Natural Language Processing

AI51701/CSE71001

Lecture 10

10/17/2023

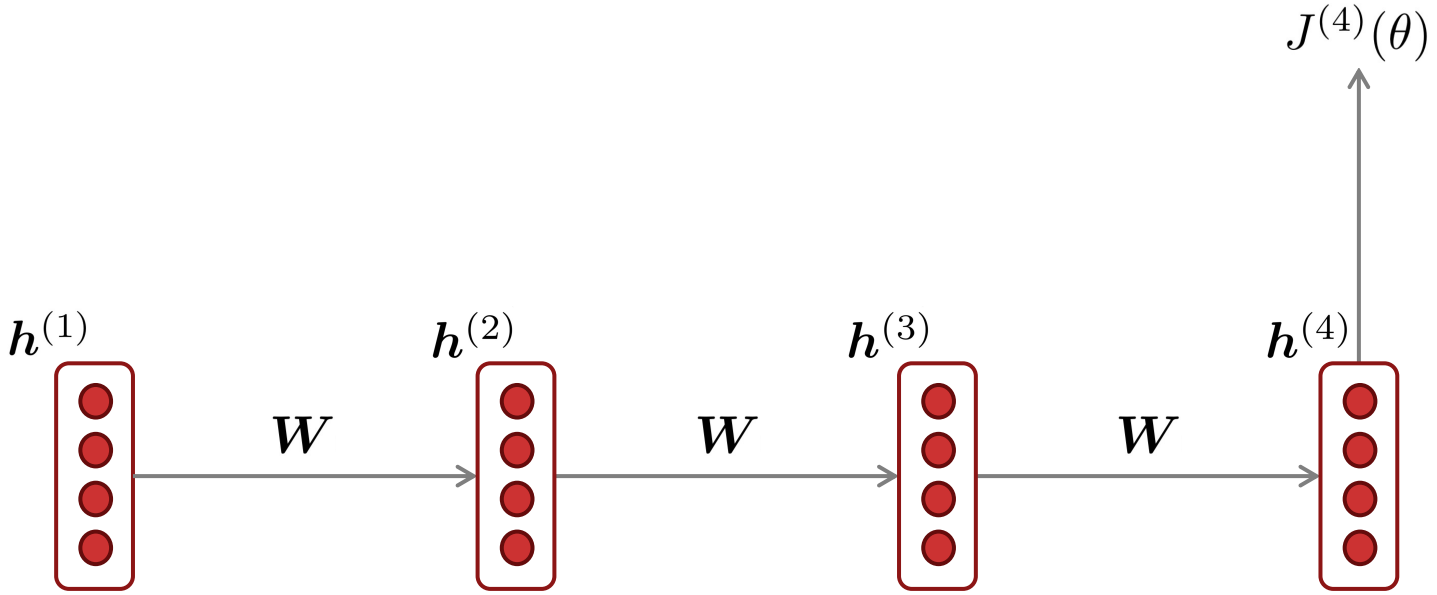
Instructor: Taehwan Kim

Announcements

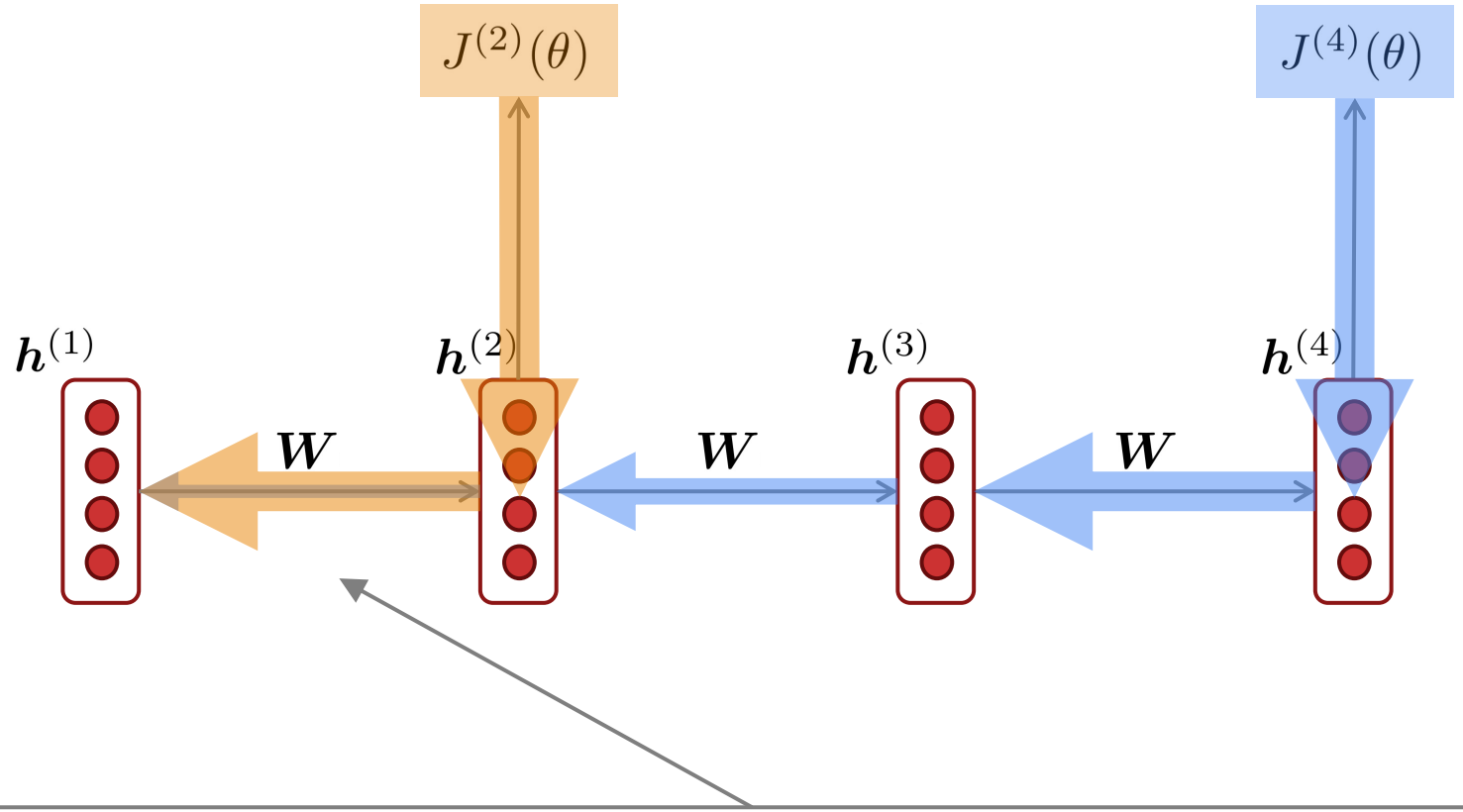
- We have **no class on Oct. 19** (midterm week)

Language Modeling + RNNs

Problems with RNNs: Vanishing and Exploding Gradients



Why is vanishing gradient a problem?



Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

Why is exploding gradient a problem?

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

- This can cause **bad updates**: we take too large a step and reach a weird and bad parameter configuration (with large loss)
- In the worst case, this will result in **Inf** or **NaN** in your network (then you have to restart training from an earlier checkpoint)

Gradient clipping: solution for exploding gradient

- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

- **Intuition**: take a step in the same direction, but a smaller step
- In practice, remembering to clip gradients is important, but exploding gradients are an easy problem to solve

Source: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013

How to fix the vanishing gradient problem?

- The main problem is that *it's too difficult for the RNN to learn to preserve information over many timesteps*.
- In a vanilla RNN, the hidden state is constantly being **rewritten**

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

- How about an RNN with separate **memory** which is added to?

Long Short-Term Memory RNNs (LSTMs)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
  - Everyone cites that paper but really a crucial part of the modern LSTM is from Gers et al. (2000)

- On step t, there is a **hidden state** $h^{(t)}$ *and* a **cell state** $c^{(t)}$
  - Both are vectors length n
  - The cell stores **long-term information**
  - The LSTM can **read**, **erase**, and **write** information from the cell
    - The cell becomes conceptually rather like RAM in a computer

"Long short-term memory", Hochreiter and Schmidhuber, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf
"Learning to Forget: Continual Prediction with LSTM", Gers, Schmidhuber, and Cummins, 2000. https://dl.acm.org/doi/10.1162/089976600300015015

Long Short-Term Memory RNNs (LSTMs)

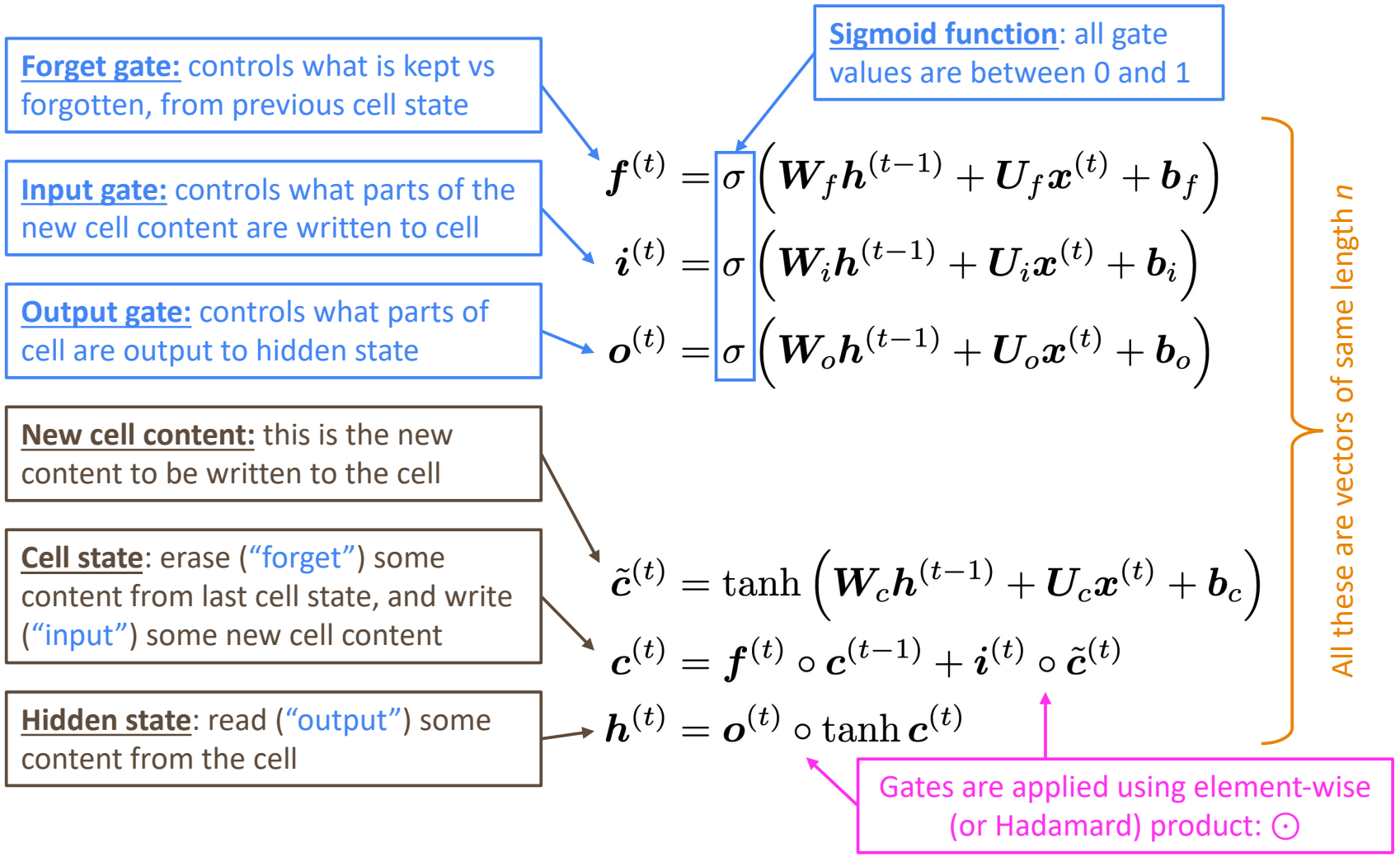
- On step t, there is a **hidden state** $h^{(t)}$ *and* a **cell state** $c^{(t)}$
  - Both are vectors length n
  - The cell stores long-term information
  - The LSTM can read, erase, and write information from the cell
    - The cell becomes conceptually rather like RAM in a computer

- The selection of which information is erased/written/read is controlled by three corresponding gates
  - The gates are also vectors of length n
  - On each timestep, each element of the gates can be **open** (1), **closed** (0), or somewhere in-between
  - The gates are **dynamic**: their value is computed based on the current context

"Long short-term memory", Hochreiter and Schmidhuber, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf
"Learning to Forget: Continual Prediction with LSTM", Gers, Schmidhuber, and Cummins, 2000. https://dl.acm.org/doi/10.1162/089976600300015015

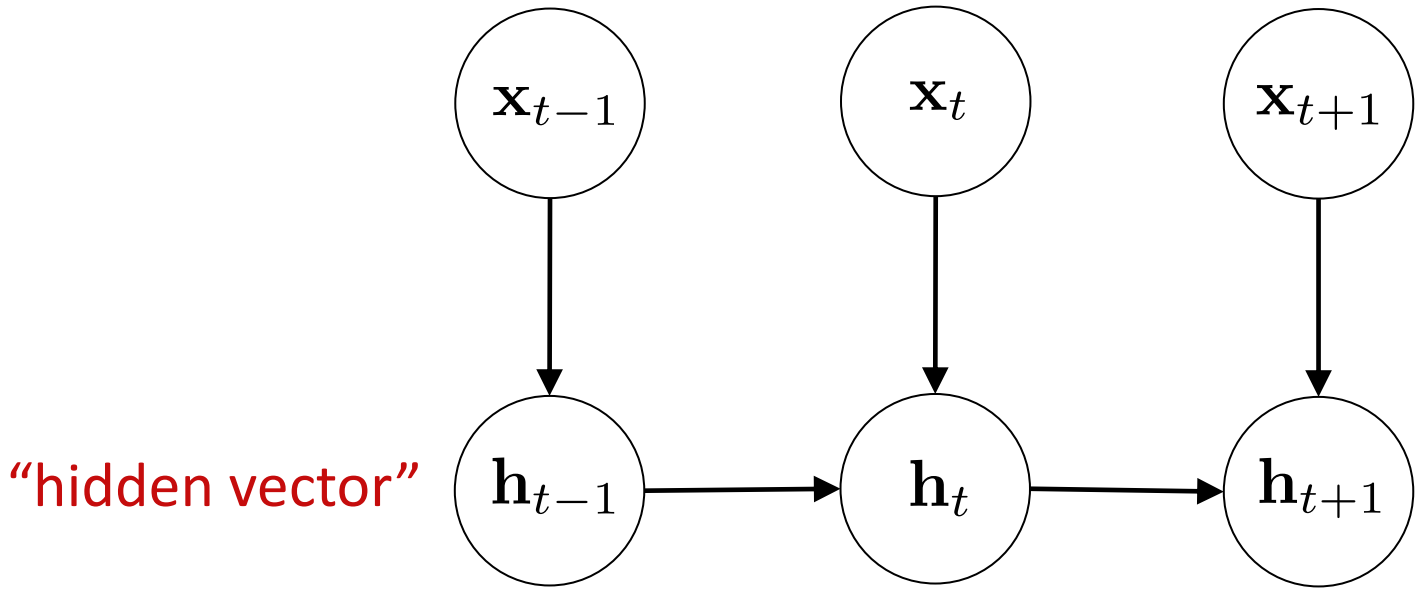
Long Short-Term Memory RNNs (LSTMs)

- We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep t:



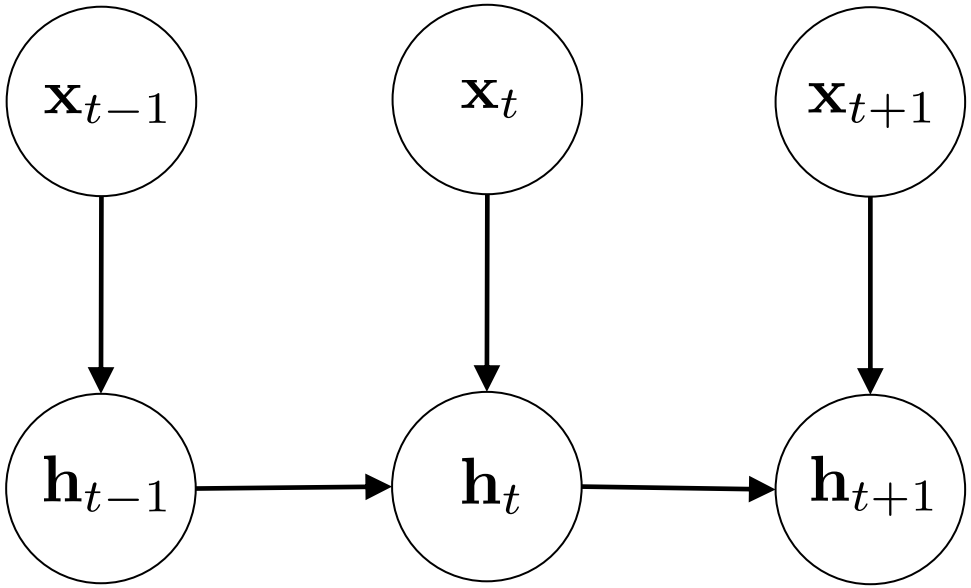
Recurrent Neural Networks

- Input is a sequence:



Recurrent Neural Networks

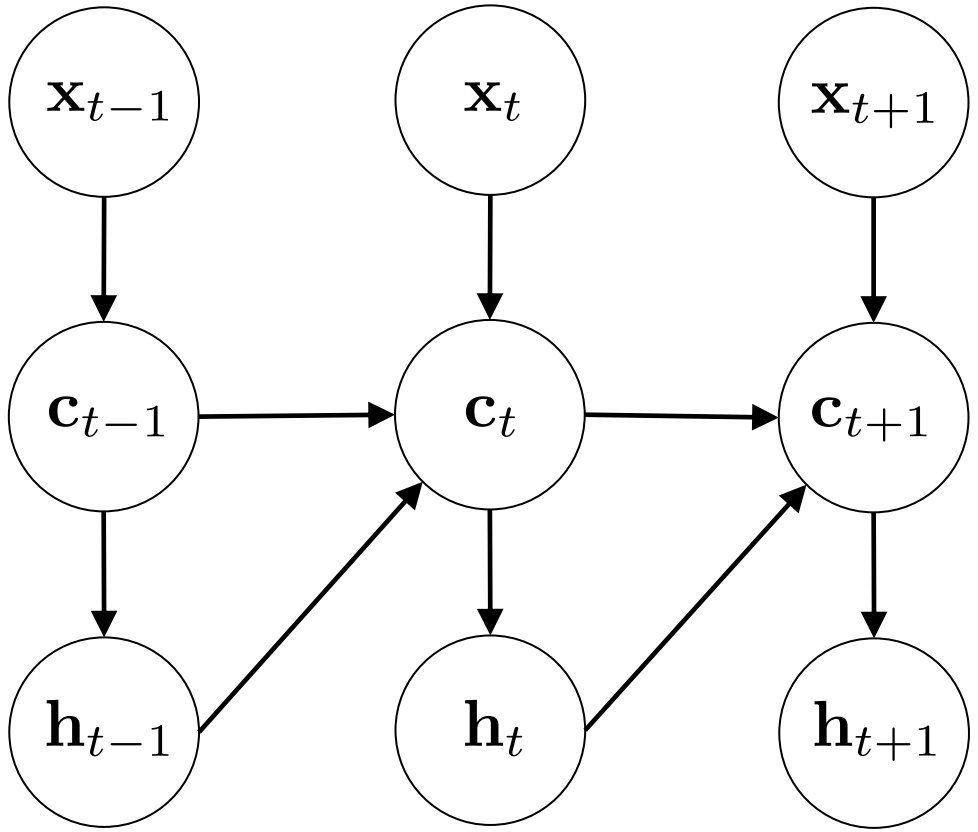
$$\mathbf{h}_t = \tanh\left(\mathbf{W}^{(x)}\mathbf{x}_t + \mathbf{W}^{(h)}\mathbf{h}_{t-1} + \mathbf{b}\right)$$



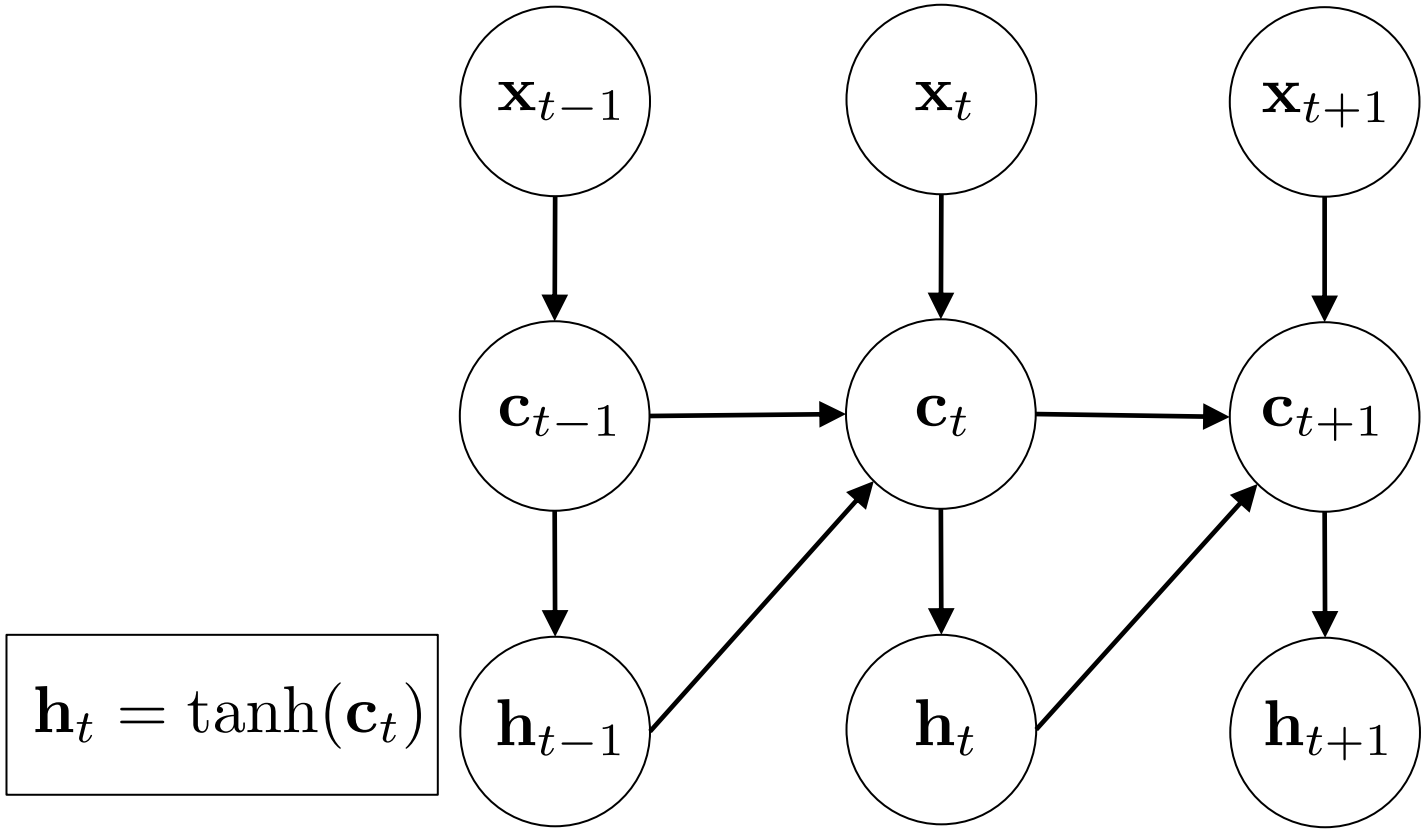
"hidden vector"

Long Short-Term Memory Networks (gateless)

"memory cell"

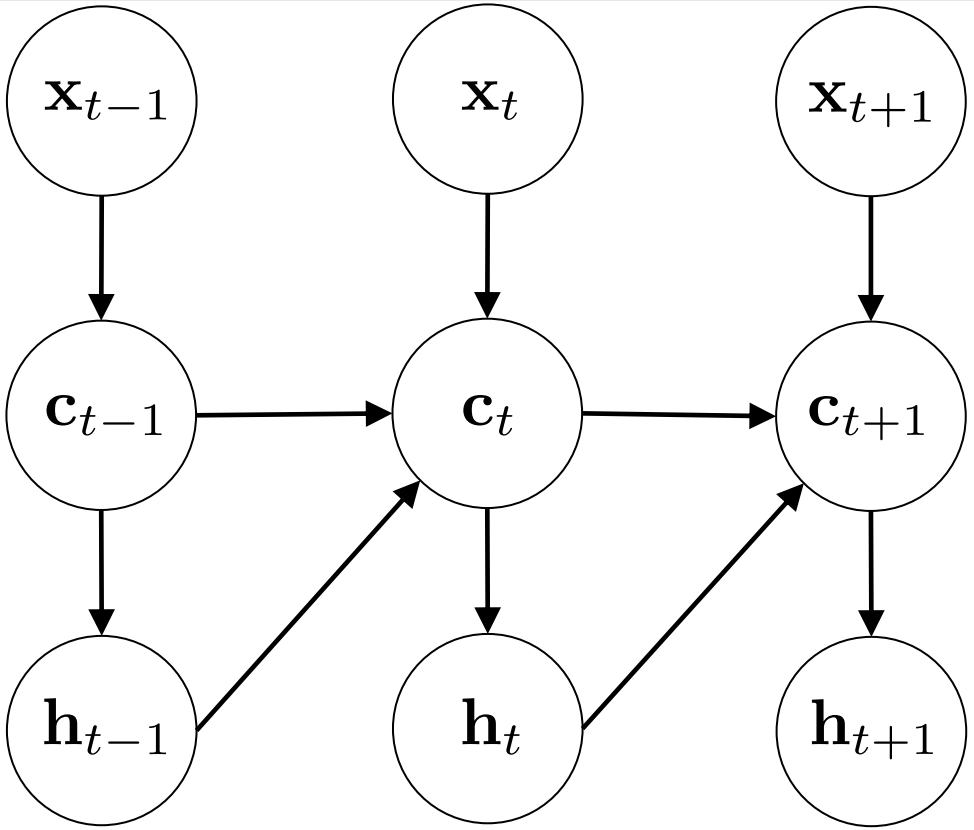


Long Short-Term Memory Networks (gateless)



