TensorFlow, Part B

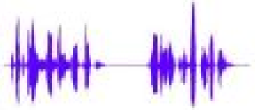
MSAIL 2017

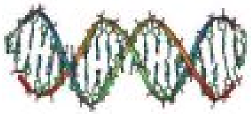
Ankit Goila, Uriah Israel, Sam Tenka

Working with Sequences

→ Sequence learning is the study of machine learning algorithms designed for sequential data [1].

→ Language model is one of the most interesting topics that use sequence labeling.





An example: Language Translation

Understand the meaning of each word, and the relationship between words. **Context** is important!

Input: one sentence in German

input = "Ich will stark Steuern senken"

Output: one sentence in English

output = "I want to cut taxes bigly" (big league?)

→ Much of the information contained in language is in the sequence of words.

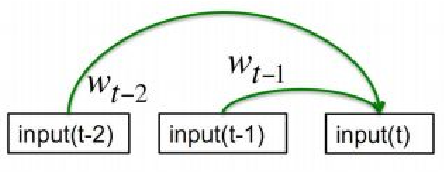
Ways to deal with Sequential data

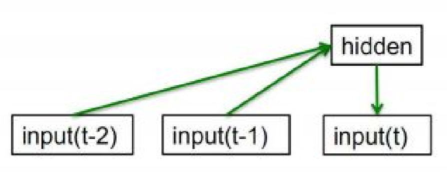
→ **Autoregressive models**

→ Predict the next term in a sequence from a fixed number of previous terms using delay taps.

→ **Feed-forward neural nets**

→ These generalize autoregressive models by using one or more layers of non-linear hidden units.





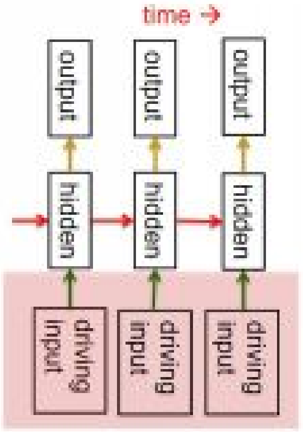
Memoryless models: limited word-memory window;
hidden state cannot be used efficiently.

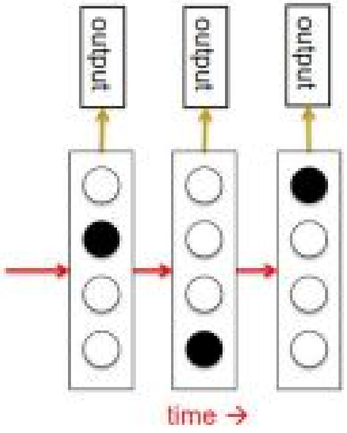
Ways to deal with Sequential data

→ **Hidden Markov Models**

→ Have a discrete one-of-N hidden state.
→ Transitions between states are stochastic and controlled by a transition matrix.
→ The outputs produced by a state are stochastic.

Memoryful models:
time-cost to infer the hidden state distribution.





Why RNNs?

→ Humans don't start their thinking from scratch every second. Our thoughts have persistence.

→ Traditional neural nets assume independent inputs (and outputs).

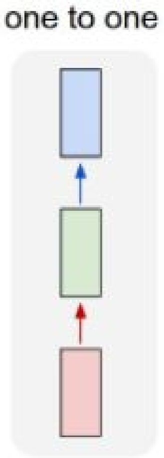
→ Bad idea for predicting next word in a sentence.

→ **Limitations** of feed-forward neural nets (and also Convolutional Networks):

1. **API is too constrained:** accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes).

   ~ Not good at length-varying inputs and outputs.

