NLP01022: PyTorch Workshop

Having question about the workshop or this notebook? Contact Erfan Moosavi Monazzah (Tel: @ErfanMoosavi2000).
This notebook is adapted from CS224n PyTorch Workshop

Plan

We'll have an introduction to PyTorch and its important modules. Also we get our hands on a simple NLP task.
This notebook is available on telegram group after this session.

Introduction

PyTorch is a deep learning framework, one of the two main frameworks alongside TensorFlow. PyTorch is a popular choice among researchers and practitioners for its ease of use, flexibility, and dynamic computation graph. It allows for seamless use of GPUs, and offers extensive support for common neural network architectures and modules.

Some of PyTorch's capabilities include:

- Dynamic computation graph
- Easy debugging and visualization with tensorboard
- Distributed training on multiple GPUs and machines
- Support for various neural network architectures and modules, including convolutional and recurrent neural networks, transformers, and more.
  Let's start by importing PyTorch:

```
import torch
import torch.nn as nn
```

We are all set to start our tutorial. Let's dive in!

Tensors

Tensors are

- PyTorch's most basic building block.
- multi-dimensional matrices.

for example: A 256x256 image might be represented by a `3x256x256` tensor (First dimension represents color channels)

🤔 How can we represent a sentence using tensors?

```
list_of_lists = [
  [1, 2, 3],
  [4, 5, 6],
]
print(list_of_lists)
```

```
[[1, 2, 3], [4, 5, 6]]
```

```
# Initializing a tensor
data = torch.tensor([
                     [0, 1],
                     [2, 3],
                     [4, 5]
                    ])
print(data)
```

```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

Each tensor has a **data type**, something like:

- `torch.float32`
- `torch.int`

  You can specify the data type explicitly when you create the tensor:

```
# Notice the dots after the numbers, which specify that they're floats
data = torch.tensor([
                     [0, 1],
                     [2, 3],
                     [4, 5]
                    ], dtype=torch.float32)
print(data)
```

```
tensor([[0., 1.],
        [2., 3.],
        [4., 5.]])
```

There are a number of utility functions to create tensors in pytorch:

- **torch**.zeros(): creates a tensor filled with zeros.
- **torch**.ones(): creates a tensor filled with ones.

- **torch**.rand(): creates a tensor filled with random values from this range [0, 1).
- **torch**.full(): creates a tensor filled with a scalar value.
- **torch**.eye(): creates a square tensor with ones on the diagonal and zeros elsewhere.
- **torch**.arange(): creates a 1D tensor with evenly spaced values in a given range, space determined by step.
- **torch**.linspace(): creates a 1D tensor with evenly spaced values between a start and end value, space determined by number of values.

```
zeros = torch.zeros(2, 5)  # shape
ones = torch.ones(3, 4) # shape
randoms = torch.rand(2, 3) # shape
full = torch.full((3,4), 56.7) # shape, fill_value
I = torch.eye(3) # diagonal_size
arange = torch.arange(0, 10, 2) # start, stop, step
linspace = torch.linspace(0, 10, 20) # start, stop, number_of_values
empty = torch.empty(2,2) # shape: faster than zeros() or ones() because it does not i

print("zeros = torch.zeros(2, 5)\n", zeros, "\n\n",
      "ones = torch.ones(3, 4)\n", ones, "\n\n",
      "randoms = torch.rand(2, 3)\n", randoms, "\n\n",
      "full = torch.full((3, 4), 56.7)\n", full, "\n\n",
      "I = torch.eye(3)\n", I, "\n\n",
      "arange = torch.arange(0, 10, 2)\n", arange, "\n\n",
      "linspace = torch.linspace(0, 10, 20)\n", linspace, "\n\n",
      "empty = torch.empty(2,2)\n", empty)
```

```
zeros = torch.zeros(2, 5)
 tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])

 ones = torch.ones(3, 4)
 tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])

 randoms = torch.rand(2, 3)
 tensor([[0.1398, 0.2671, 0.9228],
        [0.5382, 0.0678, 0.6011]])

 full = torch.full((3, 4), 56.7)
 tensor([[56.7000, 56.7000, 56.7000, 56.7000],
        [56.7000, 56.7000, 56.7000, 56.7000],
        [56.7000, 56.7000, 56.7000, 56.7000]])

 I = torch.eye(3)
 tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

```
arange = torch.arange(0, 10, 2)
tensor([0, 2, 4, 6, 8])

linspace = torch.linspace(0, 10, 20)
tensor([ 0.0000,  0.5263,  1.0526,  1.5789,  2.1053,  2.6316,  3.1579,  3.6842,
         4.2105,  4.7368,  5.2632,  5.7895,  6.3158,  6.8421,  7.3684,  7.8947,
         8.4211,  8.9474,  9.4737, 10.0000])

empty = torch.empty(2,2)
tensor([[ 1.5863e-42,  0.0000e+00],
        [-1.8424e-37,  3.0688e-41]])
```

Quiz: Under each comment write the suitable script to create the said tensor

$$A = \begin{bmatrix} 1 & 2.2 & 9.6 \\ 4 & -7.2 & 6.3 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$ (How many ways?)

```
# A
A = torch.tensor([[1, 2.2, 9.6],
                  [4, -7.2, 6.3]])
print(A)

# B
B = torch.ones(2,2)
print(B)
```

```
tensor([[ 1.0000,  2.2000,  9.6000],
        [ 4.0000, -7.2000,  6.3000]])
tensor([[1., 1.],
        [1., 1.]])
```

```
# Create two random tensors
A = torch.tensor([[1, 2, 3], [4, 5, 6]])
B = A.clone() # Modifiying the clone does not affect the original tensor

A_shape = A.shape # A Size object containing tensor dimentions' sizes

# Addition
C = A + B
C_torch = torch.add(A, B)

# Subtraction
D = A - B
D_torch = torch.sub(A, B)

# Multiplication (element-wise)
E = A * B
```

```
E_torch = torch.mul(A, B)

# Division (element-wise)
F = A / B
F_torch = torch.div(A, B)

# Transpose
G = A.T
G_torch = torch.transpose(A, 0, 1)

# Matrix multiplication
H = A @ B.T
H_torch = torch.matmul(A, B.T)

# Print results
print("A: \n", A)
print("B: \n", B)
print("A.shape: \n", A_shape)
print("A + B: \n", C)
print("torch.add(A, B): \n", C_torch)
print("A - B: \n", D)
print("torch.sub(A, B): \n", D_torch)
print("A * B: \n", E)
print("torch.mul(A, B): \n", E_torch)
print("A / B: \n", F)
print("torch.div(A, B): \n", F_torch)
print("Transpose of A: \n", G)
print("torch.transpose(A, 0, 1): \n", G_torch)
print("A @ B.T: \n", H)
print("torch.matmul(A, B.T): \n", H_torch)
```

```
A:
 tensor([[1, 2, 3],
        [4, 5, 6]])
B:
 tensor([[1, 2, 3],
        [4, 5, 6]])
A.shape:
 torch.Size([2, 3])
A + B:
 tensor([[ 2,  4,  6],
        [ 8, 10, 12]])
torch.add(A, B):
 tensor([[ 2,  4,  6],
        [ 8, 10, 12]])
A - B:
 tensor([[0, 0, 0],
        [0, 0, 0]])
torch.sub(A, B):
 tensor([[0, 0, 0],
        [0, 0, 0]])
A * B:
 tensor([[ 1,  4,  9],
        [16, 25, 36]])
```

```
torch.mul(A, B):
 tensor([[ 1,  4,  9],
        [16, 25, 36]])
A / B:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])
torch.div(A, B):
 tensor([[1., 1., 1.],
        [1., 1., 1.]])
Transpose of A:
 tensor([[1, 4],
        [2, 5],
        [3, 6]])
torch.transpose(A, 0, 1):
 tensor([[1, 4],
        [2, 5],
        [3, 6]])
A @ B.T:
 tensor([[14, 32],
        [32, 77]])
torch.matmul(A, B.T):
 tensor([[14, 32],
        [32, 77]])
```

