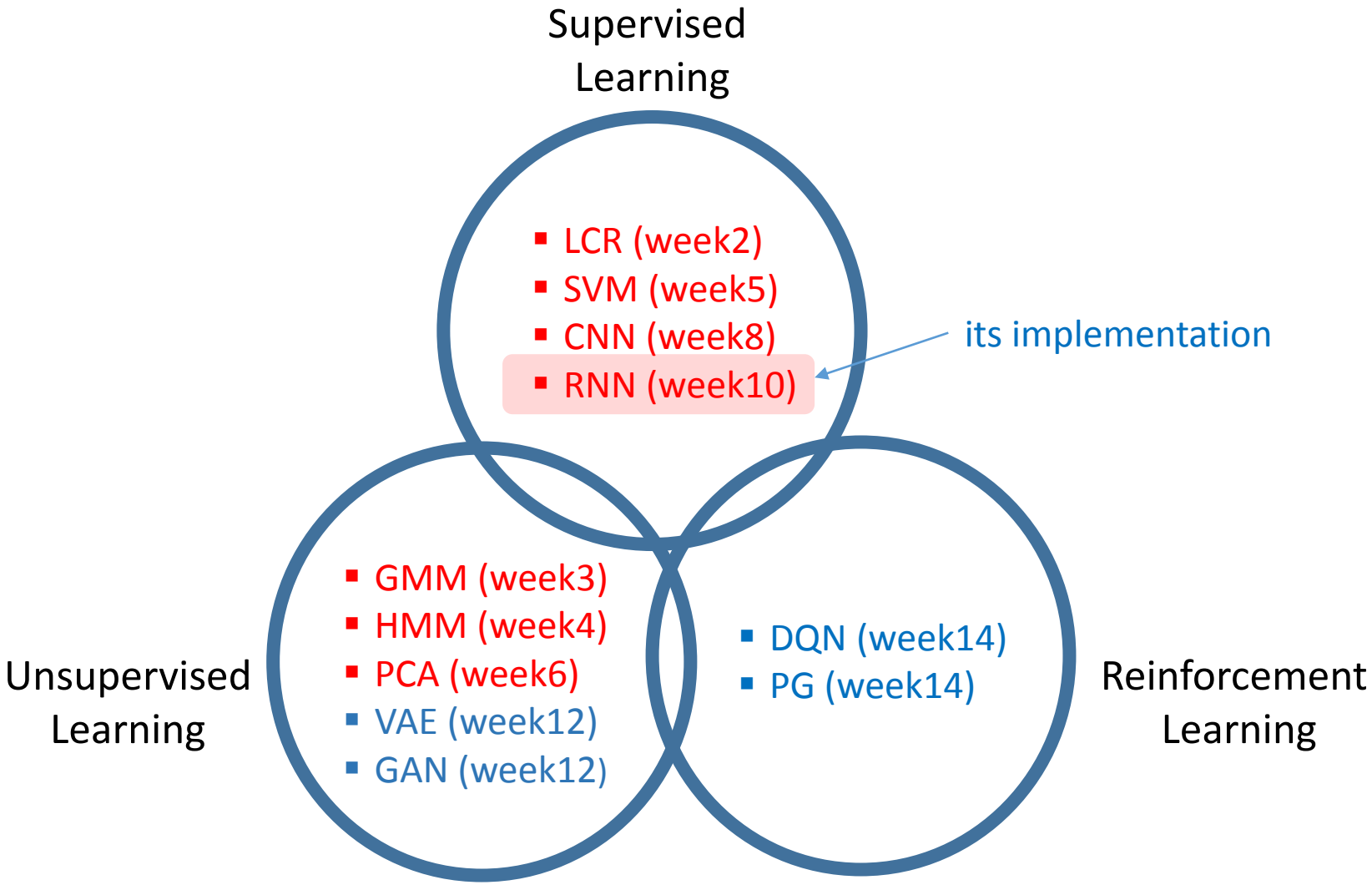
Practical Machine Learning

Lecture 11

TensorFlow – RNN/LSTM/GRU implementation

Dr. Suyong Eum

Where we are


Supervised Learning
LCR (week2)
SVM (week5)
CNN (week8)
RNN (week10)
its implementation
Unsupervised Learning
GMM (week3)
HMM (week4)
PCA (week6)
VAE (week12)
GAN (week12)
DQN (week14)
PG (week14)
Reinforcement Learning

You are going to learn

- Implementation of RNN/LSTM/GRU with Tensorflow
  - Data loading
  - Model definition
  - Evaluation
- Static and Dynamic RNN
- Dropouts in RNN

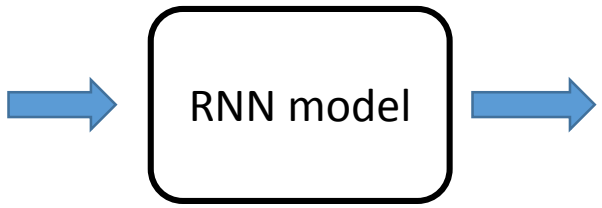
Character level language model using RNN

- Tensorflow version of character level language model
- The original one is implemented using numpy only
  - https://gist.github.com/karpathy/d4dee566867f8291f086

shakespeare.txt

- 1115390 characters
- 65 unique characters

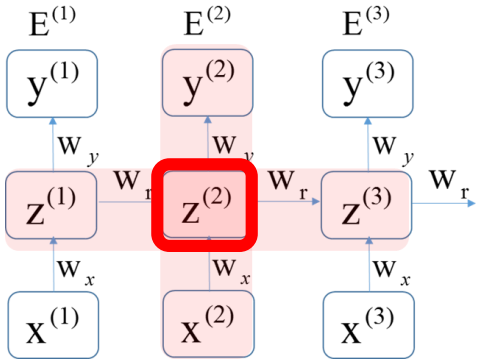
```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?
```

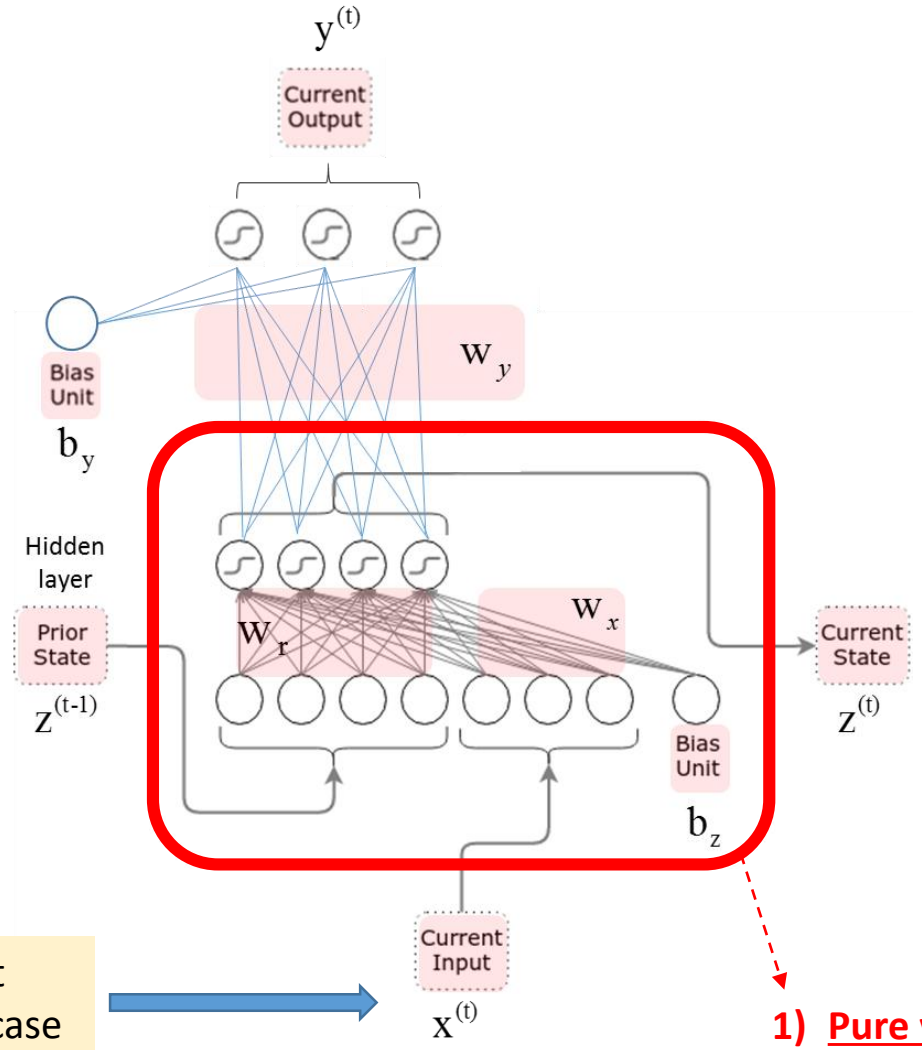


Newly generated text

KING HENRY VI I shall you sir;
When princes but friend ….

