CENG 506 Deep Learning

Lecture 4
Neural Network Training - Part 2

Slides were prepared using the course material of
Stanford's CNN Course (CS231n, Fei-Fei, Johnson, Yeung)

Today

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
- Hyperparameter Optimization
- Regularization
- Fancier Optimization

Babysitting the Learning Process

There are multiple useful quantities you should monitor during training of a neural network.

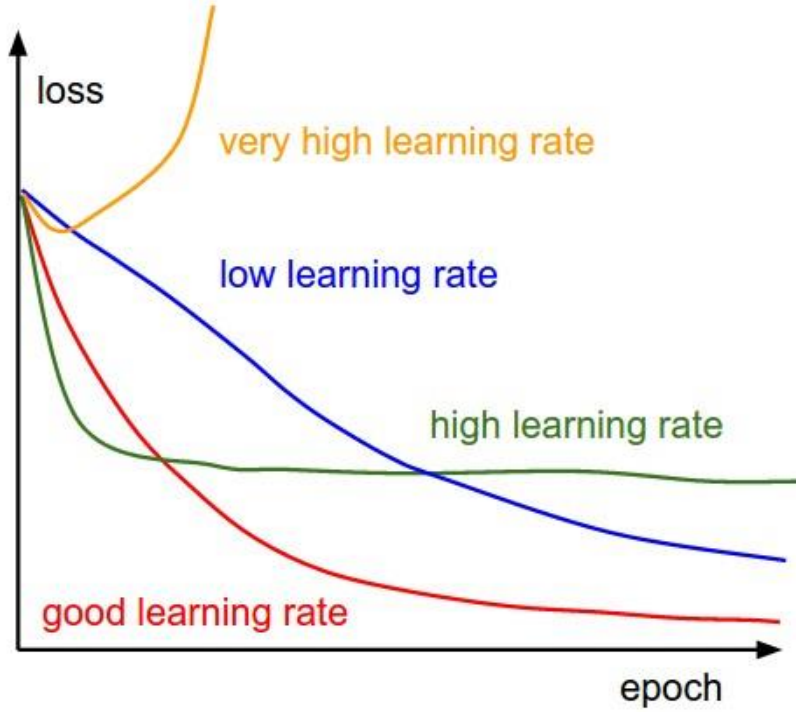
These quantities should be utilized to get intuitions about different hyperparameter settings and how they should be changed for more efficient learning.

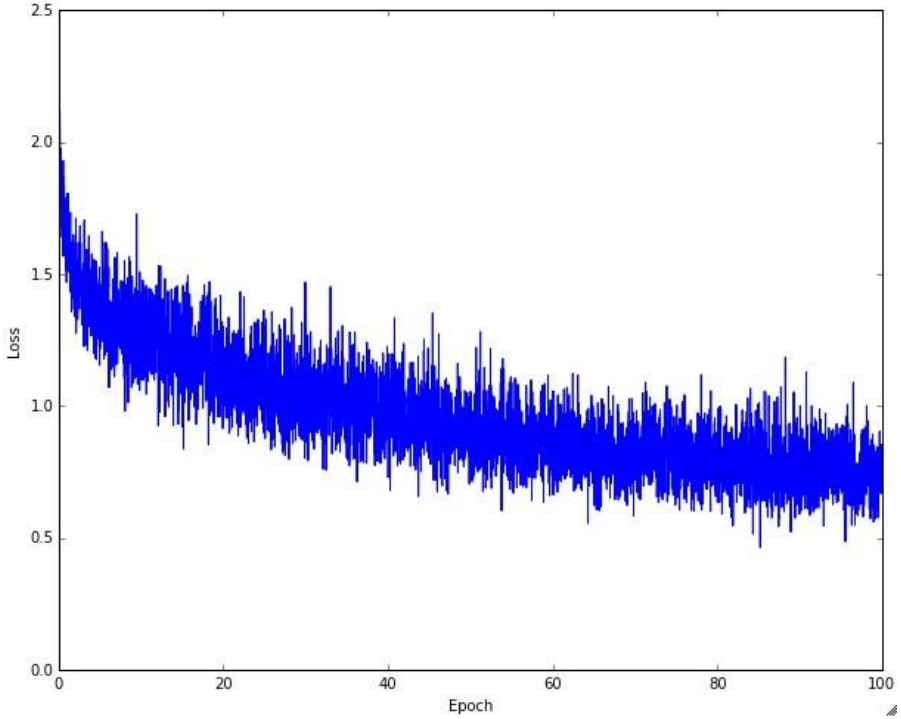
Use plots to visualize these quantities.

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

Plotting Loss

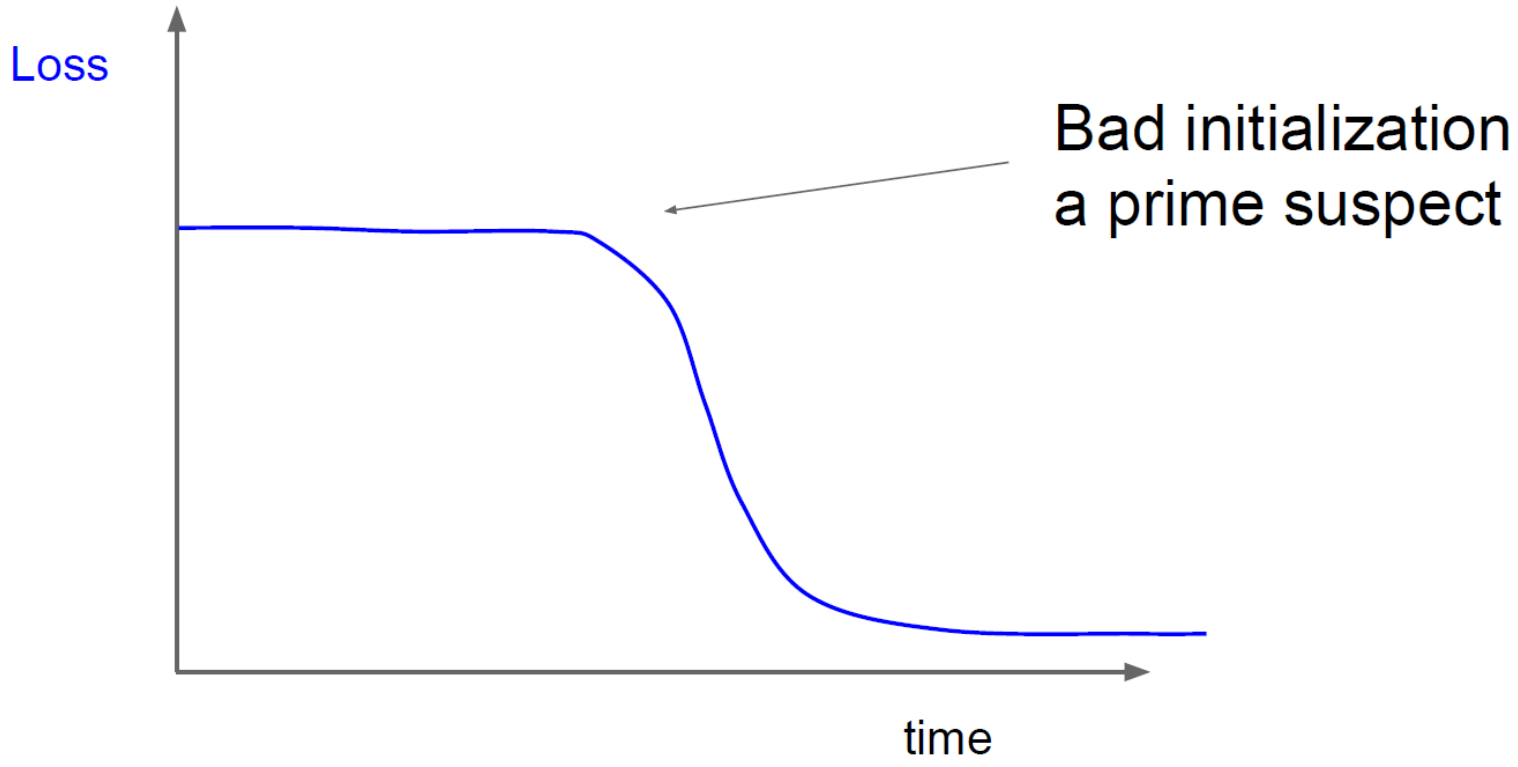
The first quantity that is useful check during training is the loss, as it is evaluated on the forward pass.





