Natural Language Processing with Pytorch

Build Intelligent Language Applications Using Deep Learning

Stephen CUI

April 20, 2023

Contents

Chapter 1

Foundational Components of Neural Networks

1.1 The Perceptron: The Simplest Neural Network

Each perceptron unit has an input (x), an output (y), and three "knobs": a set of weights (w), a bias (b), and an activation function (f). The weights and the bias are learned from the data, and the activation function is handpicked depending on the network designer's intuition of the network and its target outputs. Mathematically, we can express this as follows:

$$y = f(\mathbf{w}x + \mathbf{b})$$

t is usually the case that there is more than one input to the perceptron. We can represent this general case using vectors. That is, $\mathbf{x}$, and $\mathbf{w}$ are vectors, and the product of $\mathbf{w}$ and $\mathbf{x}$ is replaced with a dot product:

$$y = f(\mathbf{wx} + \mathbf{b})$$

essentially, a perceptron is a composition of a linear and a nonlinear function. The linear expression wx+b is also known as an *affine transform*.

1.2 Activation Functions

Activation functions are nonlinearities introduced in a neural network to capture complex relationships in data.

1.2.1 Sigmoid

The sigmoid function saturates (i.e., produces extreme valued outputs) very quickly and for a majority of the inputs. This can become a problem because it can lead to the gradients becoming either zero or diverging to an overflowing floating-point value. These phenomena are also known as *vanishing gradient problem* and *exploding gradient problem*, respectively. As a consequence, it is rare to see sigmoid units used in neural networks other than at the output, where the squashing property allows one to interpret outputs as probabilities.

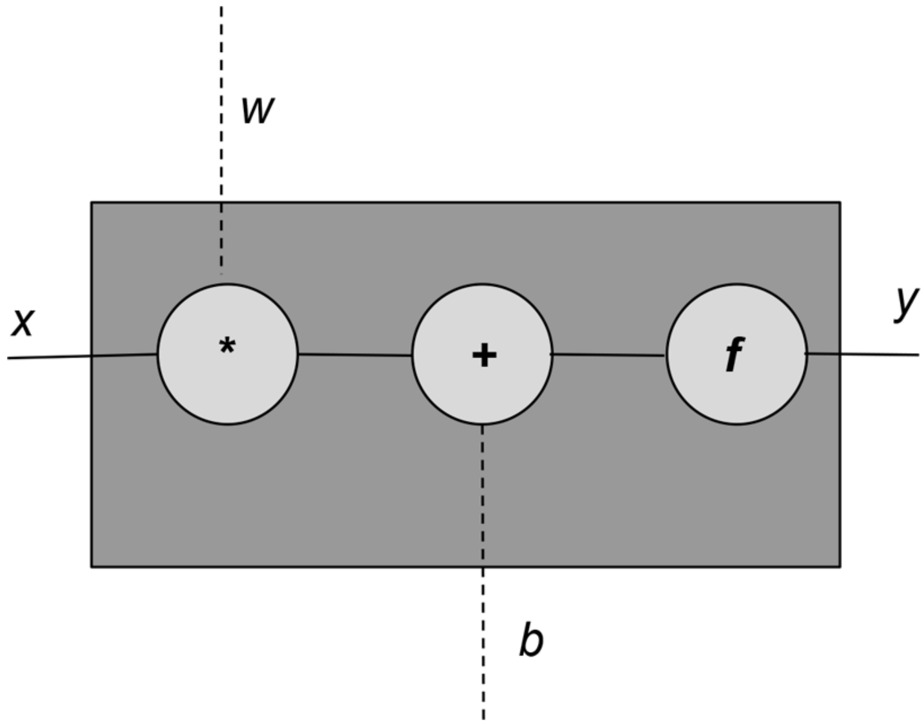

Figure 1.1: The computational graph for a perceptron with an input (x) and an output(y). The weights (w) and bias (b) constitute the parameters of the model.

1.2.2 Tanh

The tanh activation function is a cosmetically different variant of the sigmoid.

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

1.2.3 ReLU

ReLU (pronounced ray-luh) stands for *rectified linear unit*. This is arguably the most important of the activation functions.

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

The clipping effect of ReLU that helps with the vanishing gradient problem can also become an issue, where over time certain outputs in the network can simply become zero and never revive again. This is called the “dying ReLU” problem. To mitigate that effect, variants such as the Leaky ReLU and Parametric ReLU (PReLU) activation functions have proposed, where the leak coefficient a is a learned parameter.

