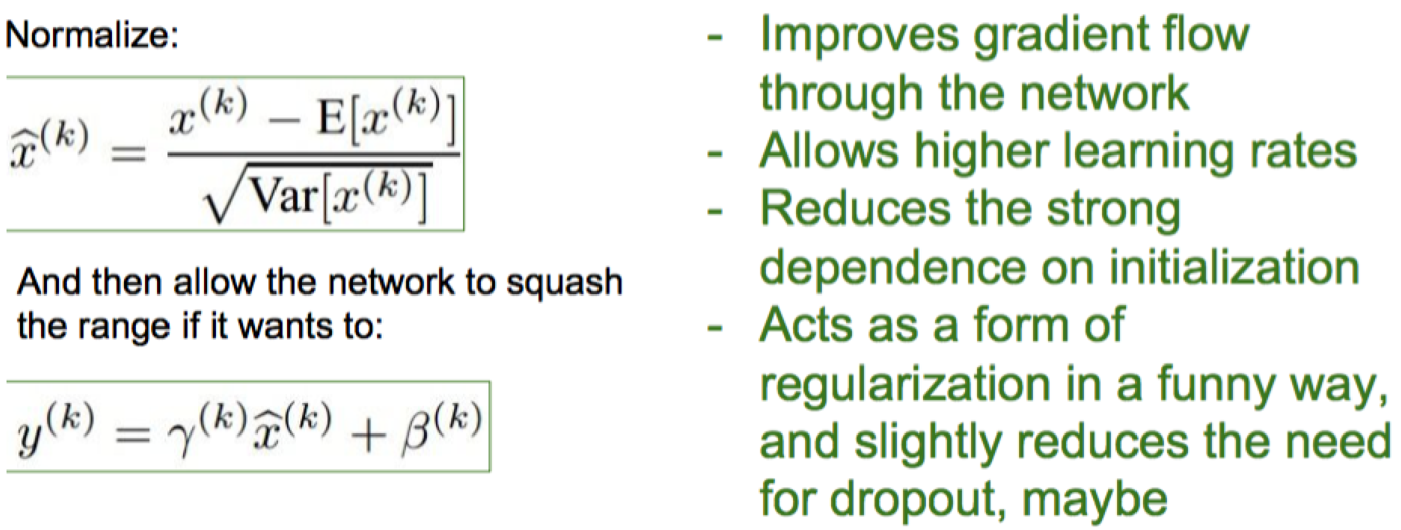
Lecture 6

Recap

In [1]:
```
#Why do we need Activation functions?
#Basically if you dont use Activation functions, your entire NN is just a sand
wich, so your
#capacity is just equal to the linear classifier.
#Activation function important in between to filter your data.
```

Batch Normalization

In [2]:
```
#What advantage does BN have?
```



Normalize:

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

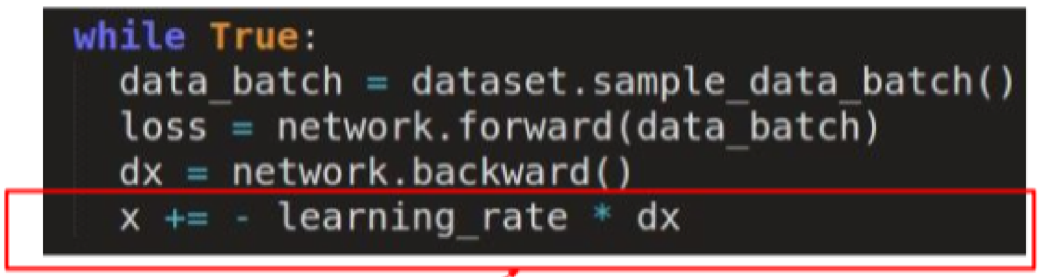
And then allow the network to squash the range if it wants to:

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

- 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

Parameter Update

Training a neural network, main loop:



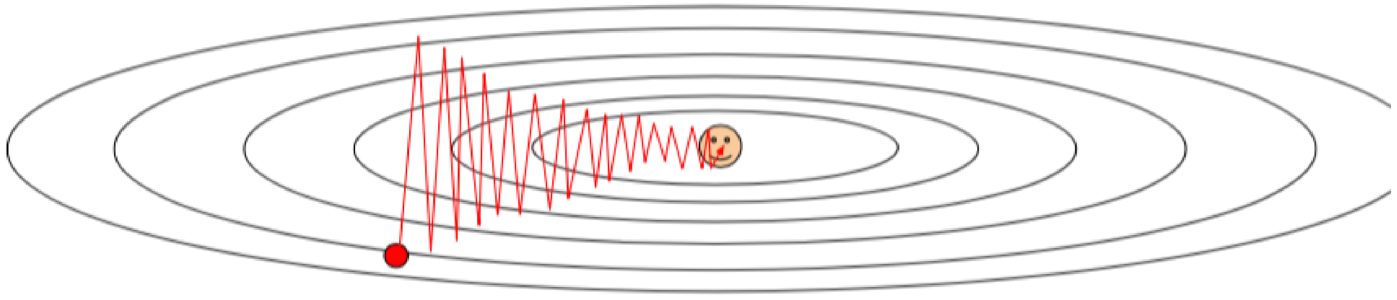
```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
```

simple gradient descent update
now: complicate.

