Training Deep Nets:

Data Preprocessing, Weight Initialization, Gradient Optimisation

Lluis Gomez i Bigorda <lgomez@cvc.uab.cat>

Computer Vision Center, Universitat Autònoma de Barcelona.

Training Deep Nets: Outline

- Introduction
- Stochastic Gradient Descent (Review)
- Data Preprocessing
- Weight Initialization
- Algorithms with Adaptive Learning Rates
- Other optimization strategies
- Practical Methodology

Introduction

Training deep neural networks is a complex and expensive task!

DATA

Millions of images

Labeled data!

COMPUTING RESSOURCES

GPUs

HUMAN RESSOURCES

Deep Learning experts

Network architecture

Optimization problem

TIME

Research

Computing

Image source: Yannis Ghazouani, 2016.

Introduction

- Find the parameters $\boldsymbol{\theta}$ of a neural network that significantly reduce a **cost function** $\boldsymbol{J(\theta)}$.

- $\boldsymbol{J(\theta)}$ typically includes a **performance measure** as well as **regularization terms**.

- Typically the cost function can be written as an average over the training set:

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

Introduction: Risk minimization

- Our **final goal is to reduce the expected generalization error** given by the corresponding objective function where the expectation is taken **across the data-generating distribution** p_{data} rather than just over the finite training set.

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

- But we do not know p_{data}!

Introduction: Empirical risk minimization

- We optimize the **empirical risk** and hope that the risk decreases as well!

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

$$\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})$$

Introduction: Surrogate Loss Functions

Empirical risk minimization has two main problems:

- It is prone to **overfitting**. Models with high capacity can simply memorize the training set!

- Many useful loss functions, such as the 0-1 loss, have **no useful derivatives**.

Introduction: Surrogate Loss Functions

For example, for classification the **negative log-likelihood** (a.k.a cross-entropy) of the correct class is typically used as a **surrogate** of the 0-1 loss.

$$L(f(x;\theta), y) = -\sum_{c} 1_{y=c}\ log\ f(x)_c = -log\ f(x)_y$$

It allows us to estimate the conditional probabilities of the classes given the input. If the model do it well then it can pick the classes that yield the least classification error in expectation.

Introduction: Batch and Minibatch algorithms

Computing the gradient of the objective function $\boldsymbol{J}$ exactly is very expensive because it requires evaluating the model on every example of the entire training set.

$$\nabla_\theta J(\theta) = \mathbb{E}_{x,y\ \hat{p}_{data}} \nabla_\theta\ log\ p_{model}(x, y; \theta)$$

In practice we can compute these expectations by randomly sampling a **small number of examples from the dataset**, then taking the average over only those examples.

Introduction: Batch and Minibatch algorithms

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

- Optimization algorithms that use only a single example at a time are called **stochastic** and sometimes **online** methods.

- Most algorithms used for deep learning fall somewhere in between, using more than one but fewer than all the training examples. These were traditionally called **minibatch** or **minibatch stochastic** methods, and it is now common to call them simply **stochastic** methods.

Minibatch based updates:

- Are much faster to compute.
- Offer a fair approximations of the true gradient.
- Exploit redundancy on the training set.

