

[Open in app](#)[Get started](#)

Published in Towards Data Science

You have **2** free member-only stories left this month.

[Sign up for Medium and get an extra one](#)



Ajinkya Pahinkar

[Follow](#)Sep 7, 2020 · 9 min read ★ · [Listen](#)[Save](#)

A Complete Guide to Using TensorBoard with PyTorch

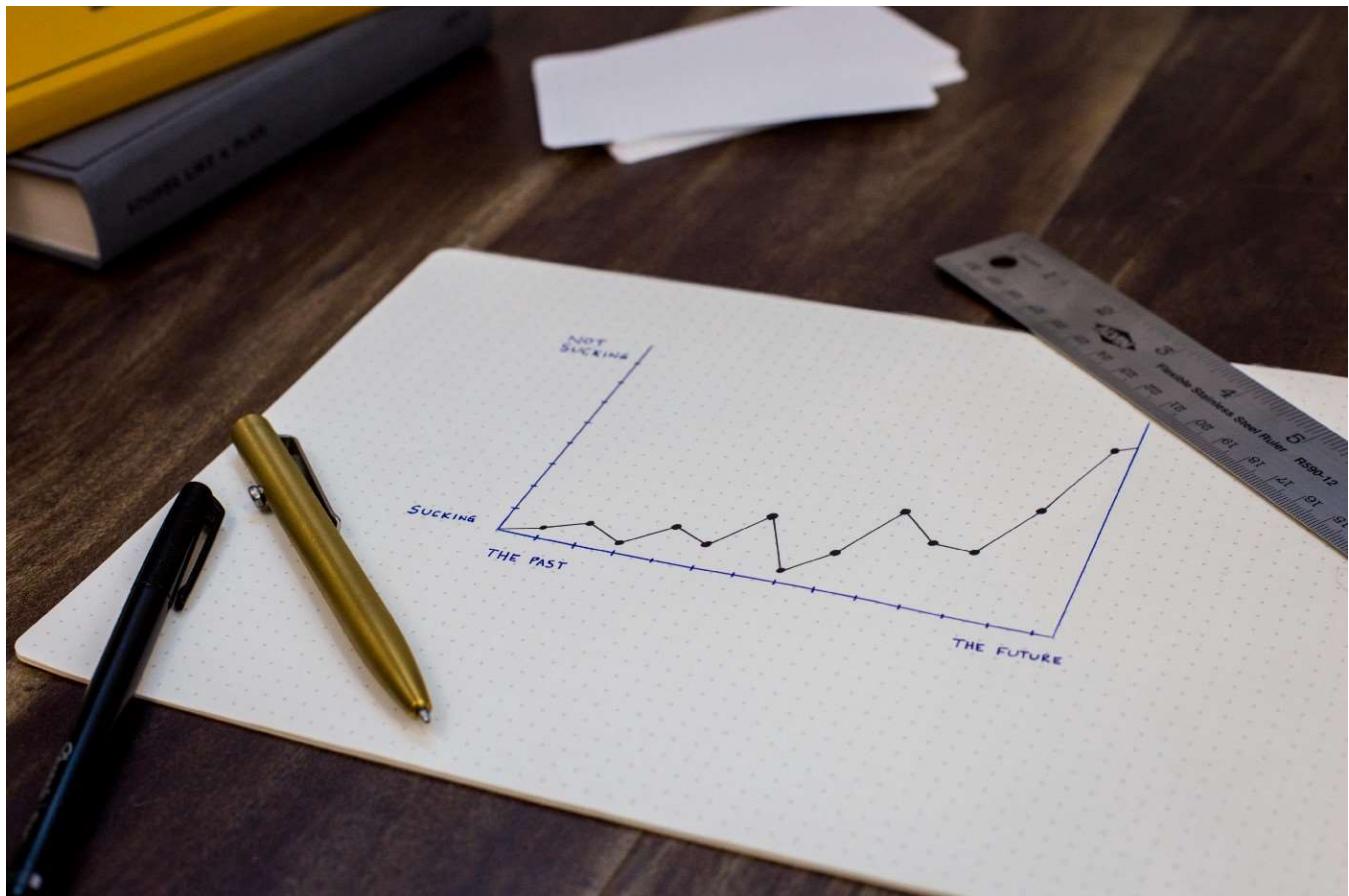


Photo by [Isaac Smith](#) on [Unsplash](#)



[Open in app](#)[Get started](#)

except audio and also learn how to use TensorBoard for efficient hyperparameter analysis and tuning.

