Réseaux de neurones en PyTorch

Jérémie Cabessa

Laboratoire DAVID, UVSQ

Ingrédients

L'implémentation d'un réseau de neurones en `PyTorch` requiert les éléments suivants:

- **Datasets:** datasets de train, validation et test sur lesquels entraîner et évaluer le modèle.
- **Dataloaders:** dataloaders de train, validation et test qui permettent de "processer" les data sous forme de batchs.
- **Model:** le réseau de neurones en soi, une classe qui hérite de `torch.nn.Module`.

Ingrédients

L'implémentation d'un réseau de neurones en `PyTorch` requiert les éléments suivants:

- **Datasets:** datasets de train, validation et test sur lesquels entraîner et évaluer le modèle.
- **Dataloaders:** dataloaders de train, validation et test qui permettent de "processer" les data sous forme de batchs.
- **Model:** le réseau de neurones en soi, une classe qui hérite de `torch.nn.Module`.

Ingrédients

L'implémentation d'un réseau de neurones en `PyTorch` requiert les éléments suivants:

- **Datasets:** datasets de train, validation et test sur lesquels entraîner et évaluer le modèle.
- **Dataloaders:** dataloaders de train, validation et test qui permettent de "processer" les data sous forme de batchs.
- **Model:** le réseau de neurones en soi, une classe qui hérite de `torch.nn.Module`.

Ingrédients

- **Loss function:** fonction de coût qui permet d'évaluer l'erreur entre les prédictions du modèle et les valeurs réelles.
- **Optimizer:** méthode d'optimisation pour minimiser la fonction de coût: en général une variante de la "stochastic gradient descent".
- **Training loop:** fonction qui utilisent tous les ingrédients précédents pour entraîner le modèle sur les data d'entraînement.
- **Testing loop:** fonction qui évalue le modèle sur les data de test.

Ingrédients

- **Loss function:** fonction de coût qui permet d'évaluer l'erreur entre les prédictions du modèle et les valeurs réelles.
- **Optimizer:** méthode d'optimisation pour minimiser la fonction de coût: en général une variante de la "stochastic gradient descent".
- **Training loop:** fonction qui utilisent tous les ingrédients précédents pour entraîner le modèle sur les data d'entraînement.
- **Testing loop:** fonction qui évalue le modèle sur les data de test.

Ingrédients

- **Loss function:** fonction de coût qui permet d'évaluer l'erreur entre les prédictions du modèle et les valeurs réelles.
- **Optimizer:** méthode d'optimisation pour minimiser la fonction de coût: en général une variante de la "stochastic gradient descent".
- **Training loop:** fonction qui utilisent tous les ingrédients précédents pour entraîner le modèle sur les data d'entraînement.
- **Testing loop:** fonction qui évalue le modèle sur les data de test.

Ingrédients

- **Loss function:** fonction de coût qui permet d'évaluer l'erreur entre les prédictions du modèle et les valeurs réelles.
- **Optimizer:** méthode d'optimisation pour minimiser la fonction de coût: en général une variante de la "stochastic gradient descent".
- **Training loop:** fonction qui utilisent tous les ingrédients précédents pour entraîner le modèle sur les data d'entraînement.
- **Testing loop:** fonction qui évalue le modèle sur les data de test.

Datasets and Dataloaders

```
# Train, validation and test datasets for MNIST
train_dataset = datasets.MNIST(root="./data", train=True,
                               download=True, transform=ToTensor())

test_dataset = datasets.MNIST(root="./data", train=False,
                              download=True, transform=ToTensor())

train_dataset, val_dataset = random_split(train_dataset, [50000, 10000])

# Dataloaders
train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)

val_dataloader = DataLoader(val_dataset, batch_size=64, shuffle=True)

test_dataloader = DataLoader(test_dataset, batch_size=64, shuffle=True)
```

Custom function

```
def reshape_batch(batch):
    """Flatten 28 x 28 matrices into 768 x 1 vectors."""

    batch_size = batch[0].shape[0]
    batch[0] = batch[0].view(batch_size, -1) # flattening

    return batch
```

Model

```
class Network(nn.Module):

    def __init__(self):

        super(Network, self).__init__()

        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 10)

    def forward(self, x):

        x = self.fc1(x)
        x = nn.ReLU()(x)
        x = self.fc2(x)
        x = nn.ReLU()(x)
        x = self.fc3(x)
        x = nn.ReLU()(x)
        x = self.fc4(x)
        x = nn.ReLU()(x)
        x = self.fc5(x)

        return x

# Put network to GPU if exists
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
network = Network()
network.to(device)
```

