Lecture 6:

Training Neural Networks, Part 2

Administrative

A2 is out. It's meaty. It's due Feb 5 (next Friday)

You'll implement:
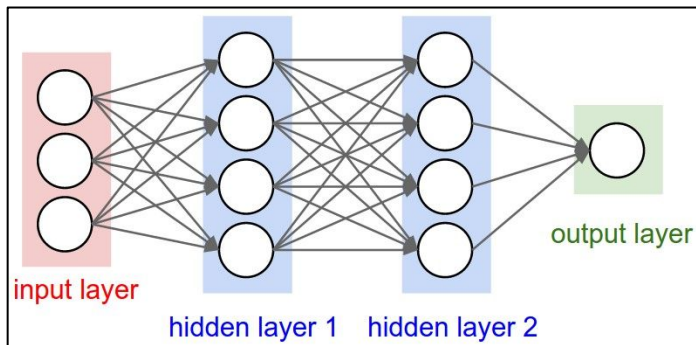> Neural Nets (with Layer Forward/Backward API)
> Batch Norm
> Dropout
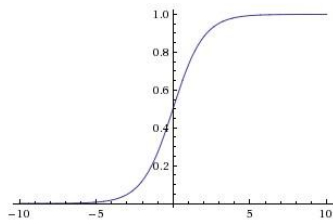> ConvNets

Mini-batch SGD

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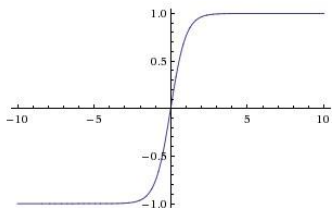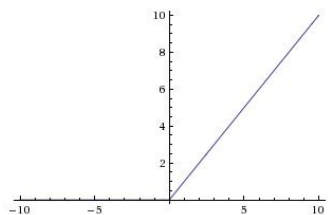4. **Update** the parameters using the gradient

Activation Functions

Leaky ReLU
max(0.1x, x)

Sigmoid

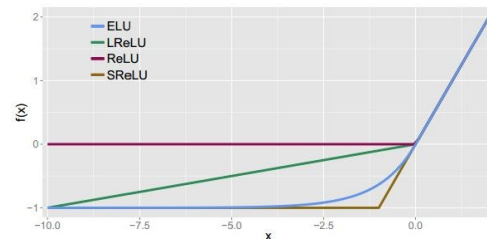$$\sigma(x) = 1/(1 + e^{-x})$$
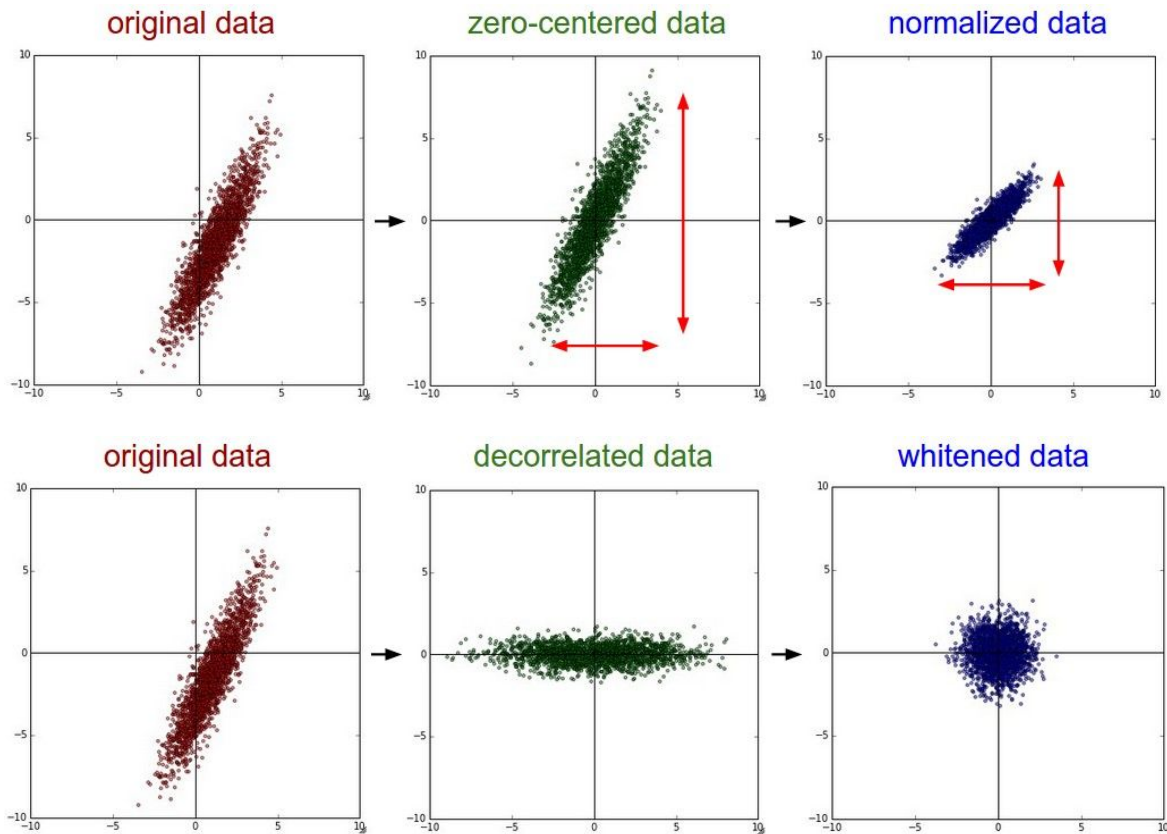

tanh   tanh(x)
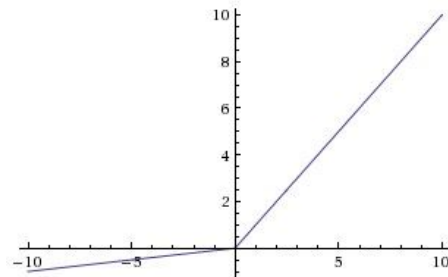
Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

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

ReLU   max(0,x)

