Recurrent Neural Networks for Language Modeling

CSE354 - Spring 2021
Natural Language Processing
Tasks

- **Language Modeling:**
  Generate next word, sentence
  ≈ capture hidden representation of sentences.

- **Word, Document Classification**
  (named entity tagging; sentiment analysis using sequence, etc...)
Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$:

probability of a next word given history

$P(fork \mid He\ ate\ the\ cake\ with\ the) = ?$
Language Modeling

Task: Estimate \( P(w_n | w_1, w_2, ..., w_{n-1}) \)
: probability of a next word given history

\[ P(\text{fork} | \text{He ate the cake with the}) = ? \]
Building a model (or system/API) that can answer the following:

- a sequence of natural language
- What is the next word in the sequence?
- The horse which was raced past the barn [tripped].
Language Modeling

Building a model (or system/API) that can answer the following:

- a sequence of natural language
- What is the next word in the sequence?

To fully capture natural language, models get very complex!

The horse which was raced past the barn [tripped].

Training Corpus

training (fit, learn)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

**Figure 9.2** Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

Activation Function

\[ h_t = g(x_t W) \]

\[ y_t = f(h_t W) \]

"hidden layer"

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Common Activation Functions

\[ z = h^{(t)}W \]

Logistic: \( \sigma(z) = \frac{1}{1 + e^{-z}} \)

Hyperbolic tangent: \( \tanh(z) = 2\sigma(2z) - 1 = \frac{e^{2z} - 1}{e^{2z} + 1} \)

Rectified linear unit (ReLU): \( ReLU(z) = \max(0, z) \)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

**Activation Function**

\[ y(t) = f(b(t)W) \]

\[ b(t) = g(x(t)W) \]

**Figure 9.2** Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Neural Networks: Graphs of Operations
(excluding the optimization nodes)

Activation Function

\[ y(t) = f(h(t)W) \]

Activation Function

\[ h(t) = g(x(t)W) \]

Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep.

(Jurafsky, 2019)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

Figure 9.2  Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

**Figure 9.2** Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

Activation Function

\[ h_t = g(h_{t-1}U + x_tV) \]

\[ y_t = f(h_tW) \]

"hidden layer"

Figure 9.2  Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous timestep. (Jurafsky, 2019)
Common Activation Functions

\( z = h_i W \)

Logistic: \( \sigma(z) = 1 / (1 + e^{-z}) \)

Hyperbolic tangent: \( \tanh(z) = 2\sigma(2z) - 1 = (e^{2z} - 1) / (e^{2z} + 1) \)

Rectified linear unit (ReLU): \( \text{ReLU}(z) = \max(0, z) \)
Example: Forward Pass

#define forward pass graph:

\[ \begin{align*} 
    h_{(0)} &= 0 \\
    \text{for } i \text{ in range}(1, \text{len}(x)):\ & & \quad h_{(i)} = g(U \ h_{(i-1)} + W \ x_{(i)}) \quad \# \text{update hidden state} \\
    y_{(i)} &= f(V \ h_{(i)}) \quad \# \text{update output} 
\end{align*} \]

(Geron, 2017)
Example: Forward Pass

```python
#define forward pass graph:

h_{(0)} = 0
for i in range(1, len(x)):
    h_{(i)} = g(U h_{(i-1)} + W x_{(i)})  #update hidden state
    y_{(i)} = f(V h_{(i)})  #update output
```
Example: Forward Pass

```python
# define forward pass graph:
h_{0} = 0
for i in range(1, len(x)):
    h_{(i)} = tanh(matmul(U,h_{(i-1)})+ matmul(W,x_{(i)}))  # update hidden state
    y_{(i)} = softmax(matmul(V, h_{(i)}))  # update output
```
Language Modeling

**Task:** Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$:
probability of a next word given history

$P($fork $| \text{ He ate the cake with the}$) = ?$

**History**

(He, at, the, cake, with, the)

**Training Corpus**

(fork, icing, the, carrots, cheese, spoon)

What is the next word in the sequence?
Language Modeling

Task: Estimate $P(w_n | w_1, w_2, ..., w_{n-1})$; probability of a next word given history

$P($fork $| He$ ate the cake with the) = ?

