Gradient descent, learning rates, and stochastic gradient descent

Feb. 14, 2020

Example: Birthweight and bronchopulmonary dysplasia

Can we estimate probability of bronchopulmonary dysplasia (BPD, a lung disease that affects newborns) as a function of the baby's birth weight?

Data from Pagano, M. and Gauvreau, K. (1993). *Principles of Biostatistics*. Duxbury Press.

$$Y_i = \begin{cases} 1 & \text{if baby number } i \text{ has BPD} \\ 0 & \text{otherwise} \end{cases}$$

$$X_i = \text{ birth weight for baby number } i$$

Import libraries

```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.special import expit as sigmoid
```

Read in data

```
bpd_data = pd.read_table("http://www.stat.cmu.edu/~larry/all-of-nonpar/=data/bpd.dat", delim_whitespace=True)
```

```
bpd_data.head()
```

```
##    birthweight  BPD
## 0          850    1
## 1         1500    0
## 2         1360    1
## 3          960    0
## 4         1560    0
```

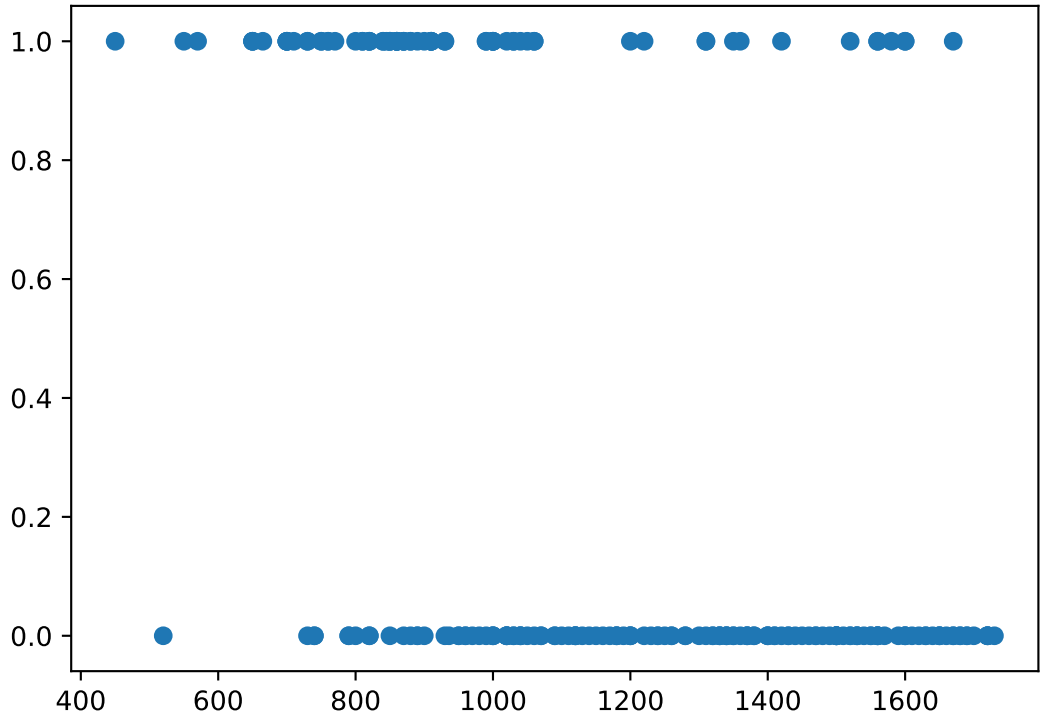
```
x = bpd_data[['birthweight']].to_numpy().T
y = bpd_data[['BPD']].to_numpy().T
print(x.shape)
```

```
## (1, 223)
```

```
print(y.shape)
```

```
## (1, 223)
```

