Introduction to Recurrent Neural Network

Bokan Bao
Professor Dasgupta
CSE 254

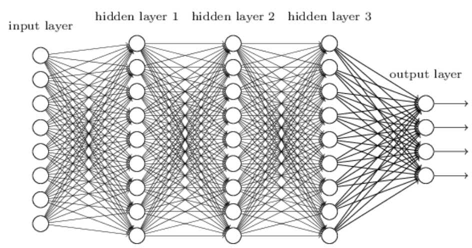
Outlines

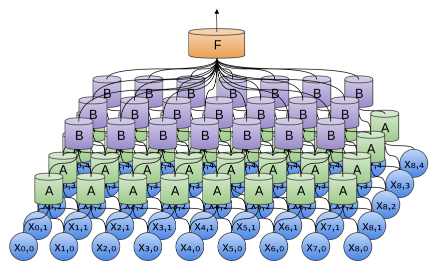
- RNN introduction
  - Limitation of Vanilla Neural Network
  - Structure of the RNN
  - PyTorch implementation

- Character-Level prediction
  - Unsmoothed Maximum Likelihood Character Level Language Model
  - Three-layer RNN

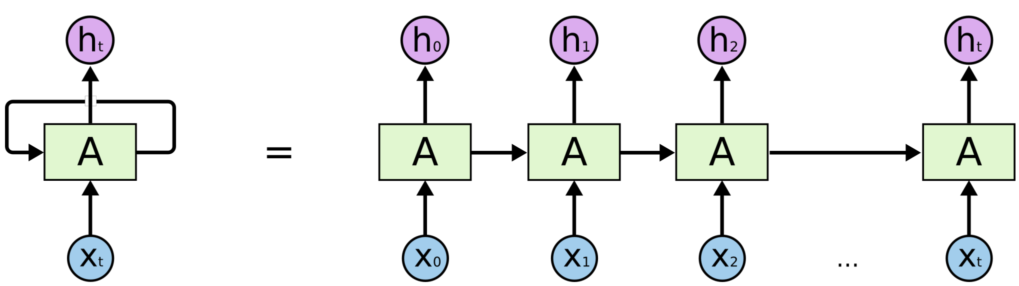
Vanilla Neural Network

They accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes).



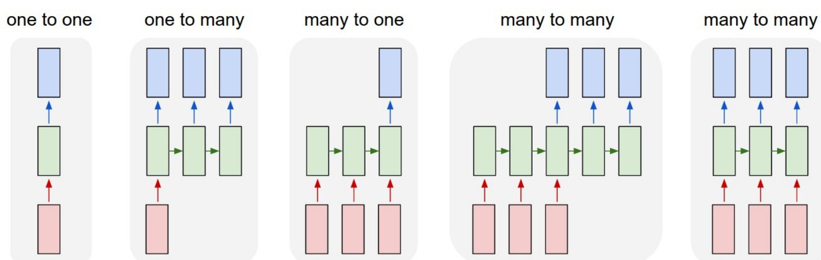


Recurrent Neural Network

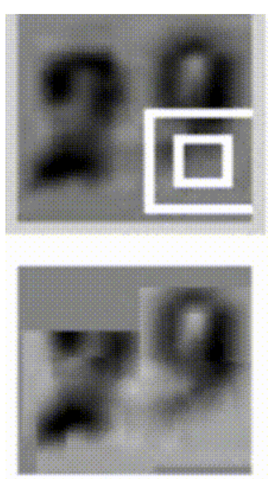


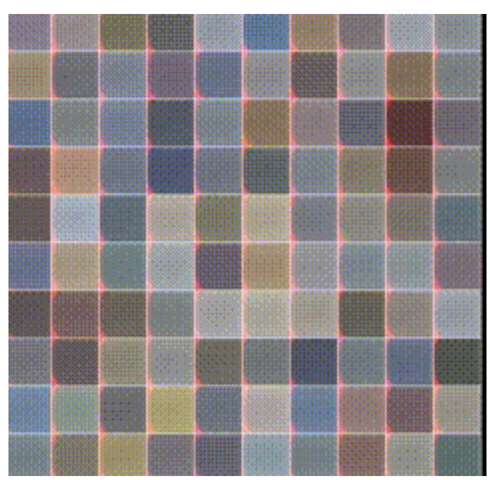
Recurrent Neural Network

- Flexible, a sequence of the data
- Music, text, motion capture
- The predictive distribution depends on the previous inputs.



