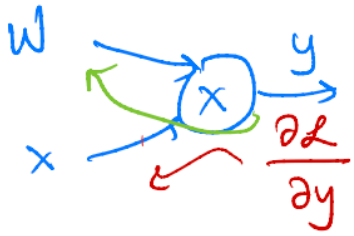
Lecture 8: Initialization and Batchnorm

Announcements:

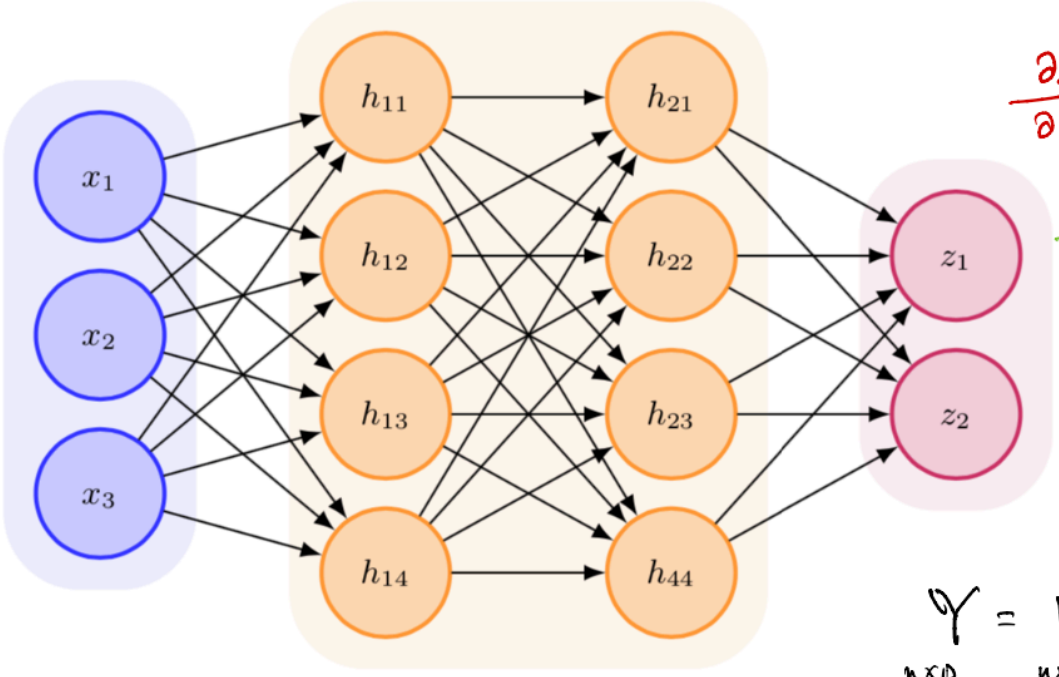
- HW #3 is due, **Wednesday, Feb 2**, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

Back to backpropagation

$y = Wx$



know $\frac{\partial \mathcal{L}}{\partial y}$



$$\frac{\partial \mathcal{L}}{\partial x} = W^T \frac{\partial \mathcal{L}}{\partial y}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y} x^T$$

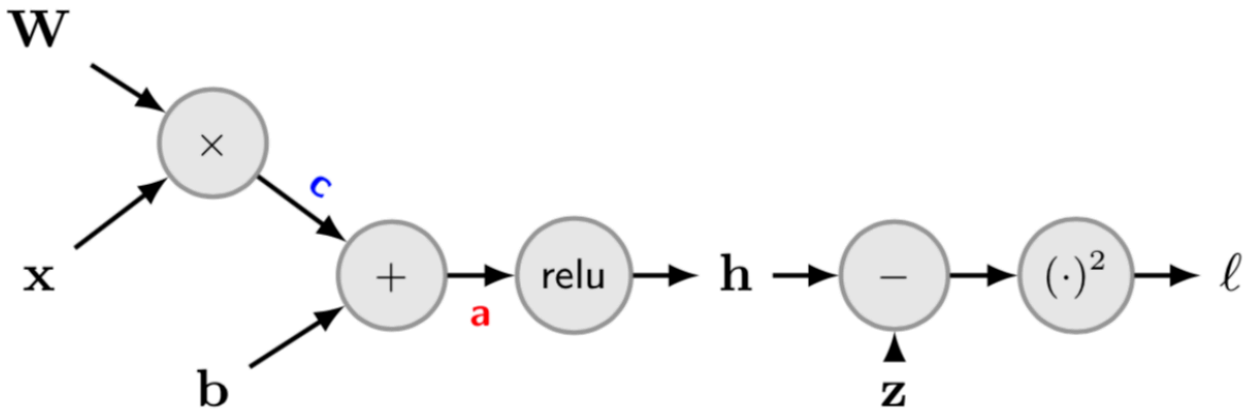
$$\underset{n \times p}{Y} = \underset{n \times m}{W} \underset{m \times p}{X}$$

know $\frac{\partial \mathcal{L}}{\partial Y}$

$$\underset{m \times p}{\frac{\partial \mathcal{L}}{\partial X}} = W^T \frac{\partial \mathcal{L}}{\partial Y}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial Y} X^T$$

Backpropagation for a neural network layer



Backpropagation: neural network layer (cont.)

Applying the chain rule, we have:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{a}} &= \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}} \\
\frac{\partial \ell}{\partial \mathbf{c}} &= \frac{\partial \ell}{\partial \mathbf{a}} \\
\frac{\partial \ell}{\partial \mathbf{x}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}}
\end{aligned}$$

A few notes:

- For $\frac{\partial \mathbf{c}}{\partial \mathbf{x}}$, see example in the Tools notes.
- Why was the chain rule written right to left instead of left to right? This turns out to be a result of our convention of how we defined derivatives (using the "denominator layout notation") in the linear algebra notes.
- Though maybe not the most satisfying answer, you can always check the order of operations is correct by considering non-square matrices, where the dimensionality must be correct.

Now we have the gradients, we can do gradient descent

With the gradients of the parameters, we can go ahead and now apply our learning algorithm, gradient descent.

However, we'll soon find that if we do this naively, performance won't be great **for neural networks** (you'll see this on HW #3).

There are many other important considerations we now need to consider before we can train these networks adequately.

Regularizations and training neural networks

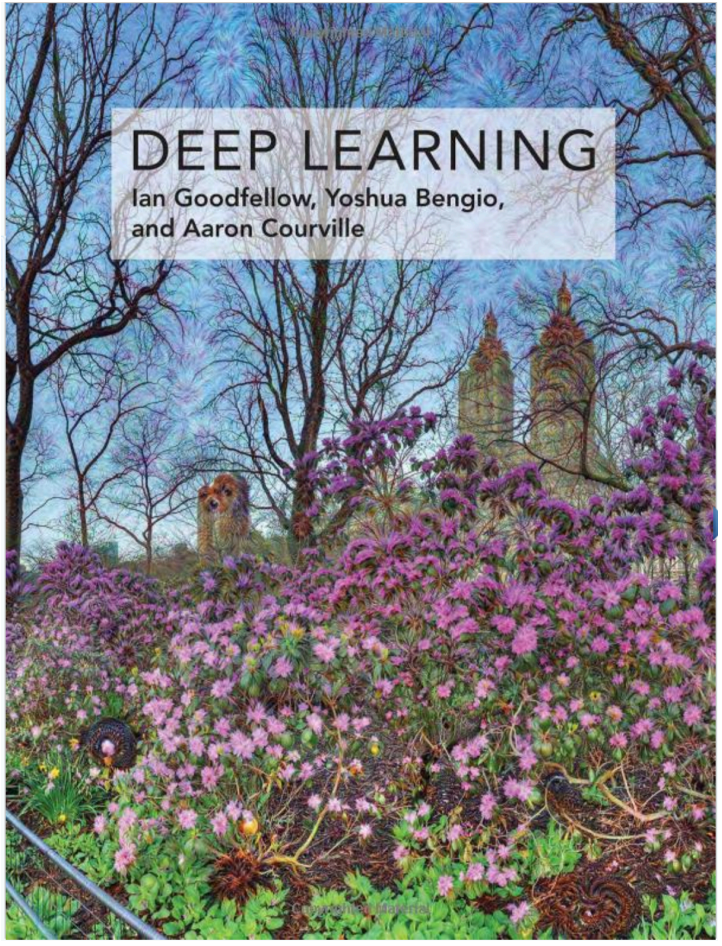
In this lecture, we'll talk about specific techniques that aid in training neural networks. This lecture will focus on a few topics that are relevant for training neural networks well. The next lecture will then focus on optimization.

- Weight initialization in neural networks
- Batch normalization
- Regularizations
  - L1 + L2 normalization
  - Dataset augmentation
  - Model ensembles / Bagging
  - Dropout

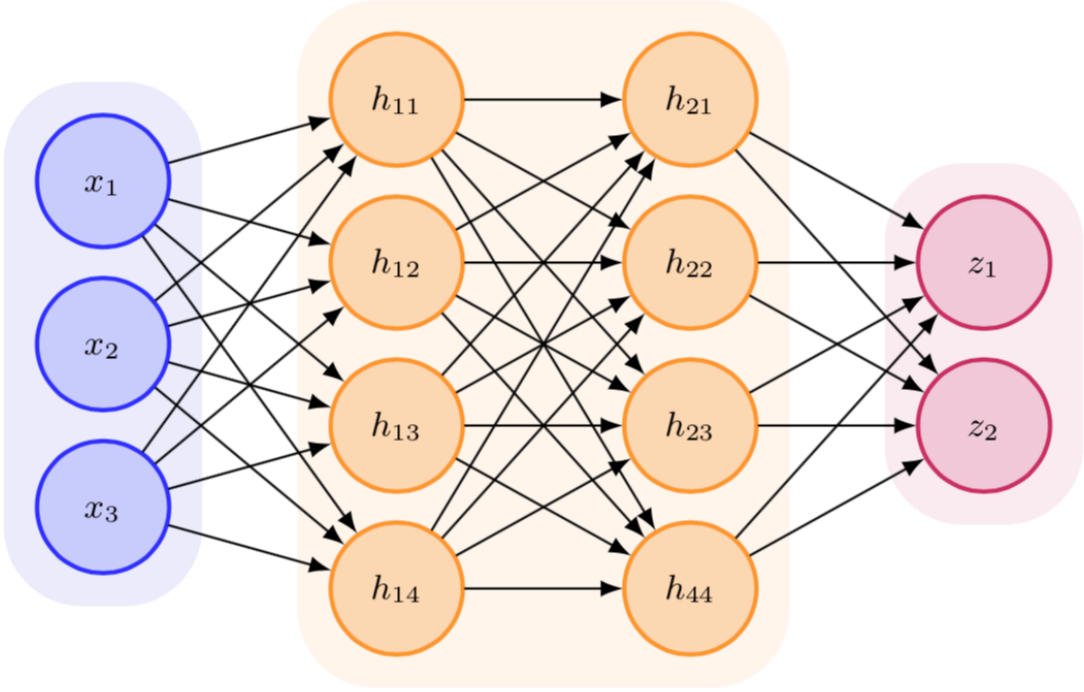
Regularizations and training neural networks

Reading:

Deep Learning, 7 (intro), 7.1, 7.2, 7.4, 7.5, 7.7, 7.9, 7.11, 7.12 (skim), 8.7.1



Initializations



Initializations matter.

Small random weight initialization

How about small random weight initializations?

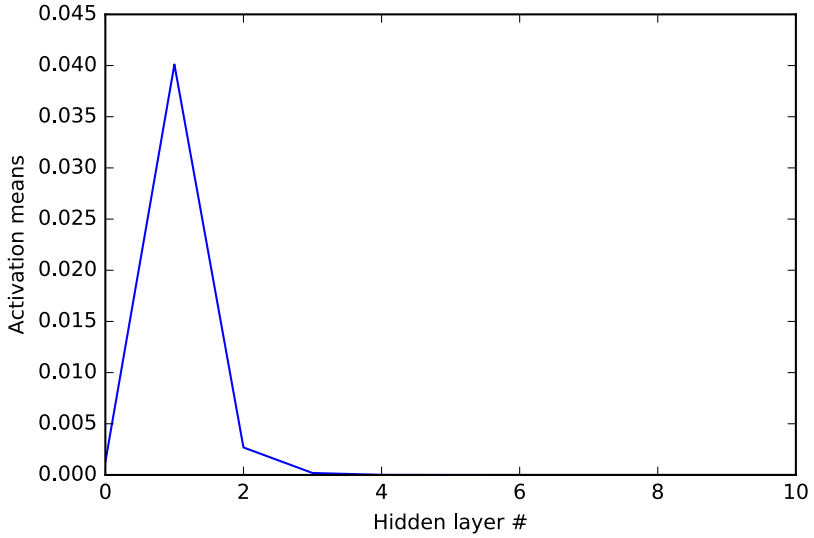
The thought is that we don't set them large enough to bias training.

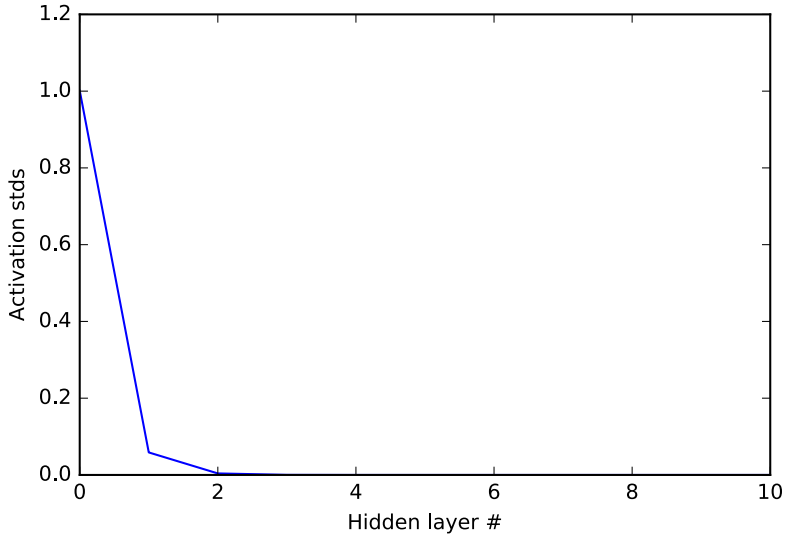
```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize small weights
    W = np.random.randn(layer_sizes[i], H.shape[0]) * 0.01
    Z = np.dot(W, H)
    H = Z * (Z > 0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

$W_{ij} \in N(0, \quad 0.01)$

Small random weight initialization

Empirically, these initializations cause all activations to decay to zero.





standard deviation

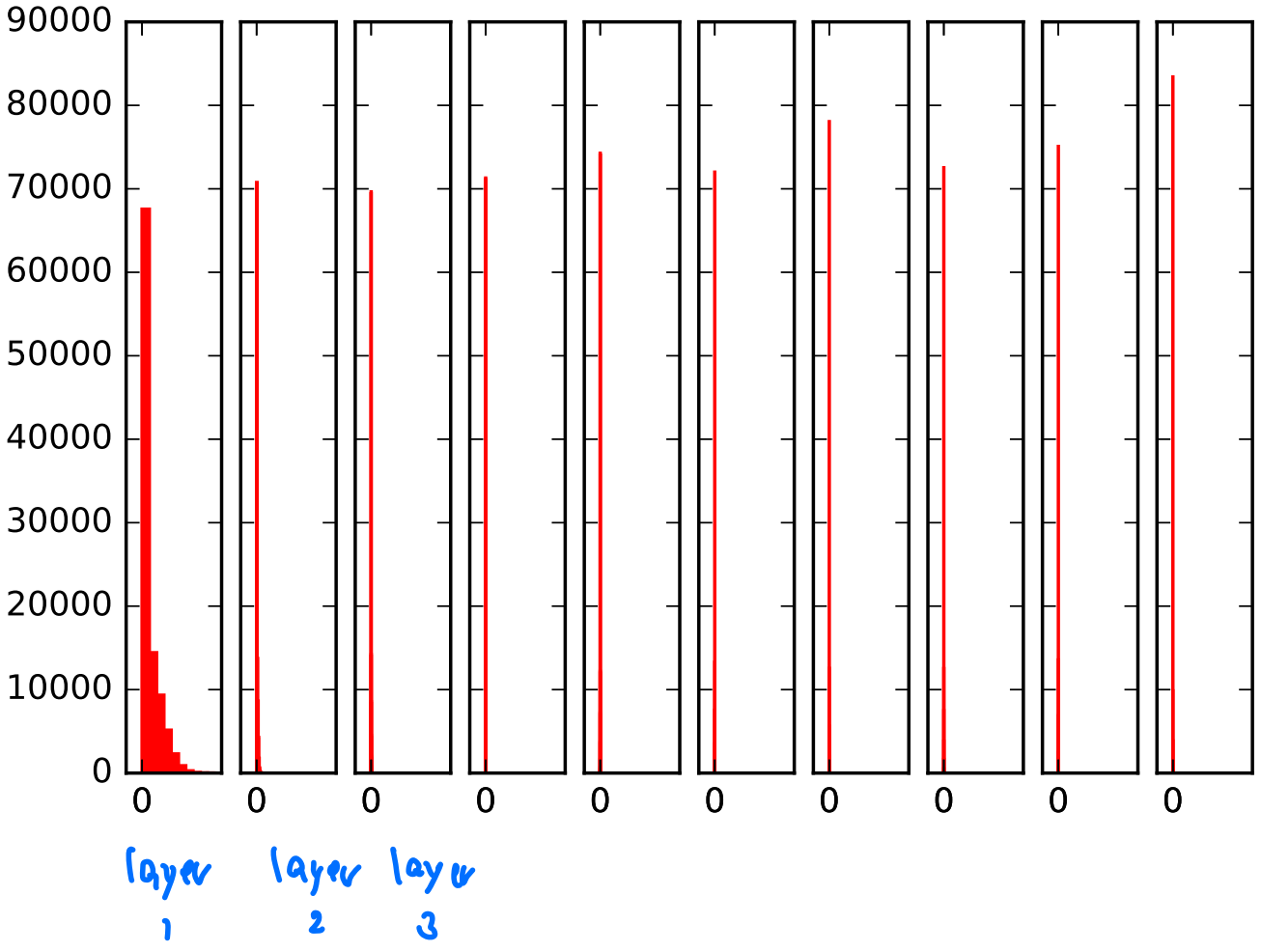
$h_1 = \text{relu}(w_1 x)$

$h_2 = \text{relu}(w_2 h_1)$

small $w \rightarrow h_1, h_2 \sim 0$

Small random weight initialization

Distribution of each layer's activations:

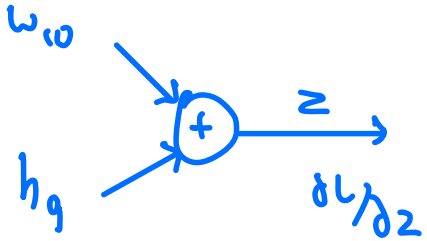


Small random weight initialization

What about backpropagation with this small weight initialization?

Think about the value of: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$

$z = w_{10} h_9$



$$\frac{\partial L}{\partial w_{10}} = \frac{\partial L}{\partial z} h_9^T$$

↳ all zeroes

$$\frac{\partial L}{\partial w_{10}} = 0 \Rightarrow \text{no training} //$$

Small random weight initialization

What about backpropagation with this small weight initialization?

Think about the value of: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$

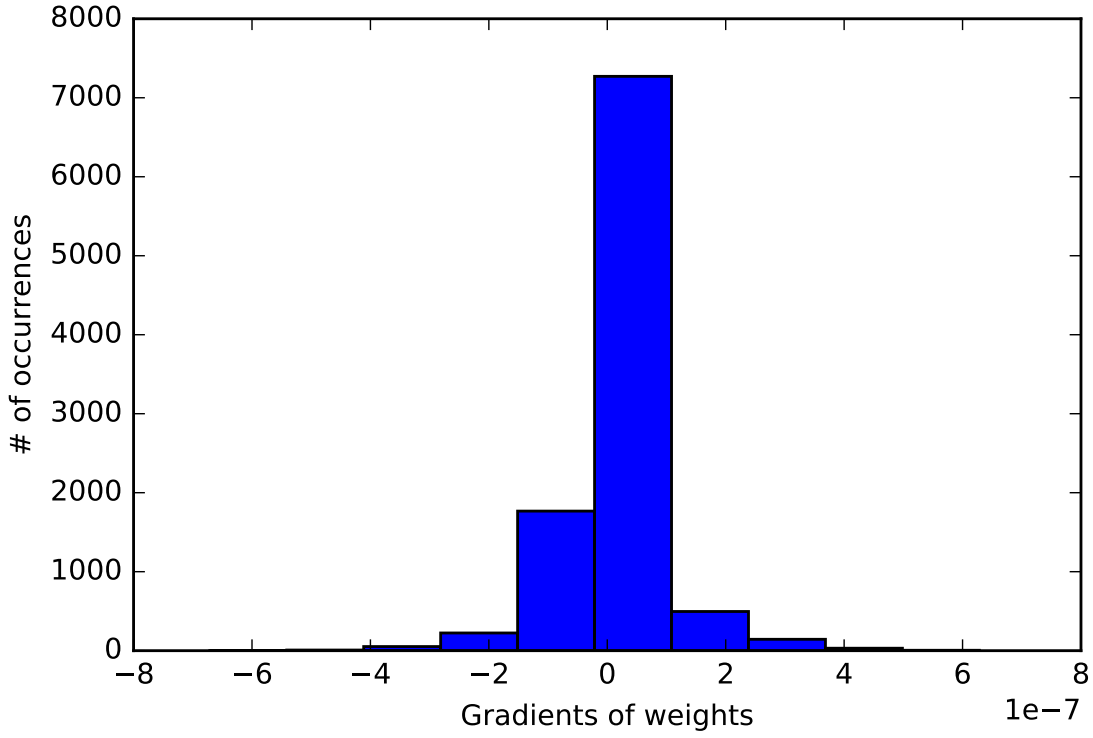
```
# Now backprop
dLdH = 100*np.random.randn(100, 1000) # 1000 losses for examples
loss_grads = [dLdH]
grads = []

for i in np.flip(np.arange(1,11), axis=0):
    loss_grad = loss_grads[-1]
    dLdZ = loss_grad * (Z[i] > 0)
    grad_W = np.dot(dLdZ, Hs[i-1].T)
    grad_h = np.dot(Ws[i-1].T, dLdZ)
    loss_grads.append(grad_h)
    grads.append(grad_W)
grads = list(reversed(grads))
loss_grads = list(reversed(loss_grads))
```

Small random weight initialization

What about backpropagation with this small weight initialization?

Think about the value of: $\frac{\partial \ell}{\partial \mathbf{W}}$

Gradients for the last layer with respect to **W**:



Small random weight initialization

In practice, small weight initializations may be appropriate for smaller neural networks.

Large random weight initialization

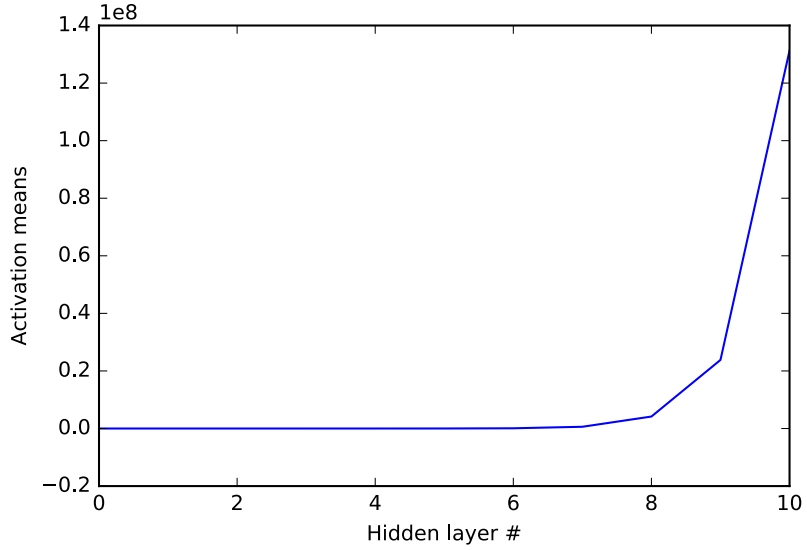
How about larger random weight initializations?

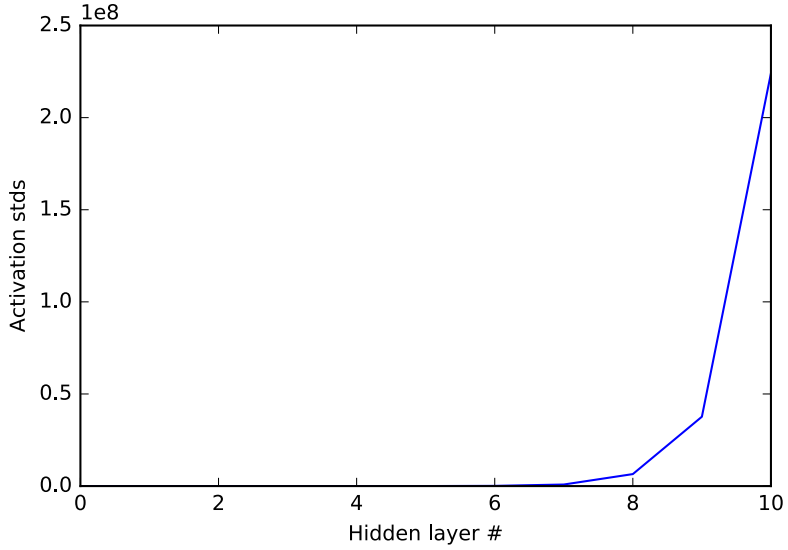
```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize large weights
    W = np.random.randn(layer_sizes[i], H.shape[0]) * 1
    Z = np.dot(W, H)
    H = Z * (Z > 0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

$W_{ij} \sim N(0,1)$

Large random weight initialization

Empirically, these cause the units to explode.





What happens to the gradients?

Again, this is a problem. Activity is large. $\frac{\partial L}{\partial w_{10}} = \frac{\partial L}{\partial z} \cdot h_9^T$ → too large

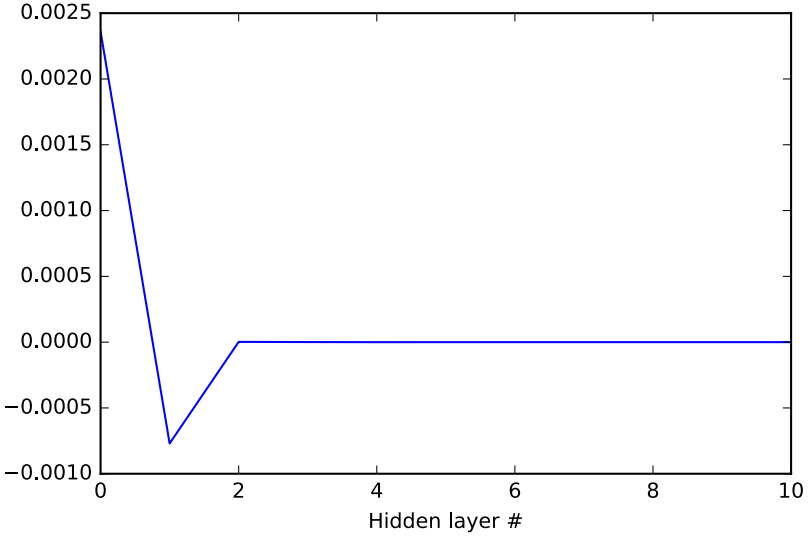
No idea where we would end up → may not be minimum.

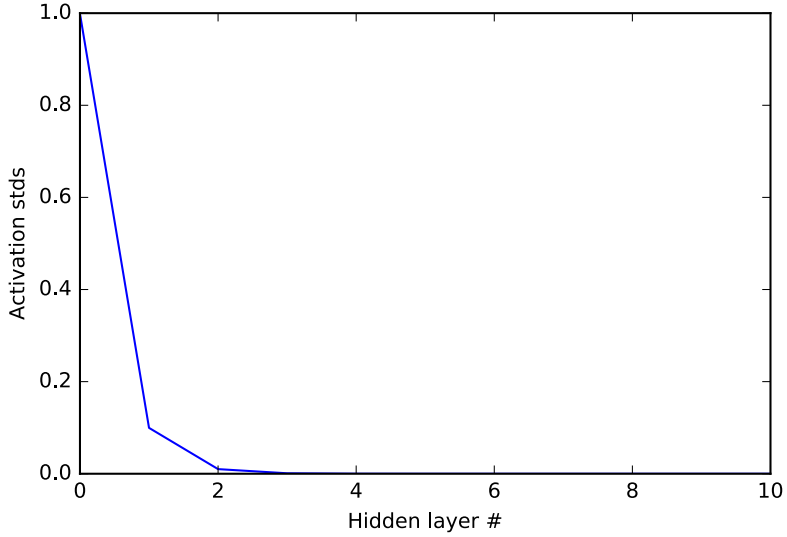
Will this problem occur for other activations?

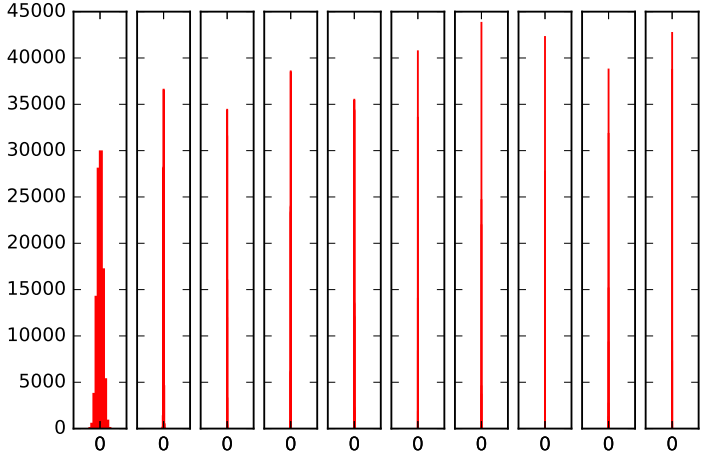
What happens when we use the tanh() activation with small initialization?

Will this problem occur for other activations?

What happens when we use the tanh() activation with small initialization?

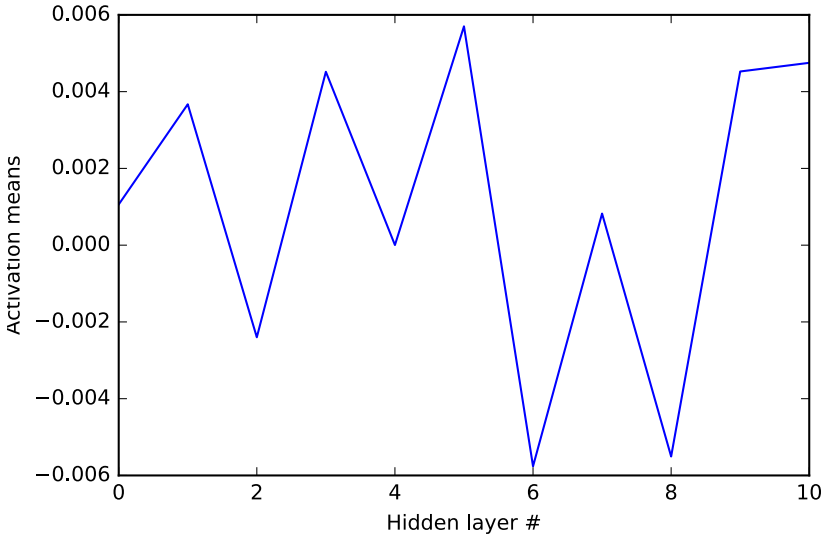


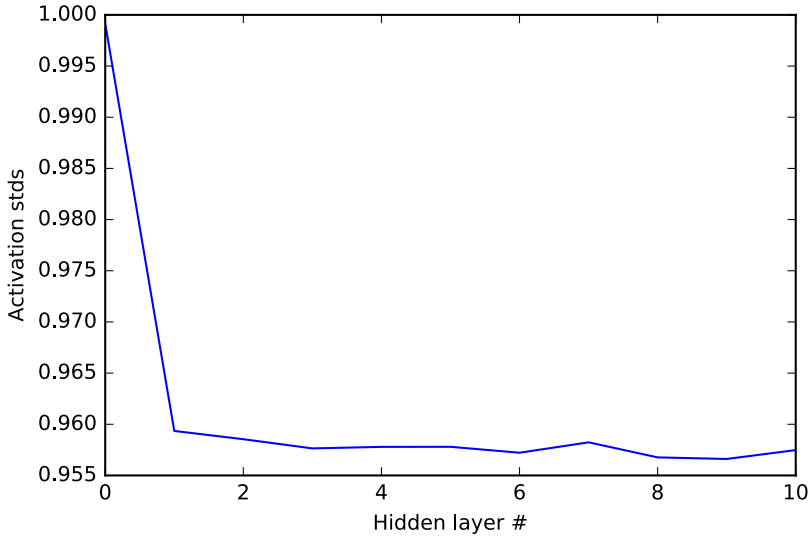


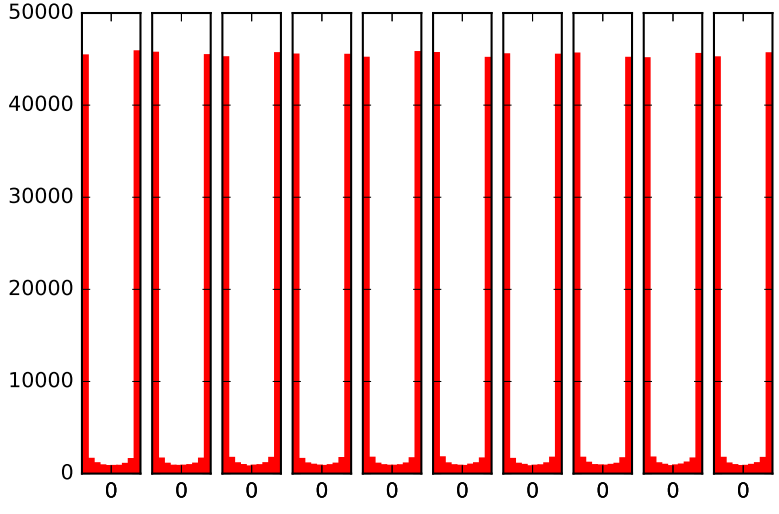


Will this problem occur for other activations?

