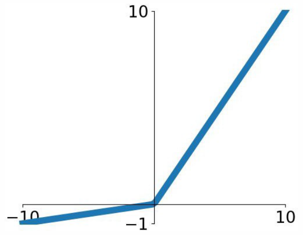
NETWORK TRAINING
OPTIMIZER, DATA, AND HYPER-PARAMETERS

Chih-Chung Hsu (許志仲)
Institute of Data Science
National Cheng Kung University
https://cchsu.info

Last time: Activation Functions

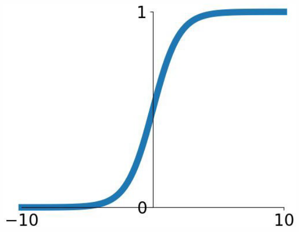
Leaky ReLU

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



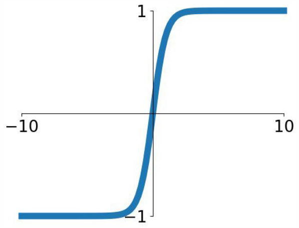
Sigmoid

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



tanh

$$\tanh(x)$$

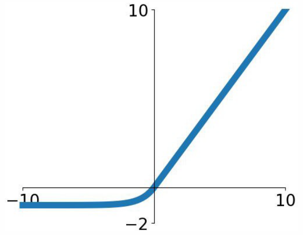


Maxout

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

ELU

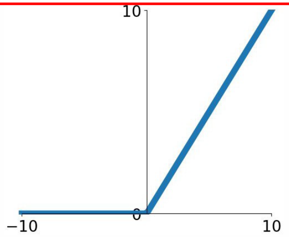
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



ReLU

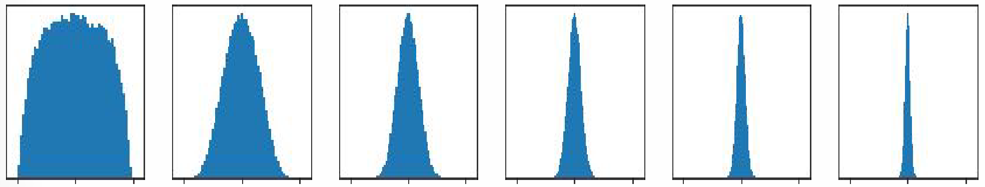
$$\max(0, x)$$

Good!!

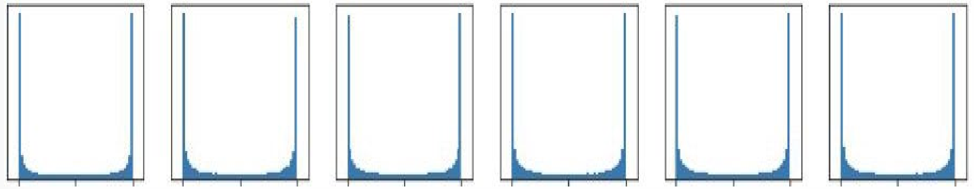


Slide Credit cs231n

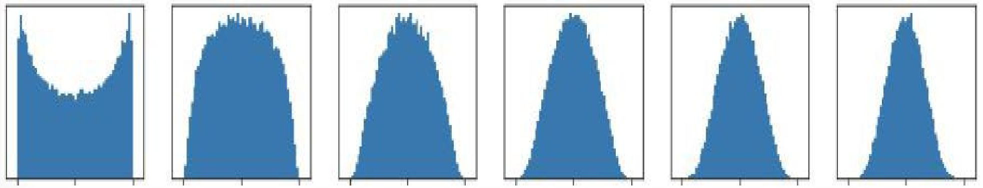
Last time: Weight Initialization



Initialization too small:
Activations go to zero, gradients also zero,
Failed to learn



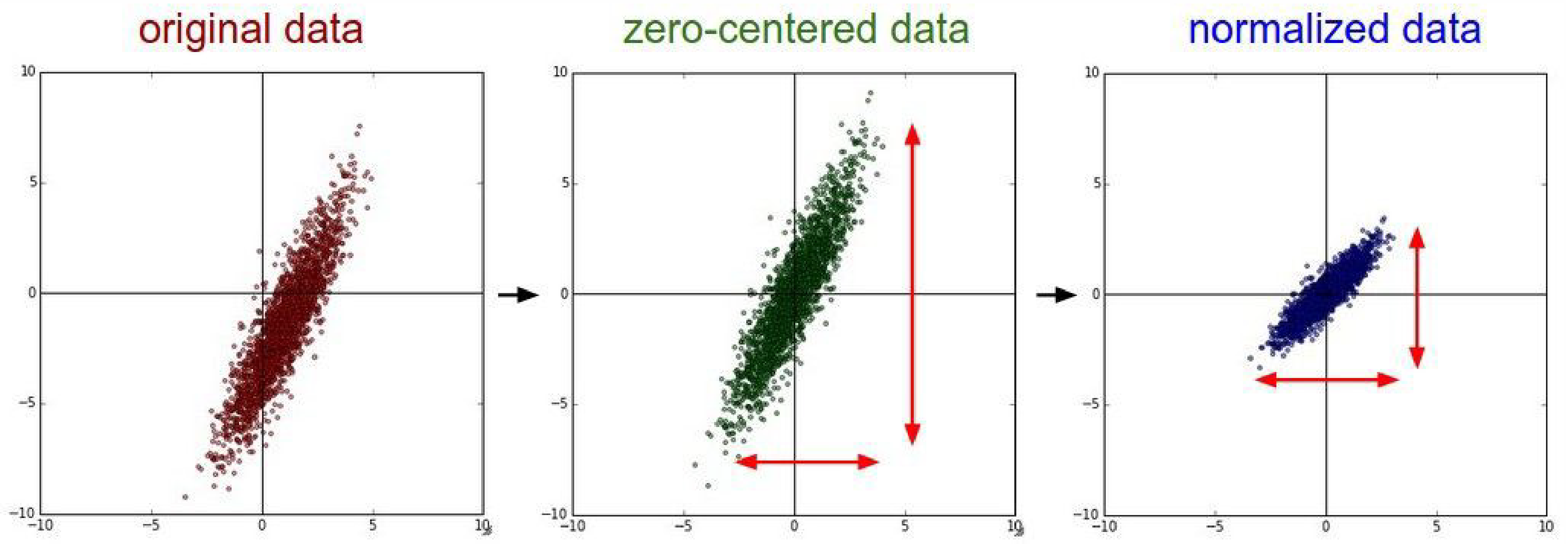
Initialization too big:
Activations saturate (for tanh),
Gradients zero, Failed to learn,



Initialization just right:
Nice distribution of activations at all layers,
Learning proceeds nicely

Slide Credit cs231n

Last time: Data Preprocessing



Slide Credit cs231n

Last Time: Batch Normalization [Ioffe and Szegedy, 2015]

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

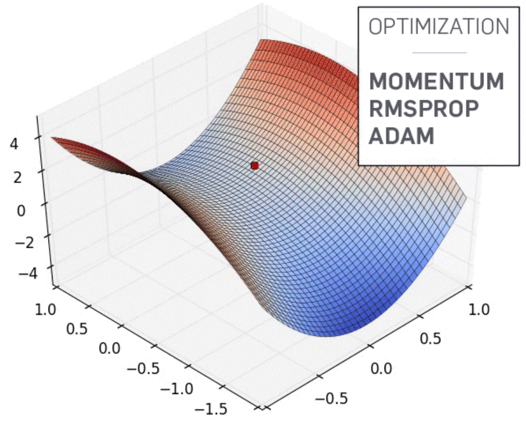
$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Slide Credit cs231n

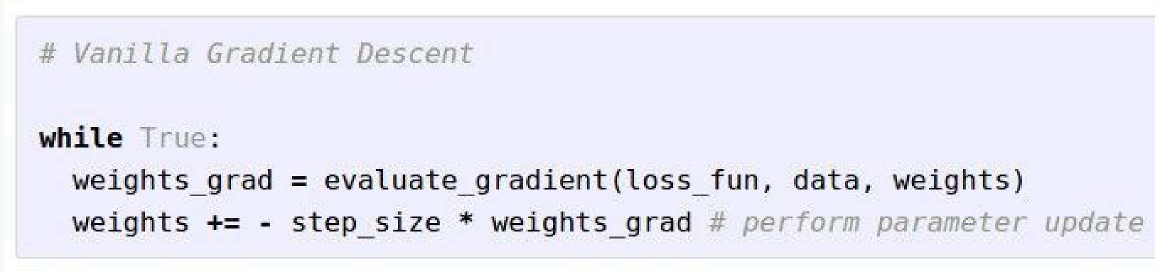
And now we will offer you

- Improve your training error:
  - Optimizers
  - Learning rate schedules
  - Data augmentation
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters

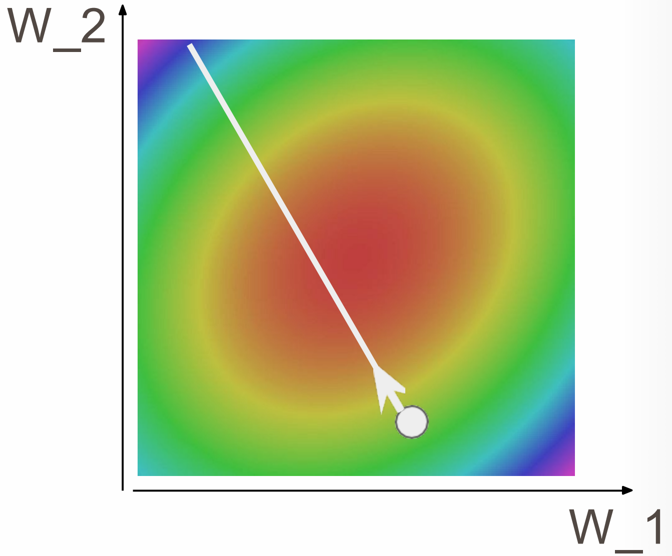


Slide Credit cs231n

Optimization



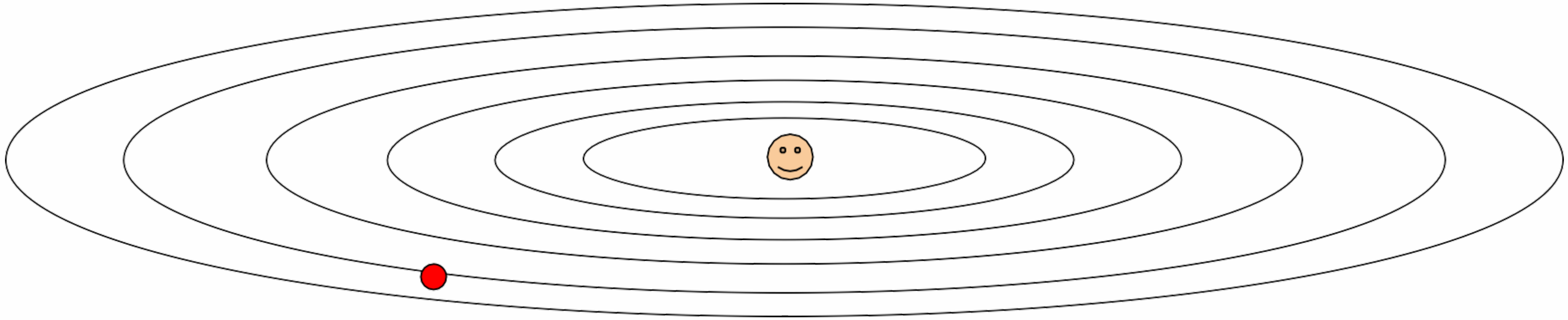
```
# Vanilla Gradient Descent

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



Slide Credit cs231n

Optimization: Problems with SGD

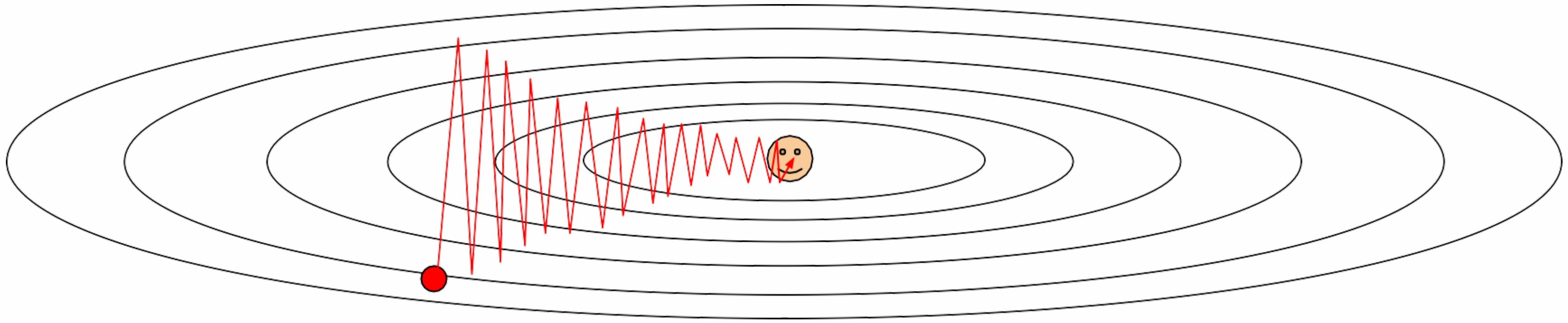
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problems with SGD

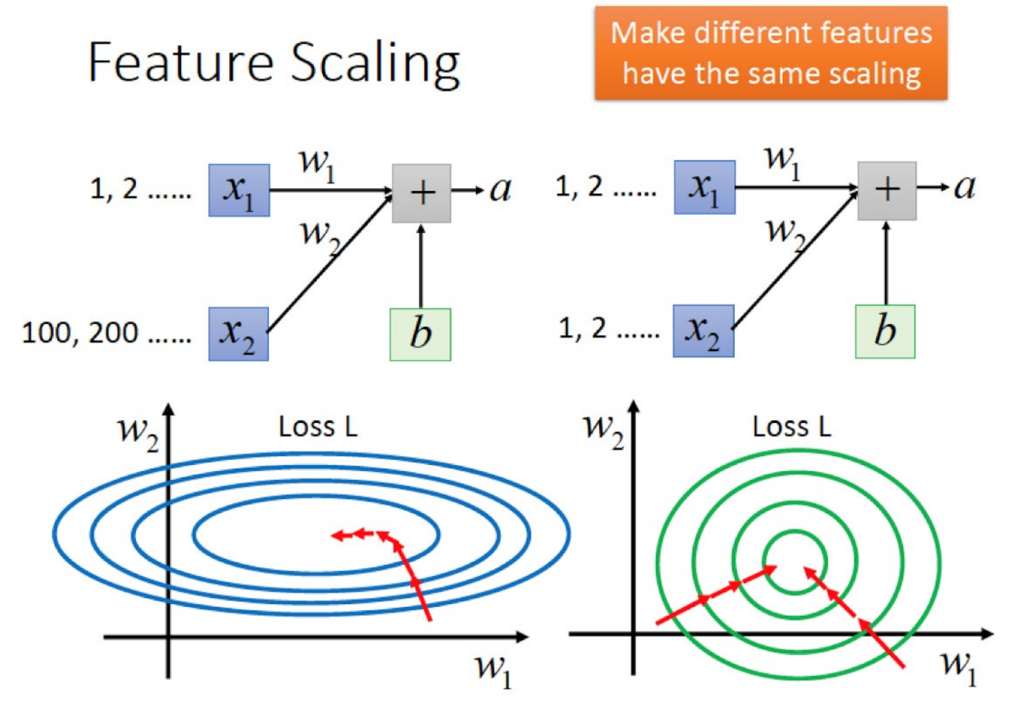
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

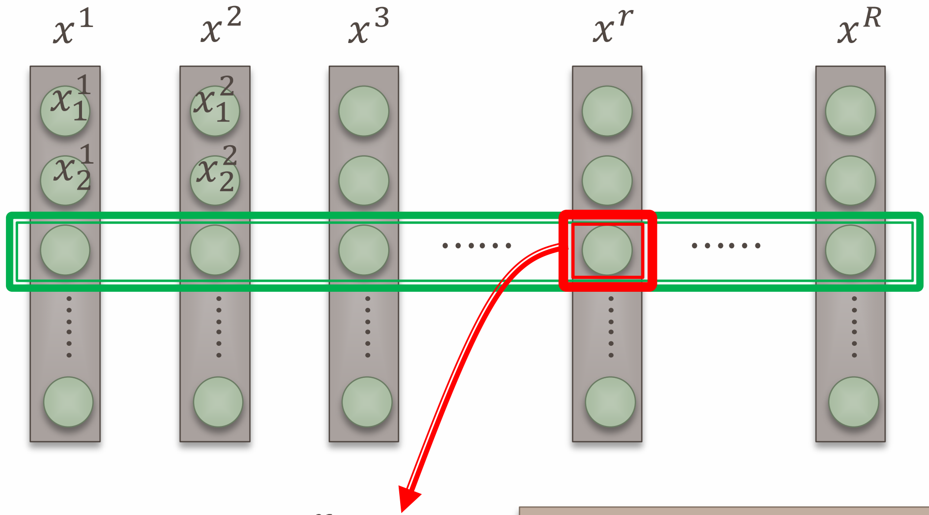
Slide Credit cs231n

Optimization Issue: scaling



Slide Credit cs231n

Feature Re-Scaling



For each dimension i:

mean: m_i

standard deviation: σ_i

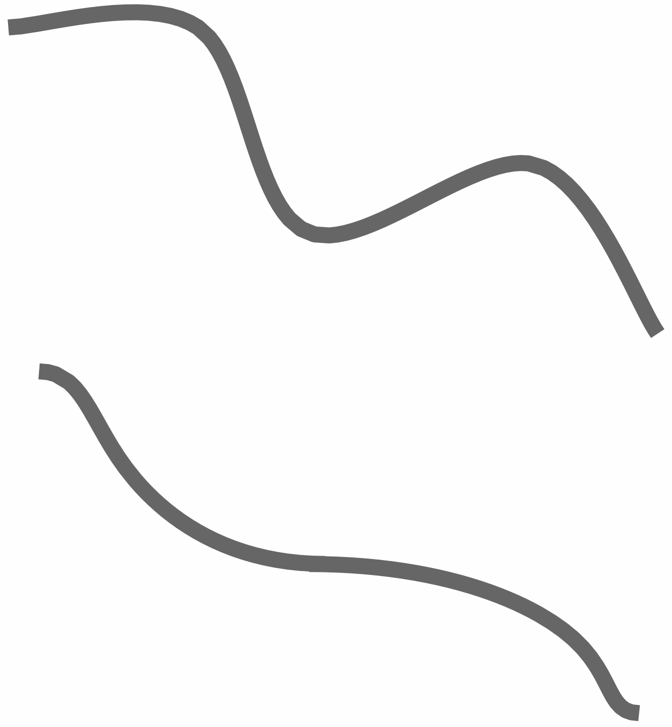
$$x_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i}$$

The means of all dimensions are 0, and the variances are all 1

In general, gradient descent converges much faster with feature scaling than without it.

Optimization: Problems with SGD

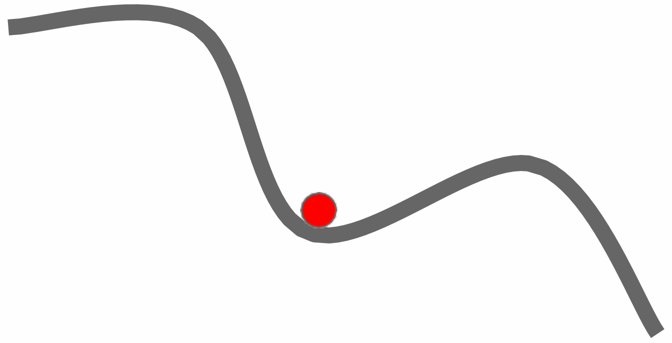
What if the loss function has a **local minima** or **saddle point**?

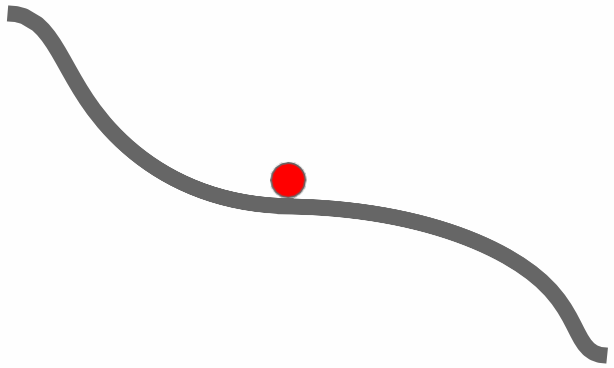


Optimization: Problems with SGD

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

Zero gradient, gradient descent gets stuck



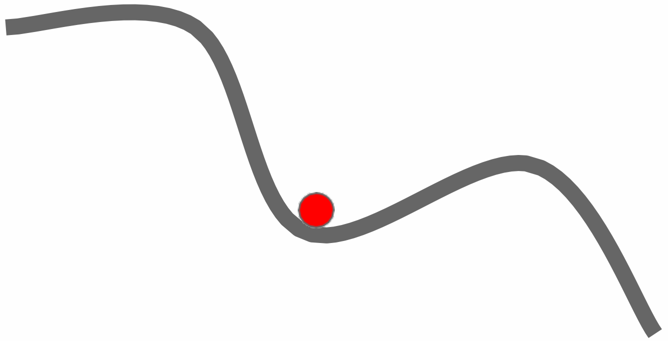


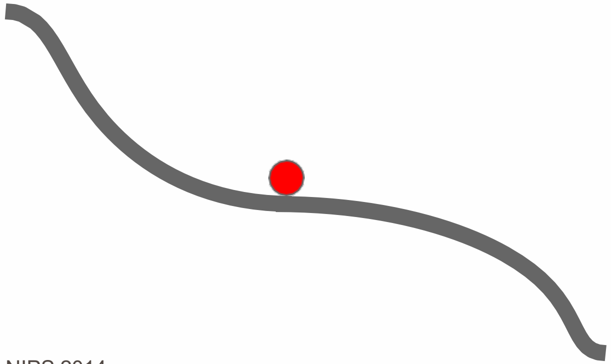
Slide Credit cs231n

Optimization: Problems with SGD

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

Saddle points much more common in high dimension





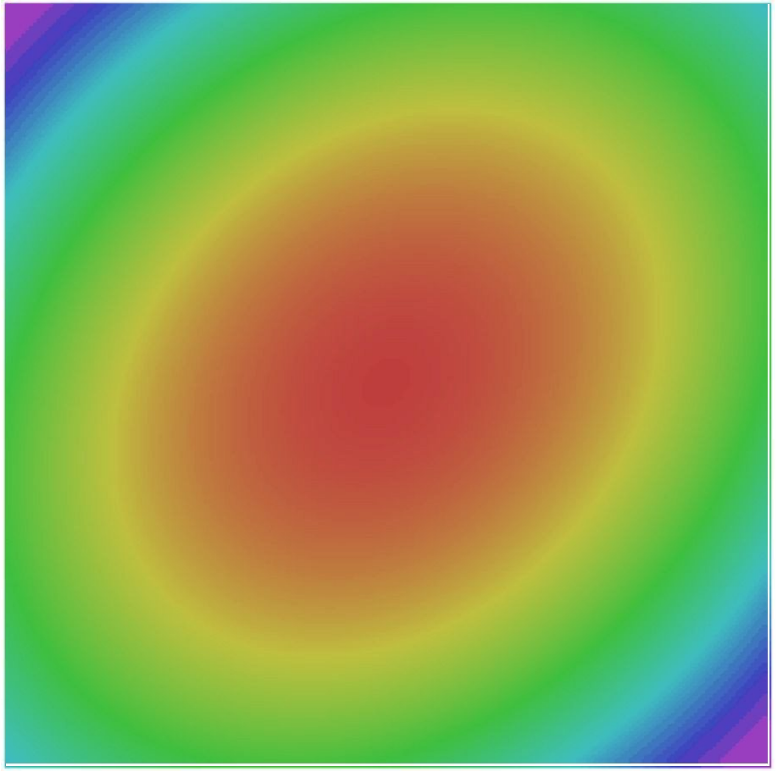
Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Slide Credit cs231n

SGD + Momentum

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 “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

SGD+Momentum

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

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

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
```

