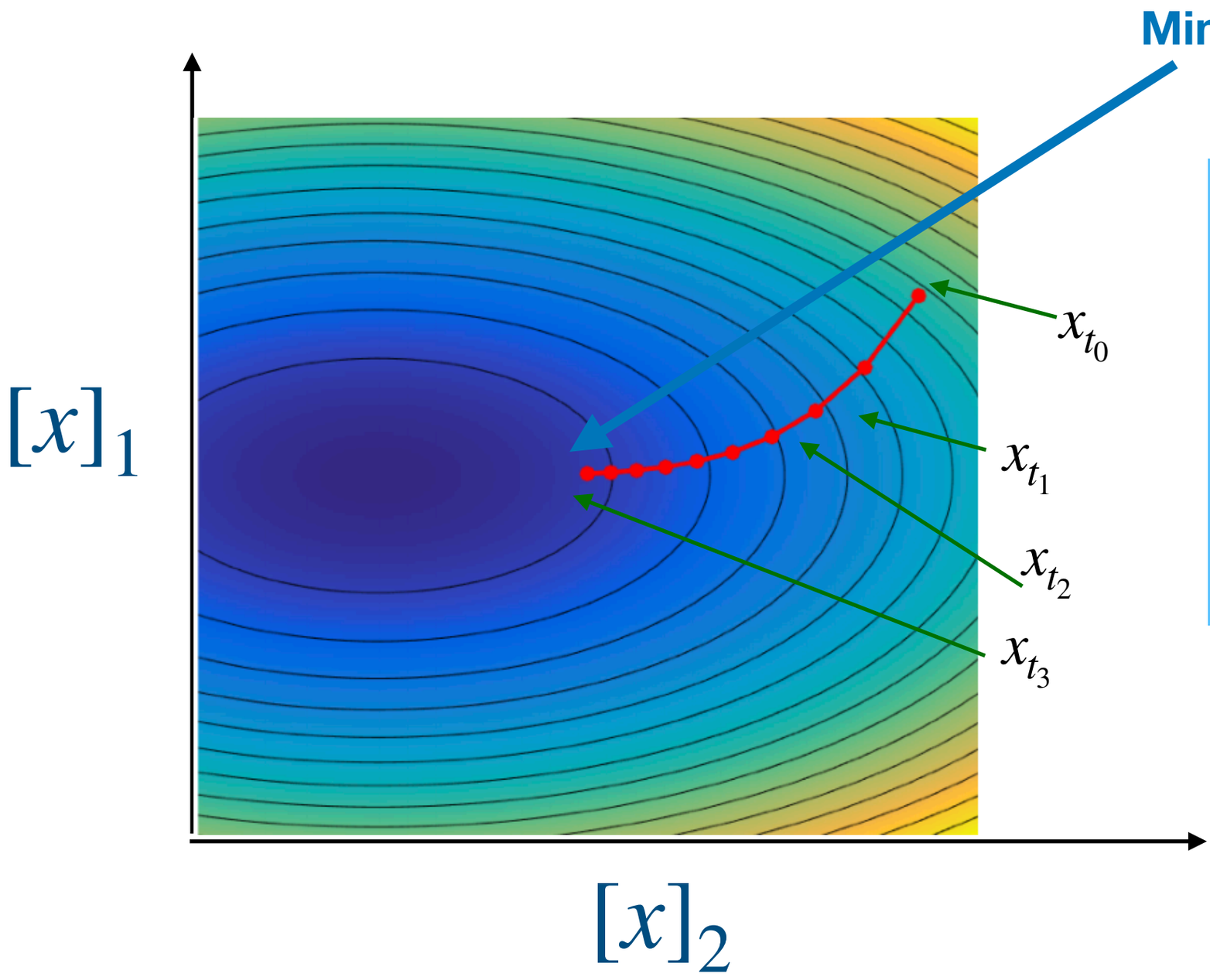
Lecture 3: Optimization for Deep Learning

Gradient Descent in 2D

Visualization in 2 dimensions using contours



Initialize $\mathbf{x}_0$

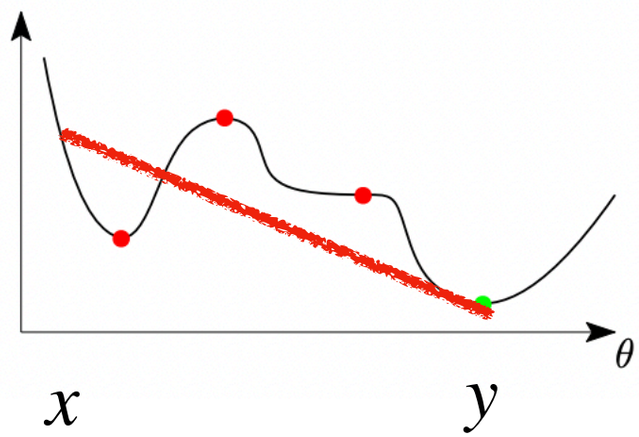
Iterate $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t)$

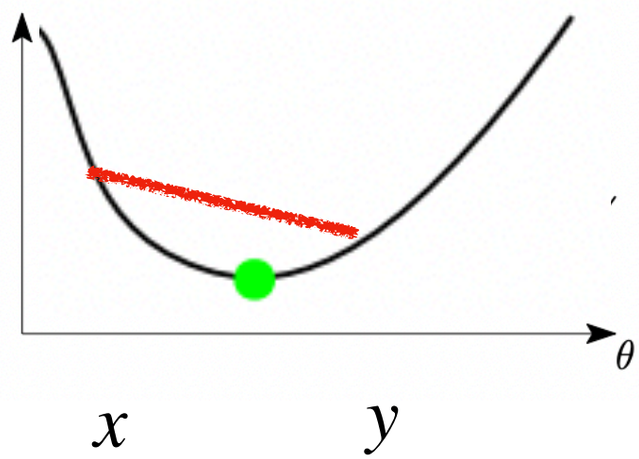
Stopping Crit. $|f(\mathbf{x}_{t+1}) - f(\mathbf{x}_t)| < \epsilon$

Convexity

$$\theta \in [0,1] \qquad f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta)f(\mathbf{y})$$

- Choose two points draw a line
- Convex if line is above graph



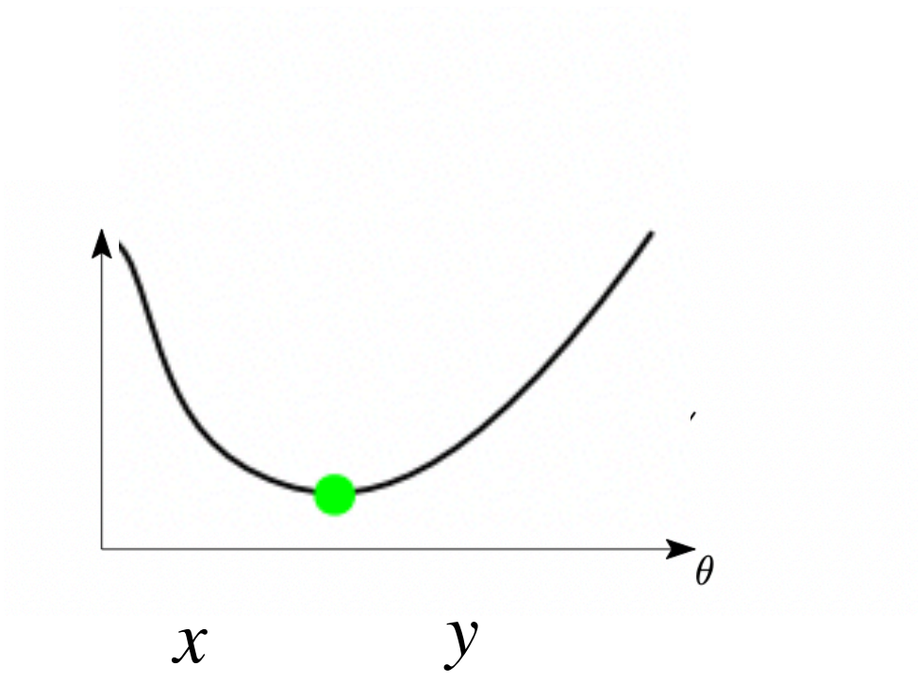


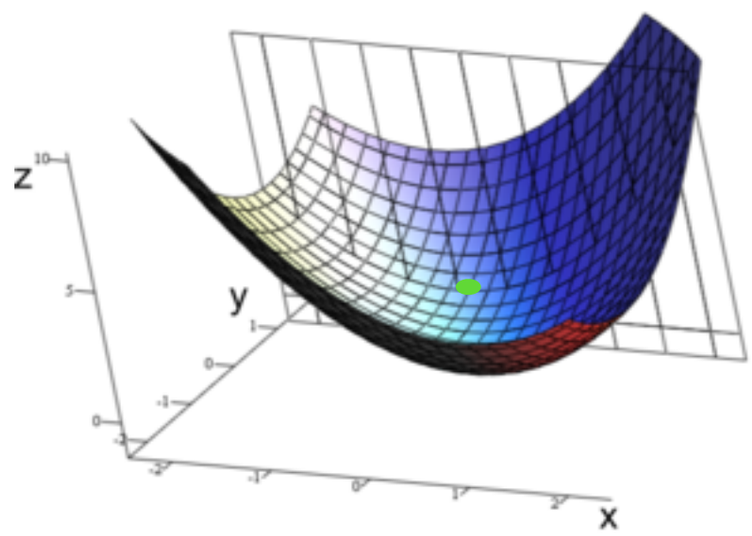
Convex Optimization

Convex function

$$\theta \in [0,1] \qquad f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$$

