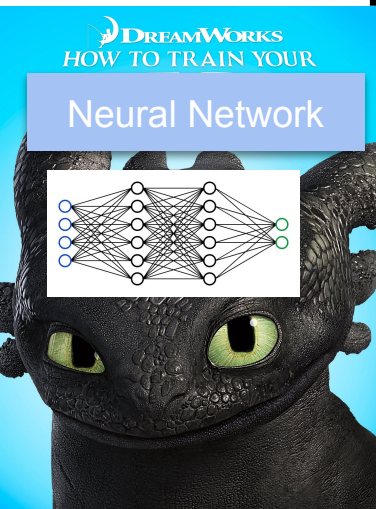


# CSC413 Tutorial: Optimization for Machine Learning



*"How to train your neural network"*

Harris Chan<sup>1</sup>

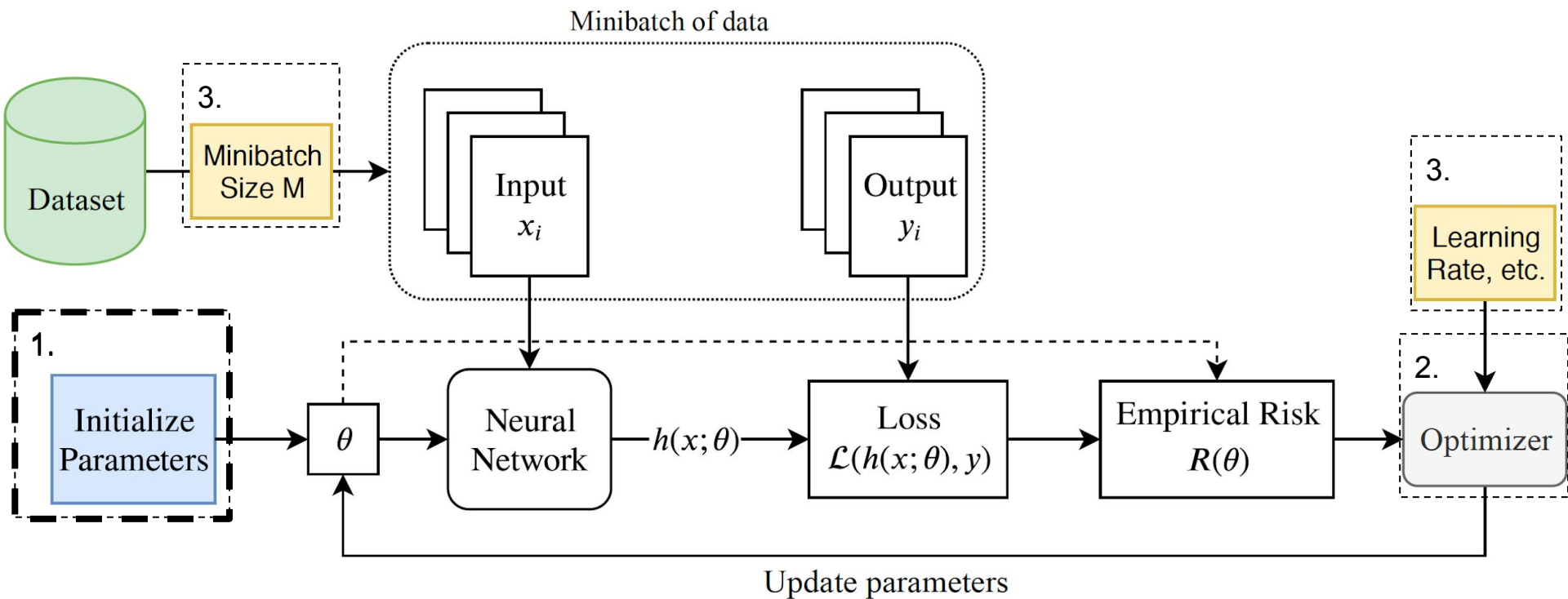
January 28, 2020

<sup>1</sup>Based on tutorials/slides by Ladislav Rampasek, Jake Snell, Kevin Swersky, Shenlong Wang & others

# Overview

- Review: Overall Training Loop
- Initialization
- Optimization
  - Gradient Descent
  - Momentum, Nesterov Accelerated Momentum
  - Learning Rate Schedulers: Adagrad, RMSProp, Adam
- Hyperparameter tuning: learning rate, batch size, regularization
- Jupyter/Colab Demo in PyTorch

# Neural Network Training Loop



# Initialization of Parameters

Initial parameters of the neural network can affect the **gradients** and learning

## Idea 1: **Constant** initialization

- *Result:* For fully connected layers: identical gradients, identical neurons. Bad!

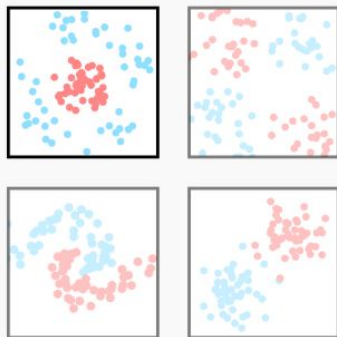
## Idea 2: **Random** weights, to break symmetry

- Too **large** of initialization: **exploding** gradients
- Too **small** of initialization: **vanishing** gradients

# Interactive Demo: Initialization

## 1. Choose input dataset

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.



Node Type: Input Relu Sigmoid

## 2. Choose initialization method

Select an initialization method for the values of your neural network parameters<sup>1</sup>.

☐ Zero ☐ Too small ☐ Appropriate ☒ Too large

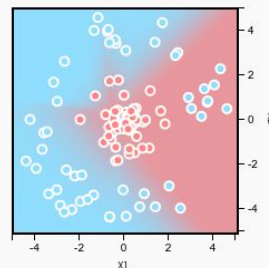
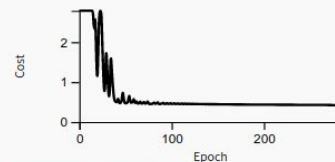


Select whether to visualize the weights or gradients of the network above.