Data Preprocessing

original data — zero-centered data — normalized data

original data — decorrelated data — whitened data

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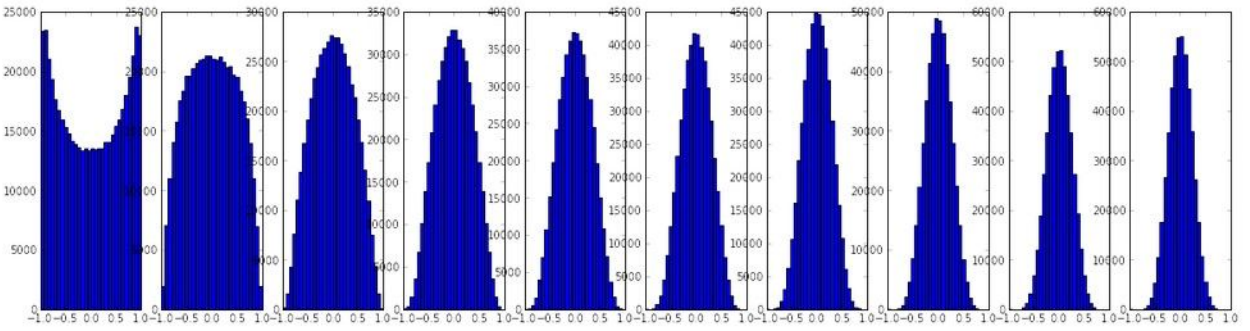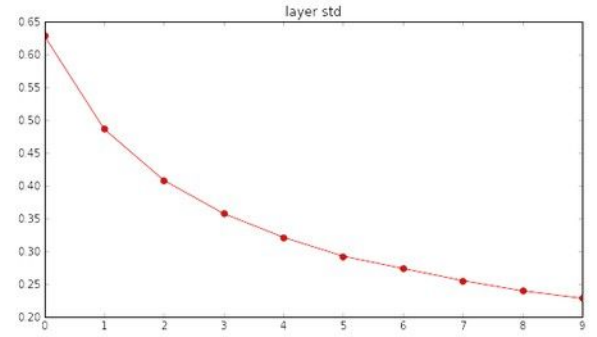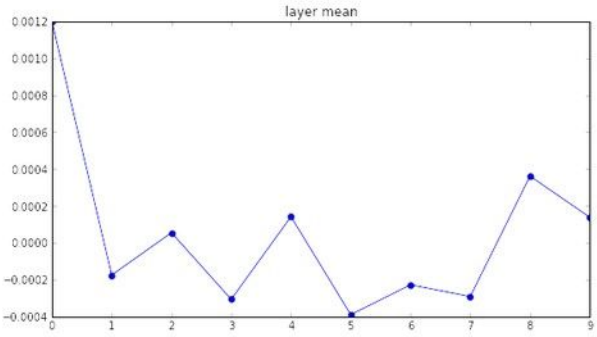
```python
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes linear activations)

Weight Initialization

Batch Normalization

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

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
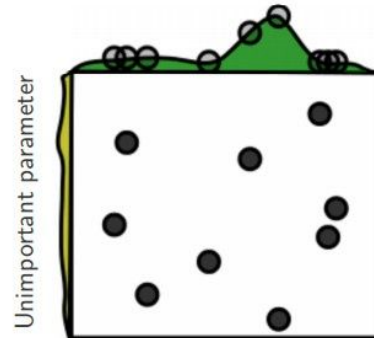```

Loss barely changing:
Learning rate is probably too low

# Cross-validation



Grid Layout

Unimportant parameter
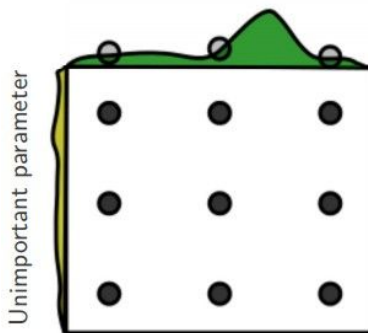
Important parameter

Random Layout

Unimportant parameter

Important parameter

# TODO

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

# Parameter Updates

Training a neural network, main loop:

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

Training a neural network, main loop:
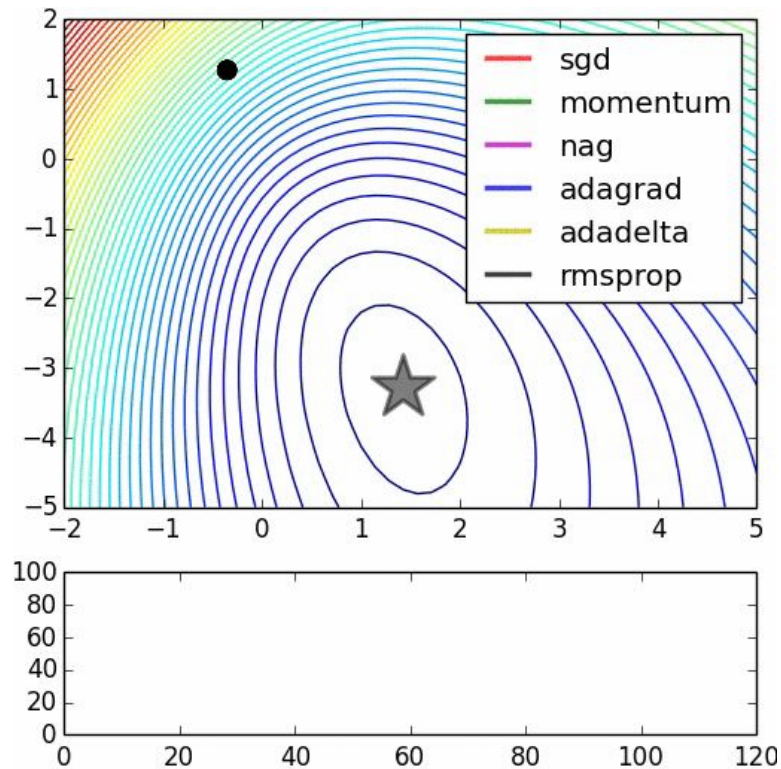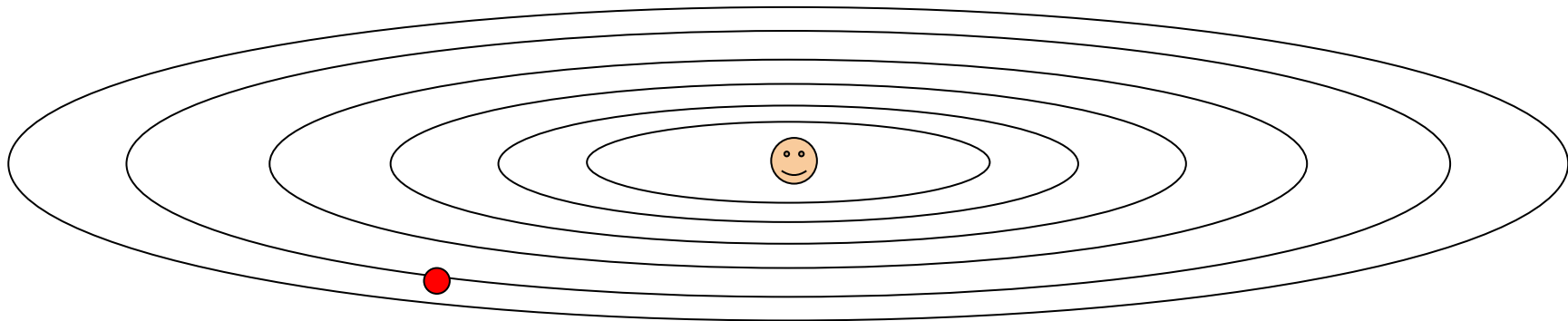
```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
```

simple gradient descent update
now: complicate.

Image credits: Alec Radford

Suppose loss function is steep vertically but shallow horizontally:

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

Suppose loss function is steep vertically but shallow horizontally:
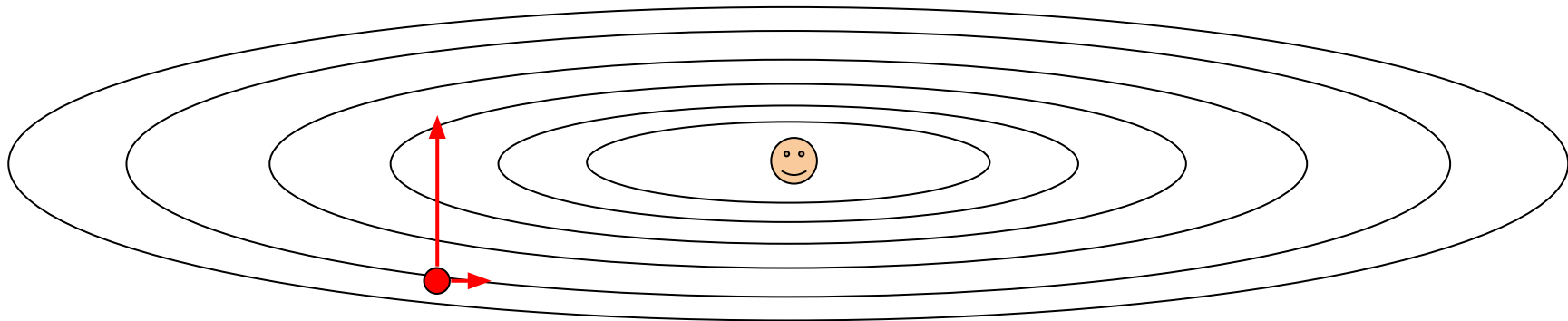
Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:
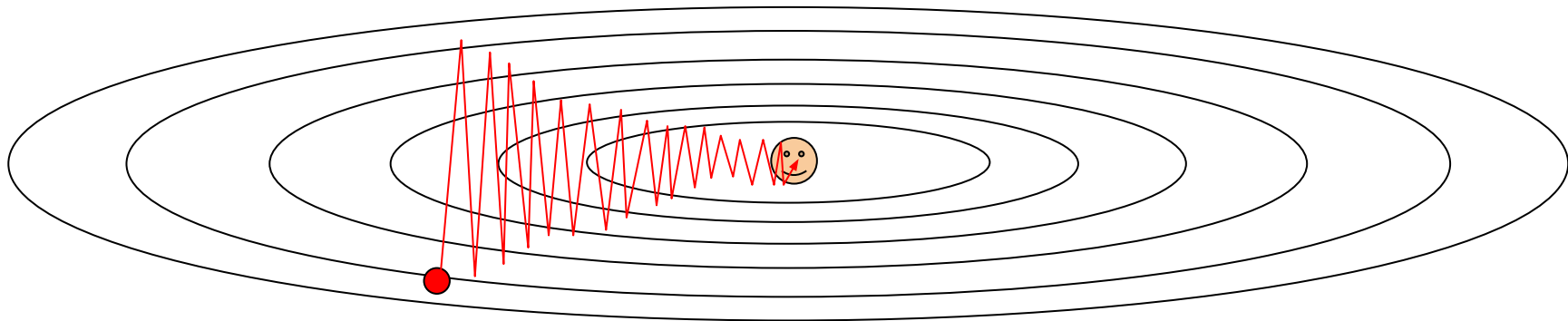
Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one

Momentum update

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

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

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

Momentum update

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

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```
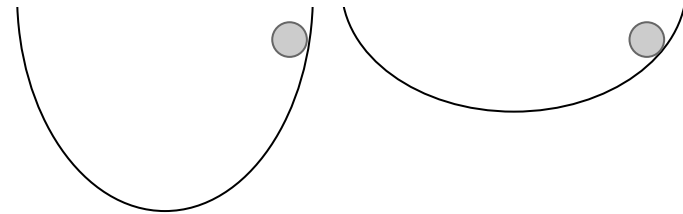
- Allows a velocity to "build up" along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD

vs

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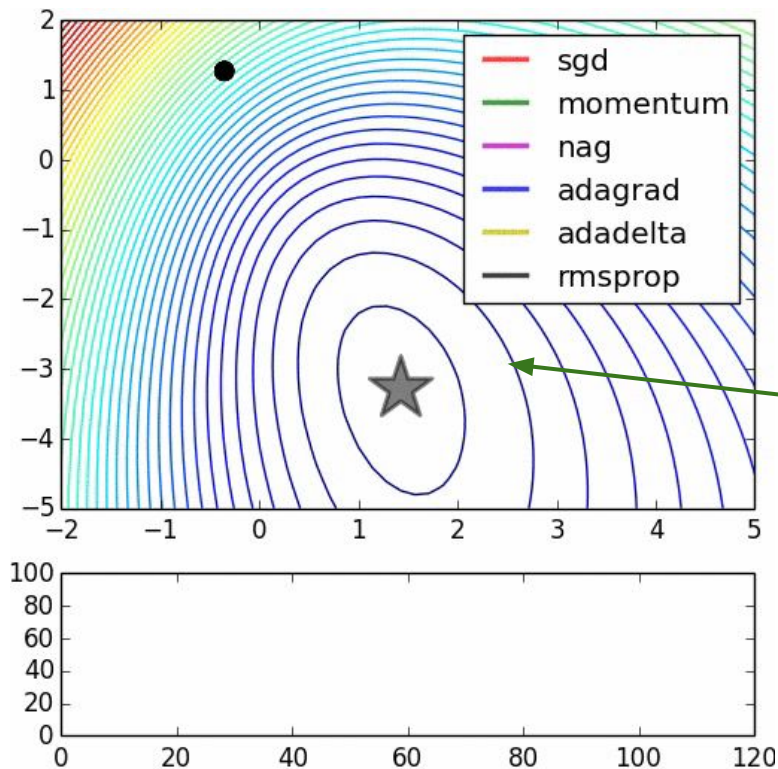
notice momentum overshooting the target, but overall getting to the minimum much faster.

Nesterov Momentum update

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

Ordinary momentum update:

momentum step

actual step

gradient step

Nesterov Momentum update

Momentum update

momentum step

actual step

gradient step

Nesterov momentum update

momentum step

