CENG 506 Deep Learning

Lecture 4
Neural Network Training

Slides were prepared using the course material of
Stanford's CNN Course (CS231n by Fei-Fei, Johnson, Yeung) and
Michael Nielsen's online book http://neuralnetworksanddeeplearning.com

Neural Network Training

- The forward flow of information and the backward flow of gradients need to be taken care of.
  - Activation function, Data pre-processing, Batch normalization
- The overfitting problem has to be avoided.
  - Regularization
- The decision strategy and a performance metric has to be selected / devised.
  - Loss function, Optimization algorithm
- Hyper-parameters have to be optimized.
  - Number/type/order of layers, Number of units in each layer
  - Learning rate, Regularization strength, …

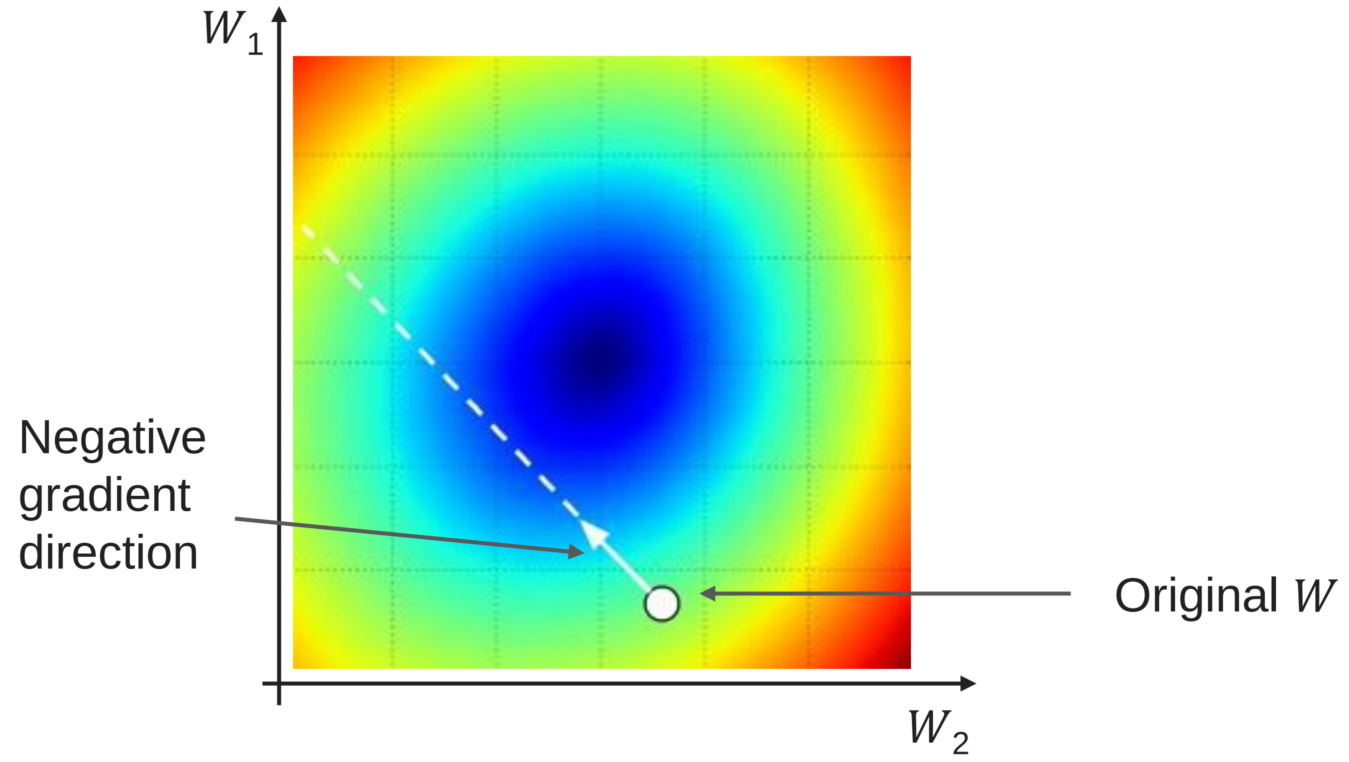
Content

Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

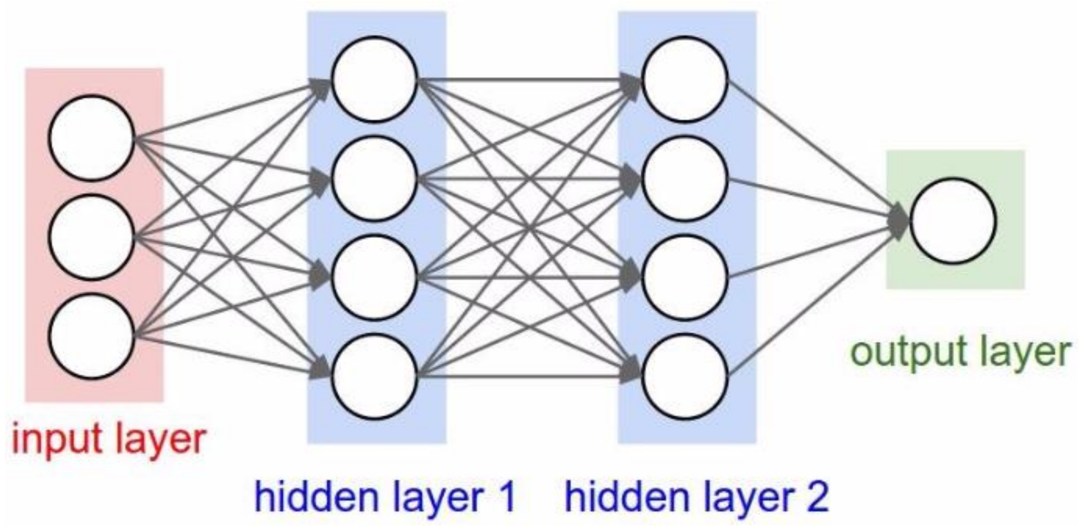
$$w_j := w_j - \alpha \frac{\partial J(w)}{w_j}$$



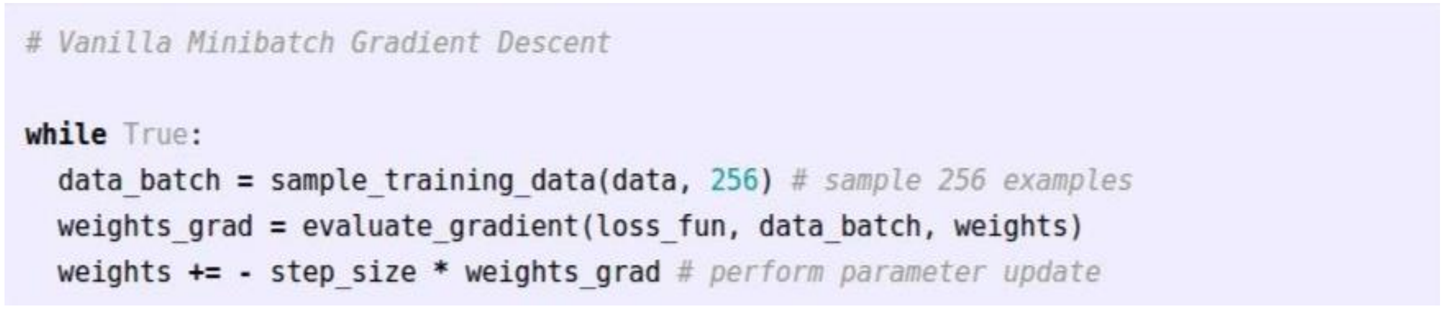
Mini-batch GD Training

Loop:

1. Sample a batch of data
2. Forward propagate it through the network, get loss
3. Backprop to calculate the gradients
4. Update the parameters using the gradient



