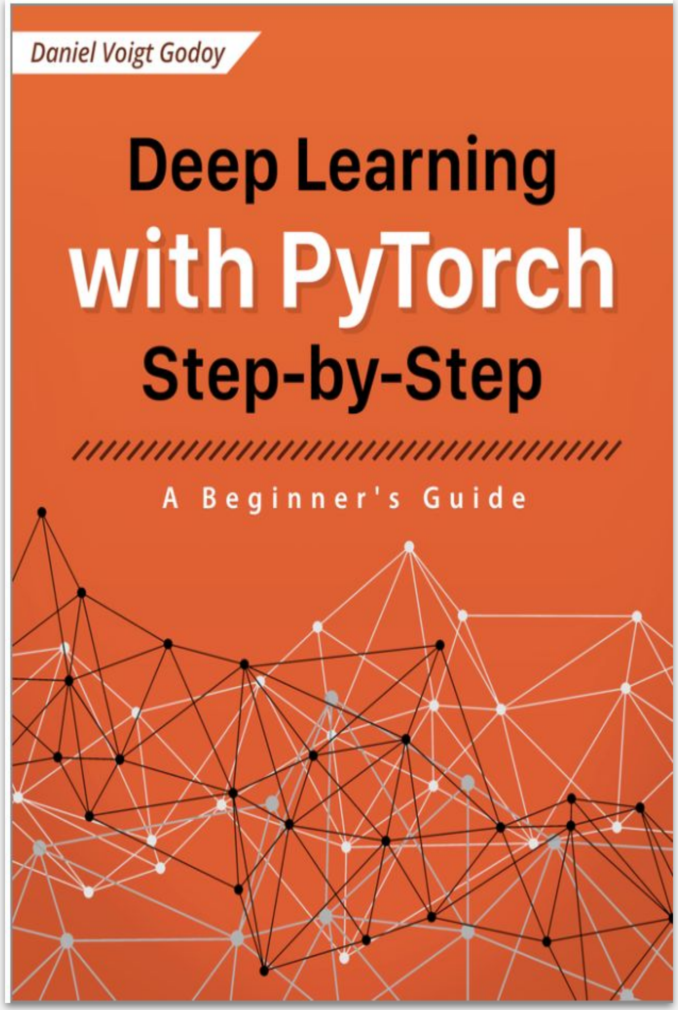
DCA0305

ivanovitch.silva@ufrn.br

Machine Learning Based Systems Design

Visualizing Gradient Descent

Regression Problem



https://github.com/dvgodoy/PyTorchStepByStep

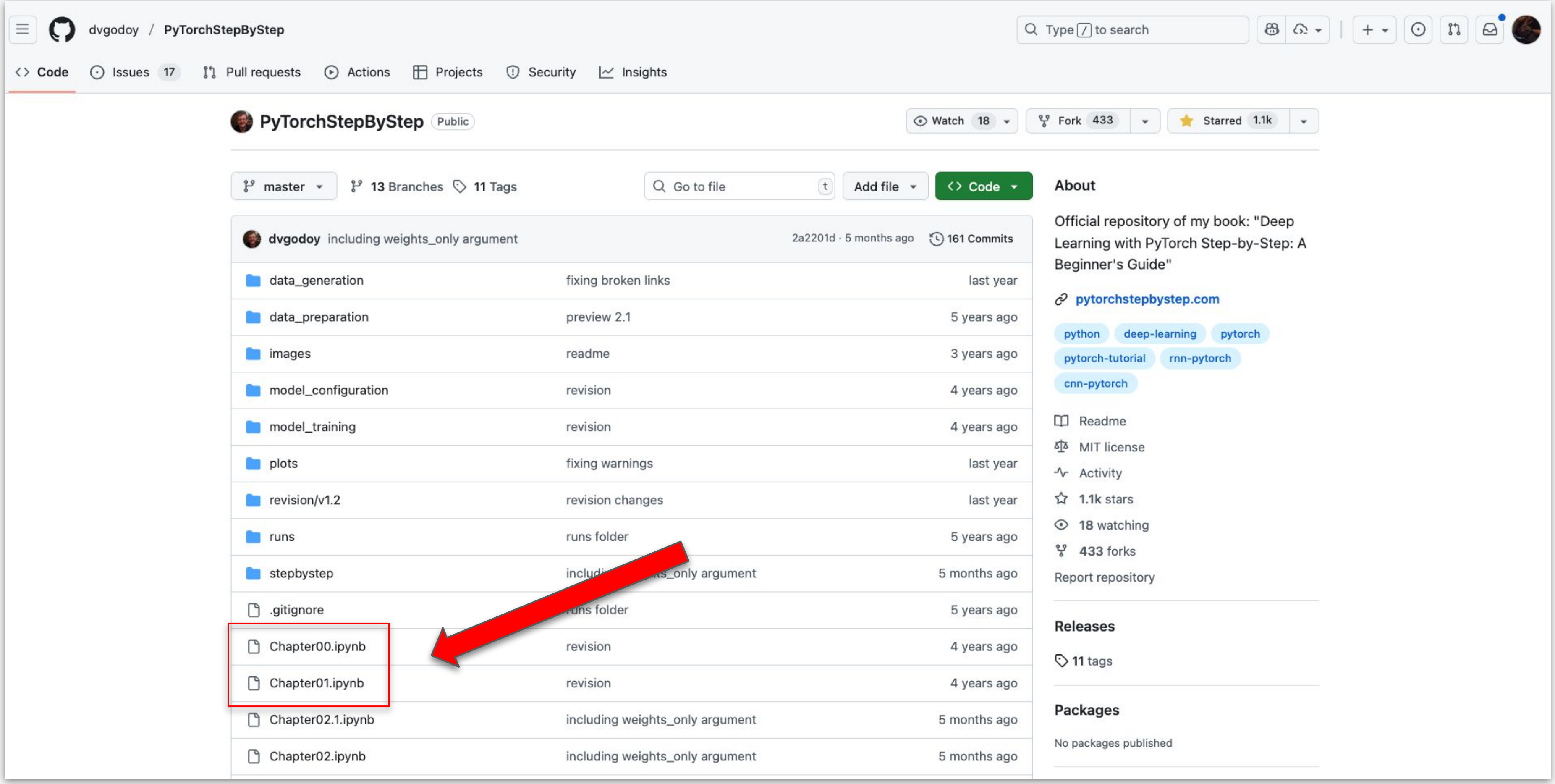
GitHub

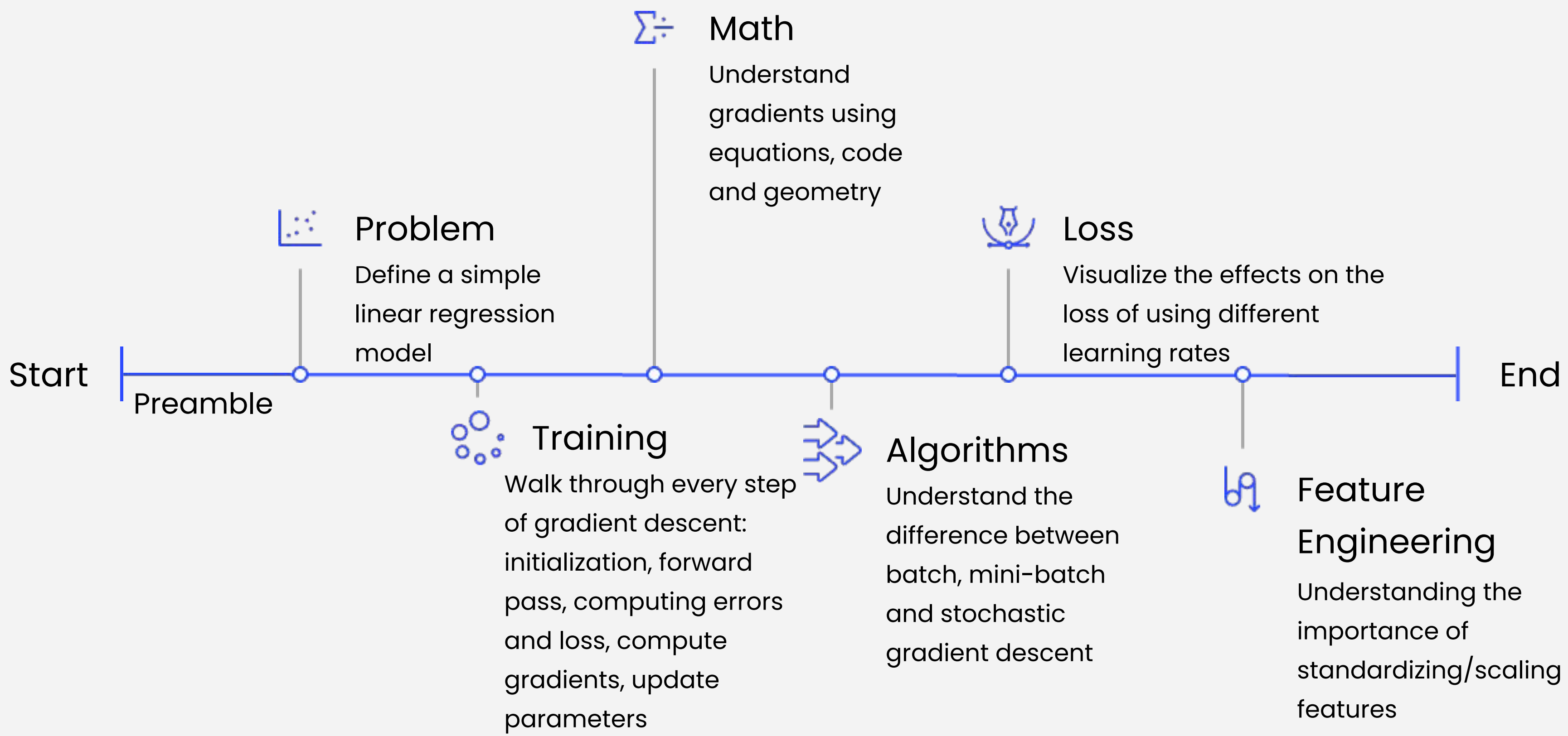
Part I: Fundamentals (gradient descent, training linear and logistic regressions in PyTorch)

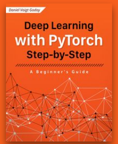
Part II: Computer Vision (deeper models and activation functions, convolutions, transfer learning, initialization schemes)

Part III: Sequences (RNN, GRU, LSTM, seq2seq models, attention, self-attention, transformers)

