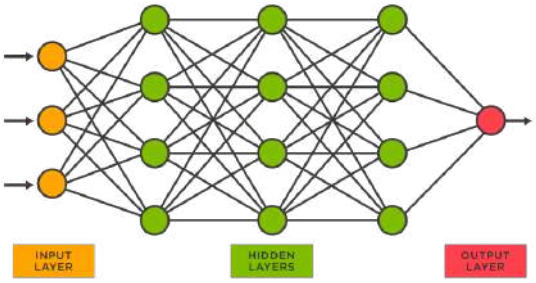
Introduction to Deep Learning

February 13, 2025

Department of Electrical Engineering
University of Notre Dame



Contents

Preface

These lecture notes were written for an introduction to deep learning course that I first offered at the University of Notre Dame during the Spring 2023 semester. I offered this course because deep learning had become such a force in both the technical and lay communities, that I felt it was critical that MS/PhD and even undergraduate students not leave the University without having some inkling of what deep learning was capable of.

Neural network learning is often described as having three distinct *waves* of research activity as shown in Fig. 1 The first wave dates back to the 1960's with the emergence of Widrow's Madaline learning system [**WH60**] and Rosenblatt's perceptron model [**Ros58**]. The second wave dates back to the 1980's with Hopfield showing that neural network models could solve NP-hard combinatoric optimization problems [**Hop82**] and the development of the backpropagation training algorithm [**RHW86**]. The third wave began in 2012 with advances in the use of deep convolutional neural networks for image classification [**KSH12**]. Since that time, interest and research in deep learning has exploded in both the technical and lay communities. The technical side saw startling advances in the use of deep generative learning for natural language processing (NLP) which have recently begun to disruptively filter into the lay community [**VDBZ+23**].

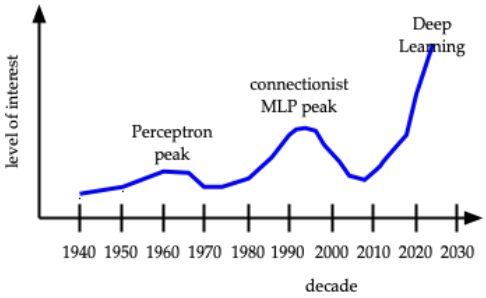


FIGURE 1. Three wave of neural network research

I received my PhD as part of the second wave. Like many other research scientists, I left neural network research in the 1990's because it appeared that the fundamental problem of feature engineering was domain specific. I then turned my attention to control theory where feedback provided a mechanism for learning features in a concrete manner. So when I picked up this deep learning course 30 years later, I was very skeptical that I would find anything truly novel there. I was mistaken.

Deep learning, particularly with recent advances in generative pre-trained transformers, appears to mimic human learning in a manner that can pass the Turing test if the examiner is rather "lazy". In reviewing how these machines are trained, I came to see close parallels with how students approach their course work and it became apparent that deep learning success stories point to a much deeper transformation in how we build autonomous machines. So, I was converted and these lecture notes are the result of my re-immersion into neural network computation.

This book has been organized into 9 chapters. Chapter 1 introduces the main problem solved by deep learning; a supervised learning problem that is often referred to as learning-by-example. Chapter 2 reviews early work from the 1980's using statistical methods to characterize the sample complexity and generalization ability of neural networks. Chapter 3 examines the three main neural network models; perceptron, multi-layer perceptron, and deep neural networks. Chapter 4 covers the use of a well-known Python software library, TensorFlow, for training deep models. After covering the deep learning basics in chapters 1-4, the book covers the major application success stories in computer vision (chapter 5), natural language processing (chapter 6), and generative models (chapter 7). Generative models have recently had a great impact on human society so I devote chapter 8 to an examination of security and fairness in deep learning models. I close the book (chapter 9) on deep reinforcement learning. I do not always teach this last chapter since the material has often been covered in greater depth by

other classes at Notre Dame. My coverage of all of these topics is not terribly ”deep” for I was interested in providing a survey that would touch on most of the major models in used to date. The field, however, is changing rapidly and there are a number of important topics that I had to leave out. Nonetheless, the coverage in this book should be sufficient to give students a good enough introduction to deep learning that they can begin using it in their own future research.

M. D. Lemmon
Department of Electrical Engineering
University of Notre Dame
Fall, 2024

CHAPTER 1

Introduction - Learning by Example

Deep learning uses neural network models with many hidden layers to solve supervisory learning problems. In supervisory learning, we have a collection of *training examples* where each example consists of an *input* and a *target*. The objective is to use these examples to select a function (a.k.a. *model* or *predictor*) that maps any input onto the correct target. We say this learning problem is supervised because our examples contain both the input and target. It is also common to refer to this as a *learning-by-example* problem. Learning problems that attempt to learn a model with only the inputs (i.e. no target) are said to be unsupervised. The purpose of this chapter is to formally state the learning-by-example problem, to describe several important versions of this problem, and to provide concrete examples illustrating how the training examples are used to select a suitable model.

1. Problem Statement

The learning-by-example problem is to select a model from a *model set* based on example's of a *system's* inputs and associated outputs. The model is selected to be optimal in the sense of minimizing the error (a.k.a. loss) between the model's output and the system's output for the same given input. The learning-by-example problem may therefore be seen as consisting of three distinct components; the *system*, the *model set*, and the *loss function*. Each of these distinct components is described in more detail below.

System: The system is defined using the block diagram in Fig. 1. The system may be seen as the cascade of two systems; a *generator* whose output

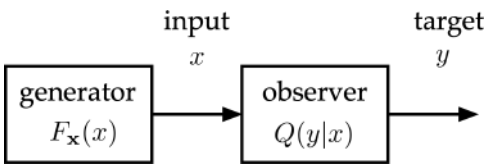


FIGURE 1. System in Learning-by-Example Problem

is driving an *observer*. The *generator* creates *inputs*, $x \in X$, where X is called the input set. The generator's output is drawn in an independent and identically distributed (i.i.d.) manner from probability distribution, $F_{\mathbf{x}}(x)$. These inputs are used by an *observer* to create a *target*, $y \in Y$ for any input $x \in X$, where Y is called the target set. The target, y, is also drawn in an i.i.d. manner from a conditional probability distribution $Q_{\mathbf{y}|x}(y|x)$. The learning-by-example problem assumes that both distributions are unknown, but that we have a finite *dataset* containing N *samples*, $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$, that were drawn in an i.i.d. manner from the joint probability distribution,

$$P_{\mathbf{x},\mathbf{y}}(x, y) = F_{\mathbf{x}}(x)Q_{\mathbf{y}|x}(y|x).$$

We refer to the set $\mathcal{D}$ as the *dataset* drawn by the system and each pair (x_k, y_k) in $\mathcal{D}$ will be called an *example* or *sample*.

Model Set: The objective of the learning-by-example problem is to use the dataset, $\mathcal{D}$, drawn by the system to select a *model*, $h : X \to Y$, that maps any given *input*, $x \in X$, onto a predicted target, $\widehat{y} = h(x) \in Y$. This model is drawn from a *model set*, $\mathcal{H}$. It is also common to refer to the model as a *predictor* since it is "predicting" the target selected by the observer. We assume that each model, $h \in \mathcal{H}$, is parameterized by a collection of parameters that are also called *weights*, $w \in W$, where W is the weight set. When it is important to mention this parameterization, we often write the model parameterized by w as $h_w : X \to Y$. In this regard, solving the learning-by-example uses the dataset, $\mathcal{D}$, to select a weight $w \in W$ for a model $h_w : X \to Y$ that makes predictions, $\widehat{y} = h_w(x)$, for the given input $x \in X$.

Loss Function:. The model $h \in \mathcal{H}$ predicts the system *observer's* response to a given input $x \in X$. The loss function $L : Y \times Y \to \mathbb{R}$ measures how well the model estimates the observer's response to x. Common examples of loss functions are *mean squared error* (MSE),

$$L(y, \widehat{y}) = (y - \widehat{y})^2,$$

used in regression problems and the *binary cross-entropy* function,

$$L(y, \widehat{y}) = y \log(\widehat{y}) + (1 - y) \log(1 - \widehat{y}),$$

used in logistic regression problems. Another common loss function is the *classification error*,

$$L(y, \widehat{y}) = \mathbb{1}(y \neq \widehat{y}),$$

used in binary classification problems where $\mathbb{1}(\cdot)$ is an indicator function [1]. These three problems (regression, classification, logistic regression) represent three major categories of learning-by-example that are often found in practice.

Ideally, one wants to select a model $h \in \mathcal{H}$ that minimizes the average *actual risk* over *all* samples that the system can generate. This *actual risk* is defined as

$$R[y] = \mathbb{E}_{\mathbf{x},\mathbf{y}} \left[L(\mathbf{y}, h(\mathbf{x})) \right] = \int \int L(y, h(x)) P_{\mathbf{y},\mathbf{x}}(x, y) dy dx.$$

The problem with this, however, is that our problem stated that the generator distribution, $F_{\mathbf{x}}(x)$, and observer distribution, $Q_{\mathbf{y}|x}(y|x)$, are unknown. This means that we cannot evaluate $R[h]$ to see if our model actually minimizes $R[h]$.

Since we cannot evaluate $R[h]$, we invoke the principle of *empirical risk minimization* [**Vap98**]. We first define the *empirical risk function* with respect to a given dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$ of N samples drawn by the

[1] $\mathbb{1}(\cdot)$ is 1 if the logical expression in the parentheses is true and is 0 if false.

system. The *empirical risk* of a model $h \in \mathcal{H}$ with respect to dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ is

$$\widehat{R}_{\mathcal{D}}[h] = \frac{1}{N}\sum_{k=1}^{N} L(y_k, h(x_k)).$$

This is, of course, the sample mean of $L(y, h(x))$ and because the samples (x_k, y_k) are drawn in an i.i.d. manner from the joint distribution $P_{\mathbf{x},\mathbf{y}}(x, y))$, we know by the weak law of large numbers (see appendix A) that $\widehat{R}_{\mathcal{D}}[h] \to R[h]$ as the dataset size, $|\mathcal{D}| = N$, goes to infinity. In particular, we say any model $h \in \mathcal{H}$ generalizes beyond its training data if its average empirical risk is close to the average actual risk. We can make this notion of closeness more precise by adopting an $\epsilon - \delta$ definition. Formally, we say a model $h \in \mathcal{H}$ *generalizes* well for any $\epsilon, \delta > 0$ if there exists a positive integer, $N_{\epsilon,\delta}$, such that for any dataset $\mathcal{D}$ with $N_{\epsilon,\delta}$ examples we have

$$\Pr_{\mathcal{D}}\left[\left|R[h] - \widehat{R}_{\mathcal{D}}[h]\right| > \epsilon\right] < \delta.$$

We refer to $N_{\epsilon,\delta}$ as the given model's *sample complexity*. The deep learning problem is, therefore, to use a given dataset, $\mathcal{D}$, to select a model $h \in \mathcal{H}$ that generalizes beyond its dataset in the sense specified above.

2. Types of Learning-by Example Problems

There are three basic types of learning-by-example problems; binary classification, regression, and logistic regression. Each problem type is described below.

2.1. Binary Classification: This is a classical hypothesis testing problem from detection/estimation theory [**VT04**]. The binary classification problem takes an input example, say a vector $x \in \mathbb{R}^n$, and classifies it as belonging to one of 2 discrete classes. The multi-class classification problem does this with M discrete classes.

Let us now define binary classification in terms of the three components identified above for a learning-by-example problem. The system inputs, x are drawn in an i.i.d. manner from a distribution $F_{\mathbf{x}}(x)$ where x is in an input set X. The target created by the observer is either 0 or 1, i.e. $y \in Y = \{0, 1\}$. The observer's classification is generated by drawing the target y from the conditional distribution $Q_{\mathbf{y}|x}(y|x)$ in an i.i.d. manner. The model set, $\mathcal{H}$, consists of function $h_w : X \to \{0, 1\}$ where $w \in W$ is a weight parameterizing the model. The loss function is the *classification error*

$$L(y, h_w(x)) = \mathbb{1}(y \neq h_w(x)).$$

In other words the loss is 1 if the model's prediction is incorrect and is 0 otherwise. The binary classification problem is to find a model, $h_w \in \mathcal{H}$, that minimizes the empirical risk

$$\widehat{R}_{\mathcal{D}}(w) = \frac{1}{N} \sum_{k=1}^{N} \mathbb{1}(y_k \neq h_w(x_k)),$$

with respect to a known dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$. Note that this simply counts up the number of samples in the dataset that are misclassified by the model h_w. By the law of large numbers we know that this converges to the true *error probability* as the size, N, of the dataset goes to infinity.

For a concrete example of a binary classification problem, let us consider a bank that must decide whether a customer's loan application is to be approved or not. The inputs are the loan applications that are randomly drawn from the pool of city residents wanting a loan. These loan applications contain many different categories of information that include numerical data attributes (age, wage, assets, and debt) and categorical data attributes (race, sex). These data attributes are then used by the bank's human loan officer to decide whether or not the loan application is approved. The target, therefore, is the loan officer's decision to approve or deny the loan application. Since there are only two options, this is a binary classification problem. The machine learning problem arises because the bank wants to replace the

human loan officer with a computer program to approve or reject all future loan applications. The computer program would compute the output of a model, h_w, that maps the data attributes in the loan application onto 1 (approve) or 0 (deny). The machine learning problem is to train this model to mimic the human loan officer's past behavior on loan approval. The historical record of all past loan applications and the bank's human decision maker would form the samples in the dataset used to train the model.

2.2. Regression Problem: This is the basic parameter estimation problem from detection/estimation theory [**VT04**]. We assume there is an unknown real-valued function $g : \mathbb{R}^n \to \mathbb{R}$ from the real input space, $\mathbb{R}^n$, to the real target space, $\mathbb{R}$. The samples $x \in X$ are drawn in an i.i.d. manner with respect to distribution $F_{\mathbf{x}}(x)$. The observer takes an input, $x \in \mathbb{R}^n$ and creates a continuous valued target $y \in \mathbb{R}$ through the equation

$$y = g(x) + n \tag{1}$$

where n is a zero mean random variable with finite variance. The model set is a chosen set of functions, $h_w : \mathbb{R}^n \to \mathbb{R}$, parameterized by the weights $w \in W$. The learning problem is to use a dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ generated by the system and use that data to select a model h_w that predicts the *noiseless* output of the unknown function g. In other words, the dataset targets, y, are noisy version of $g(x)$ and our model, h_w is trying to predict what the noiseless $g(x)$ is for a given x. The quality of this prediction is determined by the squared error loss function

$$L(h, h_w(x)) = (y - h_w(x))^2$$

where y is the "noisy" target generated by the observer in equation (1). We select the model that minimizes the empirical risk

$$\widehat{R}_{\mathcal{D}}(w) = \frac{1}{N} \sum_{k=1}^{N} (y_k - h_w(x_k))^2,$$

over the dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$. As dataset size, N, goes to infinity, this empirical risk, $\widehat{R}_{\mathcal{D}}(w)$, converges to the *mean squared error* (MSE)

$$\text{MSE} = R(w) = \mathbb{E}_{\mathbf{x},\mathbf{y}}\left[(\mathbf{y} - h_w(\mathbf{x}))^2\right].$$

of the model where the expectation is taken with respect to the unknown system distributions, $F_{\mathbf{x}}(x)$ and $Q_{\mathbf{y}|x}(y|x)$.

For a concrete example of a regression problem, let us again consider the bank that must take a customer's credit card application and decide what credit limit to place on the approved card. The inputs are applications drawn from the pool of customers wanting a credit card. The applications contain both numerical data (age, wage, debt) and categorical data (sex, race) used in deciding the credit limit. The credit limit itself is a real-valued number. The credit limit decided by human bank officers has some variability because there are several human decision makers. The machine learning problem is then to train a model that minimizes the mean squared error in the model's prediction and the actual credit limits selected by the pool of human decision makers.

2.3. Logistic Regression. The binary classification problem has binary valued targets. There are, however, applications where we want to estimate the *likelihood* of a given decision being correct or not. One example of such a problem is predicting the likelihood that a person will have a heart attacked based on various vital statistics such as cholesterol level, age, weight, etc. We cannot predict a future attack with certainty for this individual, but we can say how likely it is for one to occur given the frequency with which such events occur in the entire population for a person with the given vital statistics. So in this learning problem we want to train a model $h_w : \mathbb{R}^n \to [0, 1]$ that maps a tensor of real-valued attributes onto the interval $[0, 1]$ such that $h_w(x)$ is the probability that an individual with attributes vector $x \in \mathbb{R}^n$ can be classified as 1 (heart attack) or -1 (no heart attack).

This learning problem is called *logistic regression*. Let us try to identify the three components of logistic regression's learning-by-example problem. The *system* consists of a generator that draws a patient's attribute vector, $x \in \mathbb{R}^n$, from an unknown distribution $F_{\mathbf{x}}(x)$. The *system observer* draws a binary valued target, $y \in \{-1, 1\}$ from the conditional density $Q_{\mathbf{y}|x}(y|x)$ where $+1$ means the patient had a heart attack and -1 the patient did not have a heart attack. So the dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ is drawn from the historical record of the population for which we already know the outcome. What we want to do is train a model that "learns" this condition density $Q_{\mathbf{y}|x}(y|x)$ from the dataset $\mathcal{D}$. The logistic regression problem therefore tries to learn $Q_{\mathbf{y}\,|x}(y|x)$ from a dataset $\mathcal{D}$ whose samples (x, y) are drawn with joint distribution $F_{\mathbf{x}}(x)Q_{\mathbf{y}\,|\,x}(y|x)$ where the target y is binary valued.

We need to identify a suitable model set for this problem. To motivate our selections, let

$$q^+(x) \stackrel{\text{def}}{=} Q_{\mathbf{y}|x}(y = +1 \,|\, x) = \Pr(y = +1 \,|\, x)$$

denote the probability that a patient in the dataset with attribute x had a heart attack, $y = +1$. We can use Bayes theorem to rewrite $q^+(x)$ as

$$q^+(x) \quad = \quad \frac{Pr(x \,|\, y = +1) \times \Pr(y = +1)}{\Pr(x \,|\, y = +1) \times \Pr(y = +1) + \Pr(x \,|\, y = -1) \times \Pr(y = -1)}$$

We define the log likelihood ratio of a person having attribute x having a heart attack over not having one as

$$s(x) = \log \frac{\Pr(x, y = +1)}{\Pr(x, y = -1)} = \log \frac{\Pr(x \,|\, y = +1) \times \Pr(y = +1)}{\Pr(x \,|\, y = -1) \times \Pr(y = -1)}$$

and then note that we can rewrite $q^+(x)$ as the following function of this log likelihood ratio,

$$q^+(x) = \frac{1}{1 + e^{-s(x)}} \stackrel{\text{def}}{=} \sigma(s(x))$$

This function $\sigma : \mathbb{R} \to [0, 1]$ is called a *logistic* or *softmax* function.

We can directly express the observer's conditional density $Q_{\overline{y}|x}(y|x)$ in terms of the logistic function. In particular, note that

$$Q_{\overline{y}\,|\,x}(y|x) = \begin{cases} q^+(x) & \text{for } y = +1 \\ 1 - q^+(x) & \text{for } y = -1 \end{cases} = \begin{cases} \sigma(s(x)) & \text{for } y = +1 \\ \sigma(-s(x)) & \text{for } y = -1 \end{cases}$$

where we used the fact that the logistic function satisfies $1 - \sigma(s(x)) = \sigma(-s(x))$ to obtain the last relation. This last relation suggests that the model we should choose is a logistic function acting on the log likelihood function $s(x)$. In other words, $h_w \in \mathcal{H}$ will be

$$h_w(x) = \sigma(s_w(x))$$

where σ is a logistic activation function acting on a neural network model, $s_w(x)$ with weights w that predicts the log-likelihood ratio $s(x)$ for an input with attribute x.

To complete the characterization of the logistic regression problem, we now need to define a suitable loss function that can be used to train the models $\sigma(s_w(x))$ from the dataset $\mathcal{D}$. To do this we define a likelihood function for the model. A likelihood function measures how well a model such as $h_w(x) = \sigma(s_w(x))$ explains the observed data in dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$. The likelihood function is the probability distribution of the given dataset, $\mathcal{D}$ being generated by the system. So we can write it as

$$\mathcal{L}(\mathcal{D} \,|\, w) \;\; = \;\; \prod_{k=1}^{N} Q_{\mathbf{y}|x}(y_k \,|\, x_k) = \prod_{k=1}^{N} Q_{\mathbf{y}|x}(y_k \,|\, x_k)$$

Since we have $Y = \{-1, +1\}$, we can write

$$Q_{\mathbf{y}|x}(y_k \,|\, x_k) \approx \sigma(y_k s_w(x_k))$$

where $\sigma(y_k s_w(x_k))$ is the model's estimate of the condition probability.

The optimal set of weights for this model are those that minimize the negative log likelihood function $-\frac{1}{N} \log \mathcal{L}(\mathcal{D}|w)$. Because we deal with the log likelihood function, the product of probabilities become a sum of log

probabilities and our expression take the form of a sample mean of these log probabilities. In other words, the negative log-likelihood function is

$$\begin{aligned} -\frac{1}{N}\log \mathcal{L}(\mathcal{D}\,|\,w) &= -\frac{1}{N}\log\left(\prod_{k=1}^{N}\sigma(y_k s_w(x_k))\right) \\ &= \frac{1}{N}\sum_{k=1}^{N}\log\left(\frac{1}{\sigma(y_k s_w(x_k))}\right) \\ &= -\frac{1}{N}\sum_{k=1}^{N}\log(1+e^{-y_k s_w(x_k)}) \end{aligned}$$

The last equation shows that the negative log likelihood can be seen as the average loss seen over the dataset, in other words it is the model's *empirical risk*, if we take the loss function to be

$$L(y_k, h_w(x_k)) = -\log(1+e^{-y_k s_w(x_k)})$$

So this is the desired loss function for our logistic regression.

It is important to note that this loss function will be different if our target has different values. The preceding loss function assumes targets are in $\{-1,+1\}$. If our targets were $\{0,1\}$, then one can show that the loss function will be

$$L(y_k, h_w(x_k)) = y_k \log(\sigma(s_w(x_k))) + (1-y_k)\log(1-\sigma(s_w(x_k)))$$

This is called the *binary cross entropy* loss function.

3. Neural Network Model Sets

A model set, $\mathcal{H}$, is a collection of models, $h_w : X \to Y$, parameterized by weights, w. The learning by example problem is to find a weight, w^*, that minimizes the empirical risk with respect to dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$. Three important types of model sets; the perceptron, the multi-layer perceptron, and deep neural networks are described below.

3.1. Perceptron: The perceptron [Ros58] is an early neural network model that appeared in the 1960's. Perceptron models take the form

$$h_{w,b}(x) = \sigma(w^T x + b) \tag{2}$$

where $x \in \mathbb{R}^n$ is the input, $w \in \mathbb{R}^n$ is the weight vector and $b \in \mathbb{R}$ is another parameter called a *bias*, and $\sigma : \mathbb{R} \to \mathbb{R}$ is an *activation function* that is usually taken to be monotone increasing. Since the argument to the perceptron's activation function is a linear form, we often refer to perceptrons as *linear machines*. The activation function takes a variety of forms. Commonly used activation functions are shown in Table. 1.

linear	step	ReLu
$y = s$	$y = \begin{cases} 1 & s \geq 0 \\ 0 & s < 0 \end{cases}$	$y = \begin{cases} s & s \geq 0 \\ 0 & s < 0 \end{cases}$
softsign	**logistic**	**tanh**
$y = \frac{1}{1+\lvert s\rvert}$	$y = \frac{e^{-s}}{1+e^{-s}}$	$y = \tanh(s)$

TABLE 1. Table of Activation Functions

The model in equation (2) is just one type of linear machine. In many applications the designer introduces a set of *basis* functions, $\{\phi_k\}_{k=1}^N$ where $\phi_k : \mathbb{R}^n \to \mathbb{R}$. Each basis function maps the input $x \in \mathbb{R}^n$ onto a real number that we call a *feature*. The intensity of the ϕ_k is then a measure of how strongly the input triggers the given feature. We can stack these features to form a vector function $\boldsymbol{\phi} : \mathbb{R}^n \to \mathbb{R}^N$ whose output is then take as a user designed *feature vector* for the input, x.

Let us consider basis functions

$$\boldsymbol{\phi}(x) = \begin{bmatrix} 1 \\ x \end{bmatrix}$$

then we can rewrite the earlier perceptron model from equation (2) as

$$h_{w,b}(x) = \sigma(\boldsymbol{\theta}^T \boldsymbol{\phi}(x)) \tag{3}$$

where $\boldsymbol{\theta} = \begin{bmatrix} b \\ w \end{bmatrix}$ is an *augmented weight vector* obtained by stacking the bias b on top of the original weights, w. The perceptron models in equation (2) and (3) are clearly equivalent, so we will switch back and forth between the two representations when it is convenient.

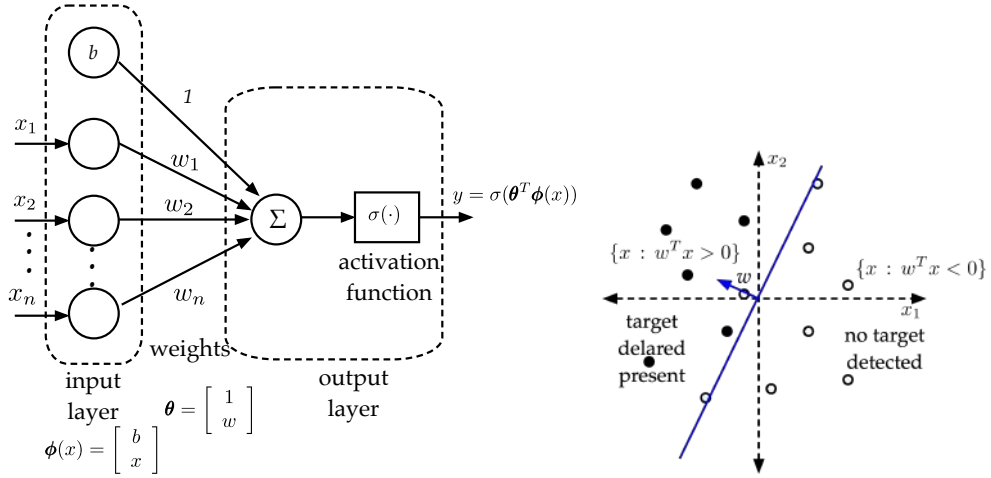


FIGURE 2. (left) Perceptron Model - (right) binary classification with perceptron

The perceptron's architecture from equation (2) can be visualized using the block diagram in Fig. 2. The model has a *layered* architecture consisting of an input layer and an output layer. The input vector x is an n-vector in $\mathbb{R}^n$, but the input layer has $n+1$ nodes, n of which hold the components of x and the last $n+1$st node holding the bias parameter b. The outputs of the input layer are then multiplied by the the weights in the augmented weight vector $\boldsymbol{\theta}$ and summed together before being passed through the activation function σ to obtain the perceptron's output, y.

Perceptrons can be readily used to solve binary classification problems. In this case we let the activation function σ be a sgn function and let x be a real valued vector in $\mathbb{R}^n$. Since the perceptron's output would then be $y = \operatorname{sgn}(w^T x)$, the vector, x will lie to one side of an $n-1$ dimensional hyperplane defined by the equation $0 = w^T x$. This hyperplane is called a

discriminant surface and the weight vector w is normal to this hyperplane. If the input x lies on the side of hyperplane pointed to by w, then the output is $+1$, otherwise it is -1. We then have

$$\text{if } x \text{ satisfies } 0 > w^T x, \text{ then } x \text{ is in class "detected" (+1)}$$
$$\text{if } x \text{ satisfies } 0 < w^T x, \text{ then } x \text{ is in class "not detected" (-1)}$$

The right side of Fig. 2 illustrates this hyperplane in a 2-dimensional input space as the blue line. The solid bullets represent inputs when the input is in class $+1$ and the circles are inputs when the input is in class -1. This perceptron in the figure misclassifies some of these inputs. In particular, we see that two of the -1 inputs are on the left side of the blue line and would therefore be declared as belonging to class $+1$. We count up these misclassified inputs to compute the empirical risk of the perceptron model. In this case, since there are 14 inputs, the empirical risk or estimated classification error is $2/14 = 1/7$.

3.2. Multi-layer Perceptron: Neural networks [**RHW86**, **MR89**] are biologically inspired models that extend the perceptron model in equation (2). These models are more powerful than perceptron models because we can prove they have a *universal approximation ability* [**Cyb89**]. This means that a neural network model can approximate any complex target function, whereas the perceptron is limited to target functions that have an underlying linearity to them.

The simplest neural network model is called a multi-layer perceptron (MLP). This model came to prominence in the late 1980's as the result of two results. The first result was the universal approximation theorem mentioned above. The second result was the development of a training algorithm for MLP's knows as *backpropagation* [**RHW86**].

The MLP simply adds one *hidden layer* with M nodes to the original perceptron shown in Fig. 2. Mathematically the output of the MLP is

$$y = \sigma\left(\sum_{k=1}^{M} \alpha_k \sigma(w_k^T x)\right) \tag{4}$$

where the input is $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$ is a scalar output. The activation function, σ, is applied to the outputs of the hidden layer and the output layer. So we now have two sets of weights. For the kth node in the hidden layer there is a weight vector $w_k \in \mathbb{R}^n$. There is also another set of scalar weights $\{\alpha_k\}_{k=1}^M$ that weights the outputs of the hidden layer. We can therefore also represent the MLP in matrix-vector form as

$$y = \sigma(\boldsymbol{\alpha}^T \sigma(\mathbf{W}x))$$

where

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_M \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} w_1^T \\ \vdots \\ w_M^T \end{bmatrix}.$$

The resulting model architecture can now be visualized as shown in Fig. 3. We again see the layered abstraction, where the hidden layer sits between the original perceptron's input and output layers.

3.3. Deep Neural Networks: While backpropagation provided an algorithmic way to efficiently train MLP's, it was soon found that obtaining good models required careful feature design on the front end. In other words, the designer would need to select a collection of basis functions $\phi_k : \mathbb{R}^n \to \mathbb{R}$ and then find the weights $\boldsymbol{\alpha}$ and $\boldsymbol{W}$ that minimized the loss function for the model

$$y = \sigma\left(\sum_{k=1}^{M} \alpha_k \sigma(w_k^T \boldsymbol{\phi}(x))\right). \tag{5}$$

The problem with this was that the selection of $\boldsymbol{\phi}$ was highly dependent on the application the model was being used for and so good models were highly dependent on the designer's sense of what "good" features should be.

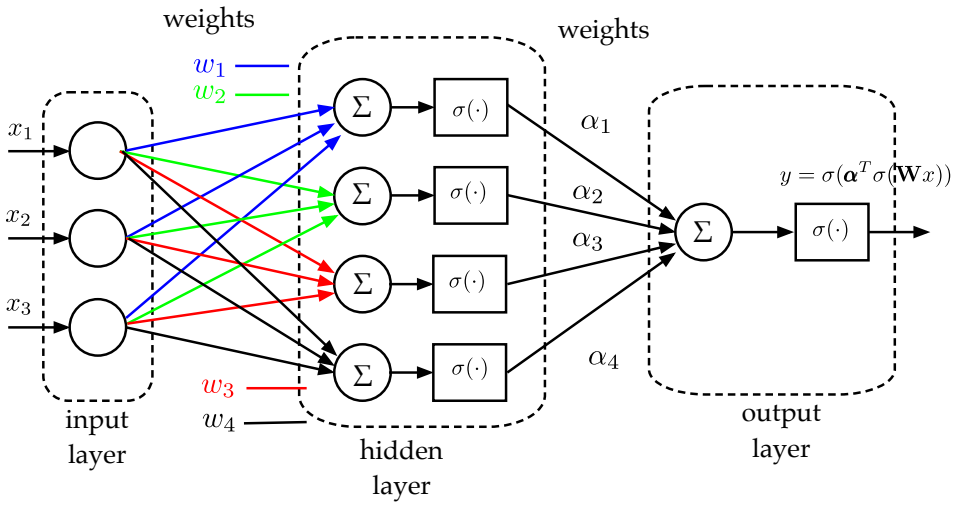


FIGURE 3. Multi-layer Perceptron Model

In other words, training the model did not really "learn" how to solve the learning-by-example problem from scratch. As a result of this realization, the second wave of neural network research began to dissipate in the 1990's and it continued to wither until new results appeared in 2012 with a new neural network architecture called a *deep convolutional neural network*.

Deep convolution neural networks were able to exceed the classification performance of earlier handwriting classifier MLP's [**LBD+89**] that had used pre-specified features, $\boldsymbol{\phi}$. These improvements were obtained by using a new model architecture called a deep convolutional neural network [**KSH12**]. The unique features of this model were that 1) it had multiple hidden layers and 2) it had a special structure to the weight matrix, $\boldsymbol{W}$, that were formed from translation invariant convolution kernels. These results were considered remarkable because they did not use prior feature engineering. So the 2012 results showed that deep neural network models were capable of learning "features" on their own without prior help from human designers. The resulting deep neural networks set off a resurgence of interest in neural network research based on a model that had deep stacks of hidden layers. This has led to remarkable progress over the past decade in

computer vision and natural language processing; progress that appears to be on the brink of transforming human society.

It should be noted that a number of other trends also played a role in the explosive interest in deep neural networks. In the first place, there were advances in object-oriented programming that made it possible to develop software frameworks that could be used to easily train these extremely large models. Another important development was the rise of the Internet that made it possible to easily access extremely large datasets that could be used for training these more sophisticated models. The other important development was the rise of Big Tech companies like Amazon, Google, and Facebook that used the Internet to gather large datasets and then sought to monetize that data through advanced deep learning applications. Another important factor was the ubiquity of mobile phones that essentially put the Internet in the hands of lay society and provided a ready source of data that the Big Tech companies took advantage of. All of these developments conspired to make deep learning well known to the lay public through the many apps these companies provided to consumers. Deep learning represents, in this regard, a transformational technology for human society so that understanding how and why it works is essential not only for expanding deep learning's potential but also is critical to understanding how we can manage its disruptive influence on human society.

4. Deep Learning Software Libraries

There are several python software libraries used for deep learning. One purpose of these lectures is to place one of these python libraries, TensorFlow/Keras, in the hands of students. This section walks through a python script that uses TensorFlow to instantiate and train a convolutional neural network using examples from the MNIST database.

MNIST is a large database of handwritten digits $(0-9)$ that is used to train various image processing systems. The database contains 60,000 training images and 10,000 testing images. All images are 28×28 monochrome images where each pixel value is an unsigned 8 bit integer $(0-255)$. This section will demonstrate how TensorFlow can be used to instantiate, train, and evaluate a simple 2D convolution neural network model that takes the pixel image and recognizes which digit that image represents.

We start our example by first loading the database. The MNIST dataset is included in TensorFlow and the following Python script loads the training and testing samples from TensorFlow's library of datasets.

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels),(test_images,test_labels) = mnist.load_data()
```

This script loads the full database into four Numpy arrays. To speed up the training time, we are only going to train and test our model on a subset of these four arrays. In particular, we will train on the first 5000 training images and first 1000 testing images.

```
train_images = train_images[:5000,:,:].reshape((5000,28,28,1))
train_labels = train_labels[:5000]
test_images = test_images[:1000,:,:].reshape((1000,28,28,1))
test_labels = test_labels[:1000]
```

The basic data type used in TensorFlow is a *tensor*. A tensor is a multi-dimensional array where the number of dimensions is called the tensor's *rank*. We usually specify a rank-n tensor's *shape* by the n-tuple, $(x_1, x_2, \ldots, x_n)$, where x_i is the number of array components along that ith dimension. A vector $x = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$, may therefore be seen as a rank-1 tensor with shape $(3,)$. A matrix $x = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$ is a rank-2 tensor with shape $(3, 3)$.

The database `train_images` array is initially a rank-3 tensor of shape (60000,28,28). The first command in the above script selects out the first

5000 images in this array and then reshapes it into a rank-4 tensor of shape (5000,28,28,1). This is done because the convolutional network we are going to instantiate is expecting rank-3 tensor images as inputs. The database `train_labels` array is a rank-1 tensor of shape (1000). The script's second command pulls out the first 5000 entries of this array. The remaining commands perform the same operations on the MNIST testing data.

The data arrays generated in the preceding script are actually Numpy arrays. Training of neural network models will be faster if we use these Numpy arrays to form *dataset* objects. Neural network training is usually not done on the whole dataset at once. Instead, we update a model's weights using a smaller *batch* of inputs in the training set. This is called *mini-batch* training and it can be done more efficiently by the computer if we create a *dataset* object to be used in training. A dataset object is an *iterator* that can be called recursively by the model's training method. One advantage of the dataset object is that it can divide up the dataset into batches, that can be called more quickly at training time. The following script uses the Numpy arrays to instantiate TensorFlow dataset objects that pre-batch the training and testing data into batches of 32 samples. The original image data array consists of unsigned 8 bit integers in the range $0 - 255$. Neural network models train better if the data type is floating point and if they are scaled to $[0, 1]$. So the following script also retypes the image data as floating point and rescales it to $[0, 1]$.

```
train_ds     = tf.data.Dataset.from_tensor_slices((train_images,train_labels))
train_ds.    = train_ds.batch(32).astype("float32")/255
test_ds      = tf.data.Dataset.from_tensor_slices((test_images,test_labels))
test_ds.     = test_ds.batch(32).astype("float32")/255
```

We instantiate a TensorFlow `model` object by declaring the layers in the model and then chaining them together. The following TensorFlow script shows how to do this for a simple sequential model with two convolutional layers. This model consists of 5 layers

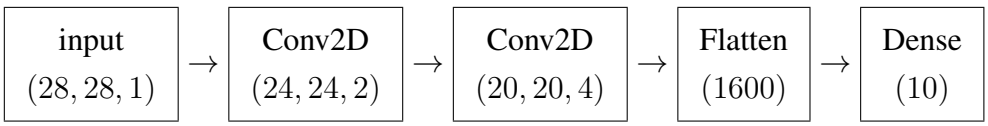
$$\text{input} \rightarrow \text{Conv2D} \rightarrow \text{Conv2D} \rightarrow \text{Flatten} \rightarrow \text{Dense}.$$

The input layer fixes the shape of the input tensors to $(28, 28, 1)$. The first two dimensions are called *spatial dimensions* and the third dimension is called a *channel*. For MNIST images, the channel only has 1 component because all images are monochrome. If these images had been RGB color images, then this third dimension would have 3 components. These inputs are fed to convolutional layers. Convolutional layers (`Conv2D`) are special layers that perform spatial convolutions on their input tensors. The output from these layers will be rank-3 tensors where the number of channels is specified as an argument in the layer's constructor. In general the spatial dimensions of this output tensor will be slightly smaller due to the convolution operation. These layers also specify an activation function, ReLu. The output of the convolutional layers is a rank-3 tensor, but the output of the model is going to be a rank-1 tensor of shape (10), one component for each of the 10 digits we want to recognize. The last two layers (`Flatten` and `Dense`) reshape the convolutional layer's outputs into the shape. The `Dense` layer is a fully connected layer similar to what we see in an MLP's hidden layer. In this example, we use a softmax activation function so the model outputs are real-valued numbers between 0 and 1.

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(28,28,1))
x = layers.Conv2D(filters=2, kernel_size=5, activation="relu")(inputs)
x = layers.Conv2D(filters=4, kernel_size=5, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10,activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs = outputs)
model.summary()
```

It is common practice to look at the shape of each layer's output. This is done to make sure the tensor shapes are correct. For this example, we therefore see the following sequence of tensor shapes



This particular model has a total of 16266 trainable weights.

We are now ready to train the model declared above using the dataset objects we created. Training is done by evaluating the gradient of the empirical risk evaluated for a mini-batch of data and then using that gradient to update the trainable weights. This procedure is done multiple times as part of a gradient search algorithm. Before conducting the gradient search, we need to configure the structures used to compute the gradient. This is done through the model's `compile` method.

```
model.compile(loss = "sparse_categorical_crossentropy",
           optimizer = "rmsprop",
           metrics = ["accuracy"])
```

The preceding `compile` command configures the optimizer used in computing the gradient. In this method we first declare which loss function we are using. Since this is a multi-class classification problem we use a variation of the binary crossentropy function called a *sparse categorical crossentropy* function. The optimizer determines how the computed gradient will be used to update trainable weights. Finally, the compile command declares which metrics will be used to assess how well model training is progressing. Recall that the gradient search tries to minimize the aggregated loss of the model on the training dataset. This loss, however, was chosen to facilitate training, it may not correspond to what the user might really be interested in optimizing. Classifiers are ultimately concerned with the prediction *accuracy*; the number of correct classifications divided by the total number of classifications made. So the last entry in the compile method specifies that it wants to keep track of training "accuracy" over the course of training. Training is done using the model's `fit` method. The following script shows how this method is invoked.

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="test_model.keras",
        save_best_only = True,
```

```
        monitor = "val_loss"
    )
]

history = model.fit(train_ds,
                    epochs=30,
                    validation_data = test_ds,
                    callbacks = callbacks)
```

Model training is done using the training dataset object, `train_ds`. Recall that this object pre-batched the first 5000 samples of the MNIST database into mini-batches of 32 samples. Each mini-batch is used to compute a gradient update to the weights. A *training epoch* corresponds to a complete pass through all of the training set's batches. This example trains the model for 30 epochs. At the end of each epoch, we use the testing dataset object, `test_ds`, to evaluate the model's aggregate loss and accuracy. At the end of each epoch we use a callback function to "save" the current model if the loss evaluated on the test data is smallest of all past evaluating testing losses. Finally, we save this the testing/training losses and accuracy for each epoch in the `history` object. The following script shows how the `history` object is used to monitor how well our training process went.

```
test_model = keras.models.load_model("test_model.keras")
test_loss, test_acc = test_model.evaluate(test_ds)

import matplotlib.pyplot as plt
train_loss = history.history["loss"]
val_loss = history.history["val_loss"]
train_acc = history.history["accuracy"]
val_acc   = history.history["val_accuracy"]
epochs = range(1, len(train_loss) + 1)

figure, axis = plt.subplots(1,2)
axis[0].plot(epochs, train_loss, "b--", label = "Training loss")
axis[0].plot(epochs, val_loss, "b", label = "Validation loss")
axis[0].set_title(f"Best Model Loss: {test_loss: .3f}")
axis[0].legend()
```

