Classification

- https://colab.research.google.com/drive/1XIeDLLtuee9semVVzgo3qrhdarEW9C1D
- https://colab.research.google.com/drive/1qM5pX2p7ce-SPGJDPXDn8RyjSui14a0A#scrollTo=BlXaWC5TkEUE

Шаги цикла обучения PyTorch:

1. **Проход вперед** `forward()` `model(x_train)`

   — модель проходит все обучающие данные один раз, выполняя

   функция расчеты (

).

2. **Рассчитать потерю** `loss = loss_fn(y_pred, y_train`)

   — выходные данные модели (прогнозы) сравниваются с истинными данными и оцениваются. чтобы увидеть, как они ошибаются (

).

3. **Нулевые градиенты** `optimizer.zero_grad()`

   — градиенты оптимизаторов установлены на ноль (по умолчанию они накапливаются), поэтому они возможно пересчитывается для конкретного шага обучения (

).

4. **Выполнить обратное распространение ошибки** `requires_grad=True` **обратное распространение** `loss.backward()`

   — вычисляет градиент потерь с учетом каждой модели. параметр для обновляться (каждый параметр с

). Это известно как

   , следовательно, «обратное» распространение ошибки; (

).

5. **Шаг оптимизатора (градиентный спуск)** `requires_grad=True` `optimizer.step()`

— обновите параметры с помощью

в отношении потери градиенты с целью их улучшения (

).

БИНАРНАЯ КЛАССИФИКАЦИЯ

Создадим датасет

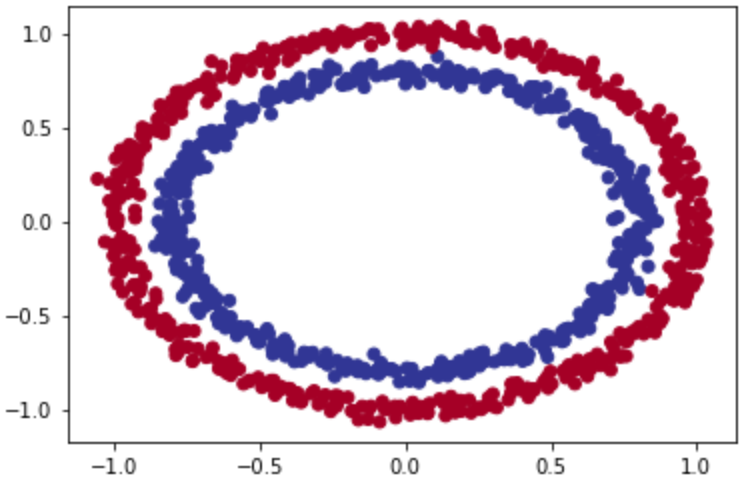
```
from sklearn.datasets import make_circles
# Make 1000 samples
n_samples = 1000

# Create circles
X, y = make_circles(n_samples,
                    noise=0.03, # немного шума в точках
                    random_state=42) # зададим рандомное знач
```

Создадим датафрейм в пандасе и выведем:

```
import pandas as pd
circles = pd.DataFrame({"X1": X[:, 0],
"X2": X[:, 1],
"label": y})
circles.head(10)
```

	X1	X2	label
0	0.754246	0.231481	1
1	-0.756159	0.153259	1
2	-0.815392	0.173282	1
3	-0.393731	0.692883	1
4	0.442208	-0.896723	0
5	-0.479646	0.676435	1
6	-0.013648	0.803349	1

7	0.771513	0.147760	1
8	-0.169322	-0.793456	1
9	-0.121486	1.021509	0




```
X_sample= X[0]
y_sample= y[0]
```

Преобразуем данные в торч тензоры

```
import torch
X= torch.from_numpy(X).type(torch.float)
y= torch.from_numpy(y).type(torch.float)
```

Разобьем данные на тренировочную и тестовую выборки:

```
from sklearn.model_selectionimport train_test_split

X_train, X_test, y_train, y_test= train_test_split(X,
                                                   y,
```

```
                                                        test_size
random_state=42)
```

Используем ГПУ при возможности, в другом случае ЦП

```
import torch
from torchimport nn

device= "cuda" if torch.cuda.is_available()else "cpu"
```

Создадим модель, которая принимает на вход 2 тензора, выдает 5, потом снова принимает 5 и выдает 1.

```
class CircleModelV0(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=
        self.layer_2= nn.Linear(in_features=5, out_features=1
# 3. Define a forward method containing the forward pass comp
        def forward(self, x):
            return self.layer_2(self.layer_1(x))
```

`self.layer_1` принимает 2 входных объекта `in_features=2` и создает 5 выходных объектов `out_features=5`.

Это называется наличием 5 **скрытых единиц** или **нейронов**.

Этот слой превращает входные данные из двух объектов в пять.

Зачем это делать?

