

Lecture 9: Visualizing CNNs and Recurrent Neural Networks

Tuesday February 28, 2017

Announcements!

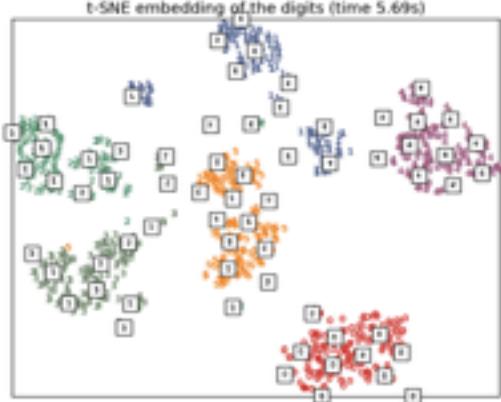
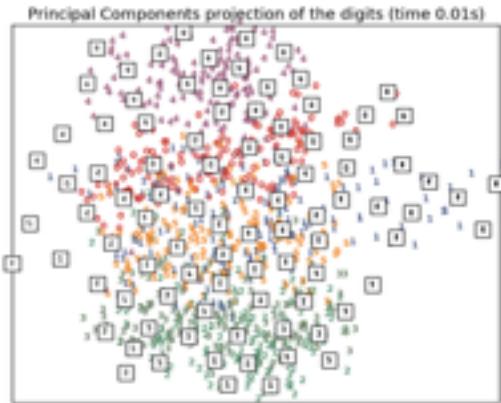
- HW #3 is out
- Final Project proposals due this **Thursday March 2**
- Papers to read: Students should read all papers on the **Schedule** tab, and are encouraged to read as many papers as possible from the **Papers** tab.
- Next paper: **March 7** *You Only Look Once: Unified, Real-Time Object Detection*. If this paper seems too deep or confusing, look at *Fast R-CNN*, *Faster R-CNN*

Python/Numpy of the Day

- t-SNE (t-Distributed Stochastic Nearest Neighbor Embedding)
 - [Scikit-Learn t-SNE](#)
 - [Examples of 2D Embedding Visualizations of MNIST dataset](#)
 - [Other Embedding functions in Scikit-Learn](#)

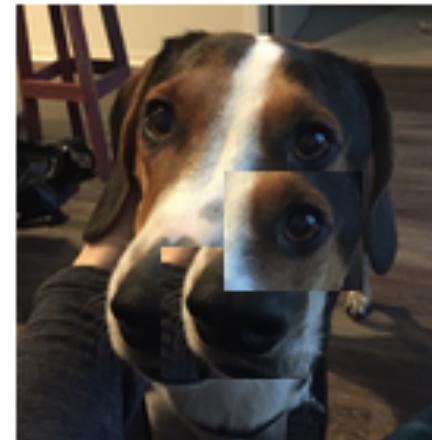
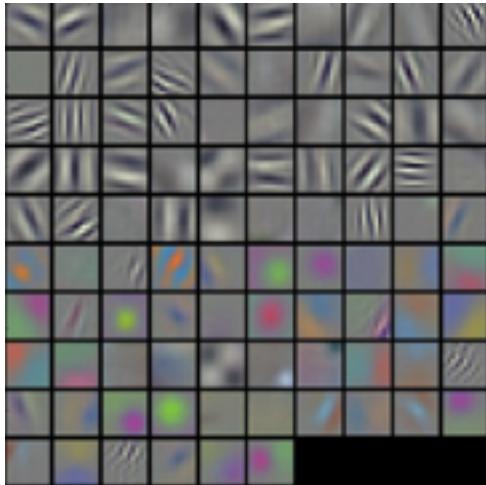
A selection from the 64-dimensional digits dataset

0	1	2	3	4	5	6	7	8	9
5	9	0	4	1	3	8	5	0	1
4	4	1	5	0	5	2	4	0	3
3	4	4	0	5	7	4	2	1	3
2	1	4	5	0	5	0	2	3	4
0	4	4	4	5	0	0	1	4	4
3	5	0	3	2	4	2	4	4	4
4	3	2	2	1	0	4	2	2	3
5	0	4	4	4	4	4	3	5	0
1	1	2	3	4	5	6	7	8	9
2	0	0	1	2	3	4	5	6	7
3	1	2	3	4	5	6	7	8	9
4	2	3	4	5	6	7	8	9	0
5	3	4	5	6	7	8	9	0	1
6	4	5	6	7	8	9	0	1	2
7	5	6	7	8	9	0	1	2	3
8	6	7	8	9	0	1	2	3	4
9	7	8	9	0	1	2	3	4	5



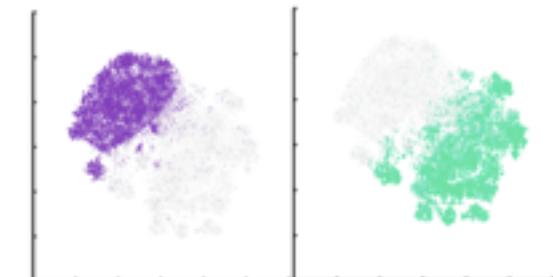
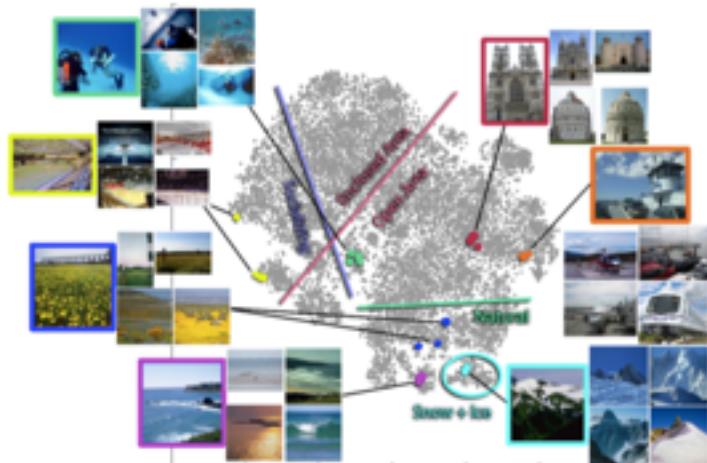
Visualizing CNN Behavior

- How can we see what's going on in a CNN?
- *Stuff we've already done:*
 - Visualize the weights
 - Occlusion experiments — ex. Jason and Lisa's AlexNet Occlusion Tests



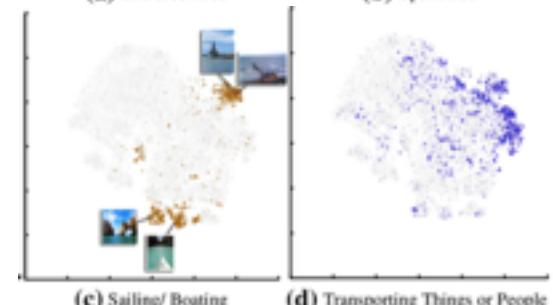
Visualizing CNN Behavior

- How can we see what's going on in a CNN?
- *Straightforward stuff to try in the future:*
 - Visualize the representation space (e.g. with t-SNE)
 - Human experiment comparisons



(a) Enclosed Area

(b) Open Area

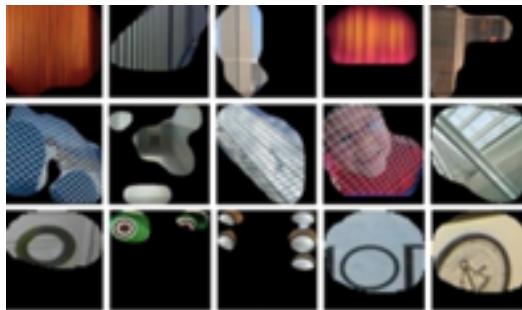


(c) Sailing/ Boating

(d) Transporting Things or People

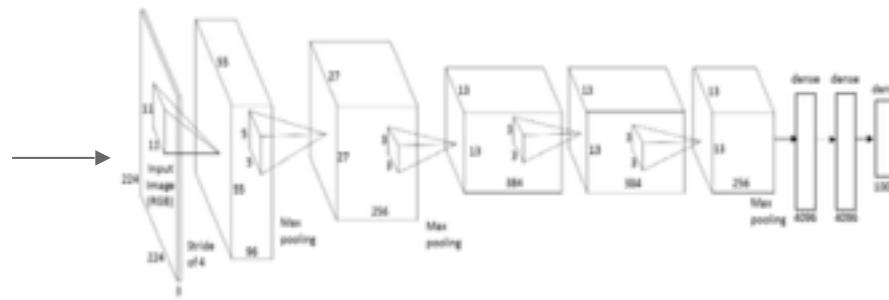
Visualizing CNN Behavior

- How can we see what's going on in a CNN?
- *More sophisticated approaches (HW #4)*
 - Visualize patches that maximally activate neurons
 - Optimization over image approaches (optimization)
 - Deconv approaches (single backward pass)



Deconv approaches - projecting backward from one neuron to see what is activating it

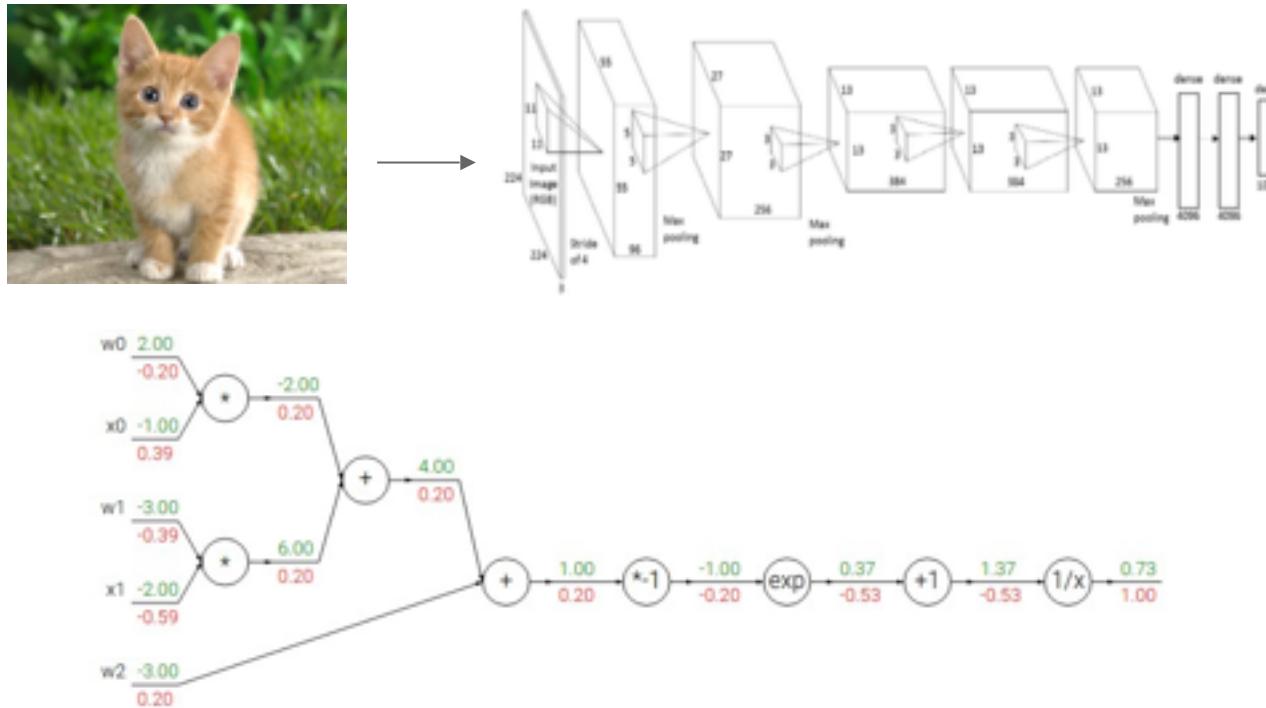
1. Feed image into net



*Q: how can we compute the gradient of any arbitrary
neuron in the network w.r.t. the image?*

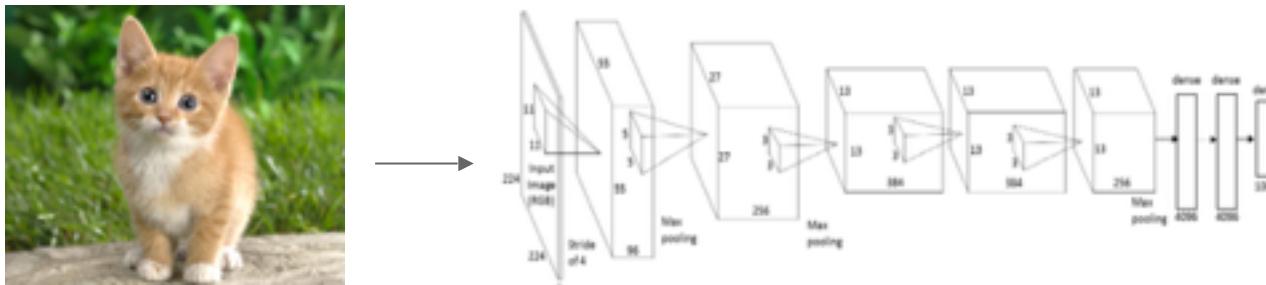
Deconv approaches

1. Feed image into net



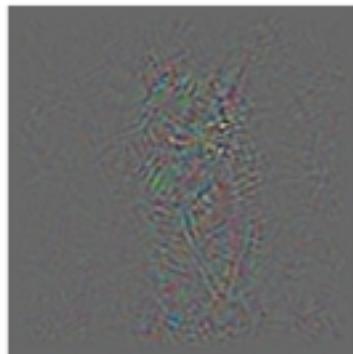
Deconv approaches

1. Feed image into net



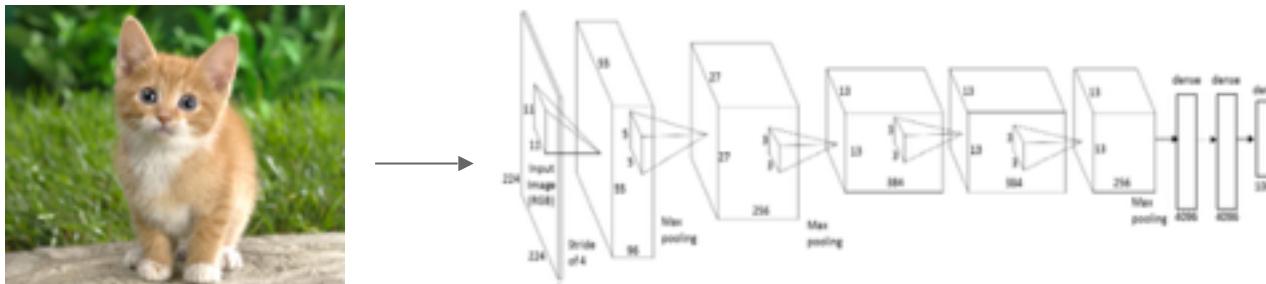
2. Pick a layer, set the gradient there to be all zero except for one 1 for some neuron of interest

3. Backprop to image:

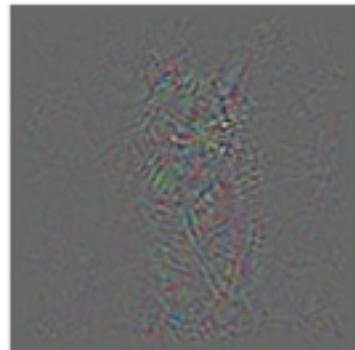


Deconv approaches

1. Feed image into net



2. Pick a layer, set the gradient there to be all zero except for one 1 for some neuron of interest
3. Backprop to image:



**“Guided
backpropagation:”**
only propagate
positive gradients

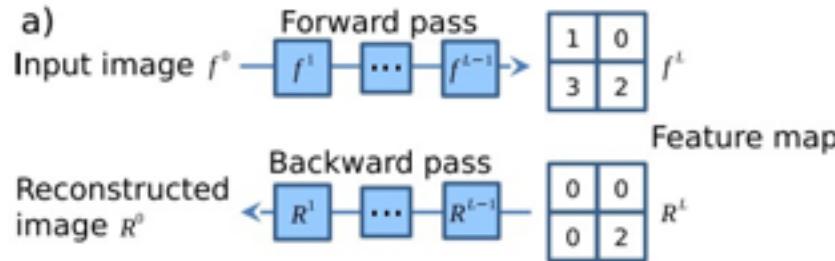


Deconv approaches

[Visualizing and Understanding Convolutional Networks, Zeiler and Fergus 2013]

[Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Simonyan et al., 2014]

[Striving for Simplicity: The all convolutional net, Springenberg, Dosovitskiy, et al., 2015]

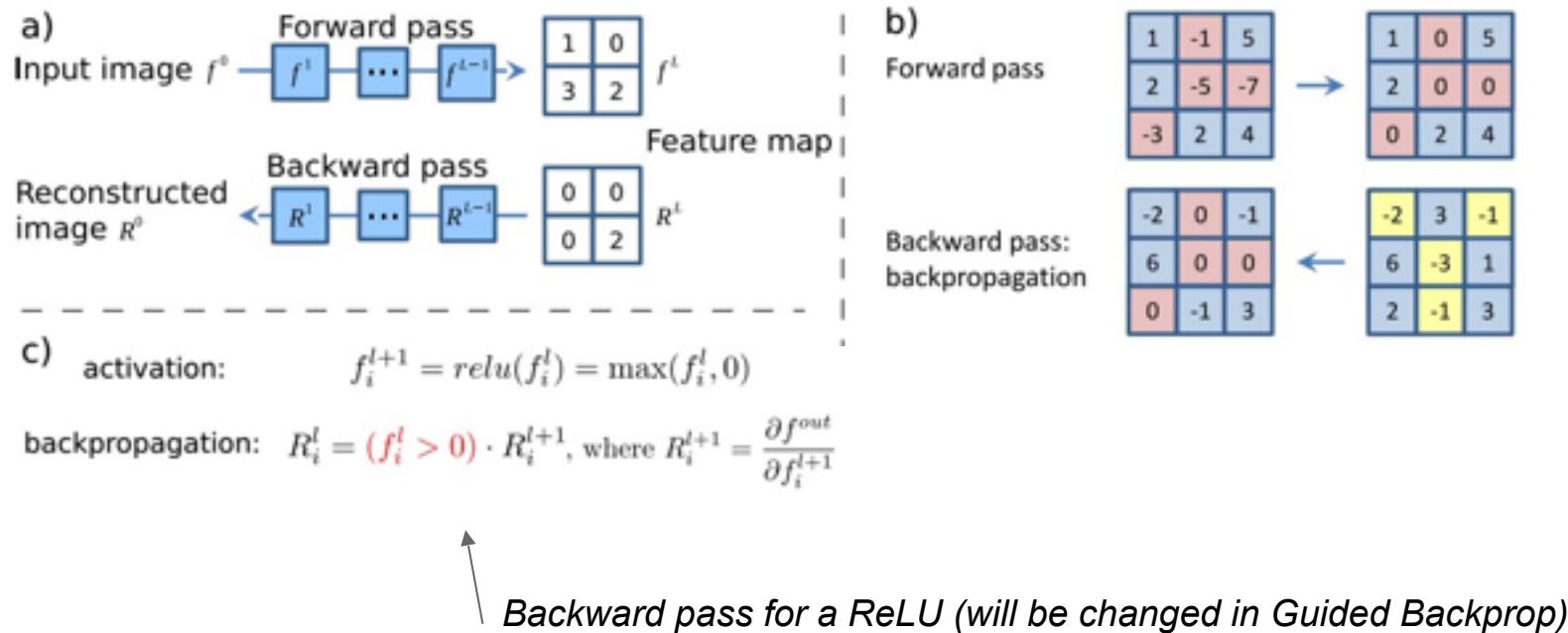


Deconv approaches

[Visualizing and Understanding Convolutional Networks, Zeiler and Fergus 2013]

[Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Simonyan et al., 2014]

[Striving for Simplicity: The all convolutional net, Springenberg, Dosovitskiy, et al., 2015]

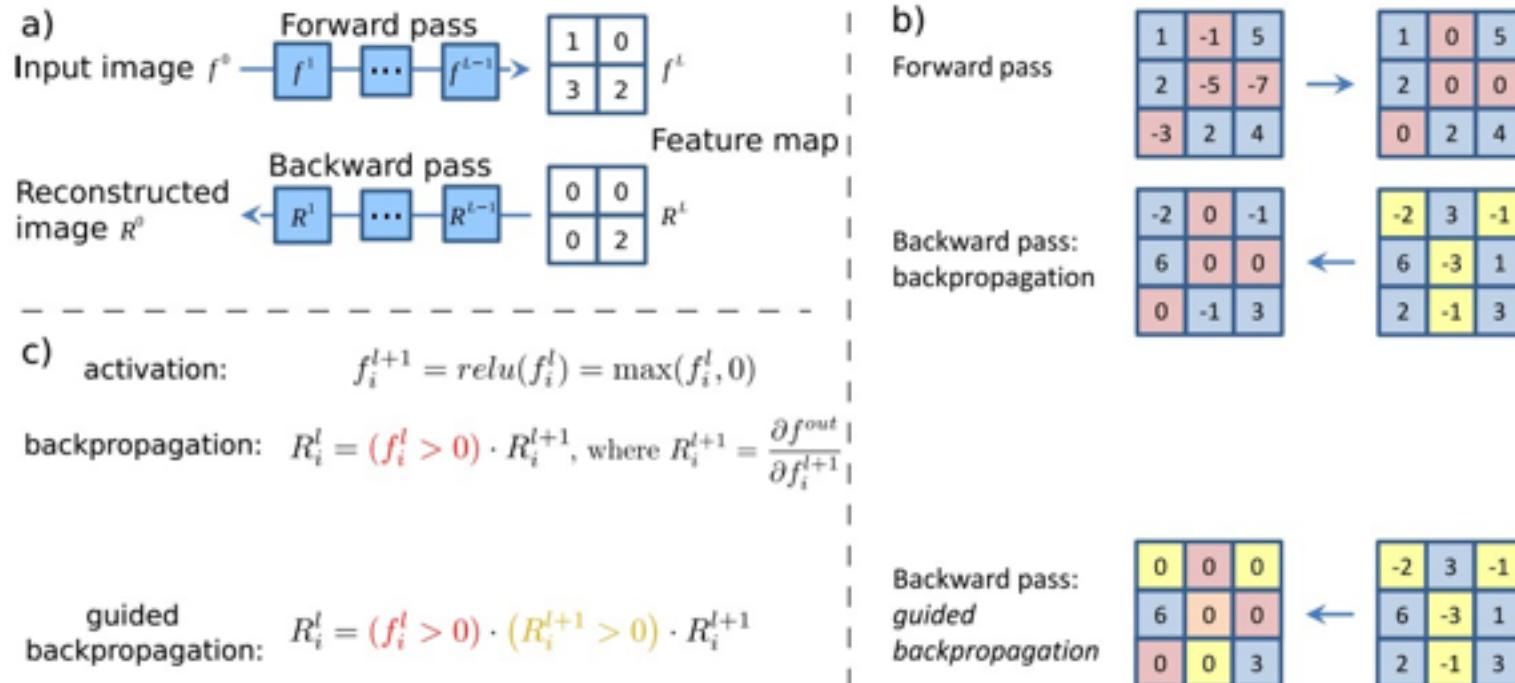


Deconv approaches

[Visualizing and Understanding Convolutional Networks, Zeiler and Fergus 2013]

[Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Simonyan et al., 2014]

[Striving for Simplicity: The all convolutional net, Springenberg, Dosovitskiy, et al., 2015]

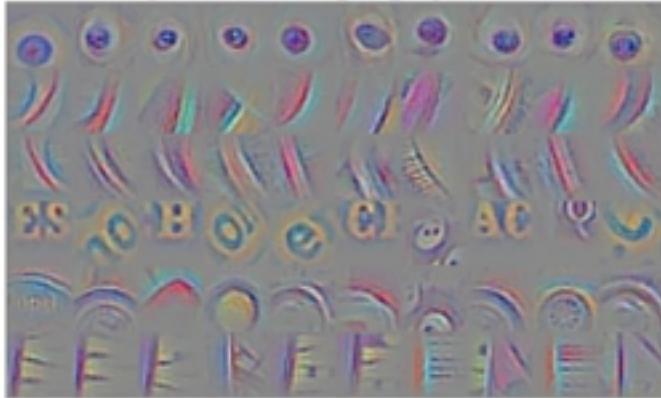


*Visualization of patterns learned by the layer **conv6** (top) and layer **conv9** (bottom) of the network trained on ImageNet.*

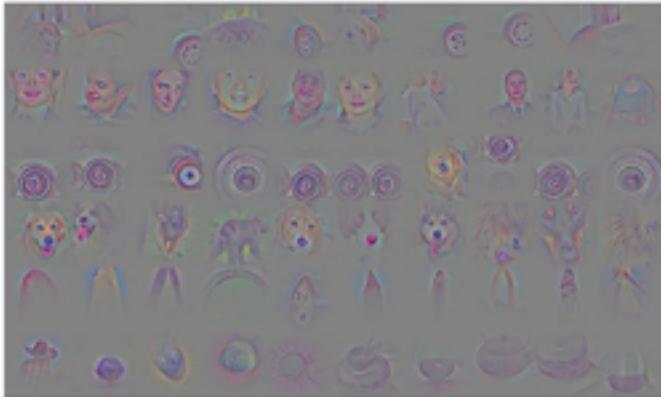
Each row corresponds to one filter.