- Many optimization methods come from convex optimization

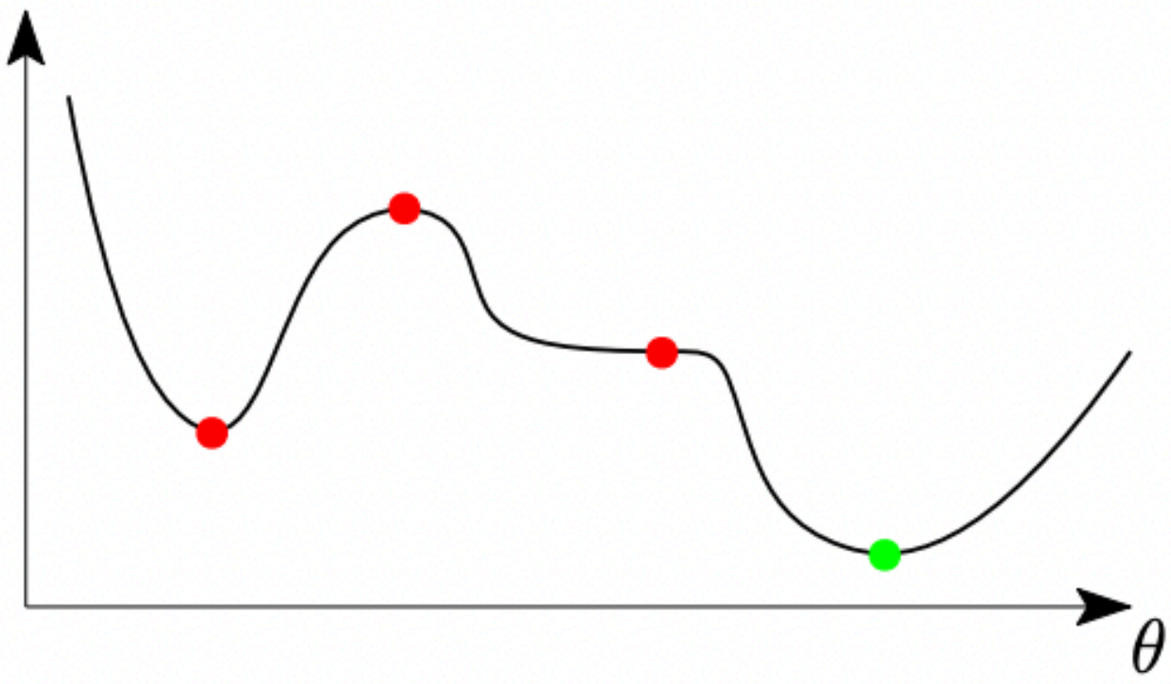


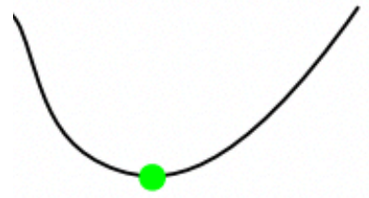


Local and Global Minimum

- Stationary Points
- Global Minimum
- Local Minimum

- For convex functions local minima are global minimum





Convex Optimization

$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta)f(\mathbf{y})$$

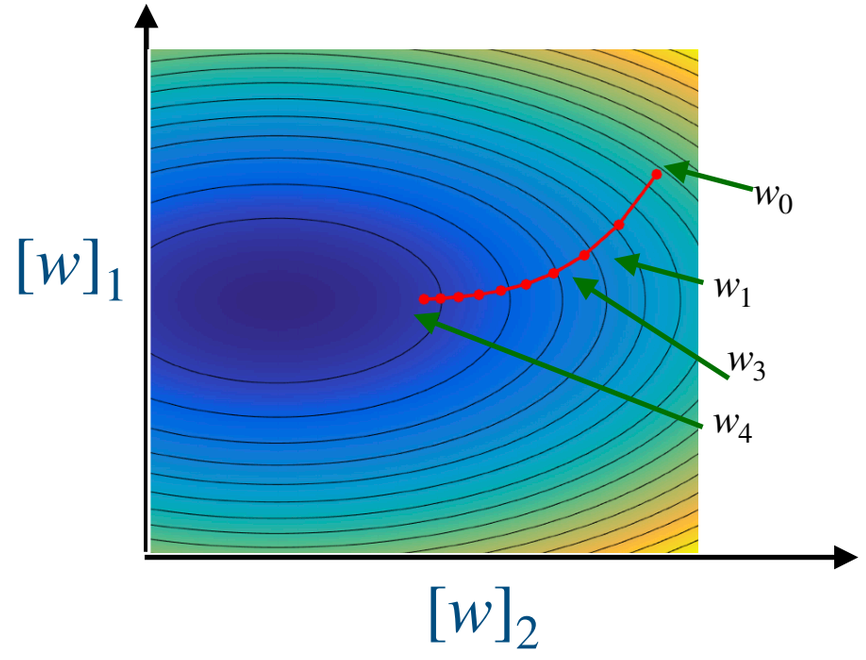
- Linear model + standard loss functions typically yield convex problems
- Convex optimization often allows us provable ways to reach a global minus of the function
- Deep networks + standard loss functions typicaly yield a highly non-convex optimization problem

Non-convex optimization

- Non-convex optimization is NP-hard
- Fortunately the objective and loss surface in deep networks has properties that allow reasonable solutions in practice

Convergence of Gradient Descent

$$w_{k+1} = w_k - \alpha \nabla f(w_k)$$



Under conditions (to follow)

For convex functions we can show

$$f(w^k) - f(w^*) \leq \frac{\|w^k - w^*\|_2^2}{2\alpha k}$$

w^k **k-th iterate**

w^* **Minimum**

α **Learning rate**

Converges at rate O(1/k)

BertSekas et al GRADIENT CONVERGENCE IN GRADIENT METHODS WITH ERRORS*

Convergence of Gradient Descent

$$f(w^k) - f(w^*) \leq \frac{\|w^k - w^*\|_2^2}{2\alpha k}$$

1. **We assume Lipschitz continuous gradient**

$$\frac{\|\nabla f(x) - \nabla f(y)\|_2}{\|x - y\|_2} \leq L$$

- Limits how fast gradients can change
- Often valid

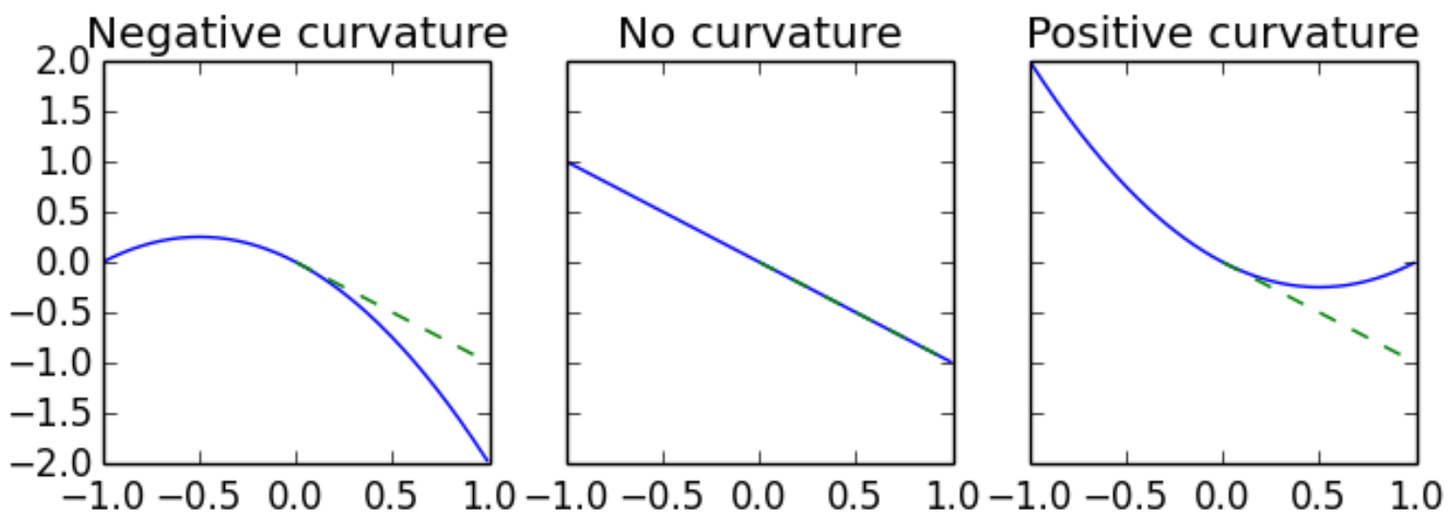
2. **Learning rate** $\alpha \leq \frac{1}{L}$

Hessian

- Matrix of all second partial derivatives
- Characterizes curvature in high dimension

$$f : \mathscr{R}^D -> \mathscr{R}$$

$$Hf = \nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial w_1^2} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_n} \\ \vdots & & \\ \frac{\partial f}{\partial w_n \partial w_1} & \cdots & \frac{\partial^2 f}{\partial w_D^2} \end{bmatrix}$$



Gradient Descent

$$\frac{\|\nabla f(x) - \nabla f(y)\|_2}{\|x - y\|_2} \leq L$$

- For C^2 functions, Lipschitz continuity of the gradient is equivalent to

$$\nabla^2 f(w) \preceq LI, \qquad \nabla^2 f(w) = H$$

  for all w.

- "Eigenvalues of the Hessian are bounded above by L".
  - For least squares, minimum L is the maximum eigenvalue of $X^T X$.

- This means $v^T \nabla^2 f(u) v \leq v^T (LI) v$ for any u and v, or that

$$v^T \nabla^2 f(u) v \leq L\|v\|^2.$$

Slide courtesy of Mark Schmidt

Gradient Descent

- For a C^2 function, a variation on the multivariate Taylor expansion is that

$$f(v) = f(w) + \nabla f(w)^T(v-w) + \frac{1}{2}(v-w)^T\nabla^2 f(u)(v-w),$$

for any w and v (with u being some convex combination of w and v).

