PyTorch Neural Network Classification

# What is a classification problem?

A [classification problem](https://en.wikipedia.org/wiki/Statistical_classification) involves predicting whether something is one thing or another.

|  |  |  |
| --- | --- | --- |
| **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). |

# What we are going to cover

|  |  |
| --- | --- |
| **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. |

# Technologies / libraries used for this session:

1. pandas -> for transforming raw data into data frames
2. scikit learn( sklearn ) 🡪 to get some sample public data and also for data visualization , also for splitting data into training and testing data sets
3. matplotlib 🡪 for plotting data graphically and data scattering
4. torch 🡪 tensors creation and neural network classes for creating layered neural networks

# 0. Architecture of a classification neural network

|  |  |  |
| --- | --- | --- |
| **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. depends on how many output labels we are expecting . ex : for any photo if we are looking for RGB values then output layer will have 3 features) |
| **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)](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.CrossEntropyLoss.html) |
| **Optimizer** | SGD (stochastic gradient descent) | Same as binary classification |

# 1.a. Make some classification data

We are using scikit learn here to get some data.

we are going to use make\_circles() method from scikit learn to generate 2 circles with different coloured dots.

from sklearn.datasets import make\_circles

# Make 1000 samples

n\_samples = 1000

# Create circles

X, y = make\_circles(n\_samples,

                    noise=0.03, # some noise for randomness

                    random\_state=42) # keep random state

X here is features and y are labels

Creating a data frame using pandas:

X values are getting partitioned into X1 and X2 below. For each x1,x2 values there is a Y value defined ( either 0,1)

# Make DataFrame of circle data

import pandas as pd

circles = pd.DataFrame({"X1": X[:, 0],

    "X2": X[:, 1],

    "label": y

})

circles.head(10)

Using matplot lib to plot the data set.

# Visualize with a plot

import matplotlib.pyplot as plt

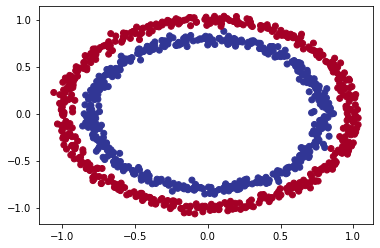
plt.scatter(x=X[:, 0],

            y=X[:, 1],

            c=y,

            cmap=plt.cm.RdYlBu);

1. less noise , more accurate and dots are more closer,
2. more noise , less accurate and dots are scattered around



# 1.b Input and output Shapes

First see what the shape of your data set ( its important to understand the shapes and dimension of your data set since we will be dealing with tensor operations)

# Check the shapes of our features and labels

X.shape, y.shape

# View the first example of features and labels

X\_sample = X[0]

y\_sample = y[0]

print(f"Values for one sample of X: {X\_sample} and the same for y: {y\_sample}")

print(f"Shapes for one sample of X: {X\_sample.shape} and the same for y: {y\_sample.shape}")

Here the X shape will be (1000,2) and y shape will be (1000,)

( 1000,2 ) means 1000 data sets for X but its in pairs like [(x1,x2),(x3,x4)…..) and for y its in scaler( single number , 1 dimension) like [y1,y2,y3…..]

1000 🡪 count of data

2 🡪 numbers of features for the data( and it can vary and increase based on your data set)

Converting data into tensors

# Turn data into tensors

# Otherwise this causes issues with computations later on

import torch

X = torch.from\_numpy(X).type(torch.float)

y = torch.from\_numpy(y).type(torch.float)

# View the first five samples

X[:5], y[:5]

Splitting data into training and test sets

# Split data into train and test sets

from sklearn.model\_selection import train\_test\_split

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

len(X\_train), len(X\_test), len(y\_train), len(y\_test)

# 2.Building a model

**Using GPU for neural networks**

Select your device first ( ie. CPU or GPU)

If you have a GPU, you can use that for running these models once created

The same will work on CPU (since the code is agnostic to which device its running on)

**Steps for model setup**

1. Subclasses nn.Module
2. Creates 2 nn.Linear layers in the constructor capable of handling the input and output shapes of X and y.
   1. Ie. Output of x , will be input of y
   2. In features 🡪 params going in the layer( since x has 2 params here , so keeping it 2)
   3. Out features 🡪 params going out from layer (This layer turns the input data from having 2 features to 5 features.)
      1. This is arbitrary number , also denotes no. of neurons per layer
      2. What’s the benefit of having out features ??
         1. This allows the model to learn patterns from 5 numbers rather than just 2 numbers, *potentially* leading to better outputs.
3. Defines a forward() method containing the forward pass computation of the model.