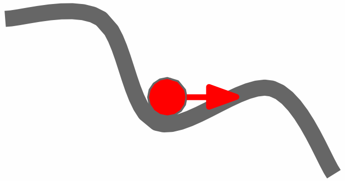
You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

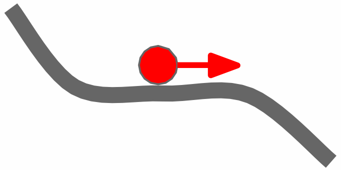
Slide Credit cs231n

SGD + Momentum

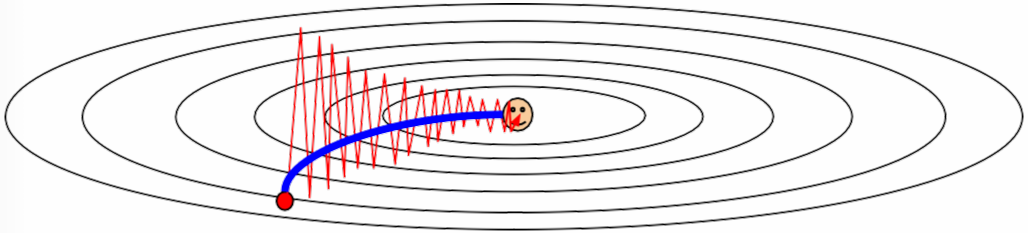
Local Minima



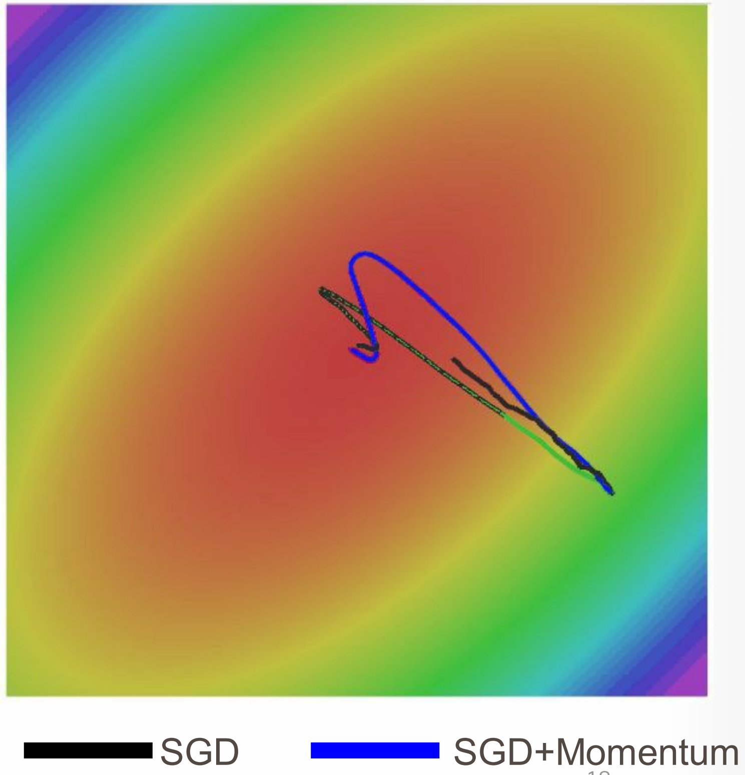
Saddle points



Poor Conditioning



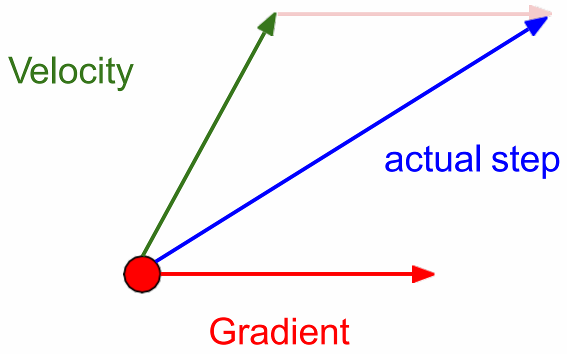
Gradient Noise



Slide Credit cs231n

SGD+Momentum

Momentum update:

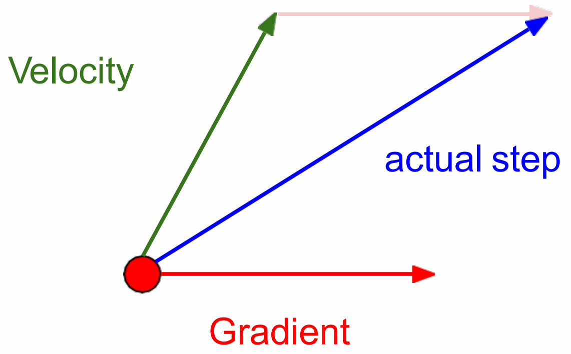


Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

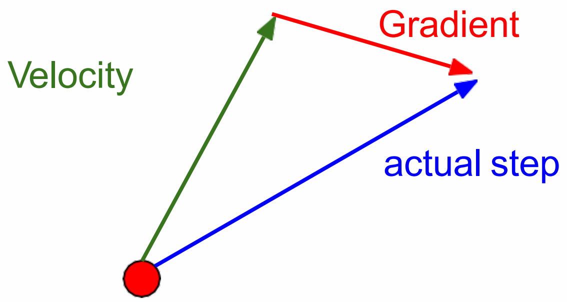
Nesterov Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



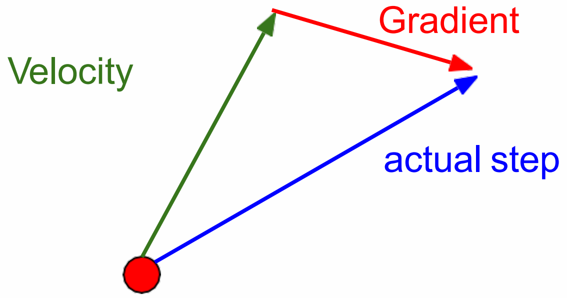
“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Nesterov Momentum

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

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

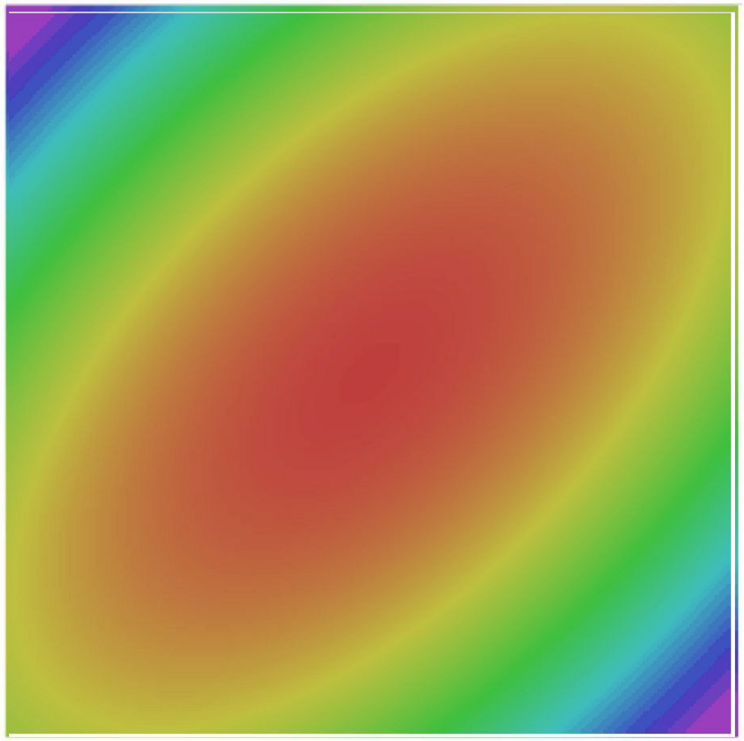


Change of variables $\tilde{x}_t = x_t + \rho v_t$ rearrange:

$$\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\ \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned}$$

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum



▬	SGD
🟦	SGD+Momentum
🟩	Nesterov

Slide Credit cs231n

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

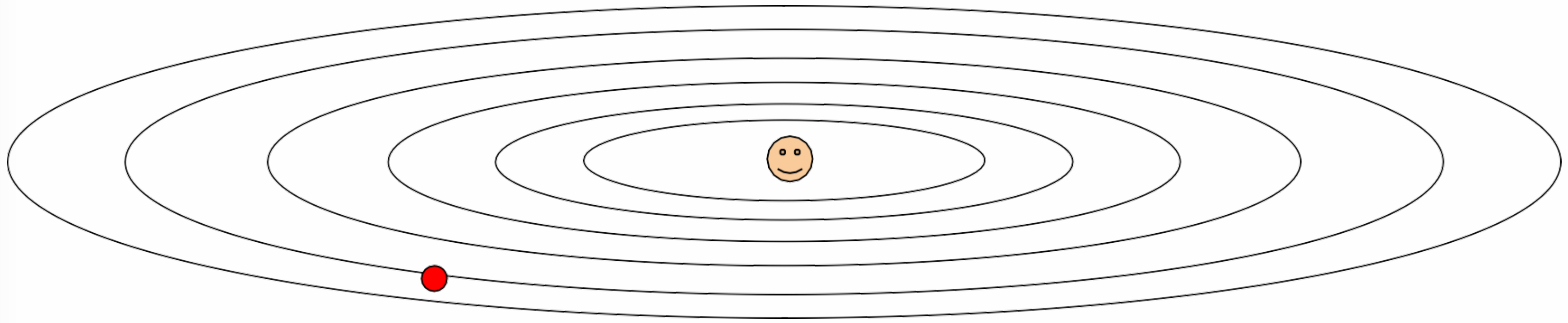
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates" or "adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



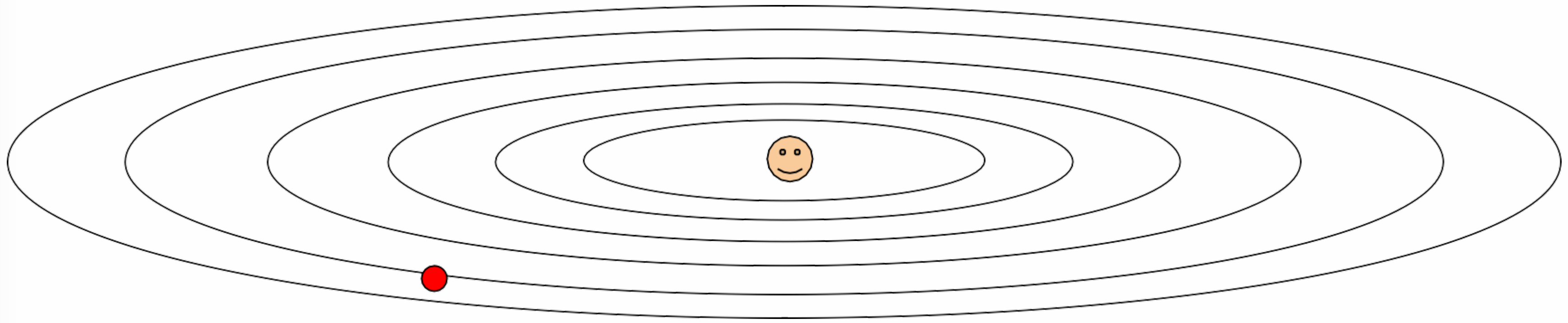
Think about it: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

Slide Credit cs231n

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Step size changing in long time?