☐ Weight ☒ Gradient

## 3. Train the network.

Observe the cost function and the decision boundary.



# Initialization: Calibrate the variance

Two popular initialization schemes:

1: **Xavier** Init: For **Tanh** activation

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}}) \quad \text{or} \quad W^{[l]} \sim \mathcal{N}(0, \frac{2}{n^{[l-1]} + n^{[l]}})$$
$$b^{[l]} = 0$$

2. **Kaiming He** Init: **ReLU** activation

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{2}{n^{[l-1]}})$$
$$b^{[l]} = 0$$

# of neurons in layer  
 $l-1$

1: [Glorot & Bengio: Understanding the difficulty of training deep feedforward neural networks](#)

2: [He et al.: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#)

# Initialization: Xavier Initialization Intuition

For networks with **Tanh** activation functions, Xavier Init aim for the following behaviour of the activations:

1. **Variance** of activation  $\sim$  **constant** across every layer

$$\text{Var}(a^l) = \text{Var}(a^{l-1})$$

2. **Mean** of the activation and weights = **zero**

$$\mathbb{E}[w^l] = 0 \qquad \mathbb{E}[a^l] = 0$$

# Initialization: Xavier Initialization Proof

For networks with **Tanh** activation functions, and **fully connected** layers:

$$z_k^{(l)} = \sum_i^{n^{l-1}} w_{ki}^{(l)} a_i^{(l-1)} \quad \Bigg| \quad \underset{\text{Activations}}{a_k^{(l)}} = f(\underset{\text{Layer}}{z_k^{(l)}})$$

Express **variance of activations** in **layer l** as function of **variance of weights**:

$$\begin{aligned} \text{Var}(a_k^{(l)}) &= \text{Var}(z_k^{(l)}) && \text{In linear region for } f = \tanh \\ &= \text{Var}\left(\sum_i^{n^{l-1}} w_{ki}^{(l)} a_i^{(l-1)}\right) \\ &= \sum_i^{n^{l-1}} \text{Var}(w_{ki}^{(l)} a_i^{(l-1)}) \end{aligned}$$

Assume product terms are independent, bring summation outside



Initialize  
Parameters

# Initialization: Xavier Initialization Proof

Assuming that the weights and activations are *independent* at init, apply identity:

$$\mathbb{E}[w^l] = 0$$

Assume mean  
of weights = 0

$$\text{Var}(AB) = \mathbb{E}[A]^2 \text{Var}(B) + \mathbb{E}[A]^2 \text{Var}(A) + \text{Var}(A)\text{Var}(B)$$

Assume mean  
of inputs = 0

**Note:** Not true  
for ReLU

$$\text{Var}(a_k^{(l)})$$

$$= \sum_i^{n^{l-1}} [\mathbb{E}(w_{ki}^{(l)})]^2 \text{Var}(a_i^{(l-1)}) + [\mathbb{E}(a_i^{(l-1)})]^2 \text{Var}(w_{ki}^{(l)}) + \text{Var}(w_{ki}^{(l)}) \text{Var}(a_i^{(l-1)})$$

$$= \sum_i^{n^{l-1}} \text{Var}(w_{ki}^{(l)}) \text{Var}(a_i^{(l-1)})$$

$$= n^{l-1} \text{Var}(w_k^{(l)}) \text{Var}(a^{(l-1)})$$

Weights are iid  
Inputs are iid

$$\text{Var}(w^{(l)}) = \frac{1}{n^{l-1}}$$

$$\text{Var}(a^{(l)}) = n^{l-1} \text{Var}(w^{(l)}) \text{Var}(a^{(l-1)})$$

# Interactive Demo: Initialization Schemes

## 1. Load your dataset

Load 10,000 handwritten digits images (MNIST).

Load MNIST (100%)

## 2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters<sup>3</sup>.

- ☐ Zero
 ☐ Uniform
 ☒ Xavier
 ☐ Standard Normal

## 3. Train the network and observe

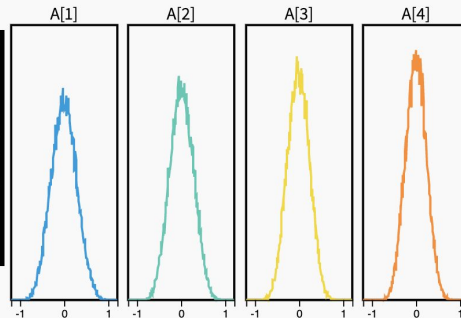
The grid below refers to the input images, Blue squares represent correctly classified images. Red squares represent misclassified images.



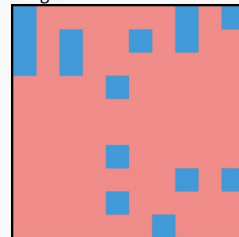
Input batch of 100 images



Batch: 0 Epoch: 0

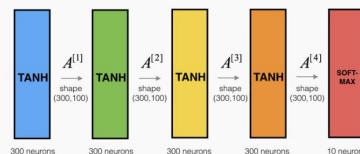


Output predictions of 100 images



Misclassified: 85/100 Cost: 2.27

$X = A^{[0]}$   
 Batch of 100 grayscale images of shape 28x28  
 X.shape = (784, 100) because 784 = 28x28



$\hat{y} = A^{[5]}$   
 output probability over 10  
 classes for a batch of  
 100 images  
 y\_hat.shape = (10, 100)

Initialize  
Parameters

# Batch Normalization Layer

Can we compensate for bad initializations in some other way?