What happens when we use the tanh() activation with large initialization?







-1 or 1

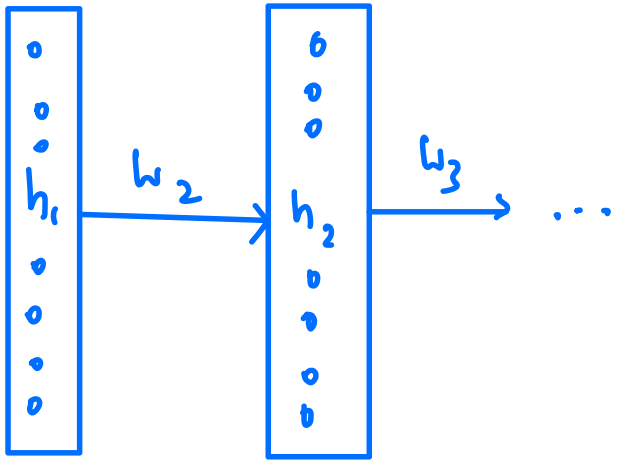
If not small and not large initialization, then what?

The next idea is to try an intermediate initialization.

But what intermediate value?

Xavier initialization

One initialization is from Glorot and Bengio, 2011, referred to commonly as the Xavier initialization. It intuitively argues that the variance of the units across all layers ought be the same, and that the same holds true for the backpropagated gradients.



If initialization

→ too low

var → 0 as layers increase

$\nabla L \to 0$ as well

→ too high → both → ∞.

We want $\text{var}(h_1) \approx \text{var}(h_2) \approx \ldots \approx \text{var}(h_i)$

Same in backpropagation $\text{var}(\nabla_{W_i} L) \approx \text{var}(\nabla_{W_j} L)$

ONLY DURING THE FIRST STEP!

Xavier initialization

For simplicity, we assume the input has equal variance across all dimensions. Then, each unit in each layer ought to have the same statistics. For simplicity we'll denote h_i to denote a unit in the ith layer, but the variances ought be the same for all units in this layer.

Concretely, the heuristics mean that

$$\mathrm{var}(h_i) = \mathrm{var}(h_j)$$

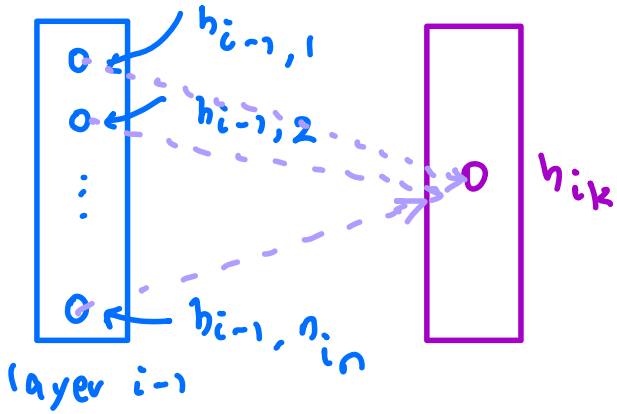
and

$$\mathrm{var}(\nabla_{h_i} J) = \mathrm{var}(\nabla_{h_j} J)$$

Xavier initialization (cont.)

If the units are linear, then

$$h_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} h_{i-1,j} \quad (1)$$



Further, if the w_{ij} and h_{i-1} are independent, and all the units in the $(i-1)$th layer have the same statistics, then using the fact that

$$\begin{aligned} \text{var}(wh) &= \mathbb{E}^2(w)\text{var}(h) + \mathbb{E}^2(h)\text{var}(w) + \text{var}(w)\text{var}(h) \\ &= \text{var}(w)\text{var}(h) \qquad \text{if } \mathbb{E}(w) = \mathbb{E}(h) = 0 \end{aligned}$$

then,

$$(1) \Rightarrow \quad \text{var}(h_i) = \text{var}(h_{i-1}) \cdot \sum_{j=1}^{n_{\text{in}}} \text{var}(w_{ij})$$

Assume, w, h_{i-1} have zero means //

All $h_{i,k}$ for all k are similar statistically.

DESIRED XAVIER CONDITION $(\text{var}(h_i) = \text{var}(h_{i-1}))$

$\Rightarrow \sum_{j=1}^{n_{in}} \text{var}(w_{ij}) = 1 \Rightarrow n_{in} \cdot \text{var}(w) = 1 \rightarrow \text{var}(w) = 1/n_{in}$

Xavier initialization

Now if the weights are identically distributed, then we get that for the unit activation variances to be equal,

$$\mathrm{var}(w_{ij}) = \frac{1}{n_{\mathrm{in}}}$$

for each connection j in layer i. The same argument can be made for the backpropagated gradients to argue that:

$$\mathrm{var}(w_{ij}) = \frac{1}{n_{\mathrm{out}}}$$

So, usually we take

$$\frac{n_{in} + n_{out}}{2}$$

$$\mathrm{var}(w_{ij}) = \frac{2}{n_{in} + n_{out}}$$

Xavier initialization (cont).

To incorporate both of these constraints, we can average the number of units together, so that

$$\text{var}(w_{ij}) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

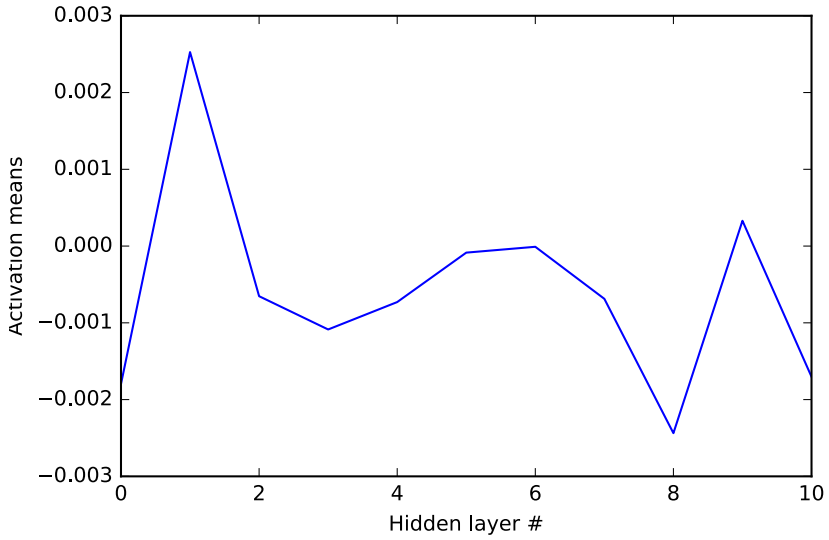
Hence, we can initialize each weight in layer i to be drawn from:

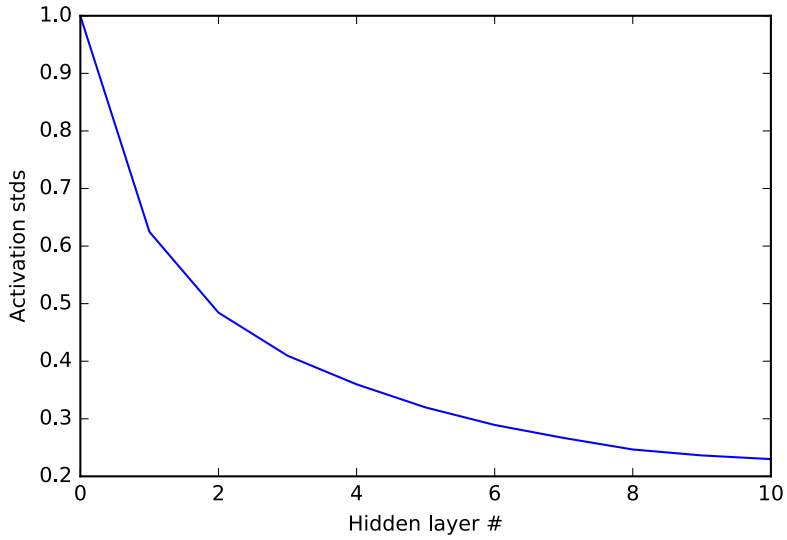
$$\mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

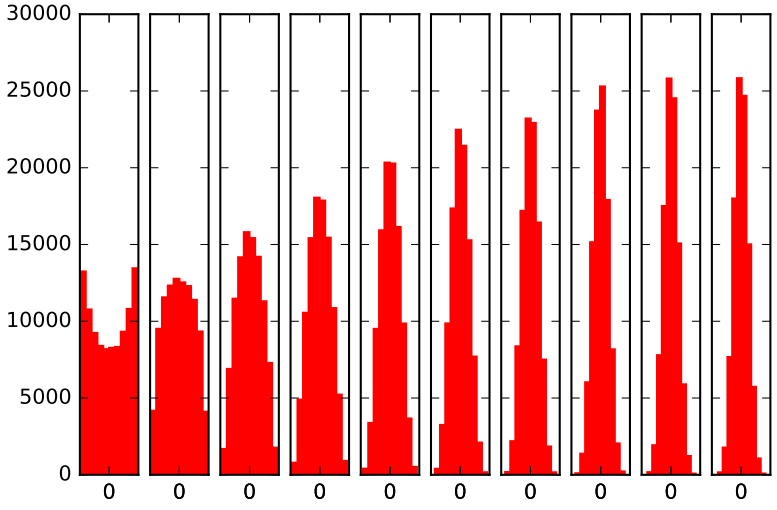
Xavier initialization with tanh

```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize Xavier
    W = np.random.randn(layer_sizes[i], H.shape[0]) * np.sqrt(2) / (np.sqrt(100 + 100))
    Z = np.dot(W, H)
    H = np.tanh(Z)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

Xavier initialization with tanh







variences not equal as all assumptions are realistic.

↑ not 0

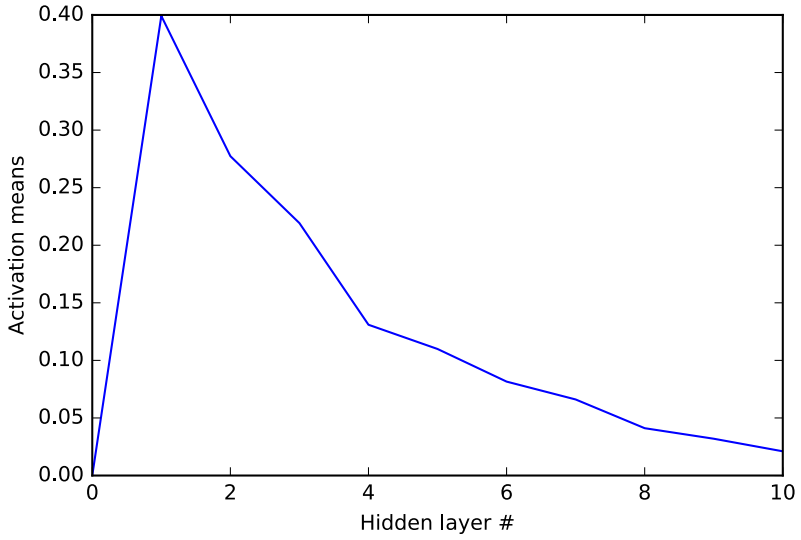
Xavier initialization

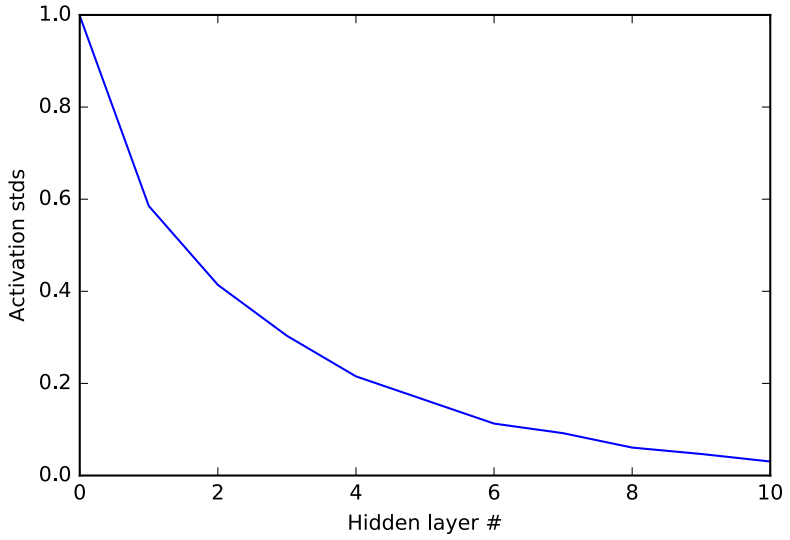
A few notes:

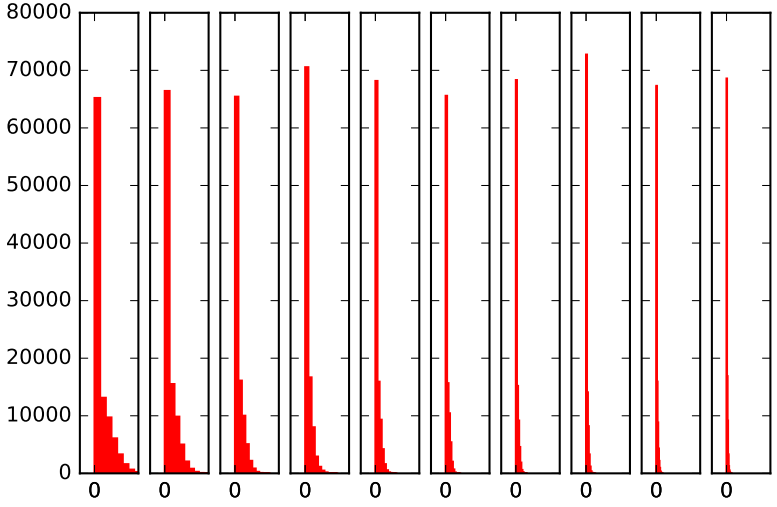
- If you use Caffe (and / or talk to other people), the Xavier initialization sets the variance to $1/n_{\text{in}}$. This is equivalent to the above form if $n_{\text{in}} = n_{\text{out}}$.
- However, the Xavier initialization (either one) typically leads to dying ReLU units, though it is fine with $\tanh$ units.
- He et al., 2015 suggest the normalizer $2/n_{\text{in}}$ when considering ReLU units. If linear activations prior to the ReLU are equally likely to be positive or negative, ReLU kills half of the units, and so the variance decreases by half. This motivates the additional factor of 2.
- Glorot and Bengio, 2015, ultimately suggest the weights be drawn from:

$$U\left(-\frac{\sqrt{6}}{n_{\text{in}}+n_{\text{out}}}, \frac{\sqrt{6}}{n_{\text{in}}+n_{\text{out}}}\right)$$

Xavier initialization with ReLU



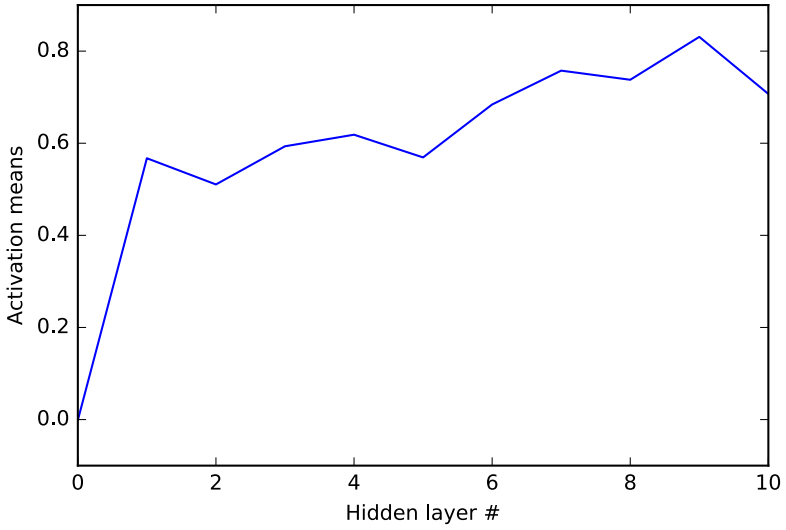


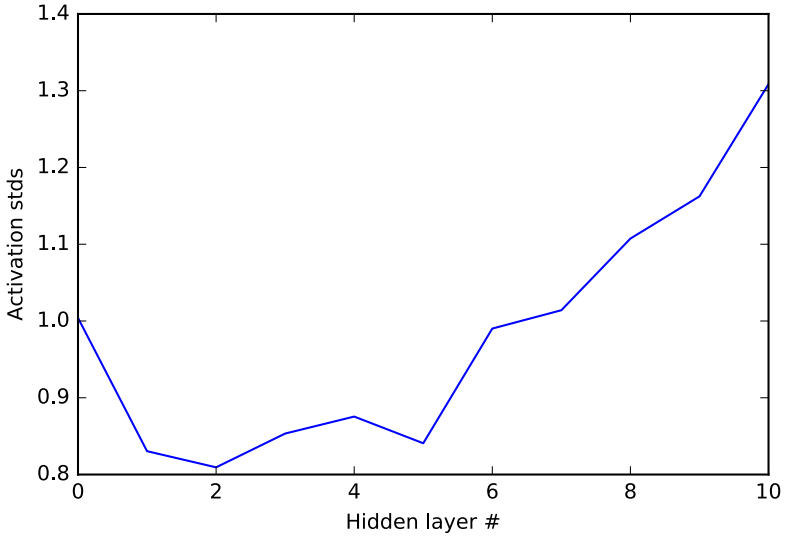


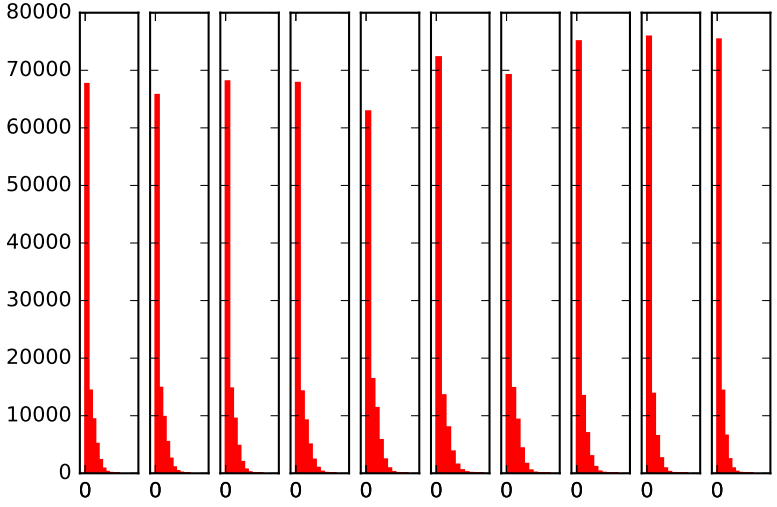
He (MSRA) initialization with ReLU

```
# Generate random data
X = np.random.randn(100,1000) # input is 100-dimensional, with 1000 data points
num_layers = 10
layer_sizes = [100] * num_layers # 10 hidden layers with 100 units each

