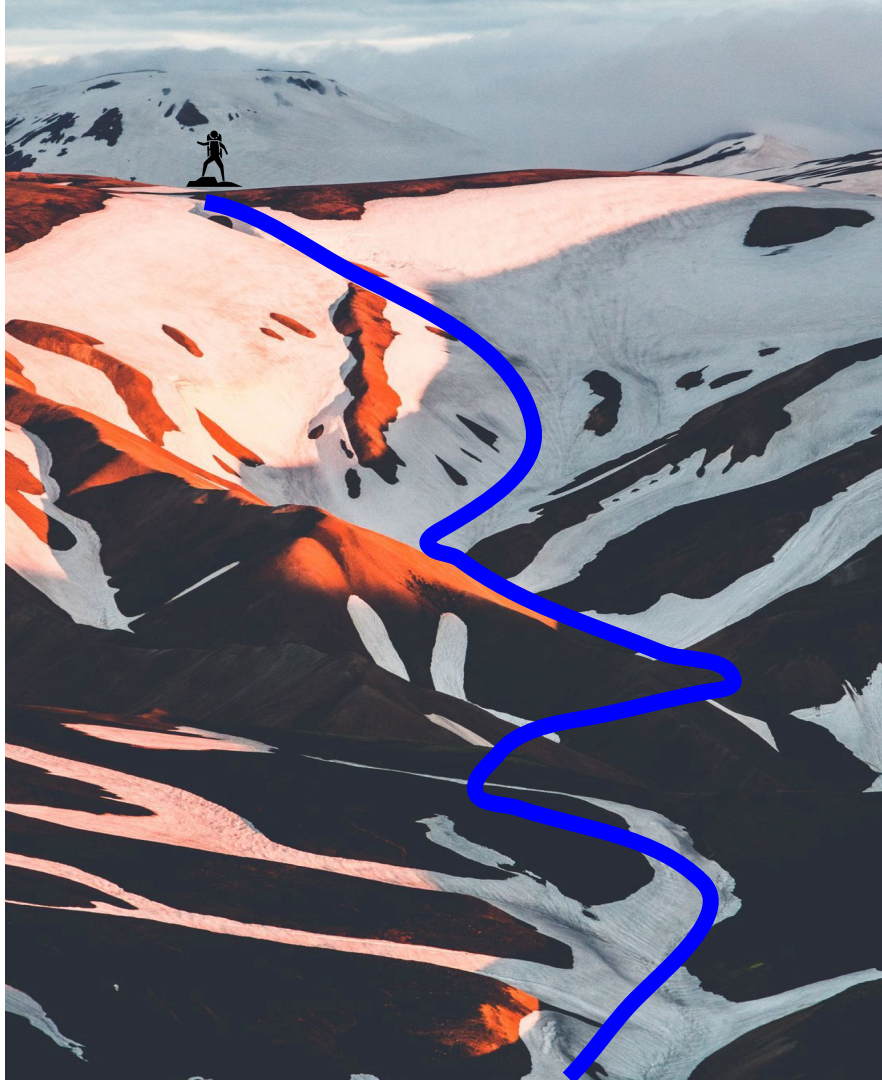
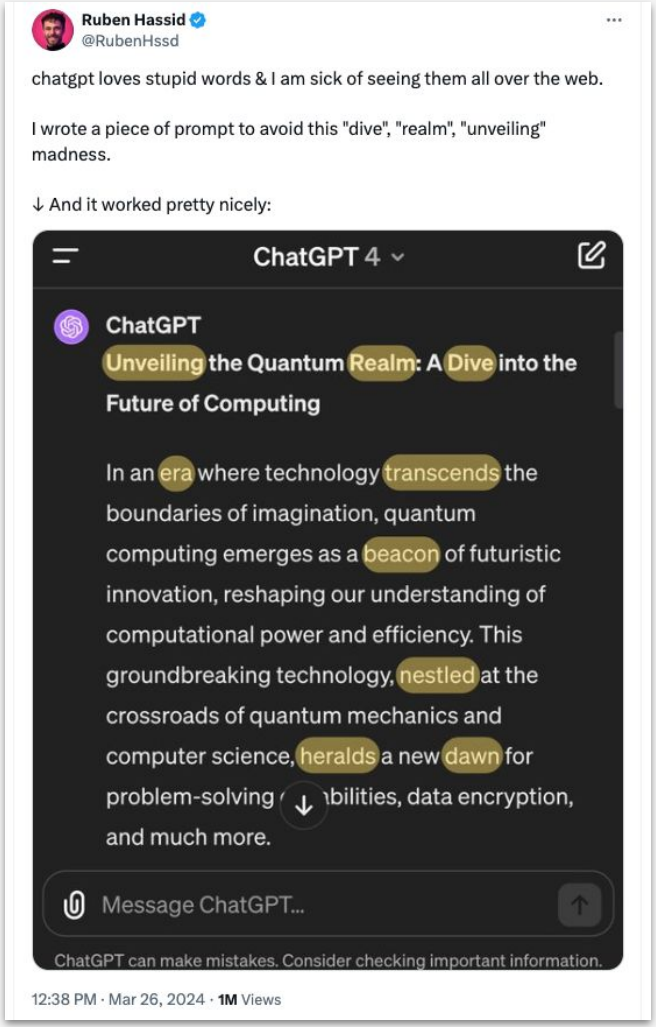
PPGEEC2318

Machine Learning

Visualizing Gradient Descent

Ivanovitch Silva
ivanovitch.silva@ufrn.br





Become an Efficient (or not) Academic Writer with AI

Mushtaq Bilal, PhD (He/Him) • Following
I simplify the process of academic writing | Helped 4,500+ become ...
3d •

How to make ChatGPT and Claude your academic writing assistant:
[This process involves no cheating and no plagiarism.]
Example prompts included

You can use any free or paid version of ChatGPT or Claude for this.

For the purposes of this post, I am using Claude 3 Opus.

1. Start by telling Claude/ChatGPT that you would like it act as your academic writing assistant.

Example prompt: "I would like you to act as my academic writing assistant. You don't have to write for me. I will write myself but I'd need your help. Does that make sense?"

Claude/ChatGPT will say it will help you with brainstorming, structuring, proofreading, etc.

2. Tell Claude/ChatGPT about your project and ask it to give you five suggestions on how to start your paper.

Example prompt: "I am working on a journal article on [your topic]. Give me five suggestion about how I should start my paper."

Read through the suggestions Claude/ChatGPT give you, pick the one you find most interesting and write a sentence.

Open a new document in MS Word and start writing. You just need to write a single single sentence.

Don't try to write a "perfect" sentence. Read the suggestion and start writing the first that comes to your mind.

3. Tell Claude/ChatGPT about the suggestion you liked and then paste sentence you wrote. Then ask it to five directions in which you can write the next sentence.

Tell Calude/ChatGPT not to give you any sentences of its own. If you use a sentence generated by Claude/ChatGPT, it will no longer remain your own work. It could be flagged as AI generated.

Example prompt:

"I followed your suggestion number 4 and wrote the following sentence:

[Sentence that you just wrote]

Please give me five directions in which I can write the next sentence. Do not give me any sentences."

Once again, Claude/ChatGPT will give you fives suggestions. Pick the one you find most interesting and write a sentence about it.

4. Repeat the above process.

Tell Claude/ChatGPT the suggestion you found usefull and paste the sentences you wrote. Then ask it to give you five directions in which to write the next sentences.

After repeating this process a few times you will have written a decent paragraph.

5. Once you have written a whole paragraph, you can ask Claude/ChatGPT to give you five suggestion on how to start the next paragraph.

Example prompt:

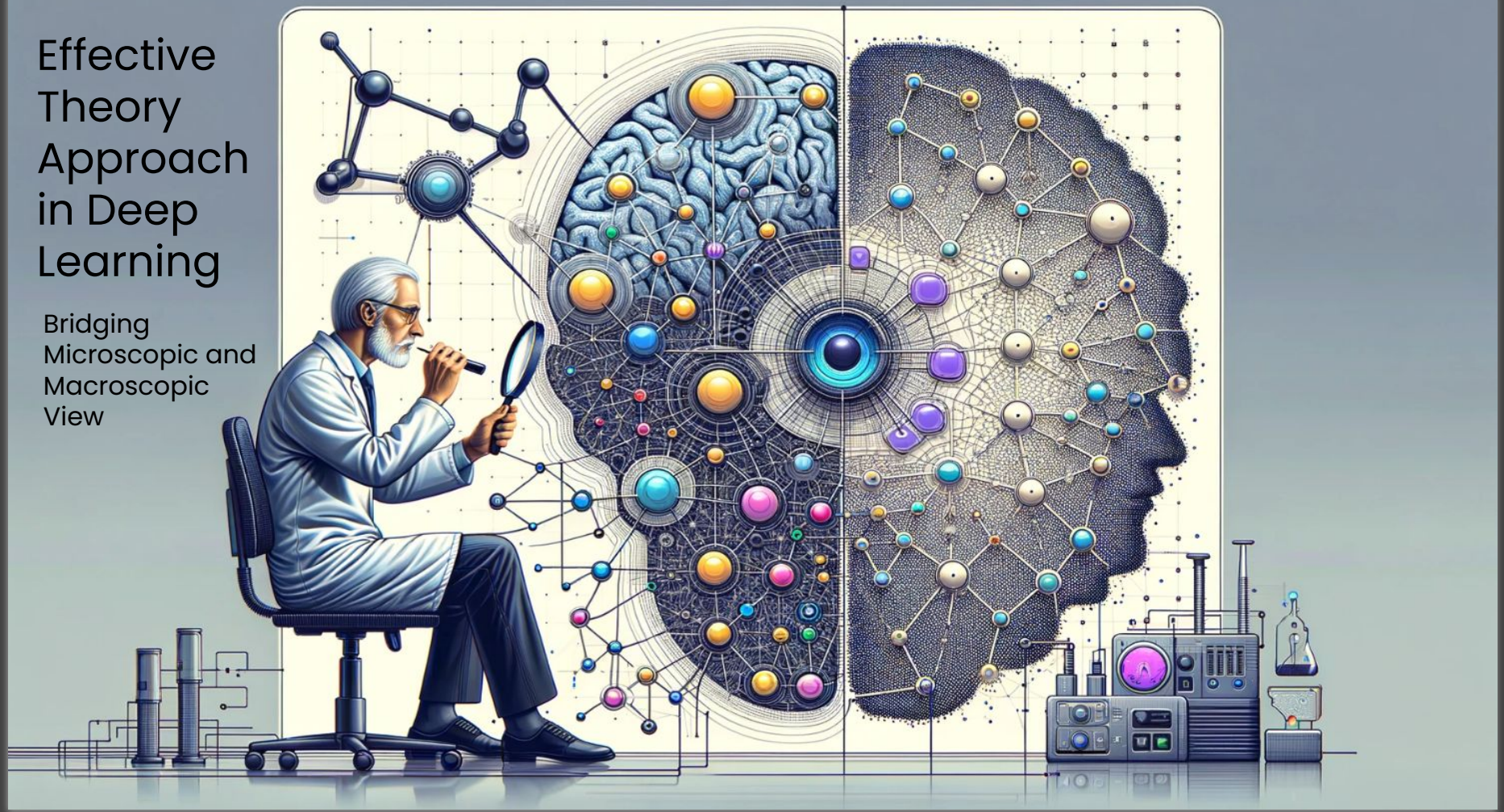
"I have followed your suggestions and now I have the following paragraph:

[paste your paragraph]

Please give me five directions in which I should start my next paragraph. Please keep in mind coherence, cohesion, and a smooth transition."

