We are starting at 14:00!

Grab a seat and get ready

Saturdays.AI
Kigali

Saturdays.AI
Kigali

#1 Basics and Pytorch

AI Saturdays Kigali

Agenda

14:00 - 16:00: The basics of Pytorch

16:00 - 16:30: Break

16:30 - 17:30: Neural Network

17:30 - 18:00: Challenges & Next steps

Pytorch Basics

Saturdays.A
Kigali

Pytorch

- Numpy
  - on a GPU
  - with all kinds of ANN related things
  - with a focus on tensors
  - and automatic differentiation

Alternatives to pytorch

- Tensorflow - strength in commercial applications
- JAX - strength in flexibility
- Matlab

Saturdays.AI
Kigali

DL in Numpy

```python
import numpy as np

# Define sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define the NaiveNet class
class NaiveNet:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases with random values
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, inputs):
        # Forward propagation
        self.hidden_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.final_output = sigmoid(self.final_input)
        return self.final_output

    def backward(self, inputs, targets, learning_rate):
        # Backpropagation
        error = targets - self.final_output
        delta_output = error * sigmoid_derivative(self.final_output)

        error_hidden = delta_output.dot(self.weights_hidden_output.T)
        delta_hidden = error_hidden * sigmoid_derivative(self.hidden_output)

        self.weights_hidden_output += self.hidden_output.T.dot(delta_output) * learning_rate
        self.bias_output += np.sum(delta_output, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += inputs.T.dot(delta_hidden) * learning_rate
        self.bias_hidden += np.sum(delta_hidden, axis=0, keepdims=True) * learning_rate
```

Saturdays.AI
Kigali

DL in Pytorch

```
class NaiveNet(nn.Module):
  # Define the structure of your network
  def __init__(self):
    super(NaiveNet, self).__init__()

    # The network is defined as a sequence of operations
    self.layers = nn.Sequential(
        nn.Linear(2, 16),
        nn.ReLU(),
        nn.Linear(16, 2),
    )

  # Specify the computations performed on the data
  def forward(self, x):
    # Pass the data through the layers
    return self.layers(x)
```

Saturdays.AI
Kigali

Everything in pytorch are tensors: how to make one

A `torch.Tensor` is a multi-dimensional (or n-dimensional) matrix containing elements of a single data type.

```
# tensor from a list
a = torch.tensor([0, 1, 2])

#tensor from a tuple of tuples
b = ((1.0, 1.1), (1.2, 1.3))
b = torch.tensor(b)

# tensor from a numpy array
c = np.ones([2, 3])
c = torch.tensor(c)
```

More tensors: common constructors

```
x = torch.ones(5, 3)
y = torch.zeros(2)
z = torch.empty(1, 1, 5)
```

Saturdays.AI
Kigali

Making random tensors

```
# Uniform distribution
a = torch.rand(1, 3)

# Normal distribution
b = torch.randn(3, 4)
```

Saturdays.AI
Kigali

Ranges in pytorch - just like in numpy

```
a = torch.arange(0, 10, step=1)
b = np.arange(0, 10, step=1)

c = torch.linspace(0, 5, steps=11)
d = np.linspace(0, 5, num=11)
```

Copying Tensors

As with any object in Python, assigning a tensor to a variable makes the variable a label of the tensor, and does not copy it (create a copy of it). For example:

```
a = torch.ones(2, 2)
b = a

a[0][1] = 561  # we change a...
print(b)       # ...and b is also altered
```

Out:

```
tensor([[  1., 561.],
        [  1.,   1.]])
```

Saturdays.AI
Kigali

Practice

Make a couple of Tensors

(10 minutes max)

[Coding Exercise 2.1]

What can we do with tensors?

Everything we do with numpy otherwise.

```
# this only works if c and d already exist
torch.add(a, b, out=c)

# Pointwise Multiplication of a and b
torch.multiply(a, b, out=d)
```

Saturdays.AI
Kigali

By default everything is pointwise

```
x + y, x - y, x * y, x / y, x**y  # The `**` is the exponentiation operator
```

Saturdays.AI
Kigali

Sums, means etc

Just like in numpy

```
print(f"Sum of every element of x: {x.sum()}")
print(f"Sum of the columns of x: {x.sum(axis=0)}")
print(f"Sum of the rows of x: {x.sum(axis=1)}")
```

Saturdays.AI
Kigali

Practice

Do a few things with Tensors

(10 minutes max)