# Forward propagation
Hs = [X]
Zs = [X]
Ws = []
for i in np.arange(num_layers):
    H = Hs[-1]

    # Initialize Xavier
    W = np.random.randn(layer_sizes[i], H.shape[0]) * np.sqrt(2) / np.sqrt(100)
    Z = np.dot(W, H)
    H = Z * (Z>0)
    Hs.append(H)
    Ws.append(W)
    Zs.append(Z)
```

He (MSRA) initialization with ReLU



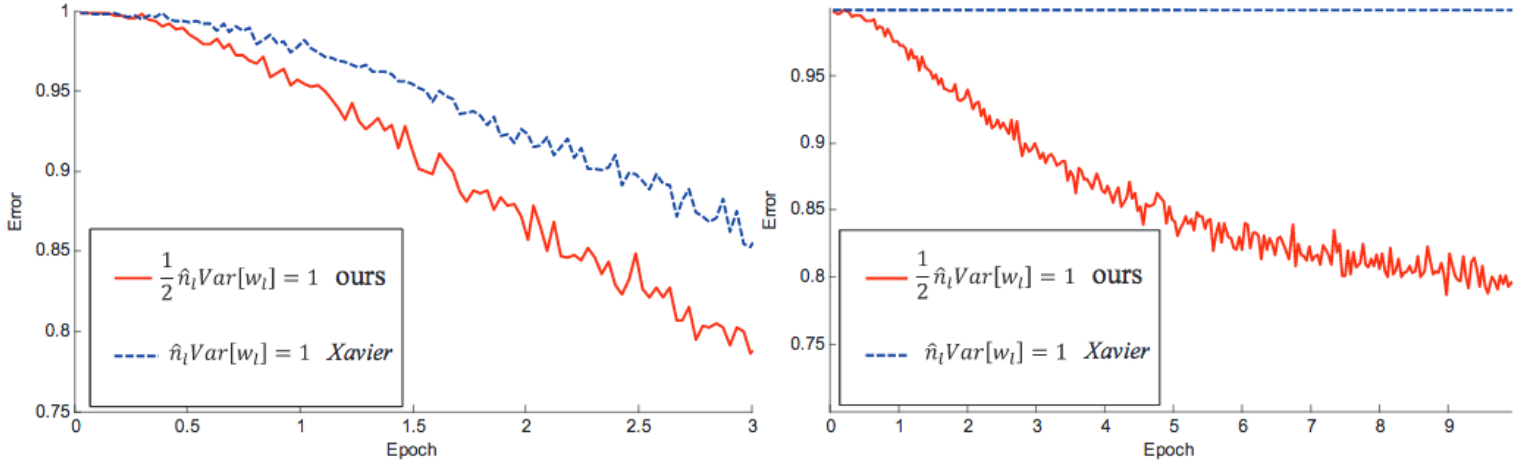




Initialization take home points

Initialization is a **very** important aspect of training neural networks and remains an active area of research.

A factor of two in the initialization can be the difference between the network learning well or not learning at all.



He et al., 2015

Sussillo and Abbott, "Random walk initialization for training very deep feedforward networks," 2014.
He et al., "Delving deep into rectifiers; surpassing human-level performance on ImageNet classification," 2015.
Mishkin and Matas, "All you need is a good init," 2015.

Initialization take home points

E.g., Mishkin and Matas, “All you need is a good init,” 2015.

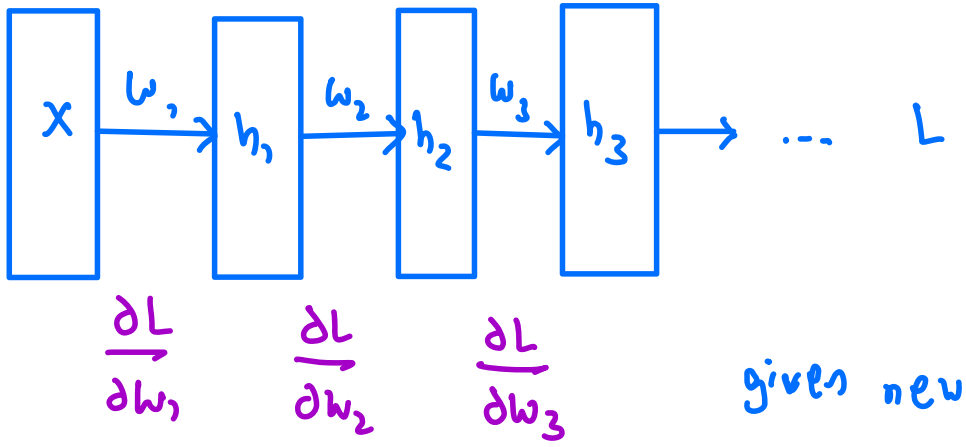
Init method	maxout	ReLU	VLReLU	tanh	Sigmoid
LSUV	**93.94**	**92.11**	92.97	89.28	n/c
OrthoNorm	93.78	91.74	92.40	89.48	n/c
OrthoNorm-MSRA scaled	–	91.93	**93.09**	–	n/c
Xavier	91.75	90.63	92.27	**89.82**	n/c
MSRA	n/c†	90.91	92.43	89.54	n/c

Avoiding saturation, high variance activations, etc.

We will motivate batchnorm through internal covariate shift, though new research suggests that batchnorm works primarily due to making the loss surface smoother.

An obstacle to standard training is that the distribution of the inputs to each layer changes as learning occurs in previous layers. As a result, the unit activations can be very variable. Another consideration is that when we do gradient descent, we're calculating how to update each paramer *assuming the other layers don't change*. But these layers may change drastically. Ultimately, these cause:

- Learning rates to be smaller (than if the distributions were not so variable).
- Networks to be more sensitive to initializations.
- Difficulties in training networks that saturate (where learning will no longer occur).



$\frac{\partial L}{\partial w_2}$ is calculated assuming w_1 is constant. As we do gradient descent together, given newer w_1, $\partial L/\partial w_2$ might have been bad choice.

Batch normalization

The idea of batch-normalization is to make the output of each layer have unit Gaussian statistics. Learning then becomes simpler because parameters in the lower layers do not change the statistics of the input to a given layer. This makes learning more straightforward.

$E(h_i) = 0$

$var(h_i) = 1$

We will apply BN before relu.

	Ioffe and Szegedy 2015 ★	RECENTLY ↘ Better
affine	affine	affine
↓	↓	↓
relu	BN	relu
↓	↓	↓
affine	relu	BN
↓	↓	↓
relu	affine	⋮
⋮	⋮	

Batch normalization (cont.)

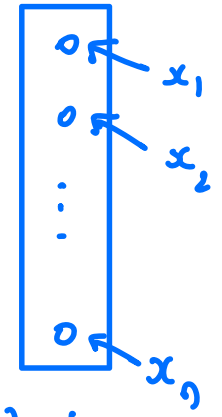
(Ioffe and Szegedy, 2015), introduced batch normalization.

- Normalize the unit activations:

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

m training examples

ε to prevent division by zero.



with

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

training example

neuron

and ε small.

$E(\hat{x}_i) = 0$

$\text{var}(\hat{x}_i) = 1$

- Scale and shift the normalized activations:

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

learnable parameters

Importantly, the normalization and scale / shift operations are included in the computational graph of the neural network, so that they participate not only in forward propagation, but also in backpropagation.

$E(y_i) = \beta_i$

$\text{var}(y_i) = \gamma_i^2$

$\gamma_i, \beta_i \rightarrow$ per neuron

Batch normalization (cont.)

A few notes on batch normalization's implementation:

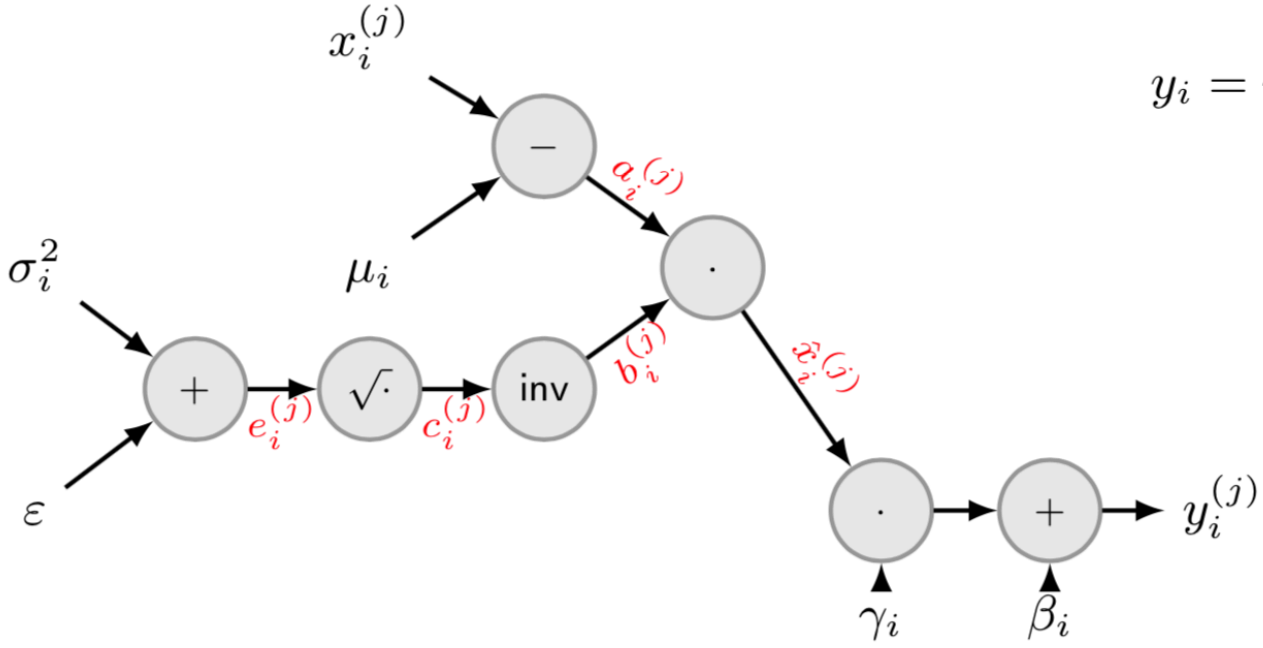
- The reason the scaling is on a per unit basis is primarily computational efficiency. It is possible to normalize the entire layer via $\Sigma^{-1/2}(\mathbf{x} - \mu)$ where μ, Σ are the mean and covariance of $\mathbf{x}$. However, this requires computation of a covariance matrix, its inverse, and the appropriate terms for backpropagation (including computation of the Jacobian of this transform).
- The scale and shift layer is inserted in case it is better that the activations not be zero mean and unit variance. As γ_i and β_i are parameters, it is possible for the network to rescale the activations. In fact, it could learn $\gamma_i = \sigma_i$ and $\beta_i = \mu_i$ to undo the normalization.

Batch normalization

Batch normalization computational graph

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

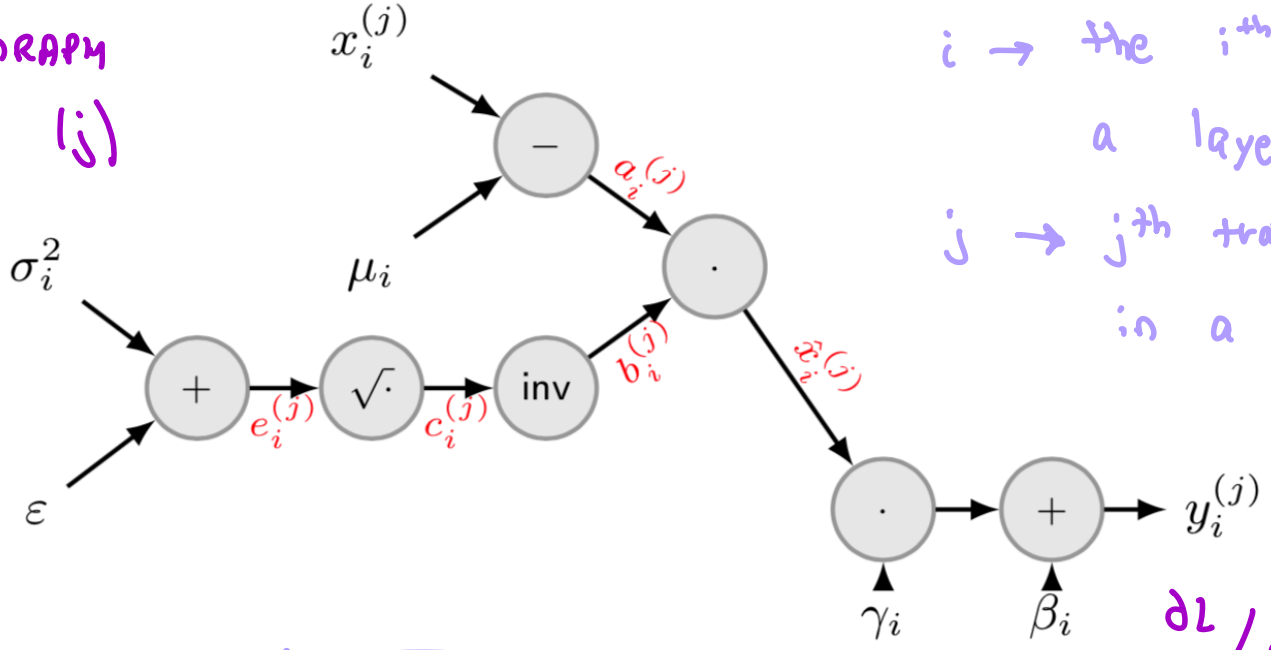


Gradients from backprop on next page.

We will drop the subscript i for convenience.

Batch normalization

COMPUTATION GRAPH FOR EXAMPLE (j)



i → the i^{th} unit / neuron in a layer

j → j^{th} training example in a mini batch.

$\partial L / \partial y^{(j)}$

LAW OF TOTAL GRADIENTS

$$\frac{\partial \ell}{\partial \beta} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial y^{(j)}} \hat{x}^{(j)}$$

$$\frac{\partial \ell}{\partial \hat{x}^{(j)}} = \frac{\partial \ell}{\partial y^{(j)}} \gamma$$

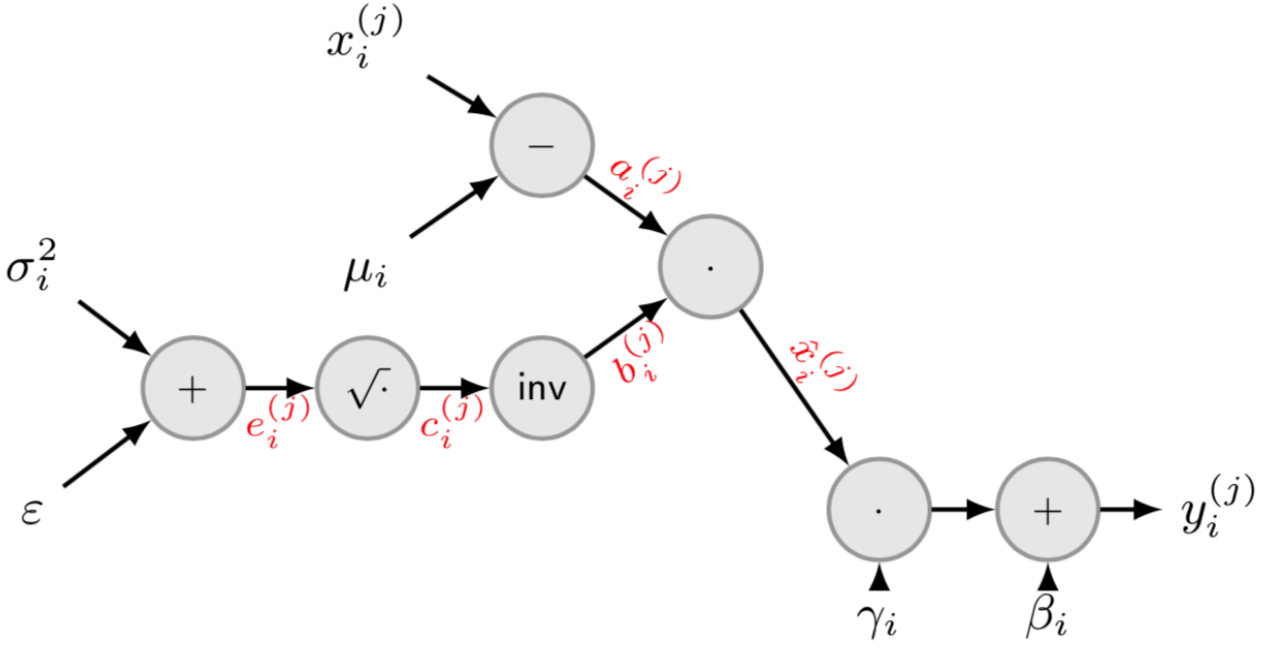
$$\frac{\partial \ell}{\partial a^{(j)}} = \underbrace{\frac{1}{\sqrt{\sigma^2 + \epsilon}}}_{b^{(j)}} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y^{(j)}}$

$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y^{(j)}} \cdot \hat{x}^{(j)}$

$\frac{\partial L}{\partial \hat{x}^{(j)}} = \frac{\partial L}{\partial y^{(j)}} \cdot \gamma$

Batch normalization



$$\frac{\partial \ell}{\partial \mu} = -\frac{1}{\sqrt{\sigma^2+\epsilon}} \sum_{j=1}^{m} \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial b^{(j)}} = (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

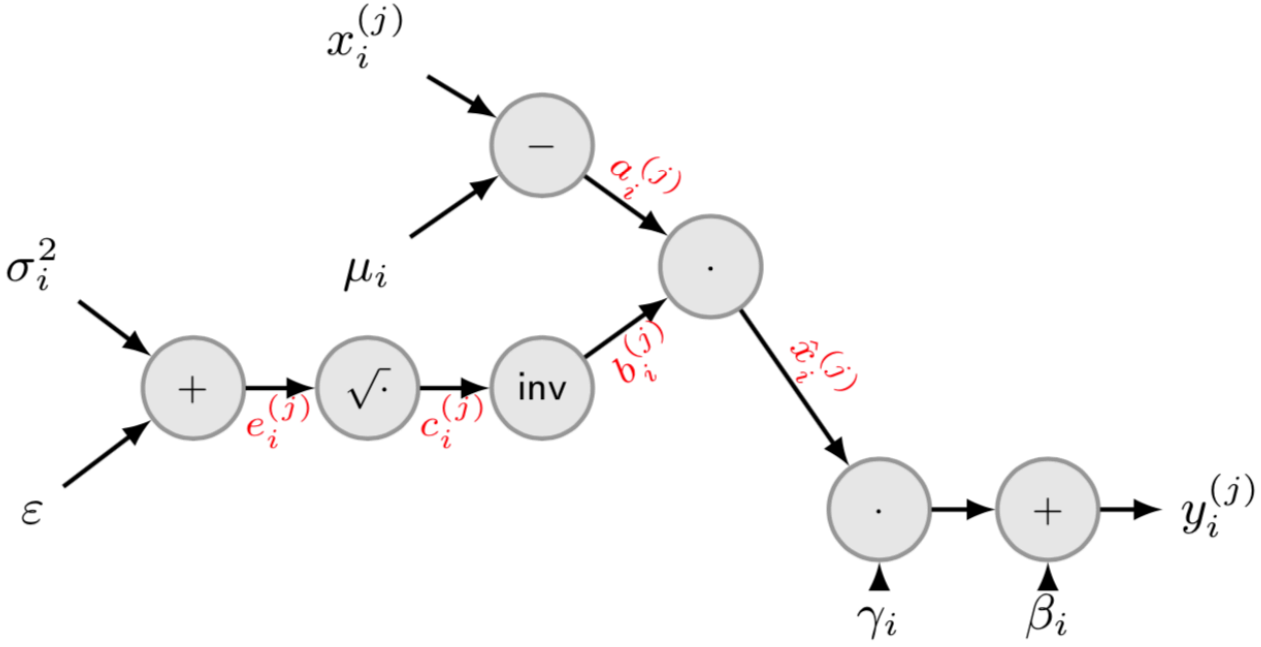
$$\frac{\partial \ell}{\partial c^{(j)}} = -\frac{1}{\sigma^2+\varepsilon}(x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}}(x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$b^{(j)} = \frac{1}{c^{(j)}}$

$\frac{\partial b^{(j)}}{\partial c^{(j)}} = \frac{-1}{(c^{(j)})^2}$

Batch normalization

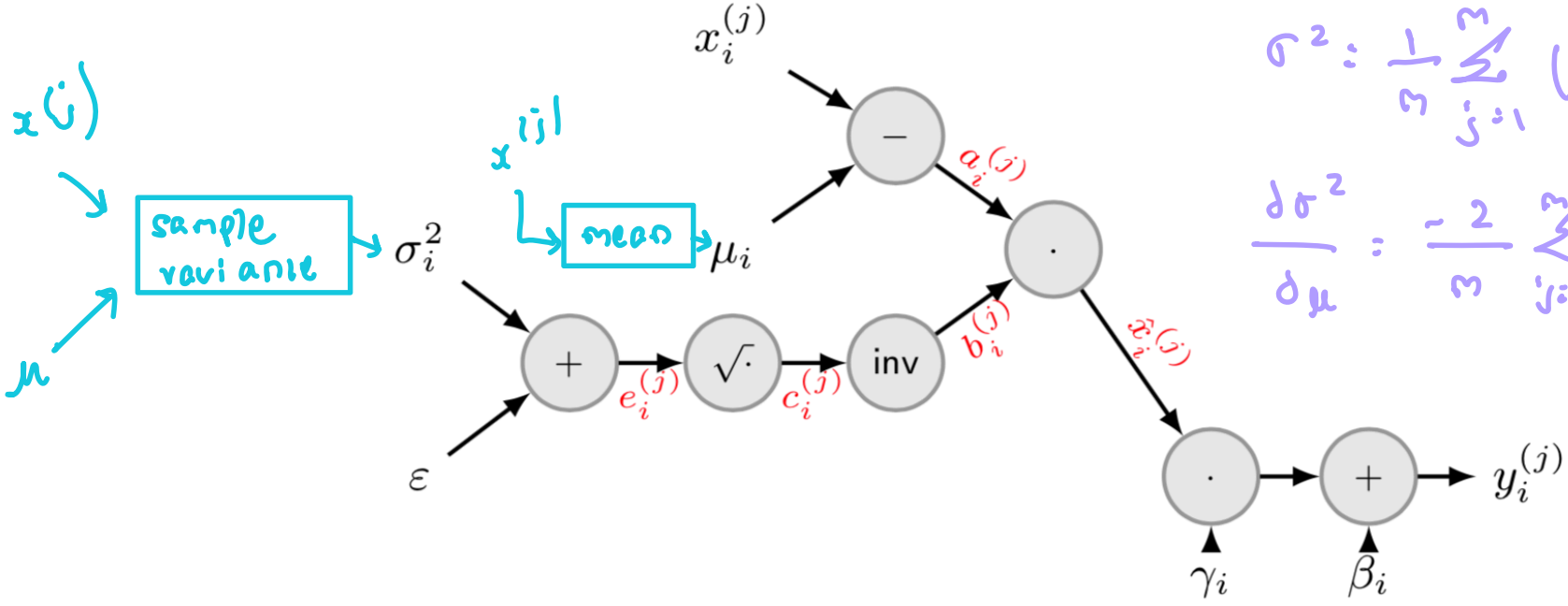


$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}} \frac{\partial \ell}{\partial \hat{x}^{(j)}} \qquad \frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial \sigma^2} = \sum_{j=1}^{m} \frac{\partial \ell}{\partial e^{(j)}}$$

$$= \sum_{j=1}^{m} -\frac{1}{2} \frac{1}{(\sigma^2+\varepsilon)^{3/2}} (x^{(j)} - \mu) \frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

Batch normalization



$$\sigma^2 = \frac{1}{m}\sum_{j=1}^{m}(x^{(j)}-\mu)^2$$

$$\frac{d\sigma^2}{d\mu} = \frac{-2}{m}\sum_{j=1}^{m}(x^{(j)}-\mu)$$

$$\frac{\partial \ell}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\epsilon}}\frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial e^{(j)}} = -\frac{1}{2}\frac{1}{(\sigma^2+\varepsilon)^{3/2}}(x^{(j)}-\mu)\frac{\partial \ell}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \ell}{\partial x^{(j)}} = \frac{\partial \ell}{\partial a^{(j)}} + \frac{\partial \sigma^2}{\partial x^{(j)}}\frac{\partial \ell}{\partial \sigma^2} + \frac{\partial \mu}{\partial x^{(j)}}\frac{\partial \ell}{\partial \mu}$$

$$= \frac{1}{\sqrt{\sigma^2+\epsilon}}\frac{\partial \ell}{\partial \hat{x}^{(j)}} + \frac{2(x^{(j)}-\mu)}{m}\frac{\partial \ell}{\partial \sigma^2} + \frac{1}{m}\frac{\partial \ell}{\partial \mu}$$

Batch normalization layer

The batch normalization layer is typically placed right before the nonlinear activation. Hence, a layer of a neural network may look like:

$$\mathbf{h}_i = f\left(\text{batch-norm}\left(\mathbf{W}_i\mathbf{h}_{i-1} + \mathbf{b}_i\right)\right)$$

Batch normalization

Empirically, batch normalization:

- Allows higher learning rates to be used.
- Reduces the strong dependence on initialization.

What is regularization?

What is regularization?

Regularizations

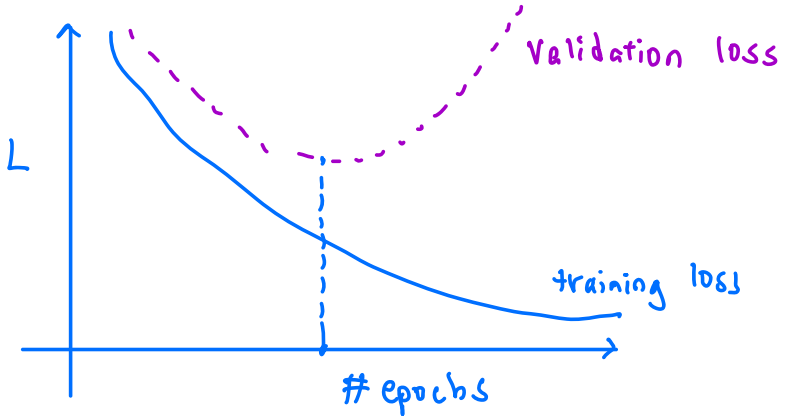
Regularizations are used to improve model generalization. Goodfellow, Bengio, and Courville define regularization in the following way (Deep Learning, p. 221):

> *[Regularization is] any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

In this manner, regularization is used to improve the *generalizability* of the model. Other intuitions:

- Regularization tends to increase the estimator bias while reducing the estimator variance.
- Regularization can be seen as a way to prevent overfitting.
- A common problem is in picking the model size and complexity. It may be appropriate to simply choose a large model that is regularized appropriately.

eg: EARLY STOPPING



What is regularization?

Types of regularization

Regularizations may take on many different types of forms. The following list is not exhaustive, but includes regularizations one may consider.

- It may be appropriate to add a soft constraint on the parameter values in the objective function.
  - To account for prior knowledge (e.g., that the parameters have a bias).
  - To prefer simpler model classes that promote generalization.
  - To make an underdetermined problem determined. (e.g., least squares with indeterminate $\mathbf{X}^T\mathbf{X}$.)
- Dataset augmentation
- Ensemble methods (i.e., essentially combining the output of several models).
- Some training algorithms (e.g., stopping training early, dropout) can be seen as a type of regularization.

A simple example of regularization

A simple example: stopping early

One straightforward (and popular) way to regularize is to constantly evaluate the training and validation loss on each training iteration, and return the model with the lowest validation error.

- Requires caching the lowest validation error model.
- Training will stop when after a pre-specified number of iterations, no model has decreased the validation error.
- The number of training steps can be thought of as another hyperparameter.
- Validation set error can be evaluated in parallel to training.
- It doesn't require changing the model or cost function.
- The following is beyond the scope of the class – but in case curious, early stopping can be seen as a form of L^2 regularization (to be discussed in the next slides). See Goodfellow et al., *Deep Learning*, p. 242-5 for an indepth discussion.

Parameter norm penalties

It is common to associate regularization with parameter norm penalties. These are not specific to neural networks. These are commonly used, e.g., even in linear regression, where specific penalty norms have their own names (e.g., Tikhonov regularization / ridge regression).

Regularization via parameter norm penalties

A common (and simple to implement) type of regularization is to modify the cost function with a *parameter norm penalty*. This penalty is typically denoted as $\Omega(\theta)$ and results in a new cost function of the form:

$$J(\theta; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\theta)$$

with $\alpha \geq 0$. A few things to note:

- α is a hyperparameter that weights the contribution of the norm penalty. When $\alpha = 0$, there is no regularization. When $\alpha \to \infty$, the cost function is irrelevant and the model will set the parameters to minimize $\Omega(\theta)$.
- The choice of α can strongly affect generalization performance.

L2 regularization

L^2 regularization

A common form of parameter norm regularization is to penalize the size of the weights (L^2 regularization). This is also commonly called "ridge regression" or "Tikhonov regularization." This promotes models with parameters that are closer to 0 (and hence, colloquially speaking, "simpler"). If $\mathbf{w}$ are the model parameters to be regularized, then L^2 regularization sets:

$$\Omega(\theta) = \frac{1}{2}\mathbf{w}^T\mathbf{w} \qquad \Omega(\theta) = \frac{1}{2}\|w\|_F^2$$

Intuitively, to prevent $\Omega(\theta)$ from getting large, L^2 regularization will cause the weights $\mathbf{w}$ to have small norm.

$$\frac{\partial \Omega(\theta)}{\partial w} = w$$

L2 regularization

L^2 regularization (cont)

More formally, when using L^2 regularization, the new cost function is:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

with corresponding gradient:

$$\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\mathbf{w} + \nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

The gradient step is:

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

This formalizes the intuition that L^2 regularization will shrink the weights, $\mathbf{w}$, before performing the usual gradient update.

no reg.

$w \leftarrow w - \varepsilon \nabla_w J$

with reg

$w \leftarrow w - \varepsilon \nabla_w \tilde{J} = w - \varepsilon(\alpha w - \nabla_w J) = (1 - \varepsilon\alpha) w - \varepsilon \nabla_w J$

weights decay

Other places you may have seen L2 regularization (NOT tested)

Other equivalent statements of L^2 regularization

While we won't discuss these at length in class, it may be worthwhile to work out these equivalences:

- L^2 regularization is equivalent to maximum a-posteriori inference, where the prior on the parameters has a unit Gaussian distribution, i.e.,

$$\mathbf{w} \sim \mathcal{N}(0, \frac{1}{\alpha}\mathbf{I})$$

- When performing L^2 regularization, the component of $\mathbf{w}$ aligned with the ith eigenvector of the Hessian is rescaled by a factor

$$\frac{\lambda_i}{\lambda_i + \alpha}$$

- In linear regression, the least squares solution $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ becomes:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

scaling the variance of each input feature. The dimensions of $\mathbf{X}^T\mathbf{X}$ that are large (i.e., high variance) aren't affected as much, while those dimensions where $\mathbf{X}^T\mathbf{X}$ is small are.

Extensions of L^2 regularization

Other related forms of regularization include:

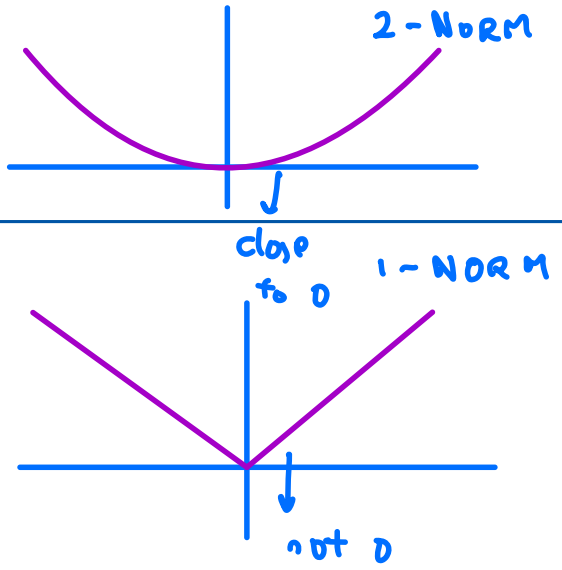
- Instead of a soft constraint that $\mathbf{w}$ be small, one may have prior knowledge that $\mathbf{w}$ is close to some value, $\mathbf{b}$. Then, the regularizer may take the form:

$$\Omega(\theta) = \|\mathbf{w} - \mathbf{b}\|_2$$

- One may have prior knowledge that two parameters, $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$, ought be close to each other. Then, the regularizer may take the form:

$$\Omega(\theta) = \left\|\mathbf{w}^{(1)} - \mathbf{w}^{(2)}\right\|_2$$

L1 regularization



L^1 regularization

L^1 regularization defines the parameter norm penalty as:

$$\begin{aligned}\Omega(\theta) &= \|\mathbf{w}\|_1 \\ &= \sum_i |w_i|\end{aligned}$$

Intuitively, this penalty also causes the weights to be small. However, because the subgradient of $\|\mathbf{w}\|_1$ is $\text{sign}(\mathbf{w})$, the gradient is the same regardless of the size of $\mathbf{w}$. (Contrast this to L^2 regularization, where the size of $\mathbf{w}$ matters.) Empirically, this typically results in sparse solutions where $w_i = 0$ for several i.

- This may be used for *feature selection*, where features corresponding to zero weights may be discarded.
- L^1 regularization is equivalent to maximum a-posteriori inference where the prior on the parameters has an isotropic Laplace distribution, i.e.,

$$w_i \sim \text{Laplace}\left(0, \frac{1}{\alpha}\right)$$

L1 regularization on the units

Sparse representations

Instead of having sparse parameters (i.e., elements of $\mathbf{w}$ being sparse), it may be appropriate to have sparse *representations*. Imagine a hidden layer of activity, $\mathbf{h}^{(i)}$. To achieve a sparse representation, one may set:

$$\Omega(\mathbf{h}^{(i)}) = \left\| \mathbf{h}^{(i)} \right\|_1$$

Dataset augmentation

Original image:

Dataset augmentation

Original image:

Flipped image:

Dataset augmentation

Original image:

Flipped image:

Cropped image:

Dataset augmentation

Original image:

Flipped image:

Cropped image:

Adjust brightness

Dataset augmentation

Original image:

Flipped image:

Cropped image:

Adjust brightness

Lens correction

Dataset augmentation

Original image:

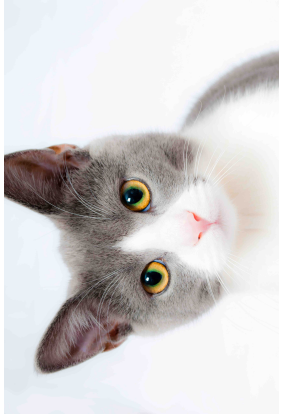
Flipped image:

Cropped image:

Adjust brightness

Lens correction

Rotate



Dataset augmentation

There are various heuristics to keeping the input size the same.

You can be creative for dataset augmentation.

GoogleNet

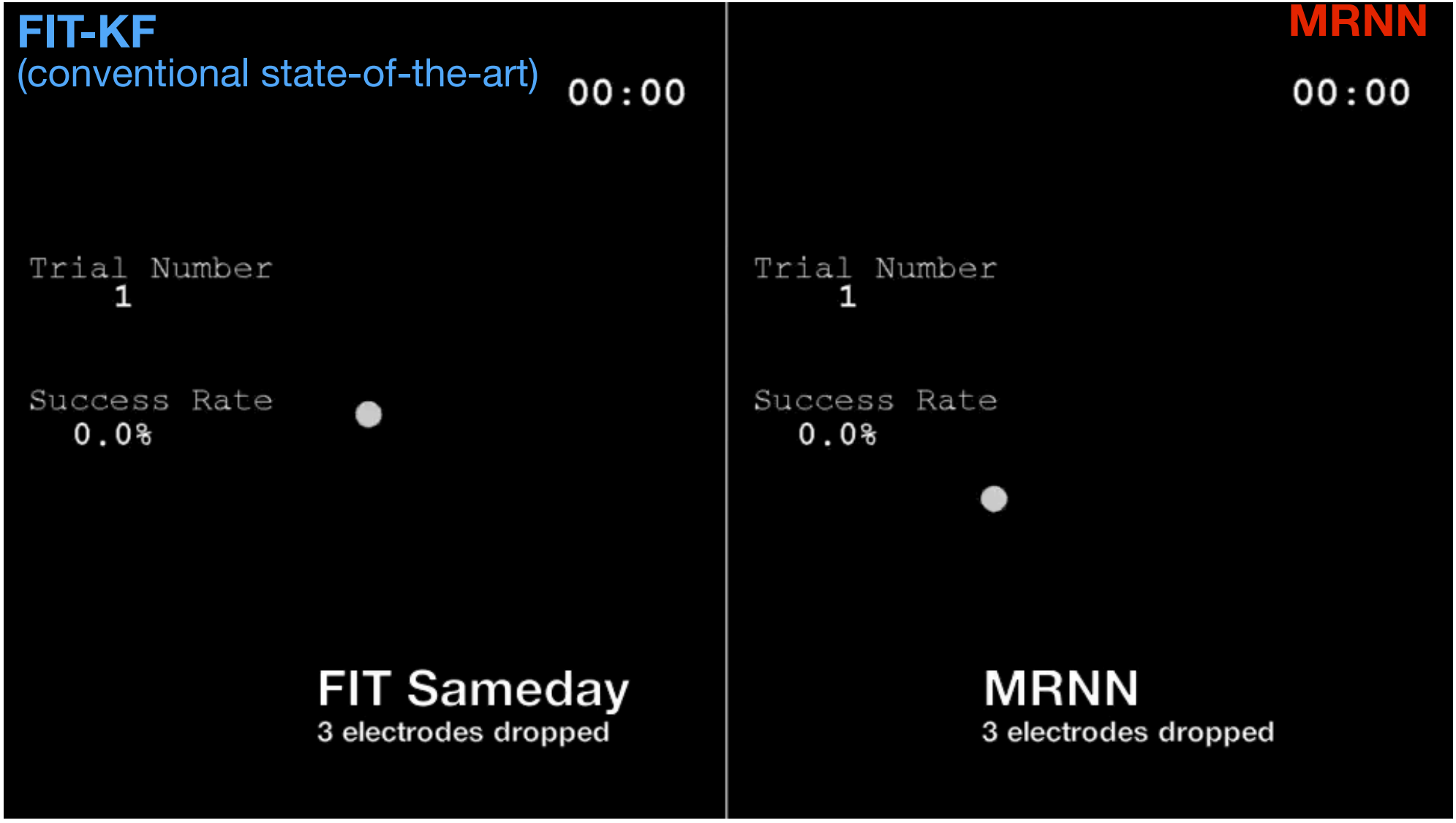
2. During testing, we adopted a more aggressive cropping approach than that of Krizhevsky et al. [9]. Specifically, we resized the image to 4 scales where the shorter dimension (height or width) is 256, 288, 320 and 352 respectively, take the left, center and right square of these resized images (in the case of portrait images, we take the top, center and bottom squares). For each square, we then take the 4 corners and the center 224×224 crop as well as the square resized to 224×224, and their mirrored versions. This leads to 4×3×6×2 = 144 crops per image. A similar ap-

Number of models	Number of Crops	Cost	Top-5 error	compared to base
1	1	1	10.07%	base
1	10	10	9.15%	-0.92%
1	144	144	7.89%	-2.18%
7	1	7	8.09%	-1.98%
7	10	70	7.62%	-2.45%
7	144	1008	6.67%	-3.45%

Szegedy et al., CVPR 2015

Dataset augmentation

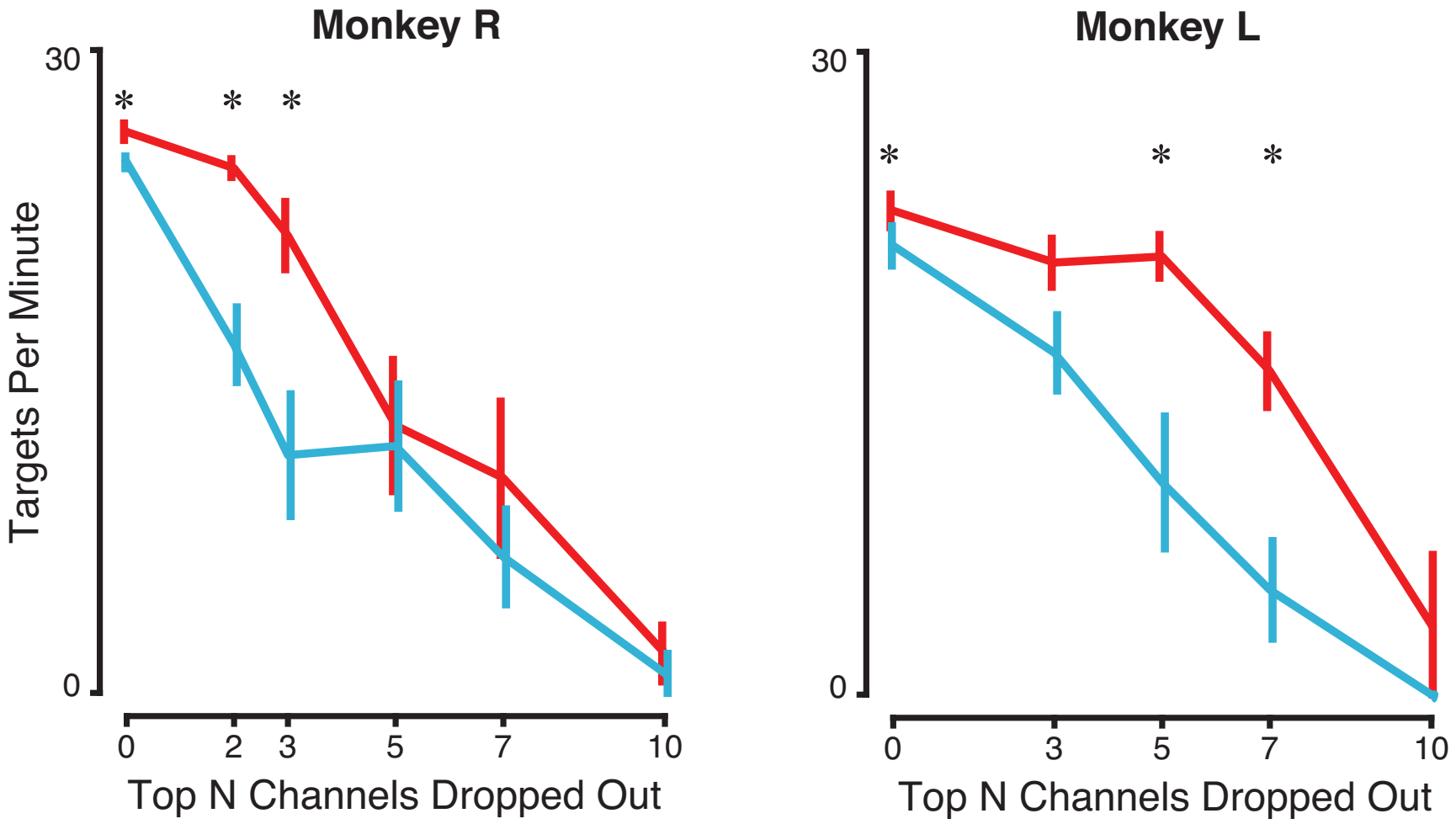
Example from brain-machine interfaces:



(back-to-back comparisons during the same experiment)

Dataset augmentation

Example from brain-machine interfaces:



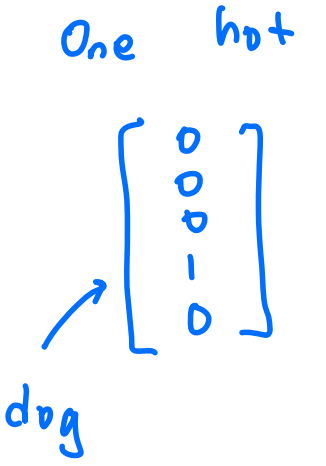
Dataset augmentation

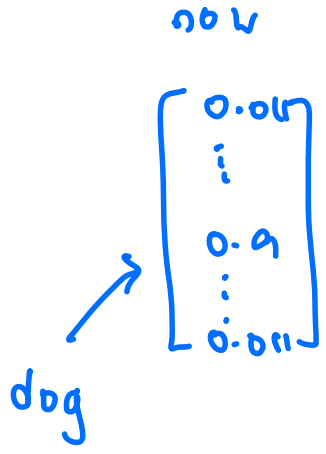
Other types of dataset augmentation:

- Inject noise into the network
- Label smoothing → labels are not one hot.

Network	Top-1 Error	Top-5 Error	Cost Bn Ops
GoogLeNet [20]	29%	9.2%	1.5
BN-GoogLeNet	26.8%	-	**1.5**
BN-Inception [7]	25.2%	7.8	2.0
Inception-v2	23.4%	-	3.8
Inception-v2 RMSProp	23.1%	6.3	3.8
Inception-v2 Label Smoothing	22.8%	6.1	3.8
Inception-v2 Factorized 7 × 7	21.6%	5.8	4.8
Inception-v2 BN-auxiliary	**21.2%**	**5.6%**	4.8

Szegedy et al., arXiv 2015

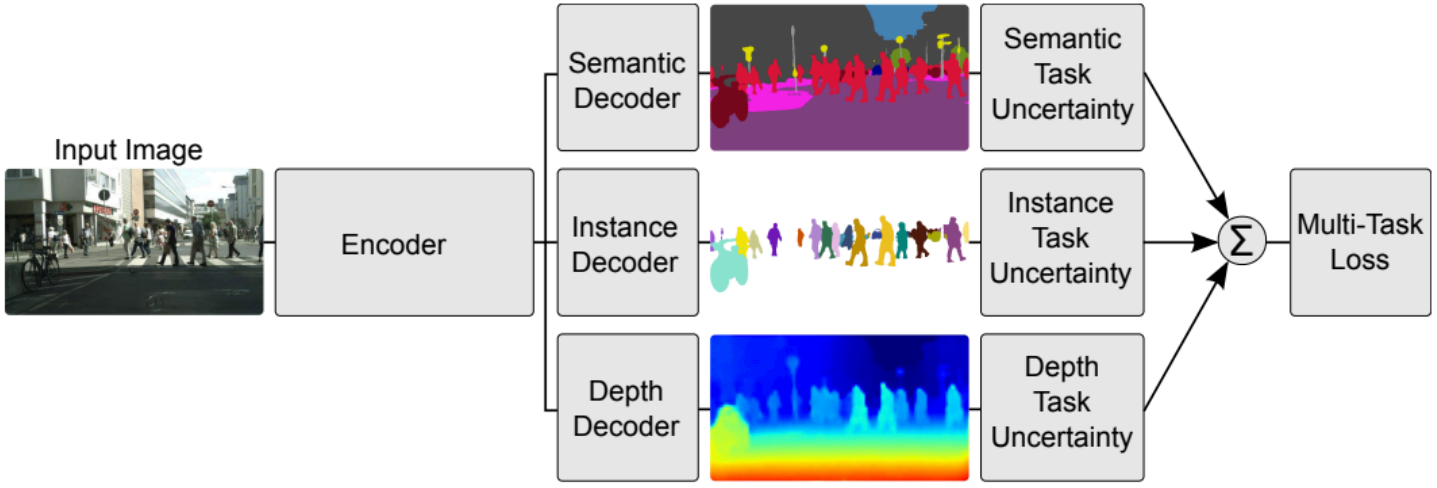




Multitask learning

Another way to improve generalization is by having the model be trianed to perform multiple tasks. This represents the prior belief that multiple tasks share common factors to explain variations in the data.

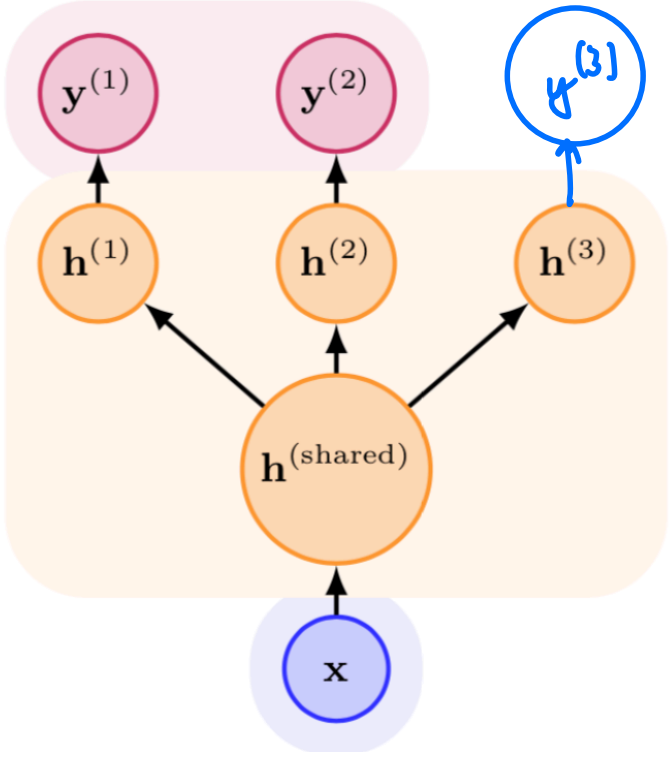
Multitask learning



Kendall et al., arXiv 2017

Multitask learning

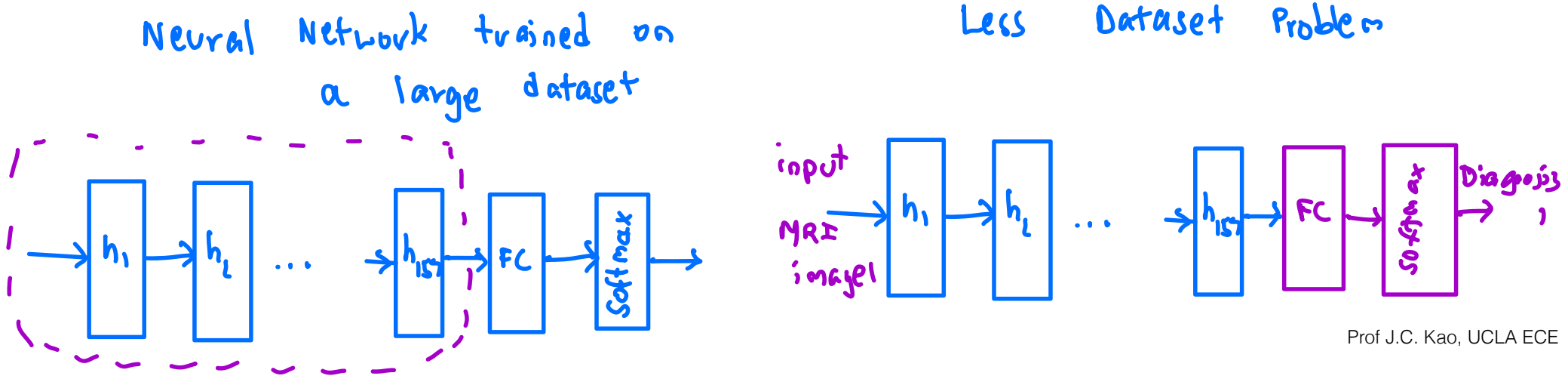
- The entire model need not be shared across different tasks.
- Here, $\mathbf{h}^{\text{shared}}$ captures common features that are then used by task-specific layers to predict $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$.
- $\mathbf{h}^{(3)}$ could represent a feature for unsupervised learning.



Transfer learning

We'll discuss this more in the convolutional neural networks lecture, but a related idea is to take neural networks trained in one context and use them in another with little additional training.

- The idea is that if the tasks are similar enough, then the features at later layers of the network ought to be good features for this new task.
- If little training data is available to you, but the tasks are similar, all you may need to do is train a new linear layer at the output of the pre-trained network.
- If more data is available, it may still be a good idea to use transfer learning, and tune more of the layers.



Ensemble methods

One way to get a boost in performance for very little cognitive work is to use ensemble methods.

The approach is:

1. Train multiple different models
2. Average their results together at test time.

This almost always increases performance by substantial amounts (e.g., a few percentage improvement in testing).

Ensemble methods

- The basic intuition between ensemble methods is that if models are independent, they will usually not all make the same errors on the test set.
- With k independent models, the average model error will decrease by a factor $\frac{1}{k}$. Denoting ϵ_i to be the error of model i on an example, and assuming $\mathbb{E}\epsilon_i = 0$ as well as that the statistics of this error is the same across all models,

$$\begin{aligned}\mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] &= \frac{1}{k^2}\sum_{i=1}^{k}\mathbb{E}\epsilon_i^2 \\ &= \frac{1}{k}\mathbb{E}\epsilon_i^2\end{aligned}$$

- If the models are not independent, it can be shown that:

$$\frac{1}{k}\mathbb{E}\epsilon_i^2 + \frac{k-1}{k}\mathbb{E}[\epsilon_i\epsilon_j]$$

worst case: $\epsilon_i = \epsilon_j$

which is equal to $\mathbb{E}\epsilon_i^2$ only when the models are perfectly correlated.

Ensemble methods

K MODELS

One implementation of ensemble methods is via bagging.

N examples → 1 : Draw N samples w/ replacement
→ 2 :
⋮
→ k :

Bagging

Bagging stands for bootstrap aggregating. It is an ensemble method for regularization. The procedure is as follows:

- Construct k datasets using the bootstrap (i.e., set a data size, N, and draw with replacement from the original dataset to get N samples; do this k times).
- Train k different models using these k datasets.

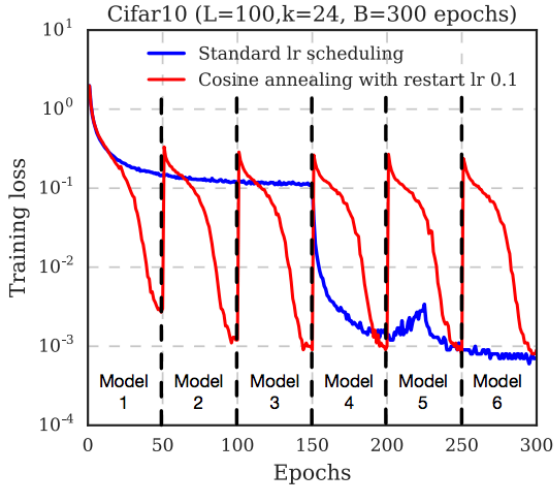
A few notes:

- In practice, neural networks reach a wide variety of solutions given different initializations, hyperparameters, etc., and so in practice even if they are trained from the same dataset, they tend to produce partially independent errors.
- While model averaging is powerful, it is expensive for neural networks, since the time to train models can be very large.

Ensemble methods

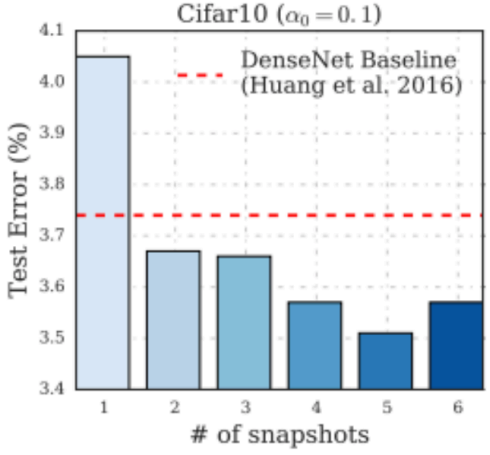
Ways to get around computational expense?

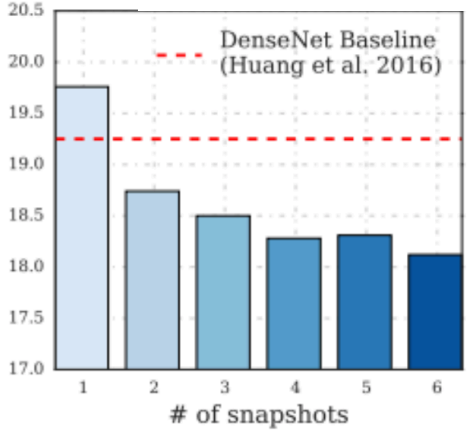
You could take snapshots of the model at different local minima and average them together. See Huang, Li et al., ICLR 2017.

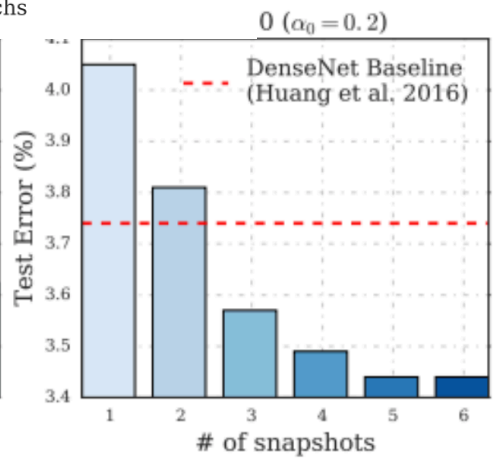


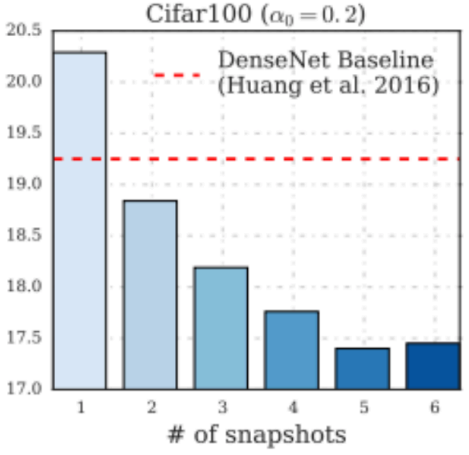
$\varepsilon \sim$ function cos

↳ learning rate









Dropout

Dropout is a computationally inexpensive yet effective method for generalization. It can be viewed as approximating the bagging procedure for exponentially many models, while only optimizing a single set of parameters. The following steps describe the dropout regularizer:

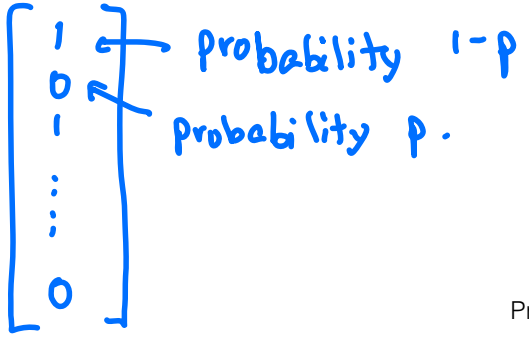
- On a given training iteration, sample a binary mask (i.e., each element is 0 or 1) for all input and hidden units in the network.
  - The Bernoulli parameter, p, is a hyperparameter.
  - Typical values are that $p = 0.8$ for input units and $p = 0.5$ for hidden units.
- Apply (i.e., multiply) the mask to all units. Then perform the forward pass and backwards pass, and parameter update.
- In *prediction*, multiply each hidden unit by the parameter of its Bernoulli mask, p.

80 1's
20 0's

0.2

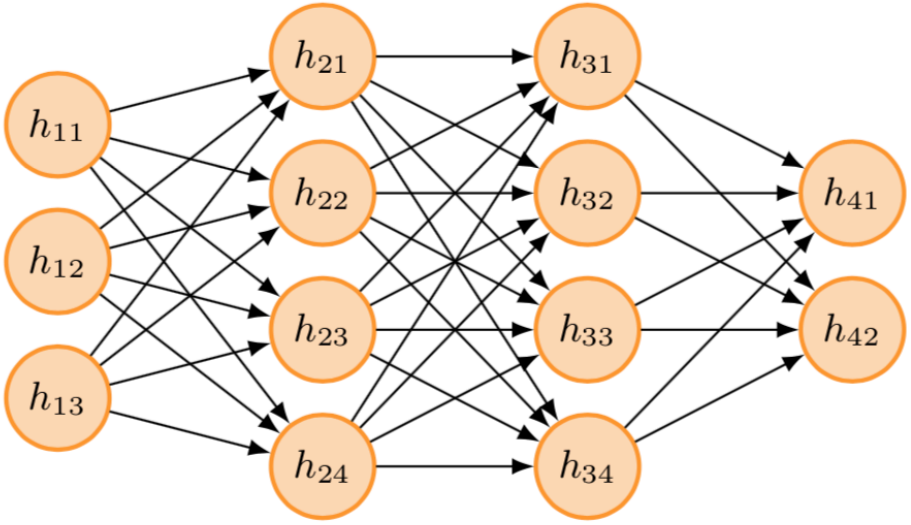
100 neurons/units

100 Bernoulli Random Units with 1-p



Dropout

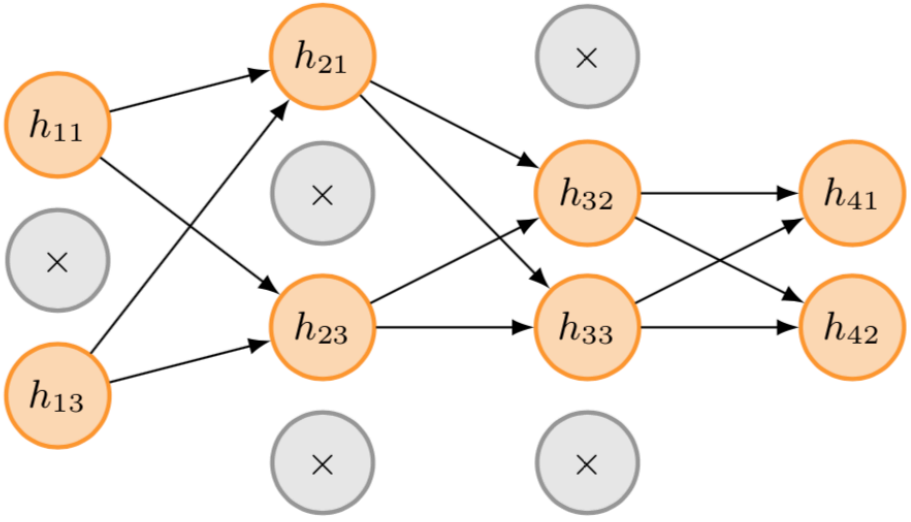
Hidden layers of network to be trained



$\mathbb{R}^{13}$

mask of 13 $\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \end{bmatrix}$

Application of random mask on iteration i



2^N possible configurations where N is the number of neurons (here: 13)

Next step → separate mask.

Dropout

Dropout in code.

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def forward(X):
    H1 = relu(np.dot(W1, X) + b1) # First hidden layer activations
    M1 = np.random.rand(*H1.shape) < p # Sample random mask
    H1 *= M1 # Dropout on first hidden layer

    H2 = relu(np.dot(W2, H1) + b2) # Second hidden layer activations
    M2 = np.random.rand(*H2.shape) < p # Sample random mask
    H2 *= M2 # Dropout on first hidden layer

    Z = np.dot(W3, H2) + b3
```