Introduction: Batch and Minibatch algorithms

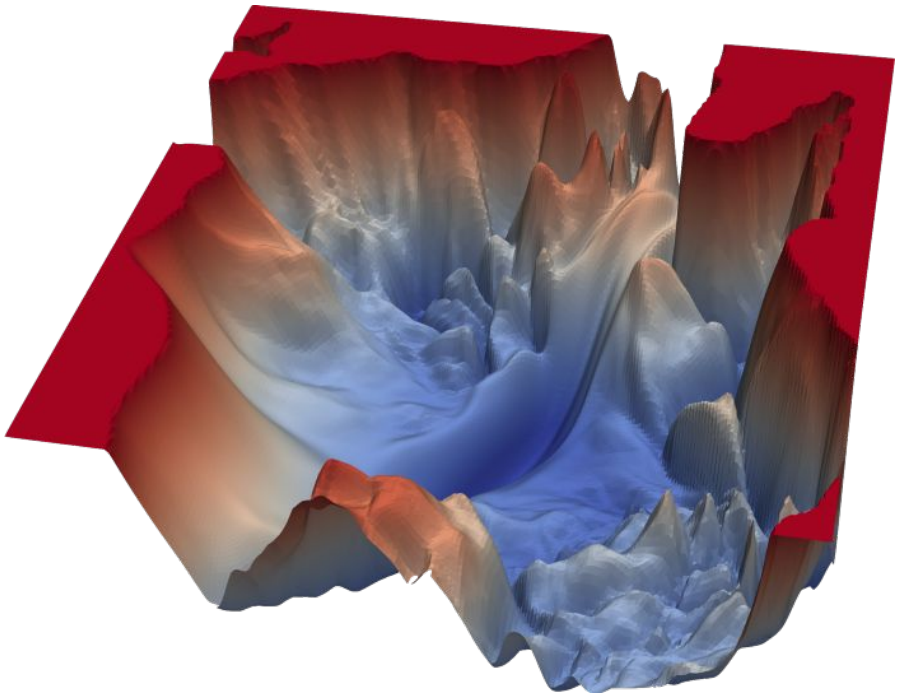
Minibatch sizes are generally driven by:

- Large batches might provide a more accurate estimate of the gradient.
- Memory usage scales with the batch size. (*)
- Small batches can offer a regularizing effect.
- Multicore architectures are usually underutilized by extremely small batches.
- Some hardware architectures scale better for specific batch sizes.

IMPORTANT: It is crucial that the minibatches are selected randomly. Sample independence, and consecutive batch independence. **Data shuffle**!

Introduction: Why are Neural Nets hard to train?

Training neural networks requires minimizing a high-dimensional non-convex loss function.



- Local minima, saddle points, flat regions...
- Stepsize issue and monotonicity.
- Vanishing/exploding gradients.
- Unstable gradients.
- Inexact gradients.
- Local vs. global structure.
- …

Figure source: Li, Hao, et al. "Visualizing the loss landscape of neural nets." *NeurIPS*. 2018.

Explore similar visualizations: https://losslandscape.com/

Introduction: Why are Neural Nets hard to train?

Local minima

Contrary to convex optimization, where any local minima is guaranteed to be a global minima, with **non-convex functions** (such as deep nets) it is possible to **have many local minima**.

If local minima with high cost are common this could pose a serious problem gradient based optimization algorithms. (Gori and Tesi 1992)