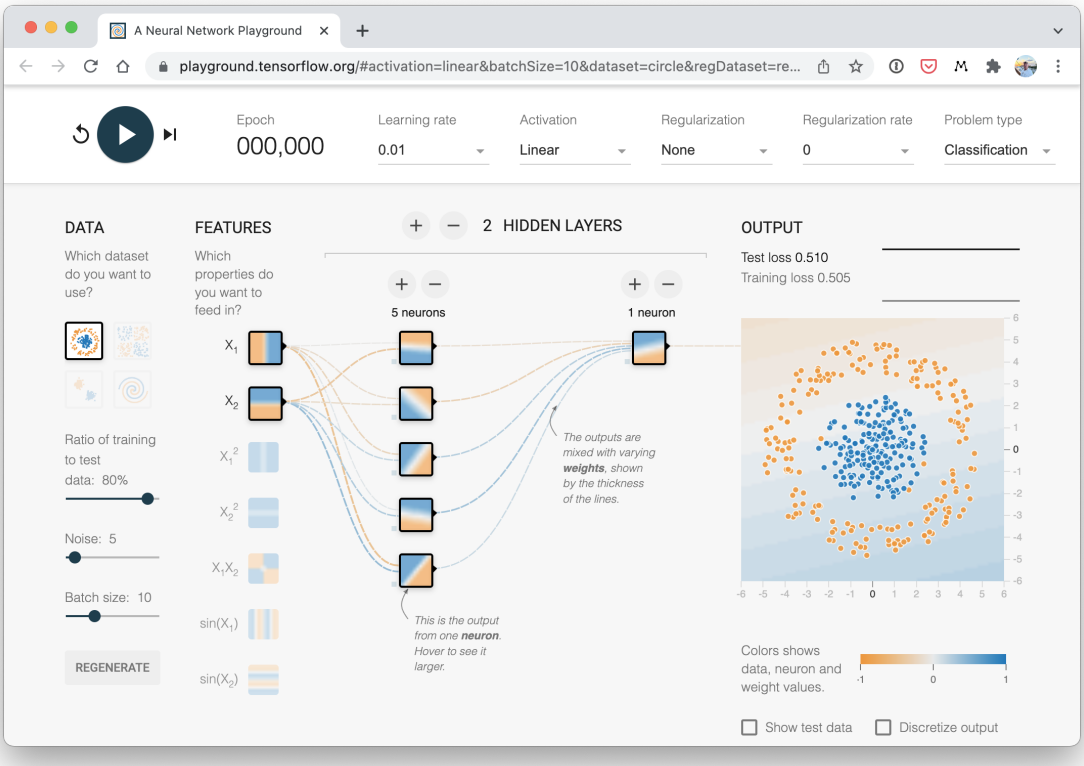
Это позволяет модели изучать закономерности на основе 5 чисел, а не только 2 чисел, *потенциально* что приводит к лучшим результатам.

Количество скрытых единиц, которые вы можете использовать в слоях нейронной сети, представляет собой **гиперпараметр** (значение, которое вы

можете установить самостоятельно)

Единственное правило со скрытыми единицами измерения заключается в том, что следующий слой, в нашем случае, `self.layer_2` должен использовать тот же `in_features`, что и предыдущий слой `out_features`.

Вот почему `self.layer_2` имеет `in_features=5`: он берет `out_features=5` из `self.layer_1` и выполняет над ними линейные вычисления, превращая их в `out_features=1` (такую же форму, как `y`).



https://playground.tensorflow.org/

Также можно использовать nn.Sequential:

```
model_0= nn.Sequential(
    nn.Linear(in_features=2, out_features=5),
```

```
    nn.Linear(in_features=5, out_features=1)
).to(device)
```

Он отлично подходит для прямых вычислений, однако, как сказано в пространстве имен, он *всегда* выполняется в последовательном порядке.

Функция потерь/Оптимизатор	Тип проблемы	Код PyTorch
Оптимизатор стохастического градиентного спуска (SGD)	Классификация, регрессия и многие другие.	`torch.optim.SGD()`
Адам Оптимизатор	Классификация, регрессия и многие другие.	`torch.optim.Adam()`
Двоичная перекрестная потеря энтропии	Бинарная классификация	`torch.nn.BCELossWithLogits` или `torch.nn.BCELoss`
Перекрестная потеря энтропии	Многоклассовая классификация	`torch.nn.CrossEntropyLoss`
Средняя абсолютная ошибка (MAE) или потеря L1	Регрессия	`torch.nn.L1Loss`
Среднеквадратическая ошибка (MSE) или потеря L2	Регрессия	`torch.nn.MSELoss`

PyTorch имеет две реализации двоичной кросс-энтропии:

1. `torch.nn.BCELoss()`

   — Создает функцию потерь, которая измеряет двоичную перекрестную энтропию между целью (меткой) и входом (объектами).

2. `torch.nn.BCEWithLogitsLoss()` `nn.Sigmoid`

   — это то же самое, что и выше, за исключением того, что в него встроен сигмовидный слой.

   `torch.nn.BCEWithLogitsLoss()` указано, что она более стабильна численно, чем использование `torch.nn.BCELoss()`

Создадим функцию потерь и оптимизатор:

```
loss_fn= nn.BCEWithLogitsLoss()
optimizer= torch.optim.SGD(params=model_0.parameters(),lr=0.
```

Метрики качества.

Accuracy(Точность)

Точность можно измерить путем деления общего количества правильных прогнозов на общее количество прогнозов.