Why do we do Para Update?

In [3]:
```
#why is SGD slow?
```

Suppose loss function is steep vertically but shallow horizontally:

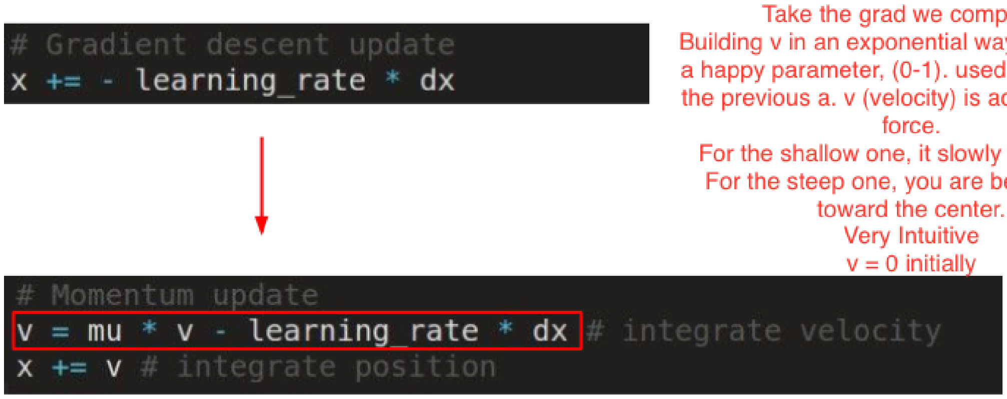


It is bouncing and is very slow to converge

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

In [4]:
```
#Allows a velocity to “build up” along shallow directions
#Velocity becomes damped in steep direction due to quickly changing sign
```



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

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

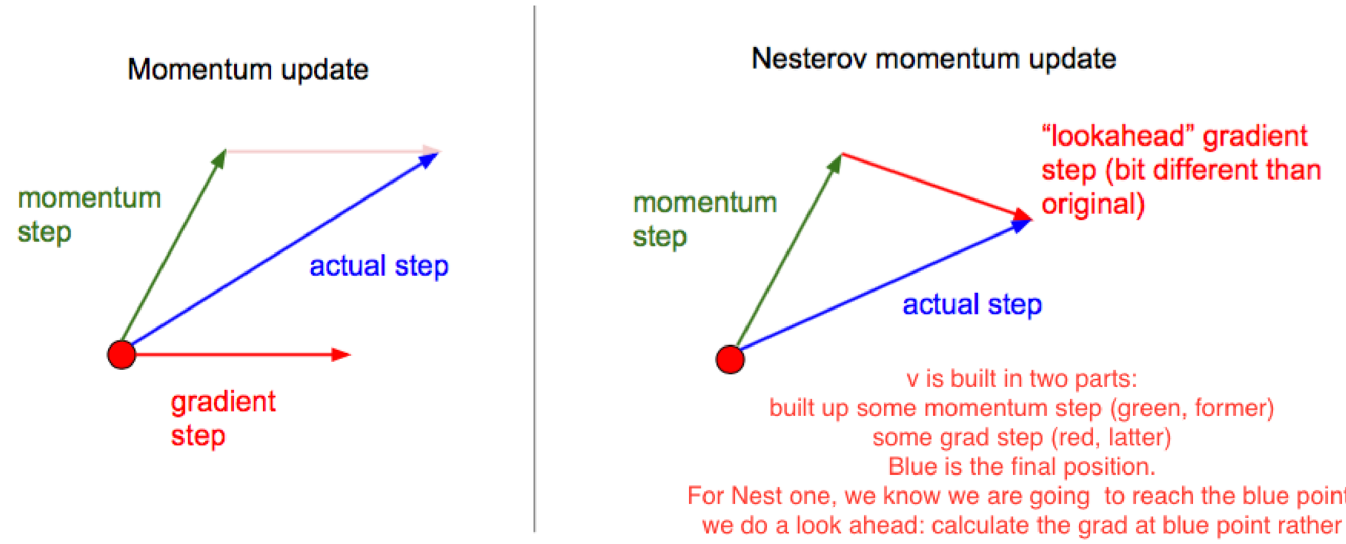
Take the grad we computed,
Building v in an exponential way. miu here is a happy parameter, (0-1). used for decaying the previous a. v (velocity) is accelerated by force.
For the shallow one, it slowly accelerate.
For the steep one, you are being pulled toward the center.
Very Intuitive
v = 0 initially

