Week 9: Recurrent Neural Networks

Nazli Ikizler Cinbis

\[
\begin{align*}
\mathcal{E}_{t-1} & \quad \frac{\partial \mathcal{E}_{t-1}}{\partial \mathbf{x}_{t-1}} \\
\mathbf{x}_{t-1} & \quad \frac{\partial \mathbf{x}_{t}}{\partial \mathbf{x}_{t-1}} \\
u_{t-1} & \\
\mathcal{E}_t & \quad \frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \\
\mathbf{x}_t & \quad \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \\
u_t & \\
\mathcal{E}_{t+1} & \quad \frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{x}_{t+1}} \\
\mathbf{x}_{t+1} & \quad \frac{\partial \mathbf{x}_{t+2}}{\partial \mathbf{x}_{t+1}} \\
u_{t+1} & 
\end{align*}
\]

Pascanu et al.
Recurrent Neural Networks

Multi-layer Perceptron

Output Layer

Hidden Layers

Input Layer

Recurrent Network
Recurrent Networks offer a lot of flexibility:

Vanilla Neural Networks
Recurrent Networks offer a lot of flexibility:

- One to one
- One to many
- Many to one
- Many to many

E.g., Image Captioning: image -> sequence of words
Recurrent Networks offer a lot of flexibility:

- One to one
- One to many
- Many to one
- Many to many

Examples:
- 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
- e.g. Machine Translation
  seq of words -> seq of words
Recurrent Networks offer a lot of flexibility:

e.g. Video classification on frame level

Slide by Fei-Fei Li & Andrej Karpathy & Justin Johnson
Sequential Processing of fixed inputs

Multiple Object Recognition with Visual Attention, Ba et al.
Sequential Processing of fixed outputs

DRAW: A Recurrent Neural Network For Image Generation, Gregor et al.
Recurrent Neural Network
Recurrent Neural Network

usually want to predict a vector at some time steps

\( x \)

\( y \)
A recurrent neural network can be thought of as *multiple copies of the same network, each passing a message to a successor*. Consider what happens when we unroll the loop:
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
- old state
- some function with parameters $W$
- input vector at some time step

$$ y $$

$$ x $$
Recurrent Neural Network

We can process a sequence of vectors $x$ 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.
(Vanilla) Recurrent Neural Network

The state consists of a single "hidden" vector $h$:

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

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

$$y_t = W_{hy}h_t$$
Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: “hello”
Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: “hello”
Character-level language model example

Vocabulary: [h, e, l, o]

Example training sequence: “hello”
Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: “hello”
Backpropagation Through Time (BPTT)

The recurrent model is represented as a multi-layer one (with an unbounded number of layers) and backpropagation is applied on the unrolled model.
Backpropagation Through Time (BPTT)

\[
\frac{\partial E}{\partial \theta} = \sum_{1 \leq t \leq T} \frac{\partial E_t}{\partial \theta}
\]

\[
\frac{\partial E_t}{\partial \theta} = \sum_{1 \leq k \leq t} \left( \frac{\partial E_t}{\partial x_t} \frac{\partial x_t}{\partial x_k} \frac{\partial^+ x_k}{\partial \theta} \right)
\]

\[
\frac{\partial x_t}{\partial x_k} = \prod_{t \geq i > k} \frac{\partial x_i}{\partial x_{i-1}} = \prod_{t \geq i > k} W_{rec}^T \text{diag}(\sigma'(x_{i-1}))
\]
min-char-rnn.py gist: 112 lines of Python

(https://gist.github.com/karpathy/d4dee566867f8291f086)
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)

BSD License

```python
import numpy as np

# data I/O

data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }
```
Initializations

```python
# hyperparameters
hidden_size = 100  # size of hidden layer of neurons
seq_length = 25  # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01  # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01  # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01  # hidden to output
bh = np.zeros((hidden_size, 1))  # hidden bias
by = np.zeros((vocab_size, 1))  # output bias
```
n, p = 0, 0
mwxh, mwhh, mwhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mbv = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
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 >= 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
        ----
        % (txt, )
    # forward seq_length characters through the net and fetch gradient
    loss, dwxh, dwhh, dwhy, dbh, dbv = lossFun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.999 + 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, Why, bh, by],
                                   [dwxh, dwhh, dwhy, dbh, dbv],
                                   [mwxh, mwhh, mwhy, mbh, mbv]):
        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

```python
n, p = 0, 0
mwxh, mhwh, mwhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by)  # memory variables for Adagrad
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
' % (txt,)

    # forward seq_length characters through the net and fetch gradient
    loss, dwxh, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.999 + 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, Why, bh, by],
                                   [dwxh, dwhh, dwhy, dbh, dby],
                                   [mwxh, mhwh, 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