0.5 for inputs

0.2 for hidden layers

Dropout

How is this a good idea?

1) Dropout approximates bagging, since each mask is like a different model. For a model with N hidden units, there are 2^N different model configurations.

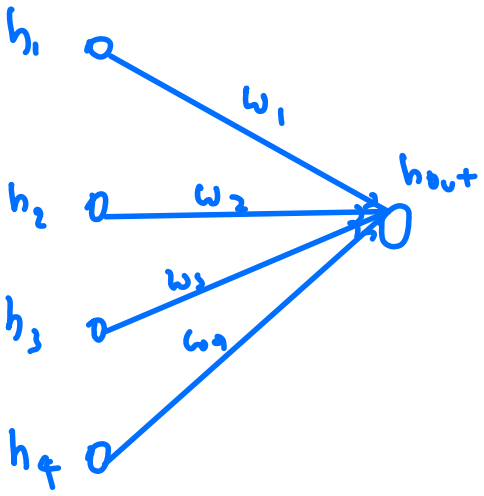
   Each of these configurations must be good at predicting the output.

2) You can think of of dropout as regularizing each hidden unit to work well in many different contexts.

3) Dropout may cause units to encode redundant features (e.g., to detect a cat, there are many things we look for, e.g., it's furry, it has pointy ears, it has a tail, a long body, etc.).

Dropout

How about during test time? What configuration do you use?



Training minibatch 1:

$$M = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad h_{out} = \text{relu}(w_1 h_1 + w_3 h_3)$$

$p = 0.5$

Training minibatch 2:

$$M = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \qquad h_{out} = \text{relu}(w_2 h_2 + w_4 h_4)$$

TEST: $h_{out} = \text{relu}(w_1 h_1 + w_2 h_2 + w_3 h_3 + w_4 h_4)$ Twice the units as before

↳ OVERACTIVATES

Over many iterations, the contribution of $w_i h_i$ to h_{out} is actually $(1-p) w_i h_i$

$h_{out} = relu((w_1h_1 + w_2h_2 + w_3h_3 + w_4h_4) \cdot (1-p))$

Dropout

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def test(X):
    H1 = relu(np.dot(W1, X) + b1) * p (1-p)
    H2 = relu(np.dot(W1, H1) + b1) * p (1-p)
    Z = np.dot(W3, H2) + b3
```

Note: an additional pro of dropout is that in testing time, there is no additional complexity. With *m* ensemble models, our test time evaluation would scale *O(m)*.

Inverted dropout

A common way to implement dropout is *inverted dropout* where the scaling by $1/p$ is done in training. This causes the output to have the same expected value as if dropout was never been performed.

Thus, testing looks the same irrespective of if we use dropout or not. See code below:

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def train_forward(X):
    H1 = relu(np.dot(W1, X) + b1) # First hidden layer activations
    M1 = (np.random.rand(*H1.shape) < p) / p # Sample random mask AND normalization by p
    H1 *= M1 # Dropout on first hidden layer

    H2 = relu(np.dot(W2, H1) + b2) # Second hidden layer activations
    M2 = (np.random.rand(*H2.shape) < p) / p # Sample random mask AND normalization by p
    H2 *= M2 # Dropout on first hidden layer

    Z = np.dot(W3, H2) + b3

def test(X):
    H1 = relu(np.dot(W1, X) + b1)
    H2 = relu(np.dot(W1, H1) + b1)
    Z = np.dot(W3, H2) + b3
```

all
(1-p)

Lecture summary

Here, we've covered tricks that we can do in initialization, regularization, and data augmentation to improve the performance of neural networks.

But what about the optimizer, stochastic gradient descent? Can we improve this for deep learning?

That's the topic of our next lecture.

Optimization for neural networks

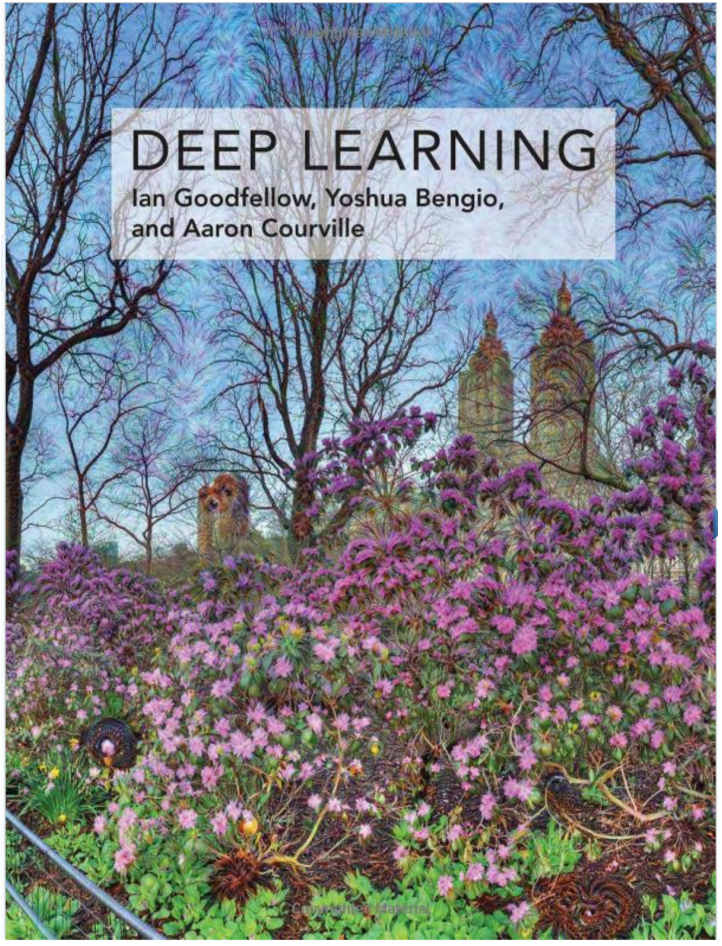
In this lecture, we'll talk about specific techniques in optimization that aid in training neural networks.

- Stochastic gradient descent
- Momentum and Nesterov momentum
- Adaptive gradients
- RMSProp
- Adaptive moments
- Overview of second order methods
- Challenges of gradient descent

Reading

Reading:

Deep Learning, Chapter 8 (intro), 8.1, 8.2, 8.3, 8.4, 8.5, 8.6 (skim)



Where we are now

At this point, we know:

- Neural network architectures.
- Hyperparameters and cost functions to use for neural networks.
- How to calculate gradients of the loss w.r.t. all parameters in the neural network.
- How to initialize the weights and regularize the network in ways to improve the training of the network.

We do know how to optimize these networks with stochastic gradient descent. But can it be improved?

In this lecture, we talk about how to make optimization more efficient and effective.

Gradient descent

A refresher on gradient descent.

- Cost function: $J(\theta)$
- Parameters: θ

Then, the gradient descent step is:

$$\theta \leftarrow \theta - \epsilon \nabla_\theta J(\theta)$$

Stochastic gradient descent

Calculating the gradient exactly is expensive, because it requires evaluating the model on all m examples in the dataset. This leads to an important distinction.

- Batch algorithm: uses all m examples in the training set to calculate the gradient.
- Minibatch algorithm: approximates the gradient by calculating it using k training examples, where $m > k > 1$.
- Stochastic algorithm: approximates the gradient by calculating it over one example.

It is typical in deep learning to use minibatch gradient descent. Note that some may also use minibatch and stochastic gradient descent interchangeably.

A note: small batch sizes can be seen to have a regularization effect, perhaps because they introduce to noise to the training process.

Stochastic gradient descent

Stochastic gradient descent proceeds as follows.

Set a learning rate ε and an initial parameter setting θ. Set a minibatch size of m examples. Until the stopping criterion is met:

- Sample m examples from the training set, $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\}$ and their corresponding outputs $\{\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(m)}\}$.
- Compute the gradient estimate:

$$\mathbf{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta J(\theta)$$

- Update parameters:

$$\theta \leftarrow \theta - \varepsilon \mathbf{g}$$

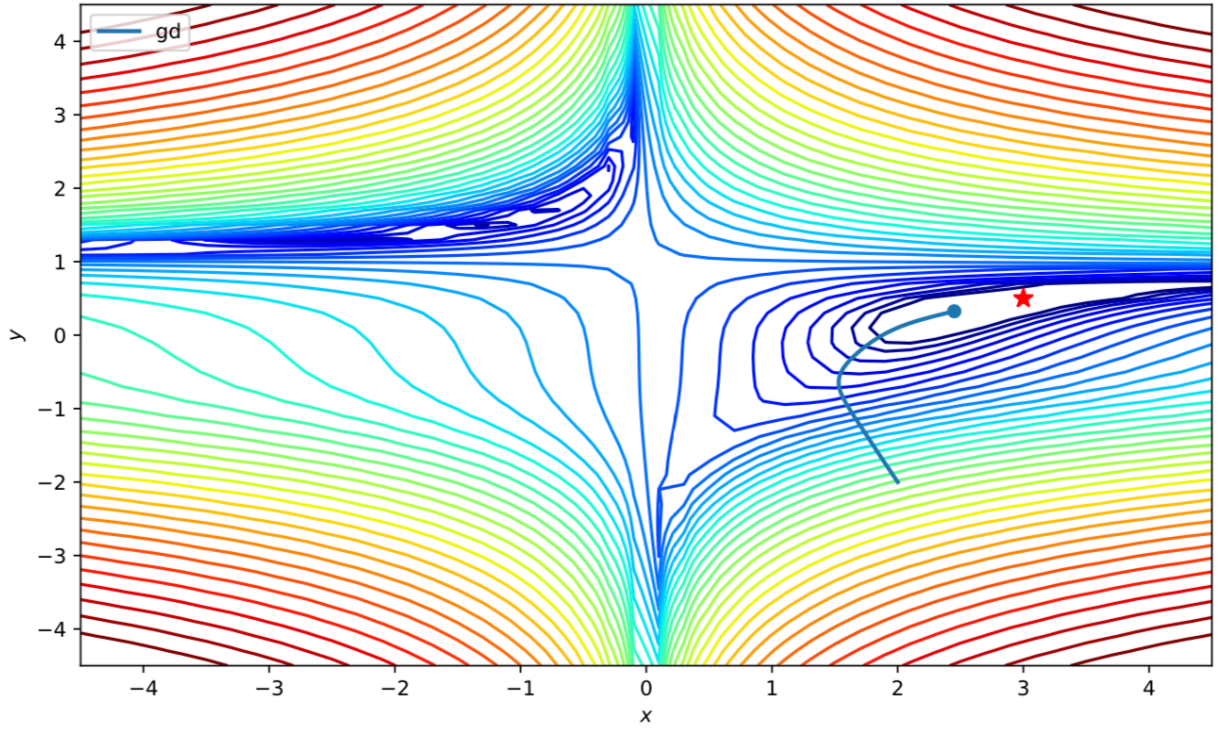
Stochastic gradient descent

Softmax/neural network loss

```
while last_diff > tol:
    cost, g = func(x)
    x -= eps*g
    last_diff = np.linalg.norm(x - path[-1])
```

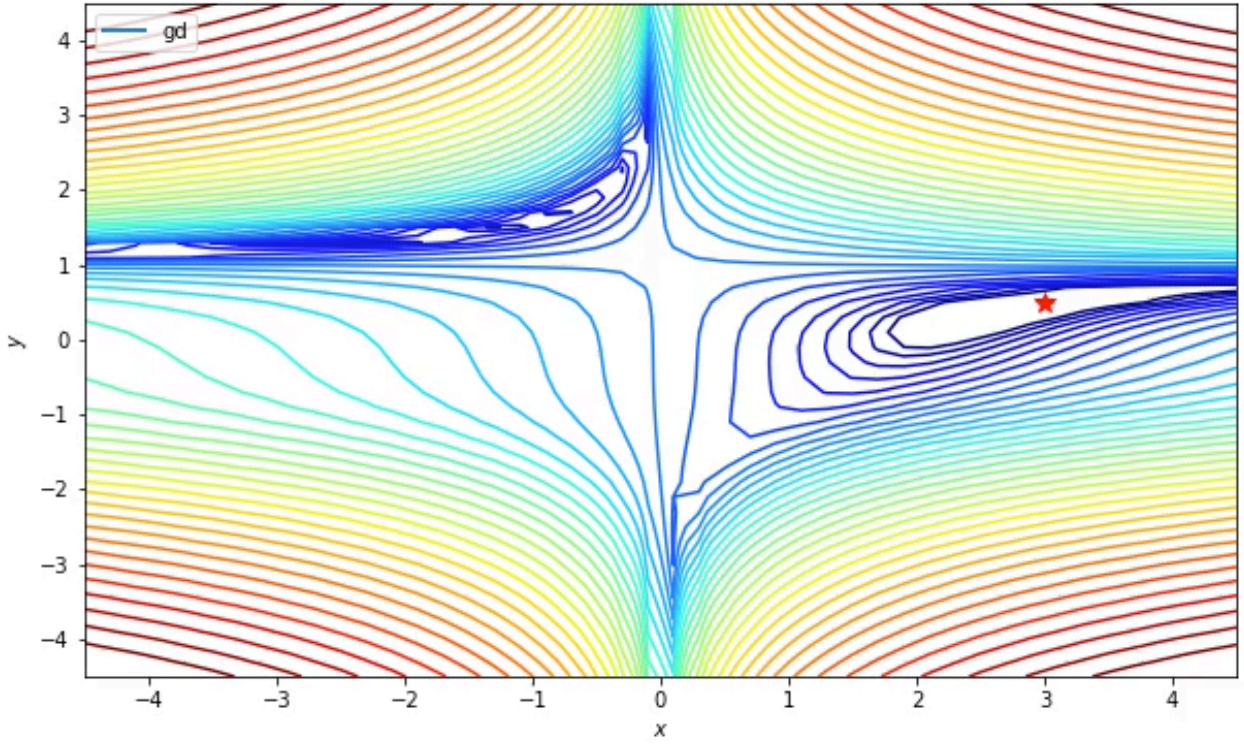
Stochastic gradient descent (opt 1)

The following shows gradient descent applied to Beale's function, for two initializations at $(2, -2)$ and $(-3, 3)$. The two initializations are shown because later on we'll contrast to other techniques. The iteration count is capped at $10,000$ iterations, so gradient descent does not get to the minimum.

