CS490/590 Lecture 8: Optimization

Eren Gultepe
Department of Computer Science SIUE

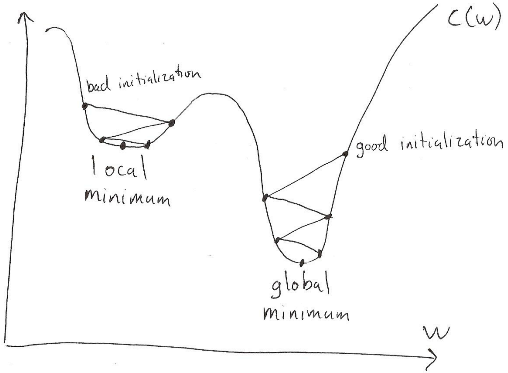
Adapted from Roger Grosse

Overview

- We've talked a lot about how to compute gradients. What do we actually do with them?
- Today's lecture: various things that can go wrong in gradient descent, and what to do about them.
- Let's take a break from equations and think intuitively.
- Let's group all the parameters (weights and biases) of our network into a single vector $\boldsymbol{\theta}$.

Optimization

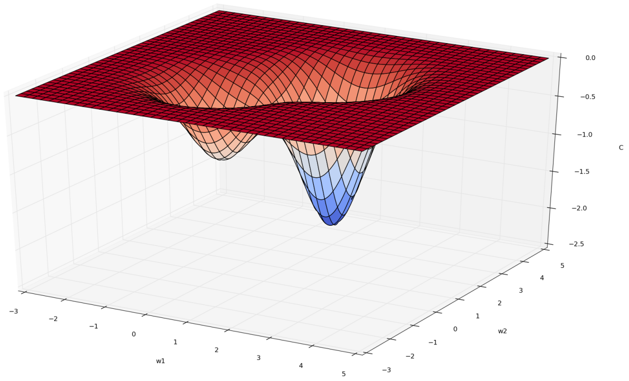
Visualizing gradient descent in one dimension: $w \leftarrow w - \epsilon \frac{\mathrm{d}\mathcal{E}}{\mathrm{d}w}$



- The regions where gradient descent converges to a particular local minimum are called basins of attraction.

Optimization

Visualizing two-dimensional optimization problems is trickier. Surface plots can be hard to interpret:



Optimization

Recall:

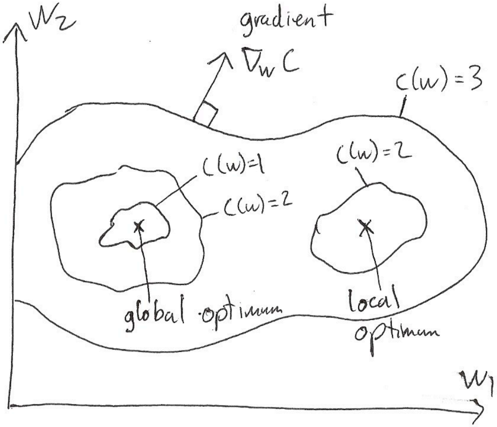
- Level sets (or contours): sets of points on which $\mathcal{E}(\boldsymbol{\theta})$ is constant
- Gradient: the vector of partial derivatives

$$\nabla_{\boldsymbol{\theta}}\mathcal{E} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = \left(\frac{\partial \mathcal{E}}{\partial \theta_1}, \frac{\partial \mathcal{E}}{\partial \theta_2} \right)$$

  - points in the direction of maximum increase
  - orthogonal to the level set

- The gradient descent updates are opposite the gradient direction.