```
axis[1].plot(epochs, train_acc, "b--", label = "Training Accuracy")
axis[1].plot(epochs, val_acc, "b", label = "Validation Accuracy")
axis[1].set_title(f"Best Model Accuracy: {test_acc: .3f}")
axis[1].legend()
```

The preceding script first reloads to "best" model that was obtained during training. It then evaluates the loss and accuracy of that model on the testing dataset `test_ds`. The past `history` of training/testing loss/accuracy is stored in arrays and then plotted to show how these measures of training performance behaved over each epoch. The results are shown below in Fig. 4.

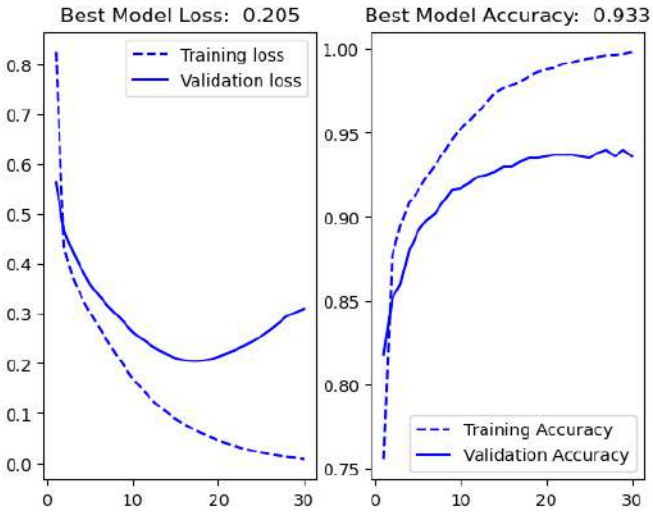


FIGURE 4. Training/Validation Loss/Accuracy as a function of epoch

Fig. 4 shows the *training curves* for this model. The left hand plot shows how accuracy changes over training epochs. The right hand plot shows how loss changes over training epochs. Let's look at the loss curves. We see that the training loss is a monotone decreasing function of epoch number. This is to be expected, since training uses follows the gradient of the aggregated loss function to reduce that loss after the update. The solid curve shows the testing (a.k.a. validation) loss. This was obtained by evaluating

the model's loss on the testing data, rather than the training data. The testing loss provides a measure of loss that is statistically independent of the training loss. In general, testing loss will be a more accurate estimate of the model's actual loss $R[h]$. What we see is that while the testing loss is initially decreasing, after about the 15th epoch the testing loss begins increasing. In other words, our training is no longer "generalizing" to data samples that were outside of its training dataset. We refer to this as *model overfitting*. It means, essentially, that the model has "memorized" the training data in such a way that it cannot recognize samples outside of that training set. Overfitting commonly occurs in training deep learning models and it is important to know how to detect when it occurs and what steps one might take to reduce its impact. The next set of lectures provide a statistical basis for explaining why overfitting occurs.

Remember that we had "saved" the model with the best testing loss as we trained it. Overfitting is the reason for doing this. At the end of training, we are left with the model at the 30th epoch. But what the training curve for loss shows is that we really should have stopped training around epoch 15 and used that as our final model. The callback we introduced during training allows us to do just that. We can then evaluate the actual testing loss/accuracy on that "best" model. The title of each plot in Fig. 4 shows those values. Now let us look at the accuracy curve. We see that training accuracy is a monotone increasing function of epoch that approach perfect accuracy. The testing accuracy is also monotone increasing but it tends to stop improving and reaches a maximum value of 0.93.

CHAPTER 2

Generalization - a statistical approach

Learning by example uses a finite dataset, $\mathcal{D}$, to select a model, h, from a model set, $\mathcal{H}$ that minimizes the empirical risk of that model on the dataset. The dataset, $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$ consists of ordered pairs of *inputs*, $x_k \in X$, and *targets*, $y_k \in Y$, that are drawn in an i.i.d. manner from a joint *system* probability distribution, $P(\mathbf{x}, \mathbf{y})$. The model, $h : X \to Y$, is selected from a *model set*, $\mathcal{H}$, containing models that map the input, x, onto a predicted target, $\widehat{y} = h(x)$. The learning-by-example problem selects an *optimal model*, h^* that minimizes the model's empirical risk on the given dataset $\mathcal{D}$,

$$\widehat{R}_{\mathcal{D}}[h] = \frac{1}{N} \sum_{k=1}^{N} L(y_k, h(x_k))$$

where $L : Y \times Y \to \mathbb{R}$ is a *loss function* representing the loss or error in using the predicted output $h(x_k)$ versus the true target y_k.

While our optimal model minimizes the empirical risk over the data set $\mathcal{D}$, we really want a model that minimizes the *actual risk*

$$R[h] = \mathbb{E}_{\mathbf{x},\mathbf{y}} \left[L(\mathbf{y}, h(\mathbf{x})] \right.$$

where $\mathbf{y}$ and $\mathbf{x}$ are random target/input variables with an *unknown* joint probability distribution. Since that distribution is unknown, however, we cannot evaluate the actual risk and can therefore not directly find the model minimizing $R[h]$. We do know that as the size N of the dataset goes to infinity, then the weak law of large numbers asserts the empirical risk $\widehat{R}_{\mathcal{D}}[h]$ converges with probability 1 to the actual risk. So the problem is to have a large enough dataset to ensure the risk difference, $|\widehat{R}_{\mathcal{D}}[h] - R[h]|$ is small.

Clearly it would be useful to know how the dataset's size, N, impacts the risk difference of the trained model.

Determining that relationship is complicated by the fact that the dataset, itself, is a random variable. This occurs because dataset samples are randomly generated in an i.i.d. manner. As a result the empirical risk, $\widehat{R}_{\mathcal{D}}[h]$ is also a random variable and so we cannot simply require that the risk difference $|\widehat{R}_{\mathcal{D}}[h] - R[h]|$ is *always* small. There will always be a chance that we selected a dataset for which the risk difference is large and so rather than requiring our model to "minimize" the risk difference, we select the model to limit the probability that the risk difference is large. If we can do this then we say that the model *generalizes beyond its training data*. Formally we say the model, h^*, generalizes beyond the training data if for any $\epsilon > 0$ there exists $\delta > 0$ such that

$$\Pr_{\mathcal{D}}\left\{\left|\widehat{R}_{\mathcal{D}}[h^*] - R[h^*]\right| \geq \epsilon\right\} \leq \delta$$

Such a model is also said to be *probably approximately correct* or PAC learnable. This statement says that the probability over all possible datasets of the risk difference being greater than ϵ is less than δ. The parameter ϵ is the model's generalization ability and $100\times(1-\delta)$ represents the confidence level of that bound on the generalization ability. In this chapter we study how this generalization ability varies as a function of dataset size N and the complexity or size of the model set $\mathcal{H}$. The results from our study will help explain the shape of the training curves in Fig. 4 in chapter 1.

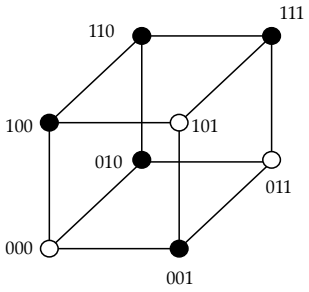
1. Infeasibility of Perfect Learning

This section uses a specific example to concretely illustrate the fact that we cannot *always* select a model that is "perfect" in the sense that minimizing the empirical risk also minimizes the actual risk. Let us consider a learning-by-example problem in which the system is modeled as a Boolean function, $f : \{0,1\}^3 \rightarrow \{0,1\}$ that maps a 3-d Boolean vector, $x \in \{0,1\}^3$, onto a binary class label $y \in \{\mathrm{T},\mathrm{F}\}$. Fig. 1 illustrates this *unknown* system

function as a 3-d hypercube whose vertices represent the 8 different inputs

$$X = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

The figure labels each vertex with one of the input vector. If the vertex is "solid", then the target for that input is TRUE. If the vertex is "open", then the target is FALSE. The hypercube shown in Fig. 1 therefore represents a specific Boolean function we are trying to learn a model for.



testing data training data

x	y	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
000	○	○	○	○	○	○	○	○	○
001	●	●	●	●	●	●	●	●	●
010	●	●	●	●	●	●	●	●	●
011	○	○	○	○	○	○	○	○	○
100	●	●	●	●	●	●	●	●	●
101	○	○	○	○	○	●	●	●	●
110	●	○	○	●	●	○	○	●	●
111	●	○	●	○	●	○	●	○	●

FIGURE 1. Perfect learning by example is not feasible

We are going to assume we have two datasets; a training dataset and a test dataset. The training dataset has 5 pairs of inputs and targets that agree with the mappings shown on the hypercube in Fig. 1. So the training dataset is

$$X_{\text{train}} = \{(000, F), (001, T), (010, T), (011, F), (100, T)\}$$

We also form a test dataset of 3 samples for the remaining vertices not in the training data

$$X_{\text{test}} = \{(101, F), (110, T), (111, T)\}$$

We now assume that we form a model set $\mathcal{H}$ consisting of Boolean functions that map all of the training inputs onto their correct training targets in X_{train}. There are 8 such Boolean functions that we denote as f_1, f_2, to f_8. The targets generated by these 8 models are shown in the table of Fig. 1.

The training data corresponds to the first 5 rows of the table and the testing data corresponds to the last 3 rows. Our unknown Boolean function shown in the hypercube is one of these 8 models. In particular, the model f_4 corresponds to the actual function. If we were to select f_4 from $\mathcal{H}$ then we minimize the empirical risk on the training data since all training samples are correctly classified and we also minimize the empirical risk on the testing data since all testing samples are also correctly classified.

The problem we have, however, is that since we can only use the training samples to select a model, there is no reason why we could not pick any of the other 7 models because those models also perfectly classify all training samples. If we look at the risk of these other models on the testing data, however, we see that the error rate ranges from 0% (for model f_4) to 100% (for model f_5). In other words, we have no rational way of selecting a model that is "always" perfect and so we must adopt a more sophisticated way of describing what it means for a model to be optimal. In particular, we want a way to ensure that a model with minimal risk on the training data also has a small risk on a set of testing data samples that are statistically independent of the training dataset samples.

2. PAC Learning

This impossibility of learning "perfect" models is a special case of a result establishing the unlearnability of regular languages from positive and negative examples [**Ang87**]. A common approach for dealing with impossible problems is to simply change the problem's solution concept. In our case, this means that instead of requiring a perfect model, we require a model that is probably approximately correct (PAC). This is referred to as PAC learning [**Val84**]. PAC learning requires that the *probability* is large (probably) for the model's prediction being within a small distance (approximately correct) of the actual target. It is this notion of PAC-learning that will allow

us to formally demonstrate that the learning-by-example problem is indeed tractable in a *statistical sense*.

To formally define the PAC concept, let us first review the problem statement we gave in chapter 1. The learning-by-example problem has three components. There is a *system* formed from the cascade connection of a *generator* and an *observer*. The generator draws an *input*, $x \in X$, in an i.i.d. manner from a generator distribution, $F_{\mathbf{x}}(x)$. The observer draws a *target*, $y \in Y$, in an i.i.d. manner from the observer's conditional distribution, $Q_{\mathbf{y}|x}(y|x)$. The training *data set*, $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$, is a finite set of samples consisting of ordered pairs of inputs, $x_k \in X$, and targets, $y_k \in Y$, drawn in an i.i.d. manner from the system's joint distribution $P_{\mathbf{x},\mathbf{y}}(x, y) \stackrel{\text{def}}{=} F_{\mathbf{x}}(x)Q_{\mathbf{y}|x}(y|x)$. We are going to confine our attention to a binary classification problem since the PAC concept is a bit easier to define for this problem. This means, therefore, that the target set, $Y = \{0, 1\}$, is binary.

The second component of our problem is a model set, $\mathcal{H}$, consisting of models, $h_w : X \to Y$, that are parameterized by a weight, $w \in W$. A model, $h_w \in \mathcal{H}$, is used to predict the system observer's output, $y \in Y$, in response to a given input $x \in X$. We use a loss function, $L : Y \times Y \to \mathbb{R}$ to characterize how close the model's prediction, $h_w(x)$, for a given $x \in X$ is to the target, $y \in Y$, selected by the system observer. Since we are confining our attention to a binary classification problem, the loss function is the classification error

$$L(y, h_w(x)) = \mathbb{1}\left\{y \neq h_w(x)\right\}.$$

which is 1 if the model disagrees with the true target and is zero otherwise. The average loss over a given data set, $\mathcal{D} = \{x_k, y_k\}_{k=1}^N$ drawn by the system, is called the *empirical risk function*. For a model $h \in \mathcal{H}$, this function is

$$\widehat{R}_{\mathcal{D}}[h] = \frac{1}{N}\sum_{k=1}^{N} \mathbb{1}(h(x_k) \neq y_k).$$

The *optimal model*, $h^* \in \mathcal{H}$, is then any model that minimizes the empirical risk, $\widehat{R}_{\mathcal{D}}[h]$ with respect to the given data set $\mathcal{D}$.

We are interested in characterizing how close the optimal model, h^*, is to minimizing the *actual risk function*,

$$R[h] = \mathbb{E}_{\mathbf{x},\mathbf{y}}\left\{\mathbb{1}(y \neq h(x))\right\}.$$

The actual risk is the expected value of the loss function with respect to the system's joint distribution, $P_{\mathbf{x},\mathbf{y}}(x, y)$. It is important to note that the training data set, $\mathcal{D}$, was selected at random by the system. This means that $\mathcal{D}$ is actually a random variable and so the empirical risk $\widehat{R}_{\mathcal{D}}[h^*]$ of the optimal model, h^*, will also be a random variable. So when we consider how close the optimal model's actual risk is to the empirical risk, we must consider this in a probabilistic sense.

We want to find models that not only minimize the empirical risk, but we also want the probability to be large for the model's empirical risk being "close" to the actual risk. We formalize this by saying that the model $h^* \in \mathcal{H}$ is *probably approximately correct* (PAC) if for any $\epsilon > 0$ there exists $\delta \in (0, 1)$ such that

$$\Pr_{\boldsymbol{\mathcal{D}}}\left\{\left|\widehat{R}_{\boldsymbol{\mathcal{D}}}[h^*] - R[h^*]\right| \geq \epsilon\right\} \leq \delta. \tag{6}$$

This inequality says that for any arbitrary tolerance level, ϵ, the probability of the risk difference,

$$\Delta R_{\mathcal{D}}[h] \stackrel{\text{def}}{=} \left|\widehat{R}_{\boldsymbol{\mathcal{D}}}[h^*] - R[h^*]\right|, \tag{7}$$

violating that tolerance is less than some small probability, $\delta \ll 1$. We also refer to the *risk difference* as the model's *generalization error*. What PAC learning says is that the probability of the generalization error being greater than ϵ is less than δ. We often refer to $100 \times (1 - \delta)$ as the confidence level of the bound. The following sections take a closer look at how we can determine this confidence level as a function of the chosen tolerance level, ϵ. We will first do this for finite model sets and then examine how it can be extended to model sets with an infinite number of models.

3. Concentration Inequalities

We are interested in determine the relation between δ and ϵ in the PAC condition of equation (6). The risk difference $\left|\widehat{R}_{\mathcal{D}}[h] - R[h]\right|$ in that equation is a *non-negative* random variable. This means we can use concentration inequalities from probability theory to find that relationship. These concentration inequalities are all "boosted" versions of the Markov inequality. This section reviews those concentration inequalities we will be using in the remainder of the chapter. The results are presented without formal proof since these proofs can be found in a variety of other references [**BLM13**].

The Markov inequality says that if $\mathbf{x}$ is a non-negative random variable, then for any $\epsilon > 0$ we have

$$\Pr\{\mathbf{x} \geq \epsilon\} \leq \frac{\mathbb{E}[\mathbf{x}]}{\epsilon} \tag{8}$$

This says, essentially, that $\mathbf{x}$ can be seen as concentrating around its mean value. To justify this assertion, let us take $\epsilon = \delta\mathbb{E}[\mathbf{x}]$ so that the Markov inequality becomes

$$\Pr\{\mathbf{x} \geq \delta\mathbb{E}[\mathbf{x}]\} \leq \frac{1}{\delta}$$

So if we let δ get large, then the probability that $\mathbf{x}$ takes values far away from its mean value gets arbitrarily small.

The bound implied by the Markov inequality, however, is relatively loose which limits its usefulness. Concentration inequalities provide a way to tighten that bound by using a *boosting function*, $\phi : X \to \mathbb{R}^+$, that is non-decreasing on X such that $\mathbb{E}\{|\phi(\mathbf{x})|\}$ is finite. The random variable $\phi(\mathbf{x})$ will also be non-negative and because ϕ is nondecreasing we can readily see that

$$\Pr\{\mathbf{x} \geq \epsilon\} \leq \Pr\{\phi(\mathbf{x}) \geq \phi(\epsilon)\}$$

If we then apply the Markov inequality to $\phi(\mathbf{x})$, we can conclude

$$\Pr\{\mathbf{x} \geq \epsilon\} \leq \frac{\mathbb{E}[\phi(\mathbf{x})]}{\phi(\epsilon)} \tag{9}$$

If the boosting function $\phi(\epsilon) = \epsilon$ is linear, then this is the original Markov inequality. But if we choose the boosting function to increase at a rate that is more than linear then we can tighten the right hand side of the preceding bound.

To see how we might use this idea, let's consider a boosting function, $\phi(\epsilon) = \epsilon^2$ that grows at a quadratic rate, rather than a linear rate. If we then consider the non-negative random variable

$$\mathbf{y} = |\mathbf{x} - \mathbb{E}[\mathbf{x}]|$$

then applying the boosted Markov inequality in equation (9) implies

$$\Pr\{|\mathbf{x} - \mathbb{E}[\mathbf{x}]| \geq \epsilon\} = \Pr\{\mathbf{y} \geq \epsilon\} \leq \frac{\mathbb{E}[\mathbf{y}^2]}{\epsilon^2} = \frac{\text{var}(\mathbf{x})}{\epsilon^2} \tag{10}$$

This last equation is called the *Chebyshev inequality*.

We can use the Chebysev inequality to prove the weak law of large numbers (WLLN). WLLN says that if we have a sequence $\{\mathbf{x}_k\}_{k=1}^N$ of N random variables drawn in an i.i.d. manner with $\mathbb{E}[|\mathbf{x}_k|] < \infty$ and $\text{var}(\mathbf{x}_k) < \sigma^2$, then the sample mean of this sequence is a random variable

$$\mathbf{z}_N = \frac{1}{N}\sum_{k=1}^{N} \mathbf{x}_k$$

that converges in probability to the expected value of $\mathbf{z}_N$ as $N \to \infty$.

The proof of the WLLN will be somewhat useful to us, so I walk through it below. First observe that

$$\mathbb{E}[\mathbf{z}_N] = \frac{1}{N}\sum_{k=1}^{N} \mathbb{E}[\mathbf{x}_k], \quad \text{var}(\mathbf{z}_N) = \frac{1}{N^2}\sum_{k=1}^{N} \text{var}(\mathbf{x}_k)$$

So we have

$$\begin{aligned}\Pr\left\{|\mathbf{z}_N - \mathbb{E}[\mathbf{z}_N]| \geq \epsilon\right\} &= \Pr\left\{\left|\frac{1}{N}\sum_{k=1}^{N}\mathbf{x}_k - \frac{1}{N}\sum_{k=1}^{N}\mathbb{E}[\mathbf{x}_k]\right| \geq \epsilon\right\} \\ &= \Pr\left\{\left|\frac{1}{N}\sum_{k=1}^{N}\mathbf{x}_k - \frac{1}{N}\sum_{k=1}^{N}\mathbb{E}[\mathbf{x}_k]\right|^2 \geq \epsilon^2\right\} \\ &\leq \sum_{k=1}^{N}\frac{\mathrm{var}(\mathbf{x}_k)}{N\epsilon^2} \leq \frac{\sigma^2}{N\epsilon^2}\end{aligned}$$

where the last inequality comes from the Chebyshev inequality.

So for any ϵ we choose, we have a $\delta(\epsilon) = \frac{\sigma^2}{N\epsilon^2}$ that goes to zero as $N \to \infty$, which is sufficient to to establish $\lim_{N\to\infty}\Pr\left\{|\mathbf{z}_N - \mathbb{E}(\mathbf{z}_n)| \geq \epsilon\right\} = 0$. This proof says that $\mathbf{z}_N$ clusters around its average as $N \to \infty$. But what is of concern to us is the rate at which that clustering occurs. From the proof we see the rate of convergence is governed by a $1/N$ term that we may find too slow to be of practical value. So we will try to obtain a faster rate of convergence on the bound by using an exponential (rather than quadratic) boosting function.

Let us consider the exponential boosting function

$$\phi(\epsilon) = e^{\lambda x}$$

where $\lambda > 0$. Using the same ideas as before we can say that

$$|\mathbf{x} - \mathbb{E}[\mathbf{x}]| \geq \epsilon \quad \Longleftrightarrow \quad e^{\lambda|\mathbf{x}-\mathbb{E}[\mathbf{x}]|} \geq e^{\lambda\epsilon}$$

which would imply that

$$\Pr\left\{|\mathbf{x} - \mathbb{E}[\mathbf{x}]| \geq \epsilon\right\} = \Pr\left\{e^{\lambda|\mathbf{x}-\mathbb{E}[\mathbf{x}]|} \geq e^{\lambda\epsilon}\right\}$$

We now apply the Markov inequality to deduce that

$$\Pr\left\{|\mathbf{x} - \mathbb{E}[\mathbf{x}]| \geq \epsilon\right\} \leq \frac{\mathbb{E}[e^{\lambda|\mathbf{x}-\mathbb{E}[\mathbf{x}]|}]}{e^{\lambda\epsilon}} \tag{11}$$

which gives us the concentration inequality known as the *Chernoff bound*.

The Chernoff bound can be used to to derive another concentration inequality for bounded random variables that is critical for establishing bounds on the generalization ability in learning-by-example problems. This concentration inequality is known as the *Hoeffding inequality* and we state it below without proof.

THEOREM 1. **Hoeffding Inequality:** *Let* $\mathbf{x}_k$ *be an i.i.d. random variable with* $\mathbf{x}_k \in [a, b]$ *for all* k*, then*

$$\Pr\left\{\frac{1}{N}\left|\sum_{k=1}^{N}(\mathbf{x}_k - \mathbb{E}[\mathbf{x}_k])\right| \geq \epsilon\right\} \leq \exp\left(\frac{-2N\epsilon^2}{(b-a)^2}\right) \tag{12}$$

The importance of the Hoeffding inequality rests on the fact that its convergence rate scales as e^{-2N} rather than $1/N$ as suggested by the WLLN. So the bounds we need to compute on the risk difference will get exponentially tighter as the dataset size $N \to \infty$, rather than the $1/N$ convergence rate suggested in our proof for the WLLN.

4. Generalization Ability of Finite Model Sets

This section derives the (ϵ, δ) bounds in the PAC learning equation (6) when the model set $\mathcal{H}$ has a finite number, M, of models. These bounds will be derived using the concentration inequalities presented in the previous section.

The bounds we derive below are probabilistic because the dataset $\mathcal{D}$ is a random variable. We will find it convenient to provide some notational conventions used in characterizing such random variables (see appendix). The dataset random variable $\mathcal{D}$ is defined with respect to a probabiity space, $(\Omega, \mathcal{F}, \mu)$ where Ω is a set of outcomes, $\mathcal{F}$ is a sigma-algebra on Ω, and $\mu : \Omega \to [0, 1]$ is a probability measure. Each element ω of Ω is called an *outcome*. The dataset random variable, $\mathcal{D} : \Omega \to \Delta_N$ that maps each outcome, ω, onto a particular dataset, where we let Δ_N denote the set of all possible datasets $\{(x_k, y_k)\}_{k=1}^N$ with N samples. We define an *event* as

any subset of Ω consistent with the sigma algebra $\mathcal{F}$. An event $A \subset \mathcal{F}$ generated by a random variable $\mathcal{D}$ may therefore be written as

$$A = \{\omega \in \Omega \,:\, \text{some condition on } \mathcal{D}(\omega)\}$$

where $\mathcal{D}(\omega)$ is the specific value that the random variable $\mathcal{D}$ takes with respect to outcome ω. For example if A is an event that occurs when the risk difference of model $h \in \mathcal{H}$ over a randomly chosen dataset, $\mathcal{D}$, is less than a specified ϵ then we could write

$$A(h) = \left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h] < \epsilon\right\}$$

where the event is a function of the given model h. The probability of that event occurring would then be written as $\mu(A(h))$.

With these notational conventions in hand and recognizing that model h's risk difference, $\Delta R_{\mathcal{D}}[h] = |\widehat{R}_{\mathcal{D}}[h] - R[h]|$, is a non-negative random variable, a direct application of the Chebyshev inequality (10) allows us to conclude that

$$\mu\left(\left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h] \geq \epsilon\right\}\right) \leq \frac{\text{var}(\mathbf{y} \neq h(\mathbf{x}))}{N\epsilon^2} \leq \frac{1}{N\epsilon^2} \tag{13}$$

Since this probability is inversely proportional to dataset size, N, we can see that as the dataset size grows, the likelihood of having a large risk difference goes to zero. This inequality states that for any $\epsilon > 0$ we can choose $\delta = \frac{1}{N\epsilon^2}$ to ensure the PAC learnability condition in equation (6) is satisfied.

The generalization error in equation (13) provides a probabilistic upper bound on the model's, h, true risk, $R[h]$. Letting $\delta = \frac{1}{N\epsilon^2}$, we can can solve for ϵ (the generalization error) as a function of N and δ and then can assert

$$\widehat{R}_{\mathcal{D}}[h] - \sqrt{\frac{1}{N\delta}} \leq R[h] \leq \widehat{R}_{\mathcal{D}}[h] + \sqrt{\frac{1}{N\delta}}$$

with a confidence level of $100 \times (1 - \delta)$. This equation sandwiches the true risk, $R[h]$, between two different terms derived from the empirical risk, $\widehat{R}_{\mathcal{D}}[h]$, that we computed on the given dataset. Note that as the dataset size increases, the difference between the two bounds goes to zero, thereby

suggesting that for large datasets we can use $\widehat{R}_{\mathcal{D}}[h]$ as an estimate of the true risk.

We define a model set's *sample complexity* as the dataset size, $N_{\epsilon,\delta}$, required to ensure a specified generalization error, ϵ with desired confidence level, $100 \times (1-\delta)$. From equation (13) we have $\frac{1}{N\epsilon^2} \leq \delta$ so the sample complexity is bounded as

$$N_{\epsilon,\delta} \geq \frac{1}{\delta\epsilon^2} \tag{14}$$

The bound we computed in equation (13), however, is not exactly what we want. This equation characterizes the PAC learnability of any model $h \in \mathcal{H}$. But we are really interested in PAC-learnability of a specific model in $\mathcal{H}$, namely the model h^* that minimizes the empirical risk with respect to a specific dataset, say $\mathcal{D}^*$. For convenience let us define the following events,

$$\begin{aligned} A_\epsilon(h) &= \left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h] < \epsilon\right\} \\ &= \left\{\begin{array}{l}\text{Event for Model } h\text{'s risk difference being } \textit{less} \text{ than } \epsilon \\ \text{on any randomly selected dataset}\end{array}\right\} \\ A_\epsilon^c(h) &= \left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h] \geq \epsilon\right\} \\ &= \left\{\begin{array}{l}\text{Event for Model } h\text{'s risk difference being } \textit{greater} \\ \text{than } \epsilon \text{ on any randomly selected dataset}\end{array}\right\} \end{aligned}$$

The following clearly holds for h^*

$$A_\epsilon(h^*) \supseteq \bigcap_{h\in\mathcal{H}} A_\epsilon(h) = \left\{\begin{array}{l}\text{Events where risk difference} \\ \text{is less than } \epsilon \text{ for all } h \in \mathcal{H}\end{array}\right\}$$

The complement of this event is

$$A_\epsilon^c(h^*) \subset \left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h^*] \geq \epsilon\right\} = \bigcap_{h\in\mathcal{H}} A_\epsilon^c(h)$$

Using the fact that $\mathcal{H} = \{h_1, h_2, \ldots, h_M\}$ is finite with M models, and applying the probability measure μ to the above events gives

$$\begin{aligned}\mu(A_\epsilon^c(h^*)) &= \left\{\begin{matrix}\text{Probability that optimal model's risk} \\ \text{difference is } \textit{greater} \text{ than } \epsilon\end{matrix}\right\} \\ &\leq \mu\left(\bigcap_{k=1}^{M} A_\epsilon^c(h_k)\right) \leq \sum_{k=1}^{M} \mu(A_\epsilon^c(h_k))\end{aligned}$$

where the last inequality was obtained Boole's inequality (aka union bound)[1].

The terms in the summation are probabilities that we bounded in equation (13), so using those bounds above we get

$$\begin{aligned}\mu(A_\epsilon^c(h^*)) &= \mu\left(\left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h^*] \geq \epsilon\right\}\right) \\ &\leq \sum_{k=1}^{M} \mu\left(\left\{\omega \in \Omega \,:\, \Delta R_{\mathcal{D}(\omega)}[h_k] \geq \epsilon\right\}\right) \\ &\leq \sum_{k=1}^{M} \frac{1}{N\epsilon^2} = \boxed{\frac{M}{N\epsilon^2}}\end{aligned}$$

The right hand side is the δ needed to establish that h^* is PAC learnable. The sample complexity of the optimal model is therefore

$$N_{\epsilon,\delta} \geq \frac{M}{\delta\epsilon^2}$$

and the optimal model's true risk can be sandwiched between two bounds formed from the model's empirical risk as

$$\widehat{R}_{\mathcal{D}}[h^*] - \sqrt{\frac{M}{N\delta}} \leq R[h^*] \leq \widehat{R}_{\mathcal{D}}[h^*] + \sqrt{\frac{M}{N\delta}}$$

with a confidence of $100 \times (1 - \delta)$.

The PAC learnability bounds established above using the Chebyshev inequality have limited value. If we require $\mu(A_\epsilon^c(h^*)) \ll 1$ to be small, then this requires $N \gg M$. In other words we must have many many more dataset samples than models in $\mathcal{H}$. Moreover while the bound on $R[h^*]$

[1] **Boole's Inequality:** Let A and B be two events in probability space $(\Omega, \mathcal{F}, \mu)$, then $\mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B) \leq \mu(A) + \mu(B)$.

goes to zero as dataset size, N goes to infinity, the rate of convergence is very slow, thereby limiting the use of these bounds to only very small model sets. For this reason we try to find a stronger bound on the PAC generalization errors in equation (**??**) by using a concentration inequality based on an exponential bound. In particular, we will use the Hoeffding inequality that was introduced in the preceding section.

An application of the Hoeffding inequality (rather than the Chebyshev inequality) gives us the following probability for event $A_\epsilon^c(h)$,

$$\mu\left(A_\epsilon^c(h)\right) \;=\; \Pr_{\mathcal{D}}\left\{\Delta R_{\mathcal{D}}[h] \geq \epsilon\right\} \leq 2e^{-2N\epsilon^2}$$

Using the same arguments for bounding the probability of $A_\epsilon^c(h^*)$ for the optimal model we get

$$\mu(A_\epsilon^c(h^*)) = \Pr_{\mathcal{D}}\left\{\left|\widehat{R}_{\mathcal{D}}[h^*] - R[h^*]\right| \geq \epsilon\right\} \leq 2Me^{-2N\epsilon^2}$$

If we then take $\delta = 2Me^{-2N\epsilon^2}$, then this again establishes the PAC learnability of the optimal model. The difference lies in our bounds for the sample complexity and the error in estimating the true risk. The sample complexity obtained using the Hoeffding bound now becomes

$$N_{\epsilon,\delta} \geq \frac{1}{2\epsilon^2}\log\left(\frac{2M}{\delta}\right)$$

So that the dataset size needed to achieve (ϵ, δ) learnability scales with $\log M$, rather than M, so our dataset sizes need not be as large. The true risk is bounded as

$$\widehat{R}_{\mathcal{D}}[h^*] - \sqrt{\frac{1}{2N}\log\frac{2M}{\delta}} \leq R[h^*] \leq \widehat{R}_{\mathcal{D}}[h^*] + \sqrt{\frac{1}{2N}\log\frac{2M}{\delta}}$$

with confidence $100 \times (1-\delta)$. We now see that if we use the empirical risk to estimate the true risk we need a much smaller testing set since again this bound scales as $\log M$ rather than M.

The preceding analysis was relevant to finite model sets, but obviously neural network model sets are not going to be finite since the weights lie in a vector (tensor) space. So we need to find a way to modify the preceding

analysis so it can be used on infinite model sets. The trick we will use is to change the way we count the size of the model set and develop an alternative measure of the model set's richness or complexity based on a quantity known as the Vapnik-Chervonenkis (VC) dimension.

5. Growth Function for Infinite Model Sets

The PAC learnability conditions from the preceding sections assumed the model set $\mathcal{H}$ had a finite number of models. This will not be the case for neural network model sets whose weight vectors lie in a dense vector space for these model sets contain an uncountably infinite number of models. To bound the generalization ability of such model sets, we will need to alternative measures of model set complexity that are inherently finite in nature.

The binary classification problem in Fig. 2 shows that model set cardinality overcounts a model set's complexity. The figure shows an input space, $\mathbb{R}^2$, where we have marked the inputs of the dataset, $\mathcal{D}$. The inputs shown as solid bullets have target label, $+1$, and the inputs shown as open bullets have target label, -1. We assume the model set $\mathcal{H}$ has three models whose weight vectors, w_1, w_2, and w_3 form the blue hyperplanes shown in Fig. 2. Observe that two of these classifiers (h_{w_2} and h_{w_3}) misclassify the same number of dataset inputs and so these tow models have the same empirical risk. This observation suggests that the number of models (in this case 3) overestimates the richness of the model set since 2 of these models have the same empirifal risk.

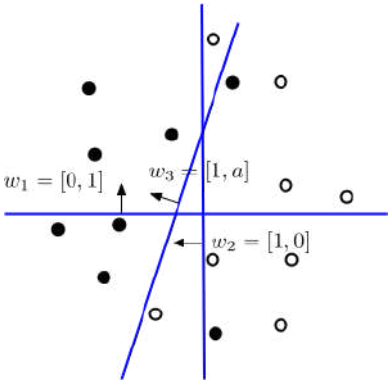


FIGURE 2. Two of these models have the same empirical risk

Fig. 2 suggests that an alternative way of measuring the model set's richness is to ground that measure in the classification task. In particular, that

richness may be better measured by counting the number of models in $\mathcal{H}$ that lead to *distinct* labelings of the dataset, rather than using the cardinality of $\mathcal{H}$. This idea is sometimes referred to a *machine capacity* [**Cov64**].

We now formalize what we mean by counting up the distinct labeling of a model set. Let $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ denote a data set with inputs $x_k \in X \subset \mathbb{R}^n$ and targets $y_k \in \{-1, +1\}$. Because these samples were drawn at random, we treat the dataset, $\mathcal{D}$, as a random variable. The model set, $\mathcal{H}$, consists of functions $h : X \to \{-1, +1\}$ that maps inputs $x_k \in X$ onto either -1 or $+1$. We define a *labeling* of the data set $\mathcal{D}$ by a model $h \in \mathcal{H}$ as the set

$$\text{lable}(h, \mathcal{D}) \stackrel{\text{def}}{=} \{h(x_k)\}_{k=1}^N$$

The set of all possible ways the given model can "classify" the dataset samples is called a *dichotomy* of the samples. The set of all such dichotomies of $\mathcal{D}$ generated by all models in $\mathcal{H}$ is then denoted as

$$\text{dichotomy}(\mathcal{H}, \mathcal{D}) \stackrel{\text{def}}{=} \bigcup_{h \in \mathcal{H}} \text{label}(h, \mathcal{D})$$

What we propose to do is use the maximum number of distinct dichotomies as a measure of the model set's richness. In particular, if we let Δ_N denote the collection of all possible datasets with N samples, then the *growth function* of model set $\mathcal{H}$ is the function $m_{\mathcal{H}} : \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$ such that for any positive integer N

$$m_{\mathcal{H}}(N) \stackrel{\text{def}}{=} \max_{\mathcal{D} \in \Delta_N} |\text{dichotomy}(\mathcal{H}, \mathcal{D})|$$

In other words $m_{\mathcal{H}}(N)$ denotes the maximum number of distinct labelings of any dataset of N samples that can be generated by any model in $\mathcal{H}$. We call this the *growth function* of the model set since it counts the number of distinct labelings generated by the model set as the size of the dataset, N, grows.

To better illustrate what the growth function measures, let us consider a specific example. In this example the model set, $\mathcal{H}$, is formed form linear

classifiers solving a binary classification problem with inputs in $\mathbb{R}^2$. Let us consider a dataset with $N = 3$ samples. These datasets fall into one of two groups; either all inputs are collinear or they are not collinear. Let Δ_{lin} denote the set of all datasets whose 3 input samples are collinear. Fig. **??** shows that =there are only 6 ways these three sample inputs can be partitioned into two classes by a linear classifier. This means, therefore that

$$|\text{dichotomy}(\mathcal{H}, \mathcal{D}_{\text{lin}})| = 6$$

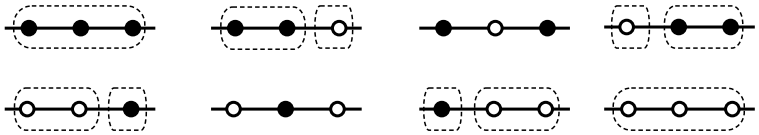


FIGURE 3. Growth function example showing there are only 6 ways that a linear classifier can partition 3 collinear input samples in $\mathbb{R}^2$ into two classes.

Now let us turn our attention to the next group of datasets, $\Delta_{\text{no-lin}}$, whose input samples are not collinear. Fig. 4 shows that there $2^3 = 8$ possible ways of partitioning three non-collinear inputs into two classes using a linear classifier. This means, therefore that

$$|\text{dichotomy}(\mathcal{H}, \mathcal{D}_{\text{no-lin}})| = 8$$

So for datasets of size $N = 3$, the growth function will be the maximum cardinality of the dichotomy sets for the collinear and non-collinear groups. In other words we have

$$m_{\mathcal{H}}(3) = \max_{\mathcal{D} \in \Delta_{\text{lin}} \cup \Delta_{\text{no-lin}}} |\text{dichotomy}(\mathcal{H}, \mathcal{D})| = \max\{6, 8\} = 8$$

Similar logic can be used to evaluate the growth function for a dataset of 4 inputs samples in $\mathbb{R}^2$. There are three cases; all points aligned, 3 out of 4 aligned, and no 3 points out of 4 are in a line. Using the same logic as before, one can deduce that if all 4 points are aligned there should be 10 distinct labelings. If three out of four points are aligned then there are 12 distinct labelings. If no 3 out of 4 points are aligned then there are 14

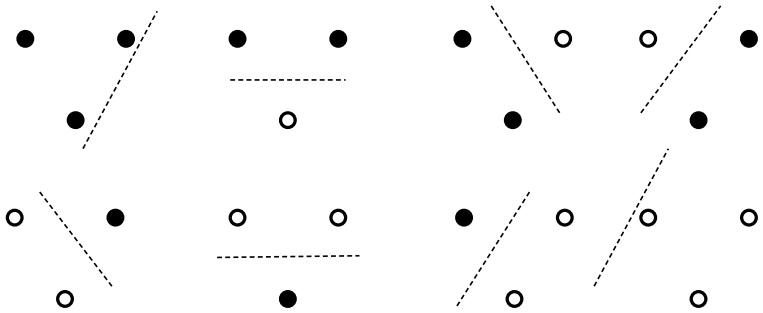


FIGURE 4. Growth function example showing there are 8 ways that a linear classifier can partition 3 non collinear input samples in $\mathbb{R}^2$ into two classes.

distinct labelings. We can therefore conclude that $m_{\mathcal{H}}(4) = 14$. This type of reasoning, however, is a task of exponentially growing complexity. That complexity means that actually computing the growth function for a model set is, in general, a computationally intractable problem. So rather than seeking an explicit way of evaluating $m_{\mathcal{H}}$, we will try to identify computationally tractable bounds on the growth function using a quantity called the Vapknik-Chervonenkis or VC dimension.

We define the VC dimension of model set $\mathcal{H}$ as follows. We say that a model set $\mathcal{H}$ *shatters* a data set $\mathcal{D}$ if it can distinctly label *all dichotomies* of the data set $\mathcal{D}$ of size N. If no data set of size k can be shattered by model set $\mathcal{H}$, then k is called a *break point* for $\mathcal{H}$. So if k is a break point then any $\ell > k$ is also a break point of the model set. The Vapnik-Chervonenkis or VC dimension of model set, $\mathcal{H}$, is the largest value of N that can be shattered by any model in the model set $\mathcal{H}$. We denote the VC dimension of $\mathcal{H}$ as $d_{\text{vc}}(\mathcal{H})$. Note that since there are most 2^N distinct dichotomies of a data set with N points, this means that $d_{\text{vc}}(\mathcal{H})$ is also equal to that value of N for which $m_{\mathcal{H}}(N) = 2^N$. We can also readily see that $d_{\text{vc}} + 1$ is the smallest break point of the model set.

To make the preceding definition more concrete, let us return to the earlier example that used a linear classifier on inputs in $\mathbb{R}^2$. That example

showed that there are $2^3 = 8$ possible dichotomies of data sets with size $N = 3$ points. Since $m_{\mathcal{H}}(3) = 8 = 2^3$, we know that 3 is not a break point for this model set since the data set is shattered by some model . On the other hand, for a data set of size $N = 4$, there were $2^4 = 16$ distinct dichotomies, but we argued that only 14 of these could be generated by a linear classifier. This means that $k = 4$ is the smallest break point of the model set of 2-d linear classifiers which means the VC dimension of this model set is 3. So while this model set has an infinite number of models in it, it has a finite VC dimension. We now show how that finite VC dimension can be used to bound the model set's growth function.

One can show using a combinatoric inequality known as Sauer's lemma that if k is a break point for the model set then

$$m_{\mathcal{H}}(N) \leq \sum_{i=0}^{k-1} \binom{N}{i}$$

where N is the size of the data set. Since $d_{\text{vc}} + 1$ is the smallest break point for the model set, and because the right hand side is a $k - 1$st order polynomial function of N, we can readily establish that

$$m_{\mathcal{H}}(N) \leq N^{d_{\text{vc}}} + 1$$

thereby showing that the growth function (i.e. model set richness) is bounded by a polynomial function of the data set size, N. Moreover, we can see that the largest exponent on that polynomial bound is d_{vc}, suggesting that VC dimension can be taken as a measure of a model set's richness.

6. Generalization Ability of Infinite Model Sets

Recall that we showed for any *finite* model set, $\mathcal{H}$ that

$$\Pr_{\boldsymbol{\mathcal{D}}}\left(\left|\widehat{R}_{\boldsymbol{\mathcal{D}}}[h^*] - R[h^*]\right| > \epsilon\right) \leq 2|\mathcal{H}|e^{-2N\epsilon^2}$$

where N was the set of the data set $\mathcal{D}$ and $h^* \in \mathcal{H}$ minimized the empirical risk $\widehat{R}_{\mathcal{D}}[h]$. In this inequality we used model set's cardinality, $|\mathcal{H}|$, as a measure of model set richness. But from the preceding discussion we

also know that a tighter way of estimating the richness of $\mathcal{H}$ is through the growth function. This suggests that we should be able to obtain a bound replacing $|\mathcal{H}|$ with $m_{\mathcal{H}}(N)$.

A more careful derivation of this bound [**Vap98, SSBD14**] for infinite model sets shows the inequality is actually

$$\Pr_{\boldsymbol{\mathcal{D}}}\left(\left|\widehat{R}_{\boldsymbol{\mathcal{D}}}[h^*] - R[h^*]\right| > \epsilon\right) \leq 4m_{\mathcal{H}}(2N)e^{-\frac{1}{8}\epsilon^2 N}.$$

If the model set $\mathcal{H}$ has a finite VC-dimension then $m_{\mathcal{H}}(N) \leq N^{d_{\text{vc}}} + 1$ and so this generalization inequality can be written as

$$\Pr_{\boldsymbol{\mathcal{D}}}\left(\left|\widehat{R}_{\boldsymbol{\mathcal{D}}}[h^*] - R[h^*]\right| > \epsilon\right) \leq 4((2N)^{d_{\text{vc}}} + 1)e^{-\frac{1}{8}\epsilon^2 N}. \tag{15}$$

From the preceding bound (15) we can readily show that the model set's sample complexity must satisfy

$$N_{\epsilon,\delta} \geq \frac{8}{\epsilon^2}\log\left(\frac{4((2N_{\epsilon,\delta})^{d_{\text{vc}}} + 1)}{\delta}\right)$$

We can also show that the true risk of the optimal model h^* can be bounded as follows with confidence $100 \times (1 - \delta)$.

$$\widehat{R}_{\mathcal{D}}[h^*] - \Omega(N, d_{\text{vc}}, \delta) \leq R[h^*] \leq \widehat{R}_{\mathcal{D}}[h^*] + \Omega(N, d_{\text{vc}}, \delta)$$

where

$$\Omega(N, d_{\text{vc}}, \delta) = \sqrt{\frac{8}{N}\log\left(\frac{4((2N)^{d_{\text{vc}}} + 1)}{\delta}\right)}$$

We can think of $\Omega(N, d_{\text{vc}}, \delta)$ as the *generalization error* , $epsilon$, of the model set. Both of these bounds are significantly tighter than the bound based on the model set's cardinality M. Let us now consider a couple of example demonstrating how these bounds might be used in practice.

Suppose a data set of size $N = 100$. What generalization error can we achieve with 90% confidence if the model set has a VC dimension off 1?

From our earlier bounds we can see the generalization error is

$$\begin{aligned}\epsilon &= \sqrt{\frac{8}{N}\log\frac{4((2N)^{d_{\text{vc}}}+1)}{\delta}} \\ &= \sqrt{\frac{8}{100}\log\left(\frac{4(201)}{1-.9}\right)} = 0.848\end{aligned}$$

So the actual risk of the optimal model h^* can be bounded as

$$R[h^*] \leq \widehat{R}_{\mathcal{D}}[h^*] + 0.848$$

with a 90% confidence level. Note that this is a poor generalization error since it will be close to 1 if $R[h^*] = 0$. To do better we would need to increase the size of the dataset. In particular, if we repeat the above computation with a dataset of size $N = 10^4$, then the true risk and empirical risk would be within 0.1 of each other.

As a second example, let us consider a model set with VC dimension $d_{\text{vc}} = 3$. How big must the dataset be to achieve a generalization error $\epsilon < 0.1$ with a confidence o 90%? This can be answered by finding the sample complexity, $N_{\epsilon,\delta}$ of the model set. Our equation for the model set, however, says

$$N_{\epsilon,\delta} \geq \frac{8}{0.1^2}\log\left(\frac{4(2N_{\epsilon,\delta})^3+4}{0.1}\right)$$

Note that $N_{\epsilon,\delta}$ is on both sides where it appears through the $\log$ function on the righthand side. This makes it difficult to solve for $N_{\epsilon,\delta}$ in closed form. So we adopt a recursive strategy that uses the method of successive approximation to find $N_{\epsilon,\delta}$. This method computes a sequence $\{N_k\}_{k=0}^{\infty}$ of approximations that converge to $N_{\epsilon,\delta}$ as $k \to \infty$. If we are given the kth estimate, N_k, of the sample complexity, we compute the $k+1$ element of the sequence as

$$N_{k+1} = \frac{8}{\epsilon^2}\log\left(\frac{4((2N_k)^{d_{\text{vc}}}+1)}{\delta}\right)$$

In the limit as $k \to \infty$, we have $N_k \to N_{\epsilon,\delta}$.

Using this recursive strategy on our example we first let $N_0 = 1000$ and compute

$$N_1 = \frac{8}{0.1^2} \log\left(\frac{4(2000)^3 + 4}{0.1}\right) = 21193$$

The next element of the sequence is

$$N_2 = \frac{8}{0.1^2} \log\left(\frac{4((2(21193))^3 + 4}{0.1}\right) = 28522$$

After a couple more recursions we get a value of about $N_{\epsilon,\delta} \approx 29299$. So we would need about 30 thousand samples in the dataset. It is interesting to see how these sample complexity bounds scale with the VC dimension. If we compute $N_{\epsilon,\delta}$ in this example for a range of v_{dc}, we get the following table. This bound therefore suggests that the number of samples is approxi-

d_{vc}	1	2	3	4	5	6
$N_{\epsilon,\delta}$	10946	19897	29299	38997	48915	59008

mately proportional to the VC dimension with a proportionality constant of 10^4. This is not what has been observed in practice. In particular, [**AM12**] suggests this proportionality constant should be closer to 10.

When a model set has finite VC dimension, then we can use it to bound the model set's generalization error. T One model set for which the VC dimension is a useful bound is the class of *linear machines* (i.e. perceptrons). The VC dimension for a linear classifier with inputs in $\mathbb{R}^n$ is $n + 1$, essentially the number of parameters. One can show that a linear machine known as a support vector machine can have a small VC dimension. So this suggests that if our problem can be solved using linear machines, then we have useful analytical tools for estimating the generalization error.

VC dimension, however, does not appear to be as useful in bounding the generalization ability of deep neural networks [**ZBH+21**]. Deep neural networks with ReLu activations have a VC dimension of $O(WL\log(W))$ where W is the number of weights and L is the number of layers [**BHLM19**].

This suggests that deep neural networks have an extremely large VC dimension and based on the preceding example, one might expect this model class to generalize poorly. This is, in fact, not really what has been observed in practice [**ZBH+21**].

In training neural network models it is common practice to take the dataset, $\mathcal{D}$, and split it into a *training set*, $\mathcal{D}^{\text{train}}$ and a *testing set*, $\mathcal{D}^{\text{test}}$. We select an optimal model h^* that minimizes the empirical risk on the training data, $\mathcal{D}^{\text{train}}$, and then we evaluate the empirical risk of that model on the testing data, $\mathcal{D}^{\text{test}}$. The basic idea is that the testing data's empirical risk will be a good approximation of the optimal model's true risk. As we will discuss in the next section, VC theory predicts that if we plot test and training risk as a function of VC dimension, these curves should have the shape shown on the lefthand plot of Fig.5. The U-shaped testing risk suggests there is an optimal complexity to the model set and the tradeoff needed to find that optimal model set is called the bias-variance tradeoff. In practice, however, we often see the double descent curve shown on the right hand side of Fig. 5. This curve shows the test risk begin decreasing after a certain point and acheiving test risks that are smaller than what were predicted using the VC theory. This double descent phenomenon is why deep neural networks work much better as the models get deeper and more complex. It is not predicted by statistical learning theory, but is a well acknowledged aspect of these models in practice. Why this should be the case is still an open research question [**BHMM19**, **LC22**].

7. Bias-Variance Tradeoff and Early Stopping

VC theory predicts that the optimal model's true risk can be bounded above as

$$R[h^*] \leq \widehat{R}_{\mathcal{D}}[h^*] + \Omega(N, d_{\text{vc}}, \delta)$$

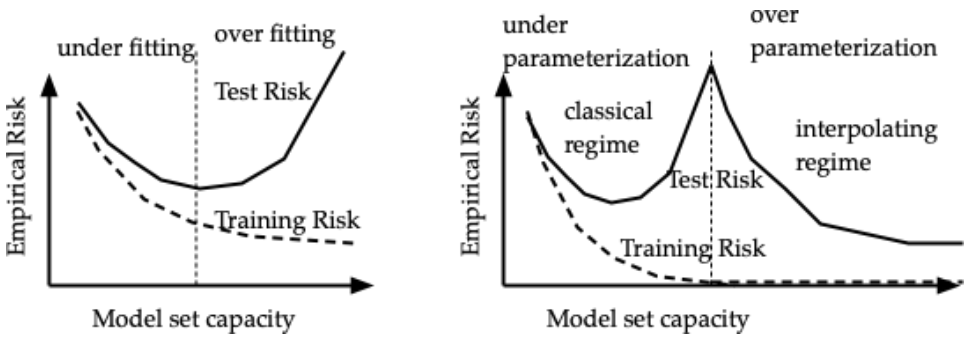


FIGURE 5. Curves for training risk (dashed line) and test risk (solid line). (left) The classical U-shaped risk curve arising from the bias-variance tradeoff. (right) The double descent risk curve, which incorporqtes the U-shaped risk curve (classical regime) with the observed behavior from using high capacity model classes. The models to the right of the interpolation threshold have zero training risk.

where the generalization error is

$$\Omega(N, d_{\text{vc}}, \delta) = \sqrt{\frac{8}{N} \log\left(\frac{4((2N)^{d_{\text{vc}}} + 1)}{\delta}\right)}$$

In general, we expect $\widehat{R}_{\mathcal{D}}[h^*]$ to decrease in a monotone manner with VC dimension since the model is getting large enough to actually memorize the dataset. On the other hand, the error term Ω will increase monotonically with d_{vc}. This means that if we plot both the empirical risk $\widehat{R}_{\mathcal{D}}[h]$ and the error term Ω as a function of d_{vc} we get the curves shown in Fig. **??**. The dashed line represents an upper bound on the true risk $R[h]$ and it has the characteristic U shape that was mentioned in the preceding section.

The U-shaped phenomenon seen in Fig. 6 is called the *bias-variance* tradeoff. The U-shape of the curve means that there is an optimal VC dimension, d^*_{vc}, for the model set that will have the lowest actual risk. This optimal VC dimension divides the true risk plot into two sections; one where $d_{\text{vc}} < d^*_{\text{vc}}$ and the other where $d_{\text{vc}} > d^*_{\text{vc}}$. The true risk for $d_{\text{vc}} < d^*_{\text{vc}}$ will be larger because the model set is not *rich* enough to capture the statistical

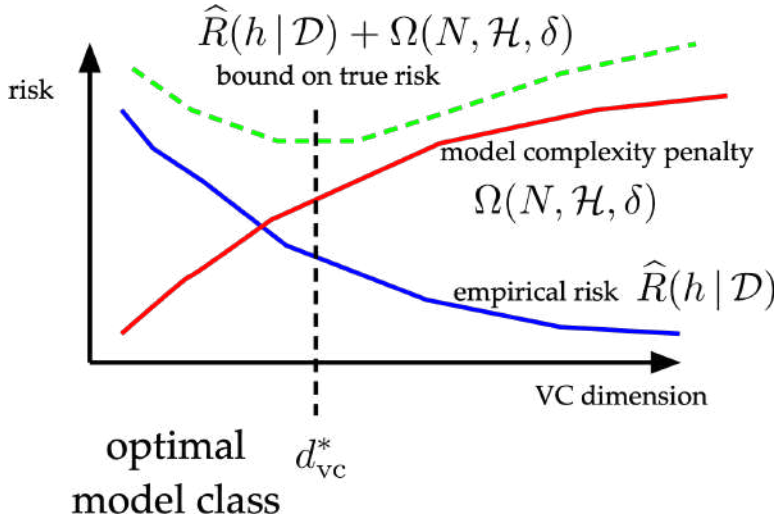


FIGURE 6. The upper bound on the optimal model's true risk as a function of the model set's VC dimension.

variations in the dataset. So this larger error is generated by a "bias" inherent in the model set. In this case, we say the model *underfits* the data. The true risk for the other region where $d_{\text{vc}} > d^*_{\text{vc}}$ has a larger error because the model set has too many degrees of freedom. In particular it means the model is so complex that it can actually "memorize" the training data and thereby capture meaningless statistical fluctuations. So this case is matching the "variance" of the data arising from noise and we say that the model *overfits* the data. One of the key challenges in the practical training of neural network models lies in formulating a training pipleline that controls model overfitting and underfitting in a manner that allows us to identify the "best" model.

In general, however, we do not directly control a model set's VC dimension. The VC theory provides a convenient way of explaining why managing the complexity of the model set leads to the bias-variance tradeoff. But the VC dimension is only one way of characterizing model complexity. The U-shape change in $R[h]$ as a function of v_{dc} seen in Fig. 6 can be seen as exploring the model space along the direction of v_{dc}. But there may be

other ways of exploring the model space that are easier to work with. In the following discussion we introduce one such method that is known as *early stopping* [**YRC07**].

Early stopping is based on the idea that stochastic gradient descent (SGD) learning also represents a way of exploring the model space. Stochastic gradient descent is an algorithm that updates the weights, w, of a neural network model using the equation

$$w_{k+1} \leftarrow w_k - \eta \frac{\partial \widehat{R}_{\mathcal{D}}(w)}{\partial w}$$

where $\eta > 0$ is called the learning rate and $\widehat{R}_{\mathcal{D}}(w)$ is a noisy estimate of the true risk's gradient with respect to weights w. That "noise" originates in the fact that we are using the empirical risk rather than true risk to compute the gradient. What this means is that if we start from w_k and then do a series of SGD updates using a batch of the samples in the dataset, we are doing a random walk through the model space. So after going through a complete epoch of samples, we have actually explored a region of the model space about the original w_k. As we continue performing additional training epochs, we explore larger and larger subsets of the model space. In other words, our search using this noisy gradient descent scheme is generating a sequence of model sets that are increasing in complexity because they are getting larger and larger. In accordance with the VC theory, this means that we should also see the same sort of U-shaped behavior in the empirical risks computed on statistically independent training and testing sets. This is, of course, exactly what we saw in the previous chapter's training curves.

This view of training as a search through successively larger model sets means that we can expect the empirical risk and true risk of the kth model to have the characteristic U-shape we saw in Fig. 6. The only difference being that now the x-axis of our plot as shown in Fig. 7 is the number of training epochs, rather than the model set's VC dimension. Unlike the VC analysis, we do not know the bound on the true risk. But we can measure the empirical risk during training and then evaluate each model we obtain

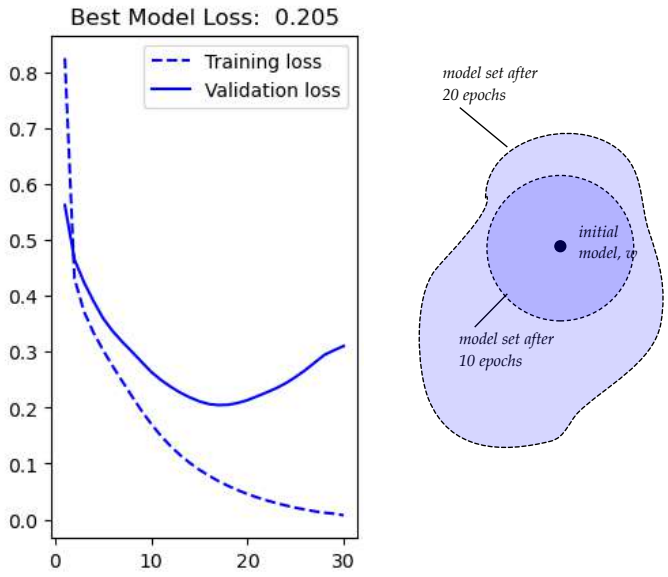


FIGURE 7. Early Stopping

at the end of an epoch on a small data set that we have held out from the training data. We can then take this estimate of the risk on the test data as a measure of the true risk. This approach to controlling model complexity is called *early stopping* and it is the main way that ML engineers use to find the "optimal" model that generalizes well.

The use of early stopping to control model complexity requires that we have a way to estimate the model's true risk, $R[h^*]$. Above, we said we can do this by holding out part of the data from training and use the resulting *test data set* to evaluate the model's loss. So, our training protocol first requires us to do a gradient update of the current model's weights using training data, and then we evaluate the loss of that model using both the training data and the testing data. The loss over the training data is the empirical risk. The loss over the testing data is taken as an estimate of the true risk.

Why should the aggregate loss over the testing data be a good estimate of the true risk? Let $\widehat{R}^{(\text{test})}[h^*]$ denote the empirical risk evaluate on the test

data set, $\mathcal{D}^{(\text{test})}$ where h^* minimizes the empirical risk, $\widehat{R}^{(\text{train})}[h^*]$ evaluated on the training data set, $\mathcal{D}^{(\text{train})}$. When we report $\widehat{R}_{\mathcal{D}^{(\text{test})}}[h^*]$ as our estimate for the true risk $R[h^*]$, we are asserting that this reported value has a smaller generalization error than that reported on the training data. The reason why this is true is because the model set over which we are evaluating the generalization error is now very small. It consists of only one model, namely h^*. This means we can use our earlier generalization error result for finite model sets and assert that

$$R[h^*] \leq \widehat{R}^{(\text{test})}[h^*] + \sqrt{\frac{1}{2N}\log\frac{2}{\delta}}$$

with a confidence $1-\delta$. The bound will be much tighter than what we obtained using the VC dimension.

As an example, consider a dataset with 600 examples. Let us set aside $N^{(\text{test})} = 200$ examples for the test data set. We use a learning model set, $\mathcal{H}$ with $M = 10000$ models and select a model h^* based on the $N^{(\text{train})} = 400$ samples not in the test set. Using our earlier Hoeffding bounds we can estimate the generalization error on $\widehat{R}^{(\text{train})}[h^*]$ and $\widehat{R}^{(\text{test})}[h^*]$. In particular, let us assume we want a 95% confidence level (i.e. $\delta = 0.05$). The training bound would be

$$|\widehat{R}^{(train)}[h^*] - R[h^*]| \leq \frac{1}{2N^{(train)}}\log\left(\frac{2M}{\delta}\right) = 0.0322$$

and the testing bound would be

$$|\widehat{R}^{(test)}[h^*] - R[h^*]| \leq \frac{1}{2N^{(test)}}\log\left(\frac{2}{\delta}\right) = 0.0092.$$

So the generalization error on the testing data is an order of magnitude smaller than that on the training data. This was computed for a finite model set. But if we consider an infinite model set using deep neural network that have an extremely large VC dimension, then we could readily conclude that the testing bound would be orders of magnitude less than the training bound.

CHAPTER 3

Neural Network Model Sets

Neural network model sets are the basic type of model used in deep learning. As discussed in chapter 1, there were three waves of neural network research that marked the emergence of three different types of neural network models; the perceptron, the artificial neural network or multi-layer perceptron, and deep neural networks. This chapter reviews all of these models and discusses methods that are used to train a model to minimize the empirical risk on a given dataset.

1. Perceptron

The perceptron is a linear machine of the form

$$h_{w,b}(x) = \sigma(w^T x + b)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is a monotone increasing *activation function*, $w \in \mathbb{R}^n$ is a vector of real valued weights, $b \in \mathbb{R}$ is a real valued bias, and $x \in \mathbb{R}^n$ is the input sample from the dataset. We refer to such models as *linear* because the arugment to the activation function is a linear function of the input x. Note that if we let $\overline{w} = \begin{bmatrix} b \\ w \end{bmatrix}$ and $\overline{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$, then we can rewrite the model as $h_{\overline{w}}(x) = \sigma(\overline{w}^T \overline{x})$. Sometimes this "unbiased" form is more convenient to use in describing neural network models and their training algorithms. So we may switch back and forth between the two represents of the perceptron. The following subsections discuss how perceptrons are trained to solve binary classification, linear regression, and logistic regression problems.

Binary Classification with Perceptron: Let us see how a perceptron that uses a "sgn" activation function works on the binary classification problem. In this case the inputs are vectors $x \in \mathbb{R}^n$ and the targets are -1 or $+1$. We assume a model set whose models are

$$h_w(x) = \text{sgn}(w^T x)$$

and we take the loss function to be

$$L(y, h_w(x)) = \mathbb{1}\left(y \neq h_w(x)\right)$$

so the loss is 1 if the predicted class $h_w(x)$ disagrees with the actual target, y, for that input and is zero otherwise.

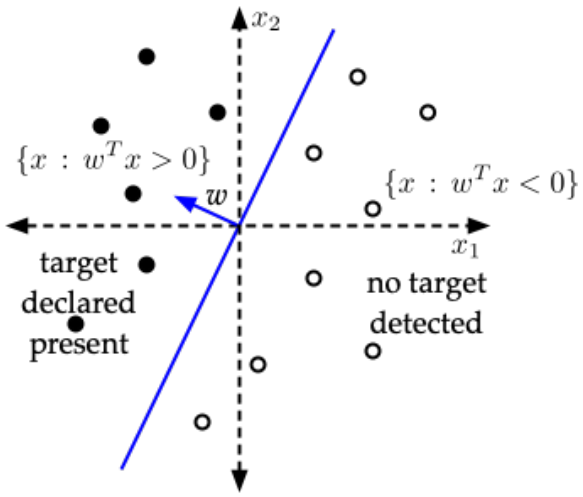


FIGURE 1. Linearly Separable Data Set

As we discussed in chapter 1, we can visualize the dataset and model as shown in Fig. 1. Each dataset input sample is shown as a point in an $\mathbf{R}^2$ input space. The model, $h_w(x)$, is associated with a hyperplane $\{x \in \mathbb{R}^2 : w^T x = 0\}$ that is characterized by the weigth vector $w \in \mathbb{R}^2$. For this weight we can draw a hyperplane that is also called a discriminant surface so that input samples lying on the side of that hyperplane pointed to by w are classified $+1$ and those on the other side of the hyperplane are classified as -1. For the dataset shown in Fig. 1 we note that all of the $+1$ samples are classified correctly by the given model. Any dataset for which there exists

a weight vector w for which the perceptron, $h_w(\cdot)$, correctly classifies every dataset sample is said to be *linearly separable*.

Training of this perceptron model for the binary classification problem can be accomplished using the *perceptron learning algorithm* (PLA). This algorithm is initialized with a randomly selected weight vector, w, and two data matrices $\mathbf{X} \in \mathbb{R}^{n \times N}$ and $\mathbb{Y} \in \mathbb{R}^N$ formed from the inputs and targets of the training dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$. The data matrices are

$$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & \cdots & x_N \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

The algorithm recursively computes an update to the weight vector. If we let w denote the weight vector in the kth recursion of the algorithm, then the algorithm selects a sample (x_ℓ, y_ℓ) from the dataset and if input x_ℓ's target, y_ℓ, is correctly predicted by $h_w(x_\ell)$, then the weight w is left unchanged. If, on the other hand, the input x_ℓ is misclassified by the perceptron (i.e. $h_w(x_k) \neq y_\ell$), then the algorithm updates the weight. We can therefore write the update as

$$w \leftarrow \begin{cases} w + \eta y_\ell x_\ell & \text{if } y_\ell w^T x_\ell < 0 \\ w & \text{if } y_\ell w^T x_\ell \geq 0 \end{cases}$$

The hyperparameter η is called the algorithm's *learning rate*. The PLA algorithm converges in a finite number of steps if the training dataset is linearly separable.

Perceptron Training for Linear Regression: Let us examine how the perceptron model might be used to solve a linear regression problem. In this case the input samples, $x_k \in \mathbb{R}^n$ and the targets are real numbers

$$y_k = g(x_k) + n_k$$

where $g : \mathbb{R}^n \to \mathbb{R}$ is an *unknown* function and $\{n_k\}_{k=0}^\infty$ is a sequence of i.i.d random variables with zero mean and finite variance. The models use

a linear activation function ($\sigma(y) = y$) so that the model's prediction is

$$h_w(x) = w^T x$$

where $w \in \mathbb{R}^n$ is the weight vector. Our problem is to find an optimal set of weights that minimize the empirical risk on the dataset $\{(x_k, y_k)\}_{k=1}^N$ defined with respect to the squared error loss function

$$L(y, h_w(x)) = (y - h_w(x))^2$$

The optimal weights can be computed in closed form as follows. Note that in terms of the data matrices, $\mathbf{X}$ and $\mathbf{Y}$, we introduced before that the empirical risk of model h_w on $\mathcal{D}$ will be

$$\widehat{R}_{\mathcal{D}}[h_w] = \frac{1}{N} |\mathbf{X}w - \mathbf{Y}|^2$$

This is a quadratic function so the optimal weight occurs when the gradient of $\widehat{R}_{\mathcal{D}}[h_w]$ with respect to w is zero. In other words, w^*, satisfies

$$0 = \nabla_w \widehat{R}_{\mathcal{D}}[h_w] = \frac{2}{N} \left(\mathbf{X}^T\mathbf{X} - \mathbf{X}^T\mathbf{Y}\right)$$

Assuming $\mathbf{X}^T\mathbf{X}$ is invertible then the optimal weight is

$$w^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

Note that we can also solve for w^* in a recursive manner using an algorithm known as the recursive least squared (RLS) algorithm. This algorithm is numerically more stable than the closed form formula since it does not have to invert the potentially large matrix $\mathbf{X}^T\mathbf{X}$. In particular, we note that

$$\mathbf{X}^T\mathbf{X} = \sum_{k=1}^{N} x_k x_k^T$$

So if we consider the sequence of matrices formed from the recursion

$$\mathbf{A}_k = \mathbf{A}_{k-1} + x_k x_k^T$$

then one can show using the Sherman-Morrison formula[1] that the sequence of the matrix inverses of $\mathbf{A}$ satisfy the recursion

$$\mathbf{A}_{k+1}^{-1} = \mathbf{A}_k^{-1} - \frac{\mathbf{A}_k^{-1} x_{k+1} x_{k+1}^T \mathbf{A}_k^{-1}}{1 + x_{k+1}^T \mathbf{A}_k^{-1} x_{k+1}}$$

We can then use this recursion to generate a sequence of weights w_k that converge to the optimal w^*. That recursion is

$$\begin{aligned} w_k &= w_{k-1} + L_k(y_k - w_{k-1}^T x_k) \\ L_k &= \frac{\mathbf{P}_k x_k}{1 + x_k^T \mathbf{P}_{k-1} x_k} \\ \mathbf{P}_k &= \mathbf{P}_{k-1} - \frac{\mathbf{P}_{k-1} x_k x_k^T \mathbf{P}_{k-1}}{1 + x_k^T \mathbf{P}_{k-1} x_k} \end{aligned}$$

with initial condition $\mathbf{P}_0 = \text{diag}(p_0)$ with $p_0 \gg 0$ and w_0 is randomly selected. The algorithm is usually referred to as the recursive least squares or RLS algorithm [**SL87**].

Perceptron Training for Logistic Regression: Logistic regression is a relaxed form of binary classification where the model estimates the probability that the input sample is "in-class". In this case the optimal model is chosen to minimize an empirical risk based on the binary cross-entropy function

$$L(y, h_w(x)) = -y \log(h_w(x)) - (1 - y) \log(1 - h(x))$$

and the model set is a perceptron $h_w(x) = \sigma(w^T x)$ whose activation function is the softmax function

$$\sigma(s) = \frac{e^s}{1 + e^s}$$

Finding the optimal model, h^* that minimizes the empirical risk is usually done through stochastic gradient descent. Let $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ be

[1] **Sherman-Morrison Formula:** $(\mathbf{A} + uv^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1} u v^T \mathbf{A}^{-1}}{1 + v^T \mathbf{A}^{-1} u}$.

the training dataset. The empirical risk we wish to minimize is

$$\widehat{R}_{\mathcal{D}}[h_w] = -\frac{1}{N}\sum_{k=1}^{N} ML(y_k, h_w(x_k))$$

where $L(\cdot,\cdot)$ is the binary cross-entropy function. The gradient of the empirical risk is

$$\nabla_w \widehat{R}_{\mathcal{D}}[h_w] = -\frac{1}{N}\sum_{k=1}^{N} \nabla_w L(y_k, h_w(x_k))$$

Because we are using a softmax activation function, one can show that $1 - \sigma(s) = \sigma(-s)$ and $\frac{d\sigma(s)}{ds} = (1-\sigma(s))\sigma(s)$. This allows us to write the gradient of the loss function as

$$\nabla_w L(y_k, h_w(x_k)) = x_k(y_k - \sigma(w^T x_k))$$

So when doing a gradient update of the weight vector, w, we can proceed by taking the gradient of $L(y_k, h_w(x_k)$ for each sample (x_k, y_k) in the data set and then update the weight for $k = 1, \ldots, N$ as

$$w \leftarrow w - \frac{\eta}{N}\sum_{k=1}^{N} x_k(y_k - \sigma(w^T x_k))$$

A single pass of this algorithm through all of the data samples is called a *training epoch*.

2. Multi-layer Perceptrons and Deep Neural Networks

The perceptron appeared in the 1960's. The 1980's saw the development of the multi-layer perceptron, which was simply a perceptron with an additional hidden layer between the perceptron's input and output layer. A *deep neural network* is simply a multi-layer perceptron with more than one hidden layer. This section establishes some of the conventions used in describing deep neural networks formed by adding multiple hidden layers between the inputs and outputs. We will use directed graphs and layered abstractions to describe these models.

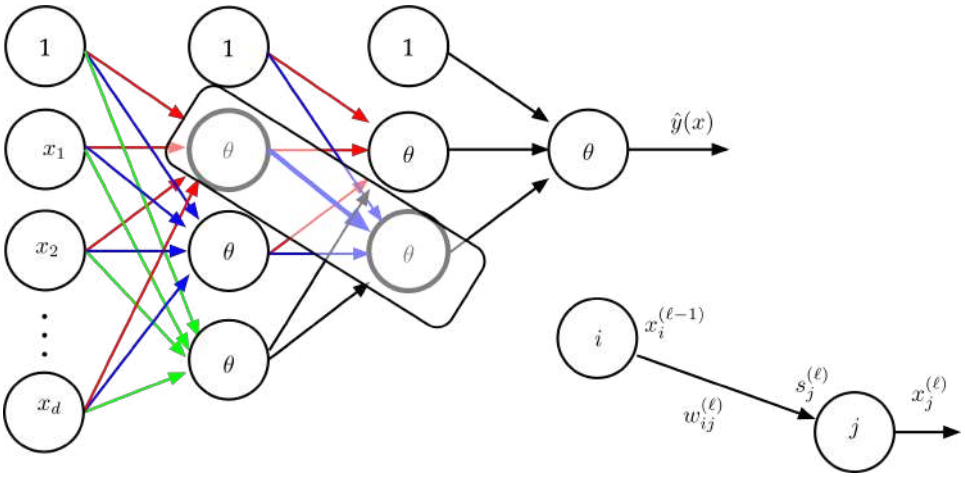


FIGURE 2. Directed Graph Representation of Sequential Neural Network

A neural network's graph abstraction is shown in Fig. 2. This network has two hidden layers (though it could have more) and uses the activation function $\sigma : \mathbb{R} \to \mathbb{R}$. The layers are labeled $\ell = 0, 1, 2, \ldots, L$ with the 0th layer being the input layer and layer L being the output layer. The remaining layers, $(0 < \ell < L)$ are called *hidden layers*. The *dimension* of layer ℓ is denoted as $d^{(\ell)}$ where that layer has $d^{(\ell)} + 1$ nodes labeled $0, 1, \ldots, d^{(\ell)}$. The zeroth node in the layer is called a *bias node* with no input and an output of 1. Every edge is directed from a node in, say, layer ℓ to a node in the next layer $\ell + 1$.

Each edge is labeled with a *weight*, Let us zoom in on a single edge of the graph in Fig. 2. This edge is shown on the bottom right hand corner of the figure and depicts an edge that originates in layer $\ell - 1$ and terminates at the jth node in layer ℓ. The output of node i in layer $\ell - 1$ is denoted as $x_i^{(\ell-1)}$. The output is multiplied by the weight $w_{ij}^{(\ell)}$ to create the input $s_j^{(\ell)}$ into the jth node of layer ℓ. That input is then passed through the activation function, σ, to create the jth node's output $x_j^{(\ell)}$.

While sequential neural networks are often depicted using directed graphs, this depiction becomes cumbersome when we consider much larger networks. Moreover for deep neural network such visual depictions of directed graph are inconvenient for representing special layers such as convolutional layers. For large deep neural networks it is more convenient to use a *layered abstraction*. The layered abstraction collects all input signals into the nodels $1, \ldots, d^{(\ell)}$ of layer ℓ into a vector $\mathbf{s}^{(\ell)} = \left[s_1^{(\ell)}, \ldots, s_{d^{(\ell)}}^{(\ell)}\right]^T$. It collects all output signals from nodes, $0, 1, \ldots, d^{(\ell)}$ of layer ℓ into the vector $\mathbf{x}^{(\ell)} = \left[x_0^{(\ell)}, x_1^{(\ell)}, \ldots, x_{d^{(\ell)}}^{(\ell)}\right]^T$. The weights of the ℓth layer are collected into a $(d^{(\ell-1)} + 1) \times d^{(\ell)}$ matrix

$$\mathbf{W}^{(\ell)} = \begin{bmatrix} w_{01}^{(\ell)} & w_{02}^{(\ell)} & \cdots & w_{0d^{(\ell)}}^{(\ell)} \\ w_{11}^{(\ell)} & w_{12}^{(\ell)} & \cdots & w_{1d^{(\ell)}}^{(\ell)} \\ \vdots & \vdots & & \vdots \\ w_{d^{(\ell-1)}1}^{(\ell)} & w_{d^{(\ell-1)}2}^{(\ell)} & \cdots & w_{d^{(\ell-1)}d^{(\ell)}}^{(\ell)} \end{bmatrix}$$

With these matrix-vector objects, the relationships between inputs to layer ℓ can be related to the outputs of layer $\ell - 1$ feeding into the activation function as

$$\mathbf{s}^{(\ell)} = \left[\mathbf{W}^{(\ell)}\right]^T \mathbf{x}^{(\ell-1)}$$

and the ℓth layer's output are then

$$\mathbf{x}^{(\ell)} = \begin{bmatrix} 1 \\ \sigma(\mathbf{s}^{(\ell)}) \end{bmatrix}$$

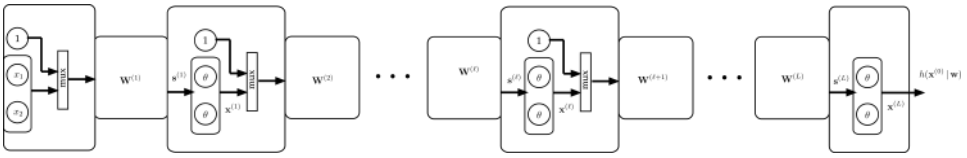


FIGURE 3. Layered Abstraction for Sequential Neural Networks

These matrix-vector relationships are more conveniently depicted in the layered abstraction shown in Fig. 3. All of the hidden layers (i.e. $0 < \ell <$

L) have the same structure. The 0th nodal layer takes inputs x_k from the dataset and the Lth layer has no bias node and outputs the nodal layer vector $\mathbf{x}^{(L)}$ as the neural network's output $h_{\mathbf{w}}[x^{(0)}]$ in response to input $\mathbf{x}^{(0)}$ with networks weights $\mathbf{w}$ being formed from all layered weight matrices $\mathbf{W}^{(\ell)}$.

The layered abstraction in Fig. 3 is the abstraction used in most deep learning libraries such as TensorFlow and PyTorch. TensorFlow and PyTorch are two of the most popular libraries used at the time this chapter was written. This book restricts its attention to TensorFlow. The following script shows how TensorFlow's Keras API would be used instantiate a neural network `Model` object for a multi-class classification problem. The input is a vector representing an image and the output is a vector whose components are the probabilities of each class.

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(28*28))
x = layers.Dense(512,activation="relu")(inputs)
outputs = layers.Dense(10,activation="sigmoid")(x)
model = keras.Model(inputs=inputs,outputs=outputs)
model.summary()
```

The output generated by this Python script is shown. below

```
Metal device set to: Apple M1

systemMemory: 16.00 GB
maxCacheSize: 5.33 GB

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 1, 784)]          0

 dense (Dense)               (None, 1, 512)            401920

 dense_1 (Dense)             (None, 1, 10)             5130
```

```
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

The preceding script first imported the Keras library and the associated sublibrary of `Layers` built into that library. In this example we only use the `Dense` layer abstraction which connects all inputs through weights to all outputs. The input layer has $28 \times 28 = 784$ nodes and it outputs a rank-2 tensor of shape $(1, 784)$. The hidden layer in this model has 512 outputs. It is a dense layer that maps each input plus the bias to an output. So the total number of trainable weights in the first layer will be $(784 + 1) \times 512 = 401,920$. The last layer is a dense layer taking 512 inputs onto 10 outputs. Again the bias term is active in this layer so the total number of trainable weights is $(512 + 1) \times 10 = 5130$. The summary() method displays these layers and their shapes. Fig. 4 graphically illustrates this sequential neural network model in terms of the three layers used to define it (Input, Dense, Dense).

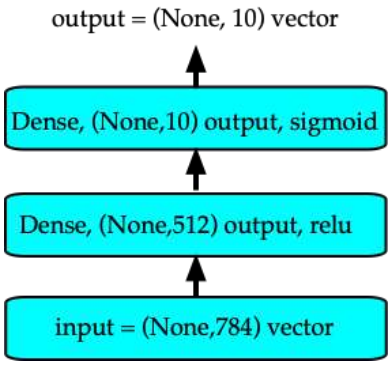


FIGURE 4. Layered Abstraction for Neural Networks

The preceding script simply scopes out the model object. To fully instantiate the model, we need to configure how it is trained. This configuration involves specifying the type of SGD optimizer to be used, the loss function, and the data to be used for training and testing. This is done through the `model` object's `compile` method. The following script configures the model to be trained using the RMSprop optimizer and the loss function is sparse categorical crossentropy. This particular loss function is used for multi-class classification problems where the system target is specified as an integer rather than a hot-encoded vector. SGD training seeks to minimize

the empirical loss function. But since this model was created for a multi-class classification problem, we are not really interested in characterizing performance in terms of ”crossentropy”. We are instead interested in the classification error, so the last argument of our compile method specifies classification *accuracy* (percentage of correct classifications) as the metric that we want to keep track of during training.

```
model.compile(optimizer = "rmsprop",
       loss = "sparse_categorical_crossentropy",
       metrics = ["accuracy"])
```

The last step in training is to use the `model` object's `fit` method. Training is done using a batched SGD algorithm. Recall that this means that the training data set is partitioned into batches of a given size. The following script trains the model for 5 epochs on the specified training data assuming a batch size of 128 samples.

```
model.fit(train_x, train_y, epochs = 5, batch_size = 128)
```

After the model has been trained, we need to estimate the model's actual risk. As mentioned before this is done by computing the empirical risk over a test set that is independent of the training set. We use the `model` object's `evaluate` method to do this

```
test_loss, test_acc = model.evaluate(test_x, test_y,verbose=0)
train_loss, train_acc = model.evaluate(train_x,train_y, verbose=0)
print(f"Test Accuracy: {test_acc:.3f}, Training Accuracy: {train_acc:.3f}")
print(f"Test Loss: {test_loss:.3f}, Training Loss {train_loss:.3f}")
```

In my case we obtained a test accuracy of 98.3%. As expected the training accuracy is very high, 99.8%. This is greater than the testing accuracy. The testing loss was 0.070 and the training loss was 0.006. Clearly while we train to minimize loss, these small loss values say little about the actual classification accuracy, this is why our training process also keeps track of the model's training and testing accuracy after each training epoch.

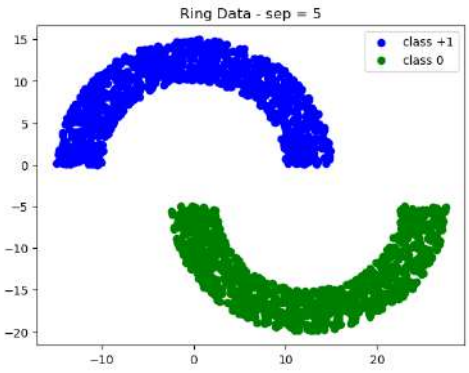


FIGURE 5. Ring Data for HW 1

The preceding discussion demonstrated the use of TensorFlow/Keras deep learning libraries to instantiate, train, and evaluate a deep neural network model. These are Python libraries that makes use of class objects to instantiate models, optimizers, loss functions, and datasets. What I'd like to do in the rest of this section is provide a deeper look at how some of the class objects are built. This will be of interest to us because at some point one often needs to customize an existing library class object for their particular application. The following discussion walks through a simple class objects for a sequential model and its dense layers. I'll be solving a logistic regression problem using the Ring Data in HW1. This data file creates a numpy array with shape $(3, 2000)$. It is a matrix whose kth column is the vector $\begin{bmatrix} x_k \\ y_k \end{bmatrix}$ with $x_k \in \mathbb{R}^2$ being the kth input and y_k being a binary class target that takes values of 0 or 1. Fig. 5 illustrates the ring dataset, by scatter plotting the input vectors, x_k, and coloring each input as blue for class 1 and green for class 0.

Python is an objected oriented programming (OOP) language. Keras uses the Python class objects to *instantiate* objects used in building a neural network model. There are, in particular, two basic classes we need to develop; the `layers` class and the `model` class. The `Model` class objects are essentially lists of `Layer` class objects. The class uses an `__init__` method to instantiate a class instance. The class uses the `__call__` method to make that class instance callable. The following Python script is for a simple `SequentialModel` class

```
class SequentialModel:
     def __init__(self, layers):
          self.layers = layers
```

```
    def __call__(self, inputs):
         x = inputs
         for layer in self.layers
              x = layer(x)
         return x

    @property
    def weights(self):
         weights = []
         for layer in self.layers:
              weights += layer.weights
         return weights
```

As can be readily seen from this script, the class constructor simply creates a "list" that holds pointers to the `Layer objects`. The `__call__` then allows that model to be executed by simply taking the output generated by one layer in the list to the next layer in the list. The class object also uses a *property decorator* to add a function that we can use to fetch the model's current set of weights. This decorator is used during training of the model.

The other major class of interest to us will be the `DenseLayer` class. This class instantiates a single layer in a network whose inputs are connected to the outputs in a dense manner. In particular, the layer's outputs is the tensor

$$\sigma(\mathbf{W}x + b)$$

where $\mathbf{W}$ is a matrix of trainable weights, b is a vector of trainable biases, x is the input vector (tensor), σ is the activation function. The following script is for the `DenseLayer` class object

```
class DenseLayer:
    def __init__(self, input_size, output_size, activation):
          self.activation = activation
          w_shape = (input_size, output_size)
```

```
            w_initial  = tf.random.uniform(w_shape, minval=0,maxval=.1)
            self.W = tf.Variable(b_initial)

      def __call_(self, inputs):
            return self.activation(tf.matmul(inputs, self.W)+self.b)

      @property
       def weights(self):
            return [ self.W, self.b]
```

Our class constructor simply initializes the persistent objects needed to compute this output and randomly initializes the weights. A close look at the `__call__` methods shows that it computes the output $\sigma(\mathbf{W}x + b)$. The class also includes a property decorator that is used by our training method to fetch and update the classes weights.

A model is instantiated by first instantiating the `DenseLayer` objects comprising the layers of our model. We then list the pointers to these `DenseLayer` objects and use that to instantiate a `SequentialModel` object

```
n_hidden = 16
relu_act = tf.nn.relu
smax_act = tf.nn.softmax
layer1 = DenseLayer(2, n_hidden, relu_act)
layer2 = DenseLayer(n_hidden, 2, smax_act)
model = SequentialModel([layer1, layer2])
```

This script instantiates a neural network that takes a rank-2 tensor input and maps it to a hidden layer of 16 nodes. The second dense layer then maps the outputs of the hidden layer to two nodes. We use a softmax activation on the last layer because we are solving a logistic regression problem in this case. In particular, we are using the ring data from the first HW assignment as the inputs.

One of the first thing we would like to do with a model is to `evaluate` its accuracy on the dataset $\mathcal{D} = (\mathbf{X}, \mathbf{Y})$. In this case $\mathbf{X}$ is a rank-2 tensor whose rows are the components of the input sample vectors in the ring dataset. The rank-1 tensor $\mathbf{Y}$ are the class targets that in this dataset take values of 0 or 1. The following script evaluates the accuracy of the model before training

```
def evaluate(model, X,Y):
      Yhat = model(X).numpy()
      preds = np.zeros(len(Y))
      indx1 = np.where(Yhat[:,0]< 0.5)
      preds[indx1] = 1
      matches = (Y==preds)
      accuracy = matches.mean()
      return accuracy

accuracy = evaluate(model,X,Y)
```

In my original version of this script the untrained model had an accuracy between 15%. So we want to train it and then see if the accuracy improves.

Model training is done through the `fit` function. This functions implements the backpropagation algorithm described above. In practice, we use a computational approach called *automatic differentiation* (to be described below) to evaluate the model's empirical risk gradient on a given dataset. Automatic differentiation constructs a *computation gtraph* that records the order of numerical operations performed when computing the model's output. Automatic differentiation then takes the computation graph and uses it to compute the gradient of the empirical risk for a specific sample from the dataset.

TensorFlow provides the `GradientTape` object to perform automatic differentiation. The following code illustrates how the `GradientTape` object is instantiated and then used in the `fit` function.

```
def loss_function(Y, Yhat):
     scce = tf.keras.losses.sparse_categorical_crossentropy
     per_sample_losses = scce(Y,Yhat)
     loss = tf.reduce_mean(per_sample_losses)
def fit(model, X,Y,n_epochs, lr):
     Yhat = model(X)
     history = [loss_function(Y, Yhat)]
     for epoch in range(n_epochs):
          with tf.GradientTape() as tape:
               Yhat = model(X)
               loss = loss_function(Y,Yhat)
          gradients = tape.gradient(loss, model.weights)
          for g,w in zip(gradients, model.weights):
               w.assign_sub(g * lr)
          history = np.vstack((history,loss))
          if epoch%100 == 0:
               print(f"epoch {epoch}: loss = {loss: .2f}")

history = fit(model, X_train, Y_train, X_test,Y_test, 1000, .1)
```

The `fit` function goes through `n_epochs` training epochs. Within that training `for` loop we see the `GradientTape` object instantiated as `tape`. This tape is the computation and it is generated by the commands within the indented section of code after `tape` was instantiated. Essentially, what the object does is record the mathematical operations performed in executing those functions. Once the `GradientTape` object has finished recording these operations, we call the object's `gradient` method to automatically compute the gradient of the `loss` function with respect to the `model.weights`. These gradients are then used to update the weights in the model using a standard gradient descent update with learning rate `lr`. The `fit` function is then called, trains the model for 1000 epochs and then plots the training loss as a function of training epoch. The figure 6 also shows the model predictions and attains a final accuracy of 90%.

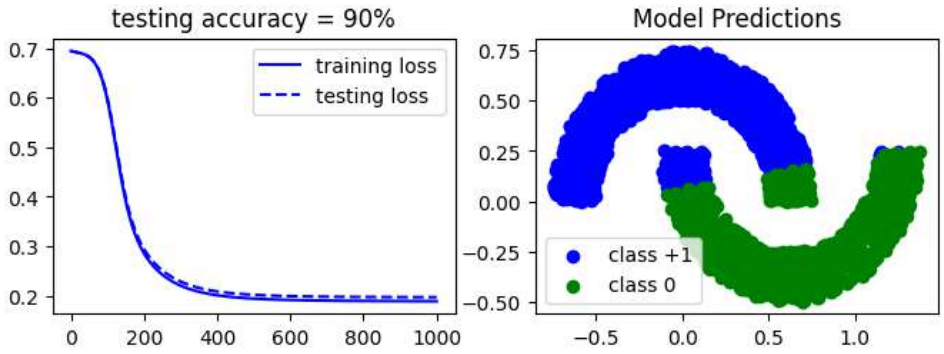


FIGURE 6. Training Results on Ring Data

3. Universal Approximation Ability

The universal approximation theorem [Cyb89] asserts that any continuous function f on a compact set can be uniformly approximated to an accuracy ϵ by a sequential neural network. A formal statement of this theorem is given below and an outline of the proof provided below.

THEOREM 2. *If the activation function in a neural network,*

$$h_w(x) = \sum_{k=1}^{N} \alpha_k \sigma(w_k^T x + b_k),$$

is sigmoidal then the model set of all such neural networks is dense in $C(I_n)$ *(the space of continuous functions defined on* $I_n = [0,1]^n$*).*

The proof of this theorem rests on concepts from Functional Analysis, where the model set, $\mathcal{H}$ is viewed as a linear subspace of a normed linear space of functions. A *linear space* abstracts the fundamental properties of a *vector space* so we can apply those properties to other types of mathematical objects. A *normed* linear spaces associates a measure of "length" to its elements. In this theorem, we consider the space of continuous functions whose inputs take values on the n-dimensional unit cube. The norm we take

is the supremum-norm that we denote as

$$\|h\| = \sup_{x \in [0,1]^n} |h(x)|$$

where $|h(x)|$ is the absolute value of the model's, h, output.

The theorem says that the model set $\mathcal{H}$ formed from sigmoidal neural networks with a single hidden layer is *dense* in the linear space of continuous functions, $C(I_n)$. For a set, $\mathcal{H}$, to be dense in $C(I_n)$ means that any limit point of an infinite sequence of functions in $\mathcal{H}$ is a function in the $C(I_n)$. In particular, let f be any function in $C(I_n)$, then this means there is a sequence of models $\{h_n\}_{n=1}^{\infty}$ in $\mathcal{H}$ such that for any $\epsilon > 0$ there exists N such for all $n > N$ we have $\|h_n - f\| < \epsilon$. In other words, any continuous function can be approximated arbitrarily closely by a model in $\mathcal{H}$.

The proof of this theorem is in Cybenko's paper [**Cyb89**]. This paper assumes that integration of functions in $C(I_n)$ is done with respect to a regular measure, μ. A *measure* is a set-valued function that maps any subset of I_n onto a positive real number. Intuitively we can view a measure as a probability distribution or an area function. Requiring the measure to be "regular" simply means we avoid pathological measures. We use such measures to generalize our notion of integration. Cybenko's paper proves that the sigmoidal activation function σ is *discriminatory*. This means that if

$$\int_{I_n} \sigma(w^T x + b) d\mu = 0$$

for all $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$, then the measure, μ, itself is identically zero. Once this has been established, then the proof of the main theorem follows as a consequence of the Hahn-Banach and Riesz representation theorem.

In particular, let $\mathcal{H} \subset C(I_n)$ and note that $\mathcal{H}$ is a linear subspace of $C(I_n)$. Now consider the closure, $\overline{\mathcal{H}}$ of model set $\mathcal{H}$. If $\mathcal{H}$ is dense in $C(I_n)$, then $\overline{\mathcal{H}} = C(I_n)$, so we will prove the theorem's assertion by contradiction.

In particular we suppose that $\overline{\mathcal{H}} \neq C(I_n)$ and prove that assertion generates a contradiction.

If our assumption is true, then the Hahn-Banach theorem allows us to assert there exists a bounded linear functional, called L, such that $L(\mathcal{H}) = L(\overline{\mathcal{H}}) = 0$ but that $L \neq 0$. The Riesz representation theorem says we can that write functional L as

$$L[h] = \int_{I_n} h(x) d\mu$$

for some measure μ and any $h \in C(I_n)$. Since by definition any neural network, h, is a member of $\mathcal{H}$ and since L is identically zero on $\mathcal{H}$ we have

$$\int_{I_n} h(x) d\mu = 0$$

But the sigmoidal activation function is discriminatory and so the measure $\mu \equiv 0$. This conclusion, however, contradicts our determination that $L \neq 0$ because

$$\mu = 0 \Rightarrow \int_{I_n} h(x) d\mu = 0, \quad \text{for all } h \in C(I_n)$$

This contraction arose because we assumed $\overline{H} \neq C(I_n)$. So the assumption is false, and $\overline{H} = C(I_n)$ which means $\mathcal{H}$ is dense in $C(I_n)$.

The universal approximation theorem does not tell us how to construct this approximation, it merely says such an approximation exists. The result was important because it clearly demonstrated that the model set of neural networks was more "expressive" than the earlier perceptron model set. A concrete example of this expressiveness is given below for the XOR function.

Consider the target function, $f : \mathbb{R}^2 \to \{-1, 1\}$, shown in Fig. 7. This target function realizes a Boolean XOR function and it cannot be written as a perceptron $\mathrm{sgn}(w^T x)$. This function, however, may be decomposed into two perceptrons realizing the hyperplanes shown in Fig. 7. The output is then obtained by combining the outputs from these perceptrons.

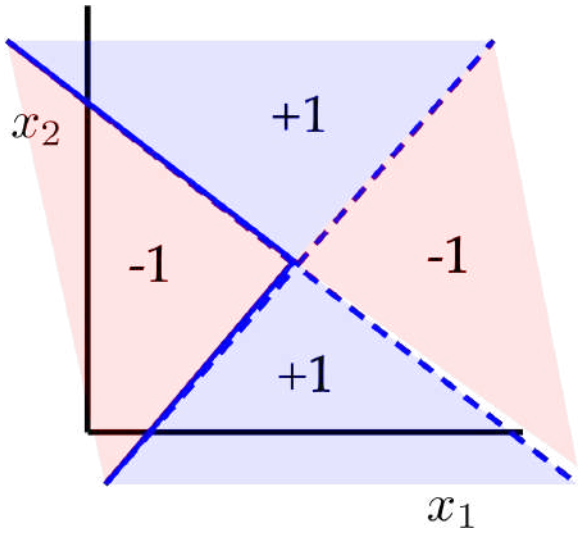


FIGURE 7. Boolean XOR Target Function

In particular, let us characterize these two perceptrons as

$$\widehat{y}_1(x; w_1) = \text{sgn}(w_1^T x), \quad \widehat{y}_2(x; w_2) = \text{sgn}(w_2^T x). \tag{16}$$

The target function's output is $+1$ when only one of the perceptron outputs equals $+1$ and is zero otherwise. This output can, therefore be characterized as

$$f(x) = \widehat{y}_1(x)\,(\neg\widehat{y}_2(x)) + (\neg\widehat{y}_1(x))\,\widehat{y}_2(x) \tag{17}$$

where multiplication is an AND operation, addition is an OR operation, and $\neg$ denotes negation (NOT). The elementary logic operations of AND and OR can be realized using the following perceptron models,

$$\begin{aligned} \text{OR}(\widehat{y}_1, \widehat{y}_2) &= \text{sgn}(\widehat{y}_1 + \widehat{y}_2 + 1.5) \\ \text{AND}(\widehat{y}_1, \widehat{y}_2) &= \text{sgn}(\widehat{y}_1 + \widehat{y}_2 - 1.5) \end{aligned} \tag{18}$$

We will find it convenient to represent the functions in equation (18) as a *directed graph*.

We use the graphs in Fig. 8 to show how one constructs a graph for the MLP realizing the XOR target function. Since addition represents an OR operation, we can view f in equation (17) as the OR of two inputs $\widehat{y}_1(\neg\widehat{y}_2)$

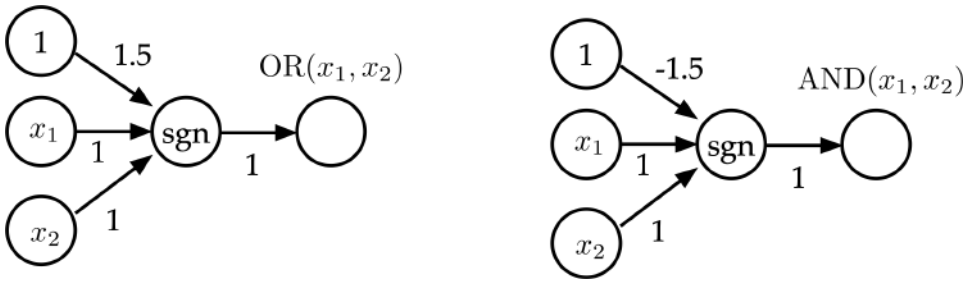


FIGURE 8. LEFT: graph for OR function, RIGHT: graph for AND function

and $(\neg\widehat{y}_1)\widehat{y}_2$. We therefore take the OR perceptron in Fig. 8 as the top layer of our MLP. This graph is shown in Fig. 9(left) as an OR graph whose input nodes are labeled $\widehat{y}_1(\neg\widehat{y}_2)$ and $(\neg\widehat{y}_1)\widehat{y}_2$.

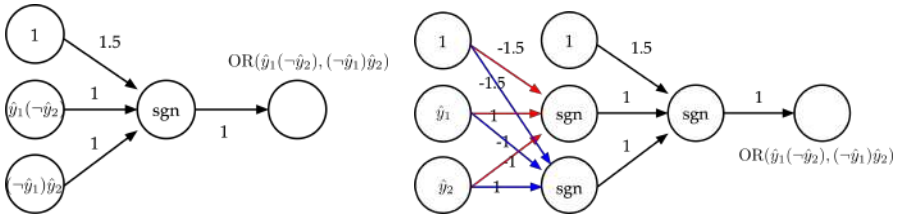


FIGURE 9. LEFT: top OR layer of MLP, RIGHT: AND perceptrons feeding into top layer of MLP

Each input to the OR graph in Fig. 9(left) is obtained by taking AND operations on the two inputs. So we concatenate these two AND graphs with the OR graph as shown in Fig. 9(right). Note that some of the weights have a negative sign to realize the AND of an input with the other input's *negation*. Fig. 9(right) has taken these two AND graphs and collapsed them into one since they have the same input nodes.

We generate the MLP's graph by recognizing that the inputs $\widehat{y}_1$ and $\widehat{y}_2$ are outputs of the two perceptrons in equation (2). These perceptrons may also be represented as graphs with inputs nodes x_1 and x_2. The edges of this graph are weighted by the vectors, w_1 and w_2. We concatenate these two

graphs with the ones in Fig 9(right) and collapse graphs for the two perceptrons into a single graph since they have the same inputs. This yields the graph shown in Fig. 10. Note that this MLP has two hidden layers and so we see that any logic function's MLP can be represented as a graph constructed by composing the elementary graphs for AND and OR functions.

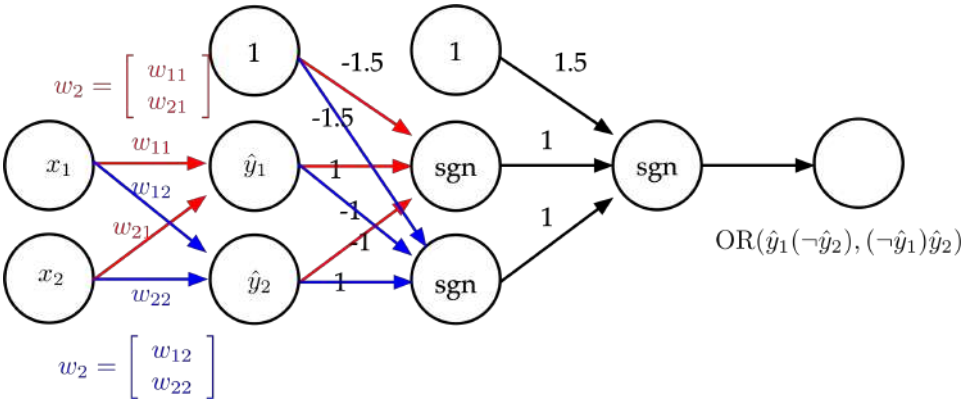


FIGURE 10. Multilayer Perceptron Model for XOR target function

4. BackPropagation

Backpropagation [**RHW86**] may be seen as an extension of the stochastic gradient descent algorithms used for training perceptrons. Backpropagation is also a gradient descent algorithm that uses the chain rule for differentiation to find the gradient of the empirical risk when the model has multiple hidden layers. This section explains the backpropagation algorithm for deep neural networks with L layers using the notational conventions from section 2 of this chapter.

Backpropagation first initializes the weights in each layer, $\mathbf{W}^{(\ell)}$, to random variables and then these weights are updated using a gradient descent update

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \nabla_{\mathbf{W}^{(\ell)}} \widehat{R}_{\mathcal{D}}[h_{\mathcal{W}}]$$

where $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ is the training dataset and $\mathcal{W} = \left\{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L)}\right\}$ is the collection of weights parameterizing the model. We will derive the gradient for the empirical risk

$$\widehat{R}_{\mathcal{D}}[\mathcal{W}] = \frac{1}{N}\sum_{k=1}^{N}(h_{\mathcal{W}}(x_k) - y_k)^2$$

assuming the loss function is the square prediction error, $L(y, x) = (y - h_{\mathcal{W}}(x))^2$.

The gradient of the empirical risk with respect to layer ℓ's weights is

$$\frac{\partial \widehat{R}_{\mathcal{D}}[h_{\mathcal{W}}]}{\partial \mathbf{W}^{(\ell)}} = \frac{1}{N}\sum_{k=1}^{N}\frac{\partial L(y_k, x_k)}{\partial \mathbf{W}^{(\ell)}}.$$

Let us define the *sensitivity vector*, $\delta^{(\ell)}$, for layer ℓ to be the gradient of the loss with respect to input signal $\mathbf{s}^{(\ell)}$.

$$\delta^{(\ell)} \stackrel{\text{def}}{=} \frac{\partial L(y_k, x_k)}{\partial \mathbf{s}^{(\ell)}}. \tag{19}$$

We use the chain rule of differentiation on the jth component of $\delta^{(\ell)}$ to get

$$\delta_j^{(\ell)} = \frac{\partial L(y_k, x_k)}{\partial x_j^{(\ell)}} \cdot \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}} = \sigma'\left(s_j^{(\ell)}\right) \cdot \frac{\partial L(y_k, x_k)}{\partial x_j^{(\ell)}} \tag{20}$$

where $\sigma'(s) = \frac{d\sigma(s)}{ds}$. An expression for the partial derivative of L with respect to $x_j^{(\ell)}$ is again obtained using the chain rule,

$$\frac{\partial L}{\partial x_j^{(\ell)}} = \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial s_k^{(\ell+1)}}{\partial x_j^{(\ell)}} \cdot \frac{\partial L}{\partial s_k^{(\ell+1)}} = \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \delta_k^{(\ell+1)}. \tag{21}$$

Inserting equation (21) into equation (20) gives

$$\delta_j^{(\ell)} = \sigma'(s_j^{(\ell)}) \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \delta_k^{(\ell+1)}. \tag{22}$$

The sum is the matrix-vector product of $\mathbf{W}^{(\ell+1)}$ with the sensitivity vector $\delta^{(\ell+1)}$ with the 0th row being deleted since the gradient with respect to the bias is zero. So we can rewrite equation (22) in matrix-vector form as

$$\delta^{(\ell)} = \sigma'(\mathbf{s}^{(\ell)}) \otimes \left[\mathbf{W}^{(\ell+1)} \delta^{(\ell+1)}\right]_1^{d^{(\ell)}} \tag{23}$$

where $\left[\mathbf{W}^{(\ell+1)}\delta^{(\ell+1)}\right]_1^{d^{(\ell)}}$ contains components $1, \ldots, d^{(\ell)}$ (i.e. excluding the bias term) of the vector $\mathbf{W}^{(\ell+1)}\delta^{(\ell+1)}$ and $\otimes$ denotes component-wise multiplication of similarly shaped tensors.

From the expression for the sensitivity, $\delta_j^{(\ell)}$, we can now examine how the gradient of loss, L, with respect to the weight would be computed. We again use the chain rule to write

$$\frac{\partial L}{\partial w_{ij}^{(\ell)}} = \frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \cdot \frac{\partial L}{\partial s_j^{(\ell)}} = x_i^{(\ell-1)} \cdot \delta_j^{(\ell)}.$$

This is the component-wise equation for the loss' gradient with respect to layer ℓ's weights. When we rewrite this in matrix-vector form we get

$$\frac{\partial L}{\partial \mathbf{W}^{(\ell)}} = \mathbf{x}^{(\ell-1)}(\delta^{(\ell)})^T. \tag{24}$$

So we can see that the sensitivity vector plays a critical role in determine this gradient.

Note that if the activation function is $\sigma(s) = \tanh(s)$, then

$$\sigma'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}.$$

In the following this simplifies the expressions used in computing the gradient. If a different activation function is used, then those simplifications cannot be used.

So the gradient of the empirical risk used to update the ℓth layer's weights will be

$$\begin{aligned} \nabla_{\mathbf{W}^{(\ell)}} \widehat{R}_{\mathcal{D}}[h_{\mathcal{W}}] &= \frac{1}{2N} \sum_{k=1}^{N} \left[\frac{\partial L}{\partial \mathbf{W}^{(\ell)}}\right]_{(x_k, y_k)} \\ &= \frac{1}{2N} \sum_{k=1}^{N} \left[\mathbf{x}^{(\ell-1)}(\delta^{(\ell)})^T\right]_{(x_k, y_k)}. \end{aligned} \tag{25}$$

where the last equation was obtained from equation (24). What this shows is that the gradient for the weights in the ℓ layer are computed from the output, $\mathbf{x}^{(\ell-1)}$, of the $\ell - 1$st layer and the sensitivity, $\delta^{(\ell)}$, of the squared error to changes in the input of the ℓth layer.

These two quantities, $\mathbf{x}^{(\ell-1)}$ and $\delta^{(\ell)}$ in the backpropagation equation (25) are computed recursively in two stages. The first stage is called *forward propagation*. It uses the input to the network, x_k, from the data set, $\mathcal{D}$, to compute the input/output signals $(\mathbf{s}^{(\ell)}, \mathbf{x}^{(\ell)})$ in each layer $\ell = 1, \ldots, L$ using the recursion

$$\begin{aligned} \mathbf{s}^{(\ell)} &= \left[\mathbf{W}^{(\ell)}\right]^T \mathbf{x}^{(\ell-1)} \\ \mathbf{x}^{(\ell)} &= \begin{bmatrix} 1 \\ \sigma(\mathbf{s}^{(\ell)}) \end{bmatrix} \end{aligned} \tag{26}$$

for $\ell = 1, 2, \ldots, L$ with the initial condition being

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ x_k \end{bmatrix}. \tag{27}$$

The second stage computes the loss sensitivities using equation (23)

$$\delta^{(\ell)} = \sigma'(\mathbf{s}^{(\ell)}) \otimes \left[\mathbf{W}^{(\ell+1)}\delta^{(\ell+1)}\right]_1^{d^{(\ell)}} \tag{28}$$

for $\ell = L-1, L-2, \ldots, 1$ where the *terminal condition* is

$$\delta^{(L)} = \frac{\partial L}{\partial \mathbf{s}^{(L)}} = 2(\mathbf{x}^{(L)} - y_k)\sigma'(\mathbf{s}^{(L)}) \tag{29}$$

where y_k is the target for the kth sample in the data set. This is a *backward recursion* that starts at $\ell = L-1$ and uses those results to compute for $\ell = L-2$ and so on until we get back to $\delta^{(1)}$. Because this computation is a backward pass, it is referred to as *backpropagation*. It is from this backward pass that the backpropagation algorithm takes its name.

Once the forward pass equations (26-27) and backward pass equations (28-29) have been completed, then the weights in each layer are updated as

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \frac{\eta}{2N}\mathbf{x}^{(\ell-1)}(\delta^{(\ell)})^T. \tag{30}$$

This forward, backward, and weight update are done for each data sample, (x_k, y_k) in the data set, $\mathcal{D}$.

Let us walk through a simple example to make the preceding description of the backpropagation algorithm more concrete. The initial network with

randomized weights is shown in Fig. 11. There is a scalar input, $x = 2$ and a scalar output $y = 1$ in the dataset. The network has two hidden layers with $d^{(1)} = 2$ and $d^{(2)} = 1$. The initial weight matrices are

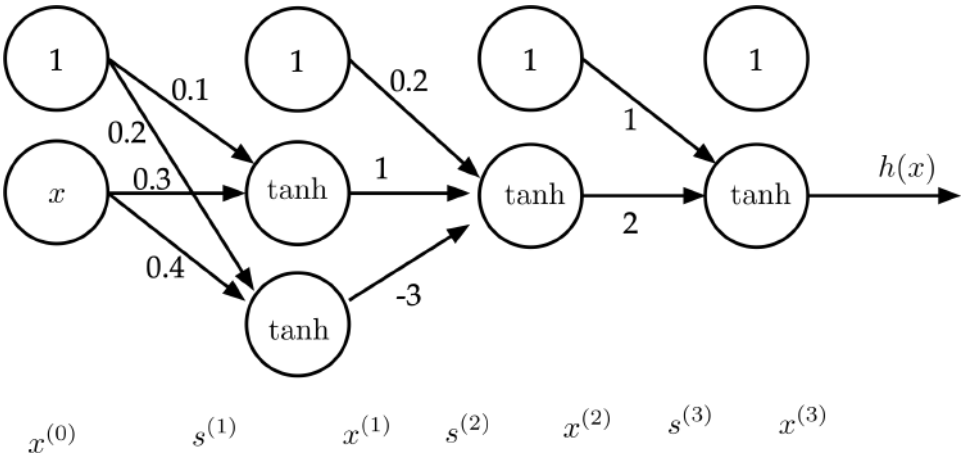


FIGURE 11. Forward Propagation Graph for Example

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix} 0.2 & 1 & -3 \end{bmatrix}, \quad \mathbf{W}^{(3)} = \begin{bmatrix} 1 & 2 \end{bmatrix}.$$

The forward pass of the algorithm first generates

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ x \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

This is multiplied by the weight matrix $\mathbf{W}^{(1)}$ to get $\mathbf{s}^{(1)}$

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)}\mathbf{x}^{(0)} = \begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 1 \end{bmatrix}.$$

We then pass $\mathbf{s}^{(1)}$ through the activation functions to get

$$\mathbf{x}^{(1)} = \begin{bmatrix} 1 \\ \tanh(\mathbf{s}_1^{(1)}) \\ \tanh(\mathbf{s}_2^{(1)}) \end{bmatrix} = \begin{bmatrix} 1 \\ \tanh(0.7) \\ \tanh(1) \end{bmatrix} = \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix}.$$

These outputs are then passed through $\mathbf{W}^{(2)}$ to produce $\mathbf{s}^{(2)}$ in the second layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)}\mathbf{s}^{(1)} = \begin{bmatrix} 0.2 & 1 & -3 \end{bmatrix} \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix} = -1.48$$

and passing this through the activation function gives

$$\mathbf{x}^{(2)} = \begin{bmatrix} 1 \\ \tanh(\mathbf{s}^{(2)}) \end{bmatrix} = \begin{bmatrix} 1 \\ -0.90 \end{bmatrix}.$$

The input to the last layer is,

$$\mathbf{s}^{(3)} = \mathbf{W}^{(3)}\mathbf{x}^{(2)} = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -0.90 \end{bmatrix} = -0.80$$

which when passed through the last layer's activation function gives

$$h(x) = \tanh(\mathbf{s}^{(3)}) = \tanh(-0.8) = -0.66.$$

With the forward pass complete, we now have $\{(\mathbf{x}^{(\ell)}, \mathbf{s}^{(\ell)}\}$ for $\ell = 1, 2, 3$ that are needed for the backward pass. The backward pass computes the risk sensitivity with respect to each input signal, $\mathbf{s}^{(\ell)}$, starting from the predicted output $h(x)$ and comparing it to the actual target, y. This gives,

$$\begin{aligned} \delta^{(3)} &= \frac{\partial L}{\partial \mathbf{s}^{(3)}} = 2(\mathbf{x}^{(3)} - y)\sigma'(\mathbf{s}^{(3)}) \\ &= 2(\mathbf{x}^{(3)} - y)(1 - (\mathbf{x}^{(3)})^2) \\ &= 2(-0.66 - 1)(1 - (-.66)^2) = -1.855. \end{aligned}$$

The next sensitivity is computed as

$$\begin{aligned} \delta^{(2)} &= \frac{\partial L}{\partial \mathbf{s}^{(2)}} = \sigma'(\mathbf{s}^{(2)}) \otimes \left[(\mathbf{W}^{(3)})^T \delta^{(3)}\right]_1^{d^{(2)}} \\ &= ([1 - .99^2]) \cdot 2 \cdot -1.855 = -0.69 \end{aligned}$$

and the final sensittivity is

$$\begin{aligned}\delta^{(1)} &= \frac{\partial L}{\partial \mathbf{s}^{(1)}} = \sigma'(\mathbf{s}^{(1)}) \otimes \left[(\mathbf{W}^{(2)})^T \delta^{(2)}\right]_1^{\delta^{(1)}} \\ &= \begin{bmatrix} 0.634 \\ 0.4199 \end{bmatrix} \otimes \begin{bmatrix} -.694 \\ 2.083 \end{bmatrix} = \begin{bmatrix} -0.44 \\ 1.67 \end{bmatrix}.\end{aligned}$$

This backward chain of computations may be visualized in Fig. 12 using the results of the forward pass. Note that this graph has the same structure as the forward pass' graph. The main differences are the removal of the bias nodes and the reversed direction of the computational flow.

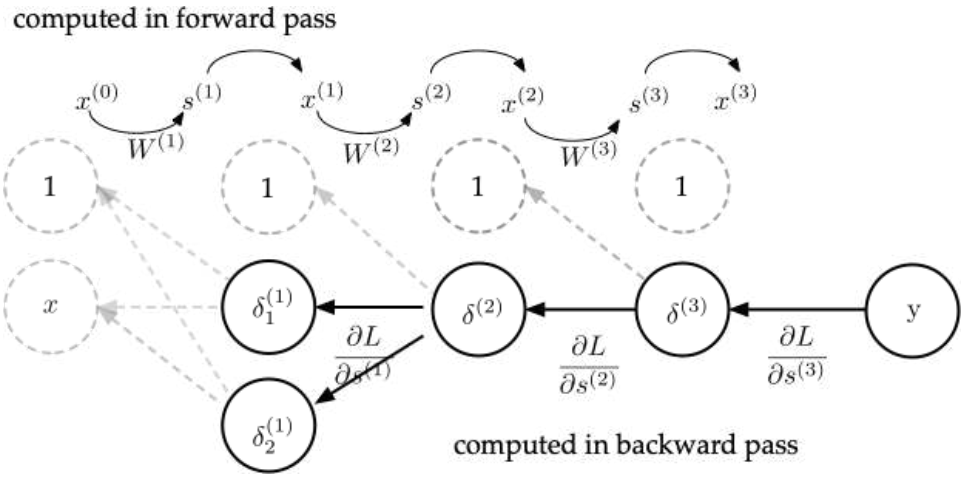


FIGURE 12. Backward Propagation to compute sensitivities $\delta^{(\ell)}$

Now that we have the sensitivities we can compute the gradient of the risk with respect to each weight matrix, $\mathbf{W}^{(\ell)}$,

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(1)}} &= \mathbf{x}^{(0)}(\delta^{(1)})^T = \begin{bmatrix} -0.44 & -0.88 \\ -0.88 & 1.75 \end{bmatrix} \\ \frac{\partial L}{\partial \mathbf{W}^{(2)}} &= \mathbf{x}^{(2)}(\delta^{(2)})^T = \begin{bmatrix} -0.69 \\ -0.42 \\ -0.53 \end{bmatrix} \\ \frac{\partial L}{\partial \mathbf{W}^{(3)}} &= \mathbf{x}^{(2)}(\delta^{(3)})^T = \begin{bmatrix} -1.85 \\ 1.67 \end{bmatrix}.\end{aligned}$$

These are the gradients computed with respect to the given data point $x = 2$ and $y = 1$. We would then use these vectors computed above to update the weights $\mathbf{w} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}\}$ in our neural network.

Our development of the backpropagation equations extends the original treatment in [**RHW86**] to a deep network with multiple layers [**AM12**]. Writing out these equations by hand is tedious and certainly not how we go about training deep network with millions of trainable parameters. Applying back propagation to deep neural networks requires an efficient computational approach that can *automate* how the backpropagation equations are generated. This method is known as *automatic differentiation* [**BPRS18**] and is the subject of the next section.

5. Automatic Differentiation

Automatic differentiation [**BPRS18**] is an algorithmic method for computing the gradient of a function that can be represented as an equation of the form

$$z = f(x_1, x_2, \ldots, x_n)$$

where x_i are input variables ($i = 1, \ldots, n$) or parameters and for which the function, f, can be parsed into a sequence of binary or unary operations. Such functions can be represented as graph data structures which then provide the basis for computing the output z for a specific set of inputs (x_1 to x_n). In our case, we use these graph data structures to compute the gradient of f for use in the backpropagation algorithm. Note that automatic differentiation is not the same as computing the gradient using symbolic computation. Symbolic tools such as Mathematica compute algebraic expressions for the gradient. In our case, we are computing the numerical value of the gradient for a specific input/output sample, (x_k, y_k).

To illustrate what we mean by a graph data structure representing the equation, let us consider the function

$$z = f(y, x) = y - max(0, w \cdot x + b)$$

where y and x are input variables representing a sample's target and input. The other variables, w and b are parameters or weights. We may decompose this function into a list of *nodes* labeled with a node name and an expression formed by the unary or binary action on other node names or input variables. Nodes can either be designated as

- *source nodes:* These are nodes whose expression is in terms of input variables/parameters.
- *sink nodes:* These are nodes whose node name is not use anywhere else in the list.
- *hidden nodes:* These are nodes whose node name is used by other nodal expressions and its nodal expression uses symbols from other nodes.

These nodes are then used to build a graph whose vertex (nodal) set consists of the listed nodes and whose edge set consists of edges from node v_i to node v_j if v_i appears in the expression for v_j. For our given function, this generates the graph shown in Fig. 13.

This graph can be used to automate the computation of $f(x, y, w, b)$ by simply setting the values in the source nodes to their appropriate values and then traversing the graph. The graph and its nodes can be easily created by parsing out the operator tokens in the equation's expression.

This graph can also be used to compute the derivative of the function with respect to various source node variables. The evaluation is done at a specific set of values that the source nodes were initialized to. So let us consider our original graph in Fig. 13 with the initial inputs being $w = 2$, $x = 1$, $b = 1$ and $y = 5$. The left side of Fig. 14 shows the value of

node	expression	node type
v_1	w	source
v_2	x	source
v_3	b	source
v_4	y	source
v_5	$v_1 \times v_2$	hidden
v_6	$v_3 + v_5$	hidden
v_7	$\max(0, v_6)$	hidden
v_8	$v_4 - v_7$	hidden
v_9	v_8	sink

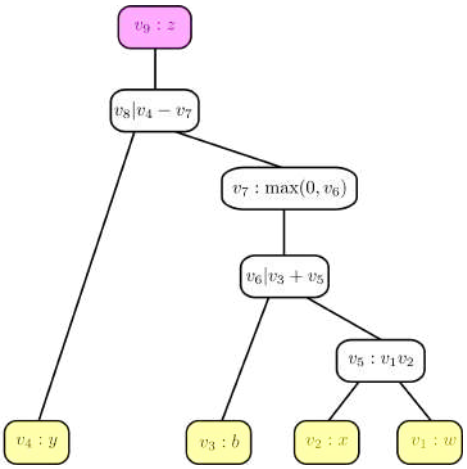


FIGURE 13. Computation Graph for $z = y - \max(0, w \cdot x + b)$

the hidden nodes and sink nodes generated by traversing the graph. These values are shown in the circle attached to each node.

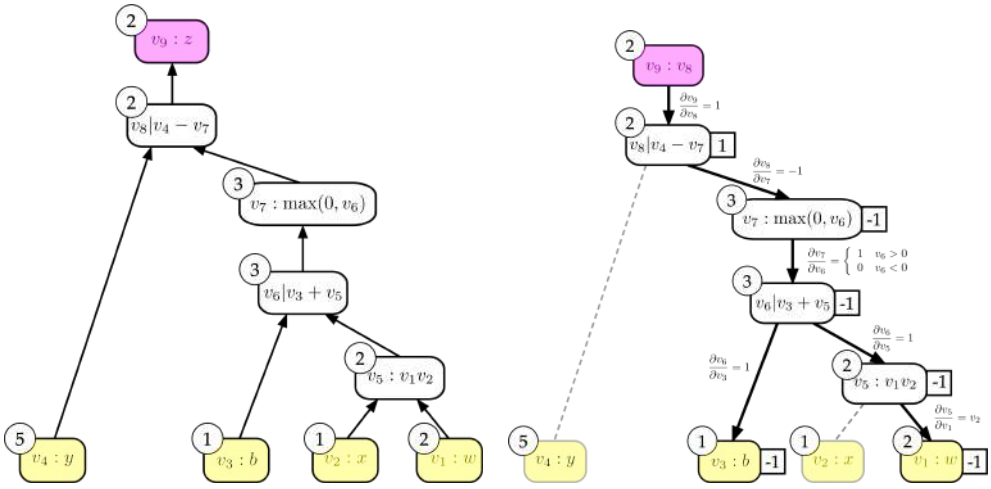


FIGURE 14. (left) forward pass through computation graph (right) backward pass through computation graph

To compute the gradient we need to calculate the partial derivatives of each hidden node variable and the sink node with respect to the nodes feeding that node. There are 4 hidden nodes, v_5, v_6, v_7, and v_8. The sink node

is v_9. For instance, if we consider the node v_5, it has two nodes, v_1 and v_2 feeding it. So there are two derivatives to consider. Letting $u(x)$ be the unit step function, we get

$$\frac{\partial v_5}{\partial v_1} = \frac{\partial(v_1 v_2)}{\partial v_1} = v_2, \quad \frac{\partial v_5}{\partial v_2} = \frac{\partial(v_1 v_2)}{\partial v_2} = v_1.$$

Repeating this for the other hidden nodes gives

$$\begin{array}{ll} \frac{\partial v_6}{\partial v_5} = \frac{\partial(v_5+v_3)}{\partial v_5} = 1 & \frac{\partial v_6}{\partial v_3} = \frac{\partial(v_5+v_3)}{\partial v_3} = 1 \\ \frac{\partial v_7}{\partial v_6} = \frac{\partial(\max(0,v_6))}{\partial v_6} = u(v_6) & \\ \frac{\partial v_8}{\partial v_4} = \frac{\partial(v_4-v_7)}{\partial v_4} = 1 & \frac{\partial v_8}{\partial v_7} = \frac{\partial(v_4-v_7)}{\partial v_7} = -1 \\ \frac{\partial v_9}{\partial v_8} = \frac{\partial(v_8)}{\partial v_8} = 1 & \end{array}.$$

We are going to compute the gradients of the output z with respect to the two parameters b and w. We first take the original graph and create the subgraph shown on the right side of Fig. 14. This subgraph reverses the directions of all edges that form a connected path from the sink node v_9 to the source nodes (v_4 and v_1) we want to take the gradient with respect to. An edge of this subgraph that goes from node v_i to v_j is labeled with an expression for the partial derivative $\frac{\partial v_i}{\partial v_j}$ that we computed above. The partial derivative, $\frac{\partial z}{\partial b}$ is then obtained by traversing the path from v_9 to v_3. Letting $u(x)$ denote the step function we get

$$\begin{aligned} \frac{\partial z}{\partial b} &= \frac{\partial v_9}{\partial v_8} \cdot \frac{\partial v_8}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_3} \\ &= 1 \times -1 \times u(v_6) \times 1 \\ &= 1 \times -1 \times 1 \times 1 = \boxed{-1}. \end{aligned}$$

The partial derivative, $\frac{\partial z}{\partial w}$ is then obtained by traversing the path from v_9 to v_1. This gives us

$$\begin{aligned} \frac{\partial z}{\partial w} &= \frac{\partial v_9}{\partial v_8} \cdot \frac{\partial v_8}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_1} \\ &= 1 \times -1 \times u(v_6) \times 1 \times v_2 \\ &= 1 \times -1 \times 1 \times 1 \times 1 = \boxed{-1}. \end{aligned}$$

The results from these computations are shown in the white squares attached to each node.

The backpropagation algorithm for deep neural networks is implemented using the computational graphs shown above. This is much more computationally efficient then direct computation of the backprop equations because the construction and traverse of the computational graph is extremely efficient. For this reason, automatic differentiation provides a computationally efficient way to compute gradients for large-scale neural networks with 10's of thousands or millions of parameters to update.

TensorFlow/Keras provides the `GradientTape` API for automatic differentiation. It is a Python object of global scope that records the tensor operations run inside it in the form of a computational graph (what TensorFlow refers to as a "tape"). This graph is then used to retrieve the gradient of any output with respect to any variable or set of variables (i.e. instances of the `tf.Variables` class).