actual step

"lookahead" gradient step (bit different than original)

Nesterov Momentum update

Momentum update

momentum
step

actual step

gradient
step

Nesterov momentum update

momentum
step

"lookahead" gradient
step (bit different than
original)

actual step
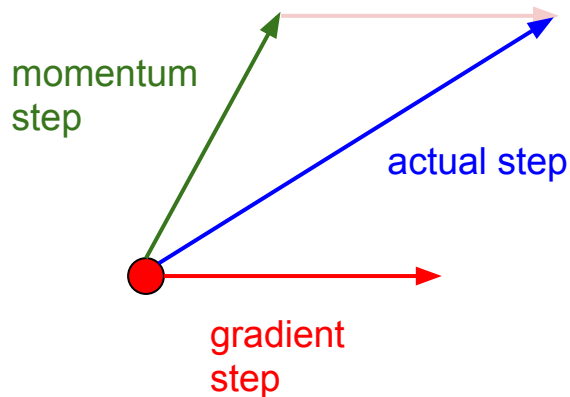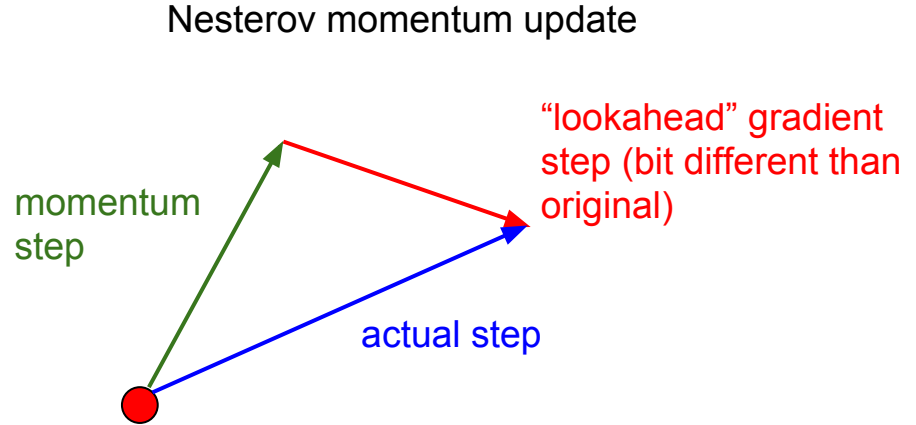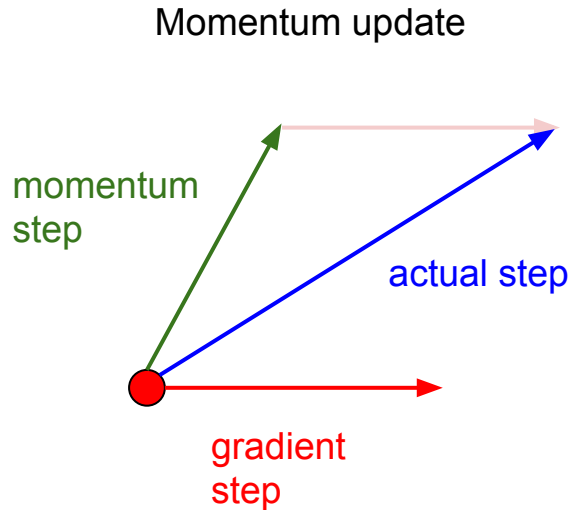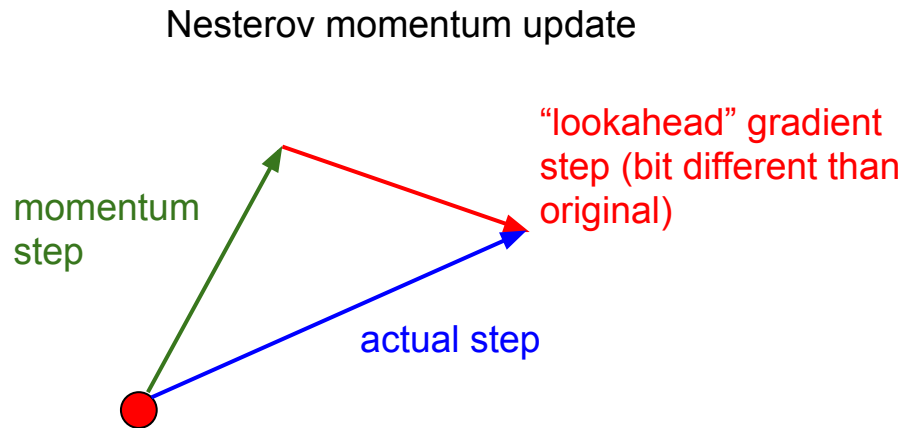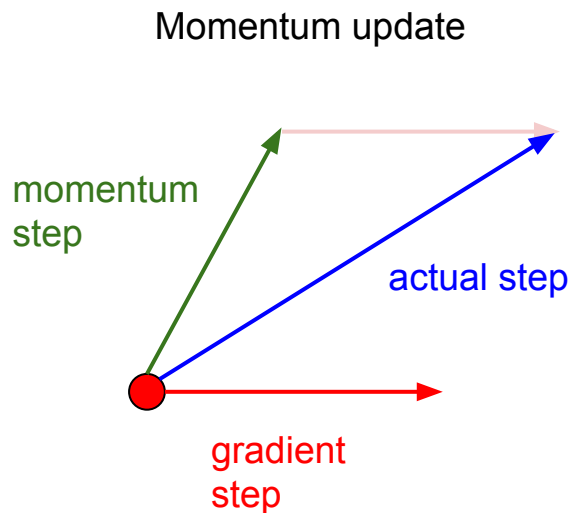
Nesterov: the only difference...

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

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

Nesterov Momentum update

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

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

Slightly inconvenient…
usually we have :

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

Nesterov Momentum update

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

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

Slightly inconvenient…
usually we have :

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

Variable transform and rearranging saves the day: $\boxed{\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}}$

Nesterov Momentum update

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

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

Slightly inconvenient…
usually we have :

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

Variable transform and rearranging saves the day:
$$\boxed{\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}}$$

Replace all thetas with phis, rearrange and obtain:

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

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

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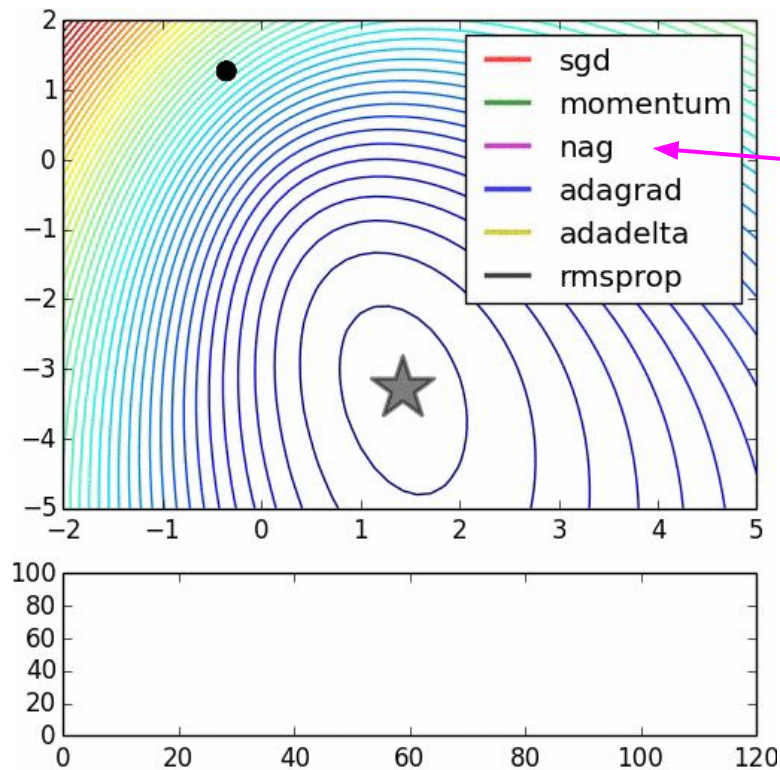
nag =
Nesterov
Accelerated
Gradient

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

AdaGrad update

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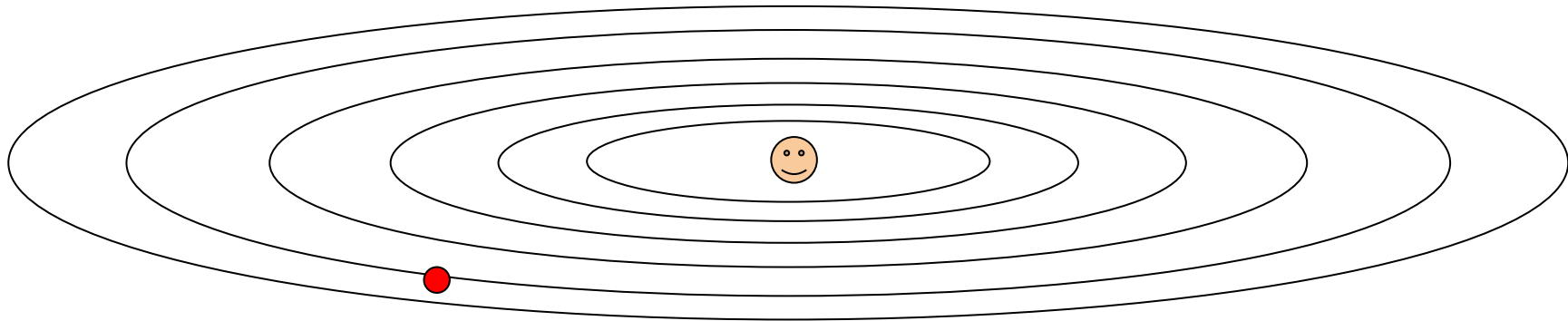

Q: What happens with AdaGrad?

AdaGrad update

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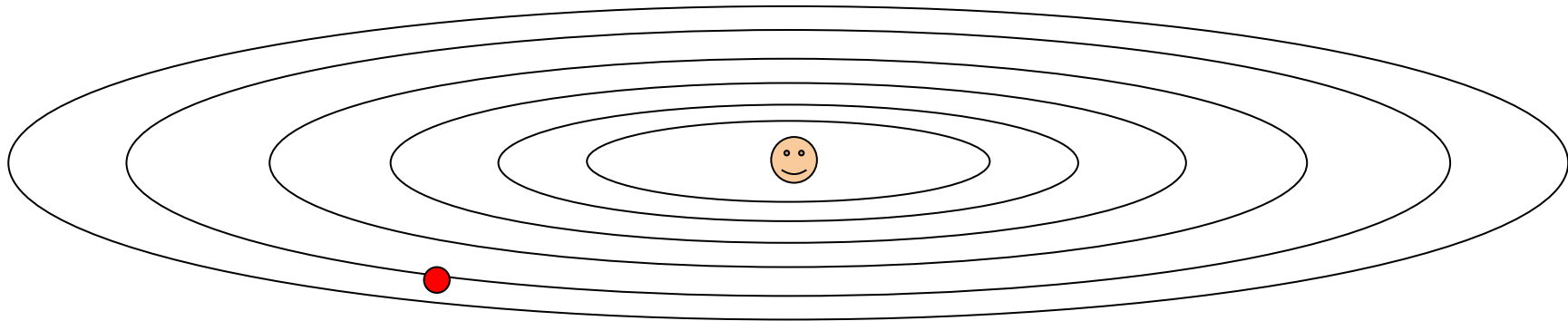

Q2: What happens to the step size over long time?

RMSProp update

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

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

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

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

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

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

