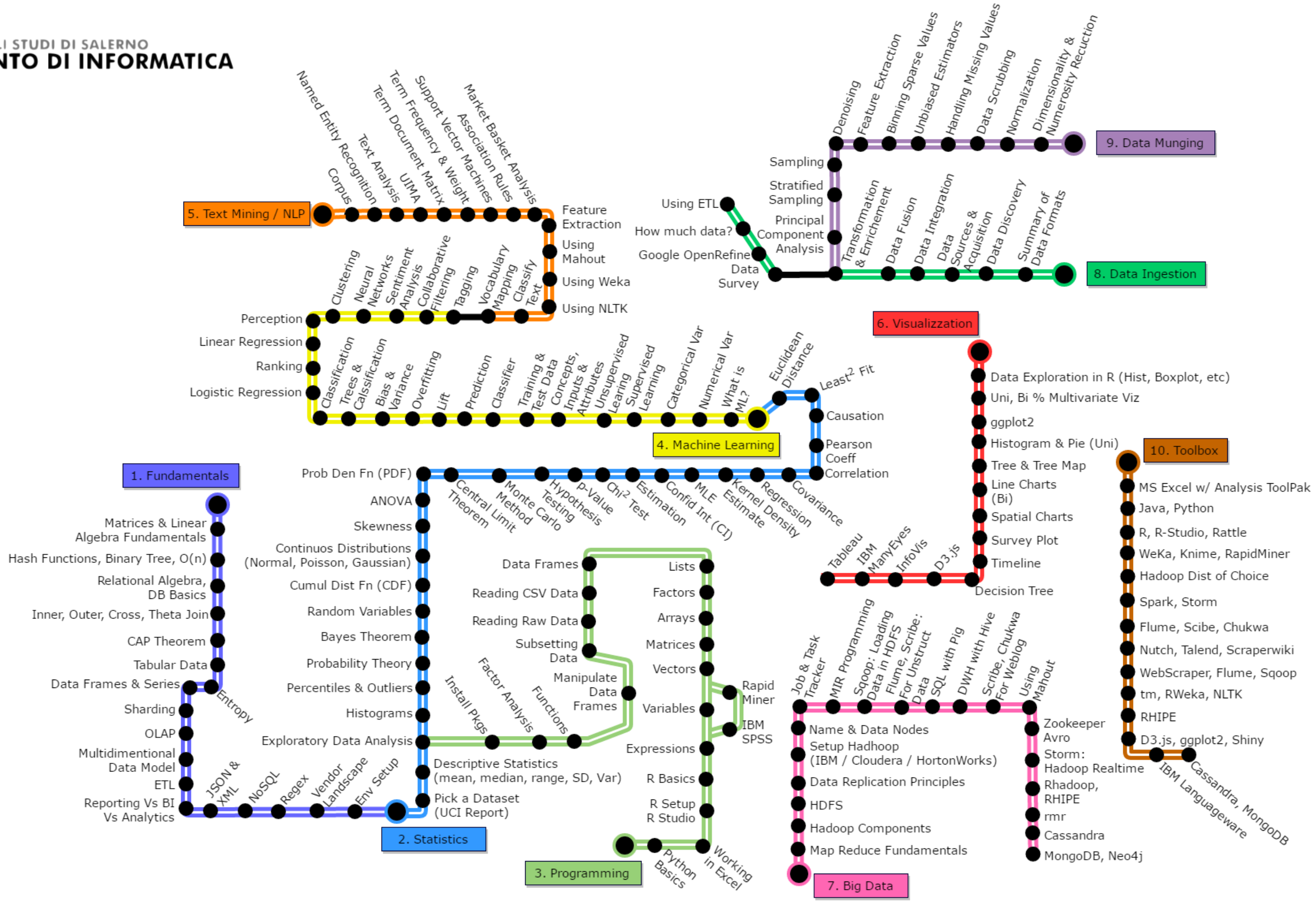
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science e Machine Learning

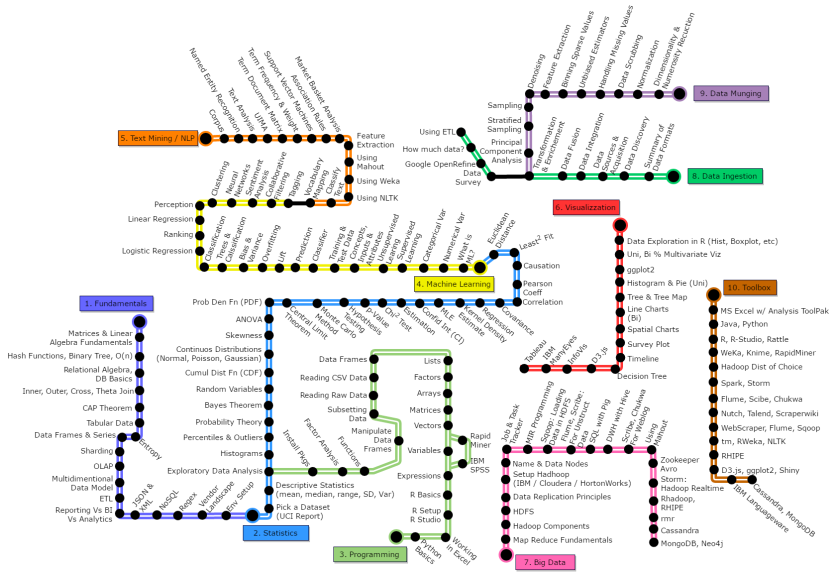
Introduction to Recurrent Neural Networks

Aurelien Geron: «Hands on Machine Learning with Scikit Learn and TensorFlow, O'Reilly ed.

Prof. Giuseppe Polese, aa 2024-25

Outline

- **General concepts**
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- Basic RNNs in TensorFlow
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



General concepts

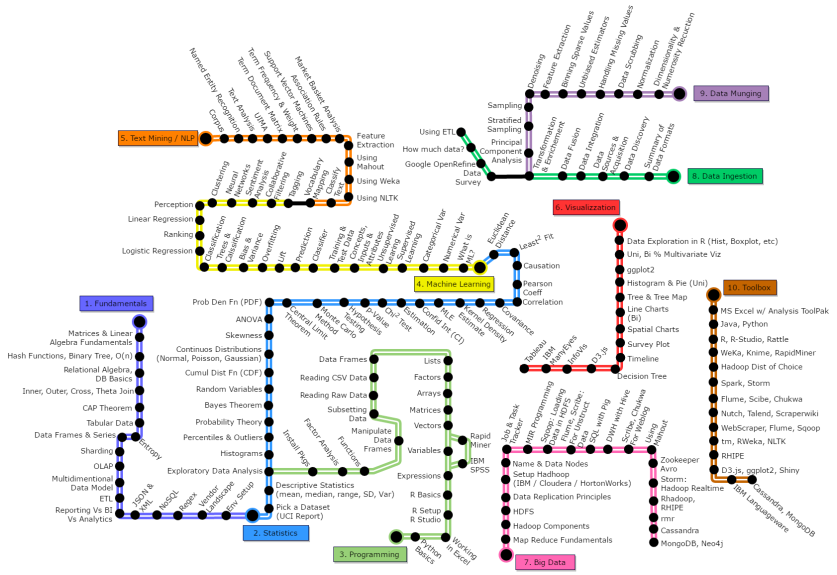
- Recurrent neural networks (RNN) is a class of nets that can predict the future
- They can analyze time series data such as stock prices. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents
- As opposed to the NN seen so far, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs
  - They can take sentences, documents, or audio samples as input, making them useful for NLP systems such as automatic translation, speech-to-text, or sentiment analysis

