Optimization

Tripp Deep Learning F23

With algorithm listings from Goodfellow et al., Deep Learning

UNIVERSITY OF WATERLOO

TODAY'S GOAL

By the end of the class, you should understand the challenges of optimizing deep networks, how the most widely used network optimization methods work, and how to find good hyperparameters.

Summary

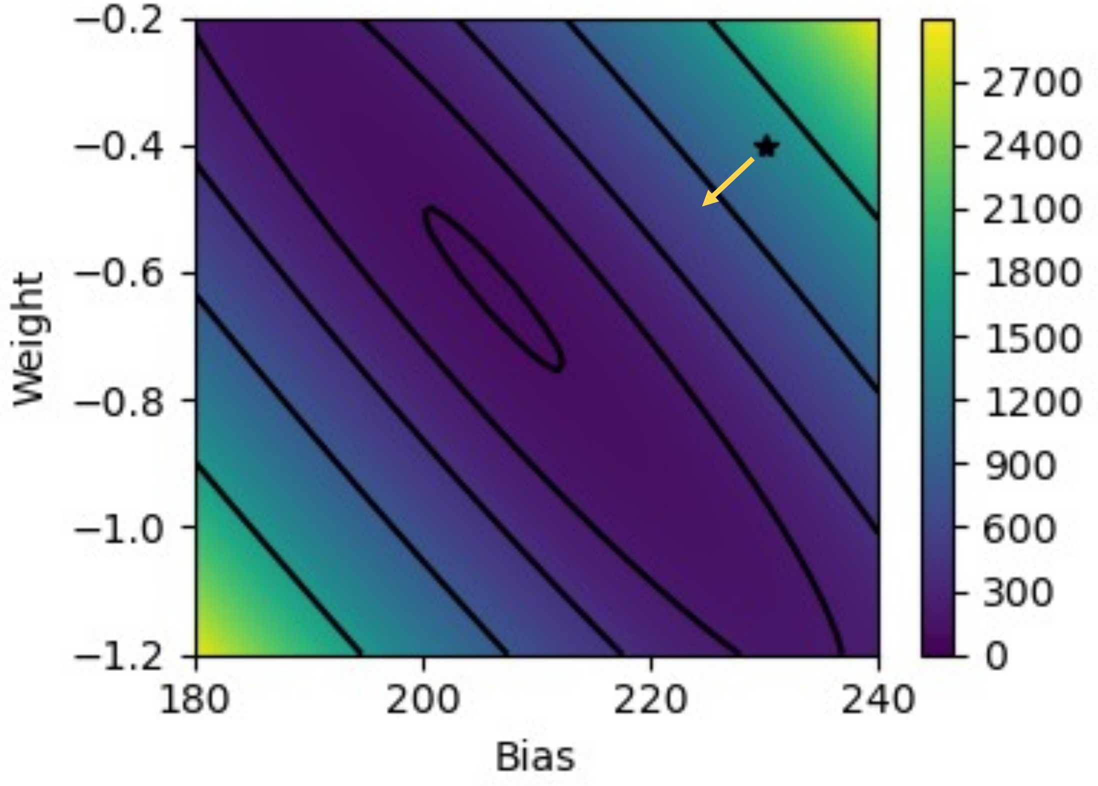
1. Challenges include high dimensionality, non-convexity, poor conditioning, and gradient cliffs
2. Deep networks probably do not suffer much from local minima
3. Parameter initialization should break symmetry and encourage stability
4. Stochastic gradient descent makes efficient use of data
5. Momentum helps find directions with small but consistent gradients
6. Adaptive algorithms reduce updates in steep directions
7. Batch normalization reduces coupling between parameters in different layers
8. Hyperparameters can be tuned by grid search, random search, or search algorithms
9. Hyperparameters can be manually tuned according to their typical effects

Terminology: Optimization vs. training

- The words are closely related but have slightly different meanings
  - Training means providing instructions and/or experiences that improve task performance
  - Optimization means finding parameters that minimize (or maximize) a function
- We train deep networks using optimization methods
  - We *train* a network to have good *test performance* by *optimizing training loss*
  - We don’t succeed at optimization in the sense that we find the global optimum – the parameter space is too large to know whether we’ve found it, and anyway it would often involve overfitting
- We won’t worry too much about these distinctions

Gradient descent

- The gradient of a scalar function of multiple parameters is a vector field that indicates the direction and rate of fastest increase of the function at each point, $\boldsymbol{\theta}$, in the parameter space
- We descend the gradient of the loss, i.e., we move the parameters in the opposite direction
- This usually reduces the loss, although we can't move too far because the gradient is different at each point



CHALLENGES INCLUDE HIGH DIMENSIONALITY, NON-CONVEXITY, POOR CONDITIONING, AND GRADIENT CLIFFS

Training deep networks is computationally intensive

- Forward and backward passes can involve millions of floating-point operations
- The input space is large, so many forward passes are needed to estimate the loss
- The parameter space is large, so it takes a lot of computation to explore even a tiny fraction of it
- The loss is not a convex function of the parameter space
- Training a modern deep network may take days or weeks or more on specialized hardware

UNIVERSITY OF WATERLOO

Loss is not a convex function of parameters

- A convex function f is one for which,

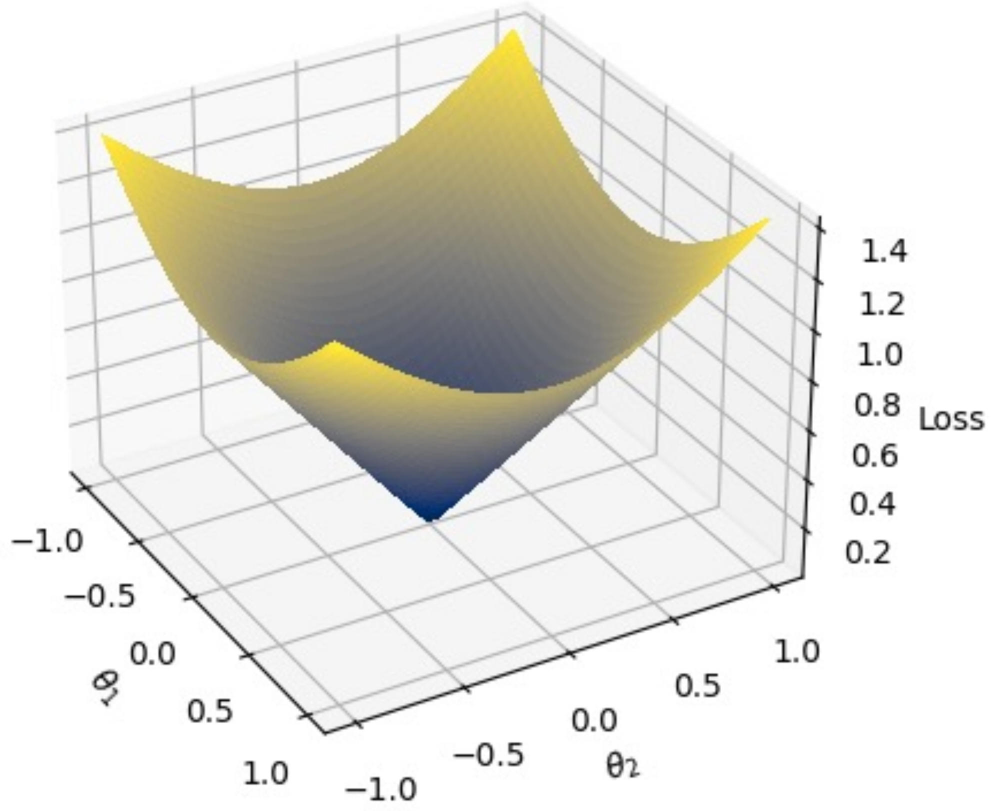
  $$f(ap_1 + (1-a)p_2) \leq af(p_1) + (1-a)f(p_2),$$

  for $0 \leq a \leq 1$

- Convex loss functions are relatively simple to optimize because they don't have local minima or saddle points