- Lipschitz continuity implies the green term is at most $L\|v-w\|^2$,

$$f(v) \leq f(w) + \nabla f(w)^T(v-w) + \frac{L}{2}\|v-w\|^2,$$

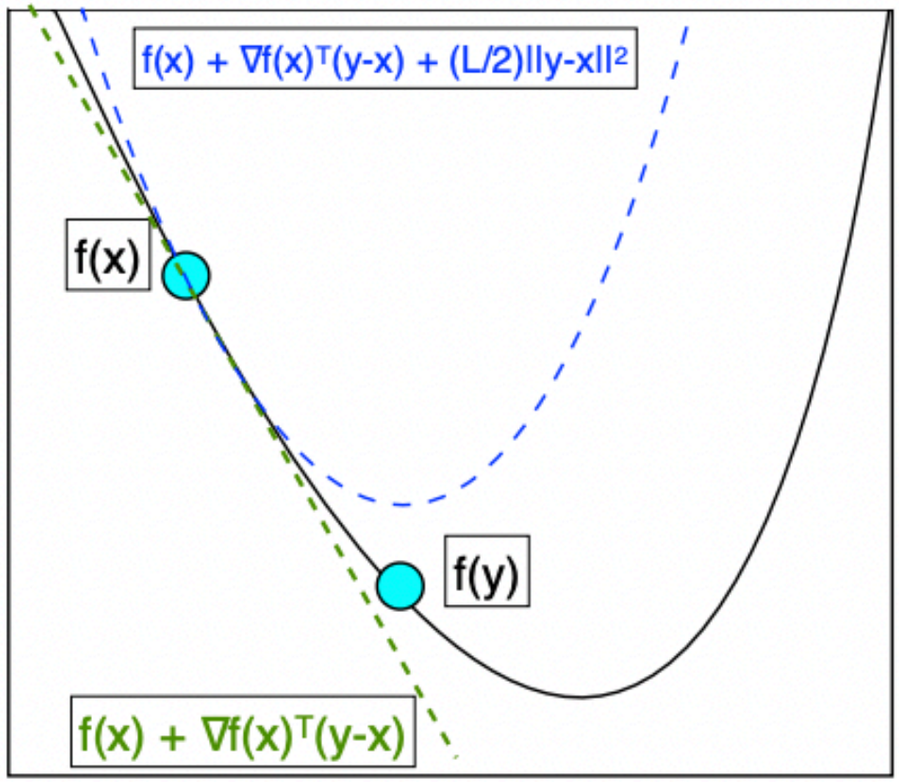
which is called the descent lemma.

Slide courtesy of Mark Schmidt

Descent Lemma

$$f(v) \leq f(w) + \nabla f(w)^T(v - w) + \frac{L}{2}\|v - w\|^2,$$

- The descent lemma gives us a convex quadratic upper bound on f:



- This bound is minimized by a gradient descent step from w with $\alpha_k = 1/L$.

Slide courtesy of Mark Schmidt

Convergence of Gradient Descent

- So let's consider doing gradient descent with a step-size of $\alpha_k = 1/L$,

$$w^{k+1} = w^k - \frac{1}{L}\nabla f(w^k).$$

- If we substitle w^{k+1} and w^k into the descent lemma we get

$$f(w^{k+1}) \leq f(w^k) + \nabla f(w^k)^T(w^{k+1} - w^k) + \frac{L}{2}\|w^{k+1} - w^k\|^2.$$

- Now if we use that $(w^{k+1} - w^k) = -\frac{1}{L}\nabla f(w^k)$ in gradient descent,

$$\begin{aligned} f(w^{k+1}) &\leq f(w^k) - \frac{1}{L}\nabla f(w^k)^T\nabla f(w^k) + \frac{L}{2}\|\frac{1}{L}\nabla f(w^k)\|^2 \\ &= f(w^k) - \frac{1}{L}\|\nabla f(w^k)\|^2 + \frac{1}{2L}\|\nabla f(w^k)\|^2 \\ &= f(w^k) - \frac{1}{2L}\|\nabla f(w^k)\|^2. \end{aligned}$$

Slide courtesy of Mark Schmidt

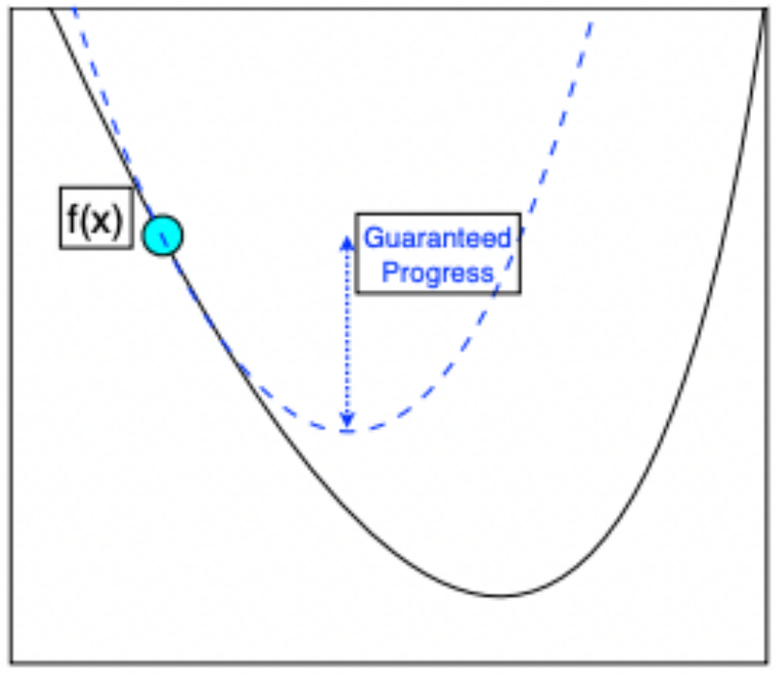
$$f(w^{k+1}) \leq f(w^k) - \frac{1}{2L}\|\nabla f(w^k)\|^2.$$

(Optional)

Convergence of Gradient Descent

Guaranteed progress

$$f(w^{k+1}) \leq f(w^k) - \frac{1}{2L}\|\nabla f(w^k)\|^2.$$



(Optional)

Gradient Descent for NN

Empiricial Risk Minimization

$$w^* = \arg\min_w \frac{1}{n} \sum_{i=1}^{n} l(f_w(x_i), y_i)$$

e.g. $$l(f_w(x), y) = \frac{1}{2}(f_w(x) - y)^2$$

$$\mathscr{L}(X, Y, w) = \frac{1}{n} \sum_{i=1}^{n} l(f_w(x), y) = \frac{1}{2n} \|Y - f_w(X)\|^2$$

Gradient of objective respect to weights

$$\nabla_w \mathscr{L}(X, Y, w)$$

w **All Parameters of Model**

X, Y **Data Matrix and Labels**

Stochastic Gradient Descent

Gradient Descent (GD)

Gradient of full objective

$$\nabla_w \mathscr{L}(X, Y, w)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_w \mathscr{L}(\mathbf{X}, \mathbf{Y}, \mathbf{w}_t)$$

Stochastic GD (SGD)

Gradient of loss w.r.t 1 sample

$$\nabla_w l(x, y, w)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_w l(\mathbf{x}, y, \mathbf{w}_t)$$

Mini-batch SGD

Gradient of loss w.r.t sub-sample

$$X_n \subset X$$

$$\nabla_w \mathscr{L}(X_n, Y_n, w)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_w \mathscr{L}(\mathbf{X}_n, \mathbf{Y}_n, \mathbf{w}_t)$$

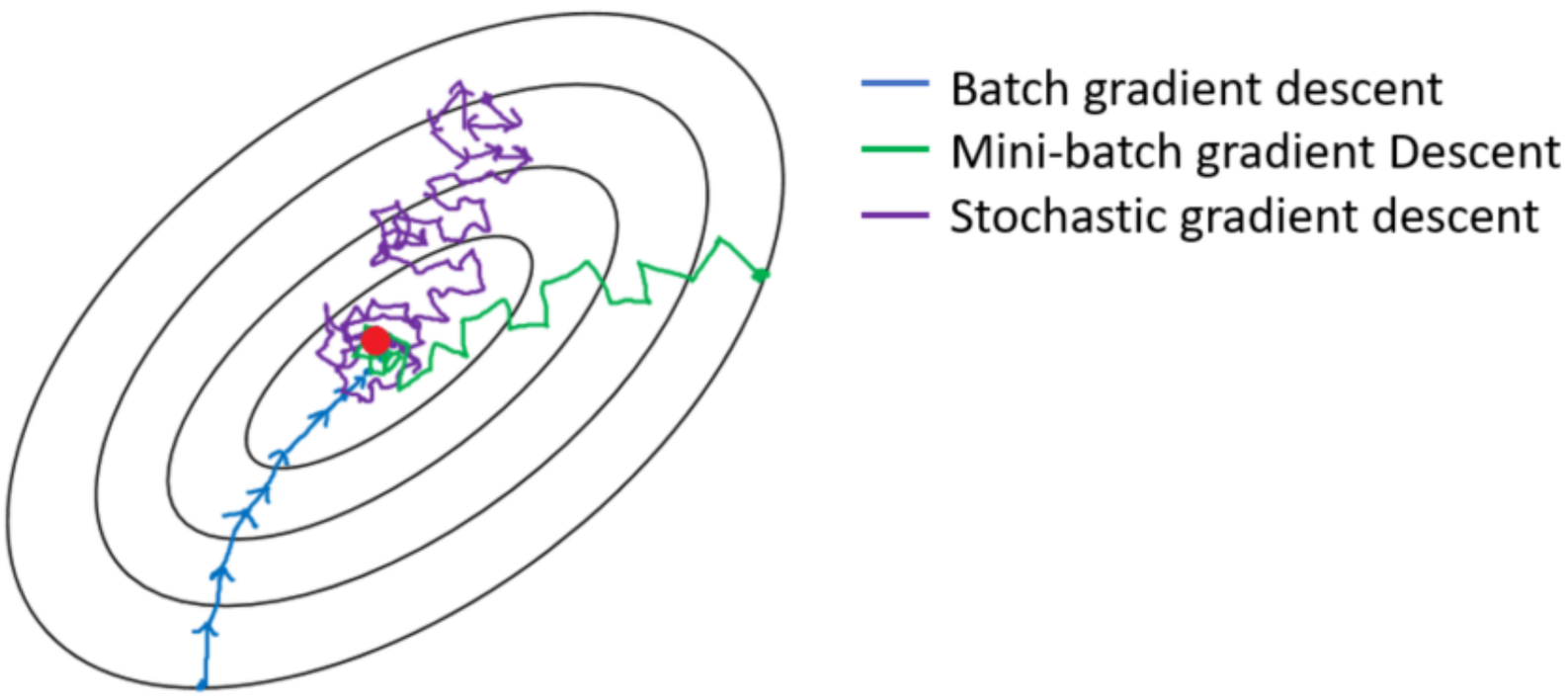
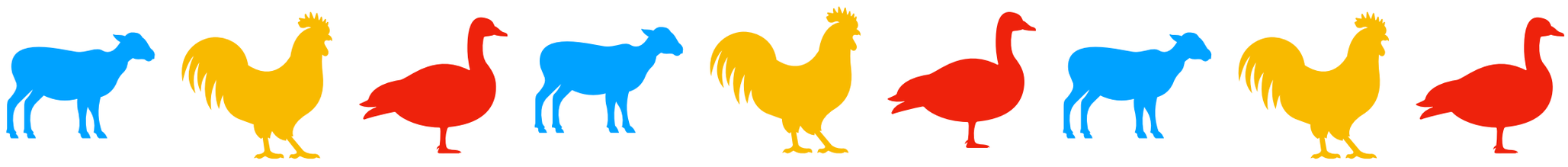