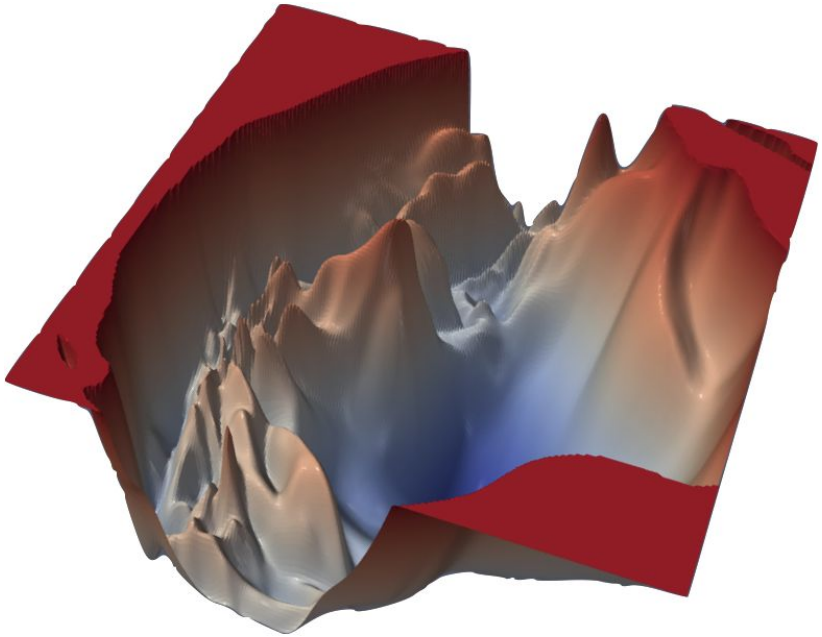
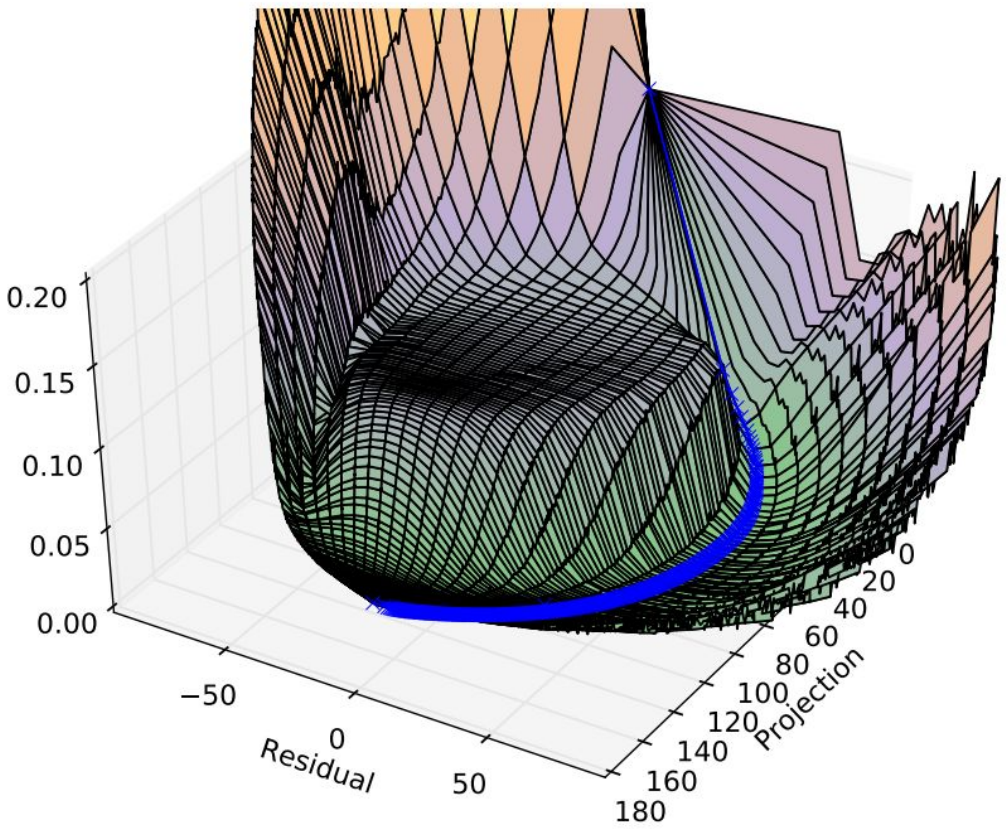


Figure source: Li, Hao, et al. "Visualizing the loss landscape of neural nets." *NeurIPS*. 2018.

