Optimization for Deep Learning: Stochastic Gradient Descent

Industrial AI Lab.

Prof. Seungchul Lee

Stochastic Gradient Descent (SGD)

- We will cover gradient descent algorithm and its variants:
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-batch Gradient Descent

- We will explore the concept of these three gradient descent algorithms with a logistic regression model.

Batch Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

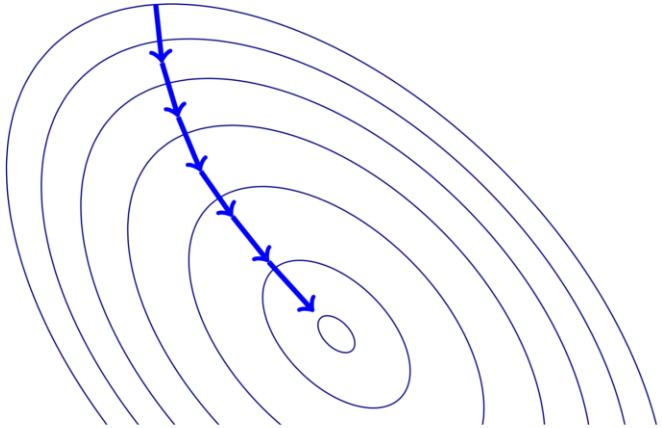
1: **while** stopping criteria not met **do**

2: Compute gradient estimate over N examples:

3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$

4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

5: **end while**



- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

Batch Gradient Descent

- So far, the cost function $\mathcal{L}$ has been the average loss over the training examples:

$$\mathcal{E}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}^{(i)} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)}, \boldsymbol{\theta}), t^{(i)}).$$

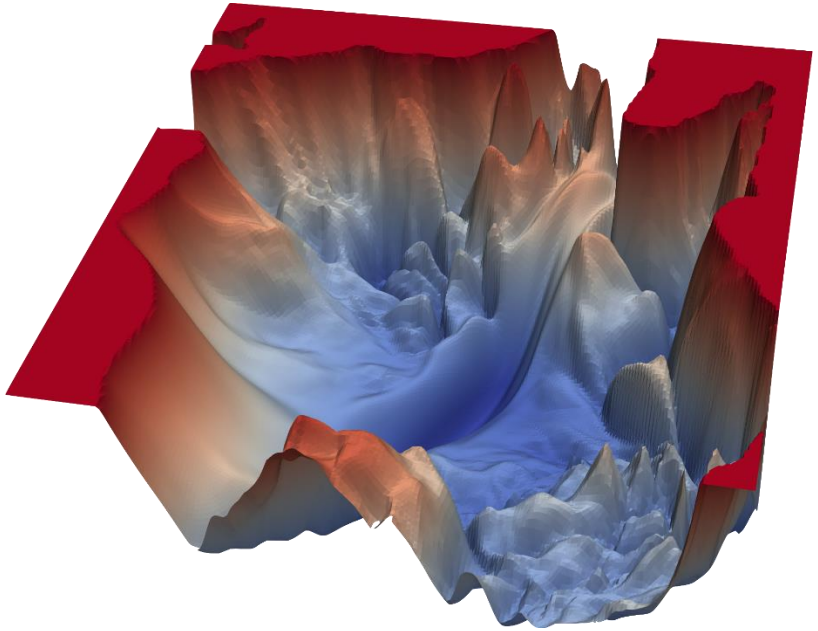
- By linearity,

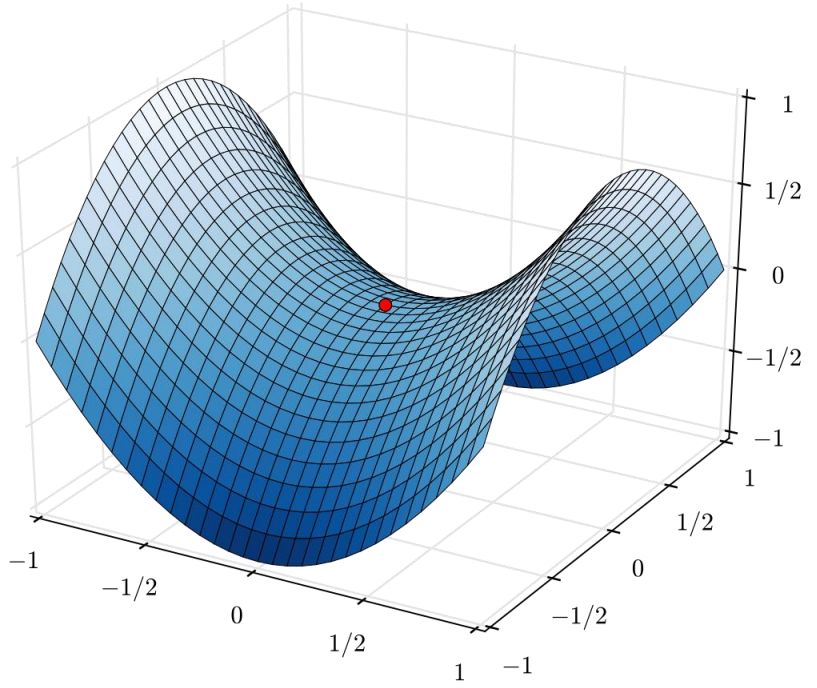
$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}.$$

- Computing the gradient requires summing over all of the training examples. This is known as batch training.
- Batch training is impractical if you have a large dataset (e.g. millions of training examples)!

Disadvantages of Batch Gradient Descent

- Still, optimizing with Batch Gradient Descent is not perfect
  - Data is often too large to compute the full gradient, so slow training
  - The loss surface is highly non-convex, so cannot compute the real gradient
  - No real guarantee that leads to a good optimum
  - No real guarantee that it will converge faster





Disadvantages of Batch Gradient Descent

- Still, optimizing with Batch Gradient Descent is not perfect

- Often loss surfaces are
  - non-quadratic
  - highly non-convex
  - very high-dimensional
- Datasets are typically really large to compute complete gradients
- No real guarantee that
  - the final solution will be good
  - we converge fast to final solution
  - or that there will be convergence

Stochastic Gradient Descent (SGD)

- Stochastic gradient descent (SGD): update the parameters based on the gradient for a single training example:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}$$

- SGD can make significant progress before it has even looked at all the data!
- Mathematical justification: if you sample a training example at random, the stochastic gradient is an unbiased estimate of the batch gradient:

$$\mathbb{E}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}\right] = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}}.$$

- Problem: if we only look at one training example at a time, we can't exploit efficient vectorized operations.