Long Short-Term Memory Networks (gateless)

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$



$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

Long Short-Term Memory Networks (gateless)

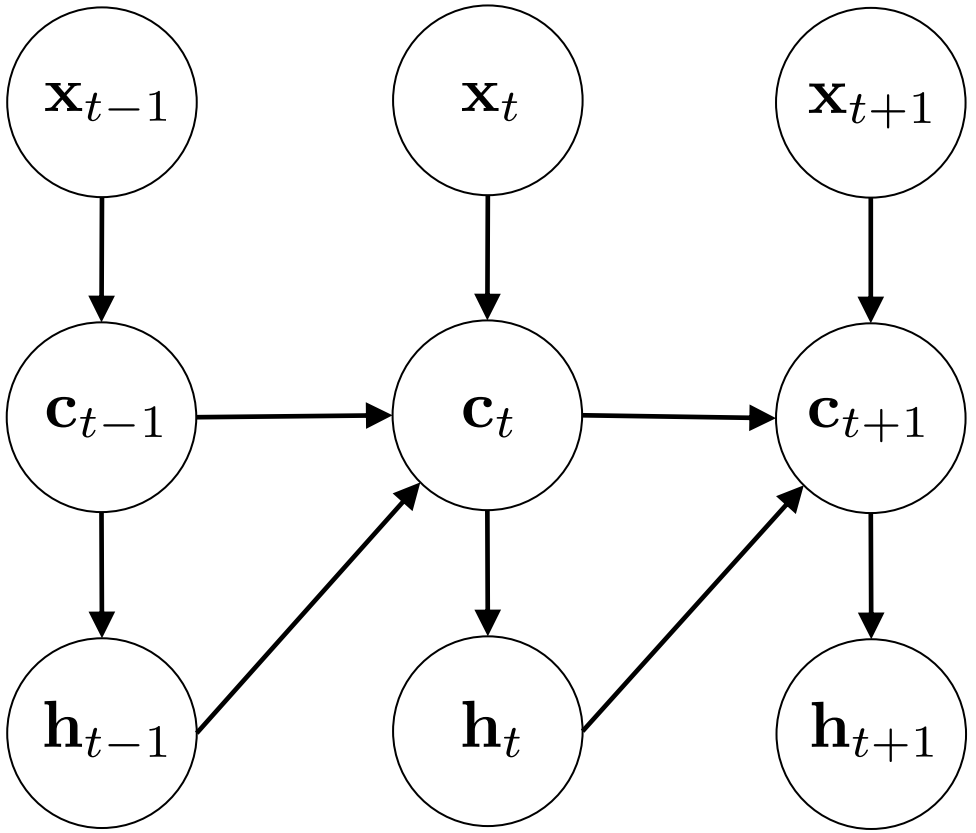
$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

Experiment: text classification

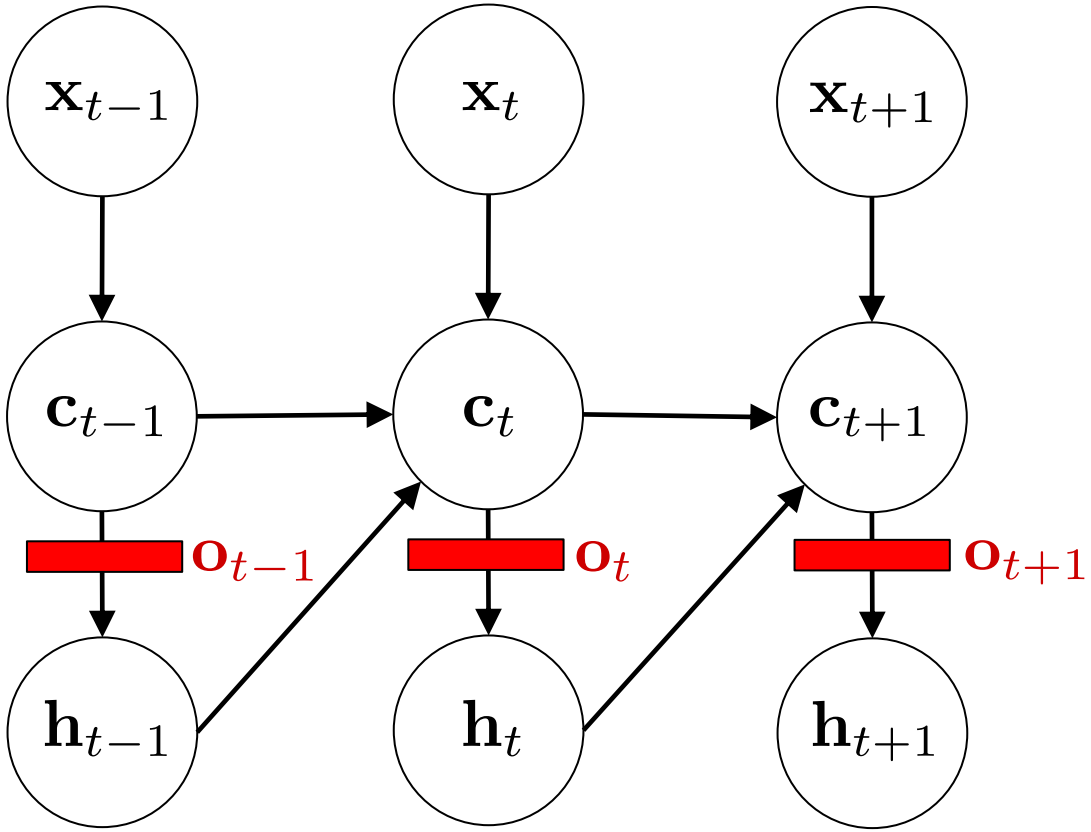
- Stanford Sentiment Treebank
  - binary classification (positive/negative)
- 25-dim word vectors
- 50-dim cell/hidden vectors
- classification layer on **final** hidden vector
- AdaGrad, 10 epochs, mini-batch size 10
- early stopping on dev set

accuracy
80.6

Adding Output Gates

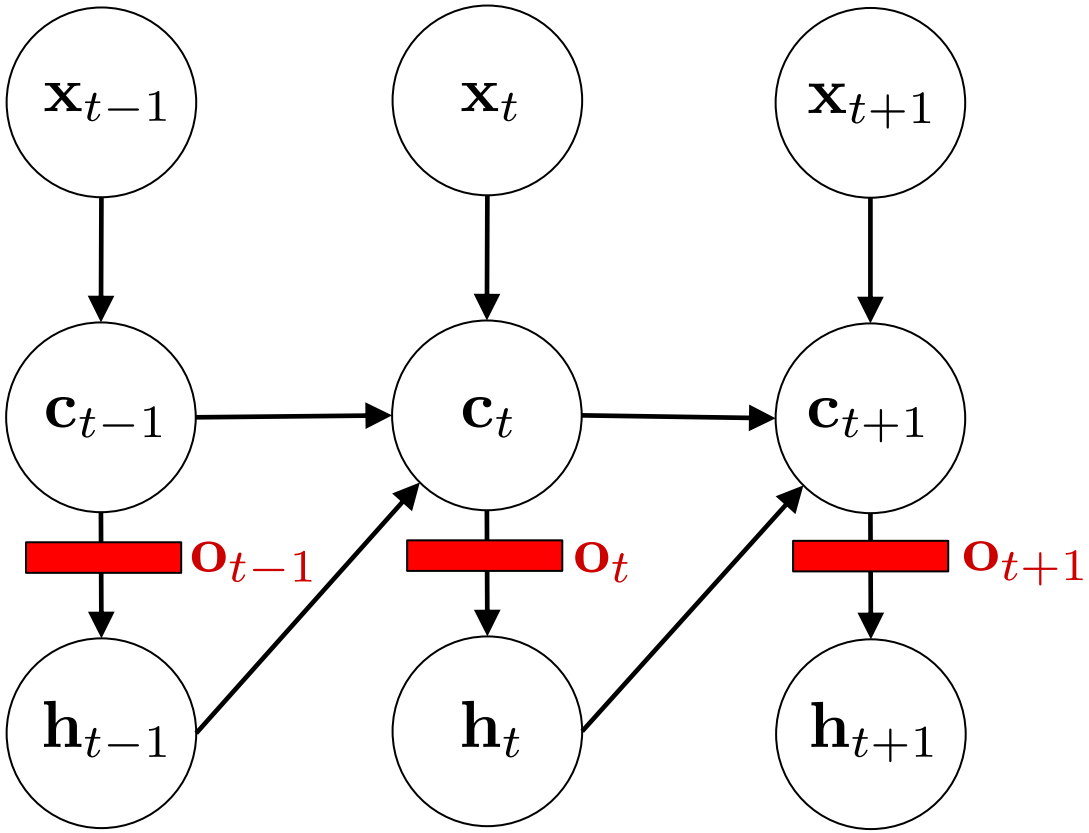


Adding Output Gates



Adding Output Gates

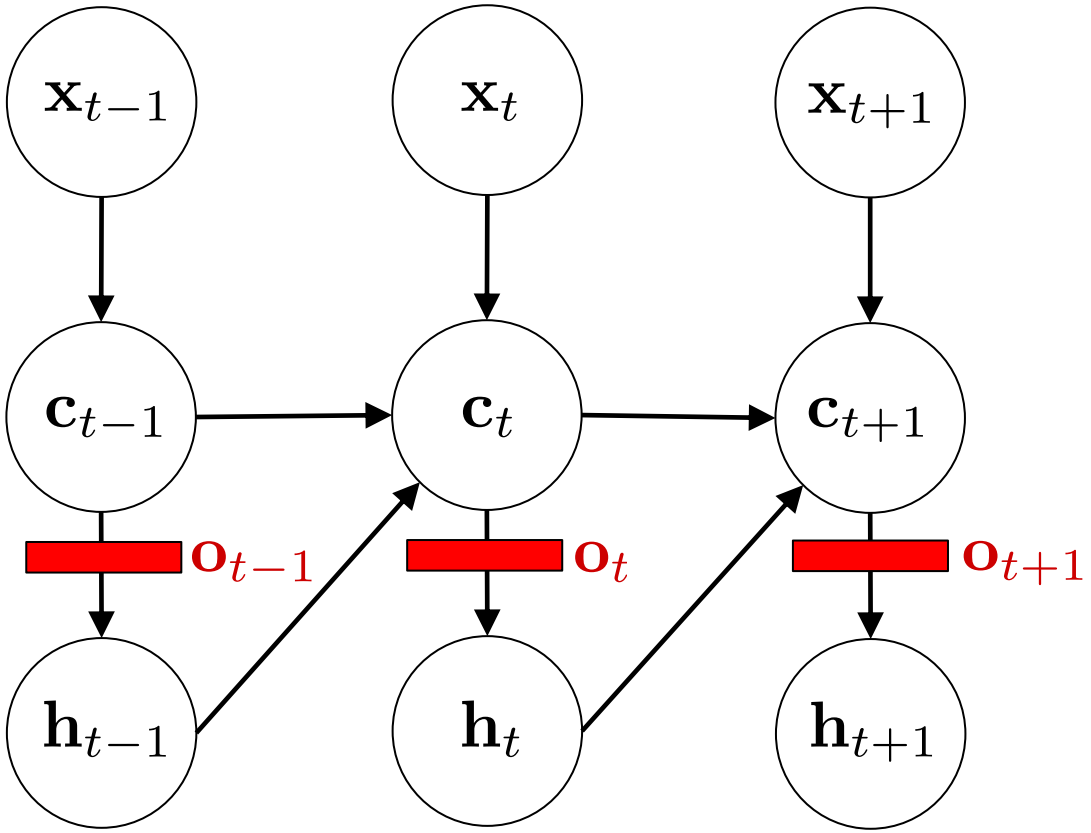
$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$



Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

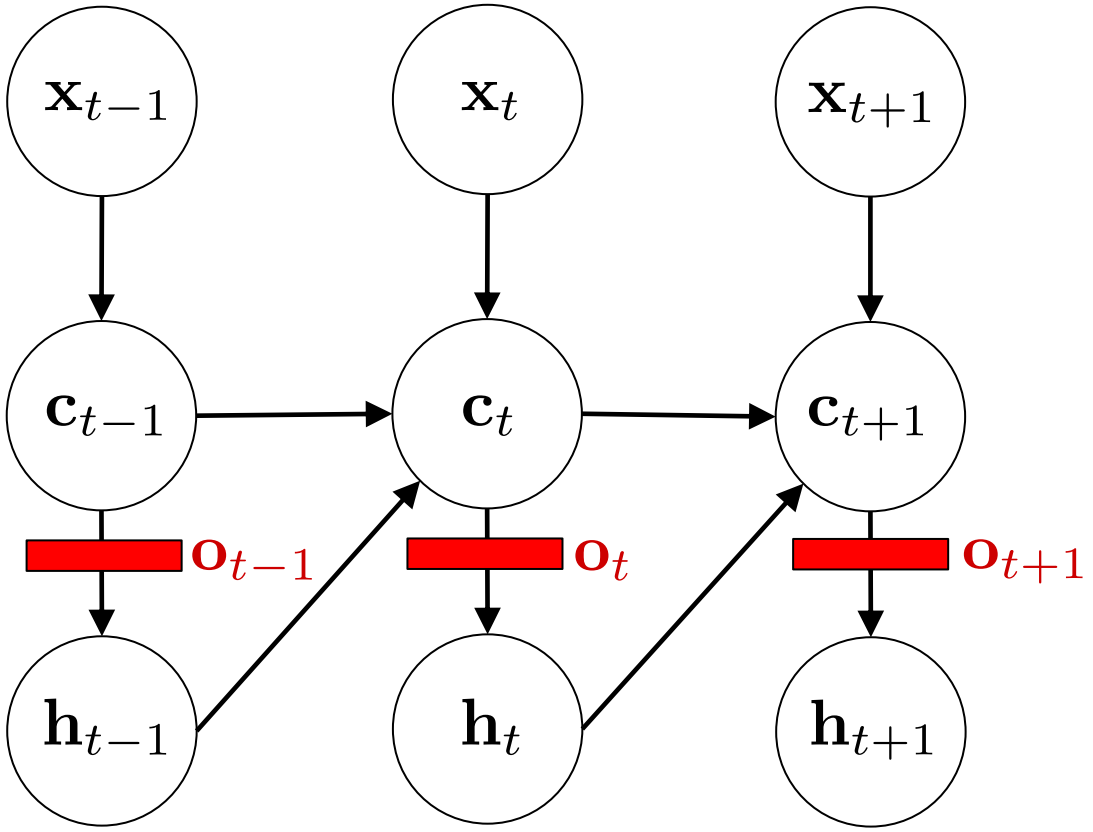


Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

this is pointwise multiplication!
$\mathbf{o}_t$ is a vector

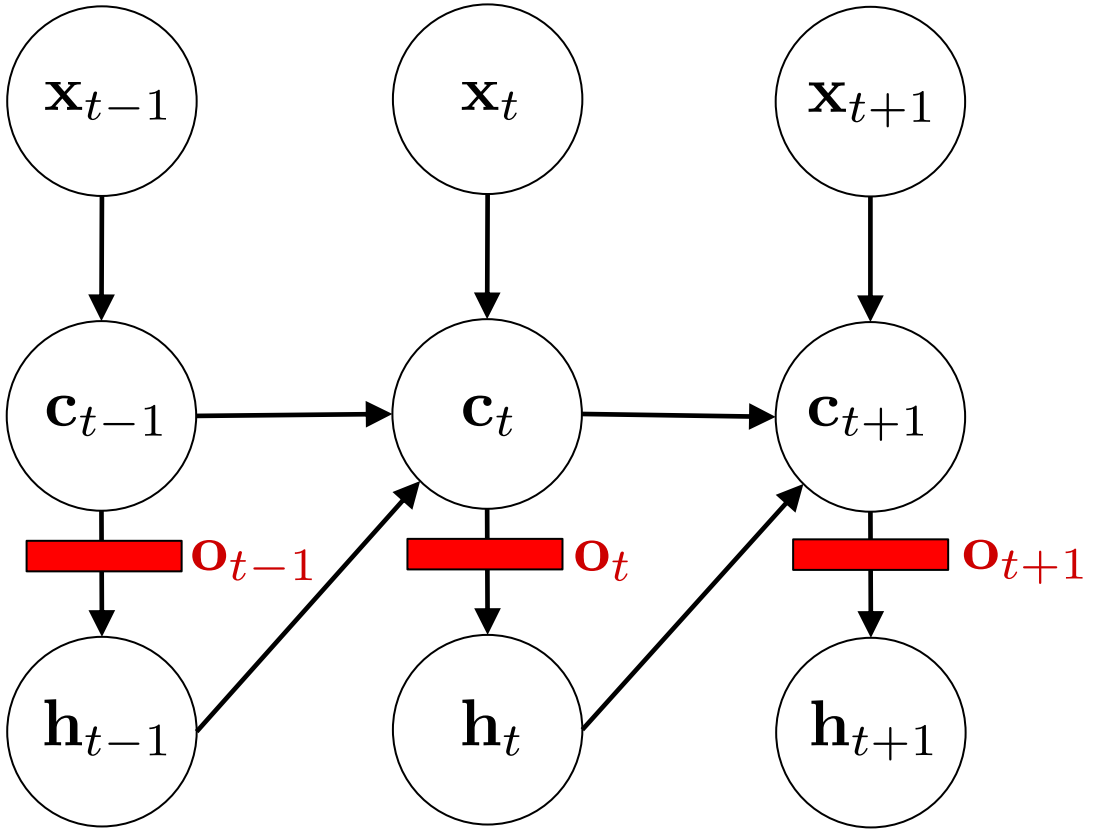


Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

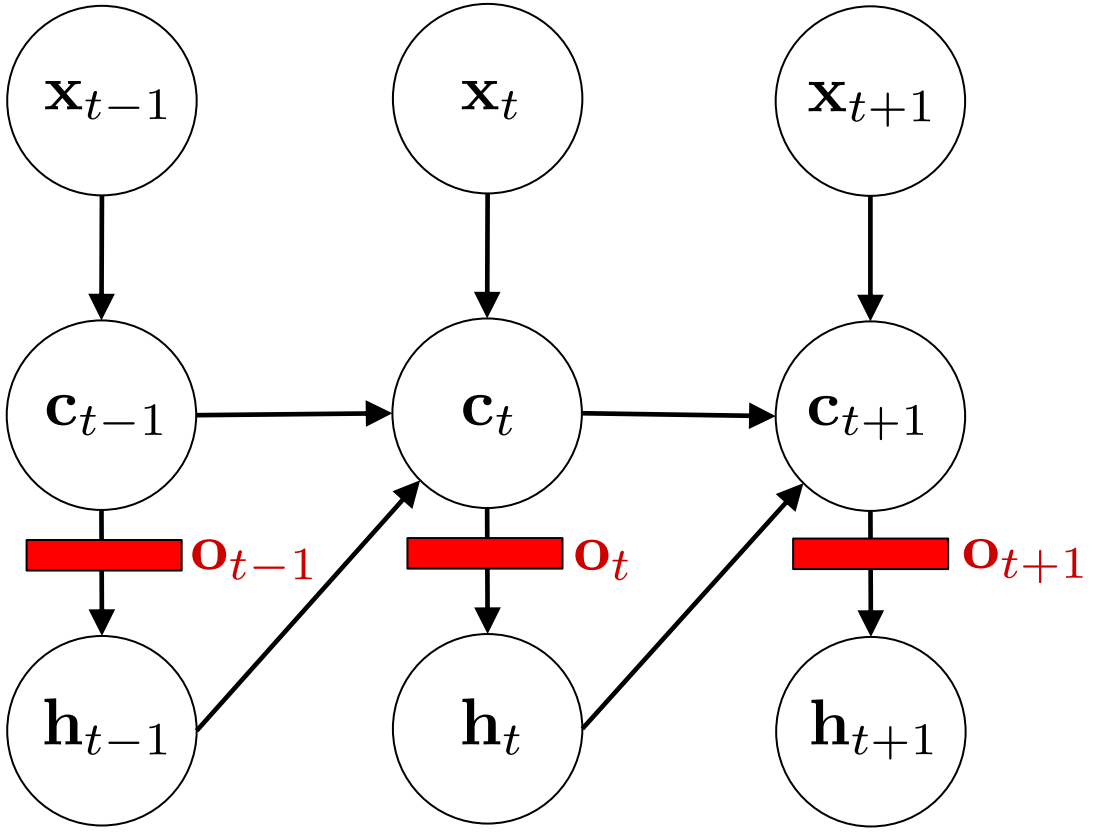
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

output gate affects how much “information” is transmitted from cell vector to hidden vector



Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$



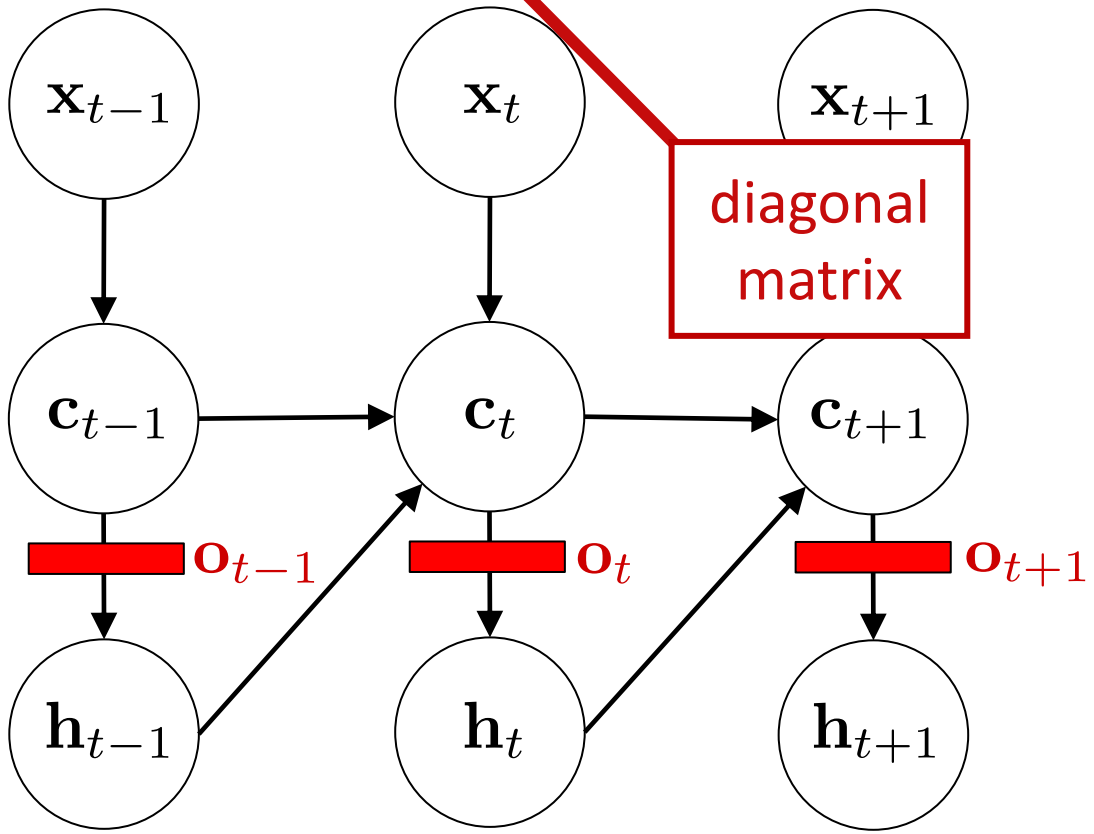
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

logistic sigmoid, so output ranges from 0 to 1

diagonal matrix

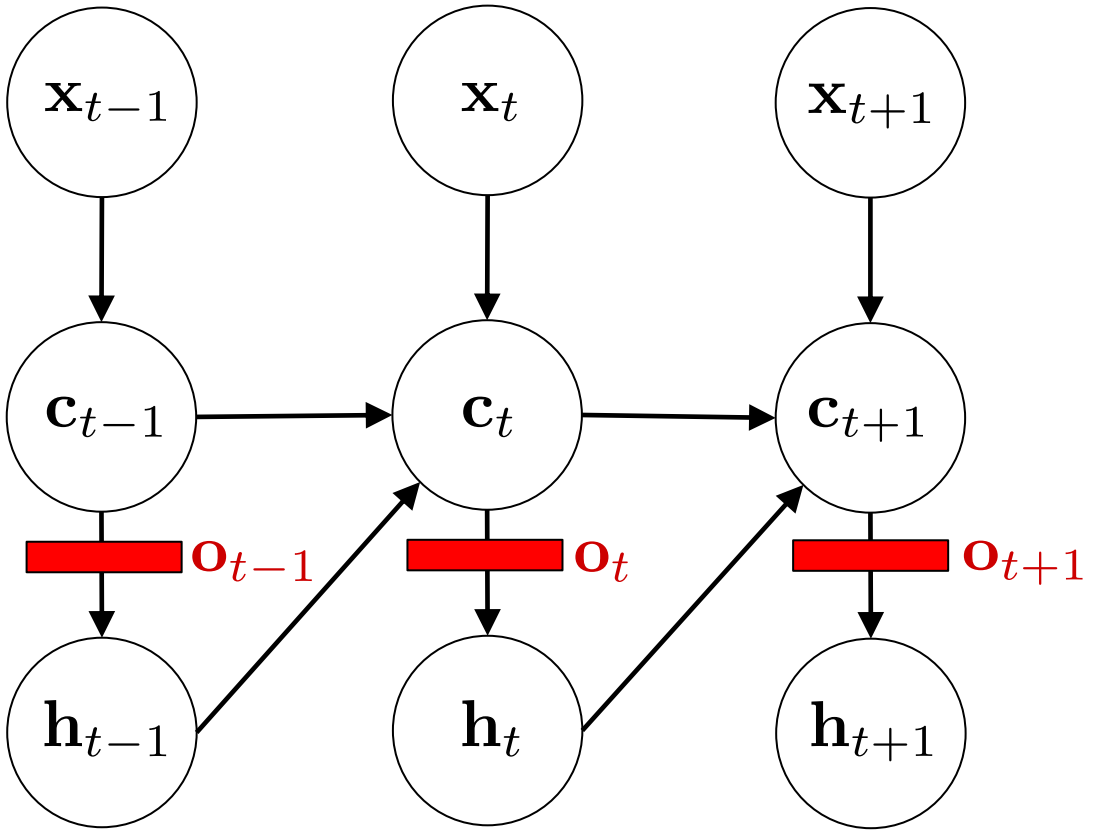


$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

output gate is a function of current observation, previous hidden vector, and current cell vector



