Scribed Notes: AI for Business Research [1]

Renyu (Philip) Zhang[2]

December 31, 2024

[1] The lecture notes are based on the scribed notes of the students taking the PhD course DOTE6635 at CUHK Business School in Spring 2024.

[2] https://rphilipzhang.github.io/rphilipzhang/index.html; questions please address to philipzhang@cuhk.edu.hk

Contents

1

Machine Learning Basics

——**Scribed by Zhi Li and Boya Peng**

1.1 Outline

The purpose of this section is to give a crash course on ML basics. The focus will be on supervised learning, whereas unsupervised learning will be covered at a later stage of this course. For the lecture notes and Jupyter notebooks of this course, we will introduce:

- The most important conceptual theory;
- The modeling details;
- The case study with the executable Python code demonstration;
- The typical applications of the method in business research.

Specifically, the outline of this document is as follows:

- In Section 1.2, we introduce the basic framework for supervised learning.
- In Section 1.3, we introduce the bias-variance trade-off in supervised learning.
- In Section 1.4, we discuss regularization.
- In Section 1.5, we briefly introduce several traditional supervised learning models: k-Nearest Neighbors, decision tree, random forests and boosting tree.

In this and other lecture notes of this course, we will use $\mathbb{R}^p$ to denote the $p-$dimensional real Euclidean Space, $\mathbb{P}[A]$ to denote the probability of an event A, and $\mathbb{E}[X]$ to denote the expectation of a random variable X. The conditional probability and expectation are denoted as $\mathbb{P}[\cdot|\cdot]$ and $\mathbb{E}[\cdot|\cdot]$, respectively. We denote $|A|$ as the cardinality of set A. You may refer to the respective chapters of Hastie et al. (2009) for more in-depth materials of the topics discussed in this document.

Heads Up

Starting from the very beginning, we will inevitably use a lot of **mathematical notations**, this is because:

- Mathematics enables us to represent some concepts, ideas, and implementations neatly, succinctly, and elegantly, which would otherwise be impossible or very cumbersome.
- The mathematical notations introduced in this course are the standard language in the ML/AI community. Familiarizing ourselves with them is important for us to communicate with other professionals in this community.

Importantly, I would emphasize that, although the mathematical notations look intimidating, you will find that they are so natural and helpful once you understand the logic behind them.

1.2 Supervised Learning in a Nutshell

Supervised learning refers to a wide variety of machine learning tasks through which one seeks to identify the underlying relationship between the ($p-$dimensional) feature $X = (X_1, X_2, ..., X_d)' \in \mathbb{R}^d$ ($'$ is the transpose operation for a matrix) and label $Y \in \mathbb{R}$. In supervised learning, Y is the key variable of interest, whose value is so important for (business) decision making that we wish to predict as accurate as possible. Typical examples of Y in different application contexts include, e.g.,:

- The daily active users (DAU) and daily usage time per user of a user-generated content (UGC) app like Tiktok;
- The survival of a restaurant in 5 years;
- Whether this text has a positive or negative sentiment.

In the literature and practical applications, Y also has some other names such as dependent variable, response variable, target variable, output variable, outcome variable, etc.

The feature vector X contains the variable(s) that could be used to explain and/or predict Y. The key value of the features X lies in their capability to accurately predict Y. Typical examples of X in applications include:

- The user demographic data of an App;
- The user behavior data of an App;
- The characteristics of a product;
- A text or image.

In different contexts, X may have some other names such as independent variable, covariate, predictor, explanatory variable, input variable, data, etc.

To be more specific, we seek to build a learner (a.k.a. model) $\hat{f}(\cdot)$ based on the training data set of sample size n,

$$\mathcal{D} := \{(Y_i, X_i) : Y_i \in \mathbb{R}, X_i \in \mathbb{R}^d, 1 \le i \le n\}$$

that predicts the outcome Y from feature vector X of the following form:

$$Y_i \approx \hat{Y}_i := \hat{f}(X_i) \text{ for all } i = 1, 2, ..., n$$

Here, $\approx$ refers to “as close as possible", or some kind of error is minimized. In this course, we use the notation $\hat{}$ to represent what is generated from data. At a high level, we train our model $\hat{f}(\cdot)$ to make as few mistakes as possible (on the training set):

$$\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(\hat{f}(X_i), Y_i) \text{ is minimized,}$$

where $\mathcal{L}(\cdot,\cdot)$ is some error measure between the predicted value $\hat{f}(X_i)$ and true value Y_i. We will discuss some specific loss functions later in this document.

We now introduce some examples of supervised learning:

- Use movie features (genre, actors, average rating, length, etc.) and user features (age, gender, location, ratings given to other films, etc.) to predict the rating a user will give to a movie s/he has never watched before.

- Use user past behaviors to predict his/her total app time of TikTok tomorrow.

- Use email texts and figures to predict whether it is a spam.

- Use CT scan images to predict whether the patient has lung cancer.

There are two types of supervised learning, regression and classification. For regression, Y is a real-valued continuous variable such as rating, app time, price, etc. The commonly used loss/error functions in regression are squared error $\mathcal{L}(\hat{f}(X_i), Y_i) = (\hat{f}(X_i) - Y_i)^2$ and absolute error $\mathcal{L}(\hat{f}(X_i), Y_i) = |\hat{f}(X_i) - Y_i|$, etc. For classification, Y a factor/categorical variable (i.e., taking discrete values, such as in binary classification, $Y \in \{0, 1\}$). Examples Y in classification problems include whether an email is a spam, whether a patient has cancer, and whether a customer will purchase a product, etc. Commonly used loss functions in classification problems include the zero-one loss

$$\mathcal{L}(\hat{f}(X_i), Y_i) = 1\{Y_i \neq \hat{f}(X_i)\} = \begin{cases} 1, & \text{if } Y_i \neq \hat{f}(X_i) \\ 0, & \text{otherwise.} \end{cases}$$

The sample data $\mathcal{D}$ should come from a broader population and we hope that the sample could reasonably represent the entire population. For example, the sample data may come from a well-designed survey with which we could understand the whole population well. Mathematically, we are assuming that both the sample data $\mathcal{D}$ and the population data are the **SAME** probabilistic data generating process/distribution. Therefore, we could extract information from the sample data to reason about the data generating process and the whole population. This rationale is called **generalization** in supervised learning.

For this document, let us consider the following general supervised learning problem:

Algorithm 1 Supervised Learning Framework

```
Input: A sample training data set $\mathcal{D} := \{Y_i, X_{ij} : 1 \le i \le n, 1 \le j \le d\}$. Here, the outcome variable $Y$ could be either discrete or continuous.
Goal: A model fitted on $\mathcal{D}$, $\hat{f}(\cdot)$, such that the generalization error $\mathbb{E}[\mathcal{L}(Y, \hat{f}(X))]$ is minimized, where $\mathcal{L}(\cdot, \cdot)$ is the loss/error function and the expectation is taken with respect to the “true" distribution that generates the sample data and the population data.
Procedure:
  Define your models, e.g., linear regression $y = \beta_0 + \boldsymbol{\beta}' \cdot \mathbf{x}$.
  Define your loss/error functions $\mathcal{L}(\cdot, \cdot)$. In most cases, the loss function $\mathcal{L}(Y, \hat{f}(X))$ takes the forms of $(Y - \hat{f}(X))^2$ (squared-/$L^2$ loss), $|Y - \hat{f}(X)|$ (absolute value/$L^1$ loss), $1\{Y \neq \hat{f}(X)\}$ (0-1 loss) and $-Y_i \log(\hat{g}(X_i)) - (1 - Y_i) \log(1 - \hat{g}(X_i))$ (cross-entropy/log loss), where $\hat{g}(X_i)$ is the predicted probability that $Y_i = 1$, i.e., $\hat{\mathbb{P}}[Y_i = 1 | X_i]$.
  Pick your optimizer(s), such as OLS $\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$, or gradient descent.
  Fit your model using your data $\mathcal{D}$ on CPUs/GPUs/Clusters.
  Select the optimal model of your choice.
```

As is clear from the above problem formulation, our goal is NOT to build a model that minimizes the prediction error on any specific data set (even not the validation or testing sets). Instead, supervised learning seeks to minimize the prediction error on the entire population (even beyond the current population, please think about this from a probabilistic perspective).

Finding the “optimal" $\hat{f}(\cdot)$ may involve selecting the best one from different classes of models:

- **Parametric Model:** Assume the underlying distribution of the error term and the functional form of $f(\cdot)$ can be characterized by a few parameters, so the number of parameters is fixed. Example: $y = \beta_0 + \boldsymbol{\beta}' \cdot \mathbf{x}$ (i.e., linear regression).

- **Non-parametric Model:** Do not assume distributions and the functional form of $f(\cdot)$, so the number of parameters grow with the training data size n. Example: k-Nearest Neighbors.

- **Semi-parametric Model:** Some parts have fixed parameters, whereas some parts do not. Example: Cox Proportional-Hazards Model.

1.2.1 Model Selection and Evaluation

Training-Validation-Testing Triplet

Algorithm 2 Training-Validation-Testing Triplet

```
Step 1: Data Split.
1: Randomly split the data $\mathcal{D}$ into 3 non-overlapping subsets: $\mathcal{D}_{tr}$ for training (i.e., to fit a
   machine learning model), $\mathcal{D}_{va}$ for validation (to pick up the best model), and $\mathcal{D}_{te}$ for testing
   (to evaluate the selected model). Usually most of the data should be put to $\mathcal{D}_{tr}$ and a
   smaller portion should be allocated to $\mathcal{D}_{va}$ or $\mathcal{D}_{te}$.
Step 2: Training.
2: Using the training set $\mathcal{D}_{tr}$, train a model for each candidate in $F$, which we denote as
   $\hat{\mathcal{F}} = \{\hat{f}_1, \hat{f}_2, ..., \hat{f}_L\}$.
Step 3: Validation.
3: Use the validation set $\mathcal{D}_{va}$ to estimate the generalization error of each model $\hat{f}_i \in \hat{\mathcal{F}}$:
       $\mathbb{E}[\mathcal{L}(Y, \hat{f}_i(X))] \approx \hat{e}_i = \frac{1}{|D_{va}|} \sum_{j \in D_{va}} \mathcal{L}(Y_j, \hat{f}_i(X_j))$
   where $|\mathcal{D}_{va}|$ is the sample size of the validation data set $\mathcal{D}_{va}$.
4: Select the model with the smallest generalization error, which we denote as $\hat{e}_* :=$
   $\min\{\hat{e}_1, \hat{e}_2, ..., \hat{e}_L\}$, and the selected model is denoted as $\hat{f}_*(\cdot)$:
       $\hat{f}_*(\cdot) = \text{argmin}_{\hat{f}(\cdot) \in \hat{\mathcal{F}}} \frac{1}{|\mathcal{D}_{va}|} \sum_{j \in \mathcal{D}_{va}} l(Y_j, \hat{f}(X_j))$
Step 4. Testing.
5: Use the testing data set $\mathcal{D}_{te}$ to evaluate the generalization error of the selected model $\hat{f}_*(\cdot)$:
       $e\hat{r}r_* = \frac{1}{|\mathcal{D}_{te}|} \sum_{j \in \mathcal{D}_{te}} \mathcal{L}(Y_j, \hat{f}_*(X_j))$
   where $|\mathcal{D}_{te}|$ is the sample size of the testing set $\mathcal{D}_{te}$.
6: Then, $e\hat{r}r_*$ is an unbiased estimation of the true generalization error for the “best" model
   in $\mathcal{F}$.
```

In supervised learning, a key problem is to effectively and efficiently select a prediction model from a class with the smallest **generalization error** using a data set at hand $\mathcal{D} := \{Y_i, X_{ij} : 1 \leq i \leq n, 1 \leq j \leq d\}$. The set of candidate models is denoted as $\mathcal{F} := \{f_1(\cdot), f_2(\cdot), ..., f_L(\cdot)\}$[1]. The loss function is denoted as $\mathcal{L}(\cdot, \cdot)$. The core idea of the train-validate-test triplet approach is that using the data following the same distribution as the population but not overlapping with the data used to train the model could help estimate the unbiased generalization error. We summarize the entire train-validate-test procedure as Algorithm 2.

One should also note that $\hat{e}_*$ is an **underestimate** of the generalization error because $\mathbb{E}[\min\{Z_1, Z_2\}] \leq \min\{\mathbb{E}[Z_1], \mathbb{E}[Z_2]\}$. If one only cares about selecting the optimal model, but not the performance of the optimal model, we only need to split the data into two parts, training and validation so that the validation-error minimization model can be selected.

[1] We implicitly assume the model class is finite, which is what a computer system can handle.

Cross Validation

An alternative approach to systematically select a supervised learning model and evaluate its generalization error is $k-$fold cross validation, illustrated as Figure 1.1.. One key drawback of the training-validation-testing triplet is that it requires a lot of data. Cross-validation, however, helps reduce the amount of data needed to select the optimal model and estimate the generalization error with the same level of statistical power by combining the validation and training data sets.

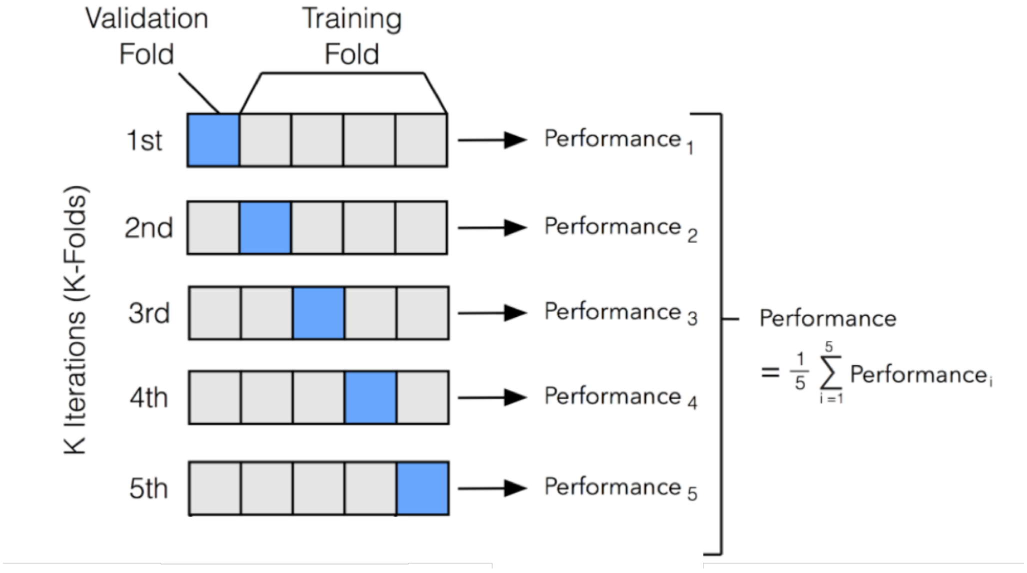


Figure 1.1: k-Fold Cross-Validation

The key idea of cross-validation is that we split the data into different parts, and use some parts to fit the model and some parts to validate the model. Furthermore, we repeat the above process the other way around by switching the training and validation sets, and aggregate the validation results together by taking the average to select the best model.

As with the training-validation-testing triplet, the data set at hand is denoted as $\mathcal{D} := \{Y_i, X_{ij} : 1 \leq i \leq n, 1 \leq j \leq d\}$, the set of candidate models is denoted as $\mathcal{F} := \{f_1(\cdot), f_2(\cdot), ..., f_L(\cdot)\}$, and the loss function is denoted as $\mathcal{L}(\cdot, \cdot)$. The $k-$fold cross validation procedure is performed as Algorithm 3.

An advantage of the cross-validation approach over the train-validate-test approach is that it has a smaller variances since the performance evaluation produced by cross-validation is averaged across errors of different rounds. There are some trade-offs to select the number of iterations/folds for cross-validation, k. In general, if k is chosen to be very large, the bias will be small, whereas the variance will be large, and the computation time is very long.[2] If k is small, the computation time is short, the variance is small, but the bias may be large. In practice, the choice of k depends on the data set size. For a large data set, even 3-Fold Cross-Validation will work very well. For a very sparse data set, we may need to use the leave-one-out method (i.e., $k = n$; so for each fold, train the model on $n - 1$ data points and perform testing on the

[2] We will discuss what bias and variance are in Section 1.3.

Algorithm 3 TRAINING-VALIDATION-TESTING TRIPLET

```
Step 1: Data Split.
  Randomly split the data $\mathcal{D}$ into 2 non-overlapping subsets: $\mathcal{D}_{tr}$ for training (i.e., to fit and cross-validate a machine learning model), and $\mathcal{D}_{te}$ for testing (to evaluate the selected model). Usually most of the data should be put to $\mathcal{D}_{tr}$ and a smaller portion should be allocated to $\mathcal{D}_{te}$.
  $k-$Fold Training Data. Randomly split the training data $\mathcal{D}_{tr}$ into $k$ parts of equal size, which we denote as $\mathcal{D}^1, \mathcal{D}^2, ..., \mathcal{D}^k$.
Step 2: Cross-Fitting.
  For each model in $l \in \mathcal{F}$, train this model on $k-1$ parts of the training data $\{\mathcal{D}^1, \mathcal{D}^2, ..., \mathcal{D}^{i-1}, \mathcal{D}^{i+1}, ..., \mathcal{D}^k\}$, and estimate the generalization error on the remaining part $\mathcal{D}^i$, denoted as $\hat{e}_l^i$. The average generalization error of model $l$ is therefore
      $$\hat{e}_l = \frac{1}{k}\sum_{i=1}^{k} \hat{e}_l^i$$
Step 3: Model Selection.
  Select the model with the smallest average generalization error, which we denote as $\hat{e}_{i^*} = \min\{\hat{e}_1, \hat{e}_2, ..., \hat{e}_L\}$. Retrain model $f_{i^*}(\cdot)$ on the entire training set $\mathcal{D}_{tr}$, which we obtain $\hat{f}_*^{cv}(\cdot)$.
Step 4. Testing.
  Finally, use the testing data set $\mathcal{D}_{te}$ to evaluate the generalization error of the selected model $\hat{f}_*^{cv}(\cdot)$, which we denote as $e\hat{r}r_*^{cv}$.
  $e\hat{r}r_*^{cv}$ is an unbiased estimation of the true generalization error for the "best" model in $\mathcal{F}$. In theory, if the sample size of the training set $|\mathcal{D}_{tr}|$ is sufficiently large, the smallest average generalization error of cross validation, $\hat{e}_{i^*}$ will also converge to the true generalization error.
```

remaining one; see Figure 1.2) to train on as many examples as possible. The common choice of k is usually set between 5 and 10.

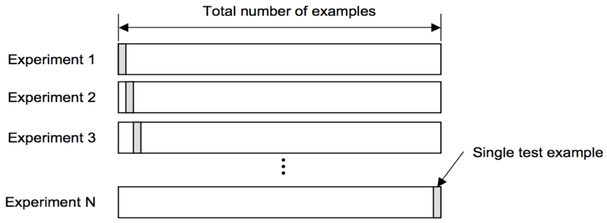


Figure 1.2: Leave-One-Out Cross-Validation ($k = n$)

If you dealing with time-series data, there may be some intertemporal correlations in your data. In this case, it is suggested that you use the rolling CV or blocked CV, as we illustrate in Figure 1.3.

In the traditional statistics and econometrics literature, model selection is performed through finding the mapping function between training and testing standard errors and adjusting the in-sample error to reflect the out-of-sample errors. Typical methods include adjusted R^2 for linear regression, Mallow's C_p, AIC and BIC. You are advised to read Hastie et al. (2009) for detailed derivations of these methods, but we only emphasize their comparisons with cross-validation:

- **Advantage of CV:** (i) When the sample size is large, the selected model (almost) guaran-

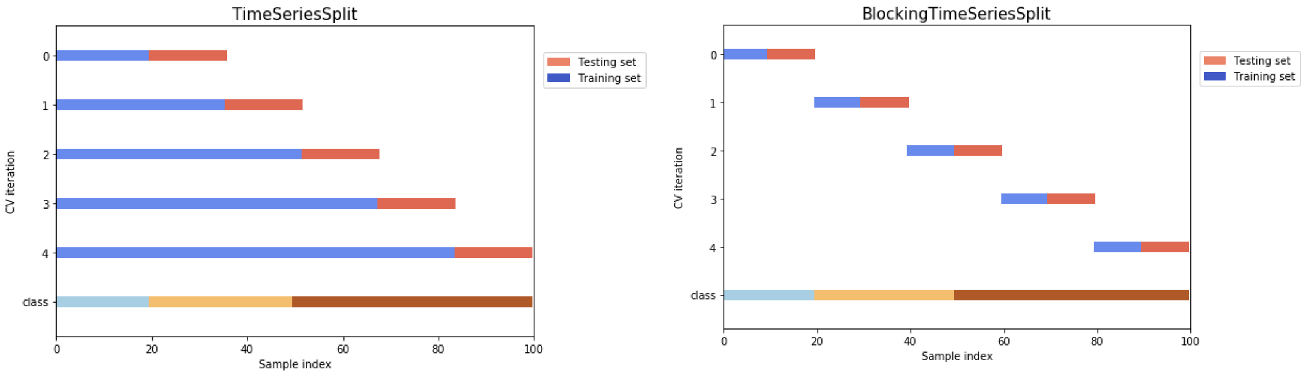


Figure 1.3: Rolling CV and Blocked CV

teed to be optimal. (ii) The model family $\mathcal{F}$ can be very general, even without close-form standard errors or likelihood functions.

- **Disadvantage of CV:** (i) It is computationally inefficient (because it tries to exploit the data as much as possible). (ii) It applies to judging the prediction performance of the model, but not the causal estimation capability.

As an end note of this section, we emphasize that cross-validation has become an increasing important idea in modern (applied) econometrics. For example, you need to do sample-splitting and cross-fitting (i.e., cross-validation) for double-machine-learning (DML) type of estimators, with the cross-validation errors well documented and analyzed in the prediction stage of your model. As another example, cross-validation has been used to perform model selection in structural estimation. The Python demonstration of cross-validation can be found here.

1.3 Bias-Variance Trade-off

We are now ready to characterize, in my opinion, the most important and fundamental issue in supervised learning: the bias-variance trade-off. This trade-off is so fundamental and prevalent that it offers the core actionable insights on how we could reduce the total generalization error of a supervised model.

We first present the famous decomposition of generalization error into three terms: bias, variance, and irreducible noise, which highlights the intrinsic trade-off between bias and variance. We consider a data sample $\mathcal{D} = \{Y_i, X_{ij} : 1 \leq i \leq n, 1 \leq j \leq d\}$. For notational simplicity and without loss of generality, we make a functional assumption on the “true" data generating process (DGP) that, for each i, $Y_i = f(X_i) + \epsilon_i$, where $f(\cdot)$ is a real-valued function and ϵ_i has mean zero ($\mathbb{E}[\epsilon_i] = 0$). For a new data point in the general population, (X, Y), it also satisfies this functional form: $Y = f(X) + \epsilon$, where X follows the same distribution as X_i and ϵ follows from the same distribution as ϵ_i. Based on the training sample $\mathcal{D}$, we train a model $\hat{f}(\cdot, \mathcal{D})$, where we explicitly specify the dependence of the model on the training set $\mathcal{D}$. Recall that our goal is to find $\hat{f}(X, \mathcal{D})$ that approximates the true data generating function $f(X)$ as close as possible for all X in the population, measured by a certain loss function $\mathcal{L}(\cdot, \cdot)$. In this section, we focus on the squared loss $\mathcal{L}(Y, \hat{f}(X, \mathcal{D})) = (Y - \hat{f}(X, \mathcal{D}))^2$ and the results can be generalized to other loss functions.

A core result in machine learning is, regardless of how we train the model $\hat{f}(\cdot, \mathcal{D})$, the following decomposition of the squared loss for an unseen data $(X, Y) \notin \mathcal{D}$ holds:

$$\underbrace{\mathbb{E}_{\mathcal{D}}[(Y - \hat{f}(X, \mathcal{D}))^2]}_{\text{Error Conditioned on } X} = \underbrace{(\mathbb{E}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] - f(X))^2}_{\text{Bias}} + \underbrace{\mathbb{E}_{\mathcal{D}}[(\mathbb{E}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] - \hat{f}(X, \mathcal{D}))^2]}_{\text{Variance}} + \underbrace{\mathbb{V}(\epsilon)}_{\text{Noise}} \tag{1.1}$$

where $\mathbb{V}(\cdot)$ refers to the taking variance operator and the expectation $\mathbb{E}_{\mathcal{D}}(\cdot)$ is taken with respect to the training set $\mathcal{D}$ sampled from underlying “true" data generating process of (X, Y). The proof of (1.1) can be found in this Wikipedia Page and Hastie et al. (2009). By taking the expectation over the distribution of X for (1.1), the generalization error can be decomposed as follows:

$$\underbrace{\mathbb{E}_{X,\mathcal{D}}[(Y - \hat{f}(X, \mathcal{D}))^2]}_{\text{MSE}} = \mathbb{E}\Big\{\text{Bias}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] + \text{Variance}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})]\Big\} + \mathbb{V}(\epsilon), \tag{1.2}$$

where $\text{Bias}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] := (\mathbb{E}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] - f(X))^2$ is the bias and $\text{Variance}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] := \mathbb{E}_{\mathcal{D}}[(\mathbb{E}_{\mathcal{D}}[\hat{f}(X, \mathcal{D})] - \hat{f}(X, \mathcal{D}))^2]$ is the variance.

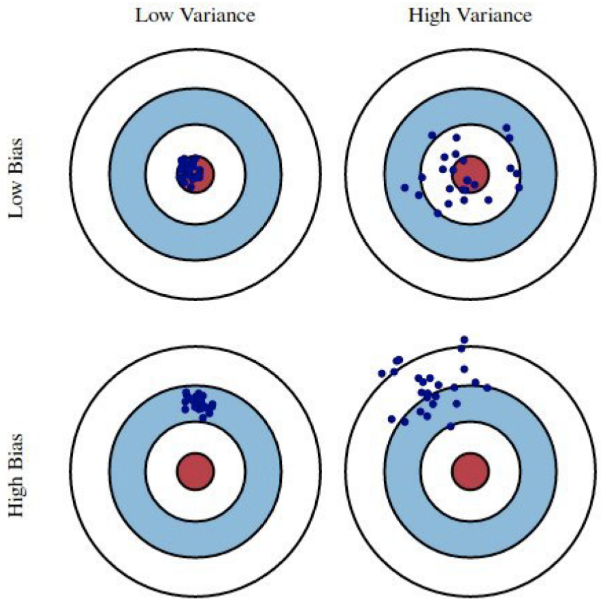


Figure 1.4: Illustration of Bias and Variance

The bias, variance, and irreducible noise terms have the following interpretations:

- **Bias** is the inherent error with your model caused by, e.g., simplifying assumptions, even with infinitely many training data. As is clear from its definition, the bias term measures how far away is the fitted model $\hat{f}(\cdot, \mathcal{D})$ from the "true model" $f(\cdot)$ under the underlying data generating process. It essentially captures the degree of underfitting for the model trained on $\mathcal{D}$.

- **Variance** is the amount of the change the model will have if trained on a different data set. As is clear from its definition, the variance term measures how far away is the fitted outcome $\hat{f}(X, \mathcal{D})$ from its mean under the true data generating process. It essentially captures the degree of overfitting for the model.

- **Noise** is the variance of the intrinsic error ϵ, which is independent of the choice of model.

A pictorial illustration of the bias and variance can be found in Figure 1.4. From the bias-variance decomposition formula (1.1), we know that both bias and variance are sensitive to the model $\hat{f}(\cdot, \mathcal{D})$. Figure 1.5 demonstrate the trade-off between bias and variance under different model complexities. A more complex model could have more flexibility to fit the “true" model (also called the ground truth in the machine learning community) so that the bias will decrease in this case. Meanwhile, it may over-fit the training data so that the variance of the model may be too high. On the other hand, a less complex model would not be that sensitive to changes in the training set (low variance), but may be too simple to capture the true model (high bias). Figure 1.6 depicts this bias-variance trade-off and shows how total generalization error changes with model complexity. As we can see, the total generalization error is minimized when the model complexity is at a moderate sweet spot.

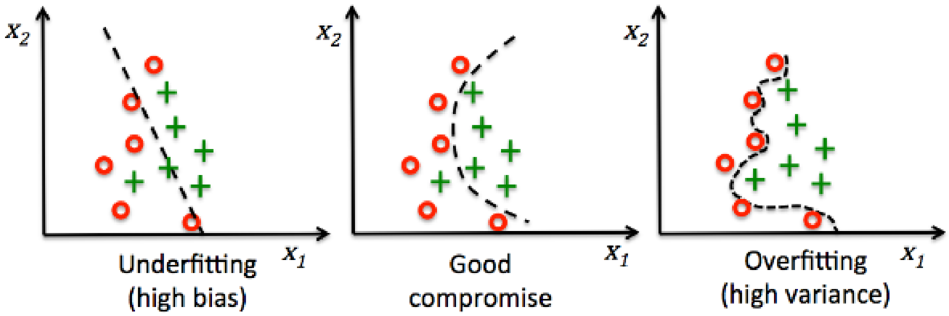


Figure 1.5: Bias-Variance Trade-Off and Model Complexity

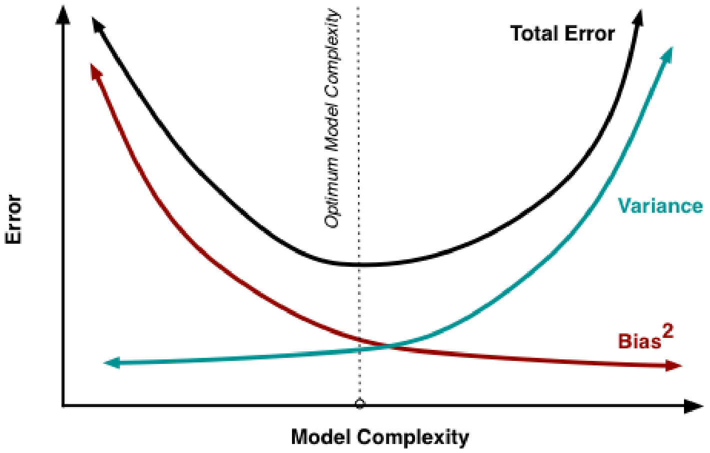


Figure 1.6: Generalization Error and Model Complexity

In your first homework, you will have the opportunity to code up the bias-variance trade-off for the family of linear models with higher polynomial orders.

1.4 Regularization

Regularization refers to a systematic approach to address the overfitting problem of overly complex models by introducing a penalty term in the objective function that penalizes an overly complex model. Specifically, given a training data set $\mathcal{D} = \{Y_i, X_{ij} : 1 \leq i \leq n, 1 \leq j \leq d\}\}$, a family of models $\mathcal{F}$, and the loss function $\mathcal{L}(\cdot, \cdot)$, the (unregularized) training is to find a model $\hat{f}(\cdot)$ that minimizes the **in-sample error**.

$$\hat{f}(\cdot) = \text{argmin}_{f(\cdot)\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(Y_i, f(X_i))$$

Under regularization, we introduce a regularizer, or a penalty function associated with the model $R(f) \geq 0$. In particular, if $f(\cdot)$ is more complex, $R(f)$ will become larger. The model complexity is represented differently for different models. For a linear regression or logistic regression, a more complex model means the number of effective features p is larger. For k-NN, a more complex model means k is small (see Section 1.5 below). We will discuss what it means by model complexity for other machine learning models as we proceed to discuss them.

The regularized model fitting is performed as follows:

$$\hat{f}(\cdot, \lambda) = \text{argmin}_{f(\cdot)\in\mathcal{F}} [\underbrace{\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(Y_i, f(X_i))}_{\text{Loss}} + \underbrace{\lambda \cdot R(f)}_{\text{Penalty}}]$$

where the hyper parameter $\lambda \geq 0$. The regularization term $\lambda \cdot R(f)$ penalizes overly complex models. In particular, the parameter λ trades off bias (under-fitting) and variance (over-fitting). The larger the λ, the higher weight we put on avoiding over-fitting. Let us now introduce some commonly used regularized regressions:

- **Lasso Linear Regression**

  $$\hat{\beta} = \text{argmin}_{\beta} \underbrace{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \beta_0 - \sum_{j=1}^{d}\beta_j X_{ij})^2}_{\text{Loss}} + \underbrace{\lambda \sum_{j=1}^{d} |\beta_j|}_{\text{Penalty}}$$

  In lasso regression, the penalty is called L^1 penalty. See Tibshirani (1996) for the original lasso paper.

- **Ridge Linear Regression**

  $$\hat{\beta} = \text{argmin}_{\beta} \underbrace{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \beta_0 - \sum_{j=1}^{d}\beta_j X_{ij})^2}_{\text{Loss}} + \underbrace{\frac{\lambda}{2} \sum_{j=1}^{d} |\beta_j|^2}_{\text{Penalty}}$$

  In ridge regression, the penalty is called L^2 penalty.

- **Elastic Net Linear Regression**

  $$\hat{\beta} = \text{argmin}_{\beta} \underbrace{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \beta_0 - \sum_{j=1}^{d}\beta_j X_{ij})^2}_{\text{Loss}} + \underbrace{\lambda \sum_{j=1}^{d} \left[\frac{1}{2}(1-\alpha)|\beta_j|^2 + \alpha|\beta_j|\right]}_{\text{Penalty}}$$

  Clearly, the elastic net linear regression is a weighted combination of lasso ($\alpha = 1$) and ridge ($\alpha = 0$) regressions. You may refer to this Wiki Page and Zou and Hastie (2005) for additional discussions of the elastic net model.

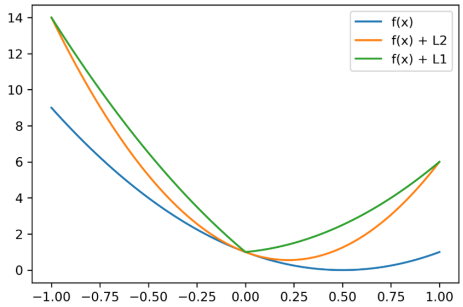
- **Lasso Logistic Regression.** We can also regularize logistic regression by subtracting L^1 and L^2 penalties to the maximum likelihood estimation:

$$\hat{\beta} = \text{argmax}_{\beta} \underbrace{\sum_{i=1}^{n} \left(Y_i \log \left(\frac{\exp(\beta_0 + \sum_{j=1}^{d} \beta_j X_{ij})}{1 + \exp(\beta_0 + \sum_{j=1}^{d} \beta_j X_{ij})} \right) + (1 - Y_i) \log \left(\frac{1}{1 + \exp(\beta_0 + \sum_{j=1}^{d} \beta_j X_{ij})} \right) \right)}_{\text{Log-Likelihood}} - \underbrace{\lambda \sum_{j=1}^{d} |\beta_j|}_{\text{Penalty}}$$

- **Ridge Logistic Regression.**

$$\hat{\beta} = \text{argmax}_{\beta} \underbrace{\sum_{i=1}^{n} \left(Y_i \log \left(\frac{\exp(\beta_0 + \sum_{j=1}^{d} \beta_j X_{ij})}{1 + \exp(\beta_0 + \sum_{j=1}^{d} \beta_j X_{ij})} \right) + (1 - Y_i) \log \left(\frac{1}{1 + \exp(\beta_0 + \sum_{j=1}^{d} \beta_j X_{ij})} \right) \right)}_{\text{Log-Likelihood}} - \underbrace{\frac{\lambda}{2} \sum_{j=1}^{d} |\beta_j|^2}_{\text{Penalty}}$$

Finally, we briefly discuss the properties of lasso (L^1) and ridge (L^2) regularizations. From their formulation, we know that lasso and ridge will shrink the estimated coefficients $\hat{\beta}$ to 0. Only the most explanatory features will be retained with regularized regressions. In particular, lasso typically yields much fewer non-zero entries of $\hat{\beta}$ than ridge or OLS. Thus, lasso shrinks $\hat{\beta}$ to 0 more sharply. For ridge regression, it shrinks $\hat{\beta}$'s to zero more softly. As is shown by Figure 1.7 (where $L1$ refers lasso regression and $L2$ refers to ridge regression), with $L2$ regularizer, the variable that minimizes the function $f(x) + \alpha|x|^2$ is closer to 0 than the minimizer of $f(x) = (2x-1)^2$. With $L1$ regularizer, the minimizer of $f(x) + \alpha|x|$ is directly shrunk to 0.



$f(x) = (2x-1)^2$
$f(x) + L2 = (2x-1)^2 + \alpha x^2$
$f(x) + L1 = (2x-1)^2 + \alpha|x|$

Figure 1.7: L^1 vs. L^2 Regularizations

We usually use the train-validate-test triplet or cross-validation to fine tune the parameter λ. The demonstration for regularization can be found here.

1.5 Supervised Learning Models

In this subsection, we introduce k-nearest-neighbors (k-NN) and tree-based models.

1.5.1 k-NN Model

The k-Nearest Neighbors (k-NN) classifier model estimates $\mathbb{P}[Y = 1|X]$ by local averaging. The key assumption of k-NN is that the distance in the feature space reliably reflects similarities in the outcomes.

The idea of k-NN hinges upon the law of large numbers. Specifically, to estimate $g(X) = \mathbb{P}[Y = 1|X]$, it suffices to have an independent sample of labels $\{y_1, y_2, ..., y_k : y_i \in \{0, 1\}\}$ where the feature of all the data points is fixed at X. The strong law of large numbers implies that, when k is large,

$$g(X) = \mathbb{P}[Y = 1|X] = \mathbb{E}[Y|X] \approx \frac{\sum_{i=1}^{k} y_i}{k}$$

In practice, however, we are unable to construct such an independent sample because there may not be sufficiently many data points in $\mathcal{D}$ with the feature being X. To address this issue, we assume that the distance (in the feature space) reliably reflects similarities in the (distribution of) outcomes. Specifically, we assume the feature space is associated with a distance function measuring the distance between two feature vectors X_1 and X_2 (sometimes also called norm), which we denote as $d(X_1, X_2)$. In most applications, we use the Euclidean distance (sometimes also called the Euclidean norm or the L^2-norm) most often. For two feature vectors $X_1 = (X_{11}, X_{12}, ..., X_{1d})$ and $X_2 = (X_{21}, X_{22}, ..., X_{2d})$, their Euclidean distance is given by

$$d(X_1, X_2) = ||X_1 - X_2||_2 = \sqrt{\sum_{j=1}^{d}(X_{1j} - X_{2j})^2},$$

and the L^p norm is given by

$$d(X_1, X_2) = ||X_1 - X_2||_p = \left(\sum_{j=1}^{d}|X_{1j} - X_{2j}|^p\right)^{\frac{1}{p}}.$$

Given the training set $\mathcal{D} := \{Y_i \in \{0, 1\}, X_{ij} : 1 \leq i \leq n, 1 \leq j \leq d\}$, the k-NN model predicts the outcome Y of a new data point with feature X by averaging the outcomes of the data points in the training set whose features are closest to X. Specifically, we can write out the k-NN algorithm as follows:

Algorithm 4 k-Nearest-Neighbors Classifier

```
Compute Distance.
  1: Compute the distance between $X$ and each data point in $\mathcal{D}$, $d_i := d(X, X_i)$ for all $i = 1, 2, ..., n$.
Identify Neighbors
  2: Find $k$ data points in $\mathcal{D}$ who are closest to $X$, i.e., who have the smallest $d_i$, which we denote as $\hat{\mathcal{D}}_k(X)$.
Take Average
  3: Denote the number of data points in $\hat{\mathcal{D}}_k(X)$ which have a positive outcome as $\hat{k}_1(X)$.
  4: We estimate that the probability that $Y = 1$ is $\hat{g}(X) = \hat{P}[Y = 1|X] = \frac{\hat{k}_1(X)}{k}$.
Classify by Majority Vote
  5: We take a majority vote to predict the outcome $Y$, i.e.,
```

$$\hat{Y} = \hat{f}(X) = \begin{cases} 1, \text{ if } \frac{\hat{k}_1(X)}{k} > 0.5 \\ 0, \text{ if } \frac{\hat{k}_1(X)}{k} < 0.5 \\ \text{either outcome with equal probability, if } \frac{\hat{k}_1(X)}{k} = 0.5 \end{cases}$$

The above model fitting procedure of the k-NN model (Algorithm 4) can be illustrated with Figure 1.8, where we want to predict the outcome of the data with the star feature. It is clear from the above figure that the prediction outcome depends on the value of k. If $k = 3$, the model predicts $\hat{\mathbb{P}}[Y = A|X] = 1/3$ and $\hat{Y} = B$. If $k = 6$, the model predicts $\hat{\mathbb{P}}[Y = A|X] = 2/3$

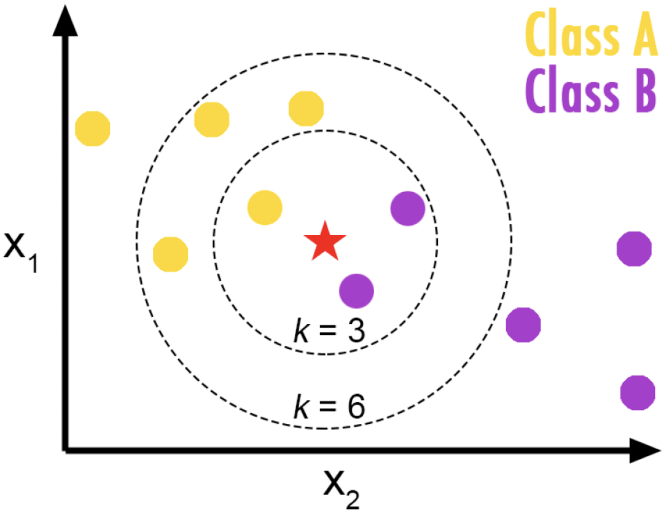


Figure 1.8: k-NN Classifier

and $\hat{Y} = A$. In the special case where $k = n$, the k-NN model makes the same prediction that Y is the more frequent outcome of the training set, regardless of the feature X. If $k = 1$, the k-NN model predicts the outcome of a new data point using that of the closest data point in the training set.

We first discuss an important data pre-processing step of k-NN: Standardization. More specifically, to use the k-NN models for prediction, we should first standardize the features by subtracting the sample mean and divided by the sample standard deviation:

$$\tilde{X}_{ij} = \frac{X_{ij} - \bar{X}_j}{\hat{\sigma}_j}$$

where $\bar{X}_j = \frac{1}{n}(\sum_{i=1}^n X_{ij})$ is the sample average of the training set and $\hat{\sigma}_j = \sqrt{\frac{1}{n-1}\sum_{i=1}^n (X_{ij} - \bar{X}_j)^2}$ is the sample standard deviation of the training set. The rationale for such standardization is that the distances between different features are very sensitive to the scale/unit we use for each feature. Without standardization, k-NN may produce some misleading results as illustrated in Figure 1.9. It is clear from the above figure that, without scaling/standardization, the first feature will have a much more substantial impact than the second feature on the result of a k-NN model, although such imbalance is just a matter of scale but may not reliably reflect the true data generating process.

To conclude, we have a few remarks regarding the k-NN model. First, the hyper-parameter k essentially captures the complexity/flexibility of the model. With a small k, the decision boundary is (overly) flexible, while with a large k, it is not sufficiently flexible. The best k is usually determined by cross-validation. The time complexity is usually an issue to implement k-NN. Whereas the training time complexity is just $\mathcal{O}(1)$, the classification time complexity is $\mathcal{O}(ndk)$, which is slow when d is large. There are some implementation guidelines for k-NN:

- In general, when p is larger, L^p norm penalizes one bad dimension more. So choose your distance metric suitable for your use case.
- Remember to standardize your data.

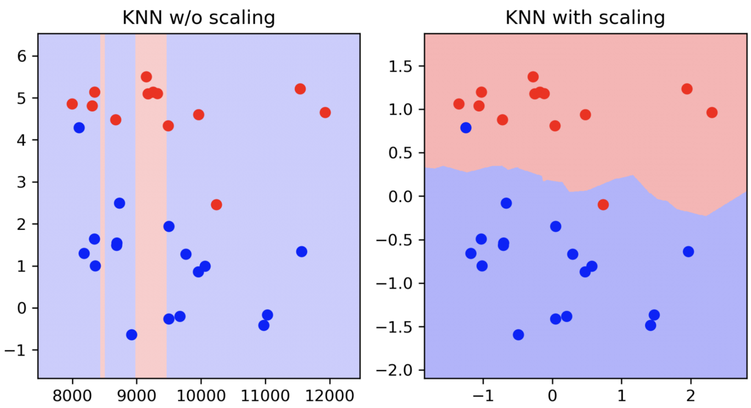


Figure 1.9: k-NN Classifier

- When d is large, try some feature selection method such as Principal Component Analysis (PCA).

- Choose k based on cross-validation. See this demo.

For completeness, we give the algorithm of k-NN regressor in Algorithm .

Algorithm 5 k-NEAREST-NEIGHBORS REGRESSOR

```
Compute Distance.
  Compute the distance between $X$ and each data point in $\mathcal{D}$, $d_i := d(X, X_i)$ for all $i = 1, 2, ..., n$.
Identify Neighbors
  Find $k$ data points in $\mathcal{D}$ who are closest to $X$, i.e., who have the smallest $d_i$, which we denote as $\hat{\mathcal{D}}_k(X)$.
Take (Conditional) Average
  Denote the number of data points in $\hat{\mathcal{D}}_k(X)$ which have a positive outcome as $\hat{k}_1(X)$.
  We estimate that the conditional expectation of $Y$ as
      $\hat{f}(X) = \hat{\mathbb{E}}[Y|X] = \frac{1}{k} \sum_{i \in \hat{\mathcal{D}}_k(X)} Y_i.$
```

1.5.2 Decision Tree

The Decision tree model partitions the input space into a set of rectangles, and then fit a simple model (like a constant) in each one. This process yields an overall model represented by a tree structure, with each leaf per region. Tree models where the target variable can take a discrete set of values are called classification trees; decision trees where the target variable can take continuous values (typically real numbers) are called regression trees.

We firstly consider regression trees. The data consists of p inputs and a response, for each of N observations: that is, (x_i, y_i) for $i = 1, 2, ..., N$, with $x_i = (x_{i1}, x_{i2}, ..., x_{ip})$. The algorithm needs to automatically decide on the splitting variables and split points, and also the shape of the tree. Suppose first that we have a partition into M regions $R_1, R_2, ..., R_M$, and we model the response as a constant c_m in each region:

$$f(x) = \sum_{m=1}^{M} c_m I(X \in R_m). \tag{1.3}$$

If we adopt as the criterion minimization of the sum of squares $\sum (y_i - f(x_i))^2$, the best $\hat{c}_m$ is the average of y_i in region R_m:

$$\hat{c}_m = ave(y_i | x_i \in R_m). \tag{1.4}$$

However, finding the best binary partition in terms of minimum sum of squares is computationally infeasible. We proceed with a greedy algorithm. Starting with all of the data, consider a splitting variable j and split point s, and define the pair of half-planes:

$$R_1(j,s) = \{X | X_j \leq s\} \quad \text{and} \quad R_2(j,s) = \{X | X > s\}. \tag{1.5}$$

Then we seek the splitting variable j and split point s that solve:

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right] \tag{1.6}$$

For any choice j and s, the inner minimization is solved by:

$$\hat{c}_1 = ave(y_i | x_i \in R_1(j,s)) \quad \text{and} \quad \hat{c}_2 = ave(y_i | x_i \in R_2(j,s)) \tag{1.7}$$

For each splitting variable, the determination of the split point s can be done very quickly and hence by scanning through all of the inputs, determination of the best pair (j,s) is feasible.

After finding the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions, Then the process is repeated on all of the resulting regions.

Tree size is a tuning parameter governing the model's complexity, and the optimal tree size should be adaptively chosen from the data. One approach would be to split tree nodes only if the decrease in sum-of-squares due to the split exceeds some threshold. This strategy is too short-sighted, however, since a seemingly worthless split might lead to a very good split below it.

The preferred strategy is to grow a large tree T_0, stopping the splitting process only when some minimum node size (say 5) is reached. Then this large tree is pruned using cost-complexity pruning, which we now describe. We define a subtree $T \subset T_0$ to be any tree that can be obtained by pruning T_0, that is, collapsing any number of its internal (non-terminal) nodes. We index terminal nodes by m, with node m representing region R_m. Let $|T|$ denote the number of terminal nodes in T. Letting

$$\begin{aligned} N_m &= \#\{x_i \in R_m\}, \\ \hat{c}_m &= \frac{1}{N_m} \sum_{x_i \in R_m} y_i, \\ Q_m(T) &= \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 \end{aligned} \tag{1.8}$$

we define the cost complexity criterion

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|. \tag{1.9}$$

The idea is to find, for each α, the subtree $T_\alpha \subseteq T_0$ to minimize $C_\alpha(T)$. The tuning parameter $\alpha \geq 0$ governs the tradeoff between tree size and its goodness of fit to the data. Large values of α result in smaller trees T_α, and conversely for smaller values of α. As the notation suggests, with $\alpha = 0$ the solution is the full tree T_0. For each α one can show that there is a unique smallest subtree T_α that minimizes $C_\alpha(T)$. To find T_α we use weakest link pruning: we successively collapse the internal node that produces the smallest per-node increase in $\sum_m N_m Q_m(T)$, and continue until we produce the single-node (root) tree. This gives a (finite) sequence of subtrees, and one can show this sequence must contain T_α. Estimation of α is achieved by five- or tenfold cross-validation: we choose the value $\hat{\alpha}$ to minimize the cross-validated sum of squares. Our final tree is $T_{\hat{\alpha}}$. We can write out the regression tree algorithm as follows:

```
Algorithm 6 REGRESSION TREE
Grow a large tree $T_0$.
1: Find the best split and repeat the splitting process on each region.
2: Stop the splitting process when some minimum node size (say 5) is reached.
Cost-complexity pruning.
3: Define a subtree $T \subset T_0$ to be any tree that can be obtained by pruning $T_0$.
4: Define the cost complexity criterion as
       $$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha|T|. \qquad (1.10)$$
5: Use weakest link pruning to find the subtree $T_\alpha \subseteq T_0$ to minimize $C_\alpha(T)$ for each $\alpha$.
Estimate $\alpha$.
6: Use five- or tenfold cross-validation and choose the value $\hat{\alpha}$ to minimize the cross-validated
   sum of squares. The final tree is $T_{\hat{\alpha}}$.
```

If the target is a classification outcome taking values $1, 2, ..., K$, the only changes needed in the tree algorithm lie in the criteria for splitting nodes and pruning the tree. For regression we used the squared-error node impurity measure $Q_m(T)$ defined in equation (8), but this need to be changed for classification. In a node m, representing a region R_m with N_m observations, let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k) \tag{1.11}$$

the proportion of class k observations in node m. We classify the observations in node m to class $k(m) = argmax_k \hat{p}_{mk}$, the majority class in node m. Different measures $Q_m(T)$ of node impurity include the following:

$$\begin{aligned} \text{Misclassification error:} \quad & \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}. \\ \text{Gini index:} \quad & \sum_{k \neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}). \qquad (1.12) \\ \text{Cross-entropy or deviance:} \quad & -\sum_{k=1}^{K} \hat{p}_{mk} log \hat{p}_{mk}. \end{aligned}$$

To guide cost-complexity pruning, any of the three measures can be used, but typically it is the misclassification rate.

1.5.3 Random Forest

The Random Forest is an ensemble learning method for classification and regression that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random forests correct for decision trees' habit of overfitting to their training set.

The core idea behind random forests is to reduce the variance of bagged trees without increasing it excessively by decreasing the correlation between individual trees. This is done by selecting a subset of input variables randomly during the tree-growing process.

Random forest introduces two sources of randomness.

- Bagging: Each tree is grown upon a bootstrap sample of the training data.
- Random split: For each tree at each node, the best split is chosen from a random sample of m features, instead of all features.

