Lecture 06: Training Neural Networks, Part 2

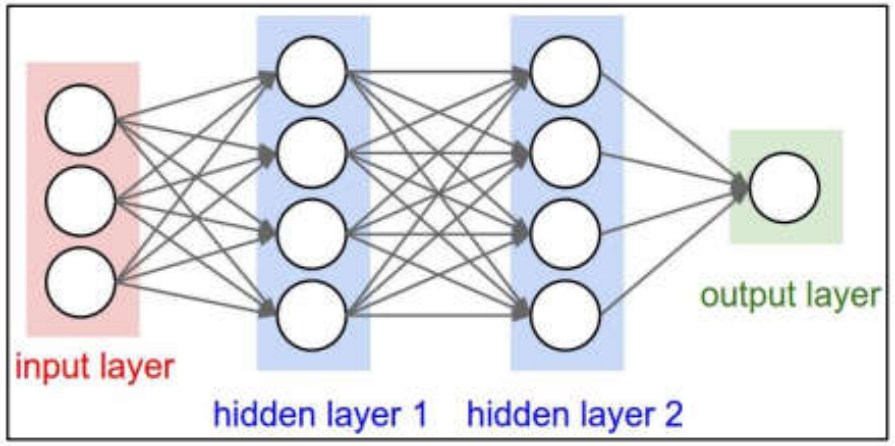
박사과정 김성빈 chengbinjin@inha.edu,
지도교수 김학일 교수 hikim@inha.ac.kr
인하대학교 컴퓨터비전 연구실

Recall From the Last Class

Mini-batch SGD

Loop:

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

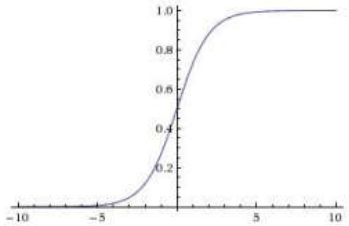


Slide from CS231n

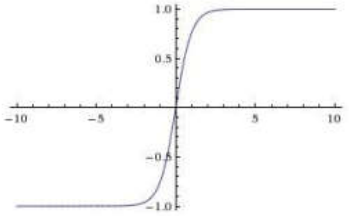
Recall From the Last Class

Sigmoid

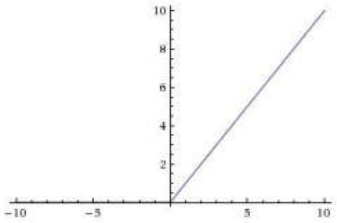
$$\sigma(x) = 1/(1+e^{-x})$$



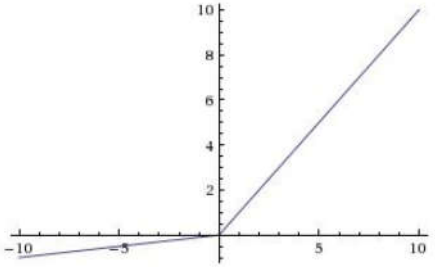
tanh tanh(x)



ReLU max(0,x)



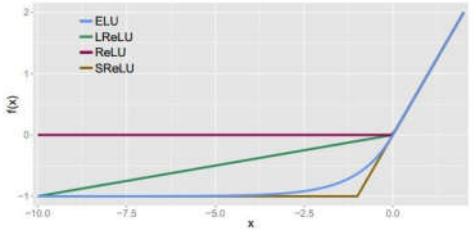
Leaky ReLU
max(0.1x, x)



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

ELU

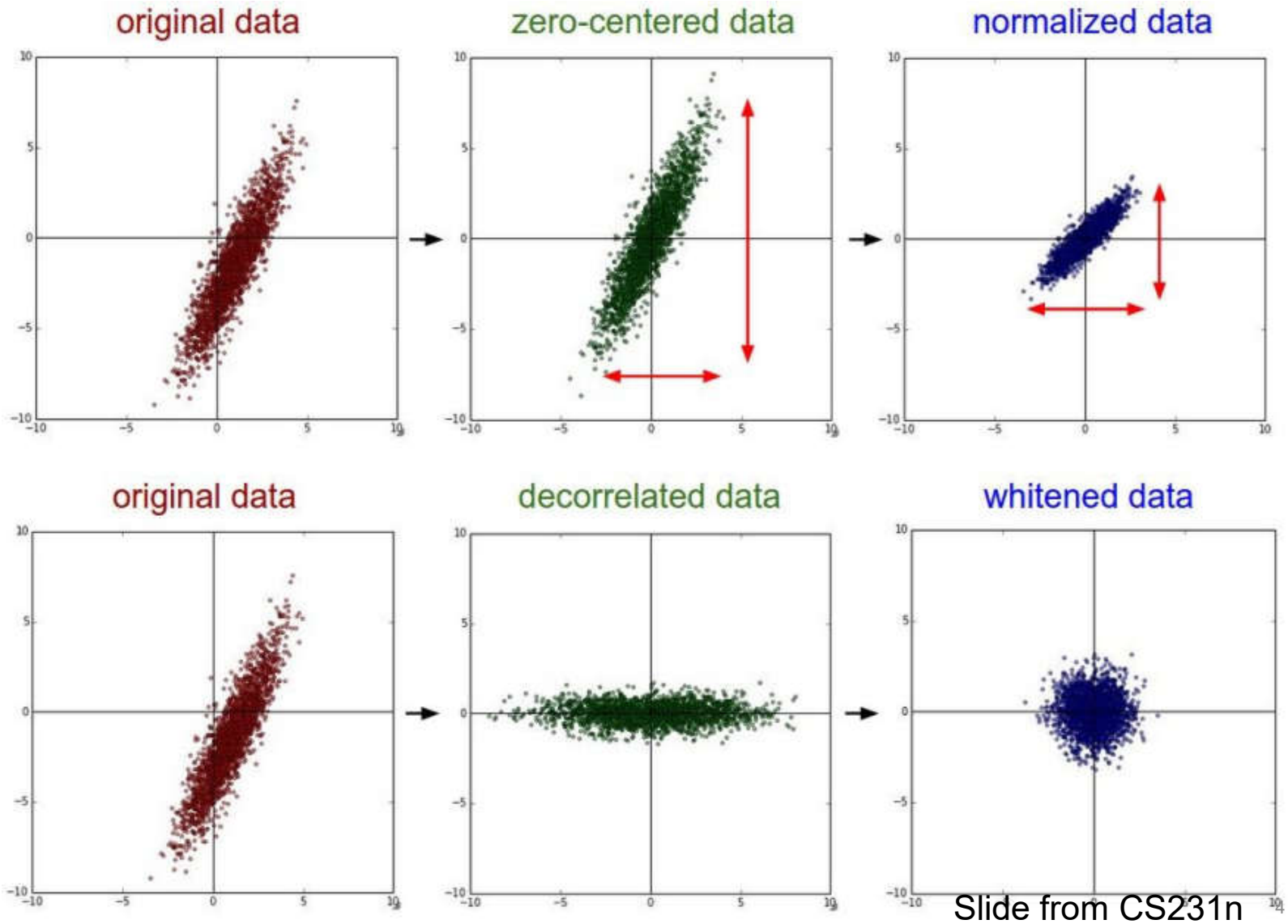
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



Slide from CS231n

Recall From the Last Class

Data Preprocessing



Slide from CS231n

Recall From the Last Class

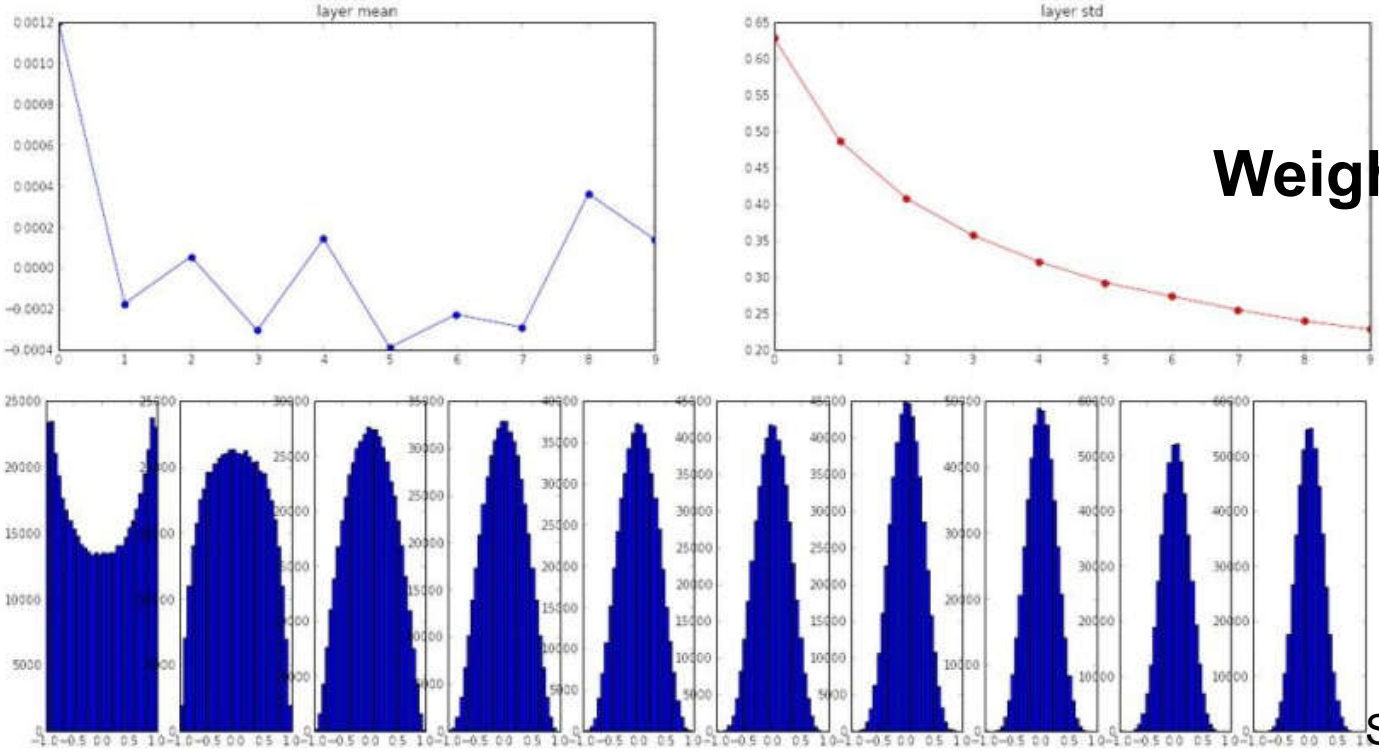
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

```
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
```

"Xavier initialization"

[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes **linear activations**)



Weight Initialization

Slide from CS231n

Recall From the Last Class

Batch Normalization

Normalize:

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

Part I

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

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

Part II

- 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…

[Ioffe and Szegedy, 20015]

Recall From the Last Class

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)
```

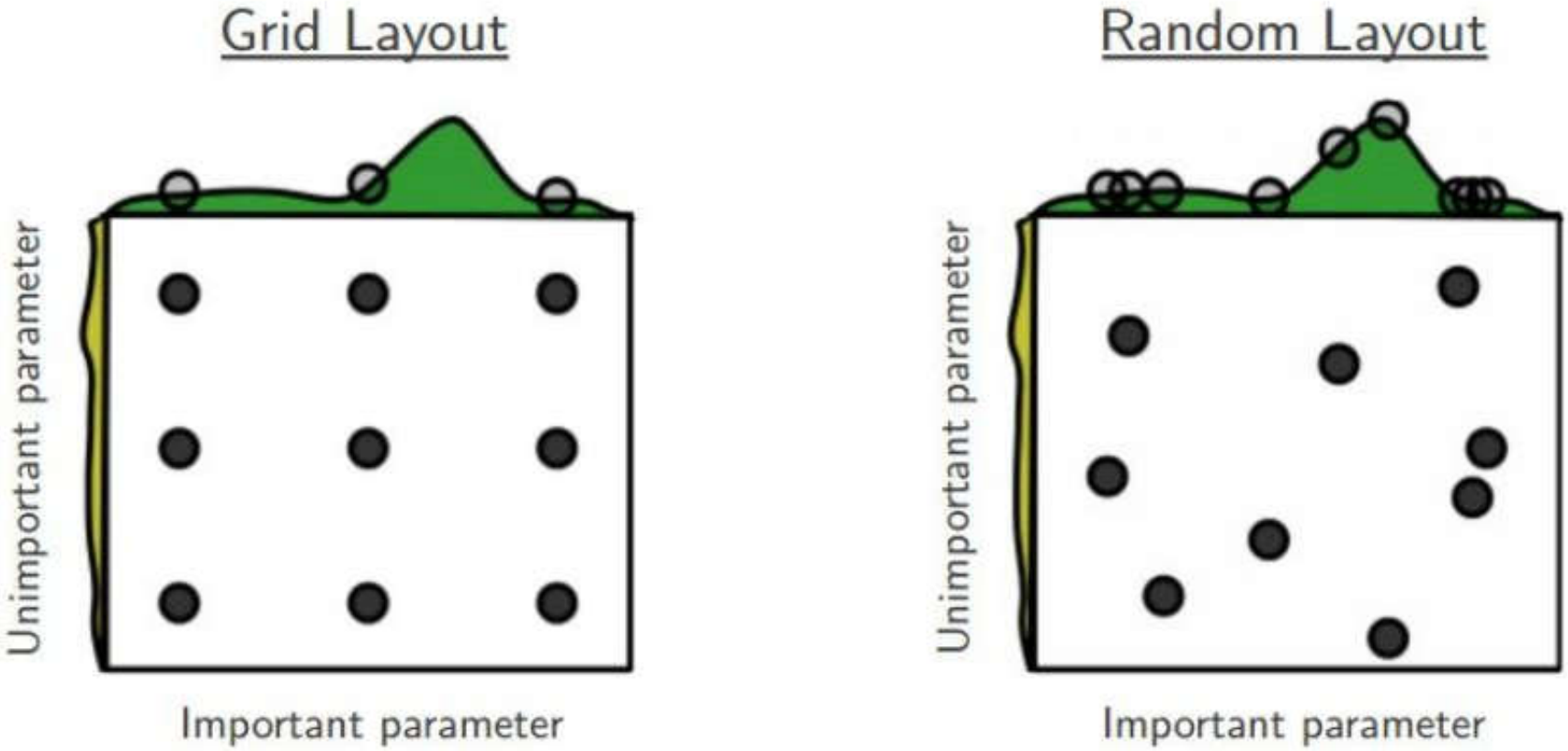
```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 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

Slide from CS231n

Recall From the Last Class

Cross-validation



Slide from CS231n

TODO

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

Parameter Updates

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
```

Slide from CS231n

Parameter Updates

- Training a neural network, main loop

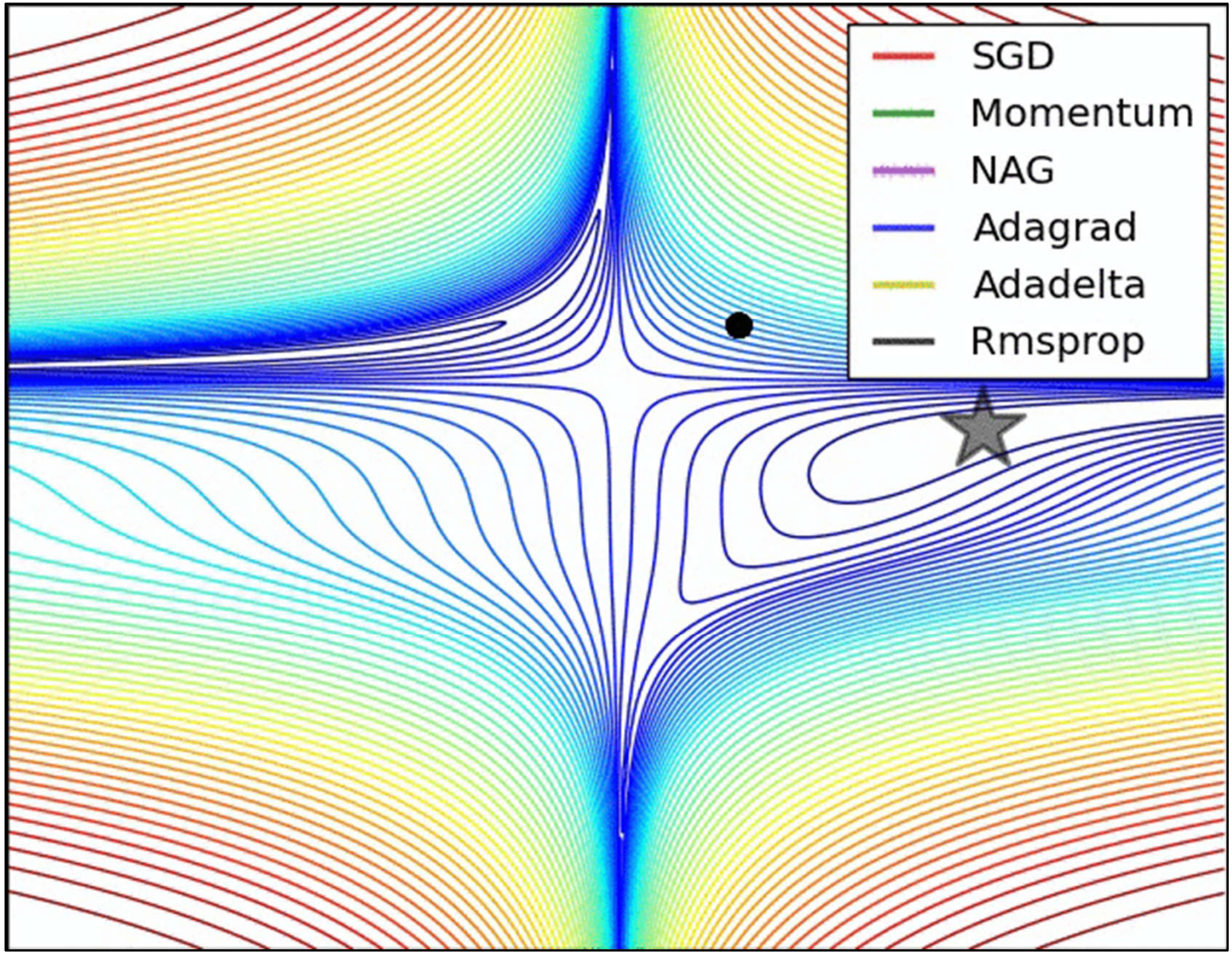
```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

SGD

Simple gradient descent update

Now: complicate.