RNNs for creative predictions

- RNNs can predict the most likely next notes in a melody, so we can randomly play one of them.
- Iterating such a process the RNN can compose a melody such as the one of Google’s Magenta project.
- RNNs can generate sentences, image captions, and much more.
- Their main problem is the vanishing/exploding gradient, which is tackled through LSTM and GRU cells.

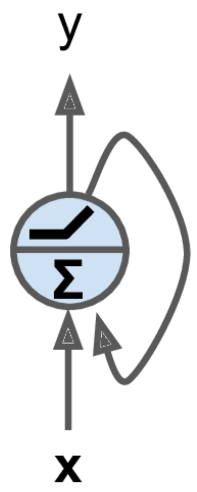
Outline

- General concepts
- **Recurrent neurons**
  - Memory cells
  - Input and output sequences
- Basic RNNs in TensorFlow
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



Recurrent Neurons

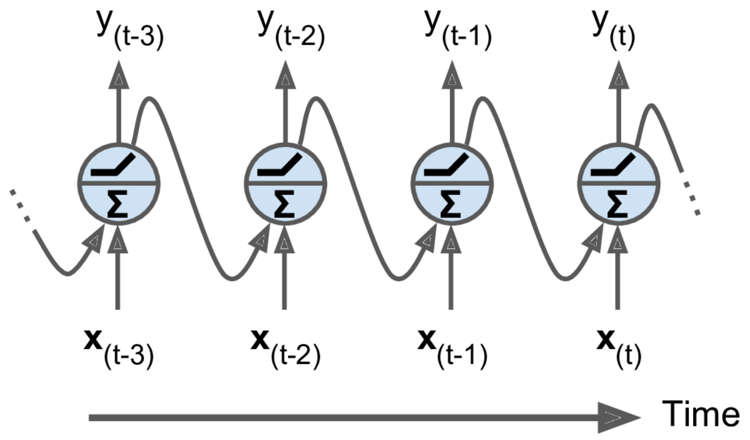
- A RNN looks like a feedforward NN, except it also has connections pointing backward.
- The simplest possible RNN is composed of just one neuron receiving inputs, producing an output, and sending this back to itself

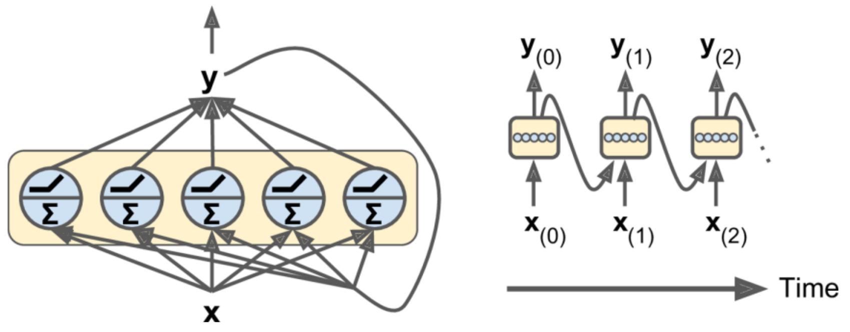


- At each time step t (*frame*), this recurrent neuron receives the inputs $x_{(t)}$ and the output from the previous time step, $y_{(t-1)}$.

Unrolling the RNN through time

- We can represent the previous tiny RNN against the time axis
  - This is called unrolling the network through time
- The figure below represents a layer of recurrent neurons
  - At each time step t, every neuron receives both the input vector $x_{(t)}$ and the output vector from the previous time step $y_{(t-1)}$
  - Inputs and outputs are vectors (with a single neuron, the output was a scalar).
  - Each recurrent neuron has two sets of weights: one for the inputs $x_{(t)}$ and the other for the outputs of the previous time step, $y_{(t-1)}$.





Output of a recurrent layer

- Let $\mathbf{w}_x$ and $\mathbf{w}_y$ represent input and output weight vectors.
- Output of a recurrent layer for a single instance:

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_y^T \cdot \mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

  - b is the bias and ϕ the activation function
- Outputs of a layer of recurrent neurons for all instances:

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \mathbf{W}_y + \mathbf{b}\right) =$$

$$= \phi\left[\mathbf{X}_{(t)} \cdot \mathbf{Y}_{(t-1)}\right] \cdot \mathbf{W} + \mathbf{b} \quad \text{with } \mathbf{W} = \begin{matrix}\mathbf{W}_x \\ \mathbf{W}_y\end{matrix}$$

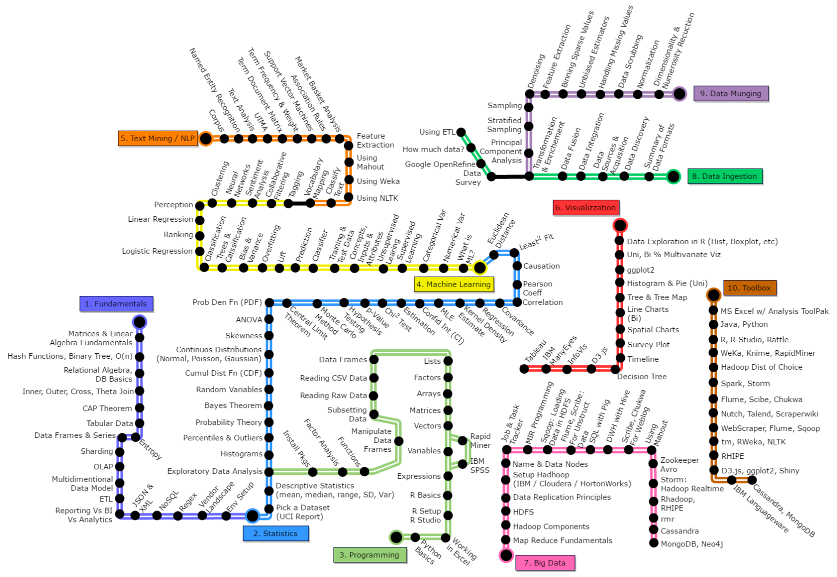
  - $\mathbf{X}_{(t)} \cdot \mathbf{Y}_{(t-1)}$ is the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$

