pytorch_NN_classif

September 14, 2023

1 Pytorch: simple NN for binary classification

For the PIMA Indians data on diabetes (already encountered in the Basic Course), we will set up a feedforward net to predict the diabetes status.

We follow the usual workflow:

1. Load the data.
2. Define the pytorch model.
3. Define the loss function and optimizer.
4. Run the training loop.
5. Evaluate the model.
6. Make predictions with the learned model.

[1]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

1.1 1. Load the data

Input Variables (X):

- Number of times pregnant
- Plasma glucose concentration at 2 hours in an oral glucose tolerance test
- Diastolic blood pressure (mm Hg)
- Triceps skin fold thickness (mm)
- 2-hour serum insulin (IU/ml)
- Body mass index BMI (weight in kg/(height in m)2)
- Diabetes pedigree function
- Age (years)

Output Variable (y):

- Class label (0 or 1)

[3]:
```
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
X = dataset[:, 0:8]
y = dataset[:, 8]
```

Convert numpy arrays to pytorch tensors.

```
[4]: X = torch.tensor(X, dtype=torch.float32)
     y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)
```

1.2 2. Define the model

We will use

- fully connected layers or dense layers using the `Linear` class in PyTorch.
- ReLU activation functions
- Sigmoid output for the final classification in the output layer (0/1)

Use a "pyramidal" architecture… (see previous Examples).

```
[5]: model = nn.Sequential(
         nn.Linear(8, 12),
         nn.ReLU(),
         nn.Linear(12, 8),
         nn.ReLU(),
         nn.Linear(8, 1),
         nn.Sigmoid() )

     print(model)
```

```
Sequential(
  (0): Linear(in_features=8, out_features=12, bias=True)
  (1): ReLU()
  (2): Linear(in_features=12, out_features=8, bias=True)
  (3): ReLU()
  (4): Linear(in_features=8, out_features=1, bias=True)
  (5): Sigmoid()
)
```

An alternative is to use a class inherited from `nn.Module`

```
[6]: class PimaClassifier(nn.Module):
         def __init__(self):
             super().__init__()
             self.hidden1 = nn.Linear(8, 12)
             self.act1 = nn.ReLU()
             self.hidden2 = nn.Linear(12, 8)
             self.act2 = nn.ReLU()
             self.output = nn.Linear(8, 1)
             self.act_output = nn.Sigmoid()

         def forward(self, x):
             x = self.act1(self.hidden1(x))
             x = self.act2(self.hidden2(x))
             x = self.act_output(self.output(x))
             return x
```

```
model = PimaClassifier()
print(model)
```

```
PimaClassifier(
  (hidden1): Linear(in_features=8, out_features=12, bias=True)
  (act1): ReLU()
  (hidden2): Linear(in_features=12, out_features=8, bias=True)
  (act2): ReLU()
  (output): Linear(in_features=8, out_features=1, bias=True)
  (act_output): Sigmoid()
)
```

1.3 3. Set up loss and optimizer

Since we have a binary classification problem, we must use the binary classification error loss function, `BCELoss`.

For the optimizer, we use standard `Adam`.

[7]:
```
loss_fn = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0005)
```

1.4 4. Train the model.

- Loop over the epochs,
  - loop over batches
    - compute loss
    - compute gradient (backward)
    - step the optimizer