RNN review: terminology



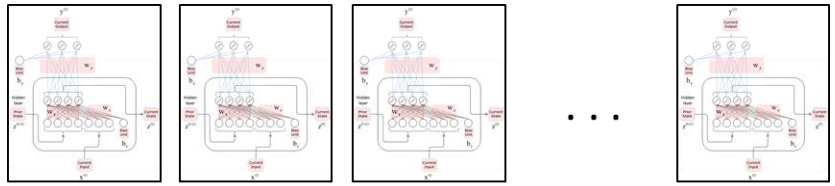


- ❑ Number of neurons in hidden layer
  - state_size
  - hidden_size

- ❑ Input dimension and Output dimension are same in this case

[0, 0, 0, …, 1, …, 0]
One hot encoding

- ❑ Number of steps
  - num_steps
  - seq_length



1) **Pure vanilla RNN cell**
2) **LSTM cell**
3) **GRU cell**

Data loading

1) Data loading

```
# Global config variables
batch_size = 200
state_size = 100
num_steps = 5

learning_rate = 0.0001

# Data reading
data = open('shakespeare.txt').read()
chars = list(set(data))
num_classes = len(chars)

char_to_ix = {ch: i for i, ch in enumerate(chars)}
ix_to_char = {i: ch for i, ch in enumerate(chars)}

# Convert the characters to index
# Creating data and corresponding label
inputs = [char_to_ix[ch] for ch in data[:]] # its shape: (1115390,)

print (data[0:14])
print (inputs[0:14])
print (len(chars))
```

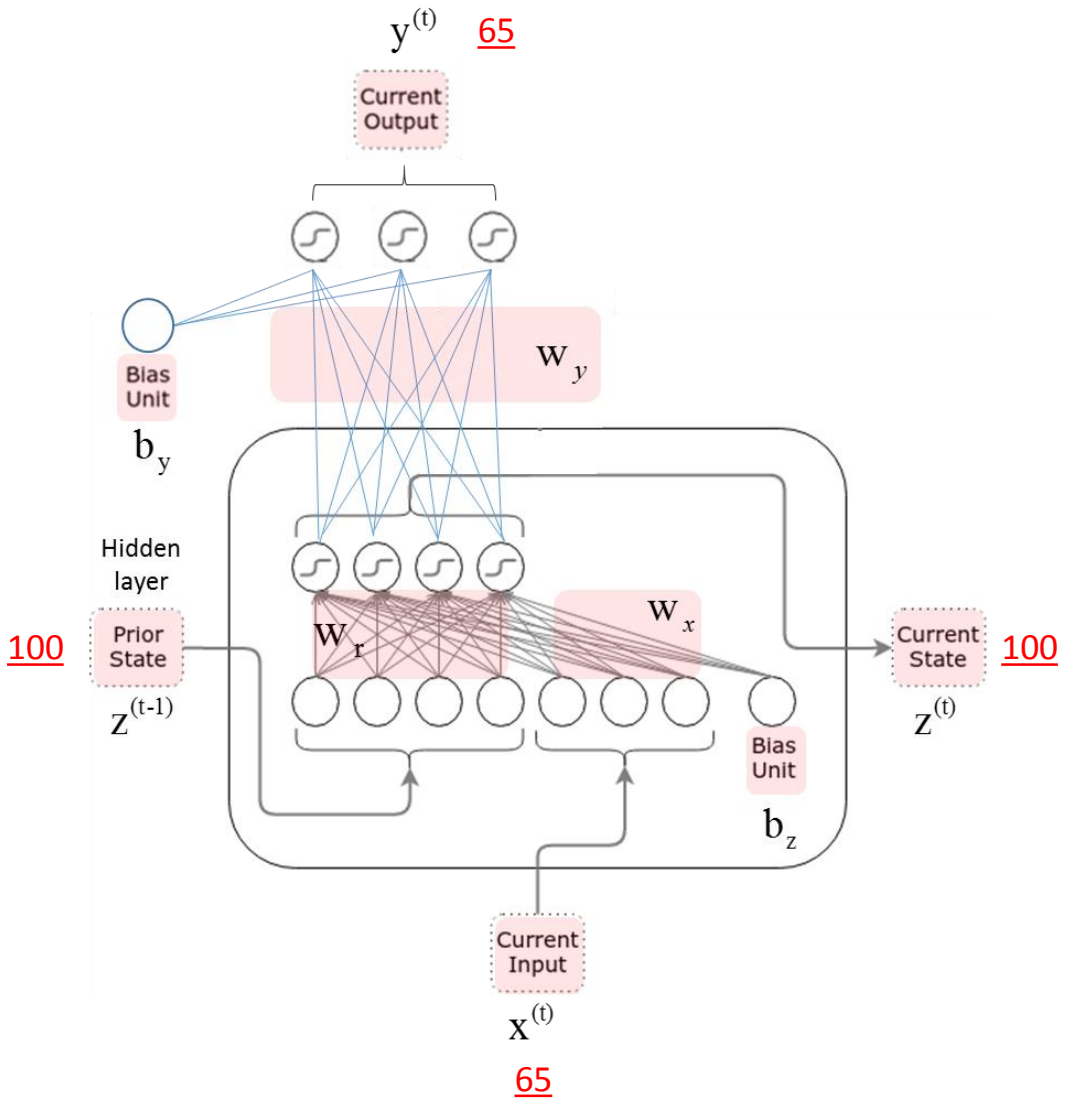
```
First Citizen:
[42, 46, 36, 23, 20, 33, 31, 46, 20, 46, 30, 57, 63, 19]
65
```

- Creating a dictionary which maps between "character" and "corresponding index"
- Converting a list of characters to a list of their corresponding numbers

1) Data loading

```
# Global config variables
batch_size = 200
state_size = 100
num_steps = 5

learning_rate = 0.0001

# Data reading
data = open('shakespeare.txt').read()
chars = list(set(data))
num_classes = len(chars)

char_to_ix = {ch: i for i, ch in enumerate(chars)}
ix_to_char = {i: ch for i, ch in enumerate(chars)}

# Convert the characters to index
# Creating data and corresponding label
inputs = [char_to_ix[ch] for ch in data[:]] # its shape: (1115390,)

print (data[0:14])
print (inputs[0:14])
print (len(chars))
```

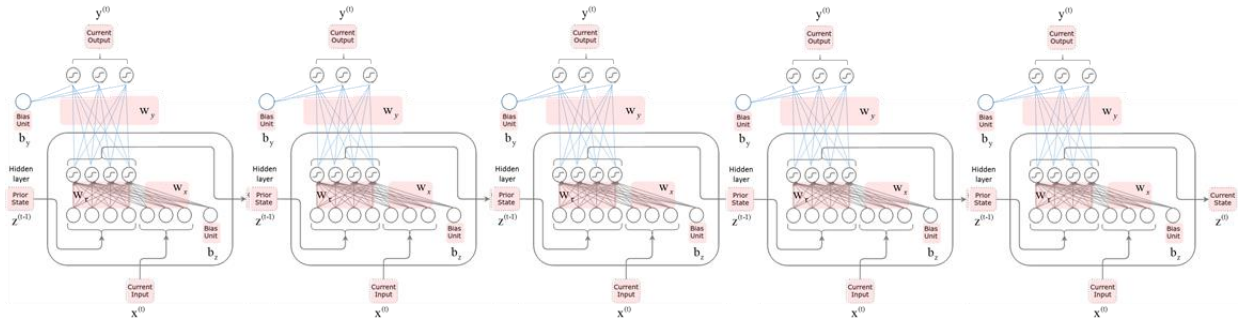
```
First Citizen:
[42, 46, 36, 23, 20, 33, 31, 46, 20, 46, 30, 57, 63, 19]
65
```



1) Data loading

```python
# Global config variables
batch_size = 200
state_size = 100
num_steps = 5

learning_rate = 0.0001

# Data reading
data = open('shakespeare.txt').read()
chars = list(set(data))
num_classes = len(chars)

char_to_ix = {ch: i for i, ch in enumerate(chars)}
ix_to_char = {i: ch for i, ch in enumerate(chars)}

# Convert the characters to index
# Creating data and corresponding label
inputs = [char_to_ix[ch] for ch in data[:]] # its shape: (1115390,)

print (data[0:14])
print (inputs[0:14])
print (len(chars))
```

```
First Citizen:
[42, 46, 36, 23, 20, 33, 31, 46, 20, 46, 30, 57, 63, 19]
65
```

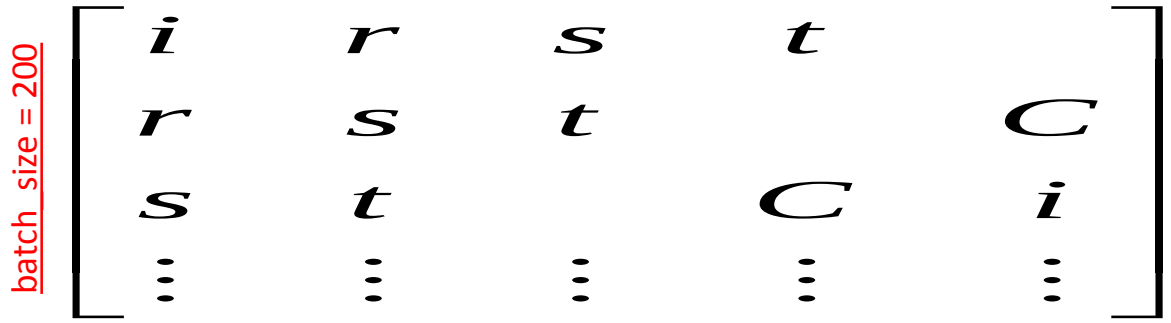


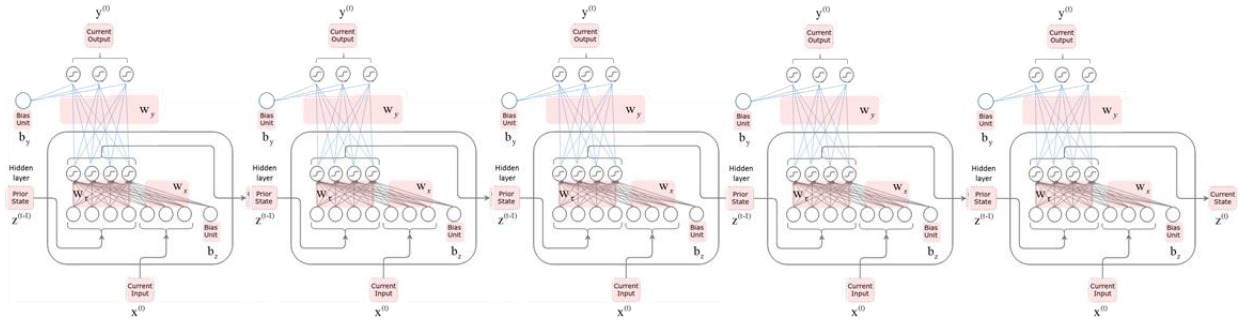
1) Data loading

```
# Global config variables
batch_size = 200
state_size = 100
num_steps = 5

learning_rate = 0.0001

# Data reading
data = open('shakespeare.txt').read()
chars = list(set(data))
num_classes = len(chars)

char_to_ix = {ch: i for i, ch in enumerate(chars)}
ix_to_char = {i: ch for i, ch in enumerate(chars)}

# Convert the characters to index
# Creating data and corresponding label
inputs = [char_to_ix[ch] for ch in data[:]] # its shape: (1115390,)

print (data[0:14])
print (inputs[0:14])
print (len(chars))
```

