

Recurrent Neural Networks

Samuel Cheng

School of ECE
University of Oklahoma

Spring, 2018

(Slides credit to Stanford CS231n and Hinton et al.)

Table of Contents

- 1 Motivation
- 2 Basic RNN
- 3 LSTM
- 4 Example: simple character-level language model
- 5 Example: image captioning
- 6 Overview of echo state networks
- 7 Conclusions

- We looked into couple use cases of CNNs previously
 - Recognition and localization
 - Object detection
 - Some use of CNNs for arts
- Up to now, the network models we have studied are all memoryless
 - We will discuss a non-memoryless model—recurrent neural networks today

Why non-memoryless models

- Almost all natural signals are sequential if we take time into account (we just cannot escape time)
 - Memory is needed to remember the past
- They also offer a simplified solution for some problems (for example, number addition)
- They can treat some unsupervised problems as supervised problems
 - Consider prediction of a stock: unsupervised? Supervised?

Why non-memoryless models

- Almost all natural signals are sequential if we take time into account (we just cannot escape time)
 - Memory is needed to remember the past
- They also offer a simplified solution for some problems (for example, number addition)
- They can treat some unsupervised problems as supervised problems
 - Consider prediction of a stock: unsupervised? Supervised?

Why non-memoryless models

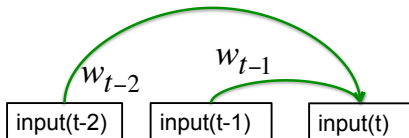
- Almost all natural signals are sequential if we take time into account (we just cannot escape time)
 - Memory is needed to remember the past
- They also offer a simplified solution for some problems (for example, number addition)
- They can treat some unsupervised problems as supervised problems
 - Consider prediction of a stock: unsupervised? Supervised?

Engineering hacks [Hinton 2012, week 7]

Memoryless models for sequences

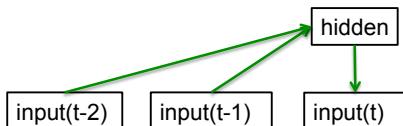
- Autoregressive models

Predict the next term in a sequence from a fixed number of previous terms using “delay taps”.



- Feed-forward neural nets

These generalize autoregressive models by using one or more layers of non-linear hidden units. *e.g. Bengio's first language model.*



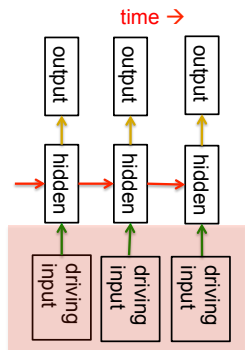
Non-memoryless models

- Benefit: memories increase the expressive power of the model
- Typically we do not know the exact values of the hidden states (that is why “hidden”). In many cases, the best we could do is just to infer a probability distribution over the hidden states
- Let's look at two classic examples

Non-memoryless models

- Benefit: memories increase the expressive power of the model
- Typically we do not know the exact values of the hidden states (that is why “hidden”). In many cases, the best we could do is just to infer a probability distribution over the hidden states
- Let's look at two classic examples

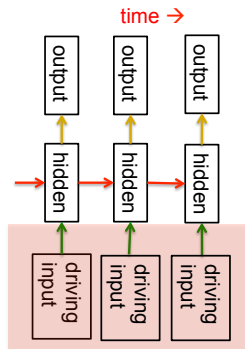
Linear dynamical systems (Engineers love them!)



- These are generative models with real **continuous** values as hidden states that cannot be observed directly
 - The hidden state has linear dynamics with Gaussian noise and produces the observations subjected to linear Gaussian noise
 - There can also be driving inputs
- To predict next output, we need to infer the hidden state

[Hinton 2012, Week 7]

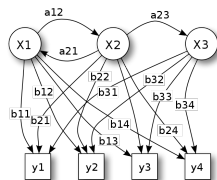
Linear dynamical systems (Engineers love them!)



- These are generative models with real **continuous** values as hidden states that cannot be observed directly
 - The hidden state has linear dynamics with Gaussian noise and produces the observations subjected to linear Gaussian noise
 - There can also be driving inputs
- To predict next output, we need to infer the hidden state

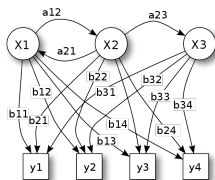
[Hinton 2012, Week 7]

State-Space Models (Computer scientists love them!)



- State-Space Models or Hidden Markov Models (HMMs) have a **discrete** one-of- N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The output produced by a state are also stochastic
 - We don't know which state produced a given output. So the state is "hidden"
 - We can represent the probability distribution across N states with N numbers
- To predict next output, we need to infer the probability distribution over the hidden state

State-Space Models (Computer scientists love them!)



- State-Space Models or Hidden Markov Models (HMMs) have a **discrete** one-of- N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The output produced by a state are also stochastic
 - We don't know which state produced a given output. So the state is "hidden"
 - We can represent the probability distribution across N states with N numbers
- To predict next output, we need to infer the probability distribution over the hidden state

A fundamental limitation of state space models

- The only information stored in the model is which state the model currently is in
 - So with N hidden states it can only remember a maximum $\log(N)$ bits of information
- Consider the speech prediction of one half from earlier half
 - The syntax needs to fit (e.g. number and tense agreement)
 - The semantics needs to fit. The intonation needs to fit
 - The accent, rate, volume, and vocal tract characteristics must all fit
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half 2^{100} states

[Hinton 2012, week 7]

A fundamental limitation of state space models

- The only information stored in the model is which state the model currently is in
 - So with N hidden states it can only remember a maximum $\log(N)$ bits of information
- Consider the speech prediction of one half from earlier half
 - The syntax needs to fit (e.g. number and tense agreement)
 - The semantics needs to fit. The intonation needs to fit
 - The accent, rate, volume, and vocal tract characteristics must all fit
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half
 2^{100} states

[Hinton 2012, week 7]

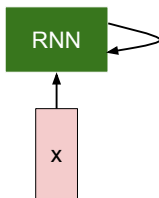
A fundamental limitation of state space models

- The only information stored in the model is which state the model currently is in
 - So with N hidden states it can only remember a maximum $\log(N)$ bits of information
- Consider the speech prediction of one half from earlier half
 - The syntax needs to fit (e.g. number and tense agreement)
 - The semantics needs to fit. The intonation needs to fit
 - The accent, rate, volume, and vocal tract characteristics must all fit
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half
 2^{100} states

[Hinton 2012, week 7]

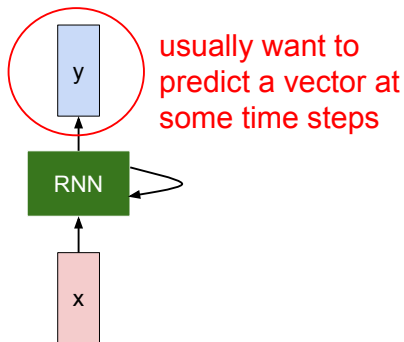
Recurrent neural networks (RNNs)

Recurrent Neural Network



Recurrent neural networks (RNNs)

Recurrent Neural Network



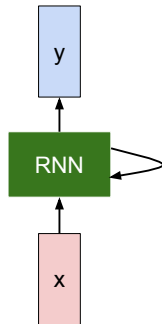
Recurrent neural networks (RNNs)

Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state some function with parameters W old state input vector at some time step



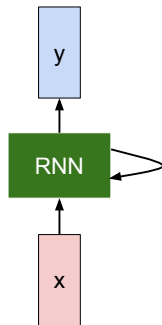
Recurrent neural networks (RNNs)

Recurrent Neural Network

We can process a sequence of vectors \mathbf{x} by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

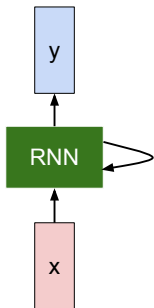
Notice: the same function and the same set of parameters are used at every time step.



Recurrent neural networks (RNNs)

(Vanilla) Recurrent Neural Network

The state consists of a single “hidden” vector h :



$$h_t = f_W(h_{t-1}, x_t)$$

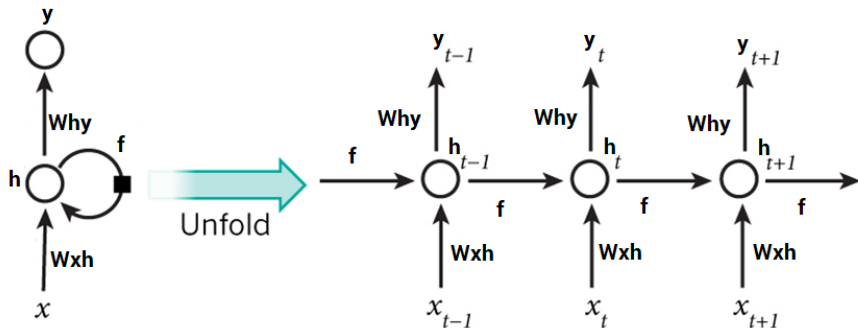


$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

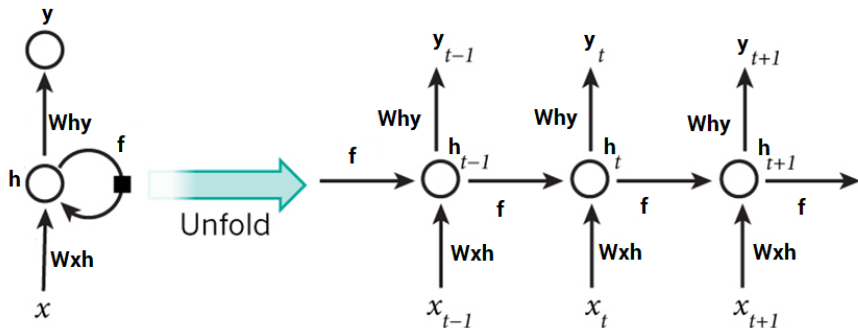
Back-Propagation Through Time (BPTT)

- For training, we can unroll all the time step to form a stack of activities and backprop will then similar to regular backprop
- The backward pass peels activities off the stack to compute the error derivatives at each time step
- After the backward pass we add together the derivatives at all the different times for each weight



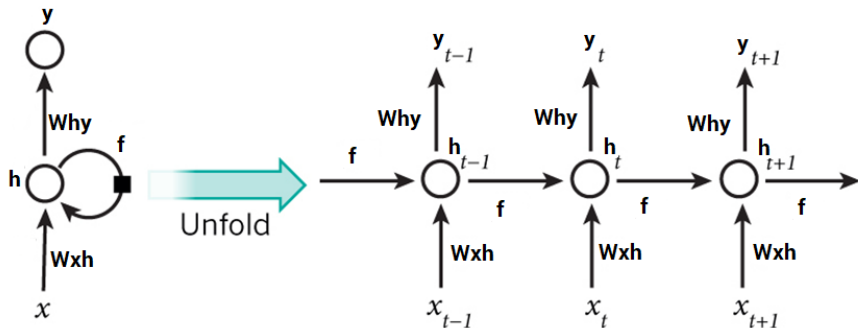
Back-Propagation Through Time (BPTT)

- For training, we can unroll all the time step to form a stack of activities and backprop will then similar to regular backprop
- The backward pass peels activities off the stack to compute the error derivatives at each time step
- After the backward pass we add together the derivatives at all the different times for each weight



Back-Propagation Through Time (BPTT)

- For training, we can unroll all the time step to form a stack of activities and backprop will then similar to regular backprop
- The backward pass peels activities off the stack to compute the error derivatives at each time step
- After the backward pass we add together the derivatives at all the different times for each weight



An irritative extra issue

- We need to specify the initial activity state of all the hidden and output units
- We could just fix these initial states to have some default value like 0.5
- But it is better to treat the initial states as learned parameters
- We learn them in the same way as we learn the weights
 - Start off with an initial random guess for the initial states
 - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state
 - Adjust the initial states by following the negative gradient

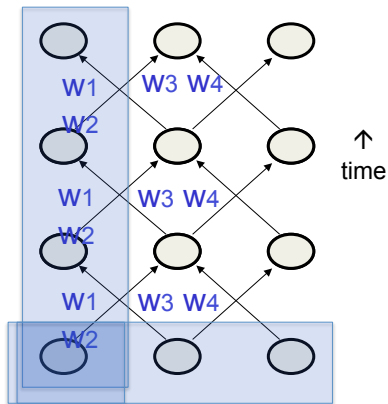
An irritative extra issue

- We need to specify the initial activity state of all the hidden and output units
- We could just fix these initial states to have some default value like 0.5
- But it is better to treat the initial states as learned parameters
- We learn them in the same way as we learn the weights
 - Start off with an initial random guess for the initial states
 - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state
 - Adjust the initial states by following the negative gradient

An irritating extra issue

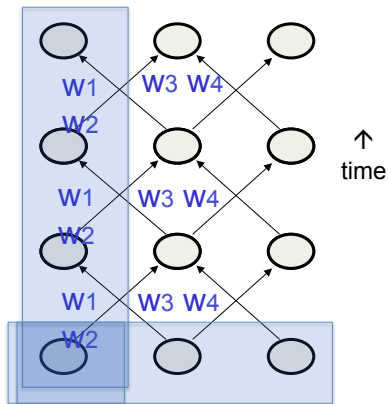
- We need to specify the initial activity state of all the hidden and output units
- We could just fix these initial states to have some default value like 0.5
- But it is better to treat the initial states as learned parameters
- We learn them in the same way as we learn the weights
 - Start off with an initial random guess for the initial states
 - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state
 - Adjust the initial states by following the negative gradient

Providing inputs to recurrent networks



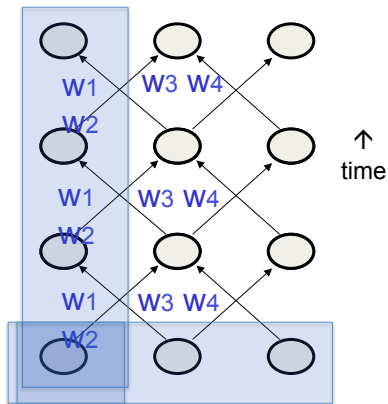
- We can specify inputs in several ways:
 - Specify the initial states of all the units
 - Specify the initial states of a subset of the units
 - Specify the states of the same subset of the units at every time step

Providing inputs to recurrent networks



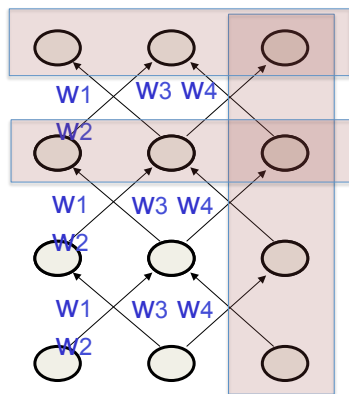
- We can specify inputs in several ways:
 - Specify the initial states of all the units
 - Specify the initial states of a subset of the units
 - Specify the states of the same subset of the units at every time step

Providing inputs to recurrent networks



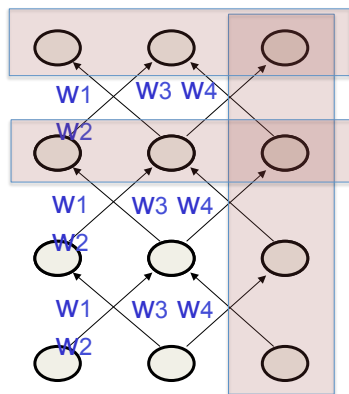
- We can specify inputs in several ways:
 - Specify the initial states of all the units
 - Specify the initial states of a subset of the units
 - Specify the states of the same subset of the units at every time step

Teaching recurrent networks to learn signals



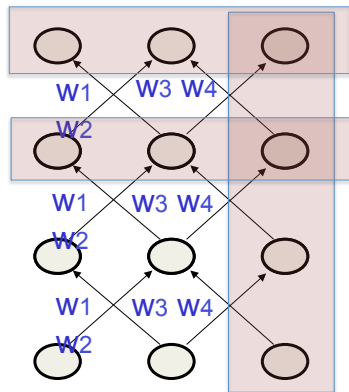
- We can specify targets in several ways:
 - Specify desired final activities of all the units
 - Specify desired activities of all units for the last few steps
 - Good for learning attractors
 - Specify the desired activity of a subset of the units.
 - The other units are input or hidden units.

Teaching recurrent networks to learn signals



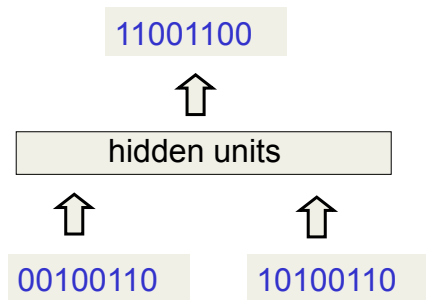
- We can specify targets in several ways:
 - Specify desired final activities of all the units
 - Specify desired activities of all units for the last few steps
 - Good for learning attractors
 - Specify the desired activity of a subset of the units.
 - The other units are input or hidden units.

Teaching recurrent networks to learn signals



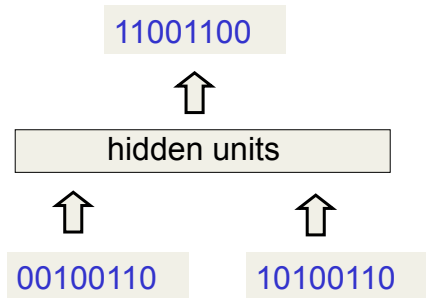
- We can specify targets in several ways:
 - Specify desired final activities of all the units
 - Specify desired activities of all units for the last few steps
 - Good for learning attractors
 - Specify the desired activity of a subset of the units.
 - The other units are input or hidden units.

Toy problem for RNN: binary addition



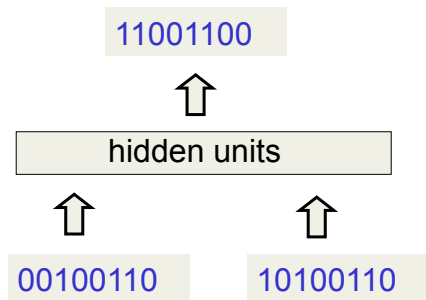
- We can train a feedforward net to do binary addition, but...
 - We must decide in advance the maximum number of digits in each number
 - We expect weights to process different bits to be the same, but it is tricky to enforce that
- As a result, feedforward nets do not generalize well for the binary addition task

Toy problem for RNN: binary addition



- We can train a feedforward net to do binary addition, but...
 - We must decide in advance the maximum number of digits in each number
 - We expect weights to process different bits to be the same, but it is tricky to enforce that
- As a result, feedforward nets do not generalize well for the binary addition task

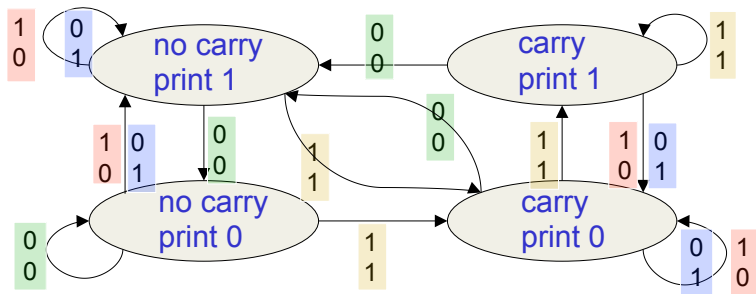
Toy problem for RNN: binary addition



- We can train a feedforward net to do binary addition, but...
 - We must decide in advance the maximum number of digits in each number
 - We expect weights to process different bits to be the same, but it is tricky to enforce that
- As a result, feedforward nets do not generalize well for the binary addition task

We are trying to learn this!

