DEEP LEARNING

Lecture 3: Regularization and Optimization

Dr. Yang Lu

Department of Computer Science and Technology

luyang@xmu.edu.cn

UNDERFITTING AND OVERFITTING

Generalization

- The central challenge in machine learning is that we must perform well on new, previously unseen inputs—not just those on which our model was trained.
- The ability to perform well on previously unobserved inputs is called generalization.
- During training, we can compute some error measure on the training set called the training error.
  - This is a typical optimization problem.
- What separates machine learning from optimization is that we want the generalization error, also called the test error, to be low as well.

Underfitting and Overfitting

- Is a model the more complex the better?
  - No. It will only perform well on the training data but poorly on the test data.
- Our goal is to make both the training error and the generalization error small.
- Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set.
- Overfitting occurs when the gap between the training error and test error is too large.

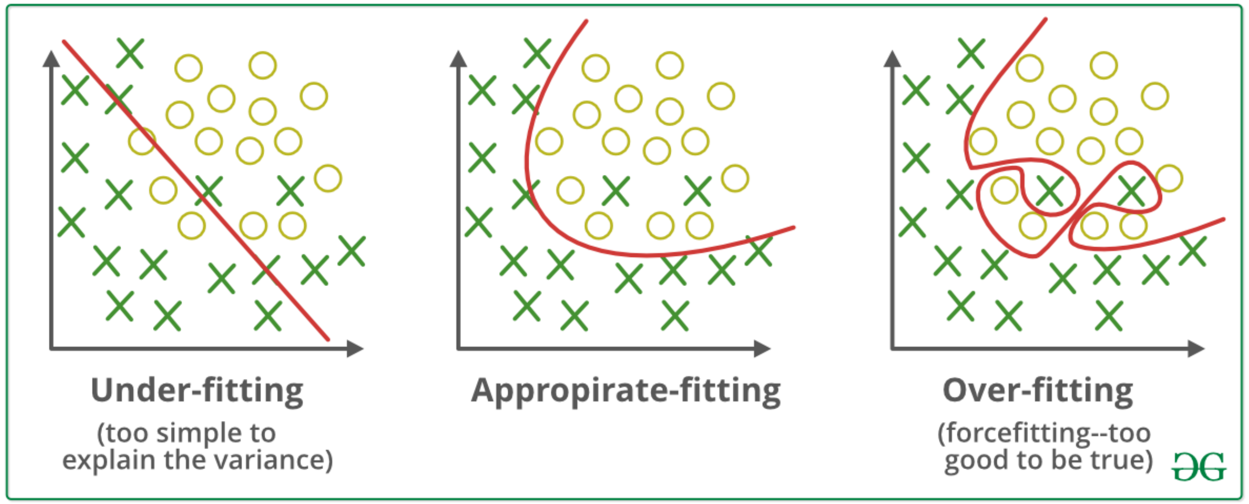

Under-fitting
(too simple to explain the variance)
Appropirate-fitting
Over-fitting
(forcefitting--too good to be true)

Image source: https://towardsdatascience.com/underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6fe4a8a49dbf

Underfitting and Overfitting

- We can control whether a model is more likely to overfit or underfit by altering its capacity or complexity.
  - For neural networks, increasing capacity can be done by adding more hidden layers or more neurons per layer.

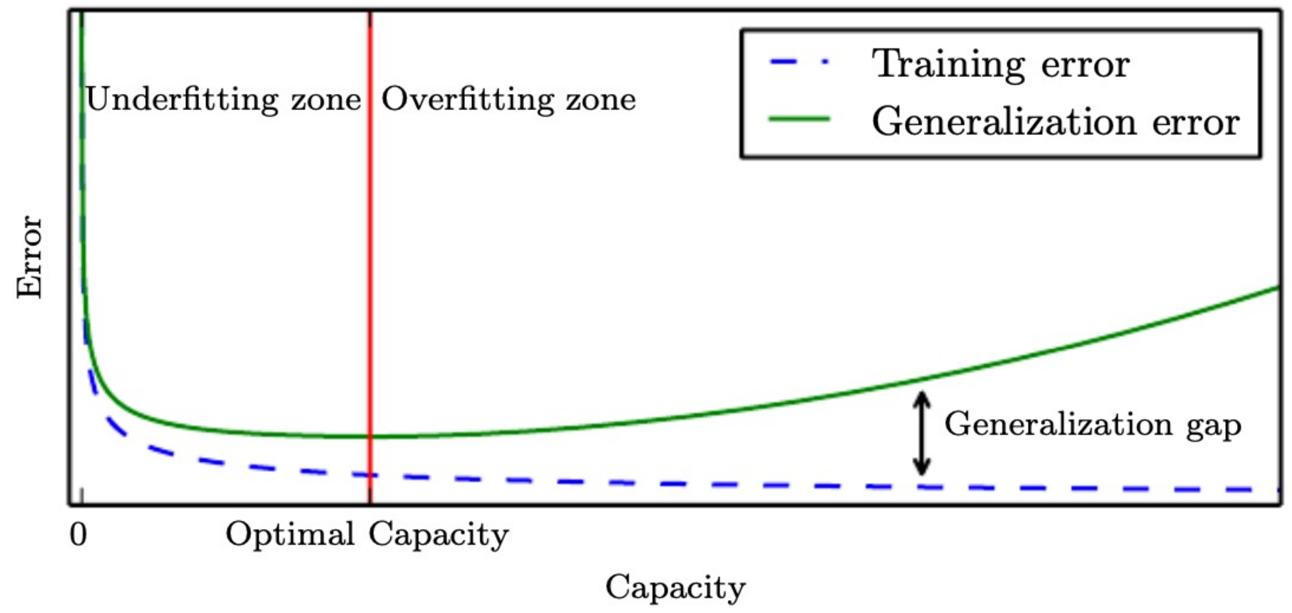


Image source: Figure 5.3, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

Underfitting and Overfitting

- One extreme example is 1-Nearest Neighbor (1NN) classifier.
  - Simply calculate the distance between the input and all samples in the training set.
  - And then use the label of the nearest sample.
- It achieves 0 training error but probably very high test error.
  - Very large generalization gap.

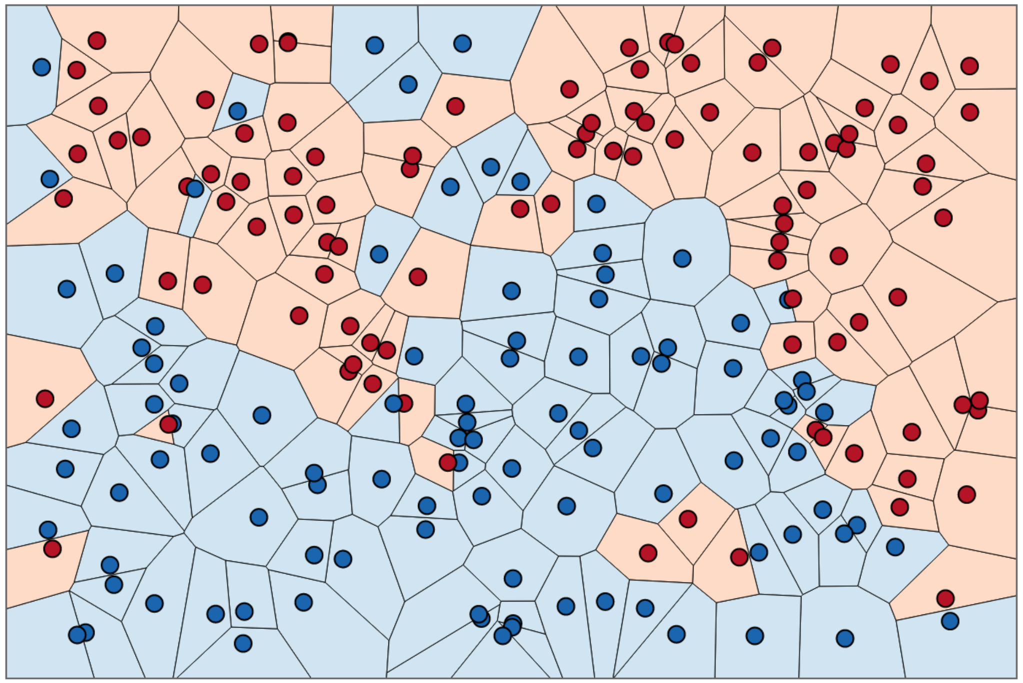

Voronoi diagram for 1NN

Image source: https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e

REGULARIZATION FOR DEEP LEARNING

Regularization

- Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.
- It is one of the central concerns of the field of machine learning.

Parameter Norm Penalties

- Many regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J.
  - E.g. neural networks, linear regression, logistic regression.
- We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}; \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}; \boldsymbol{y}) + \lambda\Omega(\boldsymbol{\theta}).$$

  where $\lambda \in [0, \infty)$ is a hyperparameter called regularization parameter, and $\Omega(\boldsymbol{\theta})$ is called the regularization term or penalty term.
- Now during training, we aim at minimizing $J(\boldsymbol{\theta}; \boldsymbol{X}; \boldsymbol{y})$ and $\Omega(\boldsymbol{\theta})$ at the same time with the tradeoff controlled by λ.

L^2 Regularization

- One of the simplest and most common kinds of parameter norm penalty is the L^2 regularization, aka L^2 norm.

$$\Omega(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 = \frac{1}{2}\sum_i w_i^2 = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}.$$

  - In machine learning, it is commonly known as weight decay.
  - In statistics, it is commonly known as ridge regression or Tikhonov regularization.

L^2 Regularization

- With L^2 regularization, the objective function becomes:

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y}) + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2.$$

- Its corresponding parameter gradient:

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y}) + \lambda\boldsymbol{w}.$$

- A single step with gradient descent:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta(\nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y}) + \lambda\boldsymbol{w})$$
$$\boldsymbol{w} \leftarrow (1-\eta\lambda)\boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y}).$$

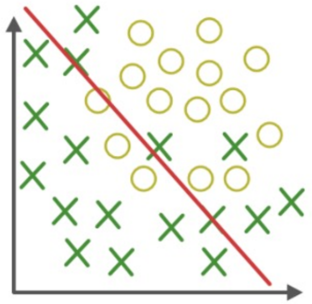
  - Compared with the one without L^2 regularization, we multiply a weight decay term $(1-\eta\lambda)$ on each step.

Parameter Norm Penalties

- Now assume that we have a polynomial classification model for 2-dimensional input (x_1 and x_2):

$$f(x) = \sigma\left(\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} w_{ij} x_1^i x_2^j\right).$$

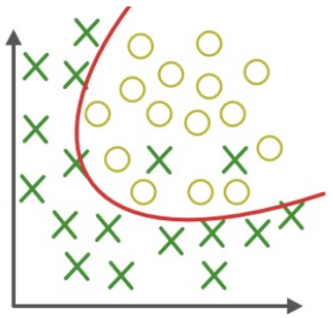
- Logistic regression is its special case when there are only w_{00}, w_{01} and w_{10} are used.



$\sigma(w_{00}$
$+ w_{10}x_1$
$+ w_{01}x_2)$

$$J(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) = 10$$
$$\frac{1}{2}\|\boldsymbol{w}\|_2^2 = 1$$

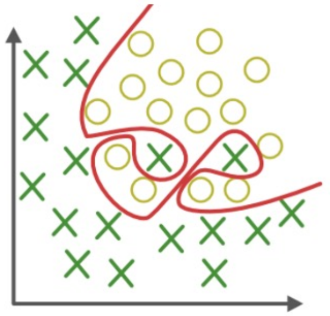
Best model when $\lambda = 5$.



$\sigma(w_{00} + w_{10}x_1$
$+ w_{01}x_2 + w_{20}x_1^2$
$+ w_{02}x_2^2$
$+ w_{11}x_1x_2)$

$$J(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) = 3$$
$$\frac{1}{2}\|\boldsymbol{w}\|_2^2 = 3$$

Best model when $\lambda = 1$.



$\sigma(w_{00} + w_{10}x_1$
$+ w_{21}x_1^2x_2$
$+ w_{22}x_1^2x_2^2$
$+ w_{23}x_1^2x_2^3$
$+ w_{52}x_1^5x_2^2 + \cdots)$

$$J(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) = 0$$
$$\frac{1}{2}\|\boldsymbol{w}\|_2^2 = 50$$

Best model when $\lambda = 0$.

Image source: https://towardsdatascience.com/underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6fe4a8a49dbf

L^1 Regularization

- While L^2 regularization is the most common form of weight decay, there are other ways to penalize the size of the model parameters.
- Another option is to use L^1 regularization, aka L^1 norm:

$$\Omega(\boldsymbol{w}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|.$$

- With L^1 regularization, the objective function becomes:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) = J(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) + \lambda \|\boldsymbol{w}\|_1.$$

- Its corresponding parameter gradient:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) = \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}; \boldsymbol{y}) + \lambda \cdot \operatorname{sign}(\boldsymbol{w}),$$

where $\operatorname{sign}(\boldsymbol{w})$ is simply the sign of $\boldsymbol{w}$ applied element-wise.

Difference between L^1 and L^2 Regularization

- In comparison to L^2 regularization, L^1 regularization results in a solution that is more sparse. Sparsity in this context refers to that most of parameters have an optimal value of zero.
  - Dense parameters: $\boldsymbol{w}^T = [0.2, \quad 0.1, \quad 0.3, \quad 0.5, \quad 0.1, \quad 0.2, \quad 0.5, \quad 0.3]^T$.
  - Sparse parameters: $\boldsymbol{w}^T = [0, \quad 0, \quad 0, \quad 2.1, \quad 0, \quad 0, \quad 1.2, \quad 0]^T$.
- From numerical point of view, L^2 regularization heavily penalizes large terms and tolerates small term, but L^1 regularization equally treats them.
- The sparsity property induced by L^1 regularization has been used extensively as a feature selection or model interpretation.

Difference between L^1 and L^2 Regularization

- Equivalently, the regularized optimization can be written as the form of constrained optimization:

$$\min J(\boldsymbol{w};\boldsymbol{X};\boldsymbol{y})$$
$$s.t.\,\Omega(\boldsymbol{w}) \le k$$

- From geometric point of view, the contour of the objective function has higher probability to hit the constraint corner of L^1 regularization.

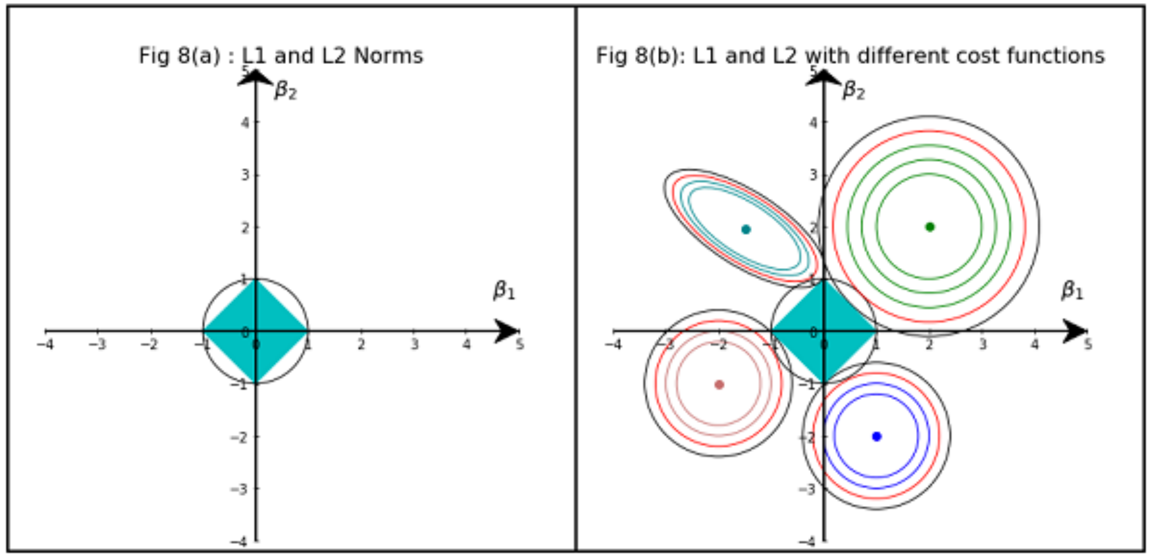


Image source: https://towardsdatascience.com/regularization-in-machine-learning-connecting-the-dots-c6e030bfaddd

Sparse Representations

- Another strategy is to place a penalty on the activations of the units in a neural network, encouraging their activations to be sparse.

$$\tilde{J}(\boldsymbol{\theta};\boldsymbol{X};\boldsymbol{y}) = J(\boldsymbol{\theta};\boldsymbol{X};\boldsymbol{y}) + \lambda\Omega(\boldsymbol{h}).$$

- To achieve it, we can also use L^1 penalty to limit the activations:

$$\Omega(\boldsymbol{h}) = \|\boldsymbol{h}\|_1 = \sum_i |h_i|.$$

Data Augmentation

- The best way to make a machine learning model generalize better is to train it on more data.
- Of course, in practice, the amount of data we have is limited.
- One way to get around this problem is to create fake data and add it to the training set.
  - For some machine learning tasks, it is reasonably straightforward to create new fake data.