## BatchNorm's Idea:

- Explicitly normalize the activations of each layer to be unit Gaussian.
- Apply immediately *after* fully connected/conv layers and *before* non-linearities
- Learn an additional **scale** and **shift** and running statistics for test time

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

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

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

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

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

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

**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

# Batch Normalization Layer

- BatchNorm significantly speeds up training in practice (Fig 1a)
- Distribution after connected layer is much more stable with BN, reducing the “**internal covariate shift**”, i.e. the change in the distribution of network activations due to the change in network parameters during training

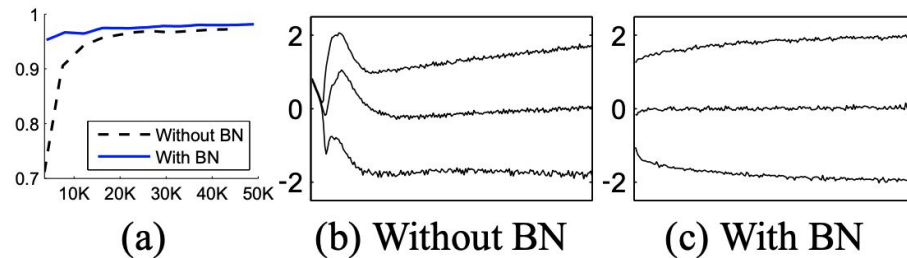
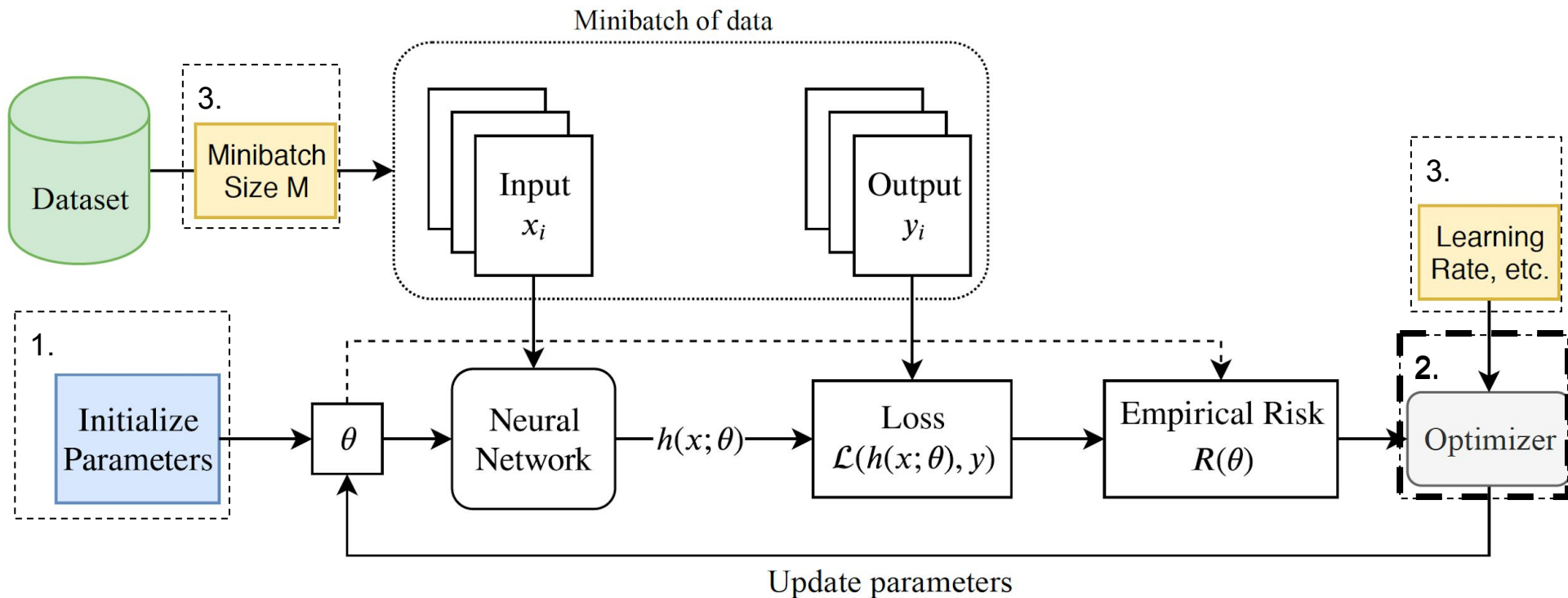


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as {15, 50, 85}th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

# Neural Network Training Loop



# Optimization

Optimizer

- **Optimization:** (informal) Minimize (or maximize) some quantity.
- Applications:
  - Engineering: Minimize fuel consumption of an automobile
  - Economics: Maximize returns on an investment
  - Supply Chain Logistics: Minimize time taken to fulfill an order
  - Life: Maximize happiness

# Optimization: Formal definition

- Given a training set:  $\{(x_1, y_1), \dots, (x_n, y_n)\}$
- Prediction function:  $h(x; \theta)$
- Define a loss function:  $\mathcal{L}(h(x; \theta), y)$
- Find the parameters:  $\theta = (\theta_1, \dots, \theta_k)$

which minimizes the **empirical risk**  $R(\theta)$ :

$$\min_{\theta} R(\theta) = \min_{\theta} \frac{1}{n} \sum_i^n \mathcal{L}(h(x_i; \theta), y_i)$$

# Optimization: Formal definition

- Empirical risk  $R(\theta)$ :

$$\min_{\theta} R(\theta) = \min_{\theta} \frac{1}{n} \sum_i^n \mathcal{L}(h(x_i; \theta), y_i)$$

- The optimum satisfies:  $\nabla R(\theta^*) = 0$

- Where  $\nabla R(\theta) = \left( \frac{\partial R}{\partial \theta_1}, \frac{\partial R}{\partial \theta_2}, \dots, \frac{\partial R}{\partial \theta_k} \right)$

- Sometimes the equation has closed-form solution (e.g. linear regression)



# Optimization: Batch Gradient Descent

Optimizer

## Batch Gradient Descent:

- Initialize the parameters randomly
- For each iteration, do until convergence:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla R(\theta^{(k)})$$

