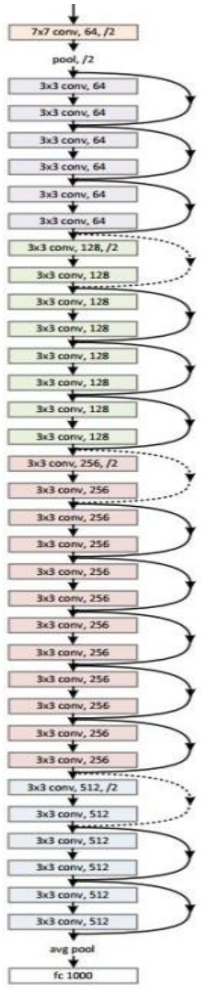
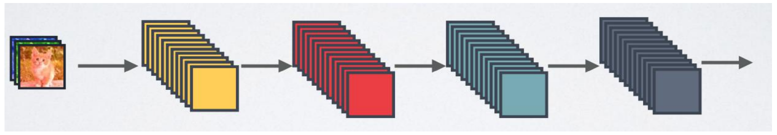
Rensselaer

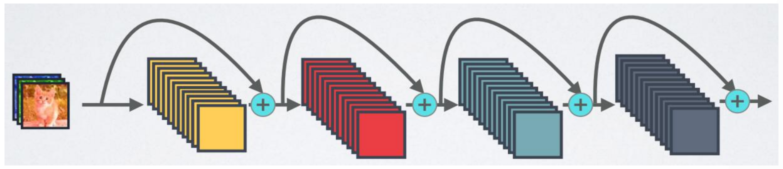
Lecture 19: Concepts for Neural Networks

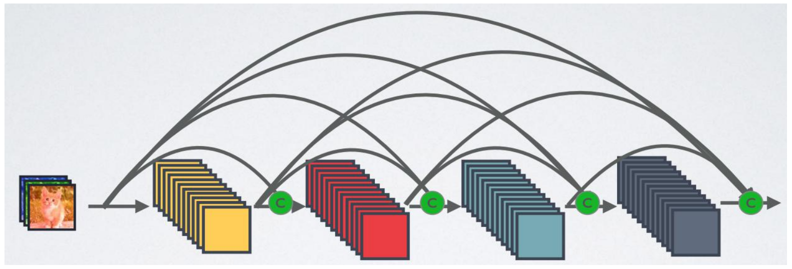
Dr. Chengjiang Long
Computer Vision Researcher at Kitware Inc.
Adjunct Professor at RPI.
Email: **longc3@rpi.edu**

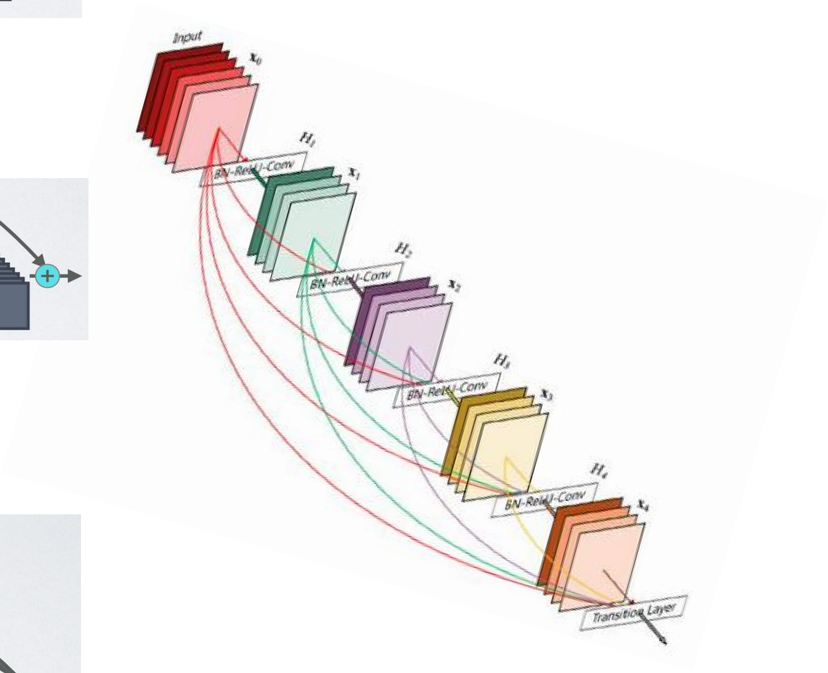
Recap Previous Lecture











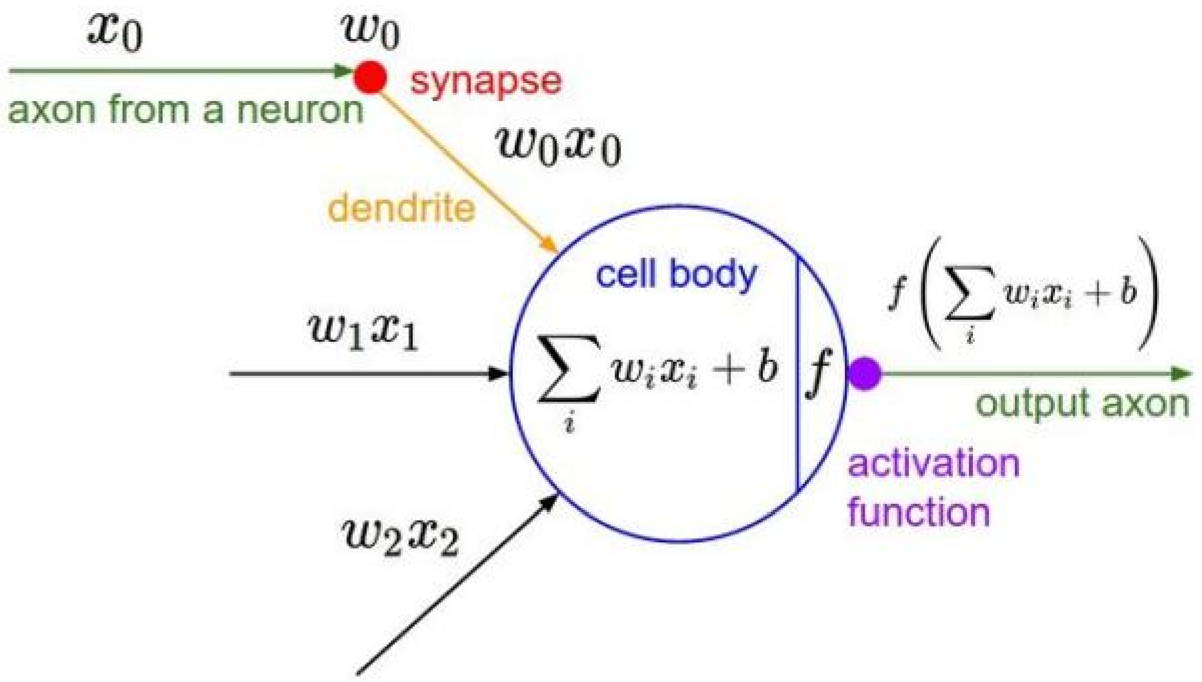
Outline

- Activation Functions
- Parameter update schemes
- Learning rate schedules
- Generalization with Dropout
- Application of ConvNets

Outline

- **Activation Functions**
- Parameter update schemes
- Learning rate schedules
- Generalization with Dropout
- Application of ConvNets

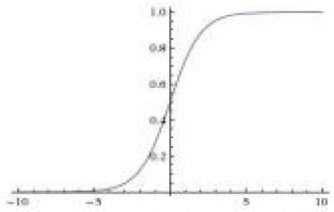
Activation Functions



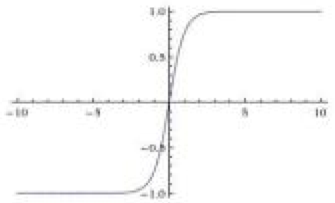
Activation Functions

Sigmoid

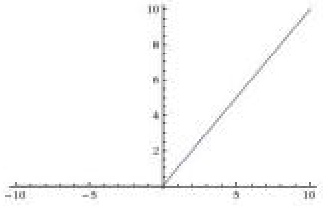
$\sigma(x) = 1/(1+e^{-x})$



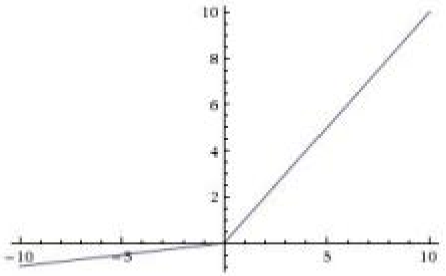
tanh tanh(x)



ReLU max(0,x)

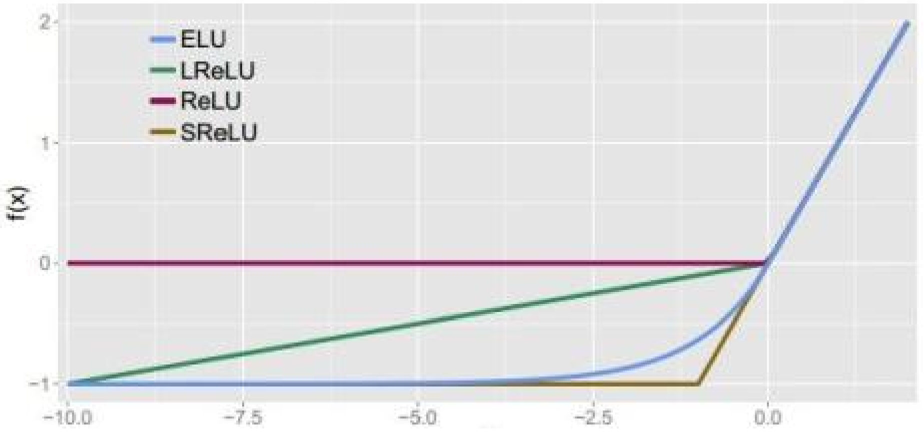


Leaky ReLU
max(0.1x, x)



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

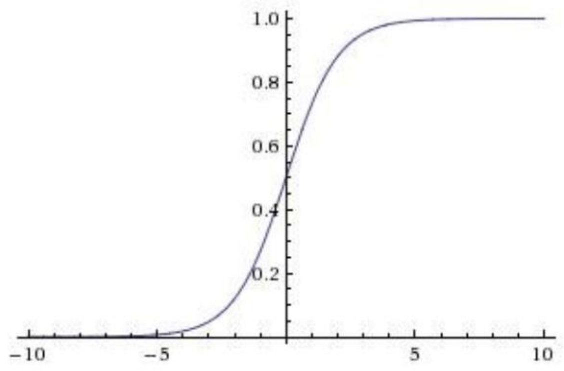
ELU $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$



SReLU (Shift Rectified Linear Unit)

max(-1, x)

Activation Functions-Sigmoid function



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

- Squashes numbers to range $[0,1]$
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- It has three problems:

① Saturated neurons "kill" the gradients

② Sigmoid outputs are not zerocentered

③ exp() is a bit compute expensive

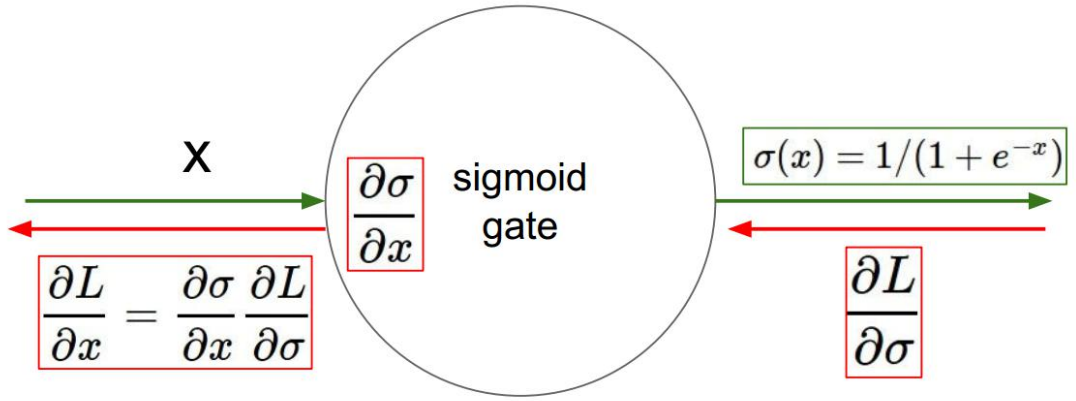
Activation Functions-Sigmoid function

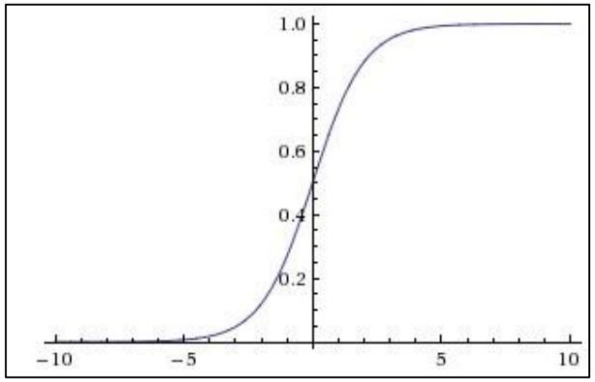
- It has three problems:

① Saturated neurons "kill" the gradients

② Sigmoid outputs are not zerocentered

③ exp() is a bit compute expensive


x
sigmoid gate
$\frac{\partial \sigma}{\partial x}$
$\sigma(x) = 1/(1+e^{-x})$
$\frac{\partial L}{\partial \sigma}$
$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$



What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Activation Functions-Sigmoid function

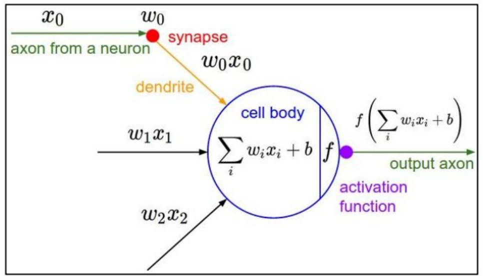
- It has three problems:

① Saturated neurons “kill” the gradients

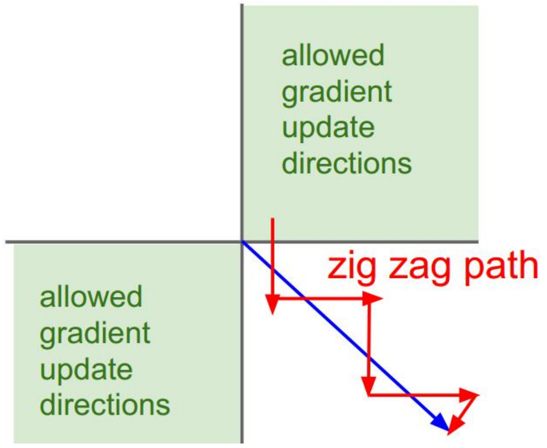
② Sigmoid outputs are not zerocentered

③ exp() is a bit compute expensive

Consider what happens when the input to a neuron (x) is always positive:



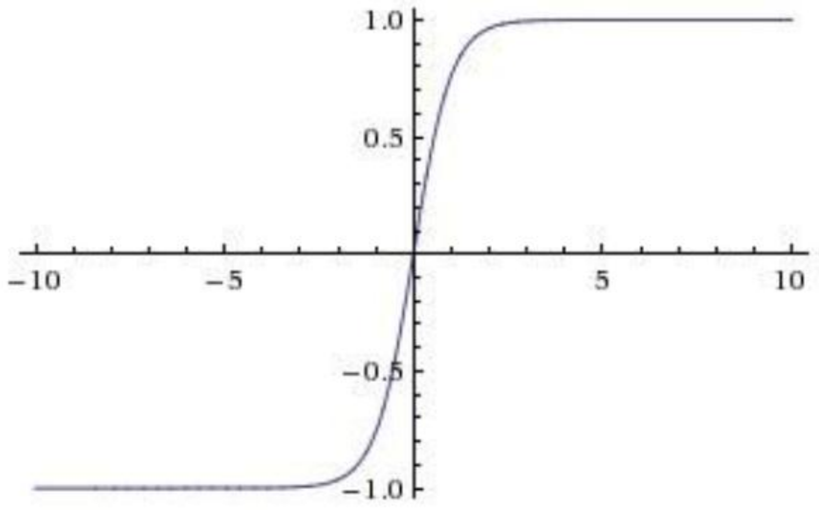
$$f\left(\sum_i w_i x_i + b\right)$$


allowed gradient update directions
allowed gradient update directions
zig zag path

What can we say about the gradients on w?
Always all positive or all negative :
(this is also why you want zero-mean data!)

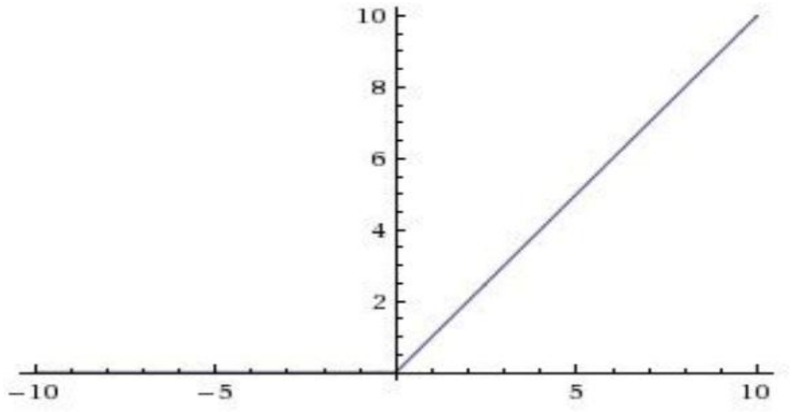
hypothetical optimal w vector

Activation Functions-tanh(x)



- Squashes numbers to range [-1,1]
- Zero centered (nice)
- Still kills gradients when saturated

Activation Functions-ReLU

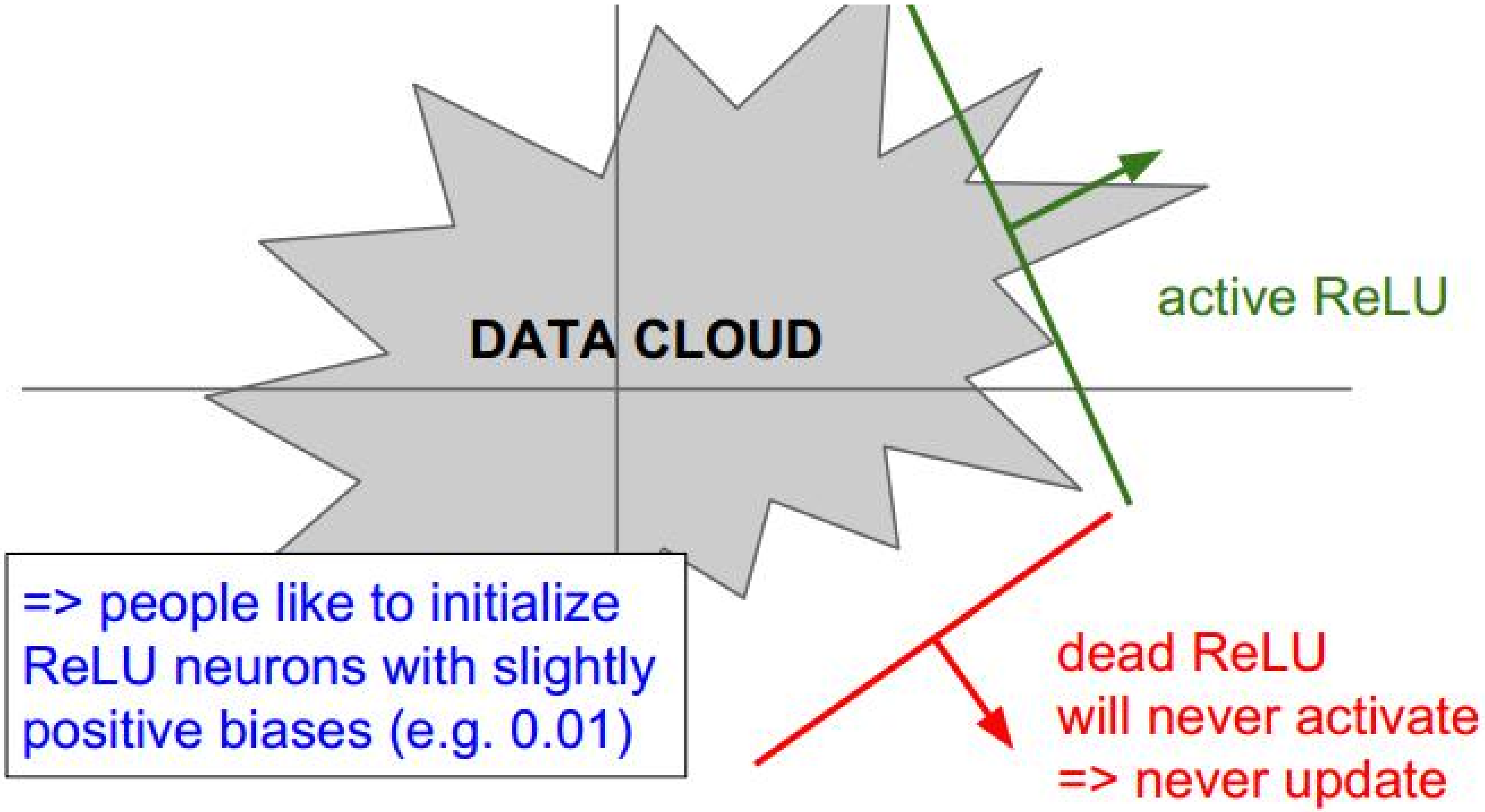


ReLU
(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:
- hint: what is the gradient when x < 0?

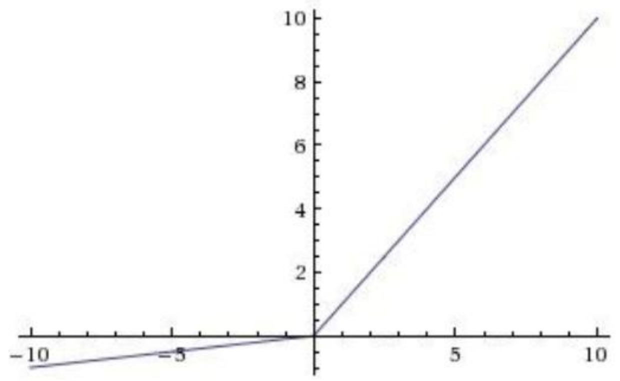
[Krizhevsky et al., 2012]

Activation Functions-ReLU



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation Functions-Leaky ReLU



$$f(x) = \max(0.01x, x)$$

[Mass et al., 2013]
[He et al., 2015]

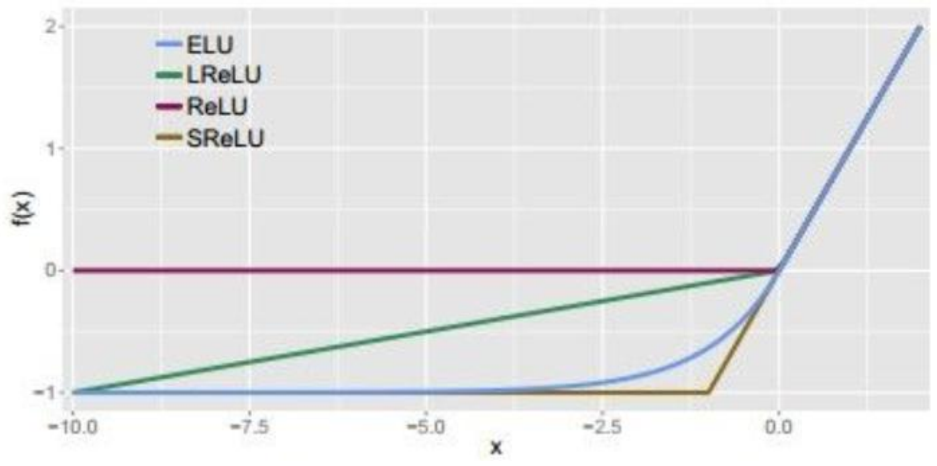
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not "die".

Parametric Rectifier (PReLU)

$$f(x) = \max(\underset{\uparrow}{\alpha} x, x)$$

backprop into \alpha (parameter)

Activation Functions-ELU



Exponential Linear Units (ELU)

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

[Clevert et al., 2015]

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- **Computation requires exp**( )

Activation Function-Maxout "Neuron"

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

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!
- Problem: doubles the number of parameters/neuron.