Data Augmentation

A few of the simple and popular data augmentation techniques for images are:

- Flipping (both vertically and horizontally)
- Rotating
- Zooming and scaling
- Cropping
- Translating (moving along the x or y axis)
- Adding Gaussian noise (distortion of high frequency features)

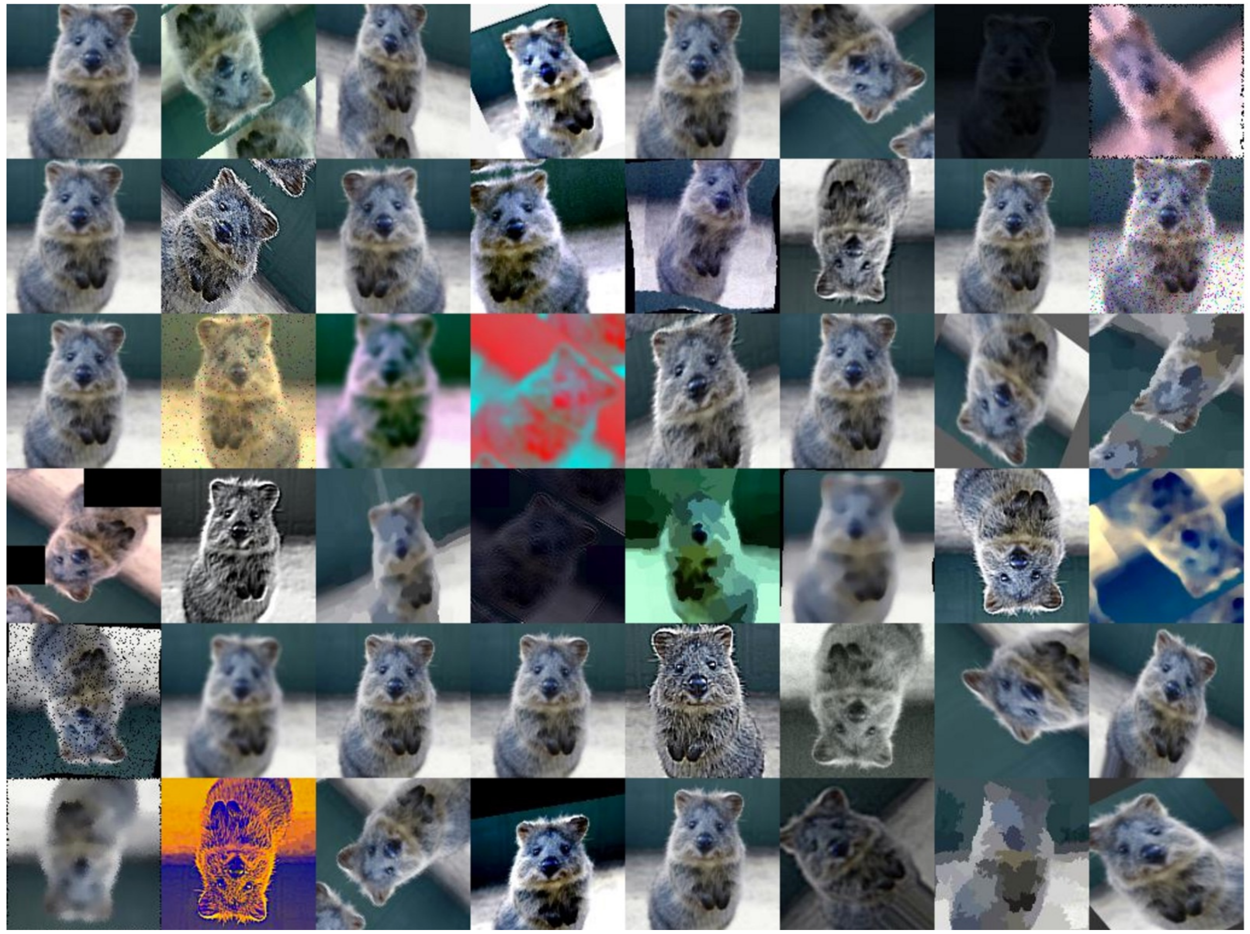


Image source: https://github.com/aleju/imgaug

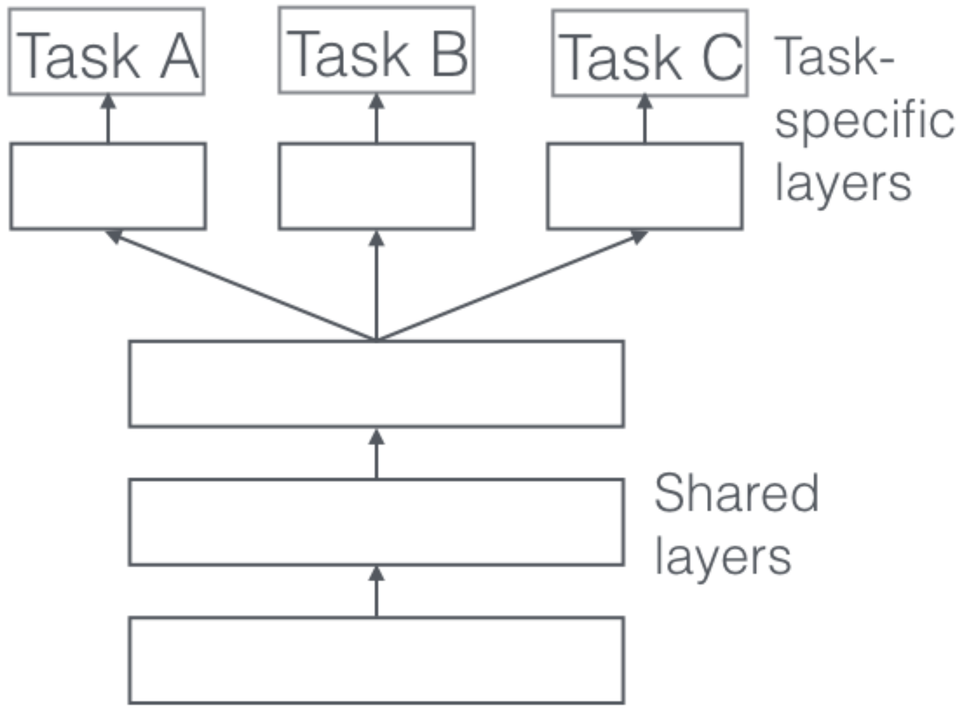
Noise Robustness

- Use the same idea as norm penalties: make the model not learn that well on the training set.

1. Inject noise to the network weights.
    - Push the model into regions where the model is relatively insensitive to small variations in the weights.
    - Find weights that are not merely minima, but minima surrounded by flat regions.
2. Inject noise at the output targets.
    - Make the model not too sensitive to the mistake label in the training set.
    - For example, label smoothing regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1-\epsilon$, respectively.

Multi-Task Learning

- Multi-task learning aims to learn several tasks at the same time.
  - Thus, each task cannot be learned too well, so that regularization is achieved.
- Multi-task learning model can generally be divided into two kinds of parts and associated parameters:
  - Task-specific layers: only benefit from the examples of their task to achieve good generalization.
  - Generic layers or shared layers: benefit from the pooled data of all the tasks.

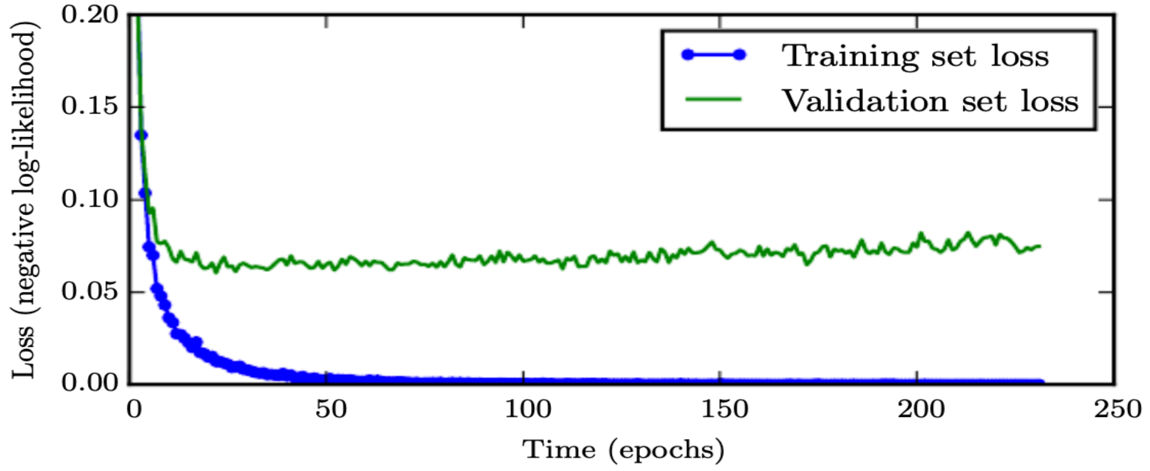


Multi-task learning model with deep neural networks

Image source: https://ruder.io/multi-task/

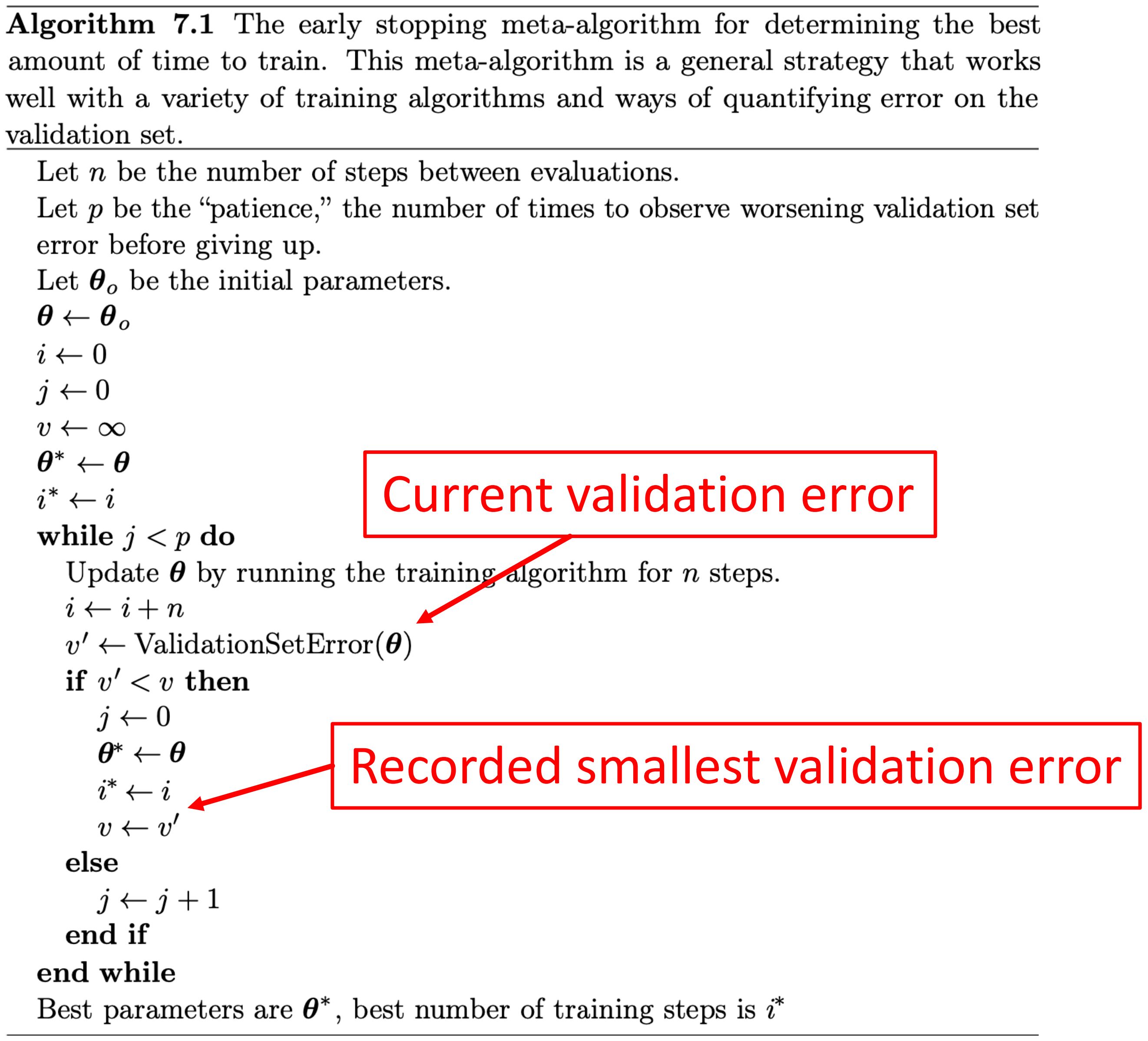
Early Stopping

- Recall that, while training, we are not allowed to use test set to select the best model.
  - It leads to data leakage.
- Instead we can refer to the validation set.
- We can obtain a model with better validation set error by returning to the parameter setting at the point in time with the lowest validation set error.
  - Every time the error on the validation set improves, we store a copy of the model parameters.



The training set loss decreases consistently over time, but the validation set average loss eventually begins to increase again.

Image source: Figure 7.3, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.



Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

```
Let $n$ be the number of steps between evaluations.
Let $p$ be the "patience," the number of times to observe worsening validation set
error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ do
  Update $\boldsymbol{\theta}$ by running the training algorithm for $n$ steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ then
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is $i^*$
```

Current validation error

Recorded smallest validation error

Image source: Algorithm 7.1, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

Advantages of Early Stopping

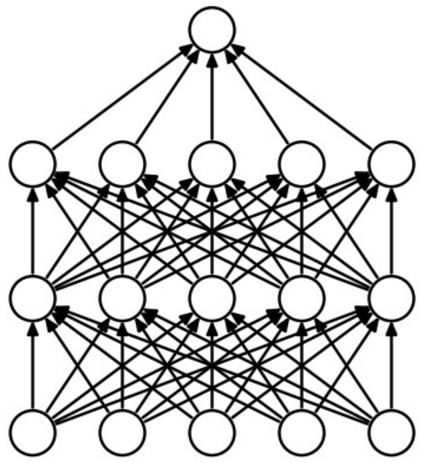
- Early stopping is a very special form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values.
  - It is easy to use early stopping without damaging the learning dynamics.
  - It can be shown that early stopping is equivalent to L^2 regularization, in the case of a simple linear model with a quadratic error function and simple gradient descent.
- Early stopping may be used either alone or in conjunction with other regularization strategies.

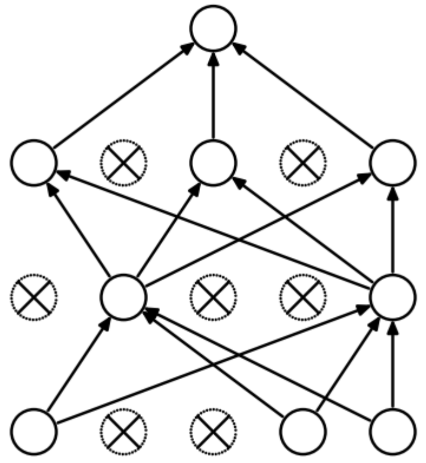
Early Stopping

- Early stopping requires a validation set, which means some training data is not fed to the model.
- To best exploit the whole training data, one can perform extra training after the initial training with early stopping has completed.
- There are two basic strategies one can use for this second training procedure:
  - Retrain on all of the data for the same number of steps as the early stopping procedure determined.
  - Continue to train on the whole training data and monitor the average loss function on the validation set.
- Both strategies are not guaranteed to be the best, because there is no validation set any more.

Dropout

- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- The term “dropout” refers to temporarily removing units (hidden and visible) from the neural network.
- Each unit is retained with a fixed probability p independent of other units
  - p can be chosen using a validation set or can simply be set at 0.5.
  - For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.


(a) Standard Neural Net


(b) After applying dropout.

Dropout

- At training time, the unit is present with probability p and is connected to units in the next layer with weights $\boldsymbol{w}$.
- At test time, the unit is always present and the weights $\boldsymbol{w}$ are multiplied by p. The output at test time is same as the expected output at training time.