# 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

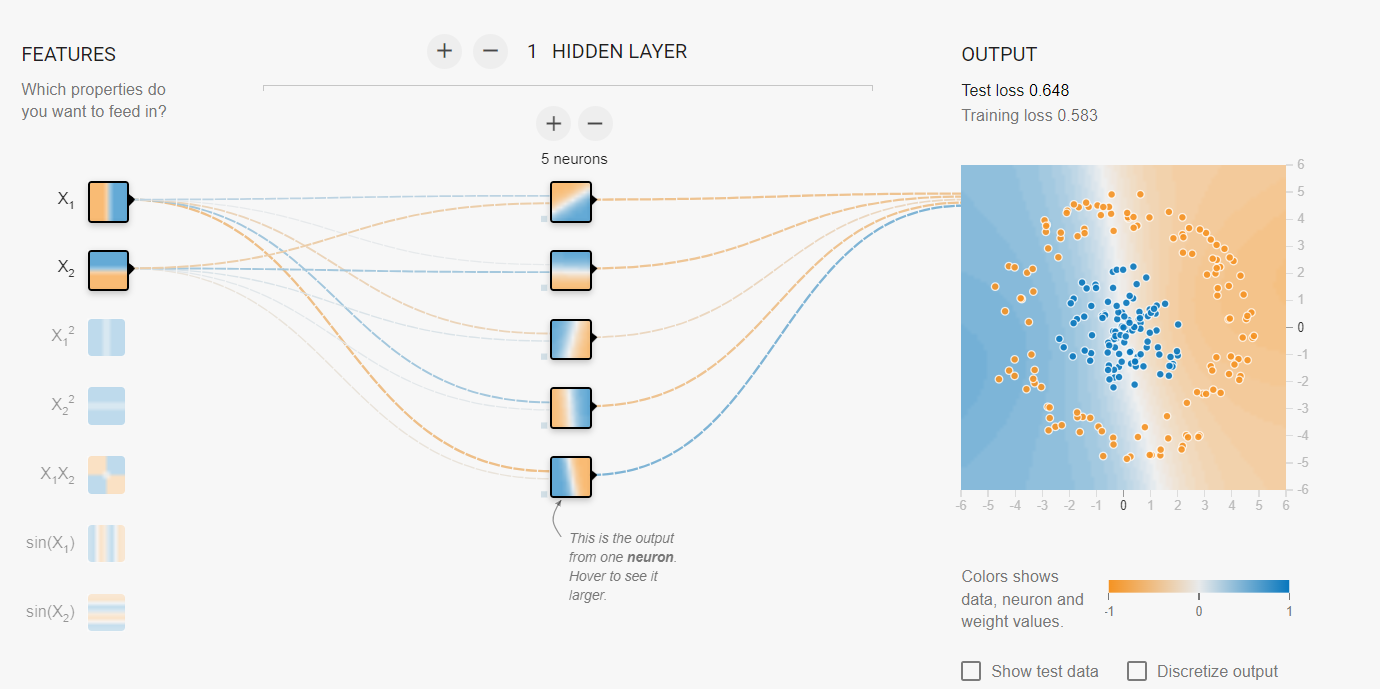
model\_0 = CircleModelV0()

**Tensor flow play ground ( for visualization)**

[playground.tensorflow.org](https://r.search.yahoo.com/_ylt=AwrKBzcu0N9kY5sULXG7HAx.;_ylu=Y29sbwNzZzMEcG9zAzEEdnRpZAMEc2VjA3Ny/RV=2/RE=1692418222/RO=10/RU=https%3a%2f%2fplayground.tensorflow.org%2f/RK=2/RS=RiIx9122z3SOJvNxVJ.bnTAOYQQ-)

2 input layers

1 hidden layer ( 5 neorons in hidden layer)



## nn.Sequential

The same model above can be constructed using nn.sequential module. This is class where output for one layer is given to another layer ( so data passes sequentially from one layer to another). This is similar to the custom class we made above (torch has an inbuilt class )

# Replicate CircleModelV0 with nn.Sequential

model\_0 = nn.Sequential(

    nn.Linear(in\_features=2, out\_features=5),

    nn.Linear(in\_features=5, out\_features=1)

)

model\_0

## #Make prediction with model

untrained\_preds = model\_0(X\_test)

print(f"Length of predictions: {len(untrained\_preds)}, Shape: {untrained\_preds.shape}")

print(f"Length of test samples: {len(y\_test)}, Shape: {y\_test.shape}")

print(f"\nFirst 10 predictions:\n{untrained\_preds[:10]}")

print(f"\nFirst 10 test labels:\n{y\_test[:10]}")

# 2.a Setup loss function and optimizer

Now based on the type of problem we define the loss function.

