The Hundred-Page

Machine Learning

Book

Andriy Burkov

"All models are wrong, but some are useful."
— *George Box*

The book is distributed on the "read first, buy later" principle.

4 Anatomy of a Learning Algorithm

4.1 Building Blocks of a Learning Algorithm

You may have noticed by reading the previous chapter that each learning algorithm we saw consisted of three parts:

1) a loss function;
2) an optimization criterion based on the loss function (a cost function, for example); and
3) an optimization routine that leverages training data to find a solution to the optimization criterion.

These are the building blocks of any learning algorithm. You saw in the previous chapter that some algorithms were designed to explicitly optimize a specific criterion (both linear and logistic regressions, SVM). Some others, including decision tree learning and kNN, optimize the criterion implicitly. Decision tree learning and kNN are among the oldest machine learning algorithms and were invented experimentally based on intuition, without a specific global optimization criterion in mind, and (like it often happens in scientific history) the optimization criteria were developed later to explain why those algorithms work.

By reading modern literature on machine learning, you often encounter references to **gradient descent** or **stochastic gradient descent**. These are two most frequently used optimization algorithms used in cases where the optimization criterion is differentiable.

Gradient descent is an iterative optimization algorithm for finding the minimum of a function. To find a *local* minimum of a function using gradient descent, one starts at some random point and takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

Gradient descent can be used to find optimal parameters for linear and logistic regression, SVM and also neural networks which we consider later. For many models, such as logistic regression or SVM, the optimization criterion is *convex*. Convex functions have only one minimum, which is global. Optimization criteria for neural networks are not convex, but in practice even finding a local minimum suffices.

Let's see how gradient descent works.

4.2 Gradient Descent

In this section, I demonstrate how gradient descent finds the solution to a linear regression problem[1]. I illustrate my description with Python source code as well as with plots that show how the solution improves after some iterations of the gradient descent algorithm.

---

[1]As you know, linear regression has a closed form solution. That means that gradient descent is not needed to solve this specific type of problem. However, for illustration purposes, linear regression is a perfect problem to explain gradient descent.

I use a dataset with only one feature. However, the optimization criterion will have two parameters: w and b. The extension to multi-dimensional training data is straightforward: you have variables $w^{(1)}$, $w^{(2)}$, and b for two-dimensional data, $w^{(1)}$, $w^{(2)}$, $w^{(3)}$, and b for three-dimensional data and so on.
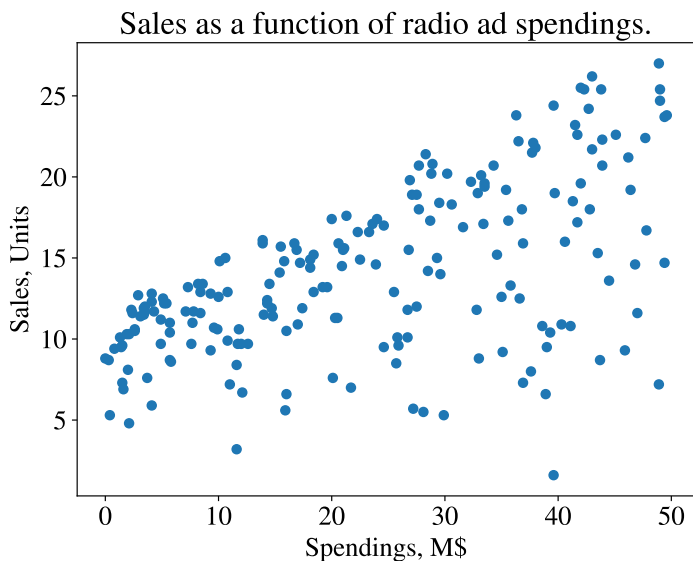

Figure 1: The original data. The Y-axis corresponds to the sales in units (the quantity we want to predict), the X-axis corresponds to our feature: the spendings on radio ads in M$.