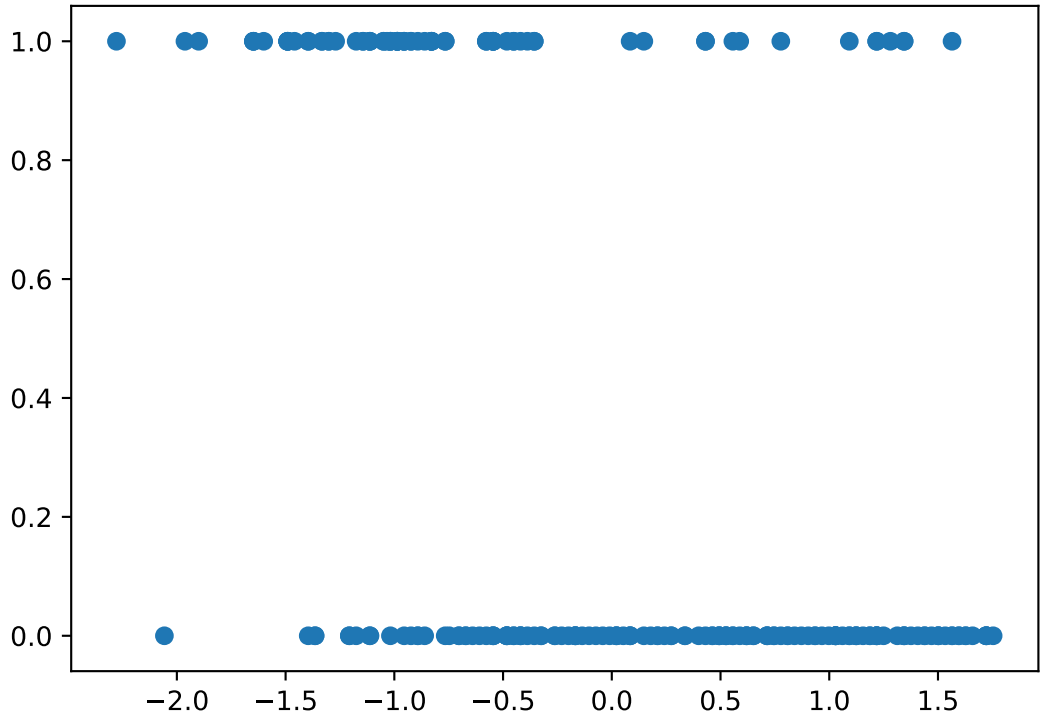
```
plt.scatter(x[0, :], y[0, :])
```



We need to center and scale the x variables (we will see why later in this handout).

```
x_standardized = (x - np.mean(x)) / np.std(x)
```

```
plt.scatter(x_standardized[0, :], y[0, :])
```



Our usual function for estimation by gradient descent

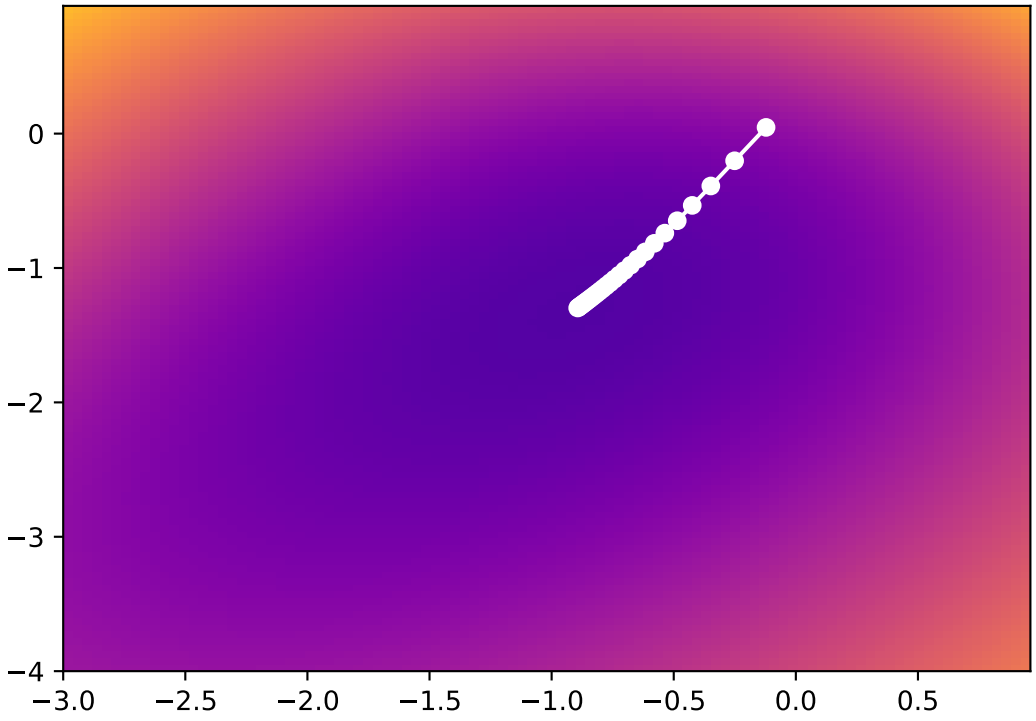
Only change is that I'm saving the history of parameter estimates so we can plot them.

```
def fit_model_grad_descent(
    x_train,
    y_train,
    num_epochs,
    learning_rate,
    initial_params):
  '''
  Fit our model by gradient descent.

  Arguments:
    - x_train: array of input features of shape (p, m)
    - y_train: array of responses of shape (1, m)
    - num_epochs: number of iterations of gradient descent to run
    - learning_rate: learning rate for gradient descent
    - initial_params: dictionary of starting parameter values

  Return:
    - Dictionary of parameter estimates b1, w1 and gradients dJdb1, dJdw1 at each step of gradient descent
  '''
  # Get initial parameter values
  params = initial_params

  # For loop should iterate over the number of epochs
  b_history = np.zeros((num_epochs + 1,))
  w_history = np.zeros((num_epochs + 1,))
  b_history[0] = params['b1'].copy()
  w_history[0] = params['w1'].copy()
```

```
  for i in range(num_epochs):
    # Do forward propagation
    forward_cache = forward_prop(params, x_train)

    # Do backward propagation
    calc_grad = backward_prop(params, x_train, y_train, forward_cache)

    # Do parameter updates
    params['b1'] = params['b1'] - learning_rate * calc_grad['dJdb1']
    params['w1'] = params['w1'] - learning_rate * calc_grad['dJdw1']

    # Save history of parameter estimates
    b_history[i+1] = params['b1']
    w_history[i+1] = params['w1']

  # Return
  return({
    'b_history': b_history,
    'w_history': w_history
  })
```