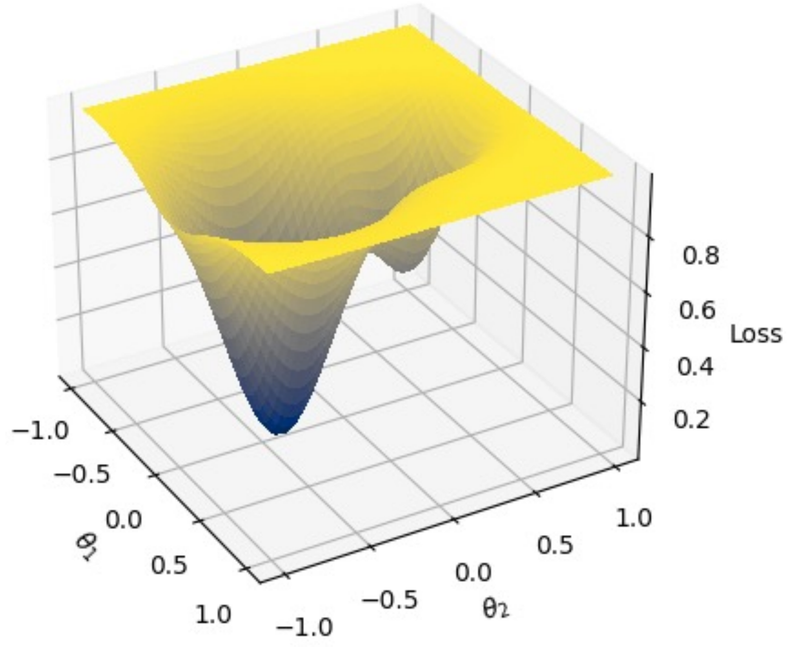
- Losses are not convex functions of deep network parameters

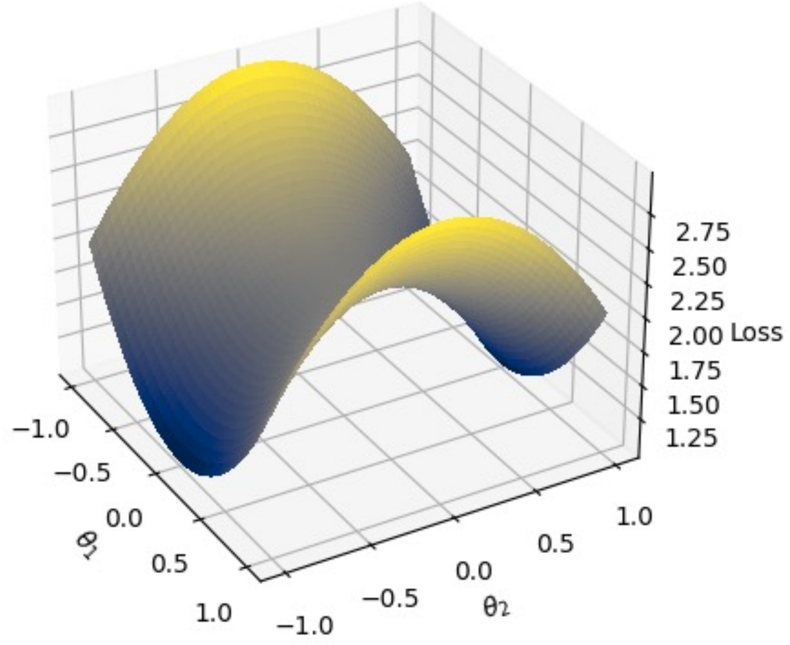
Example of a convex function



Non-Convex Difficulties

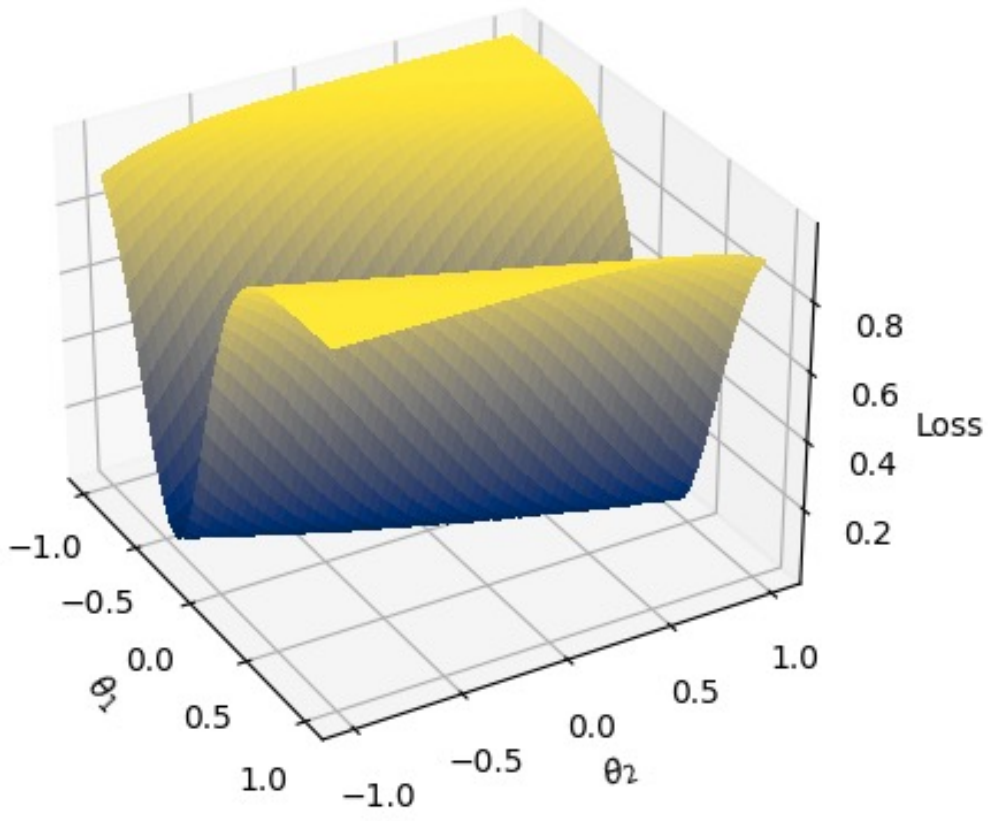
- There could be local minima
- There could be saddle points





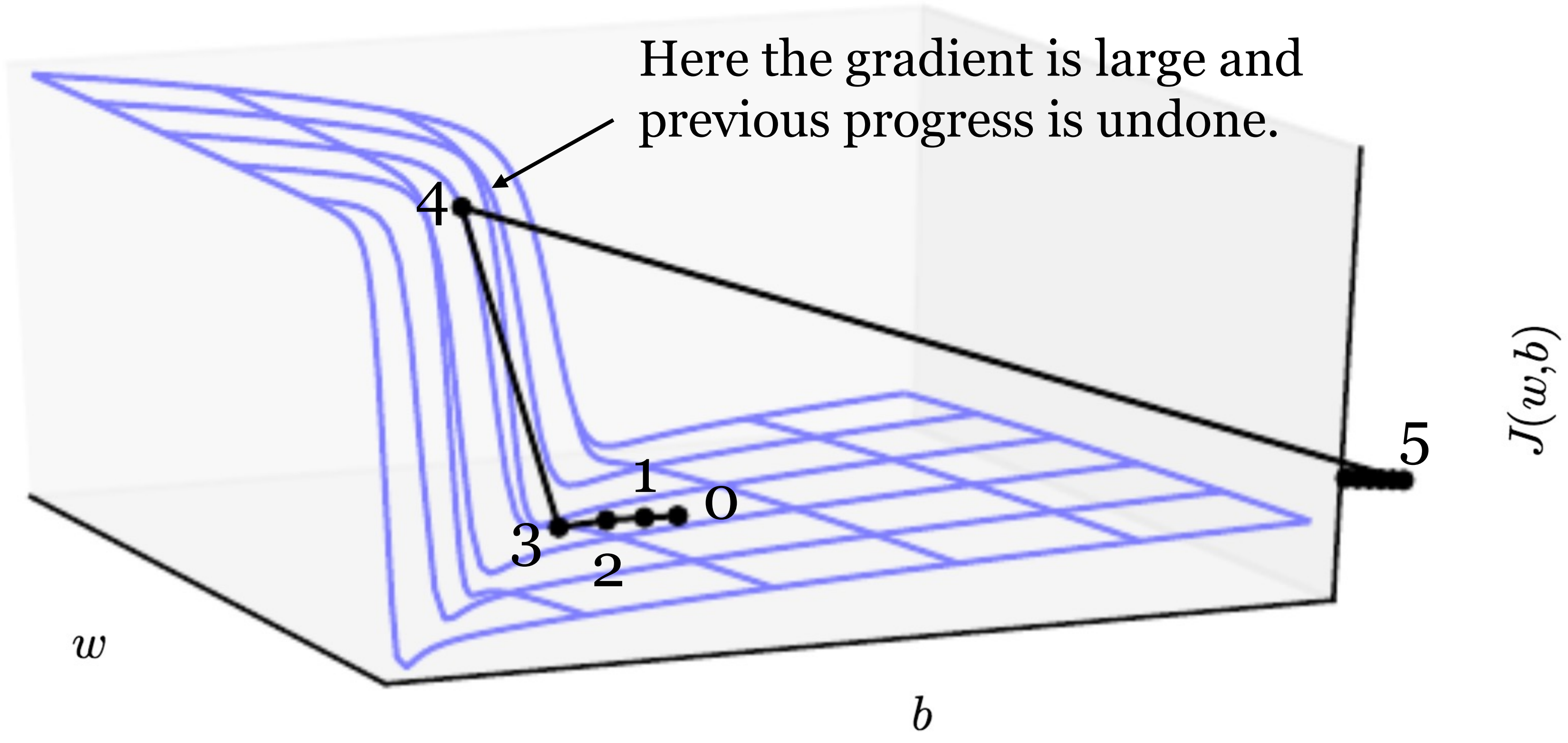
The loss function can have a wide range of curvatures

