S&DS 365 / 665

Intermediate Machine Learning

RNNs, LSTMs, GRUs and All That

April 20

Yale

Home stretch!

- Final exam: Saturday May 7 at 2pm in SPL 59
- Quiz 4 (last!) posted today
  - RL concepts

For Today

Sequence models

- Recurrent neural networks (recap)
- Memory circuits: LSTMs and GRUs
- Sequence to sequence models

Sequence models

- Generative process, any sequence (of words, characters, stock prices, nucleotides...) is assigned a probability

$$p(x_1, \ldots, x_n)$$

which can be factored as

$$p(x_1, \ldots, x_n) = p(x_1)p(x_2 \mid x_1) \ldots p(x_n \mid x_1, \ldots, x_{n-1})$$

Sequence generation

- Items generated one-by-one
- Output at time t chosen by sampling from a probability distribution
- State h_t encodes "features" of sequence $x_1, x_2, \ldots, x_t$
- We talked about state models: HMMs and RNNs

Hidden Markov Model

In an HMM, current output generated from a latent variable.

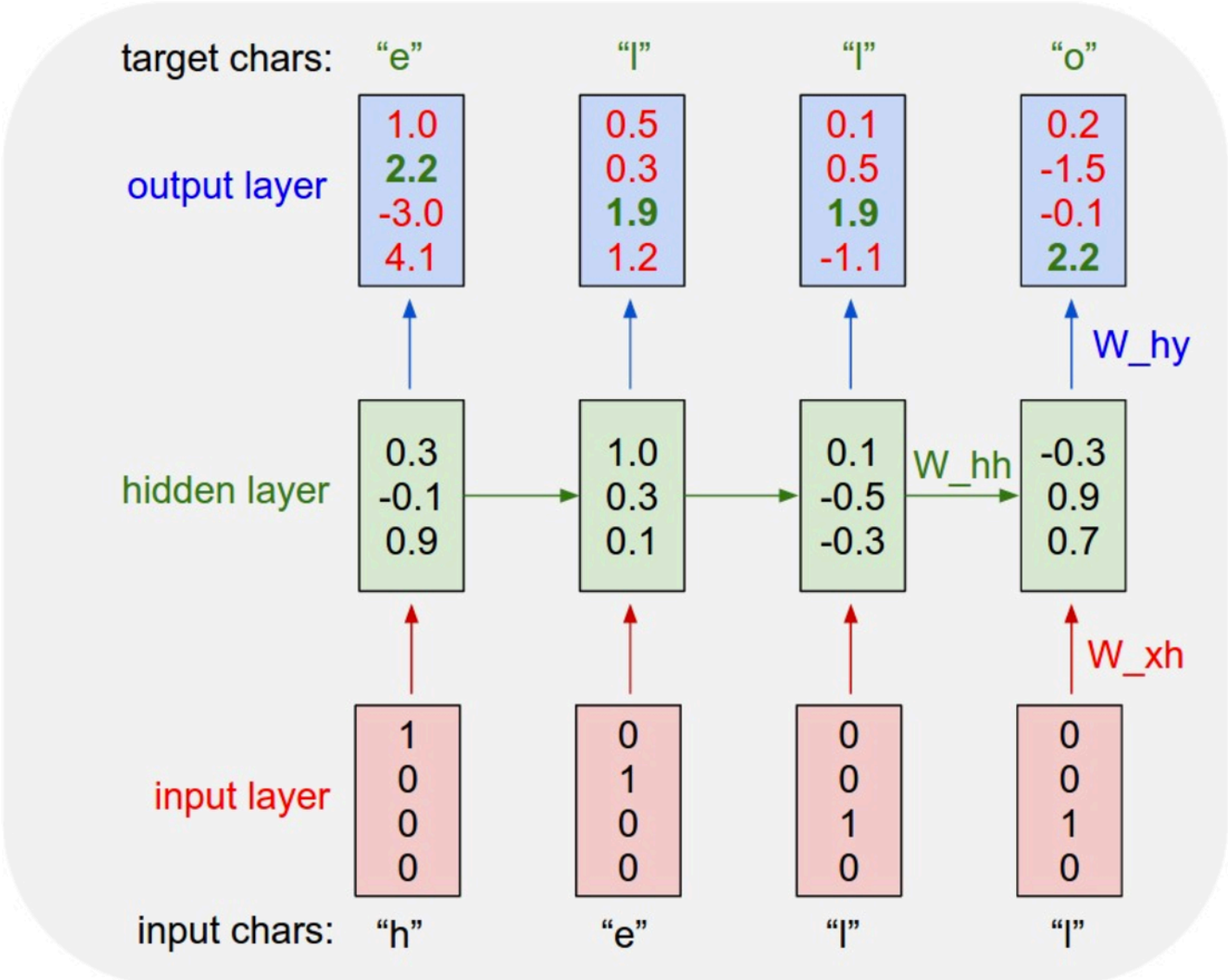
- State is chosen stochastically (that is, probabilistically, or randomly)
- As a consequence, the dependence on earlier x_ts extends much further back in time.