The algorithm for binary addition



This is a finite state automaton. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

A little bit detail

$$\mathbf{x} = [b_8, b_7, \dots, b_1]$$

$$\mathbf{y} = [c_8, c_7, \dots, c_1]$$

$$\mathbf{z} = \mathbf{x} + \mathbf{y} = [d_8, d_7, \dots, d_1]$$

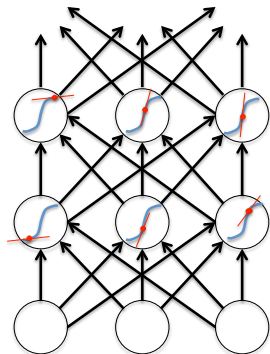
$$\hat{\mathbf{z}} = [\hat{d}_8, \hat{d}_7, \dots, \hat{d}_1]$$

Hidden unit: $h_i = \text{sigm}(W_{x,h}[b_i, c_i]^T + W_{h,h}h_{i-1})$

Output: $\hat{d}_i = \text{sigm}(W_{h,z}h_i)$

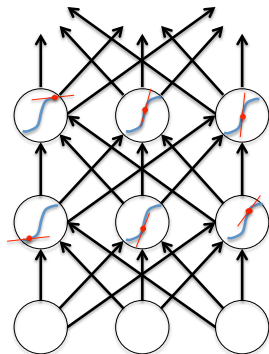
https://github.com/llSourcell/recurrent_neural_net_demo

Why training RNN is difficulty? The backward pass is linear



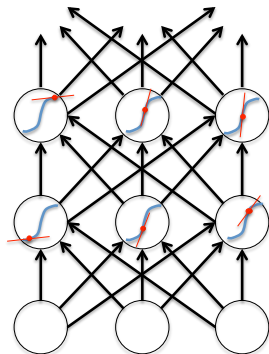
- There is a big difference between the forward and backward passes
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding
- The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double
 - The forward pass determines the slope of the linear function used for backpropagating through each neuron

Why training RNN is difficulty? The backward pass is linear



- There is a big difference between the forward and backward passes
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding
- The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double
 - The forward pass determines the slope of the linear function used for backpropagating through each neuron

Why training RNN is difficulty? The backward pass is linear



- There is a big difference between the forward and backward passes
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding
- The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double
 - The forward pass determines the slope of the linear function used for backpropagating through each neuron

The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the gradients shrink exponentially.
 - If the weights are big the gradients grow exponentially
- Typical feed-forward neural nets can cope with these exponential effects when they only have a few hidden layers
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish
 - We could avoid this by initializing the weights very carefully
- Even with good initial weights, the dependency of the current target output from an input many time-steps ago tends to be numerically unstable
 - So RNNs have difficulty dealing with long-range dependencies

The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the gradients shrink exponentially.
 - If the weights are big the gradients grow exponentially
- Typical feed-forward neural nets can cope with these exponential effects when they only have a few hidden layers
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish
 - We could avoid this by initializing the weights very carefully
- Even with good initial weights, the dependency of the current target output from an input many time-steps ago tends to be numerically unstable
 - So RNNs have difficulty dealing with long-range dependencies

The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the gradients shrink exponentially.
 - If the weights are big the gradients grow exponentially
- Typical feed-forward neural nets can cope with these exponential effects when they only have a few hidden layers
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish
 - We could avoid this by initializing the weights very carefully
- Even with good initial weights, the dependency of the current target output from an input many time-steps ago tends to be numerically unstable
 - So RNNs have difficulty dealing with long-range dependencies

The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
 - If the weights are small, the gradients shrink exponentially.
 - If the weights are big the gradients grow exponentially
- Typical feed-forward neural nets can cope with these exponential effects when they only have a few hidden layers
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish
 - We could avoid this by initializing the weights very carefully
- Even with good initial weights, the dependency of the current target output from an input many time-steps ago tends to be numerically unstable
 - So RNNs have difficulty dealing with long-range dependencies

Understanding gradient flow dynamics

Cute backprop signal video: <http://imgur.com/gallery/vaNahKE>

```
H = 5 # dimensionality of hidden state
T = 50 # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state
```

Understanding gradient flow dynamics

```

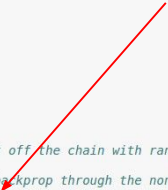
H = 5 # dimensionality of hidden state
T = 50 # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state

```

if the largest eigenvalue is > 1 , gradient will explode
 if the largest eigenvalue is < 1 , gradient will vanish



[On the difficulty of training Recurrent Neural Networks, Pascanu et al., 2013]

Four effective ways to learn an RNN

- **Long Short Term Memory:**
Make the RNN out of little modules that are designed to remember values for a long time
- **Hessian Free Optimization:**
Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature
 - The HF optimizer (Martens & Sutskever, 2011) is good at this
- **Echo State Networks:**
Initialize the input→ hidden and hidden→hidden and output→ hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input
 - ESNs only need to learn the hidden→output connections
- **Good initialization with momentum:** Initialize like in Echo State Networks, but then learn all of the connections using momentum

Four effective ways to learn an RNN

- **Long Short Term Memory:**
Make the RNN out of little modules that are designed to remember values for a long time
- **Hessian Free Optimization:**
Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature
 - The HF optimizer (Martens & Sutskever, 2011) is good at this
- **Echo State Networks:**
Initialize the input→ hidden and hidden→hidden and output→ hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input
 - ESNs only need to learn the hidden→output connections
- **Good initialization with momentum:** Initialize like in Echo State Networks, but then learn all of the connections using momentum

Four effective ways to learn an RNN

- **Long Short Term Memory:**
Make the RNN out of little modules that are designed to remember values for a long time
- **Hessian Free Optimization:**
Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature
 - The HF optimizer (Martens & Sutskever, 2011) is good at this
- **Echo State Networks:**
Initialize the input→ hidden and hidden→hidden and output→ hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input
 - ESNs only need to learn the hidden→output connections
- **Good initialization with momentum:** Initialize like in Echo State Networks, but then learn all of the connections using momentum

Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps)
 - Keep short-term memory for a long period of time, thus the name
- They designed a memory cell using logistic and linear units with multiplicative interactions
- Information gets into the cell whenever its “write” gate is on
- The information stays in the cell so long as its “keep” gate is on
- Information can be read from the cell by turning on its “read” gate

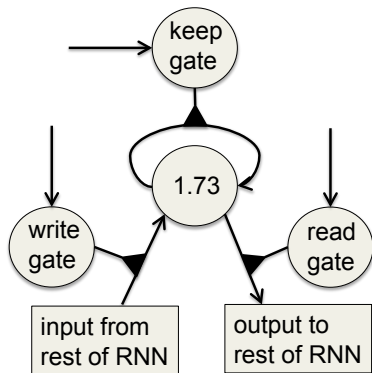
Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps)
 - Keep short-term memory for a long period of time, thus the name
- They designed a memory cell using logistic and linear units with multiplicative interactions
- Information gets into the cell whenever its “write” gate is on
- The information stays in the cell so long as its “keep” gate is on
- Information can be read from the cell by turning on its “read” gate

Long Short Term Memory (LSTM)

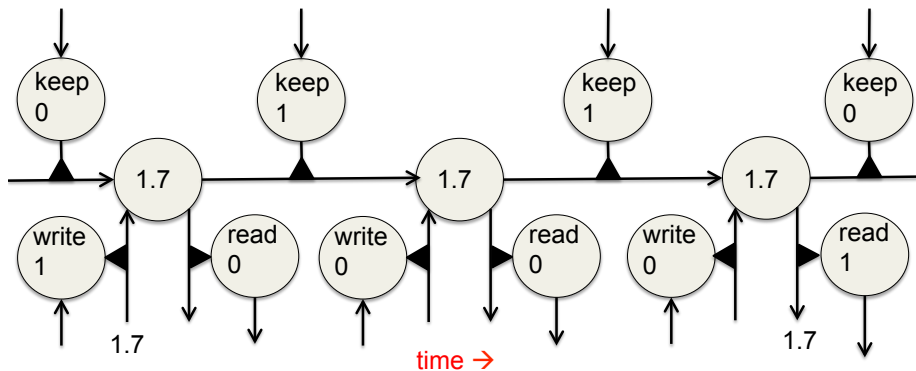
- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps)
 - Keep short-term memory for a long period of time, thus the name
- They designed a memory cell using logistic and linear units with multiplicative interactions
- Information gets into the cell whenever its “write” gate is on
- The information stays in the cell so long as its “keep” gate is on
- Information can be read from the cell by turning on its “read” gate

Implementing a memory cell in a neural network



- To preserve information for a long time in the activities of an RNN, we use a circuit mimicking an analog memory cell
 - Information is kept in the cell when "keep" gate is on
 - Information is stored in the cell by activating its write gate
 - Information is retrieved by activating the read gate
 - We can backpropagate through this circuit because logistics are have nice derivatives

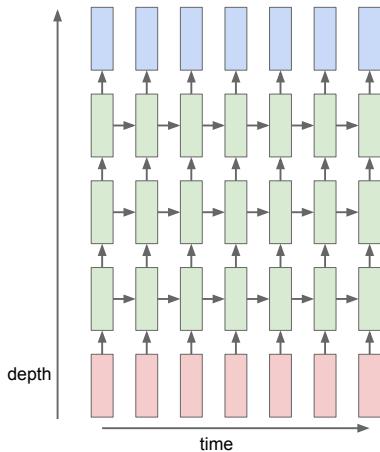
Backpropagation through a memory cell



RNN:

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$h \in \mathbb{R}^n, \quad W^l [n \times 2n]$$



Fei-Fei Li & Andrej Karpathy & Justin Johnson

Lecture 10 - 67

8 Feb 2016

RNN:

$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$h \in \mathbb{R}^n, \quad W^l [n \times 2n]$$

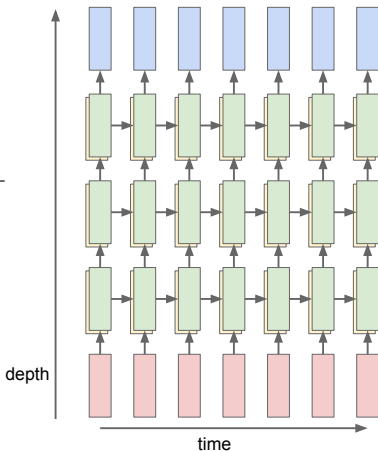
LSTM:

$$W^l [4n \times 2n]$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

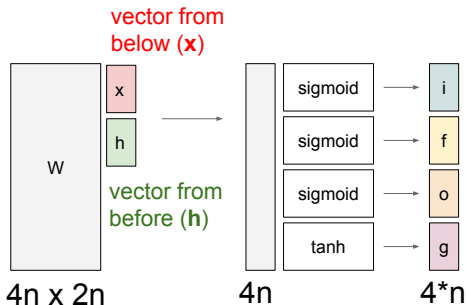
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



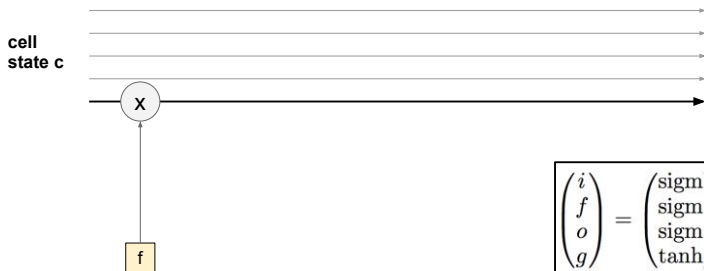
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



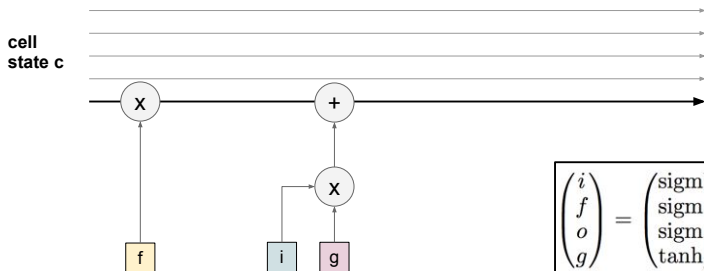
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_t^{l-1} \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



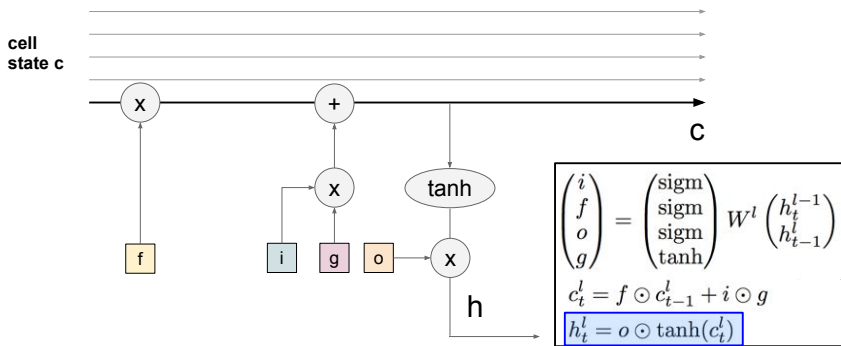
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_t^{l-1} \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



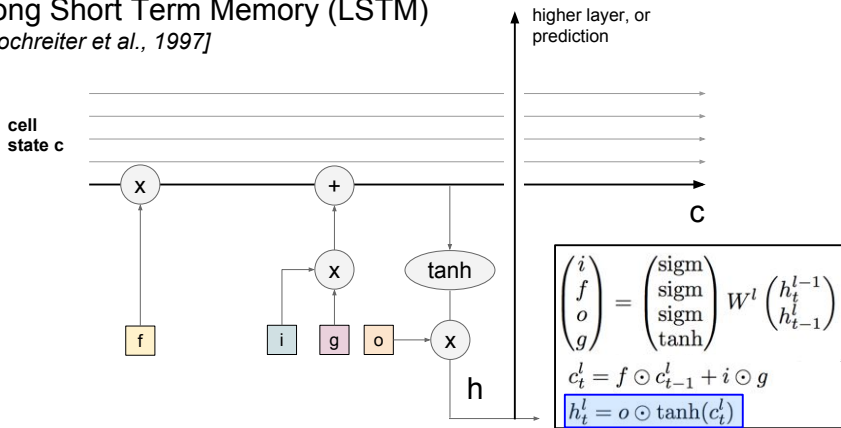
Fei-Fei Li & Andrej Karpathy & Justin Johnson

Lecture 10 - 73

8 Feb 2016

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

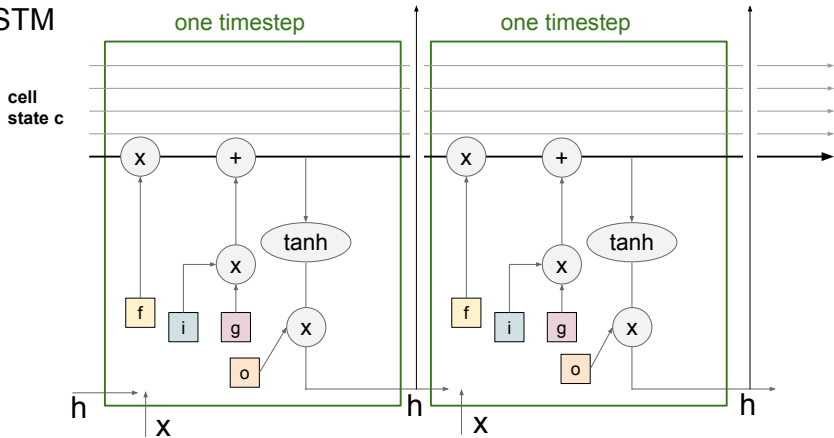


Fei-Fei Li & Andrej Karpathy & Justin Johnson

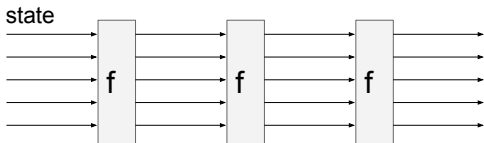
Lecture 10 - 74

8 Feb 2016

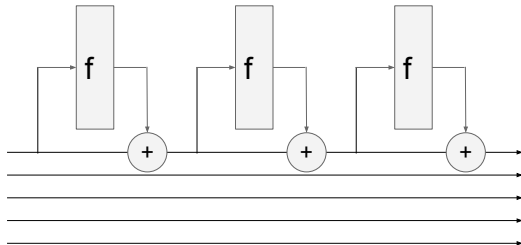
LSTM

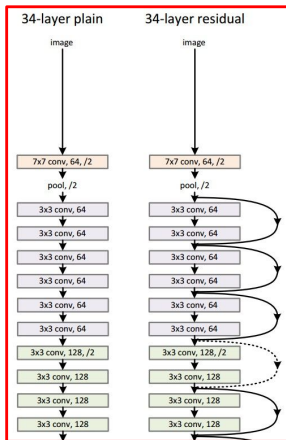


RNN



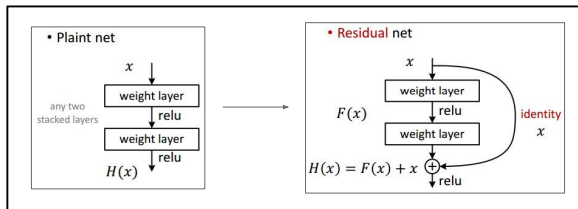
LSTM

(ignoring
forget gates)

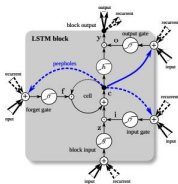


Recall: “PlainNets” vs. ResNets

ResNet is to PlainNet what LSTM is to RNN, kind of.



LSTM variants and friends



[LSTM: A Search Space Odyssey, Greff et al., 2015]

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$\begin{aligned}
 r_t &= \text{sigm}(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
 z_t &= \text{sigm}(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
 \tilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\
 h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
 \end{aligned}$$

[An Empirical Exploration of Recurrent Network Architectures, Jozefowicz et al., 2015]

MUT1:

$$\begin{aligned}
 z &= \text{sigm}(W_{xz}x_t + b_z) \\
 r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\
 h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + \tanh(x_t) + b_h) \odot z \\
 &+ h_t \odot (1 - z)
 \end{aligned}$$

MUT2:

$$\begin{aligned}
 z &= \text{sigm}(W_{xz}x_t + W_{hz}h_t + b_z) \\
 r &= \text{sigm}(x_t + W_{hr}h_t + b_r) \\
 h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{zh}x_t + b_h) \odot z \\
 &+ h_t \odot (1 - z)
 \end{aligned}$$

MUT3:

$$\begin{aligned}
 z &= \text{sigm}(W_{xz}x_t + W_{hz} \tanh(h_t) + b_z) \\
 r &= \text{sigm}(W_{xr}x_t + W_{hr}h_t + b_r) \\
 h_{t+1} &= \tanh(W_{hh}(r \odot h_t) + W_{zh}x_t + b_h) \odot z \\
 &+ h_t \odot (1 - z)
 \end{aligned}$$

Modelling text: why working with characters?

- The web is composed of character strings
- Any learning method powerful enough to understand the world by reading the web ought to find it trivial to learn which strings make words (this turns out to be true, as we shall see)
- Pre-processing text to get words is a big hassle
 - What about morphemes (prefixes, suffixes etc)
 - What about subtle effects like “sn” words?
 - What about New York vs new York Minster roof?
 - What about Finnish
 - ymmärtämättömyydellänsäkään

Modelling text: why working with characters?

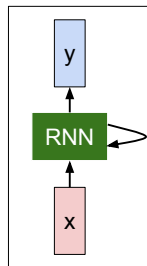
- The web is composed of character strings
- Any learning method powerful enough to understand the world by reading the web ought to find it trivial to learn which strings make words (this turns out to be true, as we shall see)
- Pre-processing text to get words is a big hassle
 - What about morphemes (prefixes, suffixes etc)
 - What about subtle effects like “sn” words?
 - What about New York vs new York Minster roof?
 - What about Finnish
 - ymmärtämättömyydellänsäkään

Simplest model: a first attempt

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

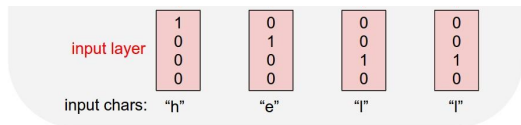


Simplest model: a first attempt

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



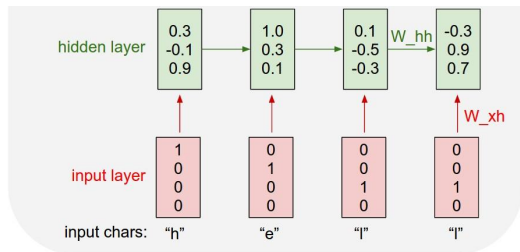
Simplest model: a first attempt

Character-level
language model
example

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Vocabulary:
[h,e,l,o]

Example training
sequence:
"hello"

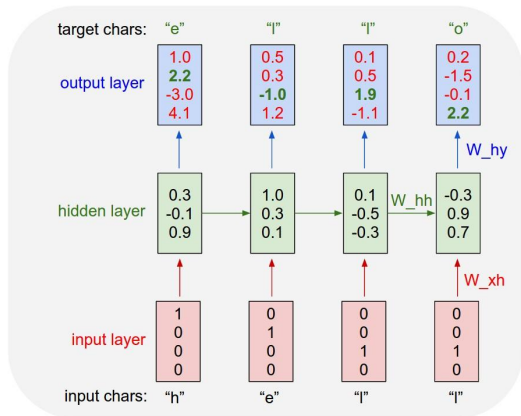


Simplest model: a first attempt

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
"hello"