In practice

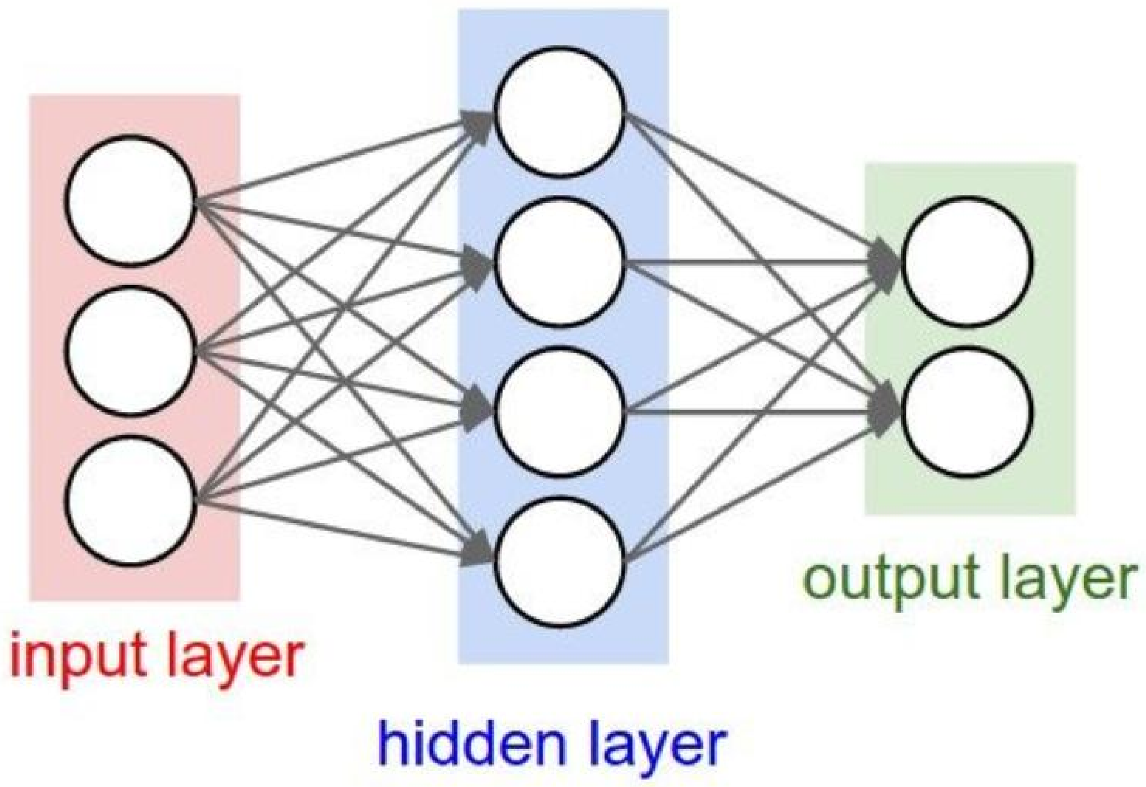
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid

Outline

- Activation Functions
- **Parameter update schemes**
- Learning rate schedules
- Generalization with Dropout
- Application of ConvNets

Weight Initialization

- Q: what happens when W=0 init is used？

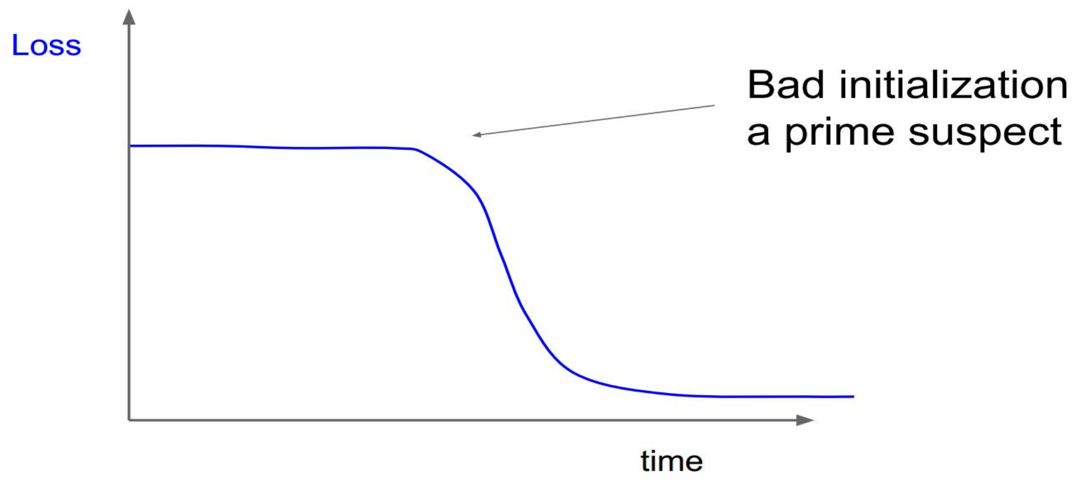


Weight Initialization

- First idea: Small random numbers (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.



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

This method takes the best of both batch and SGD, and performs an update for every mini-batch of n.

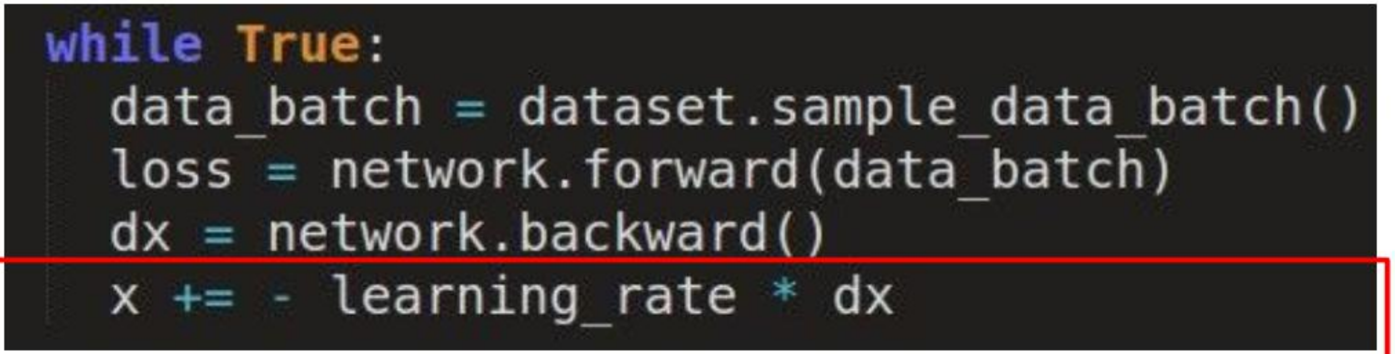
Update equation

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

Code

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Training a neural network, main loop:

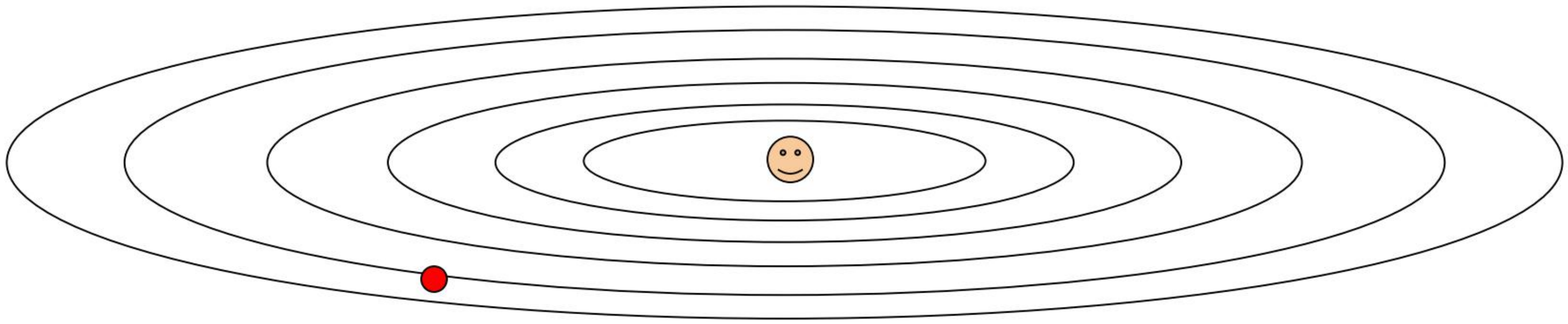


```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
```

simple gradient descent update
now: complicate.

SGD

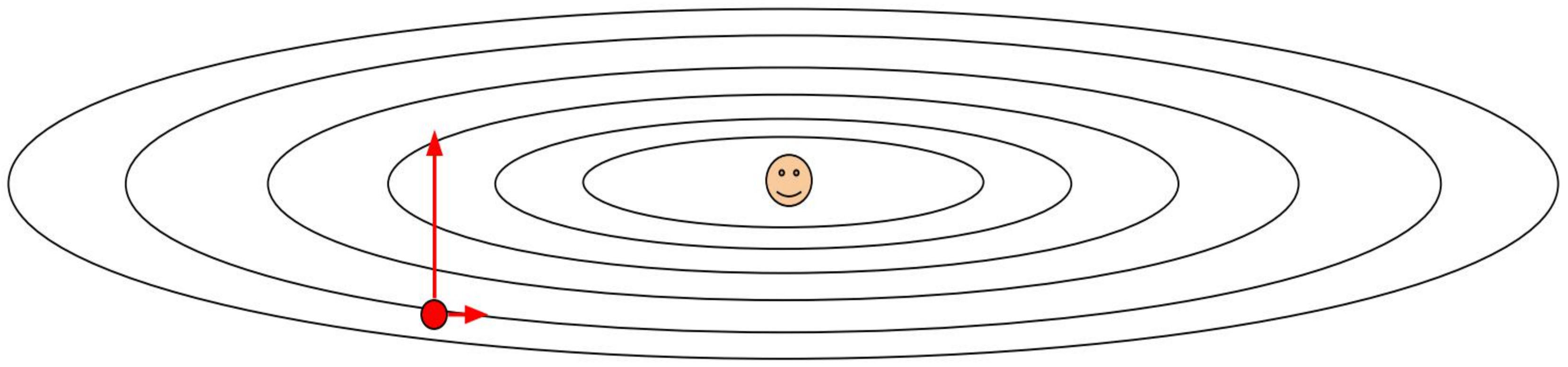
- Suppose loss function is steep vertically but shallow horizontally:



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

SGD

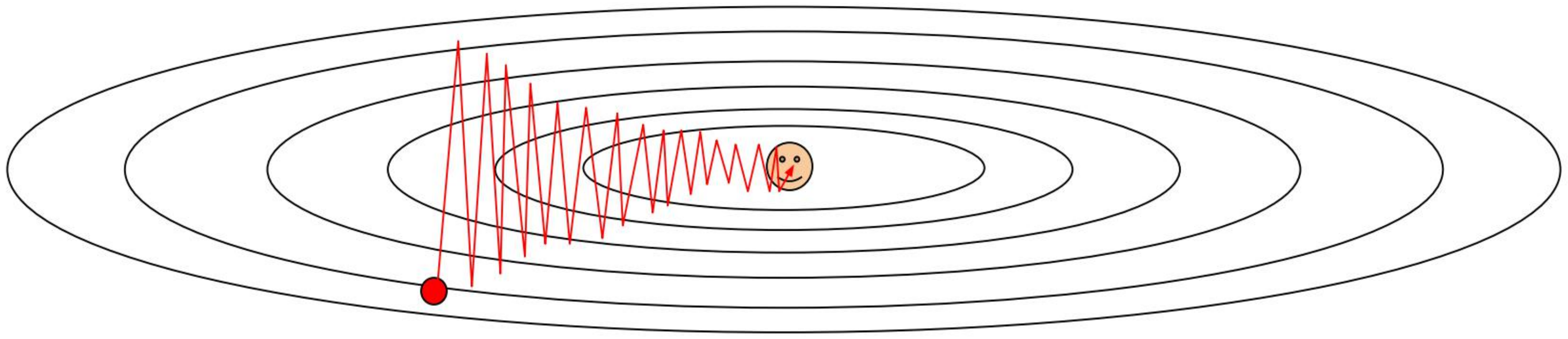
- Suppose loss function is steep vertically but shallow horizontally:



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

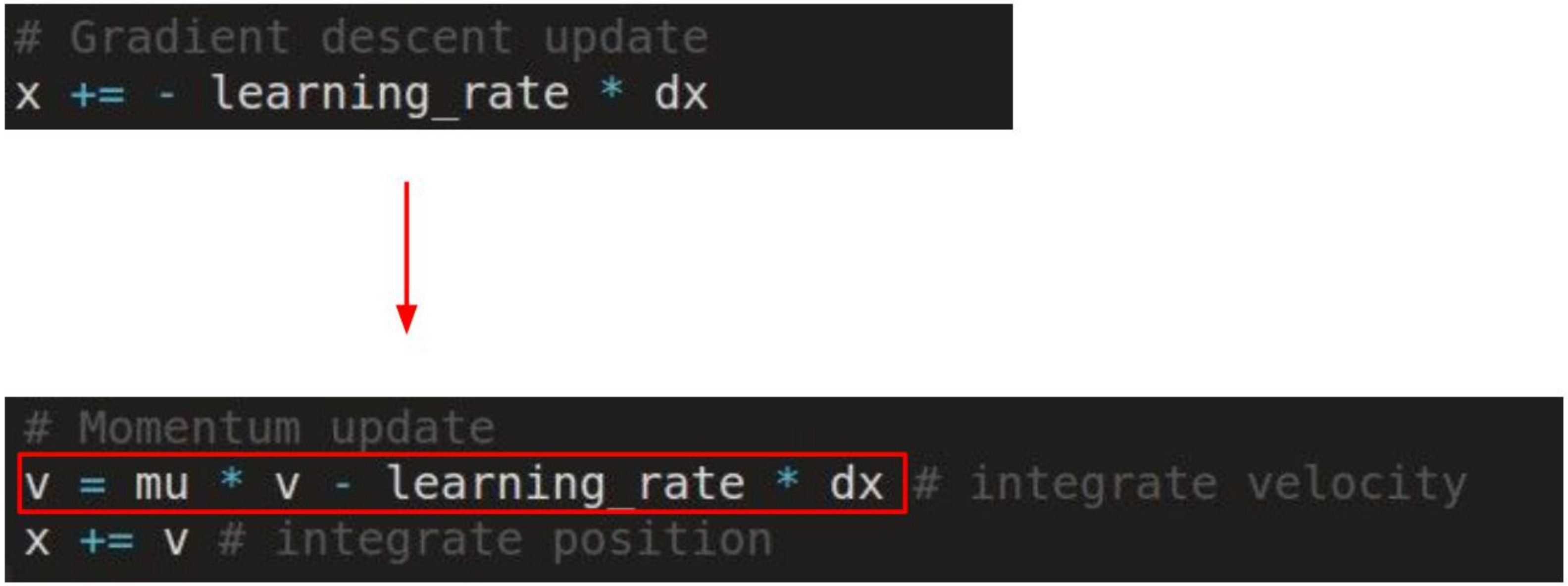
SGD

- Suppose loss function is steep vertically but shallow horizontally:



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

Momentum update

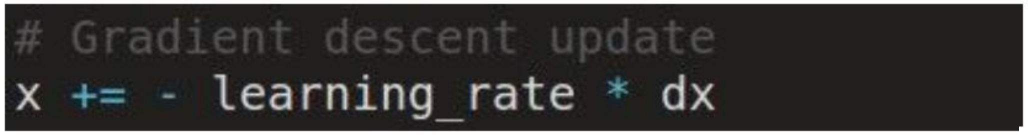


```
# Gradient descent update
x += - learning_rate * dx
```

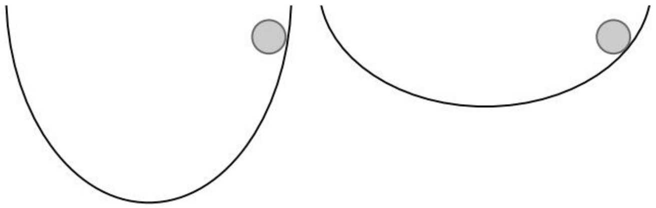
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

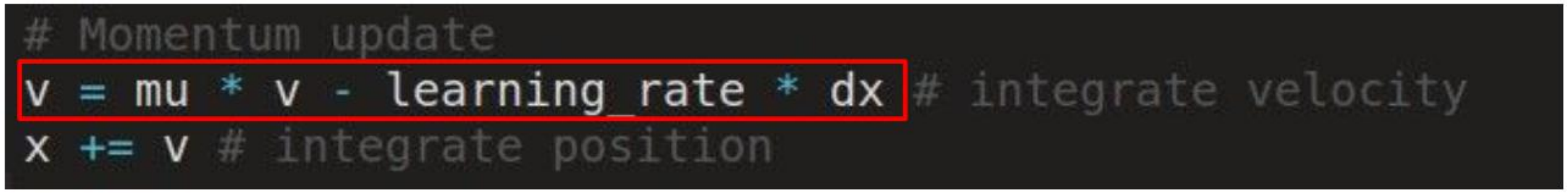
- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

Momentum update



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

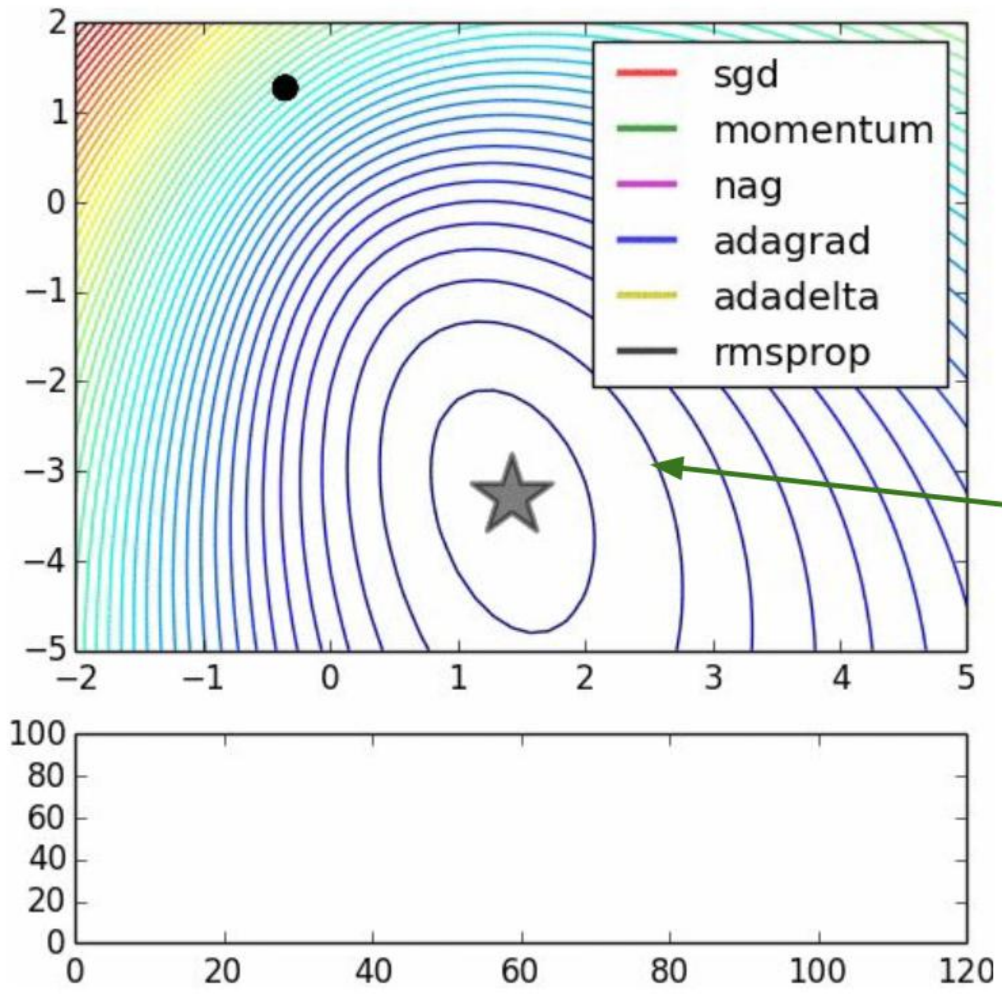




```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

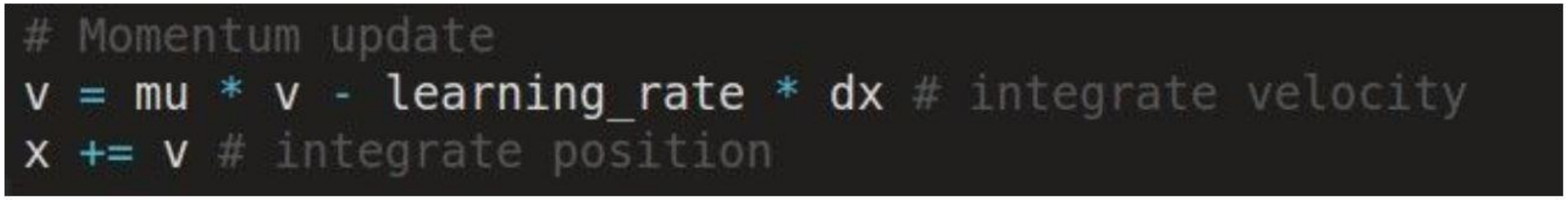
- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD vs Momentum



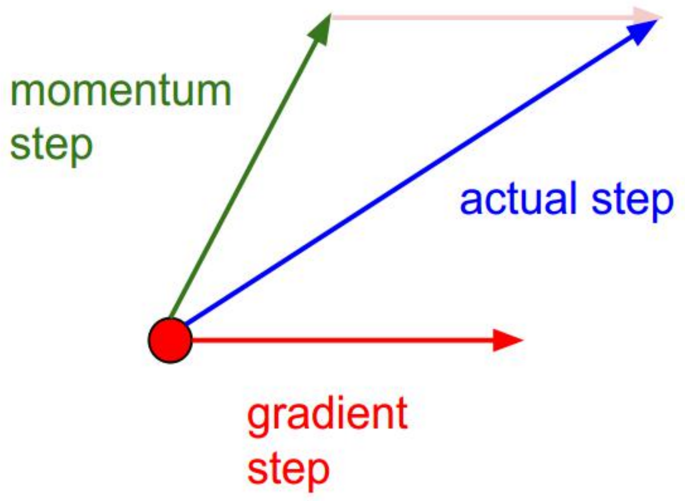
notice momentum overshooting the target, but overall getting to the minimum much faster.

Nesterov Momentum update

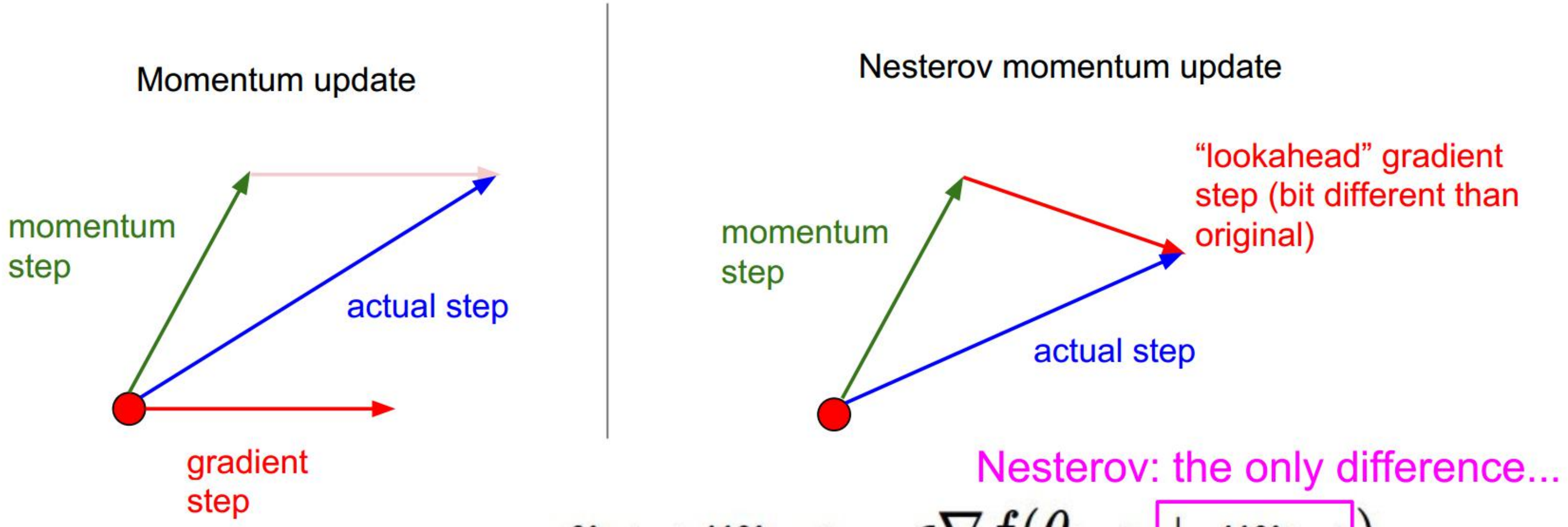


```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Ordinary momentum update:



Nesterov Accelerated Gradient (NAG)



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 Accelerated Gradient (NAG)

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

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

Slightly inconvenient…
usually we have :

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

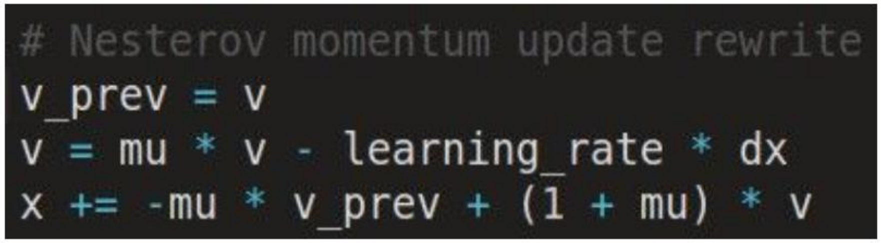
Variable transform and rearranging saves the day:

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

Replace all thetas with phis, rearrange and obtain:

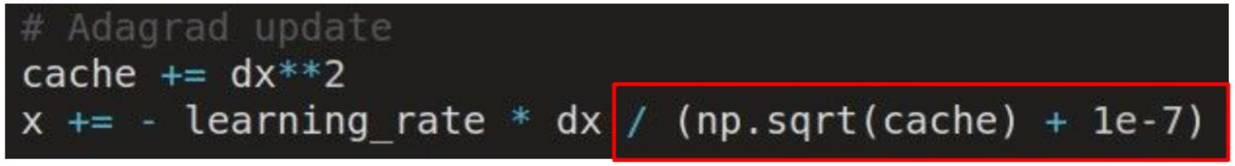
$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$