Random forest substantially reduces the model variance/overfitting, and generally works well in practice as the benchmark model to start with.

Algorithm 7 Random Forest Algorithm

```
for $b = 1$ to $B$ do
  Draw a bootstrap sample $Z^*$ of size $N$ from the training data.
  Grow a random-forest tree $T_b$ to the bootstrapped data:
  repeat
    Select $m$ variables at random from the $p$ variables.
    Pick the best variable and split-point among the $m$.
    Split the node into two daughter nodes.
  until minimum node size $n_{\text{min}}$ is reached
end for
Output the ensemble of trees $\{T_b\}_1^B$.
Regression:
$f_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$
Classification:
Let $\hat{C}_b(x)$ be the class prediction of the $b$th random-forest tree.
$\hat{C}_{\text{rf}}^B(x) = \text{majority vote}\{\hat{C}_b(x)\}_1^B$
```

Details of Random Forests

Random forests differ slightly in their implementation for classification and regression tasks:

- For classification, each tree casts a vote for the predicted class, and the overall prediction is made by majority vote:

$$\hat{C}_{\text{rf}}^B(x) = \text{majority vote}\{\hat{C}_b(x)\}_1^B \tag{1.13}$$

- For regression, the predictions from each tree are averaged:

$$\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x) \tag{1.14}$$

The default values for the number of variables m considered at each split are:

- For classification: $m = \sqrt{p}$
- For regression: $m = p/3$

where p is the total number of variables. The minimum node sizes are 1 for classification and 5 for regression. These parameters can be tuned for each problem.

Out-of-bag (oob) samples are used to construct error estimates similar to cross-validation. For each observation $z_i = (x_i, y_i)$, only the trees for which z_i was not included in the bootstrap sample are used to construct its prediction. The oob error stabilizes as the number of trees B increases, allowing training to be terminated early.

Variable importance can be assessed in two ways:

1. By the total decrease in the split-criterion (e.g., Gini index) attributed to each variable over all trees.
2. By the decrease in oob prediction accuracy when the values of each variable are randomly permuted.

Proximity plots, obtained by multi-dimensional scaling of the proximity matrix, attempt to visualize which observations are "close" in the eyes of the random forest. However, their utility is questionable as they often look similar regardless of the data.

Variance of Averaged i.i.d. Random Variables

Consider averaging B i.i.d. (independent and identically distributed) random variables, each with a given variance σ^2. This average has a variance of $\frac{1}{B}\sigma^2$. However, if these variables have a positive pairwise correlation ρ, things change a bit. Assuming they are identically distributed but not necessarily independent, the variance of their average becomes:

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

As we increase B, the impact of the second term in the equation diminishes, leaving us predominantly with the first term. This correlation among the variables constrains the benefit we get from averaging.

Once we have grown B trees denoted by $\{T(x;\Theta_b)\}_{b=1}^{B}$, we can define the random forest predictor as:

$$f_{\text{rf}}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B} T(x;\Theta_b).$$

Out-of-Bag Error Estimation

In random forests, there is no need for a separate test set to get an unbiased estimate of the test set error. It is estimated internally, during the run, as follows:

Each tree is constructed using a different bootstrap sample from the original data. About one-third of the cases are left out of the bootstrap sample and not used in the construction of the k-th tree. Put each case left out in the construction of the k-th tree down the k-th tree to get a classification. In this way, a test set classification is obtained for each case in about one-third of the trees. At the end of the run, take the majority vote of these classifications to get the random forest prediction for each case. The out-of-bag (OOB) error estimate is the proportion of times that the random forest prediction is not equal to the true classification averaged over all cases. The OOB error estimate is almost identical to that obtained by N-fold cross-validation. Therefore, random forests do not require a separate test set for getting an unbiased estimate of the test error.

Bias-Variance Tradeoff in Random Forests

Like other ensemble methods, random forests achieve a reduction in variance by introducing randomization into the model building process, then combining the outputs of the resulting models. In the case of random forests, this randomization comes in two forms: the use of bootstrap samples to build each tree, and the random selection of features at each split.

The bias of a random forest is typically slightly higher than the bias of a single unpruned decision tree, but the variance of the random forest is much lower, due to the averaging of the trees. This is a classic example of the bias-variance tradeoff: the randomization introduced by random forests increases the bias but decreases the variance, resulting in an overall better model.

More specifically, for a random forest model $\hat{f}_{rf}(x)$ at a point x, the variance is:

$$Var[\hat{f}_{rf}(x)] = \rho(x)\sigma^2(x) \tag{1.15}$$

where $\rho(x)$ is the sampling correlation between any pair of trees and $\sigma^2(x)$ is the sampling variance of any single randomly drawn tree. The additional randomization in random forests (randomly selecting a subset of features at each split) decreases the correlation ρ between trees, which decreases the variance of the ensemble.

The bias of a random forest $\hat{f}_{rf}(x)$ is:

$$Bias[\hat{f}rf(x)] = E_Z[\hat{f}rf(x;Z)] - f(x) \tag{1.16}$$

where $f(x)$ is the true function. The additional randomization in random forests can slightly increase the bias compared to unpruned decision trees, because the best split feature might not be considered at a split if $m < p$. However, this increase in bias is usually more than compensated by the decrease in variance due to decorrelation.

1.5.4 Boosting Tree

Boosting and bagging are both ensemble learning techniques, which involve combining the predictions from multiple models to improve the overall performance. Bagging involves training multiple models in parallel, each on a different bootstrap sample of the data, and then averaging their predictions. Boosting, on the other hand, is a sequential process where each model is built based on the errors made by previous models.

Key Concepts of Boosting:

The fundamental idea behind boosting is to sequentially construct a strong learner (a model with high prediction accuracy) by fitting and aggregating a series of weak learners (models that perform slightly better than random guessing). Mathematically, if we denote the set of weak learners by $f_t(x)$, where t indexes the sequence of models, then the final boosted model can be expressed as:

$$F(x) = \sum_{t=1}^{T} \alpha_t f_t(x),$$

where α_t are coefficients that weigh the contribution of each weak learner, and T is the total number of weak learners in the ensemble.

Boosting Trees:

In the context of boosting trees, each tree is grown using the information from previously grown trees rather than fitting independently to the data. This means that each new tree in the sequence is specifically grown to reduce the residuals or errors made by the previous trees. The process can be described by the following steps:

1. Initialize the model with a constant value: $F_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$,
2. For each successive tree $t = 1, 2, \ldots, T$:
   (a) Compute the negative gradient of the loss function, $r_{it} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{t-1}}$,
   (b) Fit a tree to these residuals, $f_t(x)$,
   (c) Update the model: $F_t(x) = F_{t-1}(x) + \nu\alpha_t f_t(x)$,
3. The final model is $F_T(x)$.

Here, ν is the learning rate, which controls the step size in the gradient descent optimization and helps prevent overfitting.

XGBoost:

XGBoost, or eXtreme Gradient Boosting, is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. XGBoost is particularly popular on Kaggle and other machine learning competitions due to its performance and speed. XGBoost implements not only the gradient boosting algorithm but also includes regularization terms in the objective function, which helps to control overfitting:

$$\text{Obj} = \sum_{i=1}^{N} L(y_i, F(x_i)) + \sum_{t=1}^{T} \Omega(f_t),$$

where $\Omega(f_t)$ represents the regularization term, typically including both the number of leaves in the tree and the L2 norm of the weights. This addition makes XGBoost more robust and generally better performing than standard boosting.

Algorithm 8 Gradient Tree Boosting Algorithm

```
Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y, \gamma)$
for $m = 1$ to $M$, do
    compute
        $r_{im} = -[\frac{\partial L(L_i, f(x_i))}{\partial f(x_i)}]_{f=f_{m-1}}$
    Fit a regression tree to the targets $r_{im}$ giving terminal regions $R_{jm}, j = 1, 2, ..., J_m$
    for $j = 1, 2, ..., J_m$, do
        compute
            $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma$
    end for
    Update $f_m(s) = f_{m-1}(x) + \sum_{k=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$
end for
Output $\hat{f}(x) = f_M(x)$
```

2

Introduction to Deep Learning

——**Scribed by Shu Zhang and Xinyu Xu**

2.1 Outline

This section will contain following subsections:

- In Section 2.2, we introduce gradient descent to the readers.
- In Section 2.3, we introduce the basic neural nets and the simple method with respect to training a neural networks.
- In Section 2.4, we introduce the computations problem in deep learning, which is very important.

2.2 Gradient Deecent

Algorithm 1 introduces the general framework about how to train a supervised learning model. And Deep learning means the model family is Deep Neural Nets. For example, given the optimization as follows:

$$\hat{\theta} := \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(Y_i, f(X_i, \theta)) \tag{2.1}$$

where $f(\cdot)$ is the deep neural net which will be introduced detailedly in the next subsection. For the deep learning, the key way to obtain the suitable $\hat{\theta}$ is based on gradient descent method, which is be defined in Algorithm 9. where t_k denotes the learning rate. Different learning rates lead to different outcomes. If the learning rate is too large or too small, it can result in the final parameters not reaching the optimal solution. Choosing an appropriate learning rate is a crucial work in the training process of deep learning.

Algorithm 9 Gradient Descent

```
Guess $\theta^{(0)}$, set $k \leftarrow 0$
while $\|\nabla\mathcal{L}(\theta^{(k)})\| \geq \epsilon$ do
    $\theta^{(k+1)} = \theta^{(k)} - t_k \nabla\mathcal{L}(\theta^{(k)})$
    $k \leftarrow k + 1$
end while
return $\hat{\theta} = \theta^{(k)}$
```

Example: OLS. Suppose we want to estimate the linear parameter using gradient descent method, the process can be seen as follows:

- **Loss function:**
$$S(\beta) = \sum_{i=1}^{n} \left| y_i - \sum_{j=1}^{p} X_{ij}\beta_j \right|^2 = \|y - X\beta\|^2$$
- **Closed-form Solution:**
$$\hat{\beta} = (X^\top X)^{-1} X^\top y$$
- **Gradient Descent:**
$$w^{k+1} = w^k - \alpha_k X^\top (Xw^k - y)$$
- **Statistics convergence vs. Optimization convergence:**
  - **Convergence of estimator:** Is the estimator constructed by the loss function converging to the true underlying estimand? How fast? Bias and consistency?
  - **Convergence of optimization:** Will the optimization algorithm we use converge to the minimizer of the loss function given data? How fast?

Beside the original framework, there are many variants of gradient descent as follows:

- **Momentum:** The intuition of momentum is that if successive gradient steps point in different directions, we should cancel off the directions that disagree and if successive gradient steps point in similar directions, we should go faster in that direction. Thus, this method think it is important to consider the previous direction into the gradient and modify the step 3 in Algorithm 9. Suppose $g_k = \nabla \mathcal{L}(\theta^{(k)})$ and g_k is changed as $g_k = \nabla \mathcal{L}(\theta^{(k)}) + \mu * g_{k-1}$, which g_{k-1} represent the previous direction.
- **Stochastic Gradient Descent:** When the data set is large (which is usually the case for deep learning), it is impossible to use all the data to update the gradient each time. Instead, we sample data to update gradient, which we call it as stochastic gradient and the process can be seen as follows:
  - Sample data set $\mathcal{D}_s \subset \mathcal{D}$.
  - Estimate $g_k \Leftarrow -\nabla \frac{1}{|\mathcal{D}_s|} \sum_{i=1}^{|\mathcal{D}_s|} \mathcal{L}(Y_i, f(X_i, \theta^k)) \approx \mathcal{L}(\theta^{(k)})$
  - $\theta^{k+1} \Leftarrow \theta^k - \alpha g_k$

  Each iteration is called a mini-batch. In practice, we shuffle data instead of randomly sampling.
- **Adaptive Gradient:** The adaptive gradient has two different methods. The first method is using momentum. The second method normalised the gradient using the previous gradient in the past periods. The update rule can be seen as follows:
$$\theta^{(k)} = \theta^{(k-1)} - \alpha * \frac{\nabla \mathcal{L}_k(\theta^{(k-1)})}{\sqrt{\sum_{k'=1}^{k} \nabla \mathcal{L}(\theta^{(k'-1)})^2}} \tag{2.2}$$
.
- **Adam:** Adam is the most well-used algorithm to automatically tune momentum and learning rates. It combines AdaGrad and RMSP, and it is the default optimizer tuner in deep learning. The process can be seen as Algorithm 10, where α,β_1 and β_2 denote learning rate, 1st moment decay rate and 2nd moment decay rate, respectively.

Algorithm 10 Adam

```
$M_0 = 0,\ R_0 = 0$
**for** $t = 1, \dots, T$ **do**
    $M_t = \beta_1 M_{t-1} + (1 - \beta_1)\nabla\mathcal{L}(\theta^{(t-1)})$ // 1st moment estimate
    $R_t = \beta_2 R_{t-1} + (1 - \beta_2)\nabla\mathcal{L}(\theta^{(t-1)})^2$ // 2nd moment estimate
    $\hat{M}_t = \frac{M_t}{1-(\beta_1)^t}$ // 1st moment bias correction
    $\hat{R}_t = \frac{R_t}{1-(\beta_2)^t}$ // 2nd moment bias correction
    $\theta^{(t)} = \theta^{(t-1)} - \alpha\frac{\hat{M}_t}{\sqrt{\hat{R}_t + \epsilon}}$ //update
**end for**
**return** $\theta^{(T)}$
```

2.3 Deep Neural Nets

This subsection will introduce the deep neural nets. This idea was generated by 1993. AI has two important streams to tackle it: Symbolic AI and Connectionist AI. In a really long time, Symbolic AI was considered as stat-of-art.After 2000s, especially 2012, we have Connectionist AI, superhuman models.

- **Symbolic AI**: Precept some rules and based on the rules to obtain some optimal decisions. The typical example of it is the expert system
- **Connectionist AI**: We are trying to establish connections just like the connection of nucleus and dendrites. The typical example is neural networks

1945 is an important time. Before 1945, 10^{16} computations in the entire history, but now is already have more than 10^{16} computations per second and increasing rapid especially after we have GPT.

2.3.1 Why Deep neural networks are so successful?

As shown in Figure 2.3.1, the x-axis is the amount of data we have and y-axis is the performance. The upper bound of the transitional machine learning algorithms and the shallow, even the medium neural networks are very low. However, with the recent data and computational power, we even do not know the upper bound of the performance of deep neural networks.

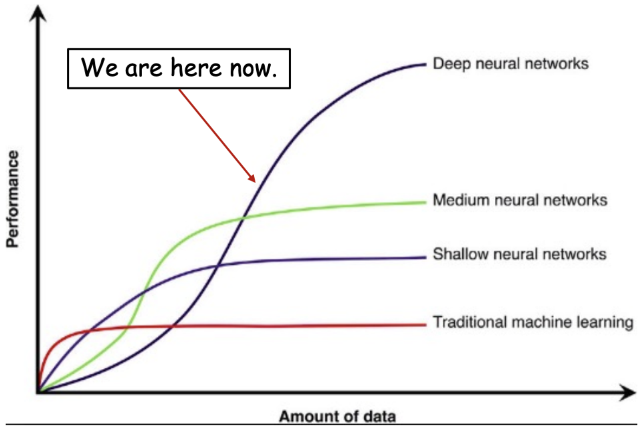


Figure 2.1: Performance of Models

2.3.2 What is the neural network?

We have a simple example of patient's mortality model to illustrate how NN works:

- **Independent Variables**: We have three inputs: Age, Gender, Stage
- **Hidden Layer**: 2 hidden nodes, thus we have 6 parameters linking 3 independent variables to 2 hidden nodes
- **Activation Function**: Activation function will determine whether use the hidden node. previously we use Sigmoid, but now we use ReLU more often
- **Prediction**: After the whole transformation, we have one output, which is the prediction

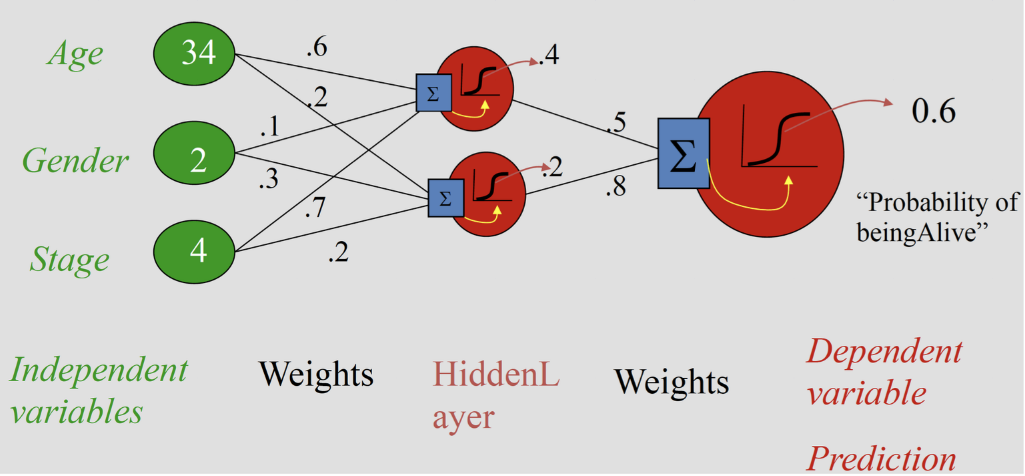


Figure 2.2: Neural Network Model

Universal Approximation Theorem: One hidden layer NN is enough to represent (not learn) an approximation of any function with an arbitrary degree of accuracy.

This theorem indicates that, for a simple NN, it can converge to any function with sufficient data. This is strong, since we only know the existence but do not knwo how to get the parameters.

2.3.3 Are Simple Multilayer Perceptrons (MLP) Outdated?

the key is to identify interesting and impact applications. A group of mathematicians believed there were some hidden structures behind a very important math problem, but they did not know what exactly it was. They have build a simple two hidden layer NN. In this small scale NN, they found some hidden structures and can be proved by the mathematicians. This is quiet interesting, since NN does not solve the problem, but shows that there exists some structures and facilitates academicians to tackle it. Thus although MLP is very outdated, researchers still can have some new applications.

2.3.4 Neural Nets

More layers means more complex functions, more powerful learning and more required data. In the personal experience, 2 hidden layers, which is 3 layers, is the sufficient enough.

- **0 hidden layers**: it is just logistic regression or linear regression, especially for classification
- **1 hidden layers**: essentially to approximate the boundary of convex region
- **2 hidden layers**: combinations of convex regions

2.3.5 Activation function

Activation functions make the whole function nonlinear, which is very important!!! If all activation functions are all linear, they are simple OLS functions. In the following figure, we have a one hidden layer NN, the activation function of the hidden layer and of the output is Sigmoid and Softmax function respectively.

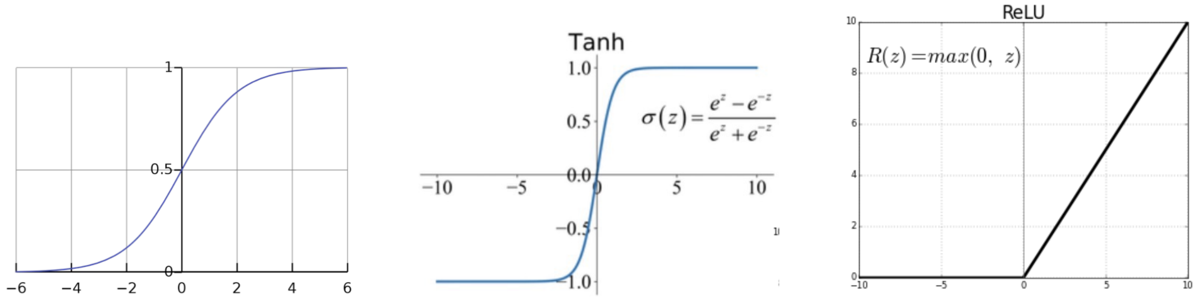


Figure 2.3: Sigmoid, Tanh, ReLu Activation Function

There are different types of activation functions:

- **Sigmoid**: The output value of Sigmoid function is between 0 and 1. If the input value is close to negative infinity, the output value is close to 0; if the input value is close to positive infinity, the output value is close to 1
- **Tanh**: Tanh is like Sigmoid but extreme. It is much more steeper near the 0, which means that near the 0, we will have a linear function but away from 0, we will get 1 or -1
- **ReLU** (mostly used): It is almost linear, beside less than 0. Since it pushes all non-positive value to 0, it might decrease the ability of the model to fit or train from the data properly

2.3.6 Estimation for Neural Nets

Loss function To train a DL model, we need to define a loss function for the model. Then we optimize on the loss function to update the parameters.

- For regression problems, the loss function is typically a mean squared error function

$$L(Y,\hat{Y})=\sum_i (Y_i-\hat{Y}_i)^2$$

- For classification problems, the loss function is typically a cross-entropy function

$$L(Y,\hat{Y})=\sum_i [Y_i\cdot\log\hat{Y}_i+(1-Y_i)\cdot\log(1-\hat{Y}_i)]$$

After we define the loss function for the model, the next step is to train, i.e., minimize the loss.

Train the model The main idea of training neural networks is the chain rule. To find the gradient of loss function, we use backpropagation. For example, the logistic regression, as shown in the following:

a) Forward Propagation Equation:

$$
\begin{aligned}
a &= \sum_{j=0}^{D} \theta_j x_j \\
y &= \frac{1}{1+\exp(-a)} \\
J &= y^* logy + (1-y^*) log(1-y)
\end{aligned}
$$

b) Backpropagation Equation:

- Compute the gradient of the loss (J) with respect to the output (y): $\frac{dJ}{dy}$

- Apply the chain rule to find the gradient of the loss with respect to the input a:

$$
\begin{aligned}
\frac{dJ}{da} &= \frac{dJ}{dy}\frac{dy}{da} \\
\frac{dy}{da} &= \frac{\exp(-a)}{(\exp(-a)+1)^2}
\end{aligned}
$$

- Compute the gradient of the loss with respect to the parameters (θ):

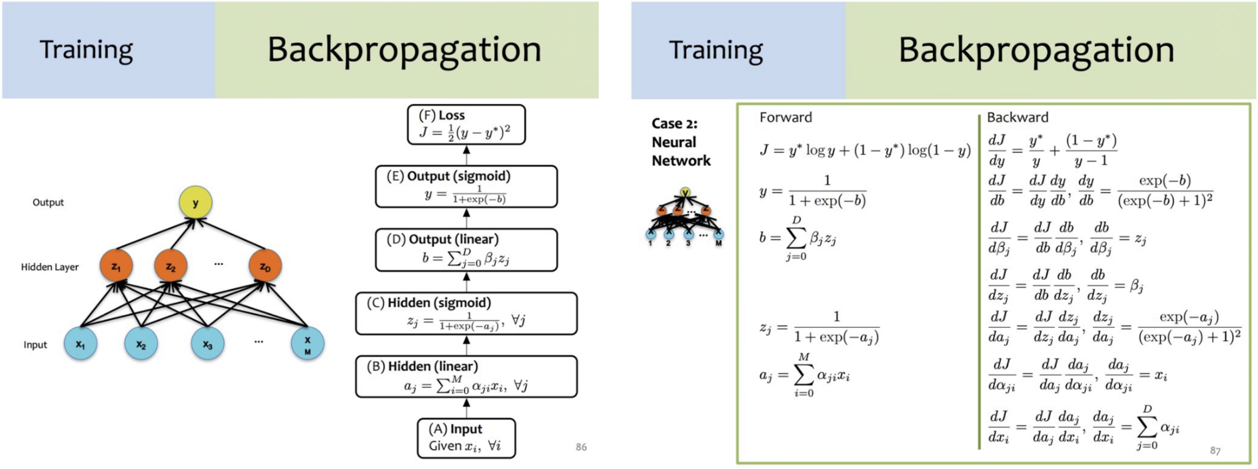
$$
\begin{aligned}
\frac{dJ}{d\theta_j} &= \frac{dJ}{da}\frac{da}{d\theta_j} \\
\frac{da}{d\theta_j} &= x_j
\end{aligned}
$$

For each epoch, the process can be summarised as follows:

- Shuffle data

- For each mini-batch:
  - Use backpropagation to compute the gradient
  - Use Adam to update the parameters

The demo of backpropagation are provided. For more layers, the idea is similar, which is shown in Figure 9.10.



Training Backpropagation

(F) **Loss** $J = \frac{1}{2}(y - y^*)^2$

(E) **Output (sigmoid)** $y = \frac{1}{1+\exp(-b)}$

(D) **Output (linear)** $b = \sum_{j=0}^{D} \beta_j z_j$

(C) **Hidden (sigmoid)** $z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

(B) **Hidden (linear)** $a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

(A) **Input** Given $x_i, \ \forall i$

86

Training Backpropagation

Case 2: Neural Network

Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \ \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \ \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \ \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \ \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \ \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \ \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$$

87

Figure 2.4: Backpropagation

2.3.7 Gradient Descent for DNN

How does the loss function landscape of DNN look like? In general, it's very complex and we don't know. This partially explains why deep learning is computationally hard. The ResNet with skip connections make it smoother.

2.3.8 Make the training process better

- **Initialization** This is about where we should start for our parameters. There are different researchers tried to randomize w, b at the beginning, such that after each backpropagation the distribution will not change much. This is conducted in the beginning of every epoch.

  When the problem is convex (e.g., linear regression, logit regression), we do not need to do it.

- **Normalization/ standardization** We conduct normalization to our training data, in order to make the optimization and the landscape stable and robust. This is conducted during the training process.

  The idea is, we are trying to make the distribution of the features the same when we are go through the entire neural network

- **Skip Connection** This is to address the gradient vanishing phenomenon.

A demo can be found here.

2.3.9 Overfitting and Regularization

DL model are tend to overfitting. Some regularization methods can be implied to address this issue

- Forward / Backward methods
- Shrinkage methods: e.d. LASSO
- Dimension reduction: Project to a low dimension space. e.g. PCA

- **Dropout** methods: adds a drop out layer with probability p, in order to add some noise into the training of the neural net

2.4 Computations in Deep Learning

GPU is very key tool to train the deep learning, and more GPUs means more high speed and high ability to handle more complex deep learning, which is shown in Figure 2.5.

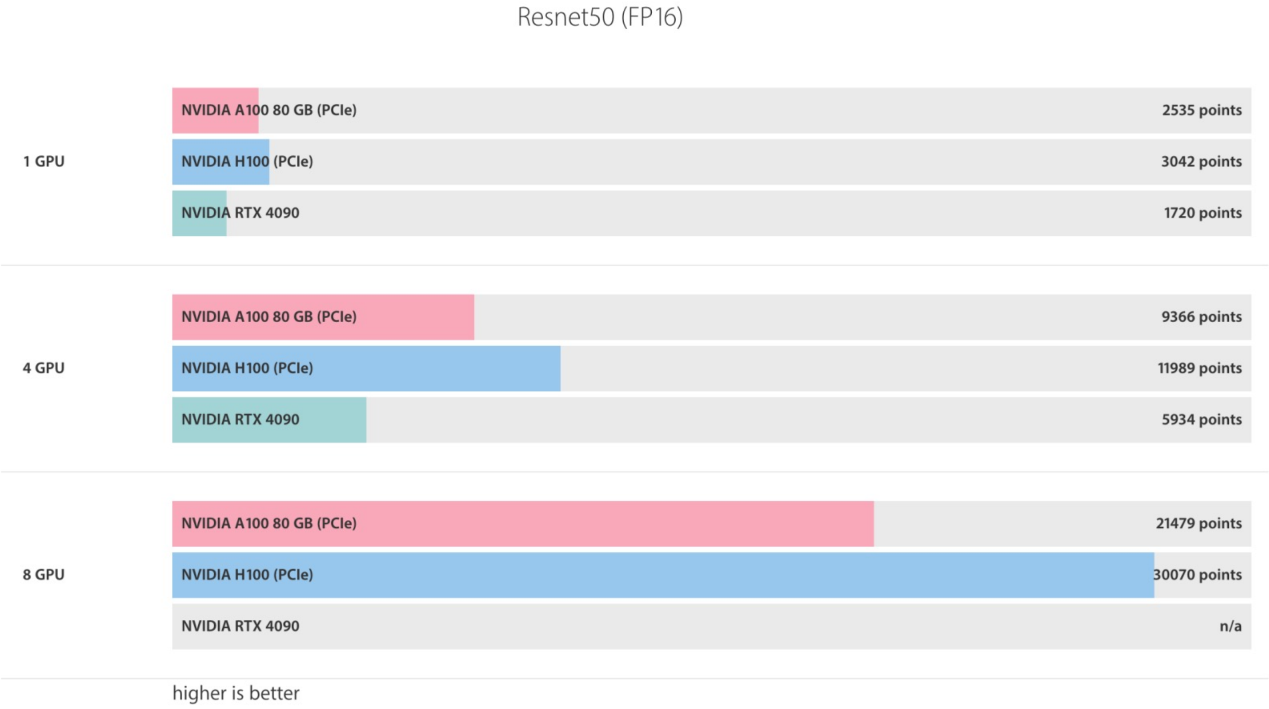


Figure 2.5: Backpropagation

2.4.1 GPU Equipment

- Self-made workstations/servers:
  - CPU + Nvidia 4090 (US$1,500, but very hard to get)
  - CUHK Business School DOT will deploy a server with 8 4090s soon
- **Cloud**
  - GCP https: US$ 4.05 per 1 A100 per hour
  - Amazon: US$32.77 per 8 A100 per hour
- **Computing costs**: Proportional to the size of the network (the number of parameters) and the size of the training data

2.4.2 Model Size and Training Time

- Resnet-50, 12 million parameters, ImageNet Data, TF32, 30mins on an 8 A100s server (DGX)
  - $0.5 * 8 * 1.5 = 6$ 10 hrs on 4090

- BERT, 110 million parameters, 170GB BooksCorpus and Wikipedia Data, TF32, 5 hrs on DGX
- BERT fine-tuning, Stanford Question Answering Data, 3 5 mins on DGX
- GPT-3, 175B parameters (3.64e23 FLOPs, 300B training tokens), TF32, 128 DGX servers (A100: 80 TFLOP/s). How long it takes to pre-train GPT-3?
  - $3.64 * e23/(80 * e12)/128/8/24/3600$ = 51 Days – 100 Days, which means 3 months

3

Natural Language Processing (1)

——**Scribed by Yiyi Zhao and Zhankun Chen**

3.1 Outline

This note will contain following sections:

- In Section 3.2, we introduce the application of deep learning in business research, highlighting its applications and challenges.

- In Section 3.3, we introduce the natural language processing, the development path, the applications in business research, the steps of text processing, as well as the corresponding technologies.

3.2 Prediction Problems in Business Research

3.2.1 Why Do We Care About Predictions?

- Relevance to Global and Political Metrics: Prediction is crucial for understanding and anticipating outcomes in diverse areas such as population growth, election results, GDP changes, tax policies, and environmental impacts. Accurate predictions influence decision-making at all levels, affecting individuals and societies globally.

- Decision-Making and Policy Formulation: Predictions enable better decisions by illustrating potential outcomes of different actions or policies. For example, Kleinberg et al. (2015) introduce a formula that decomposes the benefit of a decision into parts related to the accuracy of predictions and their direct impact on outcomes.

$$\frac{d\pi(X_0, Y)}{dX_0} = \frac{\partial \pi}{\partial X_0} \underbrace{(Y)}_{\text{prediction}} + \frac{\partial \pi}{\partial Y} \cdot \frac{\partial Y}{\underbrace{\partial X_0}_{\text{causation}}} \tag{3.1}$$

  where $\pi(X_0, Y)$ is the cost associated with decision X_0 and the fact Y.

- Direct Causal Inference: causal inference as a vital area where prediction plays a critical role, especially in determining the effects of interventions or policies.

3.2.2 Applications

- Macro Forecasting: Jean et al. (2016) discuss the use of high-resolution satellite imagery combined with machine learning to estimate consumption and wealth in hard-to-measure areas. The implication is that this method provides a new avenue for assessing poverty, which can have significant policy implications. Based on machine learning method, Van Binsbergen et al. (2024) use 200 million pages of text from 13,000 US local newspapers to construct a 170-year-long time series measure of economic sentiment, which shows predict power for economic fundamentals like GDP and consumption growth, even considering other common predictors.

- Demand Forecasting: Some Papers use ML to predict the demand (Bajari et al. (2015); Cohen et al. (2022); Cui et al. (2018)).

- Business: Farias and Li (2019) develop a method that is simpler than existing matrix completion approaches and provides significant performance improvements over state-of-the-art methods for both in-sample accuracy and out-of-sample forecast accuracy. Peng and Liang (2023) utilize a dataset from an e-commerce platform and a credit card transaction dataset to compare algorithms, and finds that view-based algorithms (VB-CF) are more effective in certain contexts, such as increasing the sale of niche products, while purchase-based algorithms (PB-CF) are more effective for popular products.

- CS: Zhan et al. (2022) propose a confounded watch-time prediction that uses a causal graph to study duration bias and a counterfactual video recommendation system to eliminate such bias, improving the accuracy of video recommendations. Likewise, Covington et al. (2016) describe the system's architecture and how deep neural networks are used for video recommendation, improving the system's accuracy and freshness.

- Other Predictions: See Gu et al. (2020) and Cao et al. (2023).

3.2.3 Comments on Applications of Deep Learning in Social Science Research

- Accuracy and Reliability: we need for accurate predictions and the limitations of certain models, especially in complex social science contexts where data can be biased or incomplete. Therefore, the empirical validation and the use of comprehensive data are very crucial to establish a reliable predictive model.

- Ethical and Practical Implications: we need concern about some ethical considerations in using predictive models, especially in sensitive areas like healthcare and criminal justice, where the stakes of predictions can be very high.

3.2.4 When do Predictions Make No sense?

- we need predict important outcome that people really care about, otherwise, the prediction process will become meaningless

- the prediction of y should be accurate enough, and we need report the accuracy in paper. In addition, we also need consider the causal identification in this process.

3.3 Natural Language Processing

3.3.1 Introduction to Natural Language Processing (NLP)

Natural Language Processing (NLP) stands at the intersection of linguistics, computer science, and artificial intelligence. As an interdisciplinary field, NLP focuses on the study and development of computational systems that can understand, interpret, and generate human language. This involves exploring the ways in which computers can process and analyze natural language data to perform tasks such as translation, sentiment analysis, and information extraction. Typical applications for NLP range from a wide range of applications, including but not limited to:

- Sentiment Classification: Assigning sentiment scores to texts to gauge opinions.
- Machine Translation: Converting text from one language to another.
- Document Similarity: Assessing and comparing documents for similarity.
- Topic Modelling: Discovering the abstract 'topics' that occur in a collection of documents.
- Additional applications cover various other aspects of language understanding and generation.

In Natural Language Processing, a classic methodology involves the use of a supervised learning framework, wherein the inputs to the system are various forms of text, and the desired output is specific characteristics or classifications of these texts.

- Sentiment Classifier: Text to Sentiment Score
- Review Classification: Text to Review Problem
- Machine Translation: Text to Other language

A classic framework in Natural Language Processing typically comprises two main components, reflecting the foundational structure upon which various NLP tasks are built and executed (Jurafsky and Martin (2024); Coursera (2021)).

STEP1: Preprocessing

- The first step in any NLP task, where raw text is transformed into a more manageable form.
- Involves cleaning the text and breaking it down into manageable pieces or tokens.
- Methods include tokenization, stemming, lemmatization, and part-of-speech tagging.
- Goal is to convert text into a numerical format that machine learning algorithms can understand.

STEP2: Classification

- Following pre-processing, the numerical data is used to train predictive models.
- Common algorithms include Naive Bayes, Support Vector Machines, and Neural Networks.
- The outcome is a probabilistic or categorical label, like sentiment scores.

Example Workflow

- A sentence (“I am happy because I am learning NLP") goes through pre-processing to extract features.
- These features are fed into a machine learning model, such as Logistic Regression (LR).
- The model is trained to classify the sentiment, outputting a label (e.g., “Positive").

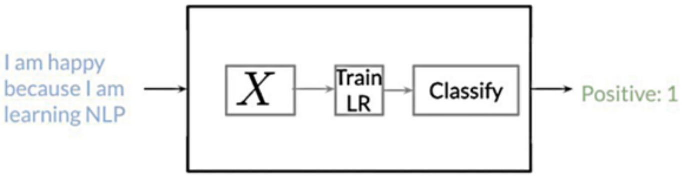


Figure 3.1: NLP workflow example

3.3.2 Historical Evolution of NLP

The timeline of NLP has evolved from early rule-based systems to advanced AI-driven methods. Key advancements include the introduction of attention mechanisms in models like BERT, enhancing text context understanding, and the refinement of models like GPT through human feedback, significantly improving nuanced language generation.

- Before 2010: Focus on probability and vector space models, with foundational methods such as TF-IDF and latent semantic analysis.
- 2013-2017: Advancements in word embeddings with Word2Vec, and sequence-to-sequence models enhancing machine translation and text understanding.
- 2018-2022: The rise of transformer-based models, such as BERT (Bidirectional Encoder Representations from Transformers), particularly the breakthrough with attention mechanisms, leading to significant improvements in numerous NLP tasks.
- 2023-Now: The development and refinement of Large Language Models (LLMs) have leveraged vast datasets to enhance their ability to predict and generate text. These models produce language outputs that are not only increasingly coherent but also contextually relevant, reflecting a significant advancement in the field of natural language processing.

NLP strategies have evolved over time due to methodological shifts, driven by advancements in technology and a deeper understanding of language complexities, paving the way for more sophisticated approaches to processing and interpreting human language.

- New DL Architecture: Discusses the advancement in deep learning architectures which enhance traditional NLP tasks and enable new applications such as machine language generation and chatbots.
- Consistency in Pre-processing: Despite advancements, the foundational approach to words and pre-processing remains constant, ensuring continuity in NLP methods across different eras.
- Advantage of Old Architectures: Highlights the benefits of older NLP models, particularly their explicit probabilistic linguistic models which are often more transparent and may be advantageous in economic-related ML research.

3.3.3 NLP Task Workflow and Traditional NLP in Econ

- Input: Starting with natural language data.
- Pre-processing: Transforming raw text into a format amenable to ML models.
- Representation Learning: Learning meaningful numerical representations of text.
- Downstream Task: Applying learned representations to a specific NLP task, such as sentiment analysis, translation, etc.
- Outcome: The result or topic of interest, exemplified as NLP in this context.

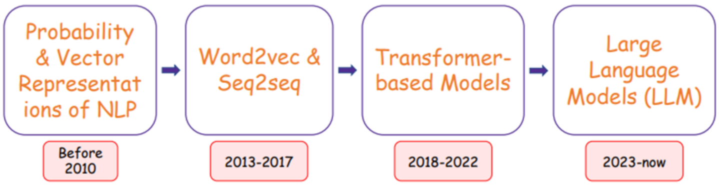


Figure 3.2: The historical development of NLP

In the field of economics, NLP techniques have been instrumental in analyzing vast amounts of textual data, such as policy documents, speeches, and news articles, to uncover valuable insights. By applying NLP, economists can explore the nuances of language in central bank communications, detect biases in congressional speeches, and evaluate media slant in various news outlets. These applications are evidenced by studies published in prestigious journals like The Quarterly Journal of Economics and Econometrica, highlighting NLP's critical contribution to advancing economic research and analysis.

- Transparency and Deliberation Within the FOMC: Utilizes computational linguistics to investigate the effects of transparency on the communication patterns within the Federal Open Market Committee (Hansen et al. (2018)).
- Measuring Group Differences in High-Dimensional Choices: Applies NLP to measure trends in partisanship in congressional speech (Gentzkow et al. (2019b)).
- What Drives Media Slant?: Constructs a new index of media slant based on the similarity of a news outlet's language to that of political parties (Gentzkow and Shapiro (2010a)).

3.3.4 Text Pre-processing Techniques

Gentzkow et al. (2019a) provid a foundational introduction to leveraging text as a valuable data source in economic research, underscoring the potential of textual analysis to capture the intricacies of human interaction, communication, and cultural expression within the digital realm. Below are the processing steps involved in analyzing text data.

- **Pre-processing:** The initial step involves preparing raw text data for analysis, which is crucial for obtaining reliable results.
- **Numerical Representation:** The raw text D is then transformed into a numerical array C, facilitating computational handling.

- **Mapping to Outcomes:** This numerical array C is used to predict values $\hat{V}$ of unknown outcomes V.

- **Application in Analysis:** The predicted values $\hat{V}$ are subsequently utilized in descriptive or causal analyses to derive insights.

The core steps in text data preprocessing include a detailed exploration of tokenization, which involves splitting text into individual words or phrases; normalization, aimed at converting text to a uniform case and removing punctuation; stemming, which reduces words to their root forms; and lemmatization, a process that converts words to their base or dictionary form, ensuring that the underlying meaning is preserved (Schütze et al., 2008; Stanford University, 2021). The key steps in text normalization involve several processes to standardize textual data, making it more amenable to analysis. These steps include:

- **Elimination of Non-words:** This includes removing elements like URLs, HTML tags, handles, and punctuation that do not contribute to text meaning.

- **Tokenization:** It involves parsing strings of text into individual words or tokens.

- **Stop-word Removal:** This step involves discarding extremely common words like "a", "an", "is", "the", and "of", which are typically irrelevant to the analysis.

- **Stemming:** Reducing words to their stem or root form, simplifying the variety of word forms to a base form. For instance, the word "tuning" from the phrase "tuning GREAT AI model" is stemmed to "tun", which is the root that connects related words such as "tune", "tuned", and "tuning". This process simplifies the dataset by collapsing these different forms into a single token, facilitating the analysis by reducing the complexity of the language data.

- **Normalization:** This includes converting all letters to lower-case and normalizing accents and diacritics to reduce variability in text data.

3.3.5 Word Representation Models

The vast vocabulary of words and their frequencies can be represented in various ways to encapsulate the meaning of sentences or documents.

- **Frequentist view:** This view suggests representing words as vectors. Such vectors are essentially a low-dimensional form of one-hot encoding, where the significance of a word is determined by its context within neighboring words.

- **Example:** In the frequentist view of word representation, words are treated as vectors. An example of this approach could be the bag-of-words model, where each word in a document is represented by its frequency. Imagine a simple sentence: "The cat sat on the mat." In a bag-of-words model, we disregard grammar and word order, and simply count the frequency of each word: "the" appears twice, while "cat," "sat," "on," and "mat" appear once. The sentence could be represented as a vector showing these word frequencies.
  For more complex models like Word2Vec, the representation might involve dense vectors that place similar words close together in the vector space based on their context within large text corpora. This means words that appear in similar contexts, like "king" and "queen," would have their vector representations closer together.

- **Bayesian Perspective:** In this approach, words are depicted as probabilities, each with a prior probability, influencing the likelihood of word occurrence. Sentences are then understood as a sequence of words with conditional probabilities.

- **Example:** Let's consider we're analyzing text related to weather forecasts, and we're particularly interested in how often the word "rain" appears after the phrase "The weather forecast predicts." Initially, we determine the overall frequency of "rain" in our dataset—say it appears in 30% of all forecasts, establishing our prior probability.

  We then refine our analysis by examining the specific context: the frequency of "rain" following the phrase "The weather forecast predicts." If, in our dataset, "rain" follows this phrase 50% of the time, this figure is the conditional probability, which adjusts our expectation based on the specific preceding context.

  This Bayesian approach allows for a dynamic understanding of language, where the likelihood of word occurrences is not static but adjusts as more context becomes available. The sequence of words in sentences is thus seen not merely as a string of independent elements but as a flow of probabilistic events, each influenced by what has come before. This method offers insights into language patterns and word usage, enhancing predictive models and interpretative analyses in NLP by incorporating both prior occurrences and specific contexts.

3.3.6 Term Frequency-Inverse Document Frequency (TF-IDF)

The concept of Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. The formula to calculate TF-IDF for a word in a document is a multiplication of two factors:

$$w_{i,j} = \mathrm{tf}_{i,j} \times \log\left(\frac{N}{\mathrm{df}_i}\right) \tag{3.2}$$

where:

- $w_{i,j}$: The weight of term i in document j.

- $\mathrm{tf}_{i,j}$: The term frequency of term i in document j.

- N: The total number of documents in the corpus.

- df_i: The document frequency of term i in the corpus (the number of documents containing term i).

The Figure 3.3 illustrates the relationship between the frequency of word occurrence in a corpus and the TF-IDF (Term Frequency-Inverse Document Frequency) value. The Y-axis represents the occurrence frequency of words in the documents, and the X-axis represents the TF-IDF value. The TF-IDF value increases when a word is rare across the corpus but frequent within specific documents, signifying the word's importance in those documents.

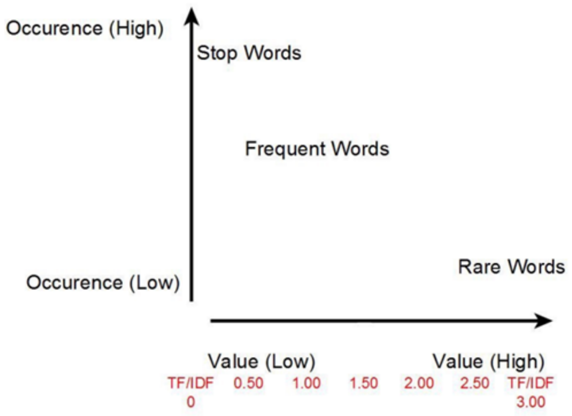


Figure 3.3: The relationship between the frequency of word occurrence and the TF-IDF value

3.3.7 One-hot Encoding

One-hot encoding is a method of representing words as binary vectors in Natural Language Processing (NLP), where each word in a vocabulary is represented by a vector with a single '1' and the rest '0's. The position of the '1' corresponds to the index of the word in the vocabulary.

This representation is sparse, meaning that most of the elements in the vector are zeros, which corresponds to the words that are not present in the document. In econometrics, this is similar to the concept of dummy variables, which are used to indicate the presence or absence of some categorical effect that may be expected to shift the outcome.

In a one-hot encoded vector, the length of the vector k is equal to the size of the vocabulary, and each position in the vector corresponds to a unique word in the vocabulary. If a word occurs in the document, its corresponding position in the vector is marked with a '1', and all other positions are set to '0'.

For example, if the vocabulary is ["I", "am", "happy", "because", "learning", "NLP"], and the document is "I am learning NLP", the one-hot encoded vector would be [1, 1, 0, 0, 1, 1], assuming the order of the words in the vocabulary corresponds to the order in the vector.

3.3.8 Low-Dimensional Dictionary

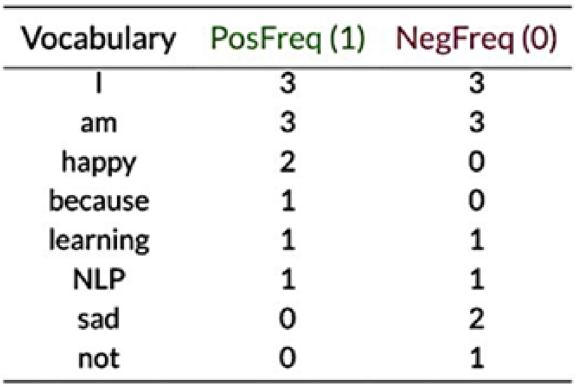
A low-dimensional dictionary is a type of vector representation for text that is more efficient than one-hot encoding. It involves mapping words to their frequency within classes of interest. In text classification, each word is associated with a frequency vector. The length of this vector corresponds to the number of classes we're classifying the text into, plus one. This extra dimension allows us to capture more information about the word's usage, which can improve the performance of classification algorithms.

- **Example:** Considering a sentiment analysis task with two classes (positive and negative), for the word "happy":
  1. PosFreq (1) would be the count of "happy" in positive contexts.
  2. NegFreq (0) would be the count in negative contexts.
  3. If the total number of classes is two, the vector for "happy" in our two-class system (positive and negative) would have 3 elements (2 classes + 1 for the total or another measure), assuming the plus one is for total frequency across all classes.

In the context of sentence representation via a low-dimensional dictionary, sentences are characterized by aggregating the vectors of individual words. This process entails summing up the frequencies at which each word occurs within both positive and negative contexts, thereby creating a composite vector that encapsulates the overall sentiment or thematic essence of the sentence.

- **Example:** Considering the sentence "I am sad, because I am not learning NLP", we aim to represent it as a vector $[x1, x2]$, where $x1$ and $x2$ correspond to the aggregated positive and negative word frequencies, respectively. Given the counts:
  - $x1$ (positive frequencies) = 3 (I) + 3 (am) + 0 (sad) + 1 (because) + 3 (I) + 3 (am) + 0 (not) + 1(learning) + 1 (NLP) = 15
  - $x2$ (negative frequencies) = 3 (I) + 3(am) + 2 (sad) + 0 (because) + 3(I) + 3 (am) + 1 (not) + 1 (learning) + 1 (NLP) = 17

  Therefore, the vector representing the sentence "I am sad, because I am not learning NLP" is $[x1, x2] = [15, 17]$.



Vocabulary	PosFreq (1)	NegFreq (0)
I	3	3
am	3	3
happy	2	0
because	1	0
learning	1	1
NLP	1	1
sad	0	2
not	0	1

Figure 3.4: Example

3.3.9 Low-Dimensional Neighbour Representation of Words

Words can be represented by considering the context in which they appear, specifically the neighbouring words. This is known as a low-dimensional neighbour representation. The representation is constructed by counting how often other words occur in the vicinity of the target word.

- For instance, with $k = 2$, we consider two words before and after the target word. For the word "data" in the sentences "I like simple data" and "I prefer simple raw data", the count of neighbouring words within the range of k is recorded.

If you have multiple sets of documents and each one is different from others, you can use a word's occurrence in these documents to represent a word's meaning. This technique involves representing words by their distribution across a collection of documents, underpinning that words with similar meanings will have similar distribution patterns.

The term-document matrix is a common way to represent words in the context of the documents in which they appear. Each row corresponds to a word, and each column corresponds to a document. The matrix entries represent the frequency of the word in each document.

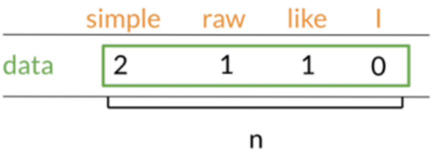
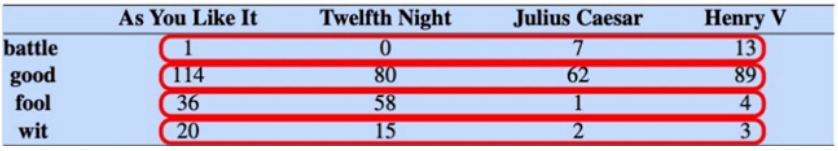


Figure 3.5: Example of Low-Dimensional Neighbour Representation of Words

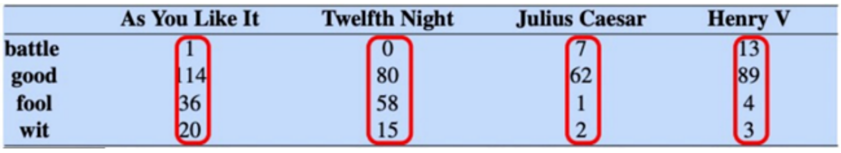
- Example: The matrix excerpt provided shows the frequency of four words across four Shakespeare plays. For instance, the word "good" appears 114 times in "As You Like It" and 89 times in "Henry V," suggesting its prominent use in these texts. (Jurafsky and Martin, 2024)
- In Figure 3.6, the representation of words across various documents is illustrated through the frequency of their occurrences, employing a numerical approach. For instance, "battle" manifests differently across four Shakespeare plays, appearing once in "As You Like It," not at all in "Twelfth Night," seven times in "Julius Caesar," and thirteen times in "Henry V." This variance is encapsulated in a row vector, which quantitatively portrays "battle" based on its presence within these texts.



	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 3.6: The term-document matrix for four words in four Shakespeare plays. The red boxes show that each word is represented as a row vector of length four.

