

Lecture 6: Training Neural Networks, Part II

Tuesday February 7, 2017

Announcements!

- Don't worry too much if you were late on HW1
- HW2 due February 24
 - fully connected multi-layer nets, batch norm, dropout, etc.
- Email me you areas of interest for final project
 - Some ideas on class webpage
- Guidelines for paper presentations on website

Python/Numpy of the Day

- `numpy.where(<condition>, x, y)`
- Vectorized version of the ternary expression `x if condition else y`, like a vectorized list comprehension

```
In [143]: result = [(x if c else y)
.....:               for x, y, c in zip(xarr, yarr, cond)]

In [144]: result
Out[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.399999999999]
```

- Not very fast for large arrays (because all the work is being done in pure Python)
- Will not work with multidimensional arrays.

```
In [147]: arr = randn(4, 4)

In [148]: arr
Out[148]:
array([[ 0.6372,  2.2043,  1.7904,  0.0752],
       [-1.5926, -1.1536,  0.4413,  0.3403],
       [-0.1798,  0.3299,  0.7827, -0.7585],
       [ 0.5857,  0.1619,  1.3583, -1.3865]])
```

```
In [149]: np.where(arr > 0, 2, -2)
Out[149]:
array([[ 2,  2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2,  2, -2],
       [ 2,  2,  2, -2]])
```

This is better

```
In [150]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[150]:
array([[ 2.    ,  2.    ,  2.    ,  2.    ],
       [-1.5926, -1.1536,  2.    ,  2.    ],
       [-0.1798,  2.    ,  2.    , -0.7585],
       [ 2.    ,  2.    ,  2.    , -1.3865]])
```

Pixel Recursive Super Resolution

Ryan Dahl * Mohammad Norouzi Jonathon Shlens

Google Brain

{rld,norouzi,shlens}@google.com

Abstract

We present a pixel recursive super resolution model that synthesizes realistic details into images while enhancing their resolution. A low resolution image may correspond to multiple plausible high resolution images, thus modeling the super resolution process with a pixel independent conditional model often results in averaging different details—hence blurry edges. By contrast, our model is able to represent a multimodal conditional distribution by properly modeling the statistical dependencies among the high resolution image pixels, conditioned on a low resolution input. We employ a PixelCNN architecture to define a strong prior over natural images and jointly optimize this prior with a deep conditioning convolutional network. Human evaluations indicate that samples from our proposed model look more photo realistic than a strong L2 regression baseline.

1. Introduction

The problem of super resolution entails artificially enlarging a low resolution photograph to recover a plausible high resolution version of it. When the zoom factor is large, the input image does not contain all of the information necessary to accurately construct a high resolution image. Thus, the problem is underspecified and many plausible high resolution images exist that match the low resolution input image. This problem is significant for improving the state-of-the-art in super resolution, and more generally



Figure 1: Illustration of our probabilistic pixel recursive super resolution model trained end-to-end on a dataset of celebrity faces. The left column shows 8×8 low resolution inputs from the test set. The middle and last columns show 32×32 images as predicted by our model vs. the ground truth. Our model incorporates strong face priors to synthesize realistic hair and skin details.

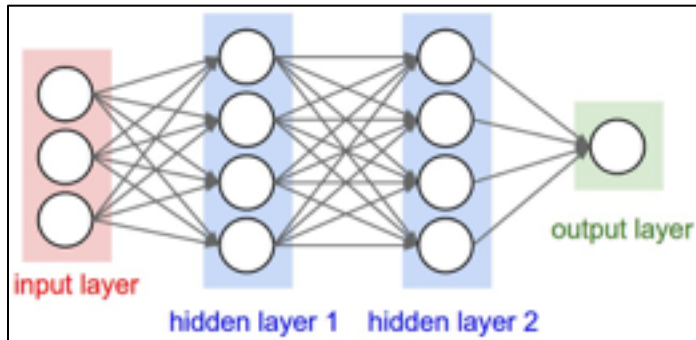
New work out on Feb 2

<https://arxiv.org/pdf/1702.00783.pdf>

Mini-batch SGD

Loop:

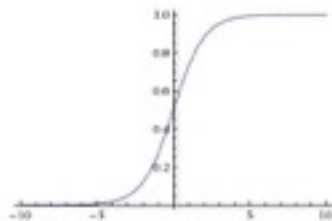
1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient



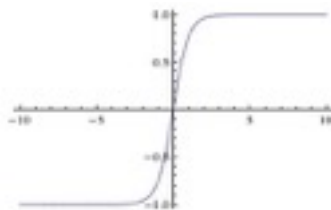
Activation Functions

Sigmoid

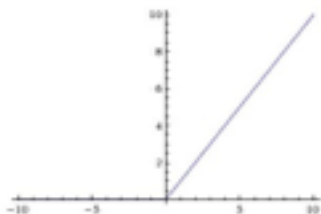
$$\sigma(x) = 1/(1 + e^{-x})$$



tanh $\tanh(x)$

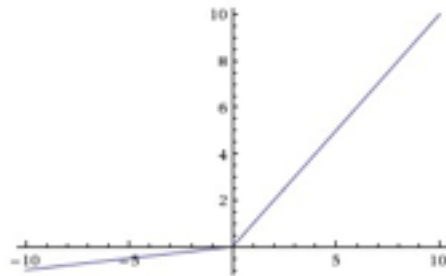


ReLU $\max(0, x)$



Leaky ReLU

$$\max(0.1x, x)$$

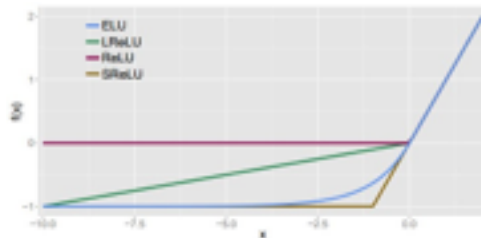


Maxout

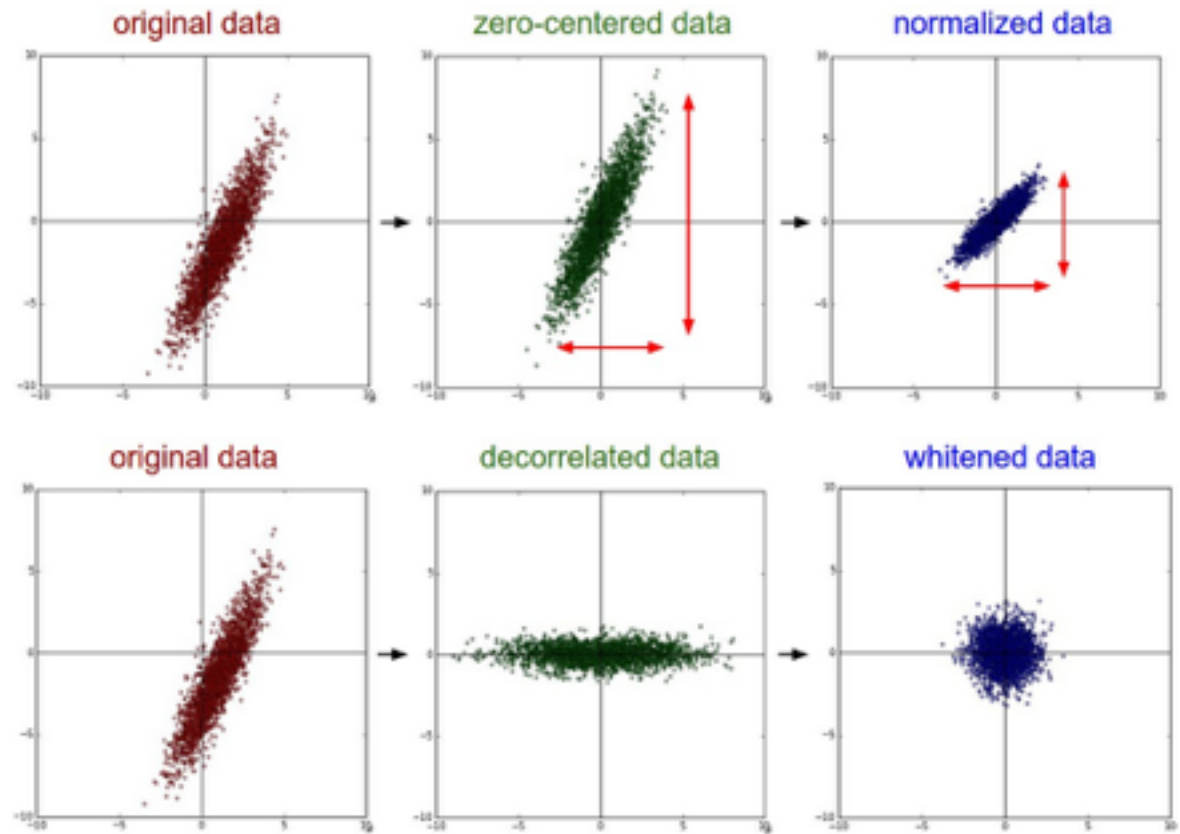
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



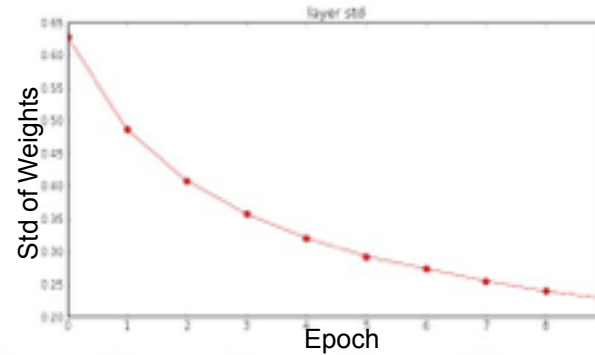
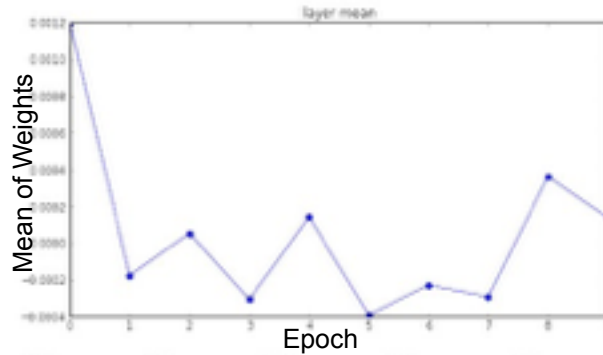
Data Preprocessing



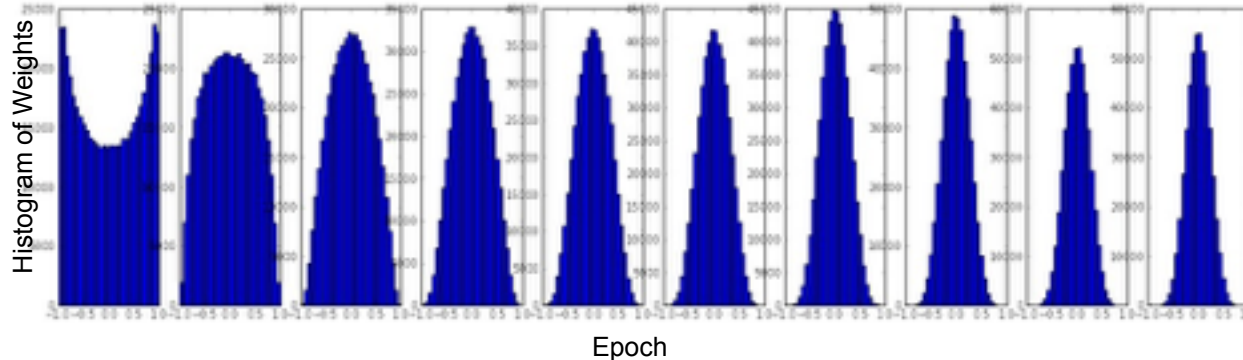
Input layer had mean 0.001800 and std 1.001311
 hidden layer 1 had mean 0.001198 and std 0.627953
 hidden layer 2 had mean -0.000175 and std 0.486051
 hidden layer 3 had mean 0.000055 and std 0.407723
 hidden layer 4 had mean -0.000306 and std 0.357108
 hidden layer 5 had mean 0.000142 and std 0.320917
 hidden layer 6 had mean -0.000389 and std 0.292116
 hidden layer 7 had mean -0.000228 and std 0.273387
 hidden layer 8 had mean -0.000291 and std 0.254935
 hidden layer 9 had mean 0.000361 and std 0.239266
 hidden layer 10 had mean 0.000139 and std 0.228008

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

“Xavier initialization”
 [Glorot et al., 2010]



Reasonable initialization.
 (Mathematical derivation
 assumes linear activations)



Weight
 Initialization

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Babysitting the learning process