Stochastic Gradient Descent (SGD)

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

while stopping criteria not met **do**

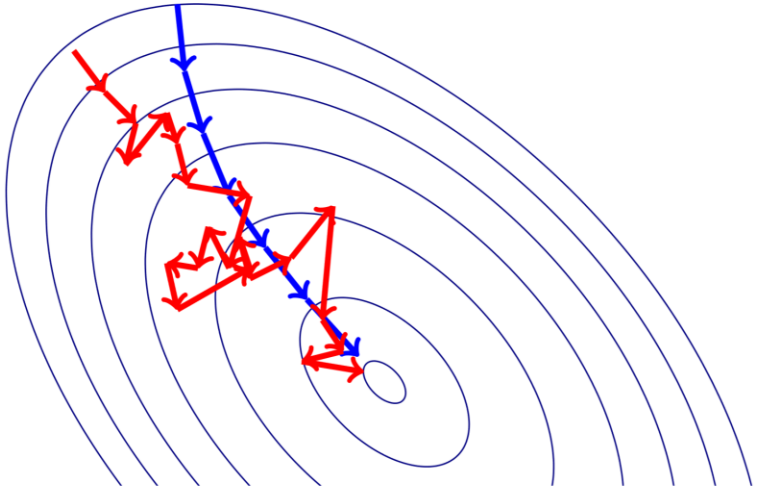
  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

  Compute gradient estimate:

  $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

  Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

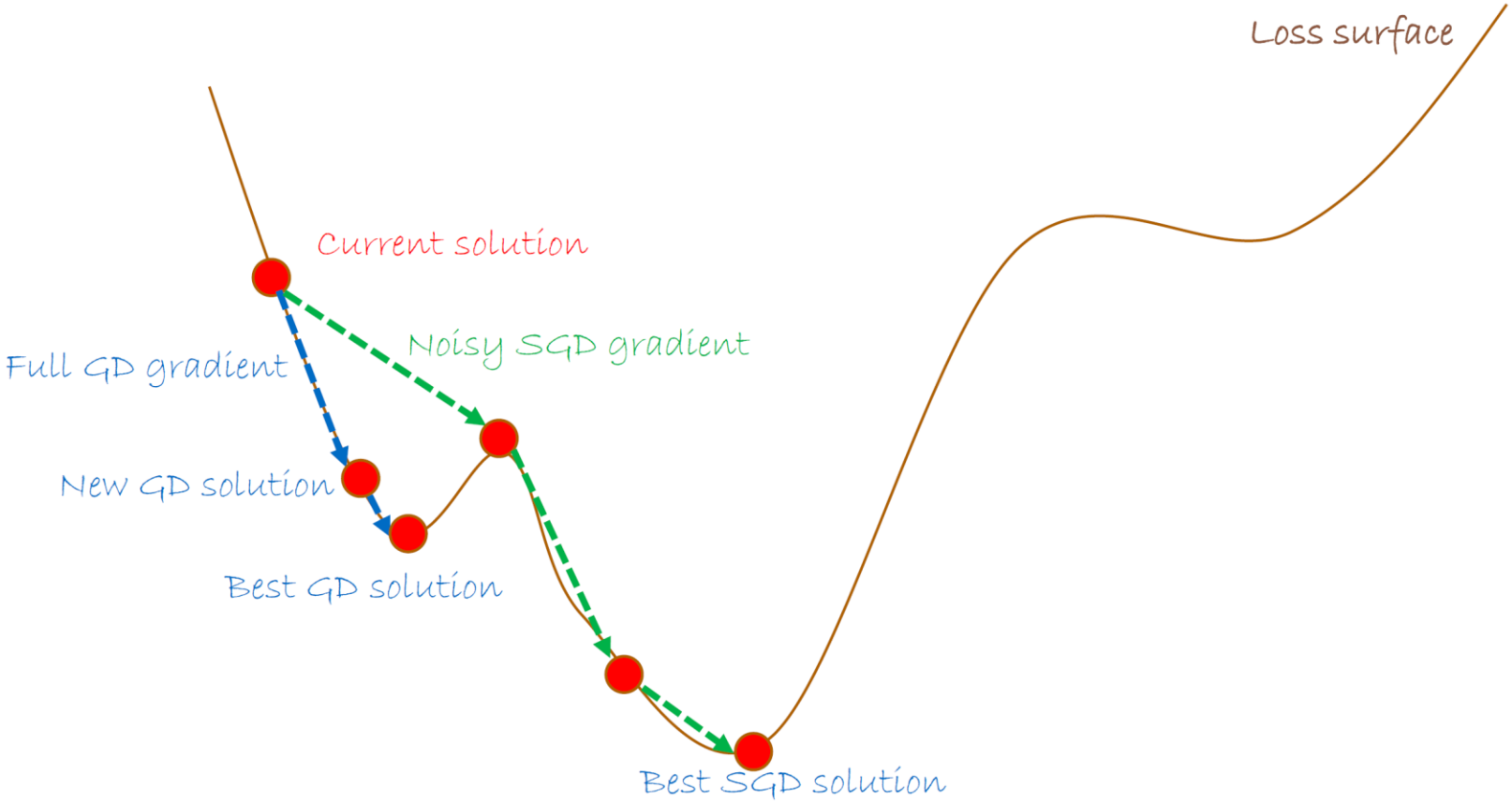
end while



- Sufficient condition to guarantee convergence:
- Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.

SGD is Sometimes Better

- No guarantee that this is what is going to always happen.
- But the noisy SGC gradients can help some times escaping local optima



Mini-batch Gradient Descent

- Compromise approach: compute the gradients on a medium-sized set of training examples, called a mini-batch.
- Each entire pass over the dataset is called an epoch.
- Stochastic gradients computed on larger mini-batches have smaller variance:

$$\mathrm{Var}\left[\frac{1}{S}\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S^2}\mathrm{Var}\left[\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S}\mathrm{Var}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right]$$

- The mini-batch size S is a hyper-parameter that needs to be set.
  - Too large: takes more memory to store the activations, and longer to compute each gradient update
  - Too small: can't exploit vectorization

Mini-batch Gradient Descent

- Potential Problem of SGD: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches

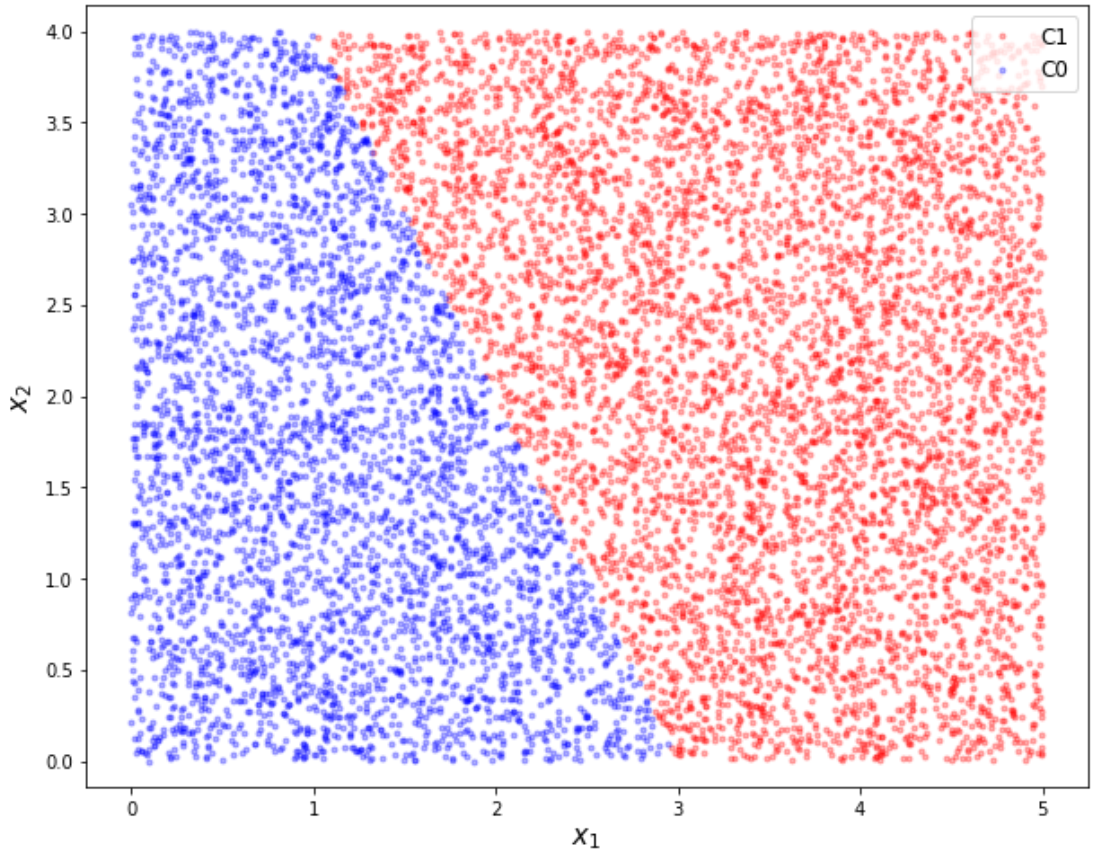
- Advantage: Computation time per update does not depend on number of training examples N