Installation Guide:

1. Make sure that your PyTorch version is above 1.10. For this guide, I'm using version **1.5.1**. Use this command to check your PyTorch version.

```
import torch  
print(torch.__version__)
```

2. There are two package managers to install TensorBoard — **pip or Anaconda**. Depending on your python version use any of the following:

Pip installation command:

```
pip install tensorboard
```

Anaconda Installation Command:

```
conda install -c conda-forge tensorboard
```

Note: Having **TensorFlow** installed is not a prerequisite to running TensorBoard, although it is a product of the TensorFlow ecosystem, TensorBoard by itself can be used with PyTorch.

Introduction:

In this guide, we will be using the **FashionMNIST** dataset (60,000 clothing images and 10 class labels for different clothing types) which is a popular dataset inbuilt in the **torch vision** library. It consists of images of clothes, shoes, accessories, etc. along with an integer label corresponding to each category. We will create a simple CNN classifier and then draw inferences from it. Nonetheless, this guide will help you extend the



[Open in app](#)[Get started](#)

Note that in this guide we will not go into details of implementing the **CNN** model and setting up the training loops. Instead, the focus of the article will be on the bookkeeping aspect of Deep Learning projects to get visuals of the inner workings of the models(weights and biases) and the evaluation metrics(**loss, accuracy, num_correct_predictions**) along with hyperparameter tuning. If you are new to the PyTorch framework take a look at my other article on [Implementing CNN in PyTorch\(working with datasets, moving data to GPU, creating models and training loops\)](#) before moving forward.

Importing Libraries and Helper Functions:

```
import torch
import torch.nn as nn
import torch.optim as opt
torch.set_printoptions(linewidth=120)
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.tensorboard import SummaryWriter
```

The last command is the one which enables us to import the Tensorboard class. We will be creating instances of “**SummaryWriter**” and then add our model’s evaluation features like loss, the number of correct predictions, accuracy, etc. to it. One of the novel features of TensorBoard is that we simply have to feed our output tensors to it and it displays the plot of all those metrics, in this way TensorBoard can take care of all the plotting for us.

```
def get_num_correct(preds, labels):
    return preds.argmax(dim=1).eq(labels).sum().item()
```

This line of code helps us to get the number of correct labels after training of the model and applying the trained model to the test set. “**argmax**” gets the index corresponding to the highest value in a tensor. It’s taken across the dim=1 because dim=0 corresponds to the batch of images.”**eq**” compares the predicted labels to the True



[Open in app](#)[Get started](#)

tensor of correct_predictions to a floating point value so that it can be appended to a list(total_correct) for plotting in TensorBoard(Tensors if appended to a list cannot be plotted in TensorBoard, hence we need to convert them to floating point values, append them to a list and then pass this list to TensorBoard for plotting).

CNN Model:

We create a simple CNN model by passing the images through two Convolution layers followed by a set of fully connected layers. Finally we will use a **Softmax Layer** at the end to predict the class labels.