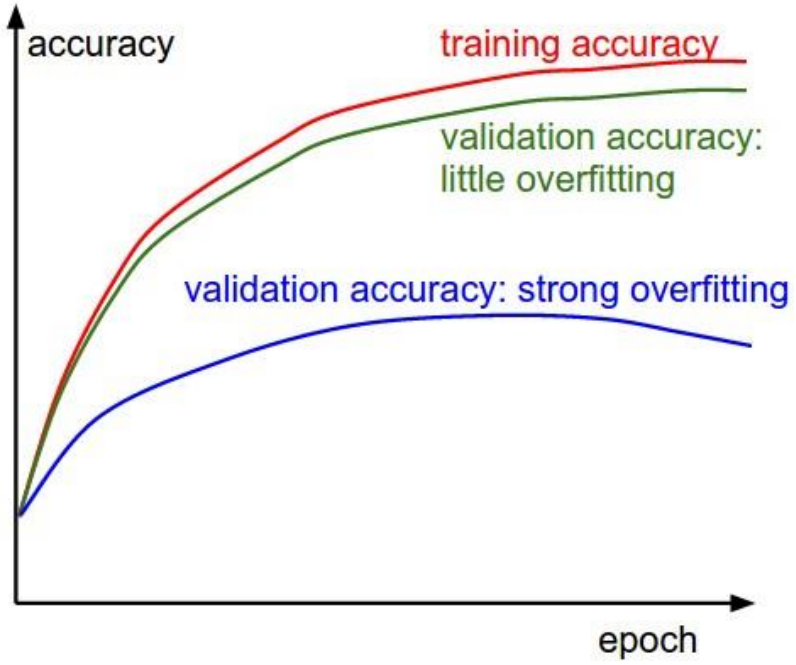
Plotting Loss

Another insight that can be obtained by plotting loss.



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

Double check the loss at the beginning.

When you initialize with small parameters. It's best to first check the data loss alone (so set regularization strength to zero).

For MNIST, with a Softmax classifier we would expect the initial 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.

```
#reg:regularization strength
reg=0.0
net = NeuralNet(input_size, hidden_size, num_classes, reg, lr)
# Train the network using batches of data.
# errorbyepoch keeps the amount of average error at each epoch
errorbyepoch = net.train(X_train, y_train, num_epoch, batch_size)
```

Average error is 2.84598379552

Babysitting the Learning Process

Check if increasing the regularization strength increases the loss.

It should!

```
#number of neurons in each layer
input_size = 28 * 28 * 1
hidden_size = 30
num_classes = 10
#reg:regularization strength
reg=0.1
net = NeuralNet(input_size, hidden_size, num_classes, reg, lr)
# Train the network using batches of data.
# errorbyepoch keeps the amount of average error at each epoch
errorbyepoch = net.train(X_train, y_train, num_epoch, batch_size)
```

Average error is 2.92588628649

Babysitting the Learning Process

Lets try to train now.

Make sure that you can overfit very small portion of the training data.

- take the first 100 examples from MNIST
- turn off regularization (reg = 0.0)

```
X_train = X_train[0:100, :]
y_train = y_train[0:100]
reg=0.0
lr=1e-3  #learning rate
num_epoch=100
batch_size=100
net = NeuralNet(input_size, hidden_size, num_classes, reg, lr)
# Train the network using batches of data.
# errorbyepoch keeps the amount of average error at each epoch
errorbyepoch = net.train(X_train, y_train, num_epoch, batch_size)
```

Average error of epoch 100 is 0.0153391290266
Train accuracy: 1.0

Very small loss, train accuracy 1.00, nice!

Babysitting the Learning Process

Lets try to train now

Start with small regularization and find learning rate that makes the loss go down.

```
reg=0.001
lr=1e-8  #learning rate
num_epoch=10
```

```
Average error of epoch  1  is 2.84359923077
Average error of epoch  2  is 2.83657729366
Average error of epoch  3  is 2.82966012293
Average error of epoch  4  is 2.82284475026
Average error of epoch  5  is 2.81613163883
Average error of epoch  6  is 2.80951487591
Average error of epoch  7  is 2.80299120561
Average error of epoch  8  is 2.79655753551
Average error of epoch  9  is 2.79020944156
Average error of epoch  10  is 2.78394462842
```

Loss barely changing: Learning rate is probably too low!

Babysitting the Learning Process

Now let's try a high learning rate.

```
reg=0.001
lr=1e-1   #learning rate
num_epoch=10
```

```
Average error of epoch  1  is nan
Average error of epoch  2  is nan
Average error of epoch  3  is nan
Average error of epoch  4  is nan
Average error of epoch  5  is nan
Average error of epoch  6  is nan
…
```

loss exploding: NaN almost always means high learning rate...

```
reg=0.001
lr=2e-2   #learning rate
num_epoch=10
```

learning rate is still too high.

```
Average error of epoch  1  is 2.6771203292
Average error of epoch  2  is 2.50513688532
Average error of epoch  3  is 2.50483412627
Average error of epoch  4  is 2.50482736583
Average error of epoch  5  is 2.50482627833
Average error of epoch  6  is 2.50482603725
Average error of epoch  7  is 2.50482597249
Average error of epoch  8  is 2.50482595353
…
```

Babysitting the Learning Process

You can also check the ratio: update magnitudes / weight magnitudes.

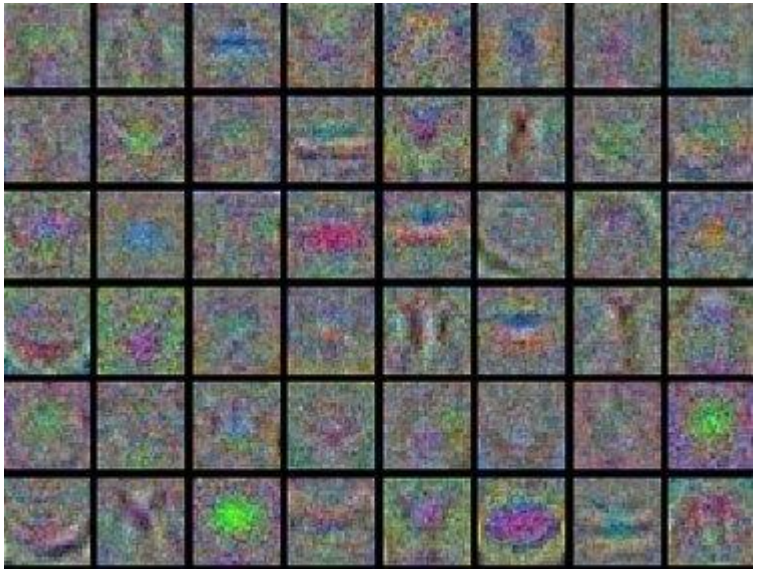
- A rough heuristic is that this ratio should be somewhere around 1e-3.
- If it is very low, then the learning rate might be too low. If it is very high then the learning rate is likely too high.

