply a linear transformation to the hidden units, and again apply a nonlinear transformation $f_{out}$ to obtain the output. Not that $f_{out}$ for the final layer is called the output function. For a regression problem, the output function can either be the identity function (called a linear activation), or another activation function. For a classification problem, we want the output function to be the softmax function, so that each of the $k = 1, ..., K$ classes can be assigned a predicted probability.

$$\hat{y}_k = f_{out}(\beta_{0k} + \beta_k^T z), \ k = 1, ..., K$$

Setting weight matrix $B \in \mathbb{R}^{K,M}$ with $\beta_1^T, ..., \beta_K^T$ as its rows, and intercept term $b_0 = (\beta_{01}, ..., \beta_{0K})$ we can represent each output $\hat{y} = (\hat{y}_1, ..., \hat{y}_K) \in \mathbb{R}^K$ in matrix notation as

$$\hat{y} = f_{out}(Bz + b_0)$$

This procedure can easily be extended to an arbitrary number of hidden layers to derive higher level and more complex hidden representations. For example, using the subscript $(i)$ to denote the $ith$ hidden layer, we have

$$\begin{aligned}
Hidden\ layer\ 1: &\ z_{m(1)} = f(\alpha_{0m(1)} + \alpha_{m(1)}^T x), \ m = 1, ..., M_1 \\
Hidden\ layer\ 2: &\ z_{m(2)} = f(\alpha_{0m(2)} + \alpha_{m(2)}^T z_{m(1)}), \ m = 1, ..., M_2 \\
&... \\
Hidden\ layer\ j: &\ z_{m(j)} = f(\alpha_{0m(j)} + \alpha_{m(j)}^T z_{m(j-1)}), \ m = 1, ..., M_j \\
Output\ layer: &\ \hat{y}_k = f_{out}(\beta_{0k} + \beta_k^T z_{m(j)}), \ k = 1, ..., K
\end{aligned}$$

In matrix form, defining the weight matrices/bias terms similarly as above

$$\begin{aligned} Hidden\ layer\ 1: &\ z_{(1)} = f(a_{0(1)} + A_{(1)}x) \in \mathbb{R}^{M_1} \\ Hidden\ layer\ 2: &\ z_{(2)} = f(a_{0(2)} + A_{(2)}z_{(1)}) \in \mathbb{R}^{M_2} \\ &... \\ Hidden\ layer\ j: &\ z_{(j)} = f(a_{0(j)} + A_{(j)}z_{(j-1)}) \in \mathbb{R}^{M_j} \\ Output\ layer: &\ \hat{y} = f_{out}(b_0 + Bz_{(j)}) \in \mathbb{R}^K \end{aligned}$$

Part of what makes neural networks so powerful is the Universal Approximation Theorem. That is, there exists a neural network that can approximate, to any desired degree of accuracy, any continuous function (on a closed and bounded subset of $\mathbb{R}^n$). This is as long as we use a "squashing" activation function such as the sigmoid or ReLU. Note that while a neural network with sufficient representation exists in theory, it is not guaranteed that we obtain such a network in training.

## 6.2 Training Neural Networks

### 6.2.1 Backpropagation

Let $\theta$ be the complete set of weights/parameters, which include for one layer

$$\{\alpha_{0m}, \vec{\alpha}_m, \beta_{0k}, \vec{\beta}_k;\ m = 1, ..., M,\ k = 1, .., K\}$$

where $M$ is the number of hidden units, and $K$ is the output dimension.
For regression problems, consider minimizing square error loss:

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \hat{y}_{ik}(x_i))^2$$

For classification problems, consider minimizing cross entropy loss:

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} log(\hat{y}_{ik}(x_i))$$

where $y_i$ represents the true label (as a one-hot vector) for the $ith$ data observation and $\hat{y}_i$ represents the softmax probability distribution which assigns a probability to each classification class. Note that we are not looking for the global minimum for these objectives, since these solutions are usually overfit. Most of the time, a local solution will strike a good bias-variance tradeoff. Additionally, gradient-based optimization algorithms do not find global solutions for non-convex objectives.

In order to use gradient descent to minimize loss, we need to take derivatives of the loss function with respect to the parameters. For example, in the regression case we have

$$R_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - \hat{y}_{ik}(x_i))^2 = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(y_{ik} - f_k(\beta_{0k} + \beta_k^T z_i))^2$$

Then taking derivative with respect to the parameters $\beta_{km}$, the $mth$ unit of the parameter vector $\beta_k$, and $\alpha_{m\ell}$, the $\ell th$ unit of the parameter vector $\alpha_m$.

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(\beta_{0k} + \beta_k^T z_i))\frac{\partial f_k}{\partial(\beta_{0k} + \beta_k^T z_i)}\frac{\partial(\beta_{0k} + \beta_k^T z_i)}{\partial \beta_{km}}$$

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(\beta_{0k} + \beta_k^T z_i))\frac{\partial f_k}{\partial(\beta_{0k} + \beta_k^T z_i)}\frac{\partial(\beta_{0k} + \beta_k^T z_i)}{\partial z_m}\frac{\partial z_m}{\partial(\alpha_{0m} + \alpha_m^T x_i)}\frac{\partial(\alpha_{0m} + \alpha_m^T x_i)}{\partial \alpha_{m\ell}}$$

We can rewrite this result as

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}$$
$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = \delta_{mi} x_{i\ell}$$

for $i = 1, .., n$ observations, $m = 1, ..., M$ hidden units, $k = 1, ..., K$ output units, and $\ell = 1, ..., p$ input features. Here $\delta_{ki}$ and $\delta_{mi}$ are called "errors" and they represent the upstream gradient.
Notice that central to the gradient calculations is the use of the chain rule to calculate derivatives with respect to a model parameter after each function evaluation, but in reverse order. This is the essence of the backpropagation algorithm, the name of the algorithm used to train neural networks. We pass gradient information backwards using the chain rule to each model parameter so that they can be updated with a gradient based optimization algorithm such as gradient descent.

Specifically, backpropagation consists of two steps. Here we let the superscript $(t)$ denote the $t^{th}$ iteration of parameter updates.

1. The Forward Pass: Fix the current weights $\theta^{(t)}$ and find the predicted values $\hat{y}^{(t)}(x_i)$. Calculate the loss at the current iteration.
2. The Backward Pass: Use the chain rule to find the errors $\delta_{ki}$ and $\delta_{mi}$. Update the weights using a gradient descent scheme.

For example, one step of a gradient descent update with step-size $\eta_t$ could look like

$$\beta_{km}^{(t+1)} = \beta_{km}^{(t)} - \eta_t \frac{\partial R_n}{\partial \beta_{km}^{(t)}} = \beta_{km}^{(t)} - \eta_t \frac{1}{n} \sum_{i=1}^{n} \frac{\partial R_i}{\partial \beta_{km}^{(t)}}$$

$$\alpha_{m\ell}^{(t+1)} = \alpha_{m\ell}^{(t)} - \eta_t \frac{\partial R_n}{\partial \alpha_{m\ell}^{(t)}} = \alpha_{m\ell}^{(t)} - \eta_t \frac{1}{n} \sum_{i=1}^{n} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(t)}}$$

Collecting all such updates, we write

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta R_n = \theta^{(t)} - \eta_t \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta R_i$$

However, the problem with this approach is that each gradient descent step requires n gradient calculations, one for each data observation. We will find that with Stochastic Gradient descent, it is possible to obtain good parameter updates with gradients calculated with only one or several data points at a time.

### 6.2.2 Stochastic Gradient Descent

Here, we take an aside to discuss Stochastic Gradient Descent, a useful optimization algorithm when standard Gradient Descent is slow due to large data sizes. Recall the goal of machine learning is to solve for the minimizer of Risk

$$R(\theta) = \mathbb{E}\ell(\theta, X, Y)$$
$$\tilde{\theta} = \underset{\theta}{\text{argmin}} R(\theta)$$

Here, $\ell$ is the loss function, and $X$ and $Y$ and random variables for the features and targets, respectively. Note, the true risk cannot be evaluated since the probability distribution of the data $X$ and $Y$ is often times unknown. Instead, in focus on minimizing true risk, we usually minimize empirical risk (estimating the expectation with n iid draws of data),

$$\hat{\theta} = \underset{\theta}{\text{argmin}} R_n(\theta) = \underset{\theta}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(\theta; x_i, y_i)$$

which gives us a deterministic optimization problem since now we are only dealing with the observed data $\{x_i, y_i\}_{i=1}^n$. The problem is, the gradient descent solution requires $n$ gradient calculations at every step, which can be slow for large $n$.

Instead, in stochastic gradient descent, we try to estimate the expectation of the gradient of the random loss directly instead of the expectation of the random loss itself. That is, we notice that

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathbb{E}\ell(\theta, X, Y) = \theta^{(t)} - \eta_t \mathbb{E}\left(\nabla_\theta \ell(\theta, X, Y)\right)$$

where $\nabla_\theta \ell(\theta, X, Y)$ is the "stochastic gradient."

We then estimate the expectation of the stochastic gradient with $m$ random draws of the data, where $m << n$. Practically, $m$ data samples can either be bootstrapped for the pool of $n$ observed data points, or drawn without replacement. This is called mini-batch stochastic gradient descent.