Effective Theory Approach in Deep Learning

Bridging Microscopic and Macroscopic View



Computer Science > Machine Learning

[Submitted on 18 Jun 2021 (v1), last revised 24 Aug 2021 (this version, v2)]

The Principles of Deep Learning Theory

Daniel A. Roberts, Sho Yaida, Boris Hanin

This book develops an effective theory approach to understanding deep neural networks of practical relevance. Beginning from a first-principles component-level picture of networks, we explain how to determine an accurate description of the output of trained networks by solving layer-to-layer iteration equations and nonlinear learning dynamics. A main result is that the predictions of networks are described by nearly-Gaussian distributions, with the depth-to-width aspect ratio of the network controlling the deviations from the infinite-width Gaussian description. We explain how these effectively-deep networks learn nontrivial representations from training and more broadly analyze the mechanism of representation learning for nonlinear models. From a nearly-kernel-methods perspective, we find that the dependence of such models' predictions on the underlying learning algorithm can be expressed in a simple and universal way. To obtain these results, we develop the notion of representation group flow (RG flow) to characterize the propagation of signals through the network. By tuning networks to criticality, we give a practical solution to the exploding and vanishing gradient problem. We further explain how RG flow leads to near-universal behavior and lets us categorize networks built from different activation functions into universality classes. Altogether, we show that the depth-to-width ratio governs the effective model complexity of the ensemble of trained networks. By using information-theoretic techniques, we estimate the optimal aspect ratio at which we expect the network to be practically most useful and show how residual connections can be used to push this scale to arbitrary depths. With these tools, we can learn in detail about the inductive bias of architectures, hyperparameters, and optimizers.

Comments:	471 pages, to be published by Cambridge University Press; v2: hyperlinks fixed, index added
Subjects:	**Machine Learning (cs.LG)**; Artificial Intelligence (cs.AI); High Energy Physics - Theory (hep-th); Machine Learning (stat.ML)
Report number:	MIT-CTP/5306
Cite as:	arXiv:2106.10165 **[cs.LG]**
	(or arXiv:2106.10165v2 **[cs.LG]** for this version)
	https://doi.org/10.48550/arXiv.2106.10165
Journal reference:	Cambridge University Press (2022)
Related DOI:	https://doi.org/10.1017/9781009023405

Submission history

From: Sho Yaida [view email]
[v1] Fri, 18 Jun 2021 15:00:00 UTC (706 KB)
[v2] Tue, 24 Aug 2021 17:12:56 UTC (718 KB)

Effective Theory Approach in Deep Learning

Bridging Microscopic and Macroscopic Views

Introduction to Effective Theory

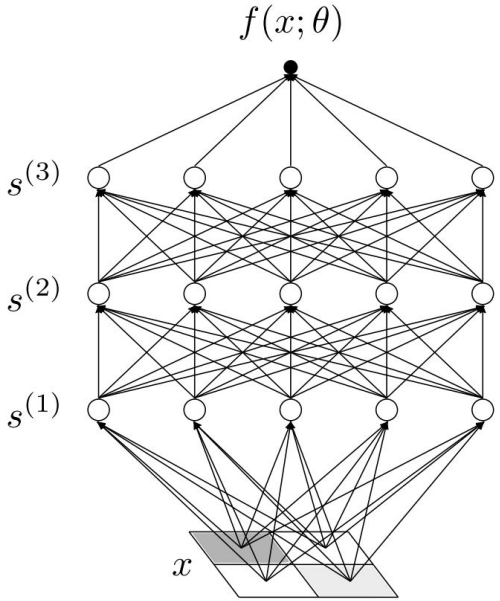
Inspired by theoretical physics, the effective theory approach seeks to simplify complex systems by focusing on macroscopic behaviors emergent from microscopic interactions.

Application in Deep Learning

In the context of deep learning, this approach aims to explain how simple operations in neural networks (microscopic level) lead to complex and intelligent behaviors (macroscopic level).

Significance

This method provides a promising route to unravel the intricacies of deep neural networks, making their analysis more tractable and comprehensible.



- ## Neural Networks as Functions

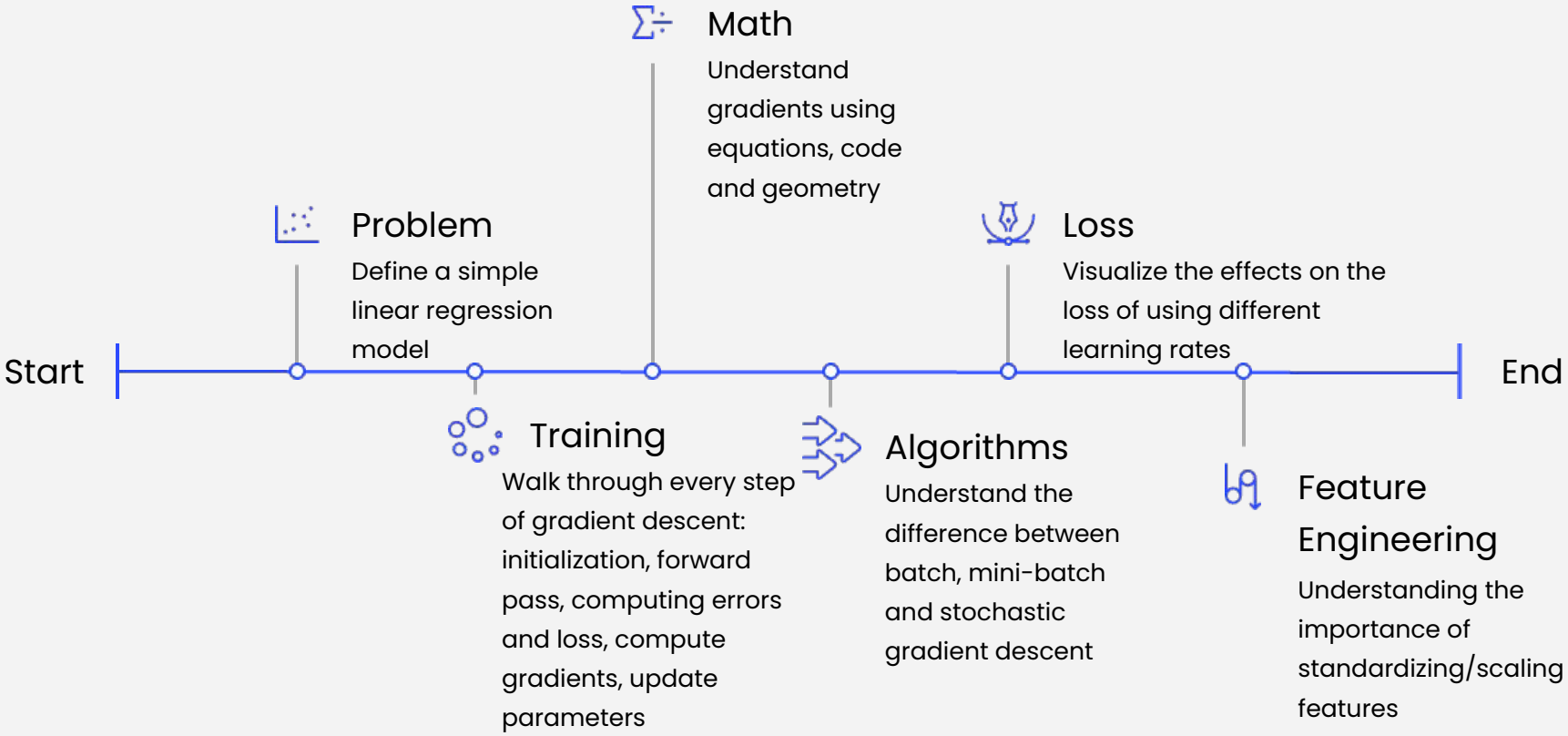
  Neural networks are viewed as a set of **parameters** (denoted by θ) and **functions** f(x; θ) where 'x' is the **input,** and 'θ' is a high-dimensional parameter vector. To make the network function useful, θ must be **finely tuned** to approximate a desired target function as closely as possible.

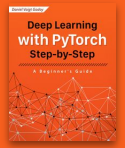
- ## Initialization and Training

  The tuning of θ happens in two major steps: a) **initialization**; where parameters are initially set by sampling from a simple probability distribution, setting the stage for learning. b) **training**; Adjustments are made to the parameters ($\theta \rightarrow \theta^*$), refining the network's function **f(x; θ*)** to approximate the target function from the training data.

- ## Goals of understanding

  The aim is to grasp the **macroscopic behavior** of the network based on its **microscopic components** (trained parameters θ*), and to understand how the network uses training data in its function approximation.





Visualizing Gradient Descent

dvgodoy / PyTorchStepByStep Public

Notifications | Fork 257 | Star 686

Code | Issues 11 | Pull requests | Actions | Projects | Security | Insights

master | 12 Branches | 11 Tags | Go to file | Code

dvgodoy fixing typo — a6fdc96 · last week — 141 Commits

data_generation	fixing type	last month
data_preparation	preview 2.1	4 years ago
images	readme	2 years ago
model_configuration	revision	3 years ago
model_training	revision	3 years ago
plots	fixing warnings	last month
runs	runs folder	4 years ago
stepbystep	ion	3 years ago
.gitignore	runs folder	4 years ago
Chapter00.ipynb	revision	3 years ago
Chapter01.ipynb	revision	2 years ago
Chapter02.1.ipynb	revision	3 years ago
Chapter02.ipynb	revision	2 years 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