Например, модель, которая делает 99 правильных прогнозов из 100, будет иметь точность 99%.

```
def accuracy_fn(y_true, y_pred):
    correct= torch.eq(y_true, y_pred).sum().item()
        acc= (correct/ len(y_pred))* 100
return acc
```

torch.eq() - выдает массив с булевыми значениями([True, True, False…])

```
y_logits = model_0(X_test.to(device))[:5]
y_logits
```

```
Вывод:
tensor([[-0.4279],
        [-0.3417],
        [-0.5975],
        [-0.3801],
        [-0.5078]], device='cuda:0', grad_fn=<SliceBackward0
```

Поскольку наша модель не была обучена, эти выходные данные в основном случайны.

Это выходные данные нашего метода `forward()`.

Реализует два уровня `nn.Linear()`, который внутренне вызывает следующее уравнение: y = x*W.T + b

необработанные результаты нашей модели часто называются **логитами**.

Однако эти цифры трудно интерпретировать. Используем сигмоиду.

```
y_pred_probs= torch.sigmoid(y_logits)
y_pred_probs
```

```
tensor([[0.3946],
        [0.4154],
        [0.3549],
        [0.4061],
        [0.3757]], device='cuda:0', grad_fn=<SigmoidBackward(
```

Теперь они представлены в форме **вероятностей прогнозирования.**

Чем ближе к 0, тем больше модель считает, что выборка принадлежит классу 0; чем ближе к 1, тем больше модель считает, что выборка принадлежит классу 1.

Более конкретно:

- Если `y_pred_probs` >= 0,5, `y=1` (класс 1)
- Если `y_pred_probs` < 0,5, `y=0` (класс 0)

Это можно сделать округлением

```
y_preds= torch.round(y_pred_probs)
y_pred_labels= torch.round(torch.sigmoid(model_0(X_test.to(de
print(torch.eq(y_preds.squeeze(), y_pred_labels.squeeze()))
y_preds.squeeze()
```

`tensor([True, True, True, True, True], device='cuda:0')`

Out[18]:

`tensor([0., 0., 0., 0., 0.], device='cuda:0', grad_fn=<SqueezeBackward0>)`

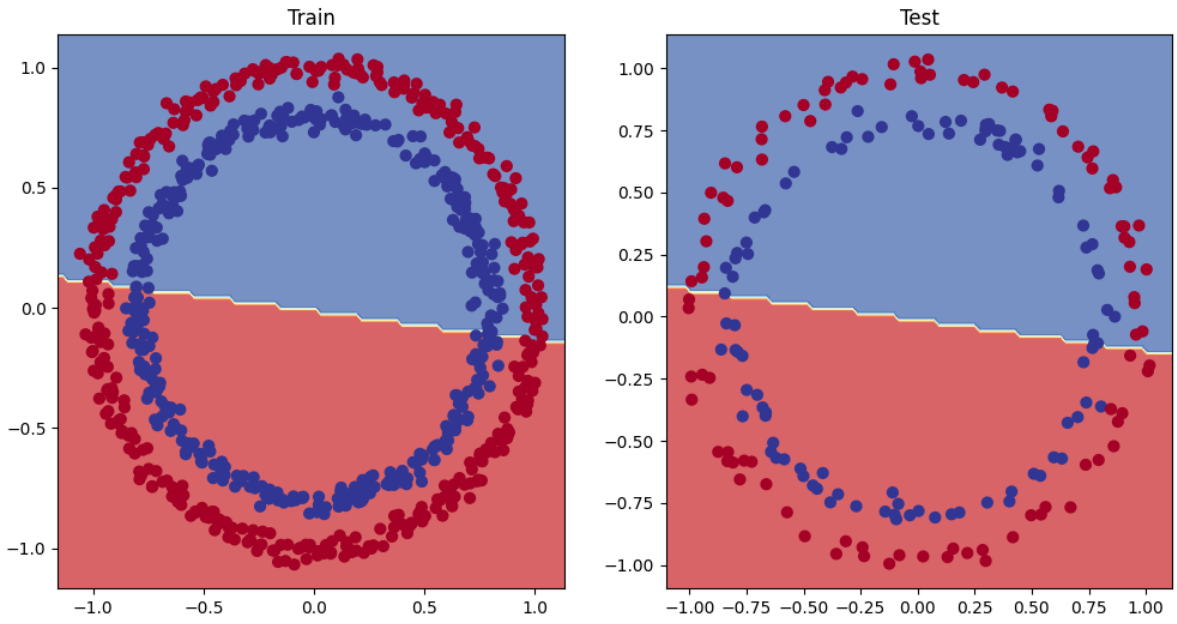
```
#torch.manual_seed(42)
epochs = 100

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):
  model.train() #включение тренировочного режима
  y_logits = model(X_train).squeeze() #получаем данные
  y_pred = torch.round(torch.sigmoid(y_logits)) #вводим сигмо

  loss = loss_fn(y_logits, y_train) #высчитываем потери
  acc = accuracy_fn(y_train, y_pred) #высчитываем точность

  optimizer.zero_grad() #обнуляем оптимизатор(по умолчанию он
  loss.backward() #накопление градиента
  optimizer.step() #обновление оптимизатора

  model.eval() #включение режима эволюции
  with torch.inference_mode():
    test_logits = model(X_test).squeeze()
    test_pred = torch.round(torch.sigmoid(test_logits))

    test_loss = loss_fn(test_logits, y_test)
    test_acc = accuracy_fn(y_test, test_pred)
  if epoch % 10 == 0:
    print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc
```

