

Classifying Images using an RNN

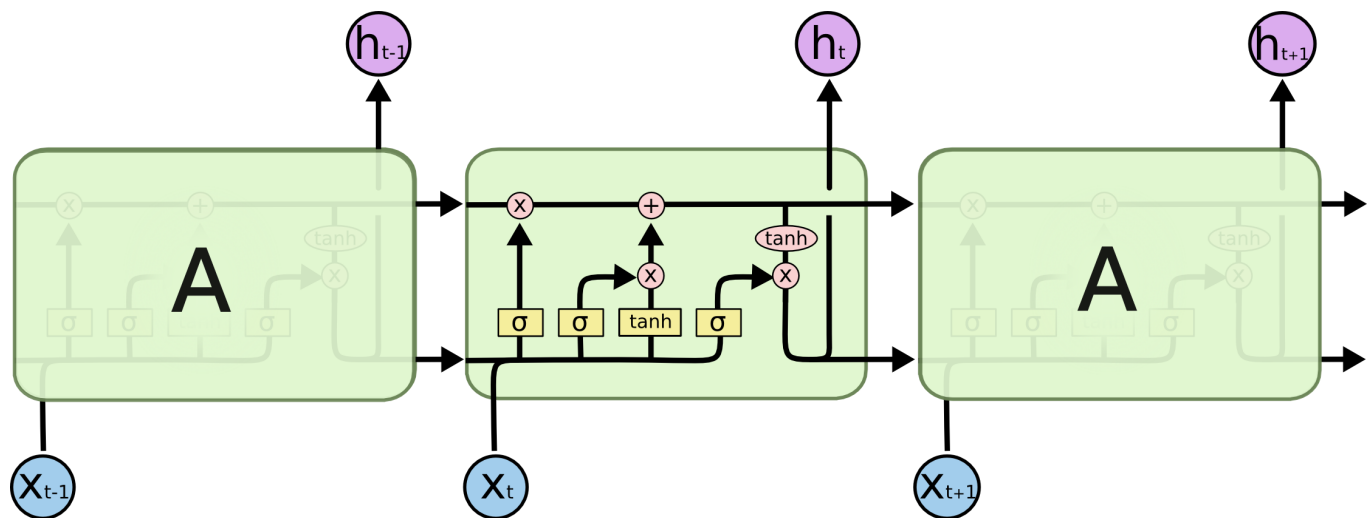
In this demo we will use recurrent neural nets (RNNs) for a slightly non-standard application: image classification. To be clear, RNN-type architectures have been typically used for applications involving sequences (such as text, audio, time series, etc.)

For text applications, one has to somehow transform a sequence of characters into a sequence of numerical (vector) *encodings*. We will get into encodings in a later demo.

Image data, on the other hand, is already in vector form. Let us imagine a setting where the pixels of an image are presented to the classifier in sequential form. Think of a problem where an image is being "raster-scanned", and where the classifier has to continuously updates its beliefs as new pixels come in. We will use a recurrent neural network to solve this problem.

We saw in class that standard recurrent neural networks have trouble keeping information through multiple inputs. So today, we will be using a Long Short Term Model (LSTM).

[This](#) blog does an excellent job explaining how the LSTM functions. Please see their diagram below for a high level picture.



The idea is a single cell state is passed through on the top level. The first operation in each block determines what previous cell state to keep around, the second operation determines what new information to add, while the final operation decides how to combine the hidden state, input, and updated cell state for the output.

In this way, the LSTM has the opportunity to keep track of inputs that appeared long ago or overwrite them, depending on what works in the training data.