$h_t$: a vector that we hope “stores” relevant history from previous inputs:

He, at, the, cake, with,

Training Corpus

Training (fit, learn)

Last word

(the)
Tensors in PyTorch

Need a workflow system catered to numerical computation. Basic idea: defines a graph of operations on tensors.
Tensors

Need a workflow system catered to numerical computation. Basic idea: defines a graph of operations on tensors

A multi-dimensional matrix
PyTorch

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors:
- A multi-dimensional matrix
- A 2-d tensor is just a matrix.
- 1-d: vector
- 0-d: a constant / scalar
PyTorch

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

A multi-dimensional matrix

A 2-d tensor is just a matrix.
1-d: vector
0-d: a constant / scalar

Linguistic Ambiguity:
“ds” of a Tensor ≠= Dimensions of a Matrix
PyTorch

A workflow system catered to numerical computation.

Basic idea: defines a graph of operations on tensors

Why?

Efficient, high-level built-in linear algebra and machine learning optimization operations (i.e. transformations).

enables complex models, like deep learning
Language Modeling

Task: Estimate $P(w_n \mid w_1, w_2, \ldots, w_{n-1})$ 
: probability of a next word given history
$P(\text{fork} \mid \text{He ate the cake with the}) = ?$

$h_t$: a vector that we hope “stores” relevant history from previous inputs:
He, at, the, cake, with,

Training Corpus

What is the next word in the sequence?

Last word
(the)

training (fit, learn)
Example: RNN

def forward(self, X):
    #Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U,h(i-1))+ torch.matmul(W,x(i))) #update hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i))) #update output

...
Example: RNN

```python
def forward(self, X):
    # Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U, h(i-1)) + torch.matmul(W, x(i)))
        # update hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i)))
        # update output

loss_func = nn.NLLLoss()  # negative log likelihood loss
#torch.mean(-torch.sum(y*y_pred))
```

```

y(t) = f(h(t)W)

**Activation Function**

b(t) = g(h(t-1)U + x(t)V)

```

"hidden layer"
Example: RNN

def forward(self, X):
    # Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U, h(i-1)) + torch.matmul(W, x(i)))  # update hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i)))  # update output

    ... 

loss_func = nn.NLLLoss()  # negative log likelihood loss
    # torch.mean(-torch.sum(y*y_pred))
def forward(self, X):
    # Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U, h(i-1)) + torch.matmul(W, x(i))) # update hidden state
        y(i) = nn.log_softmax(torch.matmul(V, h(i))) # update output

...
Solution: Unrolling

\[ y(t) = f(b(t)W) \]

**Activation Function**

\[ b(t) = g(b(t-1)U + x(t)V) \]

Figure 9.8  Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.
Solution: Unrolling

\[ y(\text{"bill"}) = f(b(\text{"bill"})W) \]

 Activation Function

\[ b(\text{"bill"}) = g(b(\text{"the"})U + x(\text{"bill"})V) \]

Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.
def forward(self, X):
    # Basic RNN Forward Pass:
    h[0] = 0
    for i in range(1, len(x)):
        h[i] = torch.tanh(torch.matmul(U, h[i-1]) + torch.matmul(W, x[i]))  # update hidden state
        y[i] = nn.log_softmax(torch.matmul(V, h[i]))  # update output

...
def forward(self, X):
    # Basic RNN Forward Pass:
    h_0 = 0
    for i in range(1, len(x)):
        h_i = torch.tanh(torch.matmul(U, h_{i-1}) + torch.matmul(W, x_i))
        hidden state
        y_i = nn.log_softmax(torch.matmul(V, h_i))
        # update output
...

loss_func = nn.NLLLoss()  # negative log likelihood loss
    # torch.mean(-torch.sum(y*y_pred))
Back Propagation

```python
def forward(self, X):
    # Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(X)):
        h(i) = torch.tanh(torch.matmul(U, h(i-1)) + torch.matmul(W, X(i)))
    # update hidden state
    y(i) = nn.log_softmax(torch.matmul(V, h(i)))
    # update output

    loss_func = nn.NLLLoss()  # negative log likelihood loss
    cost = torch.mean(-torch.sum(y * y_pred))
```

To find the gradient for the overall graph, we use back propagation, which essentially chains together the gradients for each node in the graph.