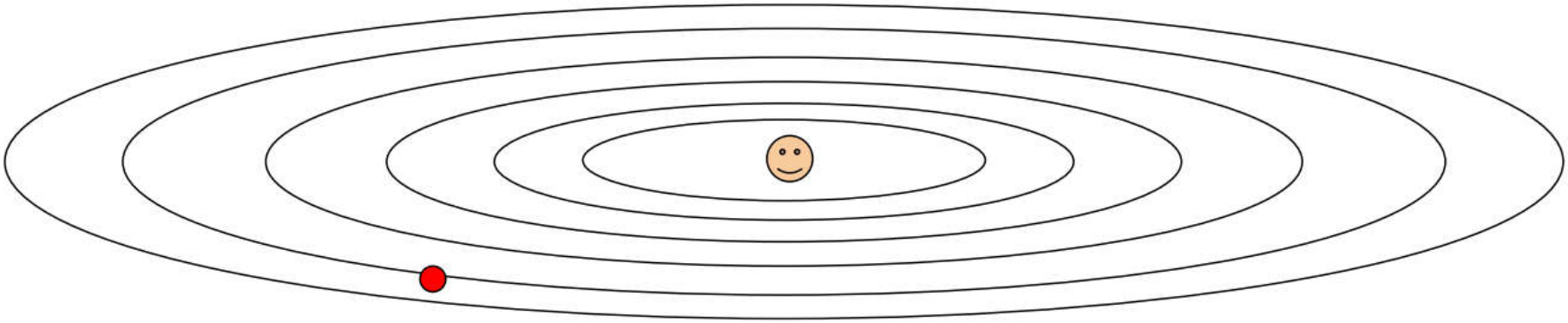
Parameter Updates



Slide from CS231n

SGD

- Suppose loss function is steep vertically but shallow horizontally:

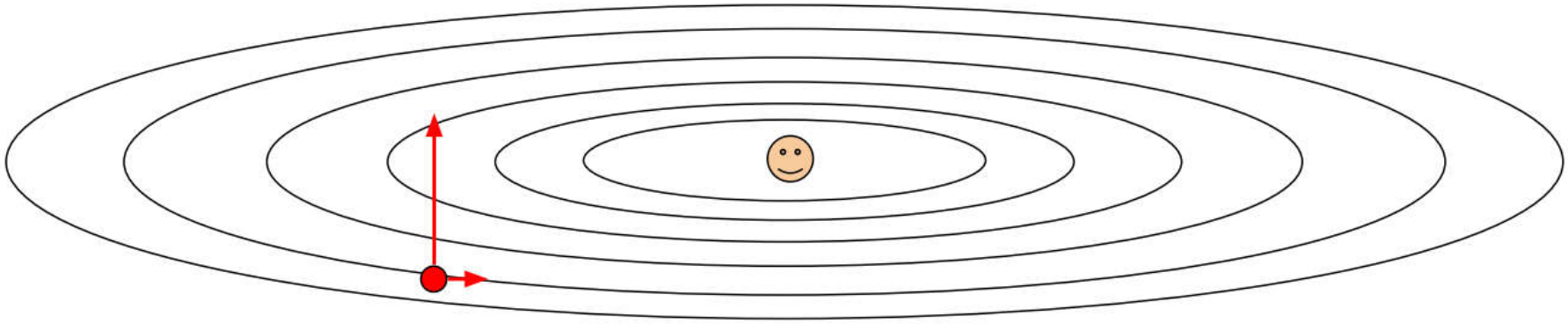


Question: what is the trajectory along which we converge toward the minimum with SGD?

Slide from CS231n

SGD

- Suppose loss function is steep vertically but shallow horizontally:

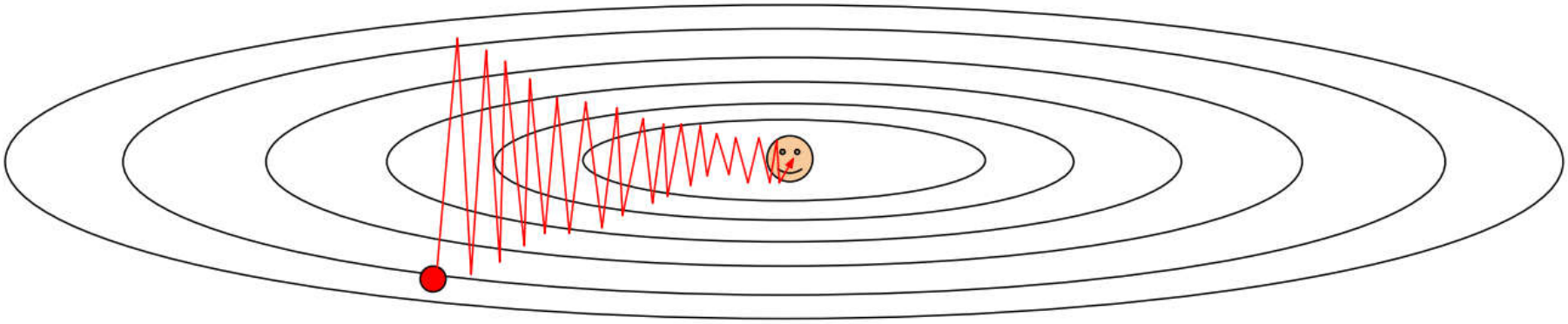


Question: what is the trajectory along which we converge toward the minimum with SGD?

Slide from CS231n

SGD

- Suppose loss function is steep vertically but shallow horizontally:

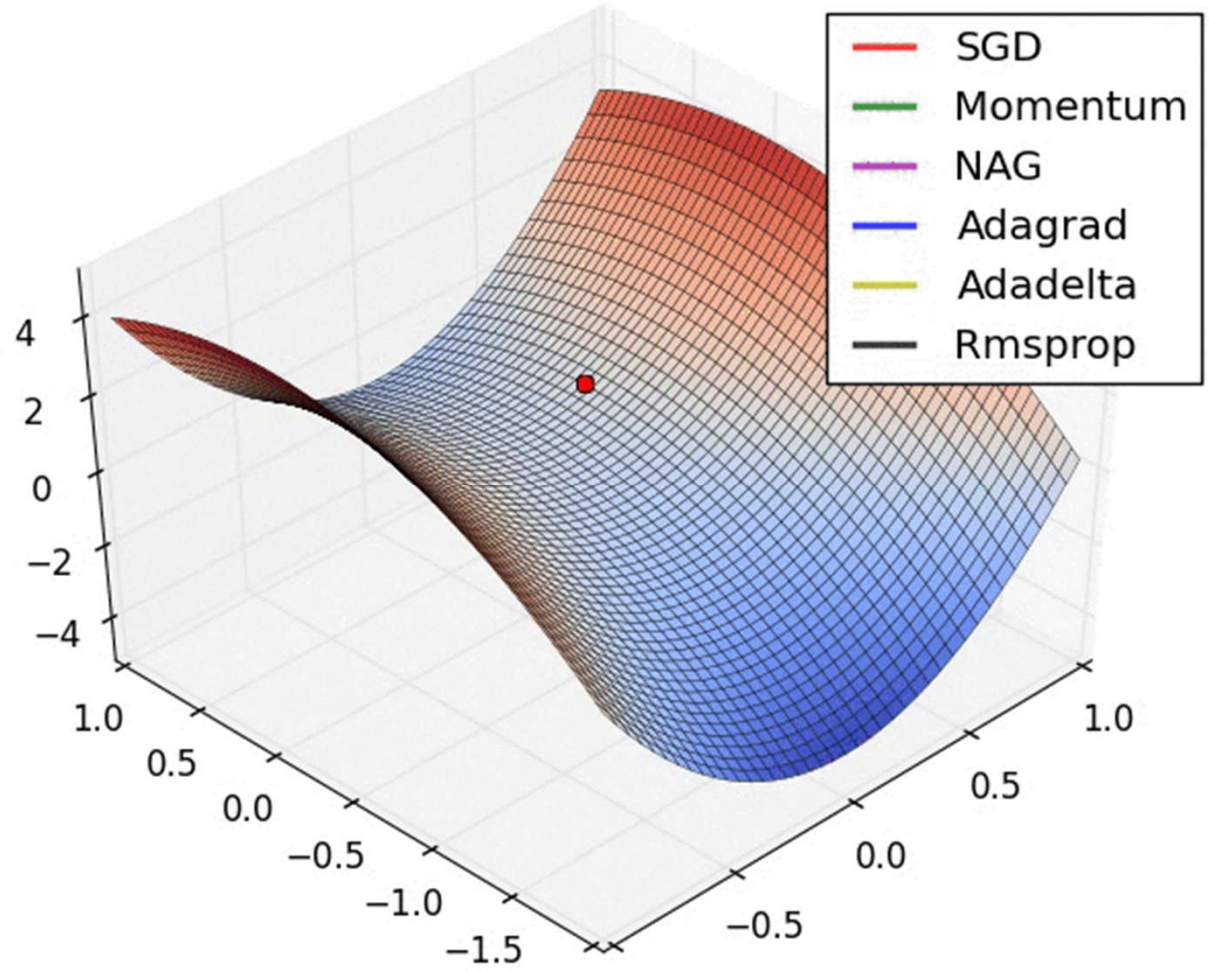


Question: what is the trajectory along which we converge toward the minimum with SGD?

- Very slow progress along flat direction, jitter along steep one.

SGD

- Suppose loss function is steep vertically but shallow horizontally:



Slide from CS231n

Momentum Update

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



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

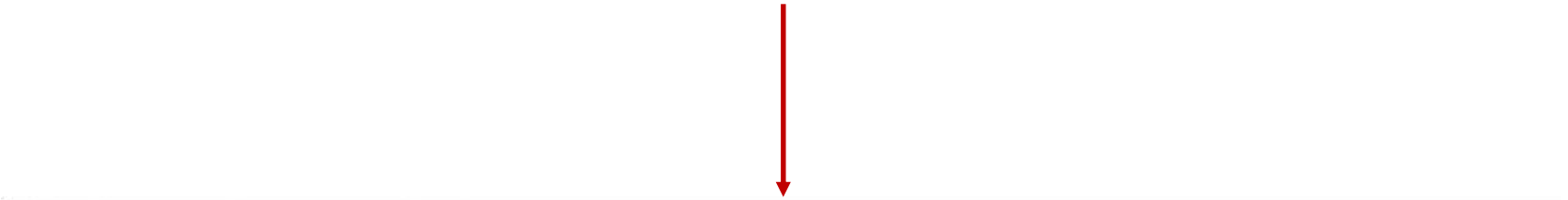
v: velocity
mu: friction
learning_rate * dx: accelerated value (force)

- Physical interpretation: 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)

Slide from CS231n

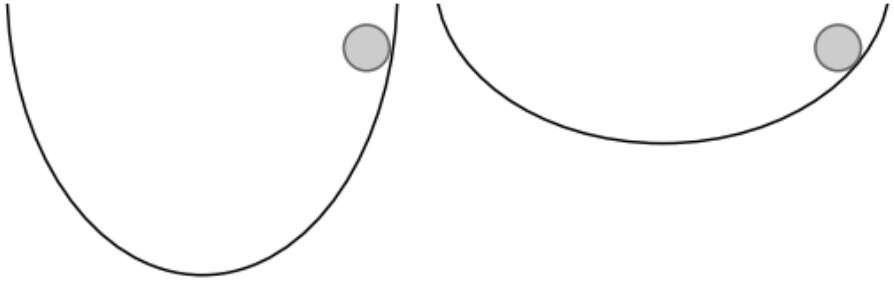
Momentum Update

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



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

v: velocity
mu: friction
learning_rate * dx: accelerated value



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

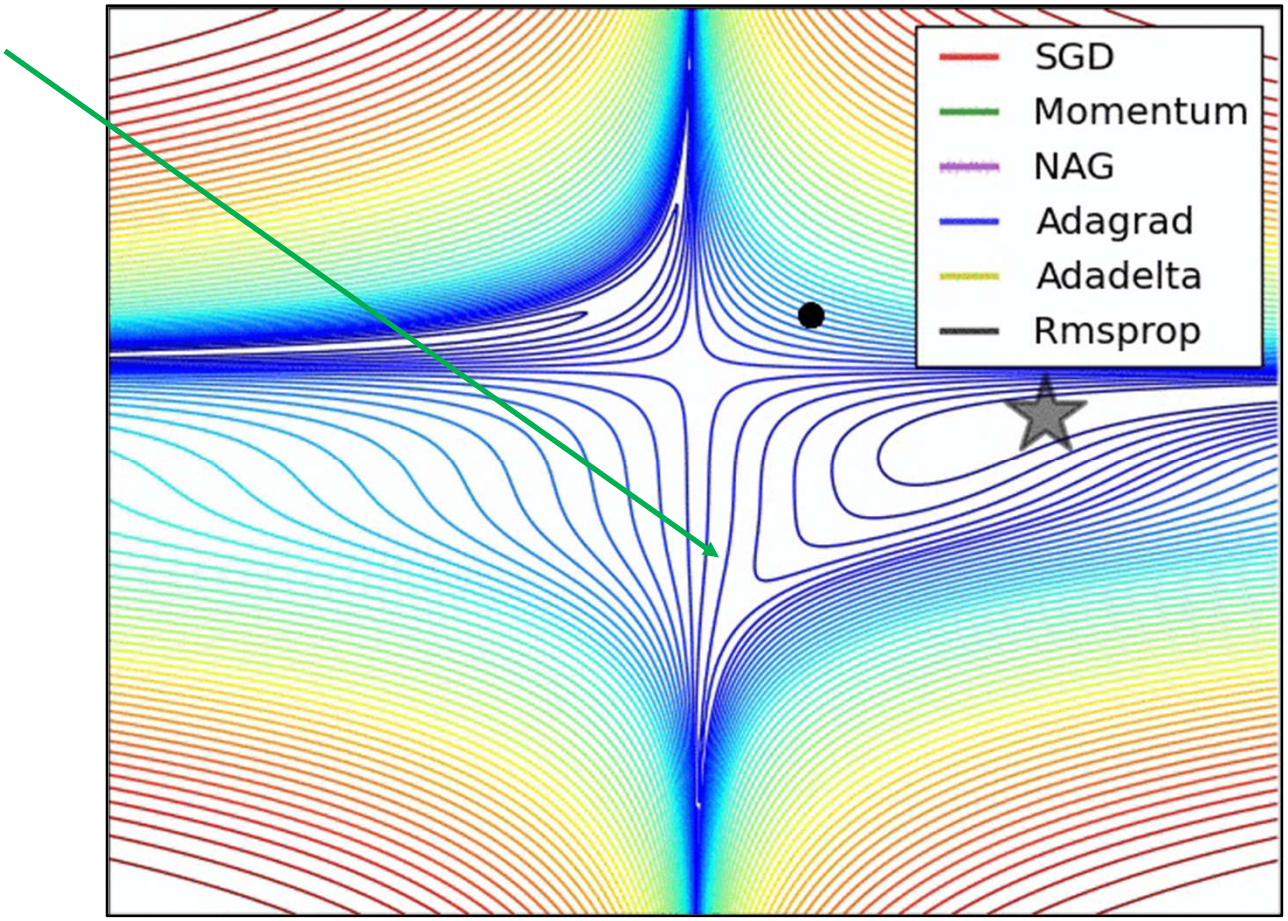
Slide from CS231n

SGD vs Momentum

Notice: momentum overshooting the target,

but overall getting to the minimum much

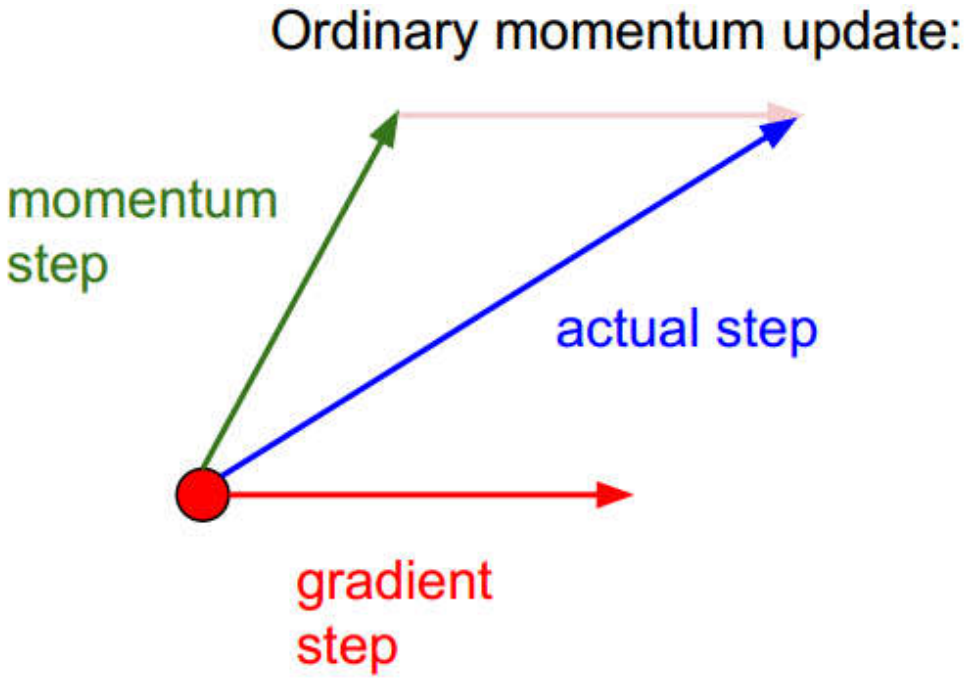
faster.