Part IV: Natural Language Processing (tokenization, embeddings, contextual word embeddings, ELMo, BERT, GPT-2)


dvgodoy / PyTorchStepByStep
Type / to search
Code
Issues 17
Pull requests
Actions
Projects
Security
Insights
PyTorchStepByStep Public
Watch 18
Fork 433
Starred 1.1k
master
13 Branches
11 Tags
Go to file
t
Add file
Code
dvgodoy including weights_only argument
2a2201d · 5 months ago
161 Commits
data_generation fixing broken links last year
data_preparation preview 2.1 5 years ago
images readme 3 years ago
model_configuration revision 4 years ago
model_training revision 4 years ago
plots fixing warnings last year
revision/v1.2 revision changes last year
runs runs folder 5 years ago
stepbystep 5 months ago
.gitignore 5 years ago
Chapter00.ipynb revision 4 years ago
Chapter01.ipynb revision 4 years ago
Chapter02.1.ipynb including weights_only argument 5 months ago
Chapter02.ipynb including weights_only argument 5 months ago
About
Official repository of my book: "Deep Learning with PyTorch Step-by-Step: A Beginner's Guide"
pytorchstepbystep.com
python
deep-learning
pytorch
pytorch-tutorial
rnn-pytorch
cnn-pytorch
Readme
MIT license
Activity
1.1k stars
18 watching
433 forks
Report repository
Releases
11 tags
Packages
No packages published





Visualizing Gradient Descent

Preamble

$$y = 1x^2 + 2x + 3$$

$$y = w_1x^2 + w_2x + w_3$$

Preamble

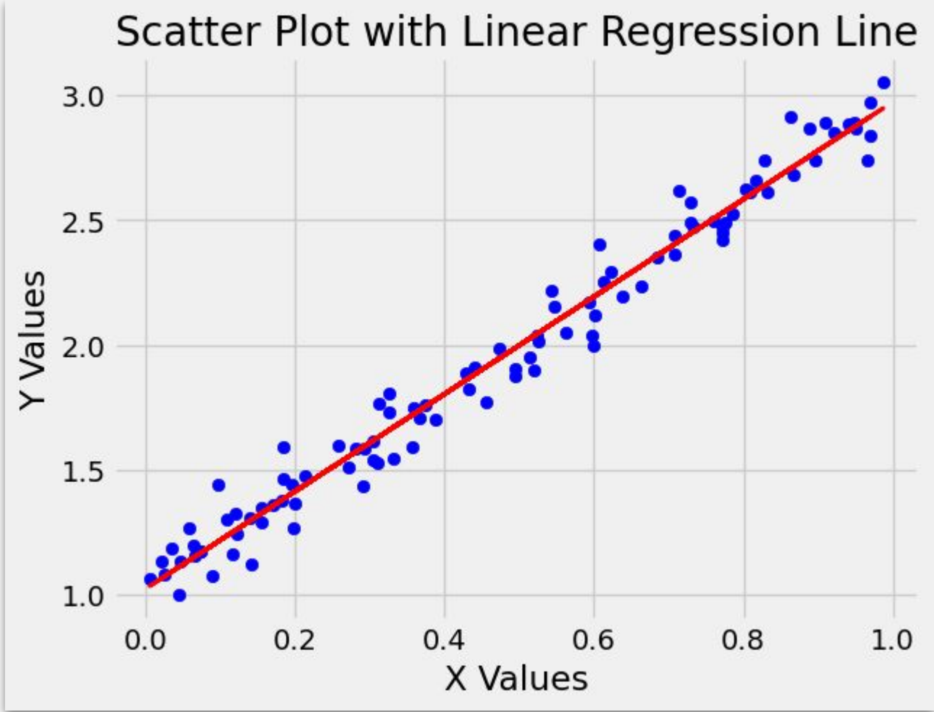
```python
import numpy as np  # Import the NumPy library

# Define a 2nd degree polynomial: 1*x^2 + 2*x + 3
polynomial = np.poly1d([1, 2, 3])

# Print the polynomial
print(polynomial)

# Set the number of samples
n = 20

# Generate 'n' random samples from a normal distribution and scale by 5
x = np.random.randn(n, 1) * 5

# Apply the polynomial to the values of x to get corresponding y values
y = polynomial(x)
```

Preamble

```
# Assume samples X and Y are prepared elsewhere
xx = np.hstack([x*x, x, np.ones_like(x)])
w = torch.randn(3, 1, requires_grad=True) # the 3 coefficients
x = torch.tensor(xx, dtype=torch.float32) # input sample
y = torch.tensor(y, dtype=torch.float32) # output sample
optimizer = torch.optim.NAdam([w], lr=0.01)

# Run optimizer
for _ in range(1000):
    optimizer.zero_grad()
    y_pred = x @ w
    mse = torch.mean(torch.square(y - y_pred))
    mse.backward()
    optimizer.step()
print(w)
```

```
tensor([[1.0050],
        [1.9969],
        [2.6778]], requires_grad=True)
```

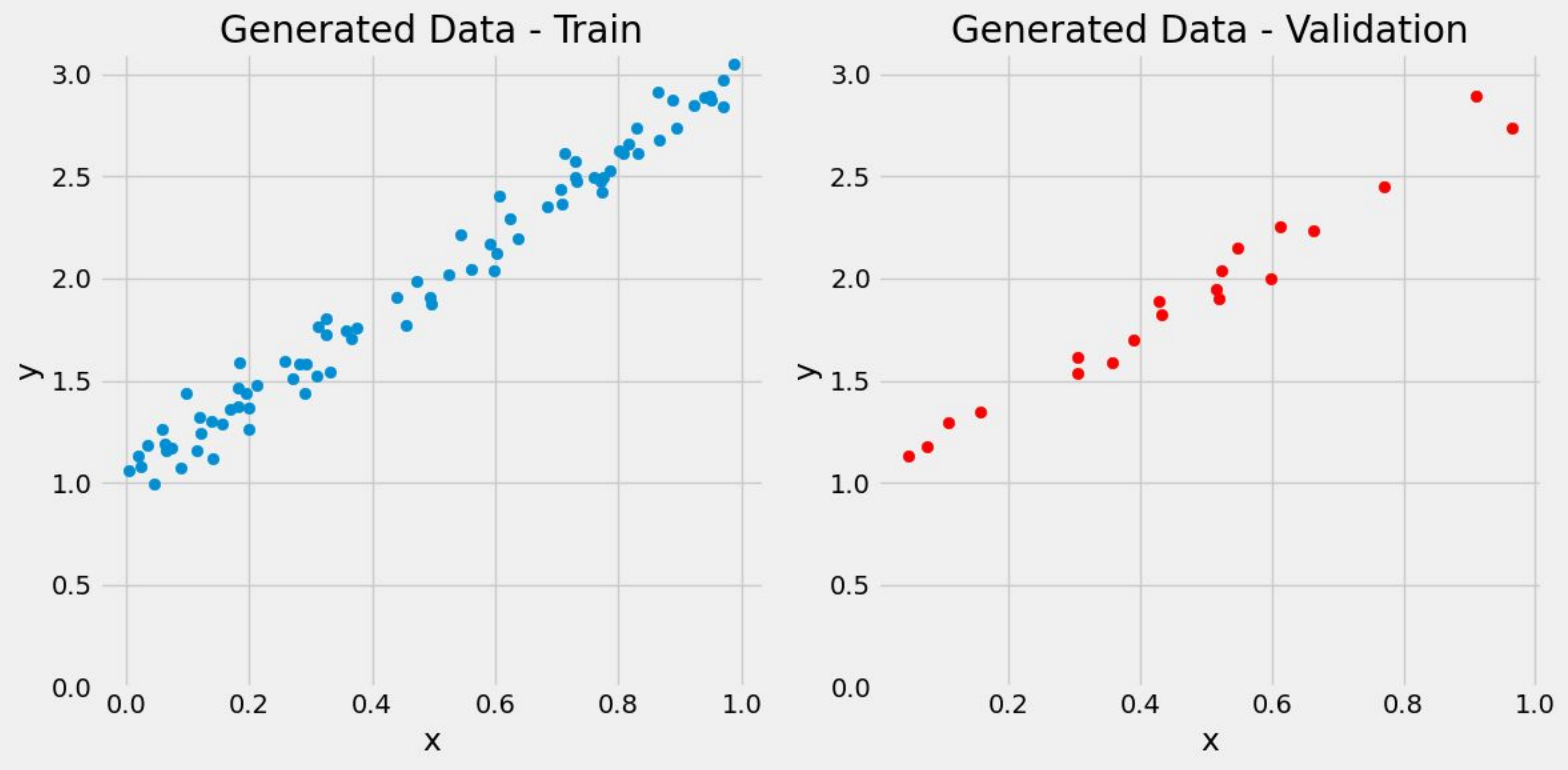
$$y = w_1 x^2 + w_2 x + w_3$$



A linear regression model

$$y = b + wx + \epsilon$$

Synthetic Data Generation and Train-Validation-Test Split



00

Step

Initializes parameters

$$y = b + wx$$

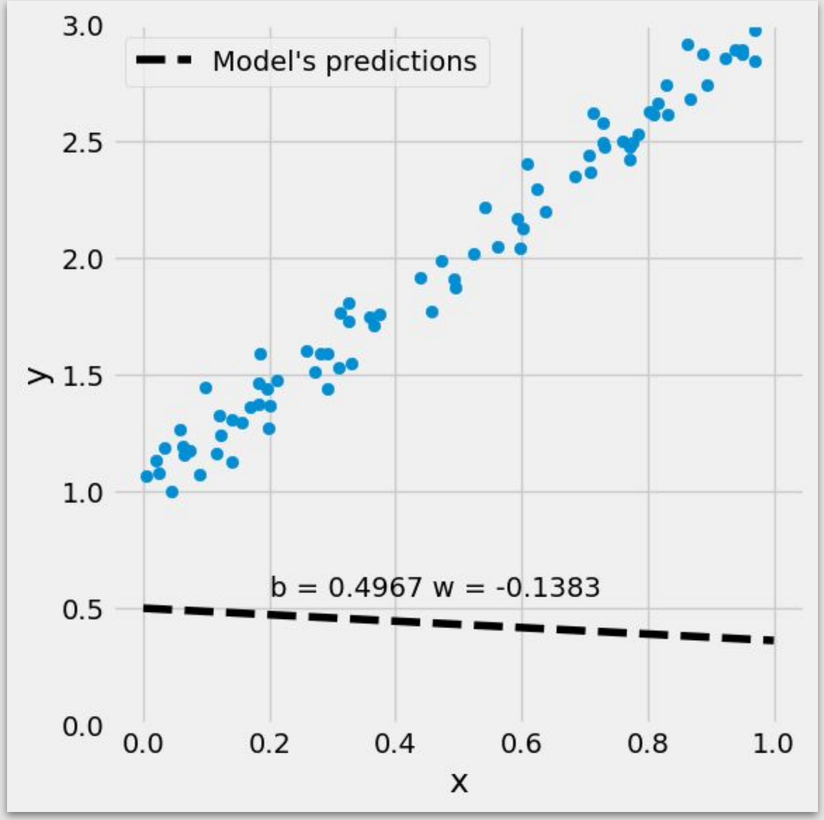
```
# Step 0 - Initializes parameters "b" and "w" randomly

# Sets the seed for the random number generator to 42 for reproducibility
np.random.seed(42)

# Generates a single random number from a standard normal distribution for the bias 'b'
b = np.random.randn(1)

# Generates a single random number from a standard normal distribution for the weight 'w'
w = np.random.randn(1)

# Prints the generated values of 'b' and 'w'
print(b, w)

[0.49671415] [-0.1382643]
```