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

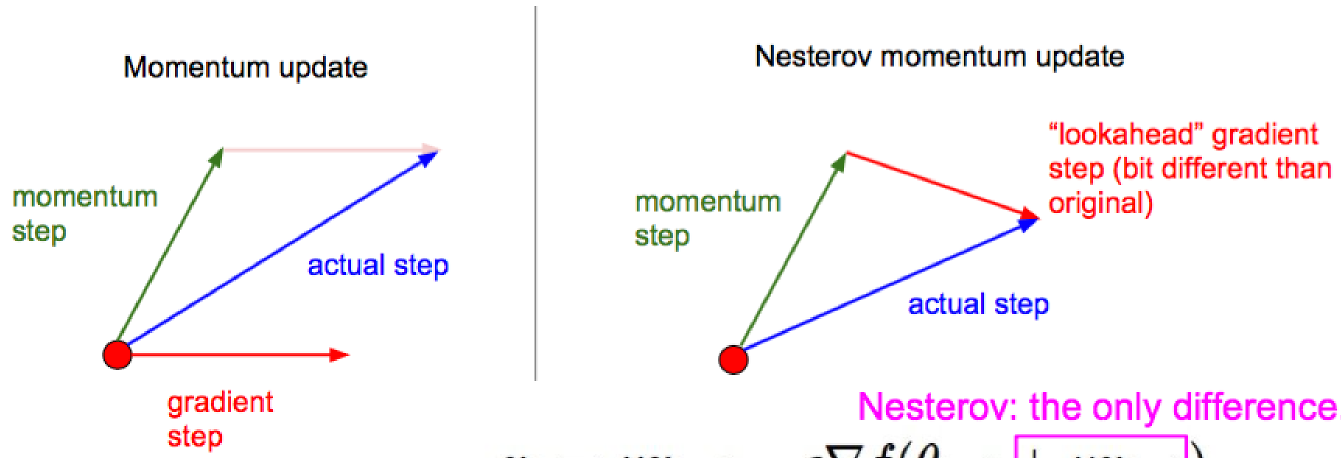
In [5]:
```
#SGD VS Momentum
#notice momentum overshooting the target, but overall getting to the minimum m
uch faster
```

Nesterov Momentum

In [6]:
```
#Build opon Momentum
#Did a look ahead
```



v is built in two parts:
built up some momentum step (green, former)
some grad step (red, latter)
Blue is the final position.
For Nest one, we know we are going to reach the blue point,
we do a look ahead: calculate the grad at blue point rather
than the start point



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

Always assures a better performance than the former one

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

green one

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

grad. at blue point

Slightly inconvenient… usually we have :

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

Theta is gradient para representing the grad at that point. Theta(t) used be only be relavant to d(Theta t), not it is relevant to d(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
```

AdaGrad Update

AdaGrad update

scaling the gradient. cache is positive only.

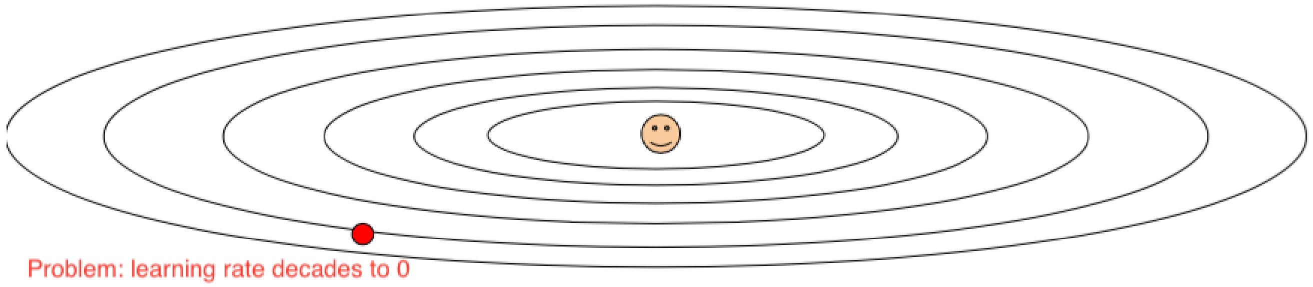
Cache here is a giant vector of the same size as your para vector. The operation here is element wise

In every dimension we keep tracking the sum of the grad.

Everything dimension has own learning rate in terms of what scale of grad you've seen

```
# 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. Divide also element wise

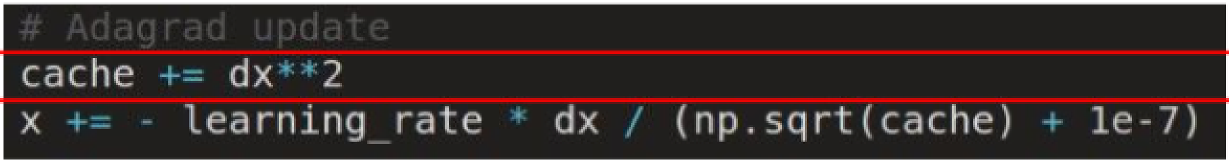


Problem: learning rate decades to 0

Q: What happens with AdaGrad? vertical (steep): large, and divided, slower. vice versa

