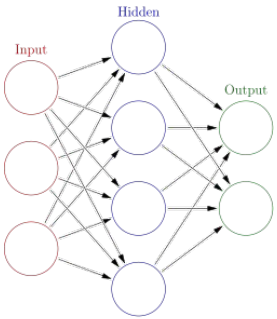
CSC413 Tutorial: Optimization for Machine Learning

Neural Network



"*How to train your neural network*"

Rex Ma[1]

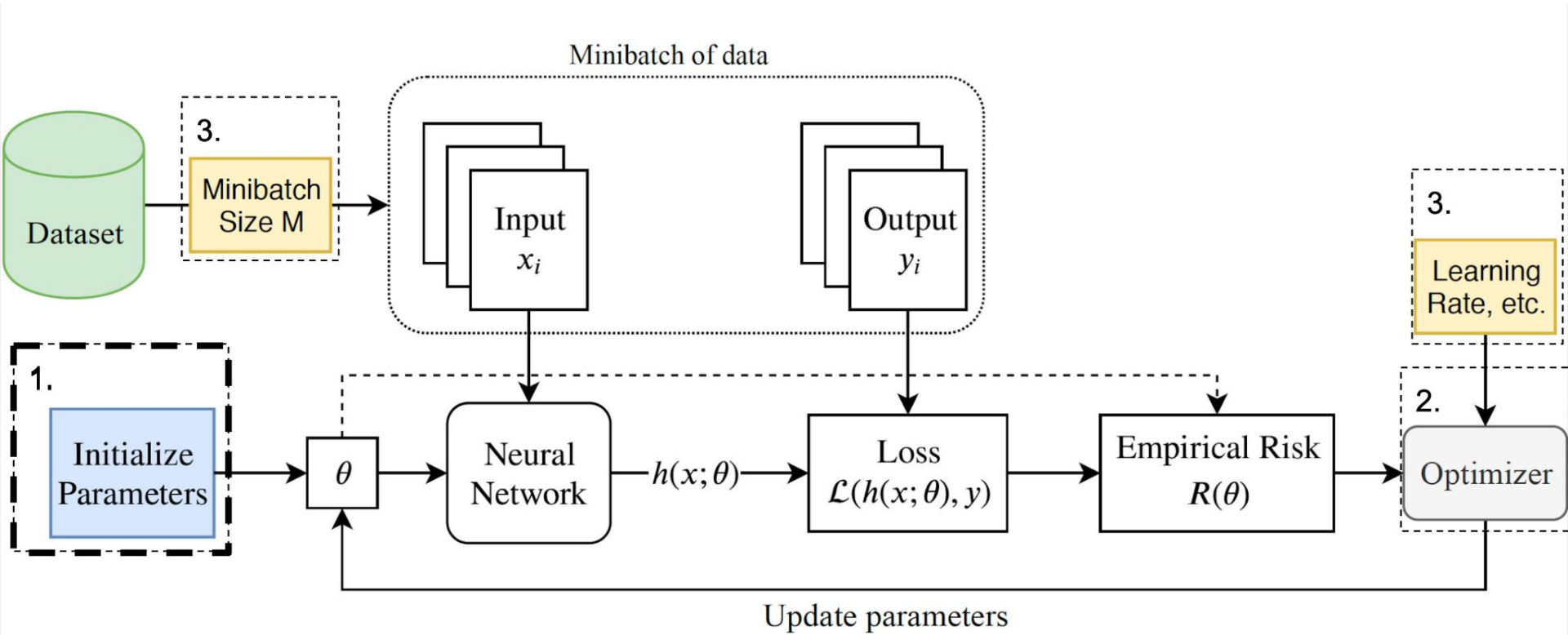
January 24, 2023

1 Based on tutorials/slides by Harris Chan, Ladislav Rampasek, Jake Snell, Kevin Swersky, Shenlong Wang, Shervin Mehryar & 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
- Jupyter/Colab Demo in PyTorch

