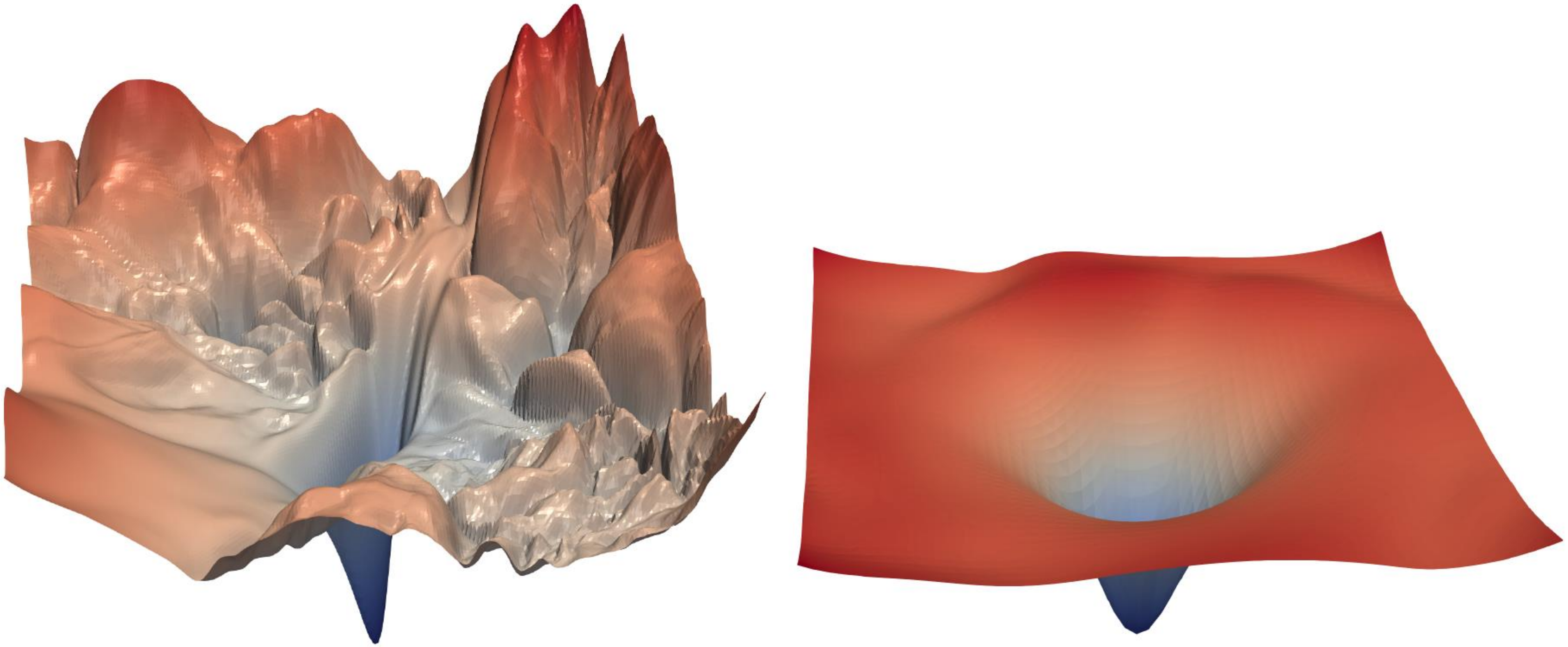
Neural Networks and Deep Learning

Optimisation methods, weight initialization, and model regularization.

Credit to Lluis Gomez

The Optimisation Landscape

Successfully training a neural network implies (1) **defining well-behaved loss landscapes** and (2) using the **right algorithms to tread through them**



Li et al "Visualizing the Loss Landscape of Neural Nets", NIPS 2018

Still Not A Learning algorithm

- We still need to understand
  - **Loss functions**: How to measure our error? This depends on the task we want to solve.
  - **Activation functions**: What kind of neurons should we use?
  - **Architectures**: How to combine neurons together to build meaningful models?

Define well-behaved landscapes

  - **Optimisation**: Is batch gradient descent the best way to use these error derivatives to discover a good set of weights?
  - **Regularisation**: How do we make sure we do not overfit?
  - **Initialisation**: Where do we start our search?

Efficient search

HOW TO TRAIN DEEP NETWORKS

Introduction

Training deep neural networks is a complex and expensive task!

Image source: Yannis Ghazouani, 2016.

Why are Deep Neural Nets hard to train?

- Solving difficult tasks implies large scale datasets and very large models

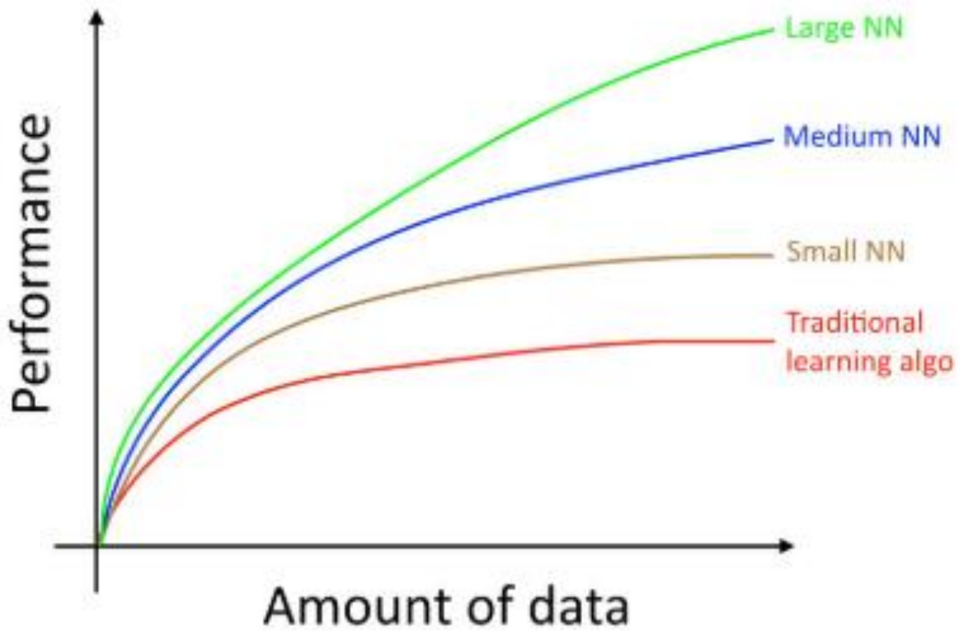


Image source: Ng, A. Machine learning yearning: Technical strategy for ai engineers in the era of deep learning. (2019).

Why are Deep Neural Nets hard to train?

- Solving difficult tasks implies large scale datasets and very large models

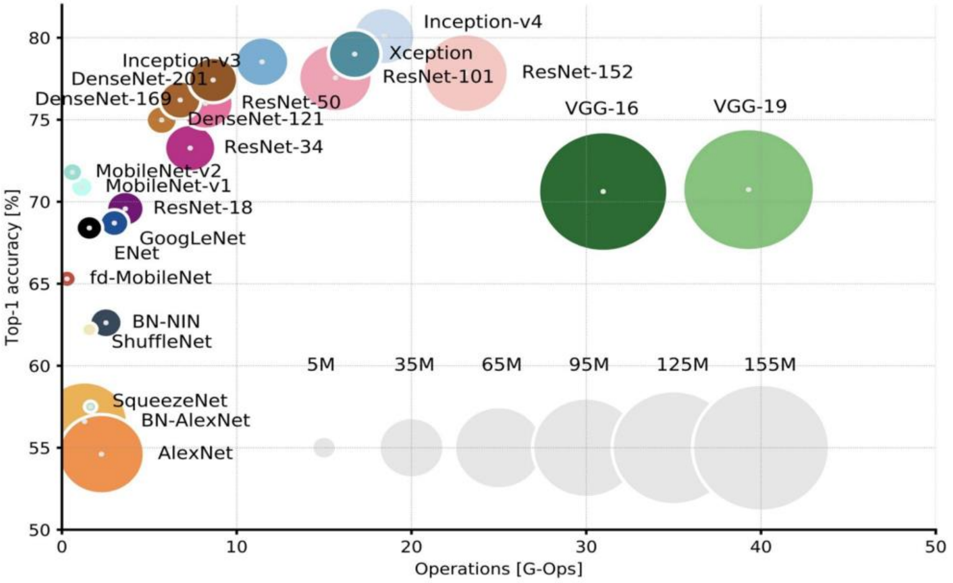


Image source: An analysis of deep neural network models for practical applications. arXiv preprint arXiv:1605.07678. 2017

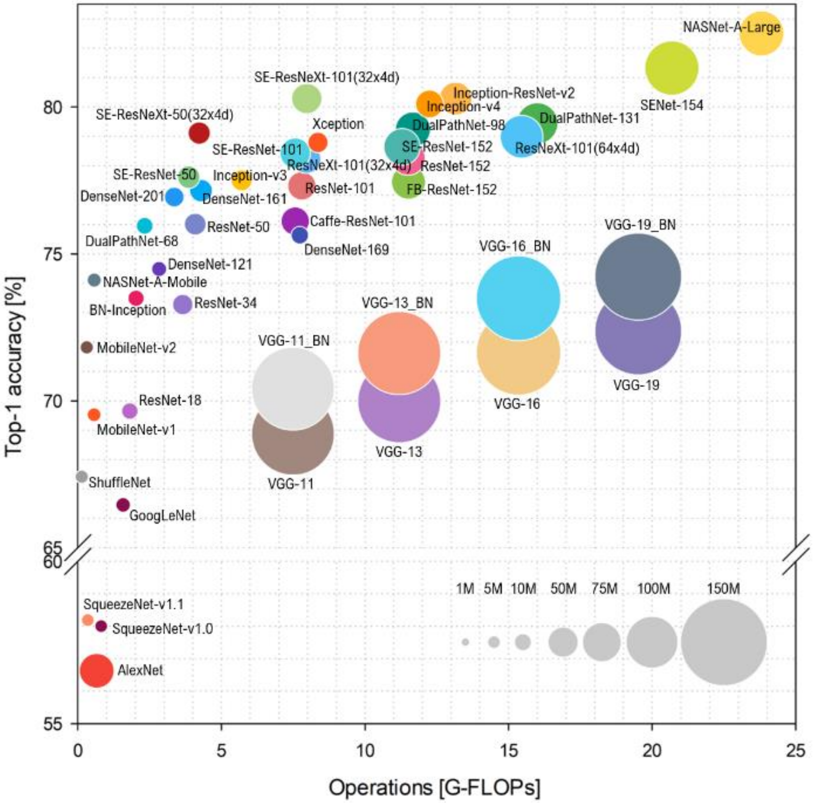
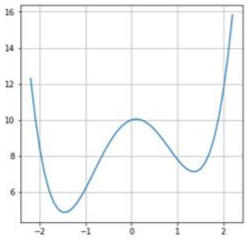


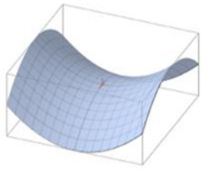
Image source: Benchmark Analysis of Representative Deep Neural Network Architectures. arXiv:1810.00736v2 2018

Why are Deep Neural Nets hard to train?

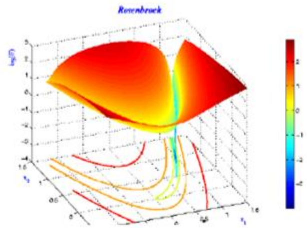
- Empirical risk minimization
- Surrogate loss functions
- Stochastic methods
- Stepsize issue and monotonicity
- Local minima
- Plateaus, saddle points, flat regions
- Vanishing/exploding gradients
- Unstable/inexact gradients
- Local vs. global structure



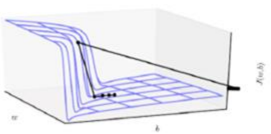
Local minima



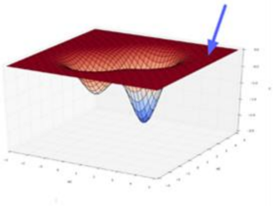
Saddle points



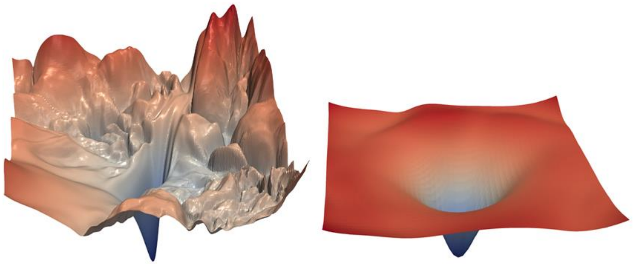
Ravines



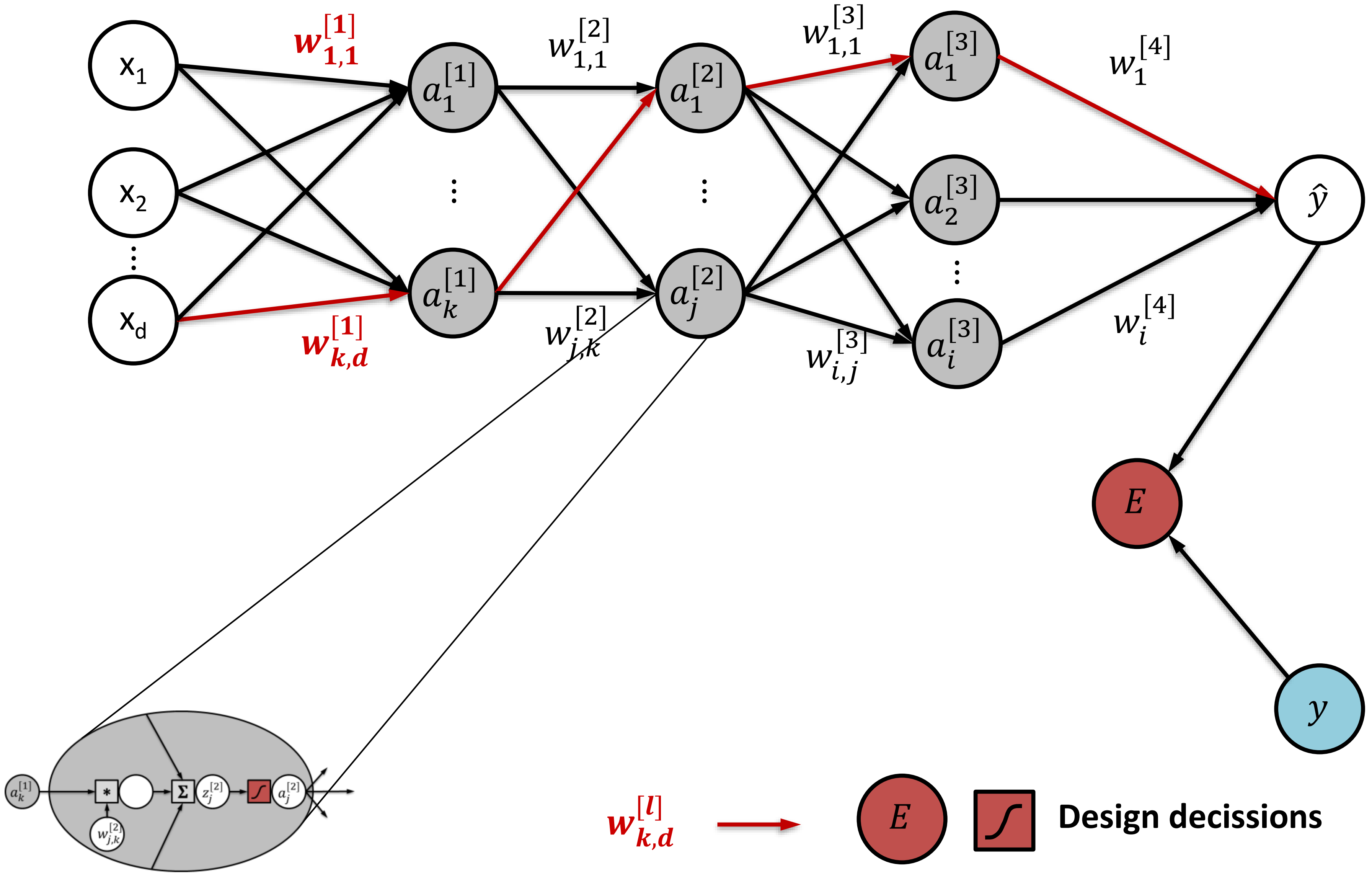
Cliffs



Plateaux



Getting gradient into the loop



Design decissions

WHERE TO START FROM

Gradient problems

- Vanishing gradients
- Exploding gradients

$$z_j = w_j a_{j-1} + b_j$$

$$\frac{\partial z_1}{\partial b_1}\,\frac{\partial a_1}{\partial z_1}\quad\frac{\partial z_2}{\partial a_1}\,\frac{\partial a_2}{\partial z_2}\quad\frac{\partial z_3}{\partial a_2}\frac{\partial a_3}{\partial z_3}\quad\frac{\partial z_4}{\partial a_3}\frac{\partial a_4}{\partial z_4}\quad\frac{\partial L}{\partial a_4}=\frac{\partial L}{\partial b_1}$$

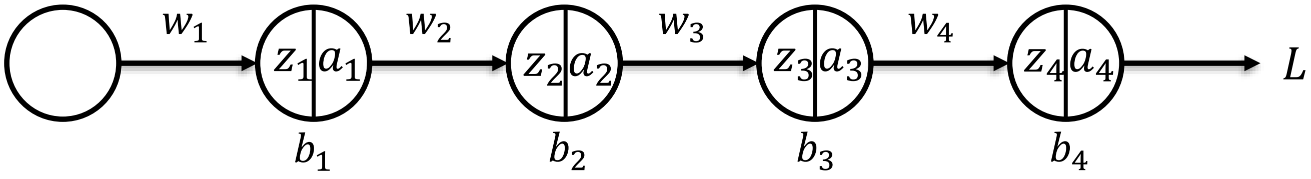


Figure by Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap5.html)

Gradient problems

- Vanishing gradients

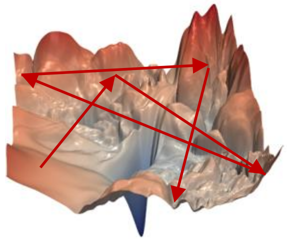
  Small w_i ⟶ small gradients $\cong 0$

  no navigation in the optimization landscape

- Exploding gradients

  Large w_i ⟶ large gradients

  unstable navigation in the OL



$$\sigma'(z_1)\,\boxed{w_2}\;\sigma'(z_2)\,\boxed{w_3}\;\sigma'(z_3)\,\boxed{w_4}\;\sigma'(z_4)\,\frac{\partial L}{\partial \alpha_4} = \frac{\partial L}{\partial b_1}$$

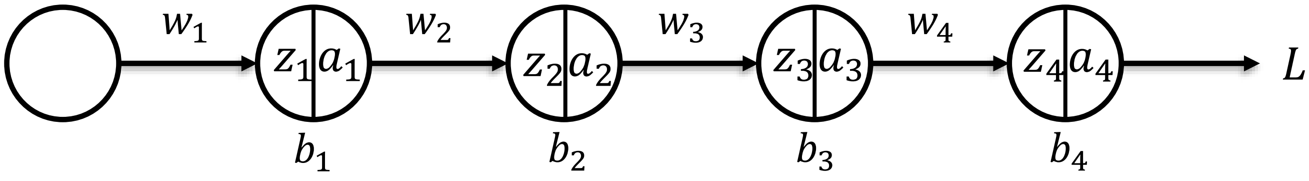


Figure by Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap5.html)

Weights initialization

Issues:

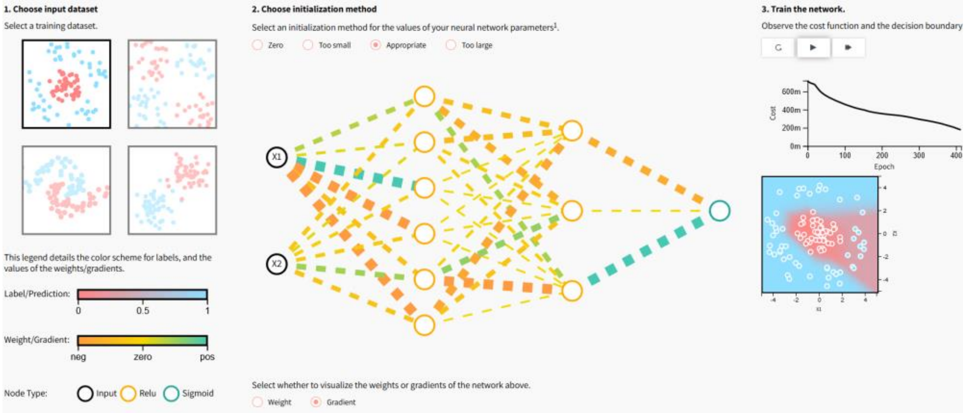
- User must specify some initial point.
- Optimization algorithms are strongly affected by the choice of initialization.
- Initial point can determine whether the algorithm converges or not
- If it does converge, the initial point might determine how quickly and which minima.