The visualization using “guided backpropagation” is based on the top 10 image patches activating this filter taken from the ImageNet dataset.

guided backpropagation



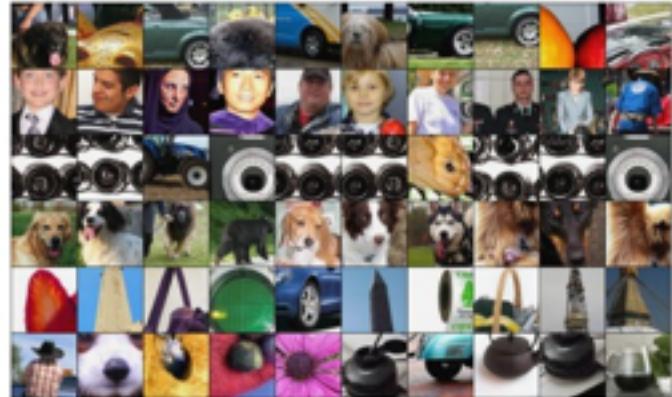
guided backpropagation



corresponding image crops



corresponding image crops



[*Striving for Simplicity: The all convolutional net*, Springenberg, Dosovitskiy, et al., 2015]

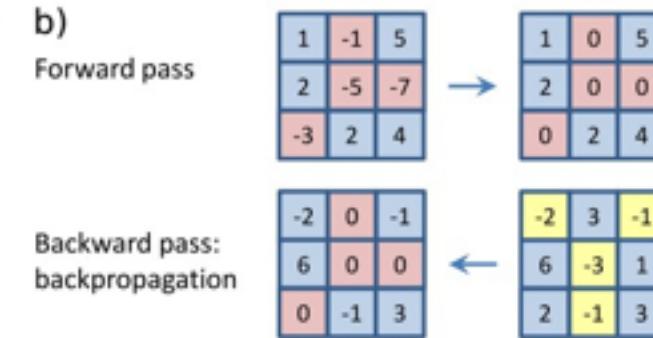
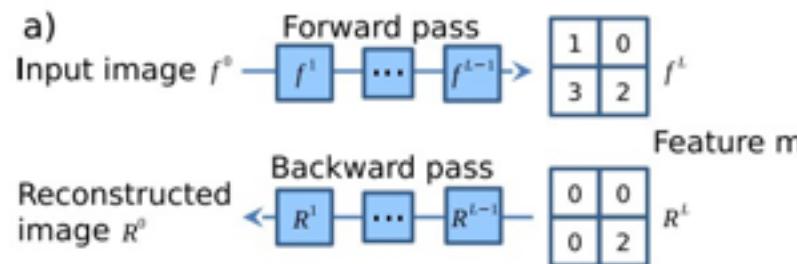
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

Deconv approaches

[Visualizing and Understanding Convolutional Networks, Zeiler and Fergus 2013]

[Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Simonyan et al., 2014]

[Striving for Simplicity: The all convolutional net, Springenberg, Dosovitskiy, et al., 2015]



c) activation: $f_i^{l+1} = \text{relu}(f_i^l) = \max(f_i^l, 0)$

backpropagation: $R_i^l = (\mathbf{f}_i^l > 0) \cdot R_i^{l+1}$, where $R_i^{l+1} = \frac{\partial f^{out}}{\partial f_i^{l+1}}$

backward 'deconvnet': $R_i^l = (\mathbf{R}_i^{l+1} > 0) \cdot R_i^{l+1}$

guided backpropagation: $R_i^l = (\mathbf{f}_i^l > 0) \cdot (\mathbf{R}_i^{l+1} > 0) \cdot R_i^{l+1}$



backprops to weights that were zero'd out by ReLu

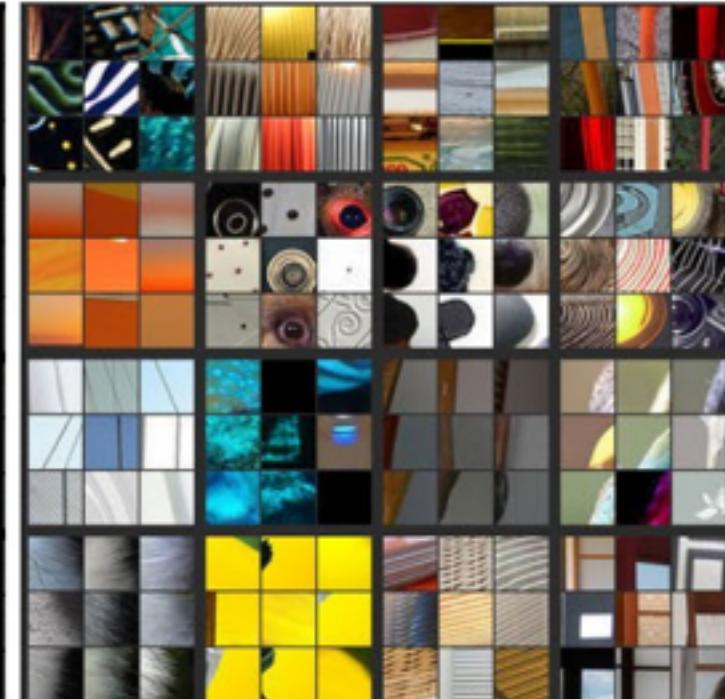
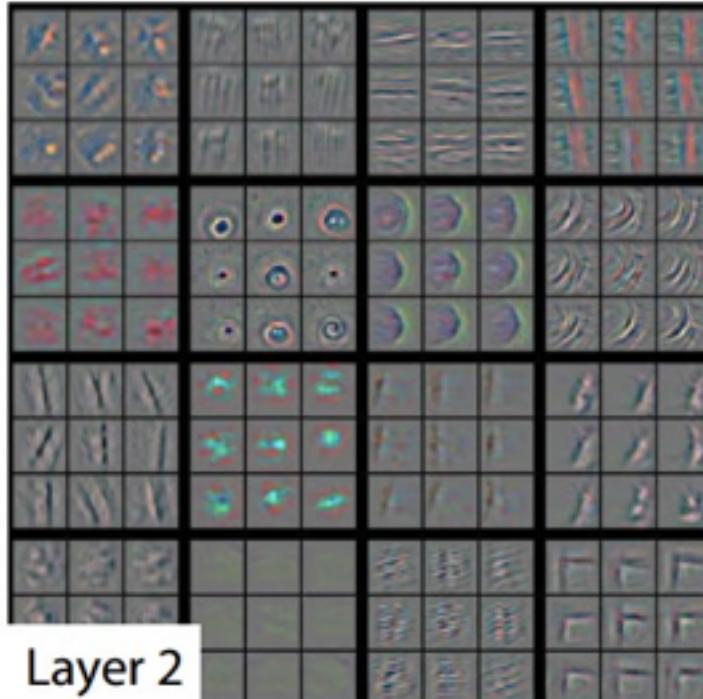
Visualizing arbitrary neurons along the way to the top...



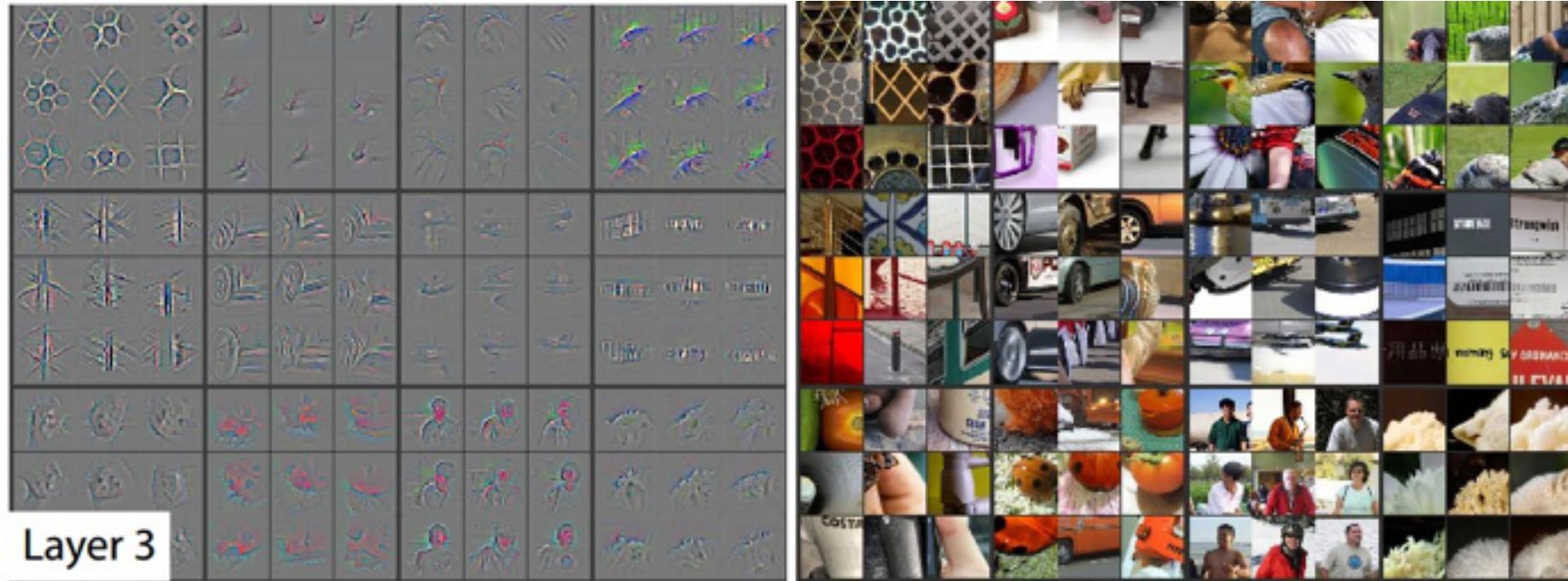
Layer 1



Layer 2

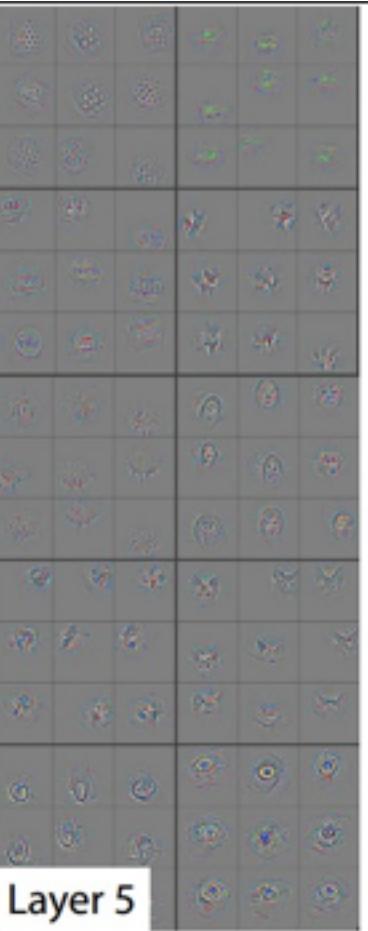


Visualizing arbitrary neurons along the way to the top...

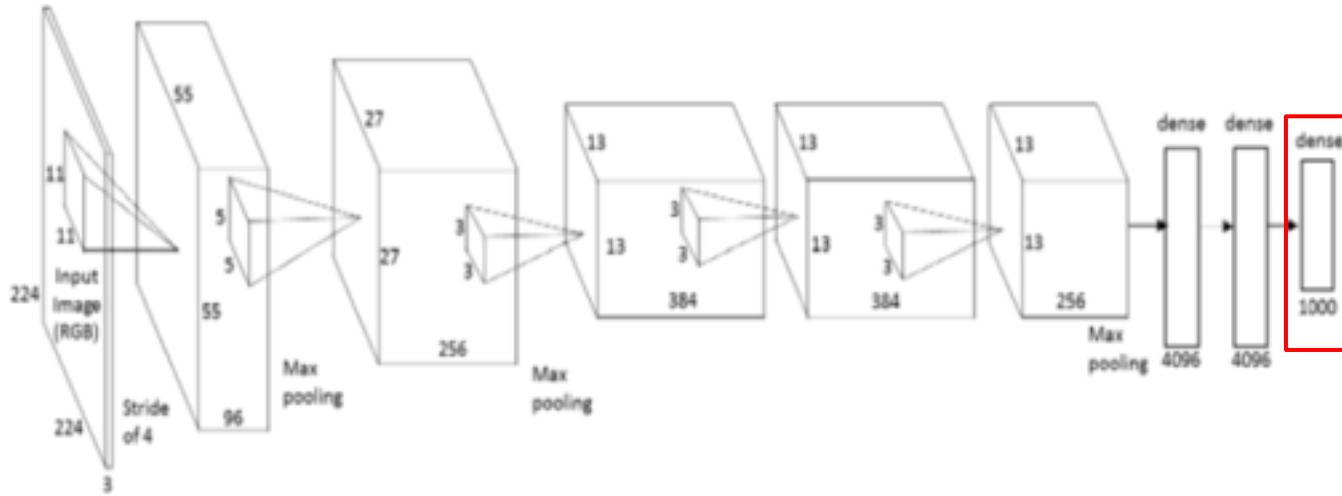


* Original slides borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

*Visualizing
arbitrary
neurons along
the way to the
top...*



Optimization to Image

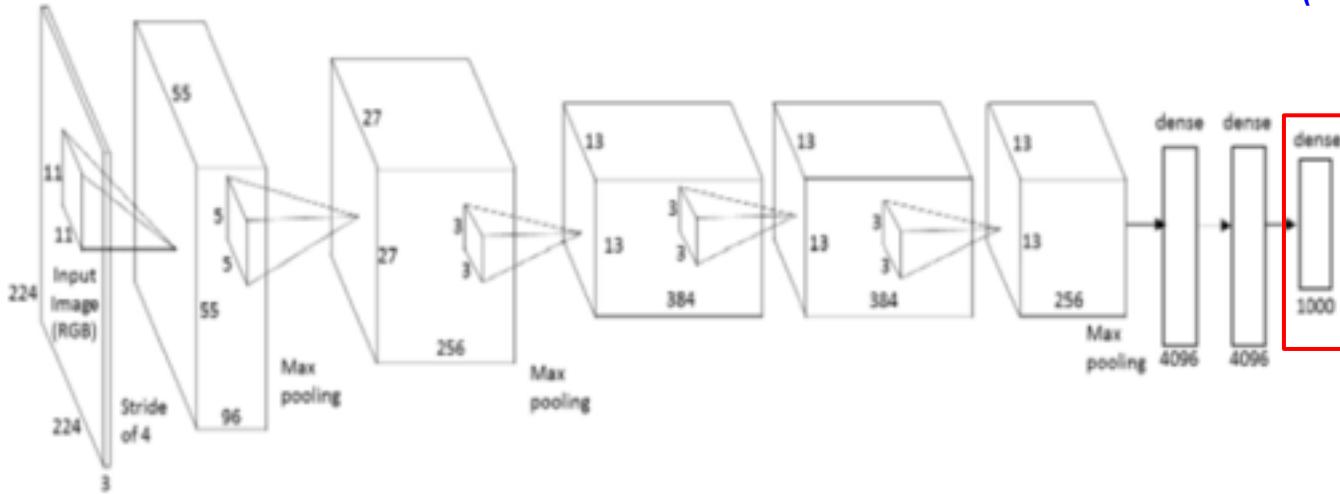


Q: can we find an image that maximizes some class score?

Optimization to Image

$$\arg \max_I S_c(I) - \lambda \|I\|_2^2$$

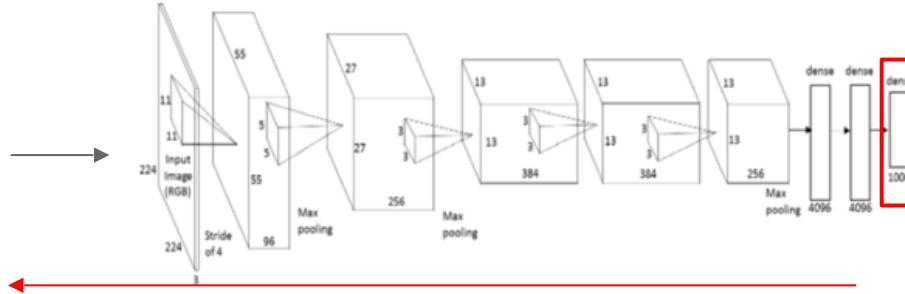
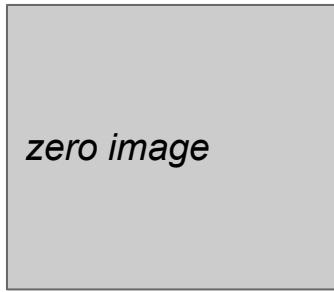
score for class c (before Softmax)



Q: can we find an image that maximizes some class score?

Optimization to Image

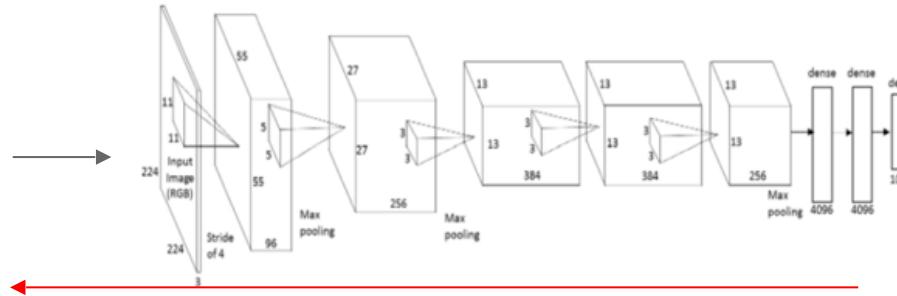
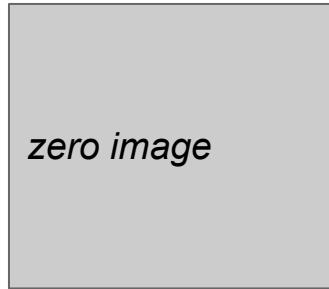
1. feed in
zeros.



2. set the gradient of the scores vector to be $[0, 0, \dots, 1, \dots, 0]$, then backprop to image

Optimization to Image

1. feed in zeros.