686 stars

11 watching

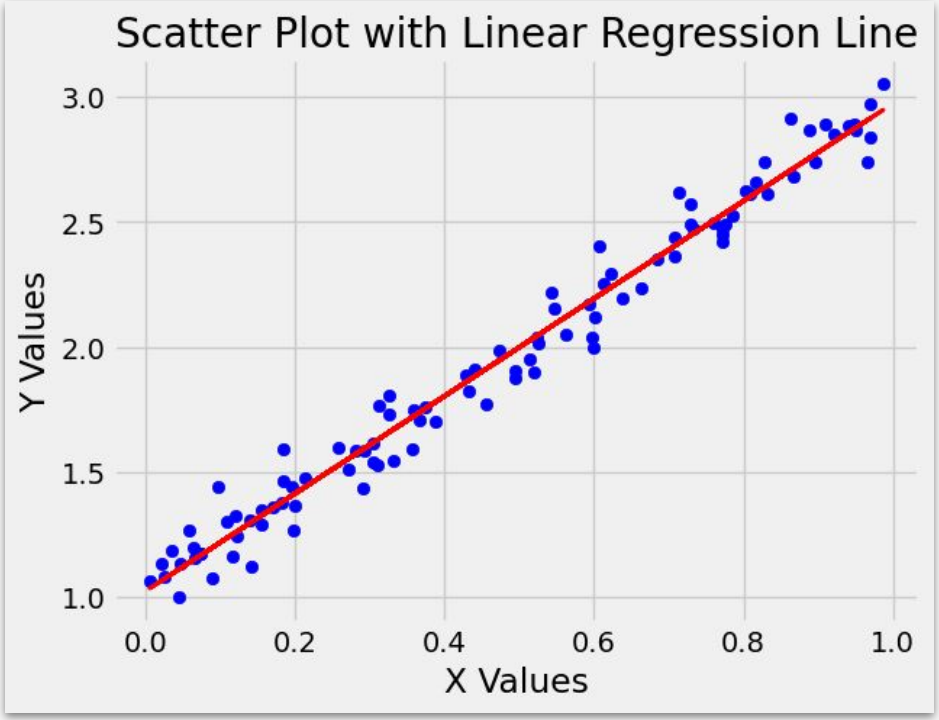
257 forks

Report repository

Releases

11 tags

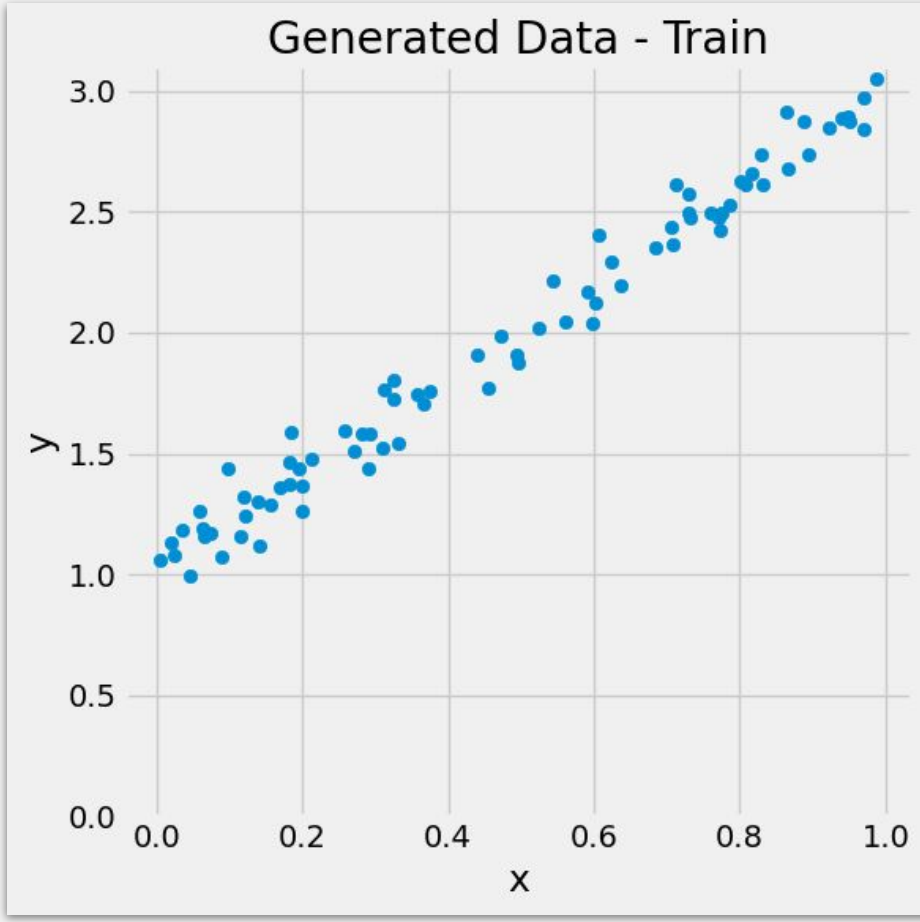
Packages

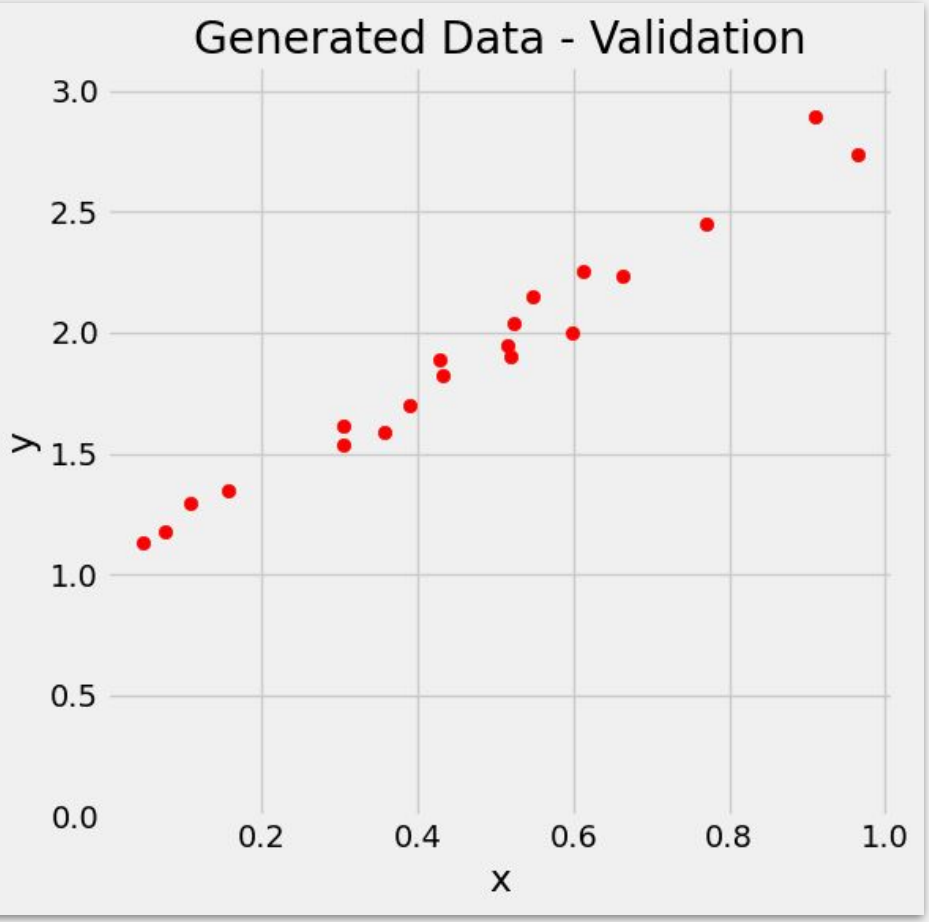


A linear regression model

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

Synthetic Data Generation and Train-Validation-Test Split





00

Step

Initializes parameters

$$y = b + wx$$

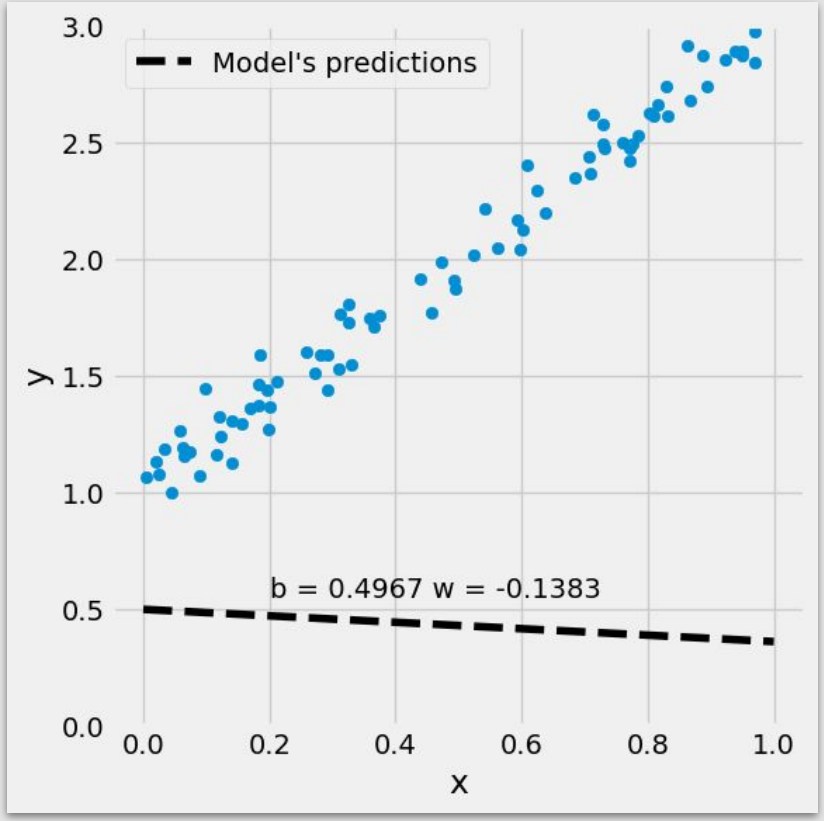
```
# Step 0 - Initializes parameters "b" and "w" randomly

# 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
```