```
import tensorflow as tf

W = tf.Variable(tf.random.uniform((2,2)))
b  = tf.Variable(tf.zeros(2,)))
x  = tf.random.uniform((2,2))

#create GradientTape object and read out the operations
# used to compute y
with tf.GradientTape() as tape:
      y = tf.matmul(x,W) + b

#use the GradientTape object (graph) to
#compute derivative of y with respect to [W, b]
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

The preceding code segment computes the gradients for $y = \mathbf{W}\mathbf{X} + b$ where $\mathbf{W}, \mathbf{X} \in \mathbb{R}^{2\times 2}$ and $b \in \mathbb{R}^2$ with respect to $\mathbf{W}$ and b for randomly chosen initial input, x and initial weights and biases. The backpropagation algorithm is configured by the `model` object's `compile` method.

This method generates the `GradientTape` object for the model and the `fit` method uses the `GradientTape` object to update the model weights. While we have discussed simple steepest gradient descent updates in equation (30), in fact that update can be done in many different ways to speed up convergence, manage overfitting, and avoiding local minima. These weight optimizers are also configured by the `compile` method and will be discussed in greater detail below. The `GradientTape` API is often used by the ML engineer when they are interested in visualizing the "meaning" of activated features in the model. We often have to use them when we need to customize the backpropagation algorithm. We will cover this use of `GradientTape` objects when we discuss deep learning for computer vision applications.

6. Mini-Batch Gradient Descent Training

Backpropagation is a gradient descent algorithm. In other words, it updates the weights, w, of the model $h_w \in \mathcal{H}$ through the recursion

$$w \leftarrow w - \eta \nabla_w \widehat{R}_{\mathcal{D}}[h_w] \tag{31}$$

where

$$\widehat{R}_{\mathcal{D}}[h_w] = \frac{1}{N} \sum_{k=1}^{N} L(h_w(x_k), y_k) \tag{32}$$

is the model's empirical risk evaluated over the data set $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$ and $L(h_w(x_k), y_k)$ is the loss in using the model to predict the target, y_k, from the input, x_k.

The key thing to note is that the weight was updated in equation (32) using all N samples in the data set, $\mathcal{D}$. Computing the gradient using equation (32) is computationally expensive since N is usually very large (10^4 samples). So in practice, we compute the gradient based on a subset of the full data set. In particular, we randomly partition $\mathcal{D}$ into batches of size $N_b < N$. This will result in $N_0 = \left\lceil \frac{N}{N_b} \right\rceil$ batches. Let $\mathcal{D}_j$ denote the

jth minibatch. The first $N_0 - 1$ batches have N_b elements of the original data set, so we can represent $\mathcal{D}_j = \{(x_{j_k}, y_{j_k})\}_{k=1}^{N_b}$ where $\{j_k\}_{k=1}^{N_b}$ are the indices for the samples in the original data set $\mathcal{D}$. The N_0th batch has $N_x = N - N_b(N_0 - 1)$ samples in it. We would then update the weights using the following update

$$w \leftarrow w - \eta \nabla_w \widehat{R}_k[h_w] \tag{33}$$

for $k = 1, \ldots, N_0 - 1$ where

$$\widehat{R}_k[h_w] = \frac{1}{N_b} \sum_{k=1}^{N_b} L(h_w(x_{j_k}), y_{j_k}). \tag{34}$$

Mini-batch gradient methods converge more quickly than methods using the entire dataset for each weight update. A training *epoch* is completed after we have cycled through all batches in the full dataset. Mini-batch training in equation (33) therefore has N_0 weight updates per epoch. The original gradient descent update in equation (31) may be seen as having a batch size of $N_b = N$ and so there is only a single update of the weight per epoch. So the mini-batch methods where $N_b < N$ are faster because they update the weight more times in a single epoch.

There is, of course, a cost associated with using mini-batch algorithms. Because mini-batch algorithms compute the empirical risk's gradient using only a portion of the data set samples, it means that $\widehat{R}_k[h_w]$ in equation (34), is only an estimate of the empirical risk in equation (32). In particular, since the batches are randomly generated, this means that $\widehat{R}_k[h_w]$ is a random variable and so the gradient in equation (33) is a noisy version of the gradient of the empirical risk. This randomness turns our original gradient descent algorithm into a *stochastic gradient descent* (SGD) algorithm. This noise level is inversely proportional to batch size. If batch size is too small, then the gradient updates are very noisy and the SGD algorithm may not converge. For larger batch sizes the algorithm will converge and jump around a minimum value. One advantage of SGD algorithms is that the noise introduced using the mini-batch can actually help perturb the weights

out of local minima and may therefore achieve a lower loss than would be seen with updates where the batch is the entire data set.

We can illustrate this impact of batch size on the earlier example from chapter 1. In that case we trained a convolutional neural network to solve the digit recognition task on the MNIST data set. We partitioned the training and testing data into batches of size 1000, 500, and 100. We then trained a model for 100 epochs using each batched data set and computed the loss on a similarly batched testing data set. The resulting training loss and validation loss as a function of epoch are shown in Fig. 15. The left hand plot shows that the training loss decreases more quickly when we train with the smaller batch sizes. The right hand plot shows that the validation loss has greater variation (noise) for the smaller batch sizes.

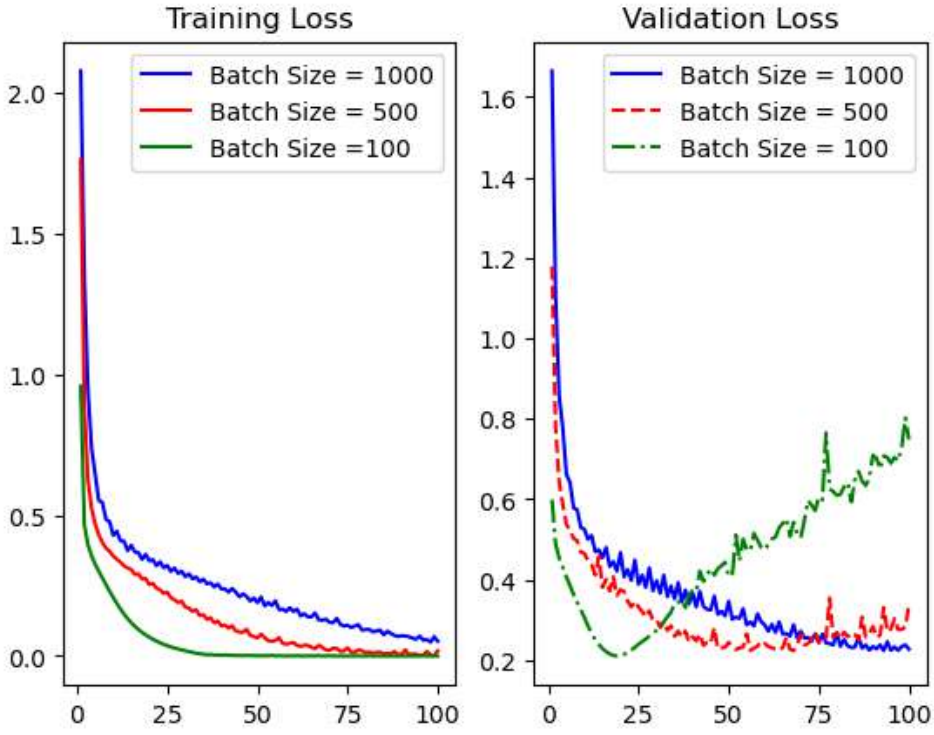


FIGURE 15. Training and Testing Loss as a function of training epoch for various batch sizes

CHAPTER 4

Training Pipelines for Deep Learning

Training pipelines or *workflows* are the procedures used to train a neural network model that can *generalize* beyond its training data. Generalization means that if the model minimizes the data set's empirical risk, that the actual risk for the model is also close to the empirical risk. Deep neural networks, however, have a very large VC dimension and this suggests that such models can *memorize* all of the samples in the data set and yet perform poorly on samples that are not in the training data. We say that such models *overfit* the training data. On the other hand, selecting a model set of limited capacity may *underfit* the training data by performing poorly on training data. One of the chief issues faced in neural network training is finding that sweet spot between overfitting and underfitting the data.

How well a model fits its data is influenced by many factors. Clearly the size of the data set is one of these factors. We saw before that we can control overfitting by limiting the number of epochs we train for. The model architecture and the number of nodes in each hidden layers will impact overfitting. Optimizer hyper-parameters such as learning rate can influence overfitting. We can also control overfitting by augmenting the loss function with a regularization kernel that constrains the model's weights. There are, therefore, a large number of tools for controlling model overfitting. ML engineers must exercise some degree of "engineering" judgement in how they use these tools to train deep neural networks.

The training process should be viewed as a series of "experiments" in which training a model for a fixed number of epochs is an experiment whose training curves provide guidance in selecting the model parameters

and optimizer hyper-parameters used in the next training experiment. This chapter's objective is to provide some insight into how these training experiments are done in practice, thereby establishing a standard workflow to be followed in solving deep learning problems. This chapter uses a single running example to illustrate the proposed workflow. The example application was first presented in chapter 1. It is concerned with the development of an app that takes the image of a handwritten digit and determines which digit between 0 to 9 it is an image of. This chapter shows how the training workflow in Fig. 1 is used to select a sequential neural network model for this problem.

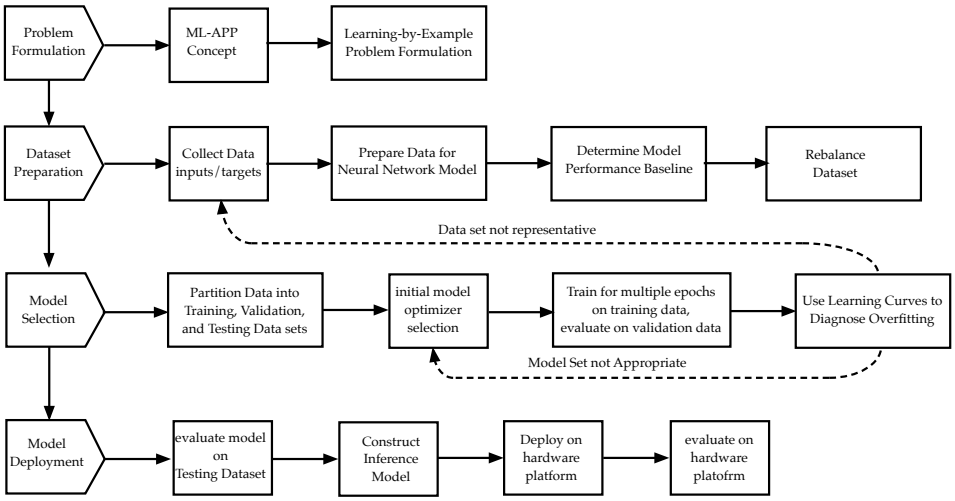


FIGURE 1. Deep Learning Application Development Workflow

The workflow in Fig. 1 has four distinct stages.

(1) *Problem Formulation* takes a customer's app concept and maps it to the learning-by-example problem described in chapter 1.
(2) *Data Preparation* involves collecting and preparing the data used in the neural network model. We use the data to identify a "baseline" model whose performance level our trained model will need

to beat. This stage of the training process also partitions the available data in a partial or *p-training* dataset, *validation* dataset and *test* dataset.

(3) *Model Selection* starts with an initial model architecture and optimizer configuration and then uses a mini-batch optimizer on the p-training dataset to fit (train) the model for several epochs. The model's loss and performance metric are evaluated after each training epoch on the p-training and validation datasets. The resulting training history is plotted as a function of epoch to form that training session's *training curves*. The training curves are used to diagnose whether the model is overfitting the p-training data and that diagnosis is used to suggest changes to the model architecture, p-training and validation datasets, optimizer hyper-parameters, and the problem's reward function. We then re-train the modified model to generate a new set of training curves and proceed in this matter until we have a model that no longer overfits the p-training data and whose performance metric on the validation dataset is deemed to be acceptable.

(4) *Model Deployment* is the final stage of our workflow. This stage evaluates the model's performance on the test dataset to assess how well the model should work in practice. We then prepare an *inference* version of our model that strips out the training portion of the code and rewrites that model to run on the platform used in our application. This inference version of the model is then evaluated in the field to see how well it actually performs with respect to the training model's performance on the test dataset.

This chapter is primarily concerned with the first three stages of the proposed workflow: problem formulation, data preparation, and model selection. The remainder of this chapter uses the digit classification application to illustrate how these workflow stages are followed in practice.

1. Problem Formulation

The problem formulation stage of ML app development forces the developer to clearly identify how the customer's application maps to the learning-by-example framework described in chapter 1. In particular, the developer wants to form a clear understanding of what the "system" is that generated the available data , have a precise understanding of the loss function used during training, and have a good metric for evaluating the trained model's performance.

This section illustrates the problem formulation stage using the handwritten digit recognition application from chapter 1. The customer for this application is the United States Postal Service (USPS). The USPS wants an app that can scan the handwritten address on a letter's envelope and then use that scanned image to automatically sort where that letter should go. We can break down this problem into the task of segmenting out each character from an image of the address, classifying which character the segmented image represents, and then outputting a string of characters for the address. As a starting point, we focus on the problem of classifying the image of a single scanned digit as being either $0, 1, \ldots, 9$. The idea being that if our app performs well on this test problem, we can expand our training to any character in the address. The customer asks for a prototype application whose *classification accuracy* is at least 99%.

A customer's description of their problem may not always be something that can be directly solved using deep learning. To see whether the USPS application is a deep learning (DL) app, we map it to the learning-by-example problem framework in chapter 1. The *system* in this problem generates the data set $\mathcal{D} = \{x_k, y_k\}_{k=1}^{N}$ of N samples. The kth input sample, x_k, is the scanned image of a digit obtained from a random sampling of letters that the USPS has processed. The kth target sample, $y_k \in \{0, 1, 2, \ldots, 9\}$, is the classification of the input image x_k as a digit $(0, 1, \ldots, 9)$ where the classification was performed by a human *observer*. The randomness in the

targets comes from the fact that several different human observers are used to classify the available input images.

The model set, $\mathcal{H}$, consists of sequential neural network models, $h : X \to [0,1]^{10}$, that map scanned images, $x \in X$, of handwritten digits onto a real 10-dimensional vector, $h(x)$, whose components are real numbers in the interval $[0,1]$. The input images are typically encoded as tensors. In our case, we are using sequential neural networks models similar to those described in chapter 3, so the inputs are vectors (rank-1 tensors) whose components are real valued and represent the value of one of the image pixels. The input set, therefore, will be taken to be $X = \mathbb{R}^n$ where n is the number of pixels in an image. The ith component ($i = 1, \ldots, 10$) of the vector, $h_i(x) \in [0,1]$ is an estimate of the probability that the input image, x, would have been classified as the digit $i-1$ by the human observer.

Note that the target set, Y, for dataset samples consists of the integers $0, 1, \ldots, 9$ whereas the model's range space consists of 10-dimensional real vectors. These are distinctly different sets because we are treating our learning-by-example problem as a multi-class logistic regression problem where the model predicts the probability of a given the input being in a specified class. We will therefore train the model using the *sparse categorical cross entropy function* discussed in chapter 3.

$$L(y_k, h(x_k)) = -\frac{1}{10}\sum_{j=1}^{10} \mathbb{1}(y_k = j-1)\log(h_j(x_k)).$$

The *performance metric* we use is the model's *classification accuracy*. In particular, given a model's prediction, $h(x)$ for a given input image, x, the model's "classification", $\widehat{y}(x) \in \{0, 1, \ldots, 9\}$ of the input will be

$$\widehat{y}(x) = (\arg\max_i h_i(x)) - 1.$$

Namely, the predicted class is based on which component of the model's output vector has the largest value. The performance metric for the model

would then be the expected value of the classification error,

$$\text{model accuracy} \quad = \quad \mathbb{E}_{\mathbf{x},\mathbf{y}}\left\{\mathbb{1}\left[\mathbf{y} = \widehat{y}(\mathbf{x})\right]\right\} \approx \frac{1}{N}\sum_{k=1}^{N} \mathbb{1}\left[y_k = \widehat{y}(x_k)\right]$$

where the second term estimates that accuracy from samples (x_k, y_k) in the available dataset.

Accuracy, however, is an average measure a classifier performance with respect to its correct classifications. In many applications, especially multi-class problems, we are interested in the per class accuracy and the likelihood of the classifications being incorrect. Such measures are better presented using a confusion matrix.

In a two class problem, the confusion matrix is shown on left side of Fig. 2. This shows a 2 by 2 grid. The x-axis represents the true target classifications and the y-axis represents the model predictions. In this case there are two classes; True (1) and False (0). Let y denote the true target and $\widehat{y}$ denote the model's prediction, then we define

- *True Positive* rate, $\mathrm{TPR} = \mathbb{E}\left\{\widehat{y} = 1 \,|\, y = 1\right\}$
- *False Positive* rate, $\mathrm{FPR} = \mathbb{E}\left\{\widehat{y} = 1 \,|\, y = 0\right\}$
- *True Negative* rate, $\mathrm{TNR} = \mathbb{E}\left\{\widehat{y} = 0 \,|\, y = 0\right\}$
- *False Negative* rate, $\mathrm{FNR} = \mathbb{E}\left\{\widehat{y} = 0 \,|\, y = 1\right\}$

The cells in the matrix show the total number of samples that were correctly classified as positive (true positive or TP), the number of samples correctly classified as negative (true negative or TN), the number of positive samples that were incorrectly classified as negative (False negative or FN), and the number of negative samples that were incorrectly classified as positive (False positive or FP). It is customary to introduce specific rates that provide a more detailed characterization of classifier performance. In particular, we define

- *Sensitivity* as $\frac{TP}{TP+FN}$,

- *Specificity* as $\frac{TN}{TN+FP}$,
- *Precision* as $\frac{TP}{TP+FP}$,
- *Negative Predictive Value* as $\frac{TN}{TN+FN}$, and
- *Accuracy* as $\frac{TP+TN}{TP+TN+FP+FN}$.

These specific error rates are shown in the binary confusion matrix. In many cases these cells are shaded to denote the value of the rate. While we show this for a binary classification problem, it can clearly be generalized to the multi-class case, thereby providing a more complete view of the classifier's performance than is provided by the single "accuracy" metric.

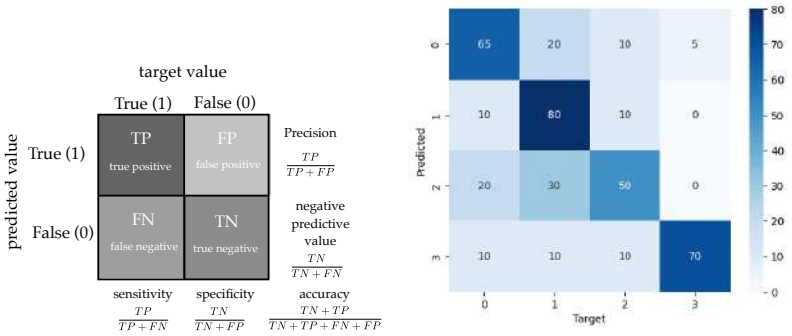


FIGURE 2. (left) Binary Class Confusion Matrix - (right) 4 class Confusion Matrix

You can quickly draw a confusion matrix in Python using the `seaborn` library. The following script draws such a heat map for a 4 class set of statistics. The resulting confusion matrix is shown on right side of Fig. 2

```
import seaborn
import numpy as np
import matplotlib.pyplot as plt

data = [[65, 20, 10, 15],[10,  80, 10,  ],
            [20,  30, 40,0],[10,   10,10, 70]]

ax = seaborn.heatmap(data, xticklabels='0123',yticklabels='0123',
                           annot = True, square = True, cmap='Blues')
ax.set_xlabel('Target')
ax.set_ylabel('Predicted')
plt.show()
```

2. Data Preparation

Data preparation is one of the most time consuming stages in developing a DL application. This stage involves the collection of the data as well as pre-processing the data so it can be directly used by a neural network model. Neural networks expect real-valued tensor inputs and the available data may not initially be in this form. For us, this stage will also include determining a baseline performance level that our trained models should beat, partitioning the available data into p-training, validation and testing data sets, and using `Dataset` objects to pre-batch the datasets so they can be more quickly fetched during the model training. This section illustrates the data preparation stage for the handwritten digit recognition application.

The database to be used in this application is the MNIST database. MNIST is a large database of scanned images of handwritten digits $(0-9)$. The database contains $60,000$ training images and $10,000$ testing images. All images are 28×28 monochrome images where each pixel is an unsigned 8 bit integer (data type `unint8`) taking values $0-255$. Each sample, (x_k, y_k), in the database consists of a scanned image $x_k \in X = \{1, 2, \ldots, 255\}^{28\times 28}$ and a target, $y_k \in Y = \{0, 1, \ldots, 9\}$. The MNIST database is included in Tensorflow. The script in Fig. 3 contains the code used to load the MNIST database and it displays one randomly drawn sample from that database.

The script in Fig. 3 shows that the shape of the training input tensor, `train_x`, is $(60000, 28, 28)$ and that the shape of the training set's target tensor, `train_y` is $(60000,)$. So the training input is a rank-3 tensor containing all of the scanned input images. The training target is a rank-1 tensor constraining all targets in the training database. The script randomly selects a single sample and displays the input image as a greyscale map, labeling it with the associated target label.

Note, however, that the `input` and `target` drawn from the database cannot be directly used as inputs to a neural network. One issue is that

PYTHON SCRIPT

```
from tensorflow.keras.datasets import mnist
(train_x, train_y), (test_x, test_y) = mnist.load_data()

import numpy as np
print(f"train_x shape = {train_x.shape}")
print(f"train_y shape = {train_y.shape}")
rindx = np.ceil(np.random.uniform(60000)-1).astype("int64")
print(f"sample {rindx} out of 60000")

input  = train_x[rindx,:,:]
target = train_y[rindx]

import matplotlib.pyplot as plt
plt.imshow(input,cmap='Greys')
plt.axis("off")
plt.title(f"Target Label = {target}, Sample = {rindx}")
```

OUTPUT

```
train_x shape = (60000, 28, 28)
train_y shape = (60000,)
sample 20355 out of 60000
```

FIGURE 3. Script and Output Loading the MNIST database and randomly selecting a single sample out

the pixel values are unsigned 8-bit integers and the neural network model expects inputs that are real-valued floating point numbers. The other issue arises because of the model set we chose. That model set contains deep sequential neural networks as described in chapter 3. Those models expected inputs that are floating point vectors rather than 28×28 integer matrices. We will therefore need to reshape and retype our training/testing samples to fit these conventions.

```
print(f"train_x shape = {train_x.shape}, train_x dtype = {train_x.dtype}")
train_x = train_x.reshape(60000,28*28).astype("float32")/255
test_x  = test_x.reshape(10000,28*28).astype("float32")/255
print(f"train_x shape = {train_x.shape}, train_x dtype = {train_x.dtype}")
```

Another important step in data preparation is developing a *baseline model* whose performance can be easily evaluated on the training data to provide a baseline performance level that our ”trained” models should beat if they are learning anything. This would indicate that our model actually learned something new during training. For the MNIST database, we can develop a baseline model for each digit class by taking the images for each class and computing an ”average” image for the classes. We then use that ”baseline” model to classify all samples in the dataset by selecting the class whose

"baseline image" is closest to the actual input image. The resulting empirical accuracy is then taken as the baseline performance level our trained models must beat. Fig. 4 shows a script used to generate this "average" model for the MNIST classes.

```
import matplotlib.pyplot as plt
class_cnt  = np.zeros(10)
class_freq = np.zeros(10)
baseline_maps = np.zeros(shape=(10,28,28))
for indx in range(db_size):
    target = train_y[indx]
    class_cnt[target]+=1
for indx in range(db_size):
    input = train_x[indx,:,:]
    target = train_y[indx]
    baseline_maps[target,:,:] += input/class_cnt[target]

figure, axis = plt.subplots(1,10)
for k in range(10):
    baseline = baseline_maps[k,:,:]
    axis[k].imshow(baseline,cmap="Greys")
    axis[k].set_axis_off()
```

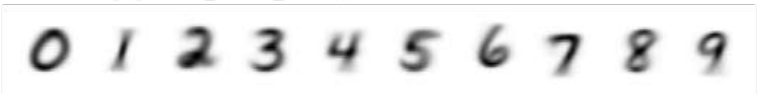


FIGURE 4. Script Generating Baseline Class Images for MNIST dataset

The images in Fig. 4 are the average baseline images for each digit class. We classify each sample in the dataset by taking the difference between the average baseline class image and the given image input. We treat the image as a vector and compute a "norm" of the resulting error vector. Note that the choice of norm greatly impacts how good our baseline model's prediction errors will be. We consider two well known norms; the RMSE (root mean squared error) and MAE (mean amplitude error). These two norms are

defined as follows. Let $\widehat{x}_k$ denote the baseline image for the kth class, then

$$\begin{aligned} \text{RMSE}(k) &= \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\widehat{x}_k - x_k)^2}, \\ \text{MAE}(k) &= \frac{1}{N}\sum_{i=1}^{N}\left|\widehat{x}_k - x_k\right|. \end{aligned}$$

Our baseline model's classification is then the class whose RMSE or MAE is smallest. The following script compute the total accuracy of our baseline model with respect to the RMSE and MAE norms. We see that the RMSE norm gives the highest total accuracy of 80%. So when we train our neural network on the dataset, we know our model is learning something if it can beat the 80% achieved using the MAE norm for classification.

```
def mae_dist(a,b):
     return np.absolute(a-b).mean()
def rmse_dist(a,b):
     MSE = np.square(np.subtract(a,b)).mean()
     return np.sqrt(MSE)

total_accuracy = 0
class_accuracy = np.zeros(10)
nsamples = 60000
for indx in range(nsamples):
      input = train_x[indx,:,:]
      target = train_y[indx]
      error = np.zeros(10)
      for k in range(10):
            baseline = baseline_maps[k,:,:]
            #error[k] = mae_dist(baseline,input)
            error[k] = rmse_dist(baseline,input)
      prediction = np.argmin(error)

      if target==prediction:
           total_accuracy+=1/nsamples
           class_accuracy[target]+=1/class_cnt[target]
```

The development of a baseline model is a critical step in the pipeline because it identifies what performance level we are trying to "beat" with our model. The preceding example used a simple statistical model based purely

norm	total_acc	0	1	2	3	4	5	6	7	8	9
RMSE	81%	87%	97%	76%	77%	82%	64%	86%	83%	71%	79 %
MAE	65%	82%	99%	45%	60%	68%	25%	77%	76%	38%	73%

TABLE 1. MNIST Baseline Model's Accuracy Performance

on the dataset. . Finally, the most common baselines are those developed by other researchers for the same dataset. These prior baseline models are actually the outcomes of another research group and being able to demonstrate that you "beat" the best prior model is critical in getting your results accepted in major machine learning conferences.

The final step is optional, but the TensorFlow library allows one to package the data in the `Dataset` objects that were introduced in chapter 1. Dataset objects have a number of methods that facilitate rescaling the data input, shuffling the inputs, and pre-batching the data. Pre-batching partitions all of the available data into batches of a fixed size that can be prefetched during training. This speeds up the training process. The following script instantiates a training and testing dataset object and pre-batches them so we can train more quickly.

```
import tensorflow as tf
train_ds = tf.data.Dataset.from_tensor_slices((train_x, train_y))
train_ds = train_ds.batch(32)
test_ds = tf.data.Dataset.from_tensor_slices((test_x, test_y))
test_ds = test_ds.batch(32)

for input,target in train_ds:
    print(f"shape of batched input = {input.shape}")
    print(f"shape of batched target = {target.shape}")
    break
```

The output from this script fetches a single batch from the batched dataset object and checks its shape. We see that the input batches have shape $(32, 784)$ and the target batches have shape $(32,)$. So each tensor represents a full batch of inputs and targets.

Note that the prior datasets had an equal number of samples for each class. In real-life datasets this may not always be the case. In medical applications, for instance, we may have many more ”normal” samples, than ”abnormal” or ”diseased” samples. Clearly it is much more important to correctly predict whether an input sample is ”diseased” than is normal. A false negative for a diseased individual has very costly negative outcomes for that individual. A false positive for a normal individual, since we can follow up with another diagnostic test that has greater precision than the original test.

Datasets where one class has many fewer samples than the other class are said to be *imbalanced*. Imbalanced datasets where the true positive are in the minority will greatly reduce the sensitivity of a binary classifier since there are relatively few true positive (TP) samples. As a result an additional data preprocessing step may be required to rebalance the data and remove biases injected into the model as a result of imbalanced data.

Datasets can be rebalanced in several ways. One approach involves using a different loss function (focal tversky loss) that weights the true positive samples more heavily that the True negative (TN) samples. But another more effective method is to resample the datasets. There are two resampling methods; undersampling and oversampling. Undersampling reduces the number of TN samples used in selecting the model. The problem with this, of course, is that we are not using all of the information. A more effective approach is *oversampling*. In this case one can builds a model for the TP sample's distribution and then resample from that distribution in order to generate new TP samples. This can be done through generative adversarial methods (discussed in chapter 7) or through clustering methods as is done with the SMOTE algorithm [**CBHK02**].

3. Model Selection

Model Selection is the third stage of the ML workflow in Fig. 1. This stage is concerned with training a model that has good performance metrics while not overfitting the training data. This stage is inherently an *experimental* process in which each training session is viewed as an experiment. The experiment's hypothesis is that a specified initial model and training configuration will lead to a model with good performance and little overfitting. If the experiment's results invalidate the hypothesis, then we modify the model architecture or change the training configuration to achieve a better outcome. We then perform another training session and continue in this manner until we have a model that performs well and does not overfit the training data.

The specific steps taken in this stage are shown in Fig. 1. These steps first involve partitioning the available training data into a partial or *p-training* data set and a *validation* data set. We then instantiate an initial model as a TensorFlow `Model` object. We use the `compile` method to configure the training optimizer. Finally we use the `fit` method to train the model for a fixed number of epochs. At each epoch, the `fit` method evaluates the loss and performance metrics on both the p-training and validation datasets. We plot the resulting p-training/validation curves and use them to assess whether the model's performance is acceptable and whether it is overfitting the training data.

If the model is unacceptable, we can either change the available training data, modify the model architecture, or reconfigure the optimizer we used for training. Modifying the model architecture may be done through changing the number of hidden layers or nodes in each layer. One could also choose a specialized architecture such as a convolutional neural network, recurrent neural network, generative neural network, or transformer. Changing the dataset is usually done to increase the amount of training data. This may be accomplished by changing the validation split or using

a sophisticated validation scheme such as k-fold cross validation. Reconfiguring the optimizer involves adjusting the type of optimizer (RMSprop versus Adam) and its hyper-parameters (learning rate). In general, these modifications require some degree of experience and judgement on the part of the developer. As a result this stage of the training workflow is also very time intensive.

The model selection stage starts with a database that has already been partitioned into a *training* and *testing* dataset. The testing dataset is reserved for evaluation of this stage's final model. This testing is done in the last stage of the workflow (Model Deployment). The training dataset is further partitioned into a "partial" or p-training dataset and a validation dataset. We use the `fit` method to train the model for a fixed number of epochs on the p-training dataset. Each epoch generates a new model and the loss and performance metrics for that sequence of models is evaluated on both the p-training and validation data for each training epoch.

This procedure generates the history of the loss and performance metric as a function of training epoch. We plot these histories as a function of epoch to obtain the *training curves* for our model. As explained in section 7 of chapter 2, each epoch expands the set of models we are searching through, thereby increasing the effective model set's complexity. So the longer we train the model, the larger the searched model set becomes and this can lead to overfitting of the training data. Overfitting occurs when the training curves show the validation loss increasing away from the p-training loss. If overfitting begins to occur after only a few epochs, this can suggest that our initial model was too complex or else that we don't have enough training data. Training curves, therefore, play a critical role in diagnosing whether our training session is generating models that overfit the training data. The rest of this section illustrates how we use training curves for the handwritten digit application described above.

Model selection for the handwritten digit app starts by instantiating an initial sequential model and then training that model for several epochs to obtain its training curves. We start with a subset of the MNIST database, using only 10,000 samples from the database's training set and then construct the dataset objects for the p-training and validation datasets assuming a batch size of 32. We will assume a 10% validation split; namely 10% of the available data is partitioned out of the training data for validation purposes.

```
from tensorflow.keras.datasets import mnist
import tensorflow as tf

(train_x, train_y), (test_x, test_y) = mnist.load_data()
train_x = train_x[:10000,:,:]
train_x = train_x.reshape(10000, 28*28).astype("float32")/255
train_y = train_y[:10000]

train_ds = tf.data.Dataset.from_tensor_slices((train_x,train_y))
train_ds.batch(32)

validation_split = 0.10
train_ds_size   = len(list(train_ds))
val_ds_size     = int(val_split*train_ds_size)
ptrain_ds_size = train_ds_size-val_ds_size

ptrain_ds  = train_ds.take(ptrain_ds_size)
val_ds      = train_ds.skip(ptrain_ds_size).take(val_ds_size)
```

For our initial model, we select a sequential network with two `Dense` hidden layers of 512 and 64 nodes, respectively, using ReLU activation functions. The output layer is `Dense` with 10 nodes and a softmax activation function. The input layer will have $28 \times 28 = 784$ nodes with a linear activation function. We use the `compile` method to configure an RMSprop training optimizer and to declare the loss function as a sparse categorical cross-entropy function.

```
from tensorflow import keras
from tensorflow.keras import layers
```

```
inputs = keras.Input(shape=(28*28))
x         = layers.Dense(512, activation="relu")(inputs)
x         = layers.Dense(64, activation="relu")(x)
outputs = layeres.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(optimizer="rmsprop",
      loss = "sparse-categorical_crossentropy",
      metrics = ["accuracy"])
```

The training session uses the `fit` method to train the model for 50 epochs. The `fit` method returns a `history` object which contains the loss and classification accuracy evaluated on the p-training and validation data set at each epoch. We use a callback function during training to save to disk the current epoch's model if it has the lowest validation loss.

```
callbacks = [
     keras.callbacks.ModelCheckpoint(
           filepath="test_model.keras",
           save_best_only = true,
           monitor = "val_loss")]

 history = model.fit(ptrain_ds, epochs =50,
             validation_data = val_ds,
             callbacks = callbacks)
```

The following script then plots the training curves for this training session, recovers the best model that was saved to disk through a callback and evaluate that best model's accuracy on the validation dataset.

```
test_model = keras.models.load_modl("test_model.keras")
best_val_loss, best_val_acc = test_model.evaluate(val_ds)

import matplotlib.pyplot as plt
train_loss = history.history["loss"]
val_loss   = history.history["val_loss"]
train_acc = history.history["accuracy"]
val_acc   = history.history["val_accuracy"]
epochs = range(1, len(train_loss)+1)

ffigure, axis = plt.subplots(1,2)
axis[0].plot(epochs, train_loss, "b--", label = "Training loss")
```

```
axis[0].plot(epochs, val_loss, "b", label = "Validation loss")
axis[0].set_title(f"Best Model Val Loss: {best_val_loss: .3f}")
axis[0].legend()
axis[1].plot(epochs, train_acc, "b--", label = "Training Accuracy")
axis[1].plot(epochs, val_acc, "b", label = "Validation Accuracy")
axis[1].set_title(f"Best Model Val Acc: {best_val_acc: .3f}")
axis[1].legend()
```

The left side of Fig. 5 shows the training curve for the initial model and datasets that we described above. There are two things to notice about the loss curves. First, the validation loss begins increasing away from the training loss after only a couple epochs. This indicates the model is overfitting. Second, the validation loss and accuracy appear to be very noisy. This can indicate the validation dataset is too small. The other thing to notice is that the best model's validation accuracy is 94.3%. This is well below the desired accuracy requested by the ML app's customer.

How poor this initial model is becomes apparent when we compare its training curves to the final model we develop below. The right side of Fig. 5 are the training curves for the model we obtained after making adjustments to the model architecture, training method, and datasets. In this case, we see that the best model has a much greater validation accuracy (97.6%) and that the loss curves do not appear to be noisy or to overfit as quickly as our initial model did.

Adjusting the model, datasets, and training procedure to achieve a good final model is done through a series of experiments whose outcomes inform the next experiment. The left side of Fig. 5 are the results for our initial model and they suggest the model is overfitting the training data. One reason for modeling overfitting may be that it has more parameters than necessary for the given amount of training data. So let us try a different model architecture in which there is only one hidden layer of 64 nodes.

```
from tensorflow import keras
from tensorflow.keras import layers
```

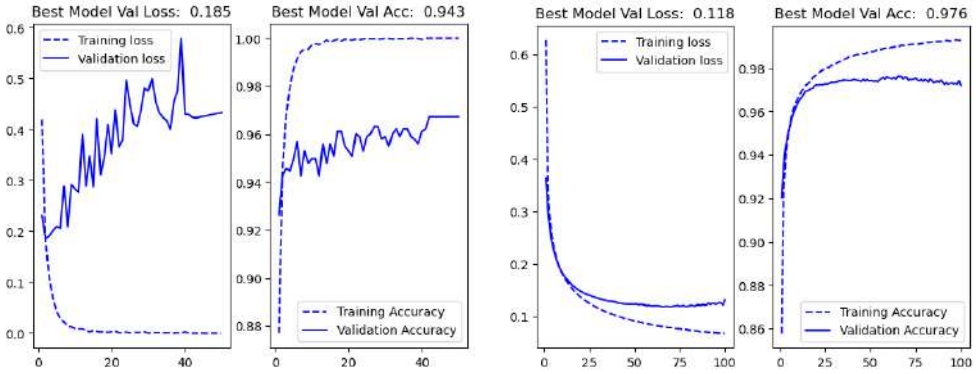


FIGURE 5. Left: training curves of initial model - Right: training curves of final model

```
inputs = keras.Input(shape= (28*28))
x = layers.Dense(64,activation="relu")(inputs)
outputs = layers.Dense(10,activation="softmax")(x)
model = keras.Model(inputs=inputs,outputs=outputs)
model.summary()
```

After retraining the smaller model with the same datasets we obtain the middle set of training curves in Fig. 6. We see that the ”noise” in the validation curves has disappeared and that the validation loss now exhibits the characteristic U-shape. In this case, the model begins overfitting around epoch 10 (rather than immediately) and the best model's validation accuracy is 94%, still too low. One other thing that impacts overfitting is the amount of training data we have. Our original dataset had 60000 training samples and we only used 10000 of them. So let us retrain the smaller model with 30000 samples. The training curves for this model/dataset are shown on the right hand side of Fig. 6. We see that overfitting still occurs around epoch 10, but that the best model's validation accuracy improves to 95.8%.

We still have an overfitting problem with the model. The last thing we can try to change is the training procedure. Recall that training ”explores” the model set and we can manage how that exploration is done by changing

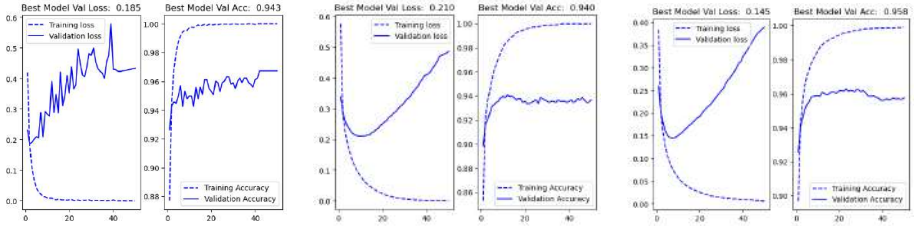


FIGURE 6. training curves for (left) initial model with two hidden layers with 512 and 64 nodes and 10000 training samples, (middle) model with a single hidden layer of 64 nodes and 10000 training samples, (right) single hidden layer model with 30000 training samples.

the batch size used in mini-batch gradient descent, changing the optimizer, or by augmenting the loss function with a regularization component. Let us try all 3 of these methods and see what happens.

- We will first increase the batch size from 32 to 256. This will slow down the training process and the results from this experiment are shown on the left side of Fig. 7. This postpones overfitting until epoch 25 and its best model has a validation accuracy of 96.1%.
- The next thing we can do is modify the loss function by adding some L2 weight regularization. Essentially, this penalizes models that whose weight tensors have large L2 norms. We can add regularization into the layer declaration.

```
inputs = keras.Input(shape= (28*28))
x = layers.Dense(64,
                 kernel_regularizer = regularizers.l2(0.001),
                 activation="relu")(inputs)
outputs = layers.Dense(10,activation="softmax")(x)
model = keras.Model(inputs=inputs,outputs=outputs)
```

The middle plot in Fig. 7 shows that this flattens out the validation loss curve, but leaves the best model's validation accuracy nearly the same (95.9%). Note that we increased the number of training epochs since regularization tends to slow down training.

- The final thing we do is introduce a different optimizer. We will talk about the various optimizers below. We started with the RMSprop optimizer. We now switch to an Adam optimizer when we compile the model.

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1.e-3),
     loss = "sparse_categorical_crossentropy",
     metrics = ["accuracy"])
```

  The left side of Fig. 7 shows training curves that exhibit little overfitting and with a best model that achieves a validation accuracy of 97%. Note that we trained the model for 100 epochs since the model seems to converge slowly.

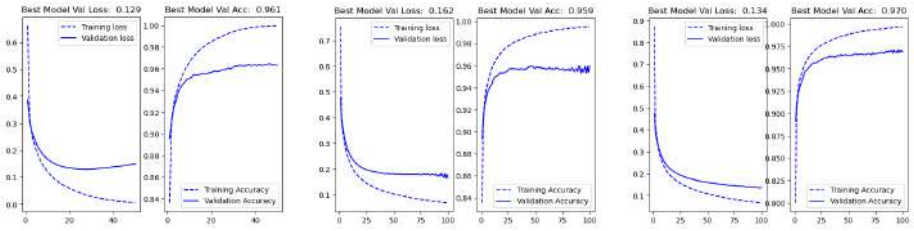


FIGURE 7. training curves for simple model with single hidden layer of 64 nodes and 30000 training samples, (left) batch size = 256, (middle) batch size = 256 and L2 regularization, (right) batch size = 256, L2 regularization, and Adam optimizer

The series of experiments whose training curves are in Figs. 6 and 7 depict a steady progression of model improvement. The final training configuration used a model with a single hidden layer of 64 nodes, a training set with 30000 samples with 10% held out for validation. We used a mini-batch Adam optimizer with a batch size of 256, a sparse catogorical cross-entropy loss function with L2 regularization. We know, however, that more training data can greatly improve our model's accuracy. So we now take the above training configuration and change it so that all 60000 training samples are used in the p-training dataset and we take the 10000 testing samples as the

validation dataset. The training curve for this model is shown on the left side of Fig. 5.

4. Optimizers

Gradient descent is the algorithm used to train deep neural networks. Every state of the art Deep Learning framework contains various implementations of these gradient based optimizers. These algorithms are often used as black-boxes with few practical explanations regarding their relative strengths and weaknesses. This section follows [**Rud16**]'s overview of the various optimizers used in deep learning frameworks.

There are three variants of the gradient descent algorithm, which differ in how much data is used to compute the objective function's gradient. Depending on the amount of data, one trades off the accuracy of the parameter update against the time it takes to perform that update.

The classical gradient descent method that we described earlier in our discussion of backpropagation is also called a *batch algorithm*. It compute the gradient of the cost function with respect to the weights, w, using all of the data in the training dataset.

$$w_{k+1} = w_k - \gamma \nabla_w \widehat{R}_N[w].$$

Since we compute the gradient of the empirical risk $\widehat{R}_N$ over the entire dataset, we just do one update. As discussed above, this means that gradient descent will be very slow and it is intractable for datasets that do not fit in memory.

Stochastic gradient descent (SGD) performs a parameter update for *each* training example (x_k, y_k) using the update

$$w_{k+1} = w_k - \gamma \nabla_w L(h(x_{i_k}; w_k), y_{i_k})$$

where the update example (x_{i_k}, y_{i_k}) is chosen at random from the data set. Batch gradient updates may perform redundant computations for large

datasets because many of the samples are very close to each other. Classical SGD does away with this redundancy by performing one update at a time for each sample. As discussed above it is usually much faster and can be used to learn online. Because the sample used for the update is randomly selected, the gradient estimate in each recursion is very noisy and we may therefore see large random fluctuations in the objective function as we work through the recursive steps.

Mini-batch gradient descent combines the batch gradient and classical SGD algorithm and performs an update for every mini-batch of N_b training examples

$$w_{k+1} = w_k - \gamma \nabla_w \left[\frac{1}{N_b} \sum_{i=1}^{N_b} L(h(x_{k_i}, w_k), y_{k_i}) \right] = w_k - \gamma \nabla \widehat{R}(w_k, \mathcal{D}_k)$$

where the collection $\mathcal{D}_k = \{(x_{k_i}, y_{k_i})\}_{i=1}^{N_b}$ is a randomly drawn set of samples from the full dataset (i.e. $N_b \ll N$). This reduces the variance in the weight updates and can make use of highly optimized matrix optimizations that make computing the gradient very computationally efficient. Common mini-batch sizes range between 50 and 256. Mini-batch gradient descent is typically what is used in training deep neural networks.

The mini-batch algorithm's convergence performance faces a number of issues that are addressed with the variations on the algorithm described below. These issues are

- It may be hard to find an appropriate learning rate, γ. Convergence is extremely slow if the learning rate is too small. Convergence may not occur if the learning rate is too large.
- Learning rate schedules adjust the learning rate during training by reducing the rate according to a pre-defined schedule or when the change in the objective begins to stall. These schedules and thresholds, however, are usually defined in advance and do not adapt to a dataset's characteristics.

- The same learning rate is often applied to all weights. But if the data is sparse then we may not want to update all weights with the same γ.
- If the error function is highly non-convex (common for neural networks), then it may be very difficult for the mini-batch algorithm to escape a saddle points and local minimum since the gradient is close to zero.

FIGURE 8. Impact of momentum in reducing oscillations in mini-batch SGD

One enhancement of the minibatch SGD algorithm is to use *momentum* [**Qia99**]. In risk surfaces that form narrow ravines, it is common for the SGD algorithm's estimate to oscillate around the smallest sloped part of the ravine and make very slow progress down the ravine. Momentum is a method that accelerates SGD in the relevant directions and dampens the oscillations seen in Fig. 8. It does this by adding a fraction η of the update vector of the past time step to the current update vector

$$\begin{aligned} g_k &= \nabla_w \widehat{R}(w_k, \mathcal{D}_k) \\ v_k &= \eta v_{k-1} + \gamma g_k \\ w_{k+1} &= w_k - v_k. \end{aligned}$$

The momentum hyperparameter, η is usually set to 0.9. An extremely useful variation on the preceding momentum algorithm is Nesterov's accelerated gradient (also called Nesterov momentum) [**Nes83**].

$$\begin{aligned} g_k &= \nabla_w \widehat{R}(w_k - \eta v_{k-1}, \mathcal{D}_k) \\ v_k &= \eta v_{k-1} + \gamma g_k \\ w_{k+1} &= w_k - v_k. \end{aligned}$$

This momentum implementation has significantly improved the training performance for recurrent neural networks.

Another technique for improving optimizer convergence is to adapt the updates to the slope of the risk function by adapting the learning rate. One example of such an example is Adagrad [**DHS11**]. This algorithm adapts the learning rate by using a larger learning rate for weights that are infrequently updated and smaller rates for weights that are often updated. For this reason, Adagrad is well suited to sparse data. Let $v \otimes w$ denote the component-wise multiplication of two equal length vectors v and w. Adagrad's update is shown below

$$\begin{aligned} g_k &= \nabla_w \widehat{R}(w_k, \mathcal{K}_k) \\ n_k &= n_{k-1} + g_k \otimes g_k \\ w_{k+1} &= w_k - \frac{\gamma}{\sqrt{n_k} + \epsilon} \otimes g_k. \end{aligned}$$

Note that the learning rate $\frac{\gamma}{\sqrt{n_k} + \epsilon}$ is a vector of the same shape as n_k. So this update rule uses a different learning rate for each component of the weighting vector w. Note that the operations in the update equations are component-wise vector multiplies, $\otimes$. Essentially, what Adagrad does is divide the base learning rate γ for each weight parameter by the sum of the square of the past gradients for that parameter.

The problem with Adagrad is that this sum is constantly increasing so that eventually the effective learning rate becomes vanishingly small. RMSprop [**TH12**] is a variation on Adagrad that replaces the sum in n_k with a decaying mean parameterized by the forgetting factor μ. The update equations, therefore take the form

$$\begin{aligned} g_k &= \nabla_w \widehat{R}(w_k, \mathcal{D}_k) \\ n_k &= \mu n_{k-1} + (1-\mu) g_k \otimes g_k \\ w_{k+1} &= w_k - \frac{\gamma}{\sqrt{n_k} + \epsilon} \otimes g_k \end{aligned}$$

where $\frac{\gamma}{\sqrt{n_k}+\epsilon}$ is a vector with the same shape as n_k. Because the component of n_k are decaying averages of the past squared gradients, the effective learning rate does not go to zero asymptotically and can "adapt" to changes in the squared gradient as we traverse the weight space.

Adagrad and RMSprop are methods that adapt the learning rate, whereas the momentum based algorithms prevent the update direction from oscillating too much. These two techniques are combined in the adaptive moment estimation (Adam) algorithm [**KB14**]. Adam combines classical momentum (using a decaying mean instead of a decaying sum) with RMSprop. Adam's update equations are given below. In these equations β_1 and β_2 are forgetting factors used to smooth the past gradients and squared gradients, respectively. Hyperparameters β_1^b and β_2^b are constants between $(0, 1)$ used for bias correction terms that offset some instances of algorithm instability.

$$\begin{aligned}
g_k &= \nabla_w \widehat{R}(w_k, \mathcal{D}_k) \\
m_k &= \beta_1 m_{k-1} + (1-\beta_1) g_k \\
\widehat{m}_k &= \frac{1}{1-\beta_1^b} m_k \\
v_k &= \beta_2 v_{k-1} + (1-\beta_2) g_k \otimes g_k \\
\widehat{v}_k &= \frac{1}{1-\beta_2^b} v_k \\
w_{k+1} &= w_k - \gamma \frac{1}{\sqrt{\widehat{v}_k}+\epsilon} \otimes \widehat{m}_k.
\end{aligned}$$

Of the preceding optimizers, Adam and RMSprop are used most often in deep learning. Let us see how SGD with momentum, Adagrad, RMSprop and Adam compare to each other on a sentiment classification problem trained on a set of Movie reviews for the Internet Movie Database (IMDB) used in [**Cho21**]. We will discuss this classification in more detail in later chapters that study natural language processing. The training scripts may be found in those chapters. The IMDB database is split into 25,000 reviews for training and 25,000 reviews for testing. Each set is split evenly between positive and negative reviews. The words in each review are encoded as

integers, where we only encode the first 10,000 words appearing most often in the reviews. Each review consists of a varying number of words and cannot therefore be directly used by a neural network expecting an input vector of fixed dimension. So we will use *multi-hot encoding* of the integer encoded review to obtain an input of fixed length. In particular, the input will be a $10,000$ length vector whose kth component is 1 if a word with integer index k appears in the review. The kth component is 0 if the word with index k never appears in the review. The model takes this $10,000$ dimensional binary vector and maps it to 0 or 1 to indicate if the review's sentiment is "positive" or "negative". The following script loads the IMDB database from TensorFlow's `datasets` library, multi-hot encodes the integer encoded reviews and then splits the training data in a ptraining and validation data set.

```
from tensorflow.keras.datasets import imdb
import numpy as np

num_words = 10000

(train_x,train_y), (test_x,test_y) = imdb.load_data(num_words = num_words)

def encode(sequences, dim = 10000):
     results = np.zeros((len(sequences), dim))
     for i, sequence in enumerate(sequences):
          for j in sequence:
               results[i,j]=1
     return results

train_x = encode(train_x)
train_y = np.asarray(train_y).astype("float32")
```

We then form the dataset objects using a 40% validation split with a batch size of 512.

```
import tensorflow as tf
batch_size = 512
train_ds = tf.data.Dataset.from_tensor_slices(train_x,train_y))
train_ds = train_ds.batch(batch_size)
train_ds_size = len(list(train_ds))
```

```
val_split = 0.4
val_size = int(val_split*train_ds_size)
ptrain_size = train_ds_size-val_size
ptrain_ds = train_ds.take(ptrain_size)
val_ds = train_ds.skip(ptrain_size).take(val_size)
```

We use a simple neural network with one hidden layers of dimension 4 and an ReLU activation function. The output layer is a single node that is densely connected to the nodes of the second hidden layers and uses a sigmoid $(1 + e^{-x})^{-1}$ activation function. We will train this model using various optimizers assuming a binary cross-entropy loss function. The following script uses a function `encode` to build the network and then selects an `optimizer`. The following script selects the SGD optimizer with Nesterov momentum.

```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import optimizers

def build_model(num_input, num_nodes, num_layers):
     inputs = keras.Input(shape = (num_input))
     x = layers.Dense(num_nodes, activation="relu")(inputs)
     for k in range(num_layers-1):
          x = layers.Dense(num_nodes, activation = "relu")(x)
     outputs = layers.Dense(1, activation = "sigmoid")(x)
     model = keras.Model(inputs=inputs, outputs=outputs)
     return model

num_input = num_words
num_nodes = 4
num_layers = 1
model = build_model(num_input,num_nodes, num_layers)

learning_rate = 0.001
optimizer = optimizers.SGD(learning_rate, momentum=.1,nesterov=True)
#optimizer = optimizers.Adagrad(learning_rate)
#optimizer = optimizers.RMSprop(learning_rate)
#optimizer = optimizers.Adam(learning_rate)
model.compile(
     optimizer = optimizer,
```

```
loss = "binary_crossentropy",
metrics = ["accuracy"])
```

The training curves for these different optimizers are shown in Fig. 9. This figure has 3 plots, one showing the training/validation loss for SGD/Adagrad, Adagrad/RMSprop, and RMSprop/Adam, all using the same learning rate of 0.001. From the figure we see that SGD, even with Nesterov momentum, learns very slowly with the given learning rate. Adagrad reduces the loss more quickly than SGD, but it still converges very slowly. If we had chosen a larger learning rate, these methods would have converged more quickly. When comparing Adagrad against RMSprop (same learning rate), we see that RMSprop converges much more quickly and begins to show signs of overfitting after the eighth epoch. The final training curve in the figure compares RMSprop and Adam for the same learning rate. We see that both have very similar curves, except for the validation loss after the models begin to overfit. In this case, we see that Adam's validation loss grows more slowly than RMSprop's validation loss.

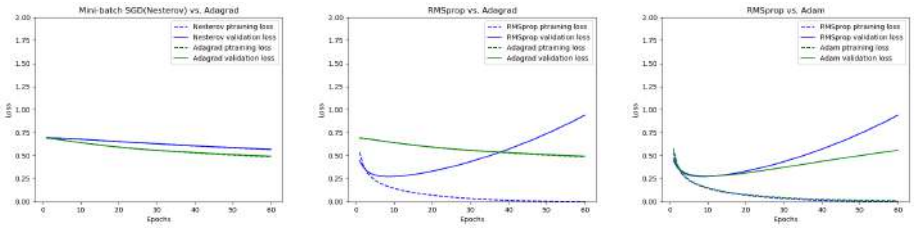


FIGURE 9. Comparison of training curves for various optimizers on a simple model for the IMDB sentiment analysis problem

Which optimizer should be used? The answer depends on the type of data and problem you are dealing with. If your input data is sparse, then you will probably achieve the best results using one of the adaptive learning-rate methods such as RMSprop or Adam. RMSprop is a simpler extension of Adagrad that addresses Adagrad's diminishing learning rates. In the above example, this was crucial to obtain a reasonable learning rate. It should be

noted that the selection of the optimizer's hyper-parameters can have a big impact on the training curves. In practice, ML engineers usually pick an optimizer whose hyper-parameters they find easy to tune. This tuning of hyper-parameters in the model, optimizer, and compiler are all critical steps in model training. Aside from dataset collection/curation, the selection of hyper-parameters is the most time consuming part of developing a model.

5. Norm Regularizers

Norm regularization is a tool that is used to prevent overfitting. It usually works by augmenting the objective function (i.e. empirical risk) with an additional term that penalizes some norm of the weight vector. Penalizing the weight vector in this way essentially reduces our search of the model space, $\mathcal{H}$, to a smaller set whose effective VC dimension is smaller, thereby preventing overfitting.

To provide a graphic illustration of the benefits of regularization, let us consider a regression problem that tries fitting samples of a lower order polynomial function, $q(x)$, with a higher order polynomial. So we consider the problem of finding the weights $\{w_i\}_{i=1}^M$ that minimize

$$J(w) = \sum_{k=1}^{N} \left(q(x_k) - \sum_{j=1}^{M} w_j x_k^j \right)^2$$

where $\{x_k\}_{k=1}^N$ is a set of $N = 10$ randomly chosen real numbers in the interval $[-1, 1]$. The observations for input x_k from the target function are $q(x_k) = x_k + 0.2x_k^2 - x_k^3 + \nu_k$ where ν_k is a zero mean i.i.d. random variable with finite variance. Fig. 10 (left) shows what happens if we select a polynomial model $\sum_{j=1}^M w_j x_k^j$ of order $M = 6$ that minimizes the mean squared error $J(w)$. This is clearly a very poor fit and if we look at the weight vector w, we see the weights vary over a large range from 0.02 up to 4.0. This large variation in weights is what gives rise to the large variations seen in the plot.

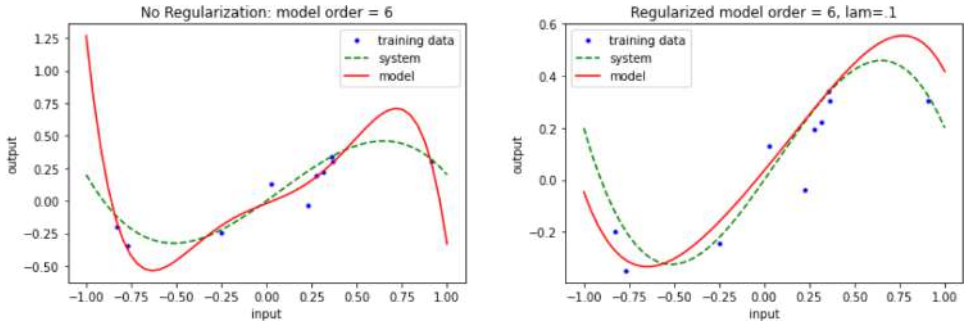


FIGURE 10. (left) regressor fit using 6th order model without regularization. (righ) regressor fit using 6th order model using norm regularization on the weight vector and a regularization penalty $\lambda = 0.1$

We will try to address this "overfitting" by introducing a penalty for the large weights. The easiest way of doing this is to augment $J(w)$ to

$$J_\lambda(w) = \sum_{k=1}^{N} \left(q(x_k) - \sum_{j=1}^{M} w_j x_k^j \right)^2 + \lambda \sum_{j=1}^{M} w_j^2$$

where $\lambda > 0$ is a positive real constant we select to adjust how large the penalty should be on the weight vector's Euclidean 2-norm. To derive the optimal weight for this augmented cost, let

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^M \\ 1 & x_2 & x_2^2 & \cdots & x_2^M \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^M \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} q(x_1) + \nu_1 \\ q(x_2) + \nu_2 \\ \vdots \\ q(x_N) + \nu_N \end{bmatrix}.$$

We can then write our augmented objective as

$$J_\lambda(w) \quad = \quad |\mathbf{X}w - \mathbf{Y}|^2 + \lambda |w|^2$$

Taking the derivative with respect to w and setting to zero gives

$$(\mathbf{X}^T\mathbf{X}w - \mathbf{X}^T\mathbf{Y}) + \lambda w = 0$$

which we rewrite as

$$\left[\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I}\right] w - \mathbf{X}^T\mathbf{Y} = 0$$

Assuming the matrix in the square brace is invertible, our regularized weight vector would be

$$w_r = \left[\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right]^{-1}\mathbf{X}^T\mathbf{Y}$$

We can then take this formula for the solution as our "regularized" model. Fig. 10 (right) shows how well this regularized 6th order model works for our given problem using a penalty term $\lambda = 0.1$. What should be apparent is that this is a much closer fit to the true target function.

The result in Fig. 10 (right) was obtained with a penalty term $\lambda = 0.1$. One might ask how does this change if we chose a different λ. In particular, if $\lambda = 0$ then we have the original problem and so there should be a large mean squared error (MSE) for the selected model. If λ is large, however, we would expect very small weights and this suggests the error could again be large. This means there is probably a "optimal" choice for the penalty parameter, λ. Fig. 11 plots the empirical risk, $\widehat{R}_\lambda(h)$, and the true risk $R_\lambda(h)$ for various λ ranging between 0 and 1. This plot indeed shows that the empirical risk increases with λ, but that the true risk shows a definite minimum for a very small value of the regularization parameter.

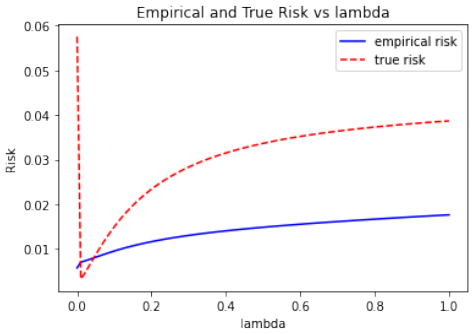


FIGURE 11. MSE versus the regularizing hyperparameter λ

Norm regularization as shown above is a common technique used to address overfitting in deep neural networks. The particular approach we demonstrated on the regression problem is known as L_2 regularization since we augmented the empirical risk function with the Euclidean 2 norm of the weights. An alternative choice for the norm is the L_1 norm. In this case the

augmented cost function takes the form

$$\begin{aligned} J_\lambda(w) &= \sum_{k=1}^{N}(q(x_k) - \sum_{j=1}^{M} w_j x_k^j)^2 + \lambda \sum_{j=1}^{M} |w_j| \\ &= |\mathbf{X}w - \mathbf{Y}|^2 + \lambda |w|_1 \end{aligned}$$

where $|w|_1 = \sum_{k=1}^{M} |w_k|$ is the L_1 norm of the vector w. Let us assume that the data matrix, $\mathbf{X}$, singular value decomposition to write

$$\mathbf{X}^T\mathbf{X} = \mathbf{U}^T\mathbf{\Sigma}\mathbf{U}$$

where $\mathbf{U}$ is a unitary matrix and $\mathbf{\Sigma} = \mathrm{diag}(\sigma_i)$ is a diagonal matrix with non-negative diagonal entries, σ_i. We then define the new vectors $v = \mathbf{U}w$ as a new "weight" vector. This would allow us to write our augmented cost in terms of the transformed weight as

$$\begin{aligned} J_\lambda(v) &= \left|\sqrt{\mathbf{\Sigma}}v - \mathbf{Y}\right|^2 + \lambda |v|_1 \\ &= \sum_{k=1}^{N} \left[(\sqrt{\sigma_k} v_k - y_k)^2 + \lambda |v_k|\right] \end{aligned}$$

Minimizing each term in the above summation shows that

$$v_k^* = \mathrm{sign}(y_k) \max\left\{\frac{|y_k|}{\sqrt{\sigma}_k} - \frac{\lambda}{\sqrt{\sigma}_k}, 0\right\}$$

Note that if $|y_k| \leq \lambda$ then the we simply take the weight $v_k^* = 0$. In other words, L_1 regularization has a tendency to zero out those weights for which the associated input is too small. This leads towards sparse weight representations which effectively reduce the number of "parameters" in the given model. For this reason, the use of L_1 regularizers tends to produce sparse models whose weights can be seen as identifying "features" in the given system.

Let us apply the L2 regularizer to our earlier IMDB example. In this case, we'll start with a model with 15 hidden layers of dimension 4. With a model this deep, we expect it to begin overfitting almost immediately. The question is whether L2 regularization can postpone overfitting of the model.

In this case, we used a batch size of 300 with the regularization parameter being $\lambda = 0.01$. Fig. 12 show the training curves with and without L2 regularization. The right plot shows the loss curves and here we clearly see that an appropriate level of L2 regularization controls the amount of model overfitting. The regularized model's validation curve does not begin getting larger as we increase the number of training epochs. As a result the best validation *accuracy* achieved by this model is slightly better than that obtained without regularization (0.855 versus 0.881).

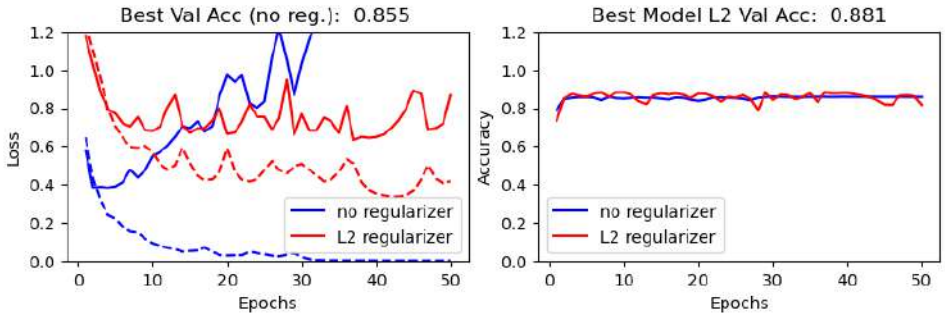


FIGURE 12. Training and Accuracy Curves for IMDB example with L2 regularization

6. Dropout Regularization

Dropout [**SHK+14**] is one of the most effective and commonly used regularization techniques for neural networks. Dropout has been shown to be first-order equivalent to L_2 regularization [**WWL13**]. But perhaps a better way to understand it [**GBC16**] is as a very efficient alternative to bagging predictors [**Bre96**].

Bagging is a technique for reducing generalization error by combining several models. The idea is to train several models separately, then have all models vote on the output for test examples. This may be impractical when each model is a large neural network since training and validation of such model is costly in time and memory.

Dropout, on the other hand may be thought of as a way to making bagging practical for ensembles of very large networks. Specifically, dropout trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network. In many deep neural networks built from linear machines with outputs passing through a nonlinearity, one can effectively remove a unit from the network by multiplying its output value by zero.

TensorFlow implements the dropout concept by introducing a layer after the hidden layer. This layer sets the outputs of the hidden layer to 0 with a frequency of specified `rate` at each step during training. This layer is only active during the training of the model. If the model is being used for inference (testing or validation), then the layer drops no values of the outputs. For our earlier IMDB example network, we would therefore declare the network with dropout as