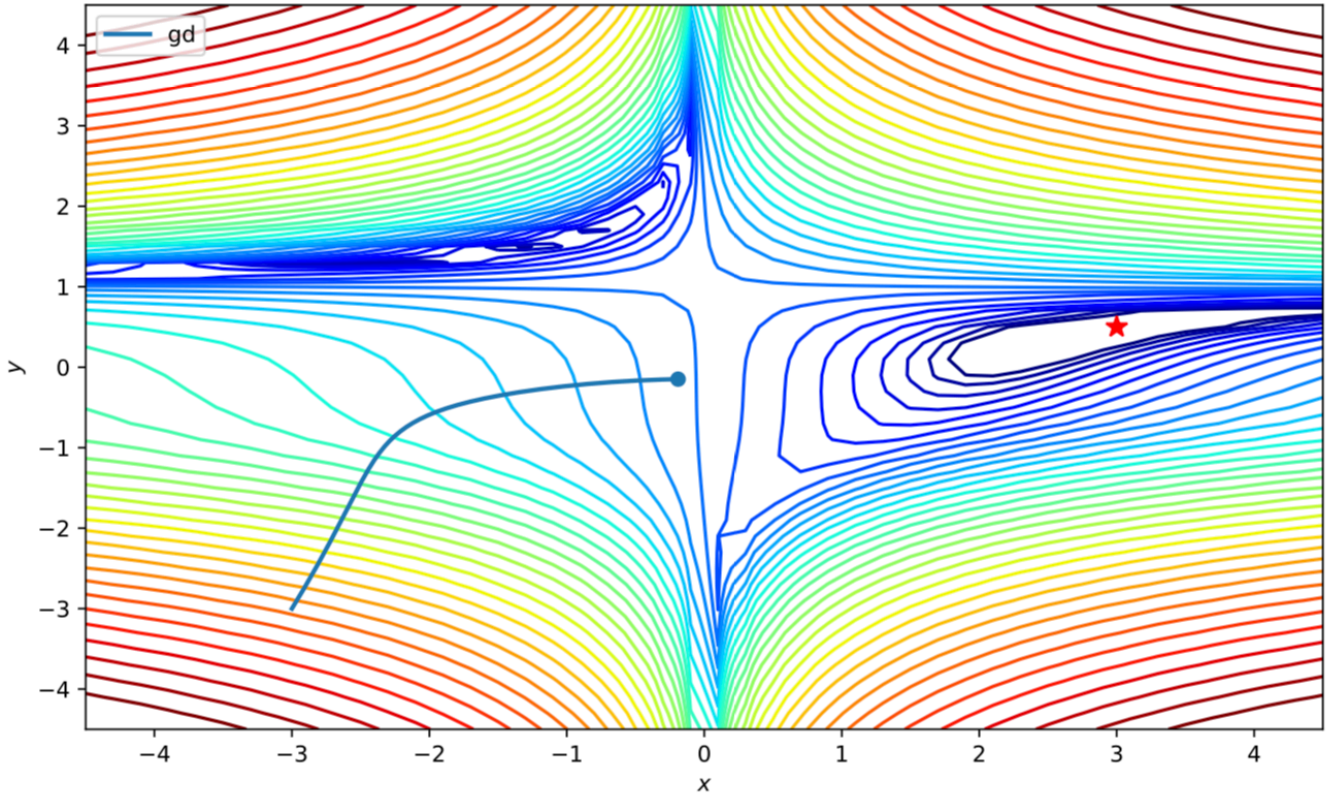


Video: http://seas.ucla.edu/~kao/opt_anim/1gd.mp4
Animation help thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/

Stochastic gradient descent



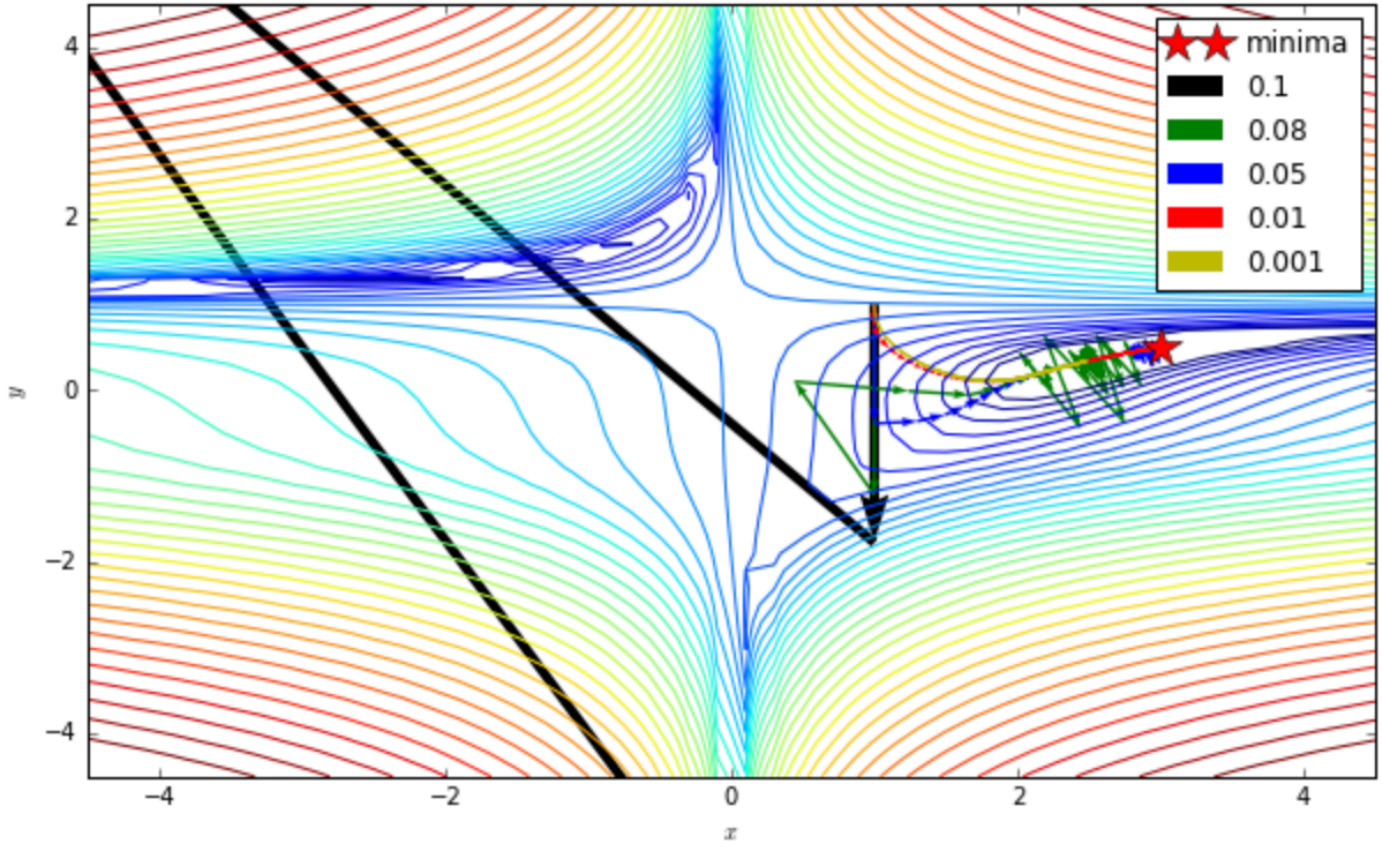
Stochastic gradient descent (opt 2)



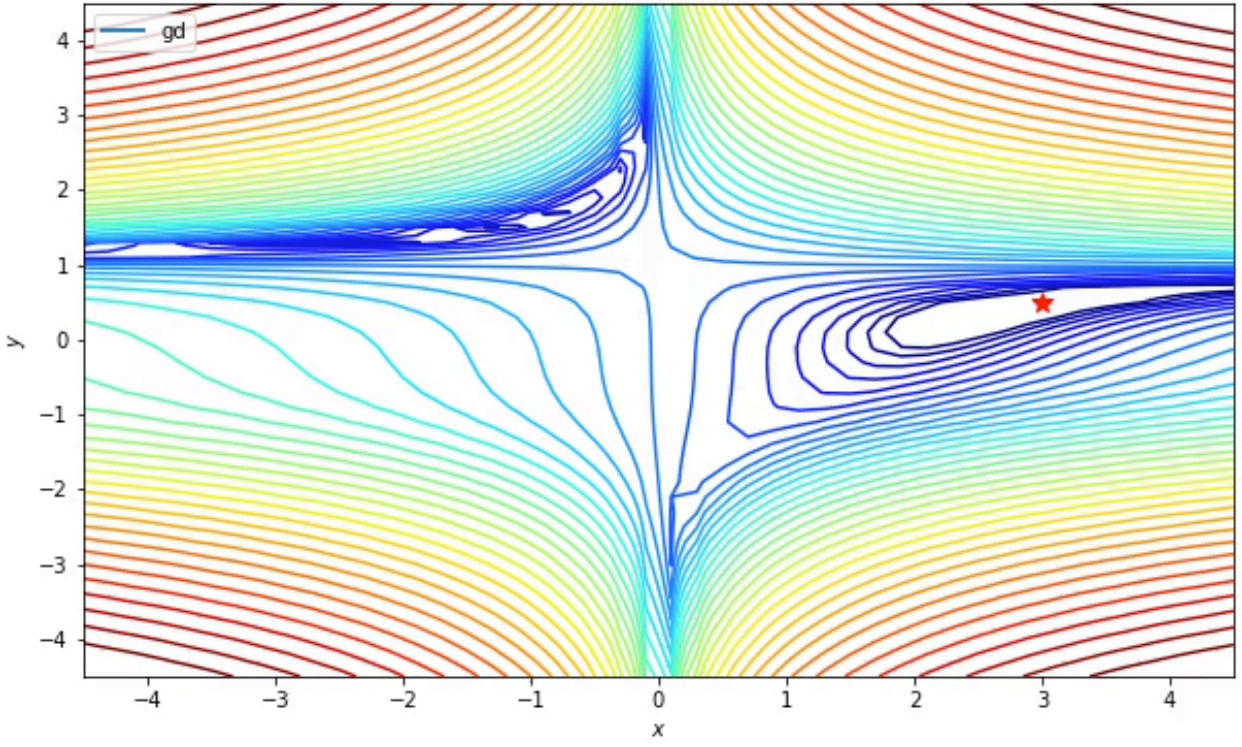
Video: `http://seas.ucla.edu/~kao/opt_anim/2gd.mp4`

Finding the optimal weights through gradient descent

Varying the learning rate:



Stochastic gradient descent



Momentum

In momentum, we maintain the running mean of the gradients, which then updates the parameters.

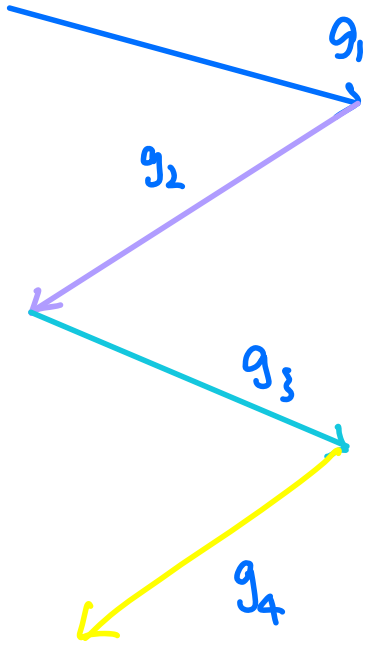
Initialize $\mathbf{v} = 0$. Set $\alpha \in [0, 1]$. Typical values are $\alpha = 0.9$ or 0.99. Then, until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Update:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \mathbf{g}$$

- Gradient step:

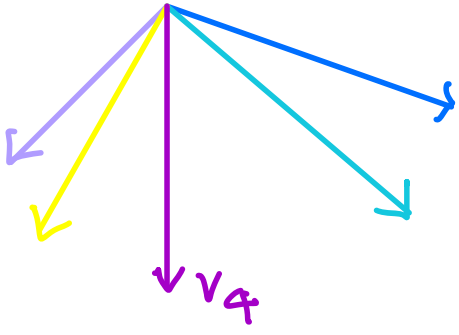
$$\theta \leftarrow \theta + \mathbf{v}$$



$v_0 = 0$

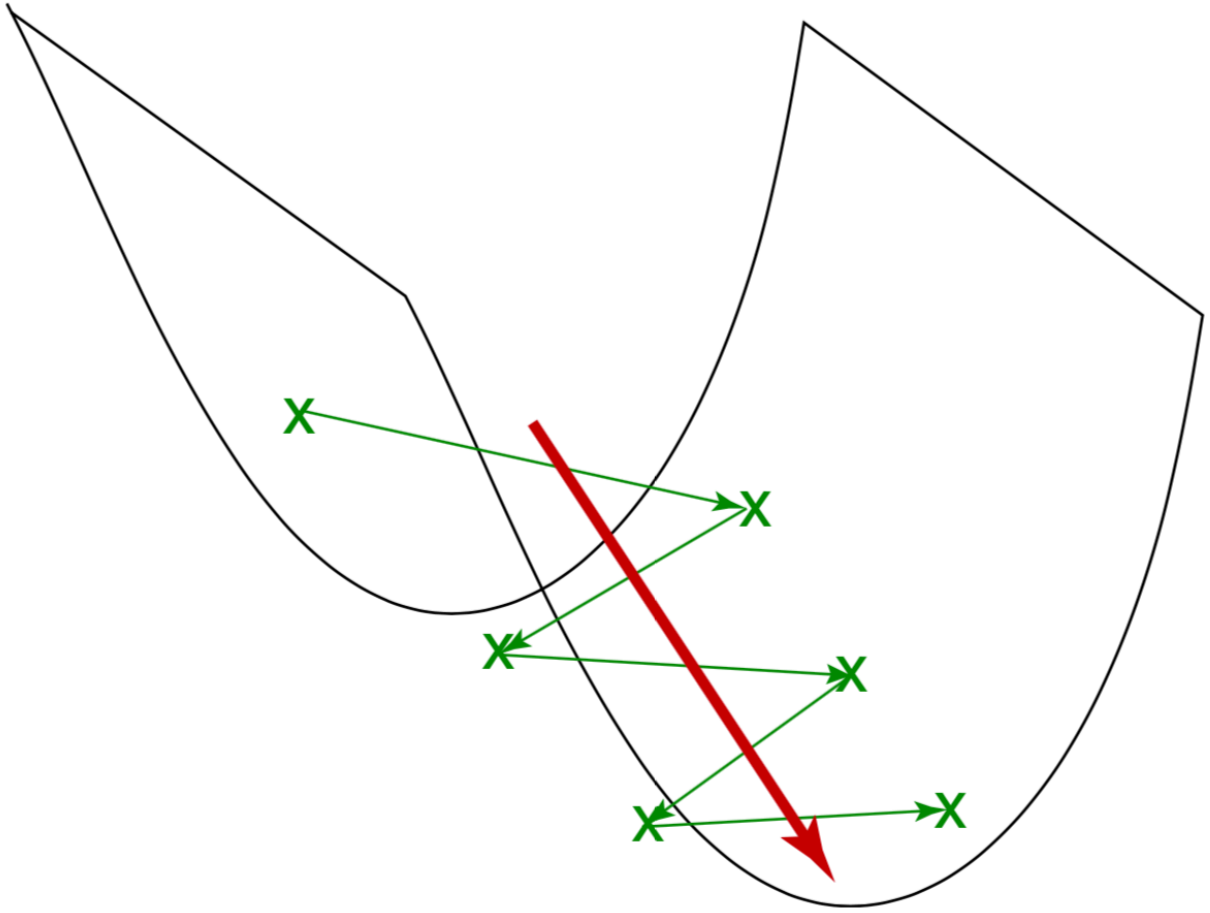
$v_1 = -\varepsilon g_1$

$v_2 = \alpha v_1 - \varepsilon g_2 = -\alpha \varepsilon g_1 - \varepsilon g_2$



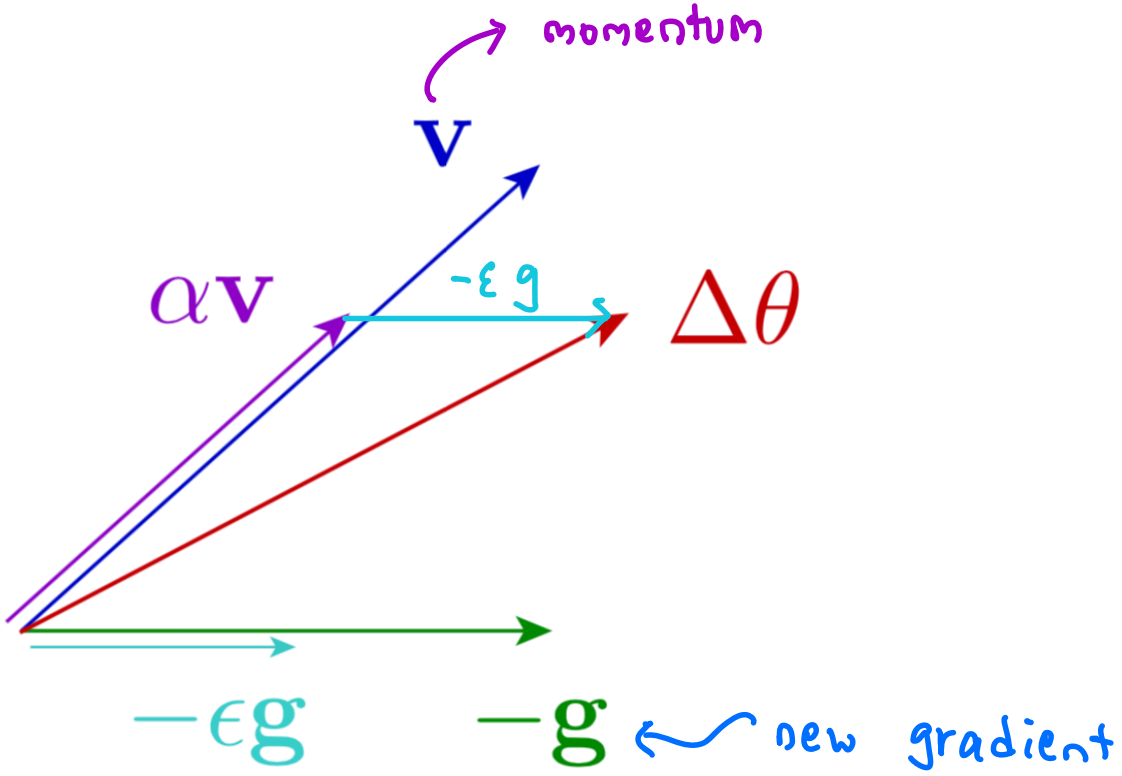
Momentum (cont.)

An example of how momentum is useful is to consider a tilted surface with high curvature. Stochastic gradient descent may make steps that zigzag, although in general it proceeds in the right direction. Momentum will average away the zigzagging components.



Momentum (cont.)

This modification augments the gradient with the running average of previous gradients, which is analogous to a gradient "momentum." The following image is appropriate to have in mind:



$v \leftarrow \alpha v - \epsilon g$

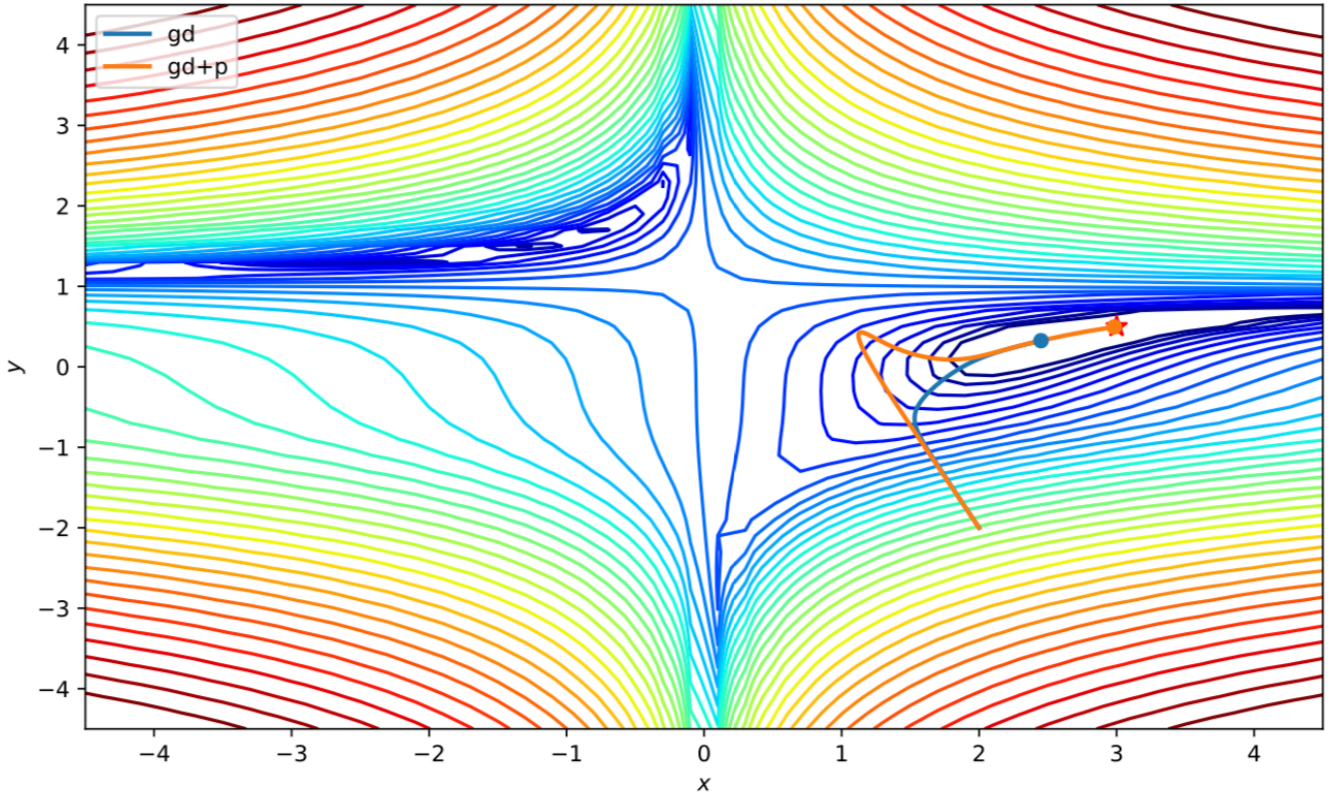
$\theta \leftarrow \theta + v$

Momentum

```
alpha = 0.9
p = 0
while last_diff > tol:
    cost, g = func(x)
    p = alpha*p - eps*g
    x += p
    last_diff = np.linalg.norm(x - path[-1])
```

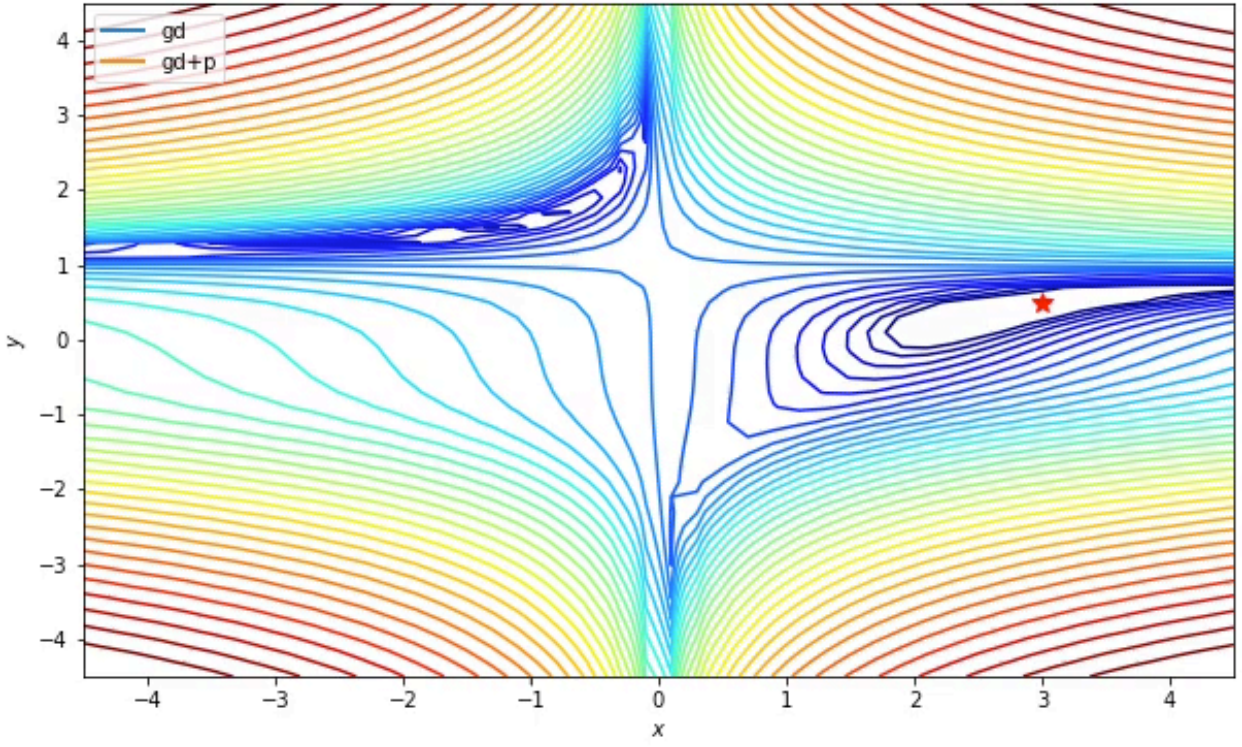
Momentum (opt 1)

gd+p denotes gradient descent with momentum.



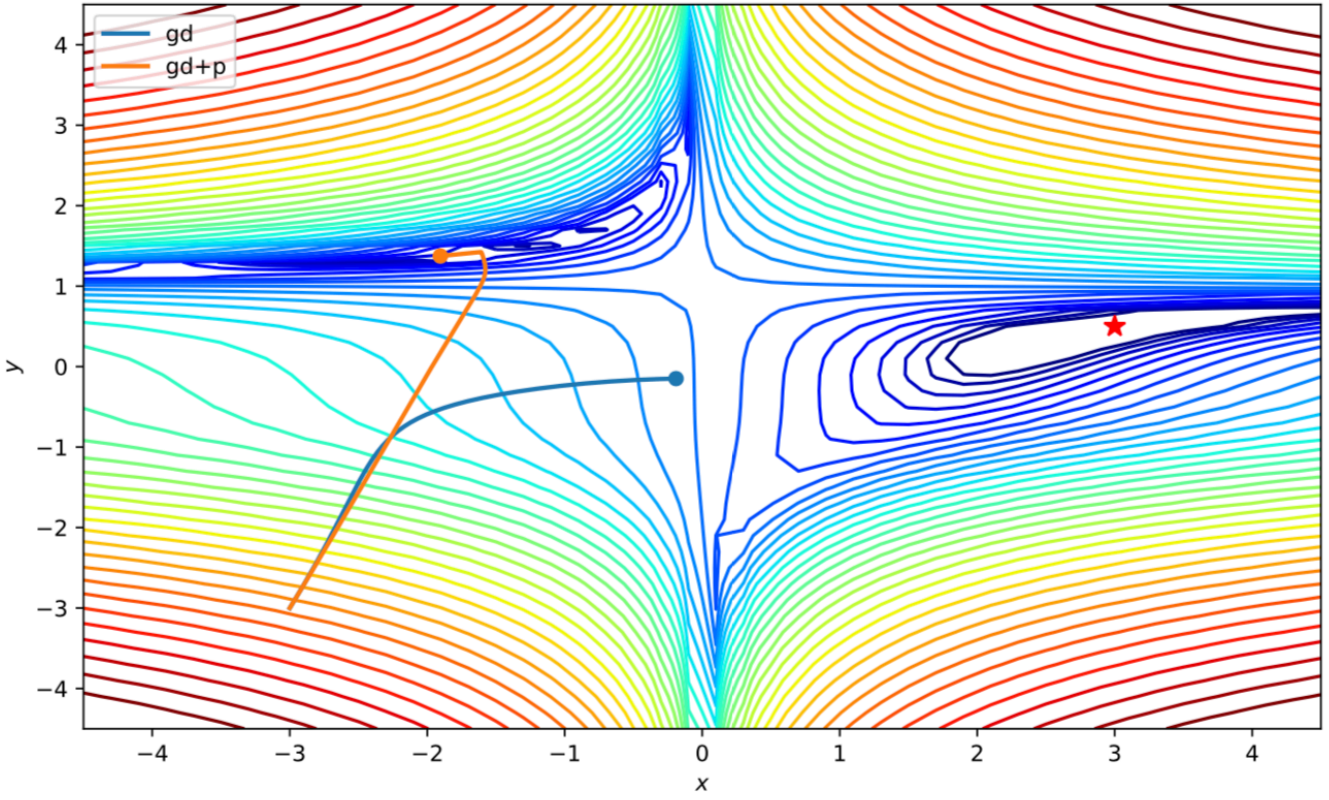
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p.mp4`

Momentum



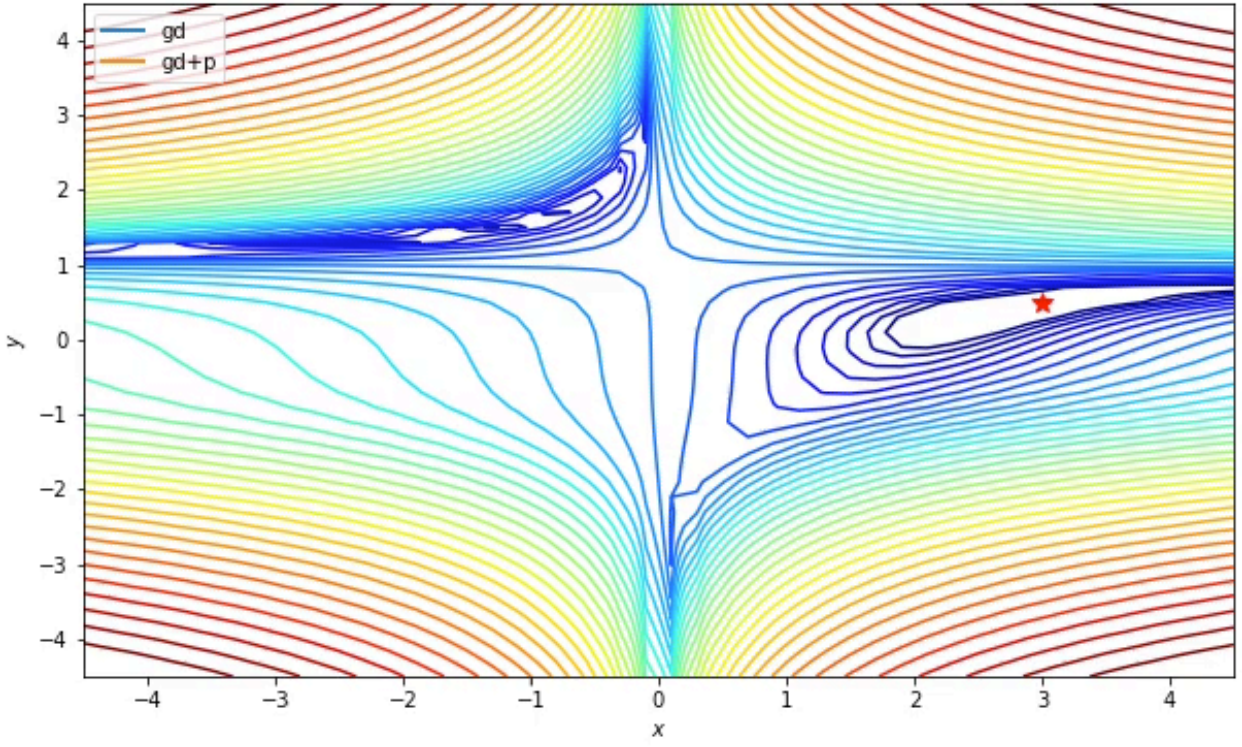
Momentum (opt 2)

Notice how momentum pushes the descent to find a local, but not global, minimum.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p.mp4`

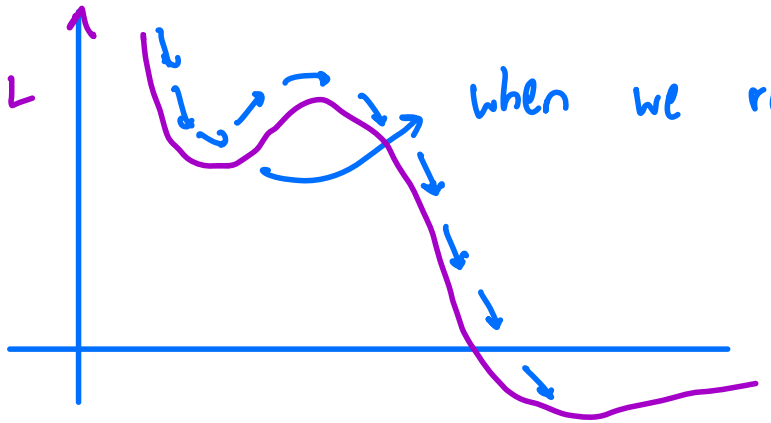
Momentum



Does momentum help with local optima?

avoid

Does momentum help with local optima?



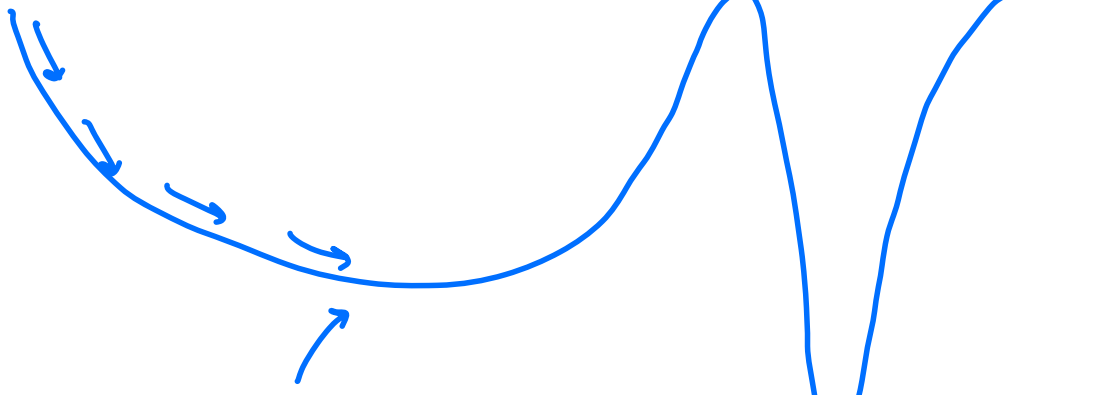
when we reach here though grad=0,

$p \neq 0$

and hence it continues to move.

What kind of local optima does momentum tend to find?

Shallow local minima

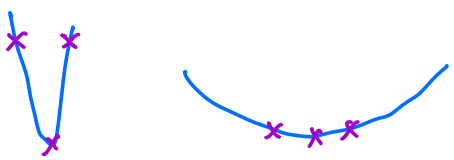


momentum might stop here (shallow local minima)

Shallow local minima

↳ less sensitive to smaller changes in the parameter setting.

↳ So, its fine.



Nesterov momentum

Nesterov momentum is similar to momentum, except the gradient is calculated at the parameter setting after taking a step along the direction of the momentum.