- Even if a function is convex, it can have a wide range of curvatures (second derivatives) along different directions in parameter space
- A wide range of curvatures appears as poor conditioning (large ratio of largest to smallest eigenvalues) of the Hessian (matrix of second derivatives)
- This complicates learning because around a minimum
  - The gradient tends to point in directions of high curvature
  - Steps may bounce back and forth across the minimum
  - If the step size is not small enough the loss can increase (bounce farther up)



The loss function can have cliffs

- Recurrent networks are particularly prone to this
- Addressed by gradient clipping (update in direction of gradient but with a maximum step size)



Goodfellow, Bengio & Courville, *Deep Learning*

DEEP NETWORKS PROBABLY DO NOT SUFFER MUCH FROM LOCAL MINIMA

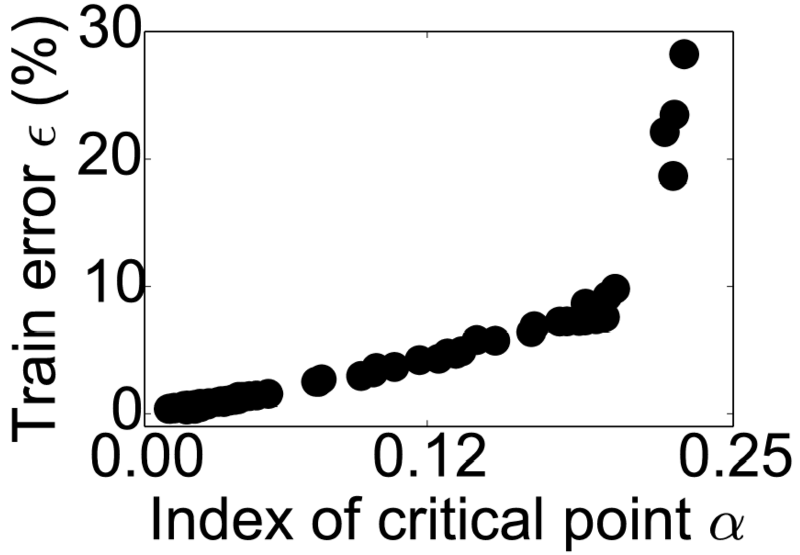
Local Minima

- A given fixed point is only a local minimum if all the eigenvalues of the Hessian (matrix of second derivatives) are non-negative
- Imagine a deep network's loss is a random function in which such eigenvalues are independent random variables
- If each can be positive or negative with equal probability, then it would be rare for all to be positive if there were many parameters (many eigenvalues)
- Saddle nodes would be more likely; these would typically slow gradient descent without stopping it completely

Local Minima

- Furthermore, with certain random functions, the fraction of negative eigenvalues has a strong tendency to increase with the value at the fixed point
- So fixed points that are harder to escape (fewer directions of negative curvature) have lower error
- This has been experimentally demonstrated to occur in some neural networks (see figure)
- Empirically, minima tend to have low error
- Relatedly, deep *linear* networks have been shown not to have local minima under reasonable assumptions

Fixed points in MNIST network with single hidden layer



(fraction of –ve eigenvalues of Hessian)

Dauphin et al., 2014, NeurIPS

PARAMETER INITIALIZATION SHOULD BREAK SYMMETRY AND ENCOURAGE STABILITY

Parameter initialization

- Deep networks require initial parameters from which iterative optimization can proceed, and these can strongly affect performance
- We must “break symmetry” between units, i.e., make them different from each other so that they experience different gradients
  - Otherwise, all the units will change in the same way and will be redundant
  - This is usually done by assigning different random initial weights
- Poor initialization can make learning very slow or unstable
- With iterative optimization and finite iterations, initial parameters correspond to a prior assumption about parameter values (e.g., that they should be small)

Uniform vs. normal

- Both uniform and normal distributions are widely used
- This choice usually doesn't make much of a difference

The importance of scale

- Suppose we sample initial weights independently from a standard normal distribution (unit variance) and set initial biases to zero
- Let's say for simplicity that we have 100 inputs with zero mean and variance 1, and we have ten layers of linear neurons with 100 neurons each
- Recall:
  - The variance of a sum of random variables is the sum of variances
  - The variance of a product of zero-mean random variables is the product of variances
- The expected variance of the weighted inputs to first-layer neurons are 1x1=1

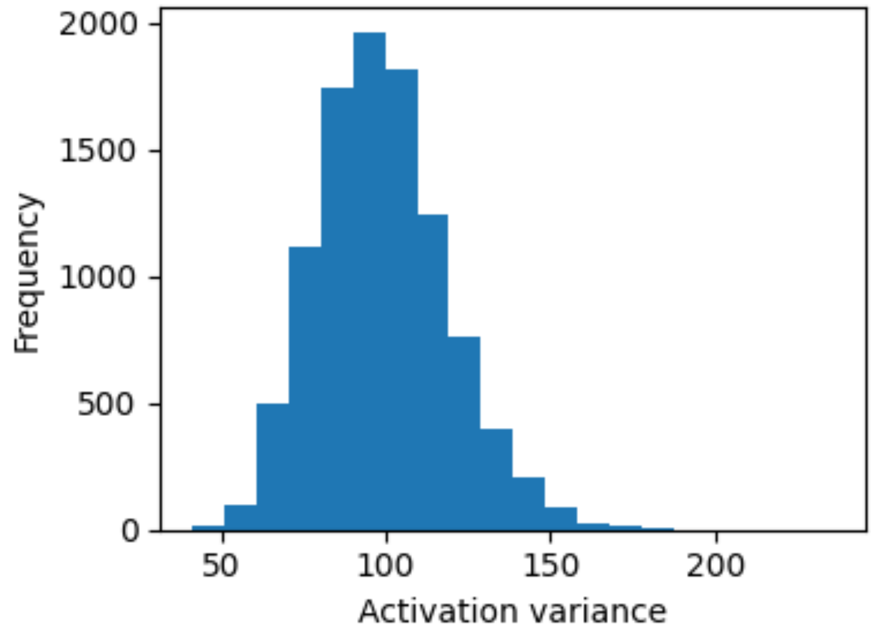
The importance of scale

- The expected variance of sums of weighted inputs (activations) to first-layer neurons is $\sum_{i=1}^{100} 1 = 100$

```
x = np.random.randn(100)
w = np.random.randn(100,100)
weighted_inputs = w*x
activations = np.matmul(w, x)
print(np.var(weighted_inputs))
print(np.var(activations))
```

```
0.9804720076487682
98.50067451065797
```

Activations fluctuate around 100 in different runs



The importance of scale

- Since the neurons are linear in this example, the expected variance of individual weighted inputs to second-layer neurons is 100
- The expected variance of total (weighted sum) input to each second-layer neurons is 100*100=10000
- At the end of the network, we have expected variance $100^{10}=10^{20}$
- This will result in very large outputs, losses, and gradients
- E.g., in the last layer, $\partial L/\partial w$ will be on the order of $\sqrt{10^{18}}\sqrt{10^{20}} = 10^{19}$, so with a typical learning rate (say 0.001) the weights will greatly increase in magnitude, making things worse
- The δs will scale up similarly during backprop, causing trouble early in the network even though the inputs are smaller

The importance of scale

- On the other hand, if the weights are too small, the gradients can become so small that many steps are needed to change them substantially
- E.g., suppose in the same network the weights were initialized with variance 0.0001
- The output is then very small, so $\delta_y = \hat{y} - y$ is approximately $-y$
- δs in the first layer are on the order of $y/\sqrt{10^{20}}$
- Assuming y is not huge, and with a typical learning rate of 0.001, then it would take about ten billion steps to change the original weights by about 10%

Glorot Initialization

- X. Glorot & Bengio, 2010
- They suggest

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

  where m is the # of inputs (fan-in) and n is the # of outputs (fan-out)

- This is a compromise between initializing all layers to have the same activation variance (scaling by m) and having the same gradient variance (scaling by n)

Why 6?