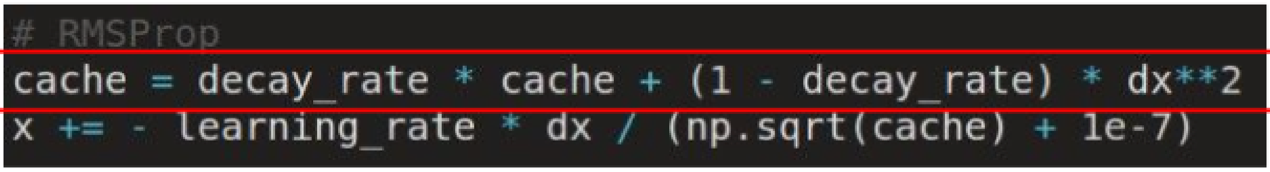
RMSProp Update

Improvement on Ada



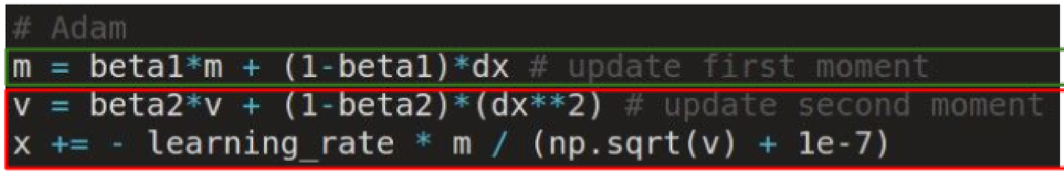
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

We have the decay rate, so the cache is leaking, learning rate is not going to 0



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

Adam Update



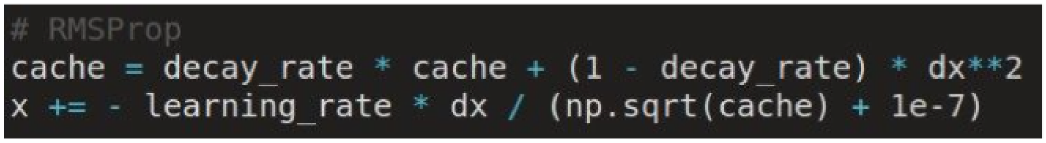
```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Keep tracking of moment in a decaying way and in the mean time scaling it

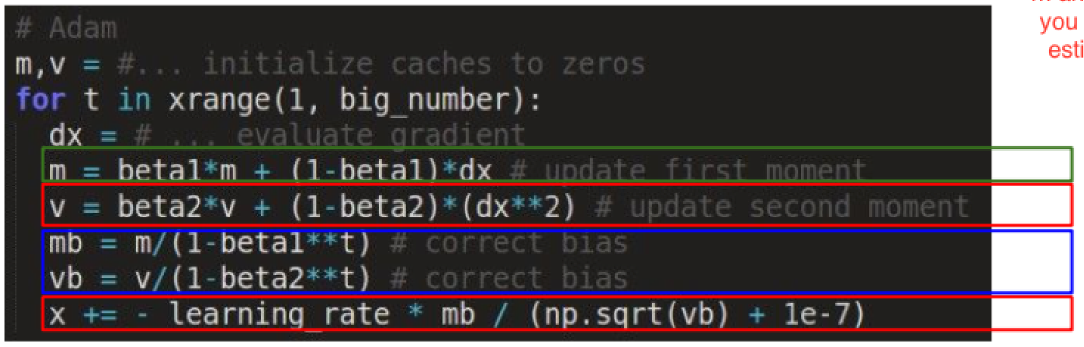
Looks a bit like RMSProp with momentum



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

In [7]:
```
#However, the above one is not complete
```

t is basically compensating that m and v are initialized 0. it scales up m and v in first few iterations so you dont end up with a biased estimate of first moment and second moment.



```
# 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)

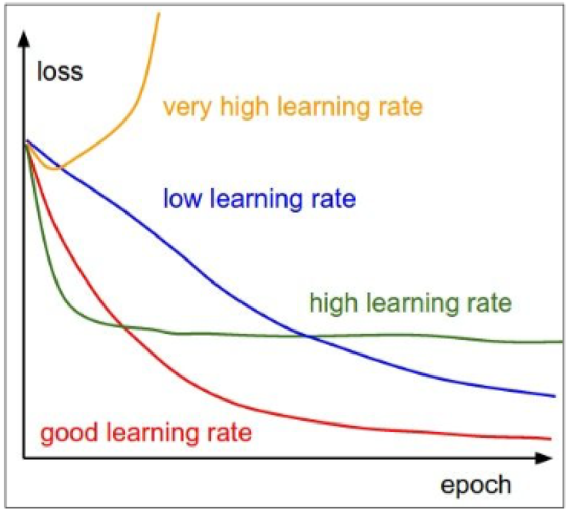
RMSProp-like

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

Choosing Learning Rate As a HyperPramater

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

Q. Which one of these rates is best to use?
Use high rate first, some point decay your learning rate (ways of decaying below)



=> 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

second-order Taylor expansion:

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

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

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

Second order methods for optimization: they end up forming a larger approximation for your lost function so they dont only approximated with this basically hpyerplane but also approximated hessian that tells you how your surface is curving so you dont only need the grad you also need the hessian (compute that)

Cons: Say 1 billion para, you have 1b * 1b Hessian

Q: what is nice about this update?