2. set the gradient of the scores vector to be $[0, 0, \dots, 1, \dots, 0]$, then backprop to image
3. do a small “image update”
4. forward the image through the network.
5. go back to 2.

$$\arg \max_I [S_c(I) - \lambda \|I\|_2^2]$$

score for class c (before Softmax)

1. Find images that maximize some class score:



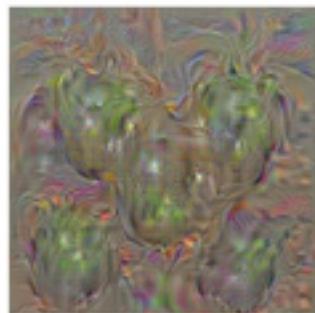
dumbbell



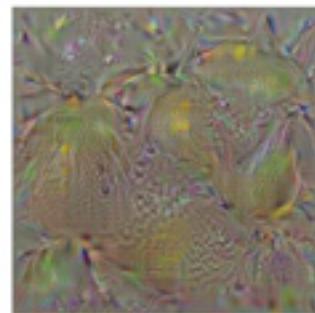
cup



dalmatian



bell pepper

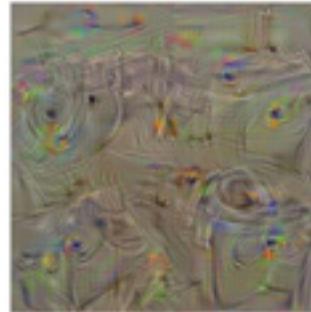


lemon

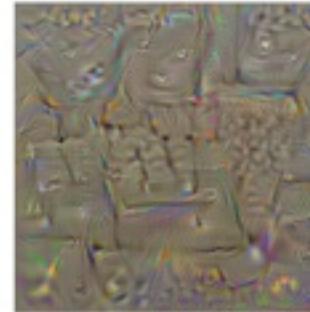


husky

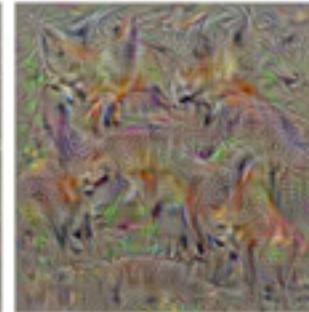
1. Find images that maximize some class score:



washing machine



computer keyboard



kit fox



goose



ostrich



limousine

2. Visualize the Data gradient:

*(note that the gradient on data has three channels.
Here they visualize M , s.t.:*



$M = ?$

$$M_{ij} = \max_c |w_{h(i,j,c)}|$$

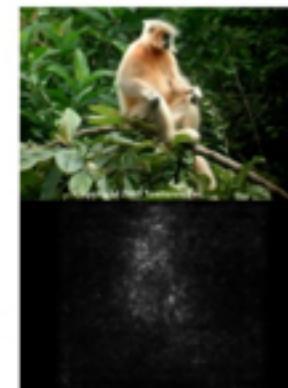
(at each pixel take abs val, and max over channels)

2. Visualize the Data gradient:

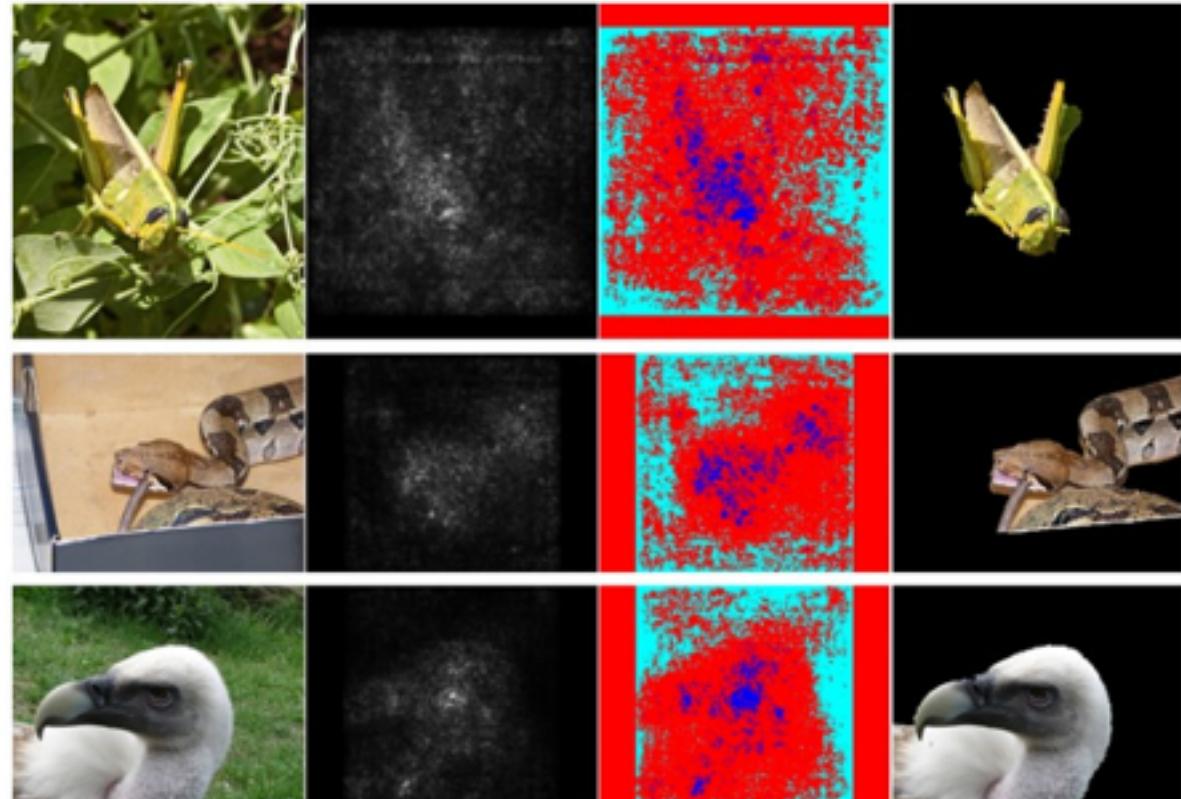
*(note that the gradient on data has three channels.
Here they visualize M , s.t.:*

$$M_{ij} = \max_c |w_{h(i,j,c)}|$$

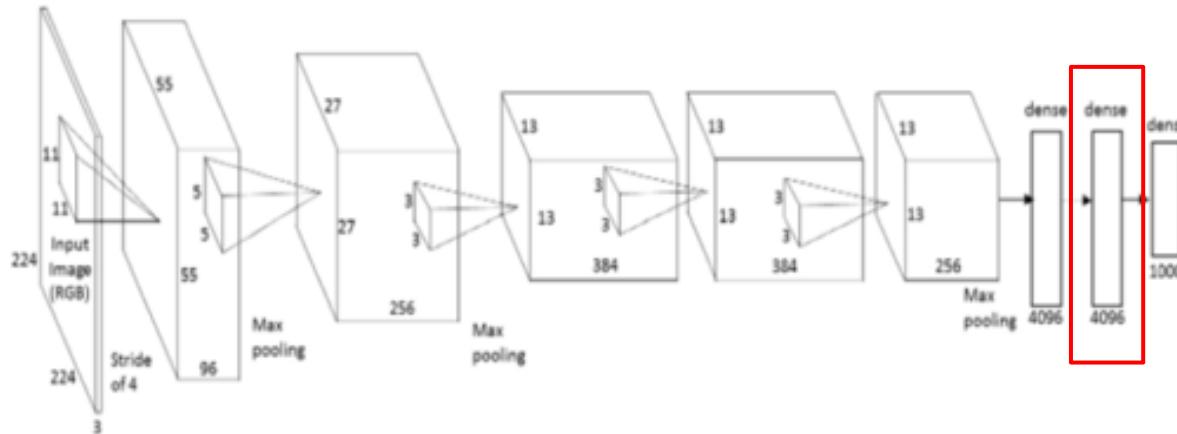
(at each pixel take abs val, and max over channels)



- Use **grabcut** for segmentation
- This optimization can be done for arbitrary neurons in the CNN



Question: Given a CNN **code**, is it possible to reconstruct the original image?

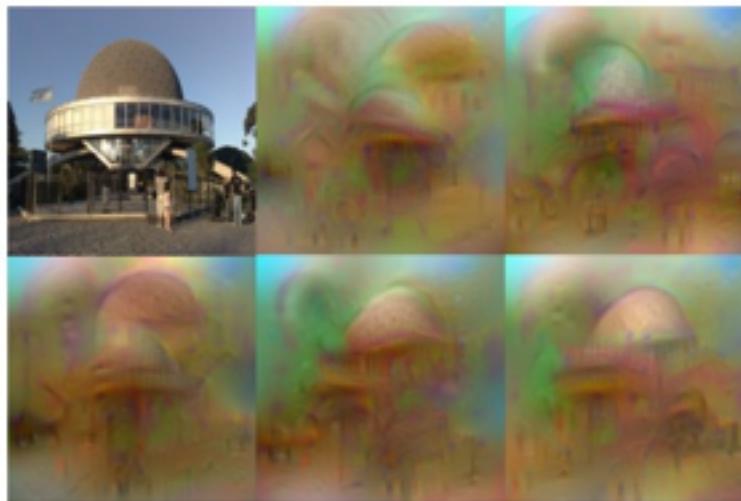


Understanding Deep Image Representations by Inverting Them
[Mahendran and Vedaldi, 2014]

Find an image such that:

- Its code is similar to a given code
- It “looks natural” (image prior regularization)

original image



*reconstructions
from the 1000
log probabilities
for ImageNet
(ILSVRC)
classes*



DeepDream <https://github.com/google/deepdream>

* Original slides borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

```

def objective_L2(dst):
    dst.diff[:] = dst.data

def make_step(net, step_size=1.5, end='inception_4c/output',
             jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''

    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]

    ox, oy = np.random.randint(-jitter, jitter+1, 2)
    src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitter shift

    net.forward(end=end)
    objective(dst) # specify the optimization objective
    net.backward(start=end)
    g = src.diff[0]
    # apply normalized ascent step to the input image
    src.data[:] += step_size/np.abs(g).mean() * g

    src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift image

    if clip:
        bias = net.transformer.mean['data']
        src.data[:] = np.clip(src.data, -bias, 255-bias)

```

```
def objective_L2(dst):
    dst.diff[:] = dst.data
```

DeepDream: set $dx = x$:)

```
def make_step(net, step_size=1.5, end='inception_4c/output',
             jitter=32, clip=True, objective=objective_L2):
    '''Basic gradient ascent step.'''
    src = net.blobs['data'] # input image is stored in Net's 'data' blob
    dst = net.blobs[end]
```

```
ox, oy = np.random.randint(-jitter, jitter+1, 2)
src.data[0] = np.roll(np.roll(src.data[0], ox, -1), oy, -2) # apply jitter shift
```

```
net.forward(end=end)
objective(dst) # specify the optimization objective
net.backward(start=end)
```

```
g = src.diff[0]
# apply normalized ascent step to the input image
src.data[:] += step_size/np.abs(g).mean() * g
```

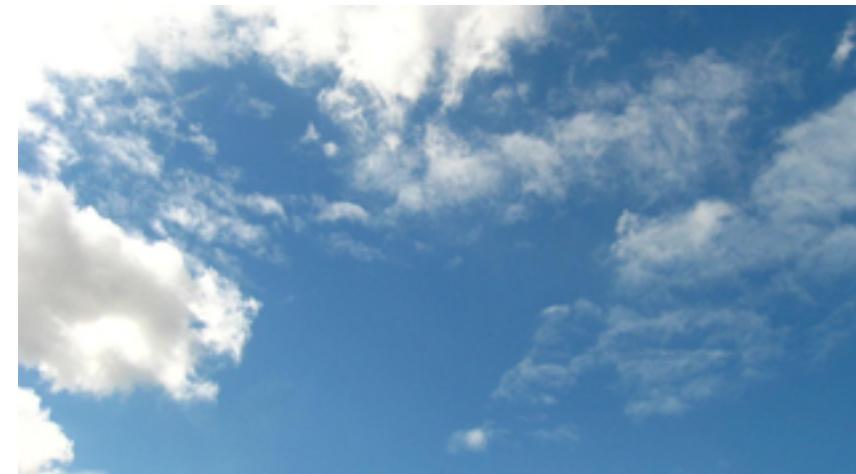
```
src.data[0] = np.roll(np.roll(src.data[0], -ox, -1), -oy, -2) # unshift image
```

```
if clip:
    bias = net.transformer.mean['data']
    src.data[:] = np.clip(src.data, -bias, 255-bias)
```

jitter regularizer

“image update”

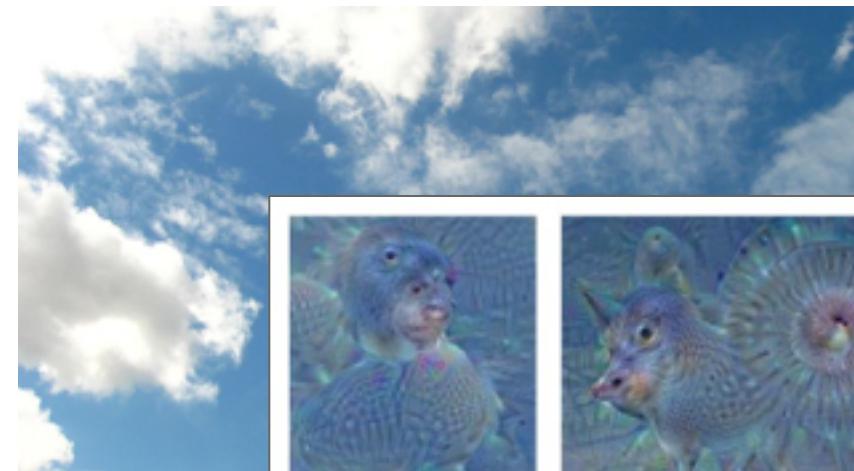
inception_4c/output



DeepDream modifies the image in a way that “boosts” all activations, at any layer

this creates a feedback loop: e.g. any slightly detected dog face will be made more and more dog like over time

inception_4c/output



"Admiral Dog!"



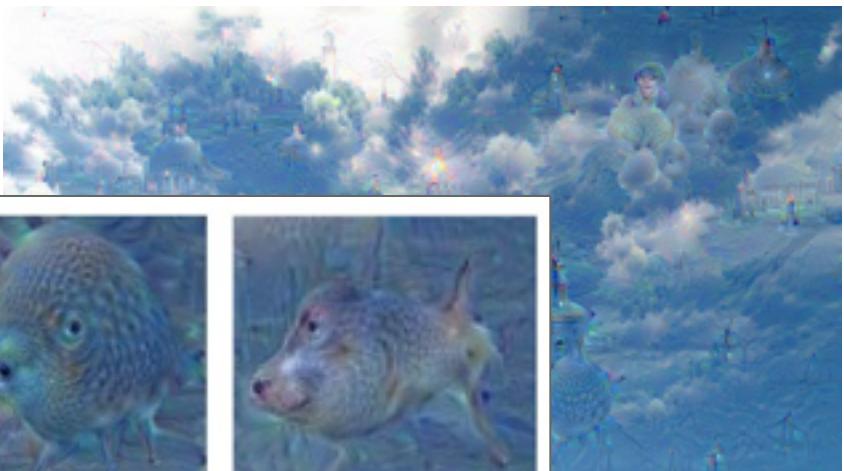
"The Pig-Snail"



"The Camel-Bird"

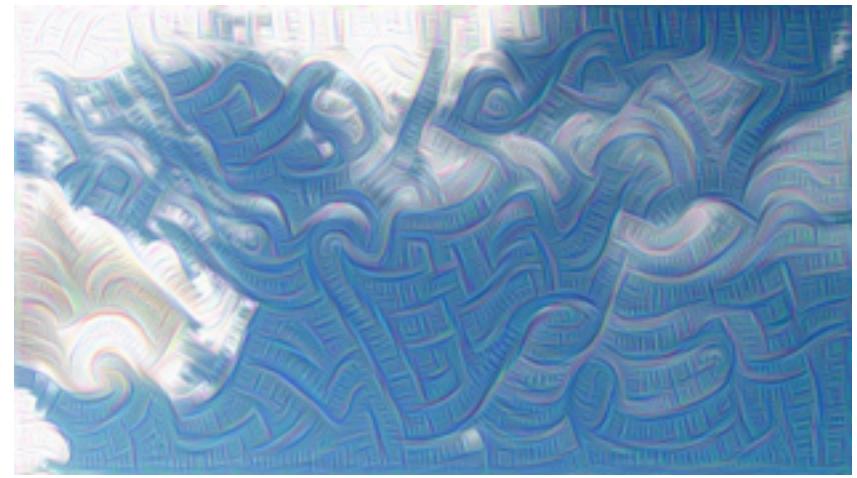
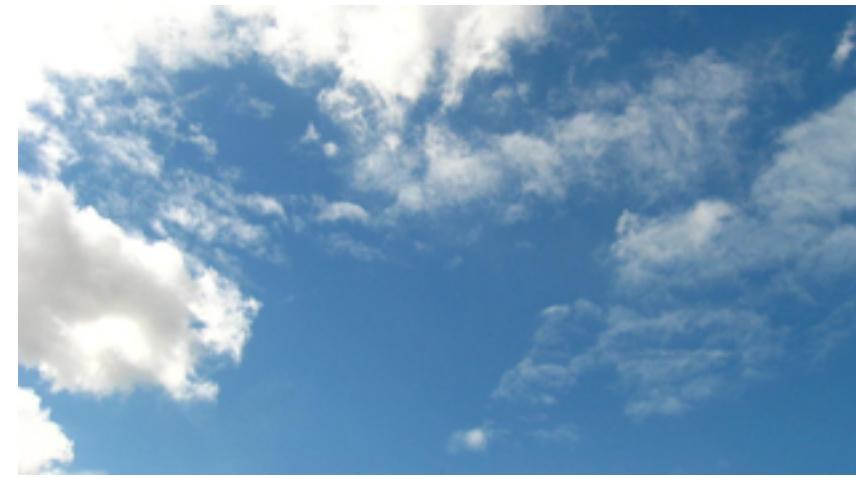


"The Dog-Fish"



DeepDream modulates the image in a way that boosts all activations, at any layer

inception_3b/5x5_reduce



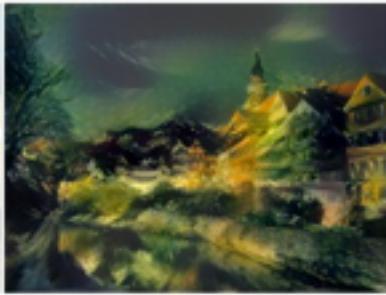
DeepDream modifies the image in a way that “boosts” all activations, at any layer

NeuralStyle

[A Neural Algorithm of Artistic Style by Leon A. Gatys,
Alexander S. Ecker, and Matthias Bethge, 2015]

good implementation by Justin in Torch:

<https://github.com/jcjohnson/neural-style>



We can pose an optimization over the input image to maximize any class score.
That seems useful.

Question: Can we use this to “fool” ConvNets?

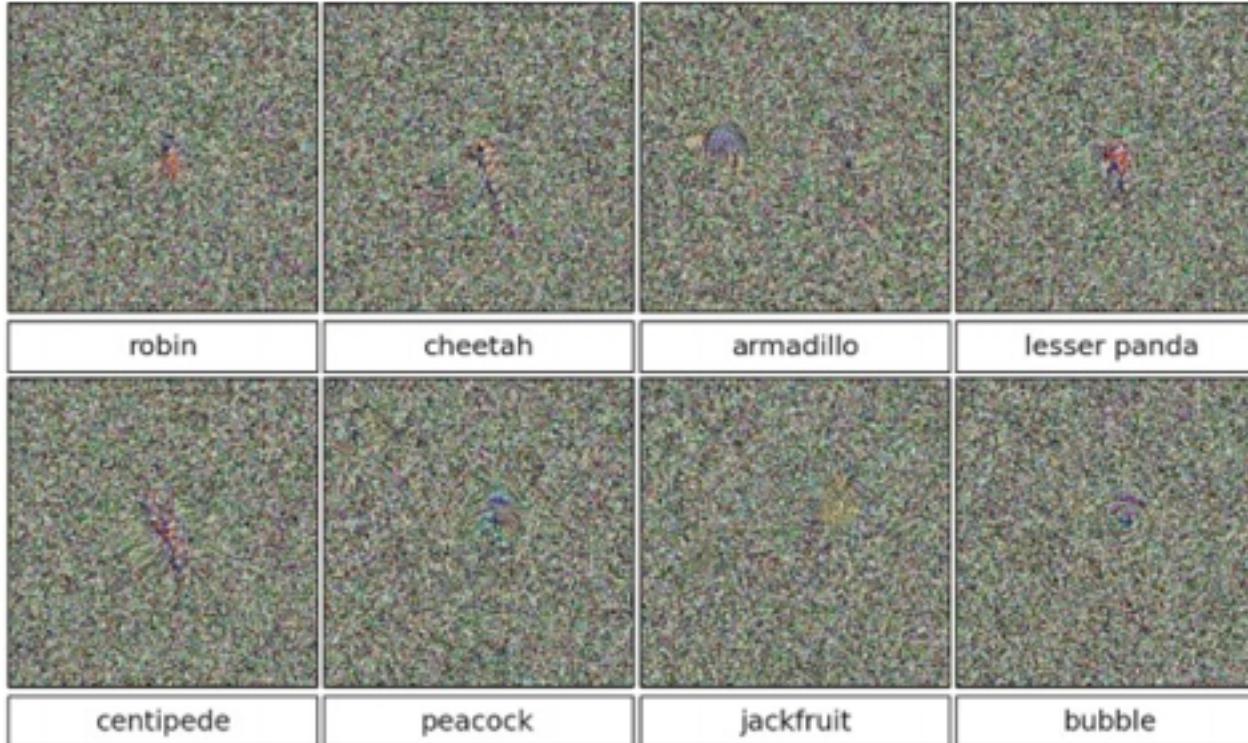
[Intriguing properties of neural networks, Szegedy et al., 2013]



* Original slides borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

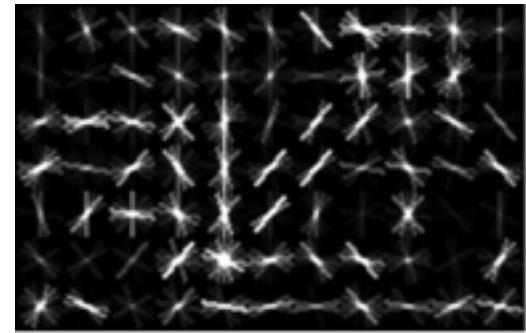
[Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images
Nguyen, Yosinski, Clune, 2014]

>99.6%
confidences



* Original slides borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

*These kinds of results were around even before ConvNets...
[Exploring the Representation Capabilities of the HOG Descriptor, Tatu et al., 2011]*

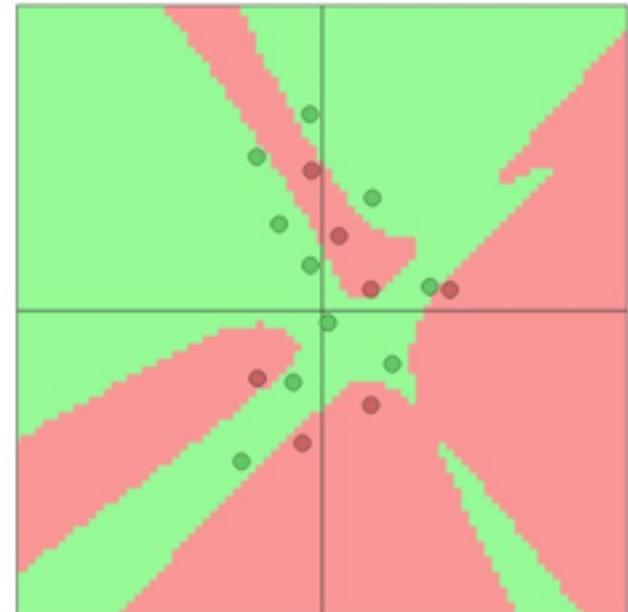


Identical HOG representation

EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES

[Goodfellow, Shlens & Szegedy, 2014]

*“primary cause of neural networks’ vulnerability to adversarial perturbation is their **linear nature**“*



Lets fool a binary linear classifier:

X	2	-1	3	-2	2	2	1	-4	5	1	← <i>input example</i>
w	-1	-1	1	-1	1	-1	1	1	-1	1	← <i>weights</i>

$$P(y = 1 | x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$

Lets fool a binary linear classifier:

X	2	-1	3	-2	2	2	1	-4	5	1	← <i>input example</i>
w	-1	-1	1	-1	1	-1	1	1	-1	1	← <i>weights</i>

class 1 score = dot product:

$$= -2 + 1 + 3 + 2 + 2 - 2 + 1 - 4 - 5 + 1 = -3$$

=> probability of class 1 is $1/(1+e^{(-(-3))}) = 0.0474$

i.e. the classifier is 95% certain that this is class 0 example.

$$P(y = 1 | x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$

Lets fool a binary linear classifier:

X	2	-1	3	-2	2	2	1	-4	5	1	← input example
W	-1	-1	1	-1	1	-1	1	1	-1	1	← weights
adversarial x	?	?	?	?	?	?	?	?	?	?	

class 1 score = dot product:

$$= -2 + 1 + 3 + 2 + 2 - 2 + 1 - 4 - 5 + 1 = -3$$

=> probability of class 1 is $1/(1+e^{-(-3)}) = 0.0474$

i.e. the classifier is 95% certain that this is class 0 example.

$$P(y = 1 | x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$

Lets fool a binary linear classifier:

X	2	-1	3	-2	2	2	1	-4	5	1	← input example
W	-1	-1	1	-1	1	-1	1	1	-1	1	← weights
adversarial x	1.5	-1.5	3.5	-2.5	2.5	1.5	1.5	-3.5	4.5	1.5	

class 1 score before:

$$-2 + 1 + 3 + 2 + 2 - 2 + 1 - 4 - 5 + 1 = -3$$

$$\Rightarrow \text{probability of class 1 is } 1/(1+e^{-(-3)}) = 0.0474$$

$$\textcolor{red}{-1.5+1.5+3.5+2.5-1.5+1.5-3.5-4.5+1.5 = 2}$$

$$\Rightarrow \text{probability of class 1 is now } 1/(1+e^{-(-2)}) = 0.88$$

i.e. we improved the class 1 probability from 5% to 88%

$$P(y=1 | x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$

Lets fool a binary linear classifier:

X	2	-1	3	-2	2	2	1	-4	5	1	← input example
W	-1	-1	1	-1	1	-1	1	1	-1	1	← weights
adversarial x	1.5	-1.5	3.5	-2.5	2.5	1.5	1.5	-3.5	4.5	1.5	

class 1 score before:

$$-2 + 1 + 3 + 2 + 2 - 2 + 1 - 4 - 5 + 1 = -3$$

=> probability of class 1 is $1/(1+e^{(-(-3))}) = 0.0474$

$$\textcolor{red}{-1.5+1.5+3.5+2.5-1.5+1.5-3.5-4.5+1.5 = 2}$$

=> probability of class 1 is now $1/(1+e^{(-(2)}) = 0.88$

i.e. we improved the class 1 probability from 5% to 88%

This was only with 10 input dimensions. A 224x224 input image has 150,528.