Temporal steps of an RNN

- Notice that $\mathbf{Y}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\mathbf{Y}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\mathbf{Y}_{(t-3)}$, and so on.
- This makes $\mathbf{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}$, $\mathbf{X}_{(1)}$, …, $\mathbf{X}_{(t)}$).
- At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Outline

- General concepts
- **Recurrent neurons**
  - ***Memory cells***
  - Input and output sequences
- Basic RNNs in TensorFlow
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



Memory cells

- Since the output of a recurrent neuron at time step t is a function of all the inputs from previous steps, it has a form of memory.
- A part of a neural network that preserves some state across time steps is called a memory cell (or a cell).
- A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell (there are more complex and powerful types of cells).

Memory cells (2)

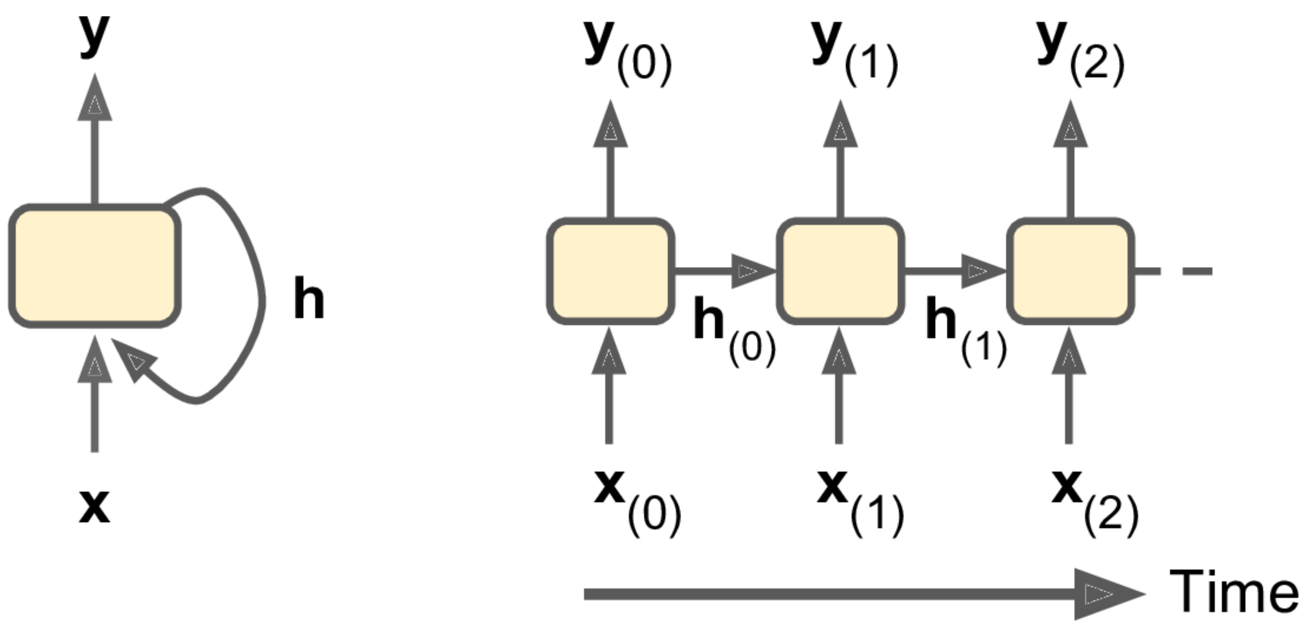
- A cell's state at time step t, denoted $h_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step:

$$\mathbf{h_{(t)} = f(h_{(t-1)}, x_{(t)})}$$

- Its output at time step t, denoted $y_{(t)}$, is also a function of the previous state and the current inputs.
- In the case of the basic cells output is simply equal to the state, but in more complex cells this is not always the case (see next slide)

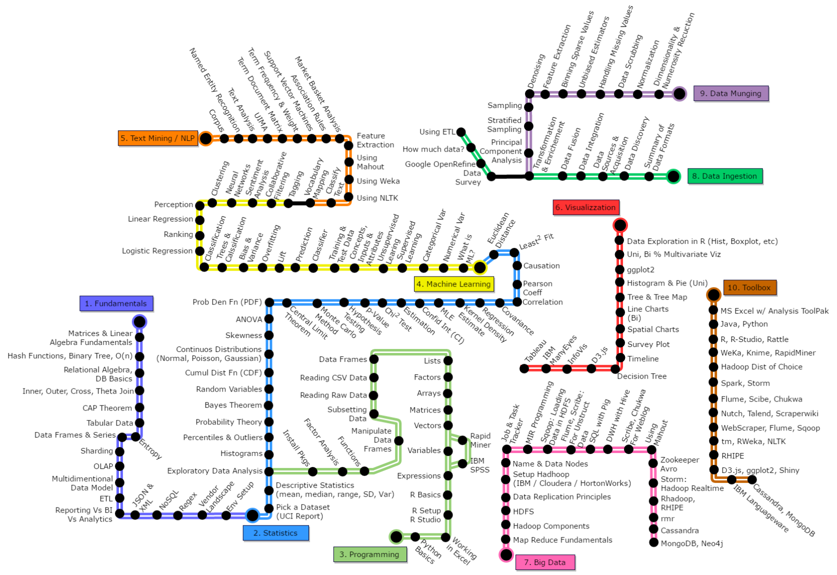
Cell's hidden state vs output

- A cell's hidden state and its output may be different:



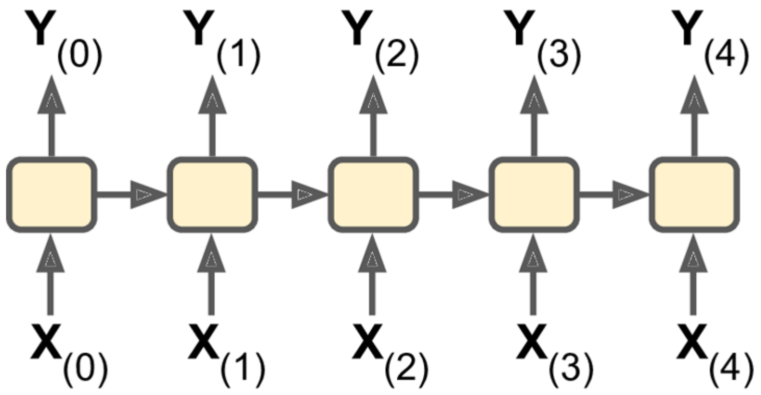
Outline

- General concepts
- **Recurrent neurons**
  - Memory cells
  - ***Input and output sequences***
- Basic RNNs in TensorFlow
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