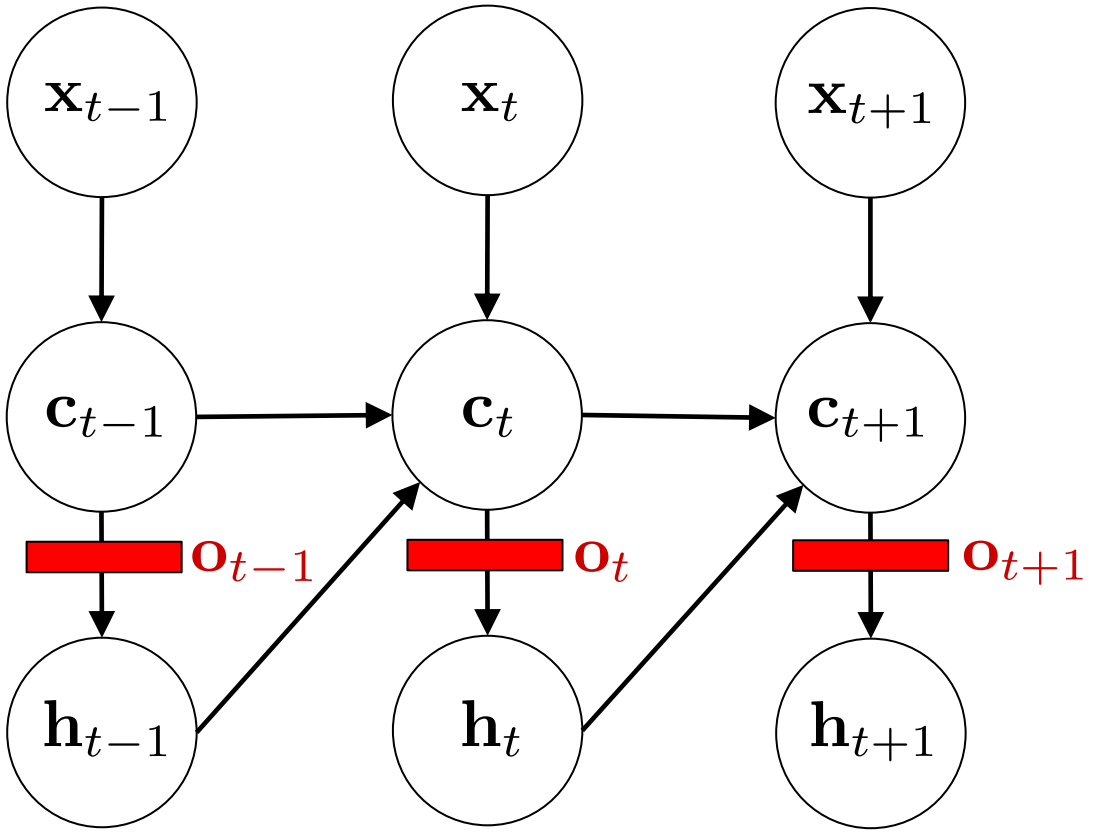
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

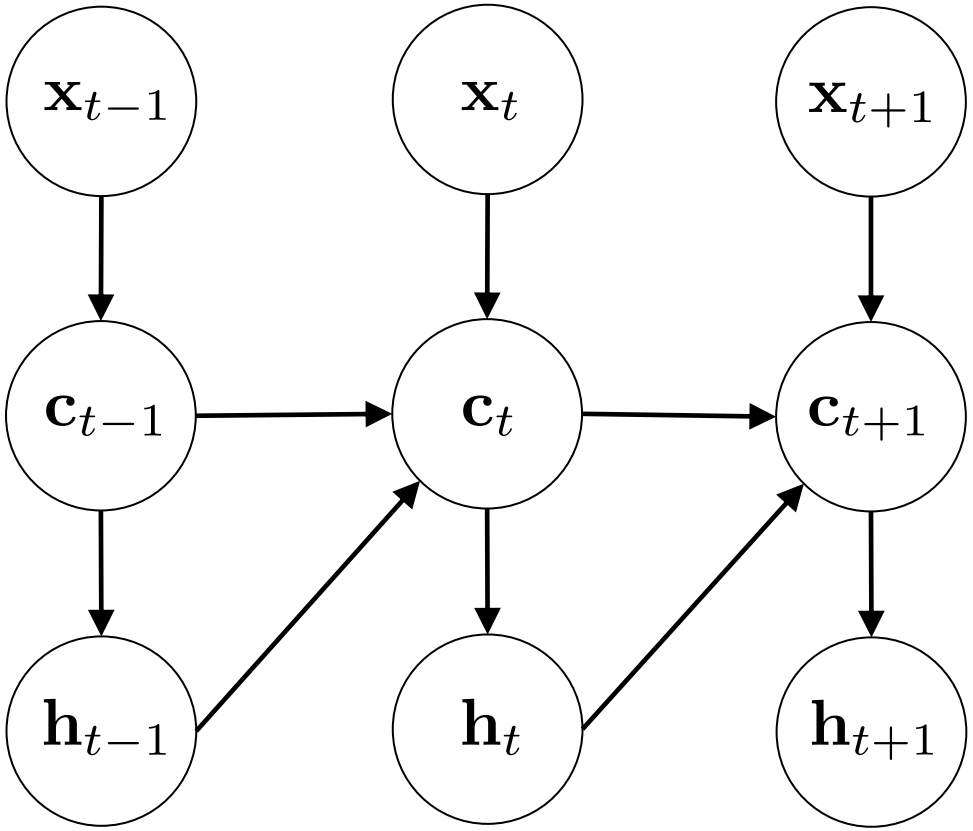
output gate is a function of current observation, previous hidden vector, and current cell vector

	acc.
gateless	80.6
output gates	81.9

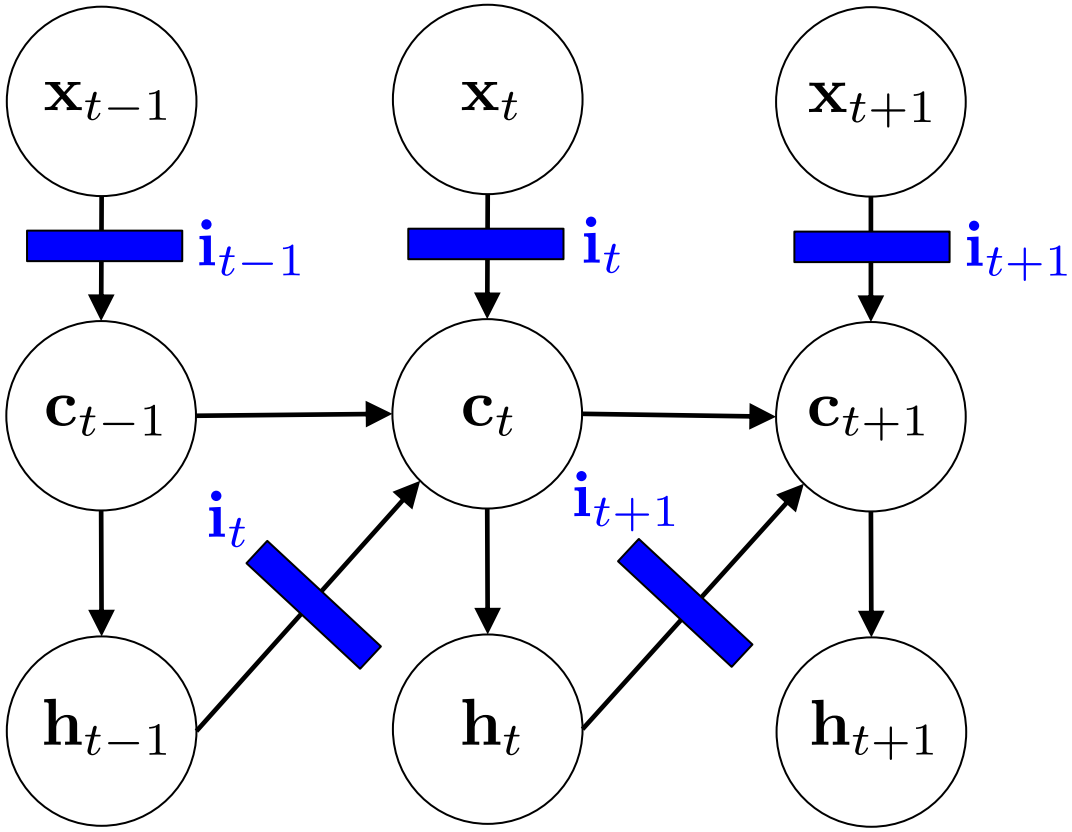


$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Input Gates



Adding Input Gates

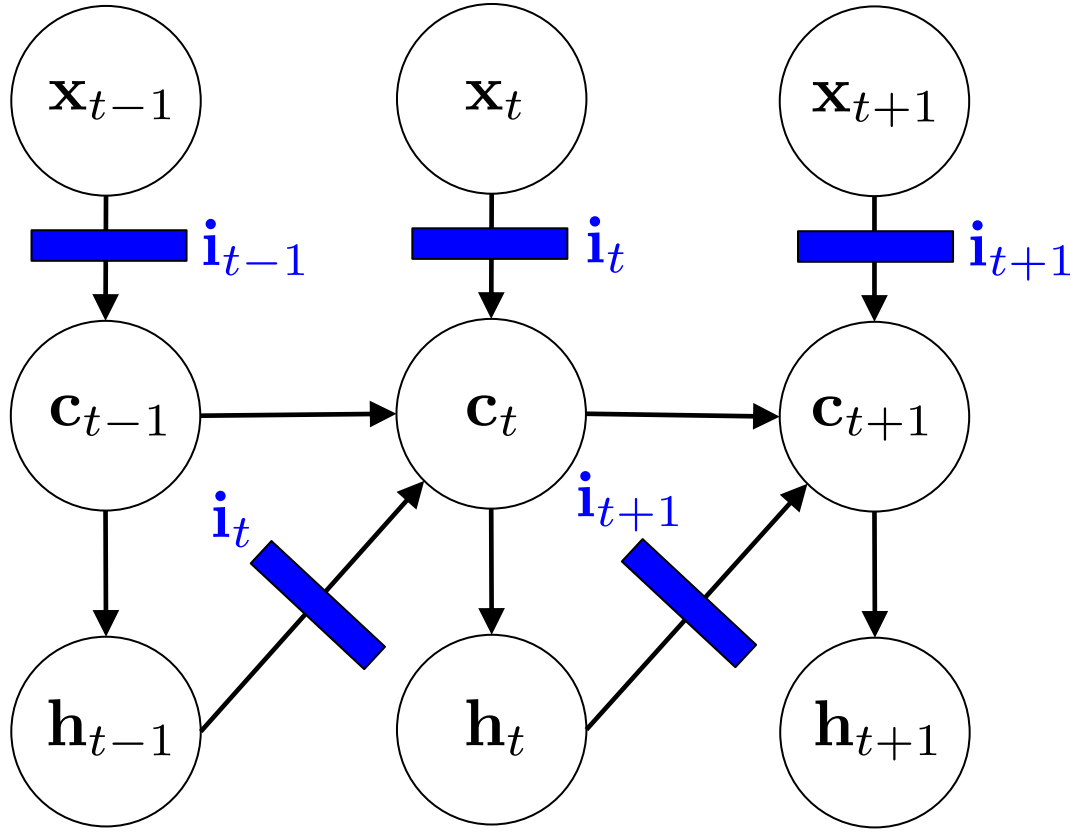


Adding Input Gates

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

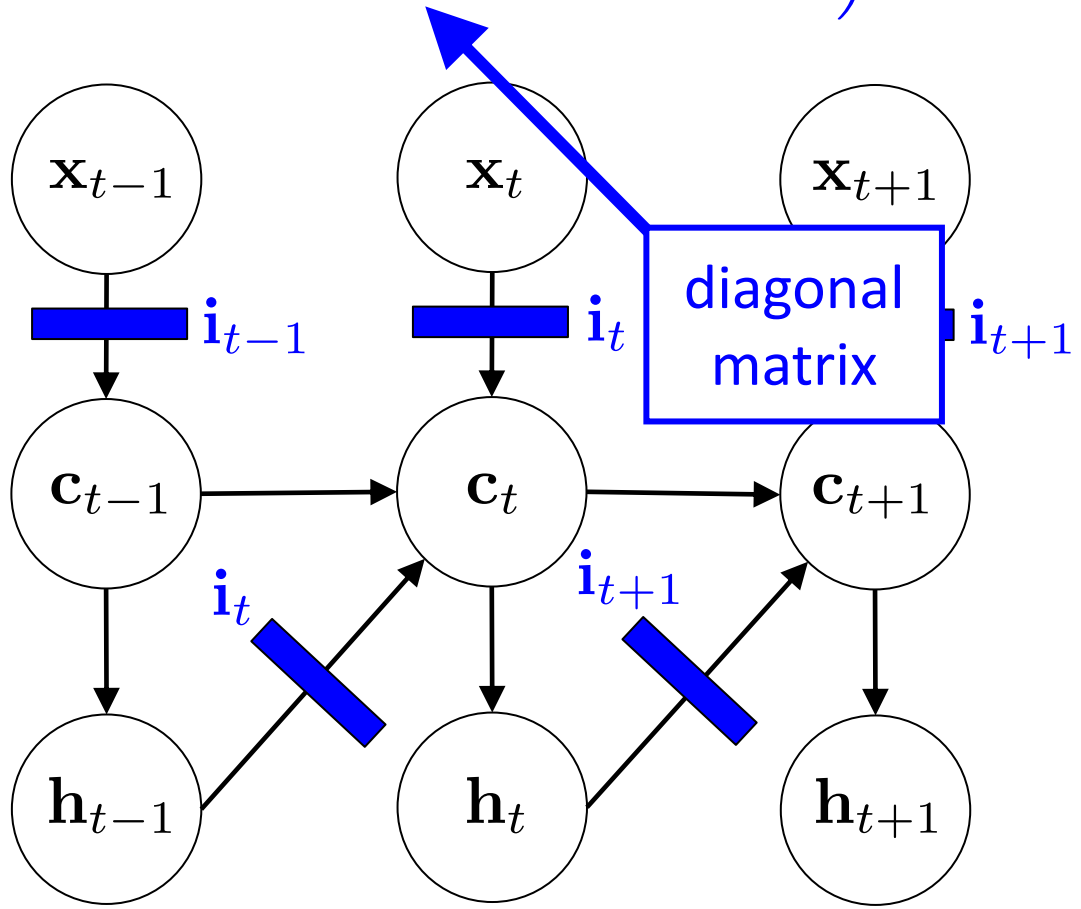
input gate controls how much cell is affected by current observation and previous hidden vector



Adding Input Gates

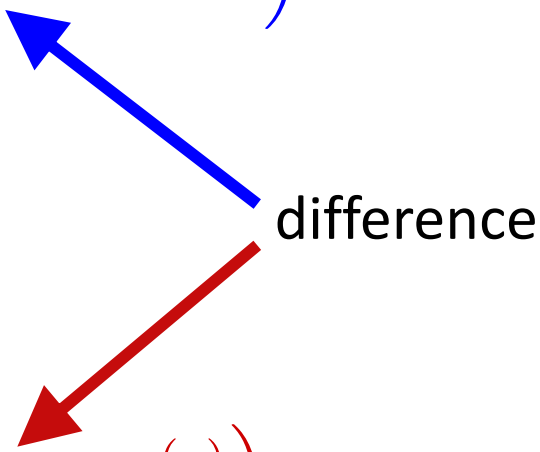
$$\mathbf{i}_t = \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right)$$

input gate is a function of current observation, previous hidden vector, and previous cell vector



Input Gates

$$\mathbf{i}_t = \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right)$$

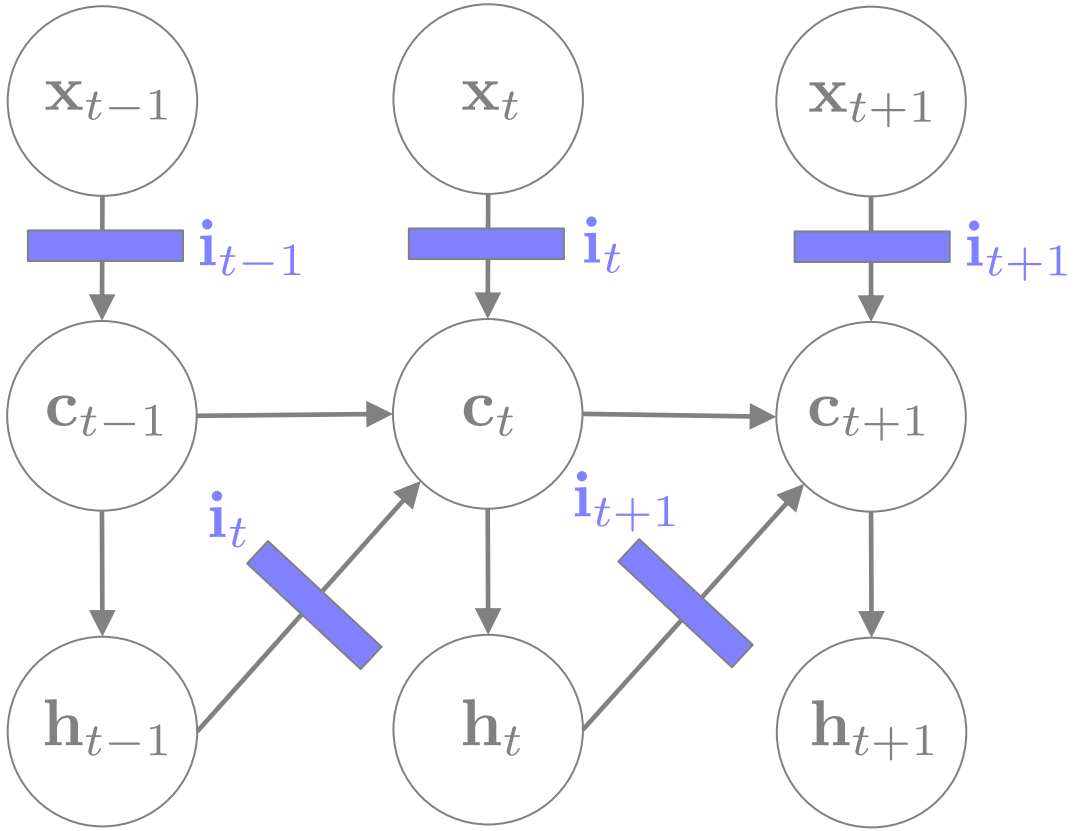


Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

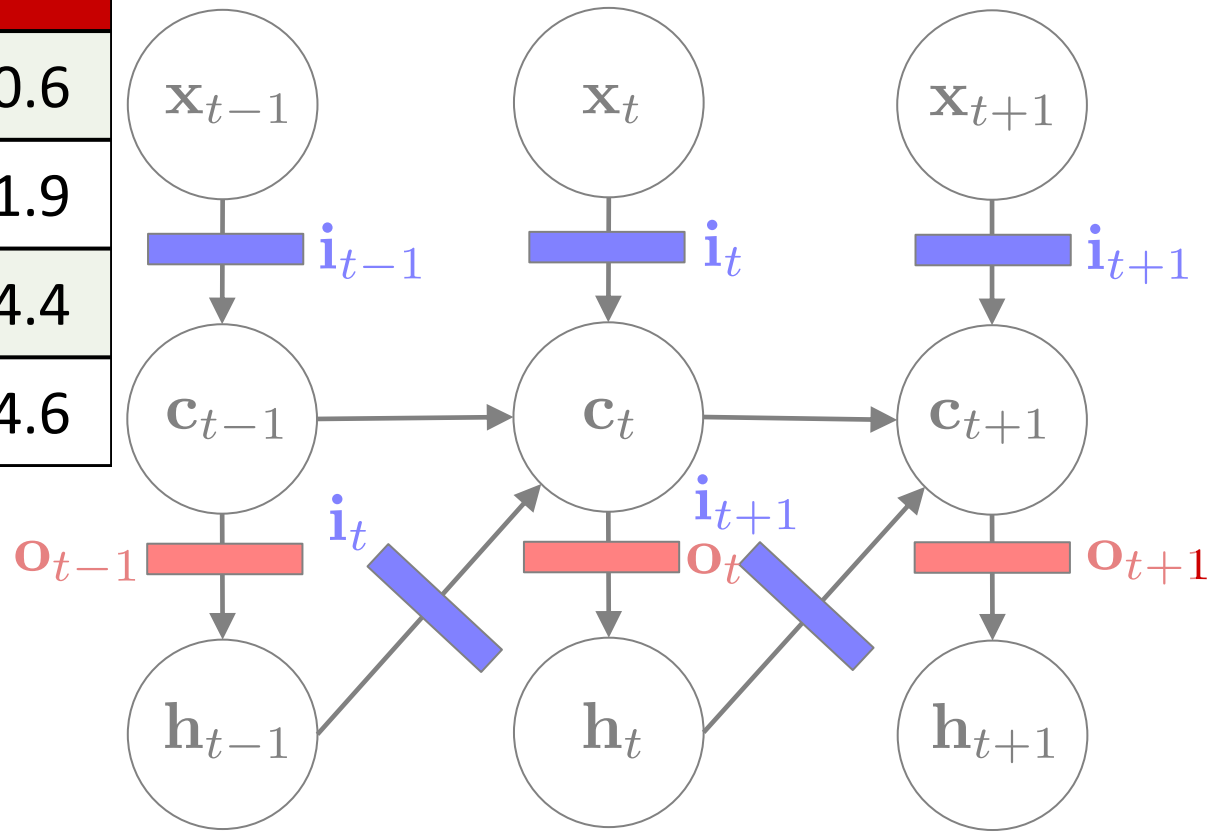
Input Gates

	acc.
gateless	80.6
output gates	81.9
input gates	84.4

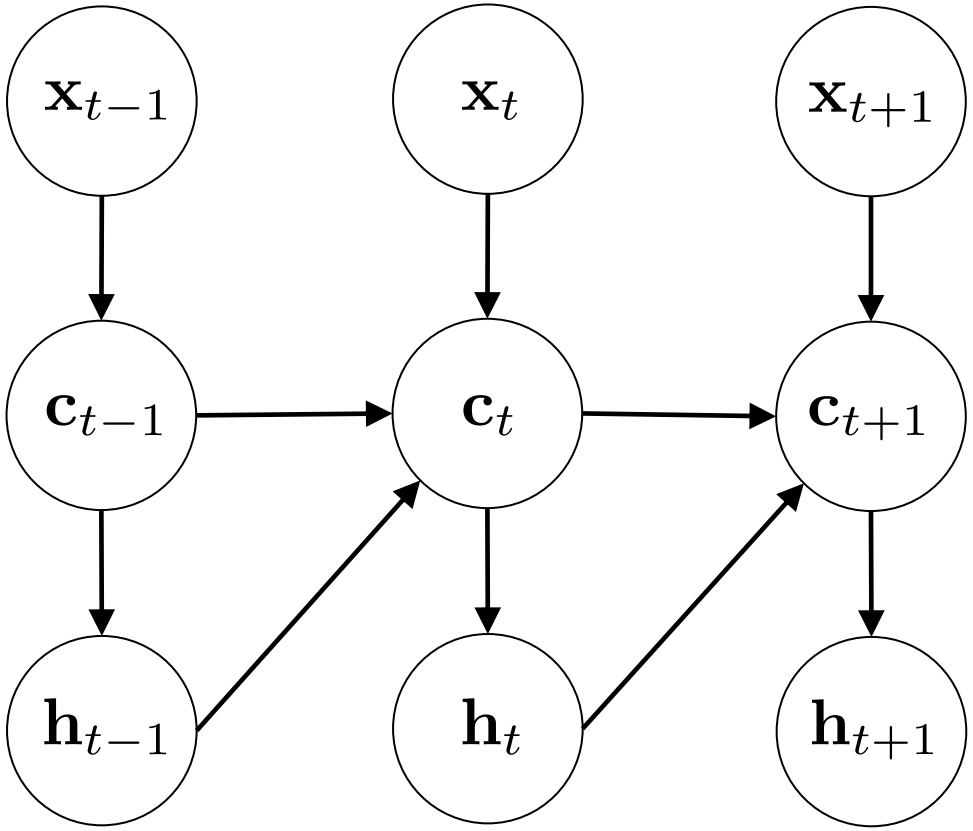


Input Gates

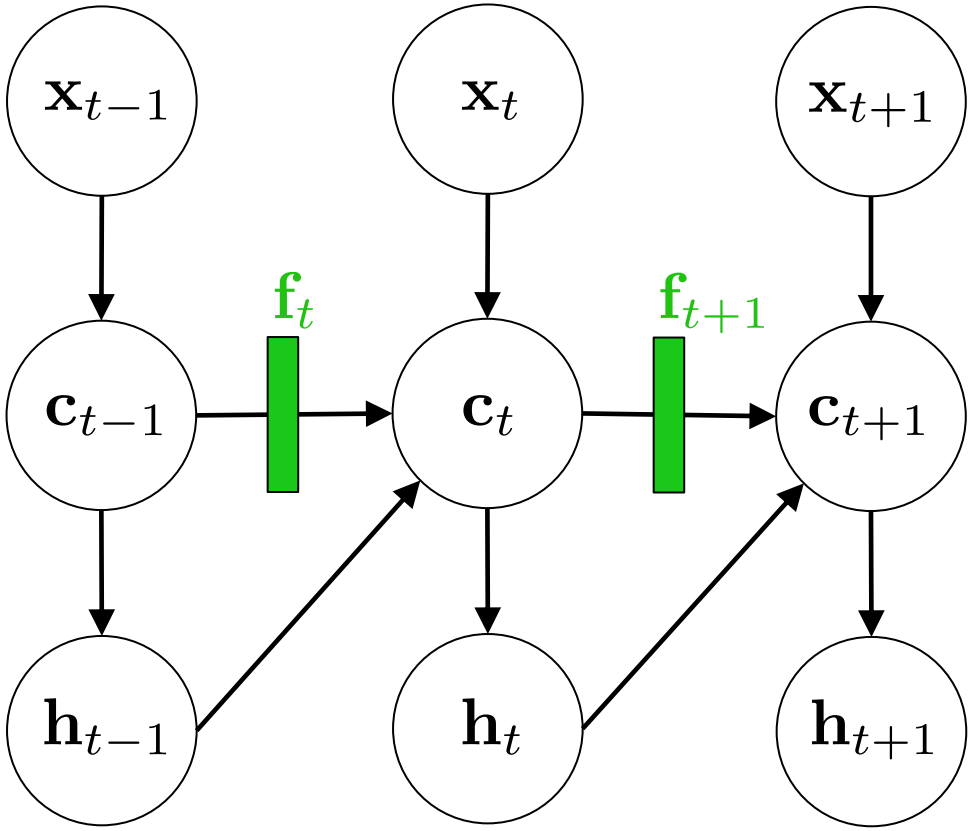
	acc.
gateless	80.6
output gates	81.9
input gates	84.4
input & output gates	84.6