Weights initialization

- Zero/constant values:

$$\frac{\partial z_k^{[l]}}{\partial w_{k,d}^{[l]}} = 0 \Rightarrow \text{no weight update} \Rightarrow \text{no learning}$$

‘Active paths’ must exists between input and output



1. Choose input dataset

Select a training dataset.

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

2. Choose initialization method

Select an initialization method for the values of your neural network parameters[1].

○ 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

https://www.deeplearning.ai/ai-notes/initialization/

‘Active path’, gradient update is effective from the output layer up to the input layer

Weights initialization

Intuition:

Makes sure the weights are ‘just right’, keeping the signal in a reasonable range of values through many layers, and heteregoneous.

Symmetry breaking.

It is actually best to initialize each unit to compute a different function from all the other units.

This motivates **random initialization**.

Weights initialization

Symmetry breaking:

Each $w_{k,d}^{[l]}$ derived from a Normal distribution, and $b_k^{[l]} = 0$

```
import numpy as np
W = np.random.randn(D, H) # D inputs, H hidden units
```

```
import numpy as x = torch.randn(512)
for i in range(100)
           w = torch.randn(512,512)
           act = w @ x
           if torch.isnan(act.std()): break
i
>> output: i=28
```

Which is the good range?

Weights initialization

Which is the good range?

$$\mathrm{z} = b + \sum_{i}^{n} w_i a_i$$

z variance increases as the number of inputs increases

Solution: Normalize the variance of each neuron output to 1 (scaling weights by the $\sqrt{n}$ (n: number of input units)

```
import numpy as np
w = np.random.randn(n) * sqrt(1.0/n)
```

This is known as the **Xavier/Glorot** initialization algorithm (Xavier Glorot and Yoshua Bengio 2010).

Weights initialization

Which is the good range?

$$z_1^{[l]} = \mathbf{w}_1^{[l]T} \boldsymbol{a}^{[l-1]} + b_1^{[l]}$$

$$a_1^{[l]} = \alpha\left(z_1^{[l]}\right)$$

$z_1^{[l]}$ is symmetric but not $a_1^{[l]}$ if any ReLU variant is used

Solution: as half of the output is 0, double the output of non zero values

```
import numpy as np
w = np.random.randn(n) * sqrt(2.0/n)
```

This is known as the **He/Kaiming** initialization algorithm
(He et al 2015).

Weights initialization in PyTorch

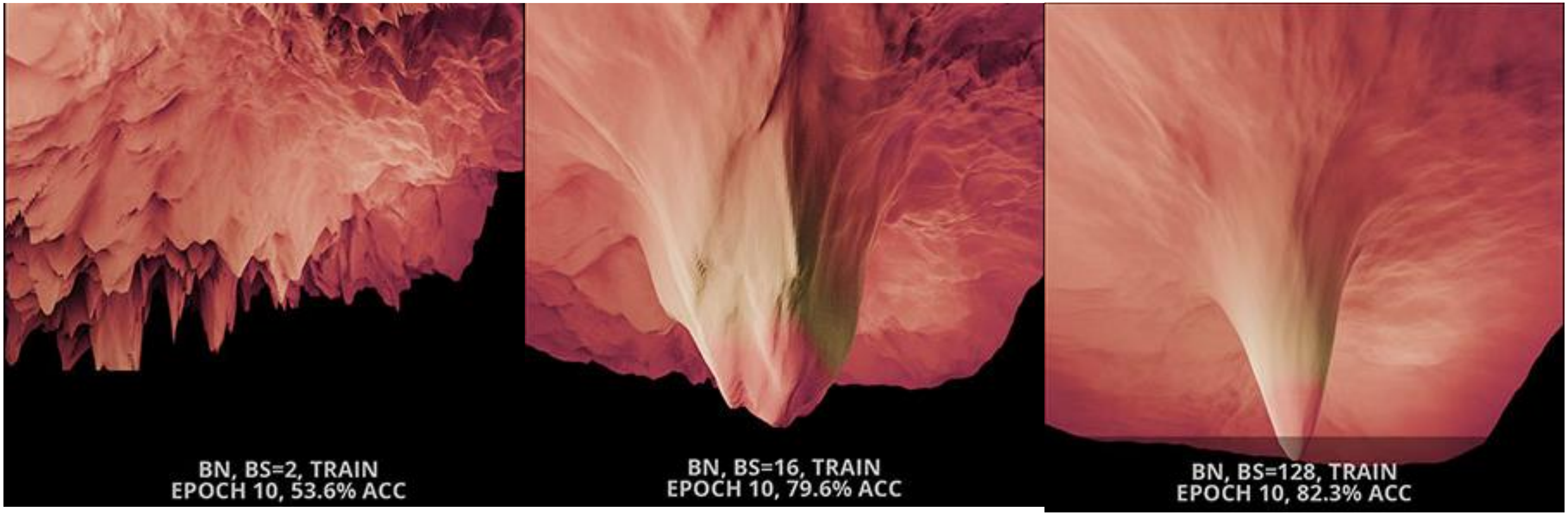
```
torch.nn.init.uniform_(tensor, a=0.0, b=1.0)

torch.nn.init.normal_(tensor, mean=0.0, std=1.0)

torch.nn.init.xavier_uniform_(tensor, gain=1.0)

torch.nn.init.xavier_normal_(tensor, gain=1.0)

torch.nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in',
                             nonlinearity='leaky_relu')

torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in',
                           nonlinearity='leaky_relu')
```

Variations

```
import numpy as np
w = np.random.randn(n) * sqrt(k/(n+m)) #n/m -> input/output units
```

HOW TO UPDATE WEIGHTS
BASIC OPTIMIZATION ALGORITHMS

Loss landscape



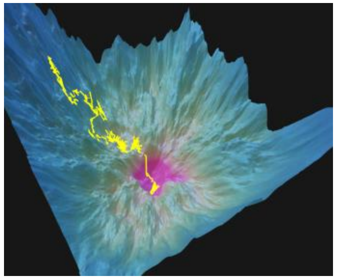
https://losslandscape.com/wp-content/uploads/2019/09/loss-landscape-angle-1.jpg

Review: Stochastic Gradient Descent (SGD)

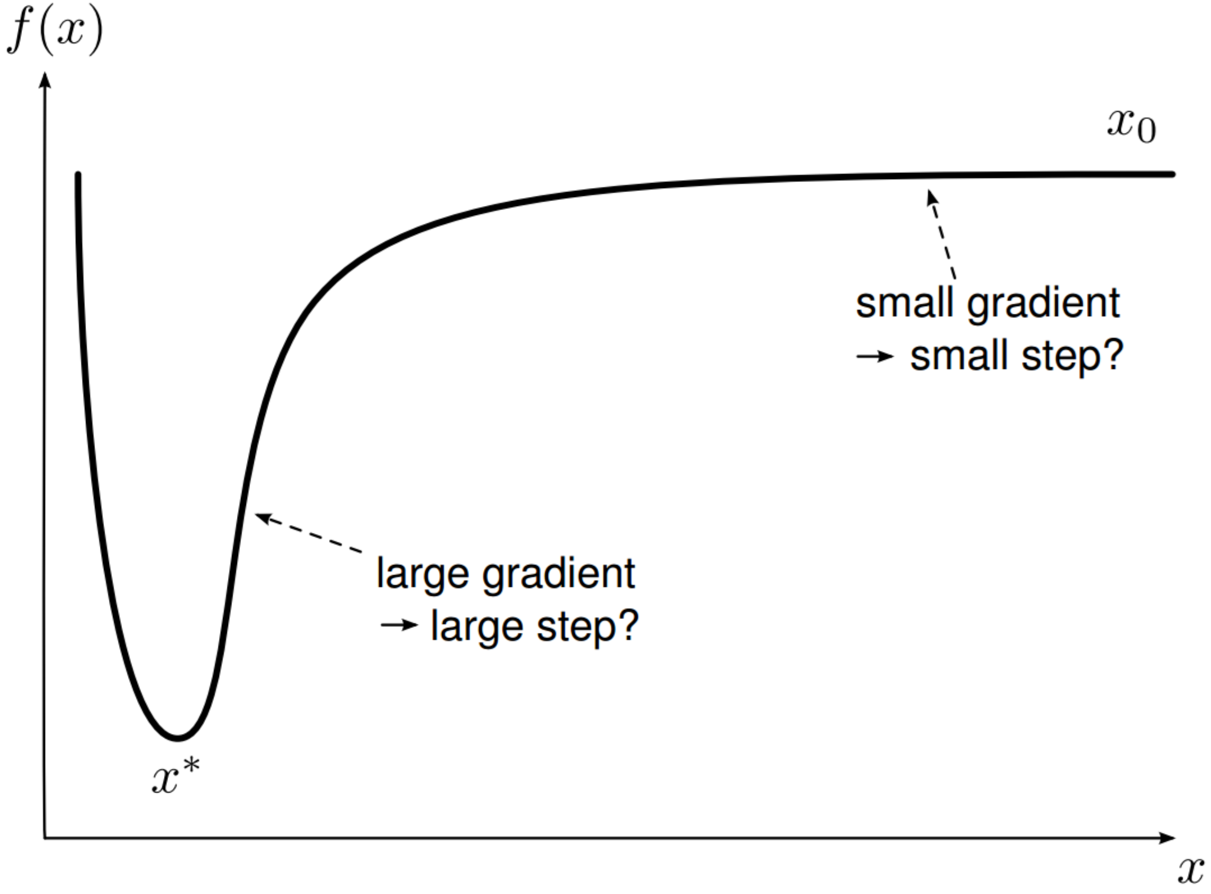
Algorithm: SGD update at training iteration *k*

```
Require: Learning rate $\epsilon_k$
Require: Initial parameters $\mathbf{w}$
                                        $J$
while stopping criterion not met do
    Sample a minibatch of m samples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$
    with corresponding targets $y^{(i)}$.
    Compute gradient estimate:   $J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$

    Apply update:                $\mathbf{w} \leftarrow \mathbf{w} - \epsilon_k \nabla_{\mathbf{w}} J(\mathbf{w})$
end while
```

https://losslandscape.com/explorer



The step size issue



M. Toussaint, "Some notes on gradient descent", FU Berlin 2012.

The step size issue

Ill-conditioned problem: gradient should decrease as learning progress ??

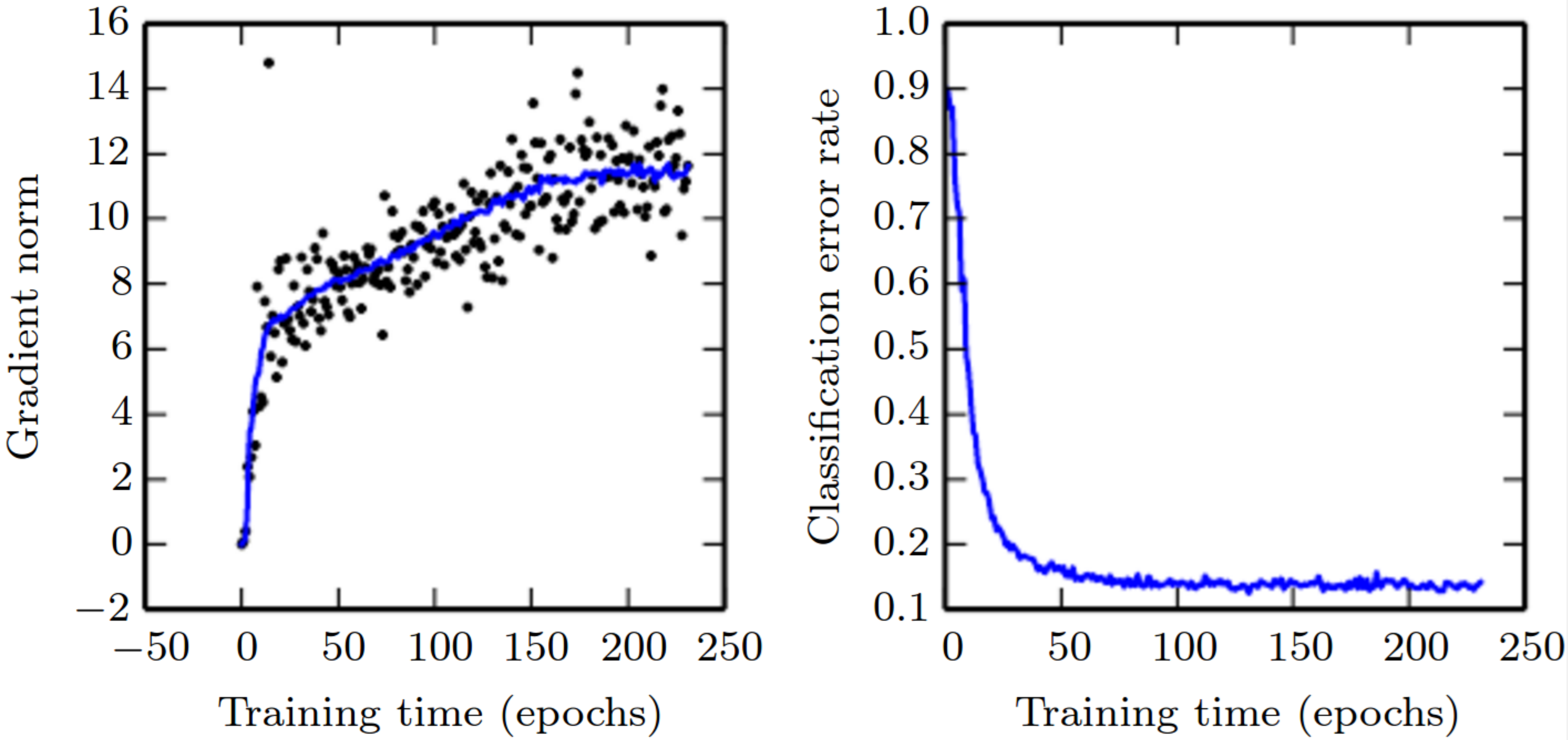
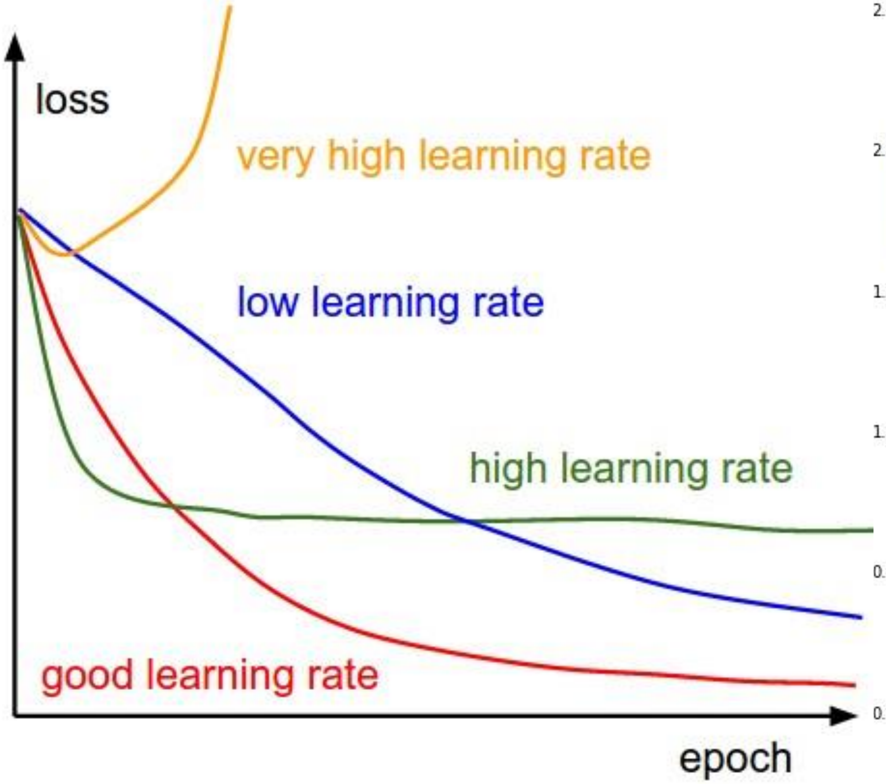


Image source: Goodfellow et al. Deep Learning. MIT Press 2016. Chap 8

Choosing the right learning rate

A crucial parameter for SGD is the learning rate ϵ .