Input and Output Sequences

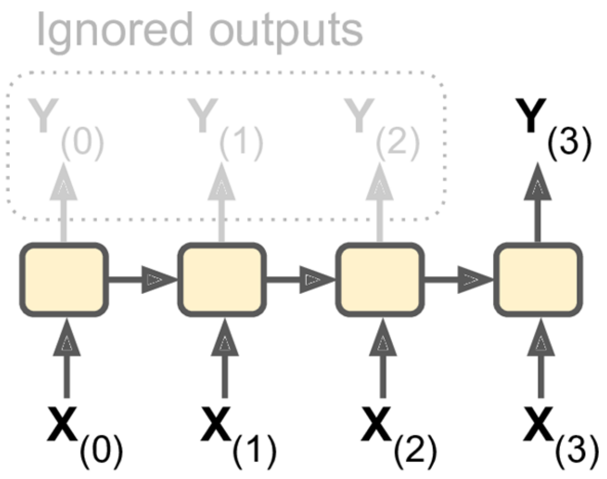
- A RNN can simultaneously take a sequence of inputs and produce a sequence of outputs:



  - This type of network is useful for predicting time series, such as stock prices: feed prices of the last N days, and the RNN outputs the prices from N – 1 days ago to tomorrow.

Sequence to vector RNN

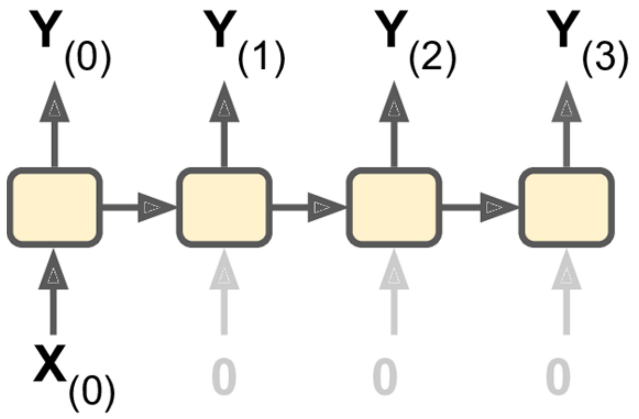
- The RNN can also take a sequence of inputs, and ignore all outputs except the last one:



  - For example, the input could be a sequence of words corresponding to a movie review, and the RNN would output a sentiment score.

Vector to sequence RNN

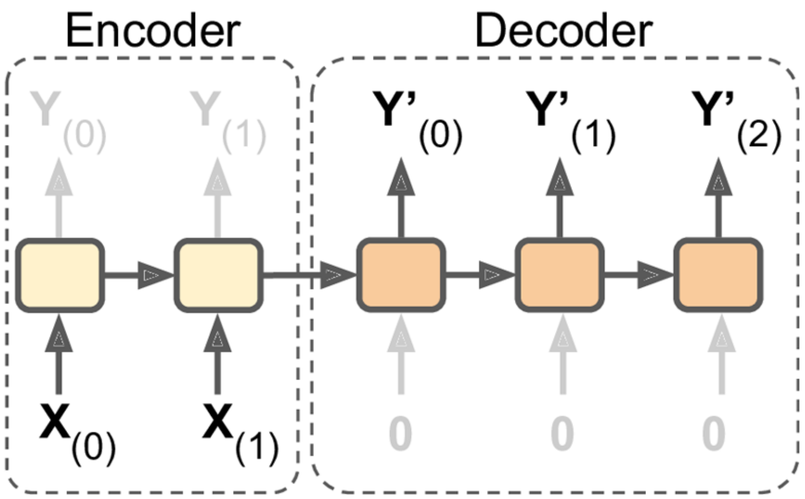
- The RNN can also take a single input at first time step (zeros for other time steps), and output a sequence:



  - For example, the input could be an image, and the output its caption.

Encoder-Decoder RNN

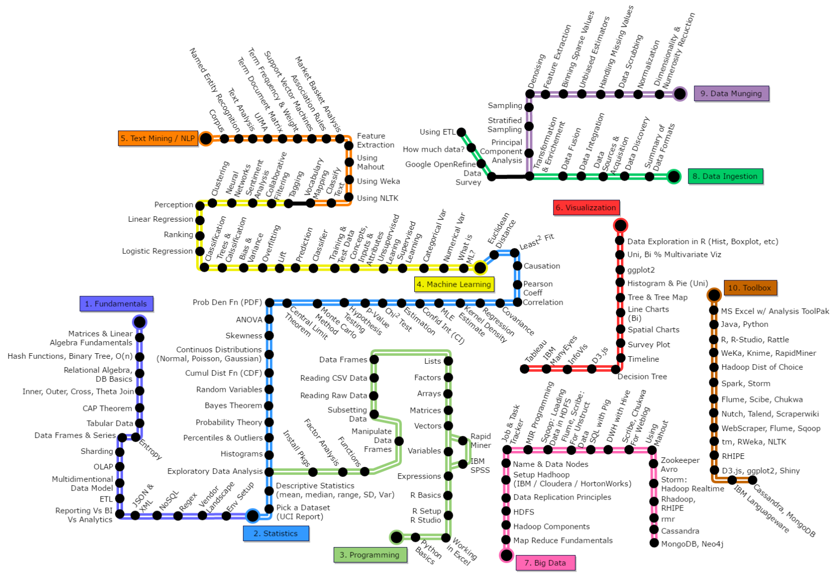
- We could also have a sequence-to-vector RNN, called an *encoder*, followed by a vector-to-sequence RNN, called a *decoder*:
  - For example, the input could be a sentence in a language, the encoder would convert it into a vector, which the decoder would convert into a sentence in another language.



  - This two-step model, called an Encoder–Decoder, works better than directly translating with a single sequence-to-sequence RNN, since the last words of a sentence can affect the meaning of the first words, so better wait the whole sentence before translating it.