[Coding Exercise 2.2]

Manipulating tensors: indexing

```
x = torch.arange(0, 10)
print(x)
print(x[-1])
print(x[1:3])
print(x[:-2])
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
tensor(9)
tensor([1, 2])
tensor([0, 1, 2, 3, 4, 5, 6, 7])
```

Similar logic as numpy for n-dimensional tensors

```python
# make a 5D tensor
x = torch.rand(1, 2, 3, 4, 5)

print(f" shape of x[0]:{x[0].shape}")
print(f" shape of x[0][0]:{x[0][0].shape}")
print(f" shape of x[0][0][0]:{x[0][0][0].shape}")
```

```
 shape of x[0]:torch.Size([2, 3, 4, 5])
 shape of x[0][0]:torch.Size([3, 4, 5])
 shape of x[0][0][0]:torch.Size([4, 5])
```

Saturdays.AI
Kigali

Flattening/ Reshaping

```
z = torch.arange(12).reshape(6, 2)
print(f"Original z: \n {z}")

# 2D -> 1D
z = z.flatten()
print(f"Flattened z: \n {z}")
```

```
Original z:
 tensor([[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7],
        [ 8,  9],
        [10, 11]])
Flattened z:
 tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Saturdays.AI
Kigali

Reshaping

```python
# and back to 2D
z = z.reshape(3, 4)
print(f"Reshaped (3x4) z: \n {z}")
```

```
Reshaped (3x4) z:
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

Irrelevant dimensions

```
x = torch.randn(1, 10)
# printing the zeroth element of the tensor will not give us the first number!

print(x.shape)
print(f"x[0]: {x[0]}")
```

```
torch.Size([1, 10])
x[0]: tensor([ 0.7274, -0.1908,  1.1524,  0.0241,  0.7664,  1.0263, -1.2895,  0.2036,
         0.5379, -0.7962])
```

Saturdays.AI
Kigali

Squeezing

```python
# Let's get rid of that singleton dimension and see what happens now
x = x.squeeze(0)
print(x.shape)
print(f"x[0]: {x[0]}")
```

```
torch.Size([10])
x[0]: 0.7273916602134705
```

Saturdays.AI
Kigali

Dimension permutation

E.g. going from RGB in dimension 1 to in dimension 3

```
# `x` has dimensions [color,image_height,image_width]
x = torch.rand(3, 48, 64)

# We want to permute our tensor to be [ image_height , image_width , color ]
x = x.permute(1, 2, 0)
# permute(1,2,0) means:
# The 0th dim of my new tensor = the 1st dim of my old tensor
# The 1st dim of my new tensor = the 2nd
# The 2nd dim of my new tensor = the 0th
print(x.shape)
```

```
torch.Size([48, 64, 3])
```

Saturdays.AI
Kigali

Concatenation

```python
# Create two tensors of the same shape
x = torch.arange(12, dtype=torch.float32).reshape((3, 4))
y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])

# Concatenate along rows
cat_rows = torch.cat((x, y), dim=0)

# Concatenate along columns
cat_cols = torch.cat((x, y), dim=1)

# Printing outputs
print('Concatenated by rows: shape{} \n {}'.format(list(cat_rows.shape), cat_rows))
print('\n Concatenated by colums: shape{}  \n {}'.format(list(cat_cols.shape), cat_cols))
```

```
Concatenated by rows: shape[6, 4]
 tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]])

 Concatenated by colums: shape[3, 8]
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]])
```

Saturdays.AI
Kigali

torch and numpy are friends

```
x = torch.randn(5)
print(f"x: {x}  |  x type:  {x.type()}")

y = x.numpy()
print(f"y: {y}  |  y type:  {type(y)}")

z = torch.tensor(y)
print(f"z: {z}  |  z type:  {z.type()}")
```

```
x: tensor([ 0.5728,  0.6117,  1.1747, -2.2578, -0.0359])  |  x type:  torch.FloatTensor
y: [ 0.5728153   0.61165035  1.1746601  -2.2578378  -0.0358642 ]  |  y type:  <class 'numpy.ndarray'>
z: tensor([ 0.5728,  0.6117,  1.1747, -2.2578, -0.0359])  |  z type:  torch.FloatTensor
```

