CS 559: NLP, RNN, and LSTM

Lecture 11

Outline

- Introduction of Natural Language Processing (NLP)
- Sequential Data
- Recurrent Neural Network (RNN)
- Long Short-Term Memory (LSTM)

Natural Language Process (NLP)

According to IBM, 2.5 exabytes (=10^9 GB) of data were generated every day in 2017.

- Roughly 300 MB for each of us every day to process.
- Unstructured text, emails, social media, phone calls.

The goal of NLP is to make machine understand our spoken and written languages.

Natural Language Process – Tasks

Analysis Tasks

- **Syntactic (Language structure-based):**
  - Tokenization – the task of separating a text corpus into atomic units.
- **Semantic (Meaning-based):**
  - Sentence/Synopsis classification – the task of labeling data or classifying, e.g., article classification
  - Named Entity Recognition (NER) – the task of extracting entities, e.g., John – person
- **Pragmatic (Open problems to solve):**
  - Word-sense Disambiguation (WSD) – the task of identifying the correct meaning of a word, e.g., bad
  - Part-of-Speech (PoS) tagging – the task of assigning words to their respective parts of speech, e.g., noun, verb

Generation Tasks

- Language generation – the task of predicting next text by training a text
- Question Answering – the task of answering questions, e.g., Siri, Alexa, etc..
- Machine Translation (MT) – the task of transforming a sentence/phrase into a different language, e.g., Google Translator

Natural Language Process – Traditional Approach

1. **Preprocess** – reducing the vocabulary and distractions (for example, punctuations marks)
   - Tokenization is another preprocessing step that might need.
2. **Feature Engineering** – transforming raw text data into an appealing numerical format so that a model can be trained on that data
   - Bag of words – creates feature representations based on the word occurrence frequency.
   - N-gram – breaks down text into smaller components consisting of n letters.
3. **Learning Algorithm** – performing at the given task using the obtained features and the external resources
   - A Hidden Markov Model (HMM) – Learn the morphological structure and grammatical properties of the language by analyzing the corpus of related phrases to give statistics
   - A sentence planner – Correct any linguistic mistakes which we might have in the phrases using a database of rules.
   - A discourse planner – Generate a set of phrases for a given set of statistics using a HMM, order and structure a set of messages that need to be conveyed.
4. Prediction

Natural Language Process – Drawbacks of traditional approach and the deep learning approach

Drawback:

1. The preprocessing steps used in traditional NLP forces a trade-off of potentially useful information embedded in the text in order to make the learning feasible by reducing the vocabulary.
2. Feature engineering needs to be performed manually by hand. In order to design a reliable system, good features need to be devised.
3. Various external resources are needed for it to perform well, and there are not many freely available one.

Deep Learning Approach:

- Deep models learned rich features from raw data instead of using limited human-engineering features.
- Deep models made the traditional workflow more efficient.
- Deep models encompass significantly more features than a human would have engineered
- However…. Deep models are considered a black box.

Natural Language Process – The current state

Many different deep models have seen the light since their inception in early 2000.
They commonly share a resemblance, such as all of them using nonlinear transformation of the inputs and parameters.

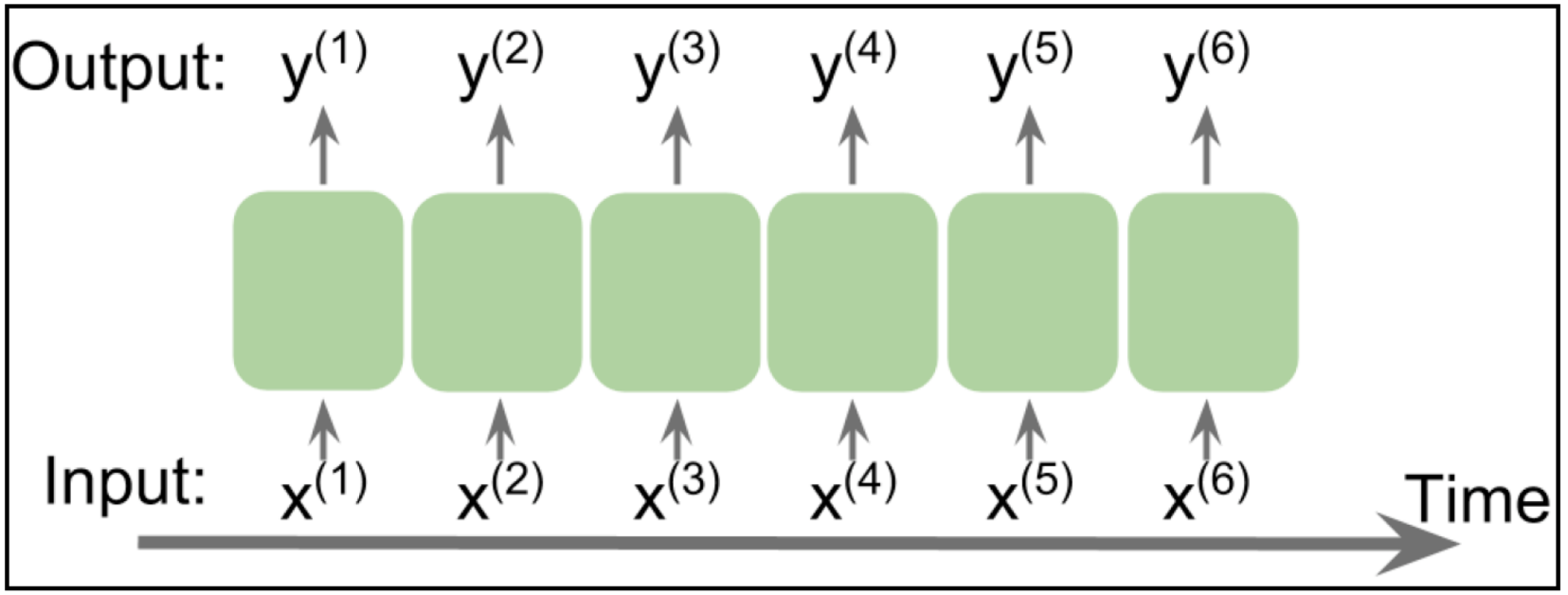
When processing text, as one of the most intuitive interpretations of text is to perceive it as a sequence of characters, the learning model should be able to do timeseries modelling, thus requiring the memory of the past, e.g., recurrent neural network (RNN).

Long Short-Term Memory (LSTM) networks are an extension of RMMs that encapsulate long-term memory.

