Lecture 6: Training Neural Networks, Part II

Tuesday February 7, 2017

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

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, 1.3999999999
```

- 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,  0.3483],
       [-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]])

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]])
```

This is better

Pixel Recursive Super Resolution

Ryan Dahl * Mohammad Norouzi Jonathon Shlens

Google Brain

{rld,mnorouzi,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.

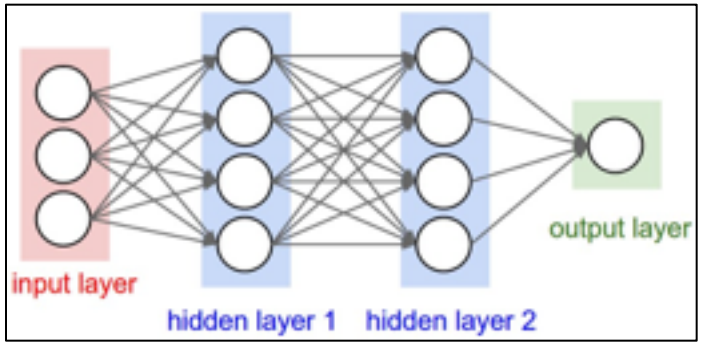
New work out on Feb 2

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

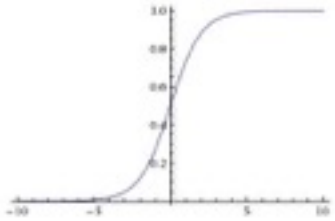


* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

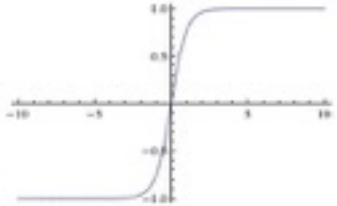
Activation Functions

Sigmoid

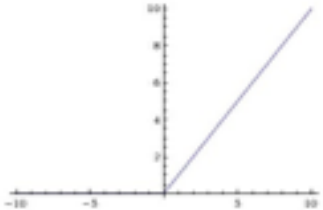
$$\sigma(x) = 1/(1+e^{-x})$$



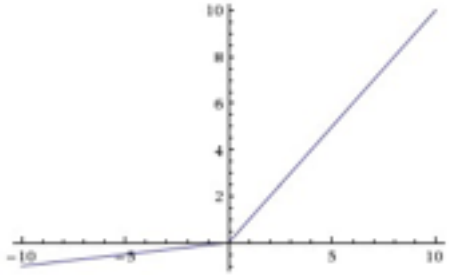
tanh tanh(x)



ReLU max(0,x)



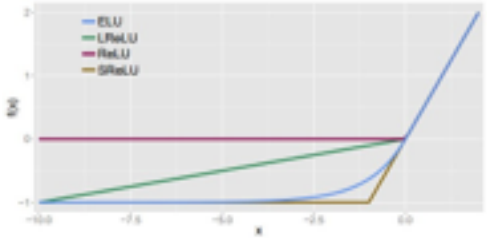
Leaky ReLU
max(0.1x, x)