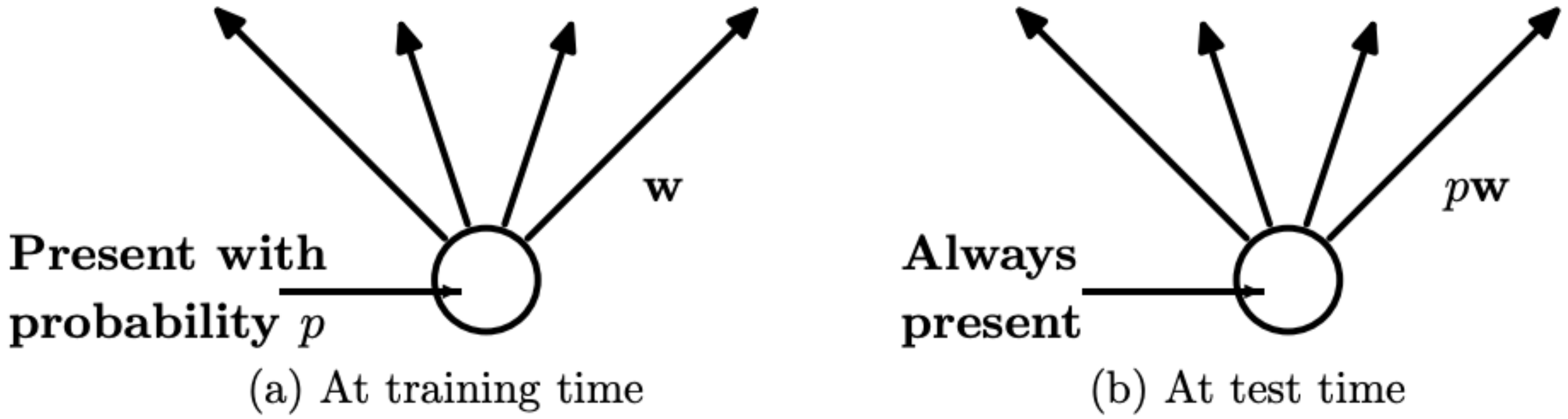
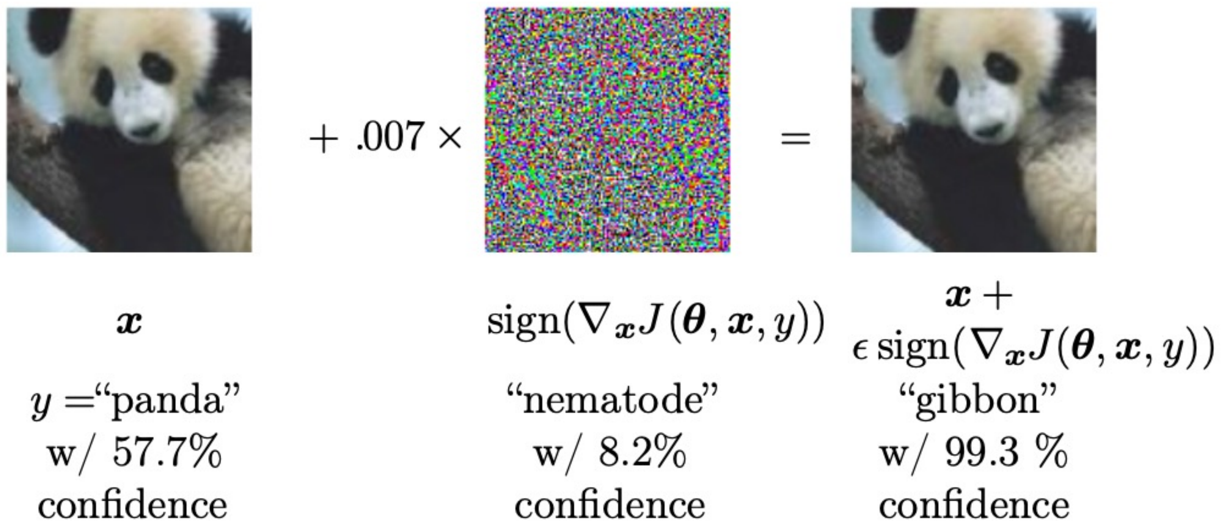


Image source: Figure 1&2, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava, et al. , 2014.

Adversarial Training

- Adversarial examples are the samples x' that can be so similar to x that a human observer cannot tell the difference, but the network can make highly different predictions.
- Adversarial training makes the model insensitive to small changes by encouraging the network to be locally constant in the neighborhood of the training data.



$+ .007 \times$
$=$
$\boldsymbol{x}$
$y =$"panda"
w/ 57.7%
confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
w/ 8.2%
confidence
$\boldsymbol{x} + \epsilon\, \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
w/ 99.3 %
confidence

By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change the classification of the image.

Image source: Figure 7.8, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

OPTIMIZATION FOR TRAINING DEEP MODELS

Batch vs. Minibatch vs. Stochastic Algorithms

- Optimization algorithms that use the entire training set are called batch or deterministic gradient methods.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{1}{N} \nabla_{\boldsymbol{w}} \sum_{i=1}^{N} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{w}), y^{(i)})$$

- The data is usually stored as a form of matrix. Feeding all data into the memory for gradient calculation is infeasible.
- Can we calculate a small bunch of m samples for one update, and conduct N/m times to iterate over all samples?

Will the gradients averaged over m samples deviate from the gradients over N samples?

Batch vs. Minibatch vs. Stochastic Algorithms

- Recall that the standard error of the mean estimated from n samples is given by $\sigma/\sqrt{n}$, where σ is the true standard deviation of the value of the samples.
- The denominator of $\sqrt{n}$ shows that there are less than linear returns to using more examples to estimate the gradient.
  - Compare two estimates of the gradient, one based on 100 samples and another based on 10,000 samples.
  - The latter requires 100 times more memory than the former, but reduces the standard error of the mean only by a factor of 10.

Batch vs. Minibatch vs. Stochastic Algorithms

- Typically the term "batch gradient descent" implies the use of the full training set.
- Optimization algorithms that use only a single sample at a time are sometimes called stochastic or sometimes online methods, e.g., "stochastic gradient descent".
- Most algorithms used for deep learning fall somewhere in between, using more than one but less than all of the training examples.
- These were traditionally called minibatch or minibatch stochastic methods and it is now common to simply call them stochastic methods.

Batch vs. Minibatch vs. Stochastic Algorithms

- It can be somewhat confusing because the word “batch” often means minibatch used by minibatch stochastic gradient descent.
  - For example, it is very common to use the term “batch size” to describe the size of a minibatch.
- Conventionally:
  - When we refer to stochastic gradient descent (SGD), it actually means minibatch SGD.
  - When we refer to batch size, it actually means minibatch size.
- After we train through all of batches, we call it one epoch.
  - Training deep models usually requires a number of epochs.

Batch vs. Minibatch vs. Stochastic Algorithms

Minibatch sizes (batch sizes) are generally driven by the following factors:

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- The amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- When using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256.
- Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process.

How Learning Differs from Pure Optimization

- In pure optimization, minimizing $J(\theta)$ is the final goal.
  - Pure optimization doesn’t care overfitting.

$$J(\theta) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- In machine learning, we minimize $J(\theta)$ on the training data but we hope to find the minimized $J(\theta)$ on the test data.

$$J^*(\theta) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

where $\hat{p}_{data}$ is the empirical distribution and p_{data} is the data generating distribution.

Empirical Risk Minimization

- If we knew the true distribution $p_{data}(x, y)$, risk minimization would be an optimization task solvable by an optimization algorithm.
- However, $p_{data}(x, y)$ is just the population distribution in statistics, which can never be known.
- Instead, we have to minimize the empirical risk

$$\mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

with some regularization methods.

Challenges in Neural Network Optimization

Local Minima

- For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case.
- Experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost.

Challenges in Neural Network Optimization

Saddle Points

- For many high-dimensional non-convex functions, local minima (and maxima) are in fact rare compared to another kind of point with zero gradient: saddle point.
  - Some points around a saddle point have greater cost than the saddle point, while others have a lower cost.

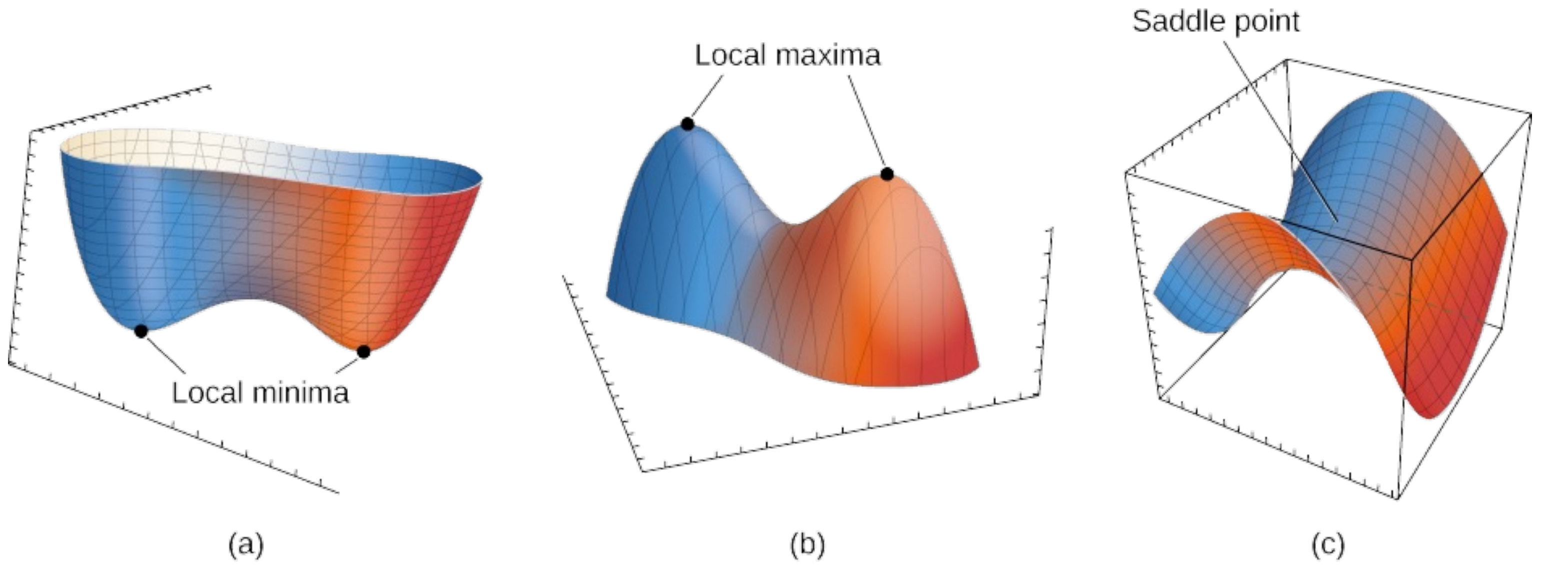


Image source: https://math.libretexts.org/Bookshelves/Calculus/Map%3A_University_Calculus_(Hass_et_al)/13%3A_Partial_Derivatives/13.7%3A_Extreme_Values_and_Saddle_Points

Challenges in Neural Network Optimization

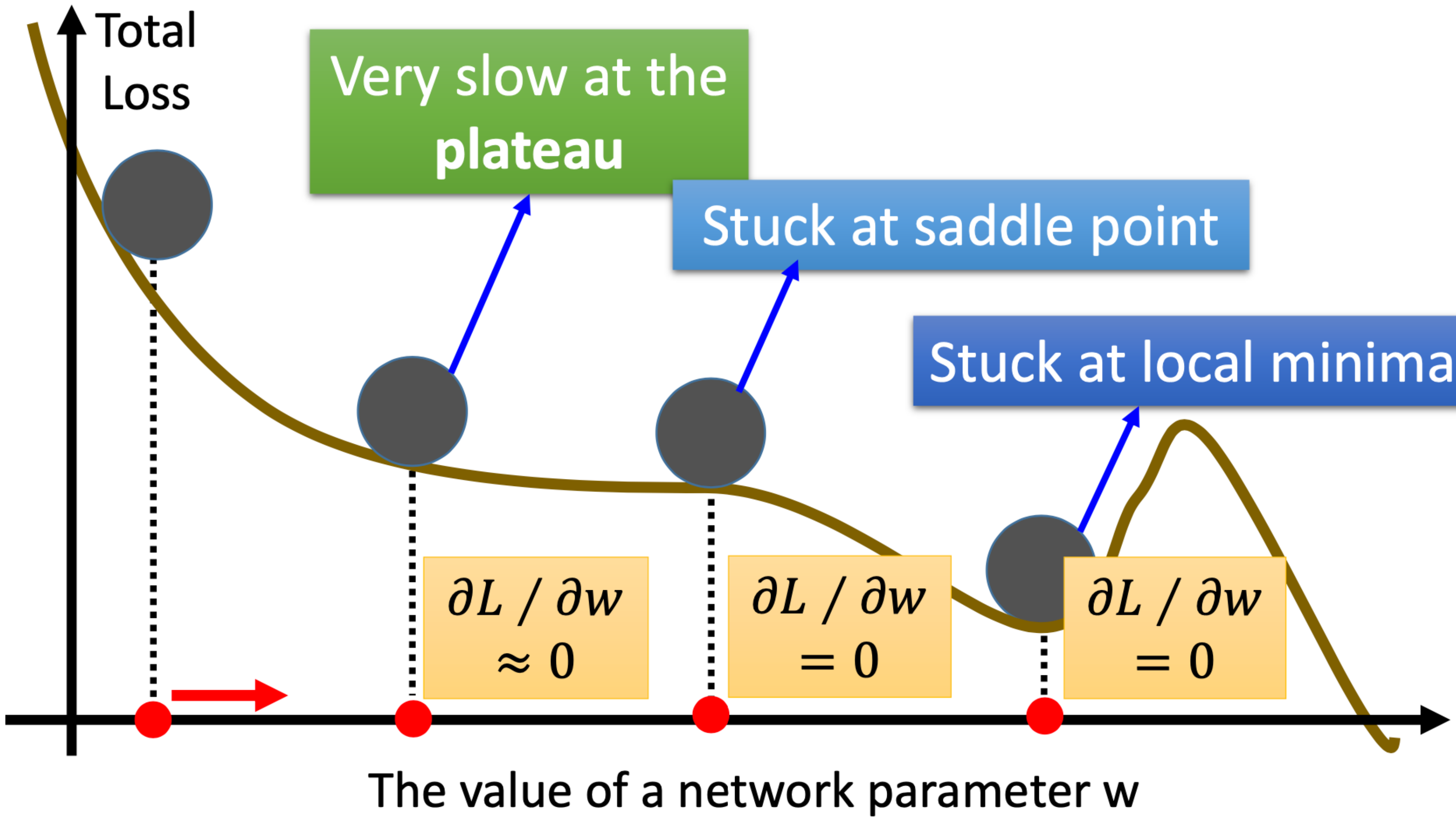

Total Loss
Very slow at the plateau
Stuck at saddle point
Stuck at local minima
$\partial L / \partial w \approx 0$
$\partial L / \partial w = 0$
$\partial L / \partial w = 0$
The value of a network parameter w

Image source: Hung-yi Lee, Understanding Deep Learning in One Day

Challenges in Neural Network Optimization

Cliffs and Exploding Gradients

- The objective function for highly nonlinear deep neural networks often contains sharp nonlinearities in parameter space.
- These nonlinearities give rise to very high derivatives in some places.
- A gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that had been done.

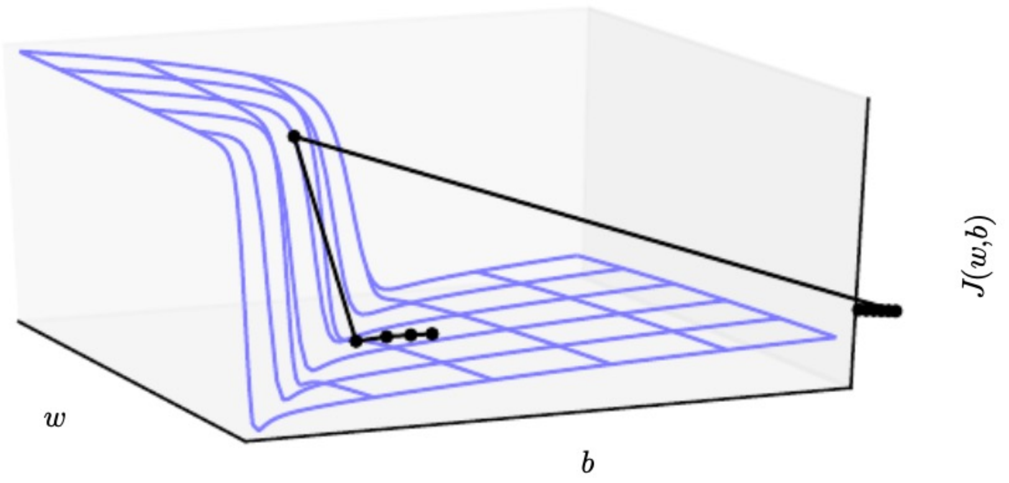
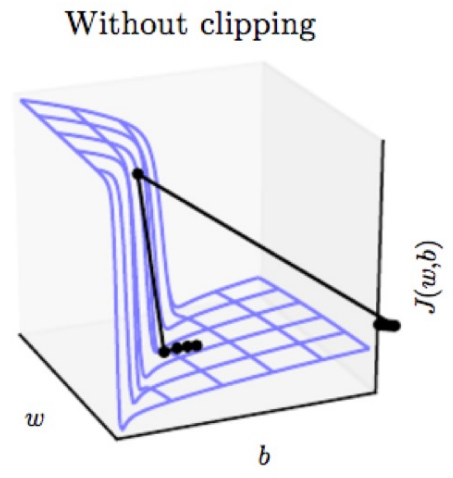


Image source: Figure 8.3, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

Challenges in Neural Network Optimization

Cliffs and Exploding Gradients

- The gradient does not specify the optimal step size, but only the optimal direction within an infinitesimal region.
- A very heuristic solution: gradient clipping, just cut the gradient if it exceeds a threshold.