Sequential Data

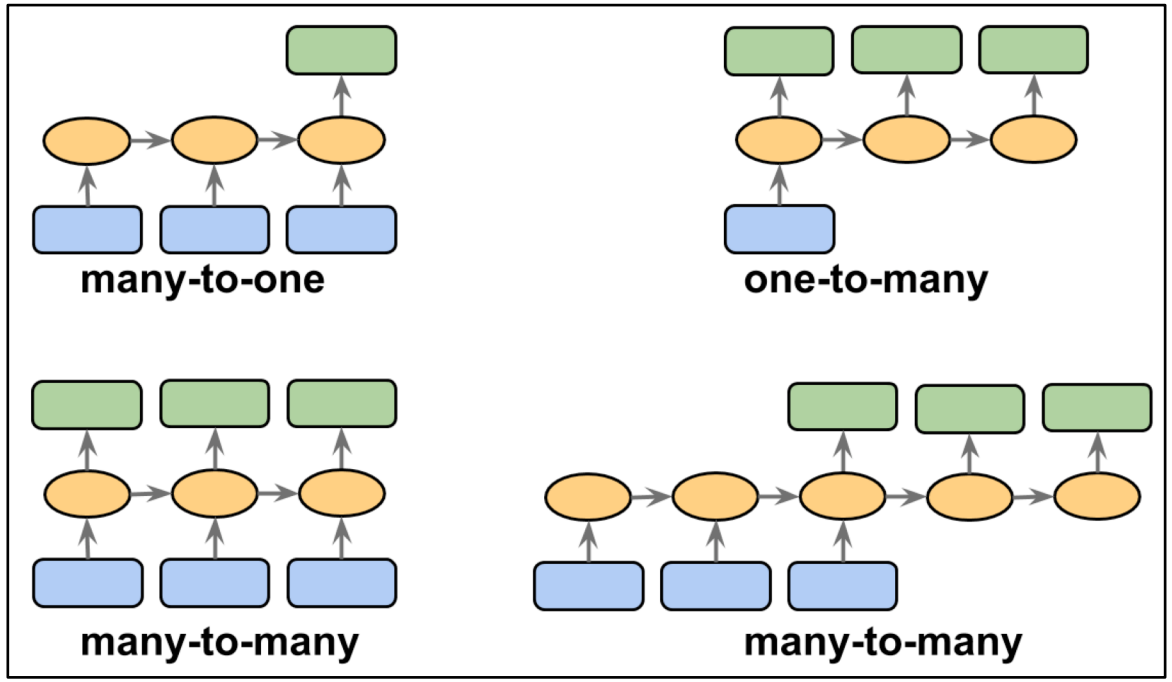
Elements in a sequence appear in a certain order and are not independent of each other.

- Supervised Learning Algorithms – the input data is independent and identically distributed
- With sequences – order matters
- We represent sequences as $\left(x^{(1)}, x^{(2)}, \ldots, x^{(T)}\right)$
  - The superscript indices indicate the order of the instance and the length of the sequence is T.
  - Time-series Data – both x's and y's naturally follow the order according to their time axis.
- CNNs – not capable of handling the order of input samples: do not have a memory of the past seen samples.
  - Weights are updated independent of the order in which the sample is processed.
- Recurrent Neural Networks (RNNs) – designed for modeling sequences and are capable of remembering past information and processing new events accordingly.



Sequential Data

We need to understand the different types of sequence modeling tasks to develop and appropriate model.



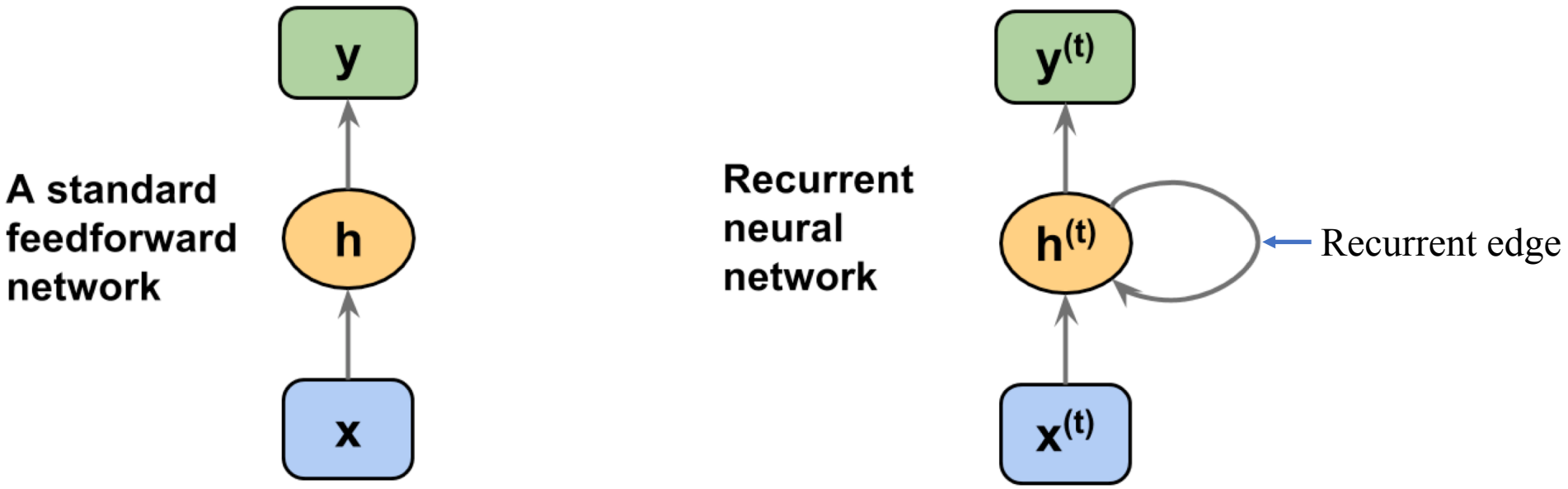
Many-to-one: The input data is a sequence, but the output is a fixed-size vector, not a sequence. E.g., the input is text-based and the output is a class label.

One-to-many: The input data is in standard format, not a sequence, but the output is a sequence. E.g., image captioning – the input is an image; the output is an English phrase.

Many-to-many: Both the input and out arrays are sequences. This category can be further divided based on whether the input and output are synchronized or not.