Quiz: Considering tensor A and B, implement the following formula:

$$((A+B)(A-B)^T)/I$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$$

```
import torch

# Create two tensors A and B
A = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])
B = A + 1

# Perform the calculation
c = torch.eye(2)
result = ((A + B) @ ((A - B).T)) / c

# Print the result
print(result)
```

```
tensor([[-15., -inf],
        [-inf, -33.]])
```

Reshaping tensors can be used to make batch operations easier (more on that later), but be careful that the data is reshaped in the order you expect:

```
rr = torch.arange(1, 16)
print("The shape is currently", rr.shape)
print("The contents are currently", rr)
print()
rr2 = rr.view(5, 3) # view is not a clone, it uses the same shared data
print("After reshaping, the shape is currently", rr2.shape)
print("The contents are currently", rr2)
print()
print('Changing the first value of rr will change the corresponding value in rr2')
rr[0] = 100
print(rr)
print(rr2)
```

```
The shape is currently torch.Size([15])
The contents are currently tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 1

After reshaping, the shape is currently torch.Size([5, 3])
The contents are currently tensor([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9],
        [10, 11, 12],
        [13, 14, 15]])

Changing the first value of rr will change the corresponding value in rr2
tensor([100,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,
         15])
tensor([[100,   2,   3],
        [  4,   5,   6],
        [  7,   8,   9],
        [ 10,  11,  12],
        [ 13,  14,  15]])
```

Finally, you can also inter-convert tensors with **NumPy arrays**:

```
import numpy as np

# numpy.ndarray --> torch.Tensor: [feed ndarray to torch.tensor]
arr = np.array([[1, 0, 5]])
data = torch.tensor(arr)
print("This is a torch.tensor", data)

# torch.Tensor --> numpy.ndarray: [use .numpy() on tensor]
new_arr = data.numpy()
print("This is a np.ndarray", new_arr)
```

```
This is a torch.tensor tensor([[1, 0, 5]])
This is a np.ndarray [[1 0 5]]
```

Vectorization

One of the reasons why we use **tensors** is *vectorized operations*: operations that be conducted in parallel over a particular dimension of a tensor.

```
data = torch.arange(1, 36, dtype=torch.float32).reshape(5, 7)
print("Data is:", data)

# We can perform operations like *sum* over each row...
print("Taking the sum over columns:")
print(data.sum(dim=0))

# or over each column.
print("Taking the sum over rows:")
print(data.sum(dim=1))

# Other operations are available:
print("Taking the std over rows:")
print(data.std(dim=1))
```

```
Data is: tensor([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14.],
        [15., 16., 17., 18., 19., 20., 21.],
        [22., 23., 24., 25., 26., 27., 28.],
        [29., 30., 31., 32., 33., 34., 35.]])
Taking the sum over columns:
tensor([ 75.,  80.,  85.,  90.,  95., 100., 105.])
Taking the sum over rows:
tensor([ 28.,  77., 126., 175., 224.])
Taking the std over rows:
tensor([2.1602, 2.1602, 2.1602, 2.1602, 2.1602])
```

** Without specifying dimentions, it just sum all the values **

```
data.sum()
```

```
tensor(630.)
```

Quiz: Write code that creates a `torch.tensor` with the following contents: $\begin{bmatrix} 1 & 2.2 & 9.6 \\ 4 & -7.2 & 6.3 \end{bmatrix}$

Now compute the average of each row (`.mean()`) and each column.

What's the shape of the results?

```
tens = torch.tensor([[1, 2.2, 9.6],
                     [4, -7.2, 6.3]])

row_avg = tens.mean(dim = 0)
col_avg = tens.mean(dim = 1)

print(row_avg.shape)
print(row_avg)

print(col_avg.shape)
print(col_avg)
```

```
torch.Size([3])
tensor([ 2.5000, -2.5000,  7.9500])
torch.Size([2])
tensor([4.2667, 1.0333])
```

Indexing & Slicing

You can access arbitrary elements of a tensor using the `[]` operator.

```
matr = torch.arange(1, 16).view(5, 3)
print(matr)
```

```
tensor([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9],
        [10, 11, 12],
        [13, 14, 15]])
```

```
matr[0]
```

```
tensor([1, 2, 3])
```

```
matr[0, :]
```

```
tensor([1, 2, 3])
```

```
matr[:, 0]
```

```
tensor([ 1,  4,  7, 10, 13])
```

```
matr[0:3]
```

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

```
matr[:, 0:2]
```

```
tensor([[ 1,  2],
        [ 4,  5],
        [ 7,  8],
        [10, 11],
        [13, 14]])
```

```
matr[0:3, 0:2]
```

```
tensor([[1, 2],
        [4, 5],
        [7, 8]])
```

It look likes they are doing the same thing?

```
print(matr[0][2])
print(matr[0,2])
```

```
tensor(3)
tensor(3)
```

```
matr[0:3, 2]
```

```
tensor([3, 6, 9])
```

```
matr[0:3][2]
```

```
tensor([7, 8, 9])
```

```
matr[0:3]
```

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

```
matr[[0, 2, 4]]
```

```
tensor([[ 1,  2,  3],
        [ 7,  8,  9],
        [13, 14, 15]])
```

Accessing python scalar value in a tensor

```
matr[0, 0]
```

```
tensor(1)
```

```
matr[0, 0].item()
```

```
1
```

Quiz: Write code that creates a `torch.tensor` with the following contents: $\begin{bmatrix} 1 & 2.2 & 9.6 \\ 4 & -7.2 & 6.3 \end{bmatrix}$

How do you get the first column? The first row?

```
tens = torch.tensor([[1, 2.2, 9.6],
                     [4, -7.2, 6.3]])

first_col = tens[:, 0]
first_row = tens[0]

print(tens)
print(first_col)
print(first_row)
```

```
tensor([[ 1.0000,  2.2000,  9.6000],
        [ 4.0000, -7.2000,  6.3000]])
tensor([1., 4.])
tensor([1.0000, 2.2000, 9.6000])
```

Autograd

Pytorch is well-known for its automatic differentiation feature. We can call the `backward()` method to ask `PyTorch` to calculate the gradients, which are then stored in the `grad` attribute.

```
# Create an example tensor
# requires_grad parameter tells PyTorch to store gradients
x = torch.tensor([2.], requires_grad=True)

# Print the gradient if it is calculated
```

```
# Currently None since x is a scalar
print(x.grad)
```

```
None
```

```
# Calculating the gradient of y with respect to x
y = x * x * 3 # 3x^2
y.backward()
print(x.grad) # d(y)/d(x) = d(3x^2)/d(x) = 6x = 12
```

```
tensor([12.])
```

Let's run backprop from a different tensor again to see what happens.

```
z = x * x * 3 # 3x^2
z.backward()
print(x.grad)
```

```
tensor([24.])
```