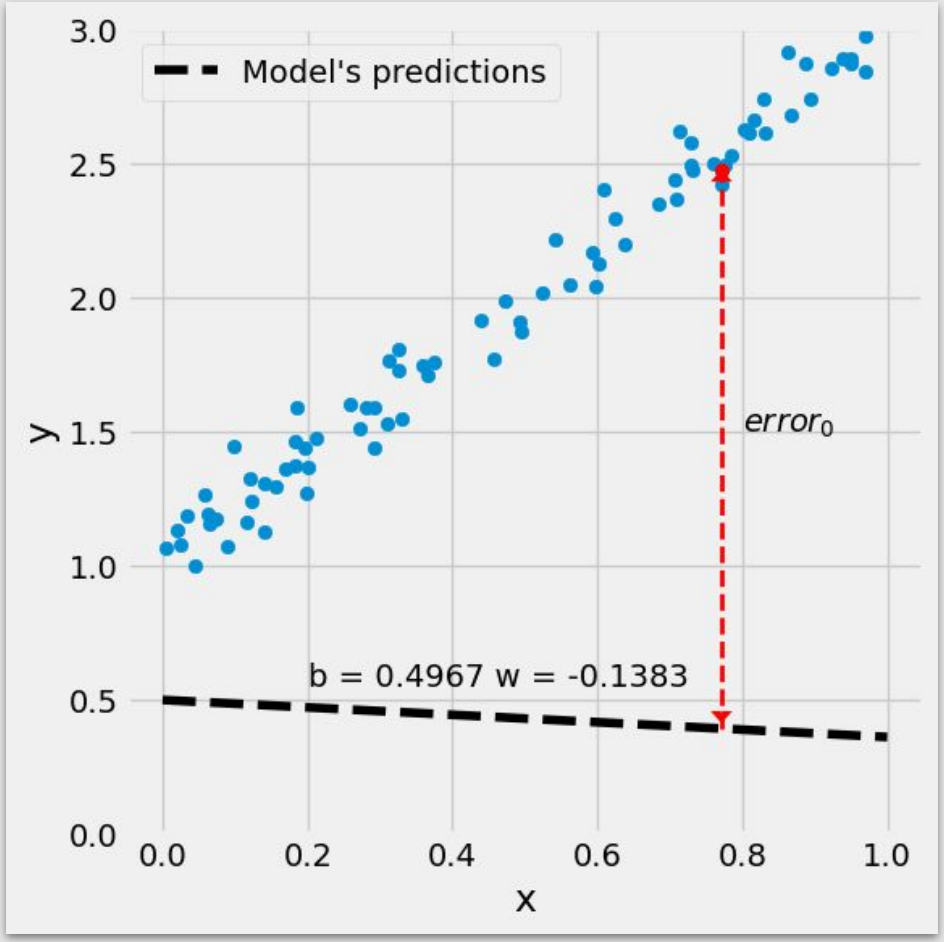
02

Step

Compute the loss

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

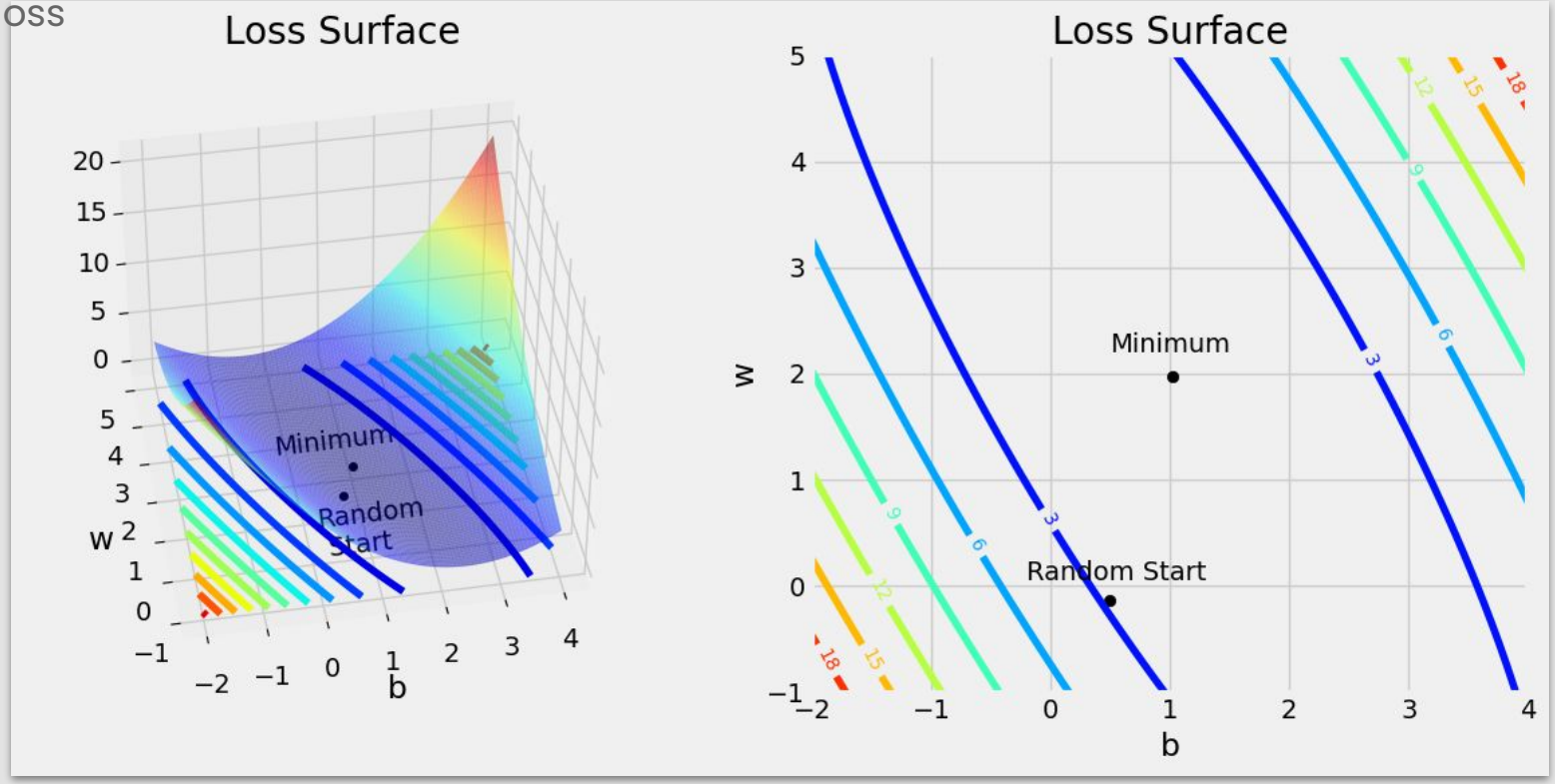
```
# Step 2 - Computing the loss
# We are using ALL data points, so this is BATCH gradient
# descent. How wrong is our model? That's the error!
error = (yhat - y_train)

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

2.7421577700550976
```