- In Figure 3.7. similarly, when aiming to represent an entire document such as "As You Like It," a distinct column vector is utilized, capturing the frequencies of selected terms—"battle," "good," "fool," and "wit"—as 1, 114, 36, and 20, respectively. This method, known as the term-document matrix, is instrumental in textual analysis. It facilitates the comparison and identification of document similarities by providing a systematic and structured approach to examining textual data, thereby offering insights into thematic consistencies and divergences across documents.



	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 3.7: The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

3.3.10 Two paths for AI development

The development of NLP is a reflection of the broader evolution of AI, which has historically branched into two main streams over the past 50 to 70 years: the symbolic and the connectionist approaches.

1. Symbolic AI:

- In its early phase, NLP was heavily influenced by the symbolic approach, characterized by rule-based systems and explicit programming of language rules.
- Techniques such as dictionary methods or probabilistic models, based on Bayes' theorem, are examples of symbolic AI, which prioritize human interpretability and understanding of language.

2. Connectionist AI:

    - The transition to connectionist AI represents a shift towards models that are less interpretable but often perform better.
    - Inspired by neural networks and the idea of mimicking the brain's connections, it has led to the development of models like Transformers. However, we notes the analogy to human brain structure is not substantiated by evidence. This approach focuses on the strength and structure of connections within the model, which may differ significantly from the neural connections in the human brain.
    - Recently, there has been a surge in connectionist-type NLP applications in business and economics research. We suggests a critical approach to the literature, emphasizing the importance of discerning valuable research in the vast array of publications.

4

Natural Language Processing (2)

——**Scribed by Zeshen YE and Yingxin LIN**

4.1 Outline

The outline of this notes is as follows:

- In section 4.2, we will introduce downstream tasks in Natural Language Processing (NLP);
- In section 4.3, we introduce the simplest kind of language model: the n-gram language model;
- In section 4.4, we introduce the naive Bayes algorithm and its application on text categorization, the task of assigning a label or category to an entire text or document;
- In section 4.5, we will give some traditional NLP applications in business research.

4.2 NLP Downstream Task

In the context of machine learning and natural language processing , a downstream task refers to specific applications or tasks that utilize the knowledge acquired from pre-trained models for further processing or analysis. The idea is that the pre-trained model captures general features of language, and then these features are fine-tuned or adapted for specific tasks, often requiring less data and computational resources compared to training a model from scratch.

4.2.1 Sentiment Classification

Sentiment Classification involves determining the sentiment expressed in a piece of text, such as positive, negative, or neutral.

Example:

- We pre-precess our data which includes $Sentence_i$ and its $Label_i$ = positive or negative.
- After pre-processing, we get $X = (X1_i, X2_i, X3_i)$ and its y_i, where X is the word/sentence/document-representation vector and y represents the label equal to 1 (positive) or 0 (negative).
- Then, we can use Machine Learning methods to find the function between X and y. For example:

- Linear probability model;
- Logistic regression or other generalized linear models;
- Deep learning.

4.2.2 Semantic Similarities

Semantic similarity refers to the degree of likeness or resemblance in meaning between two linguistic expressions, such as words, phrases, or sentences.
How to measure the similarities: Distance or Angle

1) Euclidean Distance: it measures the straight-line distance between two points, providing a quantitative measure of their dissimilarity or similarity. The smaller Euclidean distance means higher similarity.
Example:
Corpus A: (A_1, A_2) and Corpus B: (B_1, B_2).
The Euclidean distance:

$$\text{d(B,A)}=\sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2}$$

2) Cosine Similarity: a measure used to determine how similar two vectors are in a multi-dimensional space. It calculates the cosine of the angle between two vectors. 1 indicates perfect similarity, 0 indicates no similarity, and -1 indicates perfect dissimilarity.
Example:

$$\text{cosine}(\mathbf{v},\mathbf{w})=\frac{\mathbf{v}\bullet\mathbf{w}}{|\mathbf{v}|\bullet|\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} {v_i}^2}\sqrt{\sum_{i=1}^{N} {w_i}^2}}$$

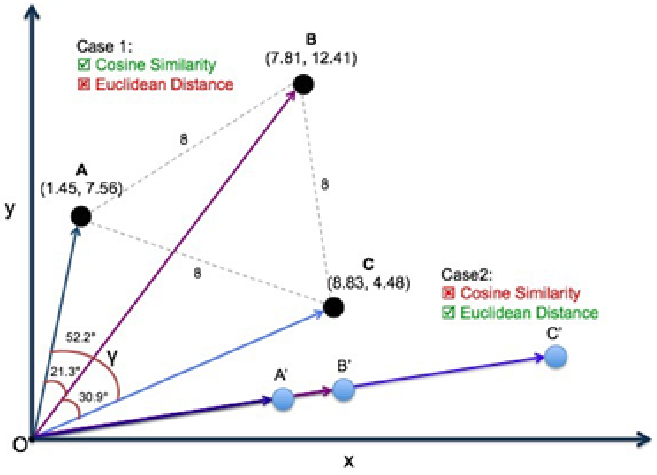


Figure 4.1: Cosine Similarity & Euclidean Similarity

4.2.3 Machine Translation

Machine translation refers to the automated process of translating text or speech from one language to another using computational algorithms and techniques.

4.2.4 Summary of NLP

1. **Pre-processing:** Take a dataset and do text normalization.
2. **Word-representation:** Use your favorite way to do word representation
3. **Sentence/Document-representation:** Use your favorite way to do sentence/document representation
4. **Downstream-task:** Use your favorite model to do downstream tasks.

4.3 N-Gram Language Model

Language models (LMs): Models that assign probabilities to upcoming words, or sequences of words in general. They can find the probability of a word given the entire history of the sentence so far.

Bayesian Perspective: N-gram: An n-gram is a sequence of n words: a 2-gram is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram is a three-word sequence of words like "please turn your", or "turn your homework". But we also (in a bit of terminological ambiguity) use the word "n-gram" to mean a probabilistic model that can estimate the probability of a word given the n-1 previous words, and thereby also to assign probabilities to entire sequences.

4.3.1 N-gram Model Example

For example, we have sentence "its water is so transparent that the...". Suppose the history is "its water is so transparent that" and we want to know the probability that the next word is "the":

$$P(the|its\ water\ is\ so\ transparent\ that)$$

One way to estimate this probability is count the numbers of "its water is so transparent that" and "its water is so transparent that the" from a very large corpus.

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{Count(its\ water\ is\ so\ transparent\ that\ the)}{Count(its\ water\ is\ so\ transparent\ that)}$$

This model is too complex and suffers from the curse of conditionality. The number of combinations of word history grow exponentially with text length, and it requires prohibitively large datasets to computer.

N-gram model refines the idea by limiting the dependencies on history. We can decompose this probability using the chain rule of probability:

$$P(X_1...X_n) \ = \ P(X_1)P(X_2|X_1)P(X_3|X_{1:2})...P(X_n|X_{1:n-1}) \ = \prod_{k=1}^{n} P(X_k\ |X_{1:k-1})$$

$$\begin{aligned} P(its\ water\ is\ so\ transparent) \ &= \ P(its)\times P(water|its)\times P(is|its\ water)\times P(so|its\ water\ is\ ) \\ &\quad \times P(transparent|its\ water\ is\ so\ ) \end{aligned}$$

We can approximate the history by just the last few words.

- Unigram: $P(w_1...w_n) \approx \prod_{i=1}^{n} P(X_i)$
- Bigram: $P(w_i|w_1...w_{i-1}) \approx P(w_i|w_{i-1})$

The assumption that the probability of a word depends only on the previous word is called a Markov assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past.

There is a trade-off of N:

- A larger N implies a longer-distance dependency.
- A larger N also implies much more data to estimate the conditional probabilities.

4.3.2 Estimating Bigram Probabilities

We use maximum likelihood estimation(MLE) to estimate probabilities.

Given a previous word w_{i-1} to compute a bigram probability of w_i, we need to count of the bigram C($w_{i-1}w_i$) and normalize by the sum of all the bigrams that have the same first word w_{i-1}:

$$P(w_i|w_{i-1}) \ = \frac{C(w_{i-1}w_i)}{C(w_{i-1})}$$

Example:

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

Bigram probabilities:

P(I|<s>) = 2/3 = .67 P(Sam|<s>) = 1/3 = .33 P(am|I) = 2/3 = .67
P(</s>|Sam) = 1/2 = 0.5 P(Sam|am) = 1/2 = .5 P(do|I) = 1/3 = .33

For general case of MLE N-gram parameter estimation:

$$P(w_i|w_{n-N+1:n-1}) \ = \frac{C(w_{n-N+1:n-1}w_i)}{C(w_{n-N+1:n-1})}$$

4.3.3 Smothing

Smoothing is to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen to keep a language model from assigning zero probability to these unseen events.

- Laplace/Add-one Smoothing: The simplest way to do smoothing is to add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1.

$$P_{Add-1}(w_i|w_{i-1}) \ = \frac{C(w_{i-1}w_i)+1}{C(w_{i-1})+V}$$

V means we see each of $(w_{i-1}|w_i)$ one more time for all w_k in the vocabulary.

4.3.4 Toolkits for N-gram models:

- SRILM: http://www.speech.sri.com/projects/srilm/
- KenLM: https://kheafield.com/code/kenlm/

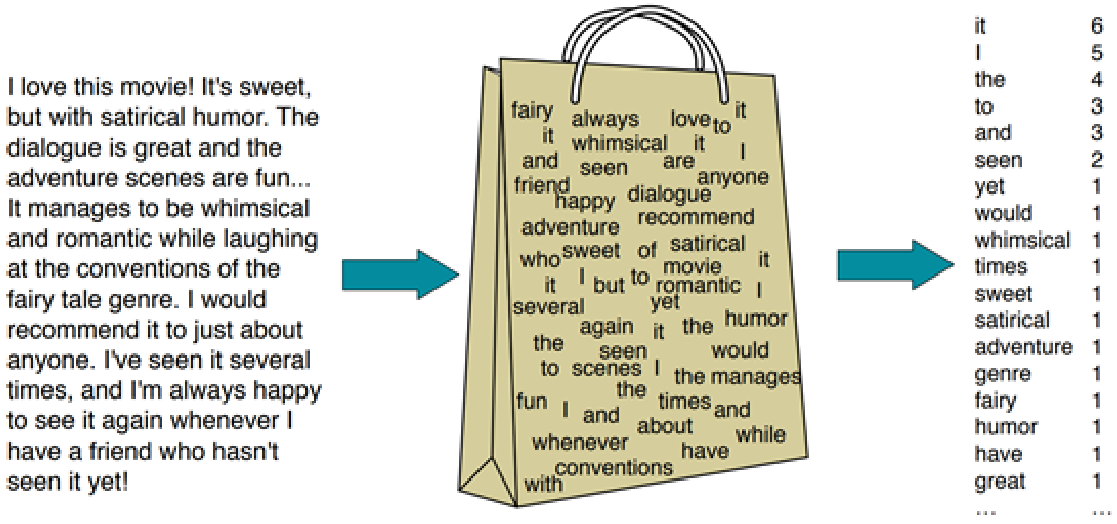
4.4 Naïve Bayes

Naïve Baye is a probabilistic classifier used for assigning a class to a text document. It's a "naïve"/simplifying classification method based on Bayes rule, relying on very simple representation of document – *bag of words*.

4.4.1 Naïve Bayes: Key Assumptions

Naïve Bayes relies on two assumptions:

- **Bag of words assumption**: Position of words in a text document does not matter; all words are of similar importance (see, Figure 4.2).
- **Conditional independence**: The feature probabilities are independent conditioned on class c.


I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!

it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
...	...

Figure 4.2: Bag of Words Assumption: Intuitions

4.4.2 How It Works?

a) Most Likely Class

To know the most likely class (c_{MAP}) of a given document d, we have:

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} \mathrm{P}\left(c|d\right)$$

According to Bayes' theorem, we can convert above equation into:

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} \frac{P\left(d|c\right) * P(c)}{P(d)}$$

As document d is given, we can further eliminate the deflector $P(d)$ and then get:

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} \;\; P\left(d|c\right) * P(c)$$

Therefore, to obtain d's probability of being each class c, we need to know $P\left(d|c\right)$ and $P(c)$ under each class c. And next we compare product of $P\left(d|c\right)$ and $P(c)$ across classes, then finding the maximum one and picking up the corresponding class as the most likely class.

b) Estimate $P\left(d|c\right)$:

Applying *Conditional independence*, we can write $P\left(d|c\right)$ as:

$$P\left(d|c\right) = P\left(w_1|c\right) * P\left(w_2|c\right) * P\left(w_1|c\right) * P\left(w_2|c\right) * \ldots * P(w_n|c)$$

Where, w_i denotes features that can represent document d: i.e., words in the text. For a specific c_j and a specific w_i, $P(w_i|c_j)$ equals to the fraction of times word w_i appears among all words in documents of class c_j:

$$P\left(w_i|c_j\right) = \frac{count\left(w_i, c_j\right)}{\sum_{w \in V} count\left(w_i, c\right)}$$

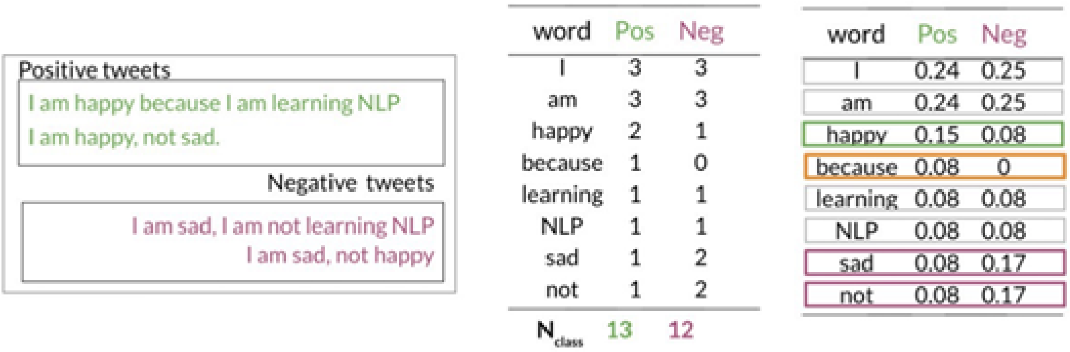
c) Estimate $P(c)$:

For a specific c_j, $P(c_j)$ equals to the fraction of documents in class c_j:

$$P\left(c_j\right) = \frac{N_{c_j}}{N_{total}}$$

4.4.3 Naïve Bayes Example

Input (See, Figure 4.3)**:**



Positive tweets

I am happy because I am learning NLP

I am happy, not sad.

Negative tweets

I am sad, I am not learning NLP

I am sad, not happy

word	Pos	Neg
I	3	3
am	3	3
happy	2	1
because	1	0
learning	1	1
NLP	1	1
sad	1	2
not	1	2
N_{class}	13	12

word	Pos	Neg
I	0.24	0.25
am	0.24	0.25
happy	0.15	0.08
because	0.08	0
learning	0.08	0.08
NLP	0.08	0.08
sad	0.08	0.17
not	0.08	0.17

Figure 4.3: Input (Right) & Word Freq Among Each Class (Left)

Task: Classify the following tweets as positive or negative:

1) $d_1 =$ "I am not sad."

$$LR_1 = \frac{P\left(pos|d_1\right)}{P\left(neg|d_1\right)} = \frac{P\left(d_1|pos\right) * P(pos)}{P\left(d_1|neg\right) * P(neg)} = \frac{P(pos) * \prod_{w_i \in d_1} P\left(w_i|pos\right)}{P(neg) * \prod_{w_i \in d_1} P\left(w_i|neg\right)} = \frac{P(pos)}{P(neg)} * \frac{\prod_{w_i \in d_1} P\left(w_i|pos\right)}{\prod_{w_i \in d_1} P\left(w_i|neg\right)}$$

Here,

$$\frac{P(pos)}{P(neg)} = \frac{\frac{13}{13+12}}{\frac{12}{13+12}} = \frac{13}{12}$$

Meanwhile,

$$\frac{\prod_{w_i \in d_1} P\left(w_i|pos\right)}{\prod_{w_i \in d_1} P\left(w_i|neg\right)} = \frac{0.24 \times 0.24 \times 0.08 \times 0.08}{0.25 \times 0.25 \times 0.17 \times 0.17} \cong 0.205$$

As $LR_1 = \frac{13}{12} * 0.205 < 1$, we can classify d_1 as negative.

2) $d_2 =$ "I am learning NLP."
Similar to the last example, we have:

$$LR_2 = \frac{P\left(pos|d_2\right)}{P\left(neg|d_2\right)} = \frac{P(pos)}{P(neg)} * \frac{\prod_{w_i \in d_2} P\left(w_i|pos\right)}{\prod_{w_i \in d_2} P\left(w_i|neg\right)}$$

Here,

$$\frac{P(pos)}{P(neg)} = \frac{\frac{13}{13+12}}{\frac{12}{13+12}} = \frac{13}{12}$$

And,

$$\frac{\prod_{w_i \in d_2} P\left(w_i|pos\right)}{\prod_{w_i \in d_2} P\left(w_i|neg\right)} = \frac{0.24 \times 0.24 \times 0.08 \times 0.08}{0.25 \times 0.25 \times 0.08 \times 0.08} \cong 0.922$$

As $LR_2 = \frac{13}{12} * 0.922 < 1$, we can classify d_2 as negative.

4.4.4 Other Points on Naïve Bayes

1) Smoothing: Like N-gram, we need to normalize $P\left(w_i|c_j\right)$ and bound it away from 0:

$$P\left(w_i|c_j\right) = \frac{count\left(w_i, c_j\right) + 1}{\sum_{w \in V} count\left(w_i, c\right) + |V|}$$

2) Sometimes Naïve Bayes works bad, the reason behind is its two assumptions may not fit the way by which human talk.

4.4.5 The process of Naïve Bayes

The process of Naïve Bayes can be summarized as follows:

1. **Labeling**: Annotate a dataset with class labels (positive, negative, etc.).
2. **Pre-processing**: Pre-process the text to words.
3. **Frequency computation**: Compute the $\text{Freq}(\text{word}, \text{class})$.
4. **Probability computation**: Compute $P(\text{word}|\text{class})$ and $P(\text{document}|\text{class})$.
5. **Classification**.

4.5 NLP Applications in Business Research

4.5.1 Authorship Identification

- Methodology: Naïve Bayes.
- Objectives: Identify the real author for books/articles/papers.
- Example:

  - Mosteller and Wallace (1963): Identify who wrote the *Federalist Papers*, Alexander Hamilton or James Madison?

  - Stock and Trebbi (2003): Identify who invented IV.

4.5.2 Sentiment and Stock Price

- Methodology: Dictionary approach.
- Objectives: Use the word counts in financial/business newspapers (e.g., WSJ) to measure market sentiment. Then, use this sentiment measure forecast stock market activity.
- Example: Tetlock (2007) and Loughran and McDonald (2011).

4.5.3 Measuring Policy Uncertainty

- Methodology: Dictionary approach.
- Objectives: Count the number of news articles containing at least one key word from 3 categories: economy, policy, and uncertainty. Use these counts to predict the level of economic policy uncertainty, defined as the simple average of the counts across different newspapers.
- Example: Baker et al. (2016)

4.5.4 Media Slant

- Methodology: N-grams.
- Objectives: Firstly, identify the frequently and asymmetrically used phrases by Democrats and Republicans. Secondly, predict newspaper slant from the counts of the selected phrases.
- Example: Groseclose and Milyo (2005) and Gentzkow and Shapiro (2010a).

4.5.5 Industry Segmentation

- Methodology: Cosine similarity.
- Objectives: Classify industries based on product descriptions of company disclosure text, the 10-K report. And industries are defined by clustering firms according to their cosine similarities.
- Example: Hoberg and Phillips (2016a)

5

Natural Language Processing (3)

——-**Scribed by Xinyu Li and Qingyu Xu**

5.1 Outline

the outline of this document is as follows:

- In Section 5.2, we introduce the Continuous Bag-Of-Words (CBOW) model.
- In Section 5.3, we introduce another Word2Vec model - the Skip-Gram model.
- In Section 5.4, we introduce the Global Vectors (GloVe).
- In Section 5.5, we discuss how we can evaluate the quality of word vectors produced by such techniques.
- In Section 5.6, we discuss some applications of Word Embedding techniques in Business/Econ Research.

Review and Heads Up

The idea of word representation in deep learning lies in the pre-processing step of NLP tasks. The Pre-processing mainly contains three steps:

- Represent raw text D as a numerical array $\mathbf{C}$;
- Map $\mathbf{C}$ to predicted values $\hat{\mathbf{V}}$ of unknown outcomes $\mathbf{V}$; and
- Map Use $\hat{\mathbf{V}}$ in subsequent descriptive or causal analysis.

The key idea of Word2Vec, concerning text representations, suggests that representing text as a numerical array in various ways may reveal new information that traditional methods cannot.

Limitations of Traditional Text Representation Historically, text representation in NLP has relied on word-document co-occurrence matrices, employing methods such as counting words and documents. While intuitive and interpretable, these methods suffer from several drawbacks:

- They produce sparse representations, particularly with one-hot encoding, leading to inefficient use of space.
- They lack the ability to accurately support advanced downstream tasks due to their simplicity and high dimensionality.

Word2Vec: Embedding Dense Representations Word2Vec introduces an embedding technique that trains dense representations from sparse data, addressing the limitations of traditional methods. Key features include:

- **Efficiency**: By embedding words into a continuous vector space, Word2Vec produces more compact and informative representations.
- **Contextual Richness**: It leverages the context of words—words that appear close together—to encode semantic meanings, aligning with the principle of distributional semantics.

Self-supervised Learning in NLP A cornerstone of Word2Vec's effectiveness is its use of self-supervised learning, where the model is trained on data that is inherently labeled. It operates on two key premises:

1. **The Learning Task**: The model learns rich text representations by predicting the context words (o) given a center word (c), or vice versa. This task exploits the natural structure of language as its own supervision.
2. **Distributional Semantics**: The principle that a word's meaning is defined by the context around it. Word2Vec quantifies this concept by embedding words in a way that words with similar contextual neighbors are brought closer in the embedding space.

5.2 Word2Vec: Continuous Bag of Words (CBOW)

Continuous Bag of Words (CBOW) is a neural network model used in natural language processing (NLP) for word prediction within a sentence or document. Unlike traditional bag-of-words models that disregard order of words, the key idea of Continuous Bag of Words is using the surrounding context (outside words) o to predict the target center word c. For example, in the sentence "I am happy because I am learning" in Fig. 5.1, with a window size of 5, the target word "happy" is predicted based on the context words "I am ... because I", including two words before and after. This model uses a moving window approach, allowing each word in the text to serve as a target word with its surrounding words as context.

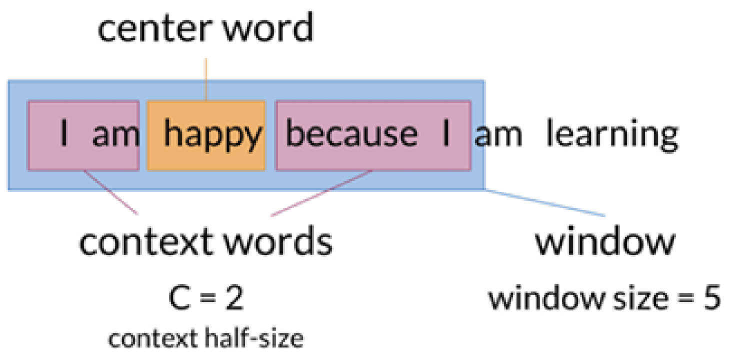


Figure 5.1: Example of sentence in CBOW

Our objective is to minimize the probability loss function : $-\log P(\text{word}_c|\text{word}_o)$

5.2.1 Model Overview

The CBOW model architecture includes three layers: an input layer, a hidden layer, and an output layer. The input layer receives the context of words surrounding a target word. The hidden layer processes this input, and the output layer predicts the target word. The model is optimized to minimize the prediction error, thereby learning word embeddings that encapsulate semantic relationships between words.

5.2.2 Formulas and Mathematical Representation

The mathematical framework of the CBOW model involves the following components:

- V: the size of the vocabulary.
- N: the dimension of the hidden layer. This is decided by your computational resources.
- W_1: weight matrix between the input layer and the hidden layer ($V \times N$), which is also used as the final word embeddings.
- W_2: weight matrix between the hidden layer and the output layer ($N \times V$).
- C: context size, the number of words considered around the target word.
- x: one-hot encoded vectors of context words (V-dimensional).
- $\hat{v}$: hidden layer vector.
- y: output vector (V-dimensional), representing the probability distribution over the vocabulary for the target word.
- y': actual one-hot encoded vector of the target word.

The hidden layer ($\hat{v}$) is computed as the average of the context word vectors transformed by the weight matrix W:

$$\hat{v} = \frac{1}{C} W_1^T \sum_{c=1}^{C} x_c$$

The output layer (y) uses the softmax function to predict the probability distribution over the vocabulary:

$$y = \text{softmax}(W_2^T \hat{v}) = \frac{\exp(W_2^T \hat{v})}{\sum_{j=1}^{V} \exp(W_2^T \hat{v}_j)}$$

As a high level illustration, we have a sentence d containing words $w_1, ..., w_c, ...w_V$, if we use w_c as the center word, then we hope to:

$$\begin{aligned}
\text{minimize} J &= -\log P(w_c | w_{c-m}, ...w_{c-1}, w_{c+1}, \ldots, w_{c+m}) \\
&= -\log P(u_c | \hat{v}) \\
&= -\log \frac{\exp(u_c^T \hat{v})}{\sum_{j=1}^{|V|} \exp(u_j^T \hat{v})} \\
&= -u_c^T \hat{v} + log \sum_{j=1}^{|V|} \exp(u_j^T \hat{v})
\end{aligned}$$

Where u_c is the word embedding of the center word, in W', and for sentence d, the total loss should be $-\log \prod_{c=1}^{V} \frac{\exp(u_c^T \hat{v})}{\sum_{j=1}^{|V|} \exp(u_j^T \hat{v})}$

5.2.3 Loss Function and Training Process

The CBOW model employs the cross-entropy loss function to measure the difference between the predicted probability distribution (y) and the actual distribution (y'):

$$\text{Loss} = -\sum_{i=1}^{V} y'_i \log(y_i)$$

The training process involves:

1. **Forward Pass:** Compute the hidden layer and output predictions based on the current weights.

2. **Compute Loss:** Calculate the loss using the cross-entropy function.

3. **Backward Pass:** Derive the gradients of the loss function with respect to each weight.

4. **Update Weights:** Adjust the weights using a learning rate to minimize the loss.

5. **Iteration:** Repeat these steps for multiple epochs or until the loss converges to a minimal value.

5.2.4 Example

Consider a sentence: “The quick brown fox jumps over the lazy dog”. To predict “fox” with a context size of 2, the input includes [“quick”, “brown”, “jumps”, “over”], aiming to predict “fox”.

1. Encode context words into one-hot vectors.

2. Compute the hidden layer as the average of these vectors, transformed by W.

3. Calculate the output vector y using the softmax function based on the hidden layer.

4. Update the model weights based on the loss between the predicted output and the actual word "fox".

5.3 Word2Vec: Skip-Gram

The Skip-Gram model is a prominent neural network architecture in natural language processing (NLP) for learning word embeddings. It inverts the task of the Continuous Bag of Words (CBOW) model by using a target word to predict context words within a specified window.

5.3.1 Model Overview

Unlike CBOW that predicts a target word from its surrounding context, the Skip-Gram model focuses on predicting the surrounding context words given a target word. This model consists of three layers: an input layer, a hidden layer, and an output layer. The input is the target word, and the output is the context words within a certain window around the target word.

5.3.2 Formulas and Mathematical Representation

Given a target word w_c, the Skip-Gram model aims to maximize the probability of context words within a window size of m on either side of w_c. The mathematical representation involves:

- V: size of the vocabulary.
- N: dimension of the hidden layer.
- W_1, W_2: weight matrices similar to those in CBOW.
- v: one-hot encoded vector of the target word from the input layer.
- $\hat{v}$: hidden layer vector.
- u: output vectors representing the context words.

The objective function for Skip-Gram can be defined as maximizing the average log probability:

$$\begin{aligned}\text{maximize } J &= \frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m, j\neq 0}\log P(w_{t+j}|w_t)\\ &= \frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m, j\neq 0}\log\frac{\exp(v_{w_{t+j}}^T\hat{v}_{w_t})}{\sum_{i=1}^{|V|}\exp(v_i^T\hat{v}_{w_t})}\end{aligned}$$

where T is the total length of the word sequence.

5.3.3 Loss Function and Training Process

The Skip-Gram model uses the cross-entropy loss, similar to CBOW, but with the target and context roles reversed. The training process involves:

1. Forward Pass: Compute the hidden layer and output predictions for context words based on the target word.
2. Compute Loss: Calculate the loss using a formulation that encourages the model to increase the probability of the actual context words.
3. Backward Pass: Update the model weights to maximize the probability of the context words given the target word.
4. Repeat the process for multiple epochs or until convergence.

Deriving Final Word Embeddings. The derivation of final word embeddings in Skip-Gram follows a similar approach to CBOW, utilizing the weight matrix W or the average of W and W'. These embeddings capture the semantic and syntactic relationships between words based on their co-occurrence patterns.

5.3.4 Subsampling of Words

The Skip-Gram model offers a powerful method for learning high-quality word embeddings from large text corpora, making it a cornerstone of modern NLP applications. However, computing the probability distribution over a large vocabulary for each training instance is computationally expensive. To address this issue, techniques such as Subsampling of Words and Negative Sampling are introduced to improve efficiency and learning quality.

Subsampling of words is a technique to reduce the number of training examples, particularly by eliminating the too frequent words that provide less informative value for training. This helps in reducing the training set size and improving the quality of word embeddings for rare words.

Formulation: The probability $P(w_i)$ of keeping a word w_i in the training set is given by:

$$P(w_i) = \left(\sqrt{\frac{z(w_i)}{0.001}} + 1 \right) \cdot \frac{0.001}{z(w_i)}$$

where $z(w_i)$ is the fraction of the total words in the corpus that are w_i. Words with a high frequency have a higher chance of being subsampled and removed from training examples.

5.3.5 Negative Sampling

Negative Sampling is a method to simplify the learning process by training the model to distinguish a target word from a few randomly chosen "negative" words, instead of all words in the vocabulary. This significantly reduces computational complexity and improves the quality of the resulting word vectors, particularly for less frequent words. In other words, it simplifies the optimization problem by reducing the number of output neurons that need to be updated for each training example. Instead of updating all weights in the output layer, only a small subset of 'negative' samples and the 'positive' sample are updated.

Formulation: Given a word pair (target, context) such as ("fox", "quick"), where "quick" is the positive context word for "fox", negative sampling updates the weights for "quick" and a small number (e.g., 5) of randomly chosen negative words not present in the current context. The objective function for negative sampling is given by:

$$\log \sigma(v_{\text{context}}^T v_{\text{target}}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P(w)} [\log \sigma(-v_{w_i}^T v_{\text{target}})]$$

where v_{context} and v_{target} are the embeddings for the context and target words respectively, k is the number of negative samples, σ is the sigmoid function, and w_i are the negative samples drawn according to the noise distribution $P(w)$.

5.4 Global Vectors for Word Representation (GloVe)

While models like Word2Vec (e.g. the skip-gram and the CBOW models) leverage each word's local context windows to infer its embedding, they fail to capture the global co-occurrence statistics apart from local contextual linguistic pattern. That is where Global Vectors for Word Representation (GloVe) comes from Pennington et al. (2014). GloVe consists of a weighted least squares model that trains on **global word-word co-occurrence counts** and makes efficient use of statistics. The model produces a word vector space with meaningful sub-structure.

5.4.1 Co-occurrence Matrix

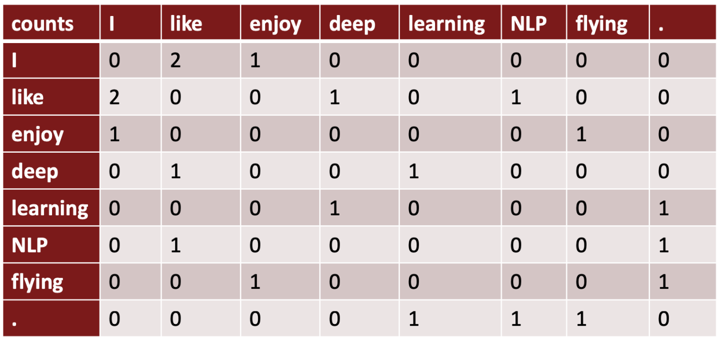
There are two ways to construct co-occurrence matrix, one of which is a window-based method, which counts the word co-occurrence within the window of each word.

- X: word-word co-occurrence matrix
- X_{ij}: number of times word j occur in the context of word i
- $X_i = \sum_k X_{ik}$: the number of times any word k appears in the context of word i
- $P_{ij} = P(w_j|w_i) = \frac{X_{ij}}{X_i}$: the probability of word j appearing in the context of word i

Specifically, we define the "context" of word i as any words in a window around it (irrelevant whether left or right context). The length of this window is commonly set from 5 to 10. Here is an example corpus and its co-occurrence matrix.

Example: Here is a corpus consisting of three sentences.

- I like deep learning
- I like NLP
- I enjoy flying



counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Figure 5.2: Example: co-occurrence matrix (window length = 1, symmetric)

Populating this matrix requires a single pass through the entire corpus to collect the statistics, which can be computationally expensive for a large corpus.

5.4.2 GloVe Model Training

Referring to the idea in skip-gram, the probability of word j appears in the context of word i can be represented as a softmax,

$$Q_{ij} = \frac{\exp\left(\vec{u}_j^T \vec{v}_i\right)}{\sum_{w=1}^{W} \exp\left(\vec{u}_w^T \vec{v}_i\right)}.$$

Our objective is to make this modeled probability approximate the real occurrences in the training corpus. Therefore, the adjusted square loss function can be constructed in the following form:

$$\hat{J} = \sum_{i=1}^{W} \sum_{j=1}^{W} X_i (\hat{P}_{ij} - \hat{Q}_{ij})^2,$$

where $\hat{P}_{ij} = X_{ij}$ and $\hat{Q}_{ij} = \exp(\vec{u}_j^T \vec{v}_i)$. Given the problem that X_{ij} often takes large values which complicates the optimization, the authors introduce log transformation to simplify this problem. Namely, they convert the original loss function into

$$\begin{aligned}\hat{J} &= \sum_{i=1}^{W}\sum_{j=1}^{W} X_i(\log(\hat{P})_{ij} - \log(\hat{Q})_{ij})^2 \\ &= \sum_{i=1}^{W}\sum_{j=1}^{W} X_i(\vec{u}_j^T \vec{v}_i - \log X_{ij})^2.\end{aligned}$$

Finally, we may use a more general weighting factor to include the possible impact of context word as well:

$$\hat{J} = \sum_{i=1}^{W}\sum_{j=1}^{W} f(X_{ij})(\vec{u}_j^T \vec{v}_i - \log X_{ij})^2.$$

A more general version formulated in the paper includes two more bias terms b_i and b_j:

$$\hat{J} = \sum_{i=1}^{W}\sum_{j=1}^{W} f(X_{ij})(\vec{u}_j^T \vec{v}_i + b_i + \hat{b}_j - \log X_{ij})^2.$$

The loss function is derived from employing a log-bilinear model, which, by utilizing differences between word vectors, captures the ratios of co-occurrence probabilities as linear components of meaning within a word vector space (refer to Section 3 in Pennington et al. (2014) for GloVe). Noted that the weighting function $f(x)$ should follow several properties:

- Non-decreasing and $f(0) = 0$: More co-occurrence word pairs should be associated with a larger factor, while no co-occurrence word pairs should not be counted.
- Not-overweighted: $f(x)$ should be relatively small for large values of x

One group of functions that work well under this standard can be parameterized as,

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}.$$

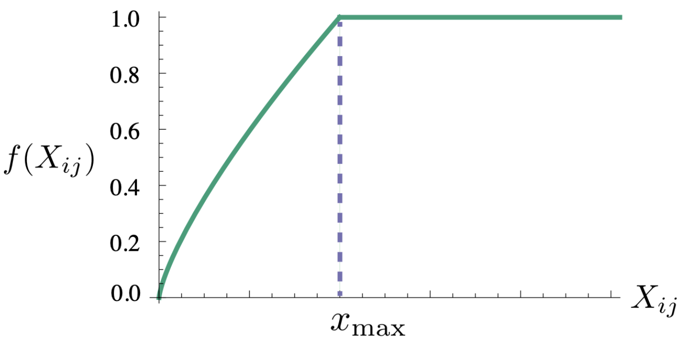


Figure 5.3: Weighting function f with $\alpha = 3/4$

Compared to the original loss function, the final loss function is much more simplified, which allows faster training than Word2Vec on huge corpses.

5.5 Word Vector Evaluations

This section delves into the methodologies employed to quantitatively assess the quality of word vectors or n-gram models. It distinguishes between two primary evaluation strategies—*intrinsic* and *extrinsic* evaluations.

5.5.1 Intrinsic Evaluation

Intrinsic evaluation assesses a model's performance on specific, intermediate tasks that are independent of any real-world application. This method directly measures the quality of word embeddings or language models based on their ability to perform well-defined linguistic tasks.

- **Pros**: It is quick to compute and useful for understanding how well a model captures linguistic properties and relationships.
- **Cons**: It's often unclear how these measures correlate with performance on practical, real-world tasks.
- **Examples**: Tasks such as word analogies (where the model predicts relationships between words) and word similarities (comparing the model's assessment of similarity with human judgments) are common intrinsic evaluation metrics.

5.5.2 Extrinsic Evaluation

Extrinsic evaluation determines the effectiveness of a language model by integrating it into an application and measuring the resulting improvements. This approach evaluates the model based on its contribution to the performance of a broader system in performing real-world tasks.

- **Cons**:
  - It can be computationally expensive, requiring deployment in a fully functioning application to assess performance improvements.
  - It may be difficult to pinpoint whether performance changes are due to the language model itself or how it interacts with other subsystems within the application.
- **Examples**: Named Entity Recognition (NER) is an illustrative example of extrinsic evaluation, where the impact of embedding a language model can be directly observed in the model's ability to accurately identify and classify named entities within text.

To sum up, intrinsic evaluation offers a direct but narrow view of a model's linguistic capabilities, while extrinsic evaluation provides a broader perspective on its utility in practical applications, despite being more complex and resource-intensive to conduct.

5.5.3 Intrinsic Evaluation Example: Word Analogy

Task Definition In a word vector analogy, we are given an incomplete analogy of the form:

$$a : a^* \; :: \; b : \underline{?},$$

where we use the analogy relationship between a and a^* to find the corresponding word b^* to b.

One example could be

$$\text{man} : \text{woman} \; :: \; \text{king} : \underline{\text{queen}}.$$

Evaluation Process To find b^* using a specific word embedding model, there are several approaches. Two commonly used algorithms for solving word analogies are *3CosAdd* and *3CosMul*.

The *3CosAdd* method, as shown in Figure 5.4, identifies b^* by maximizing the cosine similarity between the vector differences, assuming analogy relationships are linear in the vector space:

$$b^* = \underset{b'}{\operatorname{argmax}}(\cos(b', a^* - a + b)).$$

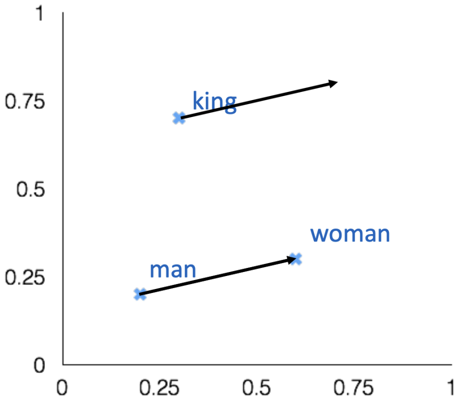


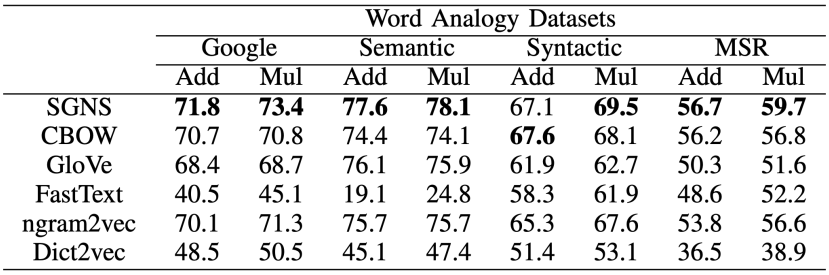
Figure 5.4: A word analogy task example

The *3CosMul* approach, on the other hand, was introduced as an alternative to address some of the limitations of the 3CosAdd method. It uses a multiplicative combination of cosine similarities instead of an additive one. The formula for *3CosMul* seeks to maximize the quantity:

$$b^* = \underset{b'}{\operatorname{argmax}} \frac{\cos(b, b') * \cos(a^*, b')}{\cos(a, b') + \epsilon},$$

where ϵ is a small value added to prevent division by zero.

Empirical studies suggest that *3CosMul* often outperforms *3CosAdd* in certain analogy tasks, especially in cases where linear relationships do not fully explain the word associations.



	Word Analogy Datasets							
	Google		Semantic		Syntactic		MSR	
	Add	Mul	Add	Mul	Add	Mul	Add	Mul
SGNS	**71.8**	**73.4**	**77.6**	**78.1**	67.1	**69.5**	**56.7**	**59.7**
CBOW	70.7	70.8	74.4	74.1	**67.6**	68.1	56.2	56.8
GloVe	68.4	68.7	76.1	75.9	61.9	62.7	50.3	51.6
FastText	40.5	45.1	19.1	24.8	58.3	61.9	48.6	52.2
ngram2vec	70.1	71.3	75.7	75.7	65.3	67.6	53.8	56.6
Dict2vec	48.5	50.5	45.1	47.4	51.4	53.1	36.5	38.9

Figure 5.5: Performance comparison of six word embedding baseline models against word analogy datasets in Wang et al. (2019)

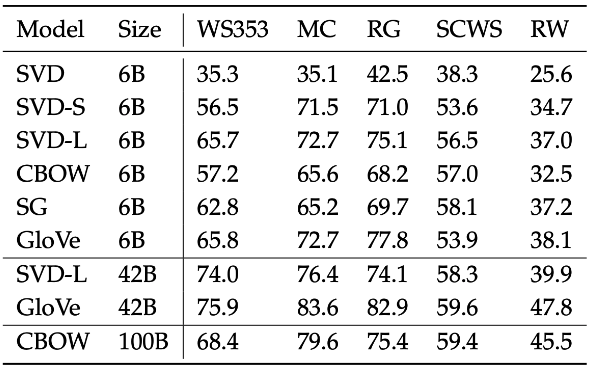
One caveat here is that the process might consistently return “king” as the most analogous word. To address this, a common practice is to exclude the input words from the search.

5.5.4 Intrinsic Evaluation Example: Word Similarity

Another method for intrinsic evaluation of word vectors is assessing word similarity, where word vector distances and correlations are compared with human judgments. Humans are asked to rate the similarity between two words on a fixed scale (e.g., 0-10), and these ratings are then compared to the cosine similarity between the corresponding word vectors:

$$\cos(w_x, w_y) = \frac{w_x \cdot w_y}{\|w_x\| \|w_y\|}.$$

The table in Pennington et al. (2014) demonstrates the correlations between word vector similarities obtained through various embedding techniques and different human judgment datasets. Additional datasets for word similarity tests are available in Wang et al. (2019).



Model	Size	WS353	MC	RG	SCWS	RW
SVD	6B	35.3	35.1	42.5	38.3	25.6
SVD-S	6B	56.5	71.5	71.0	53.6	34.7
SVD-L	6B	65.7	72.7	75.1	56.5	37.0
CBOW	6B	57.2	65.6	68.2	57.0	32.5
SG	6B	62.8	65.2	69.7	58.1	37.2
GloVe	6B	65.8	72.7	77.8	53.9	38.1
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	75.9	83.6	82.9	59.6	47.8
CBOW	100B	68.4	79.6	75.4	59.4	45.5

Figure 5.6: Correlations of different word embedding techniques against word similarity test datasets

5.5.5 Extrinsic Evaluation Example: Named Entity Recognition

Named Entity Recognition (NER) is a task aimed at identifying and classifying named entities—such as names of people, organizations, locations, dates, etc.—within text into predefined categories. NER acts as a crucial step for several advanced NLP applications, including question answering, text summarization, and knowledge extraction. For instance, given the input "Philip lives in Shenzhen and Shanghai in 2024," we would expect a classified output like "[Philip]$_{\text{Person}}$ lives in [Shenzhen]$_{\text{Location}}$ and [Shanghai]$_{\text{Location}}$ in [2024]$_{\text{Time}}$."

The fundamental concept behind NER is to categorize each word within its contextual window of neighboring words. A logistic classifier is trained on manually labeled data to determine whether the center word belongs to a particular class (yes/no)[1] based on the concatenated word vectors from a window.

Example: Classify “Paris” as +/– location in context of sentence with window length 2.

the museums in Paris are amazing to see .

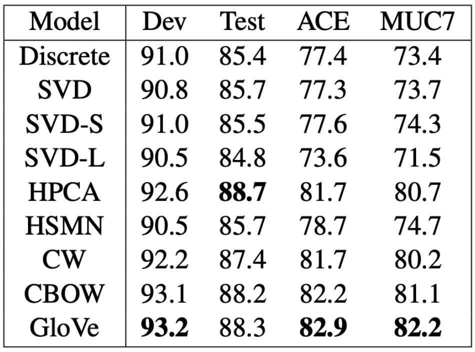
$$\mathbf{X}_{\text{window}} = \begin{bmatrix} X_{\text{museums}} & X_{\text{in}} & X_{\text{Paris}} & X_{\text{are}} & X_{\text{amazing}} \end{bmatrix}^T$$

- Resulting vector $\mathbf{X}_{\text{window}} = \mathbf{X} \in \mathbb{R}^{5\times d}$

[1] In actual NER tasks, a multi-class softmax function is typically employed to convert the network’s output into a probability distribution.

- To classify all words: run classifier for each class on the vector centered on each word in the sentence

The following table in Pennington et al. (2014) presents the F1 score on NER task with 50-dimensional vectors under different text embedding models. The GloVe model outperforms all other methods on all evaluation metrics, except for the CoNLL test set, on which the HPCA method does slightly better.



Model	Dev	Test	ACE	MUC7
Discrete	91.0	85.4	77.4	73.4
SVD	90.8	85.7	77.3	73.7
SVD-S	91.0	85.5	77.6	74.3
SVD-L	90.5	84.8	73.6	71.5
HPCA	92.6	**88.7**	81.7	80.7
HSMN	90.5	85.7	78.7	74.7
CW	92.2	87.4	81.7	80.2
CBOW	93.1	88.2	82.2	81.1
GloVe	**93.2**	88.3	**82.9**	**82.2**

Figure 5.7: F1 score on NER task with 50d vectors in Pennington et al. (2014)

5.6 Applications of Word2Vec in Business/Econ Research

5.6.1 Overview: Text Algorithms in Economics

Ash and Hansen (2023) identify four main empirical tasks that encompass most text-as-data research in economics:

1. **Measuring document similarity**: Identifying how similar two documents are. This similarity measurement can serve as a proxy for economic relevance, such as comparing firms' product descriptions to gauge competition (Hoberg and Phillips, 2016b, 2010).

2. **Concept detection**: Detecting economically significant concepts within textual data, like economic policy uncertainty (Baker et al., 2016), or skill demand (Deming and Kahn, 2018). This involves using various methods, including dictionary-based pattern matching, algorithmic approaches for associating documents with concepts, and machine predictions based on human annotations

3. **How concepts are related**: Understanding the relationships between concepts within a corpus, for example, the association between sentiment and economic conditions (Apel and Grimaldi, 2014), or risk and political exposure (Hassan et al., 2019).

4. **Associating text with metadata**: Leveraging metadata that comes with text to measure specific outcomes, such as using political affiliations or economic conditions mentioned in documents to impute values to other documents. For instance, Gentzkow and Shapiro (2010b) use regressions to map speech to predicted labels or political bias to media outlets based on their text.

Besides, the authors discuss two problems that have come up repeatedly in the use of text algorithms:

1. **Validation**: The challenge lies in ensuring the methods used are reliable and accurate, necessitating comparisons across different approaches to find the most suitable one for economic research.

2. **Interpretability**: There is a need to strike a balance between model performance and the ability to interpret their outputs, crucial for making informed decisions based on model predictions.

Finally, they conclude the possibilities with the new large pre-trained language models like BERT and GPT-4 in economics:

- LLMs' potential to include multilingual texts in empirical analysis, overcoming the limitation of English-only text analysis.

- LLMs may revolutionize language-related research tasks in economics, e.g., generating scientific abstracts and assisting in tasks like software development, data wrangling, and potentially even supporting the peer review process.

5.6.2 Corporate Culture Measurement

An example connecting with the previously mentioned task "concept detection" is the innovative use of Word2Vec by Li et al. (2021) to analyze corporate culture via earnings call transcripts.

This approach, which constructs a culture dictionary from word associations with key "value words" (*innovation*, *integrity*, *quality*, *respect*, and *teamwork*) under the trained word embeddings, enhances scalability and depth of analysis over traditional manual categorization.

Their analyses shows that strong corporate culture correlates with greater operational efficiency, more risk-taking, less earnings management, and higher firm value.

5.6.3 Product2Vec

Another application of leveraging skip-gram in business research is Product2Vec. Chen et al. (2022) use the idea of skip-gram to learn the latent product embeddings based on other products in the choice set. The skip-gram model is used to generate product embeddings by treating shopping baskets as sentences and products as words. This analogy allows the Product2Vec method to capture semantic similarities between products based on their co-occurrence in shopping baskets, akin to how word embeddings capture semantic similarities between words based on their context in text.

This paper presents a theoretical foundation demonstrating a direct relationship between product vectors generated by Product2Vec and the product attributes. This link is established through formal proof, suggesting that product embedding clusters can be interpreted as combinations of product attributes.

The paper extends the utility of Product2Vec to causal inference by integrating product vectors with traditional choice models. This integration demonstrates an improvement in model accuracy—both in terms of model fit and unbiased price coefficients—compared to models relying solely on observable attributes or those incorporating a fixed effect for every product. The approach achieves similar results to more complex models while significantly reducing the number of parameters needed for estimation. This scalable method of causal inference offers a robust way to understand consumer choice dynamics and price sensitivities across large product assortments, providing a valuable tool for marketers and researchers in analyzing and predicting consumer behavior in detail.

Product2Vec-Email Similarly, Grbovic et al. (2015) leverage the idea of skip-gram to learn the latent product embeddings based on purchases from email promotions.

In the paper, the Product2Vec applies the skip-gram model to e-commerce by treating shopping baskets as sentences and products as words. This method embeds products into a low-dimensional vector space, where products frequently purchased together are positioned closely, revealing their contextual and semantic relationships.

- **Objective**: To model consumer product choices within large assortments by learning latent product attributes.
- **Methodology**: Utilizes the skip-gram model to generate product embeddings based on their occurrence in shopping baskets.
- **Applications**: Enhances product recommendation systems by leveraging embeddings to identify complementarity and exchangeability among products.

Extending Product2Vec, they also introduce “Bagged-Prod2Vec”, which has a modification to account for the simultaneous purchase of multiple products. This approach adopts a "shopping bag" concept, optimizing a modified objective function over e-mail receipt sequences to better capture the collective context of product purchases.

- **Improvement**: Addresses the limitations of treating products independently by considering the aggregated context of purchases within e-mail receipts.
- **Technique**: Enhances product embeddings by incorporating temporal aspects of purchases and improving the diversity and relevance of recommendations through clustering.

