A batch too large: Finding the batch size that fits on GPUs

A simple function to identify the batch size for your PyTorch model that can fill the GPU memory

Bryan M. Li · Follow
Published in Towards Data Science
5 min read · Oct 19, 2022

Listen Share More

I am sure many of you had the following painful experience: you start multiple ML experiments on your GPUs to train overnight and when you came back to check 10 hours later, you realize that the progress bar has barely moved due to hardware underutilization, or worse, all the experiments failed due to out-of-memory (OOM) error. In this mini-guide, we will implement an automated method to find the batch size for your PyTorch model that can utilize the GPU memory sufficiently without causing OOM!

Photo by Nana Dua on Unsplash

On top of the model architecture and the number of parameters, the batch size is the most effective hyperparameter to control the amount of GPU memory an experiment uses. The proper method to find the optimal batch size that can fully utilize the accelerator is via GPU profiling, a process to monitor processes on the computing device. Both TensorFlow and PyTorch provide detailed guides and tutorials on how to perform profiling in their framework. In addition, the batch size can greatly affect the performance of the model. For instance, a large batch size can lead to poor generalization, check out this blog post by Kevin Shen on the effect of batch size on training dynamics if you are interested in this topic.

Nevertheless, if you simply want to train a model to test an idea, profiling or performing a hyperparameter search to find the best batch size might be overkill, especially in the early stage of the project. A common approach to find the value that allows you to fit your model without OOM is to train the model with a small batch size while monitoring the GPU utilization using tools like `nvidia-smi` or `nvitop`. You then increase the value if the model is underutilizing the GPU memory, and repeat the process until you hit the memory capacity. However, this manual process can be time-consuming. More annoyingly, when you have to run experiments on different GPUs with varying memory sizes then you have to repeat the same process for each device. Luckily, we can convert this tedious iterative process into code and run it before the actual experiment so that you know your model won't cause an OOM.

The idea is very simple:

1. Initialize your model.
2. Set batch size to 2 (for BatchNorm)
3. Create dummy data that has the sample shape as the real data.
4. Train the model for `n` steps (both forward and backward passes).
5. If the model ran without an error, then increase the batch size and go to Step 3. If OOM is raised (i.e. `RuntimeError` in PyTorch) then set the batch size to the previous value and terminate.
6. Return the final batch size.

To put this into code

```python
def get_batch_size(
    model: nn.Module,
    device: torch.device,
    input_shape: t.Tuple[int, int, int],
    output_shape: t.Tuple[int],
    dataset_size: int,
    max_batch_size: int = None,
    num_iterations: int = 5,
) -> int:
    model.to(device)
    model.train(True)
    optimizer = torch.optim.Adam(model.parameters())

    batch_size = 2
    while True:
        if max_batch_size is not None and batch_size >= max_batch_size:
            batch_size = max_batch_size
            break
        if batch_size >= dataset_size:
            batch_size = batch_size // 2
            break
        try:
            for _ in range(num_iterations):
                # dummy inputs and targets
                inputs = torch.rand(*(batch_size, *input_shape), device=device)
                targets = torch.rand(*(batch_size, *output_shape), device=device)
                outputs = model(inputs)
                loss = F.mse_loss(targets, outputs)
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()
            batch_size *= 2
        except RuntimeError:
            batch_size //= 2
            break
    del model, optimizer
    torch.cuda.empty_cache()
    return batch_size
```

find_batch_size.py hosted with ❤ by **GitHub** **view raw**

As you can see, this function has 7 arguments:

- `model` — the model you want to fit, note that the model will be deleted from memory at the end of the function.
- `device` — `torch.device` which should be a CUDA device.
- `input_shape` — the input shape of the data.
- `output_shape` — the expected output shape of the model.
- `dataset_size` — the size of your dataset (we wouldn't want to continue the search when the batch size is already larger than the size of the dataset).
- `max_batch_size` — an optional argument to set the maximum batch size to use.
- `num_iterations` — the number of iterations to update the model before increasing the batch size, default to 5.