```python
n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mbv = np.zeros_like(bh), np.zeros_like(by)  # memory variables for Adagrad
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('-----
%10d %n %d %

# forward seq_length characters through the net and fetch gradient
loss, dWxh, dWhh, dWhy, dbh, dbv, hprev = lossFun(inputs, targets, hprev)
smooth_loss = smooth_loss * 0.999 + 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, Why, bh, by],
                                 [dWxh, dWhh, dWhy, dbh, dbv],
                                 [mWxh, mWhh, mWhy, mbh, mbv]):
    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

```
  n, p = 0, 0
  mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
  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 >= 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 %d ----' % (txt, )

    # forward seq_length characters through the net and fetch gradient
    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
    smooth_loss += smooth_loss * 0.999 + 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, Why, bh, by],
                                   [dWxh, 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

```python
n, p = 0, @
mwxh, mwhh, mwhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mbv = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration @
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 = @ # 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 '-----
    %s
    %s
            %s
            %s
    %s' % (txt, )

    # forward seq_length characters through the net and fetch gradient
    loss, dwxh, dwhh, dwhy, dbh, dbv, hprev = lossFun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.999 + 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, Why, bh, by],
                                   [dwxh, dwhh, dwhy, dbh, dbv],
                                   [mwxh, mwhh, mwhy, mbh, mbv]):
        mem += dparam * dparam
        param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

