Linear Regression the Autograd Way

This chapter isn't really entirely about linear regression. It's more about how we use automatic differentiation to build models from data. But along the way, we'll also think about linear regression, likelihood, and objective functions. We'll start with univariate linear regression, extend it to multivariate, try everything out on more serious data, and think about whether the discovered models can help us interpret the data.

Linear regression originates from the method of least squares, first published in 1805 by Adrien-Marie Legendre, and later, in 1809, further connected to the normal distribution by Carl Friedrich Gauss in the analysis of astronomical observations. The term *regression* was introduced later, in the 1880s, by Francis Galton while studying heredity. He observed that very tall parents often have tall children, but on average these children tend to be somewhat closer to the population mean; he called this phenomenon *regression toward the mean*.

Data and What We Actually Want

In *univariate regression*, we deal with data that contains one independent variable x and one dependent variable y. The goal is to describe or approximate the functional relationship between them with a linear model of the form

$$\hat{y}(x) = wx + b,$$

where w is the slope (weight) that determines the inclination of the line, and b is the intercept, i.e., the intersection with the vertical axis. So the model is completely determined by two parameters, w and b.

When training the model, we have a training dataset available, consisting of pairs of values (x_i, y_i), for example:

x_i	y_i
1.4	−2.0
−4.7	−18.1
−2.2	−1.9
−2.8	−4.4
2.4	6.5

For each training example, the model predicts a value $\hat{y}_i = wx_i + b$, which can differ from the actual value y_i. We can write the prediction error for an individual example as

$$\varepsilon_i = \hat{y}_i - y_i.$$

Since we don't care about the sign of the error , we square the

We introduced the idea of the sum of squared errors here somewhat informally. We'll get back to why this makes sense a bit later.

errors and compute their average. This gives us the cost function, or loss function, which for a given training set depends only on the model parameters:

$$J(w, b) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(wx_i + b - y_i)^2.$$

The cost function measures how well the chosen parameters w and b describe the given data. The goal of learning is to find such parameter values that minimize the cost function:

$$(w^*, b^*) = \arg\min_{w,b} J(w, b).$$

The pair (w^*, b^*) represents the optimal model parameters for the given training set and is obtained through model training, i.e., machine learning from data. Machine learning is then the process where, for a given training set and a chosen model structure, we find such model parameters that optimize the chosen objective function.

Univariate Linear Regression in Python

Let's implement linear regression in a Python class `LinReg`. In the implementation, we'll use the `Value` class as developed in the previous chapter and stored from here on in the `agrad` library:

```
from agrad import Value

class LinReg:
    def __init__(self):
        self.w = Value(random.uniform(-1,1), label='w')
        self.b = Value(0.0, label='b')

    def __call__(self, x):
        return self.w * x + self.b

    def parameters(self):
        return [self.w, self.b]

    def loss(self, xs, ys):
        yhats = [self(x) for x in xs]
        return sum([(y - yhat)**2  for y, yhat in zip(ys, yhats)])

    def __repr__(self):
        return f"LinReg(w={self.w.data:.3f}, b={self.b.data:.3f})"
```

In the function `__init__()`, we define the initial values of the model parameters. The model call is implemented in the function `__call__()`. The class `LinReg` also returns a list of model parameters,

which we'll need when updating the parameters during gradient descent. Extremely important, however, is the `loss()` function, which computes the loss for given parameter values and a given training set.

For a first test of how `LinReg` works, we can try using our class to compute linear regression with specific model parameters:

```
>>> model = LinReg()
>>> model.w = Value(10); model.b = Value(3)
>>> model(10)
Value(: 103)
```

On a small training set, for example:

```
import random
random.seed(42)
n = 5
xs = [random.uniform(-5, 5) for _ in range(n)]
ys = [2*x -1 + random.gauss(0, 4) for x in xs]
```