Adding Forget Gates



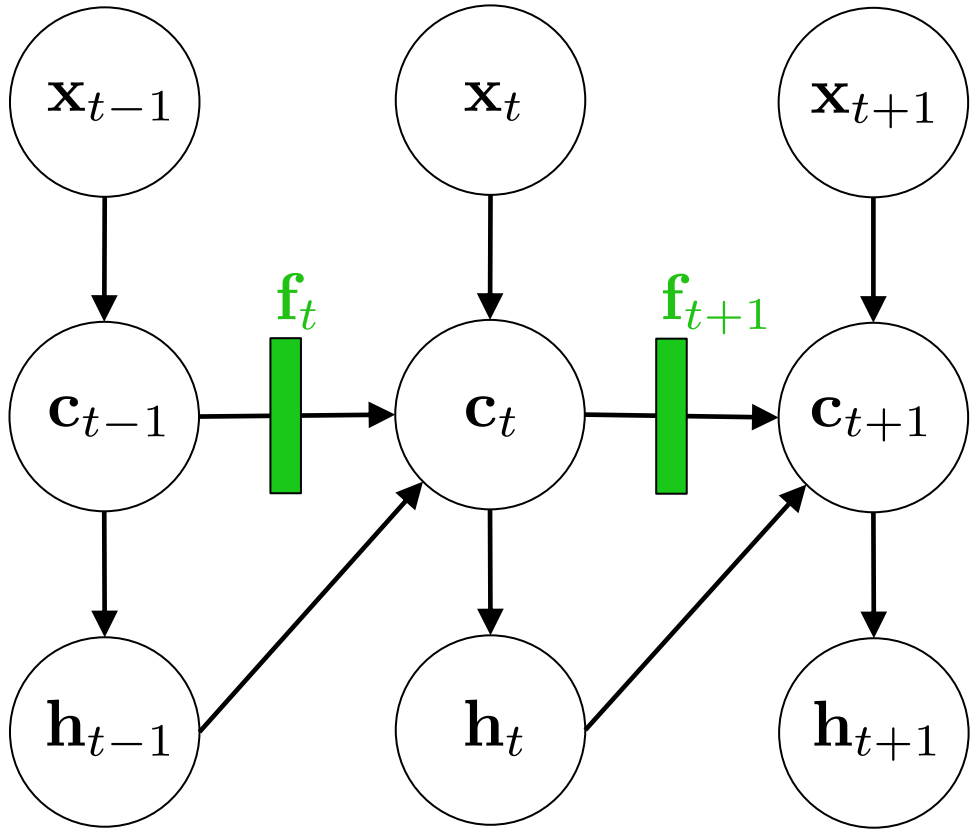
Adding Forget Gates



Adding Forget Gates

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

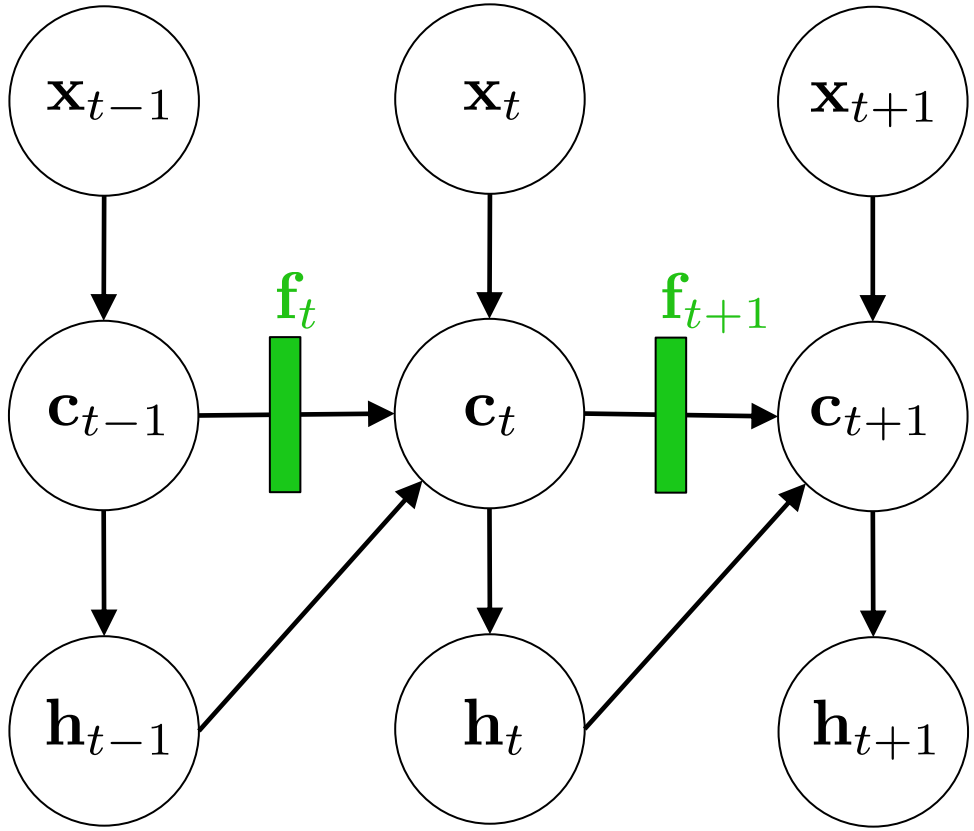
forget gate controls how much "information" is kept from the previous cell vector



Adding Forget Gates

$$\mathbf{f}_t = \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right)$$

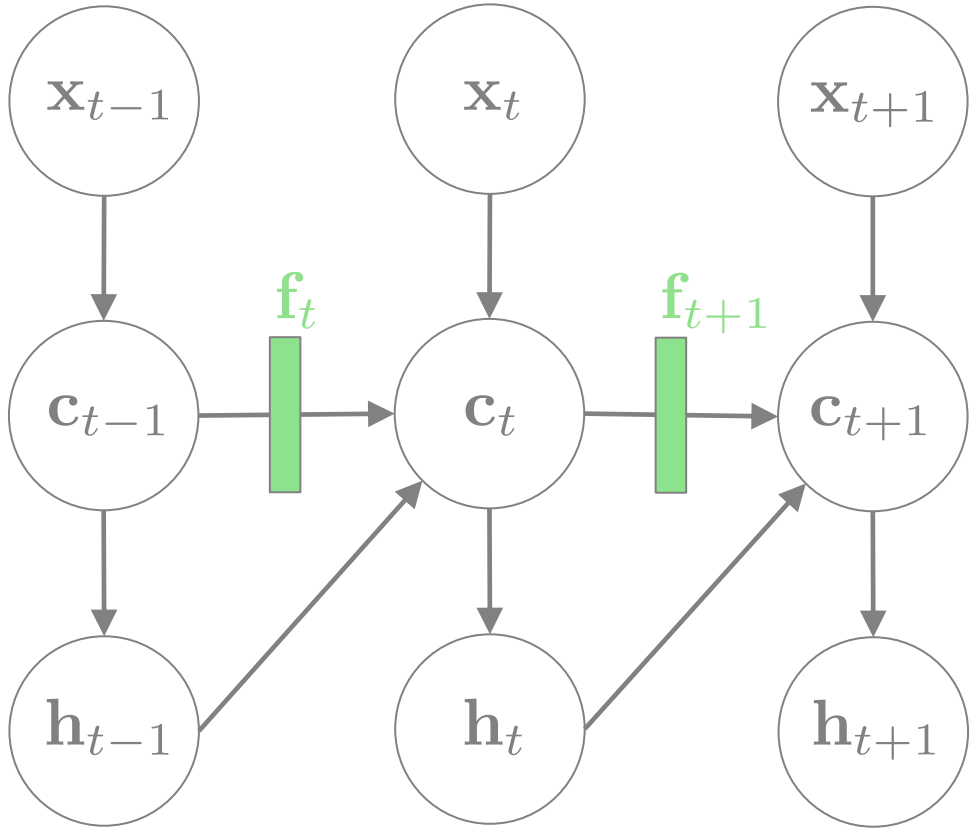
forget gate depends on current observation, previous hidden vector, and previous cell vector



Adding Forget Gates

$$\mathbf{f}_t = \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right)$$

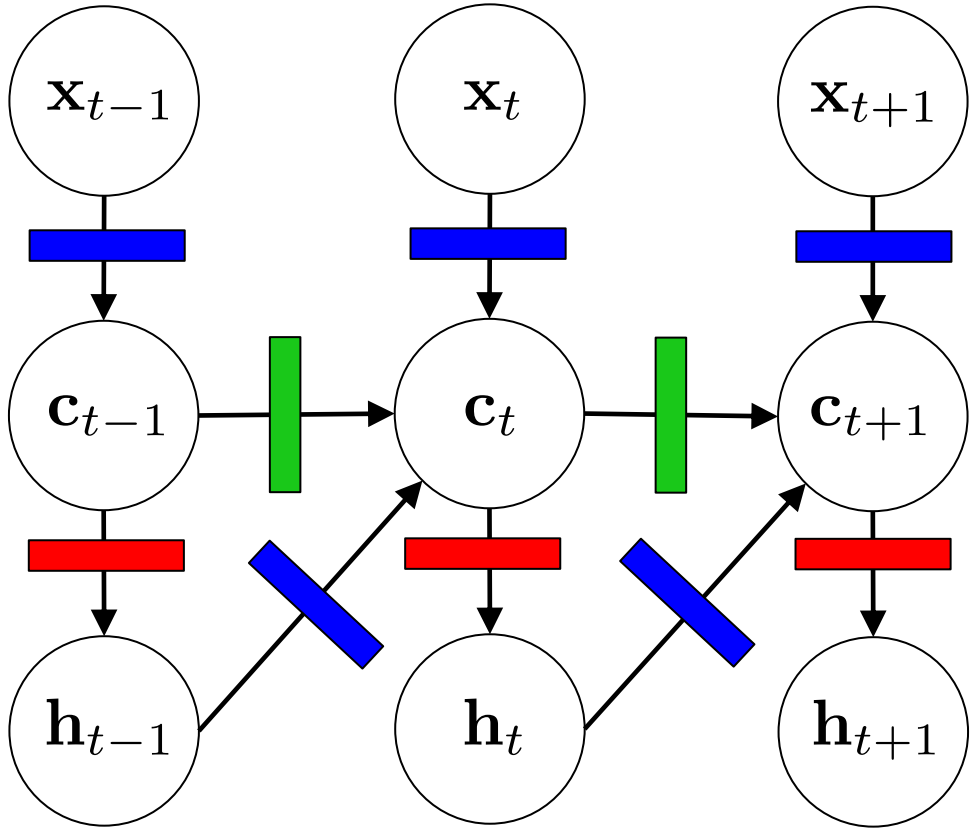
	acc.
gateless	80.6
output gates	81.9
input gates	84.4
forget gates	82.1



All Gates

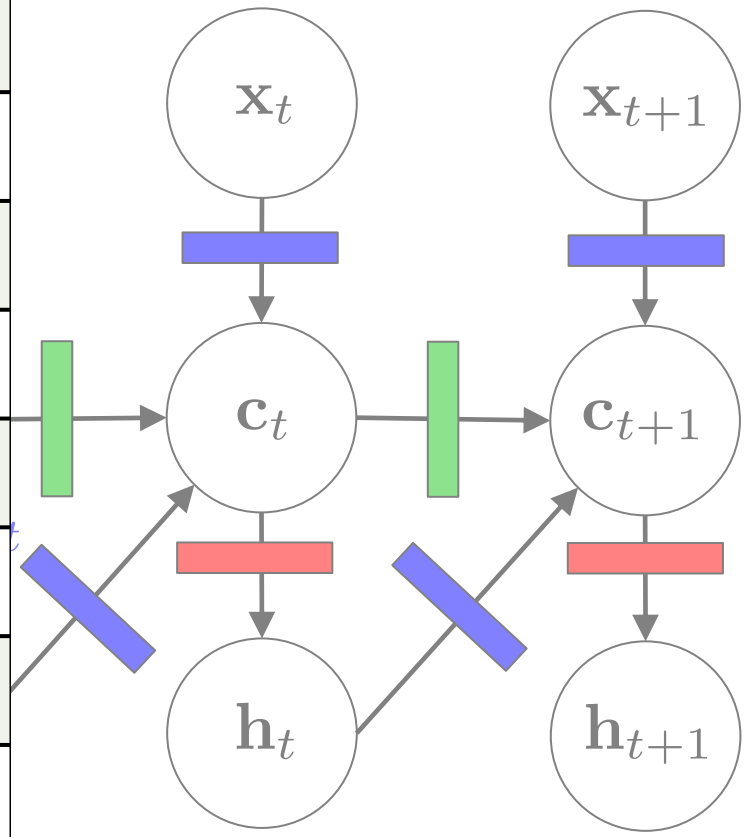
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$



All Gates

	acc.
gateless	80.6
output gates	81.9
input gates	84.4
input & output gates	84.6
forget gates	82.1
input & forget gates	84.1
forget & output gates	82.6
input, forget, output gates	**85.3**



Gated Recurrent Units (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep t we have input $\boldsymbol{x}^{(t)}$ and hidden state $\boldsymbol{h}^{(t)}$ no cell state)

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

Gated Recurrent Units (GRU)

Update gate: controls what parts of hidden state are updated vs preserved

Reset gate: controls what parts of previous hidden state are used to compute new content

New hidden state content: reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

Hidden state: update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{W}_u \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u \boldsymbol{x}^{(t)} + \boldsymbol{b}_u\right)$$

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r\right)$$

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_h(\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h \boldsymbol{x}^{(t)} + \boldsymbol{b}_h\right)$$

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

How does this solve vanishing gradient?
Like LSTM, GRU makes it easier to retain info long-term (e.g., by setting update gate to 0)

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

LSTM vs GRU

- Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used
- **Rule of thumb**: LSTM is a **good default choice** (especially if your data has particularly long dependencies, or you have lots of training data); Switch to **GRUs** for **speed** and **fewer parameters**.

How does LSTM solve vanishing gradients?

- The LSTM architecture makes it **easier** for the RNN to **preserve information over many timesteps**
  - e.g., if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely.
  - In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix W_h that preserves info in the hidden state

- LSTM doesn't *guarantee* that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies

LSTMs: real-world success

- In 2013–2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include handwriting recognition, speech recognition, machine translation
  - **LSTMs** became the **dominant approach** for most NLP tasks
- Now (2019–2022), other approaches (e.g., **Transformers**) have become dominant for many tasks
  - For example, in **WMT** (a Machine Translation conference + competition):
    - In WMT 2014, there were 0 neural machine translation systems (!)
    - In WMT 2016, the summary report contains "**RNN**" 44 times (and these systems won)
    - In WMT 2019: "**RNN**" 7 times, "**Transformer**" 105 times

Is vanishing/exploding gradient just a RNN problem?

- No! It can be a problem for all neural architectures (including **feed-forward** and **convolutional**), especially **very deep** ones.
  - Due to chain rule/choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
  - Thus, lower layers are learned very slowly (hard to train)
- Solution: lots of new deep feedforward/convolutional architectures **add more direct connections** (thus allowing the gradient to flow)
  - For example:
  - **Residual connections** aka "ResNet"
  - Also known as **skip-connections**
  - The **identity connection preserves information** by default
  - This makes **deep** networks much **easier to train**

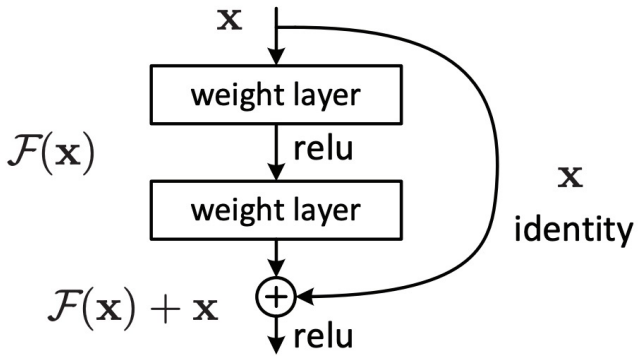


Figure 2. Residual learning: a building block.

"Deep Residual Learning for Image Recognition", He et al, 2015. https://arxiv.org/pdf/1512.03385.pdf

Is vanishing/exploding gradient just a RNN problem?

- Other methods:
  - Dense connections aka “DenseNet”
    - Directly connect each layer to all future layers!
  - Highway connections aka “HighwayNet”
    - Similar to residual connections, but the identity connection vs the transformation layer is controlled by a dynamic gate
    - Inspired by LSTMs, but applied to deep feedforward/convolutional networks

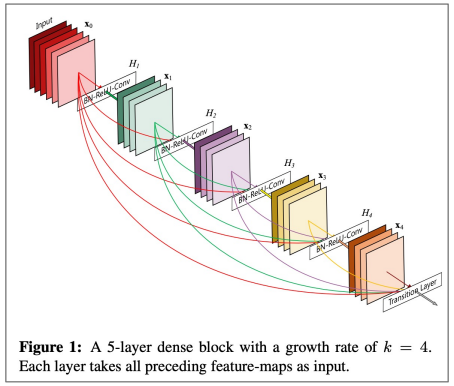
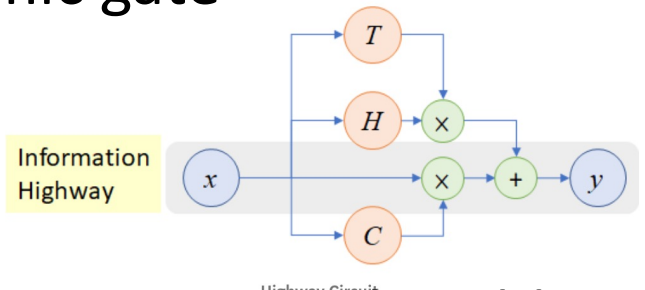


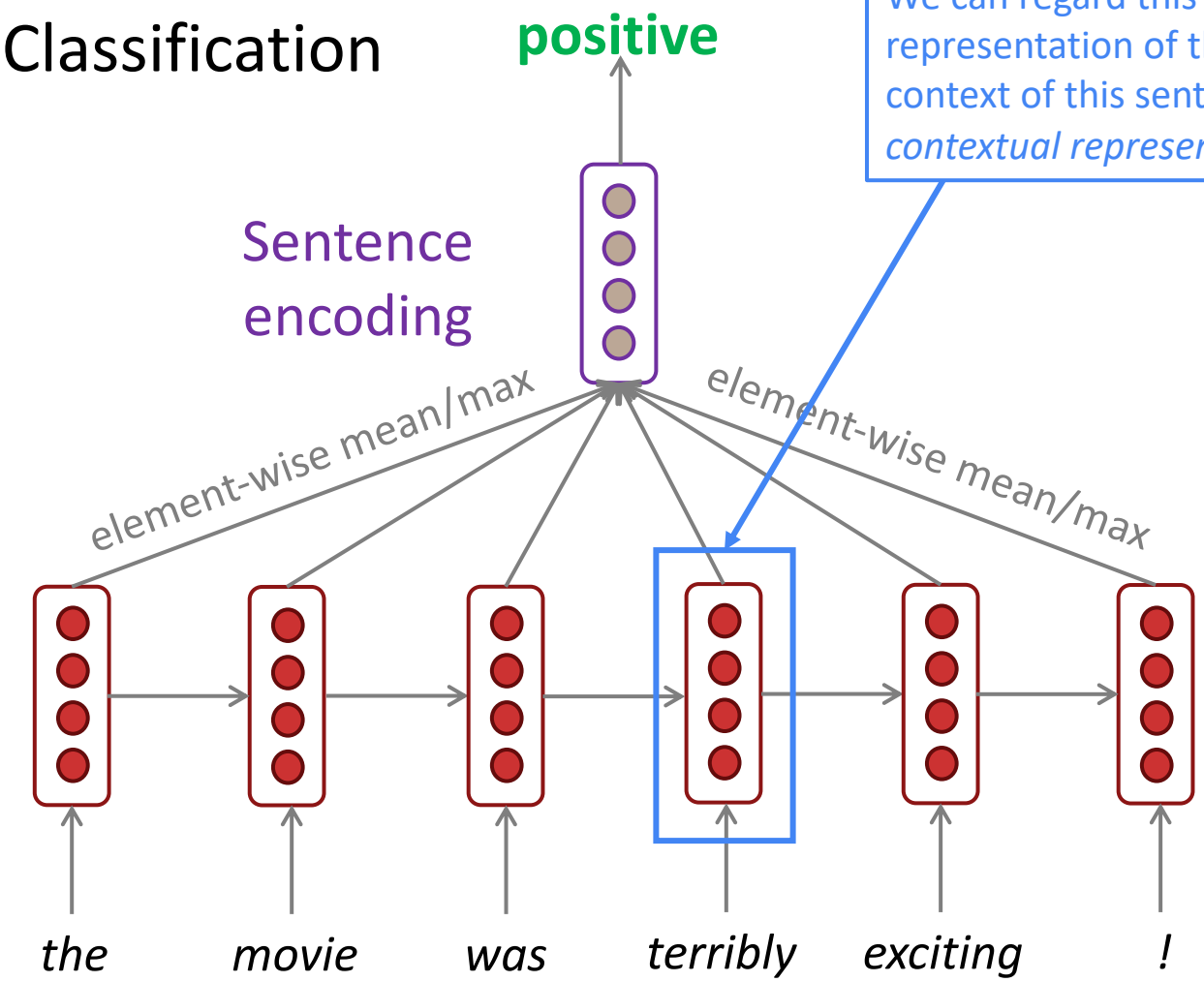
Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.



- **Conclusion**: Though vanishing/exploding gradients are a general problem, RNNs are particularly unstable due to the repeated multiplication by the same weight matrix [Bengio et al, 1994]

Bidirectional and Multi-layer RNNs: motivation

- Task: Sentiment Classification



We can regard this hidden state as a representation of the word *"terribly"* in the context of this sentence. We call this a *contextual representation.*

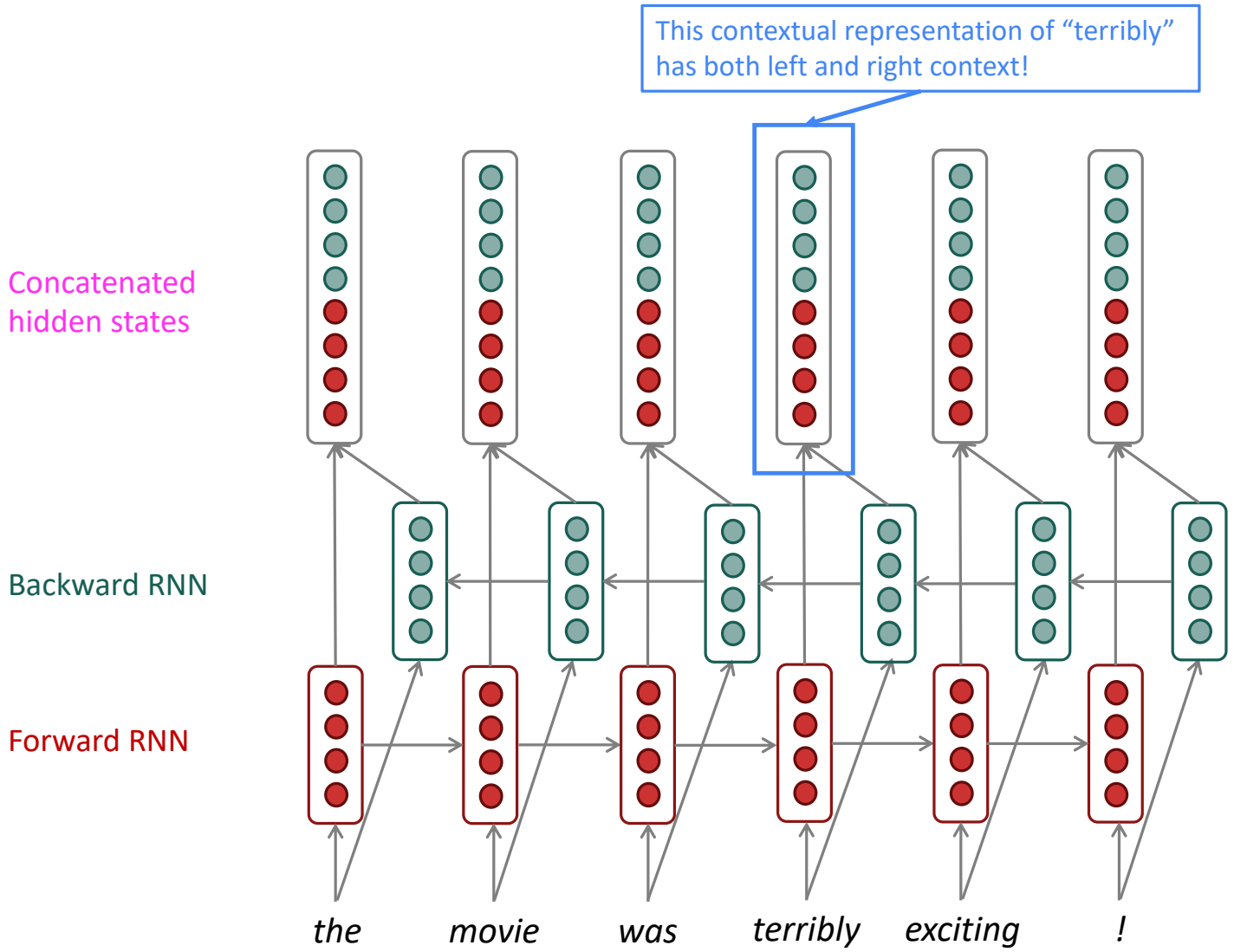
These contextual representations only contain information about the *left* context (e.g. *"the movie was"*).