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

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

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

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

Cited by several papers as:

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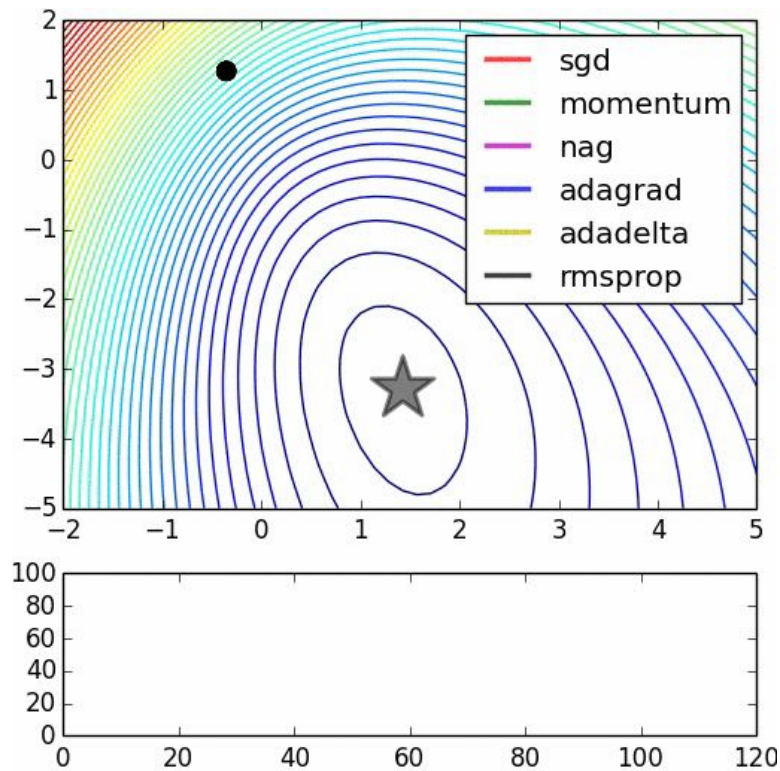
adagrad
rmsprop

Adam update

(incomplete, but close)

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

Adam update

(incomplete, but close)

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

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

(incomplete, but close)

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

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

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

Adam update

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

momentum

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

RMSProp-like

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

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.
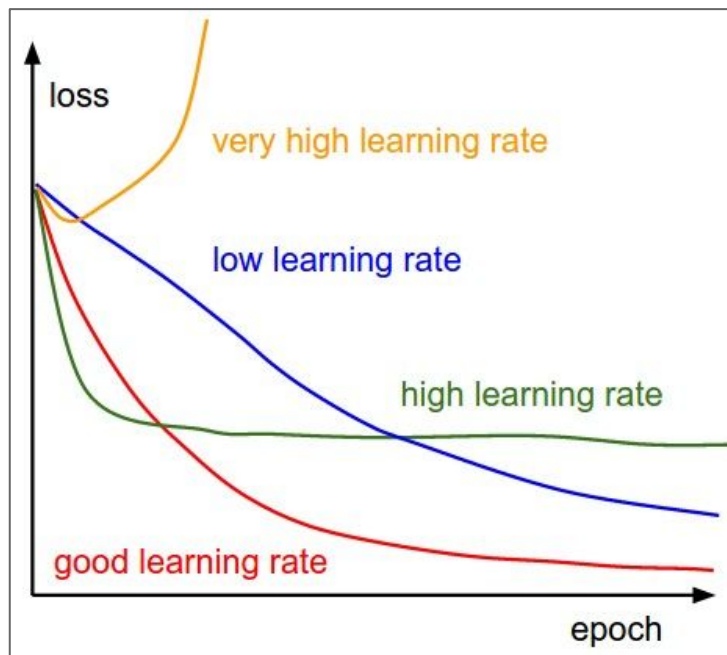
Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.
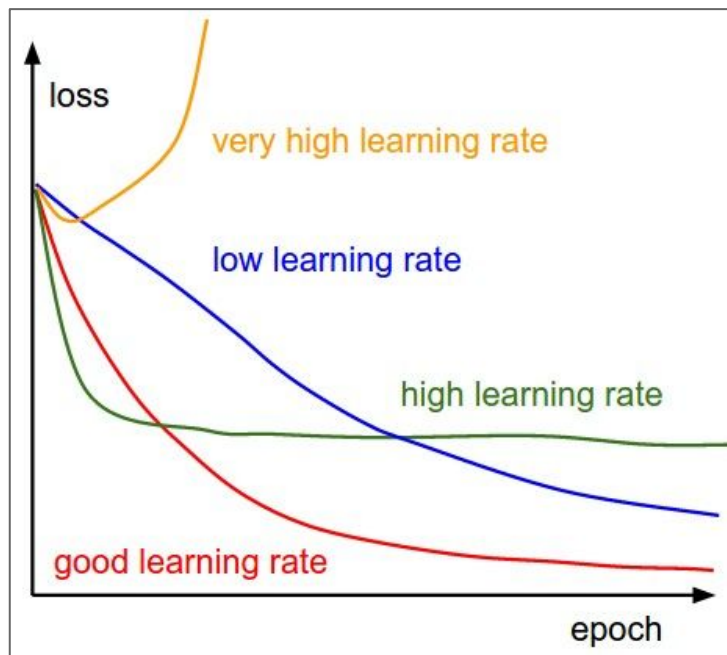
=> Learning rate decay over time!

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

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

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

Second order optimization methods

second-order Taylor expansion:

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

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

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

Q: what is nice about this update?

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)

Q2: why is this impractical for training Deep Neural Nets?

Second order optimization methods

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

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

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

L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

In practice:

- **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)

Evaluation:
Model Ensembles

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

Enjoy 2% extra performance

Fun Tips/Tricks:

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

Fun Tips/Tricks:

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

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

Regularization **(dropout)**

Regularization: **Dropout**
"randomly set some neurons to zero in the forward pass"
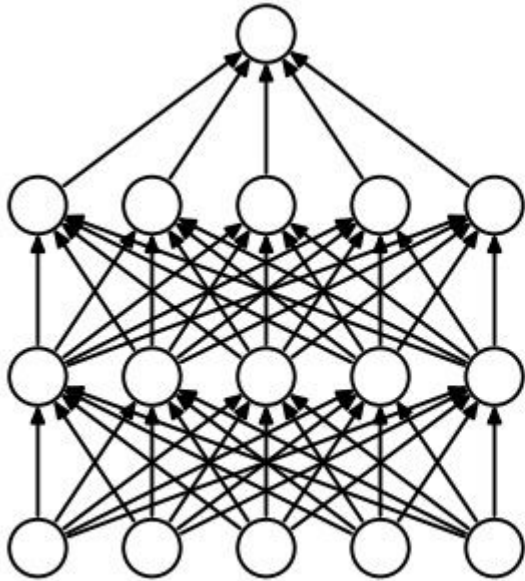