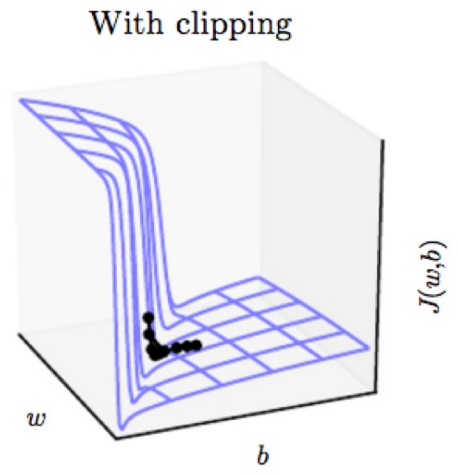
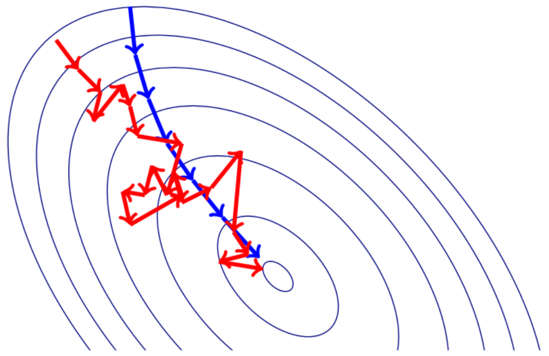


Image source: https://towardsdatascience.com/what-is-gradient-clipping-b8e815cdfb48

Stochastic Gradient Descent (SGD)

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

```
Require: Learning rate $\epsilon_k$.
Require: Initial parameter $\boldsymbol{\theta}$
  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: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while
```



stochastic vs. batch

Image source: Algorithm 8.1, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.
Lecture 6, CMSC 35246 Deep Learning Spring 2017, University of Chicago

Learning Rate of SGD

- Batch gradient descent uses fixed learning rate, because the true gradient of the total cost function becomes small when reach a minimum.
- SGD gradient estimator introduces a source of noise that does not vanish even when we arrive at a minimum.
- In practice, it is necessary to gradually decrease the learning rate of SGD over time, so we now denote the learning rate at iteration k as ε_k.

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau$$

with $\alpha = \min(1, k/\tau)$. After iteration τ, it is common to leave ϵ constant.

Learning Rate of SGD

- Learning rate is usually chosen by trial and error.
- τ is usually set to the number of iterations needed for a few hundred passes through the training data.
- ε_τ should roughly be set to 1% of ε_0.
- How to set ε_0?

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.
  - Small but consistent gradients.
  - The gradients are very noisy.

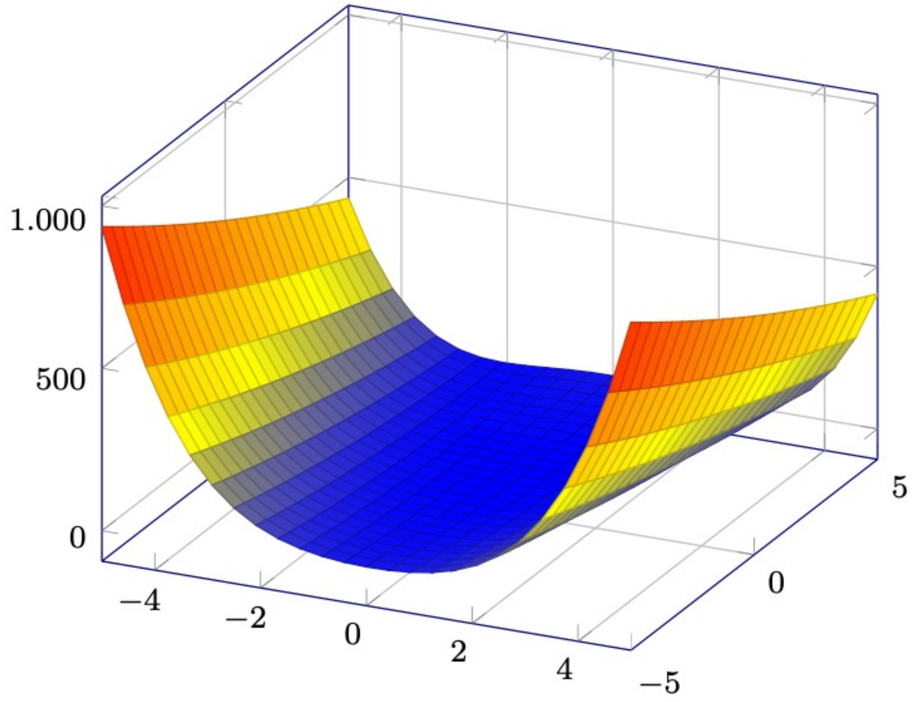
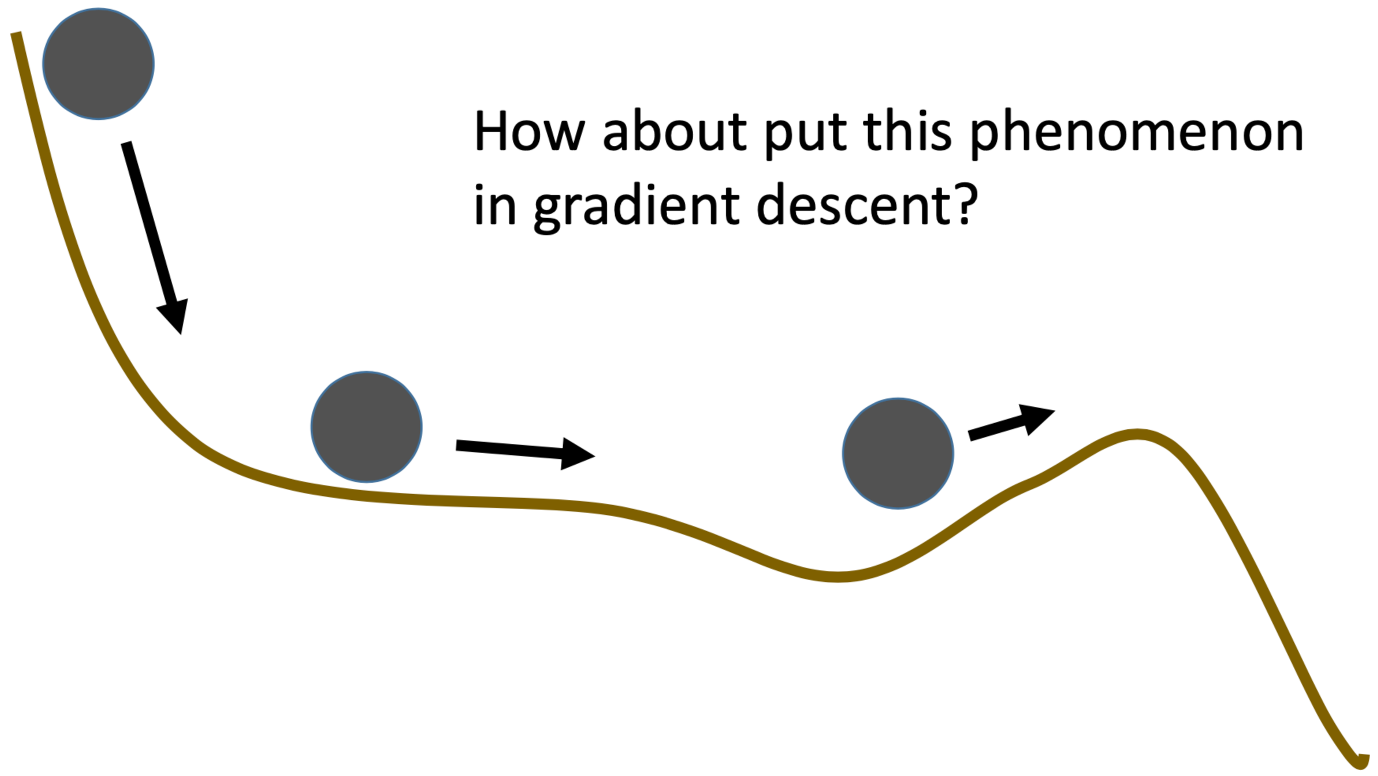


Image source: Lecture 6, CMSC 35246 Deep Learning Spring 2017, University of Chicago

Momentum

- In physical world:
  - Momentum



How about put this phenomenon in gradient descent?

Image source: Hung-yi Lee, Understanding Deep Learning in One Day

Momentum

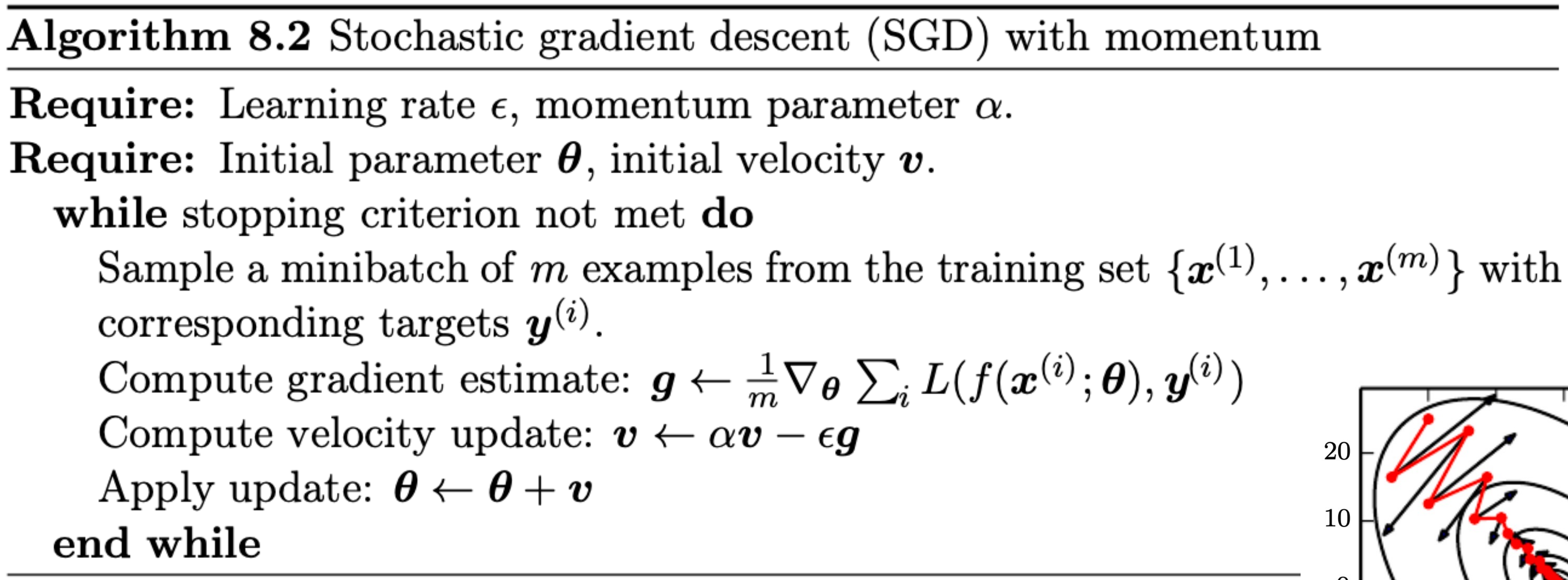
- Each step of SGD has nothing to do with the previous gradient.
- Momentum introduces a new variable $\boldsymbol{v}$, the velocity.
- The velocity is an exponentially decaying moving average of the negative gradients:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

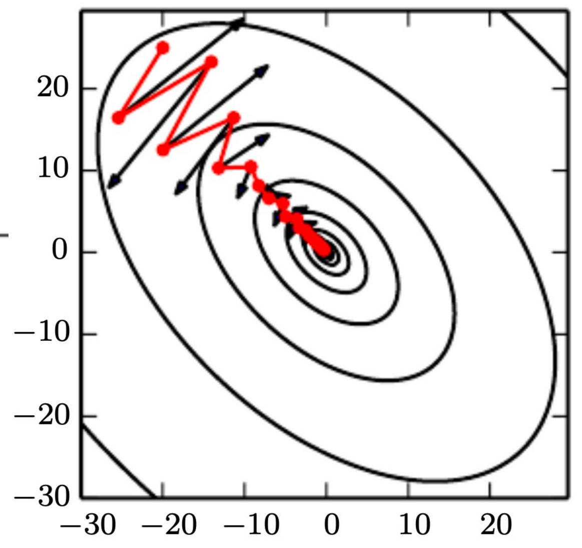
- The velocity accumulates the previous gradients.
- The role of α:
  - If α is larger than ε the current update is more affected by the previous gradients.
  - Usually values for α are set high $\approx 0.8, 0.9$.

Momentum



Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

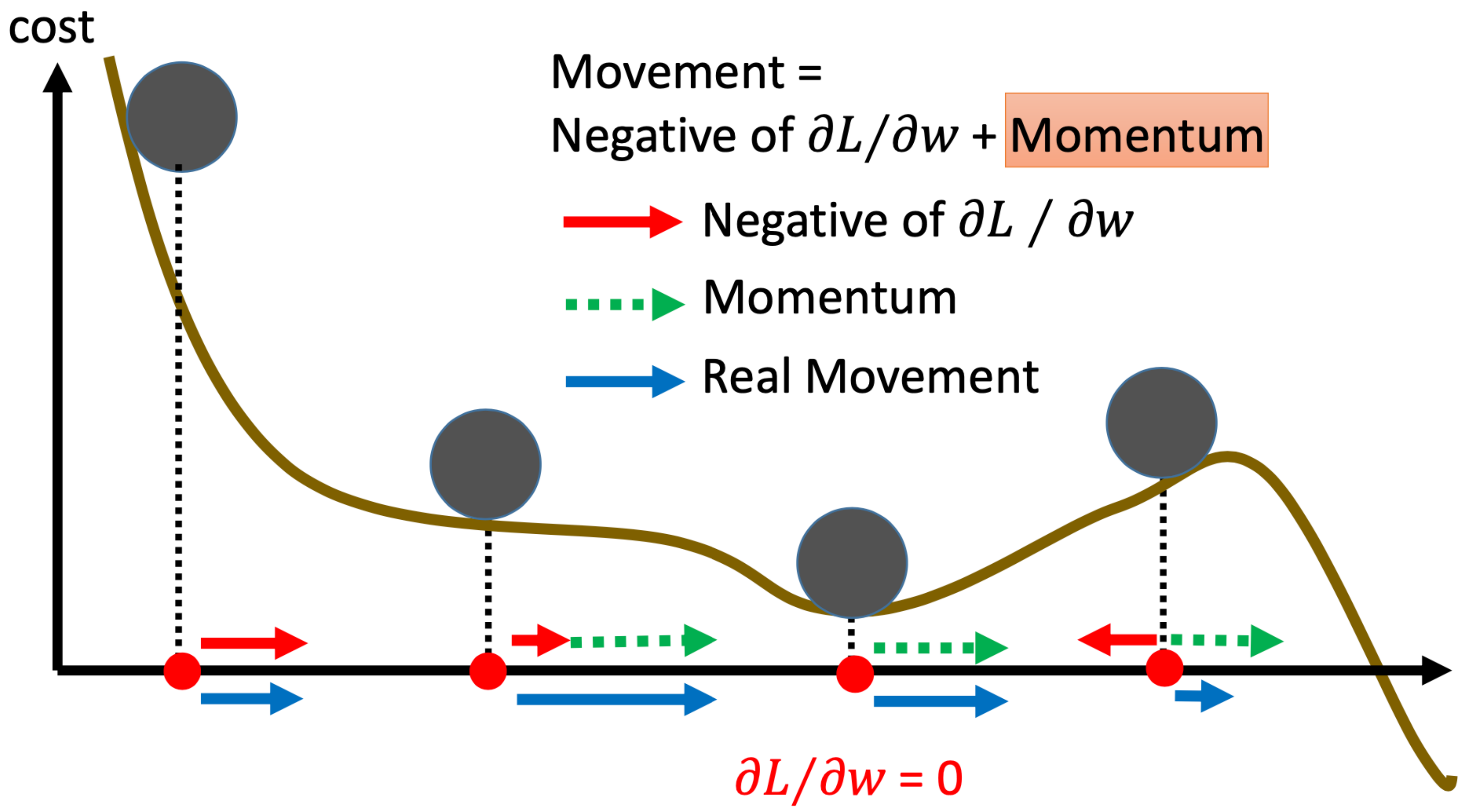
```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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
```



- The arrows indicate the steps that gradient descent would take at that point.
- The red path indicates the path followed by momentum.

