DATA SCIENCE

Transfer Learning with PyTorch

And why you shouldn't write CNN architectures from scratch.

Dario Radečić
Jun 11, 2020 6 min read

Training deep learning models, especially image-recognition-related ones, is a pretty straightforward task nowadays. There are many reasons why you shouldn’t stress too much about the architecture, the main being that someone already did that step for you. For the rest, you’ll have to read further.

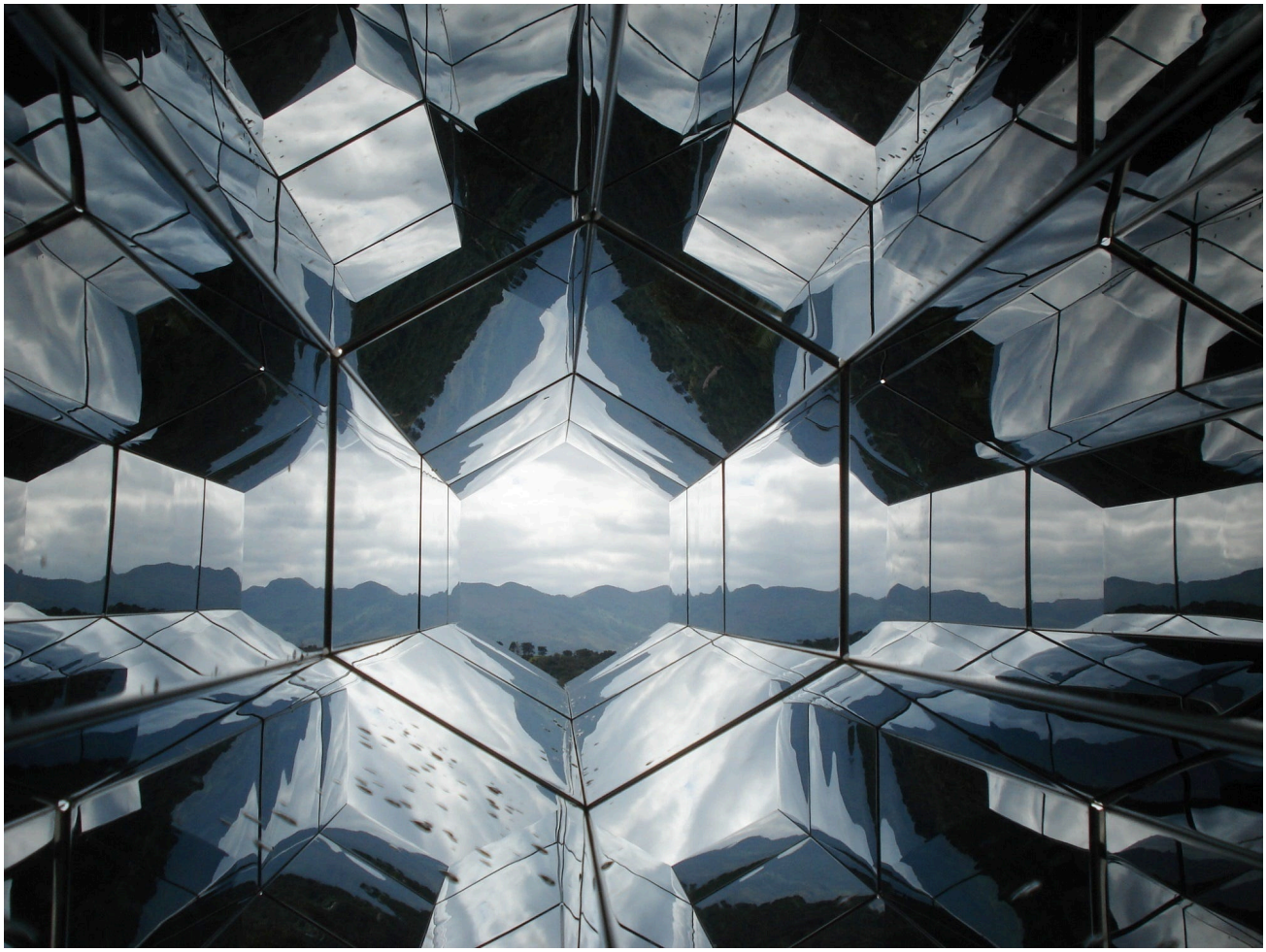


Photo by drmakete lab on Unsplash

Source code: Colab Notebook

Nowadays, the only thing on which you as an engineer should focus is **data preparation** – and in the world of deep learning that term summarizes the process of data gathering, loading, normalization, and augmentation.