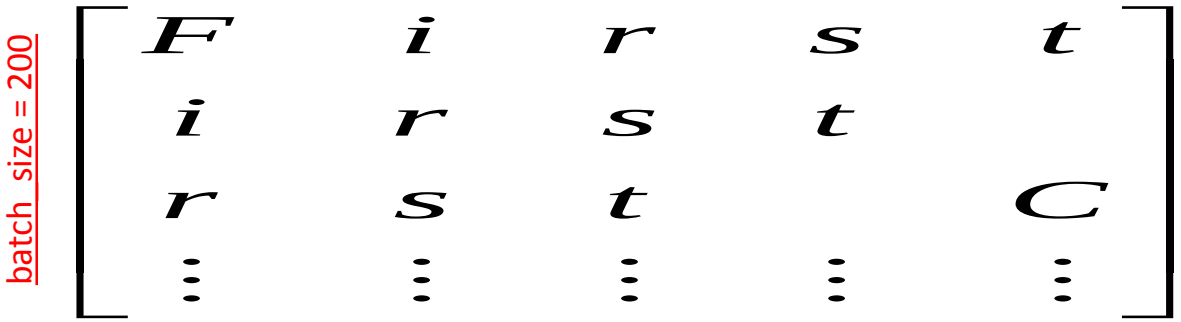
```
First Citizen:
[42, 46, 36, 23, 20, 33, 31, 46, 20, 46, 30, 57, 63, 19]
65
```

batch_size = 200





batch_size = 200



num_steps = 5

Truncated backpropagation for 5 steps

RNN model

2) RNN model

```
# RNN model from Tensorflow
input_data = tf.placeholder(tf.int64, [batch_size, num_steps])
label_data = tf.placeholder(tf.int64, [batch_size, num_steps])
init_state = tf.zeros([batch_size, state_size])

# RNN input
x_one_hot = tf.one_hot(input_data, num_classes)
rnn_inputs = tf.unstack(x_one_hot, axis=1)

# Creating RNN Cell
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.contrib.rnn.static_rnn(cell, rnn_inputs, init_state)
```

- Feeding data into the model using placeholder.
  - input_data = [3, 45, 34, …]

```
x = [[1,2,4], [1,3,4]]
num_classes=5

x_one_hot = tf.one_hot(x, num_classes)
print(sess.run(x_one_hot))
```

[[0,2,4][1,3,2]]

```
[[[ 1.  0.  0.  0.  0.]
  [ 0.  0.  1.  0.  0.]
  [ 0.  0.  0.  0.  1.]]

 [[ 0.  1.  0.  0.  0.]
  [ 0.  0.  0.  1.  0.]
  [ 0.  0.  1.  0.  0.]]]
```

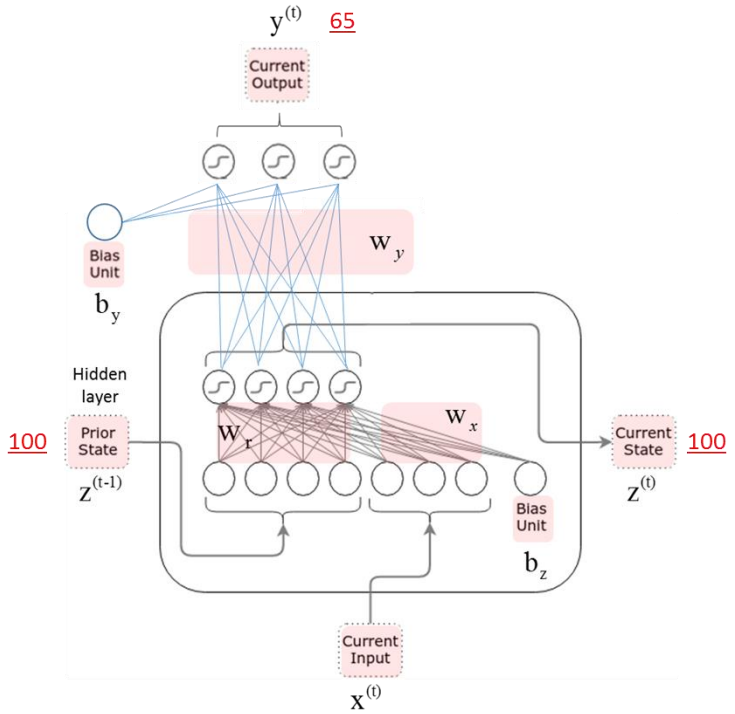
tf.unstack(one_hot, axis=1)

x = [[0,2,4]
[1,3,2]]

```
[array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.]], dtype=float32), array([[ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]], dtype=float32), array([[ 0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  1.,  0.,  0.]], dtype=float32)]
```

2) RNN model: two functions for a vanilla RNN implementation

```
# RNN model from Tensorflow
input_data = tf.placeholder(tf.int64, [batch_size, num_steps])
label_data = tf.placeholder(tf.int64, [batch_size, num_steps])
init_state = tf.zeros([batch_size, state_size])

# RNN input
x_one_hot = tf.one_hot(input_data, num_classes)
rnn_inputs = tf.unstack(x_one_hot, axis=1)

# Creating RNN Cell
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.contrib.rnn.static_rnn(cell, rnn_inputs, init_state)
```



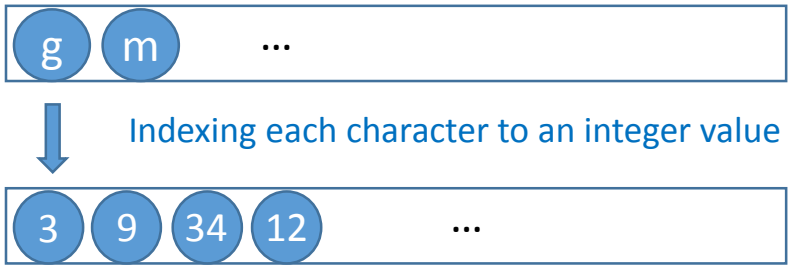
1) cell = tf.contrib.rnn.BasicRNNCell(state_size)
   - A vanilla RNN cell creation
   - The number of neurons in a cell
2) rnn_outputs, final_state = tf.nn.static_rnn(cell, rnn_inputs, inital_state)
   - three input parameters
     a. cell: cell info which is a return
     b. rnn_inputs: [batch_size][num_classes] x [num_steps]
     c. Initial_state: [batch_size][state_size]
   - two outputs
     a. rnn_outputs: [batch_size][state_size] x [num_steps]
     b. final_state: rnn_outputs[-1]

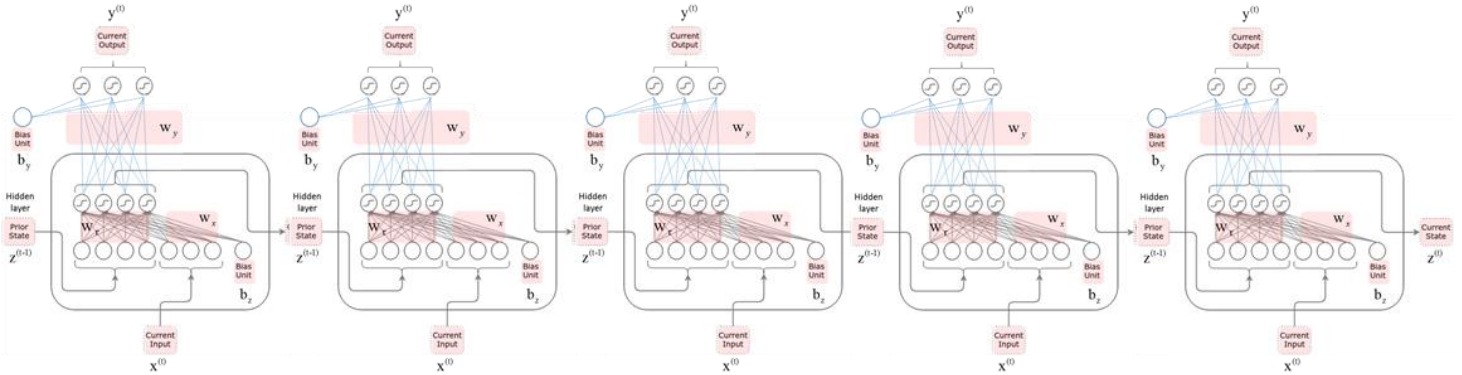
2) RNN model: rnn_inputs

shakespeare.txt

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?
```

Total characters: 1115390





num_step

batch_size

3	9	34	12	29
9	34	12	29	2
34	12	29	2	32
...	...	...	...	...

```
x_one_hot = tf.one_hot([34], 65)
print (sess.run(x_one_hot))
```

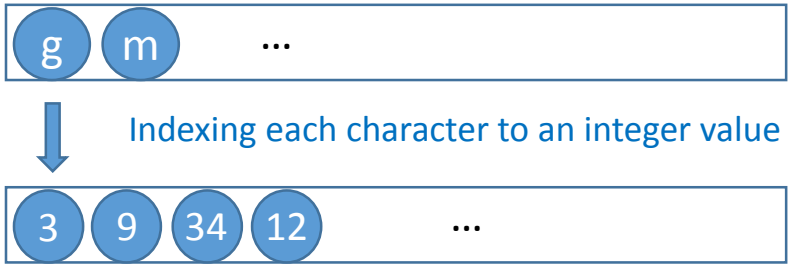
```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

