Train 2

Ramin Zarebidoky (LiterallyTheOne)

21 Aug 2025

Train 2

Introduction

In the previous tutorial, we have learned how to train our model. But our model wasn't getting properly trained. In this tutorial, we want to address that problem and try to solve it.

Modular train step and validation step

In the previous tutorial, we wrote a code to train our model as below:

```python
# -------------------[ Imports ]-------------------
import torch
from torch import nn
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader, random_split

from sklearn.datasets import load_iris

# -------------------[ Find the device ]-------------------
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

# -------------------[ Load the data ]-------------------
iris = load_iris()


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)

# -------------------[ Split the data to 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)


# -------------------[ Define model ]--------------------
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)


# -------------------[ 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)}")


# --------------------[ 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:
            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)}")
```

```
# -------------------[ Create a model ]-------------------
model = IRISClassifier()
model.to(device)

# -------------------[ Define loss function and optimizer
 ↪  ]--------------------
loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())

# -------------------[ Train the model ]-------------------
for epoch in range(5):
    print("-" * 20)
    print(f"epoch: {epoch}")
    train_step()
    val_step()
```

Let's put that code in a file called train_v1.py. Right now, `train_step` and `val_step` only work with the global variables. Let's make them more modular.

```
# -------------------[ Define Training step ]-------------------
def train_step(
        data_loader: DataLoader,
        model: nn.Module,
        optimizer: Optimizer,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.train()

    total_loss = 0
    total_correct = 0

    for batch_of_data, batch_of_target in data_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()

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

```
        loss.backward()

        optimizer.step()

    return total_loss / len(data_loader), total_correct /
     ↪ len(data_loader.dataset)


# --------------------[ Define Validation Step
 ↪ ]--------------------
def val_step(
        data_loader: DataLoader,
        model: nn.Module,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.eval()

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

        for batch_of_data, batch_of_target in data_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()

        return total_loss / len(data_loader), total_correct /
         ↪ len(data_loader.dataset)
```

As you can see, in the code above, we now give the needed arguments to `train_step` and `val_step` to work with. Also, instead of printing the results in each function, now I return the results. For both functions, I return `average loss` and `accuracy`. Now, let's make our code more organized and put it in a file named train_v2.py.

```python
# -------------------[ Imports ]-------------------
import torch
from torch import nn
from torch.optim import Adam, Optimizer
from torch.utils.data import Dataset, DataLoader, random_split

from sklearn.datasets import load_iris


# -------------------[ Define Dataset ]-------------------
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


# -------------------[ Define Model ]-------------------
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)


# -------------------[ Define Training step ]-------------------
def train_step(
        data_loader: DataLoader,
        model: nn.Module,
        optimizer: Optimizer,
        loss_fn: nn.Module,
```

```
        device: str,
) -> tuple[float, float]:
    model.train()

    total_loss = 0
    total_correct = 0

    for batch_of_data, batch_of_target in data_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()

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

        loss.backward()

        optimizer.step()

    return total_loss / len(data_loader), total_correct /
     ↪  len(data_loader.dataset)


# ------------------[ Define Validation Step
 ↪  ]------------------
def val_step(
        data_loader: DataLoader,
        model: nn.Module,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.eval()

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

        for batch_of_data, batch_of_target in data_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()

        return total_loss / len(data_loader), total_correct /
         ↪  len(data_loader.dataset)


def main():
    # --------------------[ Find the accelerator
     ↪  ]--------------------
    if torch.accelerator.is_available():
        device = torch.accelerator.current_accelerator()
    else:
        device = "cpu"

    print(device)

    # --------------------[ Load the data ]--------------------
    iris = load_iris()

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

    # --------------------[ Split the data to 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)

    # --------------------[ Create the model ]--------------------
    model = IRISClassifier()
    model.to(device)
```

```
    # -------------------[ Define loss function and optimizer
    ↪  ]--------------------
    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters())

    # -------------------[ Train and evaluate the model
    ↪  ]--------------------
    for epoch in range(5):
        print("-" * 20)
        print(f"epoch: {epoch}")
        train_loss, train_accuracy = train_step(train_loader,
↪   model, optimizer, loss_fn, device)
        val_loss, val_accuracy = val_step(val_loader, model,
↪   loss_fn, device)
        print(f"train: ")
        print(f"\tloss: {train_loss:.4f}")
        print(f"\taccuracy: {train_accuracy:.4f}")

        print(f"validation: ")
        print(f"\tloss: {val_loss:.4f}")
        print(f"\taccuracy: {val_accuracy:.4f}")

    print("-" * 20)
    test_loss, test_accuracy = val_step(test_loader, model,
↪   loss_fn, device)
    print(f"test: ")
    print(f"\tloss: {test_loss:.4f}")
    print(f"\taccuracy: {test_accuracy:.4f}")


if __name__ == "__main__":
    main()