Slide from CS231n

Nesterov Accelerated Gradient (NAG)

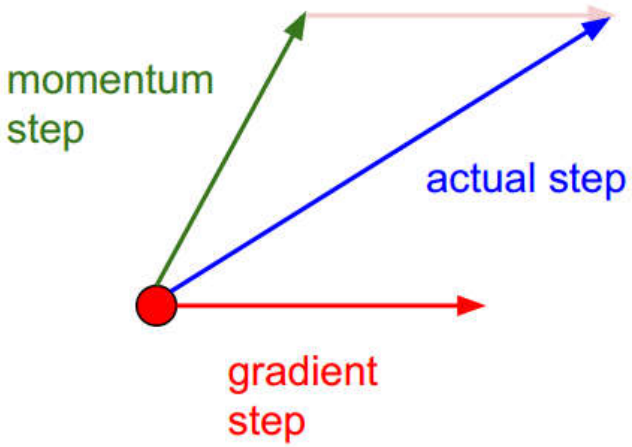
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```



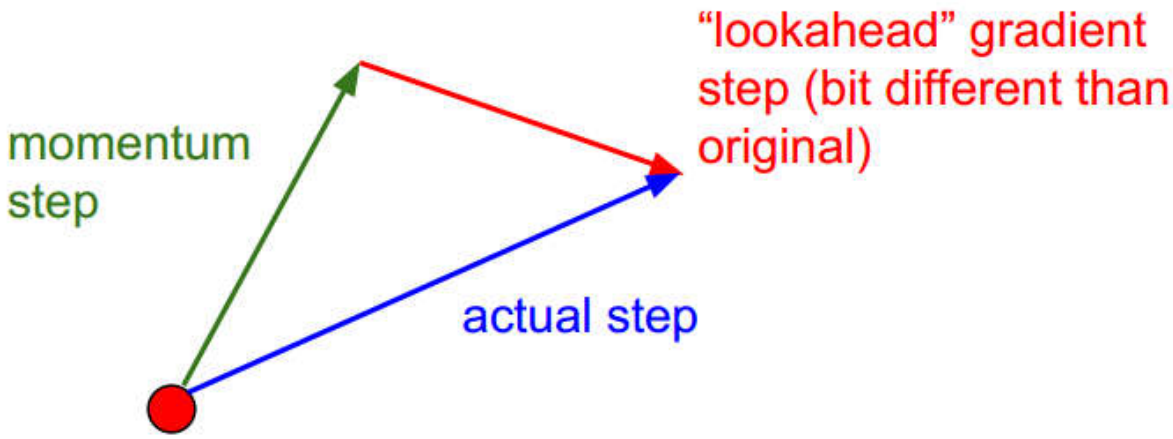
Slide from CS231n

Nesterov Accelerated Gradient (NAG)

Momentum update



Nesterov accelerated gradient

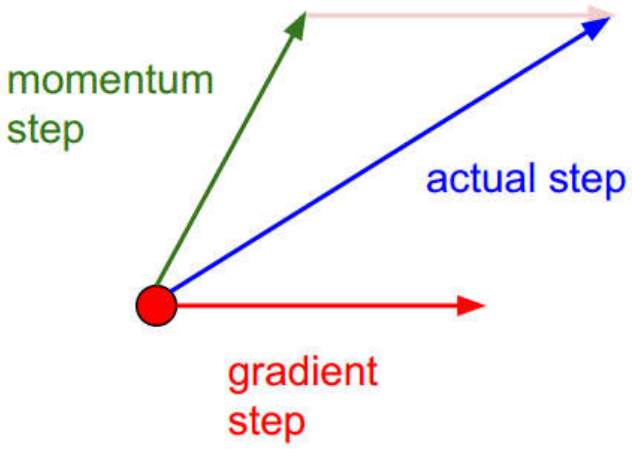


- **One step look ahead** gives you a slightly better direction
- Not just theory, **in practice** Nesterov accelerated gradient always works better than just standard momentum

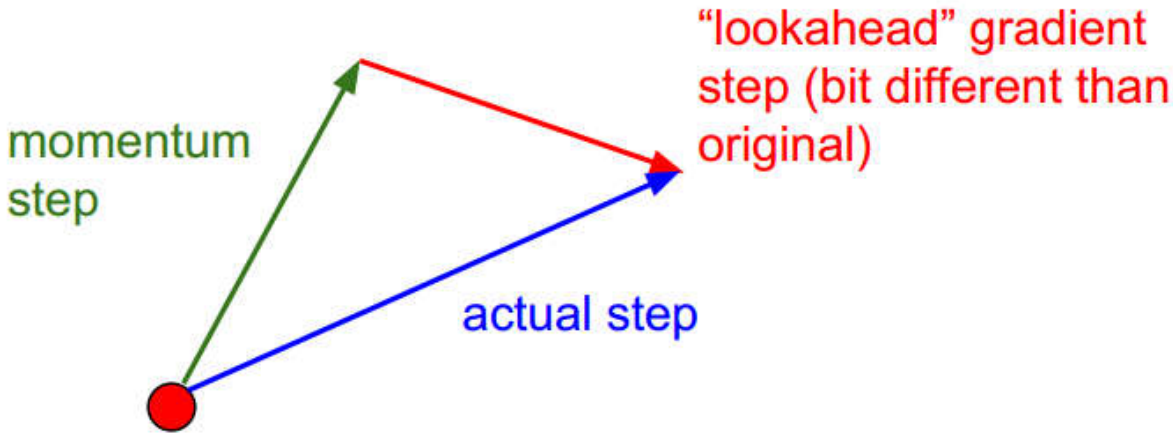
Slide from CS231n

Nesterov Accelerated Gradient (NAG)

Momentum update



Nesterov accelerated gradient



Nesterov: the only difference…

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

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

- **One step look ahead** gives you a slightly better direction
- Not just theory, **in practice** Nesterov accelerated gradient always works better than just standard momentum

Slide from CS231n

Nesterov Accelerated Gradient (NAG)

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

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

Slightly inconvenient…

usually we have:

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging:

Replace all ***thetas*** with ***phis***, rearrange

and obtain:

$$\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$$

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

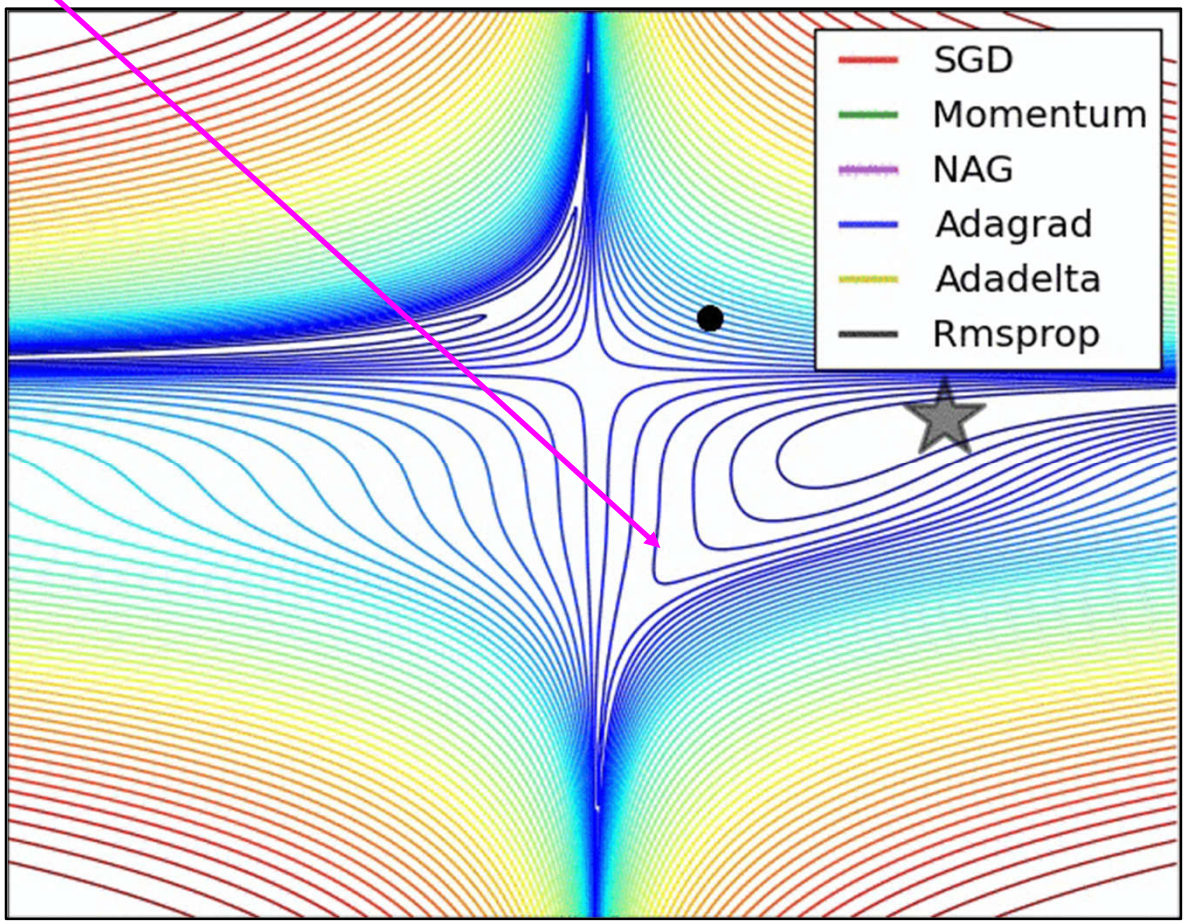
$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$

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

Slide from CS231n

NAG vs Momentum

NAG = Nesterove Accelerated Gradient



Slide from CS231n

AdaGrad Update

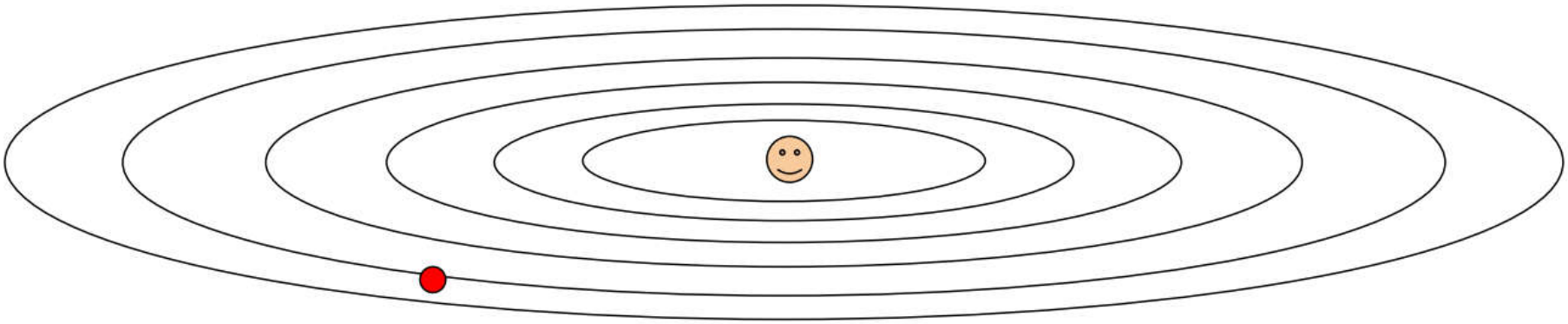
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

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

[Duchi et al., 2011]

AdaGrad Update

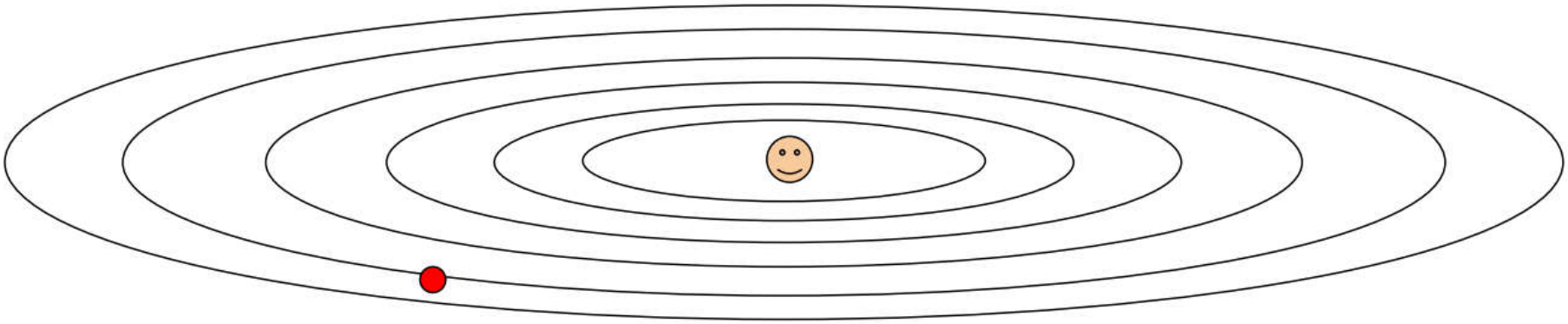
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Question: what happens with AdaGrad? (shallow horizontal and steep vertical)

AdaGrad Update

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



Question: what happens with AdaGrad? (shallow horizontal and steep vertical)

Question: what's the possible problem in AdaGrad optimizer?

Slide from CS231n

RMSProp Update

AdaGrad

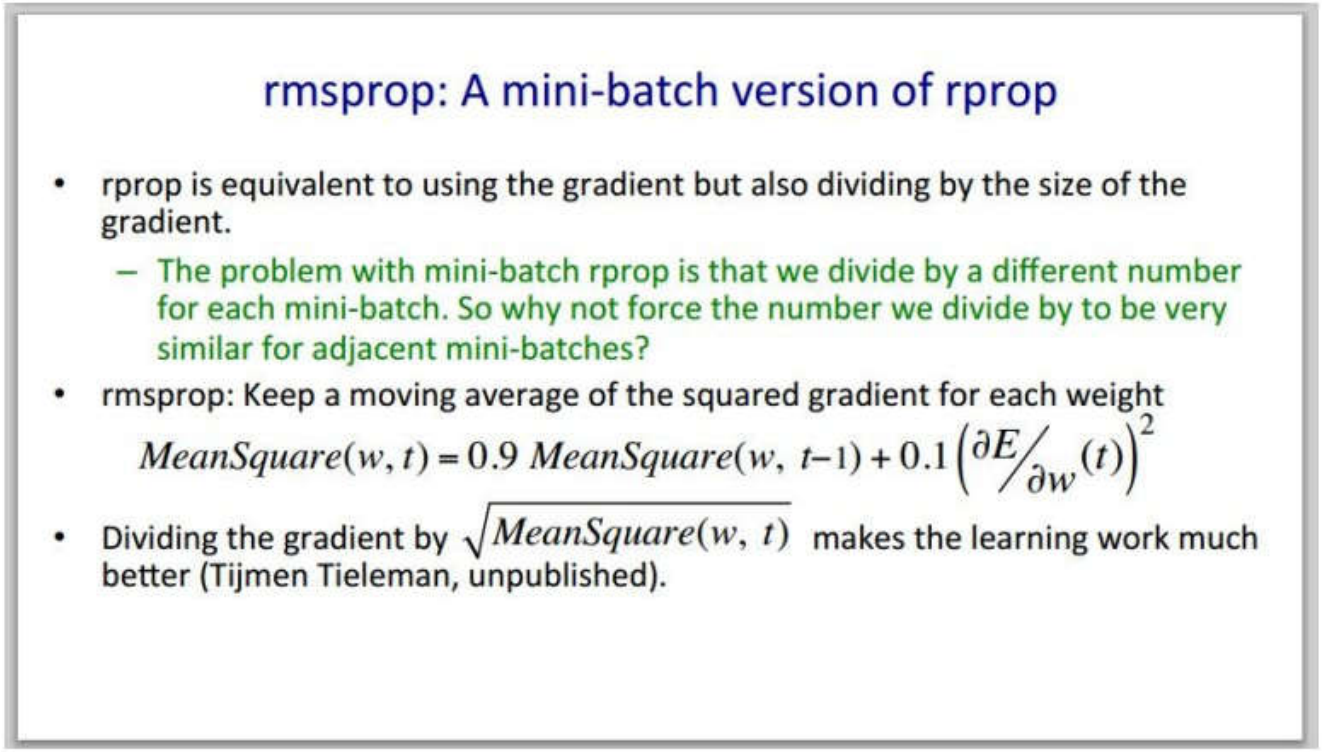
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```


RMSProp

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

[Tieleman and Hinton 2012]

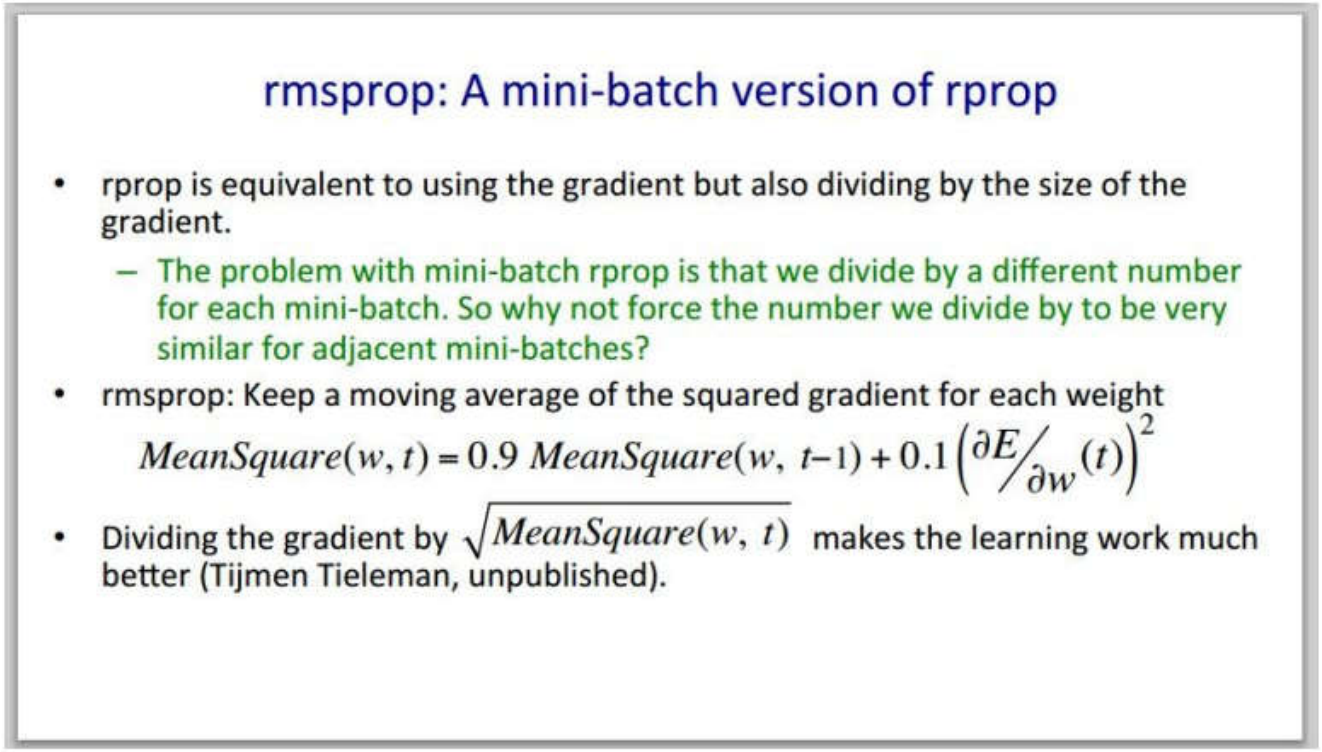
RMSProp Update



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

- **RMSProp is unpublished** but this usually works well in practice

RMSProp Update



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

- **RMSProp is unpublished** but this usually works well in practice
- Cited by several papers as:

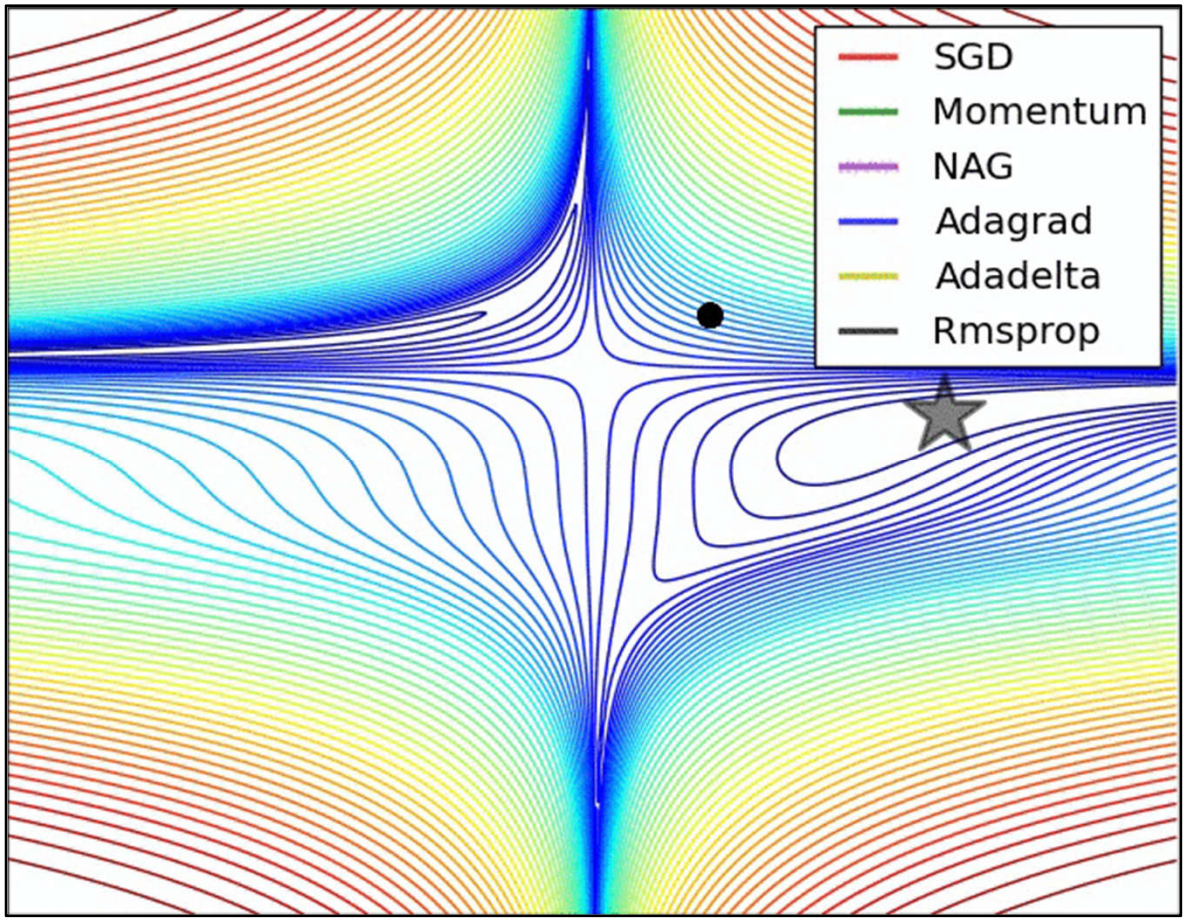
> [52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

Adagrad vs RMSProp

- In practice, adgrad stops too early and rmsprop usually winning out

Adagrad

Rmsprop



Slide from CS231n

Adam Update

```
# 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** keeps track of the first order moment of the gradient
- **Adagrad** keeps track of the second moment of the gradient

Kingma and Ba, 2014

Adam Update

```
# 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**

Kingma and Ba, 2014

Adam Update

```
# 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

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

Kingma and Ba, 2014

Adam Update

```
# 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

(incomplete, but close)

RMSProp-like

- Looks a bit like **RMSProp** with **Momentum**

- beta1 and beta2 are hyper-parameters, beta1 usually 0.9, beta2 usually 0.995… (we can cross-validation, but usually we fix these values)

RMSProp

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

Kingma and Ba, 2014

Adam Update

```
# 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)

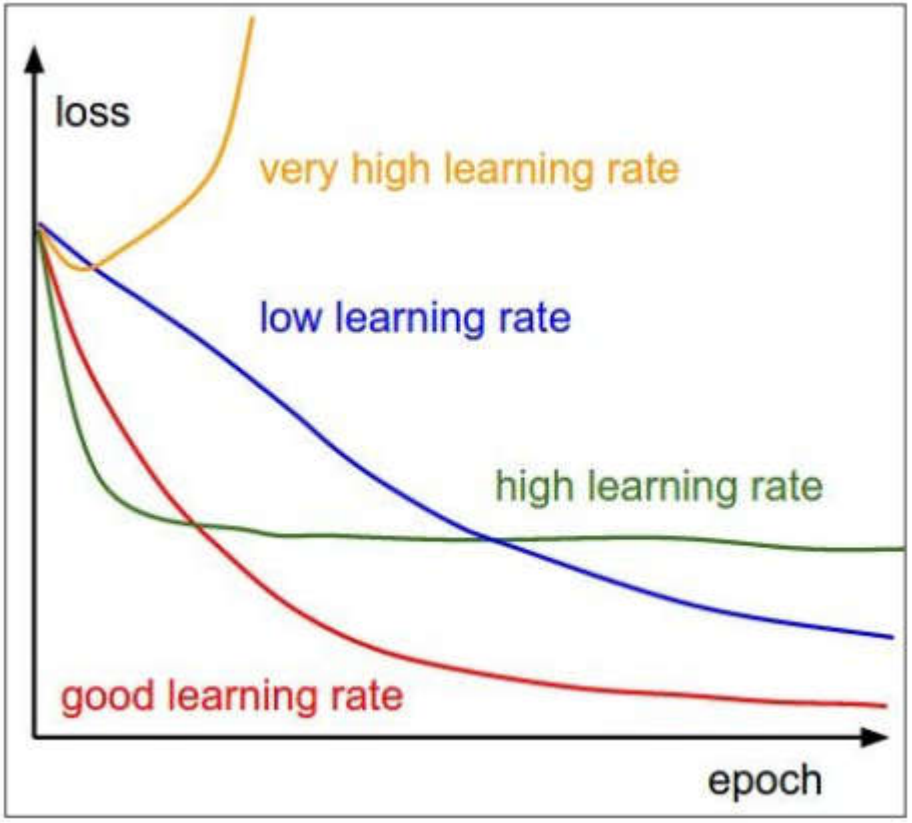
RMSProp-like

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

Slide from CS231n

Optimizers

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

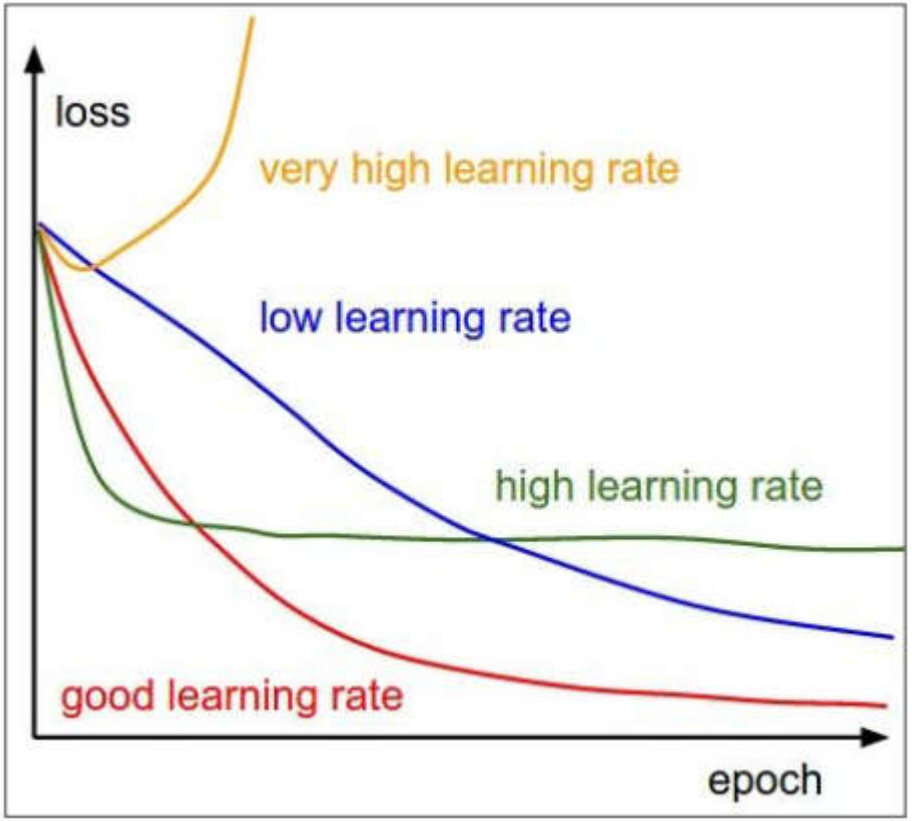


Question: which one of these learning rates is best to use?

Slide from CS231n

Optimizers

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



$\Rightarrow$ **Learning rate decay over time!**

Step decay:

e.g. decay learning rate by half every few epochs. $\alpha = a_0 \cdot t^k$

Exponential decay: $\alpha = a_0 e^{-kt}$

1/t decay: $\alpha = a_0 / (1 + kt)$

Slide from CS231n

Second Order Optimization Methods

- **Second-order Taylor expansion:**

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \boldsymbol{H} (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1} \nabla_\theta J(\theta_0)$$

Question: what is nice about this update?

Second Order Optimization Methods

- **Second-order Taylor expansion:**

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \boldsymbol{H} (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1} \nabla_\theta J(\theta_0)$$

Notice:

No hyperparameters! (e.g. learning rate)

Question: what is nice about this update?

Second Order Optimization Methods

- **Second-order Taylor expansion:**

Hessian matrix

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \boldsymbol{H} (\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1} \nabla_\theta J(\theta_0)$$

Notice:

No hyperparameters! (e.g. learning rate)

Question: why is this impractical for training Deep Neural Nets?

Second Order Optimization Methods

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1}\nabla_\theta J(\theta_0)$$

- Quasi-Newton methods (**BFGS** most popular): instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).
  - Don't calculate inverse of Hessian, but still need memory to save Hessian

- **L-BFGS** (Limited memory BFGS):
  - Does not form/store the full inverse Hessian.

Question: guess the meaning of BGFS

Slide from CS231n

Second Order Optimization Methods

$$\theta^* = \theta_0 - \boldsymbol{H}^{-1}\nabla_\theta J(\theta_0)$$

- Quasi-Newton methods (**BFGS** most popular): instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).
  - Don't calculate inverse of Hessian, but still need memory to save Hessian

- **L-BFGS** (Limited memory BFGS):
  - Does not form/store the full inverse Hessian.

Question: guess the full name of BFGS

Broyden–Fletcher–Goldfarb–Shanno algorithm

From Wikipedia, the free encyclopedia

Slide from 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 activate area of research.

In Summary

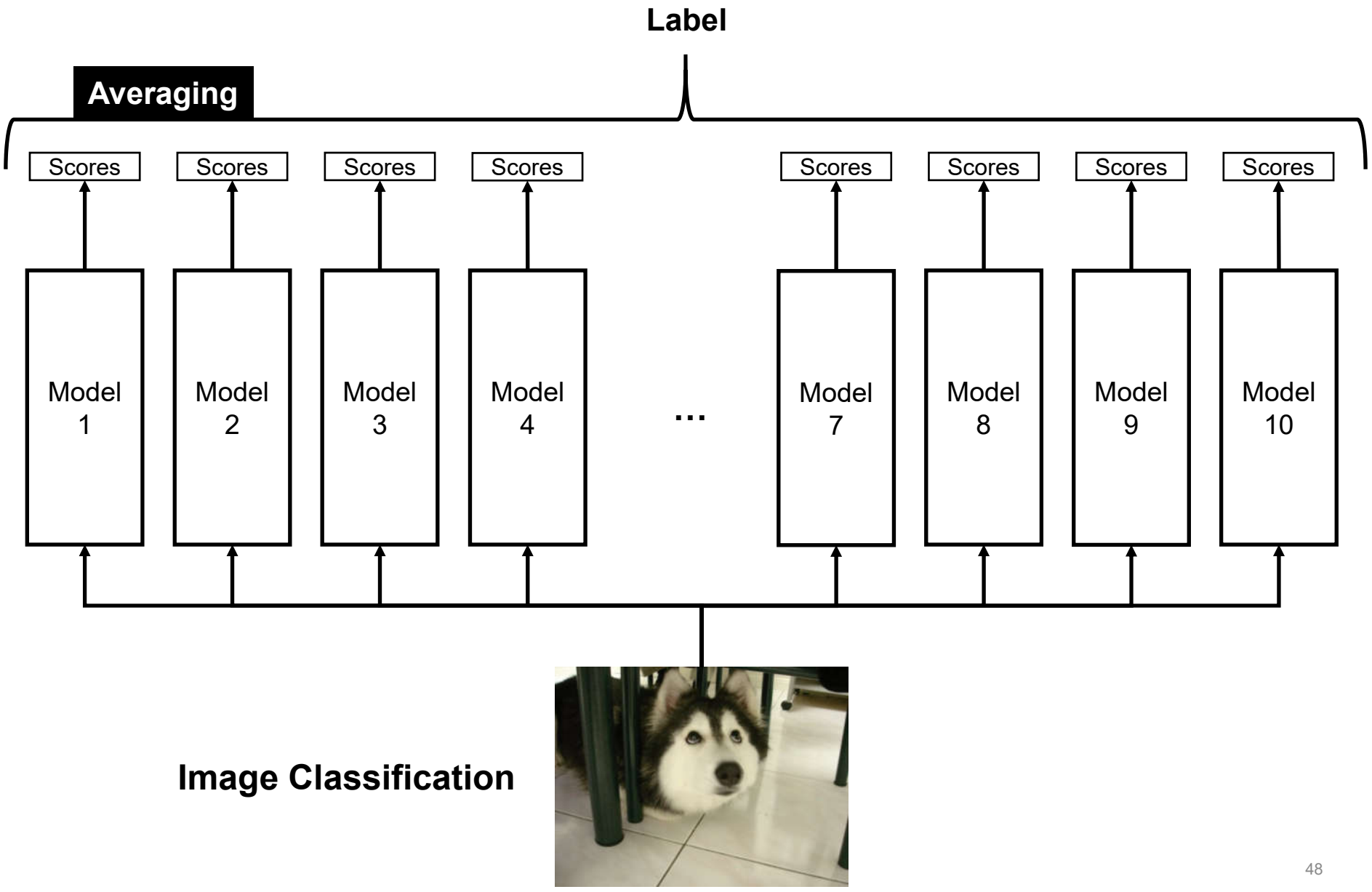
- **Adam** is a good default choice in most cases

- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

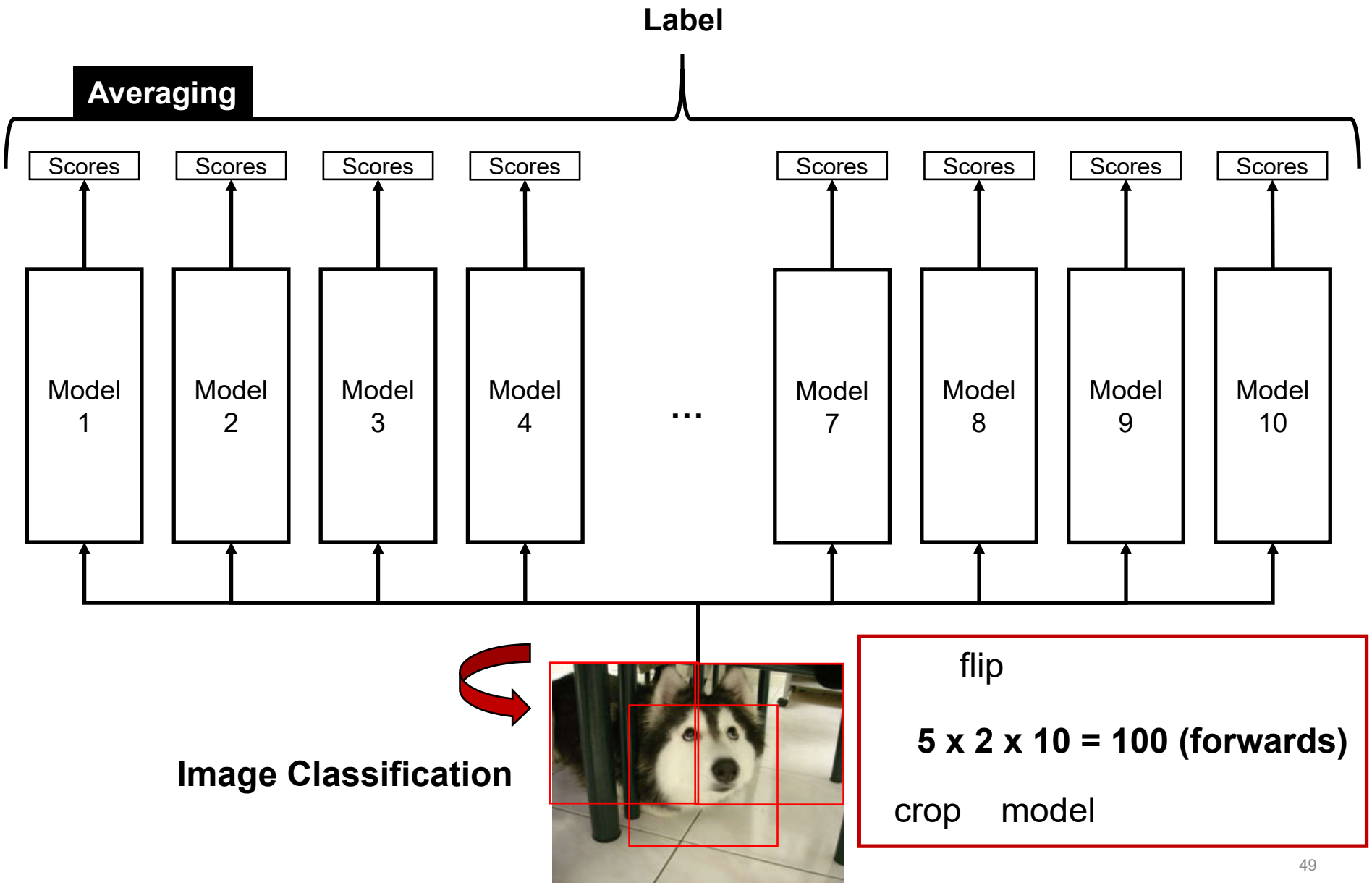
Slide from CS231n

Model Ensembles

Model Ensembles



Model Ensembles



Model Ensembles

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

Enjoy 2 % extra performance

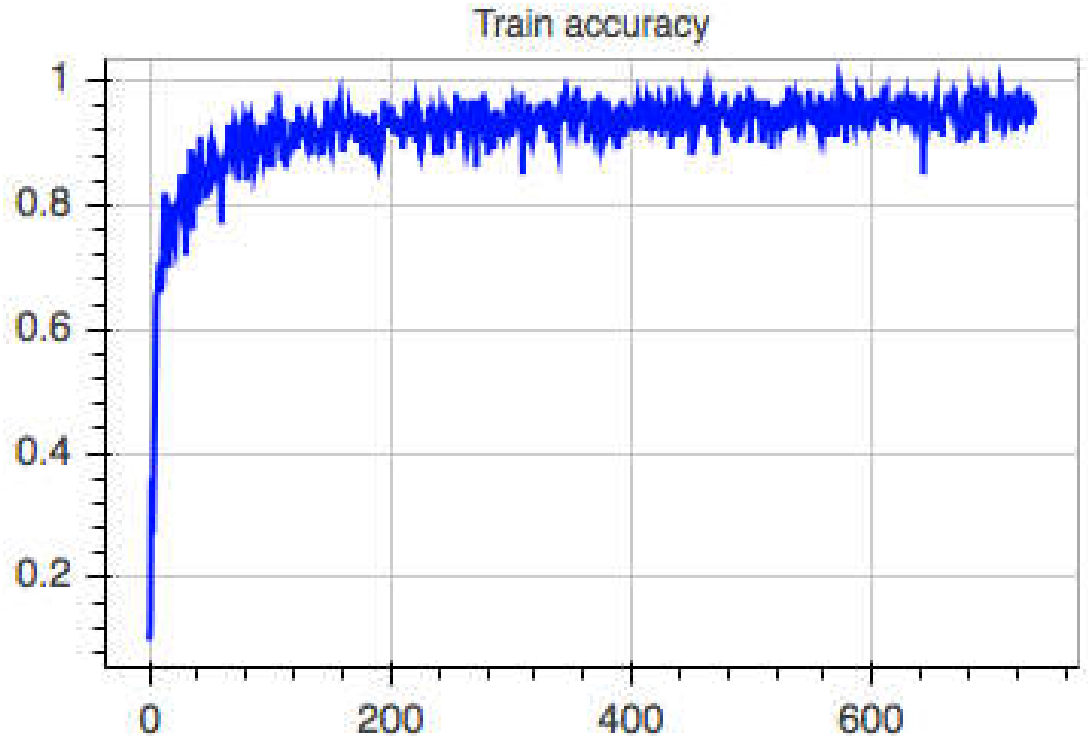
This is a not really a theoretical result here, it is kind of like an Andrej's result. In practice, this is like a good thing to do almost always works better.

Disadvantage: Complex training process…

Model Ensembles

Fun Tips/Tricks:

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



Disadvantage: too much memory…

Slide from CS231n

Model Ensembles

Fun Tips/Tricks:

1. Can also get a small boost from averaging multiple model checkpoints of a single model.
2. 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
```