(Geron, 2017)
Back Propagation

```python
def forward(self, X):
    # Basic RNN Forward Pass:
    h(0) = 0
    for i in range(1, len(x)):
        h(i) = torch.tanh(torch.matmul(U, h(i-1)) + torch.matmul(W, x(i)))
    # update hidden state
    y(i) = nn.log_softmax(torch.matmul(V, h(i)))
    # update output
    ...

loss_func = nn.NLLLoss()  # negative log likelihood loss
# torch.mean(-torch.sum(y*y_pred))
```

To find the gradient for the overall graph, we use **back propagation**, which essentially chains together the gradients for each node (function) in the graph.

With many recursions, the gradients can vanish or explode (become too large or small for floating point operations).
def forward(self, X):
    # Basic RNN Forward Pass:
    h_0 = 0
    for i in range(1, len(x)):
        h_i = torch.tanh(torch.matmul(U, h_{i-1}) + torch.matmul(W, x_i))  # update hidden state
        y_i = nn.log_softmax(torch.matmul(V, h_i))  # update output

    ...
How to address exploding and vanishing gradients?
How to address exploding and vanishing gradients?

$h_t$: a vector that we hope “stores” relevant history from previous inputs: *He, at, the, cake, with,*

Training Corpus

training (fit, learn)

Last word *(the)*

What is the next word in the sequence?

<table>
<thead>
<tr>
<th>icing</th>
<th>the</th>
<th>fork</th>
<th>carrots</th>
<th>cheese</th>
<th>spoon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>0.016</td>
<td>0.014</td>
<td>0.012</td>
<td>0.010</td>
<td>0.006</td>
</tr>
</tbody>
</table>
How to address exploding and vanishing gradients?

Ad Hoc approaches: e.g. stop backprop iterations very early. “clip” gradients when too high.
How to address exploding and vanishing gradients?

Dominant approach: Use Long Short Term Memory Networks (LSTM)
How to address exploding and vanishing gradients?

Dominant approach: Use Long Short Term Memory Networks (LSTM)

RNN model

“unrolled” depiction

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

“long term state”

Forget gate

Input gate

Output gate

“short term state”

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

“long term state”

Forget gate (always outputs 1)

Bias Neuron

Input Neuron (passthrough)

Output layer

Input layer

Outputs

LTU

“unrolled” depiction

(Geron, 2017)
How to address exploding and vanishing gradients?

The LSTM Cell

\[
\begin{align*}
i(t) & = \sigma(W_{xi}^T \cdot x(t) + W_{hi}^T \cdot h(t-1) + b_i) \\
f(t) & = \sigma(W_{xf}^T \cdot x(t) + W_{hf}^T \cdot h(t-1) + b_f) \\
g(t) & = \tanh(W_{xg}^T \cdot x(t) + W_{hg}^T \cdot h(t-1) + b_g) \\
c(t) & = f(t) \otimes c(t-1) + i(t) \otimes g(t) \\
h(t) & = f(t) \otimes c(t-1) + i(t) \otimes g(t)
\end{align*}
\]
The LSTM Cell

\[ \mathbf{c}_{(t)} = \mathbf{f}_{(t)} \odot \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \odot \mathbf{g}_{(t)} \]

\[ \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \odot \text{tanh} \left( \mathbf{c}_{(t)} \right) \]

\[ \mathbf{i}_{(t)} = \sigma(\mathbf{W}_{xi}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i) \]

\[ \mathbf{f}_{(t)} = \sigma(\mathbf{W}_{xf}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f) \]

\[ \mathbf{o}_{(t)} = \sigma(\mathbf{W}_{xo}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o) \]

\[ \mathbf{g}_{(t)} = \text{tanh} \left( \mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g \right) \]
Input to LSTM