```
def build_model_dropout(num_input, num_nodes,num_layers, rate):
    inputs = keras.Input(shape = (num_input))
    x = layers.Dense(num_nodes, activation="relu")(inputs)
    x = layers.Dropout(rate)(x)
    for k in range(num_layers-1):
        x = layers.Dense(num_nodes, activation="relu")(x)
        x = layers.Dropout(rate)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs=inputs,outputs=outputs)
    return model

rate = .2
model3 = build_model_dropout(num_input, num_nodes, num_layers, rate)
```

This example uses 20% dropout after each hidden layer. Fig. 13 shows that this level of dropout clearly reduces overfitting since the validation loss no longer has the U-shape where it gets larger for more training epochs. The validation accuracy achieved with dropout, in this case, is the same as that obtained without dropout.

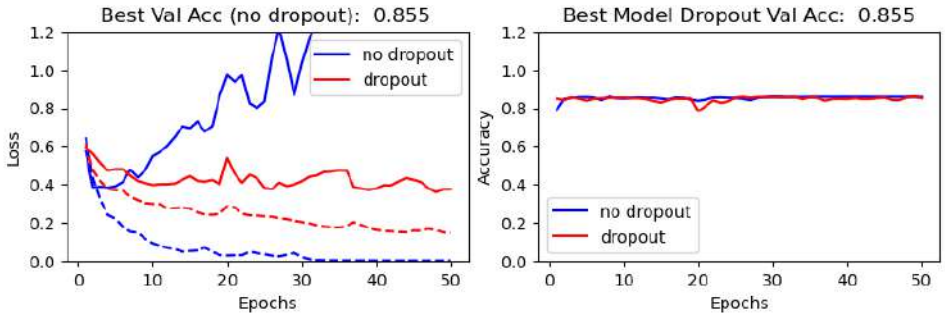


FIGURE 13. Training and Accuracy Curves for IMDB example with 20% Dropout

7. Diagnosing Model Performance with Training curves

Training curves are widely used diagnostic tools in machine learning for algorithms that learn incrementally. During training time, we evaluate model performance on both the training and hold-out validation dataset and we plot this performance for each training epoch. Reviewing training curves for a model during training is used to diagnose problems with learning, such as an underfit or overfit model, as well as whether the training and validation datasets are suitably representative. This section discusses methods in which training curves are used to diagnose underfitting or overfitting and to diagnose issues with data representation. These curves can also be used to sometimes suggest the type of changes we need to make to improve training performance. There are three basic conditions we want to identify; underfit, overfit, or optimal fit. We will take a closer look at each with examples.

Underfit training curves: Underfitting refers to a model that has not adequately learned the training dataset to obtain a sufficiently low training loss. There are two common signals for underfitting. First, our training loss curve may show a flat line or noisy values of relatively high loss, indicating that the model was unable to learn from the training dataset at all. An example is provided below in Fig. 14 and is common when the model has insufficient capacity to capture the dataset's complexity. The recommended solution to

this problem is to first add more observations to the dataset (augmentation). Another approach is to add a richer set of features to the dataset. If your model has been regularized, you can reduce the regularization parameter, or increase model capacity by making the network deeper. An underfit model may also be identified by training and validation losses that are continuing to decrease at the end of the plot. This indicates that the model is capable of further learning and that the training process was halted prematurely. In this second case, we simply increase the number of epochs or increase the learning rate to speed up training.

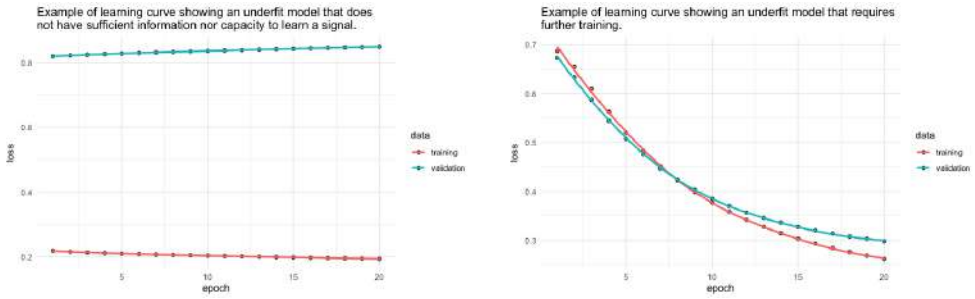


FIGURE 14. (left) training curve showing underfitting due to insufficient data or model capacity (right) training curve showing underfitting due to premature stopping of training.

Overfit training curves: Overfitting refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset. The problem with overfitting is that the more specialized the model becomes to the training data, the less likely it is to generalize to new data, thereby increasing its generalization error. Overfitting is apparent when the training loss continues to decrease while the validation loss has stopped decreasing has begun to increase. Having a model overfit is not necessarily a bad thing. It signals that the model has extracted all of the signal that the particular model could learn. The issues to be concerned about with overfitting is the magnitude and the inflection point.

A model that overfits early and has sharp "U" shape often indicates over-capacity and/or a learning rate that is too high. This situation is shown in Fig. 15. The solution involves involves using either weight decay (L2 or L1) regularization or dropout. One can also address this problem by reducing the learning rate. One can also simply reduce the number and/or size of the hidden layers, thereby reducing the complexity of the model set. Often we can minimize overfitting, but rarely can we completely eliminate it while still minimizing our loss. The right side of Fig. 15 shows an example where we have minimizing overfitting and yet not completely removed it.

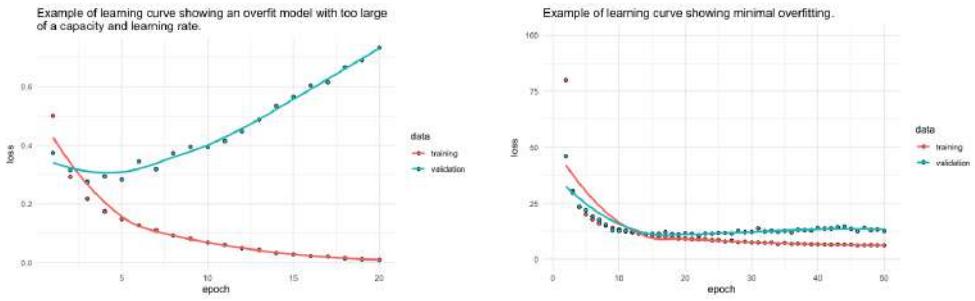


FIGURE 15. (left) training curve showing overfitting due to excess capacity over too high learning rate (right) training curve shows minimal overfitting.

Optimal Fit training curves:. An optimal fit is the goal of the learning algorithm. The loss of the model will almost always be lower on the training dataset than the validation dataset. This means that we should expect some gap between the training and validation loss training curves. This gap is referred to as the *generalization gap*. An optimal fit, therefore, is one where 1) the training loss decreases to a stable constant value, 2) the validation loss decreases to a stable constant value, and 3) the generalization gap is small. The left side of Fig. 16 shows an example of a training curve shown an optimal fit.

Diagnosing Unrepresentative Dataset: training curves can also be used to diagnose problems with the dataset; namely whether or not the dataset is

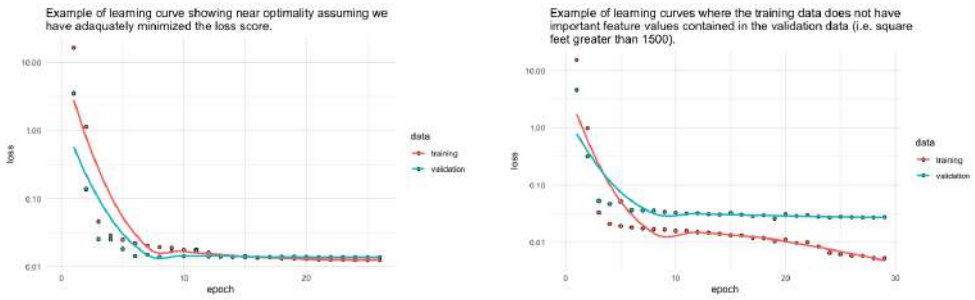


FIGURE 16. (left) training curve showing an optimal fit (right) large generalization gap occurs because validation has features not present in training dataset.

representative. An unrepresentative dataset means that the dataset may not capture the statistical characteristic relative to another dataset drawn from the same domain, such as between a training and validation dataset. This can commonly occur if the number of samples in a dataset is too small or if certain characteristics are not adequately represented. In particular, we would like to diagnose two cases; whether the training dataset or validation dataset are unrepresentative.

An unrepresentative training dataset means that the training dataset does not provide sufficient information to learn the problem, relative to the validation dataset used to evaluate it. This situation can be identified by a training and validation training curve that both show improvement, but where this is a large gap between both curves. The right side of Fig. 16 is an example of a training curve where the training dataset is unrepresentative. The obvious solution is to increase the size of the dataset or to use augmentation or cross-validation.

Cross-validation or leave-one-out training methods are often used when when the training dataset is small. The dataset is an i.i.d. sampling of the joint distribution, $F_{\mathbf{x}}(x)Q(\mathbf{y}|\mathbf{x})$, but when the number of samples is small then the variance in the empirical risk will be large. K-fold cross-validation is a way that can reduce this variance. K-fold cross-validation consists of

splitting the available data into K partitions, and training a model on $K-1$ partitions while evaluating on the one that was left out. The validation score of the model is then the average of the K validation scores. This is relatively easy to program as shown in [**Cho21**] chapter 5.

An unrepresentative validation dataset means that the validation dataset does not provide sufficient information to evaluate the ability of the model to generalize. This may occur if the validation dataset has too few examples as compared to the training dataset. This case may be identified by a training curve for training loss that looks like a good fit and a validation loss curve that shows a great deal of noise with little or no improvement. The left side of Fig. 17 shows such training curves. The solution is to add more observations to the validation dataset or perform cross-validation if the size of the dataset is limited.

Unrepresentative validation datasets may also be diagnosed if the validation loss is lower than the training loss no matter how many training iterations you perform. In this case, it suggests that the validation dataset is easier for the model to predict than the training dataset. Common causes for this situation are

- Information leakage where a feature in the training data has direct ties to observations and responses in the validation data
- Poor sampling procedures where duplicate observations exist in the datasets
- Validation dataset contains features with less variance that the training dataset

Solutions to this problem involve removing duplicate observations, taking steps to reduce information leakage between training and validation datasets, making sure that you are randomly sampling observations so that feature variance is consistent, and performing cross-validation.

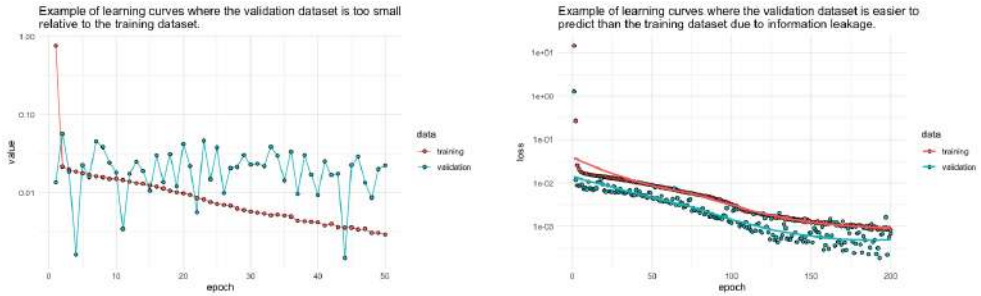


FIGURE 17. (left) validation dataset is too small relative to training data (right) val loss less training loss suggests information leakage between the two datasets.

CHAPTER 5

Convolutional Neural Networks

Convolutional neural networks or CNNs are critical to the success of computer vision applications such as self-driving cars and facial recognition systems. They are a special kind of deep neural network whose densely interconnected layers are replaced by more specialized interconnections that can be viewed as realizing a discrete convolution.

Convolutional neural networks were first introduced by Fukushima in 1983 under the name of *neocognitron* [**FMI83**]. Its architecture was inspired by a hierarchical model of the nervous system proposed by Hubel and Weisel after their early study of the cat's visual cortex [**HW62**]. In 1989, Yann LeCun used backpropagation along with neocognitron concepts to propose a model called *LeNet* which was used for handwritten zip code recognition by the U.S. Postal Service[**LBD+89**]. After this initial success there was a long lull in neural network research activity until 2012 when Hinton and others at the University of Toronto entered a CNN in the famous ImageNet challenge and ended up winning that contest[**KSH12**]. Another major achievement occurred in 2015 when a CNN neural network called *ResNet* surpassed human level error rate of 5.1% with an error rate of 3.57%[**HZRS16**].

In chapter 4, we used the handwritten digit classification problem with the MNIST database to illustrate a typical workflow used in developing a sequential neural network model for the application. In this case, we were able to obtain a final model that consisted of a single hidden layer with 64 nodes. Our validation testing for that model achieved a classification

accuracy of 97%. But recall that in section 4 of chapter 1 we trained another model with a 2D convolutional architecture to obtain a classification accuracy on the testing data of 93.3%. With some slight tweaks of that architecture, we will be able to greatly increase the model's accuracy to nearly 99%. This chapter introduces the convolutional neural network architecture and presents examples of its use in several computer vision applications.

1. MNIST Problem Revisited

This section revisits the MNIST problem presented earlier using a variation of the 2D convolutional neural network (CNN) architecture from section 4 of chapter 1. We will compare it directly against a sequential model with a single hidden layer of 64 nodes with L2 regularization kernel. We will train both models on the MNIST dataset, using 30,000 training samples and 10,000 testing samples. The training dataset will be batched with a batch size of 256 and will have a 30% validation split. The 2D convolution network model is instantiated and compiled using the following script

```
 inputs = keras.Input((28,28,1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_cnn = keras.Model(inputs=inputs, outputs=outputs)

model_cnn.summary()

import tensorflow as tf
model_cnn.compile(
     optimizer=tf.keras.optimizers.Adam(learning_rate=1.e-3),
     loss = "sparse_categorical_crossentropy",
     metrics = ["accuracy"])
```

The model summary shows the output shape of each layer and the number of trainable parameters (weights). This model has over 100,000 weights to train. Note that the output shapes of each layer in the CNN is a rank-3 tensor. The CNN inputs have shape $(28, 28, 1)$ but the output of the first convolutional layer has shape $(26, 26, 32)$. Recall that tensors are multi-dimensional matrices. For CNN's, the first two components of the tensor are the spatial dimensions of the image. The third component carries additional feature information. Since each pixel in the MNIST input image is a floating point number, the third dimension of the input tensor has only 1 component. The combined convolutional layer and max-pooling layer, however, do two things. First they downsample the size of the original image from 28 by 28 to 13 by 13 and then they expand the third dimension from 1 to 32. These 32 additional channels represent *features*. During the second wave of neural network research in the 1980's, these "features" were pre-designed by the ML engineer. What is new about our CNN is that it is learning those features by itself.

We now reshape our data set images to fit into these tensor shapes. The following script also creates dataset objects from the train and testing data. We then split the training dataset object into a p-training and validation dataset.

```
from tensorflow.keras.datasets import mnist

db_size = 30000
(train_x, train_y), (test_x,test_y) = mnist.load_data()
train_x = train_x.reshape((db_size, 28,28,1)).astype("float32")/255
test_x  = test_x.reshape((10000, 28,28,1)).astype("float32")/255

batch_size = 256
train_ds = tf.data.Dataset.from_tensor_slices((train_x, train_y))
train_ds = train_ds.batch(batch_size)
train_ds_size = len(list(train_ds))

test_ds  = tf.data.Dataset.from_tensor_slices((test_x,test_y))
test_ds  = test_ds.batch(batch_size)
test_ds_size = len(list(test_ds))
```

```
val_split = 0.30
train_ds_size = len(list(train_ds))
val_size = int(val_split*train_ds_size)
ptrain_size = train_ds_size-val_size
ptrain_ds = train_ds.take(ptrain_size)
val_ds = train_ds.skip(ptrain_size).take(val_size)
```

We then fit and save the best CNN with the lowest validation loss. The accuracy of this "best" CNN will then be evaluated on the test dataset. Fig. 1 plots the p-training and validation losses for the sequential and CNN model. The right sided plot shows the validation accuracy of both models as a function of training epoch. The titles in these plot show that the test accuracy of the best sequential model was 93.6% whereas the best CNN's test accuracy was 98.5%. This shows that clearly the CNN architecture performs better on the MNIST digit recognition task. The following sections examine the CNN in more detail.

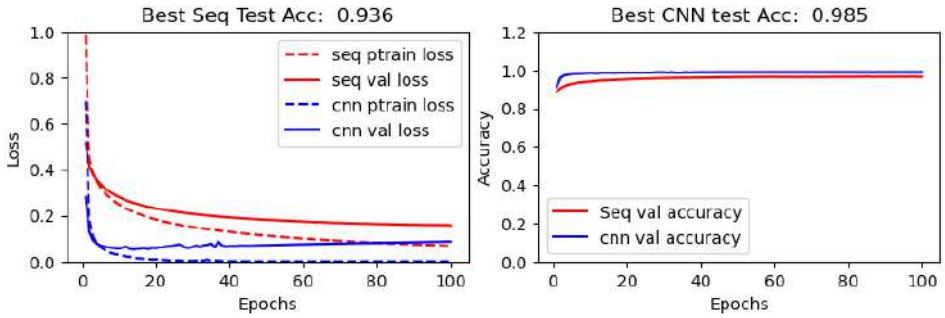


FIGURE 1. Training curves comparing a sequential neural network and a 2D CNN's loss and accuracy as a function of training epoch.

2. Computer Vision Applications

Computer vision applications were one of the first major success stories for deep learning. This success rests on the fact that the information in an image in inherently *local* and *translation invariant*. "Local" means that pixels that

are close to other are more likely to represent the same object, than pixels that are at opposite ends of the image. "Translation Invariant" means that a pattern of pixels representing a certain feature will carry the same meaning no matter where it is located in the image. These two aspects of computer vision allow one to employ neural network architectures that have fewer weights than Sequential dense models achieving similar levels of accuracy. In our MNIST application, for example, a single layer dense model could be trained to achieve a similar (98%) level of accuracy as the 2D convolution model if it had 2048 nodes and training data set of 60000 samples (rather than 30000). This sequential model would have about 1,600,000 weights whereas the 2D convolutional model only had 100,000 trainable weights. The layer that makes this possible is called a *convolutional layer*. To help motivate this layer class, I will first review how image classification has been traditionally performed in computer vision systems prior to the advent of deep learning.

Let us consider the problem of tracking a ballistic missile in flight during its boost phase using a hyper-kinetic projectile that has a vision sensor on its nose. We want to develop algorithms that can use the image created by the sensor to identify the point on the missile we wish to hit with the projectile. The problem has several vision tasks we need to solve. We first must *detect* the target in the image and then we must segment out or localize the target in that image. The target's image then has to be split into two parts; the missile body and engine plume. The missile's engine will be designated as the projectile's target and its coordinate passed to the projectile's guidance system. Note that this process has to be done every few tenths of a second because as the projectile flies toward the target, its image grows in the sensor's field of view.

The algorithms we can use for this problem involve converting the RGB image from the sensor into a greyscale image and then using a Sobel edge filter to create an outline of the missile and its plume. We then use the image morphological transformations of dilation, fill, and erosion to recover the

brightest spot in the image. That bright spot should be the engine and the segmented image shown in the final picture of Fig. 2 would be the target whose coordinates we would pass to the projectile's guidance system.

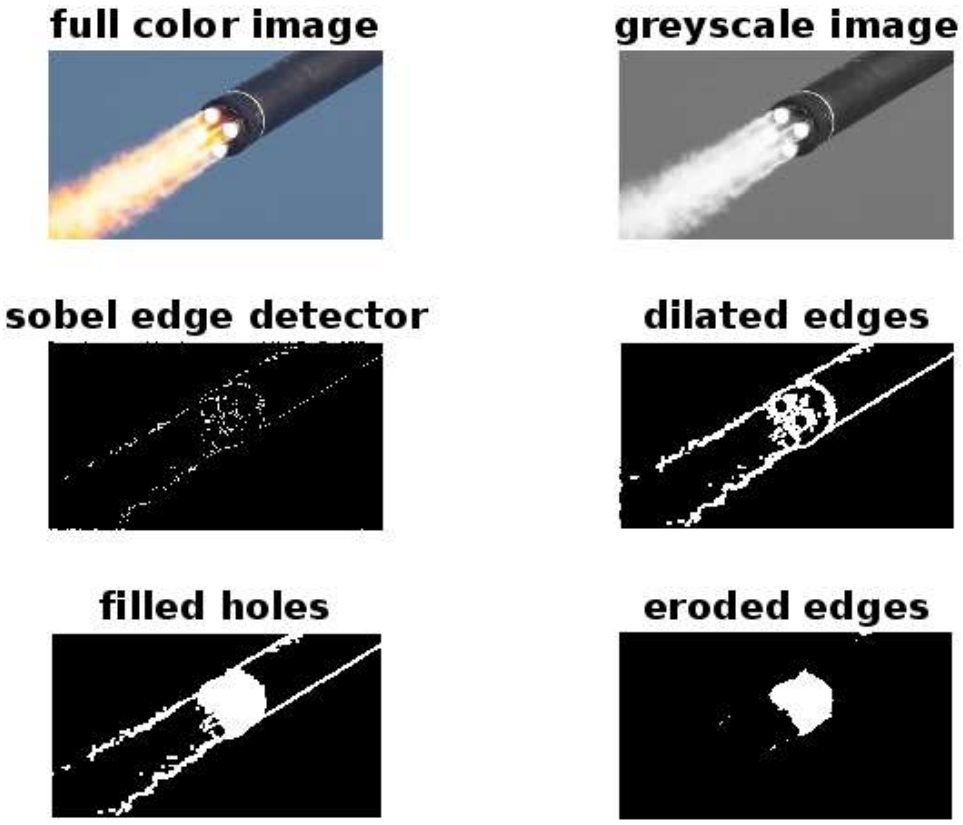


FIGURE 2. Image Processing Tasks used to Identify Missile Engine in Tracking Problem

All of the image transformations shown in Fig. 2 (sobel edge detection, morphological dilation, fill, and erosion) may be seen as taking a *structuring kernel* and applying it to the neighborhood of each pixel in the image. Spatial filtering using the structural kernel is probably the easiest way to illustrate what we mean. In particular, if we let $I_{k,\ell}$ denote the input image's k,ℓth pixel, and let $K_{i,j}$ denote the i,jth element of the structuring kernel (i,j ranges from $-M$ to M), then the filter output's k,ℓth pixel, $R_{k,\ell}$

is computing as

$$R_{k,\ell} = \sum_{i=-M}^{M} \sum_{j=-M}^{M} K_{i,j} I_{k+i,\ell+j} \stackrel{\text{def}}{=} K * I.$$

This is sometimes called a *convolution sum*, though in reality it is actually a "correlation" sum. We denote the convolution as the binary operation $*$. Fig. 3 illustrates how this convolution is realized on a 9 by 9 image, I, which shows a dark square in the center of the image. The kernel is shown to the left of the figure has size 3 so that i, j ranges from -1 to 1. The filtered image resulting from the convolution is shown on the right.

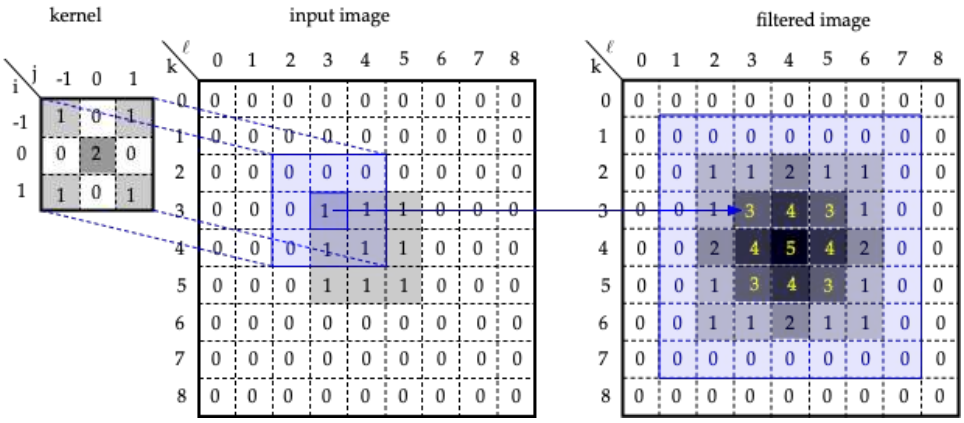


FIGURE 3. Image Convolution/Correlation

Let us look at the sum for the element $(k, \ell) = (0, 1)$. In this case the sum is

$$\begin{aligned} R_{0,1} &= \sum_{i=-1}^{1} \sum_{j=-1}^{1} K_{i,j} I_{i,1+j} \\ &= K_{-1,-1} I_{-1,0} + K_{-1,0} I_{-1,1} + K_{-1,1} I_{-1,2} \\ &\quad + K_{0,-1} I_{0,0} + K_{0,0} I_{0,1} + K_{0,1} I_{0,2} \\ &\quad + K_{1,-1} I_{1,0} + K_{1,0} I_{1,1} + K_{1,1} I_{1,2} \\ &= (1 \times ?) + (0 \times ?) + (1 \times ?) \\ &\quad + (0 \times 0) + (2 \times 0) + (0 \times 0) \\ &\quad + (1 \times 0) + (0 \times 1) + (1 \times 1). \end{aligned}$$

Note that several of the terms in the sum are unknown (?). The reason for this is that the input image pixel $(-1, 1)$ doesn't exist. There are two ways we can handle this. One way is to compute no output for the input pixel $(0, 1)$. If we applied this to all of the pixels, we would end up with an image whose size would by 7 by 7, rather than 9 by 9. This smaller output image is shown by the blue square in Fig. 3. The other way of handling this is to treat the unknown pixel values as 0. This is known as zero padding. The resulting output would be the full image shown on the right of Fig. 3, a 9 by 9 image.

Let us also compute another output value, this time for input pixel, $(3, 3)$. In this case the sum is

$$\begin{aligned} R_{3,3} &= \sum_{i=-1}^{1}\sum_{j=-1}^{1} K_{i,j} I_{3+i,3+j} \\ &= K_{-1,-1}I_{2,2} + K_{-1,0}I_{2,3} + K_{-1,1}I_{2,4} \\ &\quad + K_{0,-1}I_{3,2} + K_{0,0}I_{3,3} + K_{0,1}I_{3,4} \\ &\quad + K_{1,-1}I_{4,2} + K_{1,0}I_{4,3} + K_{1,1}I_{4,4} \\ &= (1 \times 0) + (0 \times 0) + (1 \times 0) \\ &\quad + (0 \times 0) + (2 \times 1) + (0 \times 1) \\ &\quad + (1 \times 0) + (0 \times 1) + (1 \times 1) \\ &= 3. \end{aligned}$$

The output matches what we have on the right hand image for pixel $(3, 3)$ on the output image. We applied this to the rest of the pixels in the input image to get the output distribution shown on the right side of Fig. 3. From this we can see that the action of this particular filter kernel was to "blur" the original square in the input.

The various computer vision operations of edge detection, morphological dilation, fill, and erosion [**Ser82**, **S+99**] can all be viewed as various types of convolutions. The difference being in the "structuring kernel" and in the algebraic operations used to apply that kernel to the image. The Sobel

edge filter uses two different kernels of the form

$$K_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Applying each kernel to the input image, I, produces output images $R_x = K_x * I$ and $R_y = K_y * I$. The final output of the Sobel edge detector is obtained by combining the resulting images so that the i, j pixel of the output is the root mean square of the i, jth pixels in R_x and R_y.

The image morphology transformations [**Ser82**] may also be viewed as "convolution" in that the structuring kernel, K, is also moved across the input image, I, as shown in Fig. 3. The difference is that instead of using multiplication and addition, we use nonlinear set operations on the pixels after they have been thresholded to be either 0 or 1. The structuring kernel K is also binary valued and the output is computed based on a set operation such as the intersection or union of the thresholded image and the kernel.

What should be apparent in both cases is that low-level image processing tends to be based on convolution-like operations using rather small kernels. In addition to this we see that the kernel is *translation invariant* in the sense that we use the same kernel on all pixels of the input image. This is precisely what convolutional neural networks attempt to copy. The other thing to note is that approach to image processing is also similar to how the brain processes imagery to recognize visual objects. This was discovered in early studies of the feline visual cortex. As shown in Fig. 4, the brain processes an image presented by the retina through a sequence of different regions in the brain that extract "features" from the input image and combines those features to interpret what the image is. This layered design is how the CNN we used in the last lecture was built.

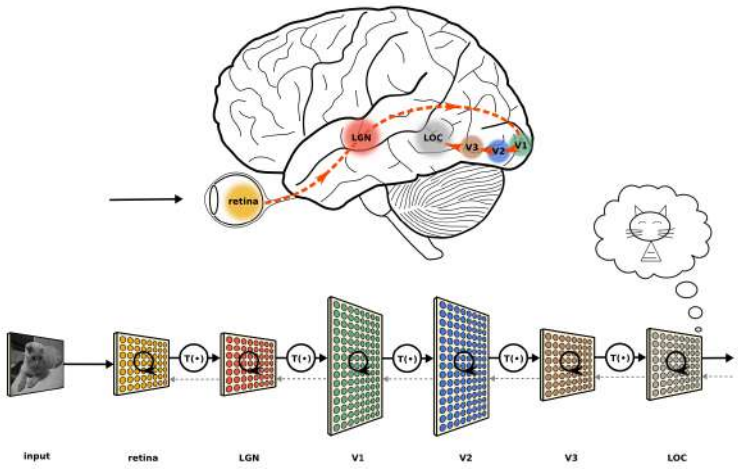


FIGURE 4. For some types of tasks (e.g. for images presented briefly and out of context), it is thought that visual processing in the brain is hierarchical: one layer feeds into the next, computing progressively more complex features. This is the inspiration for the *layered* design of modern feed-forward neural networks.

3. Convolutional Neural Networks

CNN's take advantage of the fact that the input is an image whose features are translation invariant (the same no matter where they appear in the image). The CNN model architecture is organized to take advantage of this translation invariance. To see what is new in the CNN, let us compare it to a conventional dense sequential neural network. A dense sequential neural network has a sequence of layers in which all nodes of one layer are connected to all nodes of the next layer. There is no special geometry to how the nodes are arranged so we can view them as shown on the left of Fig. 5 where the nodes in each layer are just arranged as a linear array.

The layers of a CNN, on the other hand, do have a special geometric arrangement. In particular, the nodes in a CNN have a three dimensional

spatial arrangement; width, height, and depth as shown on the right side of Fig. 5. The width (x) and height (y) are referred to as *spatial* dimensions, whereas the remaining dimension is referred to as the *channel* dimension. The other feature of CNN's used in classification is that the layer shapes are arranged so that the input has large spatial dimensions and few channels. In each subsequent layer, the spatial dimensions decrease and the channel dimension increases, until the CNN's output can have a shape dominated by the channel dimension. In practice, the "channels" can be thought of as "image features" that were obtained by "squeezing" the information out of the spatial dimensions. This general principle of reducing one set of input dimensions and growing the "feature" dimensions is a common principle in deep learning.

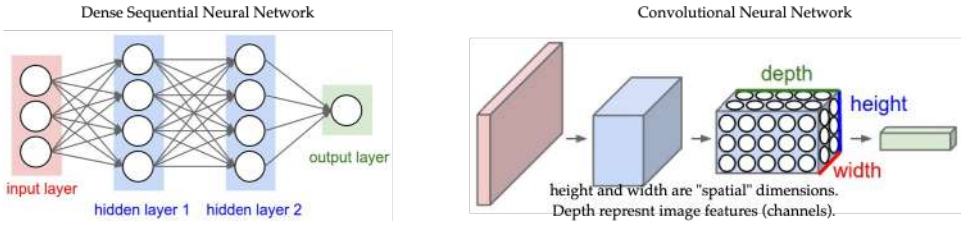


FIGURE 5. The layers of dense sequential networks have no special geometry. CNNs used for image classification have an architecture that "squeezes" (downsamples) the input's spatial dimensions and places that information in "channels" representing image features.

So if the input tensor is rank-3, rather than rank-2, then how is the convolution layer's computed? In fact it is very similar to what we described above. In this case, the image is a rank-3 tensor with shape (N, N, M) and whose i, j, kth element is denoted as $I_{i,j,k}$. We let i, j be the spatial dimensions and k be the input channels. Let the filter kernel have a kernel width of $2M_K + 1$ where M_K is a positive integer and a depth of M. If the output has a shape of $(N, N, 1)$ (assuming zero padding of the spatial dimensions,

then the output of the $i, j, 0$ element in the output would be

$$R_{i,j,0} = \sum_{k=-M_K}^{M_K} \sum_{\ell=-M_K}^{M_K} \sum_{m=0}^{M-1} K_{k,\ell,m} I_{i+k,\ell+j,m}.$$

We can therefore see that the convolution sum is only computed along the spatial dimensions. The filter kernel is also a rank-3 tensor with an odd width and height. The number of channels in the kernel is equal to the number of channels in the input.

We computed this output assuming a single output channel, but as mentioned above, CNN layers tend to increase the number of depth channels as we move deeper into the network. So in general, our output shape should be (N, N, L) where L is the number of depth channels that we would specify when defining the model's architecture. In this case, the filter kernel would then be a tensor of shape $(2M_K + 1, 2M_K + 1, L \times M)$. In other words, we would simply have L additional filter kernels and then stack them up into a single kernel. The output would then be computed as

$$R_{i,j,\ell} = \sum_{k=-M_K}^{M_K} \sum_{\ell=-M_K}^{M_K} \sum_{m=0}^{M-1} K_{k,\ell,\ell*L+m} I_{i+k,\ell+j,m}.$$

This computation is a bit easier to visualize in Fig. 6. The left side shows the input tensor with a shape $(7, 7, 3)$ and an input kernel of shape $(3, 3, 3 \times 128)$. The output shape is obtained assuming no zero-padding, so the spatial dimensions are 2 smaller and the depth channel on the output is equal to 128.

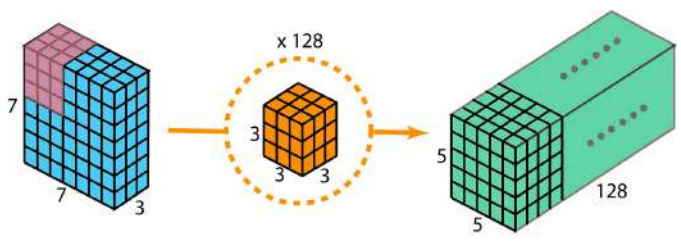


FIGURE 6. Convolutional operation on a rank-3 input tensor

Note that in this implementation of the Conv2D layer, the spatial dimensions were only reduced by 2 if we assumed no zero-padding. If we had used zero-padding then the spatial dimensions would be unchanged. But as mentioned above, CNN layers also tend to downsample or reduce the number of spatial channels. If you look back at the CNN example from the last lecture that reduction was by a factor of two. So clearly CNN models must also be doing something else to reduce the spatial dimension by so much. This is done by either specifying a *stride* to the convolution or by adding a 2D Max Pooling layer after a convolutional layer without strides. Early models used the max pooling layer. More recent models tend to use strides as they do not destroy spatial information in the input tensor as much as the max pooling layers do.

The purpose of the max-pooling layer is to reduce the spatial size of the input tensor. The pooling layer acts independently on every channel slice of the input. Pooling is realized by a passing a filter kernel over the input's spatial dimensions. The filter kernel has size $M \times M$ and it outputs the maximum value of the inputs in the kernel window. Rather than applying the kernel to every pixel, we only apply it to the pixels that are a multiple of M, thereby reducing the spatial dimensions of the output by a factor of M. The most commonly used max pooling layer with a pool size of 2 so that the filter kernels are 2×2 and are applied with a stride of 2. These layers, therefore, downsample the spatial dimensions of the input tensor by a factor of 2. We can represent this mathematically as follows. Consider an input of shape $(2N, 2N, M)$ and let us use a max-pooling layer with a pool size of 2. The (i, j)th output of the max pooling layer (ignoring the depth dimension) is

$$R_{i,j} = \max \left\{ I_{2i,2j}, I_{2i-1,2j}, I_{2i-1,2j-1}, I_{2i,2j-1} \right\}.$$

Fig. 7 illustrates how this layer works on an input tensor of size $(9, 9)$. A pool size of 2 does not divide evenly into 9, so the we simply end up with an output with the shape of $(4, 4)$. Note that the "square" shape from the input is preserved, it is has a smaller spatial extent by a factor of 2.

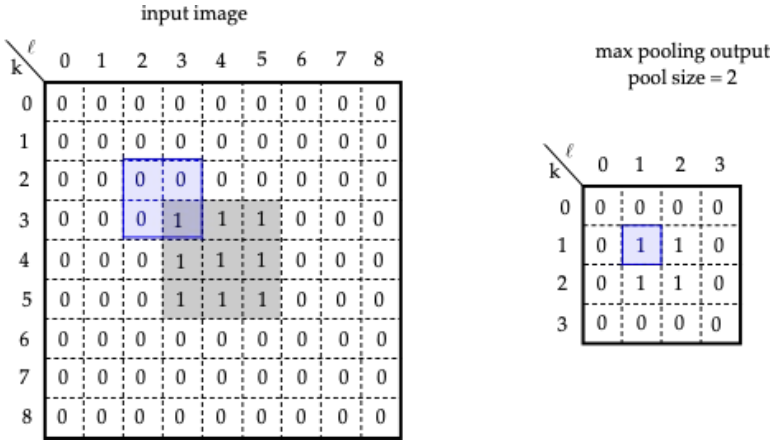


FIGURE 7. Action of Max Pooling Layer

The second way of downsampling the spatial dimensions of the input tensor is to simply compute the convolution with a stride of S. Given an image I with shape (N, N), the convolution sum is only computed on input pixels $(i + kS, j + kS)$ for where i and j range from 0 to $N - 1$. This is easy to incorporate into the Conv2D layer so it is realized directly in the Conv2D layer's API. Fig. 8 shows how the output that would have been generated assuming a stride of 2 with our earlier convolution. We assume zero-padding here. In this case the shape of the output image is 5 by 5 due to the zero-padding. The figure shows the progression of kernels computing the filtered output along the diagonal of the output image.

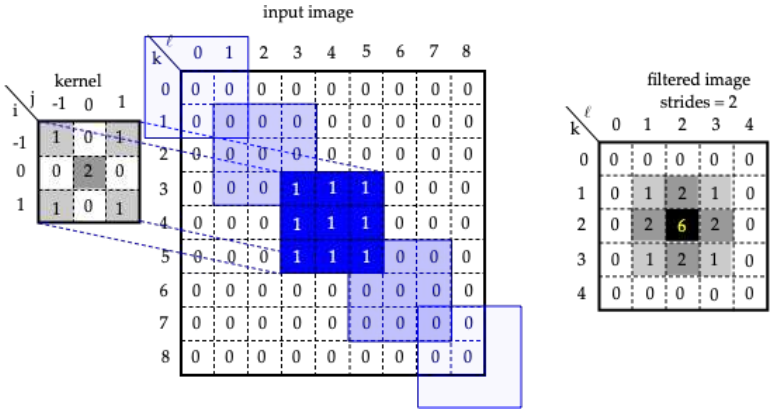


FIGURE 8. Conv2D with stride = 2

We can now go back to the earlier example where we used a convolutional neural network architecture for MNIST classification. The script building that model is shown below. This example uses max pooling layers to realize the downsampling of the spatial dimensions.

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(28,28,1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10,activation="softmax")(x)
model = keras.Model(inputs=inputs,outputs=outputs)
```

We can now use what we discussed above to determine the shape of each layer's output. The output from the Input layer is $(28, 28, 1)$. The first convolutional layer creates 32 filter channels along the depth dimension and has a kernel size of 3. Since we have not specified zero-padding, the spatial dimensions of this layer's outputs will be smaller by 2. The output tensor from this first Conv2D layer will therefore be $(26, 26, 32)$. The Max Pooling layer follows the Conv2D layer and therefore its output will downsample the input shape's spatial dimensions by a factor of 2. There is no change to the number of channels. So the output shape from the first max pooling layer will be $(13, 13, 32)$. In a similar way we can argue that the shape of the output from the second convolutional layer will be $(11, 11, 64)$ and the output from the second max pooling layer will be $(5, 5, 64)$. We now pass this last $(5, 5, 64)$ tensor to a Conv2D layer with 128 filters, a kernel size of 3, and no zero padding. So the final convolutional layer's output tensor's shape will be $(3, 3, 128)$. This is flattened to produce a vector output that has a length $3 \times 3 \times 128 = 1152$. The last dense layer takes this and maps it onto 10 outputs. This checks against the shapes stated in the model summary.

4. Image Classification Task - with limited data

The image classification task seeks to take an image and classifies it into one of finite number of categories. The MNIST handwritten digit recognition task is just one example of an image classification task. This section develops a similar application whose dataset consists of larger images. We will refer to this as the Oxford Pets dataset and it consists of a large number of color (RGB) images of cats and dogs like those shown in Fig. 9. This particular example is drawn from the textbook [**Cho21**]. We will use this example to show how we train models when there is a limited amount of data. The objective is to train a model that can declare whether the input image is a picture of a cat or dog.

FIGURE 9. Sample Images from the Cat/Dog Data set

In this example we download the dataset from the visual geometry group's website at the University of Oxford and place them in a subdirectory (`cats-vs-dogs-small`) of the main working directory. This subdirectory has three subdirectories entitled `train`, `validation`, and `test` containing p-training, validation, and testing data, respectively. Each subdirectory has two subdirectories entitled `cat` and `dog`. The p-training data set has 2000 examples of cats/dogs (1000 each). The testing set has the same number of examples. The validation dataset has 1000 examples of cats/dogs (500 each). Compared to the MNIST dataset, it is clear our Cats/Dog dataset is much smaller than the MNIST database and this will limit the performance of the trained models.

The following script uses one of Keras' utilities to create `dataset` objects for the data. These objects batch the data into batches of size 32. We use this utility to automatically convert the RGB image files into rank-3 tensors. Each image tensor has shape $(180, 180, 3)$ where the first two dimensions are the spatial dimensions and the third dimension contains the color data for the given pixel. Note that this is much larger than the earlier MNIST example whose images had shape $(28, 28, 1)$.

```
from tensorflow.keras.utils import image_dataset_from_directory
import os, shutil, pathlib

base_dir = pathlib.Path("cats-vs-dogs-small")
train_ds = image_dataset_from_directory(
       base_dir/"train", image_size = (180,180),
       batch_size = 32)

test_ds = image_dataset_from_directory(
       base_dir/"test", image_size = (180,180),
       batch_size = 32)

val_ds = image_dataset_from_directory(
       base_dir/"validation", image_size = (180,180),
       batch_size = 32)
```

With the datasets batched and ready, we can now declare a CNN model. Because our inputs have a larger shape of $(180, 180, 3)$, we will use a deeper network. We will use max pooling to downsample the input image.

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(180,180,3))
x = layers.Rescaling(1./255)(inputs)

for i in range(4):
     x = layers.Conv2D(filters=32*(i+1),
            kernel_size = 3, activation="relu")(x)
     x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=256, kernel_size=3,
            activation="relu")(x)
x = layers.Flatten()(x)
```

```
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs,outputs=outputs)
model.summary()
```

We now compile and fit the model. Since this a binary classification problem (cat or dog) we use a binary cross-entropy loss function. We will train with the optimizer RMSprop using the default learning rate parameter (0.001) with "classification accuracy" as the metric. We will train the model for 30 epochs and save the model that has the best validation loss during training.

```
model.compile(loss = "binary_crossentropy",
         optimizer="rmsprop",metrics = ["accuracy"])

callbacks = [
      keras.callbacks.ModelCheckpoint(
               filepath = "best_model.keras",
               save_best_only = True,
               monitor = "val_loss")]

num_epochs = 30
history = model.fit(train_ds, epochs = num_epochs,
             validation_data = val_ds,
              callbacks = callbacks)
```

Once training is complete, we recover the stored "best" model and compute its accuracy on the test dataset (`test_ds`) and plot its training curves.

```
test_model = keras.models.load_model("best_model.keras")
test_loss, test_acc = test_model.evaluate(test_ds)
print(f"Test Accuracy: {test_acc: .3f}")

import matplotlib.pyplot as plt
ptrain_loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1,num_epochs+1)

plt.plot(epochs, ptrain_loss, "b--", label="ptrain_loss")
plt.plot(epochs, val_loss, "b", label="val_loss")
plt.title(f"Cat/Dog Learning Curves (loss) - Test Acc: {test_acc: .3f}")
plt.legend()
```

The training curves for this model are shown in Fig. 10. From this curve we can see that the model begins overfitting almost immediately. As discussed in chapter 4, this can be due to the dataset being too small or the model set having too much complexity for the given data. We already know, however, that there are only 2000 samples in the p-training data, and since prior experience with MNIST suggests we need 10's of thousands of samples for a smaller simpler image of shape (28,28,1), the fact that our Cat/Dog images are much larger with shape (180,180,3) suggests that perhaps the problem is that the dataset is too small. We therefore need to find a way of addressing the issue of training models when there is limited data.

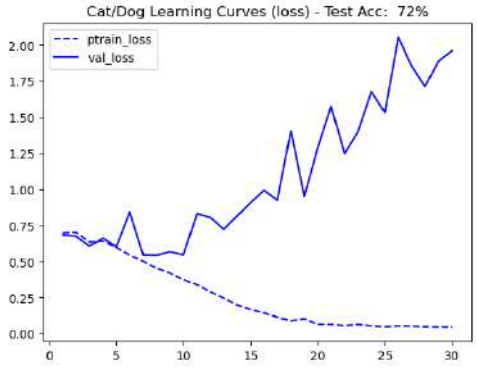


FIGURE 10. Training Curves for First Attempt to Train a Model for the Cat/Dog Classification Problem - too few p-training samples

4.1. Data Augmentation - training with limited data: One approach to train models with limited data sets can be taken when the inputs are images. In particular, we can *augment* the original dataset inputs by taking the images and applying random transformations that create believable looking images.

TensorFlow/Keras accomplishes this augmentation by adding a *data augmentation layer* at the start of the model. This layer takes the input image

and transforms it through a sequence of randomly selected base transformations. These base transformations involve ”flipping” the image across a chosen axis, rotating the image by a random amount, or zooming the image by a specified factor. The following script creates a new `data_augumentation` layer by sequentially chaining together these base transformation layers. Examples of these transformations are shown in Fig. 11.

```
import tensorflow
from tensorflow import keras
from tensorflow.keras import layers

#this was done on an M1 and uses the CPU
with tensorflow.device("/CPU:0"):
    data_augmentation = keras.Sequential([
       layers.RandomFlip("horizontal"),
       layers.RandomRotation(0.1),
       layers.RandomZoom(0.2),
    ])

import matplotlib.pyplot as plt
plt.figure(figsize=(5,5))
for images, _ in train_ds.take(2):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3,3,i+1)
        plt.imshow(augmented_images[0].numpy().astype('uint8'))
        plt.axis("off")
```

If we train a new model that uses this data augmentation layer, then the model never sees the same input twice. But the inputs it does see will still be correlated because they came from a small number of original images. The new information created in the transformed images is, therefore, of limited value because we are only remixing existing information. As a result, this may still not be enough to combat overfitting. So to further fight off overfitting we add some additional regularization. In this case we add dropout regularization. Note that data augmentation and dropout are only active during training. So in each epoch when we evaluate the model's loss and

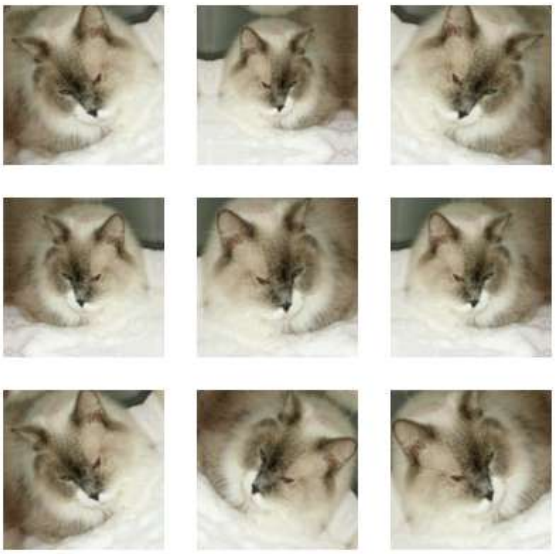


FIGURE 11. Examples of Images transformed by the augmentation layer

accuracy on the validation data, the dropout layer and data augmentation layer are deactivated by the `fit` method.

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(180,180,3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)

for i in range(4):
     x = layers.Conv2D(filters=32*(i+1),
            kernel_size = 3, activation="relu")(x)
     x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=256, kernel_size=3,
            activation="relu")(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)

outputs = layers.Dense(1, activation="sigmoid")(x)
model_aug = keras.Model(inputs=inputs,outputs=outputs)
model_aug.summary()
```

The training curves for this model with and without data augmentation are shown in Fig. 12. The red curves are for the original model trained without data augmentation and the blue curves are for the model trained using data augmentation with dropout. We see that the augmented model's validation loss appears to control the overfitting seen in the red curves. This validation curve is noisy because we are still evaluating validation loss on a small validation set without augmentation. Finally, if we compare the two models' accuracy on the testing data, we see a significant difference. The original model's test accuracy was 72%, whereas the augmented model's accuracy was 80%. Clearly data augmentation increased the accuracy of the trained model, but as mentioned before that increase is limited because we are simply remixing the information in the original smaller dataset.

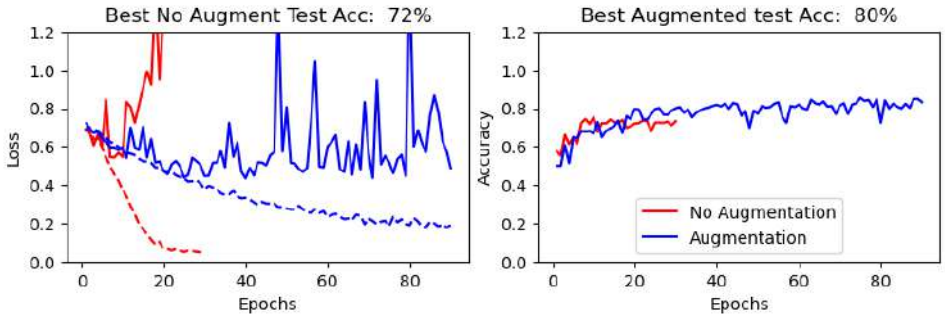


FIGURE 12. Training Curves for Second Attempt to Train a Model for the Cat/Dog Classification Problem - Data Augmentation

4.2. Transfer Learning - training with limited data: Transfer learning represents another approach used to train a model when there is not enough data. This approach attempts to *transfer* the knowledge contained in one model to another model. The underlying conjecture in this approach is that the activations in the hidden layers are triggered by features in the input that are common in many different vision applications. So if one takes a model that was extensively trained on an extremely large dataset, then one can take the first few layers of that pre-trained model and then retrain

their features to be used on a different application. In particular, this means we take the ”base” part of the pretrained model and attach a smaller dense network on top of it. That dense network is called the model’s *head*. We then freeze the weights of the base and only allow the weights of the head to be retrained by the new data for the new application. Because the head has fewer weights than the base, this model can be trained with a smaller dataset.

The justification for using transfer learning in vision applications comes from the belief that the activations of hidden layers in a neural networks appear to be structured to represent fundamental features of the input image. This is shown in Fig. 13 which looks at the ”images” that activate various neurons at different layers of a CNN. The lowest level neurons appear to be activated by features that are similar to our Sobel edge detector. The mid-level layers are activated by more complex shapes within the image. The high level layers appear to be activated by actual features within the input image. The idea in transfer learning is that these high level features are abstractions common to all vision tasks and that all we need to do to use them is learn how to combine them for a different classification task.

As an example of this approach, we can take a large CNN that was pre-trained on the ImageNet [**RDS+15**] dataset (1.5 million labeled images and 1000 different classes). ImageNet contains many animal classes including different species of cats and dogs. We will use a prior CNN model known as VGG16 [**SZ14**] that was pre-trained on Imagenet as our pre-trained convolutional base. This VGG16 network is a much larger version of the CNN we used above and its layers are shown in Fig. 14. In this example we will take VGG16 as the convolutional base and then train a new classifier on top of it using our small Cat/Dog dataset.

The VGG16 model is included in Keras. The following script recovers that model trained on ImageNet. It leaves off the ”top” layer of VGG16, so

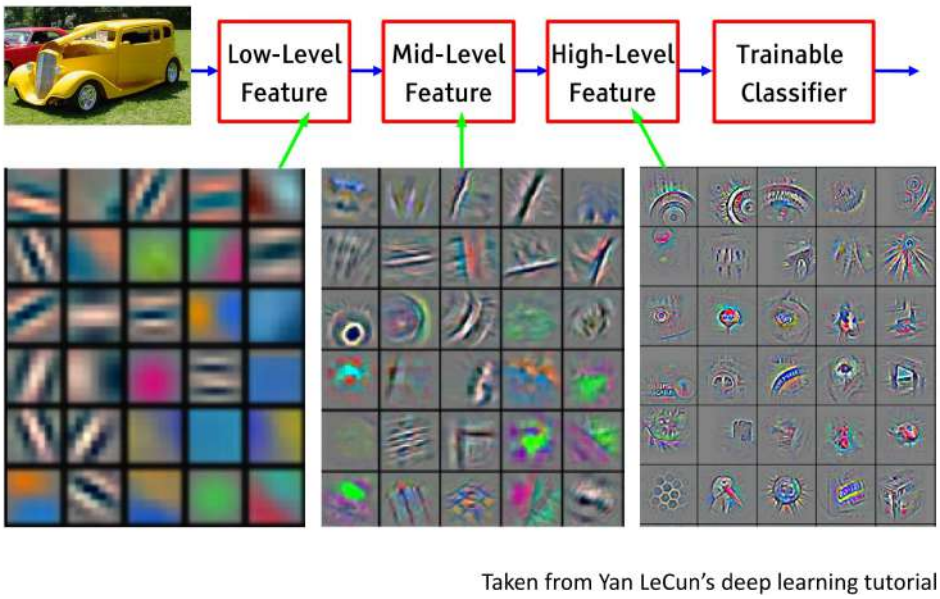


FIGURE 13. CNN learns "features" of the input image

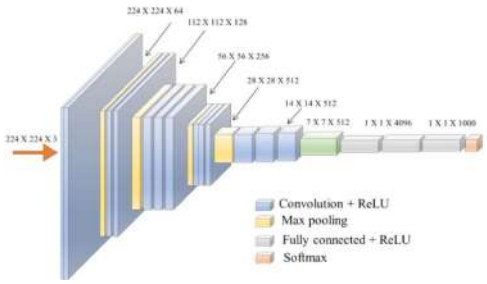


FIGURE 14. VGG16 Model Architecture

we are only recovering the base and we are freezing those weights so they cannot be changed by subsequent training.

```
import tensorflow
from tensorflow import keras
from tensorflow.keras import layers

conv_base = keras.applications.vgg16.VGG16(
         weights = "imagenet",
         include_top = False,
         input_shape = (180,180,3))
conv_base.trainable = False
```

```
conv_base.summary()
```

From the model summary, we see that the top layer's output is a tensor with shape (5,5,512). So we have downsampled the original 180 by 180 image to a 5 by 5 image and for each "pixel" in that downsampled image we have 512 channels. Each channel may be thought of as a feature. The basic idea in transfer learning is that the base has been trained on a large enough dataset so that many of the features specific to our given problem can be captured by combining a smaller set of the features in the top layer of the base model. To train a new model for our new dataset, we would simply extend the "frozen" convolutional base by adding a dense layer on top of it and retrain these new top layers only on the new data. The following script builds this new model using the frozen convolutional base. Note that just as we did with the earlier 2D convolutional model, we use data augmentation and dropout to increase the size of the dataset.

```
import tensorflow
from tensorflow import keras
from tensorflow.keras import layers

with tensorflow.device("/CPU:0"):
    data_augmentation = keras.Sequential([
       layers.RandomFlip("horizontal"),
       layers.RandomRotation(0.1),
       layers.RandomZoom(0.2),
    ])

from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(180,180,3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model_transfer = keras.Model(inputs,outputs)
model_transfer.summary()
```

From the summary for `model_transfer` we have nearly 18 million weights. But most of those weights are in the frozen convolutional basis. In particular, only 3.27 million of these weights are trainable, so the training of this model is much faster and can be done with a smaller dataset than would be required if we were training the entire VGG16 model from scratch. So we again compile and fit the model for 90 epochs. Fig. 15 shows the training curves for training a new head on top of the pre-trained VGG16 model.

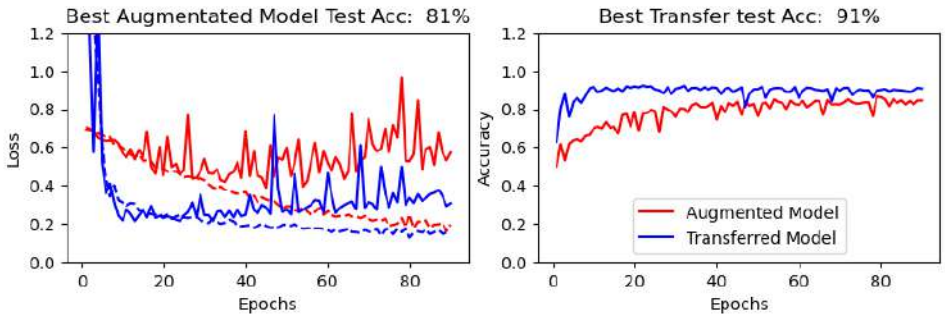


FIGURE 15. Training Curves for Third Attempt to Train a Model for the Cat/Dog Classification Problem - Transfer Learning from ImageNet using VGG16

Fig. 15 compares the training curves for the earlier convolutional model with augmentation and the model we transferred from a frozen ImageNet VGG16 base. The blue curves are for the transferred model and the red curves are for the original model. We see that the transferred model has a much lower loss function and that it trains much faster than the original model. This is to be expected because 1) the transferred model is deeper and was pre-trained with the larger ImageNet dataset and 2) we only used the Oxford Cat/Dog dataset to retrain the upper layer of the pretrained model. As a result the transferred model's best test accuracy was significant better than that of the best original model; 91% versus 81%.

5. Image Segmentation Task - U-net Architecture

Real-life computer vision tasks involve classifying and isolating components of that image that drive other high level cognitive functions. This gives rise to three essential vision tasks illustrated in Fig. 16

- *Image classification* is a task whose goal is to assign one or more labels to the entire image.
- *Image segmentation* is a task that seeks to segment or partition the image into different areas with each area representing a category.
- *Object detection* is a task that draws rectangles (also called *bounding boxes*) around objects of interest in an image and then associates each rectangle with a class.

We discussed image classification in the preceding section. This section confines its attention to image segmentation. The next section takes a quick look at object detection.

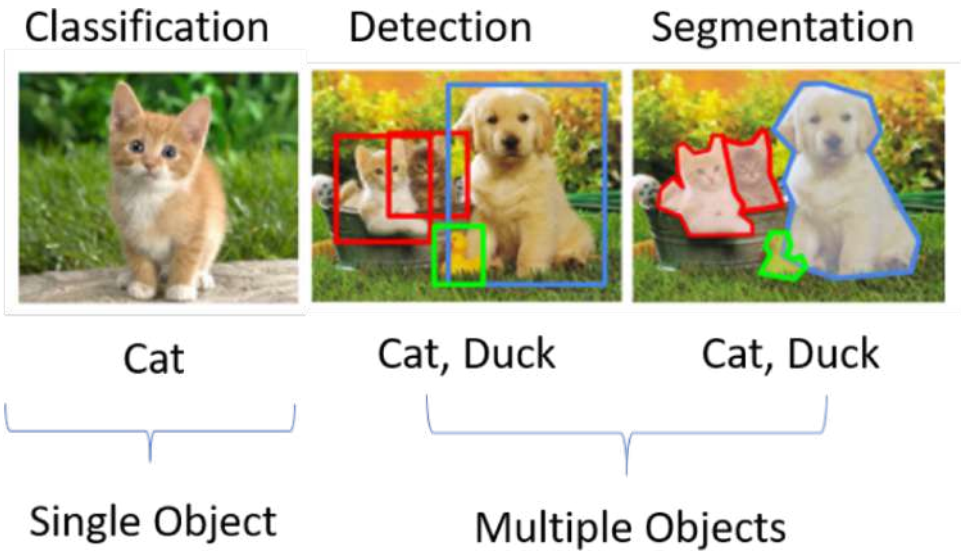


FIGURE 16. Three fundamental computer vision tasks are image classification, image segmentation, and object detection

Using deep learning for image segmentation means that we need to select a suitable neural network architecture that assigns a class to each pixel in the image, thereby partitioning the image into different zones. In this section, we work with the Oxford-IIIT Pets dataset. This dataset contains 7390 images of various breeds of dogs and cats together with foreground-background segmentation masks for each image. A segmentation mask is also an image with the same spatial dimensions as the input tensor. The channel vector for a given pixel has, however, only a single component that takes one of three integer values: 1 (foreground), 2 (background), or 3(contour).

We assume the dataset has already been loaded into a directory that has a subdirectory `images` holding the input images as `jpg` files and another subdirectory `annotations/trimaps` with the segmentation masks stored as `png` files. The following script creates a sorted tuple of path file names for the inputs (jpg images) and targets (png masks). We will use these tuples in building the datasets used in training our model.

```
import os

input_dir = "Oxford-IIIT-Pets/images/"
target_dir = "Oxford-IIIT-Pets/annotations/trimaps/"
img_size = (160, 160)
num_classes = 3
batch_size = 32

input_img_paths = sorted([
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ])
target_img_paths = sorted([
        os.path.join(target_dir, fname)
        for fname in os.listdir(target_dir)
        if fname.endswith(".png") and not fname.startswith(".")
    ])

selected_img = 600
print("Number of samples:", len(input_img_paths))
```

```
for input_path, target_path in zip(input_img_paths[selected_img],
                                   target_img_paths[selected_img]):
    print(input_path, "|", target_path)
```

The following script displays a randomly selected sample in the dataset so we can see what the image and masks look like. We use the Keras utility functios `load_img` to generate a PIL array (Python Imaging Library) that can then be displayed using `matplotlib`'s function `imshow`. This is done directly for the input image, but special processing has to be done for the segmentation mask. Since each channel of the segmentation mask is binary valued to indicate if a pixel is foreground, background, pixel, we need to convert this into an array that can be displayed by `imshow`. This involves first loading the mask into an array and then transforming each pixel in that array to `uint8` type so it represents a grey scale value. This would generate an array where each pixel has the value of 1, 2, or 3. We need to rescale these values so they are spread between 0 and 255. So we map 1 to 0 (black - foreground), 2 to 127 (background - green), and 3 to 254 (contour - yellow). The images for the 397th sample are shown side by side in Fig. 17

```
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array
import random

figure, axis = plt.subplots(1,2)
axis[0].axis("off")

sample = random.Uniform(len(input_paths))
axis[0].imshow(load_img(input_paths[sample]))
axis[0].set_title(f"input {sample: 4d}")
img = img_to_array(load_img(target_paths[9], color_mode="grayscale"))
normalized_array = (img.astype("uint8") - 1) * 127
axis[1].axis("off")
axis[1].set_title("target")
axis[1].imshow(normalized_array[:, :, 0])
```

We are now going to split the input and target samples into three mutually disjoint sets of p-training, validation, and testing samples. Before doing

FIGURE 17. Oxford Pets Database Input and Target Image

this, we shuffle the `input_paths` and `target_paths` tuples because the original ordering in the directory was sorted alphabetically. This is done to ensure our datasets are i.i.d. collections of the original database. We then split these path name tuples into the desired three collections. This leaves us with 1000 testing damples, 1917 validations samples, and 4473 p-training samples.

```
num_test_samples = 1000
num_train_samples = len(input_paths)-num_test_samples
print(f"{num_train_samples: 4d} training samples,
             {num_test_samples: 4d} testing samples")

import random
random.Random(1337).shuffle(input_paths)
random.Random(1337).shuffle(target_paths)

train_input_paths = input_paths[:-num_test_samples]
train_target_paths = target_paths[:-num_test_samples]
test_input_paths = input_paths[-num_test_samples:]
test_target_paths = target_paths[-num_test_samples:]

val_split = 0.30
num_val_samples = int(val_split*num_train_samples)
num_ptrain_samples = num_train_samples - num_val_samples
ptrain_input_paths  = train_input_paths[:-num_val_samples]
ptrain_target_paths = train_target_paths[:-num_val_samples]
val_input_paths     = train_input_paths[-num_val_samples:]
val_target_paths    = train_target_paths[-num_val_samples:]
```

We are going to use mini-batch training with a batch size of 32 samples. This can be done directly using the fit command, but if we presort the batches before calling the `fit` method, our training process will be faster. In the previous lectures we used Keras `Dataset` object to accomplish this. In particular we used a method that converted image files in a directory into the Dataset object. That function, however, assumed a particular structure to the image directories which is not satisfied by the Oxford Pet dataset. So we will need to write our own Dataset generator that has been customized for the Oxford Pet's directory structure. This dataset generator is built from the Keras base object `Sequence`. In particular, we instantiate a `Sequence` object that we call `OxfordPets`. This object returns a batched Dataset object that can be more efficiently used when fitting a model. The class definition for this object is given below.

```
class OxfordPets(keras.utils.Sequence):
    """Helper to iterate over the data (as Numpy arrays)."""

    def __init__(self, batch_size, img_size, input_img_paths, target_img_paths):
        self.batch_size = batch_size
        self.img_size = img_size
        self.input_img_paths = input_img_paths
        self.target_img_paths = target_img_paths

    def __len__(self):
        return len(self.target_img_paths) // self.batch_size

    def __getitem__(self, idx):
        """Returns tuple (input, target) correspond to batch #idx."""
        i = idx * self.batch_size
        batch_input_img_paths = self.input_img_paths[i : i + self.batch_size]
        batch_target_img_paths = self.target_img_paths[i : i + self.batch_size]
        x = np.zeros((self.batch_size,) + self.img_size + (3,), dtype="float32")
        for j, path in enumerate(batch_input_img_paths):
            img = load_img(path, target_size=self.img_size)
            x[j] = img
        y = np.zeros((self.batch_size,) + self.img_size + (1,), dtype="uint8")
        for j, path in enumerate(batch_target_img_paths):
            img = load_img(path, target_size=self.img_size, color_mode="grayscale")
            y[j] = np.expand_dims(img, 2)
            # Ground truth labels are 1, 2, 3. Subtract one to make them 0, 1, 2:
```

```
            y[j] -= 1
        return x, y
```

Our `OxfordPets` class consists of 3 methods that are used by Model class' `fit` method. These class methods are the constructor, `__init__`, a method (`__len__`) returning the number of batches in the returned Dataset, and an iterator method (`__getitem__`) which returns the next (input,target) tuple corresponding to a given batch in the dataset. This is, essentially, a utility that returns an image Dataset object from a directory where the input and target lie in directories whose paths need to be specified separately. The following script uses our custom utility to create the p-training, validation, and testing datasets. Let us check the shape of the tensors returned by our dataset object by using `__getitem__` to fetch one of the batches. We see that the input batch has shape $(32, 160, 160, 3)$, namely a batch of 32 input tensors of shape $(160, 160, 3)$. The target batch has shape $(32, 160, 160, 1)$, namely a batch of 32 target tensors of shape $(160, 160, 1)$. So the targets supplied to the model will be images whose pixels are encoded with an integer-valued classification.

```
batch_size = 32
img_size = (160, 160)
ptrain_ds = OxfordPets(batch_size, img_size,
                       ptrain_input_paths, ptrain_target_paths)
val_ds    = OxfordPets(batch_size, img_size,
                     val_input_paths, val_target_paths)
test_ds   = OxfordPets(batch_size, img_size,
                       test_input_paths, test_target_paths)

x,y = test_ds.__getitem__(3)
print(x.shape)
print(y.shape)
#(32,160,160,3)
#(32,160,160,1)
```

We now declare the model architecture used in the segmentation problem. This model has the *encoder-decoder* architecture shown in Fig. 16. The encoder has a pyramidal design that starts from an input tensor with

shape (x, y, z) where the spatial dimensions, x and y, are much larger than the channel dimension z. Each layer in this pyramid is a 2D convolutional model that down samples the spatial dimensions and increases the number of channels, so that at the top of the pyramid the channel dimension z is much larger than the spatial dimensions x, y. In particular, we think of these many output channels as *features* in the original image. For the classification problem, we mapped these features through a Dense layer onto a rank-1 tensor whose components were the likelihood of the image being of a particular class.

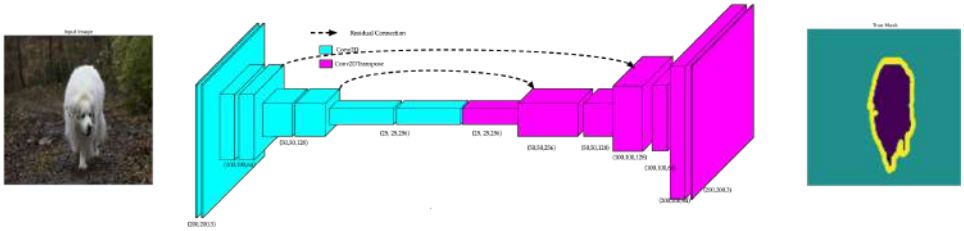


FIGURE 18. Encoder/Decoder (Unet) architecture used for Image Segmentation Tasks

In the image segmentation problem our target is another rank-3 tensor with shape (x, y, z_1) where x, y are the same as the original input image's spatial dimensions and the z_1 represents the number of classes that each pixel in the image is assigned to. In our Oxford Pet database those target classes are either 1 (foreground - the cat/dog), 2 (background), or 3 (contour). Since our model's output must be an image, we cannot use a simple Dense network on the encoder's outputs. Instead we use a *decoder*, which is an inverted pyramid that upsamples the encoder output's spatial dimensions and downsamples its feature channels until we get to the same shape as the input tensor.

The following script declares our model architecture for this problem. Note that there are a couple of differences from the model we used in the image classification problem. The first big difference is that we do not use the MaxPooling layer. The reason for this is because MaxPooling tends to

destroy local spatial information as we move up the pyramid. This loss of information will not work well with the decoder. So rather than downsampling through MaxPooling, we will use a *stride* of 2 to downsample. Note that this stride is declared directly in the convolutional layer.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

num_classes = 3
def build_model(img_size, num_classes):
    inputs = keras.Input(shape = img_size+(3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64,3,strides=2,activation="relu",padding="same")(x)
    x = layers.Conv2D(64,3,activation="relu",padding="same")(x)
    x = layers.Conv2D(128,3,strides=2,activation="relu",padding="same")(x)
    x = layers.Conv2D(128,3,activation="relu",padding="same")(x)
    x = layers.Conv2D(256,3,strides=2,activation="relu",padding="same")(x)
    x = layers.Conv2D(256,3,activation="relu",padding="same")(x)

    x = layers.Conv2DTranspose(256,3,activation="relu",padding="same")(x)
    x = layers.Conv2DTranspose(256,3,activation="relu",padding="same",strides=2)(x)
    x = layers.Conv2DTranspose(128,3,activation="relu",padding="same")(x)
    x = layers.Conv2DTranspose(128,3,activation="relu",padding="same",strides=2)(x)
    x = layers.Conv2DTranspose(64,3,activation="relu",padding="same")(x)
    x = layers.Conv2DTranspose(64,3,activation="relu",padding="same",strides=2)(x)

    outputs = layers.Conv2D(num_classes, 1, activation="softmax", padding="same")(x)

    model = keras.Model(inputs,outputs)
    return model

model = build_model(img_size = img_size, num_classes=3)
model.summary()

import tensorflow as tf
model.compile(
     optimizer=tf.keras.optimizers.Adam(learning_rate=1.e-3),
     loss = "sparse_categorical_crossentropy")
```

The preceding script follows the sequence of 2D convolutional layers with a sequence of *transposed 2D convolutional layers*, `Conv2DTranspose`. If we look at the model summary, we see that the input tensor and full model's output tensor have the same shape of $(160, 160, 3)$. This stack of transposed convolutional layers is called the model's *decoder*. The purpose of these transposed layers is to *upsample* the input, thereby increasing the spatial dimensions of the output tensor by a factor equal to the layer's stride length.

To help explain why we refer to `Conv2DTranspose` as a "transposed" layer, let us consider an input map of size 4×4 being convolved with a filter kernel 3×3 with stride 1 and no zero padding. This convolution is shown in Fig. 19. We rewrite the input and output tensors in this figure by stacking up their components as

$$X = \begin{bmatrix} x00 \\ x01 \\ x02 \\ x03 \\ x11 \\ \vdots \\ x32 \\ x33 \end{bmatrix}, \quad Y = \begin{bmatrix} y00 \\ y01 \\ y10 \\ y11 \end{bmatrix}.$$

The convolution in Fig. 19 may then be realized as the matrix-vector multiplication $Y = WX$ where W is a block Toeplitz matrix of the form

$$W = \left[\begin{array}{cccc|cccc|cccc|cccc} w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 & 0 & 0 & 0 & 0 \\ 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 \\ 0 & 0 & 0 & 0 & 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 \end{array}\right].$$

This is a matrix product that may be seen as a linear transformation from $X \in \mathbb{R}^{16}$ vector to a $Y \in \mathbb{R}^4$ vector, which "downsample" since Y has lower dimensionality. Actually a better term might be to say W project X down to the lower dimensional Y vector. The transposed product would be

$$X = W^T Y$$

where $W^T \in \mathbb{R}^{4\times16}$ is the transpose of our original W matrix given above. Since it is a linear transformation from $\mathbb{R}^4$ to $\mathbb{R}^{16}$, it may be seen as "upsampling" the input Y. Again, a more accurate term might be to say W^T lifts Y into X. The transposed convolution layer is, therefore, a convolutional layer whose weight tensor has been transposed in a similar way.

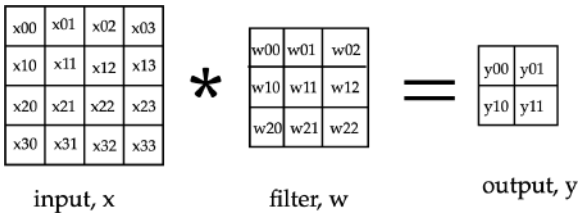


FIGURE 19. Convolution Operation

We are now ready to train our encoder-decoder model on the OxfordPets Dataset objects we have created. In particular, we will train for 20 epochs using an Adam optimizer with a sparse categorical cross-entropy loss function. Note that the sparse categorical cross-entropy loss is used because the classes in the target tensor are encoded as integers rather than one-hot encoded vectors. In this script we also saved the history returned by the fit method as a pickle file. This is done in case we want to generate plots later on without having to re-train the model.

```
callbacks = [
    keras.callbacks.ModelCheckpoint("tmp/oxford_segmentation_1.keras",
    save_best_only=True)]

epochs = 20
history = model.fit(ptrain_ds, epochs=epochs, validation_data = val_ds,
callbacks=callbacks)

import pickle
with open('history_oxford.pkl','wb') as file_pi:
    pickle.dump(history.history, file_pi)
```

The training curve is shown in Fig. 20. The training curve shows that the model begins overfitting around the 10th epoch, but that the validation loss

remains relatively low throughout the entire training session. This suggests that the model should perform well.

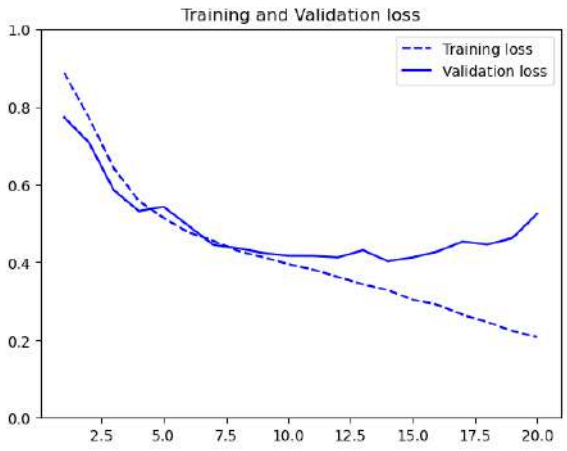


FIGURE 20. Training Curves for Encoder-Decoder Model used with Oxford Pets Segmentation Problem

Rather than using something like classification accuracy to measure the overall model's performance, we use the Intersection-over-Union metric (IoU). If we look at two sets, A and B, then

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

Namely it is the ratio of the number of elements that are in both A and B divided by the total number of distinct elements in A or B. We apply this to the segmentation mask as follows. Recall that the target segmentation mask has pixels taking values in $\{1, 2, 3\}$ whereas the output of the model whose pixels are 3-d vectors where each component's index is a class and the value of that component is the probability of that component being the correct classification. To use IoU we need to threshold the target segmentation mask so that its pixels are 1 if they are in the foreground (cat/dog) and zero otherwise. We need to transform the model's predicted output tensor into similar binary encoded map. This is done by first identifying which component in a given pixel's vector is largest and using that index to identify the foreground pixels. The following script declare a function `eval_IoU`

to compute the IoU for our particular images. The following script uses this function to compute the mean IoU metric over the entire testing data set. In this case, we obtained a mean IoU of 68%, which is considered relatively good.

```
test_model = keras.models.load_model("tmp/oxford_segmentation_1.keras")

test_preds = test_model.predict(test_ds)

import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array

num_samples = len(test_ds)*batch_size
print(f"number of test samples = {num_samples: 4d}")

def eval_IoU(img1,img2):
    img1 = img1.astype("uint8")
    img1 = (img1==1).astype("uint8")

    img2 = (img2==0).astype("uint8")

    metric = tf.keras.metrics.IoU(num_classes=2, target_class_ids=[0])
    metric.update_state(img1,img2)
    return metric.result().numpy()

mean_IoU = 0
#num_test_samples = 10
num_samples = len(test_ds)*batch_size
for i in range(num_test_samples):
    img = img_to_array(load_img(test_target_paths[i],
    target_size = img_size,color_mode="grayscale"))
    Image4 = (img.astype("uint8") - 1) * 127
    Image5 = np.argmax(test_preds[i], axis=-1)
    Image5 = np.expand_dims(Image5, axis=-1)
    pred_IoU = eval_IoU(Image4,Image5)
    mean_IoU += pred_IoU/num_test_samples

print(f" mean Test IoU = {int(mean_IoU*100): 3d}%")
```

To visualize how this IoU metric might be interpreted for the actual images, we randomly selected three samples from the test data set and then displayed the 1) original image, 2) the target segmentation mask, and 3)

the model's predicted segmentation mask along with the IoU computed for that prediction. These images are shown in Fig. 21 with IoU's ranging from 67% to 85%. The predicted masks appear to be rather close to the target masks.