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

The variance of a uniform distribution between a and b is $(b-a)^2/12$. If $m = n$ then

$$\frac{(b-a)^2}{12} = \frac{\left(2\sqrt{\frac{6}{2m}}\right)^2}{12} = \frac{1}{m}.$$

The variance of a sum of independent variables is the sum of variances. So, with m inputs of variance 1, multiplied by weights with variance $1/m$, the expected variance of the activations is 1.

Why 6?

- If the input has variance 1, then the neuron outputs in the first layer will have similar variance 1, as will the neuron outputs the second layer, and so on
- Similarly, the scale of the δs will also be consistent across layers during backpropagation

Kaiming Initialization

- Kaiming He et al., 2015
- They suggest Gaussian weights with standard deviation, $\sqrt{2/m}$.
- Assuming inputs from the previous layer with variance 1, each weighted input has variance $2/m$, and the weighted sum of inputs has variance 2 (rather than 1 as in Glorot initialization)
- This is meant to correct for rectification with ReLU nonlinearities; the rectified outputs should have variance close to 1

STOCHASTIC GRADIENT DESCENT MAKES EFFICIENT USE OF DATA

Gradient descent

- The idea of gradient descent is to calculate the gradient of the loss with respect to the parameters, and to take a small step in the opposite direction in parameter space
- The network's loss is the average of the losses for each training example, over the whole training dataset (we want the network to get close to *each* of the targets, given *each* of the inputs)
- To perform gradient descent exactly, we must first calculate the loss and the gradient for all examples in the training dataset (requires lots of computation)

Stochastic gradient descent

- Stochastic gradient descent instead takes steps based on averages of the gradient over small fractions of the training dataset (“minibatches”)
- It is called “stochastic” because minibatches are random samples of the training data, leading to random variations of the corresponding gradient estimates around the true gradient over the whole training dataset
- Each minibatch should consist of independent samples
  - This often requires random shuffling of the dataset, because adjacent samples may be correlated
  - E.g., a dataset may come from multiple sources with examples from each source together in a group

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
```

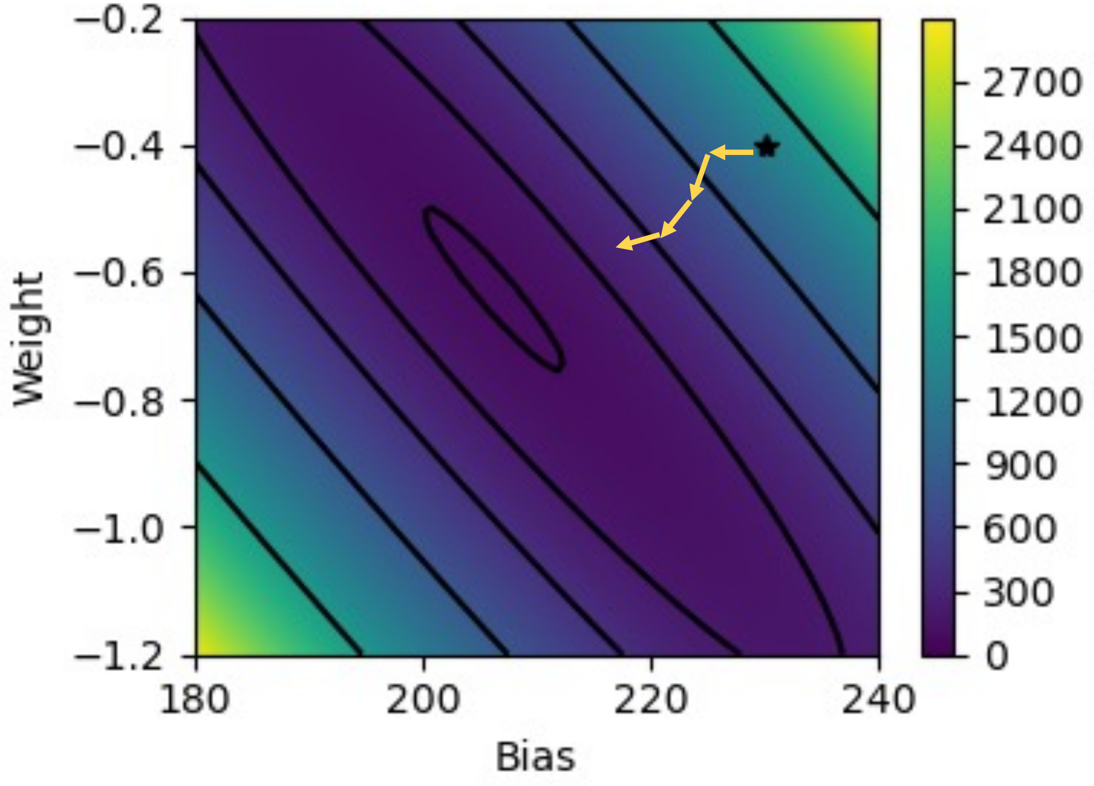
Goodfellow et al. *Deep Learning*

UNIVERSITY OF WATERLOO

Computational efficiency

- This is efficient because the stochastic estimate of the gradient is correlated with the gradient and cheaper to estimate
  - E.g., dozens or hundreds of forward/ backpropagations vs. tens of thousands or millions
- The stochastic gradient due to a single *example* is the true gradient plus a random component with some variance σ^2
- The standard deviation of the stochastic gradient averaged over m independent examples is $\sigma/\sqrt{m}$ (standard error of the mean)
- So, there are diminishing accuracy benefits as m grows large

Fast steps in roughly the right direction (right on average)



Other considerations

- Different examples may be partly redundant in that they make similar contributions to the gradient, reducing the benefit of averaging them
- GPUs are often underutilized by small minibatches, so there is no time benefit of making minibatches any smaller

MOMENTUM HELPS FIND DIRECTIONS WITH SMALL BUT CONSISTENT GRADIENTS

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

velocity of parameters

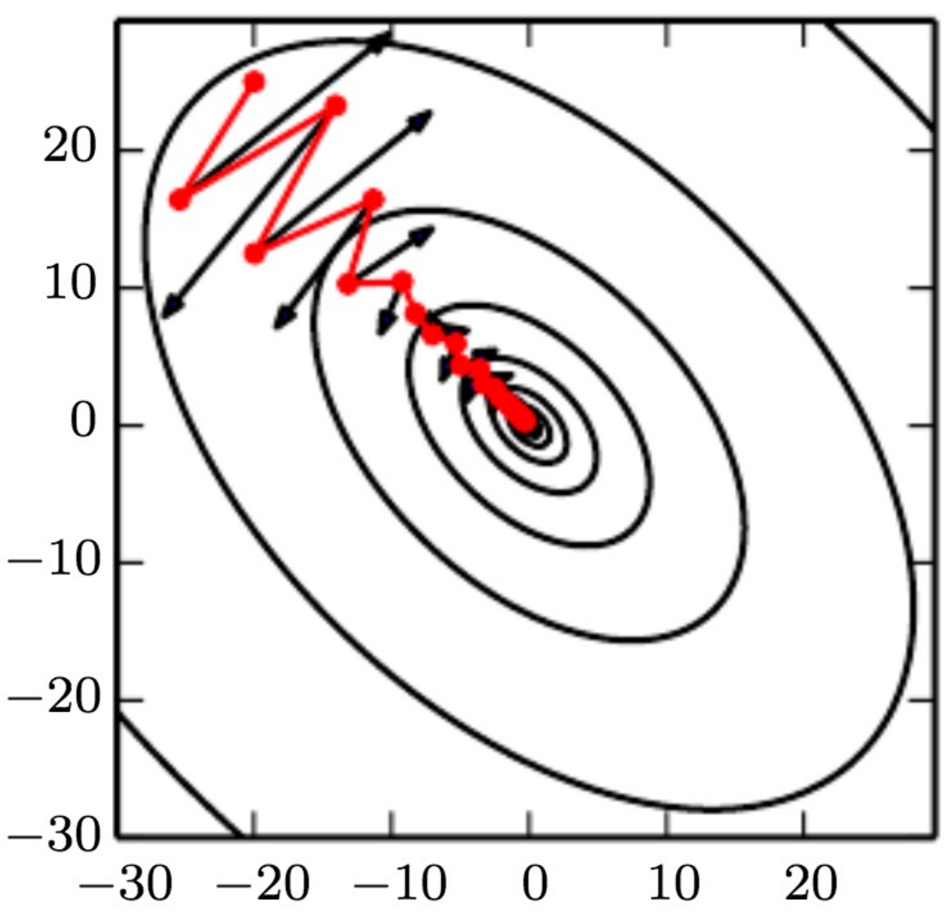
gradient acts like a force

$0 < \alpha < 1$; velocity decays as with linear damping

— Without momentum, updates inefficiently bounce back and forth across steep valley in cost function.

— With momentum, better progress is made in directions of small but consistent gradients.

Also helps ignore random fluctuations in the stochastic gradient.



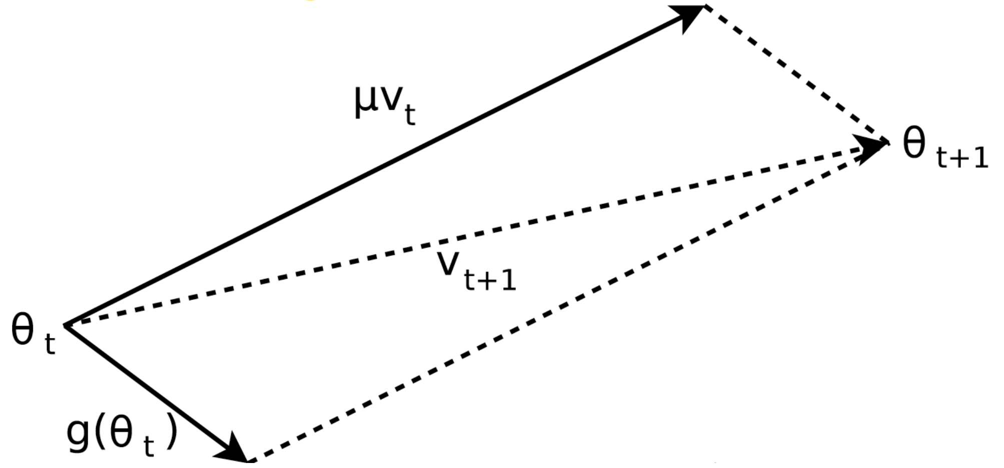
Goodfellow, Bengio & Courville, *Deep Learning*

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

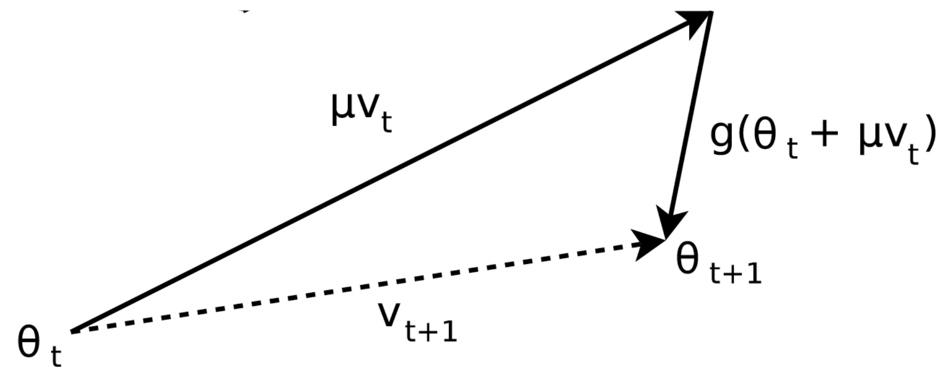
```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