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{1}{m} \sum_{j=1}^{m} \nabla_\theta \ell(\theta, x_j, y_j)$$

Note how for each update step, we now only need $m$ gradient calculations instead of $n$. Further, in most practical applications, only one sample is sufficient enough to estimate the stochastic gradient and obtain a good solution. That is

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \ell(\theta, x_j, y_j)$$

which only requires one gradient calculation per update step.

It can be shown that standard gradient descent has convergence guarantee (i.e., $||\theta^{(t)} - \theta^*|| \leq \epsilon$ where $\epsilon$ is an arbitrarily small error value and $\theta^*$ is the true optimum) on the order of $t \sim O(nlog(\frac{1}{\epsilon}))$ while stochastic gradient descent has a convergence rate of $t \sim O(\frac{1}{\epsilon})$. As a technical detail, these guarantees are for strongly convex, gradient smooth optimization problems.

So, in the regime where $O(\frac{1}{\epsilon}) < O(nlog(\frac{1}{\epsilon}))$, which is true for large $n$, stochastic gradient descent performs better.

### 6.2.3 Computational Graphs

In many deep learning frameworks, gradients are calculated automatically using computational graphs, which represent the mathematical operations of a model.
For example, if we are calculating $y = (a + b)(b - c)$ then we would have the computational graph as shown in Figure 8.
Computational graphs can then be followed backwards to compute the gradients. For example, suppose we are interested in $\frac{\partial y}{\partial a}$. We can follow the computational graph to obtain the following (see Figure 9).

$$\frac{\partial y}{\partial a} = \frac{\partial y}{\partial d}\frac{\partial d}{\partial a} = e * 1 = e$$

As an aside, note that if there are contributions to the gradient at multiple sources, these contributions are summed. That is, gradients are summed at outward branches. For example, if we were interested in $\frac{\partial y}{\partial b}$ then we would need to sum

$$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial d}\frac{\partial d}{\partial b} + \frac{\partial y}{\partial e}\frac{\partial e}{\partial b}$$

Each node receives an "upstream gradient," and the goal is to pass on the correct "downstream" gradients by multiplying the upstream gradient with the corresponding "local gradient" (see Figure 10). For example, if we were calculating $z = Wx$ at a node and we are interested in the gradient of some upstream $s$, then we would have $\frac{\partial s}{\partial W} = \frac{\partial s}{\partial z}\frac{\partial z}{\partial W}$ where $\frac{\partial s}{\partial z}$ is the upstream gradient, $\frac{\partial z}{\partial W}$ is the local gradient, and $\frac{\partial s}{\partial W}$ is the desired downstream gradient. Similarly, $\frac{\partial s}{\partial x} = \frac{\partial s}{\partial z}\frac{\partial z}{\partial x}$.

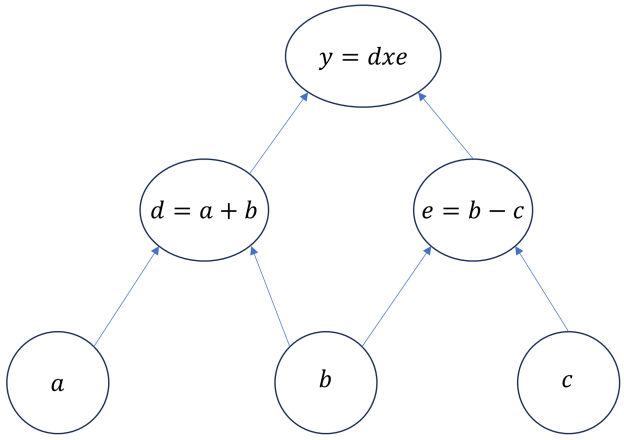


Figure 8: An example of a computational graph

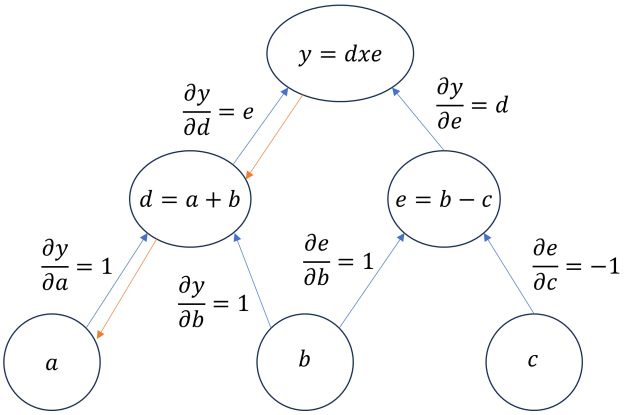


Figure 9: Gradient calculation with a computational graph

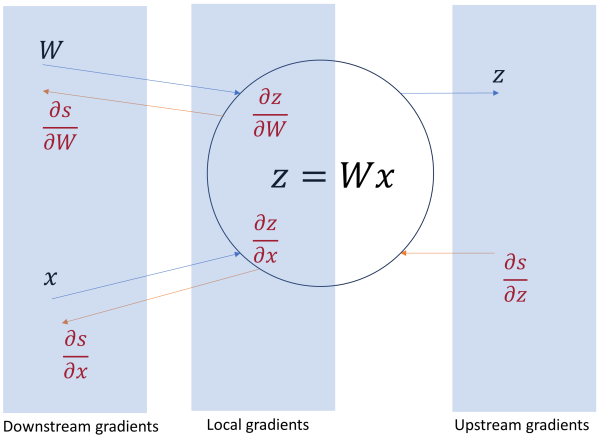


Figure 10: Backpropagation at a single node

### 6.3 Regularization

Two common ways to regularize deep neural networks is early stopping and dropout. In early stopping, the model is trained on a train dataset and losses are monitored on a separate validation dataset at every training epoch. Here, loss on the validation dataset is a proxy for generalization error. When validation loss does not improve for a set number of epochs (called the *patience*), training is stopped.

Dropout is a regularization technique that prevents feature co-adaptation, that is, features cannot only be useful when used in conjunction with particular other features. Another view is that dropout allows us to train many different "sub-models" and is a form of bagging that achieves a similar effect to a random forest. To implement dropout, we randomly set inputs to each neuron to 0 using a set fixed probability $p$ in training i.e. we randomly turn some connections in the neural network off during training. During prediction, we do not turn off model connections but instead scale all of them by $p$.

The classic view of regularization is striking a balance in the bias-variance tradeoff by preventing overfitting. However, the modern view of regularization is that it enables models to generalize well even when the model is large and overfits (i.e. a deep and complex neural network). Surprisingly, it has been shown that large neural networks that include regularization are able to interpolate and generalize well to unseen data even when they are hugely overfit (see literature on the double descent curve). Thus, in deep learning, we are most concerned with getting the best generalization error, and not worrying so much about overfitting.

## 7 Sequential Neural Networks

### 7.1 Recurrent Neural Networks (RNN)

Sometimes we want to model sequential data with temporal dependence like a language modeling task where the next predicted word depends on the context given by the words that came before it, for example. The problem with using a standard neural network to approach these problems is that it has no sense of memory to understand the context of previous events. Further, a standard feed-forward neural network has no built in concept of sequence order. Thus, to model temporal dynamics, we look beyond the feed forward network and introduce recurrent neural networks (RNN).

In a recurrent neural network, weights are repeatedly applied at every timestep, and hidden states are obtained sequentially, with each hidden state depending on the previous hidden state and the new input at the current timestep. This theoretically gives the network context of past events prior to each new timestep. Then, using each hidden state, we can optionally produce a sequence of outputs (many-to-many), or produce a single output (many-to-one) using only the last hidden state.

We begin with an initial hidden state $h^{(0)}$ which can be initialized as zeros. Here, the superscript denotes the timestep.

At each following timestep, we calculate the hidden state based on the previous hidden state and the new input. Note that the weights the same for each timestep and are applied repeatedly.

$$h^{(t)} = tanh(W_h h^{(t-1)} + W_x x^{(t)} + b_1)$$

where $W_h$ is the weight matrix corresponding to the previous hidden state, $W_x$ is the weight matrix corresponding to the new input $x$ and $b_1$ is the bias term.

Then, we can apply a transformation to the hidden states to arrive at the output.

$$\hat{y}^{(t)} = Uh^{(t)} + b_2$$

Here $U$ is the weight matrix corresponding to the output, and $b_2$ is another bias term.

We can also apply a softmax transformation to use the last hidden state $h^{(T)}$ in a sequence classification problem.

$$\hat{y} = softmax(Uh^{(T)} + b_2)$$

The RNN is commonly represented as an unrolled graph, as below in Figure 11. Note how the weights are repeated applied at each timestep.