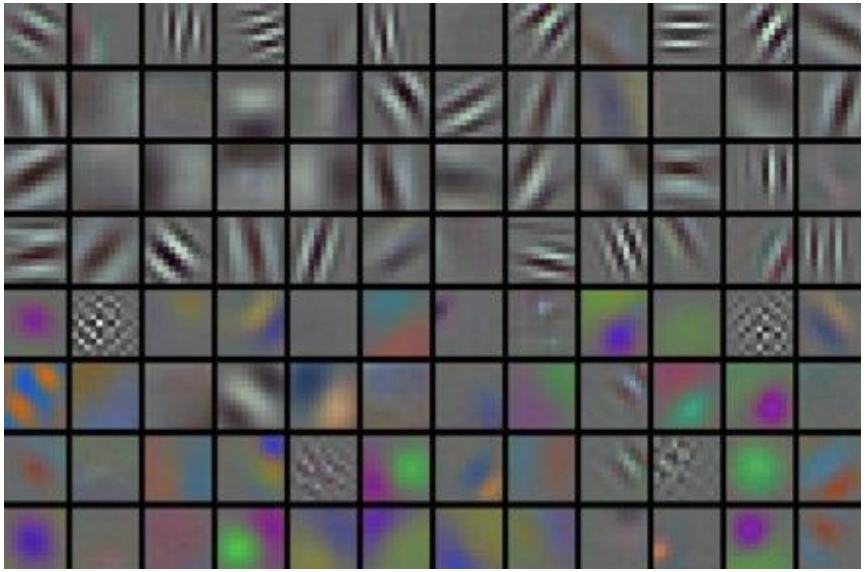
```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W)
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update)
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Babysitting the Learning Process

First-layer visualizations (with image data).

It can be helpful to plot the first-layer features visually:

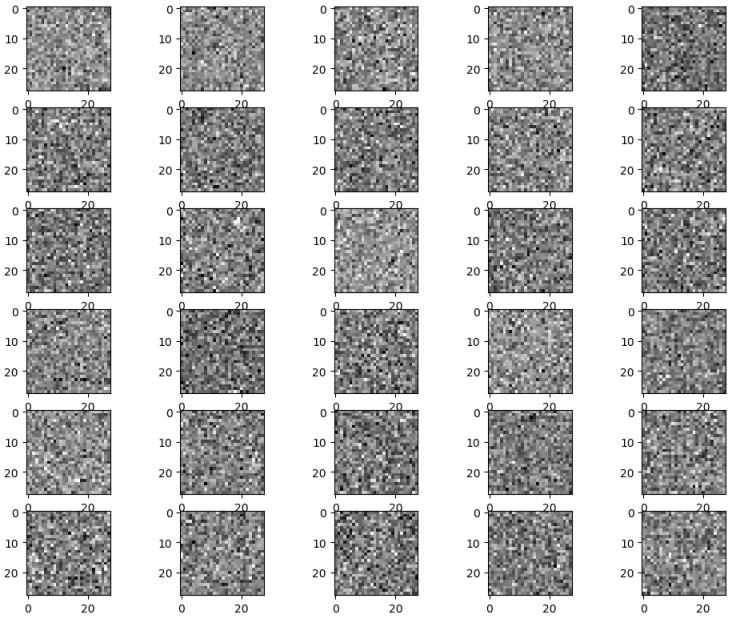




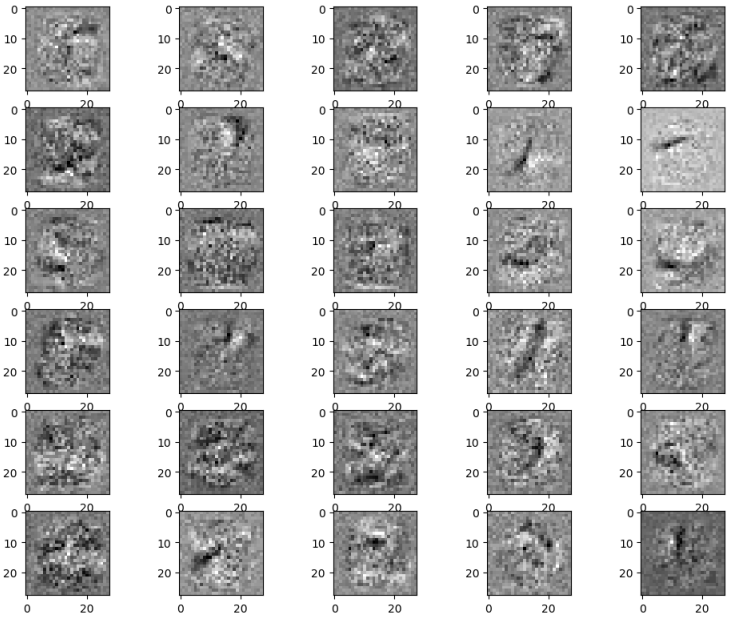
Examples of visualized weights for the first layer of a neural network. Left: Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. Right: Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

Babysitting the Learning Process

Example first-layer visualizations from MNIST dataset:



Unoptimized weights



Weights after 80 epochs

Hyperparameter Optimization

The most common hyperparameters in context of NNs include:

- 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: only a few epochs to get rough idea of what params work
  - Second stage: longer running time, finer search

Hyperparameter Optimization

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 Optimization

Now run finer search...

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

adjust range

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

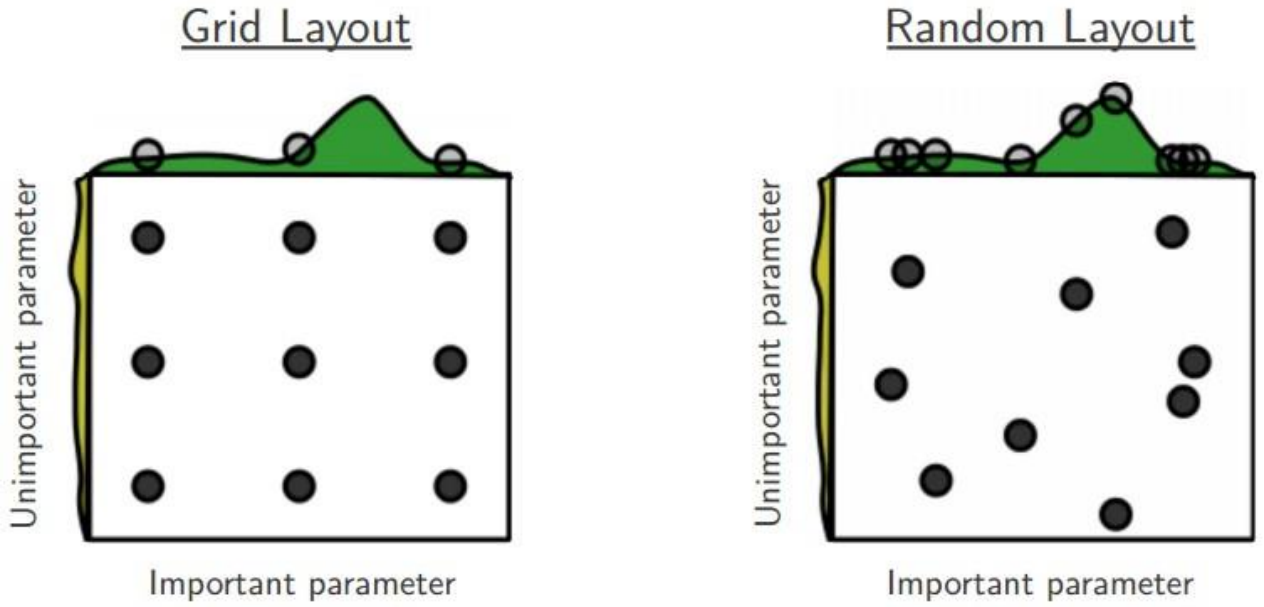
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, 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 Optimization