01

Step

Compute predictions

```
# Step 1 - Computes our model's
predicted output - forward pass
yhat = b + w * x_train
```



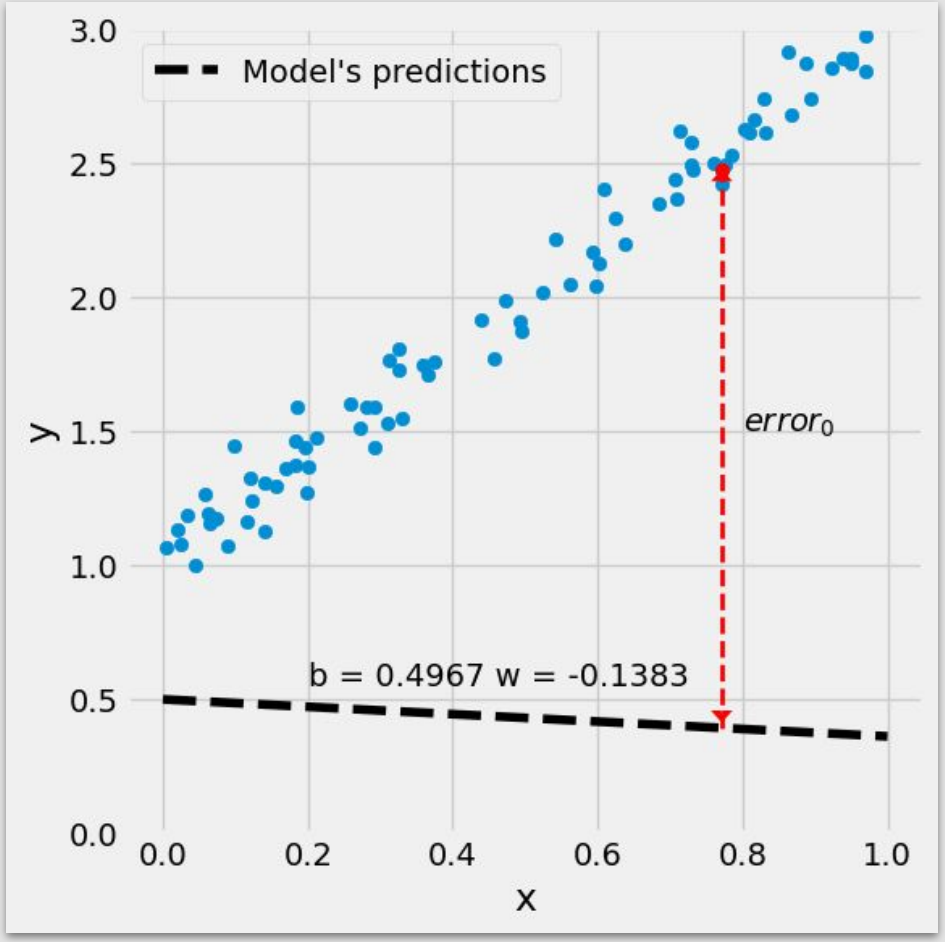
02

Step

Compute the loss

$$\begin{aligned}\text{MSE} &= \frac{1}{n}\sum_{i=1}^{n}\text{error}_i^{\,2} \\ &= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(b + wx_i - y_i)^2\end{aligned}$$

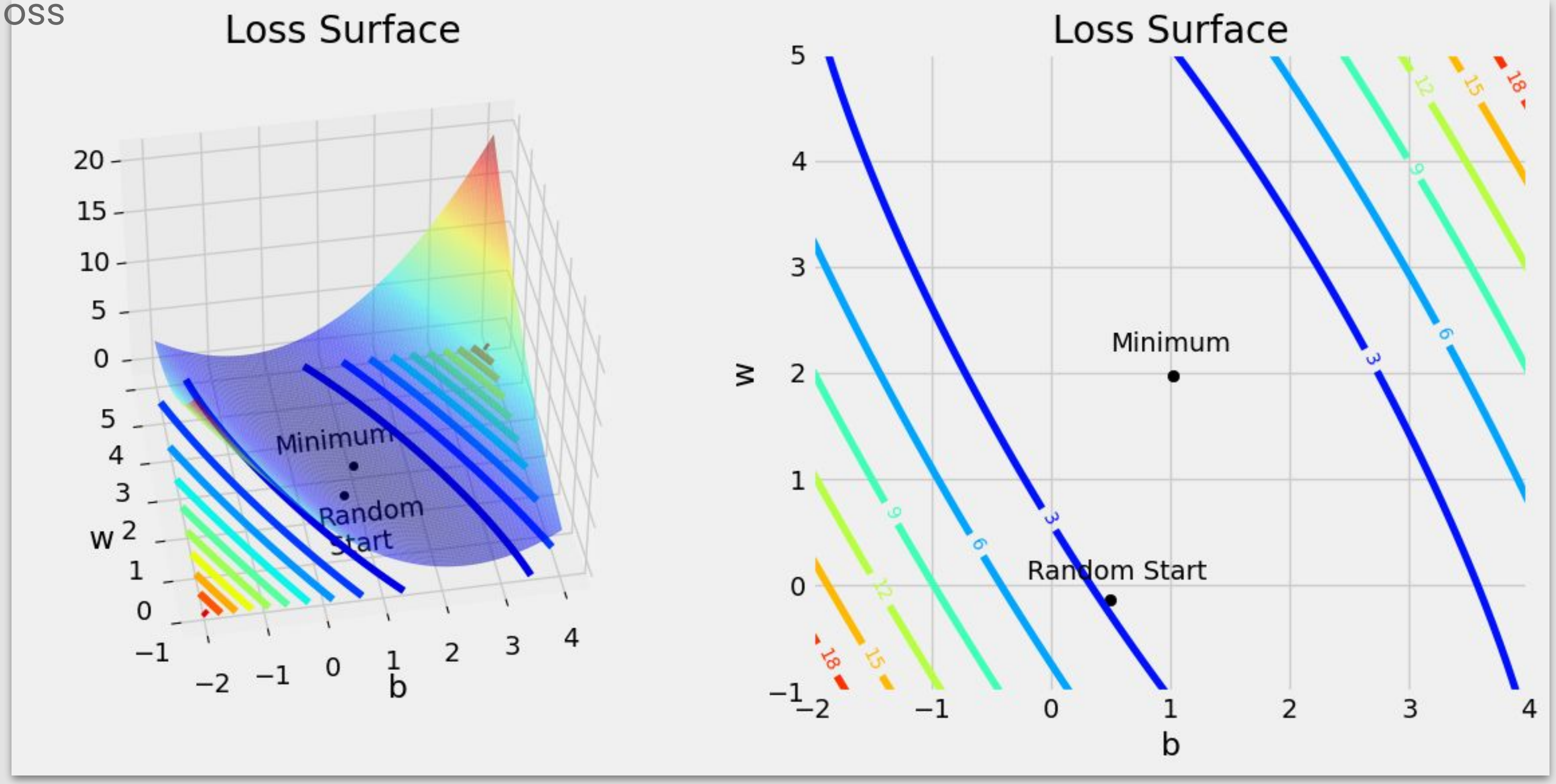
```
# Step 2 - Computing the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train)

# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()
print(loss)

2.7421577700550976
```