Image Credits: Toward Data Science

Intuition about Stochastic Methods

$$\frac{1}{|S1|}\sum_{i\in S1}\nabla_w l(x,y,w)$$

S1



$$\frac{1}{|S2|}\sum_{i\in S2}\nabla_w l(x,y,w)$$

S2

Stochastic vs Gradient Descent

- Stochastic Gradient Descent is much more scalable in large datasets
    - E.g. in convex settings convergence rates can be shown to be similar to GD while processing only one point at a time
- Stochastic Gradient Descent is a classic optimizations method that is the backbone of most modern neural network training

Mini-Batch SGD

- Classical SGD uses a single point
- Mini-batch SGD can be seen as obtaining a lower variance gradient estimate but without need for full batch

Mini-batch SGD

Gradient of loss w.r.t sub-sample $X_n \subset X$

$$\frac{1}{|X_n|} \sum_{(x_i,y_i)\in X_n} \nabla_w l(x_i, y_i, w)$$

- Mini-batch forward and backward processing is often more efficient on single GPU

Terminology: Recently, mini-batch SGD in many contexts is often just called SGD

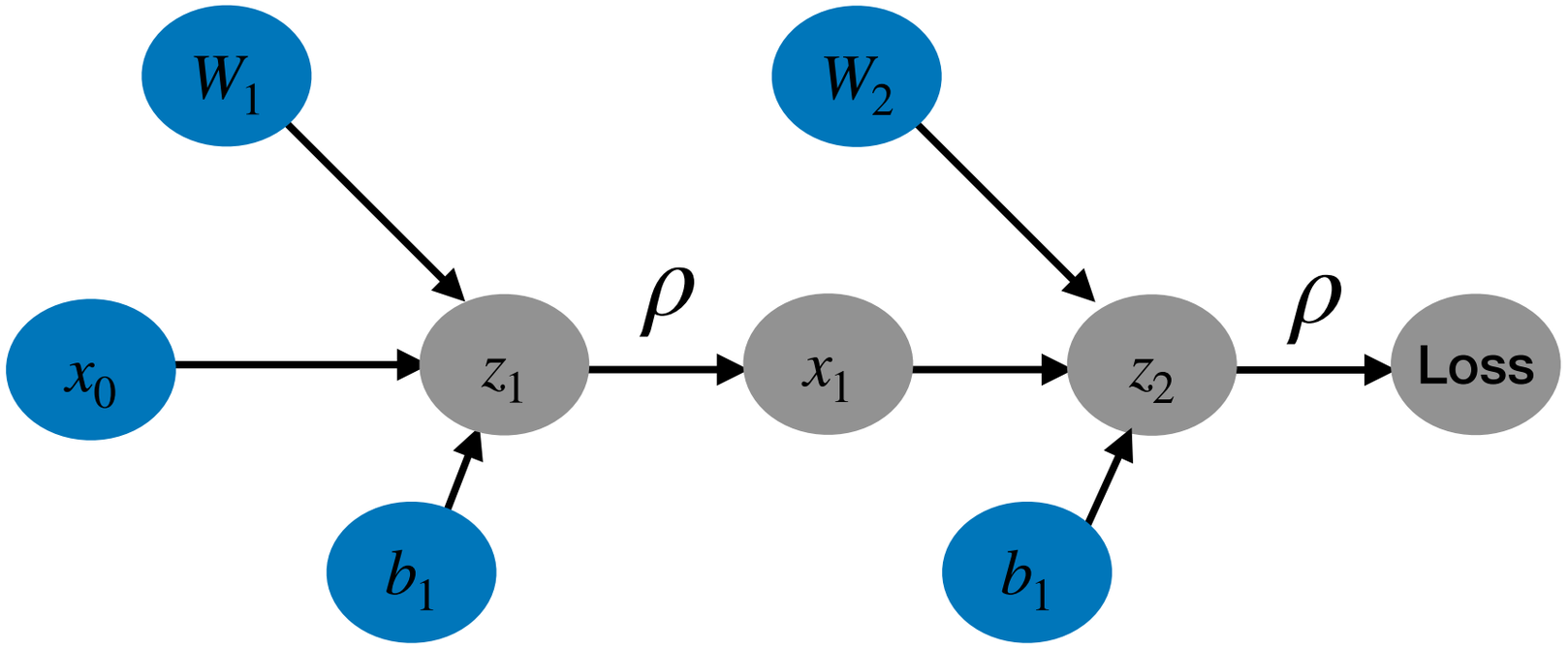
Mini-Batch SGD Algo

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
  $k \leftarrow k + 1$
end while

Mini-Batch SGD in Code

```
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.linear1 = nn.Linear(784,200)
        self.linear2 = nn.Linear(200,10)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        return self.linear2(x)

loss = nn.CrossEntropyLoss()
model = NN()
```



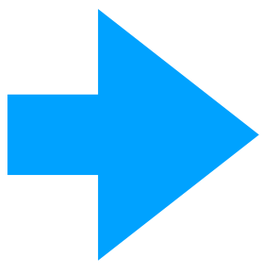

```
lr = 0.5  # learning rate
epochs = 2  # how many epochs to train for

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        data = x_train[start_i:end_i]
        target = y_train[start_i:end_i]
        pred = model(data)
        loss = loss_func(pred, target)

        loss.backward()
        with torch.no_grad():
            model.linear1.weights -= model.linear1.weights.grad * lr
            model.linear1.bias -= model.linear1.bias.grad * lr
            model.linear2.weights -= model.linear2.weights.grad * lr
            model.linear2.bias -= model.linear2.bias.grad * lr

            #zero out grad
            model.linear1.weights.grad.zero_()
            model.linear1.bias.grad.zero_()
            model.linear2.weights.grad.zero_()
            model.linear2.bias.grad.zero_()
```

Each node holds gradient buffer

Mini-Batch SGD in Code

```
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.linear1 = nn.Linear(784,200)
        self.linear2 = nn.Linear(200,10)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        return self.linear2(x)

loss = nn.CrossEntropyLoss()
model = NN()
```

```
lr = 0.5  # learning rate
epochs = 2  # how many epochs to train for

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        data = x_train[start_i:end_i]
        target = y_train[start_i:end_i]
        pred = model(data)
        loss = loss_func(pred, target)

        loss.backward()
        with torch.no_grad():
            model.linear1.weights -= model.linear1.weights.grad * lr
            model.linear1.bias -= model.linear1.bias.grad * lr
            model.linear2.weights -= model.linear2.weights.grad * lr
            model.linear2.bias -= model.linear2.bias.grad * lr

            #zero out grad
            model.linear1.weights.grad.zero_()
            model.linear1.bias.grad.zero_()
            model.linear2.weights.grad.zero_()
            model.linear2.bias.grad.zero_()
```



```
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(train_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()
```

Stochastic Gradient Descent

- Stochastic gradient descent can be shown to reach the global minimum in convex settings
- SGD can be shown to reach a stationary point in non-convex settings with some assumptions
  - Assumption - Appropriate step size sequence
  - Assumption - Lipschitz continuity

Theorem 4.9 (**Nonconvex Objective, Diminishing Stepsizes**). *Under Assumptions 4.1 and 4.3, suppose that the SG method (Algorithm 4.1) is run with a stepsize sequence satisfying* (4.19). *Then*