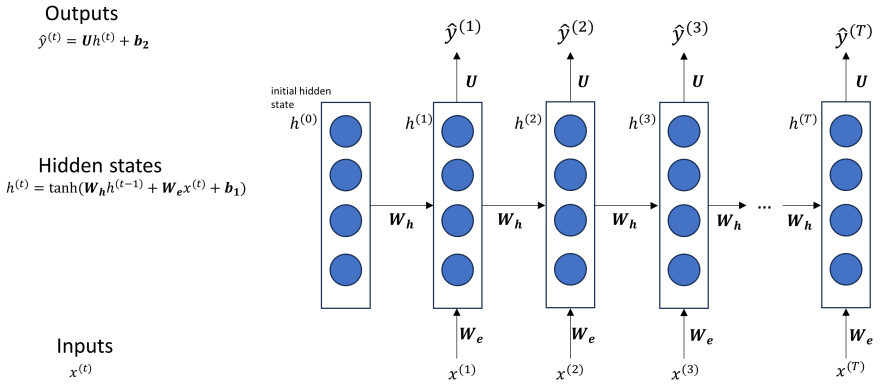


Figure 11: Graph of an unrolled RNN

A RNN can process any length input, and at timestep t, can theoretically use information from any previous timestep. Notably, the model size does not increase for longer inputs. However, recurrent computations are slow, and in practice, it is difficult to access information from many timesteps back. In the next sections, we will discuss how to address these drawbacks.

### 7.1.1 Training a RNN

At each time $t$, we can calculate the loss for that timestep. For example, if we are doing a timeseries prediction problem on MSE loss, we could write

$$J^{(t)}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i^{(t)} - y_i^{(t)})^2$$

Next, we could average losses overall all $T$ timesteps to get the overall loss

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$$

To minimize the loss, we need to calculate the derivative of $J(\theta)$ with respect to the parameters of the model. For example, what is the derivative of $J^{(t)}(\theta)$ with respect to the repeated weight matrix $W_h$? Recall that in a computational graph, gradients sum at outward branches. That is,

$$\frac{\partial}{\partial t} f\left(x(t), y(t)\right) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}$$

Thus, the gradient with respect to a repeated weight matrix is the sum of the gradient with respect to each timestep it appears. That is,

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\bigg|_{(i)}$$

Here, $\frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$ represent the gradient as it is used at time $i$. That is, at time $i$, how does changing $W_h$ affect $J^{(t)}$? Overall, applying the chain rule to compute gradients in such a manner is called "backpropagation through time" (see Figure 12).

### 7.1.2 The Vanishing Gradient Problem

Recurrent Neural Networks suffer from the vanishing gradient problem, since the gradient signal (which instructs the model about what information to store in the hidden states) tends to vanish for longer-term dependencies. Intuitively, this is because long applications of the chain rules increase the risk of vanishing gradients, especially as we chain together derivatives that are less than 1. For example, consider the derivative of the loss at the 10th timestep with respect to the hidden state at the first timestep

$$\frac{\partial J^{(10)}}{\partial h^{(1)}} = \frac{\partial J^{(10)}}{\partial h^{(10)}} \frac{\partial h^{(10)}}{\partial h^{(9)}} \frac{\partial h^{(9)}}{\partial h^{(8)}} \frac{\partial h^{(8)}}{\partial h^{(7)}} \cdots \frac{\partial h^{(2)}}{\partial h^{(1)}}$$

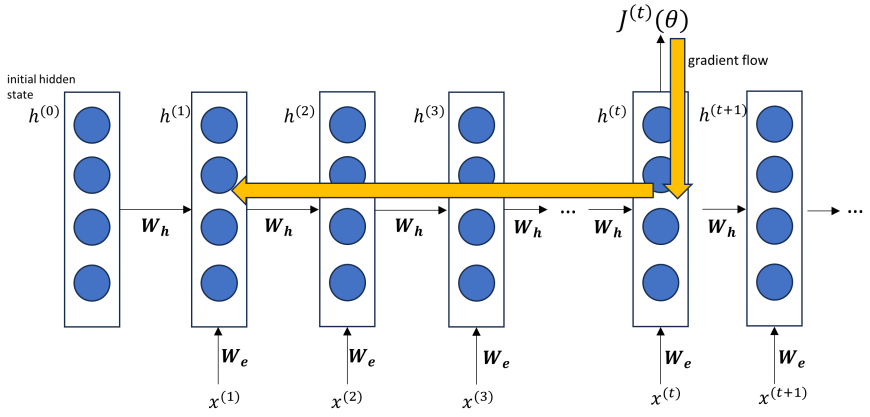


Figure 12: Backpropagation through time in a RNN

The consequence of this is that the gradient signal from far away is lost because it is much smaller than the gradient signal close by. That is, model weights tend to be updated only with respect to near effects and not long term effects.

It is too difficult for the RNN to preserve information over long periods, since in a vanilla RNN, the hidden state is constantly being rewritten at every timestep, and is not a suitable storage of long term information:

$$h^{(t)} = tanh(W_h h^{(t-1)} + W_x x^{(t)} + b_1)$$

This motivates us to consider a RNN that has a separate memory variable that can maintain information over many timesteps, which will lead us to the Long Short Term Memory (LSTM) model architecture.

As an aside, note that the vanishing gradients problem are not unique to only RNN's. Due to the chain rule, the gradient becomes vanishingly small as it backpropagates. This can be a problem for very large feed-forward or convolutional architectures. A common solution is to add more direct connections that allow the gradient to flow such as residual connections (ResNet). The figure below demonstrates ResNet or "skip-connections" (see Figure 13).

## 7.2 Long Short Term Memory (LSTM)

In a LSTM, every timestep $t$ has a hidden state $h^{(t)}$ and a cell state $c^{(t)}$. The cell states store and maintain long term information, and at every timestep, the LSTM can "read", "erase", or "write" from the cell states. When the model reads from the cell state, it releases information from the cell state into the hidden state, where it is used for prediction. As new inputs arrive, the model can also write and erase from the cell state, i.e. deleting and adding information that may be useful to store for future predictions. The selection of information to be erased, written, or read is controlled by so called "gates" that can be open (value of 1), closed (value of 0), or somewhere in between.

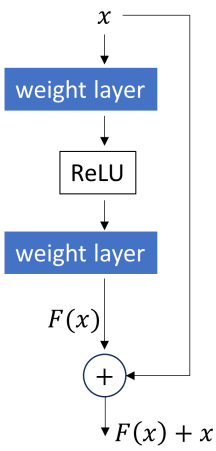


Figure 13: A residual connection

At timestep $t$, the first step would be to compute the gate values. The gate values all depend on the previous hidden state $h^{(t-1)}$ and the new input at the current timestep $x^{(t)}$. The sigmoid function $\sigma$ ensures that the gates take values between 0 and 1:

1. The forget gate (erase), $f^{(t)}$, which controlled by parameters $W_f$, $U_f$, and $b_f$, controls what information is kept versus forgotten from the previous cell state. That is, it controls what information is erased at timestep $t$.
$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f)$$
2. The input gate (write), $i^{(t)}$, controls what parts of the new input is written to the cell state at timestep $t$.
$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i)$$
3. The output gate (read), $o^{(t)}$, controls what information from the cell we output to the hidden state at timestep $t$. That is, this gate controls what information is used to make a prediction at the current timestep.
$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o)$$

After the gate values are computed for the current timestep, we can compute the new cell and hidden states:

1. We first represent new content using a candidate update, $\tilde{c}^{(t)}$. This is similar to the hidden state of a vanilla RNN, and is not to be confused with an LSTM hidden state. As with the hidden state of a vanilla RNN, the candidate update is a function of the previous hidden state and the new input features at the current timestep.
$$\tilde{c}^{(t)} = tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$
2. We then can update the cell state, $c^{(t)}$, forgetting some content from the last cell state, and inputting some new cell content
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)}$$
3. Finally, some content is outputted from the cell state to the hidden state, $h^{(t)}$, where we derive the final prediction.
$$h^{(t)} = o^{(t)} \odot tanh(c^{(t)})$$

Note that the $\odot$ represents the Hadamard product (element-wise multiplication).

Once the hidden state are calculated, the final prediction can be derived depending on the task. For example, softmax can be applied for a multiclass classification task. On the other hand, a linear layer or some

feed-forward layers can produce the output for a regression task.

Overall, the LSTM architecture makes it much easier for an RNN to preserve information over many timesteps. For example, if the forget gate is 1 and the input gate is 0, information in the cell state is preserved indefinitely. In contrast, it is much harder for the vanilla RNN to learn a recurrent weight matrix $W_h$ that preserves information in the hidden states. This behavior is what enables an LSTM to learn long term dependencies and circumvent the vanishing gradient problem.

As shown below in Figure 14, LSTM's are commonly represented as a computational graph. This also makes it easy to see that multiple LSTM units applied over time make it easy to preserve information in the cell state, given the correct gate values (see Figure 15).