02

Step

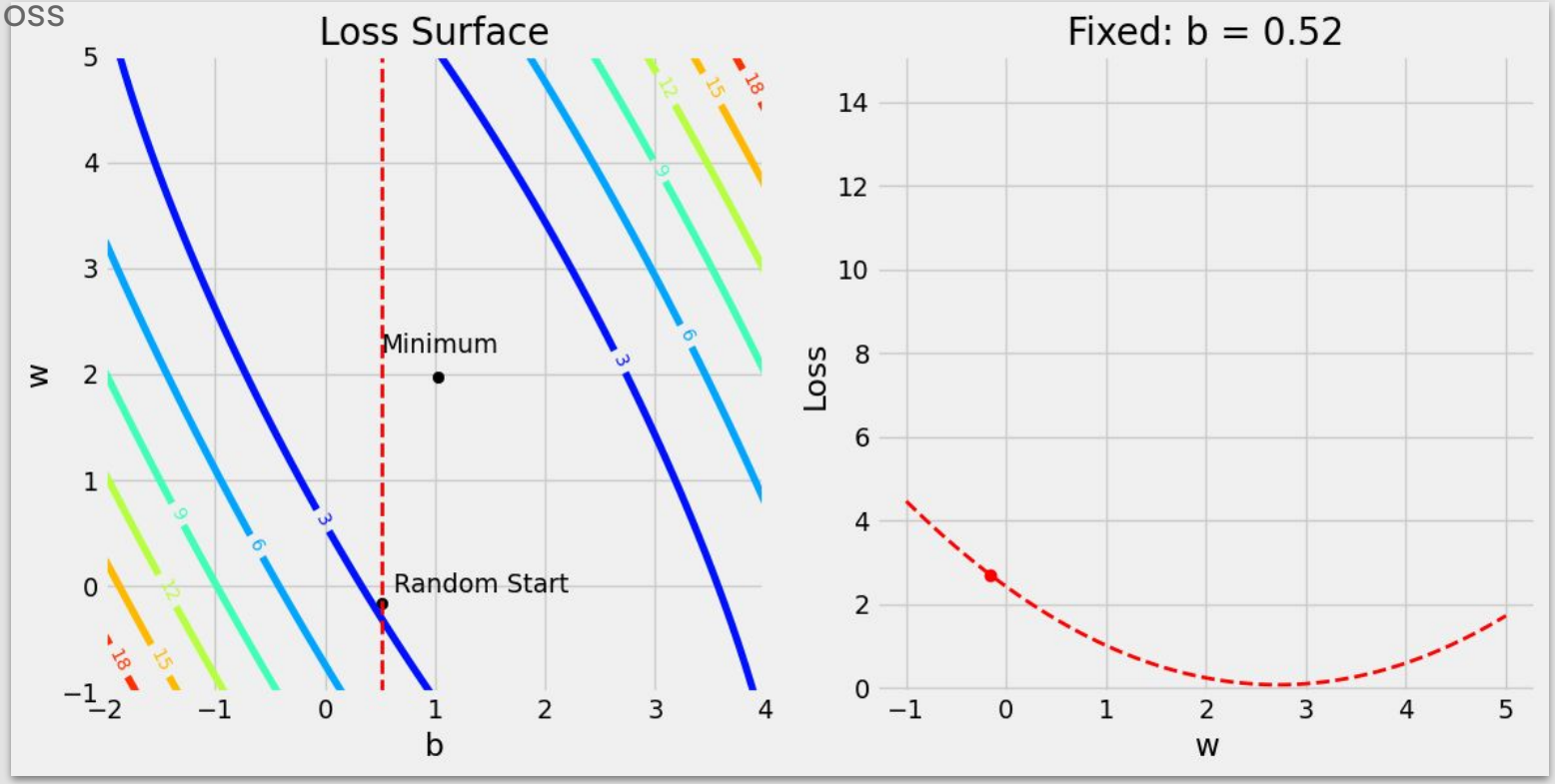
Compute the loss



02

Step

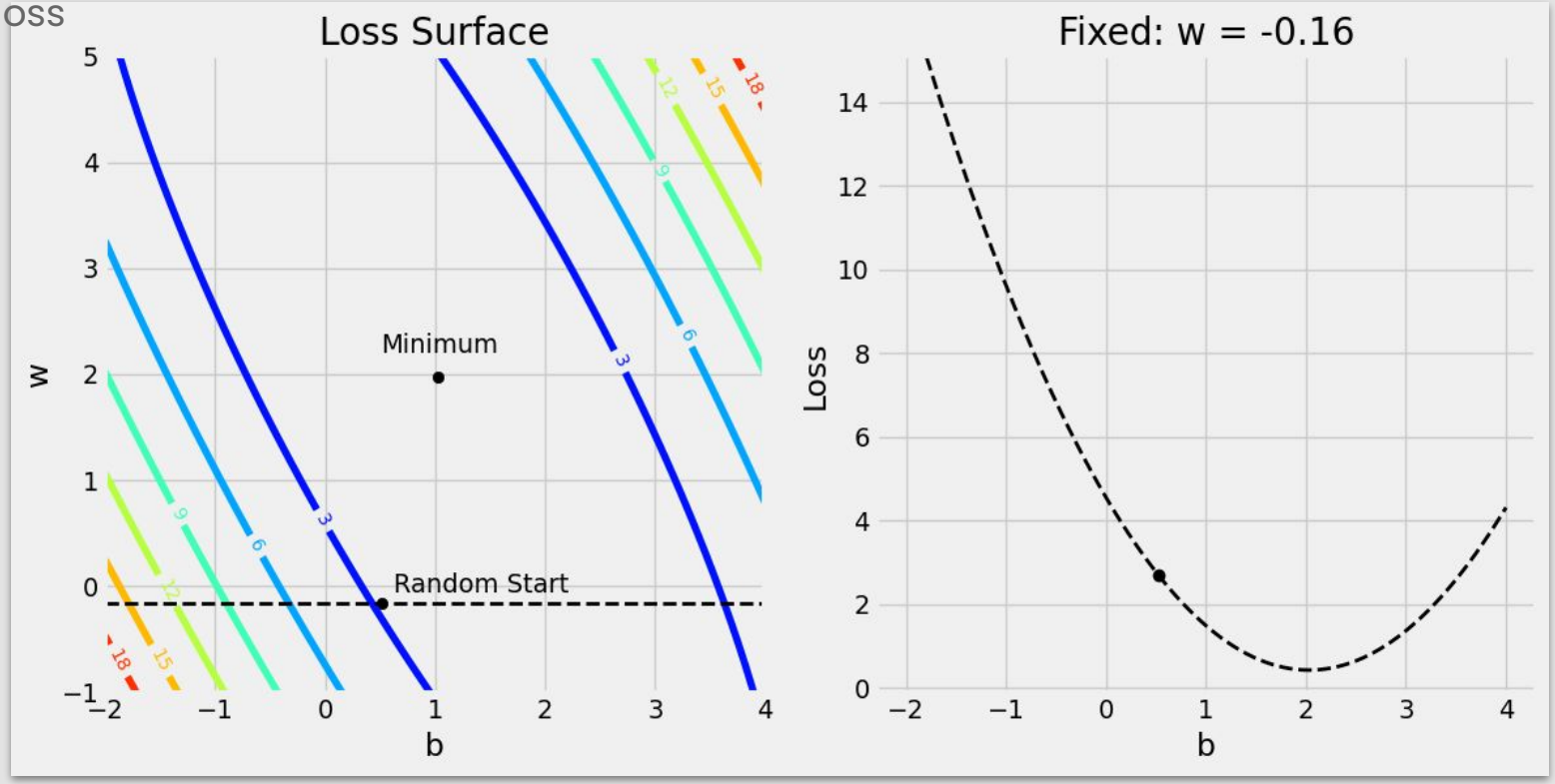
Compute the loss



02

Step

Compute the loss

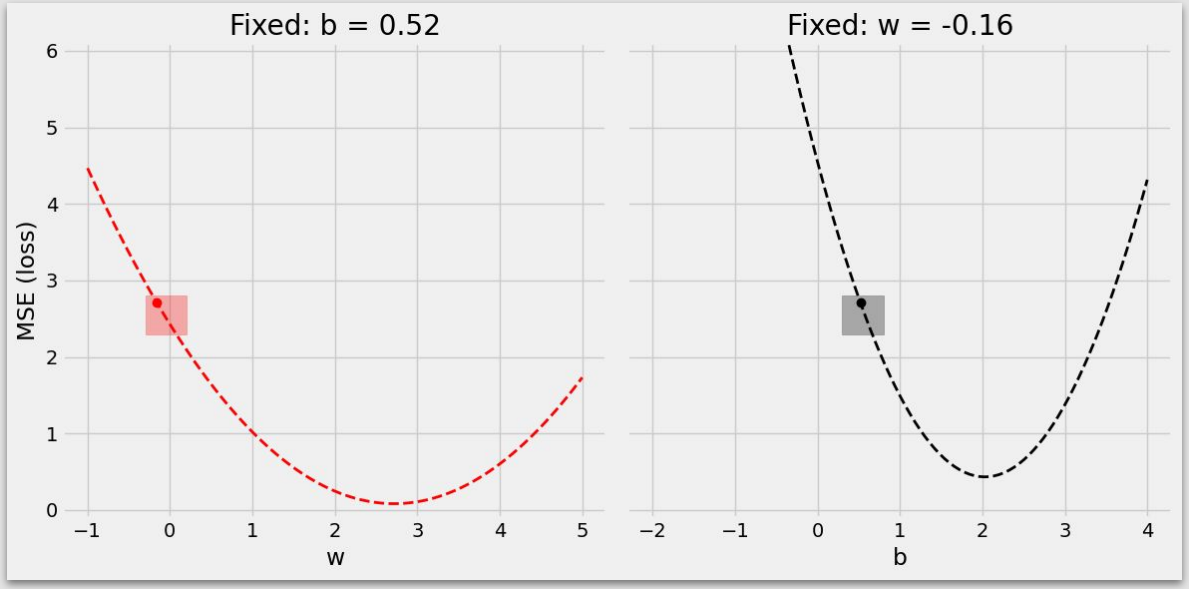


03

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

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

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

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

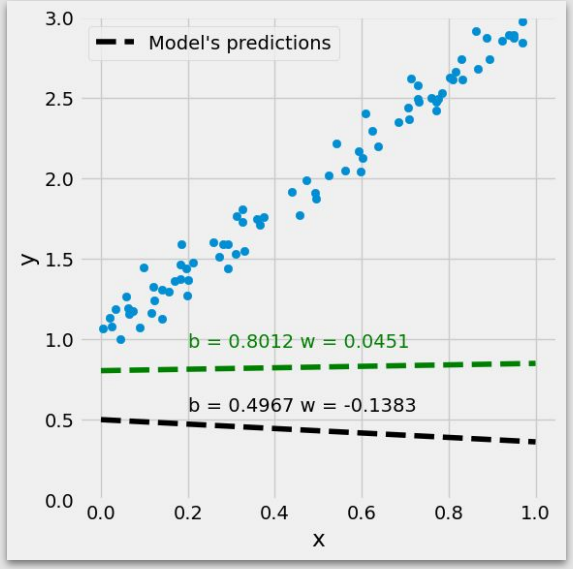
```
# Step 3 - Computes gradients for both "b" and "w" parameters
b_grad = 2 * error.mean()
w_grad = 2 * (x_train * error).mean()
print(b_grad, w_grad)

-3.044811379650508 -1.8337537171510832
```

04

Step

Update the parameters

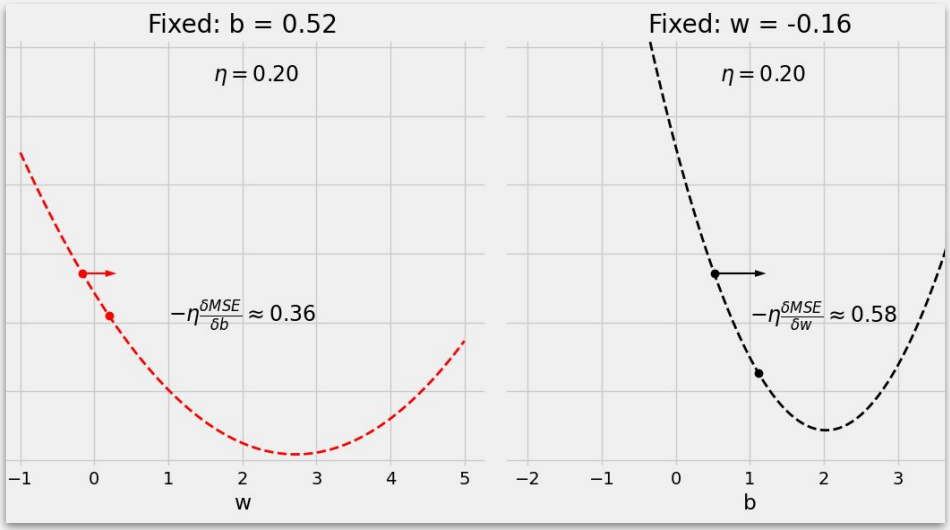
```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1
print(b, w)

# 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]
```

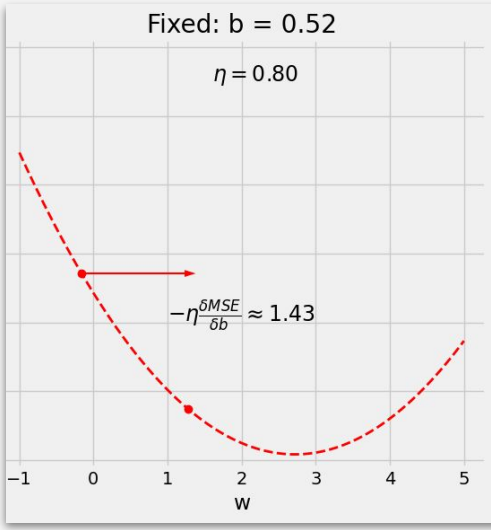


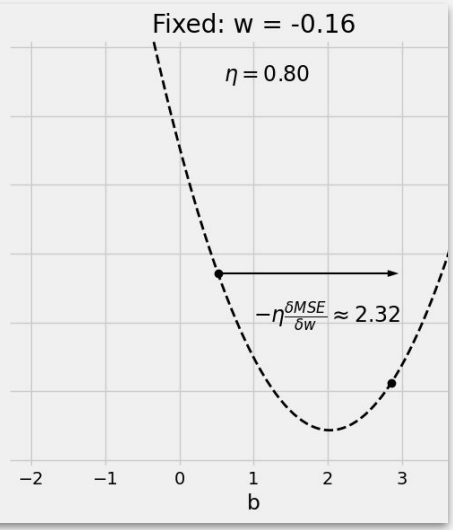
$$b = b - \eta \frac{\partial \text{MSE}}{\partial b}$$

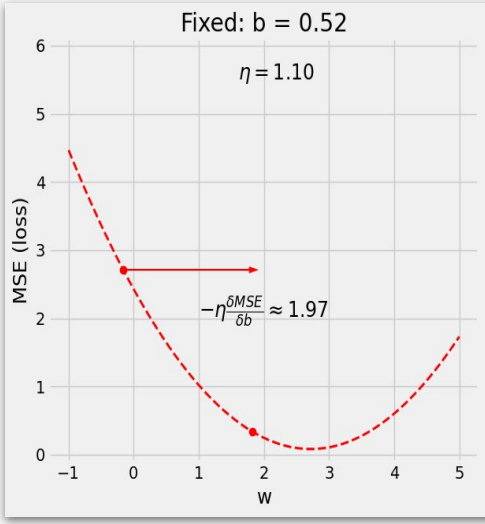
$$w = w - \eta \frac{\partial \text{MSE}}{\partial w}$$