Epoch: 0 | Loss: 0.70569, Accuracy: 50.00% | Test loss: 0.70523, Test acc: 50.00%
Epoch: 10 | Loss: 0.69760, Accuracy: 50.00% | Test loss: 0.69817, Test acc: 50.00%
Epoch: 20 | Loss: 0.69466, Accuracy: 50.00% | Test loss: 0.69567, Test acc: 50.00%
Epoch: 30 | Loss: 0.69361, Accuracy: 52.50% | Test loss: 0.69482, Test acc: 52.50%
Epoch: 40 | Loss: 0.69323, Accuracy: 54.12% | Test loss: 0.69455, Test acc: 52.00%
Epoch: 50 | Loss: 0.69309, Accuracy: 52.38% | Test loss: 0.69447, Test acc: 50.50%

Epoch: 60 | Loss: 0.69304, Accuracy: 51.75% | Test loss: 0.69446, Test acc: 48.50%
Epoch: 70 | Loss: 0.69301, Accuracy: 51.12% | Test loss: 0.69447, Test acc: 47.50%
Epoch: 80 | Loss: 0.69300, Accuracy: 50.75% | Test loss: 0.69448, Test acc: 45.50%
Epoch: 90 | Loss: 0.69300, Accuracy: 50.75% | Test loss: 0.69449, Test acc: 46.00%

Видим, что наша модель не обучается. Почему?

Визуализируем.

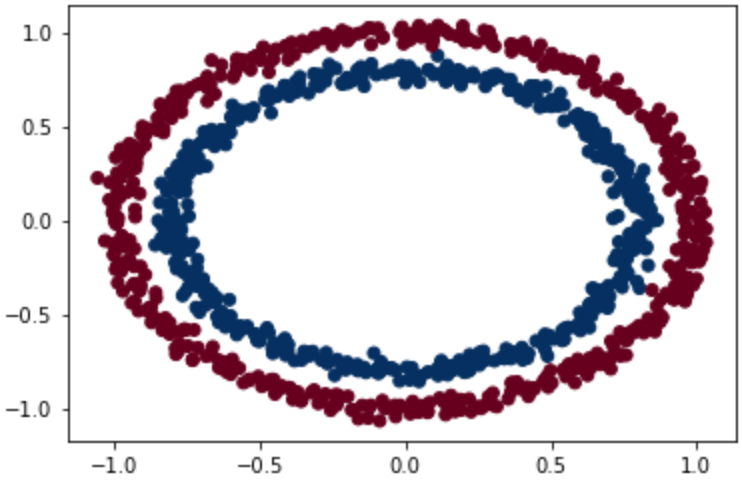


Проблема в линейности.

Начнем сначала.

```
import matplotlib.pyplotas plt
from sklearn.datasetsimport make_circles

n_samples= 1000

X, y= make_circles(n_samples=1000,
    noise=0.03,
    random_state=42,
)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu)
```




```
import torch
from sklearn.model_selectionimport train_test_split

# Turn data into tensorsX= torch.from_numpy(X).type(torch.flo
y= torch.from_numpy(y).type(torch.float)

X_train, X_test, y_train, y_test= train_test_split(X,
                                                   y,
                                                   test_size
                                                   random_st

)

X_train[:5], y_train[:5]
```

Построение нелинейной модели.

В PyTorch есть набор готовых нелинейных функций активации , которые делают похожие, но разные вещи.

Одним из наиболее распространенных и эффективных является ReLU (выпрямленная линейная единица `torch.nn.ReLU()`).

```
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()
  def forward(x, self):
    return self.layer_3(self.relu(self.layer_2(self.relu(self
model_3 = CircleModelV2().to(device)
model_3
```