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



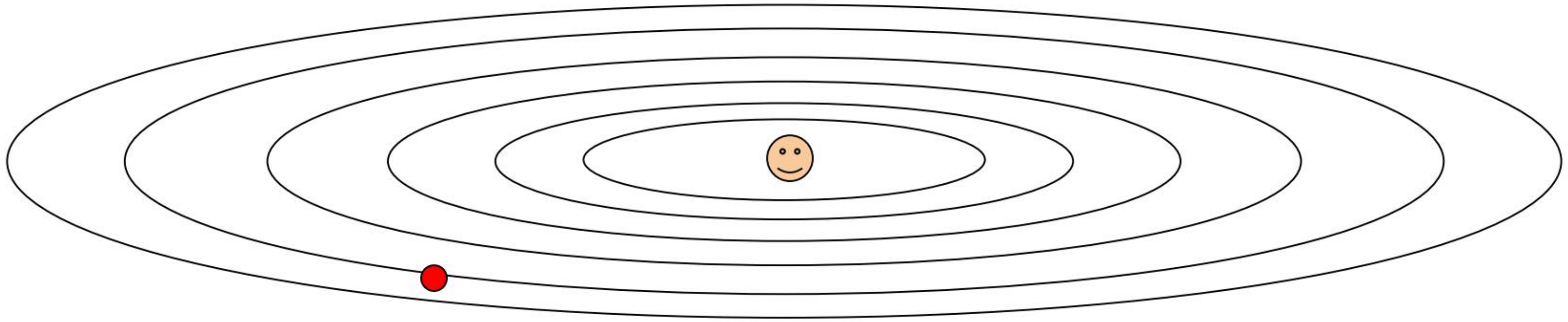
```
# Nesterov momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```

AdaGrad update



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

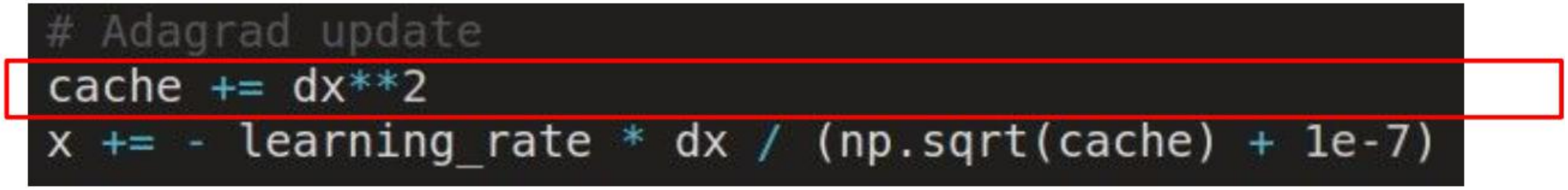
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension



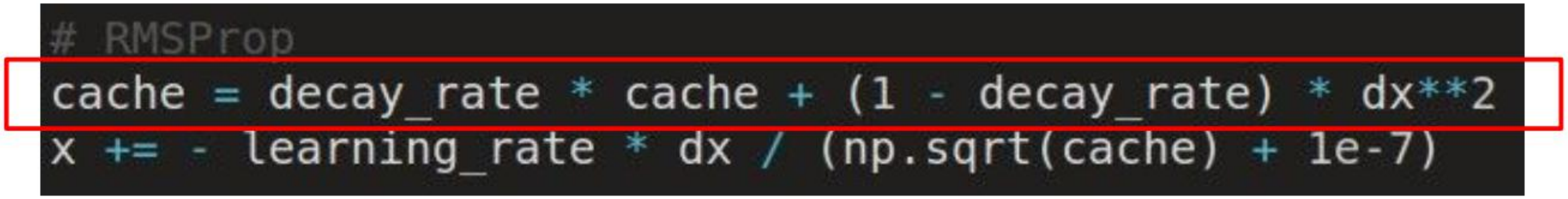
Q1: What happens with AdaGrad?
Q2: What happens to the step size over long time?

RMSProp update

[Tieleman and Hinton, 2012]

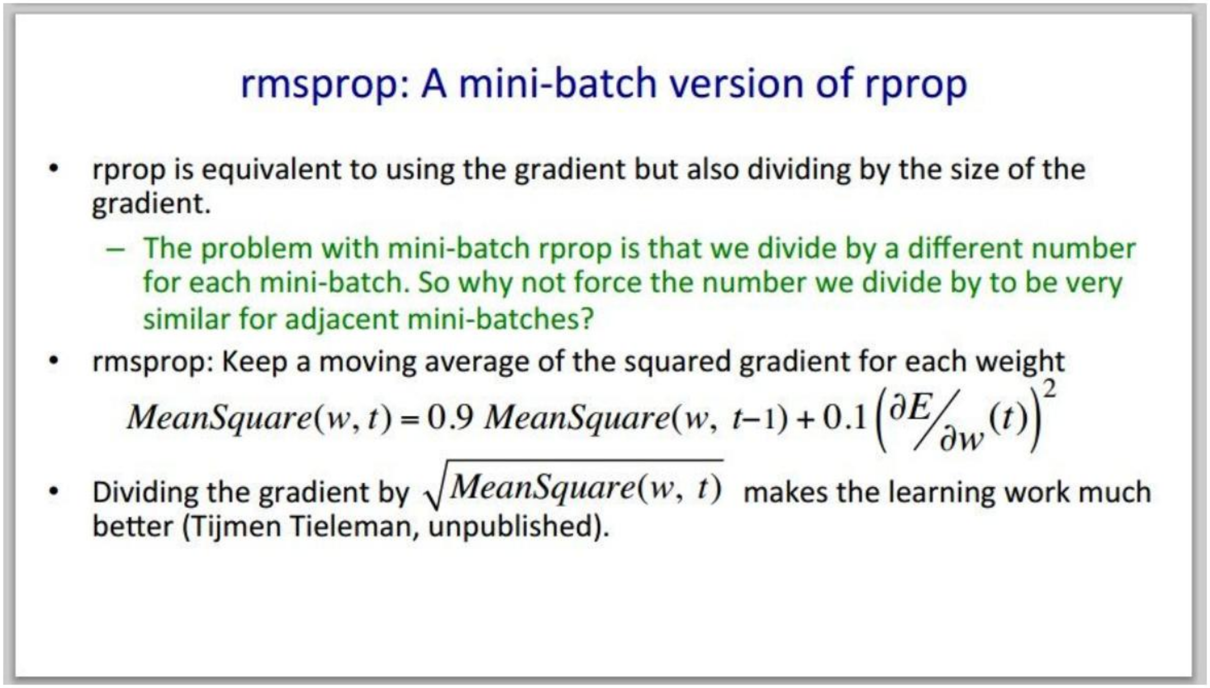


```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



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

RMSProp



rmsprop: A mini-batch version of rprop

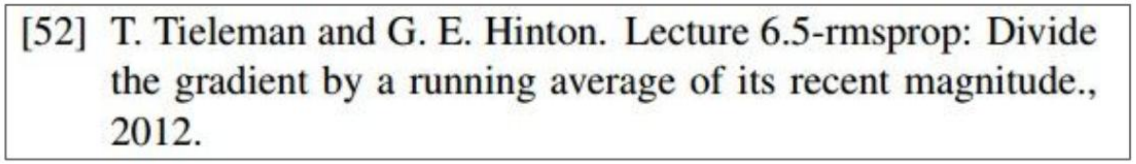
- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving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

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

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

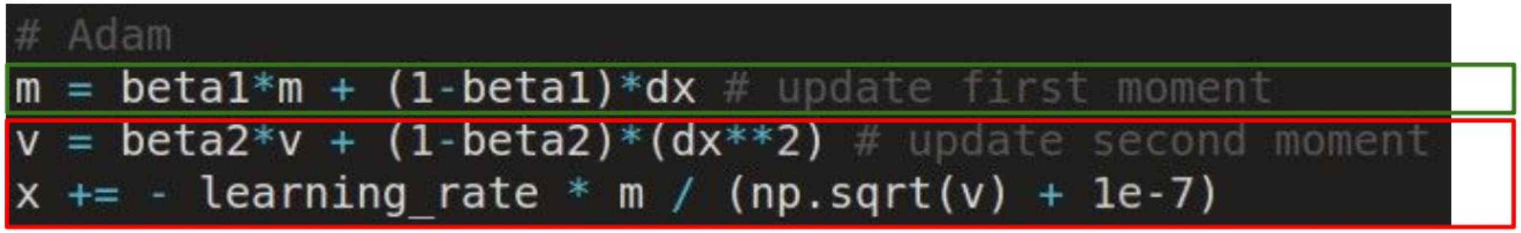
Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:



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

Adam update

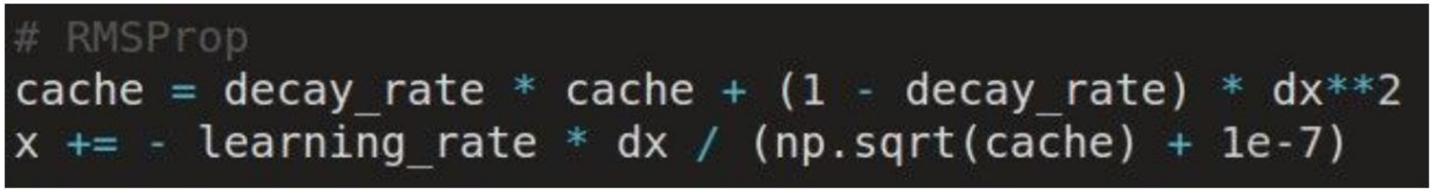


```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

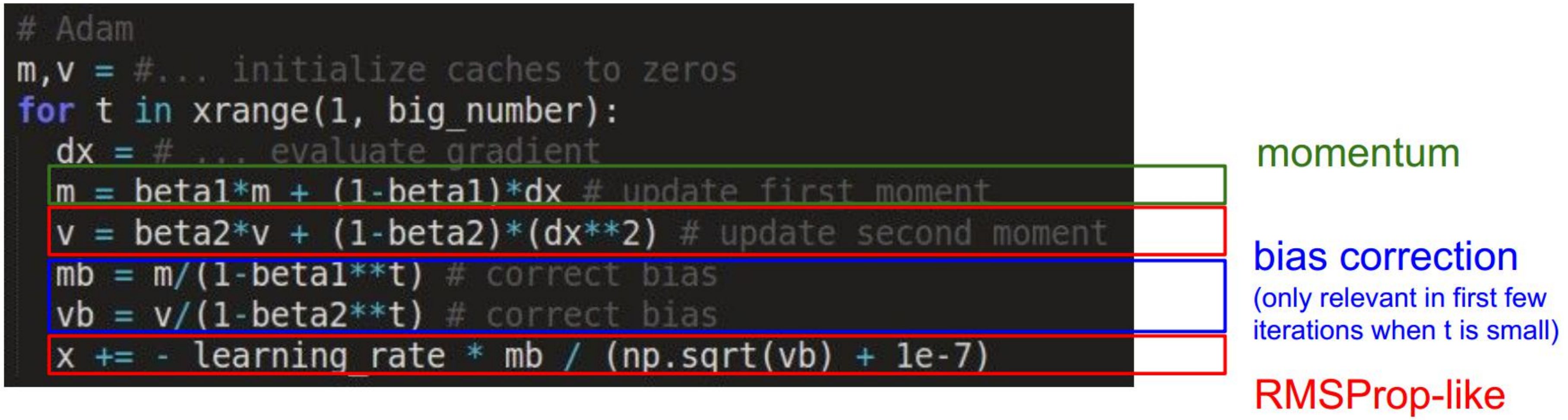
RMSProp-like

Looks a bit like RMSProp with momentum



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

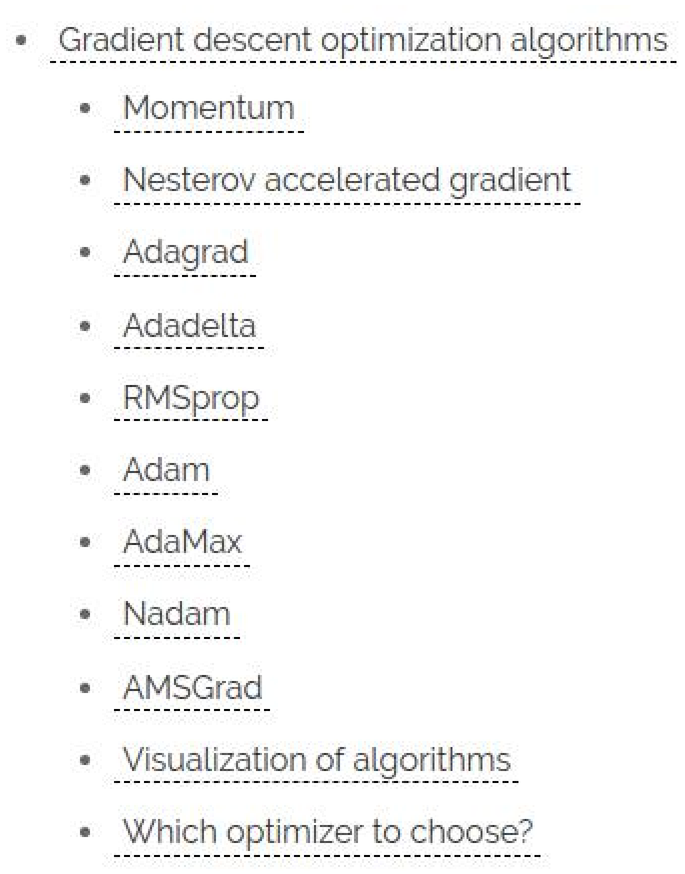
Adam update