Recurrent neural network

In an RNN, current output generated from a distributed state—a vector of neurons

- State is a deterministic, a nonlinear function of previous state and current input symbol.
- Dependence on earlier x_ts is encoded in the state

RNNs http://karpathy.github.io/2015/05/21/rnn-effectiveness/



RNNs

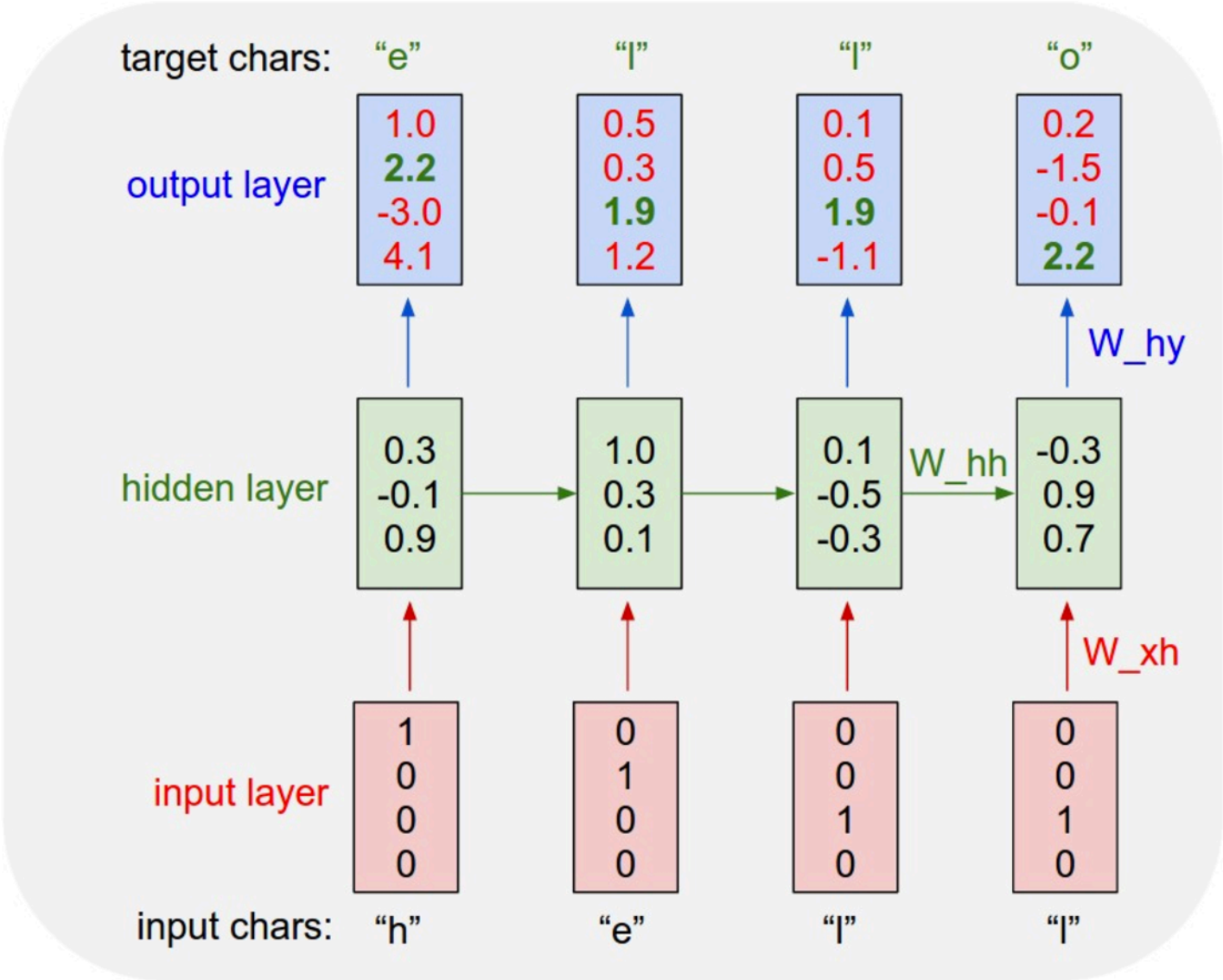
This means

$$
\begin{aligned}
h_t &= \tanh\left(W_{hh} h_{t-1} + W_{xh} x_t\right) \\
y_t &= W_{hy} h_t \\
x_{t+1} \mid y_t &\sim \text{Multinomial}(\pi(y_t))
\end{aligned}
$$

where $\pi(\cdot)$ is the soft-max function.

In this illustration, x_t is the "1-hot" representation of a character, $W_{xh} \in \mathbb{R}^{3\times 4}$, $W_{hh} \in \mathbb{R}^{3\times 3}$ and $W_{hy} \in \mathbb{R}^{4\times 3}$.

RNN architecture



RNN Python Code

```
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
```

RNN Python Code

```
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
```

One hidden layer:

```
rnn = RNN()
y = rnn.step(x) # x is an input vector, y is the RNN's output vector