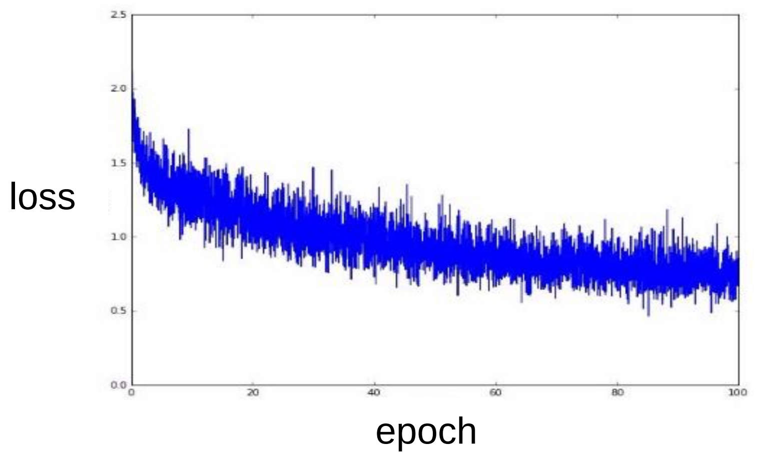
Mini-batch Gradient Descent



```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- Only use a small portion of the training set to compute the gradient.
- Common mini-batch sizes are 4/8/16/32/64/128 examples.
- The ‘vanilla’ version of mini-batch gradient descent is also called Stochastic Gradient Descent (SGD)



Example of optimization progress while training a neural network. (Loss over mini-batches goes down over time.)

Mini-batch Gradient Descent

Smaller mini-batches are less efficient than larger ones in gradient calculation. But they can make up for it with faster model convergence speeds, which may necessitate fewer epochs total.

Small-batch (SB) vs. Large-batch (LB)

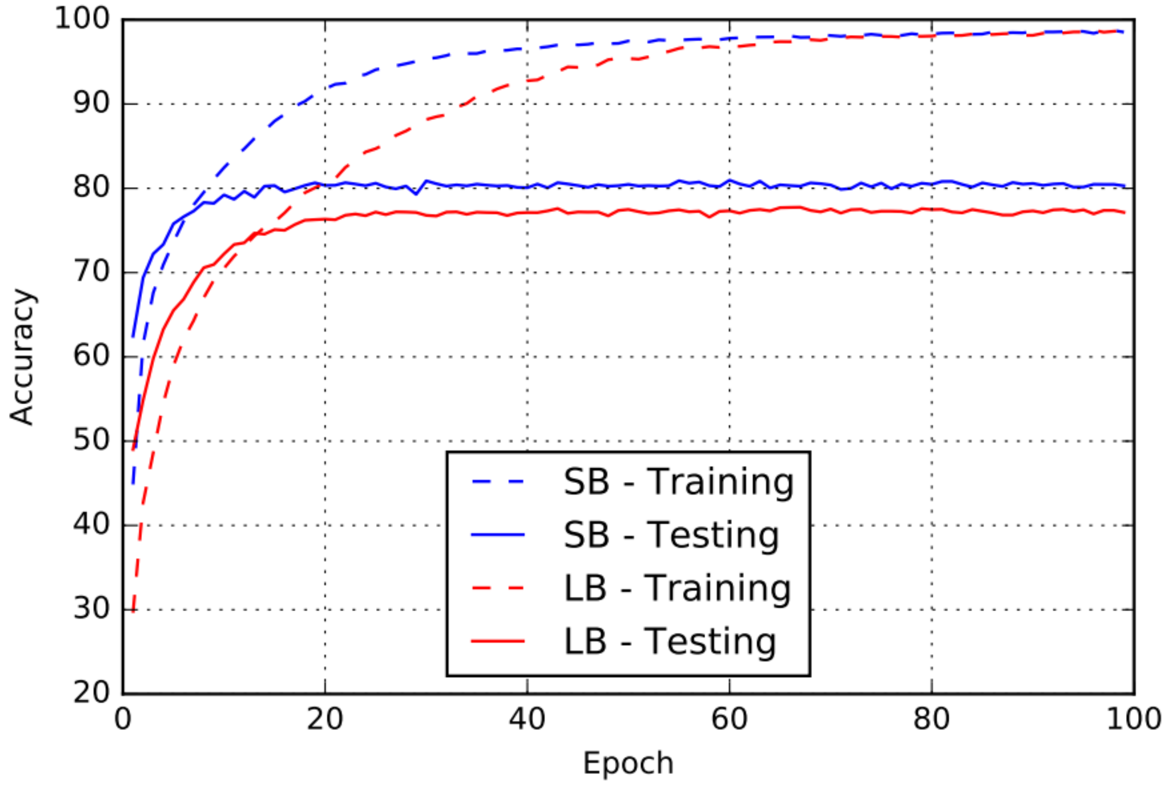
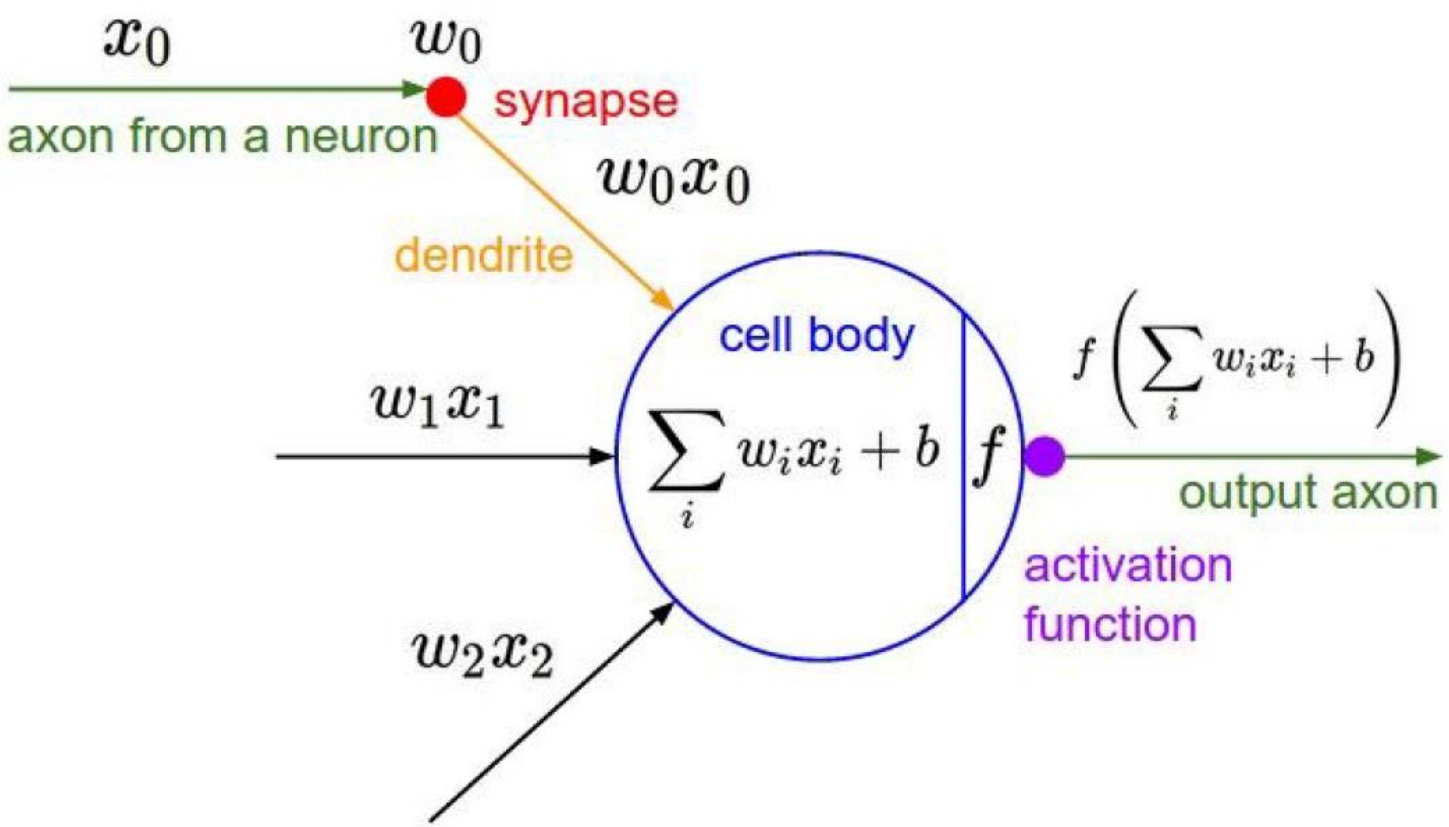
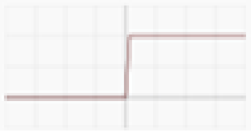
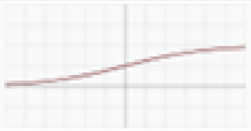
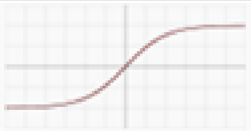
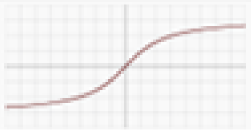
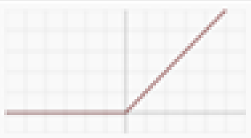
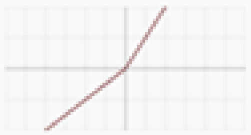
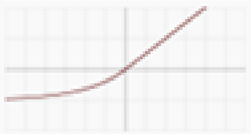


Figure source: https://arxiv.org/pdf/1609.04836.pdf

Activation function



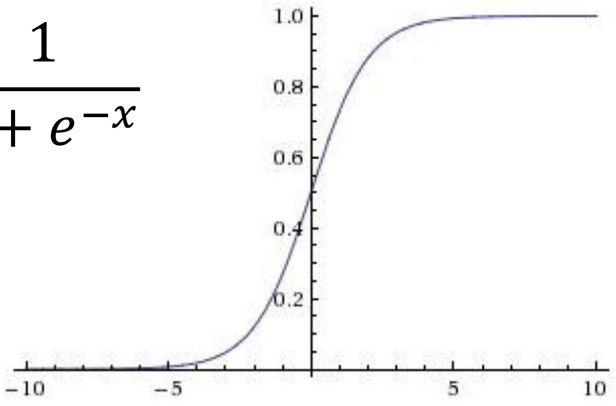
Activation functions

Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

Activation functions

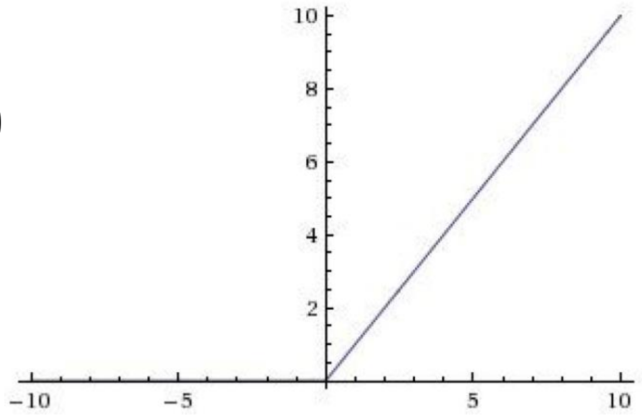
sigmoid

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



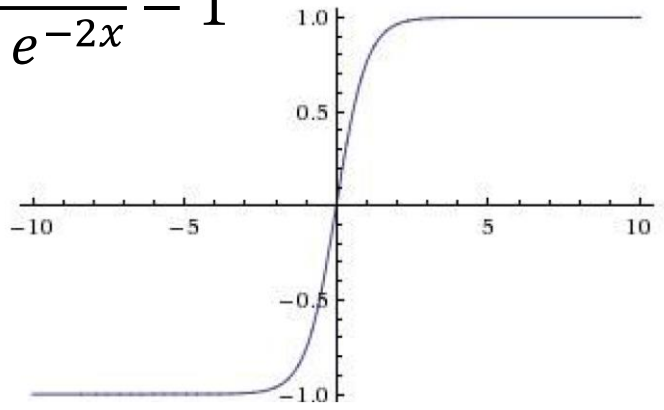
ReLU

$$\max(0, x)$$



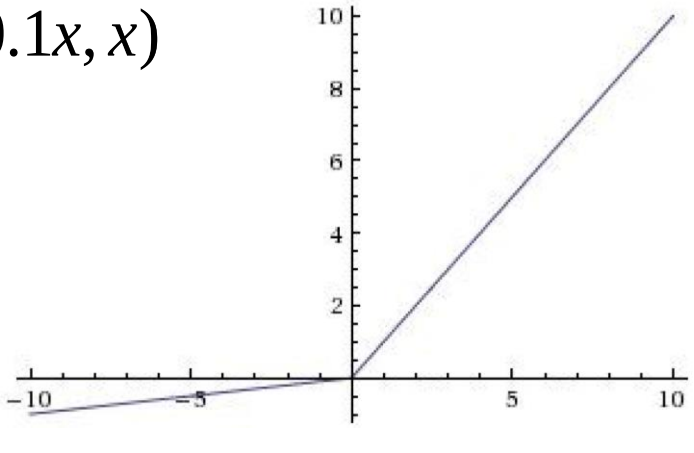
tanh

$$\sigma(x) = \frac{1}{1+e^{-2x}} - 1$$



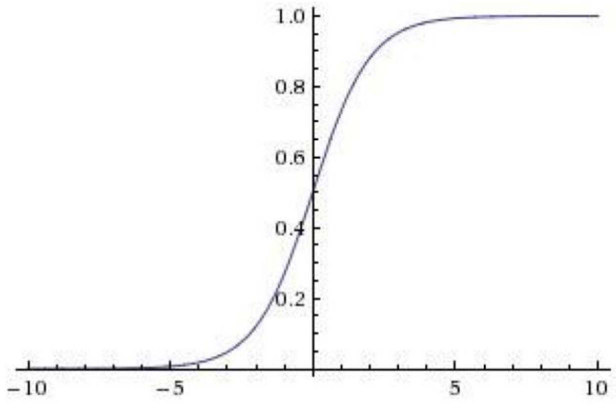
Leaky ReLU

$$\max(0.1x, x)$$



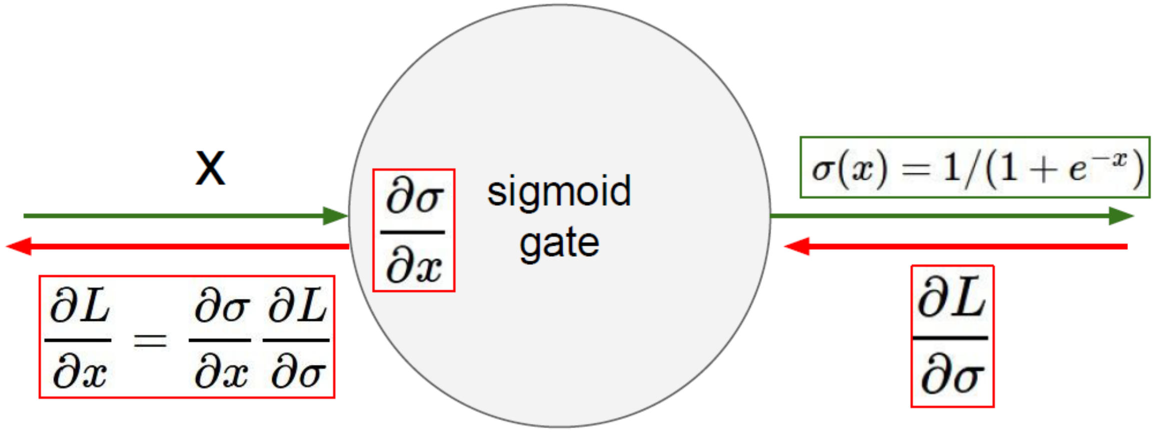
Sigmoid

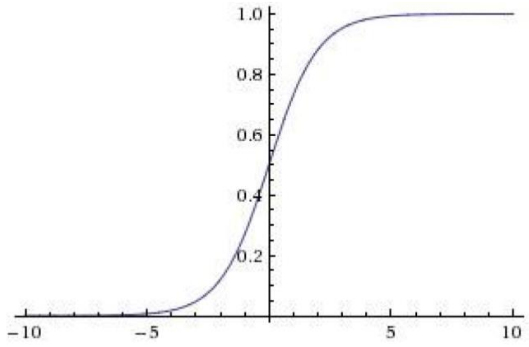
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



- Sigmoid function has seen frequent use since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing (1).
- Sigmoid is not preferred anymore.
  Two main problems:
  1) *Saturated neurons 'kill' the gradients*
  2) *Sigmoid outputs are not zero-centered (always pos.)*

Sigmoid Problem #1


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



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

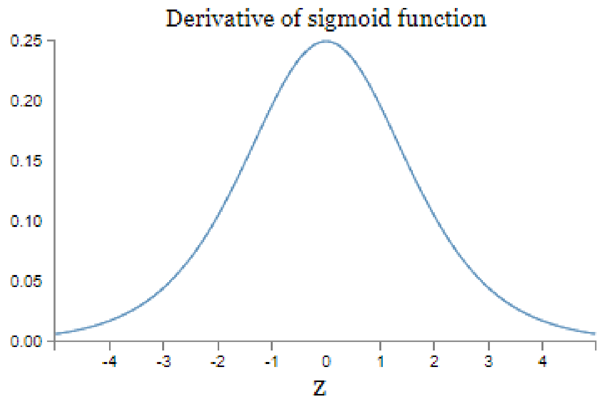
Sigmoids saturate: when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

This (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient.

Sigmoid Problem #1

Sigmoids' saturation causes severe problems

The max. gradient value of sigmoid is 0.25 (when the input is 0.5).
Think of a 4-layer network,
at each layer of backprop
gradient diminish by ¼.



$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\cdot\overbrace{w_2 \cdot \sigma'(z_2)}^{1/4} \cdot \overbrace{w_3 \cdot \sigma'(z_3)}^{1/4} \cdot w_4 \cdot \sigma'(z_4) \cdot \frac{\partial C}{\partial a_4}$$

The phenomenon is called the 'vanishing gradient problem'.

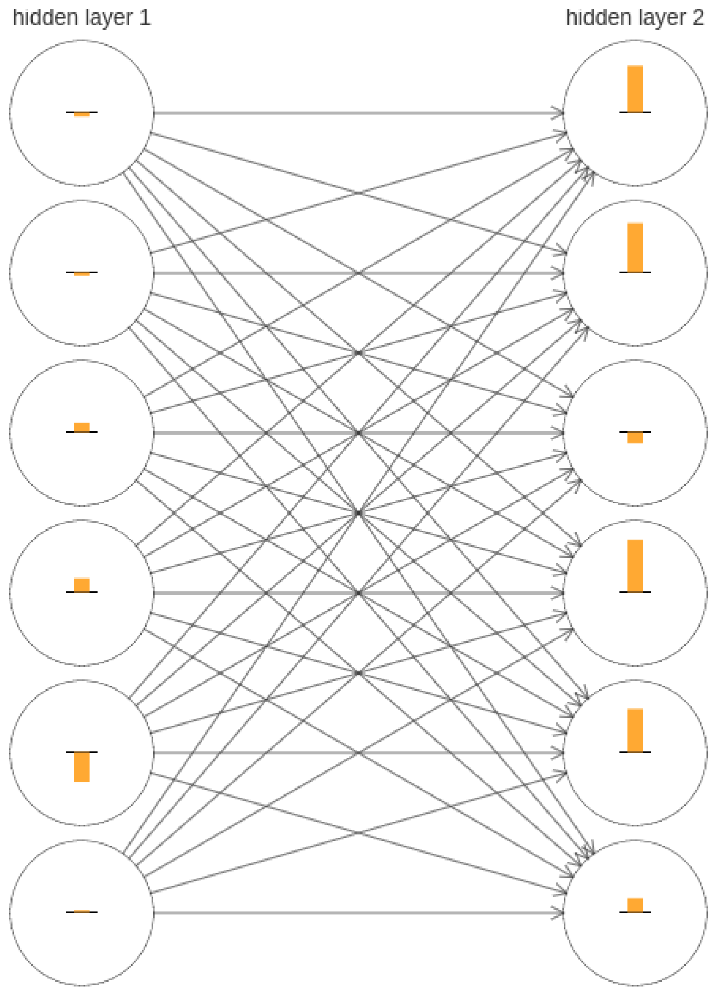
This was one of the reasons why deep networks did not perform better than shallow networks (around 90s, AI winter).

Sigmoid Problem #1

Let’s check M. Nielsen’s analysis*

The bars show the gradients on each neuron's weights:

- There's a lot of variation in how rapidly the neurons learn (big gradient = rapid learning).