Let's quickly go through what's happening in the function. We first load the model to the GPU, initialize Adam optimizer, and set the initial batch size to 2 (you can start with a batch size of 1 if you are not using BatchNorm). We can then begin the iterative process. First, we check if the current batch size is larger than the size of the dataset or the maximum desired batch size, if so, we break the loop. Otherwise, we create dummy inputs and targets, move them to GPU and fit the model. We train the model for 5 steps to ensure neither forward nor backward pass causes OOM. If everything is fine, we multiply the batch size by 2 and re-fit the model. If OOM occurs during the above steps, then we reduce the batch size by a factor of 2 and exit the loop. Finally, we clear the model and optimizer from memory and return the final batch size. That's it!

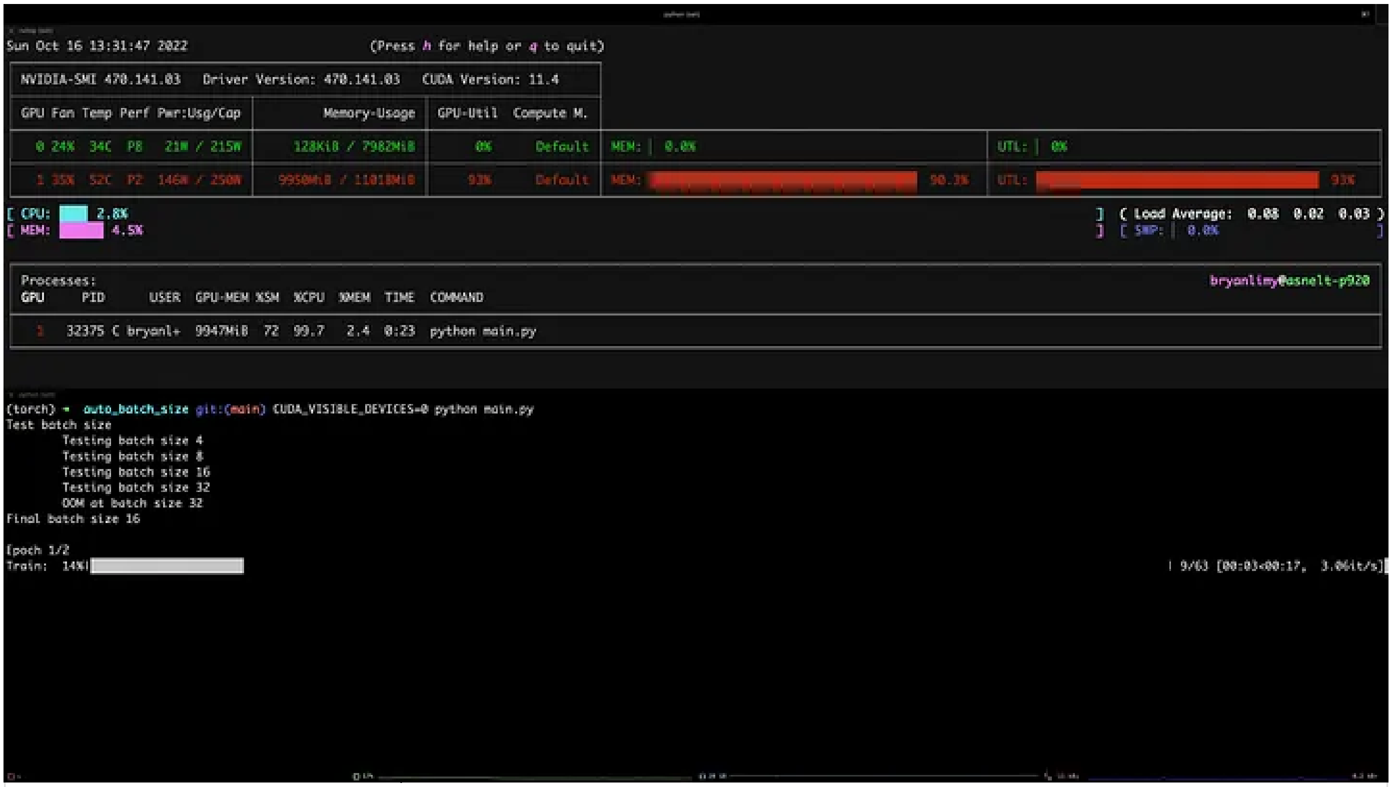
Note that, instead of simply dividing the batch size by 2 if the case of OOM, one could continue to search for the optimal value (i.e. binary search the batch size, set batch size to the mid-point between the breaking and last working value, and continue to Step 3.) to find the batch size that fit perfectly to the GPU. However, keep in mind that PyTorch/TensorFlow or other processes might request more GPU memory in the middle of an experiment and you risk OOM, I hence prefer having some wiggle room.

