CHAPTER 2

Learning Basics and Linear Models

Neural networks, the topic of this book, are a class of supervised machine learning algorithms.

This chapter provides a quick introduction to supervised machine learning terminology and practices, and introduces linear and log-linear models for binary and multi-class classification.

The chapter also sets the stage and notation for later chapters. Readers who are familiar with linear models can skip ahead to the next chapters, but may also benefit from reading Sections 2.4 and 2.5.

Supervised machine learning theory and linear models are very large topics, and this chapter is far from being comprehensive. For a more complete treatment the reader is referred to texts such as Daumé III [2015], Shalev-Shwartz and Ben-David [2014], and Mohri et al. [2012].

2.1 SUPERVISED LEARNING AND PARAMETERIZED FUNCTIONS

The essence of supervised machine learning is the creation of mechanisms that can look at examples and produce generalizations. More concretely, rather than designing an algorithm to perform a task ("distinguish spam from non-spam email"), we design an algorithm whose input is a set of labeled examples ("This pile of emails are spam. This other pile of emails are not spam."), and its output is a function (or a program) that receives an instance (an email) and produces the desired label (spam or not-spam). It is expected that the resulting function will produce correct label predictions also for instances it has not seen during training.

As searching over the set of all possible programs (or all possible functions) is a very hard (and rather ill-defined) problem, we often restrict ourselves to search over specific families of functions, e.g., the space of all linear functions with d_{in} inputs and d_{out} outputs, or the space of all decision trees over d_{in} variables. Such families of functions are called *hypothesis classes*. By restricting ourselves to a specific hypothesis class, we are injecting the learner with *inductive bias*—a set of assumptions about the form of the desired solution, as well as facilitating efficient procedures for searching for the solution. For a broad and readable overview of the main families of learning algorithms and the assumptions behind them, see the book by Domingos [2015].

The hypothesis class also determines what can and cannot be represented by the learner. One common hypothesis class is that of high-dimensional linear function, i.e., functions of the

form:[1]

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b} \tag{2.1}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}} \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}}.$$

Here, the vector $\boldsymbol{x}$ is the *input* to the function, while the matrix $\boldsymbol{W}$ and the vector $\boldsymbol{b}$ are the *parameters*. The goal of the learner is to set the values of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ such that the function behaves as intended on a collection of input values $\boldsymbol{x}_{1:k} = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ and the corresponding desired outputs $\boldsymbol{y}_{1:k} = \boldsymbol{y}_i, \ldots, \boldsymbol{y}_k$. The task of searching over the space of functions is thus reduced to one of searching over the space of parameters. It is common to refer to parameters of the function as Θ. For the linear model case, $\Theta = \boldsymbol{W}, \boldsymbol{b}$. In some cases we want the notation to make the parameterization explicit, in which case we include the parameters in the function's definition: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$.

As we will see in the coming chapters, the hypothesis class of linear functions is rather restricted, and there are many functions that it cannot represent (indeed, it is limited to *linear* relations). In contrast, *feed-forward neural networks with hidden layers*, to be discussed in Chapter 4, are also parameterized functions, but constitute a very strong hypothesis class—they are *universal approximators*, capable of representing any Borel-measurable function.[2] However, while restricted, linear models have several desired properties: they are easy and efficient to train, they often result in convex optimization objectives, the trained models are somewhat interpretable, and they are often very effective in practice. Linear and log-linear models were the dominant approaches in statistical NLP for over a decade. Moreover, they serve as the basic building blocks for the more powerful nonlinear feed-forward networks which will be discussed in later chapters.

2.2 TRAIN, TEST, AND VALIDATION SETS

Before delving into the details of linear models, let's reconsider the general setup of the machine learning problem. We are faced with a dataset of k input examples $\boldsymbol{x}_{1:k}$ and their corresponding gold labels $\boldsymbol{y}_{1:k}$, and our goal is to produce a function $f(\boldsymbol{x})$ that correctly maps inputs $\boldsymbol{x}$ to outputs $\hat{\boldsymbol{y}}$, as evidenced by the training set. How do we know that the produced function $f()$ is indeed a good one? One could run the training examples $\boldsymbol{x}_{1:k}$ through $f()$, record the answers $\hat{\boldsymbol{y}}_{1:k}$, compare them to the expected labels $\boldsymbol{y}_{1:k}$, and measure the accuracy. However, this process will not be very informative—our main concern is the ability of $f()$ to generalize well to unseen examples. A function $f()$ that is implemented as a lookup table, that is, looking for the input $\boldsymbol{x}$ in its memory and returning the corresponding value $\boldsymbol{y}$ for instances is has seen and a random value otherwise, will get a perfect score on this test, yet is clearly not a good classification function as it has zero generalization ability. We rather have a function $f()$ that gets some of the training examples wrong, providing that it will get unseen examples correctly.

[1] As discussed in Section 1.7. This book takes a somewhat un-orthodox approach and assumes vectors are *row vectors* rather than column vectors.

[2] See further discussion in Section 4.3.

Leave-one out We must assess the trained function's accuracy on instances it has not seen during training. One solution is to perform *leave-one-out cross-validation*: train k functions $f_{1:k}$, each time leaving out a different input example $\boldsymbol{x}_i$, and evaluating the resulting function $f_i()$ on its ability to predict $\boldsymbol{x}_i$. Then train another function $f()$ on the entire trainings set $\boldsymbol{x}_{1:k}$. Assuming that the training set is a representative sample of the population, this percentage of functions $f_i()$ that produced correct prediction on the left-out samples is a good approximation of the accuracy of $f()$ on new inputs. However, this process is very costly in terms of computation time, and is used only in cases where the number of annotated examples k is very small (less than a hundred or so). In language processing tasks, we very often encounter training sets with well over 10^5 examples.

Held-out set A more efficient solution in terms of computation time is to split the training set into two subsets, say in a 80%/20% split, train a model on the larger subset (the *training set*), and test its accuracy on the smaller subset (the *held-out set*). This will give us a reasonable estimate on the accuracy of the trained function, or at least allow us to compare the quality of different trained models. However, it is somewhat wasteful in terms training samples. One could then re-train a model on the entire set. However, as the model is trained on substantially more data, the error estimates of the model trained on less data may not be accurate. This is generally a good problem to have, as more training data is likely to result in better rather than worse predictors.[3]

Some care must be taken when performing the split—in general it is better to shuffle the examples prior to splitting them, to ensure a balanced distribution of examples between the training and held-out sets (for example, you want the proportion of gold labels in the two sets to be similar). However, sometimes a random split is not a good option: consider the case where your input are news articles collected over several months, and your model is expected to provide predictions for new stories. Here, a random split will over-estimate the model's quality: the training and held-out examples will be from the same time period, and hence on more similar stories, which will not be the case in practice. In such cases, you want to ensure that the training set has older news stories and the held-out set newer ones—to be as similar as possible to how the trained model will be used in practice.