Path taken by gradient descent

```
initial_params = initialize_params(num_features = 1)

param_estimates = fit_model_grad_descent(
    x_train = x_standardized,
    y_train = y,
    num_epochs = 30,
    learning_rate = 1,
    initial_params = initial_params)

make_history_plot(param_estimates, x_standardized, blim = (-3, 1), wlim = (-4, 1), contour_levels = 30)
```

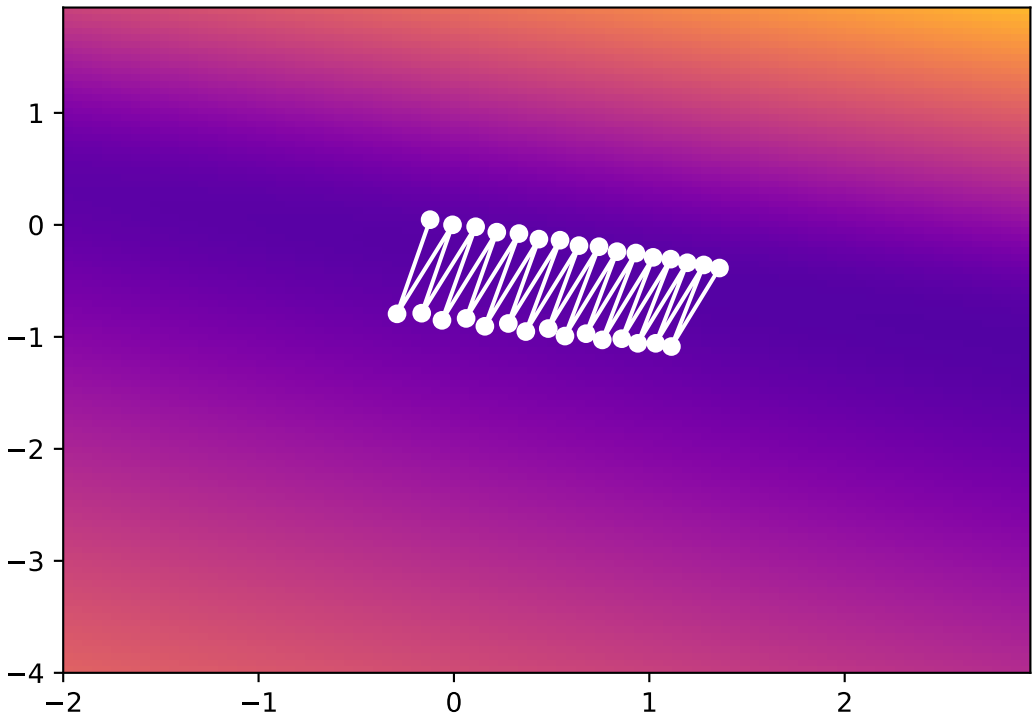


Observations:

- This is working exactly like we'd hope it would!