Now let's put this function into use. Here we fit the ResNet50 on 1,000 train synthetic images of size `(3, 224, 224)` generated by FakeData Datasets. Briefly, we first call `get_batch_size=(model=ResNet(), input_shape=IMAGE_SHAPE, output_shape=(NUM_CLASSES,), dataset_size=DATASET_SIZE)` to get the batch size that can fill the GPU memory sufficiently. Then we can initialize the model and DataLoaders, and train the model like you normally do!

```
import torch
import typing as t
import torch.nn as nn
from tqdm import tqdm
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.models import resnet50, ResNet50_Weights


from time import sleep

# dataset information
IMAGE_SHAPE = (3, 224, 224)
NUM_CLASSES = 100
DATASET_SIZE = 1000


def get_batch_size(
    model: nn.Module,
    device: torch.device,
    input_shape: t.Tuple[int, int, int],
    output_shape: t.Tuple[int],
    dataset_size: int,
    max_batch_size: int = None,
    num_iterations: int = 5,
) -> int:
    model.to(device)
    model.train(True)
    optimizer = torch.optim.Adam(model.parameters())

    print("Test batch size")
    batch_size = 2
    while True:
        if max_batch_size is not None and batch_size >= max_batch_size:
            batch_size = max_batch_size
            break
        if batch_size >= dataset_size:
            batch_size = batch_size // 2
            break
        try:
            for _ in range(num_iterations):
                # dummy inputs and targets
                inputs = torch.rand(*(batch_size, *input_shape), device=device)
```

```
                    inputs = torch.rand(*(batch_size, *input_shape), device=device)
                    targets = torch.rand(*(batch_size, *output_shape), device=device)
                    outputs = model(inputs)
                    loss = F.mse_loss(targets, outputs)
                    loss.backward()
                    optimizer.step()
                    optimizer.zero_grad()
                batch_size *= 2
                print(f"\tTesting batch size {batch_size}")
                sleep(3)
```


```
            datasets.FakeData(
                size=DATASET_SIZE,
                image_size=IMAGE_SHAPE,
                num_classes=NUM_CLASSES,
                transform=transforms.Compose([transforms.ToTensor()]),
            ),
            batch_size=batch_size,
            shuffle=True,
            num_workers=num_workers,
        )
        test_ds = DataLoader(
            datasets.FakeData(
                size=200,
                image_size=IMAGE_SHAPE,
                num_classes=NUM_CLASSES,
                transform=transforms.Compose([transforms.ToTensor()]),
            ),
            batch_size=batch_size,
            num_workers=num_workers,
        )
        return train_ds, test_ds


```




```python
        optimizer: torch.optim,
        train_ds: DataLoader,
        device: torch.device,
    ):
        model.train()
        train_loss, correct = 0, 0
        for batch_idx, (data, target) in enumerate(tqdm(train_ds, desc="Train")):
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            train_loss += loss.item()
            loss.backward()
            optimizer.step()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
        return {
            "loss": train_loss / len(train_ds),
            "accuracy": 100.0 * correct / len(train_ds.dataset),
        }


    def test(model: nn.Module, test_ds: DataLoader, device: torch.device):
        with torch.no_grad():
            model.eval()
            test_loss, correct = 0, 0
            for data, target in tqdm(test_ds, desc="Test"):
```

```
        for data, target in tqdm(test_ds, desc="Test"):
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target).item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    return {
        "loss": test_loss / len(test_ds),
        "accuracy": 100.0 * correct / len(test_ds.dataset),
    }


def main(epochs: int = 2):
    if not torch.cuda.is_available():
        raise RuntimeError("CUDA is not available.")

    device = torch.device("cuda")

    batch_size = get_batch_size(
        model=ResNet(),
        device=device,
        input_shape=IMAGE_SHAPE,
        output_shape=(NUM_CLASSES,),
        dataset_size=DATASET_SIZE,
    )

    train_ds, test_ds = get_datasets(batch_size=batch_size)
    model = ResNet().to(device)

    optimizer = optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(1, epochs + 1):
        print(f"\nEpoch {epoch}/{epochs}")
        train_result = train(
            model=model, optimizer=optimizer, train_ds=train_ds, device=device
        )
        test_result = test(model=model, test_ds=test_ds, device=device)
        print(
            f'Train loss: {train_result["loss"]:.04f}\t'
            f'accuracy: {train_result["accuracy"]:.2f}%\n'
            f'Test loss: {test_result["loss"]:.04f}\t'
            f'accuracy: {test_result["accuracy"]}%'
        )