- The bars in the second hidden layer are mostly much larger than the bars in the first hidden layer.



* http://neuralnetworksanddeeplearning.com

Sigmoid Problem #1

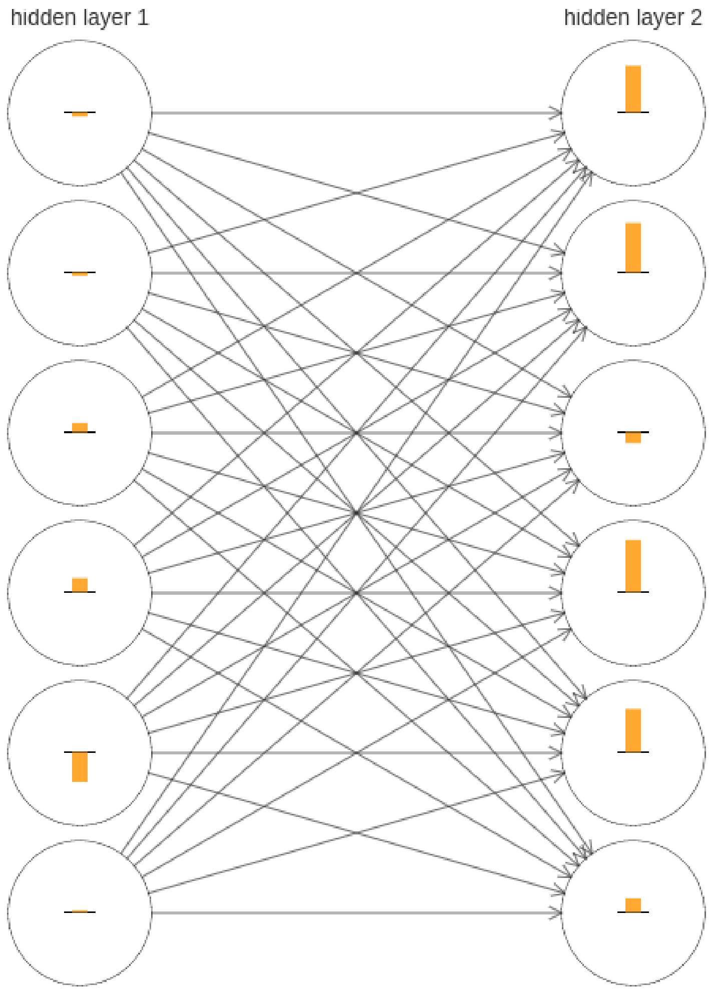
- Think gradient δ^1 as a vector of gradients in the first hidden layer and δ^2 as a vector in the 2nd hidden layer.

- Magnitudes of these vectors are (roughly) the measures of learning speed. The speed of learning at the start of training:

  2-layer: $\|\delta^1\|=0.07$ $\|\delta^2\|=0.31$

  3-layer: $\|\delta^1\|=0.012$ $\|\delta^2\|=0.060$ $\|\delta^3\|=0.283$

  4-layer: $\|\delta^1\|=0.003$ $\|\delta^2\|=0.017$ $\|\delta^3\|=0.070$ $\|\delta^4\|=0.285$



Sigmoid Problem #2

Sigmoid outputs, $a_i^{(j)}$, are not zero-centered, they are always positive.

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

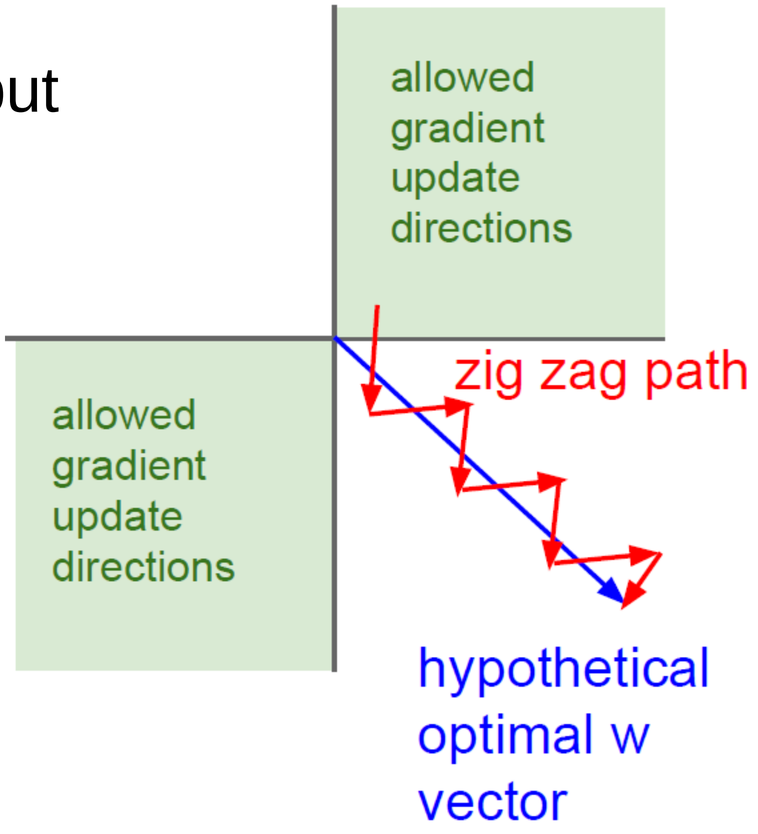
Q. What can we say about dw?

A. All positive or all negative

(All w's increase or all decrease)

$$\frac{\partial L}{\partial w_i^{(L-1)}} = \frac{\partial L}{\partial a_1^{(L)}} \frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} \frac{\partial z_1^{(L)}}{\partial w_i^{(L-1)}}$$

$$= \underbrace{(out - target)}_{\text{same for all weights}} \cdot \underbrace{a_1^{(L)} \cdot (1 - a_1^{(L)})}_{\text{sig. der. always +}} \cdot \underbrace{a_i^{(L-1)}}_{\text{always +}}$$

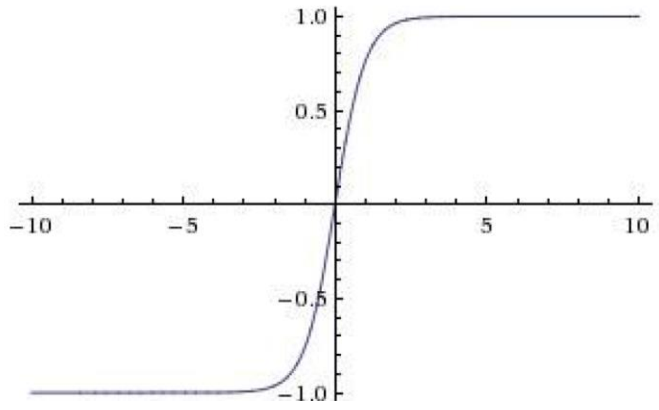

allowed gradient update directions
allowed gradient update directions
zig zag path
hypothetical optimal w vector

This has implications during gradient descent (zig-zag).

Activation functions

tanh

$\tanh(x)$



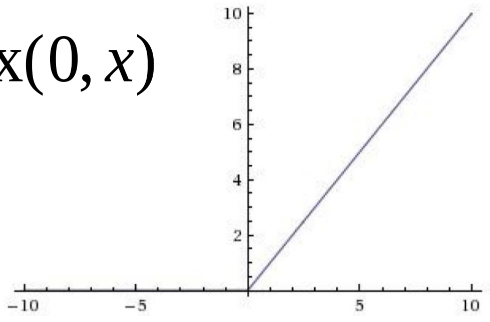
$$f'(x) = 1 - f(x)^2$$

- Squashes numbers to range [-1,1]
- Its output is zero-centered (nice)
- Like the sigmoid neuron, its activations saturate :(

Activation functions

Rectified Linear Unit (ReLU)

$\max(0, x)$

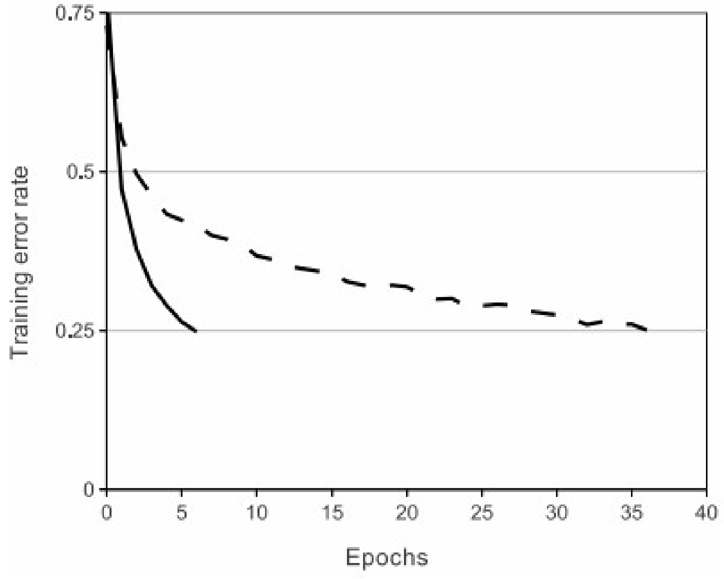


The ReLU has become very popular recently.

(+) Compared to tanh/sigmoid, computation of ReLU is very cheap.

(+) Converges much faster than sigmoid/tanh in practice (see figure)

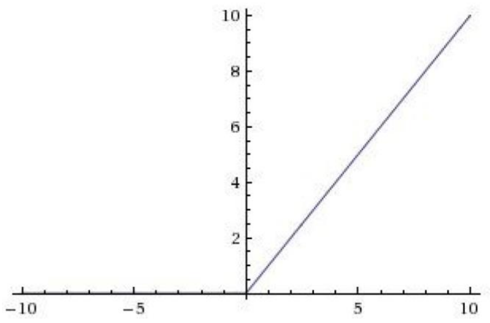
(+) Does not saturate (in +region)



ReLU vs. tanh

ReLU dying problem

Rectified Linear Unit (ReLU)



$$f(x) = \max(0, x)$$

(-) Not zero-centered output

(-) ReLU units can “die”.
A large gradient flowing back could update the weights in a negative region. Then because of zero gradient on the negative side, weights will not be updated again and neuron will never activate again.

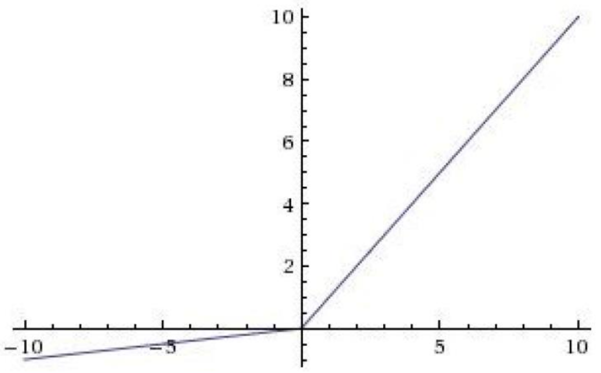
The derivative of ReLU is:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

Activation functions

Leaky ReLU

$$\max(0.1x, x)$$



An attempt to fix the “dying ReLU” problem.

Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of 0.1 or so).

Parametric ReLU

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

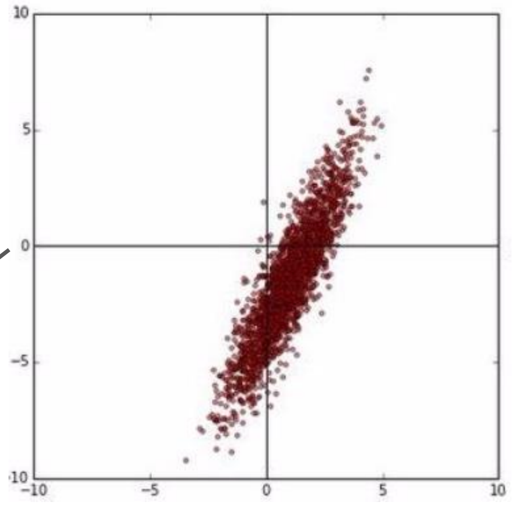
where α is a parameter for the neural network to figure out itself.

Exponential Linear (ELU)

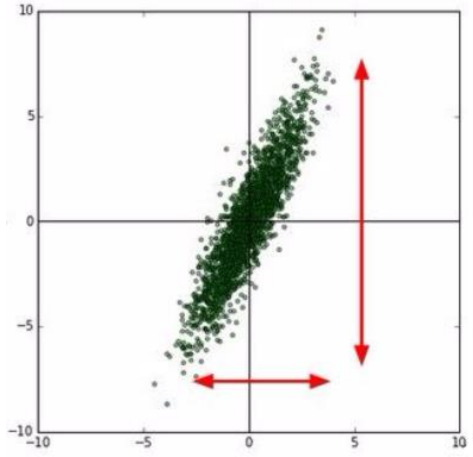
has a curve on the negative side, instead of a straight line.

Data Preprocessing

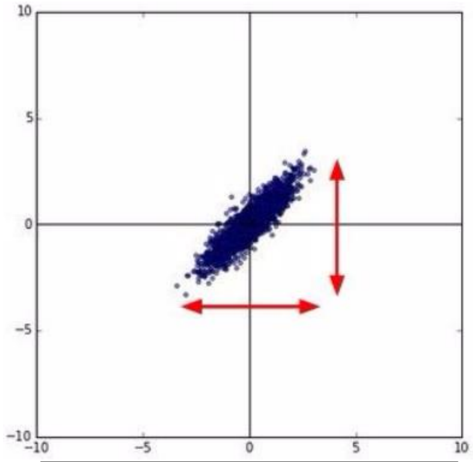
- Zero-centering
- Normalization


original data

zero-centered data



normalized data