Initialize $\mathbf{v} = 0$. Then, until stopping criterion is met:

- Update:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \nabla_\theta J(\theta + \alpha \mathbf{v})$$

- Gradient step:

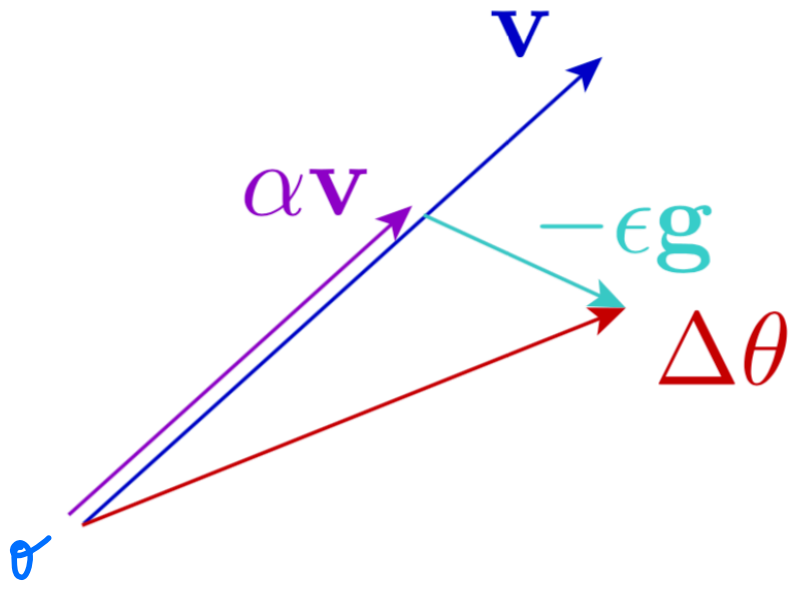
$$\theta \leftarrow \theta + \mathbf{v}$$

Classical Momentum:

$v \leftarrow \alpha v - \varepsilon \nabla_\theta J(\theta)$

Nesterov momentum (cont.)

The following image is appropriate for Nesterov momentum:



Nesterov momentum

By performing a change of variables with $\tilde{\theta}_{\text{old}} = \theta_{\text{old}} + \alpha \mathbf{v}_{\text{old}}$, it's possible to show that the following is equivalent to Nesterov momentum. (This representation doesn't require evaluating the gradient at $\theta + \alpha \mathbf{v}$.)

- Update:

$$\mathbf{v}_{\text{new}} = \alpha \mathbf{v}_{\text{old}} - \varepsilon \nabla_{\tilde{\theta}_{\text{old}}} J(\tilde{\theta}_{\text{old}})$$

- Gradient step:

$$\tilde{\theta}_{\text{new}} = \tilde{\theta}_{\text{old}} + \mathbf{v}_{\text{new}} + \alpha(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{old}})$$

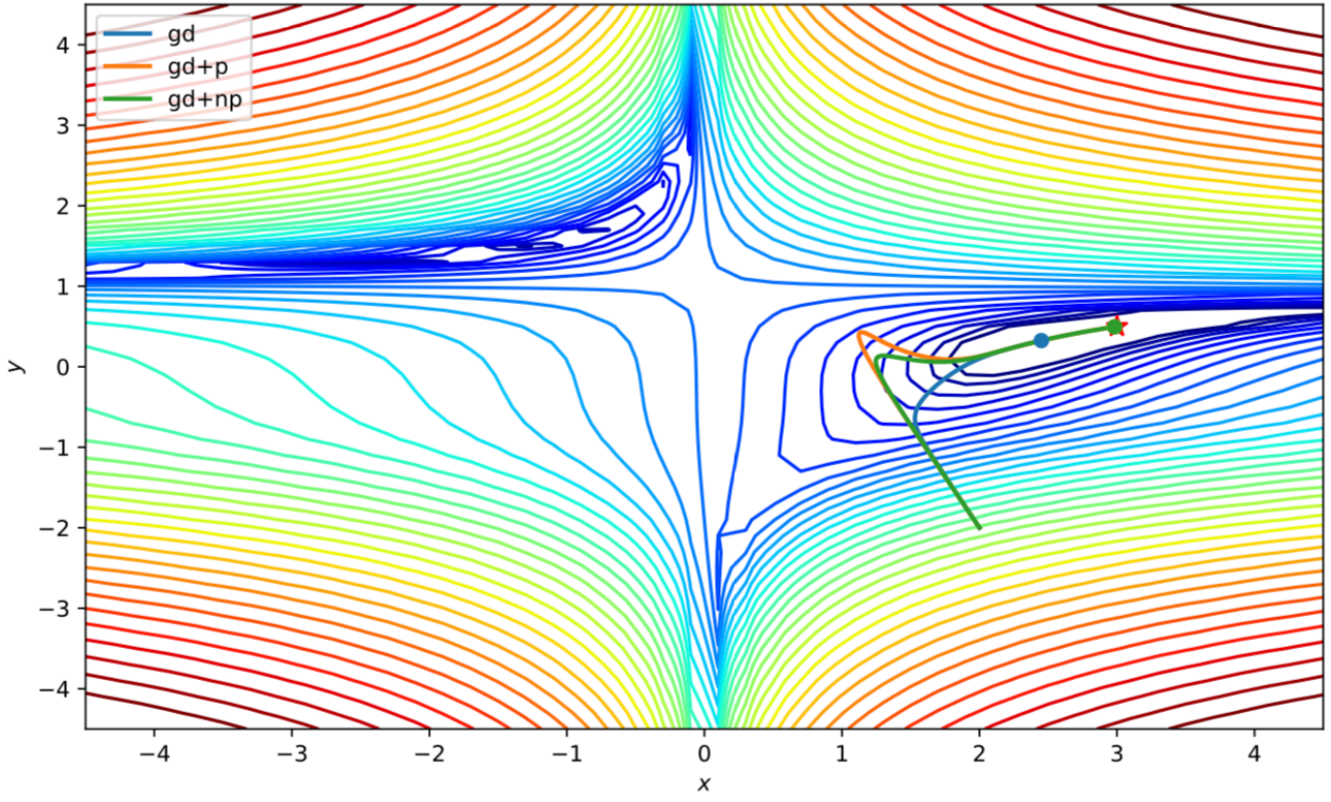
- Set $\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{old}}$, $\tilde{\theta}_{\text{new}} = \tilde{\theta}_{\text{old}}$.

Nesterov momentum

```
alpha = 0.9
p = np.zeros_like(x)
while last_diff > tol:
    cost, g = func(x)
    p_old = p
    p = alpha*p - eps*g
    x += p + alpha*(p-p_old)
    last_diff = np.linalg.norm(x - path[-1])
```

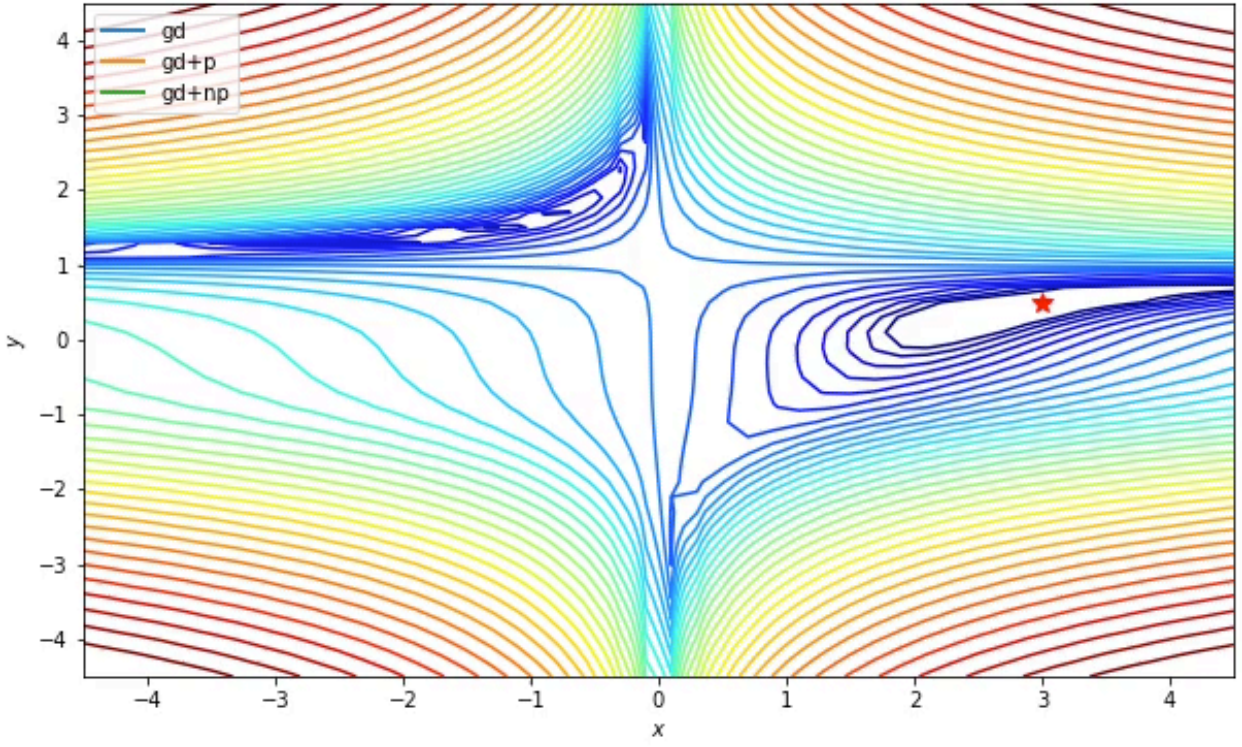
Nesterov momentum (opt 1)

gd+np denotes gradient descent with Nesterov momentum.



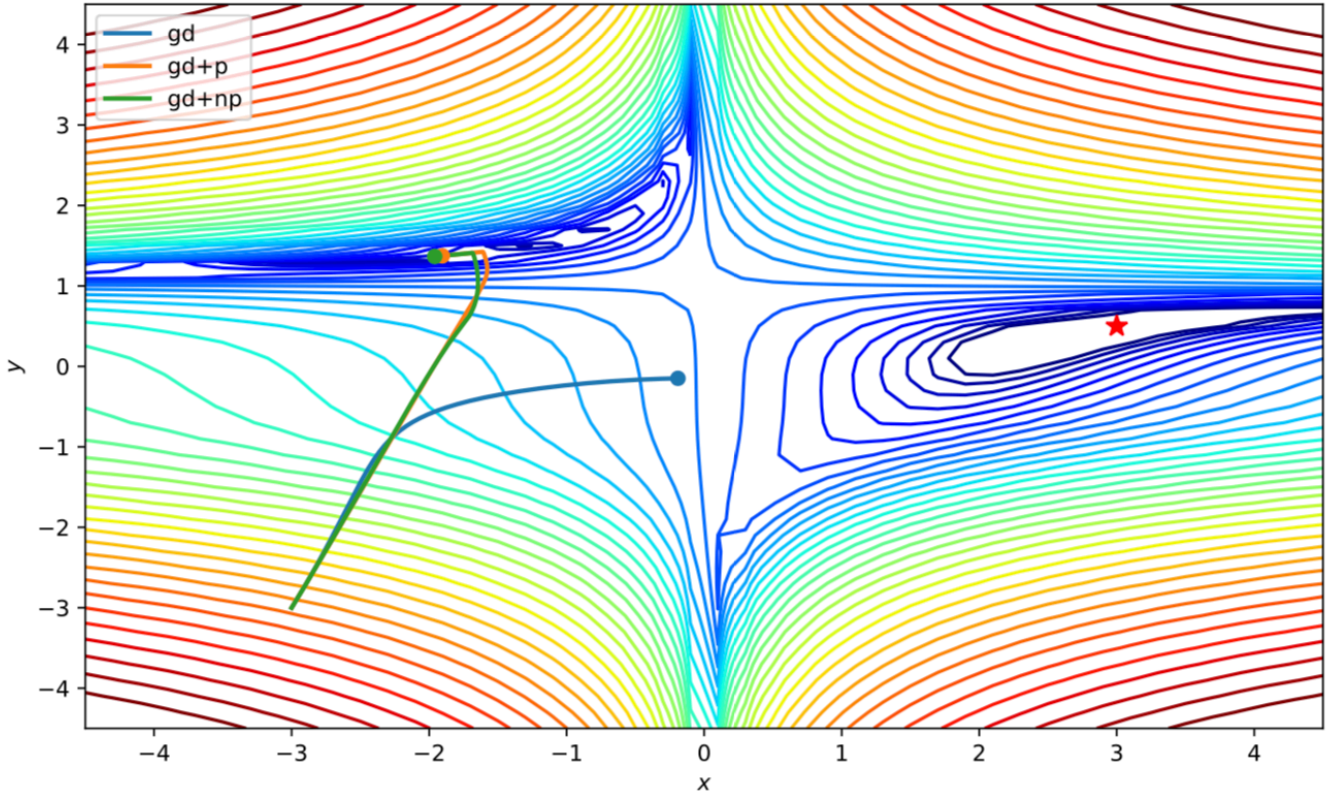
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np.mp4`

Nesterov momentum



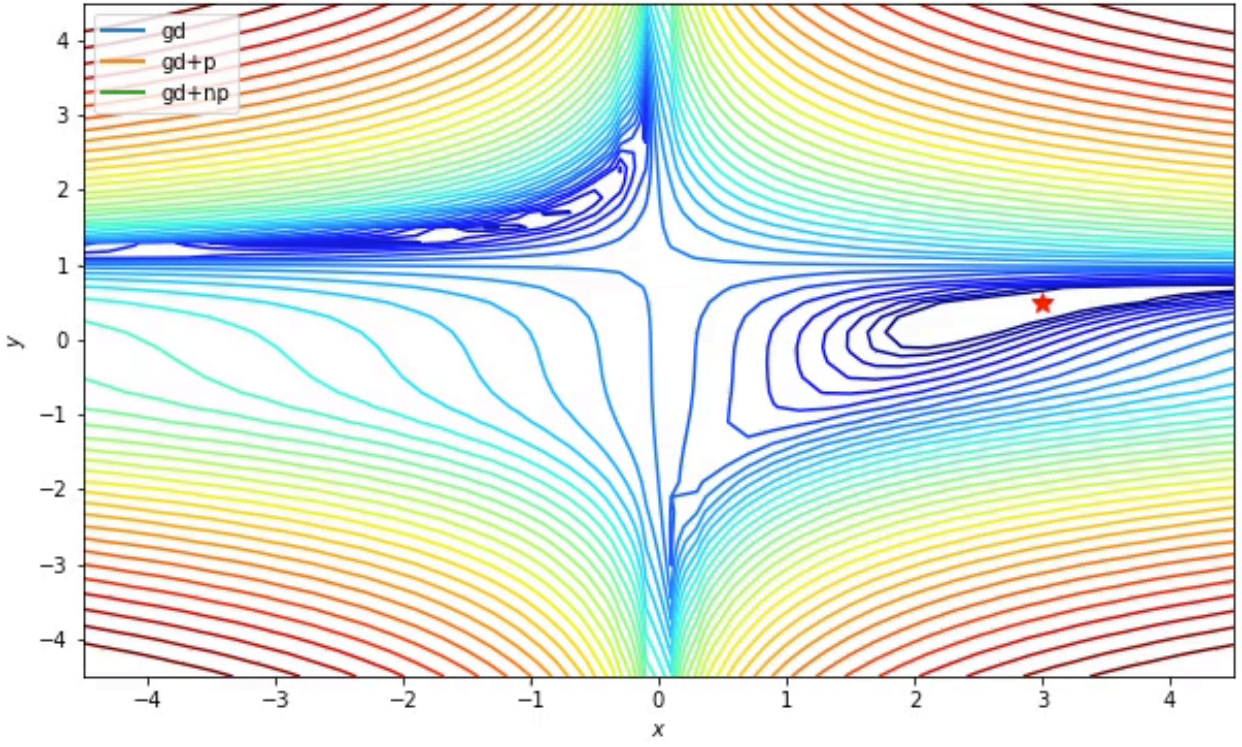
Nesterov momentum (opt 2)

Notice how Nesterov momentum finds the same local minimum as momentum.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np.mp4`

Nesterov momentum



Is there a good way to adapt the learning rule?

Techniques to adapt the learning rate

Choosing ε judiciously can be important for learning. In the beginning, a larger learning rate is typically better, since bigger updates in the parameters may accelerate learning. However, as time goes on, ε may need to be small to be able to make appropriate updates to the parameters. We mentioned before that often times, one applies a decay rule to the learning rate. This is called *annealing*. A common form to anneal the learning rate is to do so manually when the loss plateaus, or to anneal it after a set number of epochs of gradient descent.

Another approach is to update the learning rate based off of the history of gradients.

Adaptive gradient (Adagrad)

Adaptive gradient (Adagrad) is a form of stochastic gradient descent where the learning rate is decreased through division by the historical gradient norms. We will let the variable a denote a running sum of squares of gradient norms.

Initialize $\mathbf{a} = 0$. Set ν at a small value to avoid division by zero (e.g., $\nu = 1e-7$). Then, until stopping criterion is met:

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$$

↗ hadamard

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

n params

$g \in \mathbb{R}^n = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}$ $\quad g \odot g = \begin{bmatrix} g_1^2 \\ g_2^2 \\ \vdots \\ g_n^2 \end{bmatrix}$

$$\frac{\varepsilon}{\sqrt{a} + \nu} = \begin{bmatrix} \frac{\varepsilon}{\sqrt{a_1} + \nu} \\ \frac{\varepsilon}{\sqrt{a_2} + \nu} \\ \vdots \\ \frac{\varepsilon}{\sqrt{a_n} + \nu} \end{bmatrix}$$

If we too large steps in a dimension already, then we take smaller steps later.

Cons:

a always increases.

→ step size decreases in some dimension at any point.

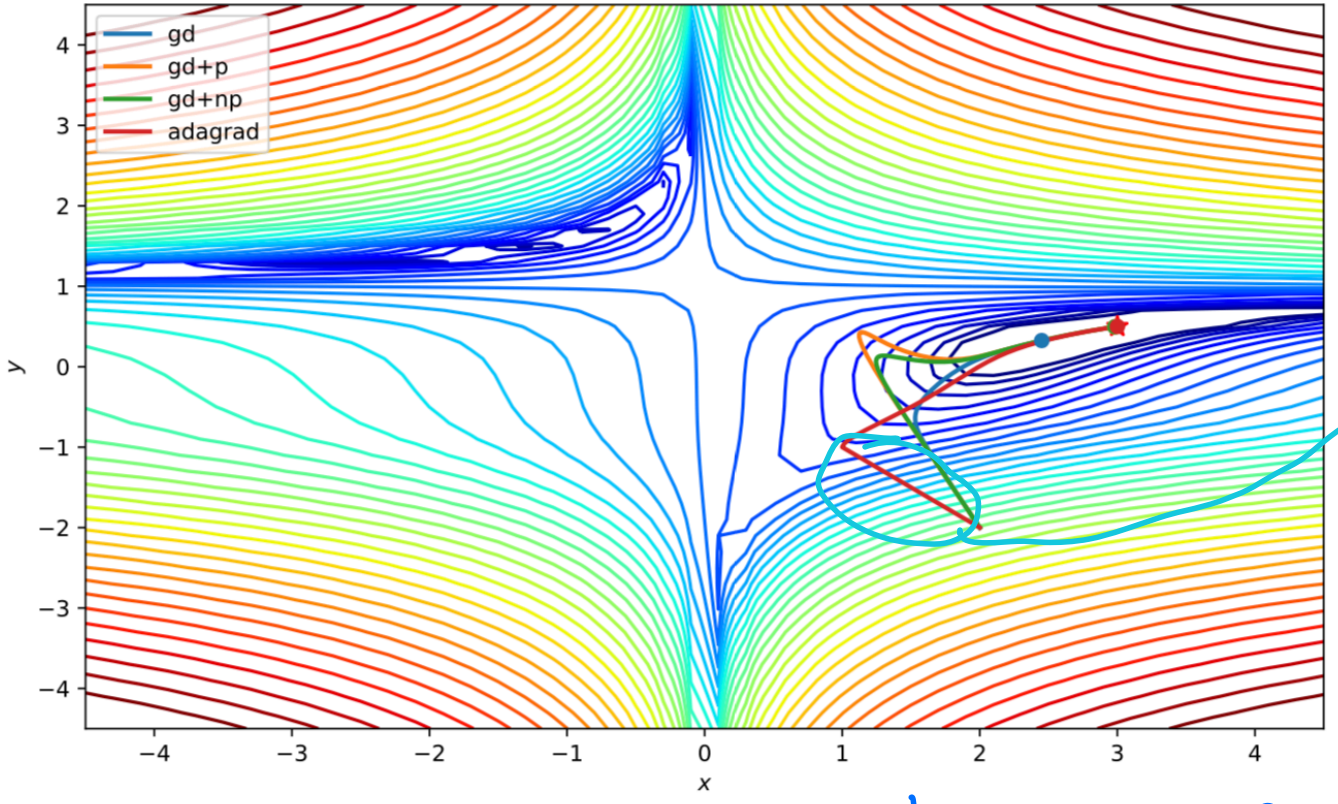
NOTE:

Adagrad does the opposite to Momentum. Momentum says go along and Adagrad reduces step size in that direction.

Adagrad

```
a = 0
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a += g*g
    x -= eps * g / (np.sqrt(a) + nu)
    last_diff = np.linalg.norm(x - path[-1])
```

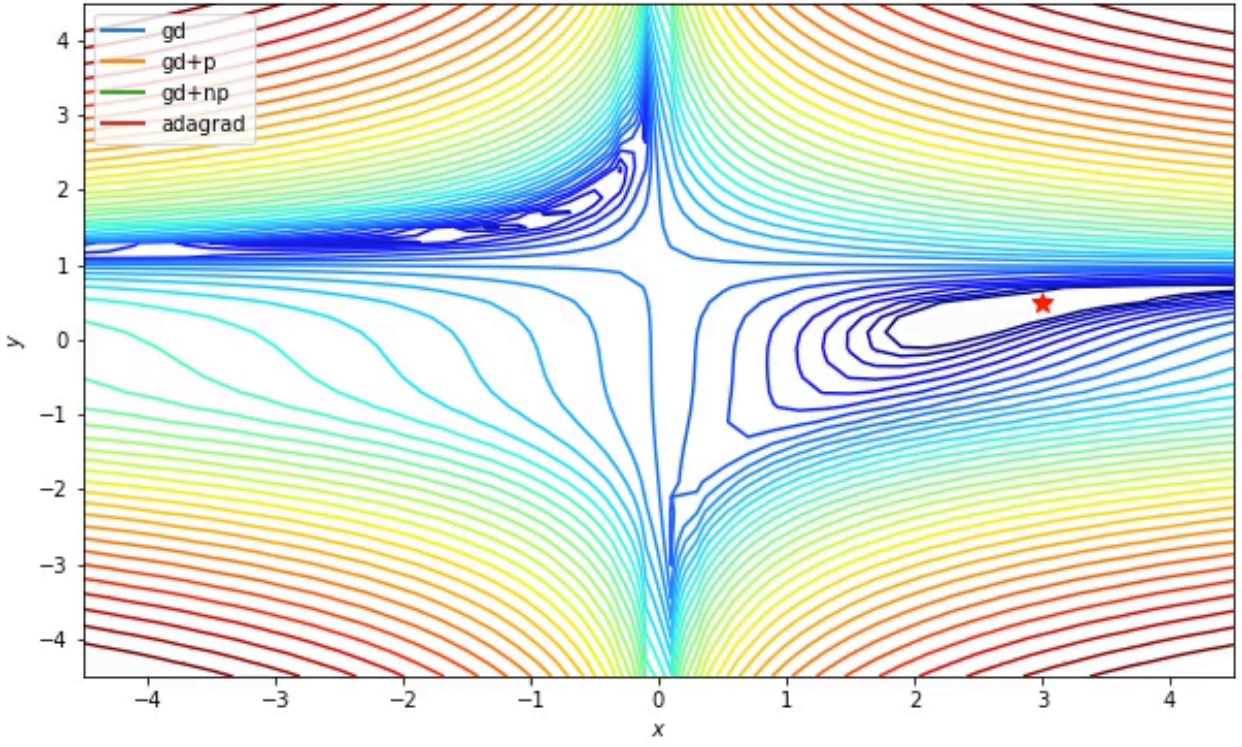
Adagrad (opt 1)



Video: http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad.mp4

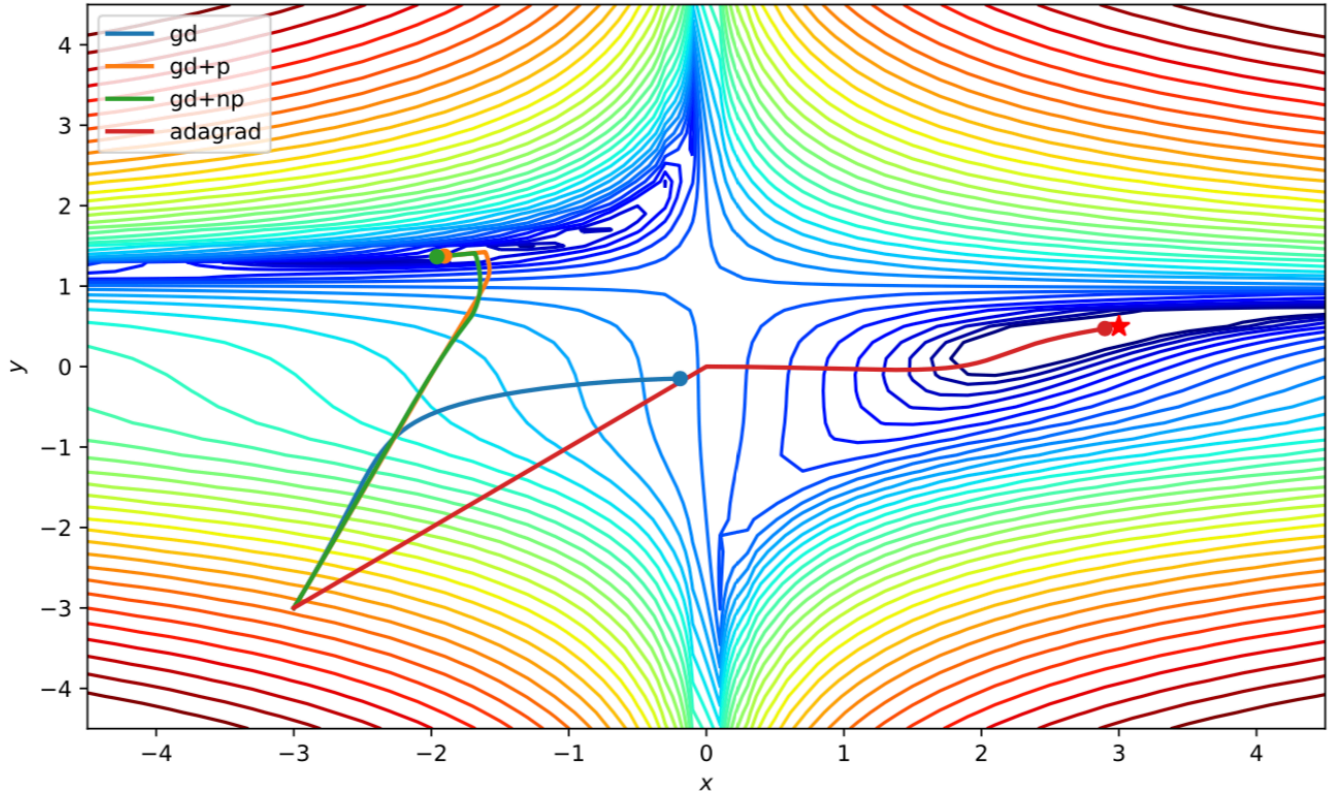
Why? As it is moving up first, it has moved a lot along y-axis. So now it has a higher step size along x-axis and hence moves left.

Adagrad



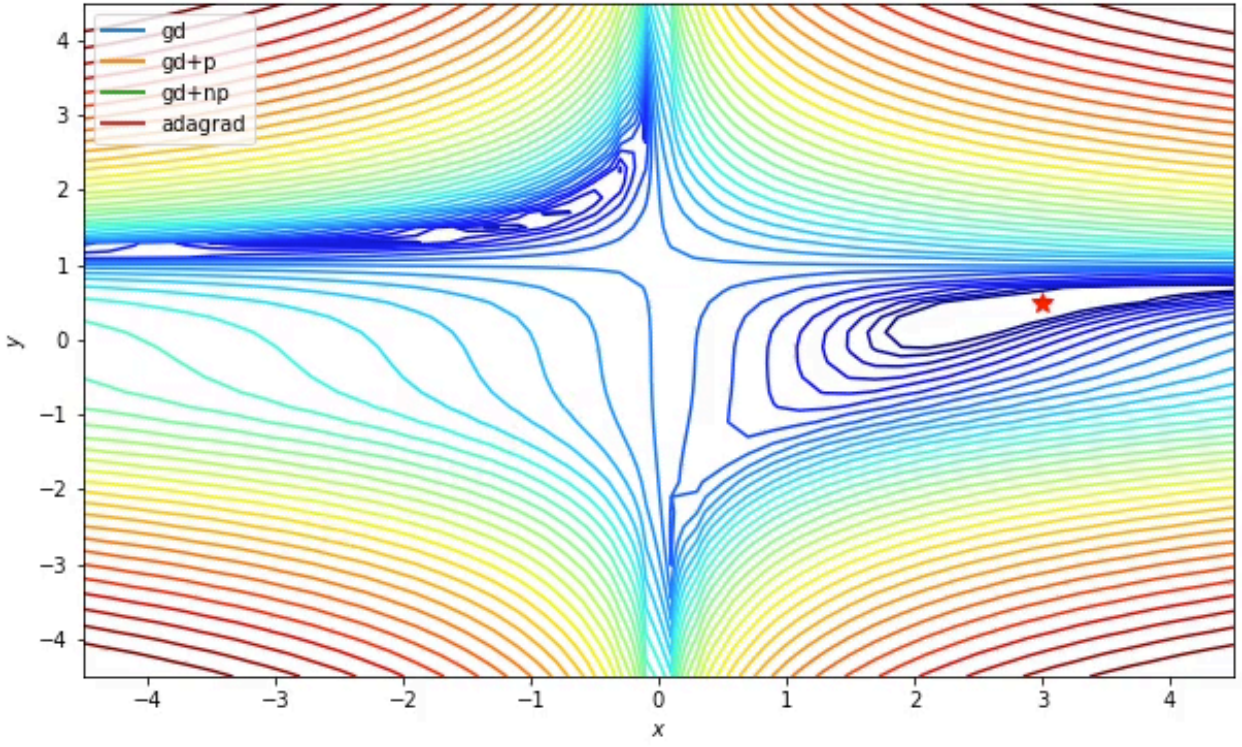
Adagrad (opt 2)

Adagrad proceeds to the global minimum.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad.mp4`

Adagrad



Adagrad

Is there a problem with adagrad?

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

RMSProp

RMSProp augments Adagrad by making the gradient accumulator an exponentially weighted moving average.

Initialize $\mathbf{a} = 0$ and set ν to be sufficiently small. Set β to be between 0 and 1 (typically a value like 0.99). Then, until stopping criterion is met:

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \beta \mathbf{a} + (1-\beta)\mathbf{g} \odot \mathbf{g}$$

→ Allows a to decrease unlike Adagrad.

- Gradient step:

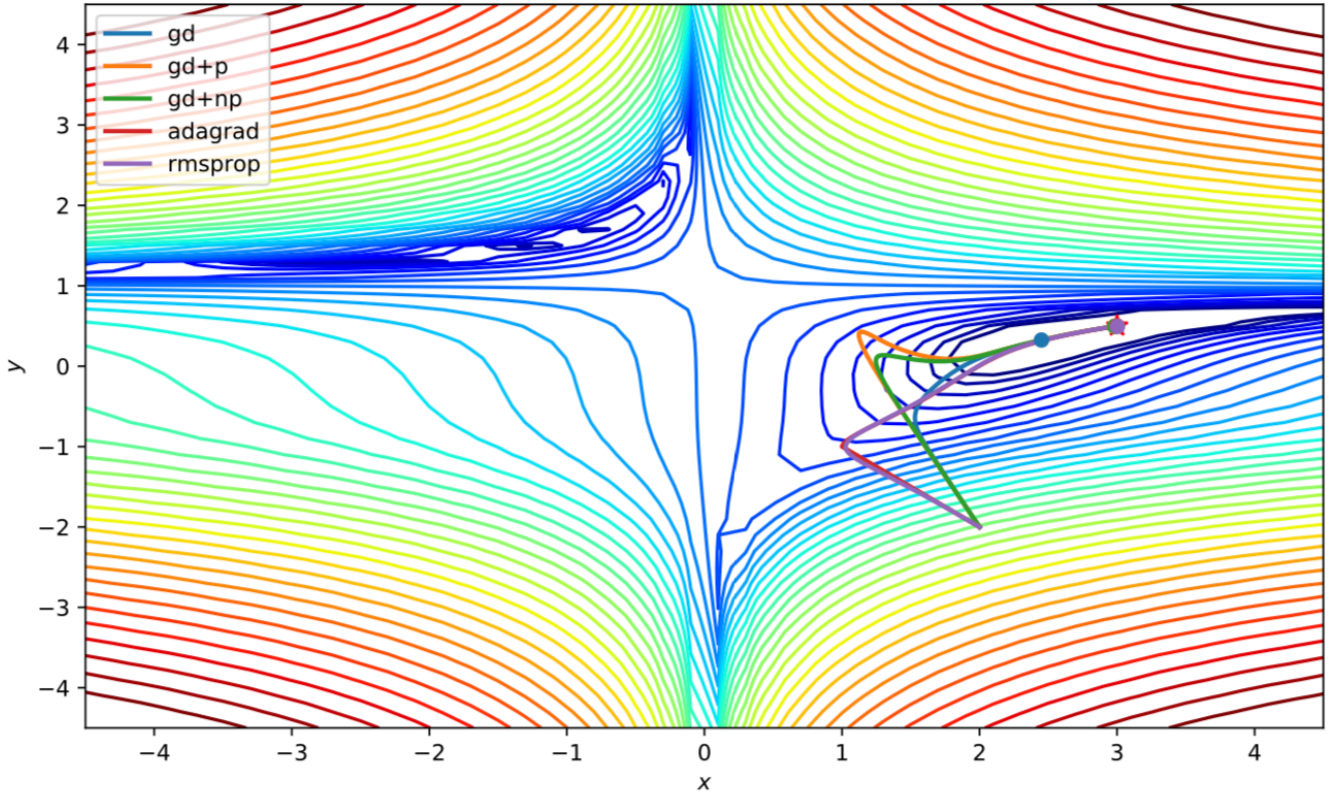
$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

So it can take larger steps than Adagrad and hence be faster.