02

Step

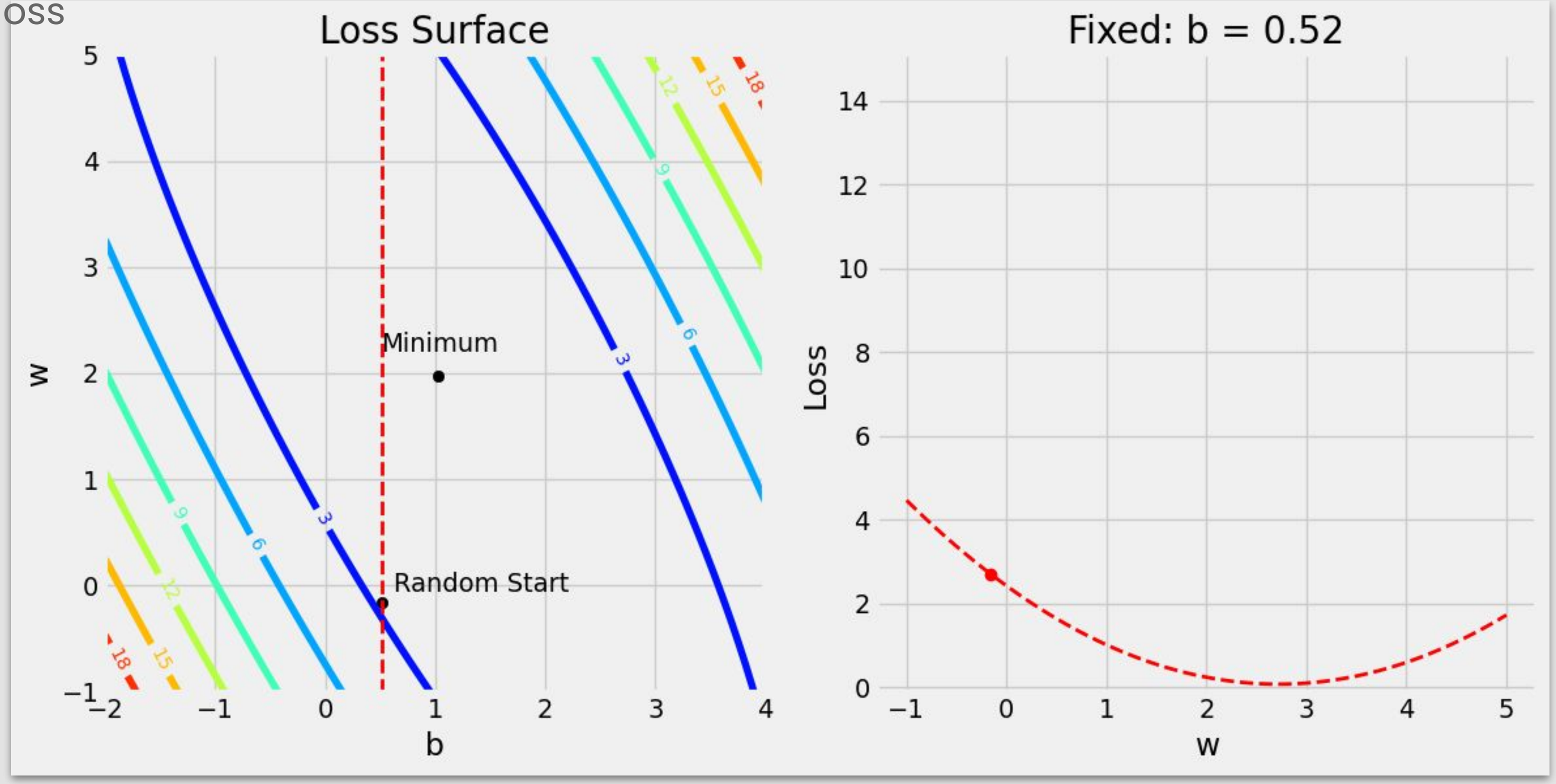
Compute the loss



02

Step

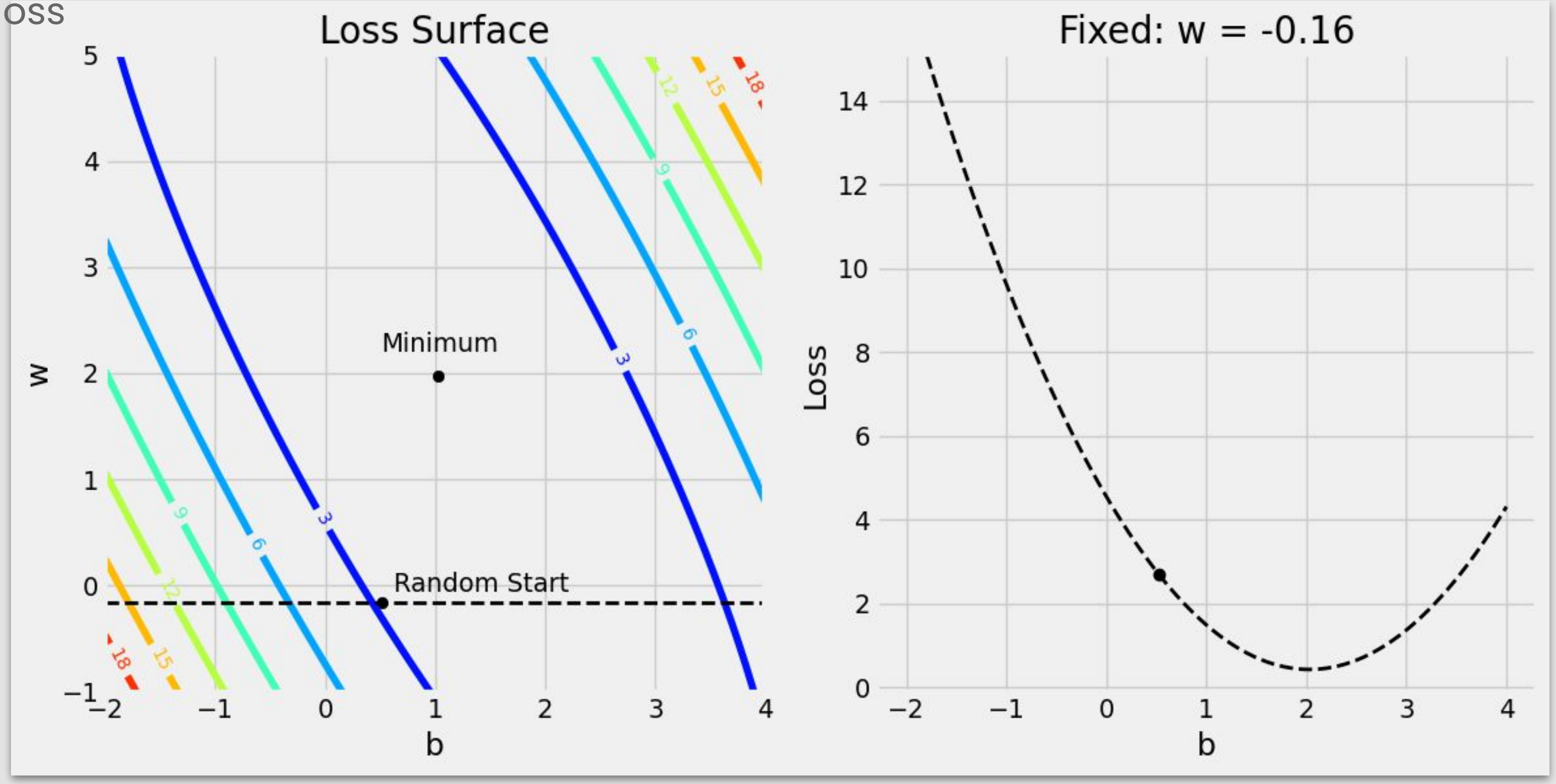
Compute the loss



02

Step

Compute the loss

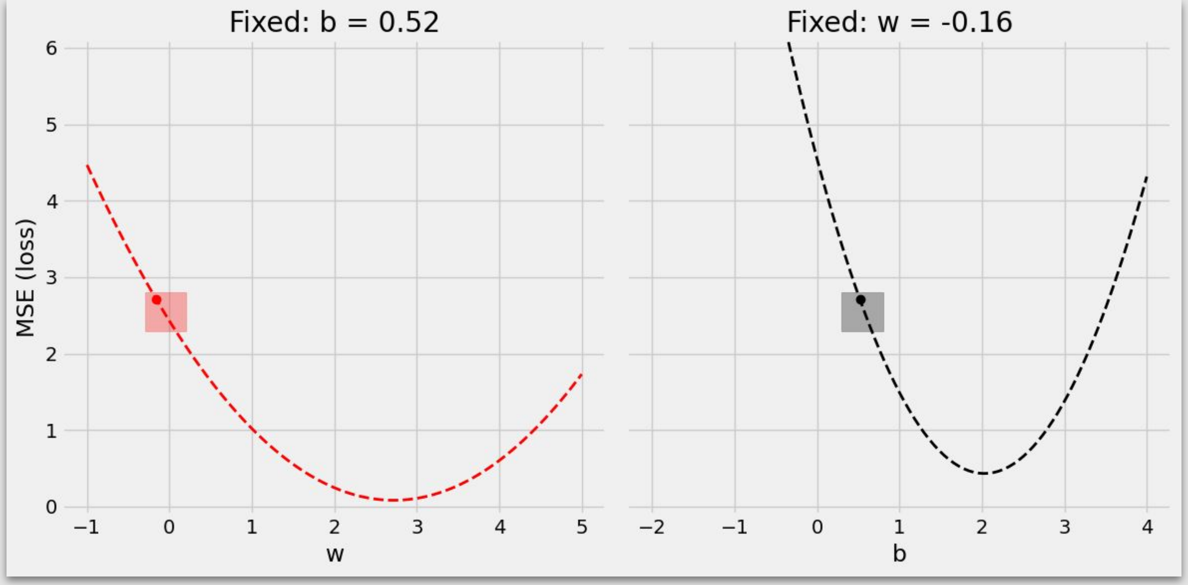


03

Step

Compute the gradients

Gradient = how much the loss changes if ONE parameter changes a little bit!



A derivative tells you how much a given quantity changes when you slightly vary some other quantity. In our case, how much does our **MSE loss change** when we **vary each of our two parameters** separately?

03

Step

Compute the gradients

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} \text{error}_i^{\,2}$$
$$= \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$
$$= \frac{1}{n}\sum_{i=1}^{n} (b + wx_i - y_i)^2$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{\partial \text{MSE}}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} 2(b + wx_i - y_i)$$
$$= 2\frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)$$

$$\frac{\partial \text{MSE}}{\partial w} = \frac{\partial \text{MSE}}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial w} = \frac{1}{n}\sum_{i=1}^{n} 2(b + wx_i - y_i)x_i$$
$$= 2\frac{1}{n}\sum_{i=1}^{n} x_i(\hat{y}_i - y_i)$$

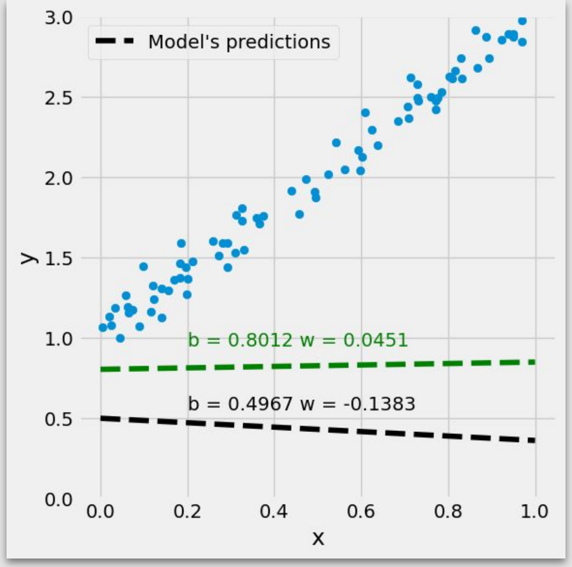
```
# Step 3 - Computes gradients for both "b" and "w" parameters
b_grad = 2 * error.mean()
w_grad = 2 * (x_train * error).mean()
print(b_grad, w_grad)

-3.044811379650508 -1.8337537171510832
```

04

Step

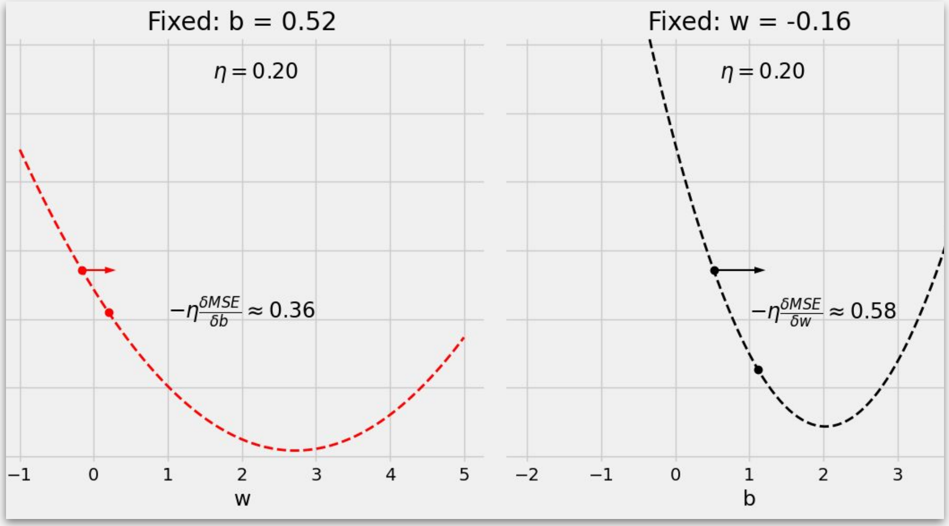
Update the parameters

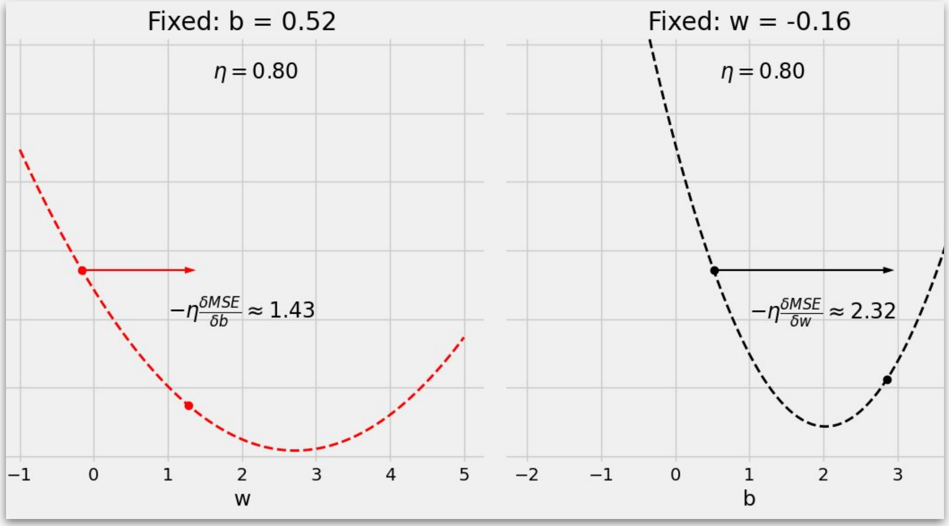
```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1
print(b, w)

# Step 4 - Updates parameters using gradients and the
# learning rate
b = b - lr * b_grad
w = w - lr * w_grad

print(b, w)
[0.49671415] [-0.1382643]
[0.80119529] [0.04511107]
```