```
x-=np.mean(X, axis=0)
```

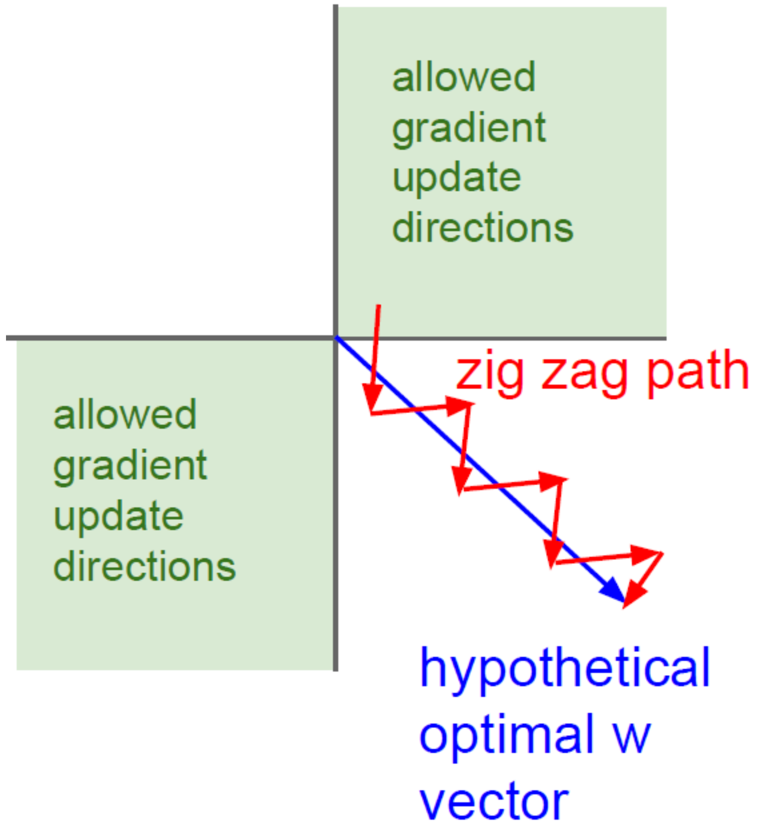
```
x/=np.std(X, axis=0)
```

Data Preprocessing

Zero-centering:
Remember what happens when the input to a neuron is always positive...

This is also why we want zero-mean data!

Normalization:
Provides balance among parameters during optimization.


allowed gradient update directions
zig zag path
allowed gradient update directions
hypothetical optimal w vector

Data Preprocessing

- <u>In practice for images</u>: zero-centering only!
- It can be applied as follows:

  Subtract the mean image (e.g. AlexNet)
  (mean image = [WxHx3] array)

  OR
  Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

- It is not common, for images, to normalize the variance, to do PCA or whitening.

Batch Normalization

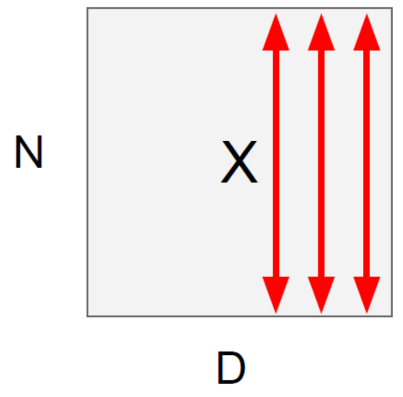
"We want unit Gaussian activations? Just make them so."

Normalization is a simple differentiable operation.

Consider a batch of activations at some layer; to make each dimension/neuron unit gaussian,

a. compute the empirical mean and variance independently for each dimension/neuron;

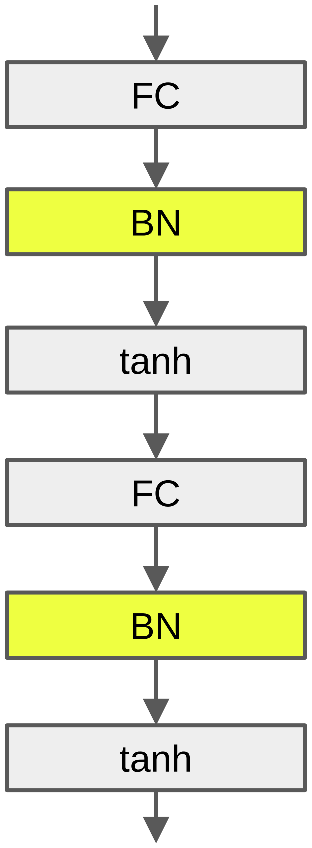
b. apply: $$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

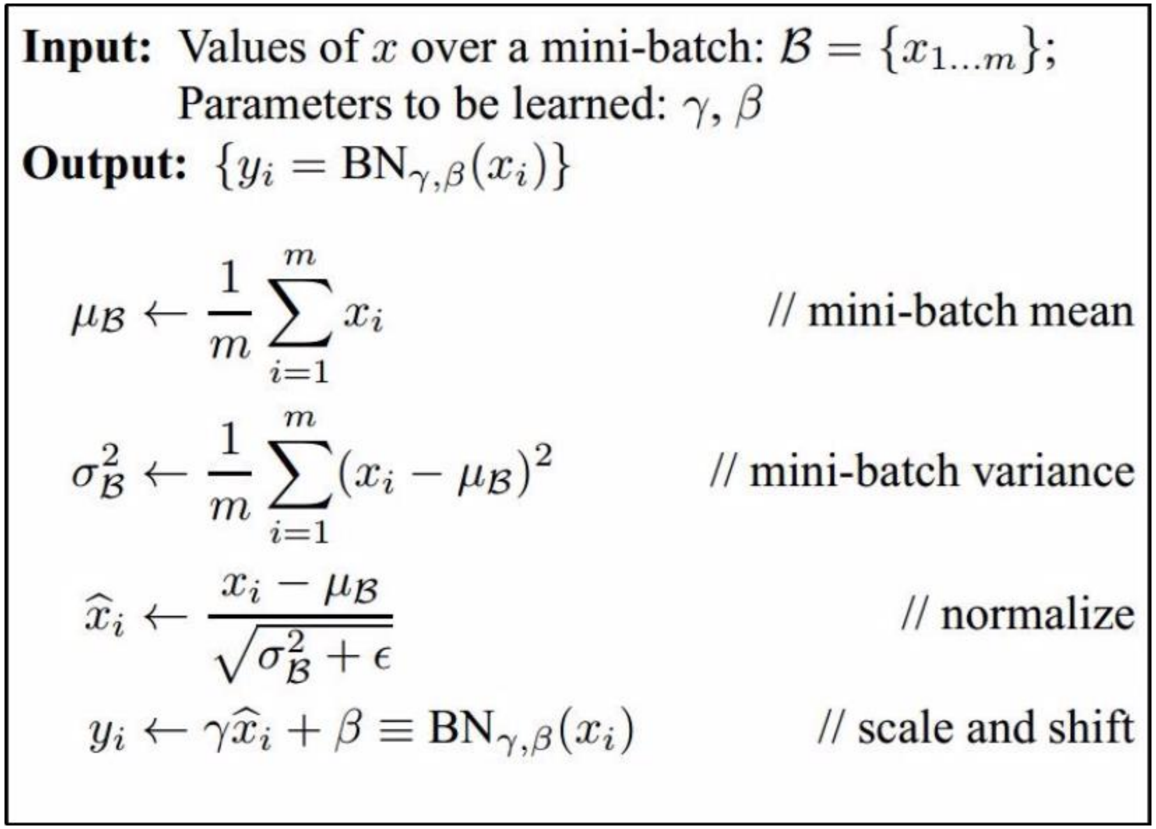


[Ioffe and Szegedy, 2015]

Batch Normalization

This technique is applied as inserting **BN** layers after fully connected (or convolutional) layers, and before nonlinearities.





Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

(+) Improves gradient flow through the network

(+) Allows higher learning rates

(+) Reduces the strong dependence on initialization

(+) Acts as a form of regularization in a funny way

Note: At <u>test time</u> BN layer functions differently:

The mean/std are not computed based on the batch.

Instead, a single fixed empirical mean of activations which was obtained during training is used.

Babysitting the Learning Process

There are multiple useful quantities (loss, accuracy etc) you should monitor during training of a neural network.

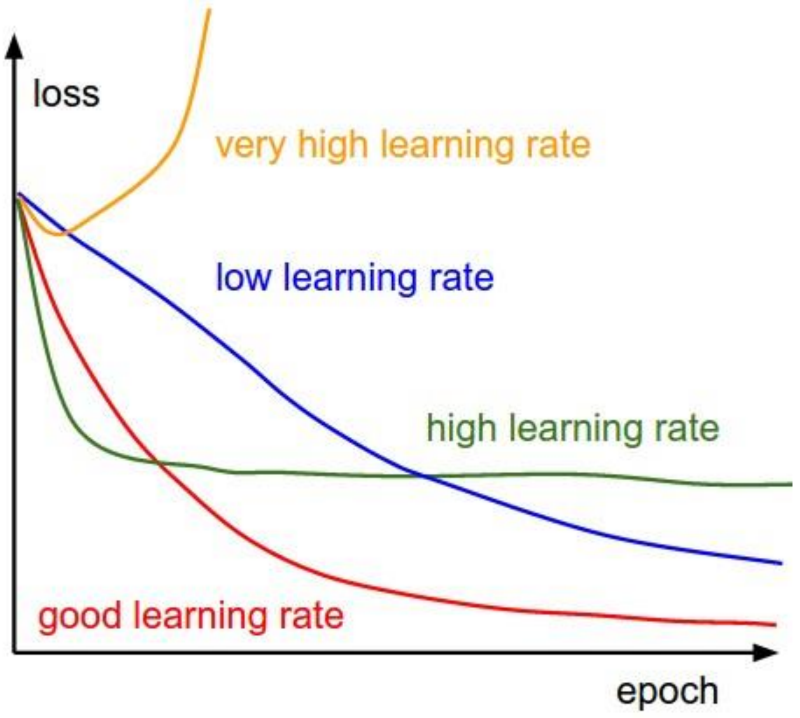
These quantities give intuitions about how hyperparameters should be changed for more efficient learning.

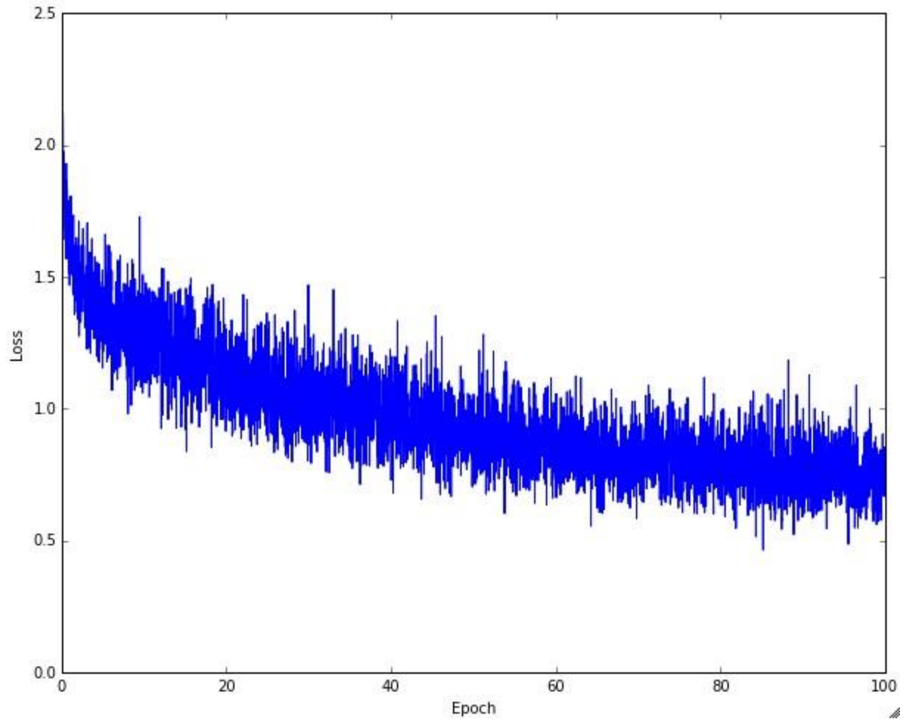
Use plots to visualize these quantities.

Plots are usually in units of epochs, which measure how many times every example has been seen during training in expectation.

Plotting Loss

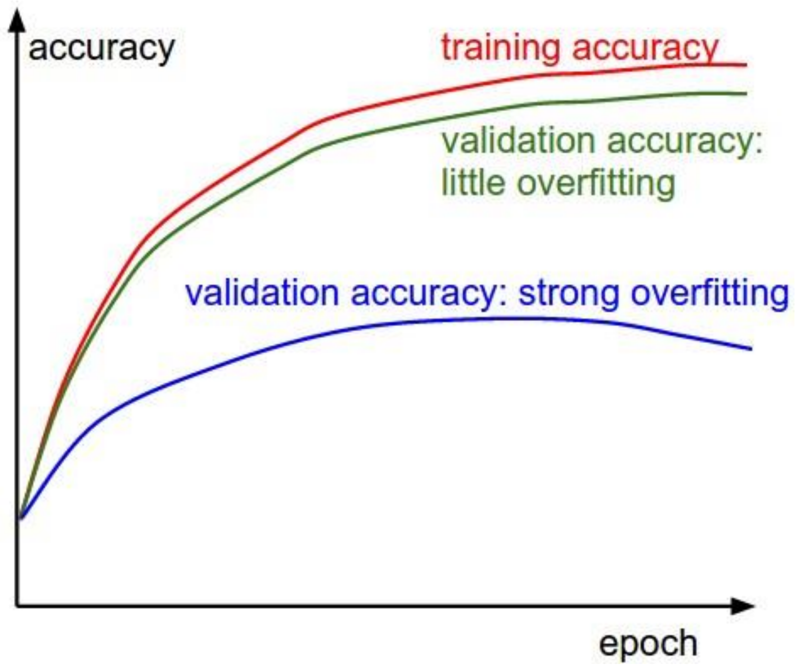
The first quantity that is useful check during training is the loss, as it is evaluated on the forward pass.





Plotting Accuracy

A second important quantity to track while training a classifier is the validation/training accuracy.



- **big gap = overfitting**
  > increase regularization strength?

- **no gap**
  > increase model capacity?

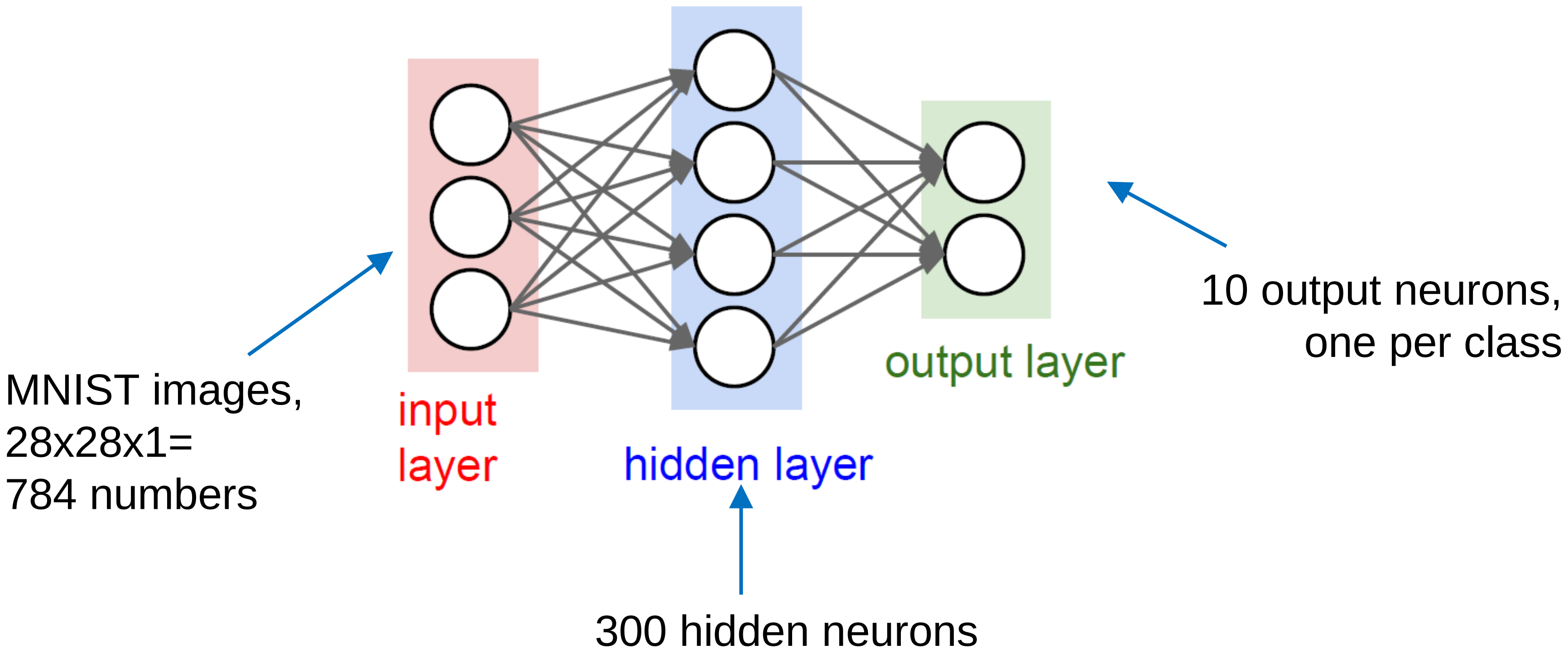
Babysitting the Learning Process

Our example is handwritten digit recognition on MNIST dataset:

Each greyscale image is 28 x 28, representing the digits 0-9. This dataset is large, consisting of 60,000 training images and 10,000 test images.

Babysitting the Learning Process

Lets pick a NN architecture:



Babysitting the Learning Process

Other choices:

- Classification loss: Softmax loss (Neg. log likelihood)

$$L = -\sum_j y_j \log p_j, \qquad p_j = \frac{e^{o_j}}{\sum_k e^{o_k}}$$

where o_j are the output neurons (scores)

- Regularization loss: L2

- Batch size: 64

- Activation function: ReLU at hidden layer,
  not any in output layer.

- Optimizer: Stochastic Gradient Decent (SGD)

Babysitting the Learning Process

Double check the loss at the beginning.

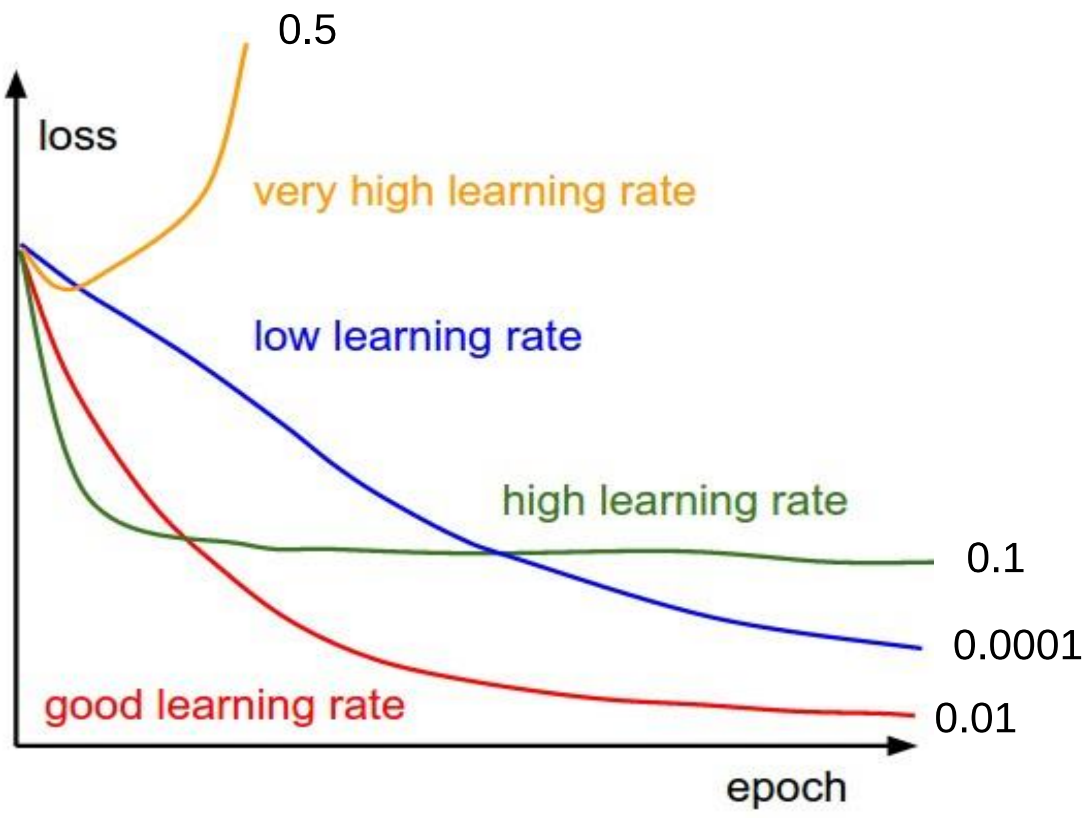
When you initialize with small random weights, check the data loss alone (set regularization strength to zero).

For MNIST, we expect the initial Softmax loss to be 2.302 or a bit more, because we expect a diffuse probability of 0.1 for each class (there are 10 classes), and Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302.

Babysitting the Learning Process

Then, use a small regularization strength and find a good learning rate.

E.g. lr=0.01



Hyperparameter Search

The most common hyperparameters in context of NNs:

- network architecture
- learning rate
- regularization strength (L2 penalty, dropout strength)

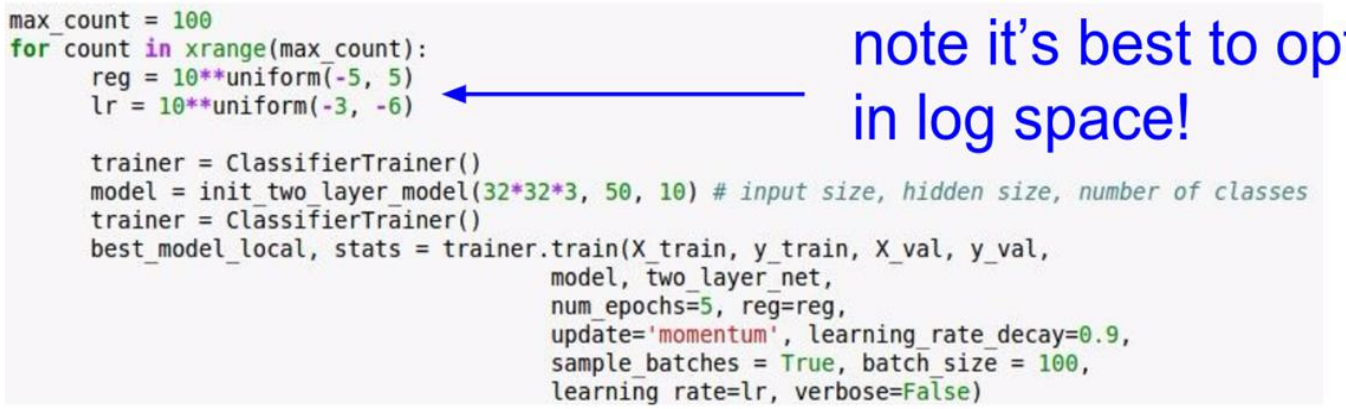
Tips:

- Search for hyperparameters on log scale
- Prefer random search to grid search
- Careful with best values on border
- Coarse-to-fine: Cross validation in stages
  - First stage: a few epochs to get rough idea of what params work
  - Second stage: longer running time, finer search

Hyperparameter Search

Random Hyperparameter Search

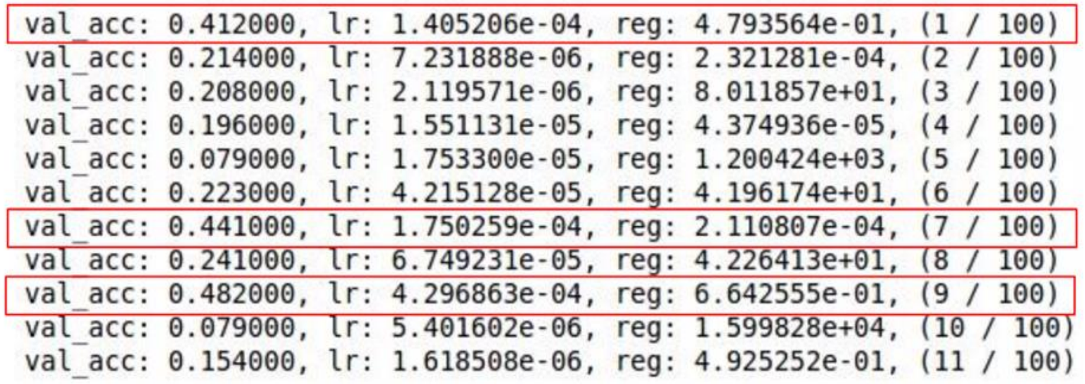
For example: run coarse search for 5 epochs