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

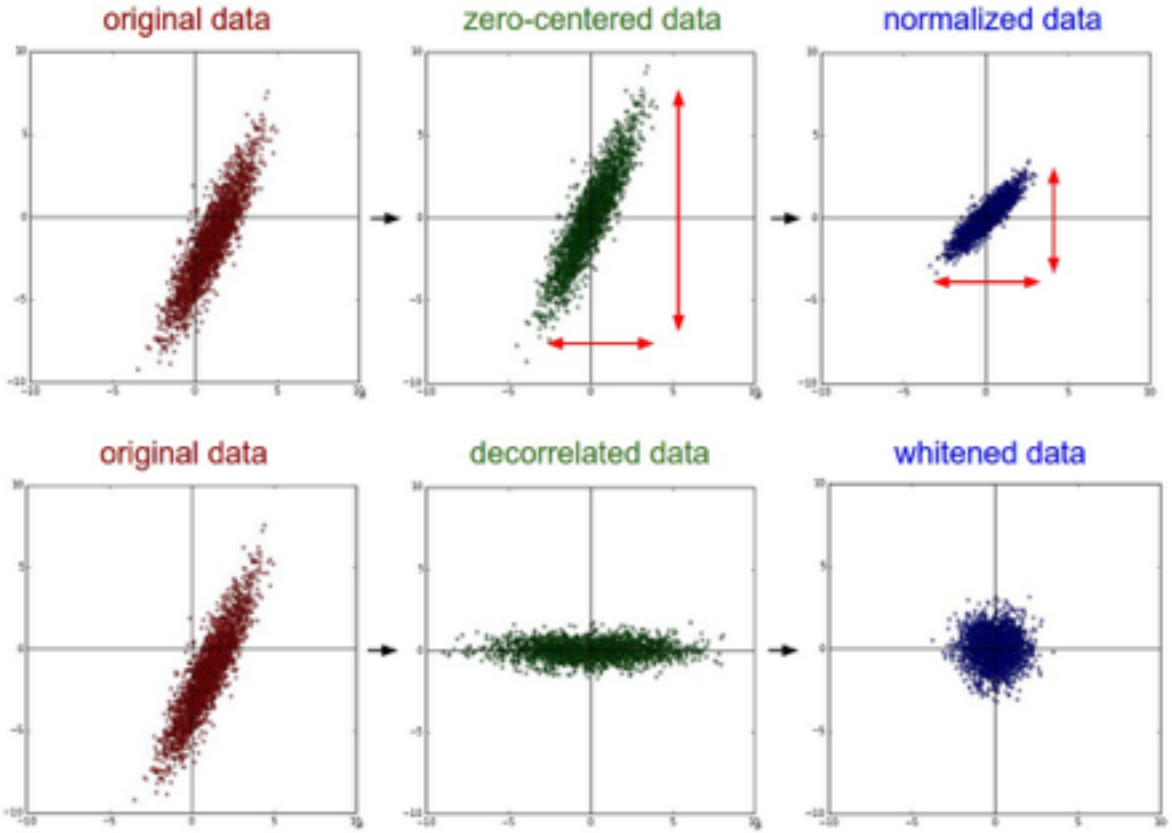
ELU

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



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

Data Preprocessing

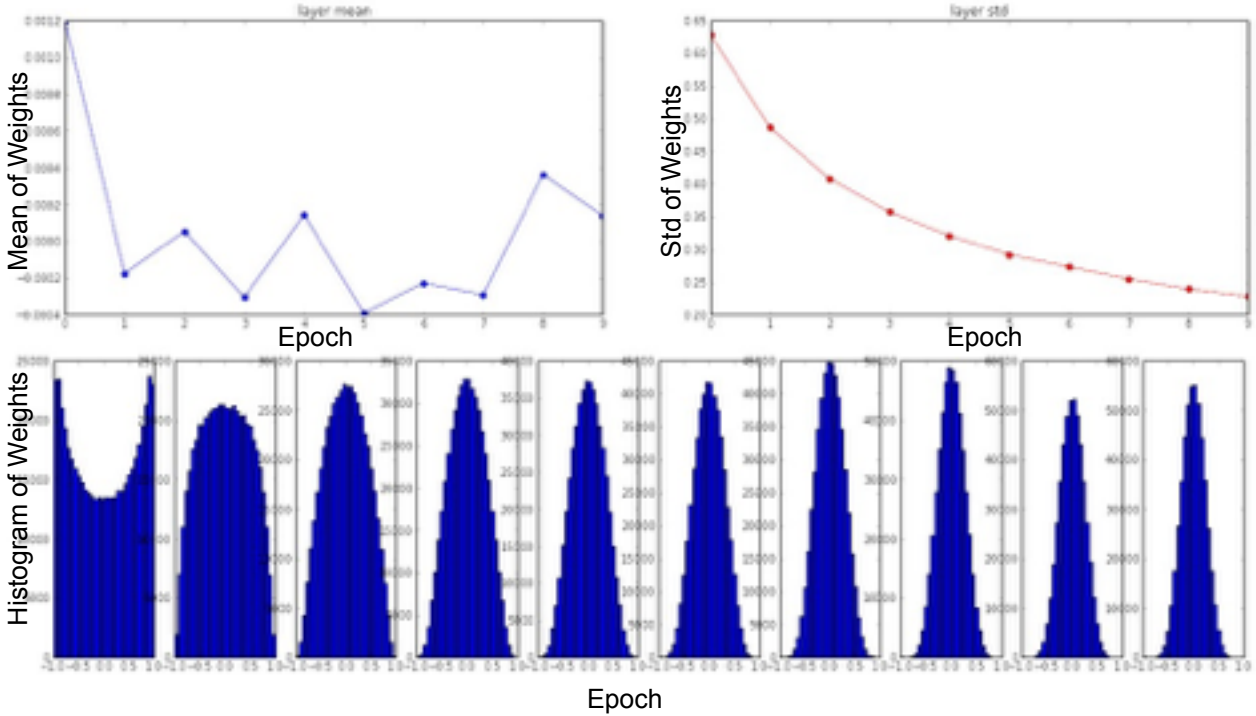


* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

```
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:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

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

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

- 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

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

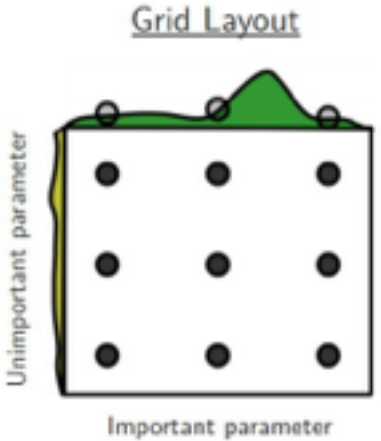
Babysitting the learning process

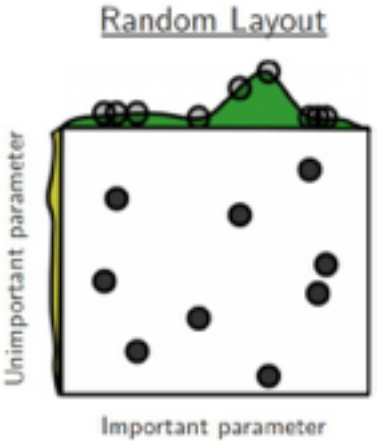
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, 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: 0.158000, 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 2.302466, 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 2.302459, train: 0.206000, 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





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

Today:

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

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

Parameter Updates

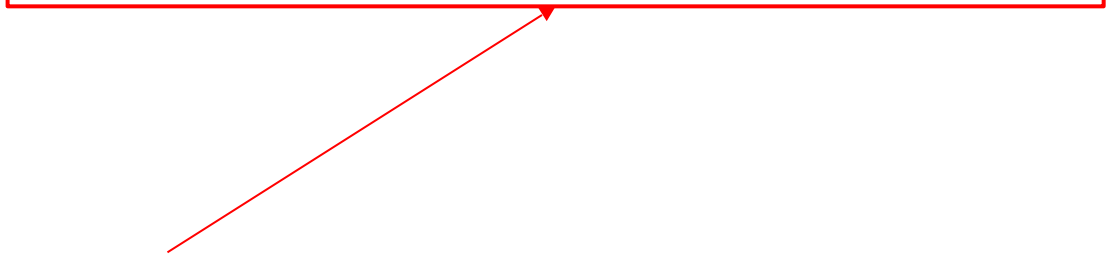
Training a neural network, main loop:

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

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

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.

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

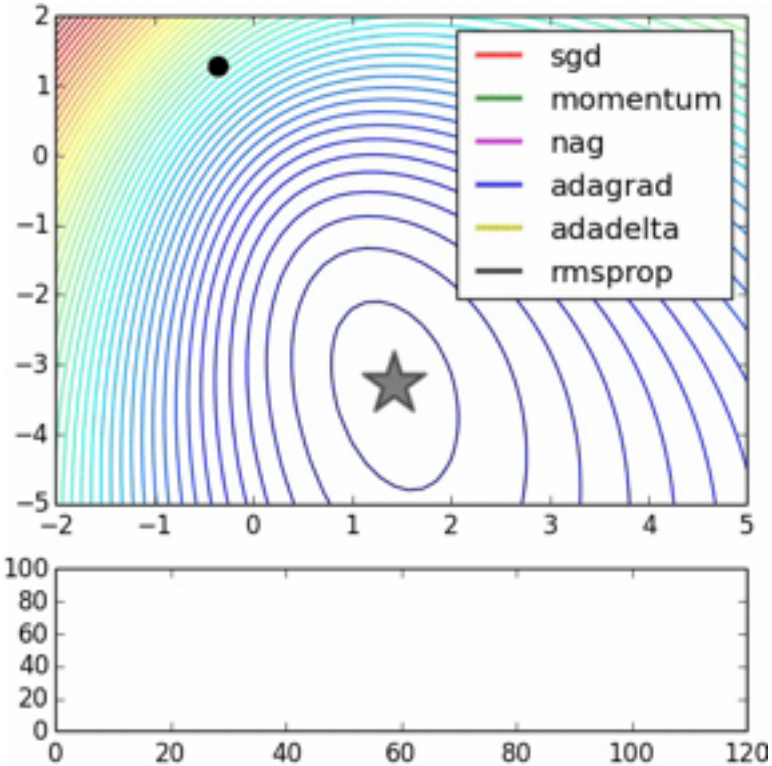
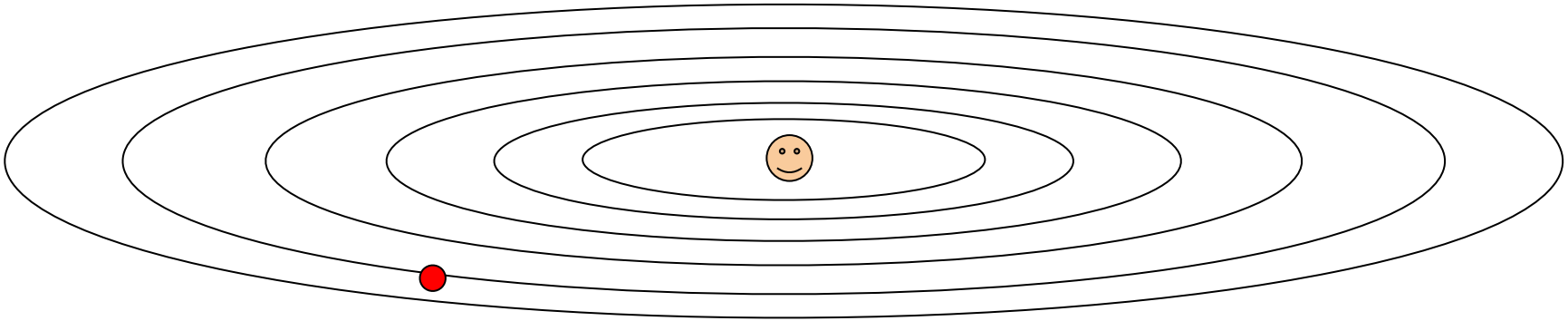


Image credits:
Alec Radford

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

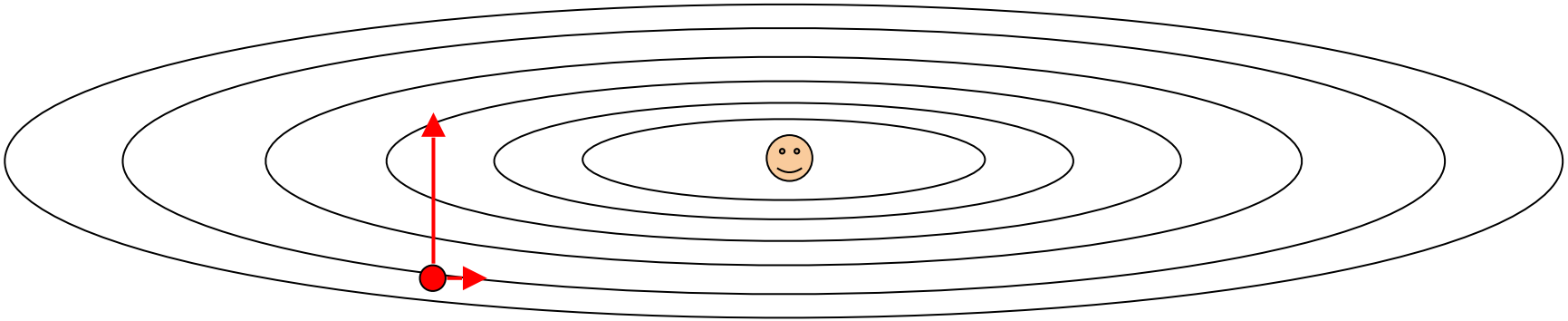
Suppose loss function is steep vertically but shallow horizontally:



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

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

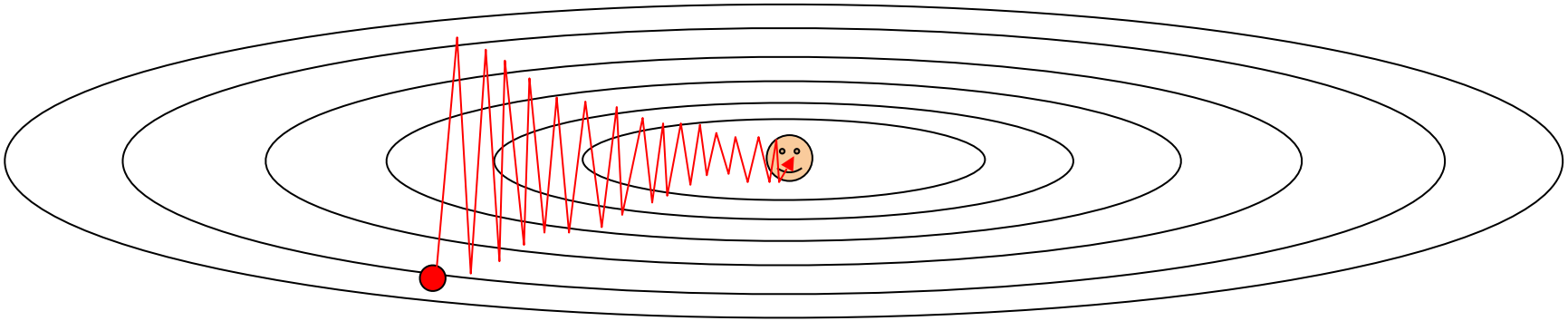
Suppose loss function is steep vertically but shallow horizontally:



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

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

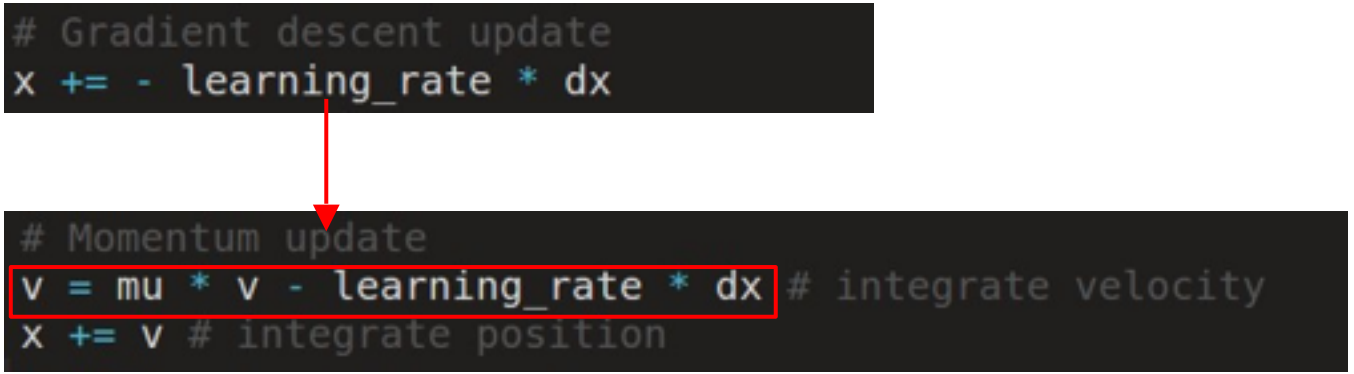
Suppose loss function is steep vertically but shallow horizontally:



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

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

Momentum update

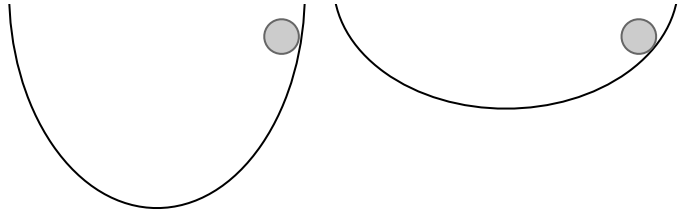


- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
  - mu = usually ~0.5, 0.9, or 0.99
  - (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

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

Momentum update

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



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

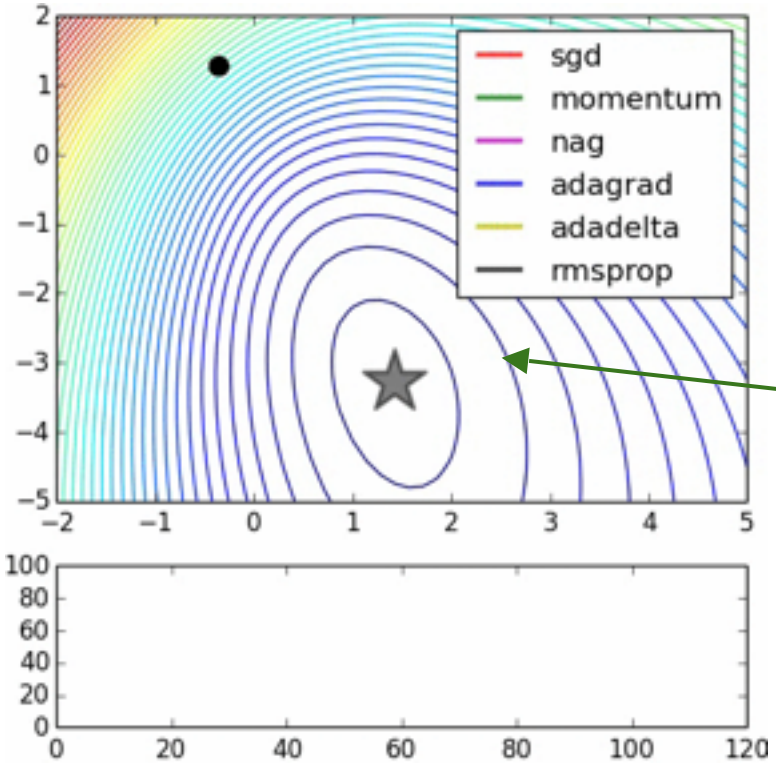
- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

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

SGD

vs

Momentum



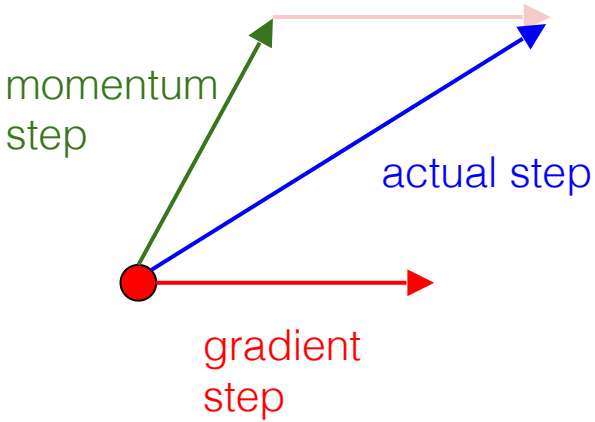
notice momentum overshooting the target, but overall getting to the minimum much faster.

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

Nesterov Momentum update

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

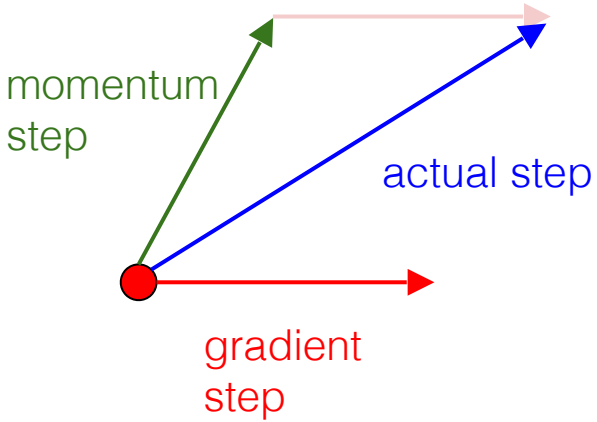
Ordinary momentum update:



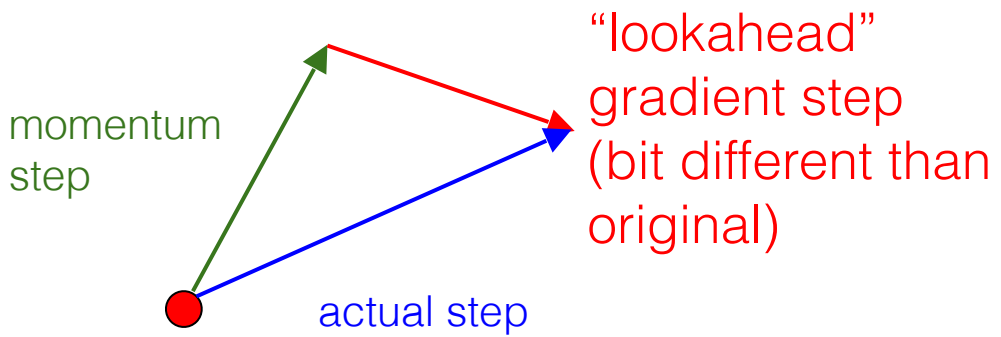
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

Nesterov Momentum update

Momentum update:



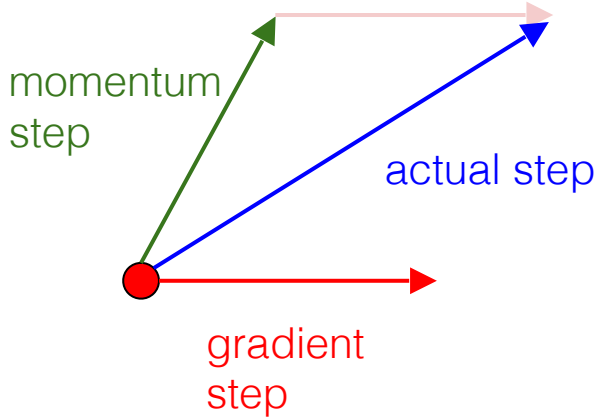
Nesterov momentum update



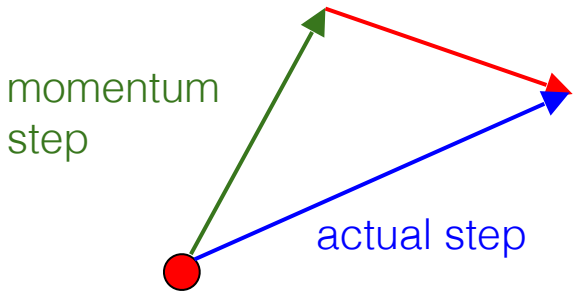
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

Nesterov Momentum update

Momentum update:



Nesterov momentum update



"lookahead" gradient step (bit different than original)

Nesterov: the only difference...

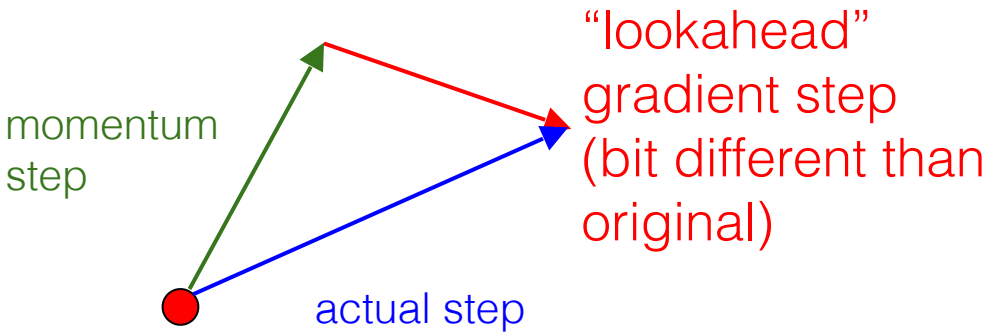
$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

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

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

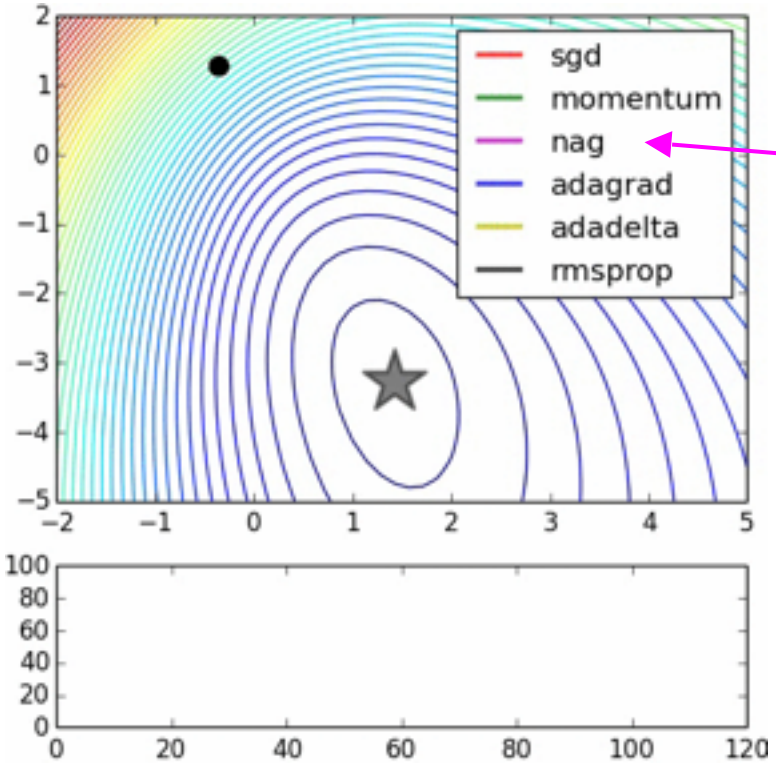
Nesterov Momentum update

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



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

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



nag = Nesterov Accelerated Gradient

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

AdaGrad update

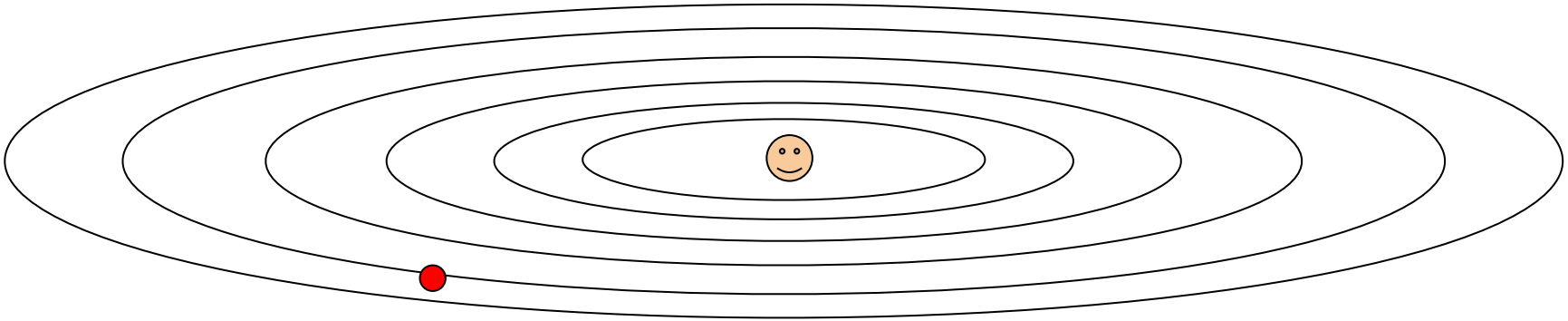
[Duchi et al., 2011]