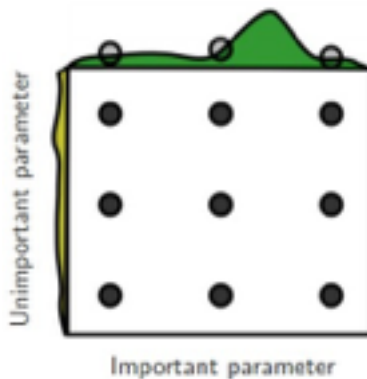
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches=10,
                                  learning_rate=1e-8, verbose=True)
```

Finished epoch 1 / 10: cost 2.302576, train: 0.000000, val 0.101000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.150000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302486, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302439, train: 0.200000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000

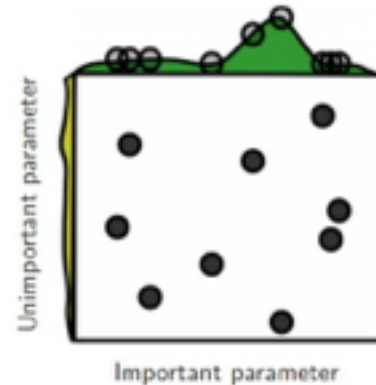
Loss barely changing:
Learning rate is probably
too low

Cross-validation

Grid Layout



Random Layout



Today:

- Parameter update schemes
- Learning rate schedules
- Dropout
- Gradient checking
- Model ensembles

Parameter Updates

Training a neural network, main loop:

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
```

Training a neural network, main loop:

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
```

simple gradient descent update
now: complicate.

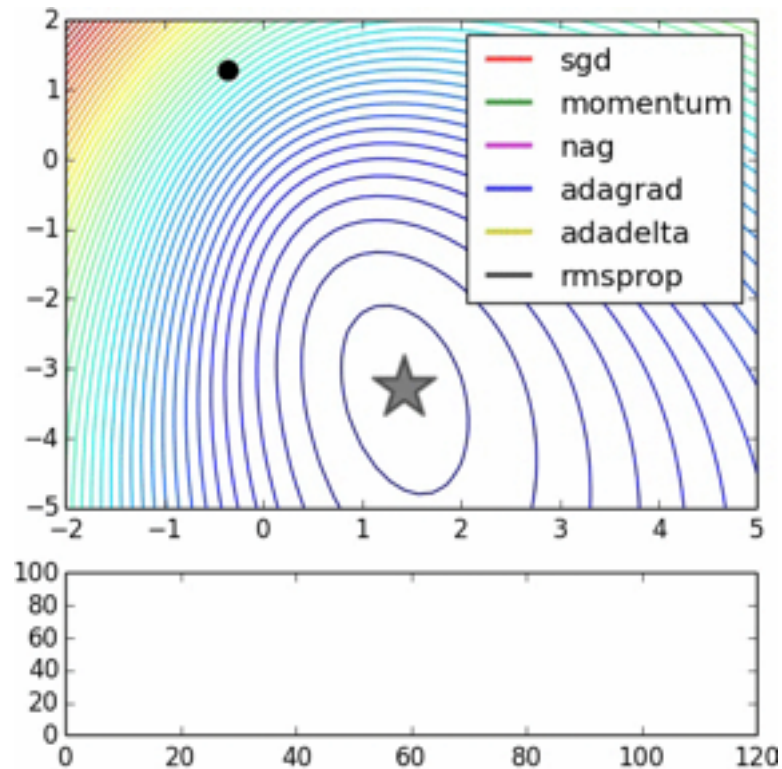
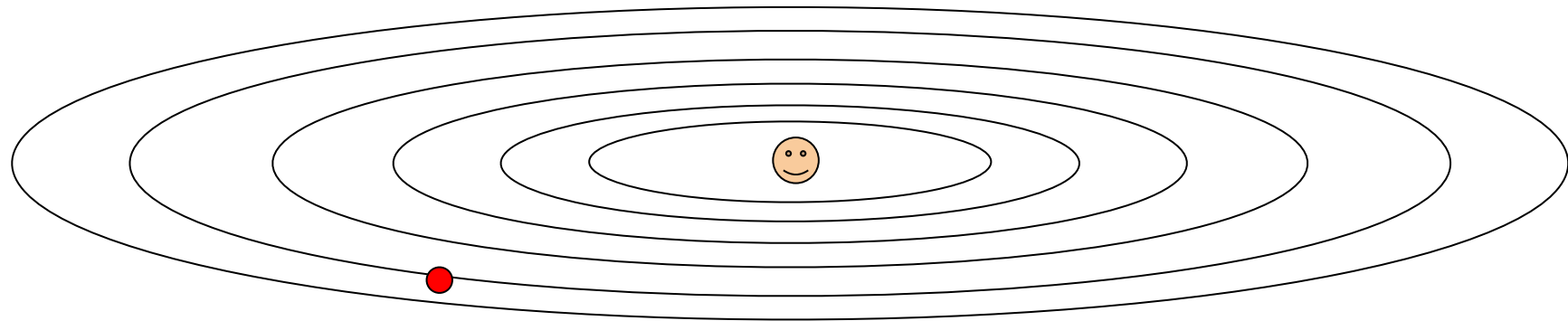


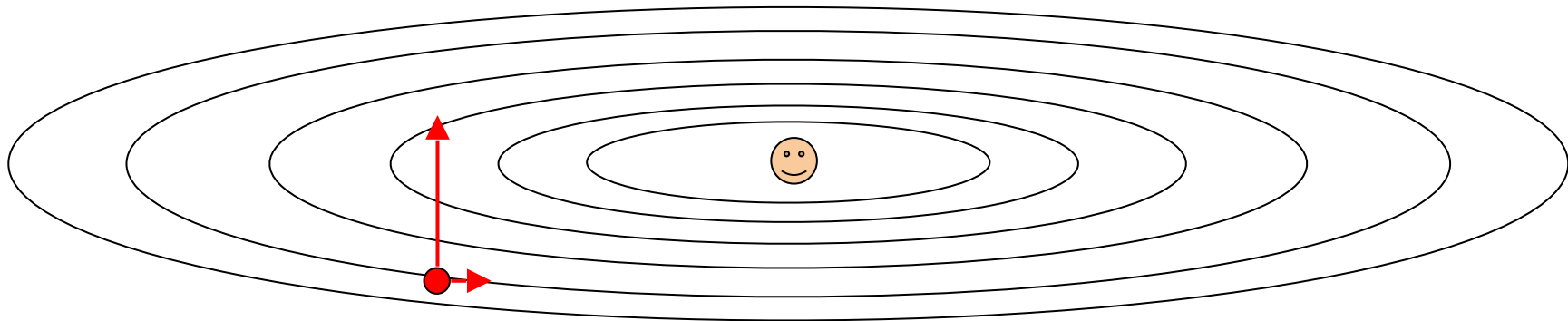
Image credits:
Alec Radford

Suppose loss function is steep vertically but shallow horizontally:



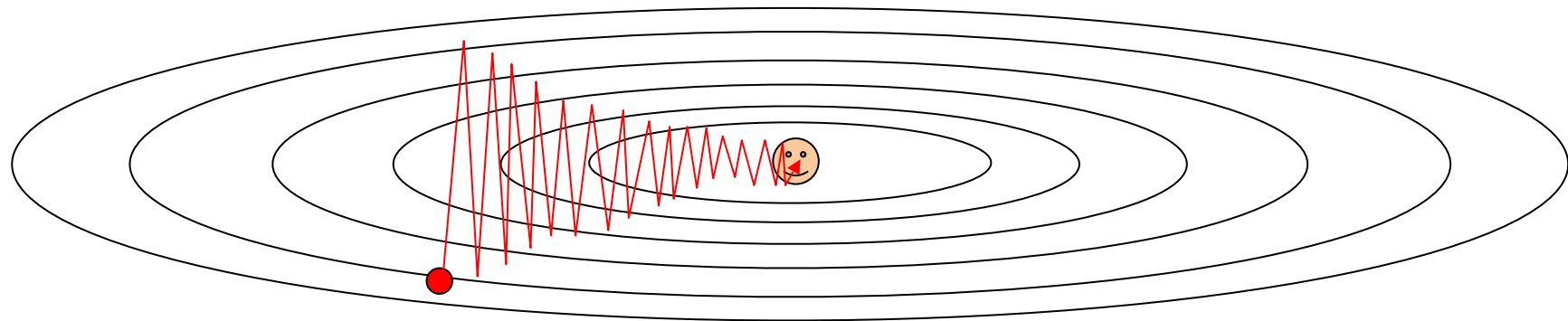
Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?


Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD? **very slow progress along flat direction, jitter along steep one**

Momentum update

```
# Gradient descent update  
x += - learning_rate * dx
```



```
# Momentum update  
v = mu * v - learning_rate * dx # integrate velocity  
x += v # integrate position
```

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).

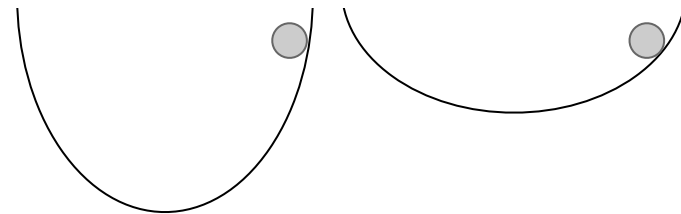
- mu = usually ~0.5, 0.9, or 0.99
- (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

Momentum update

```
# Gradient descent update  
x += - learning_rate * dx
```

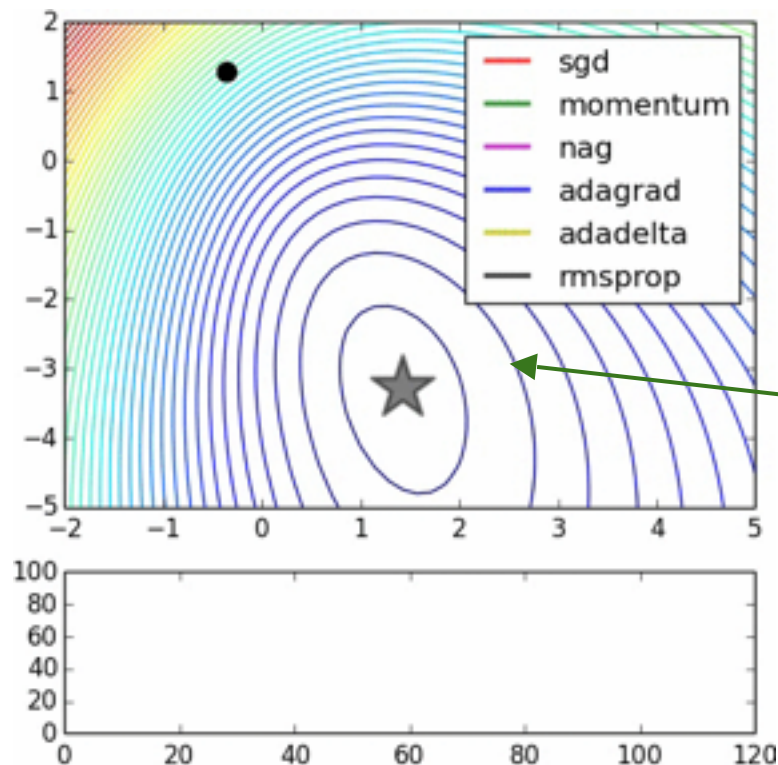