```
figure, axis = plt.subplots(3,3)
for i in range(3):
    sample = int(random.uniform(0,num_samples))
    Image3 = load_img(test_input_paths[sample],target_size=img_size)
    axis[i][0].imshow(Image3)
    axis[i][0].axis('off')
    axis[i][0].set_title(f"input {sample: 4d}")


    img = img_to_array(load_img(test_target_paths[sample],
    target_size = img_size,color_mode="grayscale"))
    Image4 = (img.astype("uint8") - 1) * 127
    axis[i][1].imshow(Image4)
    axis[i][1].axis('off')
    axis[i][1].set_title('target')

    Image5 = np.argmax(test_preds[sample], axis=-1)
    Image5 = np.expand_dims(Image5, axis=-1)

    axis[i][2].imshow(Image5)
    axis[i][2].axis('off')

    pred_IoU = eval_IoU(Image4,Image5)

    axis[i][2].set_title(f'pred IoU: {int(pred_IoU*100): 4d}%')
```

5.1. Modern CNN Architectural Patterns: A CNN's architecture is the sum of choices that went into creating it: which layers to use, how to configure them, and in what arrangement to connect them. These choices define the model space, the space of all possible models that gradient descent can search over, parameterized by the model's weights. Like feature engineering, a good model space encodes our *prior knowledge* about the particular problem. For instance in using convolutional layers you are assuming that the relevant patterns in your image are translation invariant. The

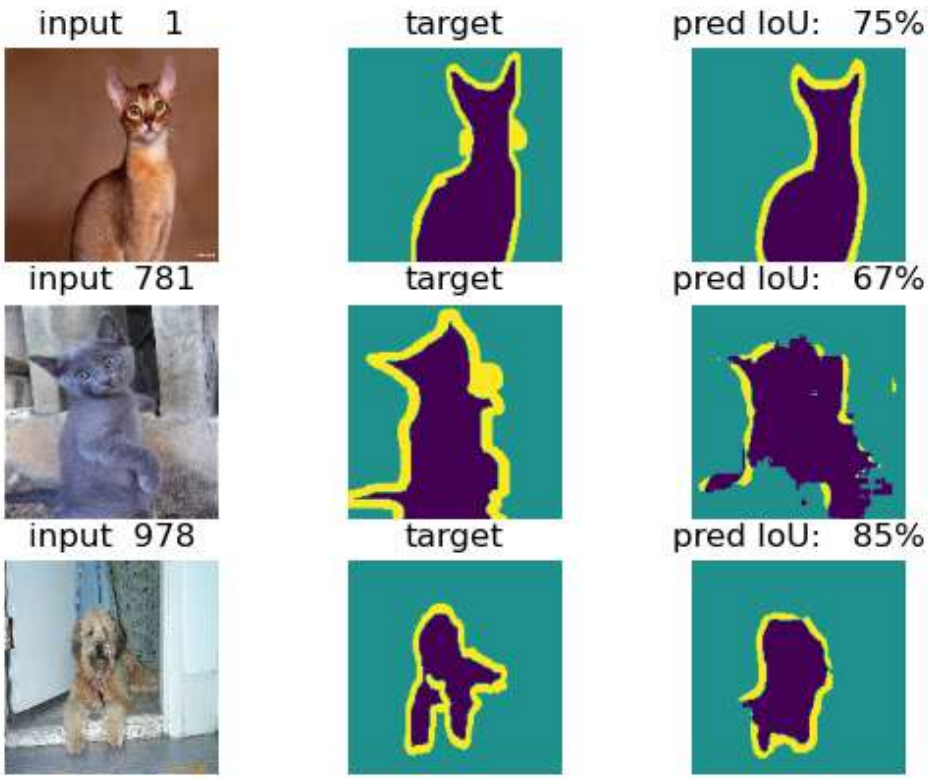


FIGURE 21. Results from Test Data

selection of good model architectures is more art than science. Experienced machine learning engineers use that "art" to cobble together high performance models. This subsection reviews several architectural features commonly found in high performance CNNs such as VGG16 [**SZ14**], Xception [**Cho17**], and U-net [**RFB15**].

One of the main themes in developing high performance CNNs is the use of *feature hierarchies*. This is basically the guiding principle in software object oriented design; where good programs are modular and hierarchically constructed from a library of modules. Deep CNNs take advantage of modularity, hierarchies, and reuse. The VGG16 base, for instance, was based on the repeating the following pattern

$$\text{Conv2D} \to \text{Conv2D} \to \text{MaxPooling}$$

over and over again as shown in Fig. 14.

High performance CNN's like VGG16 seem to suggest that the *deeper* the network is, the better it will perform. This is often seen in practice, but that performance comes at a cost. Namely, as we stack layers deeper, the gradient computed by backpropogation become smaller and smaller. This is called the *vanishing gradient problem* [**Hoc98**]. What this means is that the time it takes to train such networks gets very very long. In fact, training may appear to stall out if we don't do something. One approach for dealing with this problem is to force each function in the chain to be non-destructive. In other words, we modify the model so it retains a noiseless version of the input information by simply passing it up to the output. This is called a *residual connection*. It requires that we simply add the input to the network's output as shown in Fig. 22. The residual connection acts as an information short cut around the noisy blocks and enables gradient information from earlier layers to propagate noiselessly through the network. This technique was introduced with the ResNet family of models [**HZRS16**].

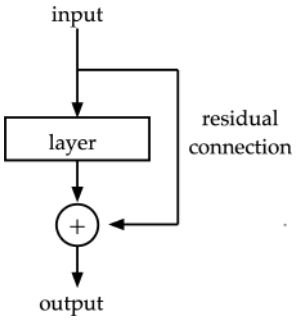


FIGURE 22. Residual Connection around processing layer

Note that adding the input back to the output of a block implies that the output should have the same shape as the input. This is not the case if your block includes convolutional layers with an increased number of filters or a max pooling layer. In such cases, we use a 1×1 `Conv2D` layer with no activation to linearly project the residual to the desired output shape. You would also use the control variable `padding=same` in the convolution layers in your target block to avoid spatial downsampling due to padding and you'd use strides in the residual projection to match any downsampling caused by a max pooling layer. For example, if we have a residual block where the number of filters changes, this would be written as

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(32,32,3))
x = layers.Conv2D(32,3, activation="relu")(inputs)

#set aside the residual
residual = x

#this is layer around which we create a residual connection
#the number of filters increases from 32 to 64
#and we use padding ="same" to avoid downsampling
x = layers.Conv2D(64,3, activation="relu", padding="same")(x)

#The residual has only 32 filters so we use a 1 times 1
#Conv2D to project it to the right shape
residual = layers.Con2D(64,1)(residual)

#then we add the residual in to the output
x = layers.add([x, residual])
```

Normalization is a broad category of methods that seek to make different samples seen by a machine learning model more similar to each other. This helps the model learn and generalize well to new data. The most common form of data normalization has already been used, where we center the data on zero by subtracting the mean and give the data a unit standard deviation by dividing by its standard deviation. Previous examples normalized data before feeding it into the model. But we may also be interested in normalizing the outputs after each layer transformation. This is what we mean by *batch normalization*. It is implemented as another layer, `BatchNormalization` that was introduced in [**IS15**]. It adaptively normalizes data even as the mean and variance change over time during training. During training, it uses the mean and variance of the current batch of data to normalize samples and during inference (forward prop) it uses an exponential moving average of the batch-wise mean and variance of the data seen during training. In practice the main effect of batch normalization appears to be that it helps with gradient propagation - similar to the residual connections - and allows for deeper networks.

The `BatchNormalization` layer can be used after any layer. Note that both the `Dense` and `Conv2D` layers have a bias vector. Because the normalization step centers the layer's output about zero, the bias is no longer necessary and so it can be switched off.

```
x = ...
x = layers.Conv2D(32,3,use_bias = False)(x)
#remove bias since layer gets normalized
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
#place activation layer after normalization
```

The *depth-wise separable convolution* layer performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise 1x1 convolution as shown by Fig. 23. This is equivalent to separating the learning of spatial features and the learning of channel-wise features. In much the same way that convolution relies on the assumption that the patterns in images are not tied to specific locations, depth-wise separable convolution relies on the assumption that spatial locations in intermediate activations are highly correlated but that different channels are highly independent. Because this assumption is generally true for the image representations learned by deep neural networks, it serves as a useful prior that helps the model make more efficient use of its training data. Depth-wise separable convolution requires significantly fewer parameters and involves fewer computations compared to regular convolution, while having comparable representational power. It results in smaller models that converge faster and are less prone to overfitting. These advantages become especially important when you are training small models from scratch with limited data. When it comes to larger scale models, depth-wise separable convolutions are the basis of the Xception architecture, a high performance CNN [**Cho17**].

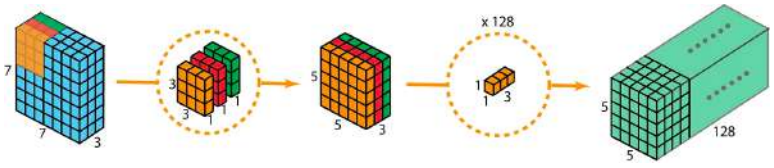


FIGURE 23. Depth-wise Separable Convolution

High performance image segmentation models often use many of the additional refinements discussed above. The U-net model in Fig. 16 incorporates all of the features shown above. The script and `visualkeras` view of this model are shown in Fig. 24.

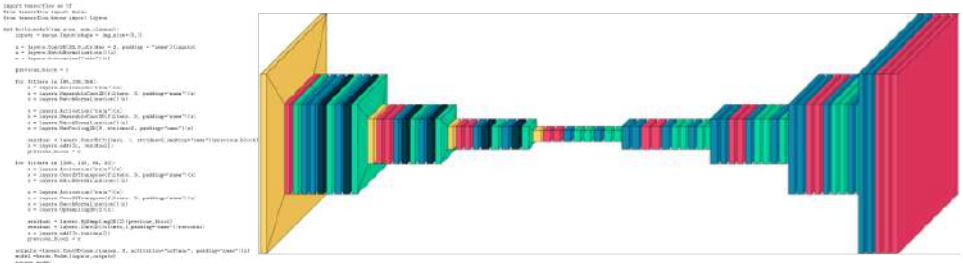


FIGURE 24. Advanced Unet Model for Image Segmentation Problem

6. Object Detection Task

Object detection is the computer vision task for finding objects of interest in an image. This is more involved than classification, which only tells you what the main subject of the image is. In object detection we find multiple objects, classify them, and then locate where they are in the image. An object detection model predicts *bounding boxes*, one for each object it finds, as well as the classification probabilities for each object as shown below in Fig. 25. It is common for object detection to predict too many bounding boxes. So each box is also given a confidence score that says how likely the model thinks this box really contains an object. As a post-processing step, we then filter out those boxes whose scores fall below a certain threshold. Object detection is harder than classification. One of the

problems encountered by object detection is that a training image can have between zero to dozens of objects in it and the model may output more than one prediction. We therefore need a way for comparing these predictions against ground-truth.

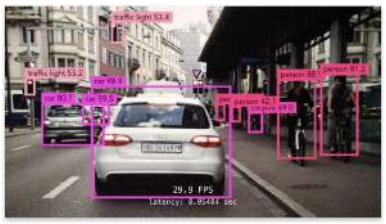


FIGURE 25. Example of Prediction made by an object detection model

Object detection models may be classified as being two-stage or one stage. Two stage models such as R-CNN [**RHGS15**] first generate so-called *region proposals* and then makes a prediction for each region. The region proposals are areas of the image that can potentially contain an object. The model then makes a separate prediction for each of these regions. This works very well but is rather slow as it requires running the detection and classification portion of the model multiple times.

One-stage detectors, on the other hand, require only a single pass through the neural network and they predict all bounding boxes in this first pass. This is much faster and more suitable for mobile devices. The most common example of one stage object detectors are YOLO [**RDGF16**] and SSD [**LAE+16**]. Unfortunately, the research papers for these models leave out important technical details. So this section is drawn from a recent blog post[**Hol18**].

Why object detection is hard? A classifier takes an image as input and produces a single output, the probability distribution over the classes. But this only gives you a summary of what is in the image as a whole, it doesn't work well when the image has multiple objects of interest. On the image in Fig. 26, a classifier might recognize that the image contains a certain

amount of "cat-ness" and certain amount of "dog-ness" but that's the best it can do.

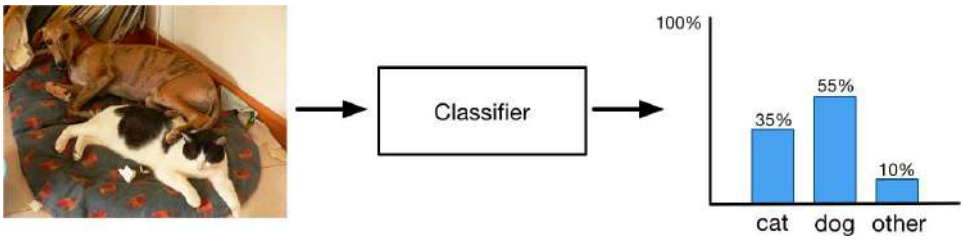


FIGURE 26. Classification Problem outputs a probability distribution for the entire image

An object detection model in Fig. 27, on the other hand, will tell you where the individual objects are by predicting a bounding box for each object. Since it can now focus on classifying the thing inside the bounding box and ignore everything outside, the model is able to give much more confident predictions for the individual objects. If your dataset comes with bounding box annotations (the so-called ground-truth boxes), it's pretty easy to add a localization output to your model. Simply predict an additional 4 numbers, one for each corner of the bounding box. Now the model has two outputs:

- The probability distribution for the classification result, and
- a bounding box regression.

The loss function for the model simply adds the regression loss for the bounding box to the cross-entropy loss for the classification, usually with the mean squared error (MSE). You then use SGD to optimize the model as usual, with this combined loss to produce the output shown below.

The left side of Fig. 28 shows an example prediction using this one-stage model. It looks reasonable, but how do we measure the accuracy of the predictor. Scoring how well the predicted box matches ground-truth is done by computing the IOU or *intersection-over-union* between the two bounding boxes. The IOU is a number between 0 and 1, with larger being

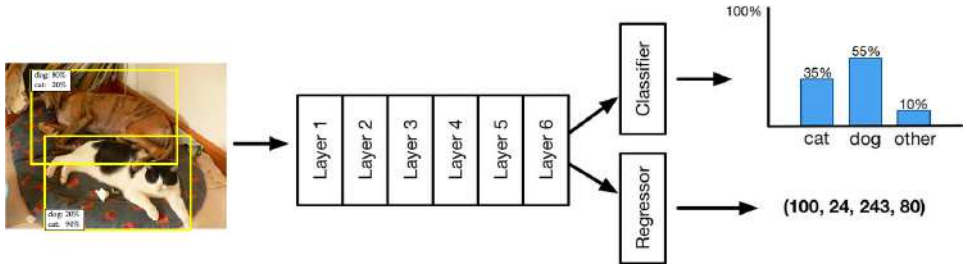


FIGURE 27. Object Detection Model outputs a classification probability for a bounding box. The bounding box is determined by a regressor model and the probability is determined by a classifier model, both built on top of a CNN base.

better. Ideally, the predicted box and the ground-truth have an IOU of 100%, but in practice anything above 50% is usually considered to be a correct prediction. For the above example the IOU was 75% and so the boxes are a good match.

FIGURE 28. One-stage Object Detection - (left) good result - (right) bad result

Using a regression output to predict a single bounding box gives good results. However, just like classification does not work well when there are multiple objects in the image, so this will fail our simple localization scheme as shown on the right side of Fig. 28. In this case the model can predict only one bounding box and so it has to choose one of two horses in the image. But rather than selecting one object, it takes an average which

gives a box somewhere in between the two horses. One might think that this can be solved by adding more bounding box detectors. But even with a model that has multiple detectors, we still get bounding boxes that all end up in the middle of the image. This occurs because the model does not know which bounding box to assign to object and so it plays it safe by placing both boxes somewhere in the middle.

One-stage detectors such as YOLO and SSD all solve this problem by assigning each bounding box detector to a specific position in the image. That way the detectors learn to specialize on objects *in certain locations*. This is done by using a *fixed grid of detectors* and this is one of the things that set apart one-stage detectors from the region proposal-based detectors such as R-CNN.

Let us consider the simplest possible architecture for this kind of model as shown in Fig. 29. It consists of a base network that acts as a feature extractor. Like most feature extractors it is typically trained on the ImageNet dataset. In the case of YOLO, the feature extractor take a 416x416 pixel impage as input. SSD typically uses 300x300 images. These are larger than the images used for classification between small details in the image may be important for correct localization.

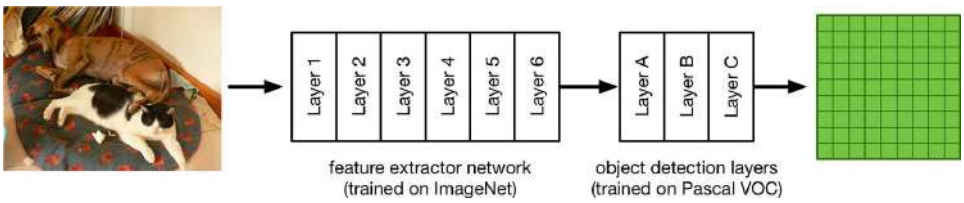


FIGURE 29. One-stage Detector Model Architecture consists of a pre-trained feature extraction network followed by a object detection layer that classifies the images inside each preset bounding box shown in the output.

The base network can be anything such as Inception, ResNet or YOLO's DarkNet. For a mobile application it makes sense to use a small fast architecture such as MobileNet [SHZ+18]. On top of the feature extractor are several convolutional layers. These are fine-tuned to learn how to predict bounding boxes and class probabilities for the objects inside these bounding boxes. This is the object detection part of the model. You can check Keras website for a notebook tutorial on Object detection using pre-trained networks. There are numerous other sites around as well.

7. Visualizing what CNNs Learn - the problem of model interpretability

A fundamental problem when building a computer vision application is that of interpretability [SWM17]. Why did the classifier make the given classification? What were the features that it identified which would be associated with that class? While one often thinks of a deep network as a black box, in fact the feature maps of a CNN can be visualized in a manner that allows us to identify what characteristics of the original image were critical in its classification. This visualization plays a major role in giving us a way to interpret how a CNN reached its decision. Three of the most useful visualization methods are

- Visualizing intermediate CNN outputs: This method visualizes the features selected by successive CNN layers for a specific input.
- Visualizing CNN Filters: This is used to visualize those input patterns responsible for maximizing the output of a particular filter in the CNN.
- Visualizing Class Activation Mappings (CAM): This is used to understand which parts of a specific image played a major role in triggering a particular feature in the CNN.

The following subsections are drawn from [Cho21] to discuss each of these interpretation methods.

7.1. Visualizing Intermediate Activations: Visualizing intermediate activations consists of displaying the values returned by various convolution and pooling layers in a model, *given a certain input*. The output of a layer is called its activation. This gives a view into how an input is decomposed across the different filters learned by the network. We want to visualize feature maps with three dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently viewing the contents of every channel as a 2D image. Let us start by loading the model that we saved after training a CNN on the Cat/Dog database with data augmentation. Let us also fetch an input image `input_img` that is not part of our original Dog/Cat dataset. The script and associated are shown in Fig. 30

```
from tensorflow import keras
test_model_aug = keras.models.load_model("best_model_aug.keras")
model.summary()

from tensorflow.keras.utils import load_img, img_to_array, get_file
img_path = get_file(fname="cat.jpg",
                    origin="https://img-datasets/s3.amazonas.com/cat.jpg")
input_img = load_img(img_path, target_size = (180,180))
input_img = img_to_array(input_img)
input_img = np.expand_dims(input_img, axis=0)
print(input_img.shape)
```

FIGURE 30. Script and Image used to visualize CNN feature activation

To extract the feature maps, we want to look at, we'll create a Keras model that takes batches of images as input, and that outputs the activation of all convolution and pooling layers. We then feed our "cat" image to this model which returns the values of the layer activations as a list.

```
from tensorflow.keras import layers
layer_outputs = []
layer_names = []
for layer in test_model_aug.layers:
    if isinstance(layer, (layers.Conv2D, layers.MaxPooling2D)):
        layer_outputs.append(layer.output)
        layer_names.append(layer.name)
activation_model = keras.Model(inputs=test_model_aug.input, outputs = layer_outputs)
activations = activation_model.predict(input_img)
```

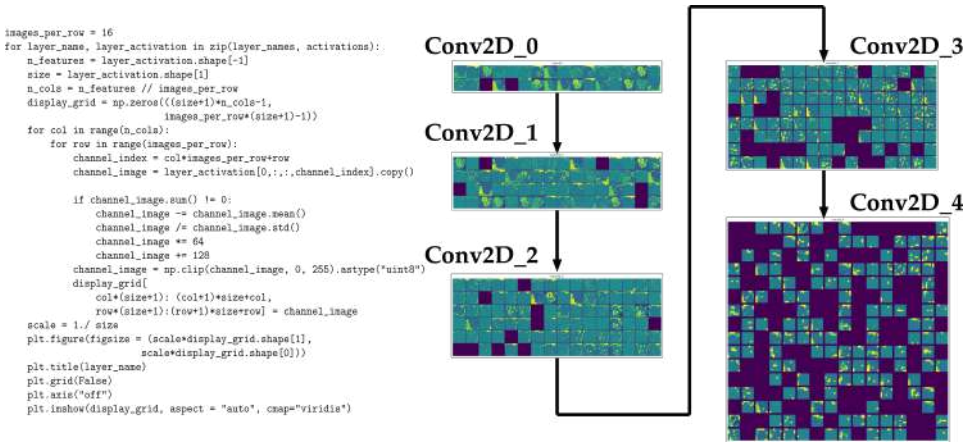


FIGURE 31. Script and Grid of Convolutional Layer Activations for Cat Image

For each layer in our list, `layer_name`, we take the activations and display them on a grid. This is done in the following script and the resulting activations are shown in Fig. 31.

There are a few things to note

- The first layer acts as a collection of various edge detectors. At that stage the activations retain almost all of the information present in the initial picture.
- As you go deeper, the activations become increasingly abstract and less visually interpretable. They begin to encode higher-level concepts such as ”cat ear” and ”cat eye”. Deeper representations carry increasingly less information about the visual content of the image and increasingly more information related to the class of the image.
- The sparsity of activations increases with the depth of the layer: in the first layer, almost all filters are activated, but in the following layers, more and more filters are blank. This means that the pattern encoded by the filter was not present in the input image.

We have just seen an important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer become increasingly abstract with the depth of the layer. The activations of higher layers carry less information about the specific input being seen and more about the target. A deep neural network effectively acts as an information distillation pipeline with raw data going in and being repeatedly transformed so that irrelevant information is filtered out and useful information is magnified and refined.

7.2. Visualizing Inputs Triggering CNN Filters: . Another way to inspect the filters learned by the model is to display the visual pattern that each filter responds to. This can be done with gradient ascent in the input space. In particular, we apply gradient descent to the value of the CNN's input image so as to maximize the response of a specific filter, starting from a blank input image. The resulting input image will be one that the chosen filter is maximally responsive to.

Let us try this with the previous CNN model we trained on the Cat/Dog dataset. We are going to pick the filter outputs from one of the convolution layers in our model. From the `model.summary()` we know these layers are named `conv2d_X` where `X` is either blank, 1,2, ,3, or 4. We will look at the 126th filter for layer `conv2d_3` and the 14th filter for layer `conv2d_1`. In particular, we need to declare a model that we will call `extractor` that goes from the given cat image to the third convolutional layer's outputs. We will then use this model to compute the activation levels for that convolutional layer. The following script does this for `conv2d_3`

```
#get our image of the cat
img_path = get_file(fname,"cat.jpg",
                    origin="https://img-datasets/s3.amazonas.com/cat.jpg")
input_img = load_img(img_path, target_size = (180,180))
input_img = img_to_array(input_img)
input_img = np.expand_dims(input_img, axis=0)

#get the model
```

```
test_model_aug = keras.models.load_model("best_model_aug.keras")

#create the extractor model
layer_name = "conv2d_3"
nfilt = 126
layer = test_model_aug.get_layer(name = layer_name)
extractor = keras.Model(inputs = test_model.aug.input, outputs = layer.output)
activation = extractor(input_img)
```

We now define a loss function for the `extractor` model and then define a function performing a gradient ascent step.

```
import tensorflow as tf
def compute_loss(image, filter_index):
    activation = feature_extractor(image)
    filter_activation = activation[:,:, :, filter_index]
    return tf.reduce_mean(filter_activation)

@tf.function
def gradient_ascent_step(image, filter_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss= compute_loss(image, filter_index)
    grads = tape.gradient(loss, image)
    grads = tf.math.l2_normalize(grads)
    image += learning_rate * grads
    return image
```

We then implement the gradient ascent training on a blank input image using the script in Fig. **??** and then display the image. This should be the input image that generated the activation pattern seen from our cat image. The right side of Figure 32 shows this maximizing input. What we notice is that these filters seem to respond the most to particular *textures* in the original image.

The fact that activation of higher level nodes appear to map to ”textures”, rather than specific cognitive features is interesting. It suggests that higher level nodes may not really be ”features” that are readily recognizable. Rather it is the pattern of activations in higher level nodes that are responsible for specific classification outcomes. This observation seems to

FIGURE 32. Maximal Inputs generating filter activations

be at odds with the traditional view that high level nodes in hierarchical CNNs encode image features. This insight has recently been taken advantage of by a new type of generative model (see chapter 7) called a *diffusion model*[**HJA20**].

7.3. Class Activation Mapping (CAM):. This visualization technique is used to identify which parts of an image give rise to a classification decision. This technique is called *class activation map (CAM) visualization*. It consists of producing heat maps of class activation over input images. A class activation heat map is a 2D grid of scores associated with a specific output class, computed for every location of any input image. These values indicate how important that part of the given image is with respect to the class under consideration. This is an important technique used in the non-destructive testing of objects. A manufactured part, for instance, may be scanned to see if the part is ”acceptable”. This is a binary classification problem. Classifying the manufactured part as ”unacceptable” might mean that it should be returned to the assembly line. CAM methods can be used to identify where the manufactured part was out of specs.

The specific implementation used in [Cho21] is described in the article [SCD+17]. Grad-CAM consists of taking the output feature map of a convolutional layer, given an input image, and weighing every channel in that feature map by the gradient of the class with respect to the channel. One may view this as weighting a spatial map by how intensely the input image activates different channels. We will demonstrate this technique using the pretrained Xception model.

```
model = keras.applications.xception.Xception(weights="imagenet")
```

We consider the image of two African elephants shown in Fig. 33 (left). Convert this image into a tensor the Xception model can read: the model was trained on images of size 299x299, preprocessed according to a few rules described in Keras version of this model. So we need to load the image, resize it, convert it into a NumPy tensor, and then apply these preprocessing rules.

```
img_path = keras.utils.get_file(
       fname="elephant.jpg",
       origin="https://img-datasets.s3.amazonaws.com/elephant.jpg")

def get_img_array(img_path, target_size):
       img = keras.utils.load_img(img_path, target_size=target_size)
       array = keras.utils.img_to_array(img)
       array = np.expand_dims(array, axis=0)
       array = keras.applications.xception.preprocess_input(array)
       return array

img_array = get_img_array(img_path, target_size=(299, 299))
```

Finally we run the pretrained network on the image and decode its prediction vector back to a human-readable format:

```
preds = model.predict(img_array)
print(keras.applications.xception.decode_predictions(preds, top=3)[0])

#[('n02504458', 'African_elephant', 0.86992675),
#('n01871265', 'tusker', 0.07696861),
#('n02504013', 'Indian_elephant', 0.023537217)]
```

So the three classes predicted for this image are African elephant (87%), Tusker (7%), and Indian elephant (2%). We can see which entry of the prediction vector that was maximally activated is the one correspoding to the ”African elephant” class at index 386.

```
np.argmax(preds[0])
# 386
```

To visualize which parts of the image are most ”African Elephant”-like, let us set up the Grad-CAM process. We first create a model that maps the input image to the activations of the convolutional layer.

```
last_conv_layer_name = "block14_sepconv2_act"
classifier_layer_names = [
        "avg_pool",
        "predictions",
]
last_conv_layer = model.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

We then create a model that maps the activations of the last convolutional layer to the final class predictions.

```
classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
x = classifier_input
for layer_name in classifier_layer_names:
           x = model.get_layer(layer_name)(x)
classifier_model = keras.Model(classifier_input, x)
```

Then we compute the gradient of the top predicted class for our input image with respect to the activations of the last convolutional layer

```
import tensorflow as tf

with tf.GradientTape() as tape:
      last_conv_layer_output = last_conv_layer_model(img_array)
      tape.watch(last_conv_layer_output)
      preds = classifier_model(last_conv_layer_output)
      top_pred_index = tf.argmax(preds[0])
      top_class_channel = preds[:, top_pred_index]
```

```
grads = tape.gradient(top_class_channel, last_conv_layer_output)
```

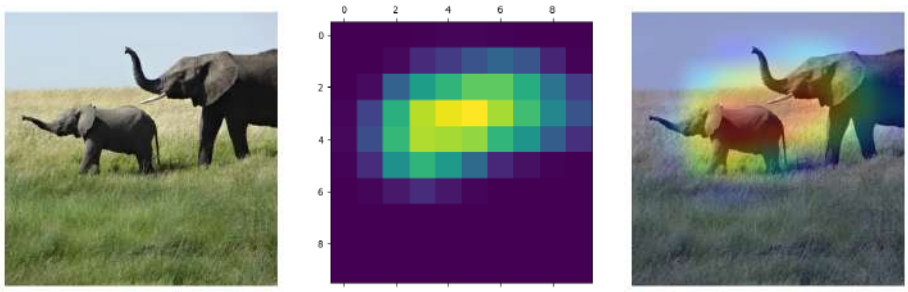


FIGURE 33. Grad-CAM heatmap visualization. (left) raw image (middle) heatmap (right) heatmap superimposed on raw image

We now apply pooling and importance weighting to the gradient tensor to obtain our heat map. We then normalize the heatmap's values between 0 and 1, display it in middle of Fig. 33 and then superimpose the heatmap over the original image (left of Fig. 33).

```
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2)).numpy()
last_conv_layer_output = last_conv_layer_output.numpy()[0]
for i in range(pooled_grads.shape[-1]):
            last_conv_layer_output[:, :, i] *= pooled_grads[i]
heatmap = np.mean(last_conv_layer_output, axis=-1)

heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)


#superimpose heatmap over image
import matplotlib.cm as cm

img = keras.utils.load_img(img_path)
img = keras.utils.img_to_array(img)

heatmap = np.uint8(255 * heatmap)

jet = cm.get_cmap("jet")
jet_colors = jet(np.arange(256))[:, :3]
```

```
jet_heatmap = jet_colors[heatmap]

jet_heatmap = keras.utils.array_to_img(jet_heatmap)
jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
jet_heatmap = keras.utils.img_to_array(jet_heatmap)

superimposed_img = jet_heatmap * 0.4 + img
superimposed_img = keras.utils.array_to_img(superimposed_img)

save_path = "elephant_cam.jpg"
superimposed_img.save(save_path)

import matplotlib.image as mpimg
img = mpimg.imread("elephant_cam.jpg")
plt.axis("off")
plt.imshow(img)
```

This seems to show that the ”ear” of the animal was most important in discriminating between the three possible classes identified above (African elephant, Tusker, Indian, elephant).

CHAPTER 6

Deep Learning for Natural Language Processing

Recurrent neural network (RNN) and Transformers are used on sequenced inputs often appearing in natural language processing applications. Consider a supervised learning problem in which a sequence of inputs are used to predict a sequence of outputs. Examples of sequence prediction tasks include

- Speech-to-text tasks that take a sampled audio waveform of human speech and outputs the text that was spoken.
- Machine translation takes a sentence in one language and produces a translation of that sentence into another language
- Time series prediction takes a sequence of input measurements observed from a dynamical system and predicts the next measurement that the system will generate.

This chapter examines methods used to solve sequence prediction problems using two types of neural network architectures; the *recurrent neural network* (RNN) [**HS97**] and the *transformer*[**VSP+17**]. These models are a powerful way of forecasting the future behavior of a complex dynamical system that might be used to model natural phenomena like the weather or turbulent flows. These models are also used for *natural language processing* (NLP); namely working with language created by humans.

This chapter looks at how RNNs and transformers are used in Natural Language Processing tasks. We start with a simple example in time series prediction to demonstrate why an RNN is better than a sequential or even a CNN in forecasting future outputs. We then turn to tasks in Natural

Language Processing such as Text Classification, Sentiment Analysis, and Neural Machine Translation (NMT). We illustrate how RNNs were initially used to solve these problem in 2015-2016. We then introduce a powerful neural network model called the *transformer* that appeared in 2017-2018 and has since become the dominant model for NLP tasks.

1. Motivating Example

This section uses an example from time series forecasting to demonstrate that feedforward sequential and 1D convolutional models do not work well in forecasting future outputs of a dynamical system. We start with a `csv` file of climate data on Kaggle and then extract the data from that file.

```
 import os
fname = os.path.join("datasets/jena_climate_2009_2016.csv")
with open(fname) as f:
       data = f.read()
lines = data.split("\n")

print(f"data line headers:\n {lines[0]}")
header = lines[0].split(",")
lines = lines[1:]

print(f"\njena dataset has {len(lines)} data lines")
for line in lines:
     print(line)
     break
```

This script creates a `list` whose entries contain the weather data taken every 10 minutes from January 1, 2009 to January 1, 2017. Each line of data has 15 data entries for time of day and various meteorological measurements including pressure, temperature, and such. So the dataset has 144 lines of data for each day over a span of 8 years. We plot one of these data entries (temperature) over an 8 year and 10 day to get some idea of its variability at different time scales. Temperature is the the third entry in each data line. The script and resulting plot are shown in Fig. 1.

```
import numpy as np
temperature = np.zeros((len(lines),1))
raw_data = np.zeros((len(lines), len(header)-1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i,:] = values[:]

from matplotlib import pyplot as plt
figure, axis = plt.subplots(1,2, figsize=(8,2.5))
figure.tight_layout()
axis[0].plot(range(len(temperature)),temperature)
axis[0].set_title("8 year temperature variation")
ndat10 = 10*144
axis[1].plot(range(ndat10),temperature[:ndat10])
axis[1].set_title("10 days temperature variation")
```

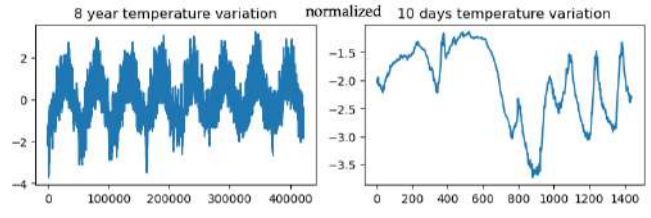


FIGURE 1. Jena Dataset Temperature data over 8 year and 10 day span

Note that not all of the data is relevant to predicting the future temperatures. We can see which data entries are most relevant by computing a correlation matrix for the dataset. This correlation heatmap is shown in Fig. 2. It shows that pressure (p mbar), temperature (T degC), maximum vapor pressure (VPmax mbar), vapor pressure deficit (VPdef mbar), specific humidity (sh g/kg), airtight (rho - specific volume - m/g**3), and wind velocity (wv m/s) are most highly correlated with temperature (T). So rather than forming our input tensor from all meteorological measurements we just use these 7.

```
import pandas as pd
df = pd.read_csv("datasets/jena_climate_2009_2016.csv")

def show_heatmap(data):
      plt.matshow(data.corr())
      plt.xticks(range(data.shape[1]),data.columns, fontsize=10, rotation=90)
      plt.gca().xaxis.tick_bottom()
      plt.yticks(range(data.shape[1]), data.columns, fontsize=10)

      cb = plt.colorbar()
      cb.ax.tic_params(labelsize=10)
      plt.title(Feature Correlation Heatmap", fontsize=14)
      plt.show()

show_heatmap(df)
```

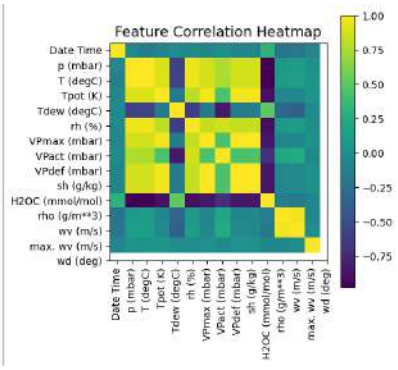


FIGURE 2. Correlation Heatmap for Data

We now form the datasets for this problem. In this example, we will split the available data into a p-training dataset (71%) and the rest will be used for validation. Note that the lines of data have a strong causal ordering. This means that future data lines are independent of past data lines. Because we want the validation data to be independent of the p-training data, we select

the validation time series data points *after* those in the p-training data. We are also going to normalize the data by subtracting off the mean and dividing by the standard deviation.

```
num_ptrain_samples = int(0.7125*len(raw_data))
num_val_samples = len(raw_data)-num_ptrain_samples
print("num_ptrain_samples:", num_ptrain_samples)
print("num_val_samples:", num_val_samples)

mean = raw_data[:num_ptrain_samples].mean(axis=0)
raw_data -= mean
temperature -= mean[1]
std = raw_data[:num_ptrain_samples].std(axis=0)
raw_data /= std
temperature /= std[1]
```

In forming the datasets, we construct the batches by "sampling" the `raw_data` and select an input sequence length. In particular, we decide to take hourly measurements of the weather over 5 days. Since `raw_data` was gathered every 10 minutes, this means we sample the time series data every 6 samples and generate 120 such samples. Our input tensor to our model will therefore be a rank-2 tensor of shape $(120, 7)$. In other words, all 7 critical meteorological measurements determined from Fig. 2 will be used to make our prediction.

Our target will be the *temperature one day (24 hours) later*. Keras has a utility that converts lines of time series data into a numpy array. This also does the sampling and batching. We also shuffle our batches. The following script creates the three datasets.

```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 -1)
batch_size = 256

ptrain_ds = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets = temperature[delay:],
    sampling_rate = sampling_rate,
```

```
    sequence_length = sequence_length,
    #shuffle=True,
    batch_size = batch_size,
    start_index = 0,
    end_index = num_ptrain_samples)

val_ds = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets = temperature[delay:],
    sampling_rate = sampling_rate,
    sequence_length = sequence_length,
    #shuffle=True,
    batch_size = batch_size,
    start_index = num_ptrain_samples,
    end_index = num_ptrain_samples+num_val_samples-delay-1)
```

Recall that one part of the data preparation phase is computing a baseline model from the raw data. Our baseline model assumes that the temperature for the next day is the same as today's temperature. The following script does this by taking the normalized samples in dataset and computing the mean squared error of the current temperature with that one day later. This computation shows that the MSE over the validation dataset is 0.15 and this value represents the MSE that our model will have to beat for us to have confidence that the model has actually learned something.

```
def baseline_model(dataset):
    total_square_err = 0
    samples_seen = 0
    for samples,targets in dataset:
        preds = samples[:,-1,1]
        total_square_err += np.sum(np.square(preds-targets))
        samples_seen += samples.shape[0]
    return (total_square_err/samples_seen)

baseline_val = baseline_model(val_ds)
print(f"Validation MSE: {baseline_val: .2f}")
```

We now consider three different models, a 1D convolutional model, a sequential model, and an RNN called a long-short term memory (LSTM) model. The CNN is a stack of two 1D convolutional layers using max

pooling for downsampling. The sequential model has 16 nodes followed by a dense layer. The LSTM is a stack of two LSTM layers. In both cases, we use an Adam optimizer with learning rate of 0.001 and we add a L2 regularization kernel with $\lambda = 0.005$. We train all 3 models for 20 epochs using the p-training data and evaluate its performance with respect to MSE and MAE on the validation dataset.

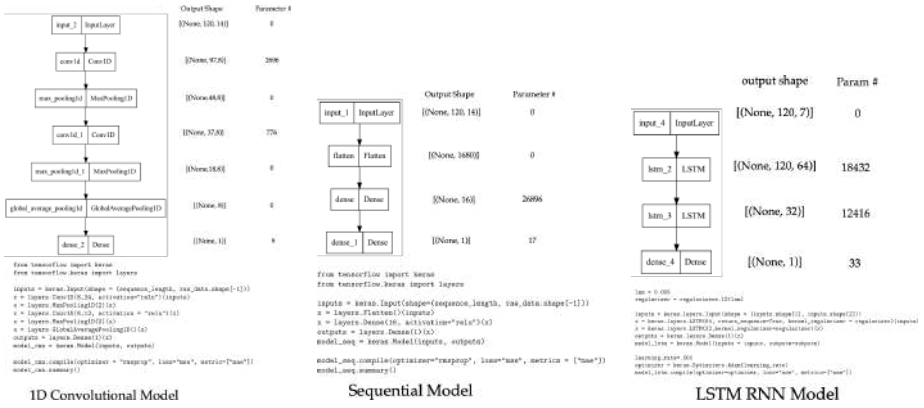


FIGURE 3. Left - 1D CNN Model, Middle - Seq Model, Right - LSTM Model

We trained each model for 20 epochs on the p-training data. Fig. 4 shows the training curves (p-training and validation) for the model's MSE (loss). These figures also plot the baseline MSE that was computed above. All of these models appear to be learning since both the p-training and validation losses are decreasing functions of training epoch. In all cases, we see the learning gap (difference between training and generalization loss) get very small. This indicates that all of the models have been trained to a point where there is nothing more than can do with the data. However, we see that the best validation MSE for the CNN model is 0.321 and this is well above the baseline value of 0.15 we computed earlier. So clearly the CNN model is not the right architecture to use for the time series data.

The sequential model's training curves are shows in the middle plot of Fig. 4. Again we see that the training error is small. We see that the validation MSE of the best sequential model is 0.182. This is slightly greater than the baseline MSE, but it is much better than what we obtained using the CNN model. This suggests the sequential model with its dense interconnections was able to do a better job of time series prediction. The training curve for the LSTM is shown on the right of Fig. 4. In this case we see the validation eventually falls so it is comparable with the baseline MSE. In fact this is probably the best that any model could have done, for most of the baseline MSE is due to noise on the signal as we can see in the time series plots. This RNN model is much better at capturing the sequential causality in the original time series. The following section discusses this RNN model in more detail.

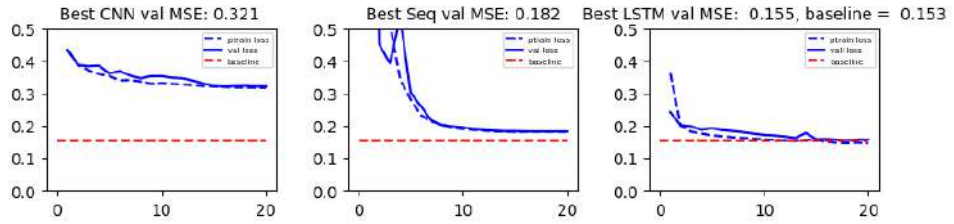


FIGURE 4. Training Curves for model MSE. (left) 1D Convolutional Model, (middle) Sequential Model, (right) LSTM model)

2. Recurrent Neural Networks

A recurrent neural network (RNN) is a model architecture where the output of a hidden node not only depends on the input, but also on the "past" output from the hidden node. You can therefore think of an RNN as a *dynamical system* in which the activation levels of the hidden layers are the system's states. This means that the computation graph of an RNN has *self loops* in contrast to the graphs of feedforward sequential models.

An RNN's computation graph is shown below on the right side of Fig. 5. We assume the input is a sequence of tensors, $\mathbf{x}^{(k)}$, being the kth element of that sequence. This input is multiplied by the input weights, $\mathbf{U}$ and then added to the *past* activation of the hidden unit, $\mathbf{s}^{(k-1)}$, weighted by the learned matrix $\mathbf{W}$ before being passed through the activation function, θ, to obtain the kth activation level, $\mathbf{s}^{(k)}$ of the hidden layer. In other words, the RNN computes its output $\mathbf{h}^{(k)}$ at time k as

$$\begin{aligned} \mathbf{s}^{(k)} &= \theta(b + \mathbf{W}\mathbf{s}^{(k-1)} + \mathbf{U}\mathbf{x}^{(k)}) \\ \mathbf{h}^{(k)} &= c + \mathbf{V}\mathbf{s}^{(k)} \end{aligned} \tag{35}$$

where b and c are biases and the output layer simply uses a linear activation function.

The computation graph on the left of Fig. 5 represents equation (35). These equations are seen in the bottom three nodes of the graph with the delayed input $\mathbf{s}^{(k-1)}$ be obtained through a delay element. The upper two nodes of this graph illustrate the computation of the loss, $\mathbf{L}^{(k)}$, at time k from the output $h^{(k)}$ and the target $\mathbf{y}^{(k)}$.

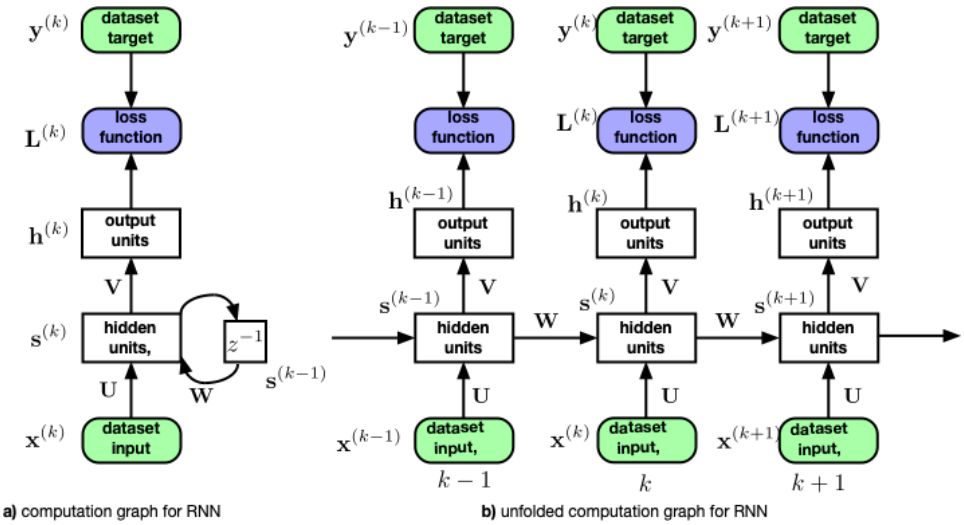


FIGURE 5. (left) RNN computation graph (right) unfolded RNN computation graph

We can also visualize how an RNN computes and trains by *unfolding* the computation graph on the left of Fig. 5. This unfolded graph is shown on the right side of Fig. 5. The unfolded graph is obtained by explicitly expanding out the delay so we show the computations done at each time step. As seen on the right of Fig. 5, the unfolded graph is, essentially, a feedforward neural network whose weights are shared between all time steps. Fig. 5 shows this unfolding for an input sequence of length 3. The graphic explicitly shows the output $\mathbf{h}^{(k)}$ for each input sequence element $\mathbf{x}^{(k)}$. This output is used to compute the model's loss, $\mathbf{L}(\mathbf{y}^{(k)}, \mathbf{h}^{(k)})$, with respect to the kth target $\mathbf{y}^{(k)}$. This graph represents the forward propagation of the network. We determine the weights, $\mathbf{W}$, $\mathbf{V}$, and $\mathbf{U}$ using the backpropagation algorithm. Since we have an explicit feedforward acyclic computation graph on the right side of Fig. 5, we can use automatic differentiation to readily compute the gradient and perform a stochastic gradient descent algorithm. This approach to training an RNN is often called *Backpropagation through time* or BPTT [**Wer90**].

Note that the run time for the update is $O(N)$ where N is the length of the input sequence. This, unfortunately, cannot be reduced through parallelization because of the sequential nature of the forward computation graph. Each time step can only be computed after the previous one was computed. As a result the state computed in the forward pass must be stored until it can be used in the backward training pass and so the memory cost of training an RNN is $O(N)$. This means that RNN's will require more computational resources (memory) for training than a similarly sized sequential feedforward network.

2.1. LSTM Recurrent Networks. Long Short Term memory networks (LSTM) [**HS97**] are a special kind of RNN that have been refined and popularized. LSTMs were routinely used in time-series prediction and language translation prior to 2017. They are still widely used so it is important to describe their structure.

Let us consider a simple RNN whose output is computed as

$$\mathbf{h}^{(k)} = \tanh\left(\mathbf{U}\mathbf{x}^{(k)} + \mathbf{W}\mathbf{h}^{(k-1)} + b\right). \tag{36}$$

In this case we assume that the hidden layer's activation, $\mathbf{s}^{(k)}$ is the model's output, $\mathbf{h}^{(k-1)}$. When we unfold this part of the forward computation graph we can the structure shown in Fig. 6 where the computation in equation (36) is represented by the layer labeled $\tanh$.

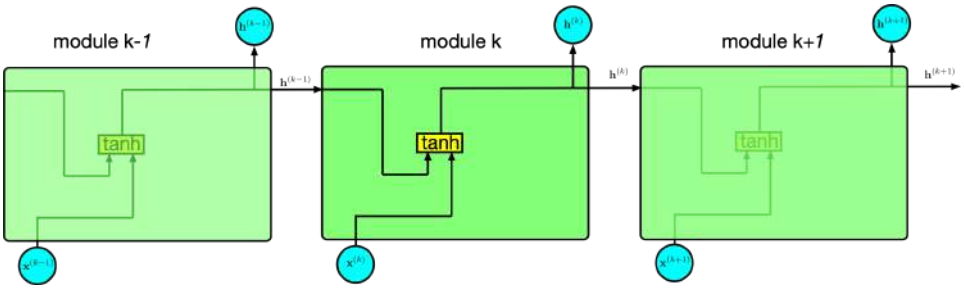


FIGURE 6. Simple RNN unfolded

LSTMs also have the chain like structure shown in Fig. 6. The difference is that the repeating module is more complicated. Instead of having a single neural network layer ($\tanh$), there are four neural network layers that interact in the ways shown in the Fig. 7. In this figure, each line carries an entire vector from the output of one node to the inputs of others. The circles with $+$ or $\times$ denote point-wise operations like vector addition and multiplication, respectively. In this figure the yellow boxes represent distinct neural network layers with either sigmoidal activation (i.e., σ) or hyperbolic tangent ($\tanh$) activation.

The key innovation in the LSTM is the idea of a *carry* state (also sometimes called a "cell" state). The carry state is information carried by the horizontal line running across the top of the unfolded LSTM in Fig. 7. The carry line runs across the top of the entire chain, with only some linear interactions that modulate the information in that line. Each LSTM module can remove or add information to the carry state through carefully regulated structures called *gates*. As shown in Fig 7, the gates are composed

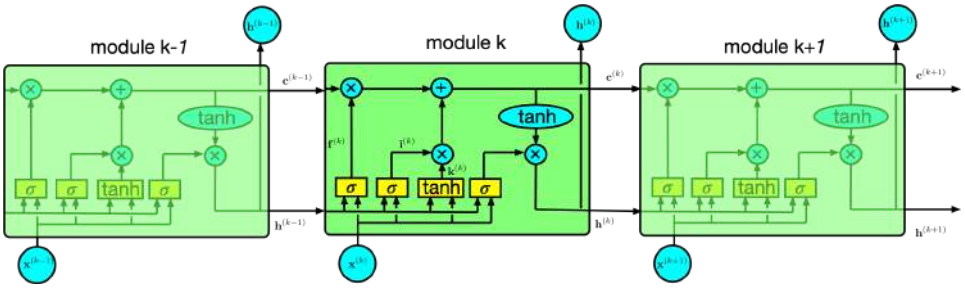
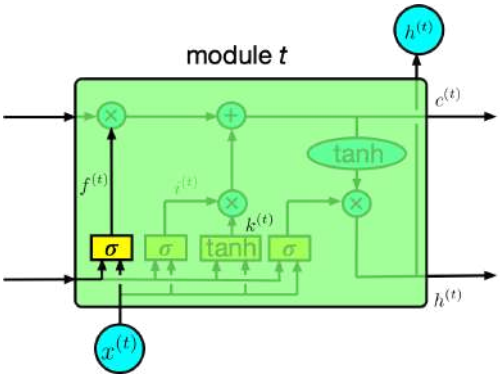


FIGURE 7. Unfolded LSTM

of sigmoid neural net layers and point-wise multiplication operations. The sigmoid layer outputs numbers in $[0, 1]$ indicating how much of the prior $(k-1)$ carry state should be transferred to the current (k) carry state. The point-wise operations either augment or delete information from the module's carry state. An LSTM has three such gates that we refer to as the *forget*, *input*, and *update* gates.



forget gate

$$f^{(t)} = \sigma\left(\mathbf{W}_f \odot \left[h^{(t-1)}, x^{(t)}\right] + b_f\right)$$

FIGURE 8. Forget Gate

The first decision to be made by an LSTM module is to decide what information that module will discard from the carry line. This decision is made by a sigmoid layer called the *forget gate layer* shown in Fig. 8. The layer looks at the output $h^{(t-1)}$ from module $t-1$ and the input, $x^{(t)}$, to module t. The forget layer then outputs a number between 0 and 1 for each component of the cell state vector, c^{t-1} carried over from module $t-1$. A 1 means keep "all" of the information in the cell state and 0 means forget the

cell state completely. Fig. 8 show that this forget signal, $f^{(t)}$ is computed as

$$f^{(t)} = \sigma(\mathbf{W}_f \odot \left[h^{(t-1)}, x^{(t)}\right] + b_f) \tag{37}$$

where $\mathbf{W}_f$ and b_f are the weighting matrix and bias for the forget gate layer and $\odot$ is the usual tensor product.

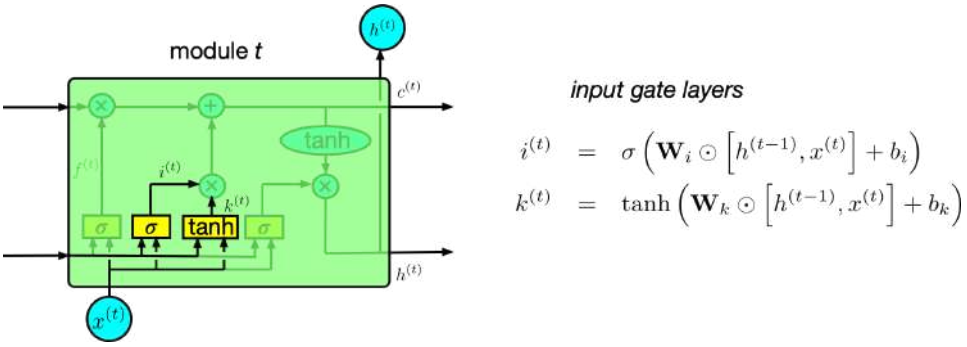


FIGURE 9. Input Gate

The next gate is the *input gate*. It decides what new information to add into the module's carry state, $c^{(t)}$. The input gate has two layers; a sigmoid layer whose output $i^{(t)}$ may be thought of as a *gain* that decides how strongly to update each component of the module's carry state. The actual information, $k^{(t)}$, used to update that component is determined by the $\tanh$ layer in Fig. 9. This means that the outputs of the two layers for the input gate can be written as

$$i^{(t)} = \sigma\left(\mathbf{W}_i \odot \left[h^{(t-1)}, x^{(t)}\right] + b_i\right) \tag{38}$$

$$k^{(t)} = \tanh\left(\mathbf{W}_k \odot \left[h^{(t-1)}, x^{(t)}\right] + b_k\right) \tag{39}$$

where $\mathbf{W}_i, b_i$ and $\mathbf{W}_k, b_k$ are the parameters for the two layers.

It is now time to update the old cell state, $c^{(t-1)}$ into the new cell state, $c^{(t)}$. We call this the *update layer* and it is shown in Fig. 10. The previous steps already decided what to do, we just need to realize those decisions. This is done by multiplying the old state by $f^{(t)}$ (forget gate activation) to forget those things we decided to forget earlier. We then add $i^{(t)} \times k^{(t)}$ to the carry state. This new candidate value, $k^{(t)}$, scaled by $i^{(t)}$ to determine how

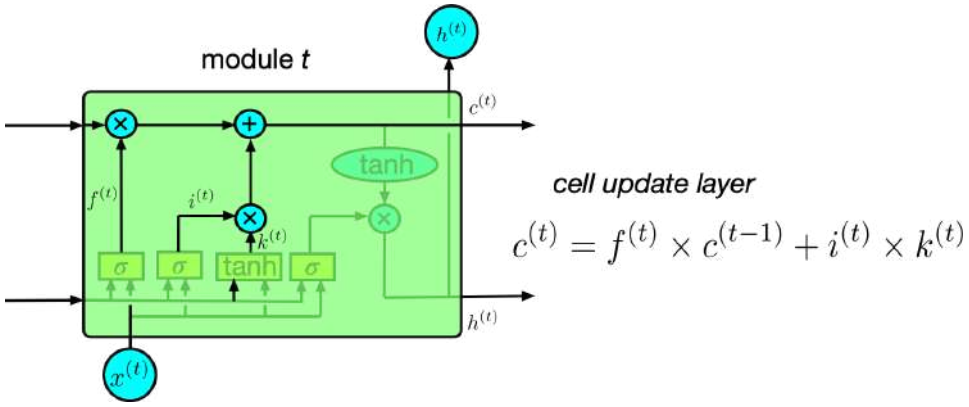


FIGURE 10. Update Gate

much we want to add to the module's carry state. The associated update equation as shown in Fig. 10 is therefore

$$c^{(t)} = f^{(t)} \times c^{(t-1)} + i^{(t)} \times k^{(t)} \tag{40}$$

where the binary operations $+$ and $\times$ are component-wise operations of addition and multiplication, respectively.

The preceding discussion is a "heuristic" explanation of how the LSTM gating process works. This particular interpretation where the module learns to "forget" and "add" information to the carry state provides a convenient way to understand how long-term dependencies in the time series or sequence can be remembered.

Keras has implemented an LSTM layer . That implementation is then trained using a `GradientTape` object. This layer was used to build the two layer LSTM shown in Fig. 3. Stacking LSTM layers is a commonly used way to avoid overfitting. Google's original language translation app (Google Translate) used a stack of seven large LSTMs [**Cho21**]. Stacking recurrent layers on top of each others requires that the intermediate layers return the full sequence of outputs, rather than their output at the last time step. As shown in Fig. 3, Keras LSTM layer does this through the control parameter `return_sequences = True`.

There are two useful variations on the LSTM that are frequently used; gated recurrent units (GRU) and bidirectional LSTMs. A GRU [**CGCB14, CVMG+14**] may be seen as a simplified version an LSTM whose operation can again be explained in terms of "gating layers". Keras also has a GRU layer whose API is similar to that of the LSTM layer.

The second type of LSTM is a *bidirectional RNN*. Note that simple RNN's and LSTM process inputs in an order-dependent manner. In other words, these networks are "causal" in the sense that the future outputs are determined solely by the past outputs. There are problems, however, where the future inputs also influence earlier outputs. This is particularly true in text and language processing applications [**GFS05**]. A bidirectional RNN uses two regular RNNs, one of which processes inputs in the forward (causal) direction and the other processing inputs in the reverse (anti-causal) direction. The layer then merges the outputs of these two layers. Keras has a bidirectional layer that that is cascaded with an RNN layer. So we could declare such a model as follows, to show how you would use it in your scripts.

```
imputs = keras.Input(shape = (sequence_length, raw_data.shape[-1]))
x = layers.Birdirectional(layers.LSTM(16))(inputs)
outputs=layers.Dense(1)(x)
model = keras.Model(inputs,outputs)
```

3. Natural Langage Processing

Human languages are *natural languages*. This stands in contrast to the *formal languages* used to program computers. A formal language is a *complete* system in the sense that all expressions in the language satisfy a rigid set of grammatical rules. Natural languages emerge in an evolutionary manner, changing over time as the way people use language changes. The grammatical rules in a natural language are often bent or broken in a manner that would confuse a machine. This freedom of expression is extremely difficult to capture in a formal language, but neural networks have the capacity to

capture such ”exceptions to the rule” that formal methods find difficult to accommodate.

Deep learning does not seek to ”understand” natural language in the way a human would. Deep learning is used to develop models that accept a fragment (sentence or phrase) of a natural language as input and return something useful. The main tasks in natural language processing (NLP) are

- *Text Classification:* determine the topic of a text sentence
- *Content Filtering:* classify a phrase as “abusive” and remove it from the sentence.
- *Sentiment Analysis:* classify whether the text is making a “positive” or “negative” statement about a topic.
- *Neural Machine Translation (NMT):* translate a phrase in one language to a phrase in a different language.

Classical NLP toolsets were based on decision trees and logistic regression. But these tools saw slow advances between 1990-2010. In early 2015, Keras made available the first open source LSTM layer and its introduction started a massive wave of interest in RNNs. From 2015-2017, RNNs were the primary tool used for NLP tasks. From 2017-2018, a new architecture begin to replace RNNs; the *transformer* [**VSP+17**]. Today that transformer plays a key role in popular applications such as ChatGPT [**Ope22**]. GPT stands for *generative pre-trained transformer* which is a pre-trained generative deep learning model based on the transformer. This section takes a closer look at deep learning for natural language processing, in particular for sentiment analysis. The first major thing we need to do is transform the text string into a numerical tensor that can be used by the neural network. This task is known as *text vectorization*.

3.1. Text Vectorization. Deep learning models only process numeric tensors. So to use neural networks for NLP, we must first transform the text

data into a numerical tensor. This is sometimes called *text vectorization* and it follows the following sequence of operations

(1) *Text Standardization:* standardize the text to make it easier to process by transforming all letters to lower-case and removing punctuation.
(2) *Tokenization:* split the text into units called *tokens*. A token may be the characters in a word, individual words, or a group or consecutive words (phrase).
(3) *Vocabulary Indexing:* convert the tokens into a numerical tensor. This is often done by *indexing* all tokens present in the data; namely associating a number or tensor to each token present in the data. These numbers/tensors are determined with respect to a given *vocabulary* dictionary.

We will now discuss each or these steps in more detail.

Text Standardization: Consider these two sentences:

”sunset came, i was staring at the Mexico sky, Isnt nature spendid?”
”Sunset came, I stared at the Mexican sky, Isn’t nature splendid?”

A human reading both sentences would have little difficulty in understanding what these sentences are trying to say. A computer, however, might find these sentences difficult to comprehend because of the slight differences in punctuation and wording that don’t follow the grammatical rules.

Text standardization is a form of feature engineering that erases these encoding differences. The simplest and most widespread standardization scheme is to convert all letters to lower case and remove punctuation. This would convert the two sentences to

”sunset came i was staring at the mexico sky isnt nature splendid”
”sunset came i stared at the mexican sky isnt nature splendid”

These two sentences are clearly more similar to each other so that when they are later transformed into numerical vectors, the ”distance” between them is much smaller.

Tokenization:. Once the text has been standardized, it must be broken up into units (tokens) that can later be vectorized. Tokenization is done in at least three different ways:

- *Word-level tokenization* is where tokens are space-separated substrings. A variant would split words into subwords when applicable such as tokenizing ”staring” into ”star+ing”.
- N*-gram tokenization* has tokens that are groups of N consecutive words. For instance, ”the cat” or ”he was” would be 2-grams.
- *Character-level tokenization* associates each character with a token. This approach is rarely used.

In general, one either uses word-level or N-tokenization. Which approach you use usually depends upon the type of text-processing model you are using. There are two types of text-processing models; *sequence models* and *bag-of-words models*. The sequence model treats the input as a sequence of tokens. The bag-of-words model treats the input as a set of tokens, discarding their original order. If you are building a sequence model, then you use word-level tokenization. If you use a bag-of-words model you use N-gram tokenization.

Vocabulary Indexing:. Once the text has been split into tokens, one needs to encode each token into a numerical representation. In practice, one does this by building an index of all terms found in the training dataset (this is called the *vocabulary*) and then assigns a unique integer to each term in the vocabulary. The following Python script shows how this might be done.

```
vocabulary = {}
for text in dataset:
     text = standardize(text)
     tokens = tokenize(text)
```

```
    for token in tokens:
        if token not in vocabulary:
            vocabuary[token] = len(vocabulary)
```

One then converts that integer in the dictionary `vocabulary` into a one-hot vector

```
def one_hot_encode_token(token):
    vector = np.zeros((len(vocabulary),))
    token_index = vocabulary(token)
    vector[token_index] = 1
```

In this approach it is common to restrict the dictionary `vocabulary` to the top 10,000 or 20,000 most common words in the training dataset.

An important detail in vocabulary indexing is to provide a way to handle tokens that were not in the training dataset. To handle this exception, one would also add an "out of vocabulary" index that we abbreviate as [UNK]. In general, one encodes [UNK] as 1 (i.e. the one-hot vector with the first element set to 1). The numerical index 0 (i.e. the one-hot vector of all zeros) is reserved for a "mask token" that we treat as a "blank" or "empty word" space. Note that the inputs generated by vocabulary indexing all have the same dimension. The `TextVectorization` layer in Keras encapsulates the preceding Python code.

```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization( output_mode = "int",)
```

By default, the `TextVectorization` layer converts to lower-case and removes punctuation for standardization. This layer "splits on white-space" for tokenization. The layer provides controls that allow you to customize the standardization and tokenization methods.

We index the vocabulary of a given set of text by calling the `adapt()` method of the layer. The input to this method is a `Dataset` object that contains the sentences we use in building the vocabulary dictionary. The input to the `adapt()` method can also just be a list of Python strings. You

can retrieve a computed vocabulary via the `get_vocabulary()` method. This is useful if you need to convert text encoded as integers back into words. The first two entries in the vocabulary are the mask token and ”out-of-vocabulary”, [UNK], token described above. Entries in the vocabulary list are sorted by frequency so that common words like ”the” and ”a” would come first. The following example illustrates the use of the TextVectorization layer in encoding a sentence and then decoding the resulting vector

```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization( output_mode = "int",)

#create a vocabulary for the dataset
dataset = [
      " I write, erase, rewrite",
      "Erase again, and then",
      "A poppy blooms.",]
text_vectorization.adapt(dataset)

#Encode a test sentence using the vocabulary
vocabulary = text_vectorization.get_vocabulary()
test_sentence = "I write, rewrite, and still rewrite again"
encoded_sentence = text_vectorization(test_sentence)
print(encoded_sentence)

#tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)

#decode the sentence using the vocabulary
inverse_vocab = dict(enumerate(vocabulary))
decoded_sentence = " ". join(inverse_vocab[int(i)] for i in encoded_sentence)
print(decoded_sentence)

#i write rewrite and [UNK] rewrite again
```

The `TextVectorization` layer in Keras encapsulates the preceding Python code. This layer performs all three tasks of Text Standardization, Tokenization, and Vocabulary Indexing. The `TextVectorization` layer performs *Text Standardization* by convert the input string to lower case and removes punctuation. The layer also provides controls for *Tokenization*. In

particular, we can specify whether we want to tokenize a single word or a sequence of N consecutive words (i.e. N-gram).

Vocabulary Indexing is done by called the layer's `adapt()` method. The input to the `adapt()` method is a `Dataset` object or a list of Python strings containing the sentences used in building the vocabulary dictionary. You can retrieve the vocabulary index via the `get_vocabulary()` method. This method is useful if one needs to convert an integer encoded text string back into words (tokens). Note that the first two vocabulary entries are the mask token and "out-of-vocabulary" ([UNK]) token described above. The remaining entries in the vocabulary are sorted by frequency so that common words like "the" and "a" would come first. The following script illustrates the use of the TextVectorization layer to encode word tokens as integers and then decodes the resulting vector.

```
from tensorflow.keras.layers import TextVectorization
#instantiate encoder
text_encoder = TextVectorization(output_mode = "int")

#create vocabulary index from list of Python strings
dataset = [
     "I write, erase, rewrite",
     "Erase again, and then",
     "A poppy blooms."]
text_vectorization.adapt(dataset)

#Encode a test sentence
vocabulary = text_vectorization.get_vocabulary()
test_sentence = "I write, rewrite, and still rewrite again"
encoded_sentence = text_encoder(test_sentence)
print(encoded_sentence)
#OUTPUT: tf.Tensor([ 6  2  4 10  1  4 11], shape=(7,), dtype=int64)

#decode encoded test sentence
inverse_vocab = dict(enumerate(vocabulary))
decoded_sentence = " ".join(inverse_vo ab[int(i)] for i in encoded_sentence)
print(decoded_sentence)
#OUTPUT: i write rewrite and [UNK] rewrite again
```

4. Bag-of-Words vs Sequence Models

There are, in general, two types of NLP models: Bag-of-Words or Sequences. Bag-of-Word models treat the encoded input as a *set* of tokens and ignores the ordering of those tokens in the original input. Sequence models treat the encoded input as a *sequence* of inputs where the order of the tokens plays an important role in the task.

4.1. Bag-of-Words Modeling: This section examines the bag-of-words modeling approach on the IMDB movie review sentiment analysis problem introduced in section 4 of chapter 4. Recall that the IMDB database contains 50,000 highly polarized movie reviews with a 50% split between training and testing data. In our earlier example, we imported the IMDB dataset from TensorFlow and the utility we used encoded the words in the review as integers. In this subsection we want to illustrate how we use the `TextVectorization` layer to generate such encodings, so we start from the "raw" dataset where the input samples have not been already encoded as integers.

We assume the raw IMDB dataset has already been downloaded from Stanford's AI lab as a compressed tar file.

```
!curl -O https://ai.stanford.edu/~amaasd/data/sentiment/aclImdb_v1.tar.gz
```

The uncompressed tar file consists of a directory `aclImdb` with two subdirectories `train` and `text` each with subsubdirectories `pos` and `neg` that contain `txt` files with the reviews. The following script changes that file structure by creating a validation, `val`, and a p-training, `ptrain`, subdirectory. The script randomly splits the `train` sentences between the `val` and `ptrain` subdirectories with a 20% validation split. We then use a TensorFlow utility to generate the p-training, validation, and test datasets

directly from these subdirectories. For these created dataset objects, the inputs are now TensorFlow `tf.string` tensors and the targets are `int32` tensors encoding values of 0 or 1.

```
import os, pathlib, shutil, random
from tensorflow import keras
batch_size = 32
nseed = 1337
val_split = 0.2

base_dir = pathlib.Path("datasets/aclImdb")
train_dir = base_dir / "train"
val_dir = base_dir / "val"
ptrain_dir = base_dir / "ptrain"

#copy train files to ptrain and val subdirectories
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    os.makedirs(ptrain_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(nseed).shuffle(files)
    num_val_samples = int(val_split*len(files))
    val_files = files[ -num_val_samples:]
    ptrain_files = files[num_val_samples:]
    for fname in val_files:
        shutil.copyfile(train_dir / category / fname,
                        val_dir   / category / fname)
    for fname in ptrain_files:
        shutil.copyfile(train_dir / category / fname,
                        ptrain_dir / category / fname)

#create dataset objects
ptrain_ds = keras.utils.text_dataset_from_directory(
    ptrain_dir, batch_size = batch_size)
val_ds    = keras.utils.text_dataset_from_directory(
    val_dir, batch_size = batch_size)
test_ds   = keras.utils.text_dataset_from_directory(
    test_dir, batch_size = batch_size)
```

The bag-of-words approach treats the tokens for a piece of text as unordered text. For sentiment analysis this simply means we look to see if

the sentence contains those "negative" words that we usually see in a negative review. We will use the `TextVectorization` layer to tokenize single words or pairs of words and then to encode these tokens as *multi-hot encoded* binary word vectors. This means we will first assign an integer vocabulary index to the 20,000 most frequent words in the dataset. But rather than encoding each sentence as a sequence of these vocabulary indices, we encode it as a $20,000$ length binary vector whose ith entry is 1 if the word with index i appears in the sentence and is 0 otherwise. We use multi-hot encodings because our neural network model needs a constant length input and the length of sentences in the database is variable. The following script takes the original dataset objects whose inputs were text strings and creates new dataset objects whose inputs are the multi-hot encoded versions of those strings.

```
from tensorflow.keras.layers import TextVectorization
text_encoder = TextVectorization(
    ngrams = 1,    #2-gram tokenization
    max_tokens = 20000,   #vocabulary index length 10000
    output_mode = "multi_hot"
)

#create vocabulary from training datasets
training_input_text = train_ds.map(lambda x, y:x)
text_encoder.adapt(training_input_text)

#create datasets whose inputs are multi-hot encoded
ptrain_multihot_ds = ptrain_ds.map(
    lambda x, y: (text_encoder(x), y),
    num_parallel_calls = 4)
val_multihot_ds   = val_ds.map(
    lambda x, y: (text_encoder(x), y),
    num_parallel_calls = 4)
test_multihot_ds  = test_ds.map(
    lambda x, y: (text_encoder(x), y),
    num_parallel_calls = 4)
```

We now build a dense sequential model with a single hidden layer of 32 nodes (relu activation) that feeds to an output layer with two nodes, one for each class (`pos` or `neg`). We will train it with an Adam optimizer and an

L2 regularizer. Because we have two outputs that estimate the likelihood of the input tensor being `pos` or `neg`, we use a sparse categorical crossentropy loss function. This gives us a model with about 640,000 weights.

```
def build_model(num_inputs, num_nodes, regularizer):
     inputs = keras.Input(shape = (num_inputs,))
     x = layers.Dense(num_nodes, activation = "relu",
                         kernel_regularizer = regularizer)(inputs)
     outputs = layers.Dense(2, activation = "sigmoid")(x)
     model = keras.Model(inputs=inputs, outputs = outputs)
     return model

num_inputs = num_words #20000
num_nodes = 32

lam = 0.005
regularizer = tf.regularizers.l2(lam)
model = build_model(num_inputs, num_nodes, regularizer)

learning_rate = 0.001
optimizer = tf.optimizers.Adam(learning_rate)
model.compile(
      optimizer = optimizer,
      loss = "sparse_categorical_crossentropy",
      metrics = ["accuracy"])
model_summary()
```

We will train this model using the multi-hot encoded inputs for 10 epochs and use a callback that saves the model with the smallest validation loss. We then take the training `history` and plot the training curves. We reload the best model saved during training and evaluate the accuracy of that model on the testing dataset. The plotted training curve is shown on left pane of Fig. 11 where the best model's test accuracy was 87%. In looking at the curve, however, we note that there is little variation in the loss. This suggests that in spite of the good accuracy level, this model is not really learning much as we train it since the loss doesn't go down very much.

```
callbacks = [
     keras.callbacks.ModelCheckpoint(
            filepath = "models/bag-of-words-imdb-model.keras",
```

```
            save_best_only = True,
            verbose = 1,
            monitor = "val_loss")]
num_epochs = 10
history = model.fit(ptrain_multihot_ds, epochs = num_epochs,
       validation_data = val_multihot_ds,
       callbacks = callbacks)
best_model = keras.models.load_model("models/bag-of-words-imdb-model.keras")
ptrain_loss = history.history["loss"]
val_loss = history.history["val_loss"]
fname = "fig/bag-of-words-imdb-model.png"
plot_training_curves(ptrain_loss, val_loss, best_test_acc,fname)
```

There are two useful variations on the preceding bag-of-words approach where we encode additional information into the input tensor. One approach simply keeps track of how many times a given token appeared in the sentence. The following `TextVectorization` layer modifies the multi-hot encoded inputs so the ith component equals the number of times the ith token appeared in the text.

```
text_encoder = TextVectorization(
     ngrams = 1,
     max_tokens = 20000,
     output_mode = "count")
```

Note that some words occur more often no matter what text fragment we are working with. Examples of such highly occurring words are the articles ”a” and ”the”. The high occurrence of these works has little relevance to sentiment analysis. We can reduce the importance of these words by normalizing their word count with respect to the number of times the token appears in the entire dataset. This is called *term frequency-inverse document frequency* or TF-IDF encoding. We can also use the `TextVectorization` layer to generate TD-IDF multi-hot encodings by simply specifying the output mode as `tf_idf`.

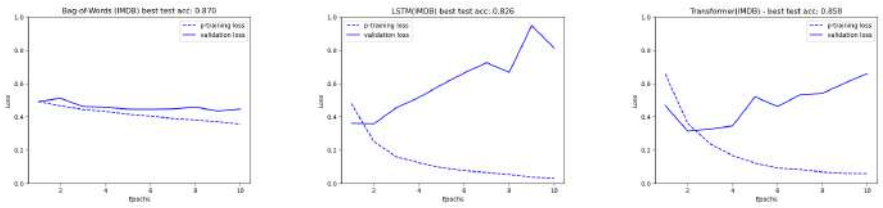


FIGURE 11. Training results on IMDB sentiment analysis problem for three different models: (left) bag-of-words dense sequential model, (center) LSTM model, (right) Transformer model.

4.2. Sequence Modeling: Word order can matter in text classification problems. This is why bigrams might be preferred to 1-grams in a bag-of-words approach. To some extent, N-gram encoding may be viewed as a form a "feature engineering". Modern machine learning practice seeks to have the model "learn" what those "features" are and this is what is done in the *sequence* modeling approach, where the model architecture is chosen to process sequentially ordered tokens. In particular, we will examine two different sequential models; LSTMs and Transformers. LSTMs are RNNs that process the vectors in a sequence one vector at a time. Transformers are models that take the entire sequence as an input. We will also "learn" how to map tokens to vectors, rather than simply specifying these vectors using multi-hot encoding. This is an example of *representational learning* where we train our model to "learn" how to represent the inputs in a lower dimensional "latent space". When this approach is applied to NLP, we refer to it as *word embedding*. The following subsection confines its attention to an LSTM sequential model. The transformer model will be discussed later in this chapter.

To develop our sequence model, we first start by creating a layer that learns how to map integer encoded inputs onto lower dimensional real-valued embedding vectors. We would then feed this sequence of embedding

vectors into a stack of neural network layers. These stacks can be formed from CNN's or RNNs. We will use a bidirectional LSTM below.

Word embeddings may be seen as establishing a topological structure on the word tokens that reflects how these words are related back to the classification task. The metric relationship means that two words whose "embedding vectors" are "close" in the metric space and therefore play a similar role in determining the sentiment of the sentence. Another way of thinking about this word vector space is that two words whose vectors are close together are semantically similar to each other. A word embedding, therefore, is a vector representation of words that maps human language onto a normed vector space. Whereas the vectors obtained through one-hot or multi-hot encoding are binary, sparse, and very high dimensional, word embeddings are low-dimensional floating point vectors.

Fig. 12 shows how four words might be embedded onto a 2D plane: cat, dog, wolf, and tiger. The vector representation chosen here reflects an assumed semantic relationship between these words. Along the x-axis we measure how "feline" like the word is. So "Tiger" and "cat" have large x-values (close to 1) along this axis, whereas "Wolf" and "Dog" have low x-values. Along the y-axis we measure how "wild" the word is. So along the y-axis, "Wolf" and "Tiger" have high y-values and "Dog" and "Cat" have low y-values.

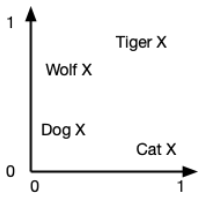


FIGURE 12. Word Embedding

Word embedding are obtained in two basic ways

- Learn the word embeddings jointly with the main task you care about. In this case you start with random word vectors and then learn word vectors the same way you would learn the weights of a neural network.
- Load pre-trained word embeddings into your model.

We first look at how one might ”learn” a word embedding. This is done using backpropagation and Keras provides an *embedding layer* for this purpose.

The `Embedding` layer may be seen as a dictionary that maps integer indices (which stand for specific words) to dense real-valued vectors. This layer takes integers as inputs, looks up these integers in an internal dictionary, and returns the associated vectors. When you instantiate an `Embedding` layer, its weights are initially random. During training these word vectors are gradually adjusted through backpropagation to structure the space into something that downstream layers can exploit in solving the task. Once fully trained, the embedding space shows a great deal of the semantic structure for the specific problem you are working on. The following example shows how to use the embedding layer in the IMDB sentiment analysis task. In this case the embedding layer feeds a Bidirectional LSTM with 32 nodes. We include a dropout layer to handle overfitting and a dense layer with 2 nodes that estimate the probability of the text input having positive or negative sentiment. The following script instantiates our model with an embedding layer generating vectors with an embedding dimension of 32. We also add dropout to help regularize the model.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers

def build_model(num_tokens,num_nodes,embedding_dim):
    inputs = keras.Input(shape = (None,),dtype = "int64")
    #embedded = tf.one_hot(inputs, depth=num_tokens)
    embedded = layers.Embedding(input_dim=num_tokens,
                            output_dim = embedding_dim)(inputs)
    x = layers.Bidirectional(layers.LSTM(num_nodes))(embedded)
    x = layers.Dropout(0.7)(x)

    outputs = layers.Dense(2, activation = "sigmoid")(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

```
embedding_dim = 32
num_nodes = 32
model = build_model(max_tokens, num_nodes, embedding_dim)

learning_rate = 0.001
optimizer = optimizers.Adam(learning_rate)
model.compile(
    optimizer = optimizer,
    loss = "sparse_categorical_crossentropy",
    metrics = ["accuracy"])

model.summary()
```

This model has about the same number of weights $(650,000)$ as our earlier sequential bag-of-words model. The difference, however, lies in how these weights are used. We train our model using an Adam optimizer and save the model with the best validation loss. We train the model for 10 epochs and then plot the training curves and compute the best model's test accuracy.

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="models/lstm-imdb-model.keras",
        save_best_only = True,
        verbose = 1,
        monitor="val_loss"
    )
]

num_epochs = 10
history = model.fit(ptrain_int_ds, epochs=num_epochs,
             validation_data = val_int_ds,
             callbacks = callbacks)

test_model = keras.models.load_model("models/lstm-imdb-model.keras")
best_test_loss, best_test_acc = test_model.evaluate(test_int_ds)

ptrain_loss = history.history["loss"]
val_loss    = history.history["val_loss"]
fname = "fig/lstm-imdb-model.png"
```

```
plot_training_curves(ptrain_loss, val_loss, best_test_acc,fname)
```

The training curve is shown in the middle pane of Fig. 11. The curve shows that the model is now learning something since the training loss decreases much more than it did for the bag-of-words model. We also see that it begins overfitting relatively early and that the best model's test accuracy was about 82%. This is less than our bag-of-words model, but it is still relatively good.