- Can slightly better than use x alone usually

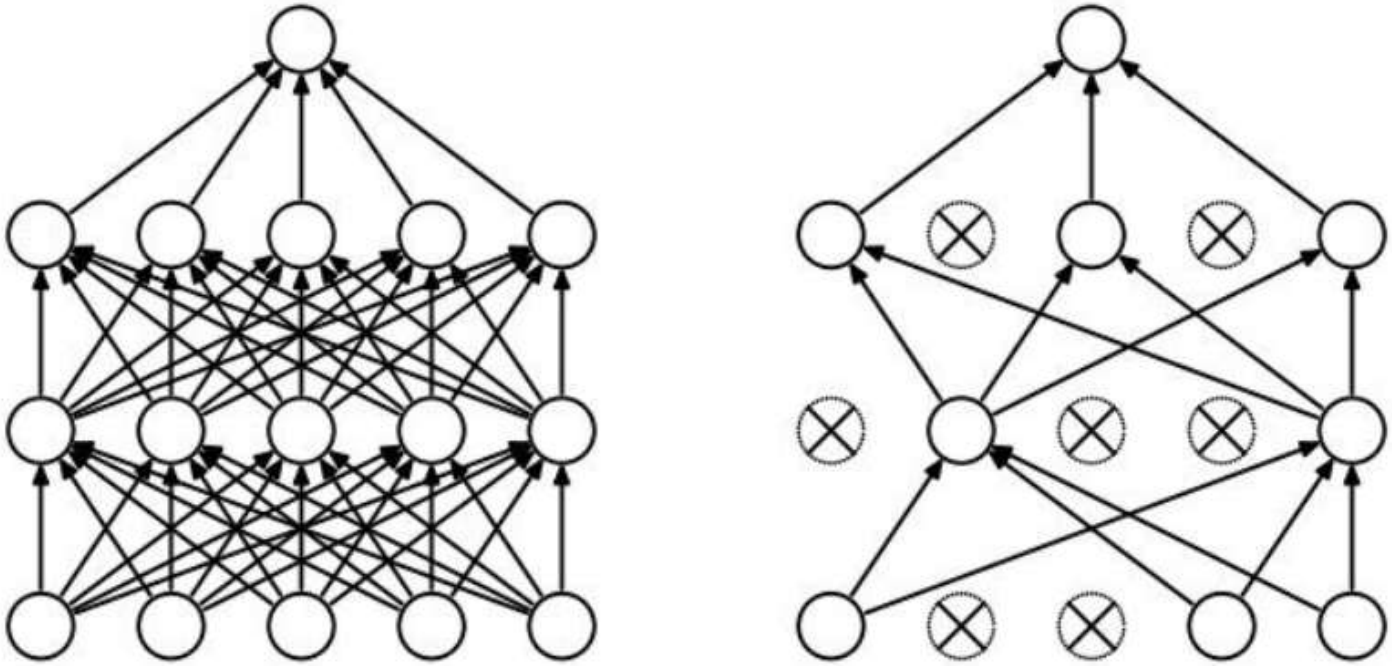
Slide from CS231n

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]

Regularization: Dropout

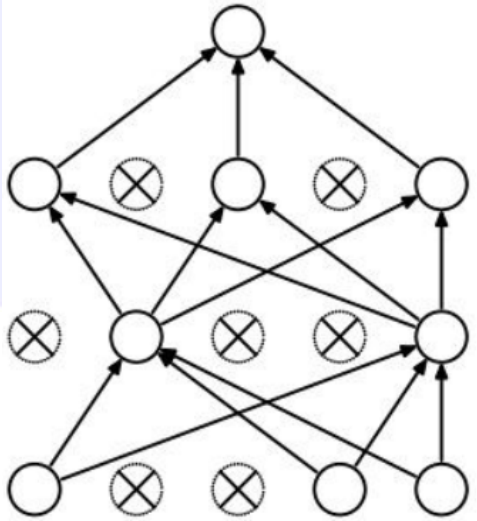
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

3 layer neural networks

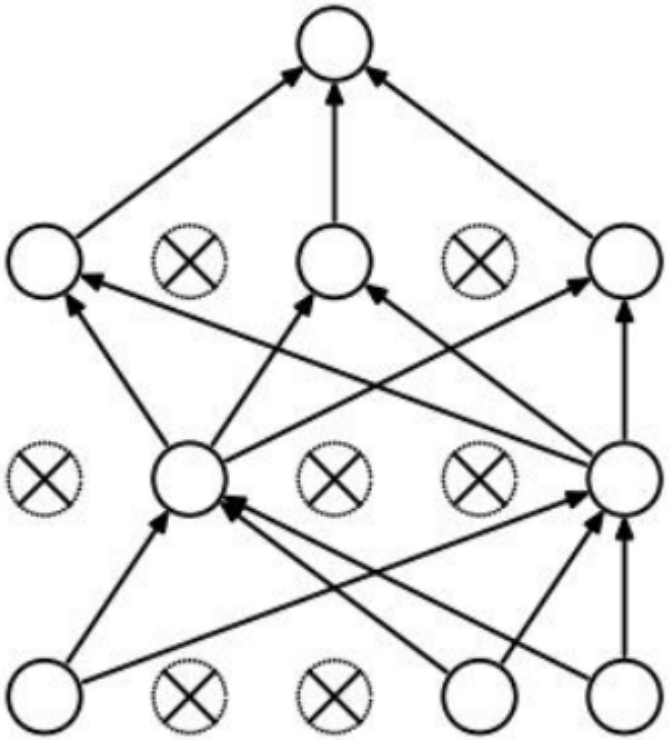


- Example forward pass with a 3-layer network using dropout

Slide from CS231n

Regularization: Dropout

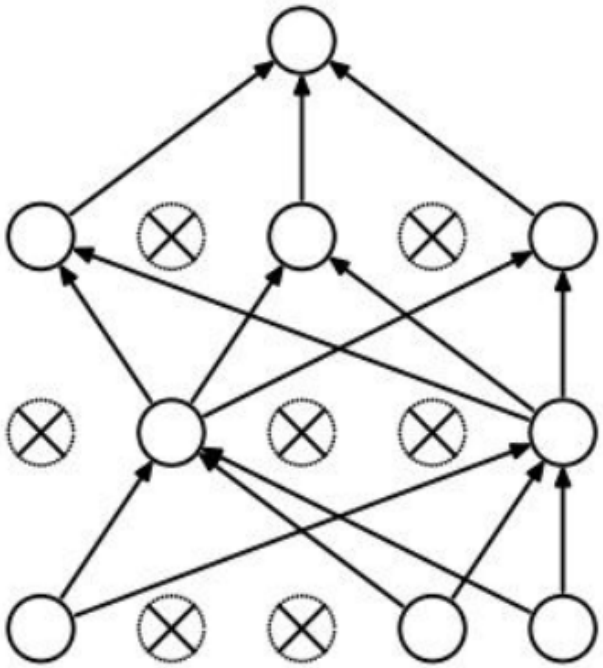
Question: how could this possible be a good idea?



Slide from CS231n

Regularization: Dropout

Question: how could this possible be a good idea?

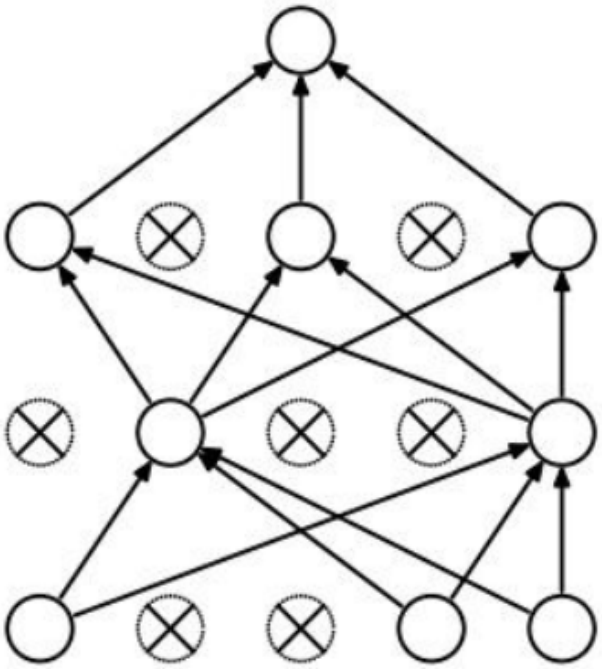


- Forces the network to have a **redundant representation**

Slide from CS231n

Regularization: Dropout

Question: how could this possible be a good idea?



Another interpretation:

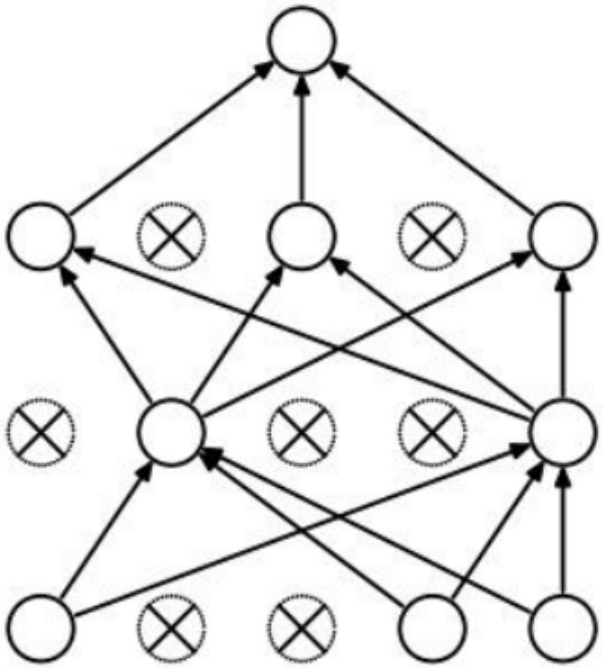
Dropout is training **a large ensemble of models** (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

- Dropout can overcome overfitting at some degree

Regularization: Dropout

At test time…



Ideally:

- Want to integrate out all the noise

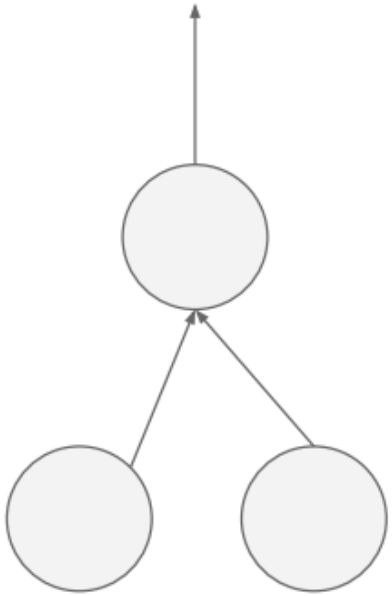
Monte Carlo approximation:

- Do many forward passes with different dropout masks, average all predictions

Regularization: Dropout

At test time…

- In fact do this with a single forward pass! **(approximately)**
- Leave all input neurons turned on **(no dropout)**.

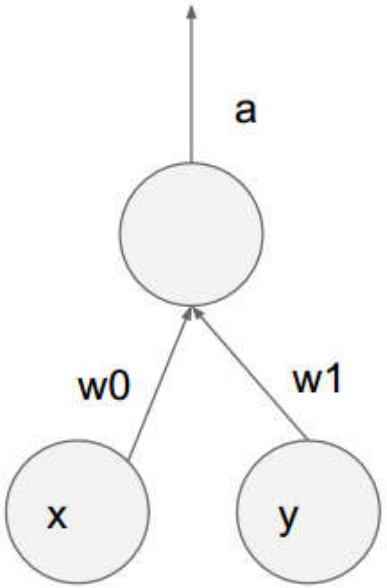


(this can be shown to be an **approximation to evaluating the whole ensemble**)

Regularization: Dropout

At test time…

- In fact do this with a single forward pass! **(approximately)**
- Leave all input neurons turned on **(no dropout)**.



Question: 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)

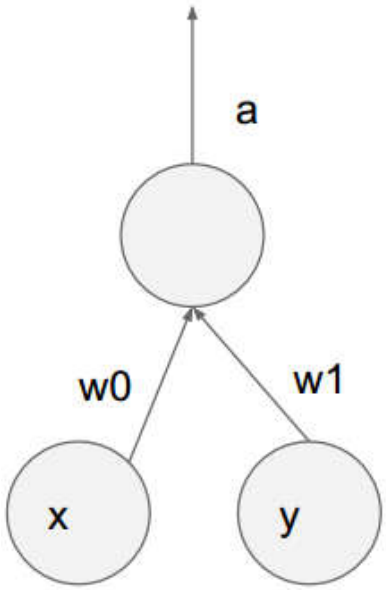
a = w0 * x + w1 * y

Slide from CS231n

Regularization: Dropout

At test time…

- In fact do this with a single forward pass! **(approximately)**
- Leave all input neurons turned on **(no dropout)**.



During test: a = w0 * x + w1 * y

During train:

E[a] = ¼ * (w0 * 0 + w1 * 0
w0 * 0 + w1 * y
w0 * x + w1 * 0
w0 * x + w1 * y)

= ¼ * (2 w0 * x + 2 w1 * y)

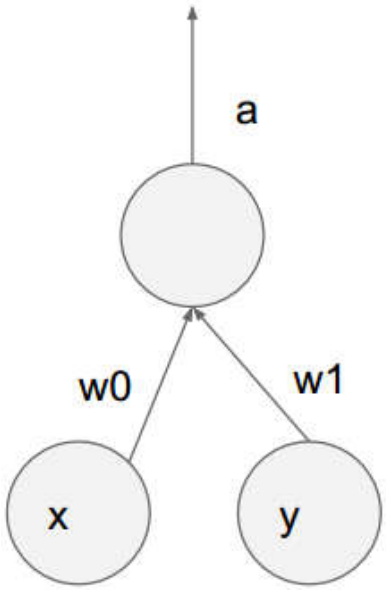
= ½ * **(w0 * x + w1 * y)**

Slide from CS231n

Regularization: Dropout

At test time…

- In fact do this with a single forward pass! **(approximately)**
- Leave all input neurons turned on **(no dropout)**.



During test: a = w0 * x + w1 * y

During train:

E[a] = ¼ * (w0 * 0 + w1 * 0

w0 * 0 + w1 * y

w0 * x + w1 * 0

w0 * x + w1 * y)

= ¼ * (2 w0 * x + 2 w1 * y)

= ½ * **(w0 * x + w1 * y)**

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network has "used to" during training!

=> Have to compensate by scaling the activations back down by 1/2

Slide from CS231n

Regularization: Dropout

```
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

Slide from 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

Slide from 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
```

Scaling here at training time

test time is unchanged!