Sampling

- Start the model with its default hidden state
- Give it a “burn-in” sequence of characters and let it update its hidden state after each character
- Then look at the probability distribution it predicts for the next character
- Pick a character randomly from that distribution and tell the net that this was the character that actually occurred
 - i.e. tell it that its guess was correct, whatever it guessed
- Continue to let it pick characters until bored

min-char-rnn.py gist: 112 lines of Python

```

1 """
2 MIN_CHAR character-level vanilla rnn model. Written by Andrej Karpathy (©Karpathy)
3 BSD License
4 """
5 import numpy as np
6
7 # Data I/O
8 data = open('input.txt', 'r').read() # should be simple plain text file
9 chars = list(set(data))
10 data_size, vocab_size = len(data), len(chars)
11 print('Data has %d characters, %d unique. %s (%d,%d,%d)' %
12      (data_size, vocab_size, data[:100], data_size, vocab_size))
13 #X, Y, X_1 = [c for l, c in enumerate(data)]
14 #X, Y, X_1 = [c for l, c in enumerate(chars)]
15
16 # Hyperparameters
17 HIDDEN_SIZE = 100 # size of hidden layer of neurons
18 rnn_length = 20 # number of steps to unroll the rnn for
19 learning_rate = 1e-3
20
21 # Model parameters
22 W0 = np.random.randn(HIDDEN_SIZE, Vocab_size) * 0.01 # input to hidden
23 W1 = np.random.randn(HIDDEN_SIZE, HIDDEN_SIZE) * 0.01 # hidden to hidden
24 W2 = np.random.randn(HIDDEN_SIZE, HIDDEN_SIZE) * 0.01 # hidden to output
25 W3 = np.zeros((Vocab_size, 1)) # hidden bias
26 W4 = np.zeros((Vocab_size, 1)) # output bias
27
28 def lstmCell(inputs, targets, hprev):
29     """
30     inputs, targets are both list of integers.
31     hprev is the array of hidden states
32     returns the loss, gradients on model parameters, and last hidden state
33     """
34     W, U, W0, W1, W2, W3, W4 = W0, W1, W2, W3, W4, W0, W1, W2, W3, W4
35     h0 = np.zeros(HIDDEN_SIZE)
36     loss = 0
37     for i in range(len(inputs)):
38         x = inputs[i]
39         h = np.tanh(W0[x] + W1[h0] + W2[hprev]) * \
40             np.sigmoid(W3[x] + W4[h0] + W1[hprev]) # hidden state
41         y = inputs[i]
42         p = np.exp(-np.abs(h - y)) # probability for next char
43         loss += -np.log(p) # softmax loss
44         # backward pass: compute gradients going backwards
45         d_h0, d_h1, d_h2, d_h3 = np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE)
46         d_h0, d_h1, d_h2, d_h3 = np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE)
47         d_hprev = np.zeros(HIDDEN_SIZE)
48         for j in reversed(range(i+1, len(inputs))):
49             dy = np.zeros(1)
50             dy[targets[j]] = 1 # backprop into y
51             d_h1 = np.dot(dy, W1[j])
52             d_h2 = np.dot(dy, W2[j])
53             d_h3 = (1 - h[j]) * d_h1 + h[j] * d_h2 # backprop through tanh nonlinearity
54             d_h0 = d_h3
55             d_hprev = np.dot(d_h0, W1[j])
56             d_loss = np.dot(d_h0, W0[j]) + d_h2
57             d_loss = np.dot(d_loss, 1) # softmax loss
58             d_loss = np.dot(d_loss, 1) # softmax loss
59             # For d_loss in [d_h0, d_h1, d_h2, d_h3, d_h4]
60             d_loss = np.dot(d_loss, 1) # softmax loss
61             return loss, d_loss, d_h0, d_h1, d_h2, d_h3, d_h4
62
63 def lstmCell(inputs, targets, hprev):
64     """
65     inputs, targets are both list of integers.
66     hprev is the array of hidden states
67     returns the loss, gradients on model parameters, and last hidden state
68     """
69     W, U, W0, W1, W2, W3, W4 = W0, W1, W2, W3, W4, W0, W1, W2, W3, W4
70     h0 = np.zeros(HIDDEN_SIZE)
71     loss = 0
72     for i in range(len(inputs)):
73         x = inputs[i]
74         h = np.tanh(W0[x] + W1[h0] + W2[hprev]) * \
75             np.sigmoid(W3[x] + W4[h0] + W1[hprev]) # hidden state
76         y = inputs[i]
77         p = np.exp(-np.abs(h - y)) # probability for next char
78         loss += -np.log(p) # softmax loss
79         # backward pass: compute gradients going backwards
80         d_h0, d_h1, d_h2, d_h3 = np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE)
81         d_h0, d_h1, d_h2, d_h3 = np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE)
82         d_hprev = np.zeros(HIDDEN_SIZE)
83         for j in reversed(range(i+1, len(inputs))):
84             dy = np.zeros(1)
85             dy[targets[j]] = 1 # backprop into y
86             d_h1 = np.dot(dy, W1[j])
87             d_h2 = np.dot(dy, W2[j])
88             d_h3 = (1 - h[j]) * d_h1 + h[j] * d_h2 # backprop through tanh nonlinearity
89             d_h0 = d_h3
90             d_hprev = np.dot(d_h0, W1[j])
91             d_loss = np.dot(d_h0, W0[j]) + d_h2
92             d_loss = np.dot(d_loss, 1) # softmax loss
93             d_loss = np.dot(d_loss, 1) # softmax loss
94             # For d_loss in [d_h0, d_h1, d_h2, d_h3, d_h4]
95             d_loss = np.dot(d_loss, 1) # softmax loss
96             return loss, d_loss, d_h0, d_h1, d_h2, d_h3, d_h4
97
98 def lstmCell(inputs, targets, hprev):
99     """
100    inputs, targets are both list of integers.
101    hprev is the array of hidden states
102    returns the loss, gradients on model parameters, and last hidden state
103    """
104    W, U, W0, W1, W2, W3, W4 = W0, W1, W2, W3, W4, W0, W1, W2, W3, W4
105    h0 = np.zeros(HIDDEN_SIZE)
106    loss = 0
107    for i in range(len(inputs)):
108        x = inputs[i]
109        h = np.tanh(W0[x] + W1[h0] + W2[hprev]) * \
110            np.sigmoid(W3[x] + W4[h0] + W1[hprev]) # hidden state
111        y = inputs[i]
112        p = np.exp(-np.abs(h - y)) # probability for next char
113        loss += -np.log(p) # softmax loss
114        # backward pass: compute gradients going backwards
115        d_h0, d_h1, d_h2, d_h3 = np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE)
116        d_h0, d_h1, d_h2, d_h3 = np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE), np.zeros(HIDDEN_SIZE)
117        d_hprev = np.zeros(HIDDEN_SIZE)
118        for j in reversed(range(i+1, len(inputs))):
119            dy = np.zeros(1)
120            dy[targets[j]] = 1 # backprop into y
121            d_h1 = np.dot(dy, W1[j])
122            d_h2 = np.dot(dy, W2[j])
123            d_h3 = (1 - h[j]) * d_h1 + h[j] * d_h2 # backprop through tanh nonlinearity
124            d_h0 = d_h3
125            d_hprev = np.dot(d_h0, W1[j])
126            d_loss = np.dot(d_h0, W0[j]) + d_h2
127            d_loss = np.dot(d_loss, 1) # softmax loss
128            d_loss = np.dot(d_loss, 1) # softmax loss
129            # For d_loss in [d_h0, d_h1, d_h2, d_h3, d_h4]
130            d_loss = np.dot(d_loss, 1) # softmax loss
131            return loss, d_loss, d_h0, d_h1, d_h2, d_h3, d_h4
132
133 """
134 """
135 """
136 """
137 """
138 """
139 """
140 """
141 """
142 """
143 """
144 """
145 """
146 """
147 """
148 """
149 """
150 """
151 """
152 """
153 """
154 """
155 """
156 """
157 """
158 """
159 """
160 """
161 """
162 """
163 """
164 """
165 """
166 """
167 """
168 """
169 """
170 """
171 """
172 """
173 """
174 """
175 """
176 """
177 """
178 """
179 """
180 """
181 """
182 """
183 """
184 """
185 """
186 """
187 """
188 """
189 """
190 """
191 """
192 """
193 """
194 """
195 """
196 """
197 """
198 """
199 """
200 """
201 """
202 """
203 """
204 """
205 """
206 """
207 """
208 """
209 """
210 """
211 """
212 """
213 """
214 """
215 """
216 """
217 """
218 """
219 """
220 """
221 """
222 """
223 """
224 """
225 """
226 """
227 """
228 """
229 """
230 """
231 """
232 """
233 """
234 """
235 """
236 """
237 """
238 """
239 """
240 """
241 """
242 """
243 """
244 """
245 """
246 """
247 """
248 """
249 """
250 """
251 """
252 """
253 """
254 """
255 """
256 """
257 """
258 """
259 """
260 """
261 """
262 """
263 """
264 """
265 """
266 """
267 """
268 """
269 """
270 """
271 """
272 """
273 """
274 """
275 """
276 """
277 """
278 """
279 """
280 """
281 """
282 """
283 """
284 """
285 """
286 """
287 """
288 """
289 """
290 """
291 """
292 """
293 """
294 """
295 """
296 """
297 """
298 """
299 """
300 """
301 """
302 """
303 """
304 """
305 """
306 """
307 """
308 """
309 """
310 """
311 """
312 """
313 """
314 """
315 """
316 """
317 """
318 """
319 """
320 """
321 """
322 """
323 """
324 """
325 """
326 """
327 """
328 """
329 """
330 """
331 """
332 """
333 """
334 """
335 """
336 """
337 """
338 """
339 """
340 """
341 """
342 """
343 """
344 """
345 """
346 """
347 """
348 """
349 """
350 """
351 """
352 """
353 """
354 """
355 """
356 """
357 """
358 """
359 """
360 """
361 """
362 """
363 """
364 """
365 """
366 """
367 """
368 """
369 """
370 """
371 """
372 """
373 """
374 """
375 """
376 """
377 """
378 """
379 """
380 """
381 """
382 """
383 """
384 """
385 """
386 """
387 """
388 """
389 """
390 """
391 """
392 """
393 """
394 """
395 """
396 """
397 """
398 """
399 """
400 """
401 """
402 """
403 """
404 """
405 """
406 """
407 """
408 """
409 """
410 """
411 """
412 """
413 """
414 """
415 """
416 """
417 """
418 """
419 """
420 """
421 """
422 """
423 """
424 """
425 """
426 """
427 """
428 """
429 """
430 """
431 """
432 """
433 """
434 """
435 """
436 """
437 """
438 """
439 """
440 """
441 """
442 """
443 """
444 """
445 """
446 """
447 """
448 """
449 """
450 """
451 """
452 """
453 """
454 """
455 """
456 """
457 """
458 """
459 """
460 """
461 """
462 """
463 """
464 """
465 """
466 """
467 """
468 """
469 """
470 """
471 """
472 """
473 """
474 """
475 """
476 """
477 """
478 """
479 """
480 """
481 """
482 """
483 """
484 """
485 """
486 """
487 """
488 """
489 """
490 """
491 """
492 """
493 """
494 """
495 """
496 """
497 """
498 """
499 """
500 """
501 """
502 """
503 """
504 """
505 """
506 """
507 """
508 """
509 """
510 """
511 """
512 """
513 """
514 """
515 """
516 """
517 """
518 """
519 """
520 """
521 """
522 """
523 """
524 """
525 """
526 """
527 """
528 """
529 """
530 """
531 """
532 """
533 """
534 """
535 """
536 """
537 """
538 """
539 """
540 """
541 """
542 """
543 """
544 """
545 """
546 """
547 """
548 """
549 """
550 """
551 """
552 """
553 """
554 """
555 """
556 """
557 """
558 """
559 """
560 """
561 """
562 """
563 """
564 """
565 """
566 """
567 """
568 """
569 """
570 """
571 """
572 """
573 """
574 """
575 """
576 """
577 """
578 """
579 """
580 """
581 """
582 """
583 """
584 """
585 """
586 """
587 """
588 """
589 """
590 """
591 """
592 """
593 """
594 """
595 """
596 """
597 """
598 """
599 """
600 """
601 """
602 """
603 """
604 """
605 """
606 """
607 """
608 """
609 """
610 """
611 """
612 """
613 """
614 """
615 """
616 """
617 """
618 """
619 """
620 """
621 """
622 """
623 """
624 """
625 """
626 """
627 """
628 """
629 """
630 """
631 """
632 """
633 """
634 """
635 """
636 """
637 """
638 """
639 """
640 """
641 """
642 """
643 """
644 """
645 """
646 """
647 """
648 """
649 """
650 """
651 """
652 """
653 """
654 """
655 """
656 """
657 """
658 """
659 """
660 """
661 """
662 """
663 """
664 """
665 """
666 """
667 """
668 """
669 """
670 """
671 """
672 """
673 """
674 """
675 """
676 """
677 """
678 """
679 """
680 """
681 """
682 """
683 """
684 """
685 """
686 """
687 """
688 """
689 """
690 """
691 """
692 """
693 """
694 """
695 """
696 """
697 """
698 """
699 """
700 """
701 """
702 """
703 """
704 """
705 """
706 """
707 """
708 """
709 """
710 """
711 """
712 """
713 """
714 """
715 """
716 """
717 """
718 """
719 """
720 """
721 """
722 """
723 """
724 """
725 """
726 """
727 """
728 """
729 """
730 """
731 """
732 """
733 """
734 """
735 """
736 """
737 """
738 """
739 """
740 """
741 """
742 """
743 """
744 """
745 """
746 """
747 """
748 """
749 """
750 """
751 """
752 """
753 """
754 """
755 """
756 """
757 """
758 """
759 """
760 """
761 """
762 """
763 """
764 """
765 """
766 """
767 """
768 """
769 """
770 """
771 """
772 """
773 """
774 """
775 """
776 """
777 """
778 """
779 """
780 """
781 """
782 """
783 """
784 """
785 """
786 """
787 """
788 """
789 """
790 """
791 """
792 """
793 """
794 """
795 """
796 """
797 """
798 """
799 """
800 """
801 """
802 """
803 """
804 """
805 """
806 """
807 """
808 """
809 """
810 """
811 """
812 """
813 """
814 """
815 """
816 """
817 """
818 """
819 """
820 """
821 """
822 """
823 """
824 """
825 """
826 """
827 """
828 """
829 """
830 """
831 """
832 """
833 """
834 """
835 """
836 """
837 """
838 """
839 """
840 """
841 """
842 """
843 """
844 """
845 """
846 """
847 """
848 """
849 """
850 """
851 """
852 """
853 """
854 """
855 """
856 """
857 """
858 """
859 """
860 """
861 """
862 """
863 """
864 """
865 """
866 """
867 """
868 """
869 """
870 """
871 """
872 """
873 """
874 """
875 """
876 """
877 """
878 """
879 """
880 """
881 """
882 """
883 """
884 """
885 """
886 """
887 """
888 """
889 """
890 """
891 """
892 """
893 """
894 """
895 """
896 """
897 """
898 """
899 """
900 """
901 """
902 """
903 """
904 """
905 """
906 """
907 """
908 """
909 """
910 """
911 """
912 """
913 """
914 """
915 """
916 """
917 """
918 """
919 """
920 """
921 """
922 """
923 """
924 """
925 """
926 """
927 """
928 """
929 """
930 """
931 """
932 """
933 """
934 """
935 """
936 """
937 """
938 """
939 """
940 """
941 """
942 """
943 """
944 """
945 """
946 """
947 """
948 """
949 """
950 """
951 """
952 """
953 """
954 """
955 """
956 """
957 """
958 """
959 """
960 """
961 """
962 """
963 """
964 """
965 """
966 """
967 """
968 """
969 """
970 """
971 """
972 """
973 """
974 """
975 """
976 """
977 """
978 """
979 """
980 """
981 """
982 """
983 """
984 """
985 """
986 """
987 """
988 """
989 """
990 """
991 """
992 """
993 """
994 """
995 """
996 """
997 """
998 """
999 """
1000 """

```

(<https://gist.github.com/karpathy/d4dee566867f8291f086>)

min-char-rnn.py gist

```

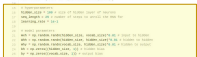
1 # Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
2 BSD License
3
4 import numpy as np
5
6 # data I/O
7 data = open('input.txt', 'r').read() # should be simple plain text file
8 chars = list(set(data))
9 data_size, vocab_size = len(data), len(chars)
10 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
11 char_to_ix = { ch:i for i,ch in enumerate(chars) }
12 ix_to_char = { i:ch for i,ch in enumerate(chars) }
13
14 # ...
15
16 # ...
17
18 # ...
19
20 # ...
21
22 # ...
23
24 # ...
25
26 # ...
27
28 # ...
29
30 # ...
31
32 # ...
33
34 # ...
35
36 # ...
37
38 # ...
39
40 # ...
41
42 # ...
43
44 # ...
45
46 # ...
47
48 # ...
49
50 # ...
51
52 # ...
53
54 # ...
55
56 # ...
57
58 # ...
59
60 # ...
61
62 # ...
63
64 # ...
65
66 # ...
67
68 # ...
69
70 # ...
71
72 # ...
73
74 # ...
75
76 # ...
77
78 # ...
79
80 # ...
81
82 # ...
83
84 # ...
85
86 # ...
87
88 # ...
89
90 # ...
91
92 # ...
93
94 # ...
95
96 # ...
97
98 # ...
99
100 # ...
101
102 # ...
103
104 # ...
105
106 # ...
107
108 # ...
109
110 # ...
111
112 # ...
113
114 # ...
115
116 # ...
117
118 # ...
119
120 # ...
121
122 # ...
123
124 # ...
125
126 # ...
127
128 # ...
129
130 # ...
131
132 # ...
133
134 # ...
135
136 # ...
137
138 # ...
139
140 # ...
141
142 # ...
143
144 # ...
145
146 # ...
147
148 # ...
149
150 # ...
151
152 # ...
153
154 # ...
155
156 # ...
157
158 # ...
159
160 # ...
161
162 # ...
163
164 # ...
165
166 # ...
167
168 # ...
169
170 # ...
171
172 # ...
173
174 # ...
175
176 # ...
177
178 # ...
179
180 # ...
181
182 # ...
183
184 # ...
185
186 # ...
187
188 # ...
189
190 # ...
191
192 # ...
193
194 # ...
195
196 # ...
197
198 # ...
199
200 # ...
201
202 # ...
203
204 # ...
205
206 # ...
207
208 # ...
209
210 # ...
211
212 # ...
213
214 # ...
215
216 # ...
217
218 # ...
219
220 # ...
221
222 # ...
223
224 # ...
225
226 # ...
227
228 # ...
229
230 # ...
231
232 # ...
233
234 # ...
235
236 # ...
237
238 # ...
239
240 # ...
241
242 # ...
243
244 # ...
245
246 # ...
247
248 # ...
249
250 # ...
251
252 # ...
253
254 # ...
255
256 # ...
257
258 # ...
259
260 # ...
261
262 # ...
263
264 # ...
265
266 # ...
267
268 # ...
269
270 # ...
271
272 # ...
273
274 # ...
275
276 # ...
277
278 # ...
279
280 # ...
281
282 # ...
283
284 # ...
285
286 # ...
287
288 # ...
289
290 # ...
291
292 # ...
293
294 # ...
295
296 # ...
297
298 # ...
299
300 # ...
301
302 # ...
303
304 # ...
305
306 # ...
307
308 # ...
309
310 # ...
311
312 # ...
313
314 # ...
315
316 # ...
317
318 # ...
319
320 # ...
321
322 # ...
323
324 # ...
325
326 # ...
327
328 # ...
329
330 # ...
331
332 # ...
333
334 # ...
335
336 # ...
337
338 # ...
339
340 # ...
341
342 # ...
343
344 # ...
345
346 # ...
347
348 # ...
349
350 # ...
351
352 # ...
353
354 # ...
355
356 # ...
357
358 # ...
359
360 # ...
361
362 # ...
363
364 # ...
365
366 # ...
367
368 # ...
369
370 # ...
371
372 # ...
373
374 # ...
375
376 # ...
377
378 # ...
379
380 # ...
381
382 # ...
383
384 # ...
385
386 # ...
387
388 # ...
389
390 # ...
391
392 # ...
393
394 # ...
395
396 # ...
397
398 # ...
399
400 # ...
401
402 # ...
403
404 # ...
405
406 # ...
407
408 # ...
409
410 # ...
411
412 # ...
413
414 # ...
415
416 # ...
417
418 # ...
419
420 # ...
421
422 # ...
423
424 # ...
425
426 # ...
427
428 # ...
429
430 # ...
431
432 # ...
433
434 # ...
435
436 # ...
437
438 # ...
439
440 # ...
441
442 # ...
443
444 # ...
445
446 # ...
447
448 # ...
449
450 # ...
451
452 # ...
453
454 # ...
455
456 # ...
457
458 # ...
459
460 # ...
461
462 # ...
463
464 # ...
465
466 # ...
467
468 # ...
469
470 # ...
471
472 # ...
473
474 # ...
475
476 # ...
477
478 # ...
479
480 # ...
481
482 # ...
483
484 # ...
485
486 # ...
487
488 # ...
489
490 # ...
491
492 # ...
493
494 # ...
495
496 # ...
497
498 # ...
499
500 # ...
501
502 # ...
503
504 # ...
505
506 # ...
507
508 # ...
509
510 # ...
511
512 # ...
513
514 # ...
515
516 # ...
517
518 # ...
519
520 # ...
521
522 # ...
523
524 # ...
525
526 # ...
527
528 # ...
529
530 # ...
531
532 # ...
533
534 # ...
535
536 # ...
537
538 # ...
539
540 # ...
541
542 # ...
543
544 # ...
545
546 # ...
547
548 # ...
549
550 # ...
551
552 # ...
553
554 # ...
555
556 # ...
557
558 # ...
559
560 # ...
561
562 # ...
563
564 # ...
565
566 # ...
567
568 # ...
569
570 # ...
571
572 # ...
573
574 # ...
575
576 # ...
577
578 # ...
579
580 # ...
581
582 # ...
583
584 # ...
585
586 # ...
587
588 # ...
589
590 # ...
591
592 # ...
593
594 # ...
595
596 # ...
597
598 # ...
599
600 # ...
601
602 # ...
603
604 # ...
605
606 # ...
607
608 # ...
609
610 # ...
611
612 # ...
613
614 # ...
615
616 # ...
617
618 # ...
619
620 # ...
621
622 # ...
623
624 # ...
625
626 # ...
627
628 # ...
629
630 # ...
631
632 # ...
633
634 # ...
635
636 # ...
637
638 # ...
639
640 # ...
641
642 # ...
643
644 # ...
645
646 # ...
647
648 # ...
649
650 # ...
651
652 # ...
653
654 # ...
655
656 # ...
657
658 # ...
659
660 # ...
661
662 # ...
663
664 # ...
665
666 # ...
667
668 # ...
669
670 # ...
671
672 # ...
673
674 # ...
675
676 # ...
677
678 # ...
679
680 # ...
681
682 # ...
683
684 # ...
685
686 # ...
687
688 # ...
689
690 # ...
691
692 # ...
693
694 # ...
695
696 # ...
697
698 # ...
699
700 # ...
701
702 # ...
703
704 # ...
705
706 # ...
707
708 # ...
709
710 # ...
711
712 # ...
713
714 # ...
715
716 # ...
717
718 # ...
719
720 # ...
721
722 # ...
723
724 # ...
725
726 # ...
727
728 # ...
729
730 # ...
731
732 # ...
733
734 # ...
735
736 # ...
737
738 # ...
739
740 # ...
741
742 # ...
743
744 # ...
745
746 # ...
747
748 # ...
749
750 # ...
751
752 # ...
753
754 # ...
755
756 # ...
757
758 # ...
759
760 # ...
761
762 # ...
763
764 # ...
765
766 # ...
767
768 # ...
769
770 # ...
771
772 # ...
773
774 # ...
775
776 # ...
777
778 # ...
779
780 # ...
781
782 # ...
783
784 # ...
785
786 # ...
787
788 # ...
789
790 # ...
791
792 # ...
793
794 # ...
795
796 # ...
797
798 # ...
799
800 # ...
801
802 # ...
803
804 # ...
805
806 # ...
807
808 # ...
809
810 # ...
811
812 # ...
813
814 # ...
815
816 # ...
817
818 # ...
819
820 # ...
821
822 # ...
823
824 # ...
825
826 # ...
827
828 # ...
829
830 # ...
831
832 # ...
833
834 # ...
835
836 # ...
837
838 # ...
839
840 # ...
841
842 # ...
843
844 # ...
845
846 # ...
847
848 # ...
849
850 # ...
851
852 # ...
853
854 # ...
855
856 # ...
857
858 # ...
859
860 # ...
861
862 # ...
863
864 # ...
865
866 # ...
867
868 # ...
869
870 # ...
871
872 # ...
873
874 # ...
875
876 # ...
877
878 # ...
879
880 # ...
881
882 # ...
883
884 # ...
885
886 # ...
887
888 # ...
889
890 # ...
891
892 # ...
893
894 # ...
895
896 # ...
897
898 # ...
899
900 # ...
901
902 # ...
903
904 # ...
905
906 # ...
907
908 # ...
909
910 # ...
911
912 # ...
913
914 # ...
915
916 # ...
917
918 # ...
919
920 # ...
921
922 # ...
923
924 # ...
925
926 # ...
927
928 # ...
929
930 # ...
931
932 # ...
933
934 # ...
935
936 # ...
937
938 # ...
939
940 # ...
941
942 # ...
943
944 # ...
945
946 # ...
947
948 # ...
949
950 # ...
951
952 # ...
953
954 # ...
955
956 # ...
957
958 # ...
959
960 # ...
961
962 # ...
963
964 # ...
965
966 # ...
967
968 # ...
969
970 # ...
971
972 # ...
973
974 # ...
975
976 # ...
977
978 # ...
979
980 # ...
981
982 # ...
983
984 # ...
985
986 # ...
987
988 # ...
989
990 # ...
991
992 # ...
993
994 # ...
995
996 # ...
997
998 # ...
999
1000 # ...
    
```