- This allows convergence on extremely large datasets

Implementation with TensorFlow

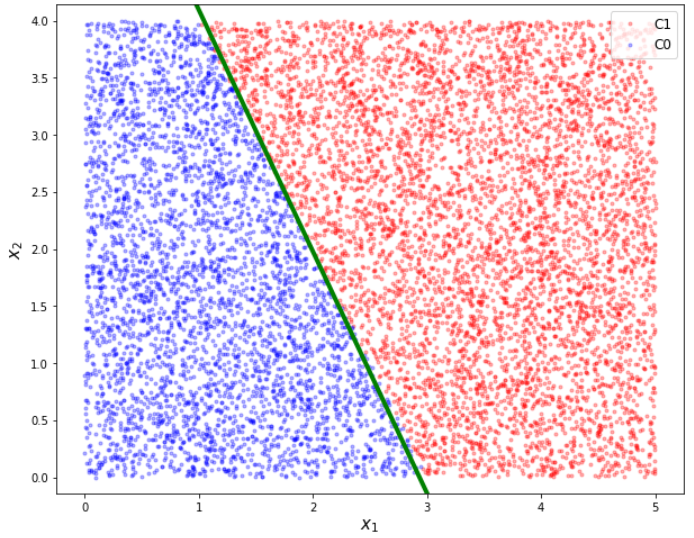
Batch Gradient Descent with TensorFlow

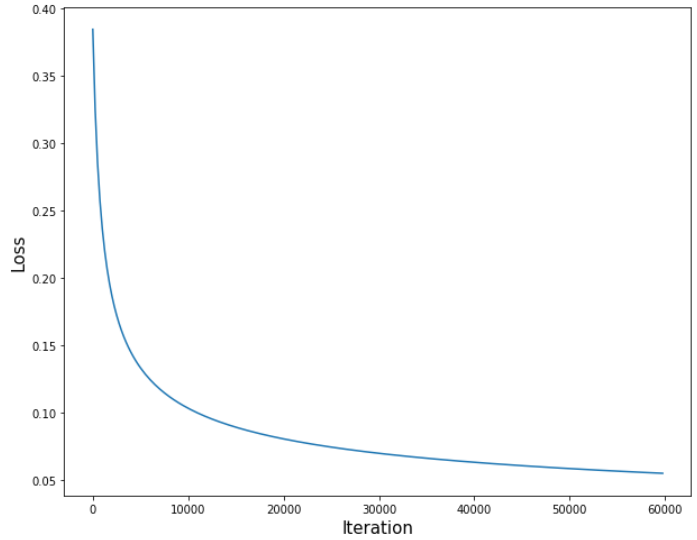
- We will explore the python codes of these three gradient descent algorithms with a logistic regression model.



Batch Gradient Descent with TensorFlow

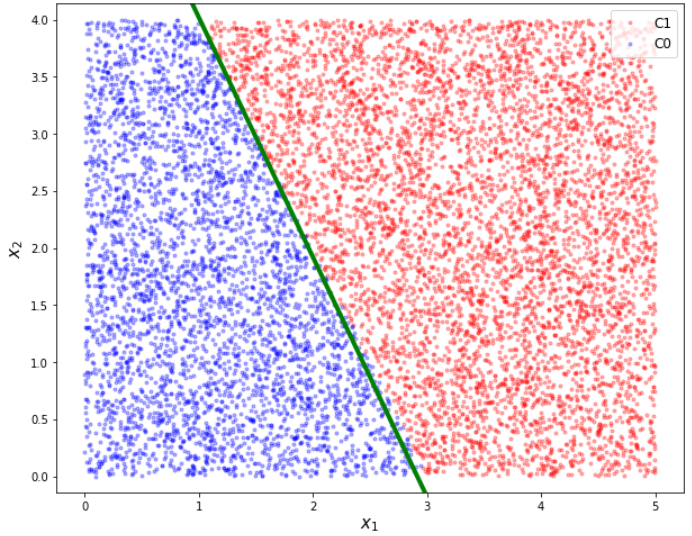
```
LR = 0.04
n_iter = 60000
n_prt = 250

x = tf.placeholder(tf.float32, [m, 3])
y = tf.placeholder(tf.float32, [m, 1])

w = tf.Variable([[0],[0],[0]], dtype = tf.float32)

y_pred = tf.matmul(x,w)
loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_pred, labels=y)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)
init = tf.global_variables_initializer()

start_time = time.time()

loss_record = []
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_iter):
        sess.run(optm, feed_dict = {x: train_X, y: train_y})

        if (epoch + 1) % n_prt == 0:
            loss_record.append(sess.run(loss, feed_dict = {x: train_X, y: train_y}))

    w_hat = sess.run(w)
```