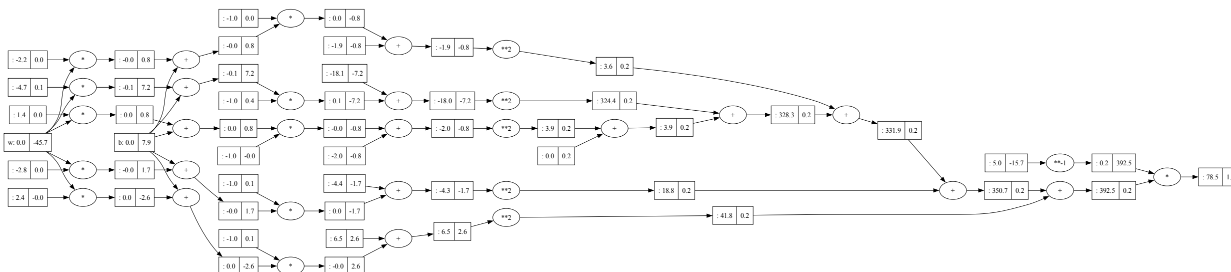
we can now check

```
>>> lr = LinReg()
>>> lr
LinReg(w=0.011, b=0.000)
>>> lv = lr.loss(xs, ys)
>>> lv
Value(: 78.49574710788079)
```

The loss (sum of squared errors) is of course large, since the above model is still untrained and random. What's interesting, though, is the computational graph for the loss. The code for drawing it is:

```
from agrad import draw_dot
lv.backward()
dot = draw_dot(lv)
dot.render('a', format='pdf', cleanup=True)
```

An attentive reader will of course notice that before drawing the computational graph, we computed the gradients. We hid the implementation of the drawing in the `agrad` library, but it's exactly the same as in the previous chapter, so we don't need to show it again here. The computational graph for our loss function is already quite complex here, even with a small training set, partly because it includes a sum over all five training examples:



Training

Ah, finally! Everything is ready to compute the *right* parameter values of our univariate model from the data. Recall: using the computational graph, we'll compute partial derivatives of the function with respect to the model parameters, adjust them a bit each time, and repeat the process until convergence or for some predefined number of iterations.

Let's now put together a training function that implements gradient descent. The optimization function below is actually quite general—we could use it for any model that, during training, uses labeled examples.

```python
def train(model, xs, ys, learning_rate=0.001, n_epochs=1000):
    for k in range(n_epochs):
        # compute loss
        loss = model.loss(xs, ys)

        # compute gradients
        for p in model.parameters():
            p.grad = 0
        loss.backward()

        # update
        for p in model.parameters():
            p.data -= learning_rate * p.grad

        if k % 50 == 0:
            print(f"{k:3} Loss: {loss.data:5.3f} {model}")
    return model
```

In each iteration, we built the computational graph for our loss, set the gradients of the model parameters to zero (since we accumulate gradients into these values), then computed the gradients and updated the parameter values.

Time to train:

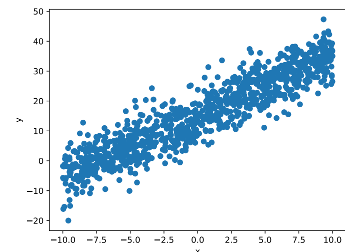
```
>>> model = train(LinReg(), xs, ys, learning_rate=0.01, n_epochs=500)
  0 Loss: 130.147 LinReg(w=-0.326, b=-0.102)
 50 Loss: 16.793 LinReg(w=2.578, b=-0.745)
100 Loss: 16.778 LinReg(w=2.566, b=-0.827)
150 Loss: 16.775 LinReg(w=2.560, b=-0.864)
200 Loss: 16.774 LinReg(w=2.558, b=-0.880)
250 Loss: 16.774 LinReg(w=2.557, b=-0.887)
300 Loss: 16.774 LinReg(w=2.556, b=-0.890)
350 Loss: 16.774 LinReg(w=2.556, b=-0.891)
400 Loss: 16.774 LinReg(w=2.556, b=-0.892)
450 Loss: 16.774 LinReg(w=2.556, b=-0.892)
```

So in a few hundred iterations, gradient descent found the (almost) correct model, i.e., the one with parameters $w = 2$ and $b = -1$. We leave it to the reader to experiment with different learning rates. Let's just say that for this data and this model, training can converge faster, but if we increase the learning rate too much, we may run into errors like `'OverflowError: (34, 'Result too large')`. Why?