Today's agenda is simple – explain what transfer learning is and how it can be used, followed by practical examples of model training with and without pre-trained architectures.

Sounds simple, so let's dive straight in!

Dataset download and basic preparation

Let's start with imports. Here we have the usual suspects like *Numpy*, *Pandas*, and *Matplotlib*, but also our favorite deep learning library – *Pytorch* – followed by everything it has to offer.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision.utils import make_grid
from torchvision import models, transforms, datasets
```

We're writing this code in **Colab**, or Colab Pro to be more precise, so we'll utilize the power of GPUs for training. If you don't know what Colab is, or wondering is it worth it to upgrade to the Pro version, feel free to check these articles:

> **Google Colab: How does it compare to a GPU-enabled laptop?**
>
> **Colab Pro: Is it Worth the Money?**

Because we're training on the GPU and that might not be the case for you, we need a robust way for handling this. Here's a standard approach:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else
device

>>> device(type='cuda', index=0)
```

It should say something like **type='cpu'** if you are training on the CPU, but as Colab is free there's no need to ever do so.

Now onto the **dataset**. We'll be using Dog or Cat dataset for this purpose. It has a plethora of images of various sizes, something which we'll handle down the road. Right now we need to download and unzip it. Here's how:

```
%mkdir data
%cd /content/data/
!wget http://files.fast.ai/data/dogscats.zip

!unzip dogscats.zip
```

After a minute or so, depending on your internet speed, the dataset is ready to use. Now we can declare it as a data directory – not required but will save us a bit time down the road.

```
DIR_DATA = '/content/data/dogscats/'
```

Data Preparation

The first part of the first part is now done. Next, we have to apply some transformations to training and validation subsets, and then load the transformed data with **DataLoaders**. Here are the transformations we applied:

- Random rotation
- Random horizontal flip
- Resizing to 224×224 – required for pre-trained architectures
- Conversion to Tensor
- Normalization

Here's the code:

```
train_transforms = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.Resize(224),
    transforms.CenterCrop((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])

valid_transforms = transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop((224, 224)),
```

```
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

Now we load the data with DataLoaders. This step is also straightforward and something you are probably familiar with:

```
train_data = datasets.ImageFolder(os.path.join(DIR_DATA, 'train')
valid_data = datasets.ImageFolder(os.path.join(DIR_DATA, 'valid')

torch.manual_seed(42)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True
valid_loader = DataLoader(valid_data, batch_size=64, shuffle=Fals

class_names = train_data.classes
class_names

>>> ['cats', 'dogs']
```

If we were now to inverse-normalize a single batch and visualize it, we'd get this:

A quick look at the image above indicates our transformation work as expected.

The data preparation part is now done and in the next section, we'll declare a custom CNN architecture, train it, and evaluate the performance.

Custom Architecture CNN

For this part, we want to do something utterly simple – 3 convolution layers, each followed by max-pooling and ReLU, and then a single fully connected layer and an output layer.

Here's the code for this architecture:

```
class CustomCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, ke
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, k
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, k
        self.fc1 = nn.Linear(in_features=26*26*64, out_features=1
        self.out = nn.Linear(in_features=128, out_features=2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv3(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = x.view(-1, 26*26*64)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, p=0.2)
        x = self.out(x)
        return F.log_softmax(x, dim=1)

torch.manual_seed(42)
model = CustomCNN()
model.to(device)
```

From here we can define an optimizer and criterion and we're ready to train:

```
custom_criterion = nn.CrossEntropyLoss()
custom_optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Since you have access to the source code, and the **train_model** function is stupidly long, we've decided not to put it here. So please refer back to the source code if you are following along. We'll train the model for 10 epochs:

```
custom_model_trained = train_model(
    train_loader=train_loader,
    test_loader=valid_loader,
    model=model,
    criterion=custom_criterion,
    optimizer=custom_optimizer,
    epochs=10
)
```

After some time here are the obtained results:

```
Epoch 10 / 10.. Train loss: 0.292.. Valid loss: 0.332.. Valid accuracy: 0.855
Epoch 10 / 10.. Train loss: 0.254.. Valid loss: 0.339.. Valid accuracy: 0.852
Epoch 10 / 10.. Train loss: 0.267.. Valid loss: 0.347.. Valid accuracy: 0.853
Epoch 10 / 10.. Train loss: 0.244.. Valid loss: 0.338.. Valid accuracy: 0.862
Epoch 10 / 10.. Train loss: 0.263.. Valid loss: 0.331.. Valid accuracy: 0.865
Epoch 10 / 10.. Train loss: 0.232.. Valid loss: 0.326.. Valid accuracy: 0.864
Epoch 10 / 10.. Train loss: 0.267.. Valid loss: 0.336.. Valid accuracy: 0.853
Epoch 10 / 10.. Train loss: 0.260.. Valid loss: 0.335.. Valid accuracy: 0.860

TRAINING FINISHED! Training time: 0:40:03
```

Not terrible results by any means, but **how can we do better?** Transfer learning to the rescue.

Transfer Learning Approach

You can easily look for formal definitions online. For us, transfer learning means downloading a pre-made architecture, which was trained on 1M+ images, and tweaking the output layers so it classifies as many classes as you need.

As we only have cats and dogs here, we'll need to modify this number to two.

For now, we'll download the pre-trained version of the ResNet101 architecture and make it's params not trainable – because the network is trained already:

```
pretrained_model = models.resnet101(pretrained=True)

for param in pretrained_model.parameters():
    param.requires_grad = False
```

Great! Let's check how does the output layer look like:

```
pretrained_model.fc

>>> Linear(in_features=2048, out_features=1000, bias=True)
```

So the architecture by default has 1000 possible classes, but we need only two – one for cats and one for dogs. Here's how to tweak it:

```
pretrained_model.fc = nn.Sequential(
    nn.Linear(2048, 1000),
    nn.ReLU(),
```

```
    nn.Dropout(0.5),
    nn.Linear(1000, 2),
    nn.LogSoftmax(dim=1)
)

pretrained_model.to(device)
```

And that's all we have to do.

Well, we still have to define and optimizer and a criterion, but you know how to do that:

```
pretrained_criterion = nn.CrossEntropyLoss()
pretrained_optimizer = torch.optim.Adam(pretrained_model.fc.param
```

The training process is the same as with the custom architecture, but we won't need so many epochs, because, well, we already know the correct values for weights and biases.

```
pretrained_model_trained = train_model(
    train_loader=train_loader,
    test_loader=valid_loader,
    model=pretrained_model,
    criterion=pretrained_criterion,
    optimizer=pretrained_optimizer,
    epochs=1
)
```

After some time here are the results obtained:

```
Epoch 1 / 1.. Train loss: 0.063.. Valid loss: 0.099.. Valid accuracy: 0.964
Epoch 1 / 1.. Train loss: 0.101.. Valid loss: 0.062.. Valid accuracy: 0.979
Epoch 1 / 1.. Train loss: 0.045.. Valid loss: 0.045.. Valid accuracy: 0.984
Epoch 1 / 1.. Train loss: 0.071.. Valid loss: 0.052.. Valid accuracy: 0.981
Epoch 1 / 1.. Train loss: 0.073.. Valid loss: 0.042.. Valid accuracy: 0.985
Epoch 1 / 1.. Train loss: 0.058.. Valid loss: 0.040.. Valid accuracy: 0.987
Epoch 1 / 1.. Train loss: 0.056.. Valid loss: 0.075.. Valid accuracy: 0.976

TRAINING FINISHED! Training time: 0:05:04
```

How amazing is that? Not only did the accuracy improve, but we've also saved a lot of time by not training for too many epochs.