Path taken by gradient descent – X not centered

```
x_plus_3_5 = x_standardized + 3.5

initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_grad_descent(
    x_train = x_plus_3_5,
    y_train = y,
    num_epochs = 30,
    learning_rate = 1,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-2, 3), wlim = (-4, 2), grid_size = 101)
```

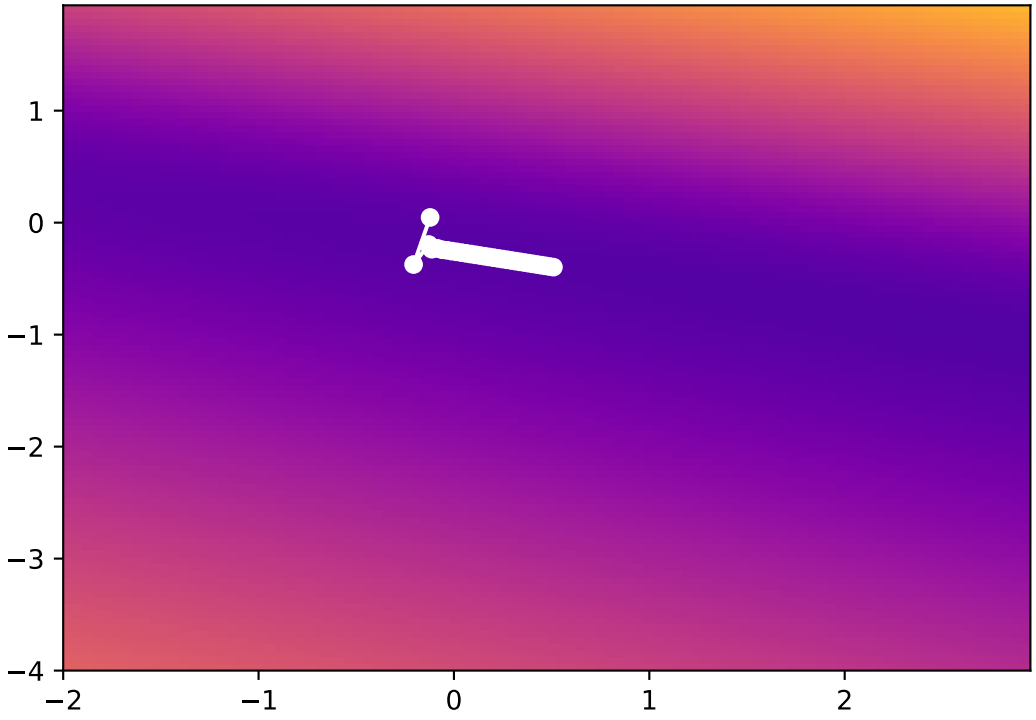


Observations:

- One effect of centering the x variable is that it can make the log-likelihood surface look more like a bowl than a valley (no guarantee centering will totally help, but it will somewhat help).
- If you have a valley, the gradient mostly points straight up the side. You can end up going back and forth on the valley walls and not progressing along the bottom very much.
- Algorithms like RMSProp and Adam address this by using a weighted average of values of the gradient over the last several steps, or adding "momentum" to our particle. This cancels out oscillations pointing in opposite directions and builds up momentum along the bottom.

Path taken by gradient descent – X not centered, learning rate 0.5

```
initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_grad_descent(
    x_train = x_plus_3_5,
    y_train = y,
    num_epochs = 30,
    learning_rate = 0.5,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-2, 3), wlim = (-4, 2), grid_size = 101)
```