Batch Learning

A reader of these notes might point out that the example in the previous section with only five samples was too simple, and that such linear regression could easily be done "by hand". Maybe—but even for such a small example, the amount of computation would already be quite large. So now is the right moment to test our implementation on a larger dataset:

```
random.seed(42)
n = 1000
xs = [random.uniform(-10, 10) for _ in range(n)]
ys = [2*x + 1 + random.gauss(0, 5) for x in xs]
```



With a large number of examples, the computational graph for the loss function grows significantly, and training can therefore become quite slow. There may be enough data to try a different approach, where we compute the loss only on a sample of the training set. This is called batch learning (or *batch learning*), and we can implement it simply with the function below, which we add to the `LinReg` class:

```
    def batch_loss(self, xs, ys, m=10):
        indices = random.sample(range(len(xs)), m)
        batch_xs = [xs[idx] for idx in indices]
        batch_ys = [ys[idx] for idx in indices]
        return self.loss(batch_xs, batch_ys)
```

For batch learning, we now only need to change the line that defines the loss function. It now becomes

```
loss = model.batch_loss(xs, ys, batch_size)
```

and assumes that we add another hyperparameter `batch_size` to the training method, with a default value, say 10.

With batch learning, the loss over the full training set will not decrease monotonically:

```
>>> model = train(LinReg(), xs, ys, learning_rate=0.01, n_epochs=500)
  0 Loss: 442.514 LinReg(w=0.620, b=0.266)
 50 Loss: 111.536 LinReg(w=1.792, b=9.632)
100 Loss: 12.292 LinReg(w=1.868, b=12.622)
150 Loss: 24.878 LinReg(w=2.042, b=13.802)
200 Loss: 46.238 LinReg(w=1.705, b=14.689)
250 Loss: 30.237 LinReg(w=1.685, b=14.732)
300 Loss: 23.126 LinReg(w=2.113, b=14.865)
350 Loss: 29.744 LinReg(w=1.662, b=14.941)
400 Loss: 25.119 LinReg(w=2.435, b=14.894)
450 Loss: 10.819 LinReg(w=1.884, b=14.988)
```

Our model isn't exactly perfect either, but the model parameters have gotten quite close to the ones we used to generate the data. Batch learning is much faster than gradient descent where we compute the loss over the entire training set each time, but we need to be careful when choosing the learning rate and the batch size.

All of a sudden, we've accumulated quite a few hyperparameters in our training procedure. These are parameters that influence the training process, including the learning rate, the number of epochs, and the batch size. We leave it to the reader to think about how to set these parameters—we'll deal with them more in the following chapters.

Multivariate Linear Regression

Let's extend our linear regression model to the multivariate case, i.e., to models that take multiple input, independent variables, which in machine learning we also call features.

```
random.seed(42)
n = 1000
X = [[random.uniform(-10, 10) for _ in range(3)] for _ in range(n)]
ys = [2*x[0] + 3*x[1] - x[2] + 1 + random.gauss(0, 5) for x in X]
```

There shouldn't be any major changes in the training code. In the `LinReg` class, the functions `loss()` and `batch_loss()` stay the same as before, while we slightly modify the others,

Our implementation could also be improved by storing the names of the variables. These become important whenever we try to interpret the model.

```
class LinReg:
    def __init__(self, n_inputs):
        self.weights = [Value(random.uniform(-1,1), label=f'w{i}')
                        for i in range(n_inputs)]
        self.b = Value(0.0, label='b')
```

```
    def __call__(self, x):
        # x is a list of input values
        return sum(w * xi for w, xi in zip(self.weights, x)) + self.b

    def parameters(self):
        return self.weights + [self.b]

    def __repr__(self):
        weights_str = ', '.join(f'w{i}={w.data:.3f}'
                                for i, w in enumerate(self.weights))
        return f"LinReg({weights_str}, b={self.b.data:.3f})"
```