```
# 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)
```

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

Overview of gradient descent optimization algorithms



- Gradient descent optimization algorithms
  - Momentum
  - Nesterov accelerated gradient
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam
  - AdaMax
  - Nadam
  - AMSGrad
  - Visualization of algorithms
  - Which optimizer to choose?

Link: http://ruder.io/optimizing-gradient-descent/

Which Optimizer to Use?

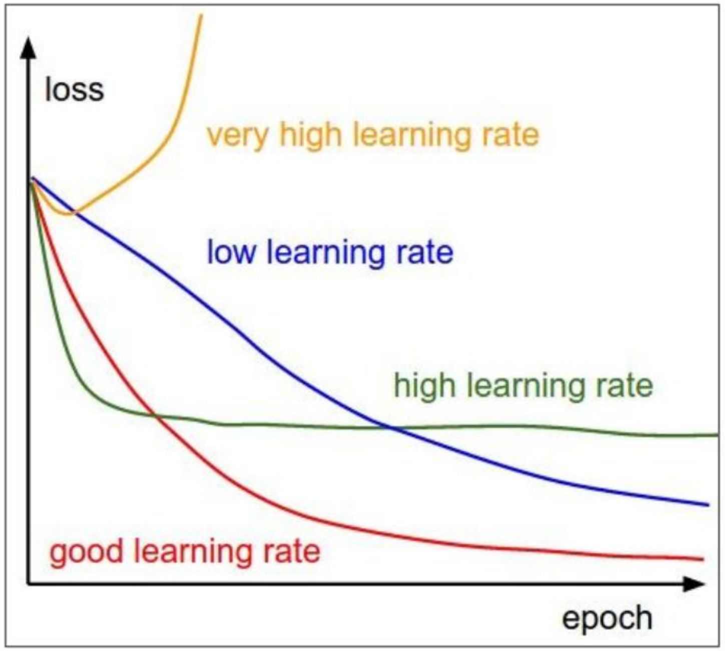
- If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods.
- RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances.
- Experiments show that bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.
- Interestingly, many recent papers use SGD without momentum and a simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima.
- If you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods

Outline

- Activation Functions
- Parameter update schemes
- **Learning rate schedules**
- Generalization with Dropout
- Application of ConvNets

Learning rate

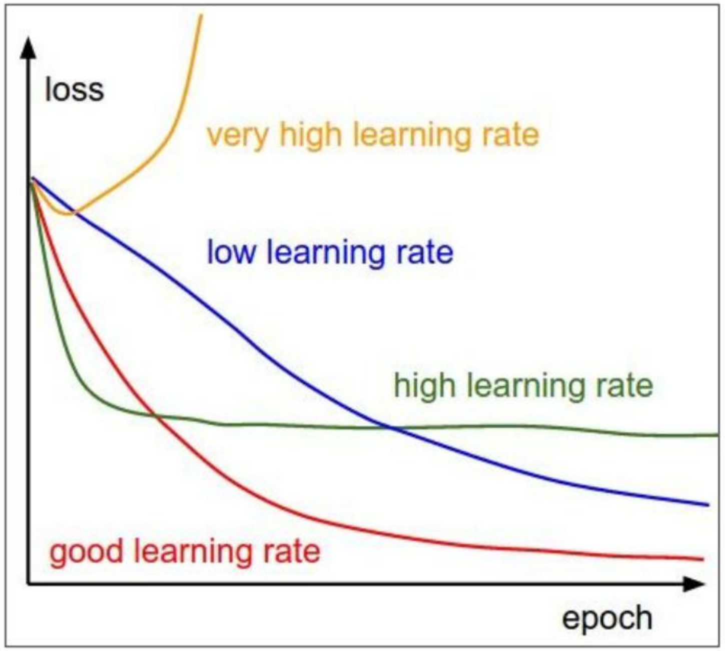
- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.



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

Learning rate

- 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:

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

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

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

Q1: 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

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

- 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

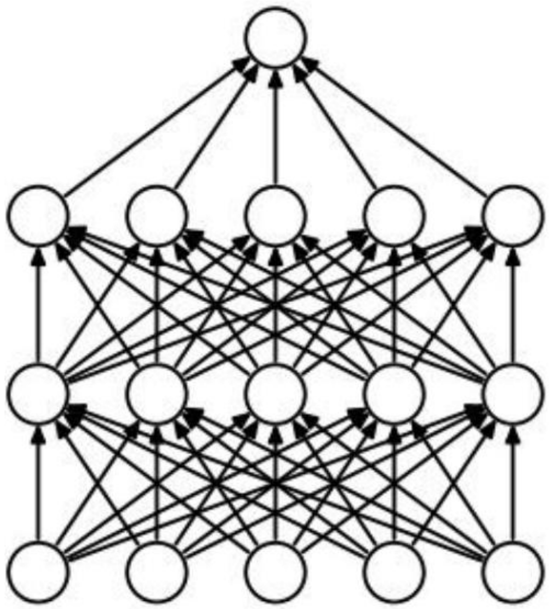
- **Usually works very well in full batch, deterministic mode.**
  - i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely
- **Does not transfer very well to mini-batch setting**.
  - Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active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