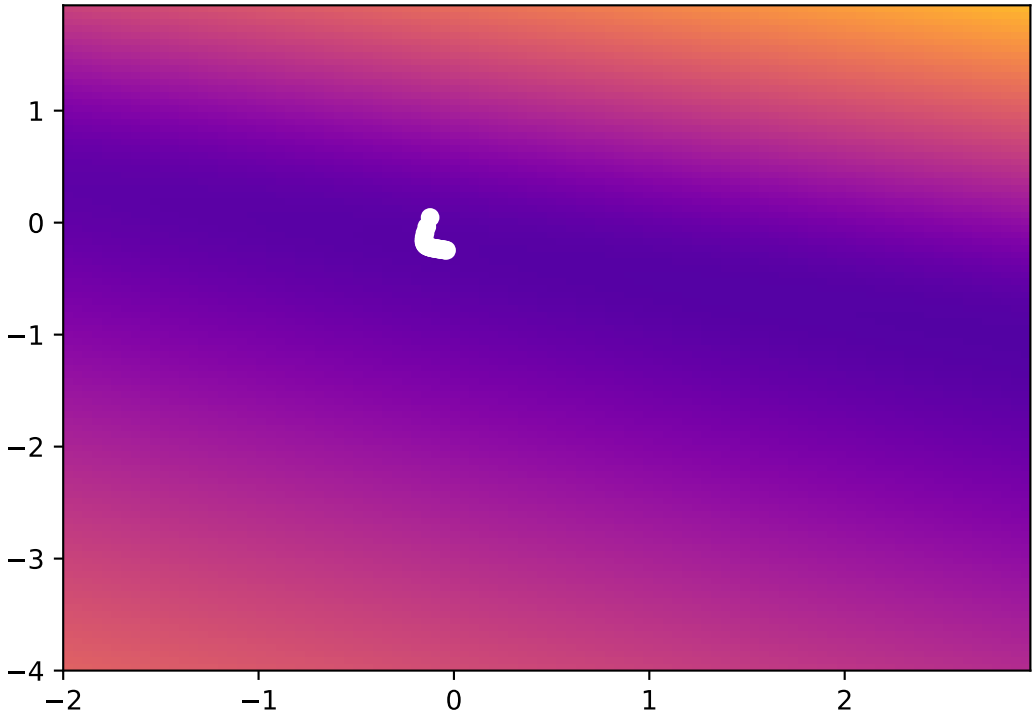


Observations:

- A smaller learning rate can reduce oscillations, but also... slows down the learning rate.

Path taken by gradient descent – X not centered, learning rate 0.1

```
initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_grad_descent(
    x_train = x_plus_3_5,
    y_train = y,
    num_epochs = 30,
    learning_rate = 0.1,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-2, 3), wlim = (-4, 2), grid_size = 101)
```

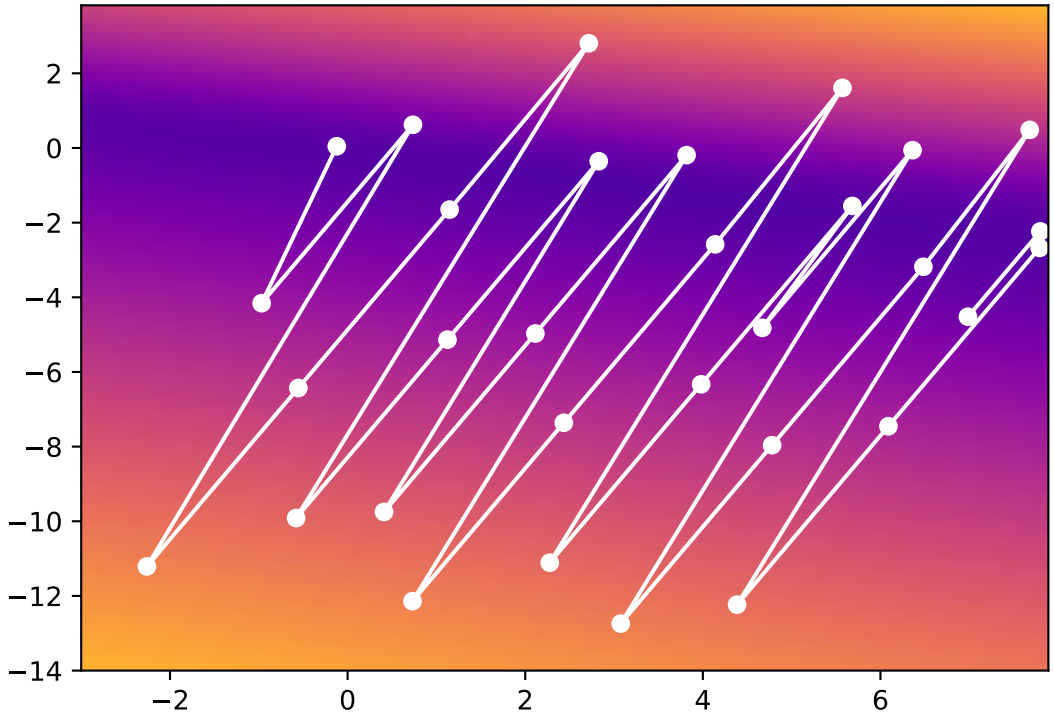


Observations:

- A really small learning rate is very not good.

Path taken by gradient descent – X not centered, learning rate 5

```
initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_grad_descent(
    x_train = x_plus_3_5,
    y_train = y,
    num_epochs = 30,
    learning_rate = 5,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-3, 8), wlim = (-14, 4), grid_size = 101)
```



Observations:

- A really big learning rate is even worse.
- In this example it's sortof OK because the model gets back to the right area.
- In more complicated models you might have shot off very far away from the region of low cost.