Neural Network Training Loop



Initialization of Parameters

Initialize Parameters

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

Idea 1: Constant initialization

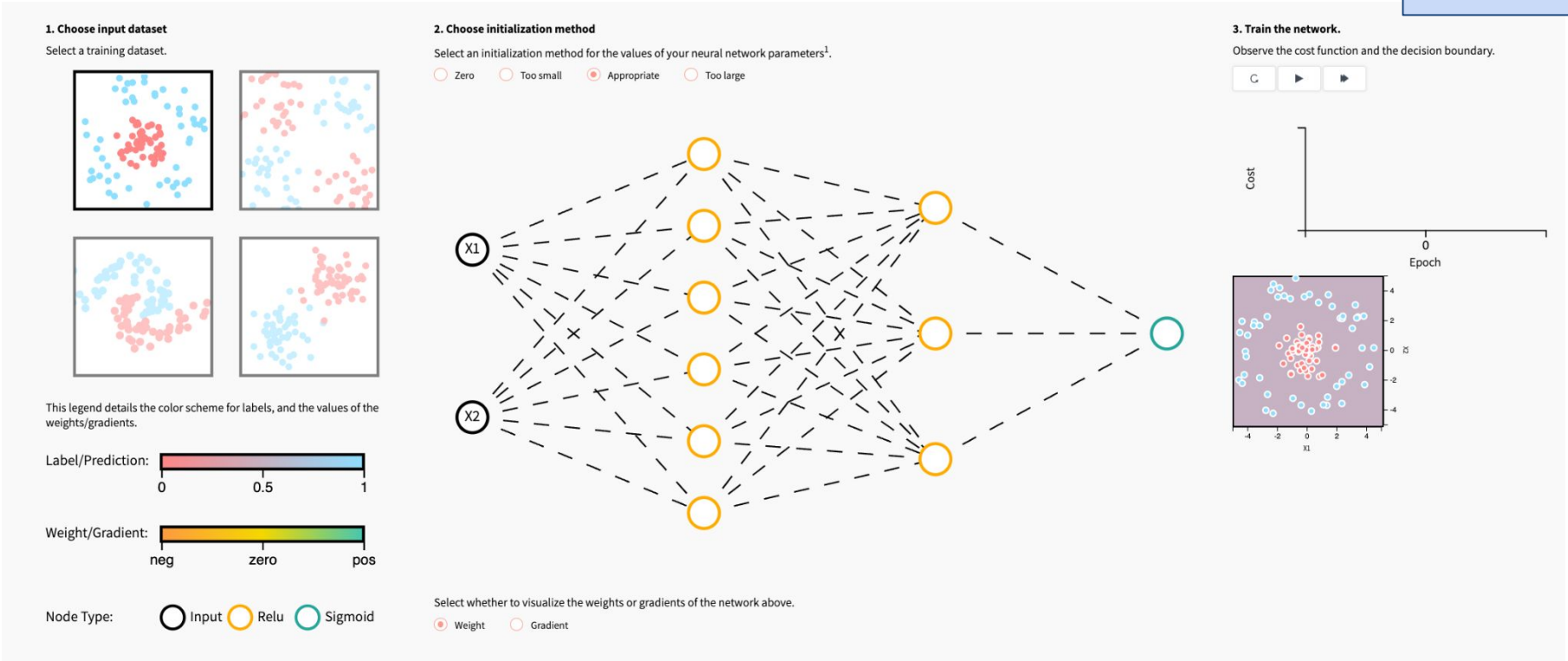
- *Result*: For fully connected layers: identical gradients, identical neurons. Bad!

Idea 2: Random weights, to break symmetry

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

Interactive Demo: Initialization

Initialize Parameters



Source: Initializing neural networks. https://www.deeplearning.ai/ai-notes/initialization/

Initialization: Calibrate the variance

Initialize Parameters

Two popular initialization schemes:

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

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

$$b^{[l]} = 0$$

or

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

$$b^{[l]} = 0$$

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

Initialize Parameters

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

1. **Variance** of activation **~ constant** across every layer

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

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

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

Initialization: Xavier Initialization Proof

Initialize Parameters

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

$$z_k^{(l)} = \sum_i^{n^{l-1}} w_{ki}^{(l)} a_i^{(l-1)} \qquad a_k^{(l)} = f(z_k^{(l)})$$

Layer

Activations

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

$$\begin{aligned} \mathrm{Var}(a_k^{(l)}) &= \mathrm{Var}(z_k^{(l)}) \\ &= \mathrm{Var}(\sum_i^{n^{l-1}} w_{ki}^{(l)} a_i^{(l-1)}) \\ &= \sum_i^{n^{l-1}} \mathrm{Var}(w_{ki}^{(l)} a_i^{(l-1)}) \end{aligned}$$