```

```
if __name__ == "__main__":
    main()
```

find_batch_size_example.py hosted with ❤ by GitHub view raw

Finally, here are some articles and papers on the topic of batch size and its effects on deep neural networks.

- Epoch vs Batch Size vs Iterations
- Effect of batch size on training dynamics
- What's the Optimal Batch Size to Train a Neural Network?
- On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima
- Don't Decay the Learning Rate, Increase the Batch Size

Deep Learning Pytorch Batch Size Programming

Follow

Written by Bryan M. Li

55 Followers · Writer for Towards Data Science

Biomedical AI CDT student at the University of Edinburgh. bryanli.io

More from Bryan M. Li and Towards Data Science

Bryan M. Li in Towards Data Science

Accelerated TensorFlow model training on Intel Mac GPUs

A mini-guide on how to train TensorFlow model on MacBook Pro dGPU via TensorFlow PluggableDevice

5 min read · Oct 26, 2021

Matt Chapman in Towards Data Science

How I Stay Up to Date With the Latest AI Trends as a Full-Time Data Scientist

No, I don't just ask ChatGPT to tell me

· 8 min read · May 1

1.6K 21

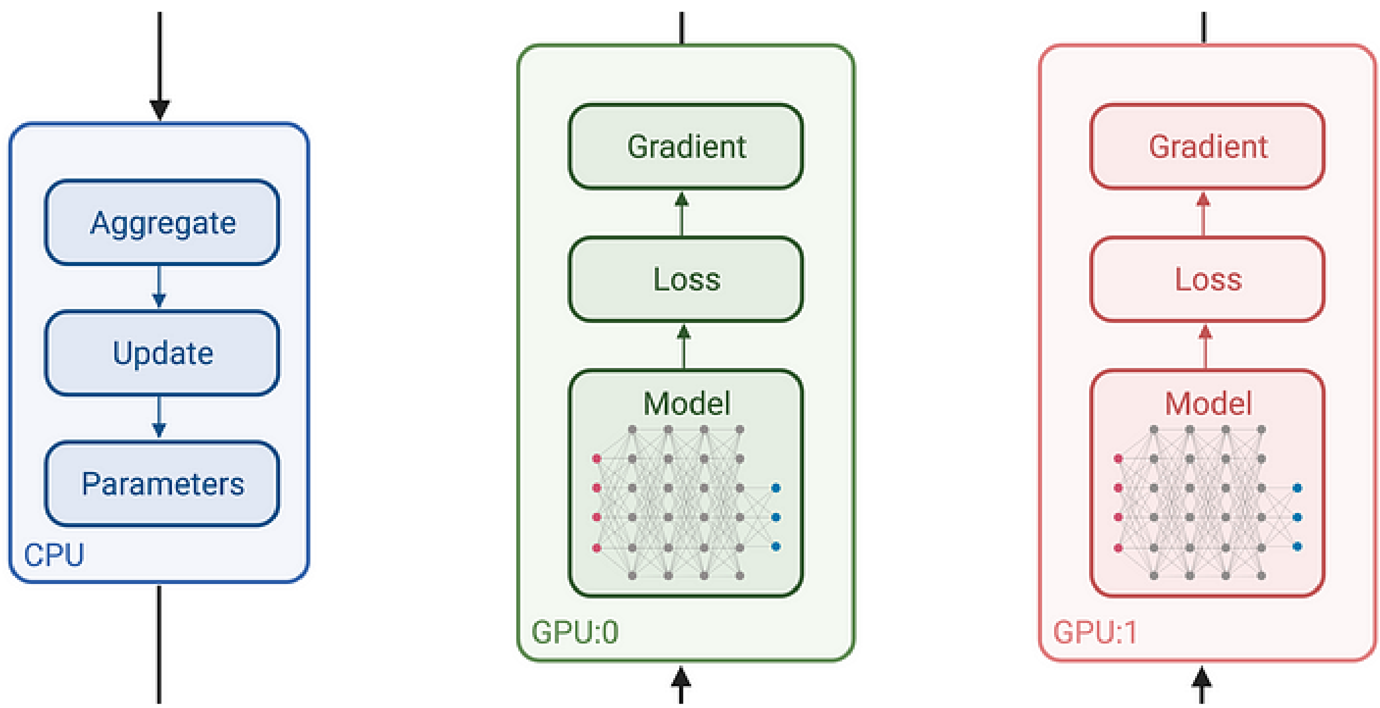
Beatriz Stollnitz in Towards Data Science