```
# 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)
```

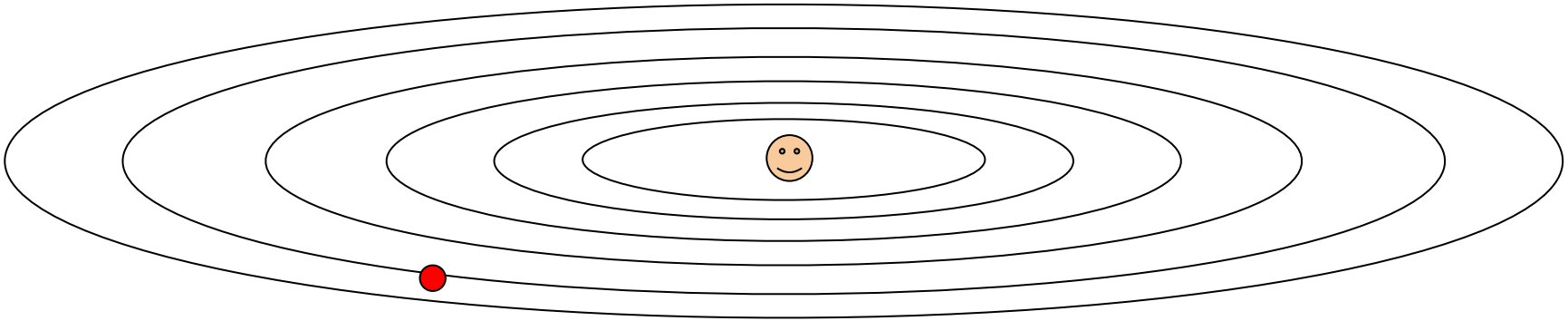


Q: What happens with AdaGrad?

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

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?

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

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

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\ t-1) + 0.1\left(\partial E / \partial w(t)\right)^2$$

- 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

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

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

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\ t-1) + 0.1\left(\partial E / \partial w(t)\right)^2$$

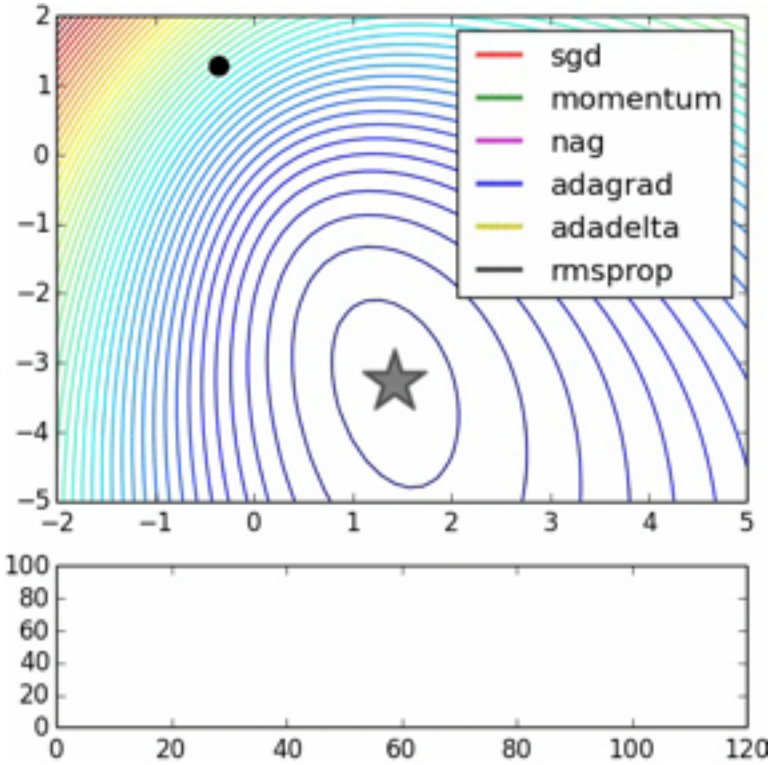
- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

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.

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



adagrad
rmsprop

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

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)
```