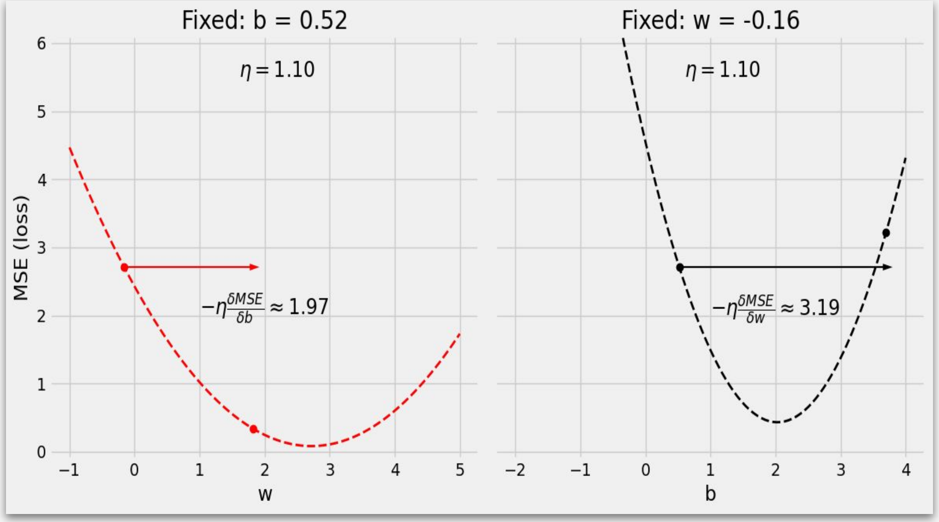
$$b = b - \eta \frac{\partial \mathrm{MSE}}{\partial b}$$