For instance, for regression , we chose MAE ( Mean Absolute error) and MSE ( Mean Squared Error).

For classification ( binary classification ) , we need binary cross entropy as the loss function.

|  |  |  |
| --- | --- | --- |
| **Loss function/Optimizer** | **Problem type** | **PyTorch Code** |
| Stochastic Gradient Descent (SGD) optimizer | Classification, regression, many others. | [torch.optim.SGD()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.optim.SGD.html) |
| Adam Optimizer | Classification, regression, many others. | [torch.optim.Adam()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.optim.Adam.html) |
| Binary cross entropy loss | Binary classification | torch.nn.BCELossWithLogits or torch.nn.BCELoss |
| Cross entropy loss | Mutli-class classification | [torch.nn.CrossEntropyLoss](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.CrossEntropyLoss.html) |
| Mean absolute error (MAE) or L1 Loss | Regression | [torch.nn.L1Loss](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.L1Loss.html) |
| Mean squared error (MSE) or L2 Loss | Regression | [torch.nn.MSELoss](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.MSELoss.html%23torch.nn.MSELoss) |

PyTorch has two binary cross entropy implementations:

* [torch.nn.BCELoss()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.BCELoss.html) - Creates a loss function that measures the binary cross entropy between the target (label) and input (features).
* [torch.nn.BCEWithLogitsLoss()](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.BCEWithLogitsLoss.html) - This is the same as above except it has a sigmoid layer ([nn.Sigmoid](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.Sigmoid.html" \t "_blank)) built-in

**In pytorch ,** [**torch.nn.BCEWithLogitsLoss()**](https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html)**is more numerically stable than using torch.nn.BCELoss() after a nn.Sigmoid layer.**

# 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)

Here comes the concept of evaluation metric

An evaluation metric can be used to offer another perspective on how your model is going.

If a loss function measures how wrong your model is, evaluation metrics as measuring how right it is.

Accuracy in evaluation metric

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

    return acc

# 3. Train Model

Recap on the pytorch training loop

1. **Forward pass** - The model goes through all of the training data once, performing its forward() function calculations (model(x\_train)).
2. **Calculate the loss** - The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are (loss = loss\_fn(y\_pred, y\_train).
3. **Zero gradients** - The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step (optimizer.zero\_grad()).
4. **Perform backpropagation on the loss** - Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with requires\_grad=True). This is known as **backpropagation**, hence "backwards" (loss. Backward()).
5. **Step the optimizer (gradient descent)** - Update the parameters with requires grad=True with respect to the loss gradients in order to improve them (optimizer.step()).

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

Logit 🡪 output of raw inputs (without sigmoid/ activation function) ( raw output)

Before training the model, let’s see what is the raw output using the raw inputs. The output which we will get would be random , since the mode is untrained

# View the first 5 outputs of the forward pass on the test data

y\_logits = model\_0(X\_test)[:5]

y\_logits

tensor([[-0.4279], [-0.3417], [-0.5975], [-0.3801], [-0.5078]]

Above raw output value are any random values . We need to convert them into ( 0,1) so that we can classify the output value.

How do we convert them into 0 or 1 values?? 🡪 By using a Sigmoid function

# Use sigmoid on model logits

y\_pred\_probs = torch.sigmoid(y\_logits)

y\_pred\_probs

They're now in the form of **prediction probabilities** (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 specifically:

* 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 build training and testing Loop**

torch.manual\_seed(42)

# Set the number of epochs

epochs = 100

# Put data to target device

X\_train, y\_train = X\_train.to(device), y\_train.to(device)

X\_test, y\_test = X\_test.to(device), y\_test.to(device)

# Build training and evaluation loop

for epoch in range(epochs):

    ### Training

    model\_0.train()

    # 1. Forward pass (model outputs raw logits)

    y\_logits = model\_0(X\_train).squeeze() # squeeze to remove extra `1` dimensions, this won't work unless model and data are on same device

    y\_pred = torch.round(torch.sigmoid(y\_logits)) # turn logits -> pred probs -> pred labls