Now you know what transfer learning can do, and also how and why to use it. Let's wrap things up in the next section.

Conclusion

And there you have it – the most simple transfer learning guide for PyTorch. Sure, the results of a custom model could be better if the network was deeper, but that's not the point. The point is, there's no need to stress about how many layers are enough, and what the optimal hyperparameter values are. At least for most cases.

Make sure to try out different architectures and feel free to inform us about the results down below in the comment section.

Thanks for reading.

Loved the article? Become a Medium member to continue learning without limits. I'll receive a portion of your membership fee if you use the following link, with no extra cost to you.

Join Medium with my referral link – Dario Radečić

WRITTEN BY

Dario Radečić

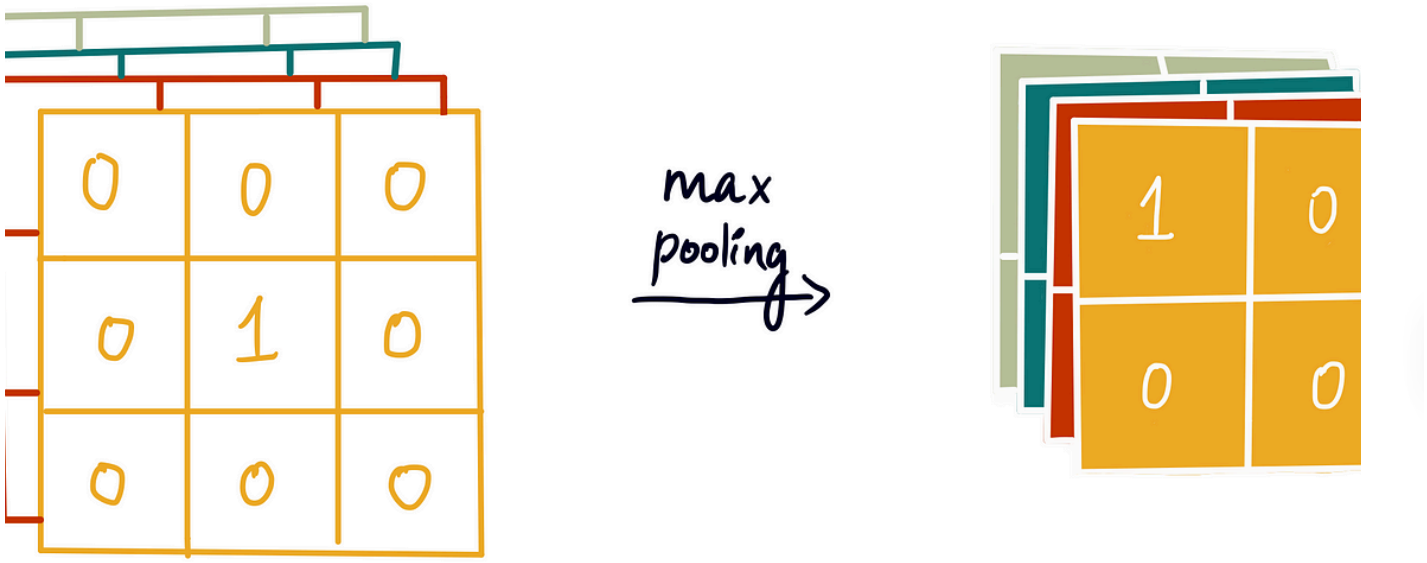
See all from Dario Radečić

Data Science | Deep Learning | Machine Learning | Programming | Python

Share This Article

Towards Data Science is a community publication. Submit your insights to reach our global audience and earn through the TDS Author Payment Program.