$\eta \in \mathbb{R}^+$

Learning rate (a small step)

# Gradient Descent

Optimizer

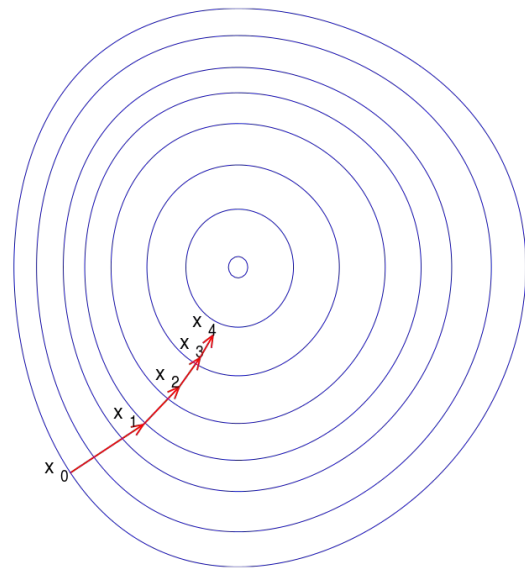
Geometric interpretation:

- Gradient is perpendicular to the tangent of the level set curve
- Given the current point, negative gradient direction decreases the function fastest

Alternative interpretation:

- Minimizing the first-order Taylor approx of  $f$  keep the new point close to the current point

$$f(x^t) + \nabla f(x^t)^T (x - x^t) + \frac{1}{2\eta} \|x - x^t\|_2^2$$



Source: Wikipedia

# Stochastic Gradient Descent

Optimizer

- Initialize the parameters randomly
- For each iteration, do until convergence:
  - Randomly select a training sample  $\mathbf{z}$  (or a small subset of the training samples)
  - Conduct gradient descent:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla f_i(\theta^{(k)})$$

- **Intuition:** A noisy approximation of the gradient of the whole dataset
- **Pro:** each update requires a small amount of training data, good for training algorithms for a large-scale dataset

- **Tips**

- Subsample *without* replacement so that you visit each point on each pass through the dataset ("epoch")
- Divide the log-likelihood estimate by the size of mini-batches, making learning rate invariant to the mini-batch size.

# Gradient Descent with Momentum

Optimizer

- Initialize the parameters randomly
- For each iteration, do until convergence:

- Update the momentum

$$\delta^{(k+1)} = -\eta \nabla R(\theta^{(k)}) + \alpha \delta^{(k)}$$

- Conduct gradient descent:

$$\theta^{(k+1)} = \theta^{(k)} + \delta^{(k+1)}$$

- **Pro:** “accelerate” learning by accumulating some “velocity/momentum” using the past gradients

# Nesterov Accelerated Gradient

- Initialize the parameters randomly
- For each iteration, do until convergence:

- Update the momentum

$$\delta^{(k+1)} = -\eta \nabla R(\theta^{(k)} + \alpha \delta^{(k)}) + \alpha \delta^{(k)}$$

- Conduct gradient descent:

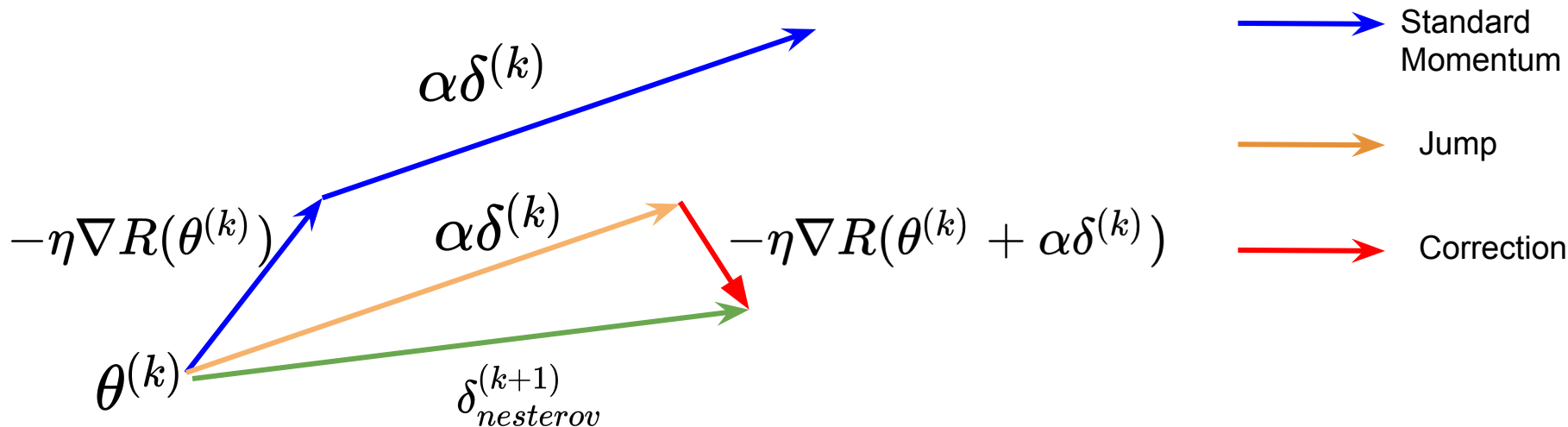
$$\theta^{(k+1)} = \theta^{(k)} + \delta^{(k+1)}$$

- **Pro:** Look into the future to see how much momentum is required

# Nesterov Accelerated Gradient

Optimizer

- **First** make a big jump in the direction of the previous accumulated gradient
- **Then** measure the gradient where you end up and make a correction



# Learning Rate Schedulers

Optimizer

What if we want to be able to have a **per-parameter learning rate**?