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

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)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

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

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)
```

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)
```

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

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

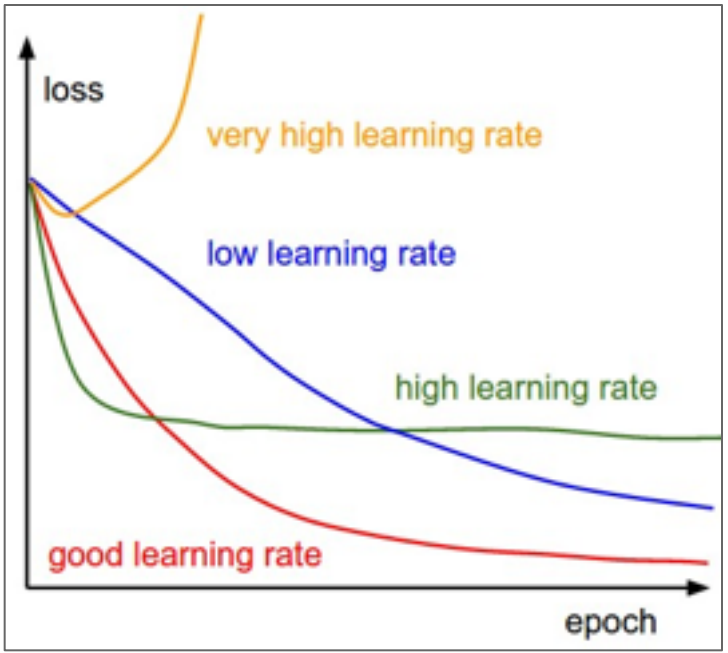
bias correction
(only relevant in first few iterations when t is small)

RMSProp-like

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

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

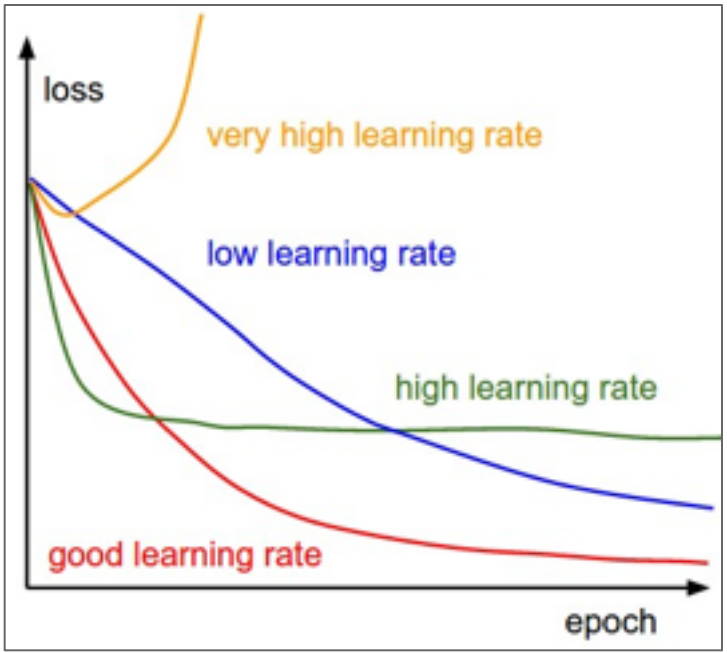
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



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

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

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

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

Second order optimization methods

second-order Taylor expansion:

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

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

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

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.

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

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.

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

Evaluation:

Model Ensembles

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

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.

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

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:

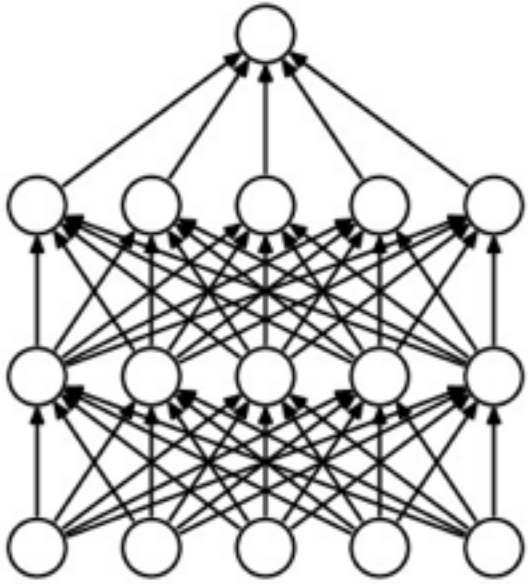
```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

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

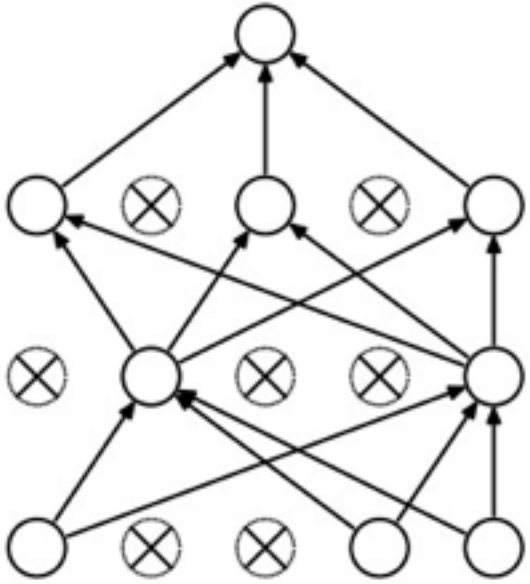
Regularization **(dropout)**

Regularization: **Dropout**

"randomly set some neurons to zero in the forward pass"



(a) Standard Neural Net



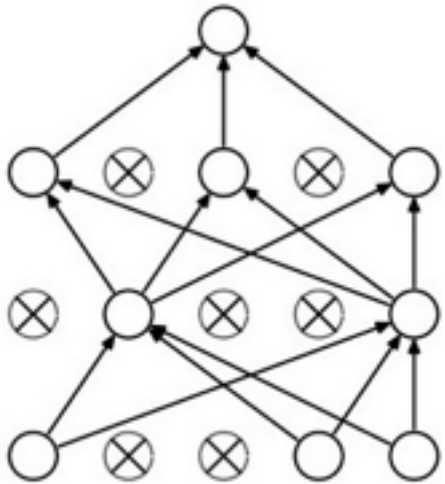
(b) After applying dropout.

[Srivastava et al., 2014]

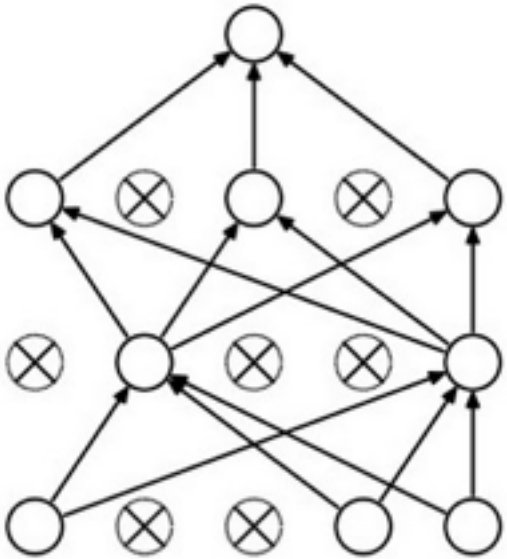
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

```python
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

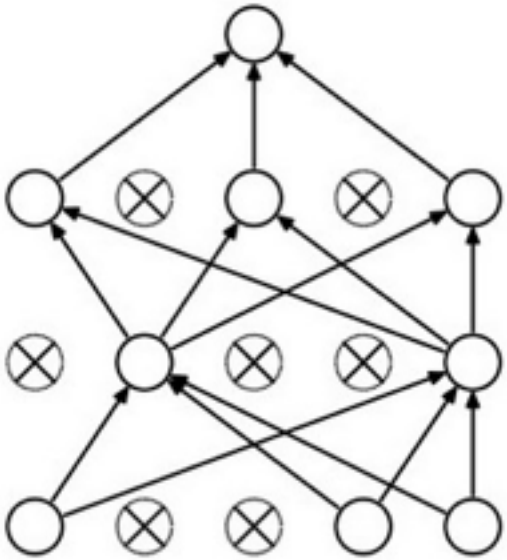


Waaaait a second…
How could this possibly be a good idea?



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

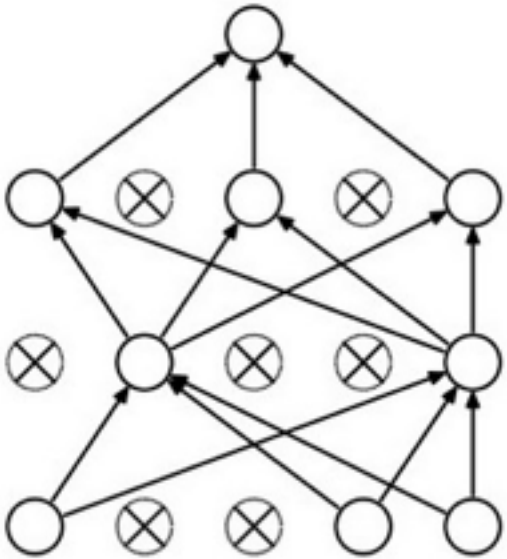
Waaaait a second…
How could this possibly be a good idea?



Forces the network to have a redundant representation.

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

At test time….