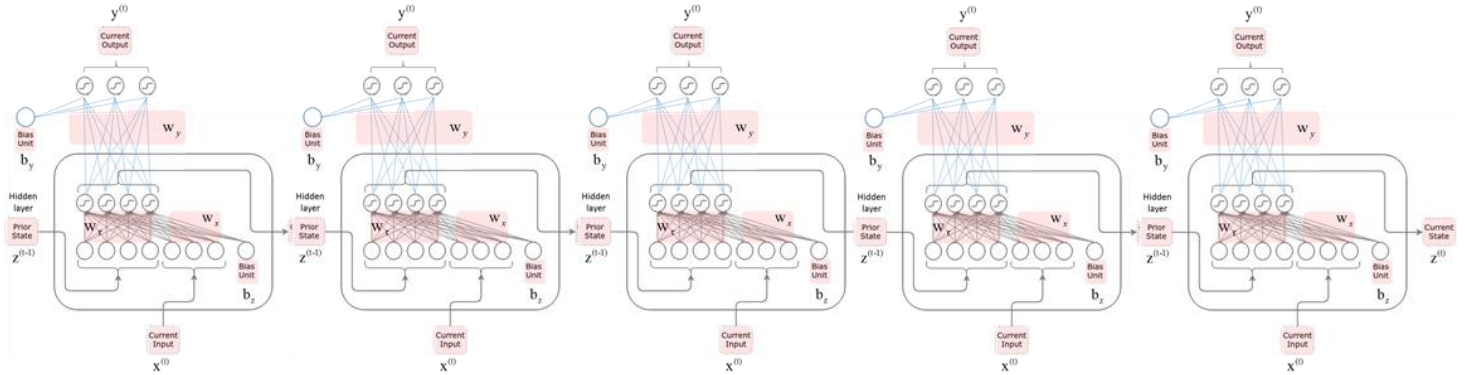
2) RNN model: rnn_inputs

shakespeare.txt

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?
```

Total characters: 1115390





num_step

batch_size

3	9	34	12	29
9	34	12	29	2
34	12	29	2	32
...	...	...	...	...

rnn_inputs[0] rnn_inputs[1] rnn_inputs[2] rnn_inputs[3] rnn_inputs[4]

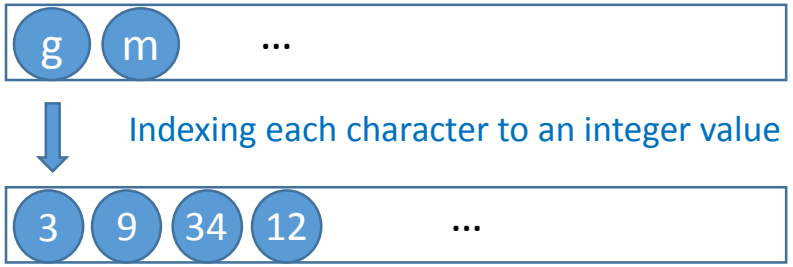
tf.unstack(one_hot, axis=1)

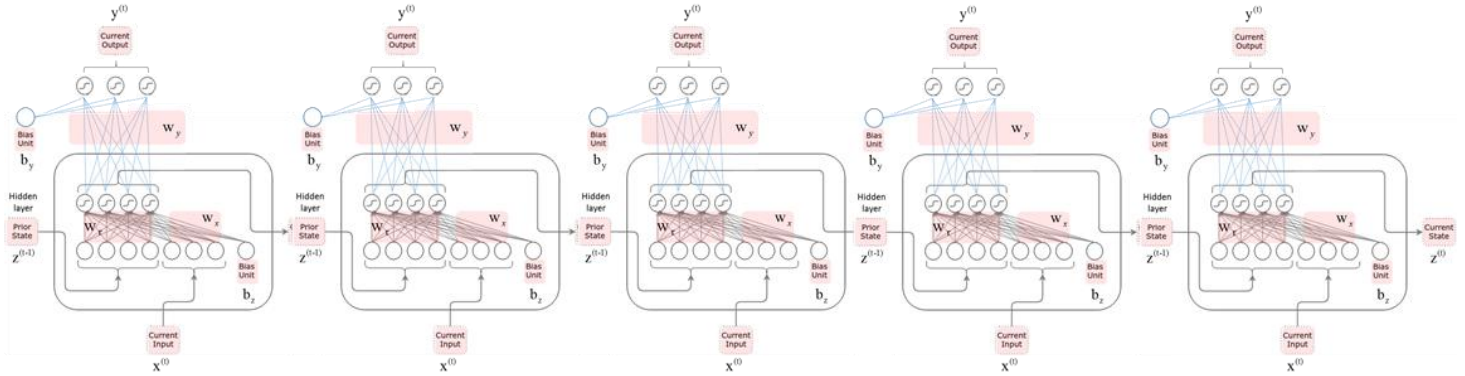
2) RNN model: rnn_inputs

shakespeare.txt

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?
```

Total characters: 1115390





num_step

batch_size

3	9	34	12	29
9	34	12	29	2
34	12	29	2	32
...	...	...	...	...

rnn_inputs[0] rnn_inputs[1] rnn_inputs[2] rnn_inputs[3] rnn_inputs[4]

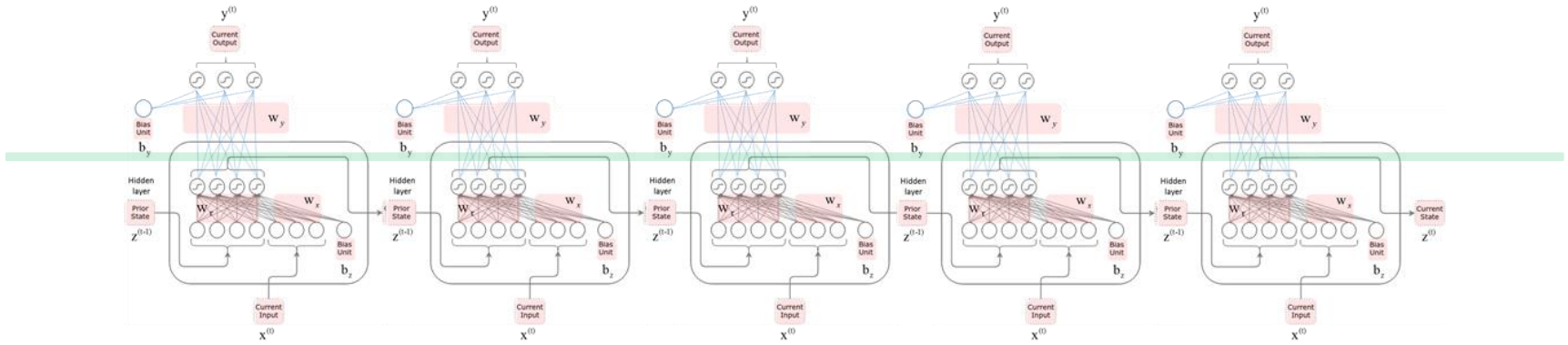
❑ rnn_inputs: [batch_size][num_classes] x [num_steps]

2) RNN model: rnn_outs

rnn_outputs[0]	rnn_outputs[1]	rnn_outputs[2]	rnn_outputs[3]	rnn_outputs[4]
3	64	9	29	2
44	42	29	2	32
12	29	1	32	7
…	…	…	…	…

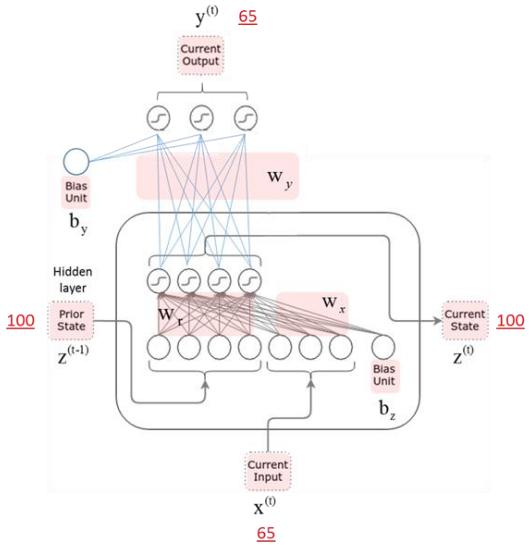
rnn_outputs[] = [batch_size] x [state_size]

final_state == rnn_outputs[4]

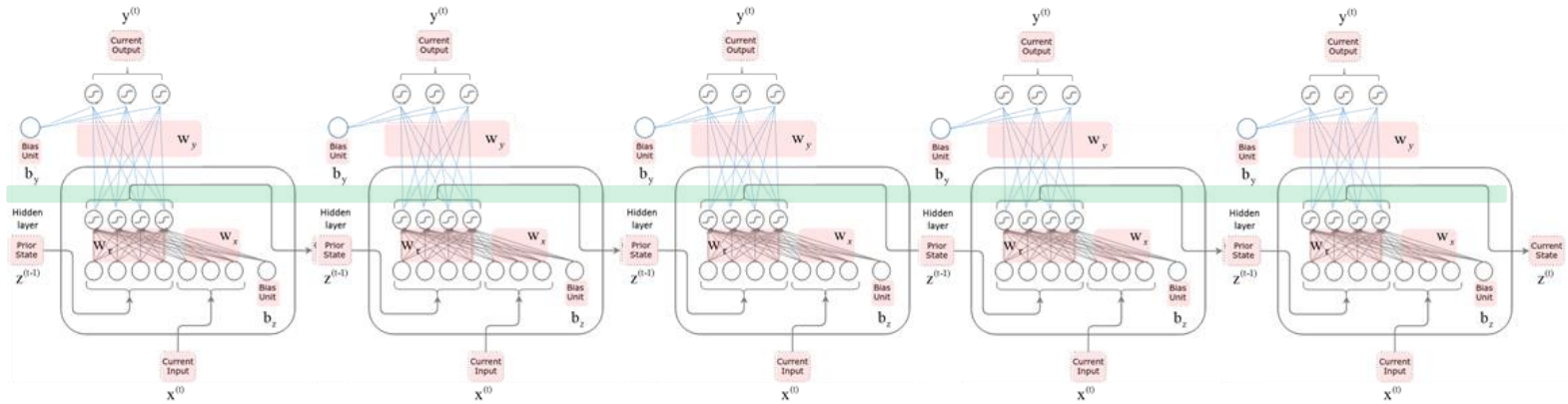


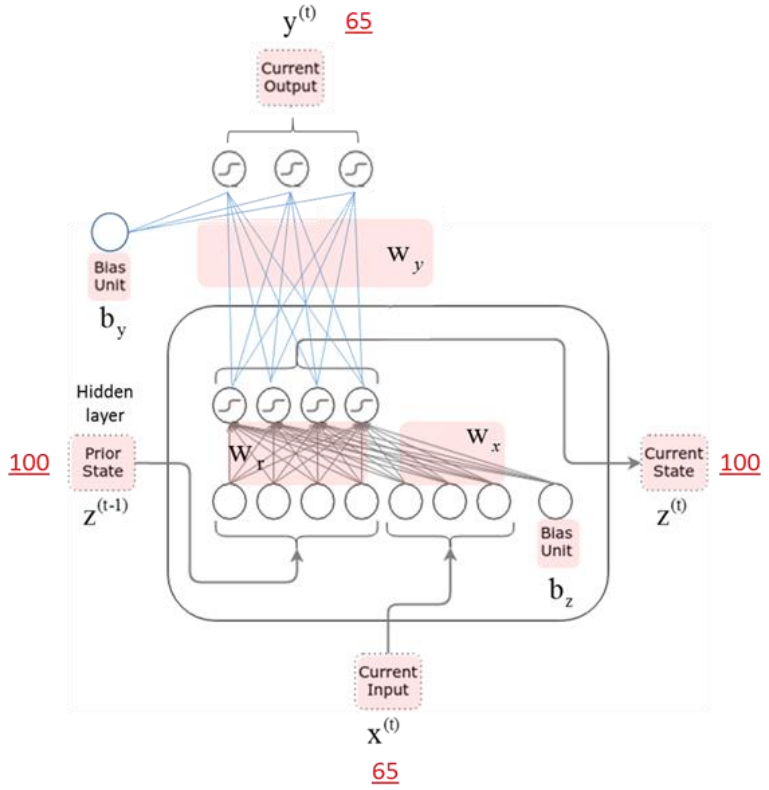
batch_size					
	3	9	34	12	29
	9	34	12	29	2
	34	12	29	2	32
	…	…	…	…	…

num_step



2) RNN model: rnn_outs






```
# Creating RNN Cell
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.static_rnn(cell, rnn_inputs, init_state)

print(rnn_inputs)
print(" ------------------------------------------------")
print(rnn_outputs)
```