Other Applications of Product2Vec

- P2V-MAP: Mapping market structures for large retail assortments (Gabel et al., 2019).
- Scalable bundling via dense product embeddings (Kumar et al., 2020).
- Item2Vec: neural item embedding for collaborative filtering (Barkan and Koenigstein, 2016).

6

Natural Language Processing (4)

——**Scribed by ZHANG Jiayi**

6.1 Outline

This section provides an introduction to the most commonly used techniques in NLP during the period from 2013 to 2017, focusing on Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Sequence-to-Sequence (Seq2Seq) models. These methods represent a significant shift towards models that can handle sequential data, thereby offering substantial improvements over previous approaches in tasks such as text translation, summarization, and question answering.

- In Section 6.2, we introduce the Recurrent Neural Network (RNN), a type of neural network that is particularly well-suited for sequential data processing. RNNs are fundamental in understanding the temporal dynamics of language.

- In Section 6.3, we delve into Long Short-Term Memory (LSTM) networks, an evolution of RNNs designed to address the issue of long-term dependencies by introducing a more complex architecture with memory cells.

- In Section 6.4, we discuss the Sequence-to-Sequence (Seq2Seq) model, a framework built upon RNNs and LSTMs for transforming sequences from one domain to another, exemplified by its application in machine translation and other sequence transformation tasks.

- In Section 6.5, I share the link of the code demo of the section in Google Colab.

6.2 Vanilla Recurrent Neural Nets (RNN)

6.2.1 Basic Concepts of RNN

- RNN stands for Recurrent Neural Nets.

- A RNN is similar to a feedforward neural network, except it has connections pointing backward.

- Consider the simplest RNN with one neuron receiving inputs, producing an output, and sending that hidden state back to itself.

- At each time step t, this recurrent neuron receives the inputs $x_{(t)}$ as well as its own hidden state the previous time step, h_t.

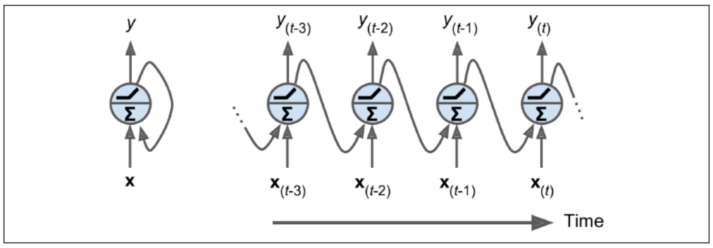


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

Figure 6.1: A recurrent neuron (left) unrolled through time

- The representation above is called unrolling the network through time (it is the same recurrent neuron represented once per time step).
- Let's create a layer of recurrent neurons. At each time step t, every neuron receives both the input vector $x_{(t)}$ and the hidden state $h_{(t-1)}$ from the previous time step.

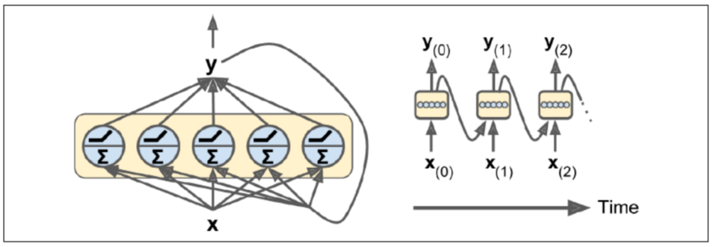


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Figure 6.2: A layer of recurrent neurons (left) unrolled through time (right)

- Each recurrent neuron has three sets of weights: one for the inputs W_x, one for the hidden state W_h and the other for the outputs U.

$$h_t = \phi\left(W_x^T X_{(t)} + W_h^T h_{(t-1)} + b_1\right)$$
$$Y_{(t)} = \text{softmax}(Uh_t + b_2)$$

where b_1 and b_2 is the bias vector and $\phi()$ is the activation function.

6.2.2 RNN in NLP

Why do we use RNN in NLP? Consider two sentences:

- This is not a real restaurant, it's a filthy burger joint.
- This is not a filthy burger joint, it's a real restaurant.

If we only use word embeddings for sentiment classification, these two opposite sentences will generate the same sentence vector (i.e., the sentence vector is the sum of the word vectors for all words in the sentence) - **Sequence matters!**

How does RNN work in NLP? Figure 6.12 illustrates the structure and working of a Recurrent Neural Network (RNN) in the context of Natural Language Processing (NLP). The RNN is used for processing sequences of words to predict the next word in a sequence.

1. Word Embeddings: Each word in the input sequence is represented as a one-hot vector $\mathbf{x}^{(t)} \in \mathbb{R}^{|V|}$, where $|V|$ is the size of the vocabulary. These one-hot vectors are then transformed into word embeddings $\mathbf{e}^{(t)}$ using the embedding matrix $\mathbf{E}$:

$$\mathbf{e}^{(t)} = \mathbf{E}\mathbf{x}^{(t)}$$

2. Hidden States. The RNN maintains hidden states $\mathbf{h}^{(t)}$ which capture information from previous time steps. The initial hidden state $\mathbf{h}^{(0)}$ is usually initialized to zero. For each time step t, the hidden state is updated using the following equation:

$$\mathbf{h}^{(t)} = \sigma\left(\mathbf{W}_h\mathbf{h}^{(t-1)} + \mathbf{W}_e\mathbf{e}^{(t)} + \mathbf{b}_1\right)$$

Here, σ represents the activation function (such as tanh or ReLU), $\mathbf{W}_h$ is the recurrent weight matrix, $\mathbf{W}_e$ is the weight matrix for the word embeddings, and $\mathbf{b}_1$ is the bias term.

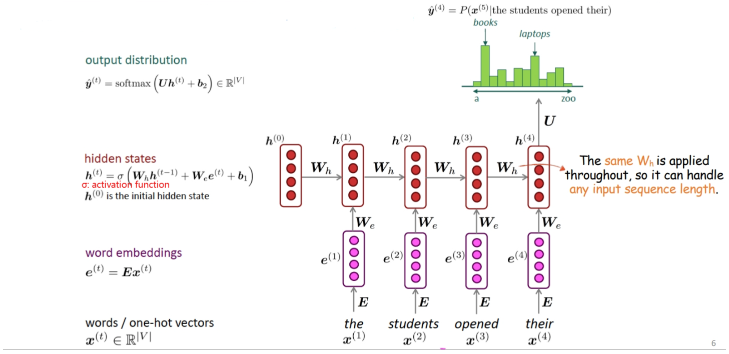
3. Output Distribution. At each time step, the RNN produces an output distribution $\hat{\mathbf{y}}^{(t)}$ over the vocabulary, which is computed using the hidden state and an output weight matrix $\mathbf{U}$:

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}\left(\mathbf{U}\mathbf{h}^{(t)} + \mathbf{b}_2\right)$$

The softmax function ensures that the output distribution sums to 1, making it suitable for predicting probabilities over the vocabulary.

4. Example. The example sentence "the students opened their" is processed as follows:

- Each word ("the", "students", "opened", "their") is converted into a one-hot vector.
- These one-hot vectors are transformed into word embeddings $\mathbf{e}^{(t)}$.
- The hidden state $\mathbf{h}^{(t)}$ is updated sequentially using the embedding of the current word and the hidden state from the previous time step.
- Finally, the output distribution $\hat{\mathbf{y}}^{(4)}$ is computed, which gives the probabilities of the next word being "books", "laptops", etc.



$\hat{y}^{(4)} = P(x^{(5)}|\text{the students opened their})$
books
laptops
a
zoo
output distribution
$\hat{y}^{(t)} = \text{softmax}\left(Uh^{(t)} + b_2\right) \in \mathbb{R}^{|V|}$
U
hidden states
$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$
σ: activation function
$h^{(0)}$ is the initial hidden state
$h^{(0)}$
$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$
W_h
The same W_h is applied throughout, so it can handle any input sequence length.
W_e
word embeddings
$e^{(t)} = Ex^{(t)}$
$e^{(1)}$
$e^{(2)}$
$e^{(3)}$
$e^{(4)}$
E
words / one-hot vectors
$x^{(t)} \in \mathbb{R}^{|V|}$
the
$x^{(1)}$
students
$x^{(2)}$
opened
$x^{(3)}$
their
$x^{(4)}$
6

Figure 6.3: RNN in NLP

How to train RNN? The whole process can be summarised as six steps, including “Data Preparation", “Model Architecture", “Model Compilation", “Training", “Evaluation and Tuning" and “Deployment". The detailed introduction can be seen as follows:

1. Data Preparation

   - Dataset Collection: Gather a dataset relevant to your NLP task, such as a corpus of text for language modeling, a collection of sentences labeled with sentiments, or parallel corpora for translation.

   - Text Preprocessing: Clean the text by removing unnecessary characters, lowercasing, and possibly correcting spelling. Tokenization is also crucial, which involves splitting text into words or subword units.

   - Vectorization: Convert text into numerical form using techniques like one-hot encoding or word embeddings (e.g., Word2Vec, GloVe). Embeddings are preferred as they capture semantic relationships between words.

   - Sequence Padding: Since RNNs require input sequences of the same length, pad shorter sequences with zeros or truncate longer ones.

2. Model Architecture

   - Define the RNN Structure: Choose between simple RNNs, LSTMs (Long Short-Term Memory), or GRUs (Gated Recurrent Units). LSTMs and GRUs are better at capturing long-range dependencies and avoiding vanishing gradient problems.

   - Layer Configuration: Start with an embedding layer if you're not using pre-trained embeddings. Then, add one or more recurrent layers according to your task's complexity. Optionally, include dropout layers to prevent overfitting.

   - Output Layer: The last layer should match your task's requirements, such as a softmax layer for classification or a linear layer for regression.

3. Model Compilation

   - Choose an Optimizer: Common choices include Adam, RMSprop, and SGD. Adam is often preferred due to its adaptive learning rate properties.

   - Layer Configuration: Start with an embedding layer if you're not using pre-trained embeddings. Then, add one or more recurrent layers according to your task's complexity. Optionally, include dropout layers to prevent overfitting.

   - Output Layer: The last layer should match your task's requirements, such as a softmax layer for classification or a linear layer for regression.

4. Training

   - Batch Size and Epochs: Choose appropriate batch sizes and the number of epochs based on your dataset size and model complexity. Larger batch sizes can reduce training time but might affect model convergence and performance.

   - Backpropagation Through Time (BPTT): RNNs are trained using a variant of backpropagation called BPTT, where gradients are calculated and weights are updated across time steps.

   - Regularization and Dropout: Apply techniques to prevent overfitting, especially in complex models or when training on small datasets.

5. Evaluation and Tuning

- Validation Set: Use a portion of your dataset to validate the model during training. This helps in tuning hyperparameters and avoiding overfitting.
- Early Stopping: Monitor performance on the validation set and stop training when performance stops improving to prevent overfitting.
- Hyperparameter Tuning: Experiment with different architectures, hyperparameters (e.g., learning rate, number of layers, hidden units), and training strategies to find the best model.

6. Deployment

- Final Evaluation: Test your model on a separate test set to evaluate its performance on unseen data.
- Integration: Integrate your trained model into an application or workflow, ensuring it can process input data and return predictions effectively.

Let's delve deeper into the loss function in RNN now. Loss functions in step t is the cross-entropy between the true 1-hot and the predicted distribution:

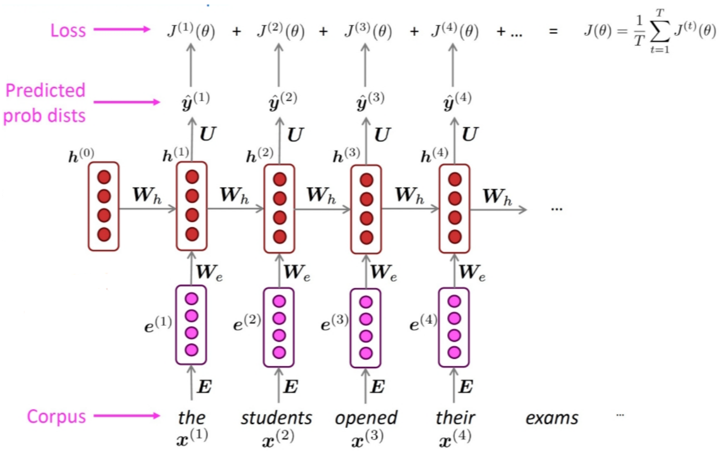


Figure 6.4: Loss function of RNN

$$\mathcal{J}^{(t)}(\theta) = CE(y^{(t)}, \hat{y}^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{y_{t+1}}^{(t)} \tag{6.1}$$

So, the overall loss for the entire training corpus is:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} \mathcal{J}^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{y}_{y_{t+1}}^{(t)} \tag{6.2}$$

We can see RNN is a kind of self-supervised learning.

Since computing the loss and the gradients across the entire corpus is computationally too expensive, in practice, we leverage the idea of SGD: Compute loss and gradients, and update weights with batches of words/sentences. Then repeat on a new batch of sentences.

RNN Advantages

- Can process any input sequence length.
- Computations (in theory) use information from many steps back.
- Same weights are applied to every step, so there's time-symmetry/invariance in how inputs are processed

RNN Disadvantages

- Recurrent computations are slow.
- In practice, it is challenging to access information from many steps back.

Vanishing (and Exploding) Gradient in RNN Vanishing and exploding gradients are two common problems encountered when training Recurrent Neural Networks (RNNs), due to their use of backpropagation through time (BPTT) for training.

Vanishing Gradient Problem The vanishing gradient problem occurs when the gradient of the loss function shrinks exponentially as it is propagated backward through the time steps of the RNN. Each time the gradient is passed back through a recurrent layer, it is multiplied by the weight matrix. If the weights are small (less than 1), then the gradient can shrink very rapidly as it is propagated backward through many time steps. As a result, the gradients become very small, effectively preventing the weights from changing their values. This means that during training, the early layers of the RNN do not learn effectively, which is particularly problematic for long sequences where the context from the earlier time steps might be important.

Exploding Gradient Problem Conversely, the exploding gradient problem occurs when the gradients grow exponentially during backpropagation. If the weights in the RNN are large (greater than 1), the gradients can become very large as they are propagated backward through the network. This can lead to very large weight updates that can cause the model to oscillate wildly, or diverge, rather than converge to a solution.

Solutions to Vanishing and Exploding Gradients

- Weight Initialization: Choosing a proper weight initialization method can help in preventing vanishing and exploding gradients.
- Gradient Clipping: This technique involves setting a threshold value for gradients and scaling them to not exceed this threshold, which can prevent gradients from becoming too large.

Algorithm 11 Pseudo-code for norm clipping

```
$\hat{g} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{g}\| \geq$ threshold then
    $\hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|}\hat{g}$
end if
```

- Use of Gated Units: Using gated units in RNNs, like Long Short-Term Memory (LSTM) units or Gated Recurrent Units (GRUs), can help alleviate the vanishing gradient problem because they are designed to have more persistent memory and thus can remember longer sequences.

- Skip Connections: Introducing skip (or residual) connections that bypass one or more layers can help mitigate the vanishing gradient problem by allowing the gradient to flow through the network more directly.
- Regularization Techniques: These can sometimes help manage exploding gradients by penalizing large weights.
- Proper Choice of Activation Function: Activation functions that are less prone to saturating can help, such as ReLU (Rectified Linear Unit) and its variants, which are less likely to cause vanishing gradients than sigmoid or tanh functions.
- Shorter Sequences: Truncating sequences or using shorter sequences for training can reduce the depth of the computational graph over which gradients are propagated and can help with both vanishing and exploding gradients.

6.2.3 Evaluating Language Models

- Perplexity: The standard metric for evaluating a language model.
- Perplexity is the exponential of the cross entropy loss, so the lower the better:

$$\text{Perplexity} = \left(\prod_{t=1}^{T}\left(\frac{1}{\hat{y}_{x_{t+1}}^{(t)}}\right)\right)^{\frac{1}{T}} = \exp\left(\frac{1}{T}\sum_{t=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)}\right) = \exp(J(\theta)) \tag{6.3}$$

- RNNs greatly improve perplexity over n-grams.

6.2.4 Multi-layer (Stacked) RNN

- RNNs are already deep on time dimension.
- We can also deepen them in another dimension by applying multiple RNNs, allowing for more complex representations and achieving better performances.
- In neural machine translation, 2-4 layers is best for encoder RNNs; 4 layers is best for decoder RNNs.
- 2 layers is much better than 1, but 3 may be a little better than 2.
- Transformer-based nets are usually much deeper.

6.2.5 Application of RNN

- Inventory Management: Use multi-quantile RNNs to provide end-to-end predictions from features to the optimal inventory decisions, whereas most of the literature applies the predict-then-optimize paradigm. A field experiment shows that the e2e approach substantially reduces the inventory costs compared with some naïve benchmarks.
- Detecting FTD: Embeddings from LSTM language models (ELMo) can more accurately detect/predict FTD than individual indicators (benchmark: coherence model, which can somehow be viewed as traditional NLP method).
- Machine Translation: RNNs are employed in machine translation to convert text from one language to another. They can capture the context of the input sentences to provide accurate translations, and when combined with an attention mechanism, they become even more powerful, allowing for translation with greater context understanding and accuracy.

- Speech Recognition: RNNs can be applied to convert spoken language into text. They analyze the audio signal over time and can capture the temporal dependencies in the spoken words, making them effective for speech to text applications.
- Sentiment Analysis: By analyzing the sequence of words in text data, RNNs can classify the sentiment conveyed in the text, such as positive, negative, or neutral. This is particularly useful for monitoring brand and product sentiment on social media and review sites.
- Text Summarization: RNNs can be used to produce concise summaries of longer texts or documents by learning to identify the most relevant information in the original text. This is useful for generating news summaries, executive summaries of business documents, or reducing the size of texts for faster processing.
- Named Entity Recognition (NER): RNNs, particularly when combined with models like LSTM (Long Short-Term Memory), are effective at identifying names, organizations, locations, and other specific entities within text, which is valuable for information extraction and organizing data for further processing.
- Part-of-Speech Tagging: RNNs can be used to assign parts of speech to each word in a sentence, like nouns, verbs, adjectives, etc. This is fundamental for many NLP tasks that require understanding of the grammatical structure of sentences.
- Question Answering: RNNs are integral to question answering systems, where the model generates answers to questions posed in natural language. They can understand the context of the question and search a given text document to find and return the appropriate answer.

6.3 Long Short-Term Memory (LSTM)

6.3.1 The Problem of Long-Term Dependencies

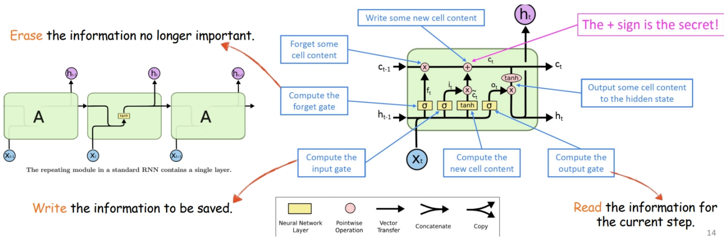
- To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network.
- It may suffer from the unstable gradients problem: it may take forever to train, or training may be unstable.
- In addition, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence.
- Long Short Term Memory networks (LSTM) - are a special kind of RNN, capable of learning long-term dependencies.

6.3.2 basic Concepts of LSTM

- Long Short-Term Memory (LSTM) networks are a special kind of Recurrent Neural Network (RNN) capable of learning long-term dependencies. They were introduced by Hochreiter and Schmidhuber in 1997, and since then, they have been improved and widely applied in various fields, particularly in sequence prediction problems.
- LSTM is a very commonly used RNN architecture that solves the vanishing gradient problem by disregarding the irrelevant past information based on the current information.
- LSTM was the dominant approach for most NLP tasks in 2013-2015.

6.3.3 LSTM Model details

- LSTM cell looks like a regular cell, except that its state is split into two vectors: $h_{(t)}$ (hidden state) and $c_{(t)}$ (“c” refers to “cell”).
- Think of $h_{(t)}$ as the short-term state and $c_{(t)}$ the long-term state.
  - The LSTM model can *read*, *erase*, and *write* information from the cell state, much like RAM.
  - Read/erase/write is controlled by three corresponding gates, which take values between 0 (closed) and 1 (open) and are dynamically computed based on the current context.
- The current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$ are fed to four different fully connected layers.



Erase the information no longer important.

Write the information to be saved.

Read the information for the current step.

The repeating module in a standard RNN contains a single layer.

Write some new cell content

Forget some cell content

Compute the forget gate

Compute the input gate

Compute the new cell content

Compute the output gate

Output some cell content to the hidden state

The + sign is the secret!

Figure 6.5: Details of LSTM model

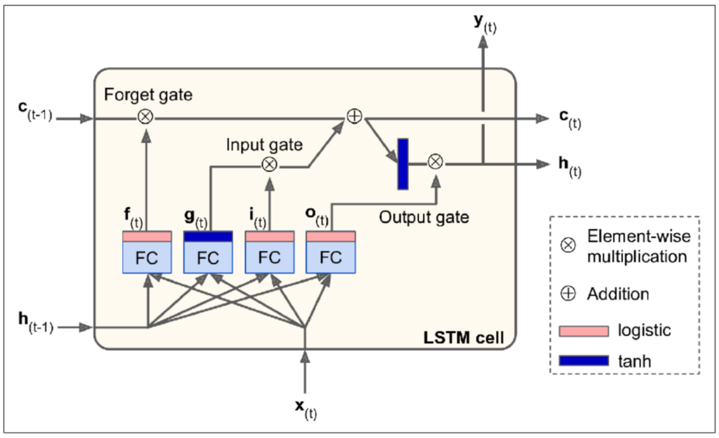
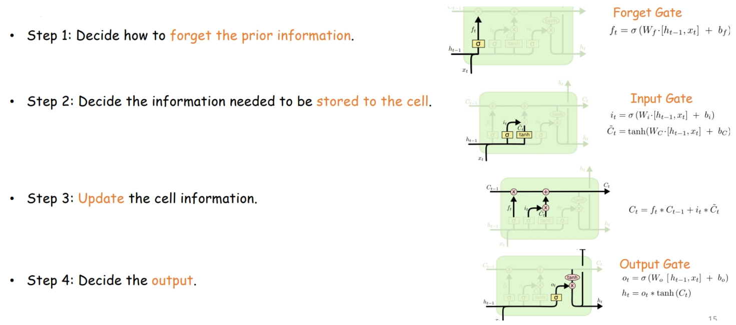


Figure 15-9. LSTM cell

Figure 6.6: LSTM cell

- The main layer is the one that outputs $g_{(t)}$.
  - It analyzes the current inputs $x_{(t)}$ and the previous (short-term) state $h_{(t-1)}$.
  - In a standard RNN cell, its output goes straight out to $y_{(t)}$ and $h_{(t)}$.
  - In an LSTM cell, this layer’s output does not go straight out, but instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are gate controllers with logistic activation function.

  - Their outputs are fed to element-wise multiplication operations, so if they output 0s they close the gate, and if they output 1s they open it.
  - The forget gate (controlled by $f_{(t)}$) controls which parts of the long-term state are erased.
  - The input gate (controlled by $i_{(t)}$) controls which parts of g(t) are added to the long-term state.
  - The output gate (controlled by $o_{(t)}$) controls which parts of the long-term state are read and output at this time step, both to $h_{(t)}$ and to $y_{(t)}$.



- Step 1: Decide how to forget the prior information.
- Step 2: Decide the information needed to be stored to the cell.
- Step 3: Update the cell information.
- Step 4: Decide the output.

Forget Gate

$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$

Input Gate

$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$

$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$

$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

Output Gate

$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$

$h_t = o_t * \tanh\left(C_t\right)$

Figure 6.7: Steps of LSTM model

6.3.4 LSTM: Mathematical Representation

$$
\begin{aligned}
i_t &= \sigma(W_{xi}X_t + W_{hi}h_{(t-1)} + b_i)\\
f_t &= \sigma(W_{xf}X_t + W_{hf}h_{(t-1)} + b_f)\\
o_t &= \sigma(W_{xo}X_t + W_{ho}h_{(t-1)} + b_o)\\
g_t &= \tanh(W_{xg}X_t + W_{hg}h_{(t-1)} + b_g)\\
c_t &= f_t \odot c_{(t-1)} + i_t \odot g_t\\
Y_t = h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Where:

- $W_{xi}, W_{xf}, W_{xo}, W_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector X_t.
- $W_{hi}, W_{hf}, W_{ho}, W_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $h_{(t-1)}$.
- b_i, b_f, b_o, and b_g are the bias terms for each of the four layers.
- σ denotes the sigmoid function and $\odot$ denotes the Hadamard product (element-wise multiplication).
- Note that TensorFlow initializes b_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

6.3.5 Training RNN or LSTM: Backpropagation Through Time

Backpropagation Through Time (BPTT): the application of the Backpropagation algorithm to RNN. 1. Present a sequence of time steps of input and output pairs to the network. 2. Unroll the network then calculate and accumulate errors across each time step. 3. Roll-up the network and update weights. 4. Repeat. BPTT can be computationally expensive as the number of time steps increases.

- If input sequences contains thousands of time steps, then this will be the number of derivatives required for a single weight update.
- This can cause weights to vanish or explode and make slow learning and model skill noisy.
- One way to minimize the exploding and vanishing gradient issue is to limit how many time steps before an update to the weights is performed.

Truncated Backpropagation Through Time (TBPTT) is a modified version of the BPTT training algorithm where the sequence is processed one time step at a time and periodically an update is performed back for a fixed number of time steps. 1. Present a sequence of k1 time steps of input and output pairs to the network. 2. Unroll the network, then calculate and accumulate errors across k2 time steps. 3. Roll-up the network and update weights. 4. Repeat k1: The number of forward-pass time steps between updates. It influences how slow or fast training will be, given how often weight updates are performed. k2: The number of time steps to which to apply BPTT. It should be large enough to capture the temporal structure for the network to learn. Too large a value results in vanishing gradients.

6.3.6 Application of LSTM

Text Generation. LSTM networks can learn the structure of language and generate new text that is similar in style to the input text. This is useful in applications such as chatbots, automatic story generation, and more.

$$P(\text{next word}|\text{previous words}) = \text{LSTM}(\text{previous words}) \tag{6.4}$$

Sentiment Analysis. By processing sequences of words in reviews or comments, LSTMs can classify the sentiment expressed as positive, negative, or neutral.

$$\text{Sentiment} = \text{LSTM}(\text{review text}) \tag{6.5}$$

Machine Translation. LSTMs are used in machine translation systems to convert text from one language to another by understanding the context and semantics of the source language and generating appropriate text in the target language.

$$\text{Target text} = \text{LSTM}(\text{Source text}) \tag{6.6}$$

Named Entity Recognition (NER). LSTMs can identify and classify named entities in text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.

$$\text{Entities} = \text{LSTM}(\text{Input text}) \tag{6.7}$$

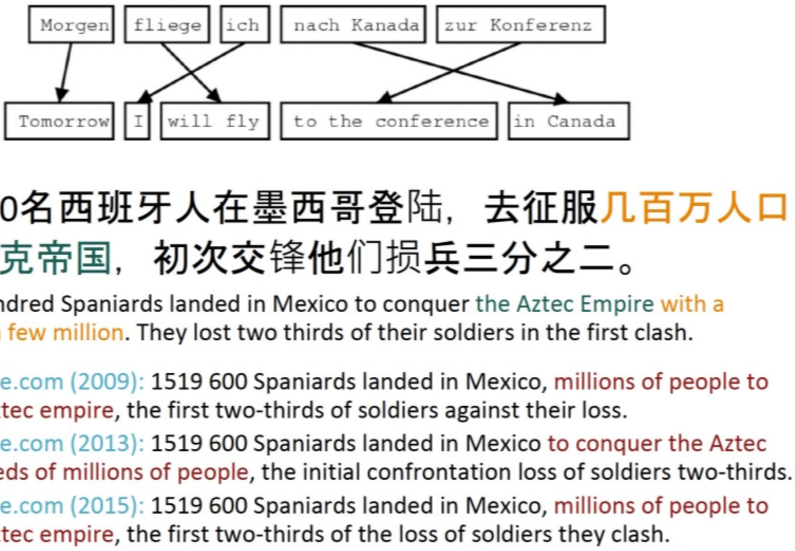
Speech Recognition. Though not strictly a text-based application, LSTMs are crucial in converting spoken language into text by modeling the temporal dependencies in acoustic signals.

$$\text{Text} = \text{LSTM}(\text{Audio input}) \tag{6.8}$$

6.4 Sequence-to-sequence (Seq2seq)

6.4.1 Neural Machine Translation (NMT)

- NMT is a way to do machine translation with a single end-to-end neural network: Sequence-to-sequence (seq2seq), which involves 2 RNNs.
- Machine translation is highly nontrivial and once was a huge research field in CS and NLP.



1519年600名西班牙人在墨西哥登陆，去征服几百万人口的阿兹特克帝国，初次交锋他们损兵三分之二。

In 1519, six hundred Spaniards landed in Mexico to conquer the Aztec Empire with a population of a few million. They lost two thirds of their soldiers in the first clash.

translate.google.com (2009): 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of soldiers against their loss.

translate.google.com (2013): 1519 600 Spaniards landed in Mexico to conquer the Aztec empire, hundreds of millions of people, the initial confrontation loss of soldiers two-thirds.

translate.google.com (2015): 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of the loss of soldiers they clash.

Figure 6.8: NMT Example

6.4.2 Basic Concepts of Seq2seq

- Sequence-to-Sequence (Seq2Seq) models are a class of deep neural network architectures designed to transform sequences from one domain into sequences in another domain, making them especially suited for applications such as machine translation, text summarization, speech recognition, and question-answering systems. The Seq2Seq model typically consists of two main components: an encoder and a decoder, both of which are often implemented using Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), or Gated Recurrent Units (GRUs).
- Encoder: The encoder takes the input sequence and processes it one element at a time, effectively compressing the information contained in the input sequence into a single, fixed-length "context vector." This context vector is intended to represent the entire input sequence in a condensed form, capturing its essential features. In practice, the last hidden state of the encoder is often used as this context vector, especially in simpler Seq2Seq models.

- Decoder: The decoder is tasked with generating the output sequence from the context vector. It starts with a special start-of-sequence token and generates the output one element at a time, based on the context vector and the previously generated elements. The process continues until the decoder generates an end-of-sequence token or reaches a predefined length.

- Training: During training, Seq2Seq models are typically trained end-to-end with a technique called Teacher Forcing, where the true output sequence (up to the current step) is fed into the decoder to predict the next element in the sequence. This approach helps stabilize training and improves the model's performance by providing the correct context during the learning process.

- Seq2seq is a Conditional Language Model: Predicting the next word of the target sentence y conditioned on the source sentence x and prior texts.

  $$P(y|x) = P(y_1|x) \prod_{t=2}^{T} P(y_t|y_1, \dots, y_{t-1}, x) \tag{6.9}$$

  Where

  $$P(y_t|y_1, \dots, y_{t-1}, x) \tag{6.10}$$

  is the probability of next target word, given target words so far and source sentence x

- Encoder-decoder architecture: Encoder takes input and produces a neural representation; Decoder produces output based on that neural representation.

  - Seq2seq: both input and output are sequences.
  - Summarization: Long text à short text
  - Dialogue: previous utterances à next utterance
  - Parsing: Input text à output parse as a sequence
  - Code generation à Natural language à Python code

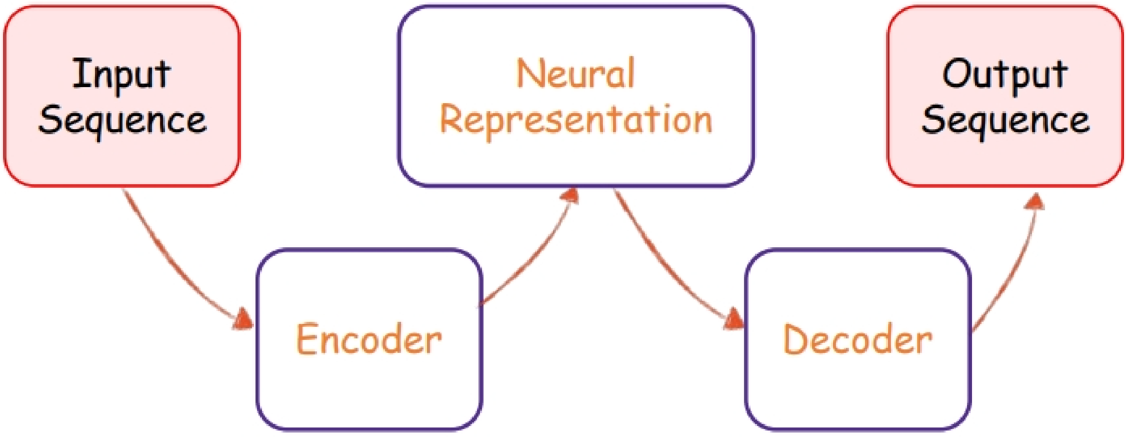


Figure 6.9: Sequence of Seq2seq

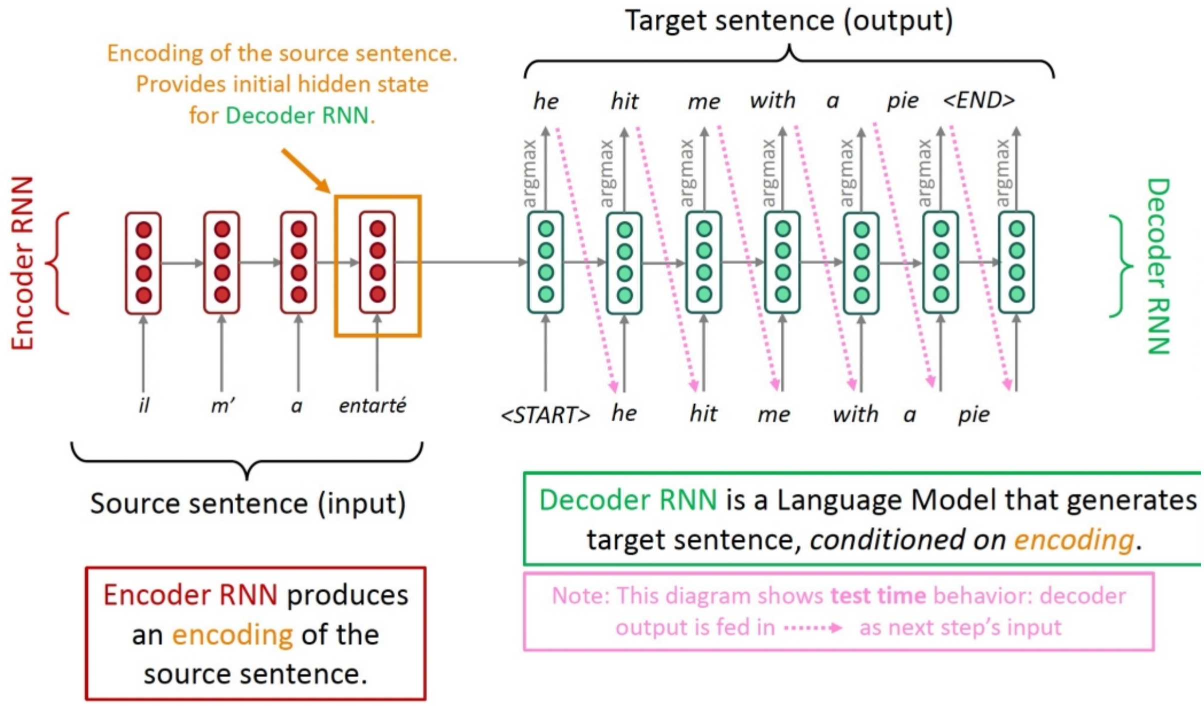

Target sentence (output)
Encoding of the source sentence. Provides initial hidden state for Decoder RNN.
he
hit
me
with
a
pie
<END>
argmax
Encoder RNN
Decoder RNN
il
m'
a
entarté
<START>
he
hit
me
with
a
pie
Source sentence (input)
Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.
Encoder RNN produces an encoding of the source sentence.
Note: This diagram shows **test time** behavior: decoder output is fed in ······▸ as next step's input

Figure 6.10: Seq2Seq Architecture

6.4.3 Applications of Seq2Seq

Machine Translation. One of the most prominent applications of Seq2Seq models is in machine translation, where the model translates text from one language to another. This involves encoding the source text into a context-sensitive representation, then decoding this representation into the target language.

Text Summarization. Seq2Seq models are also applied in automatic text summarization, which aims to generate a concise and coherent summary of a longer text document. The model learns to identify the most important parts of the source text and to produce a shorter version that retains the essential information.

Question Answering. In question answering systems, Seq2Seq models can be used to generate answers to questions based on a given context or knowledge base. The model processes the question as the input sequence and generates the answer as the output sequence.

Chatbots and Conversational Agents. Chatbots and conversational agents use Seq2Seq models to generate human-like responses in dialogues. These models can learn from large datasets of conversational exchanges and produce responses that are contextually appropriate to the input message.

Speech Recognition and Generation. Seq2Seq models are employed in speech recognition to convert speech audio into text and in speech generation to convert text into speech audio.

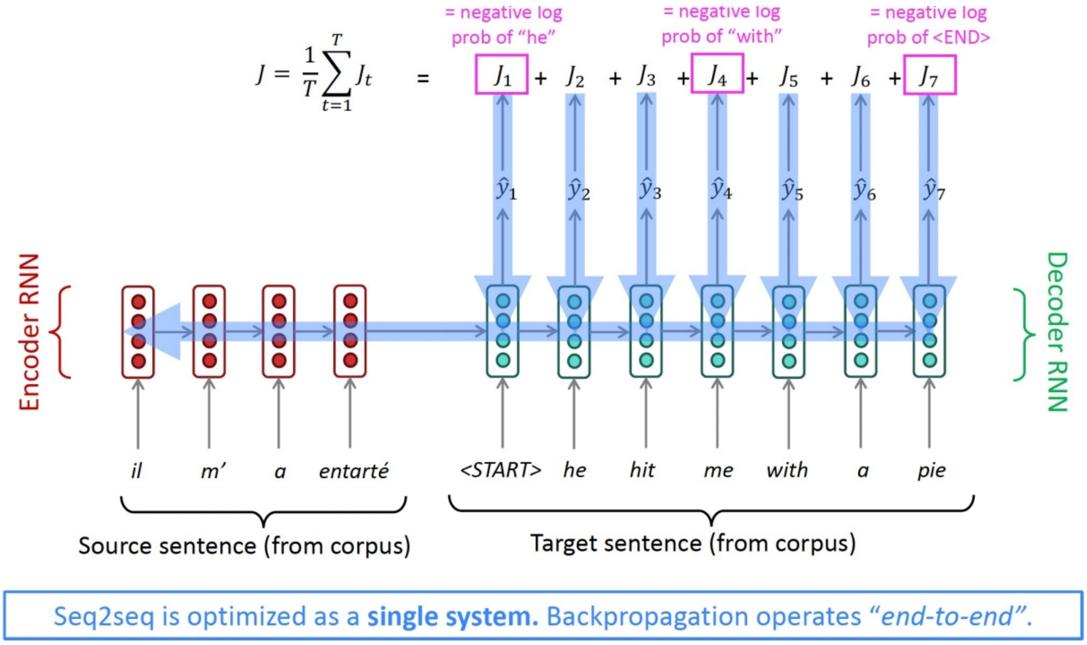
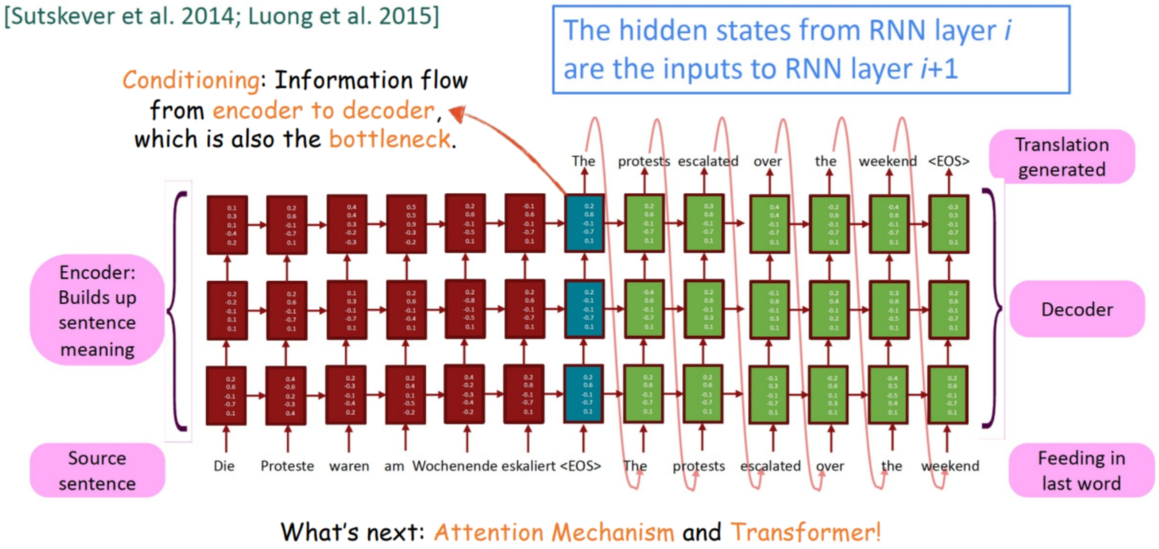

= negative log prob of "he"
= negative log prob of "with"
= negative log prob of <END>
$J = \frac{1}{T}\sum_{t=1}^{T} J_t = \boxed{J_1} + J_2 + J_3 + \boxed{J_4} + J_5 + J_6 + \boxed{J_7}$
$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$ $\hat{y}_5$ $\hat{y}_6$ $\hat{y}_7$
Encoder RNN
Decoder RNN
il m' a entarté
<START> he hit me with a pie
Source sentence (from corpus)
Target sentence (from corpus)
Seq2seq is optimized as a single system. Backpropagation operates "end-to-end".

Figure 6.11: Seq2Seq Training

This involves mapping sequences of audio signals to sequences of words (for recognition) and vice versa (for generation).

6.5 Demo

See https://colab.research.google.com/drive/1e4Wt4T7Dbril67ieZ5dUcdyN5ypCGt_S?usp=sharing



[Sutskever et al. 2014; Luong et al. 2015]

The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1

Conditioning: Information flow from encoder to decoder, which is also the bottleneck.

Encoder: Builds up sentence meaning

Decoder

Source sentence

Translation generated

Feeding in last word

What's next: Attention Mechanism and Transformer!

Figure 6.12: Multi-Layer Seq2Seq

7

Natural Language Processing (5)

——**Scribed by Qinghe GUI and Chaoyuan JIANG**

7.1 Outline

The outline of this document is as follows:

- In Section 7.2, we introduce the key issues with RNN.
- In Section 7.3, we talk about the attention mechanism in Seq2Seq.
- In Section 7.4, we show the self-attention operation.
- In Section 7.5, we explain how to consider word order in the model.
- In Section 7.6, we describe two implementations of Auto-Regression.
- In section 7.7, we present the most commonly used transformer block.

7.2 Key issues with RNN

One of the key issue of RNN is the information bottleneck (shown in Figure 7.1). When we are transferring information contained in the hidden state of the encoder sequence to the decoder sequence, this creates the so-called bottleneck.

The direct consequence is the linear interaction distance (Shown in Figure 7.2). For example, "chef" and "was" have coherent and close relationship from a semantic perspective although they are separated by a long clause here. RNN design cannot handle such issue very well.

Another key issue with RNN is the non-parallelizability (As in Figure 7.3).

- Forward and backward passes both have O(sequence length) unparallelizable operations.
- GPUs can perform independent small computations quickly in a large scale.
- Future hidden states cannot be computer (in full) before past RNN hidden states have been computer.
- Cannot scale with a very large dataset.

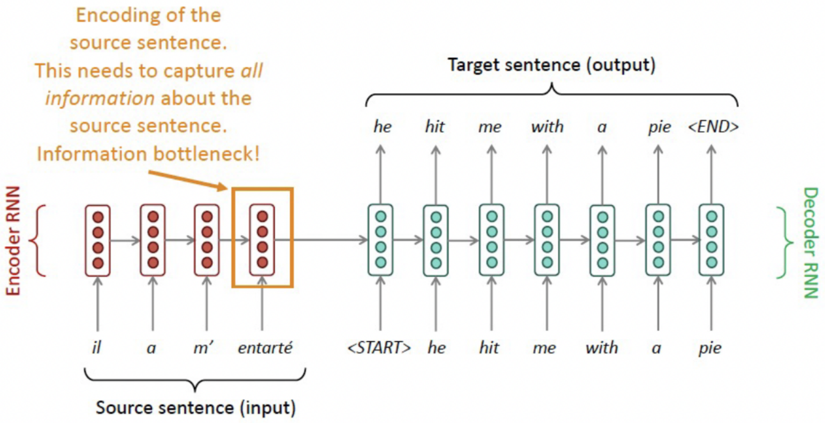

Encoding of the source sentence. This needs to capture *all information* about the source sentence. Information bottleneck!
Target sentence (output)
he hit me with a pie <END>
Encoder RNN
Decoder RNN
il a m' entarté
<START> he hit me with a pie
Source sentence (input)

Figure 7.1: information bottleneck in RNN

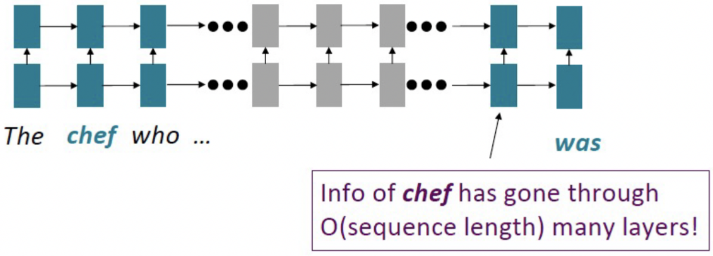

The chef who ...
was
Info of ***chef*** has gone through O(sequence length) many layers!

Figure 7.2: Linear Interaction Distance

7.3 Attention Mechanism in Seq2Seq

Idea of attention: On each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sentence.

The intuition behind the idea is that we will compute these single weight as an inner product associated with each of the hidden state or hidden vector in the encoder sequence. Then, use the inner product as a kind of weight to weigh different hidden vectors. After putting them together, it will generate a final attention output. Finally, using the attention output to concatenate together with the original hidden vector and put them together as an input factor (Figure 7.4).

Below are the mathematical equations (Figure 7.5):

Attention mechanism performs very well in NMT as shown in Figure 7.6. There are two main insights that we can observe from it.

- For those who weight attention always dominate the model without attention, suggesting that attention mechanism does help. It improves the performance of the Seq2Seq model.
- Size matters. The larger the model, trained on better high quality data, then the RNN sequence will be better.

Attention addresses RNN issues:

- Information Retrieval perspective: Attention treats each word's representation (i.e., hidden state) as a query to access and incorporate information from a set of values.
- Attention applied to a single sequence: Number of unparallelizable operations does not increase with sequence length. The maximum interaction distance is $O(1)$.

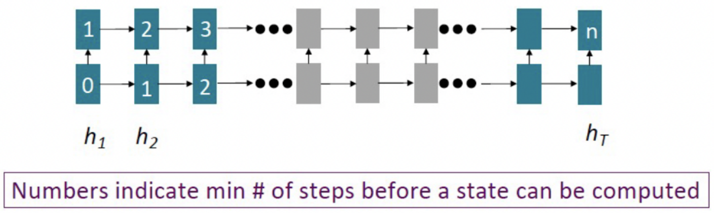

1
2
3
n
0
1
2
h_1
h_2
h_T
Numbers indicate min # of steps before a state can be computed

Figure 7.3: Non-parallelizability

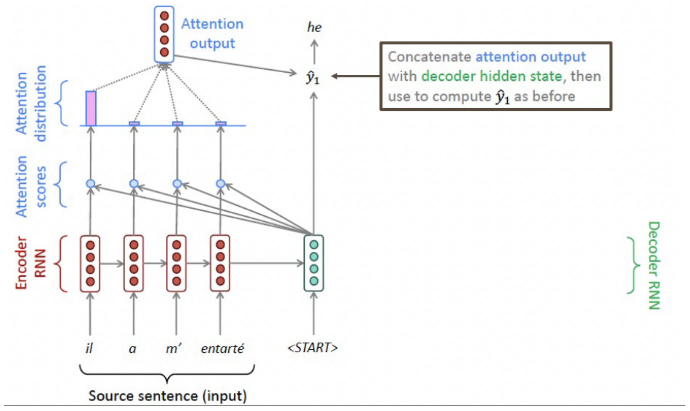
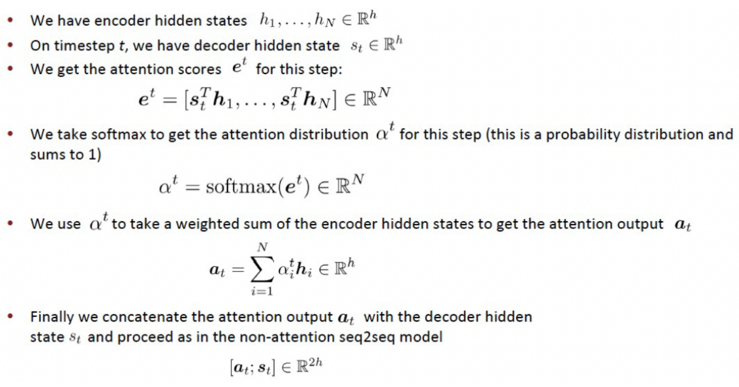


Figure 7.4: Attention Mechanism in Seq2Seq

Here, attention is also a very general Deep-Learning technique (Figure 7.7).

- Attention: Given a set of vector values and a vector of query, attention is a technique to compute a weighted sum of the values dependent on the query.
  - The weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on.
  - A fixed-size representation of an arbitrary set of representations (values), dependent on some other representation (query).
- In Seq2Seq + attention, each decoder hidden state (query) attends to all the encoder hidden states (values).

Figure 7.8 shows a family of attention models.



- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores e^t for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \operatorname{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Figure 7.5: Attention Mechanism:Equations

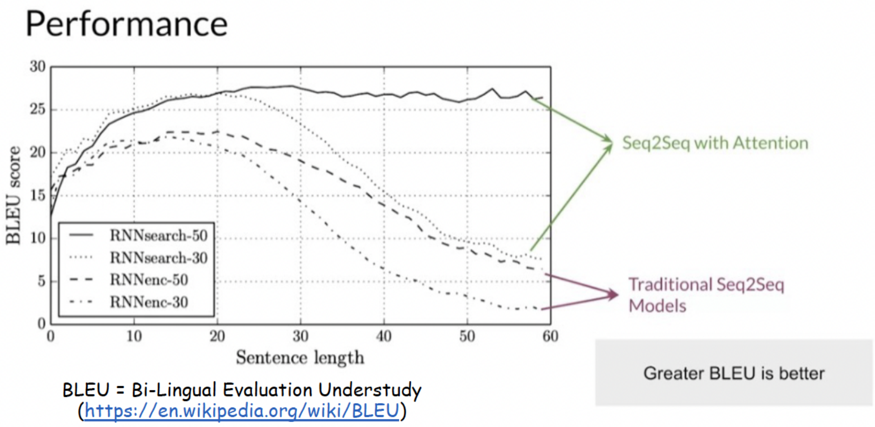


Figure 7.6: Performance

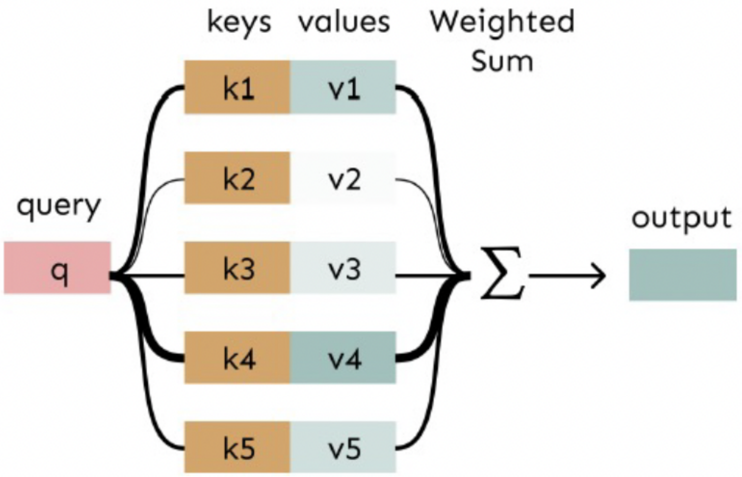


Figure 7.7: Attention

7.4 Self-attention

The fundamental operation of any transformer architecture is the self-attention operation.