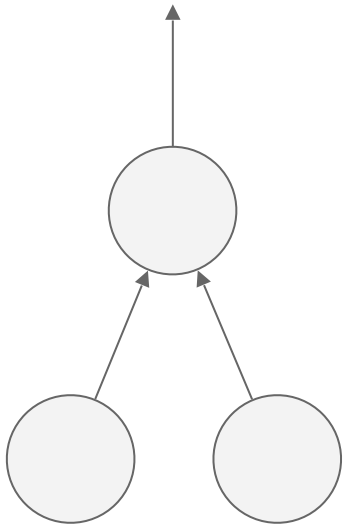
Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

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

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)

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

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

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

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

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

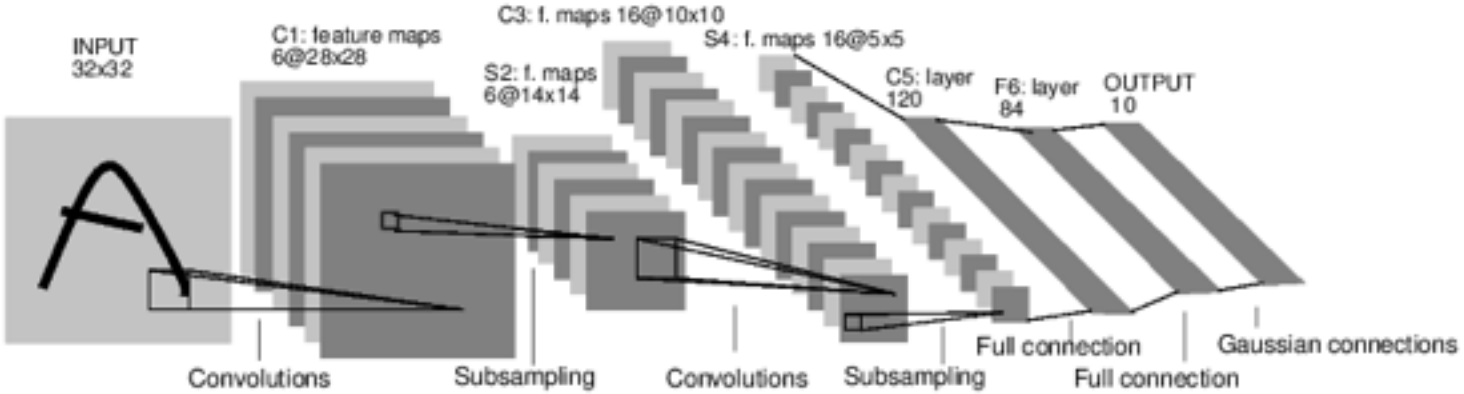
More common: “Inverted dropout”

```python
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!

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

Convolutional Neural Networks



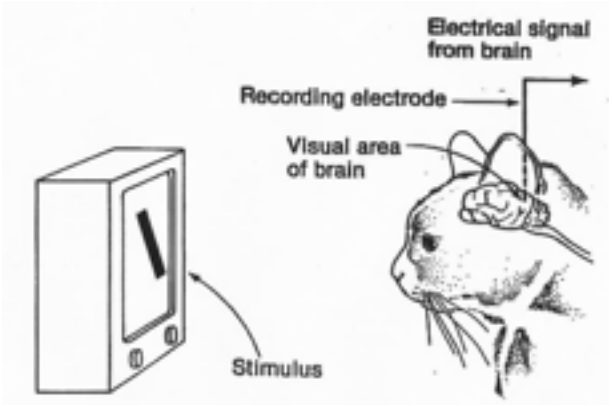
[LeNet-5, LeCun 1980]

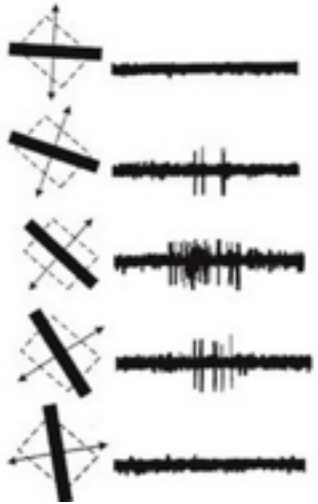
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

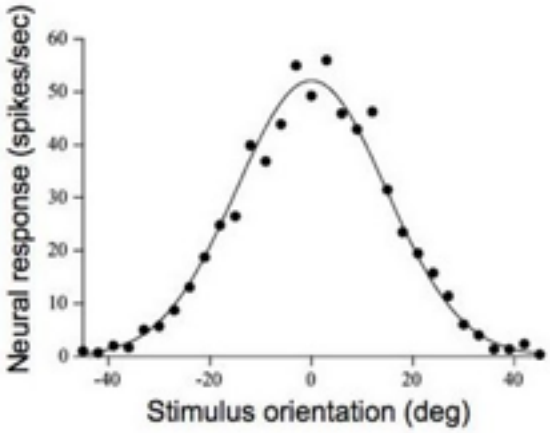
A bit of history:

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



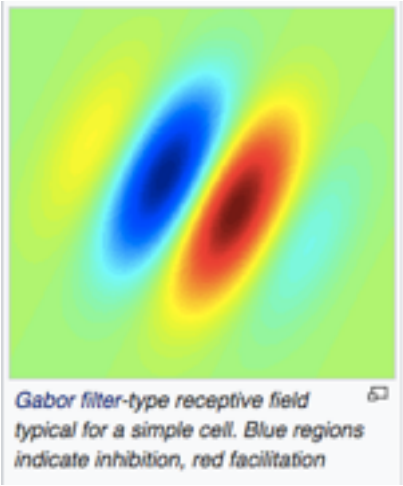




* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

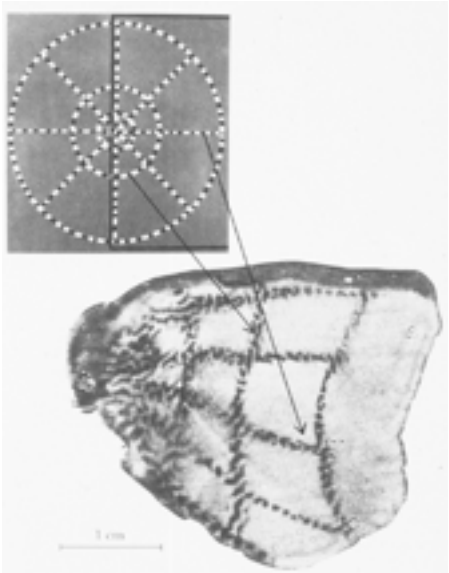
A bit of history

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

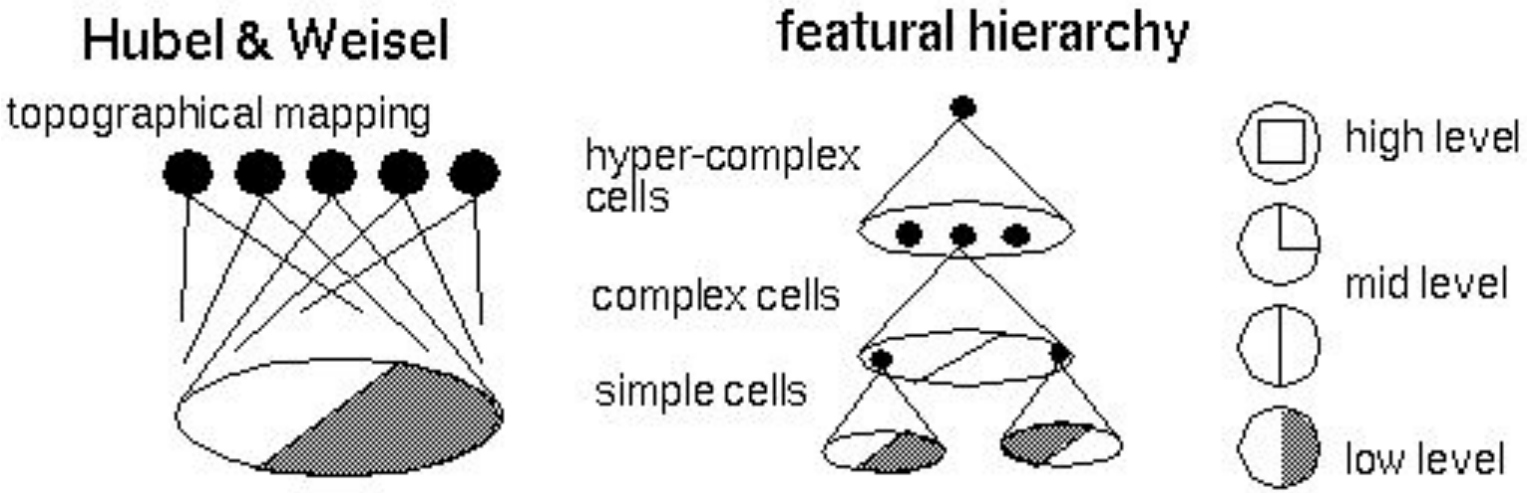


Gabor filter-type receptive field typical for a simple cell. Blue regions indicate inhibition, red facilitation

Topographical mapping in the cortex:
nearby cells in cortex represented nearby regions in the visual field



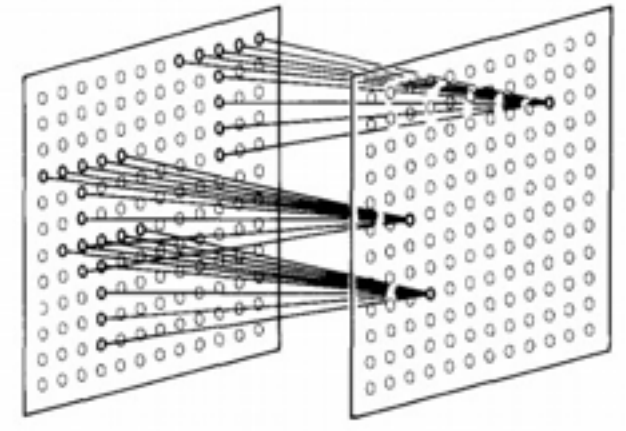
Hierarchical organization



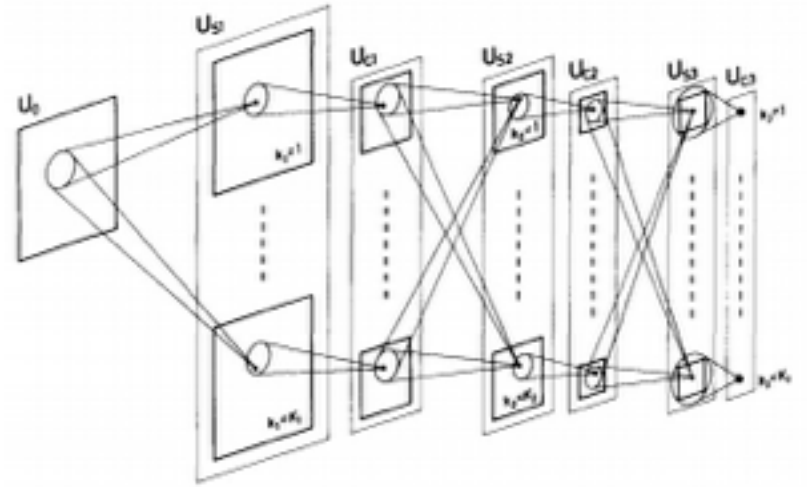
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

A bit of history:

Neurocognitron
[Fukushima 1980]



"sandwich" architecture (SCSCSC…)
simple cells: modifiable parameters
complex cells: perform pooling

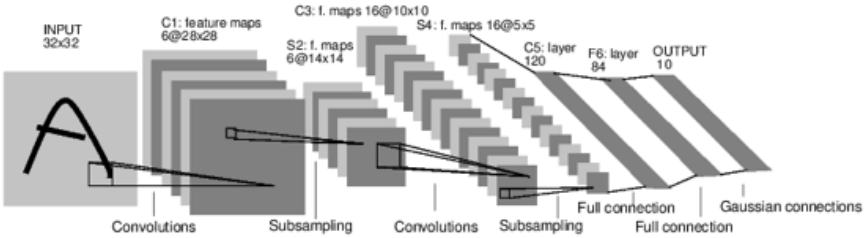


* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

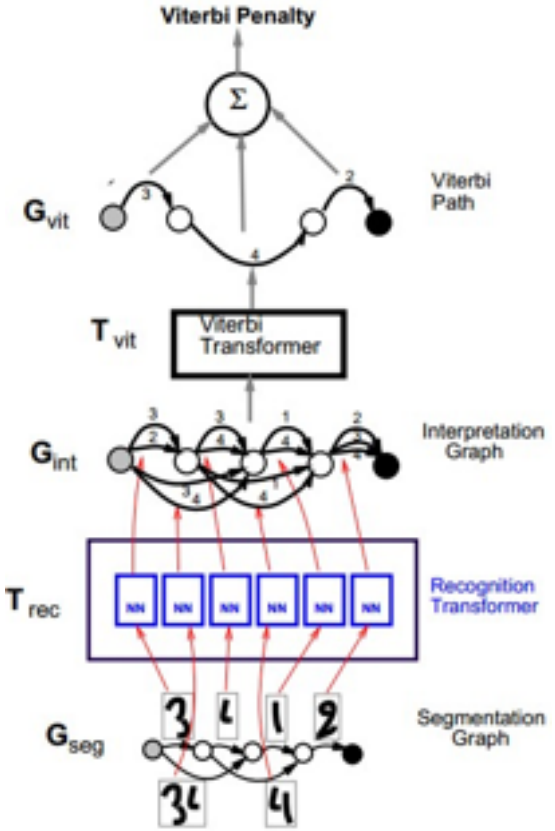
A bit of history:

Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



LeNet-5



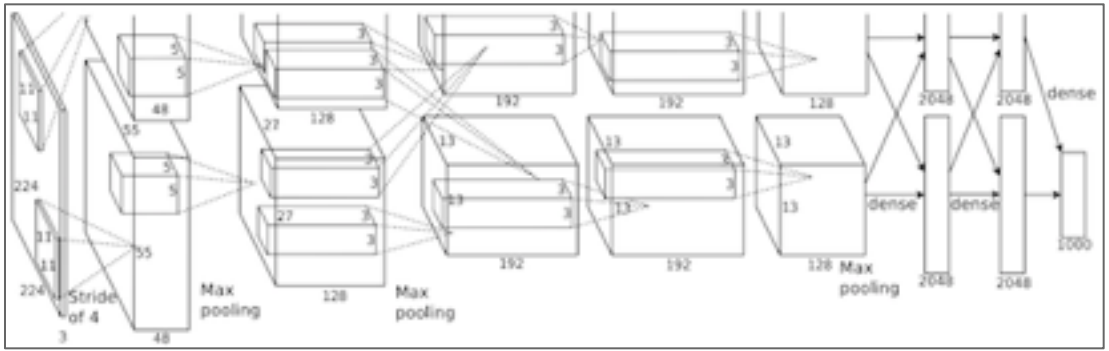
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

IMAGENET

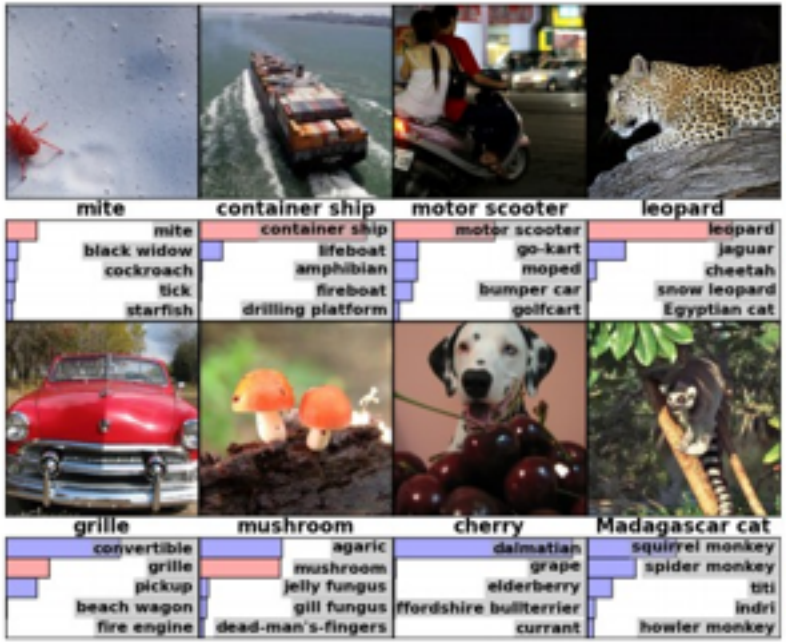


"AlexNet"

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

Fast-forward to today: ConvNets are everywhere

Classification



Retrieval

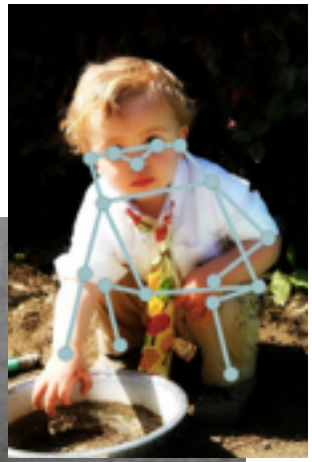
[Krizhevsky 2012]

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


traffic light
traffic light
traffic light
traffic light
person
truck
truck
car
car
frisbee


umbrella
person
car
car
car
car
car
car
person
skateboard


BOFFARD



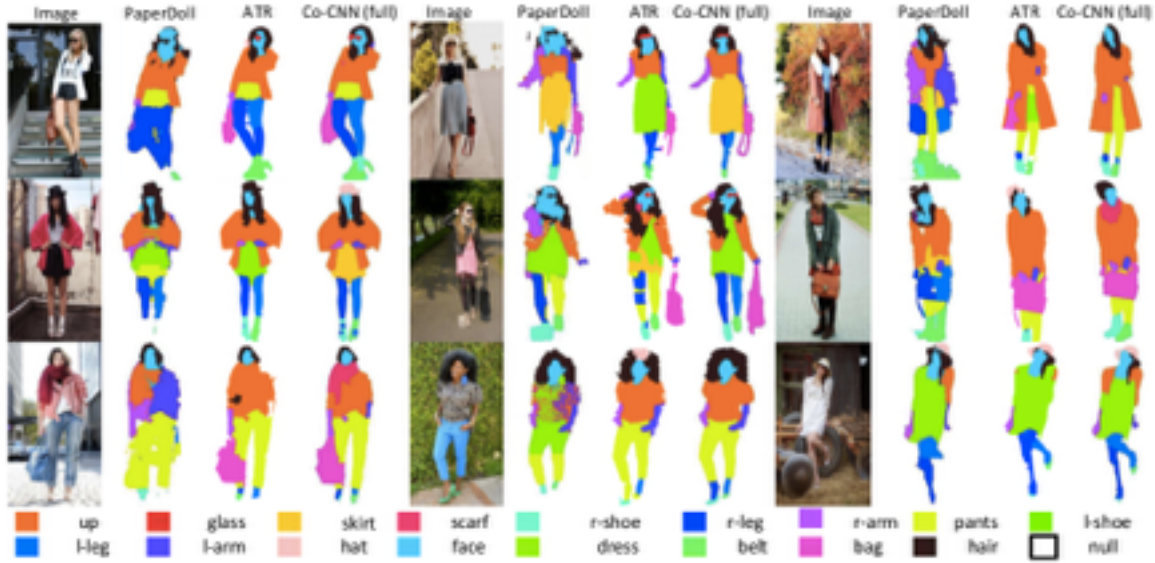
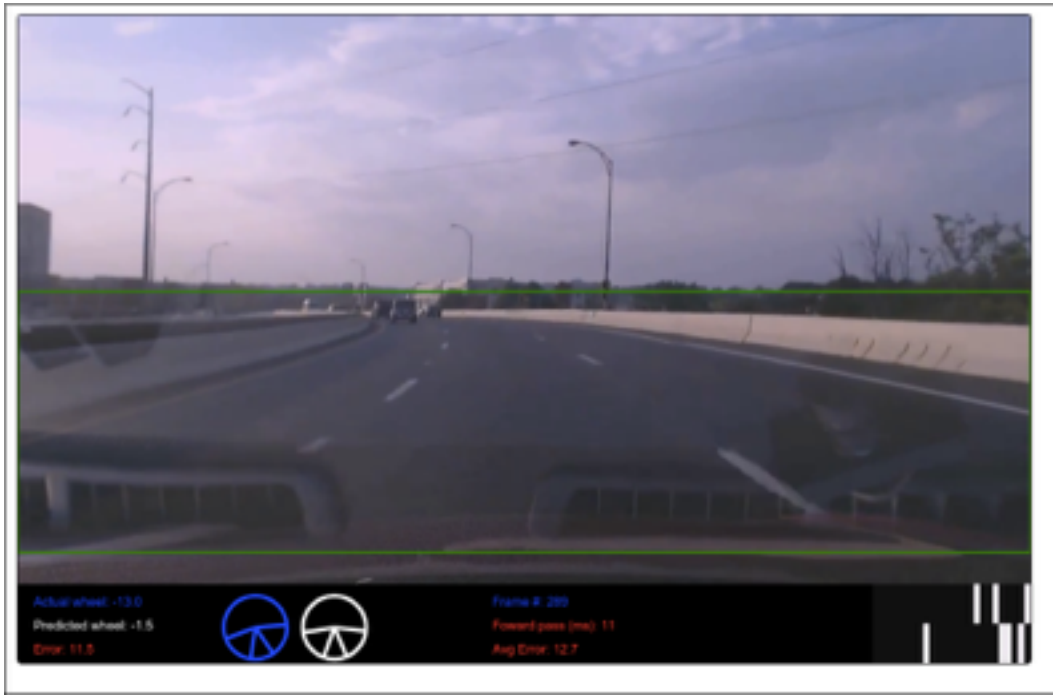
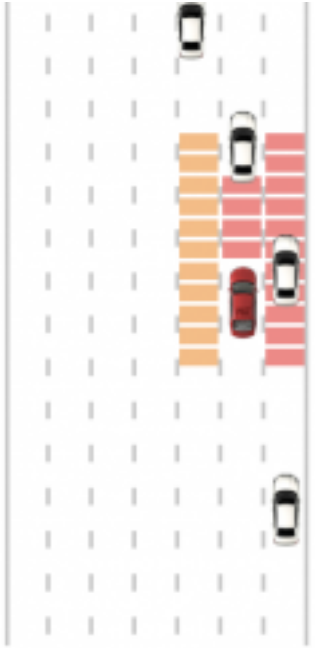


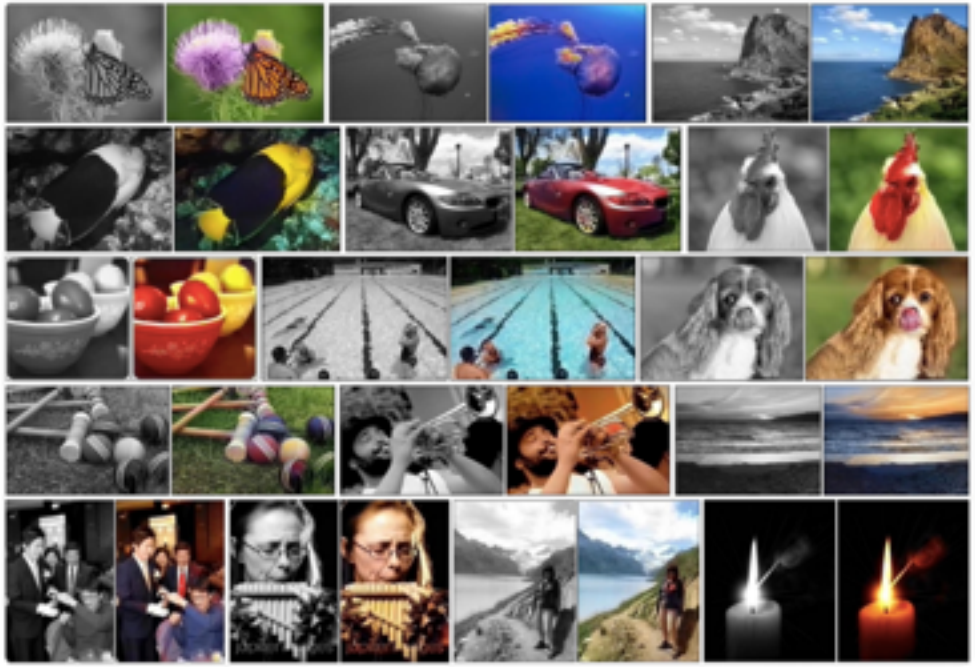
Image
PaperDoll
ATR
Co-CNN (full)
Image
PaperDoll
ATR
Co-CNN (full)
Image
PaperDoll
ATR
Co-CNN (full)
up
l-leg
glass
l-arm
skirt
hat
scarf
face
r-shoe
dress
r-leg
belt
r-arm
bag
pants
hair
l-shoe
null

Self-Driving Cars





Art



What is the color of the coat?

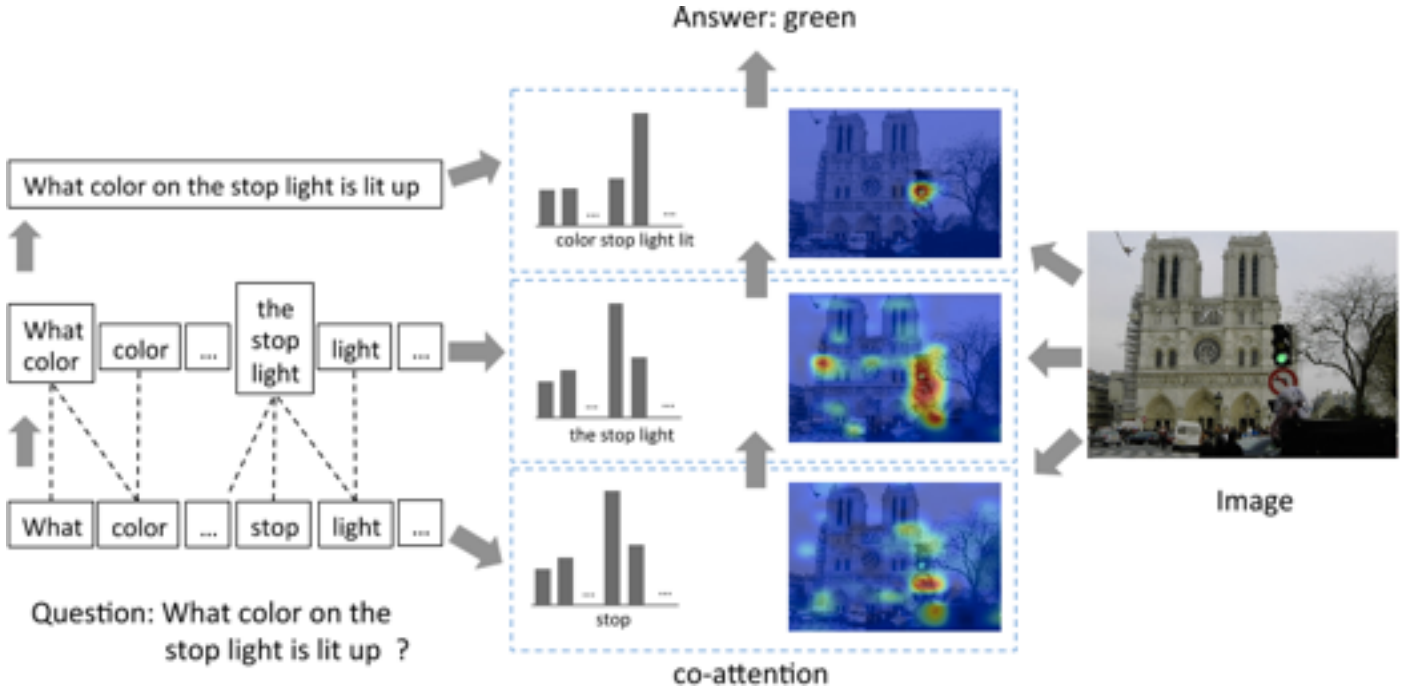
Traditional VQA: analyze the whole image -> analyze question -> give answer: brown

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

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

Caffe Overview

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

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

Most important tip...

Don’t be afraid to read the code!

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

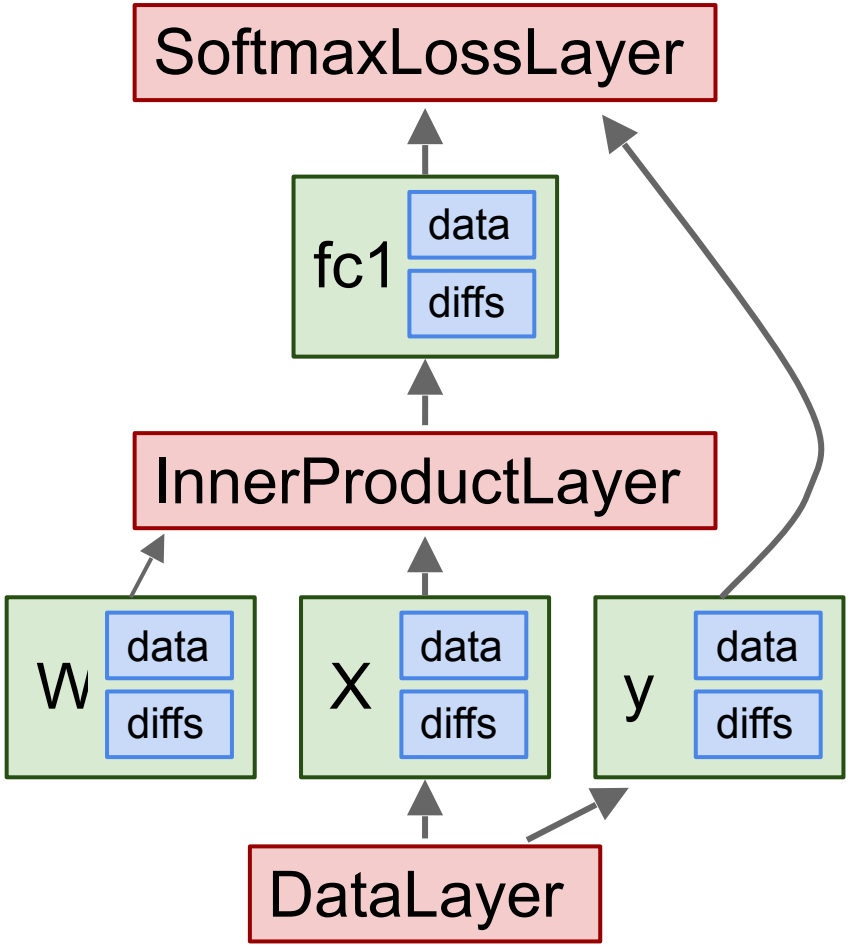
Caffe: Main classes

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)



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

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/

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

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/

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

Caffe: Protocol Buffers