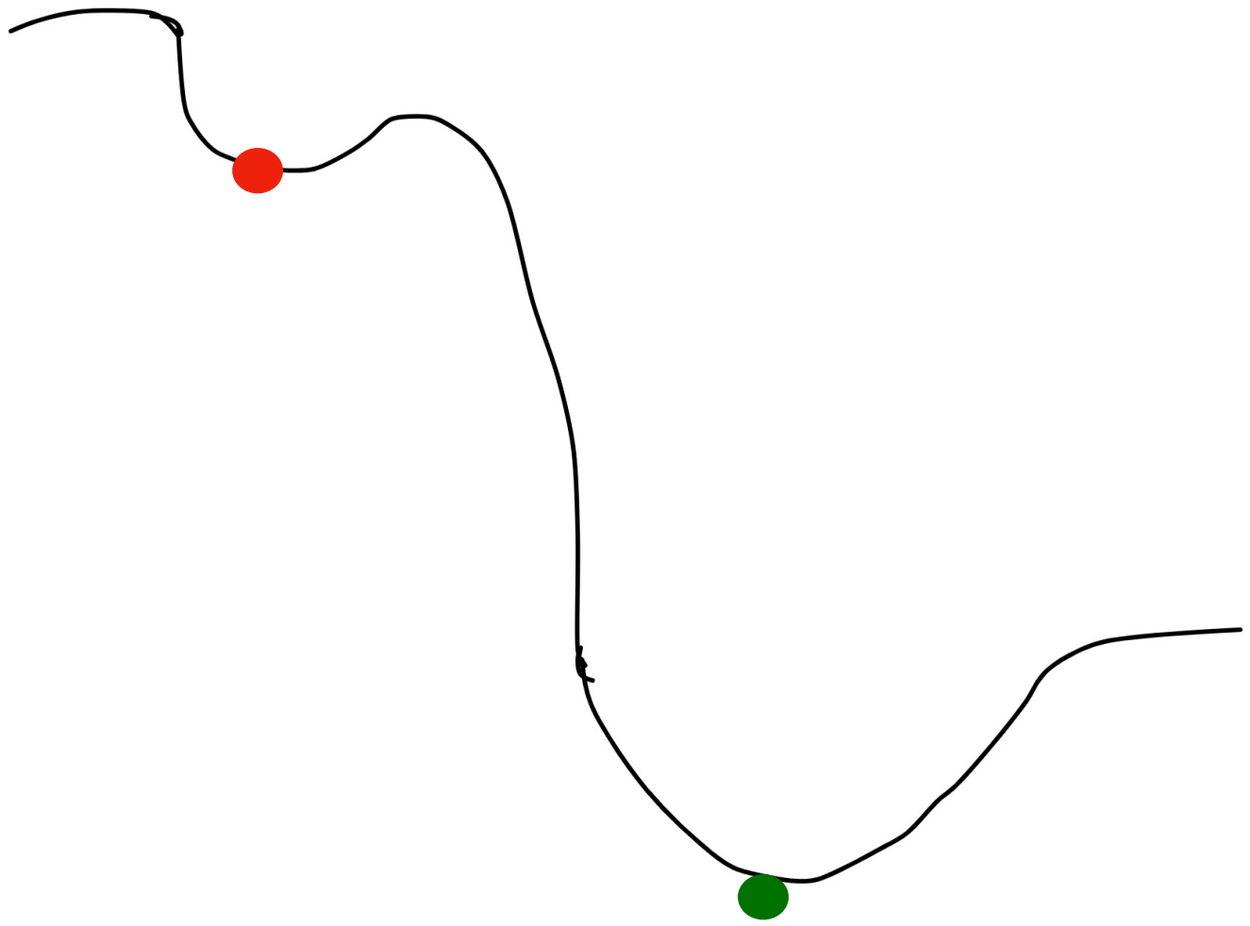
$$\liminf_{k\to\infty} \mathbb{E}[\|\nabla F(w_k)\|_2^2] = 0\,. \tag{4.29}$$

Optimization Methods for Large-Scale Machine Learning

Léon Bottou, Frank E. Curtis, Jorge Nocedal

Minimum in Non-Convex Setting

- Are local minimum useful?



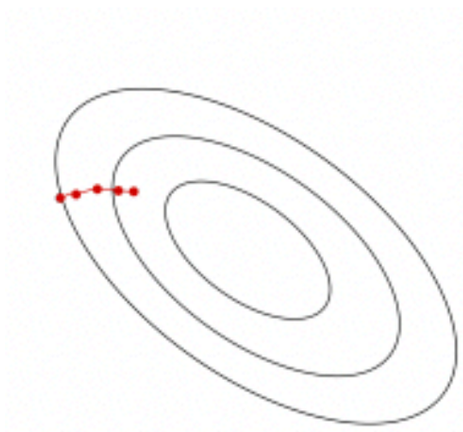
Local Minimum Can be Useful

- It has been empirically shown in some cases that many local minimum in highly non-convex functions obtained by SGD are “close” to global minimum. Still poorly understood why this occurs

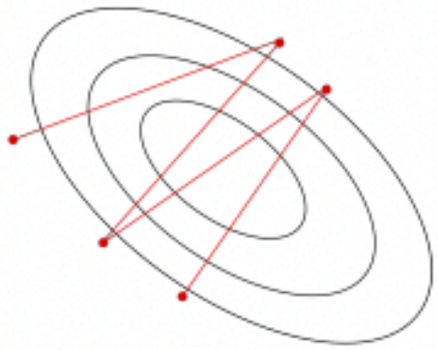
- With many assumptions and specific model classes recent results have begun to show that SGD can converge to global minima

GD Step Size

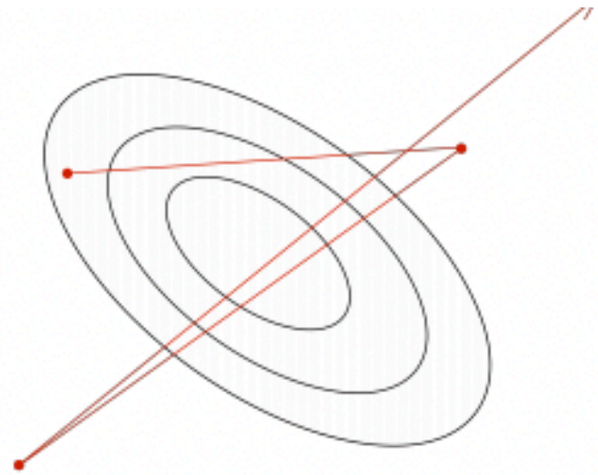
- Step size or learning rate can often greatly affect the optimization



α too small:
slow progress



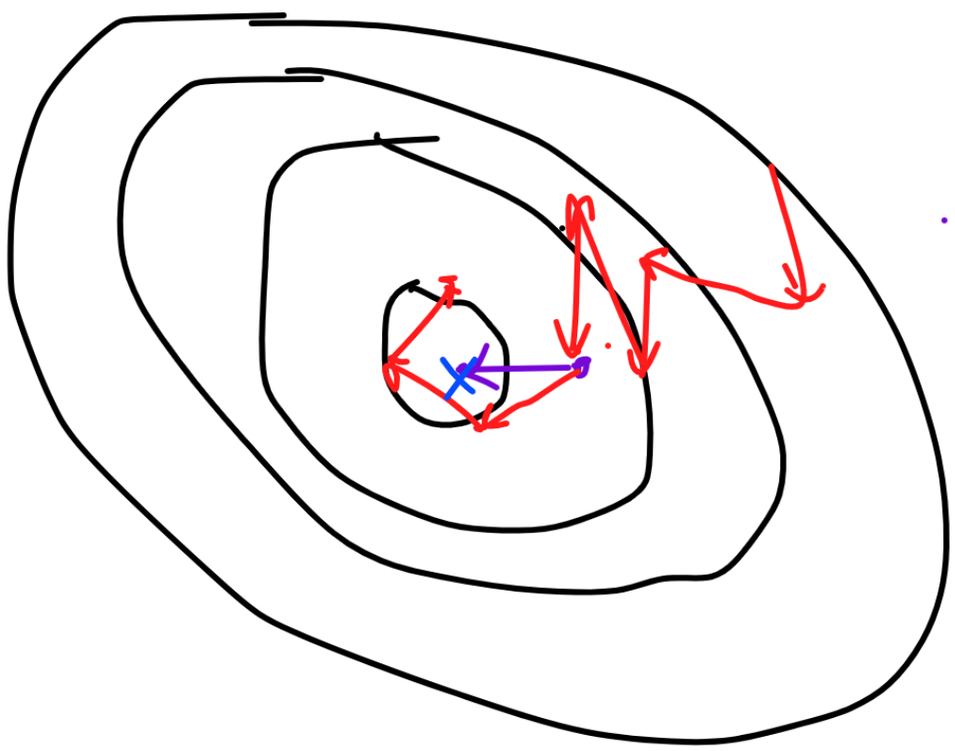
α too large:
oscillations

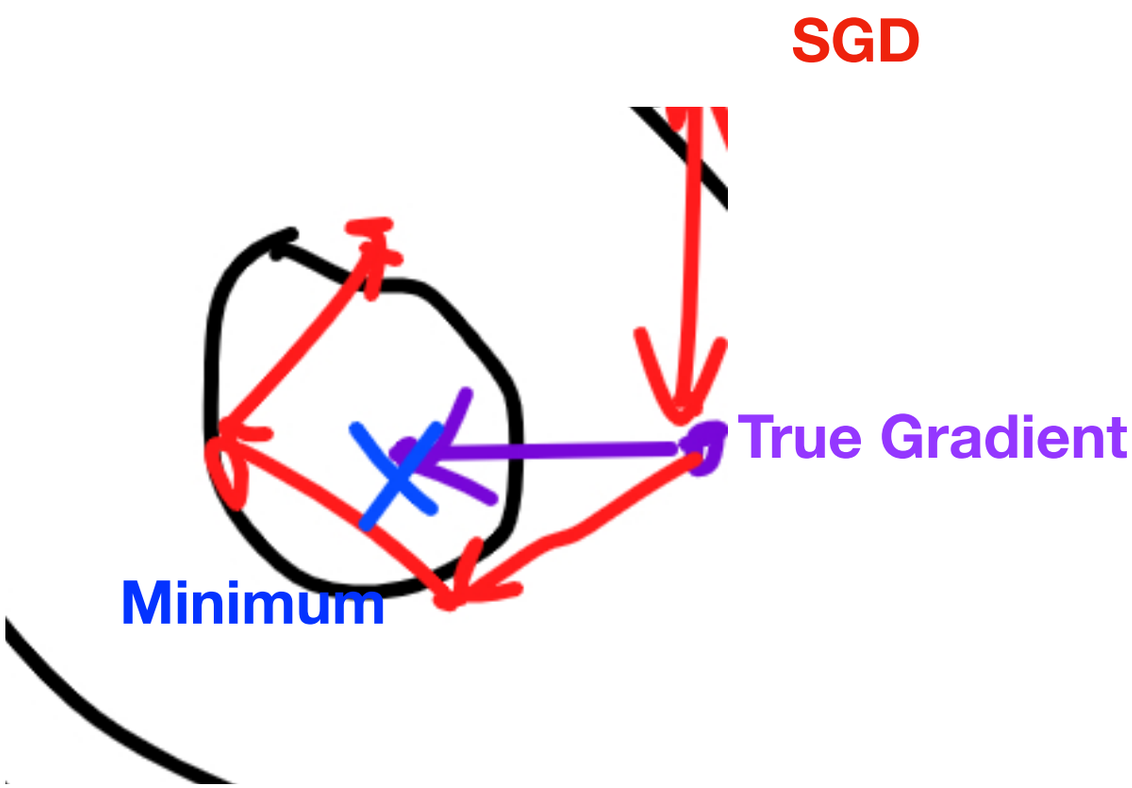


α much too large:
instability

Slide credit: Roger Grosse

Learning Rate Schedules SGD





- In stochastic optimization larger learning rates even if they are of the correct size to reach minimum may bounce around without hitting the solution

