Photo by Lindsay Henwood on Unsplash

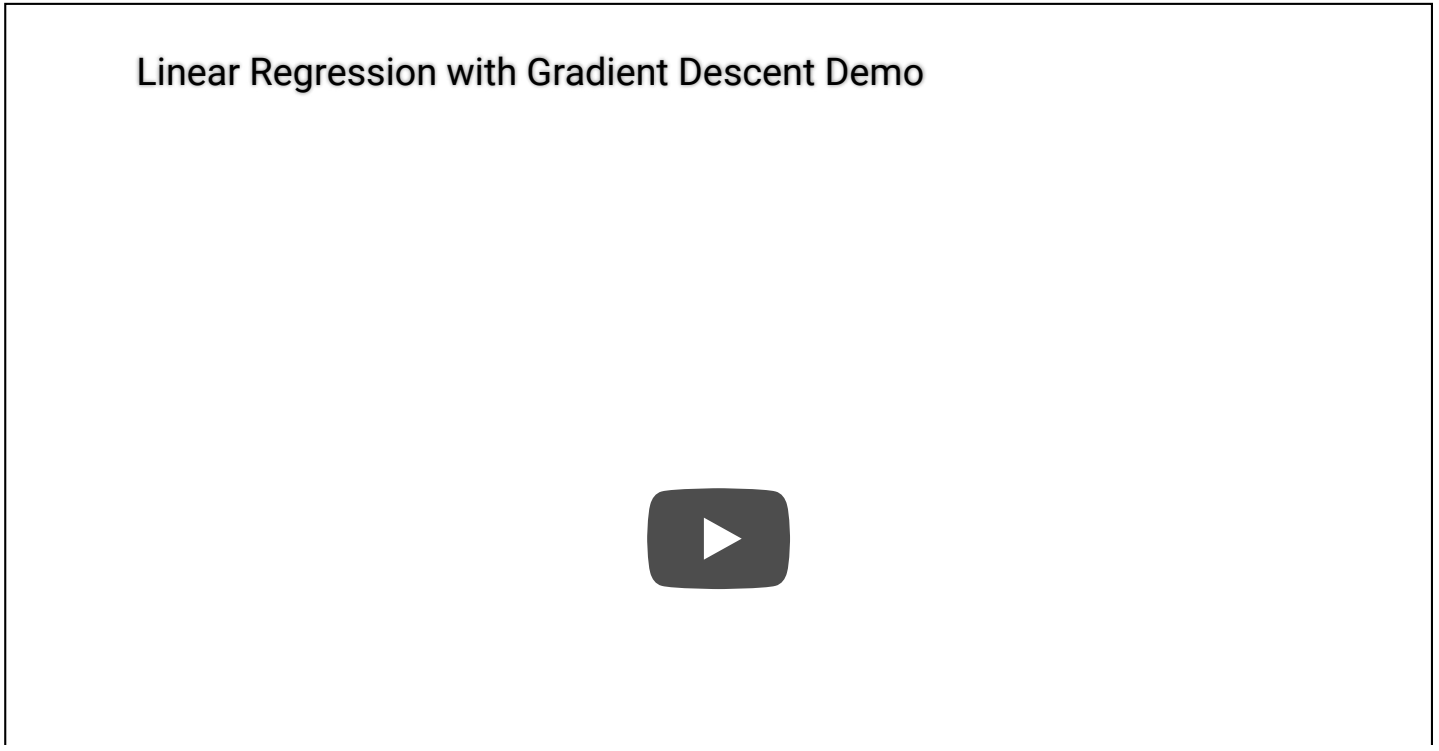
Tutorial: Linear Regression with Stochastic Gradient Descent

Implementing backpropagation in JavaScript

Raimi Karim Dec 11, 2018 · 10 min read

You can find the backpropagation demo here.

This article should provide you a good start for us to dive deep into deep learning. Let me walk you through the step-by-step calculations for a linear regression task using stochastic gradient descent.



A short YouTube clip for the backpropagation demo found here

Contents

1 Preparation

1.1 Data

We have some data: as we observe the independent variables x_1 and x_2, we observe the dependent variable (or response variable) y along with it.

In our dataset, we have 6 examples (or observations).

	x1	x2	y
1)	4	1	2
2)	2	8	-14
3)	1	0	1
4)	3	2	-1
5)	1	4	-7
6)	6	7	-8

1.2 Model

The next question to ask: "How are both x_1 and x_2 related to y?"

We believe that they are connected to each other by this equation:

$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

Our job today is to find the 'best' w and b values.

> I have used the deep learning conventions $\boldsymbol{w}$ and $\boldsymbol{b}$, which stand for **weights** and **biases** respectively. But note that linear regression is not deep learning.

1.3 Define loss function

Let's say at the end of this exercise, we've figured out our model to be

$$\hat{y} = 0.43x_1 - 0.21x_2 + 0.77$$

How do we know if our model is doing well?

We simply compare the predicted $\hat{y}$ and the observed y through a *loss function*. There are many ways to define the loss function but in this post, we define it as the squared difference between $\hat{y}$ and y.

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

Generally, the smaller the L, the better.