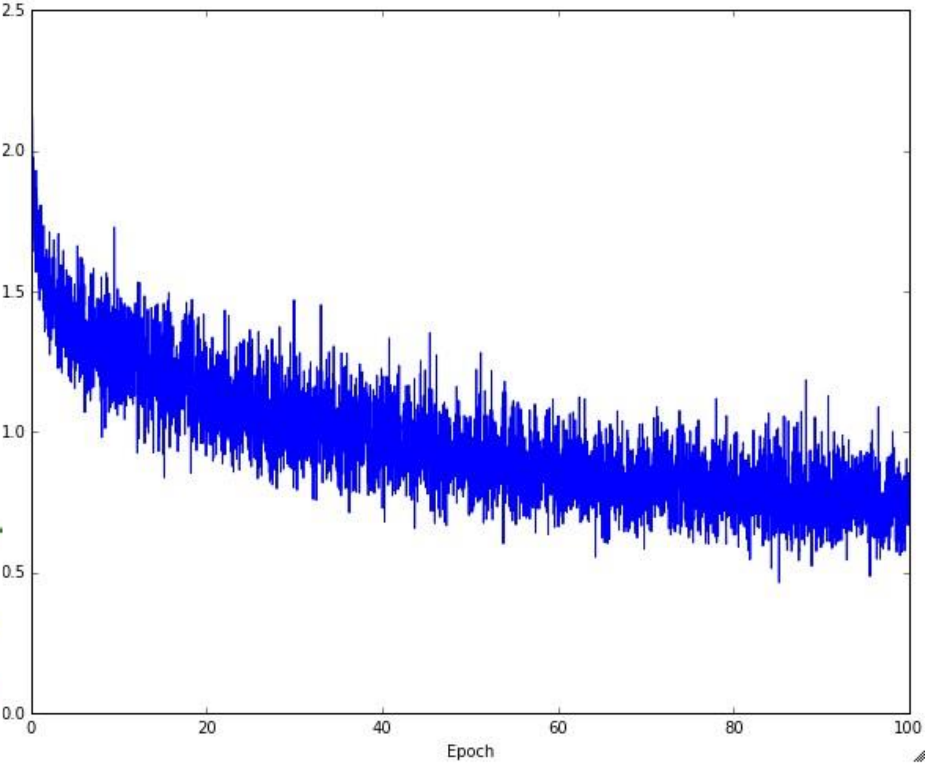


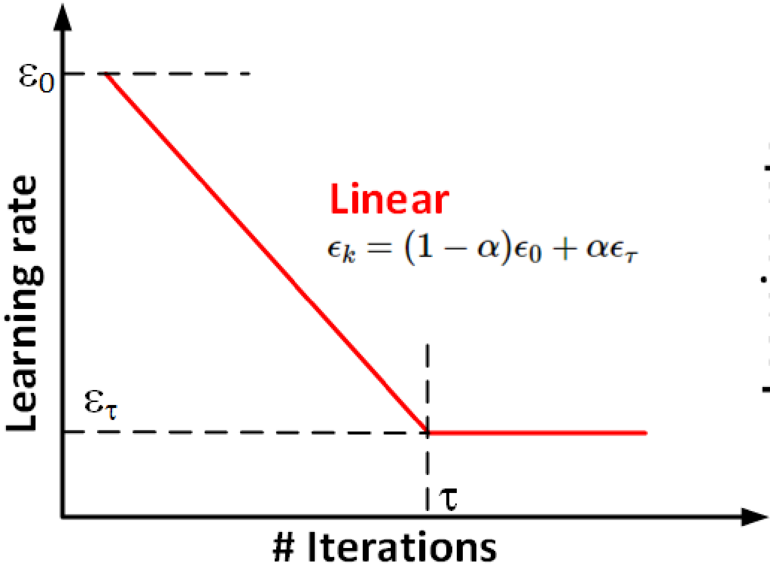
Image source: Stanford CS class [CS231n: Convolutional Neural Networks for Visual Recognition]().

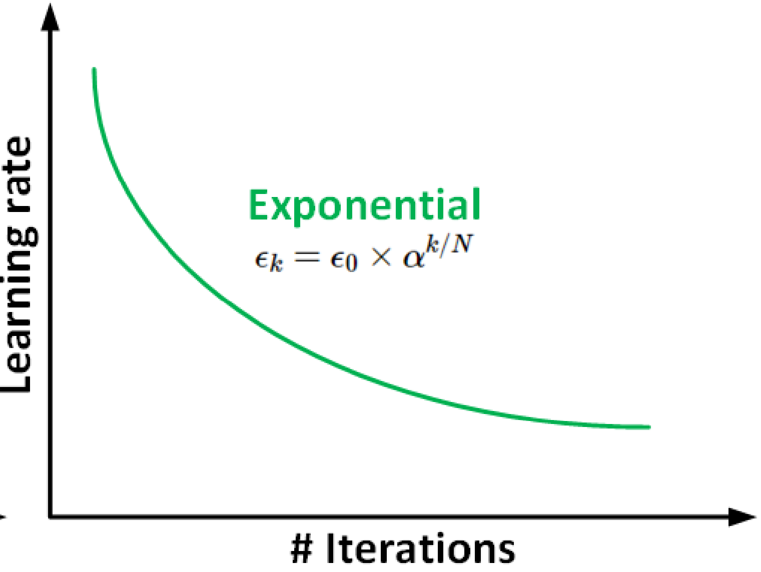
Learning rate schedules

The standard SGD uses a fixed learning rate, but in practice it is necessary to **decrease it over time**. Learning rate at iteration (epoch) k :

$$\epsilon_k = \lambda(k, \epsilon_0)$$

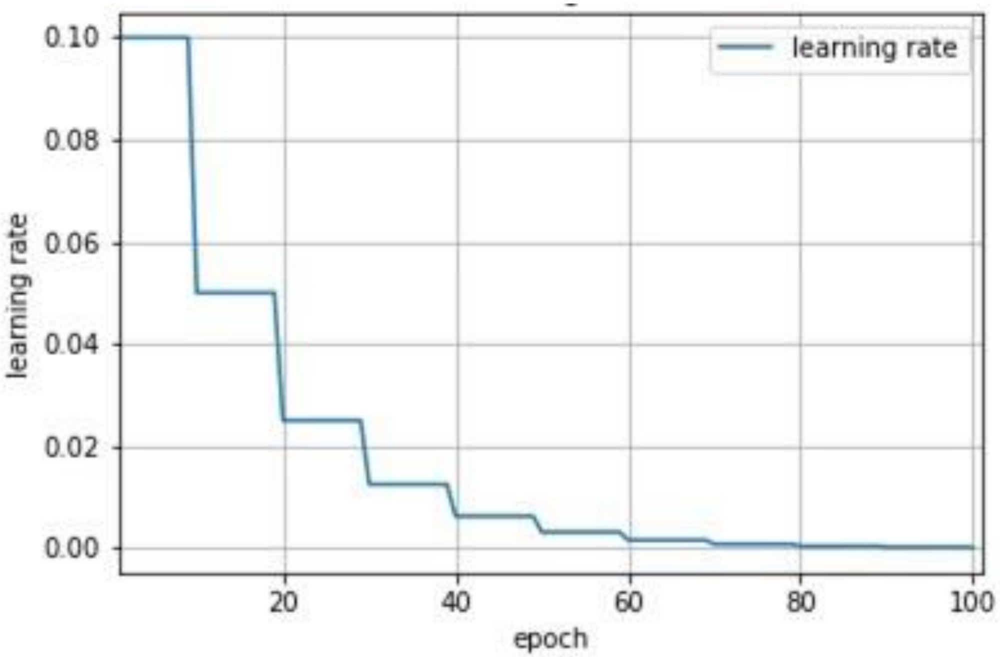
It is common to **decay** α_k monotonically until some iteration/epoch **τ**





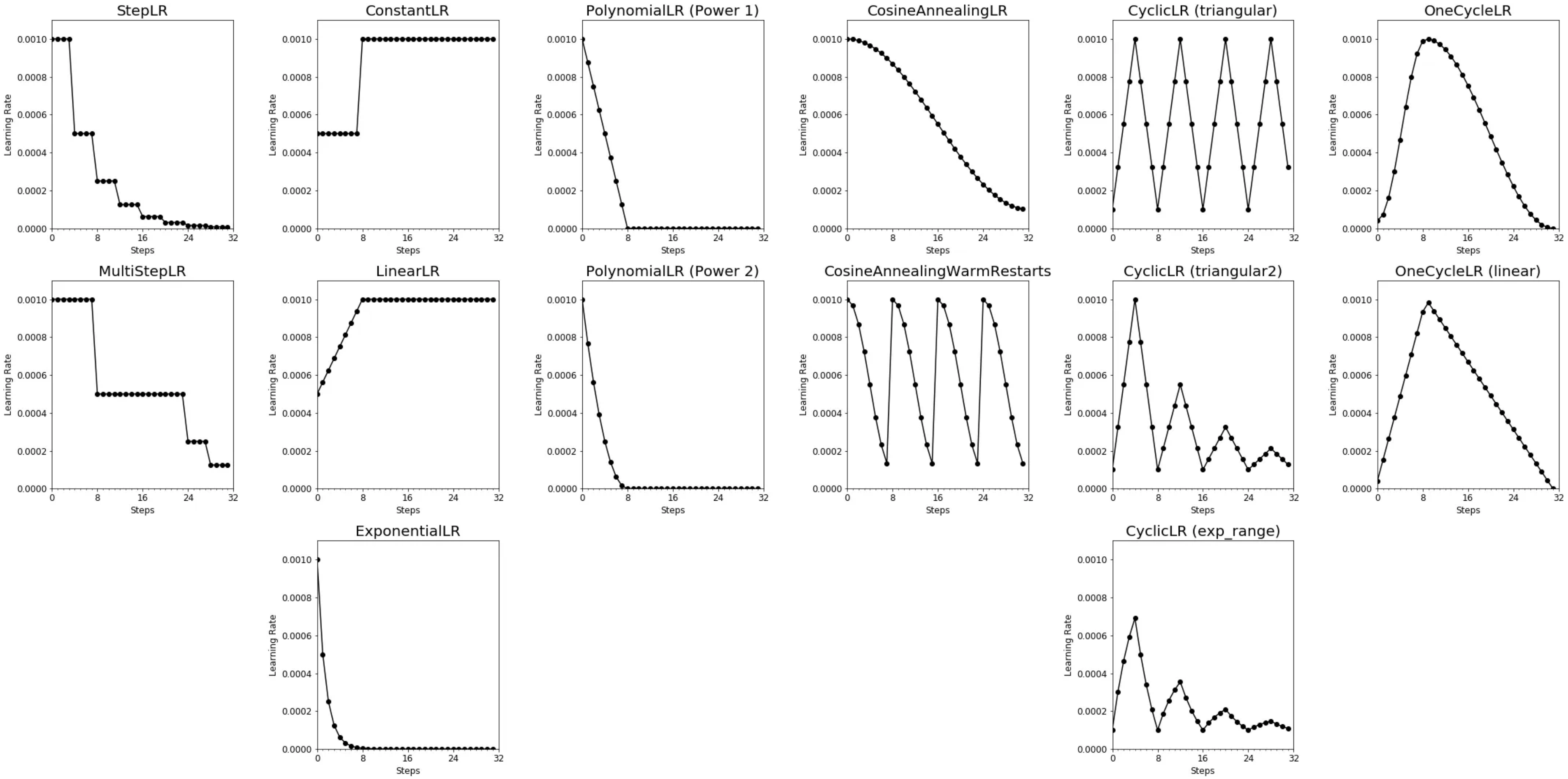
Learning rate schedules

Another common strategy when training deep models is to **drop the learning rate in "steps"** by a factor $\boldsymbol{\gamma}$ every stepsize iterations/epochs.



$$\gamma = \frac{1}{2}$$

Learning rate schedules



https://towardsdatascience.com/a-visual-guide-to-learning-rate-schedulers-in-pytorch-24bbb262c863

Learning Rate Schedulers in PyTorch

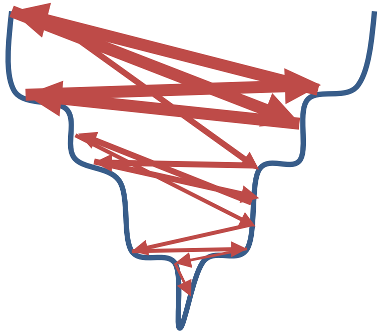
```
initial_learning_rate = 0.1

model = torch.nn.Linear(2, 1)
optimizer = torch.optim.SGD(model.parameters(),
                          lr=initial_learning_rate)
lr_lambda = lambda epoch: 0.65 ** (epoch // 10)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer,
                                    lr_lambda=lr_lambda)

# scheduling should be applied after optimizer’s update
for epoch in range(100):
    train(...) # w. optimizer.step()
    validate(...)
    scheduler.step()
```

https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

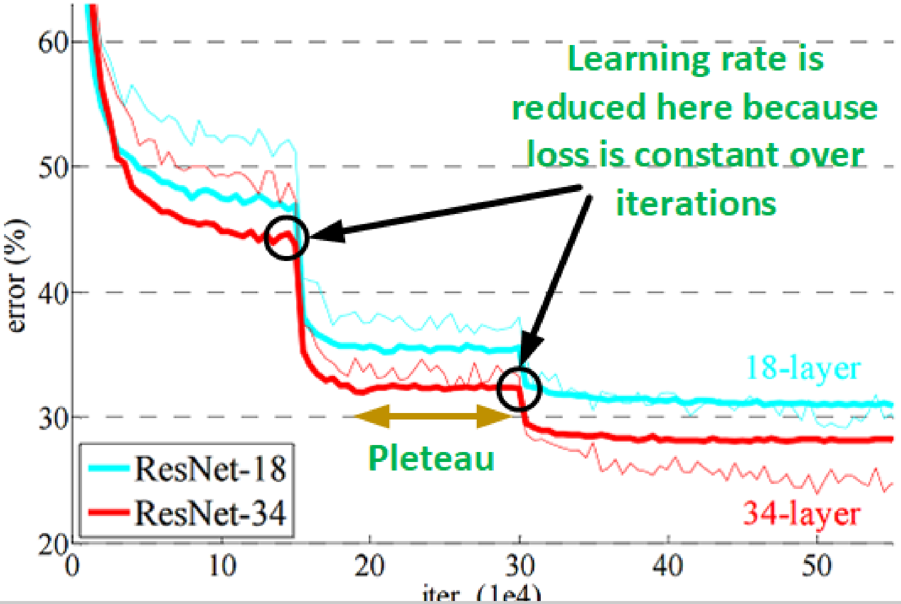
https://www.kaggle.com/isbhargav/guide-to-pytorch-learning-rate-scheduling

Learning Rate reduce on plateau

Another common approach is to **dynamically reduce the learning rate** when a metric has stopped improving.



Loss landscape



Learning rate is reduced here because loss is constant over iterations

Pleteau

```
torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
        mode='min', factor=0.1, patience=10,
        threshold=0.0001, threshold_mode='rel',
        cooldown=0, min_lr=0, eps=1e-08, verbose=False)
```

https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.ReduceLROnPlateau

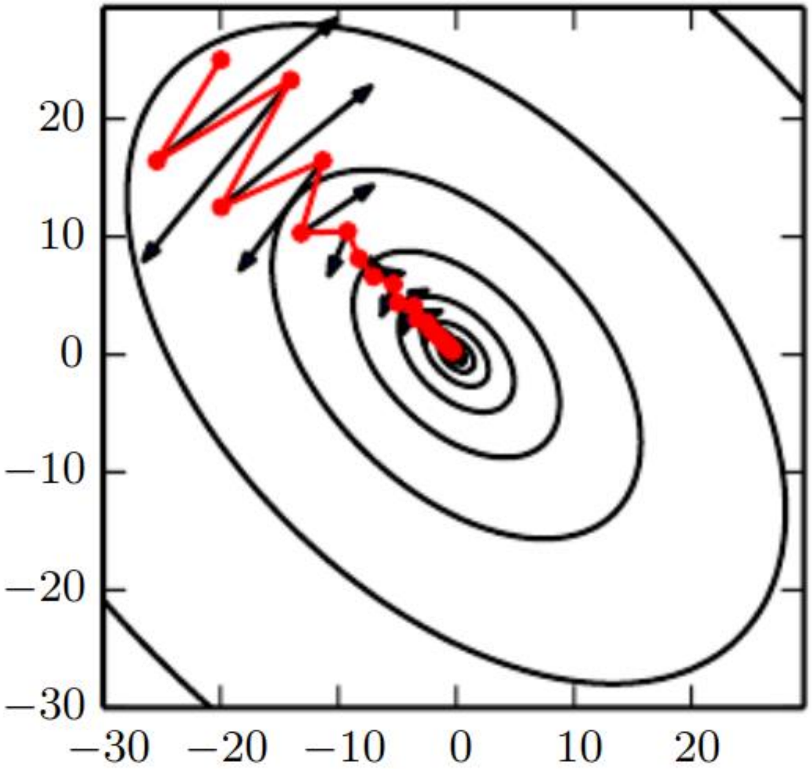
BACK TO SGD

Intial algorithm

$\mathbf{w} \leftarrow \mathbf{w} - \epsilon_k \nabla_{\mathbf{w}} J(\mathbf{w})$

SGD with Momentum

Accelerates SGD learning counitng on curvature (Polyak, 1964) aims to accelerate learning of SGD.



accumulates an exponentially decaying average of past gradients and continues to move in their direction.

$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$$

$$v \leftarrow \alpha v - \epsilon_k \nabla_{\mathbf{w}} J(\mathbf{w})$$ α weight decay

$$\mathbf{w} \leftarrow \mathbf{w} + v$$

Image source: Goodfellow et al. Deep Learning. MIT Press 2016.