A three-way split The split into train and held-out sets works well if you train a single model and wants to assess its quality. However, in practice you often train several models, compare their quality, and select the best one. Here, the two-way split approach is insufficient—selecting the best model according to the held-out set's accuracy will result in an overly optimistic estimate of the model's quality. You don't know if the chosen settings of the final classifier are good in general, or are just good for the particular examples in the held-out sets. The problem will be even worse if you perform error analysis based on the held-out set, and change the features or the architecture of the model based on the observed errors. You don't know if your improvements based on the held-

[3]Note, however, that some setting in the training procedure, in particular the learning rate and regularization weight may be sensitive to the training set size, and tuning them based on some data and then re-training a model with the same settings on larger data may produce sub-optimal results.

out sets will carry over to new instances. The accepted methodology is to use a three-way split of the data into train, validation (also called *development*), and test sets. This gives you two held-out sets: a *validation set* (also called *development set*), and a *test set*. All the experiments, tweaks, error analysis, and model selection should be performed based on the validation set. Then, a single run of the final model over the test set will give a good estimate of its expected quality on unseen examples. It is important to keep the test set as pristine as possible, running as few experiments as possible on it. Some even advocate that you should not even look at the examples in the test set, so as to not bias the way you design your model.

2.3 LINEAR MODELS

Now that we have established some methodology, we return to describe linear models for binary and multi-class classification.

2.3.1 BINARY CLASSIFICATION

In binary classification problems we have a single output, and thus use a restricted version of Equation (2.1) in which $d_{out} = 1$, making $\boldsymbol{w}$ a vector and b a scalar.

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w} + b. \tag{2.2}$$

The range of the linear function in Equation (2.2) is $[-\infty, +\infty]$. In order to use it for binary classification, it is common to pass the output of $f(\boldsymbol{x})$ through the $sign$ function, mapping negative values to -1 (the negative class) and non-negative values to $+1$ (the positive class).

Consider the task of predicting which of two neighborhoods an apartment is located at, based on the apartment's price and size. Figure 2.1 shows a 2D plot of some apartments, where the x-axis denotes the monthly rent price in USD, while the y-axis is the size in square feet. The blue circles are for Dupont Circle, DC and the green crosses are in Fairfax, VA. It is evident from the plot that we can separate the two neighborhoods using a straight line—apartments in Dupont Circle tend to be more expensive than apartments in Fairfax of the same size.[4] The dataset is *linearly separable*: the two classes can be separated by a straight line.

Each data-point (an apartment) can be represented as a 2-dimensional (2D) vector $\boldsymbol{x}$ where $\boldsymbol{x}_{[0]}$ is the apartment's size and $\boldsymbol{x}_{[1]}$ is its price. We then get the following linear model:

$$\begin{aligned}\hat{y} = \operatorname{sign}(f(\boldsymbol{x})) &= \operatorname{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b) \\ &= \operatorname{sign}(\text{size} \times w_1 + \text{price} \times w_2 + b),\end{aligned}$$

where $\cdot$ is the dot-product operation, b and $\boldsymbol{w} = [w_1, w_2]$ are free parameters, and we predict Fairfax if $\hat{y} \geq 0$ and Dupont Circle otherwise. The goal of learning is setting the values of w_1,

[4] Note that looking at either size or price alone would not allow us to cleanly separate the two groups.

Figure 2.1: Housing data: rent price in USD vs. size in square ft. Data source: Craigslist ads, collected from June 7–15, 2015.

w_2, and b such that the predictions are correct for all data-points we observe.[5] We will discuss learning in Section 2.7 but for now consider that we expect the learning procedure to set a high value to w_1 and a low value to w_2. Once the model is trained, we can classify new data-points by feeding them into this equation.

It is sometimes not possible to separate the data-points using a straight line (or, in higher dimensions, a linear hyperplane)—such datasets are said to be *nonlinearly separable*, and are beyond the hypothesis class of linear classifiers. The solution would be to either move to a higher dimension (add more features), move to a richer hypothesis class, or allow for some mis-classification.[6]

[5]Geometrically, for a given $\boldsymbol{w}$ the points $\boldsymbol{x} \cdot \boldsymbol{w} + b = 0$ define a *hyperplane* (which in two dimensions corresponds to a line) that separates the space into two regions. The goal of learning is then finding a hyperplane such that the classification induced by it is correct.

[6]Misclassifying some of the examples is sometimes a good idea. For example, if we have reason to believe some of the data-points are *outliers*—examples that belong to one class, but are labeled by mistake as belonging to the other class.

Feature Representations In the example above, each data-point was a pair of size and price measurements. Each of these properties is considered a *feature* by which we classify the data-point. This is very convenient, but in most cases the data-points are not given to us directly as lists of features, but as real-world objects. For example, in the apartments example we may be given a list of apartments to classify. We then need to make a concious decision and select the measurable properties of the apartments that we believe will be useful features for the classification task at hand. Here, it proved effective to focus on the price and the size. We could also look at additional properties, such as the number of rooms, the height of the ceiling, the type of floor, the geo-location coordinates, and so on. After deciding on a set of features, we create a *feature extraction* function that maps a real world object (i.e., an apartment) to a vector of measurable quantities (price and size) which can be used as inputs to our models. The choice of the features is crucial to the success of the classification accuracy, and is driven by the informativeness of the features, and their availability to us (the geo-location coordinates are much better predictors of the neighborhood than the price and size, but perhaps we only observe listings of past transactions, and do not have access to the geo-location information). When we have two features, it is easy to plot the data and see the underlying structures. However, as we see in the next example, we often use many more than just two features, making plotting and precise reasoning impractical.

A central part in the design of linear models, which we mostly gloss over in this text, is the design of the feature function (so called *feature engineering*). One of the promises of deep learning is that it vastly simplifies the feature-engineering process by allowing the model designer to specify a small set of core, basic, or "natural" features, and letting the trainable neural network architecture combine them into more meaningful higher-level features, or *representations*. However, one still needs to specify a suitable set of core features, and tie them to a suitable architecture. We discuss common features for textual data in Chapters 6 and 7.

We usually have many more than two features. Moving to a language setup, consider the task of distinguishing documents written in English from documents written in German. It turns out that letter frequencies make for quite good predictors (features) for this task. Even more informative are counts of letter *bigrams*, i.e., pairs of consecutive letters.[7] Assuming we have an alphabet of 28 letters (a–z, space, and a special symbol for all other characters including digits, punctuations, etc.) we represent a document as a 28×28 dimensional vector $\boldsymbol{x} \in \mathbb{R}^{784}$, where each entry $\boldsymbol{x}_{[i]}$ represents a count of a particular letter combination in the document, normalized by the document's length. For example, denoting by x_{ab} the entry of $\boldsymbol{x}$ corresponding to the

[7]While one may think that *words* will also be good predictors, letters, or letter-bigrams are far more robust: we are likely to encounter a new document without any of the words we observed in the training set, while a document without any of the distinctive letter-bigrams is significantly less likely.