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 Optimization

neural networks practitioner

music = loss function

Regularization – Add term to loss

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

data loss

regularization loss

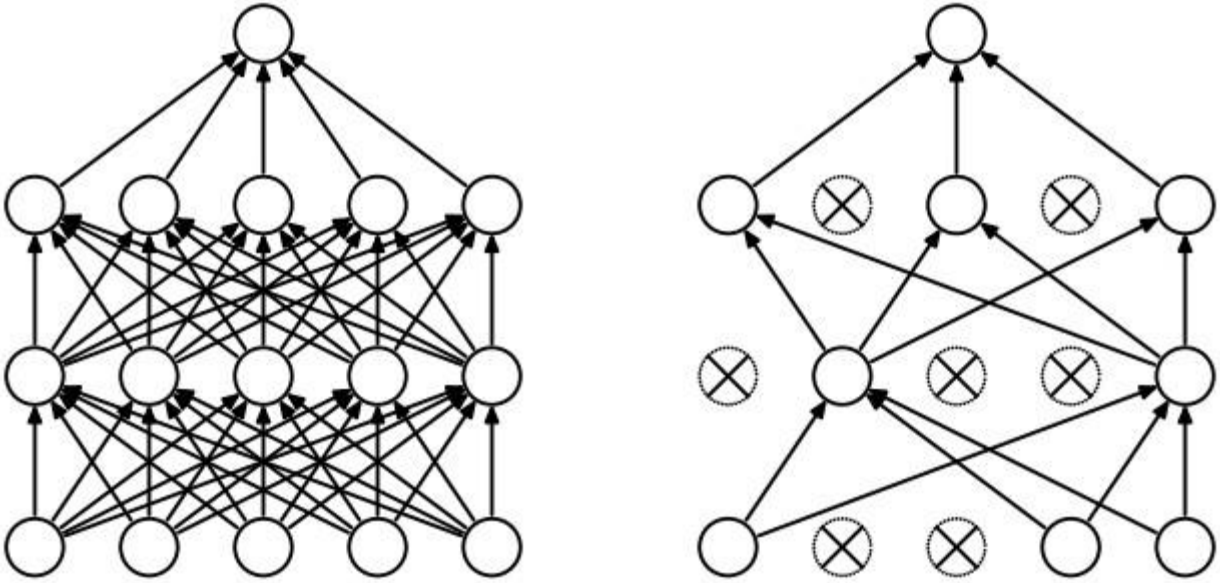
- L2 regularization $\longrightarrow R(W) = \sum_k \sum_l W_{k,l}^2$
- L1 regularization $\longrightarrow R(W) = \sum_k \sum_l |W_{k,l}|$
- Elastic net (L1 + L2) $\longrightarrow R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

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

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

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

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

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

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

Example forward pass with a 3-layer network using dropout

At test time, we are not dropping anymore, but we are performing a scaling of both hidden layer outputs by **p**.

Regularization - Dropout

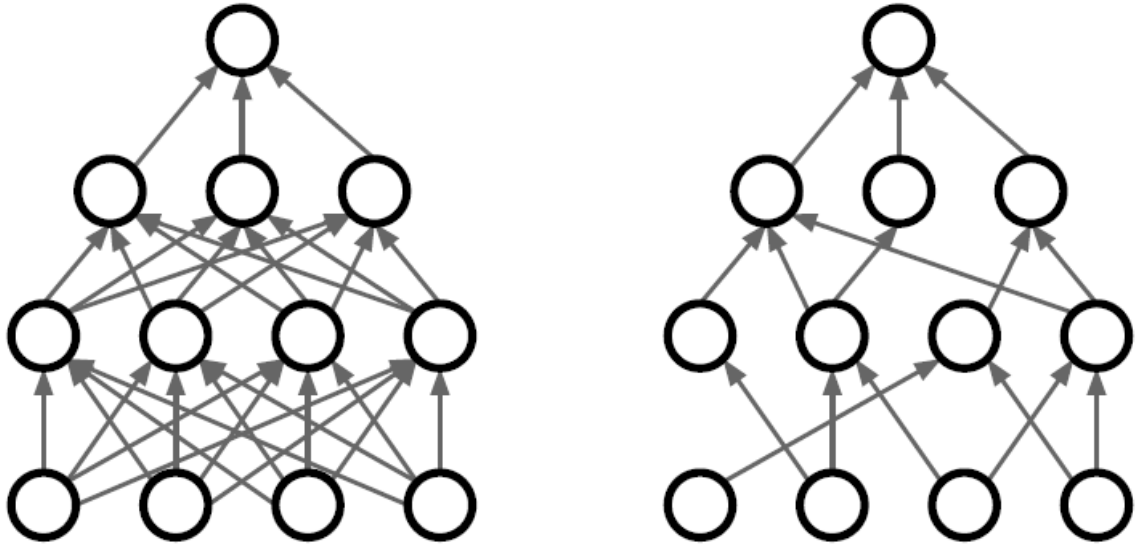
Q. How could this possibly be a good idea?

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

Regularization

Dropout introduces randomness while training the network.

Another attempt in this direction is **DropConnect**, where a random set of **weights** is set to zero during forward pass.



Regularization

Remember when we saw Batch Normalization, we said it serves as a regularization technique.