In linear region for $f = tanh$

Assume product terms are independent, bring summation outside

Initialization: Xavier Initialization Proof

Initialize Parameters

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

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

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

Assume mean of weights = 0

Assume mean of inputs = 0

Note: Not true for ReLU

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

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

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

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

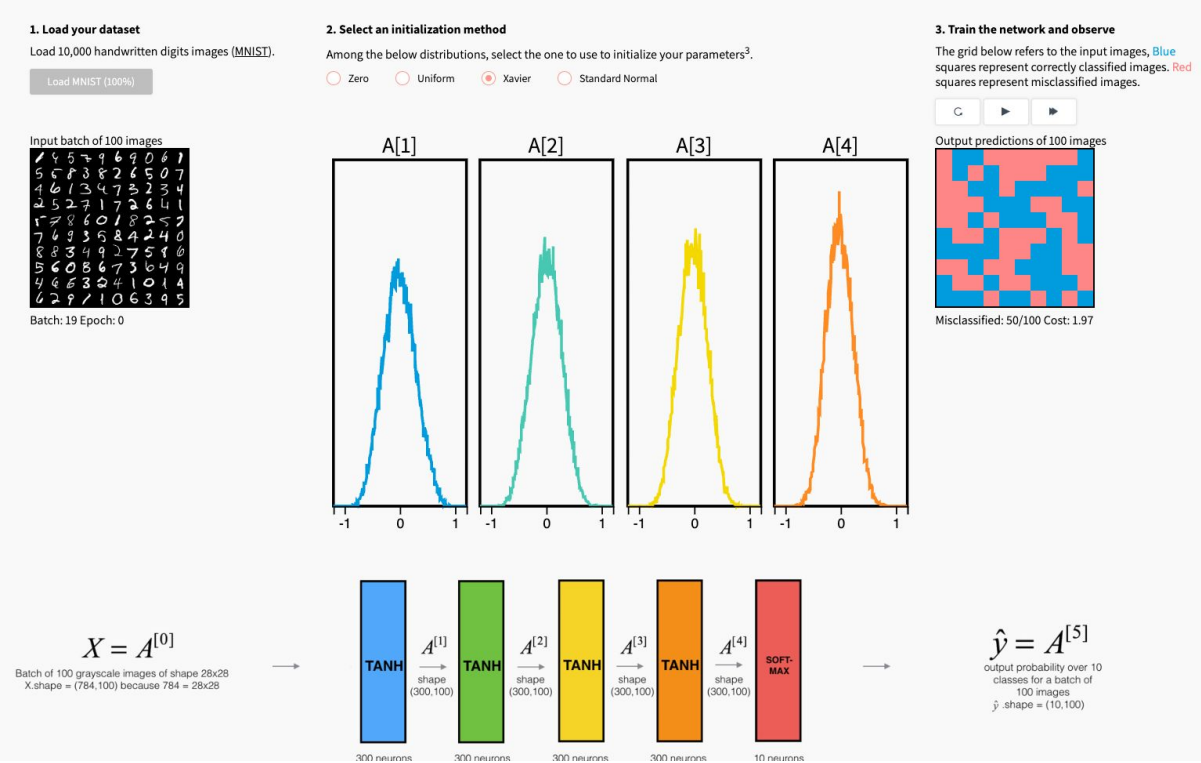
Weights are iid
Inputs are iid