$$w = w - \eta \frac{\partial \mathrm{MSE}}{\partial w}$$

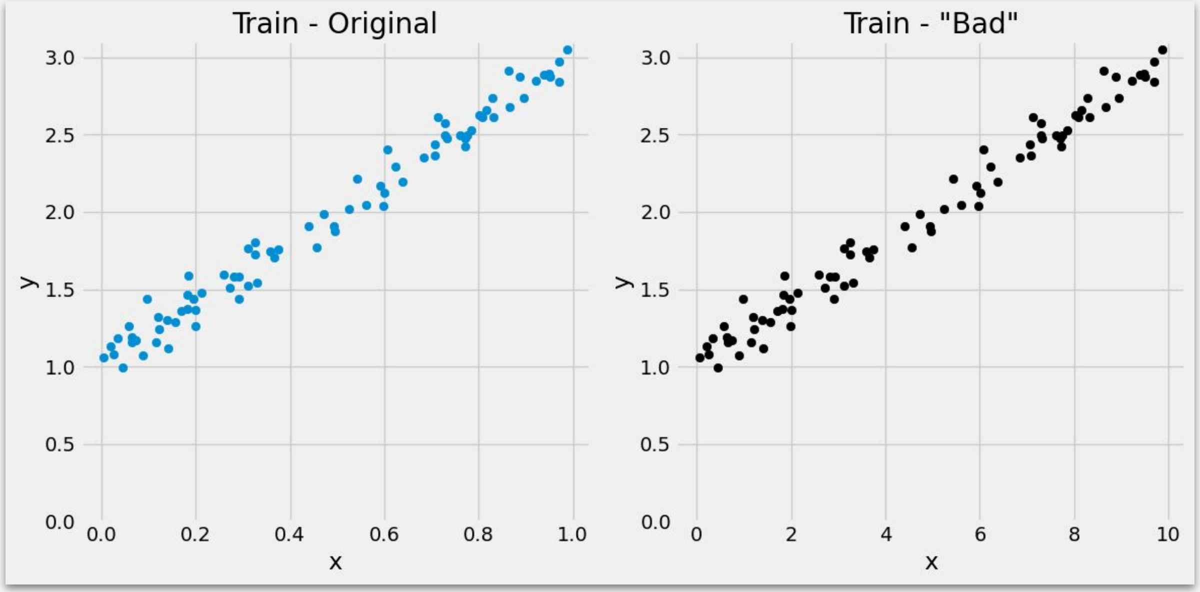


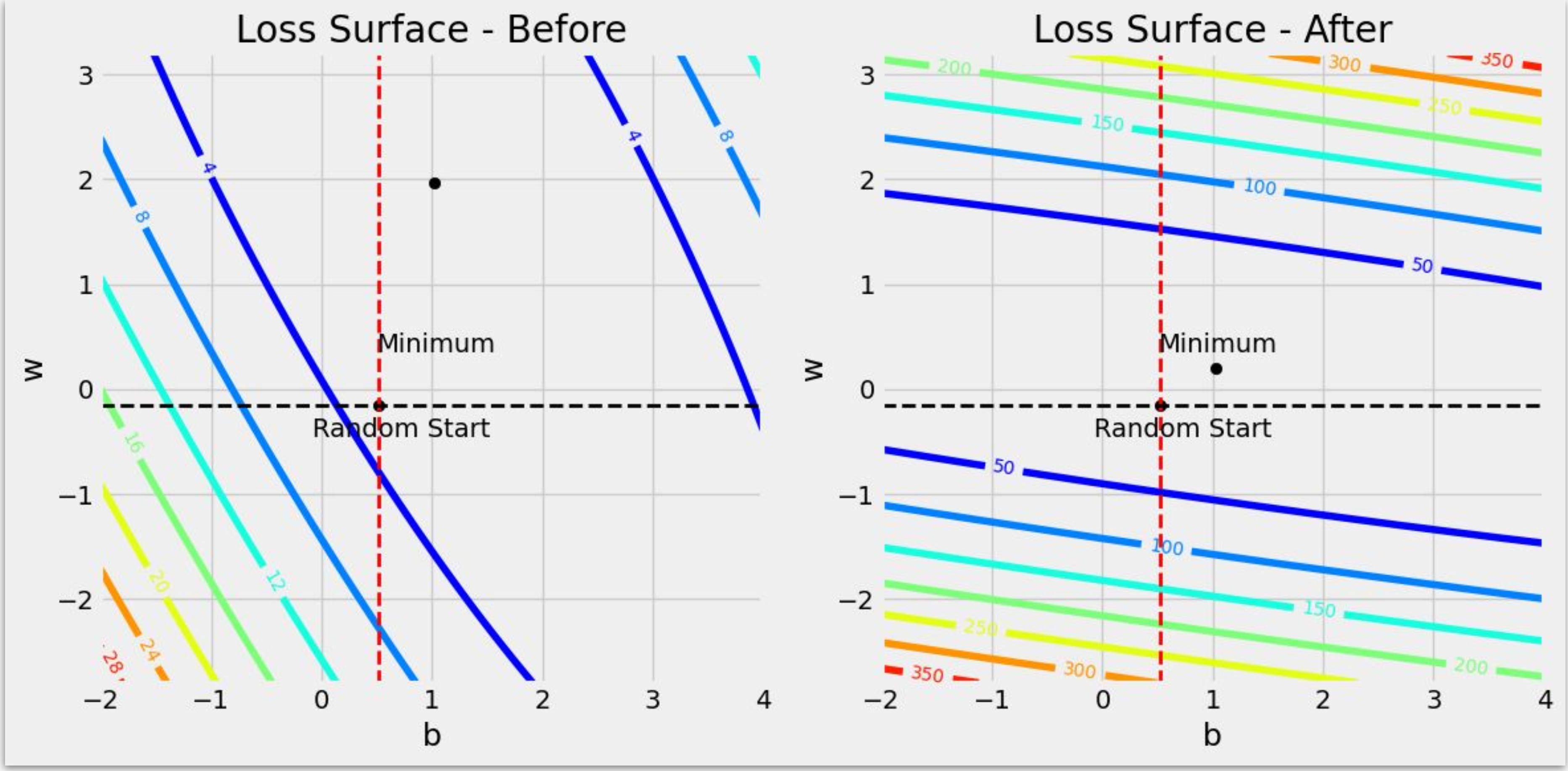


Learning Rate

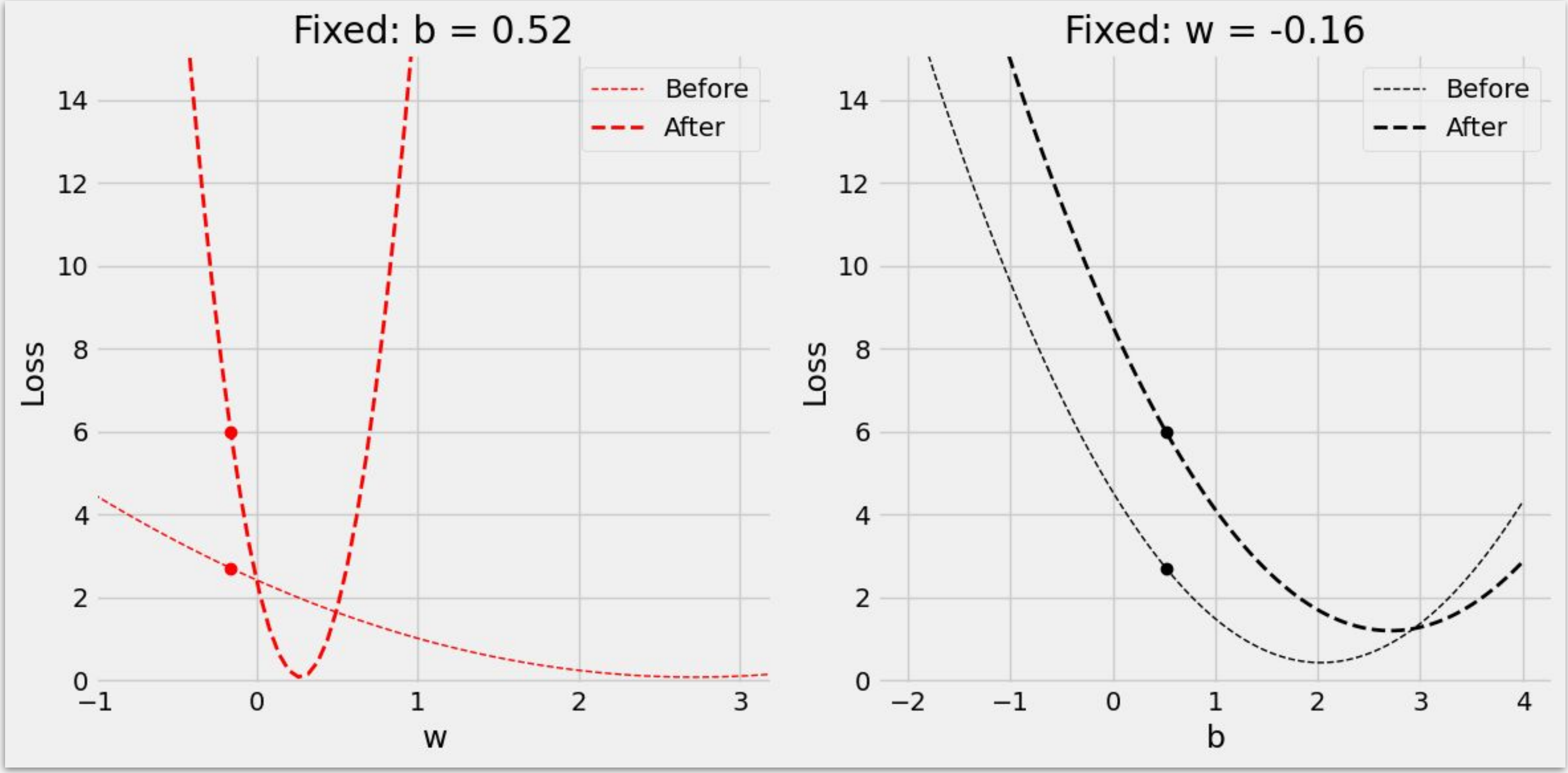


What is the influence of the X scale on the results?





Loss Surface - Before
Loss Surface - After
Minimum
Random Start
Minimum
Random Start
w
b
w
b
4
4
8
8
12
16
20
24
28
50
100
150
200
250
300
350



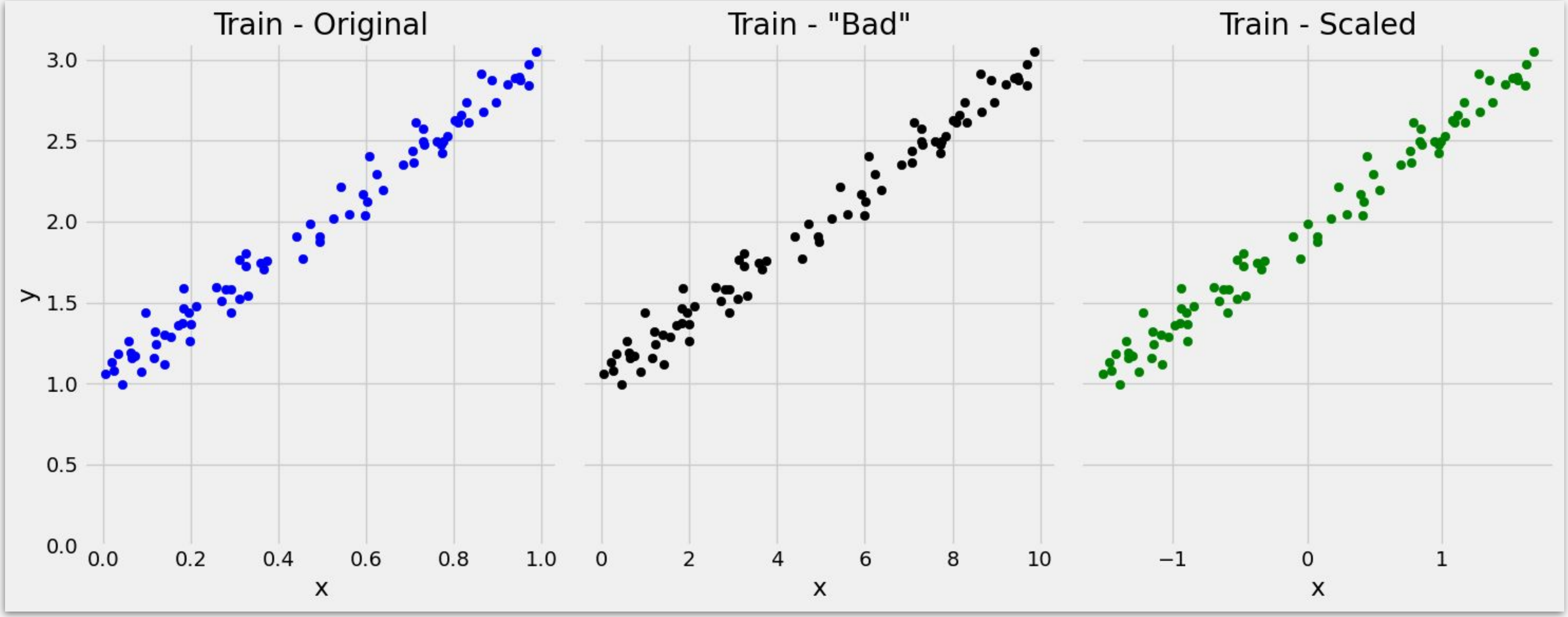
The red curve got <u>much steeper</u> (larger gradient), and thus we must use a <u>lower learning rate</u> to safely descend along it.

How can we fix it?

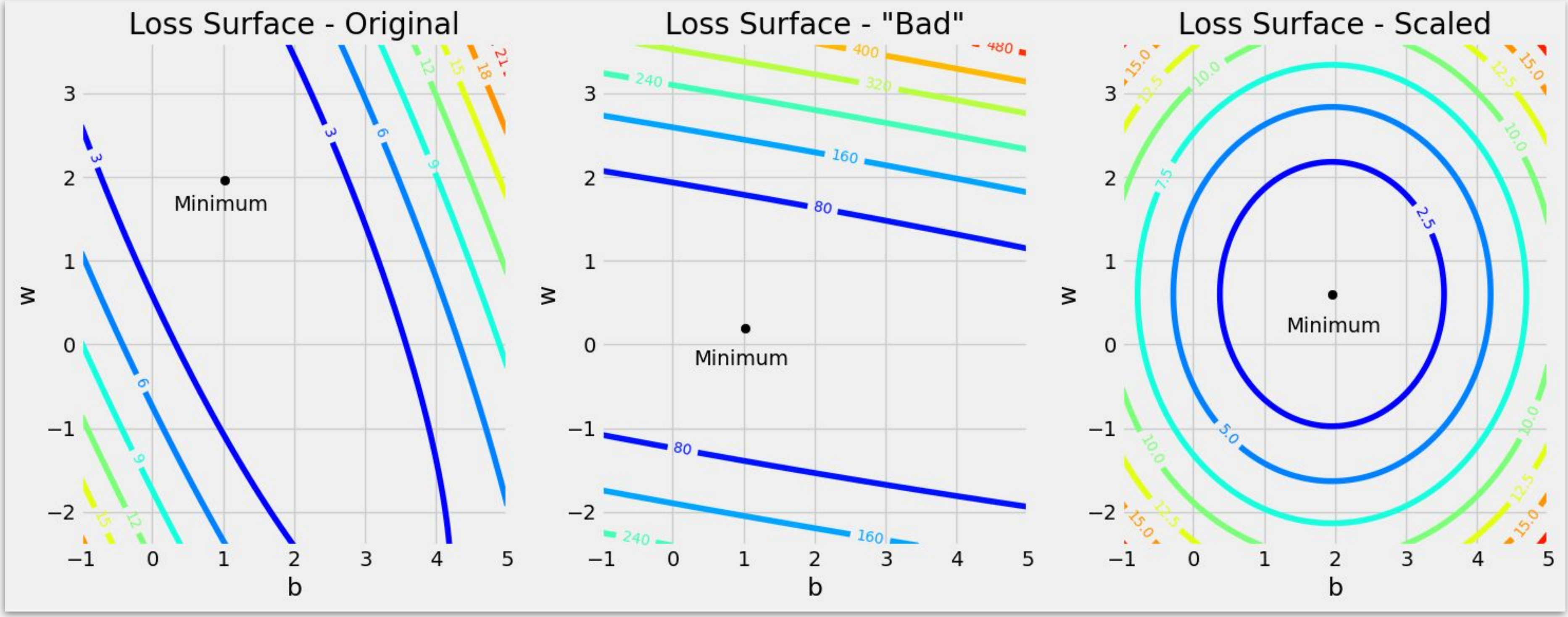
$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma(X) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{X})^2}$$

$$\text{scaled } x_i = \frac{x_i - \overline{X}}{\sigma(X)}$$



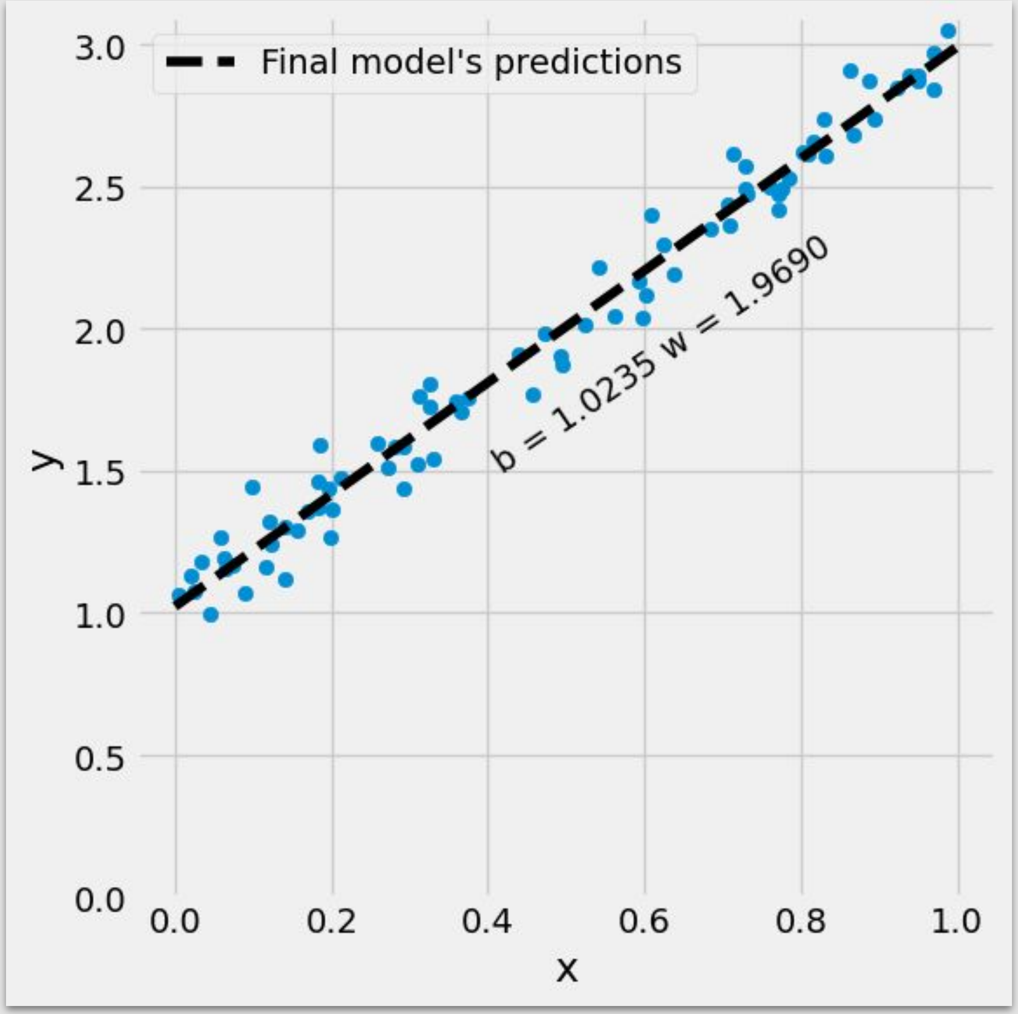
Train - Original
Train - "Bad"
Train - Scaled
y
x
x
x
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
−1
0
1


Loss Surface - Original
Loss Surface - "Bad"
Loss Surface - Scaled
Minimum
Minimum
Minimum
w
b

05

Step

Iterate and Converge



Now we use the updated parameters to <u>go back to Step 1</u> and <u>restart</u> the process.