```

RNN Python Code

```
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
```

Two hidden layers:

```
y1 = rnn1.step(x)
y = rnn2.step(y1)
```

RNN examples

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Fake νs with topics

"optimization"

$$X_L = -\frac{1}{\sum_i y_i}\pi(y_i, x_i, x > 0) \text{ subject to } x_i^T X^H x_i \leq \epsilon^2$$

$$\text{minimize } -\sum_{i=1}^{K}\sum_{c=1}^{Z} a_{mij}\|g_i\|_2^2$$

Fake νs with topics

"physics"

$$\hat{b}_{\mathrm{prc}} = \frac{\hbar^2}{R} - \mu_{\mathrm{sit}} F_{d}$$

$$\frac{\partial \delta C}{\partial r} Re_{a} a_{\mu} - \int_{V_{bor,quet}} \left\langle K^{\alpha\beta} \right\rangle_{\alpha}^{\beta} = \mathcal{H} Er + \beta_{\alpha\mu\nu}^{\top} R_{\mathrm{conseds}}^{\mu\lambda} \partial_{\mu}$$

$$\rho_{deg} \propto 11eV \left[L(003B)/20 \pm 10ifs.1.8V(10^{-2}ergs^{-1})/5; 1.13neV \right]$$

rnn-demo.ipynb: numpy version

We'll train an "np-complete" RNN on the complete works of William Shakespeare, downloaded from Project Gutenberg (specifically, this link).