```
# Momentum update  
v = mu * v - learning_rate * dx # integrate velocity  
x += v # integrate position
```



- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD VS Momentum

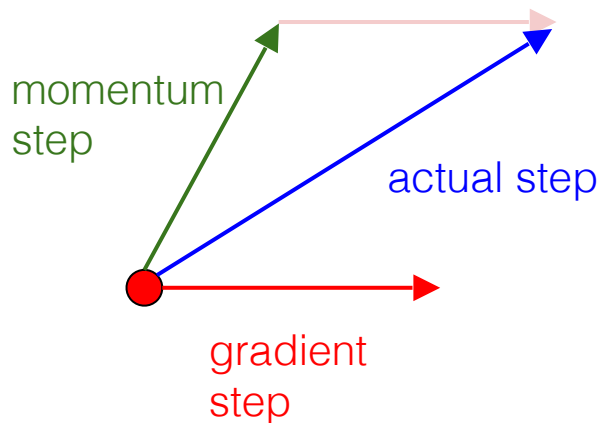


notice momentum overshooting the target, but overall getting to the minimum much faster.

Nesterov Momentum update

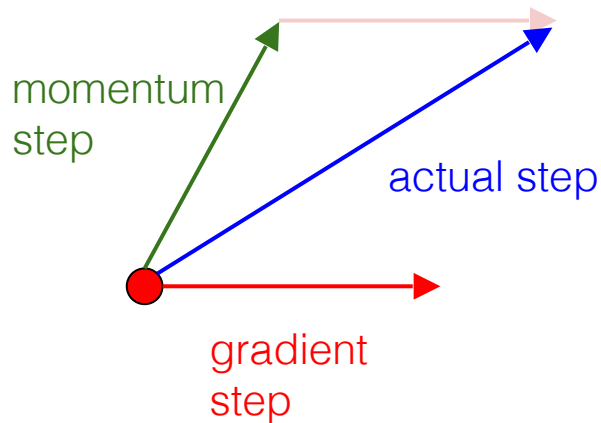
```
# Momentum update  
v = mu * v - learning_rate * dx # integrate velocity  
x += v # integrate position
```

Ordinary momentum update:

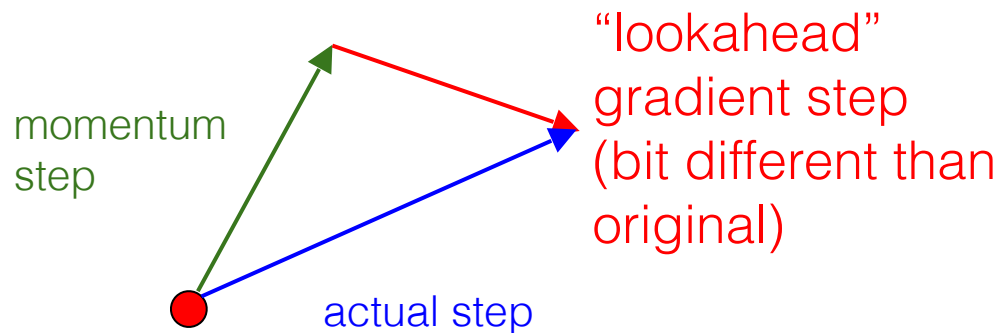


Nesterov Momentum update

Momentum update:

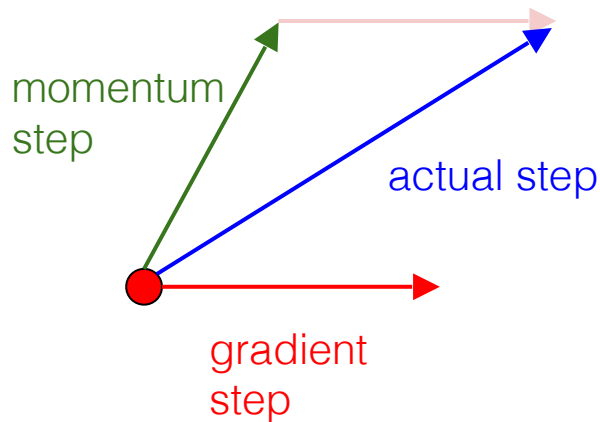


Nesterov momentum update

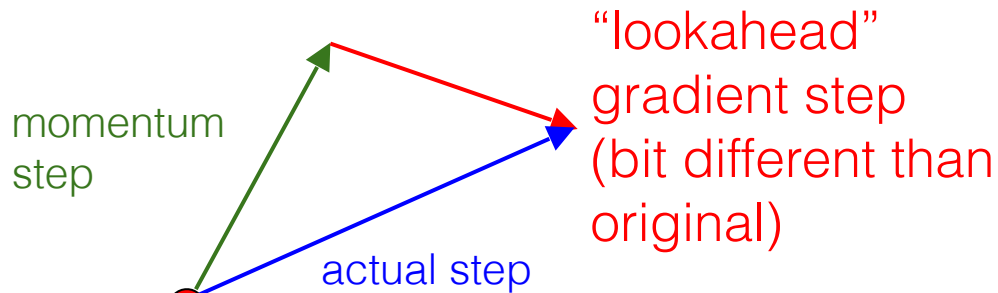


Nesterov Momentum update

Momentum update:



Nesterov momentum update



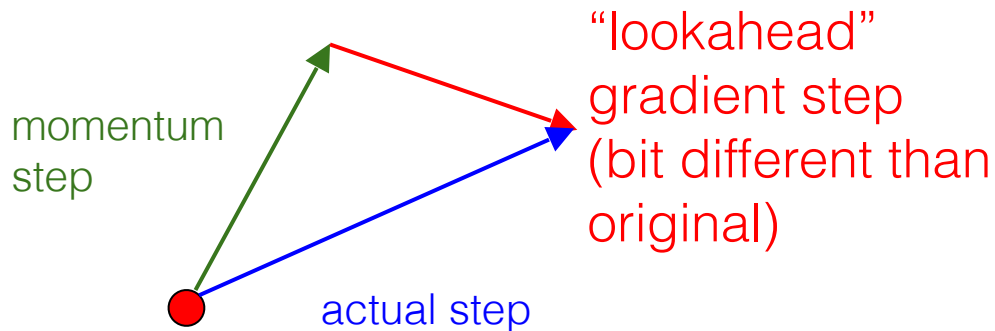
Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

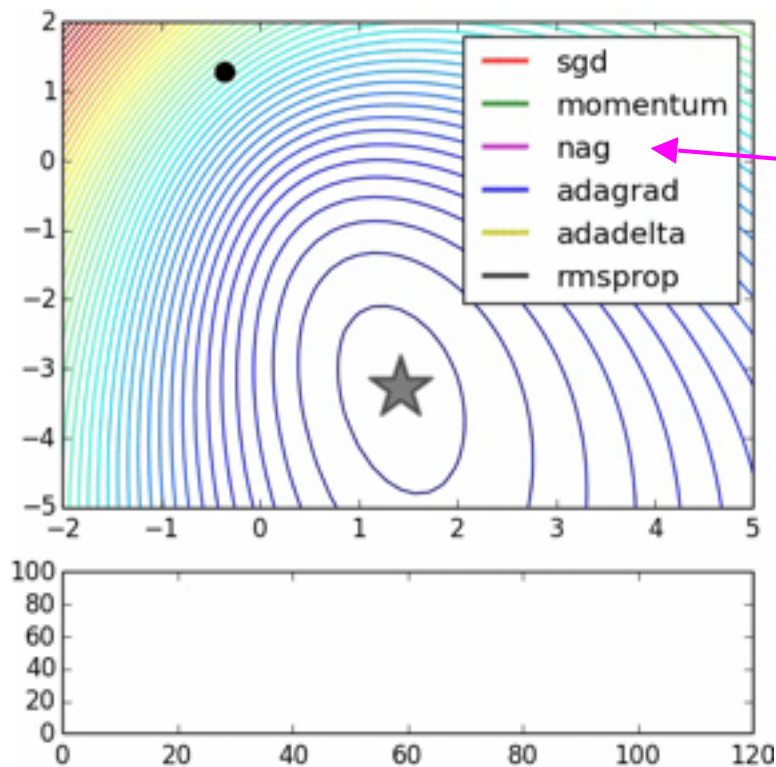
$$\theta_t = \theta_{t-1} + v_t$$

Nesterov Momentum update

```
# Nesterov momentum update rewrite  
v_prev = v  
v = mu * v - learning_rate * dx  
x += -mu * v_prev + (1 + mu) * v
```



Q: What kinds of loss functions could cause problems for the momentum methods?



nag =
Nesterov
Accelerated
Gradient

AdaGrad update

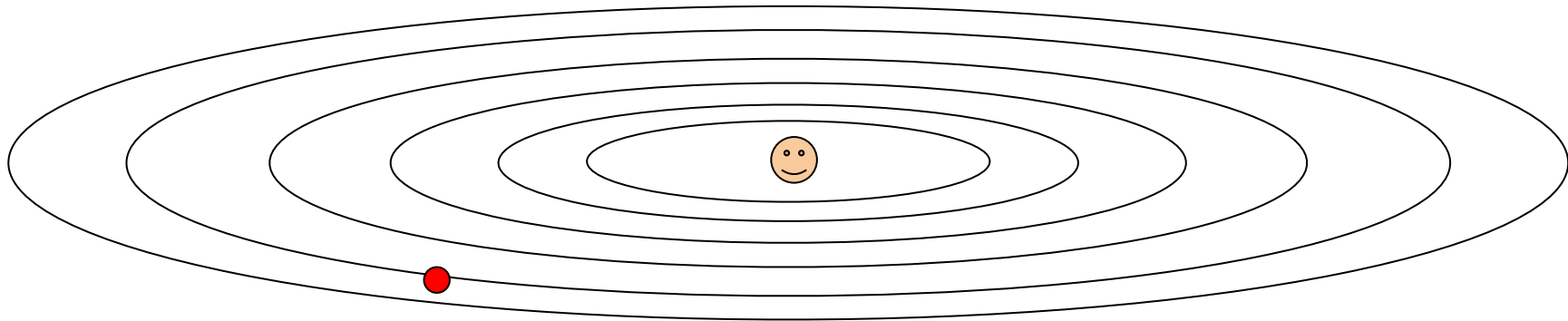
[Duchi et al., 2011]

```
# Adagrad update  
cache += dx**2  
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

AdaGrad update

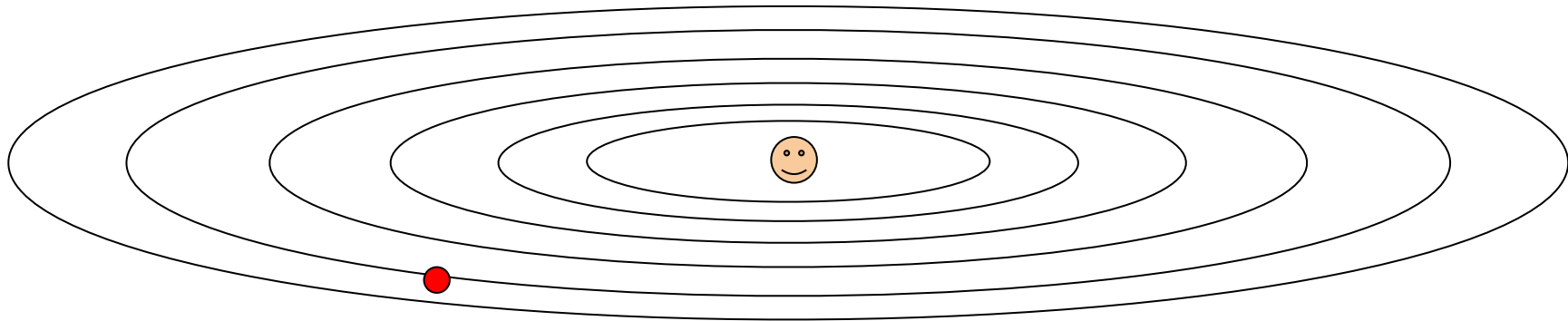
```
# Adagrad update  
cache += dx**2  
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad update

```
# Adagrad update  
cache += dx**2  
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```




Q2: What happens to the step size over long time?

RMSProp update

[Tieleman and Hinton, 2012]

```
# Adagrad update  
cache += dx**2  
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



```
# RMSProp  
cache = decay_rate * cache + (1 - decay_rate) * dx**2  
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
 - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight
$$MeanSquare(w, t) = 0.9 MeanSquare(w, t-1) + 0.1 \left(\frac{\partial E}{\partial w}(t) \right)^2$$
- Dividing the gradient by $\sqrt{MeanSquare(w, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

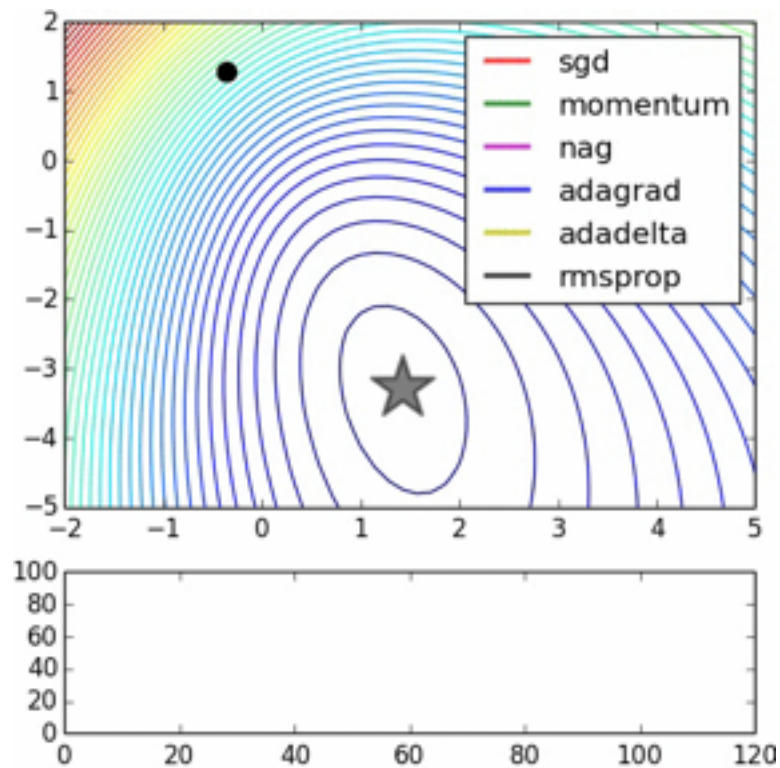
rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
 - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight
$$MeanSquare(w, t) = 0.9 MeanSquare(w, t-1) + 0.1 \left(\frac{\partial E}{\partial w}(t) \right)^2$$
- Dividing the gradient by $\sqrt{MeanSquare(w, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.



adagrad
rmsprop

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

Adam update

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]

```
# Adam
m,v = #... initialize caches to zeros
for t in xrange(1, big_number):
    dx = # ... evaluate gradient
    m = beta1*m + (1-beta1)*dx # update first moment
    v = beta2*v + (1-beta2)*(dx**2) # update second moment
    mb = m/(1-beta1**t) # correct bias
    vb = v/(1-beta2**t) # correct bias
    x += - learning_rate * mb / (np.sqrt(vb) + 1e-7)
```

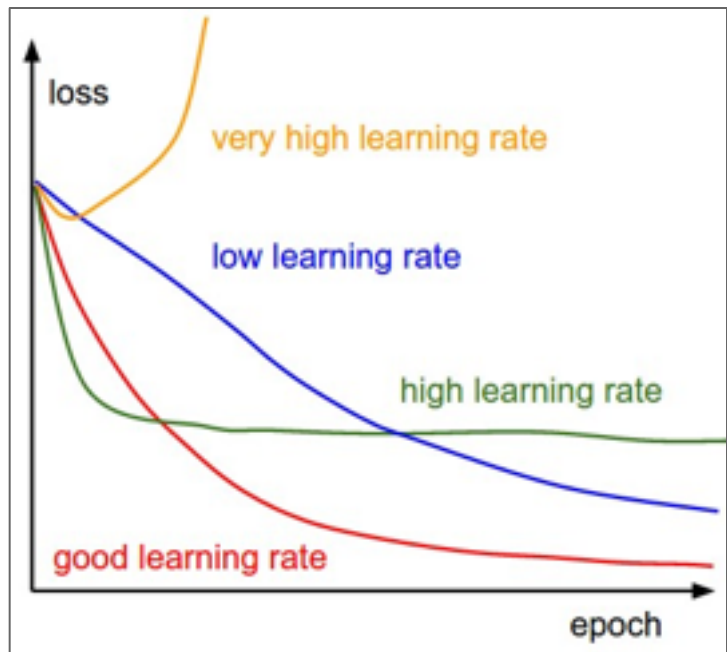
momentum

bias correction
(only relevant in first few
iterations when t is small)