Definition of Epoch

An epoch is complete whenever every point in the training set(N) has already been used in all steps: forward pass, computing loss, computing gradients, and updating parameters.

- Batch gradient descent

  This is trivial, as it uses all points for computing the loss — one epoch is the same as one update.

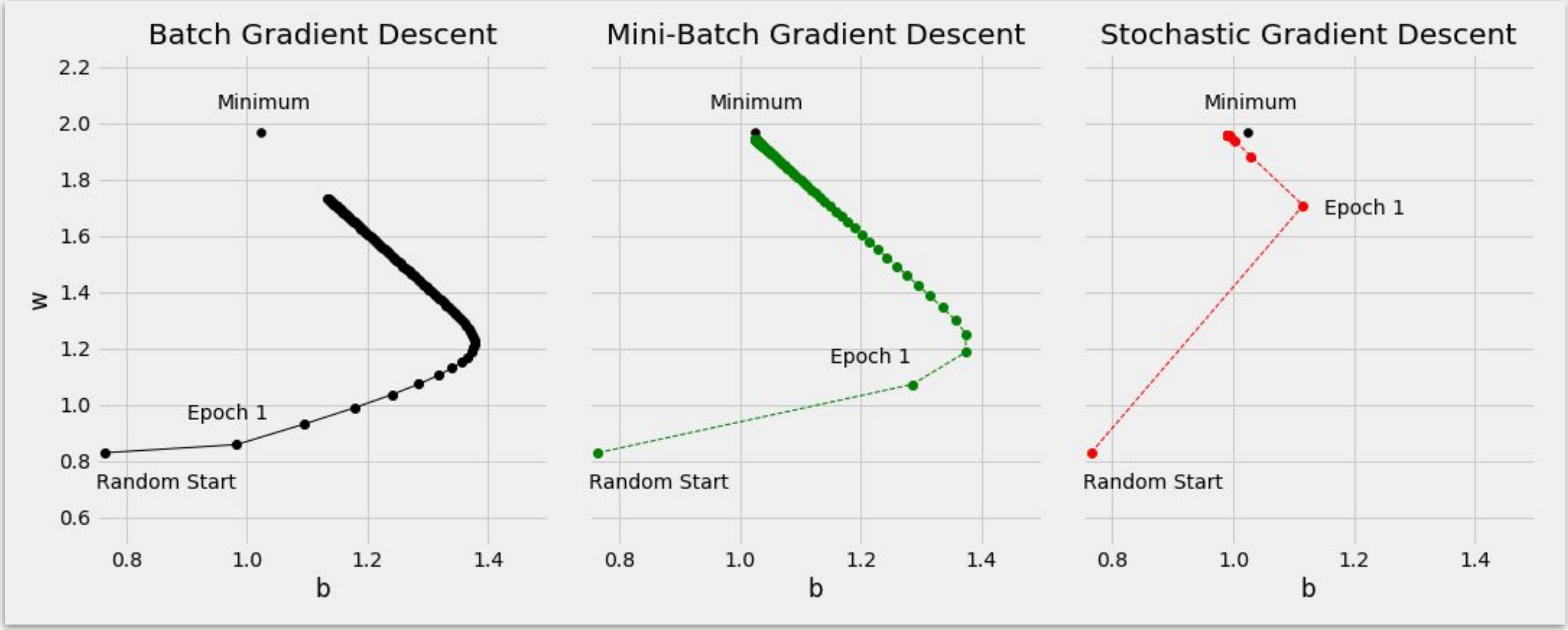
- Stochastic gradient descent

  One epoch means N updates, since every individual data point is used to perform an update.

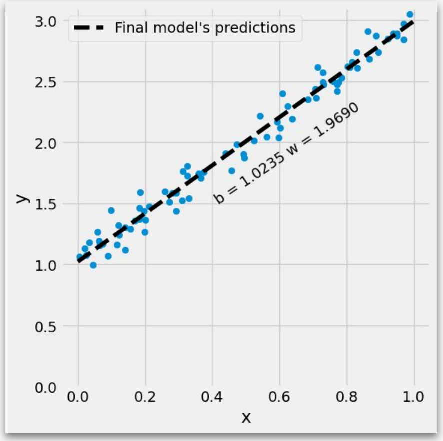
- Mini-batch gradient descent

  One epoch has N/n updates since a mini-batch of n data points is used to perform an update.

100 epochs using either 80 data points (batch), 16 data points (mini-batch), or a single data point (stochastic) for computing the loss, as shown in the figure below.



```python
# Step 0 - Initializes parameters
"b" and "w" randomly
np.random.seed(42)
b = np.random.randn(1)
w = np.random.randn(1)

print(b, w)

# Sets learning rate - this is
"eta" ~ the "n"-like Greek letter
lr = 0.1
# Defines number of epochs
n_epochs = 1000
```




```python
for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent. How wrong is our model? That's the error!
    error = (yhat - y_train)
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()

    # Step 3 - Computes gradients for both "b" and "w" parameters
    b_grad = 2 * error.mean()
    w_grad = 2 * (x_train * error).mean()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    b = b - lr * b_grad
    w = w - lr * w_grad

print(b, w)
[0.49671415] [-0.1382643]
[1.02354094] [1.96896411]
```

Tom Yeh (He/Him) • Following
Associate Professor of Computer Science at University of Colorado...
2d •

3. Linear Layer: 25 Exercises 🏋️
~ AI by Hand ✍️ Workbook ~

-- **Previous Workbooks** --

1: Dot Product
https://lnkd.in/g8TNrpfN

2: Matrix Multiplication
https://lnkd.in/g6DnVySK

-- **Workbook** --

Each workbook is comprised of 25 exercises.

Every exercise features a "missing" value, highlighted in a soft shade of red.

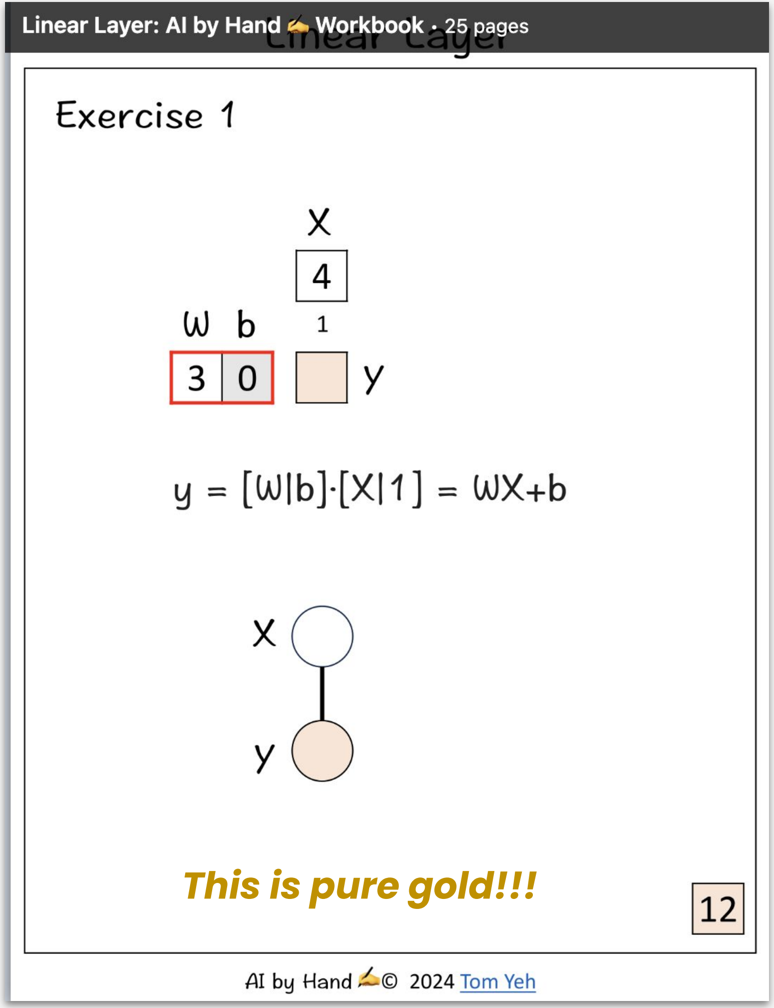
Can you calculate this missing value by hand? ✍️

For each exercise, the solution can be found in the bottom-right corner of the page. Do your best to avoid looking ahead. 👀

-- **Share** --

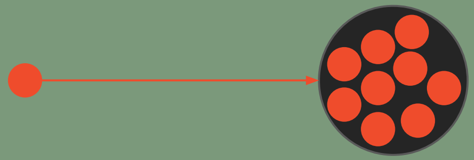
#linear #aibyhand #ai #neuralnetwork

[Repost ♻️] Please help me share this workbook with more people!



https://www.linkedin.com/feed/update/urn:li:activity:7178771842604707840

Time to TORCH it!



PyTorch

Pytorch Kickoff

Pytorch is more pythonic? TensorFlow excels in large-scale and production environments?

