PyTorch Neural Network Classification

Problem type	What is it?	Example
Binary classification	Target can be one of two options, e.g. yes or no	Predict whether or not someone has heart disease based on their health parameters.
Multi-class classification	Target can be one of more than two options	Decide whether a photo of is of food, a person or a dog.
Multi-label classification	Target can be assigned more than one option	Predict what categories should be assigned to a Wikipedia article (e.g. mathematics, science & philosohpy).

Topic	Contents
0. Architecture of a classification neural network	Neural networks can come in almost any shape or size, but they typically follow a similar floor plan.
1. Getting binary classification data ready	Data can be almost anything but to get started we're going to create a simple binary classification dataset.
2. Building a PyTorch classification model	Here we'll create a model to learn patterns in the data, we'll also choose a **loss function**, **optimizer** and build a **training loop** specific to classification.
3. Fitting the model to data (training)	We've got data and a model, now let's let the model (try to) find patterns in the (**training**) data.
4. Making predictions and evaluating a model (inference)	Our model's found patterns in the data, let's compare its findings to the actual (**testing**) data.
5. Improving a model (from a model perspective)	We've trained an evaluated a model but it's not working, let's try a few things to improve it.
6. Non-linearity	So far our model has only had the ability to model straight lines, what about non-linear (non-straight) lines?
7. Replicating non-linear functions	We used **non-linear functions** to help model non-linear data, but what do these look like?
8. Putting it all together with multi-class classification	Let's put everything we've done so far for binary classification together with a multi-class classification problem.

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape (`in_features`)	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, generally 10 to 512	Same as binary classification
Output layer shape (`out_features`)	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden layer activation	Usually ReLU (rectified linear unit) but can be many others	Same as binary classification
Output activation	Sigmoid (`torch.sigmoid` in PyTorch)	Softmax (`torch.softmax` in PyTorch)
Loss function	Binary crossentropy (`torch.nn.BCELoss` in PyTorch)	Cross entropy (`torch.nn.CrossEntropyLoss` in PyTorch)
Optimizer	SGD (stochastic gradient descent), Adam (see `torch.optim` for more options)	Same as binary classification

1. Make classification data and get it ready

1.1 Input and output shapes

1.2 Turn data into tensors and create train and test splits

Specifically, we'll need to:

1. Turn our data into tensors (right now our data is in NumPy arrays and PyTorch prefers to work with PyTorch tensors).
2. Split our data into training and test sets (we'll train a model on the training set to learn the patterns between `X` and `y` and then evaluate those learned patterns on the test dataset).

a. *# Turn data into tensors*

b. *# Otherwise this causes issues with computations later on*

c. **import** torch

d. X **=** torch**.**from_numpy(X)**.**type(torch**.**float)

e. y **=** torch**.**from_numpy(y)**.**type(torch**.**float)

f. *# View the first five samples*

g. X[: 5], y[: 5]

h. *# Split data into train and test sets*

i. ***from*** *sklearn.model_selection* ***import*** *train_test_split*

j. *X_train, X_test, y_train, y_test* **=** *train_test_split(X,*

y,

test_size=0.2, # 20% test, 80% train

random_state=42) # make the random split reproducible

k. *len(X_train), len(X_test), len(y_train), len(y_test)*

2. Building a model

We'll break it down into a few parts.

1. Setting up device agnostic code (so our model can run on CPU or GPU if it's available).
2. Constructing a model by subclassing `nn.Module`.
3. Defining a loss function and optimizer.
4. Creating a training loop

- *# Standard PyTorch imports*
- ***import*** *torch*
- ***from*** *torch* ***import*** *nn*
-
- *# Make device agnostic code*
- *device* **=** *"cuda"* ***if*** *torch.cuda.is_available()* ***else*** *"cpu"*

Let's create a model class that:

1. Subclasses `nn.Module` (almost all PyTorch models are subclasses of `nn.Module`).
2. Creates 2 `nn.Linear` layers in the constructor capable of handling the input and output shapes of `X` and `y`.
3. Defines a `forward()` method containing the forward pass computation of the model.
4. Instantiates the model class and sends it to the target `device`.

```
# 1. Construct a model class that subclasses nn.Module
class CircleModelV0(nn.Module):
    def __init__(self):
        super().__init__()
        # 2. Create 2 nn.Linear layers capable of handling X and y input and output shapes
        self.layer_1 = nn.Linear(in_features=2, out_features=5) # takes in 2 features (X), produces 5 features
        self.layer_2 = nn.Linear(in_features=5, out_features=1) # takes in 5 features, produces 1 feature (y)
    # 3. Define a forward method containing the forward pass computation
    def forward(self, x):
        # Return the output of layer_2, a single feature, the same shape as y
        return self.layer_2(self.layer_1(x)) # computation goes through layer_1 first then the output of layer_1 goes through layer_2
# 4. Create an instance of the model and send it to target device
model_0 = CircleModelV0().to(device)
```

You can also do the same as above using `nn.Sequential`.

`nn.Sequential` performs a forward pass computation of the input data through the layers in the order they appear.

1. ***# Replicate CircleModelV0 with nn.Sequential***
2. *model_0 = nn.Sequential(*

   nn.Linear(in_features=2, out_features=5),

   nn.Linear(in_features=5, out_features=1)

).to(device)

2.1 Setup loss function and optimizer

Loss function/Optimizer	Problem type	PyTorch Code
Stochastic Gradient Descent (SGD) optimizer	Classification, regression, many others.	`torch.optim.SGD()`
Adam Optimizer	Classification, regression, many others.	`torch.optim.Adam()`
Binary cross entropy loss	Binary classification	`torch.nn.BCELossWithLogits` or `torch.nn.BCELoss`
Cross entropy loss	Mutli-class classification	`torch.nn.CrossEntropyLoss`
Mean absolute error (MAE) or L1 Loss	Regression	`torch.nn.L1Loss`
Mean squared error (MSE) or L2 Loss	Regression	`torch.nn.MSELoss`

PyTorch has two binary cross entropy implementations:

1. `torch.nn.BCELoss()` - Creates a loss function that measures the binary cross entropy between the target (label) and input (features).