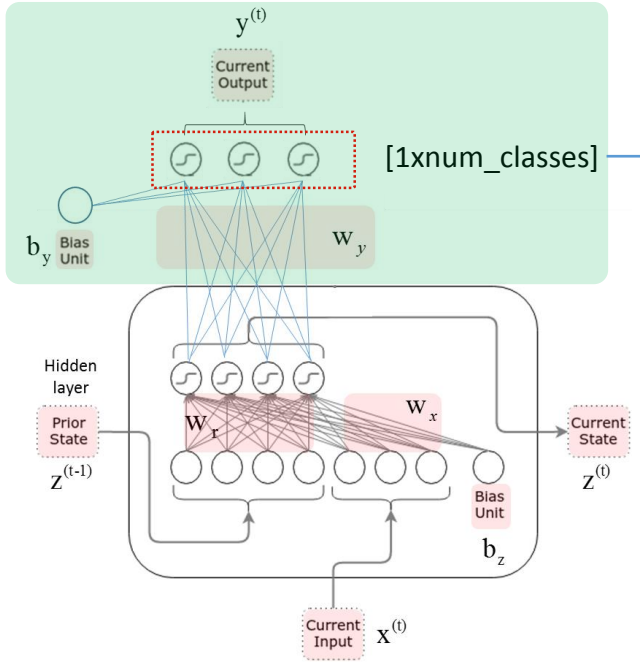
```
[<tf.Tensor 'unstack:0' shape=(200, 65) dtype=float32>, <tf.Tensor 'unstack:1' shape=(200,
65) dtype=float32>, <tf.Tensor 'unstack:2' shape=(200, 65) dtype=float32>, <tf.Tensor 'unst
ack:3' shape=(200, 65) dtype=float32>, <tf.Tensor 'unstack:4' shape=(200, 65) dtype=float32
>]
 ------------------------------------------------
[<tf.Tensor 'rnn/rnn/basic_rnn_cell/Tanh:0' shape=(200, 100) dtype=float32>, <tf.Tensor 'rn
n/rnn/basic_rnn_cell/Tanh_1:0' shape=(200, 100) dtype=float32>, <tf.Tensor 'rnn/rnn/basic_r
nn_cell/Tanh_2:0' shape=(200, 100) dtype=float32>, <tf.Tensor 'rnn/rnn/basic_rnn_cell/Tanh_
3:0' shape=(200, 100) dtype=float32>, <tf.Tensor 'rnn/rnn/basic_rnn_cell/Tanh_4:0' shape=(2
00, 100) dtype=float32>]
```

200 65 5

- rnn_inputs [batch_size][num_classes] x [num_steps]
- rnn_outputs [batch_size][state_size] x [num_steps]

200 100 5

2) RNN model: after cell



logits[]

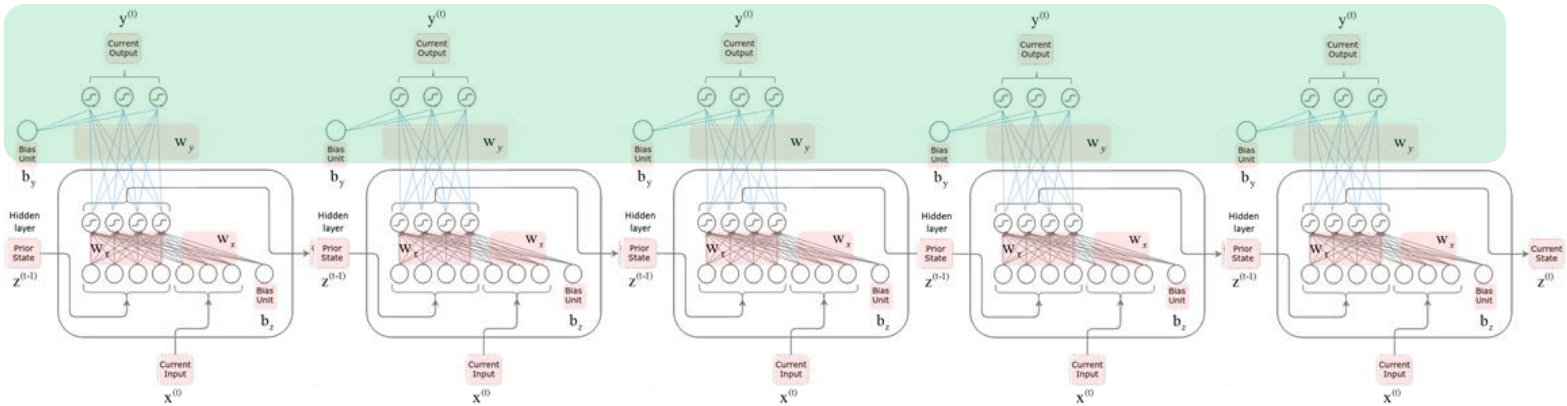
e.g., [0.1, 0.3, …, 0.1]

num_classes (65)

```
#logits and predictions
W2 = tf.get_variable('W2', [state_size, num_classes])
b2 = tf.get_variable('b2', [num_classes], initializer=tf.constant_initializer(0.0))

logits = [tf.matmul(rnn_output, W2) + b2 for rnn_output in rnn_outputs]
```

logits[0] = [batch_size] [num_classes]
= [200] [65]

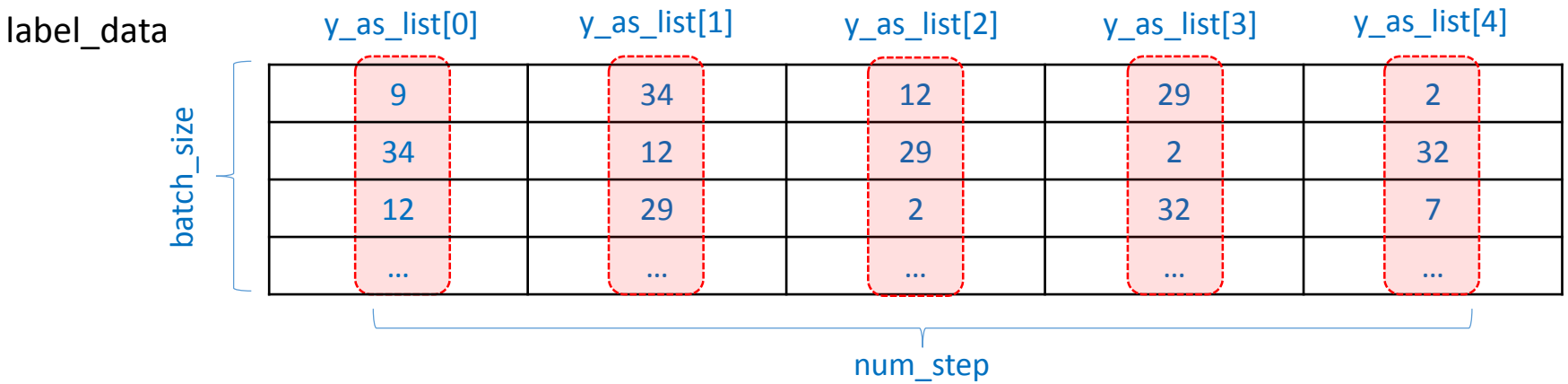
logits[0] logits[1] logits[2] logits[3] logits[4]



Evaluation

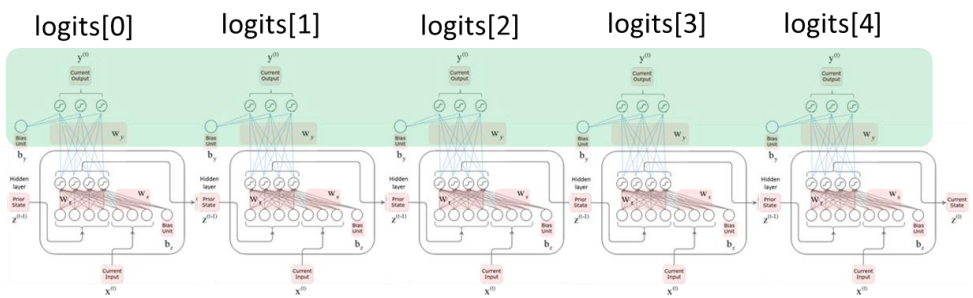
3) Evaluation: loss calculation

```
# Turn our y placeholder into a list of labels
y_as_list = tf.unstack(label_data, num=num_steps, axis=1)

# losses and train_step
losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(labels=label, logits=logit) \
          for logit, label in zip(logits, y_as_list)]
total_loss = tf.reduce_mean(losses)

# accuracy
correct_prediction = [tf.equal(tf.argmax(logit, 1), label) \
                      for logit, label in zip(logits, y_as_list)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)
```