We can see that the `x.grad` is updated to be the sum of the gradients calculated so far. When we run backprop in a neural network, we sum up all the gradients for a particular neuron before making an update. This is exactly what is happening here! This is also the reason why we need to run `zero_grad()` in every training iteration (more on this later). Otherwise our gradients would keep building up from one training iteration to the other, which would cause our updates to be wrong.

```
# let's have a look at a bit more sophisticated example:
# clearing cumulative grads
print(x.grad)
x.grad.zero_()
print(x.grad)
```

```
tensor([24.])
tensor([0.])
```

```
y = torch.tensor(3., requires_grad=True)
y
```

```
tensor(3., requires_grad=True)
```

```
f = y * x * 10
f.backward()
print(x)
print(x.grad)
print()
print(y)
print(y.grad)
```

```
tensor([2.], requires_grad=True)
tensor([30.])

tensor(3., requires_grad=True)
tensor(20.)
```

```
x.grad.zero_()
f = 3 * x
g = 10 * f
g.backward()

print(x)
print(x.grad)
```

```
tensor([2.], requires_grad=True)
tensor([30.])
```

Neural Network Module

So far we have looked into the tensors, their properties and basic operations on tensors. These are especially useful to get familiar with if we are building the layers of our network from scratch. We will utilize these in Assignment 3, but moving forward, we will use predefined blocks in the `torch.nn` module of `PyTorch`. We will then put together these blocks to create complex networks. Let's start by importing this module with an alias so that we don't have to type `torch` every time we use it.

```
import torch.nn as nn
```

Linear Layer

We can use `nn.Linear(H_in, H_out)` to create a a linear layer. This will take a matrix of `(N, *, H_in)` dimensions and output a matrix of `(N, *, H_out)`. The `*` denotes that there could be arbitrary number of dimensions in between. The linear layer performs the operation `Ax+b`, where `A` and `b` are initialized randomly. If we don't want the linear layer to learn the bias parameters, we can initialize our layer with `bias=False`.

```
# Create the inputs
input = torch.ones(2,3,4)
# N* H_in -> N*H_out


# Make a linear layers transforming N,*,H_in dimensinal inputs to N,*,H_out
# dimensional outputs
linear = nn.Linear(4, 2)
linear_output = linear(input)
linear_output
```

```
tensor([[[-0.0705,  0.4311],
         [-0.0705,  0.4311],
         [-0.0705,  0.4311]],

        [[-0.0705,  0.4311],
         [-0.0705,  0.4311],
         [-0.0705,  0.4311]]], grad_fn=<ViewBackward0>)
```

```
list(linear.parameters()) # Ax + b
```

```
[Parameter containing:
 tensor([[-0.1591, -0.1102,  0.2865,  0.0761],
         [-0.4066,  0.2974,  0.1917,  0.0553]], requires_grad=True),
 Parameter containing:
 tensor([-0.1639,  0.2933], requires_grad=True)]
```

Other Module Layers

There are several other preconfigured layers in the `nn` module. Some commonly used examples are `nn.Conv2d`, `nn.ConvTranspose2d`, `nn.BatchNorm1d`, `nn.BatchNorm2d`, `nn.Upsample` and `nn.MaxPool2d` among many others. We will learn more about these as we progress in the course. For now, the only important thing to remember is that we can treat each of these layers as plug and play components: we will be providing the required dimensions and `PyTorch` will take care of setting them up.

Activation Function Layer

We can also use the `nn` module to apply activations functions to our tensors. Activation functions are used to add non-linearity to our network. Some examples of activations functions are `nn.ReLU()`, `nn.Sigmoid()` and `nn.LeakyReLU()`. Activation functions operate on each element seperately, so the shape of the tensors we get as an output are the same as the ones we pass in.

```
linear_output
```

```
tensor([[[-0.0705,  0.4311],
         [-0.0705,  0.4311],
         [-0.0705,  0.4311]],

        [[-0.0705,  0.4311],
         [-0.0705,  0.4311],
         [-0.0705,  0.4311]]], grad_fn=<ViewBackward0>)
```

```
sigmoid = nn.Sigmoid()
output = sigmoid(linear_output)
output
```

```
tensor([[[0.4824, 0.6061],
         [0.4824, 0.6061],
         [0.4824, 0.6061]],

        [[0.4824, 0.6061],
         [0.4824, 0.6061],
         [0.4824, 0.6061]]], grad_fn=<SigmoidBackward0>)
```

Putting the Layers Together

So far we have seen that we can create layers and pass the output of one as the input of the next. Instead of creating intermediate tensors and passing them around, we can use `nn.Sequentual`, which does exactly that.

```
block = nn.Sequential(
    nn.Linear(4, 2),
    nn.Sigmoid()
)

input = torch.ones(2,3,4)
output = block(input)
output
```

```
tensor([[[0.5778, 0.4573],
         [0.5778, 0.4573],
         [0.5778, 0.4573]],

        [[0.5778, 0.4573],
         [0.5778, 0.4573],
         [0.5778, 0.4573]]], grad_fn=<SigmoidBackward0>)
```

Custom Modules

Instead of using the predefined modules, we can also build our own by extending the `nn.Module` class. For example, we can build the `nn.Linear` (which also extends `nn.Module`) on our own using the tensor introduced earlier! We can also build new, more complex modules, such as a custom neural network. You will be practicing these in the later assignment.

To create a custom module, the first thing we have to do is to extend the `nn.Module`. We can then initialize our parameters in the `__init__` function, starting with a call to the `__init__` function of the super class. All the class attributes we define which are `nn` module objects are treated as parameters, which can be learned during the training. Tensors are not parameters, but they can be turned into parameters if they are wrapped in `nn.Parameter` class.

All classes extending `nn.Module` are also expected to implement a `forward(x)` function, where `x` is a tensor. This is the function that is called when a parameter is passed to our module, such as in `model(x)`.

```
class MultilayerPerceptron(nn.Module):

  def __init__(self, input_size, hidden_size):
    # Call to the __init__ function of the super class
    super(MultilayerPerceptron, self).__init__()

    # Bookkeeping: Saving the initialization parameters
    self.input_size = input_size
    self.hidden_size = hidden_size

    # Defining of our model
    # There isn't anything specific about the naming of `self.model`. It could
    # be something arbitrary.
    self.model = nn.Sequential(
        nn.Linear(self.input_size, self.hidden_size),
        nn.ReLU(),
        nn.Linear(self.hidden_size, self.input_size),
        nn.Sigmoid()
    )

  def forward(self, x):
    output = self.model(x)
    return output
```

Here is an alternative way to define the same class. You can see that we can replace `nn.Sequential` by defining the individual layers in the `__init__` method and connecting the in the `forward` method.

```
class MultilayerPerceptron(nn.Module):

  def __init__(self, input_size, hidden_size):
    # Call to the __init__ function of the super class
    super(MultilayerPerceptron, self).__init__()
```

```
    # Bookkeeping: Saving the initialization parameters
    self.input_size = input_size
    self.hidden_size = hidden_size

    # Defining of our layers
    self.linear = nn.Linear(self.input_size, self.hidden_size)
    self.relu = nn.ReLU()
    self.linear2 = nn.Linear(self.hidden_size, self.input_size)
    self.sigmoid = nn.Sigmoid()

  def forward(self, x):
    linear = self.linear(x)
    relu = self.relu(linear)
    linear2 = self.linear2(relu)
    output = self.sigmoid(linear2)
    return output
```

