Information Retrieval

Deep Learning with Pytorch

PyTorch

- PyTorch is a powerful, yet easy-to-use mathematical library with automatic differentiation for Python, mainly used in Deep Learning.
  - PyTorch has a Python front-end component that offers a high-level programmable interface to users, while most of the code runs on a backend in C/C++ (CUDA/MPS/ROCm) for maximum efficiency.

PyTorch - Tensors

- The library itself contains multiple abstraction layers, where the most low-level data representation is called a **tensor**, on which multiple mathematical operations can be performed.
  - Tensors are a specialized data structure that are very similar to arrays and matrices.

```
x = [0,1,2,3,4]
x_tensor = torch.tensor(x)
x_tensor, x_tensor.shape, x_tensor.dtype, x_tensor.device
```

```
(tensor([0, 1, 2, 3, 4]), torch.Size([5]), torch.int64, device(type='cpu'))
```

PyTorch - Tensors

- Tensors can be allocated or sent to any compute device available in your system.

```
x_cuda_tensor = x_tensor.to("cuda")
x_cuda_tensor, x_cuda_tensor.shape, x_cuda_tensor.dtype, x_cuda_tensor.device
```

```
(tensor([0, 1, 2, 3, 4], device='cuda:0'),
 torch.Size([5]),
 torch.int64,
 device(type='cuda', index=0))
```

PyTorch - Tensors

❖ Tensors can be allocated or sent to any compute device available in your system.

```
big_matrix = torch.rand((10000,1000))
big_matrix_gpu = big_matrix.to("cuda")
```

On CPU

```
%time big_matrix @ big_matrix.T
```

```
CPU times: user 2.44 s, sys: 178 ms, total: 2.62 s
Wall time: 2.72 s
```

On GPU

```
%time big_matrix_gpu @ big_matrix_gpu.T
```

```
CPU times: user 975 µs, sys: 993 µs, total: 1.97 ms
Wall time: 2.06 ms
```

PyTorch – Neural Networks

- Neural networks are composed of a set of layers/modules that perform operations on data.
  - PyTorch torch.nn namespace provides all the building blocks you need to build your own neural network.
- Every layer/module in PyTorch subclasses the nn.Module. A neural network is a module itself that consists of other modules (layers).

torch.nn

These are the basic building blocks for graphs:

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization
  - Aliases

https://pytorch.org/docs/stable/nn.html#module-torch.nn

PyTorch – Neural Networks

- With special interest for IR:
  - Embedding Layer (e.g. torch.nn.Embedding)
  - Convolution Layers (e.g. torch.nn.Conv2d)
  - Recurrent Layers (e.g. torch.nn.LSTM)
  - Linear Layers (e.g. torch.nn.Linear)
  - Non-linear Activations (e.g. torch.nn.ReLU)

torch.nn

These are the basic building blocks for graphs:

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization
  - Aliases

https://pytorch.org/docs/stable/nn.html#module-torch.nn

PyTorch – Neural Networks

```
class AbstractModel(torch.nn.Module):

    def __init__(self):
        super().__init__()
        # Instantiation of layers and parameters

    def forward(self, x):
        # operations that use the previous created layers over the inputs
        logits = x
        return logits
```

PyTorch – Neural Networks

```
model = AbstractModel()
model(torch.tensor([1,2,3]))
```

```
tensor([2, 4, 6])
```

```
model_on_gpu = model.to("cuda")
model_on_gpu(torch.tensor([1,2,3]).to("cuda"))
```

PyTorch – Dataset and DataLoaders

- PyTorch also offers a high-level API for handling data processing and for feeding data to your model.
  - torch.utils.data.Dataset
  - torch.utils.data.DataLoader
- Dataset class acts as an abstraction that holds your data and knows how to fetch it.

```python
class SimpleDataset(torch.utils.data.Dataset):
    def __init__(self, num_samples=1000):
        super().__init__()
        # load your data here

        # dummy data
        self.features = [torch.rand(1, random.randint(3, 5)) for _ in range(num_samples)]
        self.labels = [float(torch.mean(f) > 0.5) for f in self.features]

    def __len__(self):
        # return the number of samples in the dataset
        return len(self.features)

    def __getitem__(self, idx):
        # get the sample corresponding to index "idx"
        return self.features[idx], self.labels[idx]

ds = SimpleDataset()
print(ds[10])
print(ds[31])
```

```
(tensor([[0.5195, 0.7827, 0.2147, 0.2094, 0.7416]]), 0.0)
(tensor([[0.6587, 0.7761, 0.9588, 0.7888]]), 1.0)
```