How GPT Models Work

Learn the core concepts behind OpenAI's GPT models

14 min read · May 20

1.1K 12



Bryan M. Li in Towards Data Science

Multi-GPUs and custom training loops in TensorFlow 2

A concise example of how to use tf.distribute.MirroredStrategy to train custom training loops model on multiple GPUs.

8 min read · Jun 1, 2021

84

See all from Bryan M. Li

See all from Towards Data Science

Recommended from Medium

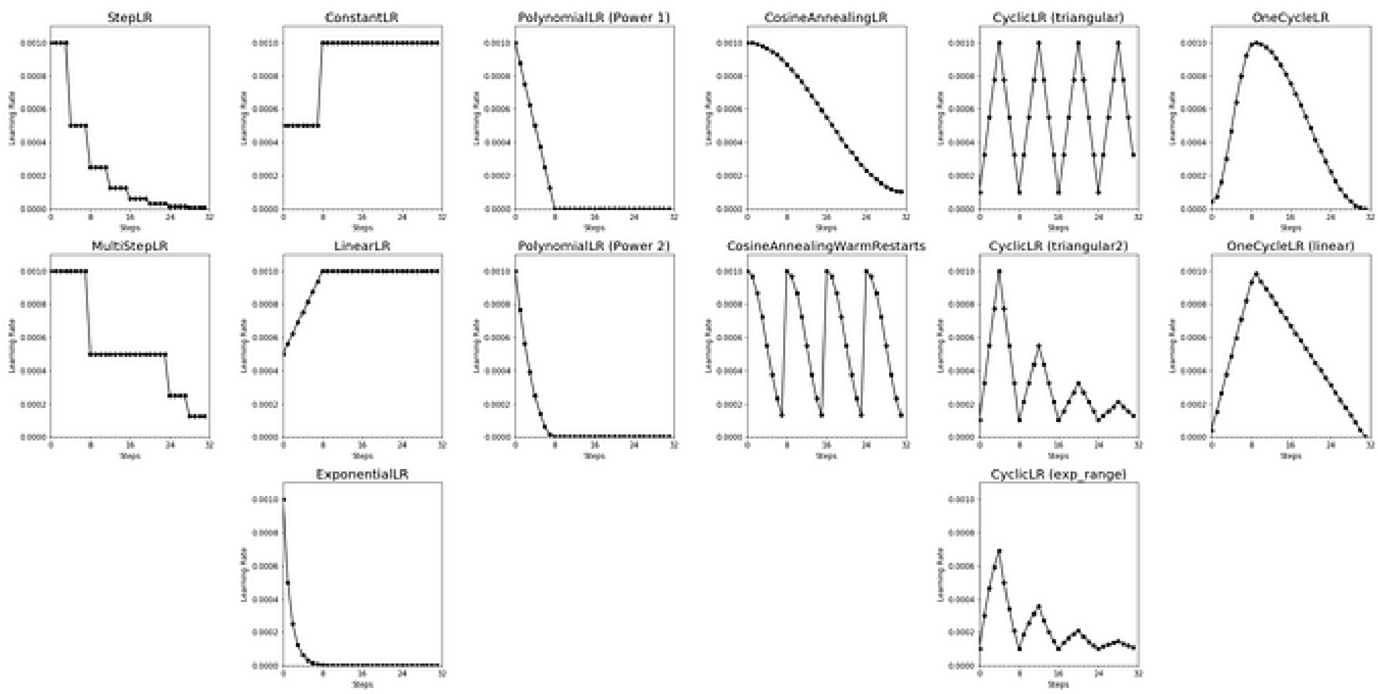
Cameron R. Wolfe in Towards Data Science