- Certain parameter may be more sensitive (i.e. have higher curvature)

# Learning Rate Schedulers: Adagrad

Optimizer

- Initialize the parameters randomly
- For each iteration, do until convergence:
  - Conduct gradient descent on i-th parameter:

$$\theta_{k+1,i} = \theta_{k,i} - \frac{\eta}{\sqrt{G_{k,i} + \epsilon}} \cdot \nabla R(\theta_{k,i})$$

$$G_{k,i} = G_{k-1,i} + (\nabla R(\theta_{k,i}))^2$$

**Intuition:** It increases the learning rate for more sparse features and decreases the learning rate for less sparse ones, according to the history of the gradient



# Learning Rate Schedulers: RMSprop/Adadelta

Optimizer

- Initialize the parameters randomly
- For each iteration, do until convergence:
  - Conduct gradient descent on i-th parameter:

$$\theta_{k+1,i} = \theta_{k,i} - \frac{\eta}{\sqrt{G_{k,i} + \epsilon}} \cdot \nabla R(\theta_{k,i})$$

$$G_{k,i} = \gamma G_{k-1,i} + (1 - \gamma) (\nabla R(\theta_{k,i}))^2$$

**Intuition:** Unlike Adagrad, the denominator places a significant weight on the most recent gradient. This also helps avoid decreasing learning rate too much.

# Learning Rate Schedulers: Adam



Optimizer

- Initialize the parameters randomly
- For each iteration, do until convergence:
  - Conduct gradient descent on i-th parameter:

$$\theta^{(k+1),i} = \theta^{(k,i)} + \frac{\eta \hat{m}^{(k,i)}}{\sqrt{\hat{G}^{(k,i)} + \epsilon}} \cdot \nabla R(\theta^{(k,i)})$$

$$G^{(k,i)} = \gamma G^{(k-1,i)} + (1 - \gamma)(\nabla R(\theta^{(k,i)}))^2$$

$$m^{(k,i)} = \alpha m^{(k-1,i)} + (1 - \alpha) \nabla R(\theta^{(k,i)})$$

$$\hat{m}^{(k,i)} = \frac{m^{(k,i)}}{1 - \alpha^t}$$

$$\hat{G}^{(k,i)} = \frac{G^{(k,i)}}{1 - \gamma^t}$$

Bias-corrected forms of  
 $m^{(k,i)}$ ,  $G^{(k,i)}$

TITLE

Adam: A method for stochastic optimization

D Kingma, J Ba

International Conference on Learning Representation

[Paper Link](#)