Outline

- General concepts
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- **Basic RNNs in TensorFlow**
  - ***Basic RNNs in TensorFlow***
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



Basic RNN in TensorFlow

- Let’s first create a simple RNN model without TensorFlow, composed of a layer of 5 recurrent neurons, using the tanh activation function.
- And let’s assume that the RNN runs over only two time steps, taking input vectors of size 3 at each time step.
- This code unrolles such RNN through 2 time steps:

```
n_inputs = 3
n_neurons = 5
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))
Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)
init = tf.global_variables_initializer()
```

Running the Basic RNN

- To run it we need to feed inputs at both time steps

```
import numpy as np
# Mini-batch: instance 0,instance 1,instance 2,instance 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1
with tf.Session() as sess:
init.run()
Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

- This mini-batch contains 4 instances, each with an input sequence composed of exactly two inputs.
- At the end, Y0_val and Y1_val contain the outputs of the RNN at both time steps for all neurons and all instances in the mini-batch.

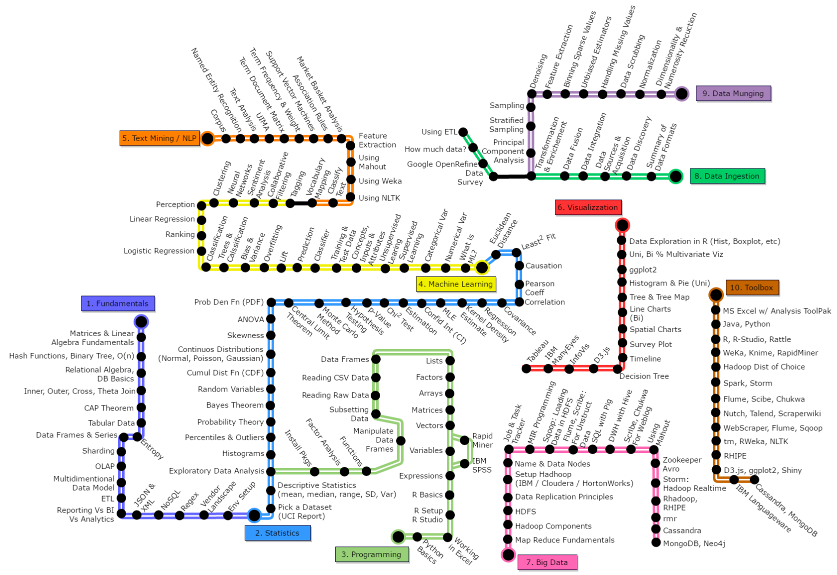
Basic RNN's output

```
>>> print(Y0_val) # output at t = 0
[[-0.0664006 0.96257669 0.68105787 0.70918542 -0.89821595] # instance 0
[ 0.9977755 -0.71978885 -0.99657625 0.9673925 -0.99989718] # instance 1
[ 0.99999774 -0.99898815 -0.99999893 0.99677622 -0.99999988] # instance 2
[ 1. -1. -1. -0.99818915 0.99950868]] # instance 3
>>> print(Y1_val) # output at t = 1
[[ 1. -1. -1. 0.40200216 -1. ] # instance 0
[-0.12210433 0.62805319 0.96718419 -0.99371207 -0.25839335] # instance 1
[ 0.99999827 -0.9999994 -0.9999975 -0.85943311 -0.9999879 ] # instance 2
[ 0.99928284 -0.99999815 -0.99990582 0.98579615 -0.92205751]] # instance 3
```

- Quite simple, but what if we want to run a RNN over 100 time steps, the graph will be pretty big.
- Let's see how to create the same model using TensorFlow's RNN operations.

Outline

- General concepts
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- **Basic RNNs in TensorFlow**
  - Basic RNNs in TensorFlow
  - ***Static Unrolling Through Time***
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



Static unrolling through time

- The following code uses TensorFlow’s RNN operations to create the same model as the previous one:

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1], dtype=tf.float32)
Y0, Y1 = output_seqs
```

  - The static_rnn() function creates an unrolled RNN network by chaining cells.
  - BasicRNNCell is like a factory creating copies of the cell to build the unrolled RNN (one for each time step).
  - Static_rnn() is invoked on the cell factory, the input tensors, and the data type of the inputs (to create the state matrix).

Static_rnn function

- static_rnn() calls the cell factory's __call__() function once per input, creating 2 copies of the cell (each with a layer of 5 recurrent neurons), with shared weights and bias terms, chaining them as we did earlier.
- It returns two objects:
  - A Python list with the output tensors for each time step
  - A tensor containing the final states of the network
  - With basic cells the final state is equal to the last output.
- With 100 time steps we should define 100 input placeholders and 100 output tensors. At run time we should feed each placeholder and manipulate 100 outputs. Let's simplify this.

Simplified TensorFlow code

- The following code below builds the same RNN, but it takes one input placeholder of shape [None, n_steps, n_inputs], the 1st dimension is the mini-batch size:

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons) #unchanged, as next line
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs, dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

  - X_seqs is a Python list of n_steps tensors of shape [None, n_inputs]
  - It first swaps the first 2 dimensions using the transpose() function, so the time steps become the 1st one. Then, it extracts a list of tensors along the 1st dimension (1 tensor per time step) using the unstack() function.
  - Finally, output tensors are merged using the stack() function, swapping first 2 dimensions, yielding a final tensor [None, n_steps, n_neurons].

Running the simplified TF code

- We can run the RNN by feeding it a single tensor containing all the minibatch sequences:

```
X_batch = np.array([
            # t = 0 t = 1
            [[0, 1, 2], [9, 8, 7]], # instance 0
            [[3, 4, 5], [0, 0, 0]], # instance 1
            [[6, 7, 8], [6, 5, 4]], # instance 2
            [[9, 0, 1], [3, 2, 1]], # instance 3
])

with tf.Session() as sess:
init.run()
outputs_val = outputs.eval(feed_dict={X: X_batch})
```

Running the simplified TF code

- We get a single outputs_val tensor for all instances, all time steps, and all neurons:

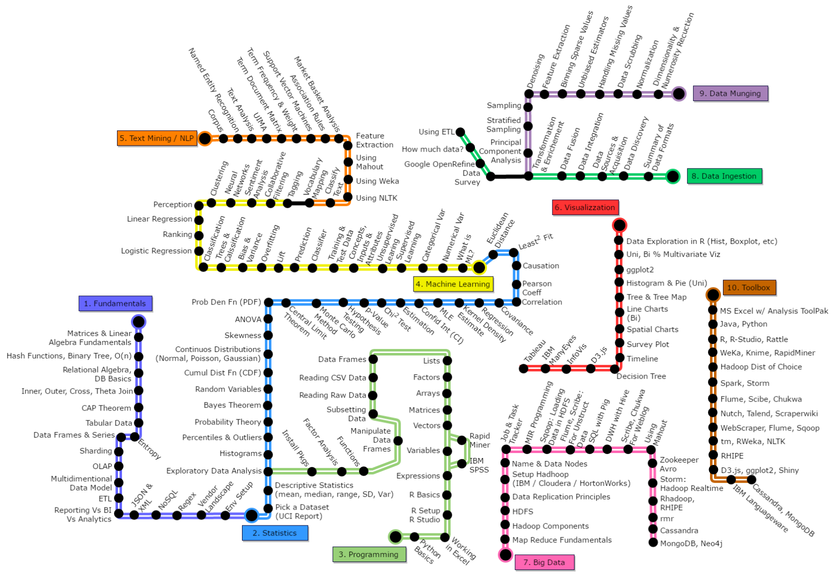
```
>>> print(outputs_val)

[[[-0.91279727 0.83698678 -0.89277941 0.80308062 -0.5283336 ]
  [-1. 1. -0.99794829 0.99985468 -0.99273592]]

 [[-0.99994391 0.99951613 -0.9946925 0.99030769 -0.94413054]
  [ 0.48733309 0.93389565 -0.31362072 0.88573611 0.2424476 ]]

 [[-1. 0.99999875 -0.99975014 0.99956584 -0.99466234]
  [-0.99994856 0.99999434 -0.96058172 0.99784708 -0.9099462 ]]

 [[-0.95972425 0.99951482 0.96938795 -0.969908 -0.67668229]
  [-0.84596014 0.96288228 0.96856463 -0.14777924 -0.9119423 ]]]
```

