📖

Chapter 15

> Preprocessing Sequences Using RNNs and CNNs

👉 GitHub Page: LINK
Google Drive: LINK

- Prediction is something people do all the time and this chapter will introduce recurrent neural network (RNN)
- RNNs can analyze time series data such as stock prices to tell you when to buy or sell
- RNNs can also analyze the car's trajectory in autonomous driving
- RNNs can also take texts, sentences, audio inputs
    - Used for automatic translation
    - speech to text
- Two main difficulty with RNN:
    - Unstable gradient, can be alleviated using various techniques including recurrent dropouts and recurrent layer normalization
    - A limited short-term memory which can be extended with LSTM
- RNNs are note the only ones capable of handling sequential data:
    - For small short sequences a deep neural network would do
    - For long sequences convolutional neural network can work quite well too
- This chapter will end by implementing a **WaveNet**, A CNN architecture capable of handling sequence

Recurrent Neurons and Layers

- All the neural networks discussed are feedforward
- Recurrent neural network look similar but it also has connections backwards
- The simplest possible RNN on Figure 15-1
    - composed of one neuron receiving inputs
    - producing and output
    - and sending that output back to itself
- The recurrent output receives the inputs $x_{(t)}$ as well as its own output from previous time step $t_{(t-1)}$
- represent a tiny neural network against time axis, called **unrolling the network through time**

> 👉 Recurrent neuron left to right though time on page 498 Figure 15-1

- Can easily create a layer of recurrent neurons:
    - At each time step t, every neuron receives both
        - input vector x_t
        - output vector from the previous time step y_{t-1}
    - both inputs and outputs are vectors now

> 👉 Layer of recurrent neurons and unrolled through time on page 499 Figure 15-2

- Recurrent neurons has two sets of weights:
    - input $x_{(t)}$
    - previous time step $y_{(t-1)}$
- weight vector w_x and w_y

$$y_{(t)} = \phi(W_t^T x_{(t)} + W_y^T y_{(t-1)} + b$$

- just like feedforward network, can compute a recurrent layer's output by replacing all input at time step t in an input matrix $X_{(t)}$

$$\begin{aligned} Y_(t) &= \phi(X_{(t)}W_x + Y_{(t-1)}W_y + b) \\ &= \phi([X_{(t)}Y_{(t-1)}]W + b) \text{ with } W = \begin{bmatrix} W_x \\ \\ W_y \end{bmatrix} \end{aligned}$$

- $Y_{(t)}$ is a $m \times n_{\text{neurons}}$ matrix containing the layer's output at time step t for each instance in mini-batch
- $X_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the inputs for all instances
- W_x is an $m \times n_{\text{neurons}}$ matrix containing the connection weights for the output from pervious step
- b is a vector of size n_{neurons} containing each neuron's bias term
- The weight matrices W_x and W_y are often concatenated vertically into a single weight matrix W of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$
- Notation $[X_{(t)}Y_{(t-1)}]$ represents the horizontal concatenation of the matrices $X_{(t)}$ and $Y_{(t-1)}$
- Notice that $Y_{(t)}$ is a function of $X_{(t)}$ and

Memory Cell

- Since recurrent neurons has time step t, you can say it has memory
- A part of neural network that preserves some state across steps is called memory cell
- A typical cell is capable to learn short, simple patterns about 10 steps long
- There's also a more powerful type of cell that could learn longer patterns
- a cell's state at time step t denoted by $h_{(t)}$ is a function of some input at that time step
- It's output at time step t denoted by $y_{(t)}$ is also a function of the previous state and the current inputs
- For the simple cell, output is simply equal to the state
- But more complex cells may not be the case

👉 A cell's hidden state and outputs diagram on page 501 Figure 15-3

Input and Output Sequence

- RNN can simultaneously take a sequence of inputs and produce a sequence of outputs
- sequence to sequence network is useful for time series prediction (top left network in Figure 15-4) like stock prices
- Alternatively, it's also possible to feed thee network sequence of inputs and ignore all outputs except the last one
- This is called a sequence to vector network
- A example would be reading a movie review and giving a score between -1[hate] to 1[love]
- Alternatively, it's possible to feed the network the same input vector over and over again at each time step and let it output a sequence
- This is called vector to sequence network, diagram on Figure 15-4 top right
- Example could be outputting caption for image
- Lastly it's possible to have a sequence to vector network called encoder
- followed by a vector to sequence network called decoder
- This could be used for translating languages
    - feed a sentence in one language
    - encoder convert sentence into a single vector representation
    - decode the vector in another language
- This two steps are called decoder-encoder
- This works way better than using single RNN to translate on the fly

👉 Different sequence networks diagram on page 502 Figure 15-4

Training RNNs

- To train RNN it uses backpropagation through time as shown in figure 15-5
- Just like regular backpropagation there's a forward pass through the unrolled network
- Then the output sequence is evaluated using cost function $C(...)$
- The gradient of the cost function is are backpropagated through the network
- Finally the model parameters update from the computed gradient from backpropagation
- Keras takes care of the complexity

