Tasks

- **Language Modeling:**
  Generate next word, sentence
  \( \approx \) capture hidden representation of sentences.

- **Recurrent Neural Network and Sequence Models**

  how?
Language Modeling

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

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

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

Trained Language Model

What is the next word in the sequence?

Training Corpus

training
(fit, learn)

icing the fork carrots cheese spoon
Language Modeling

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

- a sequence of natural language

Trained Language Model

- What is the next word in the sequence?

Training Corpus

- training (fit, learn)
Language Modeling

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

What is the next word in the sequence?

To fully capture natural language, models get very complex!
Neural Networks: Graphs of Operations
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

$h_t = g(vecmul(h_{t-1} U) + vecmul(x_t, V))$

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)

\[
y_t = f(\text{matmul}(h_t W))
\]

Activation Function

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

short hand for vector/matrix multiply

**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

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

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

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:

\[ b_{(t)} = g(b_{(t-1)} U + x_{(t)} V) \]

\[ y_{(t)} = f(b_{(t)} W) \]

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)

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

**Activation Function**

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

**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

\[ z = h(w)W \]

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): \[ ReLU(z) = \max(0, z) \]
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): \[ ReLU(z) = \max(0, z) \]
#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

(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
```
#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
    \[ y_{(i)} = \text{softmax}(matmul(V, h_{(i)})) \] #update output
Language Modeling

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

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

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

**Training Corpus**

**What is the next word in the sequence?**

- icing
- the
- fork
- carrots
- cheese
- spoon
Language Modeling

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

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

- History
  - Last word
    - (He, at, the, cake, with, the)

- Training Corpus
  - (fit, learn)

- Training
  - (fit, learn)

- What is the next word in the sequence?

- Bar chart:
  - Bar heights: icing, the, fork, carrots, cheese, spoon


**Language Modeling**

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

: probability of a next word given history

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

**Last word**

(the)

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

*He, at, the, cake, with,*

**Training Corpus**

training (fit, learn)

[Graph showing a sequence of words and a probability distribution]

What is the next word in the sequence?
How to program neural networks:

A TensorFlow based approach.
Tensors

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

(i.stack.imgur.com)
Tensors

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
Tensors

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 $\neq$ Dimensions of a Matrix
Tensors

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
Operations on tensors are often conceptualized as graphs:

A simple example:

\[
c = \text{tensorflow.matmul}(a, b)
\]
Operations on tensors are often conceptualized as graphs:

Example:

\[ d = b + c \]
\[ e = c + 2 \]
\[ a = d \times e \]

Ingredients of a TensorFlow

**tensors***
- **variables** - persistent
- **mutable tensors**
- **constants** - constant
- **placeholders** - from data

**operations**
- an abstract computation (e.g. matrix multiply, add)
- executed by device *kernels*

---

* technically, operations that work with tensors.
Ingredients of a TensorFlow

**tensors***
- *variables* - persistent, mutable tensors
- *constants* - constant
- *placeholders* - from data

**operations**
- tf.Variable(initial_value, name)
- tf.constant(value, type, name)
- tf.placeholder(type, shape, name)

**session**
defines the environment in which operations *run*.
(like a Spark context)

**devices**
the specific devices (cpus or gpus) on which to run the session.
Operations

**tensors**
- variables - persistent
- mutable tensors
- constants - constant
- placeholders - from data

**operations**
an abstract computation
(e.g. matrix multiply, add)
executed by device *kernels*

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element-wise mathematical operations</td>
<td>Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...</td>
</tr>
<tr>
<td>Array operations</td>
<td>Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...</td>
</tr>
<tr>
<td>Matrix operations</td>
<td>MatMul, MatrixInverse, MatrixDeterminant, ...</td>
</tr>
<tr>
<td>Stateful operations</td>
<td>Variable, Assign, AssignAdd, ...</td>
</tr>
<tr>
<td>Neural-net building blocks</td>
<td>SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...</td>
</tr>
<tr>
<td>Checkpointing operations</td>
<td>Save, Restore</td>
</tr>
<tr>
<td>Queue and synchronization operations</td>
<td>Enqueue, Dequeue, MutexAcquire, MutexRelease, ...</td>
</tr>
<tr>
<td>Control flow operations</td>
<td>Merge, Switch, Enter, Leave, NextIteration</td>
</tr>
</tbody>
</table>
Ingredients of a TensorFlow

*tensors*
- *variables* - persistent mutable tensors
- *constants* - constant
- *placeholders* - from data

*operations*
- an abstract computation (e.g. matrix multiply, add)
- executed by device *kernels*
import tensorflow as tf
b = tf.constant(1.5, dtype=tf.float32, name="b")
c = tf.constant(3.0, dtype=tf.float32, name="c")
d = b+c
e = c+2
a = d*e

Example
Example