(a) Standard Neural Net
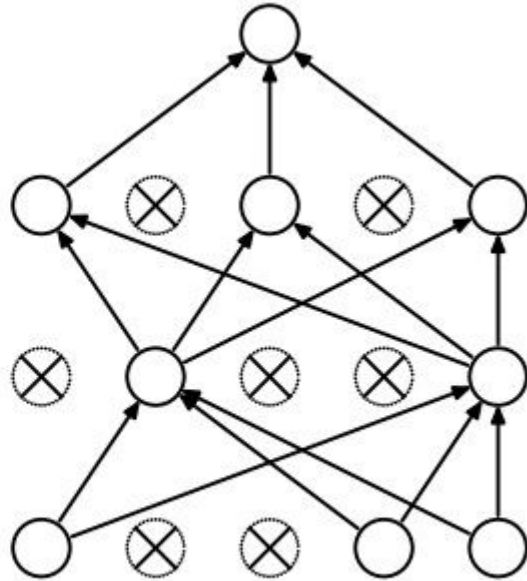
(b) After applying dropout.

[Srivastava et al., 2014]

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

def train_step(X):
  """ X contains the data """

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

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

Example forward pass with a 3-layer network using dropout
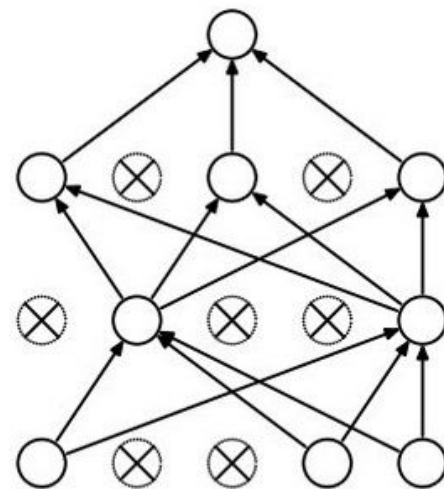
Waaaait a second…
How could this possibly be a good idea?
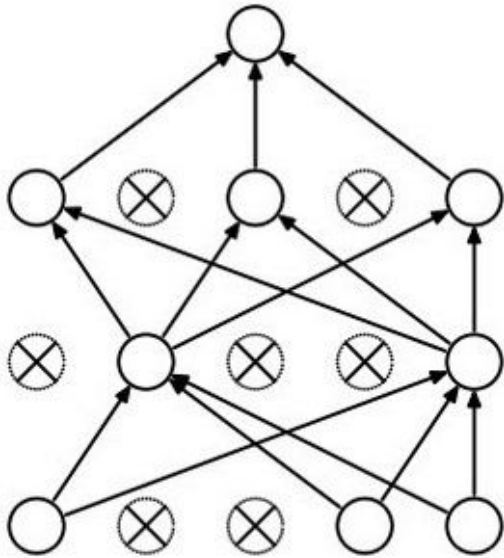
Waaaait a second…
How could this possibly be a good idea?
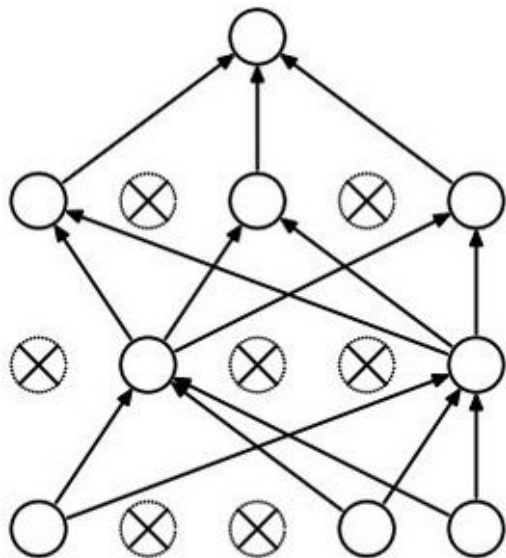
Forces the network to have a redundant representation.

has an ear ✗

has a tail

is furry ✗

has claws

mischievous look ✗

cat score
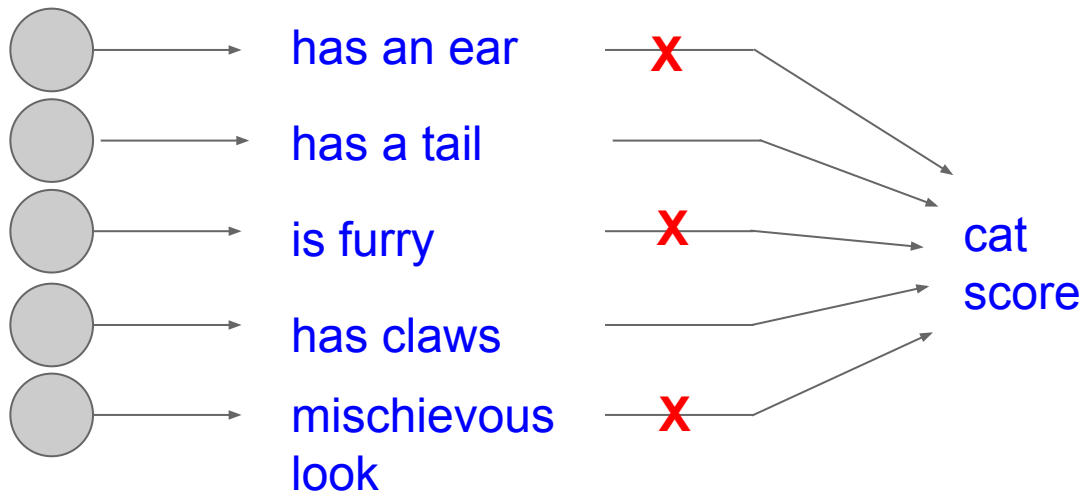
Waaaait a second…
How could this possibly be a good idea?
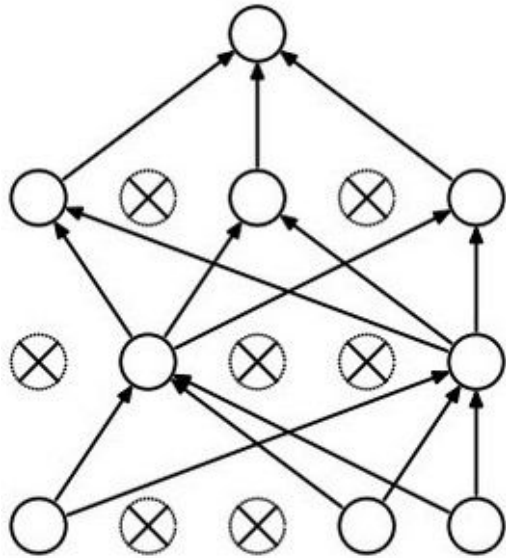
Another interpretation:

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

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

At test time….
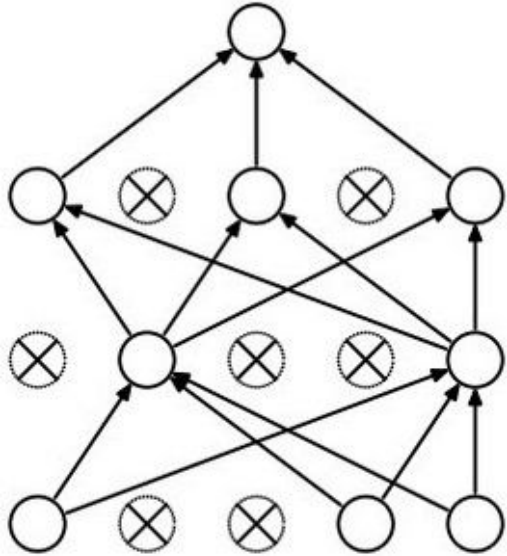
Ideally:
want to integrate out all the noise

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

At test time….

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

Leave all input neurons turned on (no dropout).
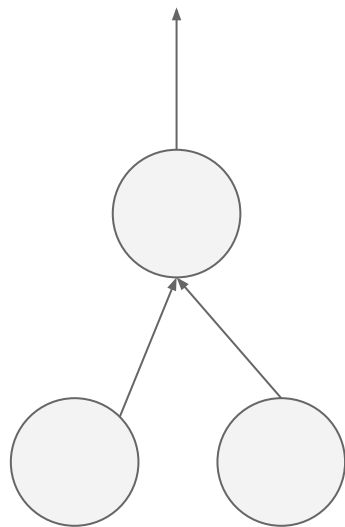
(this can be shown to be an approximation to evaluating the whole ensemble)

At test time….

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

Leave all input neurons turned on (no dropout).
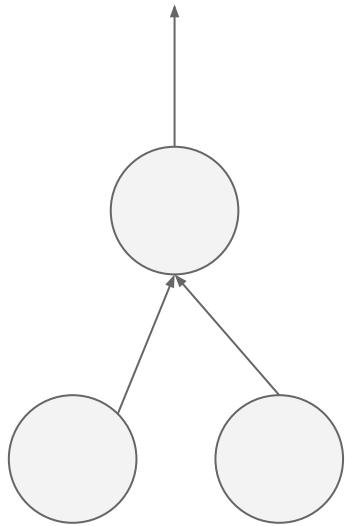
Q: Suppose that with all inputs present at test time the output of this neuron is x.

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

At test time….

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

Leave all input neurons turned on (no dropout).
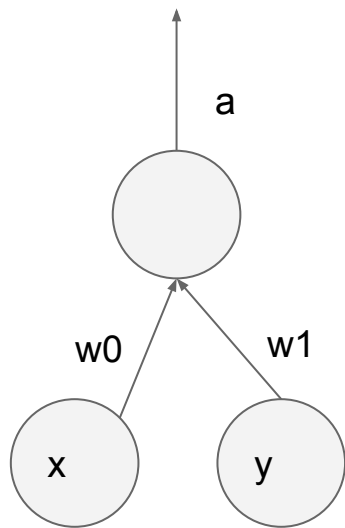
during test: **a = w0*x + w1*y**
during train:
E[a] = ¼ * (w0*0 + w1*0
             w0*0 + w1*y
             w0*x + w1*0
             w0*x + w1*y)
     = ¼ * (2 w0*x + 2 w1*y)
     = ½ * (w0*x + w1*y)

At test time….

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

Leave all input neurons turned on (no dropout).
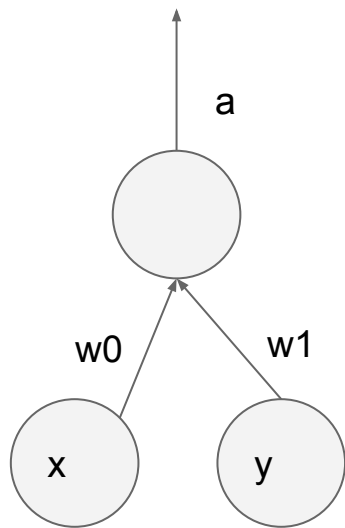
during test: **a = w0*x + w1*y**
during train:
E[a] = ¼ * (w0*0 + w1*0
           w0*0 + w1*y
           w0*x + w1*0
           w0*x + w1*y)
  = ¼ * (2 w0*x + 2 w1*y)
  = ½ * (w0*x + w1*y)

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

We can do something approximate analytically