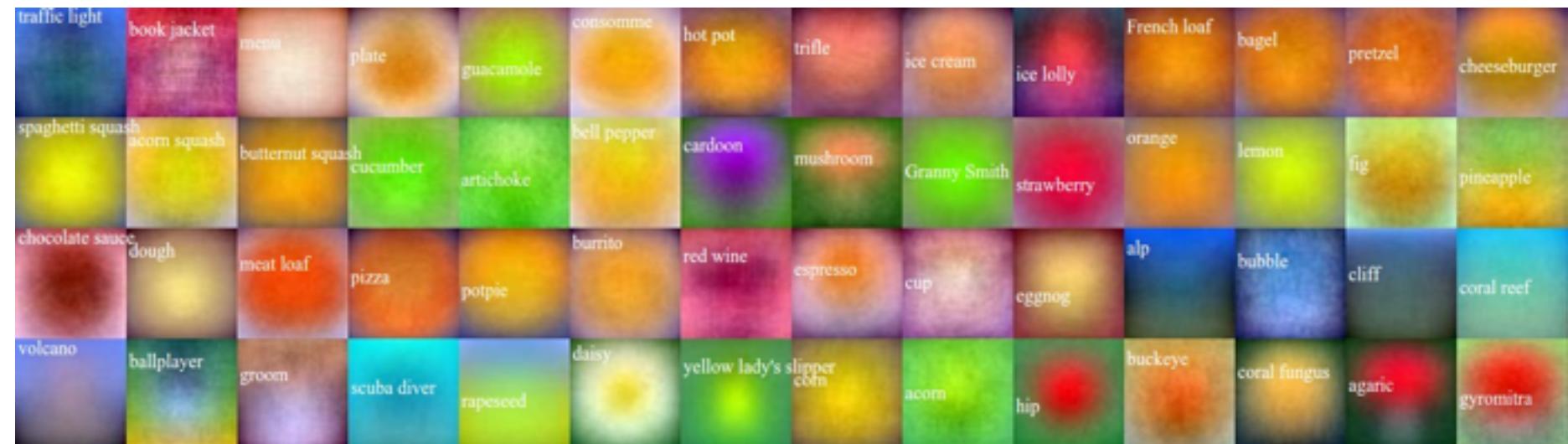
(It's significantly easier with more numbers, need smaller nudge for each)

Andrej Karpathy Blog post: Breaking Linear Classifiers on ImageNet

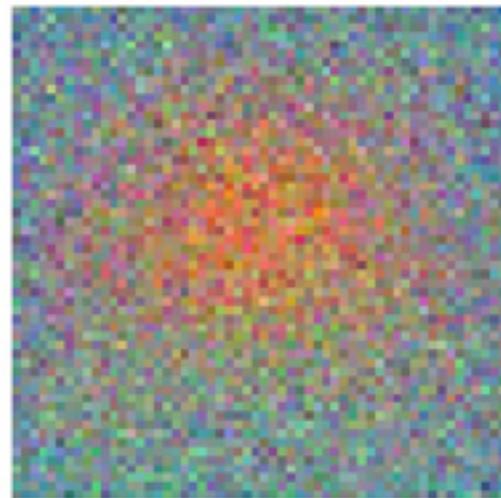
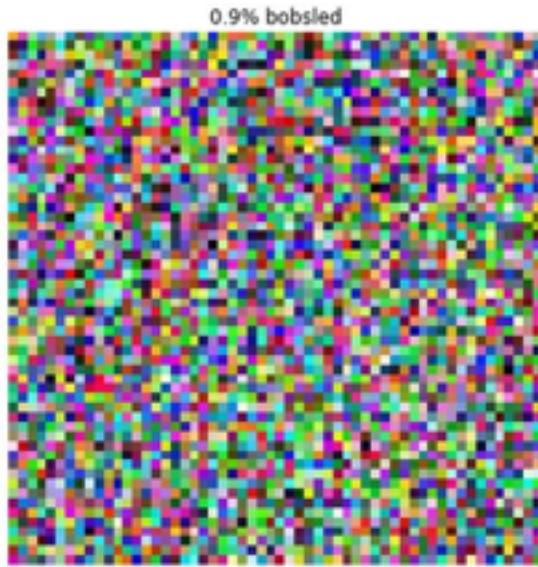
Recall CIFAR-10 linear classifiers:



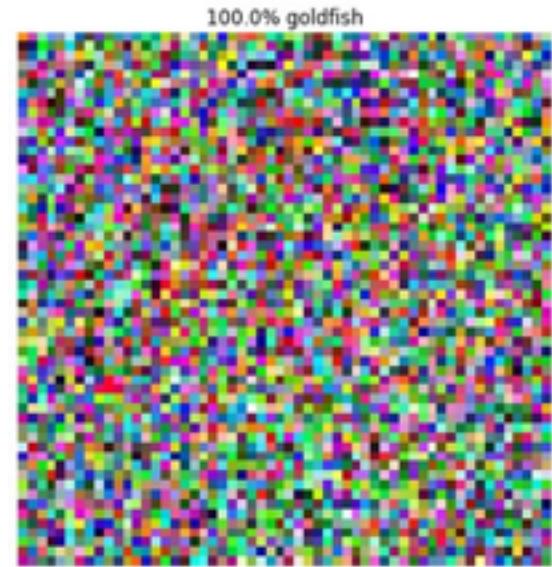
ImageNet classifiers:



*mix in a tiny bit of
Goldfish classifier weights*

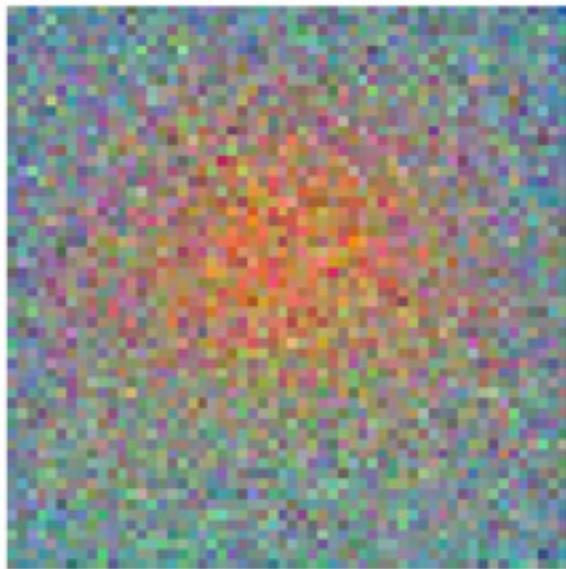


=



100% Goldfish

1.0% kit fox

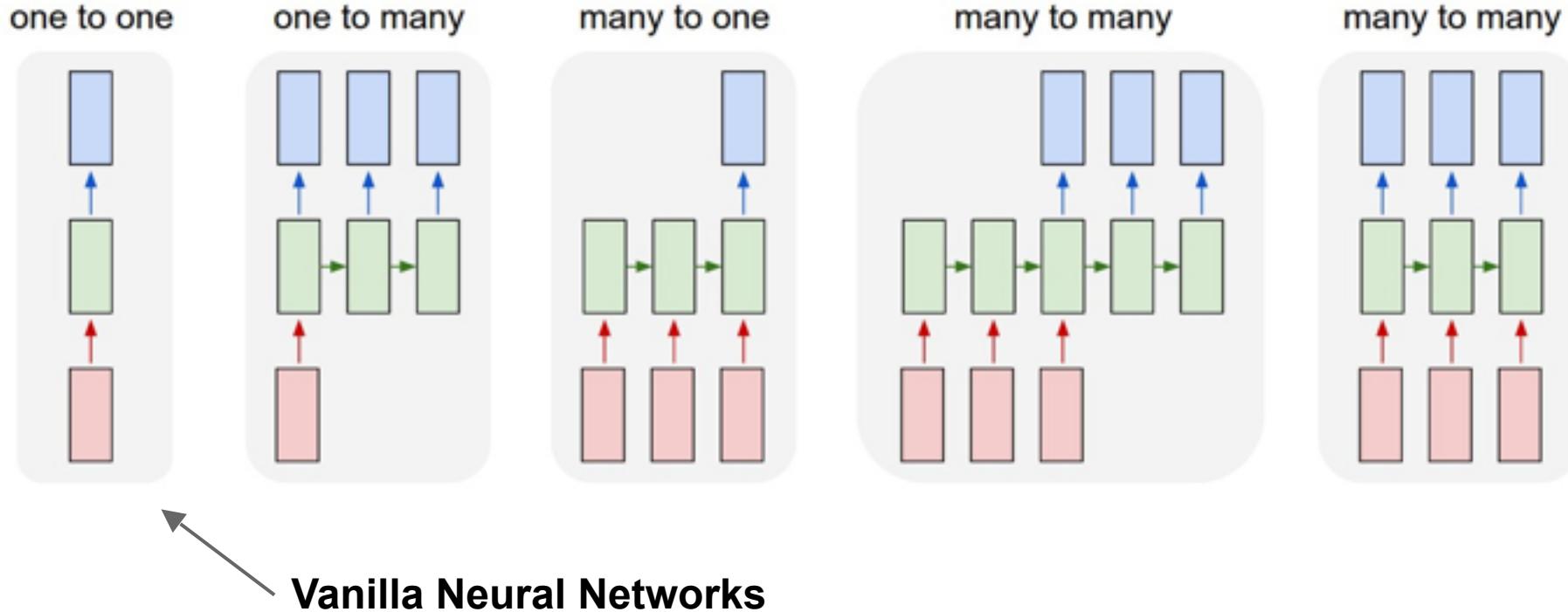


8.0% goldfish

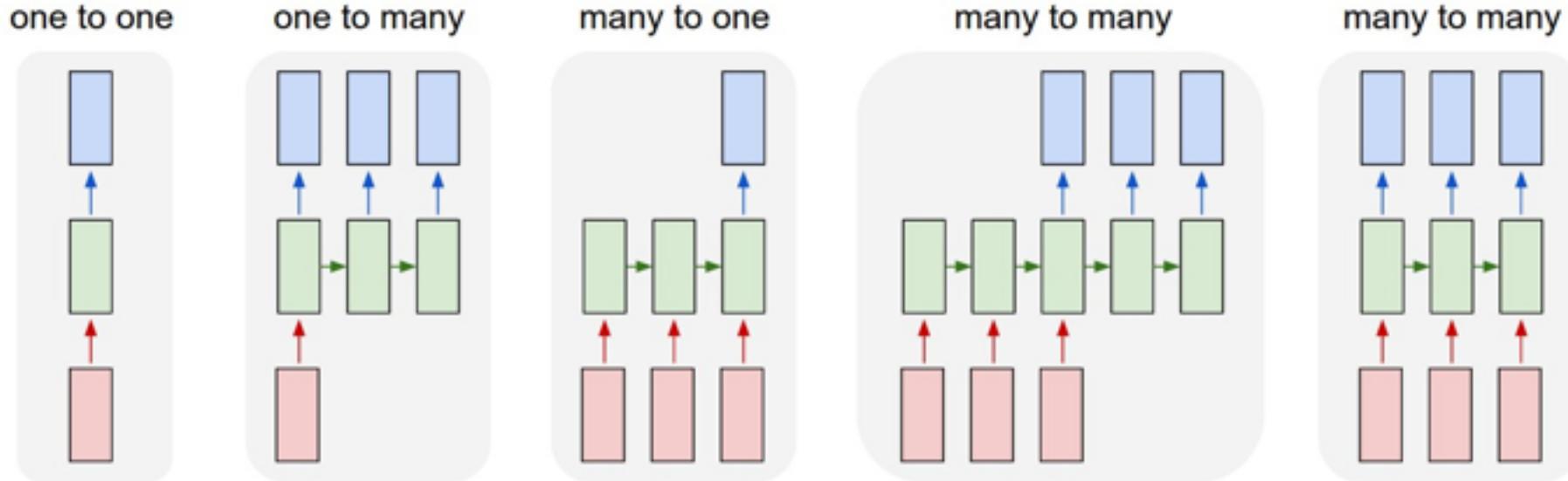


Recurrent Neural Networks

Recurrent Networks offer a lot of flexibility:



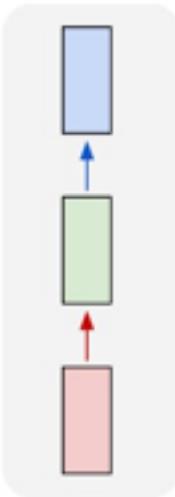
Recurrent Networks offer a lot of flexibility:



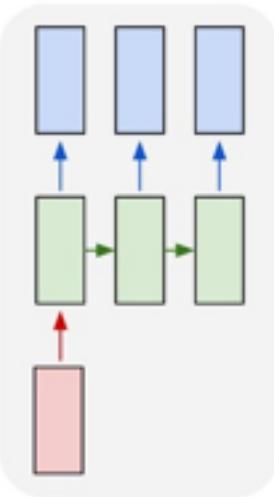
e.g. **Image Captioning**
image -> sequence of words

Recurrent Networks offer a lot of flexibility:

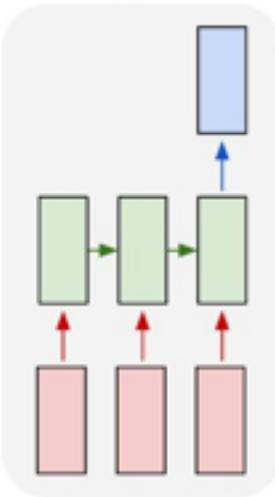
one to one



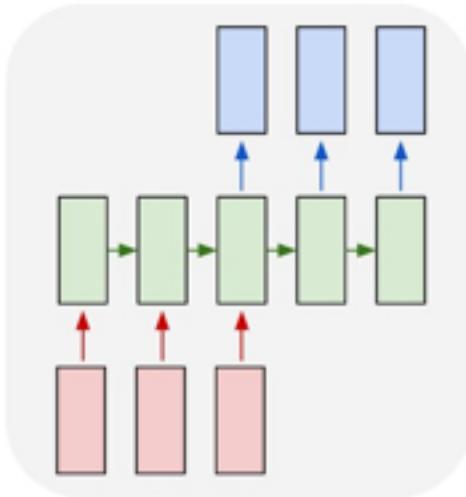
one to many



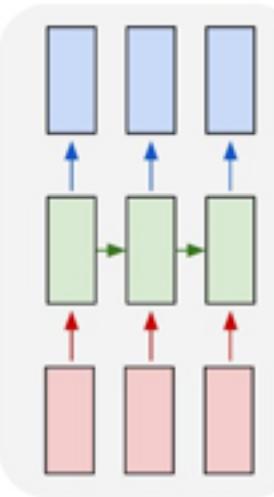
many to one



many to many

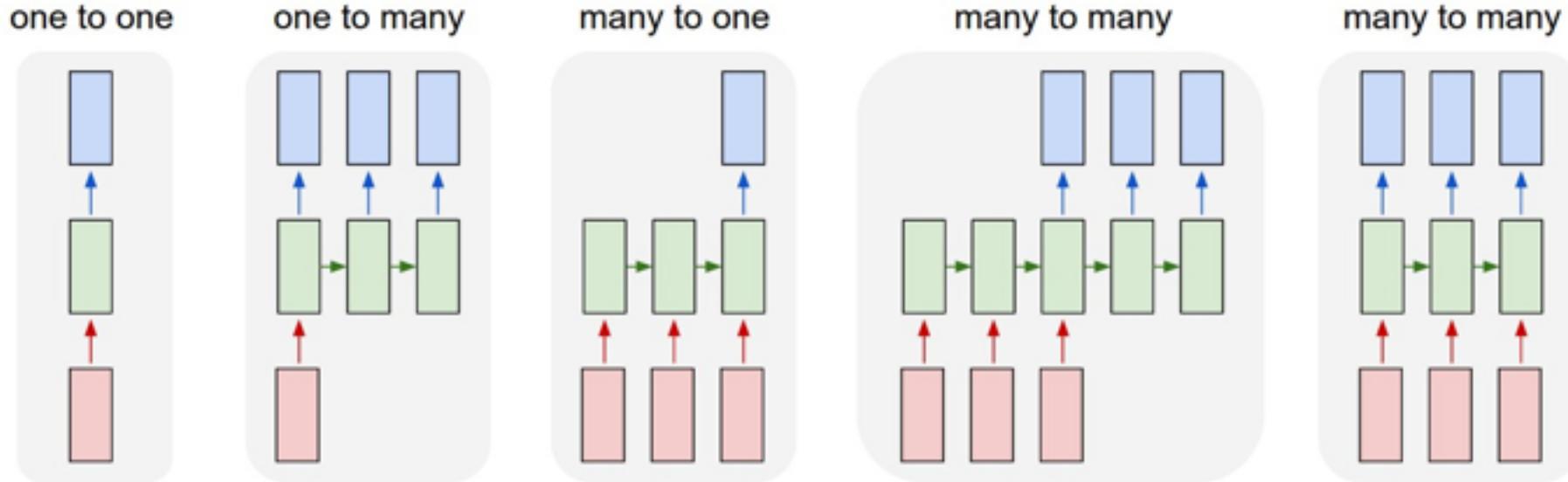


many to many



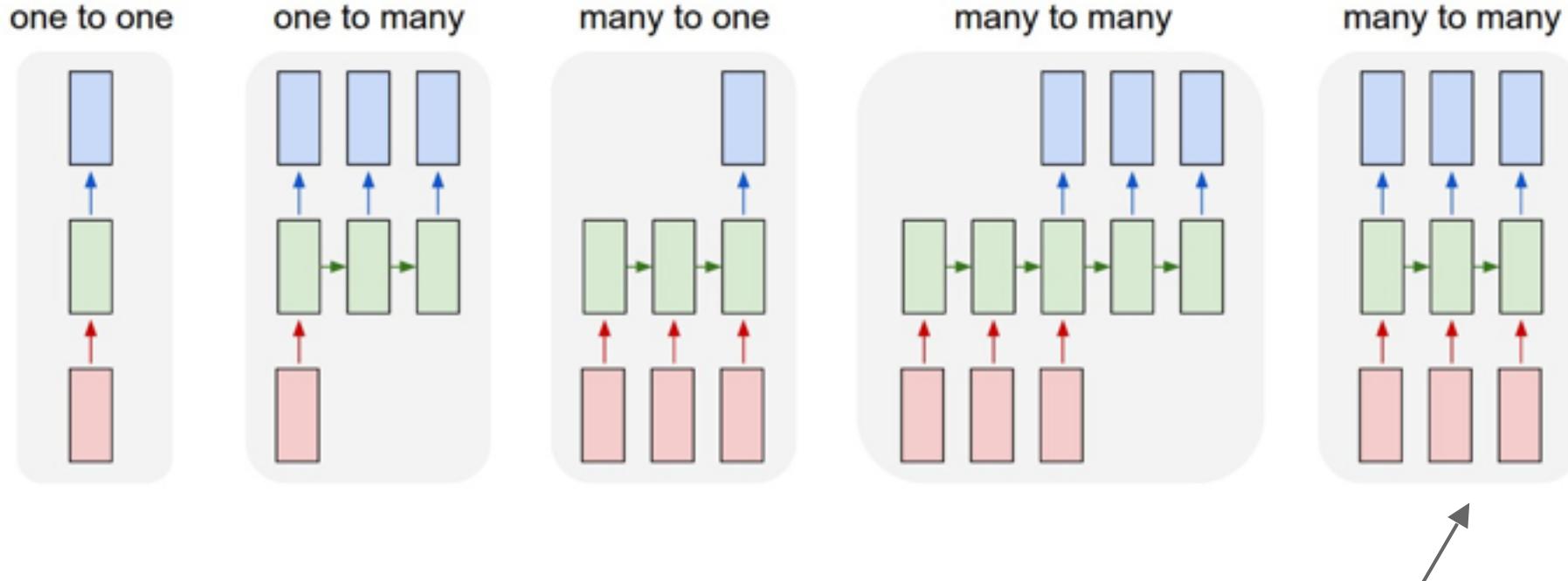
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Networks offer a lot of flexibility:

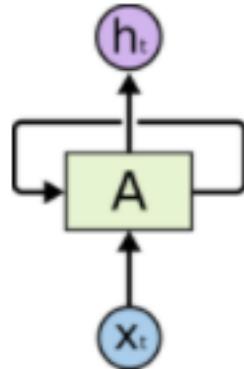


e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Networks offer a lot of flexibility:

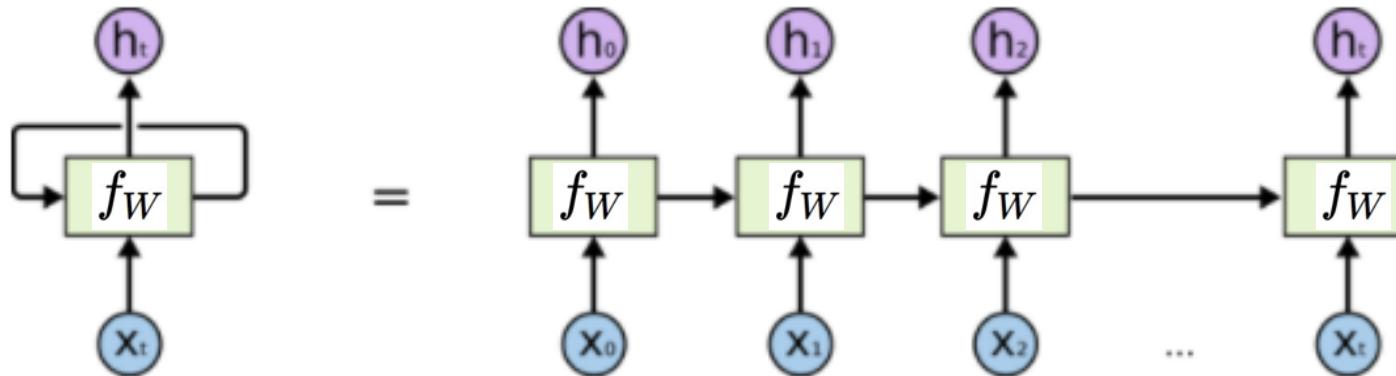


Recurrent Networks



Recurrent Neural Networks have loops.

RNN - at each time step



An unrolled recurrent neural network.