```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=5, reg=reg,
                                  update='momentum', learning_rate_decay=0.9,
                                  sample_batches = True, batch_size = 100,
                                  learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!



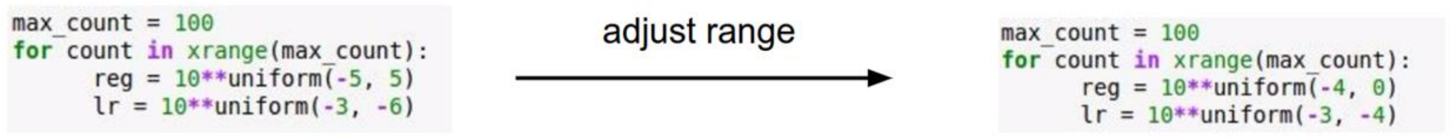
```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Hyperparameter Search

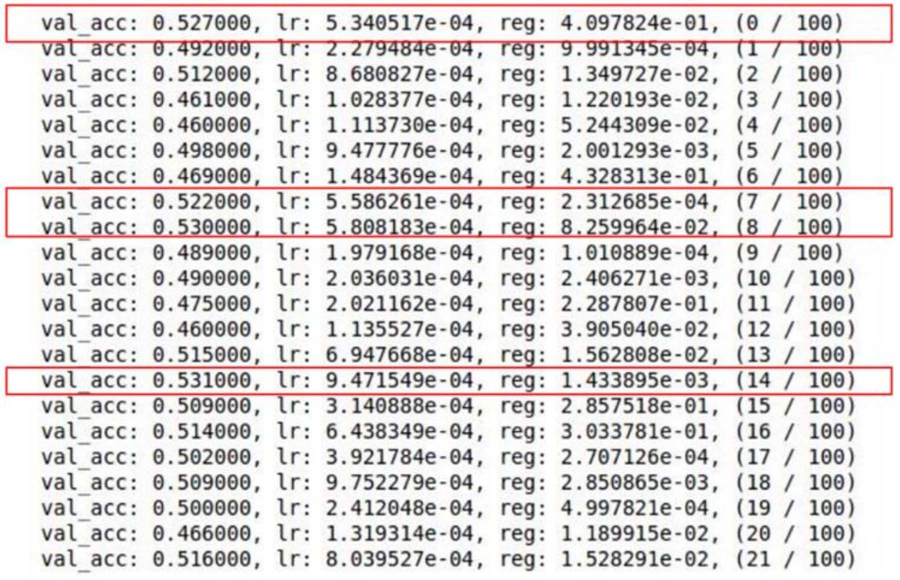
Random Hyperparameter Search

Now run finer search...



```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```



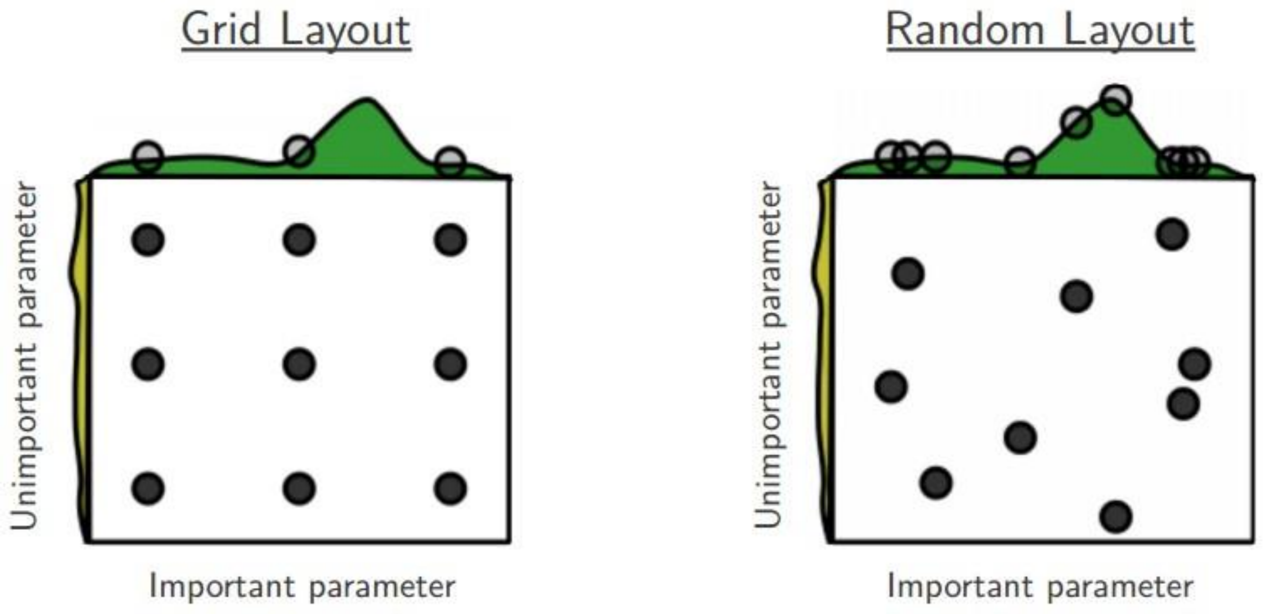
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

But this best cross-validation result is worrying. Why?