What about *right* context?

In this example, *"exciting"* is in the right context and this modifies the meaning of *"terribly"* (from negative to positive)

From Stanford CS224n course

Bidirectional RNNs



Bidirectional RNNs

On timestep t:

This is a general notation to mean "compute one forward step of the RNN" – it could be a simple, LSTM, or other (e.g., GRU) RNN computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{FW}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

Generally, these two RNNs have separate weights

Concatenated hidden states $\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

Bidirectional RNNs

- Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**
  - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g., any kind of encoding), **bidirectionality is powerful** (you should use it by default).
- For example, **BERT** (*Bidirectional* Encoder Representations from Transformers) is a powerful pretrained contextual representation system **built on bidirectionality**.
  - You will learn more about **transformers**, including BERT, in a few weeks!

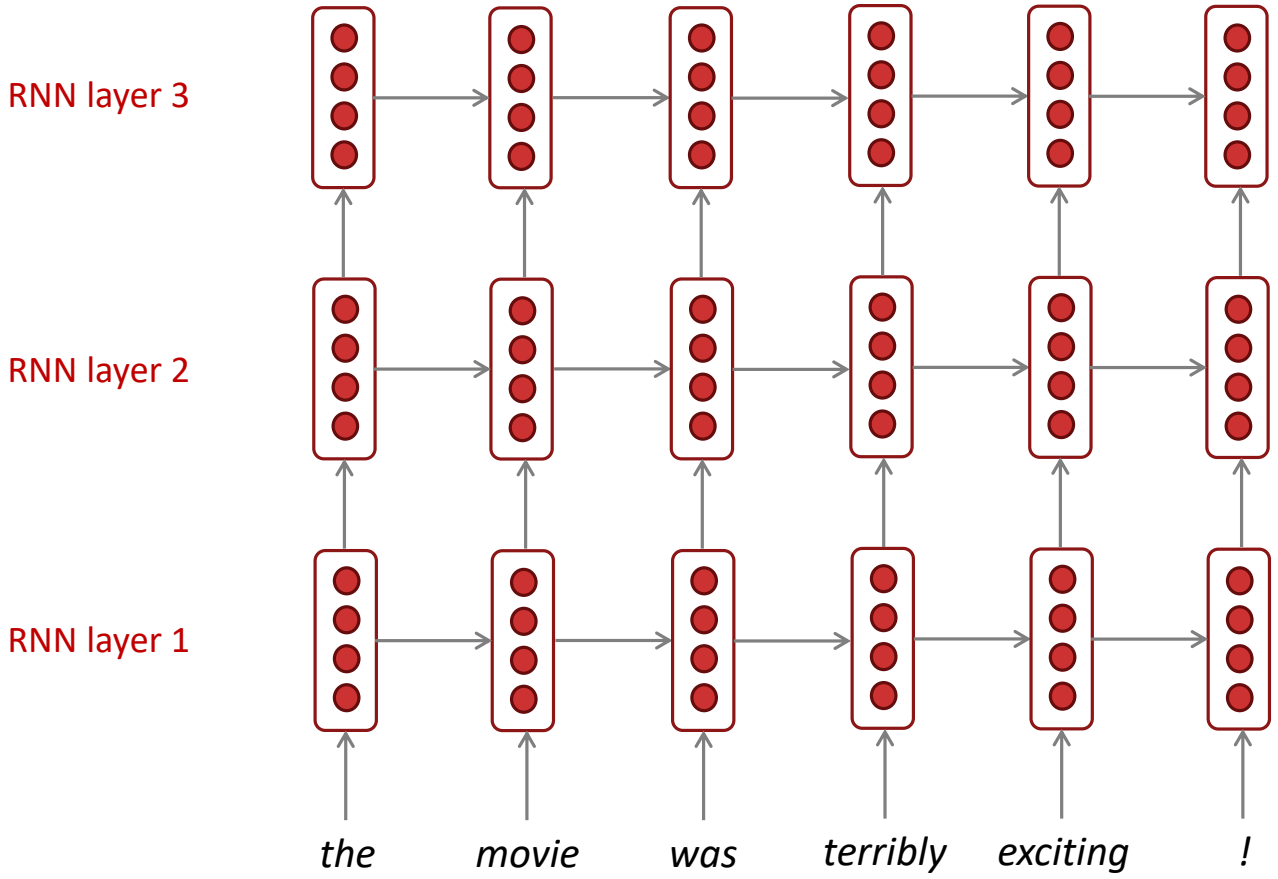
Multi-layer RNNs

- RNNs are already “deep” on one dimension (they unroll over many timesteps)
- We can also make them “deep” in another dimension by **applying multiple RNNs** – this is a multi-layer RNN.
- This allows the network to compute **more complex representations**
  - The **lower RNNs** should **compute lower-level features** and the **higher RNNs** should compute **higher-level features**.

- Multi-layer RNNs are also called *stacked RNNs*.

Multi-layer RNNs

The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1



Multi-layer RNNs in practice

- **High-performing RNNs are usually multi-layer** (but aren't as deep as convolutional or feed-forward networks)
- For example: In a 2017 paper, Britz et al. find that for Neural Machine Translation, **2 to 4 layers** is best for the encoder RNN, and **4 layers** is best for the decoder RNN
  - Often 2 layers is a lot better than 1, and 3 might be a little better than 2
  - Usually, **skip-connections/dense-connections** are needed to train deeper RNNs (e.g., **8 layers**)
- **Transformer**-based networks (e.g., BERT) are usually deeper, like **12 or 24 layers**.
  - You will learn about Transformers later; they have a lot of skipping-like connections

"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017. https://arxiv.org/pdf/1703.03906.pdf

Introduction to PyTorch

Adapted from Sung Kim’s lecture slides at KHUST

What is Python?

- Python is an **easy to learn**, **powerful** programming language.
- It has **efficient high-level** data structures and a **simple but effective** approach to **object-oriented programming**.
- Simple syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and **rapid application development** in many areas on most platforms.

Why Python for AI?

- Simple syntax & less coding
  - Very less coding and simple syntax among other programming languages such as C++.
- Inbuilt libraries for AI projects
  - Great inbuilt libraries for AI.
    – Mathematical library: Numpy, Pandas
    – Machine learning library: Scikit-learn
    – Computer vision library: OpenCV
    – NLP library: Natural Language Toolkit (NLTK)
    – Deep learning library: PyTorch, Tensorflow, Keras
  - Open source
  - Can be used for broad range of programming, from small shell script to enterprise web applications.

New to Python?

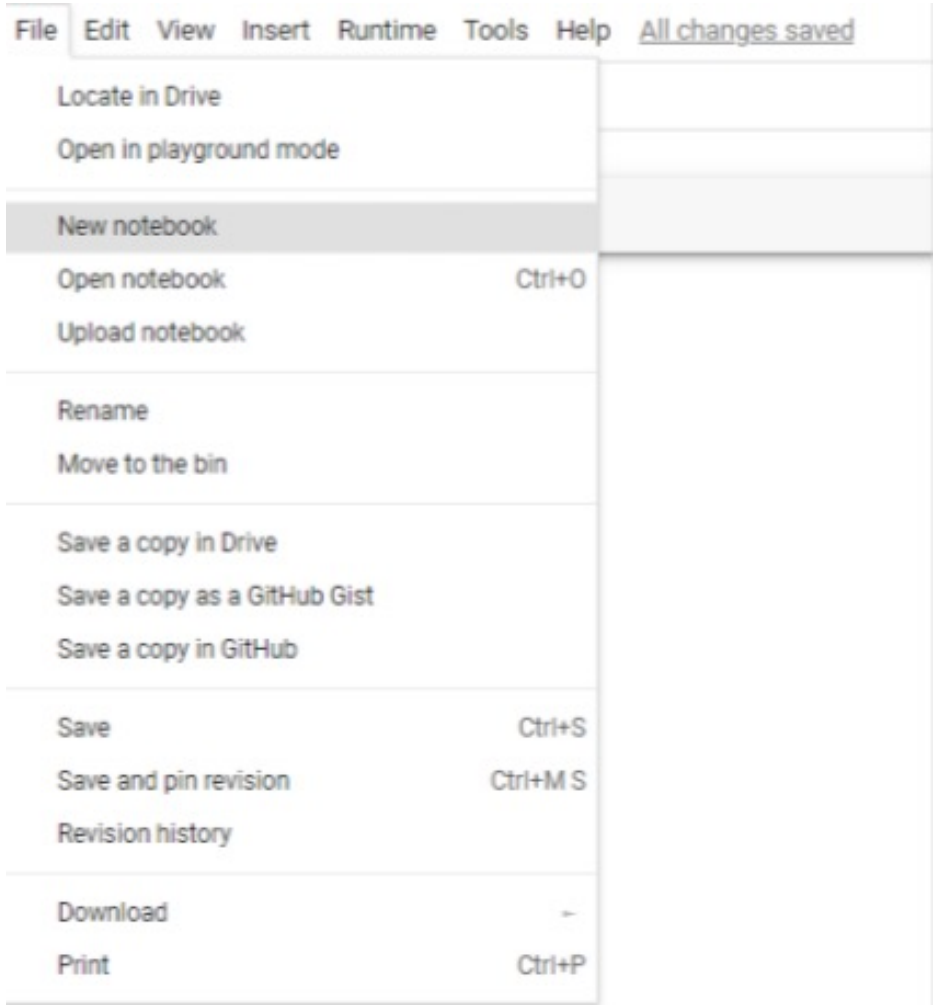
- If you are new to Python, please visit
  https://docs.python.org/3/tutorial/
  or use many other resources such as online lectures, etc.

Google Colab

- Free cloud service with GPU for developers who have gmail address.
- It is provided by Google.
- We can use python and its libraries (Numpy, SciPy, OpenCV, PyTorch, Tensorflow, ...)
- http://colab.research.google.com

Google Colab

- 1) Make a new notebook.

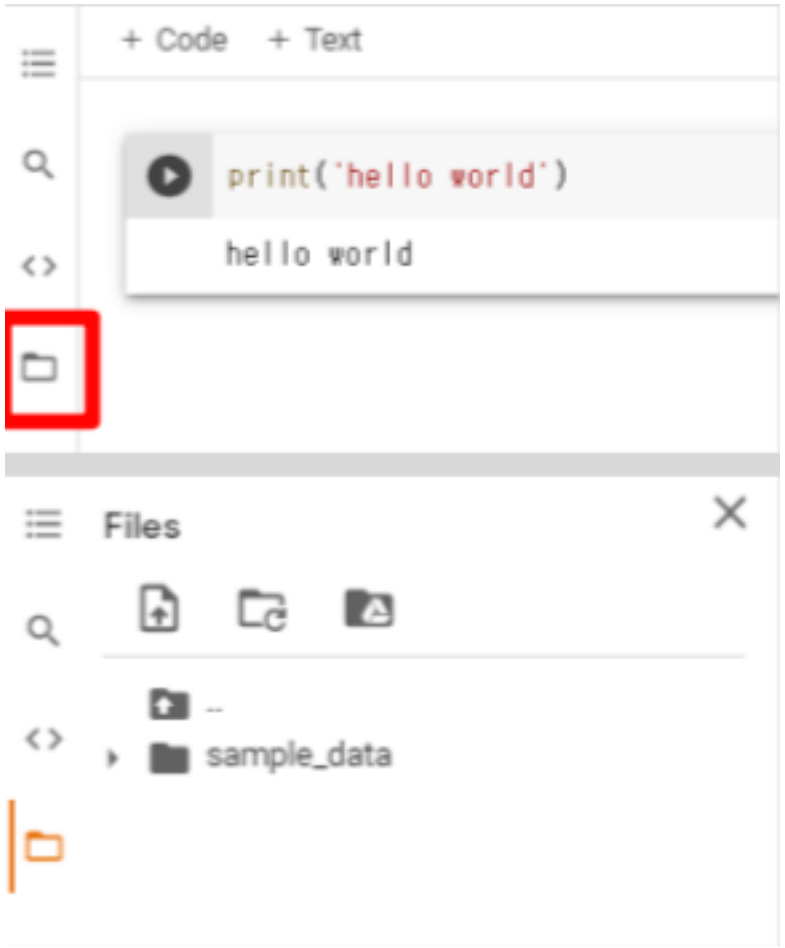


Google Colab

- 2) Type python code and it simply works.

Google Colab

- File upload
  - Click button in the left side.
  - The path is: `/content/sample_data’
  - Upload/rename/delete files.



Google Colab

- Use google drive for large data:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=ur

Enter your authorization code:
```

```
[ ] with open('/content/drive/My Drive/foo.txt', 'w') as f:
      f.write('Hello Google Drive!')
    !cat /content/drive/My\ Drive/foo.txt
```

```
Hello Google Drive!
```

Notebook

- Notebook documents (or “notebooks”, all lower case) are documents produced by the Jupyter Notebook App.
- Contain both computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc…).
- Both **human-readable** documents containing the analysis description and the **results** (figures, tables, etc..) as well as **executable** documents
- Can be run to perform data analysis.

Notebook

- Notebook example

Run some Python code!

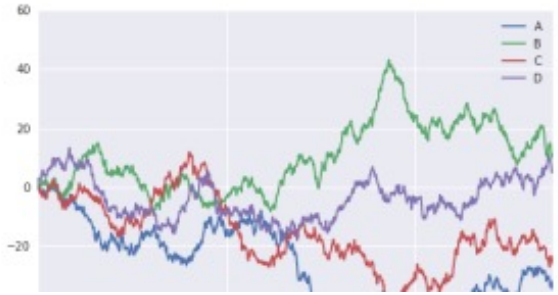
To run the code below:

1. Click on the cell to select it.
2. Press SHIFT+ENTER on your keyboard or press the play button (▶) in the toolbar above.

A full tutorial for using the notebook interface is available here.

In [1]:
```
%matplotlib inline

import pandas as pd
import numpy as np
import matplotlib

from matplotlib import pyplot as plt
import seaborn as sns

ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
ts = ts.cumsum()

df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                  columns=['A', 'B', 'C', 'D'])
df = df.cumsum()
plt.figure(); df.plot(); plt.legend(loc='best')
```

Out[1]:
```
<matplotlib.legend.Legend at 0x7fb27b72fcc0>

<matplotlib.figure.Figure at 0x7fb283672b70>
```



What is NumPy?

- NumPy (**Numerical Python**) is the fundamental package for scientific computing in Python. It provides:
  - A multidimensional array object with methods to efficiently operate on it
  - Various derived objects (such as masked arrays and matrices)
  - Routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
  - The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

NumPy Basics

- **ndarray** (also known by the alias **array**, *numpy.array*)
  - NumPy's main object, the homogeneous multidimensional array.
  - It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers.
  - In NumPy, dimensions are called *axes*.
  - For example, the coordinates of a point in 3D space [1, 2, 1] has one axis.

```
[[1., 0., 0.],
 [0., 1., 2.]]
```

  - The array above has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

New to NumPy?

- Use online tutorials:
  - https://cs231n.github.io/python-numpy-tutorial/
  - https://numpy.org/doc/stable/user/quickstart.html

What is PyTorch?

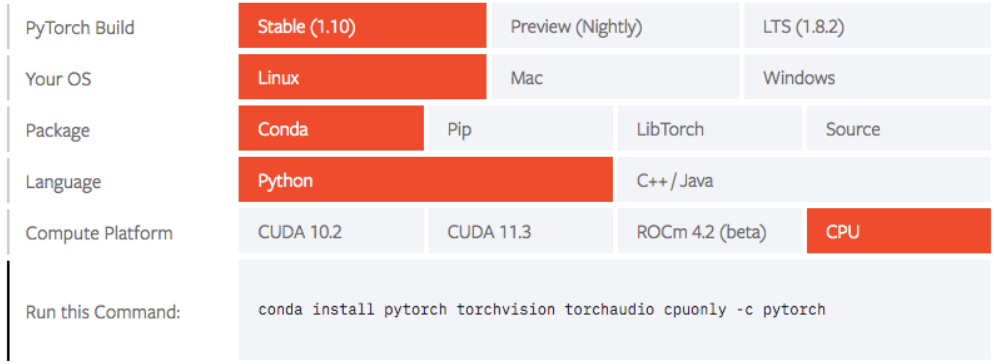
- PyTorch is a python package that provides two high-level features:
  - **Tensor computation** (like NumPy) with strong GPU acceleration
  - Deep Neural Networks built on a tape-based **autograd** system

Why PyTorch?

- More Pythonic (imperative)
  - Flexible
  - Intuitive and cleaner code
  - Easy to debug
- More Neural Networkic
  - Write code as the network works
  - forward/backward

Installing PyTorch

- https://pytorch.org/get-started/locally/
  - To check: use this command: *python -c "import torch; print(torch.__version__)"*



To install PyTorch via Anaconda, use the following conda command:

```
conda install pytorch torchvision -c pytorch
```

To install PyTorch via pip, use one of the following two commands, depending on your Python version:

```
# Python 3.x
pip3 install torch torchvision
```

Tensors

- Tensors
  - **Tensors** are a specialized data structure that are very similar to arrays and matrices.
  - In PyTorch, we use tensors to encode the *inputs* and *outputs* of a model, as well as the *model's parameters*.
  - Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.
  - Tensors are also optimized for automatic differentiation.

Initializing a Tensor

- Directly from data

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

- From a NumPy array

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

Initializing a Tensor

- From another tensor:

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")
x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
tensor([[1, 1],
        [1, 1]])

Random Tensor:
tensor([[0.3277, 0.7579],
        [0.1860, 0.8509]])
```

Initializing a Tensor

- With random or constant values:

```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
tensor([[0.2882, 0.0322, 0.4411],
        [0.5961, 0.6428, 0.1681]])

Ones Tensor:
tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros Tensor:
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Attributes of a Tensor

- Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Operations on Tensors

- Operations on Tensors
  - Over 100 tensor operations, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling and more are supported in **torch** packakge(e.g., torch.transpose()).
  - Each of these operations can be run on the **GPU** (at typically higher speeds than on a CPU). If you're using Colab, allocate a GPU by going to "Runtime > Change runtime type > GPU".
  - By default, tensors are created on the CPU. We need to explicitly move tensors to the GPU using **.to** method (after checking for GPU availability). Copying large tensors across devices can be expensive in terms of time and memory.

```
if torch.cuda.is_available():
    tensor = tensor.to('cuda')
```

Machine Learning: Our Problem

What would be the grade if I study 4 hours?

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

Training dataset

Test dataset

Model & Loss

- Linear model

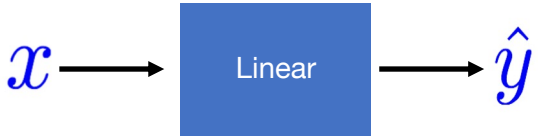
$$\hat{y} = x * w$$

```
w = 1.0  # a random guess: random value

# our model for the forward pass
def forward(x):
    return x * w
```

$$loss = (\hat{y} - y)^2$$

```
# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)
```

Computing gradient in simple network



Gradient of **loss** with respect to ***w***

$$\frac{\partial loss}{\partial w} = ?$$

```
# compute gradient
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)
```

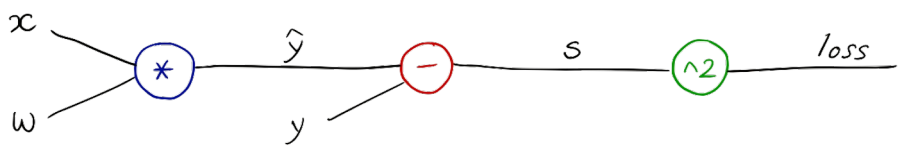
Training: updating weight

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0  # a random guess: random value

# our model forward pass
def forward(x):
    return x * w


# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)


# compute gradient
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)
```

```
# Before training
print("predict (before training)",  4, forward(4))

# Training loop
for epoch in range(100):
    for x_val, y_val in zip(x_data, y_data):
        grad = gradient(x_val, y_val)
        w = w - 0.01 * grad
        print("\tgrad: ", x_val, y_val, grad)
        l = loss(x_val, y_val)

    print("progress:", epoch, "w=", w, "loss=", l)

# After training
print("predict (after training)",  "4 hours", forward(4))
```

```
predict (before training) 4 4.0
        grad:  1.0 2.0 -2.0
        grad:  2.0 4.0 -7.84
        grad:  3.0 6.0 -16.23
progress: 0 w= 1.26 loss= 4.92
        grad:  1.0 2.0 -1.48
        grad:  2.0 4.0 -5.8
        grad:  3.0 6.0 -12.0
progress: 1 w= 1.45 loss= 2.69
        grad:  1.0 2.0 -1.09
        grad:  2.0 4.0 -4.29
        grad:  3.0 6.0 -8.87
progress: 2 w= 1.6 loss= 1.47
        grad:  1.0 2.0 -0.81
        grad:  2.0 4.0 -3.17
        grad:  3.0 6.0 -6.56
..
progress: 7 w= 1.91 loss= 0.07
        grad:  1.0 2.0 -0.18
        grad:  2.0 4.0 -0.7
        grad:  3.0 6.0 -1.45
progress: 8 w= 1.93 loss= 0.04
        grad:  1.0 2.0 -0.13
        grad:  2.0 4.0 -0.52
        grad:  3.0 6.0 -1.07
progress: 9 w= 1.95 loss= 0.02
predict (after training) 4 hours 7.80
```

Training: forward, backward, and update weight




```
# Training loop
for epoch in range(10):
    for x_val, y_val in zip(x_data, y_data):
        l = loss(x_val, y_val)
        l.backward()
        print("\tgrad: ", x_val, y_val, w.grad.data[0])
        w.data = w.data - 0.01 * w.grad.data

        # Manually zero the gradients after updating weights
        w.grad.data.zero_()

    print("progress:", epoch, l.data[0])

# After training
print("predict (after training)",  4, forward(4).data[0])
```

```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.840000152587891
    grad:  3.0 6.0 -16.228801727294922
progress: 0 7.315943717956543
    grad:  1.0 2.0 -1.478623867034912
    grad:  2.0 4.0 -5.796205520629883
    grad:  3.0 6.0 -11.998146057128906
progress: 1 3.9987640380859375
    grad:  1.0 2.0 -1.0931644439697266
    grad:  2.0 4.0 -4.285204887390137
    grad:  3.0 6.0 -8.870372772216797
progress: 2 2.1856532096862793
    grad:  1.0 2.0 -0.8081896305084229
    grad:  2.0 4.0 -3.1681032180786133
    grad:  3.0 6.0 -6.557973861694336
progress: 3 1.1946394443511963
    grad:  1.0 2.0 -0.5975041389465332
    grad:  2.0 4.0 -2.3422164916992188
    grad:  3.0 6.0 -4.848389625549316
progress: 4 0.6529689431190491
    grad:  1.0 2.0 -0.4417421817779541
    grad:  2.0 4.0 -1.7316293716430664
    grad:  3.0 6.0 -3.58447265625
progress: 5 0.35690122842788696
    grad:  1.0 2.0 -0.3265852928161621
    grad:  2.0 4.0 -1.2802143096923828
    grad:  3.0 6.0 -2.650045394897461
progress: 6 0.195076122879982
    grad:  1.0 2.0 -0.24144840240478516
    grad:  2.0 4.0 -0.9464778900146484
    grad:  3.0 6.0 -1.9592113494873047
progress: 7 0.10662525147199631
    grad:  1.0 2.0 -0.17850565910339355
    grad:  2.0 4.0 -0.699742317199707
    grad:  3.0 6.0 -1.4484672546386719
```

The Steps in PyTorch

- Your steps in PyTorch

1. **Design your model using class**
2. **Construct loss and optimizer (select from PyTorch API)**
3. **Training cycle (forward, backward, update)**

Model class in PyTorch way

```
import torch




class Model(torch.nn.Module):
  def __init__(self):
    """
    In the constructor we instantiate two nn.Linear module
    """
    super(Model, self).__init__()
    self.linear = torch.nn.Linear(1, 1) # One in and one out

  def forward(self, x):
    """
    In the forward function we accept a Variable of input data and we must return
    a Variable of output data. We can use Modules defined in the constructor as
    well as arbitrary operators on Variables.
    """
    y_pred = self.linear(x)
    return y_pred

# our model
model = Model()
```

Construct loss and optimizer

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Training: forward, loss, backward, step

```python
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```python
for x_val, y_val in zip(x_data, y_data):
        ...
        w.data = w.data - 0.01 * w.grad.data
```

Testing Model

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

hour_var = torch.Tensor([[4.0]])
print("predict (after training)", 4, model.forward(hour_var).data[0][0])

Building fun models

- Neural Net components
  - CNN
  - RNN
  - Activations
- Losses
- Optimizers

torch.nn

- Containers
- Convolution Layers
- Pooling Layers
- Padding Layers
- Non-linear Activations
- Normalization layers
- Recurrent layers
- Linear layers
- Dropout layers
- Sparse layers
- Distance functions
- Loss functions
- Vision layers

Convolution Layers

- Conv1d
- Conv2d
- Conv3d
- ConvTranspose1d
- ConvTranspose2d
- ConvTranspose3d

Recurrent layers

- RNN
- LSTM
- GRU
- RNNCell
- LSTMCell
- GRUCell

Non-linear Activations

- ReLU
- ReLU6
- ELU
- SELU
- PReLU
- LeakyReLU
- Threshold
- Hardtanh
- Sigmoid
- Tanh
- LogSigmoid
- Softplus
- Softshrink
- Softsign
- Tanhshrink
- Softmin
- Softmax
- Softmax2d
- LogSoftmax

http://pytorch.org/docs/master/nn.html

Loss functions

- **Loss functions**
  - L1Loss
  - MSELoss
  - CrossEntropyLoss
  - NLLLoss
  - PoissonNLLLoss
  - NLLLoss2d
  - KLDivLoss
  - BCELoss
  - BCEWithLogitsLoss
  - MarginRankingLoss
  - HingeEmbeddingLoss
  - MultiLabelMarginLoss
  - SmoothL1Loss
  - SoftMarginLoss
  - MultiLabelSoftMarginLoss
  - CosineEmbeddingLoss
  - MultiMarginLoss
  - TripletMarginLoss

http://pytorch.org/docs/master/nn.html

Other optimizers

- torch.optim.Adagrad
- torch.optim.Adam
- torch.optim.Adamax
- torch.optim.ASGD
- torch.optim.LBFGS
- torch.optim.RMSprop
- torch.optim.Rprop
- torch.optim.SGD