1.4 Minimise loss function

Because we want the difference between $\hat{y}$ and y to be small, we want to make an effort to minimise it. This is done through **stochastic gradient descent** optimisation. It is basically iteratively updating the values of w_1 and w_2 using the value of gradient, as in this equation:

$$w_{\text{new}} = w_{\text{current}} - \eta \frac{\partial L}{\partial w_{\text{current}}}$$

Fig. 2.0: Computation graph for linear regression model with stochastic gradient descent.

This algorithm tries to find the right weights by constantly updating them, bearing in mind that we are seeking values that minimise the loss function.

> **Intuition: stochastic gradient descent**
>
> You are $\boldsymbol{w}$ and you are on a graph (loss function). Your current value is $\boldsymbol{w}=5$. You want to move to the lowest point in this graph (minimising the loss function).
>
> You also know that, with your current value, your gradient is 2. You somehow must make use of this value to move on with life.
>
> From high school math, 2 means you're on an inclined slope and the only way you can descend is to move left, at this point.
>
> If taking $5+2$ means you're going to the right climbing up the slope, then the only way is to take 5–2 which brings you to the left, descending down. So gradient descent is all about subtracting the value of the gradient from its current value.

2. Implementation

The workflow for *training* our model is simple: forward propagation (or feed-forward or forward pass) and backpropagation.

> **Definition: training**
> Training just means regularly updating the values of your weights, put simply.

Below is the workflow. Click to jump to the section.

Let's get started.

To keep track of all the values, we first build a 'computation graph' that comprises nodes colour-coded in

1. **orange** — the placeholders (x_1, x_2 and y),
2. **dark green** — the weights and bias (w_1, w_2 and b),
3. **light green** — the model ($\hat{y}$) connecting w_1, w_2, b, x_1 and x_2, and

4. **yellow** — the loss function (L) connecting the $\hat{y}$ and y.

Fig. 2.0: Computation graph for linear regression model with stochastic gradient descent.

For forward propagation, you should read this graph from top to bottom and for backpropagation bottom to top.

> **Note**
> I have adopted the term 'placeholder', a nomenclature used in TensorFlow to refer to these 'data variables'.
> I will also use the term 'weights' to refer to w and b collectively.

2.1 Forward Propagation

2.1.1 Initialise weights (one-time)

Since gradient descent is all about updating the weights, we need them to start with some values, known as *initialising* weights.

Here we initialised the weights and bias as follows:

These are reflected in the **dark green nodes** in Fig. 2.1.1 below:

Fig. 2.1.1: Weights initialised (dark green nodes)

There are many ways to initialise weights (zeros, ones, uniform distribution, normal distribution, truncated normal distribution, etc.) but we won't cover them in this post. In this tutorial, we initialised the weights by using truncated normal distribution and the bias with 0.

2.1.2 Feed data

Next, we set the batch size to be 1 and we feed in this first batch of data.

> **Batch and batch size**
>
> We can divide our dataset into smaller groups of equal size. Each group is called a **batch** and consists of a specified number of examples, called **batch size**. If we multiply these two numbers, we should get back the number of observations in our data.
>
> Here, our dataset consists of 6 examples and since we defined the batch size to be 1 in this training, we have 6 batches altogether.

Current batch of data used to feed in the model is bolded below:

	x1	x2	y
1)	4	1	2
2)	2	8	-14
3)	1	0	1
4)	3	2	-1
5)	1	4	-7
6)	6	7	-8

Eqn. 2.1.2: First batch of data fed into model

In Fig. 2.1.2, the **orange nodes** are where we feed in the current batch of data.

Fig. 2.1.2: Feeding data to model with first batch (orange nodes)

2.1.3 Compute ŷ

Now that we have the values of x_1, x_2, w_1, w_2 and b ready, let's compute $\hat{y}$.

Eqn. 2.1.3: Compute ŷ

The value of $\hat{y}$ (=-0.1) is reflected in the **light green node** below:

Fig. 2.1.3: ŷ computed (light green node)

2.1.4 Compute loss

How far is our predicted $\hat{y}$ from the given y data? We compare them by calculating the loss function L as defined earlier.

Eqn. 2.1.4: Compute the loss

You can see this value in the **yellow node** in the computation graph.

Fig. 2.1.4A: L computed (yellow node)

It is a common practice to log the loss during training, together with other information like the epoch, batch and time taken. In my demo, you can see this under the **Training progress** panel.

Fig. 2.1.4B: Logging loss and other information

2.2 Backpropagation

2.2.1 Compute partial differentials

Before we start adjusting the values of the weights and bias $w1$, $w2$ and b, let's first compute all the partial differentials. These are needed later when we do the weight update.

Fig. 2.2.1: Indicated partial differentials to the relevant edges on the graph

Namely, we compute **all possible paths** leading to every w and b only, because these are the only variables which we are interested in updating. From Fig. 2.2.1 above, we see that there are 4 edges that we labeled with the partial differentials.