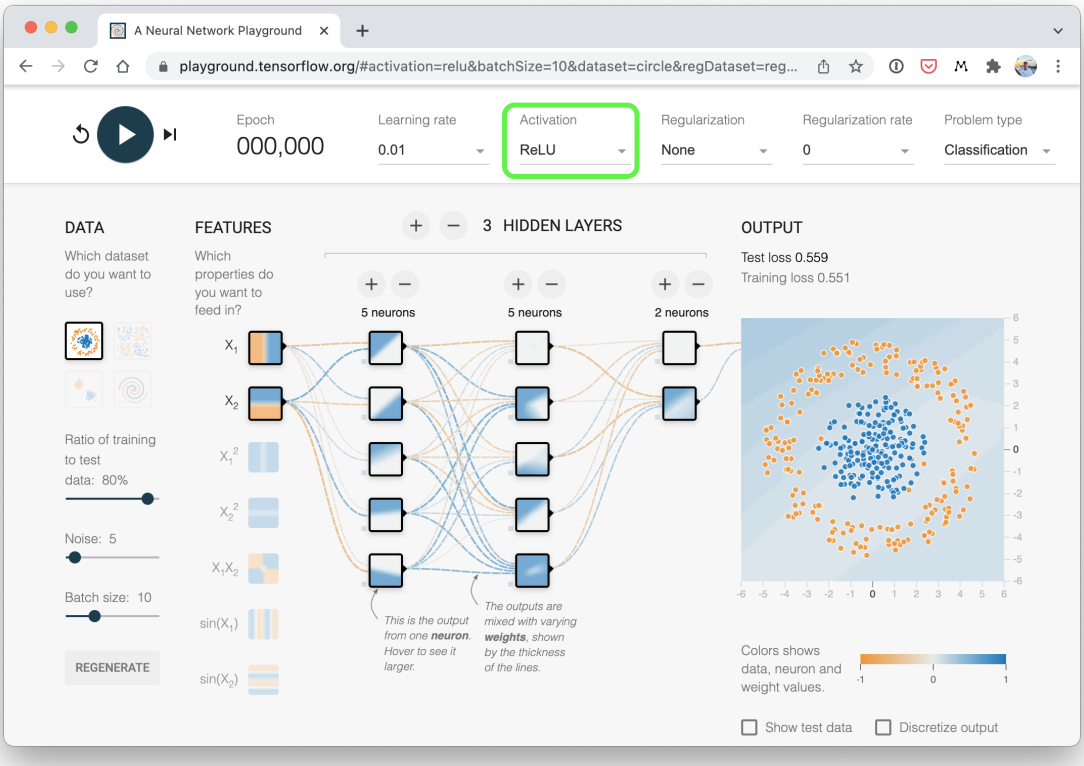
ReLU применяется между каждыми скрытыми слоями.

`nn.ReLU` представляет собой функцию активации Rectified Linear Unit (ReLU) в нейронных сетях. Эта функция активации используется для введения нелинейности в выходные данные нейрона.

ReLU определяется следующим образом:

ReLU(x)=max(0,x)

где *x* - входное значение. Простыми словами, если вход *x* положительный, то ReLU возвращает тот же самый положительный вход. Если вход отрицательный или равен нулю, то ReLU возвращает ноль. Таким образом, ReLU выпрямляет (Rectifies) отрицательные значения, оставляя положительные без изменений.



Создадим заново датасет.

```
import torch
from sklearn.model_selectionimport train_test_split

# Turn data into tensorsX= torch.from_numpy(X).type(torch.flo
y= torch.from_numpy(y).type(torch.float)

# Split into train and test setsX_train, X_test, y_train, y_t
                                                        y,
                                                        test_size
                                                        random_st

)
```

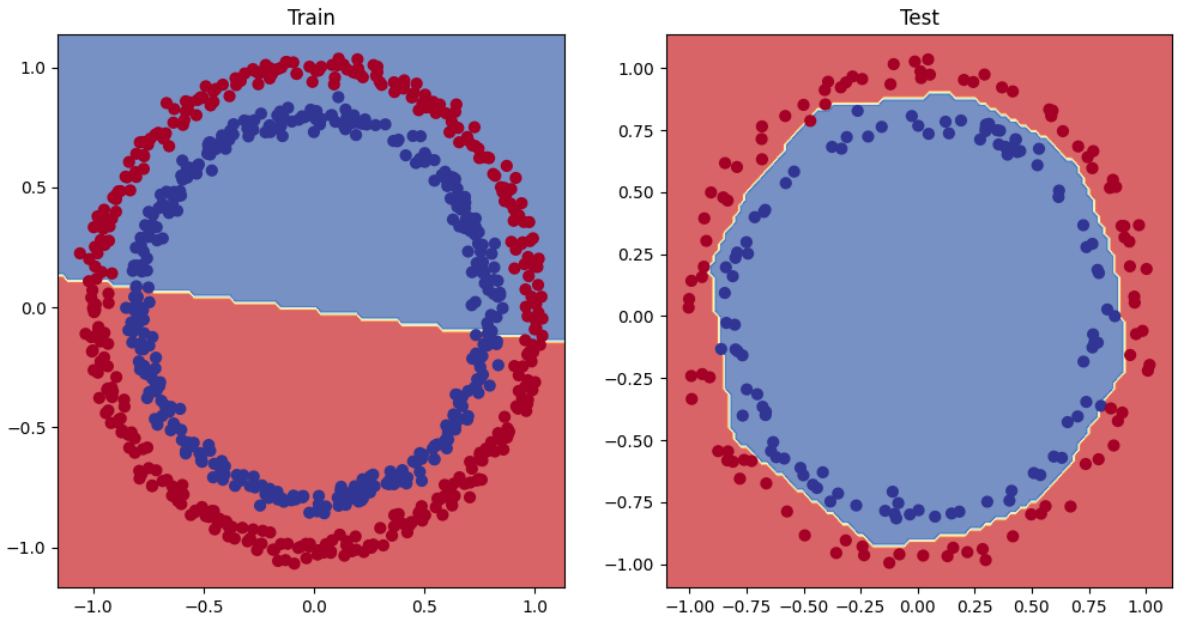
Обучим модель:

```
torch.manual_seed(42)
epochs= 1000

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):
    y_pred= torch.round(torch.sigmoid(y_logits))

        loss= loss_fn(y_logits, y_train)
        acc= accuracy_fn(y_true=y_train,
                         y_pred=y_pred)
        optimizer.zero_grad()

        loss.backward()

        optimizer.step()

        model_3.eval()
        with torch.inference_mode():
            test_logits= model_3(X_test).squeeze()
      test_pred= torch.round(torch.sigmoid(test_logits))# log
      test_acc= accuracy_fn(y_true=y_test,
                            y_pred=test_pred)

        if epoch% 100== 0:
        print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy:
```