Optimization



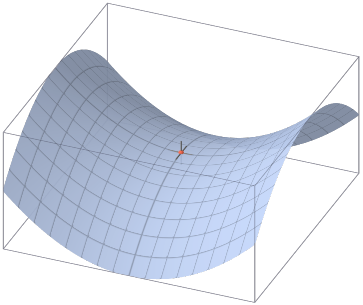
Local Minima

- Recall: convex functions don't have local minima. This includes linear regression and logistic regression.
- But neural net training is not convex!
  - Reason: if a function f is convex, then for any set of points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in its domain ,

$$f(\lambda_1 \mathbf{x}_1 + \cdots + \lambda_N \mathbf{x}_N) \leq \lambda_1 f(\mathbf{x}_1) + \cdots + \lambda_N f(\mathbf{x}_N) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1.$$

  - Neural nets have a weight space symmetry: we can permute all the hidden units in a given layer and obtain an equivalent solution.
  - Suppose we average the parameters for all $K!$ permutations. Then we get a degenerate network where all the hidden units are identical.
  - If the cost function were convex, this solution would have to be better than the original one, which is ridiculous!
- Even though any multilayer neural net can have local optima, we usually don't worry too much about them.

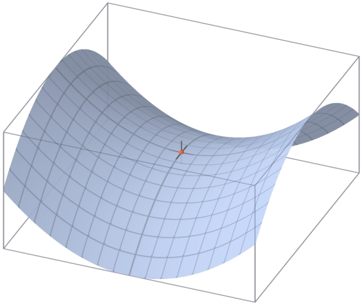
Saddle points



At a saddle point $\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = 0$, even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

When would saddle points be a problem?

Saddle points



At a saddle point $\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = 0$, even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

When would saddle points be a problem?

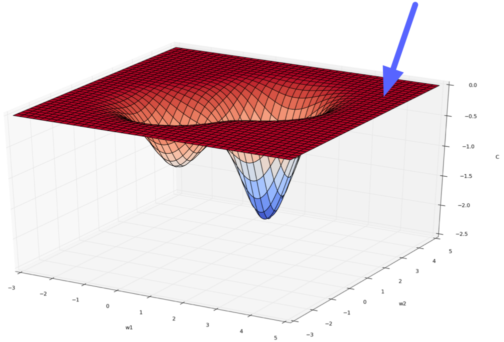
- If we're exactly on the saddle point, then we're stuck.
- If we're slightly to the side, then we can get unstuck.

Saddle points

- Suppose you have two hidden units with identical incoming and outgoing weights.
- After a gradient descent update, they will still have identical weights. By induction, they'll always remain identical.
- But if you perturbed them slightly, they can start to move apart.
- Important special case: don't initialize all your weights to zero!
    - Instead, use small random values.

Plateaux

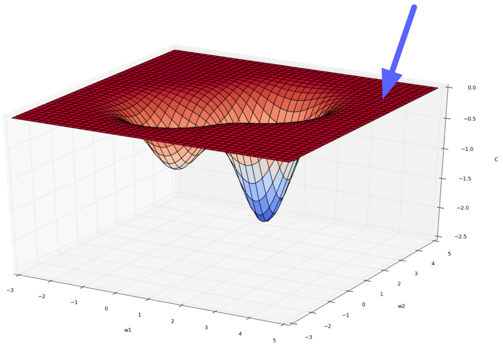
A flat region is called a plateau. (Plural: plateaux)



Can you think of examples?

Plateaux

A flat region is called a plateau. (Plural: plateaux)



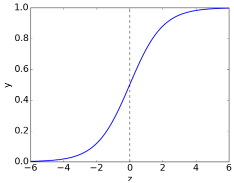
Can you think of examples?

- 0–1 loss
- hard threshold activations
- logistic activations & least squares

Plateaux

- An important example of a plateau is a saturated unit. This is when it is in the flat region of its activation function. Recall the backprop equation for the weight derivative:

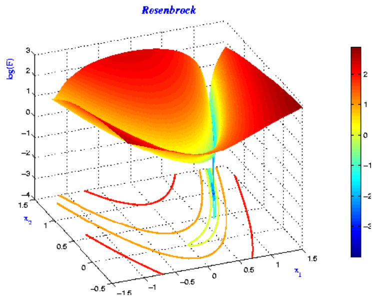
$$\begin{aligned} \overline{z_i} &= \overline{h_i}\,\phi'(z) \\ \overline{w_{ij}} &= \overline{z_i}\,x_j \end{aligned}$$

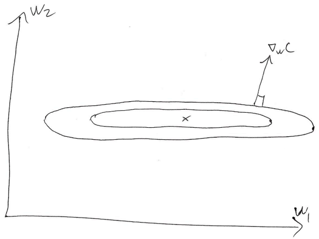


- If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be *exactly* 0. We call this a dead unit.