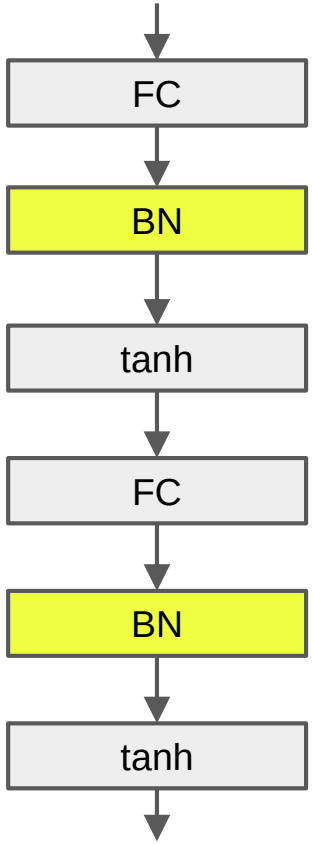
How?

<u>At training:</u>

We normalize using statistics from random minibatches. I.e. we add some kind of randomness.

<u>At test time:</u>

We average out randomness by using fixed values (statistics) from all dataset.



Regularization – In practice

Bias regularization: It is not common to regularize the bias parameters

Per-layer regularization: It is not very common to regularize different layers to different amounts (except perhaps the output layer).

In practice: It is most common to use a single, global L2 regularization strength that is cross-validated. It is also common to combine this with dropout applied after all layers. The value of **p=0.5** is a reasonable default.

Parameter Updates

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

Optimization for deep networks is currently a very active area of research.

We'll highlight some established and common techniques and briefly describe their intuition,

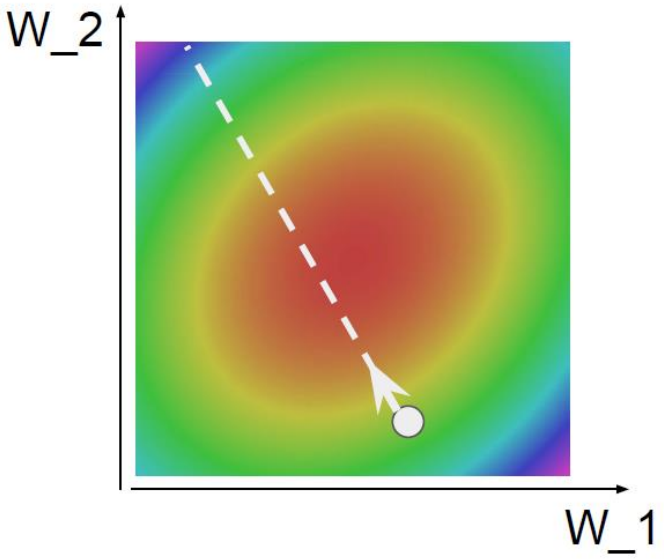
Stochastic Gradient Descent

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

gradient

vector of parameters

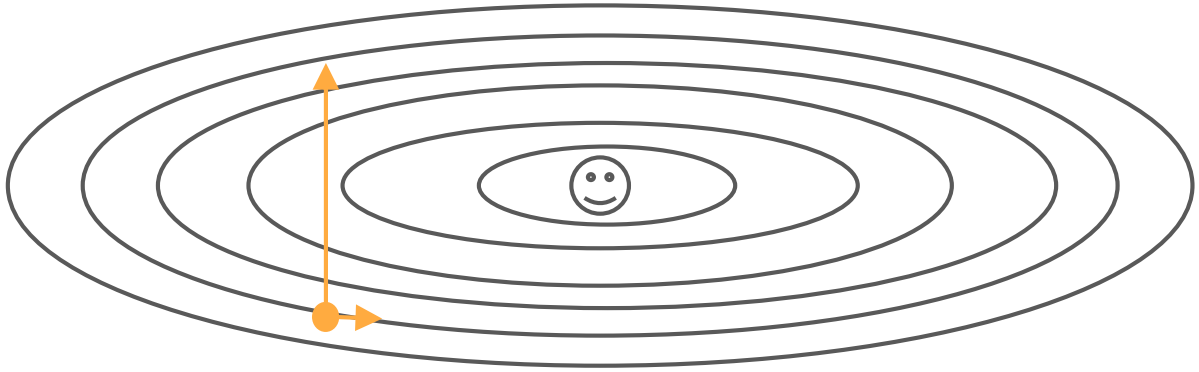
```
w += - learning_rate * dw
```



Stochastic Gradient Descent

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

What does gradient descent do?

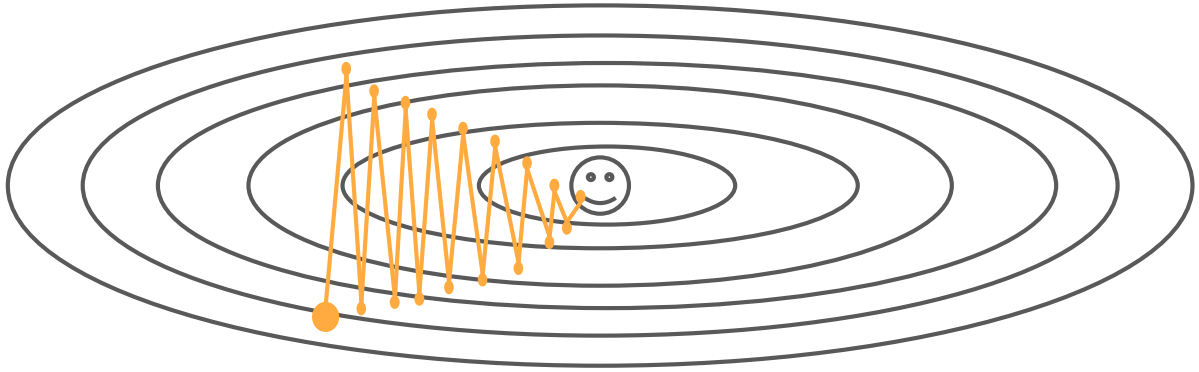


Stochastic Gradient Descent

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