Epoch: 0 | Loss: 0.69295, Accuracy: 50.00% | Test Loss: 0.69306, Test Accuracy: 50.00%
Epoch: 100 | Loss: 0.68796, Accuracy: 53.00% | Test Loss: 0.68720, Test Accuracy: 56.00%
Epoch: 200 | Loss: 0.67525, Accuracy: 54.37% | Test Loss: 0.67280, Test Accuracy: 56.50%
Epoch: 300 | Loss: 0.62461, Accuracy: 73.75% | Test Loss: 0.62162, Test Accuracy:

78.50%
Epoch: 400 | Loss: 0.37448, Accuracy: 97.38% | Test Loss: 0.40781, Test Accuracy: 92.50%
Epoch: 500 | Loss: 0.36910, Accuracy: 76.75% | Test Loss: 0.45300, Test Accuracy: 73.50%
Epoch: 600 | Loss: 0.08191, Accuracy: 100.00% | Test Loss: 0.11981, Test Accuracy: 97.50%
Epoch: 700 | Loss: 0.04117, Accuracy: 100.00% | Test Loss: 0.06974, Test Accuracy: 99.00%
Epoch: 800 | Loss: 0.02686, Accuracy: 100.00% | Test Loss: 0.04934, Test Accuracy: 99.00%
Epoch: 900 | Loss: 0.01987, Accuracy: 100.00% | Test Loss: 0.04013, Test Accuracy: 99.00%

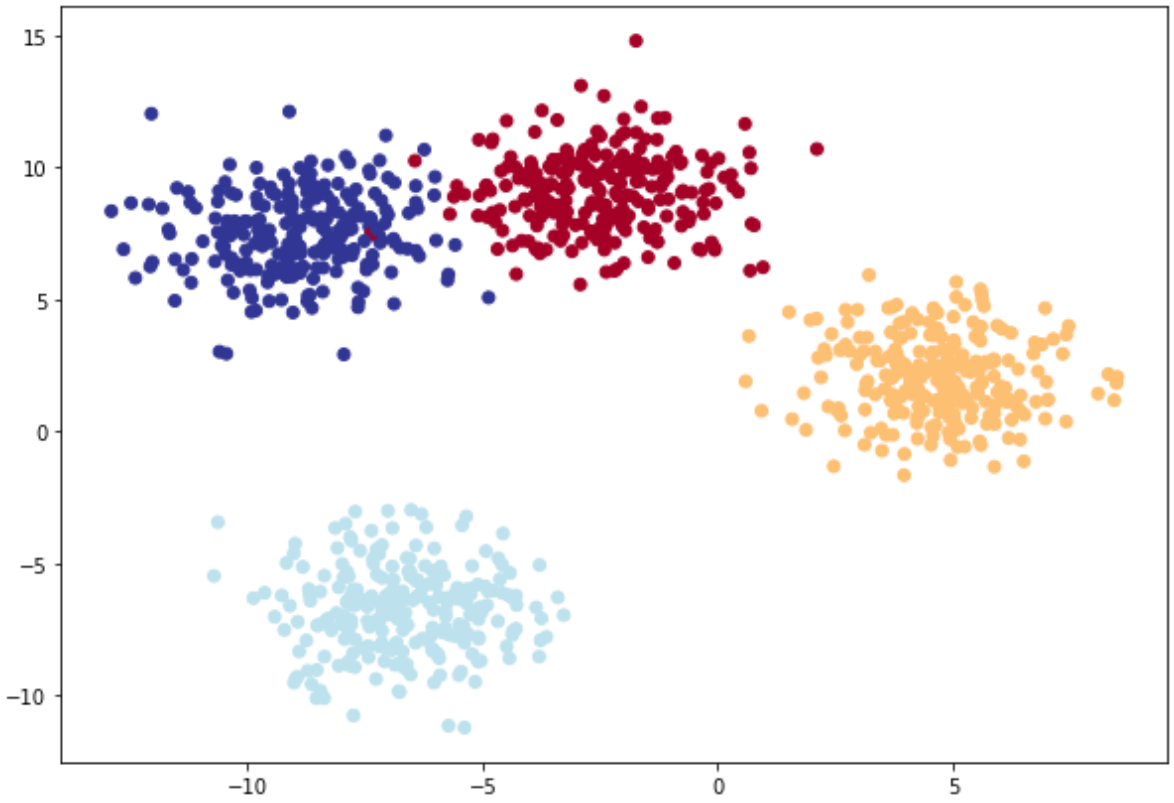
```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model, X_train, y_train) # model_1 = r
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_3, X_test, y_test)
```