2. Input-Output mapping using fixed amount of computational steps (e.g. the number of layers in the model).

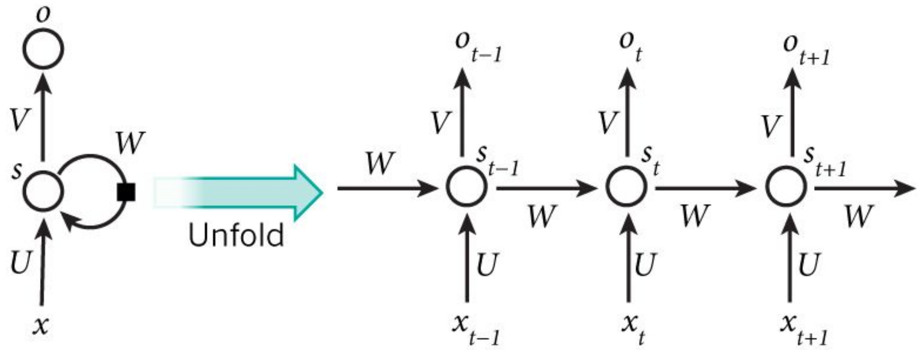


Fixed-sized input to fixed-sized output (e.g. image classification).

What are RNNs?

→ "Networks with loops in them", allowing information to persist.

→ *Recurrent*: perform the same task for every element of a sequence.
→ Output dependent on previous computations.

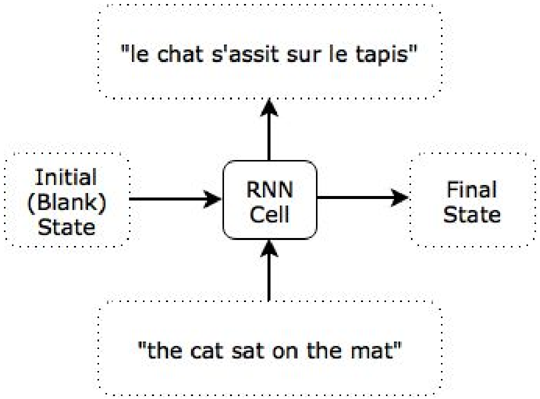


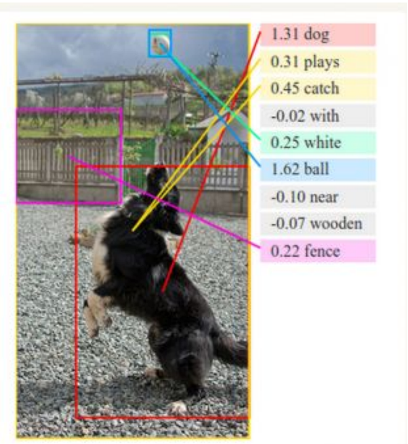
→ Update the hidden state (circle in the middle) in a **deterministic nonlinear** way.

→ **Distributed hidden state** that allows them to store a lot of information about the past efficiently.

RNN being *unrolled* (or unfolded in time) into a full network.

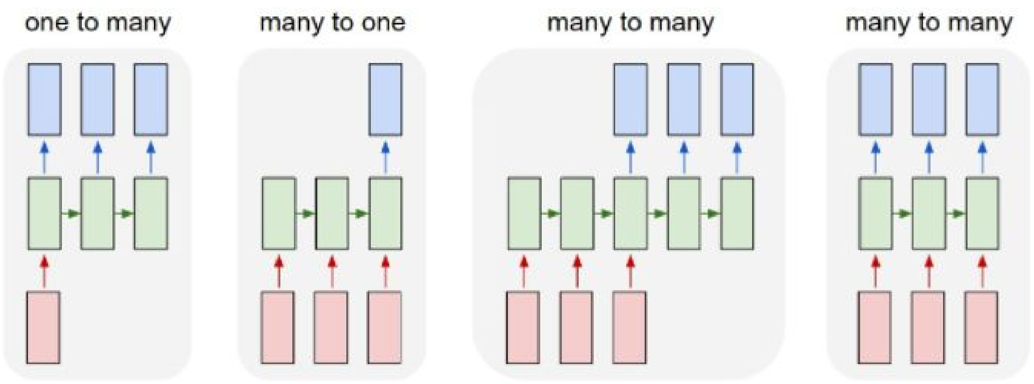
What can RNNs do?





1. Language Modeling and Generating Text
2. Machine Translation
3. Speech Recognition
4. Generating Image Descriptions (Image Captioning)
5. Compose Music!

And a lot more!



From left to right:

(1) Sequence output
(e.g. image captioning)

(2) Sequence input (e.g. sentiment analysis)

(3) Sequence input and sequence output
(e.g. Machine Translation)

(4) Synced sequence input and output
(e.g. video classification where we wish to label each frame of the video).