- Similar to GD small learning rates would be too slow

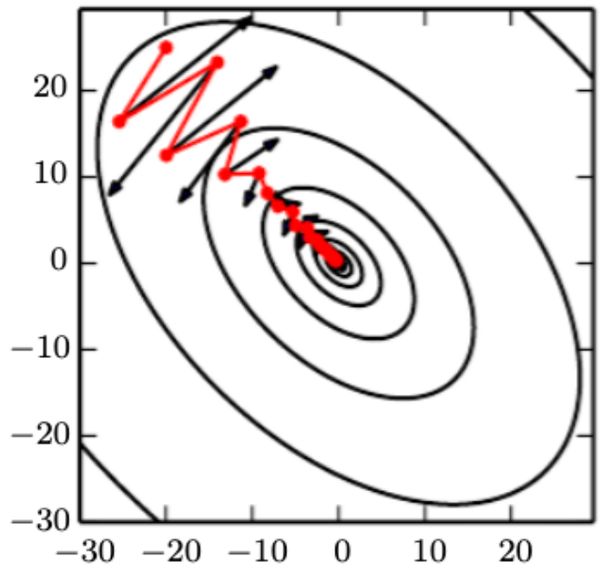
- In SGD we typically start at a high learning rate and decay

Momentum

$$\mathbf{g} = \nabla_w \mathscr{L}(X, Y, w_t)$$

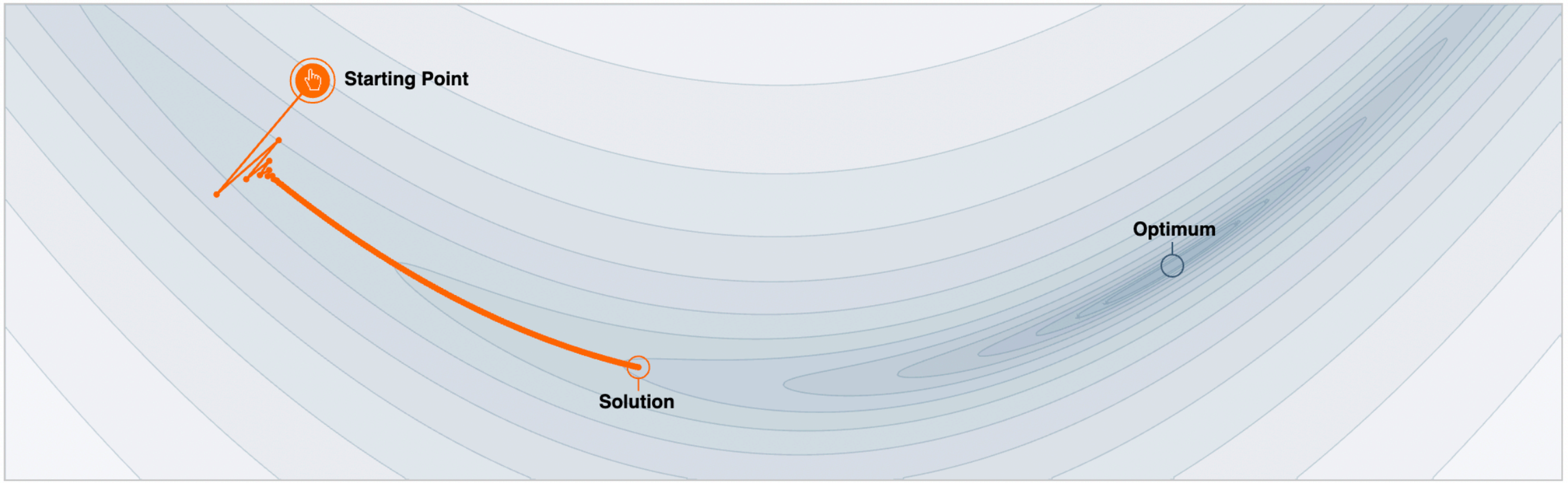
$$\mathbf{v}_{t+1} = \mu * \mathbf{v}_t + \mathbf{g}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha * \mathbf{v}_{t+1}$$



- Popular and simple approach to speed up and stabilize learning
- Dampens oscillations and noise from noisy gradient
- Often accelerates training

Visualizations



https://distill.pub/2017/momentum/

Adaptive Learning Rate

- Attempt to adjust learning rate based on rules to better suit local curvature without explicitly representing the hessian

- Typically based on heuristics

RMSProp

- Individually adapt learning rate of each parameter
- Divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- Parameters with very large gradients have less effect

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

Adam

- Extremely popular optimization algorithm

- Can be seen as a combination of momentum and Rmsprop

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while

- Adam rose to prominence in training models where learning rate schedules were extremely difficult to determine

- Robust parameters

Kingma, D Ba, J Adam: A Method for Stochastic Optimization

Second-order optimization

$$f(w) = \mathscr{L}(X, Y, w) = \frac{1}{n}\sum_{i=1}^{n} l(g_w(x), y) = \frac{1}{2n}\|Y - g_w(X)\|^2$$

$$\hat{f}(w) \approx \hat{f}(v) + \nabla f(v)^T(w - v) + \frac{1}{2}(w - v)^T \nabla^2 f(v)(w - v)$$

Newton Method

$$\hat{f}(w) \approx \hat{f}(v) + \nabla f(v)^T(w - v) + \frac{1}{2}(w - v)^T \nabla^2 f(v)(w - v)$$

H

Solve local approximation

$$\min_w \hat{f}(v) + \nabla f(v)^T(w - v) + \frac{1}{2}(w - v)^T H(w - v)$$

$$w* = v - H^{-1} \nabla f(v)$$

Second-order optimization

$$\hat{f}(w) \approx \hat{f}(v) + \nabla f(v)^T (w - v) + \frac{1}{2}(w - v)^T \nabla^2 f(v)(w - v)$$

H

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of m examples
  while stopping criterion not met **do**
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

- Requires inverse hessian
- Large memory and computation
- O(D^3) compute and O(D^2) memory

Second-order optimization

$$\hat{f}(w) \approx \hat{f}(v) + \nabla f(v)^T (w - v) + \frac{1}{2}(w - v)^T \nabla^2 f(v)(w - v)$$

H

$$v* = v - H^{-1} \nabla f(v)$$

- Common second order approaches attempt to approximate the Hessian
- BFGS is one of the more successful
- Still maintains O(D^2) computation and O(D^2) memory cost
- L-BFGS further reduces memory cost, very popular outside of DL
- KFAC, Krylov Subspace methods

Second-order optimization

- In practice at the moment second order methods are rarely used in Deep Learning

- Many properties of these methods particularly in estimating the inverse hessian and their behaviour in the stochastic setting is not yet well understood

- They hold promise for potentially more rapid optimization

- Several promising results using novel hessian approximations (KFAC)

Ba, Jimmy, Roger Grosse, and James Martens. "Distributed second-order optimization using Kronecker-factored approximations." (2016).

Generalization and Optimization

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} l(f_w(x_i), y_i) + \Omega(w)$$

- Classic machine learning often separate the optimization of the objective function and properties of the optimum as separate concepts

- In practice a lot of success is due to implicit regularization from optimization methods

SGD is good for generalization

- Growing body of evidence shows that SGD has better generalization properties
- Intuitively the process of sampling the training data in SGD mimics the process of sampling test/train
- Several works empirically show SGD can find “flatter minimum”
- This makes it particularly hard to theoretically analyze optimization algorithms in DL

Kuzborskij, Ilja and Christoph H. Lampert. “Data-Dependent Stability of Stochastic Gradient Descent.” *ICML* (2018).

Hardt, Moritz, Ben Recht, and Yoram Singer. "Train faster, generalize better: Stability of stochastic gradient descent." *International Conference on Machine Learning*. PMLR, 2016.