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

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

Interactive Demo: Initialization Schemes

Initialize Parameters



Source: Initializing neural networks. https://www.deeplearning.ai/ai-notes/initialization/

Batch Normalization Layer

Initialize Parameters

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: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

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

Ioffe & Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Batch Normalization Layer

Initialize Parameters

- 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*

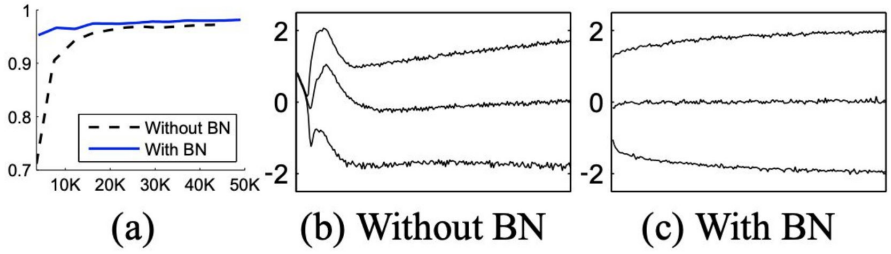
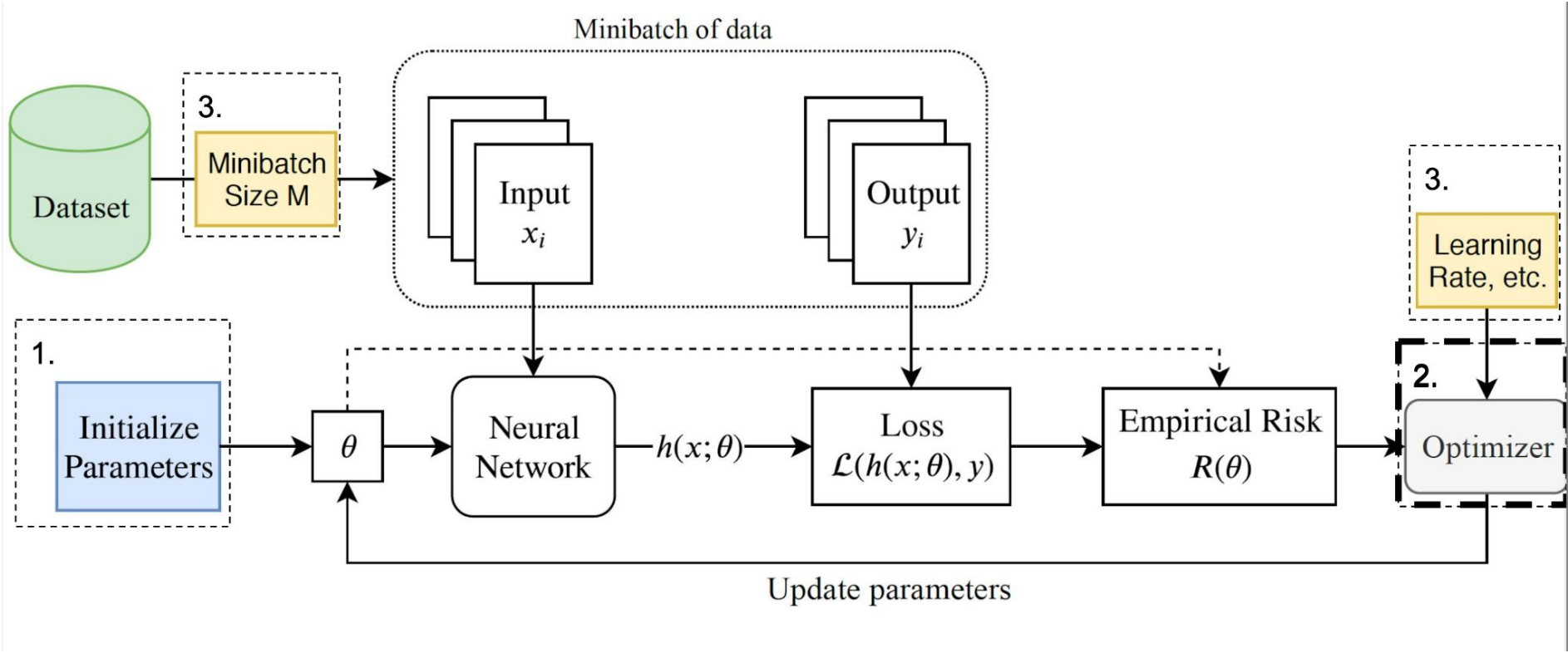