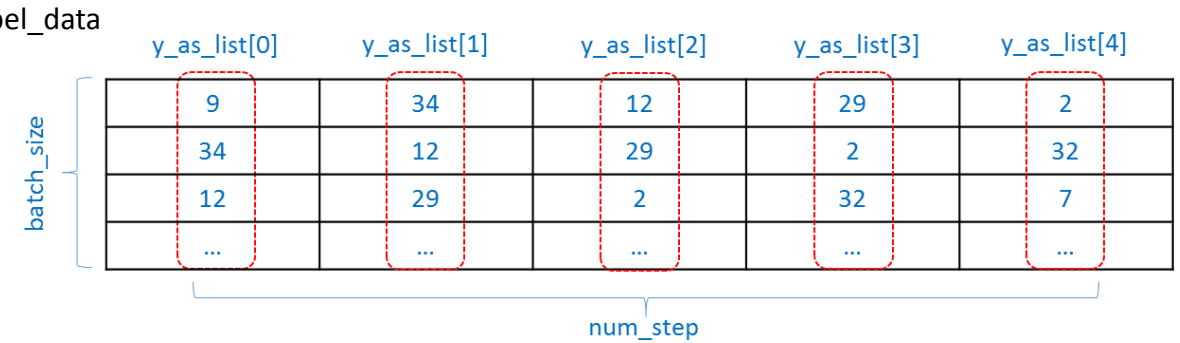


3) Evaluation: loss calculation

```
# Turn our y placeholder into a list of labels
y_as_list = tf.unstack(label_data, num=num_steps, axis=1)

# losses and train_step
losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(labels=label, logits=logit) \
          for logit, label in zip(logits, y_as_list)]
total_loss = tf.reduce_mean(losses)

# accuracy
correct_prediction = [tf.equal(tf.argmax(logit, 1), label) \
                      for logit, label in zip(logits, y_as_list)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)
```



logits[0] = [batch_size] [num_classes]
= [200] [65]



y_as_list[0] = [batch_size] [num_classes]
= [200] [65]

3) Evaluation: loss calculation

```
# Turn our y placeholder into a list of labels
y_as_list = tf.unstack(label_data, num=num_steps, axis=1)

# losses and train_step
losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(labels=label, logits=logit) \
          for logit, label in zip(logits, y_as_list)]
total_loss = tf.reduce_mean(losses)

# accuracy
correct_prediction = [tf.equal(tf.argmax(logit, 1), label) \
                      for logit, label in zip(logits, y_as_list)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)
```

- tf.nn.softmax_cross_entropy_with_logits(labels, logits)
  - logits: predicted label which is the output from the previous layer
    - e.g., [0.1, 0.4, 0.5]
  - labels: true label, one-hot encoded
    - e.g., [0, 0, 1]

batch1	logit	Label	Cross Entropy (error)	batch2	logit	Label	Cross Entropy(error)
Data1-1	0.1, 0.2, 0.7	0, 0, 1	-ln(0.1)*0-ln(0.2)*0-ln(0.7)*1 = 0.357	Data2-1	0.3, 0.3, 0.4	0, 0, 1	-ln(0.3)*0-ln(0.3)*0-ln(0.4)*1 = 0.916
Data1-2	0.1, 0.6, 0.3	0, 1, 0	-ln(0.1)*0-ln(0.6)*1-ln(0.3)*0 = 0.511	Data2-2	0.3, 0.4, 0.3	0, 1, 0	-ln(0.3)*0-ln(0.4)*1-ln(0.3)*0 = 0.916
Data1-3	0.3, 0.3, 0.4	1, 0, 0	-ln(0.3)*1-ln(0.3)*0-ln(0.4)*0 = 1.204	Data2-3	0.1, 0.1, 0.8	1, 0, 0	-ln(0.1)*1-ln(0.1)*0-ln(0.8)*0 = 2.303
		Mean	0.691			Mean	1.287

3) Evaluation: accuracy calculation

```
# Turn our y placeholder into a list of labels
y_as_list = tf.unstack(label_data, num=num_steps, axis=1)

# losses and train_step
losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(labels=label, logits=logit) \
          for logit, label in zip(logits, y_as_list)]
total_loss = tf.reduce_mean(losses)

# accuracy
correct_prediction = [tf.equal(tf.argmax(logit, 1), label) \
                      for logit, label in zip(logits, y_as_list)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)
```

- tf.argmax(tensor, 1): return index of the item which has the max value
- tf.equal(x, y): return true if x==y otherwise false

	logit	label	x=tf.argmax(logit)	y=tf.argmax(label)	tf.cast(tf.equal(x,y))
Data1-1	0.1, 0.2, 0.7	0, 0, 1	2	2	1=tf.cast(True)
Data1-2	0.1, 0.6, 0.3	0, 1, 0	1	1	1=tf.cast(True)
Data1-3	0.3, 0.3, 0.4	1, 0, 0	2	0	0=tf.cast(False)
				Mean	2/3

Static & Dynamic RNN

Static & Dynamic RNN

Static RNN

```
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.static_rnn(cell,
                                            rnn_inputs, init_state)
```

- ❑ rnn_inputs [batch_size][num_classes] x [num_steps]
- ❑ rnn_outputs [batch_size][state_size] x [num_steps]

- Creating an unrolled graph for a fixed RNN
- Slow graph creation
- Fast execution

Dynamic RNN

```
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.dynamic_rnn(cell,
                                             rnn_inputs, init_state)
```

- ❑ rnn_inputs [batch_size x num_steps x num_classes]
- ❑ rnn_outputs [batch_size x num_steps x state_size]

- Dynamically contruct a graph
- Faster graph creation
- Slow execution

Static & Dynamic RNN

Static RNN

```
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.static_rnn(cell,
                                            rnn_inputs, init_state)
```

- rnn_inputs [batch_size][num_classes] x [num_steps]
- rnn_outputs [batch_size][state_size] x [num_steps]

```
# Creating RNN Cell
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.static_rnn(cell, rnn_inputs, init_state)

print(rnn_inputs)
```

```
[<tf.Tensor 'unstack:0' shape=(200, 65) dtype=float32>, <tf.Tensor 'unstack:1' shape=(200,
 65) dtype=float32>, <tf.Tensor 'unstack:2' shape=(200, 65) dtype=float32>, <tf.Tensor 'uns
tack:3' shape=(200, 65) dtype=float32>, <tf.Tensor 'unstack:4' shape=(200, 65) dtype=float3
2>, <tf.Tensor 'unstack:5' shape=(200, 65) dtype=float32>, <tf.Tensor 'unstack:6' shape=(20
0, 65) dtype=float32>, <tf.Tensor 'unstack:7' shape=(200, 65) dtype=float32>, <tf.Tensor 'u
nstack:8' shape=(200, 65) dtype=float32>, <tf.Tensor 'unstack:9' shape=(200, 65) dtype=floa
t32>]
```

A list of a tensor

Dynamic RNN

```
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.dynamic_rnn(cell,
                                             rnn_inputs, init_state)
```

- rnn_inputs [batch_size x num_steps x num_classes]
- rnn_outputs [batch_size x num_steps x state_size]

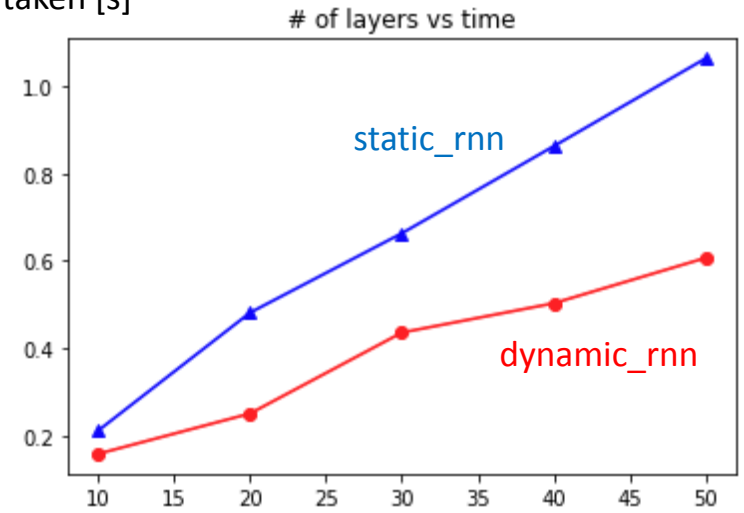
```
# Creating RNN Cell
cell = tf.contrib.rnn.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.dynamic_rnn(cell, rnn_inputs, init_state)

print(rnn_inputs)
```

```
Tensor("one hot:0", shape=(200, 10, 65), dtype=float32)
```

a tensor

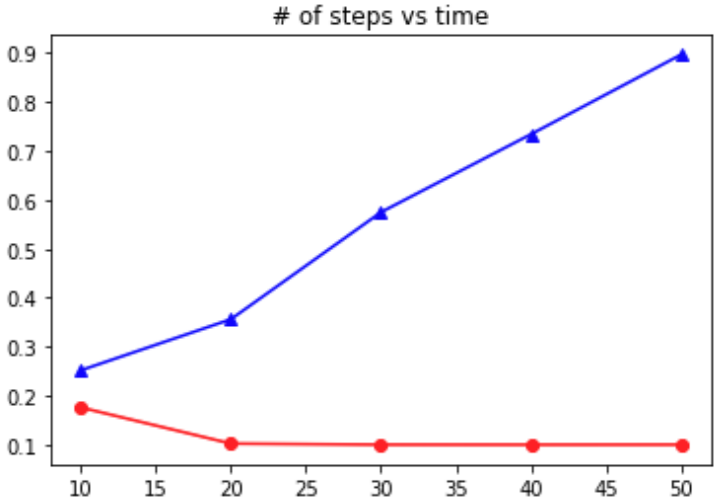
Static & Dynamic RNN: testing?

time taken [s]

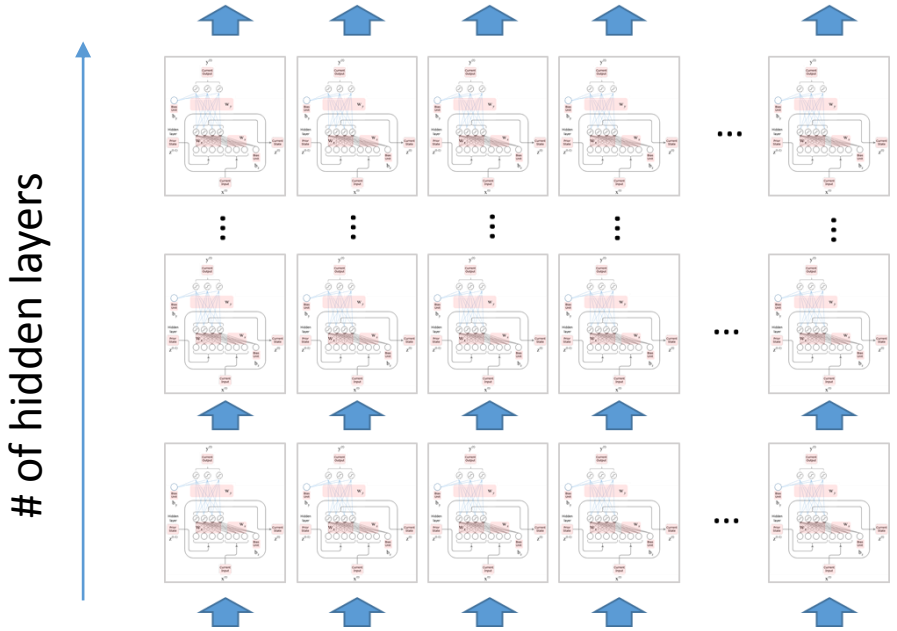


# of hidden layers

time taken [s]