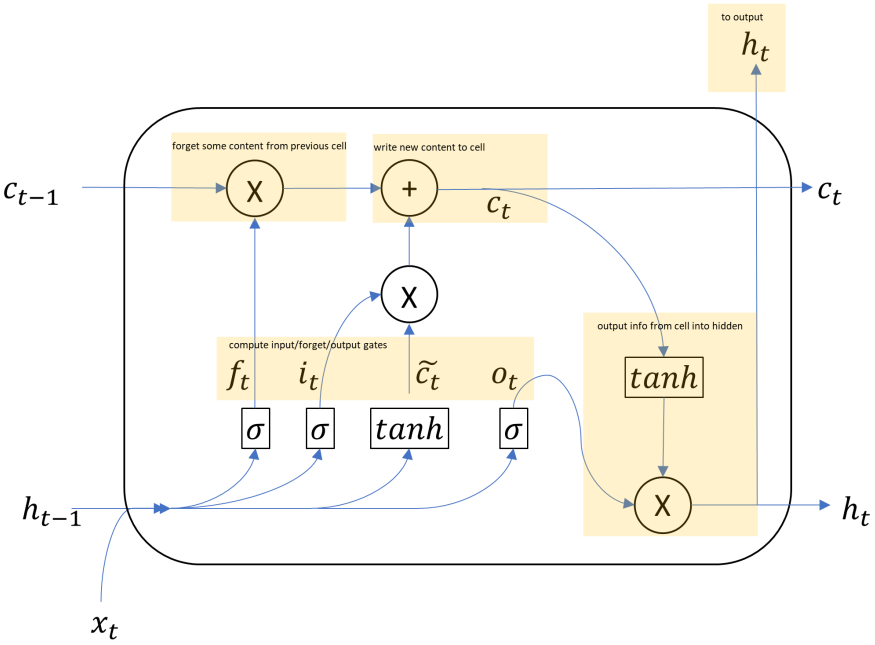


Figure 14: Computational graph for one LSTM unit

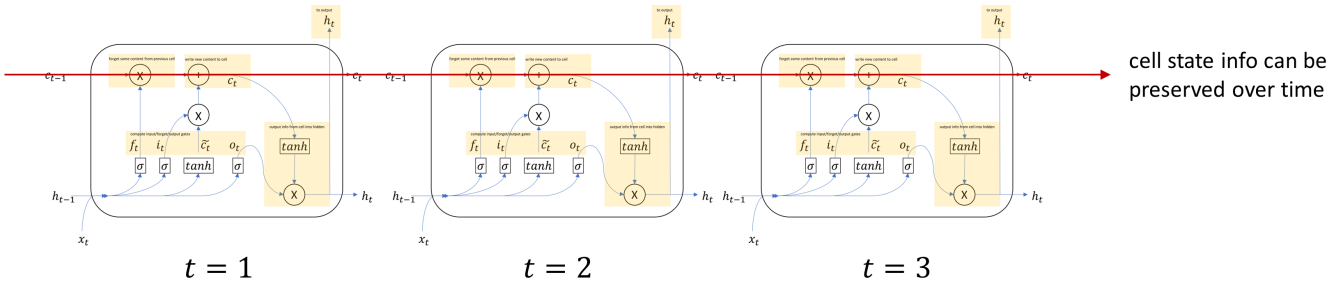


Figure 15: Demonstration of preserving cell state information over several LSTM units

## 7.3 Sequence-to-sequence models

Before we saw how RNN's and LSTM's can make predictions on sequences of data. In this section, we explore how sequences can be used as both inputs and outputs using the encoder-decoder model architecture

(see Figure 16). This is useful especially for input and target sequences of different lengths. Our motivating example will be a machine translation problem where we want to translate the 4 word French sentence "il a m' entarte"' into it's 6 word English translation "he hit me with a pie." The encoder-decoder models are two separate RNN's that are eventually trained simultaneously. The encoder RNN encodes all of the information from the input sentence into its final hidden state, where it is passed to the decoder RNN as its initial hidden state. The decoder RNN is a language model that generates the target sentence, given the output from the encoder RNN. In natural language processing, a language model is a special model that predicts the next word, given the words already predicted so far (in prediction) or the true target words so far (in training). Figure 16 demonstrates the encoder-decoder model architecture.

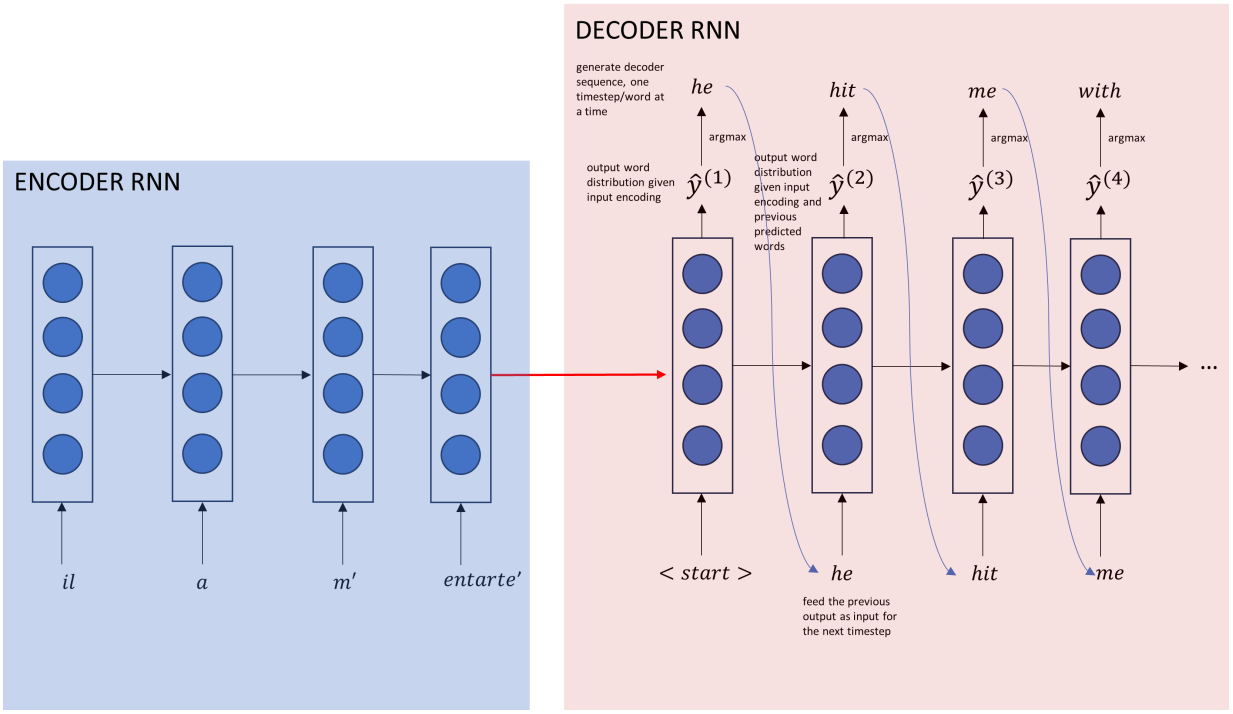


Figure 16: The encoder-decoder model as applied to a machine translation task

As an aside, note that in Figure 16 we sampled from the decoder language model by taking the argmax of each output word distribution for each timestep. This is called greedy decoding, and usually does not produce the optimal result. Instead, "beam search decoding" is more commonly used, where at each timestep, we keep track of the k most probable partial translations, where k is the beam size. Typically, k is between 5 to 10 for machine translation problems.

The encoder-decoder architecture is trained simultaneously. That is, backprogagtion operates end-to-end and updates both the encoder and decoder RNN weights. This is shown below in Figure 17.

## 7.4 Attention

Previously, we discussed how the encoder-decoder model architecture can allow for input and target sequences of different lengths. However, one flaw of such a model is that the final hidden state of the encoder RNN needs to capture all of the information about the source sentence, creating an information bottleneck. The attention mechanism provides a solution to this problem. In attention, we use direct connections to the encoder from the decoder step to focus on a particular part of the source sentence for each step of the decoder. Figure 18 below demonstrates the attention mechanism.

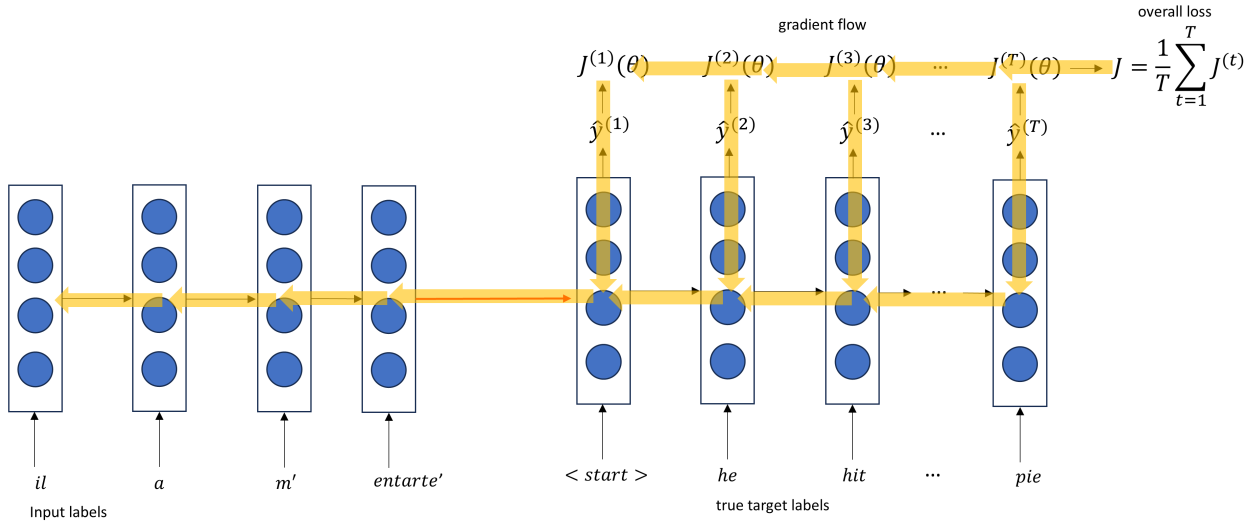


Figure 17: Simultaneous training of an encoder-decoder model as applied to a machine translation task