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of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

Ioffe & Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing 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

Optimizer

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

  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

Optimizer

- **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: $\boxed{\nabla R(\theta^*) = 0}$

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

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

Gradient Descent

Optimizer

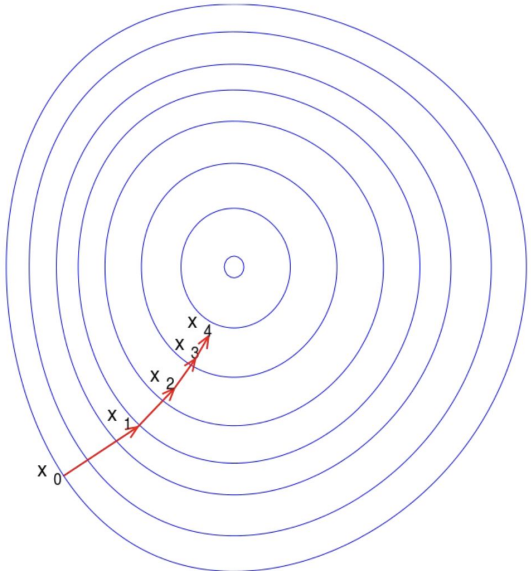
Geometric interpretation:

- 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 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

Optimization: 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)

Stochastic Gradient Descent

Optimizer

- Initialize the parameters randomly
- For each iteration, do until convergence:
  - Randomly select a training sample (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

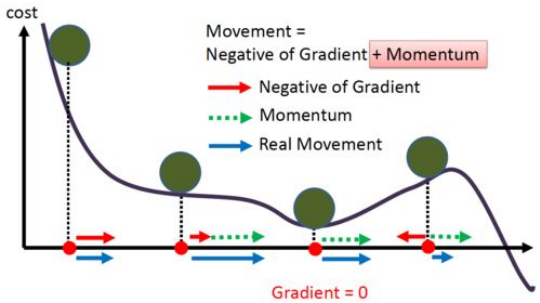
Optimizer

- 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

Optimizer

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

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

  - Conduct gradient descent:

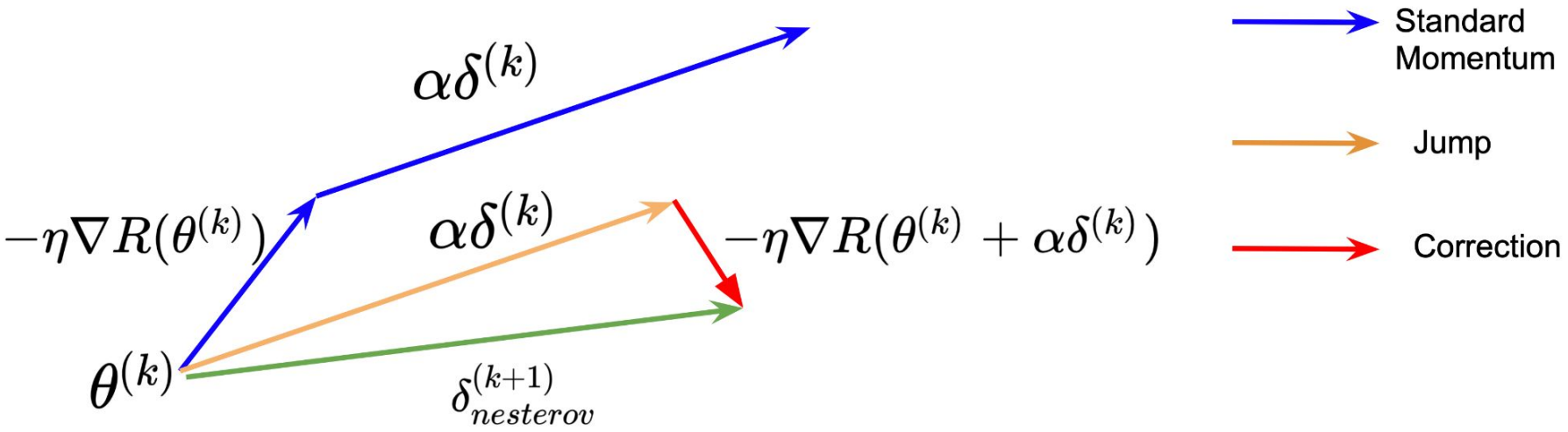
$$\theta^{(k+1)} = \theta^{(k)} + \delta^{(k+1)}$$

- **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

Optimizer

Learning Rate Schedulers: RMSprop/Adadelta

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

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

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

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:

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

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

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

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

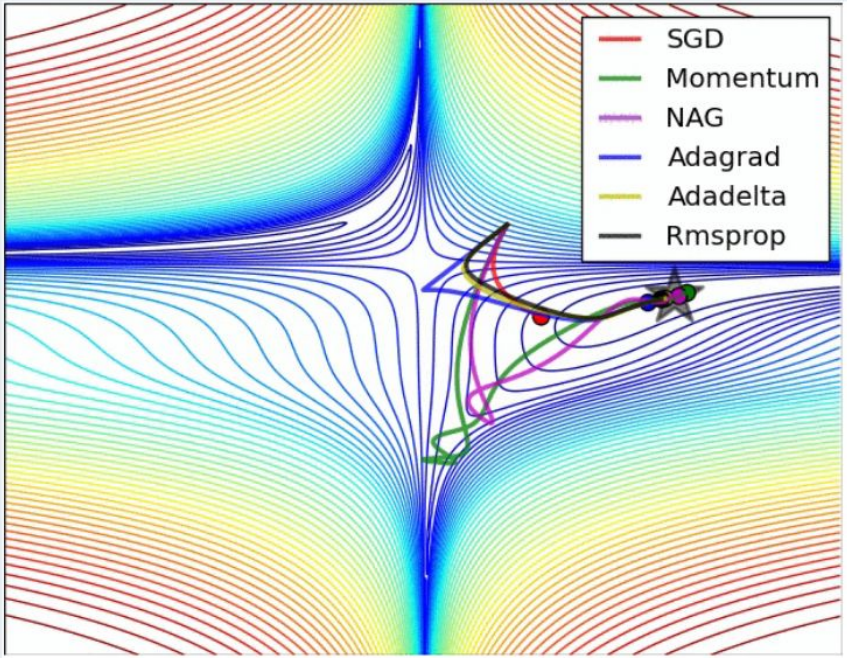
$$\hat{G}^{(k,i)} = \frac{G^{(k,i)}}{1-\gamma^t}$$

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

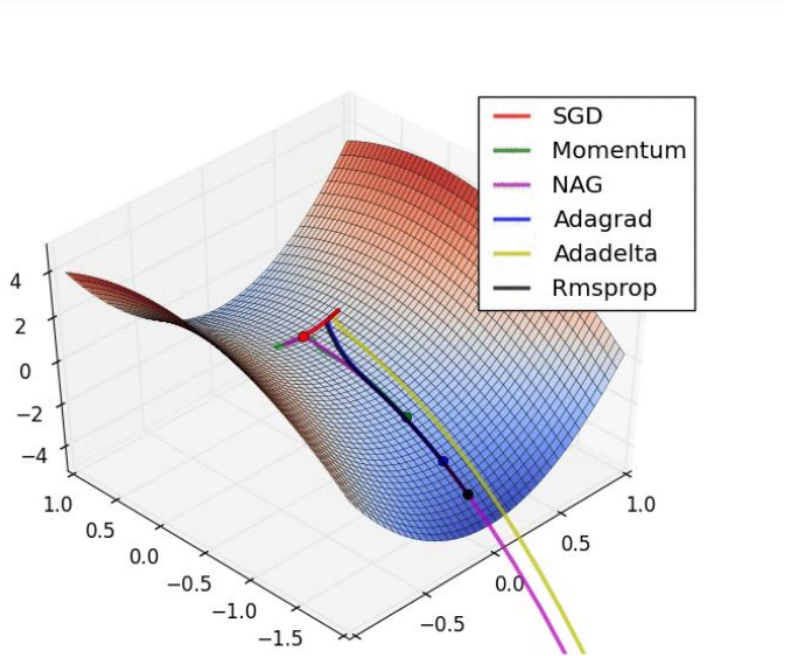
TITLE	CITED BY	YEAR
Adam: A method for stochastic optimization D Kingma, J Ba International Conference on Learning Representations	133579	2015