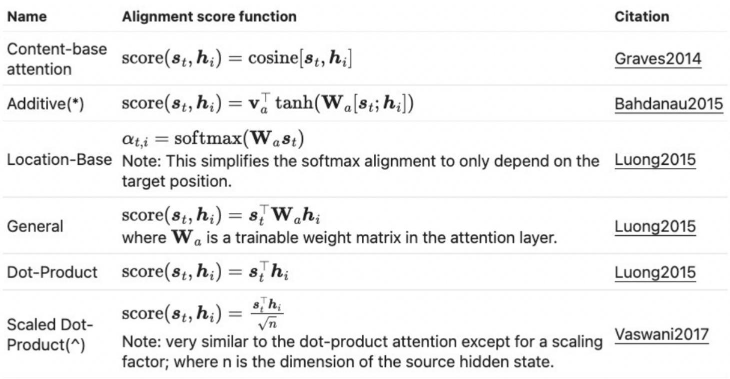
- Transformer: No RNN architecture, just attention mechanism.
- Self-attention: To generate y_t, we need to pay attention to $y_{<t}$.

Self-attention is a sequence-to-sequence operation: a sequence of vectors goes in, and a sequence of vectors comes out. Let's call the input vectors $x_1, x_2, \ldots, x_t$ and the corresponding output vectors $y_1, y_2, \ldots, y_t$. The vectors all have dimension k.

To produce output vector y_i, the self attention operation simply takes a weighted average over all the input vectors

Where j indexes over the whole sequence and the weights sum to one over all j. The weight w_{ij} is not a parameter, as in a normal neural net, but it is derived from a function over x_i and x_j. The simplest option for this function is the dot product:

The dot product gives us a value anywhere between negative and positive infinity, so we apply a softmax to map the values to $[0, 1]$ and to ensure that they sum to 1 over the whole sequence:



Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

Figure 7.8: Attention Models

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{x}_j \,.$$

And that's the basic operation of self attention. Figure 7.9 is a visual illustration of basic self-attention. Note that the softmax operation over the weights is not illustrated.

The actual self-attention used in modern transformers relies on three additional tricks:

- Queries, keys and values
- Scaling the dot product
- Multi-head attention

7.4.1 Queries, Keys and Values

Every input vector x_i is used in three different ways in the self attention operation:

- It is compared to every other vector to establish the weights for its own output y_i.
- It is compared to every other vector to establish the weights for the output of the j-th vector y_j.
- It is used as part of the weighted sum to compute each output vector once the weights have been established.

These roles are often called the **query**, the **key** and the **value**. In the basic self-attention we've seen so far, each input vector must play all three roles. We make its life a little easier by deriving new vectors for each role, by applying a linear transformation to the original input vector. In other words, we add three $k \times k$ weight matrices W_q, W_k, W_v and compute three linear transformations of each x_i, for the three different parts of the self attention:

This gives the self-attention layer some controllable parameters, and allows it to modify the incoming vectors to suit the three roles they must play.Figure 7.10 is the illustration of the self-attention with key, query and value transformations.

7.4.2 Scaling the Dot Product

The softmax function can be sensitive to very large input values. These kill the gradient, and slow down learning, or cause it to stop altogether. Since the average value of the dot product

$$w'_{ij} = \mathbf{x}_i{}^{\mathsf{T}}\mathbf{x}_j \,.$$

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}} \,.$$

grows with the embedding dimension k, it helps to scale the dot product back a little to stop the inputs to the softmax function from growing too large:

Why $\sqrt{k}$? Imagine a vector in $\mathbb{R}^k$ with values all c. Its Euclidean length is $\sqrt{k}c$. Therefore, we are dividing out the amount by which the increase in dimension increases the length of the average vectors.

7.4.3 Multi-head Attention

Multi-head attention is a way to speed up the training procedure. Instead of using a large matrix to compute all attentions, we can compute multiple attention matrices and concatenate the final vectors. It allows us for parallel computing: deploy attention mechanisms to multiple computing cores in parallel and sum them up at the end. The input is $dim = 256$, 8 attention heads, each with 32 dimensions.

Here is the whole process illustrated in Figure 7.11. It shows the basic idea of multi-head self-attention with 4 heads. To get our keys, queries and values, we project the input down to vector sequences of smaller dimension.

7.5 Using the Positions

It is not difficult to observe that in the aforementioned model, if we shuffle up the words in the sentence, we get the exact same classification, whatever weights we learn. Clearly, we want our state-of-the-art language model to have at least some sensitivity to word order, so this needs to be fixed.

The solution is simple: we create a second vector of equal length, that represents the position of the word in the current sentence, and add this to the word embedding. There are two options:

- Position Embeddings
- Position Encodings

The final input of the model is the sum of word embeddings and position embeddings. Figure 7.12 is an example.

7.5.1 Position Embeddings

We simply embed the positions like we did the words. Just like we created embedding vectors v_{cat} and v_{susan}, we create embedding vectors v_{12} and v_{25}. Up to however long we expect sequences to get. The drawback is that we have to see sequences of every length during training, otherwise the relevant position embeddings don't get trained. The benefit is that it works pretty well, and it's easy to implement.

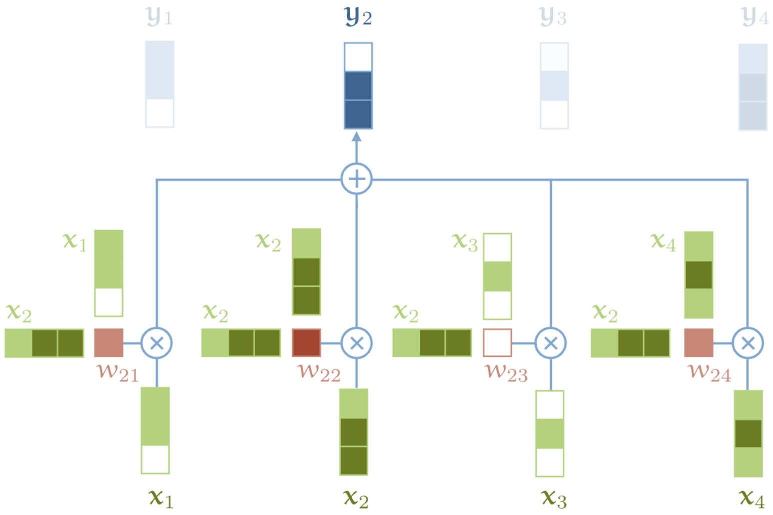


Figure 7.9: A Visual Illustration of Basic Self-attention

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i \qquad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i \qquad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

$$w'_{ij} = \mathbf{q}_i{}^\mathsf{T} \mathbf{k}_j$$
$$w_{ij} = \mathrm{softmax}(w'_{ij})$$
$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{v}_j \,.$$

7.5.2 Position Encodings

Position encodings work in the same way as embeddings, except that we don't learn the position vectors, we just choose some function $f : \mathbb{N} \to \mathbb{R}^k$ to map the positions to real valued vectors, and let the network figure out how to interpret these encodings. The benefit is that for a well chosen function, the network should be able to deal with sequences that are longer than those it's seen during training (it's unlikely to perform well on them, but at least we can check). The drawbacks are that the choice of encoding function is a complicated hyperparameter, and it complicates the implementation a little.

7.6 Auto-Regression

Auto-Regression is a self-supervised learning for transformers. To use self-attention in decoders, we need to mask the future. There are two commonly used methods:

- Inefficient Implementation: Change the set of keys and queries to include only past words. Figure 7.13 is an example.

- Parallelizable Implementation: Mask out attention to future words by setting the weight to $-\infty$. Figure 7.14 is an example.

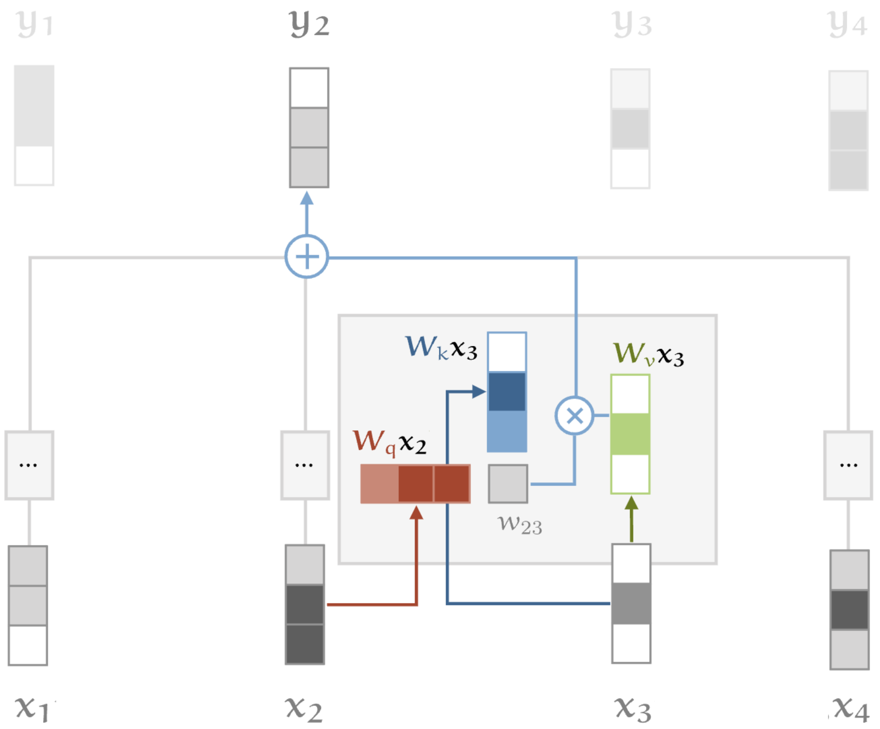


Figure 7.10: A Illustration of the Self-attention with Key, Query and Value Transformations

$$w'_{ij} = \frac{q_i^{\top} k_j}{\sqrt{k}}$$

7.7 The Transformer Block

7.7.1 Layer Normalization

Layer normalization is a trick to help models train faster. The idea is cutting down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation within each layer. LayerNorm's success may be due to its normalizing gradients. The procedures are described as follows:

- Let $x \in \mathbb{R}$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)

Then layer normalization computes:

7.7.2 Transformer

There are some variations on how to build a basic transformer block, but most of them are structured roughly like this:

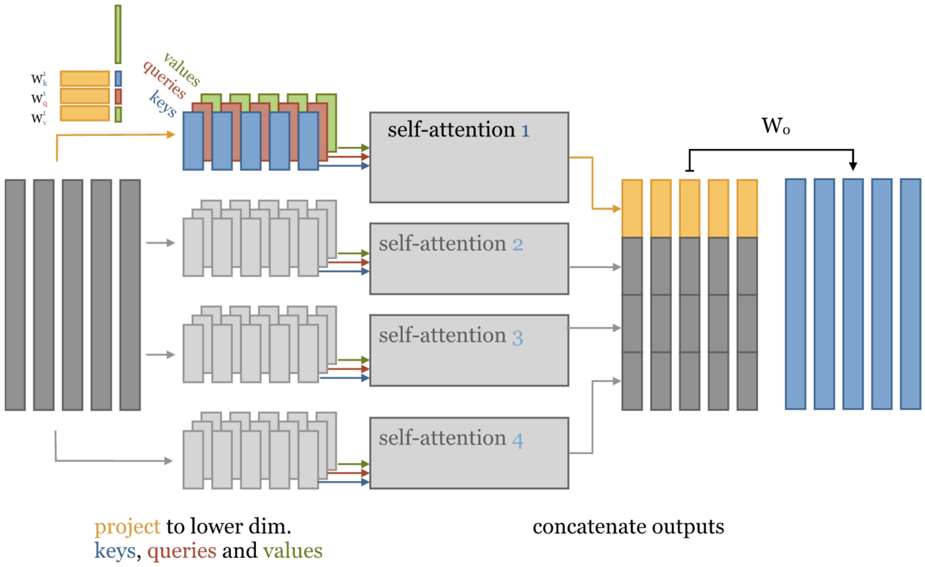


Figure 7.11: Multi-head Self-attention

That is, the block applies, in sequence: a self attention layer, layer normalization, a feed forward layer (a single MLP applied independently to each vector), and another layer normalization. Residual connections are added around both, before the normalization. The order of the various components is not set in stone; the important thing is to combine self-attention with a local feedforward, and to add normalization and residual connections.

7.7.3 Classification Transformer

The most common way to build a sequence classifier out of sequence-to-sequence layers, is to apply global average pooling to the final output sequence, and to map the result to a softmaxed class vector.

Figure 7.16 is an overview of a simple sequence classification transformer. The output sequence is averaged to produce a single vector representing the whole sequence. This vector is projected down to a vector with one element per class and softmaxed to produce probabilities.

7.7.4 The Transformer Encoder-Decoder

Recall that in machine translation, we processed the source sentence with a bidirectional model and generated the target with a unidirectional model. For this kind of seq2seq format, we often use a Transformer Encoder-Decoder. We use a normal Transformer Encoder.Our Transformer Decoder is modified to perform crossattention to the output of the Encoder. Figure 7.17 shows the model architecture of the Transformer Encoder-Decoder.

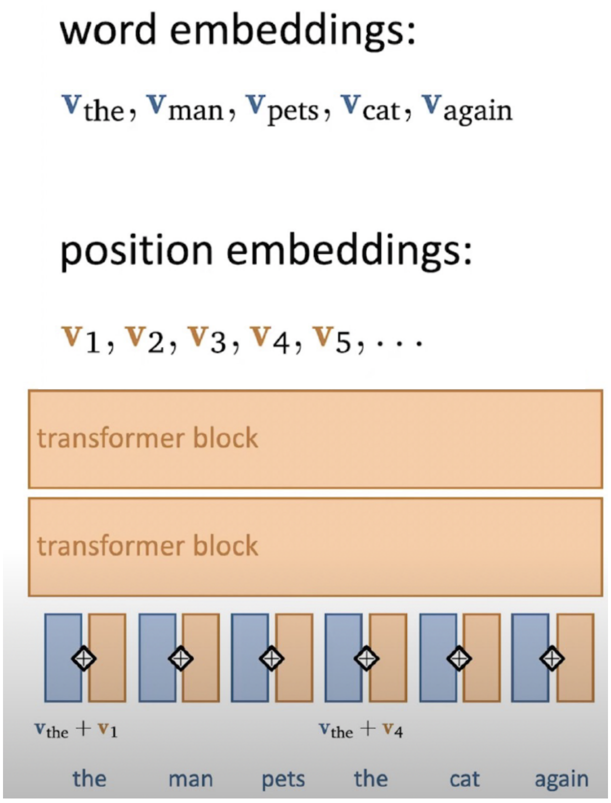

word embeddings:
$\mathbf{v}_{\text{the}}, \mathbf{v}_{\text{man}}, \mathbf{v}_{\text{pets}}, \mathbf{v}_{\text{cat}}, \mathbf{v}_{\text{again}}$
position embeddings:
$\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \ldots$
transformer block
transformer block
$\mathbf{v}_{\text{the}} + \mathbf{v}_1$
$\mathbf{v}_{\text{the}} + \mathbf{v}_4$
the
man
pets
the
cat
again

Figure 7.12: Position Encoding

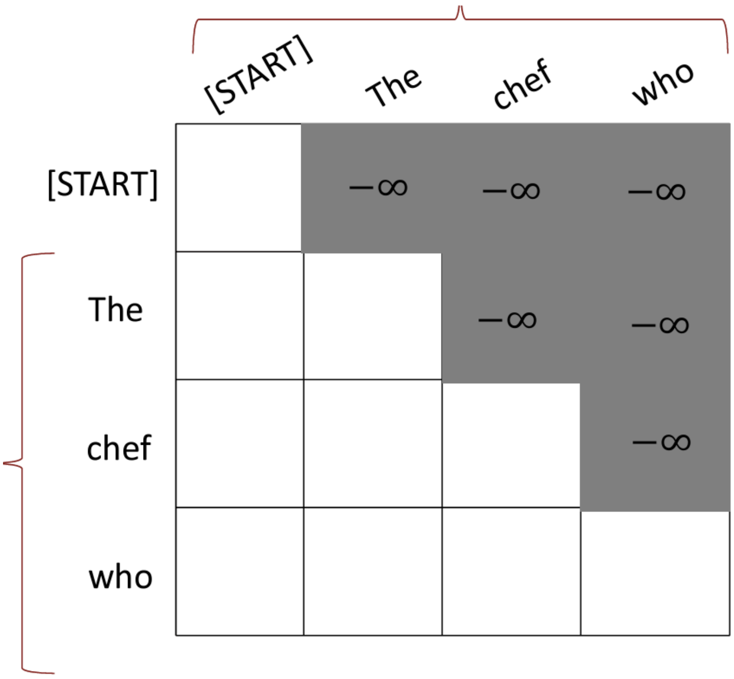


Figure 7.13: Inefficient Implementation

$$w'_{ij} = \begin{cases} q_i^T k_j, j \le i \\ -\infty, j > i \end{cases}$$

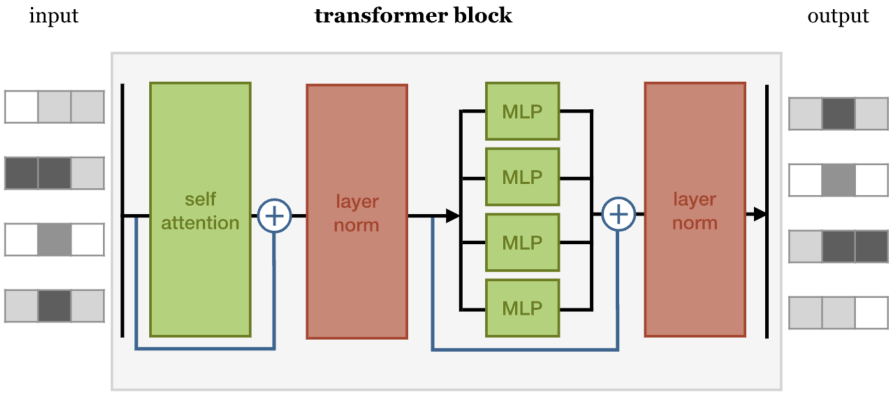
Figure 7.14: Parallelizable Implementation

$$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} * \gamma + \beta$$

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

Figure 7.15: Layer Normalization



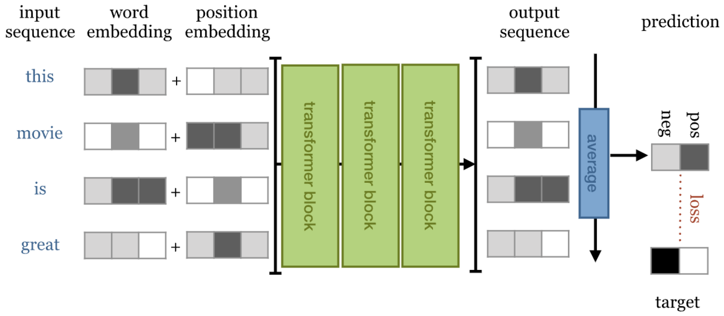


Figure 7.16: Classification Transformer

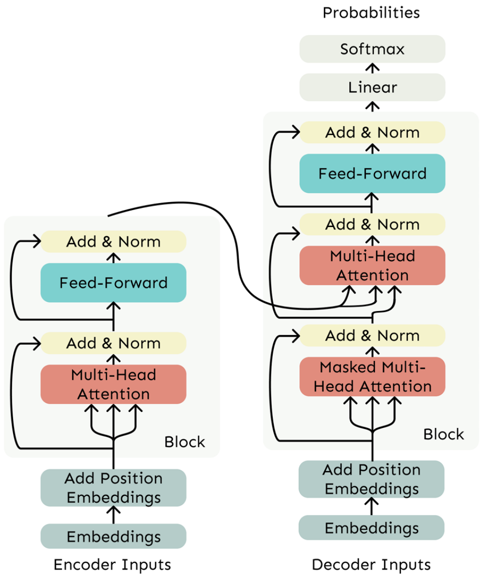


Figure 7.17: The Transformer Encoder-Decoder

8

Natural Language Processing (6)

——Scribed by Jia LIU

8.1 Outline

The purpose of this document is to give a crash course on Pretraining in Natural Language Processing. The focus will be on pre-training and two language model built on transformer architectures, BERT and GPT. Specifically, the outline of this document is as follows:

- In Section 8.2, we introduce pre-training, overall framework and three pre-training architectures.
- In Section 8.3, we introduce the BERT model and FinBERT model.
- In Section 8.4, we discuss the GPT models.

8.2 Pre-training

8.2.1 What is pre-training?

The idea of pre-training is to compress vast amounts of information from the entire internet to the foundational model before fine-tuning it on a specific task. The core of pre-training is compression.

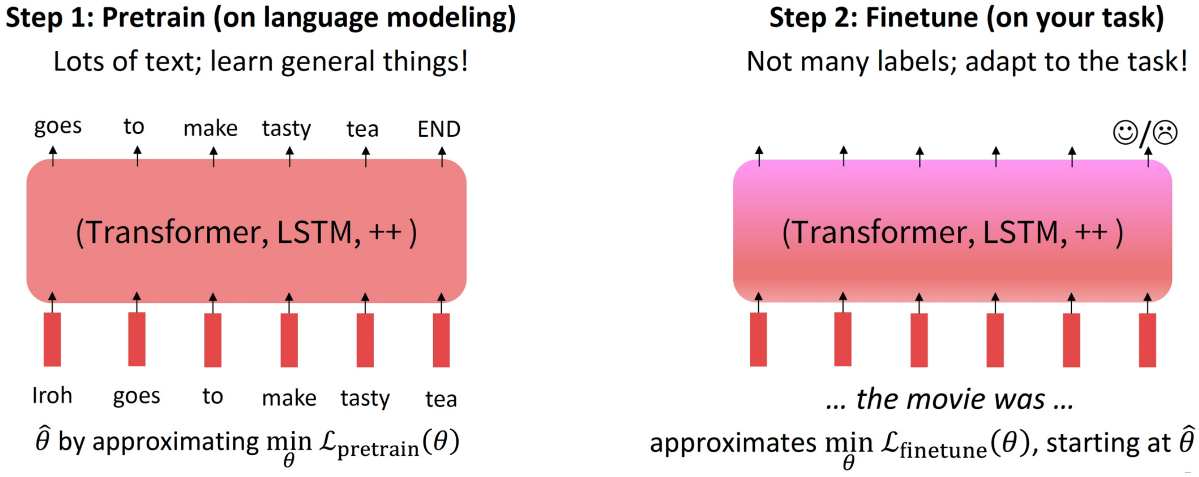
8.2.2 Insights from pre-training

- The model can process large-scale, diverse data sets.
- Never use labelled data; otherwise, you cannot scale.
- Compute-aware scaling → Scaling Laws, Kaplan et al. (2020)
  1. Larger models require fewer samples to reach the same performance.
  2. The optimal model size grows smoothly with the loss target and compute budget.

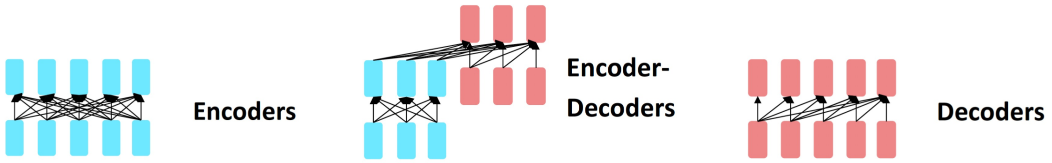
8.2.3 Overall framework

Pre-training, in a lot of cases, is based on some very simple tasks. In BERT, it is to predict the mass word in the middle of a context. In GPT and other large language models, it's about predicting next word. Basically, pre-trainging trains the very large-scale language model and obtain the value of $\hat{\theta}$.

The subsequent step is fine-tuning, which addresses some specific tasks that have downstream NLP applications like sentiment analysis, question answering, information extraction, among others. Importantly, at this stage, high-quality labeled data is essential. This requirement stems from the need to go beyond merely compressing information. With the $\hat{\theta}$ initialized in pre-training, fine-tuning is some kind of loss function that move the $\hat{\theta}$ a little bit to fine tune the model to make the subsequent task much better.


Step 1: Pretrain (on language modeling)
Lots of text; learn general things!
goes
to
make
tasty
tea
END
(Transformer, LSTM, ++)
Iroh
goes
to
make
tasty
tea
$\hat{\theta}$ by approximating $\min_{\theta} \mathcal{L}_{\text{pretrain}}(\theta)$
Step 2: Finetune (on your task)
Not many labels; adapt to the task!
☺/☹
(Transformer, LSTM, ++)
… the movie was …
approximates $\min_{\theta} \mathcal{L}_{\text{finetune}}(\theta)$, starting at $\hat{\theta}$

8.2.4 Three pre-training architectures



(a) Encoders (b) Encoders-Decoders (c) Decoders

Figure 8.1: Three pre-training architectures

a) Encoders
Condition on future information.
Example: BERT.

b) Encoder-Decoders
Combine encoder and decoder.
Example: T5.

c) Decoders
Cannot condition on future information.
Example: GPT and all large language models, because decoders approximate human intelligence and how human thinks and how human behaves.

8.3 BERT

8.3.1 What is BERT?

BERT (Devlin et al. (2018)), Bidirectional Encoder Representations from Transformers, is a language model built on the transformer architecture, but it only uses the encoder stack.
The key idea is to learn representations based on **bidirectional context**.
A famous example of BERT is, "We went to the river bank. vs. I need to go to the bank to make a deposit". It shows that both sides of the information matter.

8.3.2 Pre-training BERT

BERT's pre-training involves two main tasks: masked language modeling and next sentence prediction.
Masked language modeling: Randomly masks 15% of tokens in a sentence, where 80% of them are replaced with [MASK], 10% of them replaced with a random token and 10% of them not changed.
For example, in the example below, a sentence like "I went to the store", you might mask the words "store" with [MASK], replace the "went" with "pizza" to have "I pizza to the [MASK]".
→ Why not all masked tokens replaced with [MASK]? This is because in the follow-up procedure, fine-tuning, mask is something cannot be observed. [MASK] tokens are never seen in fine-tuning.
→ Why replaced with random tokens? To introduce noise into a training procedure to make the training more generalized.

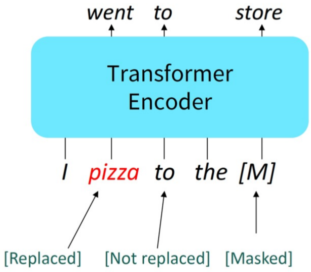
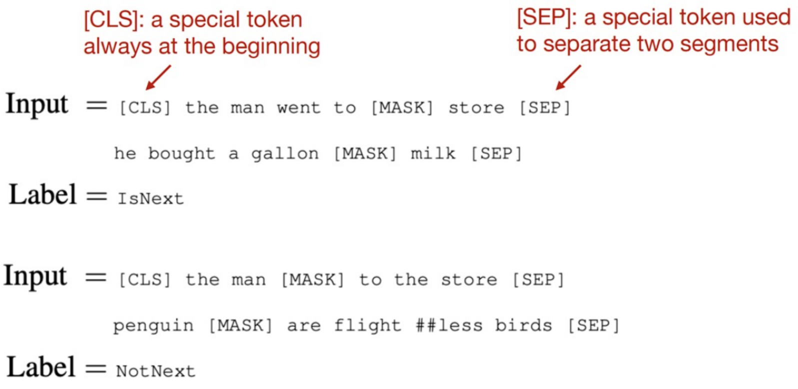


Figure 8.2: Masked process

Next sentence prediction: This helps BERT understand the relationships between two sentences, which is essential for tasks that involve understanding the context beyond individual sentences, such as question answering, natural language inference, and document summarization.

Below is how next sentence prediction works. During BERT's pre-training, inputs are formed by pairing two segments, A and B. These segments are sentences or parts of sentences from the training corpus. For next sentence prediction, 50% of the time, segment B is the actual next sentence that follows segment A in the original document (labeled as IsNext). The other 50% of the time, segment B is a random sentence from the corpus, which does not follow segment A (labeled as NotNext). BERT is trained to predict whether segment B is the true subsequent sentence following segment A. This binary classification task forces the model to learn a representation that encodes the relationship between pairs of sentences, enhancing its ability to handle tasks that require such understanding.

Subwords and Input Embeddings: To maintain consistency, BERT ensures that the



[CLS]: a special token always at the beginning
[SEP]: a special token used to separate two segments

```
Input  = [CLS] the man went to [MASK] store [SEP]
         he bought a gallon [MASK] milk [SEP]
Label  = IsNext

Input  = [CLS] the man [MASK] to the store [SEP]
         penguin [MASK] are flight ##less birds [SEP]
Label  = NotNext
```

Figure 8.3: Next sentence prediction

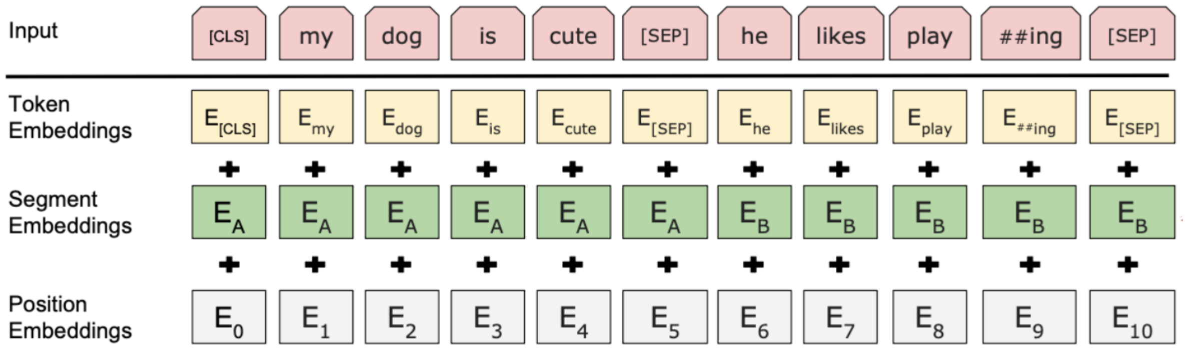
vocabulary used during training and testing remains consistent. Uncommon words are broken down into subwords or subunits, enabling the model to deal with words not seen during training. Here are some subword mapping examples:

- Common Words: Retained as whole (e.g., ’hat’, ’learn’).
- Variations: Broken down into recognizable pieces (e.g., ’taaaasty’ to ’taa##’ ’aaa##’ ’sty’).
- Misspellings: Decomposed to nearest subwords (e.g., ’laern’ to ’la##’ ’ern##’).
- Novel Terms: New or complex words are split into known subunits (e.g., ’Transformerify’ to ’Transformer##’ ’ify’).

This process allows BERT to understand and generate meaningful representations for words that are not explicitly present in its training data by combining the embeddings of known subunits.
For input embeddings, BERT’s input representation combines three types of embeddings to understand and process text:

- Token Embeddings: Capture the semantic meaning of individual tokens.
- Segment Embeddings: Differentiate between pairs of sentences (segment A and B) within one input stream.
- Position Embeddings: Encode the position of each token in the sequence, preserving word order information.

Below is an example of tokenization and embedding process. A sentence such as "my dog is cute" is tokenized into subwords, each token is then mapped to a token embedding. Next, a segment embedding is added to differentiate sentences in a pair (e.g., all tokens from one sentence receive the same segment embedding, ’E_A’ or ’E_B’). Last, position embeddings corresponding to each token’s position in the sequence are added (e.g., ’E_0’ for the first position).



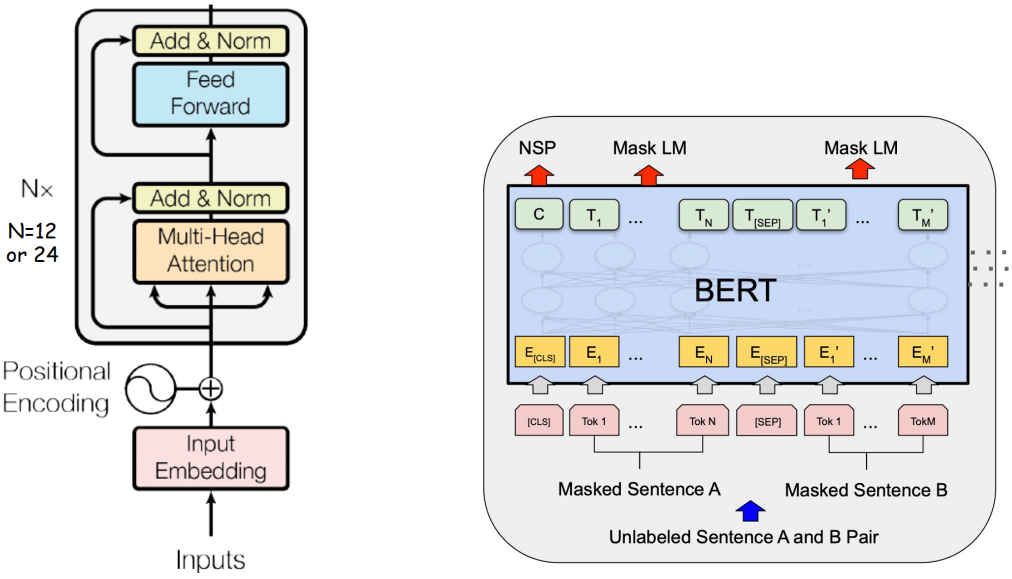
These input embeddings are crucial during the pre-training phase, allowing BERT to develop a nuanced understanding of language structure, context, and nuances from the corpus it's trained on, which includes books and Wikipedia entries.

8.3.3 BERT Pre-training Overall

BERT is designed in two sizes: BERT-base, which consists of 12 layers (transformer blocks), a hidden size of 768, 12 attention heads, and a total of 110 million parameters; and BERT-large, which is a larger variant with 24 layers, a hidden size of 1024, 16 attention heads, and a whopping 340 million parameters.

Pre-trained on an extensive corpus of text from Wikipedia, which contributes 2.5 billion words, and BookCorpus with an additional 800 million words, BERT captures a wide spectrum of language nuances and contexts. It is trained to process sequences up to 512 word pieces long, which is achieved by splitting the input text into smaller segments that can include up to 256 contiguous and 256 non-contiguous word pieces. This structure allows for the handling of longer contexts and nuanced language constructs.

BERT's training is a compute-intensive task, running for 1 million steps with a batch size of 128,000. This monumental training effort is facilitated by the use of 64 Tensor Processing Units (TPUs) over the course of 4 days, demonstrating the significant resources required to achieve its state-of-the-art performance.



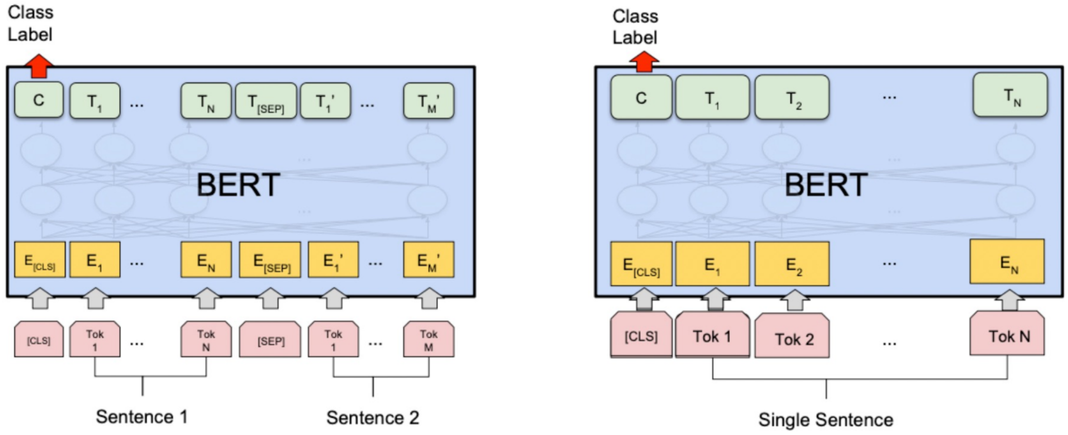
(a) BERT Model

(b) Pre-training Phase

During its pre-training phase, BERT is simultaneously trained on two tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). The model learns to predict words that have been intentionally obscured in a given sequence, which hones its ability to understand context within a sentence. Additionally, by evaluating if one sentence logically follows another, BERT acquires the capacity to grasp the relationship between separate segments of text. This dual aspect of pre-training, with the [CLS] token specifically pre-trained for NSP and the other tokens for MLM, equips BERT with a nuanced bidirectional contextual understanding. This foundational training underpins BERT's exemplary performance in downstream NLP tasks, including sentiment analysis, question answering, and language inference, where it surpasses the capabilities of previous models.

8.3.4 Fine-tuning BERT

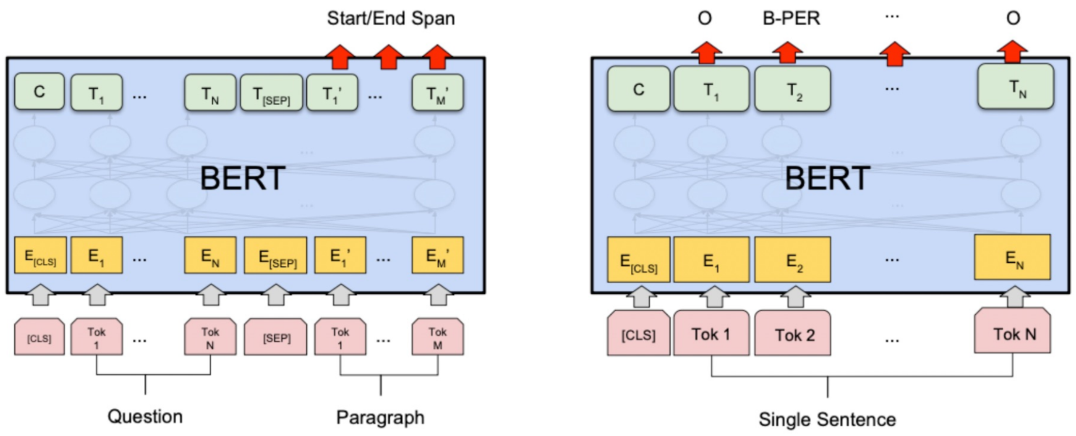
The idea of BERT fine-tuning is to use some refined downstream tasks to make the model stronger by adjust the pre-trained parameters.
After pre-training, BERT is fine-tuned using datasets specific to a particular NLP task. Examples include sentence pair classification tasks like MNLI or QQP, single-sentence tasks like SST-2, and token-level tasks like SQuAD or CoNLL 2003 NER.
For token-level tasks such as question answering or named entity recognition (NER), BERT's output for each token is used to predict the correct label, such as the answer span in SQuAD or the entity type in NER.



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

For sentence-level tasks, input sentences are fed into BERT, which provides a contextualized representation of each token. The [CLS] token's final hidden state is used as the aggregate sequence representation for classification tasks.



(c) Question Answering Tasks:
SQuAD v1.1

(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

Fine-tuning involves a relatively small number of training epochs over the task-specific data, adjusting the weights of all pre-trained layers and training the new output layer. The fine-tuning step is significantly faster than pre-training due to the smaller dataset size and the already learned language representations.

After fine-tuning, BERT has achieved state-of-the-art results on various NLP benchmarks, surpassing previous models and setting new performance standards for tasks such as MNLI, QQP, and SQuAD.

8.3.5 FinBERT

FinBERT (Liu et al. (2021)) is a specialized adaptation of the BERT-base model, pre-trained on financial datasets to better understand and interpret the language used in the financial domain. The model is pre-trained on a corpus comprising 4.9 billion tokens from various financial texts:

- Corporate annual and quarterly filings from the SEC's EDGAR website (1994-2019).
- Financial analyst reports from the Thomson Intext database (2003-2012).
- Earnings conference call transcripts from the Seeking Alpha website (2004-2019).

The fine-tuning phase involved sentiment analysis on a dataset of 10,000 sentences, which were labeled as: 36% positive, 46% neutral and 18% negative.

FinBERT is a testament to the effectiveness of fine-tuning pre-trained models on domain-specific datasets to enhance performance on specialized tasks. Its success in financial sentiment analysis demonstrates the potential for similar adaptations of BERT in other specialized fields.

8.4 GPT

8.4.1 What is GPT?

GPT, the Generative Pretrained Transformer, is a series of language models designed to perform a wide range of language understanding and generation tasks. GPT models are known for their autoregressive nature, predicting the probability of a sequence of words one after another.

8.4.2 Pre-training Decoders

The key idea behind GPT is to pre-train decoders to serve as language models. This involves using autoregression to predict the next word in a sentence based on the words that come before

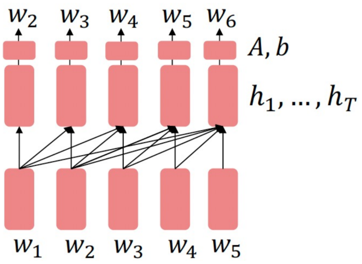
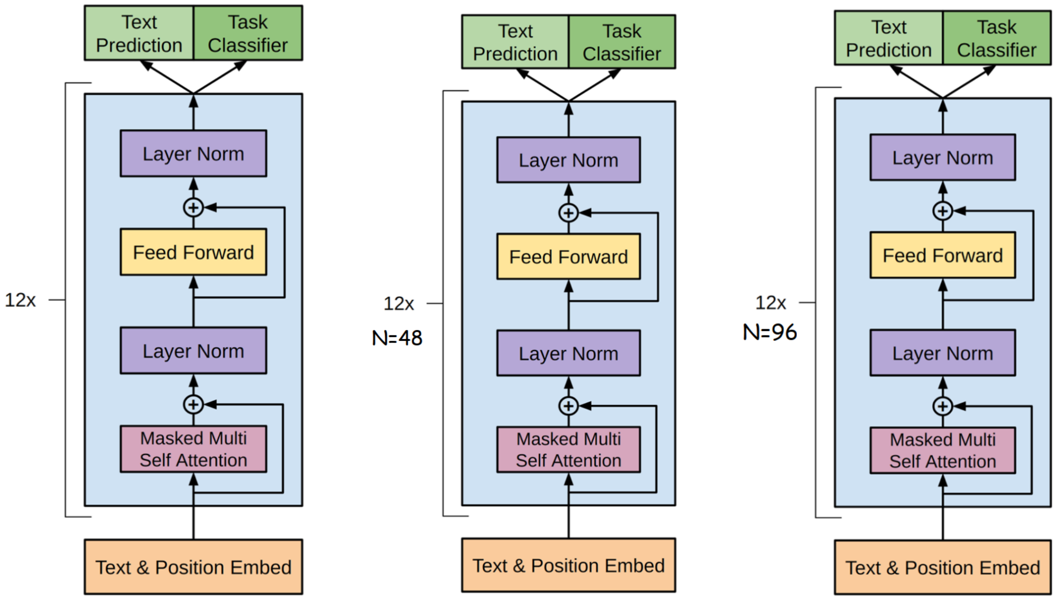


Figure 8.4: Pre-training decoders

it. Pre-training decoders is more challenging than the masked language model approach used by BERT (Radford et al. (2018)).



(a) GPT-1 Model (b) GPT-2 Model (c) GPT-3 Model

Figure 8.5: GPT models

8.4.3 GPT-1

Introduced as an architecture with only masked self-attention, GPT-1 includes 12 layers of transformer decoders. It features 768-dimensional hidden states and 117 million parameters. It was trained on the BooksCorpus dataset, which contains over 7,000 unique books.

8.4.4 GPT-2

GPT-2 scaled up the architecture significantly, with 48 transformer blocks, 1,600 hidden units per layer, and 25 attention heads. With a total of 1.5 billion parameters, GPT-2 represents an order of magnitude larger model compared to its predecessor. It was trained on a diverse and high-quality dataset of 8 million web pages.

8.4.5 GPT-3

GPT-3 marks a massive leap in the series with 175 billion parameters, incorporating 96 decoder blocks. Its context size and embedding dimensions are vastly increased to handle broader contexts and more complex tasks. Trained on an even larger dataset of 300 billion tokens, GPT-3 demonstrated significant advancements in few-shot learning, performing tasks with minimal examples given (Brown et al. (2020)).

8.4.6 Fine-tuning GPT Models

GPT models are fine-tuned for specific tasks by adding a task-specific layer on top of the pre-trained model. Fine-tuning involves adjusting the model on a task-specific dataset with a combination of text prediction and classification objectives. The loss function during fine-tuning combines the loss of text prediction and classification to optimize the model for the desired task.

9

Natural Language Processing (7)

———Scribed by Letian KONG and Liheng TAN

9.1 Outline

The purpose of this document is to give a crash course on Large Language Models (LLMs). Specifically, the outline of this document is as follows:

- In section 9.2, we will introduce scaling laws for language models.
- In section 9.3, we will introduce the large language models and the methods to finetuning the models.
- In section 9.4, we will introduce the applications of LLMs in the business and economics.

9.2 Scaling Laws for Language Models

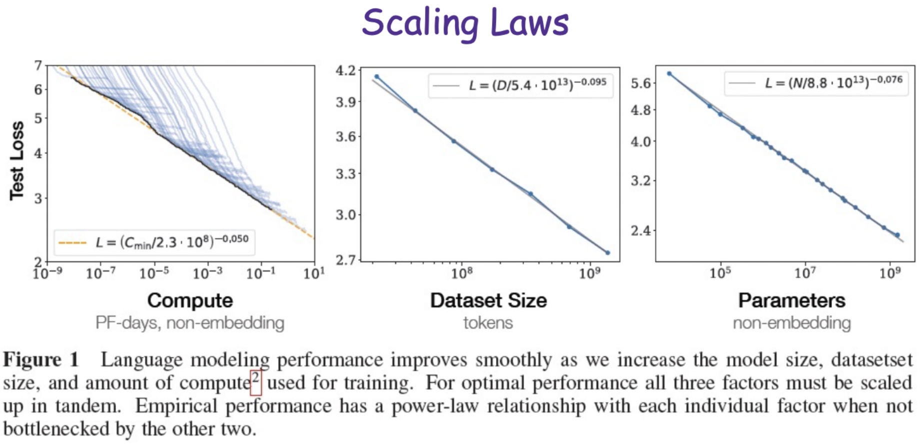


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Figure 9.1: Scaling Laws

The scaling laws for neural language models have become an important factor in their development. The performance of these models, as measured by the test loss, improves predictably with the increase in model size, dataset size, and computational power used for training. The empirical performance of these models shows a power-law relationship with each of these factors. However, it is essential to scale all three factors concurrently to avoid bottlenecks.

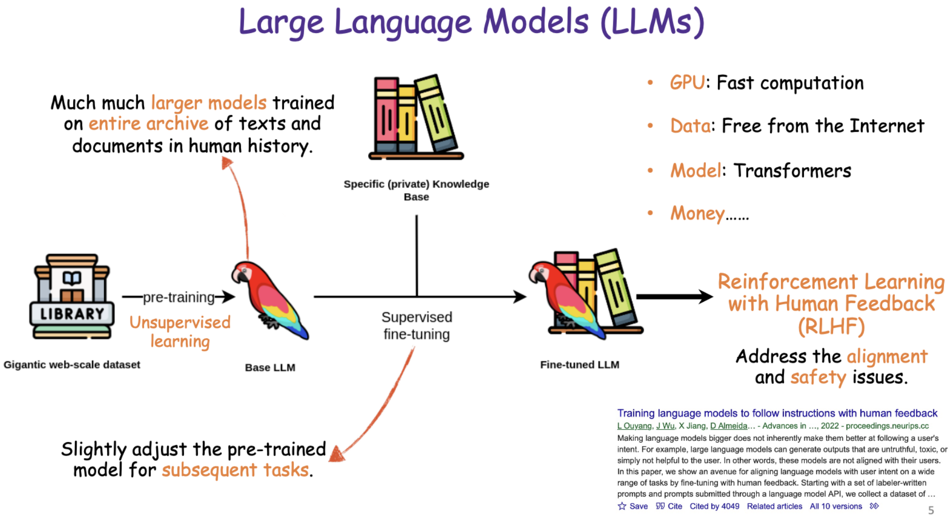
- **Compute:** The test loss decreases with more compute (measured in petaflop/s-days), following a power-law. Compute is a critical factor in training more complex models.

- **Dataset Size:** As the size of the dataset increases, the model's ability to predict and generalize improves, which is evident from the reduction in test loss.

- **Parameters:** The number of parameters in a model is directly proportional to its capacity. The scaling law shows that as we increase the parameters, the test loss decreases, indicating better performance.

The slide shows three graphs, each representing one of the scaling factors mentioned above. These graphs help in understanding how each factor independently affects the performance of language models. It is crucial to note that the improvement in performance is subject to the law of diminishing returns, which means that the amount of gain decreases with the increase in size, data, or compute.

9.3 Large Language Models (LLMs)



Large Language Models (LLMs)

Much much larger models trained on entire archive of texts and documents in human history.

- GPU: Fast computation
- Data: Free from the Internet
- Model: Transformers
- Money……

Reinforcement Learning with Human Feedback (RLHF)

Address the alignment and safety issues.

Slightly adjust the pre-trained model for subsequent tasks.

Training language models to follow instructions with human feedback

L Ouyang, J Wu, X Jiang, D Almeida… - Advances in …, 2022 - proceedings.neurips.cc

Making language models bigger does not inherently make them better at following a user's intent. For example, large language models can generate outputs that are untruthful, toxic, or simply not helpful to the user. In other words, these models are not aligned with their users. In this paper, we show an avenue for aligning language models with user intent on a wide range of tasks by fine-tuning with human feedback. Starting with a set of labeler-written prompts and prompts submitted through a language model API, we collect a dataset of …

Figure 9.2: Large Language Models

Large Language Models (LLMs) represent a significant advancement in the field of natural language processing. These models are trained on vast corpuses encompassing a wide span of human knowledge and are characterized by their large number of parameters. The process of developing these models involves several stages:

- **Pre-training:** This unsupervised learning phase uses a gigantic web-scale dataset, allowing the model to learn from the entirety of text and documents available in human history.

- **Supervised Fine-tuning:** After pre-training, the model is fine-tuned on a more specific dataset, often with a narrower scope, to hone in on particular tasks or types of knowledge.

- **Reinforcement Learning with Human Feedback (RLHF):** This step involves human trainers who provide feedback to the model, enabling it to better align with human intentions and address safety issues.

LLMs rely on a number of key resources and technologies to function:

1. **GPUs:** They provide the fast computation necessary to process the massive datasets involved.

2. **Data:** The training data, often freely available from the internet, provides the foundation for the model's knowledge.

3. **Model Architecture:** Transformers are the backbone of LLMs, enabling them to handle complex patterns in data.