Data I/O

```

1 """
2 Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3 BSD License
4 """
5
6 import numpy as np
7
8 # data I/O
9 data = open('input.txt', 'r').read() # should be simple plain text file
10 chars = list(set(data))
11 data_size, vocab_size = len(data), len(chars)
12 print 'data has %d characters, %d unique.' % (data_size, vocab_size)
13 char_to_ix = { ch:i for i,ch in enumerate(chars) }
14 ix_to_char = { i:ch for i,ch in enumerate(chars) }
    
```

min-char-rnn.py gist



```

1 #
2 #
3 #
4 #
5 #
6 #
7 #
8 #
9 #
10 #
11 #
12 #
13 #
14 #
15 #
16 #
17 #
18 #
19 #
20 #
21 #
22 #
23 #
24 #
25 #
26 #
27 #
28 #
29 #
30 #
31 #
32 #
33 #
34 #
35 #
36 #
37 #
38 #
39 #
40 #
41 #
42 #
43 #
44 #
45 #
46 #
47 #
48 #
49 #
50 #
51 #
52 #
53 #
54 #
55 #
56 #
57 #
58 #
59 #
60 #
61 #
62 #
63 #
64 #
65 #
66 #
67 #
68 #
69 #
70 #
71 #
72 #
73 #
74 #
75 #
76 #
77 #
78 #
79 #
80 #
81 #
82 #
83 #
84 #
85 #
86 #
87 #
88 #
89 #
90 #
91 #
92 #
93 #
94 #
95 #
96 #
97 #
98 #
99 #
100 #
101 #
102 #
103 #
104 #
105 #
106 #
107 #
108 #
109 #
110 #
111 #
112 #
113 #
114 #
115 #
116 #
117 #
118 #
119 #
120 #
121 #
122 #
123 #
124 #
125 #
126 #
127 #
128 #
129 #
130 #
131 #
132 #
133 #
134 #
135 #
136 #
137 #
138 #
139 #
140 #
141 #
142 #
143 #
144 #
145 #
146 #
147 #
148 #
149 #
150 #
151 #
152 #
153 #
154 #
155 #
156 #
157 #
158 #
159 #
160 #
161 #
162 #
163 #
164 #
165 #
166 #
167 #
168 #
169 #
170 #
171 #
172 #
173 #
174 #
175 #
176 #
177 #
178 #
179 #
180 #
181 #
182 #
183 #
184 #
185 #
186 #
187 #
188 #
189 #
190 #
191 #
192 #
193 #
194 #
195 #
196 #
197 #
198 #
199 #
200 #
201 #
202 #
203 #
204 #
205 #
206 #
207 #
208 #
209 #
210 #
211 #
212 #
213 #
214 #
215 #
216 #
217 #
218 #
219 #
220 #
221 #
222 #
223 #
224 #
225 #
226 #
227 #
228 #
229 #
230 #
231 #
232 #
233 #
234 #
235 #
236 #
237 #
238 #
239 #
240 #
241 #
242 #
243 #
244 #
245 #
246 #
247 #
248 #
249 #
250 #
251 #
252 #
253 #
254 #
255 #
256 #
257 #
258 #
259 #
260 #
261 #
262 #
263 #
264 #
265 #
266 #
267 #
268 #
269 #
270 #
271 #
272 #
273 #
274 #
275 #
276 #
277 #
278 #
279 #
280 #
281 #
282 #
283 #
284 #
285 #
286 #
287 #
288 #
289 #
290 #
291 #
292 #
293 #
294 #
295 #
296 #
297 #
298 #
299 #
300 #
301 #
302 #
303 #
304 #
305 #
306 #
307 #
308 #
309 #
310 #
311 #
312 #
313 #
314 #
315 #
316 #
317 #
318 #
319 #
320 #
321 #
322 #
323 #
324 #
325 #
326 #
327 #
328 #
329 #
330 #
331 #
332 #
333 #
334 #
335 #
336 #
337 #
338 #
339 #
340 #
341 #
342 #
343 #
344 #
345 #
346 #
347 #
348 #
349 #
350 #
351 #
352 #
353 #
354 #
355 #
356 #
357 #
358 #
359 #
360 #
361 #
362 #
363 #
364 #
365 #
366 #
367 #
368 #
369 #
370 #
371 #
372 #
373 #
374 #
375 #
376 #
377 #
378 #
379 #
380 #
381 #
382 #
383 #
384 #
385 #
386 #
387 #
388 #
389 #
390 #
391 #
392 #
393 #
394 #
395 #
396 #
397 #
398 #
399 #
400 #
401 #
402 #
403 #
404 #
405 #
406 #
407 #
408 #
409 #
410 #
411 #
412 #
413 #
414 #
415 #
416 #
417 #
418 #
419 #
420 #
421 #
422 #
423 #
424 #
425 #
426 #
427 #
428 #
429 #
430 #
431 #
432 #
433 #
434 #
435 #
436 #
437 #
438 #
439 #
440 #
441 #
442 #
443 #
444 #
445 #
446 #
447 #
448 #
449 #
450 #
451 #
452 #
453 #
454 #
455 #
456 #
457 #
458 #
459 #
460 #
461 #
462 #
463 #
464 #
465 #
466 #
467 #
468 #
469 #
470 #
471 #
472 #
473 #
474 #
475 #
476 #
477 #
478 #
479 #
480 #
481 #
482 #
483 #
484 #
485 #
486 #
487 #
488 #
489 #
490 #
491 #
492 #
493 #
494 #
495 #
496 #
497 #
498 #
499 #
500 #
501 #
502 #
503 #
504 #
505 #
506 #
507 #
508 #
509 #
510 #
511 #
512 #
513 #
514 #
515 #
516 #
517 #
518 #
519 #
520 #
521 #
522 #
523 #
524 #
525 #
526 #
527 #
528 #
529 #
530 #
531 #
532 #
533 #
534 #
535 #
536 #
537 #
538 #
539 #
540 #
541 #
542 #
543 #
544 #
545 #
546 #
547 #
548 #
549 #
550 #
551 #
552 #
553 #
554 #
555 #
556 #
557 #
558 #
559 #
560 #
561 #
562 #
563 #
564 #
565 #
566 #
567 #
568 #
569 #
570 #
571 #
572 #
573 #
574 #
575 #
576 #
577 #
578 #
579 #
580 #
581 #
582 #
583 #
584 #
585 #
586 #
587 #
588 #
589 #
590 #
591 #
592 #
593 #
594 #
595 #
596 #
597 #
598 #
599 #
600 #
601 #
602 #
603 #
604 #
605 #
606 #
607 #
608 #
609 #
610 #
611 #
612 #
613 #
614 #
615 #
616 #
617 #
618 #
619 #
620 #
621 #
622 #
623 #
624 #
625 #
626 #
627 #
628 #
629 #
630 #
631 #
632 #
633 #
634 #
635 #
636 #
637 #
638 #
639 #
640 #
641 #
642 #
643 #
644 #
645 #
646 #
647 #
648 #
649 #
650 #
651 #
652 #
653 #
654 #
655 #
656 #
657 #
658 #
659 #
660 #
661 #
662 #
663 #
664 #
665 #
666 #
667 #
668 #
669 #
670 #
671 #
672 #
673 #
674 #
675 #
676 #
677 #
678 #
679 #
680 #
681 #
682 #
683 #
684 #
685 #
686 #
687 #
688 #
689 #
690 #
691 #
692 #
693 #
694 #
695 #
696 #
697 #
698 #
699 #
700 #
701 #
702 #
703 #
704 #
705 #
706 #
707 #
708 #
709 #
710 #
711 #
712 #
713 #
714 #
715 #
716 #
717 #
718 #
719 #
720 #
721 #
722 #
723 #
724 #
725 #
726 #
727 #
728 #
729 #
730 #
731 #
732 #
733 #
734 #
735 #
736 #
737 #
738 #
739 #
740 #
741 #
742 #
743 #
744 #
745 #
746 #
747 #
748 #
749 #
750 #
751 #
752 #
753 #
754 #
755 #
756 #
757 #
758 #
759 #
760 #
761 #
762 #
763 #
764 #
765 #
766 #
767 #
768 #
769 #
770 #
771 #
772 #
773 #
774 #
775 #
776 #
777 #
778 #
779 #
780 #
781 #
782 #
783 #
784 #
785 #
786 #
787 #
788 #
789 #
790 #
791 #
792 #
793 #
794 #
795 #
796 #
797 #
798 #
799 #
800 #
801 #
802 #
803 #
804 #
805 #
806 #
807 #
808 #
809 #
810 #
811 #
812 #
813 #
814 #
815 #
816 #
817 #
818 #
819 #
820 #
821 #
822 #
823 #
824 #
825 #
826 #
827 #
828 #
829 #
830 #
831 #
832 #
833 #
834 #
835 #
836 #
837 #
838 #
839 #
840 #
841 #
842 #
843 #
844 #
845 #
846 #
847 #
848 #
849 #
850 #
851 #
852 #
853 #
854 #
855 #
856 #
857 #
858 #
859 #
860 #
861 #
862 #
863 #
864 #
865 #
866 #
867 #
868 #
869 #
870 #
871 #
872 #
873 #
874 #
875 #
876 #
877 #
878 #
879 #
880 #
881 #
882 #
883 #
884 #
885 #
886 #
887 #
888 #
889 #
890 #
891 #
892 #
893 #
894 #
895 #
896 #
897 #
898 #
899 #
900 #
901 #
902 #
903 #
904 #
905 #
906 #
907 #
908 #
909 #
910 #
911 #
912 #
913 #
914 #
915 #
916 #
917 #
918 #
919 #
920 #
921 #
922 #
923 #
924 #
925 #
926 #
927 #
928 #
929 #
930 #
931 #
932 #
933 #
934 #
935 #
936 #
937 #
938 #
939 #
940 #
941 #
942 #
943 #
944 #
945 #
946 #
947 #
948 #
949 #
950 #
951 #
952 #
953 #
954 #
955 #
956 #
957 #
958 #
959 #
960 #
961 #
962 #
963 #
964 #
965 #
966 #
967 #
968 #
969 #
970 #
971 #
972 #
973 #
974 #
975 #
976 #
977 #
978 #
979 #
980 #
981 #
982 #
983 #
984 #
985 #
986 #
987 #
988 #
989 #
990 #
991 #
992 #
993 #
994 #
995 #
996 #
997 #
998 #
999 #
1000 #

```

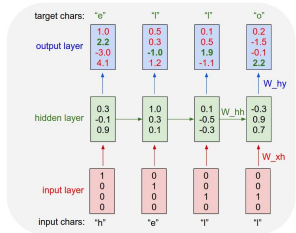
Initializations

```

15 # hyperparameters
16 hidden_size = 100 # size of hidden layer of neurons
17 seq_length = 25 # number of steps to unroll the RNN for
18 learning_rate = 1e-1
19
20 # model parameters
21 Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
22 Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
23 Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
24 bh = np.zeros((hidden_size, 1)) # hidden bias
25 by = np.zeros((vocab_size, 1)) # output bias

```

recall:



min-char-rnn.py gist

```

1 # -*- coding: utf-8 -*-
2 """
3 Example: character-level language model. Modified by Andrej Kardecy (2016/04/24)
4 """
5 import sys
6 import random
7
8 # Data loader
9 def load_data(text, vocab_size):
10     """Load data and build vocabulary"""
11     # Build vocabulary
12     vocab = {}
13     for char in text:
14         vocab[char] = vocab.get(char, 0) + 1
15     vocab_size = len(vocab)
16     # Build character to index mapping
17     char_to_ix = {}
18     for i, char in enumerate(vocab):
19         char_to_ix[char] = i
20     # Build index to character mapping
21     ix_to_char = {}
22     for i, char in enumerate(vocab):
23         ix_to_char[i] = char
24     return vocab_size, char_to_ix, ix_to_char
25
26 # Model
27 def build_model(vocab_size, hidden_size):
28     """Build model"""
29     # Embedding layer
30     W_emb = np.random.randn(vocab_size, hidden_size)
31     b_emb = np.zeros(hidden_size)
32     # Hidden layer
33     W_h = np.random.randn(hidden_size, hidden_size)
34     b_h = np.zeros(hidden_size)
35     # Output layer
36     W_o = np.random.randn(hidden_size, vocab_size)
37     b_o = np.zeros(vocab_size)
38     # Learning rates
39     lr_emb = 0.1
40     lr_h = 0.1
41     lr_o = 0.1
42     # Parameters
43     W = {'emb': W_emb, 'h': W_h, 'o': W_o}
44     b = {'emb': b_emb, 'h': b_h, 'o': b_o}
45     lr = {'emb': lr_emb, 'h': lr_h, 'o': lr_o}
46     return W, b, lr
47
48 # Training
49 def train_model(vocab_size, hidden_size, num_epochs):
50     """Train model"""
51     # Load data
52     vocab_size, char_to_ix, ix_to_char = load_data(text, vocab_size)
53     # Build model
54     W, b, lr = build_model(vocab_size, hidden_size)
55     # Training loop
56     for epoch in range(1, num_epochs + 1):
57         # Shuffle data
58         indices = np.arange(len(text))
59         random.shuffle(indices)
60         # Iterate over data
61         for i in range(0, len(text), seq_length):
62             # Get inputs and targets
63             inputs = [char_to_ix[char] for char in text[i:i+seq_length]]
64             targets = [char_to_ix[char] for char in text[i+1:i+seq_length+1]]
65             # Sample from model
66             sample_ix = sample(W, b, lr, inputs, hidden_size, vocab_size)
67             # Compute loss
68             loss = lossFun(inputs, targets, W, b, lr)
69             # Compute gradients
70             grads = gradFun(loss, W, b, lr, inputs, hidden_size, vocab_size)
71             # Update parameters
72             W, b, lr = update(W, b, lr, grads, lr)
73         # Print progress
74         print('Epoch %d, loss: %f' % (epoch, loss))
75     return W, b, lr
76
77 # Evaluation
78 def evaluate_model(W, b, lr, text):
79     """Evaluate model"""
80     # Load data
81     vocab_size, char_to_ix, ix_to_char = load_data(text, vocab_size)
82     # Build model
83     W, b, lr = build_model(vocab_size, hidden_size)
84     # Evaluate
85     indices = np.arange(len(text))
86     random.shuffle(indices)
87     for i in range(0, len(text), seq_length):
88         # Get inputs and targets
89         inputs = [char_to_ix[char] for char in text[i:i+seq_length]]
90         targets = [char_to_ix[char] for char in text[i+1:i+seq_length+1]]
91         # Sample from model
92         sample_ix = sample(W, b, lr, inputs, hidden_size, vocab_size)
93         # Compute loss
94         loss = lossFun(inputs, targets, W, b, lr)
95         # Compute gradients
96         grads = gradFun(loss, W, b, lr, inputs, hidden_size, vocab_size)
97         # Update parameters
98         W, b, lr = update(W, b, lr, grads, lr)
99     return W, b, lr
100
101 # Main
102 if __name__ == '__main__':
103     # Load data
104     vocab_size, char_to_ix, ix_to_char = load_data(text, vocab_size)
105     # Build model
106     W, b, lr = build_model(vocab_size, hidden_size)
107     # Train model
108     W, b, lr = train_model(vocab_size, hidden_size, num_epochs)
109     # Evaluate model
110     W, b, lr = evaluate_model(W, b, lr, text)
111     # Print progress
112     print('Final loss: %f' % loss)
113

```



Main loop

```

81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '----\n %s \n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101     smooth_loss = smooth_loss * 0.999 + loss * 0.001
102     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104     # perform parameter update with Adagrad
105     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                 [dWxh, dWhh, dWhy, dbh, dby],
107                                 [mWxh, mWhh, mWhy, mbh, mby]):
108         mem += dparam * dparam
109         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter

```

min-char-rnn.py gist

```

1 # -*- coding: utf-8 -*-
2 """
3 Example: character-level language model. Modified by Andrej Kardecy (2016/04/24)
4 See README
5 """
6 import numpy as np
7
8 # Model size
9 data = open('data.txt', 'r').read() # read the whole data file
10 nchars = len(set(data)) # number of unique characters
11 data = list(data) # list of characters
12 char_to_ix = {char: ix for ix, char in enumerate(data)} # char to int mapping
13 ix_to_char = {ix: char for ix, char in enumerate(data)} # int to char mapping
14
15 # Vocabulary
16 vocab_size = len(char_to_ix) # size of the vocabulary
17 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
18
19 # Parameters
20 n_embed = vocab_size # size of the embedding vectors
21 n_hidden = 128 # size of the hidden state vectors
22 n_out = vocab_size # size of the output vectors
23
24 # Layers
25 W = np.random.randn(n_hidden, n_hidden) # hidden to hidden weights
26 U = np.random.randn(n_hidden, n_embed) # hidden to input weights
27 V = np.random.randn(n_out, n_hidden) # hidden to output weights
28 Wx0 = np.zeros((n_hidden, n_hidden)) # hidden to hidden weights (initial)
29 Ux0 = np.zeros((n_hidden, n_embed)) # hidden to input weights (initial)
30 Vx0 = np.zeros((n_out, n_hidden)) # hidden to output weights (initial)
31
32 # Initial state
33 hprev = np.zeros((n_hidden,)) # hidden state
34
35 # Training
36 for i in range(1000000):
37     # Sample a batch of data
38     start_ix = np.random.randint(0, len(data)-seq_length)
39     targets = [char_to_ix[ch] for ch in data[start_ix+seq_length:]]
40     inputs = [char_to_ix[ch] for ch in data[start_ix:start_ix+seq_length]]
41
42     # Forward pass
43     h = hprev
44     for t in range(seq_length):
45         h, yhat = rnn_step(W, U, V, h, inputs[t])
46     loss = lossFun(yhat, targets)
47     smooth_loss = smooth_loss * 0.999 + loss * 0.001
48
49     # Backward pass
50     dyhat = -np.ones((seq_length,))
51     dW, dU, dV, dhprev = rnn_backward(W, U, V, h, inputs, targets, dyhat)
52     W += dW * learning_rate
53     U += dU * learning_rate
54     V += dV * learning_rate
55
56     # Print progress
57     if i % 100 == 0:
58         print 'iter %d, loss: %f' % (i, smooth_loss) # print progress
59
60 # Perform parameter update with Adagrad
61 for param, dparam, mem in zip([Wx0, Ux0, Vx0], [dW, dU, dV], [mem, mem, mem]):
62     mem += dparam * dparam
63     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
64
65 p += seq_length # move data pointer
66 n += 1 # iteration counter

```

```

67 # Sample from the model now and then
68 if n % 100 == 0:
69     sample_ix = sample(hprev, inputs[0], 200)
70     txt = ''.join(ix_to_char[ix] for ix in sample_ix)
71     print '----\n%s \n----' % (txt, )
72
73 # forward seq_length characters through the net and fetch gradient
74 loss, dxhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
75 smooth_loss = smooth_loss * 0.999 + loss * 0.001
76 if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
77
78 # perform parameter update with Adagrad
79 for param, dparam, mem in zip([Wx0, Ux0, Vx0], [dxhx, dwhh, dwhy, dbh, dby],
80                               [mem, mem, mem]):
81     mem += dparam * dparam
82     param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
83
84 p += seq_length # move data pointer
85 n += 1 # iteration counter