```
message NetParameter {
  optional string name = 1; // consider giving the network a name
  // The input blobs to the network.
  repeated string input = 3;
  // The shape of the input blobs.
  repeated BlobShape input_shape = 8;

  // 4D input dimensions -- deprecated.  Use "shape" instead.
  // If specified, for each input blob there should be four
  // values specifying the num, channels, height and width of the input blob.
  // Thus, there should be a total of (4 * #input) numbers.
  repeated int32 input_dim = 4;

  // Whether the network will force every layer to carry out backward operation.
  // If set False, then whether to carry out backward is determined
  // automatically according to the net structure and learning rates.
  optional bool force_backward = 5 [default = false];
  // The current "state" of the network, including the phase, level, and stage.
  // Some layers may be included/excluded depending on this state and the states
  // specified in the layers' include and exclude fields.
  optional NetState state = 6;

  // Print debugging information about results while running Net::Forward,
  // Net::Backward, and Net::Update.
  optional bool debug_info = 7 [default = false];
```

```
message SolverParameter {
  //////////////////////////////////////////////////////////////////////////////
  // Specifying the train and test networks
  //
  // Exactly one train net must be specified using one of the following fields:
  //     train_net_param, train_net, net_param, net
  // One or more test nets may be specified using any of the following fields:
  //     test_net_param, test_net, net_param, net
  // If more than one test net field is specified (e.g., both net and
  // test_net are specified), they will be evaluated in the field order given
  // above: (1) test_net_param, (2) test_net, (3) net_param/net.
  // A test_iter must be specified for each test_net.
  // A test_level and/or a test_stage may also be specified for each test_net.
  //////////////////////////////////////////////////////////////////////////////

  // Proto filename for the train net, possibly combined with one or more
  // test nets.
  optional string net = 24;
  // Inline train net param, possibly combined with one or more test nets.
  optional NetParameter net_param = 25;

  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!

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

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)

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

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)

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

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
}
```

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

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
}
```

Layers and Blobs often have same name!

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

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
}
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)

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

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
}
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes

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

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
}
```

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

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

```
name: "ResNet-152"
input: "data"
input_dim: 1
input_dim: 3
input_dim: 224
input_dim: 224

layer {
        bottom: "data"
        top: "conv1"
        name: "conv1"
        type: "Convolution"
        convolution_param {
                num_output: 64
                kernel_size: 7
                pad: 3
                stride: 2
                bias_term: false
        }
}

layer {
        bottom: "conv1"
        top: "conv1"
        name: "bn_conv1"
        type: "BatchNorm"
        batch_norm_param {
                use_global_stats: true
        }
}
```

```
layer {
        bottom: "res5c"
        top: "pool5"
        name: "pool5"
        type: "Pooling"
        pooling_param {
                kernel_size: 7
                stride: 1
                pool: AVE
        }
}

layer {
        bottom: "pool5"
        top: "fc1000"
        name: "fc1000"
        type: "InnerProduct"
        inner_product_param {
                num_output: 1000
        }
}

layer {
        bottom: "fc1000"
        top: "prob"
        name: "prob"
        type: "Softmax"
}
```

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

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

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
  }
}
```

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

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
  }
}
```

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

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
  }
}
```

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

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

```
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

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

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

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

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

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

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
```

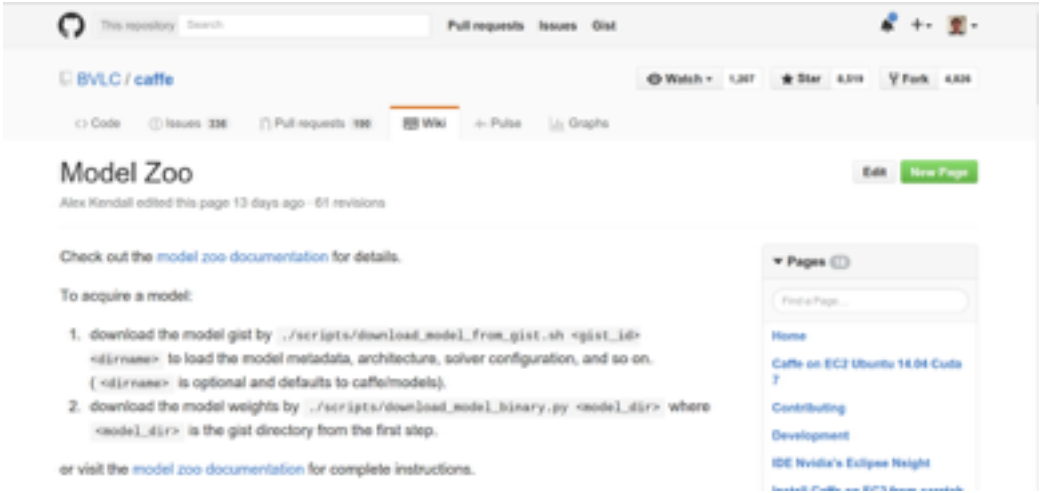
`-gpu all` for multi-GPU data parallelism