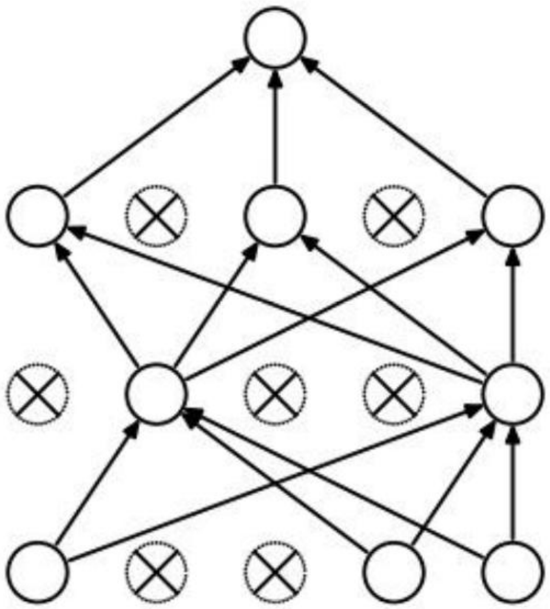
Outline

- Activation Functions
- Parameter update schemes
- Learning rate schedules
- **Generalization with Dropout**
- Application of ConvNets

Regularization: Dropout

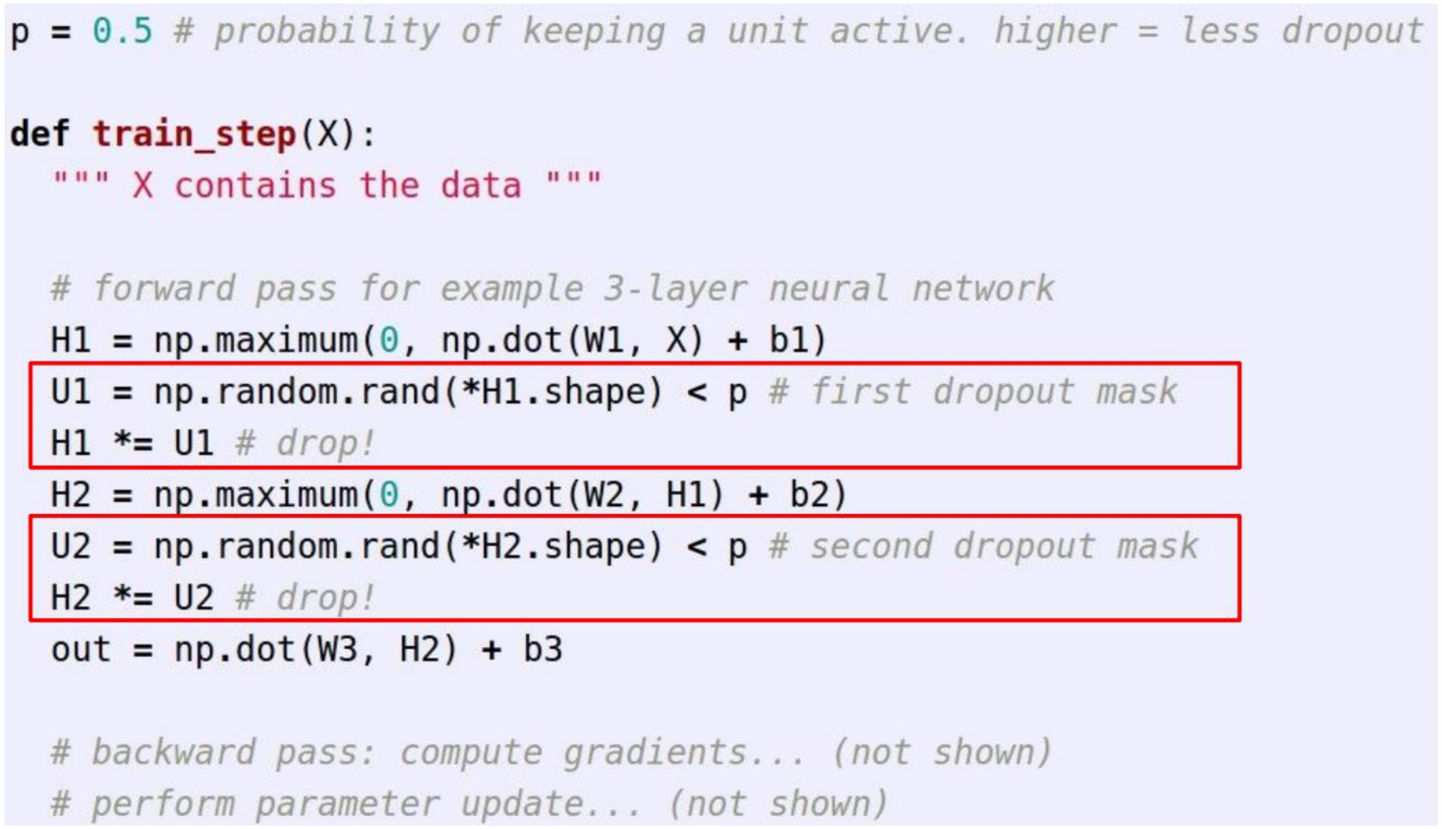
- “randomly set some neurons to zero in the forward pass”


(a) Standard Neural Net


(b) After applying dropout.

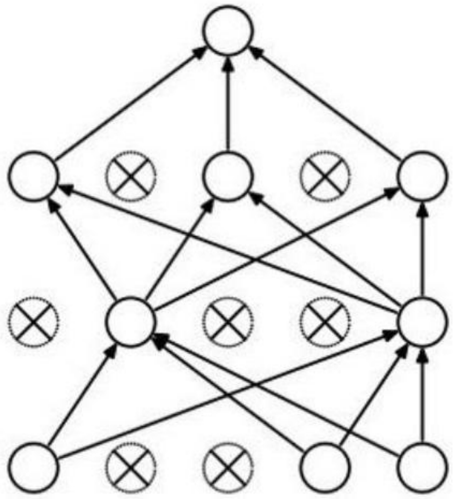
[Srivastava et al., 2014]

Example



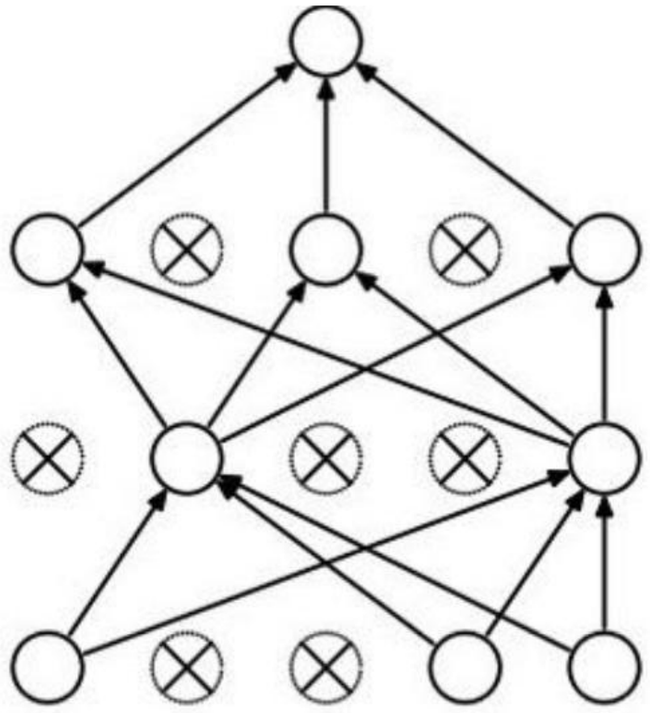
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

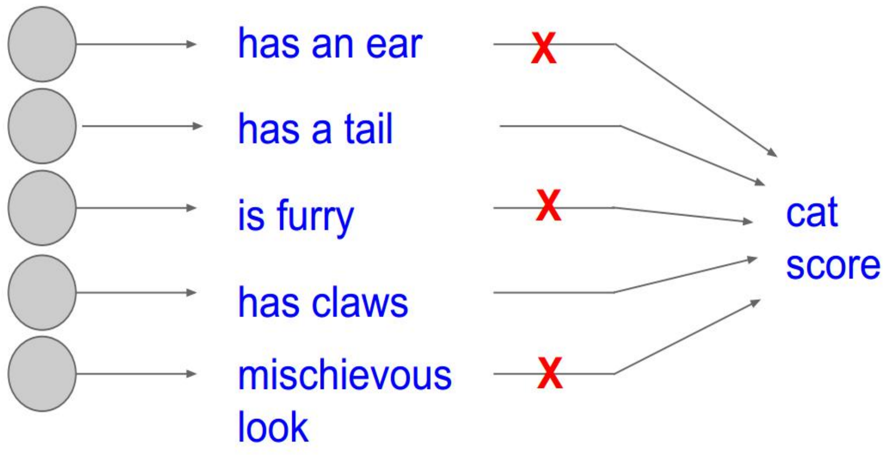


Regularization: Dropout

- Wait a second… How could this possibly be a good idea？

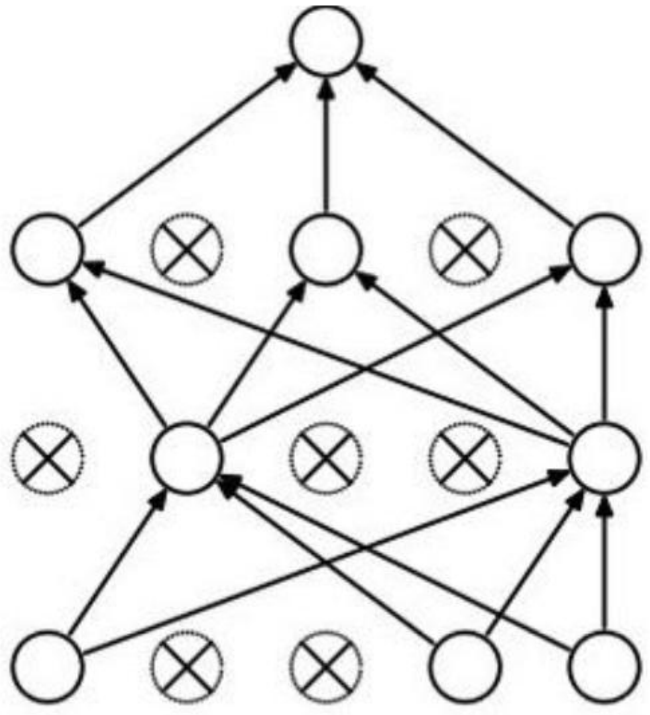


Forces the network to have a redundant representation.



Regularization: Dropout

- Wait a second… How could this possibly be a good idea？

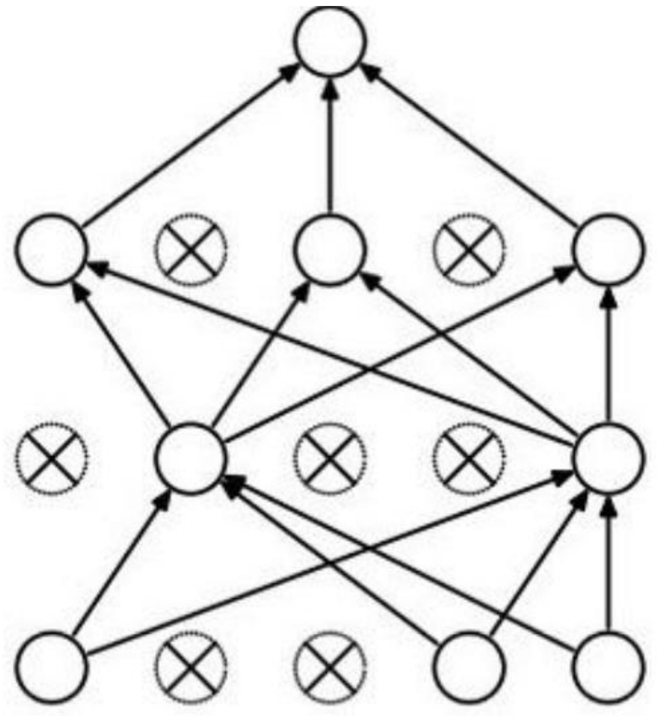


Another interpretation:

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

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

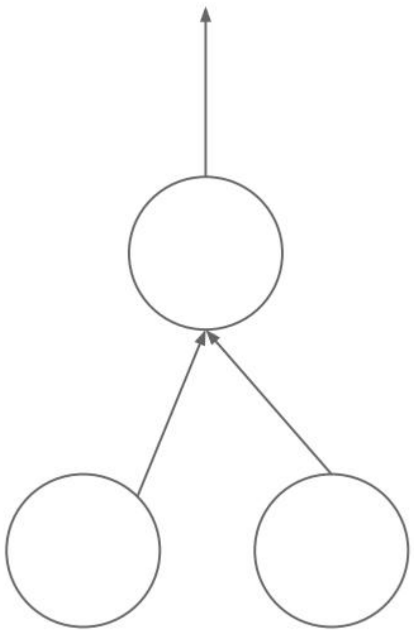
At test time….



- **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….

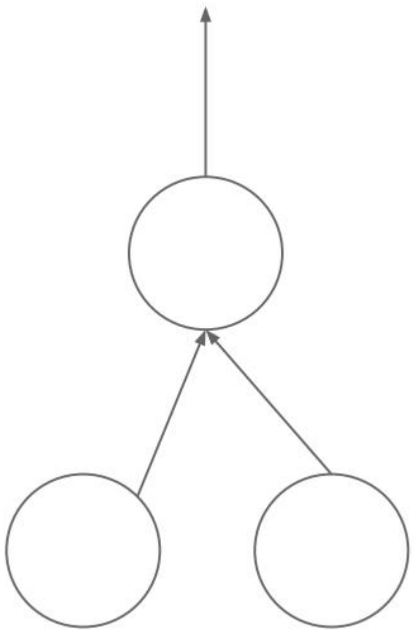
- Can in fact do this with a single forward pass! (approximately)
- Leave all input neurons turned on (no dropout).



(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).

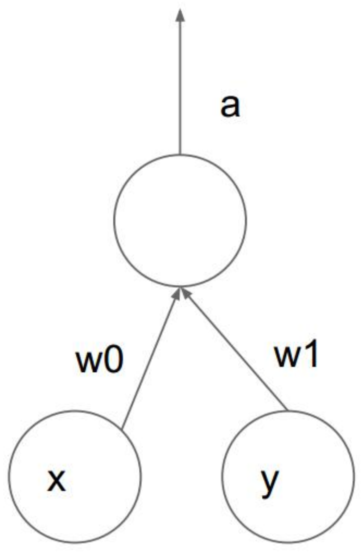


Q: Suppose that with all inputs present at test time the output of this neuron is x.

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).



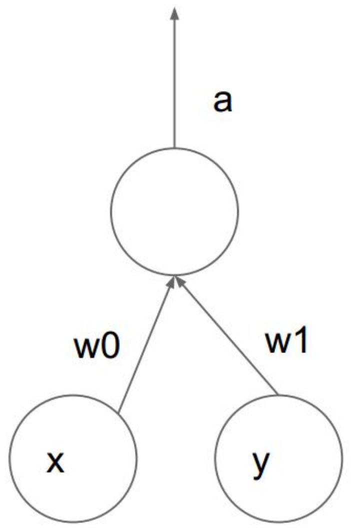
during test: **a = w0*x + w1*y**

during train:

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

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)**

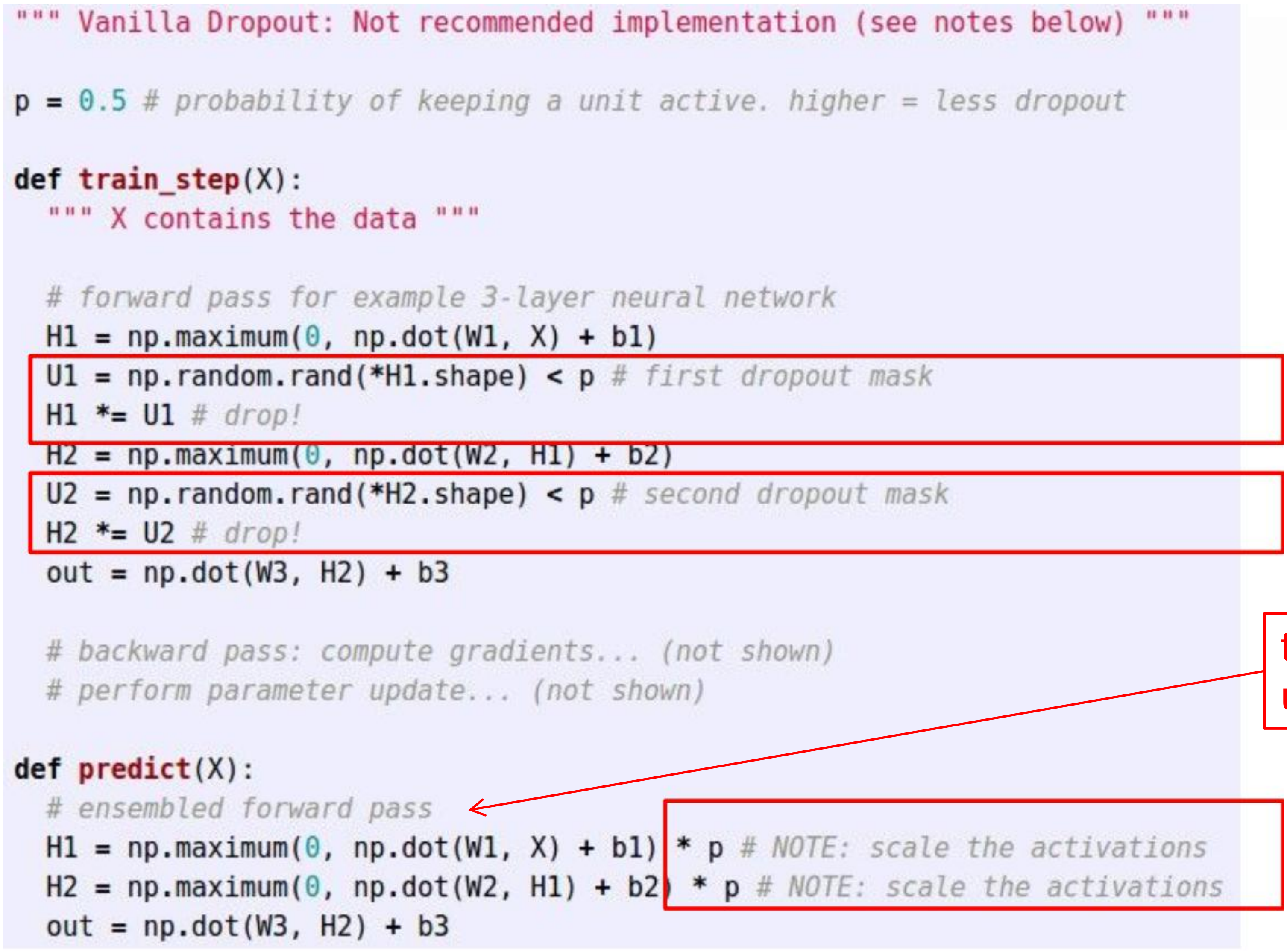
With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

We can do something approximate analytically?