Loss function and Optimizer

```
# Loss function
loss = nn.CrossEntropyLoss()

# Optimizer
optimizer = torch.optim.SGD(network.parameters(), lr=0.005)
```

Training loop

```
# Training loop
def train(model, loss=None, optimizer=None,
          train_dataloader=None, val_dataloader=None, nb_epochs=20):

    min_val_loss = torch.inf
    train_losses = []
    val_losses = []

    for e in range(nb_epochs):

        train_loss = 0.0

        # Iterate over train dataloader
        for data, labels in train_dataloader:

            # Transfer data to GPU if available
            if torch.cuda.is_available():
                data, labels = data.cuda(), labels.cuda()
            # Reset gradients to 0
            optimizer.zero_grad()
            # Forward Pass (on reshaped data)
            data, labels = reshape_batch([data, labels])
            preds = model(data)
            # Compute training loss
            current_loss = loss(preds, labels)
            train_loss += current_loss.item()
            # Compute gradients
            current_loss.backward()
            # Update weights
            optimizer.step()
```

Training loop

```
val_loss = 0.0

# Put model in eval mode
model.eval()

# Iterate over validation dataloader
for data, labels in val_dataloader:

    # Transfer data to GPU if available
    if torch.cuda.is_available():
        data, labels = data.cuda(), labels.cuda()
    # Forward Pass (on reshaped data)
    data, labels = reshape_batch([data, labels])
    preds = model(data)
    # Compute validation loss
    current_loss = loss(preds, labels)
    val_loss += current_loss.item()
```

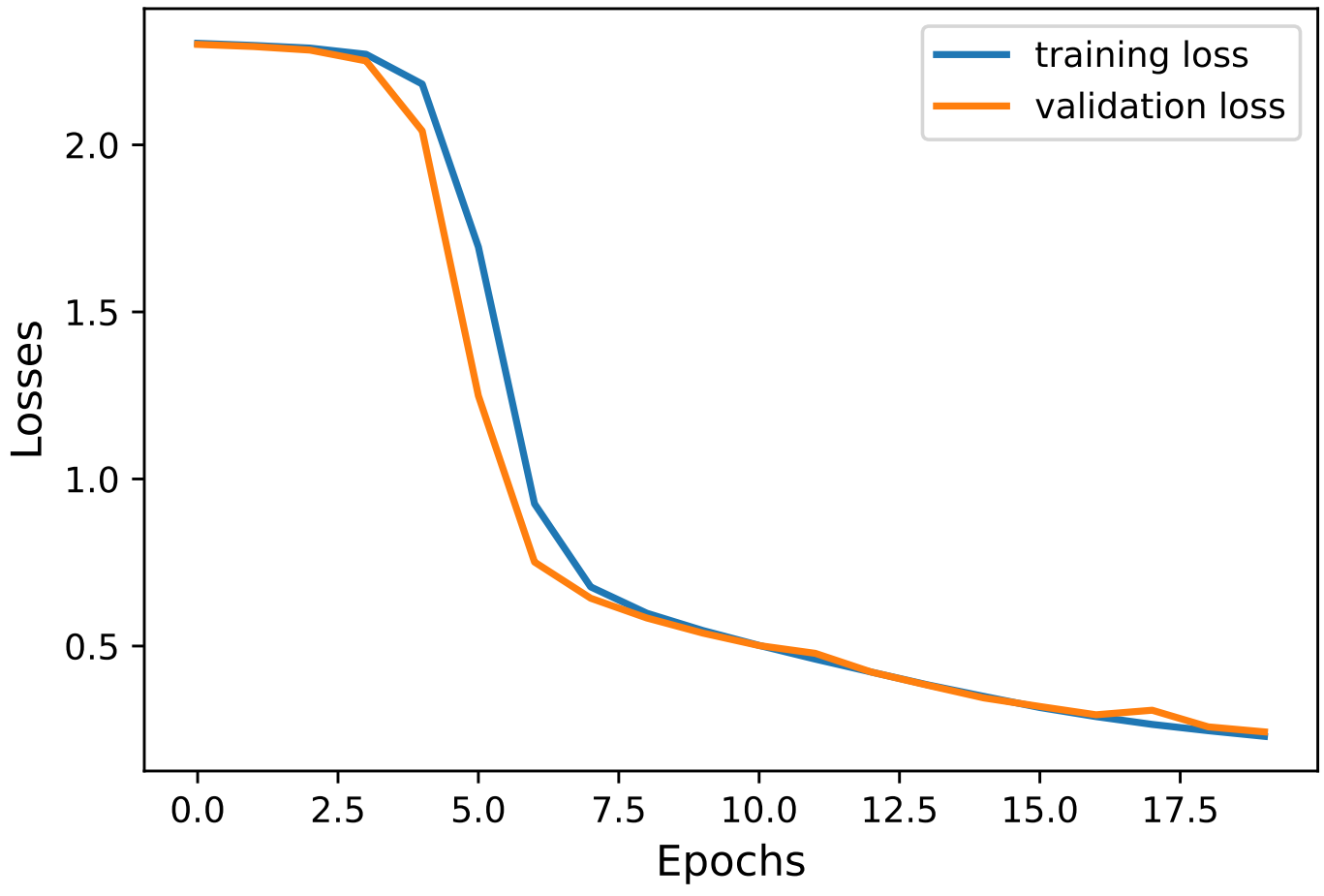
Training loop

```
        # Print and save losses
        print(f"Epoch {e+1}/{nb_epochs}", end="\t")
        print(f"Train Loss: {train_loss/len(train_dataloader):.3f}", end="\t")
        print(f"Validation Loss: {val_loss/len(val_dataloader):.3f}", end="\t")

        train_losses.append(train_loss/len(train_dataloader))
        val_losses.append(val_loss/len(val_dataloader))

        # Save model if val loss decreases
        if val_loss < min_val_loss:

            min_val_loss = val_loss
            torch.save(model.state_dict(), "best_model.pt")

    return train_losses, val_losses
```