```python
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:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout Summary

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

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

def train_step(X):
  """ X contains the data """

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

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

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

drop in forward pass

scale at test time

More common: "Inverted dropout"

```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!

\<fun story time\>
(Deep Learning Summer School 2012)

Gradient Checking

(see class notes...)

fun guaranteed.

Convolutional Neural Networks

[LeNet-5, LeCun 1980]

A bit of history:

Hubel & Wiesel,
1959
RECEPTIVE FIELDS OF SINGLE
NEURONES IN
THE CAT'S STRIATE CORTEX
1962
RECEPTIVE FIELDS, BINOCULAR
INTERACTION
AND FUNCTIONAL ARCHITECTURE IN
THE CAT'S VISUAL CORTEX
1968...
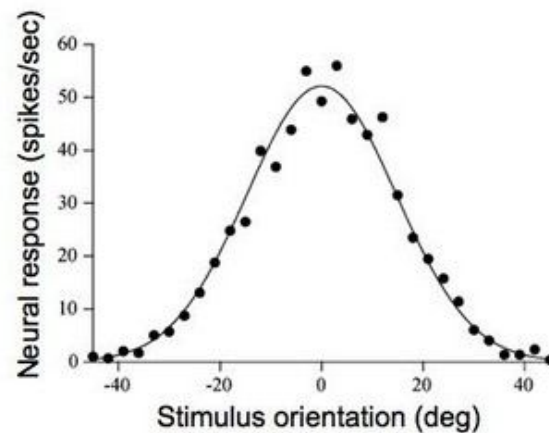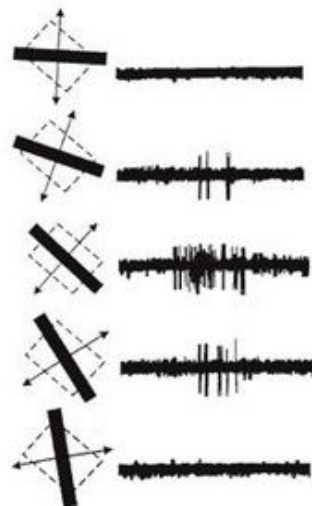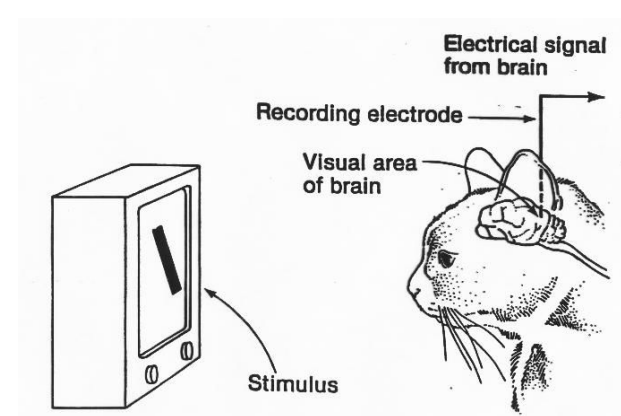
Video time https://youtu.be/8VdFf3egwfg?t=1m10s

A bit of history

Topographical mapping in the cortex:
nearby cells in cortex represented
nearby regions in the visual field
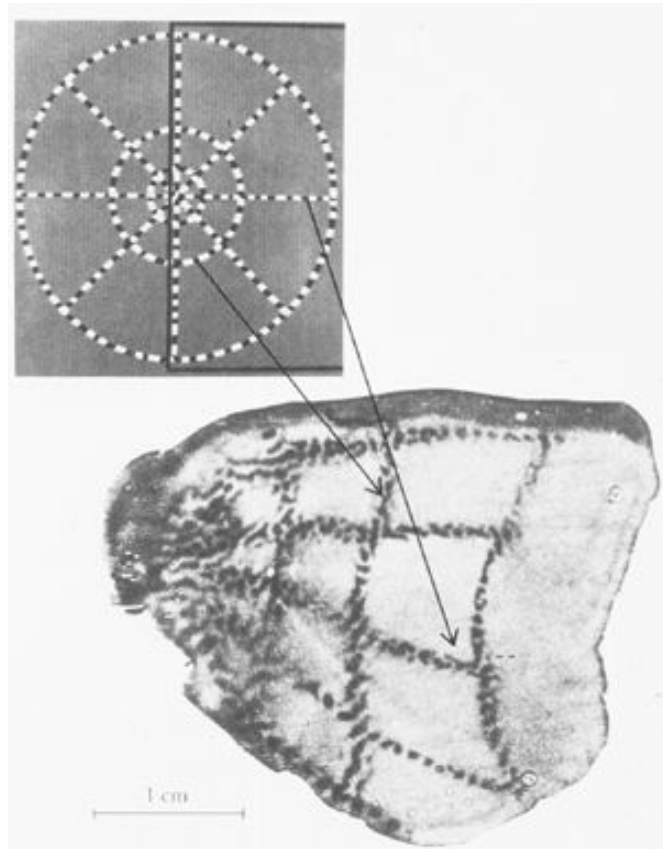