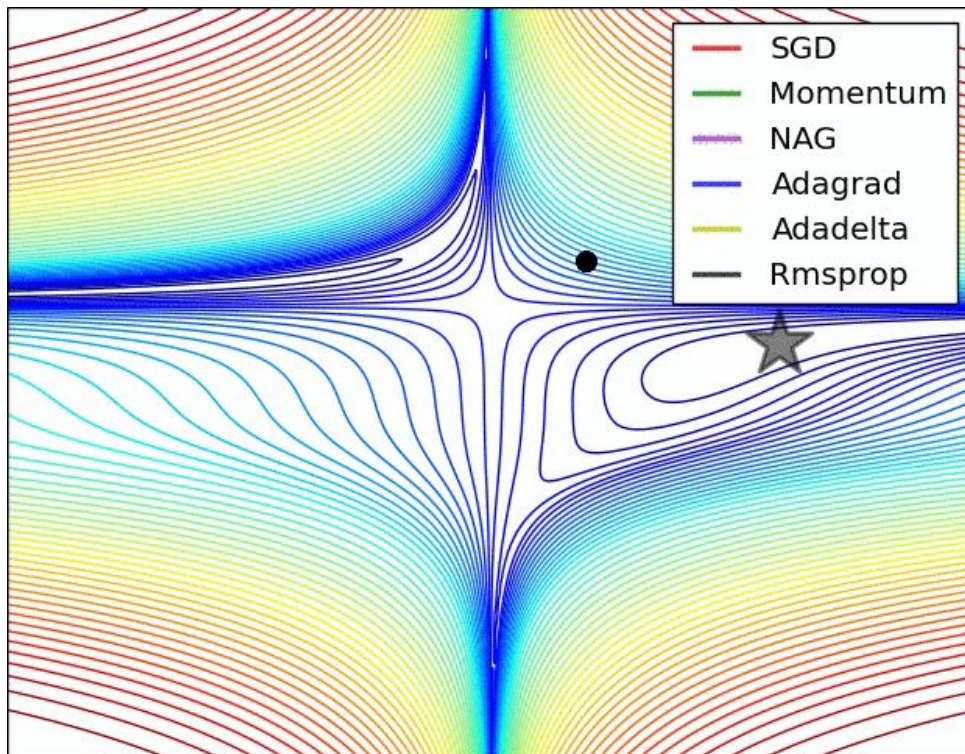
CITED BY

YEAR

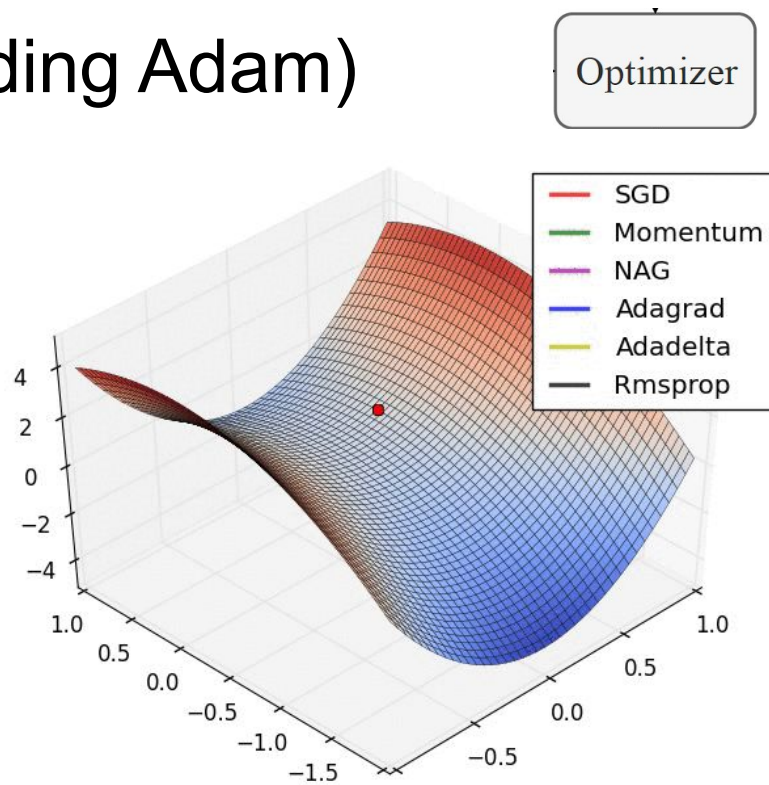
36831

2015

# Optimizers Comparison (excluding Adam)



SGD optimization on loss surface contours



SGD optimization on loss surface contours

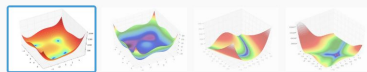
# Interactive Demo: Optimizers

Optimizer

In this visualization, you can compare optimizers applied to different cost functions and initialization. For a given cost landscape (1) and initialization (2), you can choose optimizers, their learning rate and decay (3). Then, press the play button to see the optimization process (4). There's no explicit model, but you can assume that finding the cost function's minimum is equivalent to finding the best model for your task.

## 1. Choose a cost landscape

Select an artificial landscape  $\mathcal{J}(w_1, w_2)$ .



## 2. Choose initial parameters

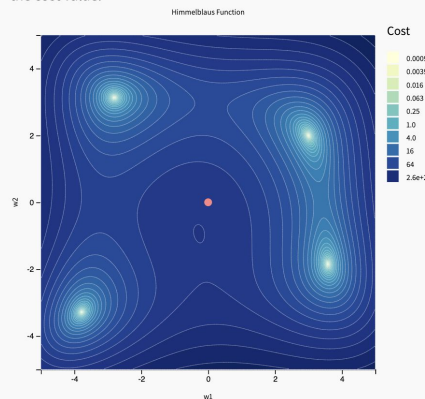
On the cost landscape graph, drag the red dot to choose initial parameter values and thus the initial value of the cost.

## 3. Choose an optimizer

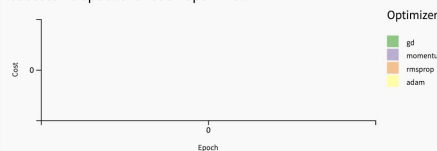
Select the optimizer(s) and hyperparameters.

Optimizer	Learning Rate	Learning Rate Decay
<input checked="" type="checkbox"/> Gradient Descent	<input type="text" value="0.001"/>	<input type="text" value="0"/>
<input checked="" type="checkbox"/> Momentum	<input type="text" value="0.001"/>	<input type="text" value="0"/>
<input checked="" type="checkbox"/> RMSprop	<input type="text" value="0.001"/>	<input type="text" value="0"/>
<input checked="" type="checkbox"/> Adam	<input type="text" value="0.001"/>	<input type="text" value="0"/>

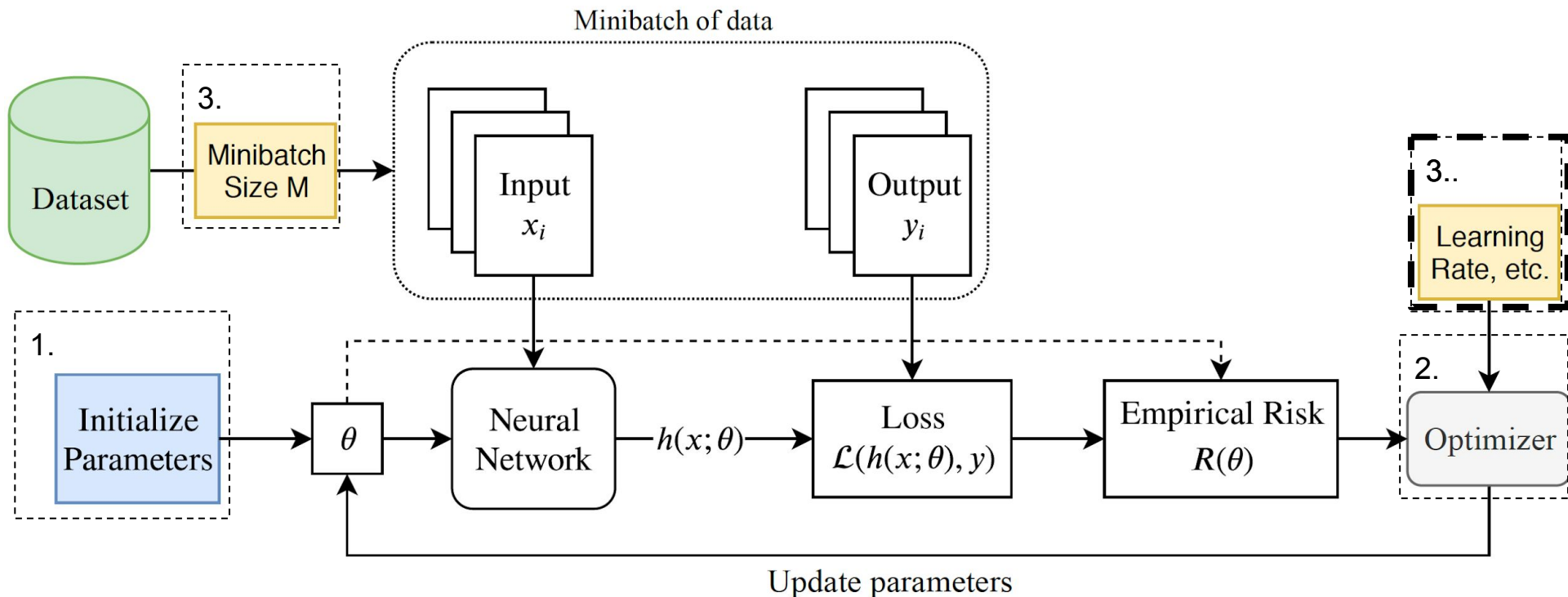
This 2D plot describes the cost function's value for different values of the two parameters ( $w_1, w_2$ ). The lighter the color, the smaller the cost value.