To give a practical example, I use the real dataset with the following columns: the Spendings of various companies on radio advertising each year and their annual Sales in terms of units sold. We want to build a regression model that we can use to predict units sold based on how much a company spends on radio advertising. Each row in the dataset represents one specific company:

Company	Spendings, M$	Sales, Units
1	37.8	22.1
2	39.3	10.4
3	45.9	9.3
4	41.3	18.5
..	..	..

We have data for 200 companies, so we have 200 training examples. Fig. 1 shows all examples on a 2D plot.

Remember that the linear regression model looks like this: $f(x) = wx + b$. We don't know

what the optimal values for w and b are and we want to learn them from data. To do that, we look for such values for w and b that minimize the mean squared error:

$$l = \frac{1}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))^2.$$

Gradient descent starts with calculating the partial derivative for every parameter:

$$\frac{\partial l}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} -2x_i(y_i - (wx_i + b));$$
$$\frac{\partial l}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} -2(y_i - (wx_i + b)). \tag{1}$$

To find the partial derivative of the term $(y_i - (wx + b))^2$ with respect to w we applied the *chain rule*. Here, we have the chain $f = f_2(f_1)$ where $f_1 = y_i - (wx + b)$ and $f_2 = f_1^2$. To find a partial derivative of f with respect to w we have to first find the partial derivative of f with respect to f_2 which is equal to $2(y_i - (wx + b))$ (from calculus, we know that the derivative $\frac{\partial f}{\partial x} x^2 = 2x$) and then we have to multiply it by the partial derivative of $y_i - (wx + b)$ with respect to w which is equal to $-x$. So overall $\frac{\partial l}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} -2x_i(y_i - (wx_i + b))$. In a similar way, the partial derivative of l with respect to b, $\frac{\partial l}{\partial b}$, was calculated.

We initialize[2] $w_0 = 0$ and $b_0 = 0$ and then iterate through our training examples, each example having the form of $(x_i, y_i) = (Spendings_i, Sales_i)$. For each training example, we update w and b using our partial derivatives. The learning rate α controls the size of an update:

$$w_i \leftarrow \alpha \frac{-2x_i(y_i - (w_{i-1}x_i + b_{i-1}))}{N};$$
$$b_i \leftarrow \alpha \frac{-2(y_i - (w_{i-1}x_i + b_{i-1}))}{N}, \tag{2}$$

where w_i and b_i denote the values of w and b after using the example (x_i, y_i) for the update.

One pass through all training examples is called an **epoch**. Typically, we need multiple epochs until we start seeing that the values for w and b don't change much; then we stop.

---

[2]In complex models, such as neural networks, which have thousands of parameters, the initialization of parameters may significantly affect the solution found using gradient descent. There are different initialization methods (at random, with all zeroes, with small values around zero, and others) and it is an important choice the data analyst has to make.

It's hard to imagine a machine learning engineer who doesn't use Python. So, if you waited for the right moment to start learning Python, this is that moment. Below we show how to program gradient descent in Python.

The function that updates the parameters w and b during one epoch is shown below:

```python
def update_w_and_b(spendings, sales, w, b, alpha):
    dl_dw = 0.0
    dl_db = 0.0
    N = len(spendings)

    for i in range(N):
        dl_dw += -2*spendings[i]*(sales[i] - (w*spendings[i] + b))
        dl_db += -2*(sales[i] - (w*spendings[i] + b))

    # update w and b
    w = w - (1/float(N))*dl_dw*alpha
    b = b - (1/float(N))*dl_db*alpha

    return w, b
```

The function that loops over multiple epochs is shown below:

```python
def train(spendings, sales, w, b, alpha, epochs):
    for e in range(epochs):
        w, b = update_w_and_b(spendings, sales, w, b, alpha)

        # log the progress
        if e % 400 == 0:
            print("epoch:", e, "loss: ", avg_loss(spendings, sales, w, b))

    return w, b
```

The function *avg_loss* in the above code snippet is a function that computes the mean squared error. It is defined as:

```python
def avg_loss(spendings, sales, w, b):
    N = len(spendings)
    total_error = 0.0
    for i in range(N):
        total_error += (sales[i] - (w*spendings[i] + b))**2
    return total_error / float(N)
```