# of steps



- # of hidden layers: 30
- # of steps: 10

	static_rnn	dynamic_rnn
1 epoch	9.04	10.70

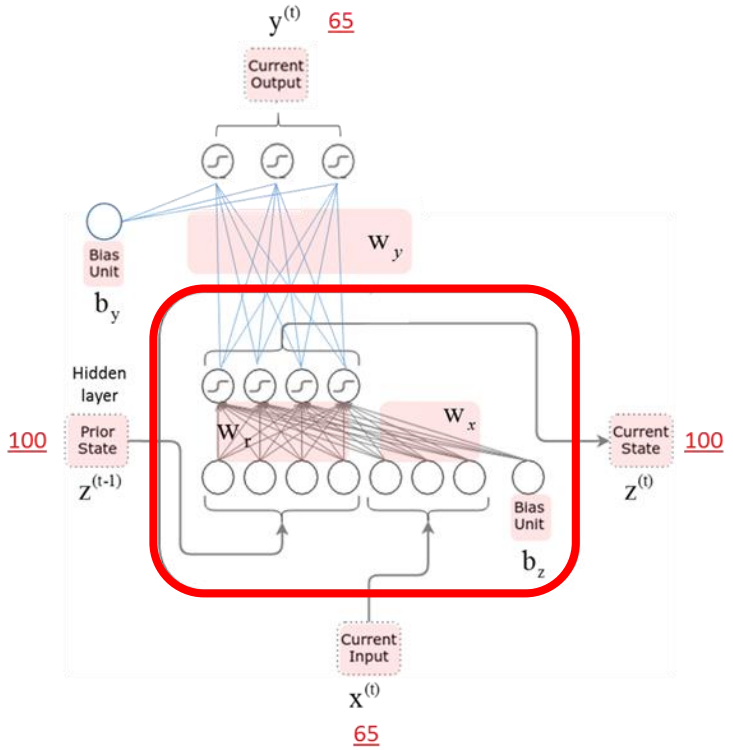
LSTM and GRU

How about LSTM & GRU?

- Just creating a different cell and the rest of them is same.

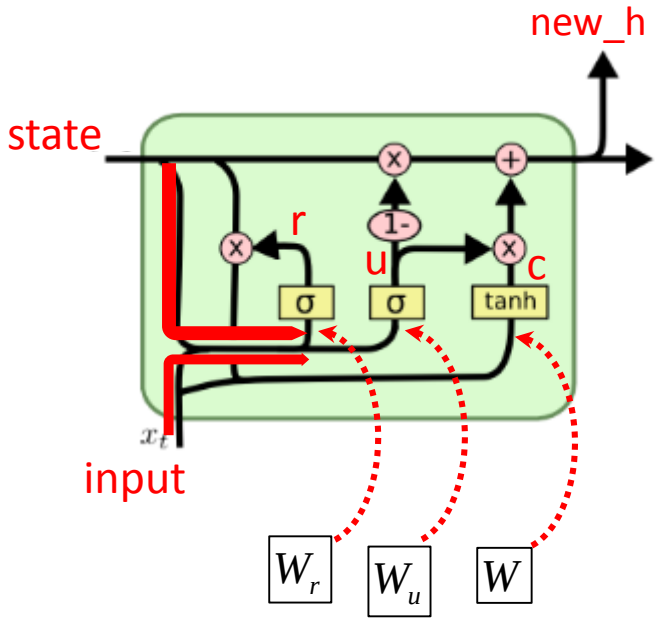
```
# RNN model from Tensorflow
input_data = tf.placeholder(tf.int64, [batch_size, num_steps], name='input_data')
label_data = tf.placeholder(tf.int64, [batch_size, num_steps], name='label_data')

embedding = tf.get_variable('embedding', [num_classes, state_size])
rnn_inputs = tf.nn.embedding_lookup(embedding, input_data)

cell = tf.nn.rnn_cell.BasicRNNCell(state_size)
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers, state_is_tuple=True)
init_state = cell.zero_state(batch_size, tf.float32)
rnn_outputs, final_state = tf.nn.dynamic_rnn(cell, rnn_inputs, initial_state=init_state)

# logits and predictions
W2 = tf.get_variable('W2', [state_size, num_classes])
b2 = tf.get_variable('b2', [num_classes], initializer=tf.constant_initializer(0.0))

logits = tf.reshape(tf.matmul(tf.reshape(rnn_outputs, [-1, state_size]), W2) \
                    + b2, [batch_size, num_steps, num_classes])
predictions = tf.nn.softmax(logits)

# loss calculation
total_loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, \
                                                                           labels=label_data))

# accuracy
correct_prediction = [tf.equal(tf.argmax(logits, 2), label_data)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdamOptimizer(learning_rate).minimize(total_loss)
```

	Types of RNN cells
RNN	cell = tf.nn.rnn_cell.BasicRNNCell(num_states)
LSTM	cell = tf.nn.rnn_cell.LSTMCell(num_states)
GRU	cell = tf.nn.rnn_cell.GRUCell(num_states)



How about a customized Cell … e.g., GRUCell from RNNCell

```python
class GRUCell(tf.nn.rnn_cell.RNNCell):

    def __init__(self, num_units):
        self._num_units = num_units

    @property
    def state_size(self):
        return self._num_units

    @property
    def output_size(self):
        return self._num_units

    def __call__(self, inputs, state, scope=None):
        with tf.variable_scope(scope or type(self).__name__):
            with tf.variable_scope("Gates"):
                ru = rnn_cell_impl._linear([inputs, state],
                             2 * self._num_units, True, tf.constant_initializer(1.0))
                ru = tf.nn.sigmoid(ru)
                r, u = tf.split(ru, 2, 1)
            with tf.variable_scope("Candidate"):
                c = tf.nn.tanh(rnn_cell_impl._linear([inputs, r * state],
                                             self._num_units, True))
            new_h = u * state + (1 - u) * c
        return new_h, new_h
```



$$u = \sigma(W_u \cdot [state, input])$$

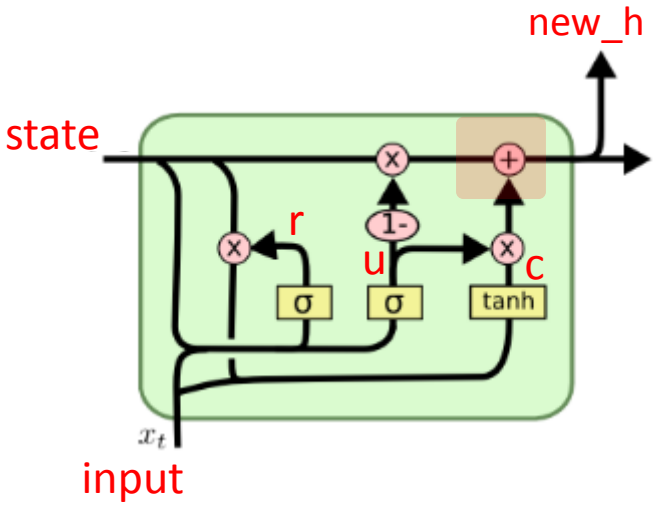
$$r = \sigma(W_r \cdot [state, input])$$

$$c = \tanh(W \cdot [r * state, input])$$

$$new\_h = (1-u) * state + u * c$$

How about a customized Cell … e.g., GRUCell from RNNCell

```python
class GRUCell(tf.nn.rnn_cell.RNNCell):

    def __init__(self, num_units):
        self._num_units = num_units

    @property
    def state_size(self):
        return self._num_units

    @property
    def output_size(self):
        return self._num_units

    def __call__(self, inputs, state, scope=None):
        with tf.variable_scope(scope or type(self).__name__):
            with tf.variable_scope("Gates"):
                ru = rnn_cell_impl._linear([inputs, state],
                            2 * self._num_units, True, tf.constant_initializer(1.0))
                ru = tf.nn.sigmoid(ru)
                r, u = tf.split(ru, 2, 1)
            with tf.variable_scope("Candidate"):
                c = tf.nn.tanh(rnn_cell_impl._linear([inputs, r * state],
                                                     self._num_units, True))
            new_h = u * state + (1 - u) * c
        return new_h, new_h
```



$$u = \sigma(W_u \cdot [state, input])$$

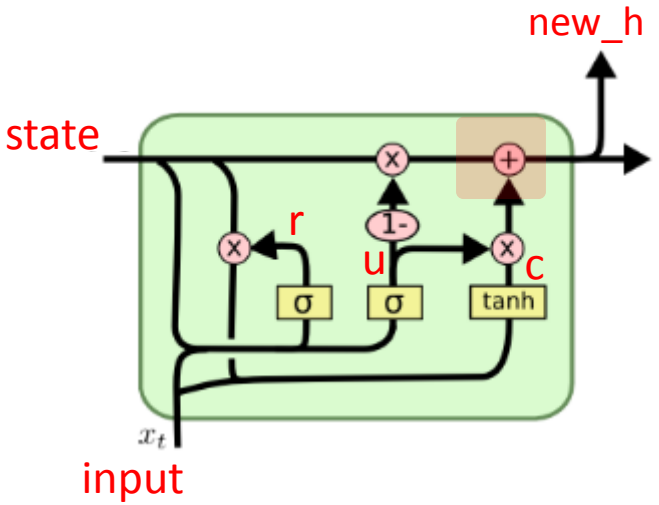
$$r = \sigma(W_r \cdot [state, input])$$

$$c = \tanh(W \cdot [r * state, input])$$

$$new\_h = (1-u) * state + u * c$$

How about a customized Cell … e.g., GRUCell from RNNCell

```python
class GRUCell(tf.nn.rnn_cell.RNNCell):

    def __init__(self, num_units):
        self._num_units = num_units

    @property
    def state_size(self):
        return self._num_units

    @property
    def output_size(self):
        return self._num_units

    def __call__(self, inputs, state, scope=None):
        with tf.variable_scope(scope or type(self).__name__):
            with tf.variable_scope("Gates"):
                ru = rnn_cell_impl._linear([inputs, state],
                                           2 * self._num_units, True, tf.constant_initializer(1.0))
                ru = tf.nn.sigmoid(ru)
                r, u = tf.split(ru, 2, 1)
            with tf.variable_scope("Candidate"):
                c = tf.nn.tanh(rnn_cell_impl._linear([inputs, r * state],
                                                     self._num_units, True))
            new_h = u * state + (1 - u) * c
        return new_h, new_h
```



$$u = \sigma(W_u \cdot [state, input])$$

$$r = \sigma(W_r \cdot [state, input])$$

$$c = \tanh(W \cdot [r * state, input])$$

$$new\_h = (1-u) * state + u * c$$

Where are “Weights” defined?