Optimizer

Optimizers Comparison (excluding Adam)



SGD optimization on loss surface contours

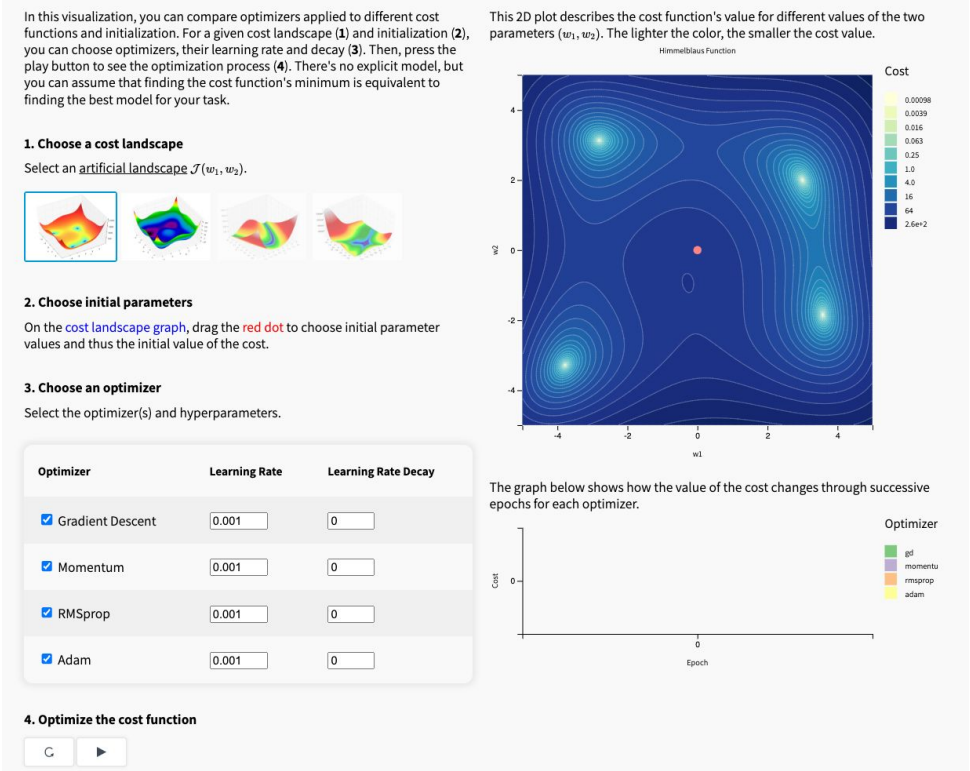


SGD optimization on loss surface contours

Source: Sebastian Ruder, https://ruder.io/optimizing-gradient-descent/

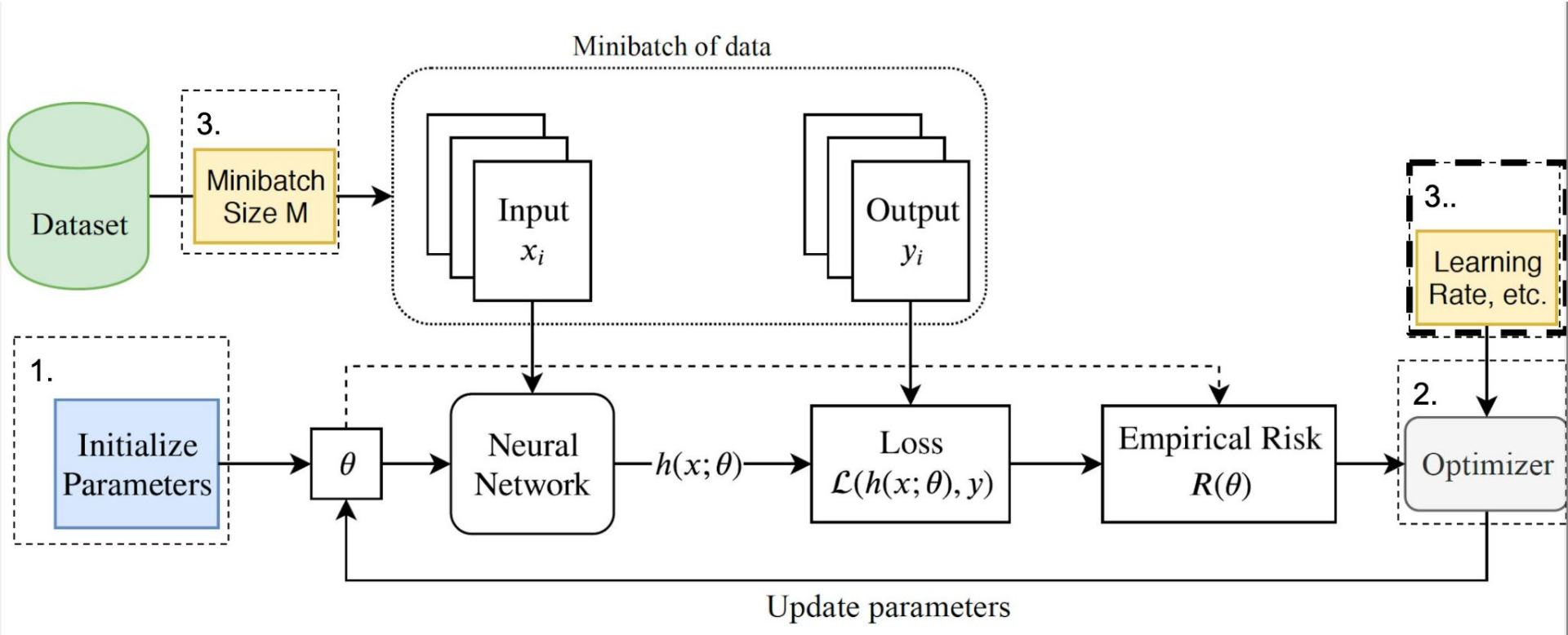
Optimizer

Interactive Demo: Optimizers



Source: Parameter optimization in neural networks: https://www.deeplearning.ai/ai-notes/optimization/

Neural Network Training Loop



Learning Rate

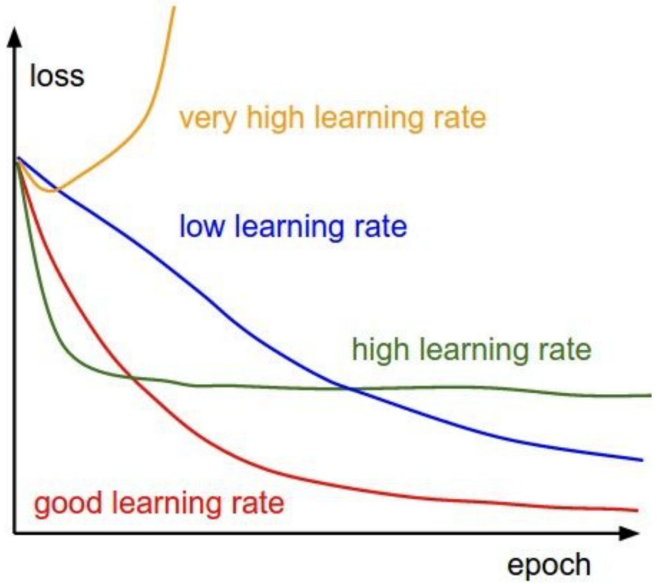
Learning Rate, etc.

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

Learning Rate, etc.