Saturdays.AI
Kigali

Practice

Do the tensor manipulation exercise

Trust me, these "easy" things are where the errors often happen

Coding Exercise 2.3

Saturdays.AI
Kigali

Graphics cards: using GPUs

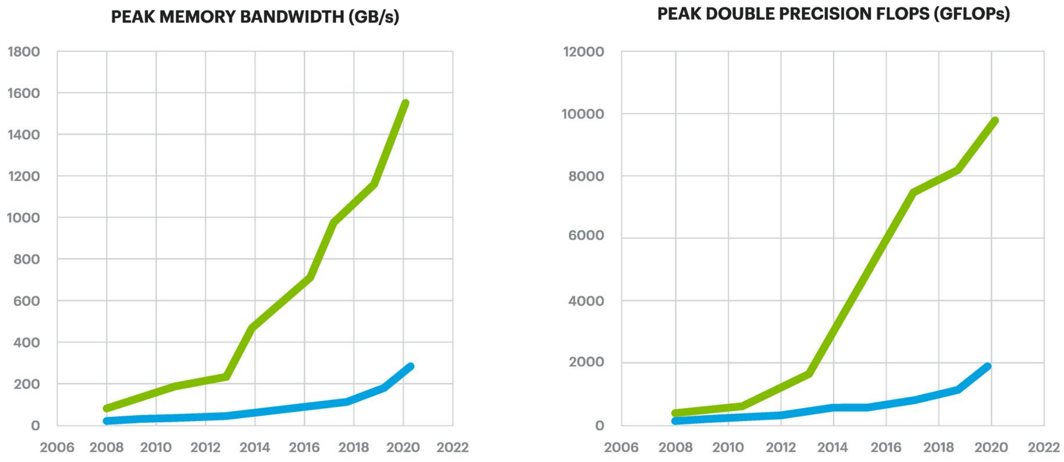


Figure 2. Comparison of evolution of memory bandwidth (left) and double precision flops (right) on GPU and CPU

Saturdays.AI
Kigali

Ask torch where a variable is

```
x = torch.randn(10)
print(x.device)
```

```
cpu
```

Saturdays.AI
Kigali

Ask torch if we have a GPU

```
print(torch.cuda.is_available())
```

```
True
```

Saturdays.AI
Kigali

Specifying devices

```python
# common device agnostic way of writing code that can run on cpu OR gpu
# that we provide for you in each of the tutorials
DEVICE = set_device()

# we can specify a device when we first create our tensor
x = torch.randn(2, 2, device=DEVICE)
print(x.dtype)
print(x.device)

# we can also use the .to() method to change the device a tensor lives on
y = torch.randn(2, 2)
print(f"y before calling to() | device: {y.device} | dtype: {y.type()}")

y = y.to(DEVICE)
print(f"y after calling to() | device: {y.device} | dtype: {y.type()}")
```

```
torch.float32
cuda:0
y before calling to() | device: cpu | dtype: torch.FloatTensor
y after calling to() | device: cuda:0 | dtype: torch.cuda.FloatTensor
```

Saturdays.AI
Kigali

Device matters: no mix and match

We can not just mix and match devices - it would be undefined where the computation happens

```
x = torch.tensor([0, 1, 2], device=DEVICE)
y = torch.tensor([3, 4, 5], device="cpu")

z = x + y
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-27-3ae8fe7d9873> in <cell line: 4>()
      2 y = torch.tensor([3, 4, 5], device="cpu")
      3
----> 4 z = x + y

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
```

SEARCH STACK OVERFLOW

Saturdays.AI
Kigali

Moving CPU<->GPU is easy

```python
x = torch.tensor([0, 1, 2], device=DEVICE)
y = torch.tensor([3, 4, 5], device="cpu")
z = torch.tensor([6, 7, 8], device=DEVICE)

# moving to cpu
x = x.to("cpu")  # alternatively, you can use x = x.cpu()
print(x + y)

# moving to gpu
y = y.to(DEVICE)  # alternatively, you can use y = y.cuda()
print(y + z)
```

```
tensor([3, 5, 7])
tensor([ 9, 11, 13], device='cuda:0')
```

Saturdays.AI
Kigali

Practice

Test the GPU effect

I promise you GPUs are faster

[Coding Exercise 2.4]

Datasets

Data

+

Model

+

Training

=

DL system

Saturdays.AI
Kigali

Doing data - basics

- Data science =
- 50% figure out the question you want to answer
- 35% sweat the data
- 10% ML
- 5% glorious DL