Problems with simplified TF code

- This approach still builds a graph with 1 cell per time step. With 100 time steps, the graph is ugly.
- It is like writing a program without using loops (e.g., Y0=f(0, X0); Y1=f(Y0, X1);...; Y100=f(Y99, X100)).
- We may even get out-of-memory (OOM) errors during backpropagation (especially with the limited memory of GPU cards), since it must store all tensor values during the forward pass so it can use them to compute gradients during the reverse pass.
- There is a better solution: the dynamic_rnn() function.

Outline

- General concepts
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- **Basic RNNs in TensorFlow**
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - ***Dynamic Unrolling Through Time***
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- Training RNNs



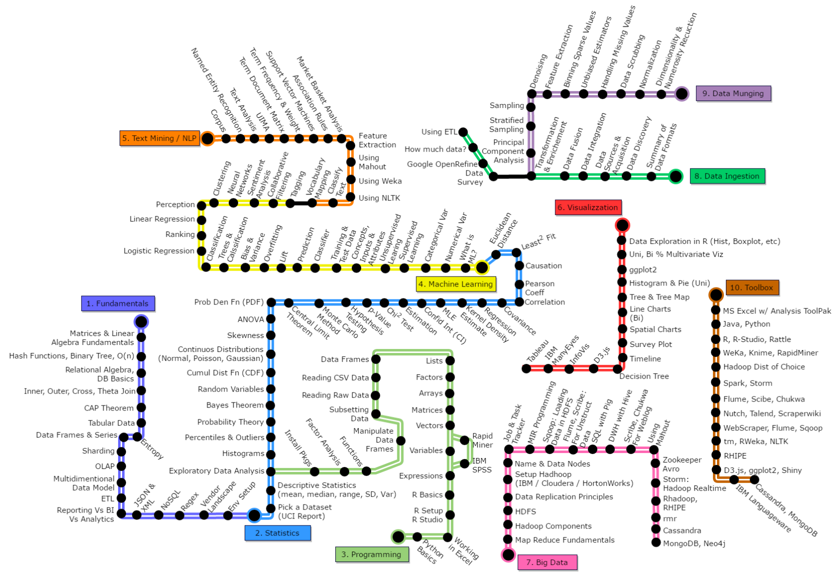
Dynamic unrolling through time

- dynamic_rnn() uses a while_loop() to run over the cell the appropriate number of times
- Setting swap_memory=True swaps GPU's memory to CPU's during backpropagation to avoid OOM errors.
- It can accept a single tensor for all inputs at each time step, and output a single tensor for all outputs at each time step.
- No need to stack, unstack, or transpose:

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

Outline

- General concepts
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- **Basic RNNs in TensorFlow**
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - ***Handling Variable Length Input Sequences***
  - Handling Variable-Length Output Sequences
- Training RNNs



Variable Length Input Sequences

- So far we have used only fixed-size input sequences (all exactly two steps long).
- What if the input sequences have variable lengths (e.g., like sentences)?
- We should set the sequence_length argument when calling dynamic_rnn() (or static_rnn()) function
  - it must be a 1D tensor indicating the length of the input sequence for each instance:

```
seq_length = tf.placeholder(tf.int32, [None])
[...]
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
sequence_length=seq_length)
```

Example of variable input

- Suppose the 2d input sequence contains only one input instead of two.
- It must be padded with a zero vector to fit in the input tensor X (the input tensor’s 2d dimension is the size of the longest sequence—i.e., 2).

```
X_batch = np.array([
# step 0 step 1
[[0, 1, 2], [9, 8, 7]], # instance 0
[[3, 4, 5], [0, 0, 0]], # instance 1 (padded with a zero vector)
[[6, 7, 8], [6, 5, 4]], # instance 2
[[9, 0, 1], [3, 2, 1]], # instance 3
])
seq_length_batch = np.array([2, 1, 2, 2])
```

Example of variable input (2)

- We need to feed values for both placeholders X and seq_length:

```
with tf.Session() as sess:
init.run()
outputs_val, states_val = sess.run(
[outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

  - The RNN outputs zero vectors for every time step past the input sequence length

RNN Output

- Look at the 2d instance's output for the 2d time step):

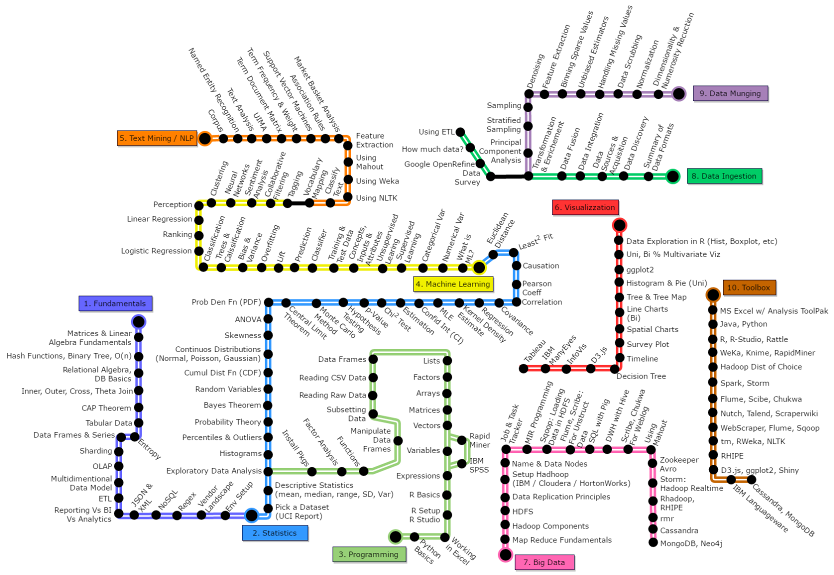
```
>>> print(outputs_val)
[[[-0.68579948 -0.25901747 -0.80249101 -0.18141513 -0.37491536]
[-0.99996698 -0.94501185 0.98072106 -0.9689762 0.99966913]] # final state
[[-0.99099374 -0.64768541 -0.67801034 -0.7415446 0.7719509 ] # final state
[ 0. 0. 0. 0. 0. ]] # zero vector
[[-0.99978048 -0.85583007 -0.49696958 -0.93838578 0.98505187]
[-0.99951065 -0.89148796 0.94170523 -0.38407657 0.97499216]] # final state
[[-0.02052618 -0.94588047 0.99935204 0.37283331 0.9998163 ]
[-0.91052347 0.05769409 0.47446665 -0.44611037 0.89394671]]] # final state
```

- The states tensor contains the final state of each cell (excluding the zero vectors):