Now that we have defined our class, we can instantiate it and see what it does.

```
# Make a sample input
input = torch.randn(2, 5)

# Create our model
model = MultilayerPerceptron(5, 3)

# Pass our input through our model
model(input)
```

```
tensor([[0.4529, 0.3961, 0.3687, 0.3740, 0.4482],
        [0.5124, 0.3533, 0.3668, 0.3947, 0.4296]], grad_fn=<SigmoidBackward0>)
```

We can inspect the parameters of our model with `named_parameters()` and `parameters()` methods.

```
list(model.named_parameters())
```

```
[('linear.weight',
  Parameter containing:
  tensor([[-0.2469,  0.4323, -0.0636,  0.1376, -0.1401],
          [-0.1910,  0.4125,  0.2824, -0.1042, -0.1327],
          [-0.1135,  0.1658,  0.4416,  0.3515, -0.1316]], requires_grad=True)),
 ('linear.bias',
  Parameter containing:
  tensor([-0.3198,  0.1241, -0.2302], requires_grad=True)),
 ('linear2.weight',
  Parameter containing:
  tensor([[ 0.2890,  0.3623, -0.5223],
          [-0.1433, -0.5252, -0.3723],
          [-0.5033,  0.4500,  0.1048],
```

```
            [ 0.1266,  0.0574, -0.3398],
            [ 0.2241,  0.5479,  0.5321]], requires_grad=True)),
 ('linear2.bias',
  Parameter containing:
  tensor([ 0.3906,  0.4117,  0.2708, -0.5031, -0.3149], requires_grad=True))]
```

Optimization

We have showed how gradients are calculated with the `backward()` function. Having the gradients isn't enought for our models to learn. We also need to know how to update the parameters of our models. This is where the optimizers comes in. `torch.optim` module contains several optimizers that we can use. Some popular examples are `optim.SGD` and `optim.Adam`. When initializing optimizers, we pass our model parameters, which can be accessed with `model.parameters()`, telling the optimizers which values it will be optimizing. Optimizers also has a learning rate (`lr`) parameter, which determines how big of an update will be made in every step. Different optimizers have different hyperparameters as well.

```
import torch.optim as optim
```

After we have our optimization function, we can define a `loss` that we want to optimize for. We can either define the loss ourselves, or use one of the predefined loss function in `PyTorch`, such as `nn.BCELoss()`. Let's put everything together now! We will start by creating some dummy data.

```
# Create the y data
y = torch.ones(10, 5)

# Add some noise to our goal y to generate our x
# We want our model to predict our original data, albeit the noise
x = y + torch.randn_like(y)
x
```

```
tensor([[ 1.4481e+00,  1.5741e+00,  2.7479e+00,  3.7356e-01,  5.6945e-01],
        [-8.3364e-02,  2.1854e+00,  1.6955e+00,  2.4019e+00,  1.1410e-02],
        [ 1.8081e+00,  1.0321e+00,  2.3767e+00,  8.3225e-01,  1.7766e+00],
        [ 5.3410e-02,  1.3933e+00,  5.3727e-01,  1.1749e-01,  1.9335e-01],
        [ 9.1281e-01,  6.1750e-01,  6.3051e-01, -3.8757e-01,  7.4699e-01],
        [ 1.0727e+00,  5.2872e-01,  2.2358e-01,  8.1719e-01, -7.8099e-02],
        [ 2.1987e+00,  1.2383e+00,  1.4726e+00,  2.4384e+00,  6.0987e-01],
        [ 3.5173e-02,  1.4249e+00, -2.2269e-01,  1.2916e+00, -3.7038e-01],
        [ 1.0281e+00,  1.6842e-01,  1.0657e+00,  9.2184e-01,  2.4698e-01],
        [ 3.9331e-01, -4.9450e-01,  1.1048e+00,  1.5435e-03,  6.2793e-01]])
```

Now, we can define our model, optimizer and the loss function.

```
# Instantiate the model
model = MultilayerPerceptron(5, 3)

# Define the optimizer
adam = optim.Adam(model.parameters(), lr=1e-1)

# Define loss using a predefined loss function
loss_function = nn.BCELoss()

# Calculate how our model is doing now
y_pred = model(x)
loss_function(y_pred, y).item()
```

```
0.6880847215652466
```

QUIZ: With nn.BCELoss, calculate the loss over first 3 columns of the second row of values

```
bce_loss = nn.BCELoss()
selected_x = x
selected_y = y[1, 0:3]
predicted_y = model(x)
loss = loss_function(predicted_y[1, 0:3], selected_y)
print(loss)
```

```
tensor(0.7208, grad_fn=<BinaryCrossEntropyBackward0>)
```

Let's see if we can have our model achieve a smaller loss. Now that we have everything we need, we can setup our training loop.

```
# Set the number of epoch, which determines the number of training iterations
n_epoch = 10

for epoch in range(n_epoch):
  # Set the gradients to 0
  adam.zero_grad()

  # Get the model predictions
  y_pred = model(x)

  # Get the loss
  loss = loss_function(y_pred, y)

  # Print stats
  print(f"Epoch {epoch}: traing loss: {loss}")

  # Compute the gradients
  loss.backward()

  # Take a step to optimize the weights
```

```
    # Take a step to optimize the weights
    adam.step()
```

```
Epoch 0: traing loss: 0.8072628974914551
Epoch 1: traing loss: 0.6672016978263855
Epoch 2: traing loss: 0.5206297039985657
Epoch 3: traing loss: 0.361960768699646
Epoch 4: traing loss: 0.23093898594379425
Epoch 5: traing loss: 0.13422948122024536
Epoch 6: traing loss: 0.08013811707496643
Epoch 7: traing loss: 0.05132236331701279
Epoch 8: traing loss: 0.03333953022956848
Epoch 9: traing loss: 0.02155839279294014
```

```
list(model.named_parameters())
```

```
[('linear.weight',
  Parameter containing:
  tensor([[ 1.0493,  1.0014,  0.6206,  0.8807,  0.6239],
          [-0.8344, -0.2371, -0.4344,  0.1150,  0.3531],
          [ 0.5409,  0.9157,  0.6304,  0.9452, -0.3108]], requires_grad=True)),
 ('linear.bias',
  Parameter containing:
  tensor([ 0.6889, -0.6511,  1.2551], requires_grad=True)),
 ('linear2.weight',
  Parameter containing:
  tensor([[ 1.2552,  0.0248,  1.0041],
          [ 1.0316,  0.8975,  1.2364],
          [ 1.4061,  0.4072,  1.1786],
          [ 1.0408, -0.1008,  1.0565],
          [ 0.7105,  0.2473,  1.2403]], requires_grad=True)),
 ('linear2.bias',
  Parameter containing:
  tensor([0.7667, 0.9808, 0.2451, 0.7969, 0.3430], requires_grad=True))]
```

You can see that our loss is decreasing. Let's check the predictions of our model now and see if they are close to our original `y`, which was all `1s`.

```
# See how our model performs on the training data
y_pred = model(x)
y_pred
```

```
tensor([[0.9996, 0.9996, 0.9997, 0.9992, 0.9975],
        [0.9688, 0.9659, 0.9630, 0.9570, 0.8938],
        [0.9119, 0.9485, 0.8897, 0.9205, 0.9074],
        [0.9998, 0.9998, 0.9999, 0.9996, 0.9984],
        [0.9999, 0.9999, 1.0000, 0.9998, 0.9995],
        [0.9997, 0.9998, 0.9998, 0.9995, 0.9990],
        [0.9997, 0.9996, 0.9998, 0.9994, 0.9975],
        [1.0000, 1.0000, 1.0000, 1.0000, 0.9999],
```

```
            [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
            [1.0000, 1.0000, 1.0000, 1.0000, 1.0000]], grad_fn=<SigmoidBackward0>)
```

```
# Create test data and check how our model performs on it
x2 = y + torch.randn_like(y)
y_pred = model(x2)
y_pred
```

```
    tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
            [1.0000, 1.0000, 1.0000, 1.0000, 0.9999],
            [1.0000, 1.0000, 1.0000, 0.9999, 0.9997],
            [0.9999, 0.9999, 1.0000, 0.9999, 0.9995],
            [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
            [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
            [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
            [0.9999, 0.9999, 0.9999, 0.9998, 0.9990],
            [1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
            [1.0000, 1.0000, 1.0000, 1.0000, 0.9999]], grad_fn=<SigmoidBackward0>)
```

Great! Looks like our model almost perfectly learned to filter out the noise from the `x` that we passed in!