How to get data

A lot of data is easy to load for our DL experiments

```
# Download and load the images from the CIFAR10 dataset
cifar10_data = datasets.CIFAR10(
    root="data",            # path where the images will be stored
    download=True,          # all images should be downloaded
    transform=ToTensor()  # transform the images to tensors
)
```

Saturdays.AI
Kigali

Practice

Display the CIFAR image

Let us look into this

Saturdays.AI
Kigali

Data

- Data is not made in heaven
- Data is made to answer questions
- We need to be agile with data
- When we try to answer questions we do not want to lie to ourselves

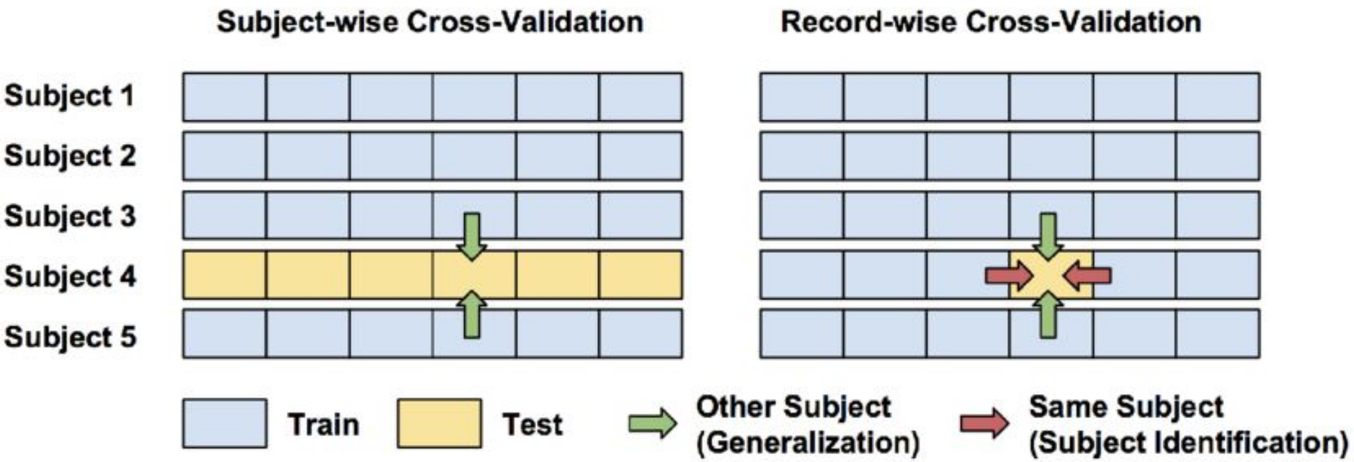
Let us not lie about data

- In DL we generally do prediction
- Caution with Causality
- The real world differs from our dataset (external validity)

Let us not lie about data: Validation

- Always have a validation/ Test set not used for training.
- Train on training set, test on test set
- For hyperparameter optimization you need to further divide the training set
- Match the cross-validation strategy to the use case

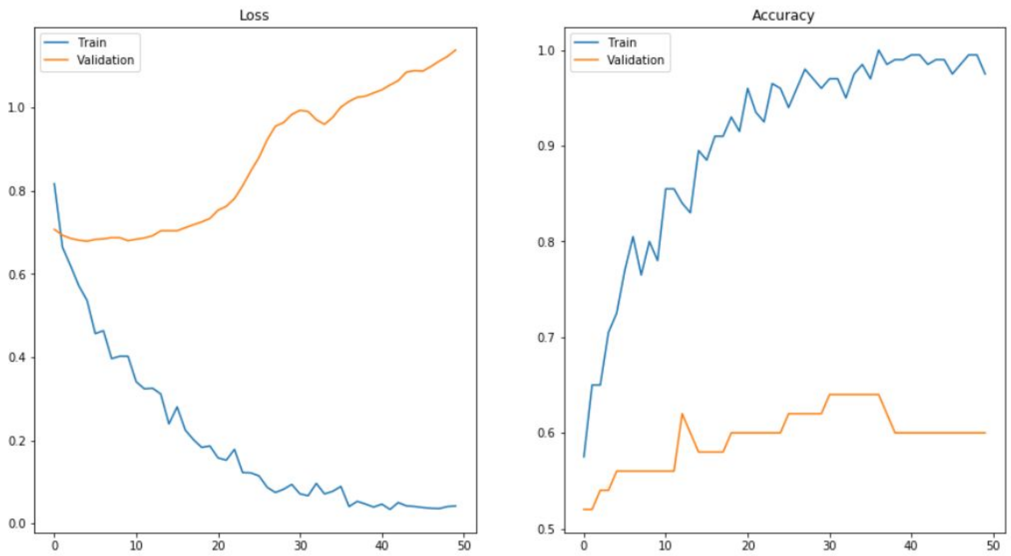
The cross-validation strategy must match the use case