"""
--------
output:
    mps
--------------------
epoch: 0
train:
    loss: 1.0473
    accuracy: 0.3714
validation:
    loss: 1.0471
    accuracy: 0.2333
```

```
--------------------
epoch: 1
train:
    loss: 0.9799
    accuracy: 0.4857
validation:
    loss: 0.9770
    accuracy: 0.6667
--------------------
epoch: 2
train:
    loss: 0.9447
    accuracy: 0.6571
validation:
    loss: 0.9077
    accuracy: 0.6667
--------------------
epoch: 3
train:
    loss: 0.9004
    accuracy: 0.7143
validation:
    loss: 0.8768
    accuracy: 0.6333
--------------------
epoch: 4
train:
    loss: 0.8546
    accuracy: 0.6857
validation:
    loss: 0.8063
    accuracy: 0.6667
--------------------
test:
    loss: 0.8586
    accuracy: 0.6000
"""
```

In the code above, I organized the code. I defined a main function, and separated the classes and functions with the code for running. I made the logging look more appealing. Also, at the end, I evaluated our model on `test` subset as well. As you can see, `training loss` and `training accuracy` are improving, but `validation loss` and `validation accuracy` might not necessarily.

Better splitting

We have learned how to split our dataset into 3 subsets (`train`, `validation`, `test`), using `random_split` in PyTorch, as below:

```
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


# -------------------[ Load the data ]-------------------
iris = load_iris()

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

# -------------------[ Split the data to 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)
```

Now, let's see how the labels are distributed.

```
label_count = {
    0: 0,
    1: 0,
    2: 0,
}

for data, target in train_data:
    label_count[target.item()] += 1

print(f"train label count: {label_count}")

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

```
train label count: {0: 33, 1: 39, 2: 33}
"""
```

```
label_count = {
    0: 0,
    1: 0,
    2: 0,
}

for data, target in val_data:
    label_count[target.item()] += 1

print(f"validation label count: {label_count}")

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

validation label count: {0: 13, 1: 6, 2: 11}
"""
```

```
label_count = {
    0: 0,
    1: 0,
    2: 0,
}

for data, target in test_data:
    label_count[target.item()] += 1

print(f"test label count: {label_count}")

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

test label count: {0: 4, 1: 5, 2: 6}
"""
```

As you can see, the distribution of the labels isn't perfect. Let's fix that by using the `train_test_split` function in `scikit-learn`.

```
iris = load_iris()

data = iris.data
target = iris.target
```

```python
train_subset, val_subset, train_target, val_target =
 ↪  train_test_split(
    data,
    target,
    test_size=0.3,
    random_state=42,
    stratify=target,
)
val_subset, test_subset, val_target, test_target =
 ↪  train_test_split(
    val_subset,
    val_target,
    test_size=0.33,
    random_state=42,
    stratify=val_target,
)

print("size of each subset: ")
print(f"\ttrain: {train_subset.shape[0]}")
print(f"\tval: {val_subset.shape[0]}")
print(f"\ttest: {test_subset.shape[0]}")

print("target distribution:")
print(f"\ttrain: {np.unique(train_target, return_counts=True)}")
print(f"\tval: {np.unique(val_target, return_counts=True)}")
print(f"\ttest: {np.unique(test_target, return_counts=True)}")

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

size of each subset:
    train: 105
    val: 30
    test: 15
target distribution:
    train: (array([0, 1, 2]), array([35, 35, 35]))
    val: (array([0, 1, 2]), array([10, 10, 10]))
    test: (array([0, 1, 2]), array([5, 5, 5]))
"""
```

In the code above, first, we split our data into 2 subsets (`train`, `val`). As a result, our `train` would be 70 of the data, and `val` would be 30. Then we split the `val` into `val` and `test`. Then, our `val` would be 30 of all data, and `test` would be 30 of all the data. As you can see, we used the `stratify` argument as

well. This argument forces the splitting to have equal distribution. As you can see, now we have `35` samples of each label for `train`, `10` samples of each label for `val`, and `5` samples of each label for `test`. Now, let's make a dataset out of them.

```
train_data = IRISDataset(train_subset, train_target)
val_data = IRISDataset(val_subset, val_target)
test_data = IRISDataset(test_sebset, test_target)
```

I have applied all the changes to train_v3.py.