A sequence of many things





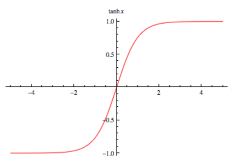
Simple RNN model

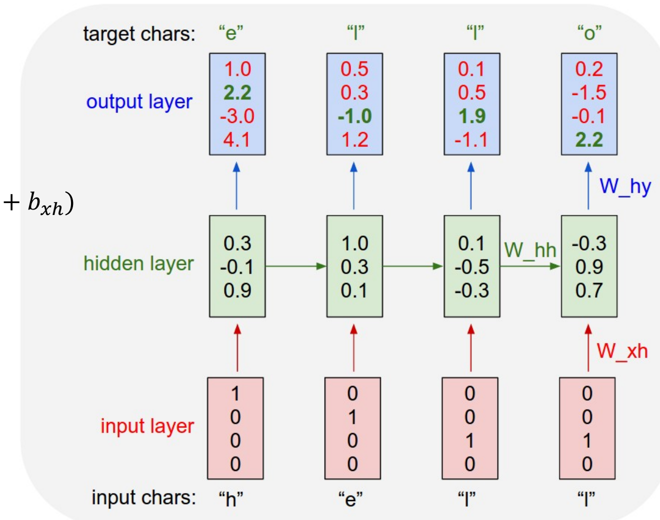
$$x_t \rightarrow W_{xh}x_t + b_{xh}$$

$$h_{t-1} \rightarrow h_t$$

$$h_t = \tanh(W_{hh}h_{t-1} + b_{hh} + W_{xh}x_t + b_{xh})$$

$$y_t = W_{hy}h_t + b_{hy}$$



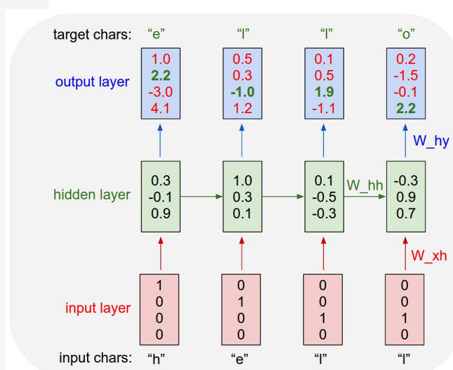


Pytorch implementation

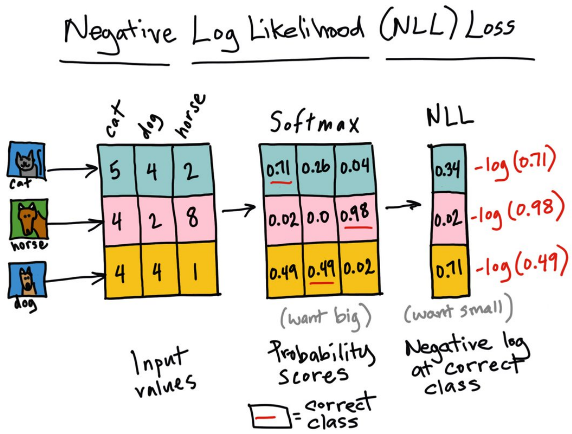
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.W_xh = nn.Linear(input_size, hidden_size)
        self.W_hh = nn.Linear(hidden_size, hidden_size)
        self.W_hy = nn.Linear(hidden_size, output_size)
        self.act = nn.Tanh()
        self.dropout = nn.Dropout(0.1)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, _input, hidden):
        hidden = self.W_xh(_input).add(self.W_hh(hidden))
        hidden = self.act(hidden)
        output = self.W_hy(hidden)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

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



Loss function: Negative Log Likelihood Loss



Correct classes are known because we are training

Predictor confidence of **horse** is high.
Lower unhappiness.

Predictor confidence of **dog** is low.
Higher unhappiness.

Optimization function: Adam

- Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled "Adam: A Method for Stochastic Optimization". I will quote liberally from their paper in this post, unless stated otherwise.
- The authors describe Adam as combining the advantages of two other extensions of stochastic gradient descent. Specifically:
  - **Adaptive Gradient Algorithm** (AdaGrad)
  - **Root Mean Square Propagation** (RMSProp)

Training

```
loss_fun = nn.NLLLoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=0.01)

while epoch < 10:
    if p + seq_length +1 >= len(data):
        epoch +=1
        p = 0
    # Generate input and target
    input_line_tensor, target_line_tensor = createTrainingExample(data[p:p+seq_length+1]
    target_line_tensor.unsqueeze_(-1)
    # Initiation
    hidden = rnn.initHidden()
    rnn.zero_grad()
    loss = 0
    # Training
    for i in range(input_line_tensor.size(0)):
        output, hidden = rnn(input_line_tensor[i], hidden)

        l = loss_fun(output, target_line_tensor[i])
        loss += l
    # Back propagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    p += seq_length # move data pointer
```