Related Articles



ARTIFICIAL INTELLIGENCE

Implementing Convolutional Neural Networks in TensorFlow

Step-by-step code guide to building a Convolutional Neural Network

Shreya Rao

August 20, 2024 6 min read

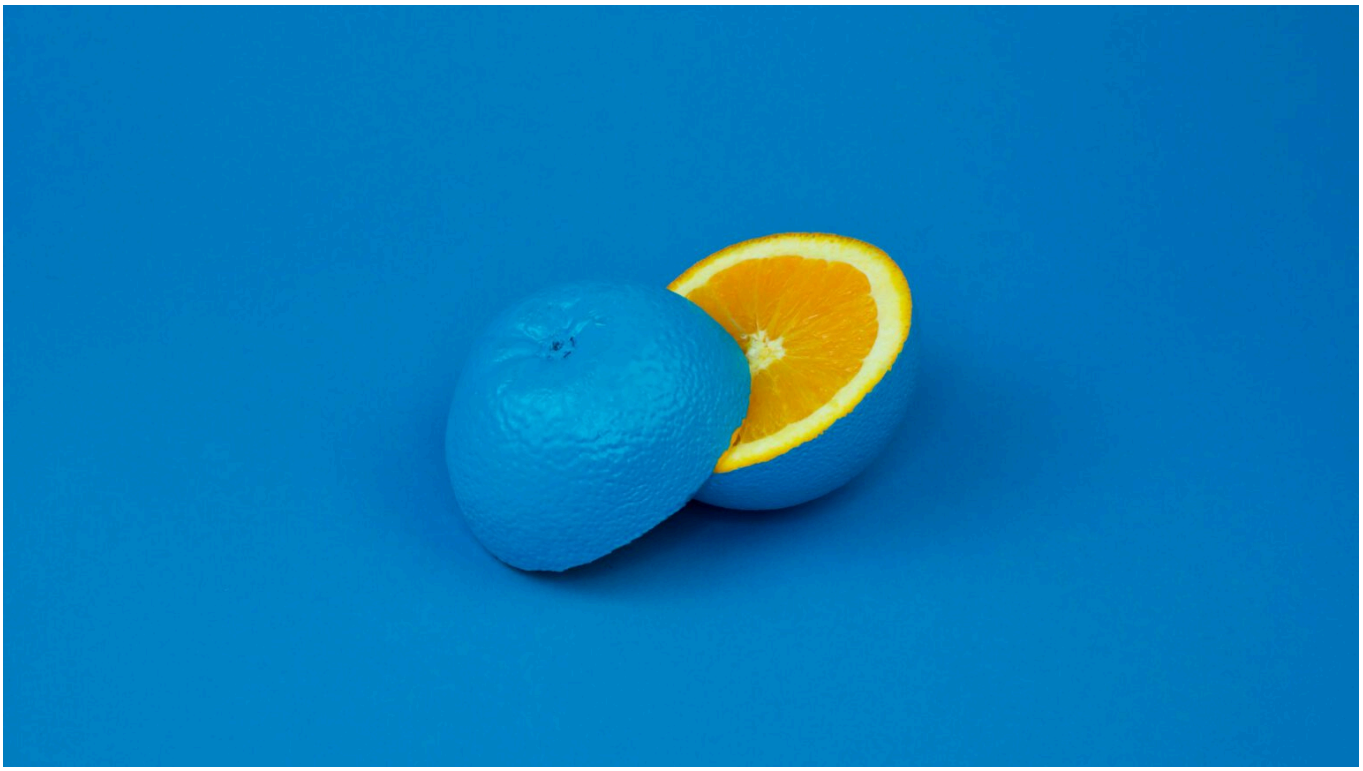
ARTIFICIAL INTELLIGENCE

How to Forecast Hierarchical Time Series

A beginner's guide to forecast reconciliation

Dr. Robert Kübler

August 20, 2024 13 min read



DATA SCIENCE

Hands-on Time Series Anomaly Detection using Autoencoders, with Python

Here's how to use Autoencoders to detect signals with anomalies in a few lines of...