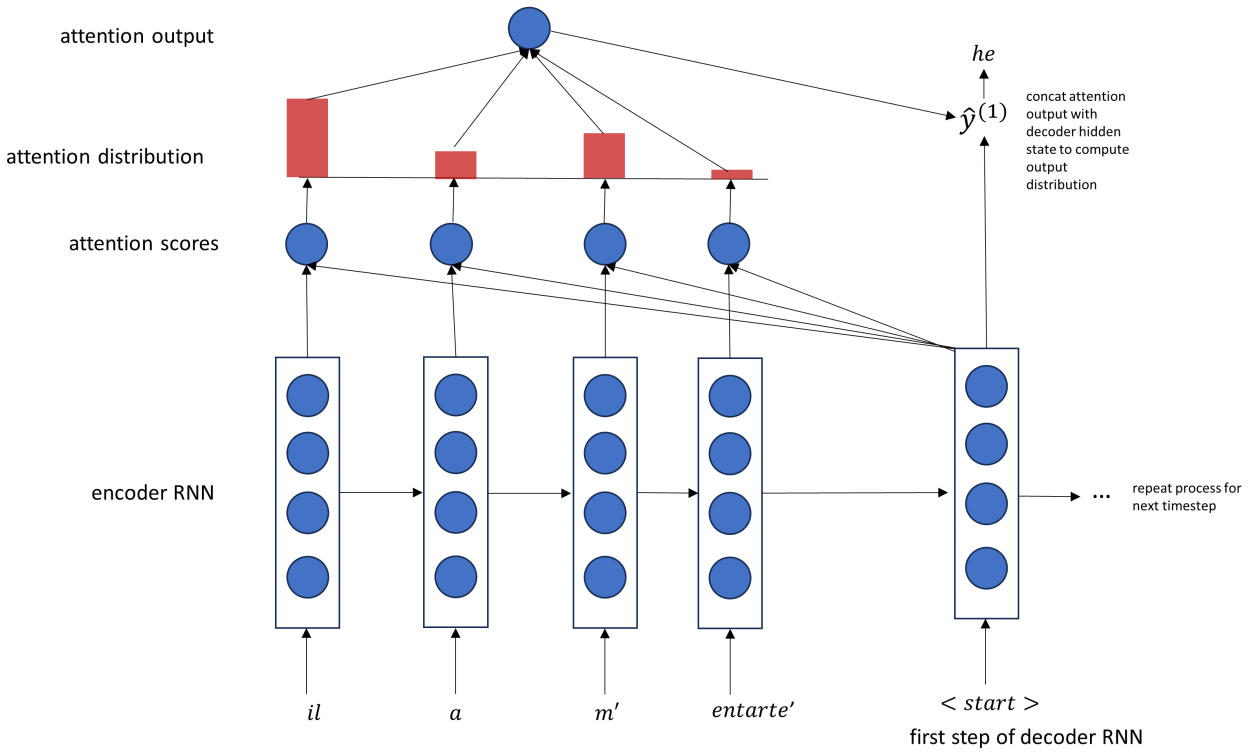


Figure 18: The attention mechanism

Specifically, let's say we have an input sequence of length $N$ and a target sequence of length $T$. Let $h_1, ..., h_N \in \mathbb{R}^h$ be the encoder hidden states and $s_t \in \mathbb{R}^h$ be the decoder hidden state for timestep $t$. The first step is to obtain the *attention scores*, $\vec{e_t} \in \mathbb{R}^N$, for timestep $t$. Here, we use dot product similarity to compare the decoder hidden state with the encoder hidden states.

$$\vec{e_t} = [s_t^T h_1, ..., s_t^T h_N] \in \mathbb{R}^N$$

Next, we want to take the softmax to normalize the attention scores and obtain the *attention distribution*, $\alpha_t$, for timestep $t$

$$\alpha_t = softmax(e_t) \in \mathbb{R}^N$$

Finally, we get the *attention output*, $a_t \in \mathbb{R}^h$, by taking a weighted sum of the encoder hidden states using the attention distribution.

$$a_t = \sum_{i=1}^{N} \alpha_{it} h_i \in \mathbb{R}^h$$

The attention output, $a_t$, and the decoder hidden state for time $t$, $s_t$, can be concatenated to proceed with the decoder step (e.g. make a prediction at timestep $t$).

$$[a_t; s_t] \in \mathbb{R}^{2h}$$

There are several variants of attention, but the main idea and terminology are the same. We always have some *values*, *keys*, and *queries*. The attention scores are computed by comparing the *queries* and the *keys*, the attention distribution is computed by taking the softmax of the attention scores, and the attention output is obtained by taking a weighted sum of the *values*. In the dot product similarity example above, the encoder hidden states $h_1, ..., h_N$ were both the *values* and *keys*, and the decoder hidden state $s_t$ was the *query*.

#### 7.4.1 Some variants of attention

1. Dot product attention: $e_i = s^T h_i$. This assumes that every hidden unit has information about attention.
2. Multiplicative attention: $e_i = s^T W h_i$ where $W$ is a weight matrix that models what parts of $s$ and $h$ to focus on. However, is it necessary to have an entire matrix of parameters in $W$?
3. Reduced rank multiplicative attention: $e_i = s^T (U^T V) h_i = (Us)^T (Vh_i)$. Here, we model $W$ using low rank "skinny matrices." The idea of using $Us$ and $Vh_i$ will be very important for Transformers.

### 7.5 Transformer

So far, we have discussed different recurrent models and their variants. However, the biggest drawback for recurrent models is their lack of parallelizability. That is, future RNN hidden states cannot be computed in full before past hidden states have been computed. Instead of using a recurrent structure, the transformer model architecture focuses on "stacked" (self) attention layers. Since all words interact with each other at every layer, the number of unparallelizable operations does not increase with sequence length. Figure 19 below demonstrates parallelizability for stacked attention layers.
Note that using attention weights for each layer is different than a fully connected layer because the weights are dynamic with the input, while in a feed-forward network, the weights are fixed after training.

#### 7.5.1 Self-attention

In self-attention, the keys, queries, and values come from the same sequence, in contrast to before where we used attention to compare the encoder and decoder hidden states.

We will discuss Transformers in the context of natural language processing, but the concepts apply for any sequence modeling task. Let $w_1, ..., w_n$ be a sequence of $n$ words (as one-hot vectors, for example). For each

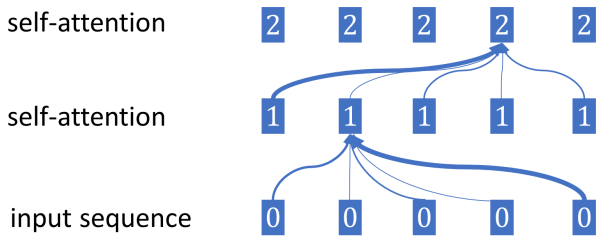


Figure 19: Parallelizability for stacked attention layers. The numbers indicate the minimum number of steps before a state can be computed.

$w_i$, let $x_i$ be its corresponding embedding where $x_i = Ew_i$ and $E \in \mathbb{R}^{dxV}$ is an embedding matrix. Here, $d$ is the dimensionality of the embedding, and $V$ is the vocabulary size of the model. Using dense embeddings provide an opportunity for dimensionality reduction, as opposed to use one-hot vectors for each word.

The first step is to transform each word embedding with weight matrices $Q, K, V \in \mathbb{R}^{dxd}$ to obtain the queries, keys, and values.

$$q_i = Qx_i$$
$$k_i = Kx_i$$
$$v_i = Vx_i$$

Next, we obtain the pairwise similarities, i.e. the key-query affinities. Below, we compare word $i$ to word $j$ in the sequence to get attention scores.

$$e_{ij} = q_i^T k_j$$

We then normalize the attention scores with softmax to obtain an attention distribution

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{j'=1}^{n} exp(e_{ij'})}$$

Finally, we compute the attention output for each word $i$ as a weighted sum of values according to the attention distribution

$$o_i = \sum_{j=1}^{n} \alpha_{ij} v_j$$

We can compute key-query-value attention in matrices. Let $X = [x_1, ..., x_n] \in \mathbb{R}^{nxd}$ be the concatenation of the whole sequence of input vectors. Note that $XK$, $XQ$, and $XV \in \mathbb{R}^{nxd}$ are the matrices containing the keys, queries, and values for each word in the sequence.

Next, the query-key dot products can be represented by

$$XQK^T X^T \in \mathbb{R}^{nxn}$$

where the resulting matrix contains all pairs of attention scores. We then take the softmax (across rows) of this matrix, and matrix multiple with the matrix of values to obtain the outputs.

$$output = softmax(XQK^T X^T)XV \in \mathbb{R}^{nxd}$$

### 7.5.2 Self-attention as a building block

Our goal is to use stacked attention layers as a "drop-in" replacement for recurrence (see Figure 20). However, the naive implementation of stacked attention layers as a direct replacement for recurrence has several barriers to its success, which we will discuss one at a time in the following subsections.