Мультиклассовая классификация:

Для начала опять же создадим датасет, но теперь с большим количеством классов.

```
import torch
import matplotlib.pyplotas plt
from sklearn.datasetsimport make_blobs
from sklearn.model_selectionimport train_test_split

# Set the hyperparameters for data creationNUM_CLASSES= 4
NUM_FEATURES= 2
RANDOM_SEED= 42

# 1. Create multi-class dataX_blob, y_blob= make_blobs(n_samp
    n_features=NUM_FEATURES,# X featurescenters=NUM_CLASSES,#
)

# 2. Turn data into tensorsX_blob= torch.from_numpy(X_blob).t
y_blob= torch.from_numpy(y_blob).type(torch.LongTensor)
print(X_blob[:5], y_blob[:5])

# 3. Split into train and test setsX_blob_train, X_blob_test,
    y_blob,
    test_size=0.2,
    random_state=RANDOM_SEED
)

# 4. Plot dataplt.figure(figsize=(10, 7))
plt.scatter(X_blob[:, 0], X_blob[:, 1], c=y_blob, cmap=plt.cm
```



Создадим код, независимый от устройства:

```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

Создадим класс. Здесь есть 2 способа:

- Первый показан выше, но продублируем и заметим, что количество выходов равно количеству классов:

```
class Model_Multi_class0(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer_1 = nn.Linear(in_features=2, out_features=1
    self.layer_2 = nn.Linear(in_features=10, out_features=
    self.layer_3 = nn.Linear(in_features=10, out_features=
    self.relu = nn.ReLU()
```

```
  def forward(self, x):
    return self.layer_3(self.relu(self.layer_2(self.relu(s
```

- Второй содержит тот же алгоритм, только используется функция `nn.Sequential`:

```
class BlobModel(nn.Module):
def __init__(self, input_features, output_features, hidden
        super().__init__()
        self.linear_layer_stack= nn.Sequential(
            nn.Linear(in_features=input_features, out_feat
                        nn.Linear(in_features=hidden_units
                        nn.Linear(in_features=hidden_units
                )
def forward(self, x):
        return self.linear_layer_stack(x)
```

Функция потерь и оптимизатор:

Так как у нас многоклассовая классификация, то будем использовать этот метод `nn.CrossEntropyLoss()`.

И мы продолжим использовать SGD со скоростью обучения 0,1 для оптимизации наших `model_4` параметров. Кроме этого существует полезный метод Адама.

```
loss_fn= nn.CrossEntropyLoss()
optimizer= torch.optim.SGD(model_4.parameters(),
                           lr=0.1)
```

Что выводит?

```
model_4(X_blob_train.to(device))[:5]
```

```
ВЫВОД:
tensor([[-1.2711, -0.6494, -1.4740, -0.7044],
        [ 0.2210, -1.5439,  0.0420,  1.1531],
        [ 2.8698,  0.9143,  3.3169,  1.4027],
        [ 1.9576,  0.3125,  2.2244,  1.1324],
        [ 0.5458, -1.2381,  0.4441,  1.1804]], device='cuda:0
       grad_fn=<SliceBackward0>)
```

Замечательно, наша модель прогнозирует одно значение для каждого имеющегося у нас класса.

Чтобы получать вероятность отнесения к каждому классу мы используем функцию активации **softmax.**

```
y_pred_probs= torch.softmax(y_logits, dim=1)
print(y_logits[:5])
print(y_pred_probs[:5])
```

```
ВЫВОД:
tensor([[-1.2549, -0.8112, -1.4795, -0.5696],
        [ 1.7168, -1.2270,  1.7367,  2.1010],
        [ 2.2400,  0.7714,  2.6020,  1.0107],
        [-0.7993, -0.3723, -0.9138, -0.5388],
        [-0.4332, -1.6117, -0.6891,  0.6852]], device='cuda:0
       grad_fn=<SliceBackward0>)
tensor([[0.1872, 0.2918, 0.1495, 0.3715],
        [0.2824, 0.0149, 0.2881, 0.4147],
        [0.3380, 0.0778, 0.4854, 0.0989],
        [0.2118, 0.3246, 0.1889, 0.2748],
```

```
         [0.1945, 0.0598, 0.1506, 0.5951]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

Отлично!

Поскольку для каждого класса в есть одно значение `y_pred_probs`, индекс самого *высокого значения — это класс, к которому, по мнению модели, больше всего* принадлежит конкретная выборка данных .