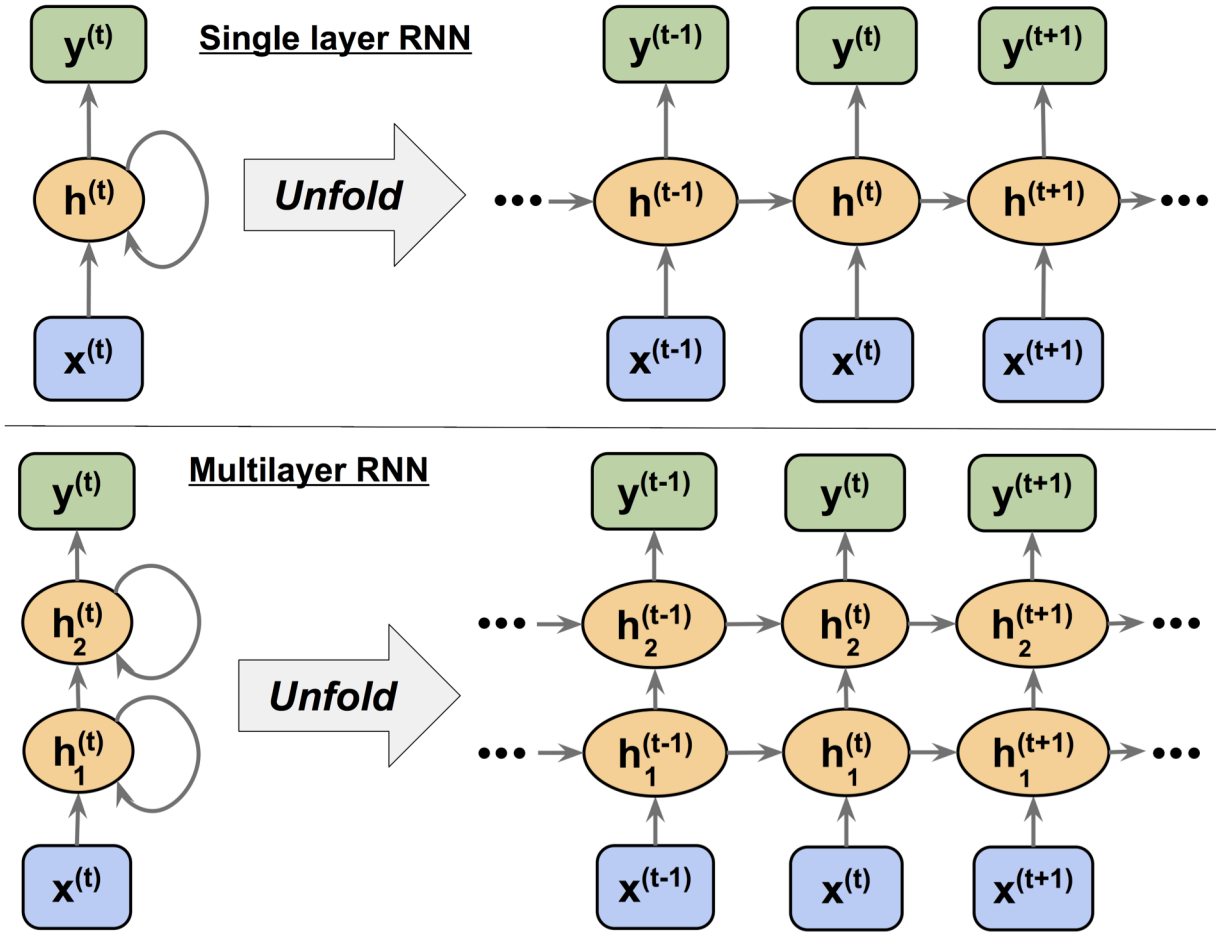
- A synchronized many to many modeling is video classification – where each frame in a video is labeled.
- A delayed many-to-many – translating a language into a another.

RNN - Architecture



Case: one hidden layer
Input layer x, hidden layer h, and output layer y are vectors with many units.
Recurrent edge - The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events.

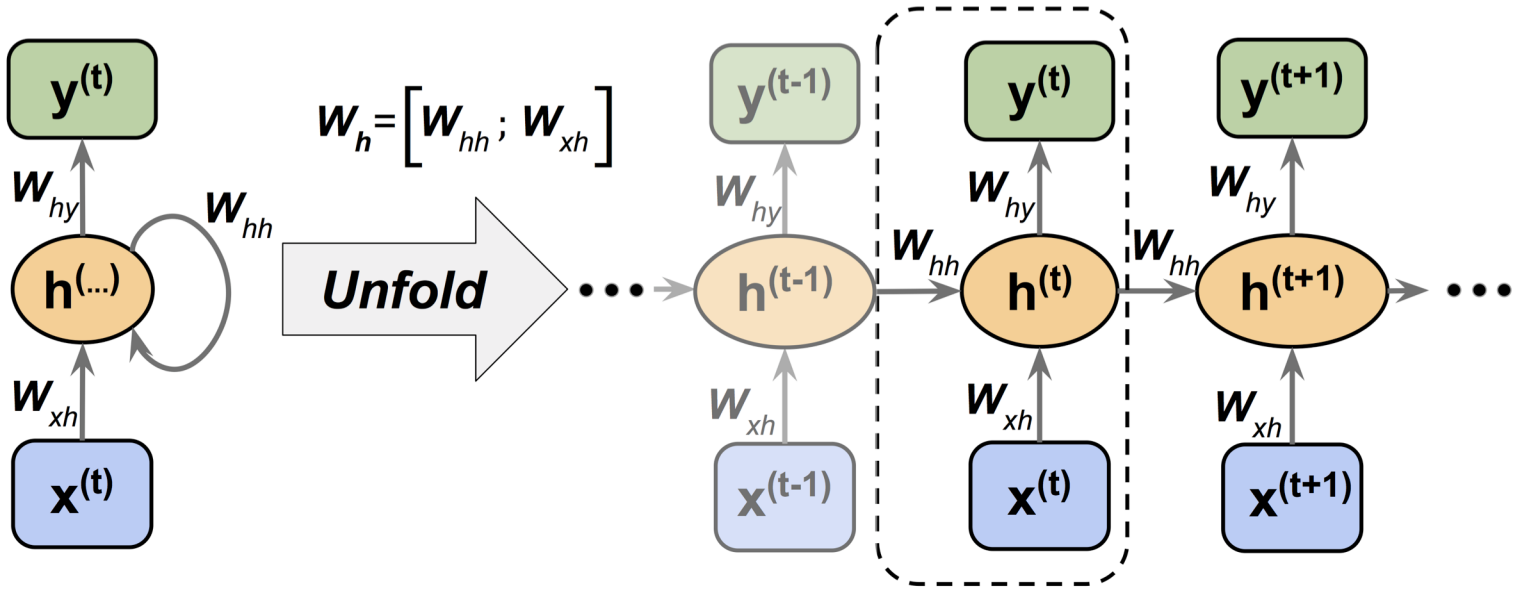
RNN - Architecture



Unfolded representation of RNN
Each hidden unit:

- Standard neural network – receives only one input – the net preactivation associated with the input layer.
- RNN – receives two distinct sets of input – the preactivation from the input layer and the activation of the same hidden layer from the previous time step $t - 1$.