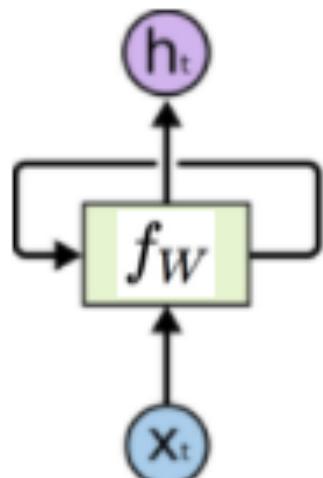
$$h_t = f_W(h_{t-1}, x_t)$$

new state old state
some function
with parameters W

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 \mathbf{h} :



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



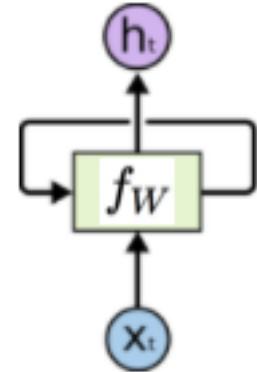
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Character-level language model example

Vocabulary:
[h,e,l,o]

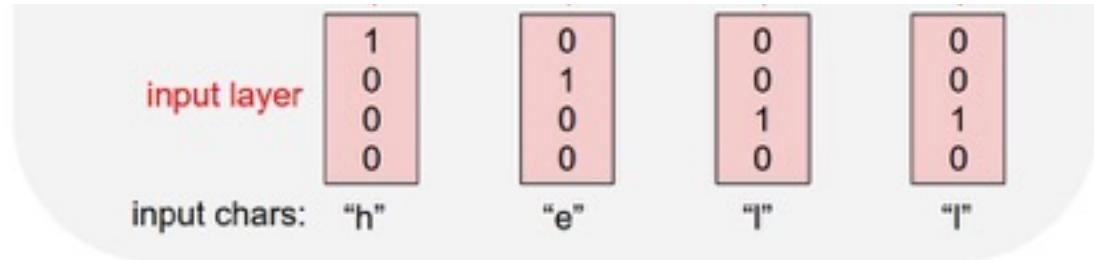
Example training
sequence:
“hello”



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

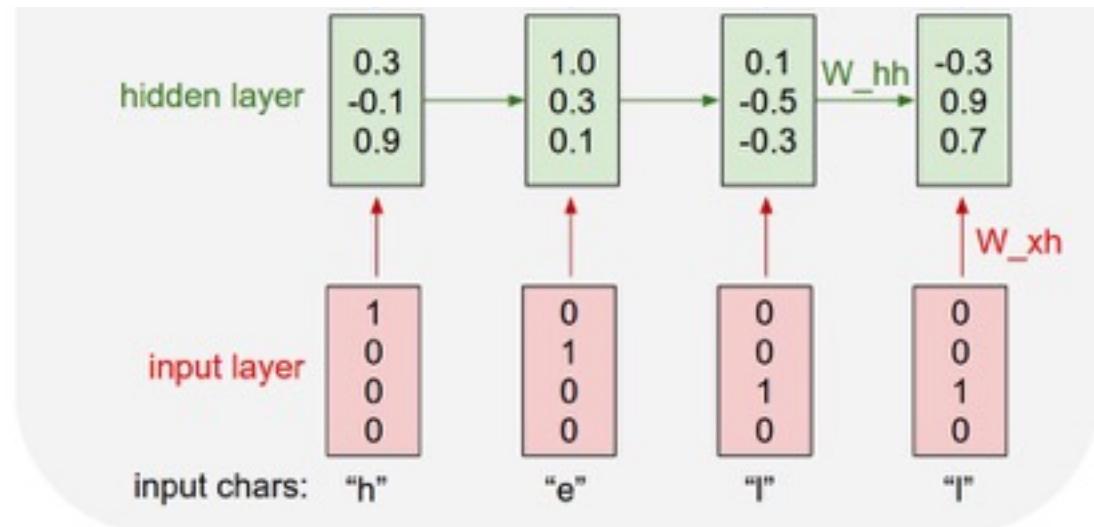


Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”

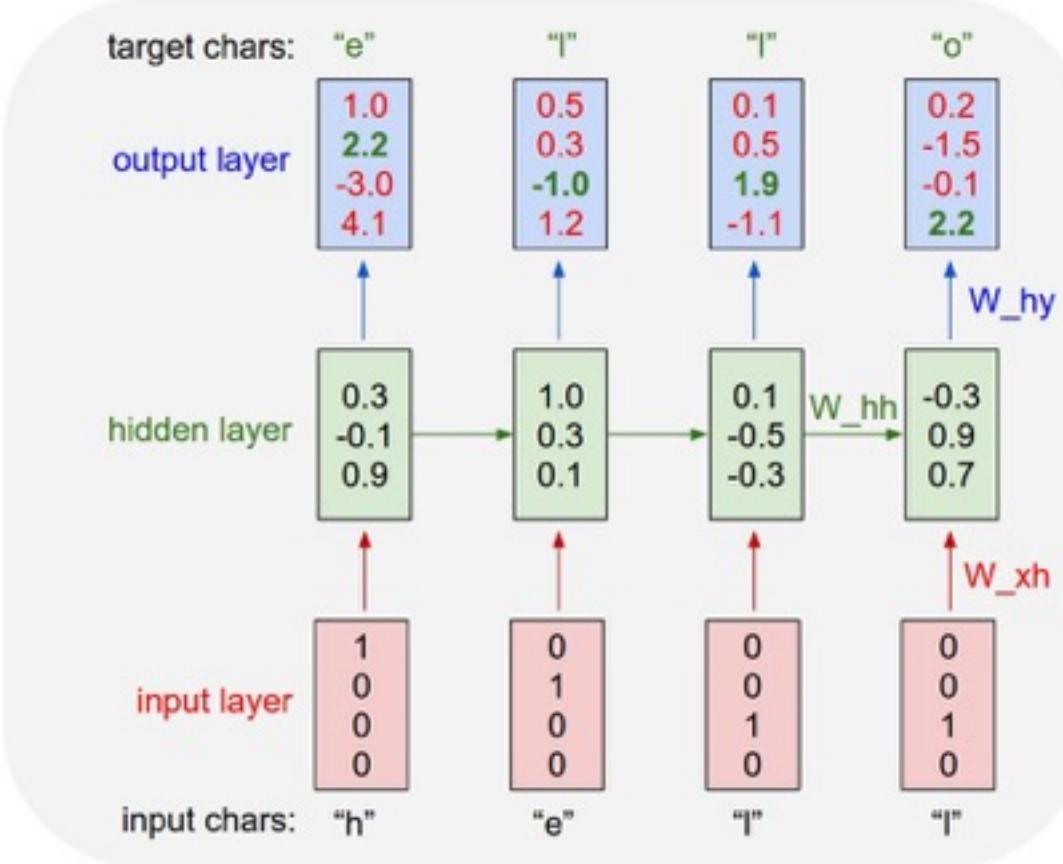
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



Character-level language model example

Vocabulary:
[h,e,l,o]

Example training
sequence:
“hello”



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

```
1  #!/usr/bin/python
2  #
3  # Minimal character-level vanilla RNN model, written by Andrej Karpathy (@karpathy)
4  # 2014.12.04
5  #
6  # Import numpy as np
7  #
8  # n: float 0.19
9  # data: file object
10 # data.read() = should be simple plain text file
11 # chars = [char for char in data]
12 # data_size, vocab_size = len(chars), len(chars)
13 # print('data has %d characters, %d unique.' % (data_size, vocab_size))
14 # char_to_ix = {char: i for i, char in enumerate(chars)}
15 # ix_to_char = {i: char for char, i in char_to_ix.items()}
16 #
17 # Hyperparameters
18 # hidden_size = 100 = size of hidden layer of neurons
19 # seq_length = 100 = number of steps to unroll the RNN for
20 # learning rate = 0.1
21 #
22 # Model parameters
23 # wih = np.random.rand(100, 100)*0.01 + 0.001 to 0.0001
24 # whh = np.random.rand(100, 100)*0.01 + 0.0001 to 0.0001
25 # why = np.random.rand(100, 100)*0.01 + 0.0001 to 0.0001
26 # bi = np.zeros((100, 1)) = hidden bias
27 # bo = np.zeros((100, 1)) = output bias
28 #
29 # Inputs/Targets, targets, hprev
30 #
31 # inputs/targets are both list of integers.
32 # hprev is init array of initial hidden state
33 # returns the loss, gradients on model parameters, and last hidden state
34 #
35 # xs, hs, ys, ps = (), (), (), ()
36 # hprev = np.copy(hprev)
37 # loss = 0
38 # a forward pass
39 # for t in xrange(len(inputs)):
40 #     hprev = np.tanh(np.dot(wih, inputs[t]) + np.dot(whh, hs[-1]) + bi) = hidden in t-of-R representation
41 #     inputs[t] = 1
42 #     hs[t] = np.tanh(np.dot(wih, inputs[t]) + np.dot(whh, hs[-1]) + bi) = hidden state
43 #     ps[t] = np.exp(why[t]) / np.sum(np.exp(why[t])) = softmax probabilities for next chars
44 #     ps[t] = np.exp(why[t]) / np.sum(np.exp(why[t])) = softmax cross-entropy loss
45 #     loss += -np.log(ps[t]) * targets[t] = loss from current step
46 #
47 # a backward pass (unrolled through time)
48 # dhnext = np.zeros((100, 1)) = gradients going backwards
49 # dhprev = np.zeros((100, 1)) = gradients going backwards
50 # dwhy = np.zeros((100, 100)) = gradients for why
51 # dwih = np.zeros((100, 100)) = gradients for wih
52 # dwhh = np.zeros((100, 100)) = gradients for whh
53 # dbi = np.zeros((100, 1)) = gradients for bi
54 # dloss = np.zeros((100, 1)) = gradients for loss
55 # dhnext = np.tanh(hs[-1]) * dloss = gradient of loss through tanh nonlinearity
56 # dhprev = (1 - hs[-1]**2) * dhnext = gradient of loss through tanh nonlinearity
57 # dwih = np.dot(inputs.T, dhnext) = gradient of loss through wih
58 # dwhh = np.dot(hs[-1].T, dhnext) = gradient of loss through whh
59 # dbi = np.sum(dhnext, axis=0) = gradient of loss through bi
60 # for dparam in [dwih, dwhh, dwhy, dbi, dloss]:
61 #     np.clip(dparam, -5, 5, out=dparam) = clip to mitigate exploding gradients
62 #
63 # return loss, dhnext, dwih, dwhh, dwhy, dbi, dloss
```

```
def sampleRNN(hprev, m):
    """
    sample a sequence of integers from the model
    h: 1d memory state, seed_ix is seed character for first time step
    m: 1d np.array of weights, 100
    nhidden = 100
    seqLength = 100
    """
    for t in range(seqLength):
        hprev = np.tanh(np.dot(wih, x) + np.dot(whh, h) + bi)
        y = np.dot(why, h) + bo
        p = np.exp(y) / np.sum(np.exp(y))
        ix = np.random.choice(np.arange(nhidden), p=p)
        x = np.zeros((nhidden, 1))
        x[ix] = 1
        hprev.append(ix)
    return hprev

t = 0
m = np.zeros((100, 100))
mbi, mbw, mbh, mbp = np.zeros(100), np.zeros(100), np.zeros(100), np.zeros(100)
mb, mbh, mbp = np.zeros(100), np.zeros(100), np.zeros(100) = memory variables for adapted softmax, loss, 1 - np.log(1.0/nvocab)*seqLength = loss at iteration 0
while True:
    # prepare inputs (we're looping from left to right in steps seqLength long)
    if t+seqLength+1 > len(data):
        hprev = np.zeros((100, 100)) = reset this memory
        p = 0 = np.random.choice(nhidden) = start of data
        inputs = [char_to_ix[p]] = For ch in data[seqLength:t+seqLength]
        targets = [char_to_ix[ch] = For ch in data[t+seqLength:t+seqLength+1]]
        p += 1 = sample from the model now and then
    else:
        sample_ix = sampleRNN(hprev, inputs[t], m)
        net = "" = join(ix, toChar[i]) = For ix in sample_ix:
        net += net = np.concatenate([net, sample_ix])
    # forward seqLength characters through the net and fetch gradient
    loss, dhnext, dwih, dwhh, dwhy, dpres = lossFun(inputs, targets, pres)
    smoothLoss = smoothLoss + 0.99 * loss - loss
    if t % 100 == 0: print("iter %d, loss: %f" % (t, smoothLoss)) = print progress
    t += 1 = perform parameter update with gradient
    for param, dparam, m in zip([wih, whh, why, bi, bo],
                                [dwih, dwhh, dwhy, dbi, dbo],
                                [mbi, mbw, mbh, mbp]):
        m += dparam * pres
        param -= learningRate * pres / np.sqrt(m + 1e-10) = adapted update
    t += 1 = seqLength + new data position
    t += 1 = iteration counter
```

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

Data I/O

```
# Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
etc

import numpy as np

# Data I/O
with open('input.txt', 'r') as f:
    raw = f.read()

# chars: unique chars in raw
# data_size, vocab_size = len(raw), 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) }

# Forward pass
def forward(inputs, targets, hprev):
    if inputs.size == 1:
        inputs, targets = np.array([inputs]), np.array([targets])
    else:
        inputs, targets = inputs[0], targets[0]

    if hprev == None:
        hprev = np.zeros((hidden_size,))

    # Unpack weights
    Wx, Wh, b, h0, hidden_size = np.load('weights.npz')
    hidden_size == 100
    hidden_size == len(chars)

    # Initial hidden state
    hprev = h0

    # Input-to-hidden
    xprev = np.zeros((hidden_size,))
    xprev[inputs] = 1.0
    hprev = np.tanh(np.dot(xprev, Wx) + np.dot(hprev, Wh) + b)

    # Hidden-to-hidden
    hprev = np.tanh(np.dot(hprev, Wh) + b)

    # Softmax output
    outputs = np.exp(hprev) / np.sum(np.exp(hprev))
    outputs = np.log(outputs) * targets + np.log(1 - outputs) * (1 - targets)

    # Loss
    loss = -np.sum(outputs) / len(targets)

    # Compute gradients
    dWx, dWh, db = np.zeros_like(Wx), np.zeros_like(Wh), np.zeros(hidden_size)
    dhprev = np.zeros_like(h0)

    # Backpropagation
    for t in reversed(range(len(inputs))):
        # Compute softmax gradient
        dhprev += -outputs[t] * targets[t] / len(targets) * np.exp(hprev[t]) / np.sum(np.exp(hprev[t]))
        dhprev -= outputs[t] / len(targets) * np.exp(hprev[t]) * np.sum(np.exp(hprev[t])) / len(targets)

        # Compute input-to-hidden gradient
        dWx += np.dot(xprev, dhprev).T
        dhprev = np.dot(dhprev, Wh.T) * (1 - np.tanh(hprev) * np.tanh(hprev)) * np.exp(hprev) * -targets[t]
        dhprev *= -1.0 / len(targets)

        # Compute hidden-to-hidden gradient
        dWh += np.dot(hprev, dhprev).T
        dhprev = np.dot(dhprev, Wh.T) * (1 - np.tanh(hprev) * np.tanh(hprev)) * np.exp(hprev) * -targets[t]
        dhprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. hidden state
        dhprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. previous hidden state
        dhprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. inputs
        dxprev = np.dot(dhprev, Wx)
        dxprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. weights and bias
        dWx += np.dot(xprev, dxprev).T
        dhprev = np.dot(dxprev, Wh.T) * (1 - np.tanh(hprev) * np.tanh(hprev)) * np.exp(hprev) * -targets[t]
        dhprev *= -1.0 / len(targets)

    # Average over steps
    dWx /= len(inputs)
    dWh /= len(inputs)
    db /= len(inputs)
    dhprev /= len(inputs)

    return loss, dWx, dWh, db, outputs, hprev, np.argmax(hprev, 1)
```

```
# Inverse operations
hidden_size = 100 # size of hidden layer of neurons
seq_length = 200 # number of steps to unroll the seq for training
hprev = np.zeros((hidden_size,))

for i in range(seq_length):
    print "Input: '%s'" % raw[i]
    raw[i] = raw[i].decode('utf-8').strip()
    assert len(raw[i]) == 1, "Input char must be a single character"
    data_size, vocab_size = len(raw), 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) }

# forward pass
for inputs, targets, hprev:
    if inputs.size == 1:
        inputs, targets = np.array([inputs]), np.array([targets])
    else:
        inputs, targets = inputs[0], targets[0]

    if hprev == None:
        hprev = np.zeros((hidden_size,))

    # Unpack weights
    Wx, Wh, b, h0, hidden_size = np.load('weights.npz')
    hidden_size == 100
    hidden_size == len(chars)

    # Initial hidden state
    hprev = h0

    # Input-to-hidden
    xprev = np.zeros((hidden_size,))
    xprev[inputs] = 1.0
    hprev = np.tanh(np.dot(xprev, Wx) + np.dot(hprev, Wh) + b)

    # Hidden-to-hidden
    hprev = np.tanh(np.dot(hprev, Wh) + b)

    # Softmax output
    outputs = np.exp(hprev) / np.sum(np.exp(hprev))
    outputs = np.log(outputs) * targets + np.log(1 - outputs) * (1 - targets)

    # Loss
    loss = -np.sum(outputs) / len(targets)

    # Compute gradients
    dWx, dWh, db = np.zeros_like(Wx), np.zeros_like(Wh), np.zeros(hidden_size)
    dhprev = np.zeros_like(h0)

    # Backpropagation
    for t in reversed(range(len(inputs))):
        # Compute softmax gradient
        dhprev += -outputs[t] * targets[t] / len(targets) * np.exp(hprev[t]) / np.sum(np.exp(hprev[t]))
        dhprev -= outputs[t] / len(targets) * np.exp(hprev[t]) * np.sum(np.exp(hprev[t])) / len(targets)

        # Compute input-to-hidden gradient
        dWx += np.dot(xprev, dhprev).T
        dhprev = np.dot(dhprev, Wh.T) * (1 - np.tanh(hprev) * np.tanh(hprev)) * np.exp(hprev) * -targets[t]
        dhprev *= -1.0 / len(targets)

        # Compute hidden-to-hidden gradient
        dWh += np.dot(hprev, dhprev).T
        dhprev = np.dot(dhprev, Wh.T) * (1 - np.tanh(hprev) * np.tanh(hprev)) * np.exp(hprev) * -targets[t]
        dhprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. hidden state
        dhprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. previous hidden state
        dhprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. inputs
        dxprev = np.dot(dhprev, Wx)
        dxprev *= -1.0 / len(targets)

        # Compute gradients w.r.t. weights and bias
        dWx += np.dot(xprev, dxprev).T
        dhprev = np.dot(dxprev, Wh.T) * (1 - np.tanh(hprev) * np.tanh(hprev)) * np.exp(hprev) * -targets[t]
        dhprev *= -1.0 / len(targets)

    # Average over steps
    dWx /= len(inputs)
    dWh /= len(inputs)
    db /= len(inputs)
    dhprev /= len(inputs)

    return loss, dWx, dWh, db, outputs, hprev, np.argmax(hprev, 1)
```

1

///

Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)

BSD License

///

4

import numpy as np

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

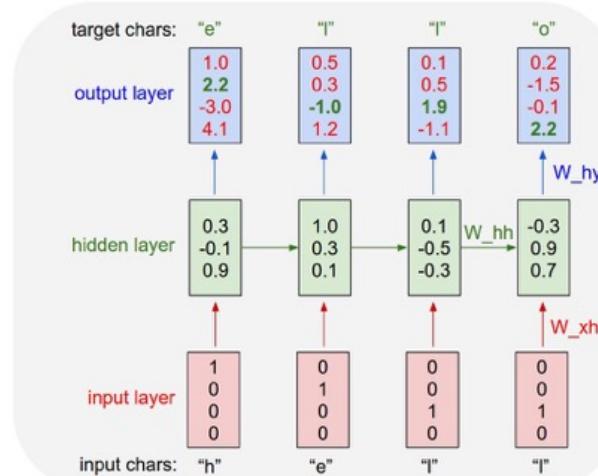
n-char-rnn.py gist

```
1 #include <assert.h>
2 #include <math.h>
3 #include <limits.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <sys/mman.h>
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <sys/mman.h>
11 #include <sys/types.h>
12 #include <sys/stat.h>
13 #include <sys/mman.h>
14 #include <sys/types.h>
15 #include <sys/stat.h>
16 #include <sys/mman.h>
17 #include <sys/types.h>
18 #include <sys/stat.h>
19 #include <sys/mman.h>
20 #include <sys/types.h>
21 #include <sys/stat.h>
22 #include <sys/mman.h>
23 #include <sys/types.h>
24 #include <sys/stat.h>
25 #include <sys/mman.h>
26 #include <sys/types.h>
27 #include <sys/stat.h>
28 #include <sys/mman.h>
29 #include <sys/types.h>
30 #include <sys/stat.h>
31 #include <sys/mman.h>
32 #include <sys/types.h>
33 #include <sys/stat.h>
34 #include <sys/mman.h>
35 #include <sys/types.h>
36 #include <sys/stat.h>
37 #include <sys/mman.h>
38 #include <sys/types.h>
39 #include <sys/stat.h>
40 #include <sys/mman.h>
41 #include <sys/types.h>
42 #include <sys/stat.h>
43 #include <sys/mman.h>
44 #include <sys/types.h>
45 #include <sys/stat.h>
46 #include <sys/mman.h>
47 #include <sys/types.h>
48 #include <sys/stat.h>
49 #include <sys/mman.h>
50 #include <sys/types.h>
51 #include <sys/stat.h>
52 #include <sys/mman.h>
53 #include <sys/types.h>
54 #include <sys/stat.h>
55 #include <sys/mman.h>
56 #include <sys/types.h>
57 #include <sys/stat.h>
58 #include <sys/mman.h>
59 #include <sys/types.h>
60 #include <sys/stat.h>
61 #include <sys/mman.h>
62 #include <sys/types.h>
63 #include <sys/stat.h>
64 #include <sys/mman.h>
65 #include <sys/types.h>
66 #include <sys/stat.h>
67 #include <sys/mman.h>
68 #include <sys/types.h>
69 #include <sys/stat.h>
70 #include <sys/mman.h>
71 #include <sys/types.h>
72 #include <sys/stat.h>
73 #include <sys/mman.h>
74 #include <sys/types.h>
75 #include <sys/stat.h>
76 #include <sys/mman.h>
77 #include <sys/types.h>
78 #include <sys/stat.h>
79 #include <sys/mman.h>
80 #include <sys/types.h>
81 #include <sys/stat.h>
82 #include <sys/mman.h>
83 #include <sys/types.h>
84 #include <sys/stat.h>
85 #include <sys/mman.h>
86 #include <sys/types.h>
87 #include <sys/stat.h>
88 #include <sys/mman.h>
89 #include <sys/types.h>
90 #include <sys/stat.h>
91 #include <sys/mman.h>
92 #include <sys/types.h>
93 #include <sys/stat.h>
94 #include <sys/mman.h>
95 #include <sys/types.h>
96 #include <sys/stat.h>
97 #include <sys/mman.h>
98 #include <sys/types.h>
99 #include <sys/stat.h>
```

```
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
26
27 def UnrollInputs(inputs, targets, hprev):
28     inputs, targets = inputs.tolist(), targets.tolist()
29     inputs[0] = np.zeros((vocab_size, 1))
30     return inputs, targets, hprev
31
32 def loss(inputs, targets, hprev, Wxh, Whh, Why, bh, by):
33     inputs, targets = inputs.tolist(), targets.tolist()
34     inputs[0] = np.zeros((vocab_size, 1))
35     loss = 0.0
36     for t in range(len(inputs)):
37         x = np.array(inputs[t]).reshape(1, -1)
38         y = np.array(targets[t]).reshape(1, -1)
39         hprev = forward(x, hprev, Wxh, Whh, Why, bh, by)
40         loss += -np.log(hprev[y[0], 0])
41     return loss
42
43 def forward(x, hprev, Wxh, Whh, Why, bh, by):
44     inputs, targets = inputs.tolist(), targets.tolist()
45     inputs[0] = np.zeros((vocab_size, 1))
46     hprev = np.zeros((hidden_size, 1))
47     hprev = np.tanh(np.dot(Wxh, inputs[0]) + np.dot(Whh, hprev) + bh)
48     for t in range(1, len(inputs)):
49         x = np.array(inputs[t]).reshape(1, -1)
50         hprev = np.tanh(np.dot(Wxh, x) + np.dot(Whh, hprev) + bh)
51     hprev = np.dot(Why, hprev) + by
52     return hprev
53
54 def backward(x, y, hprev, Wxh, Whh, Why, bh, by):
55     inputs, targets = inputs.tolist(), targets.tolist()
56     inputs[0] = np.zeros((vocab_size, 1))
57     inputs[-1] = np.zeros((vocab_size, 1))
58     hprev = np.zeros((hidden_size, 1))
59     dWxh, dWhh, dWhy, dbh, dby = np.zeros_like(Wxh), np.zeros_like(Whh),
60     dbh = np.zeros_like(bh), dby = np.zeros_like(by)
61     dhprev = np.zeros_like(hprev)
62     for t in reversed(range(len(inputs) - 1)):
63         dx = np.tanh(hprev) * (1 - np.tanh(hprev)**2)
64         dhprev = np.dot(Why.T, dhprev) + dx
65         dhprev *= np.dot(Whh.T, dhprev) + dWxh*x + dbh
66         dhprev *= np.dot(Wxh.T, dhprev) + dWhh*x + dby
67         dWxh += np.outer(x.T, dhprev)
68         dWhh += np.outer(hprev.T, dhprev)
69         dbh += dhprev
70         dby += dhprev
71     dWxh /= len(inputs) - 1
72     dWhh /= len(inputs) - 1
73     dbh /= len(inputs) - 1
74     dby /= len(inputs) - 1
75     dWxh, dWhh, dbh, dby = dWxh.T, dWhh.T, dbh.T, dby.T
76     return dWxh, dWhh, dbh, dby
77
78 def sample(hprev, Wxh, Whh, Why, bh, by, temp=1.0):
79     inputs, targets = inputs.tolist(), targets.tolist()
80     inputs[0] = np.zeros((vocab_size, 1))
81     hprev = np.zeros((hidden_size, 1))
82     hprev = forward(x, hprev, Wxh, Whh, Why, bh, by)
83     if temp == 0:
84         p = np.argmax(hprev)
85     else:
86         p = np.exp(hprev / temp)
87         p = p / np.sum(p)
88     sample_ix = np.random.choice(range(vocab_size), p=p)
89     targets[0] = sample_ix
90     inputs[0] = np.zeros((vocab_size, 1))
91     inputs[sample_ix] = 1
92     return sample_ix, targets[0]
93
94 def sample2(hprev, Wxh, Whh, Why, bh, by, temp=1.0):
95     inputs, targets = inputs.tolist(), targets.tolist()
96     inputs[0] = np.zeros((vocab_size, 1))
97     hprev = np.zeros((hidden_size, 1))
98     hprev = forward(x, hprev, Wxh, Whh, Why, bh, by)
99     if temp == 0:
100        p = np.argmax(hprev)
101    else:
102        p = np.exp(hprev / temp)
103        p = p / np.sum(p)
104    sample_ix = np.argmax(p)
105    targets[0] = sample_ix
106    inputs[0] = np.zeros((vocab_size, 1))
107    inputs[sample_ix] = 1
108    return sample_ix, targets[0]
```