```python
import tensorflow as tf
b = tf.constant(1.5, dtype=tf.float32, name="b")
c = tf.constant(3.0, dtype=tf.float32, name="c")

d = b+c  # 1.5 + 3

e = c+2  # 3+2

a = d*e  # 4.5*5 = 22.5
```
import tensorflow as tf
b = tf.constant(1.5, dtype=tf.float32, name="b")
c = tf.constant(3.0, dtype=tf.float32, name="c")

d = b+c  #1.5 + 3
e = c+2  #3+2

a = d*e  #4.5*5 = 22.5
import tensorflow as tf
b = tf.constant([1.5, 2, 1, 4.2], dtype=tf.float32, name="b")
c = tf.constant([3, 1, 5, 10], dtype=tf.float32, name="c")
d = b+c
e = c+2
a = d*e
Example: now a 1-d tensor

```python
import tensorflow as tf
b = tf.constant([1.5, 2, 1, 4.2],
                 dtype=tf.float32, name="b")
c = tf.constant([3, 1, 5, 10],
                 dtype=tf.float32, name="c")
d = b+c  # [4.5, 3, 6, 14.2]
e = c+2  # [5, 4, 7, 12]
a = d*e  # ??
```
Example: now a 2-d tensor

```python
tensorflow as tf
b = tf.constant([[[...], [...]],
    dtype=tf.float32, name="b")
c = tf.constant([[[...], [...]],
    dtype=tf.float32, name="c")
d = b+c
e = c+2
a = tf.matmul(d,e)
```
Example: Logistic Regression

```python
X = tf.constant([[...], [...]],
                dtype=tf.float32, name="X")

y = tf.constant([...],
                dtype=tf.float32, name="y")

# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")
```
Example: Logistic Regression

```python
X = tf.constant([[...], [...]],
        dtype=tf.float32, name="X")
y = tf.constant([...],
        dtype=tf.float32, name="y")

# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1.,
                                        1.), name = "beta")

# then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")
```
Example: Logistic Regression

```python
X = tf.constant([[...], [...]],
                 dtype=tf.float32, name="X")

y = tf.constant([...],
                 dtype=tf.float32, name="y")

# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")

# then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")

# Define a *cost function* to minimize:
penalizedCost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred),
                                          reduction_indices=1))  # conceptually like |y - y_pred|
```
Optimizing Parameters -- derived from **gradients**

TensorFlow has built-in ability to derive gradients given a cost function. `tf.gradients(cost, [params])`

(http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/)
Example: Logistic Regression

```python
X = tf.constant([[[...], [...]],
             dtype=tf.float32, name="X")

y = tf.constant([...],
             dtype=tf.float32, name="y")

# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1.,
                             1.), name = "beta")

#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")

#Define a *cost function* to minimize:
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred),
                    reduction_indices=1))
```
Example: Logistic Regression

```python
X = tf.constant([[[...], [...]], dtype=tf.float32, name="X")
y = tf.constant([...], dtype=tf.float32, name="y")

# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")

#then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")

#Define a *cost function* to minimize:
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred), reduction_indices=1))

#define how to optimize and initialize:
optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
```
Example: Logistic Regression

```python
X = tf.constant([[...], [...]], dtype=tf.float32, name="X")
y = tf.constant([...], dtype=tf.float32, name="y")