Hierarchical organization
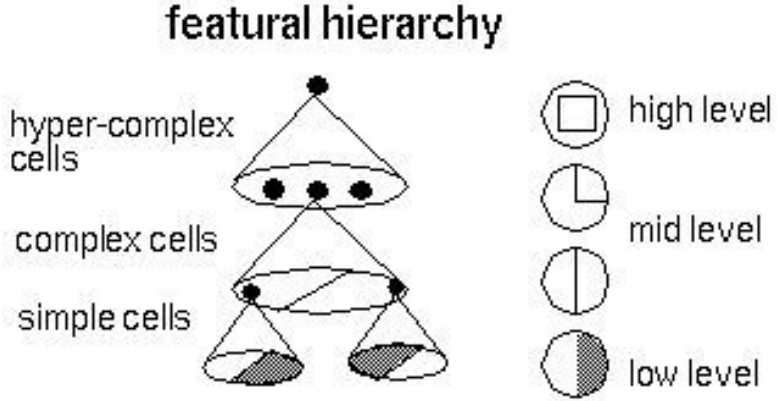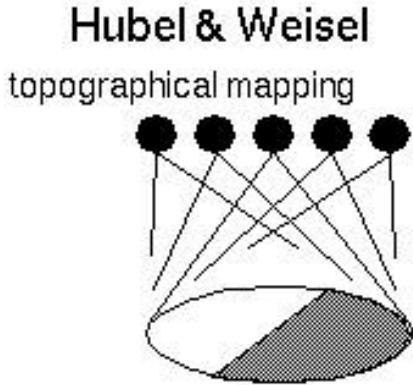
A bit of history:

Neurocognitron
[Fukushima 1980]
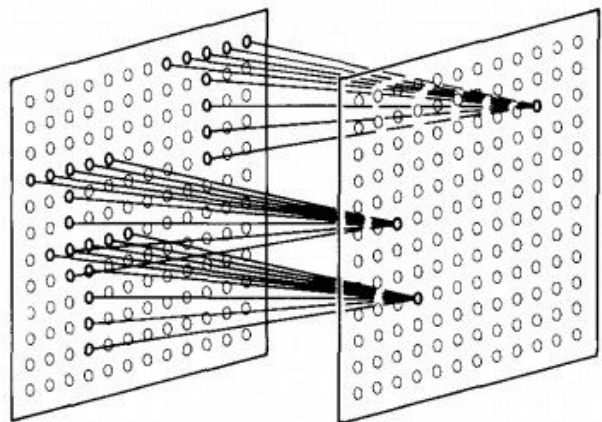
"sandwich" architecture (SCSCSC…)
simple cells: modifiable parameters
complex cells: perform pooling
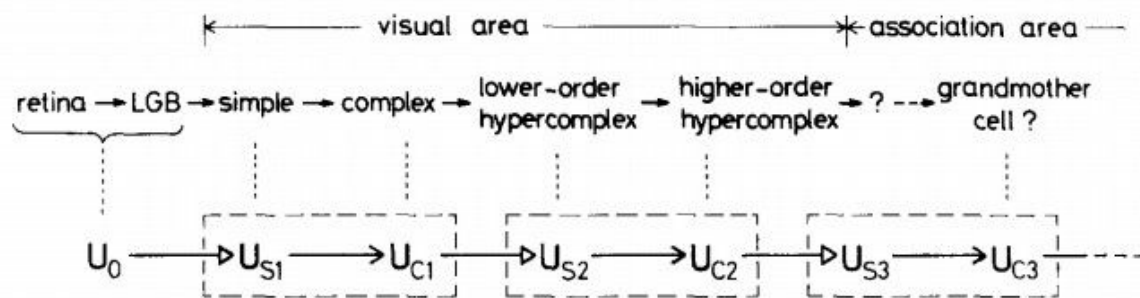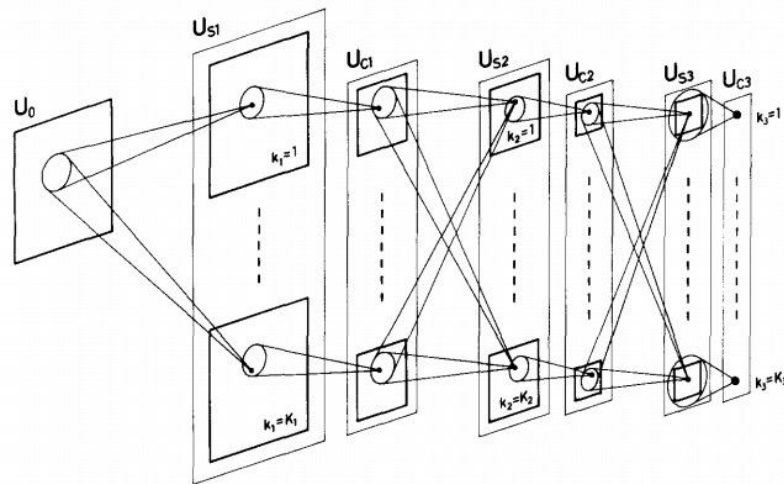
A bit of history:
Gradient-based learning applied to document recognition
[LeCun, Bottou, Bengio, Haffner 1998]
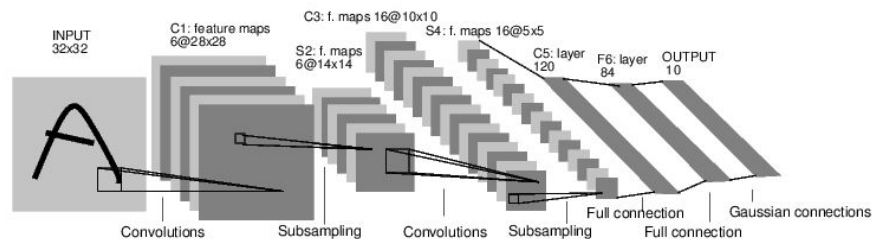
LeNet-5
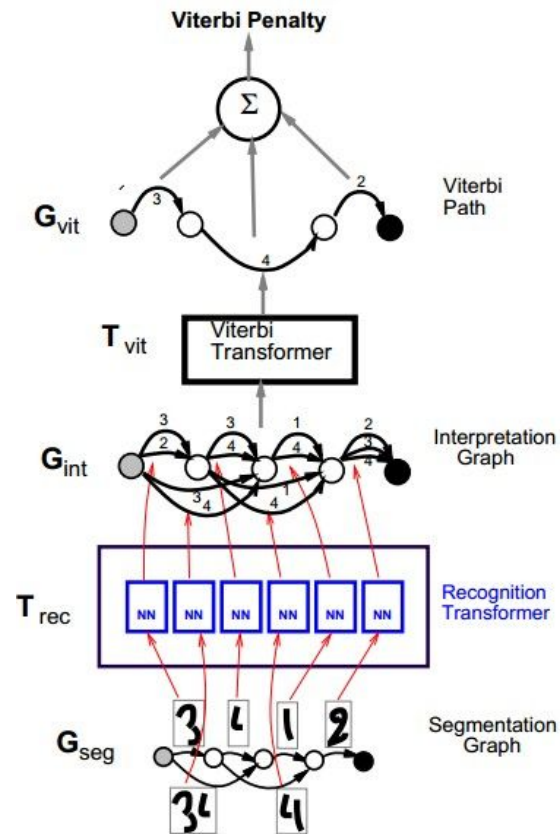
A bit of history:
ImageNet Classification with Deep Convolutional Neural Networks
[Krizhevsky, Sutskever, Hinton, 2012]
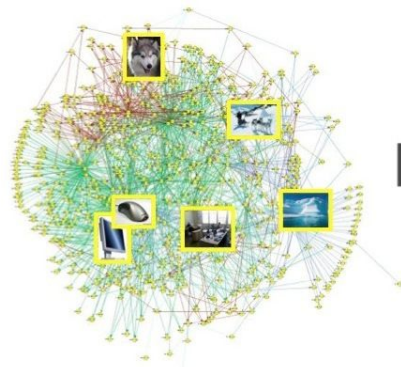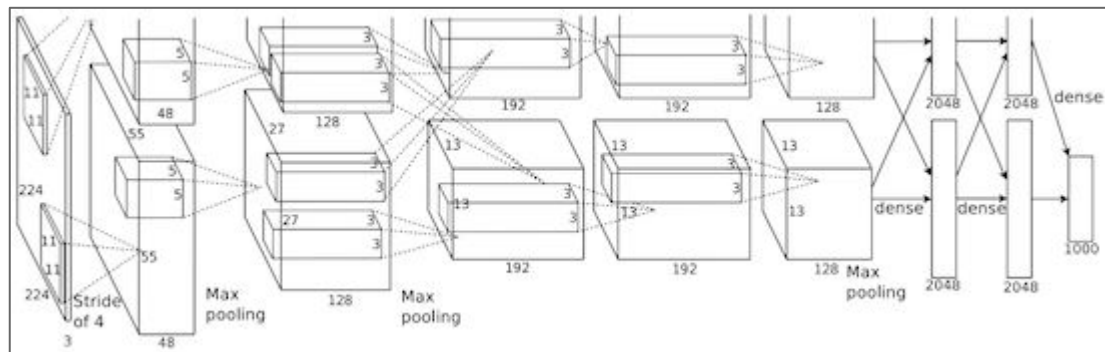
"AlexNet"

Fast-forward to today: ConvNets are everywhere

Classification
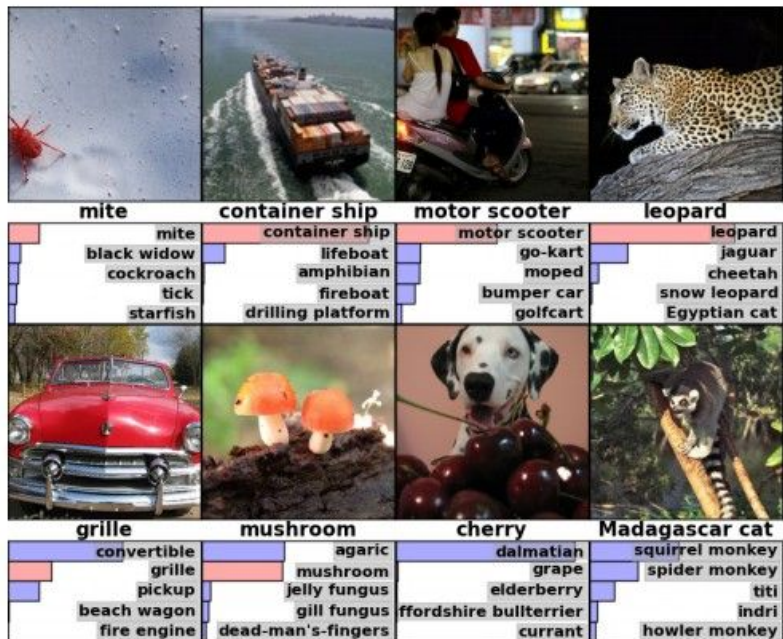
Retrieval
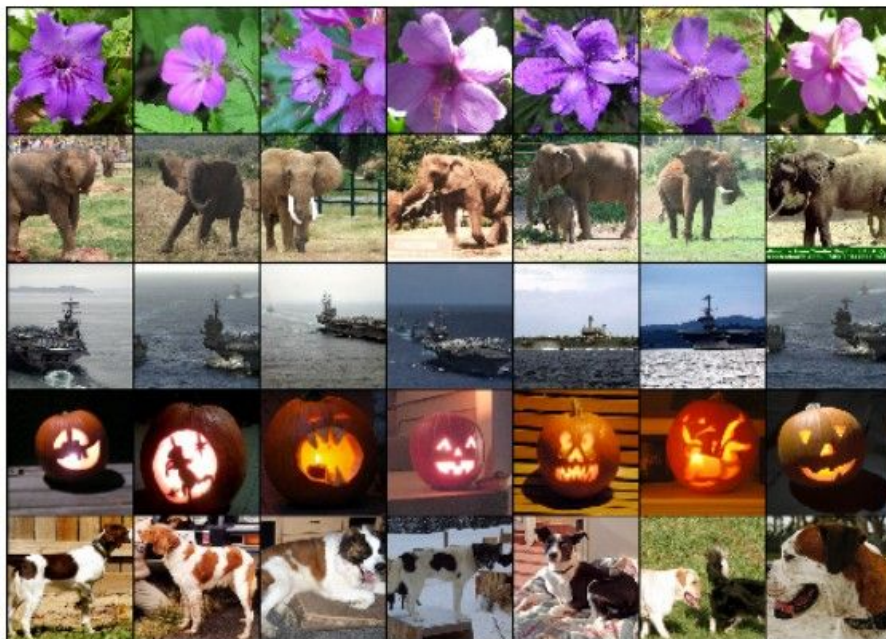
[Krizhevsky 2012]

Fast-forward to today: ConvNets are everywhere

Detection
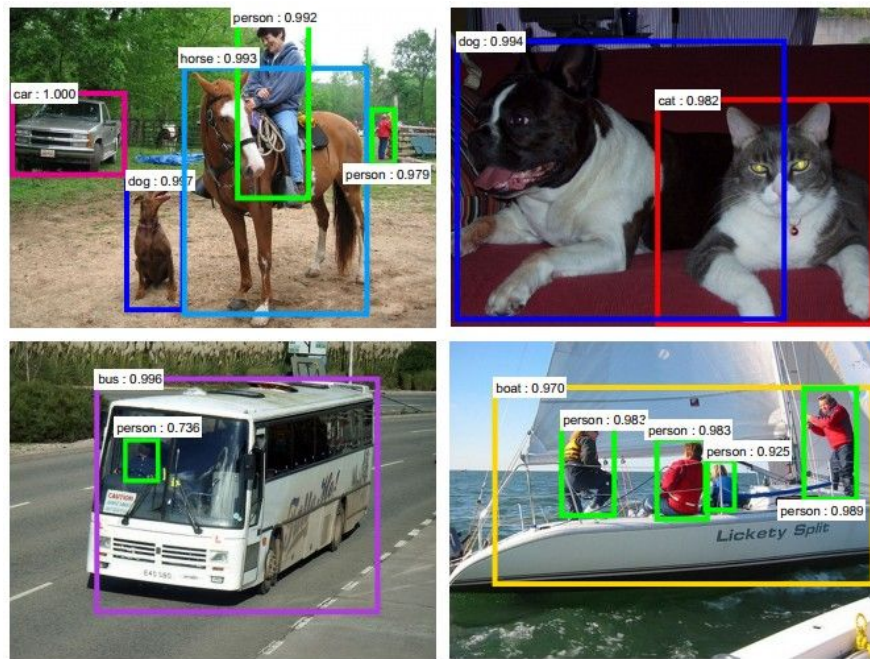
Segmentation

[Faster R-CNN: Ren, He, Girshick, Sun 2015]
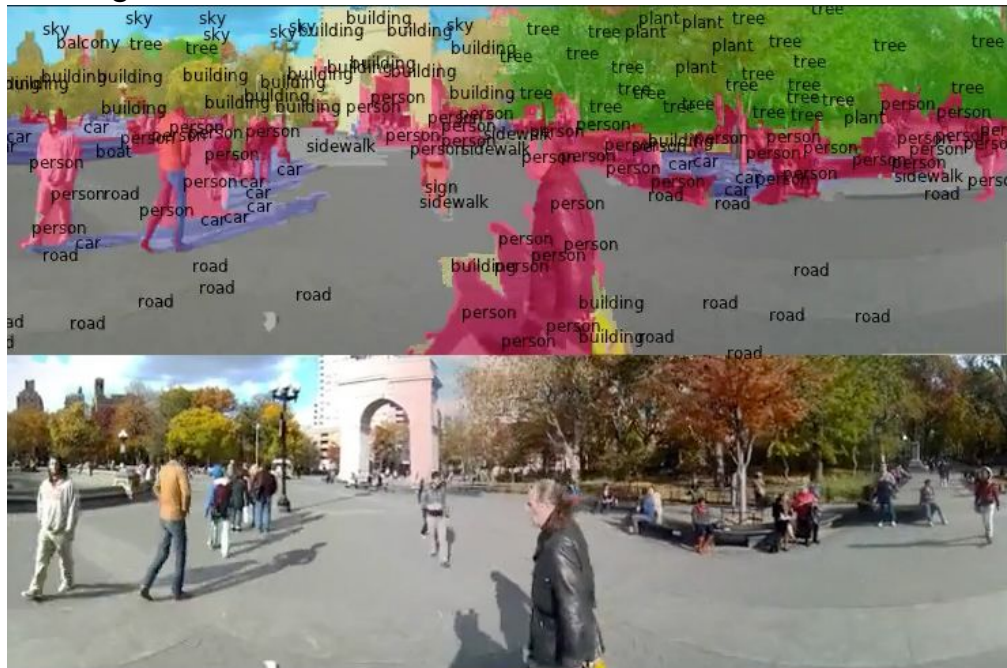
[Farabet et al., 2012]

Fast-forward to today: ConvNets are everywhere

self-driving cars

NVIDIA Tegra X1

Fast-forward to today: ConvNets are everywhere