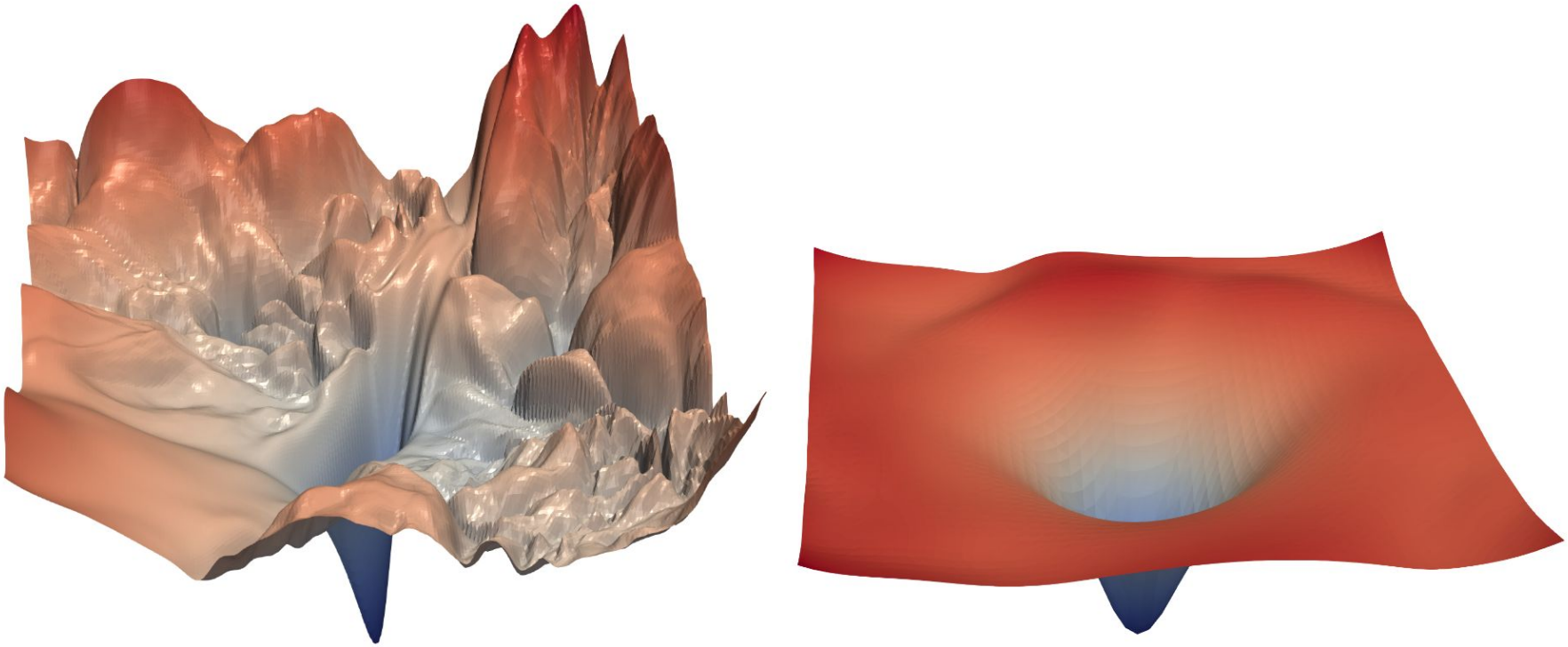
Introduction: Why are Neural Nets hard to train?

Local minima



Goodfellow et al, “Qualitatively Characterizing Neural Network Optimization Problems”, ICLR 2015.

Introduction: Why are Neural Nets hard to train?



(a) without skip connections

(b) with skip connections

The loss surfaces of ResNet-56 with/without skip connections.

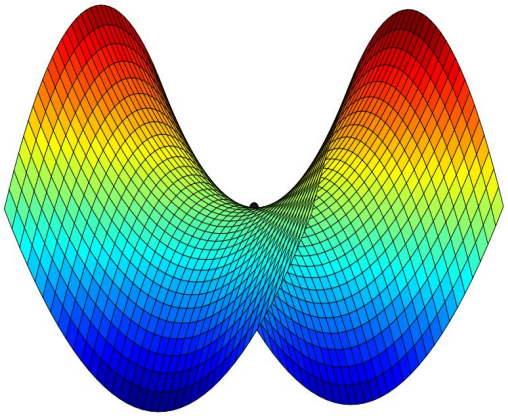
Figure source: Li, Hao, et al. "Visualizing the loss landscape of neural nets." *NeurIPS*. 2018.