Hyperparameter Search

Random Search vs. Grid Search

- As argued by Bergstra and Bengio, “randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid”.
- This is also usually easier to implement.



Random Search for Hyper-Parameter Optimization, Bergstra and Bengio, 2012

Hyperparameter Search

$\rightarrow$ DJ=neural networks practitioner

music = loss function

Parameter Optimization

Once the gradient is computed with backpropagation, the gradients are used to perform a parameter update.

Optimization for deep networks is an active research area.

We’ll highlight some established and common techniques and briefly describe the intuition behind them.

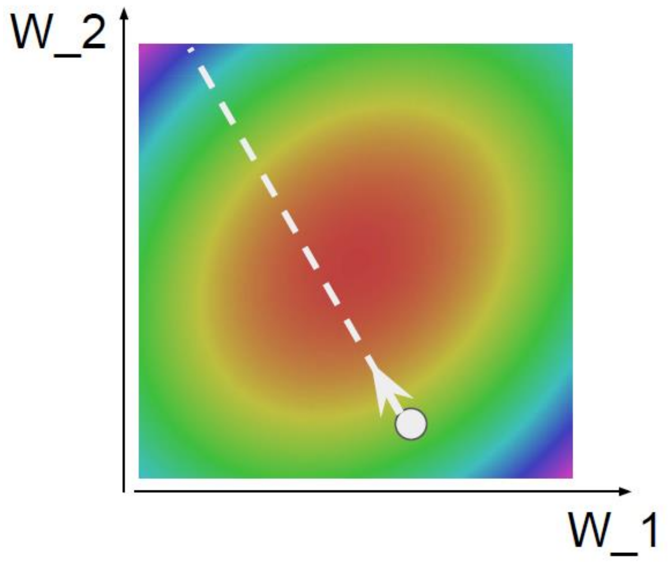
Stochastic Gradient Descent

Vanilla Update: The simplest form of update is to change the parameters along the negative gradient direction.

vector of parameters

gradient

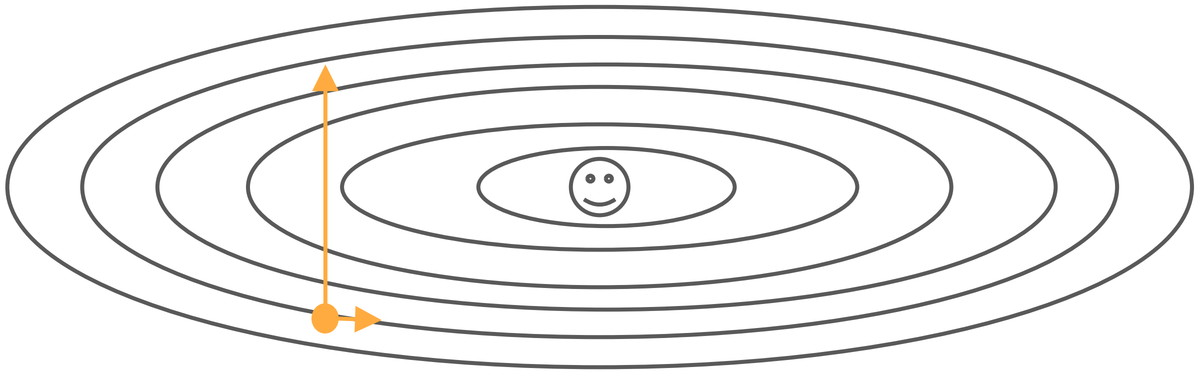
```
w += - learning_rate * dw
```



Stochastic Gradient Descent

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

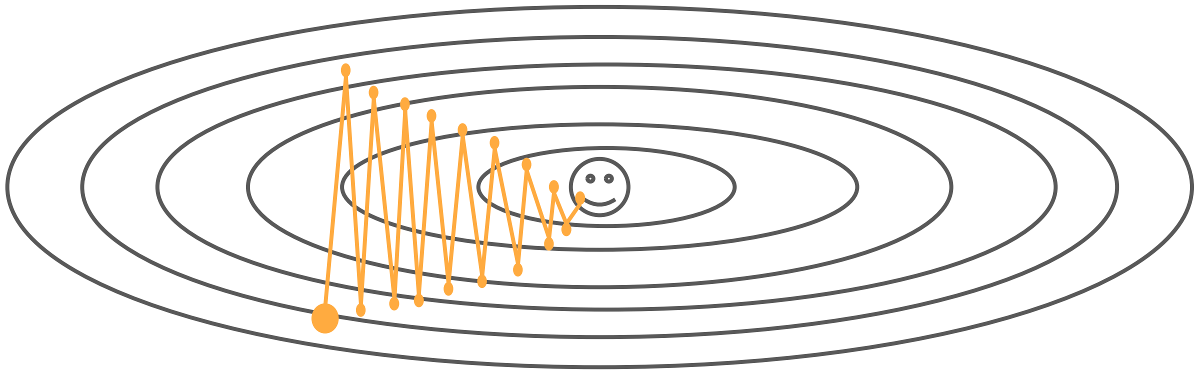


Stochastic Gradient Descent

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

Very slow progress along shallow dimension, zig-zag movement

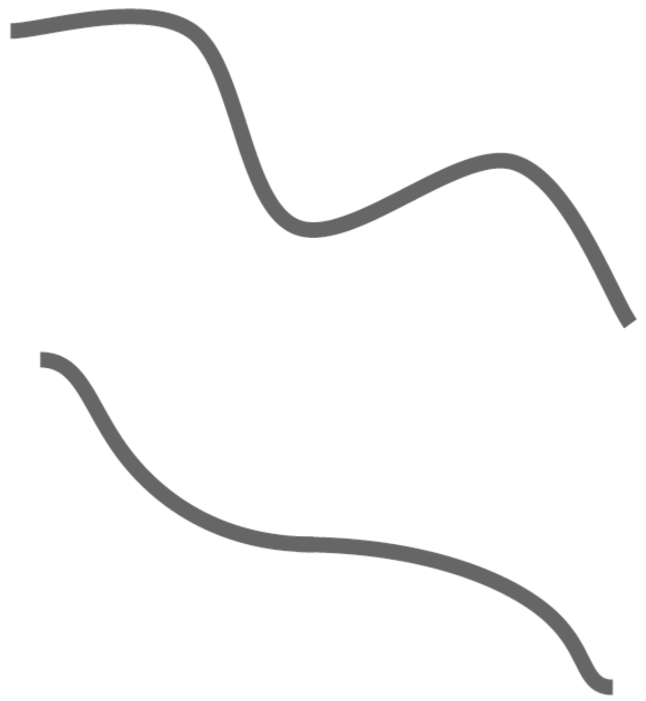


Remember that there are millions of parameters in a large NN.

Stochastic Gradient Descent

Another problem with SGD.

What if the loss function has a **local minima** or **saddle point**?

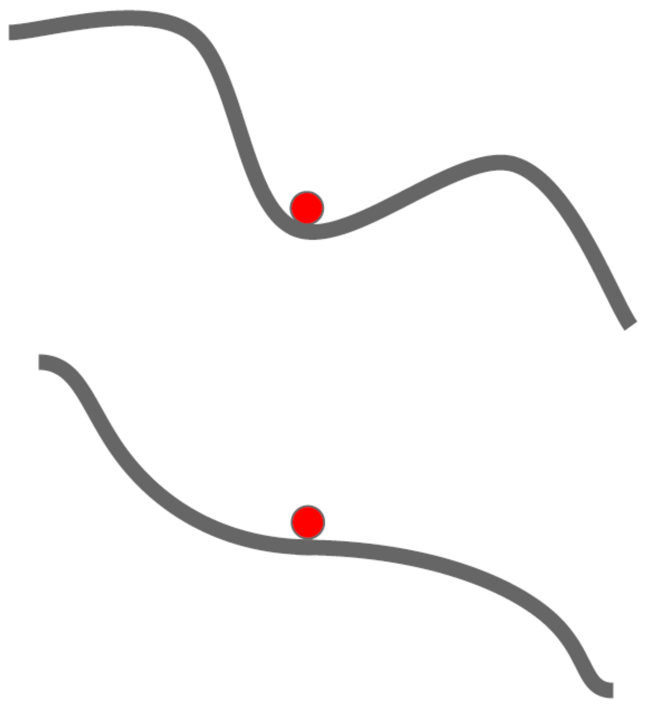


Stochastic Gradient Descent

Another problem with SGD.

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck

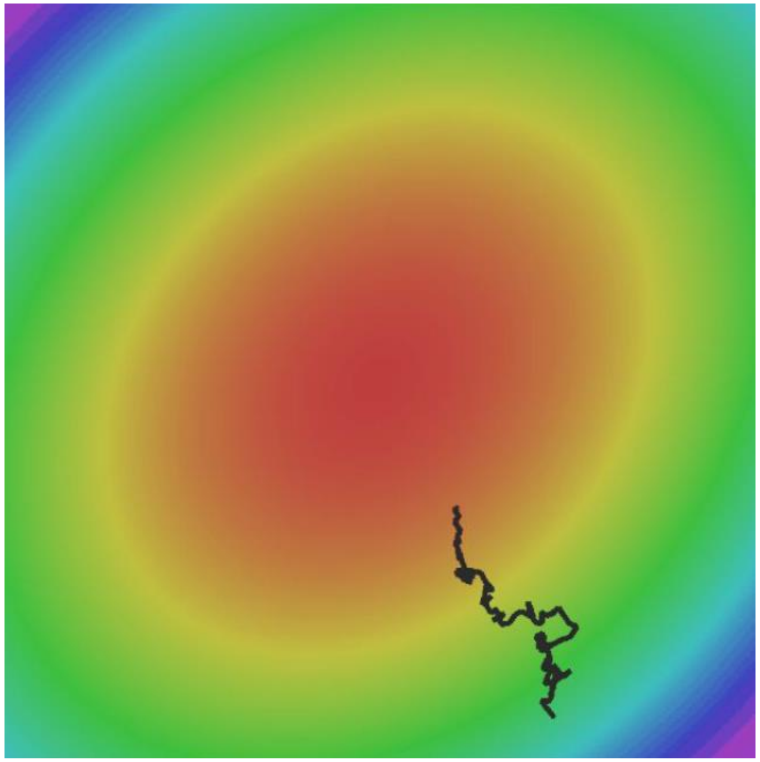


Saddle points much more common in high dimension*

* Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

Stochastic Gradient Descent

Another problem with SGD.



Our gradients come from
mini-batches so they can be noisy!

SGD + Momentum

There is a simple way to address most of these problems.

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

- Build up velocity as a running mean of gradients

- Rho gives momentum (or friction); typically rho=0.9 or 0.99

SGD + Momentum

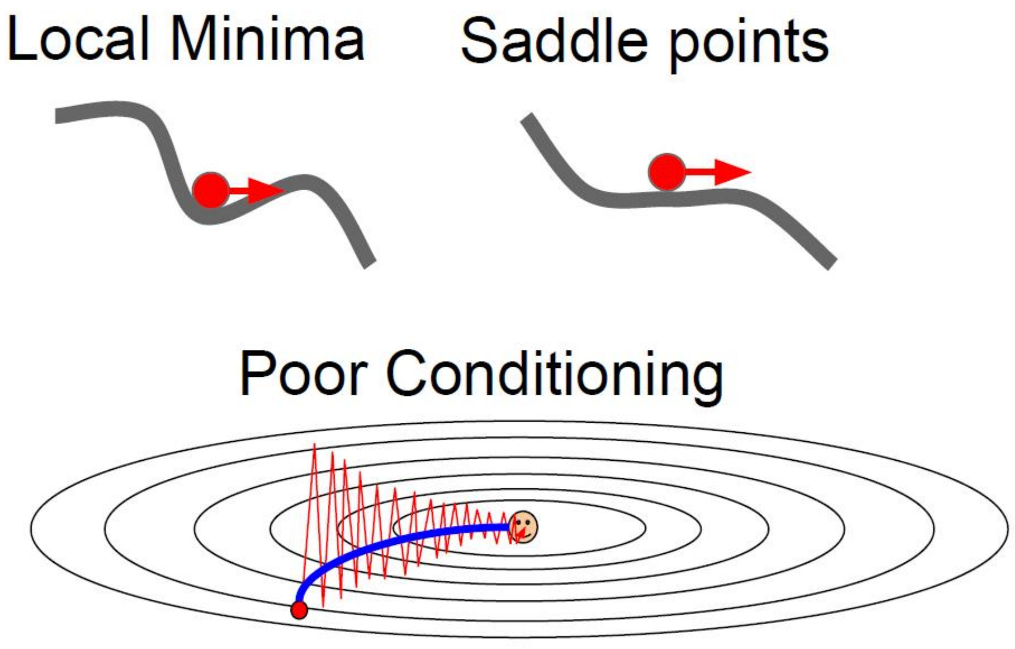
Momentum Update is motivated from a physical perspective.