Ravines

Long, narrow ravines:





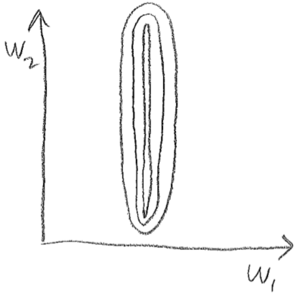
Lots of sloshing around the walls, only a small derivative along the slope of the ravine's floor.

Ravines

- Suppose we have the following dataset for linear regression.

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
$\vdots$	$\vdots$	$\vdots$

$$\overline{w_i} = \overline{y}\, x_i$$

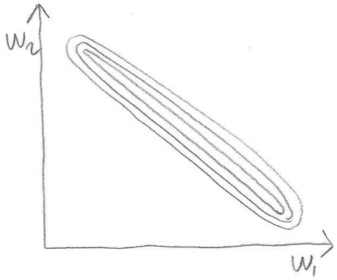


- Which weight, w_1 or w_2, will receive a larger gradient descent update?
- Which one do you want to receive a larger update?
- Note: the figure vastly *understates* the narrowness of the ravine!

Ravines

- Or consider the following dataset:

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
⋮	⋮	⋮



Ravines

- To avoid these problems, it's a good idea to center your inputs to zero mean and unit variance, especially when they're in arbitrary units (feet, seconds, etc.).

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

- Hidden units may have non-centered activations, and this is harder to deal with.
    - One trick: replace logistic units (which range from 0 to 1) with tanh units (which range from -1 to 1)
    - A recent method called **batch normalization** explicitly centers each hidden activation. It often speeds up training by 1.5-2x, and it's available in all the major neural net frameworks.

Momentum

- Unfortunately, even with these normalization tricks, narrow ravines will be a fact of life. We need algorithms that are able to deal with them.
- Momentum is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$\mathbf{p} \leftarrow \mu \mathbf{p} - \alpha \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?

Momentum

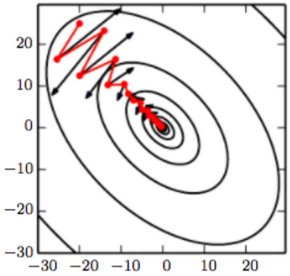
- Unfortunately, even with these normalization tricks, narrow ravines will be a fact of life. We need algorithms that are able to deal with them.
- Momentum is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$\mathbf{p} \leftarrow \mu\mathbf{p} - \alpha \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?
  - If $\mu = 1$, conservation of energy implies it will never settle down.

Momentum

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.
- In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.



- If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of

$$-\frac{\alpha}{1-\mu} \cdot \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}}$$

  This suggests if you increase μ, you should lower α to compensate.

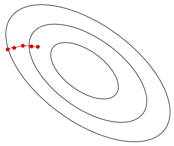
- Momentum sometimes helps a lot, and almost never hurts.

Ravines

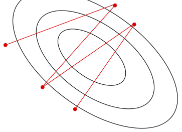
- Even with momentum and normalization tricks, narrow ravines are still one of the biggest obstacles in optimizing neural networks.
- Empirically, the curvature can be many orders of magnitude larger in some directions than others!
- An area of research known as second-order optimization develops algorithms which explicitly use curvature information (second derivatives), but these are complicated and difficult to scale to large neural nets and large datasets.
- There is an optimization procedure called Adam which uses just a little bit of curvature information and often works much better than gradient descent. It's available in all the major neural net frameworks.

Learning Rate

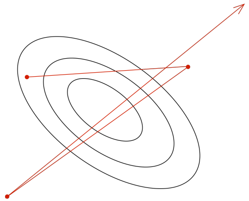
- The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong in batch mode:



α too small:
slow progress



α too large:
oscillations

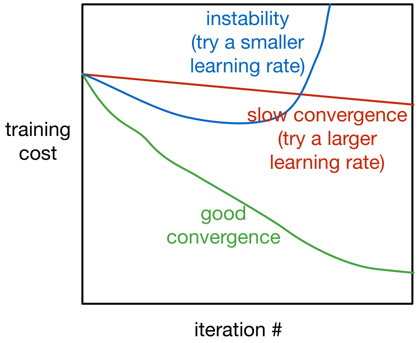


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).

Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.



- Warning: it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.

Metrics for Binary classification

- Recall that the average of 0–1 loss is the error rate, or fraction incorrectly classified.
  - We noted we couldn't optimize it, but it's still a useful metric to track.
  - Equivalently, we can track the accuracy, or fraction correct.
  - Typically, the error rate behaves similarly to the cross-entropy loss, but this isn't always the case.
- Another way to break down the accuracy:
  - P=num positive; N=num negative; TP=true positives; TN=true negatives
  - FP=false positive or a type I error
  - FN=false negative or a type II error

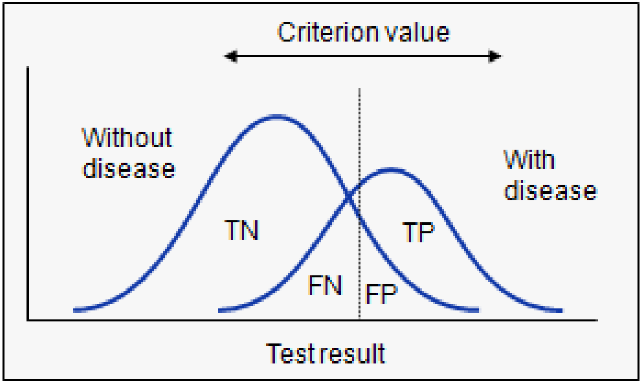
$$Acc = \frac{TP + TN}{P + N} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Discuss:** When might accuracy present a misleading picture of performance?

The limitations of accuracy

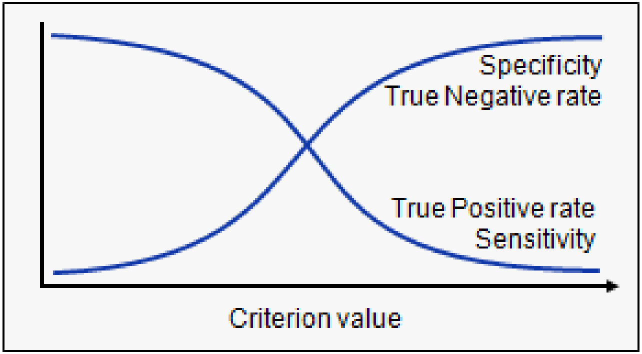
- Accuracy is highly sensitive to class imbalance.
  - Suppose you're trying to screen patients for a particular disease, and under the data generating distribution, 1% of patients have that disease.
  - How can you achieve 99% accuracy?
  - You are able to observe a feature which is 10x more likely in a patient who has cancer. Does this improve your accuracy?
- Sensitivity and specificity are useful metrics even under class imbalance.
  - Sensitivity $= \frac{TP}{TP+FN}$ [True positive rate]
  - Specificity $= \frac{TN}{TN+FP}$ [True negative rate]
  - What happens if our classification problem is not truly (log-)linearly seperable?
  - How do we pick a threshold for $y = \sigma(x)$?

Designing diagnostic tests



- You've developed a binary prediction model to indicate whether someone has a specific disease
- What happens to sensitivity and specificity as you slide the threshold from left to right?