Initializations

recall:



n-char-rnn.py gist

```
#!/usr/bin/python
# character-level Vanilla RNN Model. Written by Andrej Karpathy (@karpathy)

# This file is part of my neural net series. Check out the first post at
# http://karpathy.github.io/2015/05/21/rnn-tutorial-part-1/
# and follow along for a full implementation of a recurrent neural network in Python.

# MIT License
# Copyright (c) 2015 Andrej Karpathy

# Imports
import numpy as np
from __future__ import print_function

# Data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(data)
vocab_size = len(chars)
print('Data size: %d' % len(data), 'vocab size: %d' % vocab_size)

#常数
seq_length = 200 # number of steps to unroll the RNN for
batch_size = 100 # batch size for LSTMs to handle
backtrack = 10 # look back for LSTMs to handle

# Hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 200 # number of steps to unroll the RNN for
learning_rate = 1e-1

# Model parameters
Wxh = np.random.randn(vocab_size, hidden_size) / np.sqrt(hidden_size)
Whh = np.random.randn(hidden_size, hidden_size) / np.sqrt(hidden_size)
Why = np.random.randn(hidden_size, vocab_size) / np.sqrt(vocab_size)
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

# Inputs/outputs, targets, hprev
inputs = []
targets = []

# Inputs/outputs are both lists of integers.
# inputs is 2D array of 1000 hidden state
# returns the loss, gradients on model parameters, and last hidden state
def forward(x, y, p=0, h0=D, D=D):
    hprev = h0
    for i in range(len(x)):
        if i > 0:
            inputs.append(hprev)
        hprev = np.tanh(np.dot(x[i], Wxh) + np.dot(hprev, Whh) + bh[i-1] + p)
        ypred = np.dot(Why, hprev) + by
        yprob = softmax(ypred)
        inputs.append(ypred)
        if i < seq_length - 1:
            targets.append(int(chars[i+1]))
        else:
            targets.append(int(chars[0])) # wrap-around
        dy = np.zeros_like(yprob)
        dy[y] = 1
        dypred = dy * np.log(ypred) / seq_length # softmax cross-entropy loss
        dypred = -dypred / seq_length # backward pass through softmax
        dby = np.sum(dypred) # backward pass through by
        dby *= dy
        dbh = np.sum(dypred * hprev) # backward pass through hprev
        dbh *= dy
        dWxh = np.sum(dypred * x[i]) # backward pass through Wxh
        dWxh *= dy
        dWhh = np.sum(dypred * hprev) # backward pass through Whh
        dWhh *= dy
        dh = np.sum(dypred * Why) # backward pass through Why
        dh *= dy
        dWhy = np.sum(dh) # backward pass through Why
        dWhy *= dy
        db = np.sum(dh * hprev) # backward pass through hprev
        db *= dy
        dhprev = np.sum(dh * Why) # backward pass through Why
        dhprev *= dy
        dhprev *= -1 # backward pass through hprev
        dWxh += dhprev * x[i]
        dWhh += dhprev * hprev
        dWhy += dhprev
        dby += dypred
        dbh += dhprev
        dh += dhprev
    return hprev, inputs, targets, dWxh, dWhh, dWhy, db, dby, hprev

# 训练部分
def sample(hprev, inputs, targets, p=0, n=200):
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print('----\n%s\n----' % (txt,))

# 计算损失
def 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))
    return smooth_loss

# 训练部分
for i in range(backtrack):
    inputs, targets, hprev = forward(chars[0:batch_size], np.zeros(batch_size))
    smooth_loss = np.log(vocab_size) * batch_size / seq_length # loss at iteration 0
    while True:
        # forward seq_length characters from left to right (unroll the RNN)
        for j in range(seq_length):
            inputs, targets, hprev = forward(chars[j:j+batch_size], hprev)
        loss = np.sum(-np.log(targets)) / batch_size # loss at iteration n
        if loss < 0.001:
            break
        inputs = np.array(inputs) # for unroll the RNN
        targets = np.array(targets) # for unroll the RNN
        # backward pass through the net
        for t in range(seq_length-1, -1, -1):
            inputs[t] = np.argmax(inputs[t])
            targets[t] = np.argmax(targets[t])
        inputs = np.array(inputs)
        targets = np.array(targets)
        # backward pass through the net
        hprev, inputs, targets, dWxh, dWhh, dWhy, db, dby, hprev = forward(chars[0:batch_size], np.zeros(batch_size))
        smooth_loss = smooth_loss + loss / seq_length # print progress
        if loss < 0.001:
            break
    # 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

```
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+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, 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
```

n-char-rnn.py gist

```
#!/usr/bin/python
# (c) Andrej Karpathy, 2015. Released under MIT license. Written by Andrej Karpathy (@karpathy)
# See License

import numpy as np

# a data step
def data_step(data, data_size, seq_length):
    #assert len(data) == seq_length, "Input net", "len(data) should be sample size but %d" % len(data)
    #assert data_size >= seq_length, "Input net", "data_size must be at least as large as seq_length"
    #assert len(data) <= data_size, "Input net", "data_size must be at least as large as seq_length"
    #assert len(data) >= 1, "Input net", "seq_length must be at least 1"
    #assert len(data) <= data_size, "Input net", "seq_length must be at most data_size"
    #assert len(data) >= 1, "Input net", "data[0] must be a character"
    #assert len(data) >= 1, "Input net", "data[-1] must be a character"

    # Hyperparameters
    hidden_size = 100 # size of hidden layer of neurons
    seq_length = 2 # number of steps to unroll the net for
    num_steps = 10 # number of steps to unroll the net for
    num_epochs = 10 # number of epochs to train for
    learning_rate = 1e-1

    # model parameters
    Wxh = np.random.randn(hidden_size, vocab_size).astype(np.float32) # weight from inputs to hidden
    Whh = np.random.randn(hidden_size, hidden_size).astype(np.float32) # weight from hidden to hidden
    Why = np.random.randn(vocab_size, hidden_size).T.astype(np.float32) # weight from hidden to output
    bh = np.zeros((hidden_size,)).astype(np.float32) # hidden bias
    by = np.zeros((vocab_size,)).astype(np.float32) # output bias

    # Inputs/outputs, targets, hprev
    inputs, targets = [], []
    inputs, targets = [list() for _ in range(seq_length)]
    hprev = np.zeros((hidden_size,))

    # forward pass
    for t in xrange(len(data)-1):
        inputs[t] = np.array([ord(c) for c in data[t:t+seq_length]])
        hprev = np.tanh(np.dot(Wxh, inputs[t]) + np.dot(Whh, hprev) + bh)

    # softmax and loss
    outputs = np.exp(hprev * Why) / np.sum(np.exp(hprev * Why)) # probabilities for next chars
    loss = -np.log(outputs[np.argmax(data[1:t+1])]) # softmax cross-entropy loss

    # backward pass
    dWxh, dWhh, dWhy, dhprev, dbh, dy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why),
    dby, dh = np.zeros_like(inputs[0]), np.zeros_like(hprev)
    dbh = np.zeros_like(bh)
    dyt = np.zeros_like(data[1:t+1])
    for t in reversed(xrange(1, len(data))):
        dWhy += np.dot(dy, np.tanh(hprev).T) # backprop into Why
        dbh += np.sum(dy * np.tanh(hprev), axis=0) # backprop into bh
        dy = np.dot(Why, dy) + dh # backprop into dy
        dh = np.dot(Whh.T, dy) # backprop into dh
        dh += dbh # backprop through tanh nonlinearity
        dWhh += np.dot(dh, hprev.T) # backprop into Whh
        dbh = np.dot(Whh, dh) # backprop into dbh
        dh = np.tanh(hprev) * dh # backprop through tanh nonlinearity
        dyt[t] = dy # store dy for gradient clipping
        dy = np.zeros_like(dy) # reset dy for next time step
    dyt[0] = dy # store dy for gradient clipping

    # clip gradients
    if np.max(np.abs(dWxh)) > 500: dWxh *= 500 / np.max(np.abs(dWxh))
    if np.max(np.abs(dWhh)) > 500: dWhh *= 500 / np.max(np.abs(dWhh))
    if np.max(np.abs(dWhy)) > 500: dWhy *= 500 / np.max(np.abs(dWhy))
    if np.max(np.abs(dbh)) > 500: dbh *= 500 / np.max(np.abs(dbh))
    if np.max(np.abs(dy)) > 500: dy *= 500 / np.max(np.abs(dy))

    # 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, 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

```
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+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, 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

n-char-rnn.py gist

```

31 n, p = 0, 0
32 mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
33 mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
34 smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
35 while True:
36     # prepare inputs (we're sweeping from left to right in steps seq_length long)
37     if p+seq_length+1 >= len(data) or n == 0:
38         hprev = np.zeros((hidden_size,1)) # reset RNN memory
39         p = 0 # go from start of data
40     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
41     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
42
43     # sample from the model now and then
44     if n % 100 == 0:
45         sample_ix = sample(hprev, inputs[0], 200)
46         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
47         print '----\n%s\n----' % (txt, )
48
49     # forward seq_length characters through the net and fetch gradient
50     loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
51     smooth_loss = smooth_loss * 0.999 + loss * 0.001
52     if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress
53
54     # perform parameter update with Adagrad
55     for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
56                                   [dWxh, dWhh, dWhy, dbh, dby],
57                                   [mWxh, mWhh, mWhy, mbh, mby]):
58         mem += dparam * dparam
59         param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update
60
61     p += seq_length # move data pointer
62     n += 1 # iteration counter

```



n Andrej Karpathy
1n

n-char-rnn.py gist

```
1 #include <math.h>
2 #include <assert.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <sys/malloc.h>
6 #include <sys/conf.h>
7 #include <stropts.h>
8 #include <stropts.h>
9 #include <stropts.h>
10 #include <stropts.h>
11 #include <stropts.h>
12 #include <stropts.h>
13 #include <stropts.h>
14 #include <stropts.h>
15 #include <stropts.h>
16 #include <stropts.h>
17 #include <stropts.h>
18 #include <stropts.h>
19 #include <stropts.h>
20 #include <stropts.h>
21 #include <stropts.h>
22 #include <stropts.h>
23 #include <stropts.h>
24 #include <stropts.h>
25 #include <stropts.h>
26 #include <stropts.h>
27 #include <stropts.h>
28 #include <stropts.h>
29 #include <stropts.h>
30 #include <stropts.h>
31 #include <stropts.h>
32 #include <stropts.h>
33 #include <stropts.h>
34 #include <stropts.h>
35 #include <stropts.h>
36 #include <stropts.h>
37 #include <stropts.h>
38 #include <stropts.h>
39 #include <stropts.h>
40 #include <stropts.h>
41 #include <stropts.h>
42 #include <stropts.h>
43 #include <stropts.h>
44 #include <stropts.h>
45 #include <stropts.h>
46 #include <stropts.h>
47 #include <stropts.h>
48 #include <stropts.h>
49 #include <stropts.h>
50 #include <stropts.h>
51 #include <stropts.h>
52 #include <stropts.h>
53 #include <stropts.h>
54 #include <stropts.h>
55 #include <stropts.h>
56 #include <stropts.h>
57 #include <stropts.h>
58 #include <stropts.h>
59 #include <stropts.h>
60 #include <stropts.h>
61 #include <stropts.h>
62 #include <stropts.h>
63 #include <stropts.h>
64 #include <stropts.h>
65 #include <stropts.h>
66 #include <stropts.h>
67 #include <stropts.h>
68 #include <stropts.h>
69 #include <stropts.h>
70 #include <stropts.h>
71 #include <stropts.h>
72 #include <stropts.h>
73 #include <stropts.h>
74 #include <stropts.h>
75 #include <stropts.h>
76 #include <stropts.h>
77 #include <stropts.h>
78 #include <stropts.h>
79 #include <stropts.h>
80 #include <stropts.h>
81 #include <stropts.h>
82 #include <stropts.h>
83 #include <stropts.h>
84 #include <stropts.h>
85 #include <stropts.h>
86 #include <stropts.h>
87 #include <stropts.h>
88 #include <stropts.h>
89 #include <stropts.h>
90 #include <stropts.h>
91 #include <stropts.h>
92 #include <stropts.h>
93 #include <stropts.h>
94 #include <stropts.h>
95 #include <stropts.h>
96 #include <stropts.h>
97 #include <stropts.h>
98 #include <stropts.h>
99 #include <stropts.h>
100 #include <stropts.h>
101 #include <stropts.h>
102 #include <stropts.h>
103 #include <stropts.h>
104 #include <stropts.h>
105 #include <stropts.h>
106 #include <stropts.h>
107 #include <stropts.h>
108 #include <stropts.h>
109 #include <stropts.h>
110 #include <stropts.h>
111 #include <stropts.h>
112 #include <stropts.h>
```

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
```

n-char-rnn.py gist

```
#!/usr/bin/python
# character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)

# This file is part of my "Deep learning" course at Stanford University.
# License: https://github.com/karpathy/nlp-stanford-courses-fall-2015/blob/master/LICENSE
# If you use this code or any part of this code, please cite us as:
# Andrej Karpathy et al., "Deep learning for NLP", in ICLR 2016.

# Model parameters
Wxh = np.random.randn(256, 128) * np.sqrt(0.01)
Whh = np.random.randn(128, 128) * np.sqrt(0.01)
Why = np.random.randn(128, 26) * np.sqrt(0.01)
bh = np.zeros(128) # hidden bias
by = np.zeros(26) # output bias

# Input/outputs
hidden_size = 128 # size of hidden layer of neurons
seq_length = 20 # number of steps to unroll the net for generating text
n = 0 # iteration counter

# Model parameters
hidden_size = 128; vocab_size = 26; seq_length = 20; n = 0
Wxh = np.random.randn(hidden_size, hidden_size) * np.sqrt(0.01)
Whh = np.random.randn(hidden_size, hidden_size) * np.sqrt(0.01)
Why = np.random.randn(hidden_size, vocab_size) * np.sqrt(0.01)
bh = np.zeros(hidden_size); by = np.zeros(vocab_size)

# Inputs/outputs, targets, hprev
inputs = np.zeros((seq_length, hidden_size)) # hidden state
targets = np.zeros(seq_length) # targets, i.e. outputs from previous step
hprev = np.zeros(hidden_size) # previous hidden state

# Forward pass
for t in range(seq_length):
    inputs[t] = np.zeros(hidden_size) + encode_one_of_n(representation)
    hidden_t = np.tanh(np.dot(inputs[t], Wxh) + np.dot(hidden_t, Whh) + bh) # hidden state
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t)) # softmax for next chars
    inputs[t+1] = np.dot(hidden_t, Why) + by # softmax (cross-entropy loss)
    hidden_t = np.tanh(np.dot(inputs[t+1], Wxh) + np.dot(hidden_t, Whh) + bh)
    dloss_dxh = np.dot(dloss_dyh, hidden_t) * np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    dloss_dhh = np.dot(dloss_dyh, hidden_t) * np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    dloss_dyh = np.sum(dloss_dyh) # backward pass y
    dloss_dbh = np.sum(dloss_dyh) # backward pass h
    dloss_dyh = dloss_dyh / seq_length # average gradient
    dloss_dxh = dloss_dxh / seq_length # average gradient
    dloss_dhh = dloss_dhh / seq_length # average gradient
    hidden_t = np.tanh(np.dot(inputs[t], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+1], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+2], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+3], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+4], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+5], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+6], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+7], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+8], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+9], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+10], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+11], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+12], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+13], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+14], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+15], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+16], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+17], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+18], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+19], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))

# Initialize memory states for first time step
hprev = np.zeros(hidden_size) # hidden state, used later for first time step
ix_to_char = {i: chr(96+i) for i in range(25)} # 96 = 'a' - 1
char_to_ix = {c: i for i, c in ix_to_char.items()}

# Forward pass
for t in range(seq_length):
    if t < seq_length-1:
        inputs[t+1] = np.zeros(hidden_size) + encode_one_of_n(representation)
    hidden_t = np.tanh(np.dot(inputs[t], Wxh) + np.dot(hprev, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+1], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+2], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+3], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+4], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+5], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+6], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+7], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+8], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+9], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+10], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+11], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+12], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+13], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+14], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+15], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+16], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+17], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+18], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))
    hidden_t = np.tanh(np.dot(inputs[t+19], Wxh) + np.dot(hidden_t, Whh) + bh)
    hidden_t = np.exp(hidden_t) / np.sum(np.exp(hidden_t))

# Compute loss and fetch gradient via backprop
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

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

Main loop

n-char-rnn.py gist

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>
#include <sys/cmn.h>
#include <sys/malloc.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>
#include <sys/cmn.h>
#include <sys/malloc.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>
#include <sys/cmn.h>
#include <sys/malloc.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>
#include <sys/cmn.h>
#include <sys/malloc.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>
#include <sys/cmn.h>
```

Andrey Karpathy

The code is a Python script for an n-gram language model. It defines a forward pass (lossFun) and a backward pass (backward). The forward pass encodes inputs into a one-hot representation, calculates hidden states and softmax probabilities, and computes cross-entropy loss. The backward pass computes gradients for weights and biases, and mitigates exploding gradients using gradient clipping.