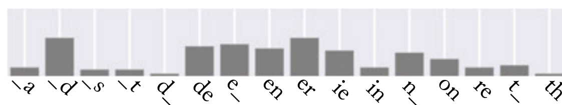
letter-bigram ab:

$$x_{ab} = \frac{\#_{ab}}{|D|}, \tag{2.3}$$

where $\#_{ab}$ is the number of times the bigram ab appears in the document, and $|D|$ is the total number of bigrams in the document (the document's length).

Figure 2.2: Character-bigram histograms for documents in English (left, blue) and German (right, green). Underscores denote spaces.

Figure 2.2 shows such bigram histograms for several German and English texts. For readability, we only show the top frequent character-bigrams and not the entire feature vectors. On the left, we see the bigrams of the English texts, and on the right of the German ones. There are clear patterns in the data, and, given a new item, such as:



you could probably tell that it is more similar to the German group than to the English one. Note, however, that you couldn't use a single definite rule such as "if it has th its English" or "if it has ie its German": while German texts have considerably less th than English, the th may and does occur in German texts, and similarly the ie combination does occur in English. The decision requires weighting different factors relative to each other. Let's formalize the problem in a machine-learning setup.

We can again use a linear model:

$$\begin{aligned}\hat{y} = \operatorname{sign}(f(\boldsymbol{x})) &= \operatorname{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b) \\ &= \operatorname{sign}(x_{aa} \times w_{aa} + x_{ab} \times w_{ab} + x_{ac} \times w_{ac} \ldots + b).\end{aligned} \tag{2.4}$$

A document will be considered English if $f(\boldsymbol{x}) \geq 0$ and as German otherwise. Intuitively, learning should assign large positive values to $\boldsymbol{w}$ entries associated with letter pairs that are much more common in English than in German (i.e., th) negative values to letter pairs that are much more common in German than in English (ie, en), and values around zero to letter pairs that are either common or rare in both languages.

Note that unlike the 2D case of the housing data (price vs. size), here we cannot easily visualize the points and the decision boundary, and the geometric intuition is likely much less clear. In general, it is difficult for most humans to think of the geometries of spaces with more than three dimensions, and it is advisable to think of linear models in terms of assigning weights to features, which is easier to imagine and reason about.

2.3.2 LOG-LINEAR BINARY CLASSIFICATION

The output $f(\boldsymbol{x})$ is in the range $[-\infty, \infty]$, and we map it to one of two classes $\{-1, +1\}$ using the $sign$ function. This is a good fit if all we care about is the assigned class. However, we may be interested also in the confidence of the decision, or the probability that the classifier assigns to the class. An alternative that facilitates this is to map instead to the range $[0, 1]$, by pushing the output through a squashing function such as the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$, resulting in:

$$\hat{y} = \sigma(f(\boldsymbol{x})) = \frac{1}{1 + e^{-(\boldsymbol{x} \cdot \boldsymbol{w} + b)}}. \tag{2.5}$$

Figure 2.3 shows a plot of the sigmoid function. It is monotonically increasing, and maps values to the range $[0, 1]$, with 0 being mapped to $\frac{1}{2}$. When used with a suitable *loss function* (discussed in Section 2.7.1) the binary predictions made through the log-linear model can be interpreted as class membership probability estimates $\sigma(f(\boldsymbol{x})) = P(\hat{y} = 1 \mid \boldsymbol{x})$ of $\boldsymbol{x}$ belonging to the positive class. We also get $P(\hat{y} = 0 \mid \boldsymbol{x}) = 1 - P(\hat{y} = 1 \mid \boldsymbol{x}) = 1 - \sigma(f(\boldsymbol{x}))$. The closer the value is to 0 or 1 the more certain the model is in its class membership prediction, with the value of 0.5 indicating model uncertainty.

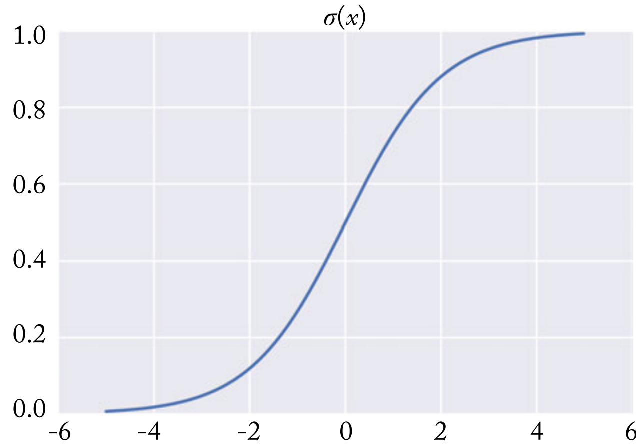


Figure 2.3: The sigmoid function $\sigma(x)$.

2.3.3 MULTI-CLASS CLASSIFICATION

The previous examples were of *binary classification*, where we had two possible classes. Binary-classification cases exist, but most classification problems are of a *multi-class* nature, in which we should assign an example to one of k different classes. For example, we are given a document and asked to classify it into one of six possible languages: *English, French, German, Italian, Spanish, Other*. A possible solution is to consider six weight vectors $\boldsymbol{w}^{\text{En}}, \boldsymbol{w}^{\text{Fr}}, \ldots$ and biases, one for each language, and predict the language resulting in the highest score:[8]

$$\hat{y} = f(\boldsymbol{x}) = \operatorname*{argmax}_{L \in \{\text{En},\text{Fr},\text{Gr},\text{It},\text{Sp},\text{O}\}} \boldsymbol{x} \cdot \boldsymbol{w}^{L} + b^{L}. \tag{2.6}$$

The six sets of parameters $\boldsymbol{w}^{L} \in \mathbb{R}^{784}$, b^{L} can be arranged as a matrix $\boldsymbol{W} \in \mathbb{R}^{784\times 6}$ and vector $\boldsymbol{b} \in \mathbb{R}^{6}$, and the equation re-written as:

$$\begin{aligned} \hat{\boldsymbol{y}} &= f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b} \\ \text{prediction} &= \hat{y} = \operatorname*{argmax}_{i} \hat{\boldsymbol{y}}_{[i]}. \end{aligned} \tag{2.7}$$

Here $\hat{\boldsymbol{y}} \in \mathbb{R}^{6}$ is a vector of the scores assigned by the model to each language, and we again determine the predicted language by taking the argmax over the entries of $\hat{\boldsymbol{y}}$.

[8]There are many ways to model multi-class classification, including binary-to-multi-class reductions. These are beyond the scope of this book, but a good overview can be found in Allwein et al. [2000].