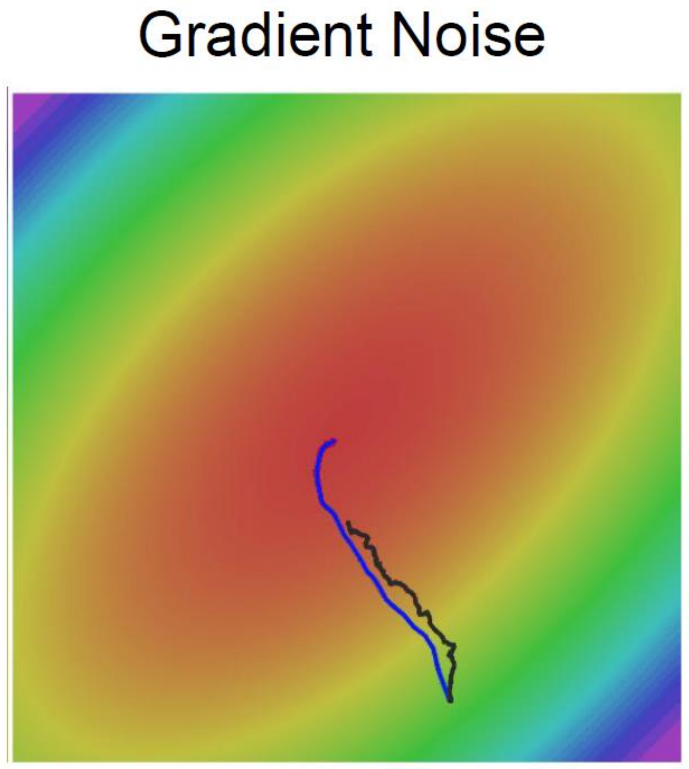
The loss can be interpreted as a the height of a hilly terrain.

The optimization process can be seen as the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.

SGD + Momentum

Momentum Update can help with problems we have seen so far.

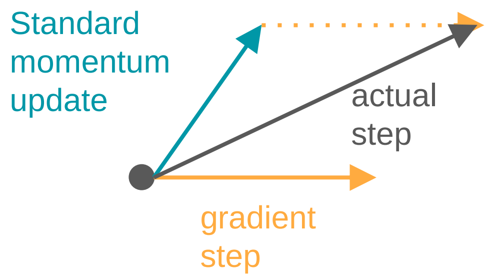


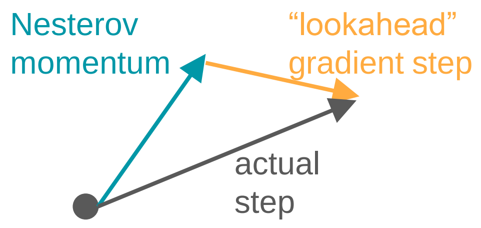


SGD + Momentum

Nesterov Momentum:

- Gradient is computed at the future approximate position `x + rho*v` instead of the current position `x`.
- It has stronger theoretical converge guarantees for convex functions.
- In practice it works slightly better than standard momentum.





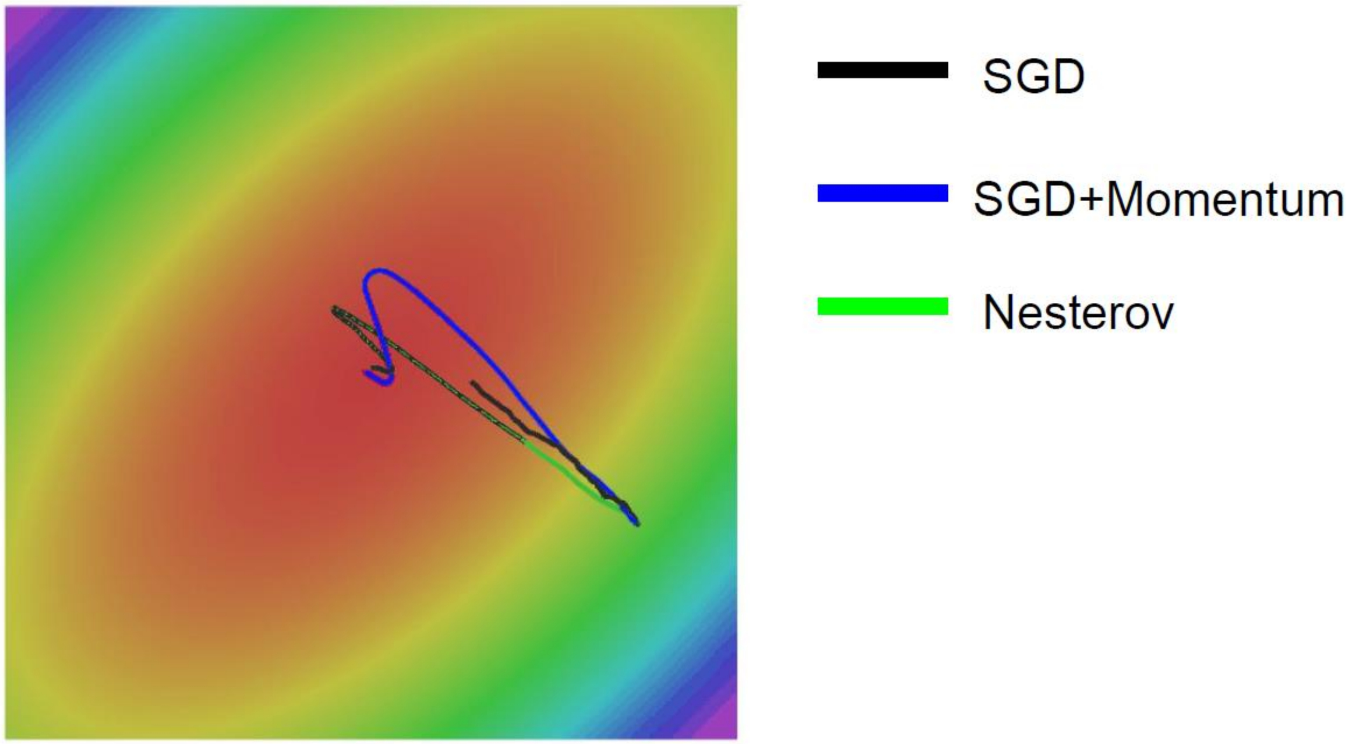
Nesterov momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

```
x_ahead = x + rho * v
# compute dx_ahead (gradient at x_ahead instead of at x)
v = rho * v - learning_rate * dx_ahead
x += v
```

SGD + Momentum

Nesterov Momentum



Parameter Updates

Per-Parameter Adaptive Learning Rate Methods:

<u>Adagrad:</u> An adaptive learning rate method proposed by Duchi et al.

```
dx = compute_gradient(x)
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

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

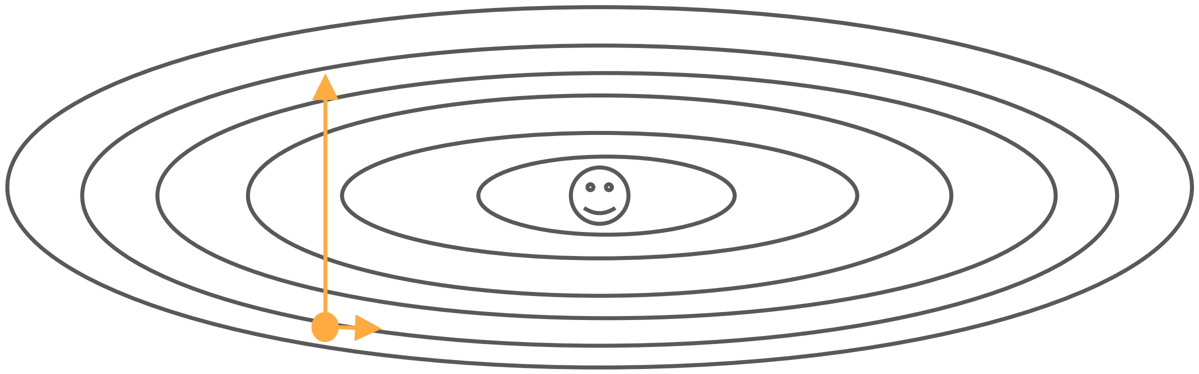
The smoothing term (usually between 1e-4 and 1e-8) which avoids division by zero.

The cache variable has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. It is then used to normalize the parameter update step, element-wise.

AdaGrad

What happens with Adagrad?

What happens to the step size over long time?



The weights that mostly receive high gradients will have their effective learning rate reduced, while weights that mostly receive small updates will have their effective updates increased.

A downside of Adagrad is that the effective learning rate usually gets small quickly and stops learning too early.

Parameter Updates

Per-Parameter Adaptive Learning Rate Methods:

RMSprop: It adjusts the Adagrad in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate.

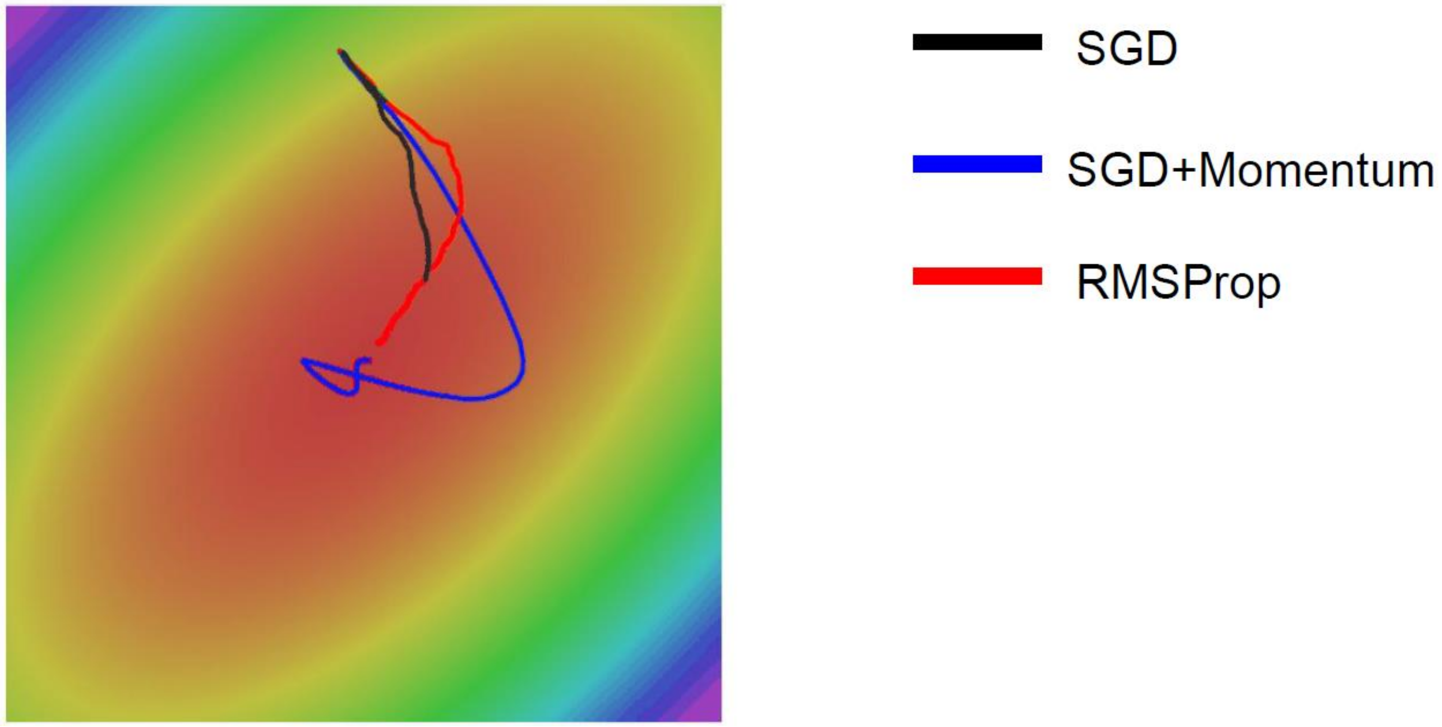
```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

decay_rate is a hyperparameter and typical values are [0.9, 0.99, 0.999].

The `x+=` update is identical to Adagrad, but the cache variable is a “leaky” one.

Parameter Updates

RMSProp vs. others



Parameter Updates

Per-Parameter Adaptive Learning Rate Methods:

Adam: A relatively recently proposed* update that looks a bit like RMSProp with momentum.