Decays to zero

RMSProp: “Leaky AdaGrad”

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

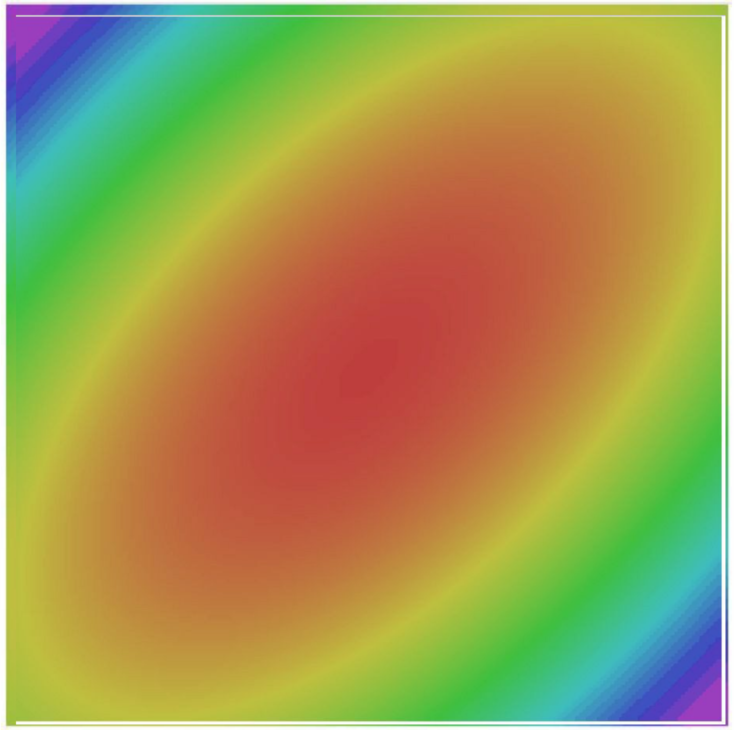
RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Slide Credit cs231n

RMSProp

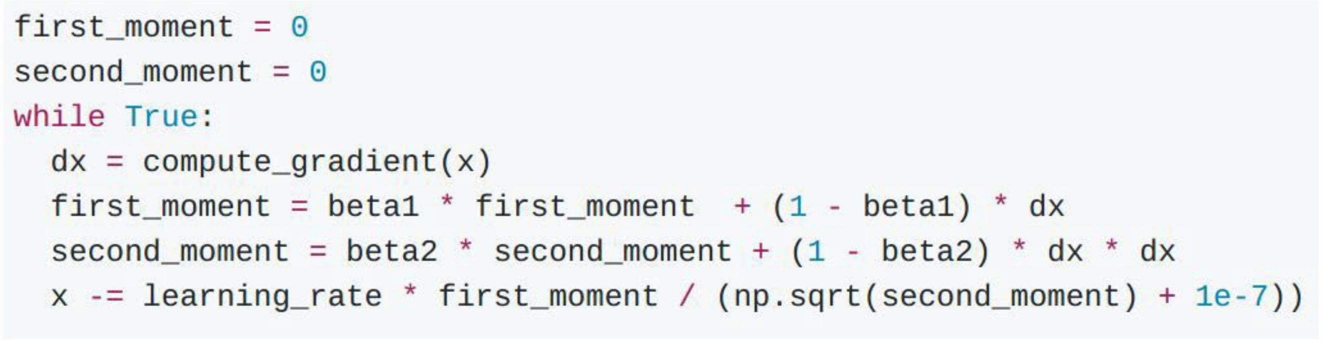


▬ SGD

🟦 SGD+Momentum

🟥 RMSProp

Adam (almost)



```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

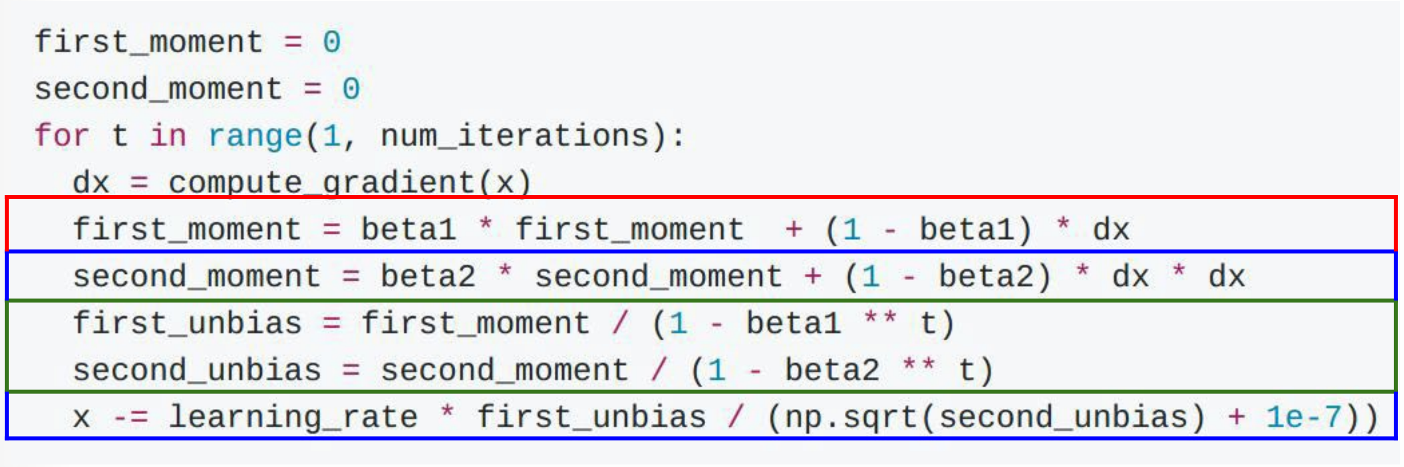
Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Adam (full form)



```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

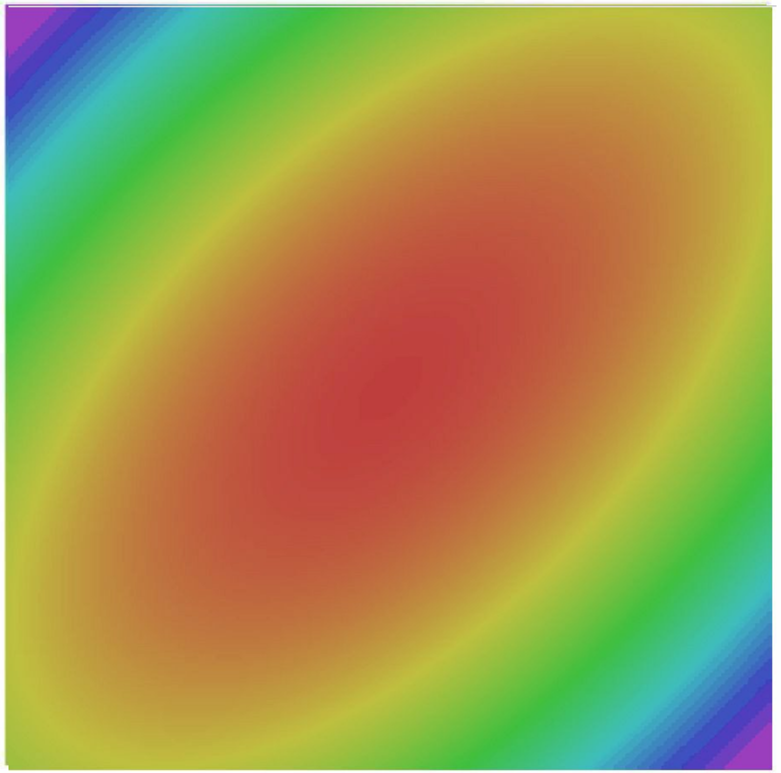
Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, “Adam: Amethod for stochastic optimization”, ICLR 2015

Slide Credit cs231n

Adam



▬ SGD

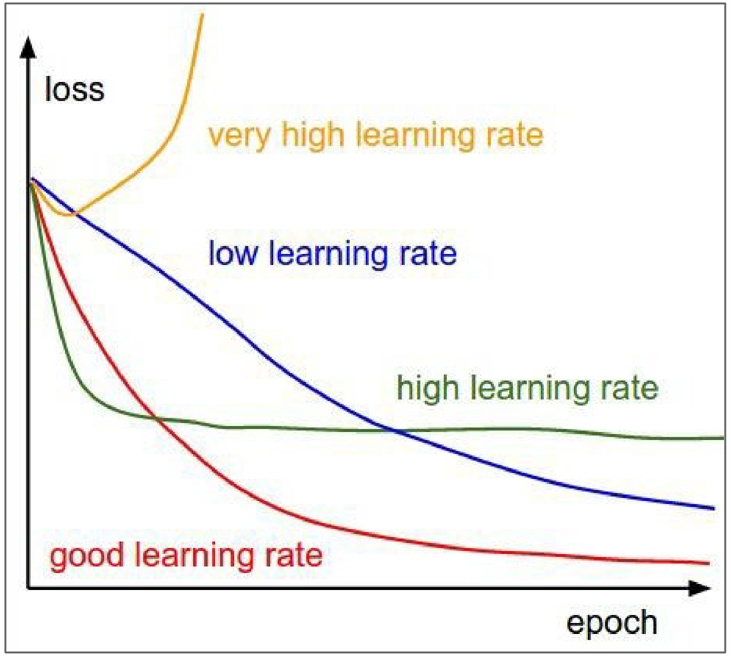
▬ SGD+Momentum

▬ RMSProp

▬ Adam

Slide Credit cs231n

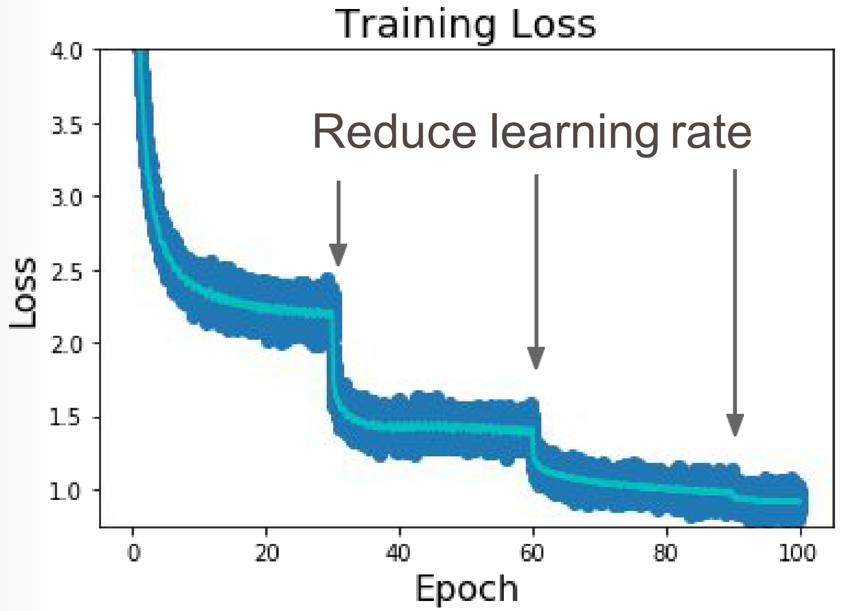
Learning Rate!



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

A: All of them! Start with large learning rate and decay over time

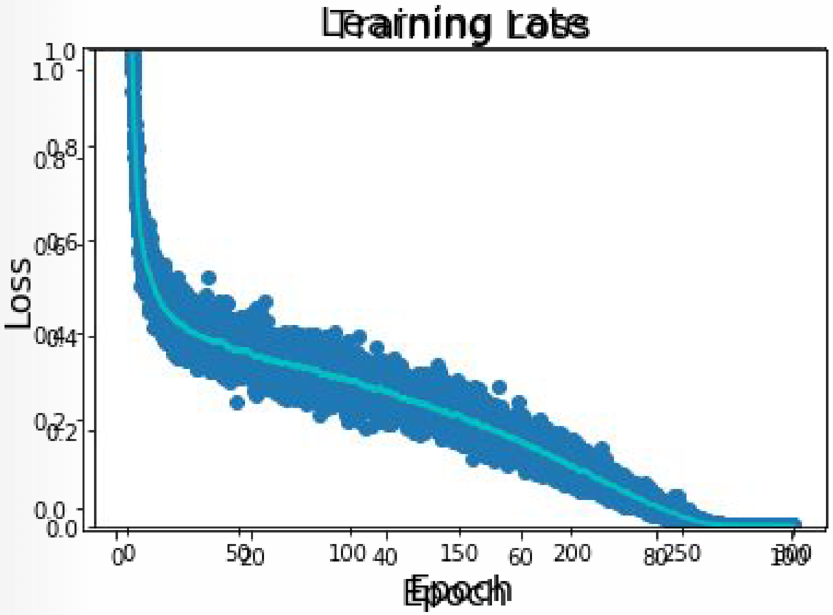
Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Slide Credit cs231n

Learning Rate Decay



Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

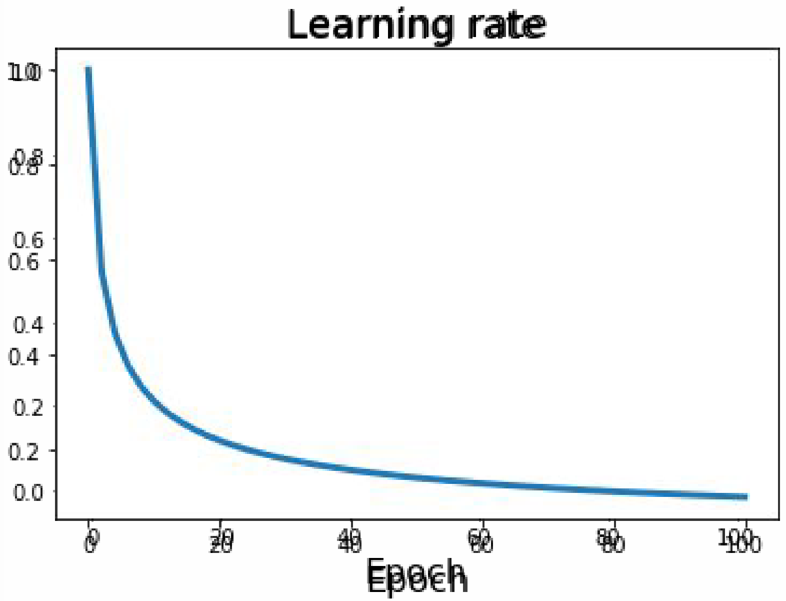
- **Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

- **Cosine**:

$$\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$

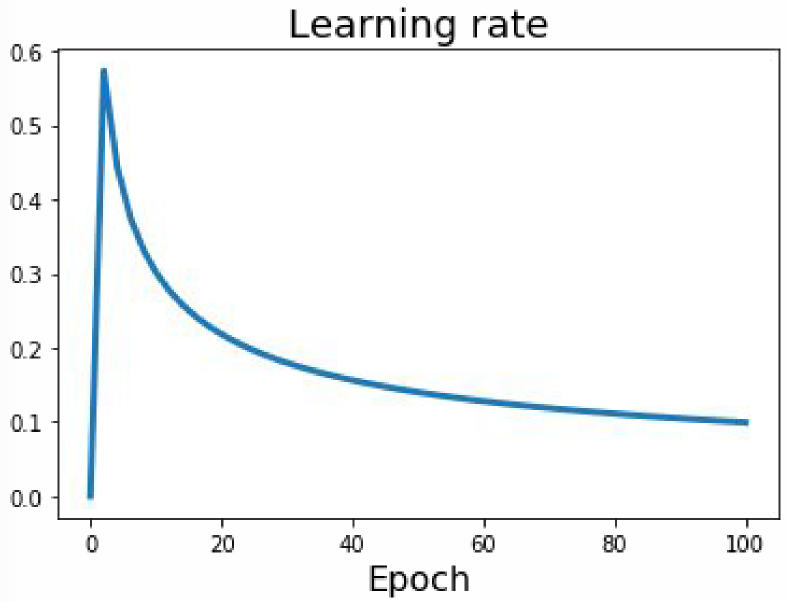
Linear: $\alpha_t = \alpha_0(1 - t/T)$

Inv sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Learning Rate Decay: Linear Warmup

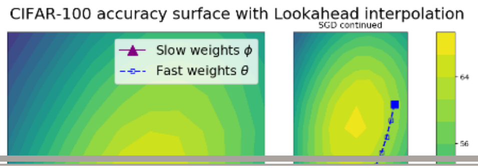


High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

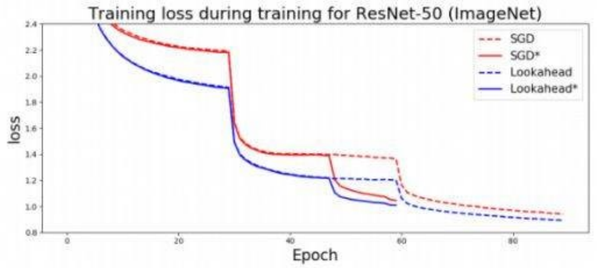
Lookahead Optimizer?



Algorithm 1 Lookahead Optimizer:

Require: Initial parameters ϕ_0, objective function L

Require: Synchronization period k, slow weights step



Optimizer	LA	SGD
Epoch 50 - Top 1	75.13	74.43
Epoch 50 - Top 5	92.22	92.15
Epoch 60 - Top 1	75.49	75.15
Epoch 60 - Top 5	92.53	92.56

Table 2: Top-1 and Top-5 single crop validation accuracies on ImageNet.

Figure 1: (Left) Visualizing Lookahead through a ResNet-32 test accuracy surface at epoch 100 on CIFAR-100. We project the weights onto a plane defined by the first, middle, and last fast (inner-loop) weights. The fast weights are along the blue dashed path. All points that lie on the plane are represented as solid, including the entire Lookahead slow weights path (in purple). Lookahead (middle, bottom right) quickly progresses closer to the minima than SGD (middle, top right) is able to. (Right) Pseudocode for Lookahead.

Zhang, Michael R., et al. "Lookahead Optimizer: k steps forward, 1 step back." (2019).

Now we have “advanced” ADAM

- R-ADAM (Rectified Adam)

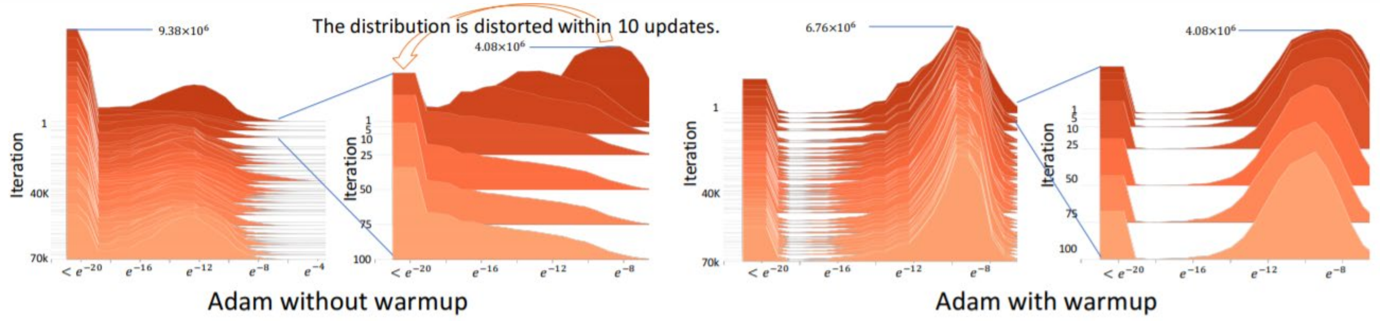
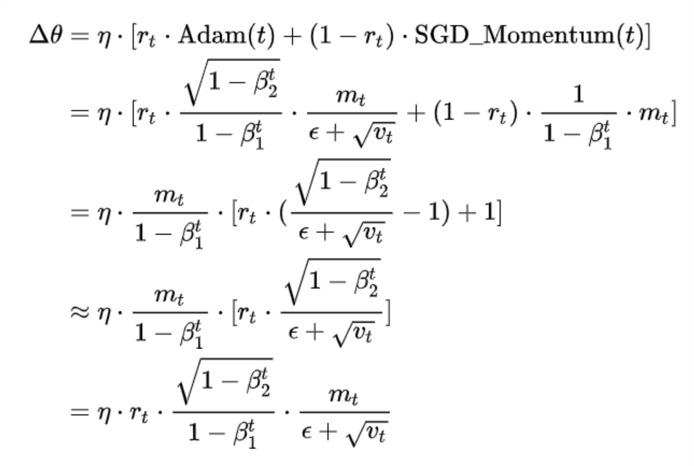


Figure 2: The absolute gradient histogram of the Transformers on the De-En IWSLT’ 14 dataset during the training (stacked along the y-axis). X-axis is absolute value in the log scale and the height is the frequency. Without warmup, the gradient distribution is distorted in the first 10 steps.