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6,
kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12,
kernel_size=5)

        self.fc1 = nn.Linear(in_features=12*4*4, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=60)
        self.out = nn.Linear(in_features=60, out_features=10)

    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 = torch.flatten(x,start_dim = 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.out(x)

    return x
```

Importing data and creating the train loader:

```
train_set = torchvision.datasets.FashionMNIST(root='./data',
train = True,
download=True,
transform=transforms.ToTensor())
```





Open in app

Get started

Displaying Images And Graphs with TensorBoard:

```
tb = SummaryWriter()
model = CNN()
images, labels = next(iter(train_loader))
grid = torchvision.utils.make_grid(images)
tb.add_image("images", grid)
tb.add_graph(model, images)
tb.close()
```

We create an instance ‘tb’ of the **SummaryWriter** and add images to it by using the **tb.add_image** function. It takes two main arguments, one for the **heading** of the image and another for the **tensor of images**. In this case, we have created a batch of 100 images and passed them to a **grid** which is then added to the tb instance. To the **tb.add_graph** function, we pass our CNN model and a single batch of input images to generate a graph of the model. After running the code a “**runs**” folder will be created in the project directory. All runs going ahead will be sorted in the folder by **date**. This way you have an efficient log of all runs which can be viewed and compared in TensorBoard.

Now use the command line(I use Anaconda Prompt) to redirect into your project directory where the runs folder is present and run the following command:

```
tensorboard --logdir runs
```

It will then serve TensorBoard on the localhost, the link for which will be displayed in the terminal:

```
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.4.0a20200818 at http://localhost:6006/ (Press CTRL+C to quit)
```

“TensorFlow not installed” warning can be ignored

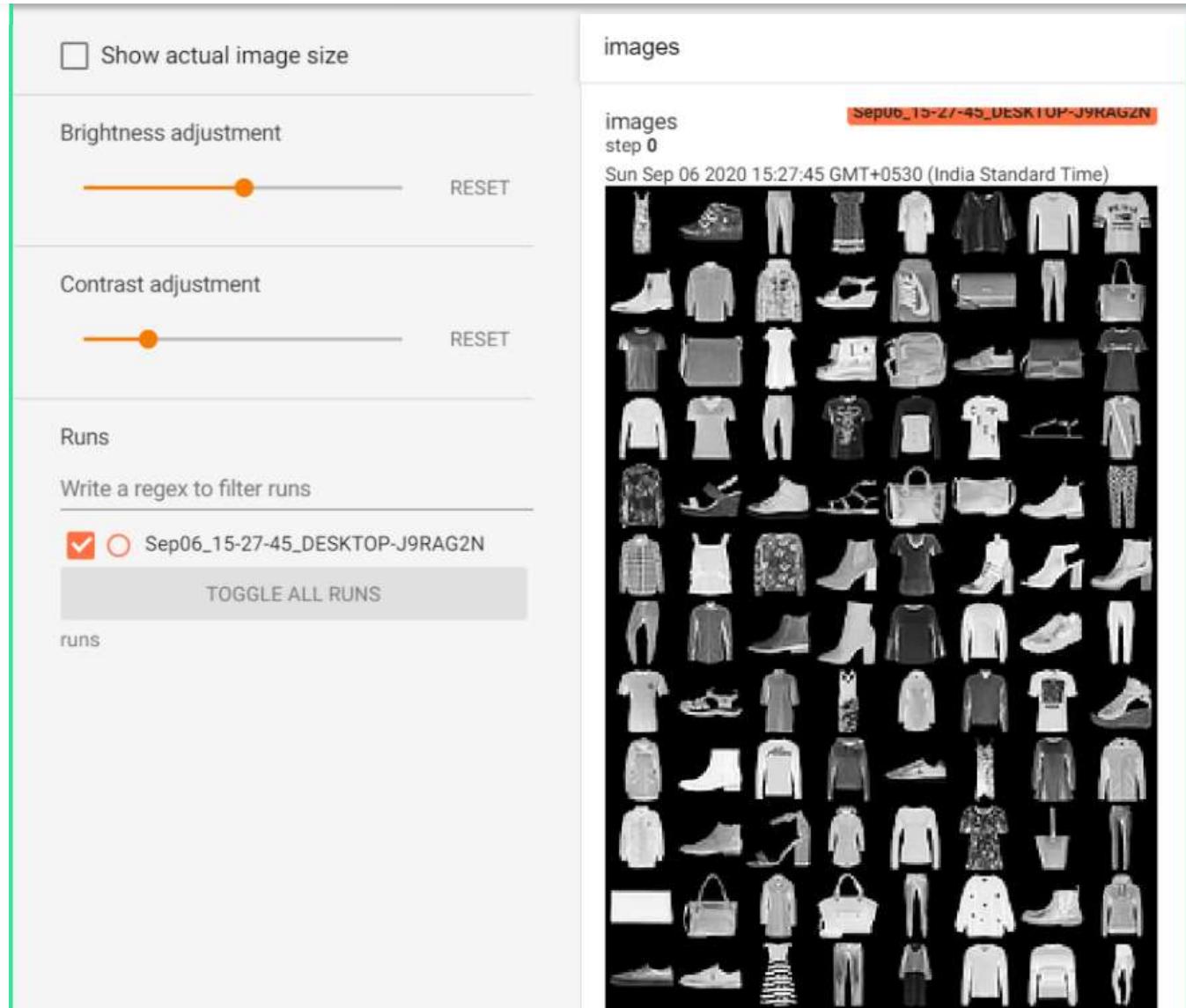
After opening the link we will be able to see all our runs. The Images are visible under the “**Images**” tab. We can use **revert** to filter through the runs and tick those that we





Open in app

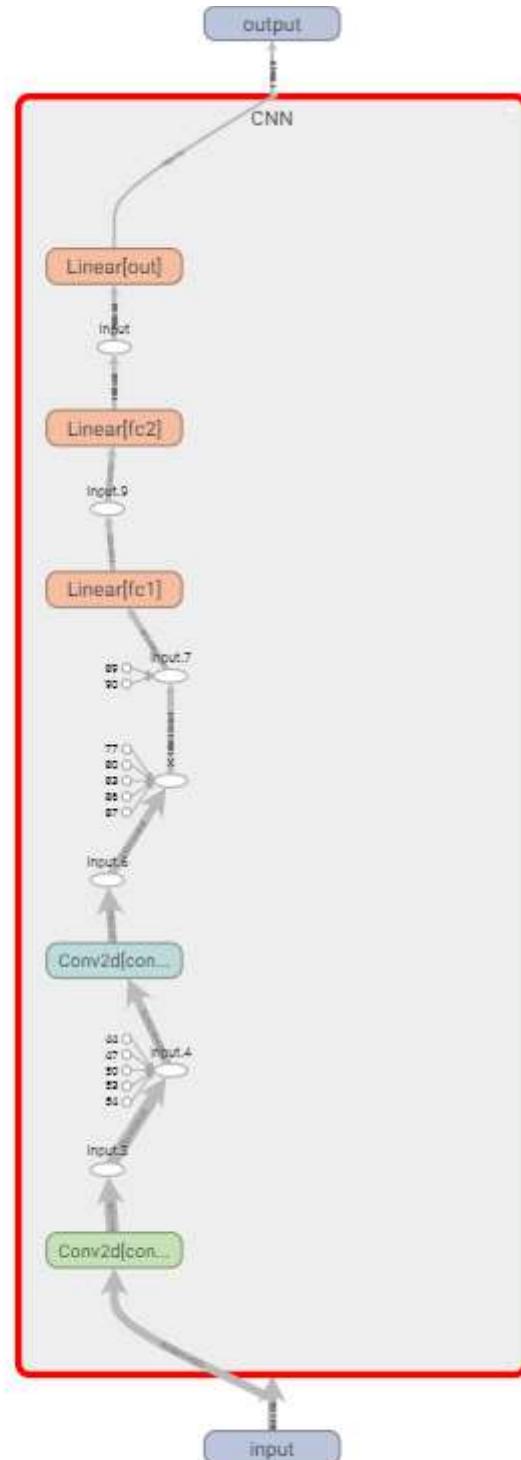
Get started



Grid Plot of Images in TensorBoard

Under the “**Graphs**” tab you will find the graph for the model. It gives details of the entire pipeline of how the dimensions of the batch of images changes after every convolution and linear layer operations. Just double click on any of the icons to grab more information from the graph. It also gives the dimensions of all the weight and bias matrices by double-clicking on any of the Conv2d or Linear layers.



[Open in app](#)[Get started](#)

Graph Generated Using TensorBoard For CNN Model

Training Loop to visualize Evaluation:

```
device = ("cuda" if torch.cuda.is_available() else cpu)
model = CNN().to(device)
```



[Open in app](#)[Get started](#)

```

tb = SummaryWriter()

for epoch in range(10):

    total_loss = 0
    total_correct = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        preds = model(images)

        loss = criterion(preds, labels)
        total_loss+= loss.item()
        total_correct+= get_num_correct(preds, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    tb.add_scalar("Loss", total_loss, epoch)
    tb.add_scalar("Correct", total_correct, epoch)
    tb.add_scalar("Accuracy", total_correct/ len(train_set), epoch)

    tb.add_histogram("conv1.bias", model.conv1.bias, epoch)
    tb.add_histogram("conv1.weight", model.conv1.weight, epoch)
    tb.add_histogram("conv2.bias", model.conv2.bias, epoch)
    tb.add_histogram("conv2.weight", model.conv2.weight, epoch)

    print("epoch:", epoch, "total_correct:", total_correct,
"loss:",total_loss)

tb.close()

```

Alternatively, we can also use a for loop to iterate through all the model parameters including the **fc** and **softmax** layers:

```

for name, weight in model.named_parameters():
    tb.add_histogram(name,weight, epoch)
    tb.add_histogram(f'{name}.grad', weight.grad, epoch)

```

We run the loop for **10 epochs** and at the end of the training loop, we pass augments to the tb variable we created. We have created **total_loss** and **total_correct** variable to keep track of the **loss** and **correct predictions** at the end of each epoch. Note that



[Open in app](#)[Get started](#)

Alternatively, it can also be set to the **batch id** by shifting the tb commands inside the for loop for a batch by using “enumerate” and set the step to **batch_id** as follows:

```
for batch_id, (images, labels) in enumerate(train_loader):
    ...
    tb.add_scalar("Loss", total_loss, batch_id)
```

As seen below running the command mentioned earlier to run TensorBoard will display the line graph and the histograms for the **loss**, **num_correct_predictions**, and **accuracy**.

Line Plot:

Moving the **orange dot** along the graph will give us a log of the respective metric(Accuracy/Correct/Loss) for that particular **epoch**.



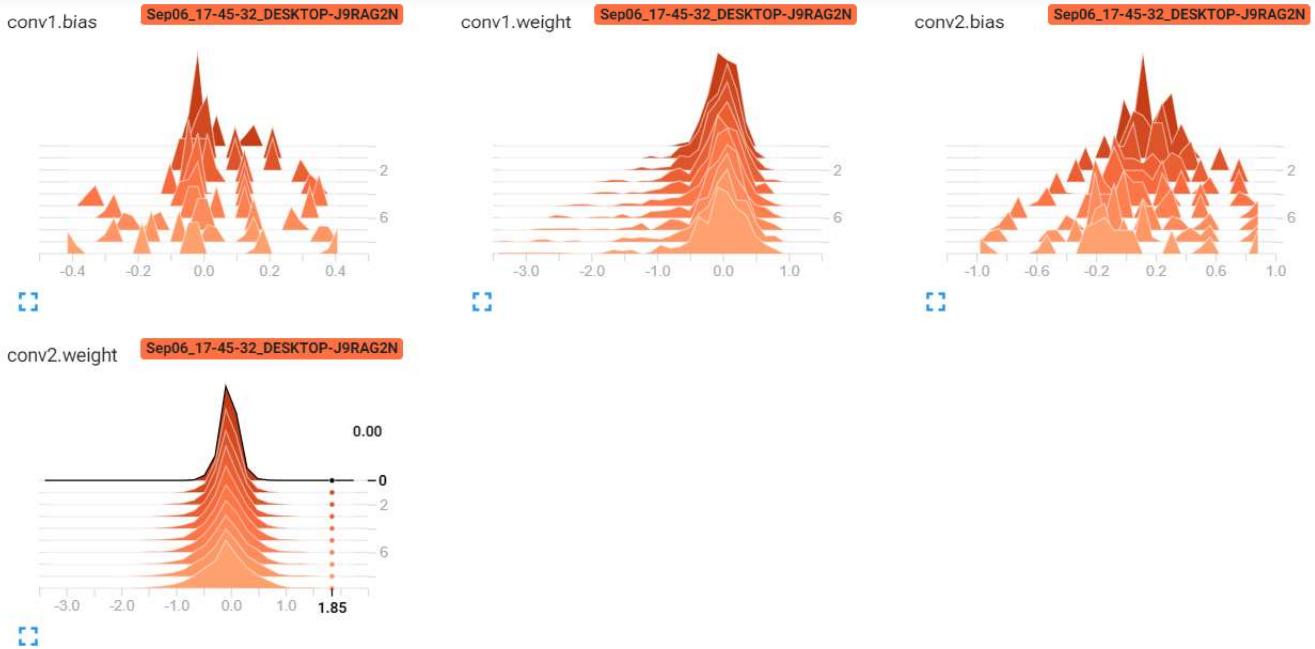
Line Plot Generated By TensorBoard

Histograms:



Open in app

Get started



Histograms generated by TensorBoard

Hyperparameter Tuning:

Firstly we need to change the `batch_size`, `learning_rate`, `shuffle` to dynamic variables. We do that by creating a dictionary as follows:

```
from itertools import product
parameters = dict(
    lr = [0.01, 0.001],
    batch_size = [32, 64, 128],
    shuffle = [True, False]
)

param_values = [v for v in parameters.values()]
print(param_values)

for lr,batch_size, shuffle in product(*param_values):
    print(lr, batch_size, shuffle)
```

This will allow us to get tuples of three, corresponding to all combinations of the **three hyperparameters** and then call a for loop on them before running each epoch loop. In this way, we will be able to have **12 runs**($2(\text{learning_rates}) * 3(\text{batch_sizes}) * 2(\text{shuffles})$) of all the different hyperparameter combinations and compare them on TensorBoard. We will modify the



[Open in app](#)[Get started](#)

```
for run_id, (lr,batch_size, shuffle) in
enumerate(product(*param_values)):
    print("run id:", run_id + 1)
    model = CNN().to(device)
    train_loader = torch.utils.data.DataLoader(train_set,batch_size
= batch_size, shuffle = shuffle)
    optimizer = opt.Adam(model.parameters(), lr= lr)
    criterion = torch.nn.CrossEntropyLoss()
    comment = f' batch_size = {batch_size} lr = {lr} shuffle =
{shuffle}'
    tb = SummaryWriter(comment=comment)
    for epoch in range(5):
        total_loss = 0
        total_correct = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            preds = model(images)

            loss = criterion(preds, labels)
            total_loss+= loss.item()
            total_correct+= get_num_correct(preds, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        tb.add_scalar("Loss", total_loss, epoch)
        tb.add_scalar("Correct", total_correct, epoch)
        tb.add_scalar("Accuracy", total_correct/ len(train_set),
epoch)

        print("batch_size:",batch_size, "lr:",lr,"shuffle:",shuffle)
        print("epoch:", epoch, "total_correct:", total_correct,
"loss:",total_loss)