Original data

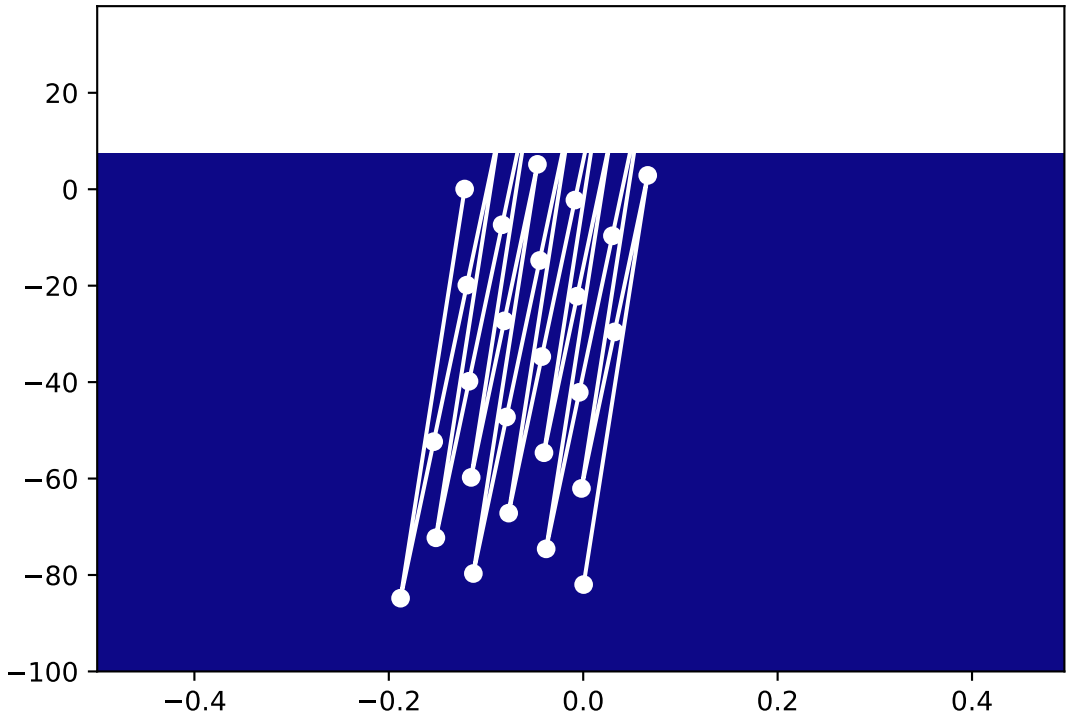
Our original data are very far from centered and scaled:

```
bpd_data.head()
```

```
##    birthweight  BPD
## 0          850    1
## 1         1500    0
## 2         1360    1
## 3          960    0
## 4         1560    0
```

```
initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_grad_descent(
    x_train = x,
    y_train = y,
    num_epochs = 30,
    learning_rate = 0.1,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-0.5, 0.5), wlim = (-100, 20), grid_size = 201)
```

```
## /Users/eray/anaconda3/bin/python:16: RuntimeWarning: divide by zero encountered in log
## /Users/eray/anaconda3/lib/python3.7/site-packages/matplotlib/colors.py:973: RuntimeWarning: invalid value
##   resdat /= (vmax - vmin)
## /Users/eray/anaconda3/lib/python3.7/site-packages/matplotlib/colors.py:527: RuntimeWarning: invalid value
##   xa[xa < 0] = -1
```



Observations:

- Nothing even works.

Stochastic Gradient Descent

Problem:

If we have a large data set, then:

- The forward and backward propagations can take a long time
- The full data set may not fit in your computer's memory

Solution:

Process your data in "minibatches" of a smaller number of observations at a time.

- Often, batch size is a relatively small power of 2: 32, 64, 128, 256, 512
- Guideline: pick batch size to be largest amount your computer can easily handle.
- If your full data set is not that large, this will not offer any advantages.