RMSProp-like

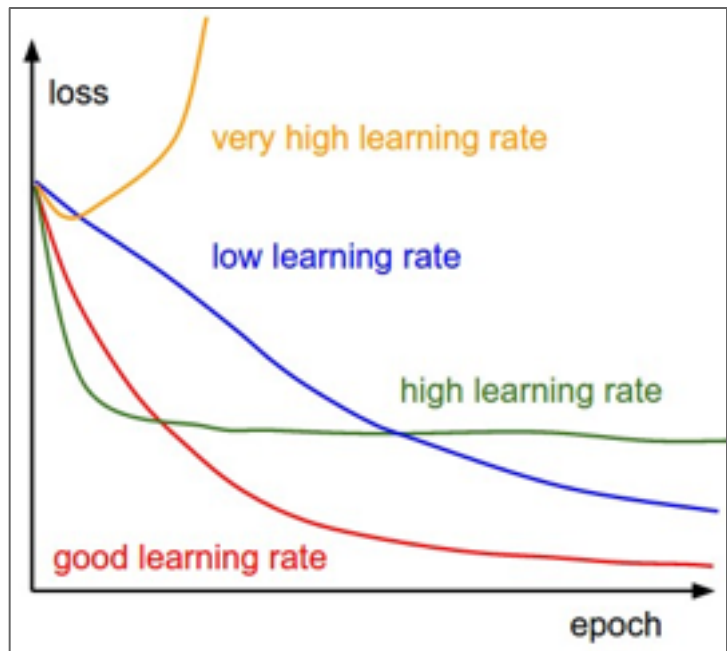
The bias correction compensates for the fact that m, v are initialized at zero and need some time to “warm up”.

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> **Learning rate decay over time!**

step decay:

e.g. decay learning rate by half every few epochs.

exponential decay: $\alpha = \alpha_0 e^{-kt}$

1/t decay: $\alpha = \alpha_0 / (1 + kt)$

Second order optimization methods

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

notice:

no hyperparameters! (e.g. learning rate)

- Quasi-Newton methods (**BGFS** most popular):
- **L-BFGS** (Limited memory BFGS):
Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode**
i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely
- **Does not transfer very well to mini-batch setting.** Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

Evaluation:

Model Ensembles

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance
All competition winners do this.

Fun Tips/Tricks:

- can also get a small boost from averaging multiple model checkpoints of a single model.

Fun Tips/Tricks:

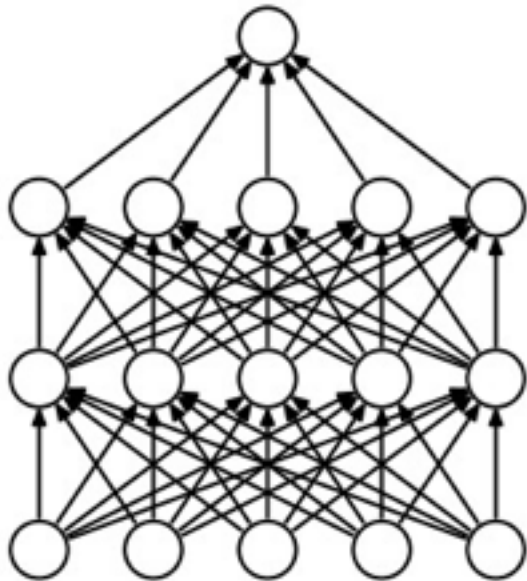
- can also get a small boost from averaging multiple model checkpoints of a single model. (different local minima)
- keep track of (and use at test time) a running average parameter vector:

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
    x_test = 0.995*x_test + 0.005*x # use for test set
```

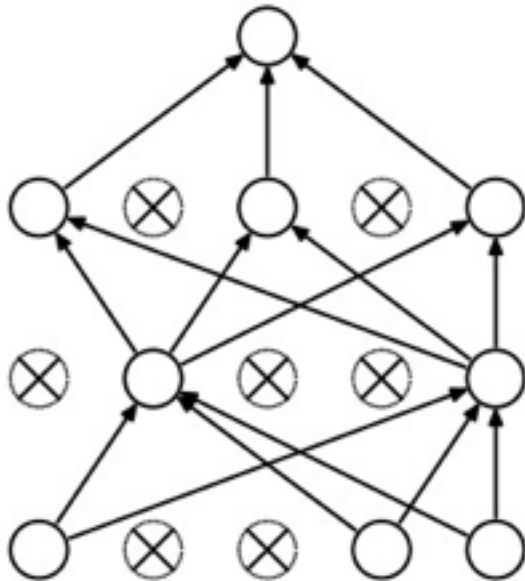
Regularization (**dropout**)

Regularization: **Dropout**

“randomly set some neurons to zero in the forward pass”



(a) Standard Neural Net



(b) After applying dropout.

[Srivastava et al., 2014]

```
p = 0.5 # probability of keeping a unit active. higher = less dropout
```

```
def train_step(X):
```

```
    """ X contains the data """
```

```
    # forward pass for example 3-layer neural network
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
    H1 *= U1 # drop!
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

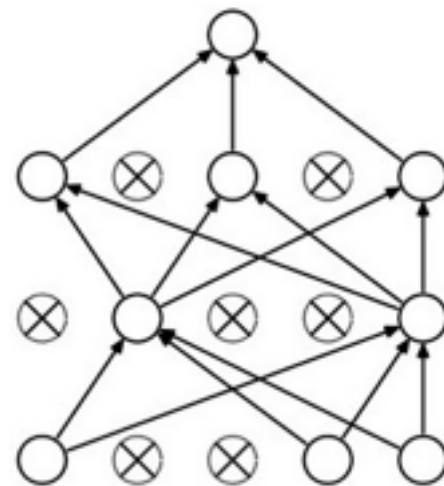
```
    H2 *= U2 # drop!
```

```
    out = np.dot(W3, H2) + b3
```

```
    # backward pass: compute gradients... (not shown)
```

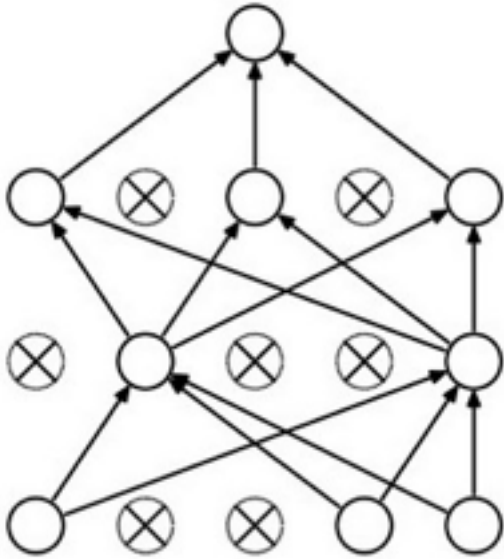
```
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Waaaaait a second...

How could this possibly be a good idea?



How could this possibly be a good idea?

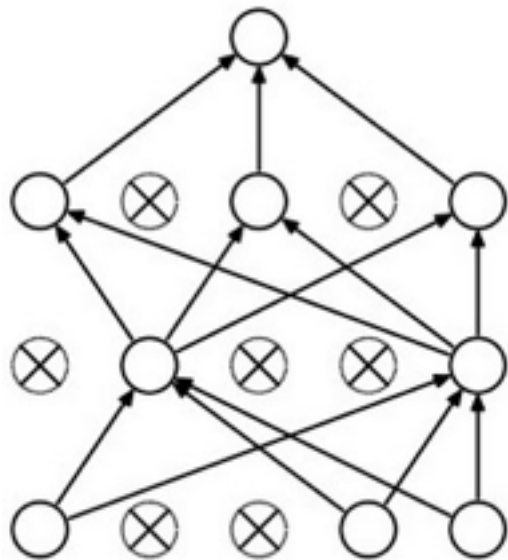
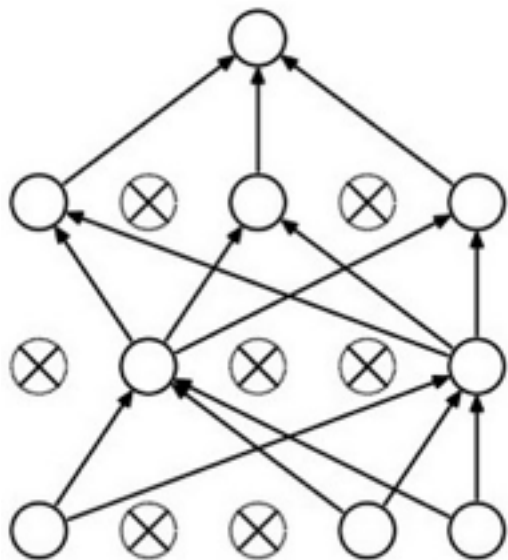


Diagram illustrating feature selection for a cat score. Five features are listed, each with a red 'X' indicating it is selected:

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

Arrows from the selected features point to the 'cat score'.

At test time....



Ideally:

want to integrate out all the noise

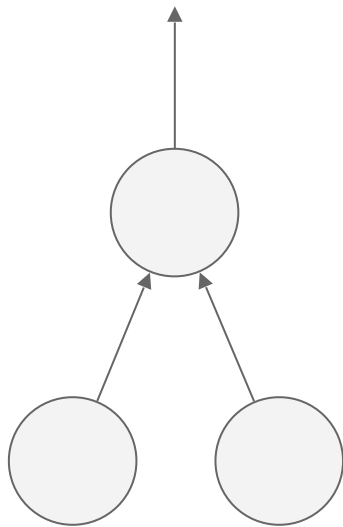
Monte Carlo approximation:

do many forward passes with different dropout masks, average all predictions

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



Q: Suppose that with all inputs present at test time the output of this neuron is x .

What would its output be during training time, in expectation? (e.g. if $p = 0.5$)

We can do something approximate analytically

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
```

```
p = 0.5 # probability of keeping a unit active. higher = less dropout
```

```
def train_step(X):
```

```
    """ X contains the data """
```

```
    # forward pass for example 3-layer neural network
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
    H1 *= U1 # drop!
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

```
    H2 *= U2 # drop!
```

```
    out = np.dot(W3, H2) + b3
```

```
    # backward pass: compute gradients... (not shown)
```

```
    # perform parameter update... (not shown)
```

```
def predict(X):
```

```
    # ensembled forward pass
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
```

```
    out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

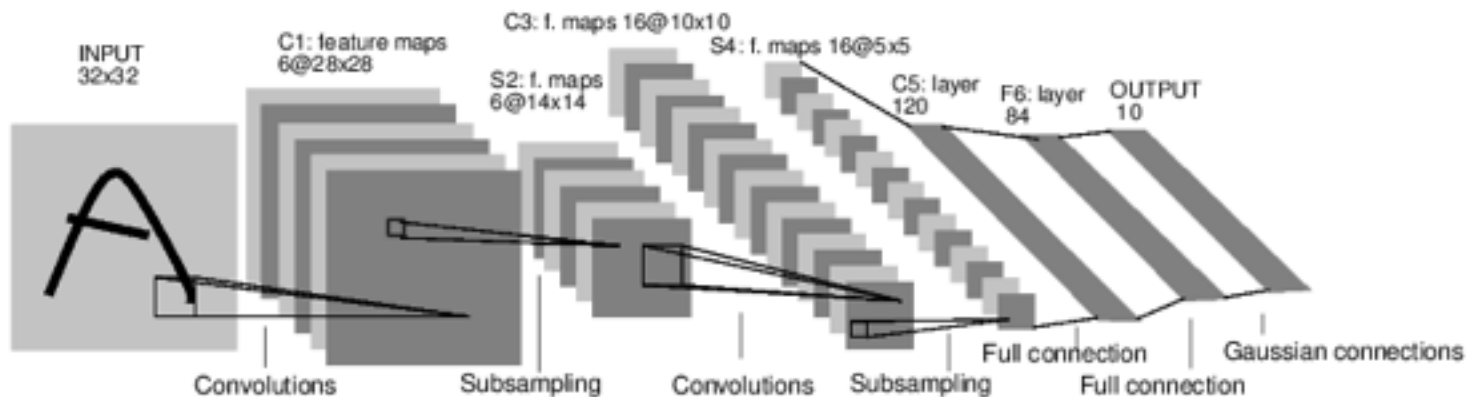
def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!