If we run the *train* function for $\alpha = 0.001$, $w = 0.0$, $b = 0.0$, and 15000 epochs, we will see the following output (shown partially):

```
epoch:   0 loss: 92.32078294903626
epoch:   400 loss: 33.79131790081576
```

```
epoch:   800 loss: 27.9918542960729
epoch:  1200 loss: 24.33481690722147
epoch:  1600 loss: 22.028754937538633
...
epoch:  2800 loss: 19.07940244306619
```
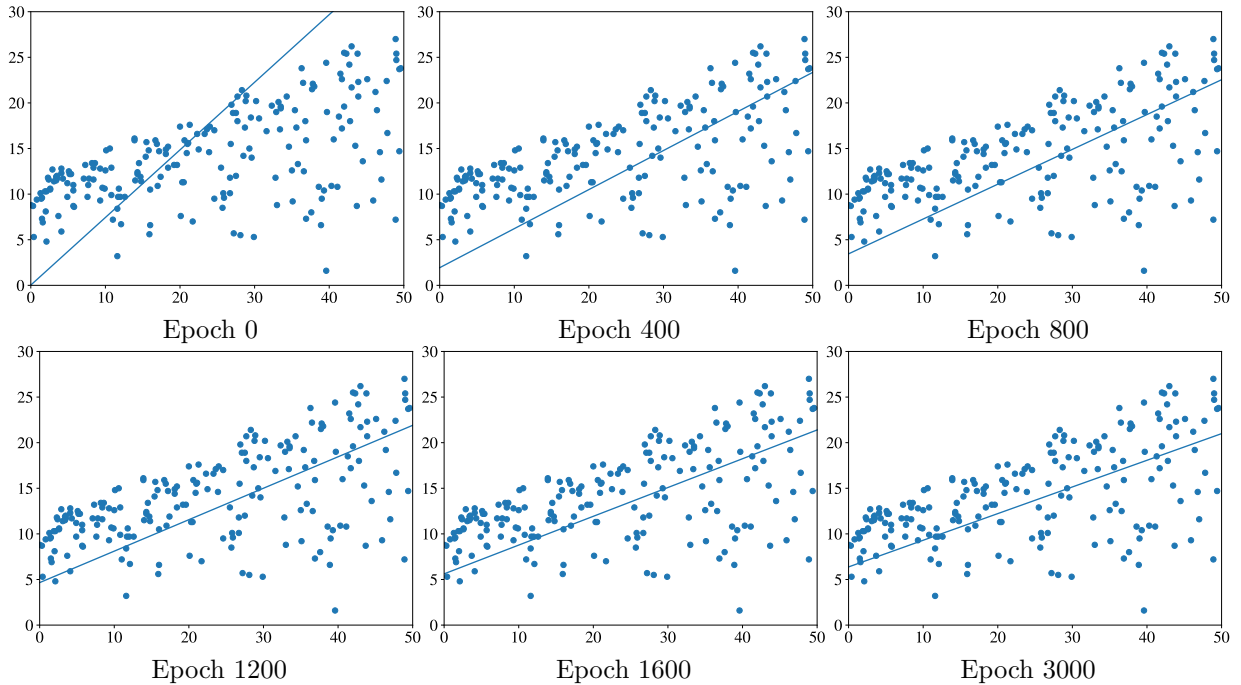

Figure 2: The evolution of the regression line through gradient descent epochs.

You can see that the average loss decreases as the *train* function loops through epochs. Fig. 2 shows the evolution of the regression line through epochs.

Finally, once we have found the optimal values of parameters w and b, the only missing piece is a function that makes predictions:

```python
def predict(x, w, b):
    return w*x + b
```

Try to execute the following code:

```python
w, b = train(x, y, 0.0, 0.0, 0.001, 15000)
x_new = 23.0
y_new = predict(x_new, w, b)
print(y_new)
```

The output is 13.97.