Exercise!

→ Time to run some code!

→ We will use vanilla RNNs to generate a script from **The Simpsons**.

What are vanilla RNNs?

Glad you asked! Details follow this exercise.

→ Read and run:

python vanilla-RNN.py (Train and save the model)
python generate_vanilla_sample.py (Generate sample script)

→ Try different number of RNN layers, and observe the difference in the generated script.

→ You can visualize the loss using Tensorboard (logged through cost scalar summary):

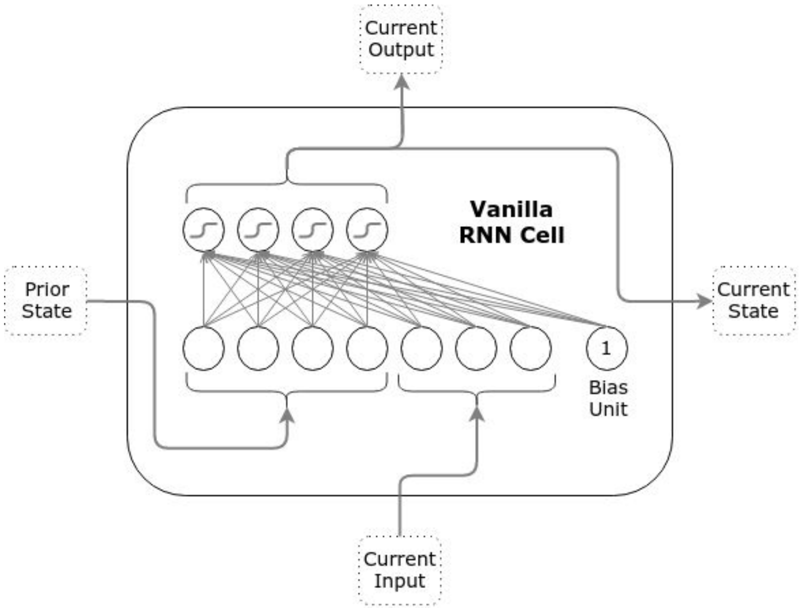
tensorboard --logdir ./logs_vanilla/1/

→ You can also see the script we used for training in **data/simpsons/**

Let’s talk about the Vanilla RNN

→ The most basic RNN cell (**Vanilla RNN**)

→ A single layer neural network



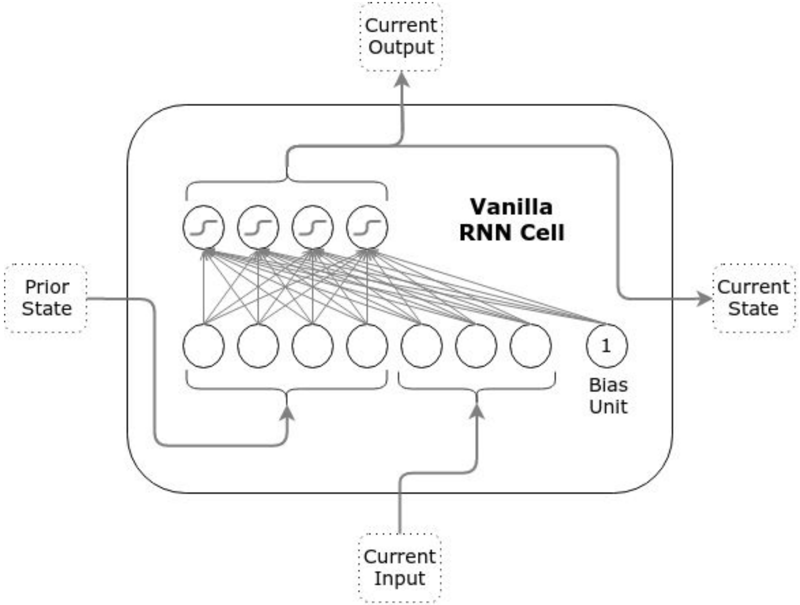
→ Output used as both the cell’s current (external) output and the current state:

→ **Prior State** vector is the same size as the **Current State** vector.