Convolutional Neural Networks



*[LeNet-5,
LeCun 1980]*

A bit of history:

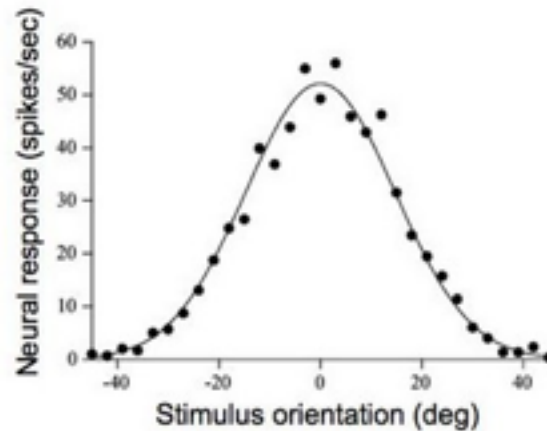
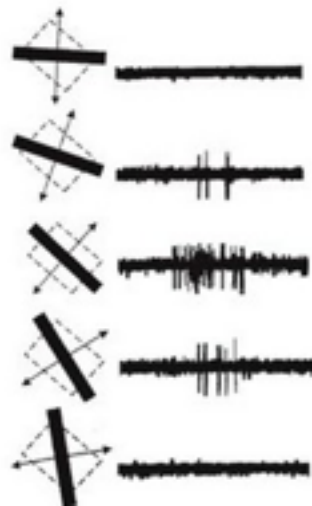
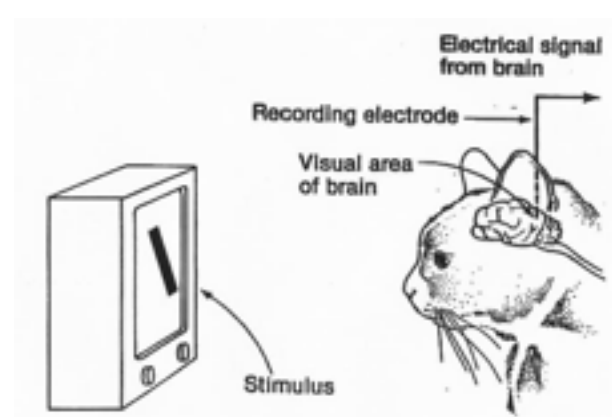
Hubel & Wiesel,

1959

Receptive Fields of Single
Neurons in Cat's Striate
Cortex

1962

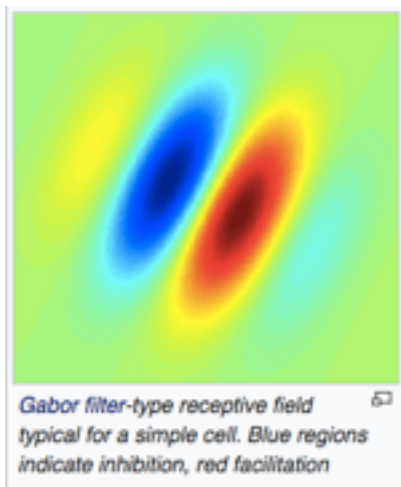
Receptive Fields, Binocular
Interaction and Functional
Architecture in Cat's Visual
Cortex



A bit of history

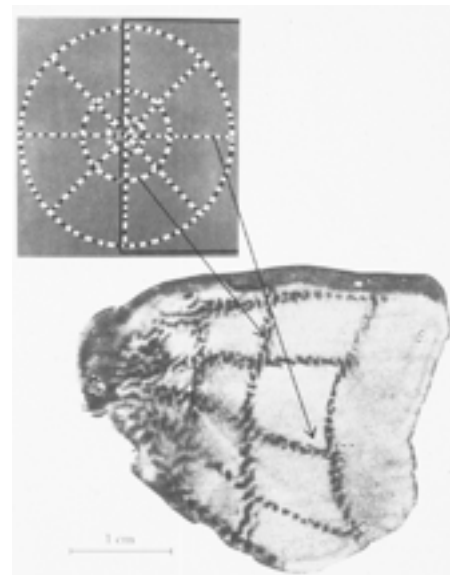
Simple Cell:

cell in the primary visual cortex that responds primarily to oriented edges and gratings

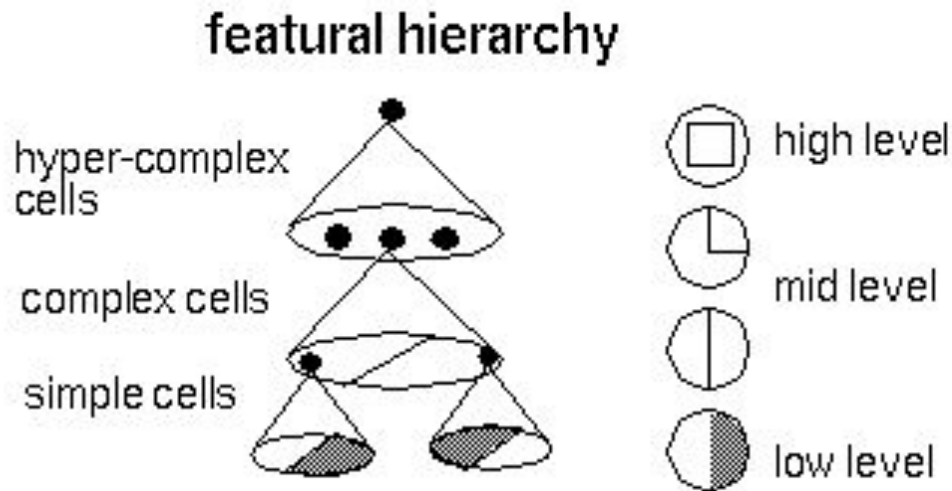
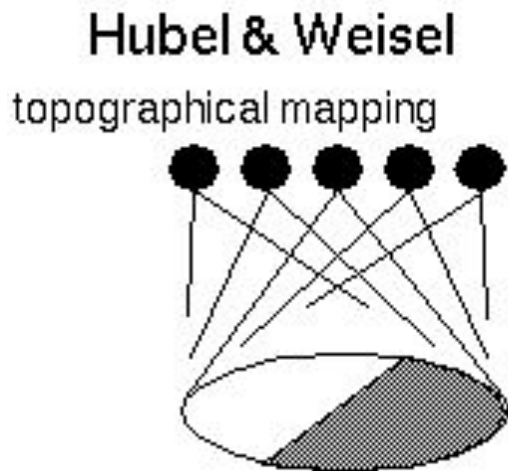


Topographical mapping in the cortex:

nearby cells in cortex represented nearby regions in the visual field

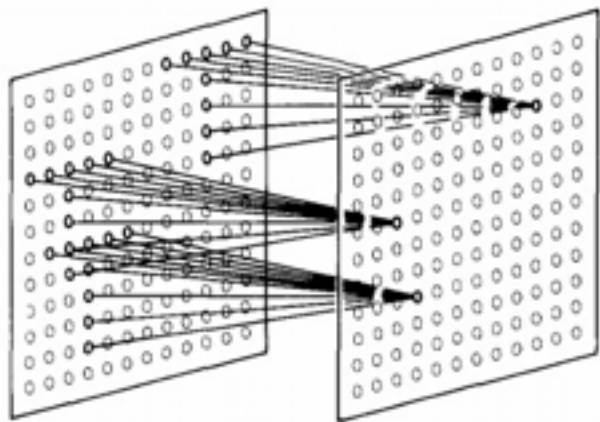


Hierarchical organization

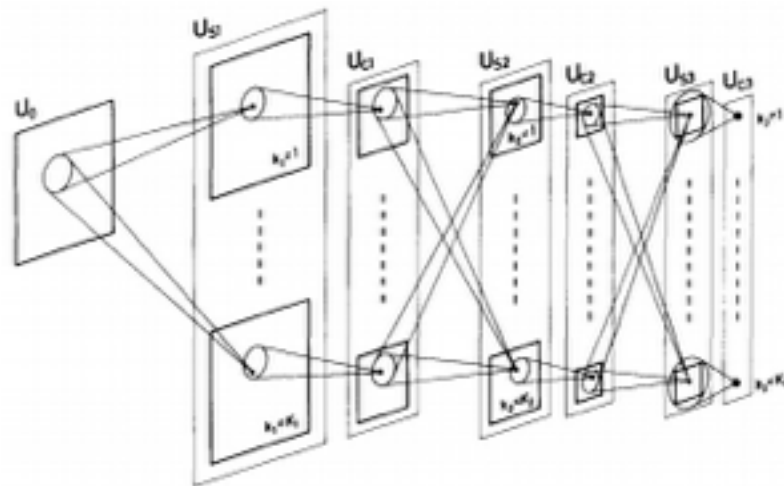


A bit of history:

Neurocognitron *[Fukushima 1980]*

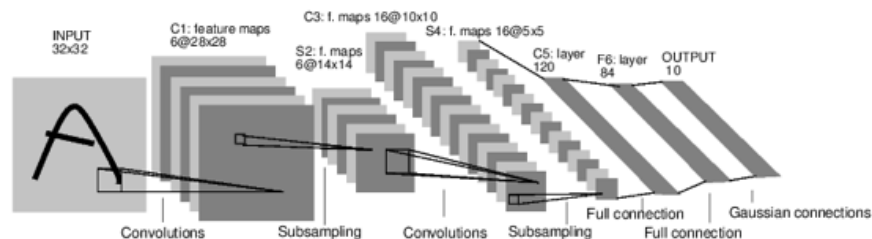


“sandwich” architecture (SCSCSC...)
simple cells: modifiable parameters
complex cells: perform pooling

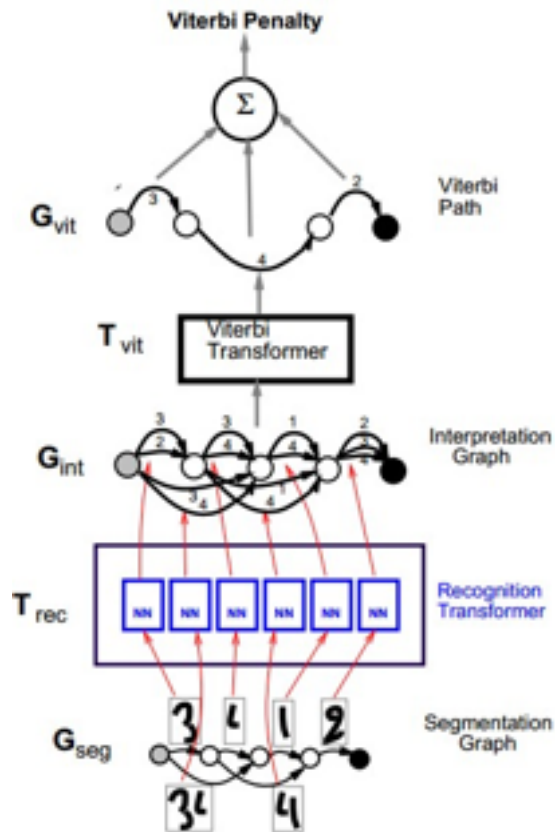


A bit of history: **Gradient-based learning applied to document recognition**

*[LeCun, Bottou, Bengio, Haffner
1998]*



LeNet-5



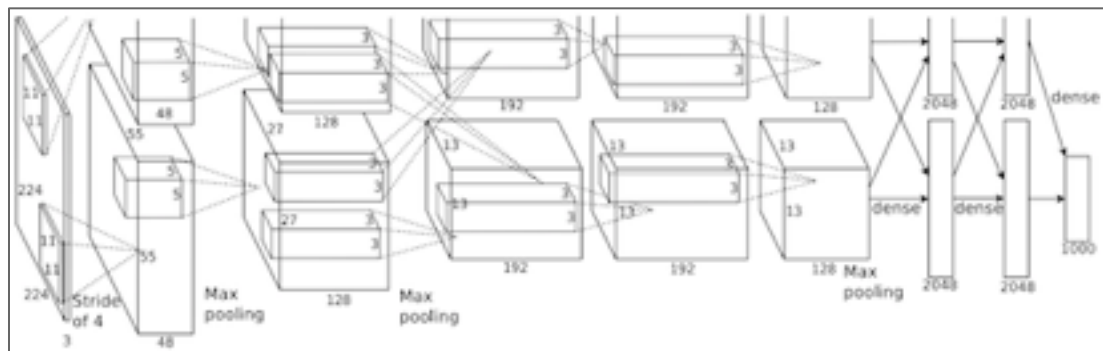
A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]



IMAGENET



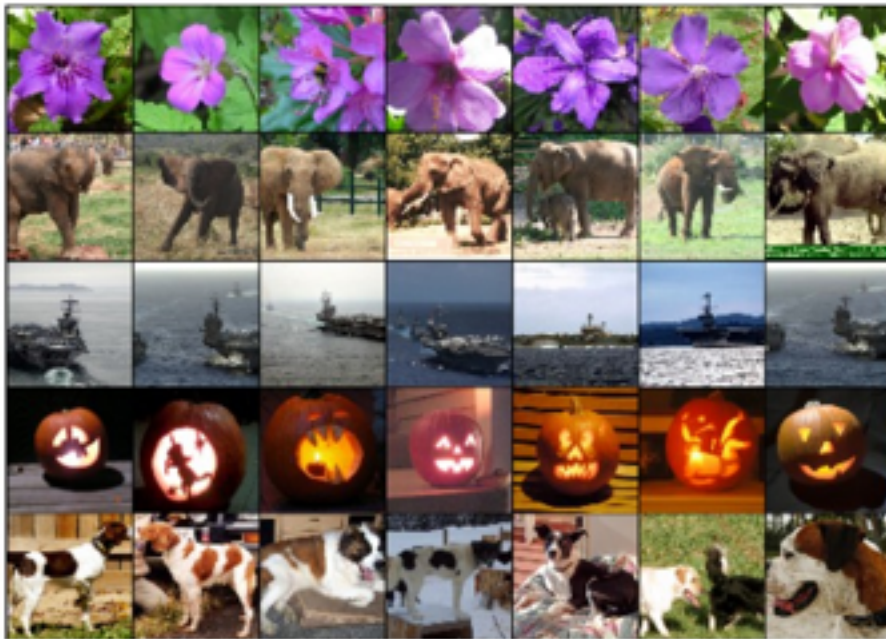
“AlexNet”