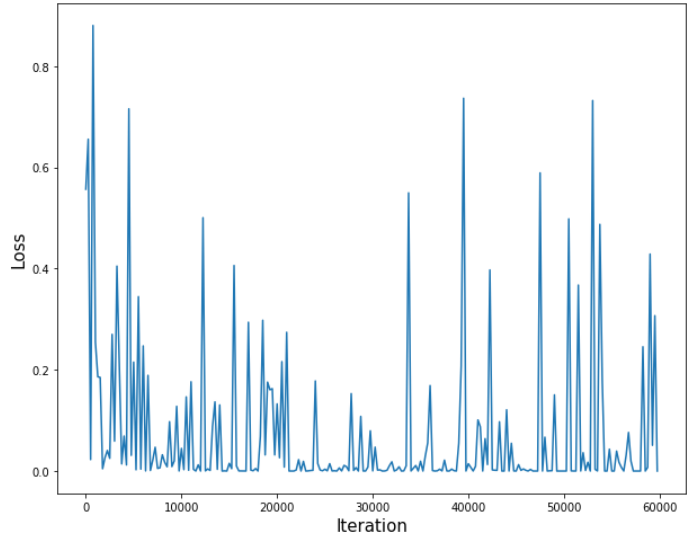




Stochastic Gradient Descent (SGD) with TensorFlow

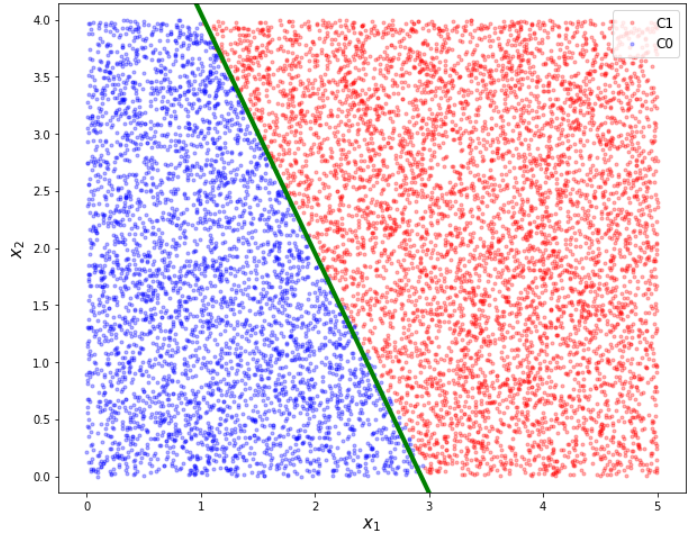
```
LR = 0.04
n_iter = 60000
n_prt = 250

x = tf.placeholder(tf.float32, [1, 3])
y = tf.placeholder(tf.float32, [1, 1])

w = tf.Variable(tf.random_normal([3,1]), dtype = tf.float32)

y_pred = tf.matmul(x,w)
loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_pred, labels=y)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)
init = tf.global_variables_initializer()

start_time = time.time()

loss_record = []
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_iter):
        idx = np.random.choice(m, 1)
        batch_X = train_X[idx,:]
        batch_y = train_y[idx]
        sess.run(optm, feed_dict = {x: batch_X, y: batch_y})

        if (epoch + 1) % n_prt == 0:
            loss_record.append(sess.run(loss, feed_dict = {x: batch_X, y: batch_y}))

    w_hat = sess.run(w)
```

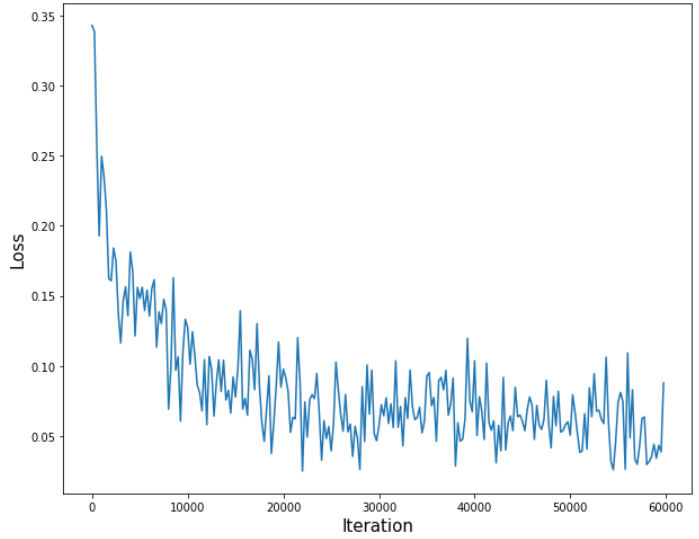




Mini-batch Gradient Descent with TensorFlow

```
LR = 0.04
n_iter = 60000
n_batch = 50
n_prt = 250

x = tf.placeholder(tf.float32, [n_batch, 3])
y = tf.placeholder(tf.float32, [n_batch, 1])

w = tf.Variable(tf.random_normal([3,1]), dtype = tf.float32)

y_pred = tf.matmul(x,w)
loss = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_pred, labels=y)
loss = tf.reduce_mean(loss)

optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)
init = tf.global_variables_initializer()

start_time = time.time()

loss_record = []
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_iter):
        idx = np.random.choice(m, size = n_batch)
        batch_X = train_X[idx,:]
        batch_y = train_y[idx]
        sess.run(optm, feed_dict = {x: batch_X, y: batch_y})

        if (epoch + 1) % n_prt == 0:
            loss_record.append(sess.run(loss, feed_dict = {x: batch_X, y: batch_y}))

    w_hat = sess.run(w)
```





Limitation of the Gradient Descent

Setting the Learning Rate

- How can we set the learning rate?

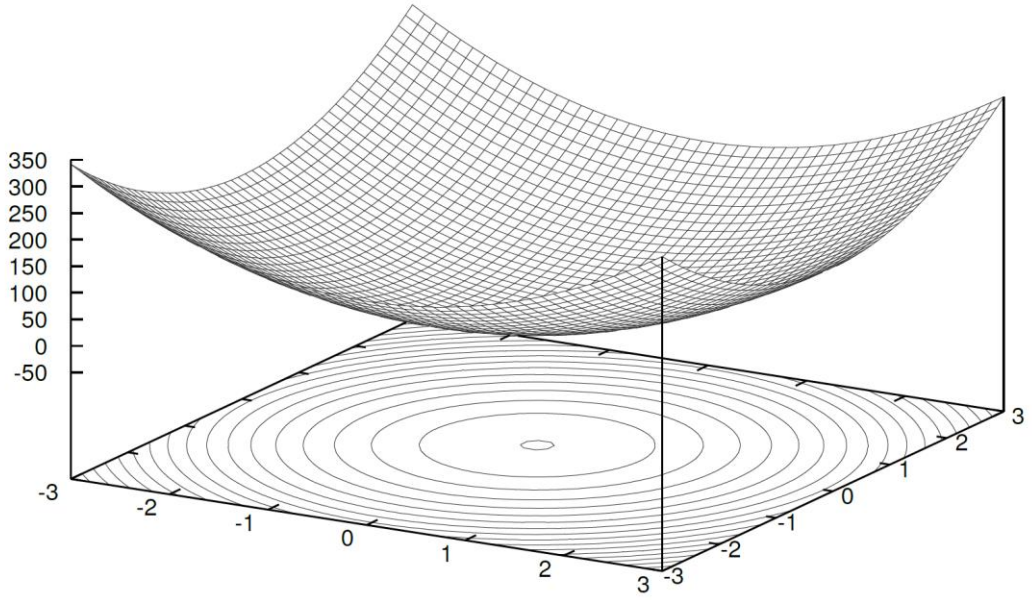
$$x \leftarrow x - \alpha \nabla_x f(x)$$

- Small learning rate converges slowly and gets stuck in false local minima
- Large learning rates overshoot, become unstable and diverge

- Idea 1
  - Try lots of different learning rates and see what works “just right”
- Idea 2
  - Do something smarter! Design an adaptive learning rate that “adapts” to the landscape
  - Temporal and spatial