SGD is good for generalization

- Several works argue that SGD with small mini-batch can find flatter minimum and that flat minimum generalize better

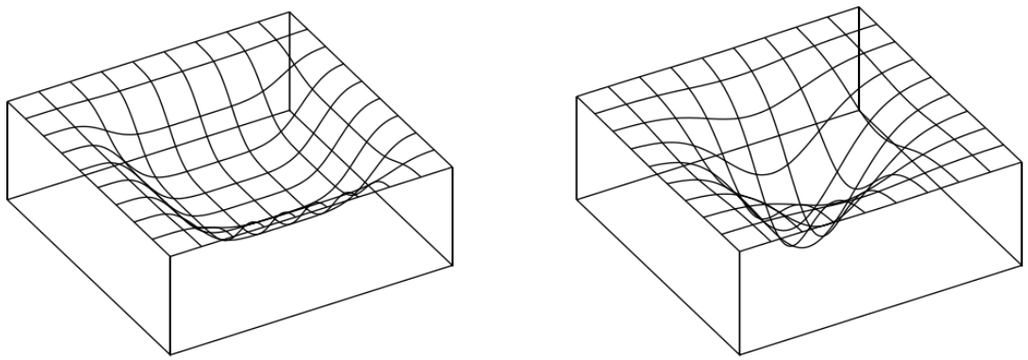
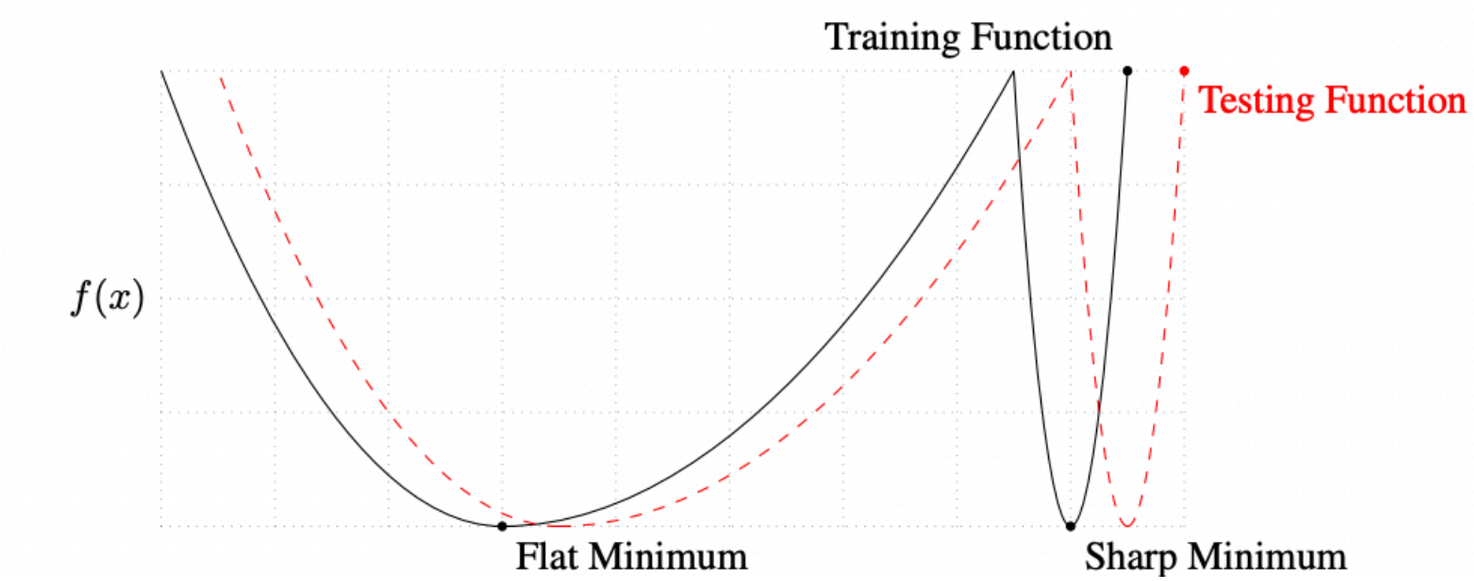


Figure 1: *Example of a "flat" minimum.*

Figure 2: *Example of a "sharp" minimum.*

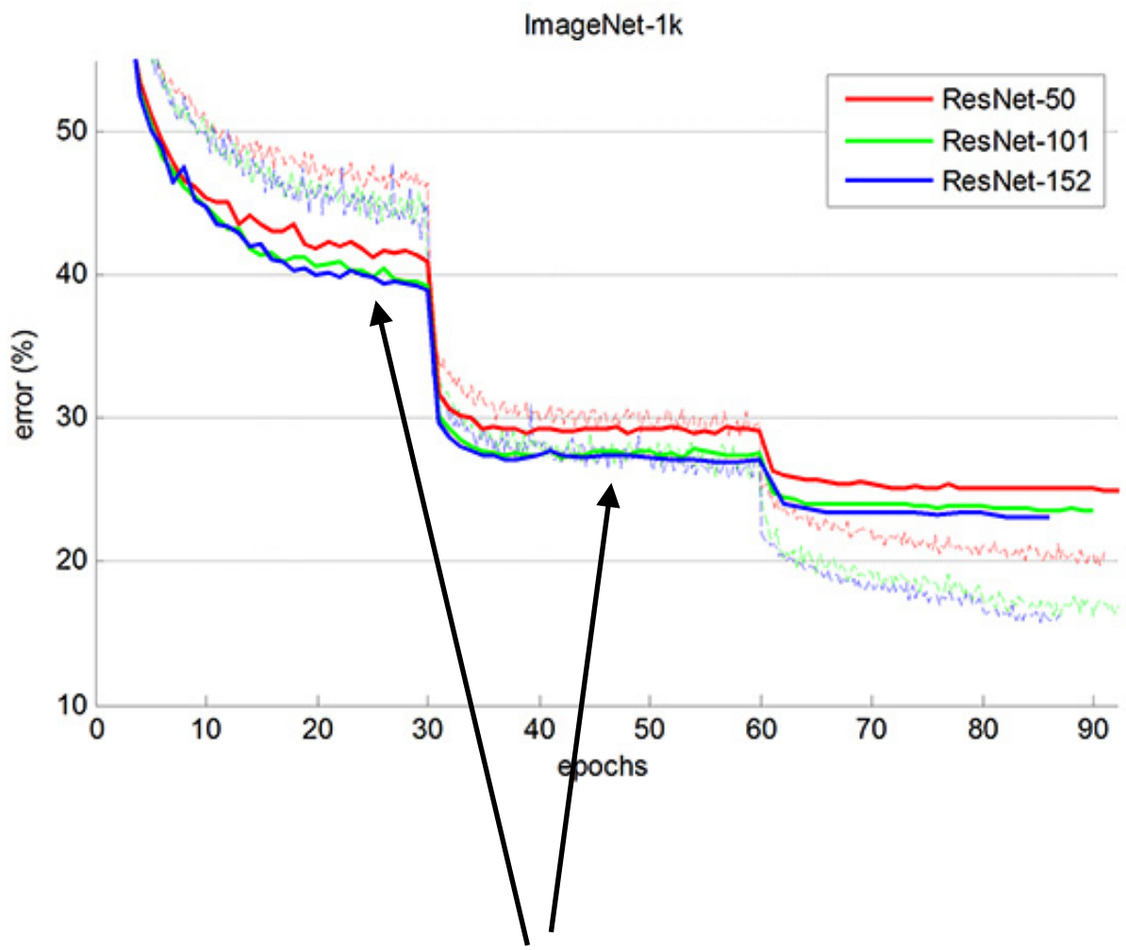
Hochreiter and Schmidhuber "Flat Minima" *1997*.



Keskar, Nitish Shirish, et al. "On large-batch training for deep learning: Generalization gap and sharp minima." *arXiv preprint arXiv:1609.04836* (2016).

SGD and Generalization

Kaiming He's 2015 Imagenet Competition Winner



Why don't we decay the learning rate in this flat regions?

Initial training at high learning rate has been observed to act as a regularizer

Some initial explanations for this effect have shown in the literature:

Li, Yuanzhi, Colin Wei, and Tengyu Ma. "Towards explaining the regularization effect of initial large learning rate in training neural networks." *Advances in Neural Information Processing Systems*. 2019.

Distributed Optimization

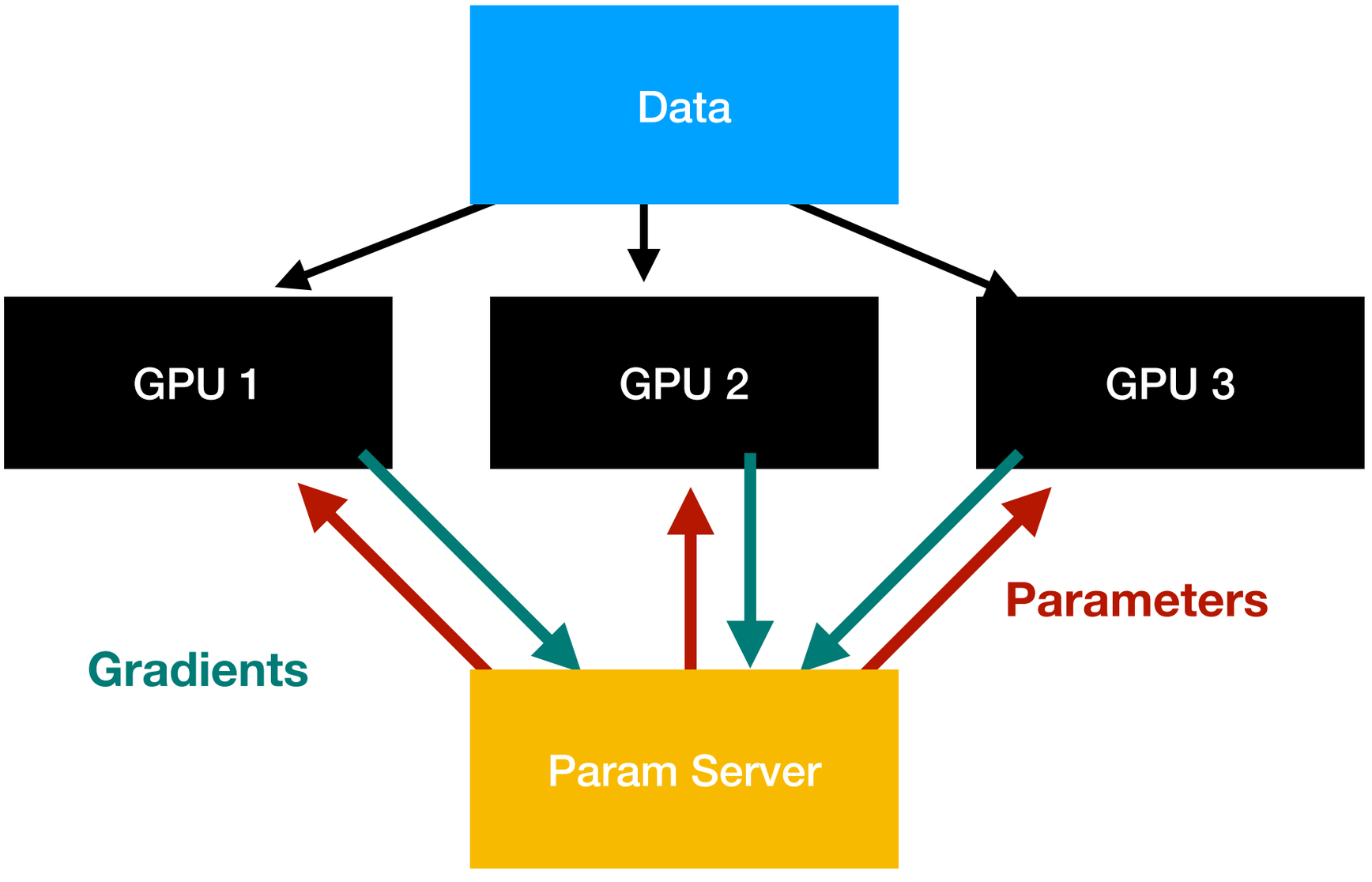
Parallelzing Deep Network Training

- Most common form of parallelism is data parallelism
  - Each node simultaneously process different mini batches
- Model parallelism - attempts to create models that split model across nodes
  - Difficult to parallelize in some cases