Fast-forward to today: ConvNets are everywhere

Classification

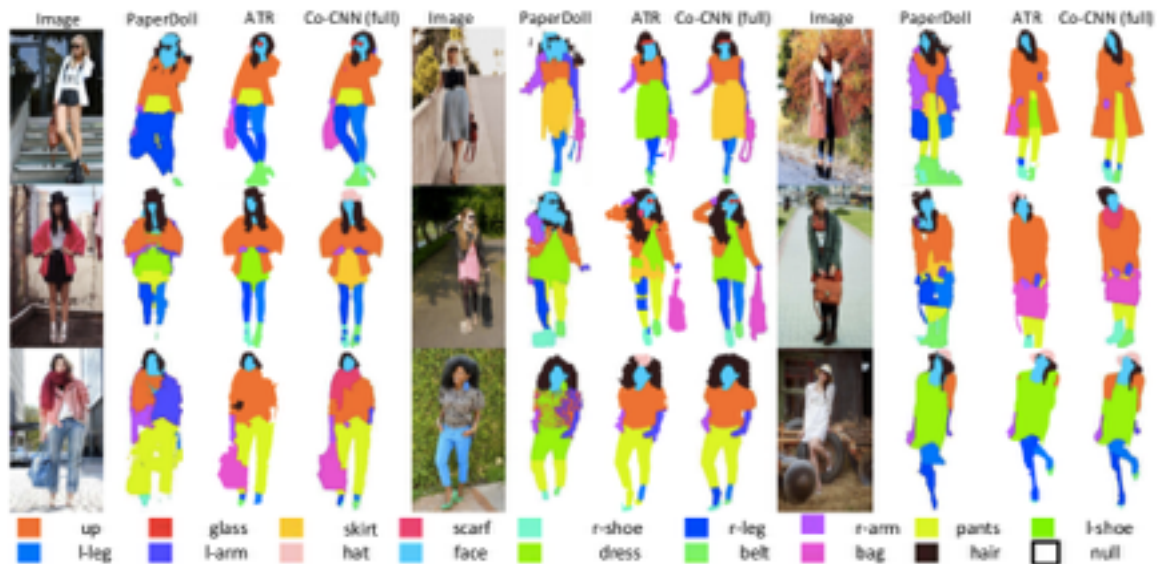


Retrieval

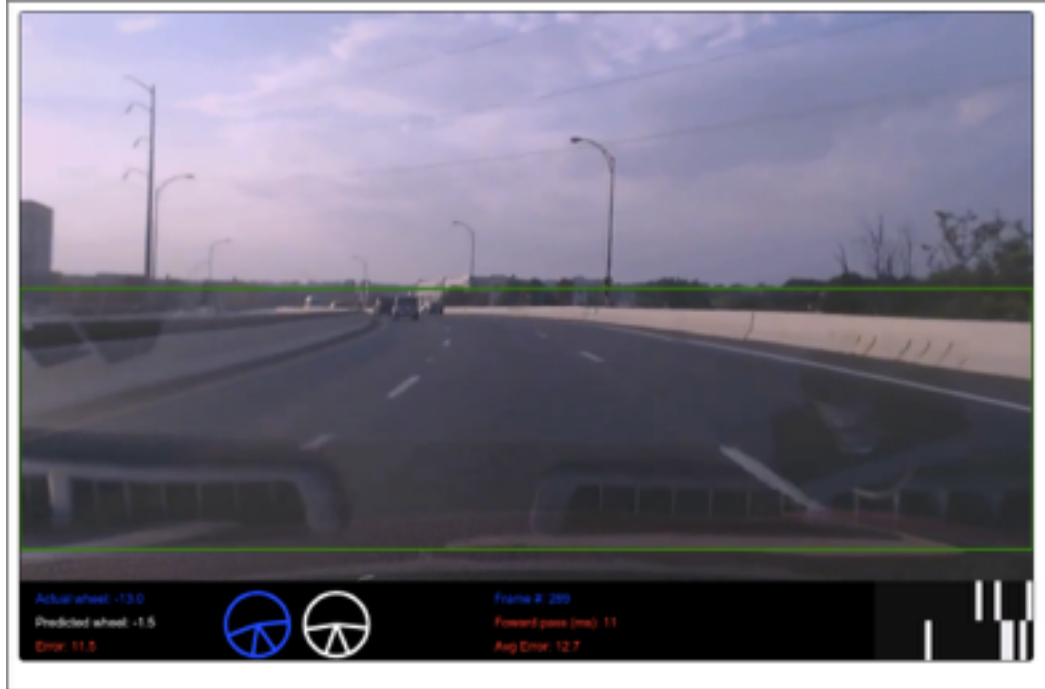


[Krizhevsky 2012]



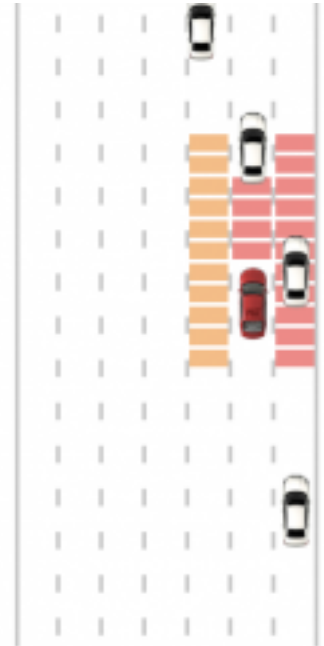


Self-Driving Cars



Road Overlay:

Safety System ↑



Road Overlay:

Safety System ↑





What is the color of the
coat?

Traditional VQA: analyze the whole
image -> analyze question -> give
answer: **brown**

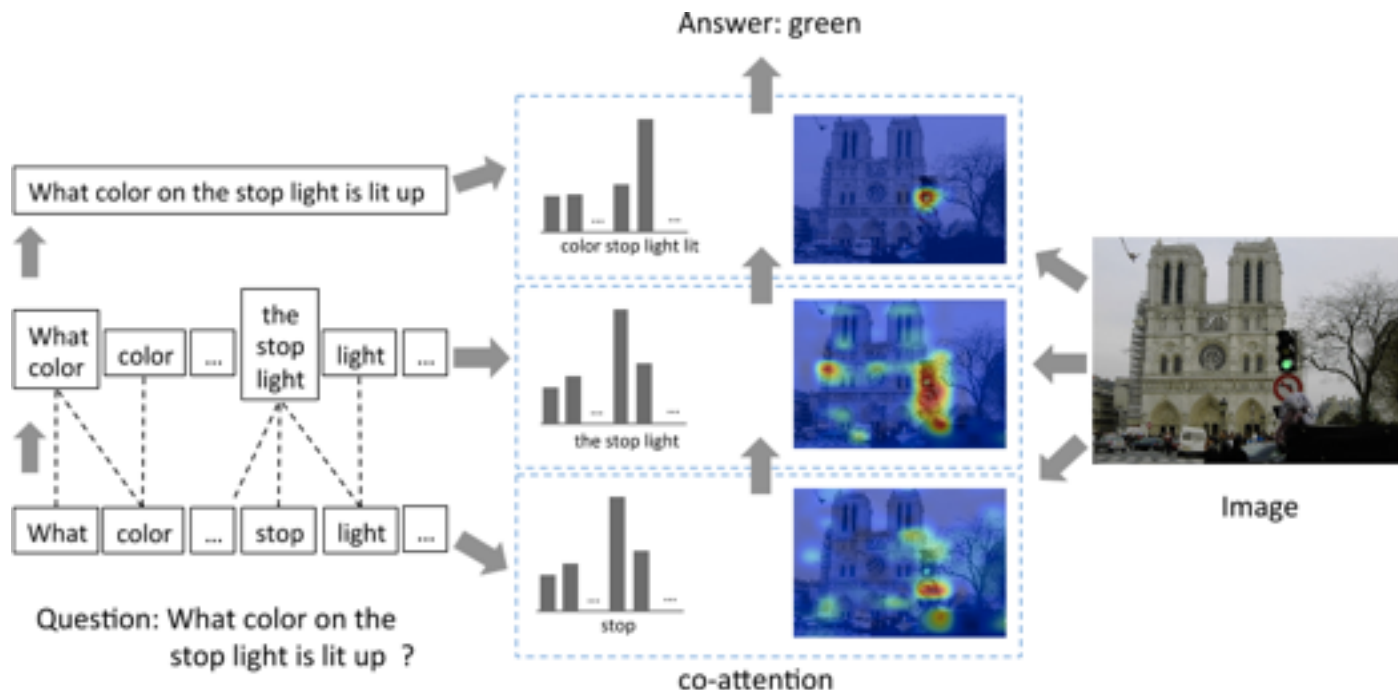
Attention based VQA: find coat ->
judge the color of coat -> give
answer: **yellow**



What is the color of the
umbrella?

Traditional VQA: analyze the whole
image -> analyze question -> give
answer: **green**

Attention based VQA: find umbrella
-> judge the color of umbrella -> give
answer: **red**



Caffe

<http://caffe.berkeleyvision.org>

Caffe Overview

- From U.C. Berkeley
- Written in C++
- Has Python and MATLAB bindings
- Good for training or finetuning feedforward models

Most important tip...

Don't be afraid to read the code!

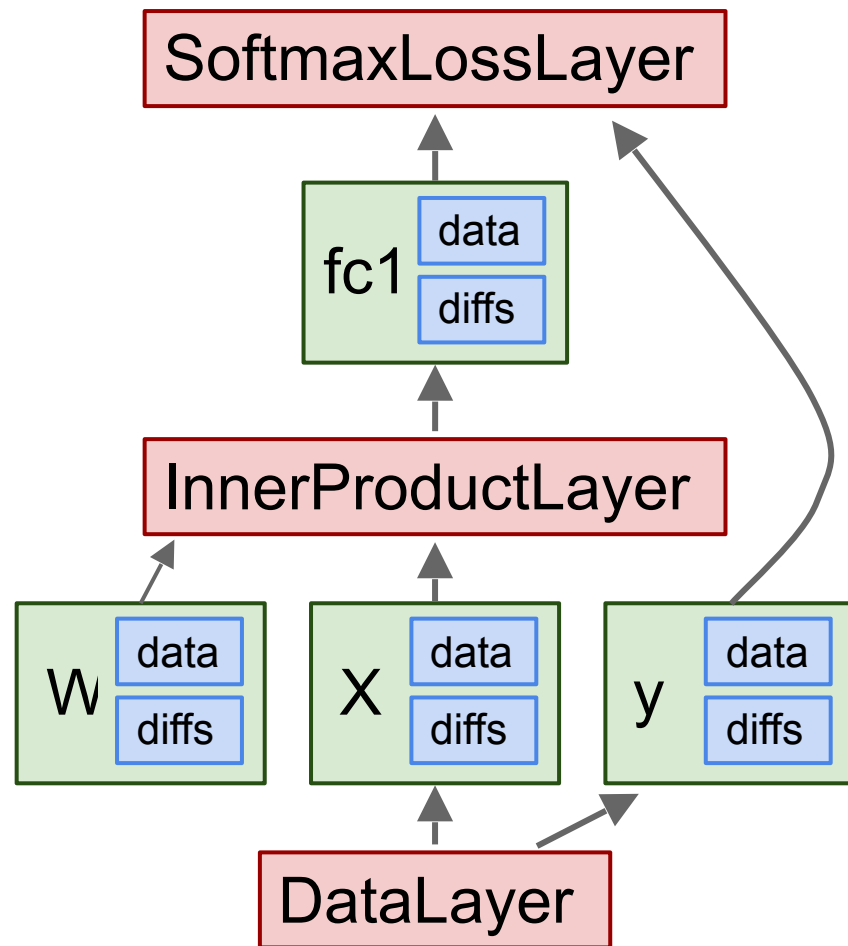
Caffe: Main classes

Blob: Stores data and derivatives ([header source](#))

Layer: Transforms bottom blobs to top blobs ([header + source](#))

Net: Many layers; computes gradients via forward / backward ([header source](#))

Solver: Uses gradients to update weights ([header source](#))



Caffe: Protocol Buffers

“Typed JSON”
from Google

Define “message types”
in .proto files

.proto file

```
message Person {  
  required string name = 1;  
  required int32 id = 2;  
  optional string email = 3;  
}
```

<https://developers.google.com/protocol-buffers/>

Caffe: Protocol Buffers

“Typed JSON”
from Google

Define “message types”
in .proto files

Serialize instances to text
files (.prototxt)

.proto file

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

.prototxt file

```
name: "John Doe"  
id: 1234  
email: "jdoe@example.com"
```

<https://developers.google.com/protocol-buffers/>

Caffe: Protocol Buffers

```
64 message NetParameter {
65   optional string name = 1; // consider giving the network a name
66   // The input blobs to the network.
67   repeated string input = 3;
68   // The shape of the input blobs.
69   repeated BlobShape input_shape = 8;
70
71   // 4D input dimensions -- deprecated. Use "shape" instead.
72   // If specified, for each input blob there should be four
73   // values specifying the num, channels, height and width of the input blob.
74   // Thus, there should be a total of {4 * #input} numbers.
75   repeated int32 input_dim = 4;
76
77   // Whether the network will force every layer to carry out backward operation.
78   // If set False, then whether to carry out backward is determined
79   // automatically according to the net structure and learning rates.
80   optional bool force_backward = 5 [default = false];
81   // The current "state" of the network, including the phase, level, and stage.
82   // Some layers may be included/excluded depending on this state and the states
83   // specified in the layers' include and exclude fields.
84   optional NetState state = 6;
85
86   // Print debugging information about results while running Net::Forward,
87   // Net::Backward, and Net::Update.
88   optional bool debug_info = 7 [default = false];
```

```
102 message SolverParameter {
103   // Specifying the train and test networks
104   //
105   // Exactly one train net must be specified using one of the following fields:
106   //   train_net_param, train_net, net_param, net
107   // One or more test nets may be specified using any of the following fields:
108   //   test_net_param, test_net, net_param, net
109   // If more than one test net field is specified (e.g., both net and
110   // test_net are specified), they will be evaluated in the field order given
111   // above: (1) test_net_param, (2) test_net, (3) net_param/net.
112   // A test_iter must be specified for each test_net.
113   // A test_level and/or a test_stage may also be specified for each test_net.
114   //
115   // Proto filename for the train net, possibly combined with one or more
116   // test nets.
117   optional string net = 24;
118   // Inline train net param, possibly combined with one or more test nets.
119   optional NetParameter net_param = 25;
120
121   optional string train_net = 1; // Proto filename for the train net.
```

<https://github.com/BVLC/caffe/blob/master/src/caffe/proto/caffe.proto>

<- All Caffe proto types defined here, good documentation!