# Define our beta parameter vector:
beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")

# then setup the prediction model's graph:
y_pred = tf.softmax(tf.matmul(X, beta), name="predictions")

# Define a *cost function* to minimize:
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred), reduction_indices=1))

# define how to optimize and initialize:
optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()

# iterate over optimization:
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        sess.run(training_op)

    # done training, get final beta:
    best_beta = beta.eval()
```
Language Modeling

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

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

Last word (the)

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

Training Corpus

training (fit, learn)

What is the next word in the sequence?

[Bar chart showing probabilities for words like icing, the, fork, carrots, cheese, and spoon]
#define forward pass graph:

\[ h(0) = 0 \]

for i in range(1, len(x)):
    \[ h(i) = tf.tanh(tf.matmul(U, h(i-1)) + tf.matmul(W, x(i))) \]  # update hidden state
    \[ y(i) = tf.softmax(tf.matmul(V, h(i))) \]  # update output

\[ \text{cost} = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred))) \]
Example: RNN

\[
\begin{align*}
    h(0) &= 0 \\
    \text{for } i \in \text{range}(1, \text{len}(x)):
    h(i) &= \text{tf.tanh}(\text{tf.matmul}(U, h(i-1)) + \text{tf.matmul}(W, x(i))) \\
    y(i) &= \text{tf.softmax}(\text{tf.matmul}(V, h(i)))
\end{align*}
\]

#update hidden

Activation Function

\[
\begin{align*}
    y(t) &= f(h(t)W) \\
    h(t) &= g(h(t-1)U + x(t)V)
\end{align*}
\]

Cost = \text{tf.reduce_mean}(\text{tf.reduce_sum}(y \cdot \text{tf.log}(y_{\text{pred}))))
Example: RNN

```python
#define forward pass graph:
    \( h(0) = 0 \)
    for i in range(1, len(x)):
        \( h(i) = \text{tf.tanh}(\text{tf.matmul}(U, h(i-1)) + \text{tf.matmul}(W, x(i))) \) #update hidden state
        \( y(i) = \text{tf.softmax}(\text{tf.matmul}(V, h(i))) \) #update output

... 

cost = \text{tf.reduce_mean}(-\text{tf.reduce_sum}(y*\text{tf.log}(y\_pred)))
```
Optimization:

Backward Propagation

...  

# define forward pass graph:

\[ h(0) = 0 \]

for \( i \) in range(1, len(x)):

\[ h(i) = \text{tf.tanh}(\text{tf.matmul}(U, h(i-1)) + \text{tf.matmul}(W, x(i))) \] # update hidden state

\[ y(i) = \text{tf.softmax}(\text{tf.matmul}(V, h(i))) \] # update output

...

cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred)))
Solution:
Unrolling
Solution: Unrolling

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

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

**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.
#define forward pass graph:

\[ h(i) = \text{tf.nn.relu}(\text{tf.matmul}(U, h(i-1)) + \text{tf.matmul}(W, x(i))) \]  
\text{update hidden state}

\[ y(i) = \text{tf.softmax}(\text{tf.matmul}(V, h(i))) \]  
\text{update output}
**Example: Forward Pass**

hidden_size, output_size = 5, 1

# define forward pass graph:

\[ h(i) = \text{tf.contrib.BasicRNNCell(num_units=hidden\_size, activation = tf.nn.relu)} \]

\[ y(i) = \text{tf.softmax(tf.matmul(V, h(i)))} \] # update output
hidden_size, output_size = 5, 1

#define forward pass graph:

\[ h^{(i)} = \text{tf.contrib.BasicRNNCell}(\text{num_units}=\text{hidden_size}, \text{activation} = \text{tf.nn.relu}) \]
\[ y^{(i)} = \text{tf.softmax}(\text{tf.matmul}(V, h^{(i)})) \] #update output
hidden_size, output_size = 5, 1

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size
)y = tf.softmax(tf.matmul(V, h))  #update output
Example: Forward Pass

```
hidden_size, output_size = 5, 1

define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size
```
Example: Forward Pass

hidden_size, output_size = 5, 1

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size
#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))  #softmax cost
optimizer = tf.train.AdamOptimizer(learing_rate=learning_rate)
Example: Forward Pass

```
hidden_size, output_size = 5, 1

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size

#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))) #softmax cost
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)
ishit = tf.global_variables_initializer()
```
hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unroll_steps, input_size])
y = tf.placeholder(tf.float32, [None, unroll_steps, output_size])

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size
)

#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))  #softmax cost
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
Example: Forward Pass

hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unroll_steps, input_size])
y = tf.placeholder(tf.float32, [None, unroll_steps, output_size])

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size
)

#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()

#execute training:
epochs = 1000
batch_size = 50
with tf.Session() as sess:
    init.run()

(Geron, 2017)
Example: Forward Pass