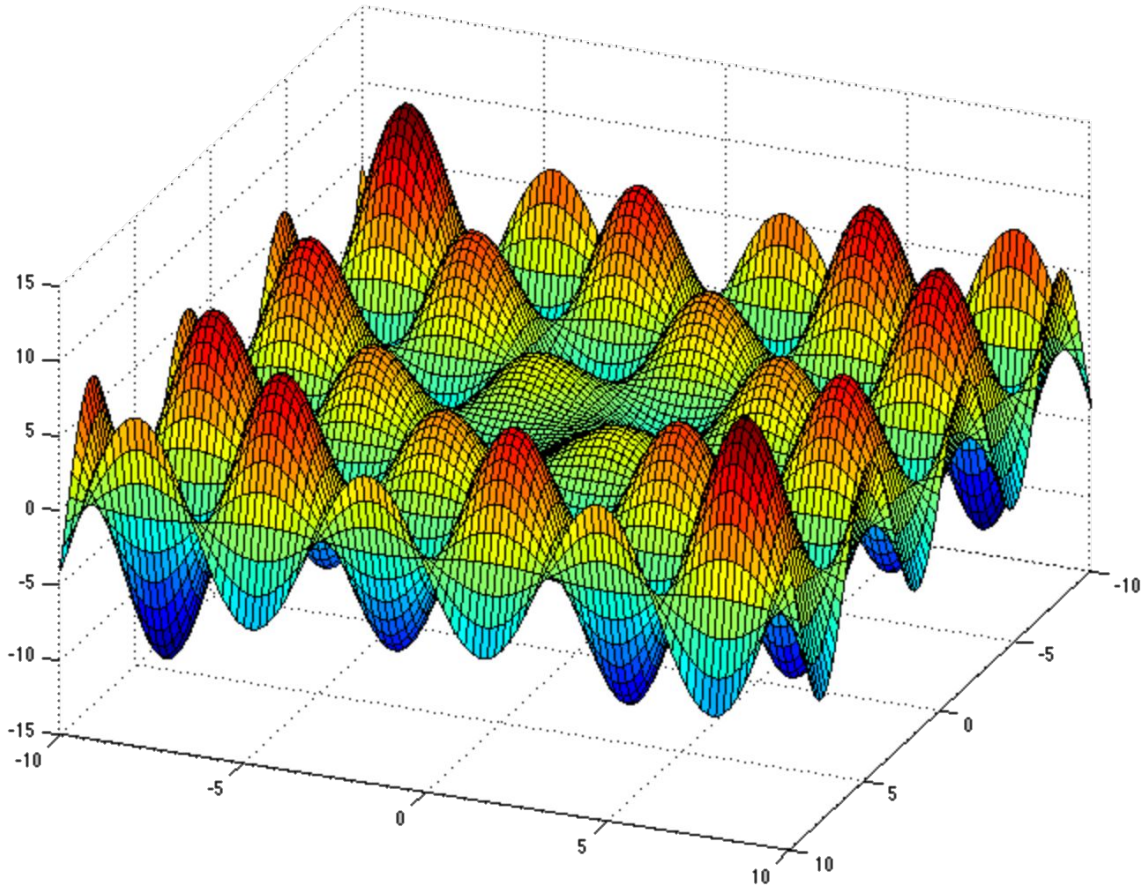
Introduction: Why are Neural Nets hard to train?