Recall the equations for the model and loss function:

Model

Loss function

The partial differentials are as follows:

L (**yellow**) — $\hat{y}$ (**light green**)**:**

Eqn. 2.2.1A: Partial differential of L w.r.t. ŷ

$\hat{y}$ (**light green**) — b (**dark green**):

Eqn. 2.2.1B: Partial differential of ŷ w.r.t. b

$\hat{y}$ (**light green**) — w_1 (**dark green**):

Eqn. 2.2.1C: Partial differential of ŷ w.r.t. w1

$\hat{y}$ (**light green**) — w_2 (**dark green**):

Eqn. 2.2.1D: Partial differential of ŷ w.r.t. w2

Note that the values of the partial differentials follow the **values from the current batch**. For example, in Eqn. 2.2.1C, $x_1 = 4$.

2.2.2 Update weights

Observe the **dark green nodes** in Fig. 2.2.2 below. We see three things:
i) b changes from 0.000 to 0.212
ii) w_1 changes from -0.017 to 0.829
iii) w_2 changes from -0.048 to 0.164

Fig. 2.2.2: Updating the weights and bias (dark green nodes)

Also pay attention to the 'direction' of the pathway from the **yellow node** to the **green node**. They go from bottom to top.

This is stochastic gradient descent — updating the weights using backpropagation, making use of the respective gradient values.

Let's first focus on updating b. The formula for updating b is

Eqn. 2.2.2A: Stochastic gradient descent update for b

where

- b — current value
- b' — value after update
- η —learning rate, set to 0.05
- $\partial L/\partial b$ — gradient i.e. partial differential of L w.r.t. b

To get the gradient, we need to multiply the **paths** from L leading to b using chain rule:

Eqn. 2.2.2B: Chain rule for partial differential of L w.r.t. b

We would require the current batch values of x, y, $\hat{y}$ and the partial differentials so let's just place them below for easy reference:

Eqn. 2.2.2C: Partial differentials

Eqn. 2.2.2D: Values from current batch and the predicted $\hat{y}$

Using the stochastic gradient descent equation in Eqn. 2.2.2A and plugging in all the values from Eqn. 2.2.2B-D gives us:

That's it for updating b! Phew! We are left with updating w_1 and w_2, which we update in a similar fashion.

End of batch iteration

Congrats! That's it for dealing with the first batch!

	x1	x2	y	
1)	4	1	2	✓
2)	2	8	-14	
3)	1	0	1	
4)	3	2	-1	
5)	1	4	-7	
6)	6	7	-8	

Now we need to iterate the above-mentioned steps to the other 5 batches, namely examples 2 to 6.

Iterating through batch 1 to 6 (apologies for the poor GIF quality!)

End of epoch

We complete 1 epoch when the model has iterated through all the batches once. In practice, we extend the epoch to more than 1.

> One **epoch** is when our setup has seen **all** the observations in our dataset once. But one epoch is almost always never enough for the loss to converge. In practice, this number is manually tuned.

At the end of it all, you should get a final model, ready for inference, say:

Let's have a review of the entire workflow in a pseudo-code:

```
initialise_weights()

for i in epochs:

    for j in batches:

        #forward propagation
        feed_batch_data()
```

```
compute_ŷ()
compute_loss()

#backpropagation
compute_partial_differentials()
update_weights()
```

Improve training

One epoch is never enough for a stochastic gradient descent optimisation problems. Remember that in Fig. 4.1, our loss is at 4.48. If we increase the number of epochs, which means just increasing the number of times we update the weights and biases, we can converge it to a satisfactory low.

Below are the things you can improve the training:

- Extend training to more than 1 epoch
- Increase batch size
- Change optimiser (see my post on gradient descent optimisation algorithms here)
- Adjust learning rate (changing the learning rate value or using learning rate schedulers)
- Hold out a train-val-test set

About

I built an interactive explorable demo on linear regression with gradient descent in JavaScript. Here are the libraries I used:

- Dagre-D3 (GraphViz + d3.js) for rendering the graphs
- MathJax for rendering mathematical notations
- ApexCharts for plotting line charts
- jQuery

Check out the interactive demo here.

You might also like to check out *A Line-by-Line Layman's Guide to Linear Regression using TensorFlow* below, which focuses on coding linear regression using the TensorFlow library.

A line-by-line layman's guide to Linear Regression using TensorFlow

Linear regression is a great start to the journey of machine learning, given that it is a pretty straightforward...

medium.com

References

Calculus on Computational Graphs: Backpropagation -- colah's blog

Backpropagation is the key algorithm that makes training deep models computationally tractable. For modern neural...

colah.github.io

Related Articles on Deep Learning

Animated RNN, LSTM and GRU

Line-by-Line Word2Vec Implementation (on word embeddings)

10 Gradient Descent Optimisation Algorithms + Cheat Sheet

Counting No. of Parameters in Deep Learning Models

Attn: Illustrated Attention

Illustrated: Self-Attention

Thanks to Ren Jie and Derek for ideas, suggestions and corrections to this article.

Follow me on Twitter @remykarem or LinkedIn. You may also reach out to me via raimi.bkarim@gmail.com. Feel free to visit my website at remykarem.github.io.

Thanks to Ren Jie Tan and Derek Chia.