→ TensorFlow’s implementation:

tf.contrib.rnn.BasicRNNCell()

Vanilla RNN computations



→ Algebraic description of the vanilla RNN cell:

$$s_t = \phi(W s_{t-1} + U x_t + b)$$

where:

- ϕ is the activation function (e.g., sigmoid, tanh, ReLU),
- $s_t \in \mathbb{R}^n$ is the current state (and current output),
- $s_{t-1} \in \mathbb{R}^n$ is the prior state,
- $x_t \in \mathbb{R}^m$ is the current input,
- $W \in \mathbb{R}^{n \times n}$, $U \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^n$ are the weights and biases, and
- n and m are the state and input sizes.

Vanilla RNN computations: forward pass in Python

→ Written as a class, the RNN's API consists of a single step function:

rnn = RNN()
y = rnn.step(x)

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

```python
class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

→ Parameters are the three matrices:
W_hh, ***W_xh***, ***W_hy***

→ The hidden state **self.h** is initialized with the zero vector.

→ The **np.tanh** function squashes the activations to the range [-1, 1].

→ Stack models (deep) by calling **step()** multiple times.

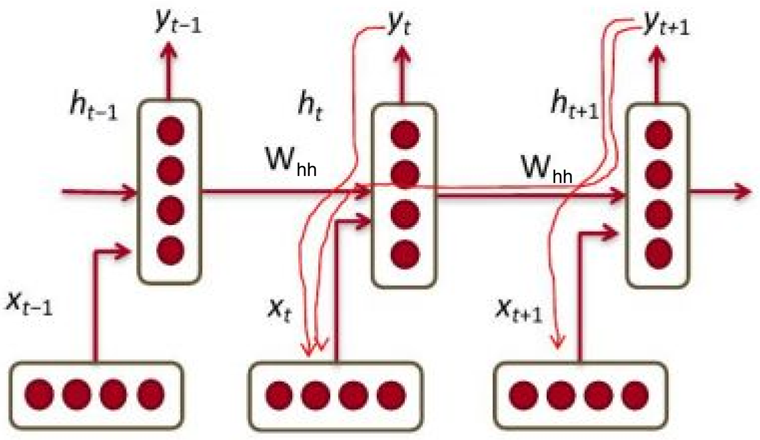
→ Backward pass same as in a traditional neural net (BPTT).

Vanishing and exploding gradients

→ Vanilla RNNs are great - theoretically!

→ Practical problem: Training vanilla RNNs with backprop is difficult
→ Vanishing and exploding sensitivity caused by repeated application of the same nonlinear func.