Fundamentals

Tensors, Loading Data, Devices and CUDA, Creating Parameters

Autograd

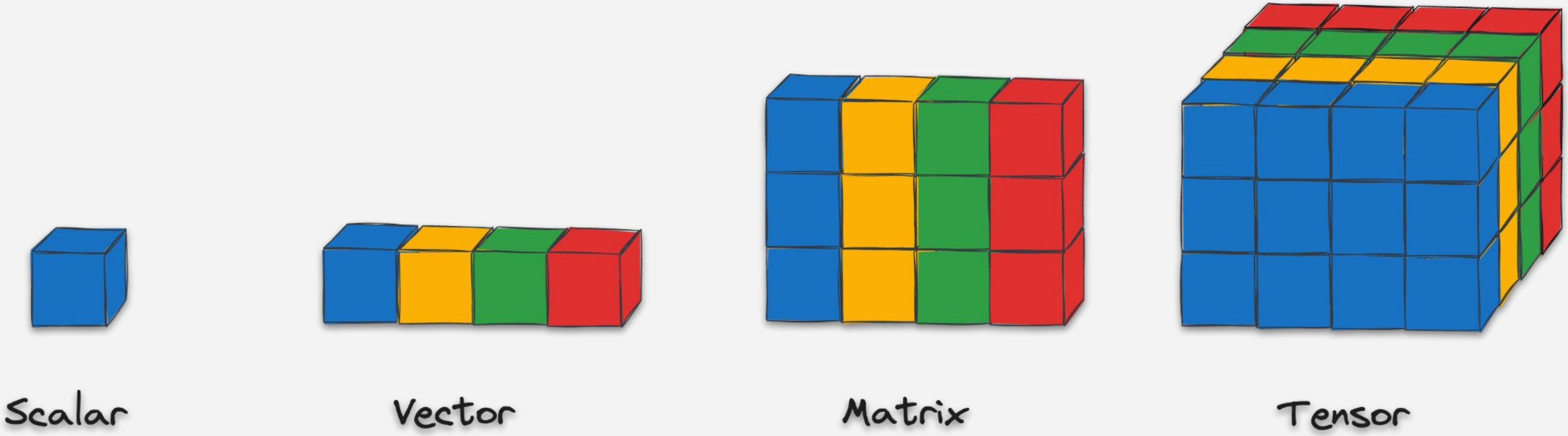
Backward, Update Parameters, Dynamic Computational Graph

Putting It All Together

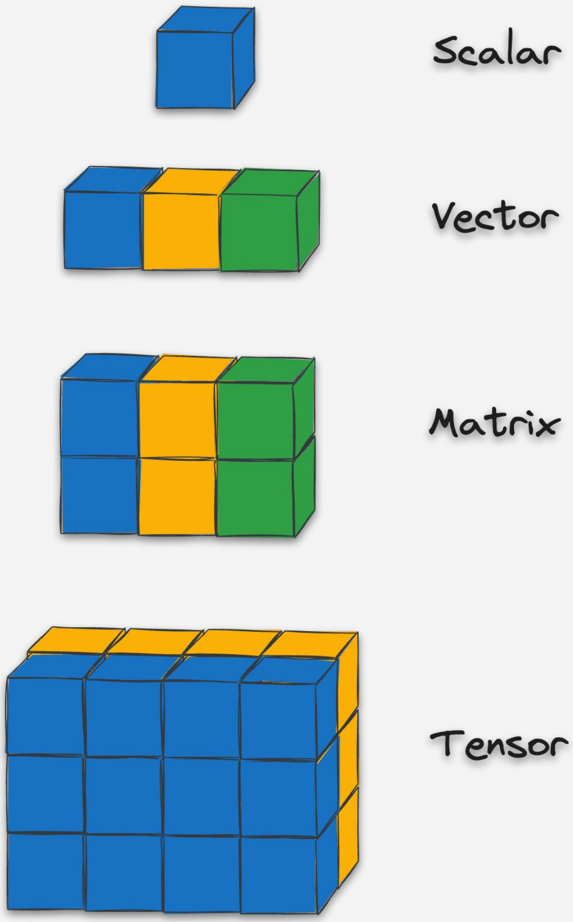
Optimizer, Loss, Models, Training

```python
import torch
import torch.optim as optim
import torch.nn as nn
from torchviz import make_dot
```

In the world of PyTorch, every data point, image, or signal takes shape as a tensor



```
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
tensor = torch.randn((2, 3, 4), dtype=torch.float)

print(scalar)
print(vector)
print(matrix)
print(tensor)


tensor(3.1416)
tensor([1, 2, 3])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[[ 0.9755, -0.8582, -1.5437,  0.8710],
         [ 2.2040,  0.2876,  1.4933, -1.6427],
         [ 0.6846,  0.2353, -0.9709, -1.3681]],

        [[ 4.4695,  0.0782, -0.6045,  1.0868],
         [ 1.1322, -0.4760,  0.6845, -0.4704],
         [ 1.5749, -0.3304, -0.7718,  0.6357]]])
```



```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

Loading Data
& Devices

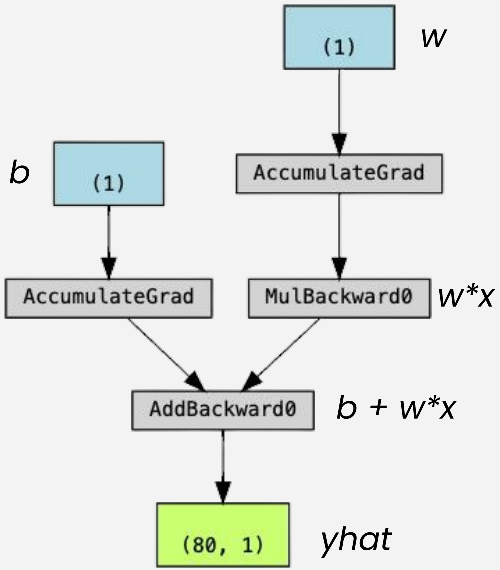
```
# We can specify the device at the moment of creation
# RECOMMENDED!

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
               dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
               dtype=torch.float, device=device)
print(b, w)

tensor([0.1940], device='cuda:0', requires_grad=True)
tensor([0.1391], device='cuda:0', requires_grad=True)
```

Best-Practices

```python
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train_tensor

# Step 2 - Computes the loss
# We are using ALL data points, so this is BATCH gradient
descent
# How wrong is our model? That's the error!
error = (yhat - y_train_tensor)
# It is a regression, so it computes mean squared error (MSE)
loss = (error ** 2).mean()

# Step 3 - Computes gradients for both "b" and "w" parameters
# No more manual computation of gradients!
# b_grad = 2 * error.mean()
# w_grad = 2 * (x_tensor * error).mean()
loss.backward()
```

Autograd

backward



Dynamic Computational Graph

Autograd

Updating Parameters

```python
# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Step 1 - Computes model's predicted output - forward pass
    yhat = b + w * x_train_tensor

    # Step 2 - Computes the loss
    # We are using ALL data points, so this is BATCH gradient
    # descent. How wrong is our model? That's the error!
    error = (yhat - y_train_tensor)
    # It is a regression, so it computes mean squared error (MSE)
    loss = (error ** 2).mean()

    # Step 3 - Computes gradients for both "b" and "w" parameters
    # No more manual computation of gradients!
    # b_grad = 2 * error.mean()
    # w_grad = 2 * (x_tensor * error).mean()
    # We just tell PyTorch to work its way BACKWARDS
    # from the specified loss!
    loss.backward()
```

```python
    # Step 4 - Updates parameters using gradients and
    # the learning rate.
    # We need to use NO_GRAD to keep the update out of
    # the gradient computation. Why is that? It boils
    # down to the DYNAMIC GRAPH that PyTorch uses...
    with torch.no_grad():
        b -= lr * b.grad
        w -= lr * w.grad

    # PyTorch is "clingy" to its computed gradients,
    # we need to tell it to let it go...
    b.grad.zero_()
    w.grad.zero_()
print(b, w)

tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

Optimizer and Loss

```python
# Sets learning rate - this is "eta" ~ the "n"-like
# Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w"
randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
               dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
               dtype=torch.float, device=device)

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([b, w], lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Defines number of epochs
n_epochs = 1000
```

```python
for epoch in range(n_epochs):
   # Step 1 - Computes model's predicted output - forward pass
   yhat = b + w * x_train_tensor

   # Step 2 - Computes the loss
   # No more manual loss!
   # error = (yhat - y_train_tensor)
   # loss = (error ** 2).mean()
   loss = loss_fn(yhat, y_train_tensor)

   # Step 3 - Computes gradients for both "b" and "w"
   loss.backward()

   # Step 4 - Updates parameters using gradients and
   # the learning rate
   optimizer.step()
   optimizer.zero_grad()

print(b, w)
tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

```python
class ManualLinearRegression(nn.Module):
   def __init__(self):
       super().__init__()
       # To make "b" and "w" real parameters of the model,
       # we need to wrap them with nn.Parameter
       self.b = nn.Parameter(torch.randn(1,
                                         requires_grad=True,
                                         dtype=torch.float))
       self.w = nn.Parameter(torch.randn(1,
                                         requires_grad=True,
                                         dtype=torch.float))

   def forward(self, x):
       # Computes the outputs / predictions
       return self.b + self.w * x
```

Models

A model is represented by a regular Python class that inherits from the Module class

Forward Pass

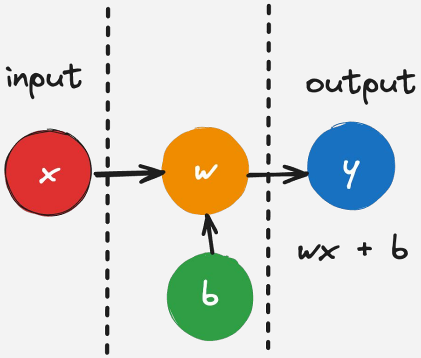
It is the moment when the model makes predictions

```
# Sets learning rate - this is "eta" ~ the "n"-like
# Greek letter
lr = 0.1

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
# Now we can create a model and send it at once to the
device
model = ManualLinearRegression().to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Defines number of epochs
n_epochs = 1000
```

```
for epoch in range(n_epochs):
   model.train() # What is this?!?

   # Step 1 - Computes model's predicted output - forward pass
   # No more manual prediction!
   yhat = model(x_train_tensor)

   # Step 2 - Computes the loss
   loss = loss_fn(yhat, y_train_tensor)

   # Step 3 - Computes gradients for both "b" and "w"parameters
   loss.backward()

   # Step 4 - Updates parameters using gradients and
   # the learning rate
   optimizer.step()
   optimizer.zero_grad()

# We can also inspect its parameters using its state_dict
print(model.state_dict())
OrderedDict([('b', tensor([1.0235], device='cuda:0')),
             ('w', tensor([1.9690], device='cuda:0'))])
```

Sequential Models

Our model was simple enough. You may be thinking: "Why even bother to build a class for it?!" Well, you have a point...




```
torch.manual_seed(42)
# Alternatively, you can use a Sequential model
model = nn.Sequential(nn.Linear(1, 1)).to(device)

model.state_dict()
OrderedDict([('0.weight',tensor([[0.7645]],device='cuda:0')),
             ('0.bias', tensor([0.8300], device='cuda:0'))])
```

```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

Data Preparation

```python
# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
```

Model Configuration

Training

```python
# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
   # Sets model to TRAIN mode
   model.train()

   # Step 1 - Computes model's predicted output - forward pass
   yhat = model(x_train_tensor)

   # Step 2 - Computes the loss
   loss = loss_fn(yhat, y_train_tensor)

   # Step 3 - Computes gradients for both "b" and "w" parameters
   loss.backward()

   # Step 4 - Updates parameters using gradients and
   # the learning rate
   optimizer.step()
   optimizer.zero_grad()

print(model.state_dict())
OrderedDict([('0.weight', tensor([[1.9690]], device='cuda:0')),
             ('0.bias', tensor([1.0235], device='cuda:0'))])
```