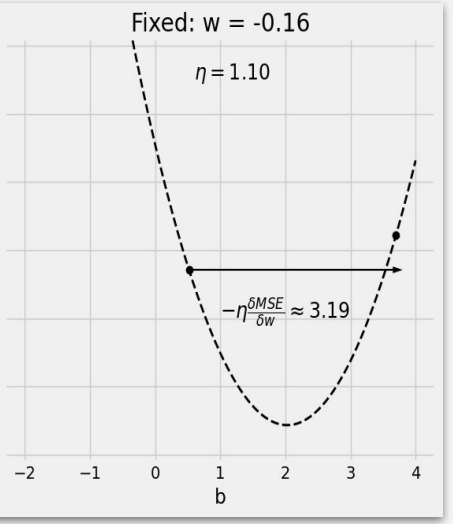
Learning Rate



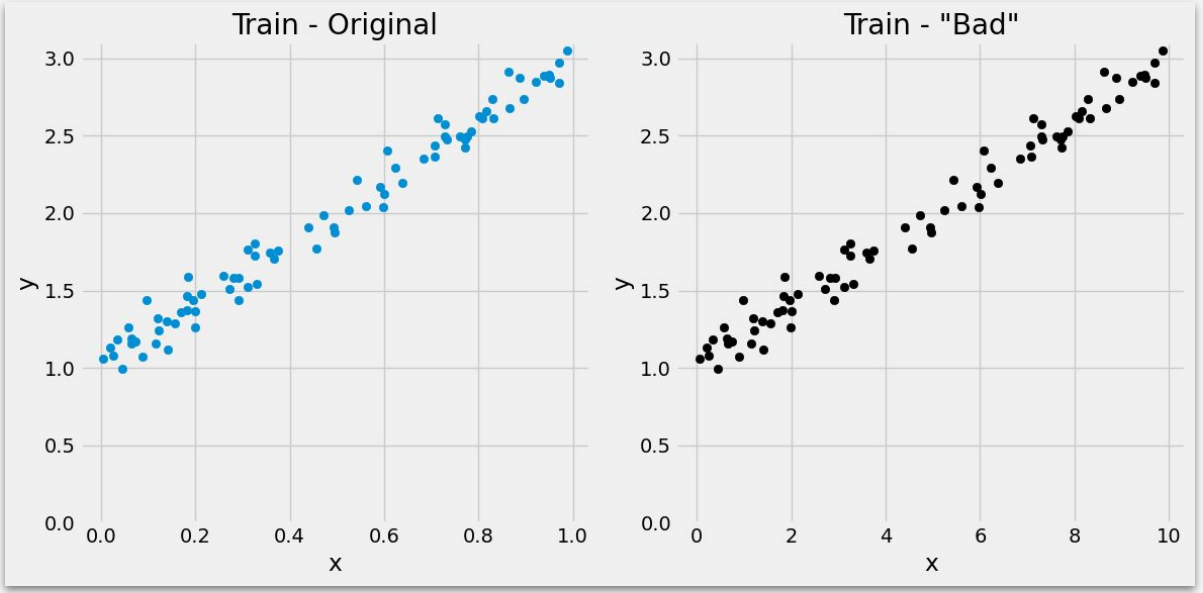


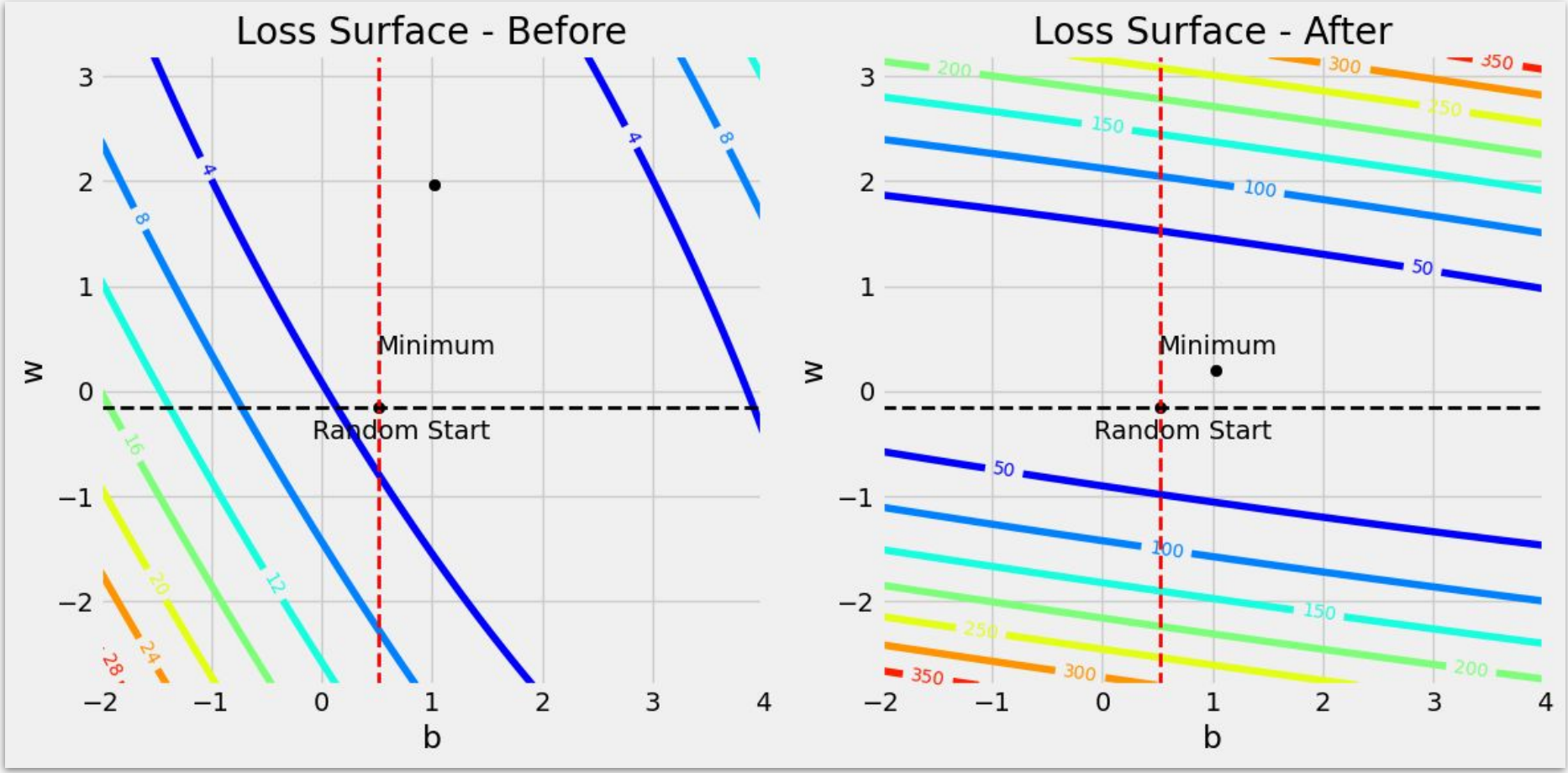


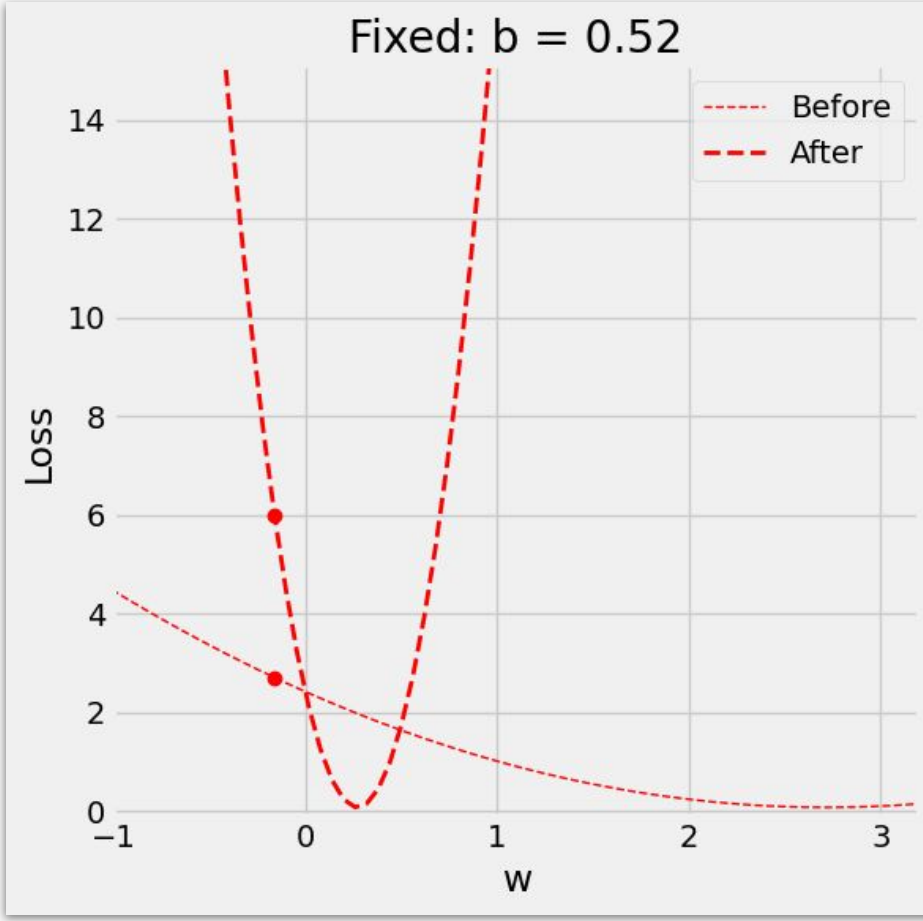


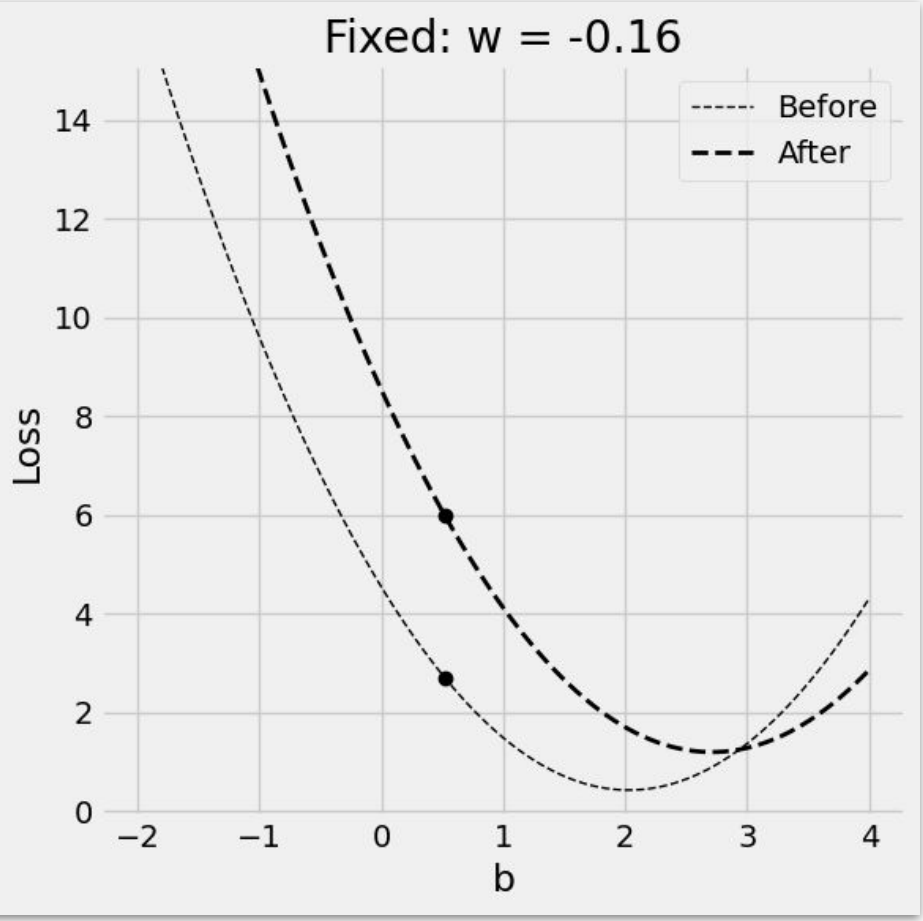


What is the influence of the X scale on the results?




Loss Surface - Before
Minimum
Random Start
w
b
Loss Surface - After
Minimum
Random Start
w
b





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

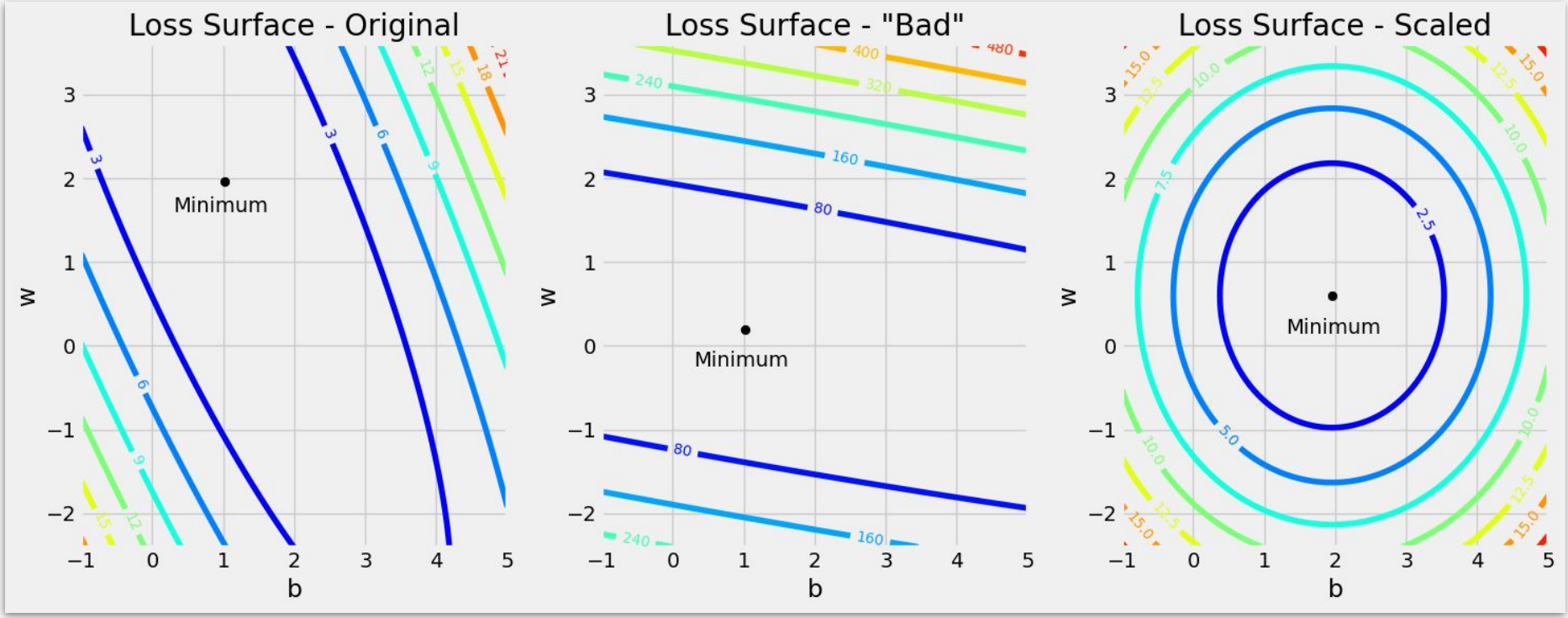
How can we fix it?