Intermission: 5 min

Demo: Word Window Classification

Until this part of the notebook, we have learned the fundamentals of PyTorch and built a basic network solving a toy task. Now we will attempt to solve an example NLP task. Here are the things we will learn:

1. Data: Creating a Dataset of Batched Tensors
2. Modeling
3. Training
4. Prediction

In this section, our goal will be to train a model that will find the words in a sentence corresponding to a `LOCATION`, which will be always of span `1` (meaning that `San Fransisco` won't be recognized as a `LOCATION`). Our task is called `Word Window Classification` for a reason. Instead of letting our model to only take a look at one word in each forward pass, we would like it to be able to consider the context of the word in question. That is, for each word, we want our model to be aware of the surrounding words. Let's dive in!

Data

The very first task of any machine learning project is to set up our training set. Usually, there will be a training corpus we will be utilizing. In NLP tasks, the corpus would generally be a `.txt` or `.csv` file where each row corresponds to a sentence or a tabular datapoint. In our toy task, we will assume that we have already read our data and the corresponding labels into a `Python` list.

```
# Our raw data, which consists of sentences
corpus = [
          "We always come to Paris",
          "The professor is from Australia",
          "I live in Stanford",
          "He comes from Taiwan",
          "The capital of Turkey is Ankara"
         ]
```

Preprocessing

To make it easier for our models to learn, we usually apply a few preprocessing steps to our data. This is especially important when dealing with text data. Here are some examples of text preprocessing:

- **Tokenization**: Tokenizing the sentences into words.
- **Lowercasing**: Changing all the letters to be lowercase. Example?
- **Noise removal:** Removing special characters (such as punctuations). Example?
- **Stop words removal**: Removing commonly used words. Example?

Which preprocessing steps are necessary is determined by the task at hand. For example, although it is useful to remove special characters in some tasks, for others they may be important (for example, if we are dealing with multiple languages). For our task, we will lowercase our words and tokenize.

```
# The preprocessing function we will use to generate our training examples
# Our function is a simple one, we lowercase the letters
# and then tokenize the words.
def preprocess_sentence(sentence):
  return sentence.lower().split()

# Create our training set
train_sentences = [preprocess_sentence(sent) for sent in corpus]
train_sentences
```

```
[['we', 'always', 'come', 'to', 'paris'],
 ['the', 'professor', 'is', 'from', 'australia'],
 ['i', 'live', 'in', 'stanford'],
```

```
 ['he', 'comes', 'from', 'taiwan'],
 ['the', 'capital', 'of', 'turkey', 'is', 'ankara']]
```

For each training example we have, we should also have a corresponding label. Recall that the goal of our model was to determine which words correspond to a `LOCATION`. That is, we want our model to output `0` for all the words that are not `LOCATION`s and `1` for the ones that are `LOCATION`s.

```
# Set of locations that appear in our corpus
locations = set(["australia", "ankara", "paris", "stanford", "taiwan", "turkey"])

# Our train labels
train_labels = [[1 if word in locations else 0 for word in sent] for sent in train_se
train_labels
```

```
[[0, 0, 0, 0, 1],
 [0, 0, 0, 0, 1],
 [0, 0, 0, 1],
 [0, 0, 0, 1],
 [0, 0, 0, 1, 0, 1]]
```

Converting Words to Embeddings

Let's look at our training data a little more closely. Each datapoint we have is a sequence of words. On the other hand, we know that machine learning models work with numbers in vectors. How are we going to turn words into numbers? You may be thinking embeddings and you are right!

Imagine that we have an embedding lookup table `E`, where each row corresponds to an embedding. That is, each word in our vocabulary would have a corresponding embedding row `i` in this table. Whenever we want to find an embedding for a word, we will follow these steps:

1. Find the corresponding index `i` of the word in the embedding table: `word->index`.
2. Index into the embedding table and get the embedding: `index->embedding`.

Let's look at the first step. We should assign all the words in our vocabulary to a corresponding index. We can do it as follows:

1. Find all the unique words in our corpus. How?
2. Assign an index to each.

```
# Find all the unique words in our corpus
vocabulary = set(w for s in train_sentences for w in s)
vocabulary
```

```
{'always',
 'ankara',
 'australia',
```

```
     'capital',
     'come',
     'comes',
     'from',
     'he',
     'i',
     'in',
     'is',
     'live',
     'of',
     'paris',
     'professor',
     'stanford',
     'taiwan',
     'the',
     'to',
     'turkey',
     'we'}
```

`vocabulary` now contains all the words in our corpus. On the other hand, during the test time, we can see words that are not contained in our vocabulary. If we can figure out a way to represent the unknown words, our model can still reason about whether they are a `LOCATION` or not, since we are also looking at the neighboring words for each prediction.

We introduce a special token, `<unk>`, to tackle the words that are out of vocabulary. We could pick another string for our unknown token if we wanted. The only requirement here is that our token should be unique: we should only be using this token for unknown words. We will also add this special token to our vocabulary.

```
# Add the unknown token to our vocabulary
vocabulary.add("<unk>")
```

Earlier we mentioned that our task was called `Word Window Classification` because our model is looking at the surroundings words in addition to the given word when it needs to make a prediction.

For example, let's take the sentence "We always come to Paris". The corresponding training label for this sentence is `0, 0, 0, 0, 1` since only Paris, the last word, is a `LOCATION`. In one pass (meaning a call to `forward()`), our model will try to generate the correct label for one word. Let's say our model is trying to generate the correct label `1` for `Paris`. If we only allow our model to see `Paris`, but nothing else, we will miss out on the important information that the word `to` often times appears with `LOCATION`s.

Word windows allow our model to consider the surrounding `+N` or `-N` words of each word when making a prediction. In our earlier example for `Paris`, if we have a window size of 1, that means our model will look at the words that come immediately before and after `Paris`, which are `to`, and, well, nothing. Now, this raises another issue. `Paris` is at the end of our sentence, so there isn't another word following it. Remember that we define the input dimensions of our `PyTorch` models when we are initializing them. If we set the window size to be `1`, it means that our model will be accepting `3` words in every pass. We cannot have our model expect `2` words from time to time.