RMSProp

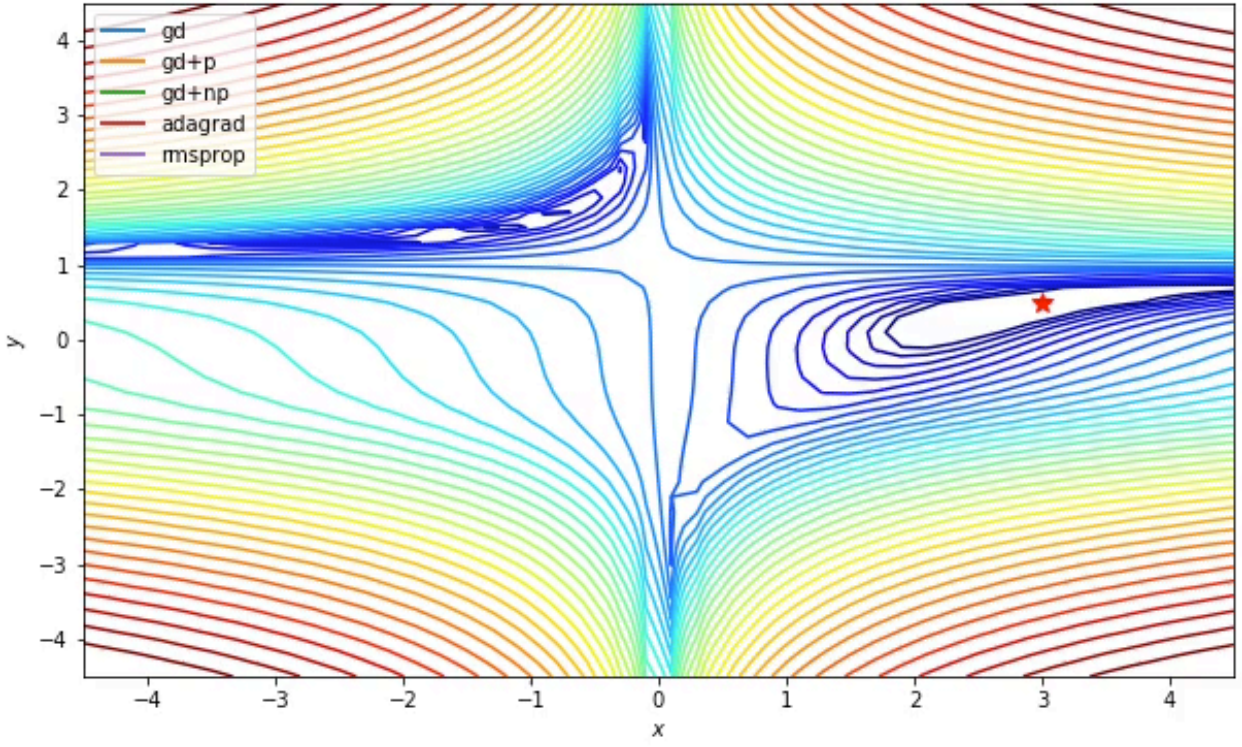
```
a = 0
beta = 0.99
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a = beta * a + (1-beta) * g * g
    x -= eps * g / (np.sqrt(a + nu))
    last_diff = np.linalg.norm(x - path[-1])
```

RMSProp (opt 1)



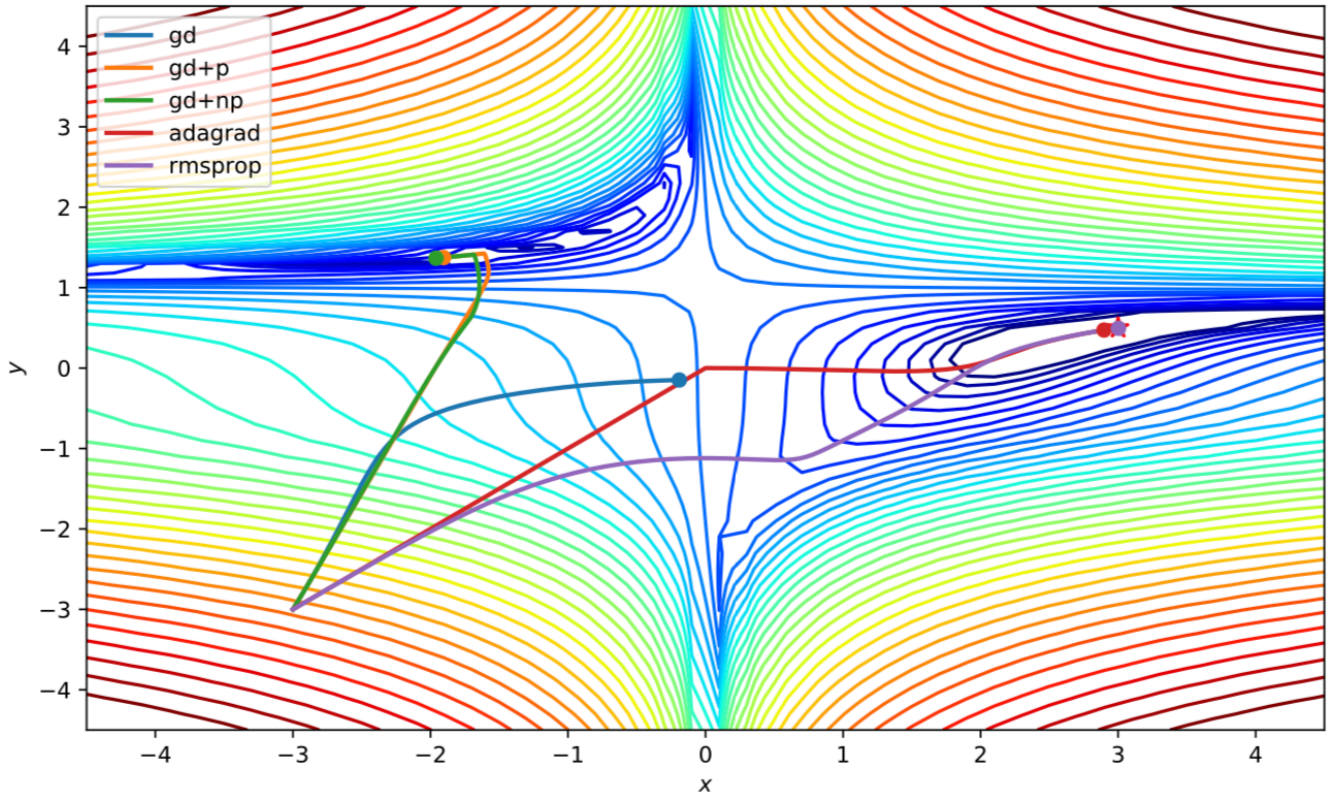
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop.mp4`

RMSProp



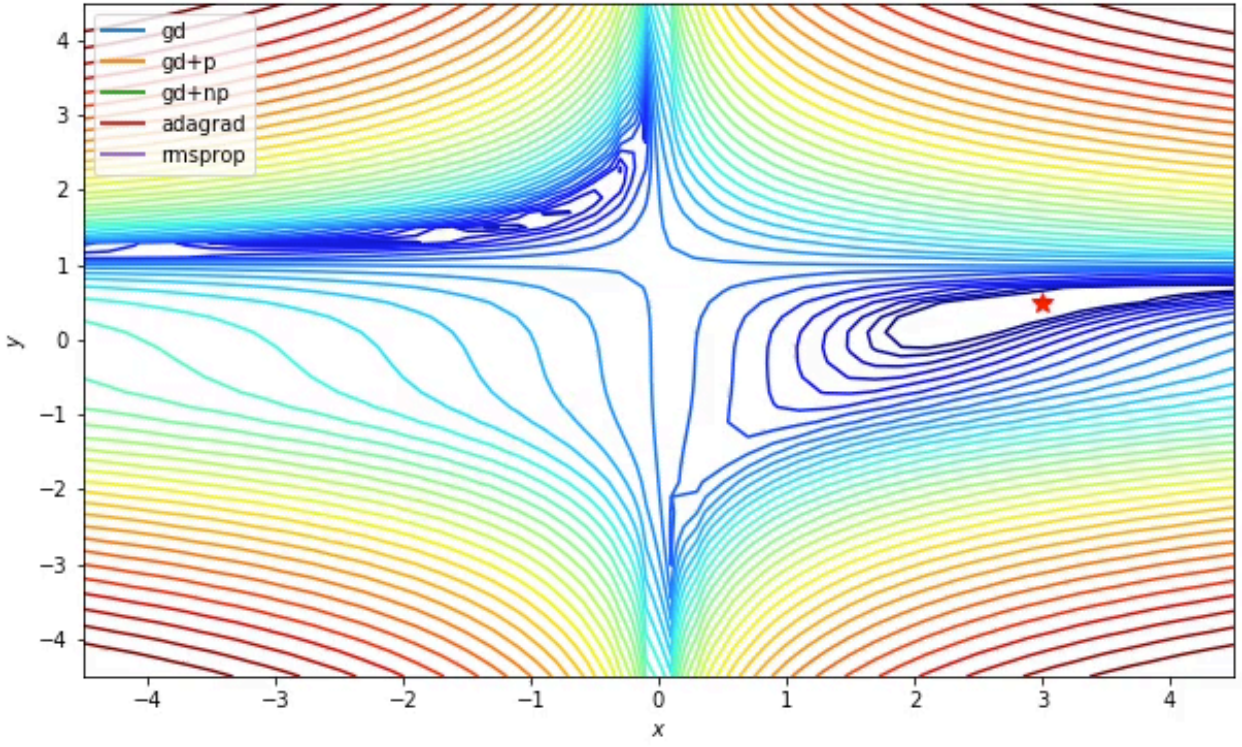
RMSProp (opt 2)

RMSProp proceeds to the global minimum, and in the video you can see it does so more quickly than Adagrad.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad_rmsprop.mp4`

RMSProp



RMSProp with momentum

RMSProp can be combined with momentum as follows.

Initialize $\mathbf{a} = 0$. Set α, β to be between 0 and 1. Set $\nu = 1e-7$. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Accumulate gradient:

$$\mathbf{a} \leftarrow \beta \mathbf{a} + (1-\beta)\mathbf{g} \odot \mathbf{g}$$

- Momentum:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \frac{\varepsilon}{\sqrt{\mathbf{a}+\nu}} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta + \mathbf{v}$$

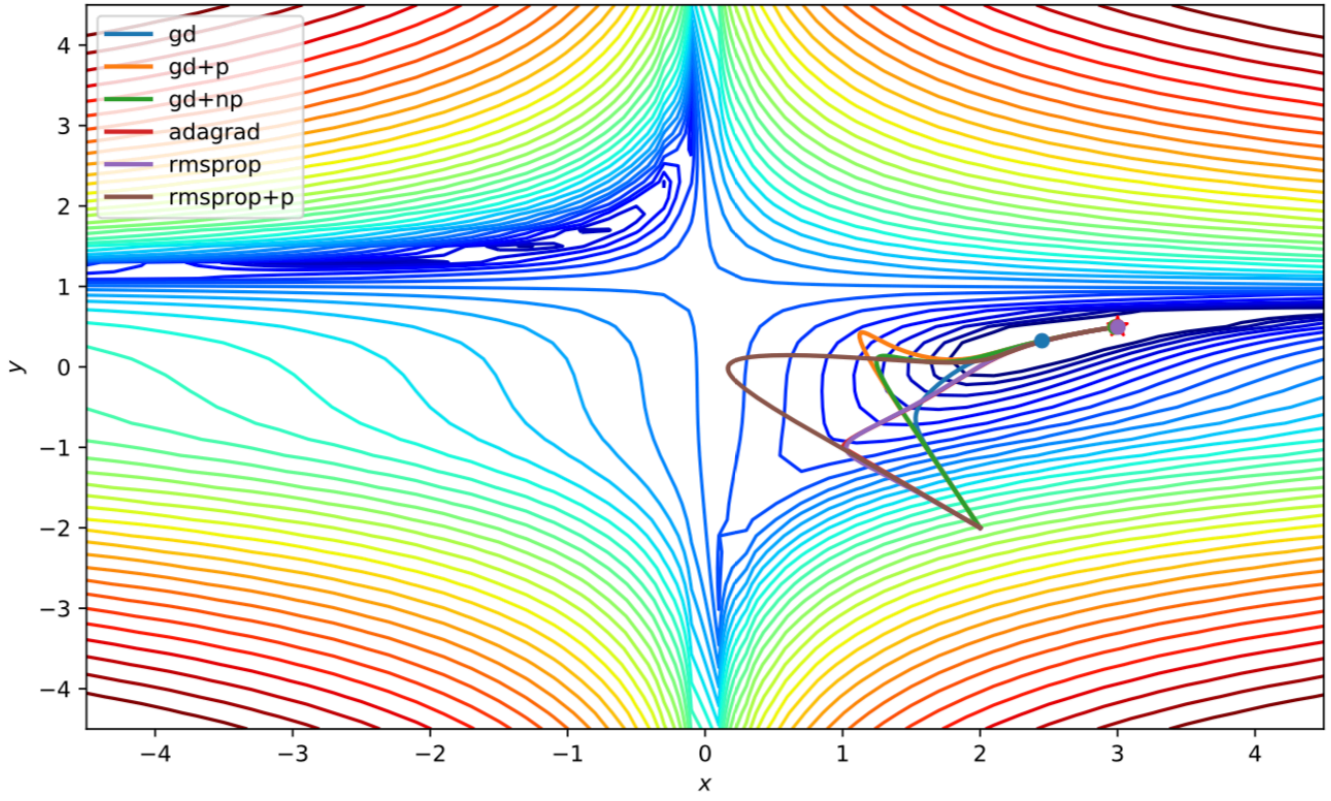
It is also possible to RMSProp with Nesterov momentum.

RMSProp + momentum

```
a = 0
p = 0
alpha = 0.9
beta = 0.99
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a = beta * a + (1-beta) * g * g
    p = alpha * p - eps * g / (np.sqrt(a) + nu)
    x += p
    last_diff = np.linalg.norm(x - path[-1])
```

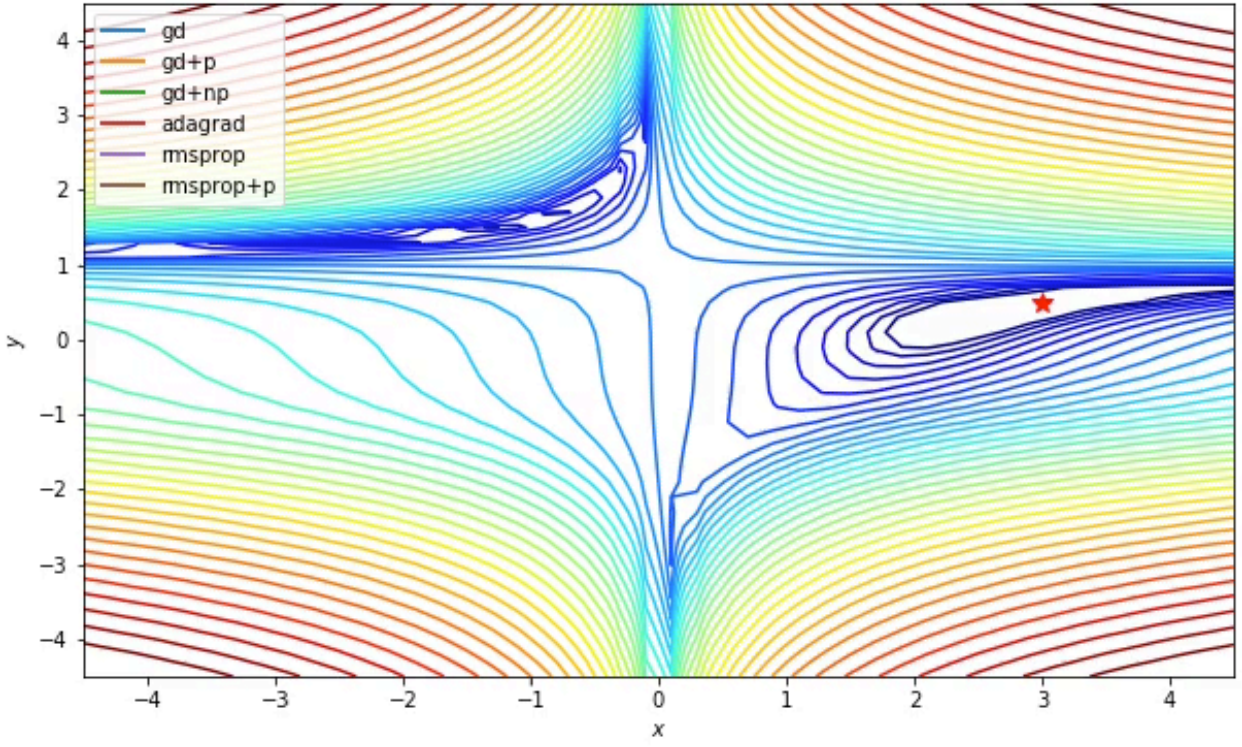
RMSProp with momentum (opt 1)

rmsprop+p denotes RMSProp with momentum. Though it makes a larger excursion out of the way, it gets to the optimum more quickly than all other optimizers. This is more apparent in the 2nd optimizer.



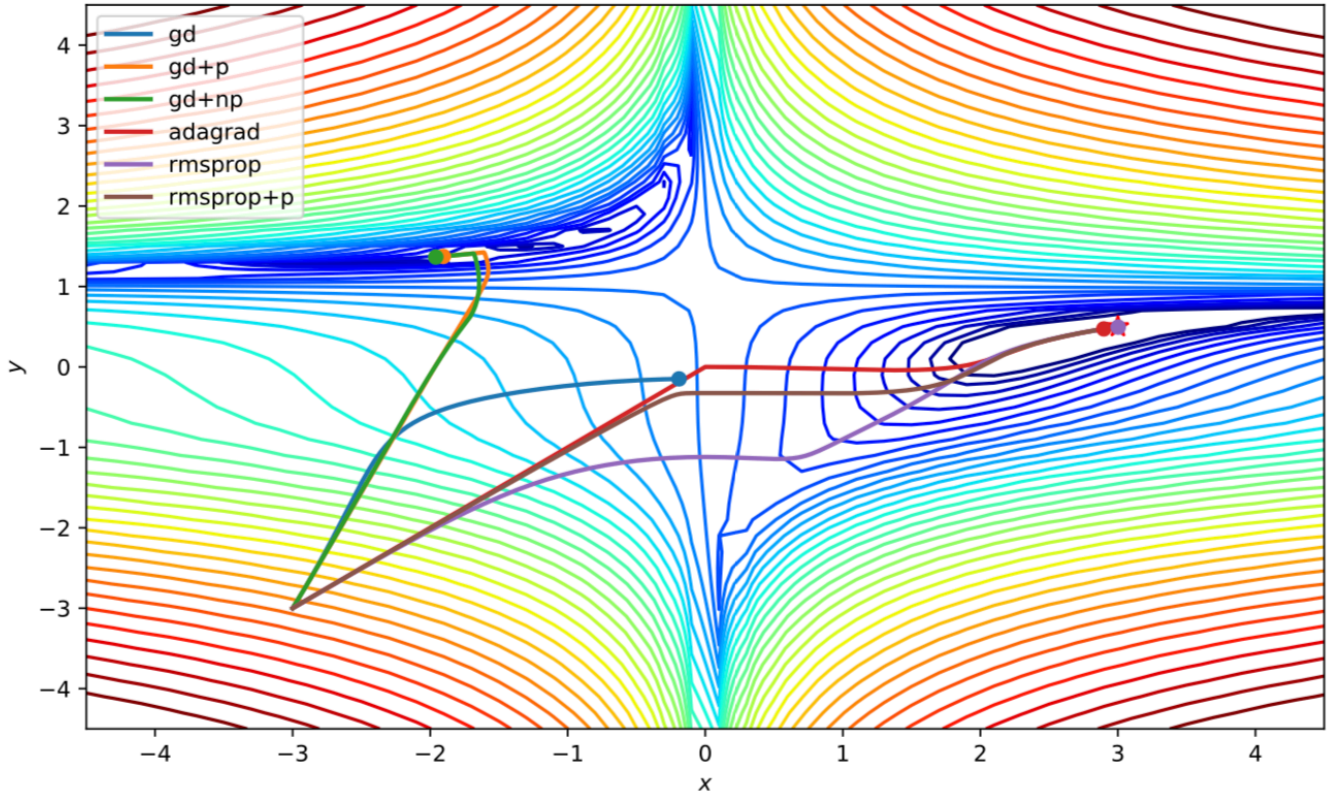
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop+p.mp4`

RMSProp + momentum



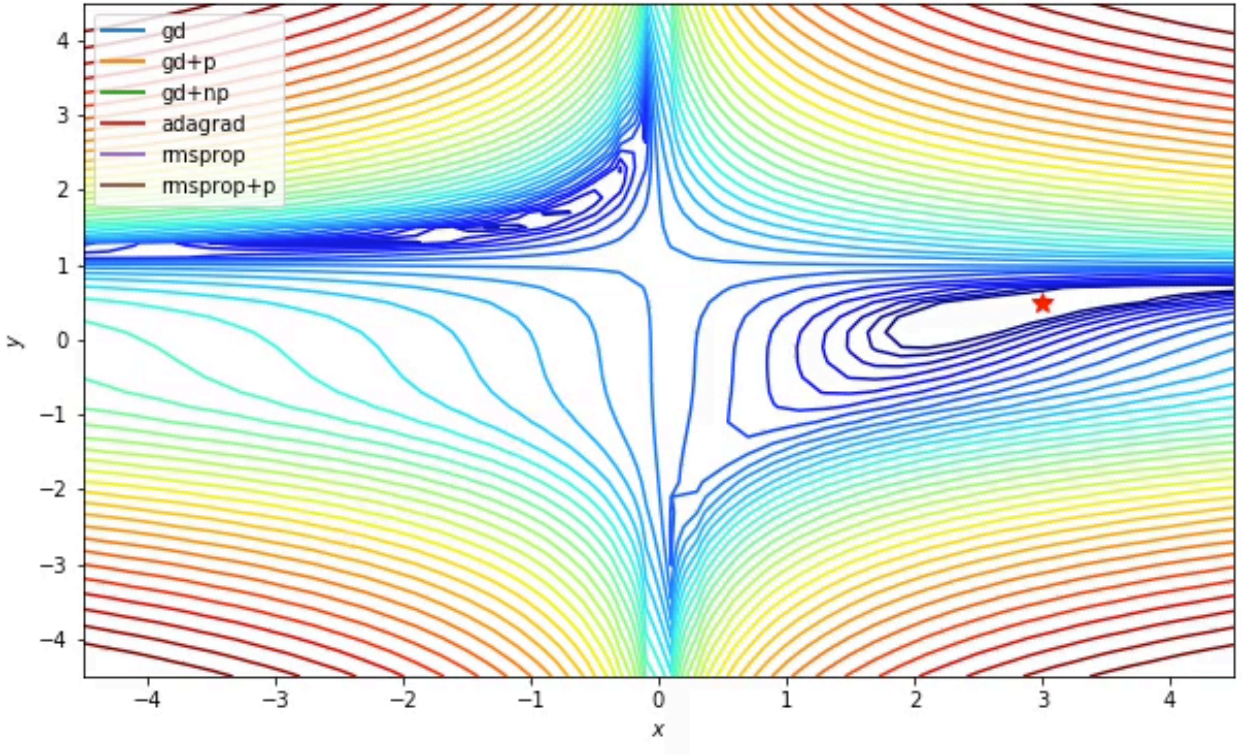
RMSProp (opt 2)

RMSProp proceeds to the global minimum, and in the video you can see it does so more quickly than Adagrad.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad_rmsprop+p.mp4`

RMSProp + momentum



Adam

Adaptive moments without bias correction

The adaptive moments optimizer (Adam) is one of the most commonly used (and robust to e.g., hyperparameter choice) optimizers. Adam is composed of a momentum-like step, followed by an Adagrad/RMSProp-like step. For intuition, we first present Adam without a bias correction step.

→ best optimizer for most purposes

Adam with no bias correction

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set β_1 and β_2 to be between 0 and 1. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize ν to be sufficiently small. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- First moment update (momentum-like):

$$\mathbf{v} \leftarrow \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

Momentum

- Second moment update (gradient normalization):

$$\mathbf{a} \leftarrow \beta_2 \mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$$

RMSProp

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{v}$$

Adaptive moments (Adam)

Adam incorporates a bias correction on the moments. The intuition for the bias correction is to account for initialization; these bias corrections amplify the second moments, so that extremely large steps are not taken at the start of the optimization.

Adam (cont.)

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set β_1 and β_2 to be between 0 and 1. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize ν to be sufficiently small. Initialize $t = 0$. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Time update: $t \leftarrow t + 1$
- First moment update (momentum-like):

$$\mathbf{v} \leftarrow \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

- Second moment update (gradient normalization):

$$\mathbf{a} \leftarrow \beta_2 \mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$$

- Bias correction in moments:

$$\tilde{\mathbf{v}} = \frac{1}{1 - \beta_1^t}\mathbf{v}$$

$$\tilde{\mathbf{a}} = \frac{1}{1 - \beta_2^t}\mathbf{a}$$

As $t \to \infty$ bias is gone

- Gradient step:

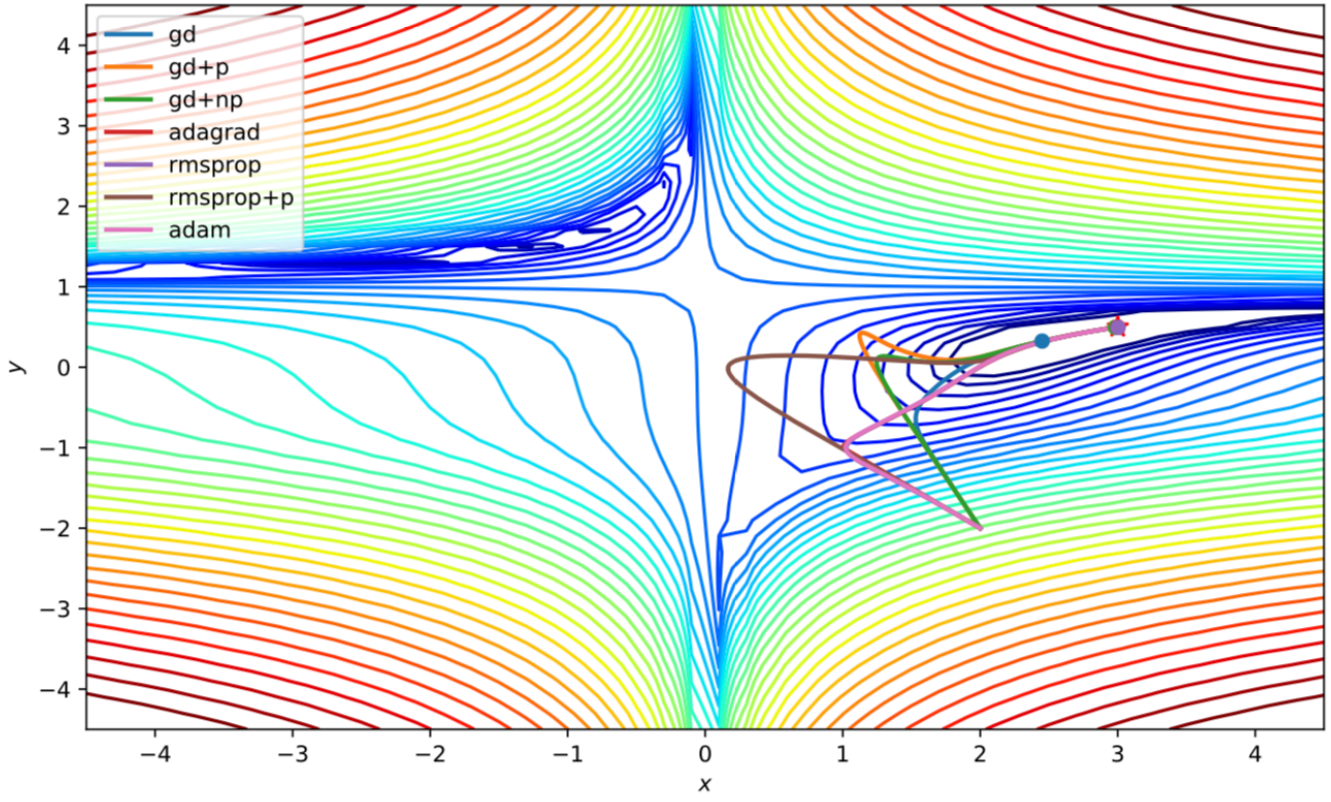
$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\tilde{\mathbf{a}}} + \nu} \odot \tilde{\mathbf{v}}$$

Adam

```
a = 0
p = 0
beta1 = 0.9
beta2 = 0.99
nu = 1e-7
t = 0
while last_diff > tol:
    cost, g = func(x)
    t += 1
    p = beta1 * p + (1-beta1) * g
    a = beta2 * a + (1-beta2) * g * g
    p_u = p / (1 - beta1**t)
    a_u = a / (1 - beta2**t)
    x -= eps * p_u / (np.sqrt(a_u) + nu)
    last_diff = np.linalg.norm(x - path[-1])
```

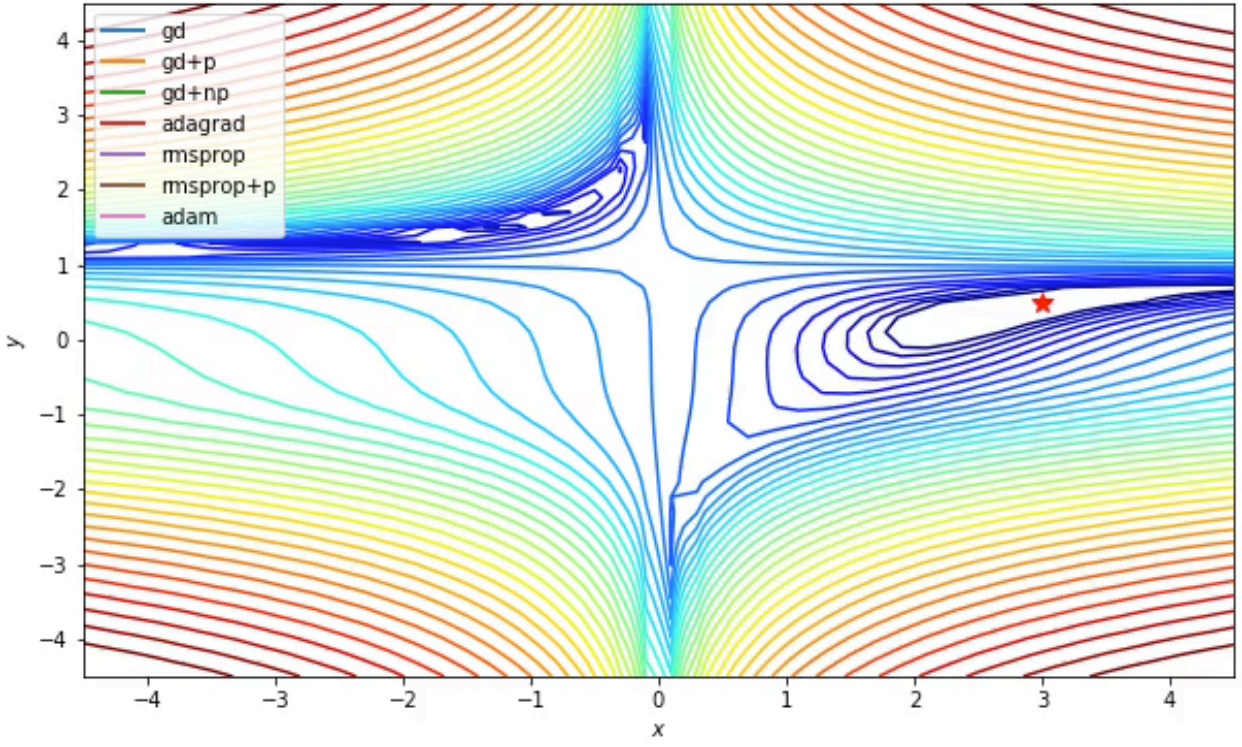
Adam (opt 1)

In both example optimizations, Adam is just slightly slower than RMSProp.