no hyperparameters! (e.g. learning rate). No need for learning. you know the grad and curvature, so you know exactly where to go

Second order optimization methods

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

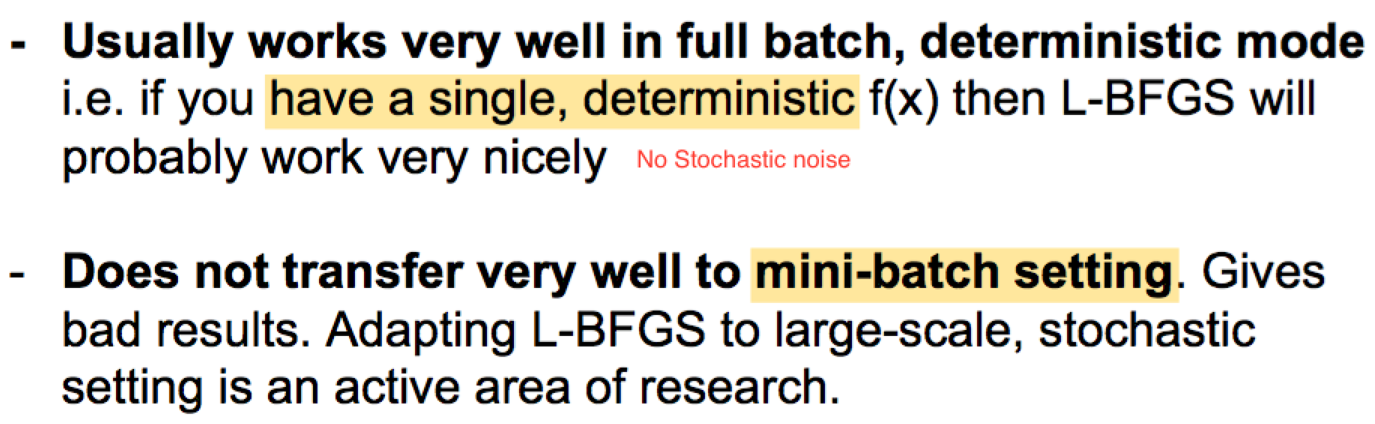
BGFS: Get away with not inverting the Hessian, it builds up an approximation through successive updates that are all ranked one (vector)

- 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).*

Solved the memory problem for BFGS

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

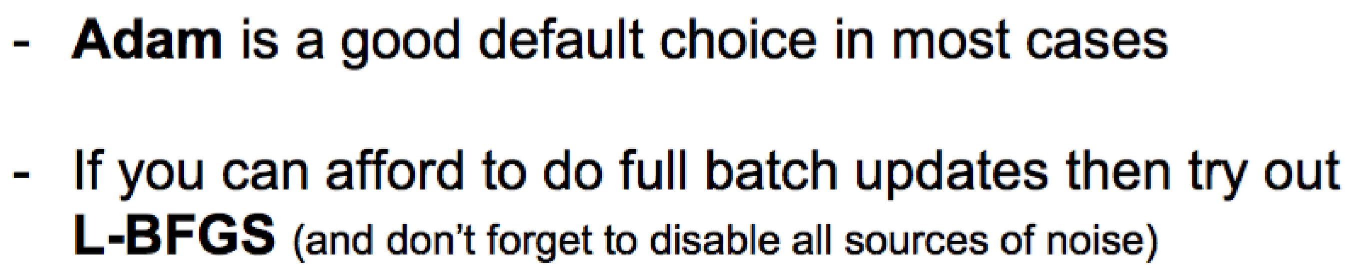
L-BFGS



- **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 No Stochastic noise

- **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.