What does gradient descent do?

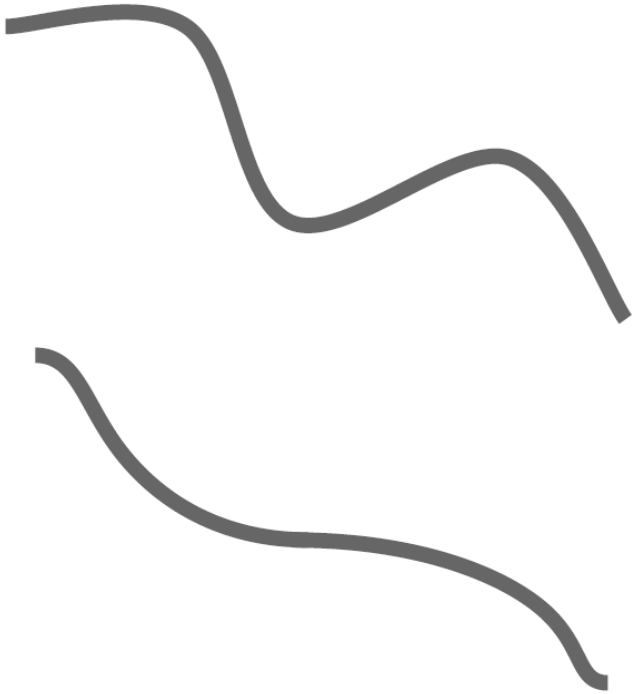
Very slow progress along shallow dimension, zig-zag movement



Stochastic Gradient Descent

Another problem with SGD.

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

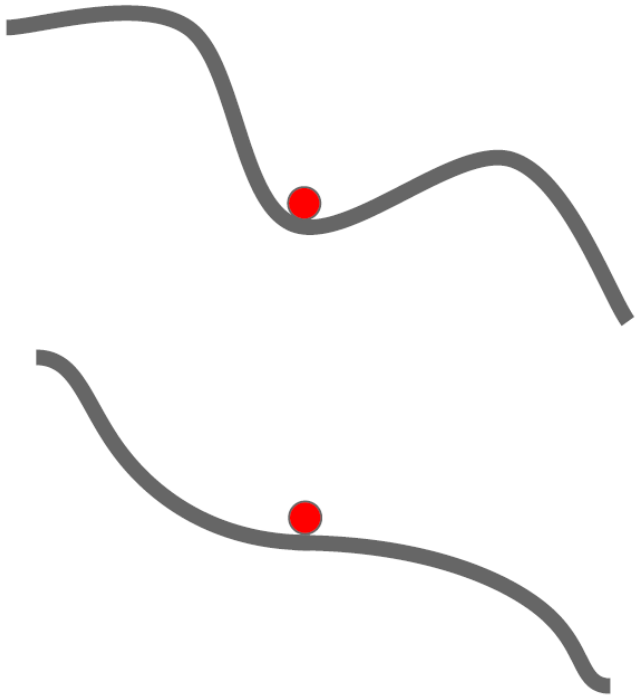


Stochastic Gradient Descent

Another problem with SGD.

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

Zero gradient, gradient descent gets stuck



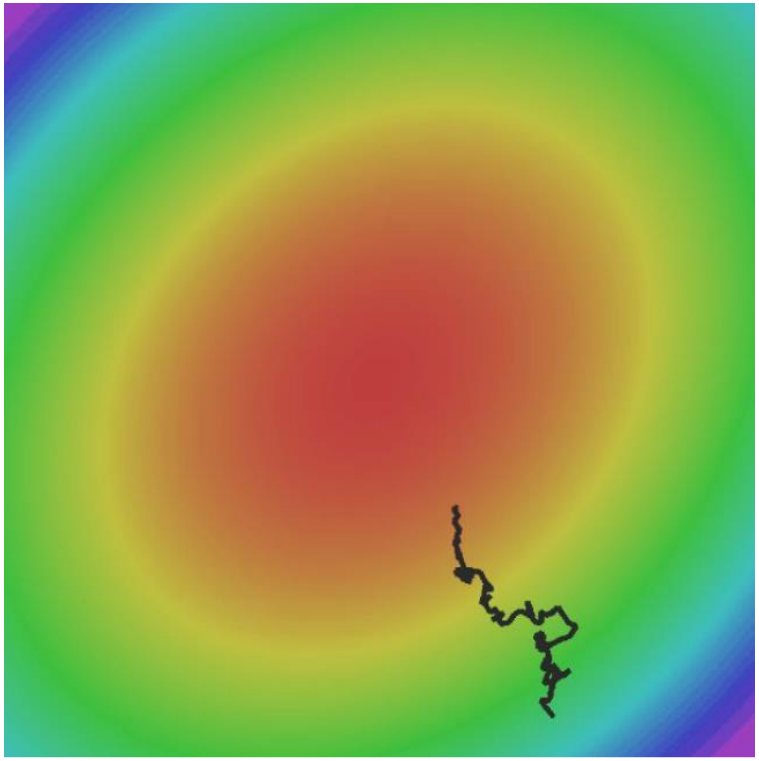
Saddle points much more common in high dimension*

* 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 minibatches 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.

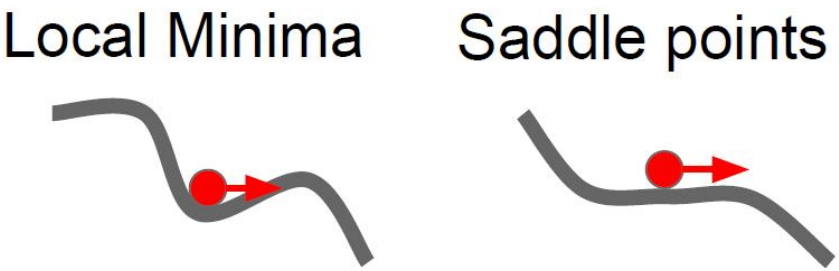
The loss can be interpreted as a the height of a hilly terrain.

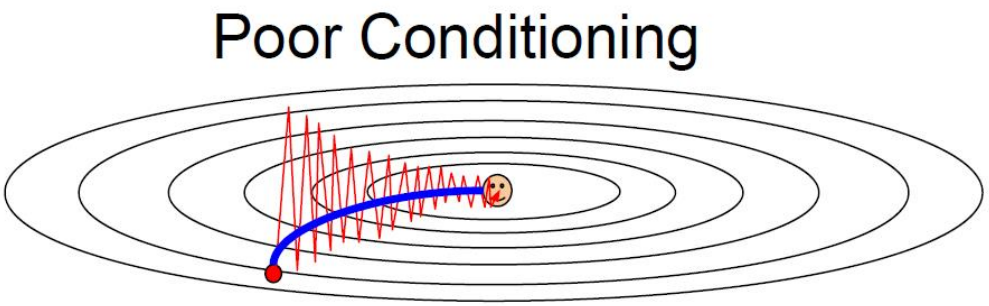
Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location.

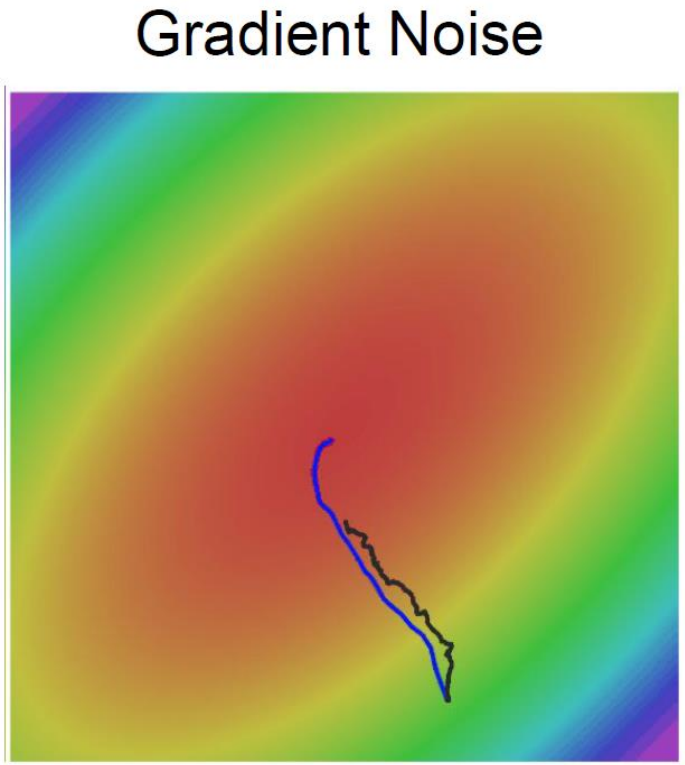
The optimization process can be seen as the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.

SGD + Momentum

Momentum Update can help with all problems we have seen.



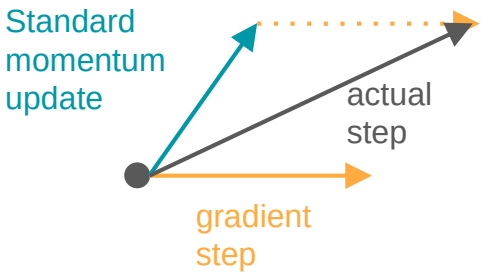


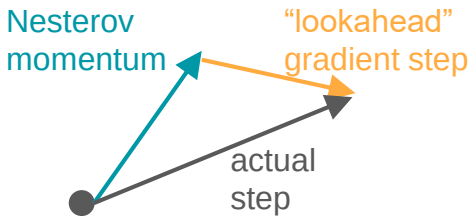


SGD + Momentum

Nesterov Momentum:

- Gradient is computed at the future approximate position `x + mu * 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

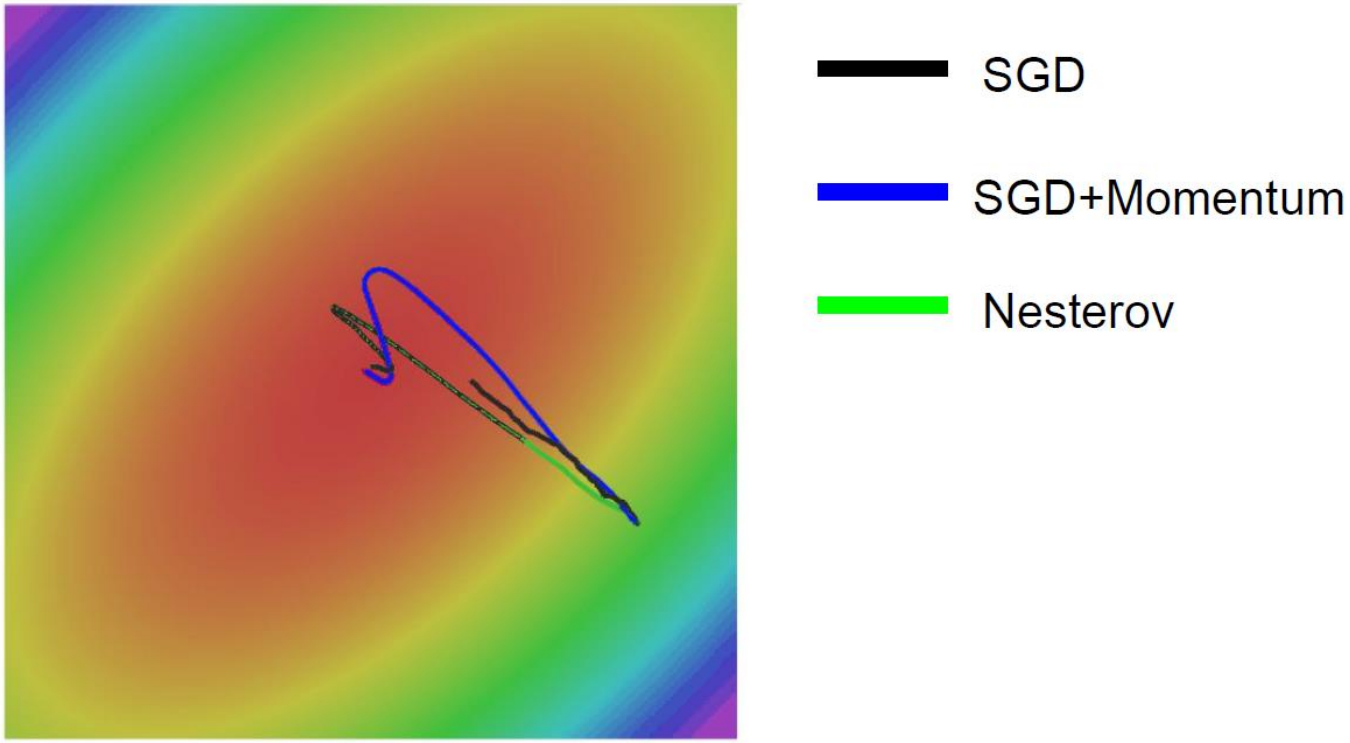
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