- One-hot encoding?
- Word Embedding
Input to LSTM

\[ \begin{align*}
&x_t \\
&c_{t-1} \\
&h_{t-1}
\end{align*} \]

\[ \begin{align*}
f_t & = \sigma(W_f x_t + W_{hf} h_{t-1} + W_c c_{t-1} + b_f) \\
g_t & = \sigma(W_g x_t + W_{hg} h_{t-1} + W_c c_{t-1} + b_g) \\
i_t & = \sigma(W_i x_t + W_{hi} h_{t-1} + W_c c_{t-1} + b_i) \\
o_t & = \sigma(W_o x_t + W_{ho} h_{t-1} + W_c c_{t-1} + b_o) \\
c_t & = f_t \odot c_{t-1} + i_t \odot g_t \\
h_t & = o_t \odot \tanh(c_t)
\end{align*} \]
Input to LSTM

- Forget gate
- Input gate
- Output gate

LSTM cell

\[
\begin{align*}
\begin{bmatrix}
-2.0 \\
5.5 \\
-0.3 \\
-1.1 \\
6.3
\end{bmatrix}
& \rightarrow
\begin{bmatrix}
\begin{array}{c}
\mathbf{c}_{t-1} \\
\mathbf{h}_{t-1} \\
x_{(t)}
\end{array}
\end{bmatrix}

& \rightarrow
\begin{bmatrix}
\begin{array}{c}
f_{(t)} \\
g_{(t)} \\
\mathbf{i}_{(t)} \\
\mathbf{o}_{(t)}
\end{array}
\end{bmatrix}

& \rightarrow
\begin{bmatrix}
\begin{array}{c}
\mathbf{c}_{t} \\
\mathbf{h}_{t}
\end{array}
\end{bmatrix}

& \rightarrow
\begin{bmatrix}
y_{(t)} \\
\text{same}
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
1.53 \\
1.5 \\
-3.2 \\
2.3 \\
10
\end{bmatrix}
& \rightarrow
\begin{bmatrix}
12 \\
0.15 \\
1.1 \\
-0.7 \\
-5.4
\end{bmatrix}
\end{align*}
\]
The GRU

Gated Recurrent Unit

(Geron, 2017)
The GRU

Gated Recurrent Unit

relevance gate

update gate

(Geron, 2017)
The GRU

Gated Recurrent Unit

relevance gate

update gate

A candidate for updating $h$, sometimes called: $h^\sim$

(h) Gated Recurrent Unit (Gerón, 2017)
The GRU

Gated Recurrent Unit

\[
\begin{align*}
    z(t) &= \sigma(W_{xz}^T \cdot x(t) + W_{hz}^T \cdot h_{t-1} + b_z) \\
    r(t) &= \sigma(W_{xr}^T \cdot x(t) + W_{hr}^T \cdot h_{t-1} + b_r) \\
    g(t) &= \tanh(W_{xg}^T \cdot x(t) + W_{hg}^T \cdot (r(t) \otimes h_{t-1}) + b_g) \\
    h(t) &= z(t) \otimes h_{t-1} + (1 - z(t)) \otimes g(t)
\end{align*}
\]

The cake, which contained candles, was eaten.
What about the gradient?

\[ z(t) = \sigma(W_{xz}^T \cdot x(t) + W_{hz}^T \cdot h(t-1) + b_z) \]

\[ r(t) = \sigma(W_{xr}^T \cdot x(t) + W_{hr}^T \cdot h(t-1) + b_r) \]

\[ g(t) = \tanh(W_{xg}^T \cdot x(t) + W_{hg}^T \cdot (r(t) \otimes h(t-1)) + b_g) \]

\[ h(t) = z(t) \otimes h(t-1) + (1 - z(t)) \otimes g(t) \]

The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of \( h \),

\[ h(t) \approx h(t-1) \]

The cake, which contained candles, was eaten.
What about the gradient?