The output is a function of $\boldsymbol{W_{hh}}$ and $\boldsymbol{h_t}$, which itself is a function of $\boldsymbol{W_{hh}}$ and $\boldsymbol{h_{t-1}}$, and so on.



→ Repeated multiplications during backprop.

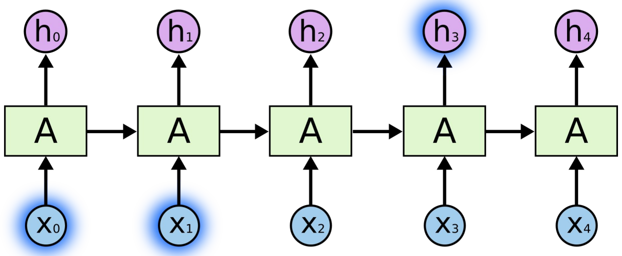
→ Makes it difficult for RNNs to learn long range interactions.

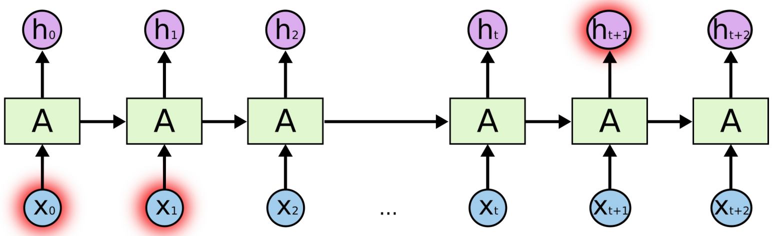
The problem of long-term dependencies

→ Sometimes, only need to look at recent information to perform a task.

→ Example: predict the last word in “The clouds are in the ***sky***”

→ Small look-up history sufficient, don’t need any further context.





→ **But**, there are cases where we need more context.

→ Example: predict the last word in “I grew up in France… I speak fluent ***French***.”

→ Need the context of France, from further back.

→ Entirely possible for the gap between the relevant information and the point where it is needed to become very large.

→ As this gap grows, Vanilla RNNs become unable to learn to connect the information. **Solution? LSTM !!**

Exercise!

→ To solve the problem of vanishing gradients, we use a more complicated cell called **Long Short-Term Memory** Cell or **LSTM**.

→ Next, we will use LSTM RNNs to generate a script from The Simpsons!

→ Read and run:

  python lstm-RNN.py (Train and save the model)
  python generate_lstm_sample.py (Generate sample script)

→ Try different number of RNN layers, and observe the difference in the generated script.