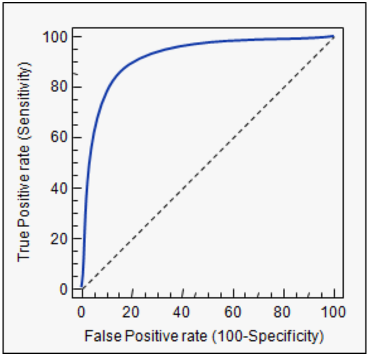
Sensitivity and specificity



- Tradeoff between sensitivity and specificity

Receiver Operating Characteristic (ROC) curve

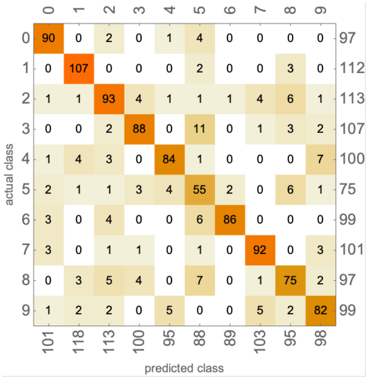
Receiver Operating Characteristic (ROC) curve



- y axis: sensitivity
- x axis: 100-specificity
- Area under the ROC curve (AUC) is a useful metric to track if a binary classifier achieves a good tradeoff between sensitivity and specificity.

Metrics for Multi-Class classification

- You might also be interested in how frequently certain classes are confused.
- Confusion matrix: $K \times K$ matrix; rows are true labels, columns are predicted labels, entries are frequencies
- Question: what does the confusion matrix look like if the classifier is perfect?



actual class	0	1	2	3	4	5	6	7	8	9	
0	90	0	2	0	1	4	0	0	0	0	97
1	0	107	0	0	0	2	0	0	3	0	112
2	1	1	93	4	1	1	1	4	6	1	113
3	0	0	2	88	0	11	0	1	3	2	107
4	1	4	3	0	84	1	0	0	0	7	100
5	2	1	1	3	4	55	2	0	6	1	75
6	3	0	4	0	0	6	86	0	0	0	99
7	3	0	1	1	0	1	0	92	0	3	101
8	0	3	5	4	0	7	0	1	75	2	97
9	1	2	2	0	5	0	0	5	2	82	99
	101	118	113	100	95	88	89	103	95	98	

predicted class

Stochastic Gradient Descent

- So far, the cost function $\mathcal{E}$ 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)!

Stochastic Gradient Descent

- 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

- 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 hyperparameter 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
  - A reasonable value might be $S = 100$.

Stochastic Gradient Descent: Batch Size

- The mini-batch size S is a hyperparameter that needs to be set.
  - **Large batches:** converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches:** perform more weight updates per second because each one requires less computation.
- **Claim:** If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal.
  - 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$.
  - Rewrite minibatch gradient descent as a for-loop:

S = 1

For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_{k-1})$$

S = 100

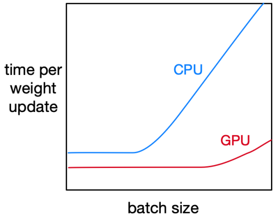
For $k = 1, \ldots, 100$:

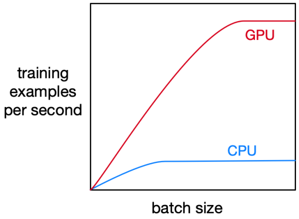
$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \frac{\alpha}{100} \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_{0})$$

  - All else being equal, you'd prefer to compute the gradient at a fresher value of $\boldsymbol{\theta}$. So $S = 1$ is better.

Stochastic Gradient Descent: Batch Size

- The reason we don't use $S = 1$ is that larger batches can take advantage of fast matrix operations and parallelism.
- **Small batches:** An update with $S = 10$ isn't much more expensive than an update with $S = 1$.
- **Large batches:** Once S is large enough to saturate the hardware efficiencies, the cost becomes linear in S.
- Cartoon figure, not drawn to scale:

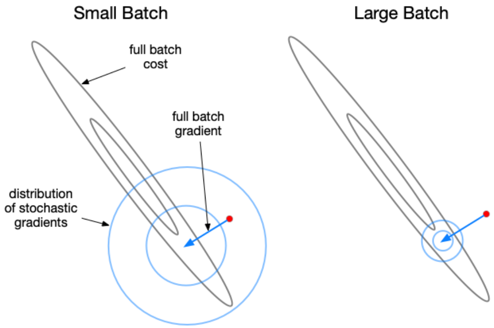


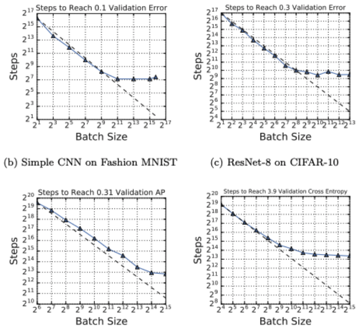


- Since GPUs afford more parallelism, they saturate at a larger batch size. Hence, GPUs tend to favor larger batch sizes.