Local minima

Saddle points

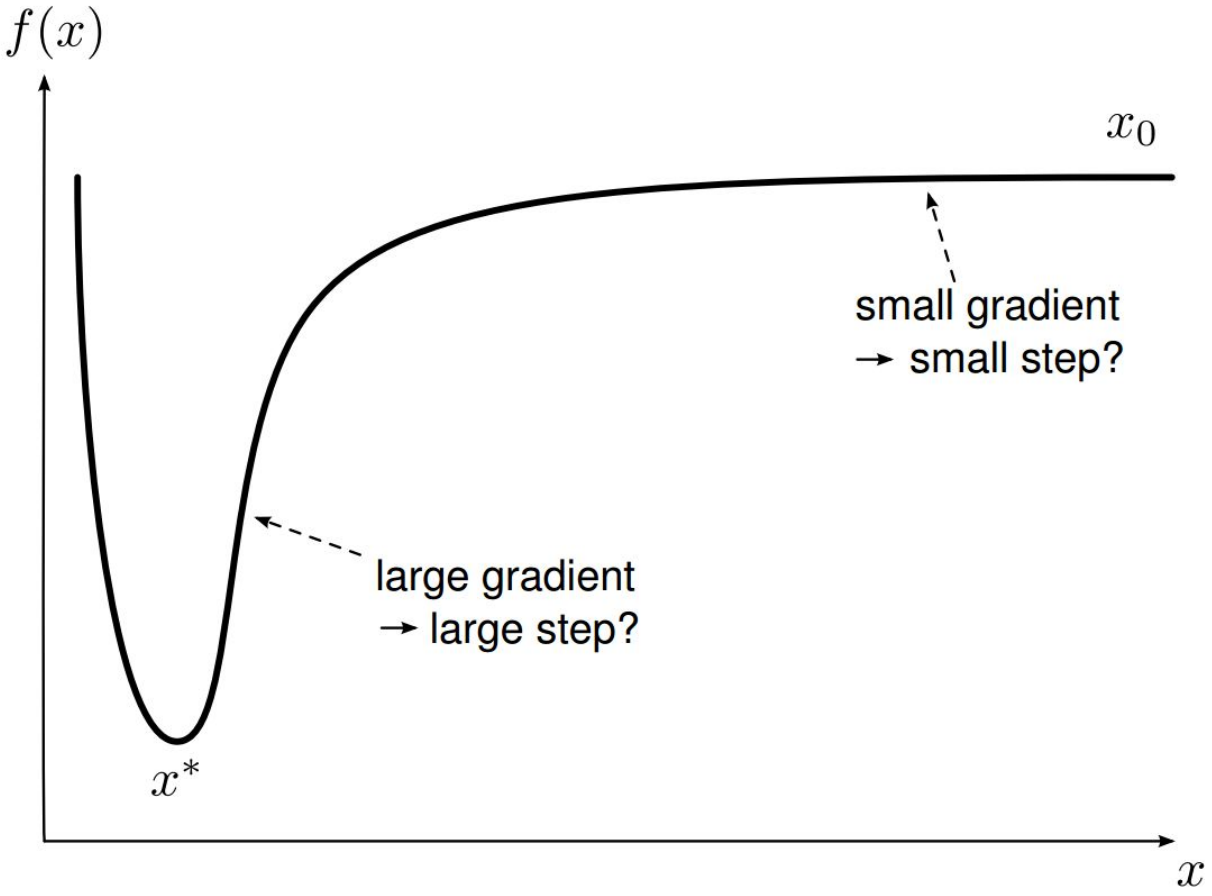
Flat regions





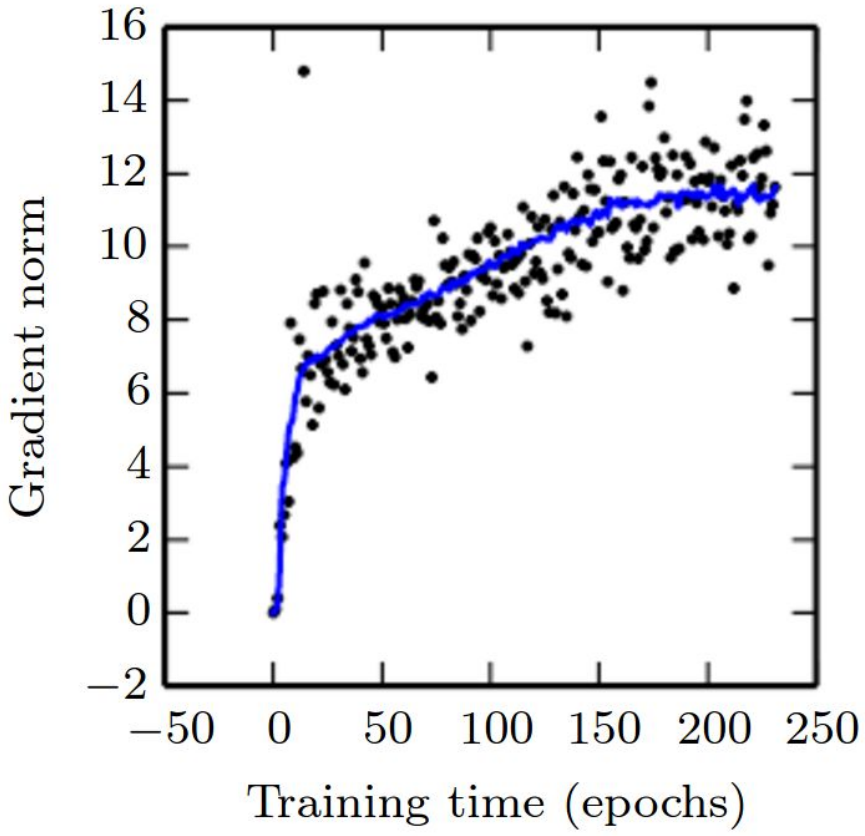
Introduction: Why are Neural Nets hard to train?

Stepsize issue and monotonicity