2.4 REPRESENTATIONS

Consider the vector $\hat{\boldsymbol{y}}$ resulting from applying Equation 2.7 of a trained model to a document. The vector can be considered as a *representation* of the document, capturing the properties of the document that are important to us, namely the scores of the different languages. The representation $\hat{\boldsymbol{y}}$ contains strictly more information than the prediction $\hat{y} = \text{argmax}_i\ \hat{\boldsymbol{y}}_{[i]}$: for example, $\hat{\boldsymbol{y}}$ can be used to distinguish documents in which the main language in German, but which also contain a sizeable amount of French words. By clustering documents based on their vector representations as assigned by the model, we could perhaps discover documents written in regional dialects, or by multilingual authors.

The vectors $\boldsymbol{x}$ containing the normalized letter-bigram counts for the documents are also representations of the documents, arguably containing a similar kind of information to the vectors $\hat{\boldsymbol{y}}$. However, the representations in $\hat{\boldsymbol{y}}$ is more compact (6 entries instead of 784) and more specialized for the language prediction objective (clustering by the vectors $\boldsymbol{x}$ would likely reveal document similarities that are not due to a particular mix of languages, but perhaps due to the document's topic or writing styles).

The trained matrix $\boldsymbol{W} \in \mathbb{R}^{784\times 6}$ can also be considered as containing learned representations. As demonstrated in Figure 2.4, we can consider two views of $\boldsymbol{W}$, as rows or as columns. Each of the 6 columns of $\boldsymbol{W}$ correspond to a particular language, and can be taken to be a 784-dimensional vector representation of this language in terms of its characteristic letter-bigram patterns. We can then cluster the 6 language vectors according to their similarity. Similarly, each of the 784 rows of $\boldsymbol{W}$ correspond to a particular letter-bigram, and provide a 6-dimensional vector representation of that bigram in terms of the languages it prompts.

Representations are central to deep learning. In fact, one could argue that the main power of deep-learning is the ability to learn good representations. In the linear case, the representations are interpretable, in the sense that we can assign a meaningful interpretation to each dimension in the representation vector (e.g., each dimension corresponds to a particular language or letter-bigram). This is in general not the case—deep learning models often learn a cascade of representations of the input that build on top of each other, in order to best model the problem at hand, and these representations are often not interpretable—we do not know which properties of the input they capture. However, they are still very useful for making predictions. Moreover, at the boundaries of the model, i.e., at the input and the output, we get representations that correspond to particular aspects of the input (i.e., a vector representation for each letter-bigram) or the output (i.e., a vector representation of each of the output classes). We will get back to this in Section 8.3 after discussing neural networks and encoding categorical features as dense vectors. It is recommended that you return to this discussion once more after reading that section.

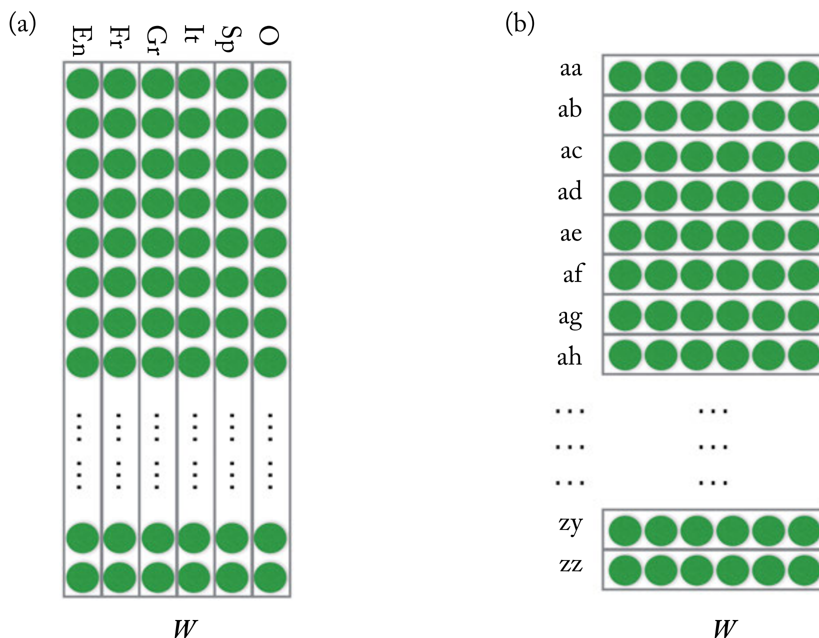


Figure 2.4: Two views of the $\boldsymbol{W}$ matrix. (a) Each column corresponds to a language. (b) Each row corresponds to a letter bigram.

2.5 ONE-HOT AND DENSE VECTOR REPRESENTATIONS

The input vector $\boldsymbol{x}$ in our language classification example contains the normalized bigram counts in the document D. This vector can be decomposed into an average of $|D|$ vectors, each corresponding to a particular document position i:

$$\boldsymbol{x} = \frac{1}{|D|} \sum_{i=1}^{|D|} \boldsymbol{x}^{D_{[i]}}; \tag{2.8}$$

here, $D_{[i]}$ is the bigram at document position i, and each vector $\boldsymbol{x}^{D_{[i]}} \in \mathbb{R}^{784}$ is a *one-hot* vector, in which all entries are zero except the single entry corresponding to the letter bigram $D_{[i]}$, which is 1.

The resulting vector $\boldsymbol{x}$ is commonly referred to as an *averaged bag of bigrams* (more generally *averaged bag of words*, or just *bag of words*). Bag-of-words (BOW) representations contain information about the identities of all the "words" (here, bigrams) of the document, without considering their order. A one-hot representation can be considered as a bag-of-a-single-word.

The view of the rows of the matrix $\boldsymbol{W}$ as representations of the letter bigrams suggests an alternative way of computing the document representation vector $\hat{\boldsymbol{y}}$ in Equation (2.7). Denoting

by $\boldsymbol{W}^{D_{[i]}}$ the row of W corresponding to the bigram $D_{[i]}$, we can take the representation $\boldsymbol{y}$ of a document D to be the average of the representations of the letter-bigrams in the document:

$$\hat{\boldsymbol{y}} = \frac{1}{|D|} \sum_{i=1}^{|D|} \boldsymbol{W}^{D_{[i]}}. \tag{2.9}$$

This representation is often called a *continuous bag of words* (CBOW), as it is composed of a sum of word representations, where each "word" representation is a low-dimensional, continuous vector.