RNN - Computation

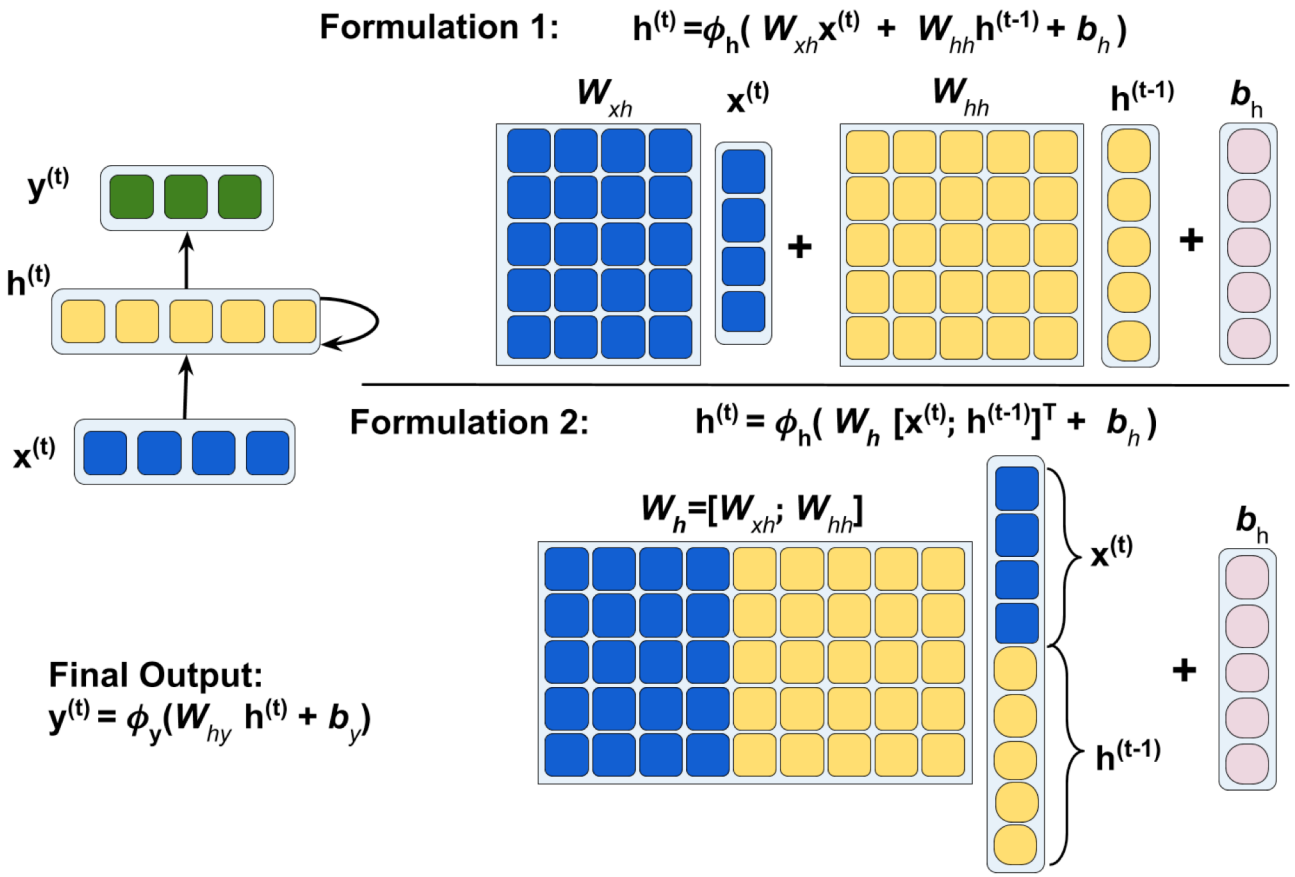


Consider a single hidden layer (the same concept applies to multilayer RNNs):
Weights do not depend on time t – they are shared across the time axis.

- W_{xh}: The weight matrix between the input $\boldsymbol{x}^{(t)}$ and the hidden layer $\boldsymbol{h}$.
- W_{hh}: The weight matrix associated with the recurrent edge.
- W_{hy}: The weight matrix between the hidden layer and output layer.
- W_h: A combined matrix where W_{xh} and W_{hh} are concatenated.

RNN - Computation

Computation of activation is very similar to standard multilayer perceptron.



For the hidden layer, the net input $\boldsymbol{z}_h$ is computed through a linear combination:

$$\boldsymbol{z}_h^{(t)} = W_{xh}\boldsymbol{x}^{(t)} + W_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h$$

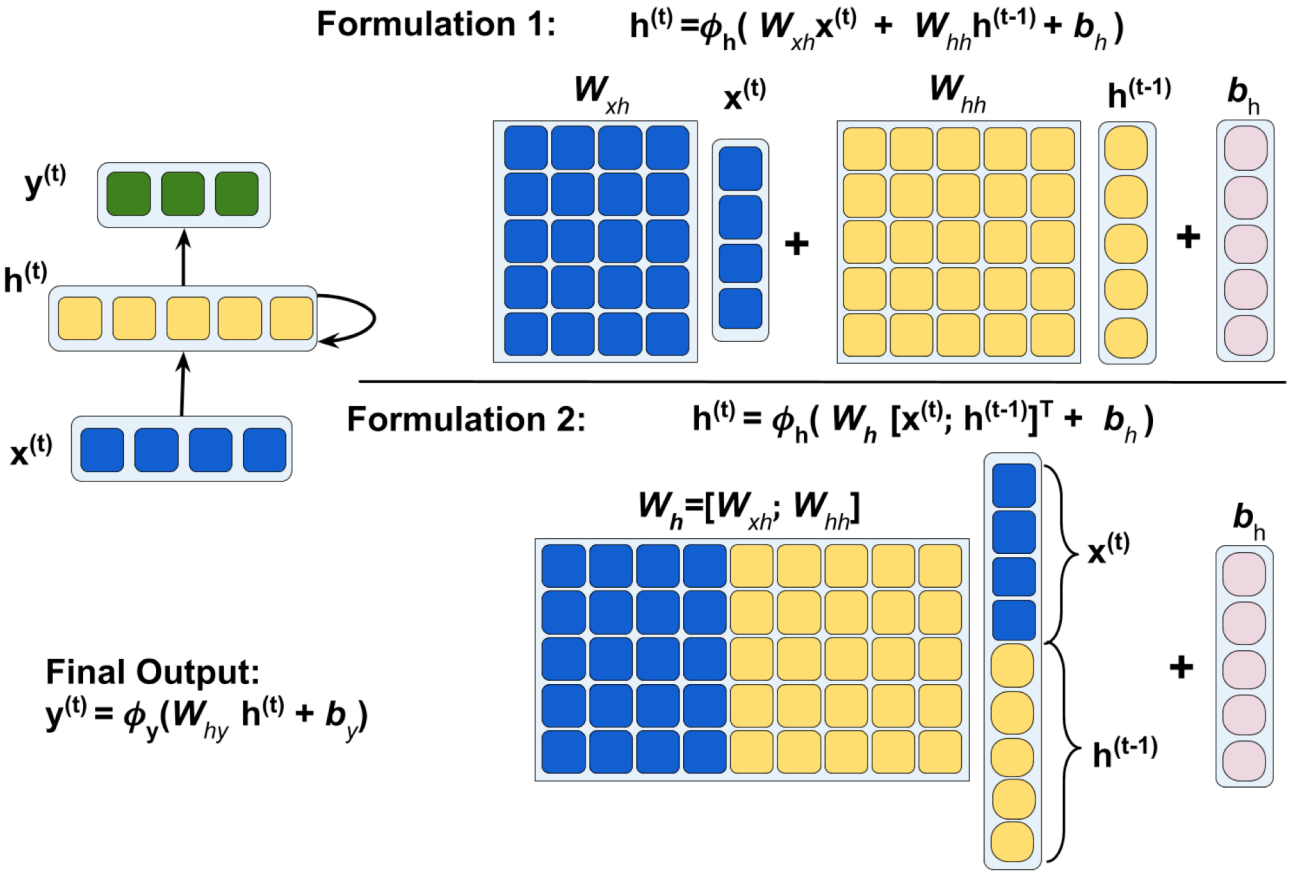
The activations of the hidden units at the time step t is:

$$\boldsymbol{h}^{(t)} = \phi_h\left(\boldsymbol{z}_h^{(t)}\right) = \phi_h\left(W_{xh}\boldsymbol{x}^{(t)} + W_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h\right)$$

where $\phi_h(\cdot)$ is the activation function of the hidden layer.

RNN - Computation

Computation of activation is very similar to standard multilayer perceptron.



Using $W_h = [W_{xh}; W_{hh}]$:
The activations of the hidden units at the time step t:

$$\boldsymbol{h}^{(t)} = \phi_h\left([W_{xh}; W_{hh}]\begin{bmatrix}\boldsymbol{x}^{(t)} \\ \boldsymbol{h}^{(t-1)}\end{bmatrix} + \boldsymbol{b}_h\right)$$

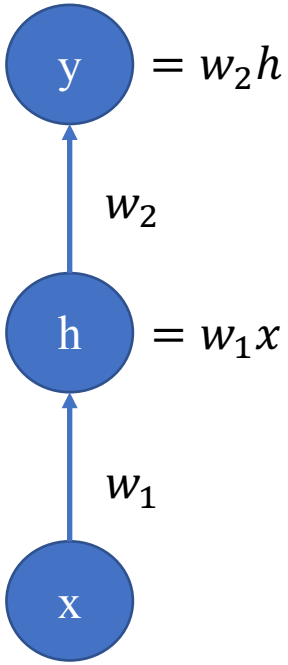
The activation of output units:

$$\boldsymbol{y}^{(t)} = \phi_y\left(W_{hy}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y\right)$$

Training RNNs using Backpropagation

In backpropagation, we

1. Calculate a prediction for a given input.
2. Calculate an error, E, of the prediction by comparing it to the actual label of the input.
3. Update the weights of the feed-forward network to minimize the loss calculated in step 2, by taking a small step in the opposite direction of the gradient $\partial E / \partial W$.



We can calculate $\partial E / \partial w_1$ using the chain rule as follows:

$$\frac{\partial E}{\partial w_1} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_1}$$

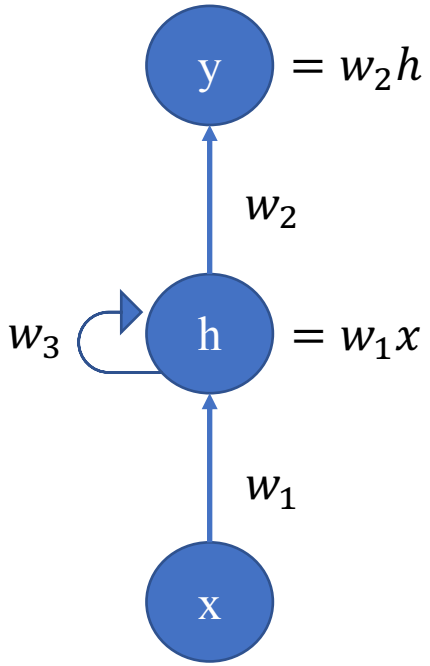
This simplifies to the following:

$$\frac{\partial E}{\partial w_1} = \frac{\partial (y-l)^2}{\partial y}\frac{\partial (w_2 h)}{\partial h}\frac{\partial (w_1 x)}{\partial w_1}$$

Training RNNs using Backpropagation

Try to do the same for the RNN:

- We have an additional recurrent weight w_3.
- To address the problem, the time components of inputs and outputs are omitted.



We can calculate $\partial E/\partial w_3$ using the chain rule as follows:

$$\frac{\partial E}{\partial w_3} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_3}$$

This simplifies to the following:

$$\frac{\partial E}{\partial w_3} = \frac{\partial (y-l)^2}{\partial y}\frac{\partial (w_2 h)}{\partial h}\left(\frac{\partial (w_1 x)}{\partial w_3} + \frac{\partial (w_3 h)}{\partial w_3}\right)$$

recursive

Problems: h is recursive (calculating h includes h itself) and h is not a constant and dependent on w_3.

The trick is to consider not a single input, but the full input sequence:

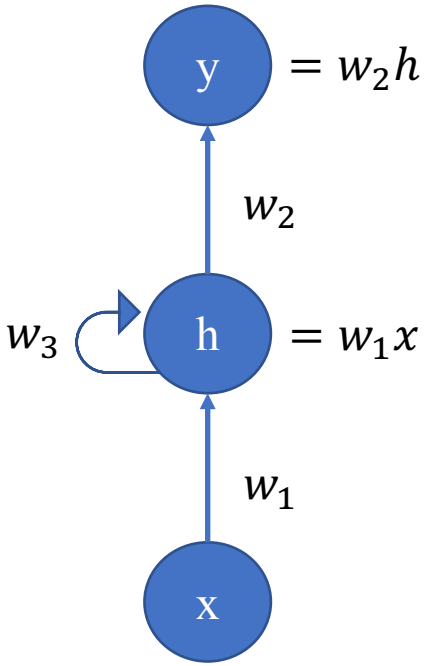
$$\frac{\partial E}{\partial w_3} = \sum_{j=1}^{3} \frac{\partial L}{\partial y_4}\frac{\partial y_4}{\partial h_4}\frac{\partial h_4}{\partial h_j}\frac{\partial h_j}{\partial w_3}$$

Training RNNs using Truncated BPTT

In TBPTT, we only need to calculate the gradients for a fixed number of T time steps.