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

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

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


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

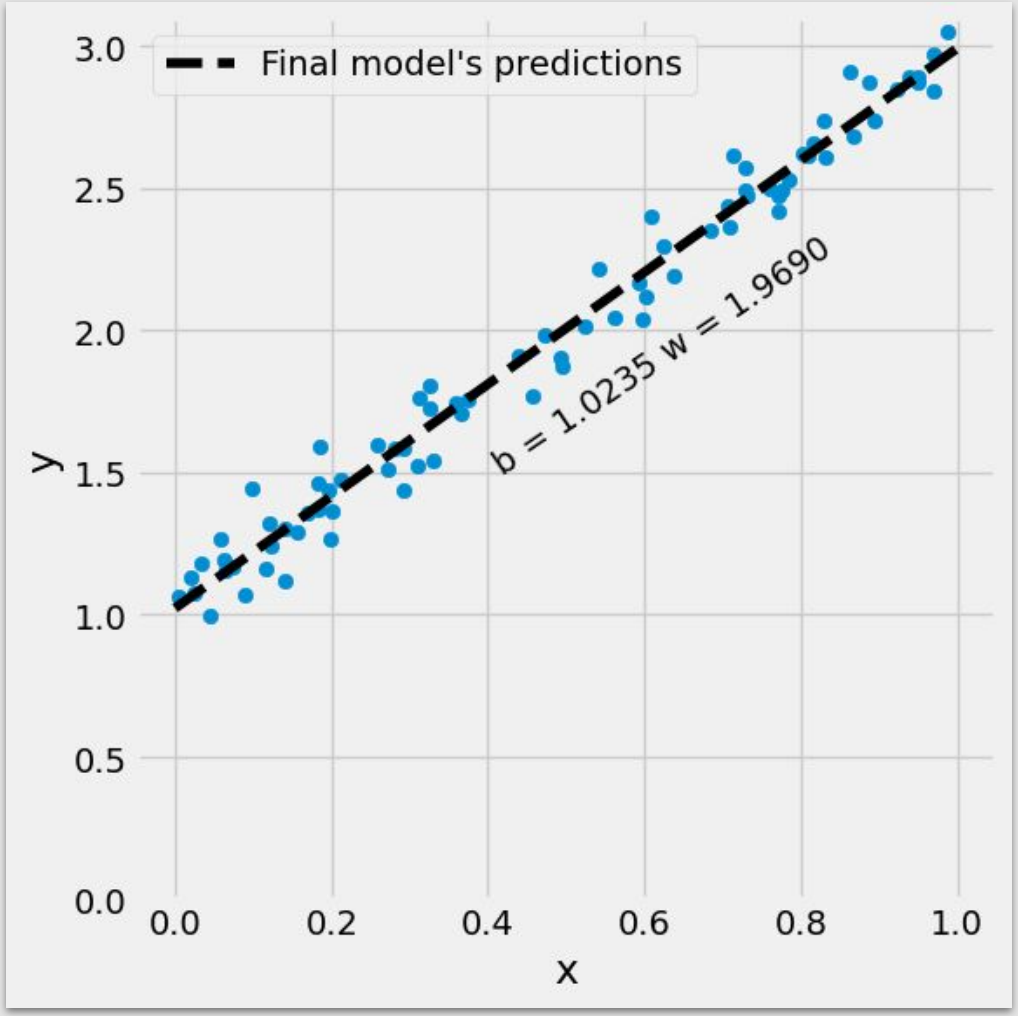

Loss Surface - Original
Minimum
w
b
Loss Surface - "Bad"
Minimum
w
b
Loss Surface - Scaled
Minimum
w
b

05

Step

Iterate and Converge

Now we use the updated parameters to go back to Step 1 and restart 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.

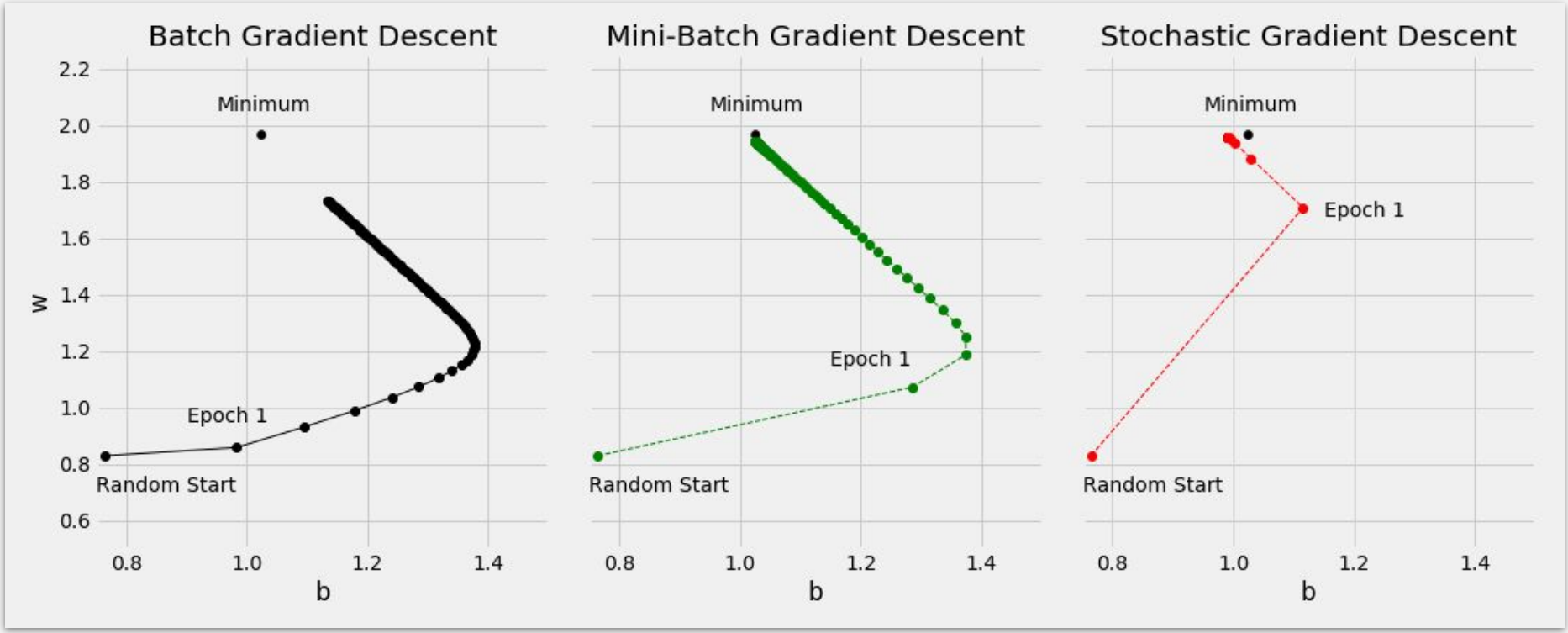
- Stochastic gradient descent

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

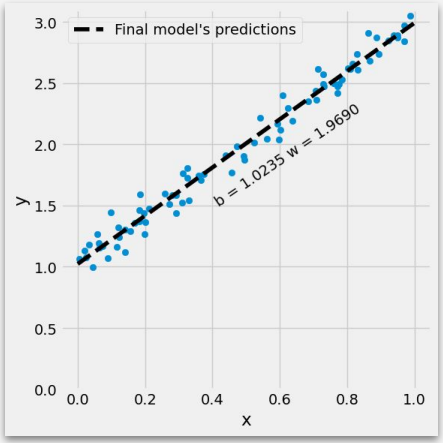
- Mini-batch gradient descent

  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.

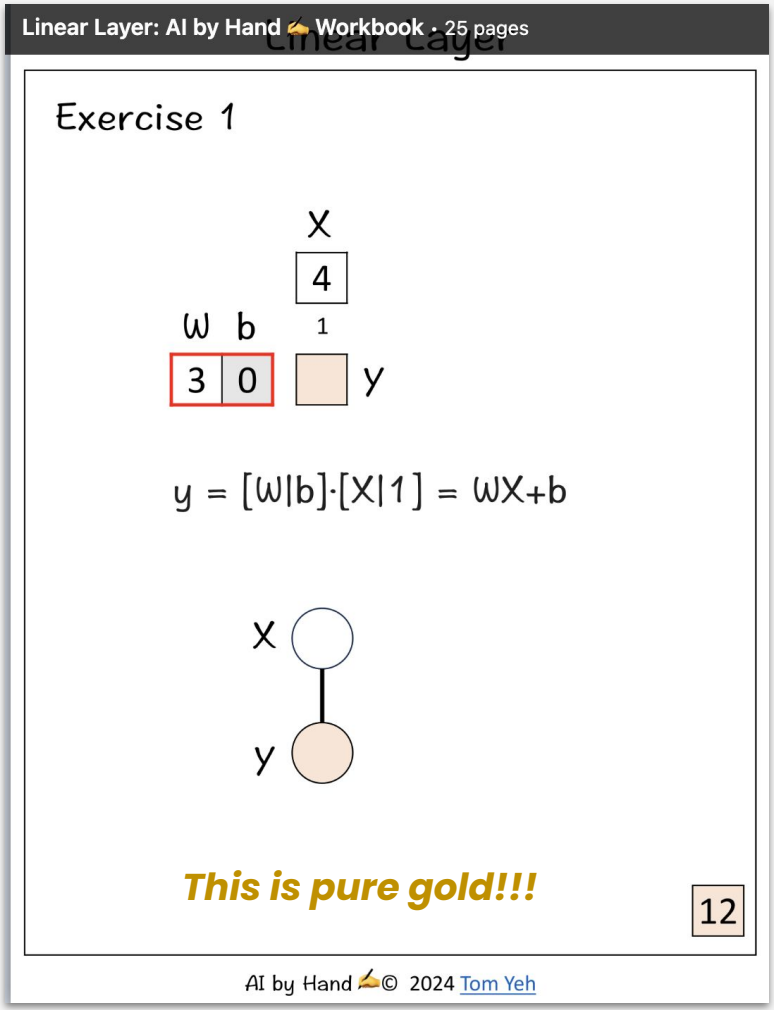
Can you calculate this missing value by hand? ✍️

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

-- **Share** --

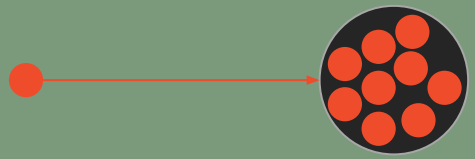
#linear #aibyhand #ai #neuralnetwork

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



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

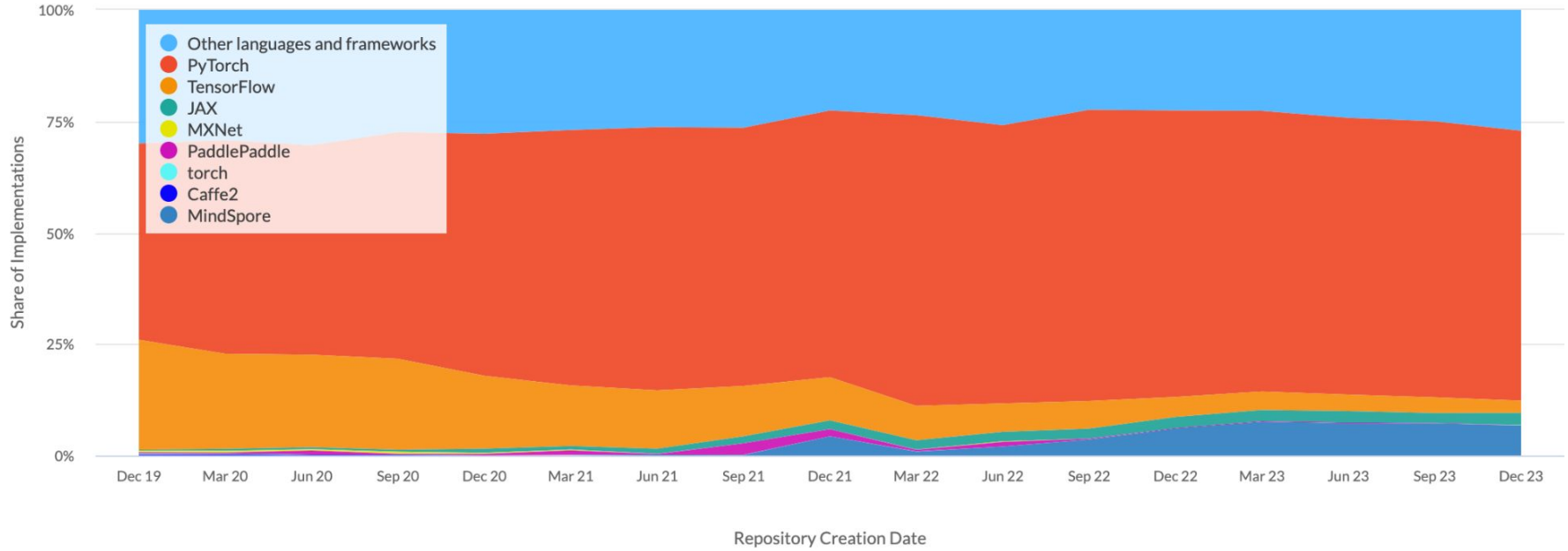
Time to TORCH it!