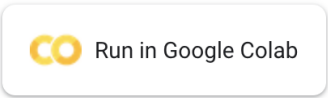
This uses a simple, direct implementation of backpropagation for RNNs, paralleling what we did for simple feedforward networks.

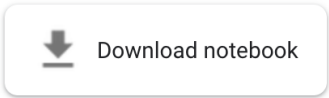
In [ ]:

```
def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0

  # forward pass
  for t in range(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)

  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(range(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-stu
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)

  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients

  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

Tensorflow implementation

Text generation with an RNN





This tutorial demonstrates how to generate text using a character-based RNN. You will work with a dataset of Shakespeare's writing from Andrej Karpathy's The Unreasonable Effectiveness of Recurrent Neural Networks. Given a sequence of characters from this data ("Shakespear"), train a model to predict the next character in the sequence ("e"). Longer sequences of text can be generated by calling the model repeatedly.

Note: Enable GPU acceleration to execute this notebook faster. In Colab: *Runtime > Change runtime type > Hardware accelerator > GPU*. If running locally make sure TensorFlow version >= 1.11.

This tutorial includes runnable code implemented using tf.keras and eager execution. The following is sample output when the model in this tutorial trained for 30 epochs, and started with the string "Q":

```
QUEENE:
I had thought thou hadst a Roman; for the oracle,
Thus by All bids the man against the word,
Which are so weak of care, by old care done;
Your children were in your holy love,
And the precipitation through the bleeding throne.

BISHOP OF ELY:
Marry, and will, my lord, to weep in such a one were prettiest;
Yet now I was adopted heir
Of the world's lamentable day,
To watch the next way with his father with his face?