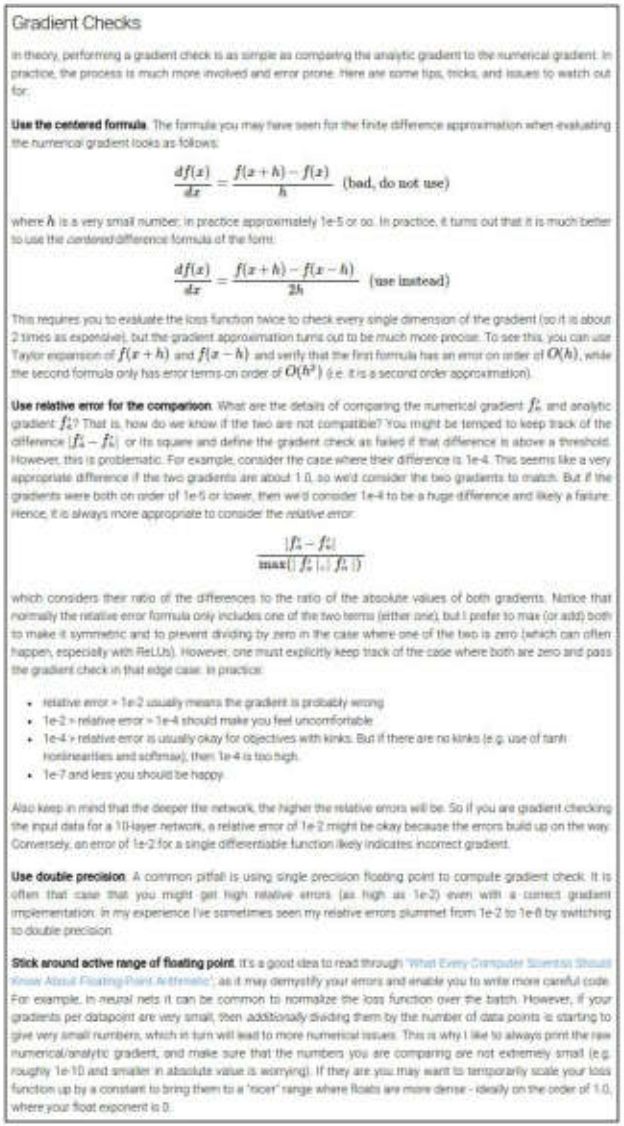
Slide from CS231n

Gradient Checking

Gradient Checking

See class notes…

- h?
- float64?
- relative error?



Gradient Checks

In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient. In practice, the process is much more involved and error prone. Here are some tips, tricks, and issues to watch out for:

Use the centered formula. The formula you may have seen for the finite difference approximation when evaluating the numerical gradient looks as follows:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

where h is a very small number, in practice approximately 1e-5 or so. In practice, it turns out that it is much better to use the *centered* difference formula of the form:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

This requires you to evaluate the loss function twice to check every single dimension of the gradient (so it is about 2 times as expensive), but the gradient approximation turns out to be much more precise. To see this, you can use Taylor expansion of $f(x+h)$ and $f(x-h)$ and verify that the first formula has an error on order of $O(h)$, while the second formula only has error terms on order of $O(h^2)$ (i.e. it is a second order approximation).

Use relative error for the comparison. What are the details of comparing the numerical gradient f_n' and analytic gradient f_a'? That is, how do we know if the two are not compatible? You might be tempted to keep track of the difference $|f_a' - f_n'|$ or its square and define the gradient check as failed if that difference is above a threshold. However, this is problematic. For example, consider the case where their difference is 1e-4. This seems like a very appropriate difference if the two gradients are about 1.0, so we'd consider the two gradients to match. But if the gradients were both on order of 1e-5 or lower, then we'd consider 1e-4 to be a huge difference and likely a failure. Hence, it is always more appropriate to consider the *relative error*:

$$\frac{|f_a' - f_n'|}{\max(|f_a'|, |f_n'|)}$$

which considers their ratio of the differences to the ratio of the absolute values of both gradients. Notice that normally the relative error formula only includes one of the two terms (either one), but I prefer to max (or add) both to make it symmetric and to prevent dividing by zero in the case where one of the two is zero (which can often happen, especially with ReLUs). However, one must explicitly keep track of the case where both are zero and pass the gradient check in that edge case. In practice:

- relative error > 1e-2 usually means the gradient is probably wrong
- 1e-2 > relative error > 1e-4 should make you feel uncomfortable
- 1e-4 > relative error is usually okay for objectives with kinks. But if there are no kinks (e.g. use of tanh nonlinearities and softmax), then 1e-4 is too high.
- 1e-7 and less you should be happy.

Also keep in mind that the deeper the network, the higher the relative errors will be. So if you are gradient checking the input data for a 10-layer network, a relative error of 1e-2 might be okay because the errors build up on the way. Conversely, an error of 1e-2 for a single differentiable function likely indicates incorrect gradient.

Use double precision. A common pitfall is using single precision floating point to compute gradient check. It is often that case that you might get high relative errors (as high as 1e-2) even with a correct gradient implementation. In my experience I've sometimes seen my relative errors plummet from 1e-2 to 1e-8 by switching to double precision.

Stick around active range of floating point. It's a good idea to read through "What Every Computer Scientist Should Know About Floating-Point Arithmetic", as it may demystify your errors and enable you to write more careful code. For example, in neural nets it can be common to normalize the loss function over the batch. However, if your gradients per datapoint are very small, then additionally dividing them by the number of data points is starting to give very small numbers, which in turn will lead to more numerical issues. This is why I like to always print the raw numerical/analytic gradient, and make sure that the numbers you are comparing are not extremely small (e.g. roughly 1e-10 and smaller in absolute value is worrying). If they are you may want to temporarily scale your loss function up by a constant to bring them to a "nicer" range where floats are more dense - ideally on the order of 1.0, where your float exponent is 0.

Kinks in the objective. One source of inaccuracy to be aware of during gradient checking is the problem of kinks. Kinks refer to non-differentiable parts of an objective function, introduced by functions such as ReLU ($\max(0, x)$), or the SVM loss, Maxout neurons, etc. Consider gradient checking the ReLU function at $x = -1e6$. Since $x < 0$, the analytic gradient at this point is exactly zero. However, the numerical gradient would suddenly compute a non-zero gradient because $f(x+h)$ might cross over the kink (e.g. if $h > 1e-6$) and introduce a non-zero contribution. You might think that this is a pathological case, but in fact this case can be very common. For example, an SVM for CIFAR-10 contains up to 450,000 $\max(0, x)$ terms because there are 50,000 examples and each example yields 9 terms to the objective. Moreover, a Neural Network with an SVM classifier will contain many more kinks due to ReLUs.

Note that it is possible to know if a kink was crossed in the evaluation of the loss. This can be done by keeping track of the identities of all "winners" in a function of form $\max(x, y)$. That is, was x or y higher during the forward pass. If the identity of at least one winner changes when evaluating $f(x+h)$ and then $f(x-h)$, then a kink was crossed and the numerical gradient will not be exact.

Use only few datapoints. One fix to the above problem of kinks is to use fewer datapoints, since loss functions that contain kinks (e.g. due to use of ReLUs or margin losses etc.) will have fewer kinks with fewer datapoints, so it is less likely for you to cross one when you perform the finite different approximation. Moreover, if your gradcheck for only ~2 or 3 datapoints then you would almost certainly gradcheck for an entire batch. Using very few datapoints also makes your gradient check faster and more efficient.

Be careful with the step size h. It is not necessarily the case that smaller is better, because when h is much smaller, you may start running into numerical precision problems. Sometimes when the gradient doesn't check, it is possible that you change h to be 1e-4 or 1e-6 and suddenly the gradient will be correct. This wikipedia article contains a chart that plots the value of **h** on the x-axis and the numerical gradient error on the y-axis.

Gradcheck during a "characteristic" mode of operation. It is important to realize that a gradient check is performed at a particular (and usually random), single point in the space of parameters. Even if the gradient check succeeds at that point, it is not immediately certain that the gradient is correctly implemented globally. Additionally, a random initialization might not be the most "characteristic" point in the space of parameters and may in fact introduce pathological situations where the gradient seems to be correctly implemented but isn't. For instance, an SVM with very small weight initialization will assign almost exactly zero scores to all datapoints and the gradients will exhibit a particular pattern across all datapoints. An incorrect implementation of the gradient could still produce this pattern and not generalize to a more characteristic mode of operation where some scores are larger than others. Therefore, to be safe it is best to use a short **burn-in** time during which the network is allowed to learn and perform the gradient check after the loss starts to go down. The danger of performing it at the first iteration is that this could introduce pathological edge cases and mask an incorrect implementation of the gradient.

Don't let the regularization overwhelm the data. It is often the case that a loss function is a sum of the data loss and the regularization loss (e.g. L2 penalty on weights). One danger to be aware of is that the regularization loss may overwhelm the data loss, in which case the gradients will be primarily coming from the regularization term (which usually has a much simpler gradient expression). This can mask an incorrect implementation of the data loss gradient. Therefore, it is recommended to turn off regularization and check the data loss alone first, and then the regularization term second and independently. One way to perform the latter is to hack the code to remove the data loss contribution. Another way is to increase the regularization strength so as to ensure that its effect is non-negligible in the gradient check, and that an incorrect implementation would be spotted.

Remember to turn off dropout/augmentations. When performing gradient check, remember to turn off any non-deterministic effects in the network, such as dropout, random data augmentations, etc. Otherwise these can clearly introduce huge errors when estimating the numerical gradient. The downside of turning off these effects is that you wouldn't be gradient checking them (e.g. it might be that dropout isn't backpropagated correctly). Therefore, a better solution might be to force a particular random seed before evaluating both $f(x+h)$ and $f(x-h)$, and when evaluating the analytic gradient.

Check only few dimensions. In practice the gradients can have sizes of million parameters. In these cases it is only practical to check some of the dimensions of the gradient and assume that the others are correct. **Be careful**: One issue to be careful with is to make sure to gradient check a few dimensions for every separate parameter. In some applications, people combine the parameters into a single large parameter vector for convenience. In these cases, for example, the biases could only take up a tiny number of parameters from the whole vector, so it is important to not sample at random but to take this into account and check that all parameters receive the correct gradients.

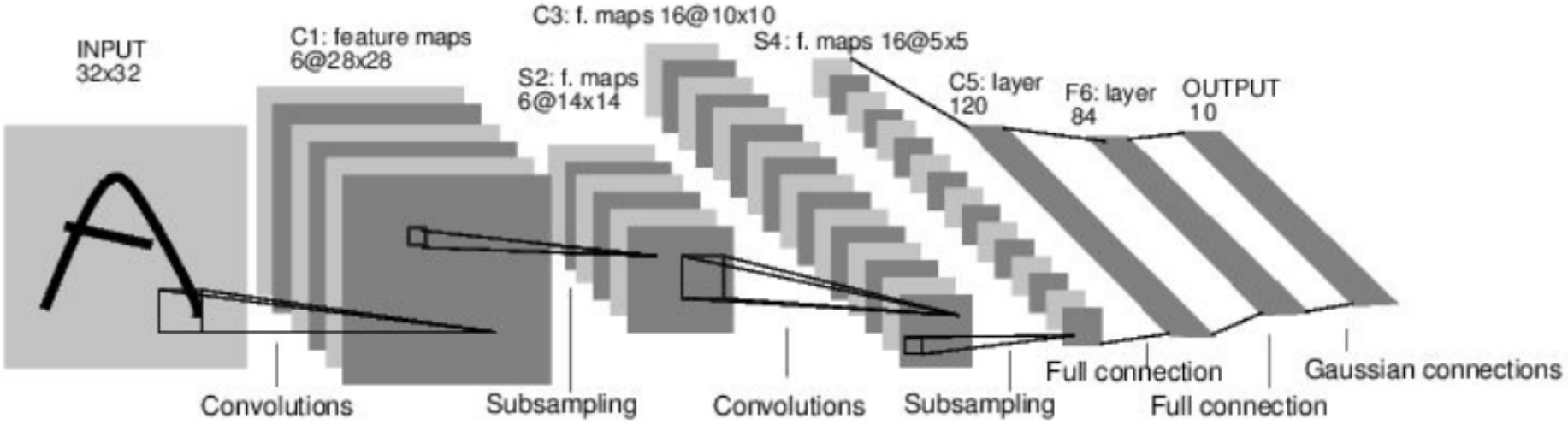
Please refer to the lecture note:
http://cs231n.github.io/neural-networks-3/

Slide from CS231n

Convolutional Neural Networks

A Bit of History

Convolutional Neural Networks



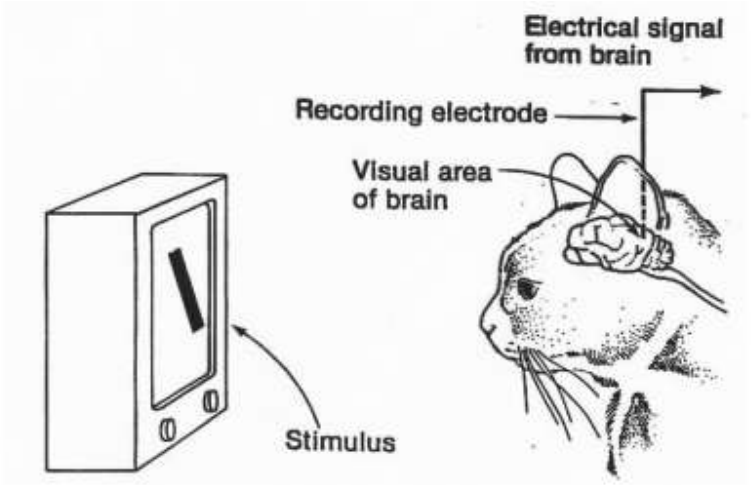
[LeNet-5, LeCun 1980s]

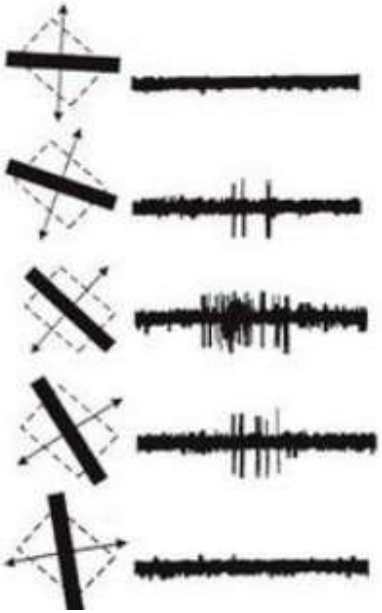
Hubel & Wiesel, 1960s

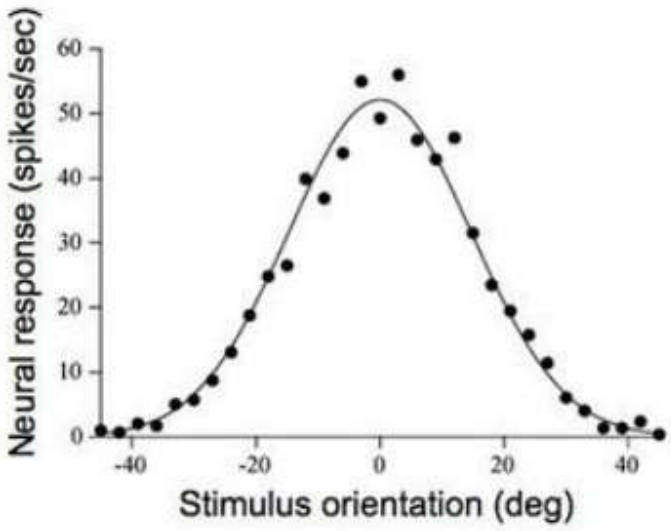
- Recording visual cortex of cat in the first visual area of processing in the back of brain called **v1**