The simplified update looks as follows:

```
dx = compute_gradient(x)
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

Momentum

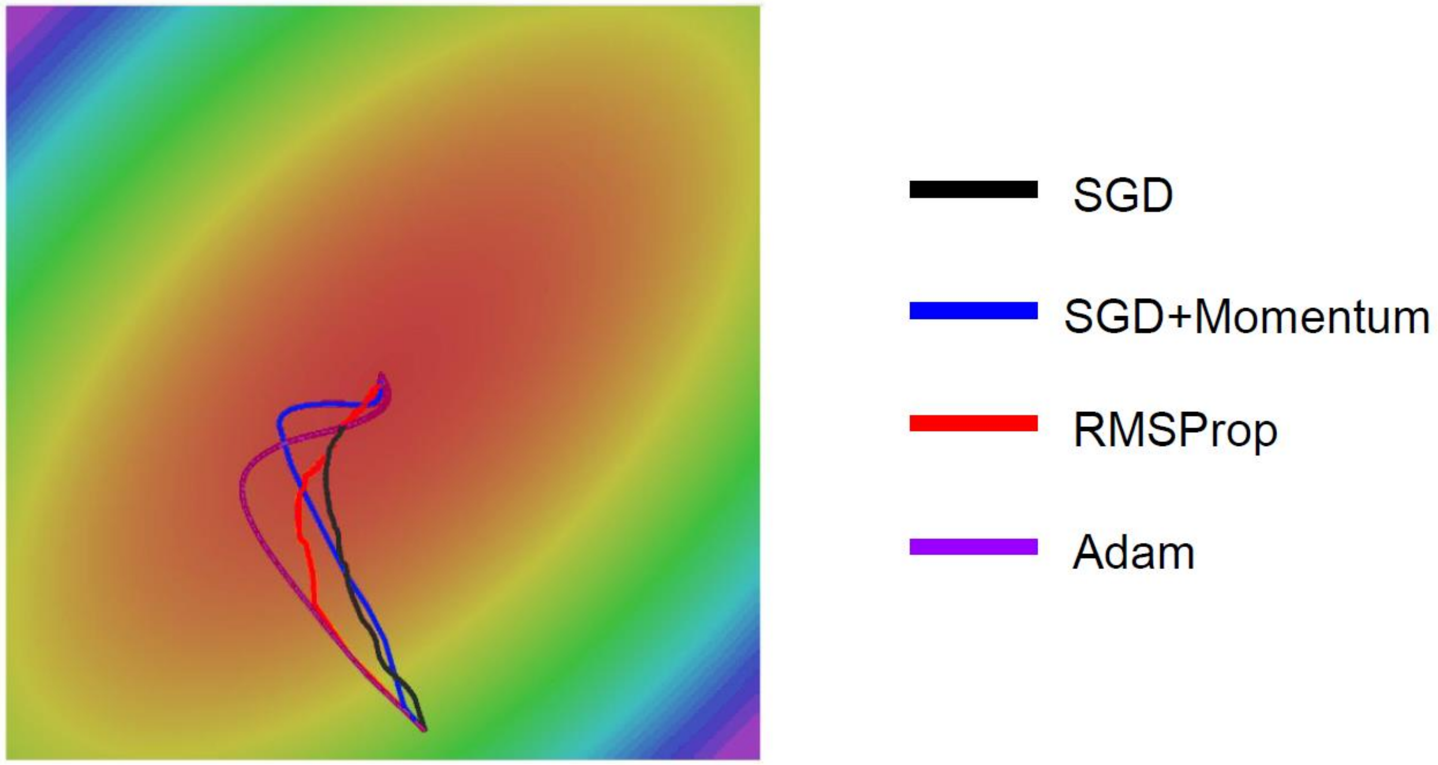
~RMSprop

The update looks exactly as RMSProp update, except the "smooth" version of the gradient `m` is used instead of the raw (and perhaps noisy) gradient vector `dx`.

* Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

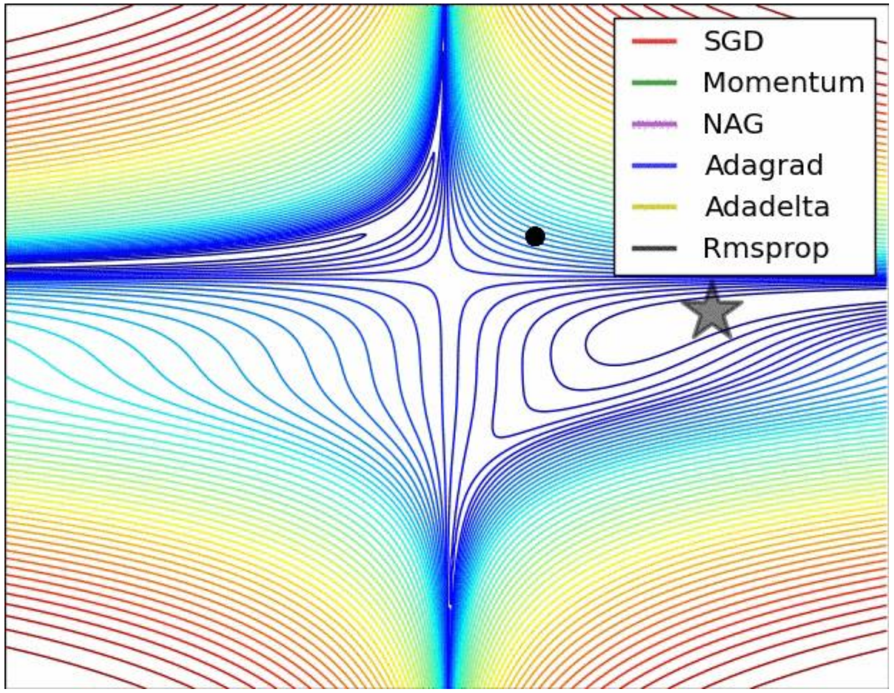
Parameter Updates

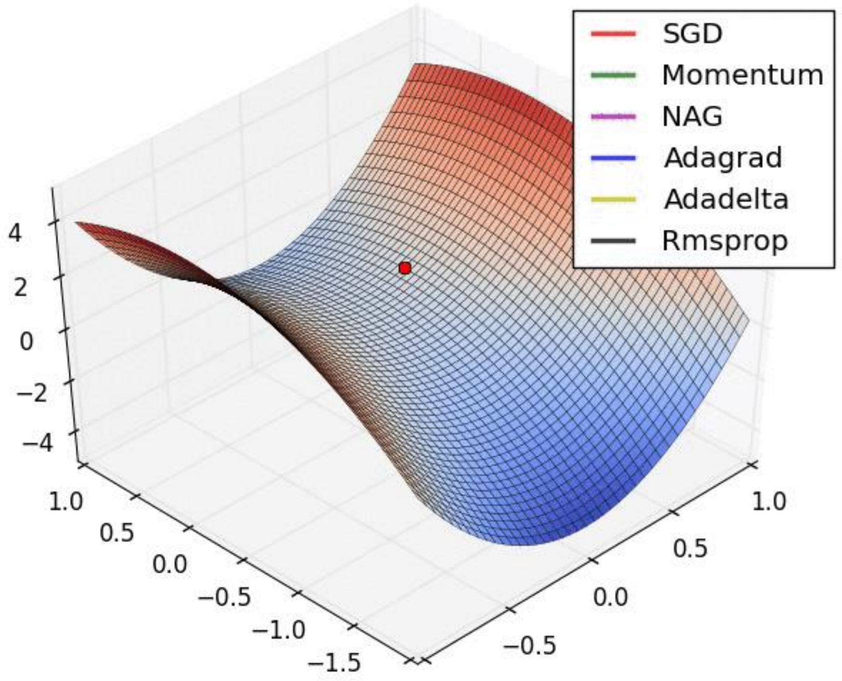
All together



You can see the animated versions of the figures at Stanford’s lecture video:
https://www.youtube.com/watch?v=_JB0AO7QxSA&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=7

Parameter Updates





Left: Contours of a loss surface and time evolution of different optimization algorithms. Notice the "overshooting" behavior of momentum-based methods, which make the optimization look like a ball rolling down the hill.
Right: A visualization of a saddle point in the optimization landscape, where the curvature along different dimension has different signs (one dimension curves up and another down). Notice that SGD has a very hard time breaking symmetry and gets stuck on the top. Conversely, algorithms such as RMSprop will see very low gradients in the saddle direction. Due to the denominator term in the RMSprop update, this will increase the effective learning rate along this direction, helping RMSProp proceed. Images credit: Alec Radford.

Source: http://cs231n.github.io/neural-networks-3/#update

Parameter Updates

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

=> Learning rate decay over time!

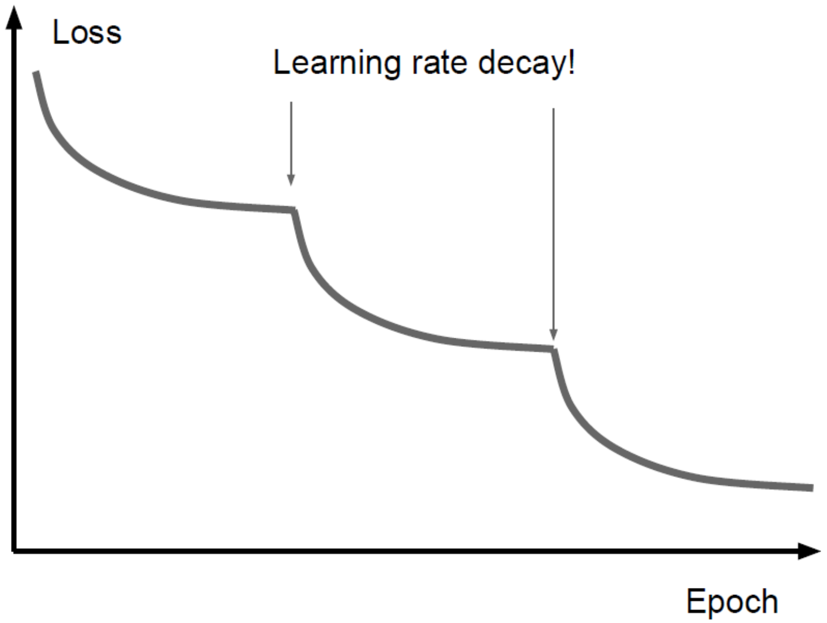
step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

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

1/t decay:

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



Using decay is more critical with SGD+Momentum, less common with Adagrad, Adam etc.

Regularization – Add term to loss

$$L = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1)}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

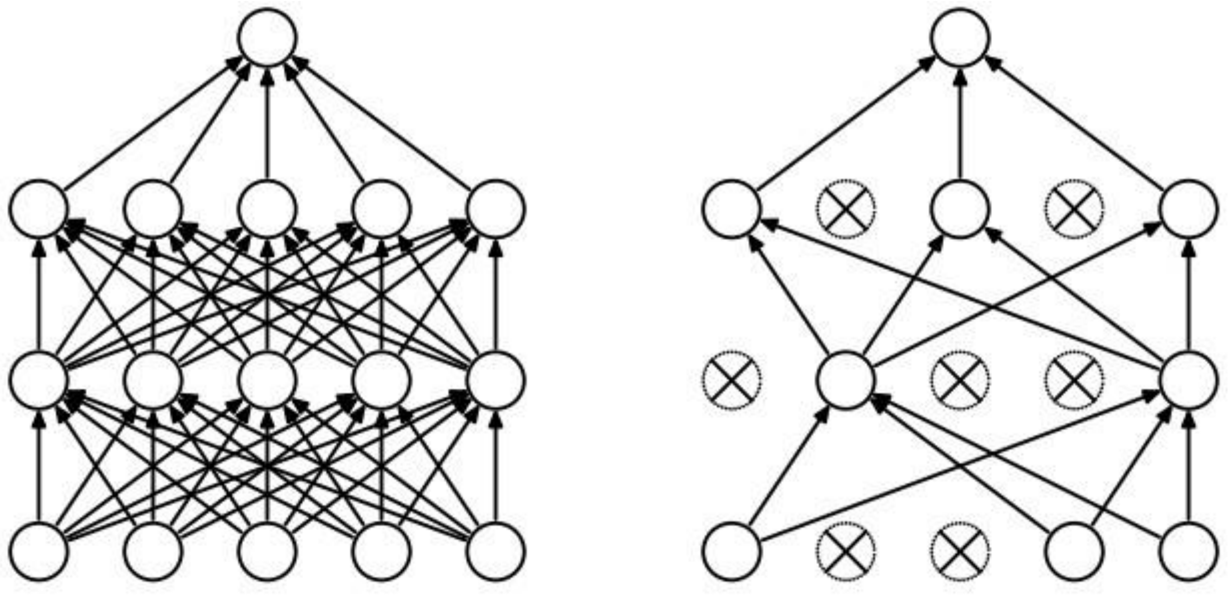
- L2 regularization $\longrightarrow$ $R(W) = \sum_k \sum_l W_{k,l}^2$
- L1 regularization $\longrightarrow$ $R(W) = \sum_k \sum_l |W_{k,l}|$
- Elastic net (L1 + L2) $\longrightarrow$ $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization - Dropout

In each forward pass, randomly set some neurons to zero.

Probability of dropping is a hyperparameter; 0.5 is common.

Dropout can be interpreted as sampling a Neural Network within the full network, and only updating the parameters of the sampled neurons.

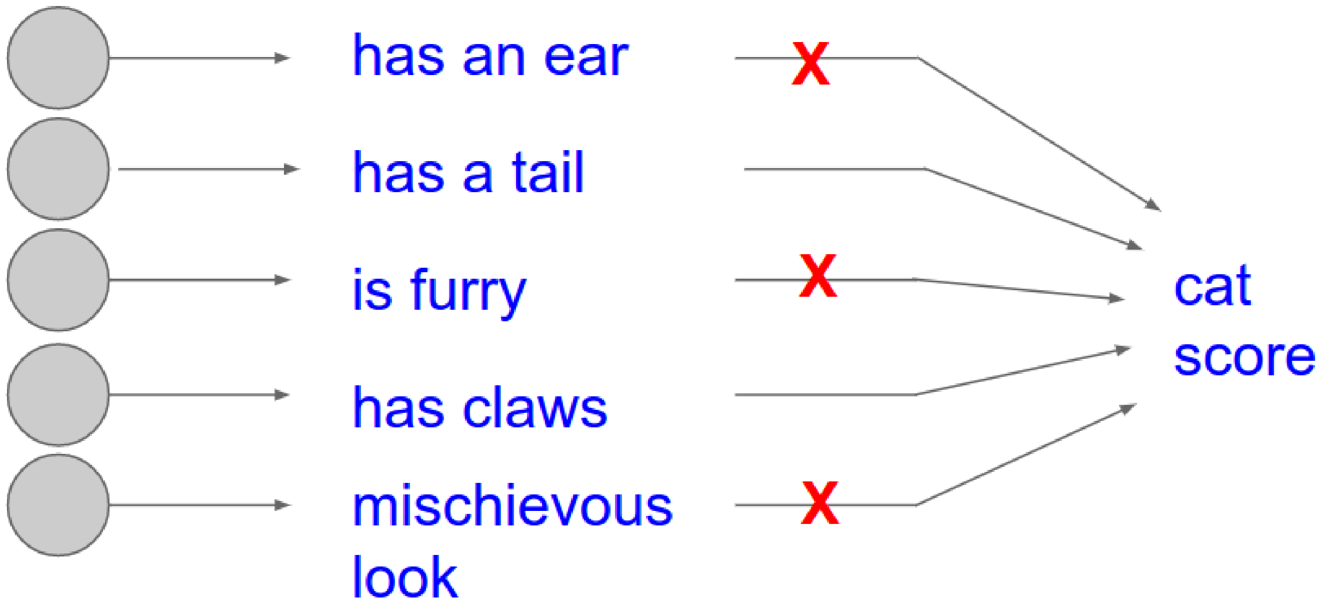


(a) Standard Neural Net (b) After applying dropout.

Regularization - Dropout

Q. How could this possibly be a good idea?

A. It forces the network to have a redundant representation.



Summary

As a **sanity check**, make sure your initial loss is reasonable and you can achieve 100% training accuracy on a very small portion of the data.

During training, **monitor** the **loss**, the **training/validation accuracy**, and when dealing with ConvNets, visualize the first-layer weights.

Search for good hyperparameters with **random search** (not grid search). Stage your search from coarse (wide ranges, training for 1-5 epochs) to fine (narrower ranges, training for many more epochs).

Use either **SGD+Nesterov Momentum** or a fancier optimization like **Adam** or **RMSProp**.

If using SGD, decay your **learning rate**. For example, halve the learning rate after a fixed number of epochs, or whenever the validation accuracy tops off.

It is also common to apply **dropout** in addition to L2 regularization.