    print("_____")
    tb.add_hparams(
        {"lr": lr, "bsize": batch_size, "shuffle":shuffle},
        {
            "accuracy": total_correct/ len(train_set),
            "loss": total_loss,
        },
    )
tb.close()
```



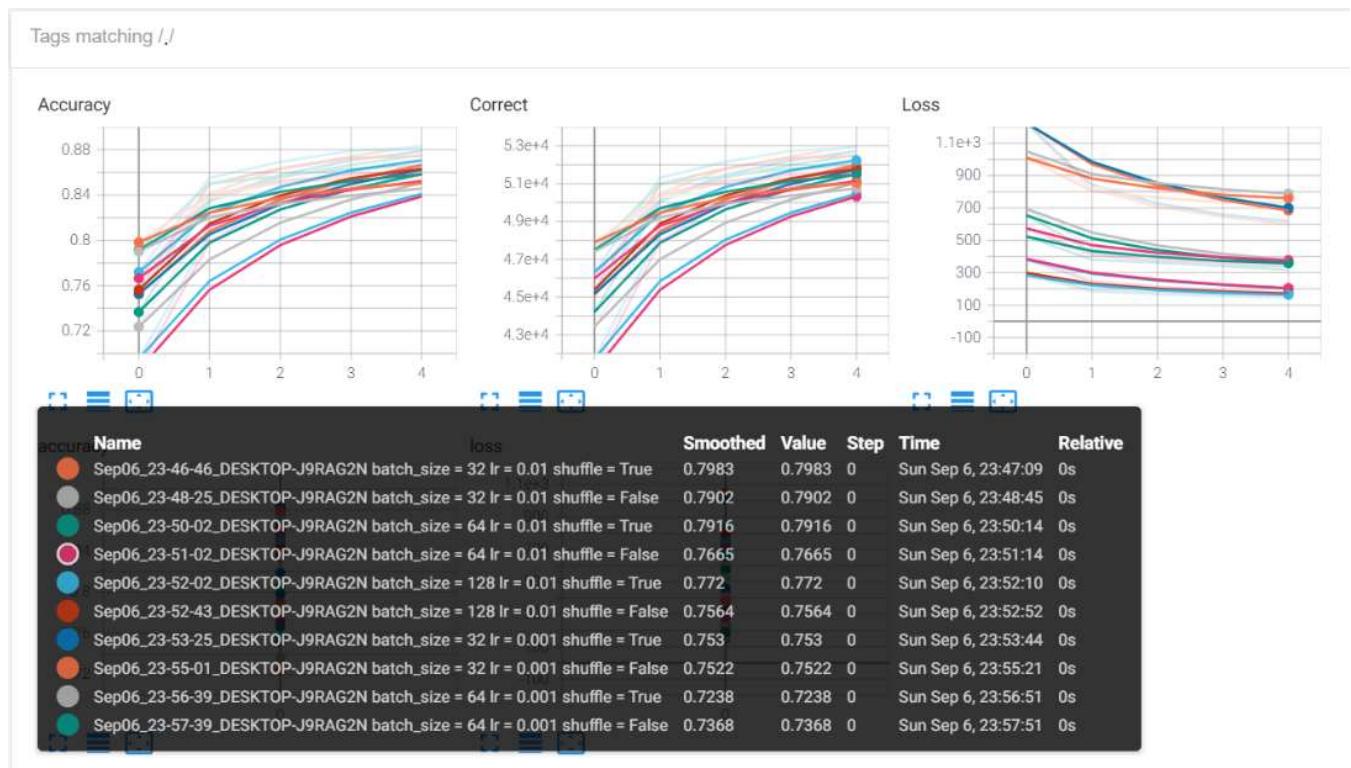


Open in app

Get started

comment as an argument to the **SummaryWriter**. Note that we will be able to see all the runs together and draw comparative analysis across all hyperparameters in TensorBoard. **tb.add_scalar** is the same as earlier just that we have it displayed for all runs this time. **tb.add_hparams** allows us to add **hyperparameters** inside as arguments to keep track of the training progress. It takes two dictionaries as inputs, one for the hyperparameters and another for the evaluation metrics to be analyzed. The results are mapped across all these hyperparameters. It will be clear from the graph mapping diagram at the bottom.

Combined Plots:



As seen above we can filter any runs that we want or have all of them plotted on the same graph. In this way, we can draw a comparative study of the performance of the model across several hyperparameters and tune the model to the ones which give us the best performance. All details like batch size, learning rate, shuffle, and corresponding accuracy values are visible in the pop-up box.