We note that Equation (2.9) and the term $\boldsymbol{x} \cdot \boldsymbol{W}$ in Equation (2.7) are equivalent. To see why, consider:

$$\begin{aligned}
\boldsymbol{y} &= \boldsymbol{x} \cdot \boldsymbol{W} \\
&= \left(\frac{1}{|D|} \sum_{i=1}^{|D|} \boldsymbol{x}^{D_{[i]}} \right) \cdot \boldsymbol{W} \\
&= \frac{1}{|D|} \sum_{i=1}^{|D|} (\boldsymbol{x}^{D_{[i]}} \cdot \boldsymbol{W}) \\
&= \frac{1}{|D|} \sum_{i=1}^{|D|} \boldsymbol{W}^{D_{[i]}}.
\end{aligned} \tag{2.10}$$

In other words, the continuous-bag-of-words (CBOW) representation can be obtained either by summing word-representation vectors or by multiplying a bag-of-words vector by a matrix in which each row corresponds to a dense word representation (such matrices are also called *embedding matrices*). We will return to this point in Chapter 8 (in particular Section 8.3) when discussing feature representations in deep learning models for text.

2.6 LOG-LINEAR MULTI-CLASS CLASSIFICATION

In the binary case, we transformed the linear prediction into a probability estimate by passing it through the sigmoid function, resulting in a log-linear model. The analog for the multi-class case is passing the score vector through the *softmax* function:

$$\text{softmax}(\boldsymbol{x})_{[i]} = \frac{e^{\boldsymbol{x}_{[i]}}}{\sum_j e^{\boldsymbol{x}_{[j]}}}. \tag{2.11}$$

Resulting in:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})$$
$$\hat{\boldsymbol{y}}_{[i]} = \frac{e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[i]}}}{\sum_j e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[j]}}}. \tag{2.12}$$

The *softmax* transformation forces the values in $\hat{\boldsymbol{y}}$ to be positive and sum to 1, making them interpretable as a probability distribution.

2.7 TRAINING AS OPTIMIZATION

Recall that the input to a supervised learning algorithm is a *training set* of n training examples $x_{1:n} = x_1, x_2, \ldots, x_n$ together with corresponding labels $y_{1:n} = y_1, y_2, \ldots, y_n$. Without loss of generality, we assume that the desired inputs and outputs are vectors: $\boldsymbol{x}_{1:n}$, $\boldsymbol{y}_{1:n}$.[9]

The goal of the algorithm is to return a function $f()$ that accurately maps input examples to their desired labels, i.e., a function $f()$ such that the predictions $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ over the training set are accurate. To make this more precise, we introduce the notion of a *loss function*, quantifying the loss suffered when predicting $\hat{\boldsymbol{y}}$ while the true label is $\boldsymbol{y}$. Formally, a loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ assigns a numerical score (a scalar) to a predicted output $\hat{\boldsymbol{y}}$ given the true expected output $\boldsymbol{y}$. The loss function should be bounded from below, with the minimum attained only for cases where the prediction is correct.

The parameters of the learned function (the matrix $\boldsymbol{W}$ and the biases vector $\boldsymbol{b}$) are then set in order to minimize the loss L over the training examples (usually, it is the sum of the losses over the different training examples that is being minimized).

Concretely, given a labeled training set $(\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n})$, a per-instance loss function L and a parameterized function $f(\boldsymbol{x}; \Theta)$ we define the corpus-wide loss with respect to the parameters Θ as the average loss over all training examples:

$$\mathcal{L}(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i). \tag{2.13}$$

In this view, the training examples are fixed, and the values of the parameters determine the loss. The goal of the training algorithm is then to set the values of the parameters Θ such that the value of $\mathcal{L}$ is minimized:

$$\hat{\Theta} = \underset{\Theta}{\text{argmin}}\, \mathcal{L}(\Theta) = \underset{\Theta}{\text{argmin}}\, \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i). \tag{2.14}$$

Equation (2.14) attempts to minimize the loss at all costs, which may result in *overfitting* the training data. To counter that, we often pose soft restrictions on the form of the solution. This

[9] In many cases it is natural to think of the expected output as a scalar (class assignment) rather than a vector. In such cases, $\boldsymbol{y}$ is simply the corresponding one-hot vector, and $\text{argmax}_i\ \boldsymbol{y}_{[i]}$ is the corresponding class assignment.

is done using a function $R(\Theta)$ taking as input the parameters and returning a scalar that reflect their "complexity," which we want to keep low. By adding R to the objective, the optimization problem needs to balance between low loss and low complexity:

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} \left(\overbrace{\frac{1}{n} \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i)}^{loss} + \overbrace{\lambda R(\Theta)}^{regularization} \right). \tag{2.15}$$

The function R is called a *regularization term*. Different combinations of loss functions and regularization criteria result in different learning algorithms, with different inductive biases.

We now turn to discuss common loss functions (Section 2.7.1), followed by a discussion of regularization and regularizers (Section 2.7.2). Then, in Section 2.8 we present an algorithm for solving the minimization problem (Equation (2.15)).

2.7.1 LOSS FUNCTIONS

The loss can be an arbitrary function mapping two vectors to a scalar. For practical purposes of optimization, we restrict ourselves to functions for which we can easily compute gradients (or sub-gradients).[10] In most cases, it is sufficient and advisable to rely on a common loss function rather than defining your own. For a detailed discussion and theoretical treatment of loss functions for binary classification, see Zhang [2004]. We now discuss some loss functions that are commonly used with linear models and with neural networks in NLP.

Hinge (binary) For binary classification problems, the classifier's output is a single scalar $\tilde{y}$ and the intended output y is in $\{+1, -1\}$. The classification rule is $\hat{y} = \operatorname{sign}(\tilde{y})$, and a classification is considered correct if $y \cdot \tilde{y} > 0$, meaning that y and $\tilde{y}$ share the same sign. The hinge loss, also known as margin loss or SVM loss, is defined as:

$$L_{\text{hinge(binary)}}(\tilde{y}, y) = \max(0, 1 - y \cdot \tilde{y}). \tag{2.16}$$

The loss is 0 when y and $\tilde{y}$ share the same sign and $|\tilde{y}| \geq 1$. Otherwise, the loss is linear. In other words, the binary hinge loss attempts to achieve a correct classification, with a *margin* of at least 1.

Hinge (multi-class) The hinge loss was extended to the multi-class setting by Crammer and Singer [2002]. Let $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}_{[1]}, \ldots, \hat{\boldsymbol{y}}_{[n]}$ be the classifier's output vector, and $\boldsymbol{y}$ be the one-hot vector for the correct output class.

The classification rule is defined as selecting the class with the highest score:

$$\text{prediction} = \underset{i}{\operatorname{argmax}}\, \hat{\boldsymbol{y}}_{[i]}. \tag{2.17}$$

[10] A gradient of a function with k variables is a collection of k partial derivatives, one according to each of the variables. Gradients are discussed further in Section 2.8.

Denote by $t = \text{argmax}_i\ \boldsymbol{y}_{[i]}$ the correct class, and by $k = \text{argmax}_{i \neq t}\ \hat{\boldsymbol{y}}_{[i]}$ the highest scoring class such that $k \neq t$. The multi-class hinge loss is defined as:

$$L_{\text{hinge(multi-class)}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \max(0, 1 - (\hat{\boldsymbol{y}}_{[t]} - \hat{\boldsymbol{y}}_{[k]})). \tag{2.18}$$

The multi-class hinge loss attempts to score the correct class above all other classes with a margin of at least 1.