    # 2. Calculate loss/accuracy

    # loss = loss\_fn(torch.sigmoid(y\_logits), # Using nn.BCELoss you need torch.sigmoid()

    #                y\_train)

    loss = loss\_fn(y\_logits, # Using nn.BCEWithLogitsLoss works with raw logits

                   y\_train)

    acc = accuracy\_fn(y\_true=y\_train,

                      y\_pred=y\_pred)

    # 3. Optimizer zero grad

    optimizer.zero\_grad()

    # 4. Loss backwards

    loss.backward()

    # 5. Optimizer step

    optimizer.step()

    ### Testing

    model\_0.eval()

    with torch.inference\_mode():

        # 1. Forward pass

        test\_logits = model\_0(X\_test).squeeze()

        test\_pred = torch.round(torch.sigmoid(test\_logits))

        # 2. Caculate loss/accuracy

        test\_loss = loss\_fn(test\_logits,

                            y\_test)

        test\_acc = accuracy\_fn(y\_true=y\_test,

                               y\_pred=test\_pred)

    # Print out what's happening every 10 epochs

    if epoch % 10 == 0:

        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss: {test\_loss:.5f}, Test acc: {test\_acc:.2f}%")

# 4 . Evaluate the model and Prediction Visualization

We will use plot library to visualize how the model predictions are happening.

# Plot decision boundaries for training and test sets

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

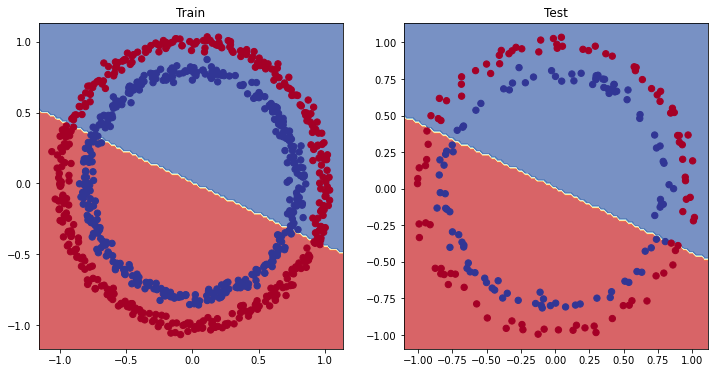
plt.title("Train")

plot\_decision\_boundary(model\_0, X\_train, y\_train)

plt.subplot(1, 2, 2)

plt.title("Test")

plot\_decision\_boundary(model\_0, X\_test, y\_test)



What’s the problem in the above prediction values?

The problem which we are trying to solve is a classification problem and the data set is not linear. But the model that we have built is having linear neural network layers within.

It's currently trying to split the red and blue dots using a straight line...

That explains the 50% accuracy. Since our data is circular, drawing a straight line can at best cut it down the middle.

In machine learning terms, our model is **underfitting**, meaning it's not learning predictive patterns from the data.

# 5 . Model Improvisation (from model perspective)

Let's try to fix our model's underfitting problem.

Focusing specifically on the model (not the data), there are a few ways we could do this.

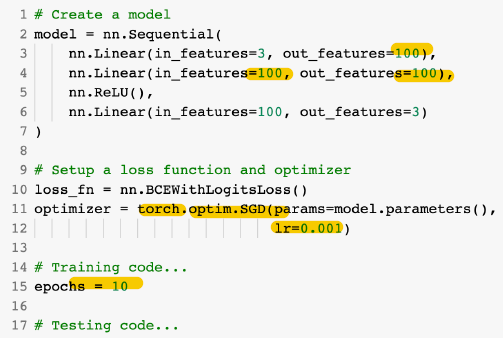
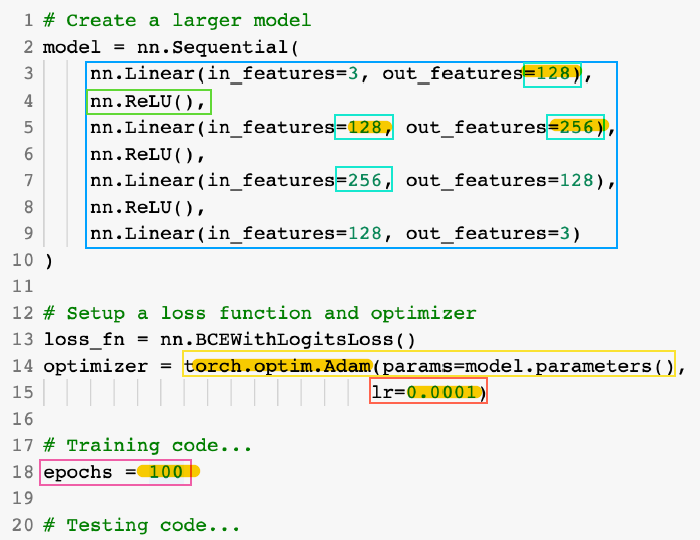
|  |  |
| --- | --- |
| **Model improvement technique** | **What does it do?** |
| Add more layers | Each layer *potentially* increases the learning capabilities of the model with each layer being able to learn some kind of new pattern in the data, more layers is often referred to as making your neural 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. |
| 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.](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fwww.learnpytorch.io%2F06_pytorch_transfer_learning%2F) |