PyTorch

Frameworks

Paper Implementations grouped by framework



https://paperswithcode.com/trends

Pytorch Kickoff

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

Fundamentals

Tensors, Loading Data, Devices and CUDA, Creating Parameters

Autograd

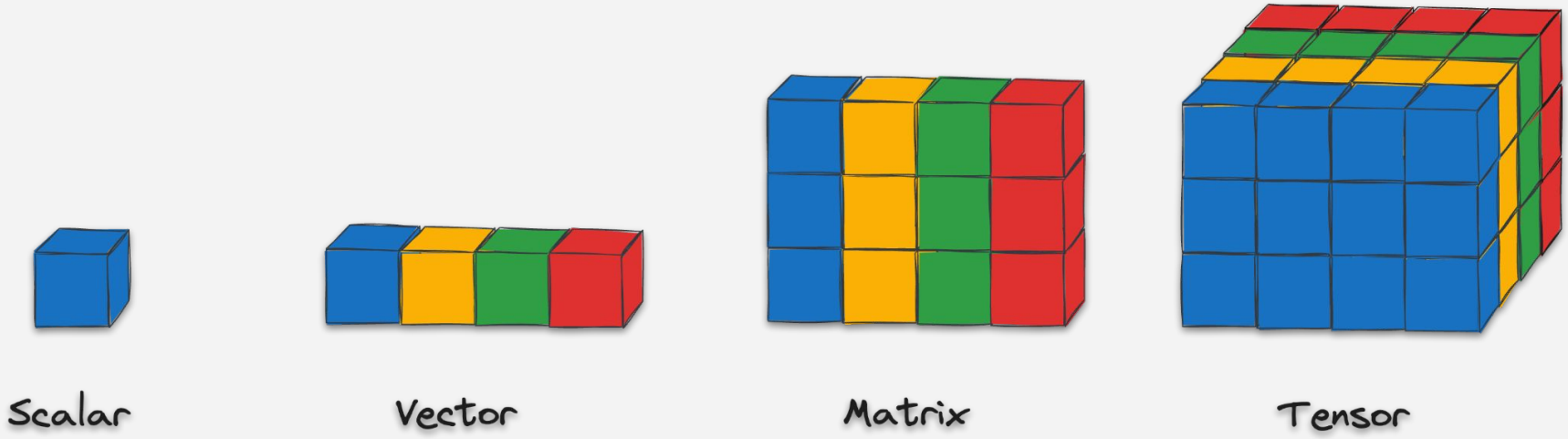
Backward, Update Parameters, Dynamic Computational Graph

Putting It All Together

Optimizer, Loss, Models, Training

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

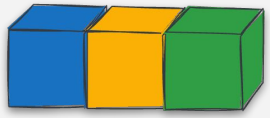
In the world of PyTorch, every data point, image, or signal takes shape as a tensor



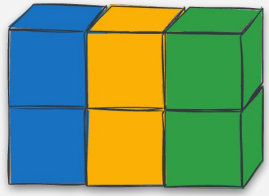
```python
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
tensor = torch.randn((2, 3, 4), dtype=torch.float)

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]]])
```

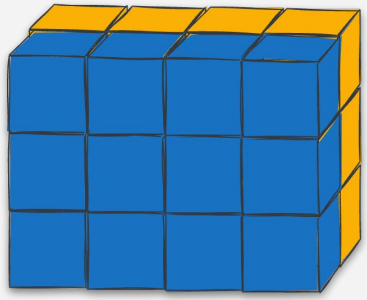
Scalar



Vector



Matrix



Tensor

```
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

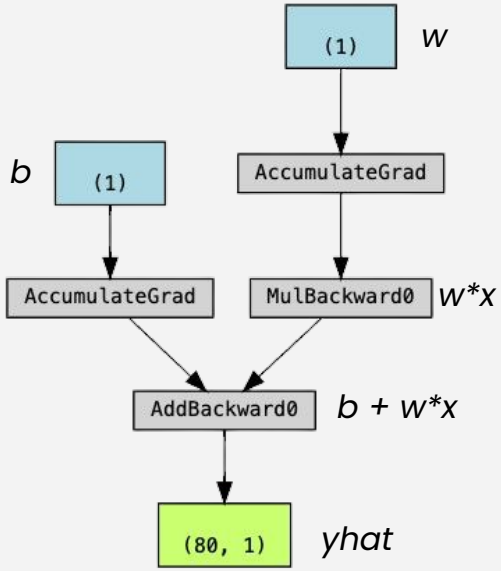
```
# We can specify the device at the moment of creation
# RECOMMENDED!

# 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

```
# 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

```
# 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()
    # 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

```
# 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
```

```
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)
```

```
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

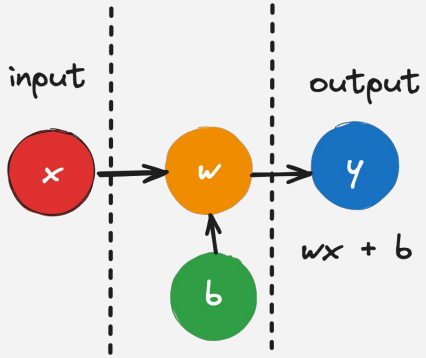
It is the moment when the model makes predictions

```
# 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'))])
```

```
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

```
# 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

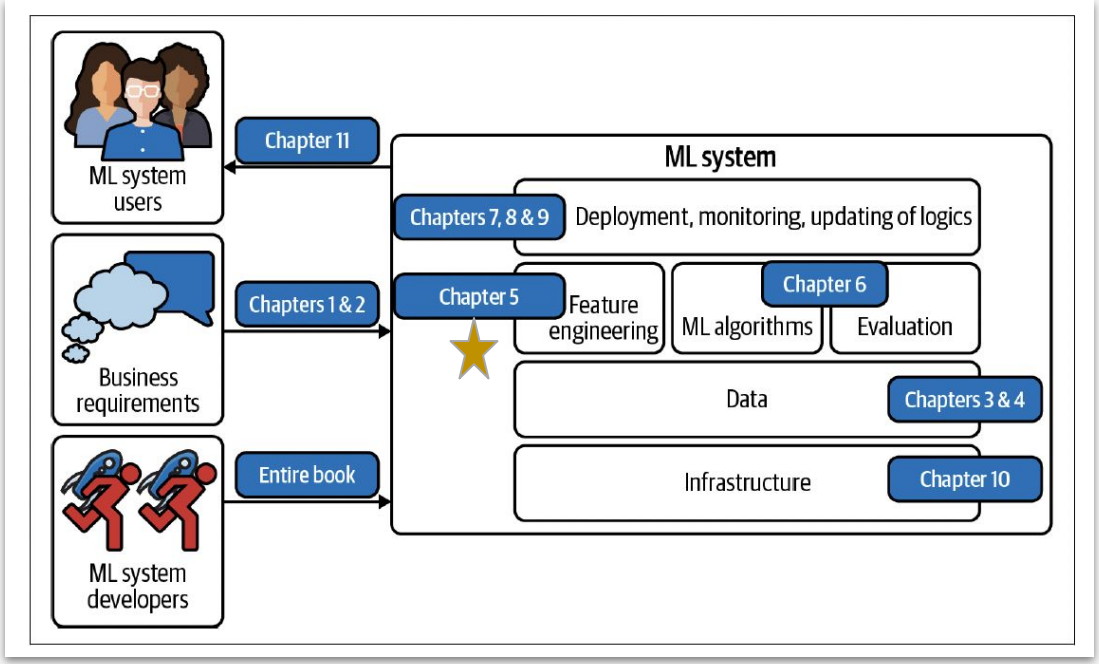
```
# Defines number of epochs
n_epochs = 1000

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'))])
```

Training

The Principles of Deep Learning Theory https://arxiv.org/pdf/2106.10165.pdf

Main Concepts: Identify and explain the primary ideas and theories presented in the chapter. Focus on the fundamental principles of deep learning theory as discussed by the authors.
Theoretical Implications: Critically analyze how these concepts contribute to our understanding of deep neural networks. Consider the implications of the effective theory approach and the focus on intuitive understanding over formal calculations.
Critical Evaluation: Provide your own perspective on the strengths and limitations of the approaches and theories discussed.
Minimal one page size

Follow this writing strategy (Link)



Key Takeaways: Summarize the essential points from Chapter 5, emphasizing the main strategies and techniques for feature engineering highlighted by the author.
Best Practices Identification: From your reading, distill the most critical best practices for feature engineering. Explain why these practices are important and how they contribute to the success of a machine learning model.
Minimal one page size

Follow this writing strategy (Link)

github.com/DataTalksClub/mlops-zoomcamp/tree/main/03-orchestration

Files

main

Go to file

- .github
- 01-intro
- 02-experiment-tracking
- 03-orchestration
  - 3.2
  - 3.3
  - 3.4
  - 3.5
  - 3.6
  - images
  - .gitignore
  - README.md
  - meta.json
  - requirements.txt
- 04-deployment
- 05-monitoring
- 06-best-practices
- 07-project

mlops-zoomcamp / 03-orchestration /

Top

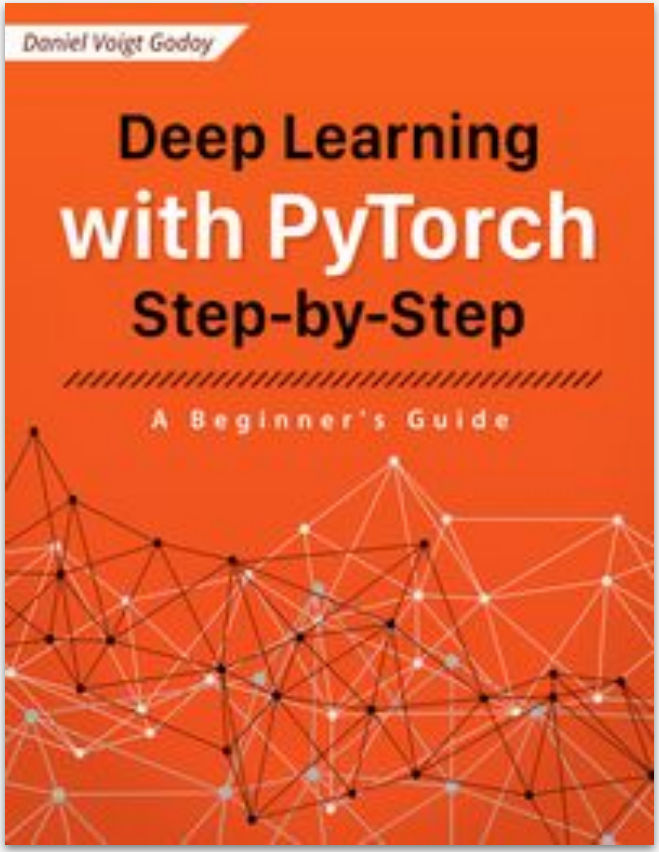
3. Orchestration and ML Pipelines

This section of the repo contains Python code to accompany the videos that show how to use Prefect for MLOps. We will create workflows that you can orchestrate and observe.

3.1 Introdution to Workflow Orchestration

3.2 Introduction to Prefect

Optional



GitHub

Weeks 10 and 11 Fundamentals of Deep Learning PDF

- Outline Video
- The perceptron Video
- Building Neural Networks Video
- Matrix Dimension Video
- Applying Neural Networks Video
- Training a Neural Networks Video
- Backpropagation with Pencil & Paper Video
- Learning rate & Batch Size Video
- Exponentially Weighted Average Video
- Adam, Momentum, RMSProp, Learning Rate Decay Video
- Hands on 🔥
  - TensorFlow Crash Course Notebook
  - Better Learning - Part I Notebook