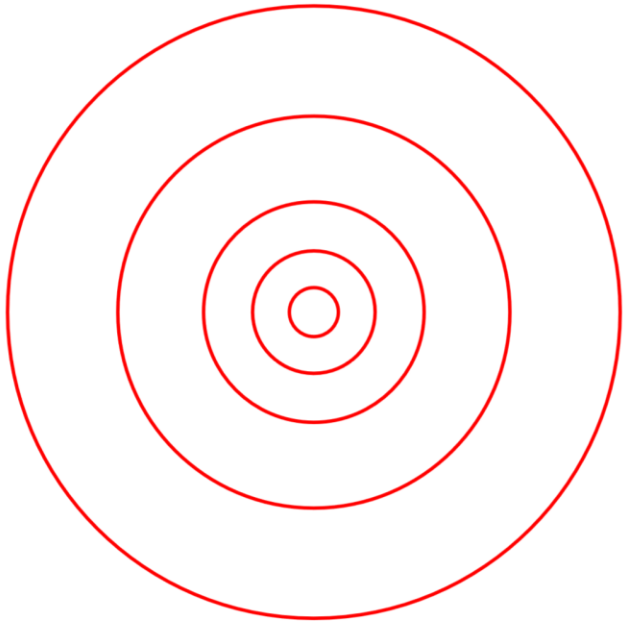
SGD Learning Rate (= Step Size)

- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

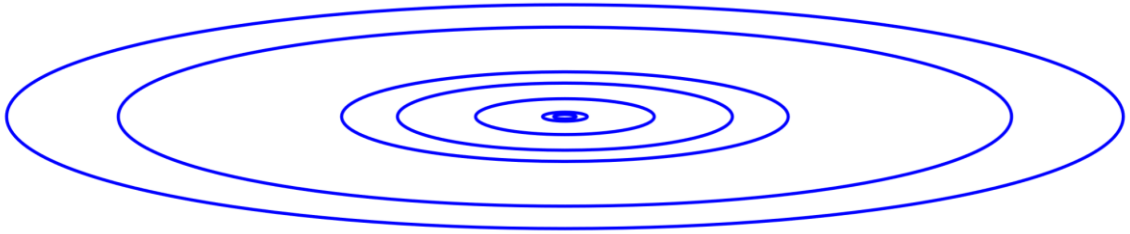


Learning Rate: Spatial

- We assign the same learning rate to all features



Nice (all features are equally important)

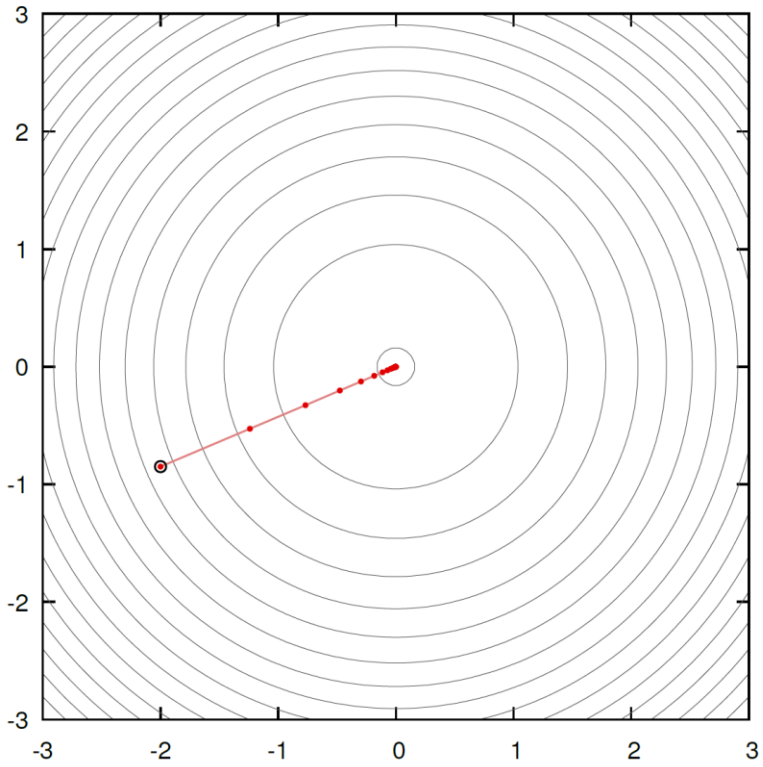


Harder !

SGD Learning Rate (= Step Size)

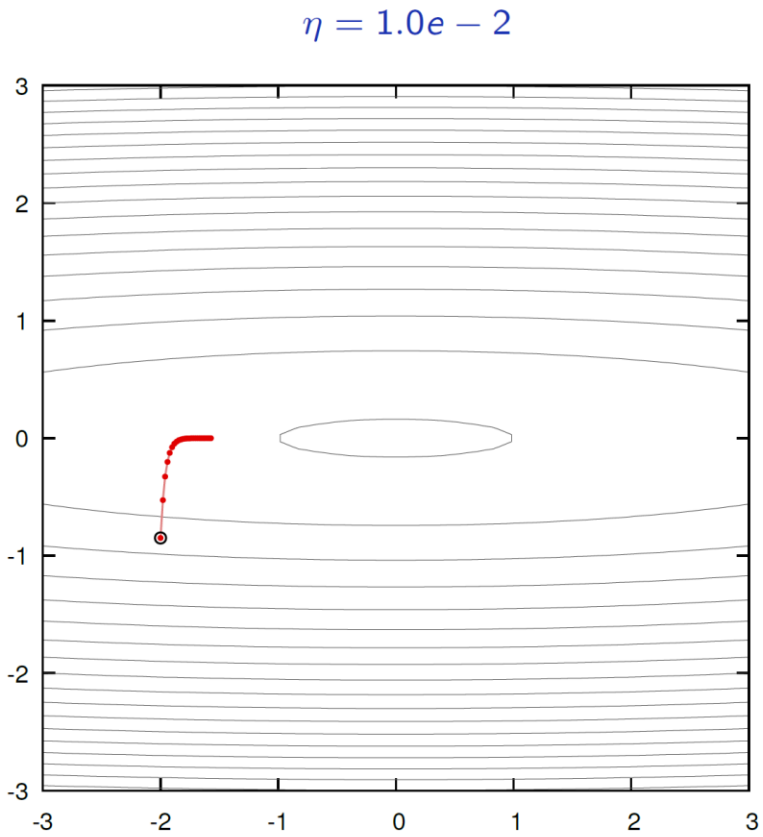
- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.
- Nice (all features are equally important)

$\eta = 1.0e-2$



SGD Learning Rate (= Step Size)