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

Introduction: Why are Neural Nets hard to train?



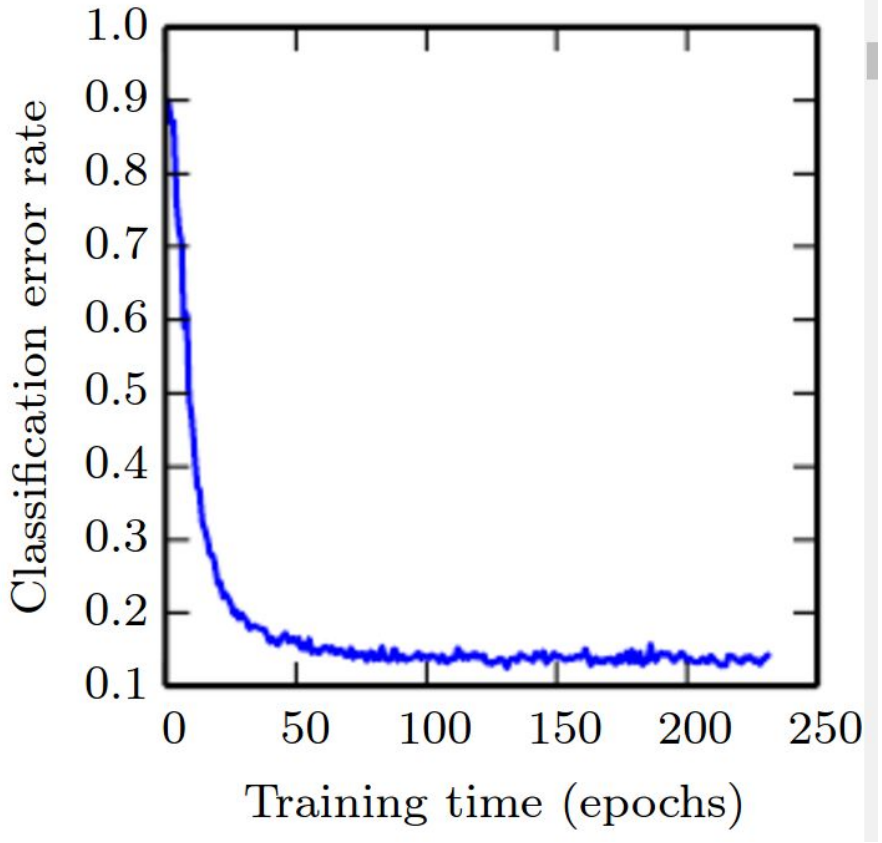


Image source: Goodfellow et al. Deep Learning. MIT Press 2016.