Sometimes there is so little training data that you cannot use your data alone to learn an appropriate word embedding of your vocabulary. In such cases, one can load embedding vectors from a previously learned embedding space. This is similar to using pre-trained networks in computer vision and then training the remaining top layers for the specific application at hand. There are various pre-trained word embeddings that you can download and use in a Keras `embedding` layer. Two of these are Word2vec [**(Go**] and GloVe [**Sta**]. The following example uses GloVe embeddings in a Keras model. After downloading the GloVe from the website and unzipping it into the directory `glove`, we first index the embedding word vectors in the database

```
import numpy as np
path_to_glove_file = "glove/glove.6B.100d.txt"
#this is a dataset with embedding dimension 100

embeddings_index = {}
with open(path_to_glove_file) as f:
      for line in f:
           word, coefs = line.split(maxsplit=1)
           coefs = np.fromstring(coefs, "f", sep=" ")
           embeddings_index[word] = coefs
print(f"Found {len(embeddings_index)} word vectors.")
```

We then build an embedding matrix to be loaded into an `Embedding` layer. This matrix has dimensions `max_words` by `embedding_dim` where the ith entry contains the `embedding_dim`-dimensional vector for the word of index i in the reference word index.

```
embedding_dim = 100

vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
       if i < max_tokens:
              embedding_vector = embeddings_index.get(word)
       if embedding_vector is not None:
             embedding_matrix[i] = embedding_vector
```

We use a `Constant` initializer to load the pre-trained embeddings into an `Embedding` layer. We then freeze the embedding layer to not change the word embeddings we imported from the GloVe database. We then train our model just as we did before. In this case, using the pre-trained database does not help much due to the model architecture. As it turns out, even though LSTM's have a carry track to keep long-term dependences, this carry track may not be enough in the IMDB sentiment analysis task. In particular, when we look at the past benchmarks on the IMDB task, we see that the best accuracy values of 92-96% occurred for LSTM, CNN, and Transformer models that used special ways of handling embedding the inputs. Our relatively simple way of using the LSTMs (and later Transformer) only attained accuracy levels of 85-88%, which is consistent with what we are seeing in these examples here.

```
embedding_layer = layers.Embedding(
     max_tokens, embedding_dim,
     embeddings_initializer = keras.initializers.Constant(embedding_matrix),
     trainable=False,
     mask_zero=True,)
```

5. Neural Attention and the Transformer Model

The *neural attention* mechanism was introduced in 2014 [**BCB14, LPM15**]. It would provide the basis for a new sequence model architecture known as the *transformer* [**VSP+17**] that would eventually come to dominate NLP

tasks and be the core model used in today's large language models found in ChatGPT. This section introduces the concept of neural attention and shows how it can be used to form sequence model built from RNN and transformer layers.

5.1. Neural Attention: Neural attention refers to weighting specific parts of the input so the model pays more "attention" to those parts. This idea has already been seen in the max-pooling layers of CNNs and TF-IDF encoding schemes. Neural attention highlights or downplays specific parts of the input sequence. It can be used to make features *context sensitive*. Recall in a word embedding, a single word is a fixed vector whose relationship to other word vectors is determined by the basis we use for that vector space. A "smart" embedding space provides a different basis representation for a word depending on the words (tokens) surrounding it. This is what we call *self-attention*. The purpose of self-attention is to modulate the vector representation of a token by using the representations of related tokens in the sequence. This produces a context aware representation of the token. This modulation may be realized by multiplying each component of the input tensor by a weight called the *attention score*. In the case of self-attention, this attention score is, essentially, the autocorrelation of the sequence with itself.

Let us consider the sentence: "The train left the station on time". Now consider one word in the sentence: "station". What kind of station are we talking about? It could be a radio station or a space station. The self-attention mechanism clarifies the context in which "station" is embedded. The first step is to compute attention scores between the embedding vector for "station" and the embedding vector for every other word (token) in the sentence. We simply use the dot (vector) production of the two embedding vectors as our attention score since it is easy to compute. In practice, we also normalize these dot products with a scaling and softmax function to keep their values in the range from 0 to 1.

The second step is to compute the sum of all word vectors in the sentence, weighted by the attention score. Words closely related to ”station” will contribute more to the sum (including the word ”station” itself), while irrelevant words will contribute almost nothing to the sum. The resulting vector is our new representation for ”station”. Namely it is a vector representation that includes the corresponding context established by surrounding words. So in our example, this means that the new representation includes a part of the ”train” vector, clarifying the fact that the sentence is talking about a ”train station” rather than a ”space station”. This processing is shown graphically in Fig. 13, where the matrix of attention scores is shown by the 2-d matrix. We then extract the vector of scores for ”station” and add them together to obtain a context-aware vector representation for the word ”station”.

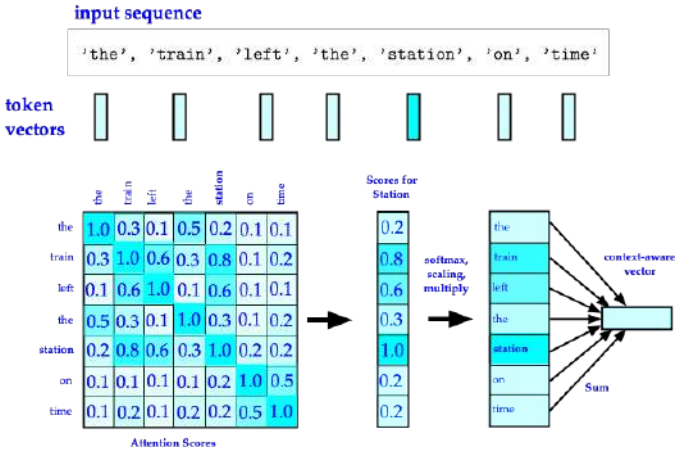


FIGURE 13. Self-attention: attention scores are computed between ”station” and every other word in the sentence, and they are then used to weight a sum of word vectors that becomes the ”new” context aware word vector for ”station”.

We repeat this algorithmic process for every word in the text fragment, producing a new sequence of vectors encoding the fragment. A NumPy pseudocode makes it apparent how this processing is actually done

```
def self_attention(input_sequnce):
```

```
    output = np.zeros(shape=input_sequence.shape)
    for i, pivot_vector in enumerate(input_sequence):
          scores = np.zeros(shape=(len(input_sequence),))
          for j, vector in enumerate(input_sequence):
                scores[j] = np.dot(pivot_vector,vector.T)
          scores /= np.sqrt((input_sequence.shape[1])
          scores = softmax(scores)
          new_pivot_representation = np.zeros(shape=pivot_vector.shape)
          for j, vector in enumerate(input_sequence):
                new_pivot_representation += vector*scores[j]
          output[i] = new_pivot_representation
    return output
```

Keras has a built-in layer to realize this self-attention mechanism. The Keras layer `MultiHeadAttention`, however, has some enhancements that are often used in practice. The following code shows how this layer would be instantiated.

```
num_heads = 4
embed_dim = 256

mha_layer = MultiHeadAttention(num_heads = num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

The are a couple of interesting aspects of this layer. The first concerns the fact that we passed three different tensors to the layer; in this case the same vector, `inputs`, three times. The second thing concerns what we mean by *multiple heads*. Let us discuss both of these items, first focusing on the reason why the layer takes three inputs.

In most of our prior layer models, we have only considered a single input. The transformer architecture, however, was originally developed for neural machine translation (NMT) where you have two sequences: the source sequence you want to translate (such as ”How is the weather today?”) and the target sequence you are converting it to (such as ”¿Qué tiempo hace hoy?”). A transformer is a sequence-to-sequence model that converts one sequence into another sequence. The self-attention mechanism performs this conversion as follows

```
outputs = sum(inputs * pairwise_scores(inputs, inputs))
```

This means "for each token in `inputs` (A), compute how much the token is related to every token in `inputs` (B), and use these scores to weight a sum of tokens from `inputs` (C)". Note that there is nothing that requires A, B, and C to refer to the same input sequence. In the general case, you could be doing this with three different sequences that we call the "query", "keys", and "values". The operation then becomes for every element in the query, compute how much the element is related to every key and use these scores to weight a sum of values.

```
outputs = sum(values * pairwise\_scores(query, keys)
```

This terminology comes from search engines and recommender systems. Imagine that you are typing up a query to retrieve a photo from your collection – "dogs on the beach". Internally, each of your pictures in the database is described by a set of keywords – "cat", "dog", "party", etc. We call these "keys". The search engine will start by comparing your query to the keys by strength of match - relevance - and it will return the pictures (values) associated with the top few matches in order of relevance.

Conceptually this is what Transformer-style attention is doing. You have a reference sequence that describes something you are looking for: the query. You have a body of knowledge that you are trying to extract information from: the values. Each value is assigned a key that describes the value in a format that can be readily compared to a query. You simply match the query to the keys and then return a weighted sum of the values.

In practice, the keys and the values are often the same sequence. In NMT, for instance, the query would be the target sequence and the source sequence would play the role of both keys and values: for each element of the target (like "tiempo"), you want to go back to the source ("How's

the weather today?") and identify bits that are related to it ("tiempo" and "weather" should have a strong match). So if you are just doing sequence classification (rather than NMT), then the query, keys and values are all the same: you are comparing the sequence to itself, to enrich each token with the context of the whole sentence. This is why the preceding instantiation of the attention layer passed `inputs` three times to the `MultiHeadAttention` layer.

We refer to this as a "multi-head" layer because we are generalizing the original self-attention mechanism introduced in [VSP+17]. The "multi-head" label refers to the fact that the output space of the self-attention layer gets factored into several independent subspaces, each learned separately. In other words the initial query, key, and value are sent through three independent sets of dense projections resulting in three separate vectors. Each vector is processed through neural attention and the three outputs are concatenated back together into a single output sequence. Each subspace is called a *head* as shown in Fig. 14.

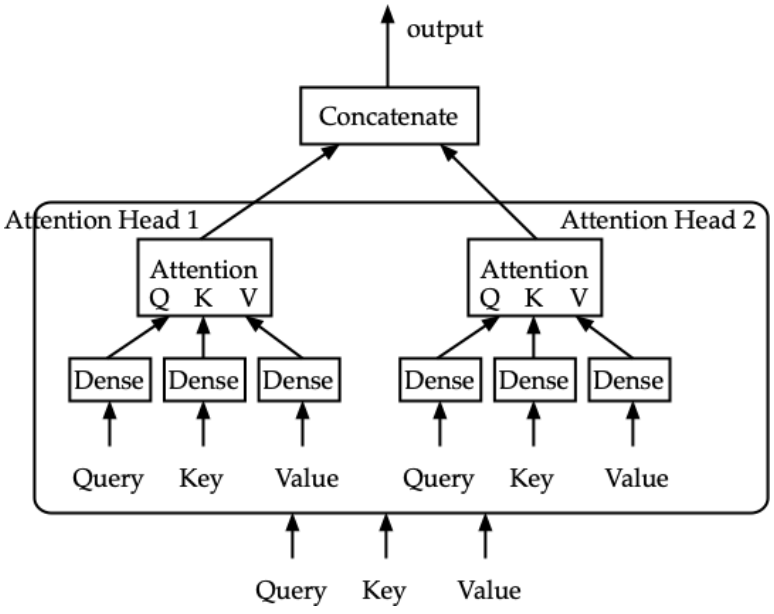


FIGURE 14. Multi-head Attention Layer

Note that this is similar to the way in which we compute different "channels" in the depth-wise convolutions found in CNNs. In that case, the output space of the convolution is factored into many subspaces (one per channel) that get learned independently. The "attention" paper [VSP+17] was written when the idea of factoring feature spaces into independent subspaces had been shown to provide a great benefits for computer vision models. The Multi-headed attention mechanism simply applies that principle to the self-attention concept.

5.2. Transformer Encoder: . The original transformer model consisted of two parts: an encoder and a decoder. The encoder takes a sequence (sentence) and maps it onto a *meaning* vector. The decoder expands that meaning vector back into a sentence. This encoder-decoder architecture works well for neural machine translation (NMT) where the input sequence is in one language and the output sequence is in a different language. To solve sentiment analysis or text classification, however, we only need to focus on the transformer encoder, which is what we do in this subsection.

The transformer encoder is based on a network of dense layers. It uses the MultiHeadAttention layer to preprocess the text vector inputs. The entire encoder also uses layer normalization and residual connections. LayerNormalization is different from the BatchNormalization used in CNNs. LayerNormalization normalizes each sequence independently from other sequences in the batch.

The following script shows how to use the `TransformerEncoder` layer in a model for the IMDB sentiment analysis task. We use this in much the same way we did the LSTM model. This means that we take training inputs that were integer encoded and pass them through an embedding layer before passing it to the Transformer. We will use custom TensorFlow layers. The first custom layer is the embedding layer `TokenAndPositionEmbedding`. Self-attention focuses on the relationships between pairs of sequence elements, so it is blind to whether those relationships occur at the beginning,

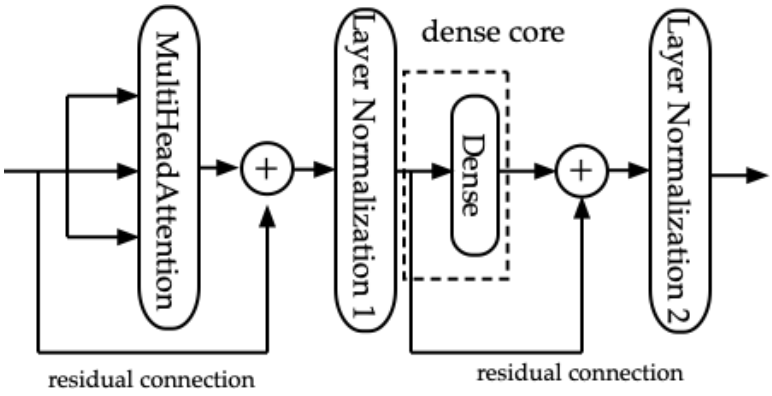


FIGURE 15. Transformer Encoder

middle, or end of the sentence. Positional embeddings inject information about a token's position by adding a *position* axis to the word vector. The following class definition creates our position embedding layer as a custom layer.

```
@tf.keras.utils.register_keras_serializable()
class TokenAndPositionEmbedding(layers.layer):
     def __init__(self, maxlen, vocab_size, embed_dim, **kwargs):
          super(TokenAndPositionEmbedding, self).__init__(**kwargs)
          self.vocab_size = vocab_size
          self.maxlen = maxlen
          self.embed_dim = embed_dim
          self.token_emb = layers.Embedding(input_dim=vocab_size,
                              output_dim=embed_dim)
          self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

     def call(self, x):
          maxlen = tf.shape(x)[-1]
          positions = tf.range(start=0,limit = maxlen, delta = 1)
          position = self.pos_mb(positions)
          x = self.token_emb(x)
          return x + positions

     def get_config(self):
          config = super().get_config()
          config['vocab_size'] = self.vocab_size
          config['maxlen'] = self.maxlen
          config['embed_dim'] = self.embed_dim
          return config
```

The `TransformerEncoder` is another custom block that we define in the following script. A block diagram for the Transformer encoder is shown in Fig. 15. This shows the encoder is passing the input through a MultiHeadAttention before passing it through a LayerNormalization layer. This then feeds to a dense sequential core, which in our case consists of a single dense layer. We use residual connections around the MultiHeadAttention layer and the dense core. The output of the dense core then goes through one more LayerNormalization layer.

```
@keras.utils.register_keras_serializable()
class TransformerEncoder(layers.Layers):
     def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1, **kwargs):
          super(TransformerEncoder, self).__init__(**kwargs)
          self.embed_dim = embed_dim
          self.num_heads = num_heads
          self.ff_dim = ff_dim
          self.rate = rate
          self.att = layers.MultiHeadAttention(num_heads = num_heads,
                                key_dim = embed_dim)
          self.ffn = keras.Sequential(
               [layers.Dense(ff_dim, activation="relu"),
                                    layers.Dense(embed_dim),]
          )
          self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
          self.layernorm2 = layers.LayerNormalization(epsilon=1.e-6)
          self.dropout1 = layers.Dropout(rate)
          self.droput2   = layers.Dropout(rate)

     def call(self, inputs, training):
          attn_output = self.att(inputs, outputs)
          attn_output = self.dropout1(attn_output, training = training)
          out1 = self.layernorm1(inputs + attn_outputs)
          ffn_output = self.ffn(out1)
          ffn_output = self.droput2(ffn_output, training=training)
          return self.layernorm2(out1 + ffn_output)

     def get_config(self):
          config = super().get_config()
          config['embed_dim'] = self.embed_dim
          config['num_heads'] = self.num_heads
          config['ff_dim']] = self.ff_dim
          config['rate'] = self.rate
```

```
        return config
```

Note that because this is a custom TensorFlow layer, we have to add a decorator at the start of each layer to register the object as a serializable object. This allows us to save the model and reload it later. We now use the Positional Embedding and Transformer Encoder layers in the same way we did for the LSTM layer

```
embed_dim = 32
num_heads = 2
ff_dim = 32

inputs = layers.Input(shape=(max_seq_length,))
embedding_layer = TokenAndPositionEmbedding(max_seq_length,
                              max_tokens, embed_dim)
x = embedding_layer(inputs)
transformer_encoder = TransformerEncoder(embed_dim, num_heads, ff_dim)
x = transformer_encoder(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Droput(0.7)(x)
x = layers.Dense(20, activation = "relu")(x)
x = layers.Dropout(0.7)(x)
outputs = layers.Dense(2, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs =outputs)

model.compile(
     optmiizer = "adam",
     loss = "sparse_categorical_crossentropy"),
     metrics = ["accuracy"])
```

The resulting model has about the same number of weights as the LSTM and sequential Bag-of-Words models $(650,000)$. So these are all models of comparable "size". We then train the model for 10 epochs and save the model with the best validation loss. We finish up by reloading the best model and plotting the training curves. The training curves are shown on the right side of Fig. 11. The transformer's training curve is similar to that of the LSTM with a best model testing accuracy of 86%.

```
callbacks = [
```

```
    keras.callbacks.ModelCheckpoint(
        filepath="models/transformer-imdb-model.keras",
        save_best_only=True,
        verbose = 1,
        monitor = "val_loss")]
history = model.fit(
    ptrain_int_ds,
    epochs = 10,
    validation_data = val_int_ds,
    callbacks = callbacks)

test_model = keras.models.load_model("models/transformer-imdb-model.keras")
best_test_loss, best_test_acc = test_model.evaluate(test_int_ds)
ptrain_loss = history.history["loss'"]
val_loss = history.history["val_loss"]
fname = "fig/transformer-imdb-model.png"
plot_training_curves(ptrain_loss, val_loss, best_test_acc,fname)
```

6. Sequence-to-sequence learning and Neural Machine Translation

Sequence-to-sequence (seq2seq) models [**SVL14**, **CVMG+14**] have enjoyed great success in NLP tasks such as machine translation, speech recognition, and text summarization. This section introduces seq2seq models and focuses on their use in Neural Machine Translation (NMT) of which one of the best examples is Google Translate[**Goo**]. Traditional phrase-based translation systems performed their task by breaking up the input sentence into multiple chunks (phrases) and then translating each chunk (phrase) one at a time. This usually leads to breaks in the translated sentence (what is referred to as *disfluency*). This is not, in general, how humans translate. Humans read the entire source sentence, understand its meaning, and then produce a translation based on that meaning. Modern NMT systems such as Google Translate mimic that approach to language translation.

In particular, a modern NMT system first reads the source sentence using an *encoder* to build a "thought" vector; a sequence of numbers that represents the sentence's meaning. It then passes this thought vector through a *decoder* that expands the vector into the translated phrase as shown in

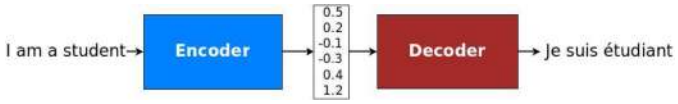


FIGURE 16. Encoder-decoder architecture - example of a general approach for NMT. The encoder converts a source sentence into a "meaning" vector which is passed through a decoder to produce a translation.

Fig. 16. This model is called the *encoder-decoder architecture*. In this manner, NMT addresses the local translation problem in traditional phrase-based approaches by capturing long-range dependencies in the language, e.g. gender agreements, syntax structures, etc and produces a much more fluent translation than was possible using older phrase-based translation systems.

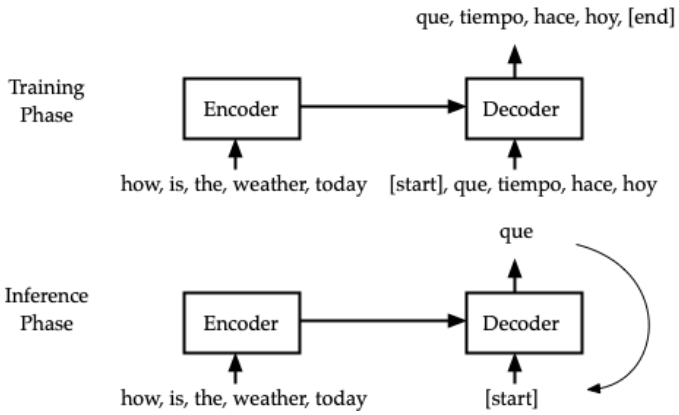


FIGURE 17. Sequence-to-sequence learning: the source sequence is processed by the encoder is then sent to the decoder. The decoder looks at the target sequence so far and predicts the target sequence offset by one step in the future. During inference, we generate one target token at a time and feed it back into the decoder.

The general template behind sequence-to-sequence models is illustrated in Fig: 17. During training

- The *encoder* model turns the source sequence into the ”thought” vector
- A *decoder* is then trained to predict the next token i in the target sequence by looking at both previous tokens (0 to $i-1$) in the target sequence and the entire encoded source sequence.

During inference, we cannot access the target sequence. We are trying to predict it in a recursive manner in which we generate it one token at a time.

- We first obtain the encoded source sequence from the encoder
- The decoder starts by looking at the encoded source sequence as well as an initial ”seek” token such as the string ”[START]”, and uses them to predict the first real token in the sequence.
- The predicted sequence is fed back into the decoder which generates the next token until we get the the ”[END]” token.

We now turn to examine how RNN's and Transformer models can be used to realize sequence-to-sequence learning on a neural machine translation example.

6.1. Neural Machine Translation. Neural machine translation (NMT) is an NLP task that takes a source sentence in one language and translates that sentence into another language. A good example is the website Google Translate [**Goo**]

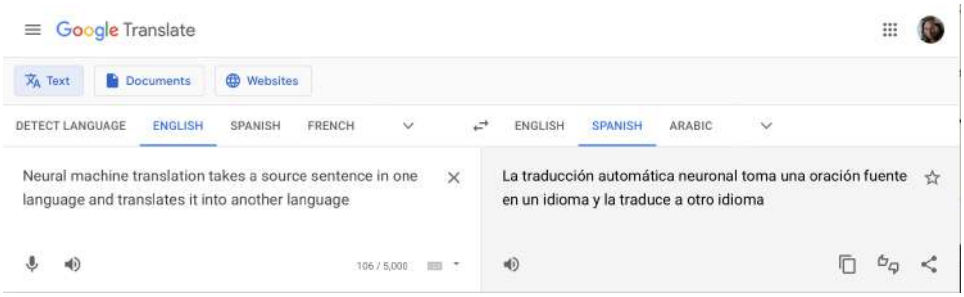


FIGURE 18. Screenshot of Google Translate Website

In order to build a model that does something similar to what Google Translate does, we first need to work with a translation dataset. In particular, we will use an English-to-Spanish dataset available at `www.manythings.org/anki`. The downloaded file will be placed in a subdirectory `spa-eng` and the file we need is `spa.txt`. This text file contains one example per line: an English sentence, followed by a tab character, followed by the corresponding Spanish sentence. Let us parse the file and then print a randomly chosen line from the file

```
text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))

import random
print(random.choice(text_pairs))

#output
#("I just don't want to talk to you.",
# '[start] Sencillamente no quiero hablar contigo. [end]')
```

Let us now shuffle the lines of this dataset and split them into the usual p-training, validation, and test sets. In particular, we reserve 15% of the sentence pairs for validation and 15% for testing.

```
import random
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

We are going to prepare two separate `TextVectorization` layers: one for English and one for Spanish. We will find it convenient to customize the way the sentences are preprocessed:

- We need to preserve the ”[start]” and ”[end]” tokens that we have inserted. By default the characters [and] would be stripped out.
- Punctuation is different in the Spanish language, since it uses inverted question marks. So we will want to strip such characters out.

```
import tensorflow as tf
import string
import re

from tensorflow.keras import layers

strip_chars = string.punctuation + "?"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")

vocab_size = 15000
sequence_length = 20

source_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
target_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
source_vectorization.adapt(train_english_texts)
target_vectorization.adapt(train_spanish_texts)
```

Finally we will turn our data into a `tf.data` pipeline. We want it to return a tuple `(inputs, targets)` where `inputs` is a dict with two

keys, "encoder_inputs" (the English sentence) and "decoder_inputs" (the Spanish sentence), and target is the Spanish sentence offset by one step ahead.

```
batch_size = 64

def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({
        "english": eng,
        "spanish": spa[:, :-1],
    }, spa[:, 1:])

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset, num_parallel_calls=4)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)

for inputs, targets in train_ds.take(1):
    print(f"inputs['english'].shape: {inputs['english'].shape}")
    print(f"inputs['spanish'].shape: {inputs['spanish'].shape}")
    print(f"targets.shape: {targets.shape}")

#inputs['english'].shape: (64, 20)
#inputs['spanish'].shape: (64, 20)
#targets.shape: (64, 20)
```

We will now use this data to first train an RNN sequence-to-sequence model.

6.2. RNN Sequence-to-Sequence Model. Recurrent neural networks dominated sequence-to-sequence learning from 2015-2017 before being displaced by the Transformer models. They were the basis for many real-world

translation systems, such as Google Translate circa 2017. It is still worth examining RNN sequence-to-sequence learning as an easy introduction to sequence-to-sequence learning and for providing a basis for understanding why the Transformer model came to overtake RNNs. The simplest way to use RNNs in sequence-to-sequence modeling is to keep the output of the RNN at each time step. In Keras, one might do this as follows

```
inputs = keras.Input(shape=(sequence_length,),dtype="int64")
x = layers.Embedding(input_dim=vocab_size, output_dim=128)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
outputs = layers.Dense(vocab_size, activation = "softmax")(x)
model = keras.Model(inputs, outputs)
```

There are, however, two major problems with this approach

- The target sequence must always be the same length as the source sequence, which is rarely true in practice. Technically, one could use padding to address this issue.
- Due to the step-by-step nature of RNNs, the model will only be looking a tokens $0, \ldots, N$ in the source sequence in order to predict token N in the target sequence. This constraint makes this step unsuitable for most tasks, and this is particularly true for translation. As we said before, translating a sentence usually requires one to read the entire source sentence to understand its meaning, before going ahead with the translation step.

So the proper way to do sequence-to-sequence translation using RNNs is shown in Fig. 19. In this case you would first use an RNN as an encoder to turn the entire source sequence into a single vector (or sequence of vectors). This vector could be the last output of the RNN or, alternatively, its final internal state vectors. You would then use this vector (or vectors) as the *initial state* of another RNN (i.e. the decoder) which would look at elements $0, \ldots, N$ in the target sequence to predict step $N + 1$ in the target sequence.

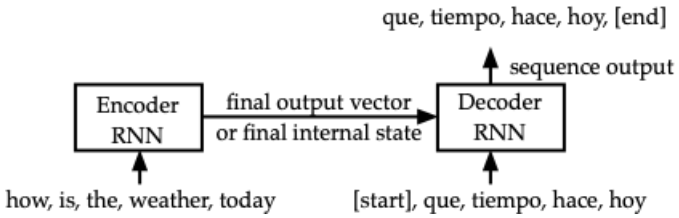


FIGURE 19. Sequence-to-sequence RNN: an RNN encoder is used to produce a vector that encodes the entire source sequence, which is used as the initial state for an RNN decoder.

We will implement this simple seq2seq-RNN model using a GRU for the encoder and decoder. This is done to make the model a bit simpler. The following code instantiates the GRU-encoder

```
embed_dim = 256
latent_dim = 1024

source = keras.Input(shape=(None,), dtype="int64", name="english")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
encoded_source = layers.Bidirectional(
    layers.GRU(latent_dim), merge_mode="sum")(x)
```

We now add the decoder as a simple GRU layer that takes its initial state the encoded source sentence. On top of it we add a `Dense` layer to produce for each output step a probability distribution over the Spanish dictionary.

```
past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target)
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
x = decoder_gru(x, initial_state=encoded_source)
x = layers.Dropout(0.5)(x)
target_next_step = layers.Dense(vocab_size, activation="softmax")(x)
seq2seq_rnn = keras.Model([source, past_target], target_next_step)
```

We now compile and train the model. During training the decoder takes as input the entire target sequence. But because of the recursive nature of the RNN, it only looks at tokens 0 to N in the input to predict token N in

the output which corresponds to the next token in the sequence since the output is offset by one step. This means we only use information from the past to predict the future, which makes sense when we use this model for inference.

```
seq2seq_rnn.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
seq2seq_rnn.fit(train_ds, epochs=15, validation_data=val_ds)
```

We used accuracy to monitor the validation performance during training. We get to 64% accuracy, i.e. the model predicts the next word in the Spanish sequence about 64% of the time. In practice, next-token accuracy is not a good metric for machine translation models because it makes the assumption that the correct target tokens from 0 to N are already known when predicting $N + 1$. In reality during inference you are generating the target sequence from scratch which means the preceding tokens may not be 100% correct. In real-world machine translation systems, one uses BLEU scores [**PRWZ02**] to evaluate the models. This metric looks at entire generated sequences and this metric seems to correlate more closely to human perception of translation quality.

Note that the amount of time taken to train this model was significant. It usually takes a long time to train RNN NMT models.

We now use our model for inference. We will pick a few sentence in the test set and check how our model translates them. We will start from the seek token, "[start]", and feed it into the decoder model, together with the encoded English sentence. We will retrieve the next-token prediction and re-inject it into the decoder repeatedly, sampling one new target token at each iteration, until we get to "[end]", or reach the maximum sentence length.

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
```

```
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
        sampled_token = spa_index_lookup(sampled_token_index)
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sequence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decoded_sequence(input_sentence))
```

This inference setup is easy to understand, but is rather inefficient since we reprocess the entire source sentence and the entire generated target sentence every time we sample a new word. In practice, one would factor the encoder and the decoder as two separate models and your decoder would only run a single step at each token-sampling iteration, reusing its previous internal state.

Here are our translation results. Our model works okay for a toy demonstration with a rather small vocabulary dataset. But it still makes many mistakes.

```
%who is in this room?
%[start] quien esta en esta habitacion [end]
% GOOGLE TRANSLATE: Quien esta en esta habitacion?
```

```
%That doesn't sound too dangerous.
%[start] eso no my difficil [end]
%GOOGLE TRANSLATE: eso no suena demasiado peligroso

%No one will stop me.
%[start] nadie me va a hacer [end]
%GOOGLE TRANSLATE: Nadie me detendra

%Tom is friendly.
%[start] tom es un buen [UNK] [end]
%GOOGLE TRANSLATE: Tom es amigable.
```

6.3. Transformer Sequence-to-Sequence Model. Sequence to sequence learning is currently done using the Transformer model. Neural attention concepts allows the Transformer to successfully process sequences that are considerably longer and more complex than those that RNNs can handle successfully.

As a human translating English to Spanish, you will not read the English sentence one word a time, keep its meaning in memory, and then generate the Spanish sentence one word at a time. That may work for simple five-word sentences, but is unlikely to work well for an entire paragraph. Instead, you would want to go back and forth between the source sentence and your translation in progress, and pay attention to different words in the source as you are writing down different parts of your translation.

That is exactly what is done with neural attention and Transformers. We already introduced the Transformer encoder that uses self-attention to produce context-aware representations of each token in an input sequence. In a sequence-to-sequence Transformer, the Transformer encoder forms the front end of the model. The Transformer encoder reads the source sequence and produces an encoded representation of it. Unlike the previous RNN encoder, though, the Transformer encoder keeps the encoded representation in a sequence format: in other words it keeps a sequence of context-aware embedding vectors.

The second half of the model is the *Transformer decoder*. Just like the RNN-decoder, it reads tokens $0, \ldots, N$ in the target sequence and tries to predict token $N+1$. But while doing this it uses neural attention to identify which tokens in the encoded source sentence are most closely related to the target token it is currently trying to predict. This is somewhat similar to the way a human translator would do it. Recall the query-key-value model: in a Transformer decoder the target sequence serves as an attention "query" that is used to pay closer attention to different parts of the source sequence (the source sequence plays the roles of both keys and values).

The Transformer Decoder: Fig. 20 shows the full sequence-to-sequence Transformer. Notice that the internal structure of the decoder is very similar to that of the encoder. What you will see, however is an extra self attention block (MHA/Normalization layers) inserted between the Dense layers producing the output and the self-attention block processing the target sequence. We can implement the Transformer decoder using Keras `layer` subclass.

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True

    def get_config(self):
        config = super().get_config()
```

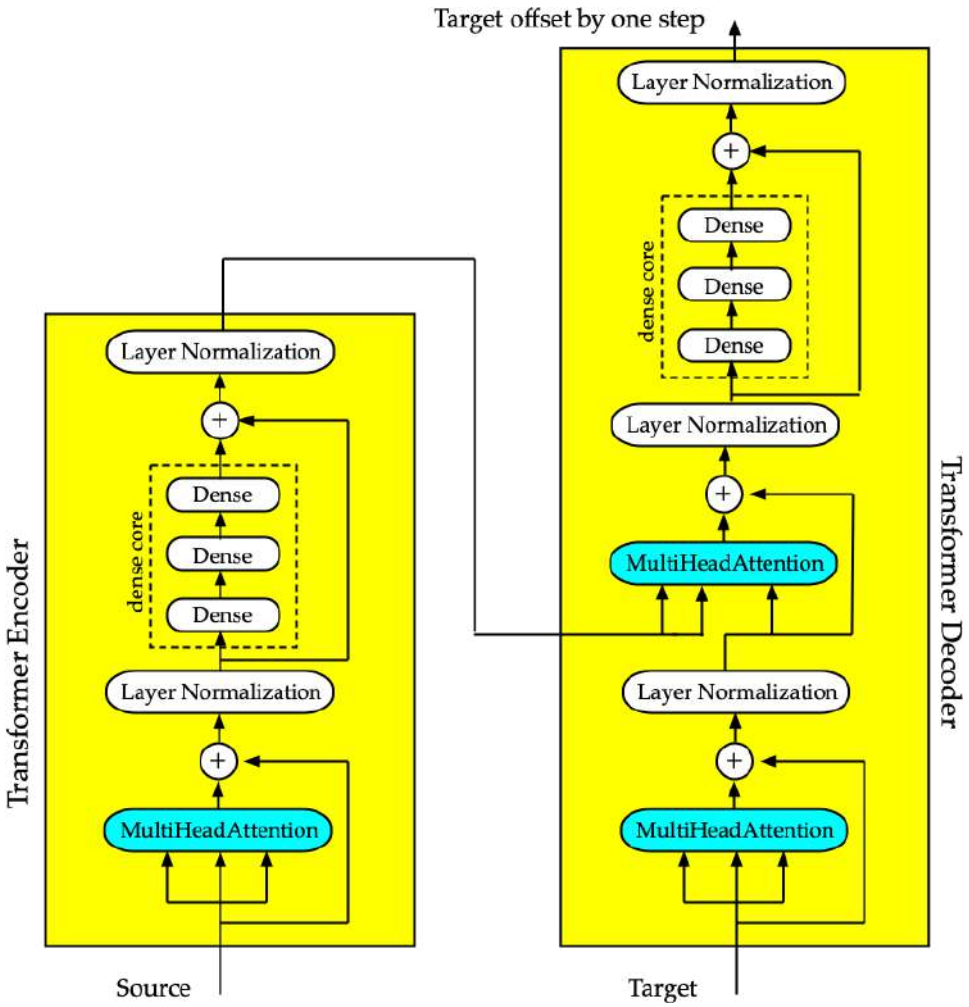


FIGURE 20. The transformer decoder is similar to the transformer encoder, except it features an additional attention block where the keys and values are the source sequence encoded by the transformer encoder. Together the encoder and decoder form an end-to-end transformer.

```
config.update({
    "embed_dim": self.embed_dim,
    "num_heads": self.num_heads,
    "dense_dim": self.dense_dim,
})
return config
```

The `call()` method is almost a straightforward rendering of the connectivity shown in Fig. 20. There is an additional detail that is needed: *causal padding*. Causal padding is critical to successfully training sequence-to-sequence Transformers. Unlike an RNN which looks at its input one step at a time and thus only has access to inputs $0, \ldots, n$ when generating the output $N + 1$st step, the Transformer Decoder is order-agnostic because it looks at the entire sequence at once. If it were allowed to use its entire input, it would simply learn to copy input step $N + 1$ to location N in the output. The model would therefore achieve perfect training accuracy, that would fail disastrously during the inference phase.

We address this issue by masking the upper half of the pairwise attention matrix to prevent the model from paying attention to information from the future. In this way only the tokens 0 to N in the target sequence are used when generating output token $N+1$. We do this by adding a causal attention mask method to the Transformer decoder as shown below.

```
def get_causal_attention_mask(self, inputs):
        input_shape = tf.shape(inputs)
        batch_size, sequence_length = input_shape[0], input_shape[1]
        i = tf.range(sequence_length)[:, tf.newaxis]
        j = tf.range(sequence_length)
        mask = tf.cast(i >= j, dtype="int32")
        mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1),
             tf.constant([1, 1], dtype=tf.int32)], axis=0)
        return tf.tile(mask, mult)

#implementation of call method
    def call(self, inputs, encoder_outputs, mask=None):
        causal_mask = self.get_causal_attention_mask(inputs)
        if mask is not None:
            padding_mask = tf.cast(
                mask[:, tf.newaxis, :], dtype="int32")
            padding_mask = tf.minimum(padding_mask, causal_mask)
        attention_output_1 = self.attention_1(
            query=inputs,
            value=inputs,
```

```
            key=inputs,
            attention_mask=causal_mask)
        attention_output_1 = self.layernorm_1(inputs + attention_output_1)
        attention_output_2 = self.attention_2(
            query=attention_output_1,
            value=encoder_outputs,
            key=encoder_outputs,
            attention_mask=padding_mask,
        )
        attention_output_2 = self.layernorm_2(
            attention_output_1 + attention_output_2)
        proj_output = self.dense_proj(attention_output_2)
        return self.layernorm_3(attention_output_2 + proj_output)
```

Position Embedding Layer: The second thing that we add into our implementation of the Encoder-Decoder is a *positional embedding layer*. Our sequence-to-sequence models relies on the neural attention mechanism. Self-attention focuses on the relationships between pairs of sequence elements so it is blind to whether these relationships occur at the beginning, middle, or end of the sentence. Position embeddings inject information about a token's position by adding a "position" axis to the word vector. The problem one might have with a direct implementation of this idea is the fact that the values along this position axis might be very large. We already know that an important part about neural network inputs is that we need to scale them so they all lie in about the same range. The way we usually do this for positional embedding is by taking the position number and passing it through a cosine function so it remains in the range $[-1, 1]$. The following script shows how we would instantiate such a position embedding layer

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim
```

```
    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super(PositionalEmbedding, self).get_config()
        config.update({
            "output_dim": self.output_dim,
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

The end to end transformer is the model we will be training. It maps the source sequence and the target sequence to the target sequence one step in the future. It combines the pieces we've built so far: Positional Embedding layers, The Transformer Encoder, and the Transformer Decoder. Note that both the encoder and decoder are shape invariant, so you could be stacking many of them to create a more powerful encoder or decoder. The following example just has a single instance of each.

```
embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)
x = layers.Dropout(0.5)(x)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

We now train our model, which does take a long time to train.

```
transformer.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

Finally we test our model by using it to translate never-seen-before English sentences from the test set. The setup is identical to what we used for the sequence-to-sequence RNN model, and so we don't repeat it here. Results from this are shown below

```
This is a song I learned when I was a kid.
[start] Esta es una cancion que aprendi cuando era chico [end]
#Google Translate: Esta es una canción que aprendí cuando era niño

She can lay the piano.
[start] ella puede tocar piano [end]
#Google Translate: Ella puede tocar el piano.

It may have rained a little last night.
[start] puede que llueve un poco el pasado [end]
#Google Translate: Puede que haya llovido un poco anoche
```

CHAPTER 7

Deep Generative Learning

Deep Generative Learning [Fos22] has recently generated considerable attention due to applications such as DALLE-2 and ChatGPT. Both of these applications can take a natural language prompt and produce a meaningful response. The response for ChatGPT is a text response as shown on the left side of Fig. 1. The response for DALL-E 2 is a picture as shown on the right side of Fig. 1.

Prompt: "Write a 75 word summary of what ChatGPT is."
Response: *ChatGPT is a large language model developed by OpenAI. It is trained on a massive dataset of text and is able to generate human-like responses to a wide range of prompts. It can be used for a variety of tasks such as language translation, text summarization, and conversation generation. It has been trained on a diverse set of internet text and is capable of understanding and generating text in a variety of languages and styles.*

ChatGPT Prompt and Response

Prompt: "Teddy bears working on new AI research underwater with 1990's technology"

DALL-E 2 Prompt and Response

FIGURE 1. Prompt and Response for ChatGPT (left) and DALL-E 2 (right)

Deep generative learning is a significant departure from the earlier supervised learning problems considered in prior chapters. Supervised learning results in *discriminative models* since it trains models that are used to discriminate whether a given input lies in a given class or not. Generative

learning, on the other hand builds models that generate new samples whose distribution match that of a given collection of training inputs. Recall that the system in our learning-by-example problem from chapter 1 generates *input* samples, x, that are drawn from an unknown probability distribution $F_{\mathbf{x}}(x)$. An observer would then create a *target*, y, by drawing from the conditional distribution $Q(y|x)$. The learning by example problem built models for $Q(y|x)$. In generative learning, we are building models for $F_{\mathbf{x}}(x)$.

There are several reasons why learning the input distribution, $F_{\mathbf{x}}(x)$, might be useful. Since the generative model creates samples that were not in the original dataset, it can be used to augment the original dataset in a way that is more powerful than the earlier transformation-based data augmentation schemes for CNNs in chapter 5. Generative models can also be used to identify fundamental features in the input data, thereby providing reduced order representations for input samples. We can use these reduced order representations to explore the input data in a way that allows us to generate new datasets that are *biased* for or against certain features in the original dataset. This last part is particularly important for creating "deep fakes" using ML and also for addressing issues of fairness and privacy in surveillance, social and medical systems. Finally, generative models can also be used to more quickly identify inputs that are inconsistent with the dataset. This last part is useful in flagging outliers in a real-time data stream and has obvious real-life applications in autonomous driving.

Until a few years ago, discriminative models were the main driver in deep learning. This is changing due to recent advances in generative adversarial networks (GAN) [**GPAM+14**], generative pre-trained transformers (GPT) [**RNS+18**] and diffusion models [**HJA20**]. These model architectures are the drivers behind the applications seen in Fig. 1 that, at least to the untrained eye, appear to pass the Turing test [**Tur09**]. As of the writing of these lectures (2023), generative learning has come to be seen as the next

major driver of deep learning technologies due not only to its technological breakthroughs but also due to the way it has penetrated the life of lay society.

The remaining sections of this chapter cover major deep learning architectures for generative modeling, starting from generative pre-trained transformers (GPT) [**RNS+18**], variational autoencoders [**KW13**], generative adversarial networks (GAN) [**GPAM+14**], and denoising diffusion probabilistic models (DDPM, also known as diffusion models) [**HJA20**]. Many of the scripts shown in this chapter were drawn from [**Fos22**].

1. Text Generation using Generative Pre-trained Transformers

The preceding chapter 6 showed how one can use RNN's or Transformers to predict the next words or next few words in a text fragment. This text fragment is called a *prompt*. For instance, if the prompt is "the cat is on the", and the model was trained on a database consisting of the name of Broadway musicals and plays, then the model would response might be "hot tin roof", followed with an attribution to the playright Tennessee Williams. Any model that learns the probability of the "next word" in a text prompt is called a *language model*. A language model captures the statistical structure of the language's *latent space*.

Once you have trained such a language model, you can *sample* from it to generate new sequences or sentences. This is done by feeding the model an initial string of text (the prompt) and asking it to generate the next character, word, or phase. That word or phrase is then added back to the model's input data and we repeat the process. This loop allows one to generate sentences (responses) of arbitrary length that reflect the structure of the data on which the model was trained. These sentences almost look "human-like" in their responses, since they were trained on natural language fragments. Fig. 2 provides a graphic illustration of this recursive approach to generating text.

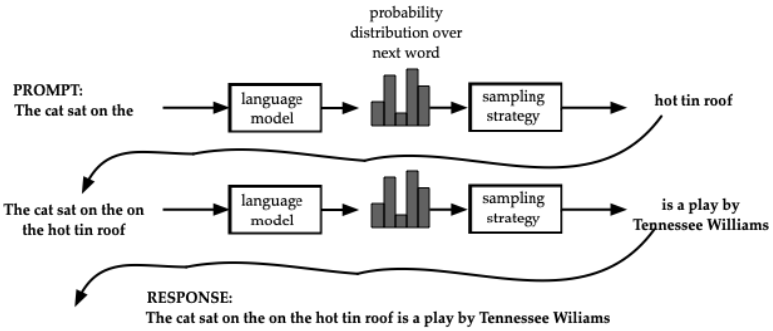


FIGURE 2. The process of word-by-word text generation using a language model

When generating text, the way one chooses the next token is critically important. A naive approach would use a *greedy sampling* strategy in which the model always selects the next most likely token. The greedy approach, however, often generates text that is predictable and may not look like coherent language. A more useful sampling strategy would select the next word in a probabilistic fashion. This is done by sampling from the probability distribution for the next word. Sampling probabilistically from the softmax output of the model allows for even unlikely words to be sampled from time to time, thereby generating more interesting reading sentences and sometimes exhibiting what one might perceive as "creativity" since it selects unexpected words that did not occur in the training data.

In practice, we would control the amount of randomness in the sampling process. If there is too much randomness, then the sentences become nonsensical. If there is too little, then the sentences because predictable. We control randomness by introducing a parameter called the *softmax temperature*. This temperature parameterizes the entropy of our draws from the probability distribution. Given a temperature value, a new probability distribution is computed from the original one. In particular, let $f(x)$ denote the original density and let T denote the softmax temperature. Then the new distribution, $g(x)$ would be

$$g(x) = e^{\log(f(x))/T}.$$

We would then sample from $g(x)$, rather than $f(x)$. Higher temperatures result in sampling distributions of higher entropy producing more surprising and unstructured sentences, whereas lower temperature results in less randomness and more predictable generated data.

We will now demonstrate how the softmax temperature impacts sentences generated by a pre-trained transformer model. This is, essentially, the same transformer that we used before in chapter 6. We refer to the model as *pre-trained* because it was trained on a large dataset (in this case the IMDB movie review dataset). We refer to the model as *generative* because we are using the transformer model to generate new movie reviews from a prompt.

In the last chapter, we built a transformer model for sequence-to-sequence learning. We trained this model by feeding a source sequence (sentence) into a transformer encoder and then fed both the encoded sequence and target sequence into a transformer decoder. The decoder was trained to predict the next word in the input sentences. Note that for text generation, we really don't the encoder. In particular, we will simply feed the same input to both input channels of the decoder.

In training this model, we used a callback to generate text using a range of different softmax temperatures after every epoch. This allows us to see how the generated text evolves as the model begins to converge, as well as the impact in the sampling strategy. We will seed our training with the text prompt, "This movie" so that all of our generated reviews start with this phrase. This model takes several hours to train and below we've shown shown some of the output generated by the model at the various temperatures.

- WIth temperature = 0.2

  This movie film moved attempts far from between comedy situations central steve plays west his ultimately affect suit gives key the approach movie to filled progress

life from adventures political humor tragedy humor violence pathos etc tolerable rookie personalities comedy cinematography ball original story music telling with how nice flooded a hollywood

- With temperature = 0.5

  This movie movie is at excellent truth funny [UNK] it is wasnt simply in boring fact history i i thought hated it it when was i over started [UNK] playing i it mean a everything tv like look this nothing movie can helicopters make and tricks better from then horrible on actors

- With temperature 0.7

  This movie movie has was musicals only in stars my mr family hair is this available movie on was a directed lot by of intervals sensation made retire hundreds after of playing times todays like youth many people movies believe the they geniuses put you forth down of to four think different

- With temperature 1.0

  This movie movie sucks was well bad into the town movie while why one did cant you be get frustrated through when so you many start people drinking are is hiroshima attempting and laugh just at because how you you invest feel in how some terrible episodes movies of should kurosawa try

- With temperature 1.5

  This movie is just fantastic so it wonderful didnt tale feel very and real that life it even was has very a nice soft score heart to based say on that what though a england rose bugs the does great a location must animation do to some show of it my too

This model, of course, does not have the degree of fluency seen in recent generative pre-trained transformer (GPT) models, so we are going to examine more recent GPT models that have proven to be particularly useful in natural language understanding tasks.

The remainder of this section describes a semi-supervised approach for fine-tuning pre-trained transformers used in natural language tasks involving comprehension and summarization. This semi-supervised approach was first discussed in [**RNS+18**]. A bidirectional version of Radford's model known as BERT (Bidirectional Encoder Representations from Transformers) [**DCT+19**] has proven to be particularly useful and pre-trained BERT models are now in TensorFlow/Keras model repositories. Another extension of BERT known as BART adds an autoregressive denoiser to the model, which makes it particularly useful for language summarization, generalization, and comprehension [**LLG+19**].

2. Feature Extraction using Principal Component Analysis

Generative models learn how to generate samples whose distribution match that of the input samples in the training dataset. This is, essentially, an *unsupervised* learning process for there are no targets we can use to guide the learning process.

One way of learning the input sample distribution would be to learn a parameterized model of the distribution. For example, let $\widehat{X} = \{\widehat{x}_k\}_{k=1}^M$ denote the collection of input samples, $x_k \in \mathbb{R}^n$, that were drawn in an i.i.d. manner from an unknown distribution $F_{\mathbf{x}}(x)$. We could use $\widehat{X}$ to construct an empirical distribution function

$$\widehat{F}_{\mathbf{X}}(x) = \frac{1}{M}\sum_{k=1}^{M}\sigma(x - \widehat{x}_k)$$

where $\sigma : \mathbb{R}^n \to \mathbb{R}$ is a monotone increasing function from 0 to 1. We know that this empirical distribution converges almost surely to the true distribution as $M \to \infty$.

The problem we have is that our input samples, x_k, have a high dimensionality, thereby meaning we would need an exponentially large number of input samples to obtain a good empirical estimate of the true distribution. Our dataset samples, however, will not usually be distributed uniformly across all of $\mathbb{R}^n$. In many real-life applications these samples are concentrated about a smooth lower dimensional surface in $\mathbf{R}^n$ called a *manifold*. These manifolds would have a lower dimensionality than n and the distribution could therefore be characterized with a smaller set of *latent variables* that represent coordinates on the manifold, rather than all of $\mathbf{R}^n$. We could also think of these latent variables as fundamental features of the input samples. So we would be learning those features containing most of the information in the original data points.

This leads to the following approach for generating samples matching the input sample's distribution. We would first identify an *encoder* that maps the input samples onto this lower dimensional manifold's latent space and have a decoder that that takes any vector in the latent space and maps it to an input sample in $\mathbb{R}^n$. This approach is, essentially, a data compression scheme where the encoder compresses the "information" in the input data and the decoder decompresses the latent variable to recover that information. Note that the compression step is usually lossy in the sense that some information may be irretrevably lost and hence cannot be recovered when decoding. So the main goal is to find the "best" encoder/decoder pair from a given family that minimizes the reconstruction error.

Principal component analysis (PCA) is one way for identifying the linear features used to encode a set of input samples. The goal of PCA is to identify a basis set of vectors forming a transformation that projects the original high-dimensional dataset onto a lower dimensional set of vectors

that minimizes the reconstruction error. The hope is that this new basis will optimally filter out "noise" and reveal hidden structure in the data that will provide a useful foundation for classification and exploration of the dataset.

To describe this approach more precisely, let the input data samples be denoted as $X = \{x_k\}_{k=1}^{M}$ where $x_k \in \mathbb{R}^n$ is an n-dimensional real-valued vector for all k. We can concretely represent X as a matrix

$$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & \cdots & x_M \end{bmatrix}$$

whose columns are the data samples, $x_k \in \mathbb{R}^n$. This matrix, therefore, lies in $X \in \mathbb{R}^{n \times M}$. Now let $\mathbf{P}$ be a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ where $m < n$. We let the concrete representation of $\mathbf{P}$ be an $\mathbb{R}^{m \times n}$ matrix. If we then look at

$$\mathbf{Y} = \mathbf{P}\mathbf{X},$$

we find that $\mathbf{Y} \in \mathbb{R}^{m \times M}$ is a matrix whose columns are the projection of the columns of $\mathbf{X}$ onto the lower m-dimensional *latent space*. In particular, we can view

$$\mathbf{P} = \begin{bmatrix} p_1^T \\ p_2^T \\ \vdots \\ p_{n_e}^T \end{bmatrix}.$$

This is a stack of m row vectors in which $p_k \in \mathbb{R}^n$ are seen as an alternative set of basis vectors spanning a subspace that approximates the data vectors in $\mathbf{X}$.

Note that the projection of $\mathbf{X}$ through $\mathbf{P}$ may lose information because the dimensionality of the latent vectors is less than that of the original data vectors. The covariance matrix of $\mathbf{Y}$ is defined as

$$\mathbf{C}_\mathbf{Y} = \frac{1}{M}\mathbf{Y}\mathbf{Y}^T = \mathbf{P}\left(\frac{1}{M}\mathbf{X}\mathbf{X}^T\right)\mathbf{P}^T = \mathbf{P}\mathbf{C}_\mathbf{X}\mathbf{P}^T$$

where $\mathbf{C}_\mathbf{X} = \frac{1}{M}\mathbf{X}\mathbf{X}^T$ is the covariance matrix of the original data matrix, $\mathbf{X}$. Ideally, we want the rows of $\mathbf{P}$ to be orthogonal vectors. If this is

the case then $\mathbf{C_Y}$ is a diagonal matrix and we say that the rows of $\mathbf{P}$ are *principal components* of $\mathbf{X}$.

Note that $\mathbf{X}\mathbf{X}^T$ is a symmetric matrix that can de decomposed as $\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$ where $\mathbf{\Lambda}$ is a diagonal matrix consisting of the eigenvalues of $\mathbf{C_X}$ and $\mathbf{V}$ is a matrix of eigenvectors of $\mathbf{C_X}$ arranged as columns. We can, therefore, see that if we choose $\mathbf{P}$ to be a matrix whose rows are eigenvectors of $\mathbf{C_X}$ then

$$\mathbf{C_Y} = \mathbf{V}^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{V} = \mathbf{\Lambda}.$$

We have just shown that the principal components of $\mathbf{X}$ are the eigenvectors of $\mathbf{C_X}$.

It is common to use singular value decompositions (SVD) to compute the principal components. SVDs represent the most numerically stable way of computing such decompositions for large data matrices. For any $m \times p$ matrix, $\mathbf{Q}$, one can prove that there exist $m \times m$ and $p \times p$ unitary matrices $\mathbf{U}$ and $\mathbf{V}$ and a real $r \times r$ diagonal matrix $\mathbf{\Sigma}$ such that

$$\mathbf{Q} = \mathbf{U}\begin{bmatrix} \mathbf{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\mathbf{V}^T.$$

The matrix $\mathbf{\Sigma}$ has the form

$$\mathbf{\Sigma} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$$

where $\sigma_i \geq \sigma_{i+1}$ for $i = 1, \ldots, r-1$ and $r \leq \min(m, p)$ is the rank of matrix $\mathbf{Q}$. The triple, $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$ is called the *singular value decomposition* of $\mathbf{Q}$. This decomposition is unique and σ_1 to σ_r are called the non-zero singular values of $\mathbf{Q}$. It can be readily shown that these non-zero singular values are also the positive roots of the non-zero eigenvalues of $\mathbf{Q}^T\mathbf{Q}$. The SVD of $\mathbf{Q}$ may also be written as

$$\mathbf{Q} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

where u_i and v_i are the ith rows of $\mathbf{U}$ and $\mathbf{V}$, respectively.

To see how this relates back to PCA, let us consider a data matrix $\mathbf{X}$ whose columns are the data sample vectors that have been centered with respect to the dataset's mean. Recall that $\mathbf{C} = \frac{1}{M}\mathbf{X}\mathbf{X}^T$ is the covariance matrix of the data matrix. We know the principal components are the eigenvectors of $\mathbf{C}_\mathbf{X}$. Now consider the SVD of the data matrix $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. Let us express the covariance matrix of $\mathbf{X}$ in terms of its SVD

$$\mathbf{X}\mathbf{X}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T.$$

We can therefore conclude that

$$\mathbf{C}_\mathbf{X} = \frac{1}{M}\mathbf{U}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

where $\mathbf{\Lambda}$ is a diagonal matrix whose diagonal elements are $\lambda_i = \frac{\sigma_i^2}{M}$. Since $\mathbf{U}$ is a unitary matrix (i.e. $\mathbf{U}^T\mathbf{U} = \mathbf{I}$) we can readily see that

$$\mathbf{C}_\mathbf{X}\mathbf{U} = \mathbf{U}\mathbf{\Lambda}.$$

This means that the columns of $\mathbf{U}$ are the principal components. Since we defined the PCA transformation $\mathbf{P}$ so its rows were the principal component vectors, we have $\mathbf{P} = \mathbf{U}^T$. If we then look at transforming all data points into the PCA coordinates we have

$$\mathbf{Y} = \mathbf{P}\mathbf{X} = \mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{\Sigma}\mathbf{V}^T.$$

This last result is used in the following example where we use the SVD of the data matrix to do a PCA of the Fisher iris dataset.

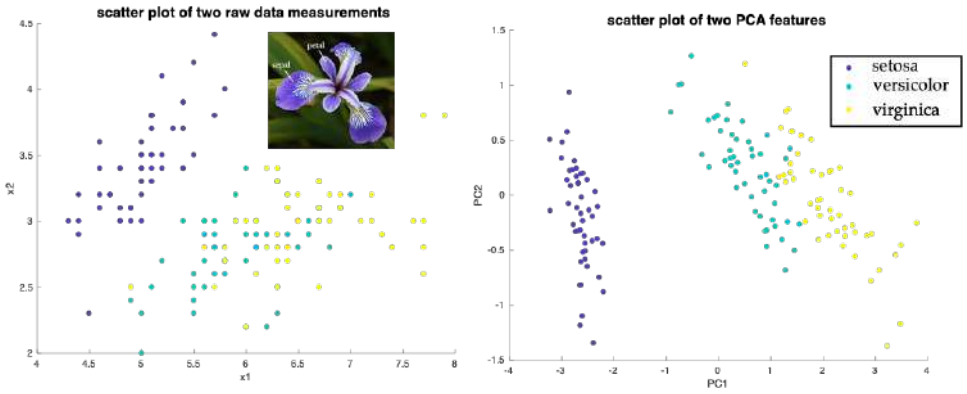


FIGURE 3. Fisher Iris Data set, PCA analysis

We consider a simple example using the SVD to find the principal components of Fisher's iris dataset [Fis36]. This dataset consists of 150 length and width measurements on the petals and sepals of several species of irises; setosa (class 0), versicolor (class 1), and virginica (class 2). This means that we have a feature vector of length 4 for each input sample. We are going to use the SVD of the data matrix to find the principal components of the dataset. I used the following MATLAB script to load the iris dataset, scatter plot the first two features, compute the SVD of the data matrix and then find the first right singular vectors with the largest singular values. The result is shown in Fig. 3. The left side shows that the data for the 3 classes are not well separated. The scatter plot for the two dominant principal components on the right show a much cleaner separation between the 3 classes.

```
load irisdata.txt
X = irisdata(:,1:4)';                  %150 measurements of length 4
spec = irisdata(:,5)';                %class labels
n = size(X,2);
figure(1)
scatter(X(1,:),X(2,:),30,spec,'filled')
xlabel("x1");ylabel("x2");
title("scatter plot of 2 raw measurements");

Xmean = mean(X,2);             %find mean
A = X - Xmean*ones(1,n);    %center the data
[U,S,V] = svd(A, 'econ')       %find SVD of centered data matrix
sigma = diag(S);
C = S(1:2,1:2)*V(:,1:2)';      %principal components of each data point
figure(2);
scatter(C(1,:),C(2,:),30,spec,'filled')
xlabel('PC1');ylabel('PC2');
title("scatter plot of 2 PCA features");
```

3. Autoencoders

An autoencoder is a deep learning model with an encoder-decoder architecture that takes an input image or text, encodes it over a space of latent variables, and then decodes that latent vector into the original image. We

can see the encoder as compressing the high dimensional input data onto the lower dimensional latent embedding vector. The decoder then decompresses that latent vector back into the original image. A diagram of this operation is shown below in Fig. 4 where the input is an image from the fashion MNIST database, the latent vector representation is denoted as z, and the decoder reconstructs the input image, albeit the reconstructed image is a bit blurry because our encoder is lossy.

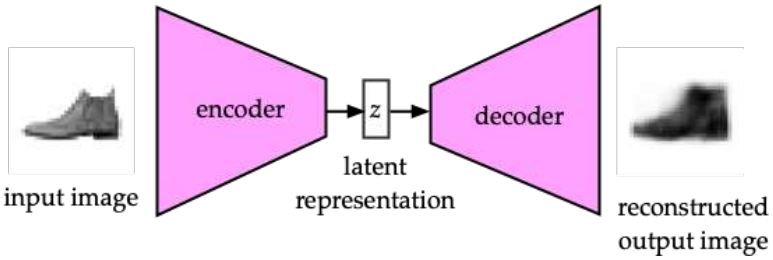


FIGURE 4. Autoencoder architecture

Note that the autoencoder is trained to *reconstruct* the input image, so that we train this with the input and target sample being the same image. This may seem strange at first, but we are not really interested in the output by itself. Our primary interest is in the latent representation, z, of that input. This is, therefore, an example of *representational learning*[**BCV13**]. The basic idea is that the various coordinates in the latent vector represent important features in the original image that we can use to topologically order how similar or different various images in the dataset might be. This will allow us to generate "new" images that were not in the original dataset.

The latent vector z is a vector in a low dimensional vector space, $\mathbb{R}^m$. If the embedding dimension, m, is 2 or 3, we can easily visualize the points in the dataset and see how various images are "close" or "far" apart. Let us do this for the fashion MNIST database . `Fashion-MNIST` is a dataset of clothing images [**XRV17**]. It consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28 by 28 grayscale image of an item of clothing with a label from one of 10 classes. Example images with labels from each class are shown in Fig. 5

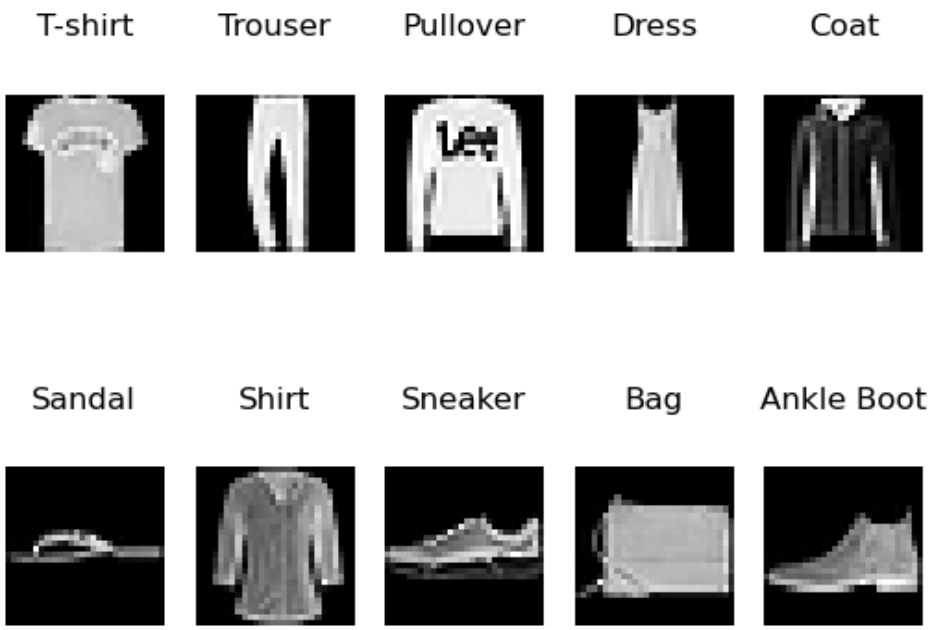


FIGURE 5. Sample Images from fashion MNIST dataset

We are now going to load the Fashion MNIST dataset and then construct an autoencoder based on CNNs. The dataset is included in TensorFlow, so we can load it as follows.

```
from tensorflow.keras import datasets
(x_train,y_train), (x_test, y_test) = dataset.fashion_mnist.load_data()
```

We are going to retype the pixel data from `uint8` to `float32` and normalize it so it takes values between 0 and 1. We will then zero pad and expand the shape of the input images so they are all $(32, 32, 1)$. This is done because our model expects a rank-3 tensor.

```
import numpy as np
def preprocess(imgs):
     imgs = imgs.astype("float32")/255.0
     imgs = np.pad(imgs, ((0,0), (2,2), (2,2)), constant_values=0.0)
     imgs = np.expand_dims(imgs, -1)
     return imgs

x_train = preprocess(x_train)
```

```
x_test  = preprocess(x_test)
```

We now declare the encoder model. The encoder consists of three 2D convolutional layers, that use strides to downsample the spatial dimension of the image and to increase the number of filte channels. The final output from the last convolutional layer has a shape of $(4, 4, 128)$. This output is then flatted to (2048) and a dense layer then connects it to the latent variable space. We will select a latent space with embedding dimension, m, of 2. So the encoder's output is simply a two dimensional vector that we can easily visualize

```
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]  # the decoder will need this!

x = layers.Flatten()(x)
encoder_output = layers.Dense(EMBEDDING_DIM, name="encoder_output")(x)

encoder = models.Model(encoder_input, encoder_output)
encoder.summary()
```

The decoder is a mirror image of the encoder that upsamples the latent vector from a shape of (2) to a shape of $(32, 32, 1)$. The 2-d latent vector is expanded through a dense layer to a shape of (2048) and then reshaped to $(4, 4, 128)$. We then use three transposed 2D convolutional layers to upsample the spatial dimensions until we get to the desired output shape.

```
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_input")
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
```

```
    64, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation="relu", padding="same"
)(x)
decoder_output = layers.Conv2D(
    CHANNELS,
    (3, 3),
    strides=1,
    activation="sigmoid",
    padding="same",
    name="decoder_output",
)(x)

decoder = models.Model(decoder_input, decoder_output)
decoder.summary()
```

Now that the encoder and decoder blocks have been declared we can create the autoencoder model by cascading the two blocks. The resulting model has $343,000$ trainable parameters.

```
autoencoder = models.Model(
    encoder_input, decoder(encoder_output)
)
autoencoder.summary()
```

We will use an Adam optimizer with an MSE loss function and then train the model for only 10 epochs. This model does not really need much training before it produces a meaningful latent space. Note that training is done using the images as the targets, rather than the class labels.

```
autoencoder.compile(optimizer="adam", loss="mse")
history = autoencoder.fit(
    x_train,x_train,
    epochs=10,
    batch_size=100,
    shuffle=True,
    validation_data=(x_test, x_test),
)
```

We can now visualize how this autoencoder mapped the original dataset images to the latent space. Recall that our latent space is 2-dimensional, so every image in the dataset was mapped onto a real-valued 2-vector. The right side of Fig. 6 shows how each sample in the dataset was embedded in $\mathbb{R}^2$. The dots are colored with the label of the input images class.

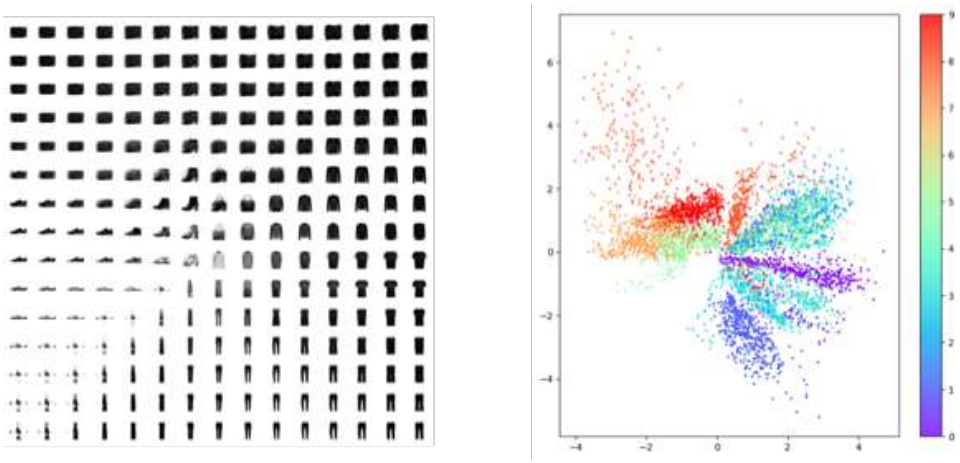


FIGURE 6. Fashion-MNIST Latent Space (right) embedding of all dataset samples (left) reconstructed images whose latent variables are regularly sampled in the 2-d latent space

As we mentioned before, we are not really interested in using the autoencoder to reconstruct known inputs. We are really interested in using the encoder to generate new sample images that were not in the original dataset. We can do this directly from the latent space by simply taking any latent vector in $\mathbb{R}^2$ and running it through the decoder only using the following script

```
grid_width, grid_height = (6, 3)
sample = np.random.uniform(
    mins, maxs, size=(grid_width * grid_height, EMBEDDING_DIM)
)
reconstructions = decoder.predict(sample)
```

The reconstructions were obtained by regularly sampling the latent space and then running the latent vector through the decoder. The left side of Fig. 6 shows the ”reconstructed” image for these latent vectors. What we see is that as we move across the space, that the images ”morph” in a somewhat predictable manner. In practice, this aspect of the autoencoder can be used to create a biased set of samples that ignore some features in the original dataset and accentuate others.

There are, however, several issues with the embeddings shown in Fig. 6. The first thing we notice is that some clothing items are represented over rather small areas of the latent space, whereas other clothing items cover a larger area. The other thing we notice is that there are large empty areas in the latent space that do not represent any actual clothing item. These observations suggest that if we were to try and ”generate” new samples by simply sampling the latent space in a uniform manner, that many of the reconstructions would not match well to any clothing item. This is a well known problem with autoencoders that we can address through a variation on the model known as a variational autoencoder or VAE.

4. Variational Autoencoders

In an autoencoder, each input sample maps to a specific point in the latent space. Variational autoencoders [**KW13**] map the inputs to a normal distribution centered at a point in the latent space. Because a normal distribution is completely characterized by its first two moments, we really only need to map the encoder's output to two different outputs; the mean vector and the variance matrix. In general, however, we assume the normal distribution has a diagonal covariance matrix, so that we really only need to have the encoder output two vectors, one for the mean of the distribution and the other for the diagonal of the covariance matrix. Since variance values are non-negative, however, we will find it more convenient to map to the log of the variance since this has values between $-\infty$ and ∞. This range of

mappings fits more nicely with our neural network models. As a result the variational autoencoder architecture changes from what we showed in Fig. 4 for the autoencoder. Fig. 7 illustrates this change where the latent variables between the encoder and decoder now are the mean, `z_mean` and the log of the variance, `z_log_var` of the distribution. These two outputs parameterize a Normal probability distribution that we then sample to obtain the actual vector, z, used by the decoder to reconstruct the input image.

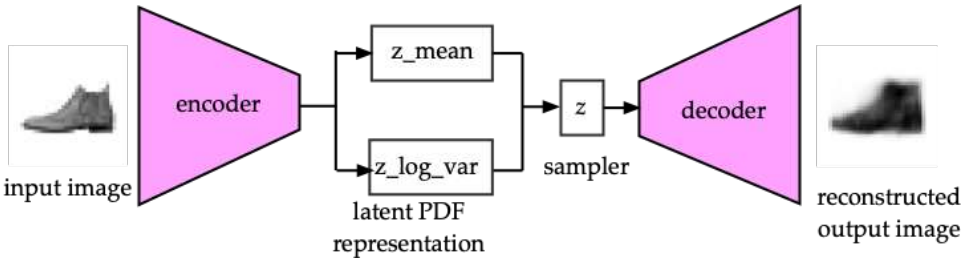


FIGURE 7. VAE Model Architecture

We can build the VAE model in much the same way we did for the autoencoder. There are, however, some significant differences. The first major difference is seen in the encoder where the encoder now has three possible outputs, `z_mean`, `z_log_var`, and `z`. The extra output `z` is a randomly drawn sample from the distribution defined by `z_mean` and `z_log_var`. This sampled output, `z`, is generated by a new `Sampling` layer class that we define at the top of the following script.

```
IMAGE_SIZE = 32
EMBEDDING_DIM = 2

#Sampling Layer
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

# Encoder
```

```
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]  # the decoder will need this!

x = layers.Flatten()(x)
z_mean = layers.Dense(EMBEDDING_DIM, name="z_mean")(x)
z_log_var = layers.Dense(EMBEDDING_DIM, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])

encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder")
encoder.summary()
```

The decoder is similar to what we used for the autoencoder

```
# Decoder
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_input")
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation="relu", padding="same"
)(x)
decoder_output = layers.Conv2D(
    1,
    (3, 3),
    strides=1,
    activation="sigmoid",
    padding="same",
    name="decoder_output",
)(x)

decoder = models.Model(decoder_input, decoder_output)
decoder.summary()
```

The other big change from our earlier encoder is our choice of loss function. The autoencoder used MSE or binary cross-entropy for the loss. The VAE, however, needs to a loss function that helps the latent layer learn the mean and variance of the normal distributions that each input maps to. This is accomplished by regularizing the loss function so it penalizes means and variance that are not close to a unit variance normal distribution. This is accomplished by taking our loss to be the sum of the *reconstruction loss* (measured as before by the MSE between the input and the reconstruction) and an additional weighted loss term that measures the "distance" of the current distribution's mean/variance against a zero-mean unit variance normal distribution. This second measure is usually taken to be the Kullback-Leibler (KL) divergence. The KL divergence for two densities p and q is defined as

$$\text{KL divergence} = D_{KL}(p, q) = \int p(x) \log \frac{p(x)}{q(x)} dx.$$

This measure equals zero when $p(x) = q(x)$. In our case, p is the normal distribution $N(\texttt{z\_mean}, \texttt{z\_log\_var})$ and q is the normal distribution $N(0, 1)$ so the KL divergence is

$$\texttt{kl\_loss} = -\frac{1}{2} \sum_i (1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2)$$

where $\texttt{z\_mean} = \mu$ and σ is the diagonal of the covariance matrix. The actual loss function used to train the VAE adds the KL divergence to the reconstruction loss

$$\text{VAE Loss} = \text{reconstruction loss} + \beta \times \text{KL divergence}$$

where β is a regularization parameter on the KL divergence.

The use of this loss function, however, requires that we build the overall VAE model as a custom subclass of the Keras `Model` class. This allows us to include the computation of the KL divergence and add it to the reconstruction loss through a custom `train_step` method. Note that this particular model uses binary cross entropy for the reconstruction loss.

```
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def call(self, inputs):
        """Call the model on a particular input."""
        z_mean, z_log_var, z = encoder(inputs)
        reconstruction = decoder(z)
        return z_mean, z_log_var, reconstruction

    def train_step(self, data):
        """Step run during training."""
        with tf.GradientTape() as tape:
            z_mean, z_log_var, reconstruction = self(data)
            reconstruction_loss = tf.reduce_mean(
                BETA
                * losses.binary_crossentropy(
                    data, reconstruction, axis=(1, 2, 3)
                )
            )
            kl_loss = tf.reduce_mean(
                tf.reduce_sum(
                    -0.5
                    * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                    axis=1,
                )
            )
            total_loss = reconstruction_loss + kl_loss

        grads = tape.gradient(total_loss, self.trainable_weights)
```

```
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))

        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)

        return {m.name: m.result() for m in self.metrics}

    def test_step(self, data):
        """Step run during validation."""
        if isinstance(data, tuple):
            data = data[0]

        z_mean, z_log_var, reconstruction = self(data)
        reconstruction_loss = tf.reduce_mean(
            BETA
            * losses.binary_crossentropy(data, reconstruction, axis=(1, 2, 3))
        )
        kl_loss = tf.reduce_mean(
            tf.reduce_sum(
                -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                axis=1,
            )
        )
        total_loss = reconstruction_loss + kl_loss

        return {
            "loss": total_loss,
            "reconstruction_loss": reconstruction_loss,
            "kl_loss": kl_loss,
        }
```

We are now in a position to create the VAE model using our preceding VAE class object. We compile this with an Adam optimizer and then train the model for 10 epochs. In this example we used a KL regularization parameter $\beta = 500$.

```
vae = VAE(encoder,decoder)
optimizer = optimizers.Adam(learning_rate=0.0005)
vae.compile(optimizer=optimizer)

history = vae.fit(
    x_train,
```

```
    epochs=10,
    batch_size=100,
    shuffle=True,
    validation_data=(x_test, x_test),
    callbacks=[model_checkpoint_callback, tensorboard_callback],)
```

The new latent space is shown in Fig. 8. We now see that the classes are more equally represented in the latent space. The white spaces between classes appears to be smaller.

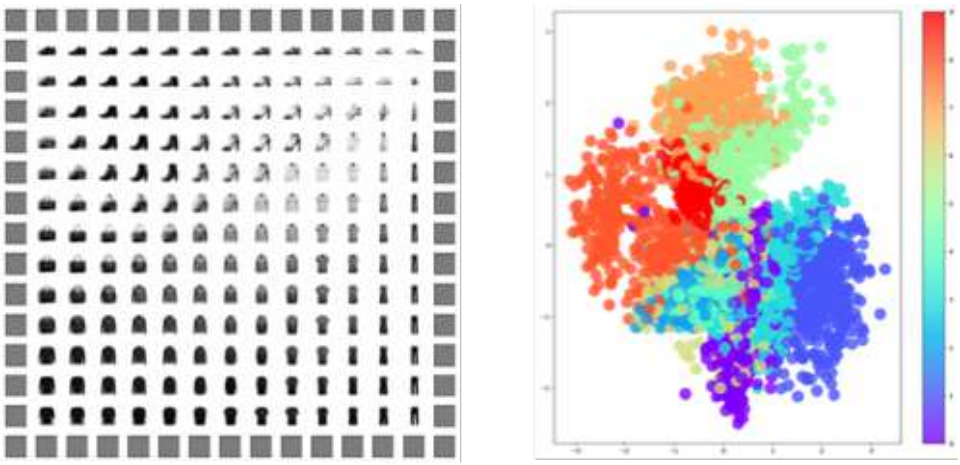


FIGURE 8. VAE latent space for fashion MNIST (left) reconstructions (right) location of sampled points

5. Generative Adversarial Networks

Generative adversarial networks (GAN) [**GPAM+14**] were first proposed in 2014. Their introduction stimulated a great deal of interest in generative learning and led to some of generative learning's most impressive accomplishments. This section introduces a version of the GAN that appeared in [**RMC15**].