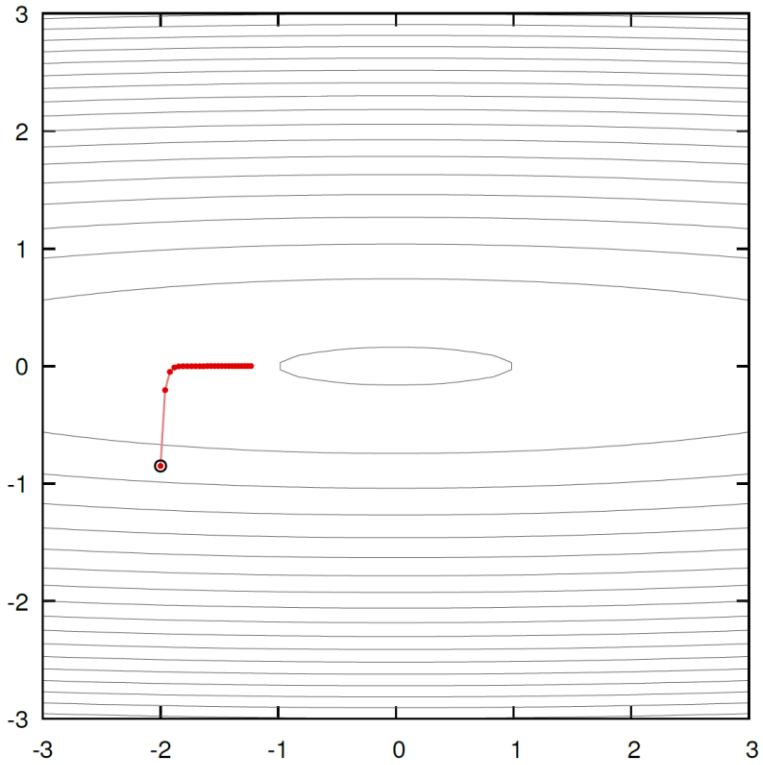
- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.
- Harder !



SGD Learning Rate (= Step Size)

- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.
- Harder !

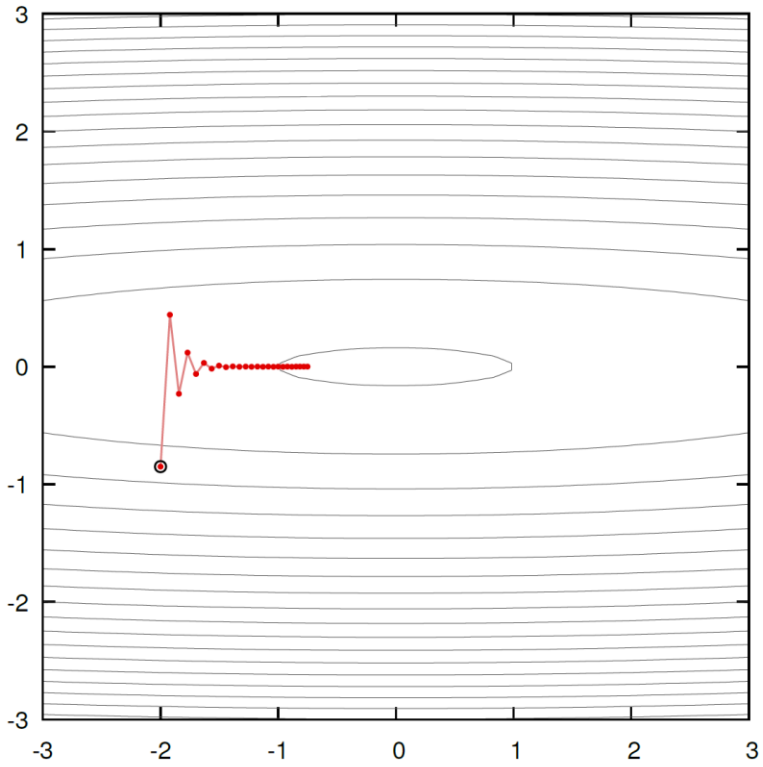
$\eta = 2.0e-2$



SGD Learning Rate (= Step Size)

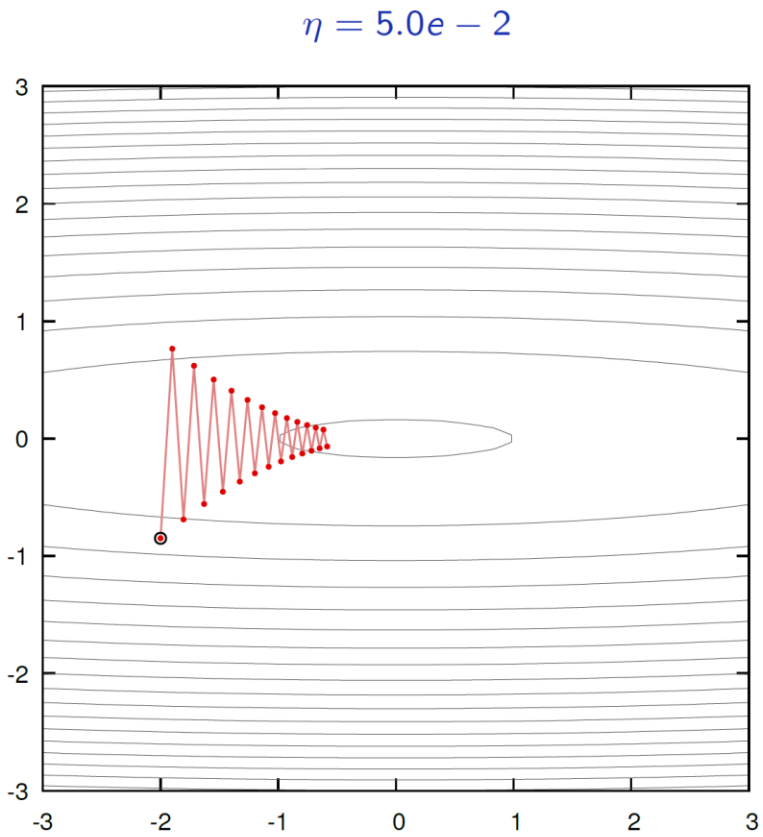
- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.
- Harder !

$\eta = 4.0e-2$



SGD Learning Rate (= Step Size)

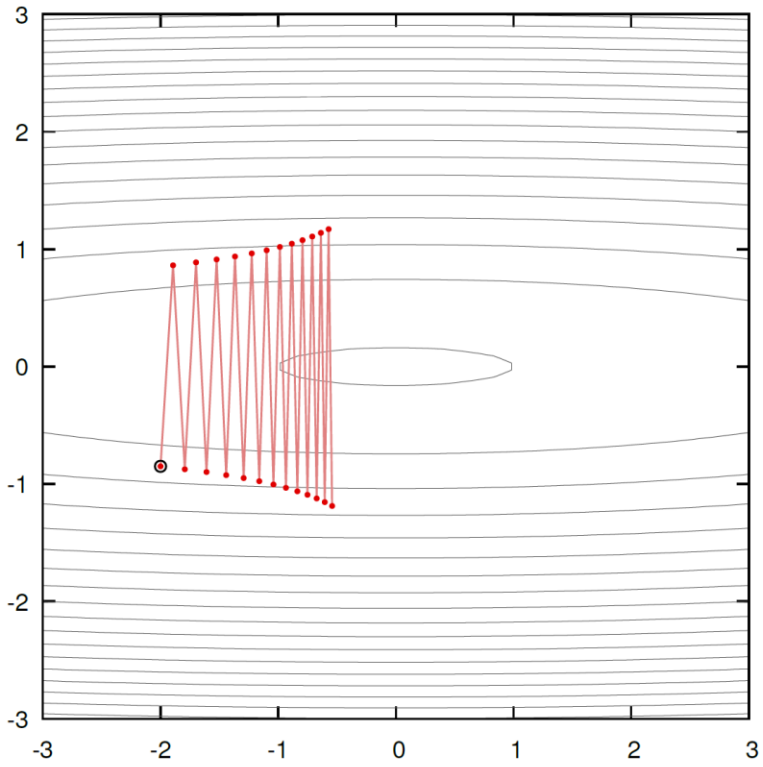
- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.
- Harder !