```
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



```
""" 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

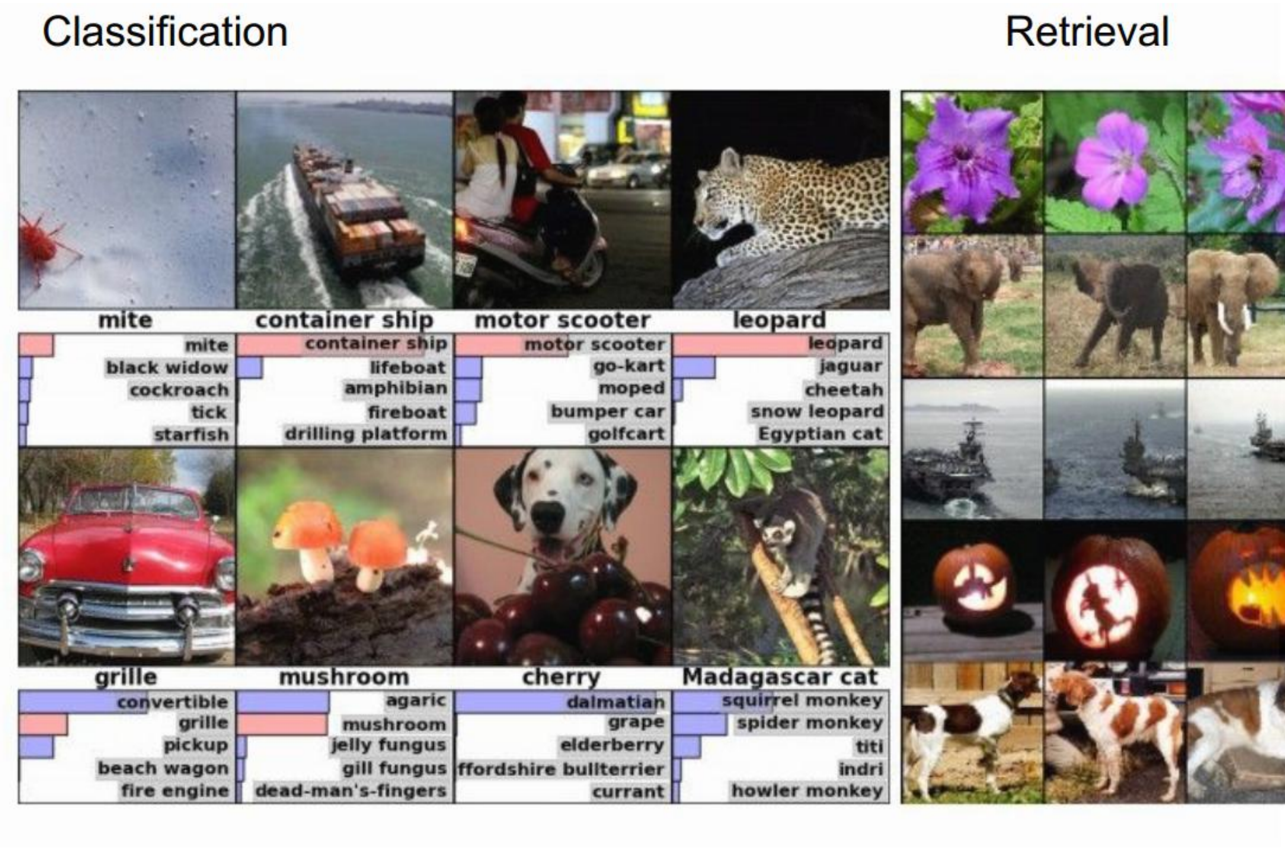
test time is unchanged!

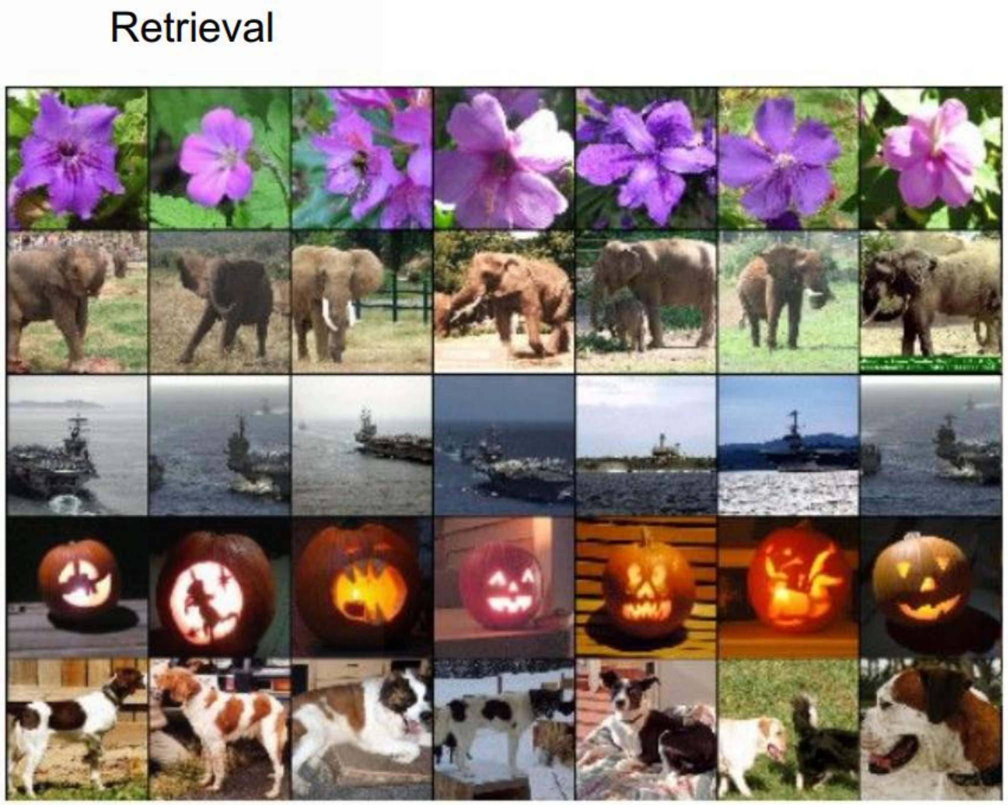
scale at test time

Outline

- Activation Functions
- Parameter update schemes
- Learning rate schedules
- Generalization with Dropout
- **Application of ConvNets**

Fast-forward to today: ConvNets are everywhere

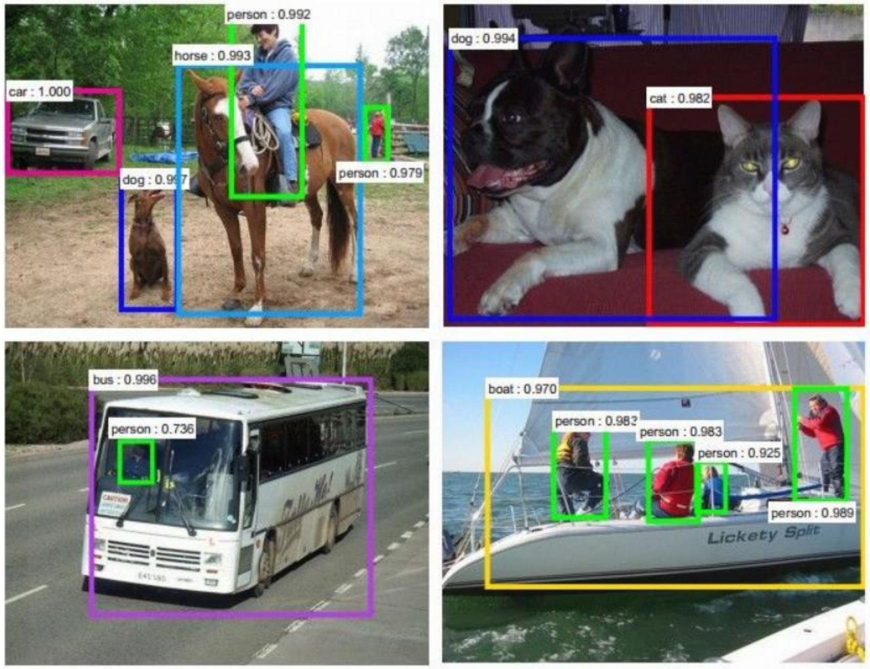




[Krizhevsky 2012]

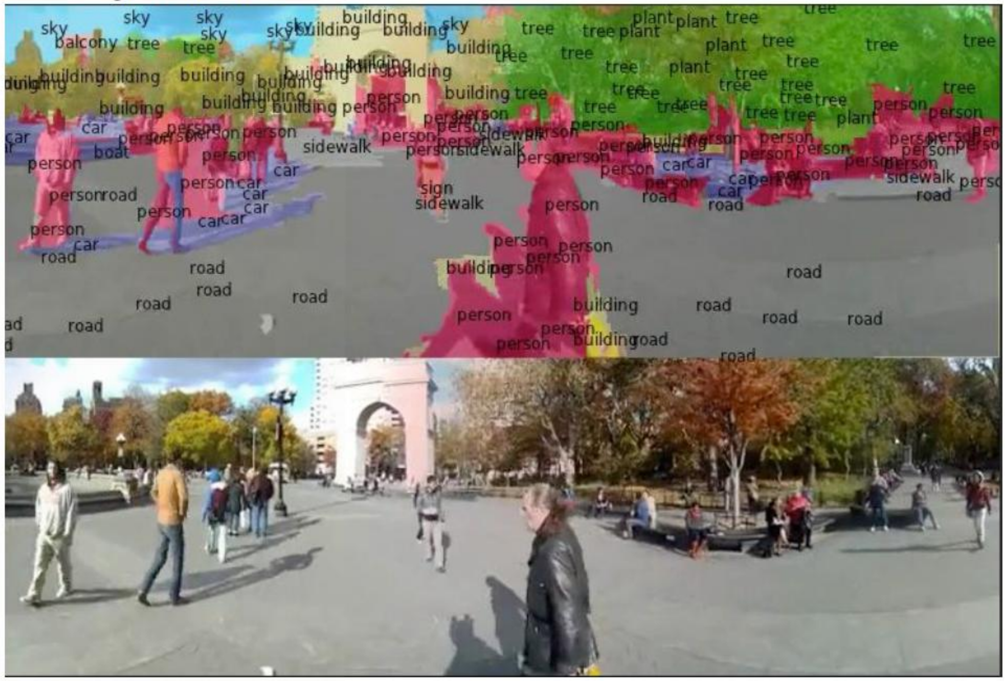
Fast-forward to today: ConvNets are everywhere

Detection



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

Segmentation



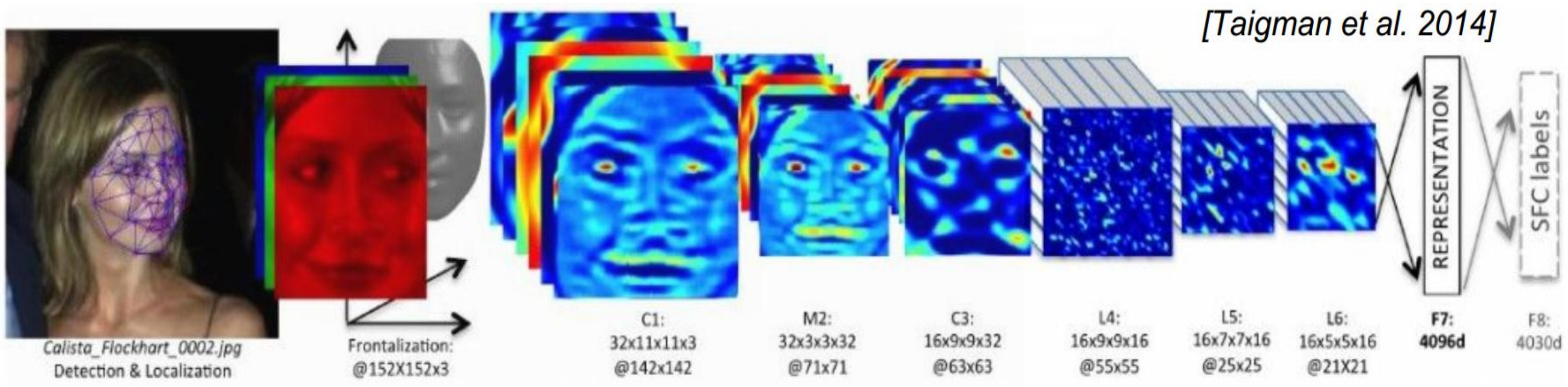
[Farabet et al., 2012]

Fast-forward to today: ConvNets are everywhere

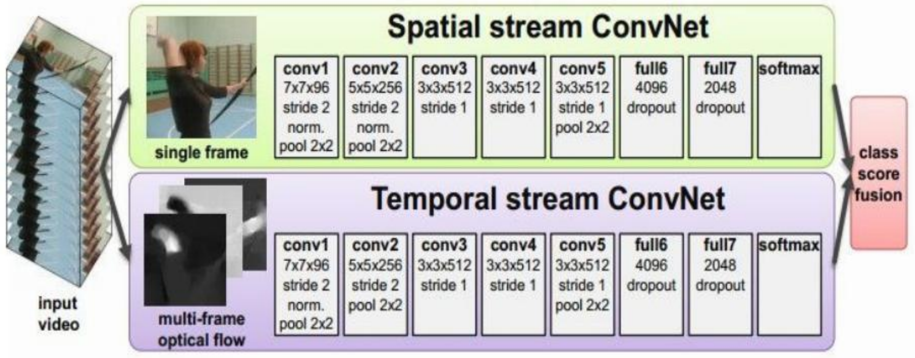
self-driving cars

NVIDIA Tegra X1

Fast-forward to today: ConvNets are everywhere



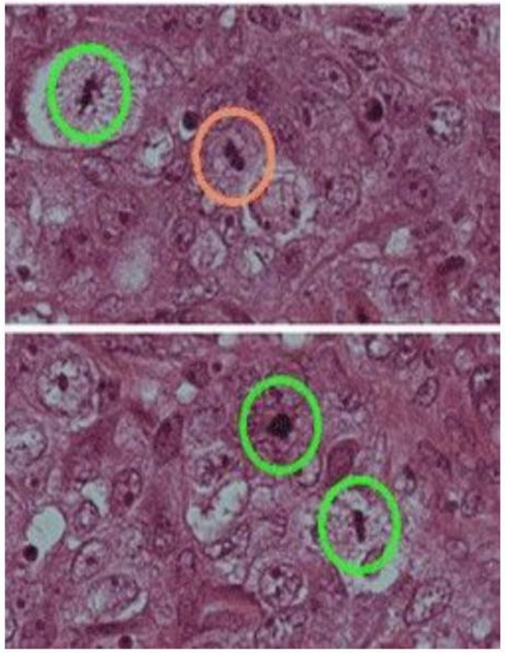
[Taigman et al. 2014]



[Simonyan et al. 2014]

[Goodfellow 2014]

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

Whale recognition, Kaggle Challenge

Mnih and Hinton, 2010

Fast-forward to today: ConvNets are everywhere

A person riding a motorcycle on a dirt road.

Two dogs play in the grass.

A skateboarder does a trick on a ramp.

A dog is jumping to catch a frisbee.

A group of young people playing a game of frisbee.

Two hockey players are fighting over the puck.

A little girl in a pink hat is blowing bubbles.

A refrigerator filled with lots of food and drinks.

A herd of elephants walking across a dry grass field.

A close up of a cat laying on a couch.

A red motorcycle parked on the side of the road.

A yellow school bus parked in a parking lot.

Q & A