- Winning a Nobel Prize

- The cells excited for edges of particular orientation

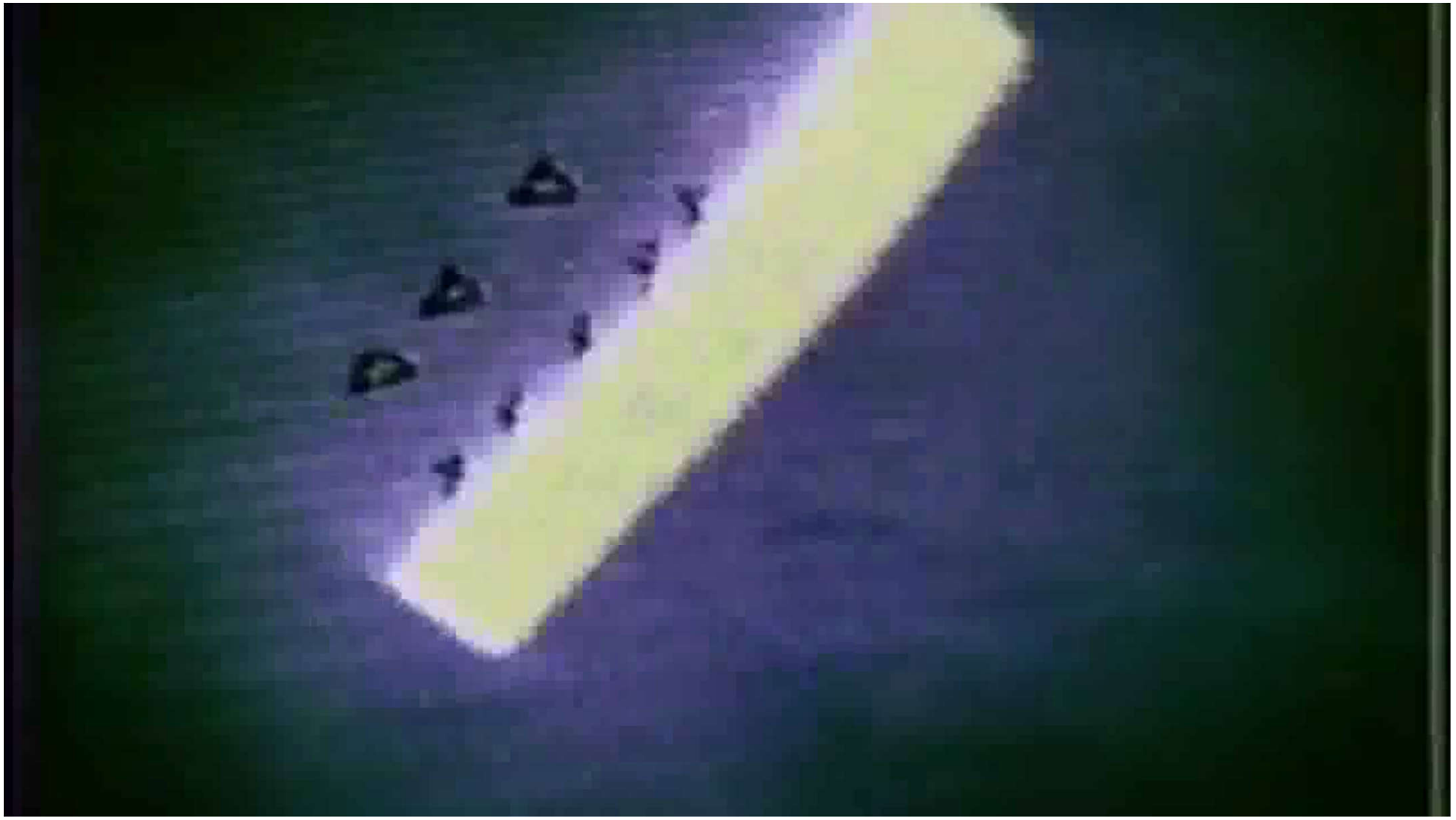






Slide from CS231n

Hubel & Wiesel, 1960s

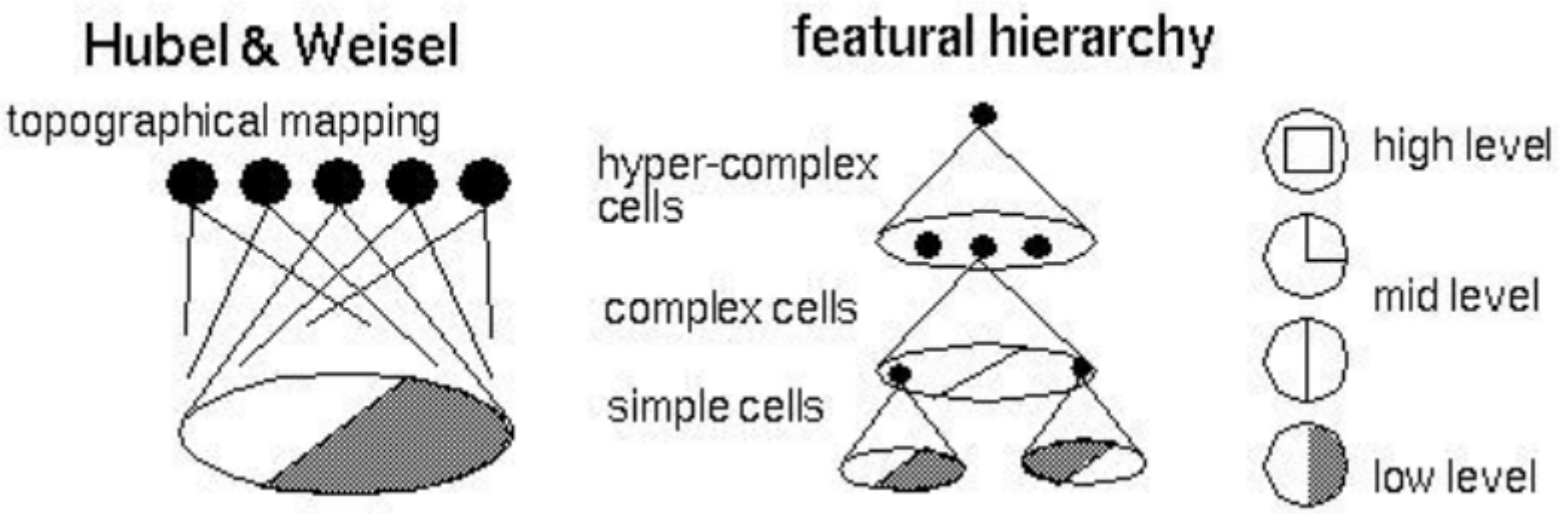


https://www.youtube.com/watch?v=8VdFf3egwfg

Slide from CS231n

Hubel & Wiesel, 1960s

- Cortex has a kind of hierarchical organization
- A simple cells that are feeding to other cells called complex cells and etc.
- These cells are building on top of each other
- The simple cells have some local receptive fields and then are built up more and more complex representations in the brain



Slide from CS231n

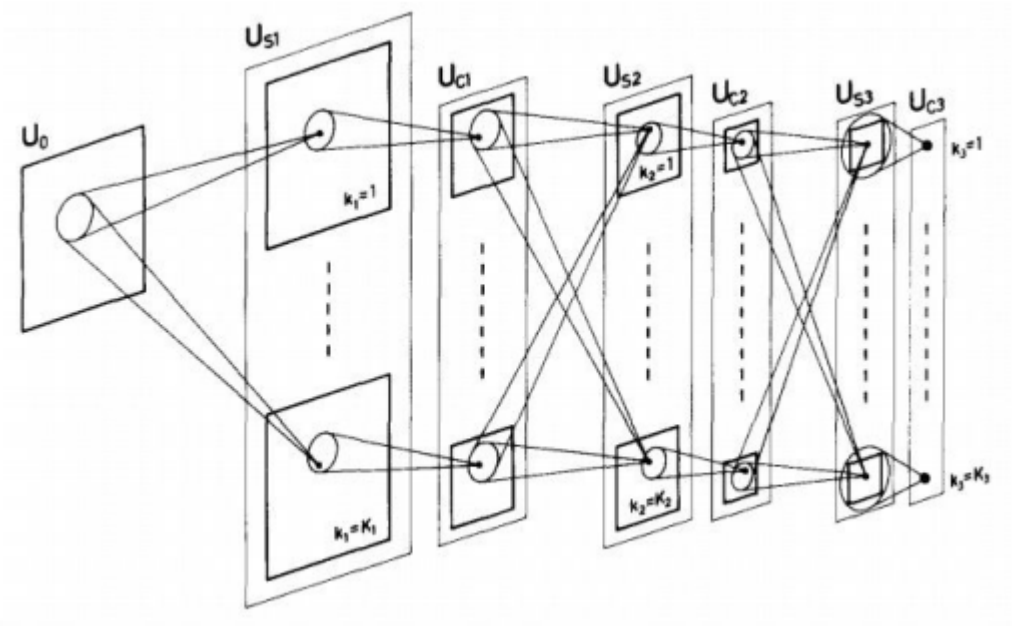
Fukushima 1980

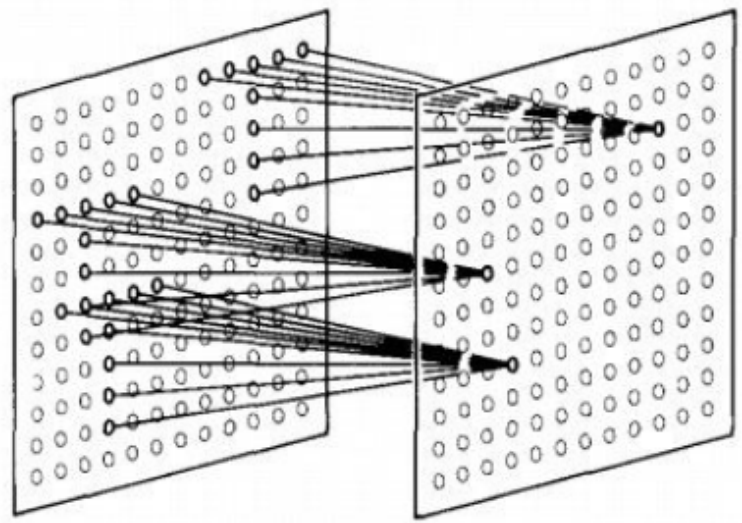
"Sandwich" architecture (SCSCSC…)

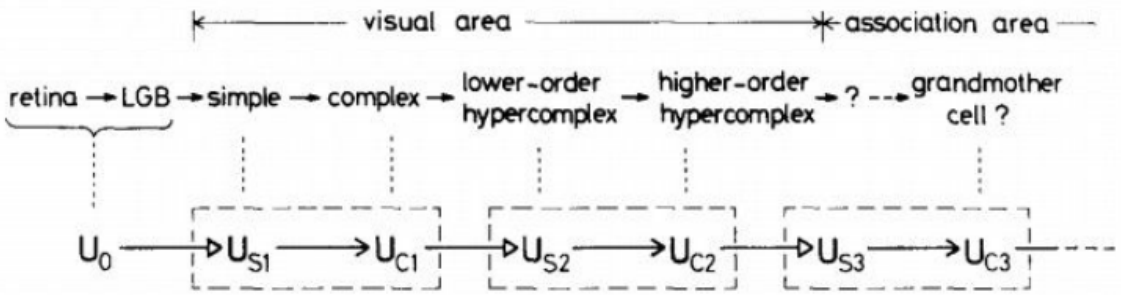
Simple cells: modifiable parameters

Complex cells: perform pooling

<u>Don't have backpropagation scheme</u>



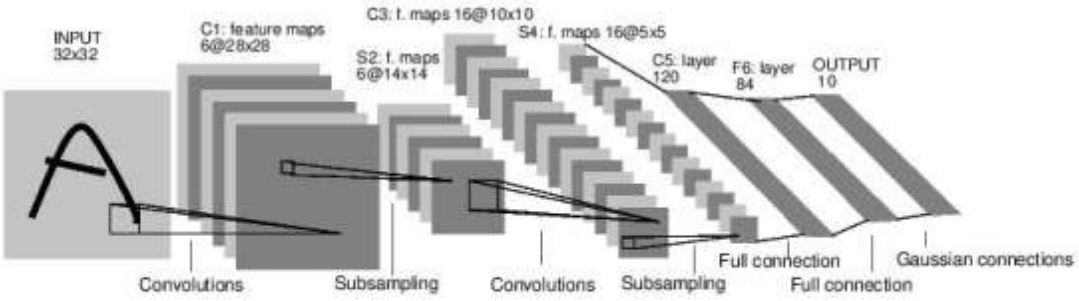




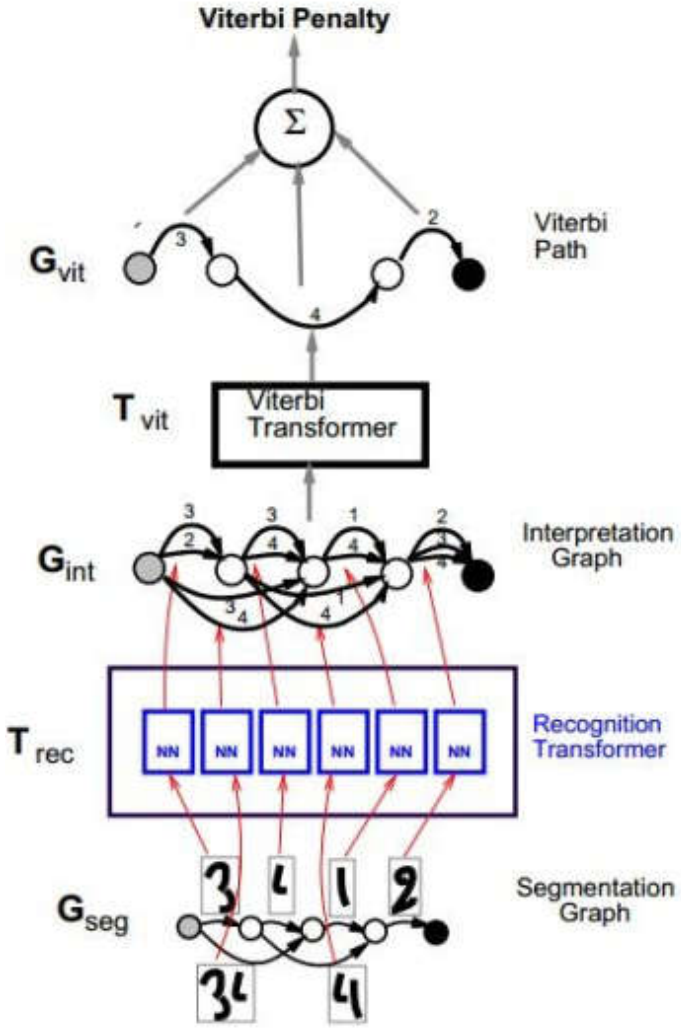
Slide from CS231n

LeCun, Bottou, Bengio, Haffner, 1998

- Gradient-based learning applied to document recognition
- Applied in postal mail service



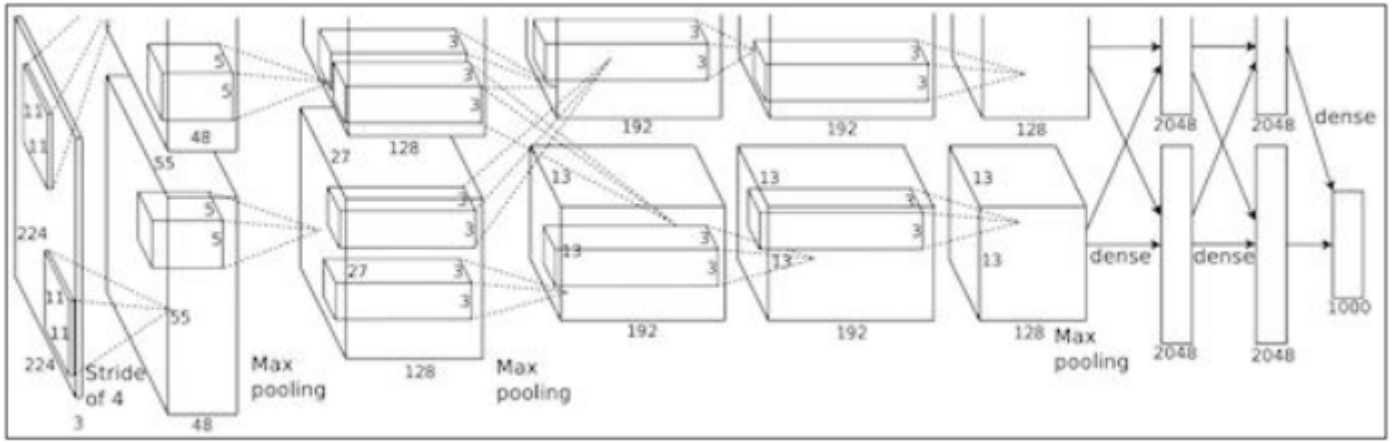
LeNet-5



Slide from CS231n

Krizhevsky, Sutskever, Hinton, 2012

- ImageNet Classification with Deep Convolutional Neural Networks
- 100 million images
- One thousand classes
- 60 million parameters

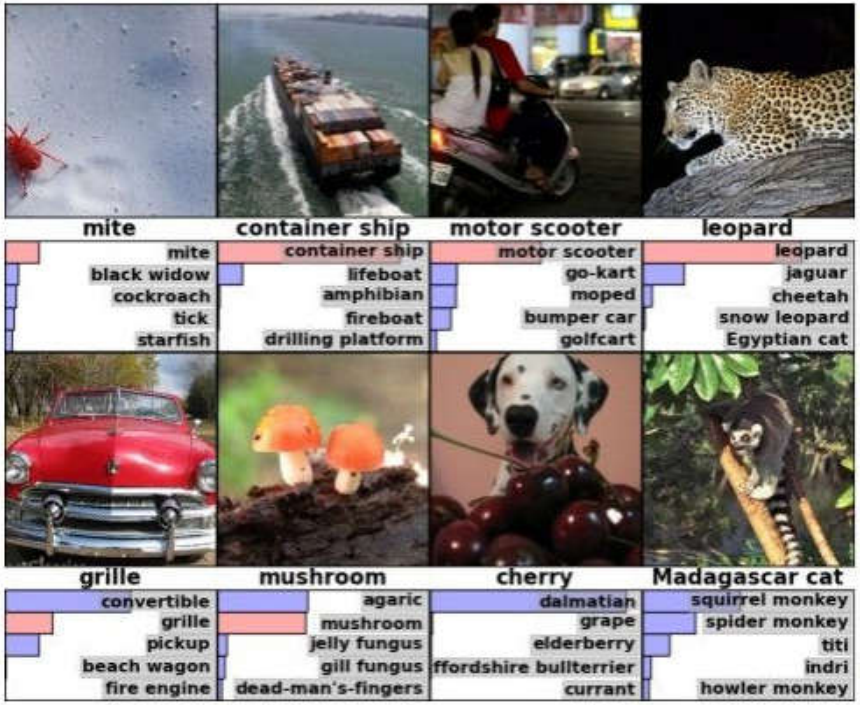


“AlexNet”

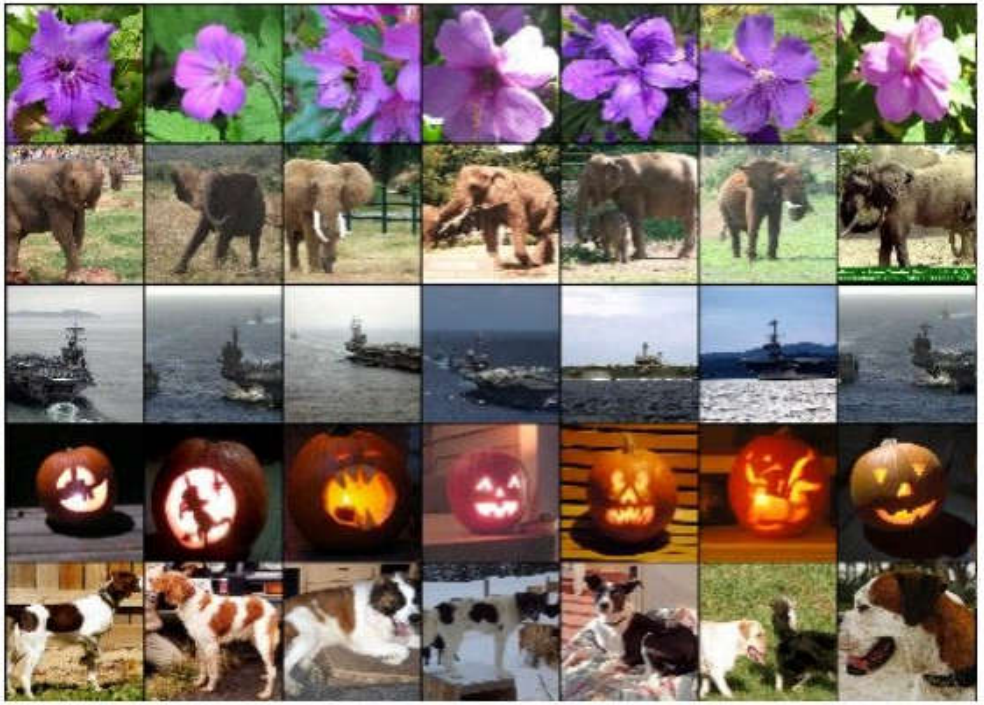
Slide from CS231n

ConvNets are Everywhere

Classification



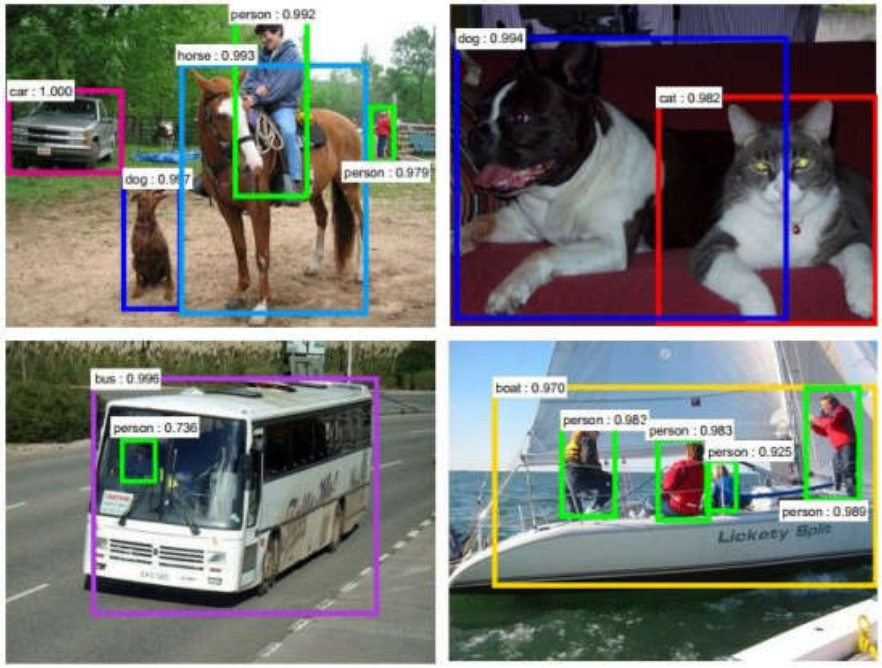
Retrieval



[Krizhevsky 2012]