    p += seq_length # move data pointer
    n += 1 # iteration counter
```
Loss function
- forward pass (compute loss)
- backward pass (compute param gradient)
def lossFun(inputs, targets, hprev):
    """
    inputs, targets are both list of integers. 
hprev is Hx1 array of initial hidden state
    returns the loss, gradients on model parameters, and last hidden state
    """
    xs, hs, ys, ps = [], [], [], []
    hs[-1] = np.copy(hprev)
    loss = 0
    # forward pass
    for t in xrange(len(inputs)):
        if inputs[t] == 0:
            continue
        xs[t] = np.zeros((vocab_size, 1)) # encode in 1-of-k representation
        xs[t][inputs[t]] = 1
        hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
        ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
        ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
        loss += -np.log(ps[t][targets[t], 0]) # softmax (cross-entropy loss)

    return loss, grads, hprev

ht = tanh(Whh*ht-1 + Wxh*x_t)
y_t = Why*ht
# backward pass: compute gradients going backwards

dWhx, dWhh, dWhy = np.zeros_like(Whx), np.zeros_like(Whh), np.zeros_like(Why)
dbh, dby = np.zeros_like(bh), np.zeros_like(by)
dhnext = np.zeros_like(hs[0])

for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1  # backprop into y
    dwhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext  # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh  # backprop through tanh nonlinearity
    dbh += dhraw
    dWhx += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)

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

return loss, dWhx, dWhh, dwhy, dbh, dby, hs[len(inputs)-1]
def sample(h, seed_ix, n):
    """
    sample a sequence of integers from the model
    h is memory state, seed_ix is seed letter for first time step
    """
    x = np.zeros((vocab_size, 1))
    x[seed_ix] = 1
    ixes = []
    for t in xrange(n):
        h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
        y = np.dot(Why, h) + by
        p = np.exp(y) / np.sum(np.exp(y))
        ix = np.random.choice(range(vocab_size), p=p.ravel())
        x = np.zeros((vocab_size, 1))
        x[ix] = 1
        ixes.append(ix)
    return ixes
A figure illustrating a Recurrent Neural Network (RNN) structure. The input is denoted by X, and the output is denoted by Y. The RNN is shown processing both X and Y, indicating a feedback loop within the network.
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:

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

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 ofter.

"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

<table>
<thead>
<tr>
<th>Part</th>
<th>Chapter</th>
<th>online</th>
<th>TeX source</th>
<th>view pdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preliminaries</td>
<td>1. Introduction</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>2. Conventions</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>3. Set Theory</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>4. Categories</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>5. Topology</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>6. Sheaves on Spaces</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>7. Sites and Sheaves</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>8. Stacks</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>9. Fields</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
<tr>
<td></td>
<td>10. Commutative Algebra</td>
<td>online</td>
<td>tex</td>
<td>pdf</td>
</tr>
</tbody>
</table>

Parts

1. Preliminaries
2. Schemes
3. Topics in Scheme Theory
4. Algebraic Spaces
5. Topics in Geometry
6. Deformation Theory
7. Algebraic Stacks
8. Miscellany

Statistics

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

Latex source
For $\mathcal{L}_{m, \ast} = 0$, hence we can find a closed subset $\mathcal{H} \subset \mathcal{H}$ and any sets $F$ on $X$, $U$ is a closed immersion of $S$, then $U \to T$ is a separated algebraic space.

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

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

and the comparably in the fibre product covering we have to prove the lemma generated by $\coprod \mathbb{Z} \times x \to V$. Consider the maps $M$ along the set of points $S_{\text{proj}}$ and $U \to U$ 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 $\text{Spec}(R) \to S$ 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 $\mathcal{O}_{x_i, x'}$ is a scheme where $x, x', x'' \in S'$ such that $O_{x, x'} \to O_{x', x''}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $GL_{\mathbb{C}}(x'/S'$) and we win.

To prove we study that $\mathcal{F} \subset \mathcal{F}_x$ for $i > 0$ and $\mathcal{F}_x$ exists and let $\mathcal{F}_x$ be a presheaf of $\mathcal{O}_{X_\mathcal{S}}$-modules on $C$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\mathcal{M} = \mathcal{F} \otimes_{\text{Spec}(k)} \mathcal{O}_{x_\mathcal{S}} = \mathcal{I}_{\mathcal{F}}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)^{\text{proj}}_{\text{proj}} \times (\text{Sch}/S)^{\text{proj}}_{\text{proj}}$$

and

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

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.

The result for prove any open covering follows from the less of Example ???. It may replace $S$ by $X_{\text{spaces, t}}$ which gives an open subspace of $X$ and $T$ equal to $S_{\text{zar}}$, 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 $\text{Proj} \mapsto \text{Spec}(B)$ over $U$ compatible with the complex

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

When in this case of to show that $\mathcal{O} \to \mathcal{O}_{X, x_\mathcal{S}}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is then the closed subschemes are etale. If $T$ is surjective we may assume that $T$ is connected with residue 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 subspace 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 form all sheaves of sheaves on $X$. But given a scheme $U$ and a surjective étale morphism $U \to X$. Let $U \cap U = \bigcup_i U_i$ be the scheme $X$ over $S$ at the schemes $U_i \to X$ and $U = \lim_i U_i$.

The following lemma surjective rest decomposes of this implies that $\mathcal{F}_{\mathcal{F}} = \mathcal{F}_{x_\mathcal{S}} = \mathcal{F}_{x_{\text{zar}}}$.

**Lemma 0.2.** Let $X$ be a locally Noetherian scheme over $S$, $E = \mathcal{F}_{\mathcal{F}_{\mathcal{S}}}$, $\mathcal{I} = I_{\mathcal{F}} \subset \mathcal{I}_{\mathcal{F}}$, since $I_{\mathcal{F}} \subset \mathcal{I}_{\mathcal{F}}$ are nonzero over $J_{\mathcal{S}}$ is a subset of $J_{\mathcal{S}}$ works.

**Lemma 0.3.** In Situation ???. Hence we may assume $q' = 0$.

**Proof.** We will use the property we see that $p$ 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 $\delta_{n+1}$ is a scheme over $S$. 


Proof. Omitted.

**Lemma 0.1.** Let $C$ be a set of the construction.
Let $C$ be a gerber covering. Let $F$ be a quasi-coherent sheaves of $O$-modules. We have to show that
\[ O_{O_C} = O_X(L) \]

Proof. This is an algebraic space with the composition of sheaves $F$ on $X_{\text{etale}}$ we have
\[ O_X(F) = \{ \text{morph}_{1} \times O_X (G, F) \} \]
where $G$ defines an isomorphism $F \rightarrow F$ of $O$-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
\[ b : X \rightarrow Y' \rightarrow Y \rightarrow Y' \times_X Y \rightarrow X. \]
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 $F$ be a quasi-coherent sheaf of $O_X$-modules. The following are equivalent

1. $F$ is an algebraic space over $S$.
2. If $X$ is an affine open covering.

Consider a common structure on $X$ and $X$ the functor $O_X(U)$ which is locally of finite type.

This since $F \in F$ and $x \in G$ the diagram

\[ \text{Spec}(K_x) \]

is a limit. Then $G$ is a finite type and assume $S$ is a flat and $F$ and $G$ is a finite type $L$. This is of finite type diagrams, and

* the composition of $G$ is a regular sequence,
* $O_X$ is a sheaf of rings.

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

Proof. This is clear that $G$ is a finite presentation, see Lemmas ??.
A reduced above we conclude that $U$ is an open covering of $C$. The functor $F$ is a “field
\[ O_{X,S} \rightarrow F_T \rightarrow (O_{X,T}) \rightarrow O_{X,T}(O_{X,T}) \]
is an isomorphism of covering of $O_{X,T}$. If $F$ is the unique element of $F$ such that $X$ is an isomorphism.
The property $F$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $O_X$-algebra with $F$ are opens of finite type over $S$. If $F$ is a scheme theoretic image points.

If $F$ is a finite direct sum $O_{X,T}$ is a closed immersion, see Lemma ??. This is a sequence of $F$ is a similar morphism.
Linux kernel source tree

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

Branch: master

- Documentation: Merge git://git.kernel.org/pub/scm/linux/kernel/git/hab/target-pending (6 days ago)
- arch: Merge branch 'x86-urgent-for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/hab... (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/habern/th/decrypt-2.6 (10 days ago)
- drivers: Merge branch 'drm-fixes' of git://people.freedesktop.org/~airilled/l... (9 hours ago)
- firmware: firmware/hex2fw.c: restore missing default in switch statement (2 months ago)
- fs: vfs: read_file_handler only once in handle_to_path (4 days ago)
- include: Merge branch 'perf-urgent-for-linus' of git://git.kernel.org/pub/scm/l... (a day ago)
- init: fix regression by supporting devices with major:minor:offset format (a month ago)
- ip...
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 & 0x00000000fffff8) & 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 ");
}
/*
 * 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/clevent.h>
#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setev.h>
#include <asm/pgproto.h>
```c
#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/sete.w.h>
#include <asm/pgproto.h>