Saturdays.AI
Kigali

Overfitting! Validation set

- Don’t trust yourself
- Overfitting is massive for smaller datasets
- Ideally have a part of the dataset you don’t
- have access to
- Even some signs for *huge* datasets (imagenet)



Saturdays.AI
Kigali

Always have both training and test data

```
# Load the training samples
training_data = datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
    )

# Load the test samples
test_data = datasets.CIFAR10(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
    )
```

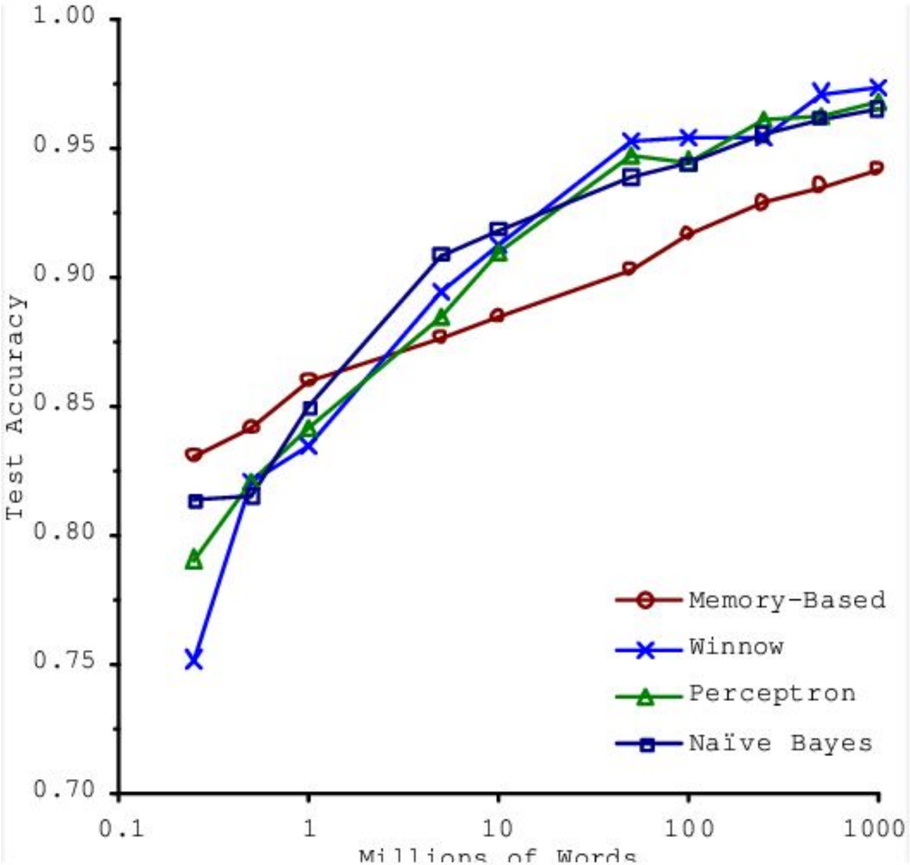
Practice

See how CIFAR train and test data are loaded

Let us load some data and divide into train and test dataset

[Coding Exercise 2.5]

More data is what it is all about



Transformations

More data = better learning

How to get more data?

Get more data

Transform the data

e.g. add color variation, etc.