> 👉 A diagram showing backpropagation through time on page 503 figure 15-5

Forecasting a Time Series

- In cases like:
    - studying active users per hour on your website
    - daily temperature in city
    - company's financial health
    - measured quarterly using multiple metrics
- In all the cases above the data will be a sequence of one or more values per time step
- This is called **time series**
- In example one and two there's is a single value per time step, these are **univariate time series**
- while example 2-3 are **multivariate time series**
- typical task is to predict future values which is called **forecasting**
- Another common task is fill in the blanks called **imputation**

- Figure 15-6 shows 3 univariate time series each of the 50-time steps long and the goal here is to forecast the value at the next time step for each of them

👉 Time series forecasting graphs on page 504 Figure 15-6

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9909778e-515e-412a-91b1-0f7d9fc6a001/time_series_forcasting.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e1be977a-aee3-4a12-aa45-8901dee7858f/time_series_forcasting.py

Baseline Metrics

- Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/8ab3051c-72d9-408e-8e9f-3fe54731a3b6/baseline_metrics.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ff21eec9-93cc-4294-b9d5-540ccc94cac0/baseline_metrics.py

Implementing a Simple RNN

- Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/fa20344d-22d1-4fe7-a64b-c5fffaa3c9fb/implementing_simple_rnn.ipynb https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9ec96e81-26e8-4268-bc3f-c3437227ea4f/implementing_simple_rnn.py

Deep RNNs

- Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7bf876a2-1539-42ee-a428-b03898809761/deep_rnns.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c4cc74b5-8d93-49c5-ba20-85c868c1ff79/deep_rnns.ipynb

Forecasting Several Time Steps Ahead

- Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/67b1c2b9-c26b-48fd-b624-fe79335ee98f/forecasting_multiple_steps_ahead_per_value.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/366596a1-614c-434b-a62e-c4759ee49605/forecasting_multiple_steps_ahead_per_value.py https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b08e6453-c937-46c0-a346-3c78a1629e9a/forecasting_multiple_steps_ahead.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ab23cc60-6bf5-4945-bbe8-6a7cd05f368b/forecasting_multiple_steps_ahead.py

Handling Long Sequences

- To train deep RNN, it's required to run it over many time steps
- This may cause issue like unstable gradient discussed in chapter 11
    - May take forever to train
    - training may be unstable
- When RNN process a long sequence it may forget the input

Fighting the Unstable Gradient Problem

- Tricks used in DNN can also be used in RNN:
    - Good parameter initialization
    - Faster optimizers
    - Dropouts
    - etc…
- Non-saturating activation function (e.g. RELU) does not help
    - Non-saturation function does not provide the slight increase or decrease in weights
    - Can reduce this by decreasing the learning rate
    - Or use a saturating activation function instead
- There's still a chance for gradient to explode

    - If training is unstable, monitor the size of gradient and perhaps use gradient clipping
- Batch normalization cannot be used in RNN
- BN can be used at each time step with the same parameters regardless of the actual scale and offsets of the inputs and hidden state
    - In practice this does not yield good result !!!
- A better normalization that often works with RNN is **layer normalization**
    - very similar to BN
    - instead of normalizing though batch, it normalizes through feature dimensions
    - Advantage is that it can compute the required statistics on the fly
    - Behaves the same way in training and testing
    - Like BN, layer normalization learns a scale and an offset parameter for each input
- The code below uses Keras to implement layer normalization inside a single cell

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/af42063e-68bf-45a7-b1fc-ab4cc1356ad0/layer_normalization.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/f47d918e-f635-4e50-8d34-1339bb088f1b/layer_normalization.ipynb

Tackling the Short-Term Memory Problem

- Due to data transformation some information may get lost on it's way
- After a while RNN contain virtually no initial input

LSTM Cells

- Long Short Term Memory Cell was created in 1997 and improved over the years
- LSTM:
    - can be used very much like a basic cell
    - Faster training
    - detect long-term dependencies in data
- keras LSTM:

```
model = keras.models.Sequential([
  keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
  keras.layers.LSTM(20, return_sequences=True),
  keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/00756692-f608-49a6-8e04-1e2edcf54f34/keras_lstm_layer.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/263a2b8d-2240-43d0-8b37-829f26ab1920/keras_lstm_layer.ipynb

- LSTM cell:

```
model = keras.models.Sequential([
  keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                   input_shape=[None, 1]),
  keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
  keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a2442d1d-f060-445c-80e0-86bc5384691b/lstm_cell.py https://s3-us-west-2.amazonaws.com/secure.notion-static.com/630a8dc4-47cf-4932-8291-77cc6d7ca5a6/lstm_cell.ipynb

- LSTM is better optimized with GPU so it's preferred
- Architecture of LSTM on figure 15-9
- LSTM is exactly like a normal cell except that its state is split into two vectors:
    - $h_{(t)}$
    - $c_{(t)}$
- h_t is short term state
- $c_{(t)}$ is long term state