gradient is calculated at a different point

Regular Momentum



Nesterov Momentum



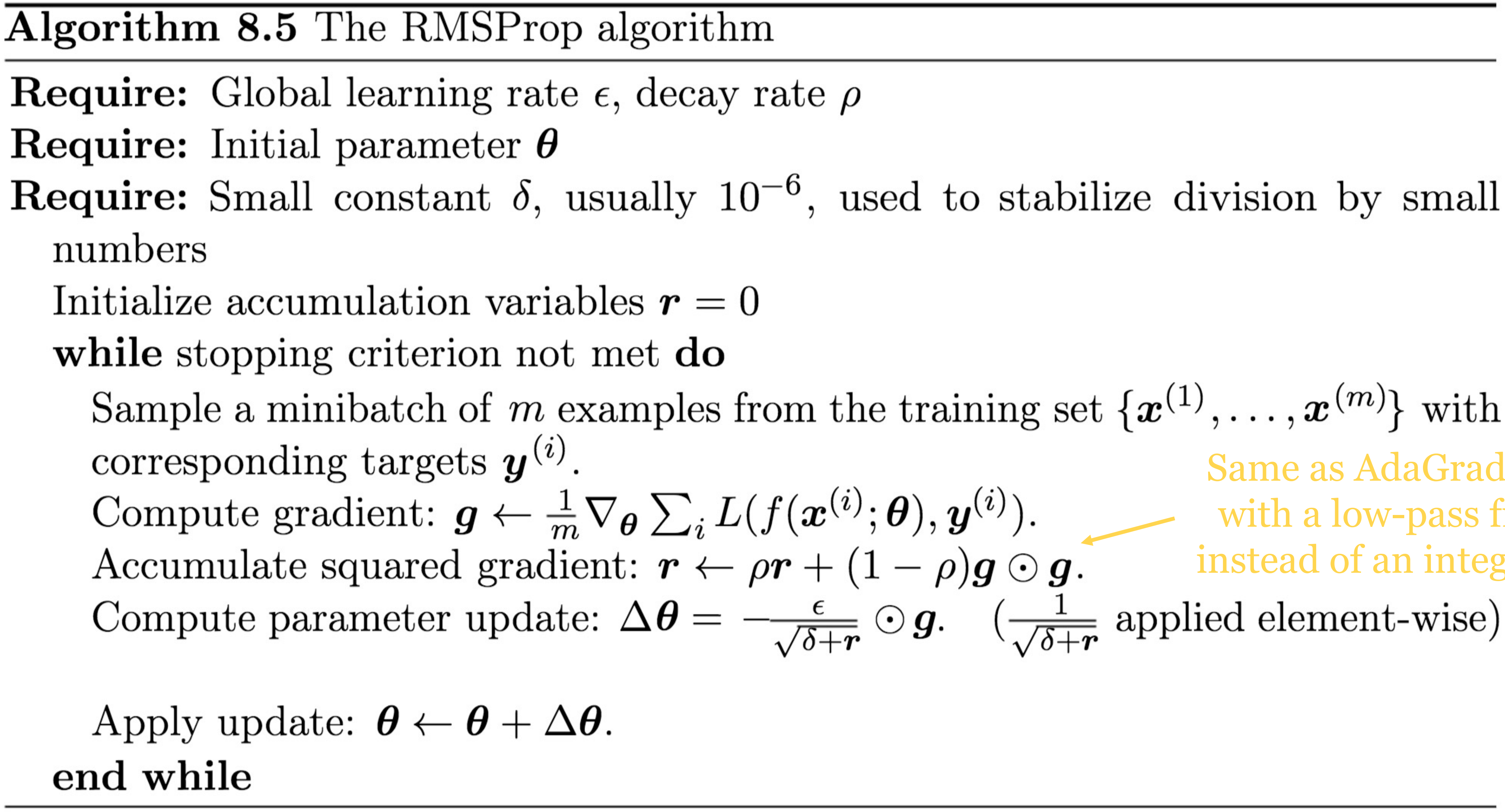
Sutskever et al. (2013)

ADAPTIVE ALGORITHMS REDUCE UPDATES IN STEEP DIRECTIONS

Algorithm 8.4 The AdaGrad algorithm

```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.   ⟵ $\odot$ is element-wise product
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
  end while
```

Shrinks learning rate in directions of large derivatives, allowing relatively faster progress in gently sloped directions. Works well for convex cost functions, but not always for deep networks.



