

Recurrent Neural Network

Alex Evans

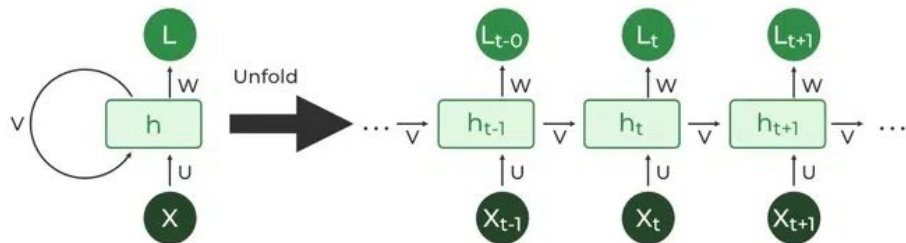
Mandy Zheng

A dark blue diagonal gradient bar that starts from the bottom left and extends towards the top right, covering the lower half of the slide.

Overview

Pros and Cons

- 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



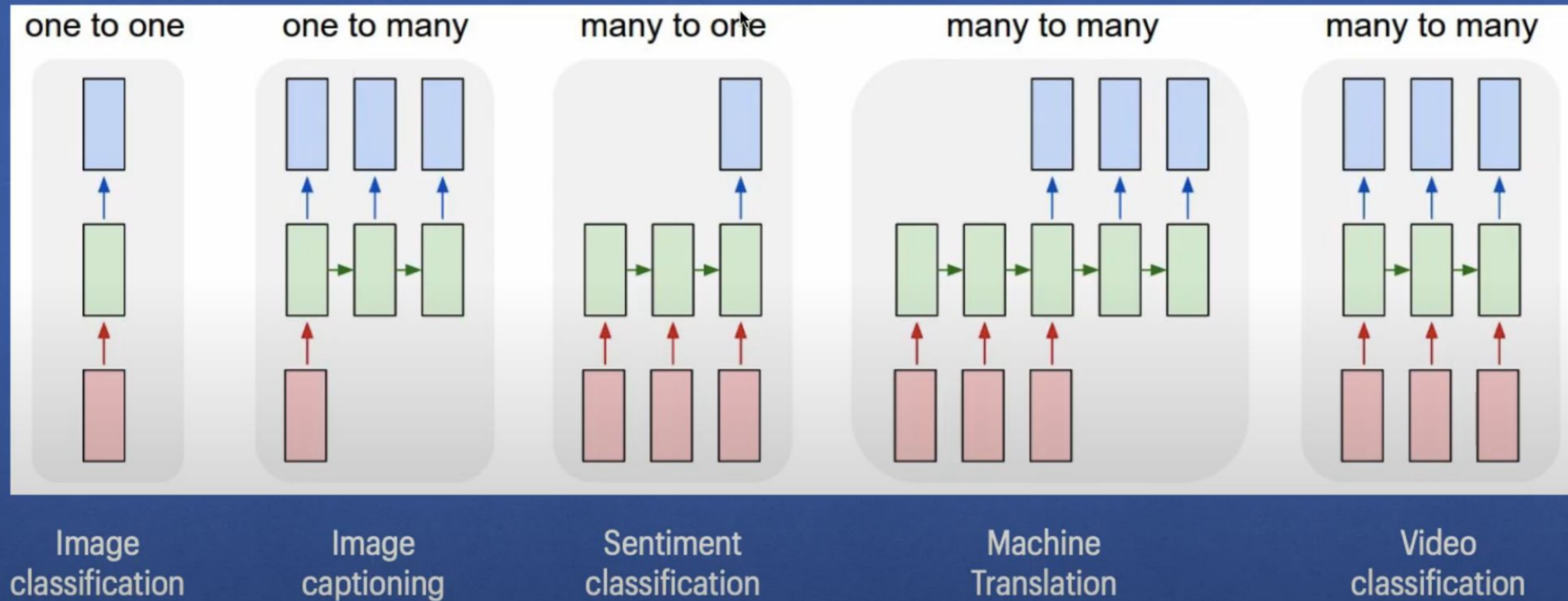
Pros

- 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).

```
Arabic.txt X
names > Arabic.txt
1 Khoury
2 Nahas
3 Daher
4 Gerges
5 Nazari
6 Maalouf
7 Gerges
8 Naifeh
9 Guirguis
10 Baba
11 Sabbagh
12 Attia
13 Tahan
14 Haddad
15 Aswad
16 Najjar
17 Dagher
18 Maloof
19 Isa

Chinese.txt X
names > Chinese.txt
1 Ang
2 Au-Yong
3 Bai
4 Ban
5 Bao
6 Bei
7 Bian
8 Bui
9 Cai
10 Cao
11 Cen
12 Chai
13 Chaim
14 Chan
15 Chang
16 Chao
17 Che
18 Chen
19 Cheng
20 Cheung
21 Chew
22 Chieu
23 Chin
24 Chong

Portuguese.txt X
names > Portuguese.txt
1 Abreu
2 Albuquerque
3 Almeida
4 Alves
5 Araújo
6 Araullo
7 Barros
8 Basurto
9 Belo
10 Cabral
11 Campos
12 Cardozo
13 Castro
14 Coelho
15 Costa
16 Crespo
17 Cruz
18 D'cruz
19 D'cruze
20 Delgado
21 De santigo
22 Duarte
23 Estéves
```

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

```
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)
```

```
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

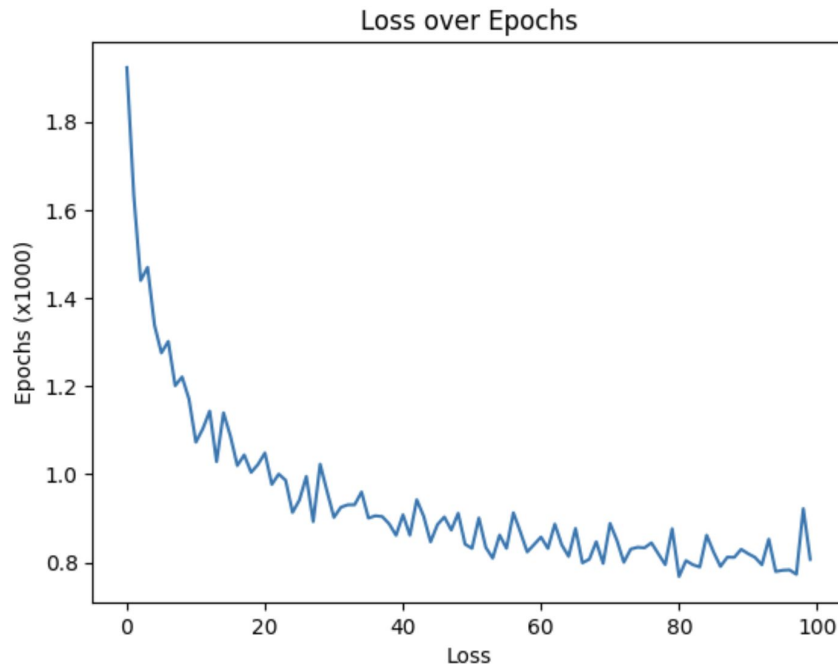
Optimization:

- 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 / 60000 60.0 0.0007 Zenkov / Russian CORRECT

Train Accuracy: 67.63%

15000 15.0 2.0180 Stupka / ...

Train Accuracy: 74.92%

Train Accuracy: 72.85%

95000 95.0 0.3383 Jakon / Russian CORRECT

Train Accuracy: 77.40%

100000 100.0 0.5052 Ghannam / Arabic CORRECT