ESCALUS:
The cause why then we are all resolved more sons.
```

Memory circuits

A variant called “Long Short-Term Memory” RNNs has a special hidden later that “includes” or “forgets” information from the past.

Intuition: In language modeling, may be useful to remember/forget gender or number of subject so that personal pronouns (“he” vs. ”she” vs. ”they”) can be used appropriately.

Useful for things like matching parentheses, etc.

A simpler alternative to the LSTM circuit is called the *Gated Recurrent Unit* (GRU)

“Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”, Cho et al. 2014
https://arxiv.org/pdf/1406.1078v3.pdf

Vanilla RNNs

In principle the state h_t can carry information from far in the past.

In practice, the gradients vanish (or explode) so this doesn't really happen

We need other mechanisms to "remember" information from far away and use it to predict future words

Vanilla RNNs

State is updated according to

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{hx}x_t + b_h\right)$$

We'll modify this with two types of "neural circuits" for storing information to be used downstream

LSTMs and GRUs

Both LSTMs and GRUs have longer-range dependencies than vanilla RNNs.

We'll go through this in detail for GRUs, which are simpler, more efficient, and more commonly used

Hadamard product

We'll need to use pointwise products. This is given the fancy name "Hadamard product" and written $\odot$:

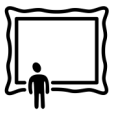
$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \odot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 4 \\ 10 \\ 18 \end{pmatrix}$$

Hadamard product