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#creating-a-custom-dataset-for-your-files

PyTorch – Dataset and DataLoaders

- The DataLoader offers an abstraction that effectively selects which sample will be fed to the model.
  - Selects a batch of data
  - Converts that data to tensors
  - Repeat

```
def collate_fn(batch):
    features, labels = zip(*batch)
    max_len = max(f.size(1) for f in features)
    padded = [torch.cat([f, torch.zeros(1, max_len - f.size(1))], dim=1) for f in features]
    return torch.cat(padded), torch.tensor(labels)

# Create dataloader
dataloader = torch.utils.data.DataLoader(
    ds,
    batch_size=4,
    shuffle=True,
    collate_fn=collate_fn
)

next(iter(dataloader))
```

```
(tensor([[0.7736, 0.3263, 0.6309, 0.0000],
         [0.0070, 0.0373, 0.2937, 0.0000],
         [0.1397, 0.7423, 0.2542, 0.0000],
         [0.4568, 0.6127, 0.1965, 0.0904]]),
 tensor([1., 0., 0., 0.]))
```


https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#creating-a-custom-dataset-for-your-files

Goal for Today!!!

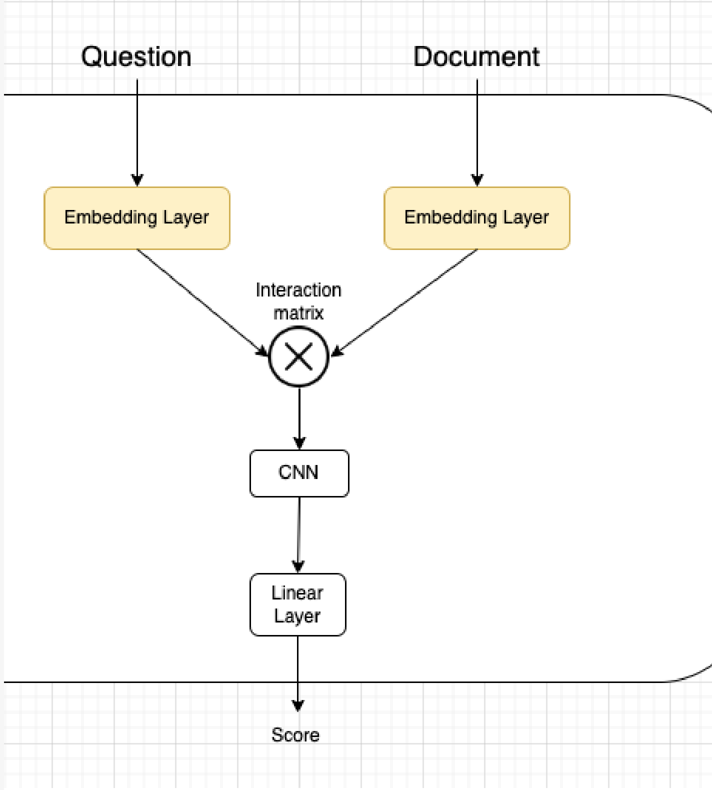
- Implement the following interaction-based Information Retrieval Model

```python
class CNNInteractionBasedModel(torch.nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        self.vocab_size = vocab_size
        # Instantiation of layers and parameters
        # - embedding layer
        # - Convolution and pooling layers
        # - Activation functions
        # - Linear layer for scoring

    def forward(self, query, document):
        print(query, document)
        # flow of the computation
        # - convert the query and document ids to document
        #   vectors with the embedding layer
        # - create an interaction matrix
        # - apply convolution and max pooling over the matrix
        # - apply linear layer to the resulting feature maps
        # - return the final logits
        # - optinally convert the logits to probabilities with sigmoid function

        return logits
```



- Start code: https://colab.research.google.com/drive/1HpQbXQ5be3MQCOtYQjntCjDeioGXwply?usp=sharing