Мы можем проверить, какой индекс имеет наибольшее значение, используя `torch.argmax()`.

```
print(y_pred_probs[0])
print(torch.argmax(y_pred_probs[0]))
```

```
ВЫВОД:
tensor([0.1872, 0.2918, 0.1495, 0.3715], device='cuda:0',
       grad_fn=<SelectBackward0>)
tensor(3, device='cuda:0')
```

Осталось натренировать модель

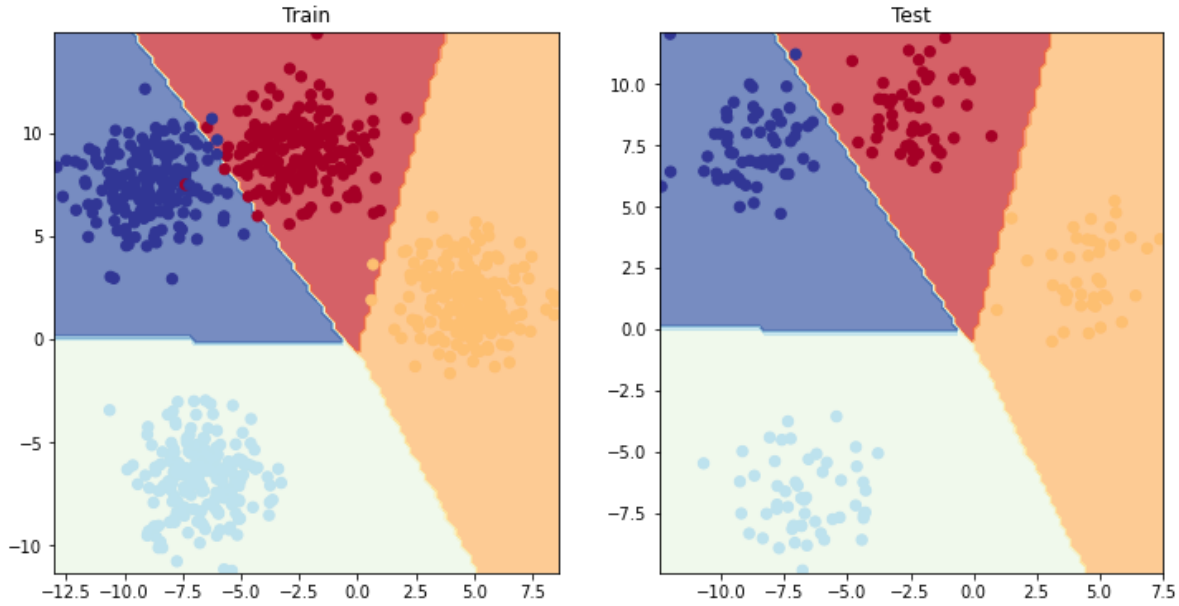
```
torch.manual_seed(42)

epochs= 100

X_blob_train, y_blob_train= X_blob_train.to(device), y_blob_t
X_blob_test, y_blob_test= X_blob_test.to(device), y_blob_test

for epochin range(epochs):
        model_4.train()

        y_logits= model_4(X_blob_train)# model outputs raw lo
```

```
        y_pred= torch.softmax(y_logits, dim=1).argmax(dim=1)#
        loss= loss_fn(y_logits, y_blob_train)
    acc= accuracy_fn(y_true=y_blob_train,
                     y_pred=y_pred)

# 3. Optimizer zero grad
        optimizer.zero_grad()

# 4. Loss backwards
        loss.backward()

# 5. Optimizer step
        optimizer.step()

### Testingmodel_4.eval()
        with torch.inference_mode():
# 1. Forward passtest_logits= model_4(X_blob_test)
          test_pred= torch.softmax(test_logits, dim=1).argmax
# 2. Calculate test loss and accuracy
               test_loss= loss_fn(test_logits, y_blob_test)
          test_acc= accuracy_fn(y_true=y_blob_test,
                                y_pred=test_pred)

# Print out what's happening
               if epoch% 10== 0:
               print(f"Epoch: {epoch} | Loss: {loss:.5f}, Ac
```

Визуализируем!

```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_4, X_blob_train, y_blob_train)
plt.subplot(1, 2, 2)
```

```
plt.title("Test")
plot_decision_boundary(model_4, X_blob_test, y_blob_test)
```



Больше метрик качества:

Название метрики/ метод оценки	Определение	Код
Точность	Сколько из 100 прогнозов окажется верным ваша модель? Например, точность 95% означает, что он дает 95/100 правильных прогнозов.	`torchmetrics.Accuracy()` или `sklearn.metrics.accuracy_score()`
Точность	Доля истинно положительных результатов от общего количества образцов. Более высокая точность приводит к меньшему количеству ложных срабатываний (модель прогнозирует 1, хотя должно было быть 0).	`torchmetrics.Precision()` или `sklearn.metrics.precision_score()`
Отзывать	Доля истинных положительных результатов от общего числа истинных положительных и ложных отрицательных результатов (модель прогнозирует 0, хотя должно было быть 1). Более высокий уровень отзыва приводит к меньшему количеству ложноотрицательных результатов.	`torchmetrics.Recall()` или `sklearn.metrics.recall_score()`
F1-оценка	Сочетает точность и полноту в одном показателе. 1 – лучшее, 0 – худшее.	`torchmetrics.F1Score()` или `sklearn.metrics.f1_score()`
Матрица путаницы	Сравнивает прогнозируемые значения с истинными значениями в табличной форме. Если они верны на 100 %, все значения в матрице будут располагаться сверху слева и справа внизу (строка диагностики).	`torchmetrics.ConfusionMatrix` или `sklearn.metrics.plot_confusion_matrix()`
Классификационный отчет	Сбор некоторых основных показателей классификации, таких как точность, полнота и показатель f1.	`sklearn.metrics.classification_report()`