```
>>> print(states_val)
[[-0.99996698 -0.94501185 0.98072106 -0.9689762 0.99966913] # t = 1
[-0.99099374 -0.64768541 -0.67801034 -0.7415446 0.7719509 ] # t = 0 !!!
[-0.99951065 -0.89148796 0.94170523 -0.38407657 0.97499216] # t = 1
[-0.91052347 0.05769409 0.47446665 -0.44611037 0.89394671]] # t = 1
```

Outline

- General concepts
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- **Basic RNNs in TensorFlow**
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - ***Handling Variable-Length Output Sequences***
- Training RNNs

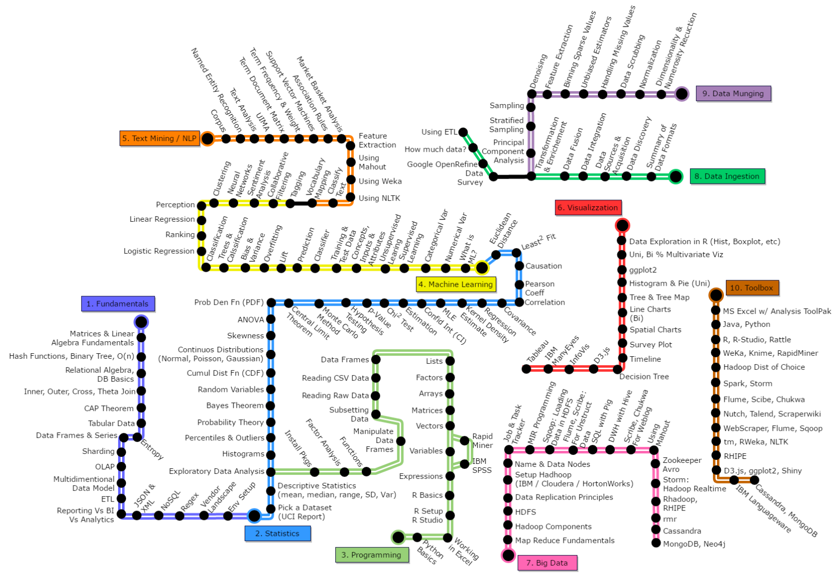


Variable Length Output Sequences

- What if the output sequences have variable lengths?
- If you know in advance what length each sequence will have, then you can set the sequence_length parameter
- In general this will not be possible: for example, the length of a translated sentence is generally different from the length of the input sentence.
- A common solution is to define a special output called an end-of-sequence token (EOS token). Any output past the EOS should be ignored.
- Okay, now we know how to build an RNN network. But how do you train it?

Outline

- General concepts
- Recurrent neurons
  - Memory cells
  - Input and output sequences
- Basic RNNs in TensorFlow
  - Basic RNNs in TensorFlow
  - Static Unrolling Through Time
  - Dynamic Unrolling Through Time
  - Handling Variable Length Input Sequences
  - Handling Variable-Length Output Sequences
- **Training RNNs**



Training RNN

- To train an RNN, the trick is to unroll it through time and then simply use regular backpropagation.
- This strategy is called backpropagation through time.

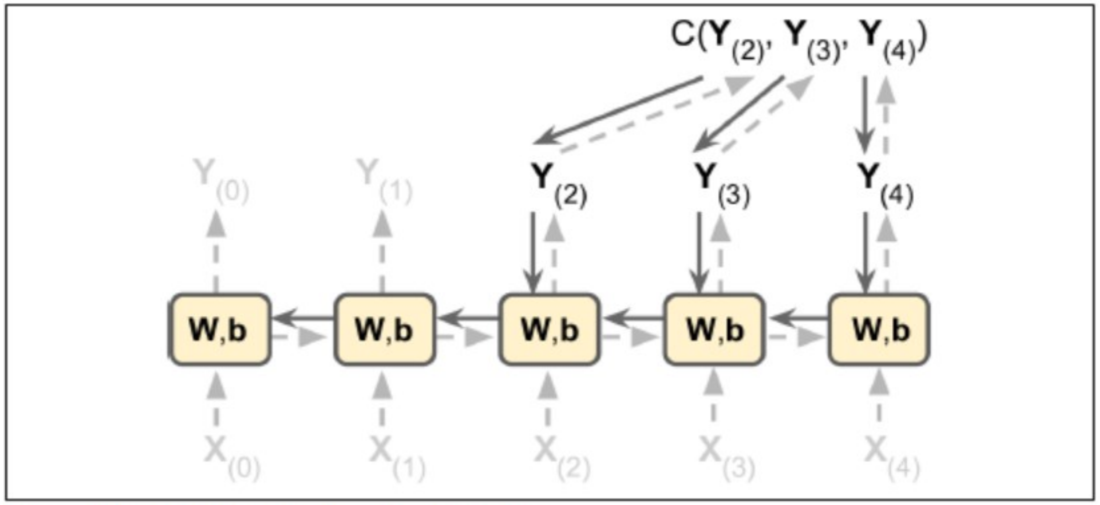


Figure 14-5. Backpropagation through time

Training RNN

- Just like in regular backpropagation, there is a first forward pass through the unrolled network
- then the output sequence is evaluated using a cost function $C(Y_{tmin}, Y_{tmin+1}, \cdots, Y_{tmax})$
- t_{min} and t_{max} are the first and last output time steps
- the gradients of the cost function are propagated backward through the unrolled network
- Finally, the model parameters are updated using the gradients computed during Back Propagation Through Time.

Back Propagation Through Time

- Gradients flow backward through all the outputs used by the cost function, not just the final output
- Es. In the Figure the cost function is computed using the last three outputs of the network, Y(2), Y(3), and Y(4), but not Y(0) and Y(1).
- since the same parameters W and b are used at each time step, BPTT will sum over all time steps.

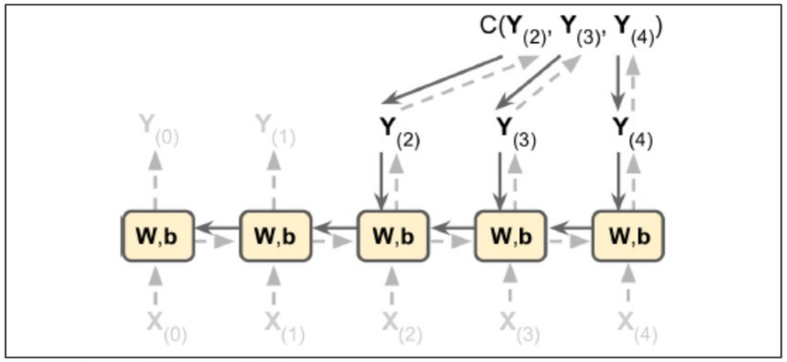


Figure 14-5. Backpropagation through time