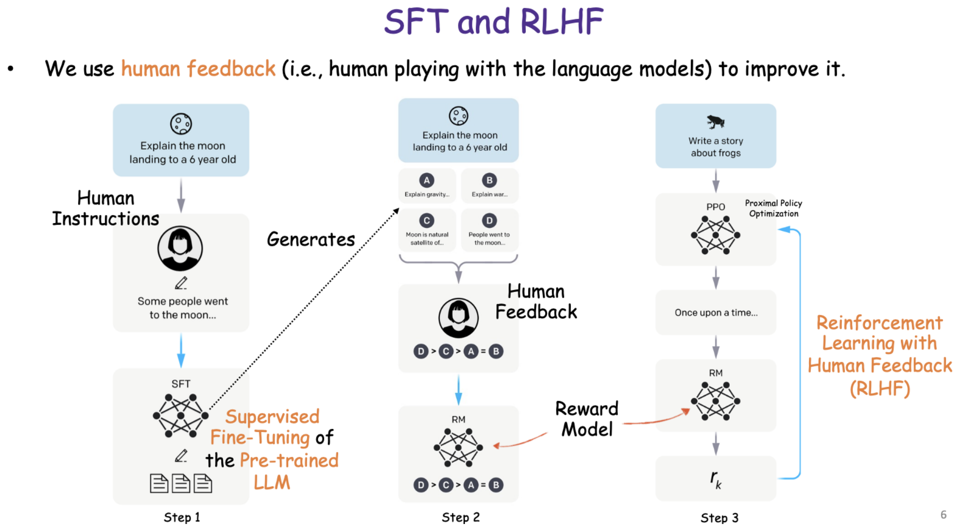
4. **Funding:** Sufficient financial investment is crucial for access to computational resources and for supporting the development process.

By adjusting the pre-trained models with these methods, LLMs can be adapted to perform a variety of tasks, demonstrating their versatility and power in understanding and generating human language.

9.3.1 Alignment and Safety in LLMs

An essential aspect of LLMs is the alignment with user intent and safety. Reinforcement Learning with Human Feedback (RLHF) is a technique used to fine-tune the model's responses to be more aligned with ethical guidelines and user expectations. This helps in mitigating risks associated with the model generating unsafe or biased content.

9.3.2 Supervised Fine-Tuning (SFT) and Reinforcement Learning with Human Feedback (RLHF)



SFT and RLHF

- We use human feedback (i.e., human playing with the language models) to improve it.

Explain the moon landing to a 6 year old
Human Instructions
Some people went to the moon...
SFT
Supervised Fine-Tuning of the Pre-trained LLM
Step 1
Generates
Explain the moon landing to a 6 year old
A
Explain gravity...
B
Explain war...
C
Moon is natural satellite of...
D
People went to the moon...
Human Feedback
D > C > A = B
RM
D > C > A = B
Step 2
Reward Model
Write a story about frogs
PPO
Proximal Policy Optimization
Once upon a time...
RM
r_k
Reinforcement Learning with Human Feedback (RLHF)
Step 3
6

Figure 9.3: SFT and RLHF

In the quest to improve the performance and alignment of LLMs with human values, we utilize two main strategies: Supervised Fine-Tuning (SFT) and Reinforcement Learning with Human Feedback (RLHF).

Step 1: Supervised Fine-Tuning (SFT) Initially, the pre-trained LLM receives human instructions, which can be as diverse and creative as explaining complex events in simple terms or invoking the generation of stories. For example, one might instruct the model to "Explain the moon landing to a 6-year-old" or "Write a story about frogs." This step generates initial outputs that serve as a basis for further refinement.

Step 2: Incorporating Human Feedback The outputs generated from the human instructions are then reviewed by humans, who provide feedback. This feedback is not just a binary good or bad, but rather a nuanced ranking or scoring of the outputs, taking into account various factors such as coherence, relevance, and adherence to the task.

Step 3: Reinforcement Learning with Human Feedback (RLHF) Utilizing the feedback from the previous step, the model undergoes reinforcement learning. This often involves algorithms like Proximal Policy Optimization (PPO), which help in fine-tuning the policy of the model towards generating better outputs. The model learns from the reward signals based on the human feedback (rankings or scores) to produce responses that are more closely aligned with what is desirable or expected by humans.

Overall, this three-step process ensures that the LLM not only generates text based on its vast pre-trained knowledge but also adapts to produce content that fits with human expectations, demonstrating an understanding of context, subtlety, and user intent.

9.3.3 Multi-Task Instruction Finetuning

Multi-Task Instruction Finetuning is an approach to train language models to handle a variety of tasks through different types of finetuning methods. This includes:

Instruction Finetuning This method involves direct instruction to the language model to perform specific tasks. For example, asking "What is the boiling point of Nitrogen?" The model is expected to provide a straightforward answer based on its pre-trained knowledge, such as "-320.4F".

Chain-of-Thought Finetuning This finetuning method requires the language model to demonstrate its reasoning process. An example task could be "The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more, how many apples do they have?" The model should then provide a step-by-step explanation leading to the answer: "The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9."

Inference: Generalization to Unseen Tasks The ultimate goal of Multi-Task Instruction Finetuning is to enable the language model to generalize its capabilities to tasks it has not seen before. For instance, asking a hypothetical question such as "Can Geoffrey Hinton have a conversation with George Washington?" requires the model to understand historical context and utilize its reasoning to explain why this conversation is not possible due to the time period each individual lived in.

The notes on this slide are based on Lecture 10 of Stanford's CS224N course, available at the following URL:

`https://web.stanford.edu/class/cs224n/slides/cs224n-2024-lecture10-instruction-tuning-rl`
`pdf`.

9.3.4 AI Misalignment

Misalignment in artificial intelligence occurs when an AI system's actions do not correspond with human intentions or expectations. The comic strip on the slide humorously exemplifies misalignment where a character wishes to be 6'5" tall, and the genie, acting as the AI, literally stretches the character's body vertically, misunderstanding the intent behind the wish.

Example 1: Literal Interpretations in Language Processing Consider a language model trained to assist with cooking recipes. When asked to "peel an orange," an AI perfectly aligned with human intent would understand it as removing the skin of the orange with a knife or by hand. A misaligned AI might interpret this literally and suggest "placing the orange in a bowl and peeling it like a bell," completely misinterpreting the context of cooking.

Example 2: Path Optimization in Navigation Systems An AI system in a navigation app aims to find the shortest route to a destination. A human might expect the "best" route to avoid heavy traffic, construction zones, or take scenic roads when on a leisure drive. A misaligned AI would strictly calculate the shortest distance without considering these human preferences, potentially leading to a less satisfactory experience.

These examples demonstrate the importance of aligning AI systems with nuanced human values, preferences, and contexts to ensure they act in ways beneficial to their users.

9.3.5 Reward Model

A Reward Model is used to evaluate and guide the performance of a language model on specific tasks, such as summarization. The output samples s generated by the language model are evaluated based on a human-defined reward function $R(s)$, with higher scores indicating better alignment with the task objectives.

Evaluating Output Samples Given two output samples, s_1 and s_2, their rewards might be quantitatively expressed as $R(s_1)$ and $R(s_2)$. For instance:

- Sample s_1: "An earthquake hit San Francisco. There was minor property damage, but no injuries."
- Reward for s_1: $R(s_1) = 8.0$
- Sample s_2: "The Bay Area has good weather but is prone to earthquakes and wildfires."
- Reward for s_2: $R(s_2) = 1.2$

Sample s_1 is more informative and relevant to the event of an earthquake and therefore receives a higher reward.

Training the Reward Model The process involves labelers making pairwise comparisons between different samples, indicating which one better accomplishes the task. These human judgments are then used to train a reward model. The goal is to learn a function that predicts the reward for any given output s, attempting to maximize the expected reward over the distribution of samples $p_\theta(s)$:

$$\max_{\theta} \mathbb{E}_{s \sim p_\theta(s)}[R(s)]$$

The expectation $\mathbb{E}_{s \sim p_\theta(s)}[R(s)]$ represents the average reward that the language model would receive for outputs it generates, under the current model parameters θ. By maximizing this expectation, we train the language model to produce outputs that are more likely to be rated highly by human evaluators.

9.3.6 Instruct-GPT for Instructional and Creative Tasks

Understanding and Executing Specific Instructions Instruct-GPT demonstrates an enhanced ability to understand and execute specific instructions compared to its predecessor GPT-3. While GPT-3 may produce outputs that are tangentially related to the prompt, Instruct-GPT shows significant improvement in delivering on the user's actual intent.

Example of Instructional Prompt The prompt "Explain the moon landing to a 6 year old in a few sentences" resulted in InstructGPT generating a concise and age-appropriate explanation, showcasing its ability to tailor content to the audience's understanding level.

Example of Creative Prompt When prompted to "Write a short poem about a wise frog," InstructGPT created an on-topic poem that captures the essence of wisdom attributed to a frog, demonstrating its capacity for creativity:

"The frog is a symbol of wisdom
He knows all the secrets of the world
He is a master of disguise
And a great teacher of life
He is a symbol of transformation
And the bringer of change
He is the frog who has seen it all
And knows the meaning of it all"

In contrast, GPT-3 provided unrelated short story prompts, indicating a lack of specific response to the poetry format.

Implications for AI Development These examples underline the importance of fine-tuning approaches like Instruct-GPT, which aim to improve the alignment between AI-generated content and human expectations. The enhanced responsiveness to the context of the prompts makes Instruct-GPT a versatile tool for both instructional and creative purposes. It can aid in educational environments and as a collaborative aid in artistic endeavors.

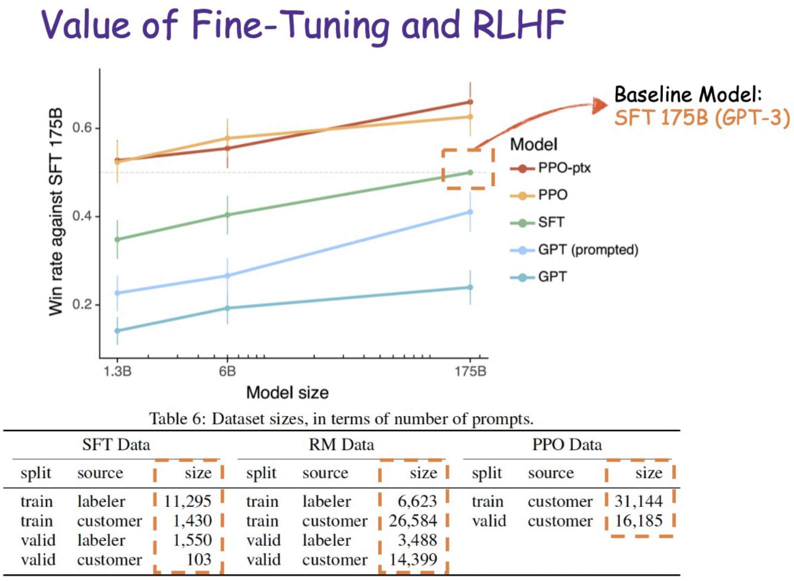
9.3.7 Value of Fine-Tuning and RLHF

The slide presents an empirical analysis of the impact of different fine-tuning methods on language models of various sizes. A graph illustrates the "win rate" of different models against a baseline Supervised Fine-Tuning (SFT) model with 175 billion parameters, known as GPT-3.

Graph Analysis The graph shows the performance of several models:

- Proximal Policy Optimization models fine-tuned with PPO (PPO-px)
- Models fine-tuned with standard Supervised Fine-Tuning (SFT)
- GPT models prompted for specific tasks (GPT (prompted))
- Standard GPT models without prompting (GPT)

Each model's performance is measured by its win rate against the SFT baseline model as the model size increases from 1.3 billion to 175 billion parameters.



Value of Fine-Tuning and RLHF

Table 6: Dataset sizes, in terms of number of prompts.

SFT Data			RM Data			PPO Data		
split	source	size	split	source	size	split	source	size
train	labeler	11,295	train	labeler	6,623	train	customer	31,144
train	customer	1,430	train	customer	26,584	valid	customer	16,185
valid	labeler	1,550	valid	labeler	3,488			
valid	customer	103	valid	customer	14,399			

Figure 9.4: Value of Fine-Tuning and RLHF

Dataset Table The accompanying table breaks down the sizes of the datasets used for each fine-tuning method in terms of the number of prompts. The datasets are split by the source (labeler or customer) and by usage (training or validation), providing insights into the quantity and source of data that were used for each fine-tuning method.

Interpretation This information highlights the importance of the quantity and quality of fine-tuning data, as well as the chosen RLHF method, in enhancing the performance of language models. As the size of the models increases, the benefits of fine-tuning and reinforcement learning become more pronounced, indicating that larger models are more capable of leveraging these advanced training techniques to improve their alignment with human preferences.

9.3.8 Issues with RLHF

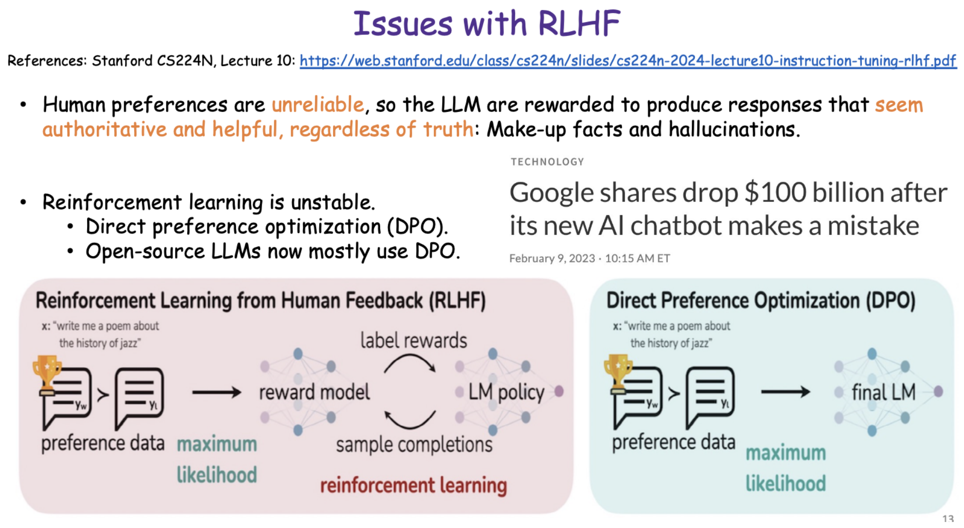
Reinforcement Learning from Human Feedback (RLHF) is a critical method in training LLMs to produce more human-like responses. However, this slide highlights two significant challenges with RLHF:

Unreliable Human Preferences Human preferences can sometimes be unreliable. This leads to LLMs being rewarded for producing responses that seem authoritative and helpful, but may lack factual accuracy. The slide emphasizes the problem of language models generating responses that, while seemingly confident, might include made-up facts or hallucinations.

Instability of Reinforcement Learning Reinforcement learning can be inherently unstable. The slide points out a specific approach known as Direct Preference Optimization (DPO), which has become more prevalent in open-source LLMs as a response to this instability. DPO aims to directly optimize the preferences without the complex reinforcement learning loop, thus providing a more stable training procedure.

The DPO Approach DPO simplifies the process by:

- Collecting preference data directly from human feedback.



Issues with RLHF

References: Stanford CS224N, Lecture 10: https://web.stanford.edu/class/cs224n/slides/cs224n-2024-lecture10-instruction-tuning-rlhf.pdf

- Human preferences are unreliable, so the LLM are rewarded to produce responses that seem authoritative and helpful, regardless of truth: Make-up facts and hallucinations.

TECHNOLOGY

Google shares drop $100 billion after its new AI chatbot makes a mistake

February 9, 2023 · 10:15 AM ET

- Reinforcement learning is unstable.
  - Direct preference optimization (DPO).
  - Open-source LLMs now mostly use DPO.

Reinforcement Learning from Human Feedback (RLHF)

x: "write me a poem about the history of jazz"

y_w > y_l

preference data

maximum likelihood

reward model

label rewards

sample completions

LM policy

reinforcement learning

Direct Preference Optimization (DPO)

x: "write me a poem about the history of jazz"

y_w > y_l

preference data

maximum likelihood

final LM

13

Figure 9.5: Issues with RLHF

- Using maximum likelihood to train the reward model.
- Applying the trained model to fine-tune the final LLM.

Real-World Consequences An example is given of a significant market impact due to a mistake made by an AI chatbot, illustrating the real-world implications of misaligned AI systems. The slide suggests the potential for substantial financial consequences when these systems do not perform as intended.

These issues underscore the necessity for careful consideration of the methods used in training LLMs, with a focus on developing stable and reliable approaches that align closely with accurate and responsible human judgment.

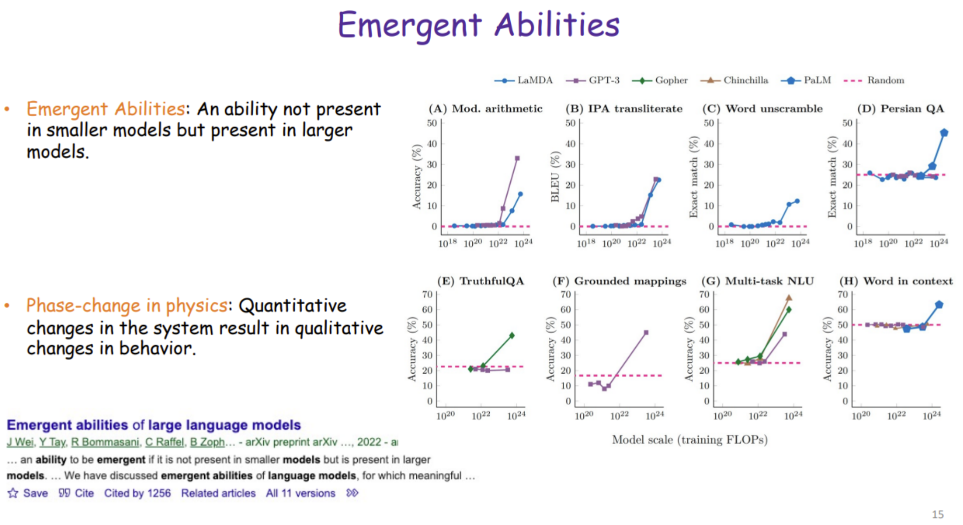
9.3.9 Hallucination

LLMs tend to excel in tasks requiring intuitive responses but struggle with tasks requiring long-term planning due to current technical constraints. Hallucination in this context refers to the phenomenon where these models generate incorrect, misleading, or entirely fabricated information despite presenting it in a confident manner. This is an inevitable outcome given the current state of technology.

9.3.10 Emergent Abilities

Emergent Abilities refer to an ability not present in smaller models but present in larger models. When we start with small model size, the performance of the model is like random guess. However, after the model size reaches some certain level, the accuracy suddenly exhibits a huge jump.

The emergent abilities are not about the ability of the model, but about how we choose the performance metric. The paper "Are emergent abilities of large language models a mirage?" found that emergent Abilities may be attributed to the choice of nonlinear or discontinuous metrics, whereas linear or continuous metrics produce smooth performance changes.



Emergent Abilities

- Emergent Abilities: An ability not present in smaller models but present in larger models.
- Phase-change in physics: Quantitative changes in the system result in qualitative changes in behavior.

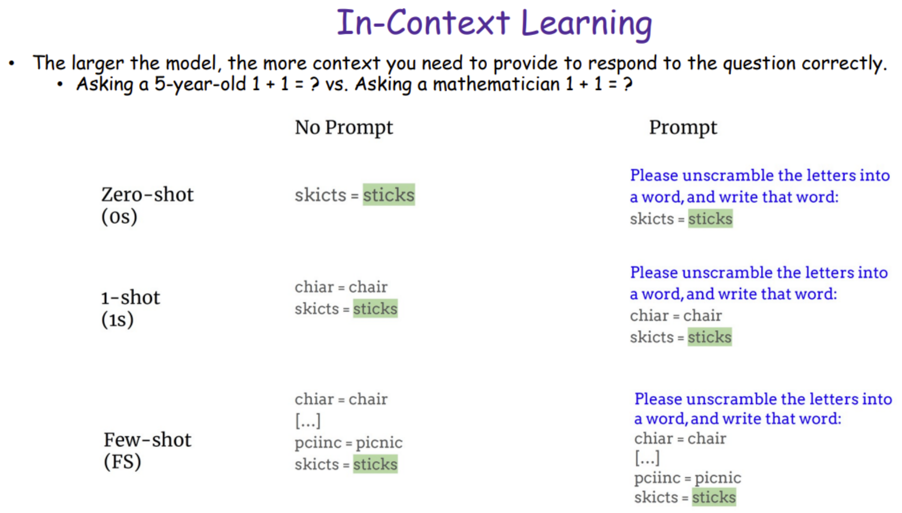
Emergent abilities of large language models
J Wei, Y Tay, R Bommasani, C Raffel, B Zoph… - arXiv preprint arXiv …, 2022 - ar
… an ability to be emergent if it is not present in smaller models but is present in larger models. … We have discussed emergent abilities of language models, for which meaningful …
☆ Save 99 Cite Cited by 1256 Related articles All 11 versions

Figure 9.6: Emergent Abilities

9.3.11 In-Context Learning

Large Language Models can infer from provided context, improving their responses with the size of the model. The effectiveness of in-context learning can be enhanced by various prompting methods:

- **Zero-shot prompting**: The model is given a task without any previous examples.
- **One-shot prompting**: A single example is provided alongside the task.
- **Few-shot prompting**: Several examples are provided to help the model generalize the task requirements better.



In-Context Learning

- The larger the model, the more context you need to provide to respond to the question correctly.
  - Asking a 5-year-old 1 + 1 = ? vs. Asking a mathematician 1 + 1 = ?

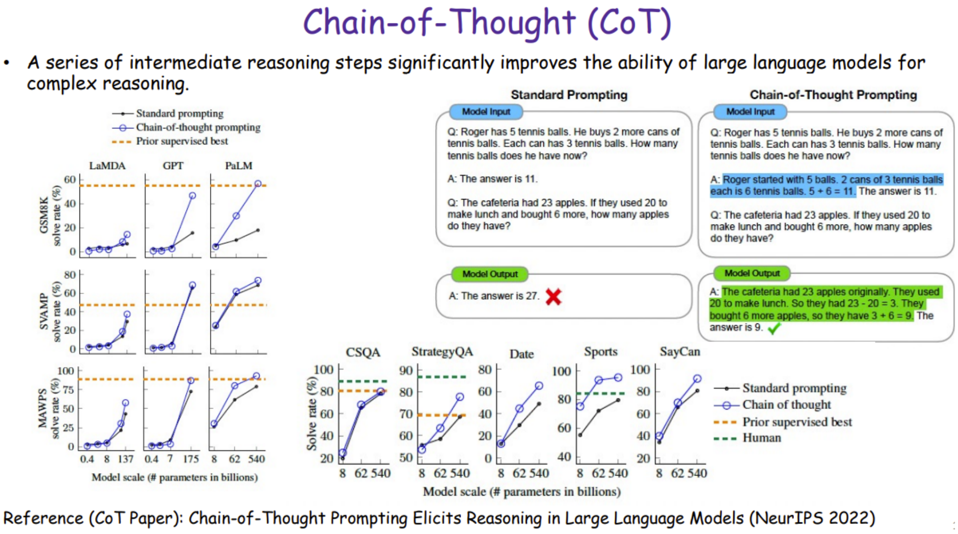
	No Prompt	Prompt
Zero-shot (0s)	skicts = sticks	Please unscramble the letters into a word, and write that word: skicts = sticks

1-shot (1s)	chiar = chair
	skicts = sticks
		Please unscramble the letters into a word, and write that word:
		chiar = chair
		skicts = sticks

Few-shot (FS)	chiar = chair
	[...]
	pciinc = picnic
	skicts = sticks
		Please unscramble the letters into a word, and write that word:
		chiar = chair
		[...]
		pciinc = picnic
		skicts = sticks

Figure 9.7: In-Context Learning

Chain-of-Thought Introduced in Wei et al. (2022), chain-of-thought (CoT) prompting enables complex reasoning capabilities through intermediate reasoning steps. You can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding.



Chain-of-Thought (CoT)

- A series of intermediate reasoning steps significantly improves the ability of large language models for complex reasoning.

Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27. ✗

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Reference (CoT Paper): Chain-of-Thought Prompting Elicits Reasoning in Large Language Models (NeurIPS 2022)

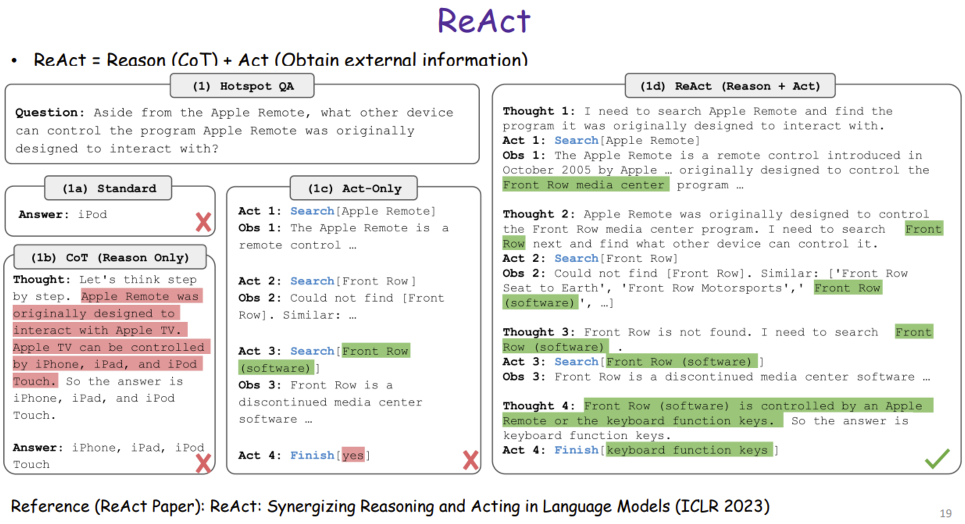
Figure 9.8: Chain-of-Thought

ReAct Yao et al., 2022 introduced a framework named ReAct where LLMs are used to generate both reasoning traces and task-specific actions in an interleaved manner. The ReAct framework can allow LLMs to interact with external tools to retrieve additional information that leads to more reliable and factual responses. ReAct is a general paradigm that combines reasoning and acting with LLMs. ReAct prompts LLMs to generate verbal reasoning traces and actions for a task. This allows the system to perform dynamic reasoning to create, maintain, and adjust plans for acting while also enabling interaction to external environments (e.g., Wikipedia) to incorporate additional information into the reasoning. The figure below shows an example of ReAct and the different steps involved to perform question answering.

Reflection: Self-Reflecting LLM Reflexion is a framework introduced by Shinn et al. (2023) to reinforce language-based agents through linguistic feedback, a new paradigm for verbal reinforcement. This framework utilizes a combination of free-form language or scalar feedback from the environment, converting this feedback into linguistic cues referred to as self-reflection. These cues are then used as context for an LLM agent in subsequent interactions, allowing rapid and effective learning from previous mistakes and leading to performance improvements in complex tasks.

Reflexion comprises three main components:

- **An Actor**: This component generates text and actions based on observed states. The Actor engages with the environment, takes actions, receives observations, and produces a trajectory. Models like Chain-of-Thought (CoT) and ReAct are utilized to enrich the Actor's responses and decisions, with a memory component added to enhance contextual understanding.
- **An Evaluator**: This component is responsible for scoring the outputs produced by the Actor. It evaluates the generated trajectories (short-term memory) and assigns a reward



ReAct

- ReAct = Reason (CoT) + Act (Obtain external information)

(1) Hotspot QA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

(1a) Standard

Answer: iPod ✗

(1b) CoT (Reason Only)

Thought: Let's think step by step. Apple Remote was originally designed to interact with Apple TV. Apple TV can be controlled by iPhone, iPad, and iPod Touch. So the answer is iPhone, iPad, and iPod Touch.

Answer: iPhone, iPad, iPod Touch ✗

(1c) Act-Only

Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control …

Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: …

Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Act 4: Finish[yes] ✗

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software) ', …]

Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys] ✓

Reference (ReAct Paper): ReAct: Synergizing Reasoning and Acting in Language Models (ICLR 2023)

19

Figure 9.9: ReAct

score, which varies based on the task complexity and the decision-making framework employed (LLMs and rule-based heuristics).

- **Self-Reflection**: The Self-Reflection component generates verbal reinforcement cues that help the Actor improve. Utilizing the feedback from the Evaluator, it provides specific, relevant feedback stored in memory. This feedback, drawn from the reward signal and the trajectory data stored in long-term memory, aids the Actor in enhancing decision-making skills for future tasks.

The Reflexion process involves several key steps: defining a task, generating a trajectory, evaluating this trajectory, performing reflection, and finally, generating the next trajectory. This cyclic process helps a Reflexion agent to iteratively optimize its behavior to effectively solve various complex tasks, including decision-making, programming, and reasoning. Reflexion extends the ReAct framework by introducing additional elements of self-evaluation, self-reflection, and memory, thereby enhancing the overall capability of the system.

9.4 Applications of LLM in Econ/Business Research

9.4.1 LLM for Mathematical Discoveries

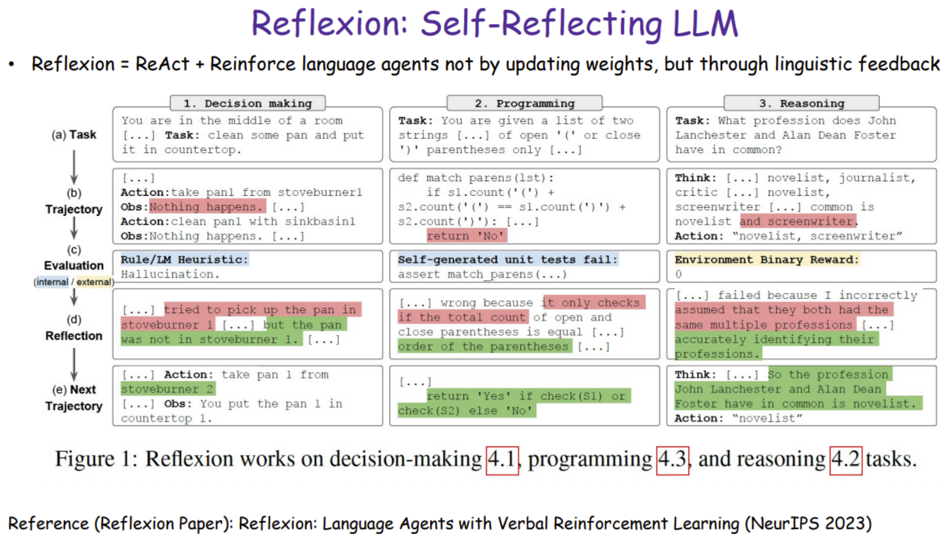
The paper "Program Synthesis with Large Language Models" was published by Nature as part of research efforts associated with institutions such as Google Research.

The main idea of the paper is that LLM helps find new solutions to challenging combinatorial problems that surpass the best-known results. They developed an evolutionary procedure that pairs a pretrained LLM with a systematic evaluator.

There's no fine-tuning in this procedure. But after some iterations, the model gives us something that is beyond state-of-the-art human intelligence.

9.4.2 Jobs Exposed to Generative AI

The paper "GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models" investigates the potential implications of large language models, specifically



Reflexion: Self-Reflecting LLM

- Reflexion = ReAct + Reinforce language agents not by updating weights, but through linguistic feedback

	1. Decision making	2. Programming	3. Reasoning
(a) Task	You are in the middle of a room [...] **Task:** clean some pan and put it in countertop.	**Task:** You are given a list of two strings [...] of open '(' or close ')' parentheses only [...]	**Task:** What profession does John Lanchester and Alan Dean Foster have in common?

(b) Trajectory	[...]
	Action:take pan1 from stoveburner1
	Obs: Nothing happens. [...]
	Action:clean pan1 with sinkbasin1
	Obs: Nothing happens. [...]
		def match parens(lst):
		    if s1.count('(') + s2.count('(') == s1.count(')') + s2.count(')'): [...]
		        return 'No'
			Think: [...] novelist, journalist, critic [...] novelist, screenwriter [...] common is novelist and screenwriter.
			Action: "novelist, screenwriter"

(c) Evaluation (internal / external)	**Rule/LM Heuristic:** Hallucination.	**Self-generated unit tests fail:** assert match_parens(...)	**Environment Binary Reward:** 0
(d) Reflection	[...] tried to pick up the pan in stoveburner 1 [...] but the pan was not in stoveburner 1. [...]	[...] wrong because it only checks if the total count of open and close parentheses is equal [...] order of the parentheses [...]	[...] failed because I incorrectly assumed that they both had the same multiple professions [...] accurately identifying their professions.
(e) Next Trajectory	[...] **Action:** take pan 1 from stoveburner 2 [...] **Obs:** You put the pan 1 in countertop 1.	[...] return 'Yes' if check(S1) or check(S2) else 'No'	**Think:** [...] So the profession John Lanchester and Alan Dean Foster have in common is novelist. **Action:** "novelist"

Figure 1: Reflexion works on decision-making 4.1, programming 4.3, and reasoning 4.2 tasks.

Reference (Reflexion Paper): Reflexion: Language Agents with Verbal Reinforcement Learning (NeurIPS 2023)

Figure 9.10: Reflexion

Generative Pre-trained Transformers (GPTs), on the U.S. labor market. It focuses on the capabilities of LLM-powered software and its integration with human expertise.

The paper identifies GPTs as a type of general-purpose technology, suggesting substantial economic, social, and policy implications across all wage levels, not just industries known for high productivity growth.

9.4.3 Generative AI Improves Productivity and Equality

The paper “Experimental Evidence on the Productivity Effects of Generative Artificial Intelligence" explores the impact of a generative AI technology, specifically the assistive chatbot ChatGPT, on midlevel professional writing tasks. The study used a preregistered online experiment involving 453 college-educated professionals who were randomly assigned writing tasks with or without access to ChatGPT. It was discovered that ChatGPT significantly increased productivity, reducing the average time taken for tasks by 40% and improving output quality by 18%.

The paper "ChatGPT outperforms crowd workers for text-annotation tasks" explores the efficiency of ChatGPT in performing various text annotation tasks compared to human crowd workers.

Both papers suggest that if we were able to use the LLM augmentation techniques well, the productivity of intelligence tasks can be improved.

9.4.4 Generative AI Powered Conversational Assistants

A more recent paper “Generative AI at Work” examines the effects of a generative AI-based conversational assistant on the productivity of 5,000 customer support agents. The study found that the tool increased productivity by 14% on average, particularly benefiting novice and low-skilled workers, while having a minimal impact on experienced and highly skilled workers

This paper provides valuable insights into how generative AI can enhance workplace efficiency and worker engagement across different skill levels.

9.4.5 Search Engine Optimization

The paper “Frontiers: Determining the Validity of Large Language Models for Automated Perceptual Analysis” explores the potential of using LLMs to substitute for human participants in market research.

What they discovered is that LLMs generate 75%-similar data to those generated from human surveys, with respect to both brand similarity and product attribute ratings.

Moreover, LLM-generated data well captures consumer heterogeneity.

9.4.6 Perceptual Analysis

The paper "Frontiers: Determining the Validity of Large Language Models for Automated Perceptual Analysis" investigates the capability of LLMs to mimic human participants in market research. The findings indicate that LLMs can generate data with 75% similarity to human-generated data, capturing significant consumer heterogeneity.

9.4.7 LLMs as Simulators for Humans

A recent stream of studies aims to answer the question whether LLMs can serve as computational models for humans? To be more specific, can LLMs work as human simulators?

What most studies do is trying to replicate classical researches by using LLM simulators instead of humans. And some qualitatively similar results to some classical economic research have been successfully replicated by LLM simulators.

But the question remains whether we can really trust results from AI agents. The boundary of using AI to simulate humans in experiments and surveys still requires further exploration.

9.4.8 LLM-Human Collaborations

The paper “Large Language Model in Creative Work: The Role of Collaboration Modality and User Expertise” conducted an experiment where expert and non-expert users are tasked to write an ad copy with and without the assistance of LLMs.

It was found that quality improves especially for non-experts while quality does not improve and decreases for experts. It was also shown that different collaboration modalities can result in different outcomes for different user types.

When using LLMs as ghost-writers, i.e., asking LLM to write the first draft for user, it produced an anchoring effect which led to lower quality ads, especially for experts. When using LLMs as sounding boards, i.e., asking LLM to provide revision advice for the first draft written by the user, it helped non-experts achieve ad content with low semantic divergence to content produced by experts, thereby closing the gap between the two types of writers.

10

Deep-Learning-Based Image Classification

——**Scribed by Qiansiqi Hu**

10.1 Outline

The purpose of this document is to give a crash course on computer vision and image classification. The focus lies on a series of deep learning models that are applied to image classification, including CNN (LeCun et al., 1998), AlexNet (Krizhevsky et al., 2012), VGG (Simonyan and Zisserman, 2014), ResNet (He et al., 2016) and ViT (Dosovitskiy et al., 2020). For lecture notes and Jupyter notebooks of this course, we will introduce:

- Preliminaries of image classification
- Details and executable Python code demonstration of the representative deep learning models
- Typical applications of these methods in business and economic research.

Specifically, the outline of this document is as follows:

- In Section 10.2, we introduce the basic framework for image classification.
- In Section 10.3, we introduce the most representative deep learning models for image classification.
- In Section 10.4, we briefly introduce how image classification techniques are applied in business and economic research.
- In Section 10.5, we list all the demos that are related to the materials of this document.

10.2 Image Classification in a Nutshell

Image classification is a fundamental task in computer vision that involves categorizing or labeling images into predefined classes. The goal is to develop models or algorithms that can automatically identify and assign appropriate labels to images based on their visual content.

10.2.1 Image Representation

Generally, the process of image classification can be considered as a representation problem and a subsequent classification problem.

Pixel-based representation is the most basic form of image representation, where each pixel in an image is treated as a separate entity. Pixel values can be directly used as features or combined to form higher-level representations, such as color histograms or texture descriptors. Figure 10.1 illustrates an example of pixel-based representation in the RGB (Red, Green, Blue) color space.

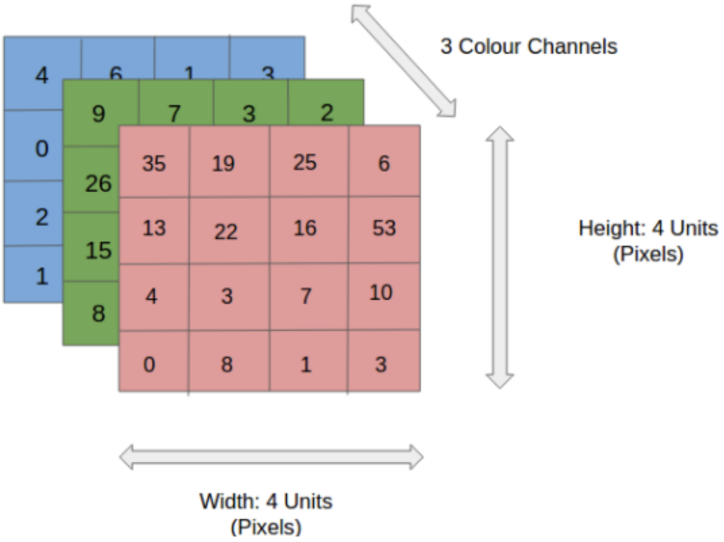


Figure 10.1: Image representation in the RGB space.

To be specific, the RGB color model represents colors as combinations of red, green, and blue channels. Each color channel is an 8-bit value ranging from 0 to 255, indicating the intensity of the respective color. A value of 0 represents the absence of that color, while 255 stands for the maximum intensity.

An image is composed of a grid of pixels, where each pixel contains information about color. In the RGB space, the image is split into three separate color channels. For instance, in Figure 10.1, the pixel in the top-left corner has value $[4, 9, 35]$. For each pixel in the image, a channel represents the intensity values of its corresponding color. In general, image with $W \times L$ pixels can be represented as a three-dimensional array of shape $3 \times W \times L$.

Pixel-based image representation is simple but lacks the ability to capture spatial relationships between pixels. With the emergence of deep learning, convolutional neural networks (CNNs) have become a popular approach for better learning image representations. It is crucial for achieving accurate and effective analysis and interpretation of visual data. The working principles of CNN will be further explained in Section 10.3.1.

10.2.2 The Role of Machine Learning in Image Classification

Representation learning is a process of extracting discriminative features from the images. The extracted features are typically encoded into a fixed-length representation that can be easily processed by classification algorithms. This step aims to capture the most informative characteristics of the image while reducing the dimensionality of the feature space. Common encoding techniques include vector quantization, histogram-based encoding (e.g., bag-of-visual-words), or dimensionality reduction methods like Principal Component Analysis (PCA) or t-SNE.

The encoded image representations, along with their labels, are used to train a classifier. Various machine learning algorithms can be employed, such as nearest neighbors, support vector machines (SVM), or random forests. During training, the classifier learns to map the encoded features to their labels, optimizing its parameters to minimize the classification error.

Now the general process is clear: an image classification model can be divided into two parts. The first part is a feature extractor $\Phi(\cdot)$ that learns representations from images. The second part is a classifier $f(\cdot)$ that predicts the label of images based on their representations.

The task of image recognition/classification is challenging in general. Before the rise of deep learning, image classification relied on handcrafted features and traditional machine learning algorithms. As shown in Figure 10.2, without an efficient way of representation learning, it faces a series of inherent problems such as viewpoint variation, deformation, etc.

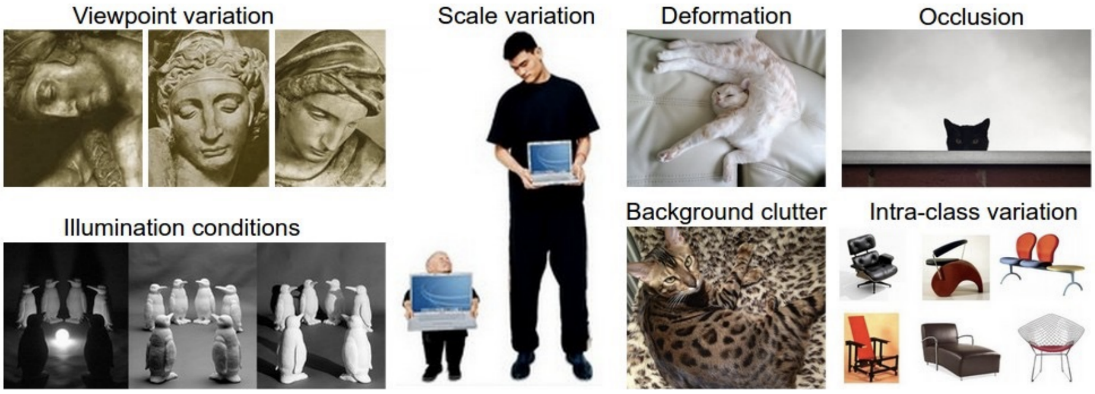


Figure 10.2: Inherent challenges in image classification (Fei-Fei and Adeli, 2024).

However, with the advent of deep learning, significant advancements have been made to image classification accuracy. Deep learning models leverage the hierarchical structure of convolutional neural networks to automatically learn and extract relevant features from images. These models consist of multiple layers of interconnected neurons that perform convolutions, pooling, and non-linear transformations. The lower layers capture low-level features such as edges and textures, while the higher layers capture more abstract and complex features.

Training a deep learning model for image classification involves feeding it with a large labeled dataset, known as the training set. We describe the general procedures in Algorithm 12. During training, the model learns to adjust its internal parameters or weights to minimize the difference between its predicted labels and the true labels of the training images. This process is typically performed using optimization algorithms like stochastic gradient descent (SGD) or its variants.

Once trained, the image classification model can be deployed to classify new, unseen images. The model takes an input image, processes it through its layers, and produces a probability distribution over the predefined classes. The class with the highest probability is considered the predicted label for the input image.

In sum, the continuous advancements in deep learning have enabled machine learning techniques to be simultaneously involved in representation learning and classification. They have significantly improved the accuracy and reliability of image classification systems, enabling their widespread adoption in various domains.

```
Algorithm 12 General Procedures for Deep-Learning-Based Image Classification
Input: A training set $\mathcal{D} := \{Y_i, X_i : 1 \leq i \leq n\}$. Here, the label $Y$ is discrete, and features $X$
    are usually three-dimensional arrays as illustrated in Section 10.2.1.
Goal: A model trained on $\mathcal{D}$, composed of $\hat{\Phi}(\cdot)$ and $\hat{f}(\cdot)$, such that the generalization error
    $\mathbb{E}\left[\mathcal{L}\left(Y, \hat{f}(\hat{\Phi}(X))\right)\right]$ is minimized, where $\mathcal{L}(\cdot,\cdot)$ is the loss function and the expectation is
    taken with respect to the "true" distribution that generates the training data.
Procedure:
1: Define $\Phi(\cdot)$ and $f(\cdot)$. Usually $\Phi(\cdot)$ is composed of convolutional layers, pooling layers and
    activation functions. $f(\cdot)$ is a feedforward neural network.
2: Pick your optimizer, such as Adam (Kingma and Ba, 2014), and set a learning rate for it.
3: Divide the training samples into multiple batches. Train the deep learning model by mini-
    batch gradient descent (usually on GPUs).
4: Evaluate the performance of $\hat{\Phi}(\cdot)$ and $\hat{f}(\cdot)$ on test sets, which consist of unseen samples
    from the "true" distribution.
```

10.3 Deep Learning Models for Image Classification

10.3.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are first introduced by LeCun et al. (1998) in the context of computer vision. The model proposed in this work is named LeNet. CNN is a type of deep learning architecture designed specifically for processing structured grid-like data. It has revolutionized the field of computer vision and is widely used for various tasks, including image classification, object detection, and image segmentation.

Convolution The key idea behind CNNs is to leverage the spatial relationships present in images. They employ learnable filters, called convolutional kernels, to perform local operations on small patches of the input image. These filters are applied across the entire image to capture different features, such as edges, textures, and shapes.

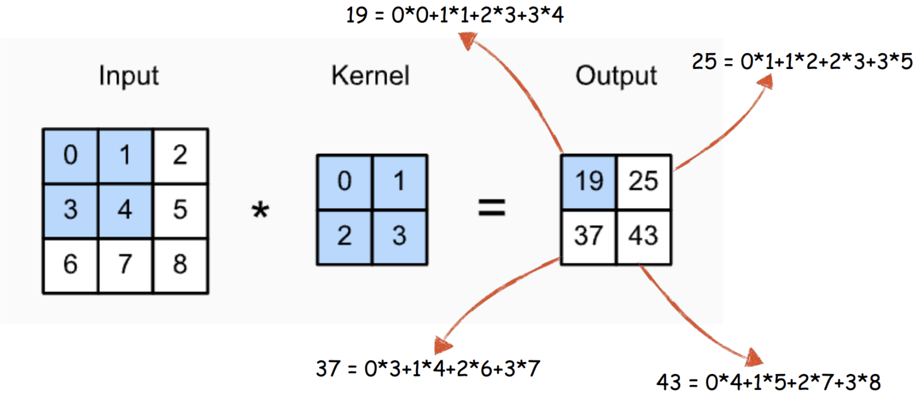


Figure 10.3: Sketch of how convolution is performed.

Figure 10.3 illustrates an example of how convolution is performed. Assume the kernel size is 2×2, then an image with 3×3 pixels can be divided into four 2×2 grids. The inner product with the kernel is respectively computed for the 4 grids, hence the result of convolution is a 2×2 matrix.

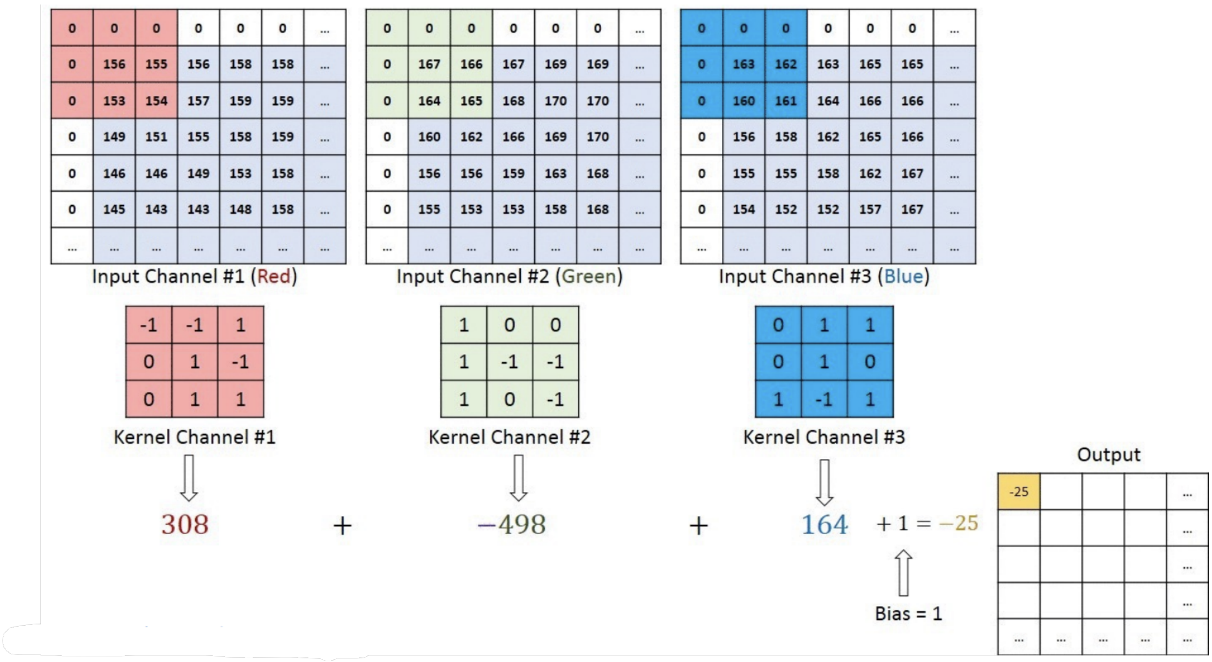
There are some additional parameters for convolution:

- Padding: Adding extra border pixels (usually zeros) around the input image before applying convolutional filters, ensuring that each pixel in image is covered the same times.
- Stride: The step size at which the convolutional filter moves across the input image or feature map. Instead of moving the filter by one pixel at a time, stride allows for skipping a certain number of pixels.

Assume the input is an $N_h \times N_w$ matrix, shape of the kernel is $K_h \times K_w$, stride equals s and padding equals p (namely the width of the extra pixel border is p). Then the output has shape $M_h \times M_w$, where

$$M_h = \frac{N_h + 2p - K_h}{s} + 1,\ M_w = \frac{N_w + 2p - K_w}{s} + 1 \tag{10.1}$$

Moreover, a multi-channel convolutional layer contains 2 additional parameters, which are the numbers of input channels and output channels. For example, the number of input channels is 3 for images in the RGB space. In this case, the input to the convolutional layer is a three-dimensional array. Assume the input has shape $C_i \times N_h \times N_w$, then the output will be a matrix of shape $C_o \times M_h \times M_w$, where C_o is the number of output channels. The convolutional layer contains $C_i \times C_o$ kernels in total, and the kernels are different from each other. Figure 10.4 well demonstrates the operation logic of a multi-channel convolutional layer.



Input Channel #1 (Red)

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Kernel Channel #1

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #2

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #3

0	1	1
0	1	0
1	-1	1

Output

-25				...
				...
				...
				...
...	...	...	...	...

Figure 10.4: Multi-channel convolution of images in the RGB space, where $N_h = N_w = 5$, $K_h = K_w = 3$, $s = 1$, $p = 1$, $C_i = 3$, $C_o = 1$.

For a single CNN layer, the computational complexity is $\mathrm{O}(C_i * C_O * K_h * K_w * M_h * M_w)$. Assume $C_i = C_o = 100$, $K_h = K_w = 5$, $M_h = M_w = 65$, then there will be around 10^9 floating point operations for one CNN layer. Assume there are 10 CNN layers and 1 million images, then the model has to complete 10^{16} floating point operations. If we use CPU, the time for one pass is around 18 hours. As a comparison, if we use GPU, say NVIDIA GeForce RTX 3090, the time cost can be reduced to 8.3 minutes.