Test loop

```
def predict(model, test_dataloader):

    labels_l = []
    preds_l = []

    # Put model in eval mode
    model.eval()

    # Testing loop
    with torch.no_grad():

        for i, batch in enumerate(test_dataloader):

            data, labels = reshape_batch(batch)
            labels_l.extend(labels.tolist())
            preds = model(data)
            preds = torch.argmax(preds, dim=1)
            preds_l.extend(preds.tolist())

    return labels_l, preds_l
```

Results


training loss
validation loss
Losses
2.0
1.5
1.0
0.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

c

Results

```
# Load best model
cwd = os.getcwd()
path = os.path.join(cwd, "best_model.pt")

network = Network()
network.load_state_dict(torch.load(path))
network.eval()

# Compute train and test predictions
train_labels, train_preds = predict(network, train_dataloader)
test_labels, test_preds = predict(network, test_dataloader)

# Get the classification tables
print(classification_report(train_labels, train_preds, digits=4))
print(classification_report(test_labels, test_preds, digits=4))
```

Train results

	precision	recall	f1-score	support
0	0.9621	0.9748	0.9684	4954
1	0.9662	0.9688	0.9675	5576
2	0.9509	0.9323	0.9415	4965
3	0.9492	0.9154	0.9320	5119
4	0.9373	0.9421	0.9397	4884
5	0.9339	0.9248	0.9294	4509
6	0.9702	0.9526	0.9613	4915
7	0.9488	0.9521	0.9504	5218
8	0.9016	0.9367	0.9188	4851
9	0.9102	0.9283	0.9192	5009
accuracy			0.9432	50000
macro avg	0.9430	0.9428	0.9428	50000
weighted avg	0.9435	0.9432	0.9433	50000

Test results

	precision	recall	f1-score	support
0	0.9443	0.9867	0.9651	980
1	0.9719	0.9762	0.9741	1135
2	0.9522	0.9273	0.9396	1032
3	0.9515	0.9327	0.9420	1010
4	0.9370	0.9389	0.9379	982
5	0.9250	0.9126	0.9187	892
6	0.9572	0.9342	0.9456	958
7	0.9542	0.9329	0.9434	1028
8	0.9013	0.9281	0.9145	974
9	0.9062	0.9286	0.9173	1009
accuracy			0.9405	10000
macro avg	0.9401	0.9398	0.9398	10000
weighted avg	0.9408	0.9405	0.9405	10000