Training to predict Time Series

- RNN can handle time series, such as stock prices, air temperature, brain wave patterns, and so on.
- The goal is to train an RNN to predict the next value.
- Each training instance is a randomly selected sequence of 20 consecutive values from the series.
- The target sequence is the same, but it is shifted by one time step the future.

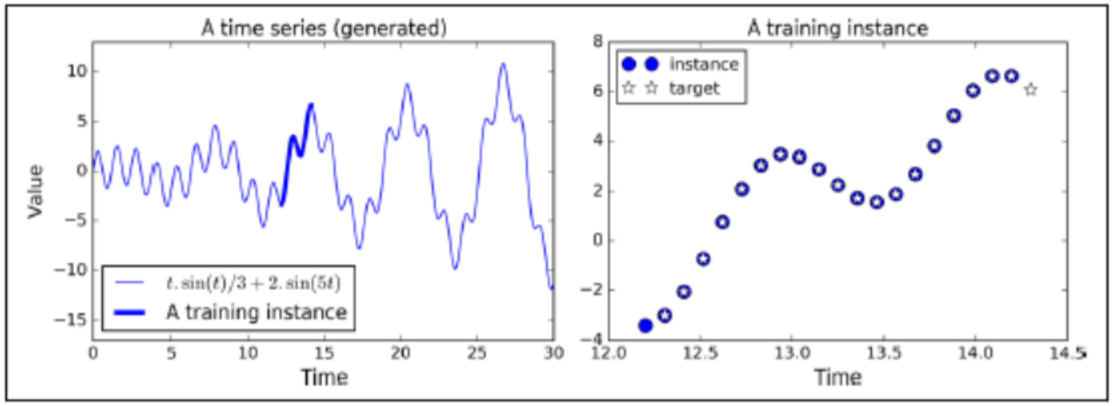


Figure 14-7. Time series (left), and a training instance from that series (right)

RNN for Time Series

- Let’s create a RNN with 100 recurrent neurons and unroll it over 20 time steps since each training instance will be 20 inputs long.
- Each input will contain only one feature (the value at that time).
- The targets are also sequences of 20 inputs, each containing a single value.

Python code RNN for Time Series

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

- At each time step we have an output vector of size 100. But we want a single output at each time step.
- The simplest solution is to wrap the cell in an OutputProjectionWrapper.

Wrapping cells

- A cell wrapper acts like a normal cell, proxying every method call to an underlying cell, but it also adds some functionality.
- The OutputProjectionWrapper adds a fully connected layer of linear neurons (i.e., without activation function) on top of each output (but it does not affect the cell state).
- All these fully connected layers share the same (trainable) weights and bias terms.
- The resulting RNN is:

RNN cells with output projections

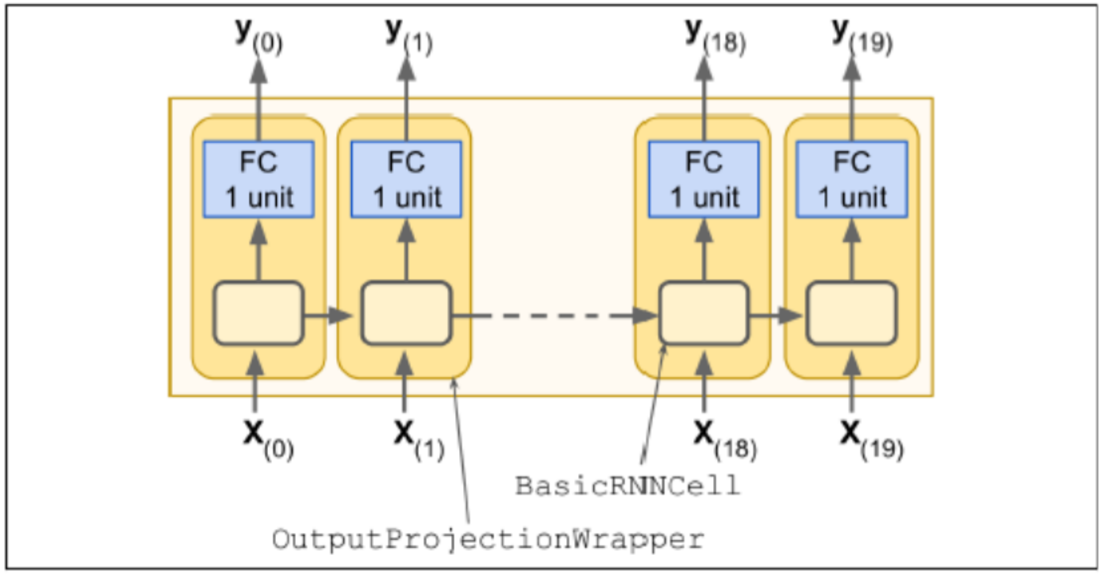


Figure 14-8. RNN cells using output projections

- Let’s tweak the preceding code by wrapping the BasicRNNCell into an OutputProjectionWrapper:

cell = tf.contrib.rnn.OutputProjectionWrapper(

tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),

output_size=n_outputs)

Cost Function

- We will use the Mean Squared Error (MSE), as we did in previous regression tasks:

```
learning_rate = 0.001
loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
init = tf.global_variables_initializer()
```

Now on to the execution phase:

```
n_iterations = 1500
batch_size = 50
with tf.Session() as sess:
init.run()
for iteration in range(n_iterations):
X_batch, y_batch = [...] # fetch the next training batch
sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
if iteration % 100 == 0:
mse = loss.eval(feed_dict={X: X_batch, y: y_batch
```

Output

- The program output is

```
0 MSE: 13.6543
100 MSE: 0.538476
200 MSE: 0.168532
300 MSE: 0.0879579
400 MSE: 0.0633425
[...]
```

Once the model is trained, you can make predictions:

```
X_new = [...] # New sequences
y_pred = sess.run(outputs, feed_dict={X: X_new})
```