▼ Data loading and preparation

We will use the familiar Fashion-MNIST dataset. The following code (until building the data-loader) should be standard.

```
import torch

from torchvision import datasets
from torchvision.transforms import ToTensor
train_data = datasets.FashionMNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.FashionMNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)
```

```

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to data/FashionMNIST/raw/t1
100% 26421880/26421880 [00:01<00:00, 24632402.76it/s]
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw/t1
100% 29515/29515 [00:00<00:00, 267144.76it/s]
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw/t1
100% 4422102/4422102 [00:00<00:00, 8481968.55it/s]
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to data/FashionMNIST/raw/t1

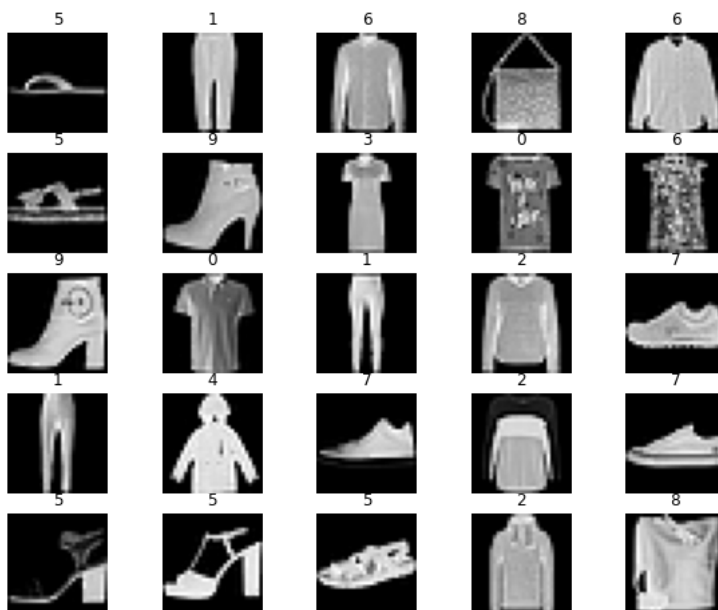
```

```
import matplotlib.pyplot as plt
```

```

figure = plt.figure(figsize=(10, 8))
cols, rows = 5, 5
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(train_data), size=(1,)).item()
    img, label = train_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()

```



```

from torch.utils.data import DataLoader
dataloaders = {
    'train' : torch.utils.data.DataLoader(train_data,
                                          batch_size=100,
                                          shuffle=True,
                                          num_workers=1),

    'test' : torch.utils.data.DataLoader(test_data,
                                          batch_size=100,
                                          shuffle=True,
                                          num_workers=1),
}

```

▼ Setting up the RNN model

Recall that FashionMNIST images are of size 28×28 . We will pretend that each row of any input image is a token (of length 28), which is presented sequentially to the RNN. Therefore there are 28 such tokens for each sequence (image).

We will assume [LSTM](#) recurrent units with 2 layers and hidden size 128, followed by an output linear layer that performs the classification.

```
from torch import nn
import torch.nn.functional as F

sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10 #number of categories of clothing
batch_size = 100 #100 pics
num_epochs = 4
learning_rate = 0.01
```

The key is to set up the RNN correctly. Note that the length of the sequence is irrelevant to the model: it can process sequences of any length.

```
class RNN(nn.Module):
    #change this code
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.U = nn.Linear(input_size, hidden_size)
        self.W = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Set initial hidden and cell states
        h = torch.zeros(hidden_size)

        for i in range(x.shape[1]):
            h = F.sigmoid(self.U(x[:, i, :]) + self.W(h))
            out = self.V(h)
            return F.softmax(out, dim = 1)
```

```
model = RNN(input_size, hidden_size, num_layers, num_classes)
print(model)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-c32900ab5c3a> in <module>
----> 1 class RNN(nn.Module):
      2     #change this code
      3     def __init__(self, input_size, hidden_size, num_layers, num_classes):
      4         super(RNN, self).__init__()
      5         self.hidden_size = hidden_size
```

```
NameError: name 'nn' is not defined
```

SEARCH STACK OVERFLOW

▼ Training the model

Everything looks good! Let us now train the model using the (familiar) cross-entropy loss and the Adam optimizer. We will use backpropagation-through-time (BPTT), which Pytorch automatically handles under the hood while calling `optimizer_backward()`.

```
loss_func = nn.CrossEntropyLoss() #softmax called within
from torch import optim
optimizer = optim.Adam(model.parameters(), lr = learning_rate)
```

```
def train(num_epochs, model, loaders):

    # Train the model
    total_step = len(dataloaders['train'])
```

```
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(dataloaders['train']):
```

```

for i, (images, labels) in enumerate(dataloaders['train']):

    images = images.reshape(-1, sequence_length, input_size)

    # Forward pass
    outputs = model(images)
    loss = loss_func(outputs, labels)
    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (i+1) % 100 == 0:
        print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.2f}'
              .format(epoch + 1, num_epochs, i + 1, total_step, loss.item()))