→ Can you see some difference between **vanilla** and **lstm** scripts?

  → Don't worry, the scripts, in general don't make sense. Trained on less than a megabyte of text.
  → To get good results, use a smaller vocabulary or get more data.

→ Observe the loss using Tensorboard (logged through cost scalar summary):

  tensorboard --logdir ./logs_lstm/1/

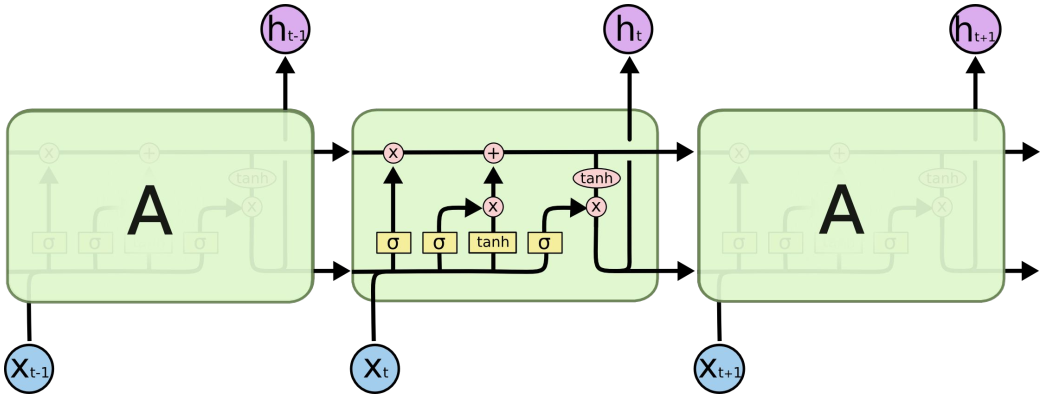
From Vanilla to LSTM

→ Conclusion from previous discussion:

→ For standard RNN architectures, the range of accessible context limited.

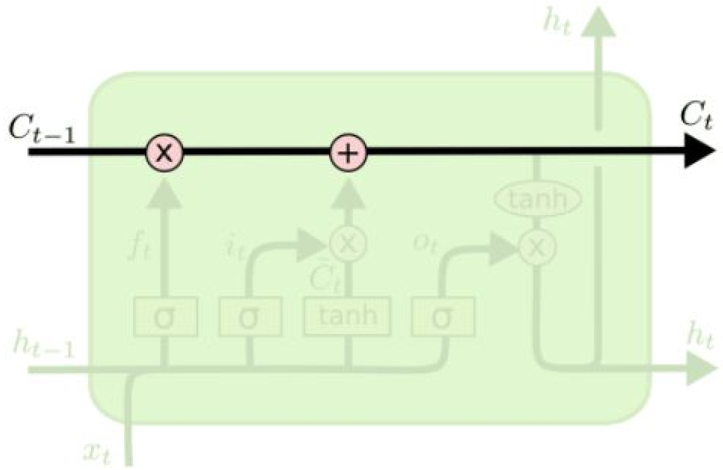
→ A given input's influence on the hidden layer and the network output, either decays or blows up exponentially as it cycles around the recurrent connections.

→ **LSTM (Long Short Term Memory)**: Most effective solution so far.
→ Explicitly designed to avoid the long-term dependency problem.



LSTM Architecture

→ **Cell State:** key component to the LSTMs - major upgrade over Vanilla RNNs.



→ Runs straight down the entire chain with minor linear interactions.

→ Convenient for information to just flow along unchanged.

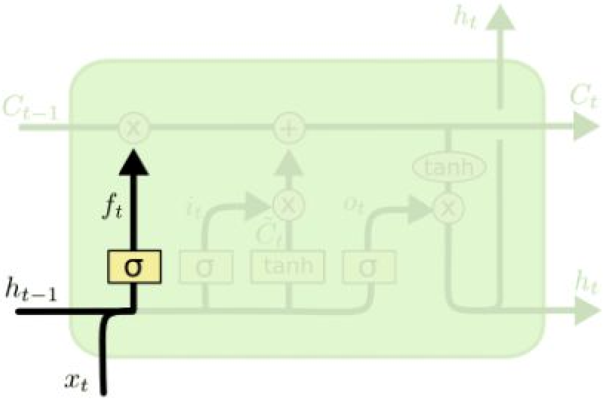
→ Element-wise operations function as **Gates** (information regulators)

→ TensorFlow's implementation:

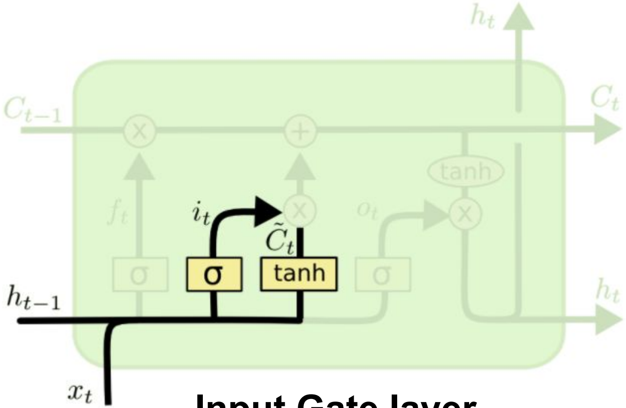
tf.contrib.rnn.BasicLSTMCell()
(more variants available)

LSTM Architecture

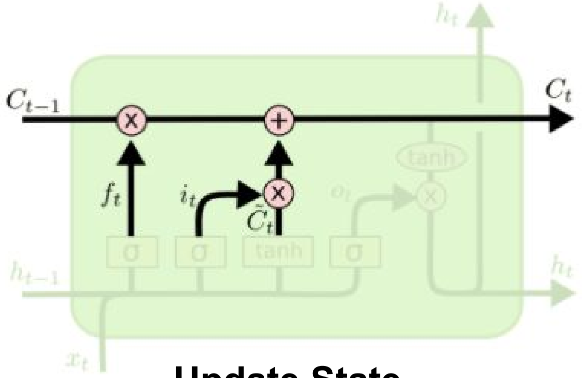