```



Main loop

```

81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '----\n%s \n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dxhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101     smooth_loss = smooth_loss * 0.999 + loss * 0.001
102     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104     # perform parameter update with Adagrad
105     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                   [dxhx, dwhh, dwhy, dbh, dby],
107                                   [mWxh, mWhh, mWhy, mbh, mby]):
108         mem += dparam * dparam
109         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter

```

min-char-rnn.py gist

```

"""
Minimal character-level neural network, written by Andre Kardecy (@kardecy)
"""
import random
import sys

# Model size
EMBED_DIM = 64
HIDDEN_DIM = 128
VOCAB_SIZE = 26

# Data
DATA_PATH = "data.txt"
EMBED_PATH = "embed.npy"
HIDDEN_PATH = "hidden.npy"

# Parameters
LEARNING_RATE = 1e-8
SMOOTH_LOSS = 0.999
BATCH_SIZE = 100

# Imports
import numpy as np
import random
import sys

# Data
def load_data():
    with open(DATA_PATH, "r") as f:
        data = f.read().lower()
    vocab = set(data)
    vocab_size = len(vocab)
    char_to_ix = {}
    ix_to_char = {}
    for i, ch in enumerate(vocab):
        char_to_ix[ch] = i
        ix_to_char[i] = ch
    return data, char_to_ix, ix_to_char, vocab_size

def load_embeddings():
    embed = np.load(EMBED_PATH)
    hidden = np.load(HIDDEN_PATH)
    return embed, hidden

# Model
def init_weights():
    Wxh = np.random.randn(EMBED_DIM, HIDDEN_DIM)
    Whh = np.random.randn(HIDDEN_DIM, HIDDEN_DIM)
    Wbh = np.random.randn(HIDDEN_DIM, VOCAB_SIZE)
    Wby = np.random.randn(HIDDEN_DIM, VOCAB_SIZE)
    return Wxh, Whh, Wbh, Wby

def forward(x, h):
    z = Wxh.dot(x) + Whh.dot(h)
    h = sigmoid(z)
    z = Wbh.dot(h) + Wby
    y = softmax(z)
    return h, y

def lossFun(y, targets):
    loss = -sum(targets * np.log(y))
    return loss

def train():
    data, char_to_ix, ix_to_char, vocab_size = load_data()
    embed, hidden = load_embeddings()
    Wxh, Whh, Wbh, Wby = init_weights()

    n = len(data)
    p = 0
    mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Wby)
    mbh, mby = np.zeros_like(hidden), np.zeros_like(vocab_size)
    smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0

    while True:
        # prepare inputs (we're sweeping from left to right in steps seq_length long)
        if p+seq_length+1 >= len(data) or n == 0:
            hprev = np.zeros((hidden_size,1)) # reset RNN memory
            p = 0 # go from start of data
            inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
            targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

            # sample from the model now and then
            if n % 100 == 0:
                sample_ix = sample(hprev, inputs[0], 200)
                txt = ''.join(ix_to_char[ix] for ix in sample_ix)
                print '----\n %s \n----' % (txt, )

            # forward seq_length characters through the net and fetch gradient
            loss, dxhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
            smooth_loss = smooth_loss * SMOOTH_LOSS + loss * 0.001

            if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

            # perform parameter update with Adagrad
            for param, dparam, mem in zip([Wxh, Whh, Wby, bh, by],
                [dxhx, dwhh, dwhy, dbh, dby],
                [mWxh, mWhh, mWhy, mbh, mby]):
                mem += dparam * dparam
                param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

            p += seq_length # move data pointer
            n += 1 # iteration counter
    
```

Main loop

```

81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Wby)
83 mbh, mby = np.zeros_like(hidden), np.zeros_like(vocab_size) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '----\n %s \n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dxhx, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101     smooth_loss = smooth_loss * 0.999 + loss * 0.001
102     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104     # perform parameter update with Adagrad
105     for param, dparam, mem in zip([Wxh, Whh, Wby, bh, by],
106                                   [dxhx, dwhh, dwhy, dbh, dby],
107                                   [mWxh, mWhh, mWhy, mbh, mby]):
108         mem += dparam * dparam
109         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter
    
```



min-char-rnn.py gist

```

1 """
2 # Licensed under the Apache License, Version 2.0 (the "License");
3 # you may not use this file except in compliance with the License.
4 # You may obtain a copy of the License at
5 # http://www.apache.org/licenses/LICENSE-2.0
6 # Unless required by applicable law or agreed to in writing,
7 # software distributed under the License is distributed on an "AS IS"
8 # BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
9 # express or implied. See the License for the specific language
10 # governing permissions and limitations under the License.
11 """
12 # Author: Matt Johnson
13 # Date: 2013-08-15
14 # This code is based on the implementation of the LSTM in the
15 # book "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron
16 # Courville.
17 # License: Apache 2.0
18 # This code is distributed as-is, without any warranty.
19 # For more information, see the README file.
20 """
21 # Imports
22 import argparse
23 import logging
24 import os
25 import sys
26 from collections import defaultdict
27 from math import exp, log, tanh, sigmoid
28 from random import random, randint
29 from time import time
30 from typing import List
31 from tqdm import tqdm
32 import numpy as np
33 import torch
34 import torch.nn as nn
35 import torch.nn.functional as F
36 from torch.autograd import Variable
37 from torch.nn.parameter import Parameter
38 from torch.nn.init import xavier_uniform, xavier_normal, constant
39 from torch.optim import Optimizer
40 from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
41 """
42 """
43 # Arguments
44 parser = argparse.ArgumentParser()
45 parser.add_argument('--data', type=str, required=True, help='Path to training data')
46 parser.add_argument('--vocab_size', type=int, default=27, help='Vocabulary size')
47 parser.add_argument('--hidden_size', type=int, default=128, help='Hidden layer size')
48 parser.add_argument('--embedding_size', type=int, default=128, help='Embedding size')
49 parser.add_argument('--num_layers', type=int, default=1, help='Number of layers')
50 parser.add_argument('--learning_rate', type=float, default=0.001, help='Learning rate')
51 parser.add_argument('--batch_size', type=int, default=128, help='Batch size')
52 parser.add_argument('--num_epochs', type=int, default=10, help='Number of epochs')
53 parser.add_argument('--seed', type=int, default=1, help='Random seed')
54 parser.add_argument('--log_dir', type=str, default='', help='Directory for logs')
55 parser.add_argument('--gpu_device', type=int, default=-1, help='GPU device ID')
56 parser.add_argument('--test_data', type=str, default='', help='Path to test data')
57 parser.add_argument('--test_batch_size', type=int, default=128, help='Test batch size')
58 parser.add_argument('--num_test_epochs', type=int, default=1, help='Number of test epochs')
59 """
60 """
61 # Global variables
62 n, p = 0, 0
63 mw, mh, mw_h, mw_hh, mw_by = np.zeros_like(w), np.zeros_like(h), np.zeros_like(why)
64 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
65 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
66 while True:
67     # prepare inputs (we're sweeping from left to right in steps seq_length long)
68     if p+seq_length+1 >= len(data) or n == 0:
69         hprev = np.zeros((hidden_size,1)) # reset RNN memory
70         p = 0 # go from start of data
71         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
72         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
73
74     # sample from the model now and then
75     if n % 100 == 0:
76         sample_ix = sample(hprev, inputs[0], 200)
77         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
78         print '----\n %s \n----' % (txt, )
79
80     # forward seq_length characters through the net and fetch gradient
81     loss, dmx, dwh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
82     smooth_loss = smooth_loss * 0.999 + loss * 0.001
83     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
84
85     # perform parameter update with Adagrad
86     for param, dparam, mem in zip([wx, wh, wx_h, wx_hh, wx_by,
87                                 [dwx, dwh, dwhy, dbh, dby],
88                                 [mw, mh, mw_h, mw_hh, mw_by]):
89         mem += dparam * dparam
90         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
91
92     p += seq_length # move data pointer
93     n += 1 # iteration counter

```

Main loop

```

81 n, p = 0, 0
82 mw, mh, mw_h, mw_hh, mw_by = np.zeros_like(w), np.zeros_like(h), np.zeros_like(why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93     # sample from the model now and then
94     if n % 100 == 0:
95         sample_ix = sample(hprev, inputs[0], 200)
96         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97         print '----\n %s \n----' % (txt, )
98
99     # forward seq_length characters through the net and fetch gradient
100     loss, dmx, dwh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101     smooth_loss = smooth_loss * 0.999 + loss * 0.001
102     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104     # perform parameter update with Adagrad
105     for param, dparam, mem in zip([wx, wh, wx_h, wx_hh, wx_by],
106                                 [dwx, dwh, dwhy, dbh, dby],
107                                 [mw, mh, mw_h, mw_hh, mw_by]):
108         mem += dparam * dparam
109         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111     p += seq_length # move data pointer
112     n += 1 # iteration counter

```

```

94     # perform parameter update with Adagrad
95     for param, dparam, mem in zip([wx, wh, wx_h, wx_hh, wx_by],
96                                 [dwx, dwh, dwhy, dbh, dby],
97                                 [mw, mh, mw_h, mw_hh, mw_by]):
98         mem += dparam * dparam
99         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

```



min-char-rnn.py gist

```

1 # -*- coding: utf-8 -*-
2 """
3 Example: character-level language model
4 """
5 # Dependencies
6 import sys
7 import random
8
9 # Parameters
10 # Number of hidden units per layer
11 n_hidden = 128
12 # Number of layers
13 n_layers = 2
14 # Learning rate
15 learning_rate = 0.01
16 # Number of epochs
17 n_epochs = 10
18 # Number of characters in the vocabulary
19 n_vocab = 27
20 # Vocabulary
21 char_to_ix = {}
22 ix_to_char = {}
23 # Training and validation data
24 data_loader = DataLoader(n_vocab, n_hidden, n_layers, learning_rate, n_epochs)
25 # Training loop
26 for epoch in range(n_epochs):
27     # Train
28     data_loader.train()
29     # Validation
30     data_loader.validate()
31     # Print progress
32     print('Epoch %d, loss: %f, validation loss: %f' % (epoch, data_loader.loss, data_loader.validation_loss))
33
34 # Save the model
35 data_loader.save_model('min-char-rnn.pkl')
36 """
37 """
38 """
39 """
40 """
41 """
42 """
43 """
44 """
45 """
46 """
47 """
48 """
49 """
50 """
51 """
52 """
53 """
54 """
55 """
56 """
57 """
58 """
59 """
60 """
61 """
62 """
63 """
64 """
65 """
66 """
67 """
68 """
69 """
70 """
71 """
72 """
73 """
74 """
75 """
76 """
77 """
78 """
79 """
80 """
81 """
82 """
83 """
84 """
85 """
86 """
87 """
88 """
89 """
90 """
91 """
92 """
93 """
94 """
95 """
96 """
97 """
98 """
99 """
100 """
101 """
102 """
103 """
104 """
105 """
106 """
107 """
108 """
109 """
110 """
111 """
112 """
113 """
114 """
115 """
116 """
117 """
118 """
119 """
120 """
121 """
122 """
123 """
124 """
125 """
126 """
127 """
128 """
129 """
130 """
131 """
132 """
133 """
134 """
135 """
136 """
137 """
138 """
139 """
140 """
141 """
142 """
143 """
144 """
145 """
146 """
147 """
148 """
149 """
150 """
151 """
152 """
153 """
154 """
155 """
156 """
157 """
158 """
159 """
160 """
161 """
162 """
163 """
164 """
165 """
166 """
167 """
168 """
169 """
170 """
171 """
172 """
173 """
174 """
175 """
176 """
177 """
178 """
179 """
180 """
181 """
182 """
183 """
184 """
185 """
186 """
187 """
188 """
189 """
190 """
191 """
192 """
193 """
194 """
195 """
196 """
197 """
198 """
199 """
200 """
201 """
202 """
203 """
204 """
205 """
206 """
207 """
208 """
209 """
210 """
211 """
212 """
213 """
214 """
215 """
216 """
217 """
218 """
219 """
220 """
221 """
222 """
223 """
224 """
225 """
226 """
227 """
228 """
229 """
230 """
231 """
232 """
233 """
234 """
235 """
236 """
237 """
238 """
239 """
240 """
241 """
242 """
243 """
244 """
245 """
246 """
247 """
248 """
249 """
250 """
251 """
252 """
253 """
254 """
255 """
256 """
257 """
258 """
259 """
260 """
261 """
262 """
263 """
264 """
265 """
266 """
267 """
268 """
269 """
270 """
271 """
272 """
273 """
274 """
275 """
276 """
277 """
278 """
279 """
280 """
281 """
282 """
283 """
284 """
285 """
286 """
287 """
288 """
289 """
290 """
291 """
292 """
293 """
294 """
295 """
296 """
297 """
298 """
299 """
300 """
301 """
302 """
303 """
304 """
305 """
306 """
307 """
308 """
309 """
310 """
311 """
312 """
313 """
314 """
315 """
316 """
317 """
318 """
319 """
320 """
321 """
322 """
323 """
324 """
325 """
326 """
327 """
328 """
329 """
330 """
331 """
332 """
333 """
334 """
335 """
336 """
337 """
338 """
339 """
340 """
341 """
342 """
343 """
344 """
345 """
346 """
347 """
348 """
349 """
350 """
351 """
352 """
353 """
354 """
355 """
356 """
357 """
358 """
359 """
360 """
361 """
362 """
363 """
364 """
365 """
366 """
367 """
368 """
369 """
370 """
371 """
372 """
373 """
374 """
375 """
376 """
377 """
378 """
379 """
380 """
381 """
382 """
383 """
384 """
385 """
386 """
387 """
388 """
389 """
390 """
391 """
392 """
393 """
394 """
395 """
396 """
397 """
398 """
399 """
400 """
401 """
402 """
403 """
404 """
405 """
406 """
407 """
408 """
409 """
410 """
411 """
412 """
413 """
414 """
415 """
416 """
417 """
418 """
419 """
420 """
421 """
422 """
423 """
424 """
425 """
426 """
427 """
428 """
429 """
430 """
431 """
432 """
433 """
434 """
435 """
436 """
437 """
438 """
439 """
440 """
441 """
442 """
443 """
444 """
445 """
446 """
447 """
448 """
449 """
450 """
451 """
452 """
453 """
454 """
455 """
456 """
457 """
458 """
459 """
460 """
461 """
462 """
463 """
464 """
465 """
466 """
467 """
468 """
469 """
470 """
471 """
472 """
473 """
474 """
475 """
476 """
477 """
478 """
479 """
480 """
481 """
482 """
483 """
484 """
485 """
486 """
487 """
488 """
489 """
490 """
491 """
492 """
493 """
494 """
495 """
496 """
497 """
498 """
499 """
500 """
501 """
502 """
503 """
504 """
505 """
506 """
507 """
508 """
509 """
510 """
511 """
512 """
513 """
514 """
515 """
516 """
517 """
518 """
519 """
520 """
521 """
522 """
523 """
524 """
525 """
526 """
527 """
528 """
529 """
530 """
531 """
532 """
533 """
534 """
535 """
536 """
537 """
538 """
539 """
540 """
541 """
542 """
543 """
544 """
545 """
546 """
547 """
548 """
549 """
550 """
551 """
552 """
553 """
554 """
555 """
556 """
557 """
558 """
559 """
560 """
561 """
562 """
563 """
564 """
565 """
566 """
567 """
568 """
569 """
570 """
571 """
572 """
573 """
574 """
575 """
576 """
577 """
578 """
579 """
580 """
581 """
582 """
583 """
584 """
585 """
586 """
587 """
588 """
589 """
590 """
591 """
592 """
593 """
594 """
595 """
596 """
597 """
598 """
599 """
600 """
601 """
602 """
603 """
604 """
605 """
606 """
607 """
608 """
609 """
610 """
611 """
612 """
613 """
614 """
615 """
616 """
617 """
618 """
619 """
620 """
621 """
622 """
623 """
624 """
625 """
626 """
627 """
628 """
629 """
630 """
631 """
632 """
633 """
634 """
635 """
636 """
637 """
638 """
639 """
640 """
641 """
642 """
643 """
644 """
645 """
646 """
647 """
648 """
649 """
650 """
651 """
652 """
653 """
654 """
655 """
656 """
657 """
658 """
659 """
660 """
661 """
662 """
663 """
664 """
665 """
666 """
667 """
668 """
669 """
670 """
671 """
672 """
673 """
674 """
675 """
676 """
677 """
678 """
679 """
680 """
681 """
682 """
683 """
684 """
685 """
686 """
687 """
688 """
689 """
690 """
691 """
692 """
693 """
694 """
695 """
696 """
697 """
698 """
699 """
700 """
701 """
702 """
703 """
704 """
705 """
706 """
707 """
708 """
709 """
710 """
711 """
712 """
713 """
714 """
715 """
716 """
717 """
718 """
719 """
720 """
721 """
722 """
723 """
724 """
725 """
726 """
727 """
728 """
729 """
730 """
731 """
732 """
733 """
734 """
735 """
736 """
737 """
738 """
739 """
740 """
741 """
742 """
743 """
744 """
745 """
746 """
747 """
748 """
749 """
750 """
751 """
752 """
753 """
754 """
755 """
756 """
757 """
758 """
759 """
760 """
761 """
762 """
763 """
764 """
765 """
766 """
767 """
768 """
769 """
770 """
771 """
772 """
773 """
774 """
775 """
776 """
777 """
778 """
779 """
780 """
781 """
782 """
783 """
784 """
785 """
786 """
787 """
788 """
789 """
790 """
791 """
792 """
793 """
794 """
795 """
796 """
797 """
798 """
799 """
800 """
801 """
802 """
803 """
804 """
805 """
806 """
807 """
808 """
809 """
810 """
811 """
812 """
813 """
814 """
815 """
816 """
817 """
818 """
819 """
820 """
821 """
822 """
823 """
824 """
825 """
826 """
827 """
828 """
829 """
830 """
831 """
832 """
833 """
834 """
835 """
836 """
837 """
838 """
839 """
840 """
841 """
842 """
843 """
844 """
845 """
846 """
847 """
848 """
849 """
850 """
851 """
852 """
853 """
854 """
855 """
856 """
857 """
858 """
859 """
860 """
861 """
862 """
863 """
864 """
865 """
866 """
867 """
868 """
869 """
870 """
871 """
872 """
873 """
874 """
875 """
876 """
877 """
878 """
879 """
880 """
881 """
882 """
883 """
884 """
885 """
886 """
887 """
888 """
889 """
890 """
891 """
892 """
893 """
894 """
895 """
896 """
897 """
898 """
899 """
900 """
901 """
902 """
903 """
904 """
905 """
906 """
907 """
908 """
909 """
910 """
911 """
912 """
913 """
914 """
915 """
916 """
917 """
918 """
919 """
920 """
921 """
922 """
923 """
924 """
925 """
926 """
927 """
928 """
929 """
930 """
931 """
932 """
933 """
934 """
935 """
936 """
937 """
938 """
939 """
940 """
941 """
942 """
943 """
944 """
945 """
946 """
947 """
948 """
949 """
950 """
951 """
952 """
953 """
954 """
955 """
956 """
957 """
958 """
959 """
960 """
961 """
962 """
963 """
964 """
965 """
966 """
967 """
968 """
969 """
970 """
971 """
972 """
973 """
974 """
975 """
976 """
977 """
978 """
979 """
980 """
981 """
982 """
983 """
984 """
985 """
986 """
987 """
988 """
989 """
990 """
991 """
992 """
993 """
994 """
995 """
996 """
997 """
998 """
999 """
1000 """

```

```

81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93         # sample from the model now and then
94         if n % 100 == 0:
95             sample_ix = sample(hprev, inputs[0], 200)
96             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97             print '----\n %s \n----' % (txt, )
98
99         # forward seq_length characters through the net and fetch gradient
100         loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101         smooth_loss = smooth_loss * 0.999 + loss * 0.001
102         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104         # perform parameter update with Adagrad
105         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                     [dWxh, dWhh, dWhy, dbh, dby],
107                                     [mWxh, mWhh, mWhy, mbh, mby]):
108             mem += dparam * dparam
109             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111         p += seq_length # move data pointer
112         n += 1 # iteration counter

```

Main loop

```

81 n, p = 0, 0
82 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
83 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
84 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
85 while True:
86     # prepare inputs (we're sweeping from left to right in steps seq_length long)
87     if p+seq_length+1 >= len(data) or n == 0:
88         hprev = np.zeros((hidden_size,1)) # reset RNN memory
89         p = 0 # go from start of data
90         inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
91         targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
92
93         # sample from the model now and then
94         if n % 100 == 0:
95             sample_ix = sample(hprev, inputs[0], 200)
96             txt = ''.join(ix_to_char[ix] for ix in sample_ix)
97             print '----\n %s \n----' % (txt, )
98
99         # forward seq_length characters through the net and fetch gradient
100         loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
101         smooth_loss = smooth_loss * 0.999 + loss * 0.001
102         if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
103
104         # perform parameter update with Adagrad
105         for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
106                                     [dWxh, dWhh, dWhy, dbh, dby],
107                                     [mWxh, mWhh, mWhy, mbh, mby]):
108             mem += dparam * dparam
109             param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
110
111         p += seq_length # move data pointer
112         n += 1 # iteration counter