We'll need to use pointwise products. This is given the fancy name "Hadamard product" and written $\odot$:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \odot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 0 \\ 4 \end{pmatrix}$$

Gated recurrent units (GRUs)



High level idea:

- Learn when to update hidden state to “remember” important pieces of information
- Keep them in memory until they are used
- Reset or “forget” this information when no longer useful

GRUs (simplified)

We'll first describe a simplified version of GRUs.

We'll make use of an update "gate" Γ_t^u. This is either zero or one.

If $\Gamma_t^u = 1$ we store/update information in a memory cell c_t.

If $\Gamma_t^u = 0$ we just keep the information in the cell; don't update it.

GRUs (simplified)

The memory cell evolves according to

$$c_t = (1 - \Gamma_t^u) \odot c_{t-1} + \Gamma_t^u \odot h_t$$

where h_t is the state computed using the usual "vanilla RNN".

We predict the next words using the memory cell c_t together with the usual RNN state h_t.

GRUs (simplified)

Everything needs to be differentiable, so the gate is actually "soft" and between zero and one.

Our equations are then

$$\Gamma_t^u = \sigma\left(W_{ux} x_t + W_{uh} h_{t-1} + b_u\right)$$

$$h_t = \tanh\left(W_{hx} x_t + W_{hh} h_{t-1} + b_h\right)$$

$$c_t = \left(1 - \Gamma_t^u\right) \odot c_{t-1} + \Gamma_t^u \odot h_t$$

where σ is the sigmoid function.

We predict the next words using the memory cell c_t together with the usual RNN state h_t.

GRUs: Example

Example:

`The flowers, even though it was still so, so cold outside in New Haven, bloomed in April`