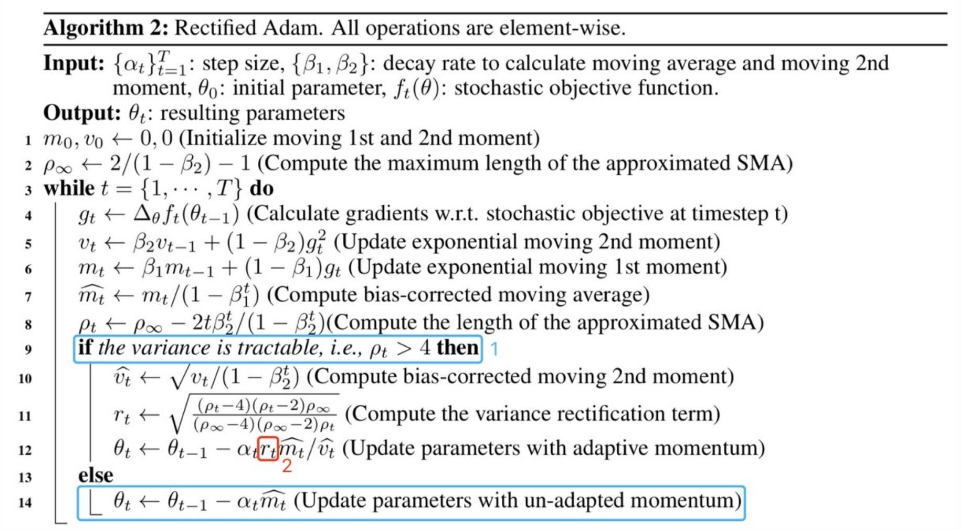
Liu, Liyuan, et al. "On the variance of the adaptive learning rate and beyond." *ICLR2020*.

RADAM

- SGD + Warmup



$$\begin{aligned}\Delta\theta &= \eta \cdot [r_t \cdot \mathrm{Adam}(t) + (1 - r_t) \cdot \mathrm{SGD\_Momentum}(t)] \\ &= \eta \cdot [r_t \cdot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \cdot \frac{m_t}{\epsilon + \sqrt{v_t}} + (1 - r_t) \cdot \frac{1}{1-\beta_1^t} \cdot m_t] \\ &= \eta \cdot \frac{m_t}{1-\beta_1^t} \cdot [r_t \cdot (\frac{\sqrt{1-\beta_2^t}}{\epsilon + \sqrt{v_t}} - 1) + 1] \\ &\approx \eta \cdot \frac{m_t}{1-\beta_1^t} \cdot [r_t \cdot \frac{\sqrt{1-\beta_2^t}}{\epsilon + \sqrt{v_t}}] \\ &= \eta \cdot r_t \cdot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \cdot \frac{m_t}{\epsilon + \sqrt{v_t}}\end{aligned}$$



Algorithm 2: Rectified Adam. All operations are element-wise.

Input: $\{\alpha_t\}_{t=1}^T$: step size, $\{\beta_1, \beta_2\}$: decay rate to calculate moving average and moving 2nd moment, θ_0: initial parameter, $f_t(\theta)$: stochastic objective function.

Output: θ_t: resulting parameters

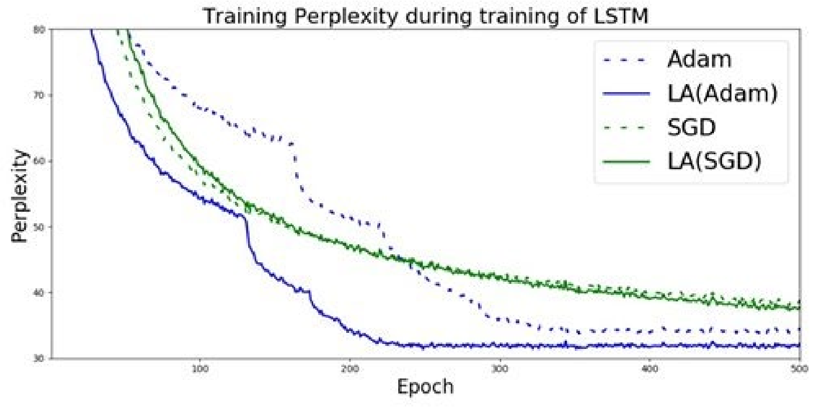
```
$m_0, v_0 \leftarrow 0, 0$ (Initialize moving 1st and 2nd moment)
$\rho_\infty \leftarrow 2/(1-\beta_2) - 1$ (Compute the maximum length of the approximated SMA)
while $t = \{1, \cdots, T\}$ do
    $g_t \leftarrow \Delta_\theta f_t(\theta_{t-1})$ (Calculate gradients w.r.t. stochastic objective at timestep t)
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ (Update exponential moving 2nd moment)
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ (Update exponential moving 1st moment)
    $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$ (Compute bias-corrected moving average)
    $\rho_t \leftarrow \rho_\infty - 2t\beta_2^t/(1-\beta_2^t)$(Compute the length of the approximated SMA)
    if the variance is tractable, i.e., $\rho_t > 4$ then 1
        $\widehat{v_t} \leftarrow \sqrt{v_t/(1-\beta_2^t)}$ (Compute bias-corrected moving 2nd moment)
        $r_t \leftarrow \sqrt{\frac{(\rho_t-4)(\rho_t-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho_t}}$ (Compute the variance rectification term)
        $\theta_t \leftarrow \theta_{t-1} - \alpha_t \underset{2}{\boxed{r_t}} \widehat{m_t}/\widehat{v_t}$ (Update parameters with adaptive momentum)
    else
        $\theta_t \leftarrow \theta_{t-1} - \alpha_t \widehat{m_t}$ (Update parameters with un-adapted momentum)
```

Slide Credit cs231n

Combination (2020 late)

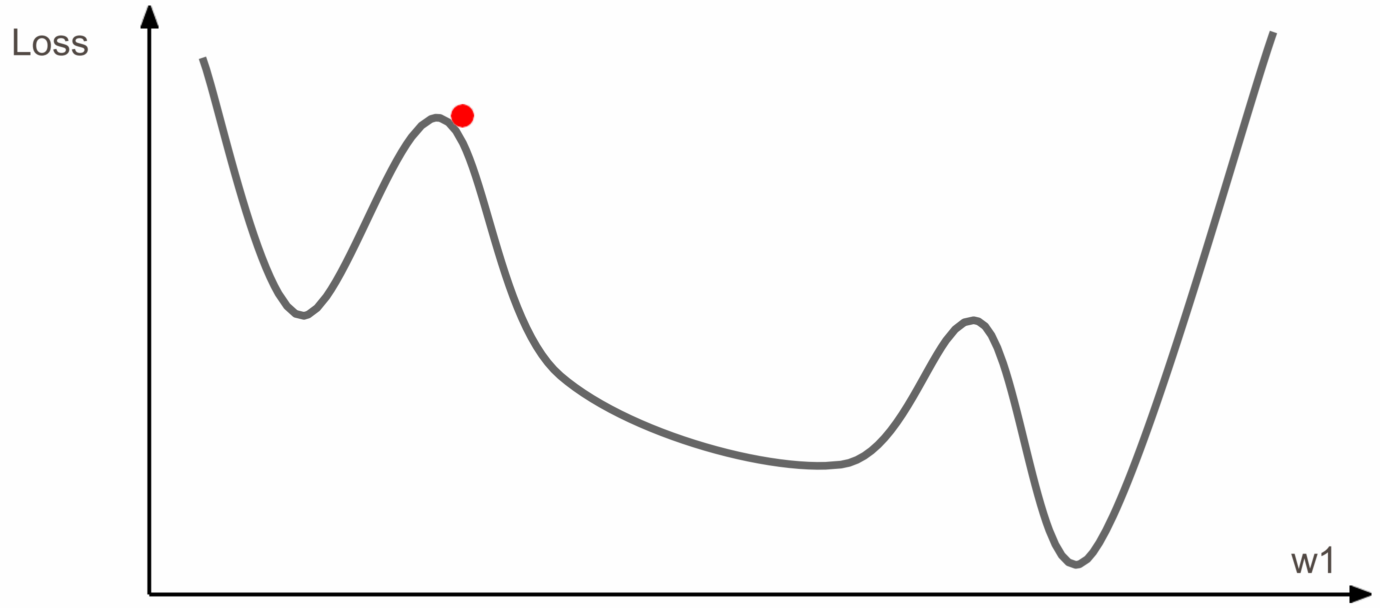
- The strongest optimizer + lookahead?
  - Yes, we have RANGER

```
class Ranger(Optimizer):

    def __init__(self, params, lr=1e-3,                       # lr
                 alpha=0.5, k=6, N_sma_threshhold=5,           # Ranger Lookahead options
                 betas=(.95,0.999), eps=1e-5, weight_decay=0,  # Adam options
                 use_gc=True,                                  # Gradient centralization on or off,
                 gc_conv_only=False                            # applied to conv layers only or conv + fc layers

                 ):
```



INSIGNIFICANT IMPROVEMENT?

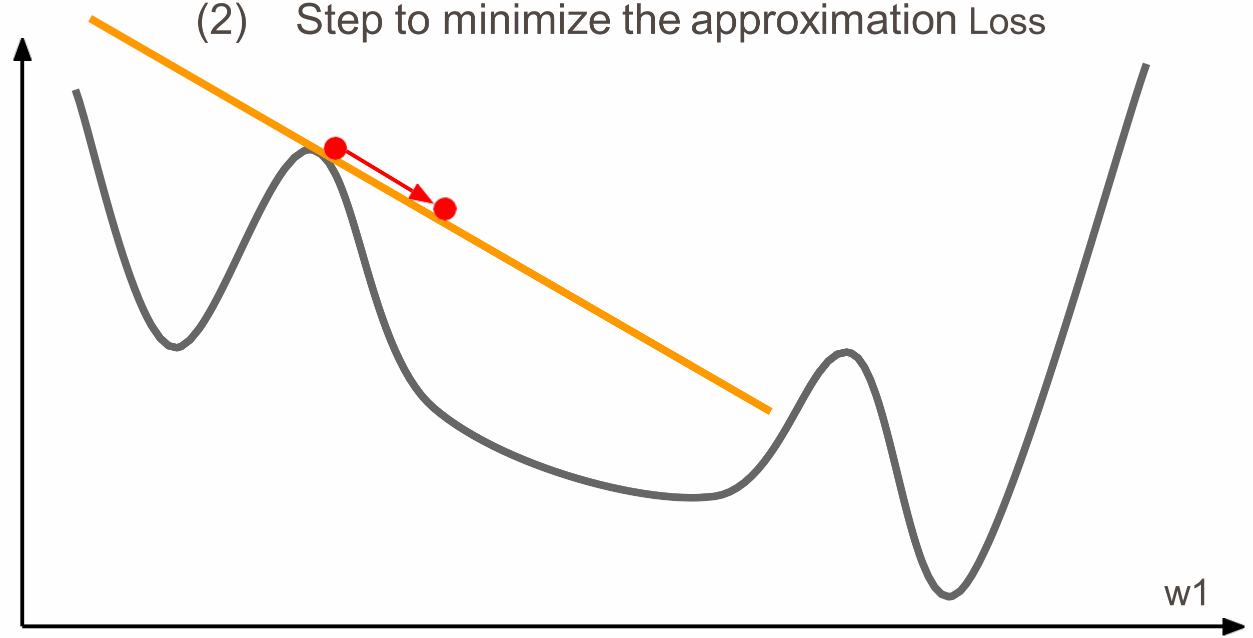
First-Order Optimization



Slide Credit cs231n

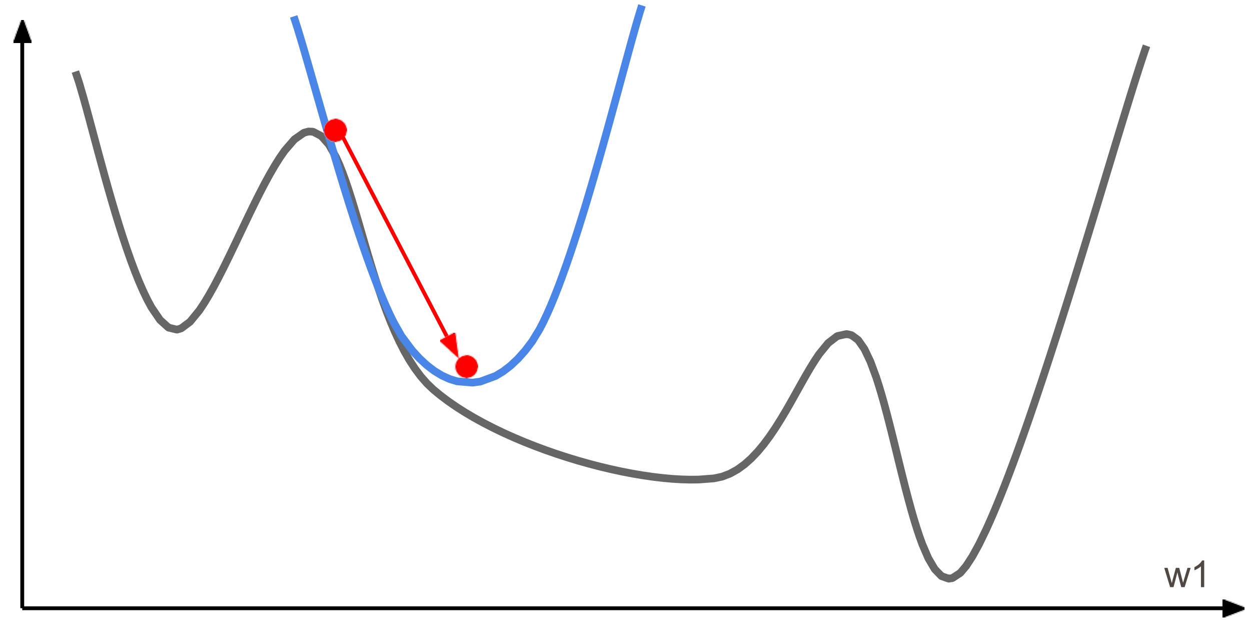
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation Loss



Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation Loss



Second-Order Optimization

second-order Taylor expansion:

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

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

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

Why is this bad for deep learning?

Hessian has $O(N^2)$ elements.
Inverting takes $O(N^3)$
N = (Tens or Hundreds of) Millions

Slide Credit cs231n

Second-Order Optimization

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

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

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

L-BFGS

- Usually works very well in full batch, deterministic mode
- i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- Does not transfer very well to mini-batch setting.
  - Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”
Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017

In practice:

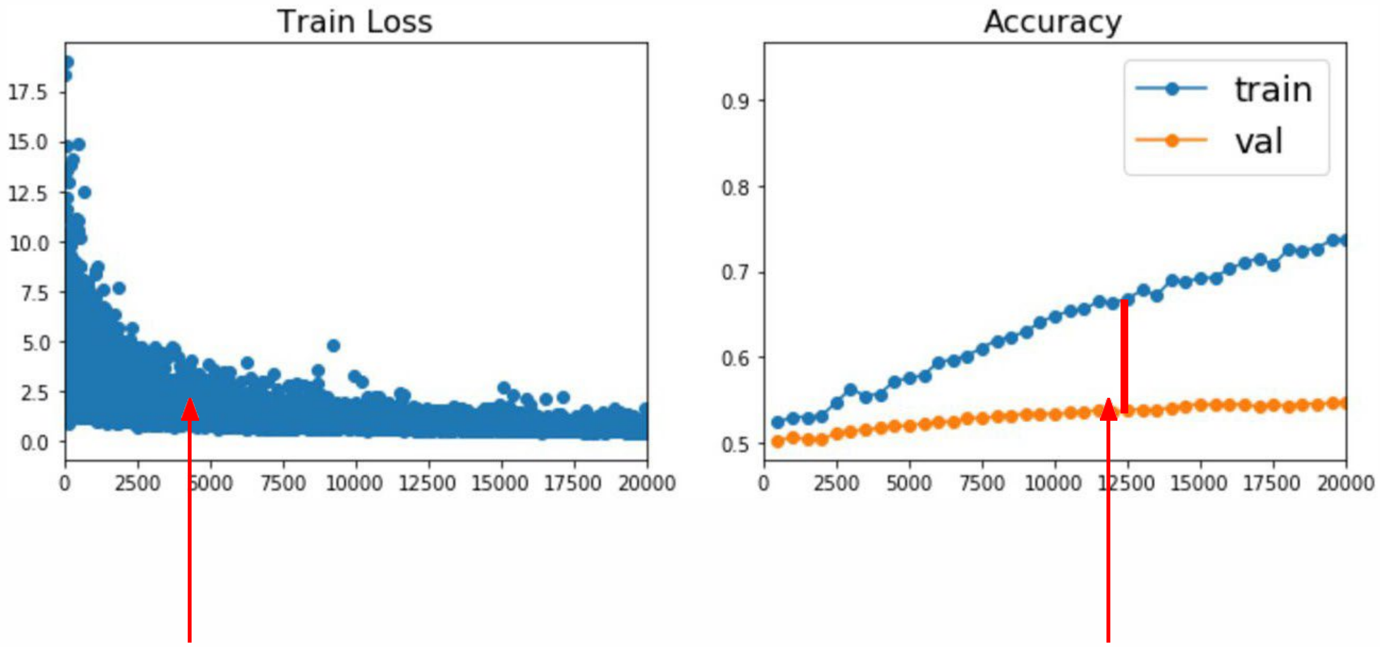
- Adam is a good default choice in many cases; it often works ok even with constant learning rate

- BUT in practice, SGD+Momentum can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

- Generative models: Adam-like optimizers still show great performance!!