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

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

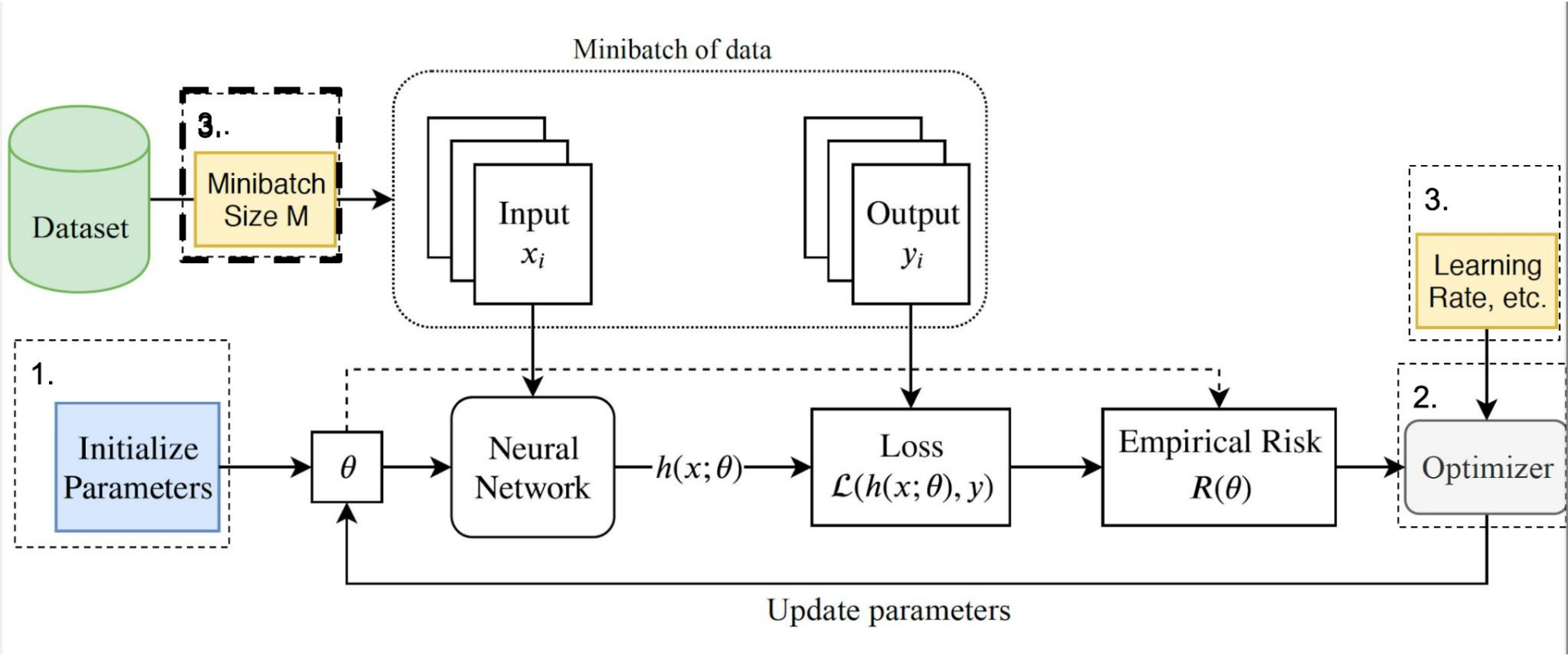
Iteration number

Hyperparam

3. **1/t decay:**

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

Neural Network Training Loop



Batch Size

Minibatch Size, etc.

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, etc.

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, etc.

Several approaches for tuning multiple hyperparameters together:

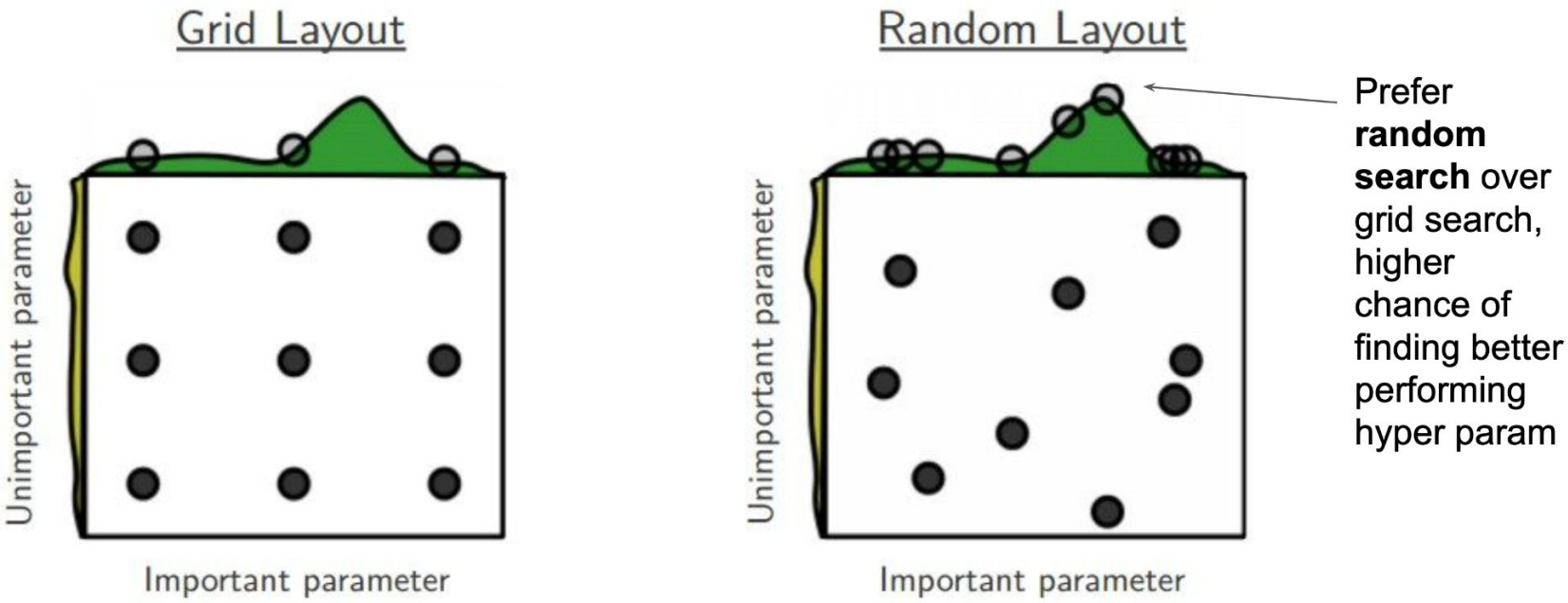


Image source: Random Search for Hyper-Parameter Optimization

Try Weights & Biases

Hyperparameter Tuning

Minibatch Size, etc.

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

Deep Learning Tuning Playbook by Google Brain

- for engineers and researchers (both individuals and teams) interested in **maximizing the performance of deep learning models**

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