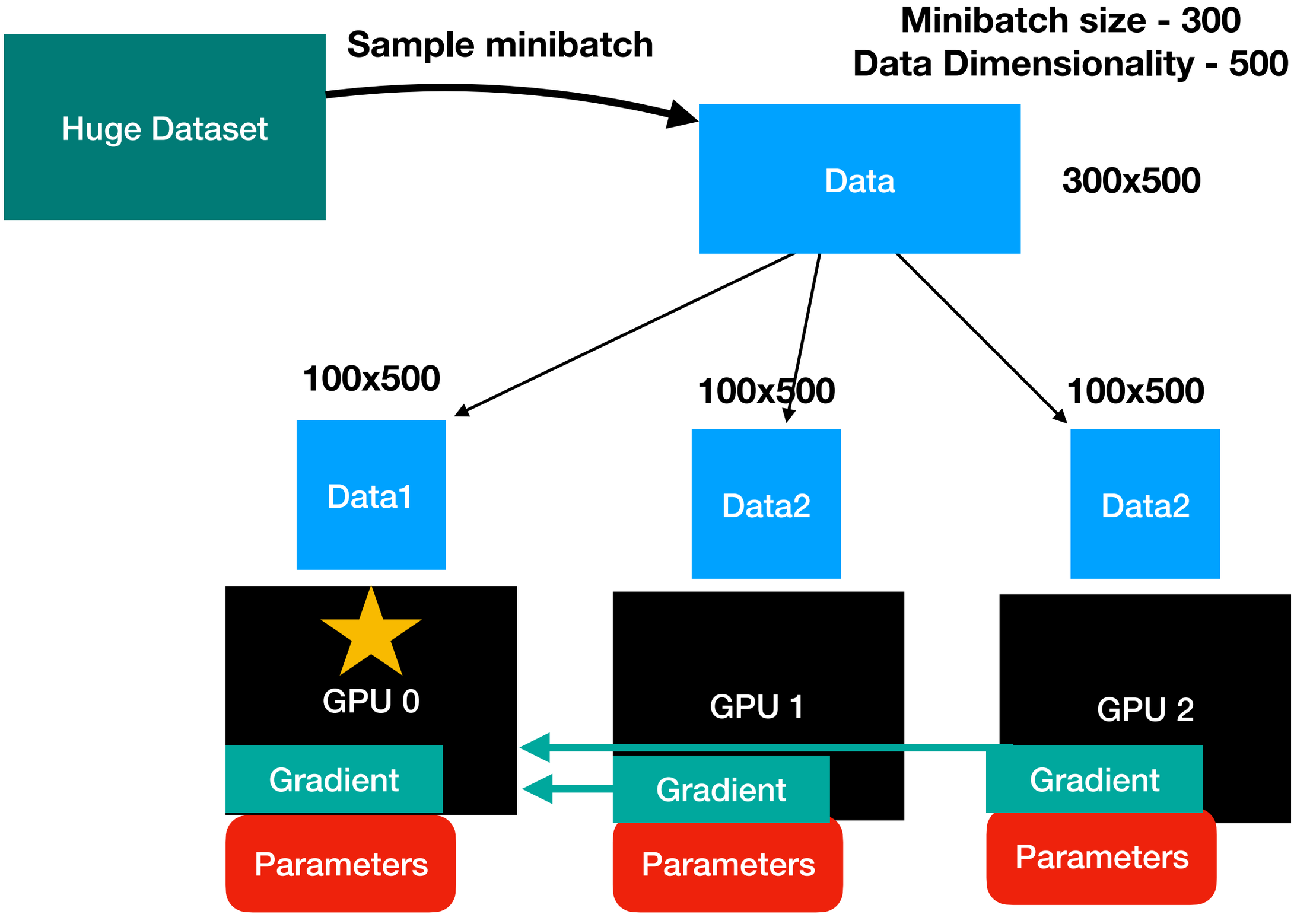
Wikimedia

Distributed Synchronous SGD

- Most common approach is Distributed Synchronous SGD
- Nodes (GPU) sample data and wait to receive parameters from a parameter server
- Parameter server will wait to aggregate gradients from all the nodes, then send new params



DataParallel SGD Pytorch



DataParallel + SGD Pytorch

```
class Model(nn.Module):
    # Our model

    def __init__(self, input_size, output_size):
        super(Model, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = self.fc(input)
        return output
```

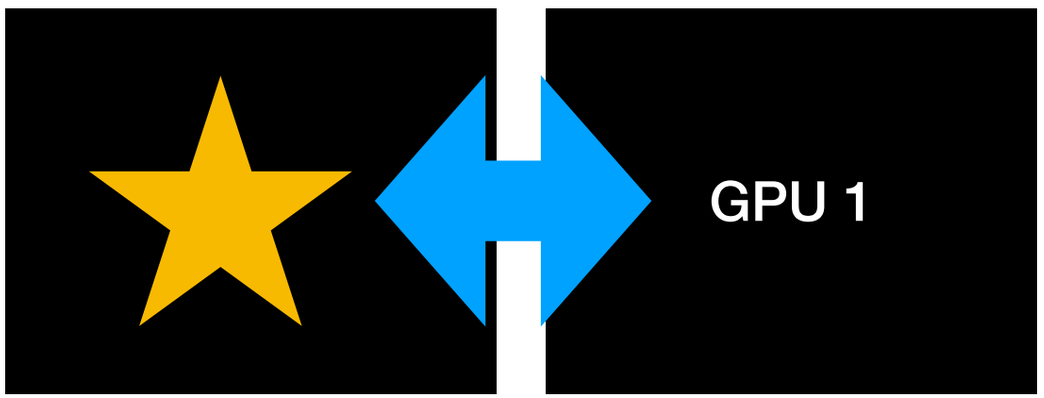
```
>>> net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
>>> output = net(input_var)  # input_var can be on any device, including CPU
```

```
optimizer = torch.optim.SGD(model.parameters(),lr=0.5)
for epoch in range(epochs):
  for batch_idx, (data, target) in enumerate(train_loader):
          data, target = data.to(device), target.to(device)
          optimizer.zero_grad()
          output = model(data)
          loss = loss_func(output, target)
          loss.backward()
          optimizer.step()
```

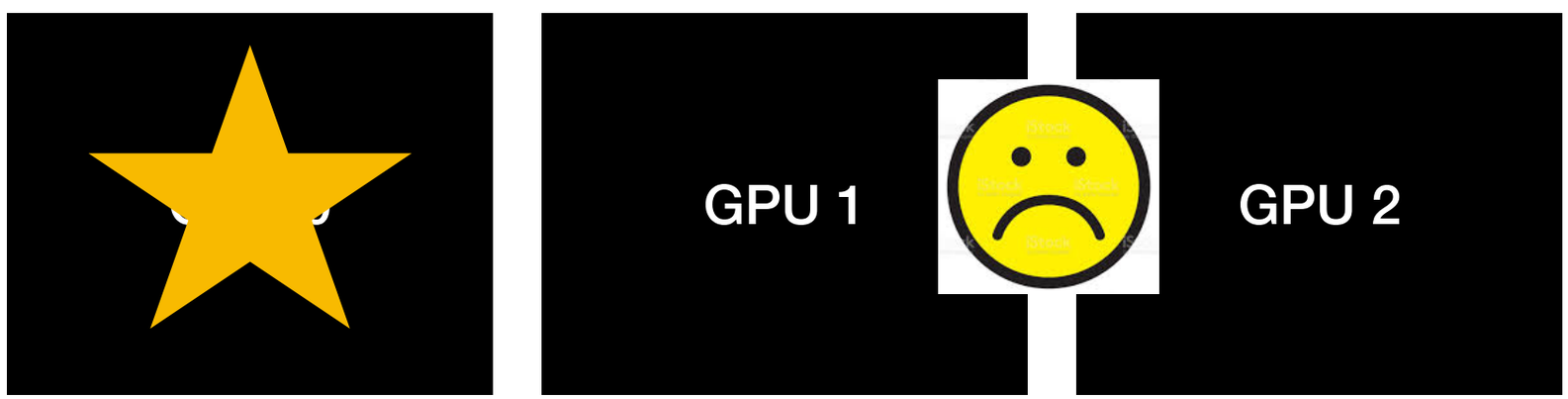
Distributed Synchronous SGD

What are some issues with Distributed Synchronous SGD?

- Bandwith needs to be high for synchronous SGD



- Requires central node



Batch size and learning rates

- With many available GPU we would want to increase the batch size to maximize

- From the point of view of variance reduction we should multiply the learning rate by sqrt(k) for k fold increase in batch size

- In practice various other rules are used most notably

> ***Linear Scaling Rule:*** *When the minibatch size is multiplied by k, multiply the learning rate by k.*

Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." *arXiv preprint arXiv:1706.02677* (2017).