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

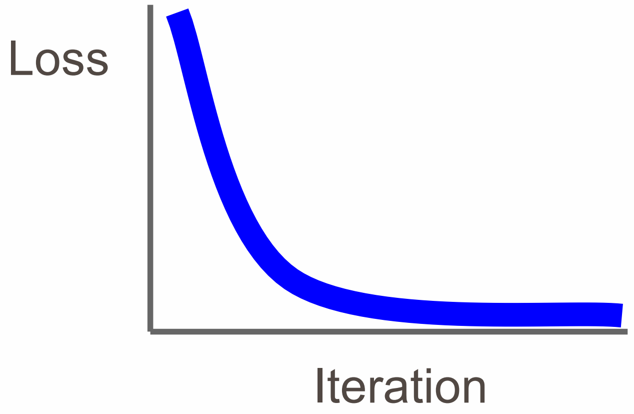
Beyond Training Error

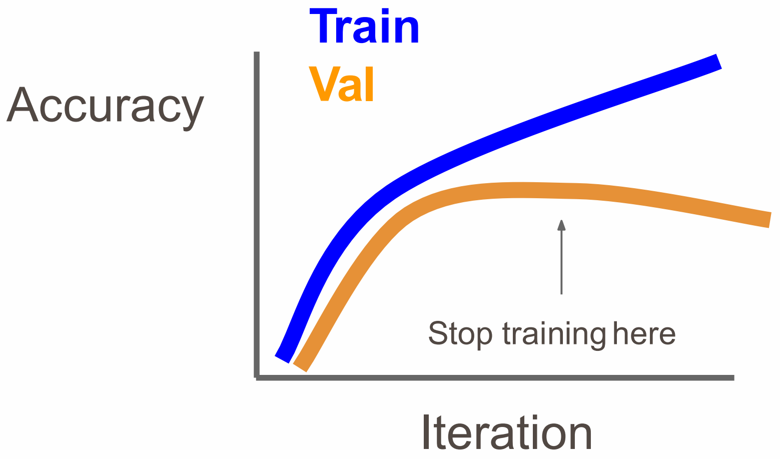


Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?

Early Stopping: Always do this





Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val

Model Ensembles

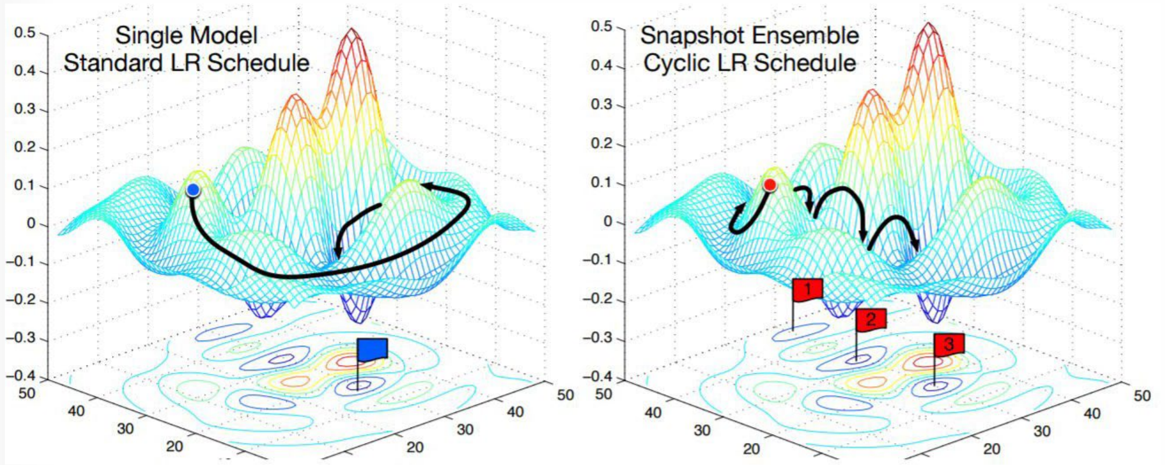
- Train multiple independent models
- At test time average their results
  - (Take average of predicted probability distributions, then choose argmax)

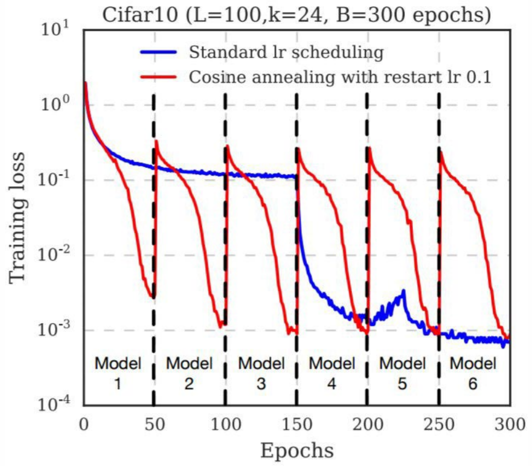
Enjoy 2% extra performance!!

Slide Credit cs231n

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



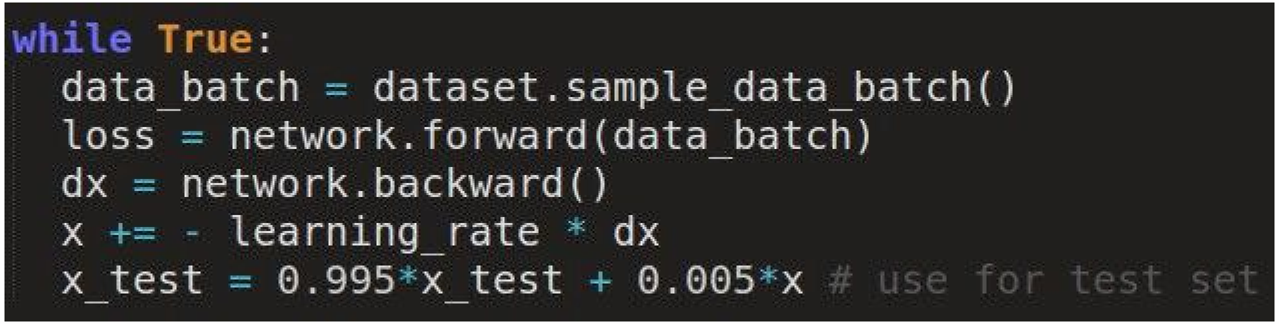


Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

Model Ensembles: Tips and Tricks

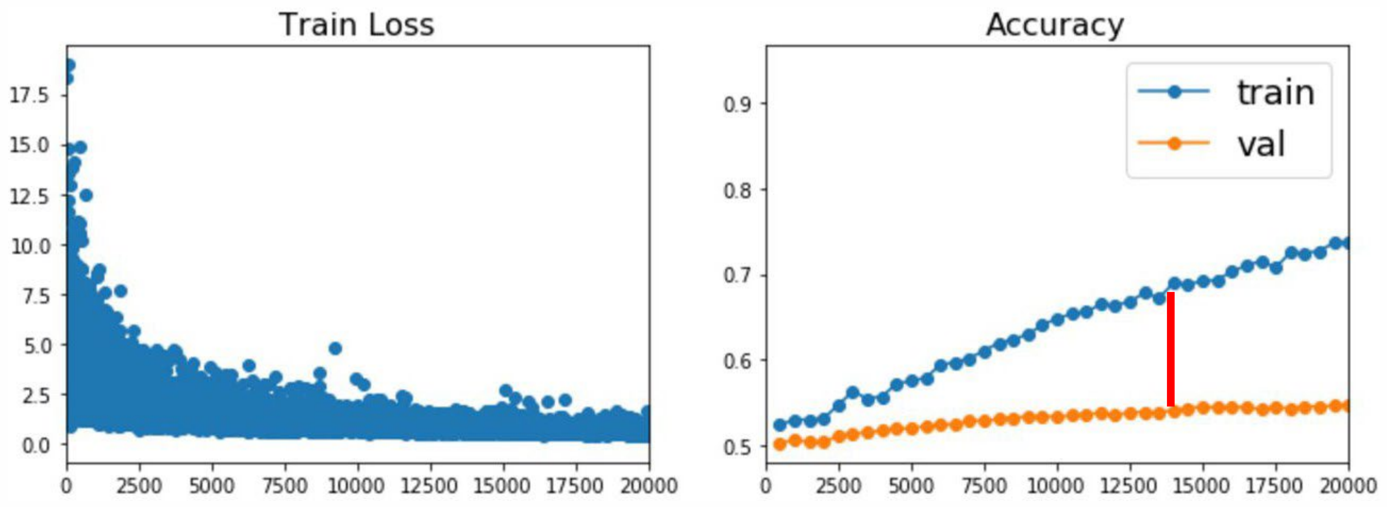
Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)
(or so-called network interpolation)



```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, “Acceleration of stochastic approximation by averaging”, SIAM Journal on Control and Optimization, 1992.

How to improve single-model performance?



Regularization

Slide Credit cs231n

Regularization: Add term to loss

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

In common use:

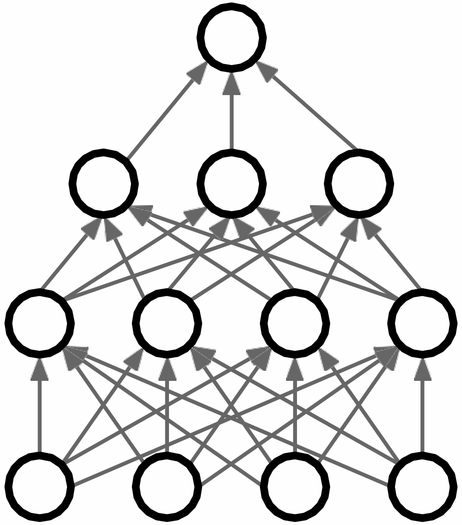
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

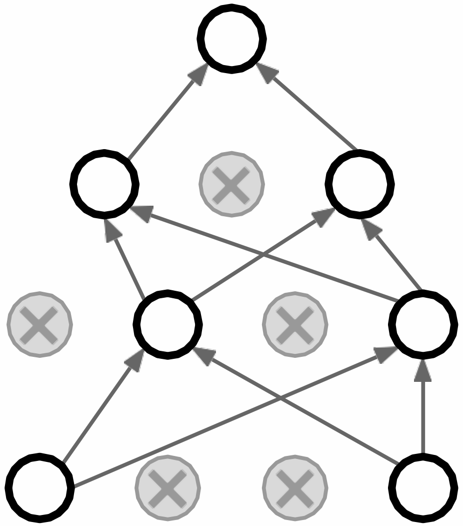
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

Regularization: Dropout

In each forward pass, randomly set some neurons to zero Probability of dropping is a hyperparameter; 0.5 is common



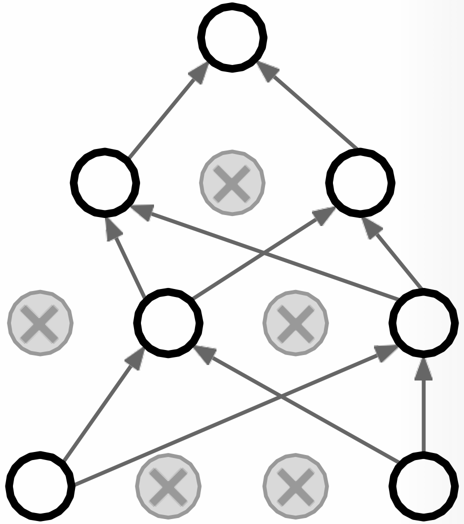


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: Dropout

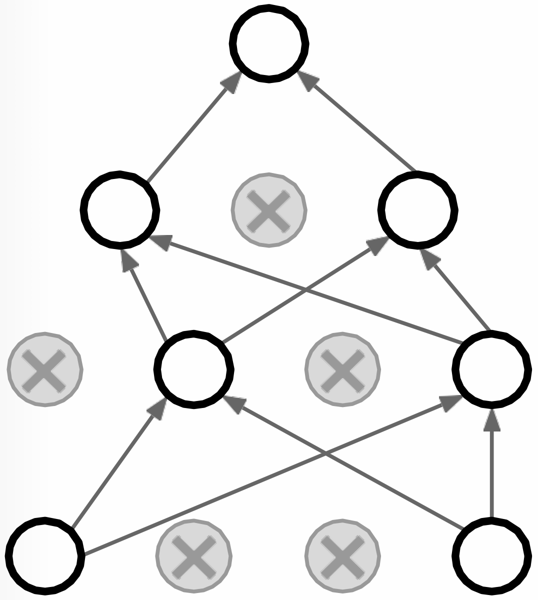
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

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

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

Example forward pass with a 3-layer network using dropout

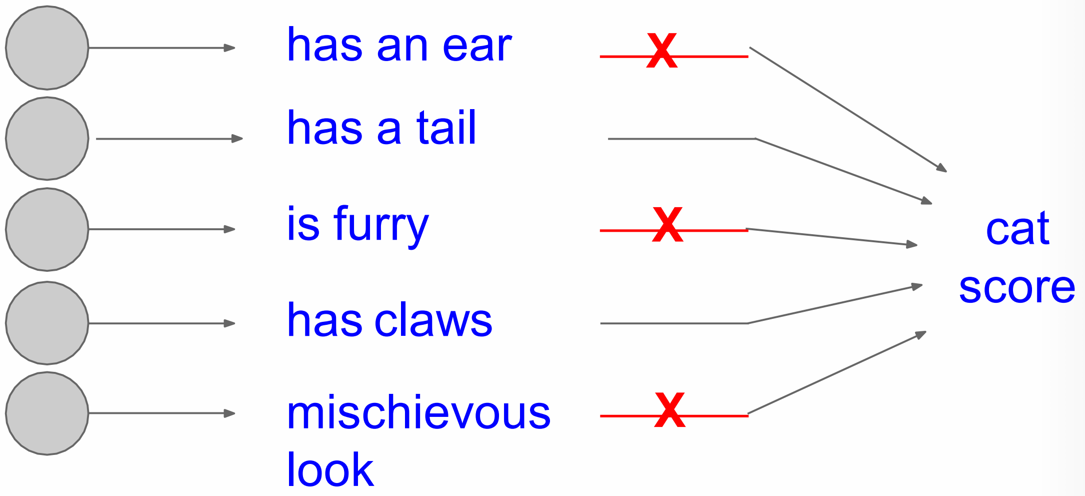


Slide Credit cs231n

Regularization: Dropout

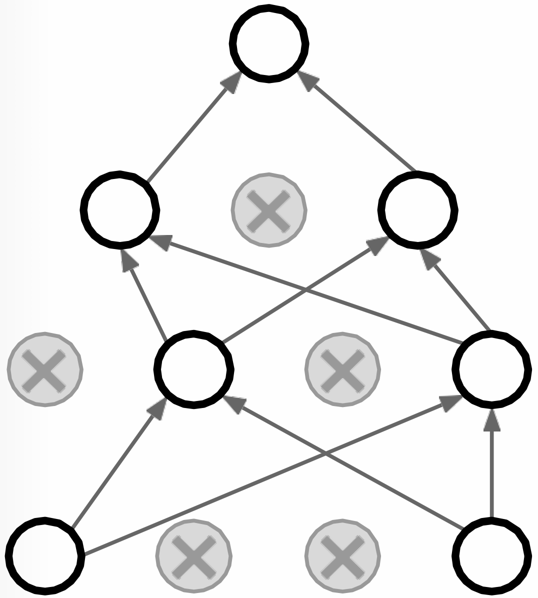


Forces the network to have a redundant representation;
Prevents co-adaptation of features



How can this possibly be a good idea?

Regularization: Dropout



Another interpretation:

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

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

Output (label)

Input (image)

$$\boxed{y} = f_W(\boxed{x}, \boxed{z})$$

Random mask

Want to "average out" the randomness at test-time

$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

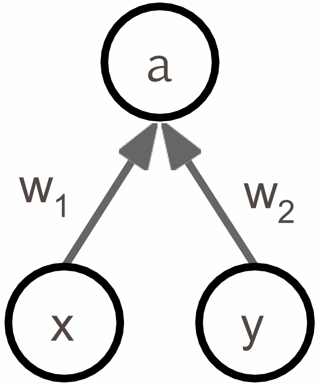
But this integral seems hard …

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z)\,dz$$

Consider a single neuron



At test time we have: $E\big[a\big] = w_1x + w_2y$

During training we have:
$$\begin{aligned} E\big[a\big] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, **multiply** by dropout probability

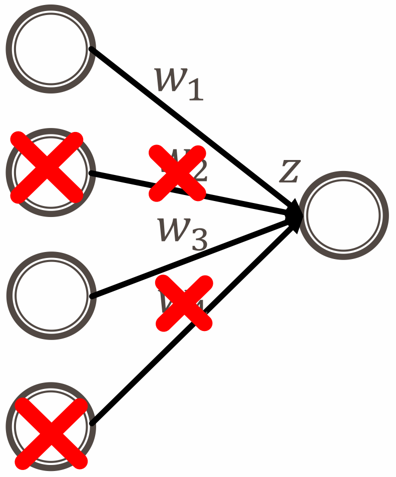
Slide Credit cs231n

Dropout

- Why the weights should multiply (1-p)% (dropout rate) when testing?

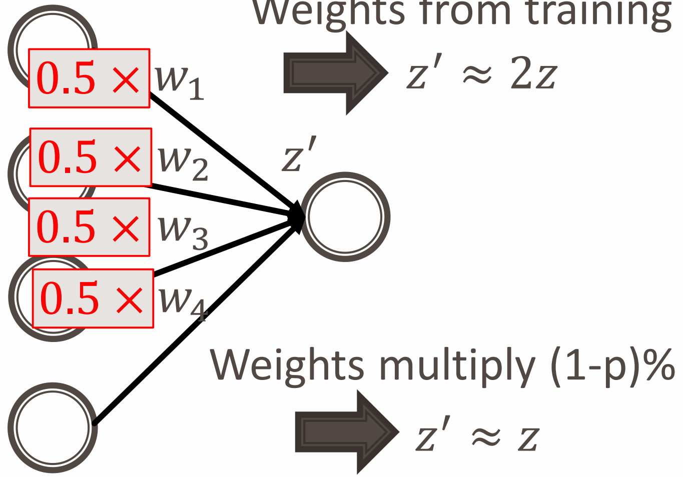
Training of Dropout

Assume dropout rate is 50%



Testing of Dropout

No dropout



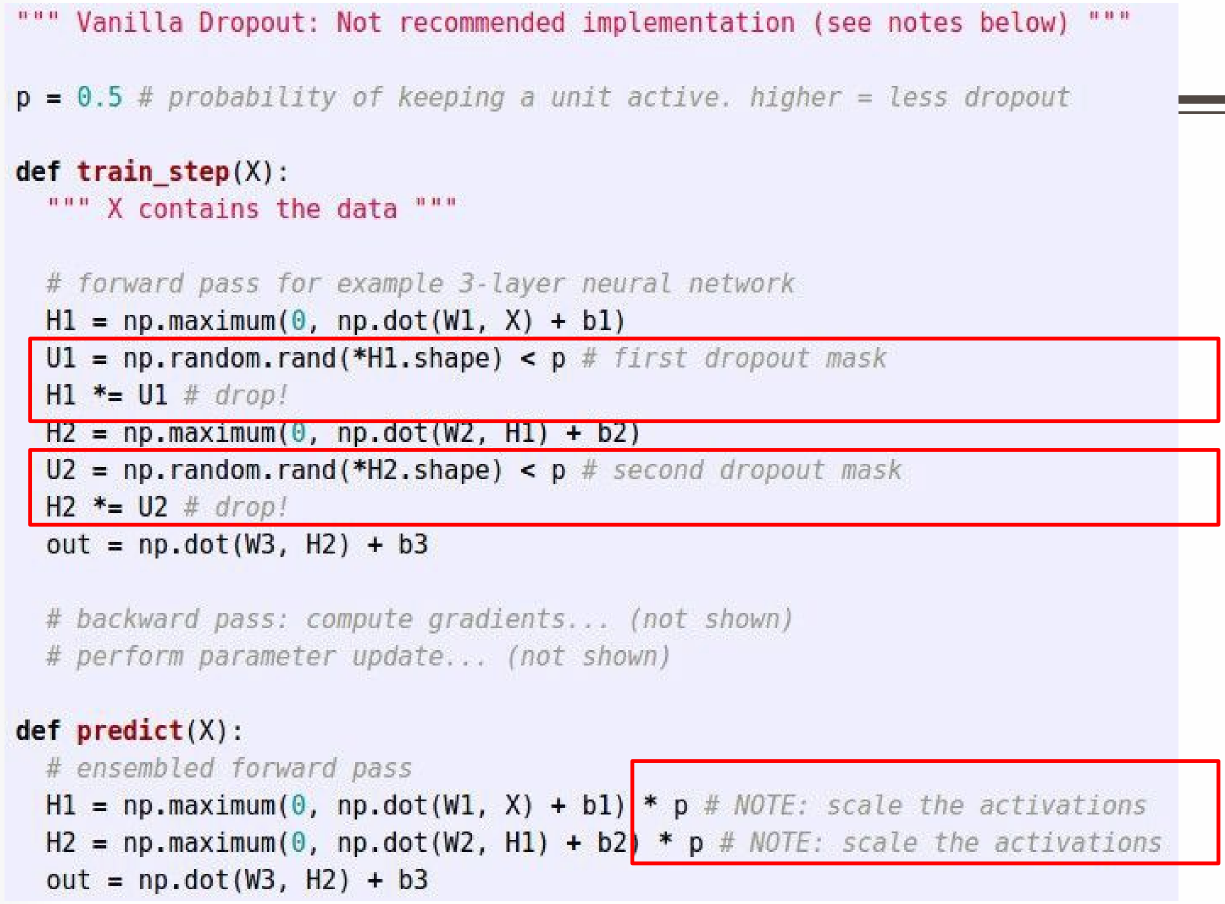
Dropout: Test time

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

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

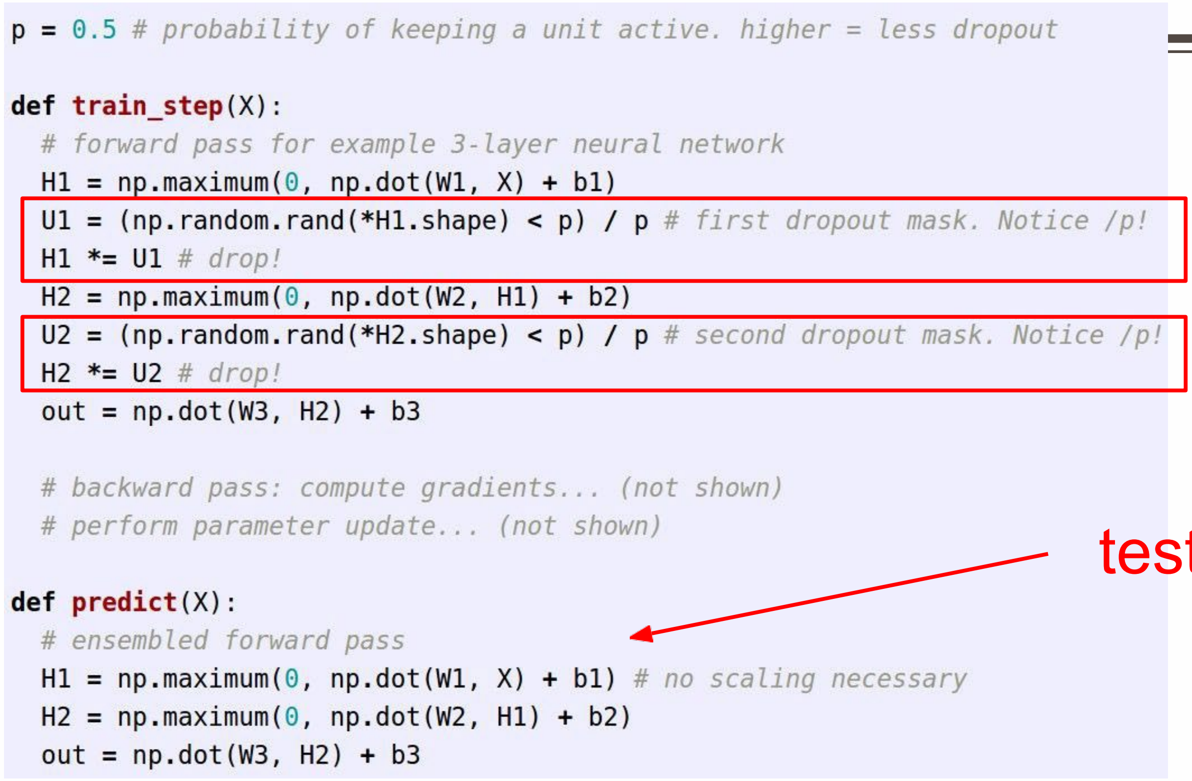