Character-Level Predict Model

- Unsmoothed Maximum Likelihood Model
    - c is character, h is a n letters history
    - P(c|h) stands for how likely is it to see c after we've seen h.
- Training data: Shakespeare
    - demo: A 4-letter model
    - Deterministic model
    - High dimensions

```
def train_char_lm(fname, order=4):
    data = file(fname).read()
    lm = defaultdict(Counter)
    pad = "~" * order
    data = pad + data
    for i in xrange(len(data)-order):
        history, char = data[i:i+order], data[i+order]
        lm[history][char]+=1
    def normalize(counter):
        s = float(sum(counter.values()))
        return [(c,cnt/s) for c,cnt in counter.iteritems()]
    outlm = {hist:normalize(chars) for hist, chars in lm.iteri
    return outlm
```

```
lm['ello']
```

```
[('!', 0.0068143100511073255),
 (' ', 0.013628620102214651),
 ("'", 0.017035775127768313),
 (',', 0.027257240204429302),
 ('.', 0.0068143100511073255),
 ('r', 0.059625212947189095),
 ('u', 0.03747870528109029),
 ('w', 0.817717206132879),
 ('n', 0.0017035775127768314),
 (':', 0.0051107325383304940),
 ('?', 0.0068143100511073255)]
```

Result from 10-letter model

```
First Citizen:
Nay, then, that was hers,
It speaks against your other service:
But since the
youth of the circumstance be spoken:
Your uncle and one Baptista's daughter.

SEBASTIAN:
Do I stand till the break off.

BIRON:
Hide thy head.

VENTIDIUS:
He purposeth to Athens: whither, with the vow
I made to handle you.

FALSTAFF:
My good knave.

MALVOLIO:
Sad, lady! I could be forgiven you, you're welcome. Give ear, sir, my doublet and hose and leave this present deat

Second Gentleman:
Who may that she confess it is my lord enraged and forestalled ere we come to be a man. Drown thyself?
```

Three layers RNN model

This is a 3-layer RNN with 512 hidden nodes on each layer.

We will need
1. 68 x 512
2. 512 x 512
3. 512 x 512
4. 512 x 512
5. 512 x 68

856,064 parameters in total
N_para x 4 bytes = 3.42 MB
Backward pass requires 3 times of the memory
3.42*3 ~ 10.26 MB

```python
class RNN3(nn.Module):
    def __init__(self, input_size=68, hidden_size=512, output_size=68, n_layers=3):
        super().__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.rnn = nn.RNN(input_size, hidden_size, n_layers, dropout=0.1)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=2)

    def forward(self, input):
        """
        input: a batch of one hot charactors [25, 1, 68]"""
        hidden = self.initHidden(input.size(1))
        return_ = []
        for i in range(input.size(0)):
            _input = input[i:i+1]
            output, hidden = self.rnn(_input, hidden)
            output = self.fc(output)
            output = self.softmax(output)
            return_.append(output)
        return return_

    def prediction(self, input, hidden):
        """
        input: a batch of one hot charactors [1, 1, 68]"""
        output, hidden = self.rnn(input, hidden)
        output = self.fc(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self, batch_size):
        return torch.zeros(self.n_layers, batch_size, self.hidden_size).cuda()
```

Three layers RNN model

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

For xml format (multi-layer LSTM)

```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">#REDIRECT [[Christianity]]</text>
  </revision>
</page>
```

Even Latex (multi-layer LSTM)

```
\begin{proof}
We may assume that $\mathcal{I}$ is an abelian sheaf on $\mathcal{C}$.
\item Given a morphism $\Delta : \mathcal{F} \to \mathcal{I}$
is an injective and let $\mathfrak q$ be an abelian sheaf on $X$.
Let $\mathcal{F}$ be a fibered complex. Let $\mathcal{F}$ be a category.
\begin{enumerate}
\item \hyperref[setain-construction-phantom]{Lemma}
\label{lemma-characterize-quasi-finite}
Let $\mathcal{F}$ be an abelian quasi-coherent sheaf on $\mathcal{C}$.
Let $\mathcal{F}$ be a coherent $\mathcal{O}_X$-module. Then
$\mathcal{F}$ is an abelian catenary over $\mathcal{C}$.
\item The following are equivalent
\begin{enumerate}
\item $\mathcal{F}$ is an $\mathcal{O}_X$-module.
\end{lemma}
```

Thanks

- References

  - https://pytorch.org/tutorials/
  - https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
  - https://nbviewer.jupyter.org/gist/yoavg/d76121dfde2618422139
  - https://karpathy.github.io/2015/05/21/rnn-effectiveness/
  - https://cs.stanford.edu/people/karpathy/char-rnn/shakespear.txt
  - https://gist.github.com/karpathy/d4dee566867f8291f086
  - https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/
  - https://www.youtube.com/watch?v=MKA6v99uYKY&t=198s

- Code on github
  - https://github.com/bobaoai/Sample_RNN.git