Train

Ramin Zarebidoky (LiterallyTheOne)

20 Aug 2025

Train

Introduction

In the previous tutorials, we have learned about **Model**, **Data**, and **Training fundamentals**. Now, let's combine them and train our model on **IRIS dataset**.

Load the data and make the model

Let's go step by step and load our data, and make our model, like the previous tutorial, to train it. First, let's load our data with the code below:

```
iris = load_iris()
```

Now, let's make a `Dataset` for our data.

```
class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target


iris_dataset = IRISDataset(iris.data, iris.target)
```

Then, it is time to split it into `train`, `validation`, and `test`.

```
g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(iris_dataset,
 ↪  [0.7, 0.2, 0.1], g1)

train_loader = DataLoader(train_data, batch_size=10,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=10, shuffle=False)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)
```

Let's create our model as well.

```
class IRISClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers = nn.Sequential(
            nn.Linear(4, 16),
            nn.Linear(16, 8),
            nn.Linear(8, 3),
        )

    def forward(self, x):
        return self.layers(x)


model = IRISClassifier()
```

Now, we are ready to start learning how to train our model.

Train the model

Right now, we know how to train our model in `PyTorch`. So, let's write an optimization step for our model. First, we need to define `loss function` and `optimizer`.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())
```

Now, let's write our training loop.

```
model.train()

for batch_of_data, batch_of_target in train_loader:
    optimizer.zero_grad()

    logits = model(batch_of_data)
```

```
    loss = loss_fn(logits, batch_of_target)
    print(f"loss: {loss.item()}")

    loss.backward()

    optimizer.step()

"""
--------
output:

loss: 1.181538462638855
loss: 1.1570122241973877
loss: 1.1441924571990967
loss: 1.1753343343734741
loss: 1.1002519130706787
loss: 1.1666862964630127
loss: 1.0838695764541626
loss: 1.1226308345794678
loss: 1.1205450296401978
loss: 1.1404510736465454
loss: 1.094001054763794
"""
```

At first, we make sure that our model is in `train` mode by using `model.train()` (When we freshly create a model, it is in `train` mode). Then, we write the code for the `optimization`. As you can see, for each batch of data, we calculated the loss and the gradients and optimized the weights. You might have noticed that the loss in each batch is not necessarily improving. Don't worry about it, because we are going to address it pretty soon.

Evaluate the model

Now, let's write a code to evaluate our model on validation dataset.

```
model.eval()

with torch.inference_mode():
    total_loss = 0

    for batch_of_data, batch_of_target in val_loader:
        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()
```

```
    print(f"average_loss: {total_loss / len(val_loader)}")

"""
--------
output:

average_loss: 1.0044949253400166
"""
```

In the code above, at first, we set the model to the `evaluation` mode, using `model.eval()`. With `torch.inference_mode()`, we disable all the gradient calculations, because we don't need to train our model; we only need to evaluate it. Then, we iterate over our validation dataset. We give each `batch_of_data` to the model to predict the output. After that, we calculate the loss and add it to the total_loss. And finally, we calculate the `average_loss` by dividing `total_loss` by the number of batches, which can be accessed by `len(val_loader)`.

Now, let's add accuracy to this as well. We can calculate the accuracy by dividing the number of correct predictions by the total number of all samples. To do so, we can change our code as below:

```
model.eval()

with torch.inference_mode():
    total_loss = 0
    total_correct = 0

    for batch_of_data, batch_of_target in val_loader:
        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        predictions = logits.argmax(dim=1)
        total_correct +=
↪   predictions.eq(batch_of_target).sum().item()

    print(f"average_loss: {total_loss / len(val_loader)}")
    print(f"accuracy: {total_correct / len(val_loader.dataset)}")

"""
--------
output:

average_loss: 1.0044949253400166
```

```
accuracy: 0.6333333333333333
"""
```

As you can see, I added a variable called `total_correct` which calculates the total number of correct predictions. To calculate if our prediction is wrong or right, as we have done before, at first, we can perform `argmax` on dimension 1 (Right now, we have two dimensions, 0 and 1). Then, we check our prediction against the correct target. Finally, we divide `total_correct` by the total number of samples, which can be accessed with `len(val_loader.dataset)`.

make train_step and val_step

Now, for convenience, let's put our **Training step** and **Validation step** into their functions. Let's start with **Training step**.

```
def train_step():
    model.train()

    total_loss = 0

    for batch_of_data, batch_of_target in train_loader:
        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        loss.backward()

        optimizer.step()

    print(f"training average_loss: {total_loss /
     ↪  len(train_loader)}")
```

As you can see, in the example above, I copied the code that we had written before, with only two changes. First, I removed the printing of `loss` in each batch, to make the output more clean. Second, I calculate `average_loss` like we did in the evaluation. Now, let's add **Validation step**.

```
def val_step():
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0
```

```
        for batch_of_data, batch_of_target in val_loader:
            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
↪   predictions.eq(batch_of_target).sum().item()

        print(f"validation average_loss: {total_loss /
         ↪  len(val_loader)}")
        print(f"validation accuracy: {total_correct /
         ↪  len(val_loader.dataset)}")
```

As you can see in the code above, I just copied the code we have written previously. Now, let's test them to see if they are working correctly.