The graph below shows how the value of the cost changes through successive epochs for each optimizer.



# Neural Network Training Loop



# Learning Rate

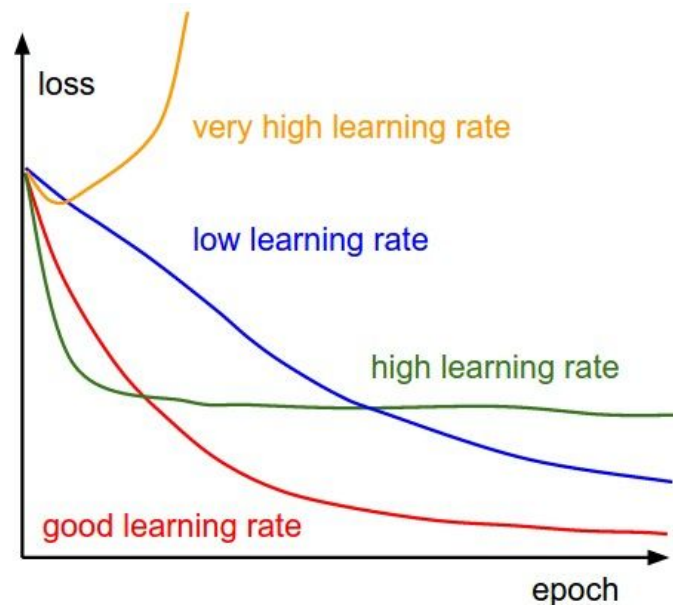
Learning  
Rate, etc.

Ideal Learning Rate should be:

- Should not be too big (objective will blow up )
- Should not be too small (takes longer to converge)

Convergence criteria:

- Change in objective function is close to zero
- Gradient norm is close to zero
- Validation error starts to increase (early-stopping)



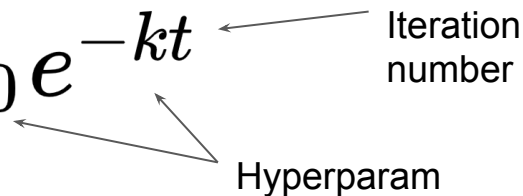
Idealized cartoon depiction of different learning rates.

Image Credit: Andrej Karpathy

# Learning Rate: Decay Schedule

Anneal (decay) learning rate over time so the parameters can settle into a local minimum. Typical decay strategies:

1. **Step Decay:** reduce by factor every few epochs (e.g. a half every 5 epochs, or by 0.1 every 20 epochs), or when validation error stops improving
2. **Exponential Decay:** Set learning rate according to the equation

$$\eta(t) = \eta_0 e^{-kt}$$


Iteration number

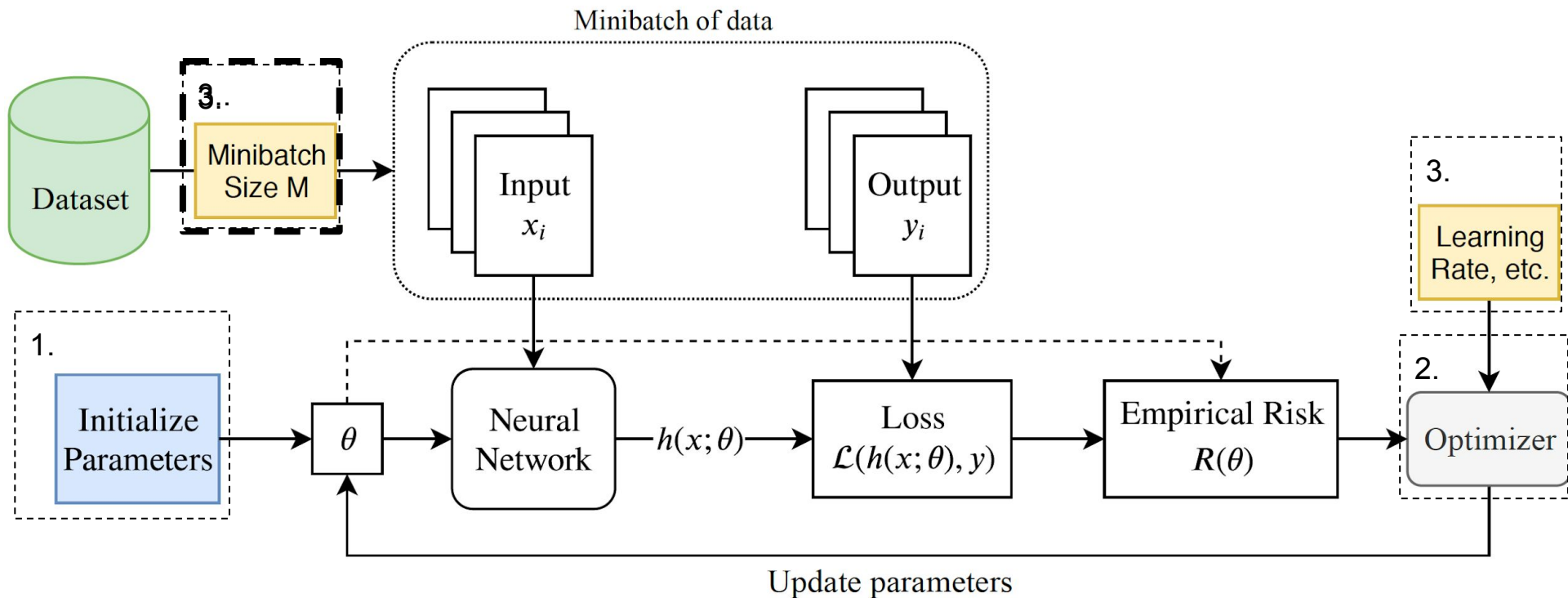
Hyperparam

3. **1/t decay:**

$$\eta(t) = \frac{\eta_0}{1+kt}$$



# Neural Network Training Loop





# Batch Size

Minibatch  
Size M

**Batch Size:** the number of training data points for computing the empirical risk at each iteration.

- Typical small batches are powers of 2: 32, 64, 128, 256, 512,
- Large batches are in the thousands