```

min-char-rnn.py gist

```

1  """
2  Minimal character-level RNN model, written by Andre Karpathy (2015/05/21)
3  """
4
5  import numpy as np
6
7  # Device
8  dev = "/dev/null"
9  if torch.cuda.is_available():
10     dev = "cuda:0"
11
12 # Parameters
13 vocab_size = 27 # Number of unique characters (including <PAD>)
14 embed_dim = 64 # Embedding dimensionality
15 hidden_dim = 128 # Hidden state dimensionality
16 num_layers = 1 # Number of RNN layers
17 num_epochs = 10 # Number of training epochs
18 num_workers = 4 # Number of workers for data loading
19 lr = 1e-3 # Learning rate
20
21 # Data loader
22 data_loader = DataLoader.from_instances(
23     glob.glob("data/*"), # Paths to training and validation data files
24     num_workers=num_workers, # Number of workers for data loading
25     shuffle=True, # Shuffle data across workers
26     batch_size=32, # Batch size per worker
27     pin_memory=True, # Pin memory for faster transfer
28 )
29
30 # Model
31 model = LSTMModel(
32     vocab_size=vocab_size, # Vocabulary size
33     embed_dim=embed_dim, # Embedding dimensionality
34     hidden_dim=hidden_dim, # Hidden state dimensionality
35     num_layers=num_layers, # Number of RNN layers
36 )
37
38 # Optimizer
39 optimizer = optim.Adam(model.parameters())
40
41 # Training
42 for epoch in range(1, num_epochs + 1):
43     train_loss = 0
44     valid_loss = 0
45     for batch_idx, (inputs, targets, lengths) in enumerate(data_loader):
46         # Forward pass
47         outputs, hidden_states = model(inputs, hidden_states)
48         # Compute loss
49         loss = cross_entropy(outputs, targets)
50         # Backward pass
51         optimizer.backward(loss)
52         optimizer.step()
53         optimizer.zero_grad()
54     # Validation
55     with torch.no_grad():
56         valid_loss += compute_loss(model, data_loader.valid_loader)
57     # Save model
58     if epoch % 10 == 0:
59         torch.save(model.state_dict(), "model_{}.pt".format(epoch))
60
61 # Test
62 with torch.no_grad():
63     test_loss = compute_loss(model, data_loader.test_loader)
64
65 # Print final results
66 print("Epoch {}: train_loss={}, valid_loss={}, test_loss={}".format(
67     num_epochs, train_loss, valid_loss, test_loss))
68
69 # Cleanup
70 data_loader.unpin_memory()

```



Loss function

- forward pass (compute loss)
- backward pass (compute param gradient)

```

27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Nx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = [], [], [], {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36
37     # forward pass
38     for t in xrange(len(inputs)):
39         xs[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
40         xs[t][inputs[t]] = 1
41         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Wyh, hs[t-1]) + bh) # hidden state
42         ys[t] = np.dot(Wyh, hs[t]) + by # unnormalized log probabilities for next chars
43         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
44         loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)
45
46     # backward pass: compute gradients going backwards
47     dwhx, dwhh, dwhy = np.zeros_like(Wxh), np.zeros_like(Wyh), np.zeros_like(Wyh)
48     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
49     dhnxt = np.zeros_like(hs[0])
50     for t in reversed(xrange(len(inputs))):
51         dy = np.copy(ps[t])
52         dy[targets[t]] -= 1 # backprop into y
53         dwhy += np.dot(dy, hs[t].T)
54         dby = dy
55         dh = np.dot(Wyh.T, dy) + dhnxt # backprop into h
56         ddraw = (1 - hs[t] * hs[t].T) * dh # backprop through tanh nonlinearity
57         dbh = ddraw
58         dwhx += np.dot(ddraw, xs[t].T)
59         dwhh += np.dot(ddraw, hs[t-1].T)
60         dhnxt = np.dot(Wxh.T, ddraw)
61
62     for dparam in [dwhx, dwhh, dwhy, dbh, dby]:
63         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
64     return loss, dwhx, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]

```

min-char-rnn.py gist

```

1 """
2 Example: character-level language model. Exercise by Andrej Kardecy (2016/04).
3 """
4
5 import numpy as np
6
7 # Model size
8 vocab_size = 27 # 26 lowercase letters + space
9 num_hidden = 100 # number of hidden units in each layer
10 num_layers = 2 # number of hidden layers
11
12 # Input and target
13 data_dir = 'data'
14 train_data = 'train.txt'
15 dev_data = 'dev.txt'
16 test_data = 'test.txt'
17
18 # Load data
19 def load_data():
20     train_data = load_data(train_data)
21     dev_data = load_data(dev_data)
22     test_data = load_data(test_data)
23
24     # Pad data with zeros
25     train_data = pad_data(train_data)
26     dev_data = pad_data(dev_data)
27     test_data = pad_data(test_data)
28
29     return train_data, dev_data, test_data
30
31 # Pad data with zeros
32 def pad_data(data):
33     # Pad data with zeros
34     padded_data = np.zeros((data.shape[0], data.shape[1] + 1))
35     padded_data[:, :-1] = data
36
37     return padded_data
38
39 # Load data
40 train_data, dev_data, test_data = load_data()
41
42 # Initialize model parameters
43 def init_model():
44     # Initialize weights
45     W_xh = np.random.randn(vocab_size, num_hidden)
46     W_hh = np.random.randn(num_hidden, num_hidden)
47     W_hy = np.random.randn(num_hidden, vocab_size)
48     W_bx = np.zeros((1, num_hidden))
49     W_bh = np.zeros((1, num_hidden))
50     W_by = np.zeros((1, vocab_size))
51
52     # Initialize biases
53     b_xh = np.zeros((1, num_hidden))
54     b_hh = np.zeros((1, num_hidden))
55     b_hy = np.zeros((1, vocab_size))
56
57     return W_xh, W_hh, W_hy, W_bx, W_bh, W_by, b_xh, b_hh, b_hy
58
59 # Compute loss
60 def compute_loss(inputs, targets, hprev):
61     # Compute loss
62     loss = 0
63     h = hprev
64     for t in xrange(len(inputs)):
65         # Compute hidden state
66         x = inputs[t]
67         h = tanh(W_xh*x + W_hh*h + W_bx)
68
69         # Compute probabilities
70         y = W_hy*h + W_bh
71         ps = np.exp(y) / np.sum(np.exp(y))
72         loss += -np.log(ps[targets[t] - 1])
73
74     return loss
75
76 # Train model
77 def train_model():
78     # Train model
79     for epoch in xrange(100):
80         # Compute training loss
81         train_loss = compute_loss(train_data, train_data, None)
82
83         # Compute validation loss
84         dev_loss = compute_loss(dev_data, dev_data, None)
85
86         # Print loss
87         print 'epoch %d: train loss %f, dev loss %f' % (epoch, train_loss, dev_loss)
88
89         # Save model
90         save_model(W_xh, W_hh, W_hy, W_bx, W_bh, W_by, b_xh, b_hh, b_hy)
91
92 # Evaluate model
93 def evaluate_model():
94     # Evaluate model
95     W_xh, W_hh, W_hy, W_bx, W_bh, W_by, b_xh, b_hh, b_hy = load_model()
96
97     # Compute test loss
98     test_loss = compute_loss(test_data, test_data, None)
99
100     # Print test loss
101     print 'test loss %f' % test_loss
102
103 # Main
104 if __name__ == '__main__':
105     train_model()
106     evaluate_model()
107 """

```

```

27 def lossFun(inputs, targets, hprev):
28     """
29     inputs, targets are both list of integers.
30     hprev is Hx1 array of initial hidden state
31     returns the loss, gradients on model parameters, and last hidden state
32     """
33     xs, hs, ys, ps = {}, {}, {}, {}
34     hs[-1] = np.copy(hprev)
35     loss = 0
36     # forward pass
37     for t in xrange(len(inputs)):
38         xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
39         xs[t][inputs[t]] = 1
40         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
41         ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
42         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
43         loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)

```

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Softmax classifier

min-char-rnn.py gist

```

1 #!/usr/bin/env python
2 """
3 min-char-rnn.py
4 """
5 import sys
6 import numpy as np
7 import random
8
9 # Parameters
10 # Number of hidden units
11 HIDDEN_DIM = 100
12 # Number of input units
13 INPUT_DIM = 26
14 # Number of output units
15 OUTPUT_DIM = 26
16 # Learning rate
17 LEARNING_RATE = 0.01
18 # Number of epochs
19 EPOCHS = 100
20 # Batch size
21 BATCH_SIZE = 100
22 # Seed
23 SEED = 12345
24
25 # Initialize weights
26 def init_weights():
27     W_xh = np.random.randn(INPUT_DIM, HIDDEN_DIM)
28     W_hh = np.random.randn(HIDDEN_DIM, HIDDEN_DIM)
29     W_hy = np.random.randn(HIDDEN_DIM, OUTPUT_DIM)
30     b_h = np.zeros(HIDDEN_DIM)
31     b_y = np.zeros(OUTPUT_DIM)
32     return W_xh, W_hh, W_hy, b_h, b_y
33
34 # Forward pass
35 def forward_pass(x, W_xh, W_hh, W_hy, b_h, b_y):
36     h = b_h
37     for i in range(len(x)):
38         h = np.tanh(W_xh.dot(x[i]) + W_hh.dot(h) + b_h)
39     y = W_hy.dot(h) + b_y
40     return y
41
42 # Backward pass
43 def backward_pass(y, W_xh, W_hh, W_hy, b_h, b_y):
44     # Compute gradients going backwards
45     dW_xh, dW_hh, dW_hy = np.zeros_like(W_xh), np.zeros_like(W_hh), np.zeros_like(W_hy)
46     dbh, dby = np.zeros_like(b_h), np.zeros_like(b_y)
47     dhnext = np.zeros_like(h)
48     for t in reversed(range(len(inputs))):
49         dy = np.copy(ps[t])
50         dy[targets[t]] -= 1 # backprop into y
51         dwhy += np.dot(dy, hs[t].T)
52         dby += dy
53         dh = np.dot(why.T, dy) + dhnext # backprop into h
54         dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += dhraw
56         dW_xh += np.dot(dhraw, xs[t].T)
57         dW_hh += np.dot(dhraw, hs[t-1].T)
58         dhnext = np.dot(W_hh.T, dhraw)
59     for dparam in [dW_xh, dW_hh, dwhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dW_xh, dW_hh, dwhy, dbh, dby, hs[len(inputs)-1]

```

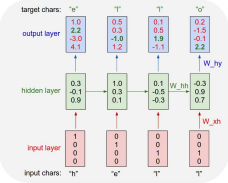
```

44 # backward pass: compute gradients going backwards
45 dW_xh, dW_hh, dwhy = np.zeros_like(W_xh), np.zeros_like(W_hh), np.zeros_like(W_hy)
46 dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47 dhnext = np.zeros_like(hs[0])
48 for t in reversed(range(len(inputs))):
49     dy = np.copy(ps[t])
50     dy[targets[t]] -= 1 # backprop into y
51     dwhy += np.dot(dy, hs[t].T)
52     dby += dy
53     dh = np.dot(why.T, dy) + dhnext # backprop into h
54     dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55     dbh += dhraw
56     dW_xh += np.dot(dhraw, xs[t].T)
57     dW_hh += np.dot(dhraw, hs[t-1].T)
58     dhnext = np.dot(W_hh.T, dhraw)
59 for dparam in [dW_xh, dW_hh, dwhy, dbh, dby]:
60     np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61 return loss, dW_xh, dW_hh, dwhy, dbh, dby, hs[len(inputs)-1]

```



recall:



min-char-rnn.py gist

```

1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 """
4 Example: simple character-level language model
5 """
6
7 import numpy as np
8
9 # Model parameters
10 # Vocabulary size
11 vocab_size = 256
12 # Embedding dimension
13 emb_dim = 128
14 # Hidden state dimension
15 hidden_dim = 128
16 # Number of hidden units
17 num_hidden = hidden_dim
18 # Number of layers
19 num_layers = 2
20 # Learning rate
21 lr = 0.001
22 # Number of epochs
23 num_epochs = 100
24 # Batch size
25 batch_size = 128
26 # Training data
27 train_data = 'data/train.txt'
28 # Validation data
29 val_data = 'data/val.txt'
30 # Test data
31 test_data = 'data/test.txt'
32 # Maximum length of a sentence
33 max_length = 100
34 # Number of samples per batch
35 num_samples_per_batch = 10000
36 # Number of samples per validation batch
37 num_samples_per_val_batch = 1000
38 # Number of samples per test batch
39 num_samples_per_test_batch = 1000
40 # Random seed
41 random_seed = 12345
42 # Embedding matrix
43 emb_matrix = np.zeros((vocab_size, emb_dim))
44 # Hidden state matrix
45 hidden_matrix = np.zeros((num_hidden, hidden_dim))
46 # Weights
47 Wxh = np.zeros((emb_dim, hidden_dim))
48 Whh = np.zeros((hidden_dim, hidden_dim))
49 Wxv = np.zeros((emb_dim, vocab_size))
50 W_hv = np.zeros((hidden_dim, vocab_size))
51 # Biases
52 bh = np.zeros((hidden_dim,))
53 bv = np.zeros((vocab_size,))
54 # Training
55 def train():
56     # Initialize weights and biases
57     np.random.seed(random_seed)
58     Wxh = np.random.randn(emb_dim, hidden_dim)
59     Whh = np.random.randn(hidden_dim, hidden_dim)
60     Wxv = np.random.randn(emb_dim, vocab_size)
61     W_hv = np.random.randn(hidden_dim, vocab_size)
62     bh = np.random.randn(hidden_dim)
63     bv = np.random.randn(vocab_size)
64     # Training loop
65     for epoch in range(1, num_epochs + 1):
66         # Shuffle training data
67         train_data = np.random.permutation(train_data)
68         # Iterate over batches
69         for i in range(0, len(train_data), batch_size):
70             # Get batch
71             batch = train_data[i:i + batch_size]
72             # Forward pass
73             # ... (omitted)
74             # Backward pass
75             # ... (omitted)
76             # Update weights and biases
77             # ... (omitted)
78     # Save weights and biases
79     np.save('weights.npy', {'Wxh': Wxh, 'Whh': Whh, 'Wxv': Wxv, 'W_hv': W_hv, 'bh': bh, 'bv': bv})
80
81 # Validation
82 def validate():
83     # Load weights and biases
84     weights = np.load('weights.npy')
85     Wxh, Whh, Wxv, W_hv, bh, bv = weights
86     # Iterate over validation data
87     for i in range(0, len(val_data), batch_size):
88         # Get batch
89         batch = val_data[i:i + batch_size]
90         # Forward pass
91         # ... (omitted)
92         # Compute loss
93         # ... (omitted)
94
95 # Test
96 def test():
97     # Load weights and biases
98     weights = np.load('weights.npy')
99     Wxh, Whh, Wxv, W_hv, bh, bv = weights
100     # Iterate over test data
101     for i in range(0, len(test_data), batch_size):
102         # Get batch
103         batch = test_data[i:i + batch_size]
104         # Forward pass
105         # ... (omitted)
106         # Compute loss
107         # ... (omitted)
108
109 # Main
110 if __name__ == '__main__':
111     train()
112     validate()
113     test()

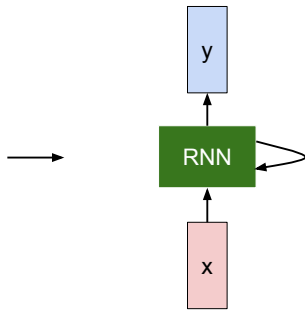
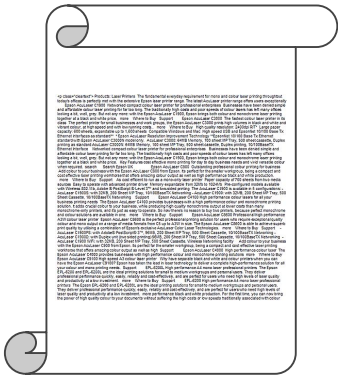
```



```

63 def sample(h, seed_ix, n):
64     """
65     sample a sequence of integers from the model
66     h is memory state, seed_ix is seed letter for first time step
67     """
68     x = np.zeros((vocab_size, 1))
69     x[seed_ix] = 1
70     ixes = []
71     for t in xrange(n):
72         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
73         y = np.dot(W_hv, h) + bv
74         p = np.exp(y) / np.sum(np.exp(y))
75         ix = np.random.choice(range(vocab_size), p=p.ravel())
76         x = np.zeros((vocab_size, 1))
77         x[ix] = 1
78         ixes.append(ix)
79     return ixes

```



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 10 - 34 8 Feb 2016

Sonnet 116 – Let me not ...

by William Shakespeare

Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove:
O no! it is an ever-fixed mark
That looks on tempests and is never shaken;
It is the star to every wandering bark,
Whose worth's unknown, although his height be taken.
Love's not Time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
If this be error and upon me proved,
I never writ, nor no man ever loved.

at first:

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgrd t o idoe ns,smtt h ne etie h,hregtrs nigtkie,aoaenns lng

↓
train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."

↓
train more

Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and offer.

↓
train more

"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.


VIOLA:

Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:





















O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

open source textbook on algebraic geometry

 **The Stacks Project**

[home](#) [about](#) [tags explained](#) [tag lookup](#) [browse](#) [search](#) [bibliography](#) [recent comments](#) [blog](#) [add slogans](#)

Browse chapters

Part	Chapter	online	TeX source	view pdf
Preliminaries				
	1. Introduction	online	tex 	pdf 
	2. Conventions	online	tex 	pdf 
	3. Set Theory	online	tex 	pdf 
	4. Categories	online	tex 	pdf 
	5. Topology	online	tex 	pdf 
	6. Sheaves on Spaces	online	tex 	pdf 
	7. Sites and Sheaves	online	tex 	pdf 
	8. Stacks	online	tex 	pdf 
	9. Fields	online	tex 	pdf 
	10. Commutative Algebra	online	tex 	pdf 

Parts

- [Preliminaries](#)
- [Schemes](#)
- [Topics in Scheme Theory](#)
- [Algebraic Spaces](#)
- [Topics in Geometry](#)
- [Deformation Theory](#)
- [Algebraic Stacks](#)
- [Miscellany](#)

Statistics

The Stacks project now consists of

- 455910 lines of code
- 14221 tags (56 inactive tags)
- 2366 sections

Latex source 

For $\bigoplus_{i=1, \dots, m}$ where $\mathcal{L}_{m_i} = 0$, hence we can find a closed subset \mathcal{H} in \mathcal{H} and any sets \mathcal{F} on X , U is a closed immersion of S , then $U \rightarrow T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparably in the fibre product covering we have to prove the lemma generated by $\prod Z \times_U U \rightarrow V$. Consider the maps M along the set of points Sch_{fpf} and $U \rightarrow V$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ?? . Hence we obtain a scheme S and any open subset $W \subset U$ in $\text{Sh}(G)$ such that $\text{Spec}(R) \rightarrow S$ is smooth or an

$$U = \bigcup U_i \times_S U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S . We claim that $\mathcal{O}_{X,p}$ is a scheme where $x, x', x'' \in S'$ such that $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}_{X,x''}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\text{GL}_{S'}(x/S')$ and we win. \square

To prove study we see that $\mathcal{F}_{|U}$ is a covering of \mathcal{X}' , and \mathcal{T}_i is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and \mathcal{F}_p exists and let \mathcal{F}_i be a presheaf of \mathcal{O}_X -modules on \mathcal{C} as a \mathcal{F} -module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\tilde{M}^* = \mathcal{I}^* \otimes_{\text{Spec}(k)} \mathcal{O}_{S,a} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)_{\text{fpf}}^{\text{ppf}}, (\text{Sch}/S)_{\text{fpf}}$$

and

$$V = \Gamma(S, \mathcal{O}) \rightarrow (U, \text{Spec}(A))$$

is an open subset of X . Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S .

Proof. See discussion of sheaves of sets. \square

The result to prove any open covering follows from the less of Example ?? . It may replace S by $X_{\text{spaces, étale}}$ which gives an open subspace of X and T equal to $S_{Z_{\text{ét}}}$, see Descent, Lemma ?? . Namely, by Lemma ?? we see that R is geometrically regular over S .

Lemma 0.1. Assume (3) and (3) by the construction in the description.

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\text{Proj}_X(A) = \text{Spec}(B)$ over U compatible with the complex

$$\text{Set}(A) = \Gamma(X, \mathcal{O}_{X, \mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \rightarrow \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with dense fields of S . Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) f is locally of finite type. Since $S = \text{Spec}(R)$ and $Y = \text{Spec}(R)$.

Proof. This is form all sheaves of sheaves on X . But given a scheme U and a surjective étale morphism $U \rightarrow X$. Let $U \cap U = \prod_{i=1, \dots, n} U_i$ be the scheme X over S at the schemes $X_i \rightarrow X$ and $U = \lim_i X_i$. \square

The following lemma surjective retrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{x, \dots, 0}$.

Lemma 0.2. Let X be a locally Noetherian scheme over S , $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_i \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq \mathfrak{p}$ is a subset of $\mathcal{J}_{n,0} \circ \mathcal{A}_2$ works.

Lemma 0.3. In Situation ?? . Hence we may assume $\mathfrak{q}' = 0$.

Proof. We will use the property we see that \mathfrak{p} is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F -algebra where δ_{n+1} is a scheme over S . \square

Proof. Omitted. □

Lemma 0.1. Let \mathcal{C} be a set of the construction.

Let \mathcal{C} be a gerber covering. Let \mathcal{F} be a quasi-coherent sheaves of \mathcal{O} -modules. We have to show that

$$\mathcal{O}_{\mathcal{X}} = \mathcal{O}_X(\mathcal{L})$$

Proof. This is an algebraic space with the composition of sheaves \mathcal{F} on $X_{\text{étale}}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{ \text{morph}_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F}) \}$$

where \mathcal{G} defines an isomorphism $\mathcal{F} \rightarrow \mathcal{F}$ of \mathcal{O} -modules. □

Lemma 0.2. This is an integer Z is injective.

Proof. See Spaces, Lemma ?? □

Lemma 0.3. Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $U \subset X$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \rightarrow Y' \rightarrow Y \rightarrow Y \rightarrow Y' \times_X Y \rightarrow X.$$

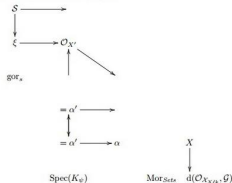
be a morphism of algebraic spaces over S and Y .

