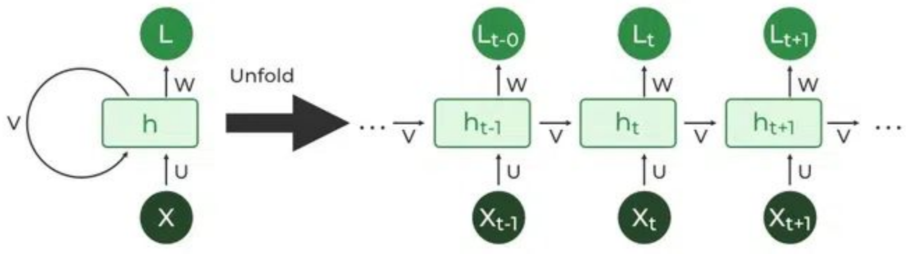
Recurrent Neural Network

Alex Evans
Mandy Zheng

Overview

- RNNs are a class of neural networks
- Allow previous outputs to be used as inputs while having hidden states
- Allows processing of sequential data into sequential outputs



Pros and Cons

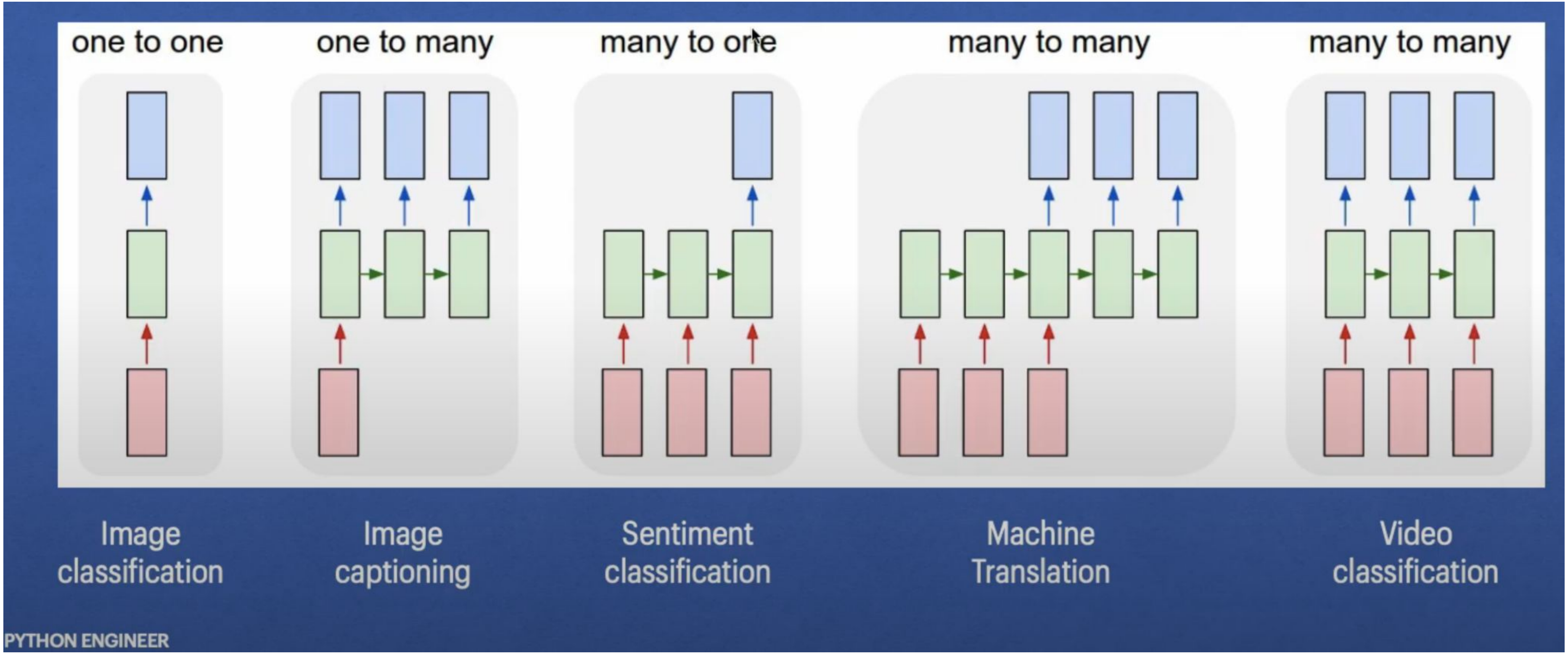
Pros

- Possibility of processing input of any length
- Model size not increasing with size of input
- Computation takes into account historical information