SGD with Nesterov Momentum

With Nesterov momentum (Sutskever et al., 2013) the gradient is evaluated after the current velocity is applied:

$$J'(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w} + \alpha v), y^{(i)})$$

$$v \leftarrow \alpha v - \epsilon_k \nabla_{\mathbf{w}} J'(\mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + v$$

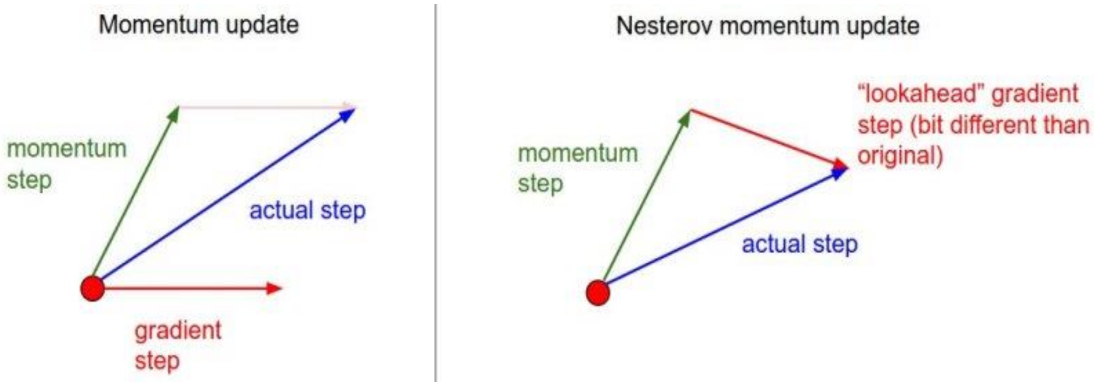


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

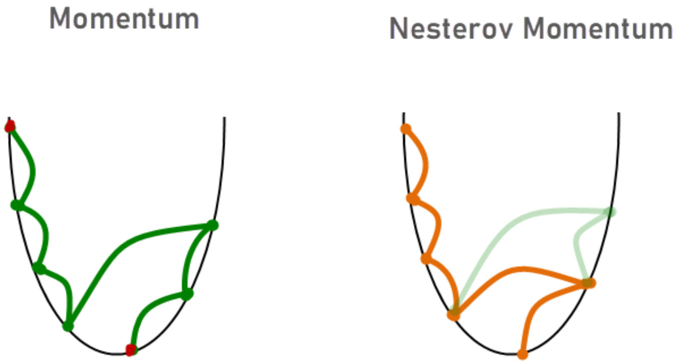


Image source: https://devskrol.com/2020/09/05/optimizer-in-neural-network/

SGD with Momentum in PyTorch

```
torch.optim.SGD(params, lr=<required parameter>,
                momentum=0, dampening=0,
                weight_decay=0, nesterov=False)
```

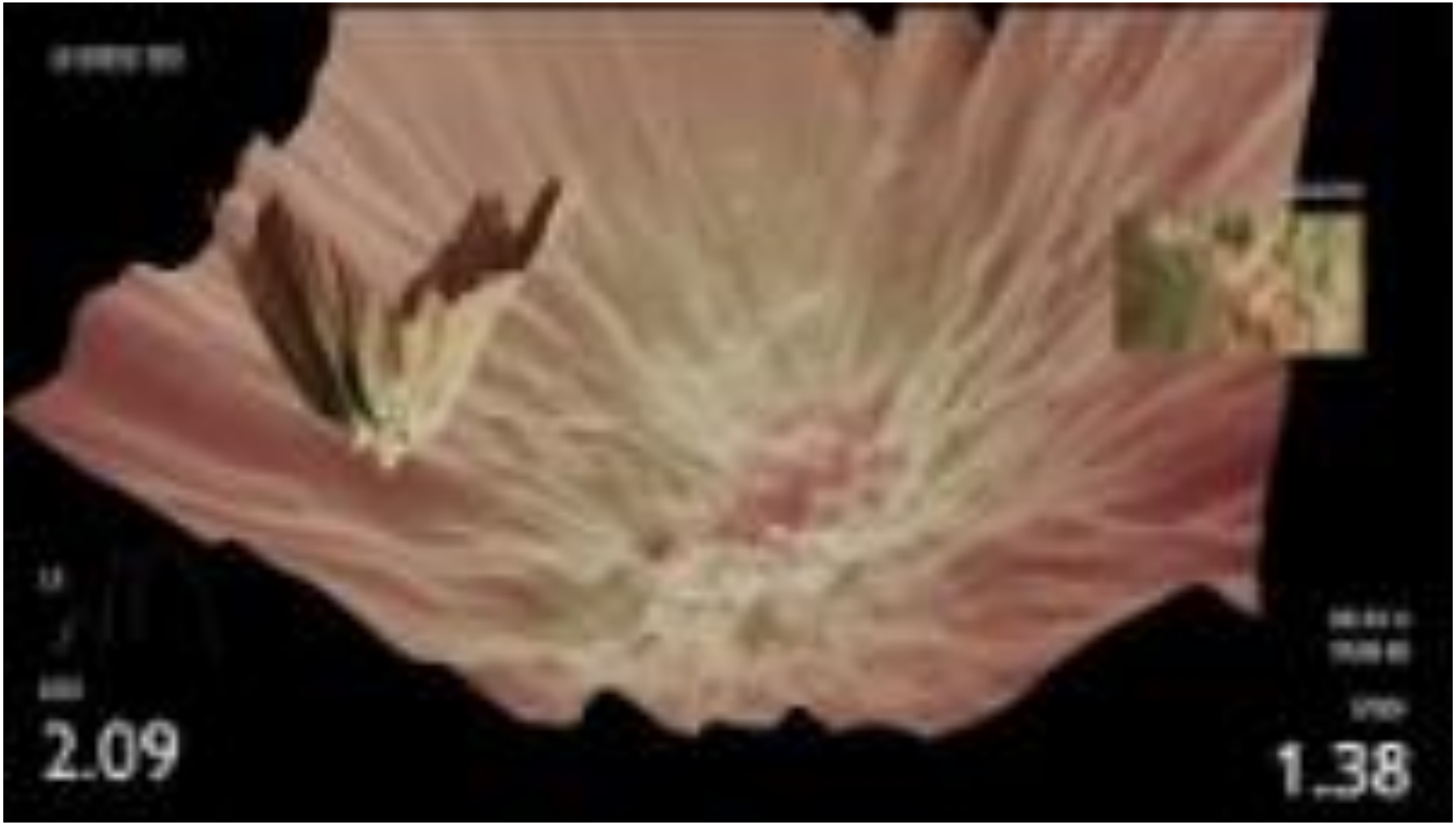
- momentum (*float*, *optional*) – momentum factor (default: 0)
- dampening (*float*, *optional*) – dampening for momentum (default: 0)
- nesterov (*bool*, *optional*) – enables Nesterov momentum (default: False)

Note that the optional "dampening" term is used as follows:

```
v = momentum * v + (1-dampening) * gradientW
W = W - lr * v
```

Also note that in PyTorch, the learning rate is also applied to the momentum terms, instead of the original definition.

https://pytorch.org/docs/stable/optim.html#torch.optim.SGD



https://losslandscape.com/
https://www.youtube.com/watch?v=QeViLO0pU1I&t=5s

ALGORITHMS WITH ADAPTIVE LEARNING RATE

Algorithms with Adaptive Learning Rates

The learning rate is one of the most difficult hyperparameters to set.

It significantly affects the performance of the model.

The cost may be often sensitive to some directions in the parameter space but insensitive to others.

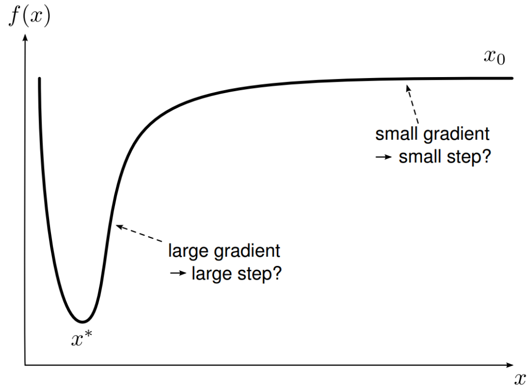
The momentum algorithm somehow mitigates this issue but at the cost of introducing another hyperparameter.

AdaGrad (Duchi et al. 2011)

Idea: adapts the learning rates of **each individual model parameter** by scaling it inversely proportional to the square root of the sum of all the historical squared values of the gradient.

Parameters with larger partial derivative of the loss have a faster decrease in their learning rate, and viceversa.

The net effect is greater progress in the more gently sloped directions of parameter space.



Intial algorithm

$\mathbf{w} \leftarrow \mathbf{w} - \epsilon_k \nabla_{\mathbf{w}} J(\mathbf{w})$

The AdaGrad algorithm

```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\mathbf{w}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability

Initialize gradient accumulation variable $r = 0$
while stopping criterion not met do
    Sample a minibatch of m samples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$
    with corresponding targets $y^{(i)}$.
    Compute gradient:              $\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
    Accumulate squared gradient:   $r \leftarrow r + \nabla_{\mathbf{w}} J(\mathbf{w}) \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Compute update:                $\Delta\mathbf{w} \leftarrow \frac{\epsilon}{\delta + \sqrt{r}} \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Apply update:                  $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$

end while
```

RMSProp (Hinton, 2012)

Modifies AdaGrad to perform better in nonconvex landscapes, changing the gradient accumulation.

AdaGrad takes the whole history, skiping local structures

RMSProp uses an exponentially decaying average to discard history from the extreme past.

The RMSProp algorithm

```
Require: Global learning rate $\epsilon$, decay rate $\rho$
Require: Initial parameter $\mathbf{w}$
Require: Small constant $\delta$, usually $10^{-6}$, for numerical stability

Initialize gradient accumulation variables $r = 0$
while stopping criterion not met do
    Sample a minibatch of m samples from the training set $\{x^{(1)}, \dots , x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient: $\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
    Accumulate squared gradient: $r \leftarrow \rho r + (1 - \rho) \nabla_{\mathbf{w}} J(\mathbf{w}) \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Compute update: $\Delta\mathbf{w} \leftarrow \frac{\epsilon}{\delta + \sqrt{r}} \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Apply update: $\mathbf{w} \leftarrow \mathbf{w} - \Delta\mathbf{w}$

end while
```

AdaDelta (Zeiler 2012)

Extension of Adagrad to reduce its aggressivness, monotonically decreasing the learning rate.
Close to RMSProp (appeared simultanously)

restricts the window of accumulated past gradients to some fixed size L.

RMSProp

$$\Delta\mathbf{w} \leftarrow \frac{\epsilon}{\delta+\sqrt{r}} \odot \nabla_{\mathbf{w}} J(\mathbf{w})$$

The AdaDelta algorithm

```
**Require**: Global learning rate $\epsilon$, decay rate $\rho$
**Require**: Initial parameter $\mathbf{w}$
**Require:** Small constant $\delta$, usually $10^{-6}$, for numerical stability

Initialize gradient accumulation variables $\boldsymbol{r} = \boldsymbol{0}$
**while** stopping criterion not met **do**
    Sample a minibatch of m samples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$
    with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient:               $\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
    Accumulate squared gradient:    $r \leftarrow \rho r + (1-\rho) \nabla_{\mathbf{w}} J(\mathbf{w}) \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Compute update:                 $\Delta\mathbf{w} \leftarrow \frac{\sqrt{\Delta\mathbf{w}}}{\delta+\sqrt{r}} \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Apply update:                   $\mathbf{w} \leftarrow \mathbf{w} - \Delta\mathbf{w}$

**end while**
```

Adam (Kingma and Ba, 2014)

Adam: “adaptive moments.”

A variant on the combination of RMSProp and momentum.

Momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient.

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.

The Adam algorithm

RMSProp

$$\Delta\mathbf{w} \leftarrow \frac{\epsilon}{\delta+\sqrt{r}} \odot \nabla_{\mathbf{w}} J(\mathbf{w})$$

```
**Require**: Global learning rate $\epsilon$, decay rates $\rho_1$ ,$\rho_2$
**Require**: Initial parameter $\boldsymbol{\theta}$
**Require:** Small constant $\boldsymbol{\delta}$, usually $10^{-6}$, for numerical stability

Initialize gradient accumulation variables $\boldsymbol{r = 0}$
**while** stopping criterion not met **do**
    Sample a minibatch of m samples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$
    with corresponding targets $y^{(i)}$.
    Compute gradient:              $\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$
    Accumulate gradient behaviour: $r \leftarrow \rho_1 r + (1-\rho_1) \nabla_{\mathbf{w}} J(\mathbf{w}) \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
                                   $v \leftarrow \rho_2 v + (1-\rho_2) \nabla_{\mathbf{w}} J(\mathbf{w})$
    Compute update:                $\Delta\mathbf{w} \leftarrow \frac{\epsilon v}{\delta+\sqrt{r}} \odot \nabla_{\mathbf{w}} J(\mathbf{w})$
    Apply update:                  $\mathbf{w} \leftarrow \mathbf{w} - \Delta\mathbf{w}$
**end while**
```

Choosing the right optimization algorithm

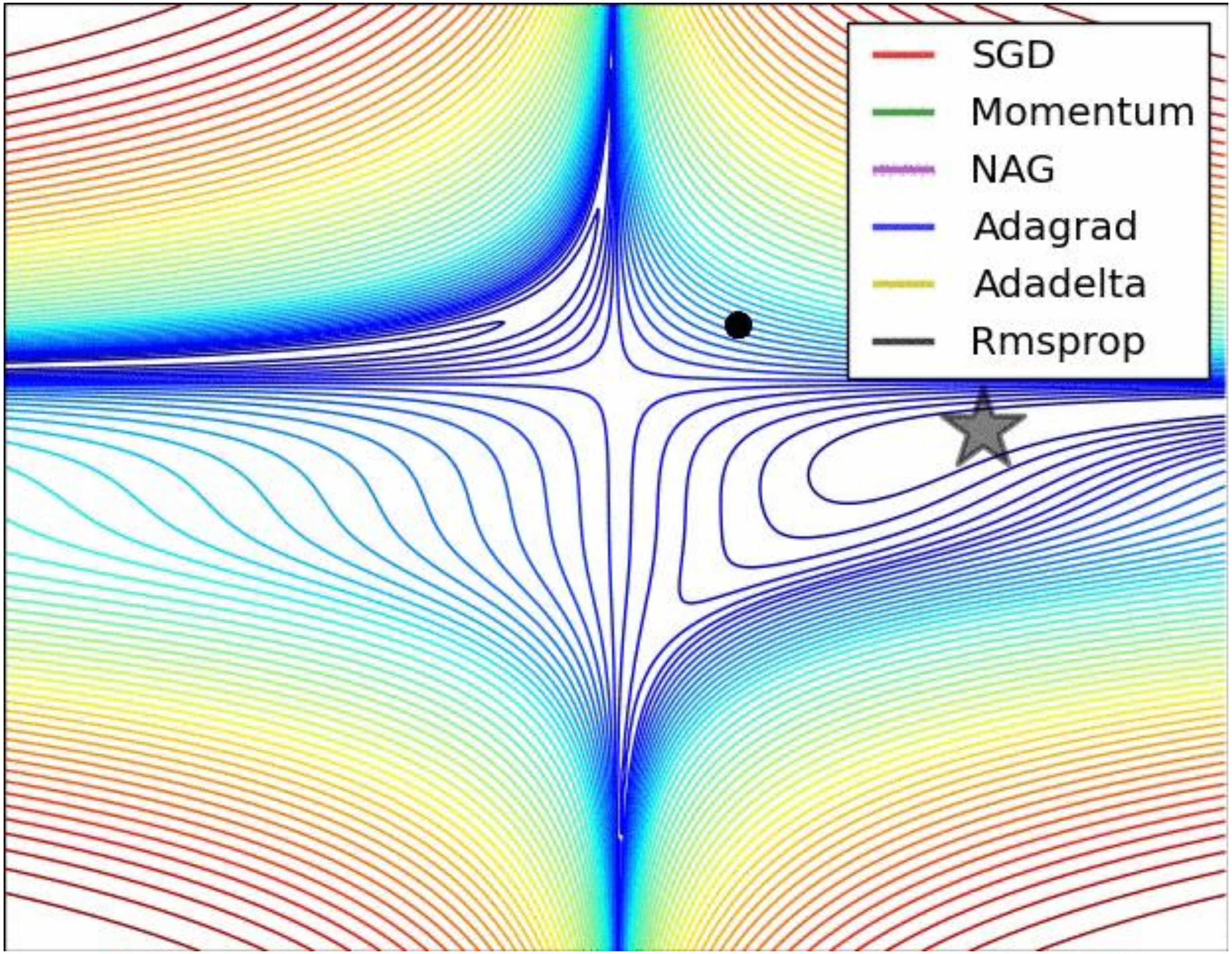
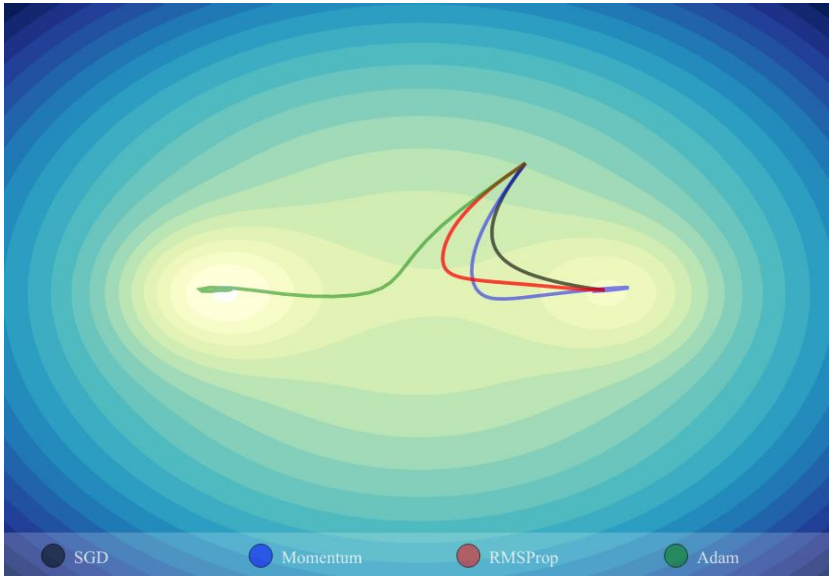
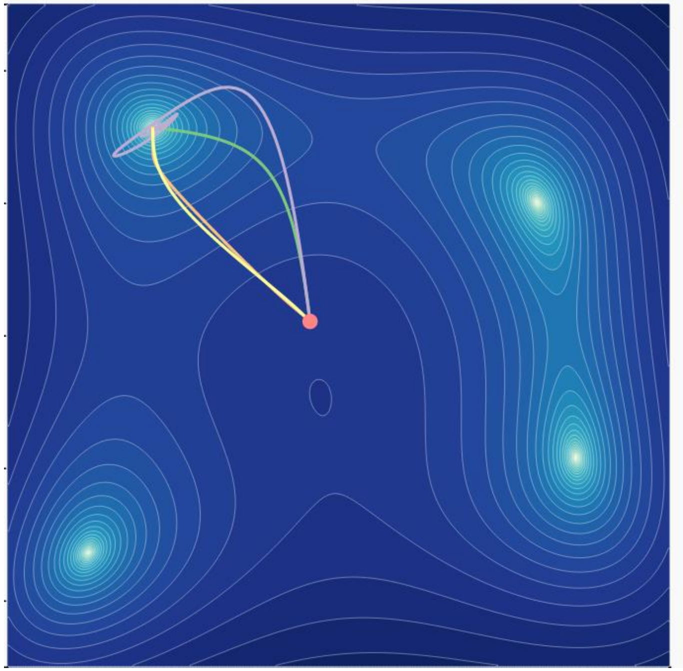


Image source: Alec Radford.

Choosing the right optimization algorithm



https://emiliendupont.github.io/2018/01/24/optimization-visualization/



https://www.deeplearning.ai/ai-notes/optimization/

Choosing the right optimization algorithm

Unfortunately, there is currently no consensus!

(Schaul et al. 2014) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks.
Results suggest that the family of algorithms with adaptive learning rates performed fairly robustly, but single best algorithm has emerged.

As a rule of thumb, **you should try first an algorithm with adaptive learning rate**.