The Best Learning Rate Schedules

Practical and powerful tips for setting the learning rate

· 16 min read · Nov 16, 2022

99



Leonie Monigatti in Towards Data Science

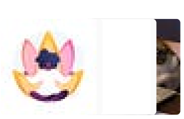
A Visual Guide to Learning Rate Schedulers in PyTorch

LR decay and annealing strategies for Deep Learning in Python

· 9 min read · Dec 6, 2022

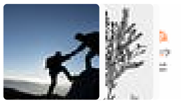
1K 6

Lists



Stories to Help You Grow as a Software Developer

19 stories · 85 saves

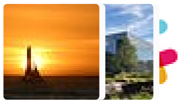


Leadership

30 stories · 31 saves

How to Run More Meaningful 1:1 Meetings

11 stories · 40 saves



Stories to Help You Level-Up at Work

19 stories · 68 saves

Thai Tran

Training PyTorch models on a Mac M1 and M2

PyTorch models on Apple Silicon M1 and M2

· 9 min read · Mar 25

17

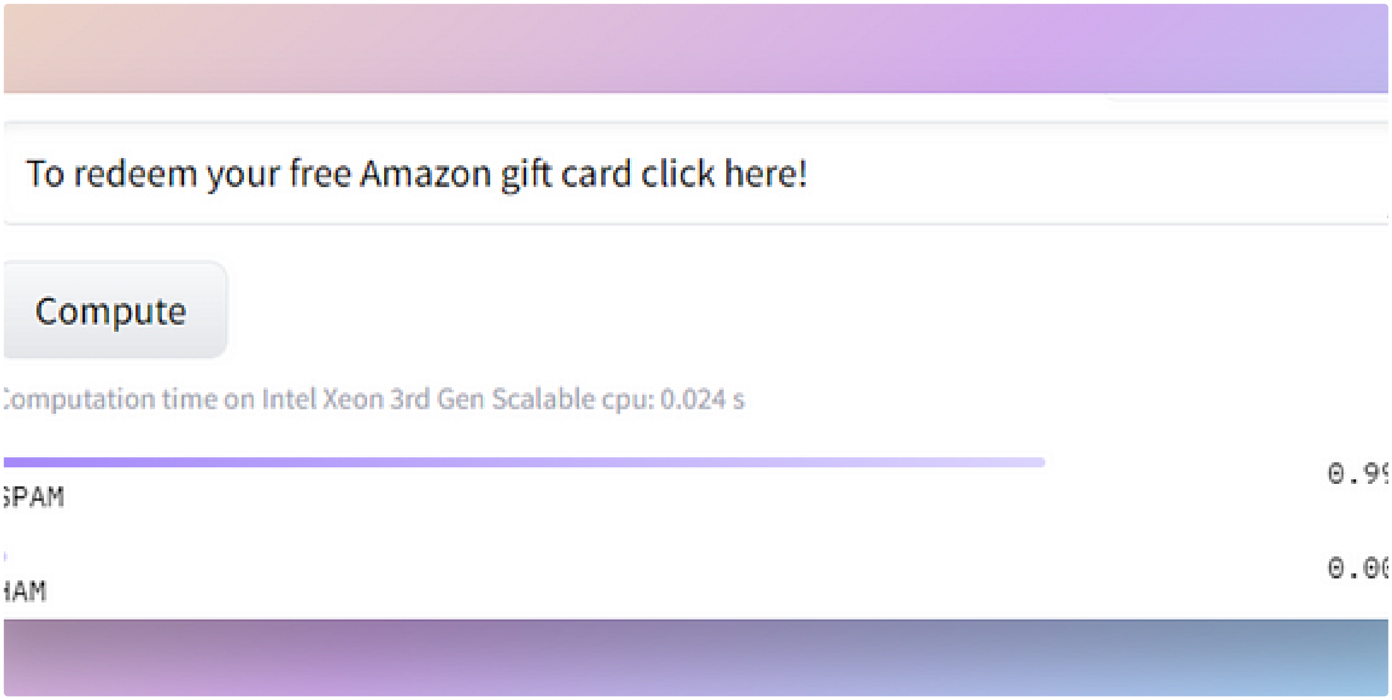