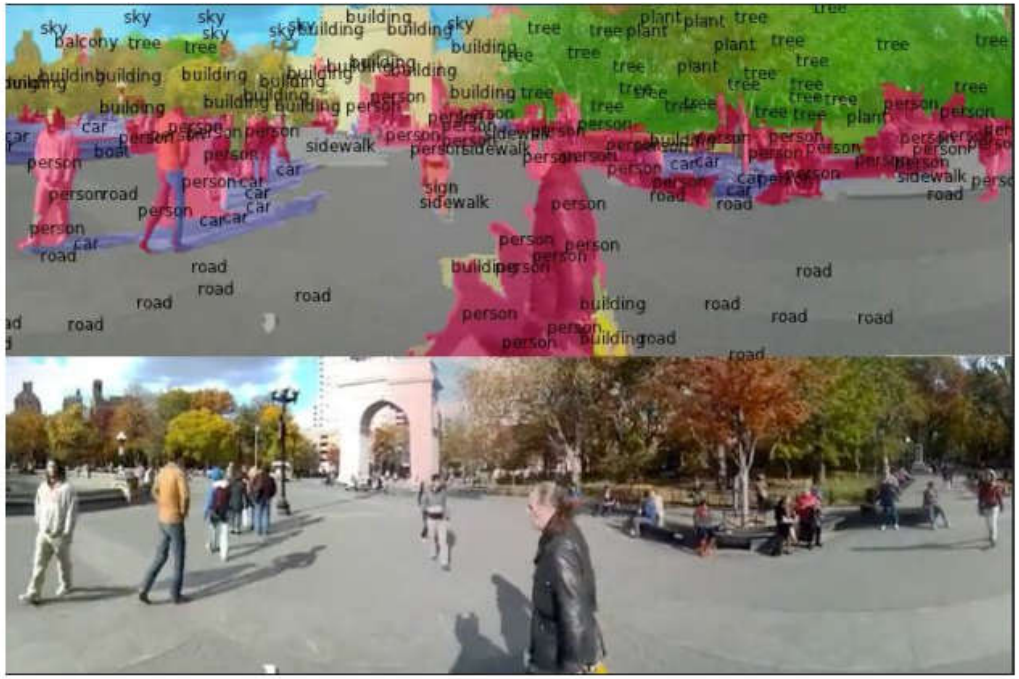
ConvNets are Everywhere

Detection



[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



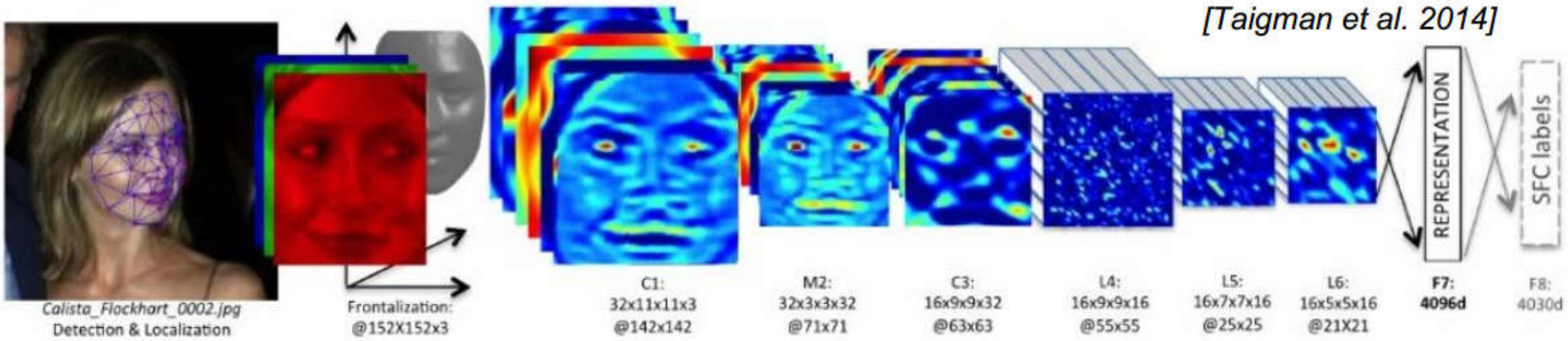
[Farabet et al., 2012]

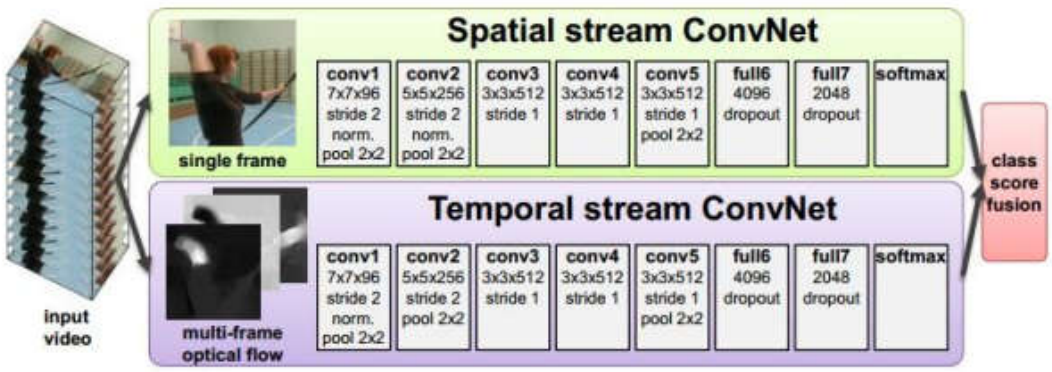
ConvNets are Everywhere

Self-driving Cars

NVIDIA Tegra X1

ConvNets are Everywhere





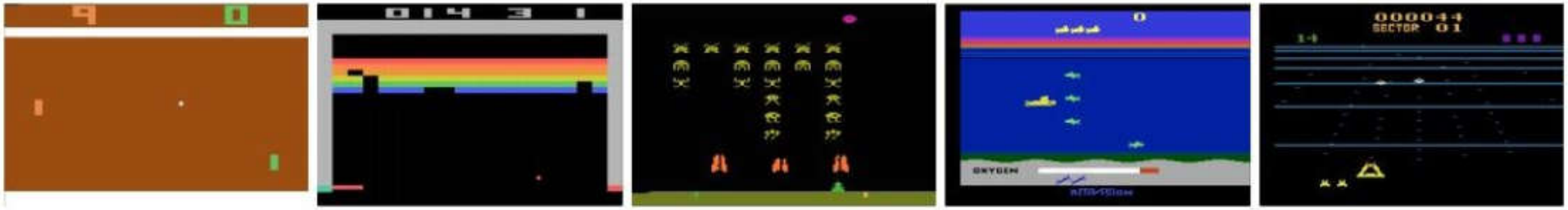
[Simonyan et al. 2014]

[Goodfellow2014]

Slide from CS231n

ConvNets are Everywhere

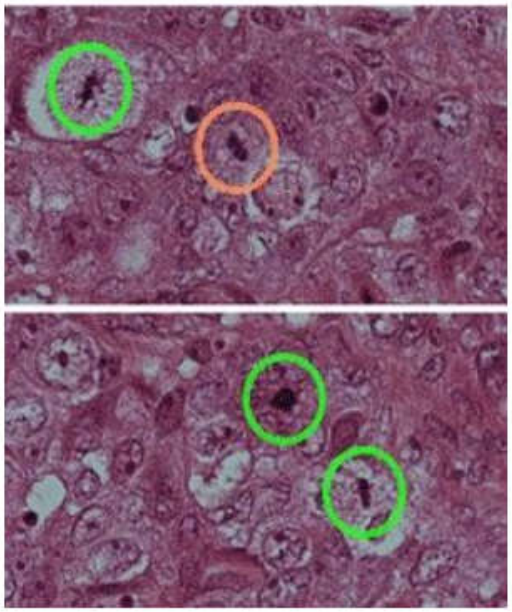
[Toshev, Szegedy 2014]



[Mnih 2013]

Slide from CS231n

ConvNets are Everywhere



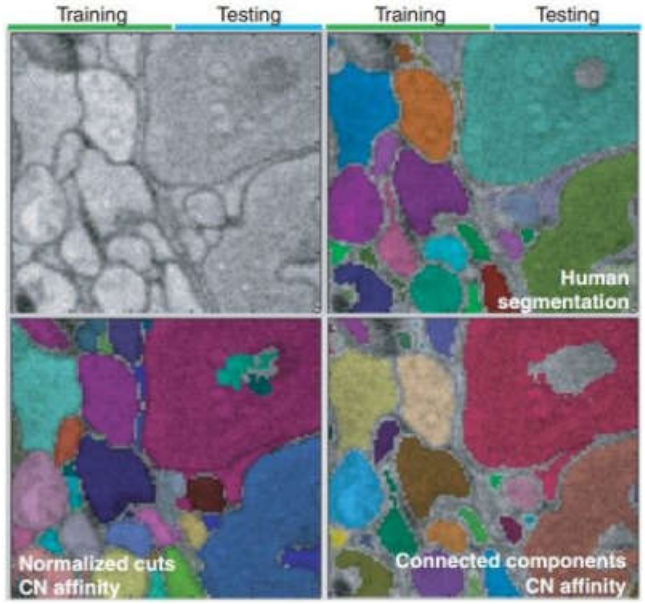
[Ciresan et al. 2013]

[Sermanet et al. 2011]

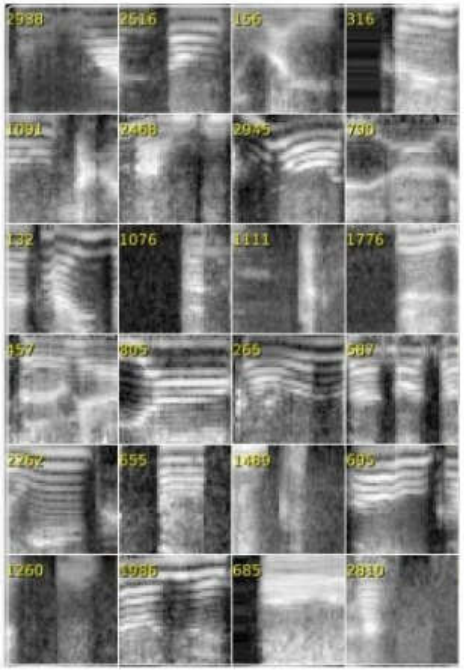
[Ciresan et al. 2013]

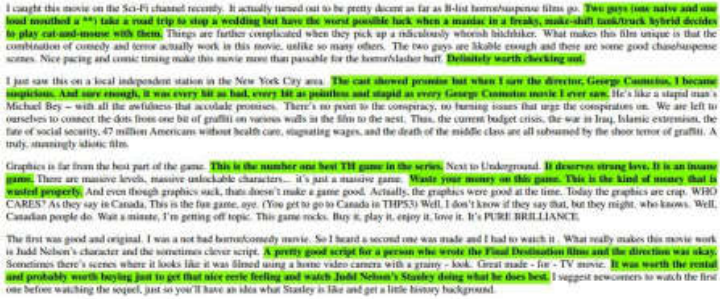
Slide from CS231n

ConvNets are Everywhere

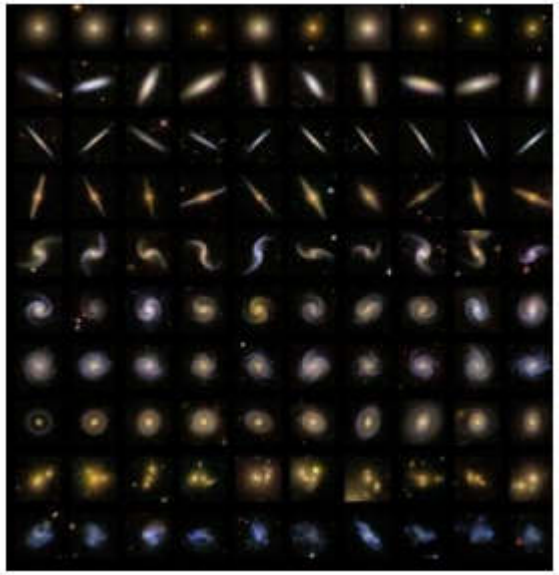


[Turaga et al., 2010]



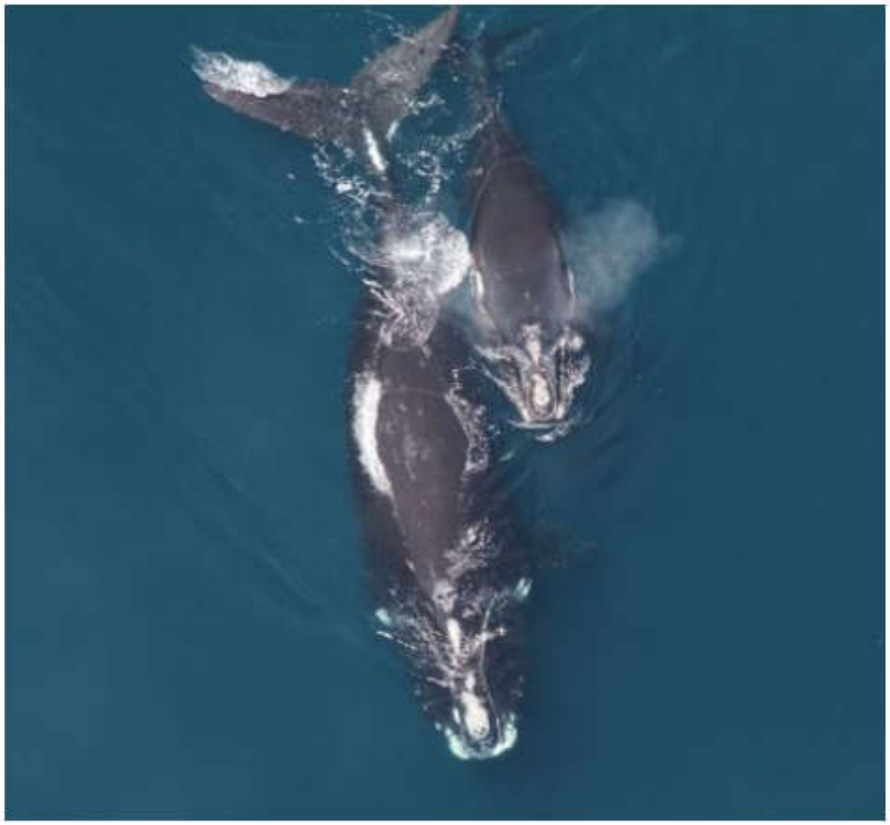


[Denil et al. 2014]



Slide from CS231n

ConvNets are Everywhere



Whale recognition, Kaggle Challenge

Mnih and Hinto, 2010

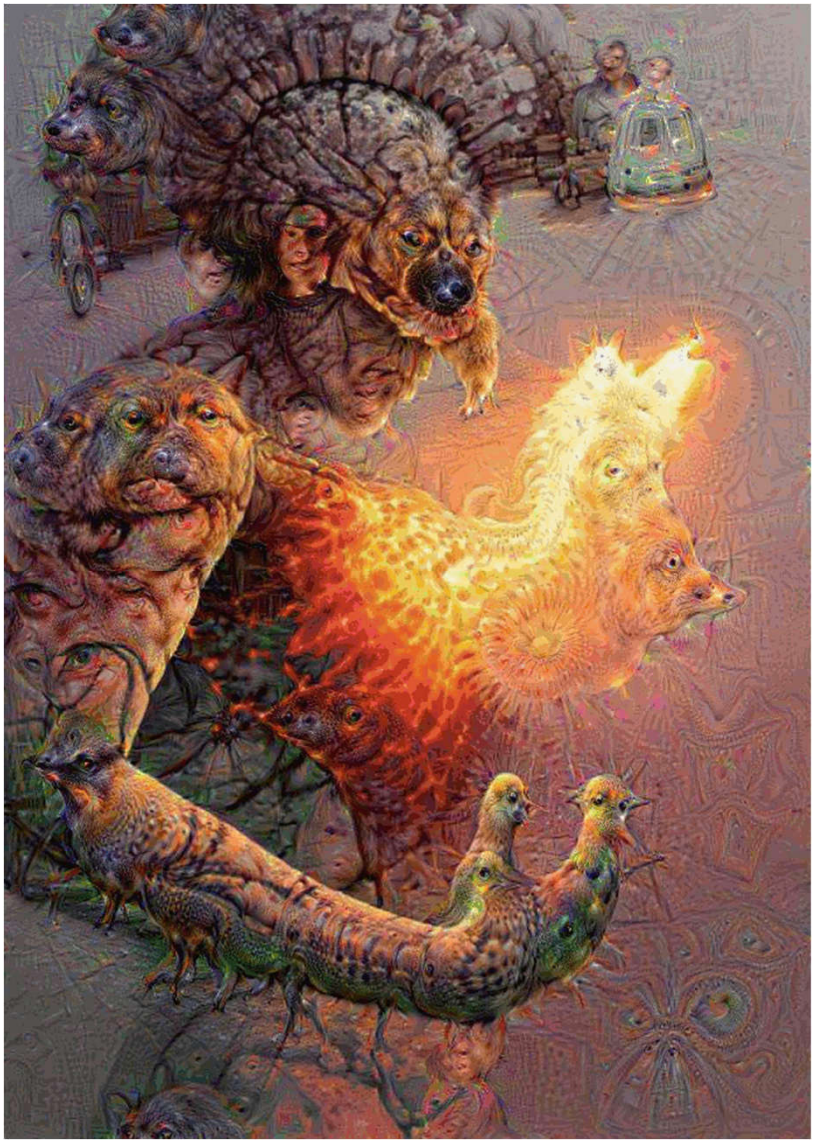
Slide from CS231n

ConvNets are Everywhere

[Vinyals et al., 2015]

Slide from CS231n

ConvNets are Everywhere



Slide from CS231n

Next Class

Thank you for your attention!