```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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

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

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

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

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

drop in forward pass

scale at test time

Slide Credit cs231n



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

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

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

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

More common: “Inverted dropout”

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\big[f(x, z)\big]$$
$$= \int p(z)f(x, z)dz$$

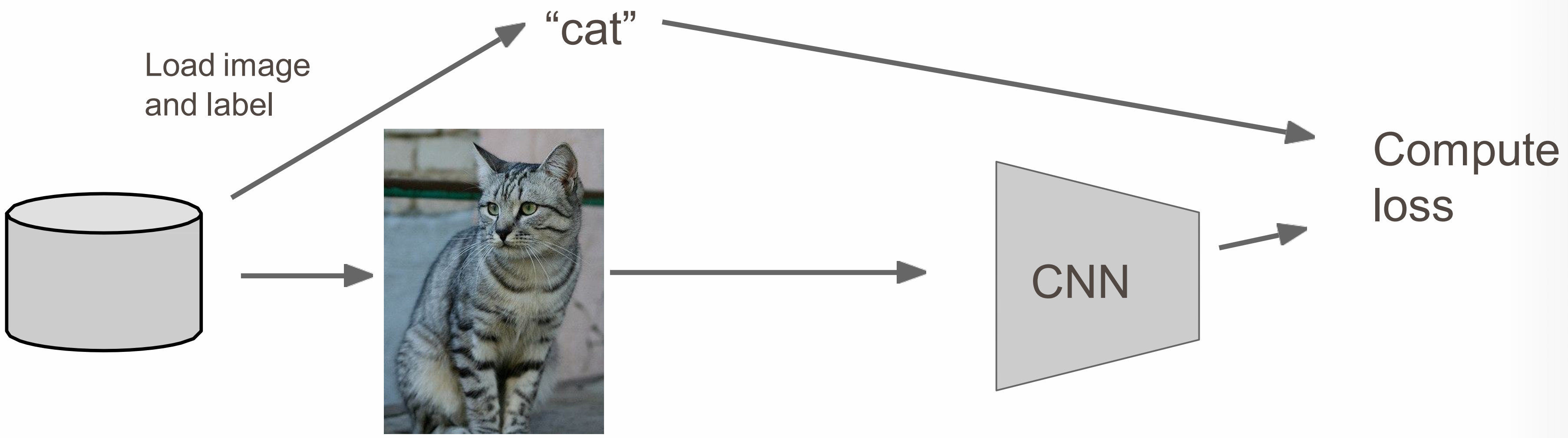
Example: Batch Normalization

Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Slide Credit cs231n

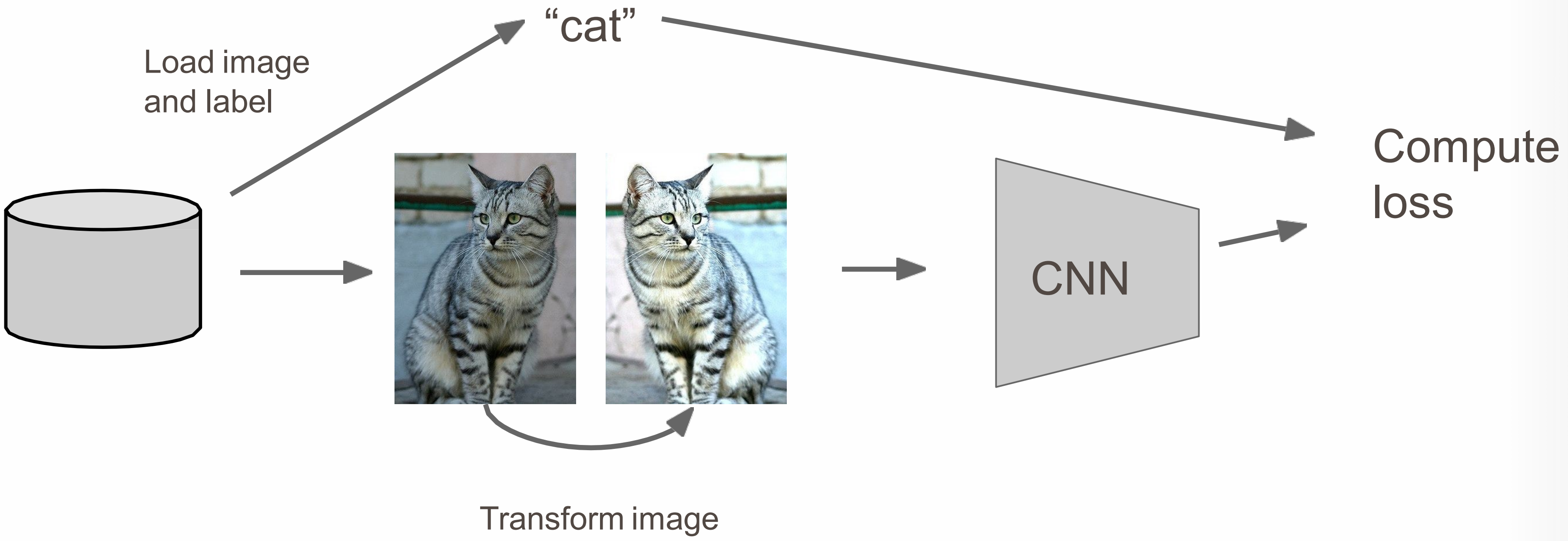
Regularization: Data Augmentation



This image by Nikita is licensed under CC-BY 2.0

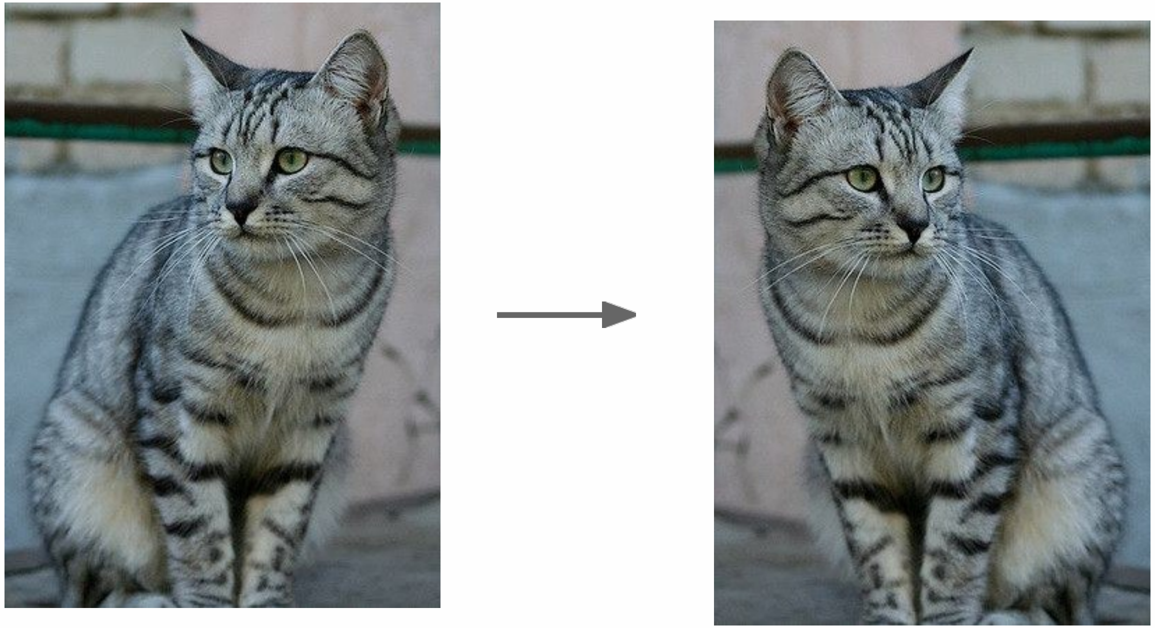
Slide Credit cs231n

Regularization: Data Augmentation



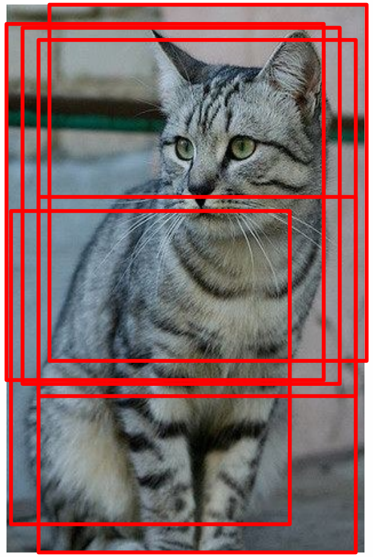
Slide Credit cs231n

Data Augmentation: Horizontal Flips



Data Augmentation

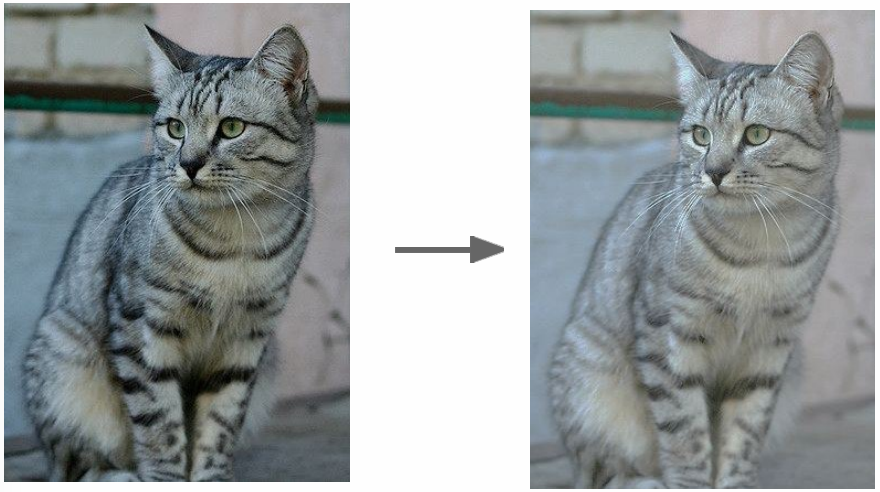
- Random crops and scales
- Training: sample random crops / scales
  - ResNet:
    - Pick random L in range [256, 480]
    - Resize training image, short side = L
    - Sample random 224 x 224 patch
- Testing: average a fixed set of crops
  - ResNet:
    - Resize image at 5 scales: {224, 256, 384, 480, 640}
    - For each size, use 10 224 x 224 crops: 4 corners + center, + flips



Slide Credit cs231n

Data Augmentation: Color Jitter

Simple: Randomize contrast and brightness



More Complex way:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

Slide Credit cs231n

Data Augmentation

- Get creative for your problem!
- Random mix/combinations of :
  - translation
  - rotation
  - stretching
  - shearing,
  - lens distortions, …

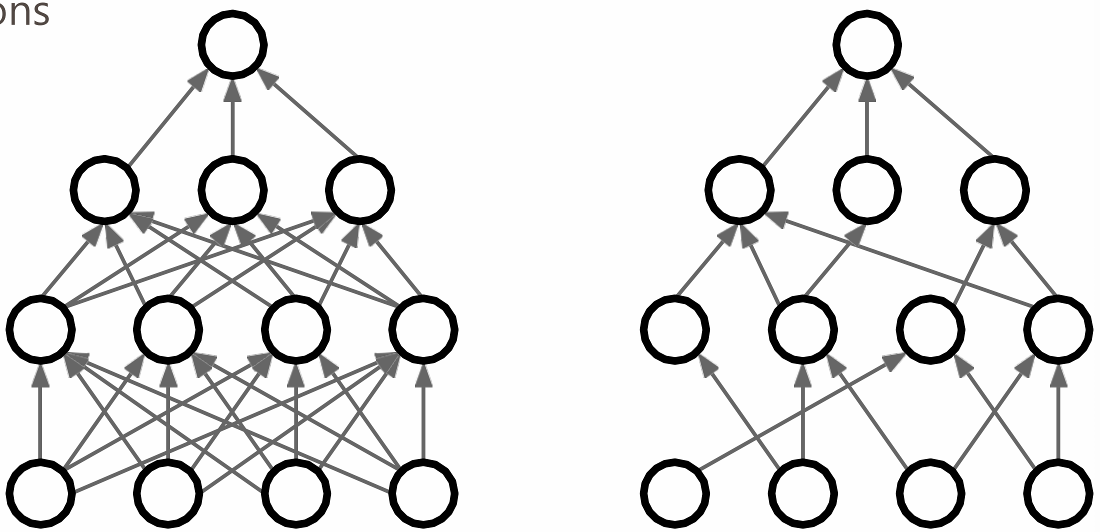
Slide Credit cs231n

Regularization: A common pattern

- Training: Add random noise
- Testing: Marginalize over the noise
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation

Regularization: DropConnect

- Training: Drop connections between neurons (set weights to 0)
- Testing: Use all the connections
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect

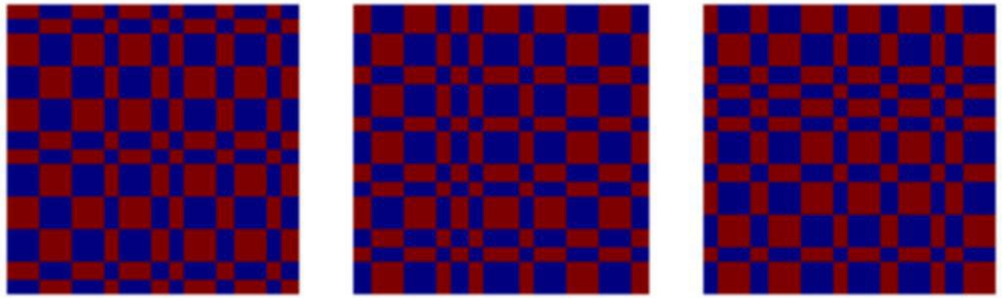


Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Slide Credit cs231n

Regularization: Fractional Pooling

- Training: Use randomized pooling regions
- Testing: Average predictions from several regions
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect
  - Fractional Max Pooling