Pooling Pooling layer is an essential component in CNN that helps reduce the spatial dimensions of feature maps while retaining the most important information. Pooling is typically applied after convolutional layers to downsample the feature maps.

There are different types of pooling operations, with the most commonly used one being Max Pooling:

1. Max Pooling: Max pooling partitions the input feature map into non-overlapping rectangular regions (usually with a stride equal to the size of region, which is referred as kernel size) and outputs the maximum value within each region. This operation captures the most prominent feature within each region and discards the rest.

2. Average Pooling: Average pooling is similar to max pooling, but instead of taking the maximum value, it calculates the average value within each pooling region. Average pooling provides a smoothed representation of the input and can be useful in certain scenarios.

We visualize the process of pooling in Figure 10.5. Similar to convolution, we can also do padding and stride in pooling operations, although padding is usually set to zero.

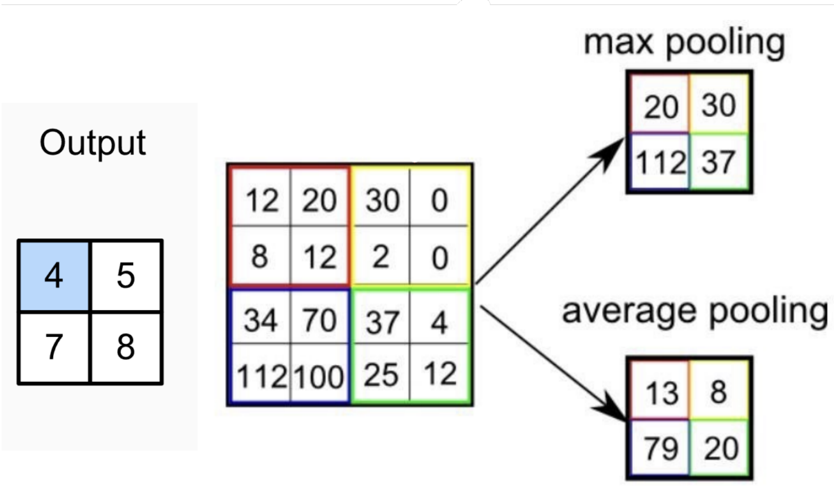


Figure 10.5: The process of pooling with stride $s = 2$.

Pooling layers have several benefits:

- Dimensionality Reduction: Pooling reduces the spatial dimensions of feature maps, resulting in a smaller number of parameters and computational requirements in subsequent layers. This helps to control overfitting and improve the efficiency of the network.

- Translation Invariance: Pooling creates a level of translation invariance by summarizing local information. This means that even if an object is slightly shifted within the input image, the pooled feature map will still capture its presence.

- Robustness to Noise: Pooling helps to reduce the impact of noise or small variations in the input by focusing on the most prominent features and disregarding minor fluctuations.

- Hierarchical Feature Extraction: Pooling is typically applied after each set of convolutional layers, allowing the network to capture increasingly abstract and high-level features as the spatial resolution decreases.

In sum, CNNs extract hierarchical features by applying convolutional filters and pooling operations. The activations of intermediate layers or the output of the last fully connected layer can be used as image representations, often referred to as feature vectors or embeddings. Geometrically, convolution allows us to emphasize on certain features that are rotational or positional invariant. Moreover, CNNs are extremely prone to overfitting, so they are easy to estimate with reasonable amount of information.

10.3.2 AlexNet

AlexNet (Krizhevsky et al., 2012) was a pioneering CNN architecture that won the ImageNet (Deng et al., 2009) Large Scale Visual Recognition Challenge (ILSVRC) in 2012, marking a breakthrough in image classification accuracy.

Compared to LeNet (LeCun et al., 1998), AlexNet is larger, deeper, and trained on more data. It requires 10^9 floating point operations for one image. The difference in architecture is shown in Figure 10.6. AlexNet demonstrates a substantial improvement in ImageNet classification error over the best non-DL method by a wide margin (from 25.8% to 16.4%). Its groundbreaking success revealed the power of deep learning and CNNs for image classification tasks. Its architecture, along with the use of GPUs for training, paved the way for subsequent advancements in computer vision.

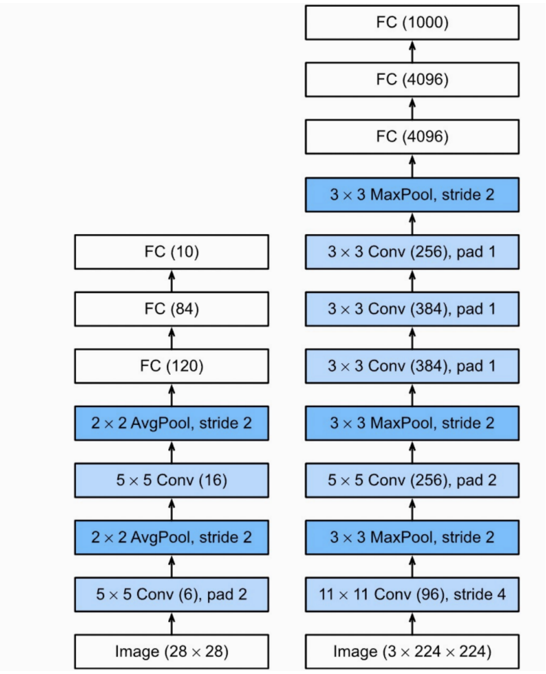


Figure 10.6: Comparison between LeNet (left) and AlexNet (right). Plotted by Zhang et al. (2021).

10.3.3 Visual Geometry Group

Visual Geometry Group (VGG) is a widely recognized CNN architecture developed by Simonyan and Zisserman (2014), which achieved excellent performance in image classification tasks.

The key characteristics of VGG are as follows:

- Deep Structure: VGG is known for its depth. It is available in different variants, such as VGG16 and VGG19, which refer to the number of layers in the network (16 or 19

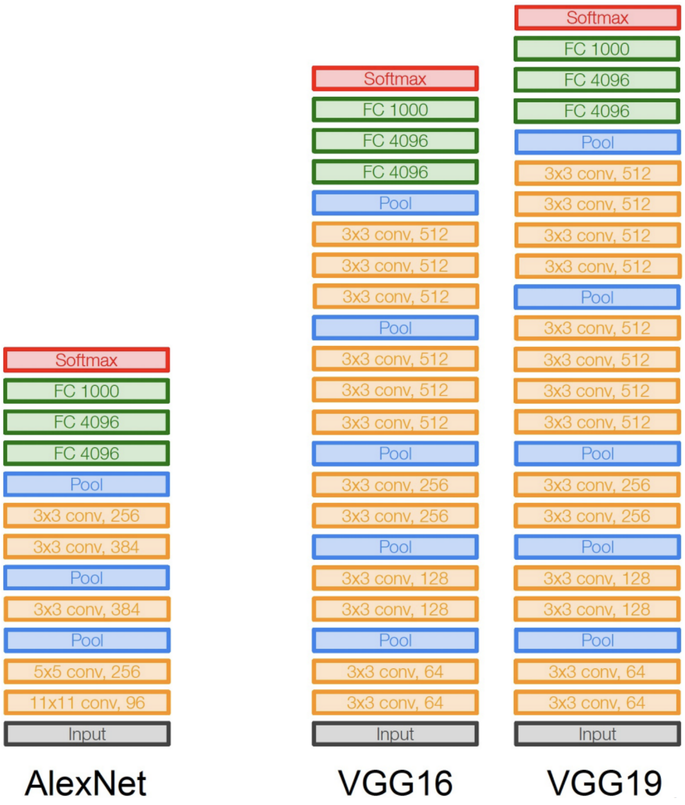


Figure 10.7: Comparison between AlexNet and VGG (Fei-Fei et al., 2023)

layers, respectively). The increased depth allows VGG to learn more complex features and capture finer details from images.

- Small Convolutional Kernels: VGGNet employs small 3×3 convolutional kernels throughout the network. Using multiple stacked 3×3 kernels allows the network to learn more sophisticated features while keeping the receptive field the same as a larger kernel size.

Difference between AlexNet and VGG in terms of architecture is illustrated in Figure 10.7. VGG reduced the classification error on ImageNet from 16.4% to 7.3%, demonstrating its effectiveness in recognizing and classifying diverse objects within images.

10.3.4 ResNet

ResNet (Residual Neural Network) is a groundbreaking CNN architecture that addresses the challenge of training deep neural networks by introducing residual connections. Proposed by He et al. (2016), ResNet has become one of the most influential and widely used CNN architectures.

Background Intuitively, adding new layers to a neural network model should, in theory, lead to effective reduction in training error. This is because the solution space of the original model is just a subspace of the solution space of the new model. In other words, if we can train the newly added layers to be an identity mapping, namely $f(x) = x$, then the new model and the original model should be equally effective. Since the new model may find better solutions to fit the training dataset, it seems easier to reduce the training error by adding layers.

However, in practice, adding too many layers often results in an increase in training error instead of a decrease. Even though the numerical stability brought by batch normalization makes it easier to train deep models, this problem still exists.

The core idea behind ResNet is that each additional neural layer should readily incorporate the identity function. These insightful considerations gave birth to a surprisingly straightforward solution, known as the residual block. ResNet outperformed VGG on ImageNet, reducing the classification error from 7.3% to 3.6% with lower time complexity. Deeper ResNets have lower error. The design has profound influence on the later development of deep neural networks, being applied to famous architectures such as graph neural networks (Kipf and Welling, 2016) and transformers (Vaswani et al., 2017).

Residual Block The basic structure of a residual block is illustrated in Figure 10.8. Assume the input is x, $H(x)$ is the desired mapping to be learnt by the neural layers. In a residual block, we hope the small subnet fit $F(x)$ instead of $H(x)$, where $F(x) = H(x) - x$. $F(x)$ is called a residual mapping with respect to identity. In practice, it is often easier to learn the residual mapping. If $H(x) = x$ is the desired mapping, then $F(x) = 0$, and we just need to make the weights and biases of the weight layers approach zero.

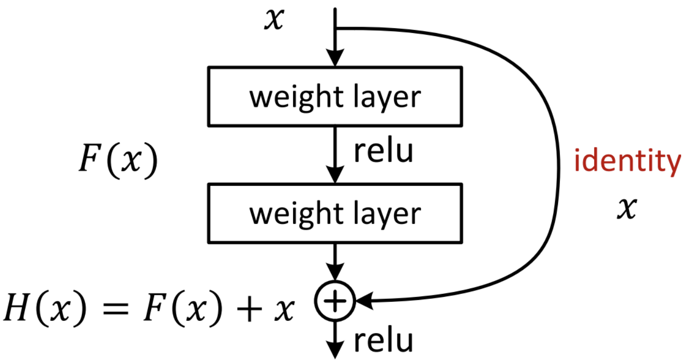


Figure 10.8: The basic structure of a residual block (He, 2018).

ResNet Model In ResNet, the first two layers include a 7×7 convolutional layer and a 3×3 max pooling layer, where 7×7 and 3×3 refer to the kernel size. Batch normalization is applied to the output of convolutional layers.

Besides, a ResNet model contains four modules that are made up of residual blocks. The detailed structure of residual block is depicted in Figure 10.9. Assume each module consists of two residual blocks. Given that each residual block contains two convolutional layers, there are 16 convolutional layers in the four modules. If we include the first 7×7 convolutional layer and the last fully connected layer, then there are 18 layers in total, and this model is well known as ResNet-18. Figure 10.10 illustrates the complete structure of ResNet-18. By increasing the number of residual blocks in the module, we can create deeper ResNet models, such as ResNet-152.

In sum, ResNet's introduction of residual connections has revolutionized the field of deep learning, enabling the successful training of extremely deep neural networks. Its ability to handle deep architectures with improved performance has been widely adopted in various computer vision tasks, including image classification, object detection, and image segmentation.

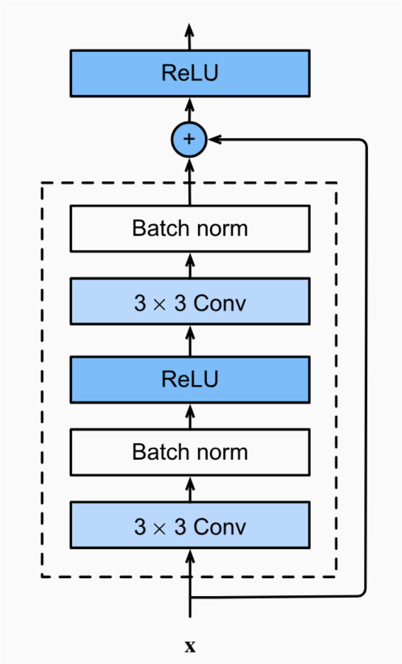


Figure 10.9: The composition of residual blocks in ResNet (Zhang et al., 2021).

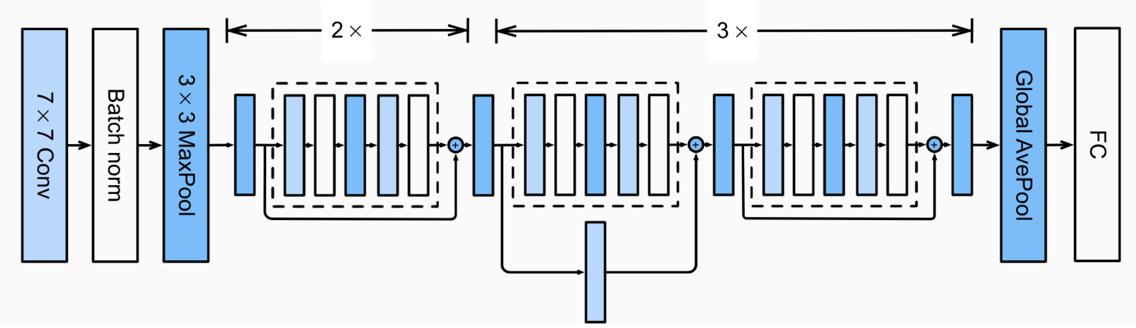


Figure 10.10: The complete structure of ResNet-18 (Zhang et al., 2021).

10.3.5 Vision Transformer

Known as ViT, vision transformer (Dosovitskiy et al., 2020) is a deep learning architecture that applies the transformer model (Vaswani et al., 2017), originally designed for natural language processing (NLP), to image classification tasks.

The transformer model leverages self-attention mechanisms to capture long-range dependencies in sequences. The same principles were applied to vision tasks through the vision transformer. It treats an image as a sequence of patches, where each patch corresponds to a small region in the image. These patches are then linearly embedded into a sequence of token vectors, which serve as input to the transformer model. An overview of the ViT model is depicted in 10.11.

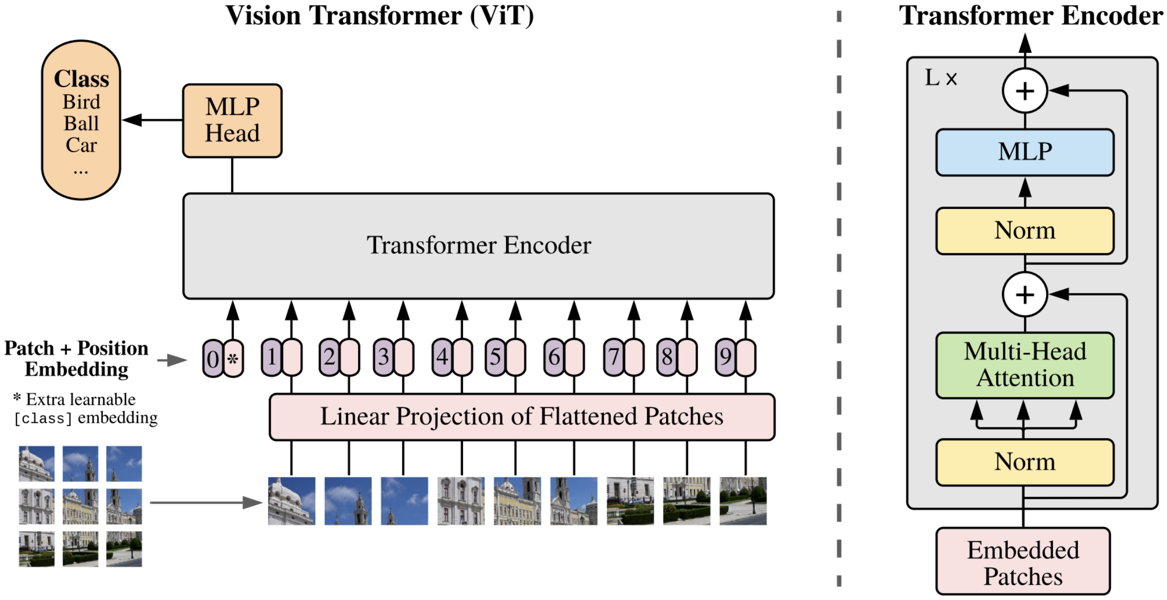


Figure 10.11: The structure of vision transformer (Dosovitskiy et al., 2020).

Figure 10.12 compares the performance of ViTs to ResNets. Within the same computational budget, vision transformers are generally better than ResNets. A hybrid of CNN and transformer may achieve even better results, since transformer lacks the inductive bias of CNN.

10.4 Applications in Business and Economic Research

In this section, we list some recent works that have applied deep learning and image classification techniques to business and economic research.

- **ML-Driven Hypothesis Generation**. Ludwig and Mullainathan (2024) apply CNN algorithms to generate interpretable and testable hypotheses on human behaviors. It is found that using mug shots and machine learning to predict judge behavior in jailing decisions can uncover (about 22.3%) new information never discovered before. They create counterfactual mug shots based on the algorithmic discovery and iterate with crowd-sourced workers to confirm what the human judges see is the same as what the ML algorithm sees, thus formalizing the algorithmic discoveries as testable hypotheses.

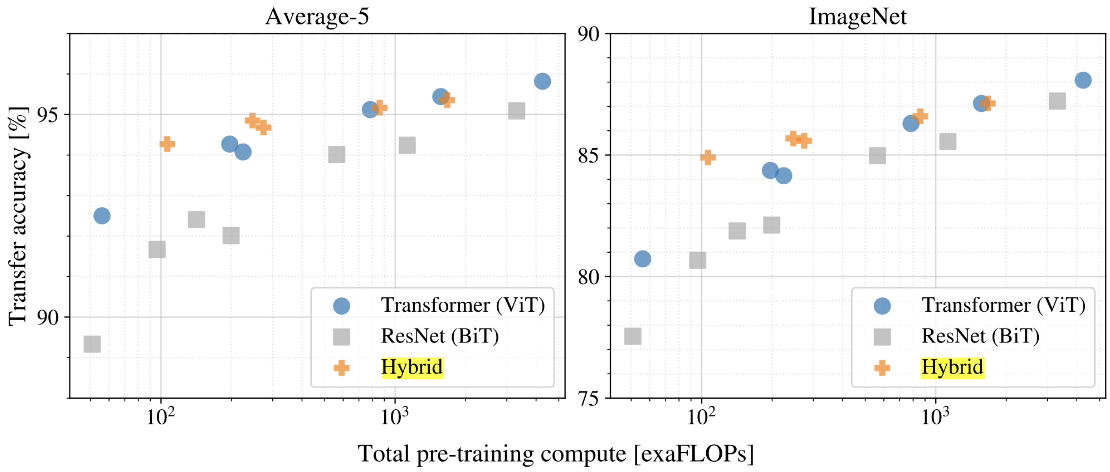


Figure 10.12: Performance of different architectures, including vision transformers, ResNets, and hybrids (Dosovitskiy et al., 2020).

- **Children Book Images**. Adukia et al. (2023) use CNN and transfer learning to classify the race, gender and age of the images in children's books. The dataset they use is the award winning children's books from 1920s. Word counts and named entity recognition (NER) are applied for text analysis, revealing the underpresentation of Black and Latino in influential children's books, though the representation of Black increases overtime. Empirical analyses are also provided, which investigate the economic behaviors underlying the representations discovered by ML in Children's books.

- **Price Trends Prediction from Charts**. Jiang et al. (2023) use CNN to directly extract new price patterns from stock-price charts and yield more accurate return predictions and more profitable investment strategies.

- **Poverty Detection with Satellite Images**. Jean et al. (2016) use satellite image data and ML to estimate consumption expenditure and asset wealth in Africa. Transfer learning is adopted, where CNN is pretrained on ImageNet and finetuned on small-scale labeled data. The CNN trained to represent the satellite images explains 75% of economic outcome variations.

- **Restaurant Survival Prediction**. Zhang and Luo (2023) use CNN to extract 18 features from consumer generated photos on Yelp, and XGBT to predict restaurant survival. It is found that photos are more informative than reviews to predict restaurant survival.

- **AirBnb Image Quality**. Zhang et al. (2022) find that properties with verified images on AirBnb have 8.98% higher occupancy. They use CNN to quantify the quality of images on AirBnb in 12 human-interpretable image attributes. It is discovered that verified photos are indeed better with respect to these image attributes.

- **AirBnb Photo Layout**. Li et al. (2023) use ResNet-50 to predict the human-labeled scores of the AirBnb photos. They use a ”structural model” to describe how photo quality scores impact consumer rent behaviors, and investigate the ”optimal” counterfactual photo layout strategy.

10.5 Demonstrations

This document also provides links to the CoLab Notebook demos of deep learning models, including LeNet & AlexNet, VGG, ResNet and ViT.

11

Deep-Learning-Based Object Detection and Video Analysis

——-**Scribed by Qiyu Dai and Yifan Ren**

11.1 Outline

This document aims to give an overview of the techniques related to image pre-processing and object detection. The organization is as follows

- In section 11.2, we briefly introduce the idea of data augmentation, which creates additional training data points by modifying the existing data.
- In section 11.3, we introduce a sequence of algorithms that detect objects within an image. Specifically, we will first discuss detecting a single object in an image. Then, we proceed to multi-object detection.
- In section 11.4, we show how to analyze videos building upon the idea of image analysis.

Notice that throughout this notes, the neural networks are trained using backpropagation, which we introduced at the beginning of this course.

11.2 Data Augmentation

Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data. It includes making minor changes to the dataset or using deep learning to generate new data points. The objective of data augmentation includes:

- To prevent models from overfitting.
- To Reduce the operational cost of labeling and cleaning the raw dataset.
- To generate a larger training set when the initial training set is too small.

The image data augmentation can take various forms, including

- Geometric transformations: randomly flip, crop, rotate, stretch, and zoom images.
- Color space transformations: randomly change RGB color channels, contrast, and brightness.

- Kernel filters: randomly change the sharpness or blurring of the image.
- Random erasing: delete some part of the initial image.
- Mixing images: blending and mixing multiple images.

11.3 Object Detection

So far, our discussion has been focused on classification only. In this section, we aim to deal with the object detection task, which is classifying and localizing all instances of target object classes in an image. In other words, for each detected object, the output of the neural network should include

- its category label, selected from a fixed and known set of classes with size c.
- its location in the image, characterized by a bounding box. A bounding box is featured by the center (x, y), the width w, and the height h.

Correspondingly, during the training process, the loss function consists of two parts

- classification loss: which is the softmax loss
- localization loss; which is the L2 difference between the output bounding box and the ground truth.

In addition, we have two evaluation metrics for the object detection task when testing

- classification: evaluated by accuracy.
- localization: evaluated by Intersection over Union (IoU), which is defined as

$$\frac{\text{intersection of the predicted object area and actual object area}}{\text{intersection of the predicted object area and actual object area}}$$

To give you a brief sense of the IoU measure, the localization has a decent precision if IoU is greater than 0.5. Besides, if IoU ≥ 0.7, the localization is “pretty good"; if IoU ≥ 0.9, then it is “almost perfect".

11.3.1 Single Object Detection

We start with single-object detection. The architecture of a neural network adapted to classification and localization is displayed in Figure 11.1. We briefly introduce the training process here. The image is first passed through a convolutional neural network, which results in a vector representation of the image. This part of the structure, referred to as the “backbone network", is often pre-trained on the ImageNet dataset and can be fine-tuned for the current problem. Then, starting from the vector representation, we build two branches. One trains a fully connected layer to calculate the score associated with each classification category, using the softmax loss. This is the same as classification models. In addition, the second branch, which is a fully connected layer trained to compute four bounding box coordinates, is constructed. We treat the localization part as a regression problem and thereby evaluate the loss by the $L2$ difference between our output set and the correct bounding box coordinates. Lastly, to apply the gradient descent to solve the parameters of the neural network, we have to define a scalar loss instead of dealing with two separate loss functions. Therefore, the regression loss and the classification loss are integrated by a weighted sum, which is called “multitask loss".

To sum up, the above approach is equivalent to modifying the classification CNN by attaching an additional fully connected layer to predict the object box. It is straightforward, and works well when knowing that we only have one object to detect.

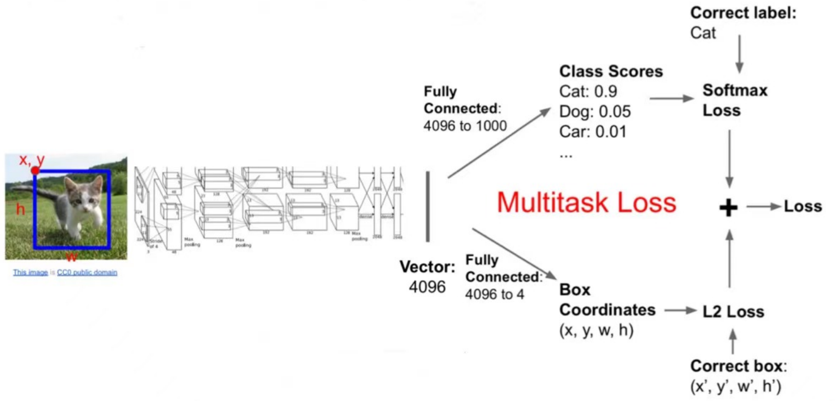


Figure 11.1: Single Object Detection

11.3.2 Multiple Objects Detection

Though it is reasonable to use the method in section 11.3.1 to tackle the single object detection task, an image can contain multiple objects in practice. As a result, two challenges are present:

- Firstly, we have no information about the exact number of objects contained in an image.
- Secondly, even knowing how many items are in each image, this number can vary across images. So, applying the technique in section 11.3.1 requires us to specify different numbers of outputs per image, which makes the training process difficult.

Thus motivated, we are going to introduce mechanisms that can find variable numbers of objects for each different image in the remainder of this section. In particular, we first discuss the limitations of the simple Sliding Window Approach, and then explain how Region-based CNN proposed by Girshick et al. (2014), Fast R-CNN proposed by Girshick (2015), Faster R-CNN proposed by Ren et al. (2015), and Single Stage Object Detetction.

Sliding Window Approach The first way to deal with multiple object detection is called the sliding window approach, which is illustrated in Figure 11.2. Roughly speaking, we slide a classification CNN over different regions of an image, and for each region, we classify it as one of the $c + 1$ categories, where c is the number of object categories, and 1 corresponds to the background.

However, it is impractical to employ this method for object detection, due to the huge computational burdens. Consider an image of size $H \times W$. If we fix the size of the box at $h \times w$, then there are $W - w + 1$ possible $x-$coordinates and $H - h + 1$ possible $x-$coordinates of center, which results in $(W - w + 1) \times (H - h + 1)$ different windows. Moreover, since the height and the width of the box can vary from 1 to H and 1 to W, respectively, the number of total possible boxes is given by

$$\sum_{h=1}^{H} \sum_{w=1}^{W} (W - w + 1)(H - h + 1) = \frac{H(H+1)}{2} \frac{W(W+1)}{2}$$

Evaluating this number under image size 800×600 gives around 58 million boxes. Therefore, it is impossible to identify the object category for each of them.

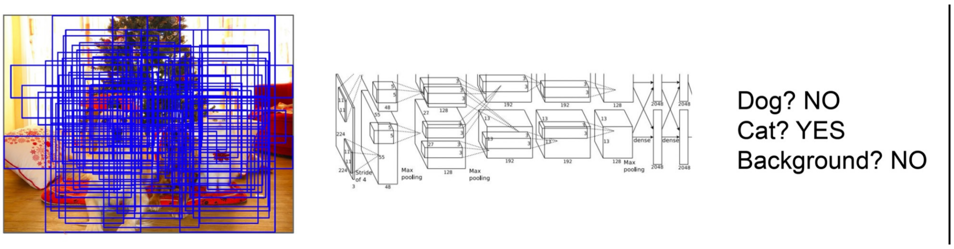


Figure 11.2: Sliding Window Approach for Multiple Objects Detection

Region-based CNN (R-CNN) To overcome the obstacle of the sliding window approach, we proceed to discuss Region-based CNN techniques in this section. Instead of evaluating the object detector on every possible region in the image, R-CNN first finds a relatively smaller set of candidate regions that have a high probability of covering all objects. One approach to generating these candidate regions is called "selective search". It works by over-segmenting the image into lots of initial regions and then merging some of them based on various similarity criteria, such as color, texture, size, and shape compatibility. Selective search gives about 2000 proposals in only a few seconds on CPU. Notice that researchers have proposed different region proposal mechanisms (See, e.g., Alexe et al. (2012), Cheng et al. (2014), Uijlings et al. (2013), Zitnick and Dollár (2014)). We are not going to dive into the details.

Building upon the idea of region proposal, we introduce the procedure of the R-CNN. It is also presented in both Figure 11.3 and 13. Speficially, we first run the region proposal method like selective search to generate a sequence of Regions of Interest (RoI). We then warp each region to a fixed size. Next, we forward each region proposal independently through a convolutional neural network, after which classification scores and the bounding box for each detected object will be output for each region. Notice that different from the object detection task, where the bounding box is output from scratch, we locate the object by transforming the region proposal inputs. Let us call the bounding box output of the CNN transformation, which is represented by (t_x, t_y, t_h, t_w). Then, given the region proposal (p_x, p_y, p_h, p_w), the output box (b_x, b_y, b_h, b_w) is calculated as

$$\begin{aligned} \text{transform the center relative to box size: } & b_x = p_x + p_w t_x, b_y = p_y + p_h t_y \\ \text{log-space scale transform of width and height: } & b_w = p_w \exp(t_w), b_h = p_h \exp(t_h) \end{aligned} \tag{11.1}$$

Lastly, we will choose a subset of modified region proposals as the output. There can be lots of selection criteria, and three of them are listed here:

- Choose all bounding boxes with the classification score of the background smaller than some threshold c_b.

- Choose a K_c of bounding boxes per category.

- Take the K proposals with the lowest background score per image.

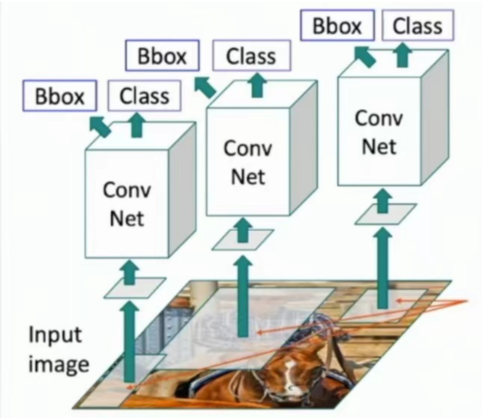


Figure 11.3: Sliding Window Approach for Multiple Objects Detection

Algorithm 13 Region-based CNN

```
Input: Single RGB Image
run the region proposal method like selective search to generate a sequence of Regions of Interest (RoI).
Resize each region to a fixed size (e.g., $224 \times 224$) and run independently through CNN to predict class scores of each category and the background, as well as the bounding box transformation.
Use classification scores to select a subset of region proposals with transformation to output.
Compare with ground truth boxes.
```

Fast R-CNN Though the RCNN can mitigate the computational burden of the sliding window approach, it still takes a considerable amount of time to complete, as we need to forward pass around 2000 regions per image independently. We can further speed up the computation by applying Fast R-CNN and Faster R-CNN.

Fast R-CNN is shown in Figure 11.4, and described as follows. Firstly, we pass the whole input image through a "backbone" network, where only convolutional layers and the fully connected layers are absent. Therefore, the output of the "backbone" network is a set of convolutional image features. Next, we run the region proposal method to obtain the RoIs from the raw image. We then project each object proposal onto the convolutional feature map and apply cropping to resize the feature map of each region. Lastly, we run a network comprising several fully connected layers of a classification CNN per region to obtain the classification scores and bounding box regression transformations.

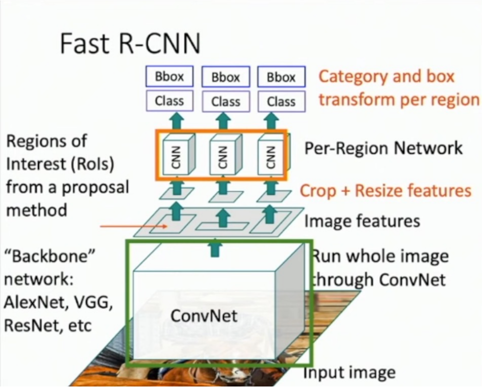

Fast R-CNN
Bbox
Class
Category and box transform per region
Regions of Interest (RoIs) from a proposal method
CNN
Per-Region Network
Crop + Resize features
Image features
"Backbone" network: AlexNet, VGG, ResNet, etc
ConvNet
Run whole image through ConvNet
Input image

Figure 11.4: Fast R-CNN

We now provide more details on the implementation of fast R-CNN. Firstly, we can define the “backbone" and per region networks by adapting any classification architectures. For example, when using AlexNet for detection, we will put the five convolutional layers on the backbone part and employ the remaining two fully connected layers for per-region classification and bounding box regression. For ResNet, the last stage is applied in the per-region network, while the rest are adopted as the “backbone". Secondly, for feature cropping, one way is to use an operator called “RoI Pool". The "RoI Pool" operator works by dividing each region proposal into a grid of roughly equal-sized cells and applying a max-pooling operation over the features in each cell, as illustrated in Figure 11.5. So, even if the region proposals are of different sizes, we can use “RoI Pool" to keep their region feature sizes identical.

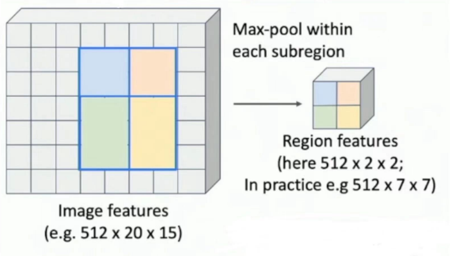

Max-pool within each subregion
Region features (here 512 x 2 x 2; In practice e.g 512 x 7 x 7)
Image features (e.g. 512 x 20 x 15)

Figure 11.5: Feature Cropping

In summary, the Fast-R-CNN is more efficient than CNN, as it reduces the computational redundancy by sharing convolutional features among the proposed regions. However, fast R-CNN still needs to spend more time computing region proposals. In fact, almost 90% of the run time is spent on generating region proposals based on heuristic algorithms like selective search. This problem is further addressed by Faster-CNN, where region proposals are instead learned with CNN as well.

Faster R-CNN Roughly speaking, Faster CNN differs from Fast CNN by inserting a Region Proposal Network (RPN) to predict proposals from features after the “backbone" network. We display the overall workflow of Faster CNN and the idea of the RPN part in Figure 11.6 and 11.7, respectively. To explain how to construct the RPN, imagine that we place a K anchor box with different sizes at each point in the feature map. Then, at each point, we predict whether the corresponding anchors contain an object or background using a convolutional layer with softmax loss. Moreover, for the boxes where objects are contained, we also predict a box transform that adjusts the anchor box to the object box. At the test stage, we sort all boxes by their score associated with containing an object and take the top around 300 ones as the region proposals.

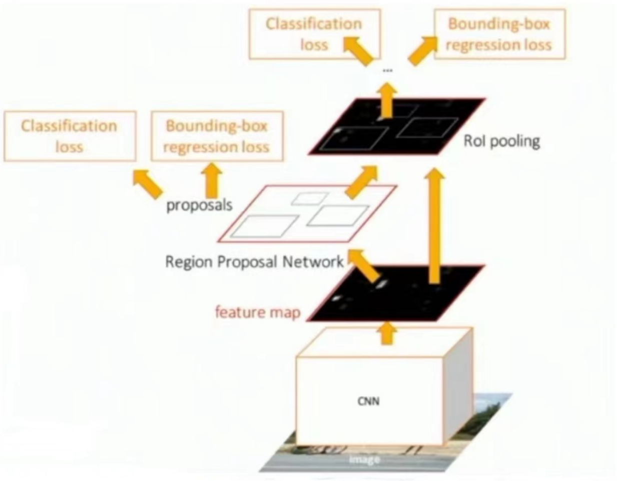


Figure 11.6: Faster RCNN

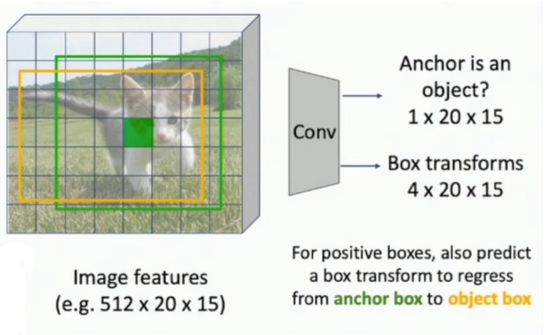


Figure 11.7: Region Proposal Network (RPN)

Overall, the Faster R-CNN is trained to minimize the sum of the following four losses, while other object detectors only have the last two. In Figure 11.8, we provide a comparison across different RCNN's. It is evident that Faster R-CNN outperforms other methods by significantly speeding up the computational process.

- RPN classification: anchor box is object/not an object
- RPN regression: predict transform from anchor box to proposal box
- Object classification: classify proposals as background/object class
- Object regression: predict transform from proposal box to object box

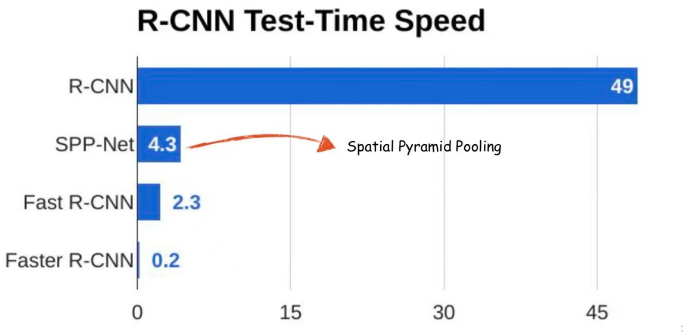


Figure 11.8: Computational Time of Different RCNN

Single Stage Object Detection Essentially, the Faster R-CNN is a two-stage object detector. In the first stage, it runs the backbone and region proposal networks once per image. In the second stage, it conducts feature cropping, object class prediction, and bounding box transformation per region. In fact, we can finish those tasks within a single stage.

Recall that the RPN in Faster CNN classifies each anchor as binary output (object or background). In contrast, the procedure of a single-stage detector is

- Run backbone CNN to get features from the input image.
- At each point in the feature map, construct K anchor boxes.
- Classify each object as one of the C categories or background.

Therefore, we do not need to bother with the second stage of Faster RCNN. Equivalently, we train the loss function, which is a summation of the following four parts

- Coordinate loss: minimize the difference between x, y, w, h prediction and x, y, w, h ground truth. This part is considered only if the object exists in the grid box.

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (y_i - \hat{y}_i)^2]$$

- Confidence loss: loss based on confidence. This part is considered only if the object exists in the grid box.

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2$$

- No object loss: based on the confidence if there is no object

$$\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2$$

- Class loss: the loss between the true class of the object in the box

$$\sum_{i=1}^{S^2} \mathbb{1}_{i}^{obj} \sum_{C \sim \text{classes}} (p_i(c) - \hat{p}_i(c))$$

11.3.3 Beyond Boxes: Semantic Segmentation

Now that we have talked about how to classify each region of an image in the context of object detection. In this part, we will introduce semantic segmentation, which labels each pixel in the image with a category label. Notice that the semantic segmentation does not differentiate instances. In other words, if we recognize two objects from the same category next to each other in the image, then semantic segmentation does not distinguish these two instances.

Given that the output of semantic segmentation has the same width and height as the input, a straightforward approach to this task is to pass the input through a set of convolutional layers. However, the convolution on high-resolution images is computationally expensive (recall that ResNet stem aggressively downsamples). To tackle this challenge, we first introduce the idea of learnable upsampling, which, intuitively, is the opposite of downsampling. One simple upsampling method is called the bed of nails, as shown in Figure 11.9. It lets the output be filled with zeros and then copies the feature vector for each region in the input into the upper left corner of each corresponding output region. Another method called nearest neighbor is also included.

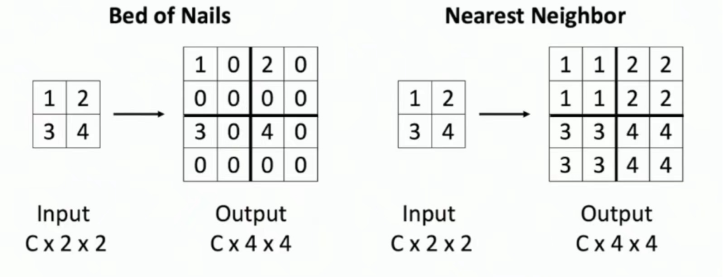


Figure 11.9: Upsampling

To incorporate some learnable parameters, we can use the transposed convolution. We use an example to illustrate the main idea in Figure 11.10. Consider that the input and the output of upsampling are 2×2 and 4×4, respectively. The filter size is 3×3, and the stride is 2. We first multiply the 3×3 filter by the first input element (a scalar). We then copy the scaled version of the filter to output. Next, we move two positions (the stride size) in the output when moving one position in the input and fill in weighting filters by a new input element. For the positions where outputs from two weighted filters are present, we will take the summation. Looping this process over each input element gives a 5×5 output. Therefore, we can trim one pixel from the top and left to obtain a 4×4 output.

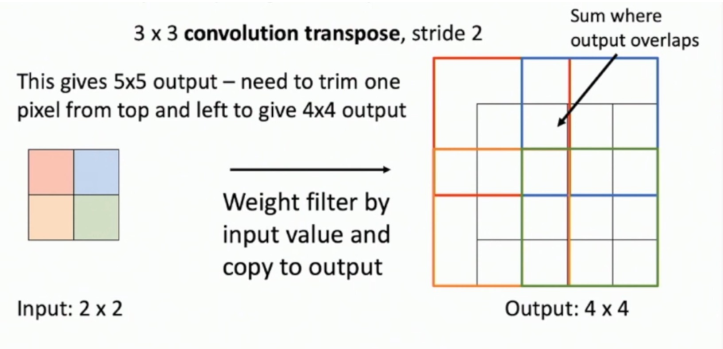

3 x 3 convolution transpose, stride 2
Sum where output overlaps
This gives 5x5 output – need to trim one pixel from top and left to give 4x4 output
Weight filter by input value and copy to output
Input: 2 x 2
Output: 4 x 4

Figure 11.10: Convolution Transpose

Using the upsampling idea, semantic segmentation works by designing the network as a bunch of convolutional layers, with downsampling and upsampling inside the network, as shown in Figure 11.11. Notice that the loss function is given by per-pixel cross-entropy.

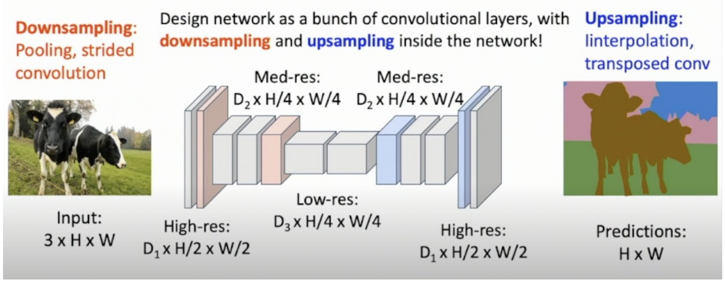

Downsampling: Pooling, strided convolution
Design network as a bunch of convolutional layers, with downsampling and upsampling inside the network!
Upsampling: linterpolation, transposed conv
Med-res: D_2 x H/4 x W/4
Med-res: D_2 x H/4 x W/4
Low-res: D_3 x H/4 x W/4
Input: 3 x H x W
High-res: D_1 x H/2 x W/2
High-res: D_1 x H/2 x W/2
Predictions: H x W

Figure 11.11: Semantic Segmentation

11.4 Video Analysis

So far, our content is all about dealing with image data. Starting from this section, we will dive into video analysis, with a focus on video classification. Recall that in image classification, we recognize object categories, such as dog, cat, fish, etc. So analogously, in videos, we can classify the actions, like swimming, jumping, running, etc.

A video is a sequence of images that unfold over time. Translating it into deep learning language, the input will be 4D tensors of size $T \times 3 \times H \times W$, where T is the time, 3 is the channel dimension, and H and W are spatial dimensions. However, unlike images, videos take up much storage. Videos usually have 30 frames per second (fps). Consequently, the size of an uncompressed video with 640×480 and 1920×1080 resolution can be around 1.5GB and 10GB per minute, respectively. Therefore, it is almost impossible to build deep neural networks that are able to process the raw video. The solution to this issue is that we train on short clips of video, which usually has three to five seconds in length. In addition, we lower the spatial resolution of each frame by subsampling. As a comparison, a clip with 3.2 seconds and $H = W = 112$ (5 fps) only uses 588 KB storage. At the testing stage, we run the model on different slips and take the average of predictions.

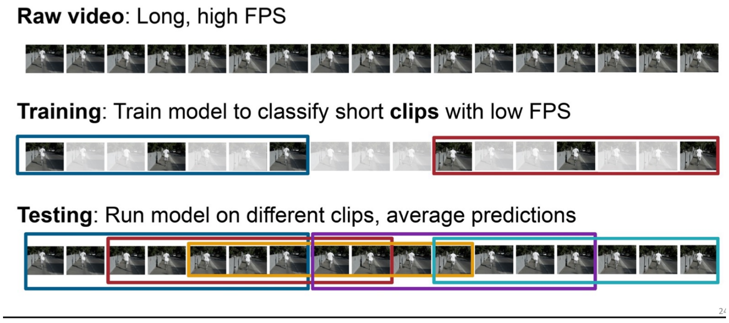

Raw video: Long, high FPS
Training: Train model to classify short **clips** with low FPS
Testing: Run model on different clips, average predictions

Figure 11.12: Clips

11.4.1 Baseline: Single Frame CNN

The first video classification model we will introduce is Single Frame CNN. The underlying idea is simple - we train regular 2D CNN on each frame of the video independently, and average the predicted probabilities at the test stage to classify videos. Although this approach ignores the temporal structures in video data, its performance is good in many scenarios. As a result, Single Frame CNN is a strong baseline for video classification.

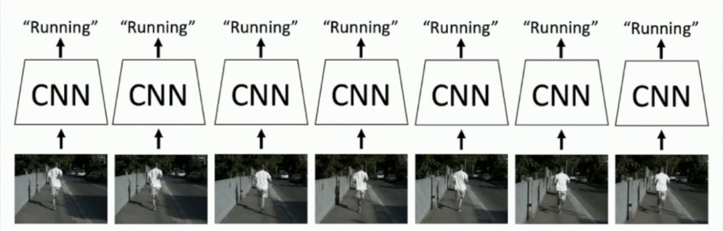
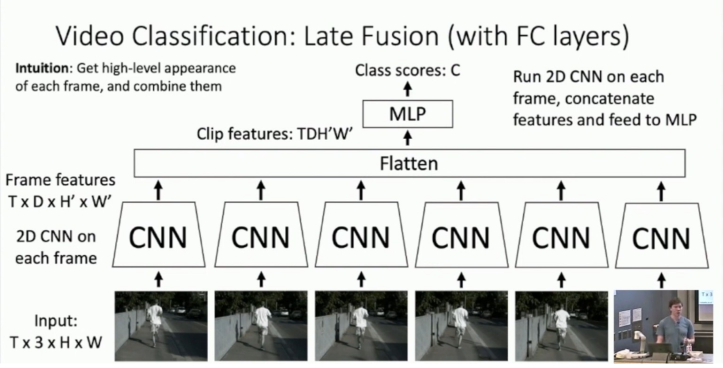


Figure 11.13: Single Frame CNN

11.4.2 Late Fusion and Early Fusion

We show the procedure of Late Fusion in Figure 11.14. Upon running CNN on each frame independently, we have obtained a set of frame-level features. Then, we flatten the frame features into a vector. Next, we apply another fully connected layer set to convert the flattened features into the final class. This process is also trained with cross-entropy loss, as we did before. Notice that we call this architecture Late Fusion because we fuse the temporal information of clips at a late phase of the classification pipeline while using per-frame modeling with CNN at earlier stages.

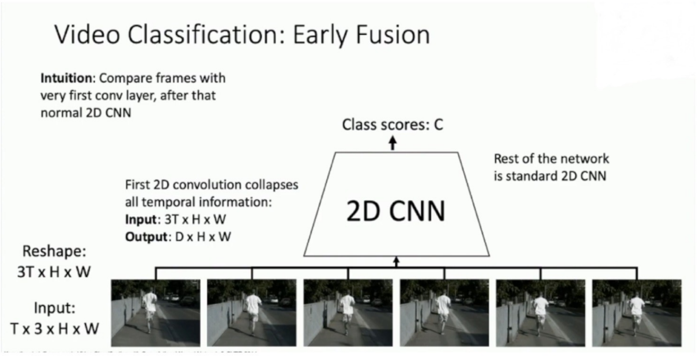

Video Classification: Late Fusion (with FC layers)

Intuition: Get high-level appearance of each frame, and combine them

Run 2D CNN on each frame, concatenate features and feed to MLP

Figure 11.14: Late Fusion

In contrast, Early Fusion fuses all the temporal information of the network at the beginning of the architecture, using the first layer of CNN. More concretely, we reshape the four-dimensional input tensor with size $T \times 3 \times H \times W$ to a three-dimensional tensor with size $3T \times H \times W$. Next, we fuse all the temporal information using a single two-dimensional convolution operator, where the output is given by a $D \times H \times W$ tensor. Now that we have collapsed all the temporal information and obtained a three-dimensional tensor to work with. Therefore, we can design the rest of the architecture as any standard two-dimensional convolutional neural network.



Video Classification: Early Fusion

Intuition: Compare frames with very first conv layer, after that normal 2D CNN

Class scores: C

Rest of the network is standard 2D CNN

First 2D convolution collapses all temporal information:
Input: 3T x H x W
Output: D x H x W

2D CNN

Reshape:
3T x H x W

Input:
T x 3 x H x W

Figure 11.15: Early Fusion

11.4.3 3D CNN

The last model we are going to discuss is 3D CNN. Intuitively, it differs from the Early Fusion model by using 3D convolution versions and pooling to slowly fuse temporal information throughout the network. That is, in each layer of the 3D CNN, we will maintain four-dimensional tensors, with one channel dimension, one temporal dimension, and two spatial dimensions. In each layer inside the CNN, we use the three-dimensional analogs of convolution and the three-dimensional analogs of pooling.

Firue 11.16 presents a comparison of the classification accuracy of various video classification methods on the Sports 1M dataset. 3D CNN gives the best performance, and the second one is the Single Frame CNN. This is consistent with our previous statement that Single Frame CNN always has robust performance, which should be tried at first for any classification task.

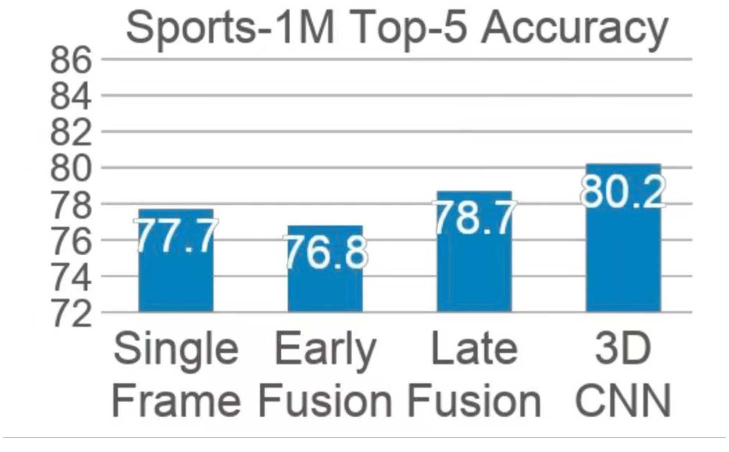


Figure 11.16: Performance of Video Clasisification Models oncthe Sports 1M Dataset

12

Unsupervised Learning (1) - Clustering, Topic Modeling, Variational Auto-Encoder

——Scribed by Qinlu,Hu and Yilin,Shi

12.1 Outline

The organization of this section is as follows:

- In section 12.2, we introduce the different clustering methods, including K-means, latent variable models, and gaussian mixture models.