```
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:

Adagrad: 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)
```

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

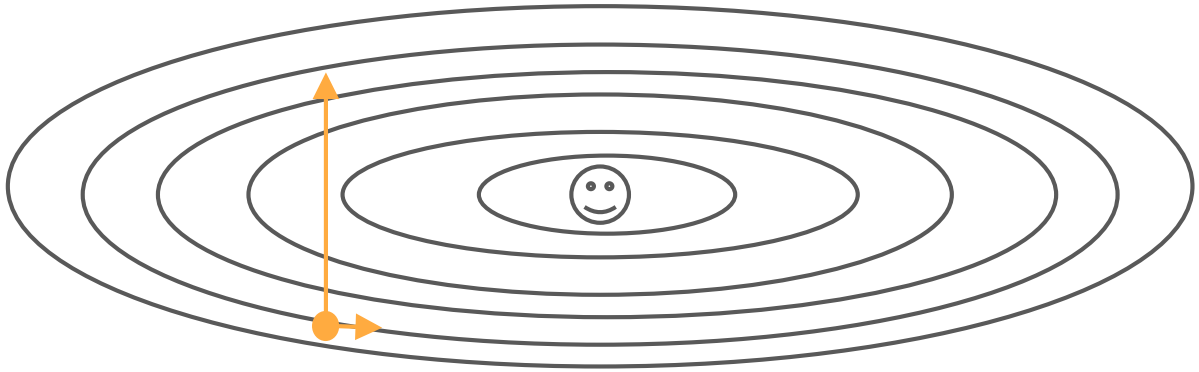
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

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.

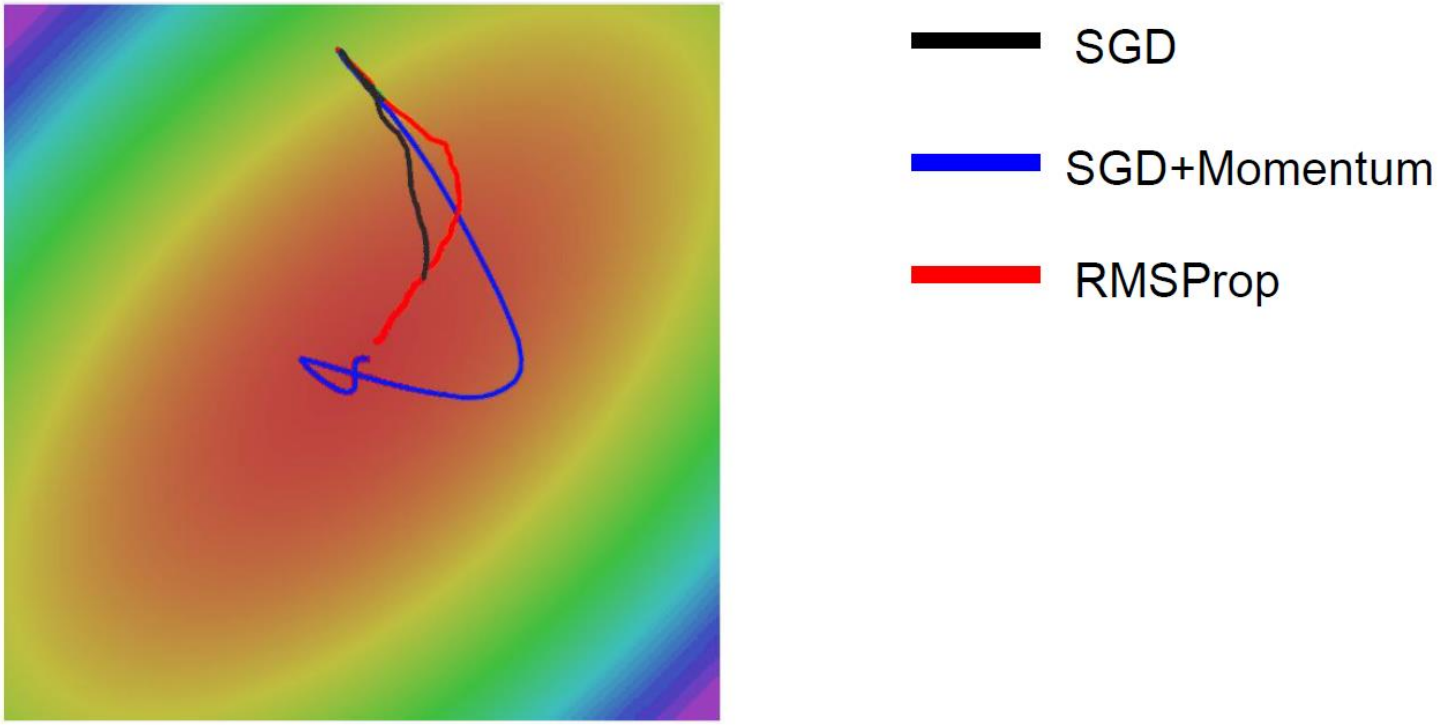
```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