Image source: Figure 8.5 & Algorithm 8.2, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

Momentum


cost
Movement =
Negative of $\partial L/\partial w$ + Momentum
Negative of $\partial L / \partial w$
Momentum
Real Movement
$\partial L/\partial w = 0$

Still not guarantee reaching global minima, but give some hope

Image source: Hung-yi Lee, Understanding Deep Learning in One Day

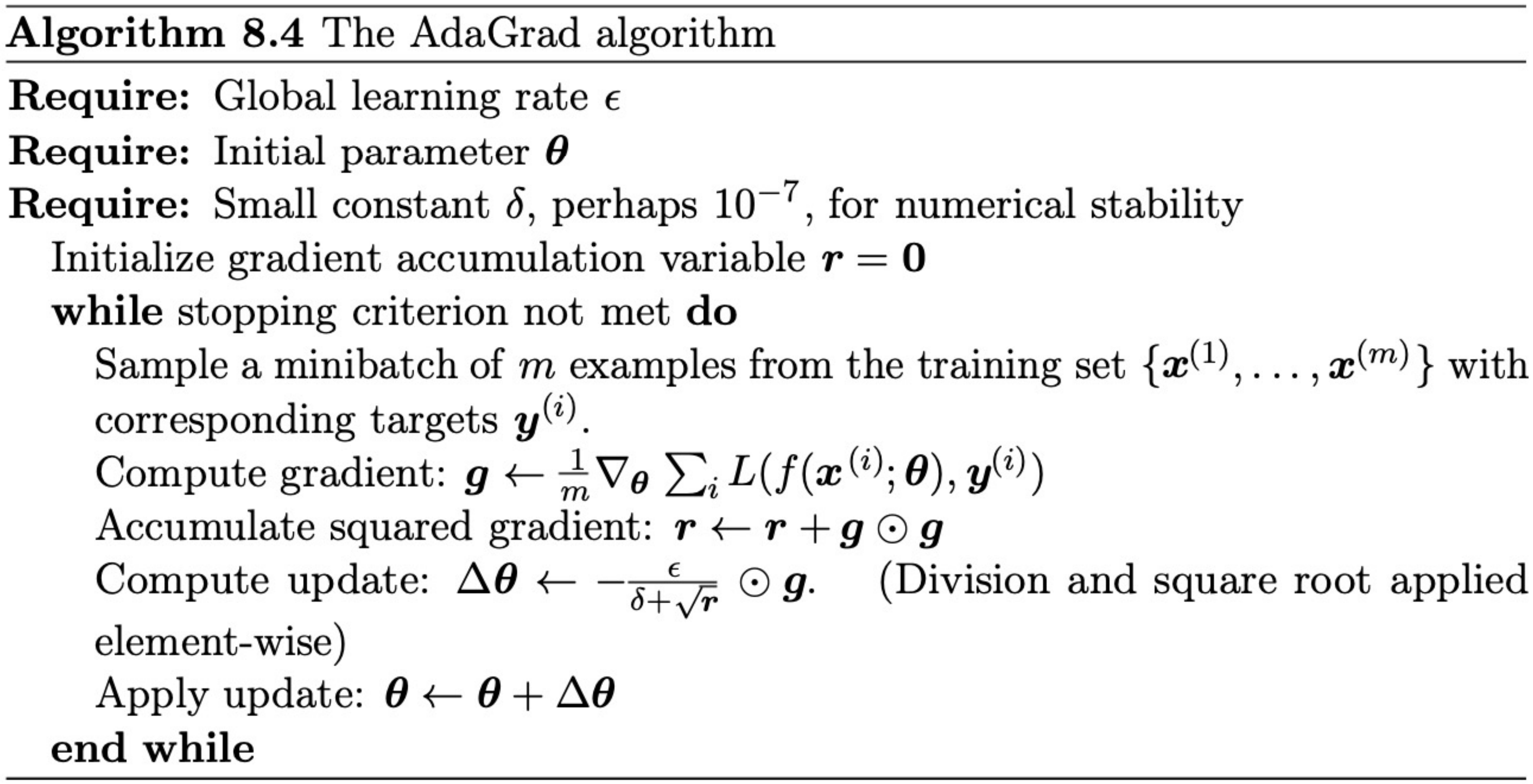
Adaptive Learning Methods

- Learning rate is one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance.
- Till now we assign the same learning rate to all parameters.
- Can we automatically adapt these learning rates for each single parameter?

AdaGrad

- Adapts the learning rates of all parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values.
  - Parameters that have large partial derivative: their learning rates are rapidly declined.
  - Parameters that have small partial derivative: their learning rates are slowly declined.
- AdaGrad performs well for some but not all deep learning models.

AdaGrad



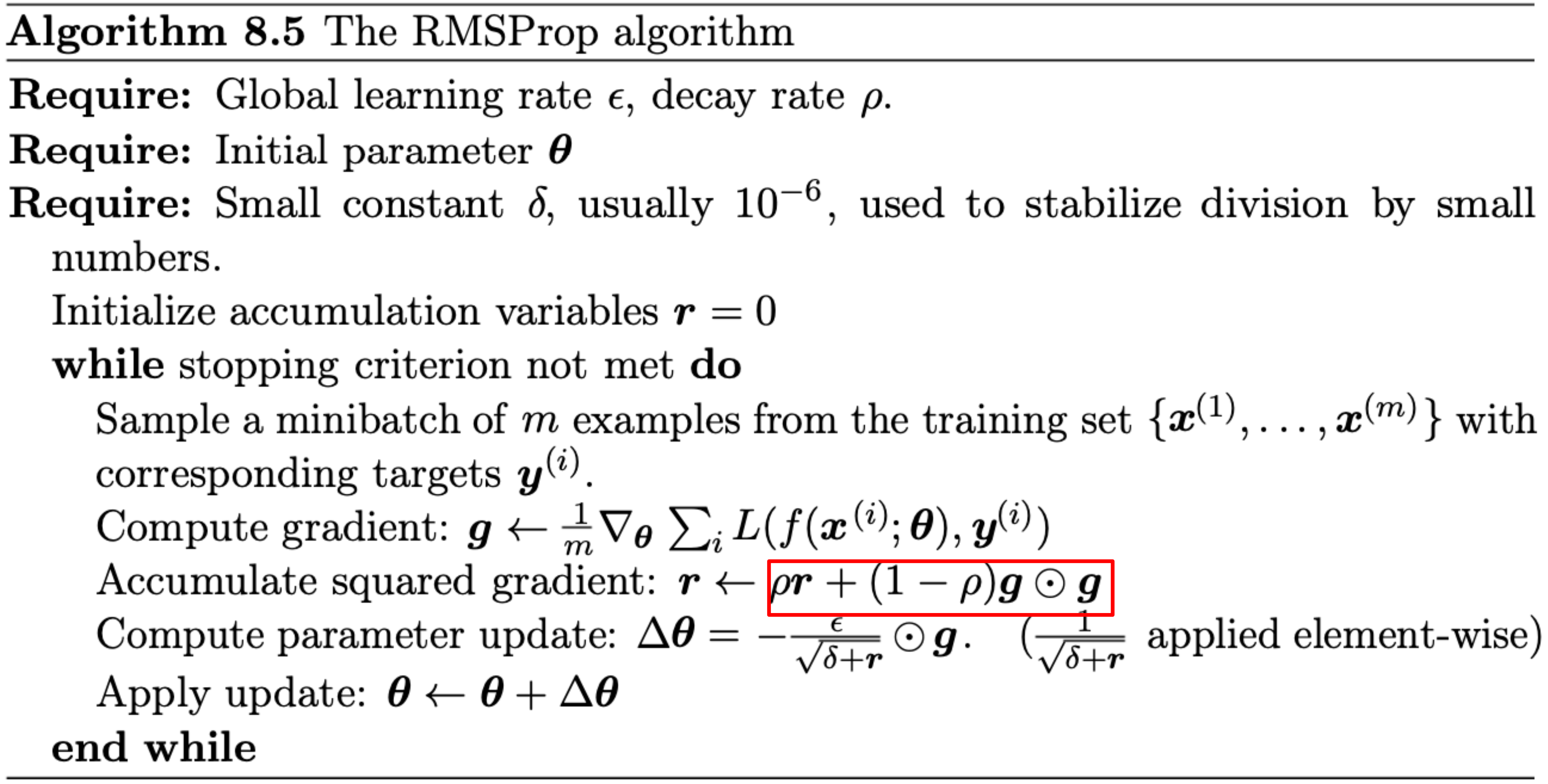
```
Algorithm 8.4 The AdaGrad algorithm
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    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
```

Image source: Algorithm 8.4, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure.
- RMSProp uses an exponentially decaying average to discard history from the extreme past.
  - Converge rapidly after finding a convex region.

RMSProp



Algorithm 8.5 The RMSProp algorithm

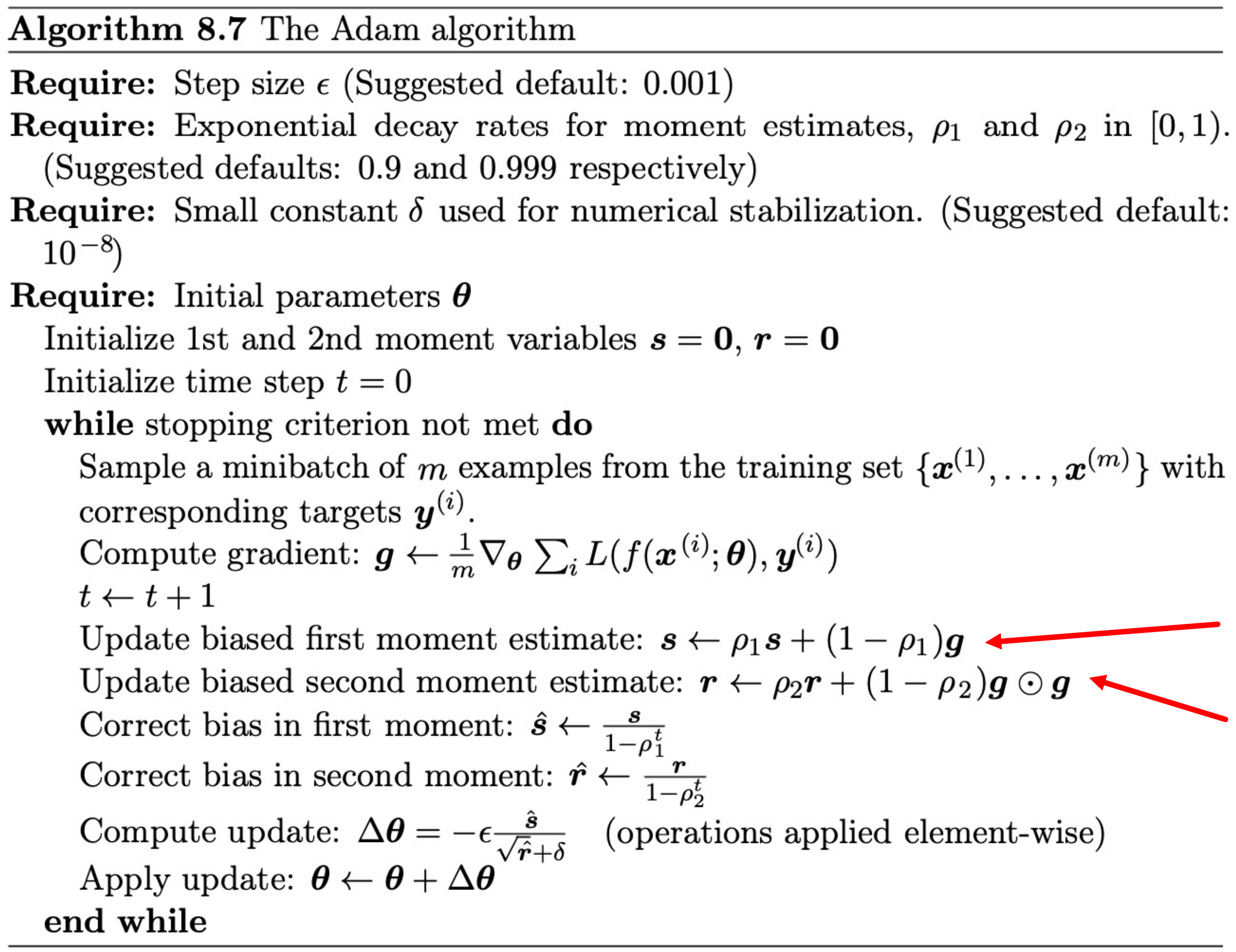
```
Require: Global learning rate $\epsilon$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small
  numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \boxed{\rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}}$
    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
```

Image source: Algorithm 8.5, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

Adam

- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments.
- Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