Summary

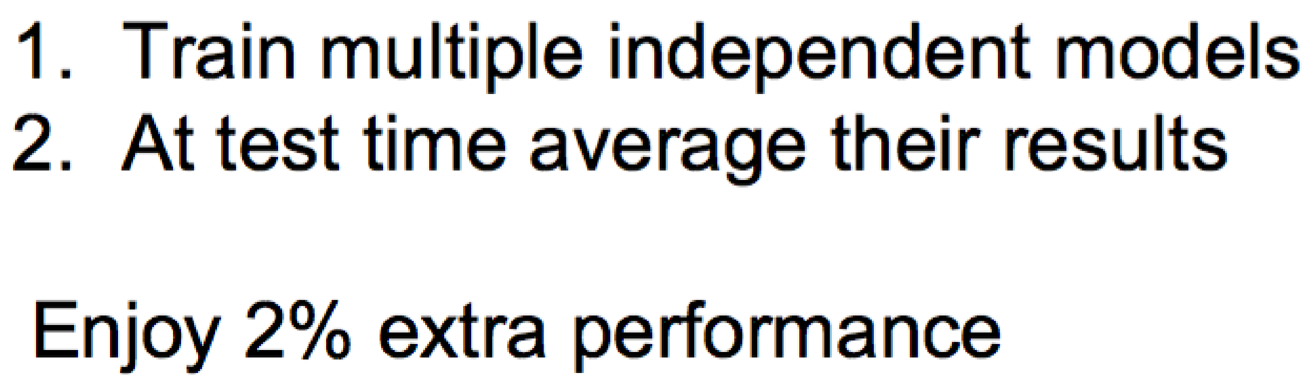


- **Adam** is a good default choice in most cases

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

Model Ensembles

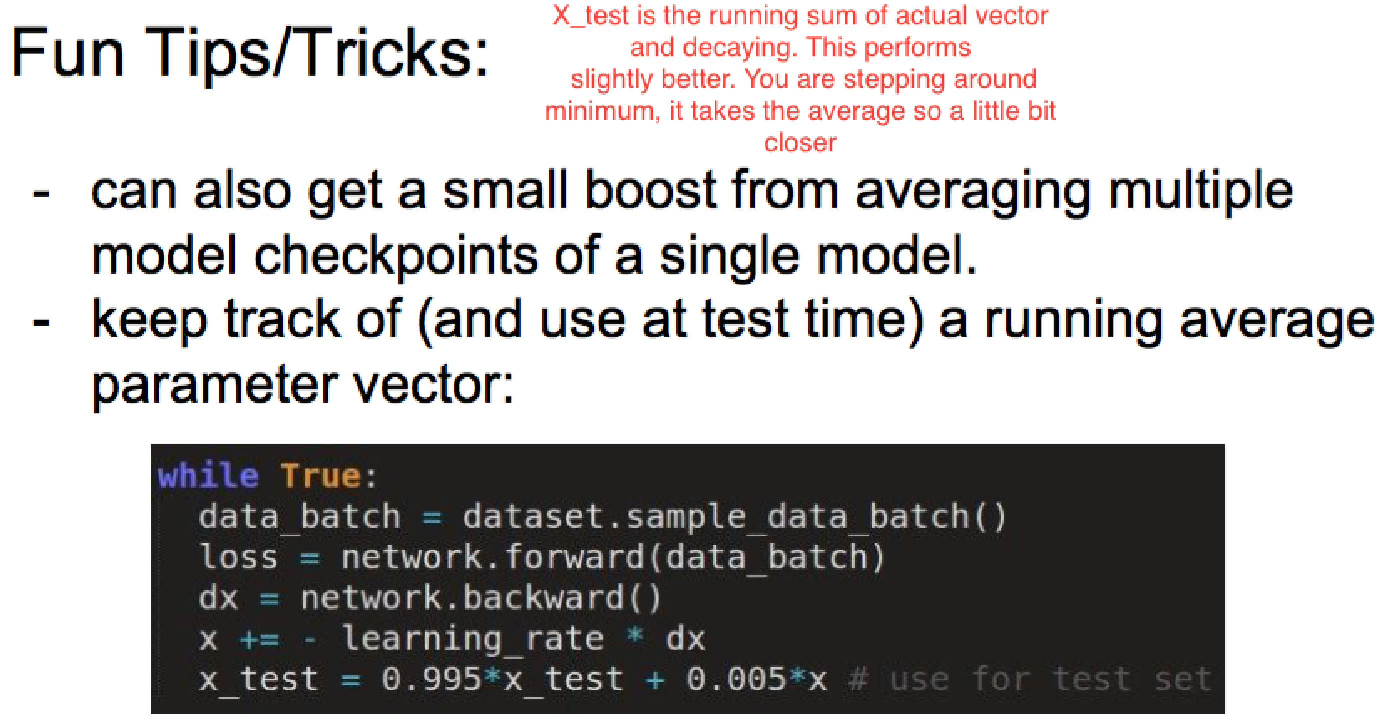
Refer to TTIC Architecture



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

Enjoy 2% extra performance

```
In [8]: #Several Tricks
```



Fun Tips/Tricks:

X_test is the running sum of actual vector and decaying. This performs slightly better. You are stepping around minimum, it takes the average so a little bit closer

- 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)

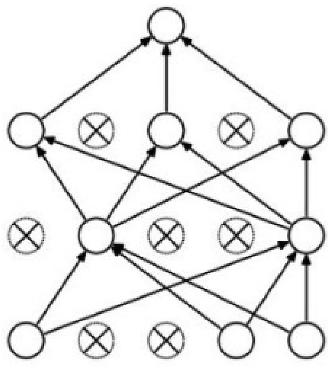
In [9]:
```
#randomly set some neurons to zero in the forward pass”
```

```
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)
```

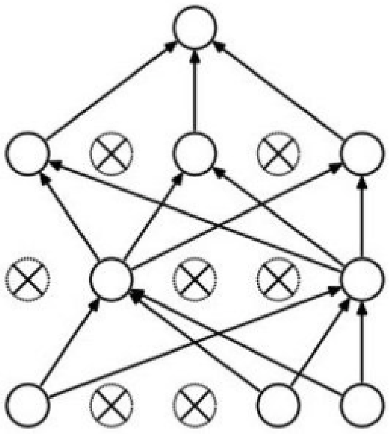
During forward pass, On possibility (1-p) of setting neurons to 0

Have to adjust back propagate too, you back propagated by multiplying by u2 and u1