#define REG_PG  vesa_slot_addr_pack
#define PPM_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;
  ```
Searching for interpretable cells

[Visualizing and Understanding Recurrent Networks, Andrej Karpathy*, Justin Johnson*, Li Fei-Fei]
Searching for interpretable cells

"You mean to imply that I have nothing to eat out of... On the contrary, I can supply you with everything even if you want to give dinner parties," warmly replied Chichagov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be animated by the same desire. Kutuzov, shrugging his shoulders, replied with his subtle penetrating smile: "I meant merely to say what I said."
Searching for interpretable cells

Cell sensitive to position in line:
The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action—the one Kutuzov and the general mass of the army demanded—namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all—carried on by vis inertiae—pressed forward into boats and into the ice-covered water and did not, surrender.
Searching for interpretable cells

```c
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask, siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig == -1)
        if (current->notifier)
            if (sigismember(current->notifier_mask, sig))
                if (!current->notifier)(current->notifier_data) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
    collect_signal(sig, pending, info);
    return sig;
}
```

if statement cell
Searching for interpretable cells

```c
/* Duplicate LSM field information. The lsm_rule is opaque, so */
static inline int audit_dupe_lsm_field(struct audit_field *df,
    struct audit_field *sf)
{
    int ret = 0;
    char *lsm_str;
    /* our own copy of lsm_str */
    lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
    if (unlikely(!lsm_str))
        return -ENOMEM;
    df->lsm_str = lsm_str;
    /* our own (refreshed) copy of lsm_rule */
    ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
        (void **)&df->lsm_rule);
    /* Keep currently invalid fields around in case they 
     * become valid after a policy reload. */
    if (ret == -EINVAL) {
        pr_warn("audit rule for LSM '%s' is invalid\n",
            df->lsm_str);
        ret = 0;
    }
    return ret;
}
```
Searching for interpretable cells