Both the binary and multi-class hinge losses are intended to be used with linear outputs. The hinge losses are useful whenever we require a hard decision rule, and do not attempt to model class membership probability.

Log loss The log loss is a common variation of the hinge loss, which can be seen as a "soft" version of the hinge loss with an infinite margin [LeCun et al., 2006]:

$$L_{log}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \log(1 + exp(-(\hat{\boldsymbol{y}}_{[t]} - \hat{\boldsymbol{y}}_{[k]})). \tag{2.19}$$

Binary cross entropy The binary cross-entropy loss, also referred to as *logistic loss* is used in binary classification with conditional probability outputs. We assume a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$. The classifier's output $\tilde{y}$ is transformed using the sigmoid (also called the logistic) function $\sigma(x) = 1/(1 + e^{-x})$ to the range [0, 1], and is interpreted as the conditional probability $\hat{y} = \sigma(\tilde{y}) = P(y = 1|\boldsymbol{x})$. The prediction rule is:

$$\text{prediction} = \begin{cases} 0 & \hat{y} < 0.5 \\ 1 & \hat{y} \geq 0.5. \end{cases}$$

The network is trained to maximize the log conditional probability $\log P(y = 1|\boldsymbol{x})$ for each training example $(\boldsymbol{x}, y)$. The logistic loss is defined as:

$$L_{\text{logistic}}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \tag{2.20}$$

The logistic loss is useful when we want the network to produce class conditional probability for a binary classification problem. When using the logistic loss, it is assumed that the output layer is transformed using the sigmoid function.

Categorical cross-entropy loss The categorical cross-entropy loss (also referred to as *negative log likelihood*) is used when a probabilistic interpretation of the scores is desired.

Let $\boldsymbol{y} = \boldsymbol{y}_{[1]}, \ldots, \boldsymbol{y}_{[n]}$ be a vector representing the true multinomial distribution over the labels $1, \ldots, n$,[11] and let $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}_{[1]}, \ldots, \hat{\boldsymbol{y}}_{[n]}$ be the linear classifier's output, which was transformed by the softmax function (Section 2.6), and represent the class membership conditional distribution $\hat{\boldsymbol{y}}_{[i]} = P(y = i|\boldsymbol{x})$. The categorical cross entropy loss measures the dissimilarity between the true label distribution $\boldsymbol{y}$ and the predicted label distribution $\hat{\boldsymbol{y}}$, and is defined as cross entropy:

$$L_{\text{cross-entropy}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_i \boldsymbol{y}_{[i]} \log(\hat{\boldsymbol{y}}_{[i]}). \tag{2.21}$$

[11]This formulation assumes an instance can belong to several classes with some degree of certainty.

For hard-classification problems in which each training example has a single correct class assignment, $\boldsymbol{y}$ is a one-hot vector representing the true class. In such cases, the cross entropy can be simplified to:

$$L_{\text{cross-entropy(hard classification)}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\log(\hat{\boldsymbol{y}}_{[t]}), \tag{2.22}$$

where t is the correct class assignment. This attempts to set the probability mass assigned to the correct class t to 1. Because the scores $\hat{\boldsymbol{y}}$ have been transformed using the softmax function to be non-negative and sum to one, increasing the mass assigned to the correct class means decreasing the mass assigned to all the other classes.

The cross-entropy loss is very common in the log-linear models and the neural networks literature, and produces a multi-class classifier which does not only predict the one-best class label but also predicts a distribution over the possible labels. When using the cross-entropy loss, it is assumed that the classifier's output is transformed using the softmax transformation.

Ranking losses In some settings, we are not given supervision in term of labels, but rather as pairs of correct and incorrect items $\boldsymbol{x}$ and $\boldsymbol{x}'$, and our goal is to score correct items above incorrect ones. Such training situations arise when we have only positive examples, and generate negative examples by corrupting a positive example. A useful loss in such scenarios is the margin-based ranking loss, defined for a pair of correct and incorrect examples:

$$L_{\text{ranking(margin)}}(\boldsymbol{x}, \boldsymbol{x}') = \max(0, 1 - (f(\boldsymbol{x}) - f(\boldsymbol{x}'))), \tag{2.23}$$

where $f(\boldsymbol{x})$ is the score assigned by the classifier for input vector $\boldsymbol{x}$. The objective is to score (rank) correct inputs over incorrect ones with a margin of at least 1.

A common variation is to use the log version of the ranking loss:

$$L_{\text{ranking(log)}}(\boldsymbol{x}, \boldsymbol{x}') = \log(1 + \exp(-(f(\boldsymbol{x}) - f(\boldsymbol{x}')))). \tag{2.24}$$

Examples using the ranking hinge loss in language tasks include training with the auxiliary tasks used for deriving pre-trained word embeddings (see Section 10.4.2), in which we are given a correct word sequence and a corrupted word sequence, and our goal is to score the correct sequence above the corrupt one [Collobert and Weston, 2008]. Similarly, Van de Cruys [2014] used the ranking loss in a selectional-preferences task, in which the network was trained to rank correct verb-object pairs above incorrect, automatically derived ones, and Weston et al. [2013] trained a model to score correct (head, relation, tail) triplets above corrupted ones in an information-extraction setting. An example of using the ranking log loss can be found in Gao et al. [2014]. A variation of the ranking log loss allowing for a different margin for the negative and positive class is given in dos Santos et al. [2015].

2.7.2 REGULARIZATION

Consider the optimization problem in Equation (2.14). It may admit multiple solutions, and, especially in higher dimensions, it can also over-fit. Consider our language identification example, and a setting in which one of the documents in the training set (call it $\boldsymbol{x}_o$) is an outlier: it is actually in German, but is labeled as French. In order to drive the loss down, the learner can identify features (letter bigrams) in $\boldsymbol{x}_o$ that occur in only few other documents, and give them very strong weights toward the (incorrect) French class. Then, for other German documents in which these features occur, which may now be mistakenly classified as French, the learner will find other German letter bigrams and will raise their weights in order for the documents to be classified as German again. This is a bad solution to the learning problem, as it learns something incorrect, and can cause test German documents which share many words with $\boldsymbol{x}_o$ to be mistakenly classified as French. Intuitively, we would like to control for such cases by driving the learner away from such misguided solutions and toward more natural ones, in which it is OK to mis-classify a few examples if they don't fit well with the rest.