Proof. Let X be a nonzero scheme of X . Let X be an algebraic space. Let \mathcal{F} be a quasi-coherent sheaf of \mathcal{O}_X -modules. The following are equivalent

- (1) \mathcal{F} is an algebraic space over S .
- (2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram



is a limit. Then \mathcal{G} is a finite type and assume S is a flat and \mathcal{F} and \mathcal{G} is a finite type f_* . This is of finite type diagrams, and

- the composition of \mathcal{G} is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

Proof. We have see that $X = \text{Spec}(R)$ and \mathcal{F} is a finite type representable by algebraic space. The property \mathcal{F} is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U . □

Proof. This is clear that \mathcal{G} is a finite presentation, see Lemmas ??
A reduced above we conclude that U is an open covering of \mathcal{C} . The functor \mathcal{F} is a "field"

$$\mathcal{O}_{X,x} \rightarrow \mathcal{F}_x \rightarrow \mathcal{O}_{X_{\text{étale}}} \rightarrow \mathcal{O}_x^1 \mathcal{O}_X(\mathcal{O}_{X,x}^0)$$

is an isomorphism of covering of $\mathcal{O}_{X'}$. If \mathcal{F} is the unique element of \mathcal{F} such that X is an isomorphism.

The property \mathcal{F} is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme \mathcal{O}_X -algebra with \mathcal{F} are opens of finite type over S . If \mathcal{F} is a scheme theoretic image points. □

If \mathcal{F} is a finite direct sum \mathcal{O}_X , is a closed immersion, see Lemma ??
This is a sequence of \mathcal{F} is a similar morphism.

This repository Search Explore Gist Blog Help karpthy +- ⚙️ 🔍

torvalds / linux Watch - 3,711 Star 23,054 Fork 9,141

Linux kernel source tree

520,037 commits 1 branch 420 releases 5,039 contributors

branch: master - linux / +

Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux

torvalds authored 9 hours ago latest commit 4b1786927d

Documentation	Merge git://git.kernel.org/pub/scm/linux/kernel/git/nab/target-pending	8 days ago
arch	Merge branch 'x86-urgent-for-linus' of git://git.kernel.org/pub/scm/...	a day ago
block	block: discard bdi_unregister() in favour of bdi_destroy()	9 days ago
crypto	Merge git://git.kernel.org/pub/scm/linux/kernel/git/herbert/crypto-2.6	10 days ago
drivers	Merge branch 'drm-fixes' of git://people.freedesktop.org/~airlied/linux	9 hours ago
firmware	firmware/ihex2fw.c: restore missing default in switch statement	2 months ago
fs	vfs: read file_handle only once in handle_to_path	4 days ago
include	Merge branch 'perf-urgent-for-linus' of git://git.kernel.org/pub/scm/...	a day ago
init	init: fix regression by supporting devices with major:minor:offset fo...	a month ago

Code 74 Pull requests Pulse Graphs

HTTPS clone URL
https://github.c

You can clone with HTTPS, SSH, or Subversion.

Clone in Desktop Download ZIP

Fei-Fei Li & Andrej Karpathy & Justin Johnson

Lecture 10 - 41

8 Feb 2016

```

static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << i))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000ffffffff) & 0x000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}

```

Generated C code

```
/*
 * Copyright (c) 2006-2010, Intel Mobile Communications. All rights reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>
```

```

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>

#define REG_PG    vesa_slot_addr_pack
#define PFM_NOCOMP  AFSR(0, load)
#define STACK_DDR(type)    (func)

#define SWAP_ALLOCATE(nr)    (e)
#define emulate_sigs()  arch_get_unaligned_child()
#define access_rw(TST)  asm volatile("movd %%esp, %0, %3" : : "r" (0)); \
    if (__type & DO_READ)

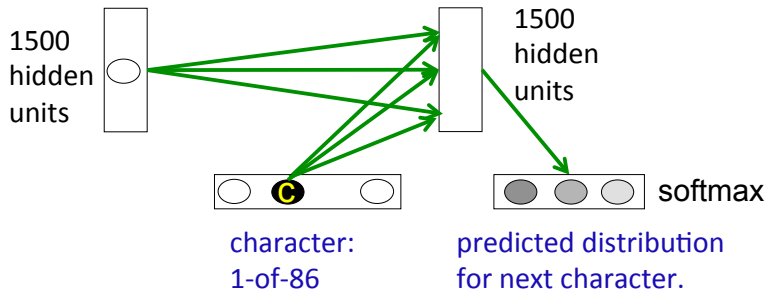
static void stat_PC_SEC __read_mostly offsetof(struct seq_argsqueue, \
    pC>[1]);

static void
os_prefix(unsigned long sys)
{
#ifdef CONFIG_PREEMPT
    PUT_PARAM_RAID(2, sel) = get_state_state();
    set_pid_sum((unsigned long)state, current_state_str(),
        (unsigned long)-1->lr_full; low;
}

```


Ideal model?

An obvious recurrent neural net

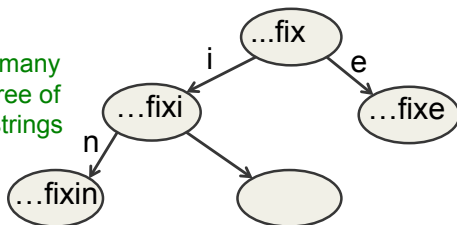


It's a lot easier to predict 86 characters than 100,000 words.

A slight tweak: Ideal tree model

An ideal model considers all previous input characters and the current character

There are exponentially many nodes in the tree of all character strings of length N .



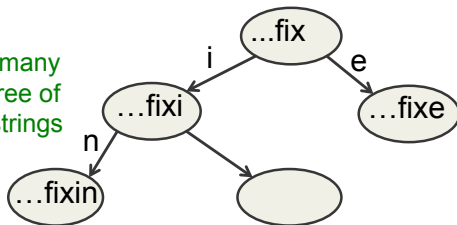
In an RNN, each node is a hidden state vector. The next character must transform this to a new node.

- The next hidden representation needs to depend on the **conjunction** of the current character and the current hidden representation
 - We expect under each hidden state vector and each current character, we should have a different transition matrix. The earlier simple model tried to capture this but is kind of indirect

A slight tweak: Ideal tree model

An ideal model considers all previous input characters and the current character

There are exponentially many nodes in the tree of all character strings of length N .



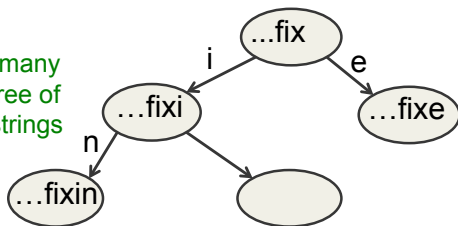
In an RNN, each node is a hidden state vector. The next character must transform this to a new node.

- The next hidden representation needs to depend on the **conjunction** of the current character and the current hidden representation
 - We expect under each hidden state vector and each current character, we should have a different transition matrix. The earlier simple model tried to capture this but is kind of indirect

A slight tweak: Ideal tree model

An ideal model considers all previous input characters and the current character

There are exponentially many nodes in the tree of all character strings of length N .



In an RNN, each node is a hidden state vector. The next character must transform this to a new node.

- The next hidden representation needs to depend on the **conjunction** of the current character and the current hidden representation
 - We expect under each hidden state vector and each current character, we should have a different transition matrix. The earlier simple model tried to capture this but is kind of indirect

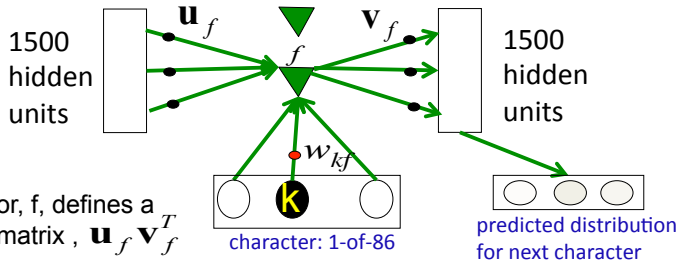
Multiplicative connections

- We may prepare a different transition matrix for each input
 - But this requires $86 \times 1500 \times 1500$ parameters (let say we have 1500 hidden variables)
 - And this could make the net overfit
- Can we achieve the same kind of multiplicative interaction using fewer parameters?
 - We want a different transition matrix for each of the 86 characters, but we want these 86 character-specific weight matrices to share parameters (the characters 9 and 8 should have similar matrices)

Multiplicative connections

- We may prepare a different transition matrix for each input
 - But this requires $86 \times 1500 \times 1500$ parameters (let say we have 1500 hidden variables)
 - And this could make the net overfit
- Can we achieve the same kind of multiplicative interaction using fewer parameters?
 - We want a different transition matrix for each of the 86 characters, but we want these 86 character-specific weight matrices to share parameters (the characters 9 and 8 should have similar matrices)

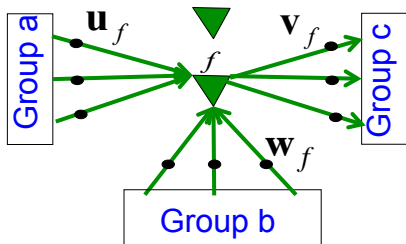
Using 3-way factors to allow a character to create a whole transition matrix



Each factor, f , defines a rank one matrix, $\mathbf{u}_f \mathbf{v}_f^T$

Each character, k , determines a gain w_{kf} for each of these matrices.

Using factors to implement multiplicative interactions

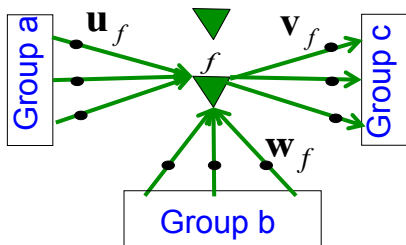


Vector input to group c:

$$c_f = \underbrace{(b^T w_f)}_{\text{Scalar input from group b}} \underbrace{(a^T u_f)}_{\text{Scalar input from group a}} v_f$$

- We can get groups a and b to interact multiplicatively by using “factors”
 - Each factor first computes a weighted sum for each of its input groups

Using factors to implement multiplicative interactions

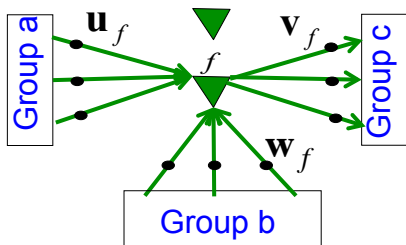


Vector input to group c:

$$c_f = \underbrace{(b^T w_f)}_{\text{Scalar input from group b}} \underbrace{(a^T u_f)}_{\text{Scalar input from group a}} v_f$$

- We can get groups a and b to interact multiplicatively by using “factors”
 - Each factor first computes a weighted sum for each of its input groups

Using factors to implement multiplicative interactions

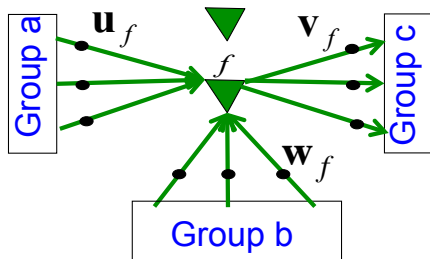


Vector input to group c:

$$c_f = \underbrace{(b^T w_f)}_{\text{Scalar input from group b}} \underbrace{(a^T u_f)}_{\text{Scalar input from group a}} v_f$$

- We can get groups a and b to interact multiplicatively by using “factors”
 - Each factor first computes a weighted sum for each of its input groups

Using factors to implement a set of basis matrices

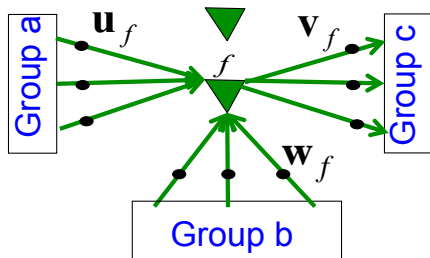


$$\begin{aligned}
 c_f &= (b^T w_f)(a^T u_f) v_f \\
 &= (b^T w_f) v_f (u_f^T a) \\
 &= \underbrace{(b^T w_f)}_{\text{scalar coefficient}} \underbrace{(v_f u_f^T)}_{\text{outer product transition matrix with rank 1}} a
 \end{aligned}$$

- We can think about factors another way:
 - Each factor defines a rank 1 transition matrix from a to c

$$c = \left(\sum_f (b^T w_f)(v_f u_f^T) \right) a$$

Using factors to implement a set of basis matrices

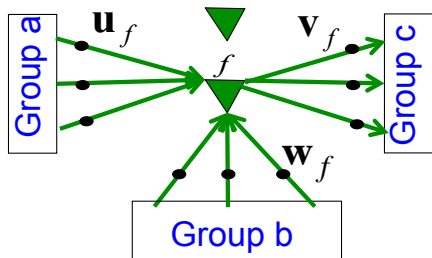


$$\begin{aligned}
 c_f &= (\mathbf{b}^T \mathbf{w}_f)(\mathbf{a}^T \mathbf{u}_f) \mathbf{v}_f \\
 &= (\mathbf{b}^T \mathbf{w}_f) \mathbf{v}_f (\mathbf{u}_f^T \mathbf{a}) \\
 &= \underbrace{(\mathbf{b}^T \mathbf{w}_f)}_{\text{scalar coefficient}} \underbrace{(\mathbf{v}_f \mathbf{u}_f^T)}_{\text{outer product transition matrix with rank 1}} \mathbf{a}
 \end{aligned}$$

- We can think about factors another way:
 - Each factor defines a rank 1 transition matrix from a to c

$$\mathbf{c} = \left(\sum_f (\mathbf{b}^T \mathbf{w}_f) (\mathbf{v}_f \mathbf{u}_f^T) \right) \mathbf{a}$$

Using factors to implement a set of basis matrices

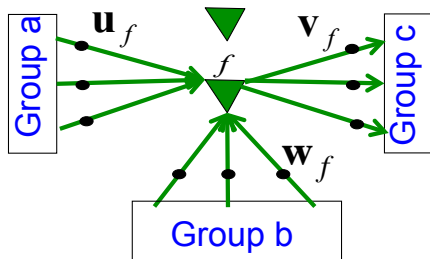


$$\begin{aligned}
 c_f &= (\mathbf{b}^T \mathbf{w}_f)(\mathbf{a}^T \mathbf{u}_f) v_f \\
 &= (\mathbf{b}^T \mathbf{w}_f) v_f (\mathbf{u}_f^T \mathbf{a}) \\
 &= \underbrace{(\mathbf{b}^T \mathbf{w}_f)}_{\text{scalar coefficient}} \underbrace{(\mathbf{v}_f \mathbf{u}_f^T)}_{\text{outer product transition matrix with rank 1}} \mathbf{a}
 \end{aligned}$$

- We can think about factors another way:
 - Each factor defines a rank 1 transition matrix from a to c

$$\mathbf{c} = \left(\sum_f (\mathbf{b}^T \mathbf{w}_f) (\mathbf{v}_f \mathbf{u}_f^T) \right) \mathbf{a}$$

Using factors to implement a set of basis matrices



$$\begin{aligned}
 c_f &= (b^T w_f)(a^T u_f) v_f \\
 &= (b^T w_f) v_f (u_f^T a) \\
 &= \underbrace{(b^T w_f)}_{\text{scalar coefficient}} \underbrace{(v_f u_f^T)}_{\text{outer product transition matrix with rank 1}} a
 \end{aligned}$$

- We can think about factors another way:
 - Each factor defines a rank 1 transition matrix from a to c

$$c = \left(\sum_f (b^T w_f) (v_f u_f^T) \right) a$$

Some note on optimization

- To optimize efficiently, they use Hessian-free (HF) method to minimize the cost
- HF is a second order method similar to Newton methods and LBFGS that take advantage of the curvature (Hessian) matrix
- In the HF method, they make an approximation to the curvature matrix and then minimize the error using conjugate gradient method. Then they make another approximation to the curvature matrix and minimize again

Some note on optimization

- To optimize efficiently, they use Hessian-free (HF) method to minimize the cost
- HF is a second order method similar to Newton methods and LBFGS that take advantage of the curvature (Hessian) matrix
- In the HF method, they make an approximation to the curvature matrix and then minimize the error using conjugate gradient method. Then they make another approximation to the curvature matrix and minimize again

Some note on optimization

- To optimize efficiently, they use Hessian-free (HF) method to minimize the cost
- HF is a second order method similar to Newton methods and LBFGS that take advantage of the curvature (Hessian) matrix
- In the HF method, they make an approximation to the curvature matrix and then minimize the error using conjugate gradient method. Then they make another approximation to the curvature matrix and minimize again

Conjugate gradient

- There is an alternative to going to the minimum in one step by multiplying by the inverse of the curvature matrix
- Use a sequence of steps each of which finds the minimum along one direction
- Make sure that each new direction is “conjugate” to the previous directions so you do not mess up the minimization you already did.
 - “conjugate” means that as you go in the new direction, you do not change the gradients in the previous directions

Conjugate gradient

- There is an alternative to going to the minimum in one step by multiplying by the inverse of the curvature matrix
- Use a sequence of steps each of which finds the minimum along one direction
- Make sure that each new direction is “conjugate” to the previous directions so you do not mess up the minimization you already did.
 - “conjugate” means that as you go in the new direction, you do not change the gradients in the previous directions

Conjugate gradient

- There is an alternative to going to the minimum in one step by multiplying by the inverse of the curvature matrix
- Use a sequence of steps each of which finds the minimum along one direction
- Make sure that each new direction is “conjugate” to the previous directions so you do not mess up the minimization you already did.
 - “conjugate” means that as you go in the new direction, you do not change the gradients in the previous directions

Training the model

- Ilya Sutskever used 5 million strings of 100 characters taken from wikipedia. For each string he starts predicting at the 11th character
- Using the HF optimizer, it took a month on a GPU board to get a really good model (back in 2011) text

Result

He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters' sisters in lower coil trains were always operated on the line of the **ephemerable** street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS). The B every chord was a "strongly cold internal palette pour even the white blade."

Result: some completions produced by the model

- Sheila thrunges (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6 th try)
- The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. (one of the first 10 tries for a model trained for longer)

Result: some completions produced by the model

- Sheila thrunges (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6 th try)
- The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. (one of the first 10 tries for a model trained for longer)

Result: some completions produced by the model

- Sheila thrunges (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6 th try)
- The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. (one of the first 10 tries for a model trained for longer)

Result: some completions produced by the model

- Sheila thrunges (most frequent)
- People thrunge (most frequent next character is space)
- Shiela, Thrungelini del Rey (first try)
- The meaning of life is literary recognition. (6 th try)
- The meaning of life is the tradition of the ancient human reproduction: it is less favorable to the good boy for when to remove her bigger. (one of the first 10 tries for a model trained for longer)

Result: what does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers
- It is good at balancing quotes and brackets
 - It can count brackets: none, one, many
- It knows a lot about syntax but its very hard to pin down exactly what grammar it actually “knows”
- It knows a lot of weak semantic associations
 - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable

Result: what does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers
- It is good at balancing quotes and brackets
 - It can count brackets: none, one, many
- It knows a lot about syntax but its very hard to pin down exactly what grammar it actually “knows”
- It knows a lot of weak semantic associations
 - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable

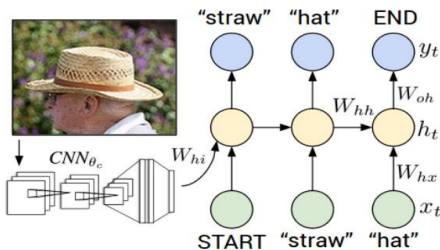
Result: what does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers
- It is good at balancing quotes and brackets
 - It can count brackets: none, one, many
- It knows a lot about syntax but its very hard to pin down exactly what grammar it actually “knows”
- It knows a lot of weak semantic associations
 - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable

Result: what does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers
- It is good at balancing quotes and brackets
 - It can count brackets: none, one, many
- It knows a lot about syntax but its very hard to pin down exactly what grammar it actually “knows”
- It knows a lot of weak semantic associations
 - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable

Image Captioning



Explain Images with Multimodal Recurrent Neural Networks, Mao et al.

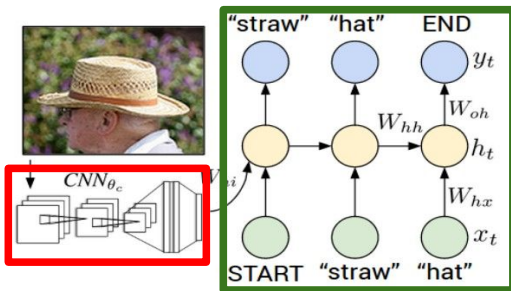
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

Show and Tell: A Neural Image Caption Generator, Vinyals et al.

Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.

Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Recurrent Neural Network



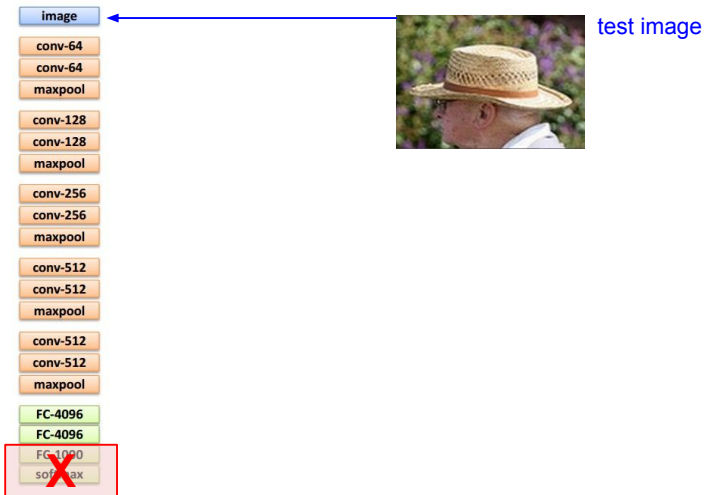
Convolutional Neural Network

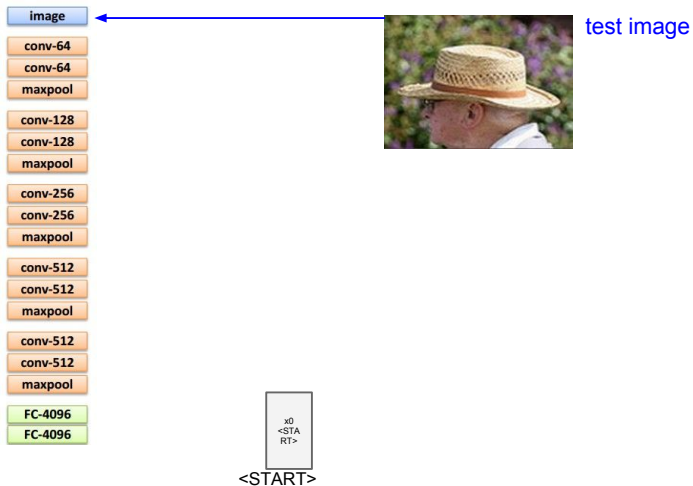


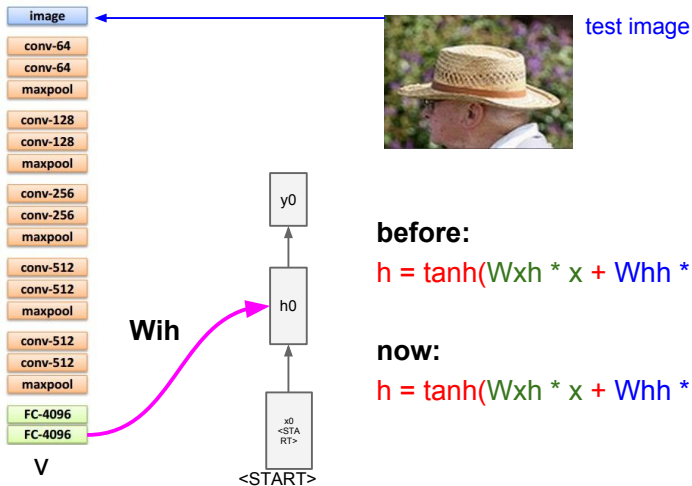
test image

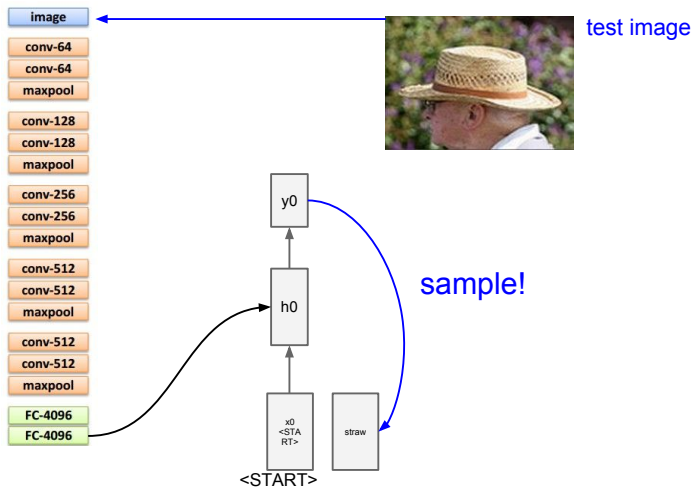


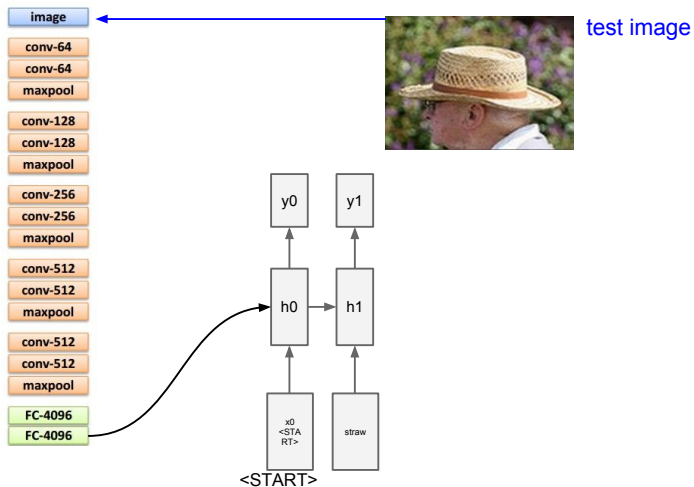
test image

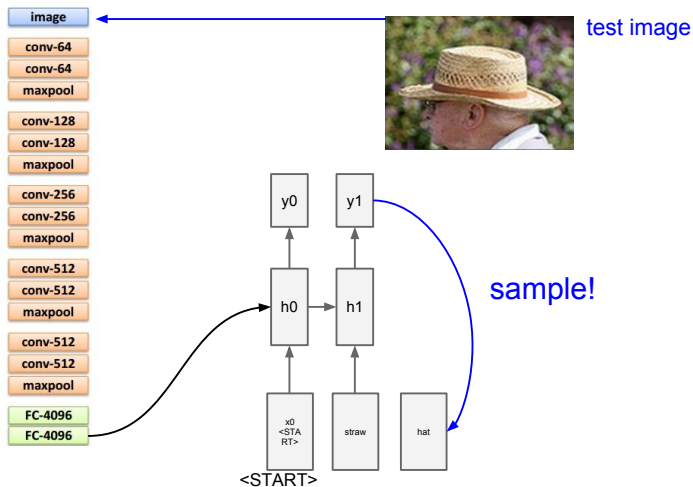


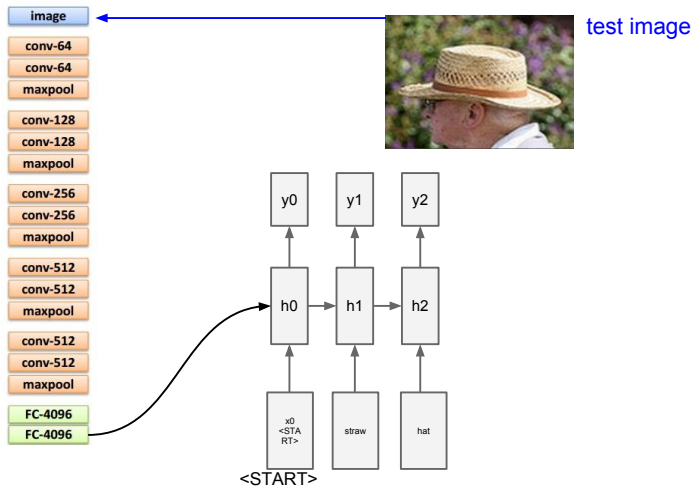












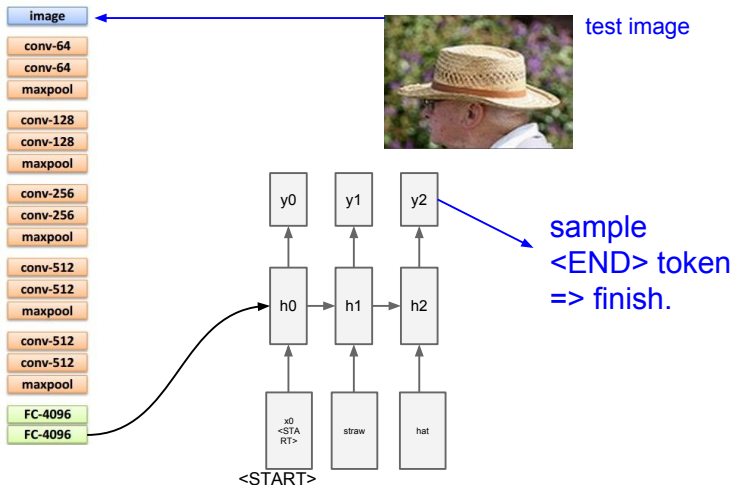


Image Sentence Datasets

a man riding a bike on a dirt path through a forest.
bicyclist raises his fist as he rides on desert dirt trail.
this dirt bike rider is smiling and raising his fist in triumph.
a man riding a bicycle while pumping his fist in the air.
a mountain biker pumps his fist in celebration.



Microsoft COCO

[*Tsung-Yi Lin et al. 2014*]

mscoco.org

currently:

~120K images

~5 sentences each



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"a young boy is holding a baseball bat."



"a cat is sitting on a couch with a remote control."



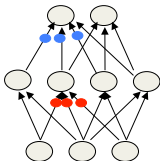
"a woman holding a teddy bear in front of a mirror."



"a horse is standing in the middle of a road."

The key idea of echo state networks (perceptrons again?)

- A very simple way to learn a feedforward network is to make the early layers random and fixed.
- Then we just learn the last layer which is a linear model that uses the transformed inputs to predict the target outputs.
 - A big random expansion of the input vector can help.



- The equivalent idea for RNNs is to fix the **input→hidden** connections and the **hidden→hidden** connections at random values and only learn the **hidden→output** connections.
 - The learning is then very simple (assuming linear output units).
 - Its important to set the random connections very carefully so the RNN does not explode or die.

How to set random connections in echo state networks

- Set the hidden→hidden weights so that the intensity of activity stays about the same after each iteration
 - Set the largest eigenvalue to 1
 - This allows the input to echo around the network for a long time
- Use sparse connectivity (i.e. set most of the weights to zero)
 - This creates lots of loosely coupled oscillators
- Choose the scale of the input→hidden connections very carefully
 - They need to drive the loosely coupled oscillators without wiping out the information from the past that they already contain
- The learning is so fast that we can try many different scales for the input→hidden weights and sparsenesses
 - This is often necessary

How to set random connections in echo state networks

- Set the hidden→hidden weights so that the intensity of activity stays about the same after each iteration
 - Set the largest eigenvalue to 1
 - This allows the input to echo around the network for a long time
- Use sparse connectivity (i.e. set most of the weights to zero)
 - This creates lots of loosely coupled oscillators
- Choose the scale of the input→hidden connections very carefully
 - They need to drive the loosely coupled oscillators without wiping out the information from the past that they already contain
- The learning is so fast that we can try many different scales for the input→hidden weights and sparsenesses
 - This is often necessary

How to set random connections in echo state networks

- Set the hidden→hidden weights so that the intensity of activity stays about the same after each iteration
 - Set the largest eigenvalue to 1
 - This allows the input to echo around the network for a long time
- Use sparse connectivity (i.e. set most of the weights to zero)
 - This creates lots of loosely coupled oscillators
- Choose the scale of the input→hidden connections very carefully
 - They need to drive the loosely coupled oscillators without wiping out the information from the past that they already contain
- The learning is so fast that we can try many different scales for the input→hidden weights and sparsenesses
 - This is often necessary

How to set random connections in echo state networks

- Set the hidden→hidden weights so that the intensity of activity stays about the same after each iteration
 - Set the largest eigenvalue to 1
 - This allows the input to echo around the network for a long time
- Use sparse connectivity (i.e. set most of the weights to zero)
 - This creates lots of loosely coupled oscillators
- Choose the scale of the input→hidden connections very carefully
 - They need to drive the loosely coupled oscillators without wiping out the information from the past that they already contain
- The learning is so fast that we can try many different scales for the input→hidden weights and sparsenesses
 - This is often necessary

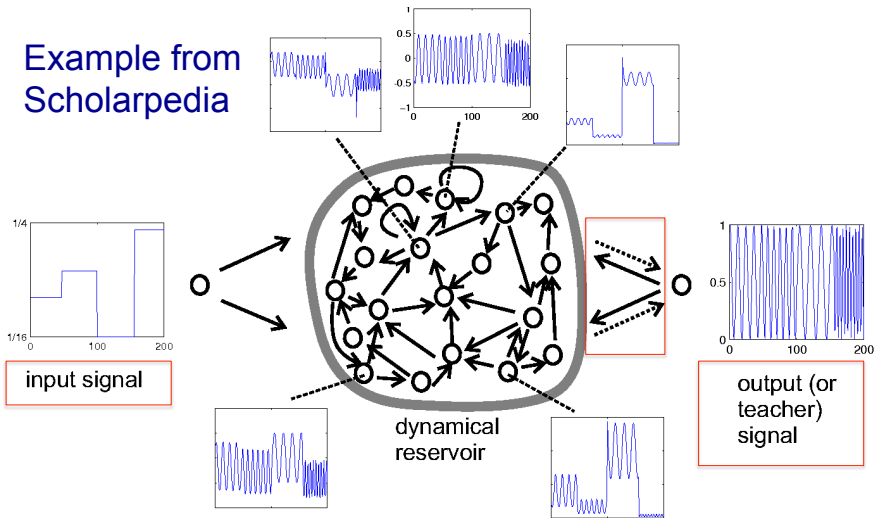
A simple example of an echo state network

INPUT SEQUENCE A real-valued time-varying value that specifies the frequency of a sine wave

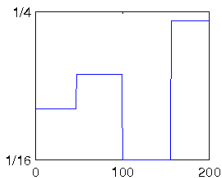
TARGET OUTPUT SEQUENCE A sine wave with the currently specified frequency

LEARNING METHOD Fit a linear model that takes the states of the hidden units as input and produces a single scalar output

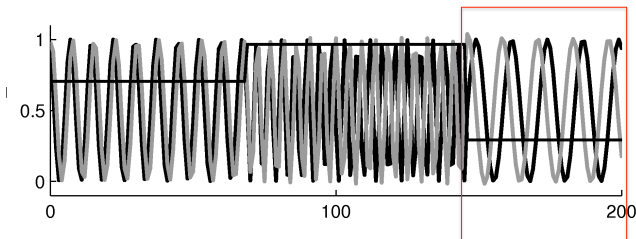
Example from Scholarpedia



The target and predicted outputs after learning



input signal



Beyond echo state networks

- Good aspects of ESNs: Echo state networks can be trained very fast because they just fit a linear model
- They demonstrate that it is very important to initialize weights sensibly
- They can do impressive modeling of one-dimensional time-series
 - but they cannot compete seriously for high-dimensional data like pre-processed speech
- Bad aspects of ESNs: They need many more hidden units for a given task than an RNN that learns the hidden→hidden weights
- Ilya Sutskever (2012) has illustrated that if the weights are initialized using the ESN methods, RNNs could be trained very effectively
 - He uses rmsprop with momentum

Beyond echo state networks

- Good aspects of ESNs: Echo state networks can be trained very fast because they just fit a linear model
- They demonstrate that it is very important to initialize weights sensibly
- They can do impressive modeling of one-dimensional time-series
 - but they cannot compete seriously for high-dimensional data like pre-processed speech
- Bad aspects of ESNs: They need many more hidden units for a given task than an RNN that learns the hidden→hidden weights
- Ilya Sutskever (2012) has illustrated that if the weights are initialized using the ESN methods, RNNs could be trained very effectively
 - He uses rmsprop with momentum

Beyond echo state networks

- Good aspects of ESNs: Echo state networks can be trained very fast because they just fit a linear model
- They demonstrate that it is very important to initialize weights sensibly
- They can do impressive modeling of one-dimensional time-series
 - but they cannot compete seriously for high-dimensional data like pre-processed speech
- Bad aspects of ESNs: They need many more hidden units for a given task than an RNN that learns the hidden→hidden weights
- Ilya Sutskever (2012) has illustrated that if the weights are initialized using the ESN methods, RNNs could be trained very effectively
 - He uses rmsprop with momentum

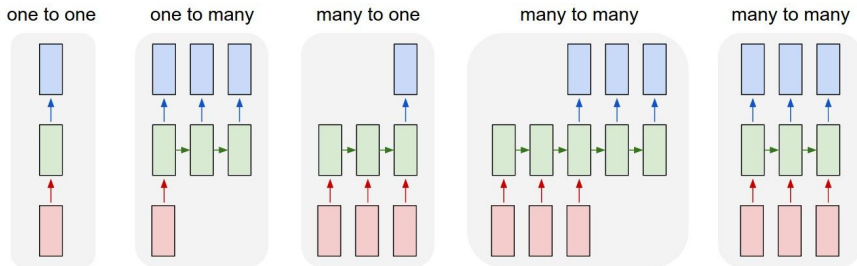
Beyond echo state networks

- Good aspects of ESNs: Echo state networks can be trained very fast because they just fit a linear model
- They demonstrate that it is very important to initialize weights sensibly
- They can do impressive modeling of one-dimensional time-series
 - but they cannot compete seriously for high-dimensional data like pre-processed speech
- Bad aspects of ESNs: They need many more hidden units for a given task than an RNN that learns the hidden→hidden weights
- Ilya Sutskever (2012) has illustrated that if the weights are initialized using the ESN methods, RNNs could be trained very effectively
 - He uses rmsprop with momentum

Conclusions

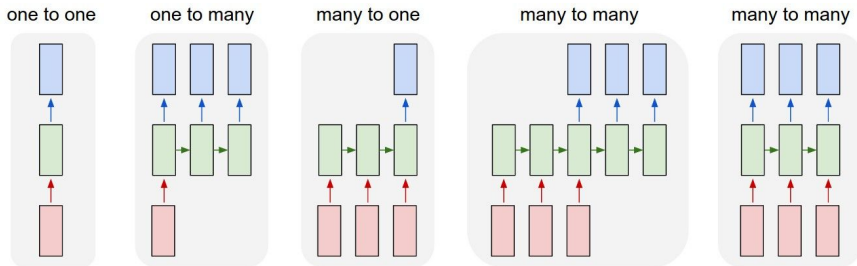
- RNNs allow a lot of flexibility in architecture design and have many applications

Recurrent Networks offer a lot of flexibility:



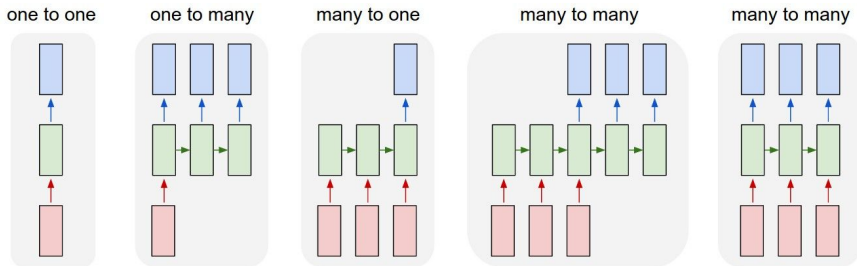
Vanilla Neural Networks

Recurrent Networks offer a lot of flexibility:



e.g. **Image Captioning**
image -> sequence of words

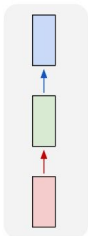
Recurrent Networks offer a lot of flexibility:



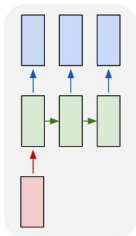
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Networks offer a lot of flexibility:

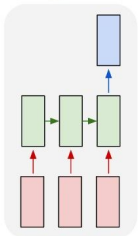
one to one



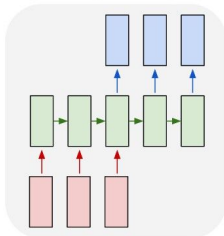
one to many



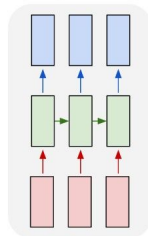
many to one




many to many

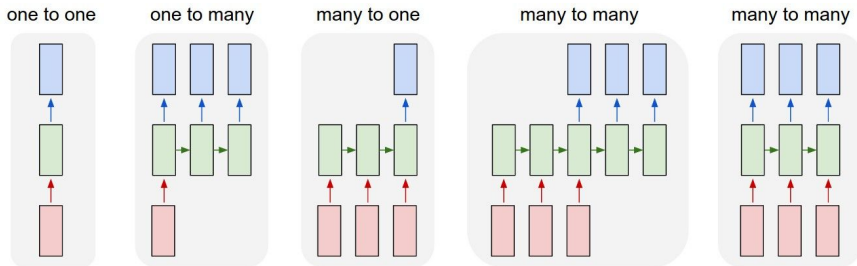


many to many




 e.g. **Machine Translation**
 seq of words \rightarrow seq of words

Recurrent Networks offer a lot of flexibility:



e.g. **Video classification on frame level**

Conclusions

- RNNs allow a lot of flexibility in architecture design and have many applications
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better optimization techniques such as Hessian-free methods could be used to avoid gating structures like LSTM
- Echo state networks are another possibility but may not work very well for high dimensional inputs

Conclusions

- RNNs allow a lot of flexibility in architecture design and have many applications
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better optimization techniques such as Hessian-free methods could be used to avoid gating structures like LSTM
- Echo state networks are another possibility but may not work very well for high dimensional inputs

Conclusions

- RNNs allow a lot of flexibility in architecture design and have many applications
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better optimization techniques such as Hessian-free methods could be used to avoid gating structures like LSTM
- Echo state networks are another possibility but may not work very well for high dimensional inputs

Conclusions

- RNNs allow a lot of flexibility in architecture design and have many applications
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better optimization techniques such as Hessian-free methods could be used to avoid gating structures like LSTM
- Echo state networks are another possibility but may not work very well for high dimensional inputs

Conclusions

- RNNs allow a lot of flexibility in architecture design and have many applications
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better optimization techniques such as Hessian-free methods could be used to avoid gating structures like LSTM
- Echo state networks are another possibility but may not work very well for high dimensional inputs