Algorithm 8.5 The RMSProp algorithm

```
Require: Global learning rate $\epsilon$, decay rate $\rho$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, 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
```

Same as AdaGrad, but with a low-pass filter instead of an integrator

Usually works well

Algorithm 8.7 The Adam algorithm

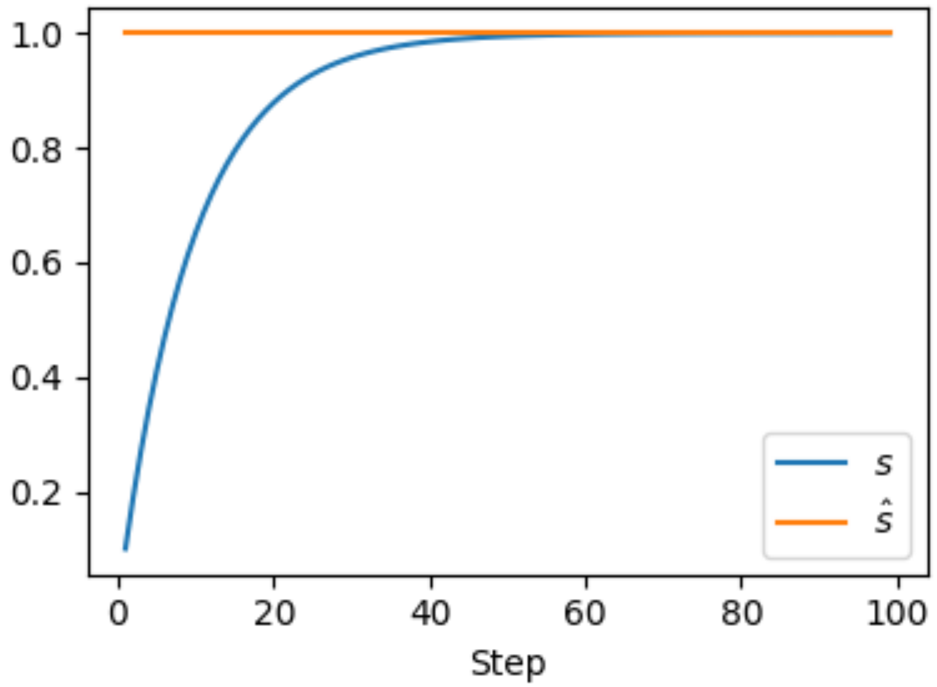
```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  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: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

s is like –ve velocity with damping

correct for small amplitudes in first few updates (due to filter states starting at 0)

Adam correction factors

- The corrections compensate for warm-up of the filters of the first and second moments
- For example, here are components of s and $\hat{s}$ (filtered and corrected first moment) that would result from a constant gradient component of 1



BATCH NORMALIZATION REDUCES COUPLING BETWEEN PARAMETERS IN DIFFERENT LAYERS

Problem: Interactions between parameters

- Following the gradient ignores interactions between parameters in different layers
  - The gradient is calculated using forward-propagated activations, considering only what happens if the weights change while their incoming activations stay the same
  - This makes the effects of parameter updates less predictable
- It is hard to choose a good learning rate, because the effect of each update depends on previous layers (which also change)

Batch normalization

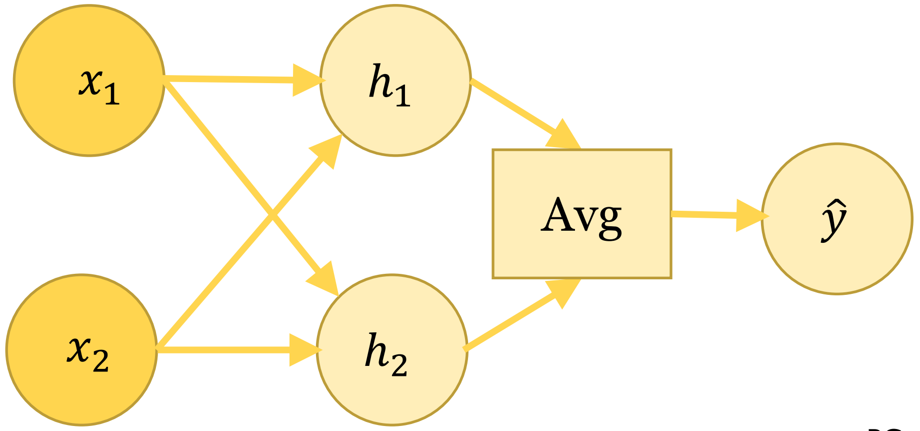
- Let $\boldsymbol{a}$ be the vector of activations of a single neuron over a minibatch
- The activations are normalized as $\boldsymbol{a}' = (\boldsymbol{a} - \mu)/\sigma$, where μ is the mean of the elements of $\boldsymbol{a}$ and σ is their standard deviation
- This gives each neuron's activation a mean of zero and a standard deviation of one, regardless of its weights
- Because these statistics of the activations do not change, the effects of parameter updates are more in line with expectations from the gradient

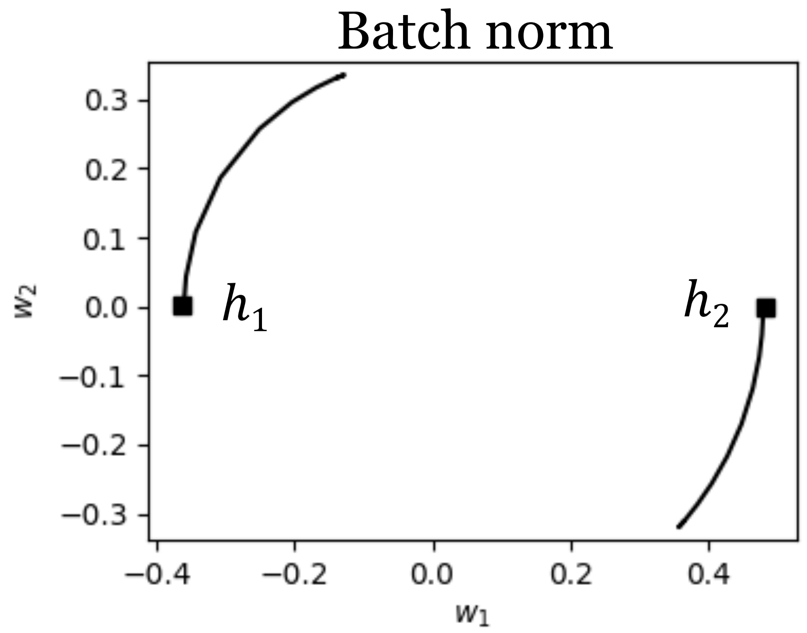
Batch normalization

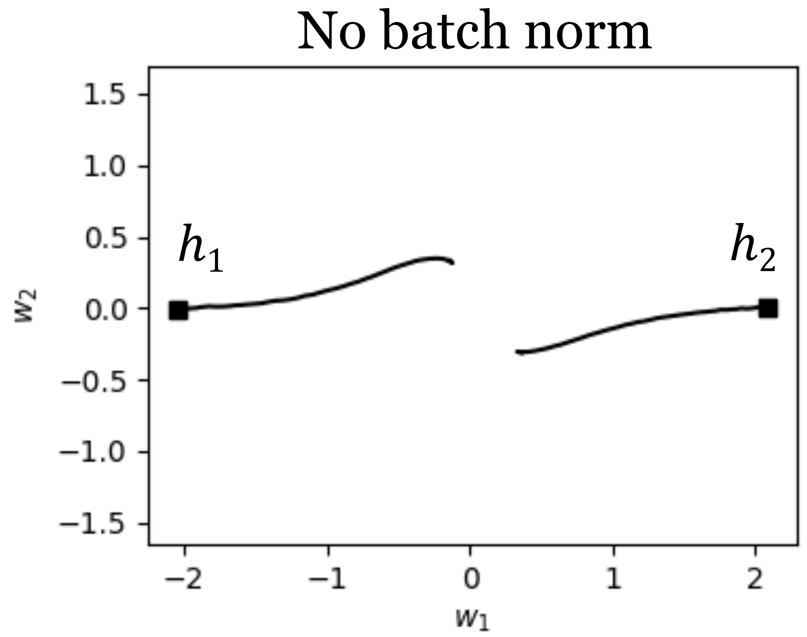
- Normalization and calculation of μ and σ are included in the computational graph, so there are no wasted updates that try to change these factors
- A uniform change in scale of a neuron's weight magnitudes can't affect the loss, so gradients are orthogonal to this direction

Batch normalization

- Example: Network with inputs x_1 and x_2
- Target $|x_1|$
- Biases are fixed at 0
- With batch norm, the network doesn't try to scale each neuron's average activations up or down





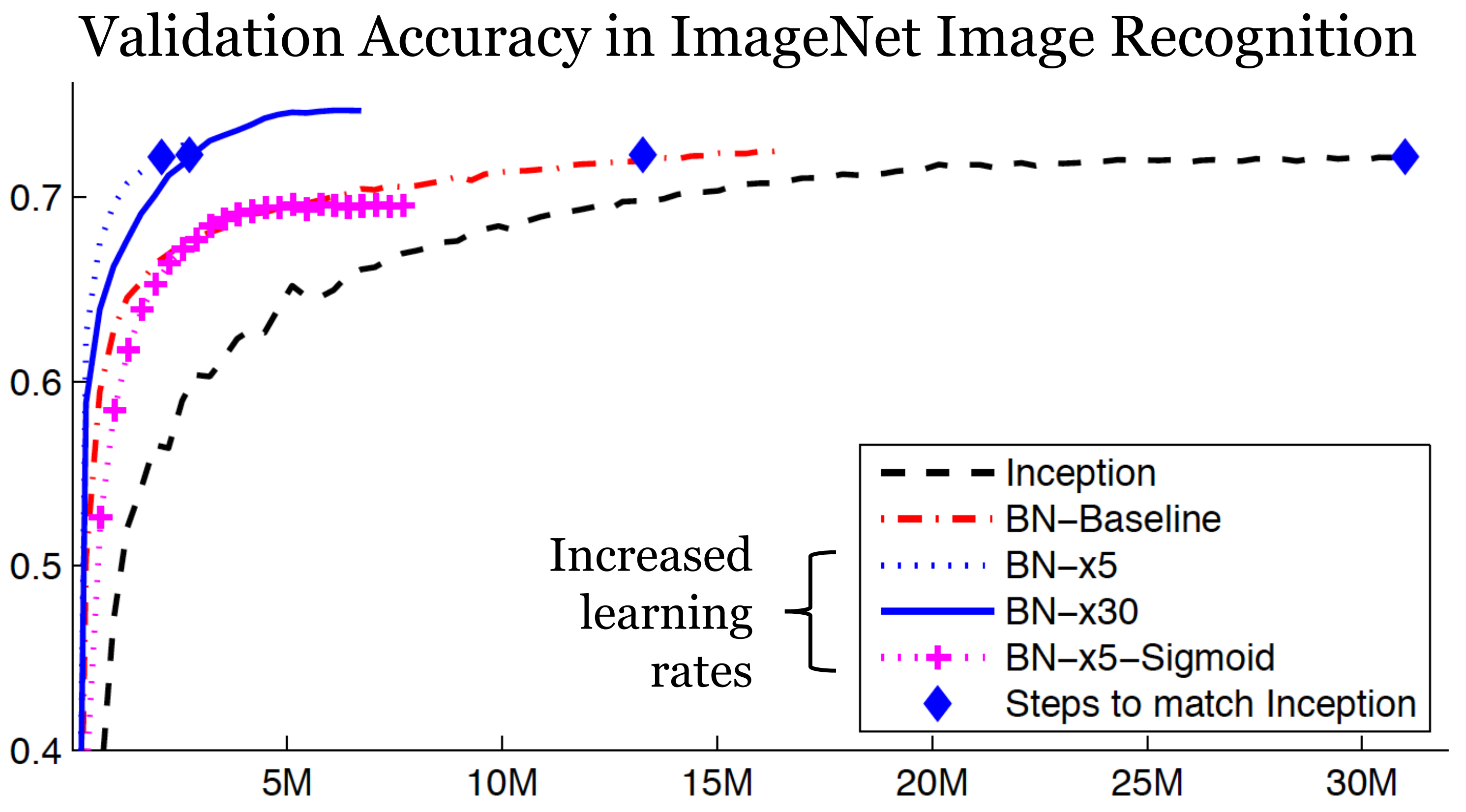


Batch normalization

- After the activations are normalized, they are multiplied by a learned scale and added to a learned bias