```c
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
#endif
```
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
before:

\[ h = \tanh(W_{xh} \ast x + W_{hh} \ast h) \]

now:

\[ h = \tanh(W_{xh} \ast x + W_{hh} \ast h + W_{ih} \ast v) \]
test image

<START>

hat

straw

sample!
test image

sample

<END> token

=> finish.
Image Sentence Datasets

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.”
Preview of fancier architectures

RNN attends spatially to different parts of images while generating each word of the sentence:
RNN:

\[ h^l_t = \tanh(W^l_t(h^{l-1}_t, h^l_{t-1})) \]

\( h \in \mathbb{R}^n \), \( W^l \in [n \times 2n] \)
The problem of long-term dependencies

(Vanilla) RNNs connect previous information to present task:
- enough for predicting the next word for “the clouds are in the sky”
- may not be enough when more context is needed
  “I grew up in France... I speak fluent French.”

Adapted from Christopher Olah
The problem of vanishing gradients

• In a traditional recurrent neural network, during the gradient backpropagation phase, the gradient signal can end up being multiplied a large number of times.

• If the gradients are small
  – Vanishing gradients, learning very slow or stops

• If gradients are large
  – Exploding gradients, learning diverges
The problem of vanishing gradients

• In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.
  – We can avoid this by initializing the weights very carefully.
• Even with good initial weights, it is very hard to detect that the current target output depends on an input from many time-steps ago.
  – So RNNs have difficulty dealing with long-range dependencies.
Understanding gradient flow dynamics


```python
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

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]
Understanding gradient flow dynamics

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

can control exploding with gradient clipping

can control vanishing with LSTM

[On the difficulty of training Recurrent Neural Networks, Pascanu et al., 2013]
Trick for exploding gradient: clipping trick

• The solution first introduced by Mikolov is to clip gradients to a maximum value.

Algorithm 1 Pseudo-code for norm clipping the gradients whenever they explode

\[
\hat{g} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta} \\
\text{if } \|\hat{g}\| \geq \text{threshold } \text{then} \\
\hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|} \hat{g} \\
\text{end if}
\]

• Makes a big difference in RNNs
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).
- 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 et al., 1997]

Memory Cell

- **Input gate** can allow incoming signal to alter the state of the memory cell or block it.
- **Output gate** can allow the state of the memory cell to have an effect on other neurons or prevent it.
- **Forget gate** can modulate the memory cell’s self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.
- **Self-recurrent connection** ensures that, barring any outside interference, the state of a memory cell can remain constant from one timestep to another.
RNN:

$$h_t^l = \tanh W^l (h_{t-1}^{l-1})$$

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

LSTM:

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

\[
\begin{pmatrix}
    i \\
    f \\
    o \\
    g
\end{pmatrix} =
\begin{pmatrix}
    \text{sigmoid} \\
    \text{sigmoid} \\
    \text{sigmoid} \\
    \tanh
\end{pmatrix} W^l (h_{t-1}^{l-1})
\]

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

\[
h_t^l = o \odot \tanh(c_t^l)
\]
Long Short Term Memory (LSTM)
[Hochreiter et al., 1997]

\[
\begin{pmatrix}
i \\
f \\
o \\
g
\end{pmatrix} =
\begin{pmatrix}
\text{sigmoid} \\
\text{sigmoid} \\
\text{sigmoid} \\
tanh
\end{pmatrix}
W^l
\begin{pmatrix}
h_{t-1}^l \\
h_{t-1}^l
\end{pmatrix}
\]

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

\[
h_t^l = o \odot \text{tanh}(c_t^l)
\]
All recurrent neural networks have the form of a chain of repeating modules of neural network

The repeating module in a standard RNN contains a single layer.

Adapted from Christopher Olah
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer there are four, interacting in a very special way.

The repeating module in an LSTM contains four interacting layers.

Adapted from Christopher Olah
The Core Idea Behind LSTMs: Cell State

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

An LSTM has three of these gates, to protect and control the cell state.

Adapted from Christopher Olah
LSTM: Forget gate

It looks at $h_{t-1}$ and $x_t$ and outputs a number between 0 and 1 for each number in the cell state $C_{t-1}$.

A 1 represents “completely keep this” while a 0 represents “completely get rid of this”.