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
    for t in xrange(len(inputs)):
        xs[t] = np.zeros([vocab_size, 1]) # encode in 1-of-k representation
        xs[t][inputs[t]] = 1
        hs[t] = np.tanh(np.dot(whh, xs[t]) + np.dot(bh, hs[t-1]) + bh) # hidden state
        ys[t] = np.exp(np.dot(why, hs[t])) / np.sum(np.exp(ys[t])) # 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)

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

    # backward: compute gradients going backwards
    dwhh, dwhy, dby = np.zeros_like(whh), np.zeros_like(whh), np.zeros_like(why)
    dbh, dyb = 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
        dhy = np.dot(dy, hs[t].T)
        dyb += dhy
        dhh = np.dot(dy, hs[t-1].T) + dhnext # backprop into h
        ddraw = (1 - hs[t]**2) * hs[t] * dh # backprop through tanh nonlinearity
        dbh += ddraw
        dwhh += np.dot(ddraw, xs[t].T)
        dhnext = np.dot(why.T, ddraw)

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

    return loss, dwhh, dwhy, dbh, dyb, hs[len(inputs)-1]
```

n Andrej Karpathy
1n

cor

n-char-rnn.py gist

```
MIT License
Copyright (c) 2014 Andrej Karpathy (karpathy)
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE OF DATA; OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

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

    # backward pass: compute gradients going backwards
    dws, dwh, dbh = np.zeros((vocab_size,hidden_size)), np.zeros((hidden_size,hidden_size)), np.zeros((hidden_size))
    dhs, dhy = np.zeros((hidden_size,hidden_size)), np.zeros((hidden_size))
    ddot = np.zeros((hidden_size,hidden_size))
    dh0 = np.zeros((hidden_size))
    for t in reversed(xrange(len(inputs))):
        ddot = np.multiply(dhy, np.exp(ys[t])) * np.exp(ys[t])
        ddot -= np.multiply(ddot, ps[t])
        ddot *= dy
        dws += np.dot(xs[t].T, ddot) # direct gradient through tanh
        dhs[t] = ddot * np.dot(Whh.T, ddot) # backprop through tanh nonlinearity
        dhs[t-1] = ddot * np.dot(Whh.T, ddot)
        dhy += np.dot(dhs[t].T, ddot)
        dbh += np.sum(ddot, axis=0)
    for param in (dws, dwh, dbh, dhs, dhy):
        np.clip(param, -5, 5, out=param) # clip gradients to mitigate exploding/ vanishing gradients
    return loss, dws, dwh, dbh, dhs, dhy, ddot, ps[t][targets[t],0]
```

```
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)):
        xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
        xs[t][inputs[t]] = 1
        hs[t] = np.tanh(np.dot(Whh, 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)
```

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

Softmax classifier

n-char-rnn.py gist

```
1 #include <math.h>
2 #include <assert.h>
3 #include <math.h>
4 #include <math.h>
5 #include <math.h>
6 #include <math.h>
7 #include <math.h>
8 #include <math.h>
9 #include <math.h>
10 #include <math.h>
11 #include <math.h>
12 #include <math.h>
13 #include <math.h>
14 #include <math.h>
15 #include <math.h>
16 #include <math.h>
17 #include <math.h>
18 #include <math.h>
19 #include <math.h>
20 #include <math.h>
21 #include <math.h>
22 #include <math.h>
23 #include <math.h>
24 #include <math.h>
25 #include <math.h>
26 #include <math.h>
27 #include <math.h>
28 #include <math.h>
29 #include <math.h>
30 #include <math.h>
31 #include <math.h>
32 #include <math.h>
33 #include <math.h>
34 #include <math.h>
35 #include <math.h>
36 #include <math.h>
37 #include <math.h>
38 #include <math.h>
39 #include <math.h>
40 #include <math.h>
41 #include <math.h>
42 #include <math.h>
43 #include <math.h>
44 #include <math.h>
45 #include <math.h>
46 #include <math.h>
47 #include <math.h>
48 #include <math.h>
49 #include <math.h>
50 #include <math.h>
51 #include <math.h>
52 #include <math.h>
53 #include <math.h>
54 #include <math.h>
55 #include <math.h>
56 #include <math.h>
57 #include <math.h>
58 #include <math.h>
59 #include <math.h>
60 #include <math.h>
61 #include <math.h>
```

```
def forward(x, targets, hprev):
    """Inputs: inputs is a list of integers, targets is a list of integers,
    hprev is the previous hidden state. Returns the loss, gradients on model
    parameters, and last hidden state
    """
    xs, hs, ys, ps, dhnext = [], [], [], [], None
    nhs, nh = len(hprev), hprev.shape[0]
    dWxh, dWhh, dWhy = np.zeros_like(x), np.zeros_like(hprev), np.zeros_like(hprev)
    dbh, dby = np.zeros_like(hprev), np.zeros_like(hprev)
    dhnxt = np.zeros_like(hprev)

    for t in xrange(len(inputs)):
        x = np.array([inputs[t]])
        h = np.tanh(np.dot(x, D) + np.dot(hprev, D))
        y = np.exp(h) / np.sum(np.exp(h)) * softmax_weights
        p = np.log(y) - np.log(np.sum(np.exp(h)))
        loss += -np.sum(targets[t] * p)

        xs.append(x)
        hs.append(h)
        ys.append(y)
        ps.append(p)

        dWxh += np.outer(x, h)
        dWhh += np.outer(hprev, h)
        dWhy += np.outer(y, np.ones_like(y))

        dbh += h * dy
        dby += np.exp(h) / np.sum(np.exp(h)) * softmax_weights * dy
        dhnxt = np.dot(Whh.T, dhnext) + np.sum(dy * np.exp(h) / np.sum(np.exp(h)) * softmax_weights)

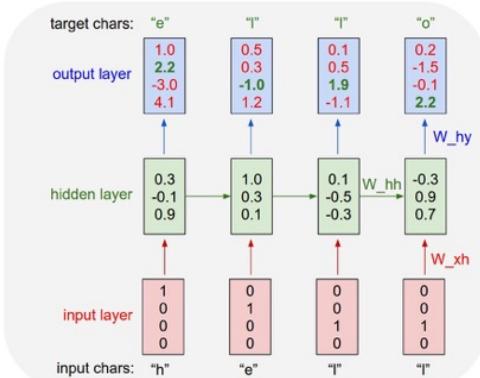
        dhnext = dhnxt * (1 - h * h)

    loss /= len(inputs)
    dWxh /= len(inputs)
    dWhh /= len(inputs)
    dWhy /= len(inputs)
    dbh /= len(inputs)
    dby /= len(inputs)

    return loss, dWxh, dWhh, dWhy, dbh, dby, hs[-1]
```

```
44     # backward pass: compute gradients going backwards
45     dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
46     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
47     dhnxt = np.zeros_like(hs[0])
48     for t in reversed(xrange(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) + dhnxt # backprop into h
54         ddraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
55         dbh += ddraw
56         dWxh += np.dot(ddraw, xs[t].T)
57         dWhh += np.dot(ddraw, hs[t-1].T)
58         dhnxt = np.dot(Whh.T, ddraw)
59     for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
60         np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
61     return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

recall:



n-char-rnn.py gist

```

n-char-rnn.py gist
=====
Mitsubishi character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
...
import numpy as np

# a data API
data = {}
data["train"] = "train.txt", "#!/usr/bin/python3 # should be simple plain text file"
data["test"] = "test.txt", "#!/usr/bin/python3"
data["size"] = vocab_size = len(data), len(data)
print("data has %d words, or tokens. vocab size: %d" % (data["size"], vocab_size))
for i in range(1, 5):
    print(data["train"][:i]) # Print first i words of train to screen
else:
    for i in range(1, 5):
        print(data["test"][:i]) # Print first i words of test to screen

# Hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 20 # number of steps to unroll the RNN for
learning_rate = 1.0 # learning rate
# model parameters
Wxh = np.random.randn(vocab_size, hidden_size) * np.sqrt(0.01)
Why = np.random.randn(hidden_size, hidden_size) * np.sqrt(0.01)
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

# Input/outputs, targets, arrays
inputs, targets = [], []
# Prints targets are both list of integers,
# Arrays are 2D array of VEC1000 hidden state
# Returns the loss, gradients on model parameters, and last hidden state
def loss_hyx(h, x, y):
    h1, y1, ys = h[0], x[0], y[0]
    if h1 == y1:
        return 0.0, np.zeros((vocab_size, hidden_size)), np.zeros((hidden_size, hidden_size)), bh
    else:
        return 1.0, np.zeros((vocab_size, hidden_size)), np.zeros((hidden_size, hidden_size)), bh

def loss_hhx(h, h1):
    if h1 == h[0]:
        return 0.0, np.zeros((vocab_size, hidden_size)), np.zeros((hidden_size, hidden_size)), bh
    else:
        return 1.0, np.zeros((vocab_size, hidden_size)), np.zeros((hidden_size, hidden_size)), bh

def forward_pass():
    for t in xrange(len(inputs)):
        if t >= seq_length:
            break
        h1, y1, ys = loss_hyx(h, inputs[t], targets[t])
        if y1 != -1: # don't care for -1, it's a blank
            h = tanh(np.dot(Wxh, x[t]) + np.dot(Why, h1) + bh) # tanh non-linearity
            y = dot(Why, h) + by # linear activation
            p = exp(y) / sum(exp(y)) # softmax (cross-entropy loss)
            loss += -log(p[y1]) # negative log-likelihood loss
            dhy = -exp(y) * delta(t, y1) # derivative w.r.t. output y
            dwhy = -x[t] * exp(y) * delta(t, y1) # derivative w.r.t. Why
            dWxh = -x[t] * exp(y) * delta(t, y1) # derivative w.r.t. Wxh
            dh1 = np.multiply(dhy, Why) # backprop through tanh nonlinearity
            dhx = np.multiply(dhy, Wxh) # backprop through dot product
            dWhy += np.outer(dhy, h1) # accumulate gradients for next time
            dWxh += np.outer(dhx, x[t]) # accumulate gradients for next time
            delta[t] = np.multiply(dhy, Why) # accumulate gradients for next time
            delta[t] = np.multiply(delta[t], np.exp(-y)) # scale down delta
            delta[t] = np.multiply(delta[t], 1.0 - np.multiply(h, h)) # scale down delta
            h = h + delta[t] # gradient descent
            h = h1 + dhx # gradient through dot product
            Why = Why - lr * dWhy # gradient descent on Why
            Wxh = Wxh - lr * dWxh # gradient descent on Wxh
            # For gradient clipping
            if np.linalg.norm(Wxh) > max_norm:
                Wxh *= max_norm / np.linalg.norm(Wxh)
            if np.linalg.norm(Why) > max_norm:
                Why *= max_norm / np.linalg.norm(Why)
            if np.linalg.norm(bh) > max_norm:
                bh *= max_norm / np.linalg.norm(bh)
            if np.linalg.norm(by) > max_norm:
                by *= max_norm / np.linalg.norm(by)

        else:
            h = np.zeros((hidden_size, 1)) # reset hidden state
            h1 = h
            y1 = -1
            ys = -1

    for t in xrange(len(inputs), seq_length):
        h1, y1, ys = loss_hhx(h, inputs[t])
        if y1 != -1: # don't care for -1, it's a blank
            h = tanh(np.dot(Wxh, x[t]) + np.dot(Why, h1) + bh) # tanh non-linearity
            y = dot(Why, h) + by # linear activation
            p = exp(y) / sum(exp(y)) # softmax (cross-entropy loss)
            loss += -log(p[y1]) # negative log-likelihood loss
            dhy = -exp(y) * delta(t, y1) # derivative w.r.t. output y
            dwhy = -x[t] * exp(y) * delta(t, y1) # derivative w.r.t. Why
            dWxh = -x[t] * exp(y) * delta(t, y1) # derivative w.r.t. Wxh
            dh1 = np.multiply(dhy, Why) # backprop through tanh nonlinearity
            dhx = np.multiply(dhy, Wxh) # backprop through dot product
            dWhy += np.outer(dhy, h1) # accumulate gradients for next time
            dWxh += np.outer(dhx, x[t]) # accumulate gradients for next time
            delta[t] = np.multiply(dhy, Why) # accumulate gradients for next time
            delta[t] = np.multiply(delta[t], np.exp(-y)) # scale down delta
            delta[t] = np.multiply(delta[t], 1.0 - np.multiply(h, h)) # scale down delta
            h = h + delta[t] # gradient descent
            h = h1 + dhx # gradient through dot product
            Why = Why - lr * dWhy # gradient descent on Why
            Wxh = Wxh - lr * dWxh # gradient descent on Wxh
            # For gradient clipping
            if np.linalg.norm(Wxh) > max_norm:
                Wxh *= max_norm / np.linalg.norm(Wxh)
            if np.linalg.norm(Why) > max_norm:
                Why *= max_norm / np.linalg.norm(Why)
            if np.linalg.norm(bh) > max_norm:
                bh *= max_norm / np.linalg.norm(bh)
            if np.linalg.norm(by) > max_norm:
                by *= max_norm / np.linalg.norm(by)

    # forward pass for the model now and then
    if n < 100000:
        sample_ix = sample(h, seed_ix, n)
        print("got %d samples" % len(sample_ix))
    else:
        print("got 100000 samples for us to sample...")

    # print progress
    if t % 10000 == 0:
        print("t = %d" % t)
        print("loss = %f" % loss)

    if t % 50000 == 0:
        for i in range(0, seq_length):
            data["samples"] = sample_ix[i:i+seq_length]
            with open(data["samples"] + ".txt", "w") as f:
                f.write("\n".join(data["samples"]))
            print("t = %d, wrote sequence %d to file" % (t, i))
        else:
            print("done!")

# forward pass through the network to predict the next character
def parse_tiny():
    inputs, data, delta, why, bh, by, hs, dhs, dwhy, ds, loss, targets, x, y, p, s, n = np.zeros((vocab_size, 1)), np.zeros((vocab_size, hidden_size)), np.zeros((hidden_size, hidden_size)), np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1)), np.zeros((vocab_size, hidden_size)), np.zeros((hidden_size, hidden_size)), np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1)), np.zeros((hidden_size, 1)), np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1)), np.zeros((hidden_size, 1)), np.zeros((hidden_size, 1)), np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1)), np.zeros((hidden_size, 1)), np.zeros((hidden_size, 1)), np.zeros((hidden_size, 1)), np.zeros((vocab_size, 1)), np.zeros((hidden_size, 1))

    for i in range(0, seq_length):
        if i >= data_size:
            break
        data["samples"] = sample_ix[i:i+seq_length]
        with open(data["samples"] + ".txt", "w") as f:
            f.write("\n".join(data["samples"]))
        print("t = %d, wrote sequence %d to file" % (t, i))
    else:
        print("done!")

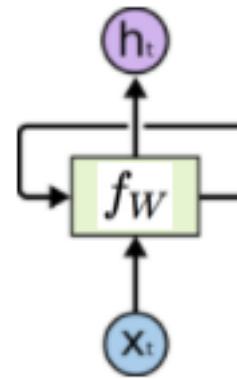
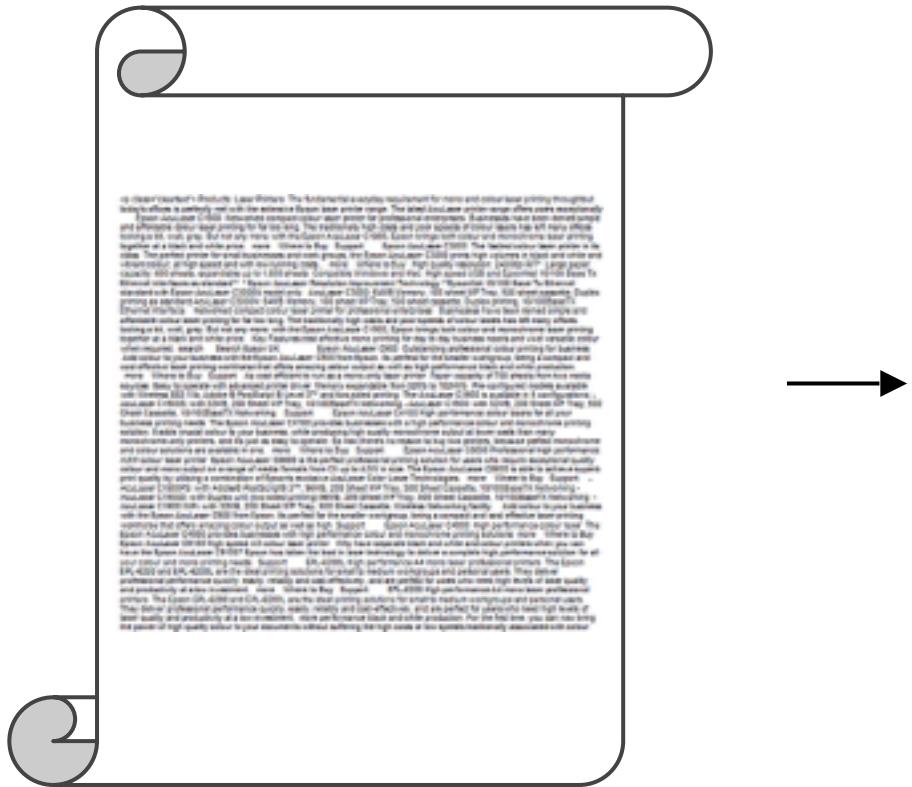
    if t % 50000 == 0:
        for i in range(0, seq_length):
            data["samples"] = sample_ix[i:i+seq_length]
            with open(data["samples"] + ".txt", "w") as f:
                f.write("\n".join(data["samples"]))
            print("t = %d, wrote sequence %d to file" % (t, i))
        else:
            print("done!")
    
```

```

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

```

n Andrej Karpathy
1n



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 tkldrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

↓ train more

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwyl 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 ofter.

↓ 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

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

The Stacks project now consists of

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

Latex source

For $\bigoplus_{m=1,\dots,m} \mathcal{L}_{m,i} = 0$, hence we can find a closed subset H in H and any sets 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 comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \rightarrow V$. Consider the maps M along the set of points Sch_{fppf} and $U \rightarrow U$ 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 $\text{Spec}(R') \rightarrow S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S . We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ 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 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

$$\widetilde{M}^\bullet = T^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} = i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)_{\text{fppf}}^{\text{op}}, (\text{Sch}/S)_{\text{fppf}}$$

and

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

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 for prove any open covering follows from the less of Example ?? . It may replace S by $X_{\text{spaces},\text{étale}}$ which gives an open subspace of X and T equal to S_{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 $X = \lim X_i$ (by the formal open covering X and a single map $\underline{\text{Proj}}_X(\mathcal{A}) = \text{Spec}(B)$ over U compatible with the complex

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

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 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 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 $U \cap U = \coprod_{i=1,\dots,n} U_i$ be the scheme X over S at the schemes $X_i \rightarrow X$ and $U = \lim X_i$. \square

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

Lemma 0.2. Let X be a locally Noetherian scheme over S , $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq p$ is a subset of $\mathcal{J}_{n,0} \circ \overline{A}_2$ 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 δ_{n+1} is a scheme over S . \square

Proof. Omitted. \square

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

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

Proof. This is an algebraic space with the composition of sheaves \mathcal{F} on X_{etale} 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. \square

Lemma 0.2. This is an integer Z is injective.

Proof. See Spaces, Lemma ???. \square

Lemma 0.3. Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset X$ 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. \square

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

$$\begin{array}{ccccc}
 S & \xrightarrow{\quad} & & & \\
 \downarrow & & & & \\
 \mathcal{L} & \xrightarrow{\quad} & \mathcal{O}_{X'} & \xleftarrow{\quad} & \\
 \text{gor}_s & & \uparrow & & \searrow \\
 & & = \alpha' & \longrightarrow & \\
 & & \downarrow & & \\
 & & = \alpha' & \longrightarrow & \\
 & & & & X \\
 & & & & \downarrow \\
 \text{Spec}(K_\psi) & & \text{Mor}_{\text{Sets}} & & \text{d}([\mathcal{O}_{X_{etale}}, \mathcal{G}])
 \end{array}$$

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 . \square

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 C . The functor \mathcal{F} is a "field"

$$\mathcal{O}_{X,s} \rightarrow \mathcal{F}_s \dashv (\mathcal{O}_{X_{etale}}) \rightarrow \mathcal{O}_{X_s}^\pm \mathcal{O}_{X_s}(\mathcal{O}_{X_s}^\pm)$$

is an isomorphism of covering of \mathcal{O}_{X_s} . 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. \square

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

This repository Search Explore Gist Blog Help

karpathy + ·

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 'dmm-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/torvalds/linux-2.6.38-rc1	6 days ago
	arch	Merge branch 'x86-urgent-for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.38-rc1	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/torvalds/linux-2.6.38-rc1	10 days ago
	drivers	Merge branch 'dmm-fixes' of git://people.freedesktop.org/~airlied/linux	9 hours ago
	firmware	firmware/hex2fw.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/linux/kernel/git/torvalds/linux-2.6.38-rc1	a day ago
	init	init: fix regression by supporting devices with major:minor:offset fo...	a month ago
	io	io: remove unnecessary check for -EIOCBQUEUED in io_submit	a month ago

HTTPS clone URL <https://github.com/torvalds/linux>

You can clone with [HTTPS](#), [SSH](#), or [Subversion](#).

Clone in Desktop Download ZIP

Code

Pulse

Graphs

```
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) & 0x0000000f) << 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/seteew.h>
#include <asm/pgproto.h>
```

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

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

#define SNAP_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]);

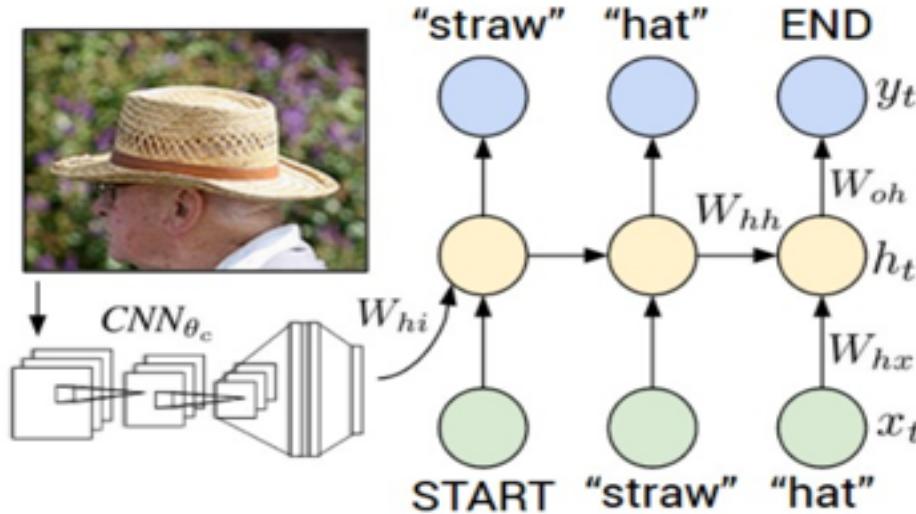
static void
os_prefix(unsigned long sys)
{
#ifdef CONFIG_PREEMPT
    PUT_PARAM_RAID(2, sel) = get_state_state();
    set_pid_sum((unsigned long)state, current_state_str(),
                (unsigned long)-1->lr_full; low;
}
}
```

Recommended Reading: *Visualizing and Understanding Recurrent Networks*

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (!*bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
    */
```

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

Image Captioning



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

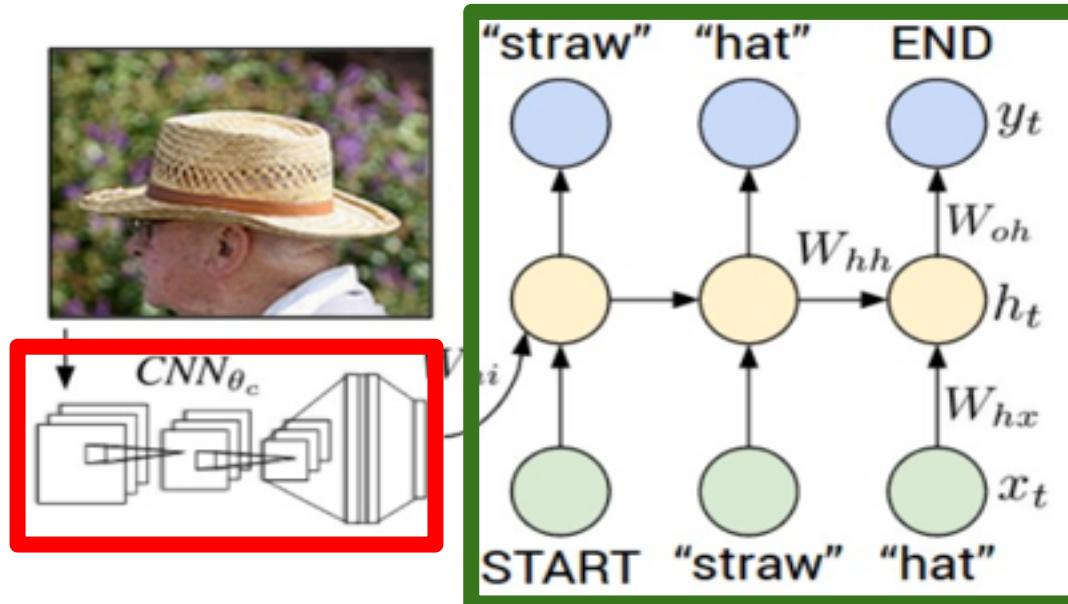
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

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



* Original slides borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000

softmax

* Original image borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

FC-1000



* © borrowed from Andrej Karpathy
and C. Olaf Carlson-Wee, Stanford cs231n

image



test image



conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

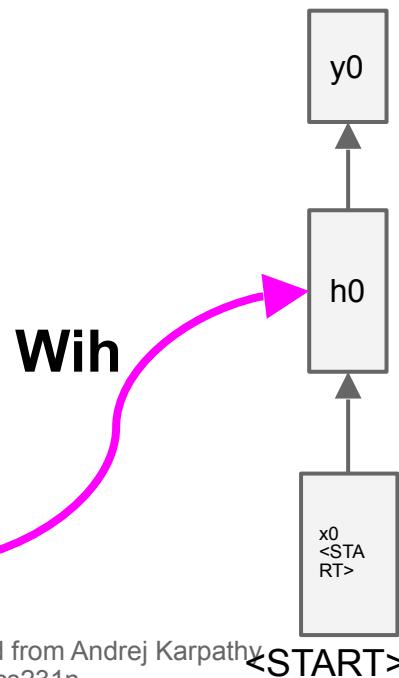
FC-4096

FC-4096





test image



before:

$$h = \tanh(Wxh * x + Whh * h)$$

now:

$$h = \tanh(Wxh * x + Whh * h + WiH * v)$$

image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

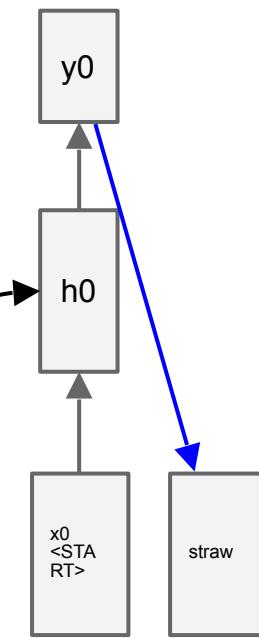
conv-512

conv-512

maxpool

FC-4096

FC-4096



sample!