When calculating $\frac{\partial E}{\partial w_3}$, for time step t, we only calculate derivates down to $t - T$:

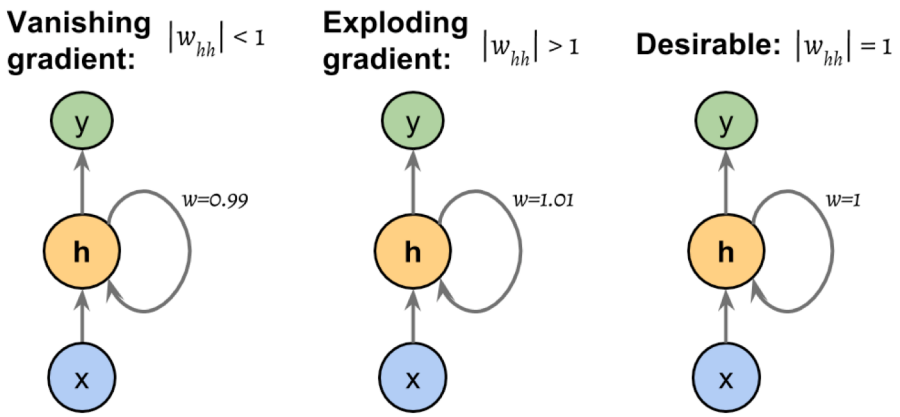
$$\frac{\partial E}{\partial w_3} = \sum_{j=t-T}^{t-1} \frac{\partial L}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_j} \frac{\partial h_j}{\partial w_3}$$



$$\frac{\partial E}{\partial w_3} = \frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial h_4}{\partial h_1} \frac{\partial h_1}{\partial w_3} = \frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial (w_1 x + w_3 h_3)}{\partial h_1} \frac{\partial (w_1 x + w_3 h_0)}{\partial w_3}$$

The issues of BP arise from the recurrent connections, we consider

$$\frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} \frac{\partial (w_3 h_3)}{\partial h_1} \frac{\partial (w_3 h_0)}{\partial w_3} \approx \frac{\partial L}{\partial y_4} \frac{\partial y_4}{\partial h_4} h_0 w_3^3$$



The learning algorithm for RNNs was introduced in 1990s Paul Werbos "Backpropagation Through Time: What It Does and How to Do It"

LSTM - A fancier family of RNNs.

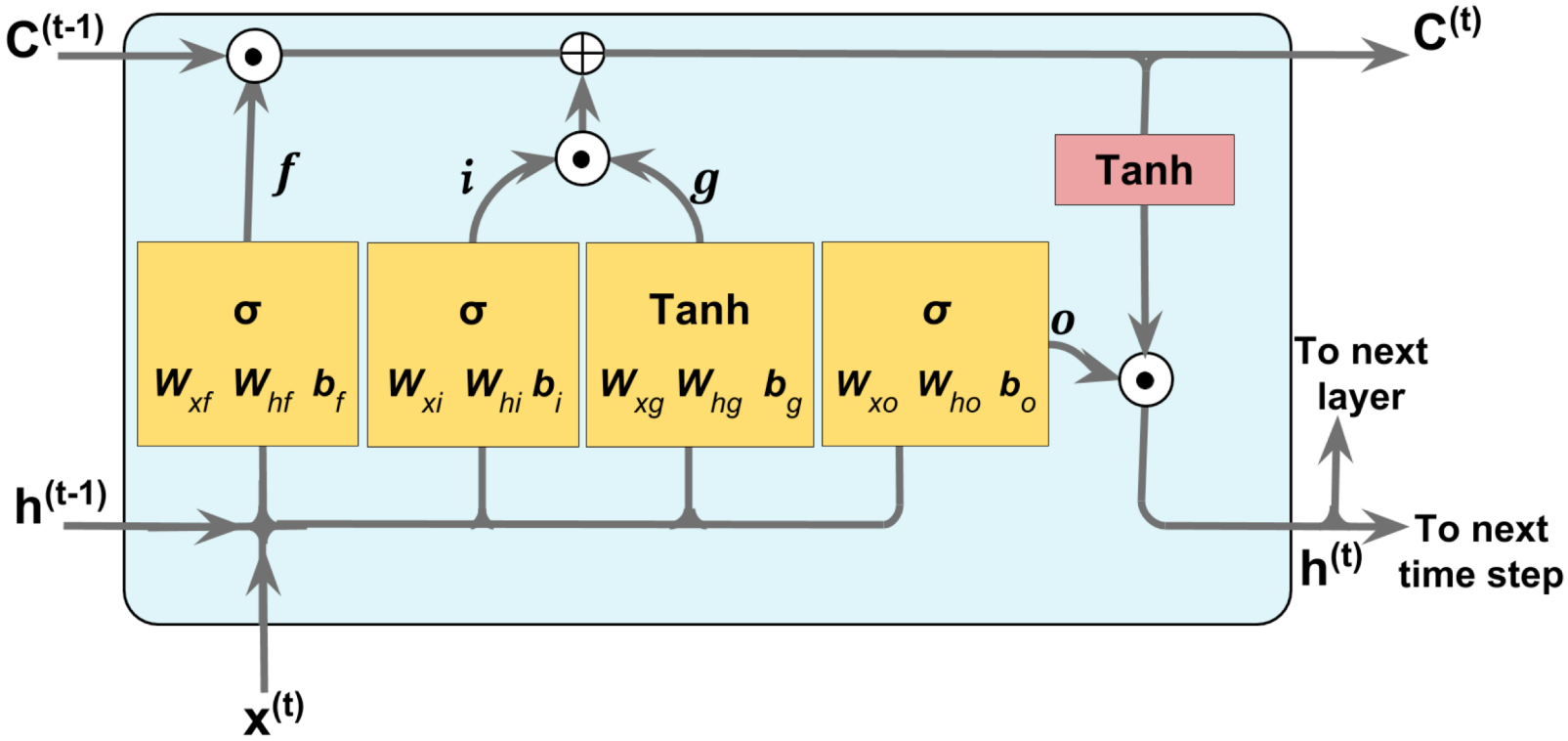
RNN to a cell architecture – the cell will output some state which is dependent (w/ nonlinear activation function) on previous cell state and the current input.

- The cell state is always changed with every incoming input.
- Leads the cell state of the RNNs to always change.

LSTM - can decide when to replace, update, or forget information stored in each neuron in the cell state.

- Cell State – The internal cell state (that is memory) of an LSTM cell.
- Hidden State – This determines how much the current input is read into the cell state.

LSTM - Overview

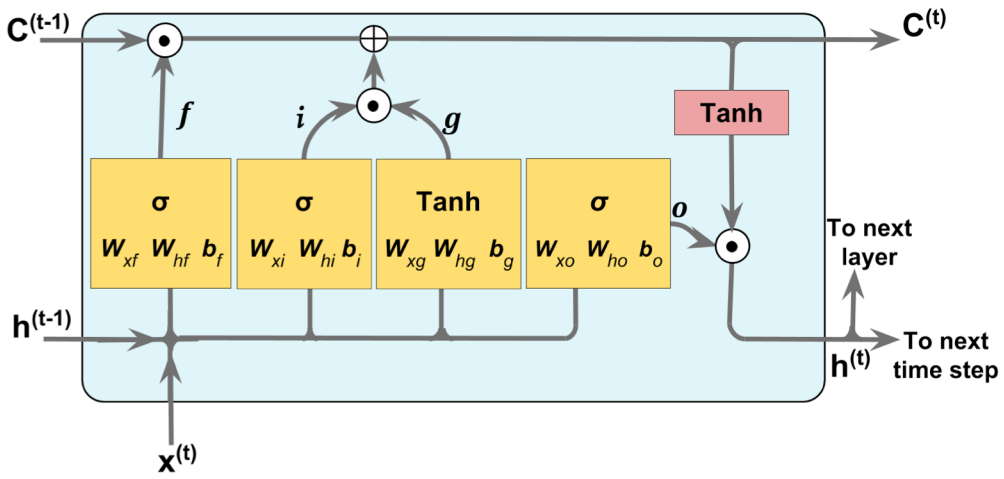


Memory Cell – The building block that essentially represents the hidden layer. There is a current edge that has the desirable weight to overcome the vanishing and exploding gradient problem. The value associated with this recurrent edge is called cell state. (S. Hochreiter and J. Schmidhuber, 1997)

The cell state from the previous time step, $C^{(t-1)}$, is modified to get the cell state at the current time step, $C^{(t)}$, without being multiplied directly with any weight factor.