The GAN takes a *game theory* approach to learning how to generate new samples from the system's input distribution, $F_{\mathbf{x}}(x)$. Game theory envisions

two players who take actions that further their own self interests while having the additional impact of interfering with the competing player's game performance. The basic idea is that these two players struggle until they reach a point from which neither player can gain an advantage over the other. Such points are called *Nash equilibria*. The GAN's two players are a *generator* and *discriminator* model. The generator generates samples that may be seen as "fake" copies of inputs in the original dataset. The generator trains itself so its distribution of "fake" samples matches the distribution, $F_{\mathbf{x}}(x)$, of inputs in the original dataset. The other player is a *discriminator* who takes an input from either the generator or dataset and updates itself so it can correctly discriminate between "fake" samples from the generator and "true" samples from the dataset. Another way of thinking about this game is that the "generator" learns to create samples that can "fool" the discriminator while the discrimnator is learning how to distinguish the generator's fake samples from the true ones.

The block diagram in Fig. 9 illustrates how one goes about training a GAN. At each training step, we randomly select an input for the discriminator that is chosen from either the training dataset or the Generator, G. The sample created by the generator is obtained by randomly selecting a latent vector and then feeding it through the generator. The sample from the training set is obtained by randomly sampling that dataset. The discriminator, D, then classifies the input as being either "fake" or "true". Since the trainer knows if the sample is fake or not, we can compute the loss function for the given sample and then use that sample's loss to drive a backpropagation update of the generator and the discriminator.

Let us now describe the training process in a more formal manner. Let $P_{\mathbf{z}}(z)$ denote the distribution of a latent vector drawn from the latent space. The chosen latent vector is used by the generator to create an output $\widehat{x}$ that has a distribution $F_{\widehat{\mathbf{x}}}(\widehat{x})$. Let $F_{\mathbf{x}}(x)$ denote the distribution of the training

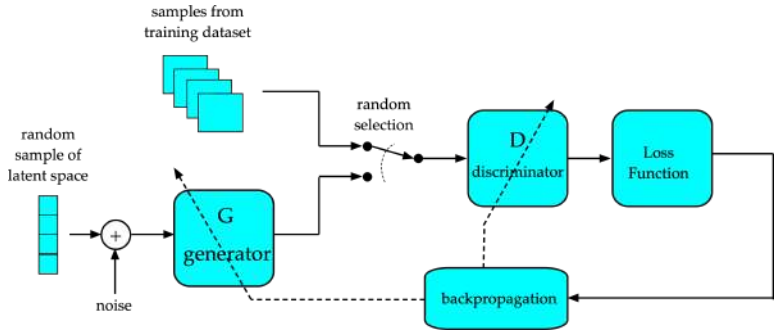


FIGURE 9. GAN Training

data set's samples, x. We train the discriminator, D, to maximize its accuracy over the "real" data point, x, by maximizing

$$\mathbb{E}_{\mathbf{x}} \left[\log D(\mathbf{x}) \right].$$

Meanwhile for the *fake* sample, $\widehat{x}$, created from the latent vector z, we train the generator, G, so the discriminator outputs a probability, $D(G((z))$, that is small for "fake" samples and close to 1 for "real" samples. This means that the generator is trained to *minimize* the discriminator's accuracy on fake data,

$$\mathbb{E}_{\mathbf{z}} \left[\log(1 - D(G(\mathbf{z})) \right].$$

So we can now define a loss function that looks like our usual binary cross entropy function,

$$L(D, G) = \mathbb{E}_{\mathbf{x}} \left[\log D(\mathbf{x}) \right] + \mathbb{E}_{\mathbf{z}} \left[\log(1 - D(G(\mathbf{z}))) \right]$$

but now treat it as a function of both D and G.

We pick D to maximize $L(D, G)$ and pick G to minimize $L(D, G)$, so that these two players are working against each other. We define the equilibrium (D^*, G^*) as a pair of models for which

$$L(D, G^*) \leq L(D^*, G^*) \leq L(D^*, G).$$

In other words, the equilibrium produces a loss such that any deviation of the generator increases the loss and any deviation of the discriminator decreases the loss. The equilibrium therefore occurs when no player (D or G) can gain a clear advantage over the other. What this means is that the update of G and D shown in Fig. 9 is actually done in an alternating manner; first training G, and then training D, and continuing back and forth in this manner until we are satisfied with the result.

We now demonstrate the construction and training of a deep convolution GAN (DC-GAN) used to generate "fake" images. This Keras implementation uses the MNIST database since it takes less time to train.

```
(train_images, train_labels),(_, _) = tf.keras.datasets.mnist.load_data()

train_images = train_images.reshape(train_images.shape[0],
                            28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5
# Normalize the images to [-1, 1]

BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images)
train_dataset.shuffle(BUFFER_SIZE)
train_dataset.batch(BATCH_SIZE)
```

The generator uses the `Conv2DTranspose` layers to upsample and produce an image from a random noise seed. A dense layer takes this seed vector as input and upsamples several times using strides to obtain the desired image size $(28, 28, 1)$. This model used a leaky ReLU activation for most of the layers as this has been shown empirically to work better than ReLu activation functions. The generator model architecture is shown in Fig. 10. The discriminator is a CNN image classifier and its model architecture is shown in Fig. 10.

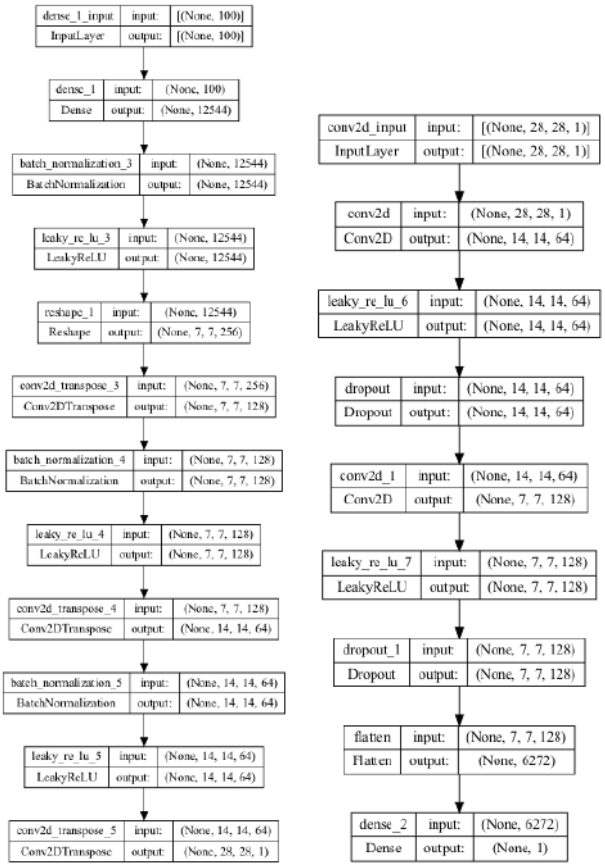


FIGURE 10. DCGAN generator and discriminator models

We define a separate loss function and optimizer for each model, since they are trained separately. The discriminator's loss function quantifies how well the discriminator is able to distinguish real images from fake images. It compares the discriminator's prediction on the real image to an array of 1's and the prediction on the fake input to zeros. So the discriminator loss function is a binary cross entropy function,

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

The generator's loss quantifies how well it is able to trick the discriminator. This loss function is simply the cross entropy function

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

We also declare two separate optimizers because we are training two different models. They both use the Adam optimizer, rather than RMSprop. The learning rate is usually set very small to help stabilize learning and in some cases the learning rate is changed adaptively.

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Training the GAN follows the flow shown in Fig. 9. We therefore have to write a custom `train` method for the GAN. This method will call the following `train_step` function which uses the GradientTape object to separately compute gradients used in updating the weights of the two models. Note that the training-step in this implementation trains the discriminator for one step, and then the generator for one step. In many applications, however, it has proven to be better to train the generator for several steps before updating the discriminator.

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss,
                                 generator.trainable_variables)
```

```
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                 discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                 generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
                                  discriminator.trainable_variables))
```

Figure 11 shows the generated images obtained after each training epoch. The 16 randomly generated images start as all identical, but quickly differentiate into figures that "look" like the numbers in the NMIST database.

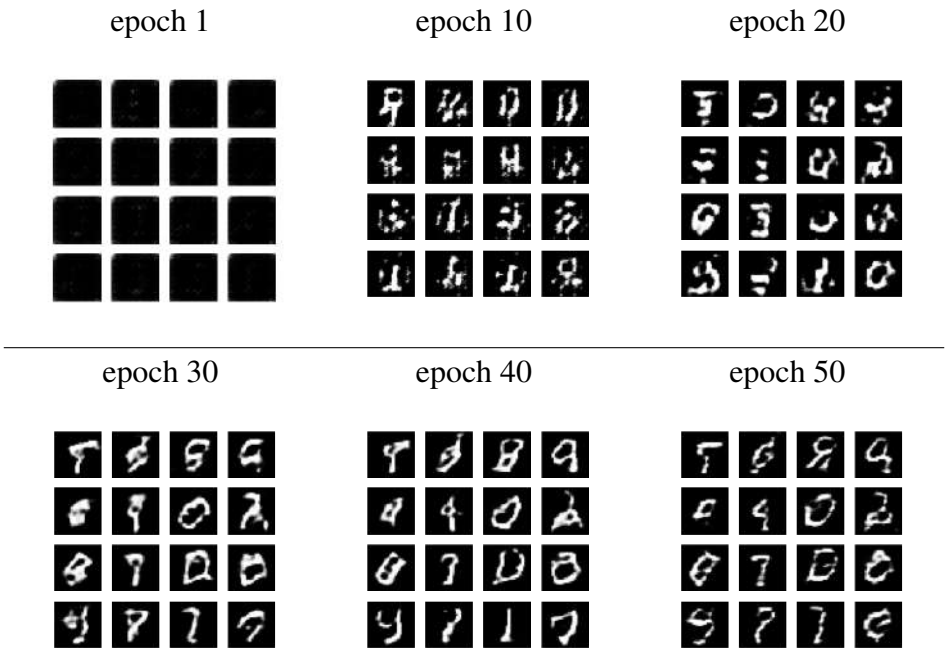


FIGURE 11. GAN generated images

Although GANs have been enormously successful in image generation, these models are notoriously difficult to train. Salimans [SGZ+16] discussed the difficult of achieving a Nash equilibrium using gradient-descent procedures. As mentioned above, the alternating nature of the training process means that every training step undoes the benefits of the preceding step. Getting this type of competitive updating to converge is very difficult and

is one reason why we usually update the generator several steps, before updating the discriminator. Another problem arises when the true distribution, $F_{\mathbf{x}}(x)$, and the fake distribution $F_{\widehat{\mathbf{x}}}(\widehat{x})$ have support on a low dimensional manifold [**AB17**]. Many real-world datasets may have training data distributions $F_{\mathbf{x}}(x)$ that appear to be high dimension, but have the actual data sitting on a lower dimensional manifold. This means that the generator's distribution, $F_{\widehat{\mathbf{x}}}(\widehat{x})$ will also sit on a low dimensional manifold. Because both $F_{\mathbf{x}}(x)$ and $F_{\widehat{\mathbf{x}}}(\widehat{x})$ are supported on low dimensional manifolds embedded in a high dimensional space, it is highly likely that this two support sets will be disjoint, thereby making it easy to find a perfect discriminator that separates real and fake samples all of the time. In this case, the discriminator will be too strong and this will make it impossible for the generator updates to converge to the optimal G^*. Several suggestions have been made to improve training of GANs [**SGZ+16**, **AB17**].

6. Diffusion Models

Alongside GANs, diffusion models are one of the most influential generative modeling techniques for images. Diffusion models now outperform previous state-of-the-art GANs and have become the go-to choice for generative modeling engineers in the visual domain. For example, OpenAI's DALL-E 2 uses diffusion models to generate the output image after a transformer has decoded the user's prompting text. The breakthrough model , known as the *denoising diffusion probabilistic model (DDPM)* , appeared in 2020 [**HJA20**] and demonstrated performance that rivaled that of GANs across several datasets. Since then the basic DDPM architecture has been extended in many ways. This section takes a quick look at a version of the DDPM described in [**Fos22**] and [**YZS+22**].

The core idea behind DDPMs is relatively simple - we train a deep learning model to denoise an image over a series of very small steps. If we start from pure random noise, in theory we should be able to keep applying the

model until will obtain an image that looks as if it were drawn from the training set. The DDPM makes use of two Markov chains (see appendix B) to achieve this. There is a *forward chain* that perturbs data to noise and a reverse chain that converts noise back to data. The forward chain is usually hand-designed with the goal of transforming any data distribution into a standard Gaussian image. The reverse chain undoes the forward chain by learning transition kernels parameterized by deep neural networks. New data points are then generated by first sampling a random vector from the standard Gaussian, followed by ancestral sampling through the reverse Markov chain.

We can describe this more formally as follows. Let assume that we have an input $x_0 \in F_{\mathbf{x}}(x)$ that was selected from the training dataset's input distribution. The forward Markov chain generates a sequence of random variables, $\{\mathbf{x}_k\}_{k=0}^T$, with transition kernel $Q(x_t|x_{t-1})$. Using the chain rule of probability and the Markov property, we can factor the joint distribution of $x_1, x_2, \ldots, x_T$ conditioned on the initial input, x_0, which we denote as $Q(x_1, \ldots, x_T \,|\, x_0)$ into

$$Q(x_1, \ldots, x_t \,|\, x_0) = \prod_{t=1}^{T} Q(x_t \,|\, x_{t-1}). \tag{41}$$

In DDPMs, the transition kernel is handcrafted to incrementally transform the data distribution $x_0 \sim F_{\mathbf{x}}(x)$ into a tractable prior distribution which is usually taken to be a standard Gaussian distribution. The most common choice for the transition kernel is

$$Q(x_t \,|\, x_{t-1}) = N(\sqrt{1-\beta_t}x_{t-1}, \beta_t \mathbf{I}) \tag{42}$$

where $\beta_t \in (0, 1)$ is a hyperparameter chosen ahead of time by the designer. Note that this transition kernel allows us to marginalize the joint distribution in equation (41) to obtain an analytic form for $Q(x_t \,|\, x_0)$ for all $t \in \{0, 1, \ldots, T\}$. Specifically if we let $\alpha_t \stackrel{\text{def}}{=} 1 - \beta_t$ and $\overline{\alpha}_t \stackrel{\text{def}}{=} \prod_{k=0}^{t} \alpha_k$, then

we have

$$\begin{aligned} x_t &= \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1} \\ &= \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\epsilon \\ &\vdots \\ &= \sqrt{\overline{\alpha}_t} x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon \end{aligned}$$

so we can conclude that

$$Q(x_t|x_0) = N(\sqrt{\alpha\alpha_t}x_0, (1-\overline{\alpha}_t)\mathbf{I}).$$

This means that given the initial input x_0, we can easily obtain a sample x_t by sampling a Gaussian vector $\epsilon \sim N(0, \mathbf{I})$ and applying the above transformation $x_t = \sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon$. When $\overline{\alpha}_T \approx 0$, then x_T is nearly a Gaussian distribution and so $Q(x_T) \approx N(0, \mathbf{I})$.

To generate new data samples, DDPMs start by first generating an unstructured noise vector from the prior distribution and then gradually removing noise through a learned Markov chain running in the reverse direction. In particular, the reverse Markov chain is parameterized by a prior distribution $P(x_T) = N(0, \mathbf{I})$ because the forward process was constructed so that $Q(x_T) \approx N(0, \mathbf{I})$. The learnable transition kernel $P_w(x_{t-1} \,|\, x_t)$ with weights w takes the form

$$P_w(x_{t-1} \,|\, x_t) = N(\mu_w(x_t, t), \Sigma_w(x_t, t))$$

where w denotes the model parameters and the man $\mu_w(x_t, t)$ and variance $\Sigma_w(x_t, t)$ are parameterized by deep neural networks. With this reverse Markov chain in hand, we can generate a data sample x_0 by first sampling a noise vector $x_T \sim p(x_T)$ and then iteratively sampling from the learnable transition kernel $x_{t-1} \sim P_w(x_{t-1} \,|\, x_t)$ until $t = 1$.

The key to the success of this sampling process is training the reverse Markov chain to match the actual time reversal of the forward Markov chain. That is, we have to adjust the parameter w so the joint distribution of

the reverse Markov chain,

$$p_w(x_0, x_1, \ldots, x_T) \stackrel{\text{def}}{=} P(x_T)\prod k = 1^T P_w(x_{t-1} \,|\, x_t)$$

closely approximates that of the forward process

$$Q(x_0, x_1, \ldots, x_T) \stackrel{\text{def}}{=} Q(x_0)\prod_{k=1}^{T} Q(x_t \,|\, x_{t-1}).$$

This is achieved by minimizing the Kullback-Leibler (KL) divergence of these two distributions.

$$\begin{aligned} KL(Q(x_0, \ldots, x_T), \quad & P_w(x_0, \ldots, x_T)) \\ = \; & -\mathbb{E}_{Q(x_0,\ldots,x_T)} \left[\log P_W(x_0, \ldots, x_T)\right] + \text{const} \\ = \; & \mathbb{E}_{Q(x_0,\ldots,x_T)} \left[-\log P(x_T) - \sum_{k=1}^{T} \log \frac{P_w(x_{t-1} \,|\, x_t)}{Q(x_t \,|\, x_{t-1})}\right] + \text{const} \\ \geq \; & \mathbb{E}\left[-\log P_w(x_0)\right] + \text{const}. \end{aligned}$$

The first equation comes from the definition of KL divergence. The second equation comes from that fact that both joint distributions are products of distributions and the last equation is from Jensen's inequality. The first term in in the second line is the variational lower bound (VLB) of the log-likelihood of the data x_0, a common objective used for training probabilistic generative models. The "const" term contains terms that are independent of the weights w and hence don't impact training. The objective of DDPM training is to maximize the VLB, which is relatively easy because it is a sum of independent terms and can therefore be estimated efficiently through Monte Carlo sampling and optimized using stochastic gradient descent.

Notice that we are free to choose a different β_t at each time step. The original DDPM paper [**HJA20**] used a linear schedule where β_t increased by a fixed increment at each time step. It was later found that a sinusoidal

schedule worked better [ND21] where

$$\overline{\alpha}_t = \cos^2\left(\frac{\pi t}{2T}\right).$$

This gives rise to the following update sampling equation

$$x_t = \cos\left(\frac{\pi t}{2T}\right) x_0 + \sin\left(\frac{\pi t}{2T}\right) \epsilon$$

where ϵ is standard Gaussian noise.

We now illustrate the DDPM model following an example in [Fos22]. We'll be using the Oxford 102 flower dataset on Kaggle. After downloading the dataset we create the training dataset. Sample images from this dataset are shown in Fig. 12.

```
# Load the data
train_data = utils.image_dataset_from_directory(
    "data/flower-dataset/dataset",
    labels=None,
    image_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=None,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
)

# Preprocess the data
def preprocess(img):
    img = tf.cast(img, "float32") / 255.0
    return img


train = train_data.map(lambda x: preprocess(x))
train = train.repeat(DATASET_REPETITIONS)
train = train.batch(BATCH_SIZE, drop_remainder=True)

# Show some flowers from the training set
train_sample = sample_batch(train)
display(train_sample)
```

Our denoising diffusion model will be based on a U-net architecture [RFB15]. Before discussing the U-net model in depth, let us look at how

FIGURE 12. Sample Images from Oxford 102 Flower Dataset

it is trained. Our model will be trained in a custom way so we need to build it as a subclass of Keras `Model` class. It is important to note that our `DiffusionModel` actually keeps two copies of the U-net model in it. One is trained using stochastic gradient descent, the other uses an exponential moving average (EMA) of the weights of the other copy. This is done because the EMA network is not as susceptible to short-term fluctuations and spikes in the trianing process, therefore making it more robust for generation of images than the actively trained network. We therefore use the EMA network when we produce an output from the network. The following script shows the subclassing of `DiffusionModel` with its special training step. This script is commented to show what is actually done in the various steps

```
class DiffusionModel(models.Model):
     def __init__(self):
          super().__init__()
          self.normalizer = layers.Normalization()
          self.network = unet
          #clone the unet to create the EMA network
          self.ema_network = models.clone_model(self.network)
          self.diffusion_schedule = cosine_diffusion_schedule
     ...
     def denoise(self, noisy_images, noise_rates, signal_rates, training):
          #use ema-network if we are not training
          if training:
               network = self.network
          else:
               network = self.ema_network
          #we use the selected network to predict what the noise is in a single step
          pred_noises = network(
               [noisy_images, noise_rates**2], training = training)
          #we then undo the forward diffusion using the predicted noise
          pred_images = (noisy_images - noise_rates*pred_noises)/signal_rates
```

```
        return pred_noises, pred_images

    def train_step(self, images):
        #normalize batch of images to be zero mean and unit variance
        images = self.normalizer(images, training=True)
        #sample the noise to match shape of input images
        noises = tf.random.normal(shape=tf.shape(images))
        batch_size = tf.shape(images)[0]
        #randomly sample diffusion times and use these in the cosine schedule
        diffusion_times = tf.random.uniform(
              shape=(batch_size,1,1,1), minval=0.0, maxval = 1.0)
        noise_rates, signal_rates = self.cosine_diffusion_schedule(
              diffusion_times)
              #the apply the noise to the image for the forward pass
        noisy_images = signal_rates*images + noise_rates*noises

        #create a GradientTape object for training
        with tf.GradientTape() as tape:
            #where we denoise the noisy image by first predicting the noise
            # and undoing the noising operation with the noise/signal rates
            pred_noises, pred_images = self.denoise(
                noisy_images, noise_rates, signal_rates, training=True)

            #compute the loss and take a gradient step
            noise_loss = self.loss(noises, pred_noises)
        gradients = tape.gradient(noise_loss, self.network.trainable_weights)
        self.optimizer.apply_gradients(
            zip(gradients, self.network.trainable_weights)

        #update the ema network
        for weight, ema_weight in zip(
            self.network.weights, self.ema_network.weights):
            ema_weight.assign(0.999*ema_weight + (1-0.999)*weight)
        return {m.name: m.result() for m in self.metrics}
```

The model we will use is a U-net model [**RFB15**]. This model consists of an encoder and decoder with residual connections between the convolutional layers as shown on left side Fig. 13. This model takes the noise variance β_t and an image x with shape $(64, 64, 3)$ as an input. The output is the model's prediction of the noise added to the image also with shape $(64, 64, 3)$. The encoder generates a latent space of size $(8, 8, 128)$ through a series of `DownBlocks`. The decoder predicts the noise added to

the image during the forward pass. The decoder consists of a sequence of upsampling `UpBlocks`. The layers in the `DownBlock`, `UpBlock`, and `ResidualBlock` are shown on the right side of Fig. 13.

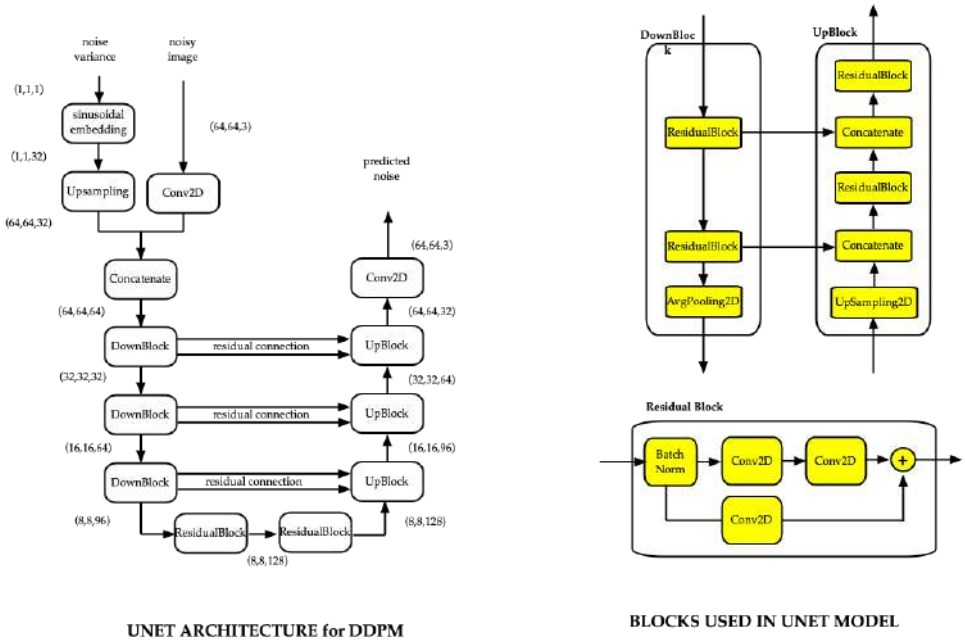


FIGURE 13. (Left) Unet model (Right) blocks used in Unet model

This model was trained on the Oxford Flower Dataset using an Adam optimizer for 50 epochs. We used a batch size of 64 samples. After each epoch we would take the output generated by denoising a Gaussian image for 10 different selections. Fig. 14 shows that this models does a very good job of producing realistic flower images that were not in the original dataset.

Note that we can also interpolate between two points in the latent space using the following script. The results in Fig. 15 show how this allows us to smoothly morph one generated into another image.

```
# Interpolation between two points in the latent space
tf.random.set_seed(100)
```

FIGURE 14. Reconstructions generated by DDPM model

```
def spherical_interpolation(a, b, t):
    return np.sin(t * math.pi / 2) * a + np.cos(t * math.pi / 2) * b


for i in range(5):
    a = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    b = tf.random.normal(shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
    initial_noise = np.array(
        [spherical_interpolation(a, b, t) for t in np.arange(0, 1.1, 0.1)]
    )
    generated_images = ddm.generate(
        num_images=2, diffusion_steps=20, initial_noise=initial_noise
    ).numpy()
    display(generated_images, n=11)
```

FIGURE 15. Interpolating between points in the latent space to morph between images

CHAPTER 8

Deep Learning and Human Society

Deep learning and human society are on a collision course. Deep learning lies at the heart of self-driving cars. It is essential for many of the convenient applications on our mobile devices. It makes possible voice assistants (Alexa, Siri) and we rely on it in helping us find directions to our destination. Deep learning appears regularly in reports that raise warnings regarding implicit racial bias in commercial AI products [**RB19**], voice concern over the use of ChatGPT in education [**Ope22**], and express astonishment over DALL-E 2 creating art from natural language descriptions [**Roo22**]. The simple fact is that deep learning applications mimic human behavior in a manner that appears to pass the Turing test (a.k.a. imitation game)[**Tur09**] under non-expert examiners. This fact should give us pause to consider how this technology might be used in lay society.

This chapter considers issues arising from the use of deep neural networks to predict and capture human behavior. In particular, we consider a scenario involving a health network managing clinics scattered over a wide geographic area. Each clinic retains data on its local residents that it uses to diagnose resident health conditions and to form treatment plans for residents who are classified as being ill. While each clinic focuses on its own given population of patients, the health network has a different set of concerns for determining treatment plans that should be used by *all clinics*. In particular, the health network may be interested in improving treatment accuracy averaged over all clinics, not just over a specific clinic's population.

With regard to this application we consider three related concerns. The first concern involves preserving the private (sensitive) information (data)

of individual patients. This privacy requirement means that an individual's detailed health history be kept by the clinic and not transferred to the health network server. The natural question then is how can the health network decide a global model when it does not have access to the all of the patient data. The second concern is whether it is possible to "skew" a clinic's outcomes by injecting "false" data when training the clinic's models. This his commonly viewed as an issue of "deep fakes" which has come to the forefront with recent advances in generative AI. The final concern involves the training of treatment models that are *fair* in the sense that treatment outcomes are statistically independent of some legally protected sensitive attribute of the population.

The following section discuss these three concerns. We first examine the problem of *adversarial examples* [**SZS+13**, **BCM+13**] as a means of disrupting model training. We then examine the notion of *differential privacy* [**Dwo08**] and then discuss a distributed approach known as *federated learning* [**KMA+21**] that trains models that can address the privacy issue. Finally we discuss the issue of *fairness* [**BHN19**], specifically in the context of federated learning frameworks.

1. Security: Creating Adversarial Examples

Collecting community datasets raises security concerns. One such concern is with *adversarial examples* [**SZS+13**, **BCM+13**]. These are perturbed data samples that are injected into a ML system to cause the system to make a false prediction or categorization. The maliciously perturbed data may be inputs to an inference version of the model, so the attacker can evade detection. These perturbed data samples can also be used during online ML training to compromise an existing model. This use of perturbed data is sometimes called data poisoning.

The concern with adversarial examples was kicked off by a surprising discovery in [SZS+13] that found several ML models, including state-of-the-art deep networks are vulnerable to adversarial examples. That work found very slight perturbations of a correctly sampled input could trick the trained model into making the wrong classification. These adversarial examples could be generated by slightly perturbing a correctly sampled input in the training data. Fig. 1 shows a widely known example, where an image classified as a "panda" with 57% confidence was perturbed with a slight bit of noise so that the perturbed image (shown on the right) was classified as a "gibbon" with 99% confidence. To the human observer the two images look nearly identical.

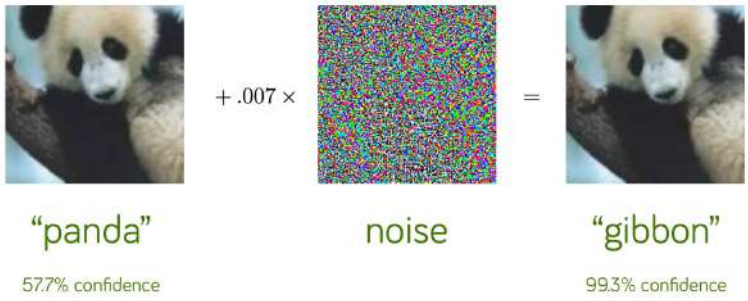


FIGURE 1. Example of an Adversarial Example

What this example shows is that neural network training may not result in models whose performance is **robust** to variations in the input data. This finding is surprising because it calls into question long-standing beliefs that the activation of a given node in a hidden layer represents a fundamental "feature" of the input that is necessary for classification. Recall that this is the approach that traditional image classification uses, where the image is projected onto a set of pre-engineered features and those features are then used for image classification and understanding. The existence of adversarial examples for deep learning models suggests that neural network performance is not really determined by the activation of these internal "features".

One way for addressing this issue is by generating adversarial examples and then adding those examples to the training data. There are several ways such adversarial examples can be generated. The following script shows a particularly simple example based on the MNIST dataset. For this example, we first have to train a model on the MNIST dataset. The model we are going to use is specified int he following script

```
#compile CNN network for MNIST classification
inputs = Input(shape=(28,28,1))
net = Conv2D(32, kernel_size=(3, 3),
                 activation='relu')(inputs)
net = Conv2D(64, kernel_size=(3, 3),
                 activation='relu')(net)
net = MaxPooling2D(pool_size=(2, 2))(net)
net = Dropout(0.25)(net)
net = Flatten()(net)
net = Dense(128, activation='relu')(net)
net = Dropout(0.5)(net)
outputs = Dense(10, activation='softmax')(net)

mnist_model = Model(inputs=inputs, outputs=outputs,
      name='classification_model')
mnist_model.compile(optimizer='nadam',
     loss='categorical_crossentropy',metrics=[categorical_accuracy])
```

We then train this network on the MNIST training data for 100 epochs and evaluate the trained model's accuracy on the training and test datasets. The test accuracy was 98.5% and the training accuracy was 98.6%.

```
earlyStop = EarlyStopping(monitor='val_categorical_accuracy',
              min_delta=0, patience=10, verbose=0, mode='auto',
                          baseline=None, restore_best_weights=True)

mnist_model.fit(x_train, y_train, batch_size=128, epochs=100,
                 verbose=0, validation_data=(x_test, y_test),
                 callbacks=[earlyStop])

print(mnist_model.evaluate(x_train, y_train))
print(mnist_model.evaluate(x_test, y_test))

#1875/1875 #[==============================] - 11s #6ms/step
#   - loss: 0.1083 - categorical_accuracy: 0.9860
```

```
#    [0.10834111273288727,0.9860333204269409]
#
#313/313 [==============================] - 2s 6ms/step
#   - loss: 0.1168 - categorical_accuracy: 0.9845
#  [0.11677185446023941, 0.9845000505447388]
```

To create an adversarial example, we first select an image in the data set that we want to perturb. We then create an image of pure noise and add it to the original image. Both of these images are shown in Fig.2.

```
img = x_train[0:1]

#create noisy version of image
quantized_noise = np.round(np.random.normal(loc=0.0, scale=0.3,
                 size=img.shape) * 255.) / 255.
noisy_img = np.clip(img + quantized_noise, 0., 1.)
```

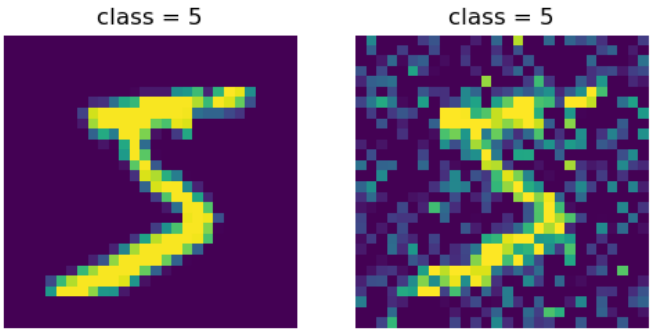


FIGURE 2. Selected image of the digit 5 and its noise perturbed version

We now create a CNN network for MNIST classification and then train it for 100 epochs. The training and test accuracy for the trained model is over 98%. We now take both images in Fig. 2 and classify them using the trained model. The correct classification would be 5 and this is indeed what our classifier declares for both the original and noise perturbed image.

```
#compile CNN network for MNIST classification
inputs = Input(shape=(28,28,1))
net = Conv2D(32, kernel_size=(3, 3),
```

```
                activation='relu')(inputs)
net = Conv2D(64, kernel_size=(3, 3),
                activation='relu')(net)
net = MaxPooling2D(pool_size=(2, 2))(net)
net = Dropout(0.25)(net)
net = Flatten()(net)
net = Dense(128, activation='relu')(net)
net = Dropout(0.5)(net)
outputs = Dense(10, activation='softmax')(net)

mnist_model = Model(inputs=inputs, outputs=outputs,
            name='classification_model')
mnist_model.compile(optimizer='nadam',
             loss='categorical_crossentropy',metrics=[categorical_accuracy])

#train MNIST classifer
earlyStop = EarlyStopping(monitor='val_categorical_accuracy',
             min_delta=0, patience=10, verbose=0,
             mode='auto',
             baseline=None, restore_best_weights=True)

mnist_model.fit(x_train, y_train,
          batch_size=128, epochs=100, verbose=0,
          validation_data=(x_test, y_test),
           callbacks=[earlyStop])

print(mnist_model.evaluate(x_train, y_train))
print(mnist_model.evaluate(x_test, y_test))

prediction = mnist_model.predict(img)[0]
predicted_class = np.argmax(prediction)

noisy_prediction = mnist_model.predict(noisy_img)[0]
predicted_noisy_class = np.argmax(noisy_prediction)
```

We now show how to construct an adversarial example that fools our MNIST classifier even though to the naked eye it looks very similar to the correctly classified images in Fig. 2. The basic idea is to create an adversarial model with the architecture shown in Fig. 3. This model has two inputs. The first is the selected image and the second input is noise. These two images are added together and then sent through our MNIST model. But what we are going to do is retrain this so it identifies the noise input patterns that

give rise to an incorrect classification. In particular, we use a categorical cross-entropy loss function and we freeze all of the weights except those generating the adversarial noise. The target for our loss function will be a "targeted" classification which is what we want the perturbed image to be classified as. In this example, we set that target to 9, so we are finding the noise input that when added to the original image causes it to be classified by the original MNIST model as 9, rather than 5.

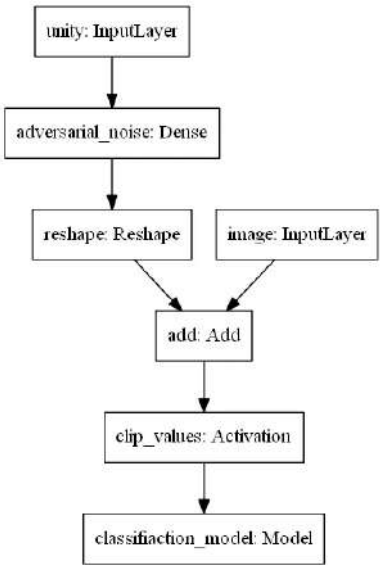


FIGURE 3. Adversarial Model Architecture

```
regularizer = l2(0.01)
loss_function = 'categorical_crossentropy'
model = mnist_model

image = Input(shape=(28,28,1),name='image')
one   = Input(shape=(1,), name = 'unity')
noise = Dense(28*28,activation=None,use_bias=False,kernel_initializer='random_normal',
             kernel_regularizer=regularizer,name='adversarial_noise')(one)
noise = Reshape((28,28,1), name='reshape')(noise)

net = Add(name='add')([noise,image])
net = Activation('clip',name='clip_values')(net)

outputs = model(net)
adversarial_model = Model(inputs=[image,one], outputs=outputs)
```

```
adversarial_model.layers[-1].trainable = False

adversarial_model.compile(optimizer='nadam', loss=loss_function, metrics=[categorical_accuracy])

adversarial_model.summary()

#target adversarial classification
target = 9                      #non-target
target_vector = np.zeros(10)
target_vector[target] = 1.

#train adversarial image
adversarial_model.fit(x={'image':img,'unity':np.ones(shape=(1,1))},
               y=target_vector.reshape(1,-1),epochs=10000,verbose=0,
                   callbacks=[checkpoint])
```

The resulting perturbed image resulting in an incorrect classification of "9" was obtained by taking the original 5 image and adding the adversarial noise obtained during training of the adversarial model. This image indeed is classified by the MNIST model as 9, but if we look at what that image is in Fig. 4, we see little difference from the original sampled image of 5. This example therefore shows how one can generate adversarial examples, which might be used later in retraining the classifier model so it cannot be fooled by such examples.

2. Deep Learning with Differential Privacy

Another security concern for community datasets regards the privacy of individual resident data when the entire dataset is being used to find out something about the community as a whole. The main concept used to address this issue with regards to databases is *differential privacy* [**Dwo08**]. With regard to databases, differential privacy ensures that the removal or addition of a single database item does not substantially affect the outcome of any analysis or query. This provides a mathematically rigorous way to manage the fact that any query to a statistical database may disclose some bits of information about individual entries.

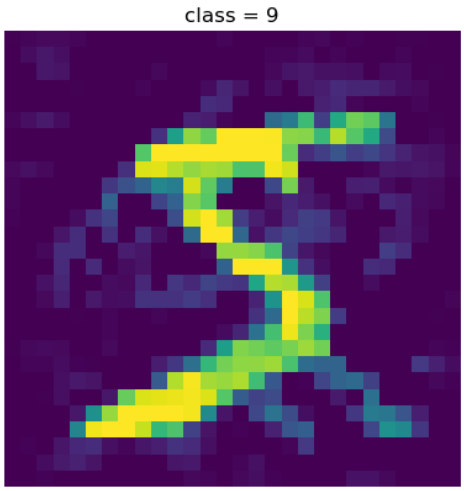


FIGURE 4. Misclassified Adversarial Example

Database privacy concerns also appear in deep learning. The neural network is trained on a large dataset and for deep networks, the model's over-parametrization means that some attributes of individual data entries may be disclosed by users of that model. These models should not disclose private information and one can develop algorithmic techniques for training [**ACG+16**] that provide ϵ-differential guarantees for individual dataset entries. This section first defines the notion of *differential privacy* for statistical databases and discusses deep learning methods [**ACG+16**] with differential privacy.

Let us consider a *statistical database*. A statistic is a quantity computed from a sample. We suppose a trusted curator gathers sensitive information from a large number of residents (the sample) with the goal of learning (and releasing) statistics for the entire population. The problem is to release this statistical information without compromising the privacy of any individual resident. We consider an *interactive* setting in which the curator sits between the users and the database. Queries from the users and responses to these queries may be modified by the curator to protect the privacy of the residents.

We define the notion of *differential privacy* in the context of this interactively curated statistical database. Intuitively, differential privacy ensure that the removal or addition of a single database item does not substantially affect the outcome of any statistical analysis. We can formalize this notion as follows [**Dwo08**]. Think of the database as a data matrix whose rows represent the attribute vectors of individual residents in the community. We define a randomized mechanism (a.k.a. algorithm) $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ that takes a dataset $D \in \mathcal{D}$ and randomly maps it to a statistic in $\mathcal{R}$. We say two datasets $D, D' \in \mathcal{D}$ are adjacent if they differ by one entry. We say this mechanism satisfies (ϵ)-differential privacy if for any two adjacent datasets $D, D' \in \mathcal{D}$ and for any subset $S \subset \mathcal{R}$ we have

$$\Pr\{\mathcal{M}(D) \in S\} \leq e^{\epsilon} \Pr\{\mathcal{M}(D') \in S\}$$

Any mechanism that satisfies this condition should address all concerns that any resident may have about leaking their personal information. This condition says that if a resident removes his/her data from the dataset, no output and so no consequences arising from that output will become more or less likely for the individual. For example, if the database were used by an insurer to determine whether or not to insure a resident, then whether or not the resident is in the database would have a negligible impact on whether the resident gets insured.

Achieving differential privacy means that we hide the presence or absence of a single individual. Consider the query "How many rows in the database satisfy property P?" The presence or absence of a single row can effect the answer by at most 1. So a differentially private mechanism for a query of this type can be designed by first computing the true answer and then adding random noise so that for any z, z' for which $|z - z'| = 1$, we have $\Pr\{z\} \leq e^{\epsilon} \Pr\{z'\}$. To see why this is so, consider any feasible response, r. For any m if m is the true answer to the query and the response is r then the random noise must have value $r - m$. Similarly, if $m - 1$ is the true answer and the response is r, then the random noise must have value $r - m + 1$. So for the response to be generated in a differentially private

manner it suffices for

$$e^{-\epsilon} \leq \frac{\Pr\{\text{noise} = r - m\}}{\Pr\{\text{noise} = r - m + 1\}} \leq e^{\epsilon}$$

For deep learning models, one seeks to protect the privacy of the training data. The neural network model is our "mechanism". One might attempt to do this by working with the final parameters after training. In general, however, one does not have useful tight bounds on how these weights vary with the training data. A more sophisticated approach aims to control the influence of the training data during the training process; specifically during the stochastic gradient descent (SGD) computation. These SGD algorithms [**ACG+16**] train a model with parameters θ by minimizing the empirical loss function $L(\theta)$. Each step of the SGD computes the gradient $\nabla_\theta L(\theta, x_i)$ for a random subset of examples, clips the ℓ_2 norm of each gradient, compute the average, add noise to protect privacy, and takes a step in the opposite direction of this average noisy gradient. At the end we also need to compute the privacy loss of the mechanism based on the information maintained by the curator.

3. Statistical Fairness in Classification

Fairness is a key consideration when machines use algorithms to make decisions. Fairness is defined with respect to population groups that are marginalized in a legal or societal manner as a result of demographic factors (gender, race, age) or socio-economic factors. In particular, it means that decisions or favorable outcomes provided to groups are independent of that group is marginalized or not. "Fairness", however, has a number of formal definitions. This section examines two statistical measures of fairness and presents a case study from [**BHN19**] illustrating their use.

Let us consider a classification problem where we wish to use data samples, X, to train a model that iners an unknown target Y. The guess that our model $h : X \to Y$ makes will be denoted as $\widehat{Y} = h(X)$. Throughout this

discussion we use capital letters to denote random variables and lower case letters to denote the values these variables take. We assume that $Y \in \{0, 1\}$ takes binary values where X can be a mixture of real-valued vectors and categorical data. The classical learning-by-example problem is to use a finite dataset $\{(x_k, y_k)\}$ of samples and labels to select h that maximizes the *accuracy* of the model. Namely we want to maximize $\Pr\left\{Y = \widehat{Y}\right\}$ or to minimize the classification error, $\Pr\left\{Y \neq \widehat{Y}\right\}$.

Accuracy and classification error are not the only criteria we might want to optimize. In some applications there may be a "cost" associated with correct and various incorrect classifications. So it is useful to introduce additional classification criteria whose predictions, $\widehat{Y}$, are conditioned on true label, Y.

Event	Condition	Description ($\mathbf{Pr}\{\text{event} \mid \text{condition}\}$)
$\widehat{Y} = 1$	$Y = 1$	True positive rate (TPR)
$\widehat{Y} = 0$	$Y = 1$	False negative rate (FNR)
$\widehat{Y} = 1$	$Y = 0$	False positive rate (FPR)
$\widehat{Y} = 0$	$Y = 0$	True negative rate (TNR)

An *optimal classifier* is one that minimizes the expected loss $\mathbb{E}\left\{L(\widehat{Y}, Y)\right\}$, or what we call the classification. For instance, if we choose a loss function that takes values $L(0, 1) = L(1, 0) = 1$ and $L(1, 1) = L(0, 0) = 0$, then the optimal classifier is

$$\widehat{Y} = h(X) = \begin{cases} 1 & \text{if } \Pr(Y = 1 \mid X = x) > 1/2 \\ 0 & \text{otherwise} \end{cases}$$

This classifier has the important property that it is essentially a threshold test applied to the function

$$r(x) = \Pr\{Y = 1 \mid X = x\} = \mathbb{E}[Y \mid X = x]$$

This function is an example of a *risk score* and the Neymann-Pearson lemma tells us that optimal classifiers can generally be written as a threshold test on this risk score function. The risk score shown above is a natural one and

is sometimes said to be *Bayes optimal* since it minimizes the square loss $\mathbb{E}(Y - r(x))^2$ among all possible real-valued risk scores, $r(X)$.

In the optimal classifier, we chose a threshold of $1/2$ to ensure we get the exact same number of false positives and false negatives. But there are applications where a false positive is significantly more costly than a false negative. In such applications we may want to move threshold away from $1/2$ to reflect the difference in cost. Each choice of threshold results in a different true positive rate and false positive rate and it is common to plot a threshold classifier's TPR vs FPR for various threshold to trace out a curve known as the Receiver Operating Characteristic (ROC). The "predictiveness" of a classifier may be measured by the area under the ROC curve (AUC) with an area of $1/2$ corresponding to random guessing.

We now to consider classifiers that discriminate on the basis of membership in specific groups of the population. US law recognizes certain *protected categories* that include race, set, religion, disability status, and place of birth. In many classification tasks the feature vectors, X implicitly or explicitly encode an individual's status in a protected category. Let use use the letter A to denote a random variable capturing various membership in these protected categories. We refer to A as a *sensitive attribute*. The choice of sensitive attributes may have profound consequences for population groups since the outcome of our classifier impacts how we allocate resources to these groups.

Some believe that by removing or ignoring sensitive attributes from the dataset, we can ensure the classifier is impartial. In a typical dataset, however, many features may be slightly correlated to the sensitive attribute. For example, membership in the website `pinterest` is slightly correlated to whether the member is female. While this correlation is too small to reliably deduce a given individual is female, when taken with a large number of other slightly correlated features, the likelihood of declaring the individual's gender becomes much greater. So simply removing "overt" features

associated with a sensitive attribute may not be sufficient to ensure impartiality. In training the classifier, it will identify how best to combine all available features in a way that maximizes its classification accuracy. As a result, simply removing "overt" attributes from the dataset will rarely be sufficient to ensure the classifier is "fair".

There are two main ways in which statistical non-discrimination criteria are formalized. Formally, these criteria are properties of the joint distribution of the sensitive attribution A, the target variable, Y, the prediction $\widehat{Y}$ (or score, R) and in some cases the features, X. This means we can decide if a non-discrimination criterion is satisfied by looking at the joint distribution of these random variables. We will look at two specific non-discrimination criteria; independence and separation.

Let us consider the sensitive attribute, A, and the score, R. We say these two variables satisfy *independence* if $A \perp R$ which means A and R are statistically independent. When applied to a binary classifier $\widehat{Y}$, independence corresponds to the condition

$$\Pr\left\{\widehat{Y} = 1 \mid A = a\right\} = \Pr\left\{\widehat{Y} = 1 \mid A = b\right\}$$

for all groups a and b. This is sometimes referred to as *demographic parity* or *group fairness*. If we think of event $\widehat{Y} = 1$ as "acceptance", the condition requires the acceptance rate to be the same in all groups. In some case, it is useful to relax the independence condition with a slack variable $\epsilon > 0$ such that

$$\Pr\left\{\widehat{Y} = 1 \mid A = a\right\} \geq \Pr\left\{\widehat{Y} = 1 \mid A = b\right\} - \epsilon$$

Independence has been studied in many papers on fairness in machine learning. It can be argued, however, that classifiers that satisfy independence can still be "unfair". To illustrate why this might be the case, image a company that hires individuals from group a in a highly diligent manner, say at rate p. On the other hand the company may be somewhat careless or less diligent in vetting individuals from group b while still maintaining the

same hiring rate, p. What it means is that those hired from group a may be much more qualified for hiring whereas those from group b may not. This apparent disparity in hiring practices can lead to discontent as those in group a feel they were not treated "fairly".

One way of addressing this issue is to acknowledge that there is a difference in accepting a true positive and true negative. In this case, the target variable, Y gives a sense that the individual with feature X is *qualified* to be accepted. So our notion of fairness needs to include both R,A, and the target Y. In particular, we say threse three random variables satisfy *separation* if $R \perp A \,|\, Y$. In other words R and A are conditionally independent with respect to Y. For a binary classifier, separation is equivalent to requiring that for all groups, a and b that

$$\begin{aligned} \Pr\left\{\widehat{Y} = 1 \,|\, Y = 1, A = a\right\} &= \Pr\left\{\widehat{Y} = 1 \,|\, Y = 1, A = b\right\} \\ \Pr\left\{\widehat{Y} = 1 \,|\, Y = 0, A = a\right\} &= \Pr\left\{\widehat{Y} = 1 \,|\, Y = 0, A = b\right\} \end{aligned}$$

Independence therefore quires that the true positive rate (TPR) is the same for both groups and the true negative reate (TNR) is also the same for both groups. These measures are sometimes referred to as *equal opportunity* (EO) measures.

This idea of equalizing error rates across groups has been controversial. Much of this controversy revolves about the fact that an optimal predictor need not have equal error rates in all groups. Specifically, when the propensity of positive outcomes differs between groups, an optimal predictor will generally have different error rates. In such cases, enforcing equality of error rates may lead to predictors that perform worse in some groups than it could be and way may then argue that this too is "unfair". One response to this criticism can be made by considering who bears the cost of misclassification. A violation of separation highlights the fact that different groups experience different costs of misclassification. There is a concern

that higher error rates coincide with historically marginalized and disadvantaged groups, thereby inflicting greater harm on these groups when they are misclassified.

A binary classifier that satisfies separation mush achieve the same TPR and same FPR for all groups. We can visualize this condition by plotting group-specific ROC curves as shown in Fig. **??**. We see the ROC curves of a score displayed for each group separately. The two groups have different curves indicating that not all tradeoffs between TPR and FPR are achievable in both groups. The tradeoffs that are achievable correspond to the intersection of the regions enclosed by the curves.

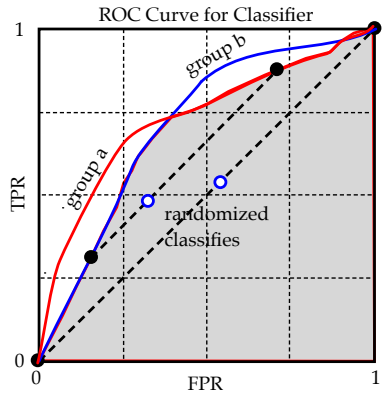


FIGURE 5. Randomized classifier from two classifiers

The highlighted region is the *feasible region* of tradeoffs that we can achieve in all groups. However, the thresholds that achieve these trade-offs are, in general, also group specific. In other words, the bar for acceptance varies by group which means these tradeoffs are not exactly on the ROC surface, but rather in the interior of the feasible region. Building such classifiers is usually done through randomization. To see this point, think about how one might realize trade-offs on the dashed line of the plot. Take one classifier that accepts everyone. This corresponds to a TPR and FPR of 1. Take another classifier that accepts no one, resulting in a TPR and FPR of 0. Now construct a third classifier that given an instance, X of features randomly picks and applies the first classifier with probability $1-p$ and the

second with probability p. This classifier achieves TPR and FPR rates of p and thus gives us one point on the dashed line in the plant. In a similar manner we could have selected any other pair of classifiers (say one from each of the ROC curves) and randomized between them to achieve a point in the area under the ROC curve along the dashed line.

Now that we have formally introduced two non-discrimination criteria, we ask how they might be achieved algorithmically. There are, in general, three different approach we might follow

- *Pre-processing* adjusts the features space to be uncorrelated with the sensitive attribute.
- *In-processing* works with non-discrimination criterion as a regularization term in the model training process.
- *Post-processing* adjusts learned classifiers (often using the randomization mechanism described above) to the resulting classifier is uncorrelated with the sensitive attribute

The three approaches have different strengths and weaknesses.

Pre-processing is a family of methods that transform the feature space into a representation that is independent of the sensitive attribute. This approach is agnostic to what we do with these features later on and so it can ensure independence under any training process on thenew space. The main criticism of pre-processing is that it requires complete access to the features and targets, which can impose a privacy issue.

In-processing introduces the non-discrimination criterion as a regularization constraint during model training. Its major issue is that it requires access to the raw training data and the training pipeline. In addition to this, the introduction of the regularization constraint may greatly slow down the convergence of the training algorithm. These are issues of particular concern in federated learning where information is passed back and forth between a central server and edge clients.

Post-processing refers to the process of taking a trained classifier and adjusting it using a randomization procedure to enforce fairness. It is applied after the training of each group's classifier. Post processing's advantage is that it works with trained classifiers and therefore does not need access to the raw data. In federated learning platforms, post processing simply transmits group statistics back and forth between the clients and server, which can be secured under a differential privacy guarantee. In addition to this, the optimal randomized classifier can usually be obtained by solving a linear program, thereby making the approach computationally efficient.

CHAPTER 9

Deep Reinforcement Learning

Reinforcement Learning (RL) [SB18] teaches an agent how to act in an unknown environment. We consider an agent that interacts with an external environment. The environment is a dynamical system that provides to the agent, at each time instant, a *state* and *reward* in response to the agent's current *action*. In general, the agent does not know the environment's reward function or dynamics. It must learn these things by seeing how the environment responds to the agent's actions. The agent then uses what it learns about the environment to identify an action policy that selects agent actions in response to the current environmental state such that the aggregate reward over a finite time horizon is maximized. Reinforcement learning can therefore be seen as *trial-and-error learning* since it acts and then learns from the positive and negative consequences of those actions. In addition to this RL is capable of learning from delayed rewards. This means that there is no reward until the agent reaches a desired final state. These two things, *trial and error learning* and *delayed rewards* are distinguishing characteristics of Reinforcement Learning.

The trial and error aspect of RL originated in the psychology of animal learning [SB81]. It was later recognized [Sut88] that RL could be equipped with a formal mathematical framework by thinking of it as *optimal control* of *Markov Decision Processes* (MDP). This formal framework for RL relied on the use of *dynamic programming* concepts [Bel54]. Dynamic programming is built around the concept of a *value function*, $V : X \times T \to \mathbb{R}$, that maps a dynamical system's state, $x \in X$, and time $t \in T$ onto the optimal cost (reward) required to move from the initial event (t, x) to a target event (T, x_T). The value function, $V(x, t)$, satisfies a recursive relation known as

the *Bellman equation*. The notion of reward in RL mapped easily to this notion of "value", so that one could interpret RL training as approximating the Bellman equation, thereby providing tremendous insight into how RL operates.

Neural network learning was originally applied to RL for learning how to play games such as Backgammon [**T+95**] and Atari video games [**MKS+13**]. But its value was quickly recognized for learning to control important real-life applications such as autonomous driving [**CXP+21**], datacenter cooling [**LWTG19**], traffic light control [**APK03**], healthcare [**YLNY21**], and robotics [**MKS+15, NKFL18**], to name a few. This chapter first examines how early RL algorithms attempted to solve the Bellman equation. We then examine a particularly influential deep learning approach to RL called the deep Q network (DQN) [**MKS+13**]. We close with looking at the use of deep neural networks in policy gradient approaches to RL.

1. Finite Markov Decision Processes

Markov Decision Processes consist of a decision maker interacting with an external environment. The decision maker is usually called an *agent*. The agent interacts with an *environment* by selecting an *action* that it takes in the environment. That interaction changes the environment's *state* in a stochastic manner. The environment then responds to this action by passing its *state* and a *reward* back to the agent. This interaction may be viewed as shown in Fig. 1. The problem is to determine the agent's *action policy* that maximizes the expected total discounted reward that the agent receives from the environment through its selection of actions.

A formal definition of an MDP is as a tuple, (S, A, p, r, S_0, S_K) where S is a finite set of environment states and A is a finite set of agent actions. We denote the state at time instant $k \in \mathbb{N}$ as s_k and the action at time k as a_k. The sets $S_0, S_K \subset S$ are called the initial and terminal state sets. The map $p : S \times A \to \mathcal{P}(S)$ maps the current state action pair $(s_k, a_k) \in S \times A$ onto

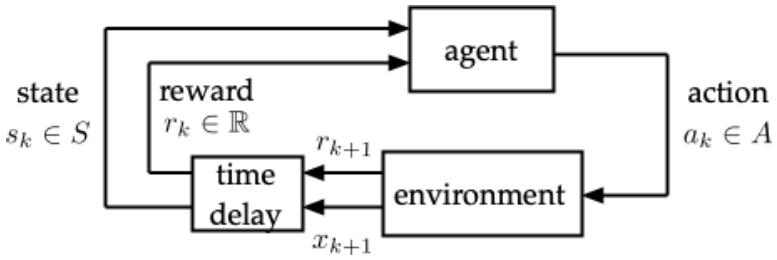


FIGURE 1. Agent-Environment Interaction

the next state's, s_{k+1}, through a conditional probability distribution function

$$p(y \,|\, x, a) = \Pr\left\{\mathbf{s}_{k+1} = y \,|\, \mathbf{s}_k = x, \mathbf{a}_k = a\right\}.$$

This conditional distribution defines the *dynamics* of the MDP. The other map, $r : S \times A \times S \to \mathbb{R}$ maps the current state-action-next-state triple (s_k, a_k, s_{k+1}) onto a numerical reward $r_{k+1} \in \mathbb{R}$.

The agent and environment interact over a sequence of time steps, $k = 0, 1, 2, 3, \ldots$. The environmental state at time 0 is in the initial state set S_0. At each time instant k, the agent selects an action a_k using a *policy*, $\pi : S \to \mathcal{P}(A)$. The policy uses the current state, s_k, to randomly select the current action, a_k, by sampling from the policy distribution $\pi(\mathbf{a}_k \,|\, \mathbf{s}_k = s)$. The set of all admissible policies will be denoted as $\mathbf{\Pi}$. The environment takes the agent's selected action and returns the environment's next state, $s_{k+1} \sim p(\cdot \,|\, s_k, a_k)$ and the next reward $r_{k+1} = r(s_k, a_k, s_{k+1})$. This interaction, therefore generates a sequence of state-action-reward triples (SAR) that we sometimes refer to as the agent's *trajectory*

$$(s_0, a_0, r_1) \to (s_1, a_1, r_2) \to (s_2, a_2, r_3) \to \cdots \to (s_{K-1}, a_{K-1}, r_K)$$

where the stopping time K occurs when the system state enters the terminal set, S_K for the first time. The terminal time, K, is a random variable called the *stopping time*.

Let us give a concrete example of a finite MDP. In a finite MDP the state, action, and reward sets are all finite. This means that the transition function $p(\cdot \,|\, s, a)$ can be represented as an indexed family of transition matrices. If

we assume that there are n states in $S = \{1, 2, \ldots, n\}$ then the transition matrix for action $a \in A$ can be written as

$$\mathbf{P}_a = \begin{bmatrix} p(1\,|\,1,a) & p(2\,|\,1,a) & \cdots & p(n\,|\,1,a) \\ p(1\,|\,2,a) & p(2\,|\,2,a) & \cdots & p(n,\,|\,2,a) \\ \vdots & \vdots & & \vdots \\ p(1\,|\,n,a) & p(2\,|\,n,a) & \cdots & p(n\,|\,n,a) \end{bmatrix}.$$

If we then let the current state, s_k, be drawn from a probability mass function that we represent as the row vector

$$\pi_k = \begin{bmatrix} \Pr(\mathbf{s}_k = 1) & \Pr(\mathbf{s}_k = 2) & \cdots & \Pr(\mathbf{s}_k = n) \end{bmatrix}$$

then the probability mass function for the next state s_{k+1} under action a will be

$$\pi_{k+1} = \pi_k \mathbf{P}_a.$$

.

It is common to use a family of labeled directed graphs to visualize the behavior of the MDP. We now use these directed graphs to illustrate a toy example.

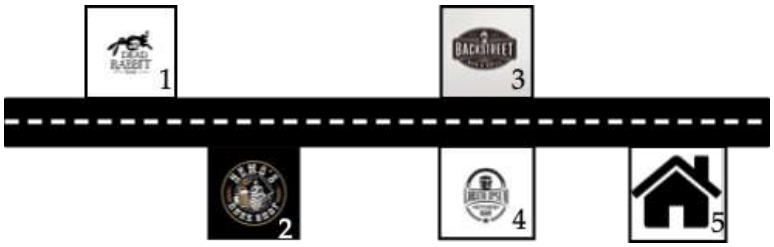


FIGURE 2. Map of Pubs

Example: We have a friend who travels between four pubs. We assume the streets connecting the pubs are as shown in Fig. 2. We can model our friend's night journey as a Markov decision process with the state space $S = \{1, 2, 3, 4, 5\}$. States $1 - 4$ are 4 different pubs. State 5 is home and the terminal state so that $S_k = \{5\}$. The initial state set is $S_0 = \{1, 2, 3, 4\}$, so our friend always starts in one of the pubs. After having a drink at the

pub, our friend makes a decision to either go to the next pub or to go home. We therefore have an action space $A = \{1, 2\}$ where $a = 1$ is go to the next pub and $a = 2$ is go home. This gives rise to two transition probability matrices, $\mathbf{P}_1$ and $\mathbf{P}_2$ defined with respect to the street map in Fig. 2.

The action of these transition matrices can be visualized using two directed graphs, one for each action's transition matrix. The nodes of the directed graph are the states of the MDP (the four bars and home). The graphs edges denote which locations are adjacent to each other along the roadway. If our friend decides to go to the next bar, he traverses the edge connecting two adjacent bars. If he decides to go home, then he either goes to the next bar that is one stop closer to home, or he goes directly home. The directed graphs associated with these two actions are depicted in Fig. 1. In this figure we have

$$\mathbf{P}_1 = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 \\ 1/3 & 0 & 1/3 & 1/3 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{P}_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The directed graph in Fig. 1 has labeled edges. Each edge, (i, j), from state i to state j is labelled with (p, r) where p is the probability, $p(j \mid i, a)$ and r is the reward $r(i, a)$. We label the terminal nodes in S_K with a terminal reward r_K of 5.

2. Optimal Actions and the Bellman Equation

The problem is to find an action policy, $\pi : S \to \mathcal{P}(A)$ that maximizes the expected total discounted reward an agent receives from the environment. The total expected discounted reward (a.k.a. *value*) for a given policy π is a function $V^\pi : S \to \mathbb{R}$ that takes values

$$V^\pi(s) = \mathbb{E}^\pi \left\{ r_K(s) + \sum_{k=0}^{K-1} \gamma^k r(s_k, \pi(s_k), s_{k+1}) \mid s_0 = s \right\}$$

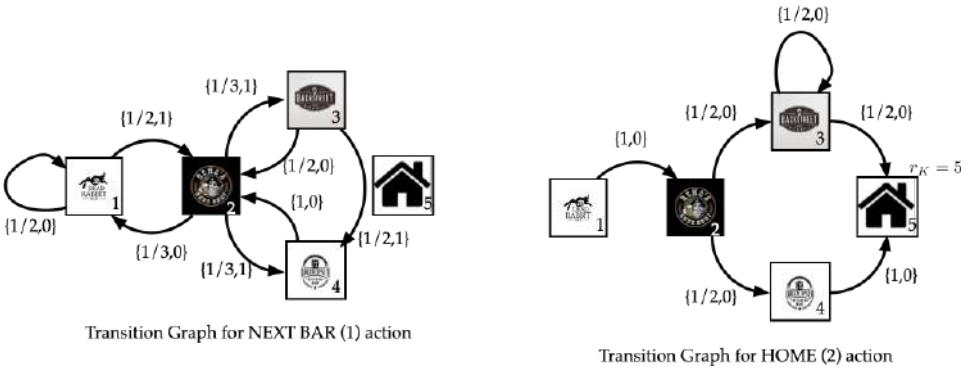


FIGURE 3. Pub Crawl

where K is the stopping time, $r_K(s)$ is the reward received for reaching the terminal state in S_K, and $\gamma \in (0, 1)$ is a *discounting* factor that discounts the rewards received later along the trajectory. The discounting factor plays an important role when we consider infinite horizon MDPs. For discounted infinite horizon problems the value function becomes

$$V^{\pi}(s) = \mathbb{E}^{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r(s_k, \pi(s_k), s_{k+1}) \,|\, s_0 = s\right\}.$$

We seek a policy, $\pi^* : S \to \mathcal{P}(A)$ that is *optimal* in the following sense

$$V^{\pi^*}(s) \geq V^{\pi}(s)$$

for all $s \in S$ and over all possible policies, $\pi \in \mathbf{\Pi}$. For simplicity we refer to the value function for the optimal policy as V^*, rather than V^{π^*}. Computing the optimal policy, π^*, is based on an optimal control method known as *dynamic programming*. Dynamic programming shows that the Value function satisfies an equation known as the *Bellman equation*

$$V^{\pi}(s) = \sum_{a \in A} \pi(a \,|\, s) \sum_{s' \in S} p(s' \,|\, s, a)\left[r(s, a, s') + \gamma V^{\pi}(s')\right]$$

for all $s \in S$. The Bellman equation provides the basis for developing recursive algorithms that can be used to algorithmically compute $V^{\pi}(s)$.

The value function can be used to determine the optimal policy π^*. In particular, we can say

$$V^*(s) = \max_{\pi \in \mathbf{\Pi}} V^\pi(s)$$

for all $s \in S$. The optimal policy, π^*, is then the policy that achieves this maximum. Directly finding π^* from $V^*(s)$, however, is difficult to do. It is more convenient to find the policy, π^*, from the state-action function or Q-function, $Q^\pi : S \times A \to \mathbb{R}$, for policy $\pi \in \mathbf{\Pi}$. The state-action function under π takes the value $Q^\pi(s, a)$ and it is the expected total reward received by the agent after it takes action a while in state s. This means that it can be directly related to the value function V^π through the equation

$$Q^\pi(s, a) = \sum_{s' \in S} p(s' \mid s, a) \left[r(s, a, s') + \gamma V^\pi(s') \right]$$

The optimal Q-function is then

$$Q^*(s, a) = \max_{\pi \in \mathbf{\Pi}} Q^\pi(s, a).$$

But because $a \sim \pi(a \mid s)$ we can readily see that the value of the optimal policy decision when the current state is s will be

$$\pi^*(s) = \arg\max_a Q^*(s, a).$$

thereby allowing one to determine the optimal policy once the optimal Q-function has been determined.

If the environment's dynamics are completely known, then the Bellman equation is a system of simultaneous linear equations where the number of unknowns equals the cardinality of the state space S. Directly solving this system of linear equations is difficult due to the high cardinality of the state space. So we resort to successive approximation methods to find the value function. In particular, consider a sequence $\left\{ \widehat{V}_\ell^\pi(s) \right\}_{\ell=0}^\infty$ of functions $\widehat{V}_\ell^\pi : S \to \mathbb{R}$ that are approximations to the true value function V^π computed

through the recursion

$$\begin{aligned}\widehat{V}^{\pi}_{\ell+1}(s) &\stackrel{\text{def}}{=} \mathbb{E}^{\pi}\left\{r(s_k,\pi(s_k),s_{k+1})+\gamma\widehat{V}^{\pi}_{\ell}(s_{k+1})\,|\,s_k=s\right\}\\ &= \sum_a \pi(a\,|\,s)\sum_{s'\in S}p(s'\,|\,s,a)\left[r(s,a,s')+\gamma\widehat{V}^{\pi}_{\ell}(s')\right]\end{aligned}$$

for all $s \in S$. Clearly the actual value function, V^{π}, is a fixed point for the recursive update. This recursion is convergent provided the discounting rate $\gamma < 1$. The successive approximation algorithm is known as the *iterative policy evaluation*.

Once we use this iteration to compute V^{π} for a given policy, π_0, we need to perturb the policy to find a better one. So let us assume for some state s that instead of picking the action $\pi_0(s)$, we pick an alternate action $a \neq \pi_0(s)$. After that we simply continue using the original policy π_0. The state-action value from s under this perturbed policy would be

$$\begin{aligned}Q^{\pi_0}(s,a) &\stackrel{\text{def}}{=} \mathbb{E}^{\pi}\left\{r(s,a)+\gamma V^{\pi_0}(s_{k+1})\right\}\\ &= \sum_{s'\in S}p(y\,|\,s,a)\left[r(s,a,s')+\gamma V^{\pi}(s')\right].\end{aligned}$$

The key thing is whether this is greater than or less than $V^{\pi_0}(s)$. Clearly since we can use the above equation to compute $Q^{\pi_0}(s,a)$, when the policy is perturbed at time k by using a instead of $\pi_0(s)$. Since there are a finite number of actions, we compute this value for each a and come up with an improved policy that uses an a that maximizes $Q^{\pi_0}(s,a)$. So we end up with a *greedy* policy of the form

$$\begin{aligned}\pi_1(s) &= \arg\max_a Q^{\pi_0}(s,a)\\ &= \arg\max_a \sum_{s'\in S}p(s'\,|\,s,a)\left[r(s,a,s')+\gamma V^{\pi}(s')\right].\end{aligned}$$

Once a policy has been improved using V^{π_0} to obtain π_1, we recompute V^{π_1} and improve it again to obtain a better π_2. We therefore obtain a sequence of monotonically improving policies and value functions

$$\pi_0 \stackrel{\text{eval}}{\rightarrow} V^{\pi_0} \stackrel{\text{improve}}{\rightarrow} \pi_1 \stackrel{\text{eval}}{\rightarrow} V^{\pi_1} \stackrel{\text{improve}}{\rightarrow} \pi_2 \stackrel{\text{eval}}{\rightarrow} \cdots \stackrel{\text{improve}}{\rightarrow} \pi^* \stackrel{\text{improve}}{\rightarrow} V^*. \tag{43}$$

Because a finite MDP only has a finite number of policies, this process must converge to an optimal policy and an optimal value function after a finite number of recursions. This search strategy for the optimal policy is called the *Policy Iteration*.

Unfortunately, the policy iteration may take many recursions before stopping at an optimal policy. The reason is that the evaluation step is itself a recursive algorithm that may take many steps to converge. One can improve the efficiency of this algorithm by truncating the evaluation phase early. In particular, one can truncate after just one recursive step and then use the resulting approximate value function to do policy improvement. This approach is called the *Value Iteration* and in practice it reduces the overall complexity of the algorithm.

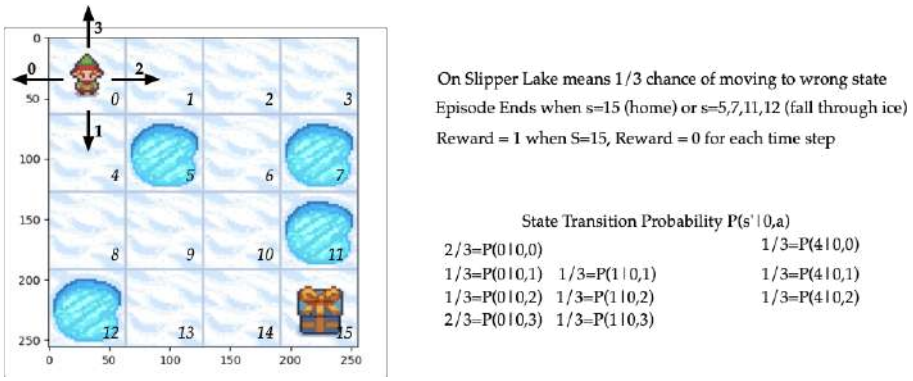


FIGURE 4. `Gym`'s Frozen Lake Environment

We will now demonstrate the Value Iteration on a particular MDP problem using OpenAI's Gym environment. In particular, we will examine the "frozen lake process" shown in Fig. 4. This figure shows the state space $S = \{0, 1, 2, \ldots, 15\}$, as a 4 by 4 grid of squares representing locations on a frozen lake. We assume a discrete-time process where each decision epoch has a period of 1 (i.e. we make a decision after every step). There are 4 actions the agent can take at each epoch, move North (3), East (2), South (1), or West (0). Because the frozen lake is slippery, the state we reach after taking a given action is random. In particular, we assume the transition

probability $p(s' \mid s, a)$, shown in Fig. 4. So, for example, if we were in state 0) and take action 3 to move South (1), we have an equal probability of $1/3$ to either say in state 0, move to state 1 or move to state 4. It is the slipperiness of the icy lake that injects the uncertainty into the outcome of the action we take.

The frozen lake problem is a finite-horizon problem. But rather than specifying the horizon time in terms of a fixed final time, K, we determine that horizon in terms of the first time when the state enters a terminating set of states. In particular, the states 5, 7, 11, 12, and 15 are terminating states. When a system enters a terminating state, the process stops and we restart our agent from the beginning state, 0. In our case, the states, 5, 7, 11 and 12 represent holes in the ice (i.e. the MDP terminates because the agent has fallen through the surface of the lake). State 15, on the other hand is a desired terminal state that the agent wants to reach. The terminal reward function, $r_K : S_K \to \mathbb{R}$, will therefore take different values depending upon which terminal state we reach. In particular, we have $r_K(s_K) = 0$ if the terminal state is on a hole in the ice (5, 7, 11, or 12) and the reward, r_K, is 1 if the terminal state is 15. The other rewards for the transitions taken to get to the terminal state $r(s, a)$ are all all zero for the other states. This means, therefore, that our agent will only get rewarded if it reaches the desired terminal state. So the agent's receives *delayed rewards* because it is only rewarded for "successful" runs.

We simulate MDP's using a toolkit called `Gym` that was developed by OpenAI. Let us suppose we are developing an agent that autonomously drives a car. We cannot train our agent in the "real-world" because RL is based on trial-and-error and since errors involve a "crash", we don't want to do this with real-life automobiles. So we use a software environment that simulates the behavior of our process and use this simulation environment to implement RL's trial-and-error approach to learning. The `Gym` environment is just one convenient Python library for developing such simulations.

The Frozen Lake Environment described in Fig. 4 has already been included in `Gym`. The following script loads this environment into a notebook.

```
import gym
import numpy as np
import matplotlib.pyplot as plt

env = gym.make('FrozenLake-v1', render_model='human')
```

We are going to simulate the Frozen Lake environment using a decision policy, π, that randomly samples the action space in a uniform manner. We run this for 100 steps and reset the environment every time the state reaches one of the terminating states or makes the maximum number (100) of allowed steps. We can render the results on the screen to show our agent moving across the frozen lake.

```
num_steps = 100
obs = env.reset()
obs = obs[0]

for step in range(num_steps):
     action = env.action_space.sample()
     obs, reward, terminate, truncated, info = env.step(action)

     env.render

     if terminate:
          env.reset()
env.close()
```

So let us demonstrate the Value Iteration on the Frozen Lake environment. The algorithm will be implemented through the function `value_iteration`. this function takes the Gym environment, `env`, the discounting factor, `gamma` as inputs. It returns the value function, `value`, the optimal policy, `policy`, and the number of iterations, `iter`, needed for convergence.

```
def value_iteration(env, gamma):
     num_iterations = 10000
     threshold = 1e-20
```

```
    value = np.zeros(env.observation_space.n)
    for iter in range(num_iterations):

         #value iteration
         value_new = np.copy(value)
         for s in range(env.observation_space.n):
              for a in range(env.action_space.n):
                   Q[s][a] = sum([prob*(r+gamma*value_new[s_])
                                  for prob, s_, r, _ in env.P[s][a])
         value[s] = max(Q[s,:])

         #policy improvement
         policy = np.zeros(env.observation_space.n)
         for s in range(env.observation_space.n):
              policy[s] = np.argmax(np.array(Q[s,:]))

        #Termination Condition
        if (np.sum(np.fabs(value_new-value))<=threshold):
             break
    return value, policy, iter
```

This function first sets the maximum number of iterations (`max_iter`) and the stopping threshold `threshold`. It then performs one step of the recursion for the state-action function for each action a

$$Q^{\pi}(s,a) = \sum_{s' \in S} p(s' \mid a, s) \left[r(s, \pi(s), s') + \gamma V^{\pi}(s') \right].$$

The state value function is then obtained by taking the maximum of $Q(s,a)$ over all a

$$V^{\pi}(s) = \max_{a} Q^{\pi}(s,a).$$

After computing this approximation to V^{π}, the algorithm performs a policy improvement step. In our code, we use V^{π} to compute the state-action function and then select the new policy π', that maximizes $Q^{\pi}(s,a)$

$$\pi'(s) = \arg\max_{a} Q^{\pi}(s,a)$$

We then repeat these two steps until we either reach `max_iter` limit or we reach the accuracy threshold $|V^{k+1} - V^{k}| <$ threshold. The following script runs the Value Iteration algorithm for the Frozen Lake environment

and then shows how this policy behaves versus a random policy. We see that the Value Iteration converged to a tolerance of 10^{-20} after $1,372$ iterations. The resulting values and optimal actions are shown in Fig. 5

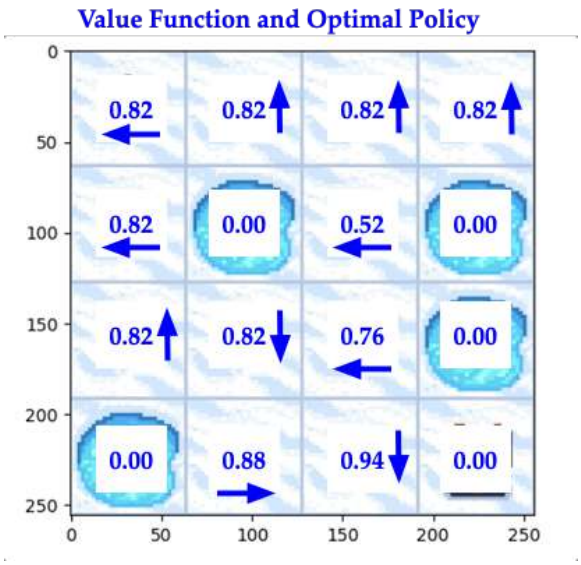


FIGURE 5. Optimal Policy and Value Function

```
value, policy, iter = value_iteration(env, 1.0)
print(value)
print(policy)
print(iter)

#[0.82352941 0.82352941 0.82352941 0.82352941 0.82352941 0.
#0.52941176 0.         0.82352941 0.82352941 0.76470588 0.
# 0.         0.88235294 0.94117647 0.         ]
#[0. 3. 3. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]
#1372
```

We used the Value Iteration to find a policy for our stochastic Frozen Lake environment, but we still need to evaluate how well it actually performed. So we used the following script to generate 1000 episodes using the optimal policy and then rerun for 1000 episodes using a random policy.