[Taigman et al. 2014]

Calista_Flockhart_0002.jpg
Detection & Localization

Frontalization:
@152X152x3

C1:
32x11x11x3
@142x142

M2:
32x3x3x32
@71x71

C3:
16x9x9x32
@63x63

L4:
16x9x9x16
@55x55

L5:
16x7x7x16
@25x25

L6:
16x5x5x16
@21X21

F7:
4096d

F8:
4030d

REPRESENTATION

SFC labels
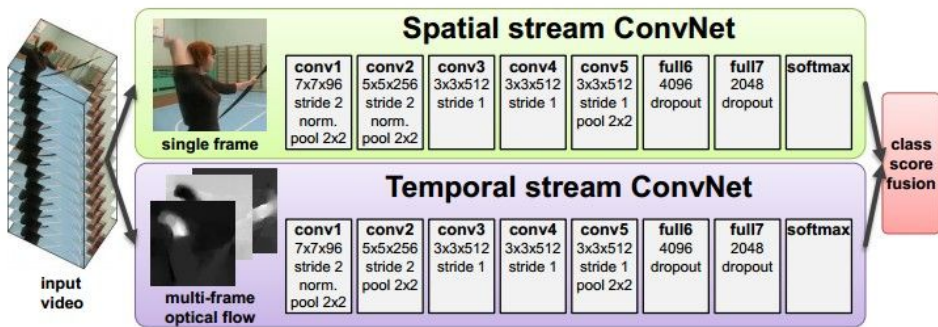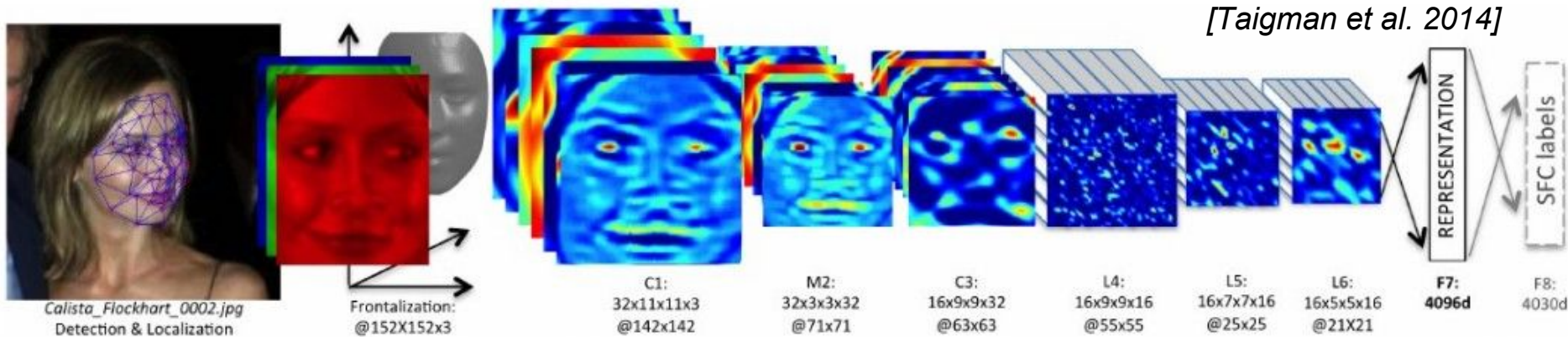
[Simonyan et al. 2014]

[Goodfellow 2014]

Fast-forward to today: ConvNets are everywhere

[Toshev, Szegedy 2014]

[Mnih 2013]

Fast-forward to today: ConvNets are everywhere
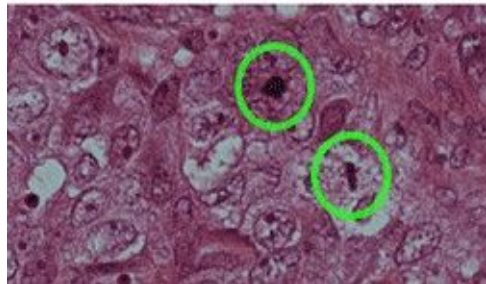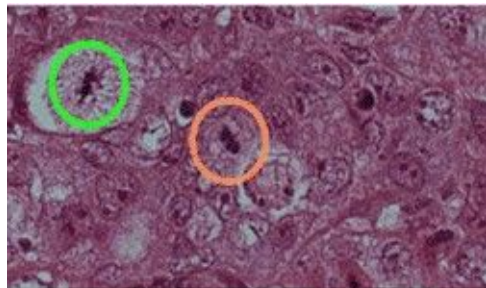
[Ciresan et al. 2013]

[Sermanet et al. 2011]
[Ciresan et al.]

Fast-forward to today: ConvNets are everywhere

[Turaga et al., 2010]

[Denil et al. 2014]

Whale recognition, Kaggle Challenge   *Mnih and Hinton, 2010*

Image Captioning

[Vinyals et al., 2015]
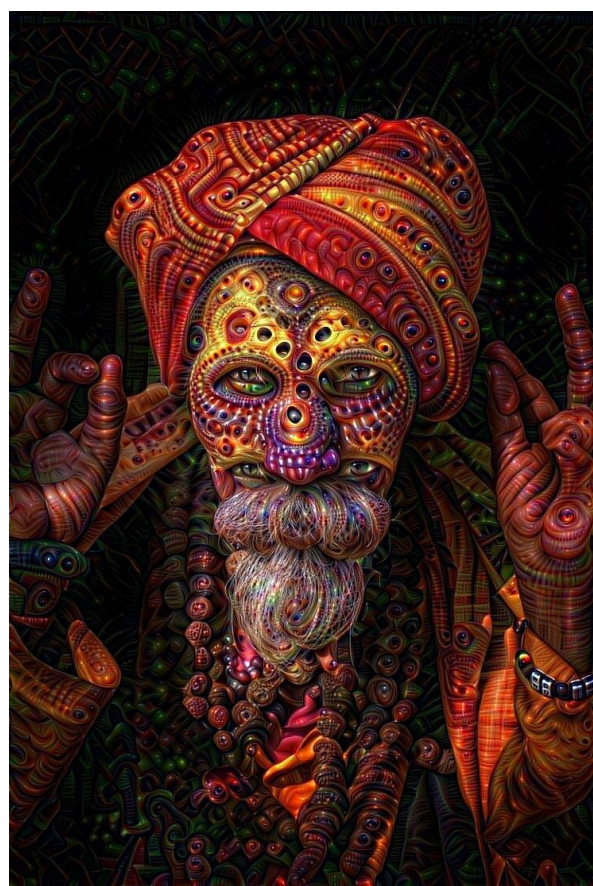
reddit.com/r/deepdream
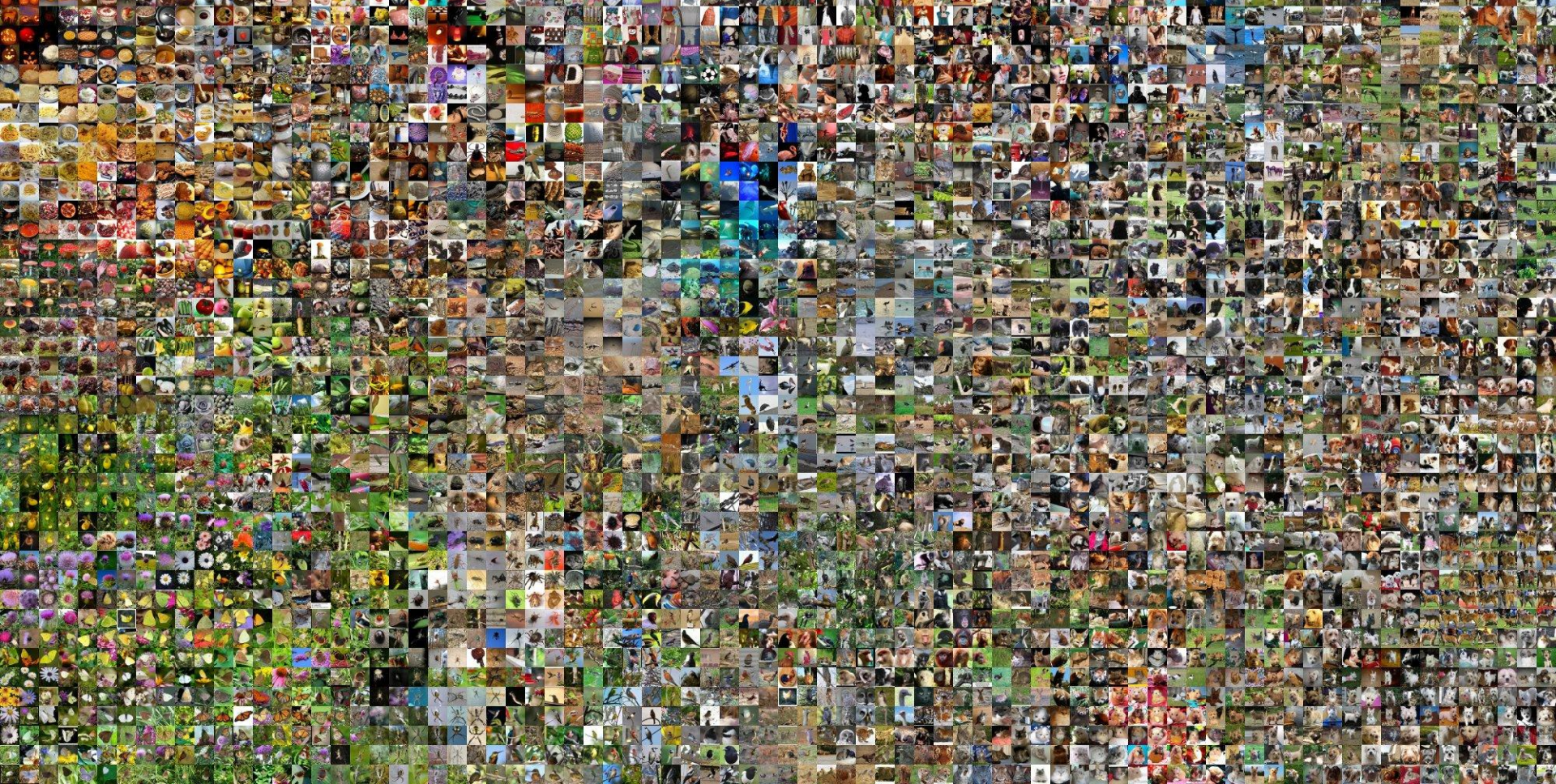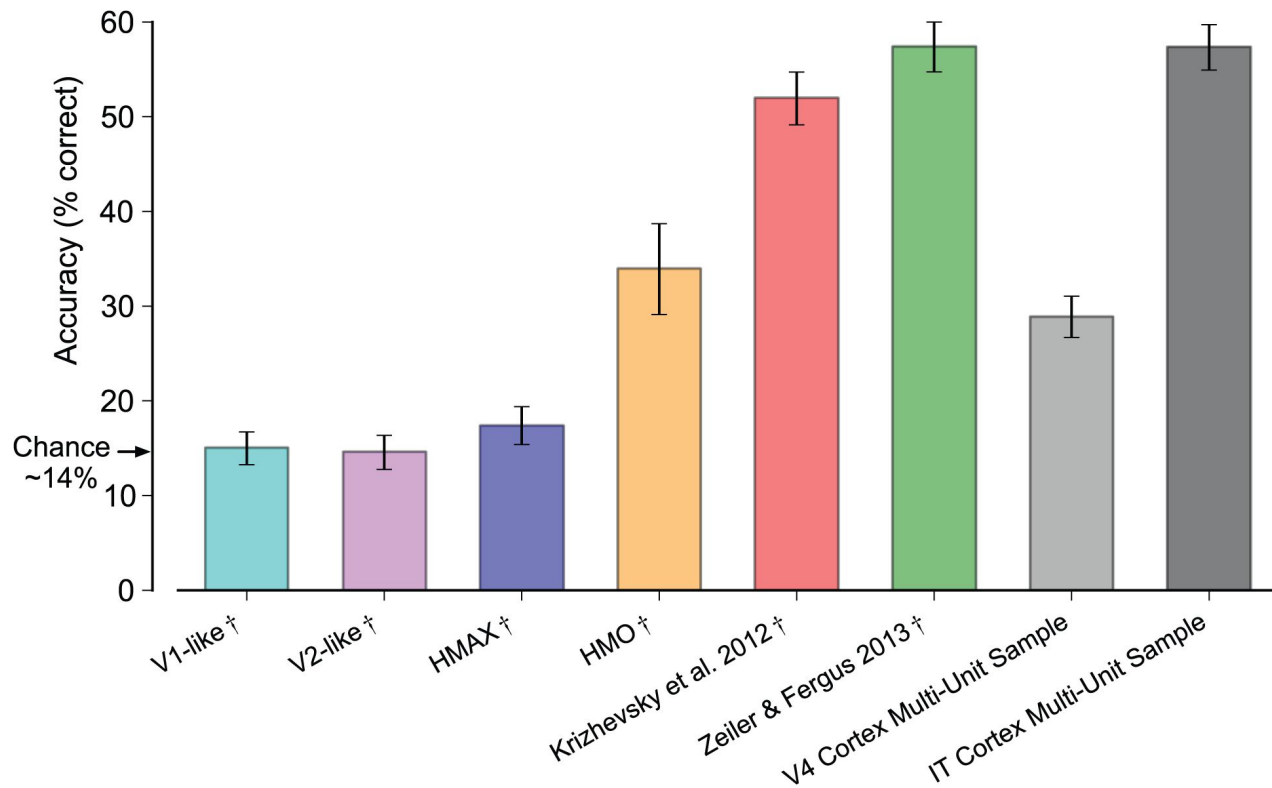
Deep Neural Networks Rival the Representation of Primate IT Cortex for Core Visual Object Recognition
[Cadieu et al., 2014]

Deep Neural Networks Rival the Representation of Primate IT Cortex for Core Visual Object Recognition
[Cadieu et al., 2014]