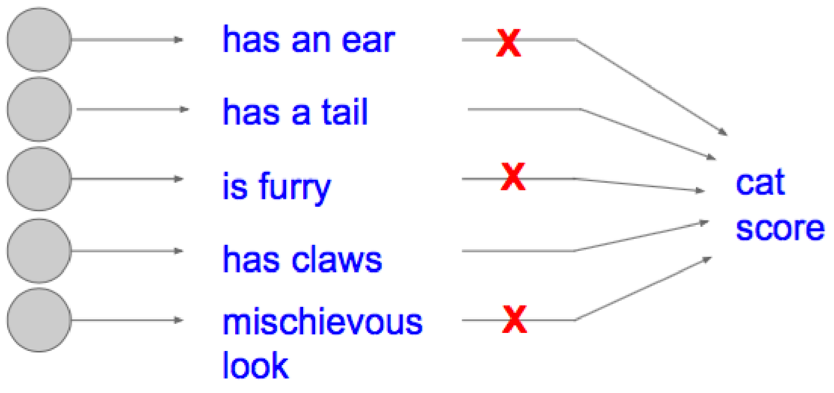
Example forward pass with a 3-layer network using dropout



In [10]:
```
#Why does it work?
```

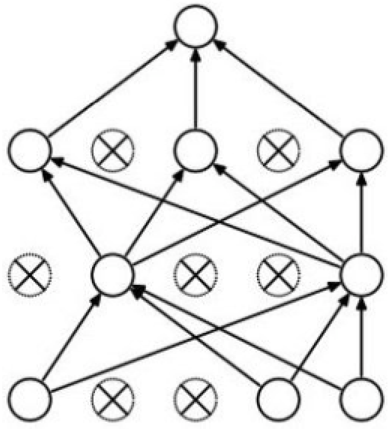


Forces the network to have a redundant representation.



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

If one neuron is dropped, it gets zero in forward propagation and it's grad is not affected in backprapagation. So its not updated. Kind of training a subset of neuron network (but all neurons share the same para)



Another interpretation:

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

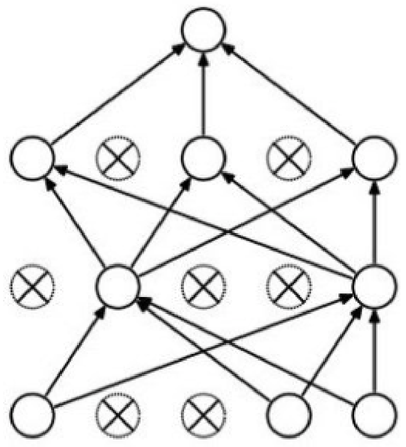
Each binary mask is one model, gets trained on only ~one datapoint.

```
In [11]: #How to do this at Test time?
```

At test time….

The noise: possibility

BGFS: Get away with not inverting the Hessian, it builds up an approximation through successive updates that are all ranked one (vector)



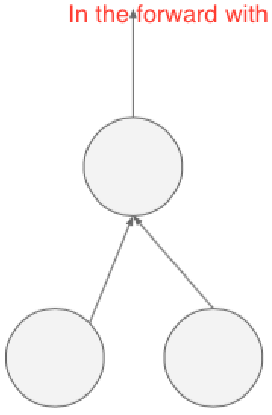
Ideally:
want to integrate out all the noise

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).

In the forward with test images we are not going to drop out any units. Be careful because during train is different



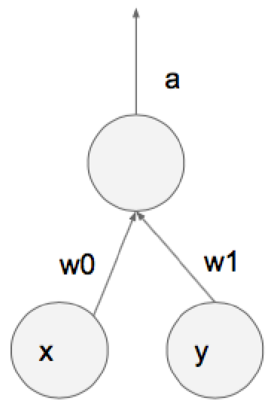
(this can be shown to be an approximation to evaluating the whole ensemble)

In [12]:
```
#Notice that we are not going to drop any neurons during test time.
#Thats why you need to compensate for it during train
```

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**

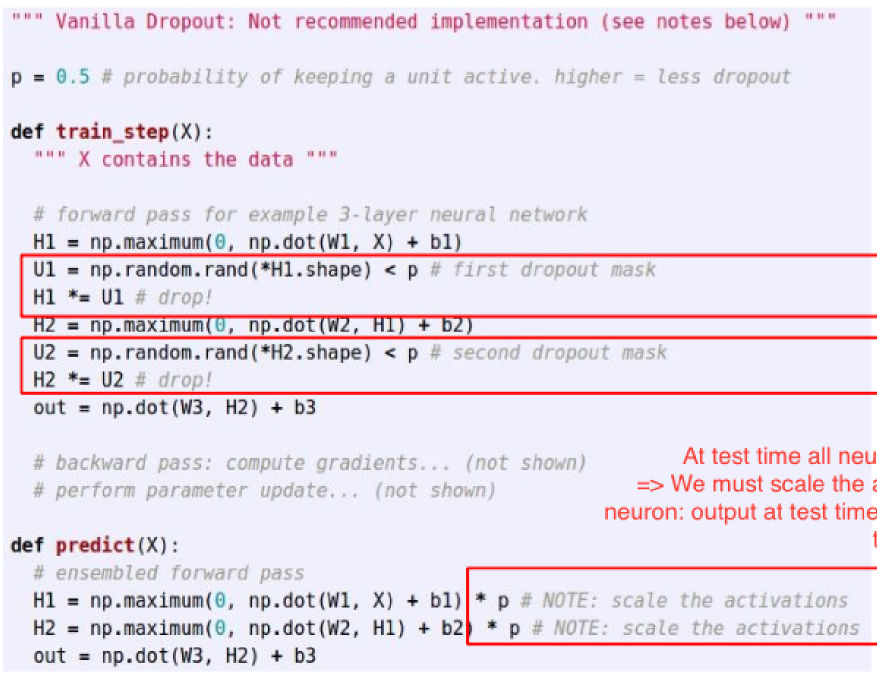
during train:

$$
\begin{aligned}
E[a] = \tfrac{1}{4} * (&w0*0 + w1*0 \\
&w0*0 + w1*y \\
&w0*x + w1*0 \\
&w0*x + w1*y) \\
= \tfrac{1}{4} * (&2\ w0*x + 2\ w1*y) \\
= \tfrac{1}{2} * (&w0*x + w1*y)
\end{aligned}
$$

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

Need to compensate for that. Scale your activation function

```
In [13]: #Summary
         #1.Normal way
```



```
""" 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
```

Dropout Summary

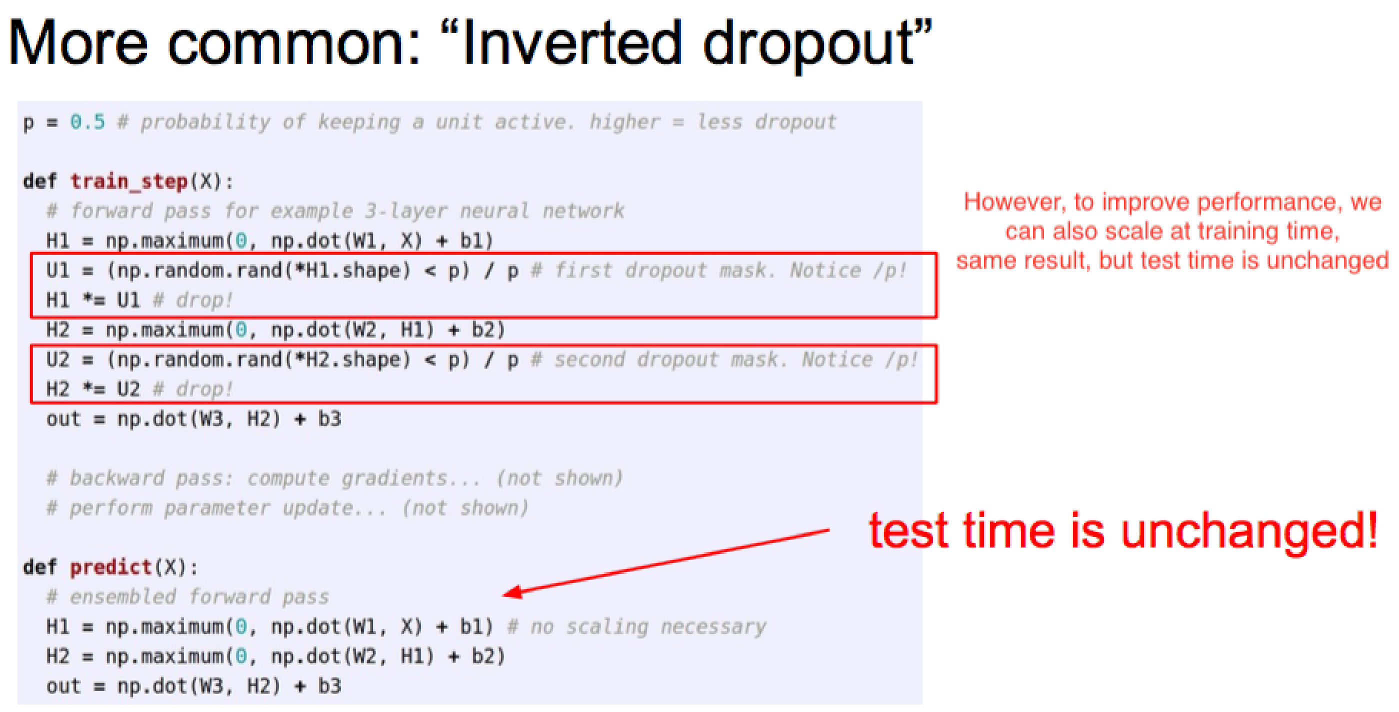
This is scaling in test time

drop in forward pass

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

scale at test time

```
In [14]: #2. Improved way
```



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
```

However, to improve performance, we can also scale at training time, same result, but test time is unchanged

test time is unchanged!

```
In [ ]:
```

```
In [ ]:
```