Caffe: Training / Finetuning

No need to write code!

1. Convert data (run a script)
2. Define net (edit prototxt)
3. Define solver (edit prototxt)
4. Train (with pretrained weights) (run a script)

Caffe Step 1: Convert Data

- DataLayer reading from LMDB is the easiest
- Create LMDB using [convert_imageset](#)
- Create HDF5 file yourself using h5py
- From memory, using Python (MemoryLayer)

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

```
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

← Layers and Blobs
← often have same name!

```
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs
often have same
name!

Learning rates
(weight + bias)

Regularization
(weight + bias)

```
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAX_LOSS
}
```


Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs
often have same
name!

Learning rates
(weight + bias)

Regularization
(weight + bias)

Number of output
classes

```
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs
often have same
name!

Set these to 0 to
freeze a layer

Learning rates
(weight + bias)

Regularization
(weight + bias)

Number of output
classes

```
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
}
layers {
  bottom: "fc1"
  bottom: "label"
  top: "loss"
  name: "loss"
  type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

- .prototxt can get ugly for big models
- ResNet-152 prototxt is 6775 lines long!
- Not “compositional”; can’t easily define a residual block and reuse

```
1  name: "ResNet-152"
2  input: "data"
3  input_dim: 1
4  input_dim: 3
5  input_dim: 224
6  input_dim: 224
7
8  layer {
9      bottom: "data"
10     top: "conv1"
11     name: "conv1"
12     type: "Convolution"
13     convolution_param {
14         num_output: 64
15         kernel_size: 7
16         pad: 3
17         stride: 2
18         bias_term: false
19     }
20 }
21
22 layer {
23     bottom: "conv1"
24     top: "conv1"
25     name: "bn_conv1"
26     type: "BatchNorm"
27     batch_norm_param {
28         use_global_stats: true
29     }
30 }
31
32 layer {
33     bottom: "conv1"
34     top: "pool5"
35     name: "pool5"
36     type: "Pooling"
37     pooling_param {
38         kernel_size: 7
39         stride: 1
40         pool: AVE
41     }
42 }
43
44 layer {
45     bottom: "pool5"
46     top: "fc1000"
47     name: "fc1000"
48     type: "InnerProduct"
49     inner_product_param {
50         num_output: 1000
51     }
52 }
53
54 layer {
55     bottom: "fc1000"
56     top: "prob"
57     name: "prob"
58     type: "Softmax"
59 }
```

<https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-152-deploy.prototxt>

Caffe Step 2: Define Net (finetuning)

Original prototxt:

```
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}
[...] ReLU, Dropout]
layer {
  name: "fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 1000
  }
}
```

Pretrained weights:

```
"fc7.weight": [values]
"fc7.bias": [values]
"fc8.weight": [values]
"fc8.bias": [values]
```

Modified prototxt:

```
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}
[...] ReLU, Dropout]
layer {
  name: "my-fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 10
  }
}
```

Caffe Step 2: Define Net (finetuning)

Original prototxt:

```
layer {  
  name: "fc7"  
  type: "InnerProduct"  
  inner_product_param {  
    num_output: 4096  
  }  
}  
[... ReLU, Dropout]  
layer {  
  name: "fc8"  
  type: "InnerProduct"  
  inner_product_param {  
    num_output: 1000  
  }  
}
```

Same name:
weights copied

Pretrained weights:

```
"fc7.weight": [values]  
"fc7.bias": [values]  
"fc8.weight": [values]  
"fc8.bias": [values]
```

Modified prototxt:

```
layer {  
  name: "fc7"  
  type: "InnerProduct"  
  inner_product_param {  
    num_output: 4096  
  }  
}  
[... ReLU, Dropout]  
layer {  
  name: "my-fc8"  
  type: "InnerProduct"  
  inner_product_param {  
    num_output: 10  
  }  
}
```

Caffe Step 2: Define Net (finetuning)

Original prototxt:

```
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}
[... ReLU, Dropout]
layer {
  name: "fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 1000
  }
}
```

Same name:
weights copied

Pretrained weights:

```
"fc7.weight": [values]
"fc7.bias": [values]
"fc8.weight": [values]
"fc8.bias": [values]
```

Different name:
weights reinitialized

Modified prototxt:

```
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}
[... ReLU, Dropout]
layer {
  name: "my-fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 10
  }
}
```

Caffe Step 3: Define Solver

Write a prototxt file defining a

[SolverParameter](#)

If finetuning, copy existing
solver.prototxt file

Change net to be your net

Change snapshot_prefix to your
output

Reduce base learning rate (divide
by 100)

Maybe change max_iter and
snapshot

```
1 net: "models/bvlc_alexnet/train_val.prototxt"
2 test_iter: 1000
3 test_interval: 1000
4 base_lr: 0.01
5 lr_policy: "step"
6 gamma: 0.1
7 stepsize: 100000
8 display: 20
9 max_iter: 450000
10 momentum: 0.9
11 weight_decay: 0.0005
12 snapshot: 10000
13 snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
14 solver_mode: GPU
```

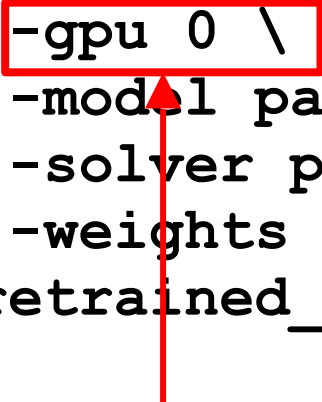
Caffe Step 4: Train!

```
./build/tools/caffe train \  
-gpu 0 \  
-model path/to/trainval.prototxt \  
-solver path/to/solver.prototxt \  
-weights path/to/  
pretrained_weights.caffemodel
```

<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe Step 4: Train!

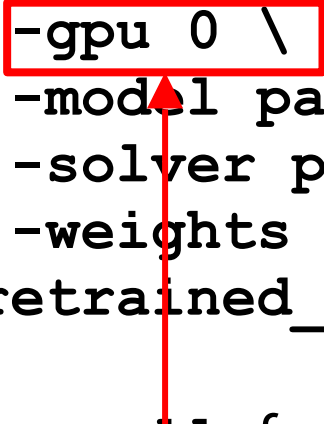
```
./build/tools/caffe train \  
-gpu 0 \  
-model path/to/trainval.prototxt \  
-solver path/to/solver.prototxt \  
-weights path/to/  
pretrained_weights.caffemodel  
  
-gpu -1 for CPU mode
```



<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe Step 4: Train!

```
./build/tools/caffe train \  
-gpu 0 \  
-model path/to/trainval.prototxt \  
-solver path/to/solver.prototxt \  
-weights path/to/  
pretrained_weights.caffemodel  
  
-gpu all for multi-GPU data parallelism
```



<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe: Model Zoo

- AlexNet, VGG, GoogLeNet, ResNet, plus others



<https://github.com/BVLC/caffe/wiki/Model-Zoo>

Caffe: Python Interface

Not much documentation...

Look at Notebooks in `caffe/examples`

Read the code! Two most important files:

[caffe/python/caffe/_caffe.cpp](#):

Exports Blob, Layer, Net, and Solver classes

[caffe/python/caffe/pycaffe.py](#)

Adds extra methods to Net class

Caffe: Python Interface

- Good for:
 - Interfacing with numpy
 - Extract features: Run net forward
 - Compute gradients: Run net backward (DeepDream, etc)
 - Define layers in Python with numpy (CPU only)

Caffe Pros / Cons

- (+) Good for feedforward networks
- (+) Good for finetuning existing networks
- (+) Train models without writing any code!
- (+) Python interface is pretty useful!
- (-) Need to write C++ / CUDA for new GPU layers
- (-) Not good for recurrent networks
- (-) Cumbersome for big networks (GoogLeNet, ResNet)

Caffe: Blobs

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}
28
29     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30     explicit Blob(const int num, const int channels, const int height,
31                 const int width);
32     explicit Blob(const vector<int>& shape);
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 const Dtype* cpu_data() const;
120 void set_cpu_data(Dtype* data);
121 const int* gpu_shape() const;
122 const Dtype* gpu_data() const;
123 const Dtype* cpu_diff() const;
124 const Dtype* gpu_diff() const;
125 Dtype* mutable_cpu_data();
126 Dtype* mutable_gpu_data();
127 Dtype* mutable_cpu_diff();
128 Dtype* mutable_gpu_diff();
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 protected:
169     shared_ptr<SyncedMemory> data_;
170     shared_ptr<SyncedMemory> diff_;
171     shared_ptr<SyncedMemory> shape_data_;
172     vector<int> shape_;
173     int count_;
174     int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations and
weights

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}
28
29     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30     explicit Blob(const int num, const int channels, const int height,
31                  const int width);
32     explicit Blob(const vector<int>& shape);
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 const Dtype* cpu_data() const;
120 void set_cpu_data(Dtype* data);
121 const int* gpu_shape() const;
122 const Dtype* gpu_data() const;
123 const Dtype* cpu_diff() const;
124 const Dtype* gpu_diff() const;
125 Dtype* mutable_cpu_data();
126 Dtype* mutable_gpu_data();
127 Dtype* mutable_cpu_diff();
128 Dtype* mutable_gpu_diff();
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 protected:
169     shared_ptr<SyncedMemory> data_;
170     shared_ptr<SyncedMemory> diff_;
171     shared_ptr<SyncedMemory> shape_data_;
172     vector<int> shape_;
173     int count_;
174     int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations and
weights

Two parallel tensors:
data: values
diffs: gradients

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}
28
29     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30     explicit Blob(const int num, const int channels, const int height,
31         const int width);
32     explicit Blob(const vector<int>& shape);
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 const Dtype* cpu_data() const;
120 void set_cpu_data(Dtype* data);
121 const int* gpu_shape() const;
122 const Dtype* gpu_data() const;
123 const Dtype* cpu_diff() const;
124 const Dtype* gpu_diff() const;
125 Dtype* mutable_cpu_data();
126 Dtype* mutable_gpu_data();
127 Dtype* mutable_cpu_diff();
128 Dtype* mutable_gpu_diff();
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 protected:
169     shared_ptr<SyncedMemory> data_;
170     shared_ptr<SyncedMemory> diff_;
171     shared_ptr<SyncedMemory> shape_data_;
172     vector<int> shape_;
173     int count_;
174     int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations
and weights

Two parallel tensors:
data: values
diffs: gradients

Stores CPU / GPU
versions of each
tensor

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}
28
29     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30     explicit Blob(const int num, const int channels, const int height,
31                 const int width);
32     explicit Blob(const vector<int>& shape);
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 const Dtype* cpu_data() const;
120 void set_cpu_data(Dtype* data);
121 const int* gpu_shape() const;
122 const Dtype* gpu_data() const;
123 const Dtype* cpu_diff() const;
124 const Dtype* gpu_diff() const;
125 Dtype* mutable_cpu_data();
126 Dtype* mutable_gpu_data();
127 Dtype* mutable_cpu_diff();
128 Dtype* mutable_gpu_diff();
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 protected:
169     shared_ptr<SyncedMemory> data_;
170     shared_ptr<SyncedMemory> diff_;
171     shared_ptr<SyncedMemory> shape_data_;
172     vector<int> shape_;
173     int count_;
174     int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Layer