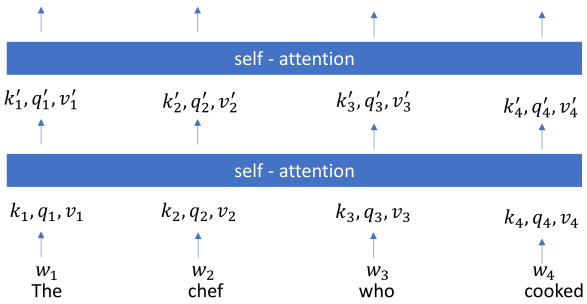


Figure 20: Stacked attention layers as a replacement for recurrence in a sequence model

### 7.5.3 Barrier #1: Order information

The first barrier to the implementation of stacked attention layers is that a naive implementation would have no sense of sequence order. The solution to this problem is to encode position into vectors $p_i$ for $i = 1, ..., n$ and add them to the inputs $x_i$:

$$\tilde{x_i} = x_i + p_i, \ i = 1, .., n$$

The original Transformer paper used sinusoidal position representations to create unique vectors for every index $i$, but a more recent approach is to let all $p_i$ be learnable parameters. Note that both methods struggle to extrapolate to indices outside of $1, ..., n$, (recall we have a sequence of $n$ words) but having $p_i$ be learnable is now the most prevalent approach.

### 7.5.4 Barrier #2: Adding nonlinearities

The second barrier to stacked attention is that there are no nonlinearities introduced into the model, since attention outputs are simply linear combinations of the original inputs. To solve this issue, we add feedforward neural network layers in between layers to introduce nonlinearities by post-processing each attention output vector (see Figure 21).

$$m_i = MLP(output_i) = W_2 ReLU(W_1 output_i + b_1) + b_2$$

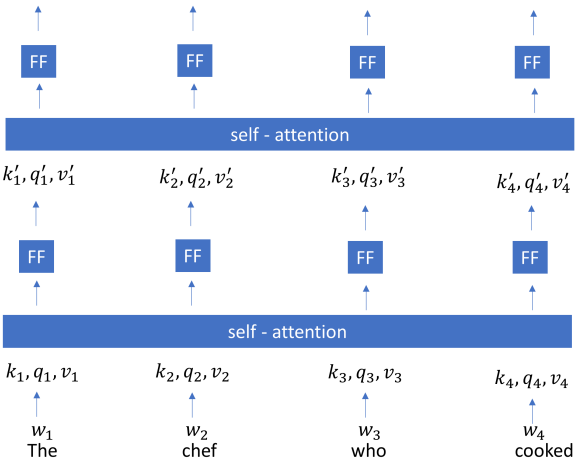


Figure 21: Stacked attention layers with feedforward networks

### 7.5.5 Barrier #3: Seeing the future

In attention, the entire sequence interacts with each other. This a problem when in sequence modeling where we cannot cheat by looking into the future. For instance, the transformer decoder is a language model that

predicts the next words (one at a time) given the *previous* predicted words, so it would defeat the purpose if the model had access to *all* target words during training. To solve this problem, we mask out attention to future words by setting the attention scores corresponding to future timesteps to $-\infty$ (so that the future timesteps will be assigned a probability of 0 in the attention distribution). See Figure 22.


Figure 22: Transformer (decoder) masking for self attention

Note that seeing the future is not a problem in decoder RNN's since they process one word at a time in order by design.

#### 7.5.6 Tricks to help with training: Scaled dot product

In addition to overcoming the difficulties of using stacked self-attention layers, the Transformer architecture includes several tricks to facilitate more stable training.

In particular, when the dimensionality, $d$, becomes large, the dot products between vectors tend to be large. With large inputs into softmax, we run the risk of small gradients. Instead, we divide attention scores by $\sqrt{d/h}$ to stop attention scores from becoming large just as a function of $d/h$ (that is, the dimensionality / # of attention heads). That is, for the $\ell th$ attention head (more details on multi-headed attention later), the attention output is

$$output_\ell = softmax\left(\frac{XQ_\ell K_\ell^T X^T}{\sqrt{d/h}}\right) XV_\ell$$

#### 7.5.7 Tricks to help with training: Residual connections

The Transformer architecture includes several residual connections to facilitate gradient flow. Recall that with residual connections, instead of modeling the output directly as a function of the inputs, we only model the residual from the previous layer. Note that residual connections allow the gradient to flow even if the gradient in the layer itself vanishes. See Figure 23 below.

#### 7.5.8 Tricks to help with training: Layer normalization

The idea of layer normalization is to cut down on uninformative variation in hidden vector values by normalizing them to unit mean and standard deviation within each layer. Let $x \in \mathbb{R}^d$ be an individual hidden

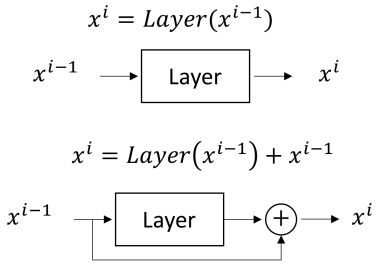


Figure 23: No residual connection vs a residual connection. Here, $x^{i-1}$ represents the output from the previous layer, and $x^i$ represents the output from the current layer.

(word) vector in the model. Let $\mu = \frac{1}{d}\sum_{j=1}^{d} x_j$ and $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}$. Then we compute

$$\frac{x - \mu}{\sigma + \epsilon} * \gamma + \beta$$

where $\epsilon, \gamma, \beta$ are optional bias parameters.

#### 7.5.9 Cross-attention

Information from the encoder is passed to the decoder through cross attention. Let $h_1, ..., h_n$ be output vectors from the transformer encoder where $h_i \in \mathbb{R}^d$. Let $z_1, ..., z_n$ be input vectors from the transformer decoder where $z_i \in \mathbb{R}^d$.

Keys and values are taken from the encoder:

$$k_i = Kh_i$$
$$v_i = Vh_i$$

and queries are taken from the decoder:

$$q_i = Qz_i$$

#### 7.5.10 Multi-headed Attention

What if we want to look in multiple places in the sentence at once? For word $i$, self attention focuses on word $j$ where $x_i^T Q^T K x_j$ is high, but what if we want to focus on different $j$ for different reasons? To address this, we define multiple attention heads through multiple $Q, K, V$ matrices.

Let $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{dx(d/h)}$ where $h$ is the number of attention heads and $\ell = 1, ..., h$.

Each attention head performs self-attention independently from the others.

$$output_\ell = softmax(XQ_\ell K_\ell^T X^T)XV_\ell \in \mathbb{R}^{d/h}$$

Then, each of the outputs are combined

$$output = [output_1, ..., output_h]$$

Each head "looks" at different things for different reasons, and constructs the value vectors differently from each other.

#### 7.5.11 Transformer architecture

Putting the above factors together, we complete the Transformer architecture as shown in Figure 24.