Classifying Diabetes

-0.411765	0.165829	0.213115	0	0	-0.23696	-0.894962	-0.7	1
-0.647059	-0.21608	-0.180328	-0.353535	-0.791962	-0.0760059	-0.854825	-0.833333	0
0.176471	0.155779	0	0	0	0.052161	-0.952178	-0.733333	1
-0.764706	0.979899	0.147541	-0.0909091	0.283688	-0.0909091	-0.931682	0.0666667	0
-0.0588235	0.256281	0.57377	0	0	0	-0.868488	0.1	0
-0.529412	0.105528	0.508197	0	0	0.120715	-0.903501	-0.7	1
0.176471	0.688442	0.213115	0	0	0.132638	-0.608027	-0.566667	0
0.176471	0.396985	0.311475	0	0	-0.19225	0.163962	0.2	1

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)

x_data = torch.from_numpy(xy[:, 0:-1])
y_data = torch.from_numpy(xy[:, [-1]])

print(x_data.data.shape) # torch.Size([759, 8])
print(y_data.data.shape) # torch.Size([759, 1])
```

Classifying Diabetes

```
class Model(torch.nn.Module):

    def __init__(self):
        """
        In the constructor we instantiate three nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = torch.nn.Linear(8, 6)
        self.l2 = torch.nn.Linear(6, 4)
        self.l3 = torch.nn.Linear(4, 1)

        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        return y_pred
```

Classifying Diabetes

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
x_data = torch.from_numpy(xy[:, 0:-1])
y_data = torch.from_numpy(xy[:, [-1]])

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = torch.nn.Linear(8, 6)
        self.l2 = torch.nn.Linear(6, 4)
        self.l3 = torch.nn.Linear(4, 1)

        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        return y_pred

# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Training loop
for epoch in range(100):
        # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

1 **Design your model using class**

2 **Construct loss and optimizer (select from PyTorch API)**

3 **Training cycle (forward, backward, update)**

Dataset & DataLoader

Manual data feed

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
x_data = torch.from_numpy(xy[:, 0:-1])
y_data = torch.from_numpy(xy[:, [-1]])

...

# Training loop
for epoch in range(100):
        # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Batch (batch size)

```
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
```

288

In the neural network terminology:

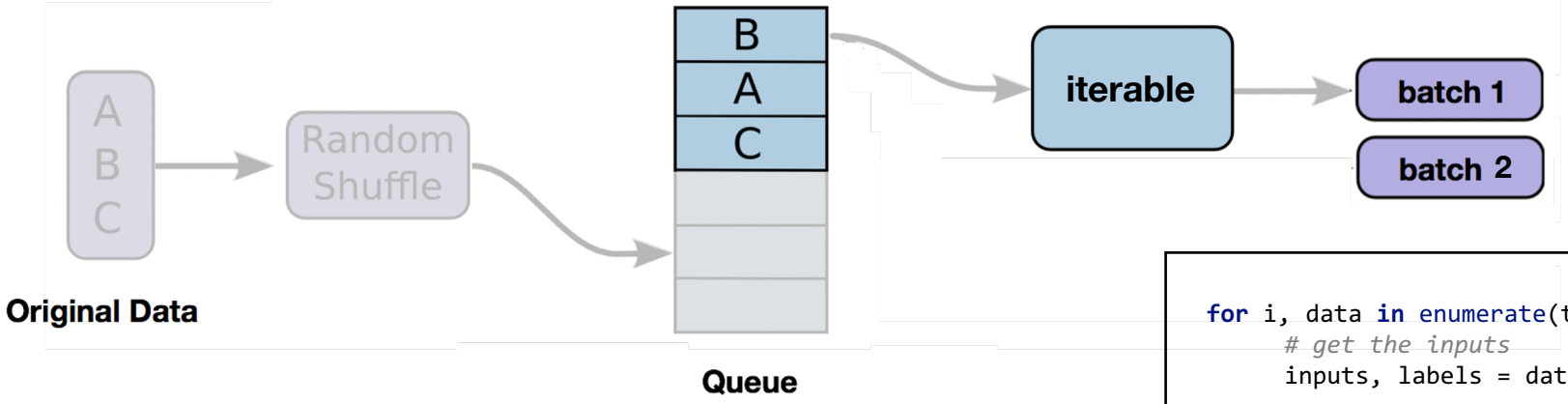
- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

https://stackoverflow.com/questions/4752626

Dataset & DataLoader

- Dataset & DataLoader
  - We want our dataset code (e.g., data processing) to be decoupled from our model training code for better readability and modularity.
  - **torch.utils.data.Dataset** stores the samples and their corresponding labels, and **torch.utils.data.DataLoader** wraps an iterable around the Dataset to enable easy access to the samples.

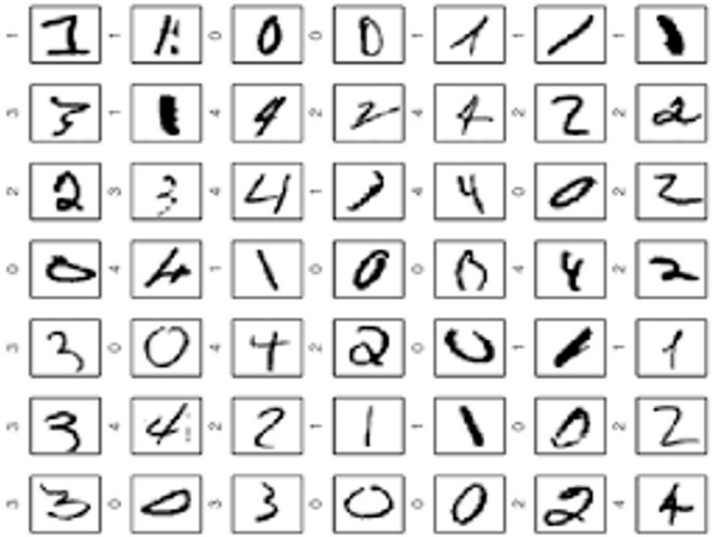



```
for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # Run your training process
        print(epoch, i, "inputs", inputs.data, "labels", labels.data)
```

Dataset & DataLoader

- Existing dataset: MNIST

```
# MNIST Dataset
train_dataset = datasets.MNIST(root='./data/',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

test_dataset = datasets.MNIST(root='./data/',
                              train=False,
                              transform=transforms.ToTensor())

# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)


for batch_idx, (data, target) in enumerate(train_loader):
    ...
```



https://github.com/yunjey/pytorch-tutorial

Dataset & DataLoader

- Available datasets:
  - MNIST and FashionMNIST
  - COCO (Captioning and Detection)
  - LSUN Classification
  - ImageFolder
  - Imagenet-12
  - CIFAR10 and CIFAR100
  - STL10
  - SVHN
  - PhotoTour

Dataset & DataLoader

- Custom dataset

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):



    def __getitem__(self, index):
        return

    def __len__(self):
        return


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

1 download, read data, etc.

2 return one item on the index

3 return the data length

https://github.com/yunjey/pytorch-tutorial

Dataset & DataLoader

- Custom dataset

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

Using DataLoader

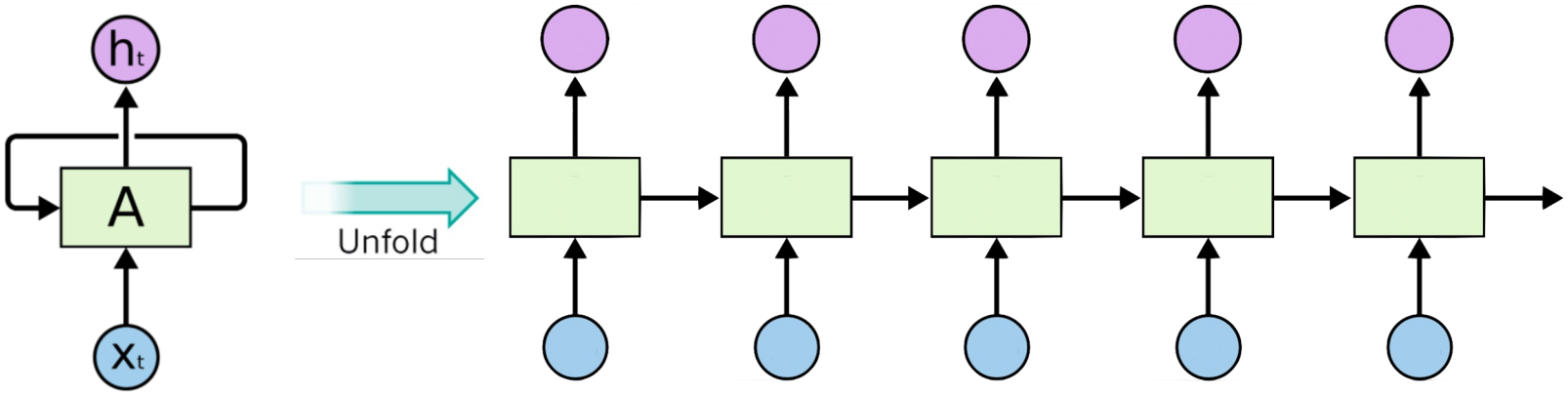
```
dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset, batch_size=32, shuffle=True, num_workers=2)


# Training loop
for epoch in range(2):
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # Forward pass: Compute predicted y by passing x to the model
        y_pred = model(inputs)

        # Compute and print loss
        loss = criterion(y_pred, labels)
        print(epoch, i, loss.data[0])

        # Zero gradients, perform a backward pass, and update the weights.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Classifying Diabetes

```python
# References
# https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/pytorch_basics/main.py
# http://pytorch.org/tutorials/beginner/data_loading_tutorial.html#dataset-class
import torch
import numpy as np
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader


class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

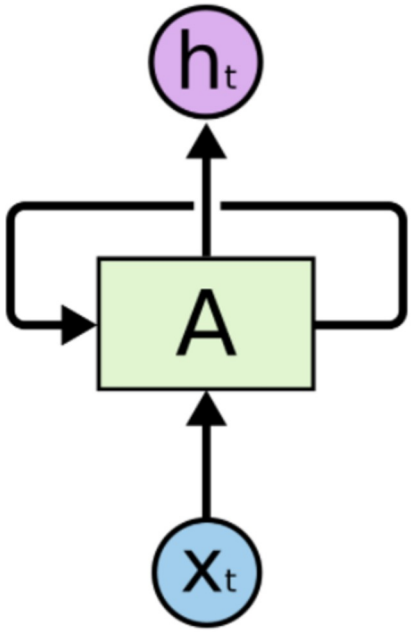
```python
class Model(torch.nn.Module):

    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = torch.nn.Linear(8, 6)
        self.l2 = torch.nn.Linear(6, 4)
        self.l3 = torch.nn.Linear(4, 1)

        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        return y_pred

# our model
model = Model()

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Training loop
for epoch in range(2):
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # Forward pass: Compute predicted y by passing x to the model
        y_pred = model(inputs)

        # Compute and print loss
        loss = criterion(y_pred, labels)
        print(epoch, i, loss.data[0])

        # Zero gradients, perform a backward pass, and update the weights.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

RNN



http://practicalquant.blogspot.hk/2013/10/deep-learning-oral-traditions.html

RNN in PyTorch

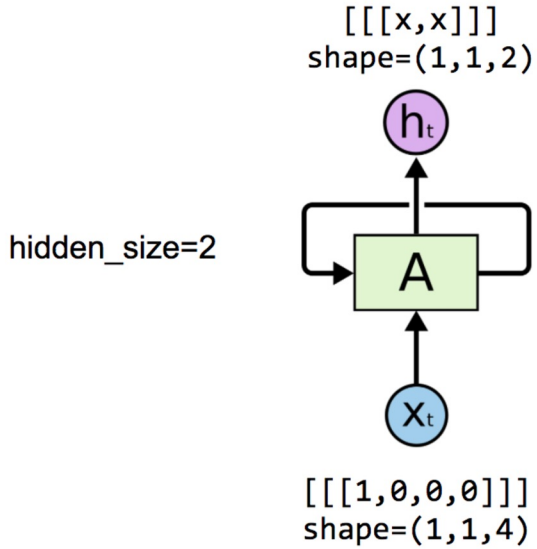



```
cell = nn.LSTM(input_size=4, hidden_size=2,
batch_first=True)


...


inputs = ... # (batch_size, seq_len, input_size) with batch_first=True
hidden = (... , ...) # (num_layers, batch_size, hidden_size)

# Bidirectional RNN*
hidden = (... , ...) # (num_layers*2, batch_size, hidden_size)
...


out, hidden = cell(inputs, hidden)
```

One node: 4 (*input_dim*) in 2 (*hidden_size*)

```
# One cell RNN input_dim (4) -> output_dim (2)
cell = nn.RNN(input_size=4, hidden_size=2, batch_first=True)

# One letter input
inputs = torch.Tensor([[h]]) # rank = (1, 1, 4)

# initialize the hidden state.
# (num_layers * num_directions, batch, hidden_size)
hidden = (torch.randn(1, 1, 2)), torch.randn((1, 1, 2)))

# Feed to one element at a time.
# after each step, hidden contains the hidden state.
out, hidden = cell(inputs, hidden)
print("out", out.data)
```

```
-0.1243  0.0738
[torch.FloatTensor of size 1x1x2]
```

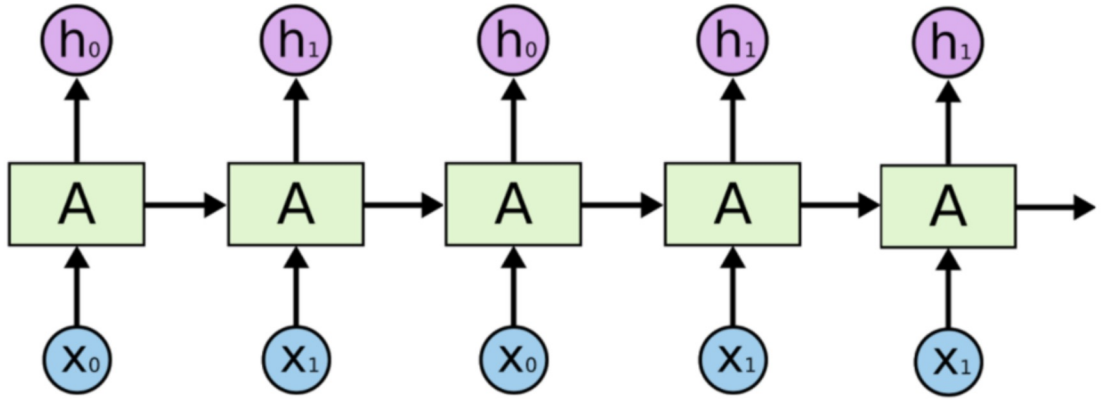



```
# One hot encoding
h = [1, 0, 0, 0]
e = [0, 1, 0, 0]
l = [0, 0, 1, 0]
o = [0, 0, 0, 1]
```

Unfolding to n sequences

hidden_size=2
sequance_length=5

shape=(1,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]]]



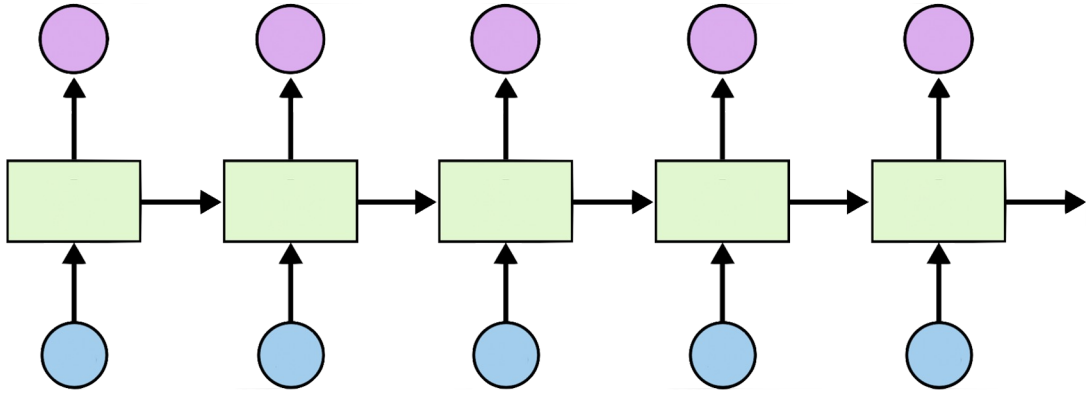
shape=(1,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]]]
h e l l o

https://github.com/hunkim/DeepLearningZeroToAll/blob/master/lab-12-0-rnn_basics.ipynb

Batching input

```
Hidden_size=2
sequence_length=5
batch_size=3
```

```
shape=(3,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]]]
```

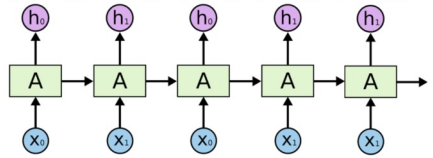


```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

Batching input

```
# One cell RNN input_dim (4) -> output_dim (2). sequence: 5, batch 3
# 3 batches 'hello', 'eolll', 'lleel'
# rank = (3, 5, 4)
inputs = torch.Tensor([[h, e, l, l, o],
                       [e, o, l, l, l],
                       [l, l, e, e, l]])
print("input size", inputs.size()) # input size torch.Size([3, 5, 4])

# (num_layers * num_directions, batch, hidden_size)
hidden = (torch.randn(1, 3, 2), torch.randn((1, 3, 2)))

out, hidden = cell(inputs, hidden)
print("out size", out.size()) # out size torch.Size([3, 5, 2])
```

```
shape=(3,5,2): [[[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]],
                [[x,x], [x,x], [x,x], [x,x], [x,x]]]
```

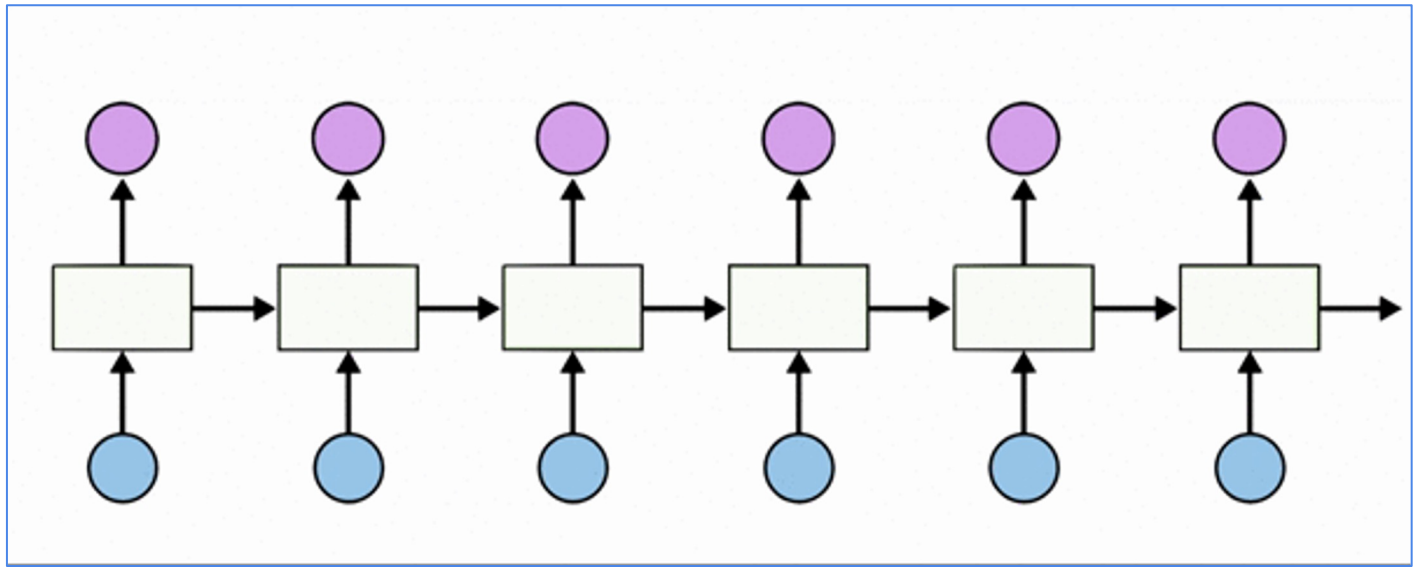



```
shape=(3,5,4): [[[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,1,0], [0,0,0,1]], # hello
                [[0,1,0,0], [0,0,0,1], [0,0,1,0], [0,0,1,0], [0,0,1,0]]  # eolll
                [[0,0,1,0], [0,0,1,0], [0,1,0,0], [0,1,0,0], [0,0,1,0]]] # lleel
```

Hidden_size=2
sequence_length=5
batch_size=3

Teach RNN ‘hihell’ to ‘ihello’

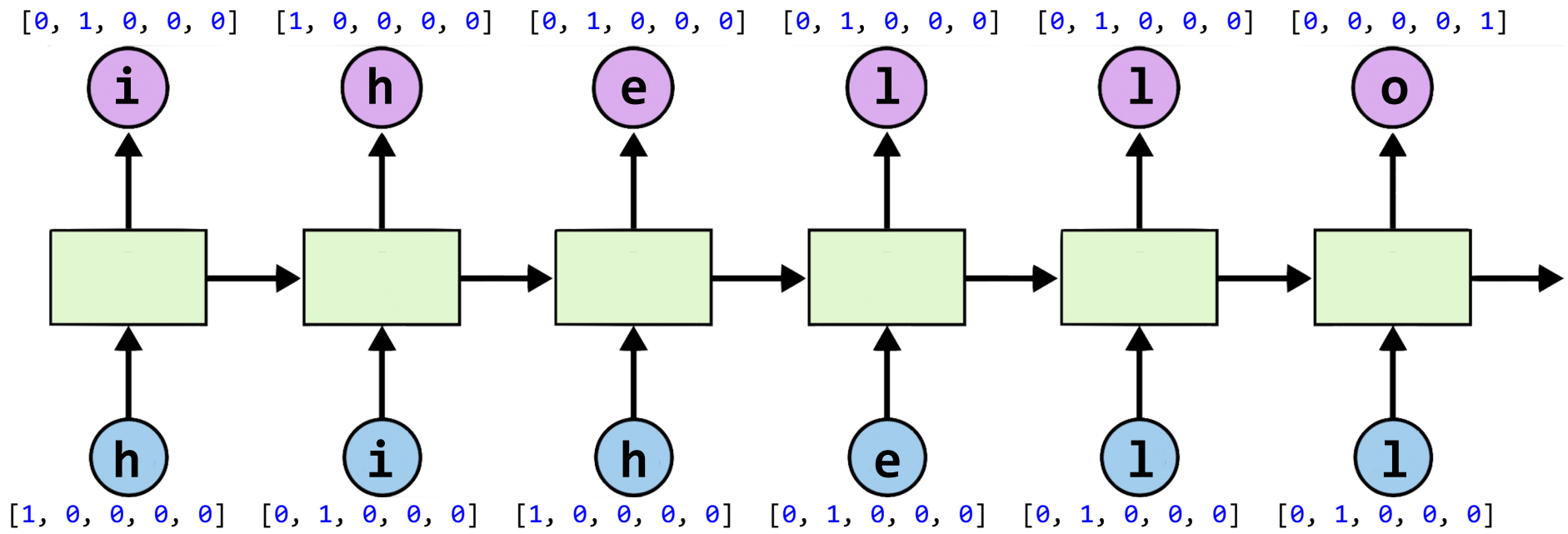
```
[1, 0, 0, 0, 0],   # h 0
[0, 1, 0, 0, 0],   # i 1
[0, 0, 1, 0, 0],   # e 2
[0, 0, 0, 1, 0],   # l 3
[0, 0, 0, 0, 1],   # o 4
```



Teach RNN ‘hihell’ to ‘ihello’

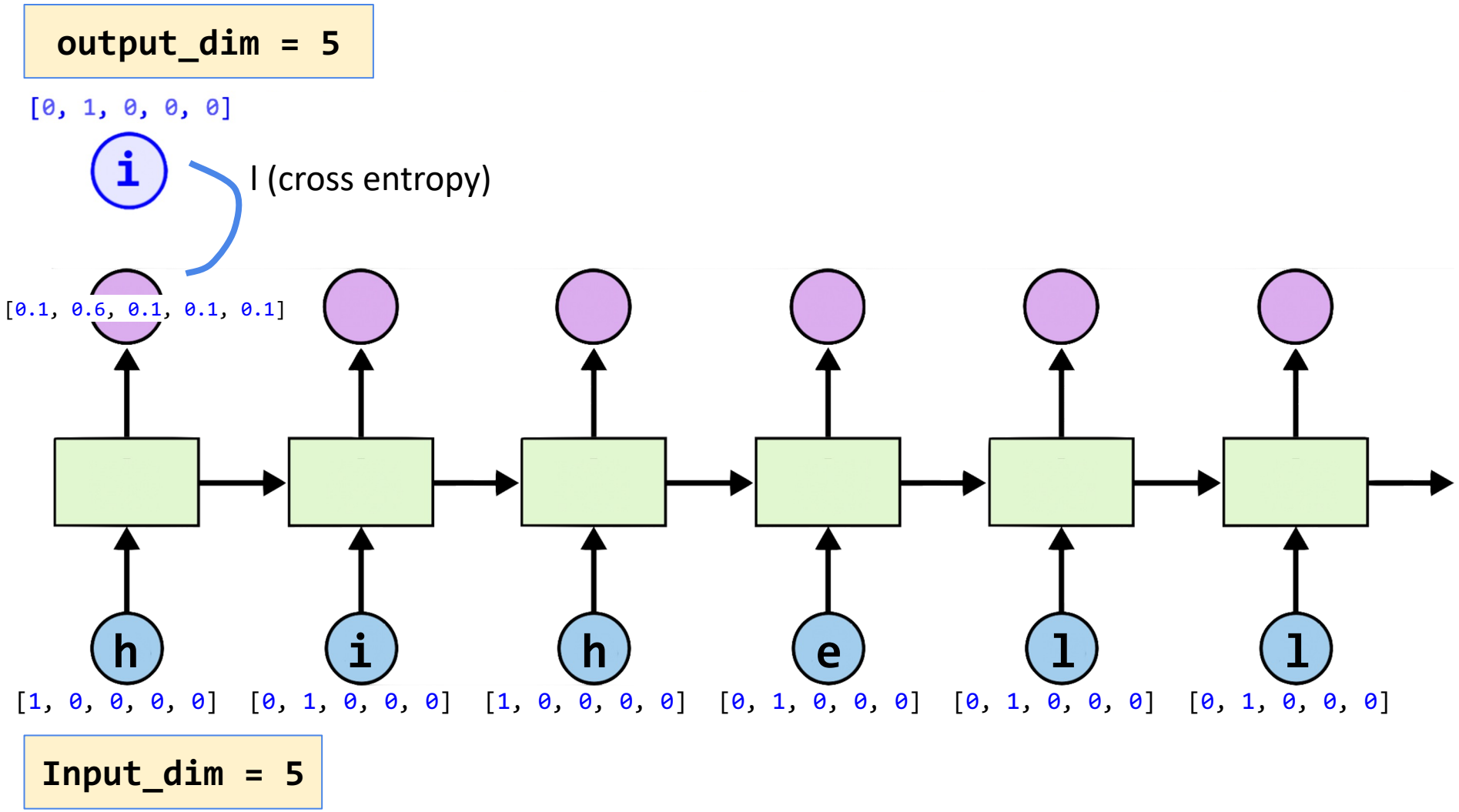
```
[1, 0, 0, 0, 0],   # h 0
[0, 1, 0, 0, 0],   # i 1
[0, 0, 1, 0, 0],   # e 2
[0, 0, 0, 1, 0],   # l 3
[0, 0, 0, 0, 1],   # o 4
```

output_dim = 5

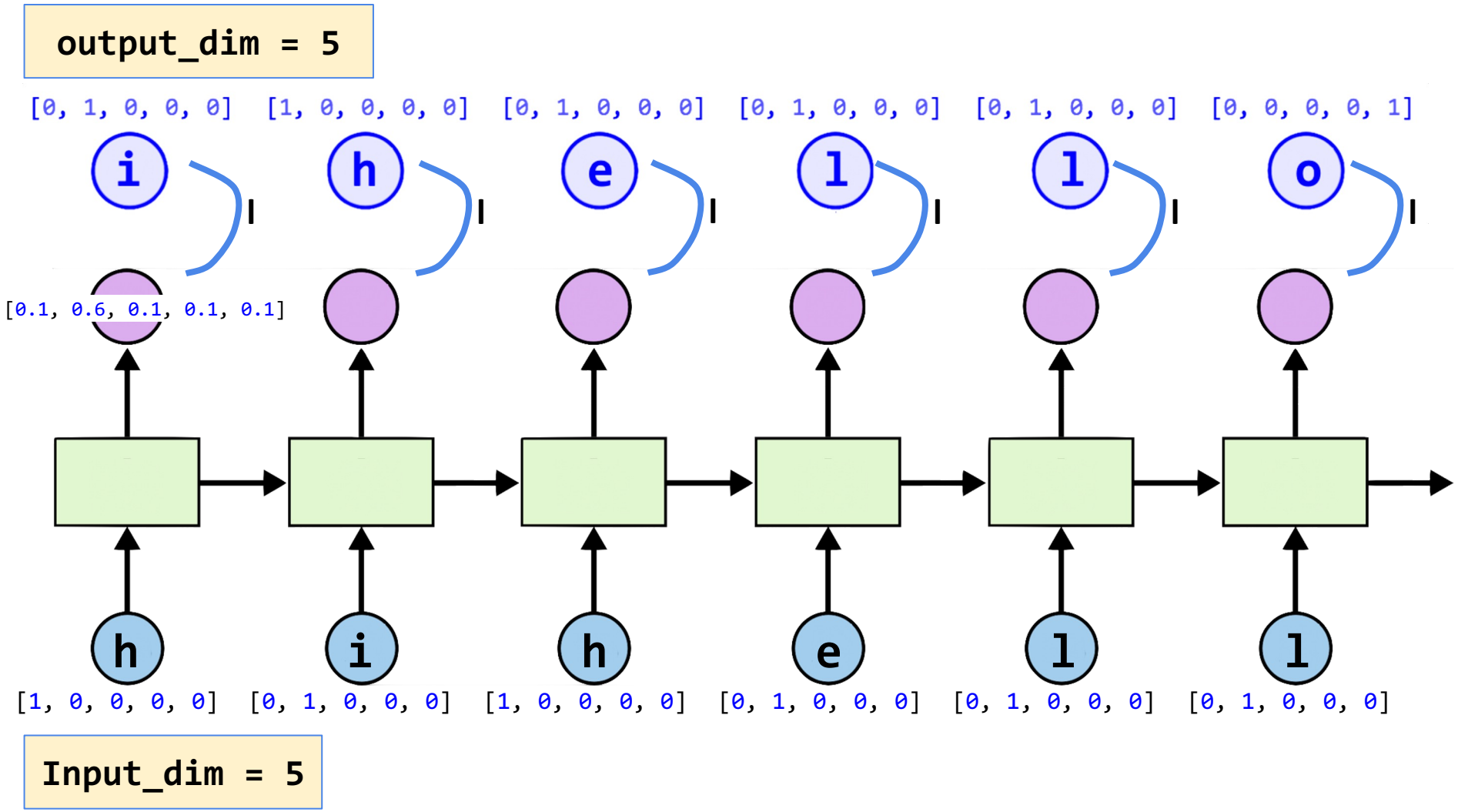


Input_dim = 5

Loss and training



Loss and training



(1) Data preparation - 1