👉 LSTM cell architecture on page 516 Figure 15-9

- The key idea is that the network can learn what to store in the long-term state, what to throw away and what to read from it
- In LSTM, as the input goes into the cell, some the inputs are changed and sent to the next cell
- At the same time, the input is copied and also sent to the next cell
- current input $x_{(t)}$ and previous memory $h_{(t-1)}$ are fed into the cell:
    - The main layer is the one that outputs $g_{(t)}$
        - It analyzes current input and previous state
        - Its most important part are stored in long term memory and then it's outputted
    - Three other layers are **gate controllers**
        - They use logistic activation function therefore outputs 0-1
        - Forget gate $f(t)$ controls which parts to forget

- Input gate $i_{(t)}$ controls which part $g_{(t)}$ should be added to long term state
- output gate $o_{(t)}$ controls which part of the long term state should be read and output at time step both $h_{(t)}$ and $y_{(t)}$

- LSTM cell can learn to recognize an important input, store it in long-term state and preserve it as long as needed and extract whenever is needed
- The equations below summarizes LSTM:

$$i_{(t)} = \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i)$$
$$f_{(t)} = \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f)$$
$$o_{(t)} = \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_o)$$
$$g_{(t)} = \tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g)$$
$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)}$$
$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})$$

- $W_{xi}, W_{xf}, W_{xo}, W_{xg}$ are weight matrices for connection to input vector $x_{(t)}$
- $W_{hi}, W_{hf}, W_{ho}, W_{hg}$ are weight metrices for connection to previous short-term state $h_{(t-1)}$
- b_i, b_f, b_o, b_g are bias terms for each four layers

Peephole connections

- Regular LSTM can only look at the input $x_{(t)}$ and the previous short-term state $h_{(t-1)}$
- good idea to give more context by letting them to peek at long term states as well
- LSTM variant called peephole connections allows it to do that
- previous long term state is added as an input to and often increases performance
- Keras LSTMCell does not support peekhole connections yet
- There's lots of variations of LSTM cells, GRU is one of them

GRU cells

- Gated Recurrent Unit (GRU)
- Introduced Encoder-Decoder network

👉 GRU cell diagram shown on page 510 Figure 15-10

- GRU is simplified version of LSTM cell and performs just as well:
  - Both state vectors are merged into a single vector $h_{(t)}$
  - A single gate controller $z_{(t)}$ controls both forget gate and input gate
  - There is not output gate, full vector is passed through every time
- The equations below summaries GRU

$$z_{(t)} = \sigma(W_{xz}^T x_{(t)} + W_{hz}^T h_{(t-1)} + b_z)$$
$$r_{(t)} = \sigma(W_{xr}^T x_{(t)} + W_{hr}^T h_{(t-1)} + b_r)$$
$$g_{(t)} = \tanh(W_{xg}^T x_{(t)} + W_{hg}^T (r_{(t)} \otimes h_{(t-1)}) + b_g)$$
$$h_{(t)} = z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)}$$

- Keras provides keras.layers.GRU layers based on GRUCell memory cell
- GRU and LSTM still have hard times learning sequences of 100 time steps or more like audio samples
- One way to solve this is to shorten the input sequences using 1D convolutional layers

Using 1D Convolutional layers to process sequences

- 1D convolutional layer is similar to 2D convolutional layer
- 1D convolutional layer slides several kernels across a sequence producing a 1D feature map
- To create a 1D convolutional network with GRU in Keras

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c6f2221f-e463-457a-8a92-31b6b3bf6766/1d_cnn_and_gru.py https://s3-us-west-2.amazonaws.com/secure.notion-static.com/87257b56-06db-4739-98bd-5a479a8e347f/1d_cnn_and_gru.ipynb

- Best result so far in compared to all other models

WaveNet

- Stacked 11D convolutional layer, doubling the dilation rate at every layer
    - First convolutional layer gets a glimpse of just two time steps each time
    - next one sees four
    - next one sees eight
    - and so on...
- lower layers learn short-term patterns and higher layers learn long term patterns

👉 Diagram showing WaveNet on page 522 Figure 15-11

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3c9860cf-252f-41df-be04-6b082ebcf088/WaveNet.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c4d72d33-fc34-4bb8-b47b-7d87f97fdf12/wavenet.py

Exercises

1. Applications of RNNs include:
   - Speech recognition
   - Autonomous driving
   - Text prediction

Sequence to Sequence applications:
- classifying music
- analyzing the statement of book review
- predicting probability
vector to vector applications:
- music playlist based on embedding
- melody based on parameters

2. RNN layer must have 3-dimensional inputs:
   - first dimension is batch size
   - second represents the number of time steps
   - third holds the inputs at each step

   The outputs are also 3-dimensional, the first two are the same but the last dimension is equal to the number of neurons
3. For sequence to sequence all layers must have return sequences set to True
4. I would use simple RNN architecture
5. The main difficulty of RNN is:
   - Unstable gradient
   - limited short term memory
6. Refer back to notes
7. 1D convolutional finds patterns between the 1D data like text. 1D convolutional layer also doesn't have memory it just processes data
8. I would use Convolutional neural network for image processing and then feed the values into an sequence-to-sequence RNN network and then get a result
9. Code:
   N/A (Too Difficult)
10. Code:
    N/A (Too Difficult)