image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

wed from Andrej Karpathy
rd cs231n

<START>

comp150dl



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

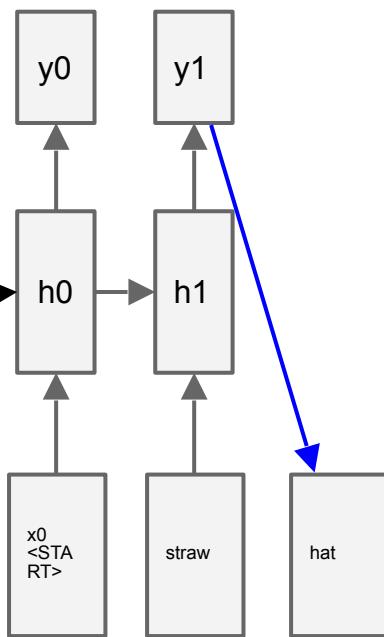
conv-512

conv-512

maxpool

FC-4096

FC-4096



sample!

image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

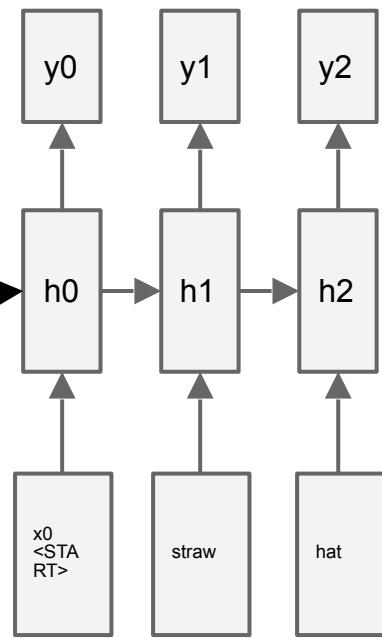
conv-512

conv-512

maxpool

FC-4096

FC-4096



image



test image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

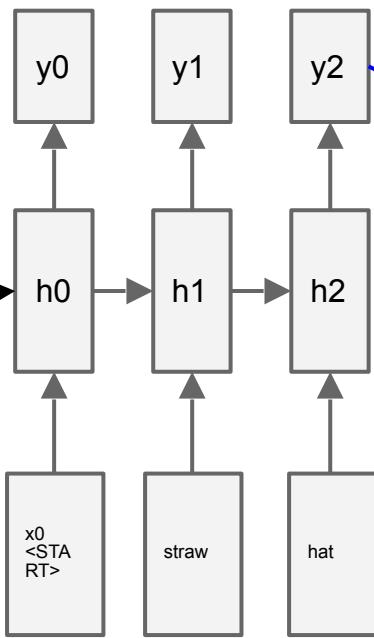
conv-512

conv-512

maxpool

FC-4096

FC-4096



sample
<END> token
=> finish.

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."



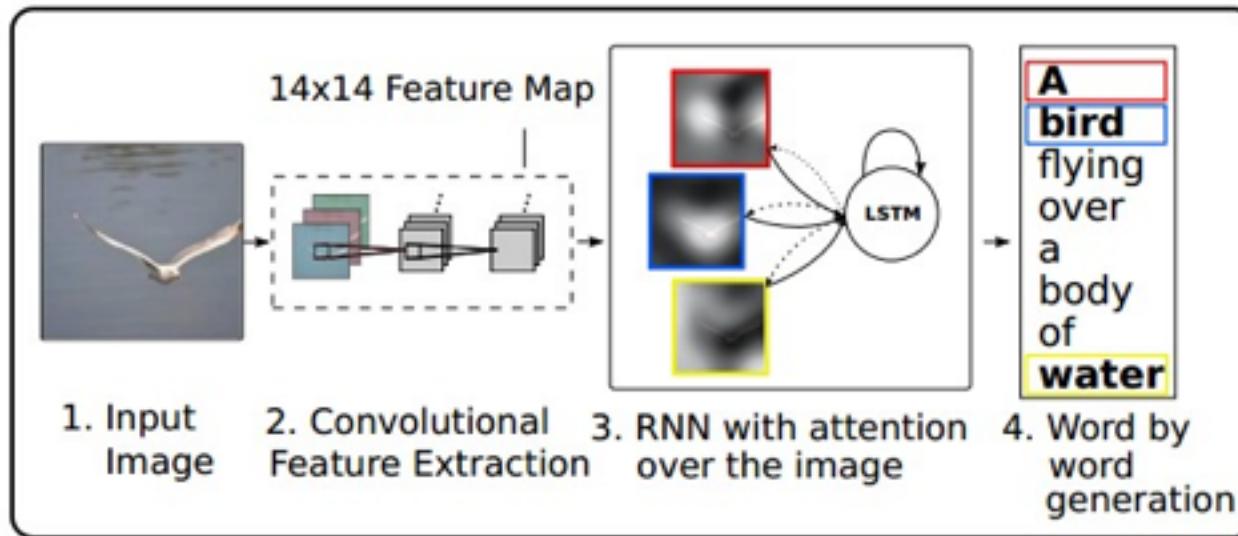
"a woman holding a teddy bear in front of a mirror."



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

Preview of fancier architectures

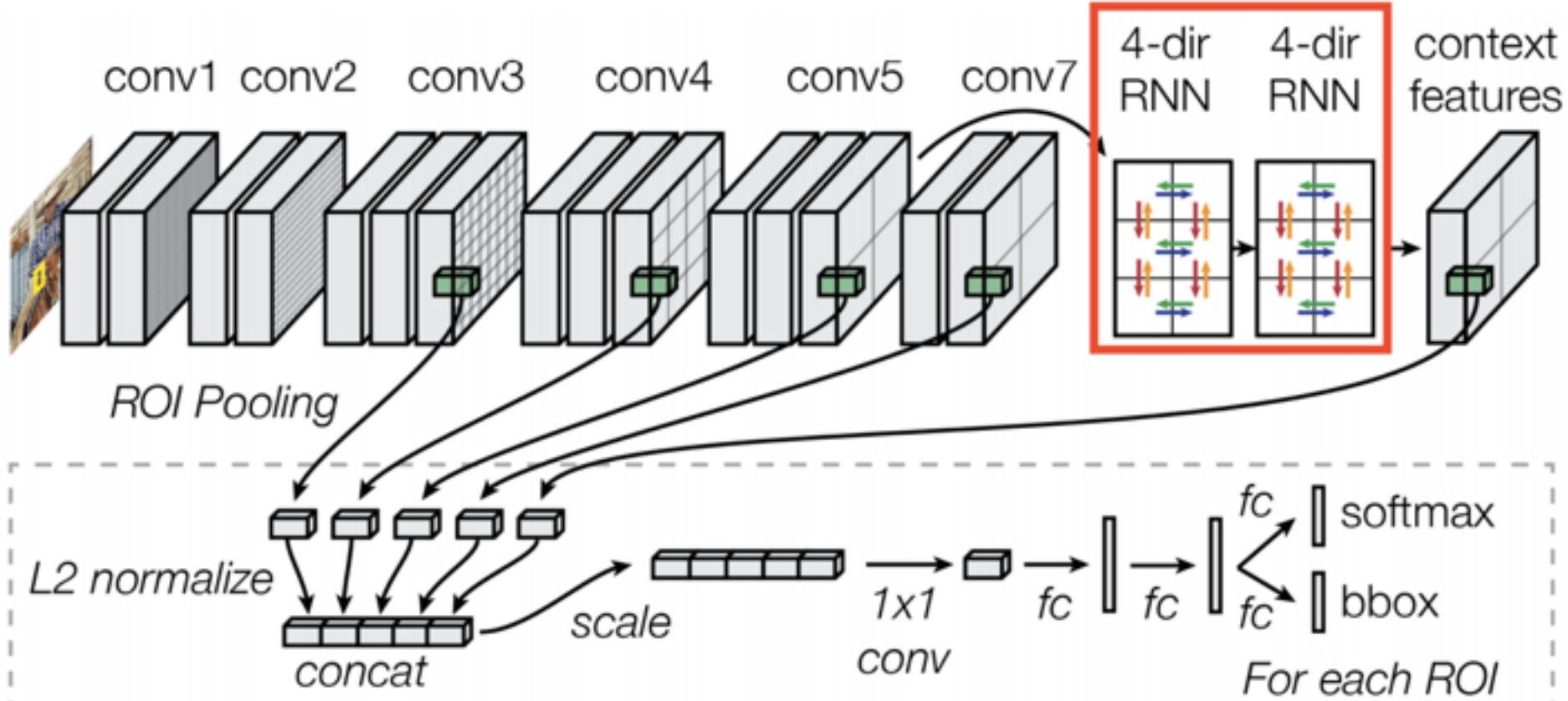
RNN attends spatially to different parts of images while generating each word of the sentence:



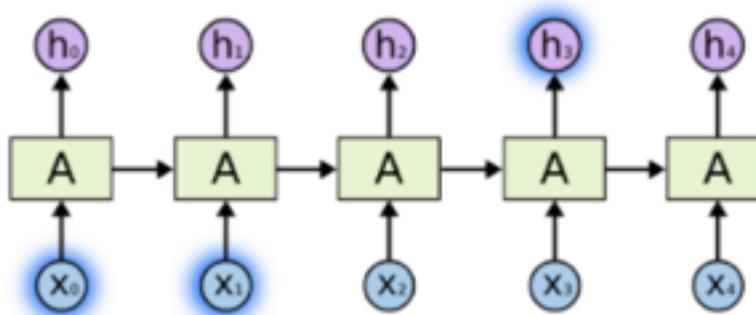
Show Attend and Tell, Xu et al., 2015

* Original slides borrowed from Andrej Karpathy
and Li Fei-Fei, Stanford cs231n

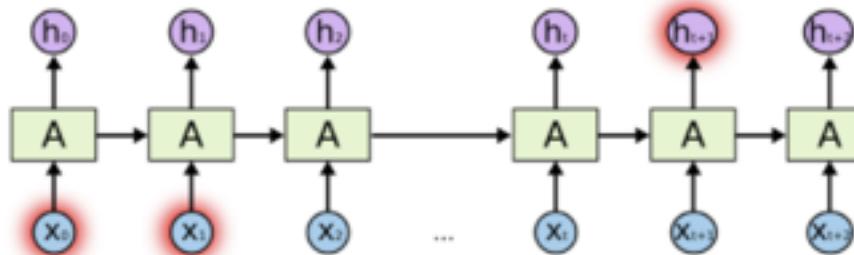
ION: INSIDE-OUTSIDE NET



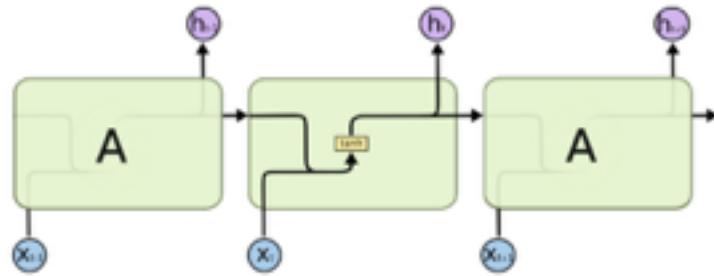
Limitations of RNNs



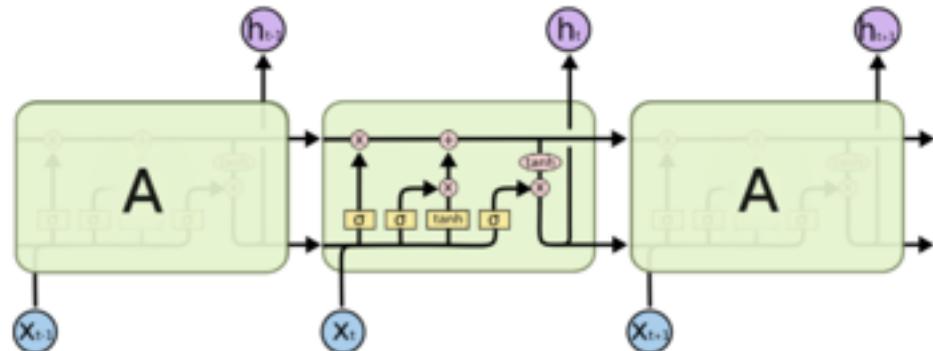
"I grew up in France... I speak fluent **French**."



Long Short Term Memory Networks



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



The repeating module in an LSTM contains four interacting layers.

RNN:

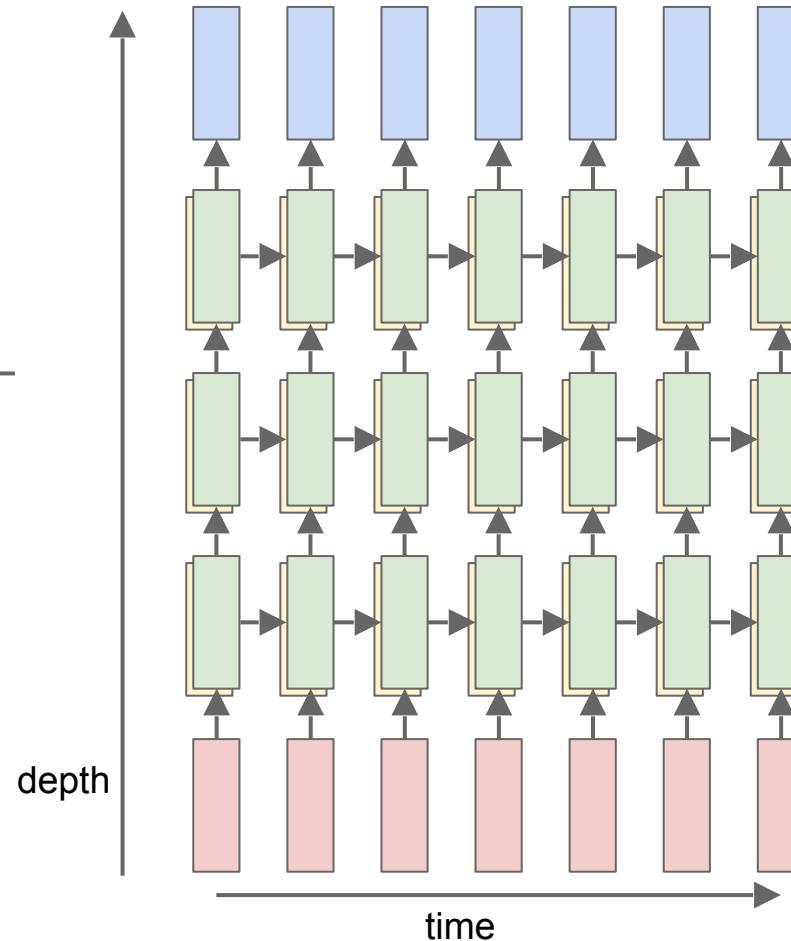
$$h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

LSTM:

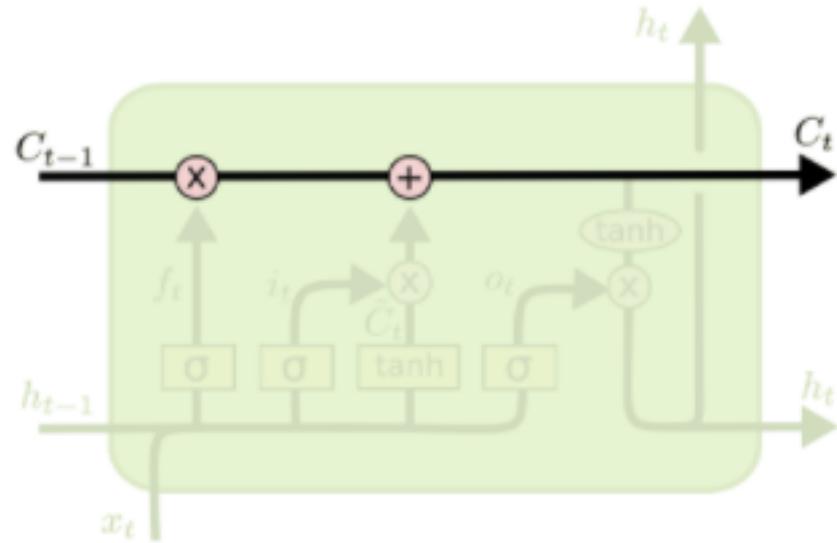
$$W^l [4n \times 2n]$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$



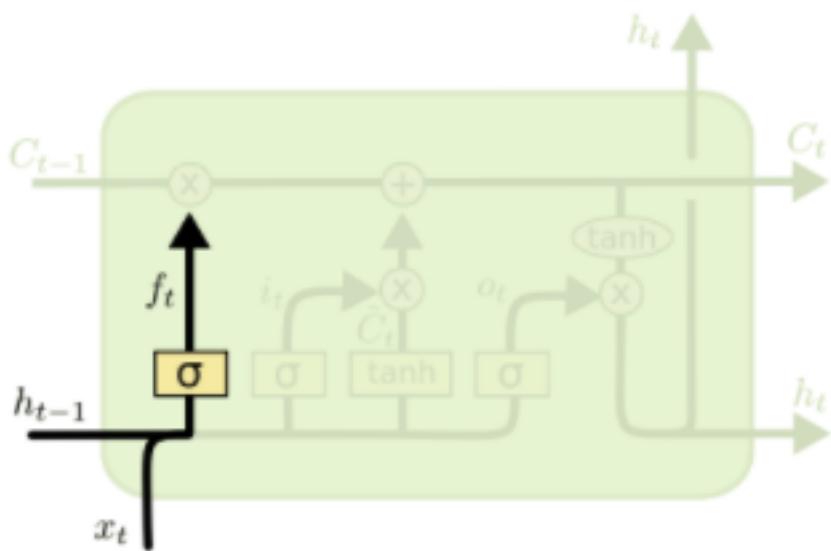
LSTM: Cell State

long running memory of the network



LSTM: Forget Gate f

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



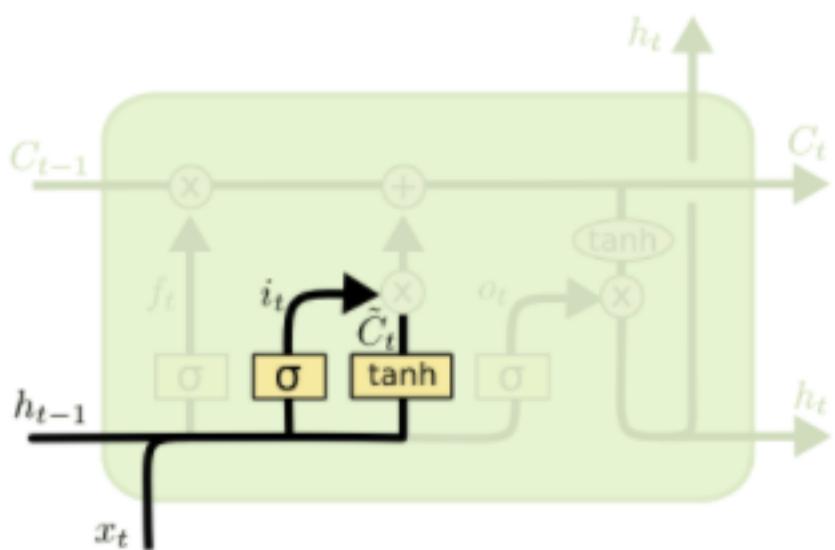
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

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

* figures courtesy Chris Olah

LSTM: Ignore Gate i

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



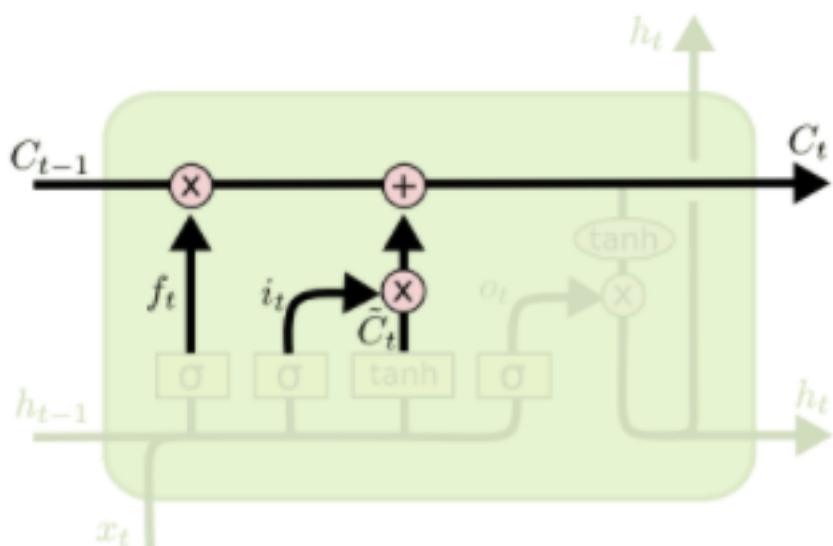
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

* figures courtesy Chris Olah

LSTM: Block Gate g

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



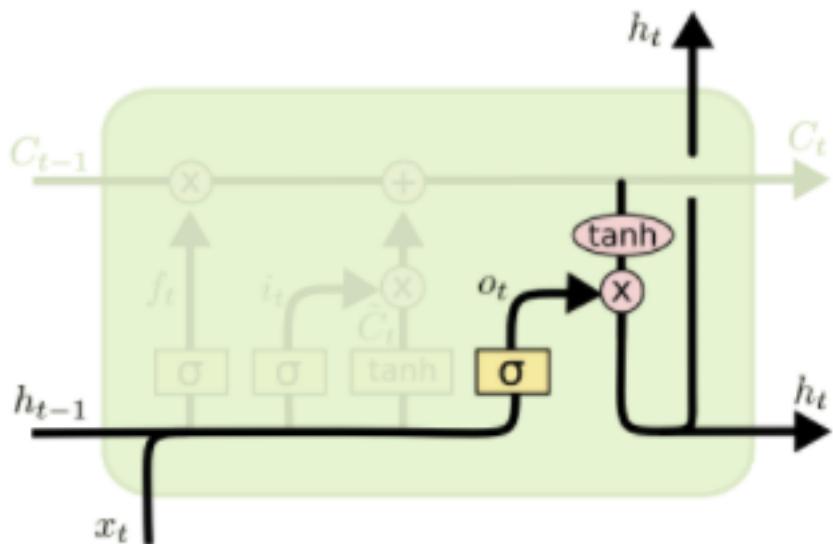
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

* figures courtesy Chris Olah

LSTM: Output Gate o

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



$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

* figures courtesy Chris Olah

Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don't work very well
- Common to use LSTM: 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)
- Additional resource for RNNs and LSTMs for Deep NLP:
cs224d.stanford.edu