Terminology:

- One **batch** is a subset of your training data used to calculate one gradient descent step
- One **epoch** is a complete pass through the full data set (many batches)
- The method is called **stochastic gradient descent** because you are processing a random subset of your data in each gradient descent step

Batch Sizes in Our Example:

```
print(x_plus_3_5.shape)
```

```
## (1, 223)
```

```
223 / 64
```

```
## 3.484375
```

```
223 // 64
```

```
## 3
```

```
223 - 64*3
```

```
## 31
```

Suppose I use a batch size of 64. There will be 4 batches per epoch:

- Batch 1 has 64 observations (indices 0 to 63)
- Batch 2 has 64 observations (indices 64 to 127)
- Batch 3 has 64 observations (indices 128 to 191)
- Batch 4 has 31 observations (indices 192 to 222)

Note about for loops:

For loops repeat a block of code many times:

```
for i in range(5):
  print(str(i**2))
```

```
## 0
## 1
## 4
## 9
## 16
```

You can nest multiple loops:

```
for i in range(4):
  for j in range(3):
    print("i = " + str(i) + ", j = " + str(j))
```

```
## i = 0, j = 0
## i = 0, j = 1
## i = 0, j = 2
## i = 1, j = 0
## i = 1, j = 1
## i = 1, j = 2
## i = 2, j = 0
## i = 2, j = 1
## i = 2, j = 2
## i = 3, j = 0
## i = 3, j = 1
## i = 3, j = 2
```

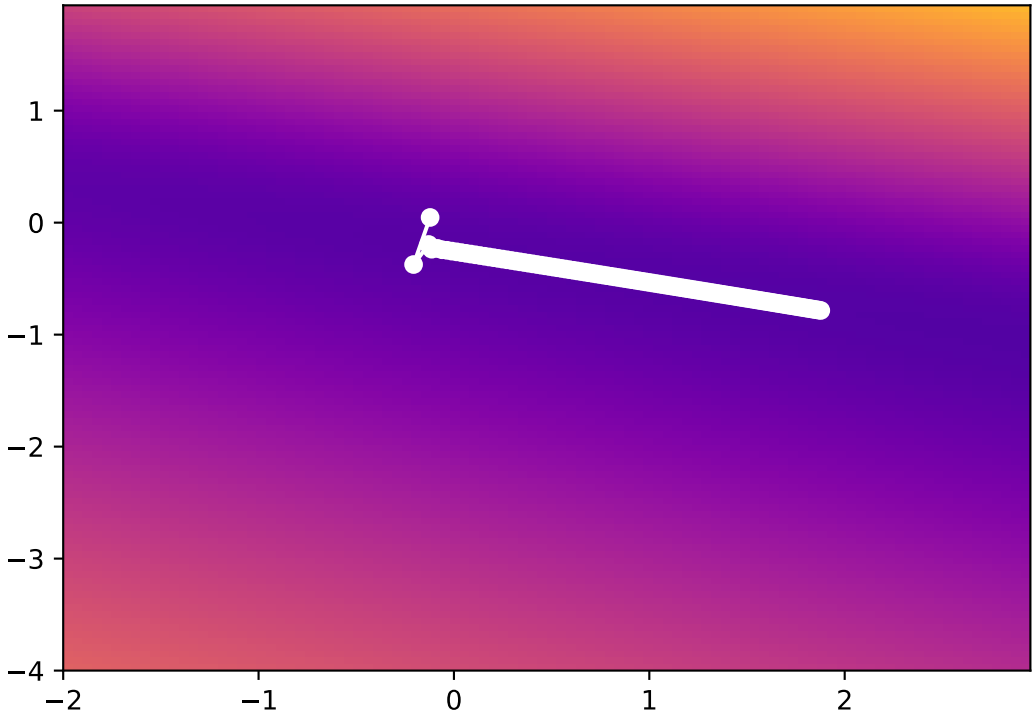
Outline of code for stochastic gradient descent

1. Figure out how many batches you will need for the specified batch size
2. For each epoch `i = 0, ..., (number of epochs - 1)`
    1. For each batch `j = 0, ..., (number of batches - 1)`
        1. Subset data to the observations in batch `j`
        2. Do forward propagation based on observations in batch `j`
        3. Do backward propagation based on observations in batch `j`
        4. Do parameter updates based on gradients from observations in batch `j`