```
train_step()

"""
--------
output:

training average_loss: 1.1503298336809331
"""
```

```
val_step()

"""
--------
output:

validation average_loss: 1.2322160800298054
validation accuracy: 0.2333333333333334
"""
```

Epoch

Now that we have our **Training and Validation step** ready, let's talk about `epoch`. When we train our model on all the batches for one time, we take one `epoch`. If we repeat this loop for `n` times, we took `n epochs`. Let's create a fresh model, define our loss function, give the model's parameters to our optimizer, and train our model for `5 epochs`

```
model = IRISClassifier()

loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())

for epoch in range(5):
    print("-" * 20)
    print(f"epoch: {epoch}")
    train_step()
    val_step()

"""
--------
output:

--------------------
epoch: 0
training average_loss: 1.1236063025214456
validation average_loss: 1.0980798403422039
validation accuracy: 0.2
--------------------
epoch: 1
training average_loss: 1.0682959123091265
validation average_loss: 1.043296257654826
validation accuracy: 0.5333333333333333
--------------------
epoch: 2
training average_loss: 1.0306043733250012
validation average_loss: 1.0079283316930134
validation accuracy: 0.6
--------------------
epoch: 3
training average_loss: 0.9916351871056699
validation average_loss: 0.9691224495569865
validation accuracy: 0.8333333333333334
--------------------
epoch: 4
training average_loss: 0.9554464546116915
validation average_loss: 0.9225329160690308
validation accuracy: 0.7666666666666667
"""
```

As you can see, in the code above, we have trained and evaluated our model in each `epoch`. Your results and outputs might be different from mine. Because we are working on a small dataset, we haven't learned all the layers and training techniques, so the training results might seem a little bit random. But don't

worry about it, we are going to fix that pretty soon.

Run on Accelerator

We learned how to find the available accelerator in the previous tutorials. Now, we are going to do that, and also make some changes in the code in order to train and evaluate our model on the accelerator.

```
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

"""
--------
output:

mps
"""
```

In the code above, I have found the current accelerator, which for me is `mps`. Now, let's change our `train_step`.

```
def train_step():
    model.train()

    total_loss = 0

    for batch_of_data, batch_of_target in train_loader:
        batch_of_data = batch_of_data.to(device)
        batch_of_target = batch_of_target.to(device)

        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        loss.backward()

        optimizer.step()

    print(f"training average_loss: {total_loss /
     ↪  len(train_loader)}")
```

As you can see, I changed the `device` of `batch_of_data` and `batch_of_target` to the current device. I should do the same for my `val_step` as well.

```
def val_step():
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0

        for batch_of_data, batch_of_target in val_loader:
            batch_of_data = batch_of_data.to(device)
            batch_of_target = batch_of_target.to(device)

            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
 ↪  predictions.eq(batch_of_target).sum().item()

        print(f"validation average_loss: {total_loss /
         ↪  len(val_loader)}")
        print(f"validation accuracy: {total_correct /
         ↪  len(val_loader.dataset)}")
```

Now, I should only change the `device` of the `model` too and run the training procedure again.

```
model = IRISClassifier()
model.to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())

for epoch in range(5):
    print("-" * 20)
    print(f"epoch: {epoch}")
    train_step()
    val_step()

"""
--------
output:
```

```
--------------------
epoch: 0
training average_loss: 1.1559315432201733
validation average_loss: 1.0502928098042805
validation accuracy: 0.36666666666666664
--------------------
epoch: 1
training average_loss: 1.078606204553084
validation average_loss: 1.0400715271631877
validation accuracy: 0.36666666666666664
--------------------
epoch: 2
training average_loss: 1.025301640683954
validation average_loss: 1.0179588794708252
validation accuracy: 0.2
--------------------
epoch: 3
training average_loss: 0.9952371987429532
validation average_loss: 0.9651865760485331
validation accuracy: 0.6
--------------------
epoch: 4
training average_loss: 0.9447547793388367
validation average_loss: 0.9108109474182129
validation accuracy: 0.7666666666666667
"""
```

As you can see, everything is working correctly.

Save and load our model

Now, to save our model, we can use `torch.save` function.

```
torch.save(model.state_dict(), "model.pth")
```

With the code above, we save all the weights of our model to a file called `model.pth`. Now, let's load it into a new model, using `torch.load`.

```
new_model = IRISClassifier()

weights = torch.load("model.pth")

new_model.load_state_dict(weights)

new_model = new_model.to(device)
```

In the code above, I have created a new instance of our model with the name of

`new_model`. Then, I loaded the saved weights with `torch.load`. After that, I used `load_state_dict` to load the weights. Finally, I changed the device of our model to the current accelerator. To test if we have done everything correctly, we can use the code below:

```
for key in new_model.state_dict().keys():
    if key not in model.state_dict().keys():
        print(f"Key {key} not in model.state_dict()")
        break

    if not torch.allclose(new_model.state_dict()[key],
     ↪  model.state_dict()[key]):
        print("Values are different")
        break
```

In the code above, we check if all the layers and weights that we loaded are the same as the model that we used for saving.

Conclusion

In this tutorial, we have trained a simple model with simple layers. The outputs right now are pretty random. But moving forward, we are going to learn more about the different layers and how to get better results. Right now, we know what a simple **Deep Learning** project looks like. We trained our model and then evaluated it. We learned about **Epoch** and learned how to use the accelerator. Finally, we learned how to save our model and load it again.