- The resulting values can have any mean and variance, just like non-normalized activations
- However, the means and variances depend only on these learned values, not on interactions with all the previous layers

Batch normalization

- For inference we use μ and σ estimated from training data
- Batch normalization speeds up training, allows higher learning rates, and improves performance



Ioffe & Szegedy (2015) ICML

HYPERPARAMETERS CAN BE TUNED BY GRID SEARCH, RANDOM SEARCH, OR SEARCH ALGORITHMS

Hyperparameters

- Models usually have many parameters (e.g., weights) that are tuned by the optimization algorithm
- Hyperparameters are additional parameters that are not tuned by the optimization algorithm, such as the number of layers, and number of hidden units per layer
- They can be continuous (e.g., learning rate), discrete (e.g., kernel size), or categorical (e.g., whether to use batch norm in a certain layer)

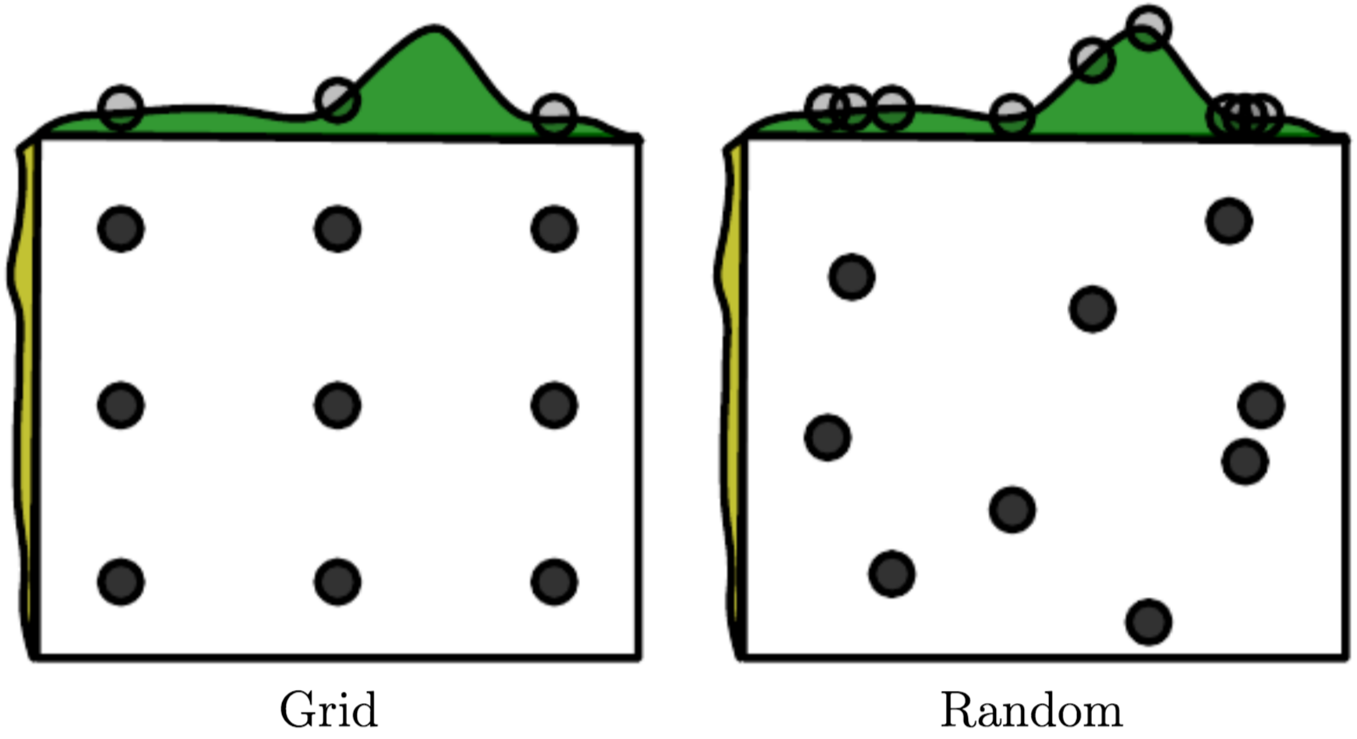
General approaches to hyperparameter selection

- Use hyperparameters that others have already found to work well (e.g., a successful network architecture; default parameters of an optimizer)
- Manual tuning (this requires knowledge their effects)
- Try lots of values (grid search or random search)
- Automatic search (with algorithms like Tree-Structured Parzen Estimator and tools like Optuna)

UNIVERSITY OF WATERLOO

Grid vs. random search

- Usually, some parameters are more important than others
- Random search is more efficient, because sampling less important dimensions causes denser sampling of more important dimensions too



Goodfellow, Bengio & Courville, *Deep Learning*

Hyperparameter optimization algorithms

- These algorithms do not require a gradient
- They treat a performance measure $m(\boldsymbol{h})$ as a black-box function of the hyperparameters $\boldsymbol{h}$
- The most widely used methods develop a statistical model of $m(\boldsymbol{h})$ from samples $\boldsymbol{h}_1, \boldsymbol{h}_2, \boldsymbol{h}_3, \ldots$ and choose next hyperparameters $\boldsymbol{h}_i$ that are likely to result in better performance according to the model
- Example: Tree-Structured Parzen Estimators
  - Use kernel density estimation to model two probability distributions: parameters that give top x% performance and those that don't
  - Choose a $\boldsymbol{h}_i$ that is most likely to belong to the former category rather than the latter

Hyperparameter optimization tools

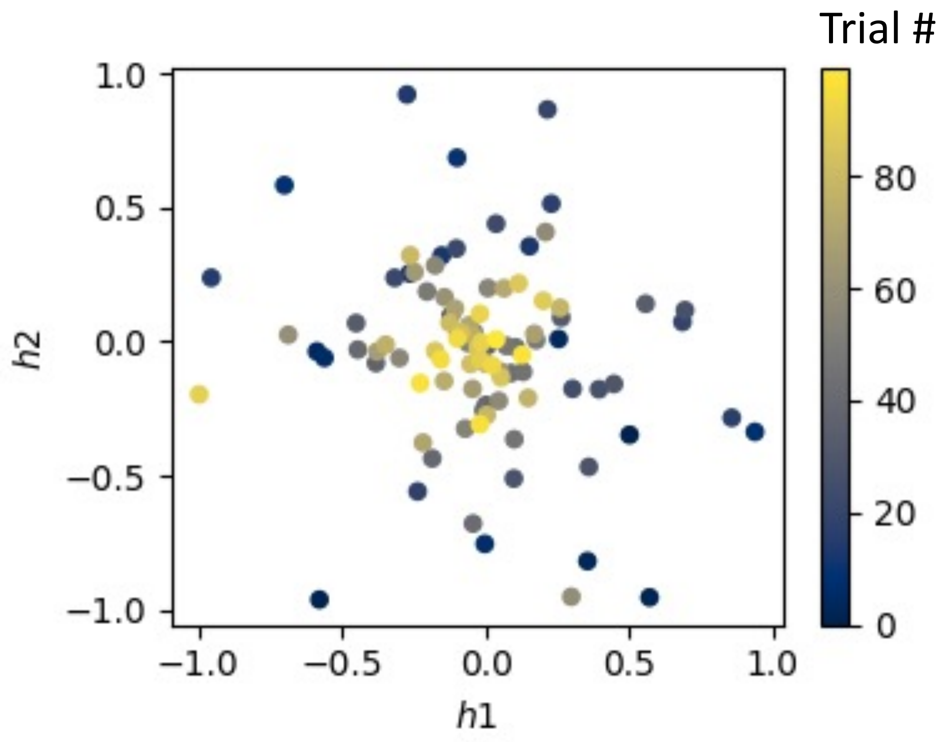
- These usually implement multiple algorithms including random search and Tree-Structured Parzen Estimators and may combine multiple algorithms at different stages
- Some provide support for parallel execution on multiple machines
- Examples include Hyperopt, Ray Tune, Optuna

Optuna example