```python
hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20

X = tf.placeholder(tf.float32, [None, unroll_steps, input_size])
y = tf.placeholder(tf.float32, [None, unroll_steps, output_size])

# define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
output_size = output_size
)

# define training parameters:
learning_rate = 0.001

cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs)))

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(cost)

init = tf.global_variables_initializer()

# execute training:
epochs = 1000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iter in range(epochs):
        X_batch, y_batch = ...
        sess.run(training_op, feed_dict={X:X_batch, y:y_batch})

(Geron, 2017)
```
Example: Forward Pass

hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unroll_steps, input_size])
y = tf.placeholder(tf.float32, [None, unroll_steps, output_size])

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
output_size = output_size)

#define training parameters:
learning_rate = 0.001
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()

#execute training:
ePOCHS = 1000
batch_size = 50
with tf.Session() as sess:
    init.run()
    for iter in range(ePOCHS):
        X_batch, y_batch = ...
        sess.run(training_op, feed_dict={X:X_batch, y:y_batch})
        if iter % 100 == 0:
            c = cost.eval(feed_dict={X:X_batch, y:y_batch})
            print(iter, "	cost: ", c)
(Geron, 2017)
Example: Forward Pass

hidden_size, output_size = 5, 1
input_size, unroll_steps = 10, 20
X = tf.placeholder(tf.float32, [None, unroll_steps, input_size])
y = tf.placeholder(tf.float32, [None, unroll_steps, output_size])

#define forward pass graph:
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.BasicRNNCell(num_units=hidden_size, activation = tf.nn.relu),
    output_size = output_size
)
#define training parameters:
learning_rate = 0.001

#softmax cost
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(outputs)))

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()

#execute training:
epochs = 1000
batch_size = 50
with tf.Session() as sess:
    init.run()
    for iter in range(epochs):
        X_batch, y_batch = ...
        #fetch next batch
        sess.run(training_op, feed_dict={X:X_batch, y:y_batch})
        if iter % 100 == 0:
            c = cost.eval(feed_dict={X:X_batch, y:y_batch})
            print(iter, "cost: ", c)

(Geron, 2017)
Neural Networks: Graphs of Operations (excluding the optimization nodes)

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

Activation Function

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

"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)
Language Modeling

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

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

Training Corpus

Last word

(the)

training (fit, learn)

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>0.014</td>
<td>0.016</td>
<td>0.016</td>
<td>0.008</td>
<td>0.008</td>
<td>0.002</td>
</tr>
</tbody>
</table>
Optimization:

Backward Propagation

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.

```python
# define forward pass graph:
h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U, h(i-1)) + tf.matmul(W, x(i)))  # update hidden state
    y(i) = tf.softmax(tf.matmul(V, h(i)))  # update output
...
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_pred)))
```
Optimization:

Backward Propagation

... 

#define forward pass graph:

h(0) = 0
for i in range(1, len(x)):
    h(i) = tf.tanh(tf.matmul(U, h(i-1)) + tf.matmul(W, x(i))]
state
    y(i) = tf.softmax(tf.matmul(V, h(i))
...

cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(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).
Optimization:

Backward Propagation

For the overall graph, we use backpropagation, which essentially chains together the gradients for each node (function) in the graph.

With many recursions, the gradients can become too large or too small for floating point operations.

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.
Optimization:

Backward Propagation

(Geron, 2017)
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)

(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

"unrolled" depiction

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

The LSTM Cell

"long term state"

\[ c_{(t)} \quad x_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \]

"short term state"
How to address exploding and vanishing gradients?

The LSTM Cell