This is achieved by adding a *regularization term* R to the optimization objective, whose job is to control the complexity of the parameter value, and avoid cases of overfitting:

$$\begin{aligned}\hat{\Theta} &= \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\Theta) + \lambda R(\Theta) \\ &= \underset{\Theta}{\operatorname{argmin}}\, \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i;\Theta), \boldsymbol{y}_i) + \lambda R(\Theta).\end{aligned} \tag{2.25}$$

The regularization term considers the parameter values, and scores their complexity. We then look for parameter values that have both a low loss and low complexity. A hyperparameter[12] λ is used to control the amount of regularization: do we favor simple model over low loss ones, or vice versa. The value of λ has to be set manually, based on the classification performance on a development set. While Equation (2.25) has a single regularization function and λ value for all the parameters, it is of course possible to have a different regularizer for each item in Θ.

In practice, the regularizers R equate complexity with large weights, and work to keep the parameter values low. In particular, the regularizers R measure the norms of the parameter matrices, and drive the learner toward solutions with low norms. Common choices for R are the L_2 norm, the L_1 norm, and the elastic-net.

L_2 regularization In L_2 regularization, R takes the form of the squared L_2 norm of the parameters, trying to keep the sum of the squares of the parameter values low:

$$R_{L_2}(\boldsymbol{W}) = ||\boldsymbol{W}||_2^2 = \sum_{i,j}(\boldsymbol{W}_{[i,j]})^2. \tag{2.26}$$

[12] A *hyperparameter* is a parameter of the model which is not learned as part of the optimization process, but needs to be set by hand.

The L_2 regularizer is also called a *gaussian prior* or *weight decay*.

Note that L_2 regularized models are severely punished for high parameter weights, but once the value is close enough to zero, their effect becomes negligible. The model will prefer to decrease the value of one parameter with high weight by 1 than to decrease the value of ten parameters that already have relatively low weights by 0.1 each.

L_1 regularization In L_1 regularization, R takes the form of the L_1 norm of the parameters, trying to keep the sum of the absolute values of the parameters low:

$$R_{L_1}(\boldsymbol{W}) = ||\boldsymbol{W}||_1 = \sum_{i,j} |\boldsymbol{W}_{[i,j]}|. \tag{2.27}$$

In contrast to L_2, the L_1 regularizer is punished uniformly for low and high values, and has an incentive to decrease all the non-zero parameter values toward zero. It thus encourages a sparse solutions—models with many parameters with a zero value. The L_1 regularizer is also called a *sparse prior* or *lasso* [Tibshirani, 1994].

Elastic-Net The elastic-net regularization [Zou and Hastie, 2005] combines both L_1 and L_2 regularization:

$$R_{\text{elastic-net}}(\boldsymbol{W}) = \lambda_1 R_{L_1}(\boldsymbol{W}) + \lambda_2 R_{L_2}(\boldsymbol{W}). \tag{2.28}$$

Dropout Another form of regularization which is very effective in neural networks is *Dropout*, which we discuss in Section 4.6.

2.8 GRADIENT-BASED OPTIMIZATION

In order to train the model, we need to solve the optimization problem in Equation (2.25). A common solution is to use a gradient-based method. Roughly speaking, gradient-based methods work by repeatedly computing an estimate of the loss $\mathcal{L}$ over the training set, computing the gradients of the parameters Θ with respect to the loss estimate, and moving the parameters in the opposite directions of the gradient. The different optimization methods differ in how the error estimate is computed, and how "moving in the opposite direction of the gradient" is defined. We describe the basic algorithm, *stochastic gradient descent* (SGD), and then briefly mention the other approaches with pointers for further reading.

Motivating Gradient-based Optimization Consider the task of finding the scalar value x that minimizes a function $y = f(x)$. The canonical approach is computing the second derivative $f''(x)$ of the function, and solving for $f''(x) = 0$ to get the extrema points. For the sake of example, assume this approach cannot be used (indeed, it is challenging to use this approach in function of multiple variables). An alternative approach is a numeric one: compute the first derivative $f'(x)$. Then, start with an initial guess value x_i. Evaluating $u = f'(x_i)$

will give the direction of change. If $u = 0$, then x_i is an optimum point. Otherwise, move in the opposite direction of u by setting $x_{i+1} \leftarrow x_i - \eta u$, where η is a *rate parameter*. With a small enough value of η, $f(x_{i+1})$ will be smaller than $f(x_i)$. Repeating this process (with properly decreasing values of η) will find an optimum point x_i. If the function $f()$ is convex, the optimum will be a global one. Otherwise, the process is only guaranteed to find a local optimum.

Gradient-based optimization simply generalizes this idea for functions with multiple variables. A gradient of a function with k variables is the collections of k partial derivatives, one according to each of the variables. Moving the inputs in the direction of the gradient will increase the value of the function, while moving them in the opposite direction will decrease it. When optimizing the loss $\mathcal{L}(\Theta; \boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n})$, the parameters Θ are considered as inputs to the function, while the training examples are treated as constants.

Convexity In gradient-based optimization, it is common to distinguish between *convex* (or *concave*) functions and *non-convex* (*non-concave*) functions. A *convex function* is a function whose second-derivative is always non-negative. As a consequence, convex functions have a single minimum point. Similarly, *concave functions* are functions whose second-derivatives are always negative or zero, and as a consequence have a single maximum point. Convex (concave) functions have the property that they are easy to minimize (maximize) using gradient-based optimization—simply follow the gradient until an extremum point is reached, and once it is reached we know we obtained the global extremum point. In contrast, for functions that are neither convex nor concave, a gradient-based optimization procedure may converge to a local extremum point, missing the global optimum.

2.8.1 STOCHASTIC GRADIENT DESCENT

An effective method for training linear models is using the SGD algorithm [Bottou, 2012, LeCun et al., 1998a] or a variant of it. SGD is a general optimization algorithm. It receives a function f parameterized by Θ, a loss function L, and desired input and output pairs $\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}$. It then attempts to set the parameters Θ such that the cumulative loss of f on the training examples is small. The algorithm works, as shown in Algorithm 2.1.

The goal of the algorithm is to set the parameters Θ so as to minimize the total loss $\mathcal{L}(\Theta) = \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \theta), \boldsymbol{y}_i)$ over the training set. It works by repeatedly sampling a training example and computing the gradient of the error on the example with respect to the parameters Θ (line 4)—the input and expected output are assumed to be fixed, and the loss is treated as a function of the parameters Θ. The parameters Θ are then updated in the opposite direction of the gradient, scaled by a learning rate η_t (line 5). The learning rate can either be fixed throughout the

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

training process, or decay as a function of the time step t.[13] For further discussion on setting the learning rate, see Section 5.2.

Note that the error calculated in line 3 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss $\mathcal{L}$ that we are aiming to minimize. The noise in the loss computation may result in inaccurate gradients. A common way of reducing this noise is to estimate the error and the gradients based on a sample of m examples. This gives rise to the *minibatch SGD* algorithm (Algorithm 2.2).