SGD Learning Rate (= Step Size)

- The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.
- Harder !

$\eta = 5.3e - 2$



SGD Learning Rate (= Step Size)

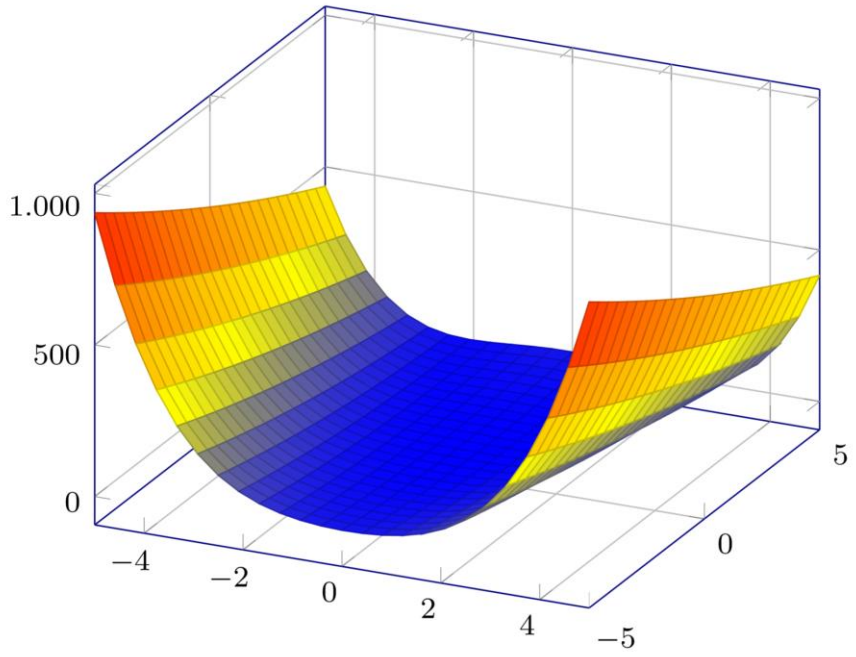
- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations

- Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.
  - However for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worst.

- Deep-learning generally relies on a smarter use of the gradient, using statistics over its past values to make a “smarter step” with the current one.

Momentum

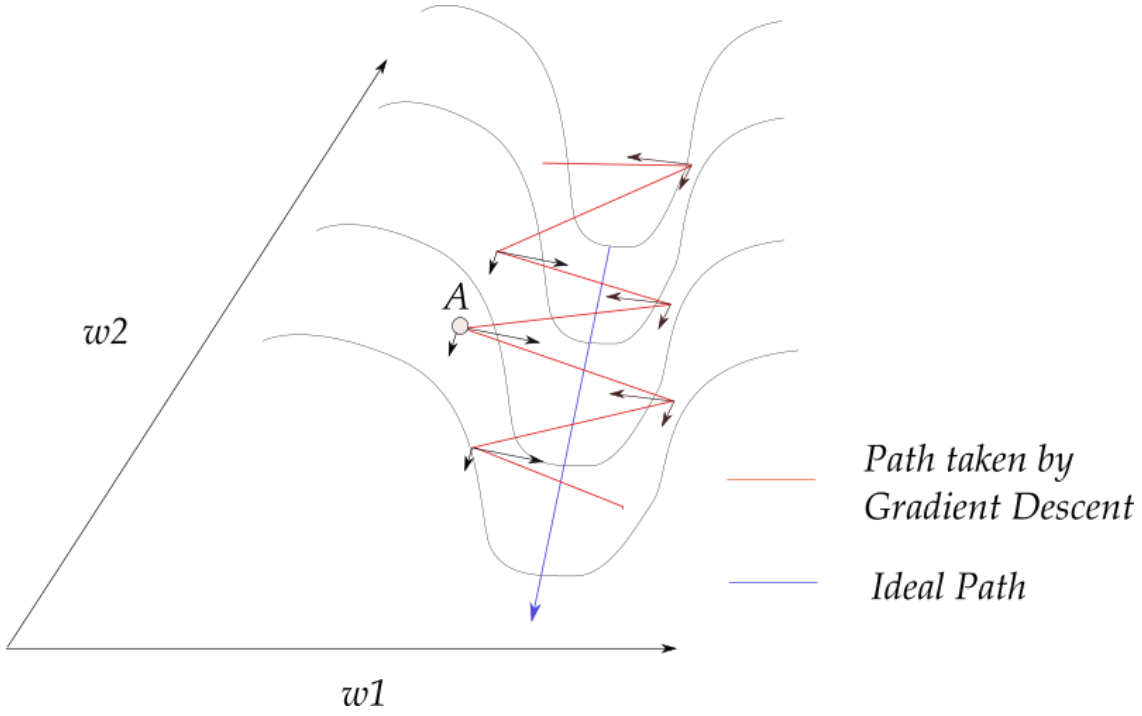
- The Momentum method is a method to accelerate learning using SGD
- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - We get small but consistent gradients
  - The gradients are very noisy



- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

Momentum

- How do we try and solve this problem?
- Introduce a new variable v, the velocity
- We think of v as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an exponentially decaying moving average of the negative gradients



Mini-batch SGD with Momentum

- The “vanilla” mini-batch stochastic gradient descent (SGD) consists of

$$w_{t+1} = w_t - \eta g_t,$$

where

$$g_t = \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t)$$

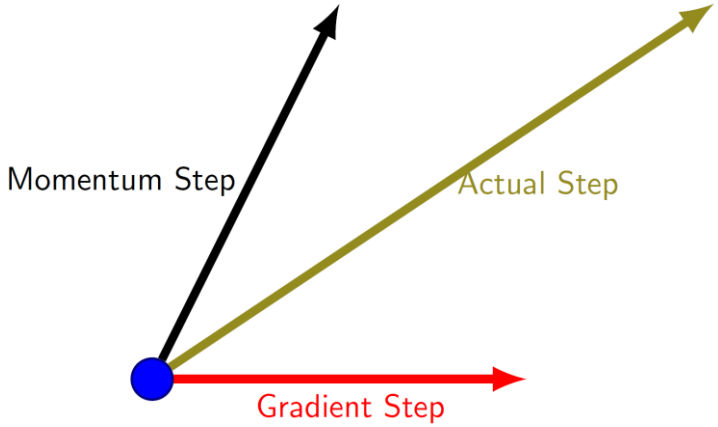
is the gradient summed over a mini-batch.

Mini-batch SGD with Momentum

- The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

$$u_t = \gamma u_{t-1} + \eta g_t$$
$$w_{t+1} = w_t - u_t.$$

- With $\gamma = 0$, this is the same as vanilla SGD.