Adam



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
```

Momentum

RMSProp

Image source: Algorithm 8.7, Goodfellow, Bengio, and Courville, Deep Learning, Cambridge: MIT press, 2016.

Summary of Optimization Methods

- SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \hat{\boldsymbol{g}}$
- Momentum: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \hat{\boldsymbol{g}}, \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
- AdaGrad: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}, \quad \Delta\boldsymbol{\theta} \leftarrow -\frac{\varepsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}, \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
- RMSProp: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}, \quad \Delta\boldsymbol{\theta} \leftarrow -\frac{\varepsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}, \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
- Adam: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}, \quad \boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}, \quad \hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$$

$$\Delta\boldsymbol{\theta} \leftarrow -\varepsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}, \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$$

Summary of Optimization Methods

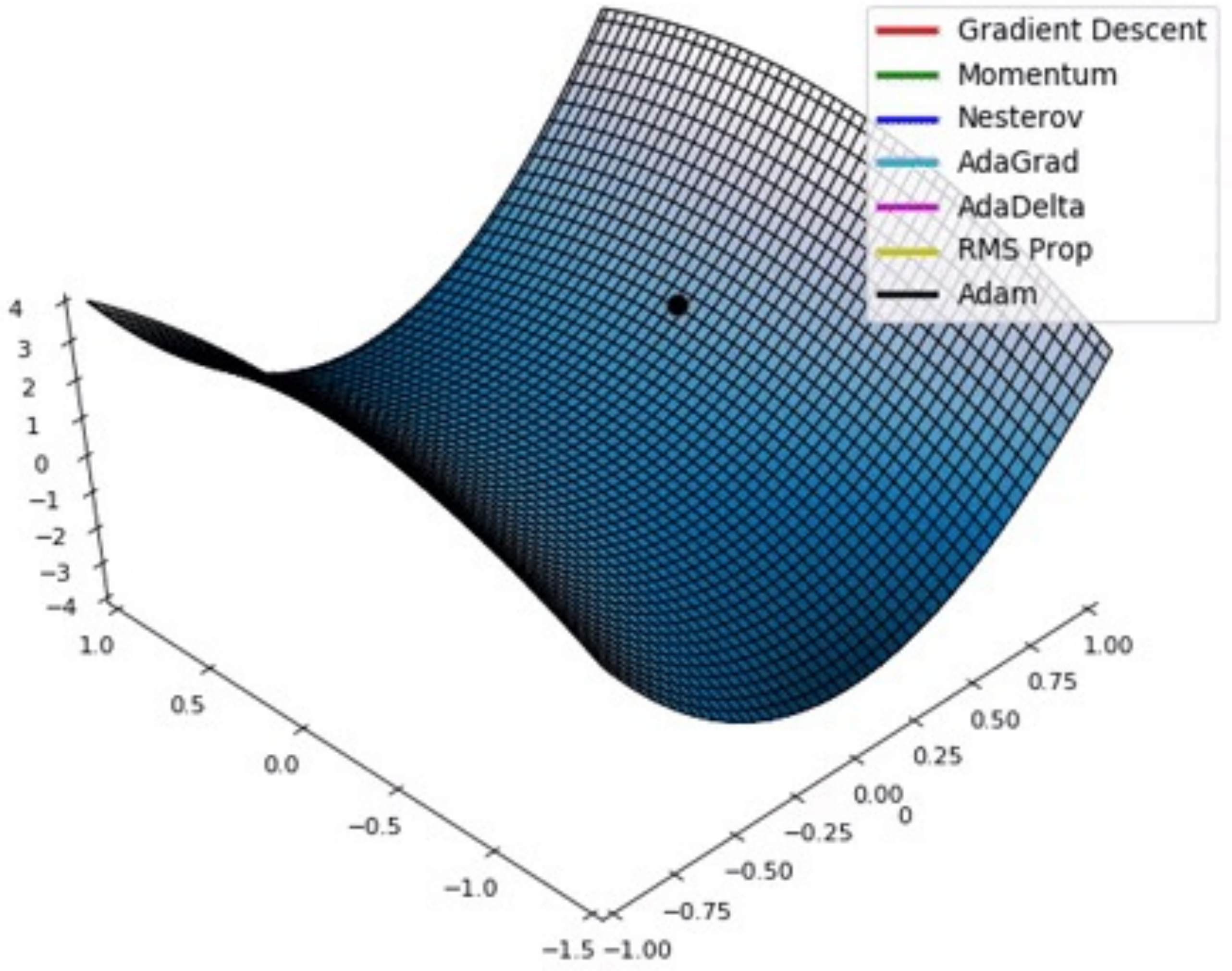


Image source: https://mlfromscratch.com/optimizers-explained/#/

Summary of Optimization Methods

- At this point, a natural question is:

  which optimization algorithm should one choose?

- Unfortunately, there is currently no consensus on this point.
- It depends on:
  - the complexity of the optimization problem,
  - user’s familiarity with the algorithm,
  - trail and error.

Batch Normalization

- The distribution of each layer’s input changes during training.
- Small changes to the network parameters amplify as the network becomes deeper.
- This phenomenon is called internal covariate shift.
- One solution is called batch normalization (BatchNorm).
  - It is one of the most exciting innovations in optimizing deep neural networks.

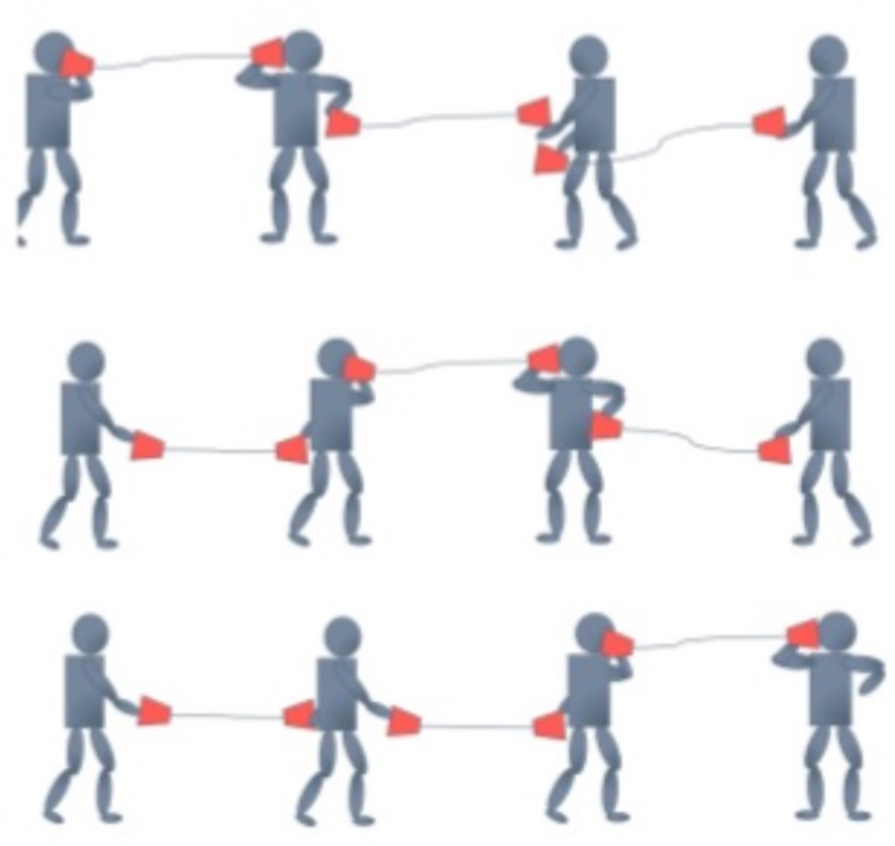


Image source: https://www.slideshare.net/ssuserd6984b/batch-normalization

Batch Normalization

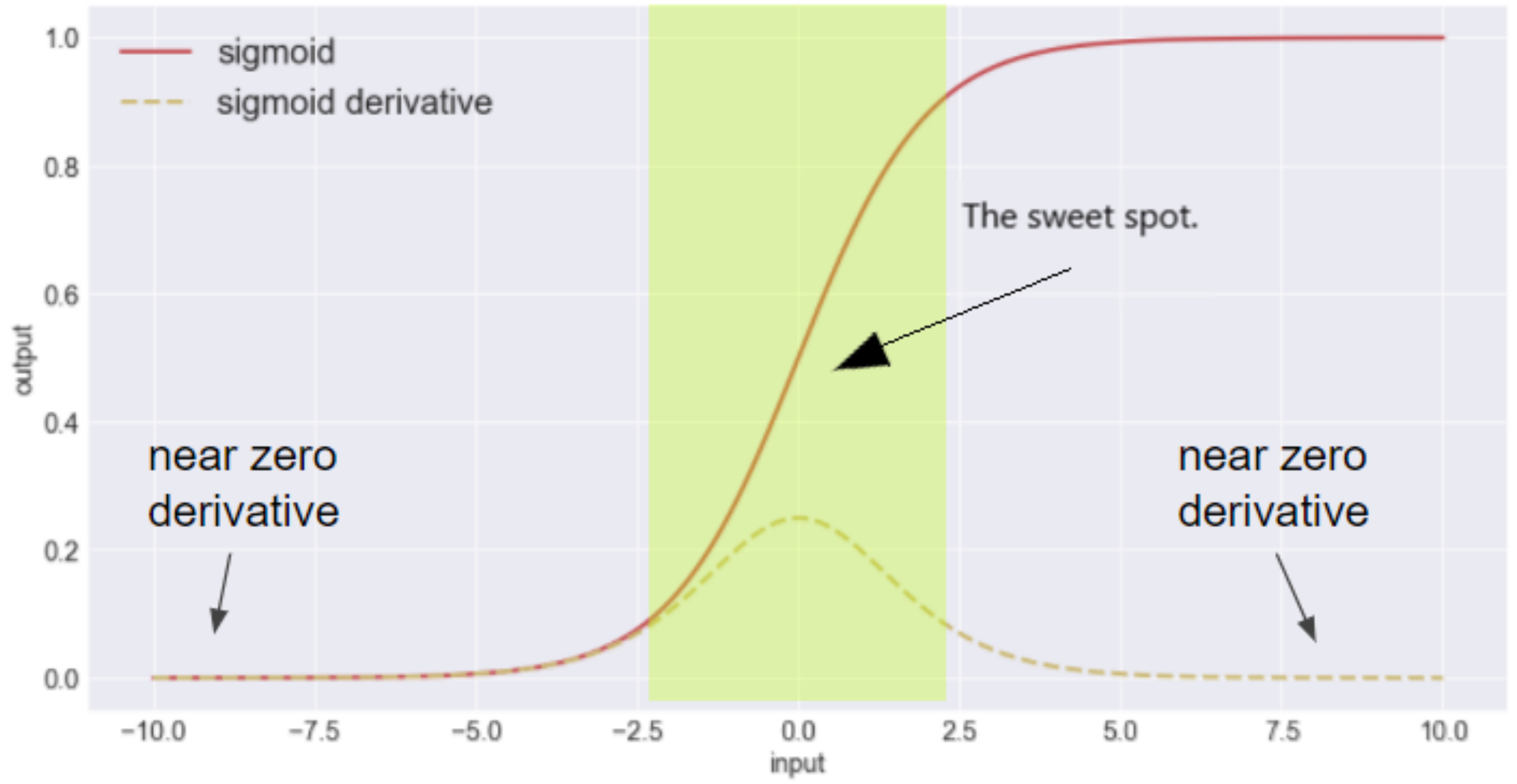
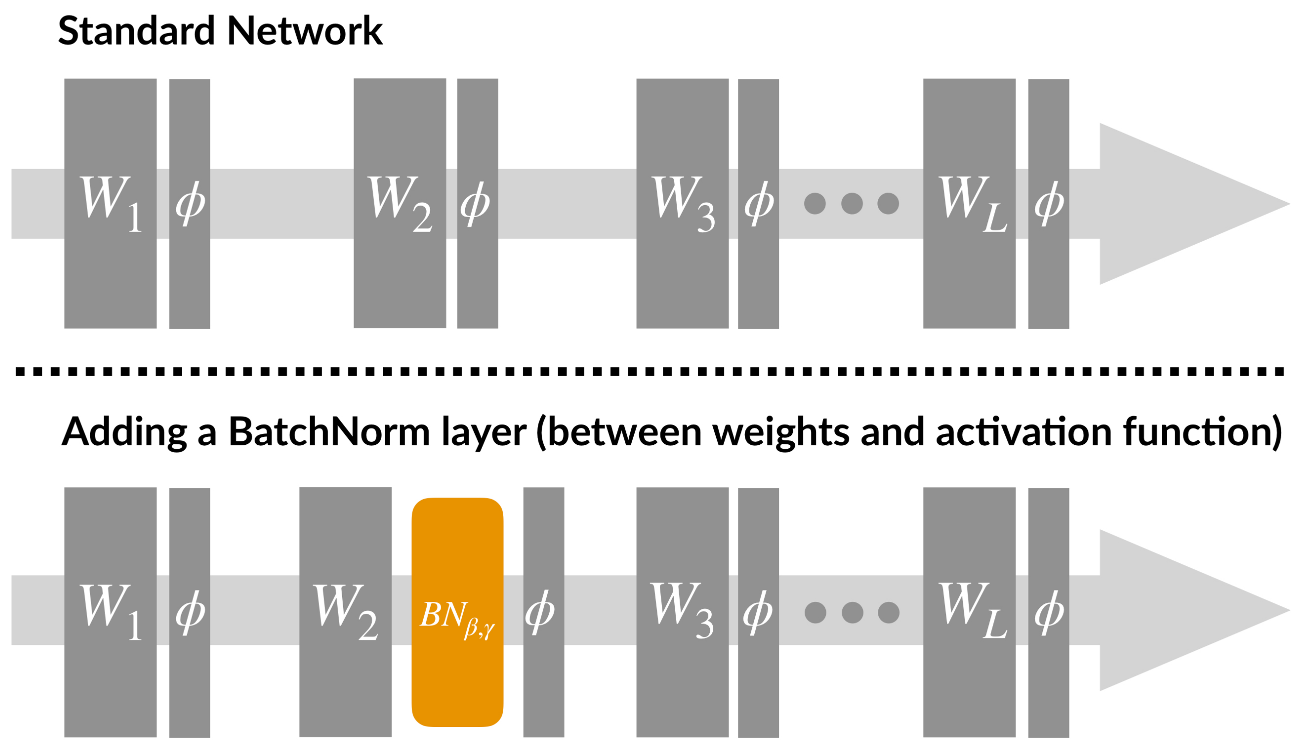


Image source: https://towardsdatascience.com/backpropagation-and-batch-normalization-in-feedforward-neural-networks-explained-901fd6e5393e

Batch Normalization


Standard Network
W_1 ϕ W_2 ϕ W_3 ϕ W_L ϕ
Adding a BatchNorm layer (between weights and activation function)
W_1 ϕ W_2 $BN_{\beta,\gamma}$ ϕ W_3 ϕ W_L ϕ

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

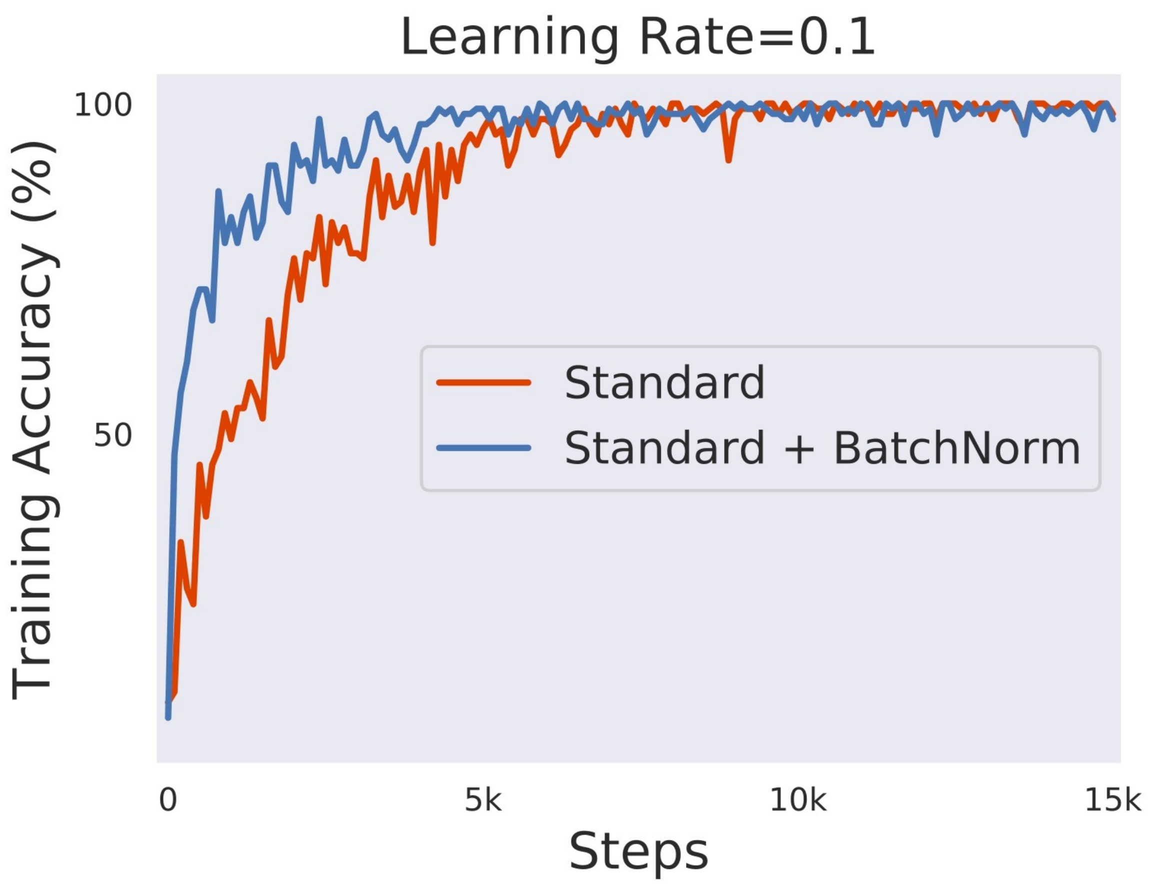
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Image source: https://gradientscience.org/batchnorm/

Batch Normalization

- Allow us to use much higher learning rates and be less careful about initialization.
- Improve training efficiency.
- Act as a regularizer, in some cases eliminating the need for Dropout.

Batch Normalization



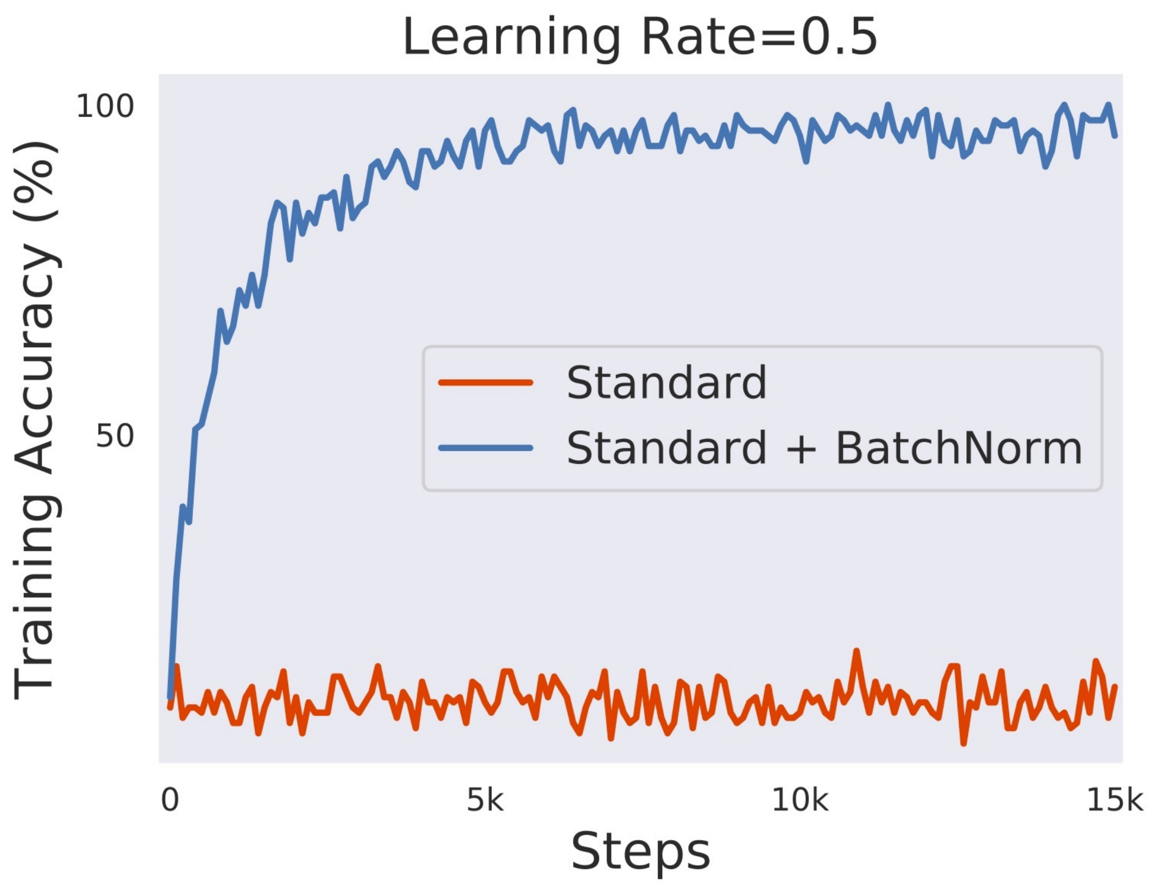


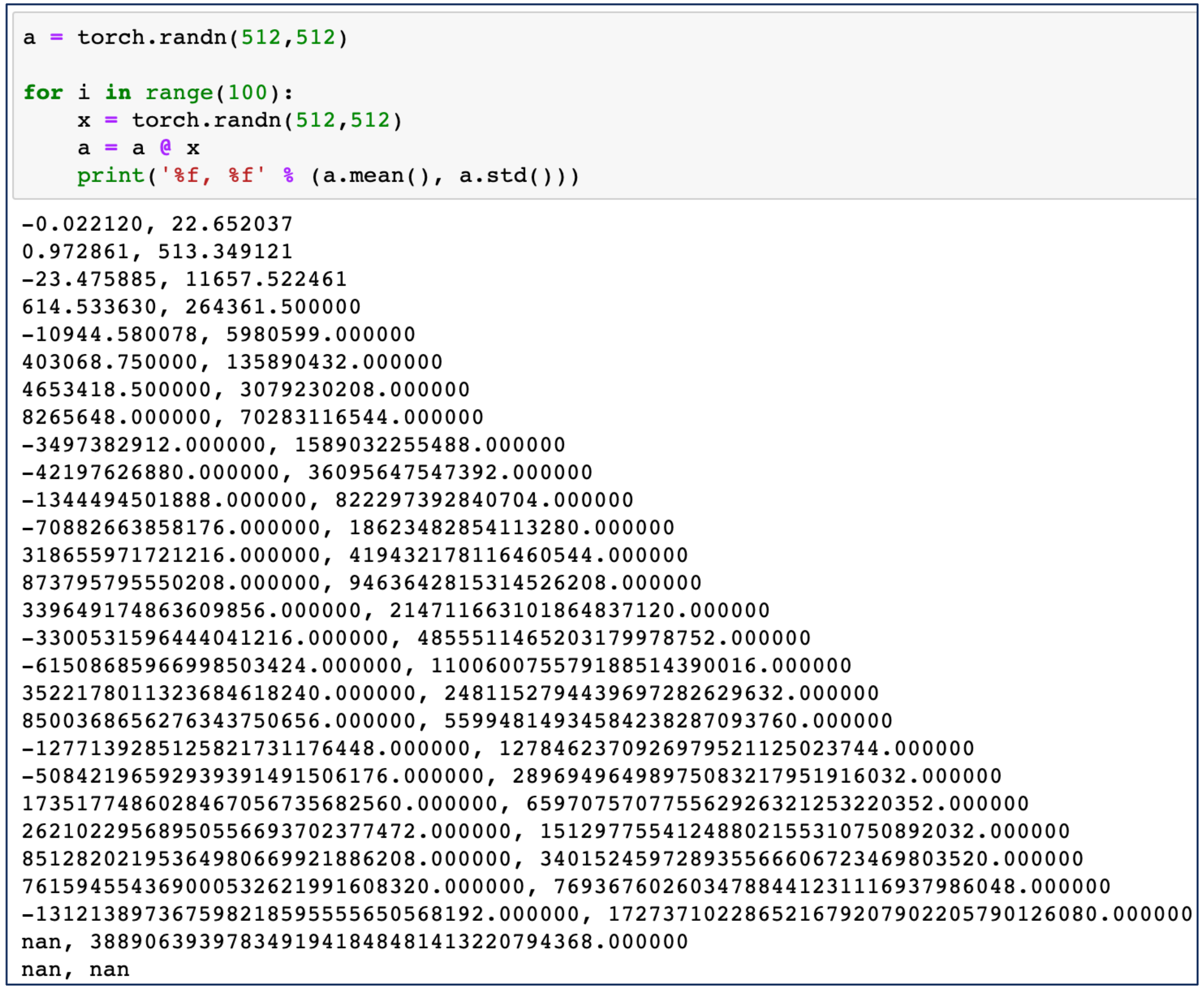
Image source: https://gradientscience.org/batchnorm/

Parameter Initialization Strategies

- In convex problems, no matter what the initialization, convergence is guaranteed.
- In the non-convex regime initialization is much more important.
  - Some parameter initialization can be unstable, not converge.
- Neural networks are not well understood to have principled, mathematically nice initialization strategies.
- We are only sure about one thing: the initial parameters need to “break symmetry” between different units.

Parameter Initialization Strategies

- Initialize weights by: $W_{i,j} \sim \mathcal{N}(0,1)$
- Then simply calculate matrix multiplication iteratively.
- It explodes…

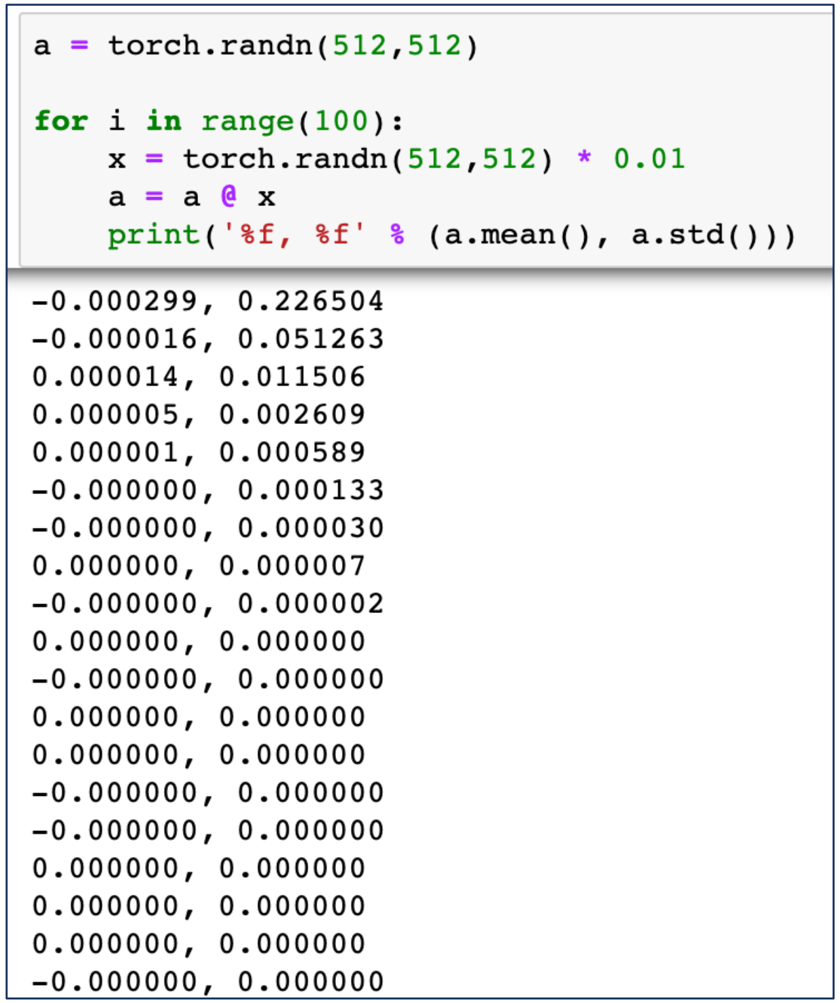