Transformations are crucial across DL

Saturdays.AI
Kigali

Data Loaders

- In practice we do not load all data.
- But small pieces (minibatches)
- For that we have a function that does the loading

```
# Create dataloaders with
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Saturdays.AI
Kigali

Practice

Load CIFAR images as grayscale

Practice transformation

Break

Saturdays.A
Kigali

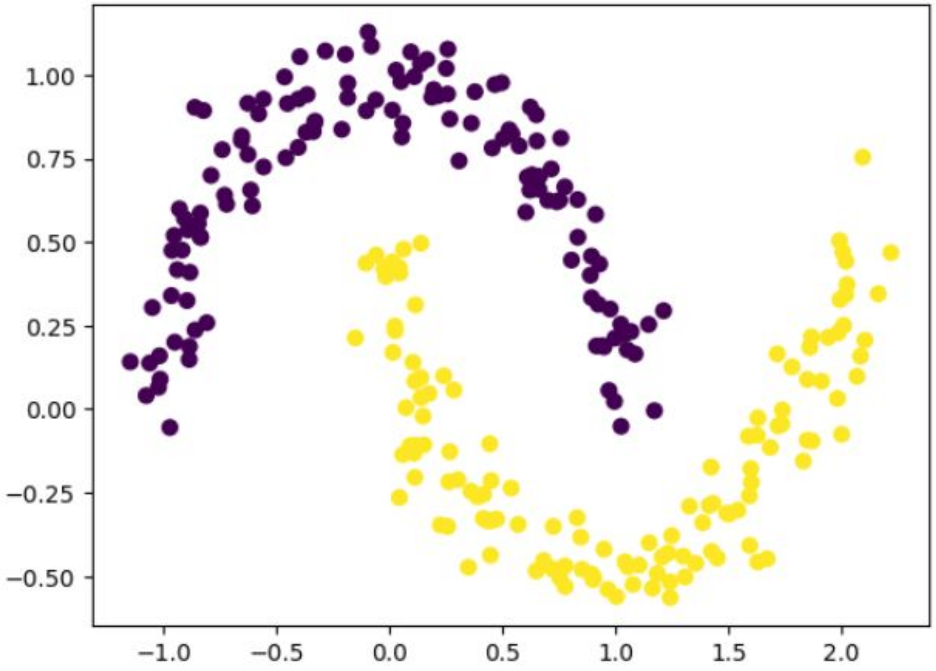
Neural Networks

Now, let us design a neural network

- (step 0) Get Data
- (step 1) All the variables and structures we need. We need to initialize them
- (step 2) And then we need to use these variables to define the compute in our network
- (step 3) And then we need gradients
- (step 4) And then we need to optimize
- (step 5) And then we need to test

Let us get the data from a csv file

Why? Because many real world datasets are in that format



Saturdays.AI
Kigali

Practice

Load from csv and put on GPU into torch

Can you think of other datasets you could load this way?

Saturdays.AI
Kigali

Let us see the anatomy of the network

First, we need to initialize the relevant variables

`__init__`

And then we need to specify how information travels through network

`forward`

Saturdays.AI
Kigali

We will often need to make predictions

While many people just use `forward` we will separate it and use

```
predict
```

and then we need to

```
train
```

Saturdays.AI
Kigali

With __init__ we make network structure

```
# Define the structure of your network
def __init__(self):
  super(NaiveNet, self).__init__()

  # The network is defined as a sequence of operations
  self.layers = nn.Sequential(
      nn.Linear(2, 16),
      nn.ReLU(),
      nn.Linear(16, 2),
  )
```

Saturdays.AI
Kigali

The other components

```python
# Specify the computations performed on the data
def forward(self, x):
  # Pass the data through the layers
  return self.layers(x)

# Choose the most likely label predicted by the network
def predict(self, x):
  # Pass the data through the networks
  output = self.forward(x)

  # Choose the label with the highest score
  return torch.argmax(output, 1)

# Train the neural network (will be implemented later)
def train(self, X, y):
  pass
```

Practice

Run your first neural network

Check if it actually works and provides the outputs we expect

Saturdays.AI
Kigali

Ok hold on. What has just happened

We have a neural network

It is initialized

It produces outputs

But these outputs are not better than chance yet!

Training = lots of small steps into good direction

```
# The Cross Entropy Loss is suitable for classification problems
loss_function = nn.CrossEntropyLoss()

# Create an optimizer (Stochastic Gradient Descent) that will be used to train the network
learning_rate = 1e-2
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Number of epochs
epochs = 15000
```

The anatomy of the training loop

```
for i in range(epochs):
  # Pass the data through the network and compute the loss
  y_logits = model.forward(X)
  loss = loss_function(y_logits, y)

  # Clear the previous gradients and compute the new ones
  optimizer.zero_grad()
  loss.backward()

  # Adapt the weights of the network
  optimizer.step()
```

Saturdays.AI
Kigali

Practice

We give you code. What happens?

Train your network

Challenges & Next steps!

Saturdays.AI
Kigali

Kahoot!

Any questions?

Saturdays.AI
Kigali

Saturdays.AI
Kigali

THANKS

kigali@saturdays.ai
coming soon
coming soon