[8]:
```
n_epochs = 100
batch_size = 10

for epoch in range(n_epochs):
    for i in range(0, len(X), batch_size):
        X_batch = X[i:i+batch_size]
        y_pred  = model(X_batch)
        y_batch = y[i:i+batch_size]
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f'At end of epoch {epoch}, loss = {loss}')
```

```
At end of epoch 0, loss = 0.6514326333999634
At end of epoch 1, loss = 0.4459029734134674
At end of epoch 2, loss = 0.4550330638885498
At end of epoch 3, loss = 0.4522790312767029
```

```
At end of epoch 4, loss = 0.4477901756763458
At end of epoch 5, loss = 0.44486045837402344
At end of epoch 6, loss = 0.4469968378543854
At end of epoch 7, loss = 0.4473118484020233
At end of epoch 8, loss = 0.44760721921920776
At end of epoch 9, loss = 0.44560447335243225
At end of epoch 10, loss = 0.4458388388156891
At end of epoch 11, loss = 0.4470572769641876
At end of epoch 12, loss = 0.4462440311908722
At end of epoch 13, loss = 0.44738638401031494
At end of epoch 14, loss = 0.448482871055603
At end of epoch 15, loss = 0.4486386179924011
At end of epoch 16, loss = 0.4485207200050354
At end of epoch 17, loss = 0.4487292170524597
At end of epoch 18, loss = 0.44715577363967896
At end of epoch 19, loss = 0.4507614076137543
At end of epoch 20, loss = 0.45241180062294006
At end of epoch 21, loss = 0.4519551694393158
At end of epoch 22, loss = 0.45340585708618164
At end of epoch 23, loss = 0.4532201886177063
At end of epoch 24, loss = 0.4546703100204468
At end of epoch 25, loss = 0.4543384313583374
At end of epoch 26, loss = 0.45460227131843567
At end of epoch 27, loss = 0.45427581667900085
At end of epoch 28, loss = 0.45509073138237
At end of epoch 29, loss = 0.4552815556526184
At end of epoch 30, loss = 0.45616254210472107
At end of epoch 31, loss = 0.4565679132938385
At end of epoch 32, loss = 0.4559328854084015
At end of epoch 33, loss = 0.4559021592140198
At end of epoch 34, loss = 0.4552902579307556
At end of epoch 35, loss = 0.45252037048339844
At end of epoch 36, loss = 0.4505394697189331
At end of epoch 37, loss = 0.44697901606559753
At end of epoch 38, loss = 0.4421866238117218
At end of epoch 39, loss = 0.43972349166870117
At end of epoch 40, loss = 0.4366328716278076
At end of epoch 41, loss = 0.4346984028816223
At end of epoch 42, loss = 0.43227678537368774
At end of epoch 43, loss = 0.43260636925697327
At end of epoch 44, loss = 0.4313752055168152
At end of epoch 45, loss = 0.4310115575790405
At end of epoch 46, loss = 0.4309276342391968
At end of epoch 47, loss = 0.4298495650291443
At end of epoch 48, loss = 0.42906877398490906
At end of epoch 49, loss = 0.4287033677101135
At end of epoch 50, loss = 0.42760005593299866
At end of epoch 51, loss = 0.42689645290374756
```

```
At end of epoch 52, loss = 0.4261499047279358
At end of epoch 53, loss = 0.42485707998275757
At end of epoch 54, loss = 0.4237571060657501
At end of epoch 55, loss = 0.42292535305023193
At end of epoch 56, loss = 0.4225464463233948
At end of epoch 57, loss = 0.4221575856208801
At end of epoch 58, loss = 0.4213806986808777
At end of epoch 59, loss = 0.4214887320995331
At end of epoch 60, loss = 0.4206361472606659
At end of epoch 61, loss = 0.42072466015815735
At end of epoch 62, loss = 0.42008012533187866
At end of epoch 63, loss = 0.41975465416908264
At end of epoch 64, loss = 0.4193488359451294
At end of epoch 65, loss = 0.4189695417881012
At end of epoch 66, loss = 0.4194849133491516
At end of epoch 67, loss = 0.419280081987381
At end of epoch 68, loss = 0.41929611563682556
At end of epoch 69, loss = 0.4180874824523926
At end of epoch 70, loss = 0.4179556965827942
At end of epoch 71, loss = 0.4178284704685211
At end of epoch 72, loss = 0.4190225303173065
At end of epoch 73, loss = 0.4191664457321167
At end of epoch 74, loss = 0.4188390076160431
At end of epoch 75, loss = 0.4191761314868927
At end of epoch 76, loss = 0.4176833927631378
At end of epoch 77, loss = 0.41882672905921936
At end of epoch 78, loss = 0.41919586062431335
At end of epoch 79, loss = 0.41920986771583557
At end of epoch 80, loss = 0.4184538722038269
At end of epoch 81, loss = 0.4185410141944885
At end of epoch 82, loss = 0.4179275631904602
At end of epoch 83, loss = 0.4187832772731781
At end of epoch 84, loss = 0.41979092359542847
At end of epoch 85, loss = 0.4191528856754303
At end of epoch 86, loss = 0.41897183656692505
At end of epoch 87, loss = 0.4188595414161682
At end of epoch 88, loss = 0.4189137816429138
At end of epoch 89, loss = 0.4185986816883087
At end of epoch 90, loss = 0.4165605902671814
At end of epoch 91, loss = 0.41700321435928345
At end of epoch 92, loss = 0.41613277792930603
At end of epoch 93, loss = 0.413994163274765
At end of epoch 94, loss = 0.4150134027004242
At end of epoch 95, loss = 0.4148235023021698
At end of epoch 96, loss = 0.41368621587753296
At end of epoch 97, loss = 0.41401827335357666
At end of epoch 98, loss = 0.4144798219203949
At end of epoch 99, loss = 0.41475769877433777
```

1.5 5. Evaluate the model precision

Just evaluate on all the training data...not very satisfactory.

[9]:
```
# compute accuracy (no_grad is optional)
with torch.no_grad():
    y_pred = model(X)

accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")
```

```
Accuracy 0.7421875
```

1.6 6. Make predictions with model

[10]:
```
# make class predictions with the model
predictions = (model(X) > 0.5).int()
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

```
[6.0, 148.0, 72.0, 35.0, 0.0, 33.599998474121094, 0.6269999742507935, 50.0] => 1
(expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.600000381469727, 0.35100001096725464, 31.0] => 0
(expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.299999237060547, 0.671999990940094, 32.0] => 1
(expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.100000381469727, 0.16699999570846558, 21.0] =>
0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.099998474121094, 2.2880001068115234, 33.0] =>
1 (expected 1)
```

1.7 Conclusions

1. In spite of relatively large learning error, and no real cross-validation, the classifier attained 100% accuracy on the "test" set.
2. The model should be redone with
   - complete EDA (see previous Example)
   - proper trian/test split
   - tuning of architecture
   - tuning of optimizer
   - rigorous reporting.

[ ]:
```

```