Four interacting layers:



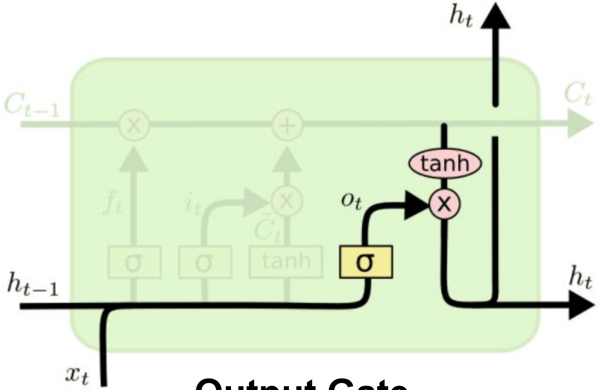
Forget Gate layer



Input Gate layer



Update State



Output Gate

Exercise!

→ We will now visualize word vectors as learned from a **Word2Vec** model using Vanilla RNN and LSTM-based RNN on the Simpsons data set.

→ **What are these?** Learned vector representations of words in our data set.

More like learning semantic relationships between words (man:king::woman:queen etc)

→ Read and run:
embedding-visualizer.ipynb (iPython Notebook)

→ We use T-SNE and Tensorboard to visualize embeddings.

For Tensorboard, run:

tensorboard --logdir ./<your model’s log dir>/1/ and head to the **Embeddings** tab.

Word Embeddings

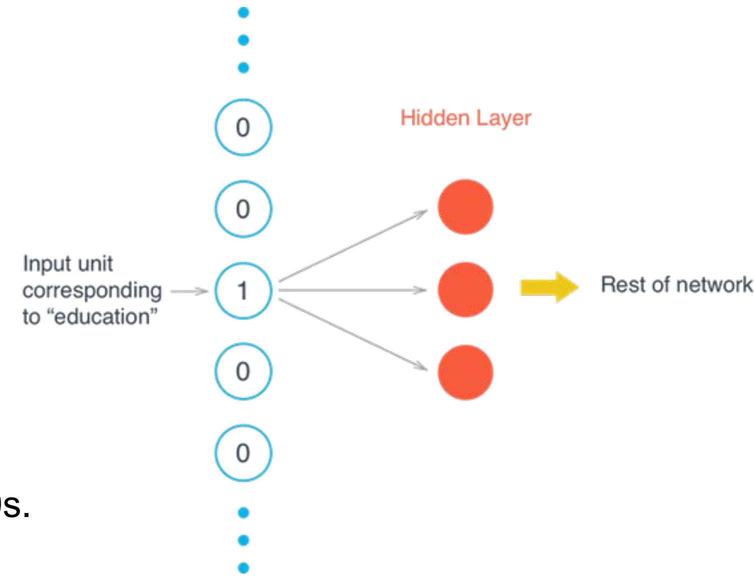
→ Consider **The Simpsons** data set:

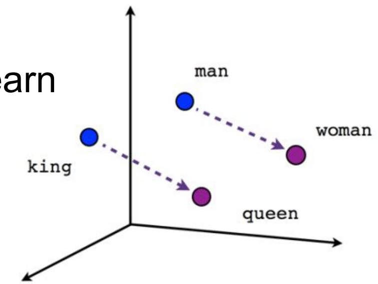
→ Dealing with ~10,000 words = ~10,000 classes to predict
→ Traditional one-hot encoding massively inefficient

→ One element: 1, rest ~10,000: 0
→ Most resulting matrix multiplications end up being 0s.

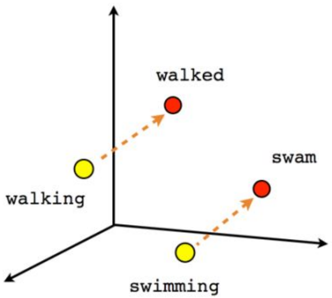
→ Use **embeddings** to represent data with huge number of classes more efficiently.

→ Use **Word2Vec** model to learn embeddings.

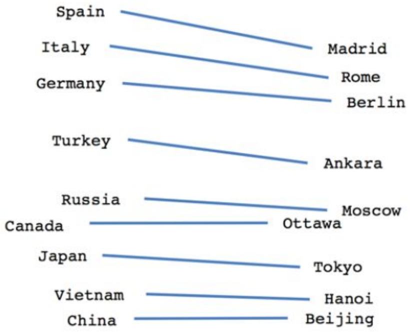




Male-Female



Verb tense



Country-Capital

Word Embeddings and Word2Vec

→ Look at the Tensorboard embedding visualizations again (from the previous exercise)

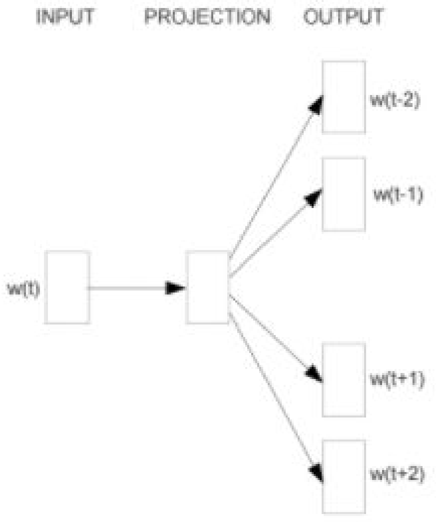