```

```
train(num_epochs, model, dataloaders)
```

```

↳ /usr/local/lib/python3.8/dist-packages/torch/nn/functional.py:1967: UserWarning: nn.functional.sigmoid is deprecated. Use torch.nn.functional.sigmoid instead.
warnings.warn("nn.functional.sigmoid is deprecated. Use torch.nn.functional.sigmoid instead.")
Epoch [1/4], Step [100/600], Loss: 2.07
Epoch [1/4], Step [200/600], Loss: 1.91
Epoch [1/4], Step [300/600], Loss: 2.11
Epoch [1/4], Step [400/600], Loss: 2.06
Epoch [1/4], Step [500/600], Loss: 2.02
Epoch [1/4], Step [600/600], Loss: 1.99
Epoch [2/4], Step [100/600], Loss: 2.05
Epoch [2/4], Step [200/600], Loss: 2.04
Epoch [2/4], Step [300/600], Loss: 2.07
Epoch [2/4], Step [400/600], Loss: 2.00
Epoch [2/4], Step [500/600], Loss: 2.11
Epoch [2/4], Step [600/600], Loss: 1.98
Epoch [3/4], Step [100/600], Loss: 1.84
Epoch [3/4], Step [200/600], Loss: 1.95
Epoch [3/4], Step [300/600], Loss: 1.87
Epoch [3/4], Step [400/600], Loss: 1.91
Epoch [3/4], Step [500/600], Loss: 1.90
Epoch [3/4], Step [600/600], Loss: 2.02
Epoch [4/4], Step [100/600], Loss: 1.96
Epoch [4/4], Step [200/600], Loss: 1.80
Epoch [4/4], Step [300/600], Loss: 1.82
Epoch [4/4], Step [400/600], Loss: 1.92
Epoch [4/4], Step [500/600], Loss: 1.78
Epoch [4/4], Step [600/600], Loss: 1.84

```

▼ Testing

The loss looks decent enough! Let's test the model now.

```

model.eval() # dropout to false, use all inputs and weights
with torch.no_grad(): #does not keep track of gradient as it goes
    correct = 0
    total = 0
    for images, labels in dataloaders['test']:
        images = images.reshape(-1, sequence_length, input_size)
        outputs = model(images) # or images.squeeze() and no line above
        _, predicted = torch.max(outputs.data, 1)
        total = total + labels.size(0)
        correct = correct + (predicted == labels).sum().item()
print('Test accuracy: {} %'.format(100 * correct / total))

```

```
Test accuracy: 60.14 %
```

```

sample = next(iter(dataloaders['test']))
imgs, lbls = sample

```

```

test_output = model(imgs[:10].view(-1, 28, 28))
predicted = torch.max(test_output, 1)[1].data.numpy().squeeze()
labels = lbls[:10].numpy()
print(f"Predicted label: {predicted}")
print(f"Actual label: {labels}")

```

```

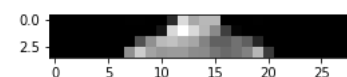
Predicted label: [7 0 3 7 2 8 7 1 3 7]
Actual label: [7 3 3 7 2 5 7 1 3 8]

```

Not too bad! We get ~88% accuracy on FashionMNIST where the pixels are presented sequentially, which is comparable to what we got using regular dense architectures.

An advantage of RNN-based classification for images is that since RNNs work for variable-length inputs, it allows for declaring a class label even if the input is partially available (for example, if only the top one-third of the pixels were rasterized). Let's see how this would work.

```
idx = 32
img = imgs[idx]
num_rows = 4
img_crop = img[:, :num_rows, :]
test_output_on_crop = model(img_crop.view(-1, num_rows, 28))
predicted = torch.max(test_output_on_crop, 1)[1].data.numpy().squeeze()
labels = lbls[idx].numpy()
figure = plt.figure(figsize=(4, 4))
plt.imshow(img_crop.squeeze(), cmap='gray')
plt.show()
print(f"Predicted label: {predicted}")
print(f"Actual label: {labels}")
```



Predicted label: 0
Actual label: 3