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

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

Caffe: Model Zoo

- AlexNet, VGG, GoogLeNet, ResNet, plus others



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

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

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

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

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)

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

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)

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

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);
```

```
219   const Dtype* cpu_data() const;
220   void set_cpu_data(Dtype* data);
221   const int* gpu_shape() const;
222   const Dtype* gpu_data() const;
223   const Dtype* cpu_diff() const;
224   const Dtype* gpu_diff() const;
225   Dtype* mutable_cpu_data();
226   Dtype* mutable_gpu_data();
227   Dtype* mutable_cpu_diff();
228   Dtype* mutable_gpu_diff();
```

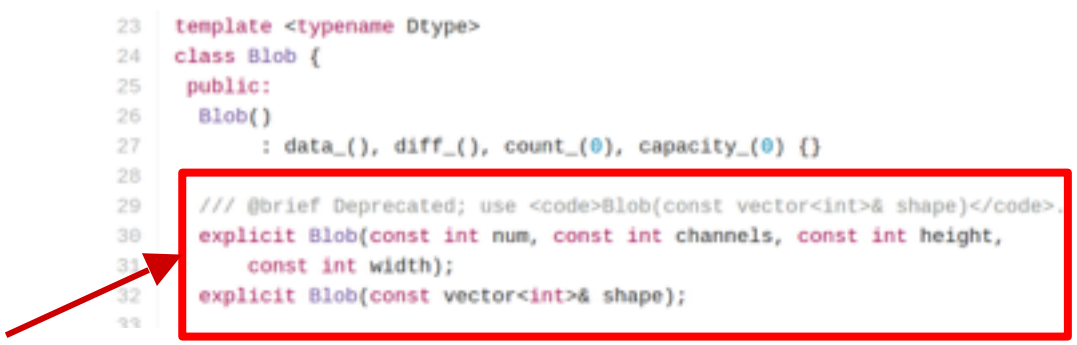
```
268  protected:
269   shared_ptr<SyncedMemory> data_;
270   shared_ptr<SyncedMemory> diff_;
271   shared_ptr<SyncedMemory> shape_data_;
272   vector<int> shape_;
273   int count_;
274   int capacity_;
```

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

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

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);
219   const Dtype* cpu_data() const;
220   void set_cpu_data(Dtype* data);
221   const int* gpu_shape() const;
222   const Dtype* gpu_data() const;
223   const Dtype* cpu_diff() const;
224   const Dtype* gpu_diff() const;
225   Dtype* mutable_cpu_data();
226   Dtype* mutable_gpu_data();
227   Dtype* mutable_cpu_diff();
228   Dtype* mutable_gpu_diff();
268  protected:
269   shared_ptr<SyncedMemory> data_;
270   shared_ptr<SyncedMemory> diff_;
271   shared_ptr<SyncedMemory> shape_data_;
272   vector<int> shape_;
273   int count_;
274   int capacity_;
```

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

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

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);

219   const Dtype* cpu_data() const;
220   void set_cpu_data(Dtype* data);
221   const int* gpu_shape() const;
222   const Dtype* gpu_data() const;
223   const Dtype* cpu_diff() const;
224   const Dtype* gpu_diff() const;
225   Dtype* mutable_cpu_data();
226   Dtype* mutable_gpu_data();
227   Dtype* mutable_cpu_diff();
228   Dtype* mutable_gpu_diff();

268  protected:
269   shared_ptr<SyncedMemory> data_;
270   shared_ptr<SyncedMemory> diff_;
271   shared_ptr<SyncedMemory> shape_data_;
272   vector<int> shape_;
273   int count_;
274   int capacity_;
```

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

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

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);

219   const Dtype* cpu_data() const;
220   void set_cpu_data(Dtype* data);
221   const int* gpu_shape() const;
222   const Dtype* gpu_data() const;
223   const Dtype* cpu_diff() const;
224   const Dtype* gpu_diff() const;
225   Dtype* mutable_cpu_data();
226   Dtype* mutable_gpu_data();
227   Dtype* mutable_cpu_diff();
228   Dtype* mutable_gpu_diff();

268   protected:
269    shared_ptr<SyncedMemory> data_;
270    shared_ptr<SyncedMemory> diff_;
271    shared_ptr<SyncedMemory> shape_data_;
272    vector<int> shape_;
273    int count_;
274    int capacity_;
```

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

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

Caffe: Layer

A small unit of computation

```
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

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

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

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

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:

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

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

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

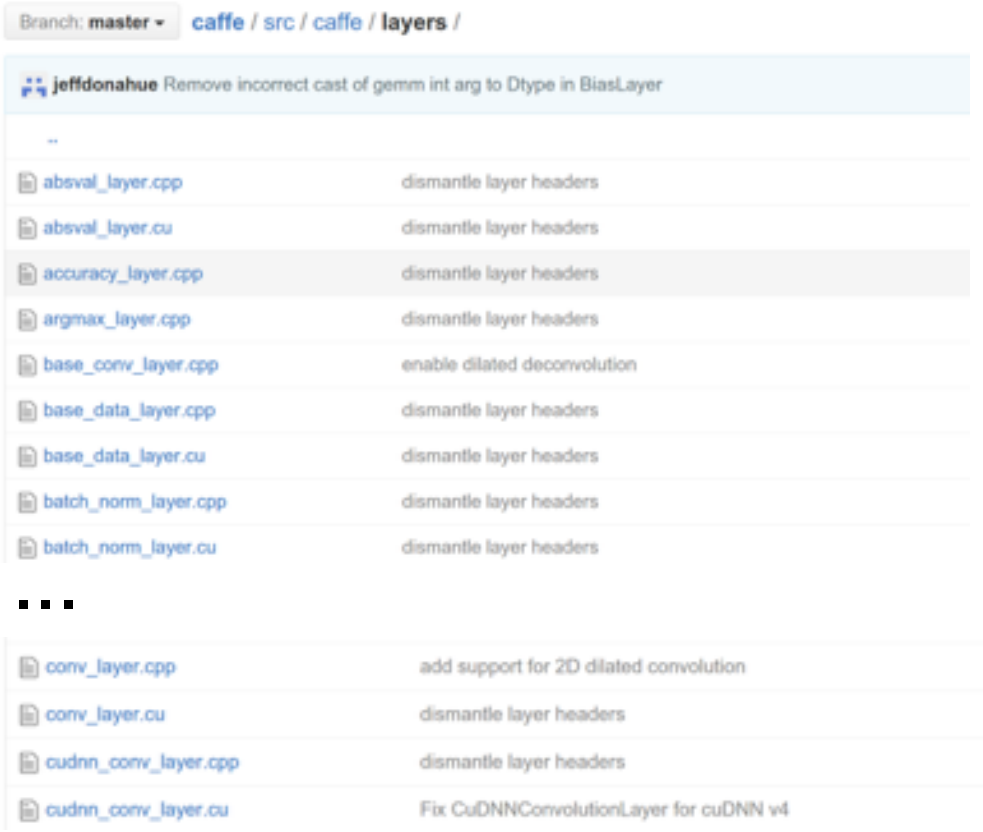
```
32  template <typename Dtype>
33  class Layer {
34   public:
334   /** @brief Using the CPU device, compute the layer output. */
335   virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
336       const vector<Blob<Dtype>*>& top) = 0;
337   /**
338    * @brief Using the GPU device, compute 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

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

Caffe: Layer

Tons of different layer types:



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

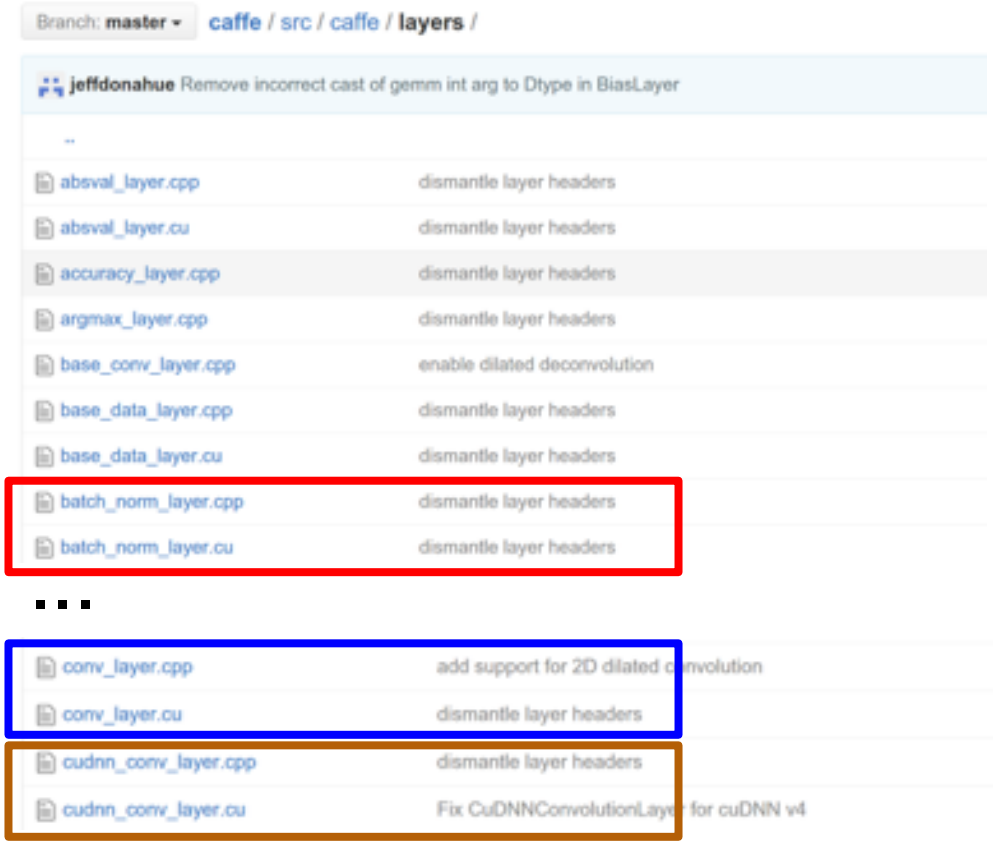
* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

Caffe: Layer

Tons of different layer types:

- **batch norm**
- **convolution**
- **cuDNN convolution**

.cpp: CPU implementation
.cu: GPU implementation



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

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

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

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

Caffe: Solver

```
40  template <typename Dtype>
41  class Solver {
42   public:
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

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

Caffe: Solver

```
40  template <typename Dtype>
41  class Solver {
42   public:
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();
```

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

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

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

Caffe: Solver

```
40  template <typename Dtype>
41  class Solver {
42   public:
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();
```

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

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

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:
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> {

83  template <typename Dtype>
84  class RMSPropSolver : public SGDSolver<Dtype> {

130 template <typename Dtype>
131 class AdamSolver : public SGDSolver<Dtype> {
```

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

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

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)

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