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

1.2.4 Softmax

Another choice for the activation function is the softmax. Like the sigmoid function, the softmax function squashes the output of each unit to be between 0 and 1.

$$softmax(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{k} \exp(x_j)}$$

This transformation is usually paired with a probabilistic training objective, such as categorical cross entropy.

1.3 Loss Functions

1.3.1 Mean Squared Error Loss

For regression problems for which the network's output ($\hat{y}$) and the target ($\hat{y}$) are continuous values, one common loss function is the mean squared error (MSE):

$$L_{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y-\hat{y})^2$$

The MSE is simply the average of the squares of the difference between the predicted and target values. There are several other loss functions that you can use for regression problems, such as mean absolute error (MAE) and root mean squared error (RMSE), but they all involve computing a real-valued distance between the output and target.

1.3.2 Categorical Cross-Entropy Loss

The categorical cross-entropy loss is typically used in a multiclass classification setting in which the outputs are interpreted as predictions of class membership probabilities. The target (y) is a vector of n elements that represents the true multinomial distribution over all the classes. If only one class is correct, this vector is a one-hot vector. The network's output ($\hat{y}$) is also a vector of n elements but represents the network's prediction of the multinomial distribution. Categorical cross entropy will compare these two vectors $(y, \hat{y})$ to measure the loss:

$$L_{cross\_entropy}(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

如果类索引是$[0,C)$的范围，这里C是类别的数量，如果$ignore_index$指定的情况下，交叉熵损失函数接收这个类索引（这个索引未必在类范围内）。这种情况下未衰减的损失可以用下式描述：

$$l(x,y) = L = \{l_1, \ldots, l_N\}^T,\ l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} I(y_n \neq ignore\_index) \tag{1.1}$$

式中x是输入，y是目标，w是权重，C是类别数量，N是minibatch的跨度，更多查看PyTorch的CrossEntropyLoss。

1.3.3 Binary Cross-Entropy Loss

Sometimes, our task involves discriminating between two classes—also known as binary classification. For such situations, it is efficient to use the binary cross-entropy (BCE) loss.

$$l(x,y) = L = \{l_1, \ldots, l_N\}^T,\ l_n = -w_{y_n}[y_n \log x_n + (1-y_n) * \log(1-x_n)] \tag{1.2}$$

1.4 Diving Deep into Supervised Training

1.4.1 Constructing Toy Data

Choosing an optimizer

The PyTorch library offers several choices for an optimizer. Stochastic gradient descent (SGD) is a classic algorithm of choice, but for difficult optimization problems, SGD has convergence issues, often leading to poorer models. The current preferred alternative are adaptive optimizers, such as Adagrad or Adam, which use information about updates over time.

1.4.2 Putting It Together: Gradient-Based Supervised Learning

Let's take a look at how this gradient-stepping algorithm looks. First, any bookkeeping information, such as gradients, currently stored inside the model object is cleared with a function named zero_grad(). Then, the model computes outputs (y_pred) given the input data (x_data). Next, the loss is computed by comparing model outputs (y_pred) to intended targets (y_target). This is the supervised part of the supervised training signal. The PyTorch loss object (criterion) has a function named backward() that iteratively propagates the loss backward through the computational graph and notifies each parameter of its gradient. Finally, the optimizer (opt) instructs the parameters how to update their values knowing the gradient with a function named step().

> In the literature, and also in this notes, the term minibatch is used interchangeably with batch to highlight that each of the batches is significantly smaller than the size of the training data; for example, the training data size could be in the millions, whereas the minibatch could be just a few hundred in size.

1.5 Auxiliary Training Concepts

1.5.1 Knowing When to Stop Training

The most common method is to use a heuristic(启发式算法) called *early stopping*. Early stopping works by keeping track of the performance on the validation dataset from epoch to epoch and noticing when the performance no longer improves. Then, if the performance continues to not improve, the training is terminated. The number of epochs to wait before terminating the training is referred to as the *patience*. In general, the point at which a model stops improving on some dataset is said to be when the model has *converged*. In practice, we rarely wait for a model to completely converge because convergence is time-consuming, and it can lead to overfitting.

1.5.2 Regularization

This smoothness constraint in machine learning is called L2 *regularization*. In PyTorch, you can control this by setting the weight_decay parameter in the optimizer. The larger the weight_decay value, the more likely it is that the optimizer will select the smoother explanation (that is, the stronger is the L2 regularization).