We want to keep `flowers` in memory. It's the subject of the sentence, and plural.

It also has a meaning that is relevant when we predict the words `bloomed` and `April`.

Let's step through on the blackboard how the GRU handles this.

GRUs: Example

These are the main ideas behind GRUs.

A more commonly used version is a little bit more complex, but still based on the same idea

GRUs (slightly more complex)

This version uses two gates.

One gate Γ^u to update the memory cell.
Another gate Γ^r to decide if the state should be "reset"

The new equations are

$$\Gamma_t^u = \sigma\left(W_{ux} x_t + W_{uh} h_{t-1} + b_u\right)$$

$$\Gamma_t^r = \sigma\left(W_{rx} x_t + W_{rh} h_{t-1} + b_r\right)$$

$$h_t = \tanh\left(W_{hx} x_t + W_{hh}\left(\Gamma_t^r \odot h_{t-1}\right) + b_h\right)$$

$$c_t = (1 - \Gamma_t^u) \odot c_{t-1} + \Gamma_t^u \odot h_t$$

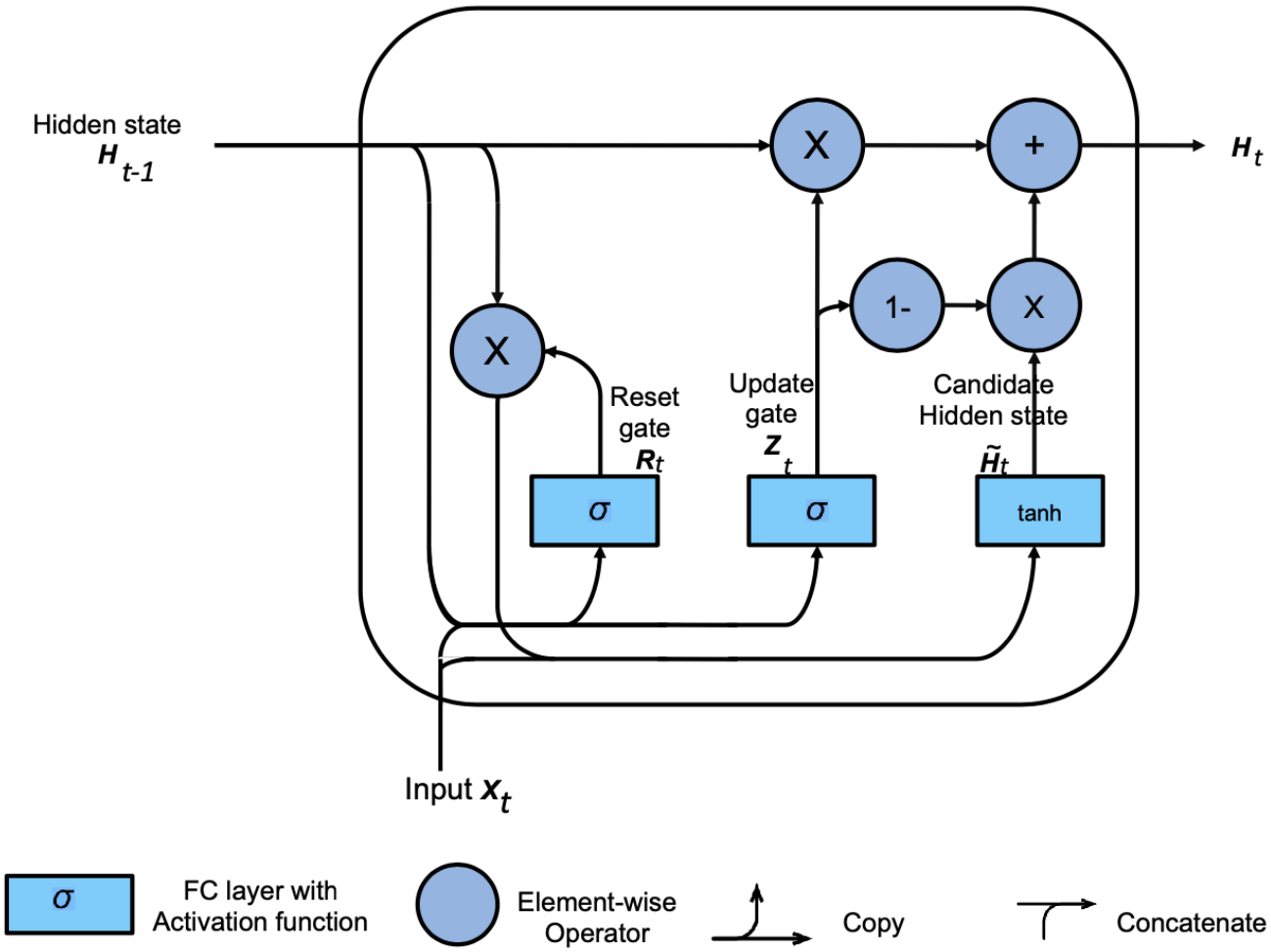
GRUs

If entries of the gate Γ_t^r are close to 1, acts just like the vanilla RNN update rule

If entries are close to 0, the acts serves to “reset” the state, capturing new short-term information

As long as $\Gamma_t^u = 0$, the memory cell is propagated over long distances, without vanishing gradients.

GRU Diagram (using slightly different notation)



LSTMs

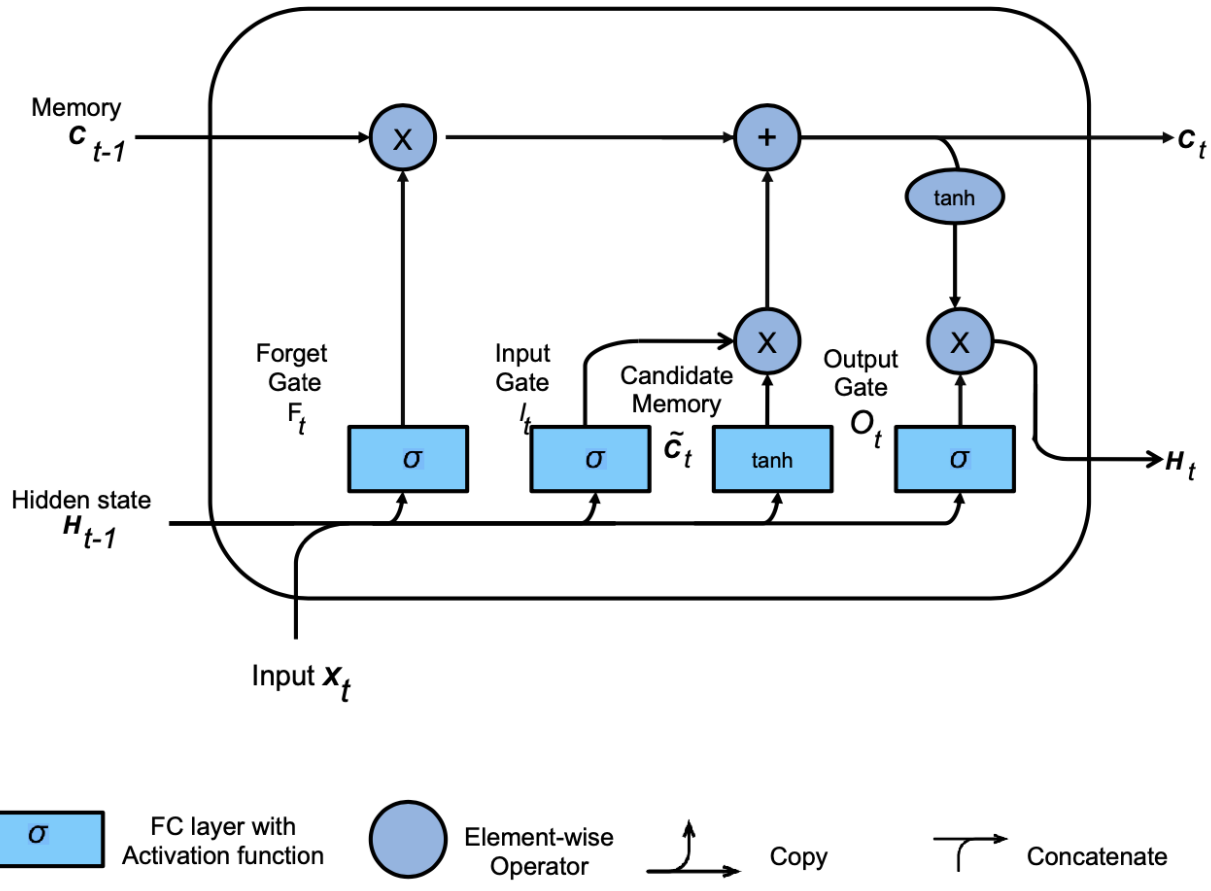
A variant called “Long Short-Term Memory” RNNs has a special context/hidden layer that “includes” or “forgets” information from the past.

$$\text{forget: } F_t = \sigma\left(W_{fh} h_{t-1} + W_{fx} x_t + b_f\right)$$
$$\text{include: } I_t = \sigma\left(W_{ih} h_{t-1} + W_{ix} x_t + b_i\right)$$