```python
def black_box_model_performance(h1, h2):
    """
    :param h1: a hyperparameter
    :param h2: another hyperparameter
    :return: a performance measure (lower is better)
    """
    return h1 ** 2 + h2 ** 2


def evaluate_with_new_hyperparams(trial):
    h1 = trial.suggest_uniform('h1', -1, 1)
    h2 = trial.suggest_uniform('h2', -1, 1)
    return black_box_model_performance(h1, h2)
```

```python
study = optuna.create_study()
study.optimize(evaluate_with_new_hyperparams, n_trials=100)
```



HYPERPARAMETERS CAN BE MANUALLY TUNED ACCORDING TO THEIR TYPICAL EFFECTS

Manual tuning of model complexity

- Recall
  - Performance is generally worse on validation data than training data
  - If performance is poor on training data the model is underfitting, suggesting the complexity is too low
  - If performance is good on training data but poor on validation data the model is overfitting, suggesting complexity is too high
- Heuristics for manual hyperparameter tuning
  - Adjust hyperparameters to make complexity more appropriate (e.g., increase if too low)
  - Note that small changes may not have much effect (changing by factors of two may be better)

Manual tuning

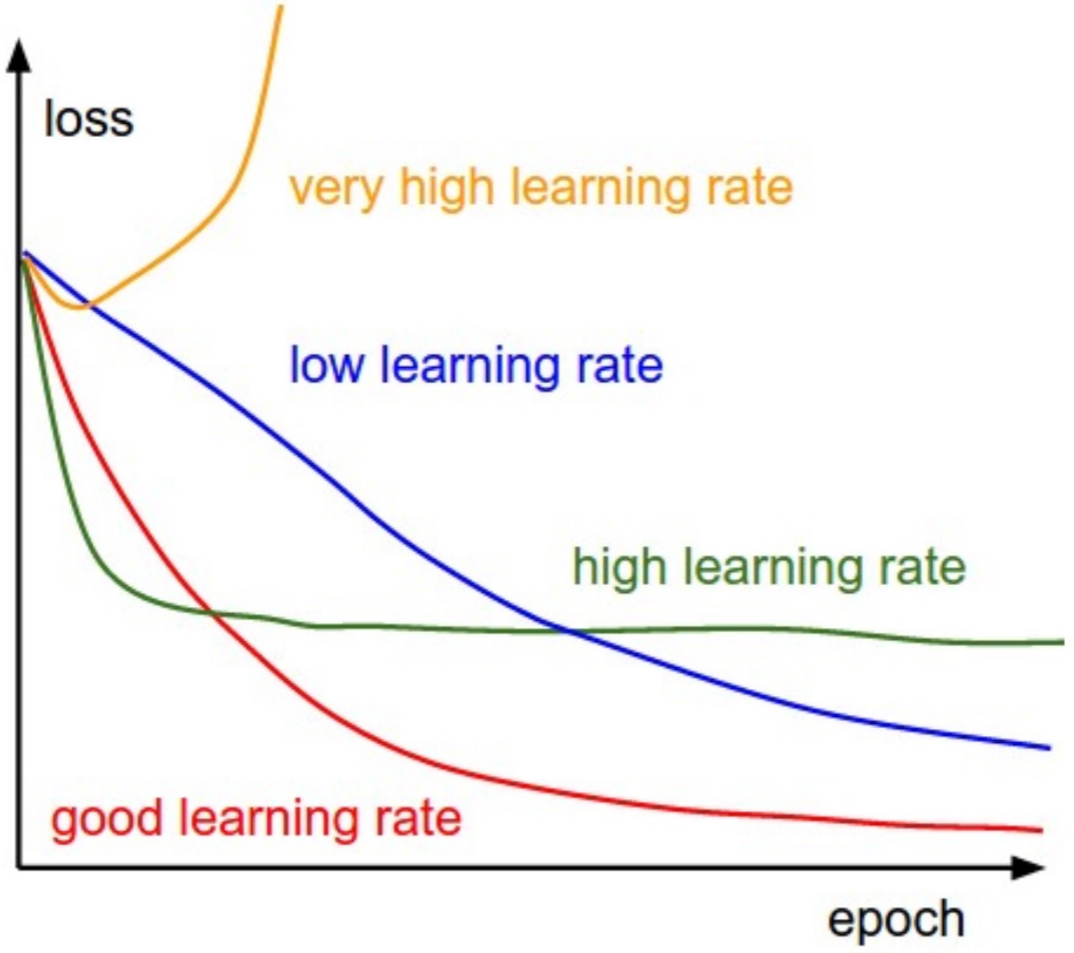
Hyperparameter	**Increases complexity when**
Number of hidden units	Increased
Depth	Increased
Learning rate	Tuned optimally
Convolution kernel width	Increased
Zero padding	Increased
Weight decay coefficient	Decreased
Dropout rate	Decreased

We'll see these soon

Typical effects of learning rate

- This depends on the algorithm
- SGD performance is relatively sensitive to learning rate, but adaptive methods such as Adam are less sensitive

Training error with various rates



http://cs231n.github.io/neural-networks-3/

Summary

1. Challenges include high dimensionality, non-convexity, poor conditioning, and gradient cliffs
2. Deep networks probably do not suffer much from local minima
3. Parameter initialization should break symmetry and encourage stability
4. Stochastic gradient descent makes efficient use of data
5. Momentum helps find directions with small but consistent gradients
6. Adaptive algorithms reduce updates in steep directions
7. Batch normalization reduces coupling between parameters in different layers
8. Hyperparameters can be tuned by grid search, random search, or search algorithms
9. Hyperparameters can be manually tuned according to their typical effects

References

- Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in Neural Information Processing Systems*, *27*.
- Goodfellow, Bengio & Courville (2016) *Deep Learning*, MIT Press
- Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456). *PMLR*.
- Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In International conference on machine learning (pp. 1139-1147). *PMLR*.