Training, where the `train` function remains exactly as we developed it in the previous sections, successfully converges to the correct solution:

```
>>> lr = LinReg(n_inputs=3)
>>> model = train(lr, X, ys, n_epochs=10000, batch_size=50, learning_rate=0.01)
   0 Loss: 863.724 LinReg(w0=1.623, w1=2.440, w2=-0.938, b=-0.068)
 500 Loss: 26.504 LinReg(w0=2.054, w1=2.709, w2=-0.993, b=0.805)
1000 Loss: 32.210 LinReg(w0=2.174, w1=2.939, w2=-0.932, b=0.787)
...
4500 Loss: 18.590 LinReg(w0=1.745, w1=2.719, w2=-0.993, b=0.832)
5000 Loss: 32.224 LinReg(w0=2.269, w1=2.903, w2=-1.074, b=0.841)
```

It's actually quite surprising how easily we extended our implementation to learning from data with an arbitrary number of features. Still, just to remind you, we're using the automatic differentiation library we developed in the previous chapter, which is perfectly capable even for this last, not exactly small example.

Likelihood

Now it's time to take a closer look at why we defined the objective function as minimizing the sum of squared errors. Although this choice is intuitive, it also has a clear statistical explanation. In multivariate linear regression, we assume that observations are generated according to the linear model

$$y_i = \mathbf{w}^\top \mathbf{x}_i + b + \varepsilon_i,$$

where $\mathbf{x}_i = (x_{i1}, \dots, x_{id})$ is the vector of input features, $\mathbf{w} = (w_1, \dots, w_d)$ is the vector of weights, b is the intercept, and ε_i is the random error for the i-th example. We further assume that the errors ε_i are independent and identically distributed.

This assumption is often written as i.i.d., meaning the errors are *independent and identically distributed*.

Additionally, we assume that the errors are normally distributed with mean zero:

$$\varepsilon_i \sim \mathcal{N}(0, \sigma^2).$$

The assumption of a normal distribution is often reasonable. According to the central limit theorem, the sum of a large number of small, independent effects tends toward a normal distribution, so the normal distribution is often a good model for measurement noise.

Since

$$y_i = \mathbf{w}^\top \mathbf{x}_i + b + \varepsilon_i,$$

it follows that y_i is also normally distributed around the value predicted by the model:

$$y_i \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x}_i + b, \sigma^2).$$

The probability density of observing y_i is therefore

$$p(y_i \mid \mathbf{x}_i, \mathbf{w}, b) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(y_i - (\mathbf{w}^\top \mathbf{x}_i + b)\right)^2}{2\sigma^2}\right).$$

Now let's define the likelihood as a function of the model parameters, which tells us *how likely the observed data would be if they were actually generated by the model with given parameter values*. Since we assumed that the examples are independent, the likelihood for the entire training set is the product of the probabilities of all examples:

The concept of likelihood emerged in the early 20th century in statistics, when Ronald A. Fisher developed the method of maximum likelihood as a general approach for estimating parameters of statistical models. The idea is simple: we choose the model parameters for which the observed data are most likely. In machine learning, this concept became central in modern probabilistic machine learning, as it enables a systematic derivation of training criteria. It is important because it provides a statistical justification for learning models: instead of ad-hoc criteria, we optimize a function that directly measures how well the model explains the observed data.

$$L(\mathbf{w}, b) = \prod_{i=1}^{n} p(y_i \mid \mathbf{x}_i, \mathbf{w}, b).$$

We choose the model parameters so that the likelihood is as large as possible. This approach is called maximum likelihood (or *maximum likelihood estimation*). Since the product of many terms can be inconvenient to compute, we usually maximize the log-likelihood:

$$\log L(\mathbf{w}, b) = \sum_{i=1}^{n} \log p(y_i \mid \mathbf{x}_i, \mathbf{w}, b).$$

If we plug in the expression for the normal distribution, we get

$$\log L(\mathbf{w}, b) = -\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y_i - (\mathbf{w}^\top \mathbf{x}_i + b)\right)^2.$$

The first term is a constant that does not depend on the parameters $\mathbf{w}$ and b. The factor $1/(2\sigma^2)$ also does not affect the maximum. So maximizing the log-likelihood is equivalent to minimizing the expression

$$\sum_{i=1}^{n} \left(y_i - (\mathbf{w}^\top \mathbf{x}_i + b)\right)^2.$$

And that is exactly the sum of squared errors that we used as the objective function when training linear regression.