2. `torch.nn.BCEWithLogitsLoss()` - This is the same as above except it has a sigmoid layer (`nn.Sigmoid`) built-in (we'll see what this means soon).
    - **# Create a loss function**
    - **# loss_fn = nn.BCELoss() # BCELoss = no sigmoid built-in**
    - loss_fn = nn.BCEWithLogitsLoss() **# BCEWithLogitsLoss = sigmoid built-in**

      # Create an optimizer

    - optimizer = torch.optim.SGD(params=model_0.parameters(),

      lr=0.1)

Now let's also create an **evaluation metric**.

If a loss function measures how *wrong* your model is, I like to think of evaluation metrics as measuring how *right* it is.

There are several evaluation metrics that can be used for classification problems but let's start out with **accuracy**.

Accuracy can be measured by dividing the total number of correct predictions over the total number of predictions.

- ***# Calculate accuracy (a classification metric)***
- *def accuracy_fn(y_true, y_pred):*
- *correct = torch.eq(y_true, y_pred).sum().item()* ***# torch.eq() calculates where two tensors are equal***
- *acc = (correct / len(y_pred)) * 100*

3. Train model

3.1 Going from raw model outputs to predicted labels (logits -> prediction probabilities -> prediction labels)

The *raw outputs* (unmodified) of this equation (y) and in turn, the raw outputs of our model are often referred to as **logits**.

We'd like some numbers that are comparable to our truth labels.

To get our model's raw outputs (logits) into such a form, we can use the sigmoid activation function.

- ***# Use sigmoid on model logits***
- *y_pred_probs* **=** *torch.sigmoid(y_logits)*
- *y_pred_probs*

Okay, it seems like the outputs now have some kind of consistency (even though they're still random).

They're now in the form of **prediction probabilities** (I usually refer to these as `y_pred_probs`), in other words, the values are now how much the model thinks the data point belongs to one class or another.

In our case, since we're dealing with binary classification, our ideal outputs are 0 or 1.

So these values can be viewed as a decision boundary.

The closer to 0, the more the model thinks the sample belongs to class 0, the closer to 1, the more the model thinks the sample belongs to class 1.

More specificially:

- If `y_pred_probs` >= 0.5, `y=1` (class 1)
- If `y_pred_probs` < 0.5, `y=0` (class 0)

To turn our prediction probabilities in prediction labels, we can round the outputs of the sigmoid activation function.

- ***# Find the predicted labels (round the prediction probabilities)***
- *y_preds = torch.round(y_pred_probs)*
- ***# In full***
- *y_pred_labels = torch.round(torch.sigmoid(model_0(X_test.to(device))[:5]))*
- ***# Check for equality***
- *print(torch.eq(y_preds.squeeze(), y_pred_labels.squeeze()))*
- ***# Get rid of extra dimension***
- *y_preds.squeeze()*

3.2 Building a training and testing loop

4. Make predictions and evaluate the model

5. Improving a model (from a model perspective)

Model improvement technique*	What does it do?
Add more layers	Each layer *potentially* increases the learning capabilities of the model with each layer being able to learn some kind of new pattern in the data, more layers is often referred to as making your neural network *deeper*.
Add more hidden units	Similar to the above, more hidden units per layer means a *potential* increase in learning capabilities of the model, more hidden units is often referred to as making your neural network *wider*.
Fitting for longer (more epochs)	Your model might learn more if it had more opportunities to look at the data.
Changing the activation functions	Some data just can't be fit with only straight lines (like what we've seen), using non-linear activation functions can help with this (hint, hint).
Change the learning rate	Less model specific, but still related, the learning rate of the optimizer decides how much a model should change its parameters each step, too much and the model overcorrects, too little and it doesn't learn enough.
Change the loss function	Again, less model specific but still important, different problems require different loss functions. For example, a binary cross entropy loss function won't work with a multi-class classification problem.
Use transfer learning	Take a pretrained model from a problem domain similar to yours and adjust it to your own problem. We cover transfer learning in notebook 06.

6. The missing piece: non-linearity

6.1 Recreating non-linear data (red and blue circles)

6.2 Building a model with non-linearity

PyTorch has a bunch of ready-made non-linear activation functions that do similiar but different things.

One of the most common and best performing is ReLU (rectified linear-unit, `torch.nn.ReLU()`).

Rather than talk about it, let's put it in our neural network between the hidden layers in the forward pass and see what happens.

- ***# Build model with non-linear activation function***
- *from torch import nn*
- *class CircleModelV2(nn.Module):*
- *def __init__(self):*
- *super().__init__()*
- *self.layer_1 = nn.Linear(in_features=2, out_features=10)*
- *self.layer_2 = nn.Linear(in_features=10, out_features=10)*
- *self.layer_3 = nn.Linear(in_features=10, out_features=1)*
- *self.relu = nn.ReLU() # <- add in ReLU activation function*

  # Can also put sigmoid in the model

  # This would mean you don't need to use it on the predictions

  # self.sigmoid = nn.Sigmoid()

- *def forward(self, x):*

  # Intersperse the ReLU activation function between layers

  return self.layer_3(self.relu(self.layer_2(self.relu(self.layer_1(x)))))

- *model_3 = CircleModelV2().to(device)*
- *print(model_3)*