The gradient descent algorithm is sensitive to the choice of the step α. It is also slow for large datasets. Fortunately, several significant improvements to this algorithm have been proposed.

Stochastic gradient descent (SGD) is a version of the algorithm that speeds up the computation by approximating the gradient using smaller batches (subsets) of the training data. SGD itself has various "upgrades". **Adagrad** is a version of SGD that scales α for each parameter according to the history of gradients. As a result, α is reduced for very large gradients and vice-versa. **Momentum** is a method that helps accelerate SGD by orienting the gradient descent in the relevant direction and reducing oscillations. In neural network training, variants of SGD such as **RMSprop** and **Adam**, are most frequently used.

Notice that gradient descent and its variants are not machine learning algorithms. They are solvers of minimization problems in which the function to minimize has a gradient in most points of its domain.

4.3 How Machine Learning Engineers Work

Unless you are a research scientist or work for a huge corporation with a large R&D budget, you usually don't implement machine learning algorithms yourself. You don't implement gradient descent or some other solver either. You use libraries, most of which are open source. A library is a collection of algorithms and supporting tools implemented with stability and efficiency in mind. The most frequently used in practice open-source machine learning library is *scikit-learn*. It's written in Python and C. Here's how you do linear regression in scikit-learn:

```python
def train(x, y):
    from sklearn.linear_model import LinearRegression
    model = LinearRegression().fit(x,y)
    return model


model = train(x,y)


x_new = 23.0
y_new = model.predict(x_new)
print(y_new)
```

The output will, again, be 13.97. Easy, right? You can replace LinearRegression with some other type of regression learning algorithm without modifying anything else. It just works. The same can be said about classification. You can easily replace *LogisticRegression* algorithm with *SVC* algorithm (this is scikit-learn's name for the Support Vector Machine algorithm), *DecisionTreeClassifier*, *NearestNeighbors* or many other classification learning algorithms implemented in scikit-learn.

4.4 Learning Algorithms' Particularities

Here we outline some practical particularities that can differentiate one learning algorithm from another. You already know that different learning algorithms can have different hyperparameters (C in SVM, ϵ and d in ID3). Solvers such as gradient descent can also have hyperparameters, like α for example.

Some algorithms, like decision tree learning, can accept categorical features. For example, if you have a feature "color" that can take values "red", "yellow", or "green", you can keep this feature as is. SVM, logistic and linear regression, as well as kNN (with cosine similarity or Euclidean distance metrics), expect numerical values for all features. All algorithms implemented in scikit-learn expect numerical features. I show in the next chapter how to convert categorical features into numerical ones.

Some algorithms, like SVM, allow the data analyst to provide weightings for each class. These weightings influence how the decision boundary is drawn. If the weight of some class is high, the learning algorithm tries to not make errors in predicting training examples of this class (typically, for the cost of making an error elsewhere). That could be important if instances of some class are in the minority in your training data, but you would like to avoid misclassifying examples of that class as much as possible.

Some classification models, like SVM and kNN, given a feature vector only output the class. Others, like logistic regression or decision trees, can also return the score between 0 and 1 which can be interpreted as either how confident the model is about the prediction or as the probability that the input example belongs to a certain class[3].

Some classification algorithms (like decision tree learning, logistic regression, or SVM) build the model using the whole dataset at once. If you have got additional labeled examples, you have to rebuild the model from scratch. Other algorithms (such as Naïve Bayes, multilayer perceptron, SGDClassifier/SGDRegressor, PassiveAggressiveClassifier/PassiveAggressiveRegressor in scikit-learn) can be trained iteratively, one batch at a time. Once new training examples are available, you can update the model using only the new data.

Finally, some algorithms, like decision tree learning, SVM, and kNN can be used for both classification and regression, while others can only solve one type of problem: either classification or regression, but not both.

Usually, each library provides the documentation that explains what kind of problem each algorithm solves, what input values are allowed and what kind of output the model produces. The documentation also provides information on hyperparameters.

---

[3]If it's really necessary, the score for SVM and kNN predictions could be synthetically created using some simple techniques.