```python
a = torch.randn(512,512)

for i in range(100):
    x = torch.randn(512,512)
    a = a @ x
    print('%f, %f' % (a.mean(), a.std()))
```

```
-0.022120, 22.652037
0.972861, 513.349121
-23.475885, 11657.522461
614.533630, 264361.500000
-10944.580078, 5980599.000000
403068.750000, 135890432.000000
4653418.500000, 3079230208.000000
8265648.000000, 70283116544.000000
-3497382912.000000, 1589032255488.000000
-42197626880.000000, 36095647547392.000000
-1344494501888.000000, 822297392840704.000000
-70882663858176.000000, 18623482854113280.000000
318655971721216.000000, 419432178116460544.000000
873795795550208.000000, 9463642815314526208.000000
339649174863609856.000000, 214711663101864837120.000000
-3300531596444041216.000000, 4855511465203179978752.000000
-61508685966998503424.000000, 110060075579188514390016.000000
352217801132368461824 0.000000, 2481152794439697282629632.000000
8500368656276343750656.000000, 55994814934584238287093760.000000
-127713928512582173117 6448.000000, 1278462370926979521125023744.000000
-50842196592939391491506176.000000, 28969496498975083217951916032.000000
1735177486028467056735682560.000000, 659707570775562926321253220352.000000
26210229568950556693702377472.000000, 15129775541248802155310750892032.000000
851282021953649806699218862 08.000000, 340152459728935566606723469803520.000000
7615945543690005326219916083 20.000000, 7693676026034788441231116937986048.000000
-131213897367598218595555650568192.000000, 172737102286521679207902205790126080.000000
nan, 388906393978349194184848141322079 4368.000000
nan, nan
```

Parameter Initialization Strategies

- Shrink it by a constant: $W_{i,j} \sim \mathcal{N}(0,1) * 0.01$
- Quickly decrease to zero…

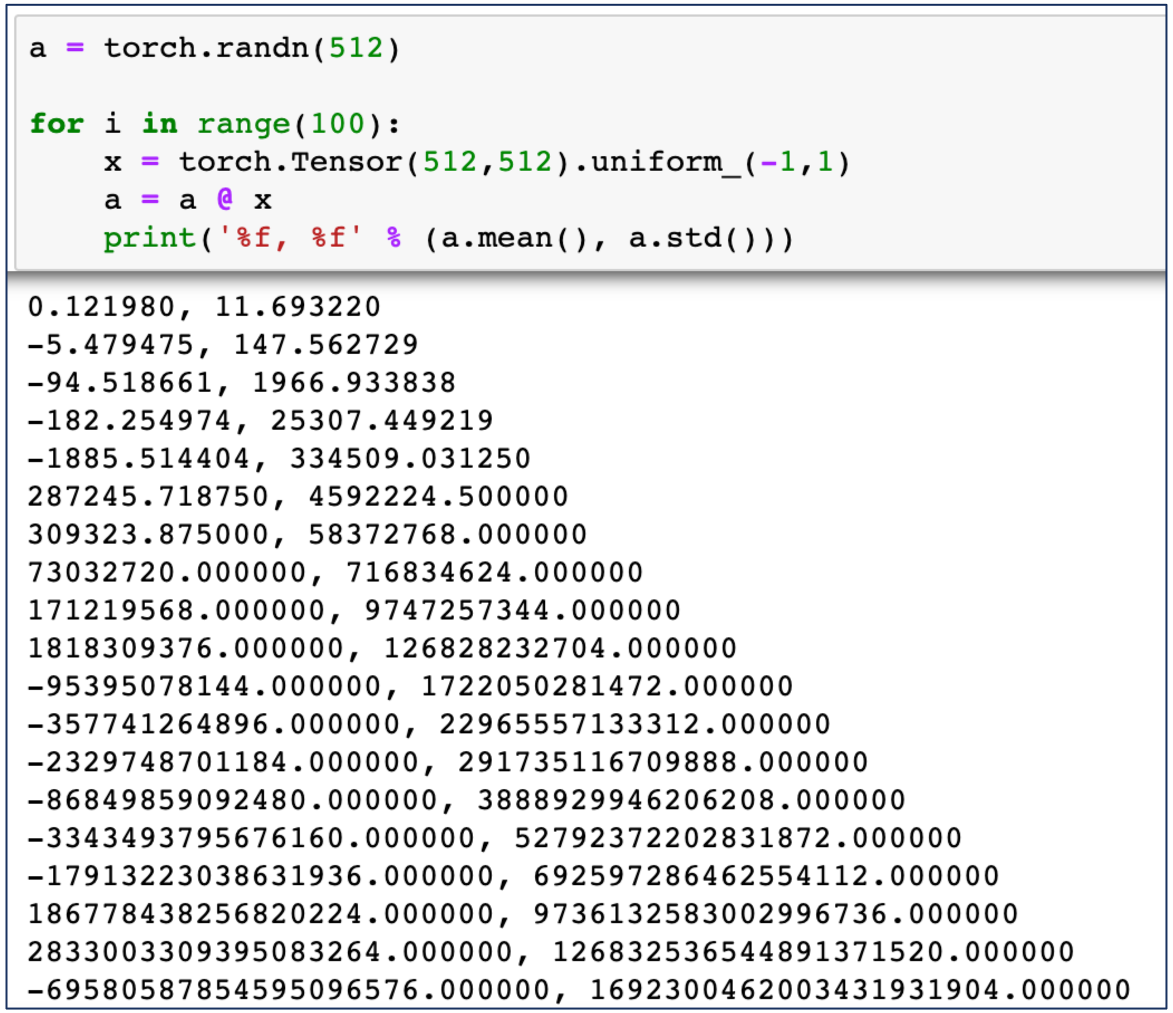


```
a = torch.randn(512,512)

for i in range(100):
    x = torch.randn(512,512) * 0.01
    a = a @ x
    print('%f, %f' % (a.mean(), a.std()))
```

```
-0.000299, 0.226504
-0.000016, 0.051263
0.000014, 0.011506
0.000005, 0.002609
0.000001, 0.000589
-0.000000, 0.000133
-0.000000, 0.000030
0.000000, 0.000007
-0.000000, 0.000002
0.000000, 0.000000
-0.000000, 0.000000
0.000000, 0.000000
0.000000, 0.000000
-0.000000, 0.000000
-0.000000, 0.000000
0.000000, 0.000000
0.000000, 0.000000
0.000000, 0.000000
-0.000000, 0.000000
```

Parameter Initialization Strategies

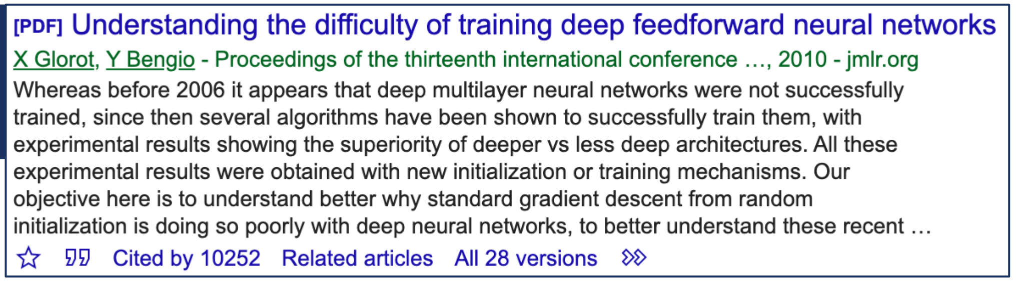
- How about uniform distribution?



```
a = torch.randn(512)

for i in range(100):
    x = torch.Tensor(512,512).uniform_(-1,1)
    a = a @ x
    print('%f, %f' % (a.mean(), a.std()))
```

```
0.121980, 11.693220
-5.479475, 147.562729
-94.518661, 1966.933838
-182.254974, 25307.449219
-1885.514404, 334509.031250
287245.718750, 4592224.500000
309323.875000, 58372768.000000
73032720.000000, 716834624.000000
171219568.000000, 9747257344.000000
1818309376.000000, 126828232704.000000
-95395078144.000000, 1722050281472.000000
-357741264896.000000, 22965557133312.000000
-2329748701184.000000, 291735116709888.000000
-86849859092480.000000, 3888929946206208.000000
-334349379567616O.000000, 52792372202831872.000000
-17913223038631936.000000, 692597286462554112.000000
186778438256820224.000000, 9736132583002996736.000000
2833003309395083264.000000, 126832536544891371520.000000
-6958058785459509657 6.000000, 1692300462003431931904.000000
```

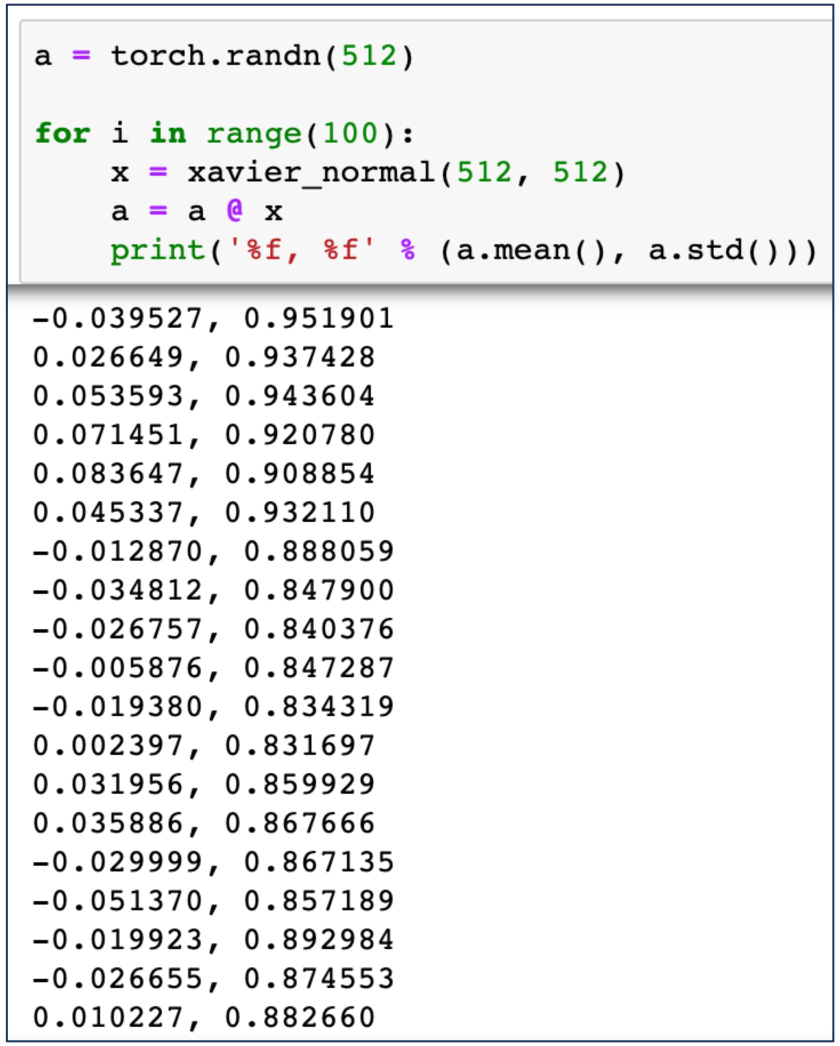
Xavier Initialization


[PDF] Understanding the difficulty of training deep feedforward neural networks
X Glorot, Y Bengio - Proceedings of the thirteenth international conference …, 2010 - jmlr.org
Whereas before 2006 it appears that deep multilayer neural networks were not successfully trained, since then several algorithms have been shown to successfully train them, with experimental results showing the superiority of deeper vs less deep architectures. All these experimental results were obtained with new initialization or training mechanisms. Our objective here is to understand better why standard gradient descent from random initialization is doing so poorly with deep neural networks, to better understand these recent …
☆ 99 Cited by 10252 Related articles All 28 versions »

- Xavier Initialization (uniform and normal).
- For a fully connected layer with m inputs and n outputs:

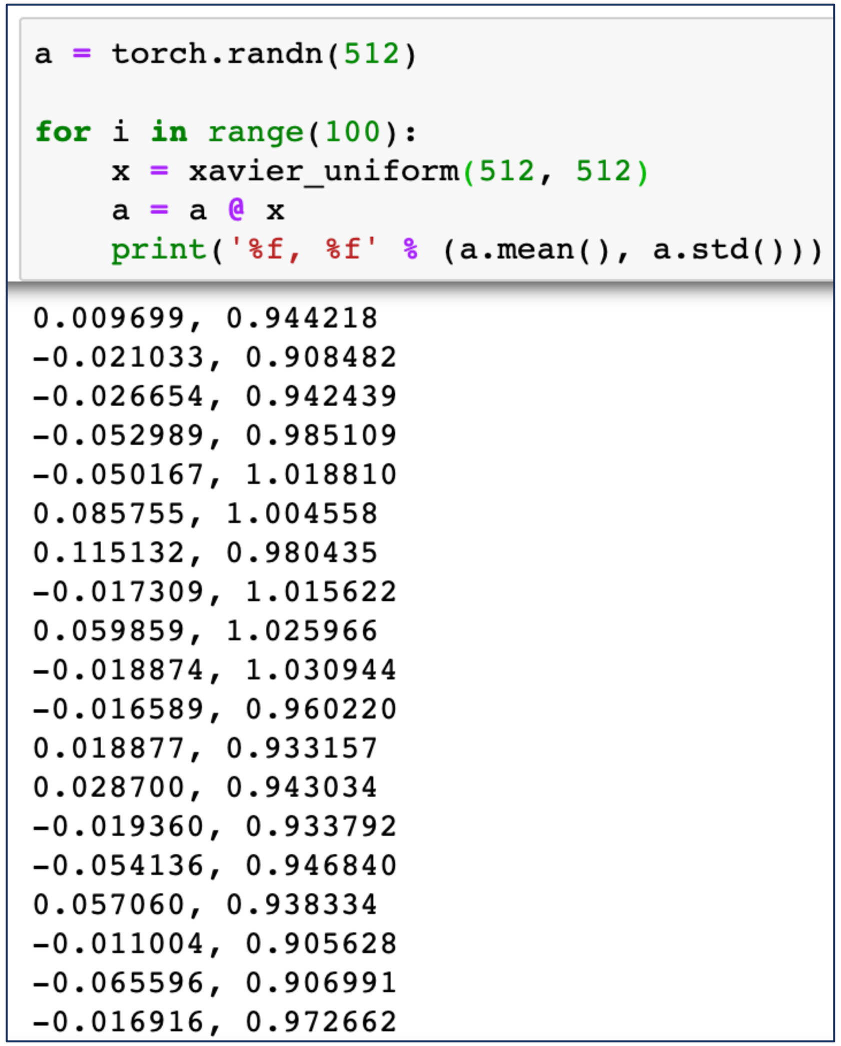
$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right), \qquad W_{i,j} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{m+n}}\right)$$

Xavier Initialization