The solution is to introduce a special token, such as `<pad>`, that will be added to our sentences to make sure that every word has a valid window around them. Similar to `<unk>` token, we could pick another string for our pad token if we wanted, as long as we make sure it is used for a unique purpose.

```
# Add the <pad> token to our vocabulary
vocabulary.add("<pad>")

# Function that pads the given sentence
# We are introducing this function here as an example
# We will be utilizing it later in the tutorial
def pad_window(sentence, window_size, pad_token="<pad>"):
  window = [pad_token] * window_size
  return window + sentence + window

# Show padding example
window_size = 2
pad_window(train_sentences[0], window_size=window_size)
```

```
['<pad>', '<pad>', 'we', 'always', 'come', 'to', 'paris', '<pad>', '<pad>']
```

Now that our vocabularly is ready, let's assign an index to each of our words.

```
# We are just converting our vocabularly to a list to be able to index into it
# Sorting is not necessary, we sort to show an ordered word_to_ind dictionary
# That being said, we will see that having the index for the padding token
# be 0 is convenient as some PyTorch functions use it as a default value
# such as nn.utils.rnn.pad_sequence, which we will cover in a bit
ix_to_word = sorted(list(vocabulary))

# Creating a dictionary to find the index of a given word
word_to_ix = {word: ind for ind, word in enumerate(ix_to_word)}
word_to_ix
```

```
{'<pad>': 0,
 '<unk>': 1,
 'always': 2,
 'ankara': 3,
 'australia': 4,
 'capital': 5,
 'come': 6,
 'comes': 7,
 'from': 8,
```

```
     'he': 9,
     'i': 10,
     'in': 11,
     'is': 12,
     'live': 13,
     'of': 14,
     'paris': 15,
     'professor': 16,
     'stanford': 17,
     'taiwan': 18,
     'the': 19,
     'to': 20,
     'turkey': 21,
     'we': 22}
```

```
print(ix_to_word[1])
print(word_to_ix['<unk>'])
```

```
<unk>
1
```

Great! We are ready to convert our training sentences into a sequence of indices corresponding to each token.

```
# Given a sentence of tokens, return the corresponding indices
def convert_token_to_indices(sentence, word_to_ix):
  indices = []
  for token in sentence:
    # Check if the token is in our vocabularly. If it is, get it's index.
    # If not, get the index for the unknown token.
    if token in word_to_ix:
      index = word_to_ix[token]
    else:
      index = word_to_ix["<unk>"]
    indices.append(index)
  return indices

# More compact version of the same function
def _convert_token_to_indices(sentence, word_to_ix):
  return [word_to_ix.get(token, word_to_ix["<unk>"]) for token in sentence]

# Show an example
example_sentence = ["we", "always", "come", "to", "kuwait"]
example_indices = convert_token_to_indices(example_sentence, word_to_ix)
restored_example = [ix_to_word[ind] for ind in example_indices]

print(f"Original sentence is: {example_sentence}")
print(f"Going from words to indices: {example_indices}")
print(f"Going from indices to words: {restored_example}")
```

```
Original sentence is: ['we', 'always', 'come', 'to', 'kuwait']
Going from words to indices: [22, 2, 6, 20, 1]
Going from indices to words: ['we', 'always', 'come', 'to', '<unk>']
```

In the example above, `kuwait` shows up as `<unk>`, because it is not included in our vocabulary.

Let's convert our `train_sentences` to `example_padded_indices`.

```
# Converting our sentences to indices
example_padded_indices = [convert_token_to_indices(s, word_to_ix) for s in train_sent
example_padded_indices
```

```
[[22, 2, 6, 20, 15],
 [19, 16, 12, 8, 4],
 [10, 13, 11, 17],
 [9, 7, 8, 18],
 [19, 5, 14, 21, 12, 3]]
```

Now that we have an index for each word in our vocabularly, we can create an embedding table with `nn.Embedding` class in `PyTorch`. It is called as follows `nn.Embedding(num_words, embedding_dimension)` where `num_words` is the number of words in our vocabulary and the `embedding_dimension` is the dimension of the embeddings we want to have. There is nothing fancy about `nn.Embedding`: it is just a wrapper class around a trainabe `NxE` dimensional tensor, where `N` is the number of words in our vocabulary and `E` is the number of embedding dimensions. This table is initially random, but it will change over time. As we train our network, the gradients will be backpropagated all the way to the embedding layer, and hence our word embeddings would be updated. We will initiliaze the embedding layer we will use for our model in our model, but we are showing an example here.

```
# Creating an embedding table for our words
embedding_dim = 5
embeds = nn.Embedding(len(vocabulary), embedding_dim)

# Printing the parameters in our embedding table
list(embeds.named_parameters())
```

```
[('weight',
  Parameter containing:
  tensor([[ 0.1569, -0.6988,  1.1663,  0.8902,  0.5453],
          [-2.1172,  0.7566,  0.3789, -1.0536,  0.9380],
          [ 0.6544, -1.2428,  1.4539,  2.2869, -0.1435],
          [ 1.2256, -1.8408,  0.1393, -0.1612,  0.4784],
          [ 0.4751, -0.5841, -1.4732, -0.5266, -1.1685],
          [-0.2660,  0.2014,  0.2694,  0.7087, -0.7643],
          [ 0.1409, -0.1442, -0.2889,  1.5485, -0.6084],
          [ 0.5124, -1.0330, -0.5464, -0.0711,  1.7220],
          [ 1.1607, -1.9755, -1.5873,  0.8921, -0.9809],
```

```
        [ 0.2979,  1.8946, -0.7578, -1.5342,  0.6010],
        [-0.1775, -1.6259, -0.2892,  1.6260, -0.8319],
        [ 0.0848, -1.2990, -2.1170,  0.3250,  2.7506],
        [ 2.1503, -2.1468, -1.1847, -0.9481, -0.4090],
        [-0.1488,  0.7945, -0.8798, -1.1888, -0.1931],
        [ 0.2187, -1.2733,  1.4725, -0.6138,  2.0192],
        [-0.8501, -1.2288, -0.8952,  0.8554, -0.0104],
        [-0.6896, -0.4180,  1.6557, -1.2526, -0.3655],
        [-0.3766, -0.9682, -1.6094,  0.0058,  0.7992],
        [-1.2860,  0.7100, -0.7964, -0.1671,  0.2793],
        [-0.9450, -1.4015, -0.4525,  2.4425,  0.2647],
        [-1.3036,  0.3038,  0.8656, -1.5797,  0.3329],
        [-0.3678,  1.1626, -0.6572, -0.0705,  1.3600],
        [ 1.0464,  0.7210,  1.3009,  1.3571,  0.7237]], requires_grad=True))]
```

To get the word embedding for a word in our vocabulary, all we need to do is to create a lookup tensor. The lookup tensor is just a tensor containing the index we want to look up `nn.Embedding` class expects an index tensor that is of type Long Tensor, so we should create our tensor accordingly.

```
# Get the embedding for the word Paris
index = word_to_ix["paris"]
index_tensor = torch.tensor(index, dtype=torch.long)
paris_embed = embeds(index_tensor)
paris_embed
```

```
tensor([-0.8501, -1.2288, -0.8952,  0.8554, -0.0104],
       grad_fn=<EmbeddingBackward0>)
```

```
# embeds(index) # throws: TypeError: embedding(): argument 'indices' (position 2) mus
```

```
# We can also get multiple embeddings at once
index_paris = word_to_ix["paris"]
index_ankara = word_to_ix["ankara"]
indices = [index_paris, index_ankara]
indices_tensor = torch.tensor(indices, dtype=torch.long)
embeddings = embeds(indices_tensor)
embeddings
```

```
tensor([[-0.8501, -1.2288, -0.8952,  0.8554, -0.0104],
        [ 1.2256, -1.8408,  0.1393, -0.1612,  0.4784]],
       grad_fn=<EmbeddingBackward0>)
```

Usually, we define the embedding layer as part of our model, which you will see in the later sections of our notebook.