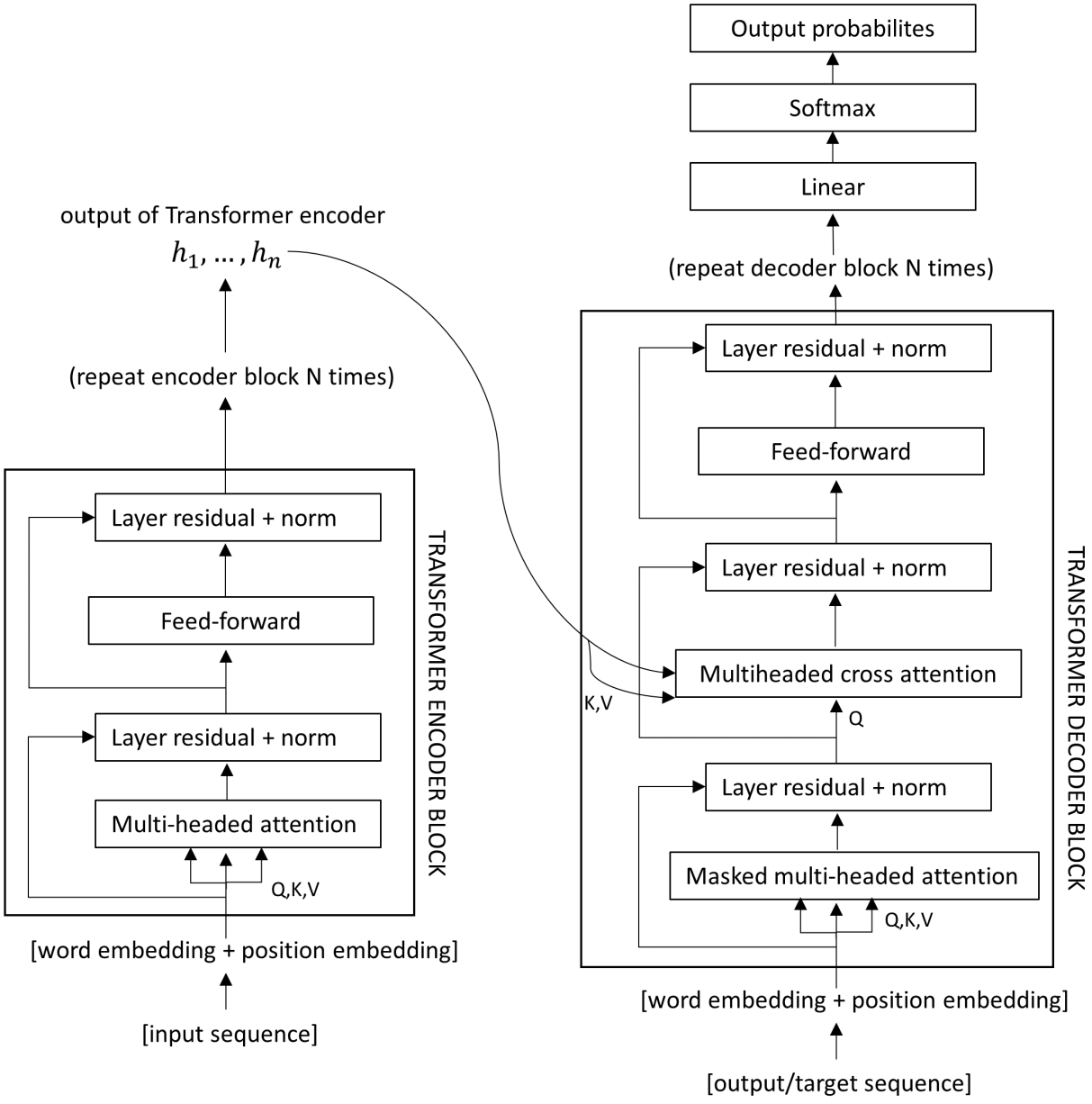


Figure 24: The Transformer architecture

# 8 Unsupervised Learning

Let $\mathbb{P}(X, Y)$ represent the joint probability distribution of the data. In supervised learning, we are concerned with the conditional probability distribution of $Y$ given a specific value of $X = x$. That is, we want to learn about $\mathbb{P}(Y|X = x)$ with a labelled dataset. In the supervised setting, we can learn about $\hat{y}$ since the true $y$ is known in the training dataset.

The goal of unsupervised learning, however, is different. Here, there is no knowledge of the "true values." Instead, we wish to learn about the properties of $\mathbb{P}(X, Y)$ directly. Under this framework, we often do not have a "target" variable as in supervised learning, so we simplicity we can simply denote $\mathbb{P}(X)$ with $X$ representing all of the data.

For example, in dimensionality reduction techniques (PCA, self-organizing maps, etc...), the goal is to learn lower dimensional manifolds that represent the data. In clustering analysis, the goal is to identify regions that contain the modes of $\mathbb{P}(X)$. Gaussian Mixture Models, for example, answer the question if $\mathbb{P}(X)$ can be described by a mixture of simpler Gaussian densities. Association Rule Mining, given a large and sparse binary data, finds simple descriptions that describe areas of high data density.

## 8.1 Association Rules

In association rule mining, the goal is to find values of $\vec{X}$ that jointly have a relatively large probability with respect to the density $\mathbb{P}(X)$. However, searching for specific values of $\vec{X}$ is intractable, so we focus on finding regions $s_j$ for each $X_j$ with relatively high probability. That is, supposing that $\vec{X}$ contains $p$ variables, we are looking for $s_1, ..., s_p$ where the following is relatively large

$$\mathbb{P}\left[\bigcap_{j=1}^{p}(X_j \in s_j)\right]$$

In Market Basket Analysis, this can be simplified further, considering only binary data (that is after one-hot-encoding the dataset, for example). Suppose we have $K$ binary variables, represented by $Z_1, ..., Z_K$. This is particularly useful for finding patterns in very large commercial datasets. For example, consider a tabular dataset where the columns are dummy variables representing a grocery store's catalog, and the rows represent the purchases of different customers and we want to know what items are frequently purchased together (hence the name, market basket analysis). In this situation, we can further simplify the above expression considering an "item set", which we denote $\mathcal{K}$. $\mathcal{K}$ is a set of integers representing which dummy variables are in our item set, $\mathcal{K} \subset \{1, ..., K\}$. Then we are searching for item sets where the following probability is relatively large

$$\mathbb{P}\left[\bigcap_{k\in\mathcal{K}}(Z_k = 1)\right] = \mathbb{P}\left[\prod_{k\in\mathcal{K}} Z_k = 1\right]$$

We can estimate this probability using the fraction of observations in the data that contains the item set $\mathcal{K}$. Let $z_{ik}$ denote the outcome of the dummy variables $Z_k$ for the ith data observation. We have

$$\hat{\mathbb{P}}\left[\prod_{k\in\mathcal{K}} Z_k = 1\right] = \frac{1}{n}\sum_{i=1}^{n}\prod_{k\in\mathcal{K}} z_{ik}$$

The above quantity is called the support of the item set $\mathcal{K}$, which we can denote as $T(\mathcal{K})$. The support represents the prevalence of a particular item set in the data. In association rule mining, we are searching for all item sets that have a support greater than some threshold $t$. That is, we are searching for the set

$$\{\mathcal{K}_\ell | T(\mathcal{K}_\ell > t\}$$

This problem can efficiently be solved using the Apriori Algorithm, with only a few passes of the data needed (with sufficiently sparse data). In the first pass, we consider item sets of size 1, and discard all sets that do not satisfy the support threshold. In the second pass, we consider item sets of size 2, using only only the survivors of the first pass, and so on for the subsequent passes. Then, each high support item sets resulting from the Apriori algorithm is broken into two disjoint subsets $A \cup B = \mathcal{K}$ where we can write the association rule $A \rightarrow B$ . $A$ is called the "antecedent" and $B$ is called the "consequent."

The "confidence" of the association rule, which we denote as $C(A \rightarrow B)$ can be written as

$$C(A \rightarrow B) = \frac{T(A \rightarrow B)}{T(A)} = \frac{T(\mathcal{K})}{T(A)}$$

This is an estimate of the conditional probability measure $\mathbb{P}(B|A)$.

Finally, we can define the "lift" of the association rule, which is an estimate of $\mathbb{P}(A \cap B)/\mathbb{P}(A)\mathbb{P}(B)$. Lift is defined as

$$L(A \rightarrow B) = \frac{C(A \rightarrow B)}{T(B)}$$

The overall goal of this analysis is to find all association rules with both high support and high confidence. That is, after finding high support item sets using the Apriori Algorithm, we can to return

$$\{A \rightarrow B | C(A \rightarrow B) > c\}$$

for some confidence threshold c.

## 8.2 K-means Clustering

In cluster analysis, datapoints are assigned to clusters such that the points inside the cluster are more similar than the points in the other clusters. In k-means clustering, the datapoints are assigned to clusters with the nearest mean, and seeks to minimize the within-cluster (unexplained) variance.
Given $k$ different clusters/sets $S = \{S_1, ..., S_k\}$, the goal is to solve for $S$ such that the within-cluster sum of squares is minimized

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||_2^2$$

where $\mu_i = \frac{1}{|S_i|} \sum_{x \ \in S_i} x$ are the cluster means.

This problem is equivalent to maximizing the between cluster sum of squares since total variance is constant for the dataset (recall sum of square decomposition from linear regression).

Unfortunately, finding the optimal solution to the problem as stated above is computationally difficult. Commonly, heuristic algorithms are used to find a local solution. For example, in Lloyd's algorithm the following process is repeated until the cluster assignments have converged. We begin with randomly assigned cluster assignments are repeat

1. Calculate means (centers) for each assigned cluster
2. Update cluster assignments by re-assigning points to the closest mean

## 8.3 Mixture Models and the EM Algorithm

Mixture models are useful when data is randomly drawn from several different distributions. For example, we can write the data generating process for $x$ as a mixture of $K$ different clusters/distributions

$$x \sim p_\theta(x) = \sum_{j=1}^{K} p_\theta(Z=j)p_\theta(x|Z=j) = \sum_{j=1}^{K} p_\theta(x, Z=j)$$

where $Z$ represents the cluster assignment and $\theta$ represents the model parameters. $p_\theta(Z=j)$ are called the mixture proportions. In many problems, we face a situation where the cluster assignments and mixture proportions are unknown.

We have that the maximum likelihood estimator (MLE) for $\theta$ over $n$ data observations is

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} log p_\theta(x_i) = \underset{\theta}{\text{argmin}} \sum_{i=1}^{n} log \sum_{j=1}^{K} p_\theta(x_i, Z_i = j)$$

which is a difficult non-convex optimization problem. We turn to the EM algorithm, which is a non-convex optimization strategy that is designed to solve for the MLE in problems with missing or unobserved data (such as unknown cluster assignments).
We can rewrite the log-likelihood by introducing a new pmf denoted by $q_i$

$$\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{n} log \sum_{j=1}^{K} p_\theta(x_i, Z_i = j) \\
&= \sum_{i=1}^{n} log \sum_{j=1}^{K} q_i(Z_i = j) \frac{p_\theta(x_i, Z_i = j)}{q_i(Z_i = j)} \\
&= \sum_{i=1}^{n} log\ \mathbb{E}_{q_i} \frac{p_\theta(x_i, Z_i)}{q_i(Z_i)}
\end{aligned}$$

By Jensen's Inequality, that is $\mathbb{E}f(X) \geq f(\mathbb{E}X)$ for convex $f$, we have that

$$\begin{aligned}
\ell(\theta) &\geq \sum_{i=1}^{n} \mathbb{E}_{q_i} log \frac{p_\theta(x_i, Z_i)}{q_i(Z_i)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{K} q_i(Z_i = j) \frac{p_\theta(x_i, Z_i = j)}{q_i(Z_i = j)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{K} q_i(Z_i = j) log\ p_\theta(x_i, Z_i = j) - q_i(Z_i = j) log\ q_i(Z_i = j)
\end{aligned}$$

We define the first term as

$$Q(\theta) := \sum_{i=1}^{n} \sum_{j=1}^{K} q_i(Z_i = j) log\ p_\theta(x_i, Z_i = j)$$

noting that the second term does not depend on $\theta$ and is not needed for the optimization problem.