```
a = torch.randn(512)

for i in range(100):
    x = xavier_normal(512, 512)
    a = a @ x
    print('%f, %f' % (a.mean(), a.std()))
```

```
-0.039527, 0.951901
0.026649, 0.937428
0.053593, 0.943604
0.071451, 0.920780
0.083647, 0.908854
0.045337, 0.932110
-0.012870, 0.888059
-0.034812, 0.847900
-0.026757, 0.840376
-0.005876, 0.847287
-0.019380, 0.834319
0.002397, 0.831697
0.031956, 0.859929
0.035886, 0.867666
-0.029999, 0.867135
-0.051370, 0.857189
-0.019923, 0.892984
-0.026655, 0.874553
0.010227, 0.882660
```

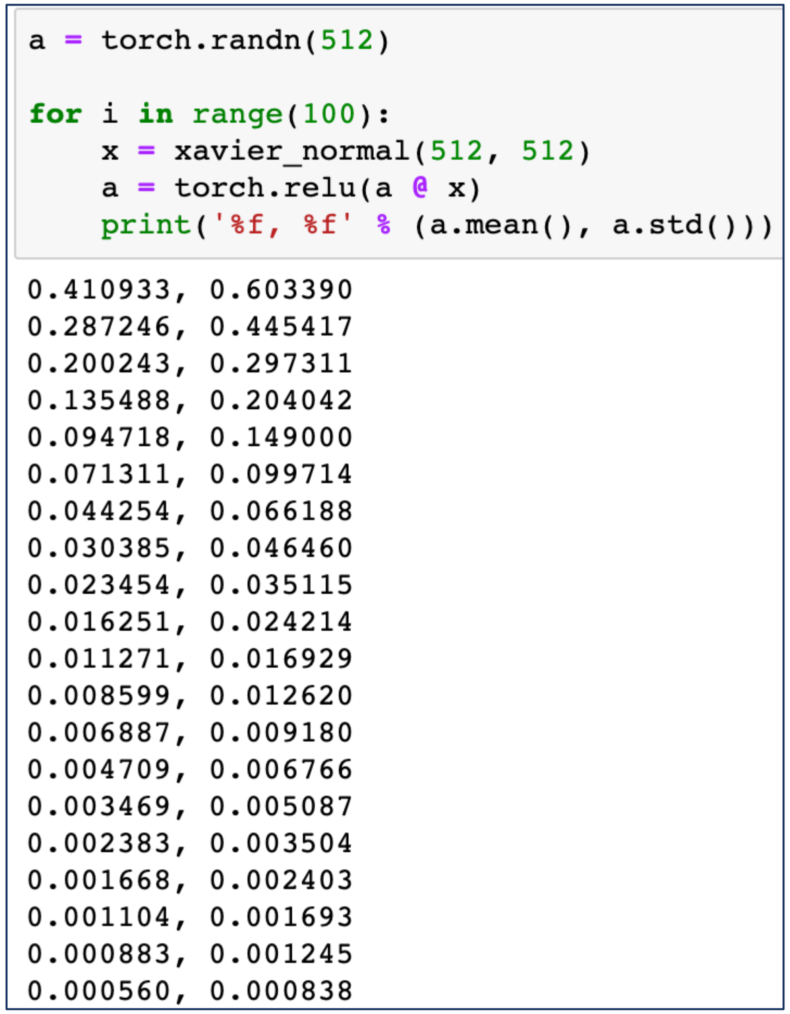


```
a = torch.randn(512)

for i in range(100):
    x = xavier_uniform(512, 512)
    a = a @ x
    print('%f, %f' % (a.mean(), a.std()))
```

```
0.009699, 0.944218
-0.021033, 0.908482
-0.026654, 0.942439
-0.052989, 0.985109
-0.050167, 1.018810
0.085755, 1.004558
0.115132, 0.980435
-0.017309, 1.015622
0.059859, 1.025966
-0.018874, 1.030944
-0.016589, 0.960220
0.018877, 0.933157
0.028700, 0.943034
-0.019360, 0.933792
-0.054136, 0.946840
0.057060, 0.938334
-0.011004, 0.905628
-0.065596, 0.906991
-0.016916, 0.972662
```

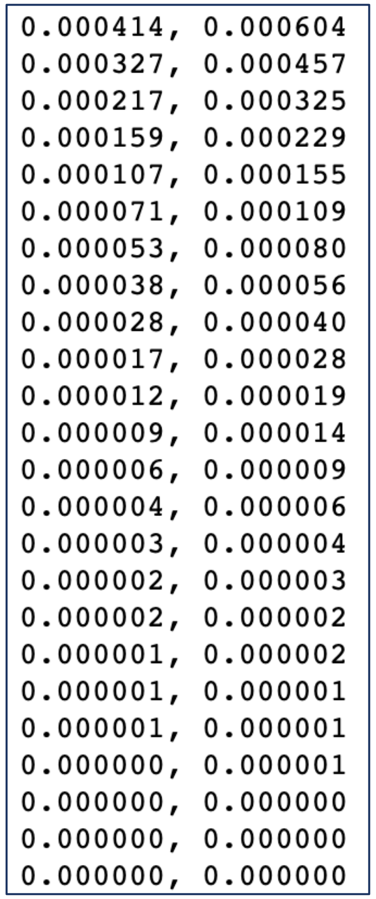
Xavier Initialization

- Doesn’t work well with ReLU.



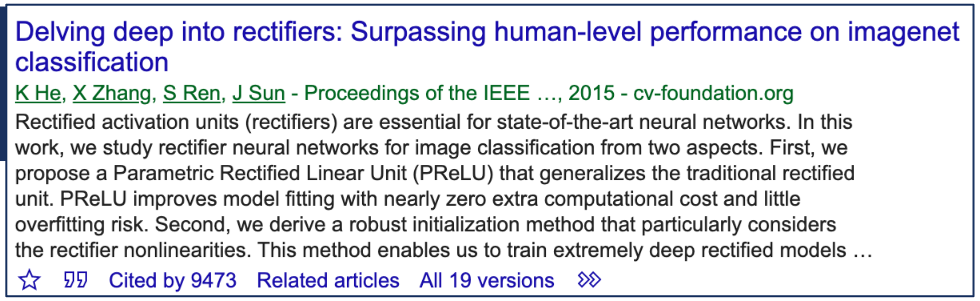
```
a = torch.randn(512)

for i in range(100):
    x = xavier_normal(512, 512)
    a = torch.relu(a @ x)
    print('%f, %f' % (a.mean(), a.std()))
```

```
0.410933, 0.603390
0.287246, 0.445417
0.200243, 0.297311
0.135488, 0.204042
0.094718, 0.149000
0.071311, 0.099714
0.044254, 0.066188
0.030385, 0.046460
0.023454, 0.035115
0.016251, 0.024214
0.011271, 0.016929
0.008599, 0.012620
0.006887, 0.009180
0.004709, 0.006766
0.003469, 0.005087
0.002383, 0.003504
0.001668, 0.002403
0.001104, 0.001693
0.000883, 0.001245
0.000560, 0.000838
```



```
0.000414, 0.000604
0.000327, 0.000457
0.000217, 0.000325
0.000159, 0.000229
0.000107, 0.000155
0.000071, 0.000109
0.000053, 0.000080
0.000038, 0.000056
0.000028, 0.000040
0.000017, 0.000028
0.000012, 0.000019
0.000009, 0.000014
0.000006, 0.000009
0.000004, 0.000006
0.000003, 0.000004
0.000002, 0.000003
0.000002, 0.000002
0.000001, 0.000002
0.000001, 0.000001
0.000001, 0.000001
0.000000, 0.000001
0.000000, 0.000000
0.000000, 0.000000
0.000000, 0.000000
```

Kaiming Initialization



Delving deep into rectifiers: Surpassing human-level performance on imagenet classification

K He, X Zhang, S Ren, J Sun - Proceedings of the IEEE …, 2015 - cv-foundation.org

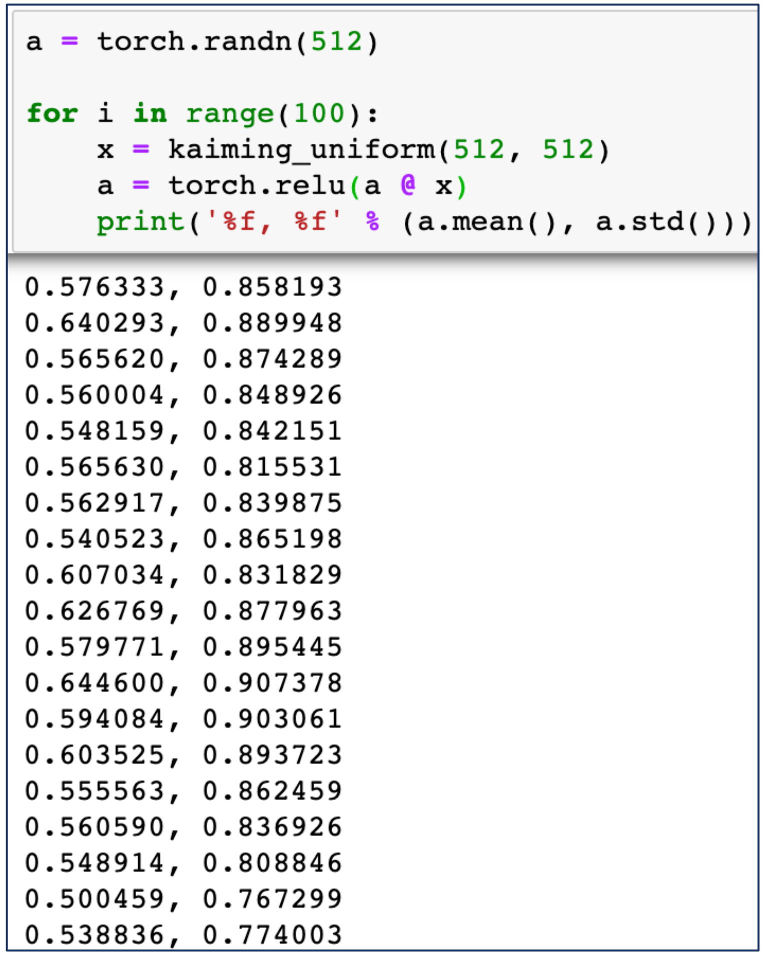
Rectified activation units (rectifiers) are essential for state-of-the-art neural networks. In this work, we study rectifier neural networks for image classification from two aspects. First, we propose a Parametric Rectified Linear Unit (PReLU) that generalizes the traditional rectified unit. PReLU improves model fitting with nearly zero extra computational cost and little overfitting risk. Second, we derive a robust initialization method that particularly considers the rectifier nonlinearities. This method enables us to train extremely deep rectified models …

☆ 99 Cited by 9473 Related articles All 19 versions »

- Kaiming Initialization (uniform and normal):

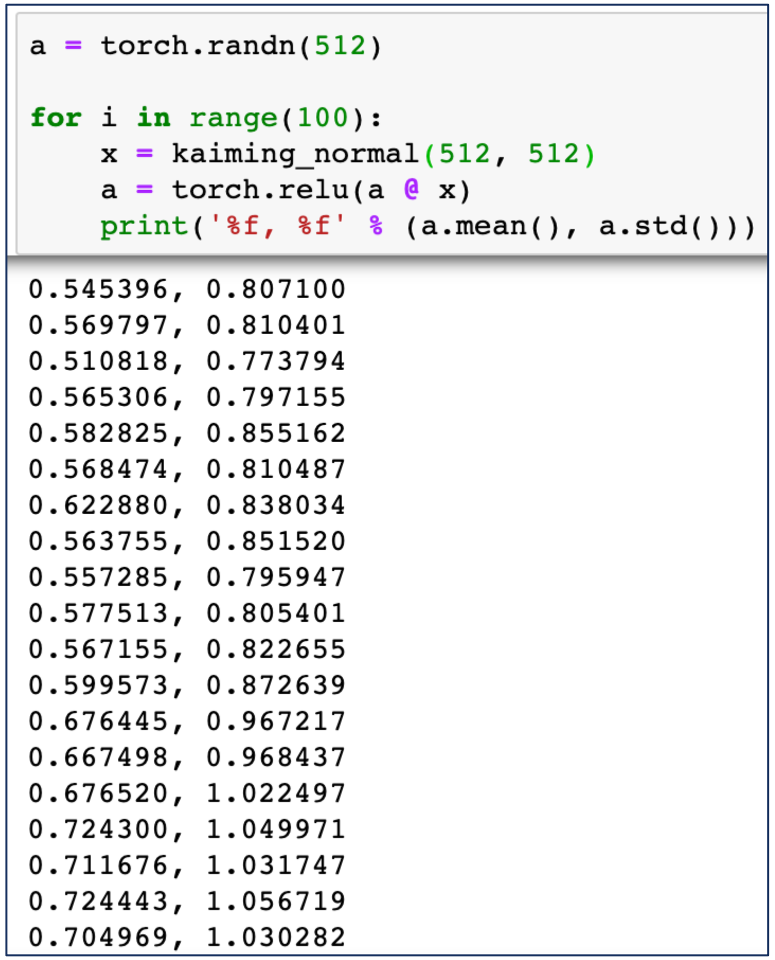
$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m}}, \sqrt{\frac{6}{m}}\right), \qquad W_{i,j} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{m}}\right)$$

Kaiming Initialization



```
a = torch.randn(512)

for i in range(100):
    x = kaiming_uniform(512, 512)
    a = torch.relu(a @ x)
    print('%f, %f' % (a.mean(), a.std()))
```

```
0.576333, 0.858193
0.640293, 0.889948
0.565620, 0.874289
0.560004, 0.848926
0.548159, 0.842151
0.565630, 0.815531
0.562917, 0.839875
0.540523, 0.865198
0.607034, 0.831829
0.626769, 0.877963
0.579771, 0.895445
0.644600, 0.907378
0.594084, 0.903061
0.603525, 0.893723
0.555563, 0.862459
0.560590, 0.836926
0.548914, 0.808846
0.500459, 0.767299
0.538836, 0.774003
```



```
a = torch.randn(512)

for i in range(100):
    x = kaiming_normal(512, 512)
    a = torch.relu(a @ x)
    print('%f, %f' % (a.mean(), a.std()))
```

```
0.545396, 0.807100
0.569797, 0.810401
0.510818, 0.773794
0.565306, 0.797155
0.582825, 0.855162
0.568474, 0.810487
0.622880, 0.838034
0.563755, 0.851520
0.557285, 0.795947
0.577513, 0.805401
0.567155, 0.822655
0.599573, 0.872639
0.676445, 0.967217
0.667498, 0.968437
0.676520, 1.022497
0.724300, 1.049971
0.711676, 1.031747
0.724443, 1.056719
0.704969, 1.030282
```

Conclusion

After this lecture, you should know:

- What is underfitting and overfitting?
- What is regularization?
- What are the commonly used regularization methods?
- How do optimization methods work?
- What is batch normalization?
- How to initialize parameters?

Suggested Reading

- Deep learning textbook chapter 7-8.
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting
- Sparsity and the LASSO
- Training with Noise is Equivalent to Tikhonov Regularization
- Why Momentum Really Works
- Large Scale Machine Learning with Stochastic Gradient Descent
- On the Importance of Initialization and Momentum in Deep Learning
- Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