Recurrent Neural Networks

Weizi Li

Department of Computer Science
University of Memphis

- Recap CNN
- Introduction
- Vanilla RNN
- Gated RNN
- Example Code

Recap CNN

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(),
              metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
```

- What if it doesn't work?
- Try popular network architecture and hyperparameter settings (link1)
- Consult successful examples (link2)
- Train more epochs
- Data, data, data!

Introduction

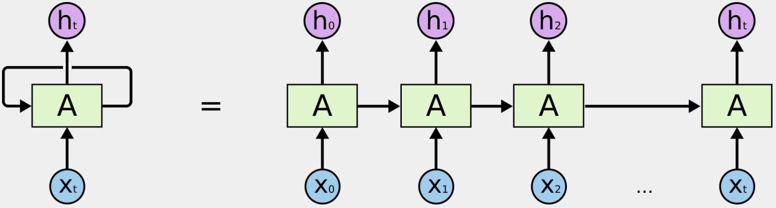
- CNN can only process known grid-structure data and produce a fixed-sized vector as output (e.g., 10 probabilities for 10 classes).
- CNN uses a fixed amount of computational steps, i.e., the number of layers is fixed.

- RNN is designed to process a sequence of values, which can be either continuous or discrete. The "values" can be time series, images, language scripts, etc.
- We use "time steps" to describe the order of elements of a sequence.

Traditional Sequence Modeling vs. RNN

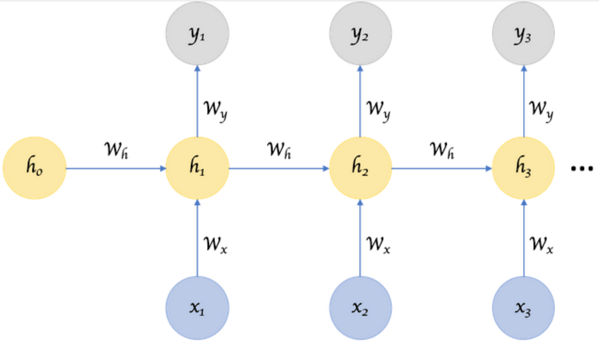
- Traditional approach requires manually designing training examples/features.
- A moving window approach is usually adopted.
  - Large window size: computationally expensive
  - Small window size: not enough information
  - Essentially, we don't know the length of the historical data that we should pay attention to.
  - Sometimes, the history that we should pay attention to is of variable length.
- RNN can handle input with different lengths.

- "Recurrent" refers to information flowing back to itself, which is different from "Feedforward" neural networks such as CNNs.



Vanilla RNN

- x: input; y: output
- h: hidden layer ("memory" from previous layers)
- $h_t = f(W_h h_{t-1} + W_x x_t)$ (f: activation function)
- $y_t = p(W_y h_t)$ (p: post-processing function such as softmax)
- Ws are weights to be learnt



- Input is received at each layer (unique to RNN), as well as from the previous layer (same for RNN and CNN)
- The number of layers changes based on the length of a sequence
- Shared parameters between the layers; in CNN each layer has different parameters (kernels, etc)

- Bengio et al., “Learning long-term dependencies with gradient descent is difficult”, 1994
- The probability of successful training of a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20

- As the gap between the relevant information and the current prediction grows, RNN is unable to use the far away information for generating correct predictions

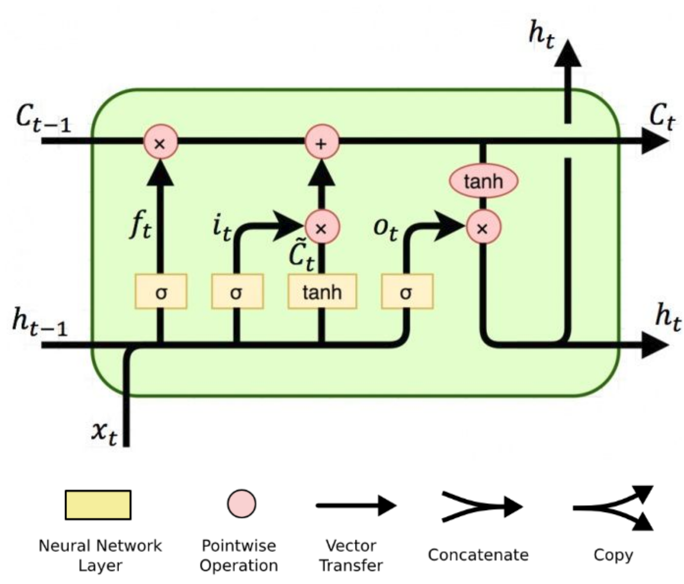
- Another problem is that h (hidden layers) get re-written completely at every time step