Piero Paialunga
August 21, 2024 12 min read

MACHINE LEARNING

3 AI Use Cases (That Are Not a Chatbot)

Feature engineering, structuring unstructured data, and lead scoring

Shaw Talebi
August 21, 2024 7 min read

DATA SCIENCE

Solving a Constrained Project Scheduling Problem with Quantum Annealing

Solving the resource constrained project scheduling problem (RCPSP) with D-Wave's hybrid constrained quadratic model (CQM)

Luis Fernando PÉREZ ARMAS, Ph.D.
August 20, 2024 29 min read

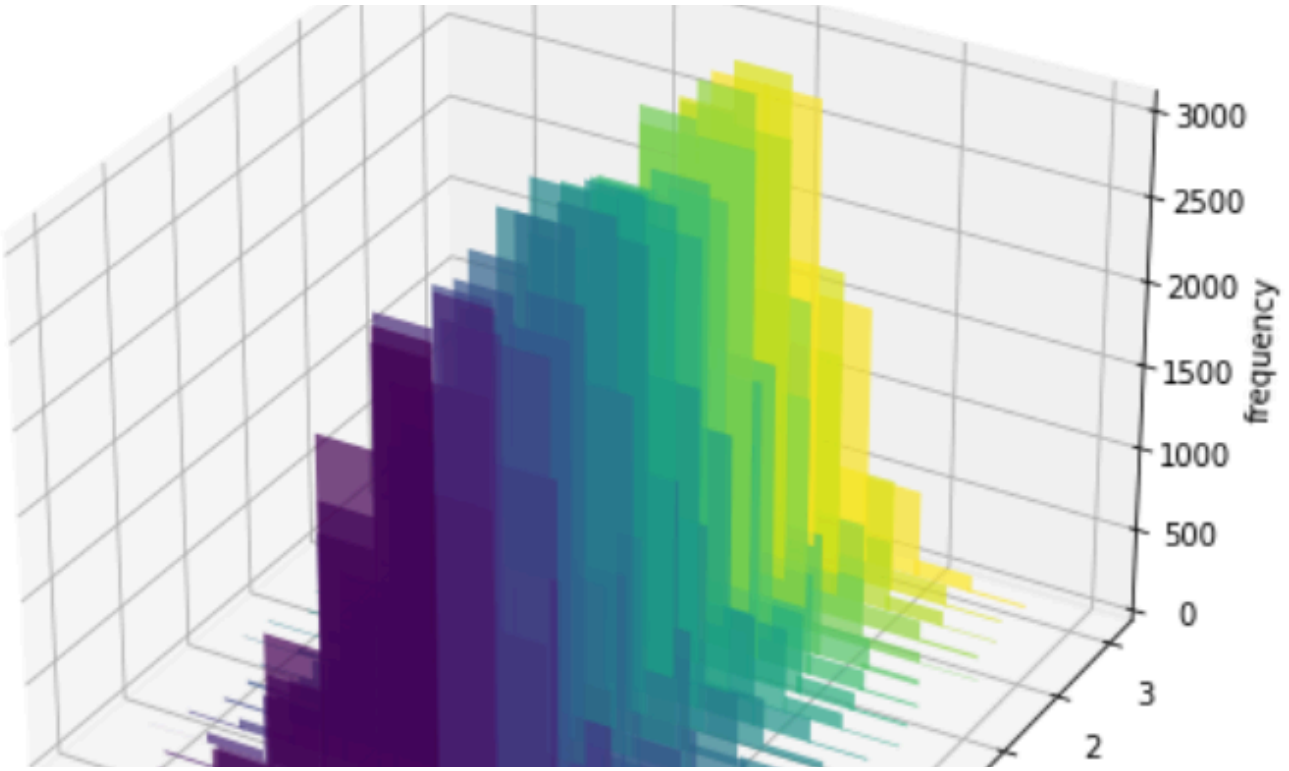
DATA SCIENCE

Back To Basics, Part Uno: Linear Regression and Cost Function

An illustrated guide on essential machine learning concepts

Shreya Rao

February 3, 2023 6 min read



DATA SCIENCE

Must-Know in Statistics: The Bivariate Normal Projection Explained

Derivation and practical examples of this powerful concept

Luigi Battistoni

August 14, 2024 7 min read