**Large Batch Size** has:

- **Fewer frequency of updates**
- More **accurate** gradient
- More **parallelization** efficiency / accelerates wallclock training
- **May hurt generalization**, perhaps by causing the algorithm to find poorer local optima/plateau.

# Batch Size

Minibatch  
Size M

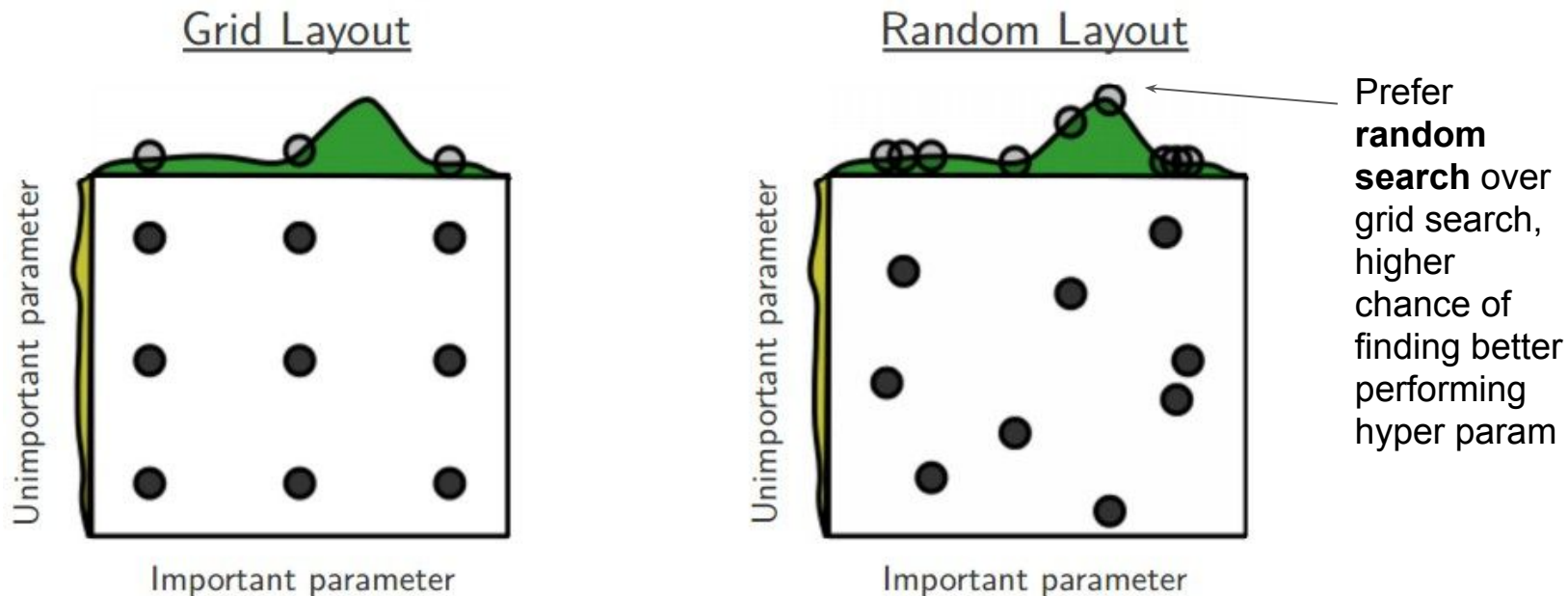
Related papers on batch size:

- [Goyal et al., Accurate, large minibatch SGD](#)
  - Proposes to increase the learning rate by of the minibatch size
- [Hoffer et al., Train longer generalize better](#)
  - Proposes to increase the learning rate by **square root** of the minibatch size
- [Smith et al., Don't decay the learning rate, increase the batch size](#)
  - Increasing batch size reduce noise, while maintaining same step size

# Hyperparameter Tuning

Minibatch  
Size  $M$

Several approaches for tuning multiple hyperparameters together:



# Hyperparameter Tuning

Search hyperparameter on **log scale**:

- `learning_rate = 10 ** uniform(-6, 1)`
  - Learning rate and regularization strength have multiplicative effects on the training dynamics
- Start from coarse ranges then narrow down, or expand range if near the boundary of range

One validation fold vs cross-validation:

- Simplifies code base to just use one (sizeable) validation set vs doing cross validation

# Jupyter/Colab Demo in PyTorch

See Colab notebook

# References

- Notes and tutorials from other courses:
  - [ECE521 \(Winter 2017\)](#) tutorial on [Training neural network](#)
  - [Stanford's CS231n notes](#) on [Stochastic Gradient Descent](#), [Setting up data and loss](#), and [Training neural networks](#)
  - [Deeplearning.ai's](#) interactive notes on [Initialization](#) and [Parameter optimization in neural networks](#)
  - Jimmy Ba's Talk for [Optimization in Deep Learning](#) at [Deep Learning Summer School 2019](#)
- Academic/white papers:
  - [SGD](#) tips and tricks from Leon Bottou
  - [Efficient BackProp](#) from Yann LeCun
  - [Practical Recommendations for Gradient-Based Training of Deep Architectures](#) from Yoshua Bengio