- It's helpful if we can control how much old information to retain and how much new information to add (at each hidden layer)

Gated RNN

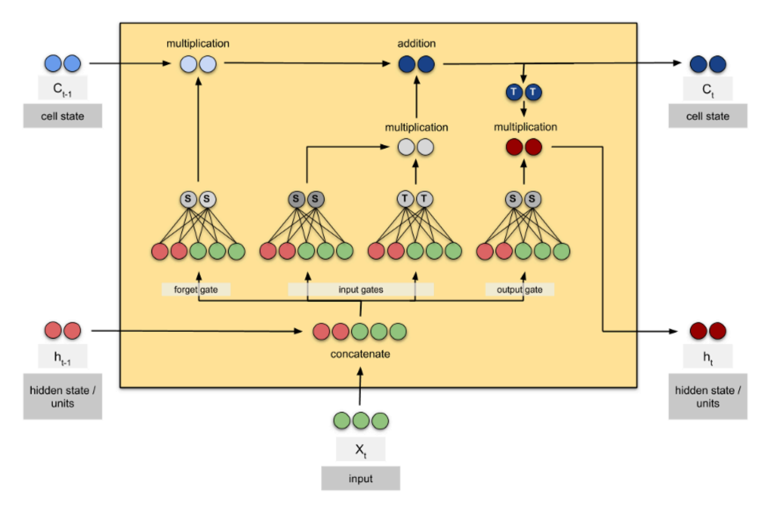
- Gated RNN can alleviate the problems of vanilla RNN
- A simple gated RNN takes the following form (g_t is the added gate):
  - $g_t = f_1(W_{g,h}h_{t-1} + W_{g,x}x_t)$
  - $h_t = (1 - g_t)h_{t-1} + g_t f_2(W_{h,h}h_{t-1} + W_{h,x}x_t)$
- Popular gated RNNs include LSTM and GRU

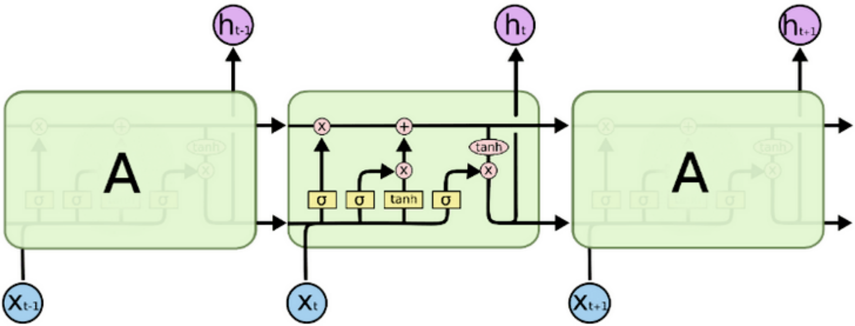
- Hochreiter and Schmidhuber, Long short-term memory, 1997
- LSTMs can learn long-term dependencies much better than vanilla RNNs

- **Forget gate**
  - A neural network layer; forget some information from the previous state
  - $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$; σ is the Sigmoid function
- **Input gate**
  - A neural network layer; add some information from the current input
  - $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- **Output gate**
  - A neural network layer; decide the output
  - $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
- **Memory cell**
  - A neural network layer; flow information from one LSTM cell to the next LSTM cell
  - $\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
  - $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- **Visible state**
  - $h_t = o_t * tanh(C_t)$

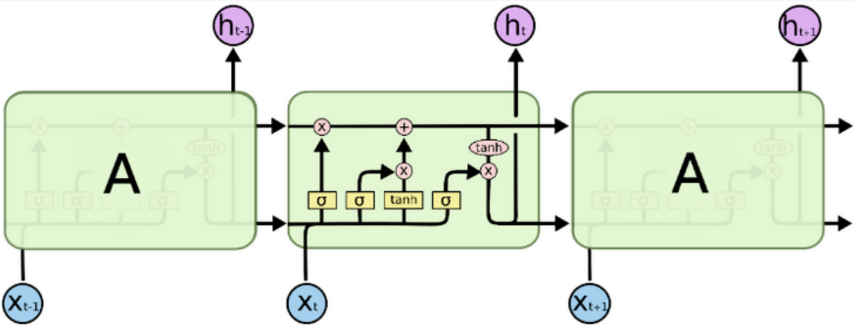

h_t
C_{t-1}
C_t
×
+
tanh
f_t
i_t
×
o_t
×
$\tilde{C}_t$
σ
σ
tanh
σ
h_{t-1}
h_t
x_t
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

- An LSTM with 2 hidden units and an input dimension 3





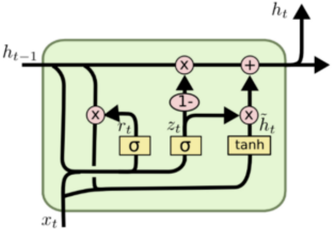
- Number of learnable parameters: $[(\textit{num\_units} + \textit{input\_dim} + 1) * \textit{num\_units}] * 4$



- Number of learnable parameters:
  $[(num\_units + input\_dim + 1) * num\_units] * 4$
- $+1$: bias
- $*4$: four neural network layers

- Compared to vanilla RNN (contains a single neural network layer), LSTM contains 4 neural network layers
- Sigmoid outputs a value between 0 and 1, describing how much information should be let through
  - 0: let nothing through
  - 1: let everything through

- Cho et al., On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, 2014
- 3 neural network layers
- 2 gates (i.e., reset gate and update gate)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- GRU use less training parameters and therefore use less memory, execute faster and train faster than LSTM
- LSTM is more accurate on dataset with longer sequences
- In short, if sequence is large or accuracy is very critical, go for LSTM whereas for less memory consumption and faster operation go for GRU

Example Code

- Simple LSTM (link)
- Stacked LSTM (link)
- CNN + LSTM (link)