Pytorch classification

Binary classification: one of two options ... multiclass classification: one of more than two options ...

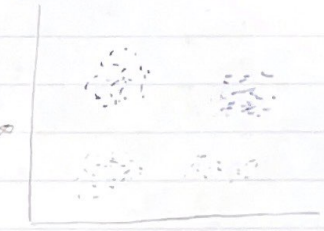
Multi-label classification: one or more of more than two options

hyperparameter:	Binary classification: B	Multiclass classification: M
input layer shape:	#of features	
hidden layers:	1-00	
neurons per hidden layer:	generally 10-512	
output layer shape:	(B) 1	(M) 1 per class
hidden layer activation:	usually ReLU	
output activation:	(B) torch.sigmoid	(M) torch.softmax
Loss function:	(B) torch.nn.BCELoss (with logits)	(M) torch.nn.CrossEntropyLoss (Regression) torch.nn.L1Loss or torch.nn.MSELoss
Optimizer:	torch.optim.SGD() or torch.optim.Adam()	

evaluation methods: on from torchmetrics import Accuracy

torchmetrics.Accuracy() ... torchmetrics.Precision() ... torchmetrics.Recall() ... torchmetrics.F1Score() ... torchmetrics.ConfusionMatrix

02-8 Multiclass Pytorch model:



```
#import dependencies
#set hyperparameters for data creation
#create multiclass data
#turn data into tensors
#split into train and test sets
#plot data
device = "cuda" if torch.cuda.is_available() else "cpu"
from torch import nn
class BlobModel(nn.Module):
    def __init__(self, input_features, output_features, hidden_units=8):
        super().__init__()
        self.linear_layer_stack = nn.Sequential(
            nn.Linear(in_features=input_features, out_features=hidden_units),
            nn.Linear(in_features=hidden_units, out_features=hidden_units),
            nn.Linear(in_features=hidden_units, out_features=output_features),
        )
```

```
    def forward(self, x):
        return self.linear_layer_stack(x)

model_4 = BlobModel(input_features=____, output_features=____, hidden_units=____).to(device)

loss_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model_4.parameters(), lr=0.1)

torch.manual_seed(42) ... epochs=100

X_blob_train, y_blob_train = X_blob_train.to(device), y_blob_train.to(device)

X_blob_test, y_blob_test = X_blob_test.to(device), y_blob_test.to(device)

for epoch in range(epochs):
    model_4.train()
    y_logits = model_4(X_blob_train)
    y_pred = torch.softmax(y_logits, dim=1).argmax(dim=1)      # logits -> pred probs -> pred labels
    loss = loss_fn(y_logits, y_blob_train)
    acc = accuracy_fn(y_true=y_blob_train, y_pred=y_pred)      # Homemade fn
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    model_4.eval()
    with torch.inference_mode():
        test_logits = model_4(X_blob_test)                     } forward pass
        test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)   }
        test_loss = loss_fn(test_logits, y_blob_test)
        test_acc = accuracy_fn(y_true=y_blob_test, y_pred=test_pred)
    if epoch % 10 == 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Acc: {acc:.2f}% | Test loss: {test_loss:.5f}, Test Acc: {test_acc...
```

8.6 make and evaluate predictions

```
model_4.eval()
with torch.inference_mode():
    y_logits = model_4(X_blob_test)
y_pred_probs = torch.softmax(y_logits, dim=1)
y_preds = y_pred_probs.argmax(dim=1)
print(f"Predictions: {y_preds[:10]}\nLabels: {y_blob_test[:10]}")
print(f"Test accuracy: {accuracy_fn(y_true=y_blob_test, y_pred=y_preds)}%")

#plot decision boundary
```