1.6 Example: Classifying Sentiment of Restaurant Reviews

After understanding the dataset, you will see a pattern defining three assisting classesthat is repeated throughout this book and is used to transform text data into a vectorized form: the Vocabulary, the Vectorizer, and PyTorch's DataLoader. The Vocabulary coordinates the integer-to-token mappings. We use a Vocabulary both for mapping the texttokens to integers and for mapping the class labels to integers. Next, the Vectorizerencapsulates the vocabularies and is responsible for ingesting string data, like areview's text, and converting it to numerical vectors that will be used in the trainingroutine. We use the final assisting class, PyTorch's DataLoader, to group and collatethe individual vectorized data points into minibatches.

1.6.1 The Vocabulary, the Vectorizer, and the DataLoader

The Vocabulary, the **Vectorizer**, and the **DataLoader** are three classes that we use in nearly every example in this book to perform a crucial pipeline: converting text inputs to vectorized minibatches. The pipeline starts with preprocessed text; each data point is a collection of tokens. The three classes presented in the following subsections are responsible for mapping each token to an integer, applying this mapping to each data point to create a vectorized form, and then grouping the vectorized data points into a minibatch for the model.

Vocabulary

The standard methodology is to have a bijection—a mapping that can be reversed—between the tokens and integers.

1.6.2 A Perceptron Classifier

We parameterize the forward() method to allow for the sigmoid function to be optionally applied. To understand why, it is important to first point out that in a binary classification task, binary cross-entropy loss (torch.nn.BCELoss()) is the most appropriate loss function. It is mathematically formulated for binary probabilities. However, there are numerical stability issues with applying a sigmoid and then using this loss function. To provide its users with shortcuts that are more numerically stable, PyTorch provides BCEWithLogitsLoss(). To use this loss function, the output should not have the sigmoid function applied. Therefore, by default, we do not apply the sigmoid. However, in the case that the user of the classifier would like a probability value, the sigmoid is required, and it is left as an option.

1.6.3 The Training Routine

The training loop

The training loop uses the objects from the initial instantiation to update the model parameters so that it improves over time. More specifically, the training loop is composed of two loops: an inner loop over minibatches in the dataset, and an outer loop, which repeats the inner loop a number of times. In the inner loop, losses are computed for each minibatch, and the optimizer is used to update the model parameters.

Chapter 2

Feed-Forward Networks for Natural Language Processing

One of the historic downfalls of the perceptron was that it cannot learn modestly nontrivial patterns present in data.

In this chapter, we explore a family of neural network models traditionally called feed-forward networks. We focus on two kinds of feed-forward neural networks: the multilayer perceptron (MLP) and the convolutional neural network (CNN).

The multilayer perceptron structurally extends the simpler perceptron by grouping many perceptrons in a single layer and stacking multiple layers together. The second kind of feed-forward neural networks studied in this chapter, the convolutional neural network, is deeply inspired by windowed filters in the processing of digital signals.

As we walk through different models, one useful way to make sure you understand how things work is to pay attention to the size and shape of the data tensors as they are being computed. Each type of neural network layer has a specific effect on the size and shape of the data tensor it is computing on, and understanding that effect can be extremely conducive to a deeper understanding of these models.

2.1 The Multilayer Perceptron

The perceptron takes the data vector2 as input and computes a single output value. In an MLP, many perceptrons are grouped so that the output of a single layer is a new vector instead of a single output value. An additional aspect of an MLP is that it combines multiple layers with a nonlinearity in between each layer.

The power of MLPs comes from adding the second Linear layer and allowing the model to learn an intermediate representation that is linearly separable. *Learning intermediate representations that have specific properties, like being linearly separable for a classification task, is one of the most profound consequences of using neural networks and is quintessential to their modeling capabilities.*

2.1.1 A Simple Example: XOR

2.1.2 Implementing MLPs in PyTorch

We can quickly test the “ wiring” of the model by passing some random inputs. Because the model is not yet trained, the outputs are random. Doing this is a useful sanity check before spending time training a model.

However, if you want to turn the prediction vector into probabilities, an extra step is required. Specifically, you require the softmax activation function, which is used to transform a vector of values into probabilities. The softmax function has many roots. In physics, it is known as the Boltzmann or Gibbs distribution; in statistics, it’ s multinomial logistic regression; and in the natural language processing (NLP) community it’ s known as the maximum entropy (MaxEnt) classifier. But you should not use softmax with specific loss functions, because the underlying implementations can leverage superior mathematical/com putational shortcuts.

2.2 Example: Surname Classification with an MLP

The SurnameClassifier Model

The softmax function is optionally applied to make sure the outputs sum to 1; that is, are interpreted as “probabilities”. The reason it is optional has to do with the mathematical formulation of the loss function we use—the cross-entropy loss, introduced in “Loss Functions”. Recall that cross-entropy loss is most desirable for multiclass classification, but computation of the softmax during training is not only wasteful but also not numerically stable in many situations.