The gates (i.e. multiplications based on a logistic) often end up keeping the hidden state exactly (or nearly exactly) as it was. Thus, for most dimensions of $h$,

$$h(t) \approx h(t-1)$$

This tends to keep the gradient from vanishing since the same values will be present through multiple times in backpropagation through time. (The same idea applies to LSTMs but is easier to see here).

The cake, which contained candles, was eaten.
How to train an LSTM-style RNN

\[ RNN_{\text{cost}} = \text{torch.mean}(-\text{torch sum}(y*\text{torch log}(y_{\text{pred}}))) \]

# where did this come from?

Logistic Regression Likelihood:

\[ L(\beta_0, \beta_1, ..., \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i} \]

Final Cost Function:

\[ J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{V} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)} \text{ -- "cross entropy error"} \]
How to train an LSTM-style RNN

RNN_cost = torch.mean(-torch.sum(y*torch.log(y_pred)))

# where did this come from?

Logistic Regression Likelihood:

$$L(\beta_0, \beta_1, \ldots, \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i}$$

Log Likelihood:

$$\ell(\beta) = \sum_{i=1}^{N} y_i log p(x_i) + (1-y_i) log (1-p(x_i))$$

Final Cost Function:

$$J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{V} y_{i,j}^{(t)} log \hat{y}_{i,j}^{(t)} \quad -- \text{"cross entropy error"}$$
How to train an LSTM-style RNN

\[ \text{RNN\_cost} = \text{torch\_mean}(-\text{torch\_sum}(y*\text{torch\_log}(y\_pred))) \]

# where did this come from?

Logistic Regression Likelihood: \[ L(\beta_0, \beta_1, \ldots, \beta_k | X, Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i} \]

Log Likelihood: \[ \ell(\beta) = \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p(x_i)) \]

Log Loss: \[ J(\beta) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p(x_i)) \]

Final Cost Function: \[ J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)} \quad \text{"cross entropy error"} \]
How to train an LSTM-style RNN

RNN_cost = torch.mean(-torch.sum(y*torch.log(y_pred)))

# where did this come from?

Logistic Regression Likelihood: \[ L(\beta_0, \beta_1, \ldots, \beta_k|X,Y) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i} \]

Log Likelihood:
\[ \ell(\beta) = \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p(x_i)) \]

Log Loss:
\[ J(\beta) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log p(x_i) + (1 - y_i) \log (1 - p(x))(x_i) \]

Cross-Entropy Cost:
\[ J = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j} \log p(x_{i,j}) \quad \text{(a “multiclass” log loss)} \]

Final Cost Function: \[ J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)} \quad \text{-- "cross entropy error"} \]
How to train an LSTM-style RNN

\[ \text{RNN\_cost} = \text{torch\_mean}\left(-\text{torch\_sum}(y*\text{torch\_log}(y\_\text{pred}))\right) \]

#where did this come from?

Logistic Regression Likelihood:

Log Likelihood:

Log Loss:

Cross-Entropy Cost: \( (a \text{ "multiclass" log loss}) \)

Final Cost Function: -- "cross entropy error"

To Optimize Betas (all weights within LSTM cells):

Stochastic Gradient Descent (SGD)

-- optimize over one sample each iteration

Mini-Batch SDG:

-- optimize over \( b \) samples each iteration

Final Cost Function:  \[
J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{|V|} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)}
\] -- "cross entropy error"
RNN-Based Language Models

Take-Aways

- Simple RNNs are powerful models but they are difficult to train:
  - Just two functions $h(t)$ and $y(t)$ where $h(t)$ is a combination of $h(t-1)$ and $x(t)$.
  - Exploding and vanishing gradients make training difficult to converge.

- LSTM and GRU cells solve
  - Hidden states pass from one time-step to the next, allow for long-distance dependencies.
  - Gates are used to keep hidden states from changing rapidly (and thus keeps gradients under control).
  - To train: mini-batch stochastic gradient descent over cross-entropy cost.