Batching Sentences

We have learned about batches in class. Waiting our whole training corpus to be processed before making an update is constly. On the other hand, updating the parameters after every training example causes the loss to be less stable between updates. To combat these issues, we instead update our parameters after training on a batch of data. This allows us to get a better estimate of the gradient of the global loss. In this section, we will learn how to structure our data into batches using the `torch.util.data.DataLoader` class.

We will be calling the `DataLoader` class as follows: `DataLoader(data, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)`. The `batch_size` parameter determines the number of examples per batch. In every epoch, we will be iterating over all the batches using the `DataLoader`. The order of batches is deterministic by default, but we can ask `DataLoader` to shuffle the batches by setting the `shuffle` parameter to `True`. This way we ensure that we don't encounter a bad batch multiple times.

If provided, `DataLoader` passes the batches it prepares to the `collate_fn`. We can write a custom function to pass to the `collate_fn` parameter in order to print stats about our batch or perform extra processing. In our case, we will use the `collate_fn` to:

1. Window pad our train sentences.
2. Convert the words in the training examples to indices.
3. Pad the training examples so that all the sentences and labels have the same length. Similarly, we also need to pad the labels. This creates an issue because when calculating the loss, we need to know the actual number of words in a given example. We will also keep track of this number in the function we pass to the `collate_fn` parameter.

Because our version of the `collate_fn` function will need to access to our `word_to_ix` dictionary (so that it can turn words into indices), we will make use of the `partial` function in `Python`, which passes the parameters we give to the function we pass it.

```
from torch.utils.data import DataLoader
from functools import partial

def custom_collate_fn(batch, window_size, word_to_ix):
  # Break our batch into the training examples (x) and labels (y)
  # We are turning our x and y into tensors because nn.utils.rnn.pad_sequence
  # method expects tensors. This is also useful since our model will be
  # expecting tensor inputs.
  x, y = zip(*batch)

  # Now we need to window pad our training examples. We have already defined a
  # function to handle window padding. We are including it here again so that
```

```
  # everything is in one place.
  def pad_window(sentence, window_size, pad_token="<pad>"):
    window = [pad_token] * window_size
    return window + sentence + window

  # Pad the train examples.
  x = [pad_window(s, window_size=window_size) for s in x]

  # Now we need to turn words in our training examples to indices. We are
  # copying the function defined earlier for the same reason as above.
  def convert_tokens_to_indices(sentence, word_to_ix):
    return [word_to_ix.get(token, word_to_ix["<unk>"]) for token in sentence]

  # Convert the train examples into indices.
  x = [convert_tokens_to_indices(s, word_to_ix) for s in x]

  # We will now pad the examples so that the lengths of all the example in
  # one batch are the same, making it possible to do matrix operations.
  # We set the batch_first parameter to True so that the returned matrix has
  # the batch as the first dimension.
  pad_token_ix = word_to_ix["<pad>"]

  # pad_sequence function expects the input to be a tensor, so we turn x into one
  x = [torch.LongTensor(x_i) for x_i in x]
  x_padded = nn.utils.rnn.pad_sequence(x, batch_first=True, padding_value=pad_token_i

  # We will also pad the labels. Before doing so, we will record the number
  # of labels so that we know how many words existed in each example.
  lengths = [len(label) for label in y]
  lenghts = torch.LongTensor(lengths)

  y = [torch.LongTensor(y_i) for y_i in y]
  y_padded = nn.utils.rnn.pad_sequence(y, batch_first=True, padding_value=0)

  # We are now ready to return our variables. The order we return our variables
  # here will match the order we read them in our training loop.
  return x_padded, y_padded, lenghts
```

This function seems long, but it really doesn't have to be. Check out the alternative version below where we remove the extra function declarations and comments.

```
def _custom_collate_fn(batch, window_size, word_to_ix):
  # Prepare the datapoints
  x, y = zip(*batch)
  x = [pad_window(s, window_size=window_size) for s in x]
  x = [convert_tokens_to_indices(s, word_to_ix) for s in x]

  # Pad x so that all the examples in the batch have the same size
  pad_token_ix = word_to_ix["<pad>"]
  x = [torch.LongTensor(x_i) for x_i in x]
```

```
  x_padded = nn.utils.rnn.pad_sequence(x, batch_first=True, padding_value=pad_token_i

  # Pad y and record the length
  lengths = [len(label) for label in y]
  lenghts = torch.LongTensor(lengths)
  y = [torch.LongTensor(y_i) for y_i in y]
  y_padded = nn.utils.rnn.pad_sequence(y, batch_first=True, padding_value=0)

  return x_padded, y_padded, lenghts
```

Now, we can see the `DataLoader` in action.

```
# Parameters to be passed to the DataLoader
data = list(zip(train_sentences, train_labels))
batch_size = 2
shuffle = True
window_size = 2
collate_fn = partial(custom_collate_fn, window_size=window_size, word_to_ix=word_to_i

# Instantiate the DataLoader
loader = DataLoader(data, batch_size=batch_size, shuffle=shuffle, collate_fn=collate_

# Go through one loop
counter = 0
for batched_x, batched_y, batched_lengths in loader:
  print(f"Iteration {counter}")
  print("Batched Input:")
  print(batched_x)
  print("Batched Labels:")
  print(batched_y)
  print("Batched Lengths:")
  print(batched_lengths)
  print("")
  counter += 1
```

```
Iteration 0
Batched Input:
tensor([[ 0,  0, 10, 13, 11, 17,  0,  0],
        [ 0,  0,  9,  7,  8, 18,  0,  0]])
Batched Labels:
tensor([[0, 0, 0, 1],
        [0, 0, 0, 1]])
Batched Lengths:
tensor([4, 4])

Iteration 1
Batched Input:
tensor([[ 0,  0, 19, 16, 12,  8,  4,  0,  0],
        [ 0,  0, 22,  2,  6, 20, 15,  0,  0]])
Batched Labels:
tensor([[0, 0, 0, 0, 1],
        [0, 0, 0, 0, 1]])
```

```
Batched Lengths:
tensor([5, 5])

Iteration 2
Batched Input:
tensor([[ 0,  0, 19,  5, 14, 21, 12,  3,  0,  0]])
Batched Labels:
tensor([[0, 0, 0, 1, 0, 1]])
Batched Lengths:
tensor([6])
```

The batched input tensors you see above will be passed into our model. On the other hand, we started off saying that our model will be a window classifier. The way our input tensors are currently formatted, we have all the words in a sentence in one datapoint. When we pass this input to our model, it needs to create the windows for each word, make a prediction as to whether the center word is a `LOCATION` or not for each window, put the predictions together and return.

We could avoid this problem if we formatted our data by breaking it into windows beforehand. In this example, we will instead how our model take care of the formatting.

Given that our `window_size` is `N` we want our model to make a prediction on every `2N+1` tokens. That is, if we have an input with `9` tokens, and a `window_size` of `2`, we want our model to return `5` predictions. This makes sense because before we padded it with `2` tokens on each side, our input also had `5` tokens in it!