```
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
```

bias term
Common Activation Functions

\[ z = b(W) \]

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): \[ \text{ReLU}(z) = \max(0, z) \]
LSTM

The LSTM Cell

"long term state"

\[
\begin{align*}
    i_{(t)} &= \sigma(W_{xi}^T \cdot x_{(t)} + W_{hi}^T \cdot h_{(t-1)} + b_i) \\
    f_{(t)} &= \sigma(W_{xf}^T \cdot x_{(t)} + W_{hf}^T \cdot h_{(t-1)} + b_f) \\
    g_{(t)} &= \tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot h_{(t-1)} + b_g) \\
    c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
    o_{(t)} &= \sigma(W_{xo}^T \cdot x_{(t)} + W_{ho}^T \cdot h_{(t-1)} + b_o) \\
    h_{(t)} &= o_{(t)} \otimes \tanh(c_{(t)}) \\
    y_{(t)} &= W_{yo}^T \cdot h_{(t)} + b_y
\end{align*}
\]

"short term state"

Element-wise multiplication
Addition
logistic
tanh
The LSTM Cell

\[ i(t) = \sigma(W_{xi}^T \cdot x(t) + W_{hi}^T \cdot h(t-1) + b_i) \]

\[ f(t) = \sigma(W_{xf}^T \cdot x(t) + W_{hf}^T \cdot h(t-1) + b_f) \]

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

\[ c(t) = f(t) \odot c(t-1) + i(t) \odot g(t) \]

\[ h(t) = o(t) \odot \tanh(c(t)) \]

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

Element-wise multiplication

Addition
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) \\
  o(t) &= \sigma(W_{xo}^T \cdot x(t) + W_{ho}^T \cdot h(t-1) + b_o) \\
  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) &= o(t) \otimes \tanh(c(t)) \\
  y(t) &= h(t)
\end{align*}
```
Input to LSTM
Input to LSTM

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

![LSTM Cell Diagram]

- $c_{(t-1)}$
- $h_{(t-1)}$
- $x_{(t)}$
- $f_{(t)}$
- $g_{(t)}$
- $i_{(t)}$
- $o_{(t)}$
- $y_{(t)}$
- $c_{(t)}$
- $h_{(t)}$

Values:

- $-0.5$
- $3.5$
- $3.21$
- $-1.3$
- $1.6$
Input to LSTM

\[
\begin{align*}
\mathbf{c}_{(t-1)} & \rightarrow \text{Forget gate} \\
\mathbf{x}_{(t)} & \rightarrow \text{Input gate} \\
\mathbf{h}_{(t-1)} & \rightarrow \text{Output gate}
\end{align*}
\]

\[
\begin{align*}
\mathbf{c}_t & = f_{(t)} \cdot \mathbf{c}_{(t-1)} + i_{(t)} \cdot g_{(t)} \\
\mathbf{h}_t & = o_{(t)} \cdot \text{tanh}(\mathbf{c}_t)
\end{align*}
\]

\[
\begin{align*}
y_{(t)} & = \mathbf{W}_y \cdot \mathbf{h}_t + \mathbf{b}_y
\end{align*}
\]
Input to LSTM

\[
\begin{bmatrix}
-2.0 \\
5.5 \\
-0.3 \\
-1.1 \\
6.3 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.53 \\
2.5 \\
3 \\
-2.3 \\
0.76 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
-0.5 \\
3.5 \\
3.21 \\
-1.3 \\
1.6 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.53 \\
1.5 \\
-3.2 \\
2.3 \\
10 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
12 \\
0.15 \\
1.1 \\
-0.7 \\
-5.4 \\
\end{bmatrix}
\]
The GRU

Gated Recurrent Unit

(Geron, 2017)
The GRU

Gated Recurrent Unit

relevance gate

update gate

(Gerön, 2017)
The GRU

Gated Recurrent Unit

relevance gate

update gate

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

(Geron, 2017)
The cake, which contained candles, was eaten.
What about the gradient?

$$z_{(t)} = \sigma(W_{xz}^T \cdot x_{(t)} + W_{hz}^T \cdot h_{(t-1)} + b_z)$$

$$r_{(t)} = \sigma(W_{xr}^T \cdot x_{(t)} + W_{hr}^T \cdot h_{(t-1)} + b_r)$$

$$g_{(t)} = \tanh \left( W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot (r_{(t)} \otimes h_{(t-1)}) + b_g \right)$$

$$h_{(t)} = z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)}$$

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_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_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:

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

$\text{RNN}\_\text{cost} = \text{tf.reduce\_mean(-tf.reduce\_sum(y*tf.log(y\_pred)))}$

# where did this come from?

Logistic Regression Likelihood:

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

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:

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

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

RNN_cost = tf.reduce_mean(-tf.reduce_sum(y*tf.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:

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

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

\[
\text{RNN\_cost} = \text{tf.reduce\_mean}(-\text{tf.reduce\_sum}(y*\text{tf.log}(y\_pred)))
\]

#where did this come from?

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:

\[
J^{(t)} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{V} y_{i,j}^{(t)} \log \hat{y}_{i,j}^{(t)} \quad \text{-- "cross entropy error"}
\]
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