Stochastic Gradient Descent: Batch Size

- The convergence benefits of larger batches also see diminishing returns.
- **Small batches:** large gradient noise, so large benefit from increased batch size
- **Large batches:** SGD approximates the batch gradient descent update, so no further benefit from variance reduction.





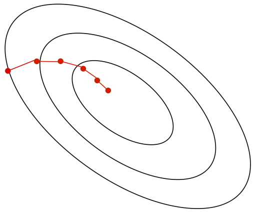
(b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

(e) ResNet-50 on Open Images (f) Transformer on LM1B

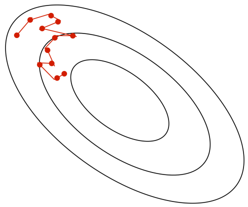
- **Right:** # iterations to reach target validation error as a function of batch size. (Shallue et al., 2018)

Stochastic Gradient Descent

- Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.



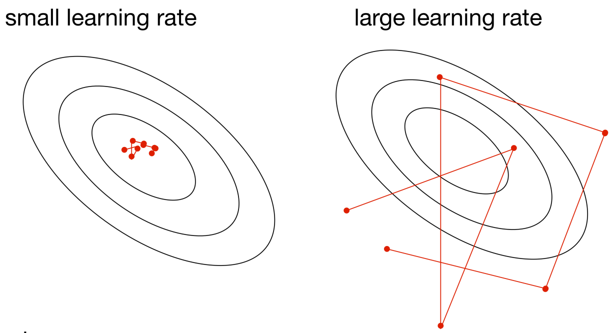
batch gradient descent



stochastic gradient descent

SGD Learning Rate

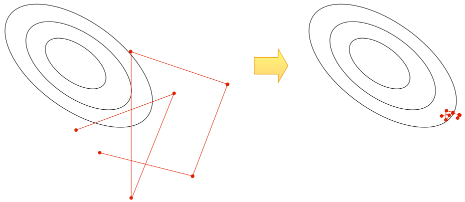
- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

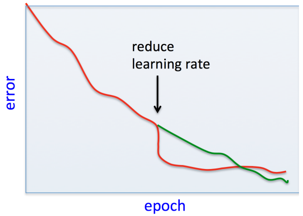


- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations

SGD Learning Rate

- Warning: by reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly. But this can come at the expense of long-run performance.





RMSprop and Adam

- Recall: SGD takes large steps in directions of high curvature and small steps in directions of low curvature.
- RMSprop is a variant of SGD which rescales each coordinate of the gradient to have norm 1 on average. It does this by keeping an exponential moving average s_j of the squared gradients.
- The following update is applied to each coordinate j independently:

$$s_j \leftarrow (1-\gamma)s_j + \gamma[\tfrac{\partial \mathcal{J}}{\partial \theta_j}]^2$$
$$\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{s_j + \epsilon}} \frac{\partial \mathcal{J}}{\partial \theta_j}$$

- If the eigenvectors of the Hessian are axis-aligned (dubious assumption), then RMSprop can correct for the curvature. In practice, it typically works slightly better than SGD.
- Adam = RMSprop + momentum
- Both optimizers are included in TensorFlow, Pytorch, etc.

Recap

Problem	**Diagnostics**	**Workarounds**
incorrect gradients	finite differences	fix them, or use autodiff
local optima	(hard)	random restarts
symmetries	visualize $\mathbf{W}$	initialize $\mathbf{W}$ randomly
slow progress	slow, linear training curve	increase α; momentum
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α; momentum
fluctuations	fluctuations in training curve	decay α; iterate averaging
dead/saturated units	activation histograms	initial scale of $\mathbf{W}$; ReLU
ill-conditioning	(hard)	normalization; momentum; Adam; second-order opt.