So it turns out that minimizing the squared error is not just an intuitive choice, but follows directly from the assumption that the

measurement errors in the data are normally distributed. In this framework, we estimate the parameters of the linear model using maximum likelihood.

Interpretation

The parameters of linear regression models are actually feature weights, and linear regression is a weighted sum of the inputs. The weights are related to the role of each feature, i.e., its influence on the output value of the linear function. Smaller weights correspond to less important features, while larger weights correspond to more important ones. The sign of the weight is also important, as it tells us whether the feature is negatively or positively related to the output. There is still room here for further thinking about interpretation, maybe even some interesting visualizations, but let's take it step by step and focus here only on the weights.

We would also like, for example, to determine the smallest set of features that still gives us a good model and ignore the rest. But that will be the topic of the next chapter.

Let's start with a practical example. We'll use a dataset with body measurements of men. For each example, we have the body fat percentage, which was computed using the Brozek formula (not important for us, but that's where the name of the target variable comes from, *Body Fat Brozek*) based on body density measured using underwater weighing. In addition, the data contains age, weight, height, and a number of easily measured body circumferences such as neck, chest, abdomen, and so on. Our goal will be to build a model that predicts body fat percentage from these simple measurements, and at the same time estimate which of the measurements plays the most important role. This is a typical regression problem: the examples are described by multiple input features and one continuous output variable.

The data comes from the *Body Fat Prediction Dataset*, which is publicly available on Kaggle and dates back to 1985. It contains 252 samples, 14 features, and a target variable representing body fat percentage.

Let's load the data and print the first five examples for a few selected features:

```
df = pd.read_csv("body-fat-brozek.csv")
feature_names = df.columns[1:].tolist()
ys = df.iloc[:, 0].tolist()
X = df.iloc[:, 1:].values.tolist()

preview = (
    df[["body fat brozek", "age", "weight", "ankle"]]
    .head()
)
print(preview.to_string(index=False))
```

Below is a table with the first five examples and selected features. The first column is the target, and the other three are some of the features. You can see that the feature scales are different—weight

values (clearly not in kg) are much larger than the values used for ankle circumference.

body fat brozek	age	weight	ankle
12.6	23	154.25	21.9
6.9	22	173.25	23.4
24.6	22	154.00	24.0
10.9	26	184.75	22.8
27.8	24	184.25	24.0

Table 1: First five examples from the dataset.

If we left the data in this form, the weights would depend on the scale of the feature values, and even if the weight for weight were smaller than that for ankle circumference, the two features could still be equally important. To remove the influence of feature scale, we normalize the data (features):

```
X_arr = np.array(X)
mean = X_arr.mean(axis=0)
std = X_arr.std(axis=0)
X_norm = ((X_arr - mean) / std).tolist()
```

Now everything is ready to build the model and analyze the resulting weights:

```
lr_norm = LinReg(n_inputs=len(feature_names))
model_norm = train(lr_norm, X_norm, ys, n_epochs=1000, batch_size=20, learning_rate=0.05)

def print_weights(model, feature_names):
    pairs = [(name, w.data) for name, w in zip(feature_names, model.weights)]
    pairs.sort(key=lambda p: abs(p[1]), reverse=True)
    for name, w in pairs:
        print(f"{w:6.3f} {name}")
```

The result,

```
>>> print_weights(model_norm, feature_names)
9.433 abdomen
-2.489 weight
-1.519 hip
-1.498 wrist
 1.367 tight
 1.330 forearm
-1.121 neck
 0.767 age
 0.347 ankle
 0.284 adiposity
 0.229 biceps
-0.078 chest
-0.041 height
```

```
-0.024 knee
```

is actually in line with expectations: to estimate body fat percentage, it's best to know the abdomen circumference. Next come weight, hip circumference, and wrist circumference, which have negative weights. A negative sign does not necessarily mean that higher weight or larger hips reduce body fat; it simply means that, when taking all other features into account, the model uses these variables as corrections to the prediction. Such situations are common when features are strongly correlated with each other.

We leave it to the reader to try what happens if the data is not normalized. Just a hint: the results would be very different.