- Weights are shared across time

Cons

- Computation is slow
- Difficulty of accessing information from a long time ago
- Cannot consider any future input for the current state

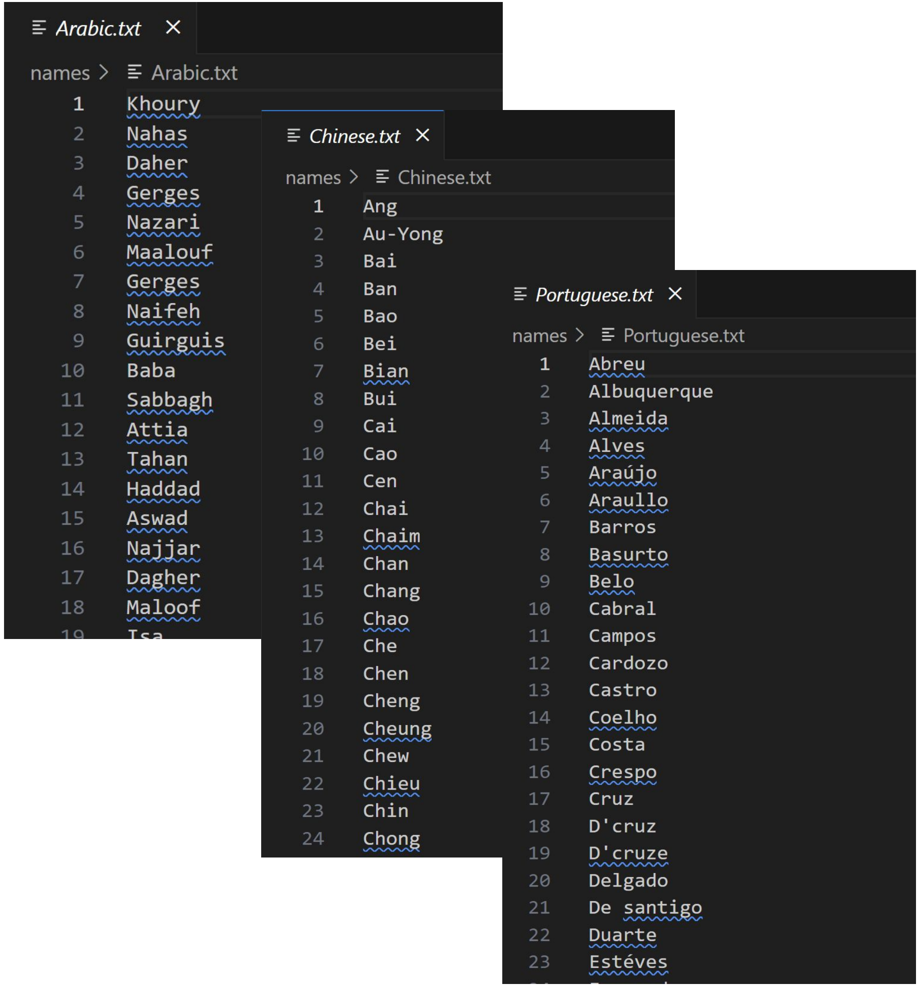
Sequential Data Applications



PROBLEM

Name Classification

The data files contain various names native to a certain country. We will train the RNN to process each sequential data input (name) to an sequential output (country of origin).