Algorithms with Adaptive Learning Rates in

```
torch.optim.Adagrad(params, lr=0.01, lr_decay=0,
                  weight_decay=0, initial_accumulator_value=0,
                  eps=1e-10)

torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08,
                    weight_decay=0, momentum=0, centered=False)

torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06,
                     weight_decay=0)

torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999),
                eps=1e-08, weight_decay=0, amsgrad=False)
```

https://pytorch.org/docs/stable/optim.html#algorithms

Cliffs and Exploding Gradients

Neural Networks with many layers often have extremely steep regions resembling cliffs.
On the face of an extremely steep cliff the gradient update step can move the parameters extremely far. → Exploding gradients

Workaround: **Gradient clipping heuristic.**

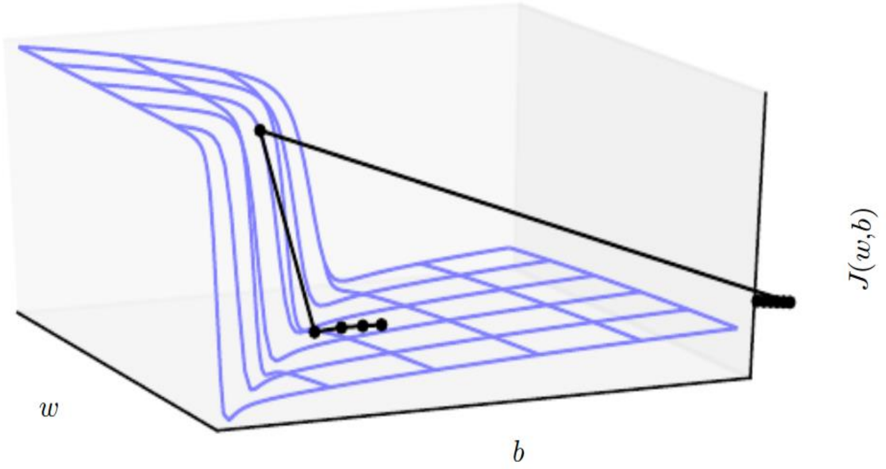


Image source: Goodfellow et al. Deep Learning. MIT Press 2016.

Gradient Clipping in PyTorch

```
nn.utils.clip_grad_norm_(model.parameters(), clip)
```

Clips gradient norm of an iterable of parameters.

The norm is computed over all gradients together, as if they were concatenated into a single vector. Gradients are modified in-place.

Should be called before `optimizer.step()`

https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html

OTHER OPTIMIZATION STRATEGIES

Regularization: L2/L1 Parameter regularization

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \sum_i w_i^2$$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha \sum_i |w_i|$$

$\boldsymbol{\alpha}$ = weight decay parameter!

Regularization: Early stopping

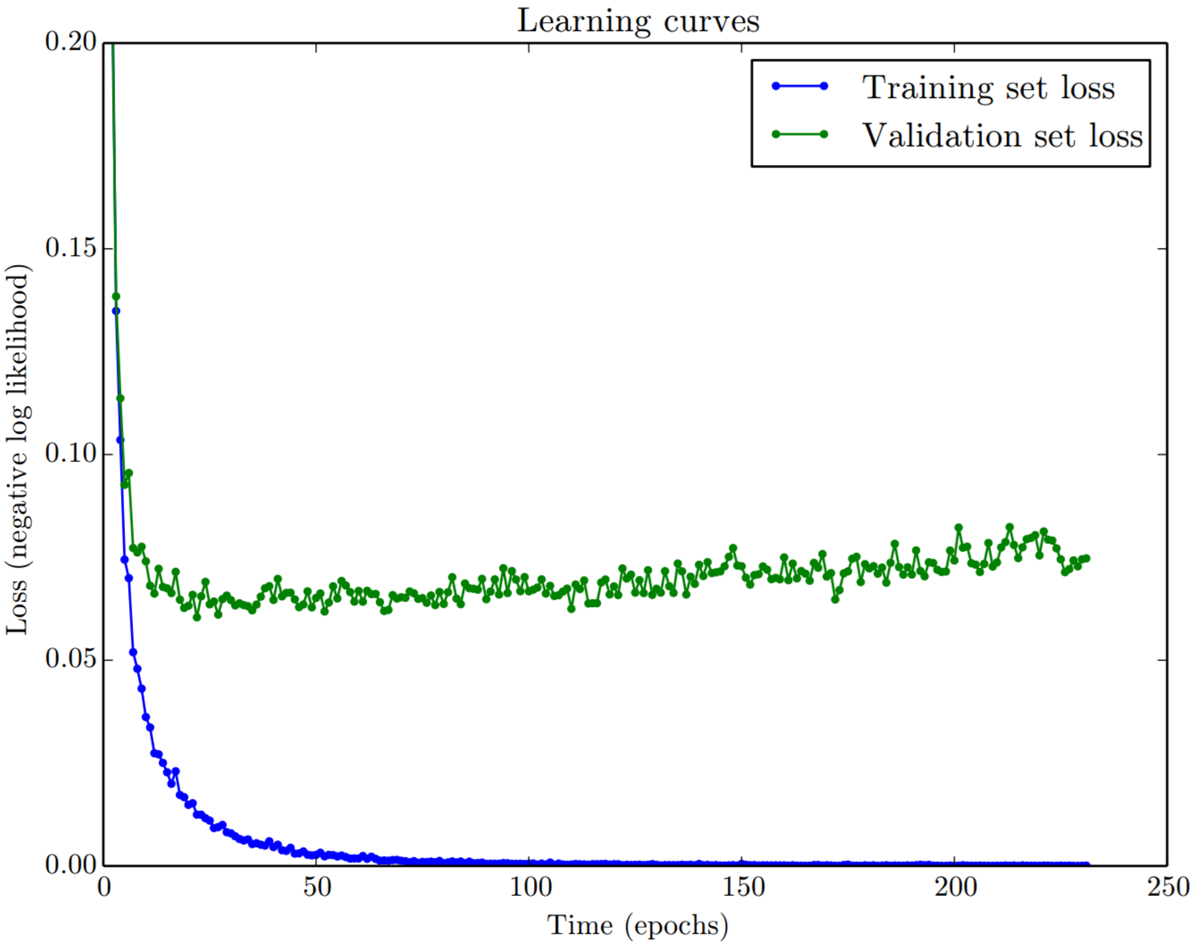
Training algorithms do not usually halt at a local minimum.

A machine learning algorithm typically halts when a convergence criterion based on **early stopping** is satisfied.

The early stopping criterion is often based on the true underlying loss function, such as the 0-1 loss measured on a validation set.

It is designed to halt whenever overfitting begins to occur. This not necessarily means when the gradients became very small.

Regularization: Early stopping



Regularization: Early stopping in PyTorch

```
best_err = -np.Inf
wait = 0
patience = 10

for epoch in range(100):
    train(...)
    val_err = validate(...)
    wait +=1
    if val_err < best:
       best = val_err
       # torch.save(model.state_dict(), PATH)
       wait = 0

    if wait >= patience:
      break
```

Data preparation

- Data quantity (never enough): data aumentation

- Data range: features should have similar ranges: data normalization

Data Augmentation

Best way to make a machine learning model generalize better is to train it on more data -> **Create new data!**

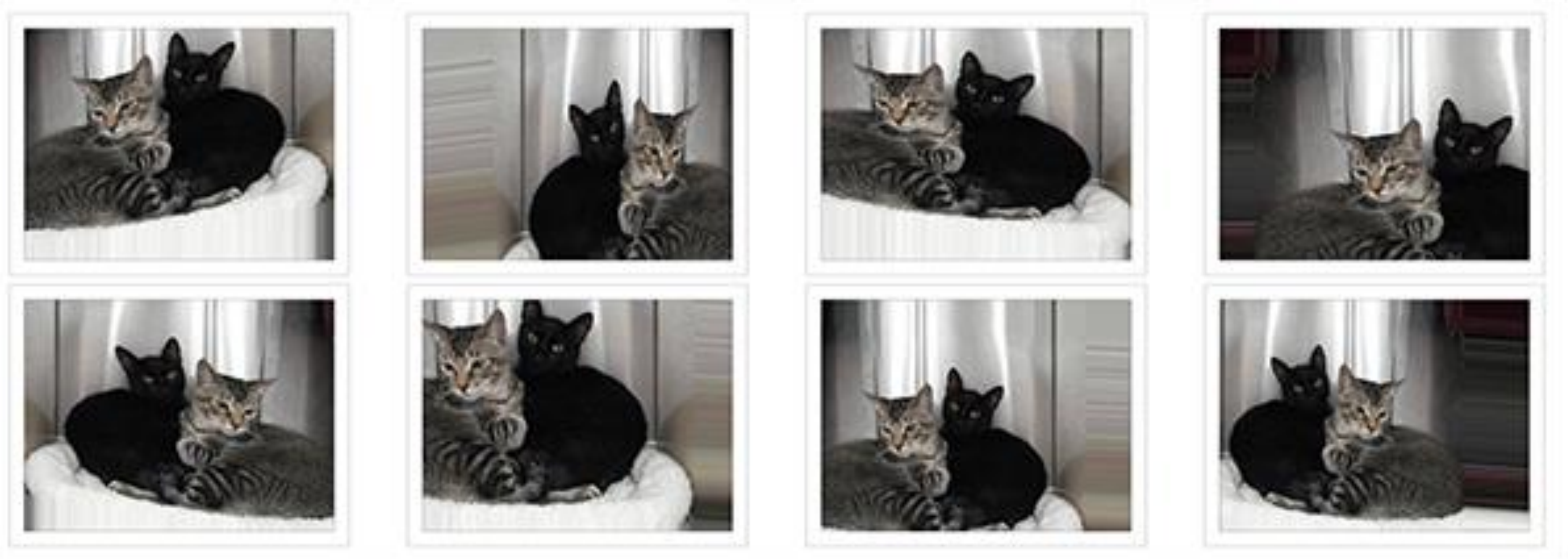


Image source: [Building powerful image classification models using very little data]()

Data Augmentation in PyTorch

```
From torchvision import transforms

rgb_mean = (0.4914, 0.4822, 0.4465)
rgb_std = (0.2023, 0.1994, 0.2010)

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(rgb_mean, rgb_std),
])
```

https://pytorch.org/vision/stable/transforms.html

Data Augmentation in PyTorch

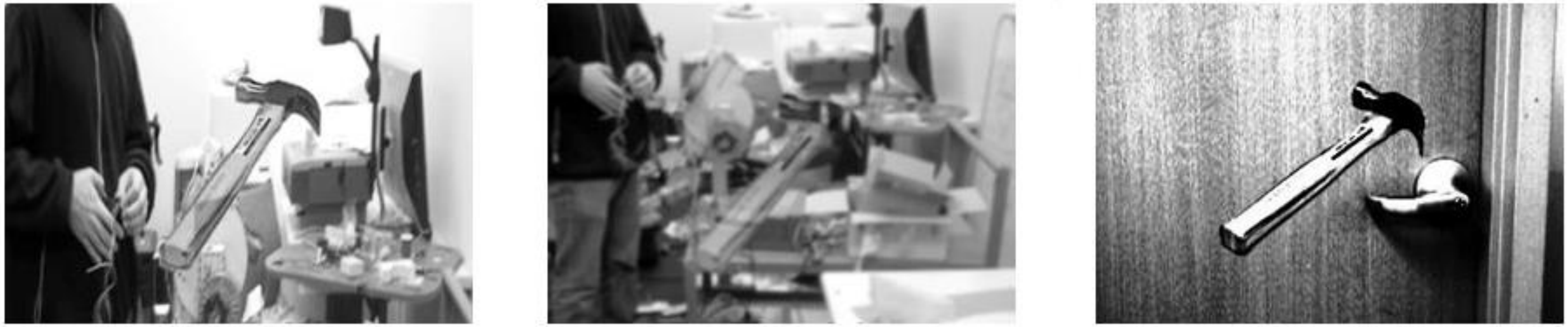
```
From torchvision import transforms

rgb_mean = (0.4914, 0.4822, 0.446
rgb_std = (0.2023, 0.1994, 0.2

transform_train = transfo        pose([
    transforms.RandomC          padding=4),
    transforms.Rand          ontalFlip(),
    transforms.To          (),
    transforms          ize(rgb_mean, rgb_std),
])
```

Never ever on validation or test sets!!!!!

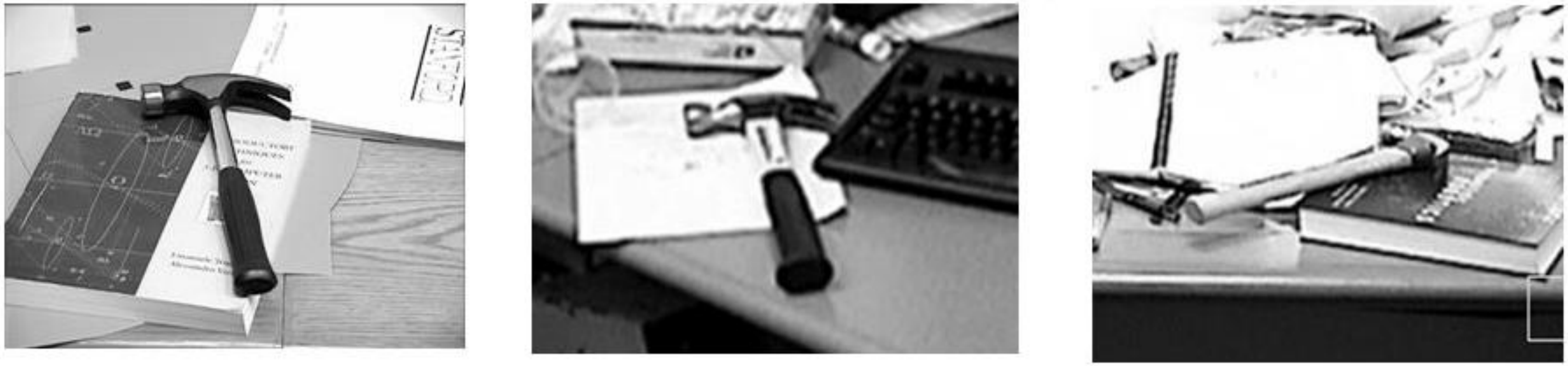
https://pytorch.org/vision/stable/transforms.html

Synthetic Data

Synthetic Examples (Training set)



Real Examples (Test set)



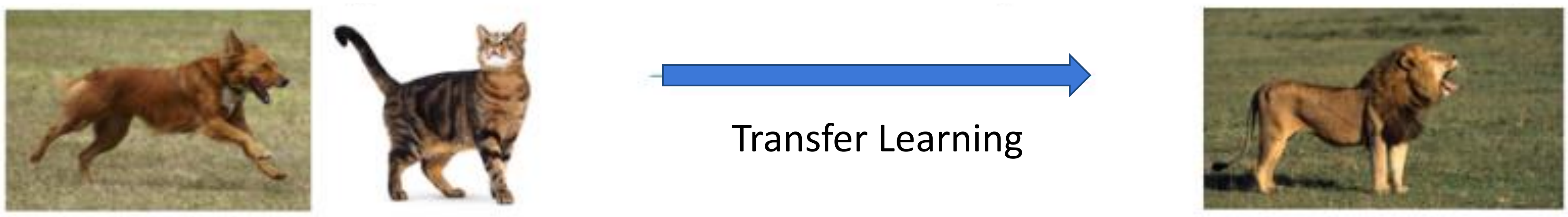
Synthetic Data

Supervised pre-training (fine-tuning)

- Initialize the weights of your model using the optimal weights learnt on a similar task!
- Fine-tuning can provide a reasonably good model even when we have very few training data.

If you know how to recognize...

You will be able to recognize...



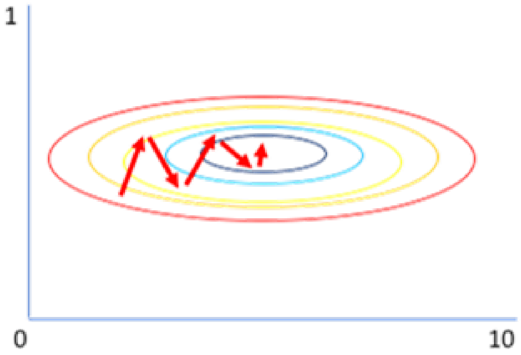
Transfer Learning

Image source: Yannis Ghazouani, 2016.

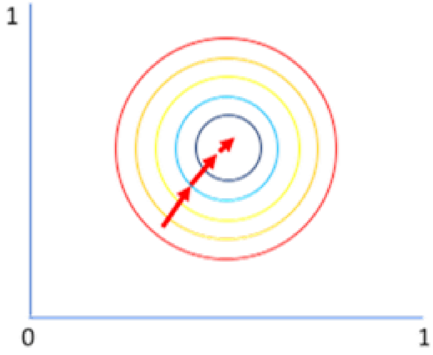
Data Normalization

Aim: Get features on the same range

get statistics on training data, apply on evaluation data



Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

- MinMax scaler
- Data standarization (mean=0, std=1)
- Whitening

Figure: https://www.jeremyjordan.me/batch-normalization/

Data Preprocessing: PCA/ZCA Whitening

The data is first centered.

```
# Assume input data matrix X of size [N x D]
X -= np.mean(X, axis = 0) # zero-center the data (important)
```

Then, we can compute the covariance matrix that tells us about the correlation structure in the data:

```
cov = np.dot(X.T, X) / X.shape[0] # get the data covariance matrix
```

Data Preprocessing: PCA/ZCA Whitening

The covariance matrix is symmetric and positive semi-definite. We can compute its SVD factorization:

```
U,S,V = np.linalg.svd(cov)
```

U:eigenvectors S:eigen values

Decorrelate data, projecting them (zero-centered) into the eigenbasis:

```
Xrot = np.dot(X, U)  # decorrelate the data
```

Takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale:

```
# whiten the data: divide by the eigenvalues
Xwhite = Xrot / np.sqrt(S + 1e-5)
```

Data Preprocessing: PCA/ZCA Whitening

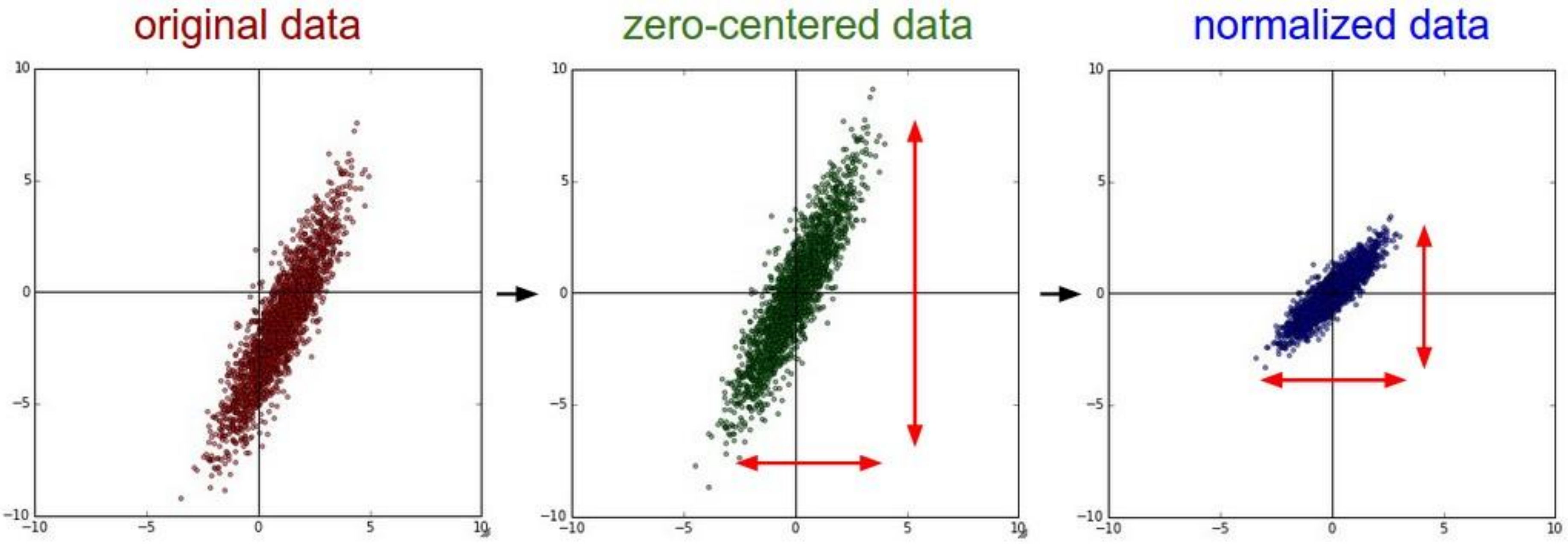


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

See: LeCun, Y., Bottou, L., Orr, G., and Muller, K. Efficient backprop. In Orr, G. and K., Muller (eds.), Neural Networks: Tricks of the trade. Springer, 1998b.