“Memory cell” or “context” c_t evolves according to

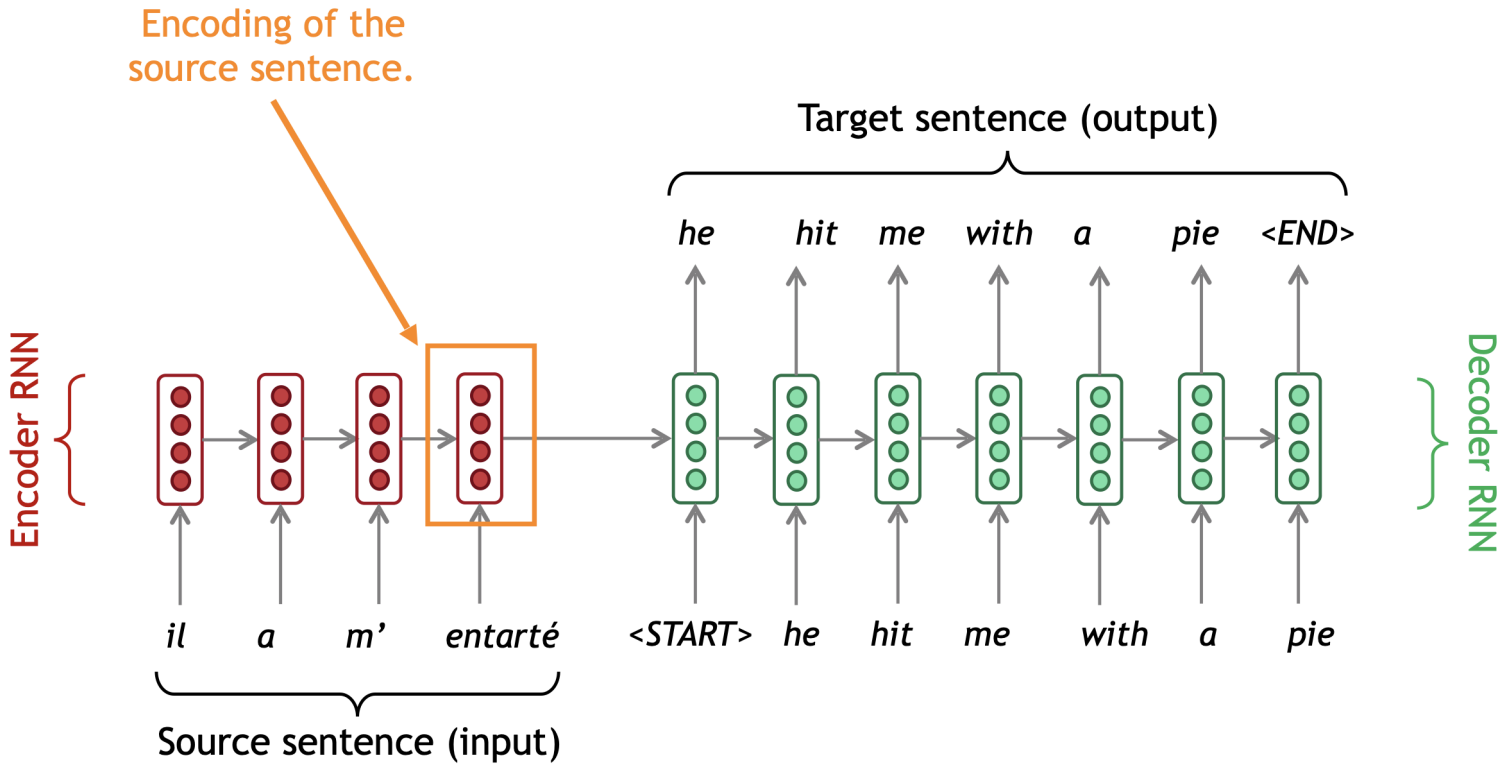
$$c_t = F_t \odot c_{t-1} + I_t \odot \widetilde{c}_t$$
$$\widetilde{c}_t = \tanh\left(W_{ch} h_{t-1} + W_{cx} x_t + b_c\right)$$

LSTM Diagram (using slightly different notation)



Sequence-to-sequence models

- Important in translation
- Uses two RNNs (GRUs or LSTMs): Encoder and Decoder



"Sequence to sequence learning with neural networks," Sutskever et al. 2014, https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf. Figure source: http://web.stanford.edu/class/cs224n/slides/cs224n-2020-lecture08-nmt.pdf

Sequence-to-sequence models

- The goal of Seq2seq is to estimate the conditional probability

$$p(y_1, \ldots, y_T \mid x_1, \ldots, x_S)$$

- Encoder RNN computes the fixed dimensional representation $h(x)$; decoder RNN then computes

$$\prod_{t=1}^{T} p(y_t \mid h, y_1, \ldots, y_{t-1})$$

- Original paper uses 4-layer LSTMs with 1000 neurons in each layer; trained for 10 days.

Recall: This is very similar to what we did with LaTeX equation models conditioned on topics.

Sequence-to-sequence models

This results in a “bottleneck” problem—all the information needs to be funneled through that final state.

To be continued next time!

Summary

- Recurrent neural networks are used for sequential data
- LSTMs and GRUs model longer range dependencies through a “memory” cell
- RNNs are unsupervised, generative models, able to generate realistic looking sequences when sufficiently well trained.
- Seq2seq pairs together two RNNs, encoding the input sequence as a state, and decoding to generate an output sequence.