Standard Scaler

One of the usual techniques in **Deep Learning** is to **Normalize** our data. Right now, every feature has a different **average** and **standard deviation** (**std**). Let's print them out.

```
print(f"Mean of the features:\n\t {train_subset.mean(axis=0)}")
print(f"Standard deviation of the features:\n\t
 ↪  {train_subset.std(axis=0)}")

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

Mean of the features:
     [5.87333333 3.0552381  3.7847619  1.20571429]
Standard deviation of the features:
     [0.85882164 0.45502087 1.77553646 0.77383751]
"""
```

We want to change the **average** of each feature to `0` and their **std** to `1`. To do so, we can use `NormalScaler` in `scikit-learn`.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(train_subset)

train_subset_normalized = scaler.transform(train_subset)
val_subset_normalized = scaler.transform(val_subset)
test_subset_normalized = scaler.transform(test_subset)

print(f"Mean of the features after scaling:")
print(f"\ttrain: {train_subset_normalized.mean(axis=0)}")
print(f"\tval: {val_subset_normalized.mean(axis=0)}")
print(f"\ttest: {test_subset_normalized.mean(axis=0)}")
print(f"Standard deviation of the features after scaling:")
```

```
print(f"\ttrain: {train_subset_normalized.std(axis=0)}")
print(f"\tval: {val_subset_normalized.std(axis=0)}")
print(f"\ttest: {test_subset_normalized.std(axis=0)}")

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

Mean of the features after scaling:
    train: [ 2.38327876e-15 -1.12145742e-15 -1.37456184e-16
 ↪  -6.97854473e-17]
    val: [-0.14360762  0.06174494 -0.04398402 -0.02030696]
    test: [-0.06210059 -0.07744281 -0.06275769 -0.04184464]
Standard deviation of the features after scaling:
    train: [1. 1. 1. 1.]
    val: [0.88306745 0.81063775 0.97257027 0.93027831]
    test: [0.80131426 0.8871022  0.96009651 0.9513319 ]
"""
```

In the code above, I have created an instance of `StandardScaler`. Then, I fitted my `scaler` only with `train_subset`. The reason for that was that we want `validation` and `test` subsets to be unseen. Then I have used the `transform` function to normalize each subset. As you can see, the average of each feature in `train_subset`, is now super close to zero, and the std of each of them is `1`. Because we only trained our scaler on `train_subset`, the average and the std of the `validation` and `test` subsets are not perfect. Now, let's make datasets from our normalized subsets.

```
train_data = IRISDataset(train_subset_normalized, train_target)
val_data = IRISDataset(val_subset_normalized, val_target)
test_data = IRISDataset(test_subset_normalized, test_target)
```

Now, it's time to train and evaluate our model to see what happens. I have applied all the changes in train_v4.py. Let's run train_v4.py.

```
"""
--------
output:
mps
--------------------
epoch: 0
train:
    loss: 1.1541
    accuracy: 0.1238
validation:
    loss: 1.0946
```

```
    accuracy: 0.2667
--------------------
epoch: 1
train:
    loss: 1.0744
    accuracy: 0.3810
validation:
    loss: 1.0350
    accuracy: 0.6667
--------------------
epoch: 2
train:
    loss: 1.0080
    accuracy: 0.6571
validation:
    loss: 0.9773
    accuracy: 0.7000
--------------------
epoch: 3
train:
    loss: 0.9450
    accuracy: 0.7810
validation:
    loss: 0.9198
    accuracy: 0.7000
--------------------
epoch: 4
train:
    loss: 0.8759
    accuracy: 0.8000
validation:
    loss: 0.8617
    accuracy: 0.7333
--------------------
test:
    loss: 0.8406
    accuracy: 0.8000
"""
```

As you can see, right now our evaluation results are not random anymore.

Conclusion

In this tutorial, we have discussed 2 techniques that are being used to enhance our training. First, we explained how to split our data in a way that labels are equally distributed. Then, we introduced `StandardScaler`, which is one of the

most important preprocessing techniques. Although they are not specifically `PyTorch` modules, they are being used in `PyTorch` projects.