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”.

Parameter Updates

RMSProp vs. others



Parameter Updates

Per-Parameter Adaptive Learning Rate Methods:

Adam: A 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

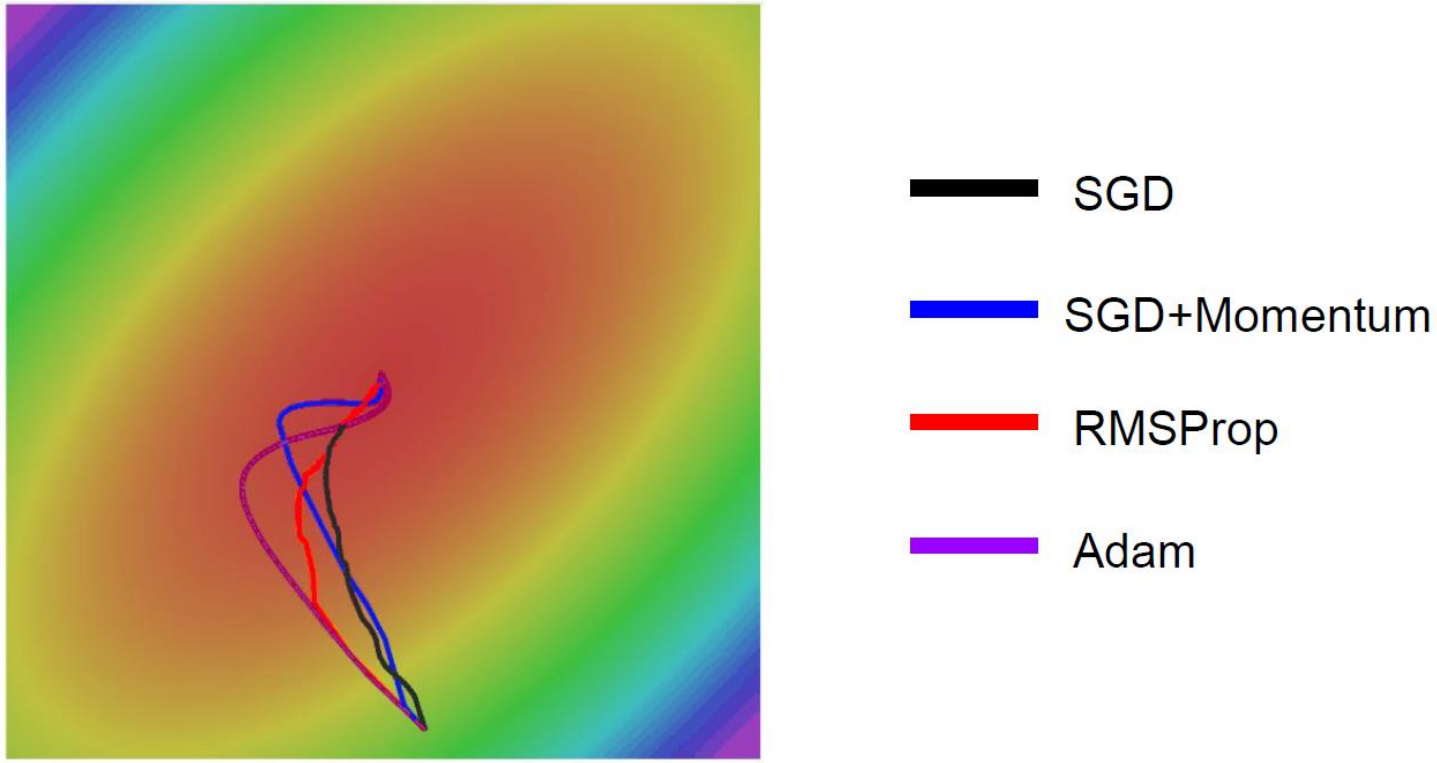
~RMSprop

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`.

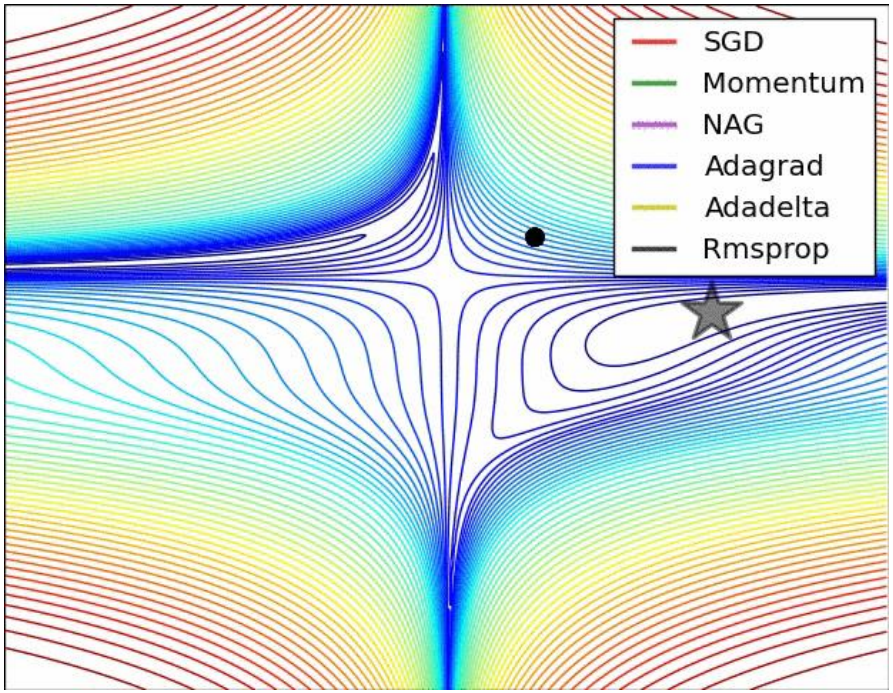
* Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

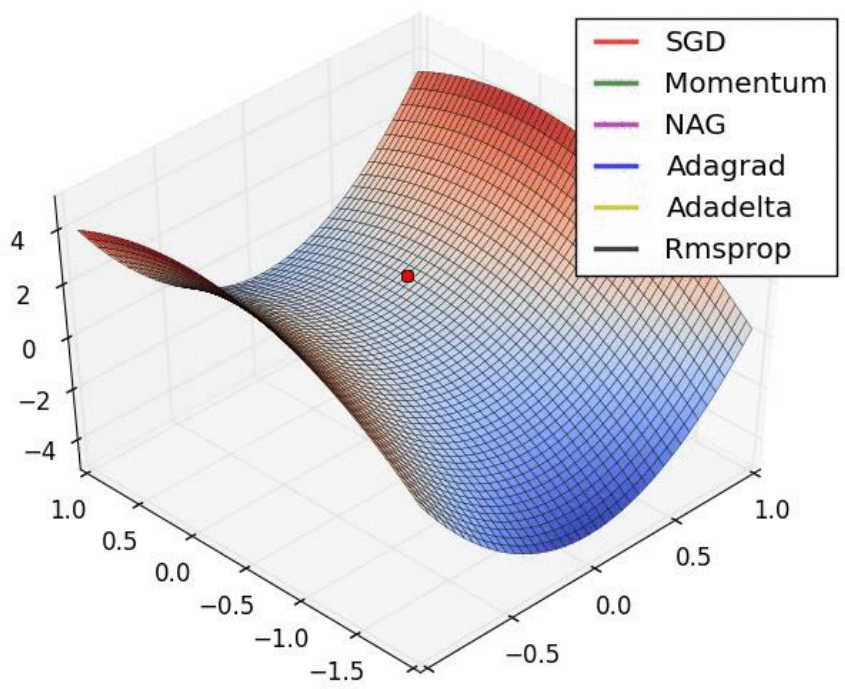
Parameter Updates

All together



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!

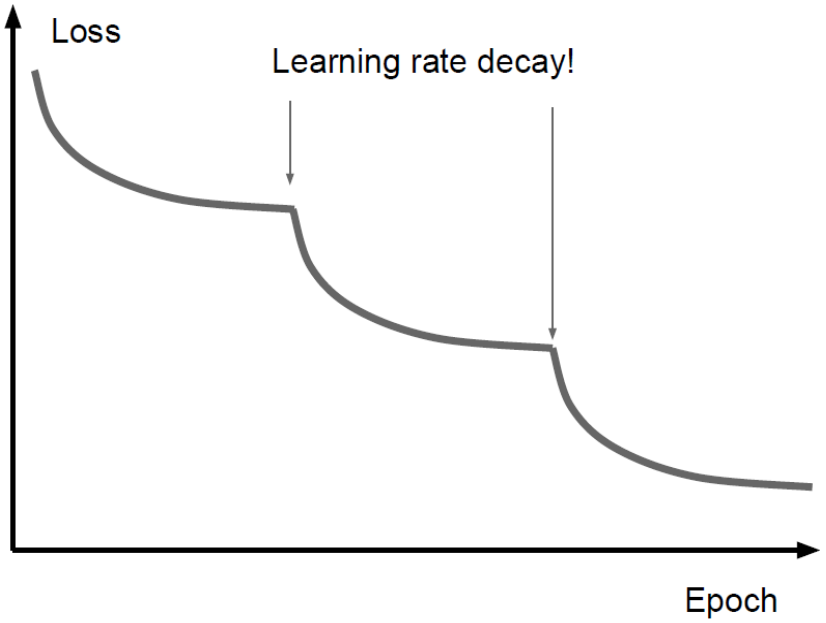
step decay:
e.g. decay learning rate by half every few epochs.

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.

Summary

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

During training, **monitor** the **loss**, the **training/validation accuracy**, and if you're feeling fancier, the **magnitude of updates** in relation to parameter values (it should be ~1e-3), 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 only for 1-5 epochs) to fine (narrower ranges, training for many more epochs).

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

The two recommended updates to use are either **SGD+Nesterov Momentum** or **Adam**.

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