```
idx2char = ['h', 'i', 'e', 'l', 'o']

# Teach hihell -> ihello
x_data = [[0, 1, 0, 2, 3, 3]]   # hihell
x_one_hot = [[[1, 0, 0, 0, 0],   # h 0
              [0, 1, 0, 0, 0],    # i 1
              [1, 0, 0, 0, 0],    # h 0
              [0, 0, 1, 0, 0],    # e 2
              [0, 0, 0, 1, 0],    # l 3
              [0, 0, 0, 1, 0]]]   # l 3

y_data = [1, 0, 2, 3, 3, 4]    # ihello

# As we have one batch of samples, we will change them to variables only once
inputs = torch.Tensor(x_one_hot)
labels = torch.LongTensor(y_data)
```

(1) Data preparation - 2

```
idx2char = ['h', 'i', 'e', 'l', 'o']

# Teach hihell -> ihello
x_data = [0, 1, 0, 2, 3, 3]   # hihell
one_hot_lookup =[[1, 0, 0, 0, 0],  # 0
                 [0, 1, 0, 0, 0],  # 1
                 [0, 0, 1, 0, 0],  # 2
                 [0, 0, 0, 1, 0],  # 3
                 [0, 0, 0, 0, 1]]  # 4

y_data = [1, 0, 2, 3, 3, 4]    # ihello
x_one_hot = [one_hot_lookup[x] for x in x_data]

# As we have one batch of samples, we will change them to variables only once
inputs = torch.Tensor(x_one_hot)
labels = torch.LongTensor(y_data)
```

(2) Our model

```
class Model(nn.Module):

    def __init__(self):
        super(Model, self).__init__()
        self.rnn = nn.RNN(input_size=input_size,
                          hidden_size=hidden_size, batch_first=True)

    def forward(self, hidden, x):
        # Reshape input in (batch_size, sequence_length, input_size)
        x = x.view(batch_size, sequence_length, input_size)

        # Propagate input through RNN
        # Input: (batch, seq_len, input_size)
        # hidden: (batch, num_layers * num_directions, hidden_size)
        out, hidden = self.rnn(x, hidden)
        out = out.view(-1, num_classes)
        return hidden, out

    def init_hidden(self):
        # Initialize hidden and cell states
        # (batch, num_layers * num_directions, hidden_size) for batch_first=True
        return torch.zeros(batch_size, num_layers, hidden_size)
```

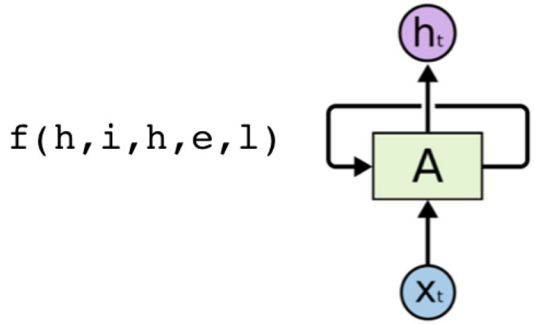
```
num_classes = 5
input_size = 5  # one-hot size
hidden_size = 5  # output from the RNN.
batch_size = 1   # one sentence
sequence_length = 1
num_layers = 1  # one-layer rnn
```

Designing Loss

```
out = rnn_out.view(-1, 5)
```

Predict Y, one of five {h, i, e, l, o} $Y \in R^{N \times 5}$

o `[0, 0, 0, 0, 1]`



CrossEntropy

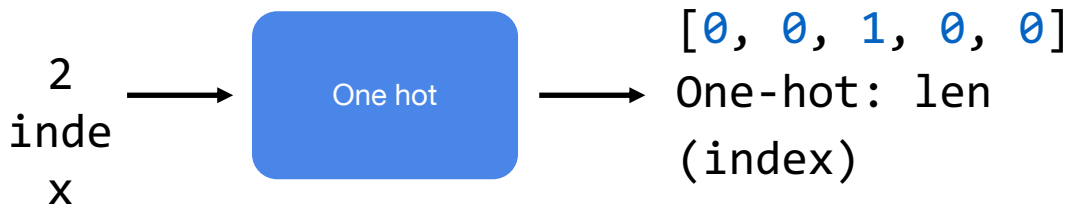
```
criterion = nn.CrossEntropyLoss()
```

Some input: X`1` `[0, 0, 0, 1, 0]`

(3) Loss & Training

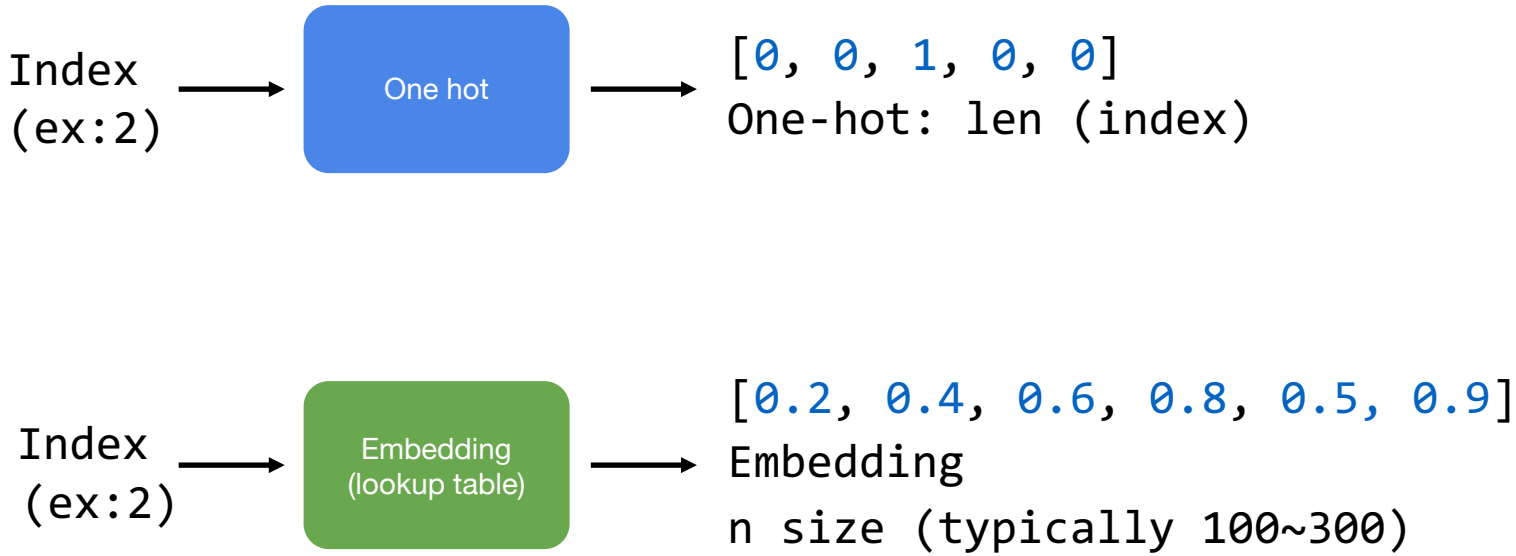
```
# Instantiate RNN model
model = Model()

# Set loss and optimizer function
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

# Train the model
for epoch in range(100):
    optimizer.zero_grad()
    loss = 0
    hidden = model.init_hidden()
    sys.stdout.write("predicted string: ")
    for input, label in zip(inputs, labels):
        # print(input.size(), label.size())
        hidden, output = model(hidden, input)
        val, idx = output.max(1)
        sys.stdout.write(idx2char[idx.data[0]])
        loss += criterion(output, label)

    print(", epoch: %d, loss: %1.3f" % (epoch + 1, loss.data[0]))

    loss.backward()
    optimizer.step()
```

One hot VS embedding



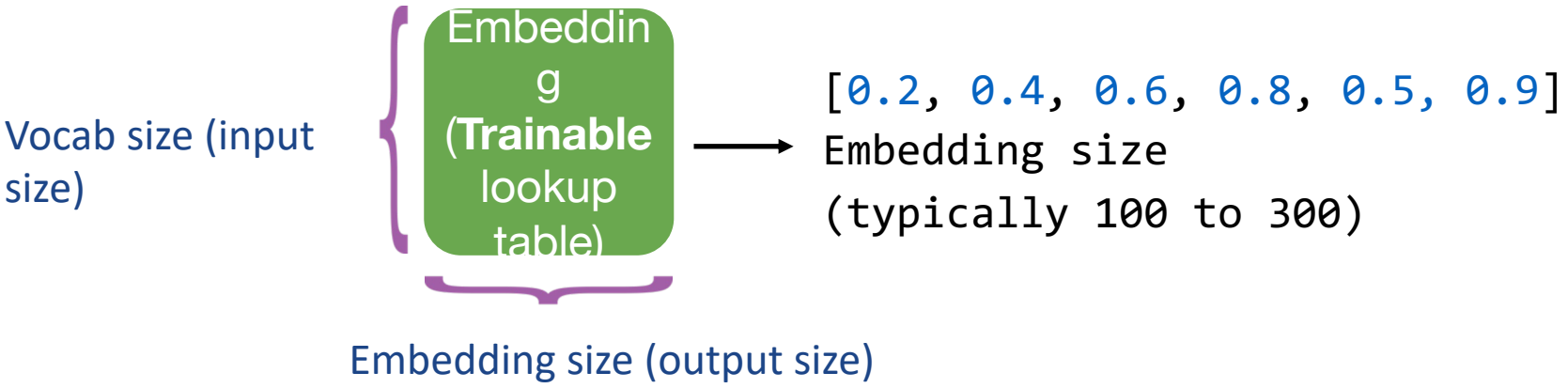

```
x_one_hot = [[[1, 0, 0, 0, 0],   # h 0
              [0, 1, 0, 0, 0],   # i 1
              [1, 0, 0, 0, 0],   # h 0
              [0, 0, 1, 0, 0],   # e 2
              [0, 0, 0, 1, 0],   # l 3
              [0, 0, 0, 1, 0]]]  # l 3
```

One hot VS embedding



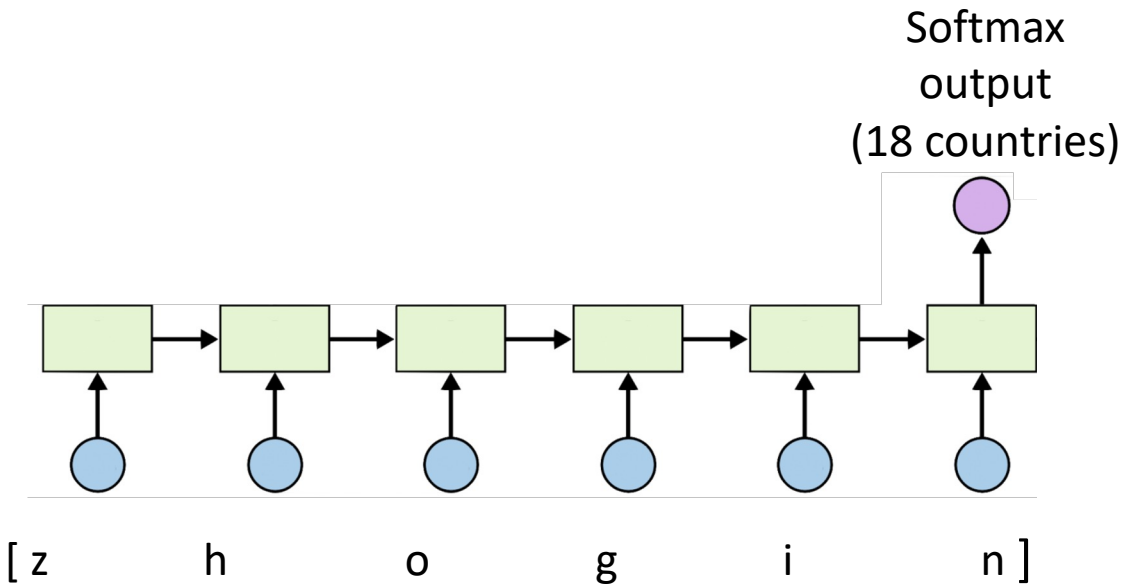
One hot VS embedding

```
self.embeddings = nn.Embedding(vocab_size, output_size)
...
emb = self.embeddings(x)
```



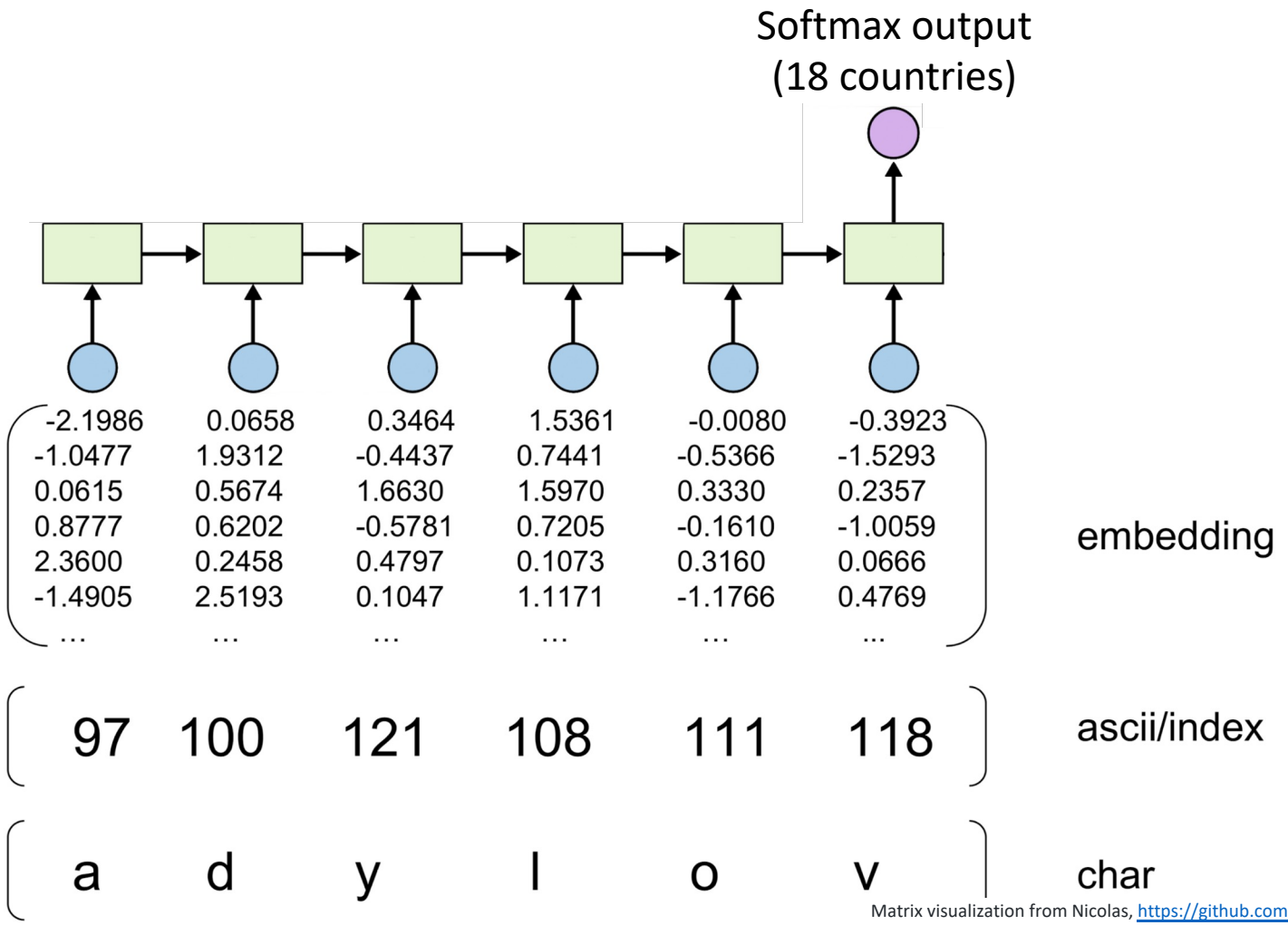
Name Classification: Dataset

0	Nader	Arabic
1	Malouf	Arabic
2	Terajima	Japanese
3	Huie	Chinese
4	Chertushkin	Russian
5	Davletkildeev	Russian
6	Movchun	Russian
7	Pokhvoschev	Russian
8	Zhogin	Russian
9	Hancock	English
10	Tomkins	English



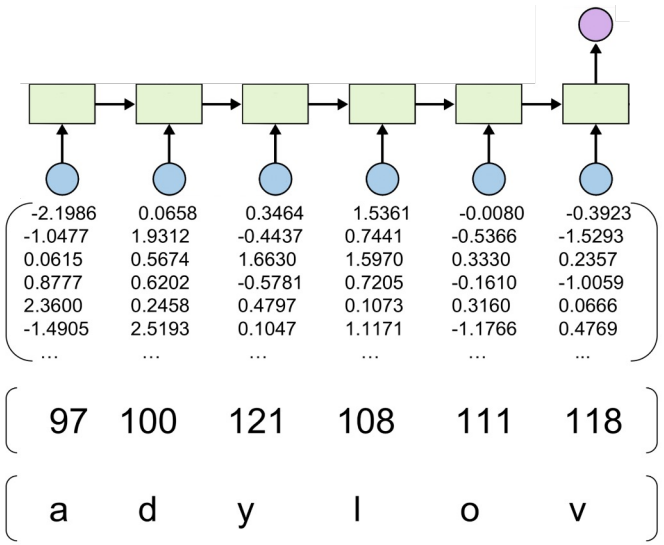
https://github.com/hunkim/PyTorchZeroToAll

Input representation: char embedding



Matrix visualization from Nicolas, https://github.com/ngarneau

Data preparation

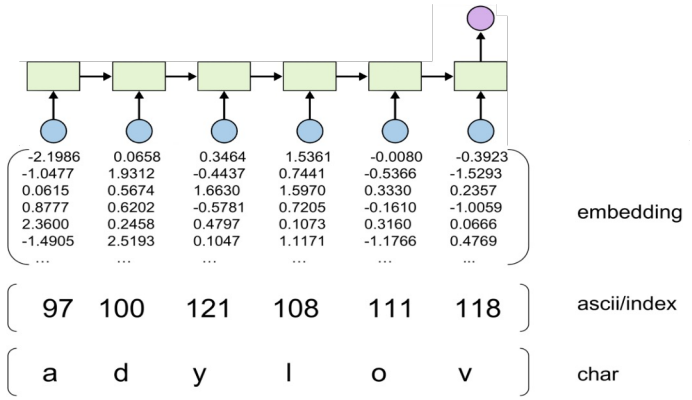



```
self.embedding =
 nn.Embedding(input_voc_size,
rnn_input_size)

...
embedded = self.embedding(input)
```

```
def str2ascii_arr(name):
    arr = [ord(c) for c in
name]
    return arr, len(arr)
```

Matrix visualization from Nicolas, https://github.com/ngarneau

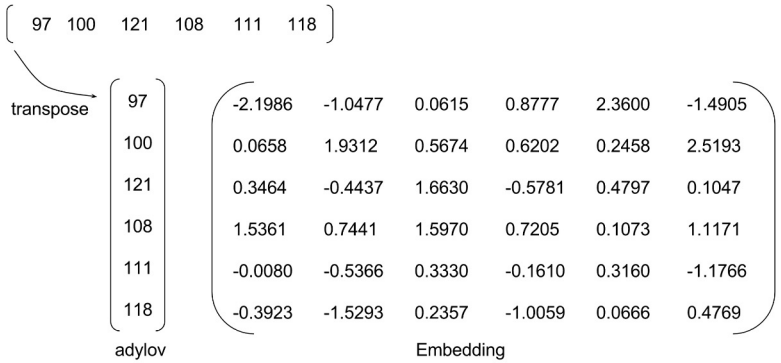
Implementation



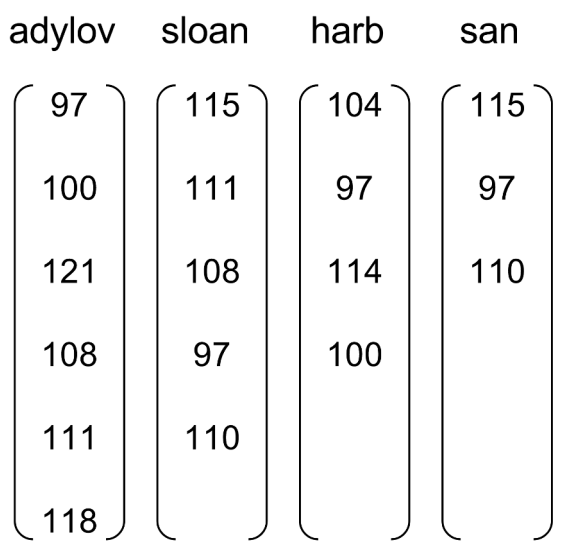

```
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS,
                               HIDDEN_SIZE, N_CLASSES)

    arr, _ = str2ascii_arr('adylov')
    inp = torch.LongTensor([arr])
    out = classifier(inp)
    print("in", inp.size(), "out", out.size())
    # in torch.Size([1, 6])
    #out torch.Size([1, 1, 18])
```

```
class RNNClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1):
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, input):
        # Note: we run this all at once (over the whole input sequence)
        # input = B x S . size(0) = B
        batch_size = input.size(0)

        # input:  B x S  -- (transpose) --> S x B
        input = input.t()

        # Embedding S x B -> S x B x I (embedding size)
        print("  input", input.size())
        embedded = self.embedding(input)
        print("  embedding", embedded.size())