Data Preprocessing: PCA/ZCA Whitening

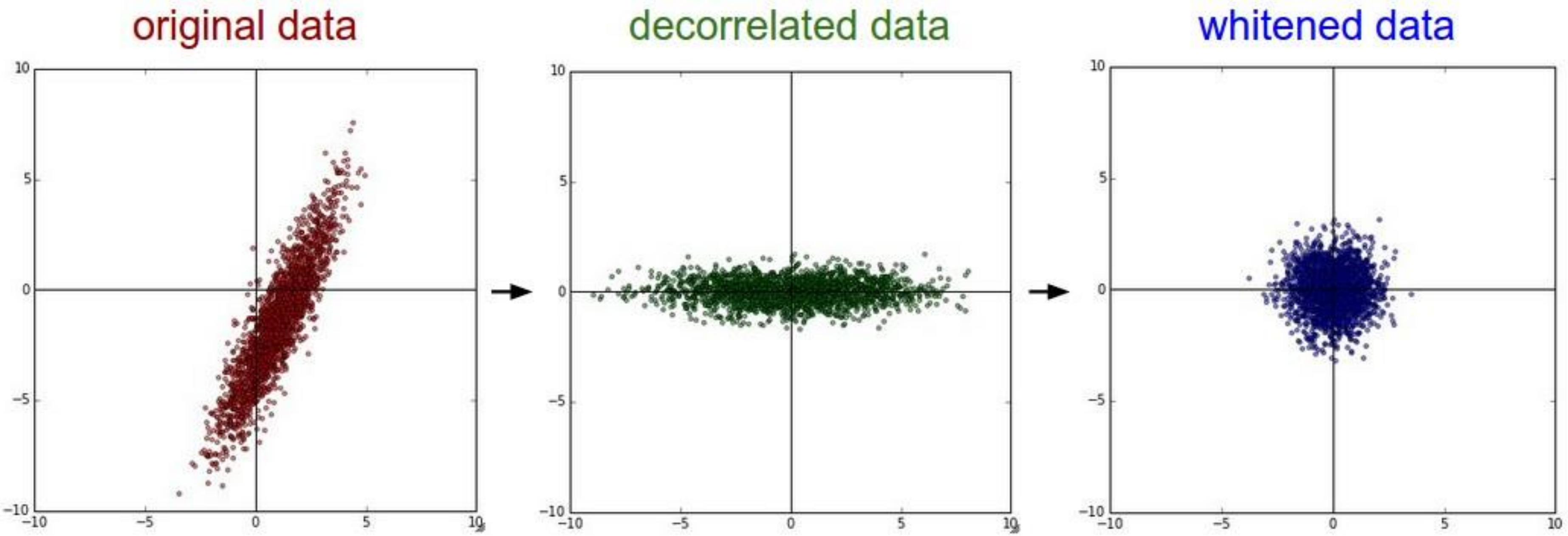
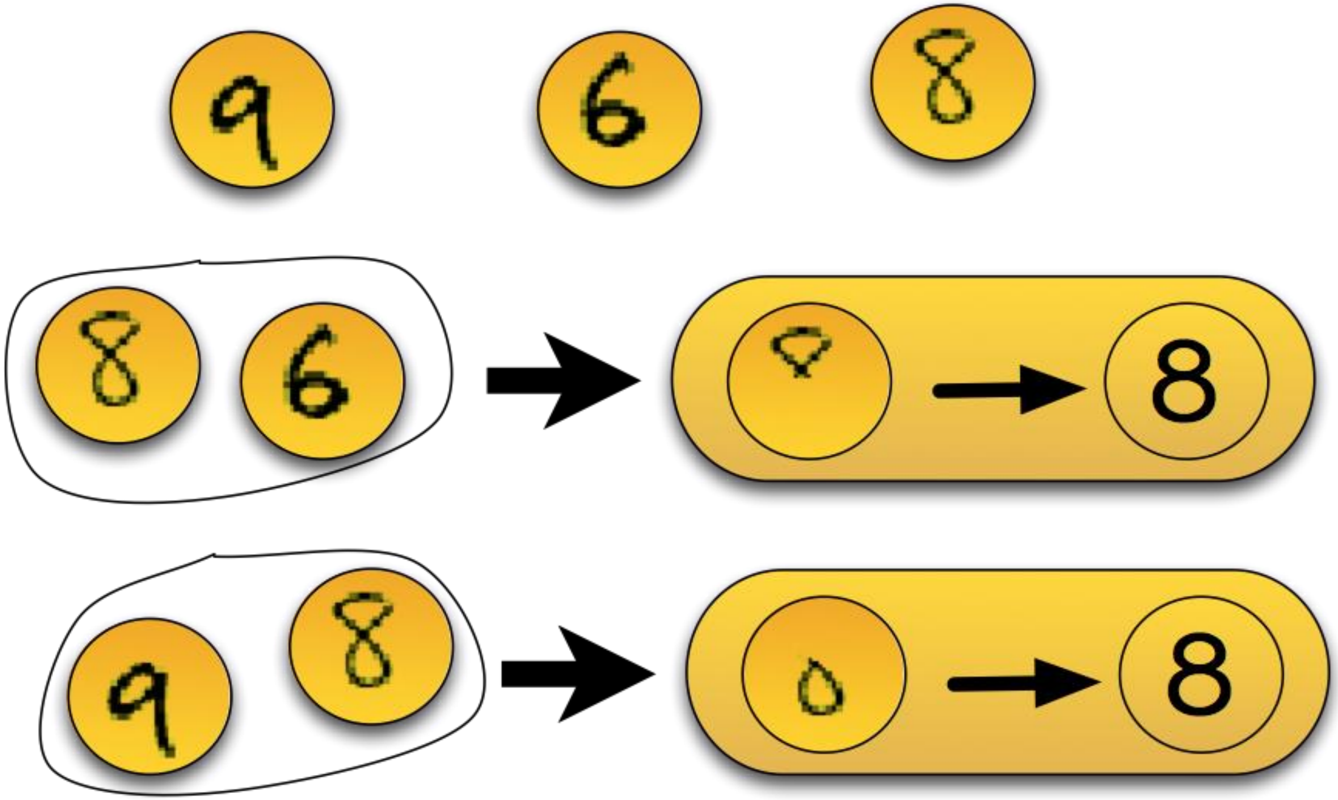


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

See: LeCun, Y., Bottou, L., Orr, G., and Muller, K. Efficient backprop. In Orr, G. and K., Muller (eds.), Neural Networks: Tricks of the trade. Springer, 1998b.

Regularization: Bagging



Slide credits: I. Goodfellow. Introduction to Deep Learning, AERFAI Deep Learning Autumn School, Nov 5, 2015, Valencia.

Regularization: Dropout

Dropout training: simulate bagging by dropping each unit in the network with probability p

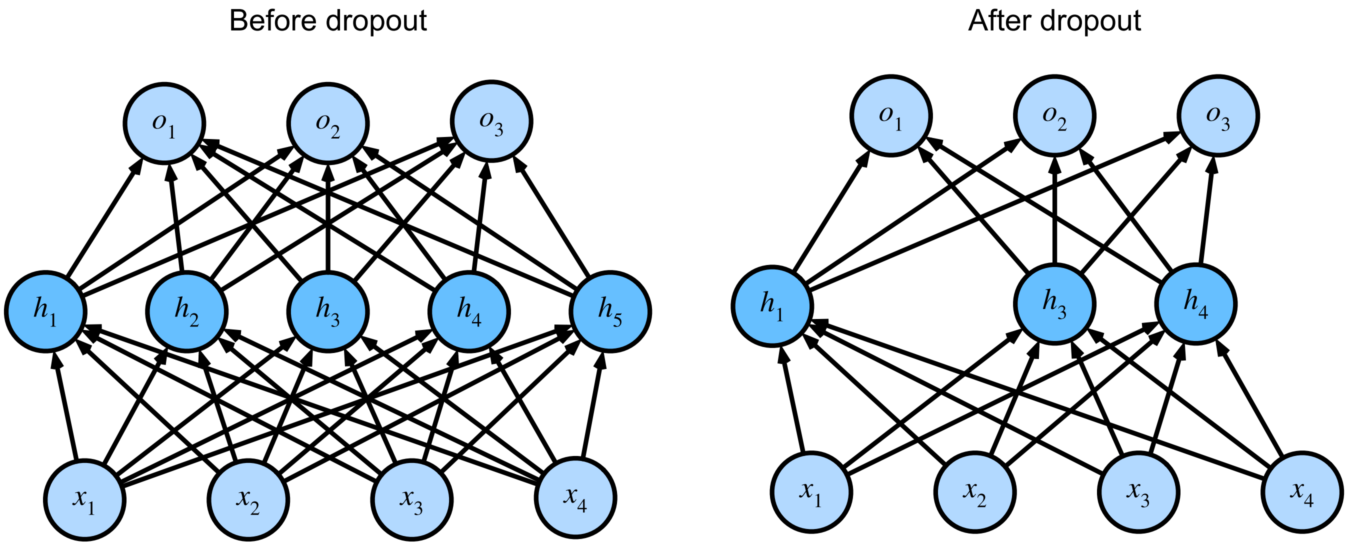


Figure: https://d2l.ai/chapter_multilayer-perceptrons/dropout.html

Regularization: Dropout in PyTorch

```
torch.nn.Dropout(p=0.5, inplace=False)
```

https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

https://losslandscape.com/ https://youtu.be/2PqTW_p1fls

Batch Normalization

Better and Faster Way to Train Convolutional Networks

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
Sergey Ioffe, Christian Szegedy, ICML'15

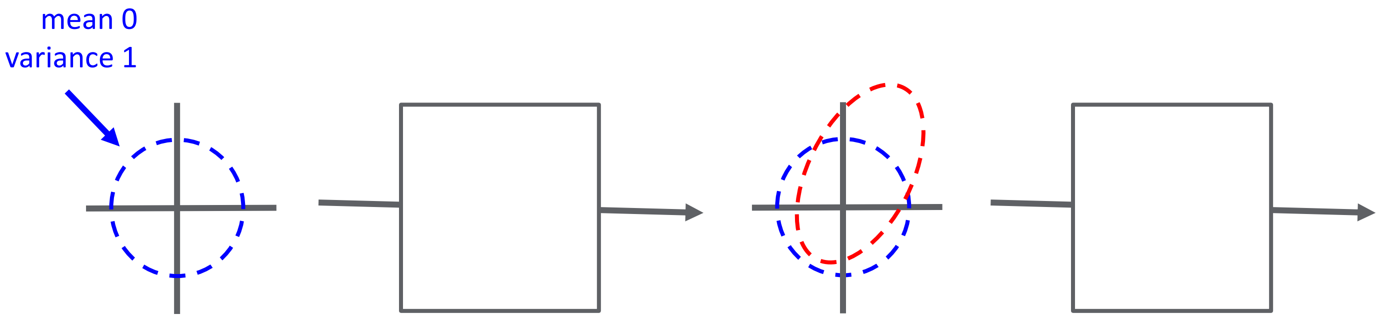
10x speedup in training, 2% improvement in performance on ImageNet!

Beautifully simple method attacking the core of what makes deep networks difficult to train.

Batch Normalization

SGD learns better on whitened data.

This is true for the inputs, but also for every layer up the stack:

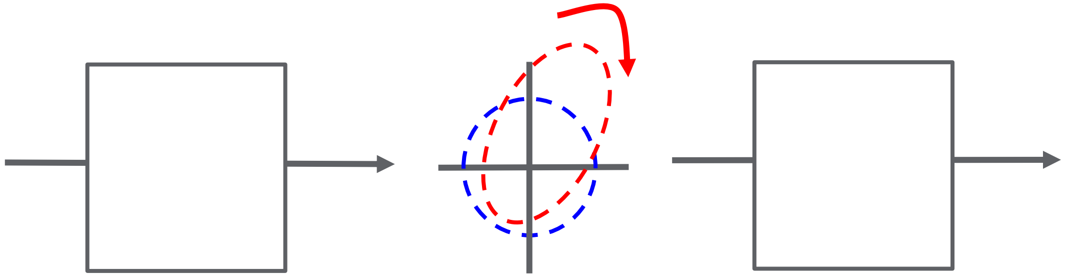


Problem: the distribution of activations changes over time!

Slide credits: I. Goodfellow. Deep Learning for Computer Vision, AERFI Deep Learning Autumn School, Nov 5, 2015, Valencia. Adapted from slides by Vincent Vanhoucke.

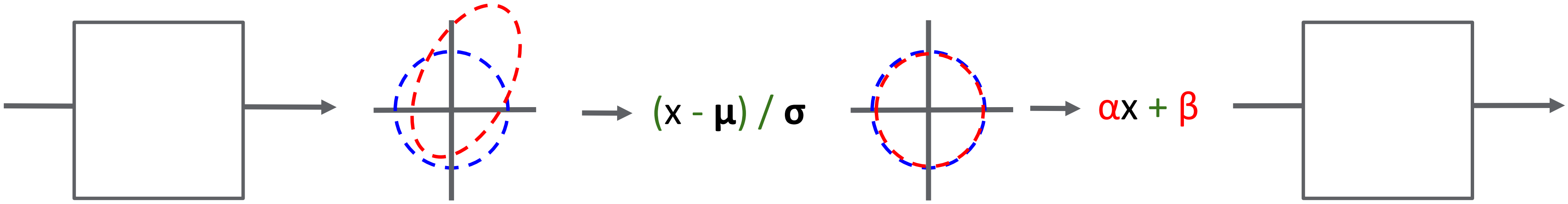
Batch Normalization

Before:



“How to train your neural network”

After:



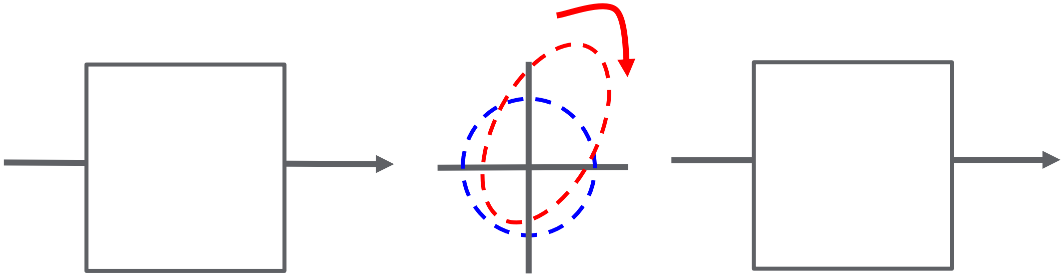
$(x - \mu) / \sigma$

$\alpha x + \beta$

Slide credits: I. Goodfellow. Deep Learning for Computer Vision, AERFI Deep Learning Autumn School, Nov 5, 2015, Valencia. Adapted from slides by Vincent Vanhoucke.

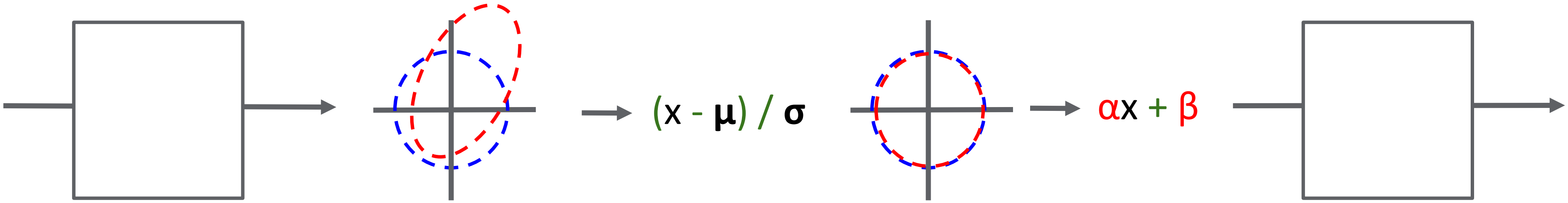
Batch Normalization

Before:



“How to train your neural network”

After:

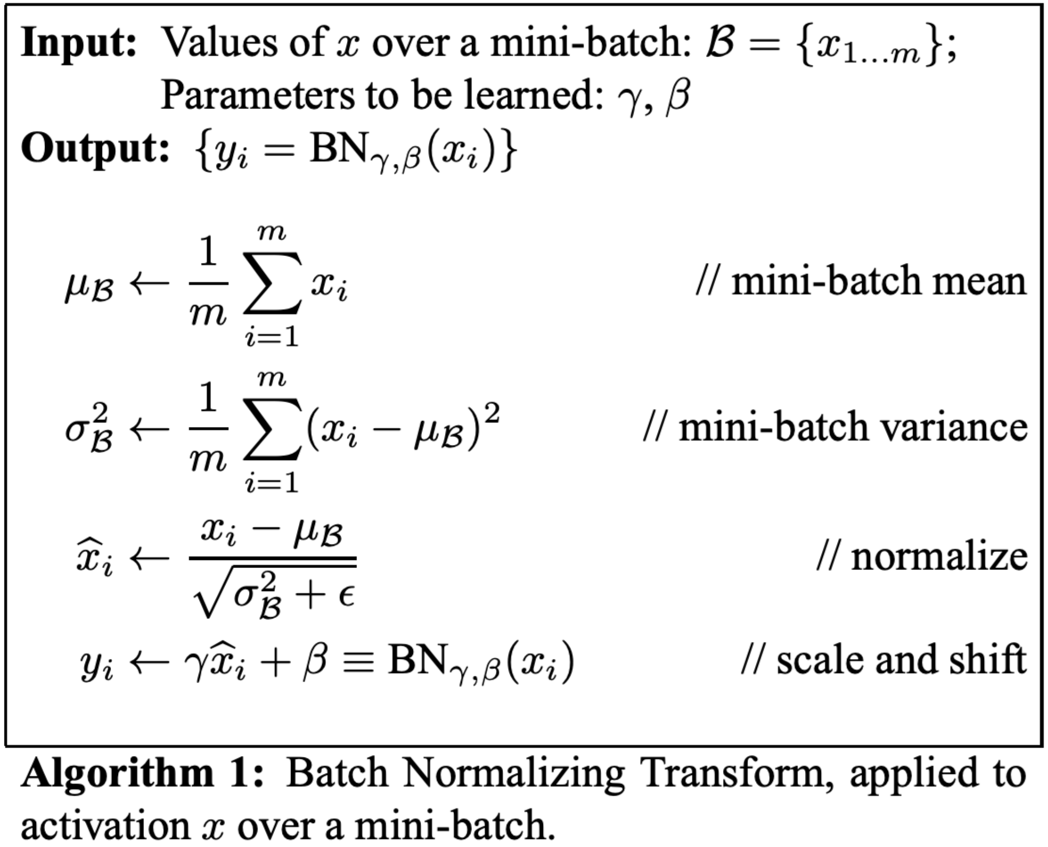


$(x - \mu) / \sigma$

$\alpha x + \beta$

Slide credits: I. Goodfellow. Deep Learning for Computer Vision, AERFI Deep Learning Autumn School, Nov 5, 2015, Valencia. Adapted from slides by Vincent Vanhoucke.