→ Words show up as ids. These are words encoded as integers in our pre-processing.
→ Process known as **Embedding Lookup** (TF provides an easy soln: **tf.nn.embedding_lookup()**)
→ Uses a weight matrix as a lookup table, trained just like any weight matrix.

→ **Word2Vec:**

→ Finds vectors containing semantic information, that represent words.

→ Words showing up in similar **contexts** (eg "black", "white", "red") have vectors near each other.

→ We use **Skip-Gram** architecture (performs better than **Continuous Bag Of Words** architecture).



Skip-gram

Word2Vec

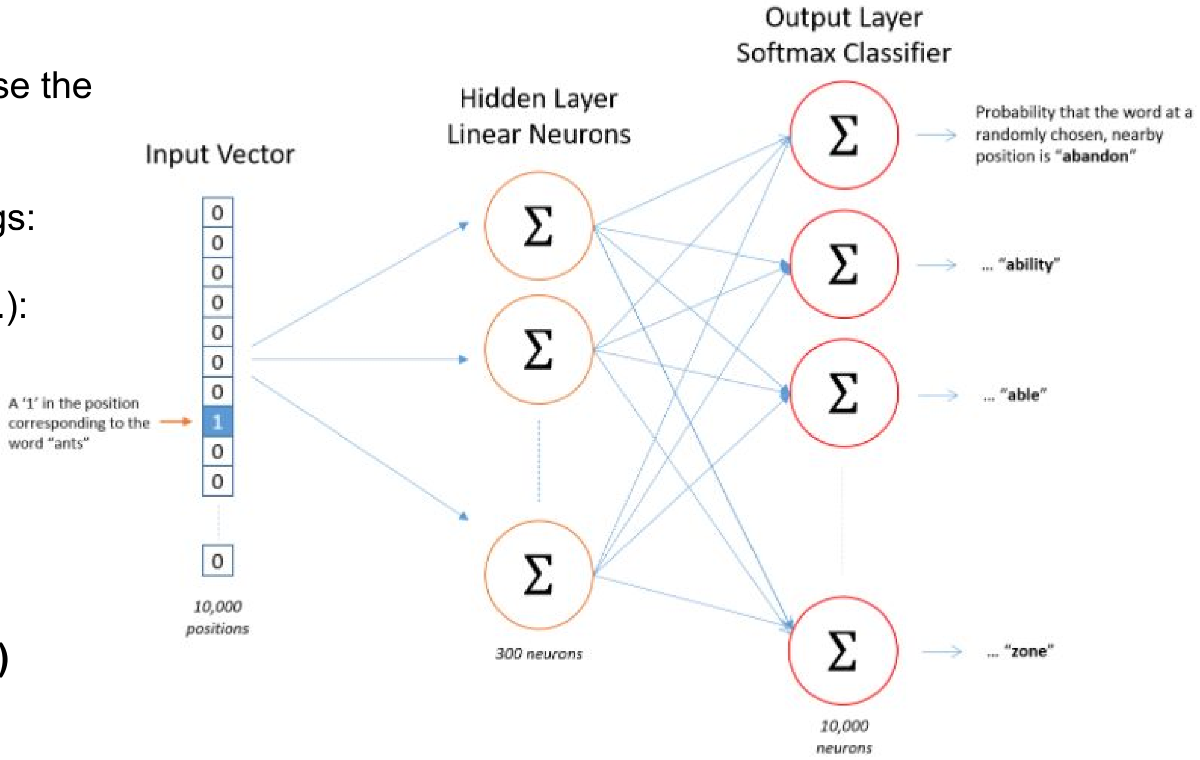
→ Training the hidden layer weight matrix to find efficient representations for words.

→ Discard Softmax and then use the trained **Embedding Layer**.

→ Some ways to improve embeddings:

→ **Subsampling** (Mikolov et al.): Remove “noisy” words that show up often (“the”, “of” …) but don’t provide much context to nearby words.

→ **Negative Sampling**: Using TensorFlow’s **tf.nn.sampled_softmax_loss()**



Questions?

Some References

1. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
2. http://karpathy.github.io/2015/05/21/rnn-effectiveness/
3. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
4. https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html
5. http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
6. http://www.thushv.com/natural_language_processing/word2vec-part-1-nlp-with-deep-learning-with-tensorflow-skip-gram/