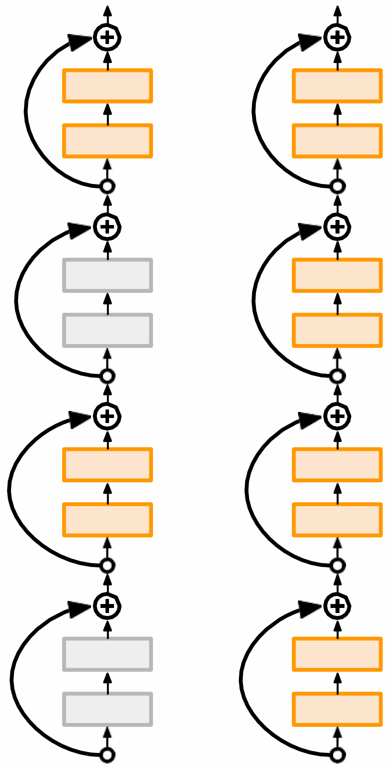


Graham, "Fractional Max Pooling", arXiv 2014

Slide Credit cs231n

Regularization: Stochastic Depth

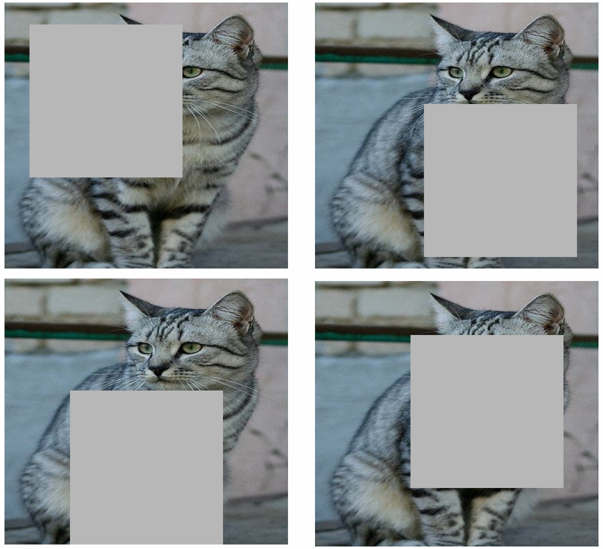
- Training: Skip some layers in the network
- Testing: Use all the layer
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect
  - Fractional Max Pooling
  - Stochastic Depth



Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

Regularization: Cutout

- Training: Set random image regions to zero
- Testing: Use full image
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect
  - Fractional Max Pooling
  - Stochastic Depth
  - Cutout



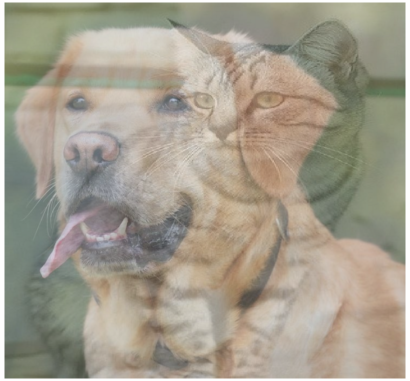
Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017

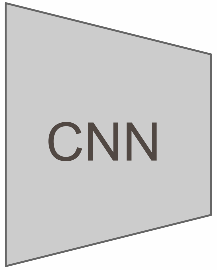
Slide Credit cs231n

Regularization: Mixup

- Training: Train on random blends of images
- Testing: Use original images
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect
  - Fractional Max Pooling
  - Stochastic Depth
  - Cutout
  - Mixup



Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

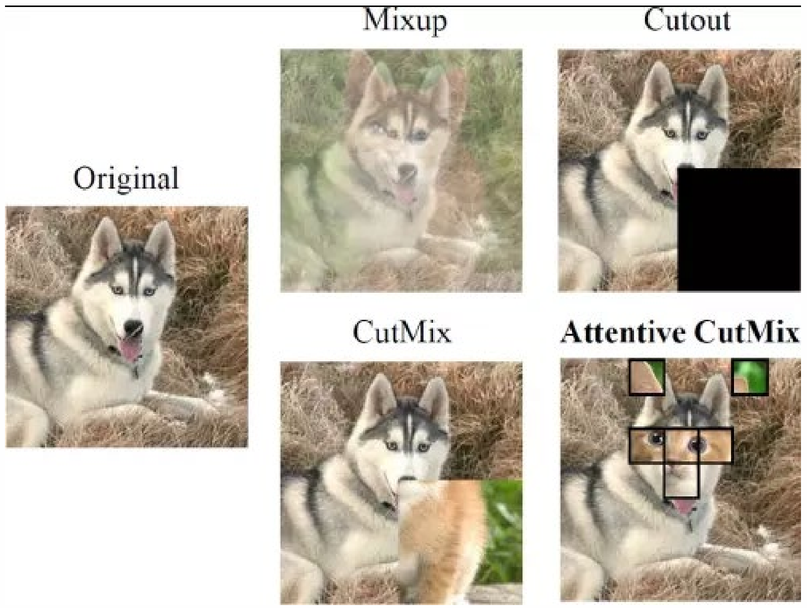


Target label:
cat: 0.4
dog: 0.6

Zhang et al, “*mixup*: Beyond Empirical Risk Minimization”, ICLR 2018

Regularization: Mixup

- Training: Train on random blends of images
- Testing: Use original images
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect
  - Fractional Max Pooling
  - Stochastic Depth
  - Cutout
  - Mixup
  - CutMix



Yun, Sangdoo, et al. "Cutmix: Regularization strategy to train strong classifiers with localizable features." *ICCV 2019*

Regularization

- Training: Add random noise
- Testing: Marginalize over the noise
- Examples:
  - Dropout
  - Batch Normalization
  - Data Augmentation
  - DropConnect
  - Fractional Max Pooling
  - Stochastic Depth
  - Cutout
  - Mixup
  - CutMix

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout and mixup especially for small classification datasets

Slide Credit cs231n

CHOOSING HYPERPARAMETERS

(without tons of GPUs)

Choosing Hyperparameters

- Step 1: Check initial loss
- Turn off weight decay, sanity check loss at initialization
- e.g. log(C) for softmax with C classes

Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample

- Try to train to 100% training accuracy on a small sample of training data
  - #minibatches < 10
  - Tuning your architecture, learning rate, weight initialization

- Loss not going down? LR too low, bad initialization
- Loss explodes to Inf or NaN? LR too high, bad initialization

Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find LR that makes loss go down

- Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

- Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Slide Credit cs231n

Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find LR that makes loss go down
- Step 4: Coarse grid, train for ~1-5 epochs

- Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

- Good weight decay to try: 1e-4, 1e-5, 1e-6, even 0

Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find LR that makes loss go down
- Step 4: Coarse grid, train for ~1-5 epochs
- Step 5: Refine grid, train longer

- Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

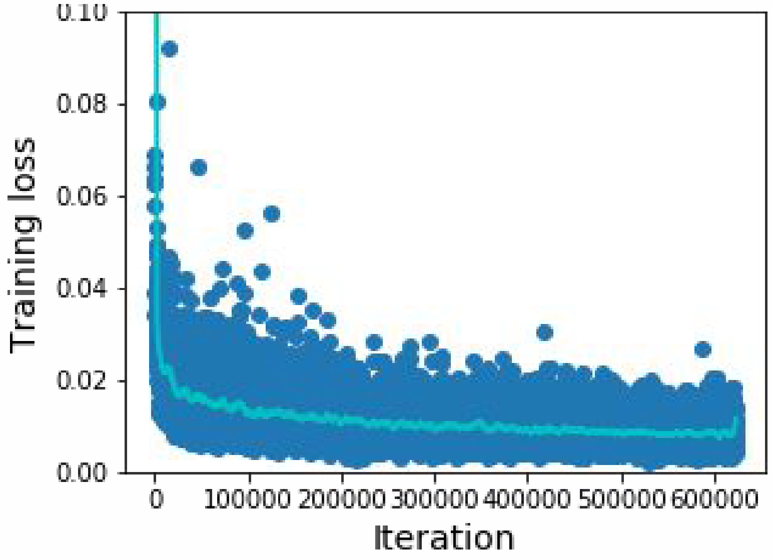
Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find LR that makes loss go down
- Step 4: Coarse grid, train for ~1-5 epochs
- Step 5: Refine grid, train longer
- Step 6: Look at loss curves

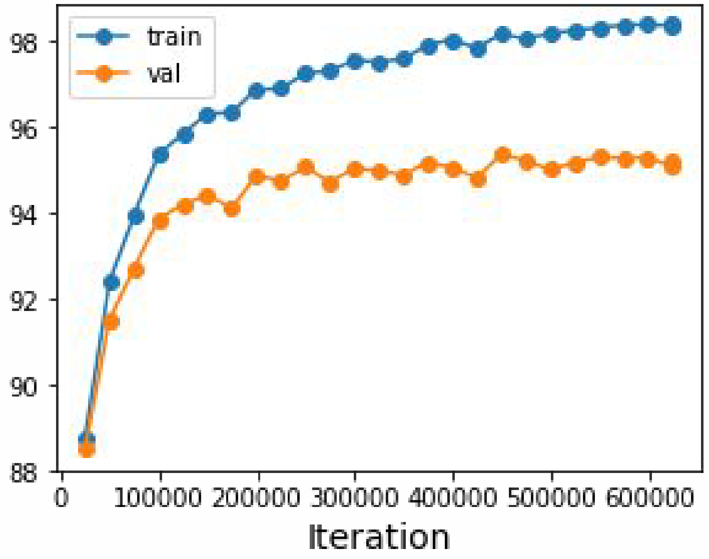
Slide Credit cs231n

Look at learning curves!