$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

Adapted from Christopher Olah
The next step is to decide what new information we’re going to store in the cell state.

a sigmoid layer called the “**input gate layer**” decides which values we’ll update.

\[ i_t = \sigma (W_i \cdot [h_{t-1}, x_t] + b_i) \]

a tanh layer creates a vector of new candidate values, that could be added to the state.

\[ \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \]
LSTM: Input gate and Cell State

It’s now time to update the old cell state into the new cell state.

\[ C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \]

We multiply the old state by \( f_t \) forgetting the things we decided to forget earlier.

Then, we add the new candidate values, scaled by how much we decided to update each state value.

Adapted from Christopher Olah
Finally, we need to decide what we’re going to output.

First, we run a sigmoid layer which decides what parts of the cell state we’re going to output.

$$o_t = \sigma \left( W_o \left[ h_{t-1}, x_t \right] + b_o \right)$$

Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$h_t = o_t \ast \text{tanh} \left( C_t \right)$$
LSTM variants: peephole connections

Introduced in Gers & Schmidhuber (2000), where we let the gate layers look at the cell state.

\[
\begin{align*}
  f_t &= \sigma \left( W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f \right) \\
  i_t &= \sigma \left( W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i \right) \\
  o_t &= \sigma \left( W_o \cdot [C_t, h_{t-1}, x_t] + b_o \right)
\end{align*}
\]

Adapted from Christopher Olah
LSTM variants: Gated Recurrent Unit (GRU)

Introduced by Cho et al. (2014) It combines the forget and input gates into a single “update gate.” It also merges the cell state and hidden state, and more.

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

Adapted from Christopher Olah
The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby mitigating the vanishing gradient problem.

As long as the input gate remains closed, the activation of the cell will not be overwritten by the new inputs arriving in the network, and can be made available to the net much later in the sequence.
LSTM variants and friends

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

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

• Watch out for overfitting
• Regularization helps: regularization methods include l1, l2, and dropout among others.
• The larger the network, the more powerful, but it’s also easier to overfit. Don’t want to try to learn a million parameters from 10,000 examples – parameters > examples = trouble.
• More data is almost always better
• Train over multiple epochs (complete passes through the dataset).
• The learning rate is the single most important hyperparameter. Tune this using http://cs231n.github.io/neural-networks-3/#baby
LSTM Hyperparameter Tuning

• In general, stacking layers can help.
• For LSTMs, use the softsign (not softmax) activation function over tanh (it’s faster and less prone to saturation (~0 gradients)).
• Updaters: RMSProp, AdaGrad or momentum are usually good choices. AdaGrad also decays the learning rate, which can help sometimes.
• Finally, remember data normalization, MSE loss function + identity activation function for regression.
Bi-directional Recurrent Neural Networks (BRNN)

BRNNs process the data in both directions with two separate hidden layers:
- **Forward hidden sequence**: iterates from t=T:1
- **Backward hidden sequence**: iterates from t=1:T

Adapted from Alex Graves
BRNN forward pass

\[
\text{for } t = 1 \text{ to } T \text{ do }
\]
Forward pass for the forward hidden layer, storing activations at each timestep

\[
\text{for } t = T \text{ to } 1 \text{ do }
\]
Forward pass for the backward hidden layer, storing activations at each timestep

\[
\text{for all } t, \text{ in any order do }
\]
Forward pass for the output layer, using the stored activations from both hidden layers

**Algorithm 3.1: BRNN Forward Pass**
for all $t$, in any order do
    Backward pass for the output layer, storing $\delta$ terms at each timestep
for $t = T$ to 1 do
    BPTT backward pass for the forward hidden layer, using the stored $\delta$ terms from the output layer
for $t = 1$ to $T$ do
    BPTT backward pass for the backward hidden layer, using the stored $\delta$ terms from the output layer

Algorithm 3.2: BRNN Backward Pass
Applications:
Multi-label image classification

Wang et al CVPR 2016
Applications: Segmentation

Zheng et al ICCV 2015
Applications: Videos to Natural Text

Input Video → Convolutional Net → Recurrent Net → Output

Donahue et al. CVPR 2015, Venugopalan et al. NAACL-HLT 2015
Summary

- RNNs allow a lot of flexibility in architecture design
- 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/simpler architectures are a hot topic of current research
- Better understanding (both theoretical and empirical) is needed.
References

- Blog: http://colah.github.io/posts/2015-08-Understanding-LSTMs/