```
episodes = 1000
nb_success = 0
num_steps = 1000
```

```
value_iteration_policy_active = True
for iter in range(episodes):
    if iter%25==0:
        print(str(iter)+"-",end="")
    state = env.reset()[0]
    for iter in range(num_steps):
        if value_iteration_policy_active:
            action = int(policy[state])
        else:
            action = env.action_space.sample()
        new_state, reward, terminate, truncate, info = env.step(action)
        state = new_state
        if terminate:
            nb_success += reward
            break
print(f"Success rate = {nb_success/episodes*100}%")
```

For the optimal policy we had a success rate of 82%, whereas the random policy only had a success rate of 3%. So clearly, the value iteration was able to dramatically improve the agent's likelihood of safely reaching the desired terminal state.

3. Learning Optimal Action Policies

The preceding section determine the optimal policy for an MDP assuming the agent already knows the environment's state transition probability. In many cases, the agent does not have this prior knowledge and must learn its optimal action policy by observing how the environment responds to the agent's actions. This section discusses three methods used to learn an agent's *optimal policy*; Monte Carlo methods, Temporal-Difference learning, and Q-learning. All of these methods rely on a trial-and-error approach to solving the Bellman equation, the differences lie in how they generate experimental trials and how they use the outcomes of those trials. These algorithms all learn the state or state-action value function and for this reason they are called *value gradient* algorithms.

3.1. Monte Carlo Methods: Monte Carlo (MC) methods are computational methods that use simulations to estimate the likelihood of uncertain events. MC methods build a simulation model of the environment and then execute that simulation model starting from multiple initial conditions to generate several possible process trajectories. Each trajectory is called an *episode* and we then average these episodes to estimate the fundamental statistics of the process.

We use MC methods to solve MDP problems by first constructing a simulation model of the environment, fixing the agent's action policy, π, and then generating a number of different episodes for the environment under the fixed policy. We use these episodes to estimate the value function and then use that estimated value function to update the policy. We then repeat this episodic process until we converge to the optimal action policy.

Formally, let $V^{\pi}(s)$ denote the value of the environmental state $s \in S$ under a fixed policy $\pi : S \to A$. We use our simulation model to generate a set of *episodes* obtained by following π and passing through the state s. Each occurrence of state s in an episode is called a *visit* to s. The state s may be visited multiple times in the same episode. The *first visit* to s refers to the first time an episode passes through state s. The *first-visit* MC method estimates $V^{\pi}(s)$ as the average of the total reward received by the agent following its first visit to s. Alternatively, we could have averaged the total reward of the agent after *every visit* to state s. This would be the *every-visit* MC method. Both methods converge to $V^{\pi}(s)$ as the number of visits to s goes to infinity. For the first-visit MC method this is an obvious consequence of the law of large numbers. The every-visit MC's convergence to V^{π} is more difficult to establish, but one can show that it has a faster (quadratic) convergence rate than the first-visit method.

The obvious idea is to use either a first or every visit MC method to estimate $V^{\pi}(s)$ for a given policy and then apply policy improvement. The problem with this, of course, is that it takes an intractably long time for this

approach to converge to the optimal action policy. Another potential issue is that our random generation of the episodes may not fully explore the state space for a complete evaluation of the value function. In other words, it is possible that our MC simulation only explores an easily accessible portion of the state space, and that early bias in episode generation biases our future action policies from exploring other states that may lead to more optimal paths. For this reason, engineers rarely (if ever) use pure MC methods to solve MDP problems. Instead we use methods that combine Monte Carlo methods with the dynamic programming ideas of the preceding section.

3.2. Temporal-Difference (TD) Learning: Temporal difference (TD) learning combines ideas from Monte Carlo methods and Dynamic Programming. Like MC methods, TD methods use randomly generated episodes to learn how the environment responds to agent actions. But TD methods update an agent's estimate of the value function using the Bellman equation, rather than simply averaging the total rewards under a fixed policy. This subsection describes two such TD learning algorithms; SARSA [**RN94, Sut95**] and Q-learning [**WD92**]. The differences between them are described below.

SARSA Algorithm: The MC methods described in the preceding subsection first *predict* the value function for a given policy, and then improve the policy. Let $\{s_k\}_{k=0}^{\infty}$ denote the episode generated under policy π. Every-visit MC methods would approximate the value function from state s_k in the episode as

$$\widehat{V}^{\pi}(s_k) \leftarrow \widehat{V}^{\pi}(s_k) + \alpha \left[\sum_{\ell=0}^{\infty} \gamma^{\ell} r(s_{k+\ell}, \pi(s_{k+\ell}), s_{k+\ell+1}) - \widehat{V}^{\pi}(s_k) \right].$$

where α controls the size of the update. This update would be computed at the end of an episode because we have to wait until the end to compute total reward, $g_k = \sum_{\ell=0}^{\infty} \gamma^{\ell} r(s_{k+\ell}, \pi(s_{k+\ell}), s_{k+\ell+1})$. TD learning, on the other hand uses the Bellman equation to estimate $\widehat{V}^{\pi}(s_k)$ on a step by step basis

as we generate the episode. So the simplest type of TD value prediction algorithm would estimate the value function at the kth state, s_k, in the episode as

$$\begin{aligned} \widehat{V}^{\pi}(s_k) &\leftarrow \widehat{V}^{\pi}(s_k) \\ &\quad +\alpha\left[r(s_k,\pi(s_k),s_{k+1})+\gamma\widehat{V}^{\pi}(s_{k+1})-\widehat{V}^{\pi}(s_k)\right]. \end{aligned} \tag{44}$$

This update would be computed after the agent using the action $\pi(s_k)$ at time k has received the updated state, s_{k+1}, and reward $r_{k+1} = r(s_k, \pi(s_k), s_{k+1})$ from the environment. So rather than approximating $\widehat{V}^{\pi}$ after the episode has finished, the approximation is computed as we are generating the episode.

The TD prediction algorithm in equation (44) can be used to find the optimal policy. We do this by following the similar strategy portrayed in equation (43) for the Policy Iteration. We could use the TD-prediction equation (44) to compute estimates of the value function and then use that value function to improve the policy. The only issue is that the original Policy Iteration improved its policy through the Q^{π} (state-action value function), rather than the state value function, V^{π}. Estimating the Q^{π}-function is very similar to estimating the V^{π}-function. To see this recall that an episode consists of an alternating sequence of (state,action) pairs and rewards

$$(s_k, a_k) \rightarrow r_{k+1} \rightarrow (s_{k+1}, a_{k+1}) \rightarrow r_{k+2} \rightarrow (s_k + 2, a_{k+2}) \cdots$$

Now consider sequences from state-action pair to state action pair, and learn the value of the state-action pairs. Formally, this is identical to the earlier TD-prediction equation, so we can simply rewrite equation (44) in terms of an approximation $\widehat{Q}^{\pi}$ function to the state action function Q^{π}.

$$\begin{aligned} \widehat{Q}^{\pi}(s_k, a_k) &\leftarrow \widehat{Q}^{\pi}(s,a_k) \\ &\quad +\alpha\left[r_{k+1}+\gamma\widehat{Q}^{\pi}(s_{k+1},a_{k+1})-\widehat{Q}^{\pi}(s_k,a_k)\right] \end{aligned} \tag{45}$$

where $a_k = \pi(s_k)$ and $r_{k+1} = r(s_k, a_k, s_{k+1})$. The variables used in equation (45) are

$$(s_k, a_k, r_{k+1}, s_{k+1}, a_{k+1})$$

and so the Q^π prediction in equation (45) forms the basis of a version of TD-learning called the SARSA algorithm[**RN94, Sut95**]. .

Since the SARSA algorithm computes $\widehat{Q}^\pi$ directly, we can readily use $\widehat{Q}^\pi$ to improve the policy π by *greedily* selecting actions that maxmize $\widehat{Q}^\pi(s,a)$. This is called the *greedy-SARSA* algorithm. It is also called an *on-policy* algorithm because the policy it is learning is also used at the same time to control the process and generate the training data. Unfortunately, greedy algorithms may not fully explore the state space because of their preference for always maximizing the approximated Q function. To avoid this issue we often employ an ϵ-greedy version of the SARSA algorithm that has the agent randomly select an arbitrary action $a \in A$, rather than the optimal action, with a probability of ϵ. The ϵ-greedy policy is seen as providing agents with the capacity to switch between *exploration* of the state space (i.e. selecting the random action) and *exploitation* of the prior experience embodied in the $\widehat{Q}^\pi$ function (i.e. picking the greedy action). ϵ-greedy SARSA algorithms can be shown to converge with probability 1 to the optimal Q^* function as long as all state-action pairs are visited an infinite number of times and it converges to the optimal policy provided $\epsilon \to 0$ at a suitable rate.

Q-Learning: One of the early breakthroughs in reinforcement learning was the development of a TD-algorithm known as Q-learning [**WD92**]. Q-learning was an off-policy algorithm that improves a policy not actually being used to generate the training episodes. For instance an algorithm that uses a policy that randomly selects actions to learning the optimal Q function is an example of an off-policy algorithm. Once the optimal Q function has been determined, one would then use it to derive the optimal policy that would subsequently be used to control the process. The original off-policy

Q-learning algorithm [**WD92**] relies on the following recursion to approximate the Q function

$$\begin{aligned} \widehat{Q}(s_k, a_k) &\leftarrow \widehat{Q}(s_k, a_k) \\ &\quad +\alpha \left[r_{k+1} + \gamma \max_a \widehat{Q}(s_{k+1}, a) - \widehat{Q}(s_k, a_k) \right] \end{aligned} \tag{46}$$

where $a_k = \pi(s_k)$ is generated by some policy. In this recursion, the learned state action function, $\widehat{Q}$, is trying to directly approximate Q^*, rather than Q^π for a specific policy π being used to generate the episodes. This approach to TD-learning greatly simplified the convergence analysis of the algorithm, thereby providing the first convergence proofs for TD-type algorithms[**WD92**]

We are now going to illustrate Q-learning on the Frozen Lake environment. As before, we start by initializing `Gym`'s Frozen Lake Gym environment.

```
import gym
import random
import numpy as np
env = gym.make("FrozenLake-v1", is_slippery=True, render_mode="ansi")
env.reset()
```

We then run the Q-learning algorithm in equation (46) using an ϵ-greedy policy with $\alpha = 0.5$, $\gamma = 0.9$ and $\epsilon = 1.0$ for 1500 episodes. So this may be viewed as a partially on-policy version of Q-learning, rather than a fully off-policy version. The use of the on-policy version does result in much faster convergence of the algorithm. After each episode we subtract 0.001 from ϵ so that as time goes on we begin to taking policy actions in a more greedy manner.

```
# We re-initialize the Q-table
qtable = np.zeros((env.observation_space.n, env.action_space.n))
reward_lst[]

# Hyperparameters
episodes = 1500      # Total number of episodes
num_steps = 1000
alpha = 0.5
```

```
gamma = .9
epsilon  = 1.0
epsilon_decay = .001

# Training
for iter in range(episodes):
    s = env.reset()[0]
    done = False
    if iter%100==0:
      print(iter)

    for steps in range(num_steps):
        #epsilon-greedy
        if random.uniform(0,1) < epsilon:
             a = env.action_space.sample()
        else:
             a = np.argmax(qtable[s])
        #take a step
        s_new, reward, term, trunc, info = env.step(a)
        # Update Q(s,a)
        qtable[s, a] = qtable[s, a] + alpha *
           (reward + gamma * np.max(qtable[s_new]) - qtable[s, a])
        s = s_new

        # update success table

        if term:
            epsilon = max(epsilon - epsilon_decay, 0)
            if reward:
                print(str(iter)+' success - ',end="")
            break

     reward_list.append(reward)
```

The following script outputs the learned Q-function, value function and the optimal policy. The value function and optimal policy are shown on the left side of Fig. 6. This figure shows that the value function determined by Q-learning is not the same as that obtained using the Policy Iteration. The policy iteration value function (left hand side) is the true value function since it was obtained using our prior knowledge of the state transition probabilities. If we had run Q-learning for many more iterations, it should

converge to the true value function, but that time might be prohibitively long. Moreover, in looking at the Q-learning value function, we see that it really only identified a single path around the first ice hole in state 5. What this shows is that we simply did not run the learning algorithm long enough to fully explore the state space.

```
print('==========================================')
print('Q-table after training:')
print(qtable)

value = np.max(qtable,1)
print('==========================================')
print('Value Function after training:')
print(value)

print('==========================================')
print('Optimal Policy after training')
policy=np.argmax(qtable,1)
print(policy)

#==========================================
#Q-table after training:
#[[6.89339870e-02 7.77818251e-03 7.87500113e-03 7.57177488e-03]
# [5.57130344e-03 5.15946469e-03 6.46200431e-03 4.71729734e-02]
# [6.04461778e-03 6.41147999e-03 6.48856569e-03 2.63501542e-02]
# [4.20343071e-03 3.49276892e-03 3.17097517e-03 1.30104292e-02]
# [1.39291968e-01 7.88339371e-03 7.20365848e-03 6.53618787e-03]
# [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
# [1.82348555e-03 4.69839847e-04 3.35633562e-02 2.44303585e-03]
# [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
# [1.61179819e-02 1.22447986e-02 1.26201540e-02 1.80753711e-01]
# [2.07325529e-02 3.01033340e-01 1.28806604e-02 1.57001918e-02]
# [9.75764351e-02 9.51369934e-03 9.17115126e-03 7.78605619e-03]
# [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
# [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
# [7.01654757e-02 5.72436835e-02 3.45159006e-01 7.97923902e-02]
# [1.67523873e-01 1.50890356e-01 6.89571115e-01 1.68932401e-01]
# [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
#==========================================
#Value Function after training:
#[0.06893399 0.04717297 0.02635015 0.01301043 0.13929197 0.
# 0.03356336 0.         0.18075371 0.30103334 0.09757644 0.
# 0.         0.34515901 0.68957111 0.        ]
```

```
#=========================================
#Optimal Policy after training
#[0 3 3 3 0 0 2 0 3 1 0 0 0 2 2 0]
```

After learning the optimal Q function, we determine the actual action policy and then evaluate how well it did. That evaluation generated 1000 episodes and counted the percentage of times the agent successfully reached the destination. The output from this script showed a 75% success rate, which is "close", though certainly less than the 82% optimal success rate computed using the value iteration. So that even though our Q-learning session did not learn the optimal value function, it did learn enough to significantly outperform a purely random action policy.

```
episodes = 1000
nb_success = 0


for iter in range(episodes):
    delt = int(episodes/10)
    if iter%delt==0:
        print(str(iter)+"-",end="")
    s = env.reset()[0]

    while not done:
        a = np.argmax(qtable[s])

        s_new, reward, term, trunc, info = env.step(a)

        s = s_new
        if term:
            nb_success += reward
            break

#0-100-200-300-400-500-600-700-800-900-
#Success rate = 74.9%
```

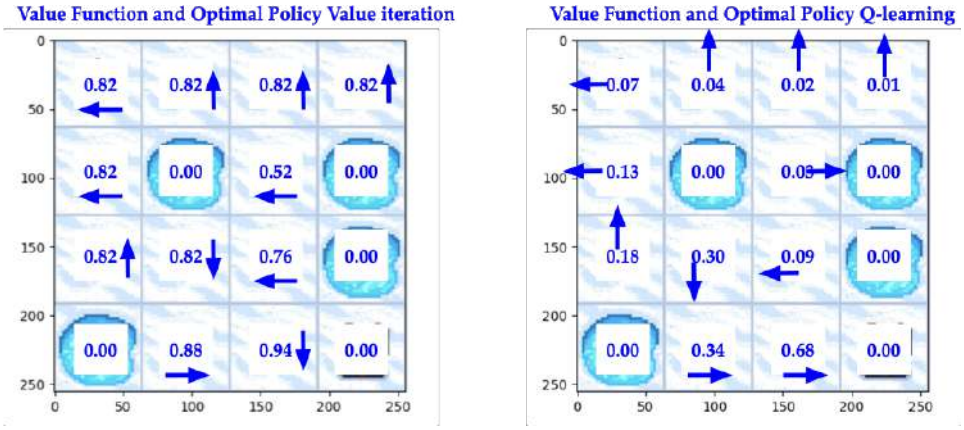


FIGURE 6. Value function and policy obtained using Value-Iteration for the Frozen Lake environment (right) and Q-learning (left)

4. Deep Q Learning (DQN)

Notice that the preceding sections discussed Reinforcement Learning without ever referring to neural networks. Deep Reinforcement Learning (DRL) uses a deep neural network to approximate the optimal state-action value function, Q^*. In regular Q-learning, the Q^* function is represented as a table. Deep Q learning represents the Q^* function as a neural network that maps the environment's state, s, onto the Q value for each action in the action space. A deep Q network agent or DQN agent is one that uses such a neural network to estimate Q^*.

Neural networks have been used in Reinforcement Learning since 1995 [**T+95**] where a neural network was used with TD-learning to play backgammon. This early demonstration obtained impressive results by handcrafting features that simplified the training problem. The significant advance that sparked recent interest in deep reinforcement learning appeared in 2013 [**MKS+13**]. That paper demonstrated the use of convolutional neural networks called a DQN agent that did not require prior feature engineering. The DQN agent was trained to play the entire suite of Atari video games.

That training was done without prior feature engineering and taught itself features that could be used for all games made by Atari. The DQN agent took raw input images from the game's video screen as input and used that to select game policies.

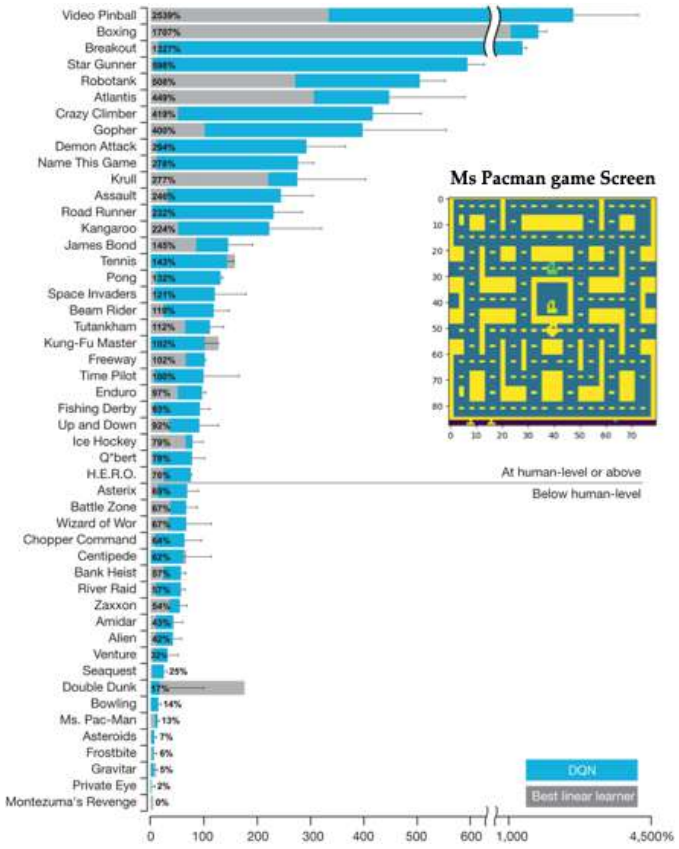


FIGURE 7. Performance of DQN on Atari Video Games [MKS+13]

The DQN agent in [MKS+13] out performed professional human game testers on many of the Atari games. Fig. 7 was taken from [MKS+13] and lists those games on which DQN outperformed professional human testers. The screen image in Fig. 7 showed the raw input image used by the DQN

agent in playing Ms. Pacman. The figure's performance measure was expressed as a percentage

$$100 \times \frac{\text{DQN score - random play score}}{\text{human score - random play score}}.$$

The figure shows that DQN often performed better than professional game testers.

The DQN agent can be instantiated as a Python class object to facilitate its interaction with `Gym`. The remainder of this section shows how this might be done for the `FrozenLake` environment. The DQN agent is trained using past agent interactions with the environment that have been saved into a first-in first-out *replay buffer*. In particular, each interaction is represented as a tuple

$$(s_k, a_k, r_{k+1}, s_{k+1})$$

that represents that environmental reward and state, r_{k+1} and s_{k+1}, returned to the DQN agent for taking action a_k when the environment is in state s_k. This tuple is popped onto a FIFO queue (the replay buffer) of finite length. The DQN agent then uses a randomly selected batch of tuples in the replay buffer for a single step of the backpropagation algorithm updating the agent's weights.

The DQN agent takes the current state, s, and outputs the Q^* value for each action $a \in A$. This means that the model is trained to solve a multivariate regression problem, so the loss function is the mean squared error (MSE) between the optimal Q function, Q^*, and the model's estimated Q value, $\widehat{Q}$. The problem, of course, is that we do not really know the optimal Q-function. From Bellman's equation, we know that the optimal Q function satisfies

$$Q^*(s,a) = \mathbb{E}_{s' \in s}\left[r(s,a,s') + \gamma \max_{a' \in A} Q^*(s',a')\right].$$

The expectation may be approximated by the sample mean evaluated over a minibatch of size M drawn from the replay buffer and rather than using

$Q^*(s', a')$ on the righthand side of the equation we use the current estimated state-action function $\widehat{Q}$ in the above equation. This means that the loss function to be minimized by backpropagation has the following form

$$L(w) \quad = \quad \frac{1}{M}\sum_{k=1}^{M}\left(r_{k+1} + \gamma \max_{a' \in A} \widehat{Q}_w(s_{k+1}, a) - \widehat{Q}_w(s_k, a_k)\right)^2$$

where w represents the trainable parameters of the neural network and $\widehat{Q}_w$ denotes the state-action valued predicted by the neural network with weights w. In other words the target we are using to train our model is

$$\text{target} = r_{k+1} + \gamma \max_{a' \in A} \widehat{Q}_w(s_{k+1}, a). \tag{47}$$

The sample average is computed from a minibatch of tuples, $(s_k, a_k, r_{k+1}, s_{k+1})$ drawn from the replay buffer.

We will now illustrate how one might implement a DQN agent in Python and Keras. Note that this is not the most efficient implementation of the DQN agent. Some of the deep learning frameworks like TensorFlow provide DQN agent modules (i.e. `tf_agent`) that hide most of the functionality of the agent from the user. Our purpose is to show how one might define an agent class object and then demonstrate its use in solving the FrozenLake environment provided by `Gym`.

The first thing we do in our notebook is initialize the FrozenLake environment and import the Python modules used in the notebook.

```
import gym
import random
import numpy as np
import matplotlib.pyplot as plt

from keras.optimizers import Adam
from keras.layers import Dense
from keras.models import Sequential
from collections import deque     #queue object for replay buffer

env = gym.make("FrozenLake-v1", is_slippery=False, render_mode='ansi')
train_episodes = 1000
test_episodes  = 100
```

```
max_steps       = 300
state_size        = env.observation_space.n
action_size      = env.action_space.n
batch_size       = 32

env.reset()
env.render()
```

The main thing we need to do is create an `Agent` class object. This object will encapsulate the DQN agent to be trained. Aside from the basic parameters, the *replay buffer* will be a private object (`self.memory`) maintained by the agent object.

```
class Agent:
    def __init__(self, state_size, action_size):
        self.memory = deque(maxlen=2500)
        self.learning_rate = 0.001
        self.epsilon = 1
        self.max_eps = 1
        self.min_eps = 0.01
        self.eps_decay = 0.001/3
        self.gamma = 0.9
        self.state_size = state_size
        self.action_size = action_size
        self.epsilon_lst=[]
        self.model = self.buildmodel()
```

The agent object has a number of methods to support its interaction with the `Gym` environment. The main actions regarding this interaction are defined below. The `buildmodel` methods builds the DQN neural network. In this case our model is a sequential network with two hidden layers. The hidden layers each have 32 nodes and the number of nodes in the output layer equals the number of actions. As discussed above, this model is compiled using the MSE loss function and in this case we will use an Adam optimizer. The main method used to interact with the environment are `action` which implements an ϵ-greedy policy.

```
    def buildmodel(self):
        model = Sequential()
        model.add(Dense(32, input_dim=self.state_size, activation="relu"))
```

```
        model.add(Dense(32, activation = "relu")
        model.add(Dense(self.action_size, activation="linear"))
        model.compile(loss = "mse", optimizer = Adam(lr=self.learning_rate))
        return model

    def action(self, state):
        if np.random.rand() > self.epsilon:
            return np.random.randint(0,4)
        return np.argmax(self.model.predict(state, verbose=0))

    def pred(self, state):
        return np.argmax(self.model.predict(state, verbose=0))
```

One of the main jobs of the DQN object is to encapsulate the replay buffer and provide tools managing the buffer. The replay buffer is instantiated as a private `deque` object called `self.memory`. The main methods working with the replay buffer are

- `add_memory` pops the current tuple $(s_k, a_k, r_{k+1}, s_{k+1})$ onto the buffer.

```
def add_memory(self, new_state, reward, done, state, action):
    self.memory.append((new_state, reward, done, state, action))
```

- `replay` is a method that pulls a mini-batch from the replay buffer and then loops through each tuple in the minibatch to compute the estimated target using equation (47) and then use that to update the DQN neural network's weights. In other words, we perform a single step of backpropagation for every tuple in the minibatch.

```
def replay(self, batch_size):
    minibatch = random.sample(self.memory, batch_size)
    for new_state, reward, done, state, action, in minibatch:
        target = reward
        if not done:
            target = reward +
            self.gamma*np.amax(self.model.predict(new_state,verbose=0))
        target_f = self.model.predict(state,verbose=0)
        target_f[0][action] = target
        self.model.fit(state, target_f, epochs=1, verbose=0)

    if self.epsilon > self.min_eps:
```

```
            self.epsilon = (self.max_eps - self.min_eps)*
                np.exp(-self.eps_decay*episode)+ self.min_eps

        self.epsilon_lst.append(self.epsilon)
```

The remaining methods for `Agent` object (`load` and `save`) are used to save or reload the trained model's weights from a file.

We can now build the DQN agent and begin training it. The main training loop is shown in the following script. The script executes a number of episodes. Each episode executes for a fixed number of steps, or until the agent reaches a terminal state. The agent uses the ϵ-greedy policy based on the current estimated Q^* to generate a tuple and that tuple is popped into the replay buffer. Once the episode is done, we then `replay` the buffer. Recall that *replaying* the buffer really corresponds to using the samples in a minibatch to update the neural network model weights.

```
reward_lst=[]
for episode in range(train_episodes):
    state= env.reset()[0]
    state_arr=np.zeros(state_size)
    state_arr[state] = 1
    state= np.reshape(state_arr, [1, state_size])
    reward = 0
    done = False
    for t in range(max_steps):
        # env.render()
        action = agent.action(state)
        new_state, reward, done, truncate, info = env.step(action)
        new_state_arr = np.zeros(state_size)
        new_state_arr[new_state] = 1
        new_state = np.reshape(new_state_arr, [1, state_size])
        agent.add_memory(new_state, reward, done, state, action)
        state= new_state

        if done:
            print(f'Episode: {episode:4}/{train_episodes} and step: {t:4}.
                      Eps: {float(agent.epsilon):.2}, reward {reward}')
            break

    reward_lst.append(reward)
```

```
    if len(agent.memory)> batch_size:
        agent.replay(batch_size)


print(' Train mean % score= ', round(100*np.mean(reward_lst),1))
```

We ran this training loop for 1000 episodes. The output from the script shows the reward obtained from that episode. It is important to note that the output shows more positive rewards (1.0) as we train longer, thereby indicating that we are learning how to find the FrozenLake's desired terminal state.

Once trained, we can then use the trained DQN agent to independently test how well our learning policy reached the goal.

```
# test
test_wins=[]
for episode in range(test_episodes):
    state = env.reset()[0]
    state_arr=np.zeros(state_size)
    state_arr[state] = 1
    state= np.reshape(state_arr, [1, state_size])
    done = False
    reward=0
    state_lst = []
    state_lst.append(state)
    print('******* EPISODE ',episode, ' *******')

    for step in range(max_steps):
        action = agent.pred(state)
        new_state, reward, done, truncate, info = env.step(action)
        new_state_arr = np.zeros(state_size)
        new_state_arr[new_state] = 1
        new_state = np.reshape(new_state_arr, [1, state_size])
        state = new_state
        state_lst.append(state)
        if done:
            print(reward)
            # env.render()
            break
```

```
    test_wins.append(reward)
env.close()


print(' Test mean % score= ', int(100*np.mean(test_wins)))
```

In this case we tested the model for 100 episodes and found that the learned policy was successful 100% of the time. This is to be expected here because we used the "deterministic" FrozenLake environment. It is also worth looking at the training score. This is shown below in Fig. 8. The training score is the "reward" received at the end of each episode. What this shows is that the likelihood of receiving a reward of 1.0 (i.e. getting to the desired destination) becomes more likelihood the longer we train. This graph therefore shows that our training procedure is working well.

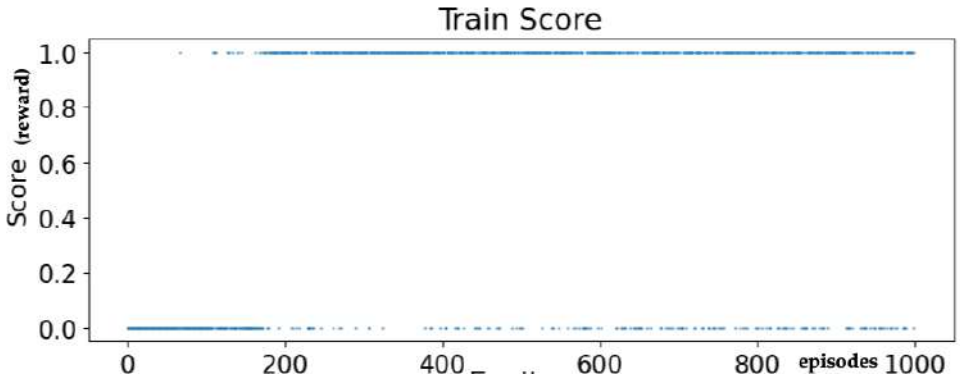


FIGURE 8. Success rate for DQN agent on FrozenLake (non-slippery)

5. Policy Gradient Methods - REINFORCE and Actor-Critic

All of the preceding RL algorithms (SARSA, Q-learning, DQN) were based on first learning the value function and then determining the optimal policy. These algorithms are therefore referred to as *value gradient* methods. Another approach to RL learns a model for the policy directly. In this case the policy is written as $\pi(a \mid s, \theta)$ where θ is a set of parameters that we need

to learn. Reinforcement learning algorithms that learn a model for the policy are called *policy gradient* methods. In this case we define a performance measure $J(\theta)$ for the policy model and then use gradient ascent to find those parameters θ that maximize that performance measure

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla_\theta J(\theta_t)}$$

where $\widehat{\nabla_\theta J(\theta_t)}$ is a stochastic estimate whose expectation approximates the gradient of $J(\theta)$. Policy gradient methods are methods that learn a model for the action policy, $\pi(a|s,\theta)$. Some of these methods methods also train a model for the value function as well. These algorithms are called *actor-critic* methods. All of these algorithms are referred to as policy gradient methods, whether or not they also learn an approximate value function.

One advantage that policy-gradient methods have over value-gradient methods is that the approximate policy automatically "explores" the state space since the policy $\pi(a|s,\theta)$ is stochastic. This means we don't have to use ϵ-greedy strategies whose randomized ϵ move may be really very poor. Another important advantage of policy gradient methods is that the policy function $\pi(a|s,\theta)$ may be much simpler than the value function model and so will be easier to learn. In addition to this, it is somewhat easier to inject prior information about the process into a control policy than a learned value function.

The policy function $\pi(a \mid s,\theta)$ can be parameterized in any way we wish, we simply need to make sure it is differentiable with respect to its parameters. If the action and state spaces are discrete and not too large, then one can form a set of parameterized numerical preferences, $h(s,a,\theta) \in \mathbb{R}$ for each state-action pair. The actions with the highest preference in each state are given the highest probabilities of being selected using, for example, a soft max distribution function

$$\pi(a \mid s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}.$$

These preferences, $h(s, a, \theta)$, can be parameterized in many ways. They many be computed by a deep neural network, or they could simply be linear functions,

$$h(s, a, \theta) = \theta^T \mathbf{x}(s, a)$$

of a predefined set of feature vectors $\mathbf{x}(s, a)$. The choice of these preference functions therefore represents a way to inject prior information into the training process that can help reduce the complexity of the learning problem.

Let us consider an episodic version of a policy gradient method where the policy is updated after an episode has been completed. We will define the performance function as the value function from the initial state s_0.

$$J(\theta) = V^{\pi_\theta}(s_0)$$

where V^{π_θ} is the true value function for policy π_θ. In the following derivation we assume no discounting (i.e. $\gamma = 1$) because it makes the derivation much more complicated without providing any greater insight into what is being done. The basic result we will establish is the **policy gradient equation**

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a Q^\pi(s, a) \nabla_\theta \pi(a|\, s\theta) \tag{48}$$

where $\mu(s)$ is the fraction of time spent in state s over the given episode and $Q^\pi(s, a)$ is the state-action value function.

To establish this result it will first be useful to obtain an explicit expression for $\mu(s)$. In particular, let $h(s)$ denote the probability that an episode begins in state s and let $\eta(s)$ denote the number of time steps, on average, the system is in state s in a single episode. We say time is spent in state s if the episode starts in s or if transitions are made into s from a preceding state in which time has been spent. This means, therefore, that $\eta(s)$ satisfies

$$\eta(s) = h(s) + \sum_{\overline{s}} \eta(\overline{s}) \sum_a \pi(a \,|\, \overline{s}) p(s \,|\, \overline{s}, a)$$

for all $s \in S$. This system of equations can be solved for the expected number of visits, $\eta(s)$ and the fraction of time spent in each state can then be written as

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in S. \tag{49}$$

We will need to use this equation in our derivation of the policy gradient equation (48).

To simplify notation, our following derivation drops the explicit dependence of π on θ. We first note that the gradient of the value function can be written in terms of the state action value function.

$$\begin{aligned}
\nabla V^{\pi}(s) &= \nabla \left[\sum_a \pi(a \,|\, s) Q^{\pi}(s,a) \right], \quad \text{for all } s \in S \\
&= \sum_a \left[\nabla \pi(a \,|\, s) Q^{\pi}(s,a) + \pi(a \,|\, s) \nabla Q^{\pi}(s,a) \right] \\
&= \sum_a \left[\nabla \pi(a|s) Q^{\pi}(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s', r \,|\, s, a)(r + V^{\pi}(s')) \right] \\
&= \sum_a \left[\nabla \pi(a|s) Q^{\pi}(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla V^{\pi}(s') \right].
\end{aligned}$$

We take this last equation and unroll it by expanding out $\nabla V^{\pi}(s')$ to get

$$\begin{aligned}
\nabla V^{\pi}(s) &= \sum_a \left[\nabla \pi(a|s) Q^{\pi}(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \right. \\
&\qquad \left. \sum_{a'} \left[\nabla \pi(a', s') Q^{\pi}(s', a') + \pi(a'|s') \sum_{s''} p(s''|s', a') \nabla V^{\pi}(s'') \right] \right]
\end{aligned}$$

and if we continue unrolling we finally get

$$\nabla V^{\pi}(s) = \sum_{x \in S} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) Q^{\pi}(x,a)$$

where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. We then obtain

$$\begin{aligned}\nabla J(\theta) &= \nabla V^{\pi}(s_0) \\ &= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a).\end{aligned}$$

Since the average time spent in state s is $\eta(s) = \sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi)$ we can express the gradient as

$$\begin{aligned}\nabla J(\theta) &= \sum_s \eta(s) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a) \\ &= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) Q^{\pi}(s,a).\end{aligned}$$

If we then use equation (49) for $\mu(s)$ (the fraction of time spent in s) we obtain

$$\begin{aligned}\nabla J(\theta) &= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a) \\ &\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) Q^{\pi}(s,a)\end{aligned}$$

thereby giving us the policy gradient equation (48). We will use this equation in developing our first policy-gradient algorithm known as REINFORCE [**Wil92**].

5.1. REINFORCE: Monte Carlo Policy Gradient: Recall from equation (48) that

$$\begin{aligned}
\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a Q^\pi(s,a) \nabla \pi(a|s,\theta) \\
&= \mathbb{E}_\pi \left[\sum_a Q^\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\theta) Q^\pi(S_t,a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(S_t,A_t) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right]
\end{aligned}$$

where G_t is the total return for the episode. Using this last equation for $\nabla J(\theta)$ we obtain the following gradient ascent update for the model's parameter

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\
&= \boxed{\theta_t + \alpha G_t \ln \pi(A_t|S_t,\theta_t)}.
\end{aligned}$$

The last equation comes about using the fact that $\nabla \ln x = \frac{\nabla x}{x}$.

```
import gym
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import Sequential
from tensorflow.keras.models import load_model
from tensorflow.keras import optimizers, losses

  ## Config ##
  ENV="CartPole-v1"
  RANDOM_SEED=12345
  N_EPISODES=500

  # random seed (reproduciblity)
  np.random.seed(RANDOM_SEED)
  tf.random.set_seed(RANDOM_SEED)

  # set the env
  env=gym.make(ENV,RANDOM_SEED) # env to import
  #env.seed(RANDOM_SEED)
  env.reset()[0] # reset to env
```

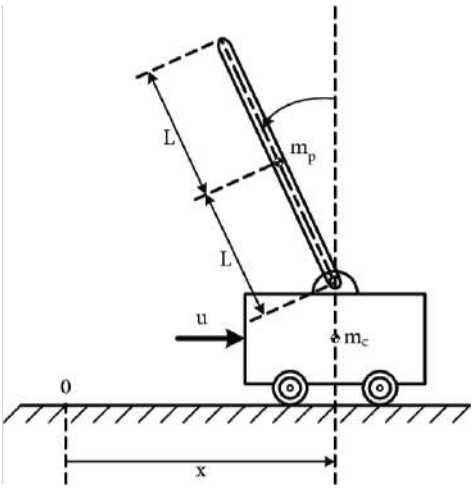


FIGURE 9. Cart Pole System

We are now going to demonstrate an implementation of the REINFORCE algorithm on a Cart Pole system [**BSA83**]. The code in Fig. 9 is used to set up the cart pole system in the Gym environment. This system is a benchmark problem in RL learning and in traditional feedback control. Fig. 9 shows a sideview of the system. The physical system consists of a 4 wheeled cart whose wheels are actuated by a DC servomotor. The "pendulum" is attached to the cart in a manner that allows it to only move in a single plane. It is assume there is an optical encoder on the pendulum that measures the angle θ of the pendulum with respect to the vertical. We also have an optical encoder on one of the cart's wheels that measures the angle of the motor's shaft. By knowing the radius of the wheels it then becomes possible to map the motor's shaft angle onto the cart's horizontal position, x. The objective of the problem is to determine the applied force, F, that the motor must deliver on the cart so the pendulum angle, θ, remains within a desired neighborhood of the vertical position and such that the cart's position, x, remains within a desired box. There are therefore 4 states, θ, $\dot{\theta}$, x, and $\dot{x}$. These dynamic variables satisfy the following second order differential equations

$$\begin{aligned}
\ddot{\theta} &= \frac{(M+m)g\sin\theta - \cos\theta\left[F + m\ell\dot{\theta}^2\sin\theta\right]}{(M+m)} \\
\ddot{x} &= \frac{\left(F + m\ell\left[\dot{\theta}^2\sin\theta - \ddot{\theta}\cos\theta\right]\right)}{(M+m)}
\end{aligned}$$

where M is the cart's mass, m is the mass of the pendulum, g is gravitational accerlation, F is the force applied to to the cart, and ℓ is the length of the pendulum.

The objective of the RL agent is to control the cartpole system so the angular position of the pendulum remains with $\pm 12^\circ$ from vertical and the linear position of the cart is within ± 2.4 meters of its starting position. So the state space is $S \subset \mathbb{R}^4$ and the states s are real-valued vectors of dimension 4. The force F will be quantized into either move left or move

right, with the same constant force. So the action space, A, of this problem consists of two values (R or L). In particular we will one-hot encode these actions as $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The reward function will be 1 every time instant the state stays within the desired bounding box $\theta \in [-12^\circ, 12^\circ]$ and $x \in [-2.4, 2.4]$. If the states are outside of this box, then the episode is over. We will also terminate training after 500 time steps, so that the maximum aggregate reward (return) that can be received is 500. We will implement the REINFORCE algorithm using Keras/Tensorflow on the Cart Pole environment in OpenAI Gym. We will run the REINFORCE algorithm until one of two termination conditions is satisfied. The first termination condition is a maximum limit of 1000 on the number of training episodes. The second termination condition is defined with regard to the kth episode's running reward, R_k, defined by the recursion

$$R_k = 0.05 * G_k + (1 - 0.05)R_{k-1}$$

for $k = 1, \ldots, 1000$ with $R_0 = 0$ and G_k being the aggregate reward (return) in the kth episode. The second termination condition stops training when R_k is greater than 99% of the maximum number of steps per episode.

Our notebook for the REINFORCE algorithm starts by creating the cart-pole environment using the script in Fig. 9. We then define the parameters for our problem.

```
state_shape = env.observation_space.shape
action_shape = env.action_space.n
gamma = 0.98            #discounting rate
alpha    = 1.e-4           #learning rate of policy gradient update
learning_rate = 0.005 # learning rate for optimizer
max_steps_per_episode = 500
N_EPISODES = 1000
termination_condition = 0.99
```

We will use a simple neural network to model the action policy. In particular, we use a sequential model with two hidden layers of 24 and 6 nodes. These hidden dense layers use a ReLu activation function. The output dense

layer has two nodes (one for the probability of each action) with a softmax activation function.

```
model = Sequential()
model.add(Dense(24, input_shape = state_shape, activation="relu"))
model.add(Dense(6, activation="relu"))
model.add(Dense(action_shape, activation = "softmax"))
```

Rather than using the `compile` method, we will simply declare an optimizer and then build our own model training routine using the `GradientTape` class. We'll declare several lists for logging the training process. We will also use a modified MSE loss function called the Huber loss. This loss function is slightly less sensitive to outliers.

```
optimizer = keras.optimizers.legacy.Adam(learning_rate=learning_rate)
huber_loss = keras.losses.Huber()
action_probs_history = []
rewards_history = []
episode_reward_history = []
running_reward_history = []
```

We are now ready to define our main training loop which is a while loop that we break out of when one of the two termination conditions described above is satisfied.

```
running_reward = 0
episode_count  = 0
while True:
     state = env.reset()[0]           #reset the environment
     episode_reward = 0
     with tf.GradientTape() as tape:     #record ops for backprop
          for timestep in range(1, max_steps_per_episode+1):
               state = state.reshape((1,4))
               #sample action from policy
               action_probs = model(state)      #get action probabilities
               action = np.ranodm.choice(action_shape, p=np.squeeze(action_probs))
               action_probs_histor.append(tf.math.log(action_probls[0, action]))

               #apply sampled action to environment
               state, reward, done, trunc, info = env.step(action)
               rewards_history.append(reward)
```

```
            episode_reward += reward

            if done:
                break

        #update running reward used for termination
        running_reward = 0.05 *episode_reward + (1-0.05)*running_reward

        #compute expected value from rewards
        returns = []
        discounted_sum = 0
        for r in rewards_history[::-1]:
            discounted_sum = r + gamma * discounted_sum
            returns.insert(0,discounted_sum)

        #calculate loss values
        history = zip(action_probs_history, returns)
        actor_losses = []
        for log_prob, ret in history:
            actor_losses.append(-log_prob * ret)
        loss_value = sum(actor_losses)

        #compute gradient and update weights
        grads = tape.gradient(loss_value, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        action_probs_history.clear()
        rewards_history.clear()
    #logging
    episode_reward_history.append(episode_reward)
    running_reward_history.append(running_reward)

    episode_count += 1
    #termination conditions
    if episode_count % 10 == 0:
          template = "running/episode reward: {:.2f}/{:.2f} at episode {}"
          print(template.format(running_reward, episode_reward, episode_count))
if (episode_count > N_EPISODES):
    print("FINISHED! - maxiter")
    break
if (running_reward>max_steps_per_episode*termination_condition):
    print(f"FINISHED! - termination condition - {episode_count}:{running_reward}")
    break
```

The result from running this script is shown in Fig. 10. This figure plots the running reward (dashed red line) and the episode rewards (blue dots) for all episodes. As we can see the RL algorithm has a great deal of variance in its average reward, but eventually the running reward reached the 99% level of 495, needed to terminate the session. So this policy gradient algorithm indeed appears to be learning a control policy that can keep the cartpole state within the box for nearly 500 time steps.

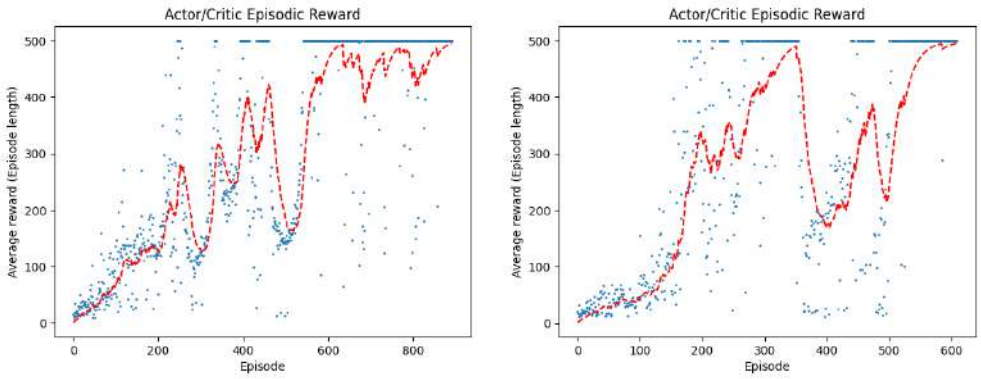


FIGURE 10. (left) results for REINFORCE (right) results for Actor-Critic

5.2. Actor-Critic Reinforcement Learning: The policy gradient equation (48) can be generalized to compare the state-action value function, $q_\pi(s,a)$ against a *baseline* function of state, $b(s)$.

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a \left(Q^\pi(s,a) - b(s)\right) \nabla_\theta \pi(a|s,\theta).$$

This baseline can be anything, even a random variable, as long as it is independent of the action a. Using arguments similar to derive the REINFORCE algorithm, we can develop the following policy update

$$\theta_{t+1} = \theta_t + \alpha \left(G_t - b(S_t)\right) \ln \pi(A_t|S_t,\theta_t).$$

By using a good "baseline", one can improve the learning speed. Obviously the natural choice for the baseline is an estimate of the state value function

$\widehat{V}(S_t, \mathbf{w})$ where $\mathbf{w}$ is the weight vector parameterizing that value function estimate.

Actor-Critic methods use a deep neural network to compute the value function estimate $\widehat{V}(S_t, \mathbf{w})$. In particular, one-step actor-critic methods may be seen as using TD methods to form the estimate $\widehat{V}(S_t, \mathbf{w})$. In this case, then the update of the policy takes the form

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha \left(G_{t:t+1} - \widehat{V}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t \mid S_t, \theta_t)}{\pi(A_t \mid S_t, \theta_t)} \\
&= \theta_t + \alpha \left(R_{t+1} + \gamma \widehat{V}(S_{t+1}, \mathbf{w}) - \widehat{V}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t \mid S_t, \theta_t)}{\pi(A_t \mid S_t, \theta_t)} \\
&= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t \mid S_t, \theta_t)}{\pi(A_t \mid S_t, \theta_t)}
\end{aligned}$$

where

$$\delta_t = R_{t+1} + \gamma \widehat{V}(S_{t+1}, \mathbf{w}) - \widehat{V}(S_t, \mathbf{w}).$$

This equation shows the update performed by the "policy actor". The update for the *critic* estimating $\widehat{V}(S_t, \mathbf{w})$ is done using a standard TD(0) method.

We can implement an actor-critic algorithm on the cartpole system using much of the same script we used for the REINFORCE algorithm in the preceding subsection. The main difference will be the structure of a model whose base part is used by both the critic and actor. This model is shown in the following script. The state, s, input is connected through a dense layer of 64 nodes that is common to both the actor and critic. There is then one additional dense layer for each actor and critic. So this model has two different types of outputs; an output representing the action probabilities, $\pi(a|s)$ and an output that estimates the state value function $\widehat{V}(s)$.

```
num_inputs = 4
num_actions = 2
num_hidden = 64

inputs = layers.Input(shape(num_inputs,))
common = layers.Dense(num_hidden, activation="relu")(inputs)
action = layers.Dense(num_actions, activation="softmax")(common)
```

```
critic = layers.Dense(1)(common)

model = keras.Model(inputs= inputs, outputs = [action, critic])
```

The structure of the training loop is very similar to the code we showed earlier for the REINFORCE algorithm. There is a main `while True:` loop that starts by fetching the initial environment state and then uses a `GradientTape` object to record the computations done during the run for at most 500 time steps. We then call the model to generate the actor's action probabilities (`actor_probs`) and value function estimate (`critic_value`). The action probabilities are used to randomly select an action which is then fed to the environment. The environment then returns the next state (s'), the reward, and a Boolean flag (`done`) indicating if the system state fell outside of the admissible box. This example uses a modified MSE loss function for the critic called a Huber loss function. The actor's loss function is the same as that used for the REINFORCE algorithm. These loss functions are less sensitive to outliers. We then compute the gradient and apply it to the actor/critic model. The full training loop is shown below.

```
while True:  # Run until solved
    state = env.reset()[0]
    episode_reward = 0
    with tf.GradientTape() as tape:
        for timestep in range(1, max_steps_per_episode+1):
            state = state.reshape((1,4))
            action_probs, critic_value = model(state)
            critic_value_history.append(critic_value[0, 0])
            action = np.random.choice(num_actions, p=np.squeeze(action_probs))
            action_probs_history.append(tf.math.log(action_probs[0, action]))

            # Apply the sampled action in our environment
            state, reward, done, trunc, info = env.step(action)
            rewards_history.append(reward)
            episode_reward += reward

            if done:
                break

        # Update running reward to check condition for solving
```

```
        running_reward = 0.05 * episode_reward + (1 - 0.05) * running_reward

        # Calculate expected value from rewards
        returns = []
        discounted_sum = 0
        for r in rewards_history[::-1]:
            discounted_sum = r + gamma * discounted_sum
            returns.insert(0, discounted_sum)

        # Normalize
        returns = np.array(returns)
        returns = (returns - np.mean(returns)) / (np.std(returns) + eps)
        returns = returns.tolist()

        # Calculating loss values to update our network
        history = zip(action_probs_history, critic_value_history, returns)
        actor_losses = []
        critic_losses = []
        for log_prob, value, ret in history:
            diff = ret - value           #this is where baseline = value
            actor_losses.append(-log_prob * diff)

            # critic uses a Huber loss function
            critic_losses.append(
                huber_loss(tf.expand_dims(value, 0), tf.expand_dims(ret, 0))
            )

        # Backpropagation
        loss_value = sum(actor_losses) + sum(critic_losses)
        grads = tape.gradient(loss_value, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        # Clear the loss and reward history
        action_probs_history.clear()
        critic_value_history.clear()
        rewards_history.clear()

    # Log details
    episode_reward_history.append(episode_reward)
    running_reward_history.append(running_reward)
    episode_count += 1
    if episode_count % 10 == 0:
        template = "running/episode reward: {:.2f}/{:.2f} at episode {}"
        print(template.format(running_reward, episode_reward, episode_count))
    if (episode_count > NUM_EPISODES):
```

```
        print("FINISHED! - maxiter")
        break
    if (running_reward>max_steps_per_episode*termination_condition):
        print(f"FINISHED! - termination condition - {episode_count}:{running_reward}")
        break
```

The result from this training run is shown on the right side of Fig. 10. The dashed red line is the running reward and the blue dots are the episode rewards. While this run eventually satisfies the 99% termination condition around the 600th episode, we see that it almost met that termination condition around episode 350, after which the running reward dropped down until it was able to recover. This type of variability is often seen in these RL training sessions and they highlight an important weakness of RL policies. The termination condition simply says that the policy was mostly successful on a consecutive set of random episodes, but we have no analytical guarantees that the policy will be effective on other samples. In particular, we have no guarantees that an RL policy will *stabilize* the dynamic environment in some well defined sense. This is one of the main open questions around RL methods, how does one go about stabilizing a policy?

APPENDIX A

Probability Review

Probability plays a major role in understanding how deep learning models can generalize beyond their training data. This appendix reviews those probability concepts needed to discuss statistical learning theory.

Let Ω be a set of *experimental outcomes*. Any subset of Ω is called an *event*. The set of all events is the power set 2^{Ω}. Two sets, $\omega_1, \omega_2 \in \Omega$ are *mutually disjoint* if $\omega_1 \cap \omega = \emptyset$. For any event $\omega \in 2^{\Omega}$, we define the *event probability* by the function $P : 2^{\Omega} \rightarrow [0, 1]$ such that

$$P(\omega) \geq 0 \text{ and } P(\Omega) = 1.$$

Moreover, we require that if ω_1 and ω_2 are any two mutually disjoint events then

$$P(\omega_1 \cup \omega_2) = P(\omega_1) + P(\omega_2).$$

We further restrict our attention to any subset (possibly infinite), $\mathcal{F}$, of 2^{Ω} that is closed with respect to the binary operations of set union and intersection. We refer to $\mathcal{F}$ as σ-algebra if for any infinite sequence, $\{\omega_k\}_{k=1}^{\infty}$ of pairwise disjoint events in $\mathcal{F}$ that

$$P\left(\bigcup_{k=1}^{\infty} \omega_k\right) = \sum_{k=1}^{\infty} P(\omega_k).$$

With these notational conventions we refer to the triple $(\Omega, \mathcal{F}, P)$ as a *probability space*.

Given the probability space, $(\Omega, \mathcal{F}, P)$, a *random variable* (RV) is a function $\mathbf{x} : \Omega \rightarrow \mathbb{R}$ that assigns a real number to every outcome in Ω

such that the set $\{\omega \in \Omega \,:\, \mathbf{x}(\omega) \leq x\}$ has a well defined probability measure. This means that this set is a measurable event in $\mathcal{F}$. The value of the random variable for outcome, $\omega \in \Omega$ is denoted as $\mathbf{x}(\omega)$ where we will routinely denote random variables as boldfaced letters. The probability of the event $\{\omega \in \Omega \,|\mathbf{x}(\omega \leq x\}$ is denoted as $F_{\mathbf{x}}(x) = P(x)$. We can further show that $P(\mathbf{x}(\omega) = \infty) = 0 = P(\mathbf{x}(\omega) = -\infty)$.

The *distribution* function of RV $\mathbf{x}$ is the function $F_{\mathbf{x}} : \mathbb{R} \to [0, 1]$ such that

$$F_{\mathbf{x}}(x) = P(\{\omega \in \Omega \,|\, \mathbf{x}(\omega) \leq x\}) = P(x) = \Pr\{\mathbf{x} \leq x\}.$$

The derivative of the distribution function

$$f_{\mathbf{x}}(x) = \frac{dF_{\mathbf{x}}(x)}{dx}$$

is called the *probability density function* of RV $\mathbf{x}$.

Let $\mathbf{x}$ be a random variable defined with respect to probability space $(\Omega, \mathcal{F}, P)$ with probability distribution $F_{\mathbf{x}}$. Let $g : \mathbb{R} \to \mathbb{R}$ be a function and define the new random variable, $\mathbf{y} = g(\mathbf{x})$. The probability distribution of RV $\mathbf{y}$ is

$$F_{\mathbf{y}}(y) = P\left(\{\omega \in \Omega \,:\, g(\mathbf{x}(\omega)) \leq y\}\right)$$

if g is invertible then we can readily see that

$$F_{\mathbf{y}}(y) = P\left(\left\{\omega \in \Omega \,:\, \mathbf{x}(\omega) \leq g^{-1}(y)\right\}\right) = F_{\mathbf{x}}(g^{-1}(y))$$

which expresses the distribution for $\mathbf{y}$ in terms of the distribution for $\mathbf{x}$.

The following example shows that some care must be taken in expressing $F_{\mathbf{y}}$ in terms of $F_{\mathbf{x}}$. Let $\mathbf{y} = a\mathbf{x} + b$ where a and b are real constants. The analytical form of $\mathbf{y}$'s distribution depends upon whether a is positive or negative. In particular, one can show that

$$F_{\mathbf{y}}(y) = \begin{cases} P\left(\left\{\omega \,:\, \mathbf{x}(\omega) \leq \frac{y-b}{a}\right\}\right) = F_{\mathbf{x}}\left(\frac{y-b}{a}\right) & \text{if } a > 0 \\ P\left(\left\{\omega \,:\, \mathbf{x}(\omega) \geq \frac{y-b}{a}\right\}\right) = 1 - F_{\mathbf{x}}\left(\frac{y-b}{a}\right) & \text{if } a < 0 \end{cases}.$$

The *expected value* or mean of RV $\mathbf{x}$ is

$$\mathbb{E}(\mathbf{x}) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} x f_{\mathbf{x}}(x) dx.$$

Given RV's $\mathbf{x}$ and $\mathbf{y}$ defined over the same probability space, then the mean of $\mathbf{y} = g(\mathbf{x})$ is

$$\mathbb{E}(\mathbf{y}) = \int_{-\infty}^{\infty} y f_{\mathbf{y}}(y) dy = \int_{-\infty}^{\infty} g(x) f_{\mathbf{x}}(x) dx = \mathbb{E}(g(\mathbf{x})).$$

The *variance* of RV $\mathbf{x}$ is

$$\sigma^2 = \text{var}(\mathbf{x}) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} (x - \mathbb{E}(\mathbf{x}))^2 f_{\mathbf{x}}(x) dx.$$

A random variable, $\mathbf{x}$, is said to be *normally distributed* if its density function is

$$N(\mathbf{x}\,;\, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where $\mathbb{E}(\mathbf{x}) = \mu$ and $var(\mathbf{x}) = \sigma^2$. A jointly Gaussian random variables $\mathbf{x} : \Omega \to \mathbb{R}^n$ has the density

$$N(x \,:\, \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}} \frac{1}{\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$

where $\mu = \mathbb{E}(\mathbf{x})$ is the mean vector and $\Sigma = \mathbb{E}((\mathbf{x}-\mu)(\mathbf{x}-\mu)^T)$ is the random variable's *covariance matrix*.

Let A and B be two events in probability space $(\Omega, \mathcal{F}, P)$. The *conditional probability* of event A assuming event B has occurred is

$$P(A \,|\, B) \stackrel{\text{def}}{=} \frac{P(A \cap B)}{P(B)}.$$

If $U = \{A_1, \ldots, A_n\}$ is a partition of Ω and B is any event in $\mathcal{F}$, then the total probability of event B is

$$P(B) = \sum_{k=1}^{n} P(B \,|\, A_k) P(A_k).$$

Two events A and B are said to be independent if $P(A \cap B) = P(A)P(B)$. The joint probability of two events A and B is denoted as

$$P(A, B) = P(A)P(B \mid A).$$

Bayes theorem states that

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

$P(A)$ is called the *a priori* probability and $P(A \mid B)$ is called the *a posteriori probability*. The union bound or Boole's inequality says that for any two events we have

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \leq P(A) + P(B).$$

Stochastic Convergence:. Consider a random vector $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \end{bmatrix}$. The *characteristic function* of this random vector is

$$M_{\mathbf{x}}(\lambda) = \mathbb{E}\left\{e^{j\lambda\mathbf{x}}\right\} = \mathbb{E}\left\{e^{j(\lambda_1 x_1 + \cdots + \lambda_n x_n)}\right\}$$

where $\lambda = \begin{bmatrix} \lambda_1 & \cdots & \lambda_n \end{bmatrix}$ is a real valued vector in $\mathbb{R}^n$. Note that $M_{\mathbf{x}}(\lambda)$ may be seen as the Fourier transform of the random variable's density function.

If $\mathbf{x}_k$ are independent with densities $f_k(x_k)$, then the density of the sum

$$\mathbf{z} = \mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_n$$

can be easily shown to be

$$f_{\mathbf{z}}(z) = f_{\mathbf{x}_1}(z) * f_{\mathbf{x}_2}(z) * \cdots * f_{\mathbf{x}_n}(z)$$

where $g * f$ denotes the convolution integral of the two densities g and f. This is easily proven from the fact that the Fourier transform of the density function is its characteristic function.

We are interested in sequences of random variables because we can view the data set as a sequence

$$\mathcal{D} = \{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{M}$$

that is used to select an estimate for the probability distributions in the system. In general, these estimates take the form of sums (sample averages) and so we are interested in how these series converge as their length gets large. The notion of convergence we use, however, is a probabilistic one. There are several notions of stochastic convergence

- **Convergence with probability 1 (almost everywhere):** There is a set of outcomes, ω, such that

$$\lim \overline{x}_k(\omega) = \mathbf{x}(\omega), \quad \text{as } k \to \infty$$

  exists and the probability of the limiting event is 1.
- **Convergence in MS sense:**

$$\mathbb{E}\left\{(\mathbf{x}_k - \mathbf{x})^2\right\} \to 0, \quad \text{as } k \to \infty.$$

- **Convergence in Probability:**

$$P\left\{|\mathbf{x}_k - \mathbf{x}| > \epsilon\right\} \to 0, \quad \text{as } k \to \infty.$$

- **Convergence in Distribution:**

$$F_{\mathbf{x}_k}(x) \to F_{\mathbf{x}}(x), \quad \text{as } k \to \infty.$$

The relationship between these notions of convergence are that

$$\text{MS or AE} \Rightarrow \text{Probability} \Rightarrow \text{Distribution}.$$

The notions of MS and AE convergence may be too strong to discuss generalization in machine learning problems. So we use the relaxed notion of convergence in probability when talking about our ability to generalize a given model. This probabilistic notion of learning is sometimes called *probably almost correct* or PAC learning [**Val84**].

APPENDIX B

Markov Chains

Markov chains a probabilistic models for trials of random experiments that allow us to consider when the future evolution of the process depends on current state, rather than the history leading up to that state. We can formalize this notion as follows. Let $\{\mathbf{x}_k \ : \ k \geq 0\}$ be a sequence of random variables taking values in a finite set X. Since this is a discrete finite set we will find it convenient to represent it a set of integer indices, i.e. $X = \{1, 2, \ldots, n\}$. Further assume that

$$P(\mathbf{x}_{k+1} = y \,|\, \mathbf{x}_0 = x_0, \mathbf{x}_1 = x_1, \cdots \mathbf{x}_k = x) = P(\mathbf{x}_{k+1} = y \,|\, \mathbf{x}_k = x) = p(x, y) \tag{50}$$

for all $k \geq 0$ and all states $x, y, x_{0,1}, \ldots, x_{k-1}$ in X. Then $\{\mathbf{x}_k \,|\, k \geq 0\}$ is called a *Markov chain* with state space X and an $n \times n$ *transition matrix* $\mathbf{P}$. Assuming that X can be represented (wlog) as a set of positive integers $\{1, 2, \ldots, n\}$, then for any $x, y \in X$ we have component in the x^{th} row and y^{th} column of $\mathbf{P}$ equal to the transition probability $p(x, y)$ from state x to state y. Markov chains, therefore, are stochastic processes whose future state is dependent only upon the present state.

We may think of the Markov chain $(X, \mathbf{P})$ as a dynamical system where the "state" at any time $k \geq 0$ is a probability distribution over X. The characterization of this system's future behavior starts from an *initial distribution*. In particular let $\pi_x(k)$ denote the probability of the MC state being in state $x \in X$ at time instant k. We let $\pi(k)$ denote the *probability row vector* (state) of the process at time k and assuming X is a finite set of positive integers $1, 2, \ldots, n$, then

$$\pi(k) = \left[\begin{array}{cccc} \pi_1(k) & \pi_2(k) & \cdots & \pi_n(k) \end{array} \right]$$

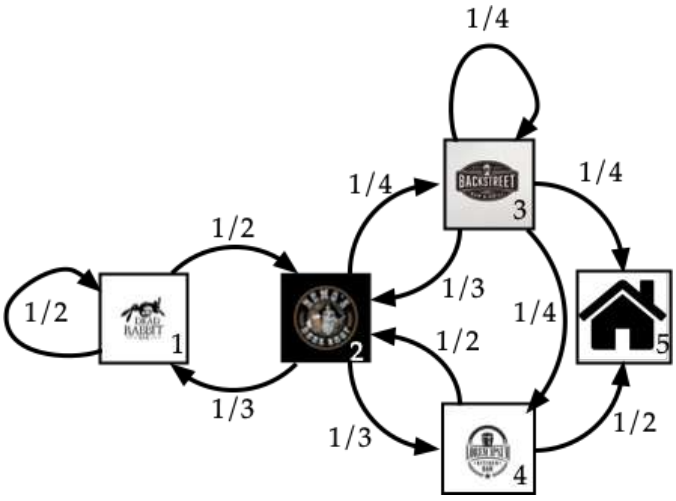


FIGURE 1. Pub Crawl

and the *initial distribution* for the MC is $\pi(0)$.

Note that equation (50) may be viewed as asserting that the conditional distribution of the random future state, $\mathbf{x}_{k+1}$ depends only on the present state $\mathbf{x}_k$ and is statistically independent of the past states $\mathbf{x}_0, \ldots, \mathbf{x}_{k-1}$.

One may think of a Markov chain as a model for *jumping* from state x to state y where each jump is governed by the jump probability $p(x, y)$. Note that $p(x, y)$ is a probability distribution with respect to y. In other words $p(x, y) \geq 0$ and for all $x, y \in X$ we have $\sum_{y \in X} p(x, y) = 1$.

Let us consider an example of a MC. We have a friend who travels between four pubs. We assume the streets connecting the pubs are as shown in Fig. 1. We can model our friend's night journey as a Markov chain with the state space $X = \{1, 2, 3, 4\}$. Assuming that after leaving a pub the friend travels to the next pub down the road, making a uniformly random decision which road to follow should there be more than one road, then we obtain

the state transition matrix

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1/3 & 0 & 1/3 & 1/3 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 1/2 & 1/2 & 0 \end{bmatrix}$$

Let us suppose that when his pub crawl starts, our friend's chooses one of the 4 pubs in a uniformly random manner so the initial distribution is

$$\pi(0) = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}$$

Let us determine the probability that our friend visits all of the pubs in order. We can readily see that

$$\begin{aligned} P(\mathbf{x}_0 = 1, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3, \mathbf{x}_3 = 4) &= \pi_1(0)p(1,2)p(2,3)p(3,4) \\ &= \frac{1}{4} \times 1 \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{24} \end{aligned}$$

Now let us consider the probability that our friend is at a specified pub $y \in 1, 2, 3, 4$ after he is done visiting the first one. This probability is

$$\begin{aligned} P(\mathbf{x}_1 = y) &= \sum_{x=1}^{4} p(x,y) = \sum_{x=1}^{4} \pi_x p(x,y) \\ &= \begin{bmatrix} \pi_1(0) & \pi_2(0) & \pi_3(0) & \pi_4(0) \end{bmatrix} \begin{bmatrix} p(1,y) \\ p(2,y) \\ p(3,y) \\ p(4,y) \end{bmatrix} \end{aligned}$$

this is the product of the probability row vector $\pi(0)$ with the y^{th} column of the state transition matrix. We can therefore see that the probability row vector at the first time instant 1 will be

$$\pi(1) = \pi(0)\mathbf{P}$$

These observations suggest the following theorem

Theorem 1: Let $\mathbf{P}$ denote the transition matrix of a Markov chain $\{\mathbf{x}_k : k \geq 0\}$ with initial distribution π. Then

$$P(\mathbf{x}_k = y) \quad = \quad y^{\text{th}} \text{ entry of } \pi\mathbf{P}^k$$

and

$$P(\mathbf{x}_k = y \,|\, \mathbf{x}_0 = x) = xy^{\text{th}} \text{ entry of } \mathbf{P}^k$$

Proof:. Assume that $X = \{1, 2, \ldots, n\}$. Let $\pi(k)$ be the probability distribution of states at time instant $k \geq 0$. Note that clearly

$$\begin{aligned} \pi(1) &= \pi\mathbf{P} \\ \pi(2) &= \pi(1)\mathbf{P} = \pi\mathbf{P}^2 \\ &\vdots \\ \pi(k) &= \pi(n-1)\mathbf{P} = \cdots = \pi\mathbf{P}^k \end{aligned}$$

So $P(\mathbf{x}_k = y)$ is the y^{th} entry of $\pi\mathbf{P}^k$.

Now let π be an elementary vector in the sense that for some $x \in X$ we have $\pi_x = 1$ and $\pi_y = 0$ for $y \neq x$. We can then see that

$$\pi\mathbf{P}^k = \left[p^k(x,1), \ldots, p^k(x,n)\right]$$

where $p^k(x,y) = P(\mathbf{x}_k = y \,|\, \mathbf{x}_0 = x)$. Note that beause of the special form of **pi**, we have $\pi\mathbf{P}^k$ being the x^{th} row of $\mathbf{P}^k$ and so $p^k(x,y)$ is simply the xy^{th} entry of $\mathbf{P}^k$. $\diamondsuit$

A Markov chain $(X, \mathbf{P})$ is said to be *ergodic* or *irreducible* if it is possible to eventually get from every state to every other state with a positive probability. Formally this means that for any pair of states $x, y \in X$ there exists a positive constant, N such that the xy^{th} entry of $\mathbf{P}^N$ is positive. A Markov chain is said to be *regular* if some power, N, of its transition matrix has only positive entries. The following theorem is stated without proof concerning the asymptotic behavior of regular transition matrices.

Theorem: Let $\mathbf{P}$ denote the transition matrix of a regular Markov chain with finite state space, then there exists a constant $n \times n$ matrix $\mathbf{W}$ such that

$$\lim_{k\to\infty} \mathbf{P}^k = \mathbf{W}$$

and all rows of $\mathbf{W}$ have the same strictly positive probability vector.

Since this theorem ensures all rows of $\mathbf{W}$ are identical, we can use this fact to compute $\mathbf{W}$ and provide an interpretation for that common row vector. This is address in the next theorem which is also presented without proof.

Theorem: Let $\mathbf{P}$ be a transition matrix for a regular finite Markov chain with state space $X = \{1, 2, \ldots, n\}$. Then the $n \times n$ system of linear equations given by $\mathbf{xP} = \mathbf{x}$ has a unique probability row vector solution, $\mathbf{w}$ and this solution is the common row in $\lim_{k\to\infty} \mathbf{P}^k$. Furthermore, if π is an arbitrary probability row vector, then

$$\lim_{k\to\infty} \pi\mathbf{P}^k = \mathbf{w}$$

Let $\mathbf{P}^k(x, y)$ denote the xy^{th} component of $\mathbf{P}^k$. This theorem therefore states that the long-run probability of being in state y at time k is

$$\sum_{x=1}^{n} \pi_x \mathbf{P}^k(x, y)$$

is approimately equal to the y^{th} entry of $\mathbf{w}$ for all y. In addition we see that if $\mathbf{w}$ is the common row vector of $\mathbf{W}$ then we also have

$$\mathbf{wP}^k = \mathbf{w}$$

for all $k \geq 0$. This means that if we start with the initial distribution $\mathbf{w}$, then this represent an *equilibrium* or *stationary* distribution for the Markov chain.

Note that since $\mathbf{w}$ satisfies $\mathbf{w} = \mathbf{wP}$, then we can see that $\mathbf{w}$ is a left eigenvector of the state transition matrix with associated eigenvalue 1. We can use this fact to solve for $\mathbf{w}$.

As an example, consider the Markov chain with transition matrix

$$\mathbf{P} = \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{bmatrix}$$

with state space $X = \{1, 2, 3\}$. If we look $\mathbf{P}^2$ we get

$$\mathbf{P}^2 = \begin{bmatrix} 7/16 & 3/16 & 3/8 \\ 3/8 & 1/4 & 3/8 \\ 3/8 & 3/16 & 7/16 \end{bmatrix}$$

since all components are positive we know $\mathbf{P}$ is regular and so we know the limiting matrix $\mathbf{W}$ exists whose rows all have the same value, $\mathbf{w}$. To see what this common row vector is we solve the linear algebraic equation $\mathbf{xP} = \mathbf{x}$ subject to $\mathbf{x}$ being a row probability vector. In other words, $x_1 + x_2 + x_3 = 1$ and $x_i \geq 0$. The system of linear algebraic equations is

$$\begin{aligned} x_1/2 + x_2/2 + x_3/4 &= x_1 \\ x_1/4 + x_3/4 &= x_2 \\ x_1/4 + x_2/2 + x_3/2 &= x_3 \end{aligned}$$

Solving this 3 by 3 system we get $\mathbf{w} = \mathbf{x} = [2/5, 1/5, 2/5]$

Note that for non-negative ergodic chains, this fundamental limit theorem may fail. We really do need the additional restriction of the chain being regular. However, for ergodic Markov chains with finite state space there is still a unique *stationary* probability vector $\mathbf{w}$ such that $\mathbf{wP} = w$. This means that if we used this $\mathbf{w}$ as the initial distribution of the chain, then the chain's states would have the same distribution for all future times. For a regular Markov chain the initial distribution $\mathbf{w}$ that satisfies $\mathbf{wP}^k = \mathbf{w}$ may be seen as a long run probability vector for being in various states. For

ergodic chains, the limits of these individual n-step probabilities may not exist.

Bibliography

[AB17] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

[ACG+16] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[AM12] Yaser S Abu-Mostafa. *Learning from data: a short course*. AMLBook, 2012.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[APK03] Baher Abdulhai, Rob Pringle, and Grigoris J Karakoulas. Reinforcement learning for true adaptive traffic signal control. *Journal of Transportation Engineering*, 129(3), 2003.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[BCM+13] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*, pages 387–402. Springer, 2013.

[BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[Bel54] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[BHLM19] Peter L Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *The Journal of Machine Learning Research*, 20(1):2285–2301, 2019.

[BHMM19] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[BHN19] Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and Machine Learning: Limitations and Opportunities*. fairmlbook.org, 2019. http://www.fairmlbook.org.

[BLM13] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration inequalities: A nonasymptotic theory of independence*. Oxford university press, 2013.

[BPRS18] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[Bre96] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[BSA83] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[Cho17] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[Cho21] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[Cov64] Thomas M Cover. Classification and generalization capabilities of linear threshold units. Technical report, STANFORD RESEARCH INST MENLO PARK CA MENLO PARK United States, 1964.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning

phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[CXP+21] Zhong Cao, Shaobing Xu, Huei Peng, Diange Yang, and Robert Zidek. Confidence-aware reinforcement learning for self-driving cars. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[DCT+19] Jacob Devlin, Ming-Wei Chang, Toutanova, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2. Minneapolis, Minnesota, 2019.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[Dwo08] Cynthia Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, pages 1–19. Springer, 2008.

[Fis36] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[FMI83] Kunihiko Fukushima, Sei Miyake, and Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (5):826–834, 1983.

[Fos22] David Foster. *Generative deep learning*. " O'Reilly Media, Inc.", 2022.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[GFS05] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. In *International conference on artificial neural networks*, pages 799–804. Springer, 2005.

[(Go] Tomas Mikolov (Google). Word2vec. https://code.google.com/archive/p/word2vec.

[Goo] Google. Google translate.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[Hoc98] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[Hol18] Matthijs Hollemans. One-stage object detection. blog post at https://machinethink.net/blog/object-detection/, June 9, 2018.

[Hop82] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HW62] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106, 1962.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KMA+21] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[LAE+16] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LC22] Eng Hock Lee and Vladimir Cherkassky. Vc theoretical explanation of double descent. *arXiv preprint arXiv:2205.15549*, 2022.

[LLG+19] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[LWTG19] Yuanlong Li, Yonggang Wen, Dacheng Tao, and Kyle Guan. Transforming cooling optimization for green data center via deep reinforcement learning. *IEEE transactions on cybernetics*, 50(5):2002–2013, 2019.

[MKS+13] V. Mnih, K. Kvaukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playting Atari with deep reinforcement learning. In *Workshop in Deep Learning*. Neural Information Processing Systems (NIPS), 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[MR89] James L McClelland and David E Rumelhart. *Explorations in parallel distributed processing: A handbook of models, programs, and exercises*. MIT press, 1989.

[ND21] Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In *International Conference on Machine Learning*, pages 8162–8171. PMLR, 2021.

[Nes83] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady an ussr*, volume 269, pages 543–547, 1983.

[NKFL18] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[Ope22] TB OpenAI. Chatgpt: Optimizing language models for dialogue, 2022.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[RB19] Inioluwa Deborah Raji and Joy Buolamwini. Actionable auditing: Investigating the impact of publicly naming biased performance results of commercial ai products. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 429–435, 2019.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[RHGS15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[RN94] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[RNS+18] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. Preprint, 2018.

[Roo22] Kevin Roose. An ai-generated picture won an art prize. artists aren't happy. *The New York Times*, 2:2022, 2022.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[S+99] Pierre Soille et al. *Morphological image analysis: principles and applications*. Springer, 1999.

[SB81] Richard S Sutton and Andrew G Barto. Toward a modern theory of adaptive networks: expectation and prediction. *Psychological review*, 88(2):135, 1981.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SCD+17] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[Ser82] J. Serra. *Image Analysis and Mathematical Morphol-ogy*. Academic Press, 1982.

[SGZ+16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29, 2016.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[SL87] T. Soderstrom and L. Ljung. *Theory and Practice of Recursive Identification*. MIT Press, 1987.

[SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[Sta] Stanford. Glove. https://nlp.stanford.edu/projects/glove/.

[Sut88] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[Sut95] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, 8, 1995.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[SWM17] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*, 2017.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[SZS+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[T+95] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[TH12] Tijmen Tieleman and G. Hinton. Lecture 6.4-rmsprop: divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 4, 2012.

[Tur09] A. M. Turing. Computing machinery and intelligence (1950). In *Parsing the Turing Test*, pages 23–65. Springer (Netherlands), 2009.

[Val84] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Vap98] Vladimir Vapnik. *Statistical learning theory*. Wiley, 1998.

[VDBZ+23] Eva AM Van Dis, Johan Bollen, Willem Zuidema, Robert van Rooij, and Claudi L Bockting. Chatgpt: five priorities for research. *Nature*, 614(7947):224–226, 2023.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[VT04] Harry L Van Trees. *Detection, estimation, and modulation theory, part I: detection, estimation, and linear modulation theory*. John Wiley & Sons, 2004.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[Wer90] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[WH60] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. Technical report, Stanford Univ Ca Stanford Electronics Labs, 1960.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[WWL13] Stefan Wager, Sida Wang, and Percy S Liang. Dropout training as adaptive regularization. *Advances in neural information processing systems*, 26, 2013.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[YLNY21] Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

[YRC07] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[YZS+22] Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Yingxia Shao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. Diffusion models: A comprehensive survey of methods and applications. *arXiv preprint arXiv:2209.00796*, 2022.

[ZBH+21] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.