A small unit of
computation

```
32 template <typename Dtype>
33 class Layer {
34 public:
35     /** @brief Using the CPU device, compute the layer output. */
36     virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
37                             const vector<Blob<Dtype>*>& top) = 0;
38     /**
39      * @brief Using the GPU device, compute the layer output.
40      *      Fall back to Forward_cpu() if unavailable.
41      */
42     virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
43                             const vector<Blob<Dtype>*>& top) {
44         // LOG(WARNING) << "Using CPU code as backup.";
45         return Forward_cpu(bottom, top);
46     }
47     /**
48      * @brief Using the CPU device, compute the gradients for any parameters and
49      *      for the bottom blobs if propagate_down is true.
50      */
51     virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
52                              const vector<bool>& propagate_down,
53                              const vector<Blob<Dtype>*>& bottom) = 0;
54     /**
55      * @brief Using the GPU device, compute the gradients for any parameters and
56      *      for the bottom blobs if propagate_down is true.
57      *      Fall back to Backward_cpu() if unavailable.
58      */
59     virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
60                              const vector<bool>& propagate_down,
61                              const vector<Blob<Dtype>*>& bottom) {
62         // LOG(WARNING) << "Using CPU code as backup.";
63         Backward_cpu(top, propagate_down, bottom);
64     }
65 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

A small unit of computation

Forward: Use “bottom”
data to compute “top”
data

```
32 template <typename Dtype>
33 class Layer {
34 public:
334 /** @brief Using the CPU device, compute the layer output. */
335 virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
336 const vector<Blob<Dtype>*>& top) = 0;
337 /**
338  * @brief Using the GPU device, compute the layer output.
339  *      Fall back to Forward_cpu() if unavailable.
340  */
341 virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
342 const vector<Blob<Dtype>*>& top) {
343 // LOG(WARNING) << "Using CPU code as backup.";
344 return Forward_cpu(bottom, top);
345 }
346
347 /**
348  * @brief Using the CPU device, compute the gradients for any parameters and
349  *      for the bottom blobs if propagate_down is true.
350  */
351 virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
352 const vector<bool>& propagate_down,
353 const vector<Blob<Dtype>*>& bottom) = 0;
354 /**
355  * @brief Using the GPU device, compute the gradients for any parameters and
356  *      for the bottom blobs if propagate_down is true.
357  *      Fall back to Backward_cpu() if unavailable.
358  */
359 virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
360 const vector<bool>& propagate_down,
361 const vector<Blob<Dtype>*>& bottom) {
362 // LOG(WARNING) << "Using CPU code as backup.";
363 Backward_cpu(top, propagate_down, bottom);
364 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

A small unit of computation

Forward: Use “bottom” data
to compute “top” data

Backward: Use “top” diffs
to compute “bottom” diffs

```
32 template <typename Dtype>
33 class Layer {
34 public:
35
36     /** @brief Using the CPU device, compute the layer output. */
37     virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
38                             const vector<Blob<Dtype>*>& top) = 0;
39
40     /**
41      * @brief Using the GPU device, compute the layer output.
42      *      Fall back to Forward_cpu() if unavailable.
43      */
44     virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
45                              const vector<Blob<Dtype>*>& top) {
46         // LOG(WARNING) << "Using CPU code as backup.";
47         return Forward_cpu(bottom, top);
48     }
49
50     /**
51      * @brief Using the CPU device, compute the gradients for any parameters and
52      *      for the bottom blobs if propagate_down is true.
53      */
54     virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
55                               const vector<bool>& propagate_down,
56                               const vector<Blob<Dtype>*>& bottom) = 0;
57
58     /**
59      * @brief Using the GPU device, compute the gradients for any parameters and
60      *      for the bottom blobs if propagate_down is true.
61      *      Fall back to Backward_cpu() if unavailable.
62      */
63     virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
64                               const vector<bool>& propagate_down,
65                               const vector<Blob<Dtype>*>& bottom) {
66         // LOG(WARNING) << "Using CPU code as backup.";
67         Backward_cpu(top, propagate_down, bottom);
68     }
69 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

A small unit of computation

Forward: Use “bottom” data to compute “top” data

Backward: Use “top” diffs to compute “bottom” diffs

Separate **CPU** / **GPU** implementations

```
32 template <typename Dtype>
33 class Layer {
34 public:
334 /** @brief Using the CPU device, compute the layer output. */
335 virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
336 const vector<Blob<Dtype>*>& top) = 0;
337
338 * @brief Using the GPU device, compute the layer output.
339 * Fall back to Forward_cpu() if unavailable.
340 */
341 virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
342 const vector<Blob<Dtype>*>& top) {
343 // LOG(WARNING) << "Using CPU code as backup.";
344 return Forward_cpu(bottom, top);
345 }
346
347 /**
348 * @brief Using the CPU device, compute the gradients for any parameters and
349 * for the bottom blobs if propagate_down is true.
350 */
351 virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
352 const vector<bool>& propagate_down,
353 const vector<Blob<Dtype>*>& bottom) = 0;
354
355 * @brief Using the GPU device, compute the gradients for any parameters and
356 * for the bottom blobs if propagate_down is true.
357 * Fall back to Backward_cpu() if unavailable.
358 */
359 virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
360 const vector<bool>& propagate_down,
361 const vector<Blob<Dtype>*>& bottom) {
362 // LOG(WARNING) << "Using CPU code as backup.";
363 Backward_cpu(top, propagate_down, bottom);
364 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

Tons of different layer types:

Branch: master ▾ caffe / src / caffe / layers /

jeffdonahue Remove incorrect cast of gemm int arg to Dtype in BiasLayer

..

absval_layer.cpp	dismantle layer headers
absval_layer.cu	dismantle layer headers
accuracy_layer.cpp	dismantle layer headers
argmax_layer.cpp	dismantle layer headers
base_conv_layer.cpp	enable dilated deconvolution
base_data_layer.cpp	dismantle layer headers
base_data_layer.cu	dismantle layer headers
batch_norm_layer.cpp	dismantle layer headers
batch_norm_layer.cu	dismantle layer headers
...	
conv_layer.cpp	add support for 2D dilated convolution
conv_layer.cu	dismantle layer headers
cudnn_conv_layer.cpp	dismantle layer headers
cudnn_conv_layer.cu	Fix CuDNNConvolutionLayer for cuDNN v4

<https://github.com/BVLC/caffe/tree/master/src/caffe/layers>

Caffe: Layer

Tons of different layer types:

batch norm

convolution

cuDNN convolution

.cpp: CPU implementation

.cu: GPU implementation

Branch: master caffe / src / caffe / layers /

jeffdonahue Remove incorrect cast of gemm int arg to Dtype in BiasLayer

..	
absval_layer.cpp	dismantle layer headers
absval_layer.cu	dismantle layer headers
accuracy_layer.cpp	dismantle layer headers
argmax_layer.cpp	dismantle layer headers
base_conv_layer.cpp	enable dilated deconvolution
base_data_layer.cpp	dismantle layer headers
base_data_layer.cu	dismantle layer headers
batch_norm_layer.cpp	dismantle layer headers
batch_norm_layer.cu	dismantle layer headers
...	
conv_layer.cpp	add support for 2D dilated convolution
conv_layer.cu	dismantle layer headers
cudnn_conv_layer.cpp	dismantle layer headers
cudnn_conv_layer.cu	Fix CuDNNConvolutionLayer for cuDNN v4

<https://github.com/BVLC/caffe/tree/master/src/caffe/layers>

Caffe: Layer

Collects layers into a DAG

Run all or part of the net
forward and **backward**

```
23 template <typename Dtype>
24 class Net {
25 public:
26     explicit Net(const NetParameter& param, const Net* root_net = NULL);
27     explicit Net(const string& param_file, Phase phase,
28                 const Net* root_net = NULL);
29     virtual ~Net() {}

41 /**
42  * The From and To variants of Forward and Backward operate on the
43  * (topological) ordering by which the net is specified. For general DAG
44  * networks, note that (1) computing from one layer to another might entail
45  * extra computation on unrelated branches, and (2) computation starting in
46  * the middle may be incorrect if all of the layers of a fan-in are not
47  * included.
48  */
49     Dtype ForwardFromTo(int start, int end);
50     Dtype ForwardFrom(int start);
51     Dtype ForwardTo(int end);
52     /// @brief Run forward using a set of bottom blobs, and return the result.
53     const vector<Blob<Dtype>*>& Forward(const vector<Blob<Dtype>* > & bottom,
54                                         Dtype* loss = NULL);

67 /**
68  * The network backward should take no input and output, since it solely
69  * computes the gradient w.r.t the parameters, and the data has already been
70  * provided during the forward pass.
71  */
72     void Backward();
73     void BackwardFromTo(int start, int end);
74     void BackwardFrom(int start);
75     void BackwardTo(int end);
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/net.hpp>

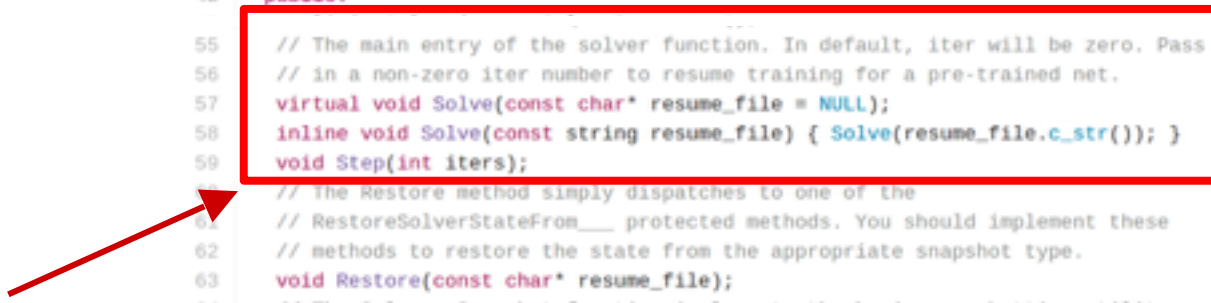
Caffe: Solver

```
40 template <typename Dtype>
41 class Solver {
42 public:
43     .....
44     .....
45     .....
46     .....
47     .....
48     .....
49     .....
50     .....
51     .....
52     .....
53     .....
54     .....
55     // The main entry of the solver function. In default, iter will be zero. Pass
56     // in a non-zero iter number to resume training for a pre-trained net.
57     virtual void Solve(const char* resume_file = NULL);
58     inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
59     void Step(int iters);
60     // The Restore method simply dispatches to one of the
61     // RestoreSolverStateFrom___ protected methods. You should implement these
62     // methods to restore the state from the appropriate snapshot type.
63     void Restore(const char* resume_file);
64     // The Solver::Snapshot function implements the basic snapshotting utility
65     // that stores the learned net. You should implement the SnapshotSolverState()
66     // function that produces a SolverState protocol buffer that needs to be
67     // written to disk together with the learned net.
68     void Snapshot();
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/solver.hpp>

Caffe: Solver

```
40  template <typename Dtype>
41  class Solver {
42  public:
43      // The main entry of the solver function. In default, iter will be zero. Pass
44      // in a non-zero iter number to resume training for a pre-trained net.
45      virtual void Solve(const char* resume_file = NULL);
46      inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
47      void Step(int iters);
48      // The Restore method simply dispatches to one of the
49      // RestoreSolverStateFrom___ protected methods. You should implement these
50      // methods to restore the state from the appropriate snapshot type.
51      void Restore(const char* resume_file);
52      // The Solver::Snapshot function implements the basic snapshotting utility
53      // that stores the learned net. You should implement the SnapshotSolverState()
54      // function that produces a SolverState protocol buffer that needs to be
55      // written to disk together with the learned net.
56      void Snapshot();
```



Trains a Net by running it forward / backward, updating weights

<https://github.com/BVLC/caffe/blob/master/include/caffe/solver.hpp>

Caffe: Solver

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

Trains a Net by running it forward / backward, updating weights

Handles snapshotting, restoring from snapshots

<https://github.com/BVLC/caffe/blob/master/include/caffe/solver.hpp>

Caffe: Solver

Trains a Net by running it
forward / backward,
updating weights

Handles snapshotting,
restoring from snapshots

Subclasses implement
different update rules



```
40  template <typename Dtype>
41  class Solver {
42  public:
43      ...
44      ...
45      ...
46      ...
47      ...
48      ...
49      ...
50      ...
51      ...
52      ...
53      ...
54      ...
55      // The main entry of the solver function. In default, iter will be zero. Pass
56      // in a non-zero iter number to resume training for a pre-trained net.
57      virtual void Solve(const char* resume_file = NULL);
58      inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
59      void Step(int iters);
60      // The Restore method simply dispatches to one of the
61      // RestoreSolverStateFrom___ protected methods. You should implement these
62      // methods to restore the state from the appropriate snapshot type.
63      void Restore(const char* resume_file);
64      // The Solver::Snapshot function implements the basic snapshotting utility
65      // that stores the learned net. You should implement the SnapshotSolverState()
66      // function that produces a SolverState protocol buffer that needs to be
67      // written to disk together with the learned net.
68      void Snapshot();
```

```
15  template <typename Dtype>
16  class SGDSolver : public Solver<Dtype> {
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  template <typename Dtype>
84  class RMSPropSolver : public SGDSolver<Dtype> {
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  template <typename Dtype>
131  class AdamSolver : public SGDSolver<Dtype> {
```

https://github.com/BVLC/caffe/blob/master/include/caffe/sgd_solvers.hpp

Overview

	Caffe	Torch	Theano	TensorFlow
Language	C++, Python	Lua	Python	Python
Pretrained	Yes ++	Yes ++	Yes (Lasagne)	Inception
Multi-GPU: Data parallel	Yes	Yes cunn.DataParallelTable	Yes platoon	Yes
Multi-GPU: Model parallel	No	Yes fbcunn.ModelParallel	Experimental	Yes (best)
Readable source code	Yes (C++)	Yes (Lua)	No	No
Good at RNN	No	Mediocre	Yes	Yes (best)