_linear() in core_rnn_cell_impl.py

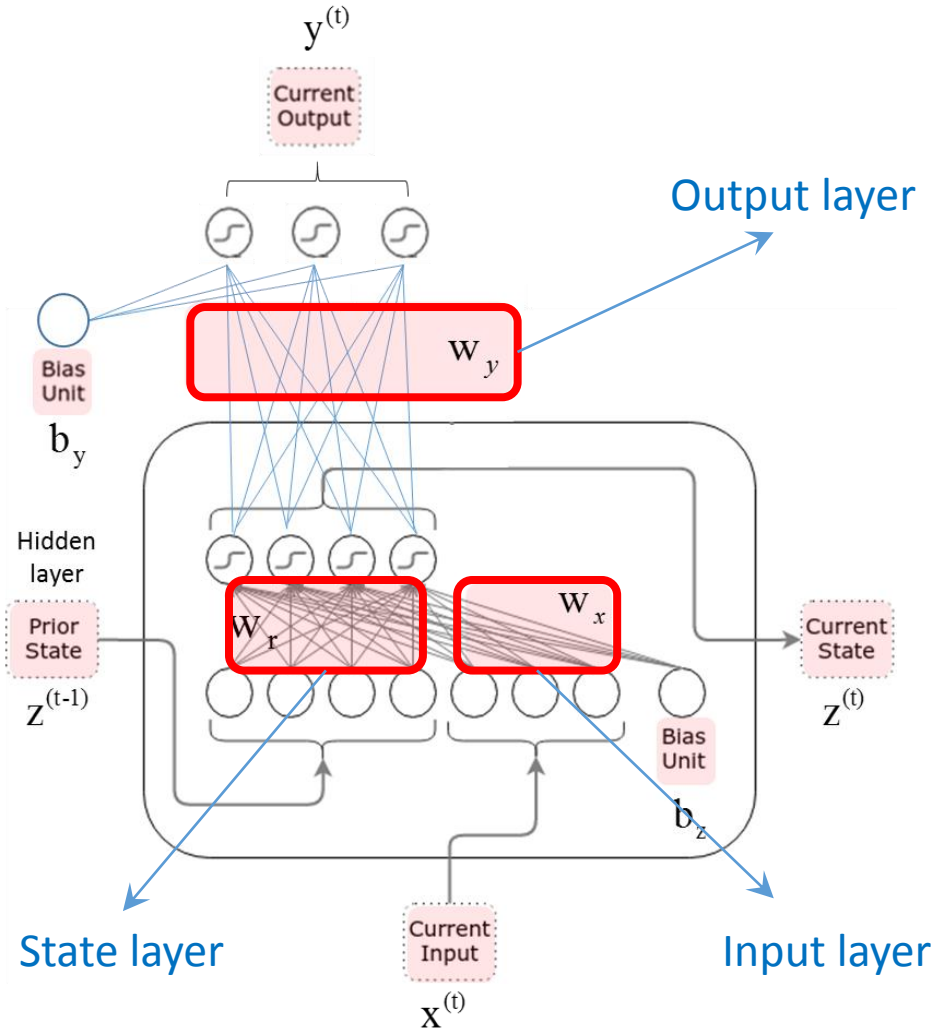
Dropouts in RNN

Dropouts which layers and how?

- Dropouts have been used to prevent overfitting in various neural networks.
- How to dropout links in each layer of RNN?
  - Input layer
  - Output layer
  - State layer

```
cell = tf.nn.rnn_cell.BasicRNNCell(state_size)

# Dropout
cell = tf.nn.rnn_cell.DropoutWrapper(cell)

rnn_outputs, final_state = tf.nn.dynamic_rnn(cell, rnn_inputs,
```

```
__init__(
    cell,
    input_keep_prob=1.0,
    output_keep_prob=1.0,
    state_keep_prob=1.0,
    variational_recurrent=False,
    input_size=None,
    dtype=None,
    seed=None,
    dropout_state_filter_visitor=None
)
```

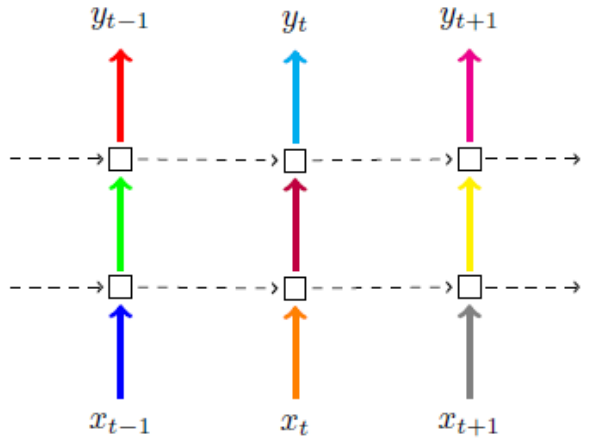


Dropouts which layers and how?

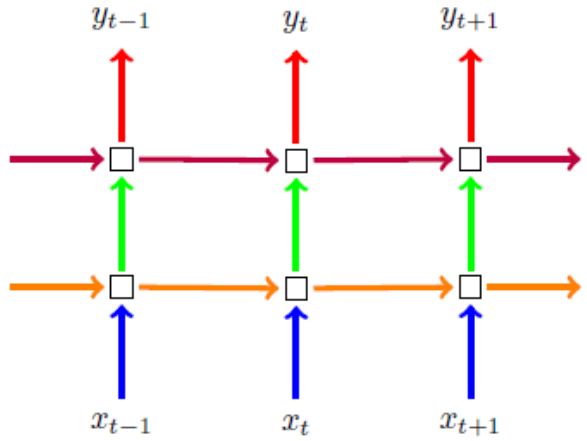
- Dropouts have been used to prevent overfitting in various neural networks.
- How to dropout links in each layer of RNN?
  - Input layer
  - Output layer
  - State layer

```
cell = tf.nn.rnn_cell.BasicRNNCell(state_size)

# Dropout
cell = tf.nn.rnn_cell.DropoutWrapper(cell)

rnn_outputs, final_state = tf.nn.dynamic_rnn(cell, rnn_inputs,
```

```
__init__(
    cell,
    input_keep_prob=1.0,
    output_keep_prob=1.0,
    state_keep_prob=1.0,
    variational_recurrent=False,
    input_size=None,
    dtype=None,
    seed=None,
    dropout_state_filter_visitor=None
)
```



(a) Naive dropout RNN



(b) Variational RNN

- variational_recurrent = False
- Different droptout mask at each time step

- variational_recurrent = True
- Same dropout mask at each time step

https://arxiv.org/abs/1512.05287

Backup Slides

Static & Dynamic RNN

static_rnn

```
# RNN model from Tensorflow
input_data = tf.placeholder(tf.int64, [batch_size, num_steps], name='input_data')
label_data = tf.placeholder(tf.int64, [batch_size, num_steps], name='label_data')
#init_state = tf.zeros([batch_size, state_size])

embedding = tf.get_variable('embedding', [num_classes, state_size])
#rnn_inputs = tf.nn.embedding_lookup(embedding, input_data)
rnn_inputs = [tf.squeeze(i) for i in tf.split(tf.nn.embedding_lookup(embedding, input_data),
                                              num_steps, 1)]

cell = tf.nn.rnn_cell.BasicRNNCell(state_size)
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers, state_is_tuple=True)
init_state = cell.zero_state(batch_size, tf.float32)
rnn_outputs, final_state = tf.nn.static_rnn(cell, rnn_inputs, initial_state=init_state)

#logits and predictions
W2 = tf.get_variable('W2', [state_size, num_classes])
b2 = tf.get_variable('b2', [num_classes], initializer=tf.constant_initializer(0.0))

logits = [tf.matmul(rnn_output, W2) + b2 for rnn_output in rnn_outputs]

y_as_list = [tf.squeeze(i, squeeze_dims=[1]) for i in tf.split(label_data, num_steps, 1)]

loss_weights = [tf.ones([batch_size]) for i in range(num_steps)]
losses = tf.contrib.legacy_seq2seq.sequence_loss_by_example(logits, y_as_list, loss_weights)
total_loss = tf.reduce_mean(losses)

# accuracy
correct_prediction = [tf.equal(tf.argmax(logit, 1), label) \
                      for logit, label in zip(logits, y_as_list)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdamOptimizer(learning_rate).minimize(total_loss)
```

dynamic_rnn

```
# RNN model from Tensorflow
input_data = tf.placeholder(tf.int64, [batch_size, num_steps], name='input_data')
label_data = tf.placeholder(tf.int64, [batch_size, num_steps], name='label_data')

embedding = tf.get_variable('embedding', [num_classes, state_size])
rnn_inputs = tf.nn.embedding_lookup(embedding, input_data)

cell = tf.nn.rnn_cell.BasicRNNCell(state_size)
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers, state_is_tuple=True)
init_state = cell.zero_state(batch_size, tf.float32)
rnn_outputs, final_state = tf.nn.dynamic_rnn(cell, rnn_inputs, initial_state=init_state)

# logits and predictions
W2 = tf.get_variable('W2', [state_size, num_classes])
b2 = tf.get_variable('b2', [num_classes], initializer=tf.constant_initializer(0.0))

logits = tf.reshape(tf.matmul(tf.reshape(rnn_outputs, [-1, state_size]), W2) \
                    + b2, [batch_size, num_steps, num_classes])
predictions = tf.nn.softmax(logits)

# loss calculation
total_loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, \
                                                                           labels=label_data))

# accuracy
correct_prediction = [tf.equal(tf.argmax(logits, 2), label_data)]
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

# Training
train_step = tf.train.AdamOptimizer(learning_rate).minimize(total_loss)
```