We can create these windows by using for loops, but there is a faster `PyTorch` alternative, which is the `unfold(dimension, size, step)` method. We can create the windows we need using this method as follows:

```
# Print the original tensor
print(f"Original Tensor: ")
print(batched_x)
print("")

# Create the 2 * 2 + 1 chunks
chunk = batched_x.unfold(1, window_size*2 + 1, 1)
print(f"Windows: ")
print(chunk)
```

```
Original Tensor:
tensor([[ 0,  0, 19,  5, 14, 21, 12,  3,  0,  0]])

Windows:
tensor([[[ 0,  0, 19,  5, 14],
         [ 0, 19,  5, 14, 21],
         [19,  5, 14, 21, 12],
         [ 5, 14, 21, 12,  3],
```

```
        [14, 21, 12,  3,  0],
        [21, 12,  3,  0,  0]]])
```

Model

Now that we have prepared our data, we are ready to build our model. We have learned how to write custom `nn.Module` classes. We will do the same here and put everything we have learned so far together.

```
class WordWindowClassifier(nn.Module):

  def __init__(self, hyperparameters, vocab_size, pad_ix=0):
    super(WordWindowClassifier, self).__init__()

    """ Instance variables """
    self.window_size = hyperparameters["window_size"]
    self.embed_dim = hyperparameters["embed_dim"]
    self.hidden_dim = hyperparameters["hidden_dim"]
    self.freeze_embeddings = hyperparameters["freeze_embeddings"]

    """ Embedding Layer
    Takes in a tensor containing embedding indices, and returns the
    corresponding embeddings. The output is of dim
    (number_of_indices * embedding_dim).

    If freeze_embeddings is True, set the embedding layer parameters to be
    non-trainable. This is useful if we only want the parameters other than the
    embeddings parameters to change.

    """
    self.embeds = nn.Embedding(vocab_size, self.embed_dim, padding_idx=pad_ix)
    if self.freeze_embeddings:
      self.embed_layer.weight.requires_grad = False

    """ Hidden Layer
    """
    full_window_size = 2 * window_size + 1
    self.hidden_layer = nn.Sequential(
      nn.Linear(full_window_size * self.embed_dim, self.hidden_dim),
      nn.Tanh()
    )

    """ Output Layer
    """
    self.output_layer = nn.Linear(self.hidden_dim, 1)

    """ Probabilities
    """
    self.probabilities = nn.Sigmoid()
```

```
    def forward(self, inputs):
      """
      Let B:= batch_size
          L:= window-padded sentence length
          D:= self.embed_dim
          S:= self.window_size
          H:= self.hidden_dim

      inputs: a (B, L) tensor of token indices
      """
      B, L = inputs.size()

      """
      Reshaping.
      Takes in a (B, L) LongTensor
      Outputs a (B, L~, S) LongTensor
      """
      # Fist, get our word windows for each word in our input.
      token_windows = inputs.unfold(1, 2 * self.window_size + 1, 1)
      _, adjusted_length, _ = token_windows.size()

      # Good idea to do internal tensor-size sanity checks, at the least in comments!
      assert token_windows.size() == (B, adjusted_length, 2 * self.window_size + 1)

      """
      Embedding.
      Takes in a torch.LongTensor of size (B, L~, S)
      Outputs a (B, L~, S, D) FloatTensor.
      """
      embedded_windows = self.embeds(token_windows)

      """
      Reshaping.
      Takes in a (B, L~, S, D) FloatTensor.
      Resizes it into a (B, L~, S*D) FloatTensor.
      -1 argument "infers" what the last dimension should be based on leftover axes.
      """
      embedded_windows = embedded_windows.view(B, adjusted_length, -1)

      """
      Layer 1.
      Takes in a (B, L~, S*D) FloatTensor.
      Resizes it into a (B, L~, H) FloatTensor
      """
      layer_1 = self.hidden_layer(embedded_windows)

      """
      Layer 2
      Takes in a (B, L~, H) FloatTensor.
      Resizes it into a (B, L~, 1) FloatTensor.
```

```
    """
    output = self.output_layer(layer_1)

    """
    Softmax.
    Takes in a (B, L~, 1) FloatTensor of unnormalized class scores.
    Outputs a (B, L~, 1) FloatTensor of (log-)normalized class scores.
    """
    output = self.probabilities(output)
    output = output.view(B, -1)

    return output
```

Training

We are now ready to put everything together. Let's start with preparing our data and intializing our model. We can then intialize our optimizer and define our loss function. This time, instead of using one of the predefined loss function as we did before, we will define our own loss function.

```
# Prepare the data
data = list(zip(train_sentences, train_labels))
batch_size = 2
shuffle = True
window_size = 2
collate_fn = partial(custom_collate_fn, window_size=window_size, word_to_ix=word_to_i

# Instantiate a DataLoader
loader = DataLoader(data, batch_size=batch_size, shuffle=shuffle, collate_fn=collate_

# Initialize a model
# It is useful to put all the model hyperparameters in a dictionary
model_hyperparameters = {
    "batch_size": 4,
    "window_size": 2,
    "embed_dim": 25,
    "hidden_dim": 25,
    "freeze_embeddings": False,
}

vocab_size = len(word_to_ix)
model = WordWindowClassifier(model_hyperparameters, vocab_size)

# Define an optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Define a loss function, which computes to binary cross entropy loss
def loss_function(batch_outputs, batch_labels, batch_lengths):
    # Calculate the loss for the whole batch
```

```
    bceloss = nn.BCELoss()
    loss = bceloss(batch_outputs, batch_labels.float())

    # Rescale the loss. Remember that we have used lengths to store the
    # number of words in each training example
    loss = loss / batch_lengths.sum().float()

    return loss
```

Unlike our earlier example, this time instead of passing all of our training data to the model at once in each epoch, we will be utilizing batches. Hence, in each training epoch iteration, we also iterate over the batches.

```
# Function that will be called in every epoch
def train_epoch(loss_function, optimizer, model, loader):

  # Keep track of the total loss for the batch
  total_loss = 0
  for batch_inputs, batch_labels, batch_lengths in loader:
    # Clear the gradients
    optimizer.zero_grad()
    # Run a forward pass
    outputs = model.forward(batch_inputs)
    # Compute the batch loss
    loss = loss_function(outputs, batch_labels, batch_lengths)
    # Calculate the gradients
    loss.backward()
    # Update the parameteres
    optimizer.step()
    total_loss += loss.item()

  return total_loss


# Function containing our main training loop
def train(loss_function, optimizer, model, loader, num_epochs=10000):

  # Iterate through each epoch and call our train_epoch function
  for epoch in range(num_epochs):
    epoch_loss = train_epoch(loss_function, optimizer, model, loader)
    if epoch % 100 == 0: print(epoch_loss)
```

Let's start training!

```
num_epochs = 1000
train(loss_function, optimizer, model, loader, num_epochs=num_epochs)
```

```
0.2576564848423004
0.22127815708518028
0.19427763670682907
0.133215494453907
0.11663811840116978
0.0849589966237545
0.0594865120947361
0.05912754498422146
0.04591026436537504
```

Prediction

Let's see how well our model is at making predictions. We can start by creating our test data.

```
# Create test sentences
test_corpus = ["She comes from Paris"]
test_sentences = [s.lower().split() for s in test_corpus]
test_labels = [[0, 0, 0, 1]]

# Create a test loader
test_data = list(zip(test_sentences, test_labels))
batch_size = 1
shuffle = False
window_size = 2
collate_fn = partial(custom_collate_fn, window_size=2, word_to_ix=word_to_ix)
test_loader = torch.utils.data.DataLoader(test_data,
                                          batch_size=1,
                                          shuffle=False,
                                          collate_fn=collate_fn)
```

Let's loop over our test examples to see how well we are doing.

```
for test_instance, labels, _ in test_loader:
  outputs = model.forward(test_instance)
  print(labels)
  print(outputs)
```

```
tensor([[0, 0, 0, 1]])
tensor([[0.1245, 0.0452, 0.0449, 0.9717]], grad_fn=<ViewBackward0>)
```