**Why are these called “from respect to Model”?**

1) cos these directly impact how model is functioning.

2) No change in data / data quality.

3) these parameters which are getting changed are also called “hyper parameters”

## **5 .1 Change in hidden layers, hidden units , epochs**

* Hidden Units ( 5 🡪 10)
* Number of layers ( 2 🡪 3)
* Number of epochs ( 100 🡪 1000)

class CircleModelV1(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) # extra layer

        self.layer\_3 = nn.Linear(in\_features=10, out\_features=1)

    def forward(self, x): # note: always make sure forward is spelt correctly!

        # Creating a model like this is the same as below, though below

        # generally benefits from speedups where possible.

        # z = self.layer\_1(x)

        # z = self.layer\_2(z)

        # z = self.layer\_3(z)

        # return z

        return self.layer\_3(self.layer\_2(self.layer\_1(x)))

model\_1 = CircleModelV1().to(device)

model\_1

#loss function

# loss\_fn = nn.BCELoss() # Requires sigmoid on input

loss\_fn = nn.BCEWithLogitsLoss() # Does not require sigmoid on input

optimizer = torch.optim.SGD(model\_1.parameters(), lr=0.1)

# set the manual seed

torch.manual\_seed(42)

# training and testing loop

epochs = 1000 # Train for longer

# Put data to target device

X\_train, y\_train = X\_train.to(device), y\_train.to(device)

X\_test, y\_test = X\_test.to(device), y\_test.to(device)

for epoch in range(epochs):

    ### Training

    # 1. Forward pass

    y\_logits = model\_1(X\_train).squeeze()

    y\_pred = torch.round(torch.sigmoid(y\_logits)) # logits -> prediction probabilities -> prediction labels

    # 2. Calculate loss/accuracy

    loss = loss\_fn(y\_logits, y\_train)

    acc = accuracy\_fn(y\_true=y\_train,

                      y\_pred=y\_pred)

    # 3. Optimizer zero grad

    optimizer.zero\_grad()

    # 4. Loss backwards

    loss.backward()

    # 5. Optimizer step

    optimizer.step()

    ### Testing

    model\_1.eval()

    with torch.inference\_mode():

        # 1. Forward pass

        test\_logits = model\_1(X\_test).squeeze()

        test\_pred = torch.round(torch.sigmoid(test\_logits))

        # 2. Caculate loss/accuracy

        test\_loss = loss\_fn(test\_logits,

                            y\_test)

        test\_acc = accuracy\_fn(y\_true=y\_test,

                               y\_pred=test\_pred)

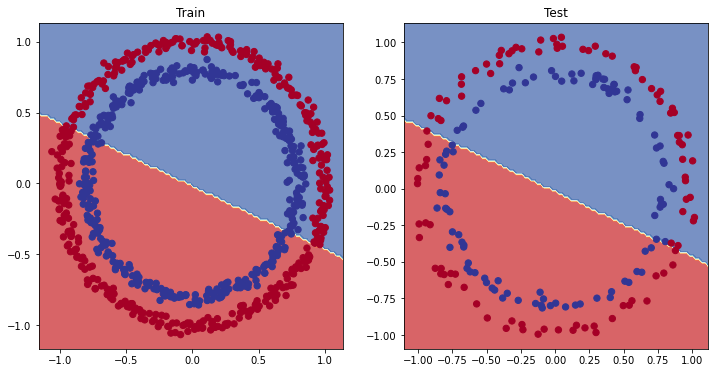
    # Print out what's happening every 10 epochs

    if epoch % 100 == 0:

        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss: {test\_loss:.5f}, Test acc: {test\_acc:.2f}%")

The model still **fails** to predict with higher accuracy.

Let’s visualize the results



5.2