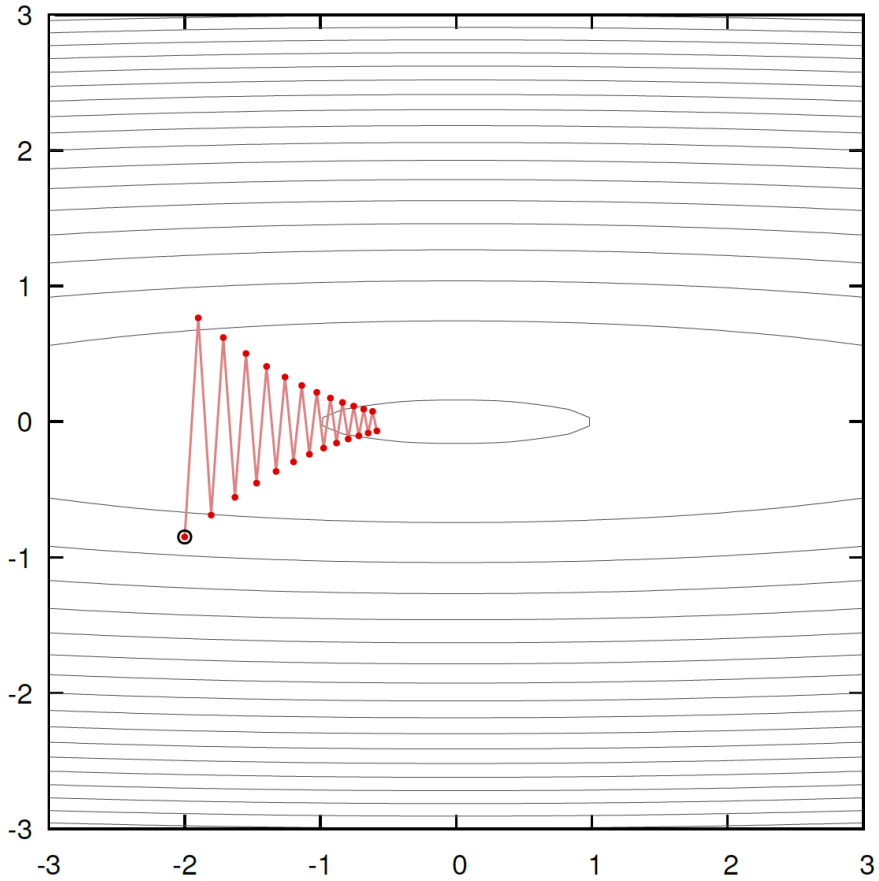


- With $\gamma > 0$, this update has three nice properties:
  - it can "go through" local barriers
  - it accelerates if the gradient does not change much:
  - it dampens oscillations in narrow valleys.

$$(u = \gamma u + \eta g) \Rightarrow \left(u = \frac{\eta}{1-\gamma} g\right),$$

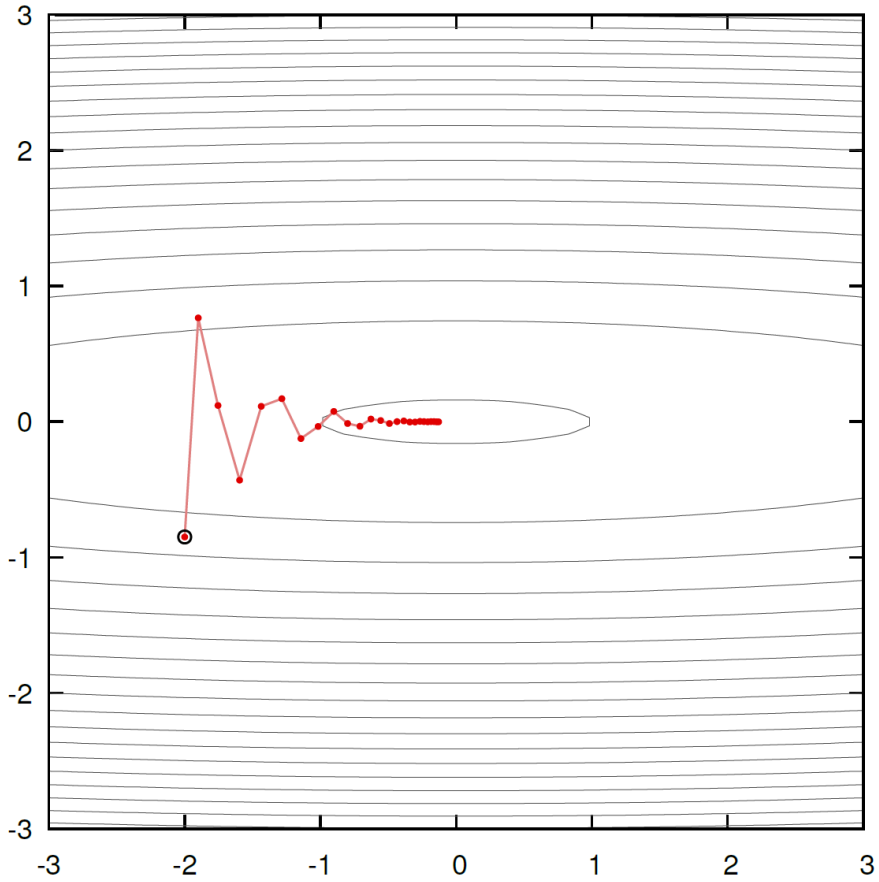
Mini-batch SGD with Momentum

$\eta = 5.0e-2, \gamma = 0$



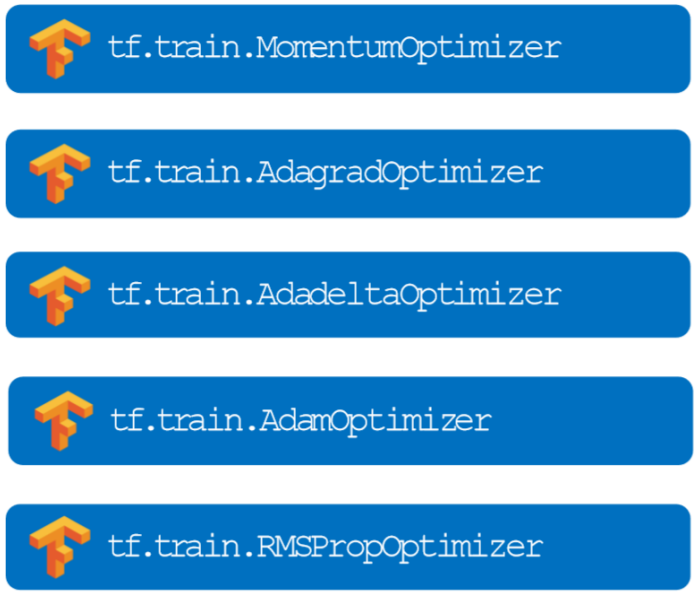
Mini-batch SGD with Momentum

$\eta = 5.0e-2, \gamma = 0.5$



Adaptive Learning Rate Methods

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp



Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

- Additional detail: http://ruder.io/optimizing-gradient-descent/