The question remains on how to chose $q_i(Z_i)$ so that the bound of the inequality is as tight as possible. If we choose

$$\begin{aligned}
q_i(Z_i = k) &= \frac{p_{\theta^{(t)}}(x_i, Z_i = k)}{\sum_{j=1}^{K} p_{\theta^{(t)}}(x_i, Z_i = j)} = \frac{p_{\theta^{(t)}}(x_i | Z_i = k) p_{\theta^{(t)}}(Z_i = k)}{\sum_{j=1}^{K} p_{\theta^{(t)}}(x_i | Z_i = j) p_{\theta^{(t)}}(Z_i = j)} \\
&= p_{\theta^{(t)}}(Z_i = j | x_i)
\end{aligned}$$

where $\theta^{(t)}$ represents the $t^{th}$ iteration for $\theta$ of the algorithm, then the bound from Jensen's Inequality becomes

an equality when $\theta^{(t)} = \theta$ since the quantity

$$\begin{aligned}\frac{p_\theta(x_i, Z_i)}{q_i(Z_i)} &= \frac{p_\theta(x_i, Z_i)}{p_\theta(x_i, Z_i)/\sum_{j=1}^K p_\theta(x_i, Z_i = j)} \\ &= \sum_{j=1}^K p_\theta(x_i, Z_i = j)\end{aligned}$$

is deterministic.

The EM algorithm consists of iterating the following steps

1. E-step (Expectation): Solve for $p_{\theta^{(t)}}(Z_i = j|x_i)$ for all $j = 1, ..., K$ using the current parameters $\theta^{(t)}$
2. M-step (Maximization): Update the parameters by solving maximizing $Q(\theta)$

$$\theta^{(t+1)} = \underset{\theta}{\text{argmax}} \sum_{i=1}^n \sum_{j=1}^K p_{\theta^{(t)}}(Z_i = j|x_i) log\ p_\theta(x_i, Z_i = j)$$

Note that the M-step often has an analytical solution and is easy to compute.

### 8.3.1 Gaussian Mixture Model

Consider the following data generating process

$$x \sim p_\theta(x) = \sum_{j=1}^K \pi_j \mathcal{N}(x; \mu_j, \Sigma_j)$$

where $\pi_j = p_\theta(Z = j)$ and the unknown parameters are

$$\theta = (\pi_1, ..., \pi_K, \mu_1, ..., \mu_K, \Sigma_1, ..., \Sigma_K)$$

We now apply the EM Algorithm. In the E-step, we have by Bayes's Rule that

$$p_{\theta^{(t)}}(Z_i = j|x_i) = \frac{\mathcal{N}(x_i; \mu_j^{(t)}, \Sigma_j^{(t)})\pi_j^{(t)}}{\sum_{j'=1}^K \mathcal{N}(x_i; \mu_{j'}^{(t)}, \Sigma_{j'}^{(t)})\pi_{j'}^{(t)}}$$

and that

$$Q^{(t)}(\theta) = \sum_{i=1}^n \sum_{j=1}^K p_{\theta^{(t)}}(Z_i = j|x_i) log\left(\mathcal{N}(x_i; \mu_j, \Sigma_j)\pi_j\right)$$

In the M-step, we solve for

$$\theta^{(t+1)} = \underset{\theta}{\text{argmax}} Q(\theta)$$

and obtain the following updates

In practice, the Gaussian Mixture Model can be used as a soft clustering algorithm. That is, clusters can be assigned for each datapoint $x_i$ by comparing the conditional probabilities $p_{\theta^{(t)}}(Z_i = j|x_i)$ for $j = 1, .., K$. By doing so, we also have the benefit of built-in uncertainty estimates.

Additionally, clustering using the Gaussian Mixture Model has a more flexible structure compared to K-means clustering due to the flexibility in the covariance parameters $\Sigma_j$ and allows for ellipsoid clusters. In contrast, K-means clustering only models spherical clusters.

## 8.4 Principal Component Analysis (PCA)

Consider a random vector $\vec{x} \in \mathbb{R}^p$ with mean $\mu$ and covariance matrix $\Sigma$. If $\vec{x}$ is high-dimensional ($p$ is large), we may be interested in representing $\vec{x}$ in $k < p$ components with the goal of explaining as much variance in $\vec{x}$ as possible using these $k$ components. Principal Components Analysis (PCA) gives us the best possible *linear* approximation of $\vec{x}$ using $k$ uncorrelated components. For the special case when $\vec{x}$ is multivariate Gaussian, i.e. $\vec{x} \sim \mathcal{N}(\mu, \Sigma)$, PCA gives us the best possible approximation of $\vec{x}$ using $k$ independent components (recall that uncorrelated Gaussian random variables are independent).

Assume without loss of generality that $\vec{x}$ has been scaled to have zero mean. We begin by finding a *linear* combination of the elements of $\vec{x}$ that has maximal variance, expressed as the vector dot product $v^T x$ where $v \in \mathbb{R}^p$. Recall that the variance of $v^T x$ is $v^T \Sigma v$. So among solutions where $||v||_2 = 1$, we want to solve

$$\underset{v,\ ||v||=1}{\text{argmax}}\ v^T \Sigma v$$

Writing the Lagrangian, we have

$$\mathcal{L}(v, \lambda) = v^T \Sigma v - \lambda(v^T v - 1)$$

where $\lambda \in \mathbb{R}$ is the Lagrange multiplier. Then taking the gradient of $\mathcal{L}(v, \lambda)$ and setting it to zero we arrive at the solution

$$\begin{aligned} \nabla_v \mathcal{L}(v, \lambda) &= 2\Sigma v - 2\lambda v = 0 \\ \Sigma v &= \lambda v \end{aligned}$$

This can be recognized as an eigenvector equation for the matrix $\Sigma$. Note that we have variance

$$v^T \Sigma v = v^T(\lambda v) = \lambda v^T v = \lambda$$

Thus, to maximize the variance quantity, we want to choose $\lambda = \lambda_1$ as the largest eigenvalue of $\Sigma$ and $v = e_1$, the first eigenvector of $\Sigma$. The product $z_1 = e_1^T x$ is known as the *first principal component*, and we just showed that the variance of the first principal component is $\lambda_1$

To find the second principal component (and so on) we want to search for $v$ so that it is orthogonal to $e_1$, that is, so that the first and second principal components are uncorrelated. Solving for

$$\underset{v,\ ||v||=1,\ v^T e_1 = 0}{\text{argmax}}\ v^T \Sigma v$$

we find that $v = e_2$, the second eigenvector of $\Sigma$ and that the variance of the second principal component is $\lambda_2$. Similarly, each subsequent $i \in \{1, ..., p\}$ principal component has variance $\lambda_i$ and direction $e_i$.

In general, we can express all $p$ principal components as $z \in \mathbb{R}^p$

$$z = Q^T(x - \mu)$$

where $Q \in \mathbb{R}^{p,p}$ is a matrix with the eigenvectors of $\Sigma$ in its columns. Recall that since $\Sigma$ is symmetric and square, it will have $p$ orthogonal eigenvectors.

We can interpret PCA as an orthogonal change of basis so that the data in the new coordinate system is uncorrelated and in order of decreasing variance (for example, we can write $Ix = Qz$ where $I$ and $Q$ contain in their columns the old and new basis vectors, respectively, to represent the transformation $z = Q^T x$). With properly chosen eigenvectors (they are only uniquely defined up to sign), this transformation is a rotation of the original canonical basis.

We can see that the new data $z = Q^T x$ is uncorrelated (again assuming that $x$ has been scaled to zero mean) since

$$\begin{aligned}\Sigma_z &= \mathbb{E}zz^T = \mathbb{E}(Q^T x)(Q^T x)^T \\ &= Q^T \mathbb{E}xx^T Q = Q^T \Sigma_x Q \\ &= Q^T (Q\Lambda Q^T) Q = \Lambda\end{aligned}$$

where $\Lambda = diag(\lambda_1, ..., \lambda_p)$ has zero off-diagonal values.

To understand dimensionality reduction, we represent the total variance of $\vec{x}$ as

$$Tr(\Sigma) = Tr(Q\Lambda Q^T) = Tr(\Lambda Q Q^T) = Tr(\Lambda) = \sum_{i=1}^{p} \lambda_i$$

and the proportion of variance explained by selecting $k < p$ principal components is

$$\sum_{i=1}^{k} \lambda_i / \sum_{i=1}^{p} \lambda_i$$

In practice, we conduct PCA on the sample covariance matrix instead of the population $\Sigma$.