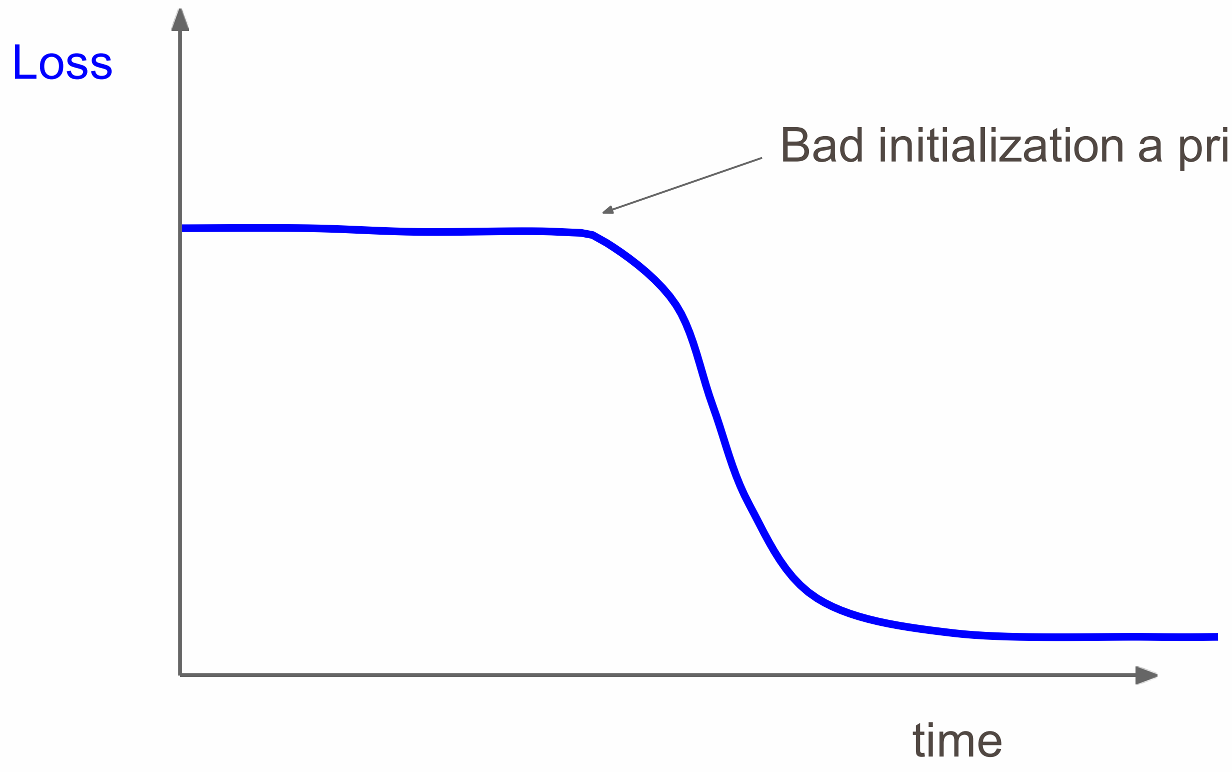
Training Loss

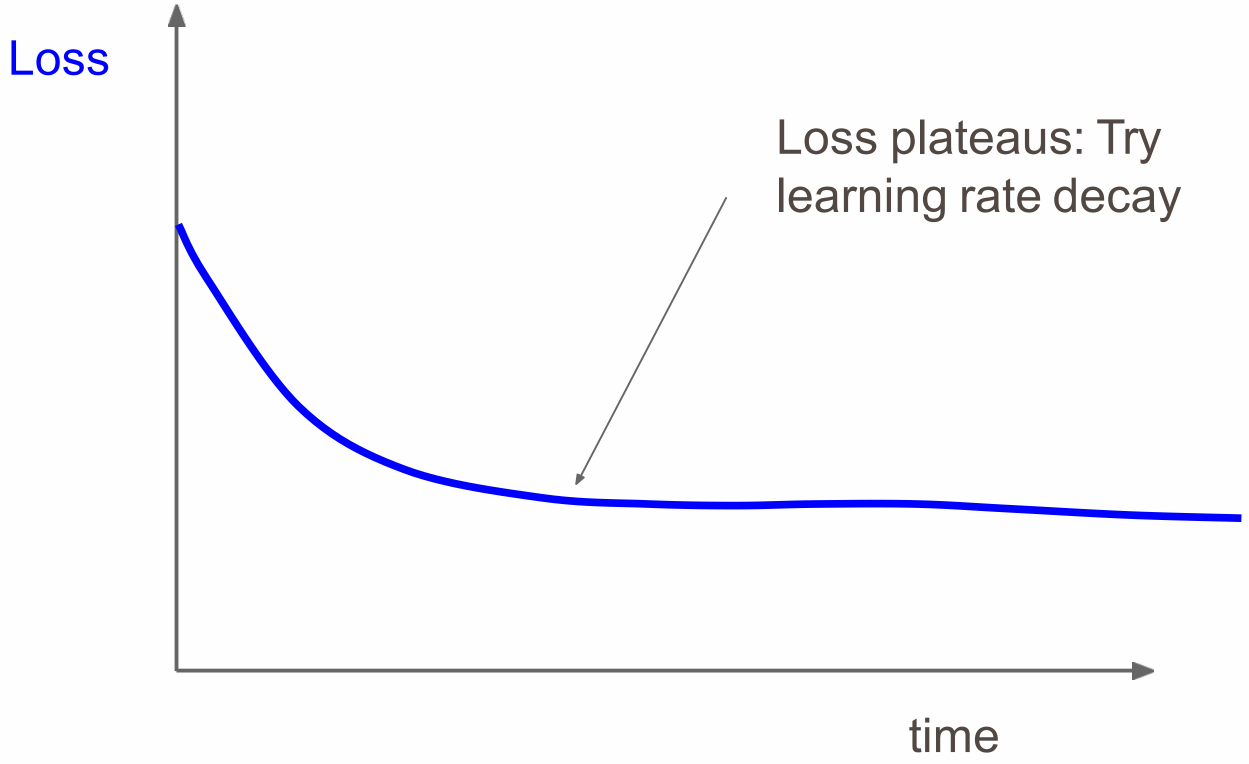


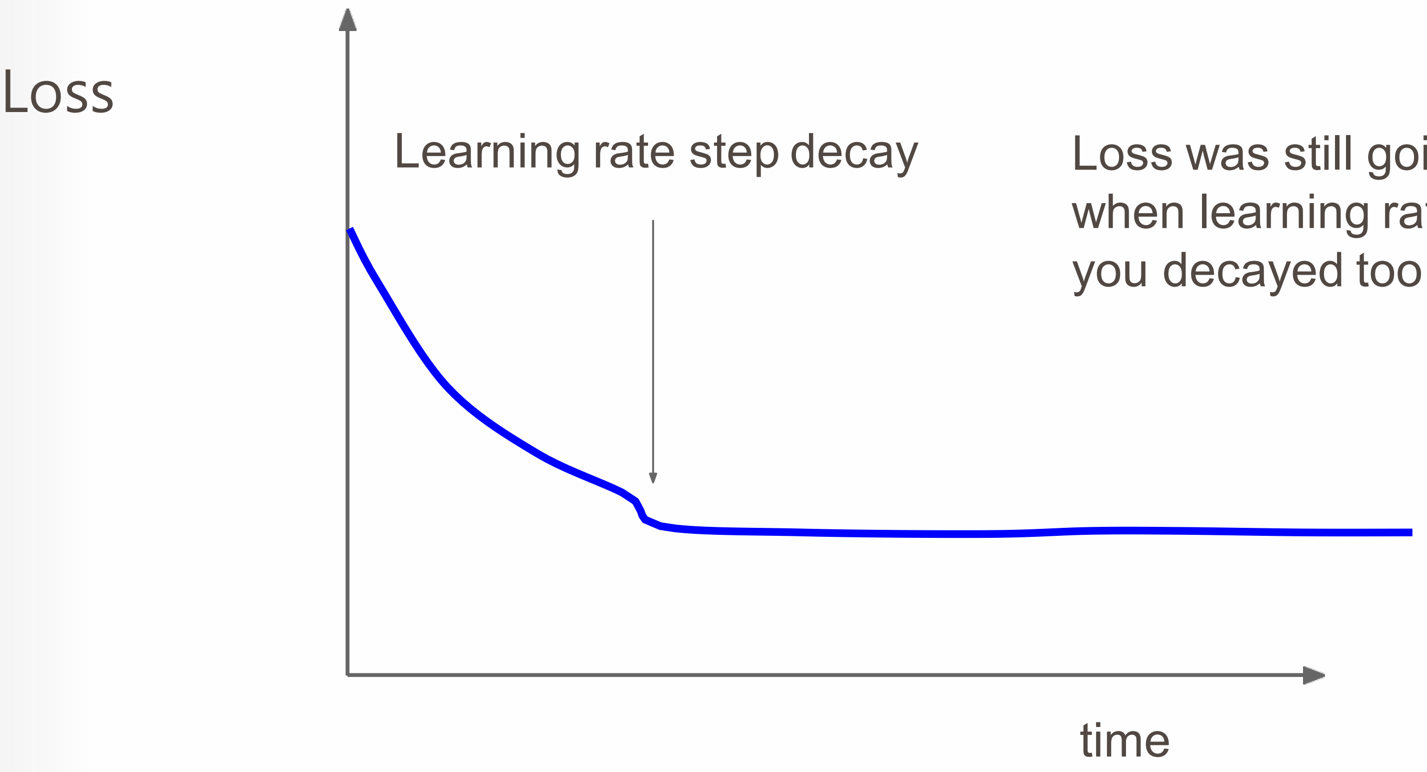
Train / Val Accuracy



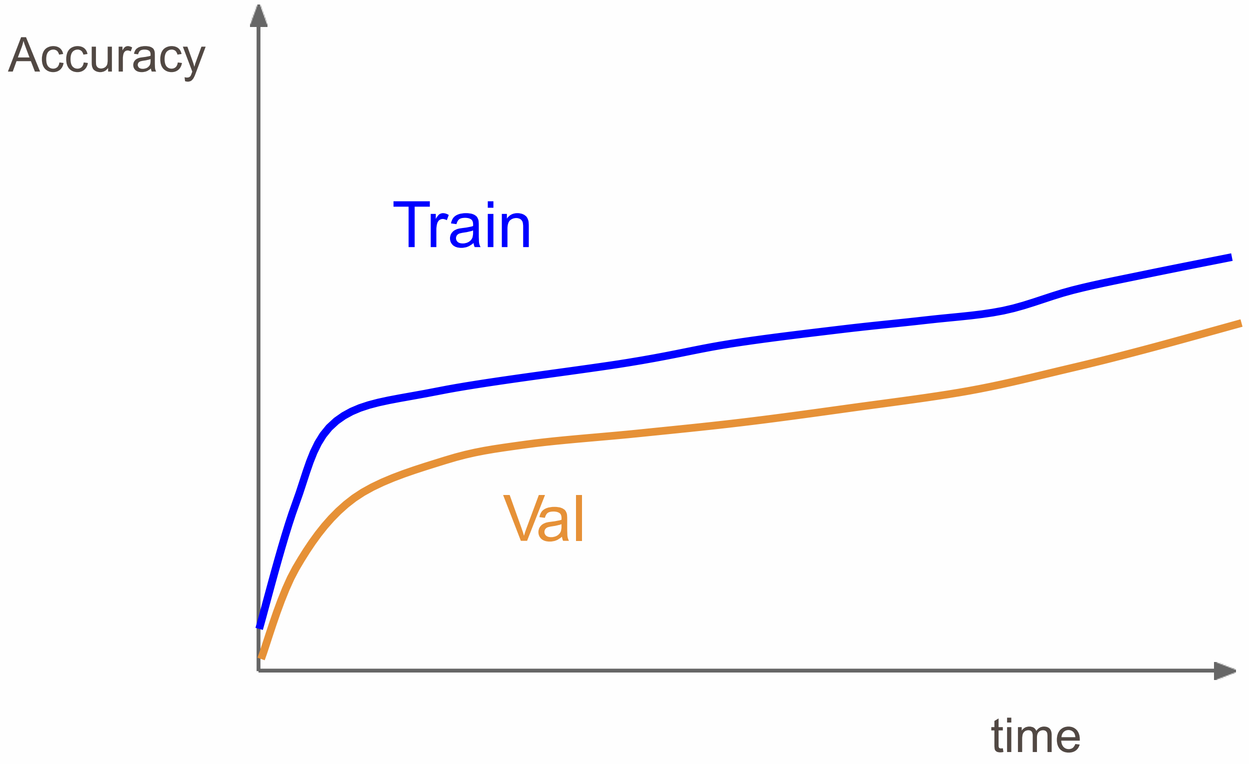
Losses may be noisy, use a scatter plot and also plot moving average to see trends better


Loss
Bad initialization a prime suspect
time

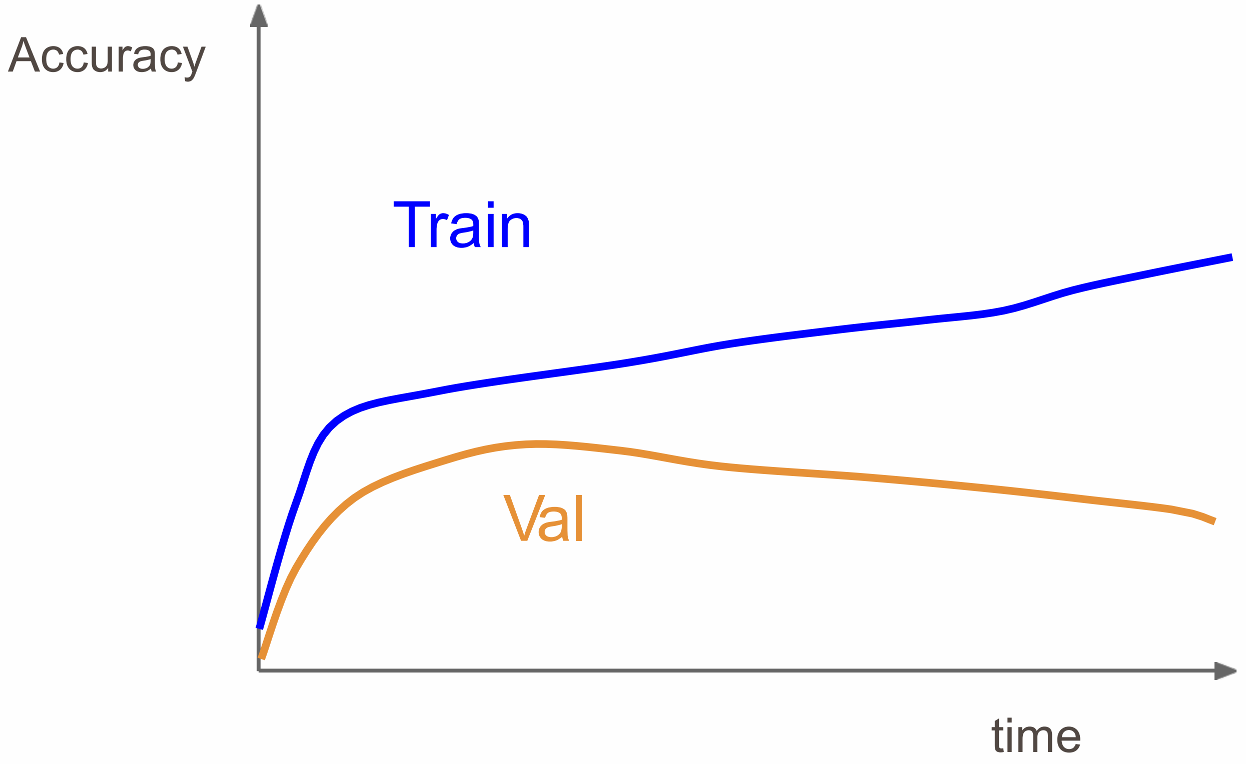

Loss
Loss plateaus: Try learning rate decay
time



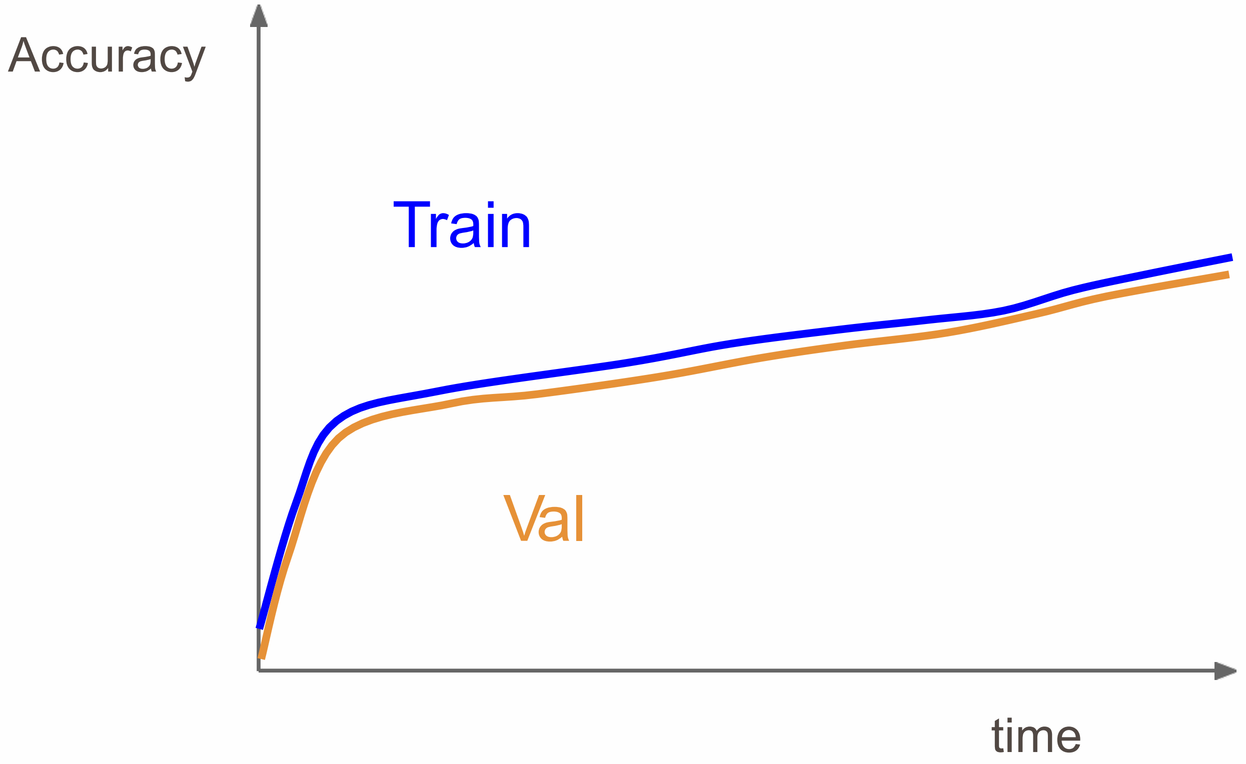
Loss was still going down when learning rate dropped, you decayed too early!



Accuracy still going up, you need to train longer



Huge train / val gap means overfitting! Increase regularization, get more data



No gap between train / val means underfitting: train longer, use a bigger model

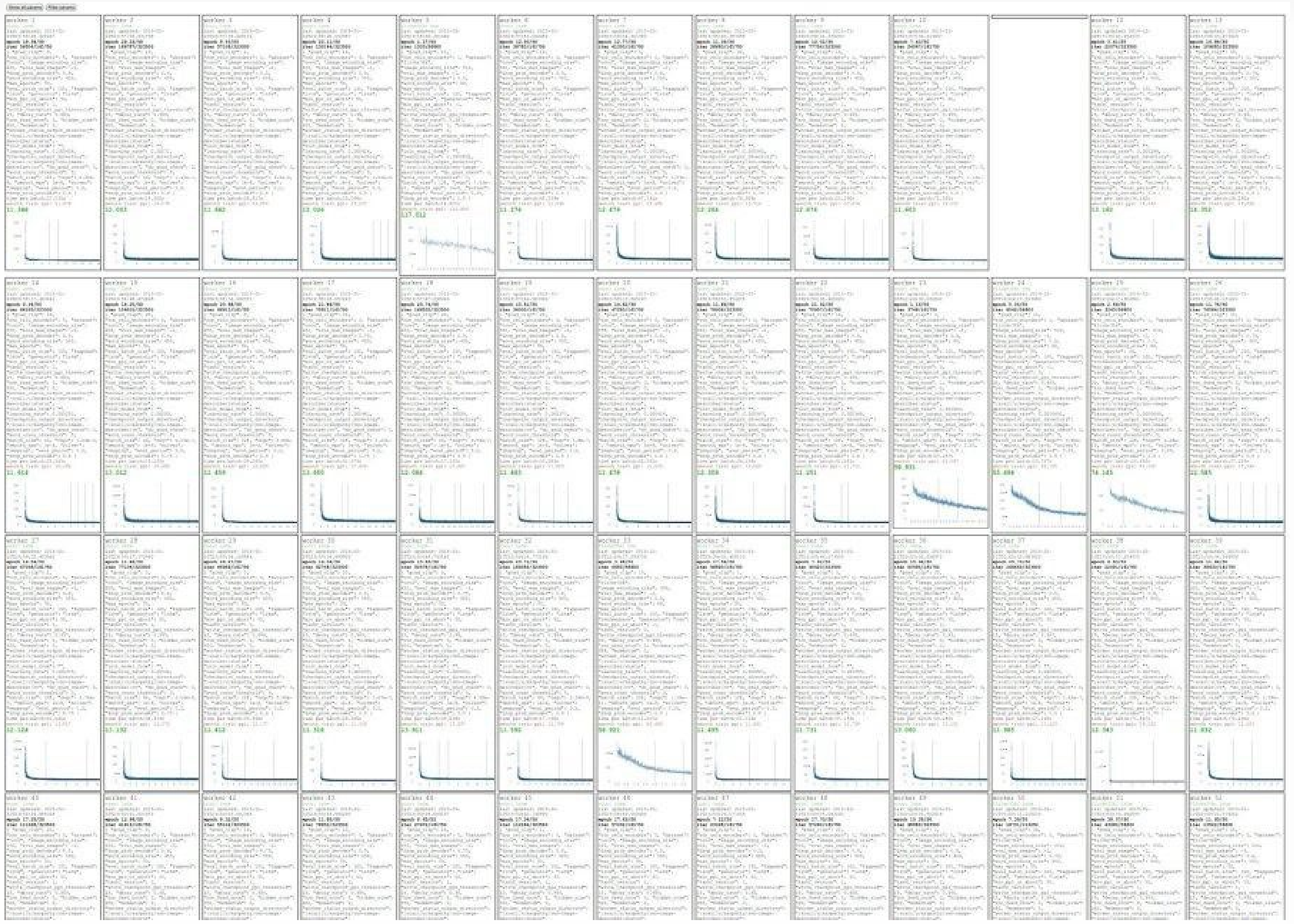
Choosing Hyperparameters

- Step 1: Check initial loss
- Step 2: Overfit a small sample
- Step 3: Find LR that makes loss go down
- Step 4: Coarse grid, train for ~1-5 epochs
- Step 5: Refine grid, train longer
- Step 6: Look at loss curves
- Step 7: GOTO step 5

Hyperparameters to play with

- Network architecture
- Learning rate, its decay schedule, update type
- Regularization (L2/Dropout strength)

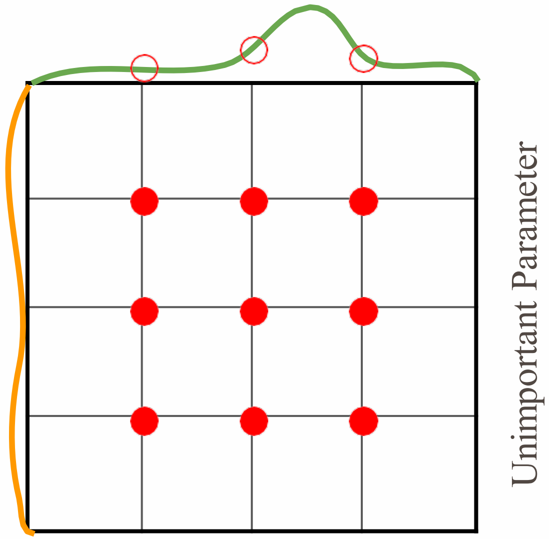
Cross-validation "command center"



Random Search vs. Grid Search

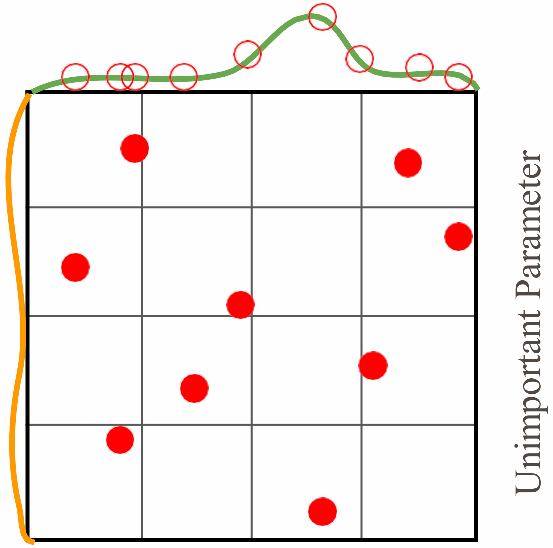
Hyper-Parameter Optimization Bergstra and Bengio, 2012

- Random Search for



Important Parameter

Grid Layout

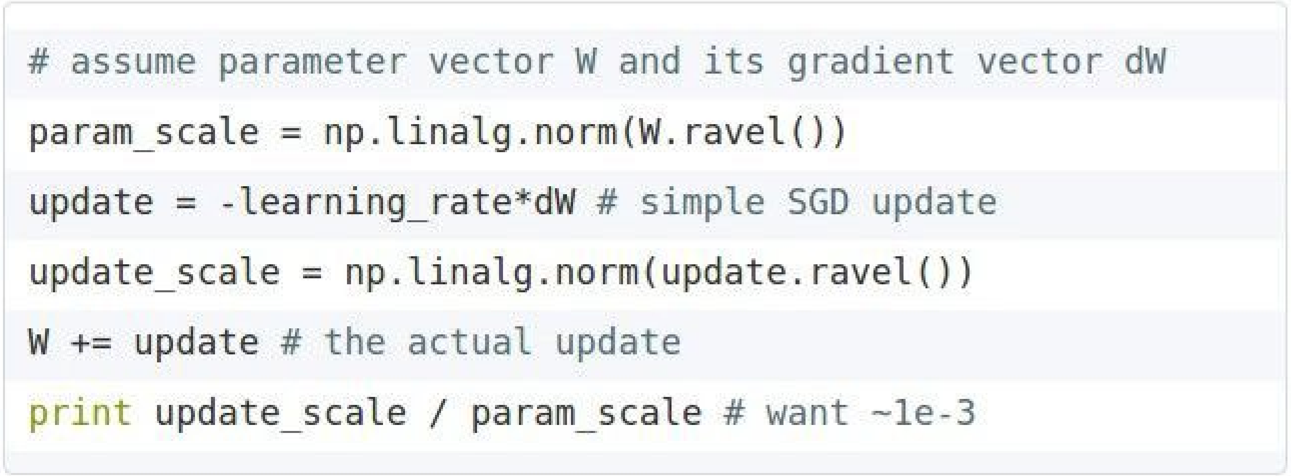


Important Parameter

Random Layout

Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Track the ratio of weight updates / weight magnitudes:



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

ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters

NEXT: HOW TO BUILD THE CNN WITH TF/PYTORCH?