The flow of information in the memory cell is controlled by some units of computation.
Gates: Passes output units.

LSTM - Gates



⊙: the element-wise product
⊕: the element-wise summation

- Forget Gate ($\boldsymbol{f}_t$) – allows the memory cell to reset the cell state without growing indefinitely. The forget gate decides which information is allowed to go through and which information to suppress. (”Learning to forget”, Gres et al. 2000)

$$\boldsymbol{f}_t = \sigma\left(W_{xf}\boldsymbol{x}^{(t)} + W_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f\right)$$

- Input Gate ($\boldsymbol{i}_t$) and input node ($\boldsymbol{g}_t$) – are responsible for updating the cell state.

$$\boldsymbol{i}_t = \sigma\left(W_{xi}\boldsymbol{x}^{(t)} + W_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right)$$
$$\boldsymbol{g}_t = \tanh(W_{xg}\boldsymbol{x}^{(t)} + W_{hg}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_g)$$

The cell state at time t is

$$C^{(t)} = \left(C^{(t-1)} \odot f_t\right) \oplus (i_t \odot g_t)$$

- Output Gate ($\boldsymbol{o}_t$) – decides how to update the values of hidden units

$$\boldsymbol{o}_t = \sigma\left(W_{xo}\boldsymbol{x}^{(t)} + W_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right)$$

- The hidden units at the current time step

$$\boldsymbol{h}^{(t)} = o_t \odot \tanh\left(C^{(t)}\right)$$

LSTMs to solve the vanishing gradient problem

Assume the hidden state is calculated as follows for a standard RNN

$$h_t = \sigma(W_x x_t + W_h h_{t-1})$$

Focusing on the recurrent part,

$$h_t = \sigma(W_h h_{t-1})$$

And

$$\frac{\partial h_t}{\partial h_{t-k}} = \prod_{i=0}^{k-1} W_h \sigma(W_h h_{t-k+i})\big(1 - \sigma(W_h h_{t-k+i})\big) = W_h^k \prod_{i=0}^{k-1} \sigma(W_h h_{t-k+i})\big(1 - \sigma(W_h h_{t-k+i})\big)$$

$$\frac{\partial h_t}{\partial h_{t-k}} \to 0$$

Looking at the cell state,

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t$$

In a similar way for LSTMs,

$$\frac{\partial c_t}{\partial c_{t-k}} = \prod_{i=0}^{k-1} \sigma\big(W_{fh} h_{t-k+i}\big)$$

The gradient will vanish if $W_{fh} h_{t-k+i} \ll 0$ and if $W_{fh} \gg 0$, the derivative will decrease much slower than it would in a standard RNN.

LSTMs to solve the vanishing gradient problem

Derivations for $\frac{\partial h_t}{\partial h_{t-k}}$ are trickier:

$$\frac{\partial h_t}{\partial h_{t-k}} = \frac{\partial(o_t \tanh(c_t))}{\partial h_{t-h}}$$

And once you solve this, we get something like below,

$$\tanh(\cdot)\,\sigma(\cdot)[1-\sigma(\cdot)]w_{oh} + \sigma(\cdot)[1-\tanh^2(\cdot)]\{c_{t-1}\sigma(\cdot)[1-\sigma(\cdot)]w_{fh} + \sigma(\cdot)[1-\tanh^2(\cdot)]w_{ch} + \tanh(\cdot)\,\sigma(\cdot)[1-\sigma(\cdot)]w_{ih}\}$$

We are only interested on w_{oh}, w_{fh}, w_{ch}, and w_{ih}:

$$\frac{\partial h_t}{\partial h_{t-k}} \approx \prod_{i=0}^{k-1} \gamma(\cdot)w_{oh} + c_{t-1}\gamma(\cdot)w_{fh} + \gamma(\cdot)w_{ch} + \gamma(\cdot)w_{ih}$$

Example: Movie Review Data

Data Description:
Data has two columns – review and sentiment (0 or 1)

Data preparation:

1. Need to encode the input into numeric values.
    - Find the unique words in the entire dataset – using **Counter** from the collections package**.**

```
## Preprocessing the data:
## Separate words and
## count each word's occurrence

counts = Counter()
pbar = pyprind.ProgBar(len(df['review']),
                       title='Counting words occurences')
for i,review in enumerate(df['review']):
    text = ''.join([c if c not in punctuation else ' '+c+' ' for c in review]).lower()
    df.loc[i,'review'] = text
    pbar.update()
    counts.update(text.split())
```

Example: Movie Review Data

1. Encoding
2. Create a mapping in the form of dictionary that maps each unique words to a unique integer number.

```
## Create a mapping:
## Map each unique word to an integer

word_counts = sorted(counts, key=counts.get, reverse=True)
print(word_counts[:5])
word_to_int = {word: ii for ii, word in enumerate(word_counts, 1)}


mapped_reviews = []
pbar = pyprind.ProgBar(len(df['review']),
                       title='Map reviews to ints')
for review in df['review']:
    mapped_reviews.append([word_to_int[word] for word in review.split()])
    pbar.update()
```

3. Make all the sequences have the same length.

```
## Define fixed-length sequences:
## Use the last 200 elements of each sequence
## if sequence length < 200: left-pad with zeros

sequence_length = 200  ## sequence length (or T in our formulas)
sequences = np.zeros((len(mapped_reviews), sequence_length), dtype=int)
for i, row in enumerate(mapped_reviews):
    review_arr = np.array(row)
    sequences[i, -len(row):] = review_arr[-sequence_length:]
```

Example: Movie Review Data

1. Encoding
2. Mapping
3. Set into the same length
4. Split the data into separate training and test sets.
5. Define a helper function that breaks a given dataset into chucks and return a generator to iterate through these chucks called **mini-batches**.

```
np.random.seed(123) # for reproducibility

## Function to generate minibatches:
def create_batch_generator(x, y=None, batch_size=64):
    n_batches = len(x)//batch_size
    x= x[:n_batches*batch_size]
    if y is not None:
        y = y[:n_batches*batch_size]
    for ii in range(0, len(x), batch_size):
        if y is not None:
            yield x[ii:ii+batch_size], y[ii:ii+batch_size]
        else:
            yield x[ii:ii+batch_size]
```