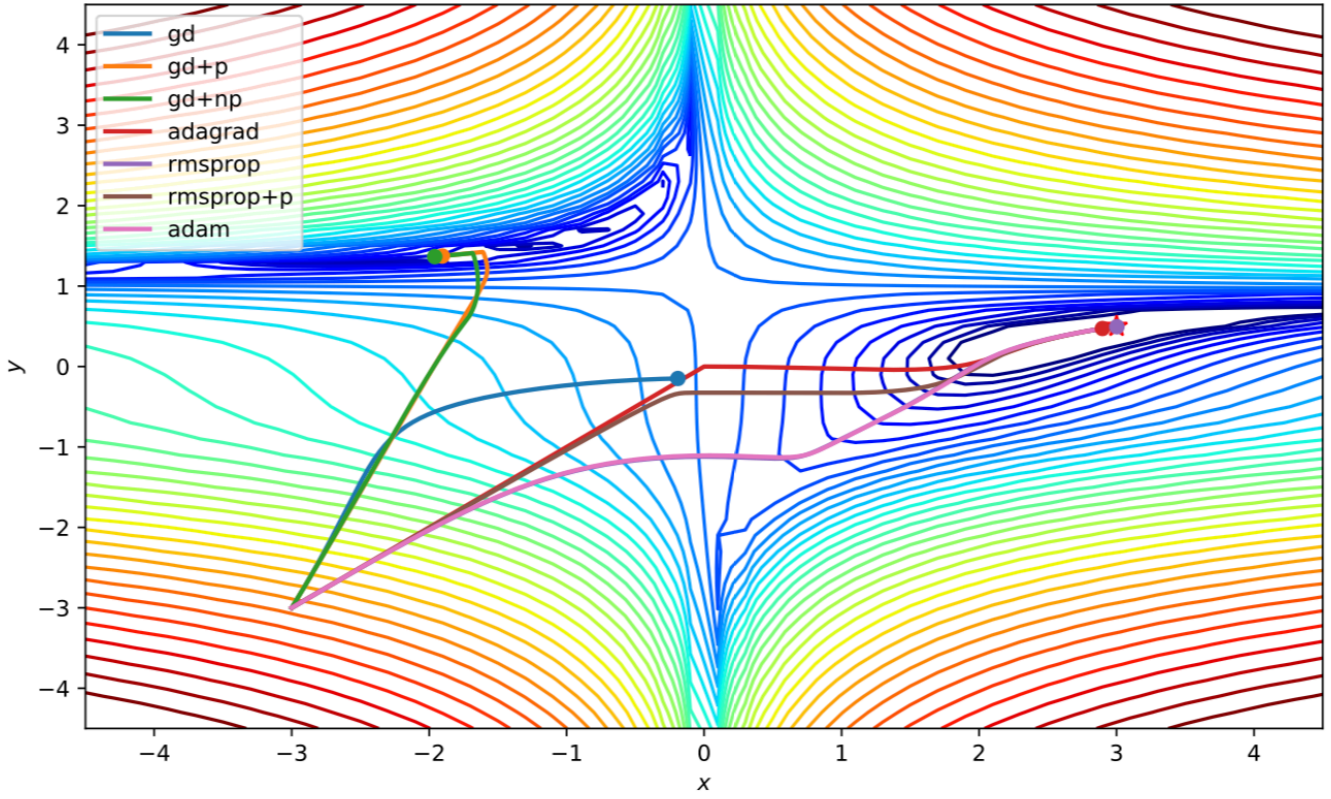


Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop+p_adam.mp4`

Adam

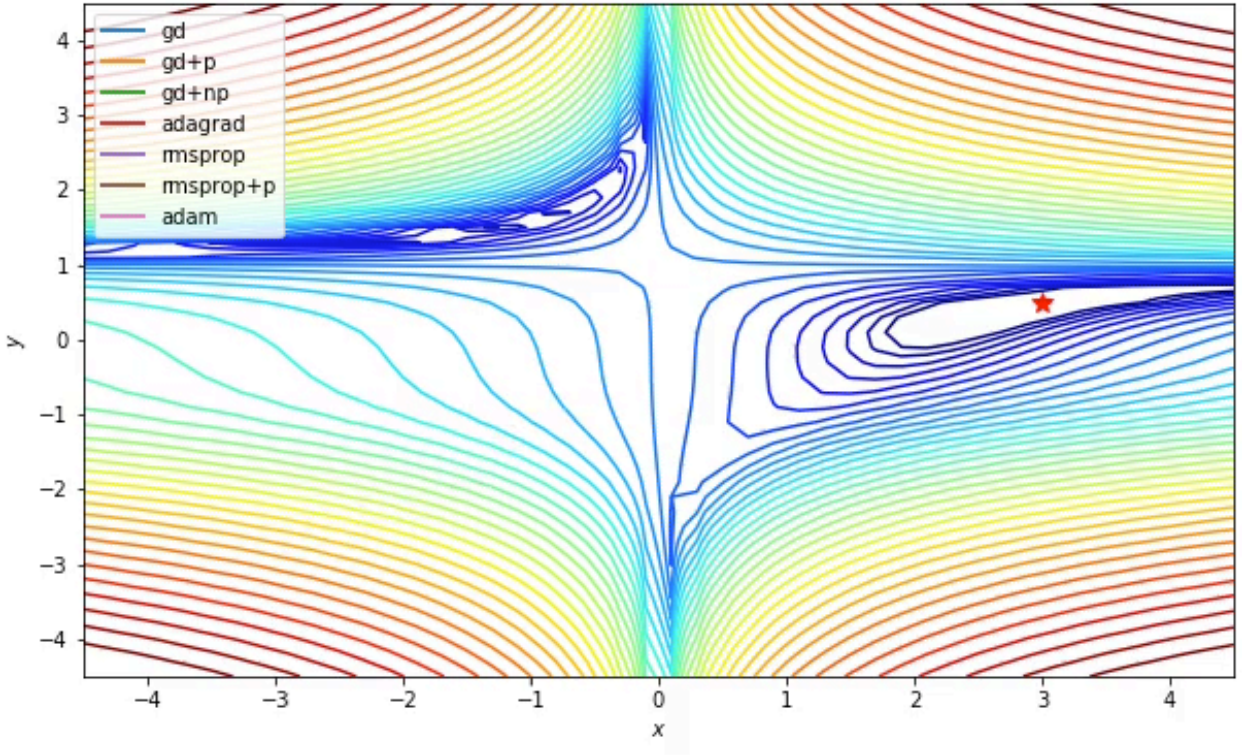

gd
gd+p
gd+np
adagrad
rmsprop
rmsprop+p
adam
4
3
2
1
0
−1
−2
−3
−4
y
−4
−3
−2
−1
0
1
2
3
4
x

Adam (opt 2)



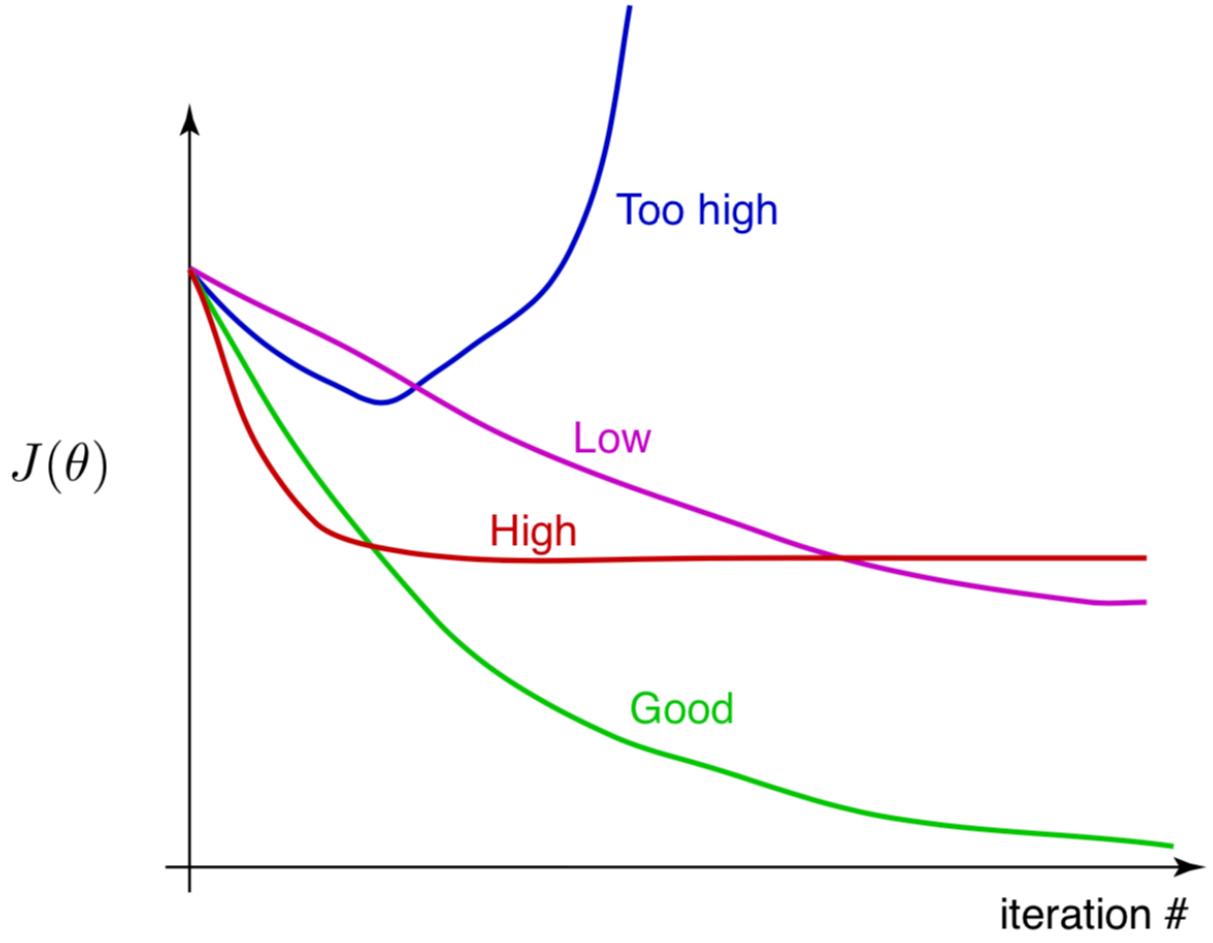
Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad_rmsprop+p_adam.mp4`

Adam


gd
gd+p
gd+np
adagrad
rmsprop
rmsprop+p
adam
y
x
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

Interpreting the cost

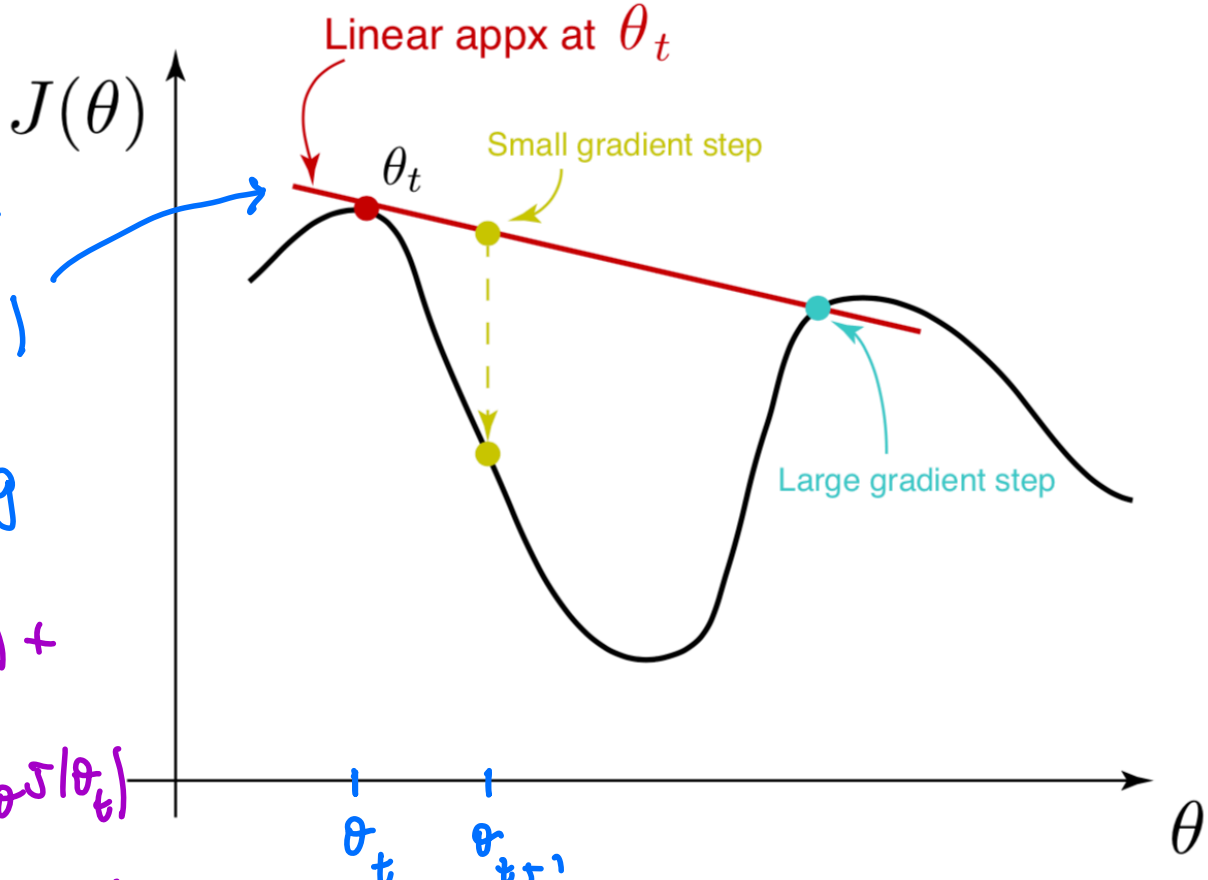
The cost function can be very informative as to how to adjust your step sizes for gradient descent.



First order vs second order methods

SGD, and optimizers used to augment the learning rate (Adagrad, RMSProp, and Adam), are all *first order* methods and have the learning rate ε as a hyperparameter.

- First-order refers to the fact that we only use the first derivative, i.e., the gradient, and take linear steps along the gradient.
- The following picture of a first order method is appropriate.



$J(\theta) = J(\theta_t) + (\theta - \theta_t)^T \nabla_\theta J(\theta_t)$

$\theta_{t+1} = \theta_t - \varepsilon g$

$J(\theta_{t+1}) = J(\theta_t) + (\theta_t - \varepsilon g - \theta_t)^T \nabla_\theta J(\theta_t)$

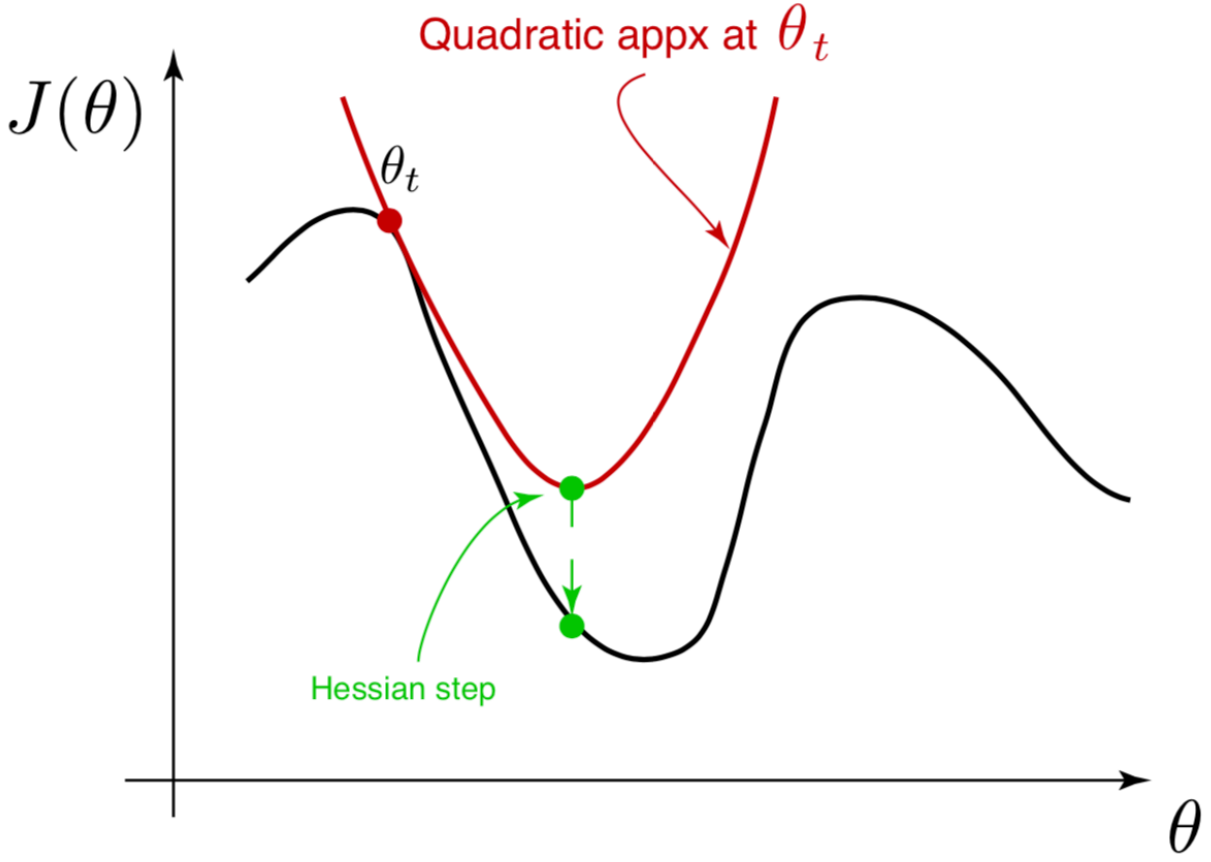
$= J(\theta_t) - \varepsilon g^T g$

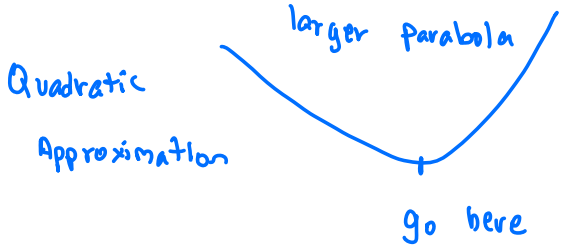
≥ 0

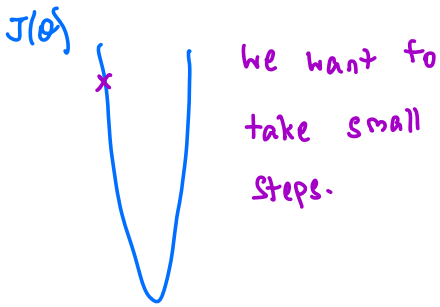
Second order methods

First order vs second order methods (cont)

It is possible to also use the *curvature* of the cost function to know how to take steps. These are called second-order methods, because they use the second derivative (or Hessian) to assess the curvature and thus take appropriate sized steps in each dimension. See following picture for intuition:







smaller parabola.

Newton's method

$$H = \begin{bmatrix} \partial^2 J/\partial\theta_1 & \partial^2 J/\partial\theta_1\partial\theta_2 \\ \partial^2 J/\partial\theta_2\partial\theta_1 & \ddots \end{bmatrix}$$

Newton's method

NOT TESTED

Second order optimization

The most widely used second order method is Newton's method. To arrive at it, consider the Taylor series expansion of $J(\theta)$ around θ_0 up to the second order terms, i.e.,

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

If this were just a second order function, we could minimize it by taking its derivative and setting it to zero:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta J(\theta_0) + \mathbf{H}(\theta - \theta_0) \\ &= \mathbf{0} \end{aligned}$$

$g + H(\theta - \theta_0) = 0$

This results in the Newton step,

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

$\theta = -H^{-1} g + \theta_0$

If the function is quadratic, this step takes us to the minimum. If not, this approximates the function as quadratic at θ_0 and goes to the minimum of the quadratic approximation.

Does this form of step make intuitive sense?

Newton's method

Newton's method, by using the curvature information in the Hessian, does not require a learning rate.

Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$ $\in \mathbb{R}^n$
- Compute Hessian: $\mathbf{H}$ $\in \mathbb{R}^{n \times n}$
- Gradient step:

$$\theta \leftarrow \theta - \mathbf{H}^{-1}\mathbf{g}$$

Why Newton's Method is not used?

(1) Memory : store the Hessian.

$n = 1 \times 10^6$

4 bytes

3.8 MB

3.8 TB Hessian

Cannot fit in RAM.

(2) Invert the Hessian $O(n^3)$

(3) Hessian typically requires a large batch

Newton's method (cont.)

A few notes about Newton's method:

- When the Hessian has negative eigenvalues, then steps along the corresponding eigenvectors are gradient ascent steps. To counteract this, it is possible to regularize the Hessian, so that the updates become:

$$\theta \leftarrow \theta - (\mathbf{H} + \alpha \mathbf{I})^{-1} \nabla_\theta J(\theta_0)$$

  As α becomes larger, this turns into first order gradient descent with learning rate $1/\alpha$.

- Newton's method requires, at every iteration, calculating and then inverting the Hessian. If the network has N parameters, then inverting the Hessian is $\mathcal{O}(N^3)$. This renders Newton's method impractical for many types of deep neural networks.

Quasi-Newton methods

To get around the problem of having to compute and invert the Hessian, quasi-Newton methods are often used. Amongst the most well-known is the BFGS (Broyden Fletcher Goldfarb Shanno) update.

- The idea is that instead of computing and inverting the Hessian at each iteration, the inverse Hessian $\mathbf{H}_0^{-1}$ is initialized at some value, and it is recursively updated via:

$$\mathbf{H}_k^{-1} \leftarrow \left(\mathbf{I} - \frac{\mathbf{s}\mathbf{y}^T}{\mathbf{y}^T\mathbf{s}}\right)\mathbf{H}_{k-1}^{-1}\left(\mathbf{I} - \frac{\mathbf{y}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}\right) + \frac{\mathbf{s}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}$$

  for

$$\mathbf{s} = \theta_k - \theta_{k-1} \quad \text{and} \quad \mathbf{y} = \nabla J(\theta_k) - \nabla J(\theta_{k-1})$$

- The proof of this result is beyond the scope of this class; if you'd like to learn more about this (and about optimization in general), consider taking ECE 236C.

Quasi-Newton methods (cont.)

- An important aspect of this update is that the inverse of any Hessian can be reconstructed from the sequence of $\mathbf{s}_k$, $\mathbf{y}_k$, and the initial $\mathbf{H}_0^{-1}$. Thus a recurrence relationship can be written to calculate $\mathbf{H}_k^{-1}\mathbf{x}$ without explicitly having to calculate $\mathbf{H}_k^{-1}$. However, it does require iterating over $i = 0, \ldots, k$ examples.
- A way around this is to use limited memory BFGS (L-BFGS), where you calculate the inverse Hessian using just the last m examples assuming $\mathbf{H}_{k-m}^{-1}$ is some $\mathbf{H}_0^{-1}$ (e.g., it could be the identity matrix).
- Quasi-Newton methods usually require a full batch (or very large minibatches) since errors in estimating the inverse Hessian can result in poor steps.

Conjugate gradient methods

CG methods are also beyond the scope of this class, but we bring it up here in case helpful to look into further. Again, ECE 236C is recommended if you'd like to learn more about these techniques.

- CG methods find search directions that are *conjugate* with respect to the Hessian, i.e., that $\mathbf{g}_k^T \mathbf{H} g_{k-1} = 0$.
- It turns out that these derivatives can be calculated iteratively through a recurrence relation.
- Implementations of "Hessian-free" CG methods have been demonstrated to converge well (e.g., Martens et al., ICML 2011).

Challenges in gradient descent

- **Exploding gradients.**
  Sometimes the cost function can have "cliffs" whereby small changes in the parameters can drastically change the cost function. (This usually happens if parameters are repeatedly multiplied together, as in recurrent neural networks.) Because the gradient at a cliff is large, an update can result in going to a completely different parameter space. This can be ameliorated via gradient clipping, which upper bounds the maximum gradient norm.

$$\|g\| > \text{clip}$$

$$g \leftarrow \frac{g}{\|g\|} \cdot \text{clip} \cdot$$

Challenges in gradient descent

- **Vanishing gradients.**
  Like in exploding gradients, repeated multiplication of a matrix $\mathbf{W}$ can cause vanishing gradients. Say that each time step can be thought of as a layer of a feedforward network where each layer has connectivity $\mathbf{W}$ to the next layer. By layer t, there have been $\mathbf{W}^t$ multiplications. If $\mathbf{W} = \mathbf{U}\Lambda\mathbf{U}^{-1}$ is its eigendecomposition, then $\mathbf{W}^t = \mathbf{U}\Lambda^t\mathbf{U}^{-1}$, and hence the gradient along eigenvector $\mathbf{u}_i$ is shrunk (or grown) by the factor λ_i^t. Architectural decisions, as well as appropriate regularization, can deal with vanishing gradients.

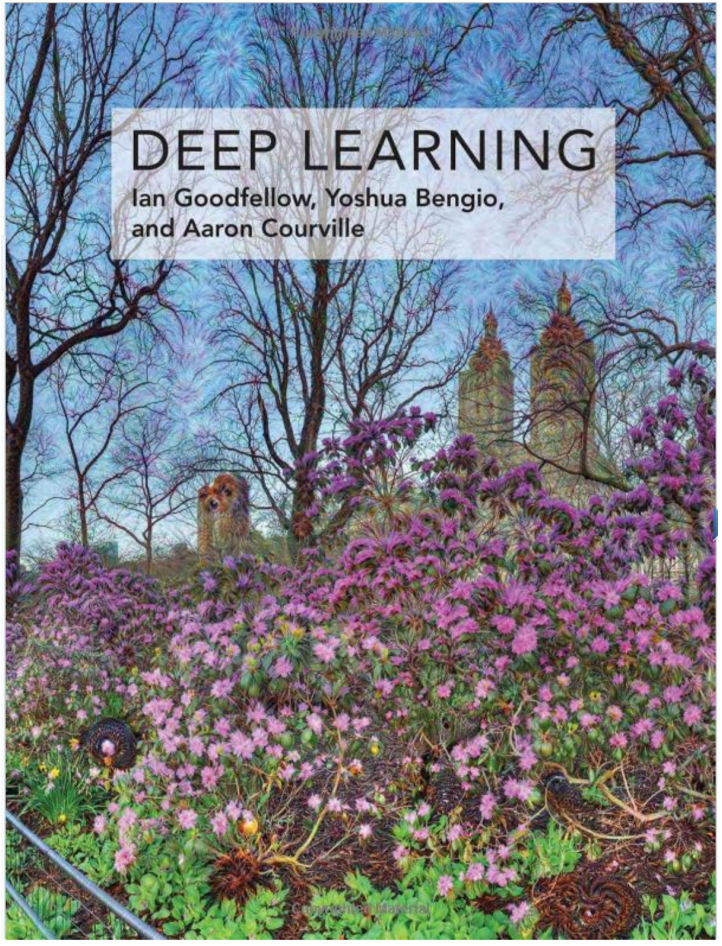
Lecture 8: Convolutional neural networks

In this lecture, we'll talk about the convolutional neural network.

Reading

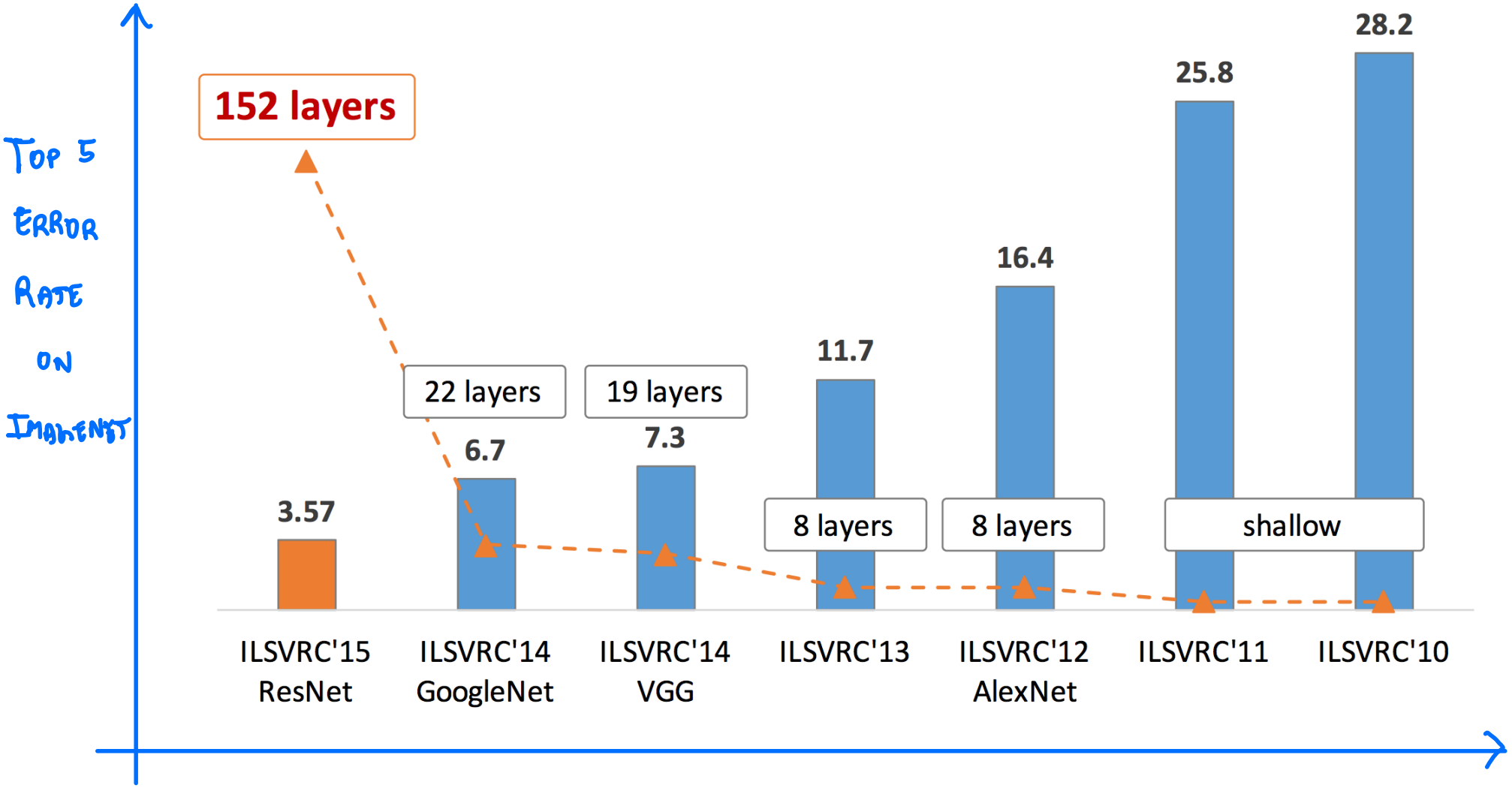
Reading:

Deep Learning, Chapter 9



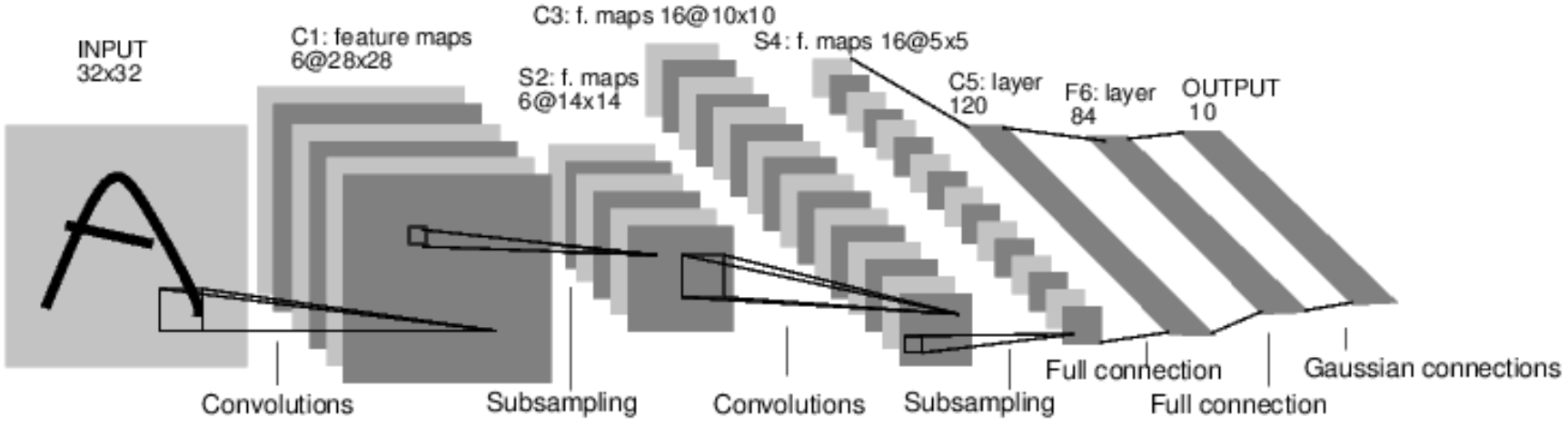
Background: convolutional neural networks

The CNN played a large role in this last "revival" of neural networks (i.e., the past 5 years).



Background: convolutional neural networks

CNNs have been around since the 1990's. In 1998, Yann LeCun introduced LeNet, which is the modern convolutional neural network.



Background: convolutional neural networks

CNNs have been around since the 1990's. In 1998, Yann LeCun introduced LeNet, which is the modern convolutional neural network.