Example code for stochastic gradient descent

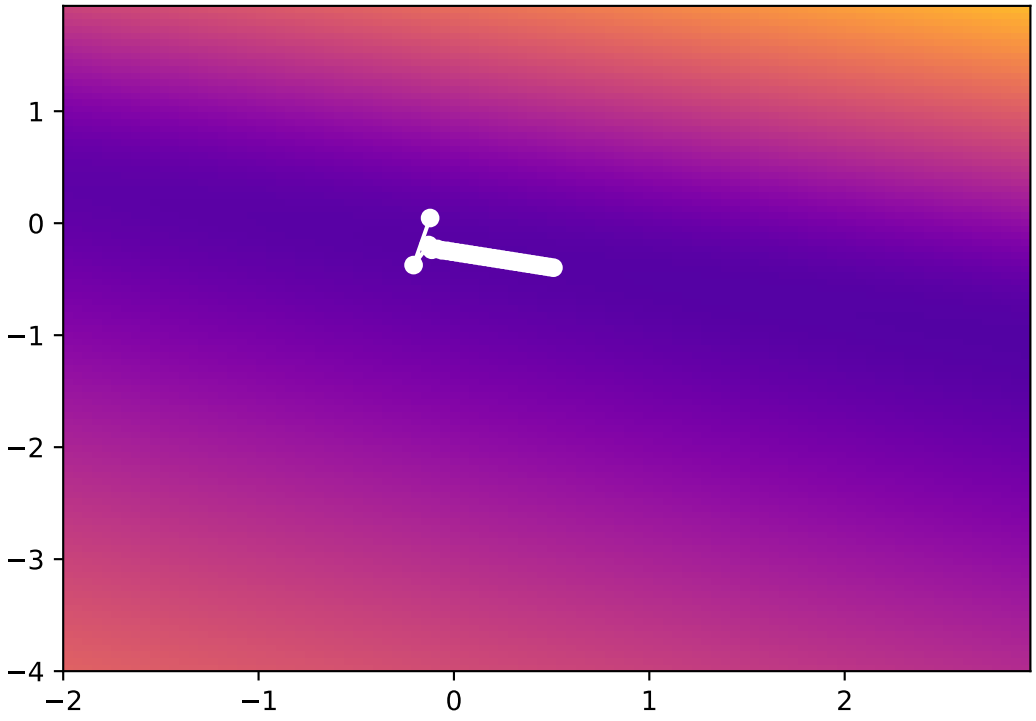
```
def fit_model_stochastic_grad_descent(
    x_train,
    y_train,
    num_epochs,
    batch_size,
    learning_rate,
    initial_params):
  '''
  Fit our model by stochastic gradient descent.

  Arguments:
    - x_train: array of input features of shape (p, m)
    - y_train: array of responses of shape (1, m)
    - num_epochs: number of iterations of gradient descent to run
    - batch_size: number of observations to include in each batch
    - learning_rate: learning rate for gradient descent
    - initial_params: dictionary of starting parameter values

  Return:
    - Dictionary of parameter estimates b1, w1 and gradients dJdb1, dJdw1 at each step of gradient descent
  '''
  params = initial_params

  # Figure out how many batches to use
  m = x_train.shape[1]
  num_batches = math.ceil(m / batch_size)

  # Outer for loop iterates from 0 to number of epochs - 1
  for i in range(num_epochs):
    # Inner loop iterates from 0 to number of batches - 1
    for j in range(num_batches):
      # Set up data for current batch
      end_ind = (j+1)*batch_size
      if j == num_batches:
        # last batch includes up to the last observation
        end_ind = m

      x_current_batch = x_train[:, (j*batch_size):end_ind]
      y_current_batch = y_train[:, (j*batch_size):end_ind]

      # Do forward propagation
      forward_cache = forward_prop(params, x_current_batch)

      # Do backward propagation
      calc_grad = backward_prop(params, x_current_batch, y_current_batch, forward_cache)

      # Do parameter updates
      params['b1'] = params['b1'] - learning_rate * calc_grad['dJdb1']
      params['w1'] = params['w1'] - learning_rate * calc_grad['dJdw1']

  # Return
  return(params)
```

```
initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_stochastic_grad_descent(
    x_train = x_plus_3_5,
    y_train = y,
    num_epochs = 30,
    batch_size = 64,
    learning_rate = 0.5,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-2, 3), wlim = (-4, 2), grid_size = 101)
```



Compare to previous equivalent version with gradient descent

```
initial_params = initialize_params(num_features = 1)
param_estimates = fit_model_grad_descent(
    x_train = x_plus_3_5,
    y_train = y,
    num_epochs = 30,
    learning_rate = 0.5,
    initial_params = initial_params)

make_history_plot(param_estimates, x_plus_3_5, blim = (-2, 3), wlim = (-4, 2), grid_size = 101)
```



- We are able to make substantially more progress in the same number of passes through our data set.
- Basically, we made 4 times as many gradient update steps with the same amount of computation.
- In practice, it takes time for the computer to get data from your drive to your processor. If your full data set is not that large and can can be processed all at once by your CPU or GPU then stochastic gradient descent may not make a practical difference in compute time.