- In section 12.3, we introduce the topic modeling method.

- In section 12.4, we introduce the variational Auto-Encoder

12.2 Clustering

Data: $\{X_i \in \mathbb{R} : i = 1, 2, 3, \ldots, n\}$

Output: A (non-overlapping) partition of the dataset $C_1, C_2, C_3, \ldots, C_k$. As shown in Figure 12.2, we cluster customers according to their weight and height

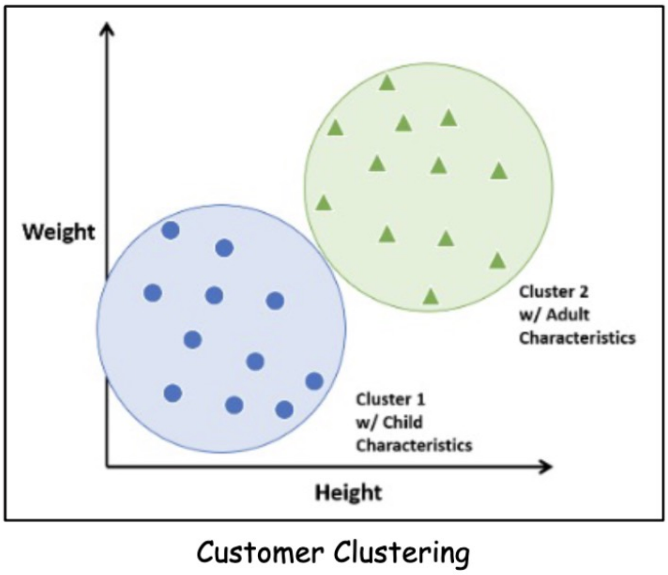


Figure 12.1: Customer cluster

12.2.1 K-Means Clustering

Setting:

- Data $= \{X_i \in \mathbb{R}^d : i = 1, 2, 3, \ldots, n\}$
- Output: A (non-overlapping) partition of the dataset $C_1, C_2, C_3, \ldots, C_k$
- Equivalent output for K-means, cluster centers: $y_j = \sum_{i \in C_j} X_i / |C_j|$ for $j = 1, 2, \ldots, k$

The detailed process can be seen as Algorithm 14.

Convergence of K-Means:

- **Criteria of k-means (sometimes called inertia):** The squared distance between each data observation and the center of the cluster it belongs to.

$$L(\mu) = \sum_{i=1}^{n} \min_{j \in \{1,\ldots,k\}} \|x_i - \mu_j\|^2.$$

- **Each maximization step will always (weakly) reduce the criteria.**

$$L(\mu', z') - L(\mu, z) = \sum_{i=1}^{n} (\|x_i - \mu'_{z_i}\|^2 - \|x_i - \mu_{z_i}\|^2) < 0.$$

- **Each expectation step will also always (weakly) reduce the criteria.**

$$L(\mu', z') - L(\mu, z) \leq \sum_{j=1}^{k} \left(\sum_{i:z'_i=j} \|x_i - \mu_j\|^2 \right) - \sum_{j=1}^{k} \left(\sum_{i:z_i=j} \|x_i - \mu_j\|^2 \right) \leq 0.$$

- **Therefore, k-means will always converge.**

Algorithm 14 K-Means Clustering (Lloyd's Algorithm)

```
Require: Data vectors $\{x_n\}_{n=1}^{N}$, number of clusters $K$
Initialize all of the responsibilities.
for $n \leftarrow 1 \ldots N$ do
    $r_n \leftarrow [0, 0, \ldots, 0]$
    $k' \leftarrow \text{RandomInteger}(1, K)$
    $r_{nk'} \leftarrow 1$
end for
repeat
    for $k \leftarrow 1 \ldots K$ do
        $N_k \leftarrow \sum_{n=1}^{N} r_{nk}$
        $\mu_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} x_n$
    end for
    for $n \leftarrow 1 \ldots N$ do
        $r_n \leftarrow [0, 0, \ldots, 0]$
        $k' \leftarrow \arg\min_k \|x_n - \mu_k\|^2$
        $r_{nk'} \leftarrow 1$
    end for
until none of the $r_n$ change
return assignments $\{r_n\}_{n=1}^{N}$ for each datum, and cluster means $\{\mu_k\}_{k=1}^{K}$
```

12.2.2 Latent Variable Models

Idea: The data generating process may have some low-dimensional hidden representations which could be automatically identified by latent variable models.

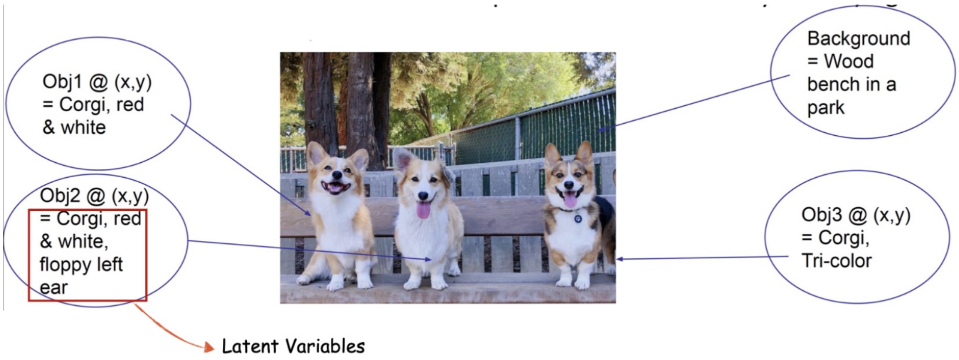

Obj1 @ (x,y) = Corgi, red & white
Obj2 @ (x,y) = Corgi, red & white, floppy left ear
Latent Variables
Background = Wood bench in a park
Obj3 @ (x,y) = Corgi, Tri-color

Figure 12.2: Latent variable in corgi figure.

Steps:

1. Generate the latent variable z from prior distribution $p(z)$.

2. Estimate θ and generate x conditioned on z from the distribution $p(x|z)$.

3. Update the distribution $P(z)$ based on $p(x|z)$ and data.

4. Repeat Steps 1 to 3 until we cannot find a better $P(z)$ to generate z.

12.2.3 Gaussian Mixture Models (GMM)

Idea:

- A generative model for data clustering.
- Data assumed generated from a mixture of K Gaussians.
- Let $0 \leq \pi_k \leq 1$ denote the "mixing weight" of the k-th Gaussian. It means:
- π_k is the fraction of points generated from the k-th Gaussian.
- $\pi_k = p(z_n = k)$ is the prior probability of x_n belonging to the k-th Gaussian.
- Let $\pi = (\pi_1, \pi_2, \ldots, \pi_K)$ denote the vector of mixing weights of K Gaussians. This is a probability vector and sums to 1, i.e., $\sum_{k=1}^{K} \pi_k = 1$.
- Notation $z_n = k$ is equivalent to a size K one-hot vector z_n where:

$$z_n = [0 \ldots 0\ 1\ 0 \ldots 0] \quad \text{(all zeros except the } k\text{-th bit, i.e., } z_{nk} = 1)$$

GMM Estimation/EM algorithm: ***(EM algorithm additional notes)***

$$(\pi_1, \pi_2, \pi_3, \ldots \pi_k)$$
$$\sum_{i=1}^{k} \pi_k = 1$$

E-step:
Update the posterior $\Pr(Z_n = k) \triangleq Y_{nk}$
$Y_{nk} \propto$ prior $\times$ likelihood
where prior is π_k and likelihood is $\quad N(X_n \mid \mu_k, \Sigma_k)$
Normalization: $\sum_{k=1}^{K} Y_{nk} = 1$
(2) M-step.

$$L_{k(\mu_k, \Sigma_k)} = \sum_{n=1}^{N} Y_{nk} \cdot N(x_n \mid \mu_k, \Sigma_k)$$

Let the first-order condition equal to zero:

$$\mu_k = \frac{\sum_{n=1}^{N} Y_{nk} x_n}{N_k} \quad N_k = \sum_{n=1}^{N} Y_{nk}$$
$$\Sigma_k = \frac{\sum_{n=1}^{N} Y_{nk}(x_n - \mu_n)(x_n - \mu_n)^T}{N_k}$$
$$L^\pi = \sum_{k=1}^{K} \sum_{n=1}^{N} Y_{nk} \log(\pi_k) + \lambda \left(1 - \sum_{k=1}^{K} \pi_k\right)$$
$$\partial u_k L^\pi = 0 \Rightarrow \pi_k = \frac{N_k}{N}$$

Repeat the E-step and M-step until convergence.

Application: Similarly we have used the latent class model to capture heterogeneity in many structural estimations EM-type of algorithms are the standard approach to estimate such models.

12.3 Topic Modeling

Topic modeling: An unsupervised way of simultaneously (a) finding topics from a set of documents and (b) classify these documents into these topics.

Input: A bunch of documents without labels.

Output: Topics and representation of topics by words; Classification of documents into topics.

12.3.1 Latent Dirichlet Allocation (LDA)

A latent (generative) model that can generate new data instances using latent variables.

Topic Modeling Generative Process: Another way to visualize the generative model of LDA: Marginal distribution of the word sequence w in each document. ***(Topic modeling process notes) (Topic modeling variational inference notes)***

(1) k topics. $(\alpha_1, \alpha_2, \cdots \alpha_k)$
Doc 1: $w_1, w_2, \cdots w_n$
Doc 2 : $w_1, w_2, \cdots w_n$
...
(2) Vocabulary $\{1, 2, \ldots V\}$
(3) $\beta_{N|XK} = \begin{pmatrix} \beta_{11}, & \cdots & \beta_{1K} \\ \beta_{N1}, & \cdots & \beta_{NK} \end{pmatrix}$ $\beta_{ij} = \Pr(w = i \mid Z = j)$.
Number of document: N $\sim$ Poisson
*. For each document, $\theta \sim$ Dirichlet (d), summation of $\theta = 1$
*: $Z_n \sim$ Multinomial(θ)

- $w_n \sim \beta = \Pr\left(w_n = i \mid z_n = j\right), i \in \{1, 2, \cdots v\}, j \in \{1, 2, \cdots k\}$

$P(\theta, z \mid \omega, \alpha, \beta)$

$$q\left(\theta, z \mid Y \cdot \phi_n\right) = q(\theta \mid Y) \cdot \prod_{i=1}^{n} q\left(z_n \mid \phi_n\right)$$

KL distribution: $KL(q \| p) = -\sum_i p(i) \left[\log \frac{p(i)}{q(i)}\right] \geqslant 0$.

$$KL(q \| p) = 0, \text{ if } p = q$$

$KL(q \| p) \neq KL(p \| q)$
$q(\theta \mid Y) \sim$ Dirichlet(Y)
$q\left(z_n \mid \phi_n\right) \sim$ Dirichlef $\left(\phi_n\right)$

12.3.2 Other Topic Modeling Methods

Methods: There are more topic modelling methods:

- Hierarchical topic modelling
- Dynamic topic modelling
- Fat-tail topic modelling.

Related Business Research: Using LDA

12.4 Variational Auto-Encoder

Data: $\{X_i \in \mathbb{R} : i = 1, 2, 3, \ldots, n\}$

Output: an encoder and decoder mapping as representation learning for the data.

Loss: Reconstruction noise + regularization

Notes: Autoencoder is not generative but VAE is generative

13

Unsupervised Learning (2) - Diffusion Models

——**Scribed by Zhenkang Peng**

13.1 Introduction

The purpose of this document is to give a crash course on diffusion models. We will introduce:

- The principles and processes of the diffusion model;
- The diffusion Model's application in text-to-image generation;
- The potential applications of diffusion models in Biz/Econ research;

Specifically, the outline of this document is as follows:

- In Section 13.2, we introduce the basic framework for denoised diffusion probabilistic models.
- In Section 13.3, we introduce all kinds of diffusion models which are utilized in text-to-image generation.
- In Section 13.4, we briefly introduce the potential applications in business and economic research.

In this lecture, we will use $\mathbb{R}^p$ to denote the $p-$dimensional real Euclidean Space, $\mathbb{P}[A]$ to denote the probability of an event A, and $\mathbb{E}[X]$ to denote the expectation of a random variable X. The conditional probability and expectation are denoted as $\mathbb{P}[\cdot|\cdot]$ and $\mathbb{E}[\cdot|\cdot]$, respectively. We denote $|A|$ as the cardinality of set A.

Heads Up

Starting from the very beginning, we will inevitably use a lot of **mathematical notations**, this is because:

- Mathematics enables us to represent some concepts, ideas, and implementations neatly, succinctly, and elegantly, which would otherwise be impossible or very cumbersome.

- The mathematical notations introduced in this course are the standard language in the ML/AI community. Familiarizing ourselves with them is important for us to communicate with other professionals in this community.

Importantly, I would emphasize that, although the mathematical notations look intimidating, you will find that they are so natural and helpful once you understand the logic behind them.

13.2 Denoised Diffusion Probabilistic Models

Diffusion models, are characterized by step-by-step updates, wherein the combination of all steps forms the encoder-decoder structure, and the transition between states is facilitated by a denoiser. As illustrated in Figure 13.1, there are two sequences. The first sequence involves adding Gaussian noise gradually to an original image $\boldsymbol{x}_0 \in \mathbb{R}^d$ until the image becomes unrecognizable, which will be elaborated on in Section 13.2.1. The second sequence focuses on extracting the noise to restore the original image, with detailed processes and algorithms discussed in Section 13.2.2.

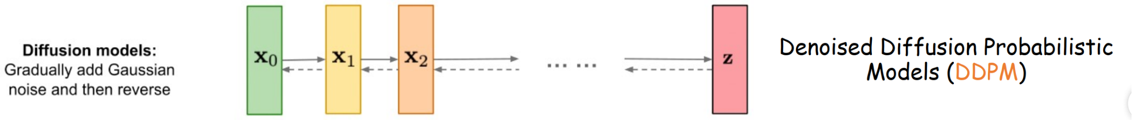

Diffusion models:
Gradually add Gaussian noise and then reverse
$\mathbf{x}_0$
$\mathbf{x}_1$
$\mathbf{x}_2$
……
$\mathbf{z}$
Denoised Diffusion Probabilistic Models (DDPM)

Figure 13.1: The whole process in the diffusion models.

13.2.1 Forward Diffusion Process

The gradual addition of Gaussian noise is termed the forward diffusion process. Illustrated in Figure 13.2, if we consider T steps starting from an original cat picture, Gaussian noise is incrementally added based on the previous step's output until reaching step T

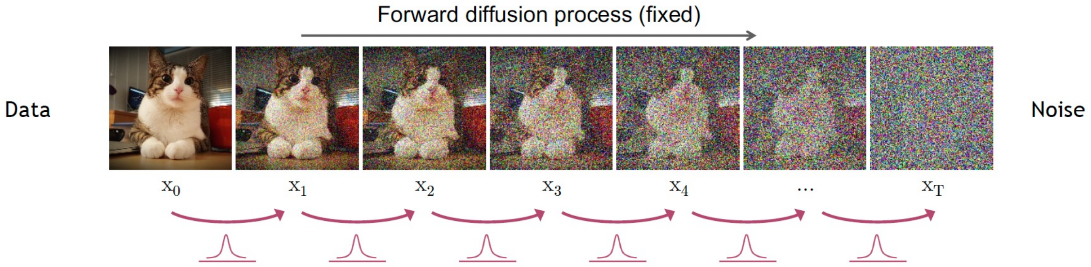


Figure 13.2: Forward diffusion process.

Given the output of the previous step $\boldsymbol{x}_{t-1}$, The formal mathematical formulation about the distribution with respect to $\boldsymbol{x}_t$ can be represented as:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t \boldsymbol{I}), \tag{13.1}$$

where $\boldsymbol{x}_t = \sqrt{1-\beta_t}\boldsymbol{x}_{t-1} + \sqrt{\beta_t}\boldsymbol{\varepsilon}_t$ and $\boldsymbol{\varepsilon}_t$ is $d-$dimensional standard normal independent of anything. β_t represents the degree of noise added at step t. For example, when β_t equals 1, it means that the output at step t is fully randomized. On the other hand, when β_t equals 0, no noise is added at step t. We define $\alpha_t = 1 - \beta_t$ The joint mathematical formulation from step 1 to T can be written as:

$$q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}). \tag{13.2}$$

In the above analysis, we discussed step-by-step addition of noise starting from the original image $\boldsymbol{x}_0$ and provided a detailed derivation of the distribution at each step. Next, we present a more generalized expression, as shown in Figure 13.3, for the distribution expression at any given t.

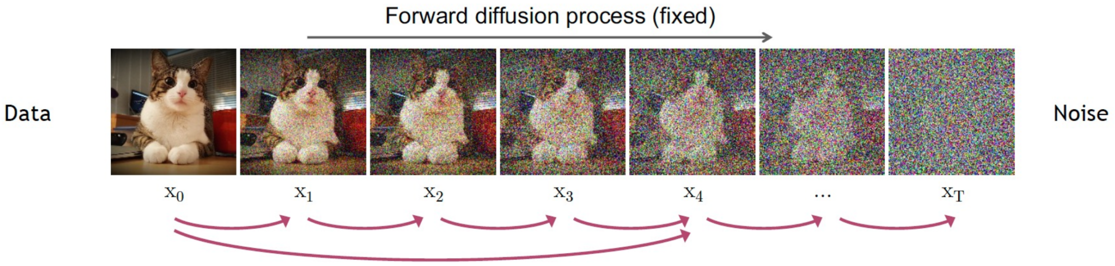


Figure 13.3: Generalised forward diffusion process.

We define $\bar{\alpha}_t = \prod_{s=1}^{t}(1 - \beta_s)$, then the distribution of $\boldsymbol{x}_t$ given the initial state $\boldsymbol{x}_0$ can be represented as:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1 - \bar{\alpha}_t)\boldsymbol{I}) \tag{13.3}$$

where the function $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ is called as diffusion kernel.

13.2.2 Denoising

In Section 13.2.1, we introduce how we get $\boldsymbol{x}_T$ from $\boldsymbol{x}_0$ by adding noise step by step. In this subsection, we will introduce how we reconstruct $\boldsymbol{x}_0$ from $\boldsymbol{x}_T$ and how we train the DDPM (Denoised Diffusion Probabilistic Models).

The process from $\boldsymbol{x}_T$ to $\boldsymbol{x}_0$ is called denoising and is like sculpture. As shown in the Figure 13.4, we gradually denoise a standard Gaussian $\boldsymbol{x}_T$ to generate the original data $\boldsymbol{x}_0$.

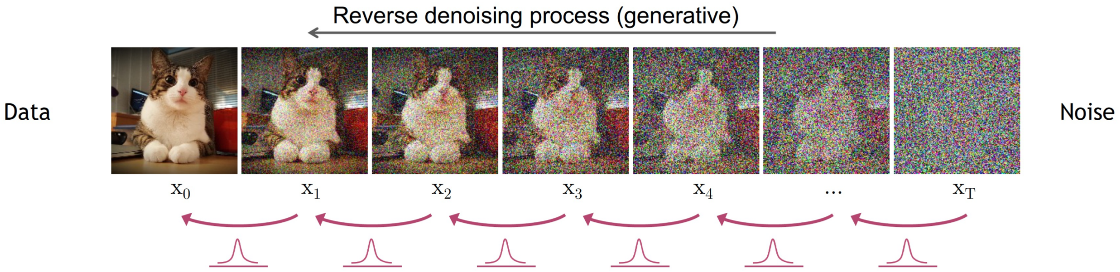


Figure 13.4: Reverse denoising process.

The process can be generalized as following steps:

- Sample $\boldsymbol{x}_T \sim \mathcal{N}(\boldsymbol{x}_T : 0, \boldsymbol{I})$.
- Iteratively sample $\boldsymbol{x}_{t-1} \sim q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ where $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \propto q(\boldsymbol{x}_{t-1})q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is the true denoising distribution

However, $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is intractable and we can use a normal distribution p to approximate it. The process can be generalized as follows:

- Sample $\boldsymbol{x}_T \sim p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T : 0, \boldsymbol{I})$.
- Iteratively sample $\boldsymbol{x}_{t-1} \sim p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1}; \mu_\theta(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I})$ where $\mu_\theta(\boldsymbol{x}_t, t)$ denotes the trainable network, such as U-net, Denoising Autoencoder. $\sigma_t^2 = \frac{(1-\alpha_t)(1-\alpha_{t-1})}{1-\alpha_t}$.

Based on the above analysis, we can know that The effectiveness of recovering from $\boldsymbol{x}_T$ to $\boldsymbol{x}_0$ depends on the degree of learning of the $p_\theta(\cdot)$ function. Therefore, we need to consider the size of $log(p_\theta(\boldsymbol{x}_0))$, and we obtain a lower bound for the value of $log(p_\theta(\boldsymbol{x}_0))$ as:

$$\begin{aligned} ELBO = & \underbrace{\mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}[logp_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)]}_{\text{reconstruction term}} - \underbrace{D_{KL}(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p(\boldsymbol{x}_T))}_{\text{prior matching term}} \\ & - \underbrace{\sum_{t=2}^{T} \mathbb{E}_{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}[D_{KL}(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \parallel p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t))]}_{\text{denoising matching term}}, \end{aligned}$$

where,

- **Reconstruction term:** How good the neural nets $p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)$ is to recover $\boldsymbol{x}_0$ from $\boldsymbol{x}_1$ when the sample is drawn from $q(\boldsymbol{x}_1|\boldsymbol{x}_0)$.
- **Prior matching term:**How close the distribution of the final notified input is to the standard Gaussian prior.
- **Denoising matching term:**How close the denoising transition distribution $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ approximates the tractable ground-truth denoising transition step distribution $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$, which can be derived by

  $$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_{t-1}; \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\boldsymbol{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\boldsymbol{x}_0}{1-\bar{\alpha}_t}, \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{I})$$

  The DDPM Inference is based on $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$.

Thus, training the $p_\theta(\cdot)$ is similiar to train the $\hat{\varepsilon}_\theta(\boldsymbol{x}_t)$ (Diffusion models often use U-Net architectures (Ronneberger et al., 2015) with ResNet blocks and self-attention layers to represent $\hat{\varepsilon}_\theta(\boldsymbol{x}_t)$), which is a neural network that maps $\boldsymbol{x}_t$ to ε_0, where $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon_0$ with the objective to minimize the following equation:

$$ELBO_\theta = -\sum_{t=1}^{T} \mathbb{E}_{q(\boldsymbol{x}_t|\boldsymbol{x}_0)} \left[\frac{1}{2\sigma_q^2(t)} \frac{(1-\alpha_t)^2 \bar{\alpha}_{t-1}}{(1-\bar{\alpha}_t)^2} \parallel \hat{\varepsilon}_\theta(\boldsymbol{x}_t) - \varepsilon_0 \parallel^2 \right] \tag{13.4}$$

According to the loss function (13.4), we can write the detailed training algorithm for DDPM as shown in Algorithm 15.

Algorithm 15 TRAINING

```
repeat:
  $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$.
  $t \sim \text{Uniform}(\{1, \ldots, T\})$.
  $\varepsilon \sim \mathcal{N}(0, \boldsymbol{I})$.
  Take gradient descent step on $\nabla_\theta \parallel \varepsilon - \varepsilon_\theta(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon, t) \parallel^2$.
  until converged.
```

- Step 1: Randomly select a picture from the dataset.
- Step 2: Randomly select a time period t from $\{1, \ldots, T\}$.
- Step 3: Randomly generate noise and add the noise to the picture according to the equation (13.3).
- Step 4: Training the $\varepsilon_\theta(\theta)$ to recover the noise process as soon as possible.

After we get the trained neural network, we can generate the pictures by Algorithm 16.

Algorithm 16 Sampling

```
$\boldsymbol{x}_0 \sim \mathcal{N}(0, \boldsymbol{I})$.
$t \sim \text{Uniform}(1, \ldots, T)$.
for $t = T, \ldots, 1$ do
    $\boldsymbol{z} \sim \mathcal{N}(0, \boldsymbol{I})$ if $t > 1$ else $\boldsymbol{z} = \mathbf{0}$
    $\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\boldsymbol{x}_t - \frac{1-\bar{\alpha}_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{z}$
end for
return $\boldsymbol{x}_0$
```

Figure 13.5, which is the experimental results in Ho et al. (2020), shows IS (Inception Scores), FID (Fréchet Inception Distance), and NLL Test (Negative Log-Likelihood Test) on CIFAR10. With DDPM FID score of 3.17, the DDPM achieves better sample quality than most models in the literature, including class conditional models.

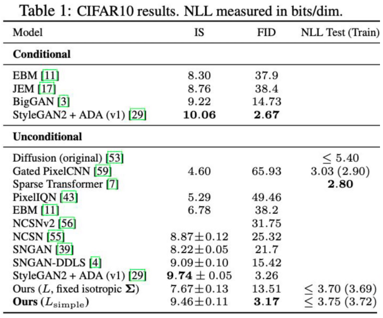


Table 1: CIFAR10 results. NLL measured in bits/dim.

Model	IS	FID	NLL Test (Train)
Conditional			
EBM [11]	8.30	37.9	
JEM [17]	8.76	38.4	
BigGAN [3]	9.22	14.73	
StyleGAN2 + ADA (v1) [29]	**10.06**	**2.67**	
Unconditional			
Diffusion (original) [53]			≤ 5.40
Gated PixelCNN [59]	4.60	65.93	3.03 (2.90)
Sparse Transformer [7]			**2.80**
PixelIQN [43]	5.29	49.46	
EBM [11]	6.78	38.2	
NCSNv2 [56]		31.75	
NCSN [55]	8.87±0.12	25.32	
SNGAN [39]	8.22±0.05	21.7	
SNGAN-DDLS [4]	9.09±0.10	15.42	
StyleGAN2 + ADA (v1) [29]	**9.74** ± 0.05	3.26	
Ours (L, fixed isotropic $\mathbf{\Sigma}$)	7.67±0.13	13.51	≤ 3.70 (3.69)
Ours (L_{simple})	9.46±0.11	**3.17**	≤ 3.75 (3.72)

Figure 13.5: Results in Ho et al. (2020).

13.3 Text-to-Image Generation

Text-to-image generation is an intriguing field of research and development that centers on creating lifelike images based on textual descriptions. Its applications span various domains, aiding artists and designers in conceptual visualization and improving accessibility for individuals with visual impairments. Illustrated in Figure 13.6, the process can be distilled into three steps. Initially, the text encoder embeds the text, following which the diffusion model, acting as the generation model, generates latent image representations. Finally, the decoder produces the image output.

In this section, we will introduce all kinds of diffusion models to the reader which will be used as generation model including latent diffusion, CLIP (Contrastive Language-Image Pretraining), DALL-E, and diffusion transformer.

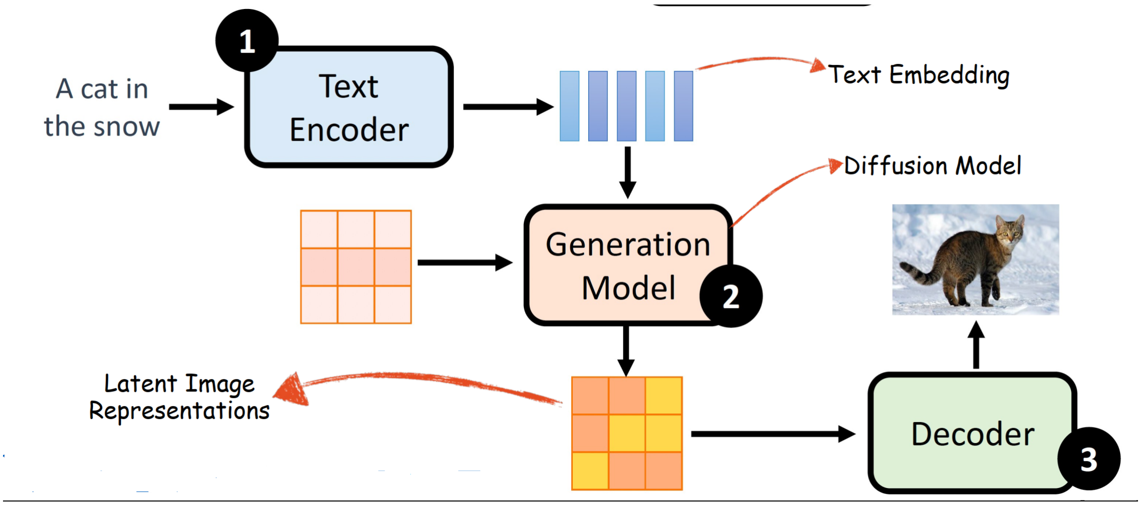


Figure 13.6: Text-to-image generation.

13.3.1 Latent Diffusion Models

The latent diffusion model, introduced by Rombach et al. (2022), is a generating images model by iterating through "denoising" data in a latent representation space and subsequently decoding the representation into complete images.

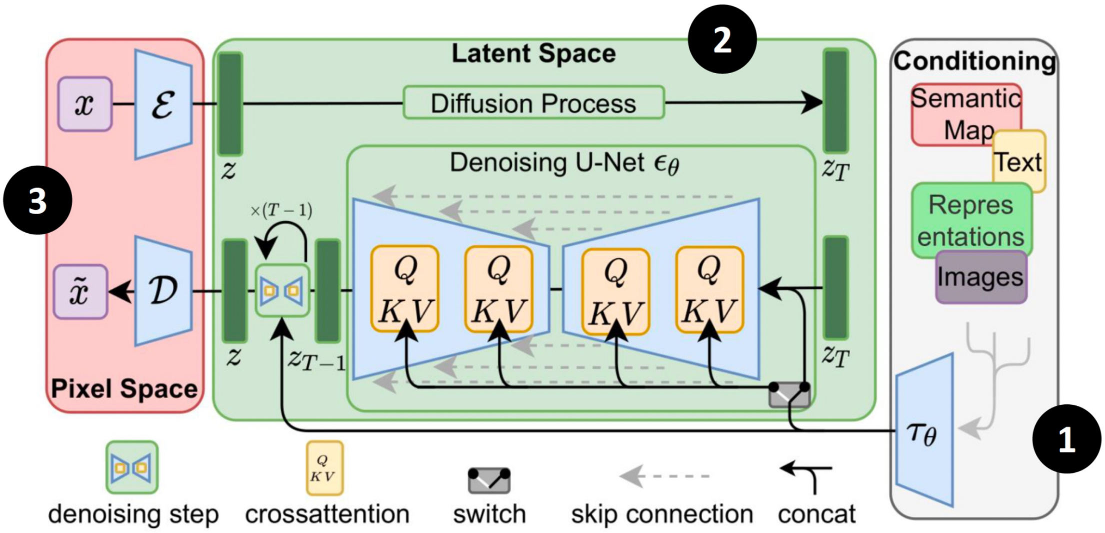


Figure 13.7: Stable/Latent diffusion model.

The overall framework of Latent Diffusion Models is illustrated in Figure 13.7. Firstly, it requires training a well-established autoencoder model, which consists of an encoder ε and a decoder $\mathcal{D}$. With this setup, we can compress images using the encoder, perform diffusion operations in the latent representation space, and finally restore them to the original pixel space using the decoder. The paper refers to this method as Perceptual Compression. Personally, I believe that this approach, which compresses high-dimensional features into a low-dimensional space and operates within that space, is versatile and can be easily extended to other domains such as text, audio, and video.

The process of diffusion operations in the latent representation space is similar to standard diffusion models, with the specific implementation of a time-conditional U-Net diffusion model. However, an important aspect highlighted in the paper is the introduction of conditioning mechanisms for diffusion operations, achieved through cross-attention, enabling multimodal training, and facilitating conditional image generation tasks.

13.3.2 Contrastive Language-Image Pre-training

Contrastive Language-Image Pre-training, proposed by Radford et al. (2021), is a cutting-edge technique that involves training a model on a large dataset containing both textual descriptions and corresponding images. The goal is to enable the model to learn rich representations of both modalities and understand the semantic relationships between them.

As shown in Figure 13.8, the process typically involves two main stages:

- **Pre-training with Contrastive Learning:** In this stage, the model learns to encode textual descriptions and images into a shared latent space where similar pairs (e.g., an image and its corresponding description) are brought closer together while dissimilar pairs are pushed apart. This is achieved through contrastive learning objectives such as InfoNCE (Noise Contrastive Estimation) or NT-Xent (Normalized Temperature-scaled Cross Entropy). The idea is to enhance the model's ability to capture meaningful connections between textual and visual information.

- **Fine-tuning for Downstream Tasks:** After pre-training, the model is fine-tuned on specific downstream tasks such as image captioning, visual question answering, or image retrieval. Fine-tuning helps the model specialize its representations for the particular task at hand, leading to improved performance. Contrastive Language-Image Pre-training has gained significant attention and success due to its ability to learn semantically meaningful representations across modalities, allowing for better understanding and generation of text-image pairs. This approach has been leveraged in various applications, including multimodal search engines, content creation tools, and assistive technologies for individuals with visual impairments.

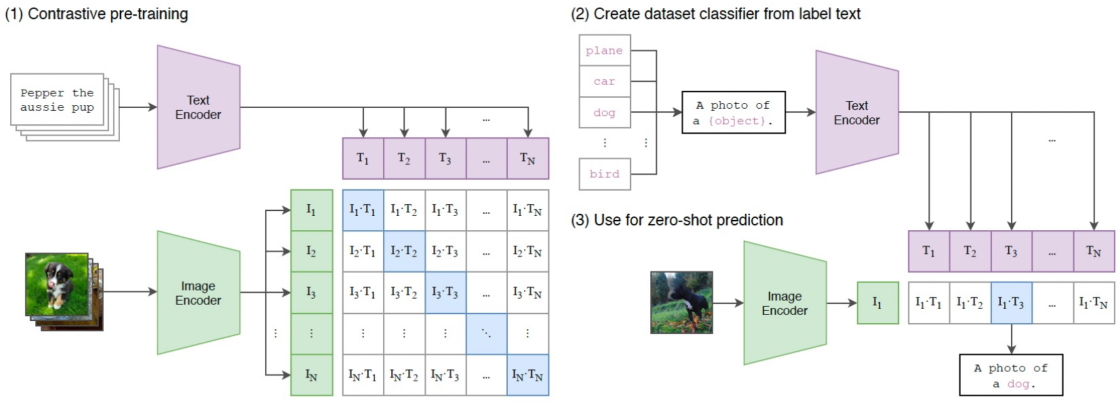

(1) Contrastive pre-training
Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ... I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$
(2) Create dataset classifier from label text
plane
car
dog
bird
A photo of a {object}.
Text Encoder
(3) Use for zero-shot prediction
Image Encoder
I_1
T_1 T_2 T_3 ... T_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
A photo of a dog.

Figure 13.8: Contrastive Language-Image Pre-training.

13.3.3 DALL-E2

As shown in Figure 13.9, DALL-E2, proposed by Ramesh et al. (2022), is an advanced version of the DALL-E (Generative Pre-trained Transformer 3 for Image Generation) model developed by

OpenAI. It is specifically designed for generating diverse and high-quality images from textual descriptions. The name "DALL-E2" is a play on words, combining "DALL-E" and "E2" (which could stand for "Enhanced" or "Extended").

Here are some key features and improvements of DALL-E2:

- **Enhanced Image Generation:** DALL-E2 builds upon the capabilities of its predecessor, DALL-E, to generate even more diverse and realistic images based on textual input. It leverages advanced deep learning techniques, including transformer-based architectures, to understand and interpret complex textual descriptions and translate them into corresponding images.

- **Improved Training:** DALL-E2 benefits from improved training methodologies, larger datasets, and refined optimization techniques. This results in better model performance, increased image diversity, and higher quality outputs compared to earlier versions.

- **Multimodal Understanding:** One of the strengths of DALL-E2 is its ability to understand and synthesize multimodal information. It can generate images that capture not only the visual aspects described in text but also abstract concepts, imaginative scenarios, and creative compositions.

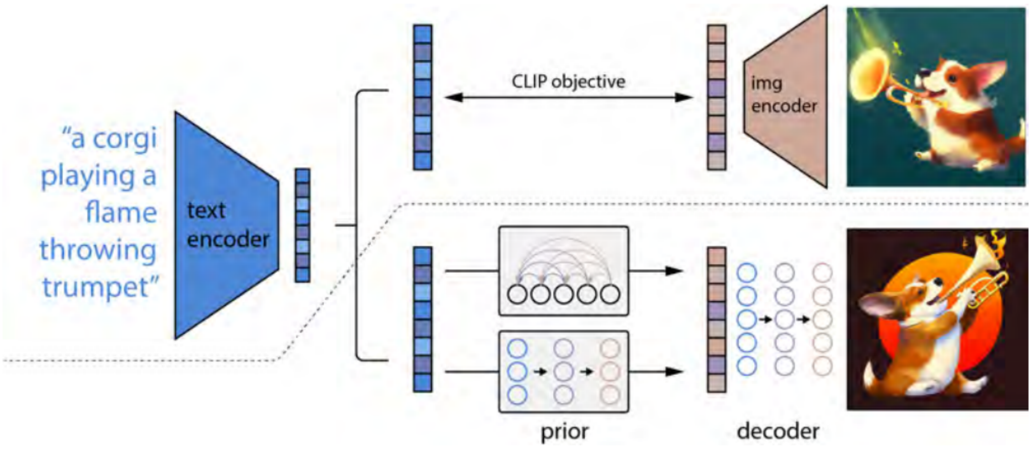


Figure 13.9: DALL-E2.

13.3.4 Diffusion Transformer

The Diffusion Transformer (DiT) is a novel architecture proposed by Peebles and Xie (2023) that combines elements from diffusion models and transformers, two powerful classes of deep learning models. This fusion aims to improve the capabilities of language understanding and generation tasks by leveraging the strengths of both approaches.

Here are the key aspects and features of the Diffusion Transformer (DiT):

- **Diffusion Model Integration:** DiT incorporates ideas from diffusion models, which are probabilistic generative models used for image and text generation tasks. Diffusion models operate by iteratively denoising data, gradually revealing the underlying structure. DiT adapts this concept to language modeling and understanding tasks within the transformer architecture.

- **Transformer Architecture:** Like traditional transformers, DiT uses self-attention mechanisms and multi-layer perceptrons (MLPs) to process input sequences. Just as shown in Figure 13.10, they replace the U-Net in latent diffusion models (LDMs) with a transformer. Transformers have shown remarkable success in various natural language processing (NLP) tasks due to their ability to capture long-range dependencies and contextual information effectively.

- **Denoising and Generation:** The core concept of DiT involves denoising tokens within the input sequence iteratively. This denoising process helps the model refine its understanding of the input and generate more coherent and contextually relevant outputs. DiT utilizes denoising objectives similar to those used in diffusion models to guide the training process.

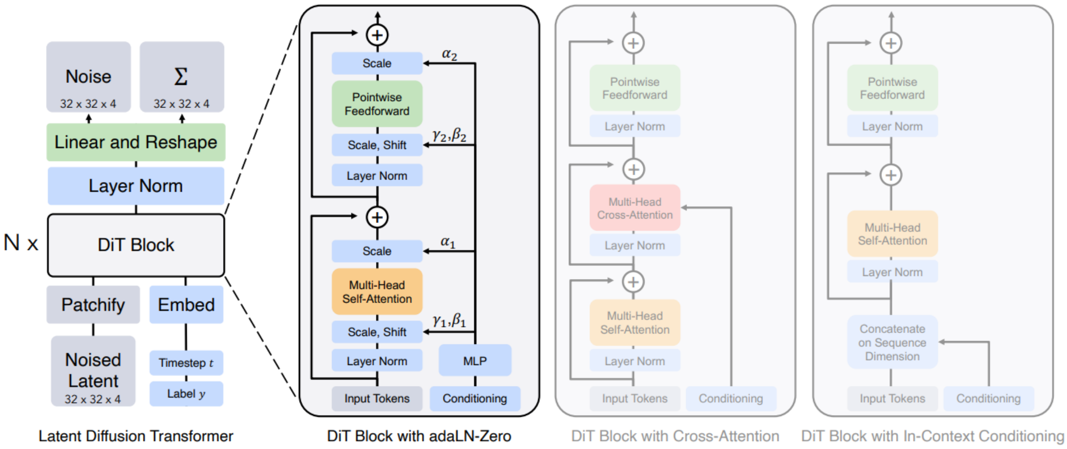


Figure 13.10: Diffusion Transformer (DiT).

13.4 Potential Applications of Diffusions in Biz/Econ Research

The diffusion models have a lot of potential applications in business and economics research. Here are some potential applications:

- **ML-Driven Hypothesis Generation:** Ludwig and Mullainathan (2024) utilize Convolutional Neural Network (CNN) algorithms directly to generate hypotheses that are interpretable and testable regarding human behaviors. They employed mug shots alongside machine learning predictions to assess behavior in decisions related to jailing, leading to the revelation of new information (approximately 22.3%) previously undiscovered. Additionally, they generated counterfactual mug shots based on algorithmic discoveries and iterated with crowd-sourced workers to validate that what human judges perceive aligns with the CNN algorithm's interpretations, thereby formalizing algorithmic discoveries into testable hypotheses.

- **Product Aesthetics Design:** Burnap et al. (2023) employ a combination of Variational Autoencoder (VAE) and Generative Adversarial Network (GAN) to enhance the aesthetics of product design, resulting in improved predictive capabilities regarding the appeal of new aesthetic designs when compared to established benchmarks. They demonstrated that automatically generated designs not only appeal to consumers but also closely resemble designs that have been introduced to the market.

- **GenAI for Logo Design:** Dew et al. (2022) apply image processingsegmentation algorithms to extract information from brand logos and multimodal-VAE to learn the latent representation of logos, and to generate new logos and predict consumer preferences. Finally, they show that manipulating the model's learned representations through what they term "brand arithmetic" yields new brand identities and can help with ideation.

- **Impact of GenAI on Art Creativity:** Zhou and Lee (2024) demonstrate that text-to-image artificial intelligence (AI) models, over time, can notably boost human creative productivity by 25% and elevate the value, as quantified by the likelihood of receiving a favorite per view, by 50%.

Bibliography

Adukia, A., Eble, A., Harrison, E., Runesha, H. B., and Szasz, T. (2023). What we teach about race and gender: Representation in images and text of children's books. *The Quarterly Journal of Economics*, 138(4):2225–2285.

Alexe, B., Deselaers, T., and Ferrari, V. (2012). Measuring the objectness of image windows. *IEEE transactions on pattern analysis and machine intelligence*, 34(11):2189–2202.

Apel, M. and Grimaldi, M. B. (2014). How informative are central bank minutes? *Review of Economics*, 65(1):53–76.

Ash, E. and Hansen, S. (2023). Text algorithms in economics. *Annual Review of Economics*, 15(1):659–688.

Bajari, P., Nekipelov, D., Ryan, S. P., and Yang, M. (2015). Machine learning methods for demand estimation. *American Economic Review*, 105(5):481–485.

Baker, S. R., Bloom, N., and Davis, S. J. (2016). Measuring economic policy uncertainty. *The quarterly journal of economics*, 131(4):1593–1636.

Barkan, O. and Koenigstein, N. (2016). Item2vec: neural item embedding for collaborative filtering. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Burnap, A., Hauser, J. R., and Timoshenko, A. (2023). Product aesthetic design: A machine learning augmentation. *Marketing Science*, 42(6):1029–1056.

Cao, K., Xia, Y., Yao, J., Han, X., Lambert, L., Zhang, T., Tang, W., Jin, G., Jiang, H., Fang, X., et al. (2023). Large-scale pancreatic cancer detection via non-contrast ct and deep learning. *Nature medicine*, pages 1–11.

Chen, F., Liu, X., Proserpio, D., and Troncoso, I. (2022). Product2vec: Leveraging representation learning to model consumer product choice in large assortments. *NYU Stern School of Business*.

Cheng, M.-M., Zhang, Z., Lin, W.-Y., and Torr, P. (2014). Bing: Binarized normed gradients for objectness estimation at 300fps. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3286–3293.

Cohen, M. C., Zhang, R., and Jiao, K. (2022). Data aggregation and demand prediction. *Operations Research*, 70(5):2597–2618.

Coursera (2021). Natural language processing specialization.

Covington, P., Adams, J., and Sargin, E. (2016). Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198.

Cui, R., Gallino, S., Moreno, A., and Zhang, D. J. (2018). The operational value of social media information. *Production and Operations Management*, 27(10):1749–1769.

Deming, D. and Kahn, L. B. (2018). Skill requirements across firms and labor markets: Evidence from job postings for professionals. *Journal of Labor Economics*, 36(S1):S337–S369.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dew, R., Ansari, A., and Toubia, O. (2022). Letting logos speak: Leveraging multiview representation learning for data-driven branding and logo design. *Marketing Science*, 41(2):401–425.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

Farias, V. F. and Li, A. A. (2019). Learning preferences with side information. *Management Science*, 65(7):3131–3149.

Fei-Fei, L., , Li, Y., and Gao, R. (2023). Cnn architectures. Online; accessed Apr. 21, 2024.

Fei-Fei, L. and Adeli, E. (2024). Convolutional networks for visual recognition. Online; accessed Apr. 21, 2024.

Gabel, S., Guhl, D., and Klapper, D. (2019). P2v-map: Mapping market structures for large retail assortments. *Journal of Marketing Research*, 56(4):557–580.

Gentzkow, M., Kelly, B., and Taddy, M. (2019a). Text as data. *Journal of Economic Literature*, 57(3):535–574.

Gentzkow, M. and Shapiro, J. M. (2010a). What drives media slant? evidence from us daily newspapers. *Econometrica*, 78(1):35–71.

Gentzkow, M. and Shapiro, J. M. (2010b). What drives media slant? evidence from u.s. daily newspapers. *Econometrica*, 78(1):35–71.

Gentzkow, M., Shapiro, J. M., and Taddy, M. (2019b). Measuring group differences in high-dimensional choices: method and application to congressional speech. *Econometrica*, 87(4):1307–1340.

Girshick, R. (2015). Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448.

Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

Grbovic, M., Radosavljevic, V., Djuric, N., Bhamidipati, N., Savla, J., Bhagwan, V., and Sharp, D. (2015). E-commerce in your inbox: Product recommendations at scale. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1809–1818.

Groseclose, T. and Milyo, J. (2005). A measure of media bias. *The quarterly journal of economics*, 120(4):1191–1237.

Gu, S., Kelly, B., and Xiu, D. (2020). Empirical asset pricing via machine learning. *The Review of Financial Studies*, 33(5):2223–2273.

Hansen, S., McMahon, M., and Prat, A. (2018). Transparency and deliberation within the fomc: A computational linguistics approach. *The Quarterly Journal of Economics*, 133(2):801–870.

Hassan, T. A., Hollander, S., Van Lent, L., and Tahoun, A. (2019). Firm-level political risk: Measurement and effects. *The Quarterly Journal of Economics*, 134(4):2135–2202.

Hastie, T., Tibshirani, R., Friedman, J. H., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer.

He, K. (2018). Learning deep representations for visual recognition. Online; accessed Apr. 22, 2024.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Ho, J., Jain, A., and Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851.

Hoberg, G. and Phillips, G. (2010). Product Market Synergies and Competition in Mergers and Acquisitions: A Text-Based Analysis. *The Review of Financial Studies*, 23(10):3773–3811.

Hoberg, G. and Phillips, G. (2016a). Text-based network industries and endogenous product differentiation. *Journal of Political Economy*, 124(5):1423–1465.

Hoberg, G. and Phillips, G. (2016b). Text-based network industries and endogenous product differentiation. *Journal of Political Economy*, 124(5):1423–1465.

Jean, N., Burke, M., Xie, M., Davis, W. M., Lobell, D. B., and Ermon, S. (2016). Combining satellite imagery and machine learning to predict poverty. *Science*, 353(6301):790–794.

Jiang, J., Kelly, B., and Xiu, D. (2023). (re-) imag (in) ing price trends. *The Journal of Finance*, 78(6):3193–3249.

Jurafsky, D. and Martin, J. H. (2024). *Speech and Language Processing (3rd ed. draft)*. Stanford University.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Kleinberg, J., Ludwig, J., Mullainathan, S., and Obermeyer, Z. (2015). Prediction policy problems. *American Economic Review*, 105(5):491–495.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Kumar, M., Eckles, D., and Aral, S. (2020). Scalable bundling via dense product embeddings. *arXiv preprint arXiv:2002.00100*.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Li, H., Simchi-Levi, D., Wu, M. X., and Zhu, W. (2023). Estimating and exploiting the impact of photo layout: A structural approach. *Management Science*, 69(9):5209–5233.

Li, K., Mai, F., Shen, R., and Yan, X. (2021). Measuring corporate culture using machine learning. *The Review of Financial Studies*, 34(7):3265–3315.

Liu, Z., Huang, D., Huang, K., Li, Z., and Zhao, J. (2021). Finbert: A pre-trained financial language representation model for financial text mining. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence*, pages 4513–4519.

Loughran, T. and McDonald, B. (2011). When is a liability not a liability? textual analysis, dictionaries, and 10-ks. *The Journal of finance*, 66(1):35–65.

Ludwig, J. and Mullainathan, S. (2024). Machine learning as a tool for hypothesis generation. *The Quarterly Journal of Economics*, 139(2):751–827.

Mosteller, F. and Wallace, D. L. (1963). Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed federalist papers. *Journal of the American Statistical Association*, 58(302):275–309.

Peebles, W. and Xie, S. (2023). Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4195–4205.

Peng, J. and Liang, C. (2023). On the differences between view-based and purchase-based recommender systems. *MIS Quarterly*, 47(2).

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3.

Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. (2022). High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695.

Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer.

Schütze, H., Manning, C. D., and Raghavan, P. (2008). *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Stanford University (2021). Basic text processing-regular expressions.

Stock, J. H. and Trebbi, F. (2003). Retrospectives: Who invented instrumental variable regression? *Journal of Economic Perspectives*, 17(3):177–194.

Tetlock, P. C. (2007). Giving content to investor sentiment: The role of media in the stock market. *The Journal of finance*, 62(3):1139–1168.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., and Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, 104:154–171.

Van Binsbergen, J. H., Bryzgalova, S., Mukhopadhyay, M., and Sharma, V. (2024). (almost) 200 years of news-based economic sentiment. Technical report, National Bureau of Economic Research.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, B., Wang, A., Chen, F., Wang, Y., and Kuo, C.-C. J. (2019). Evaluating word embedding models: Methods and experimental results. *APSIPA transactions on signal and information processing*, 8:e19.

Zhan, R., Pei, C., Su, Q., Wen, J., Wang, X., Mu, G., Zheng, D., Jiang, P., and Gai, K. (2022). Deconfounding duration bias in watch-time prediction for video recommendation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4472–4481.

Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.

Zhang, M. and Luo, L. (2023). Can consumer-posted photos serve as a leading indicator of restaurant survival? evidence from yelp. *Management Science*, 69(1):25–50.

Zhang, S., Lee, D., Singh, P. V., and Srinivasan, K. (2022). What makes a good image? airbnb demand analytics leveraging interpretable image features. *Management Science*, 68(8):5644–5666.

Zhou, E. and Lee, D. (2024). Generative artificial intelligence, human creativity, and art. *PNAS nexus*, 3(3):pgae052.

Zitnick, C. L. and Dollár, P. (2014). Edge boxes: Locating object proposals from edges. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 391–405. Springer.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 67(2):301–320.