        # Make a hidden
        hidden = self._init_hidden(batch_size)
        output, hidden = self.gru(embedded, hidden)
        print("  gru hidden output", hidden.size())
        # Use the last layer output as FC's input
        # No need to unpack, since we are going to use hidden
        fc_output = self.fc(hidden)
        print("  fc output", fc_output.size())
        return fc_output

    def _init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_size)
        return hidden
```

Batch?

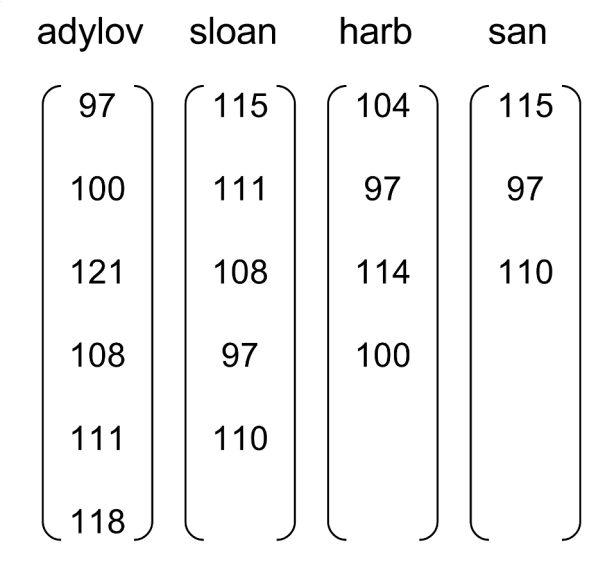



```python
if __name__ == '__main__':
    classifier = RNNClassifier(N_CHARS,
                               HIDDEN_SIZE, N_CLASSES)

    arr, _ = str2ascii_arr('adylov')
    inp = torch.LongTensor([arr])
    out = classifier(inp)
    print("in", inp.size(), "out", out.size())
    # in torch.Size([1, 6])
    #out torch.Size([1, 1, 18])
```

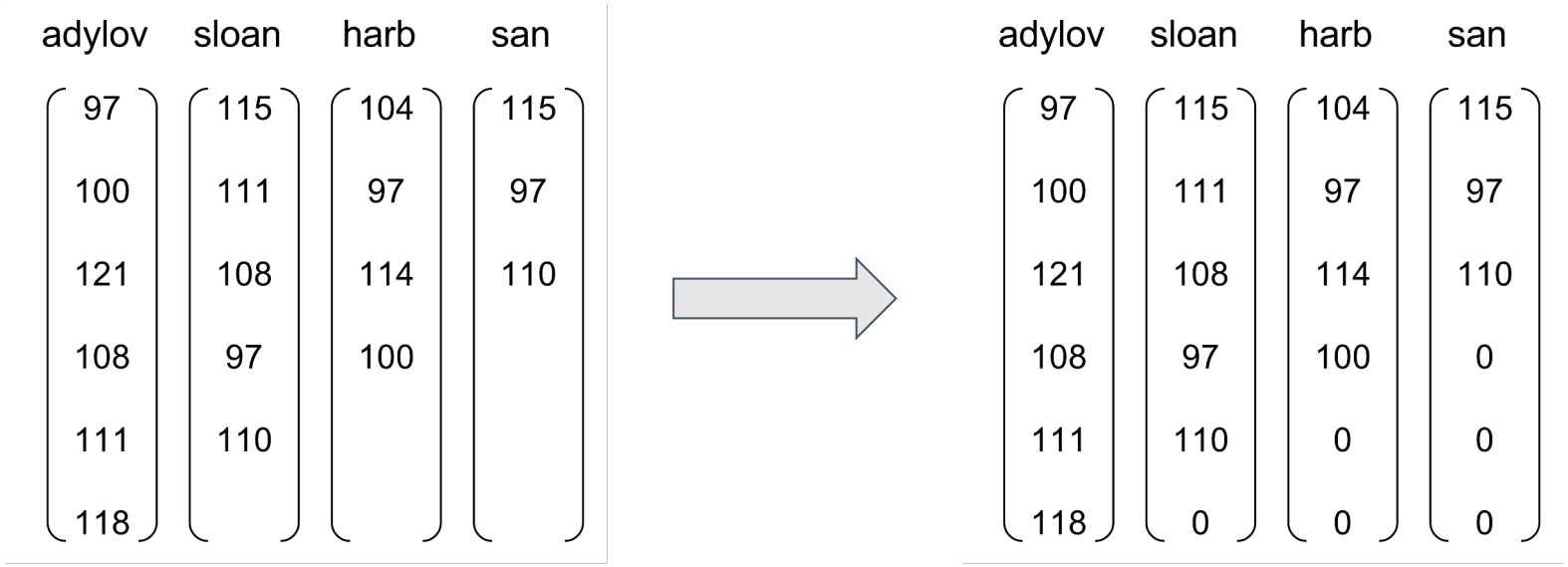



```python
if __name__ == '__main__':
    names = ['adylov', 'solan', 'hard', 'san']
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_CLASSES)

    for name in names:
        arr, _ = str2ascii_arr(name)
        inp = torch.LongTensor([arr])
        out = classifier(inp)
        print("in", inp.size(), "out", out.size())

    # in torch.Size([1, 6]) out torch.Size([1, 1, 18])
    # in torch.Size([1, 5]) out torch.Size([1, 1, 18])
    # ...
```

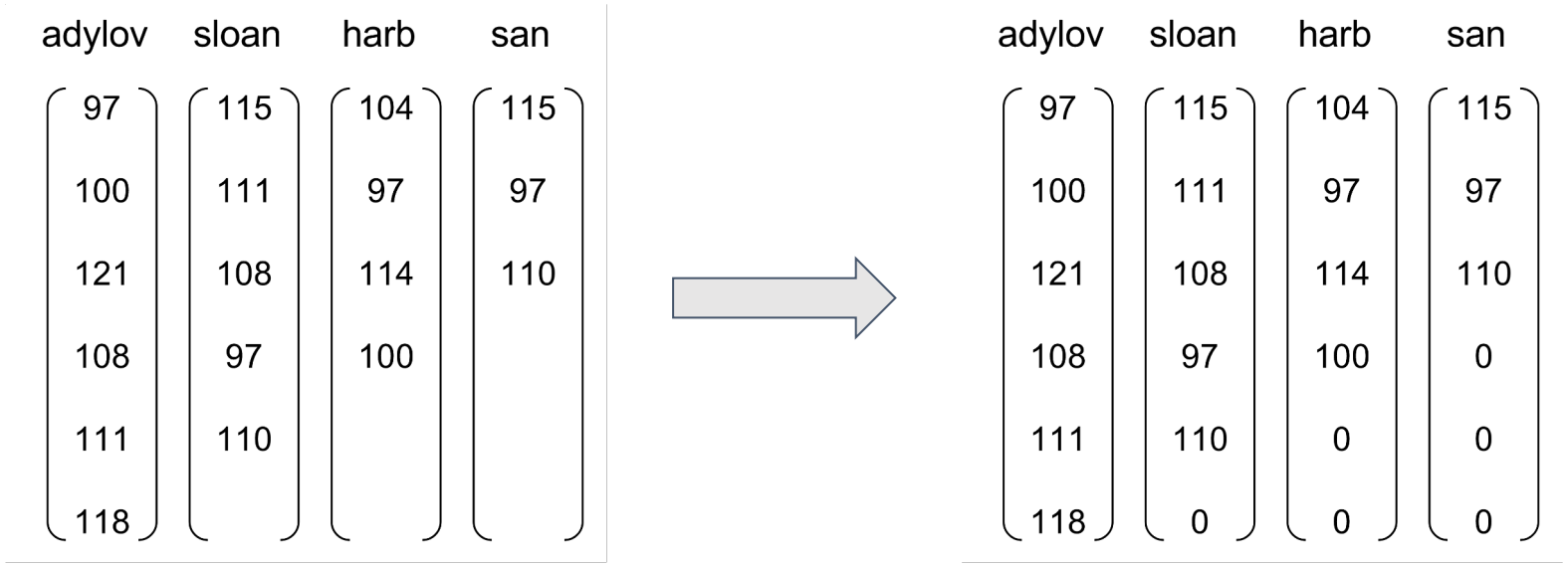
Zero padding



Zero padding



Zero padding




```
def pad_sequences(vectorized_seqs, seq_lengths):
    seq_tensor = torch.zeros((len(vectorized_seqs), seq_lengths.max())).long()
    for idx, (seq, seq_len) in enumerate(zip(vectorized_seqs, seq_lengths)):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)
    return  seq_tensor
```

Embedding

adylov	sloan	harb	san
97	115	104	115
100	111	97	97
121	108	114	110
108	97	100	0
111	110	0	0
118	0	0	0

-2.1986	-1.0477	0.0615	0.8777	2.3600	-1.4905	A			
0.4186	0.4627	-1.0357	-0.9848	3.0676	0.2172		S		
-1.8650	-0.6240	-1.7311	0.7280	0.3623	-0.1245			H	
0.4186	0.4627	-1.0357	-0.9848	3.0676	0.2172				S
0.0658	1.9312	0.5674	0.6202	0.2458	2.5193	d			
1.5361	0.7441	1.5970	0.7205	0.1073	1.1171		l		
1.5617	-0.6109	1.3775	0.7142	1.2291	0.5016			a	
1.5617	-0.6109	1.3775	0.7142	1.2291	0.5016				a
0.3464	-0.4437	1.6630	-0.5781	0.4797	0.1047	y			
-0.0080	-0.5366	0.3330	-0.1610	0.3160	-1.1766		o		
1.5768	-0.6135	-0.6442	0.1905	0.3791	0.1173			r	
0.3804	-1.9142	1.2423	1.1947	-0.7331	-1.2344				n
1.5361	0.7441	1.5970	0.7205	0.1073	1.1171	l			
1.5617	-0.6109	1.3775	0.7142	1.2291	0.5016		a		
0.6582	1.7296	-0.7542	-0.6202	0.2330	1.7715			b	
1.6387	-1.2216	-0.1584	1.9363	-0.3439	-0.2585				⌀
-0.0080	-0.5366	0.3330	-0.1610	0.3160	-1.1766	o			
0.3804	-1.9142	1.2423	1.1947	-0.7331	-1.2344		n		
1.6387	-1.2216	-0.1584	1.9363	-0.3439	-0.2585			⌀	
1.6387	-1.2216	-0.1584	1.9363	-0.3439	-0.2585				⌀
-0.3923	-1.5293	0.2357	-1.0059	0.0666	0.4769	v			
1.6387	-1.2216	-0.1584	1.9363	-0.3439	-0.2585		⌀		
1.6387	-1.2216	-0.1584	1.9363	-0.3439	-0.2585			⌀	
1.6387	-1.2216	-0.1584	1.9363	-0.3439	-0.2585				⌀

```
# Embedding S x B -> S x B x I (embedding size)
embedded = self.embedding(input)
```

Full implementation

```
def str2ascii_arr(msg):
    arr = [ord(c) for c in msg]
    return arr, len(arr)

# pad sequences and sort the tensor
def pad_sequences(vectorized_seqs, seq_lengths):
    seq_tensor = torch.zeros((len(vectorized_seqs), seq_lengths.max())).long()
    for idx, (seq, seq_len) in enumerate(zip(vectorized_seqs, seq_lengths)):
        seq_tensor[idx, :seq_len] = torch.LongTensor(seq)
    return seq_tensor

# Create necessary variables, lengths, and target
def make_variables(names):
    sequence_and_length = [str2ascii_arr(name) for name in names]
    vectorized_seqs = [sl[0] for sl in sequence_and_length]
    seq_lengths = torch.LongTensor([sl[1] for sl in sequence_and_length])
    return pad_sequences(vectorized_seqs, seq_lengths)


if __name__ == '__main__':
    names = ['adylov', 'solan', 'hard', 'san']
    classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, N_CLASSES)

    inputs = make_variables(names)
    out = classifier(inp)
    print("batch in", inp.size(), "batch out", out.size())

    # batch in torch.Size([4, 6]) batch out torch.Size([1, 4, 18])
```

```
optimizer =
torch.optim.Adam(classifier.parameters(),
                                        lr=0.001)
criterion = nn.CrossEntropyLoss()

…

loss = criterion(output, target)

classifier.zero_grad()
loss.backward()
optimizer.step()
```

`torch.nn.utils.rnn.` **`PackedSequence`** ***(_cls, data, batch_sizes)*** [source]

Holds the data and list of batch_sizes of a packed sequence.

All RNN modules accept packed sequences as inputs.

Note

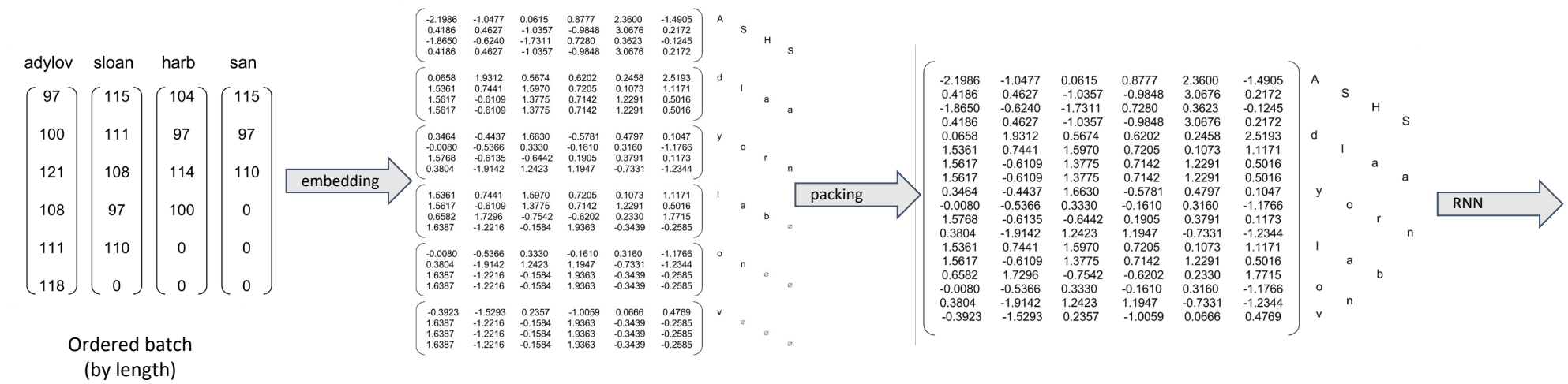
Instances of this class should never be created manually. They are meant to be instantiated by functions like `pack_padded_sequence()`.

Variables:

- **data** (*Variable*) – Variable containing packed sequence
- **batch_sizes** (*list[int]*) – list of integers holding information about the batch size at each sequence step

Efficiently handling batched sequences with variable lengths

- pack_padded_sequence

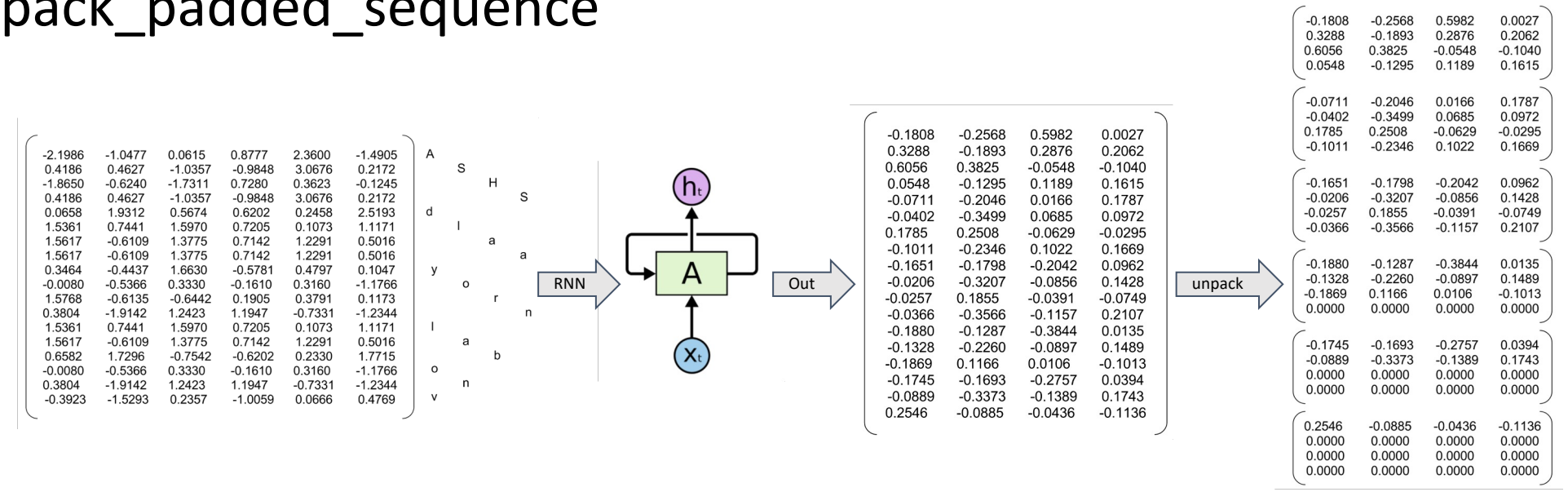


Matrix visualization from Nicolas, https://github.com/ngarneau

https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_4_pack_pad.py

Efficiently handling batched sequences with variable lengths

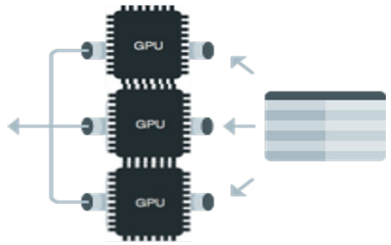
- pack_padded_sequence



Matrix visualization from Nicolas, https://github.com/ngarneau

https://github.com/hunkim/PyTorchZeroToAll/blob/master/13_4_pack_pad.py

GPU/Data Parallel



1 Copy all tensors to gpu

```
if torch.cuda.is_available():
   return tensor.cuda()
else:
   return tensor
```

2 Put your models on gpu

```
classifier = RNNClassifier(N_CHARS, HIDDEN_SIZE, ...
if torch.cuda.is_available():
   classifier.cuda()
```

3 Wrap your model using data parallel

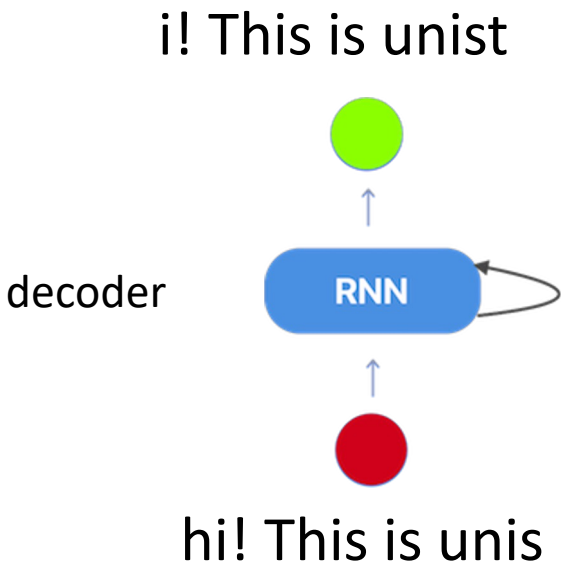
```
if torch.cuda.device_count() > 1:
   print("Let's use", torch.cuda.device_count(), "GPUs!")
   # dim = 0 [33, xxx] -> [11, ...], [11, ...], [11, ...] on 3 GPUs
   classifier = nn.DataParallel(classifier)
```

http://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html#sphx-glr-beginner-blitz-data-parallel-tutorial-py

(Character level) Language Modeling

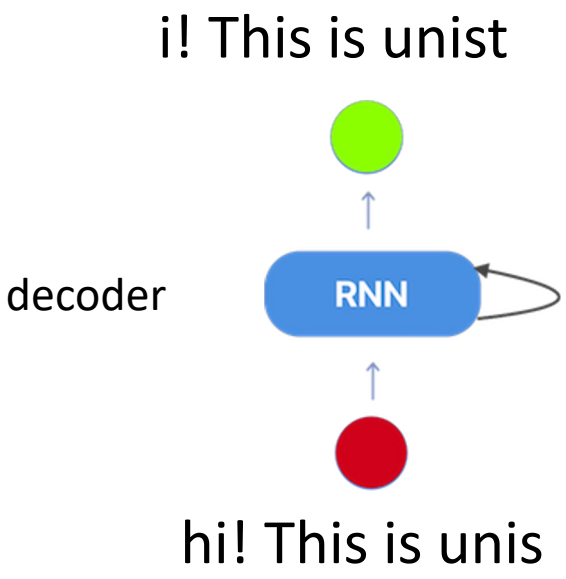
$$p(y_t|y_1, y_2, \cdots, y_{t-1})$$

Language Modeling using RNN



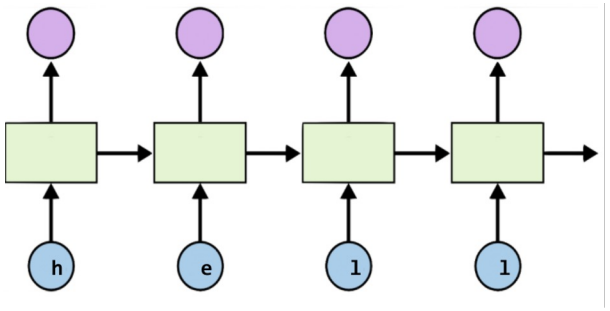

```
def generate(decoder, prime_str='A', predict_len=100, temperature=0.8):
    hidden = decoder.init_hidden()
    prime_input = str2tensor(prime_str)
    predicted = prime_str

    # Use priming string to "build up" hidden state
    for p in range(len(prime_str) - 1):
        _, hidden = decoder(prime_input[p], hidden)

    inp = prime_input[-1]
    for p in range(predict_len):
        output, hidden = decoder(inp, hidden)

        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]

        # Add predicted character to string and use as next input
        predicted_char = chr(top_i)
        predicted += predicted_char
        inp = str2tensor(predicted_char)

    return predicted
```

Training




```
def train_teacher_forching(line):
    input = str2tensor(line[:-1])
    target = str2tensor(line[1:])

    hidden = RNNdecoder.init_hidden()
    loss = 0

    for c in range(len(input)):
        output, hidden = decoder(input[c], hidden)
        loss += criterion(output, target[c])

    decoder.zero_grad()
    loss.backward()
    decoder_optimizer.step()

    return loss.data[0] / len(input)
```

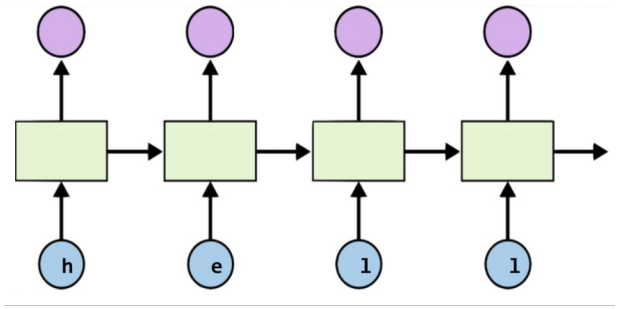
Teacher Forcing



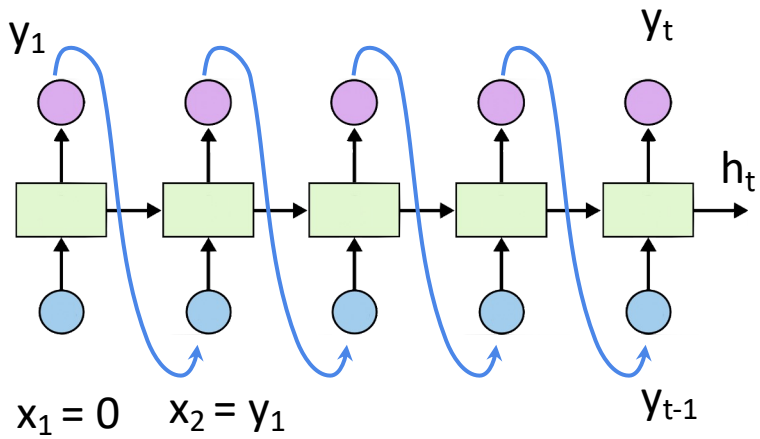
Teacher Forcing

```
def train_teacher_forching(line):
    input = str2tensor(line[:-1])
    target = str2tensor(line[1:])

    hidden = RNNdecoder.init_hidden()
    loss = 0

    for c in range(len(input)):
        output, hidden = decoder(input[c], hidden)
        loss += criterion(output, target[c])

    decoder.zero_grad()
    loss.backward()
    decoder_optimizer.step()

    return loss.data[0] / len(input)
```

Teacher Forcing



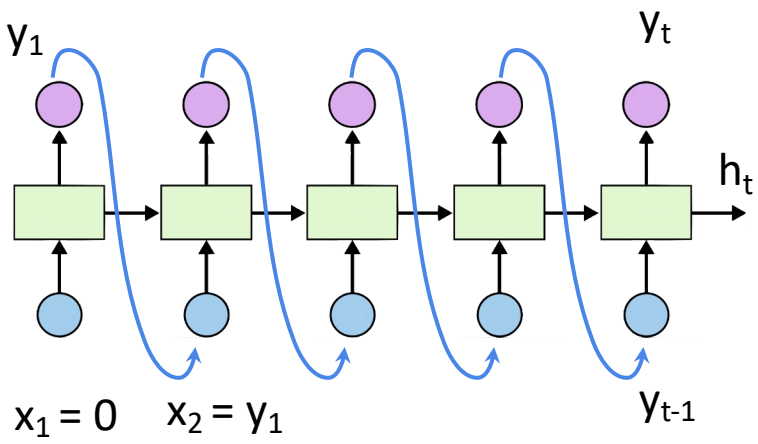
Teacher Forcing



No Teacher Forcing

https://www.quora.com/What-is-the-teacher-forcing-in-RNN

Training: no teacher forcing




```
def train(line):
    input = str2tensor(line[:-1])
    target = str2tensor(line[1:])

    hidden = decoder.init_hidden()
    decoder_in = input[0]
    loss = 0

    for c in range(len(input)):
        output, hidden = decoder(decoder_in, hidden)
        loss += criterion(output, target[c])
        decoder_in = output.max(1)[1]

    decoder.zero_grad()
    loss.backward()
    decoder_optimizer.step()

    return loss.data[0] / len(input)
```