Batch Normalization



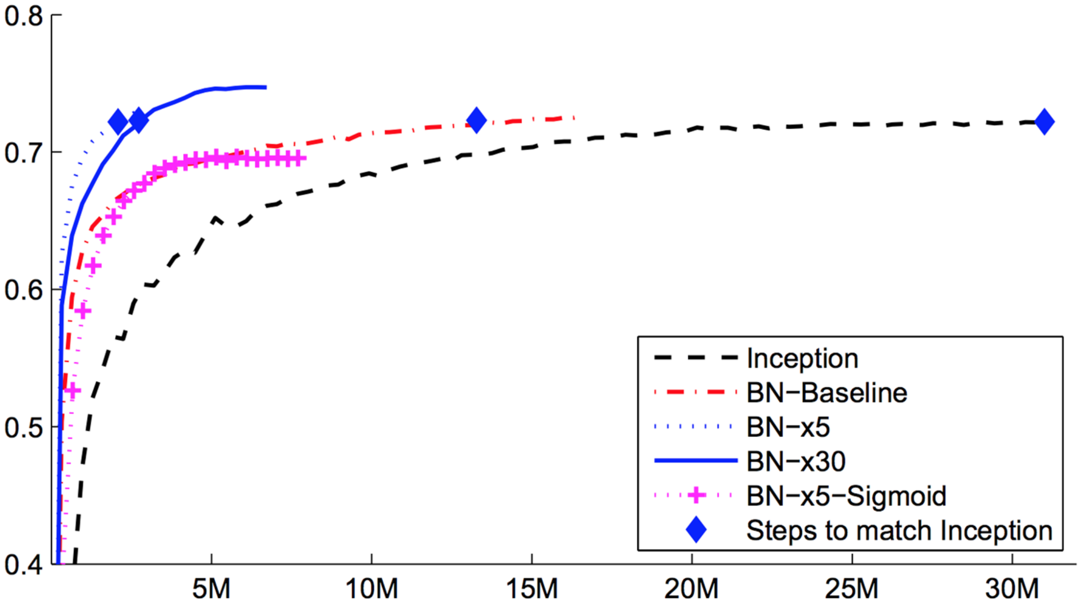
```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$      // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$   // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$     // 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

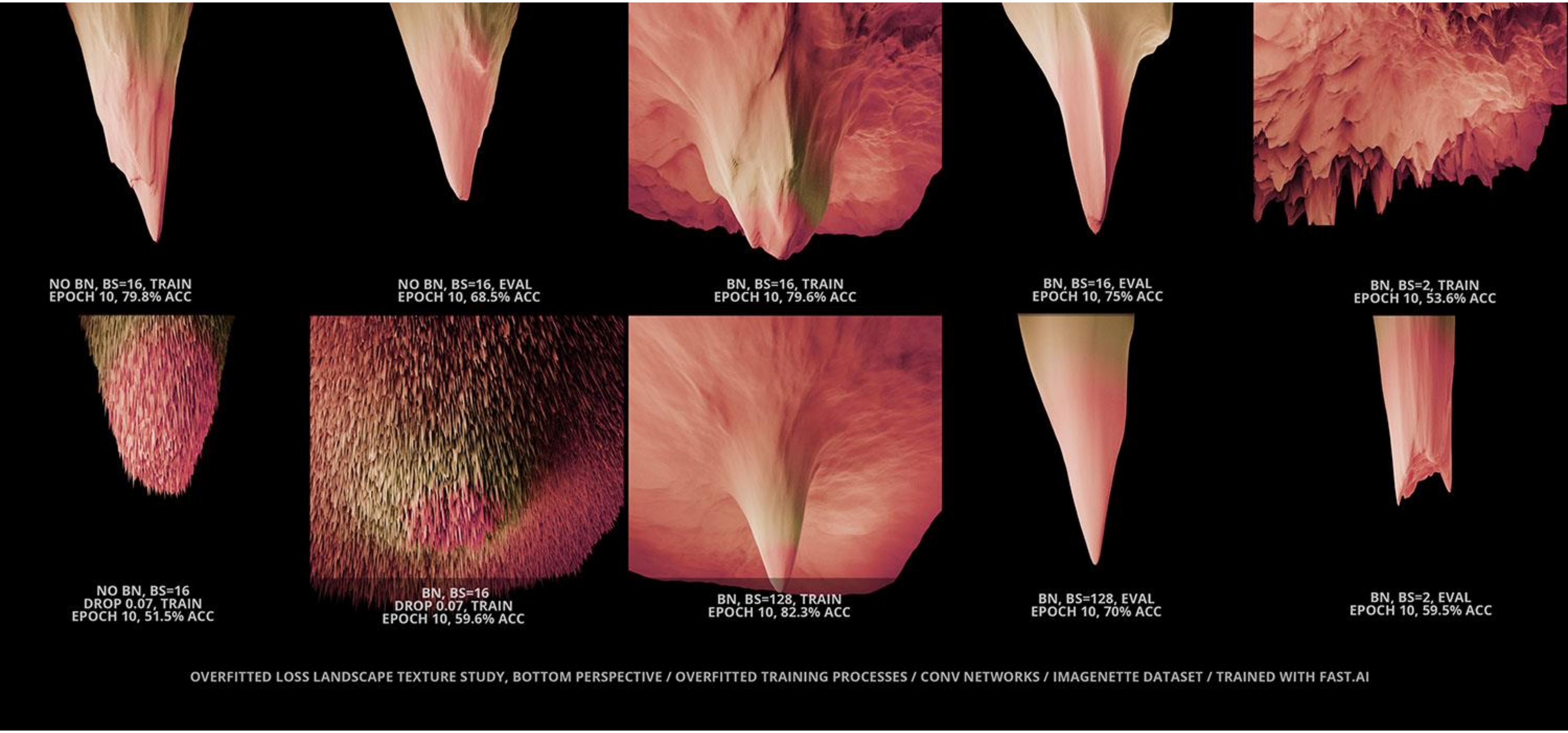
Batch Normalization

Results



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift - Sergey Ioffe, Christian Szegedy

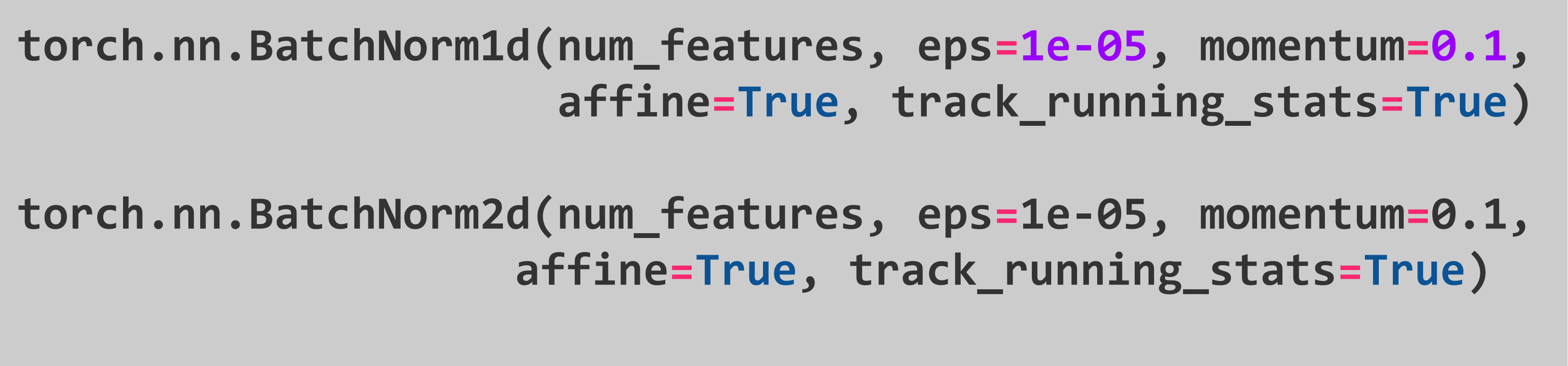
Batch Normalization



OVERFITTED LOSS LANDSCAPE TEXTURE STUDY, BOTTOM PERSPECTIVE / OVERFITTED TRAINING PROCESSES / CONV NETWORKS / IMAGENETTE DATASET / TRAINED WITH FAST.AI

Losslandscape.com

Batch Normalization in PyTorch



```
torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1,
                     affine=True, track_running_stats=True)

torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1,
                    affine=True, track_running_stats=True)
```

https://pytorch.org/docs/stable/nn.html#normalization-layers

PRACTICAL METHODOLOGY

Practical Methodology

- **Good strategy to start training a net for a new problem: find a network that works for another similar problem and start from there!**

- Hyper-parameters **search**.

- **Babysitting** the learning process

- Error analysis / debugging strategies (do we need more data? bigger model?)

Practical Methodology

Hyperparameters search

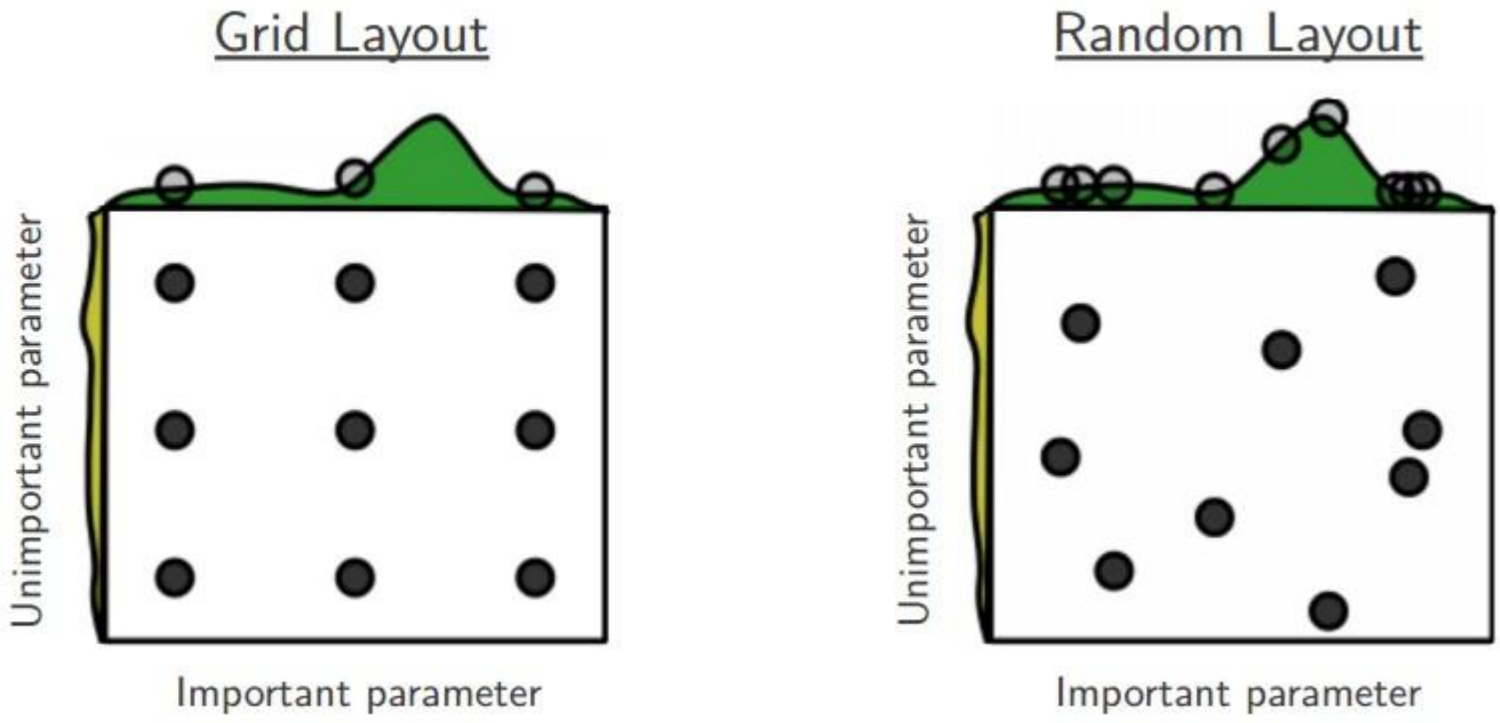
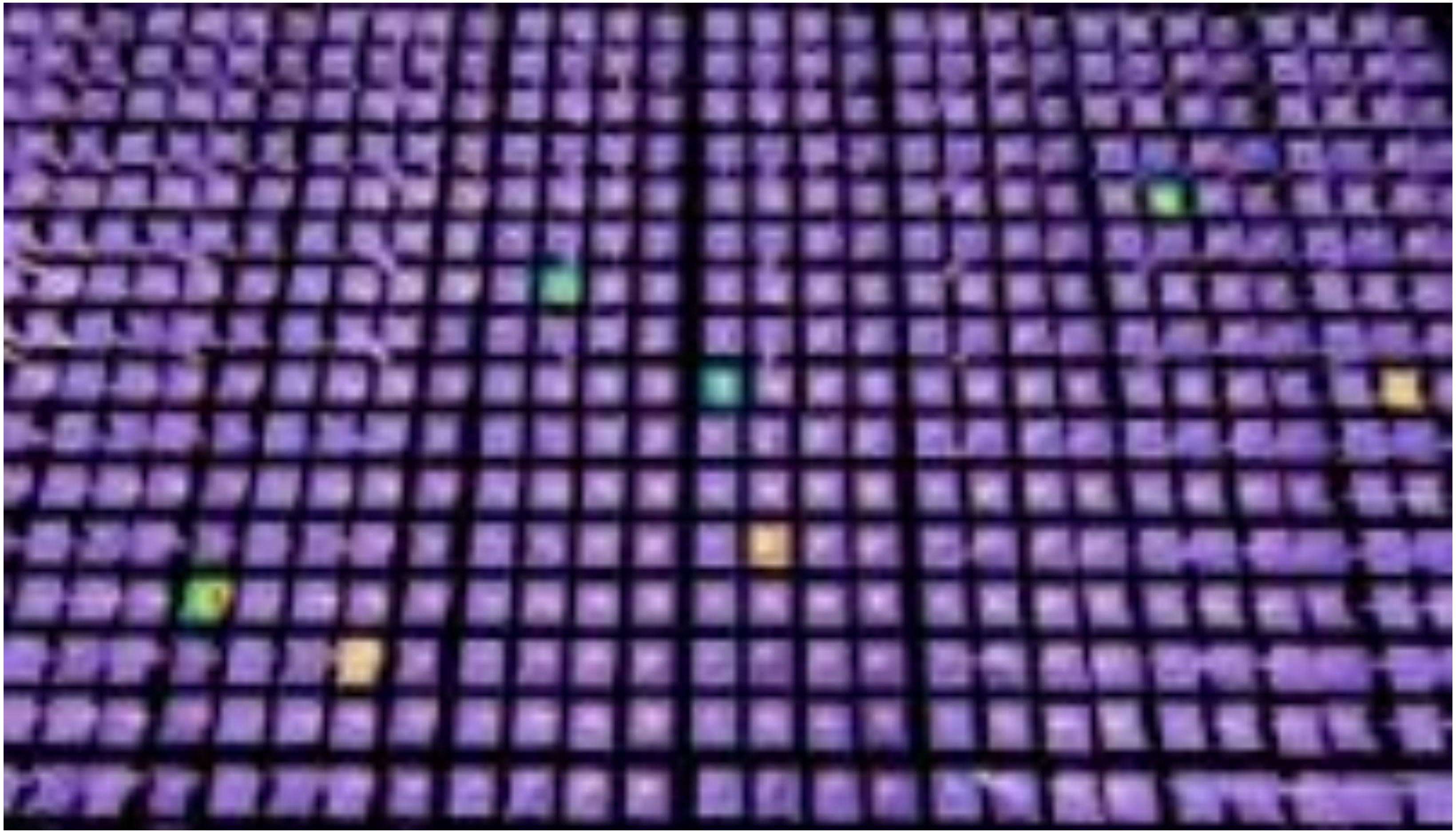


Image source: (Bergstra and Bengio, 2012).

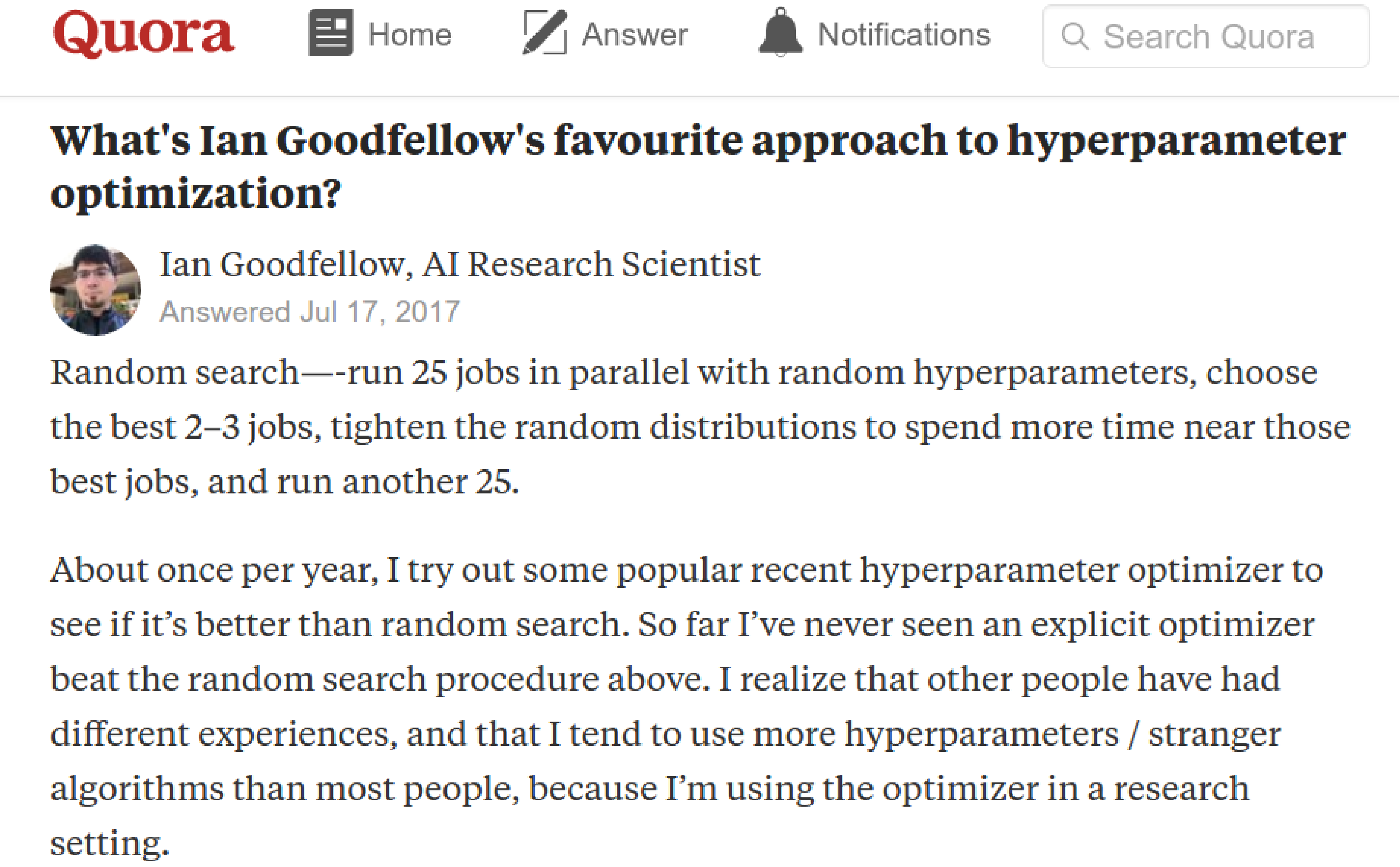
Practical Methodology

Hyperparameters search: Grid Layout



https://losslandscape.com/ https://youtu.be/GB1Ga-6tmpw

Practical Methodology



What's Ian Goodfellow's favourite approach to hyperparameter optimization?

Ian Goodfellow, AI Research Scientist
Answered Jul 17, 2017

Random search—-run 25 jobs in parallel with random hyperparameters, choose the best 2–3 jobs, tighten the random distributions to spend more time near those best jobs, and run another 25.

About once per year, I try out some popular recent hyperparameter optimizer to see if it’s better than random search. So far I’ve never seen an explicit optimizer beat the random search procedure above. I realize that other people have had different experiences, and that I tend to use more hyperparameters / stranger algorithms than most people, because I’m using the optimizer in a research setting.

Practical Methodology

Make **babysitting** of the learning process!

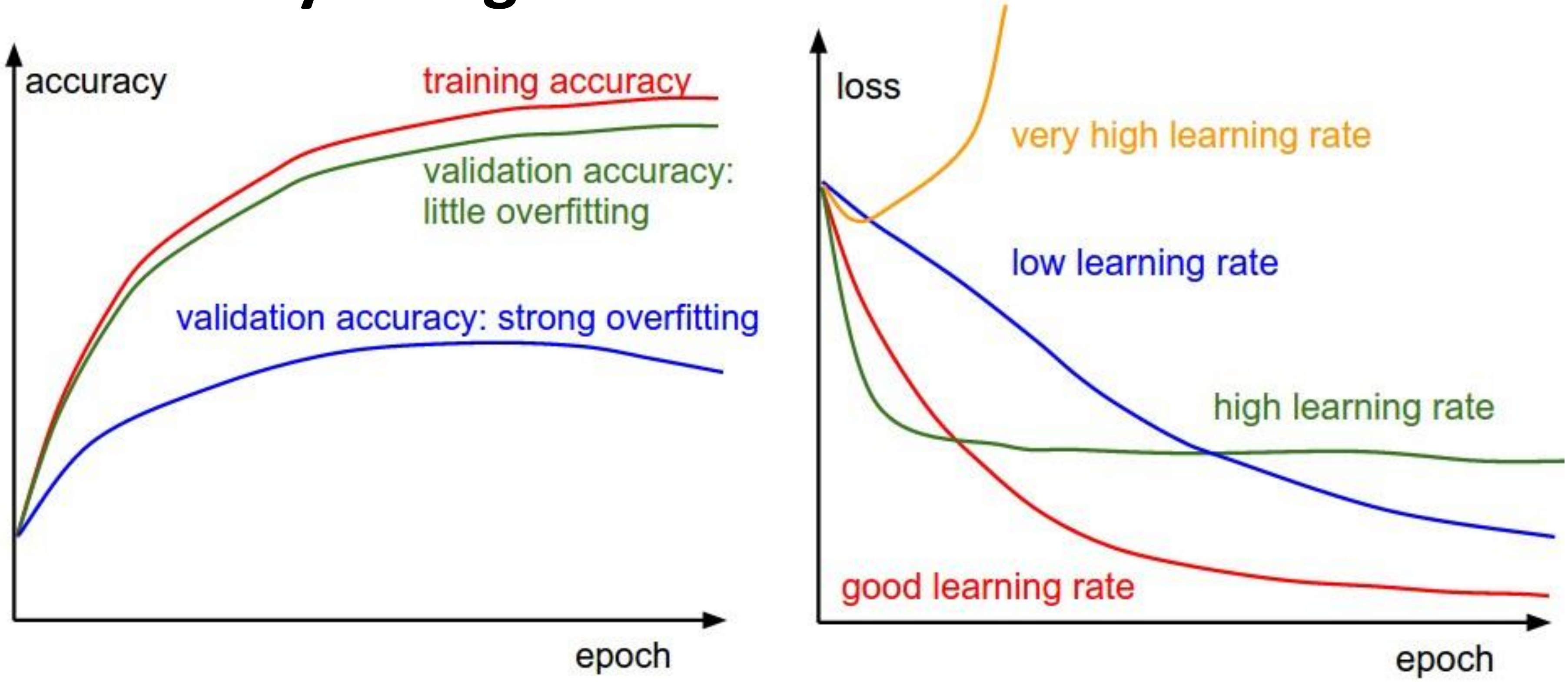


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

Image Credit: Andrej Karpathy

Practical Methodology

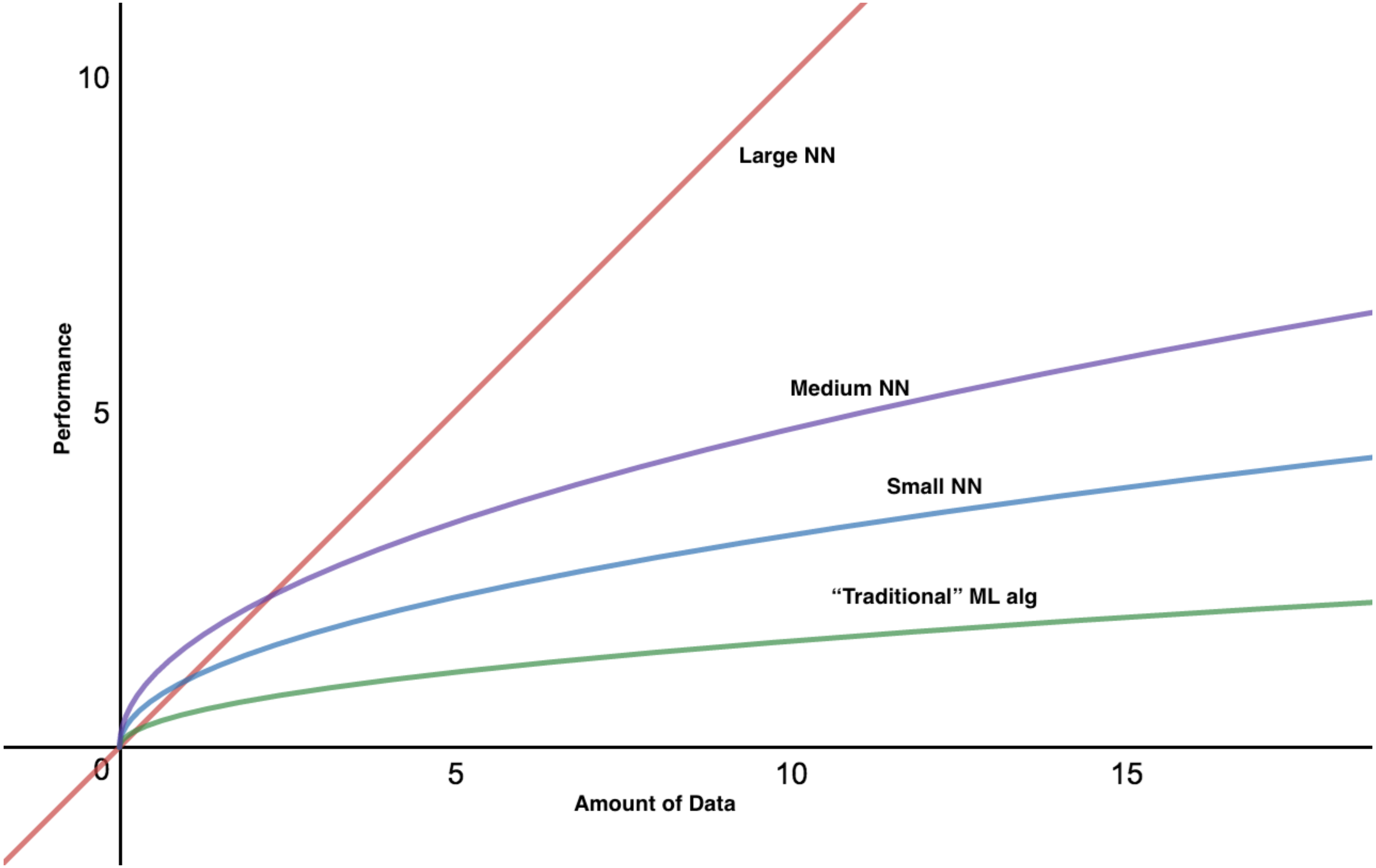
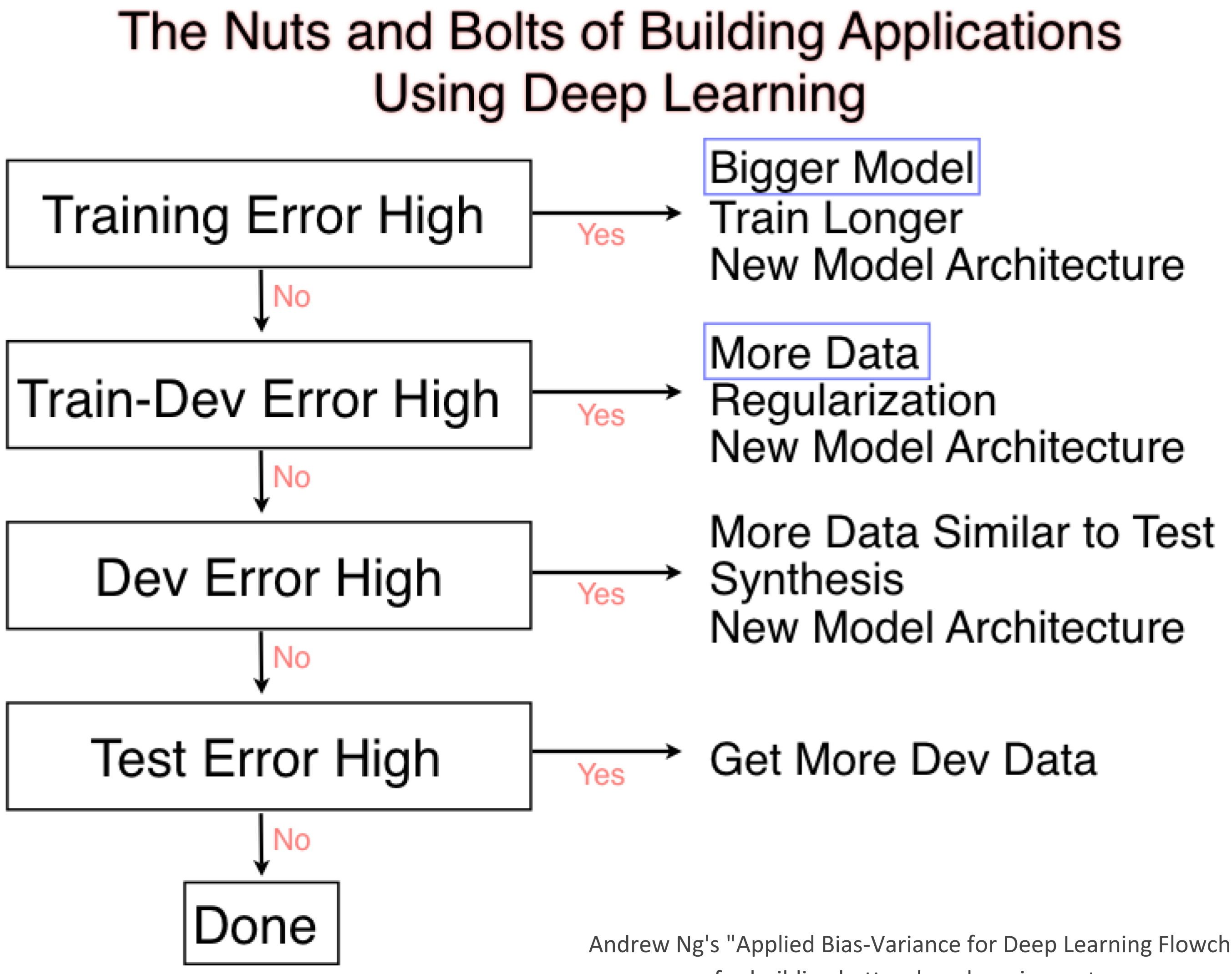


Image source: https://kevinzakka.github.io/2016/09/26/applying-deep-learning/

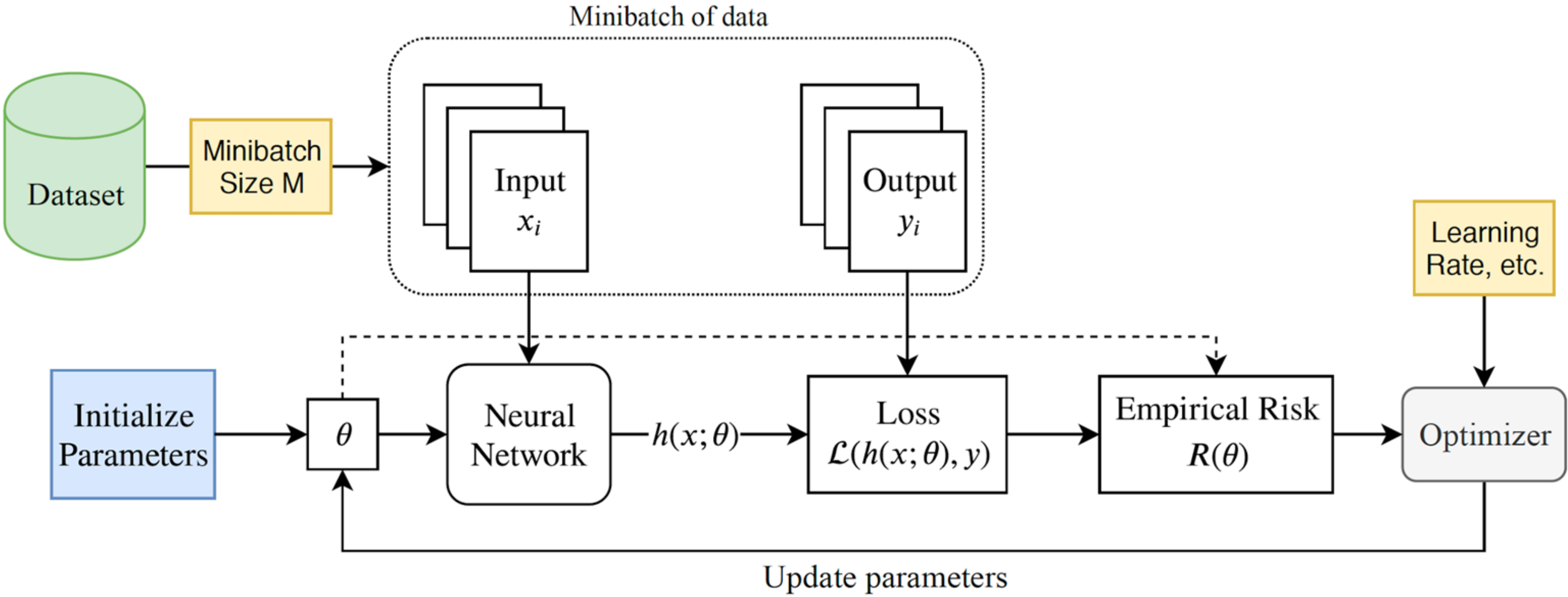
Practical Methodology



The Nuts and Bolts of Building Applications Using Deep Learning

Andrew Ng's "Applied Bias-Variance for Deep Learning Flowchart"
for building better deep learning systems.

Neural Network Training Loop



Harris Chan: *"How to train your neural network"*

Summary

- The standard procedure for training deep models puts together a set of clever optimization techniques that have been developed in recent years.
- Without some of those bits (ReLU, Momentum, Adaptive Learning Rate, Data Preprocessing, Weights Initialization, etc.) it would be almost impossible to train deep models in practice due to vanishing or exploding gradients.
- Still, there are a lot of hyperparameters to set!
- In many cases this is more an art than a science!
- **The only way to develop good intuitions is through practice.**

References

- (Gori and Tesi 1992) “On the problem of local minima in backpropagation.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- (Goodfellow et al. 2015) Goodfellow et al, “Qualitatively Characterizing Neural Network Optimization Problems”, ICLR 2015.
- (Goodfellow et al. 2016) “Deep Learning”, MIT Press 2016.
- (Hochreiter et al. 2001) “Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies.”
- (Nielsen 2015) “Neural Networks and Deep Learning”, Determination Press.
- (Polyak, 1964) "Some methods of speeding up the convergence of iteration methods." *USSR Computational Mathematics and Mathematical Physics.*
- (Sutskever et al., 2013) "On the importance of initialization and momentum in deep learning." *International conference on machine learning*. 2013.
- (Xavier Glorot and Yoshua Bengio 2010) "Understanding the difficulty of training deep feedforward neural networks." *International Conference on Artificial Intelligence and Statistics*.

References

- (He at al 2015) “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, CVPR.
- (Duchi et al. 2011) "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of Machine Learning Research*.
- (Hinton, 2012) “Neural Networks for Machine Learning”, Coursera video lectures.
- (Kingma and Ba, 2014) "Adam: A method for stochastic optimization."
- (Schaul et al. 2014) "Unit tests for stochastic optimization."
- (Ioffe and Szegedy 2015) "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *International Conference on Machine Learning*.
- (Srivastava et al. 2014) "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research.*
- (Bergstra and Bengio 2012) “Random Search for Hyper-Parameter Optimization.” *Journal of machine learning research.*