Rukshan Pramoditha in Data Science 365

Determining the Right Batch Size for a Neural Network to Get Better and Faster Results

Guidelines for choosing the right batch size to maintain optimal training speed and accuracy while saving computer resources

· 4 min read · Sep 27, 2022

35



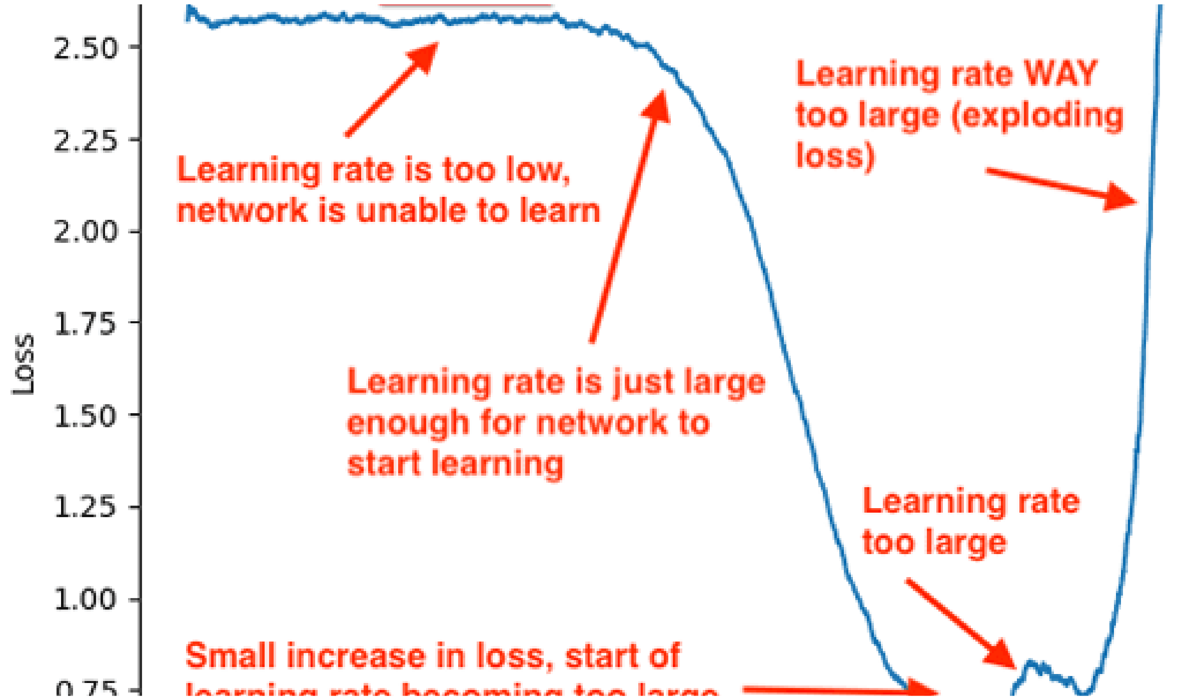
Skanda Vivek in Towards Data Science

Transformer Models For Custom Text Classification Through Fine-Tuning

A tutorial on how to build a spam classifier (or any other classifier) by fine-tuning the DistilBERT model

· 4 min read · Jan 20

140 1



Moklesur Rahman

Learning Rate Scheduler in Keras

The learning rate is considered one of the most important hyperparameters for training deep learning models, but choosing it can be quite…

· 7 min read · Jan 23

30

See more recommendations