It is evident from the results that a batch size of 32, shuffle set to True, and a learning

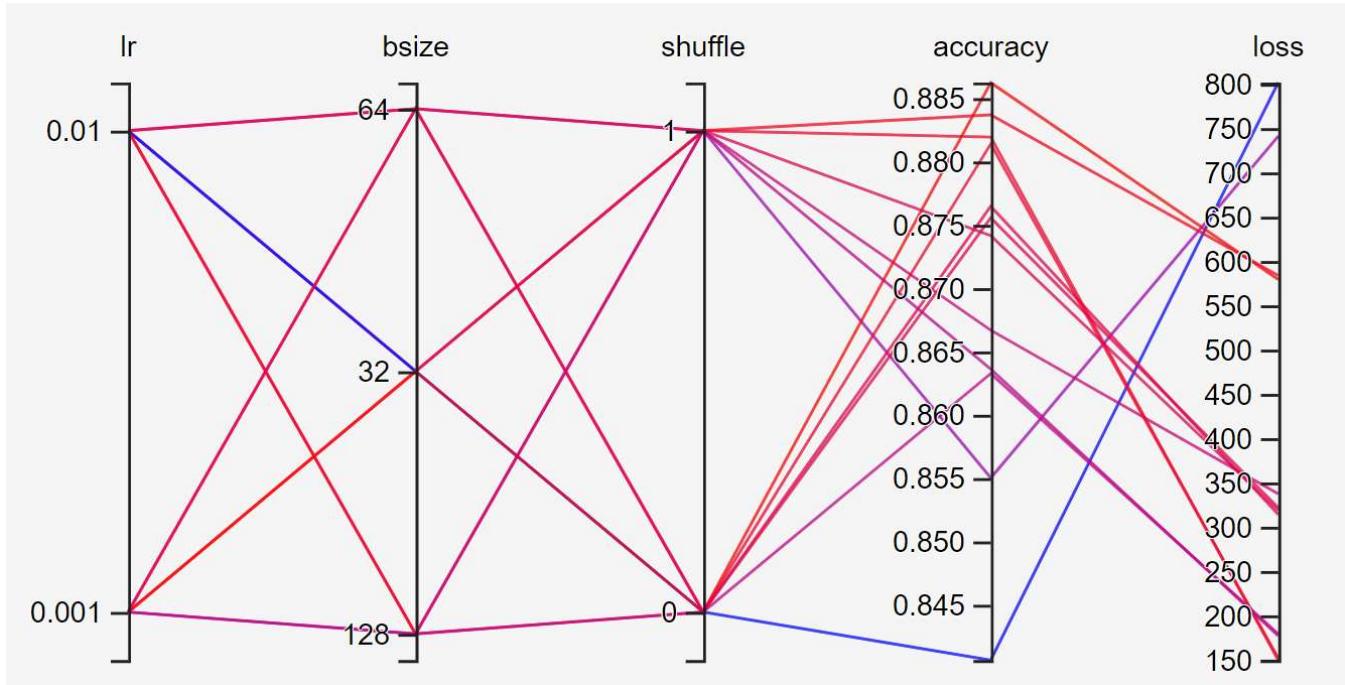




Open in app

Get started

Hyperparameter Graph:



Hyperparameter Graph generated by TensorBoard

This graph has the combined logs of all 12 runs so that you can use the highest accuracy and lowest loss value and trace it back to the corresponding batch size, learning rate and shuffle configurations.

From the Hyperparameter Graph it is very clear that setting shuffle to False(0) tends to yield very poor results. Hence setting shuffle to always True(1) is ideal for training as it adds randomization.

Conclusion:

Feel free to play around with TensorBoard, try to add more hyperparameters to the graph to gain as much information on the pattern of loss convergence and performance of the model given various hyperparameters. We could also potentially add a set of optimizers other than Adam and draw a comparative study. In case of Sequence models like LSTMs, GRUs one can add the time-steps as well to the graph and draw insightful conclusions. Hope this article gave you a thorough insight into using TensorBoard with PyTorch.



[Open in app](#)[Get started](#)

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

[Get this newsletter](#)