Example: Movie Review Data

Building an RNN model with four methods:

1. A constructor to set all the model parameters and to build the multilayer RNN model.

```
class SentimentRNN(object):
    def __init__(self, n_words, seq_len=200,
                 lstm_size=256, num_layers=1, batch_size=64,
                 learning_rate=0.0001, embed_size=200):
        self.n_words = n_words
        self.seq_len = seq_len
        self.lstm_size = lstm_size   ## number of hidden units
        self.num_layers = num_layers
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.embed_size = embed_size

        self.g = tf.Graph()
        with self.g.as_default():
            tf.compat.v1.(123)
            self.build()
            self.saver = tf.compat.v1.train.Saver()
            self.init_op = tf.compat.v1.global_variables_initializer()
```

Example: Movie Review Data

Building an RNN model with four methods:

1. Self.build.
2. A build method to declare three placeholders for input data, input labels, and the keep-probability for the dropout configuration of the hidden layer.

```
def build(self):
    ## Define the placeholders
    tf_x = tf.compat.v1.placeholder(tf.int32,
                shape=(self.batch_size, self.seq_len),
                name='tf_x')
    tf_y = tf.compat.v1.placeholder(tf.float32,
                shape=(self.batch_size),
                name='tf_y')
    tf_keepprob = tf.compat.v1.placeholder(tf.float32,
                name='tf_keepprob')
    ## Create the embedding layer
    embedding = tf.Variable(
                tf.random.uniform(
                    (self.n_words, self.embed_size),
                    minval=-1, maxval=1),
                name='embedding')
    embed_x = tf.nn.embedding_lookup(
                params=embedding, ids=tf_x,
                name='embeded_x')
```

Embedding:
A feature learning technique to automatically learn the salient features to represent the words in dataset.

- A reduction in the dimensionality of the feature space to decrease the effect of the curse of dimensionality.
- The extraction of salient features since the embedding layer in a neural network is trainable.

Example: Movie Review Data

Building an RNN model with four methods:

1. A constructor to set all the model parameters and to build the multilayer RNN model.
2. A build method to declare three placeholders for input data, input labels, and the keep-probability for the dropout configuration of the hidden layer.
    1. Build the RNN network with LSTM cells
        1. Define multilayer RNN cells
        2. Define the initial states for the RNN cells
        3. Create an RNN specified by the RNN cells and their initial states.

```
                name='embeded_x')

    ## Define LSTM cell and stack them together
    cells = tf.compat.v1.nn.rnn_cell.MultiRNNCell(
            [tf.contrib.rnn.DropoutWrapper(
               tf.compat.v1.nn.rnn_cell.BasicLSTMCell(self.lstm_size),
               output_keep_prob=tf_keepprob)
             for i in range(self.num_layers)])

    ## Define the initial state:
    self.initial_state = cells.zero_state(
            self.batch_size, tf.float32)
    print('  << initial state >> ', self.initial_state)

    lstm_outputs, self.final_state = tf.compat.v1.nn.dynamic_rnn(
            cells, embed_x,
            initial_state=self.initial_state)
    ## Note: lstm_outputs shape:
    ##  [batch_size, max_time, cells.output_size]
    print('\n  << lstm_output   >> ', lstm_outputs)
    print('\n  << final state   >> ', self.final_state)

    ## Apply a FC layer after on top of RNN output:
    logits = tf.compat.v1.layers.dense(
            inputs=lstm_outputs[:, -1],
            units=1, activation=None,
            name='logits')

    logits = tf.squeeze(logits, name='logits_squeezed')
    print ('\n  << logits        >> ', logits)

    y_proba = tf.nn.sigmoid(logits, name='probabilities')
    predictions = {
        'probabilities': y_proba,
        'labels' : tf.cast(tf.round(y_proba), tf.int32,
            name='labels')
    }
    print('\n  << predictions   >> ', predictions)

    ## Define the cost function
    cost = tf.reduce_mean(
            input_tensor=tf.nn.sigmoid_cross_entropy_with_logits(
            labels=tf_y, logits=logits),
            name='cost')

    ## Define the optimizer
    optimizer = tf.compat.v1.train.AdamOptimizer(self.learning_rate)
    train_op = optimizer.minimize(cost, name='train_op')
```

Example: Movie Review Data

Building an RNN model with four methods:

1. A constructor to set all the model parameters and to build the multilayer RNN model.
2. A build method to declare three placeholders for input data, input labels, and the keep-probability for the dropout configuration of the hidden layer.
3. A train method that creates a TensorFlow session for launching the computation graph, iterates through the mini-batches of data, and runs for a fixed number of epochs, to minimize the cost function defined in the graph.
4. A predict method that creates a new session, restores the last checkpoint saved during the training process and carries out the predictions for the test data.

```
def train(self, X_train, y_train, num_epochs):
    with tf.compat.v1.Session(graph=self.g) as sess:
        sess.run(self.init_op)
        iteration = 1
        for epoch in range(num_epochs):
            state = sess.run(self.initial_state)

            for batch_x, batch_y in create_batch_generator(
                        X_train, y_train, self.batch_size):
                feed = {'tf_x:0': batch_x,
                        'tf_y:0': batch_y,
                        'tf_keepprob:0': 0.5,
                        self.initial_state : state}
                loss, _, state = sess.run(
                        ['cost:0', 'train_op',
                         self.final_state],
                        feed_dict=feed)

                if iteration % 20 == 0:
                    print("Epoch: %d/%d Iteration: %d "
                          "| Train loss: %.5f" % (
                           epoch + 1, num_epochs,
                           iteration, loss))

                iteration +=1
            if (epoch+1)%10 == 0:
                self.saver.save(sess,
                    "model/sentiment-%d.ckpt" % epoch)

def predict(self, X_data, return_proba=False):
    preds = []
    with tf.compat.v1.Session(graph = self.g) as sess:
        self.saver.restore(
            sess, tf.train.latest_checkpoint('model/'))
        test_state = sess.run(self.initial_state)
        for ii, batch_x in enumerate(
            create_batch_generator(
                X_data, None, batch_size=self.batch_size), 1):
            feed = {'tf_x:0' : batch_x,
                    'tf_keepprob:0': 1.0,
                    self.initial_state : test_state}
            if return_proba:
                pred, test_state = sess.run(
                    ['probabilities:0', self.final_state],
                    feed_dict=feed)
            else:
                pred, test_state = sess.run(
                    ['labels:0', self.final_state],
                    feed_dict=feed)

            preds.append(pred)

    return np.concatenate(preds)
```

Addition: Time Series Using LSTM – much relevant to the project

Project Extension?

Preferred Exam Date?