In lines 3–6, the algorithm estimates the gradient of the corpus loss based on the minibatch. After the loop, $\hat{\boldsymbol{g}}$ contains the gradient estimate, and the parameters Θ are updated toward $\hat{\boldsymbol{g}}$. The minibatch size can vary in size from $m = 1$ to $m = n$. Higher values provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence. Besides the improved accuracy of the gradients estimation, the minibatch algorithm provides opportunities for improved training efficiency. For modest sizes of m, some computing architectures (i.e., GPUs) allow an efficient parallel implementation of the computation in lines 3–6. With a properly decreasing learning rate, SGD is guaranteed to converge to a global optimum if the function is convex, which is the case for linear and log-linear models coupled with the loss functions and regularizers discussed in this chapter. However, it can also be used to optimize non-convex functions such as multi-layer neural network. While there are no longer guarantees of finding a global optimum, the algorithm proved to be robust and performs well in practice.[14]

[13]Learning rate decay is required in order to prove convergence of SGD.

[14]Recent work from the neural networks literature argue that the non-convexity of the networks is manifested in a proliferation of saddle points rather than local minima [Dauphin et al., 2014]. This may explain some of the success in training neural networks despite using local search techniques.

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a minibatch of m examples $\{(\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_m}, \boldsymbol{y_m})\}$
3: $\hat{\boldsymbol{g}} \leftarrow 0$
4: **for** $i = 1$ to m **do**
5: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
6: $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} +$ gradients of $\frac{1}{m} L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
7: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
8: **return** Θ

2.8.2 WORKED-OUT EXAMPLE

As an example, consider a multi-class linear classifier with hinge loss:

$$\hat{y} = \underset{i}{\operatorname{argmax}}\, \hat{\boldsymbol{y}}_{[i]}$$

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$$

$$\begin{aligned} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) &= \max(0, 1 - (\hat{\boldsymbol{y}}_{[t]} - \hat{\boldsymbol{y}}_{[k]})) \\ &= \max(0, 1 - ((\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})_{[t]} - (\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})_{[k]})) \end{aligned}$$

$$t = \underset{i}{\operatorname{argmax}}\, \boldsymbol{y}_{[i]}$$

$$k = \underset{i}{\operatorname{argmax}}\, \hat{\boldsymbol{y}}_{[i]} \quad i \neq t.$$

We want to set the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ such that the loss is minimized. We need to compute the gradients of the loss with respect to the values $\boldsymbol{W}$ and $\boldsymbol{b}$. The gradient is the collection of the

partial derivatives according to each of the variables:

$$\frac{\partial L(\hat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[1,1]}} & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[1,2]}} & \cdots & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[1,n]}} \\ \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[2,1]}} & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[2,2]}} & \cdots & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[2,n]}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[m,1]}} & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[m,2]}} & \cdots & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{W}_{[m,n]}} \end{pmatrix}$$

$$\frac{\partial L(\hat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \boldsymbol{b}} = \begin{pmatrix} \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{b}_{[1]}} & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{b}_{[2]}} & \cdots & \frac{\partial L(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{b}_{[n]}} \end{pmatrix}.$$

More concretely, we will compute the derivate of the loss w.r.t each of the values $\boldsymbol{W}_{[i,j]}$ and $\boldsymbol{b}_{[j]}$. We begin by expanding the terms in the loss calculation:[15]

$$\begin{aligned} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) &= \max(0, 1 - (\hat{\boldsymbol{y}}_{[t]} - \hat{\boldsymbol{y}}_{[k]})) \\ &= \max(0, 1 - ((\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})_{[t]} - (\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})_{[k]})) \\ &= \max\left(0, 1 - \left(\left(\sum_i \boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,t]} + \boldsymbol{b}_{[t]}\right) - \left(\sum_i \boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,k]} + \boldsymbol{b}_{[k]}\right)\right)\right) \\ &= \max\left(0, 1 - \sum_i \boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,t]} - \boldsymbol{b}_{[t]} + \sum_i \boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,k]} + \boldsymbol{b}_{[k]}\right) \\ t &= \operatorname*{argmax}_i \boldsymbol{y}_{[i]} \\ k &= \operatorname*{argmax}_i \hat{\boldsymbol{y}}_{[i]} \quad i \neq t. \end{aligned}$$

The first observation is that if $1 - (\hat{\boldsymbol{y}}_{[t]} - \hat{\boldsymbol{y}}_{[k]}) \leq 0$ then the loss is 0 and so is the gradient (the derivative of the max operation is the derivative of the maximal value). Otherwise, consider the derivative of $\frac{\partial L}{\partial \boldsymbol{b}_{[i]}}$. For the partial derivative, $\boldsymbol{b}_{[i]}$ is treated as a variable, and all others are considered as constants. For $i \neq k, t$, the term $\boldsymbol{b}_{[i]}$ does not contribute to the loss, and its derivative it is 0. For $i = k$ and $i = t$ we trivially get:

$$\frac{\partial L}{\partial \boldsymbol{b}_{[i]}} = \begin{cases} -1 & i = t \\ 1 & i = k \\ 0 & \text{otherwise.} \end{cases}$$

[15] More advanced derivation techniques allow working with matrices and vectors directly. Here, we stick to high-school level techniques.

Similarly, for $\boldsymbol{W}_{[i,j]}$, only $j = k$ and $j = t$ contribute to the loss. We get:

$$\frac{\partial L}{\partial \boldsymbol{W}_{[i,j]}} = \begin{cases} \frac{\partial(-\boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,t]})}{\partial \boldsymbol{W}_{[i,t]}} & = -\boldsymbol{x}_{[i]} \quad j = t \\ \frac{\partial(\boldsymbol{x}_{[i]} \cdot \boldsymbol{W}_{[i,k]})}{\partial \boldsymbol{W}_{[i,k]}} & = \boldsymbol{x}_{[i]} \quad j = k \\ 0 & \text{otherwise.} \end{cases}$$

This concludes the gradient calculation.

As a simple exercise, the reader should try and compute the gradients of a multi-class linear model with hinge loss and L_2 regularization, and the gradients of multi-class classification with softmax output transformation and cross-entropy loss.

2.8.3 BEYOND SGD

While the SGD algorithm can and often does produce good results, more advanced algorithms are also available. The *SGD+Momentum* [Polyak, 1964] and *Nesterov Momentum* [Nesterov, 1983, 2004, Sutskever et al., 2013] algorithms are variants of SGD in which previous gradients are accumulated and affect the current update. Adaptive learning rate algorithms including AdaGrad [Duchi et al., 2011], AdaDelta [Zeiler, 2012], RMSProp [Tieleman and Hinton, 2012], and Adam [Kingma and Ba, 2014] are designed to select the learning rate for each minibatch, sometimes on a per-coordinate basis, potentially alleviating the need of fiddling with learning rate scheduling. For details of these algorithms, see the original papers or [Bengio et al., 2016, Sections 8.3, 8.4].