METHOD

- Input layer: receives the sequential data (names)
- Output layer: produces the category (country of origin)
- The input is combined with the hidden layer to create an input to output(i2o) and an input to hidden(i2h).
    - i2h: used for the next combination of input + hidden iteration
    - I2o: classifies to an output

```python
class RNN(nn.Module):
    # implementing RNN from scratch rather than using nn.RNN
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input_tensor, hidden_tensor):
        combined = torch.cat((input_tensor, hidden_tensor), 1)

        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)
```

```python
criterion = nn.NLLLoss()
learning_rate = 0.005
optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate)

def train(line_tensor, category_tensor):
    hidden = rnn.init_hidden()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    return output, loss.item()
```

RESULTS

Optimization:

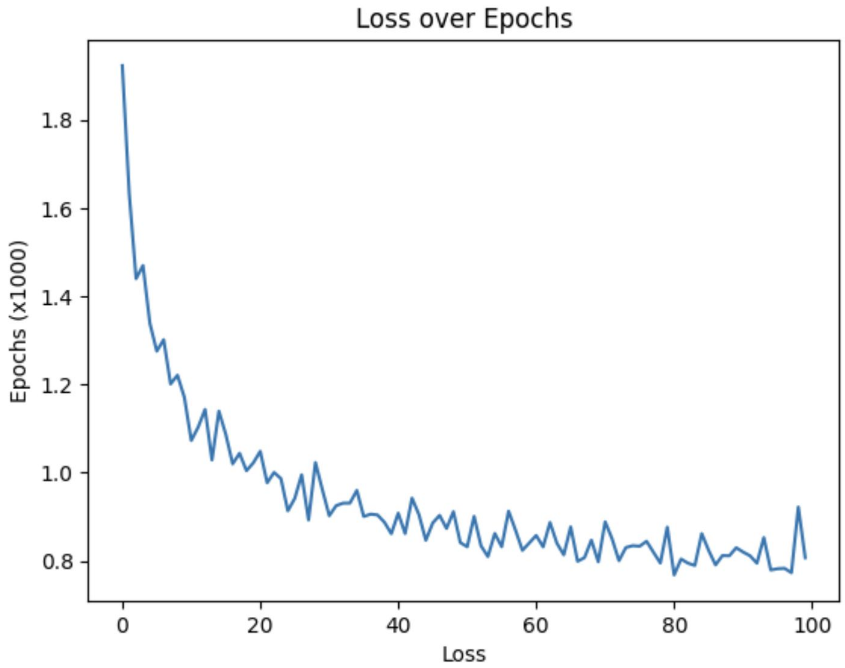
- Used CrossEntropyLoss to eliminate the need for NLLoss with LogSoftmax (already within CrossEntropyLoss)
- Used Adam optimizer instead of Stochastic Gradient Descent (SGD)
- Reduced learning rate to 0.0005
- Increased hidden_size to 256 (2x)

- Implement class weights

```
Test Accuracy: 24.01%
```

```
Test Accuracy: 62.23% | Train Accuracy: 63.21%
```

```
Test Accuracy: 74.96% | Train Accuracy: 77.40%
```




```
Train Accuracy: 62.39%
5000 5.0 2.6898 Kyubei / Russian WRONG (Japanese)
Train Accuracy: 68.35%
10000 10.0 0.3781 Teterev /
Train Accuracy: 67.63%
15000 15.0 2.0180 Stupka /
```

```
Train Accuracy: 74.92%
60000 60.0 0.0007 Zenkov / Russian CORRECT
Train Accuracy: 74.85%
...
Train Accuracy: 72.85%
95000 95.0 0.3383 Jakon / Russian CORRECT
Train Accuracy: 77.40%
100000 100.0 0.5052 Ghannam / Arabic CORRECT
```