Introduction: Why are Neural Nets hard to train?

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. Workaround: **Gradient clipping heuristic.**

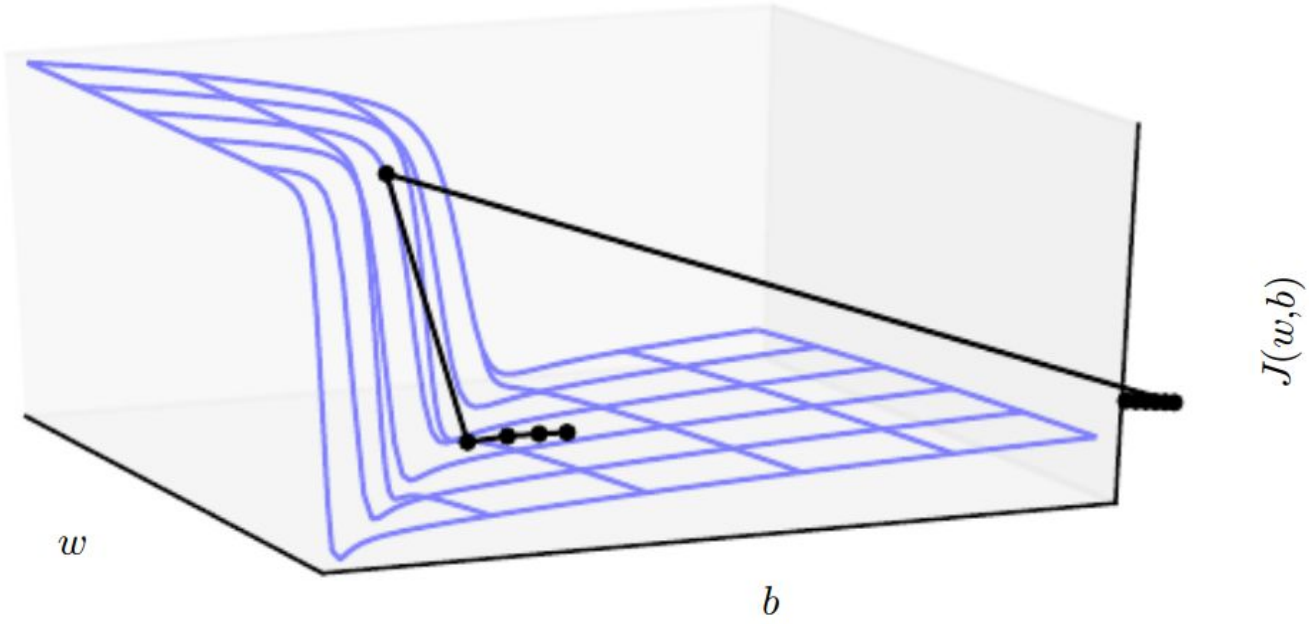


Image source: Goodfellow et al. Deep Learning. MIT Press 2016.

Introduction: Why are Neural Nets hard to train?

Gradient Clipping in Keras

```
from keras import optimizers

# All parameter gradients will be clipped to
# a maximum norm of 1.
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)

# All parameter gradients will be clipped to
# a maximum value of 0.5 and
# a minimum value of -0.5.
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)
```

https://keras.io/optimizers/

Introduction: Why are Neural Nets hard to train?

Long-Term dependencies

The gradient tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers. **Vanishing gradient problem**. (Hochreiter at al. 2001)

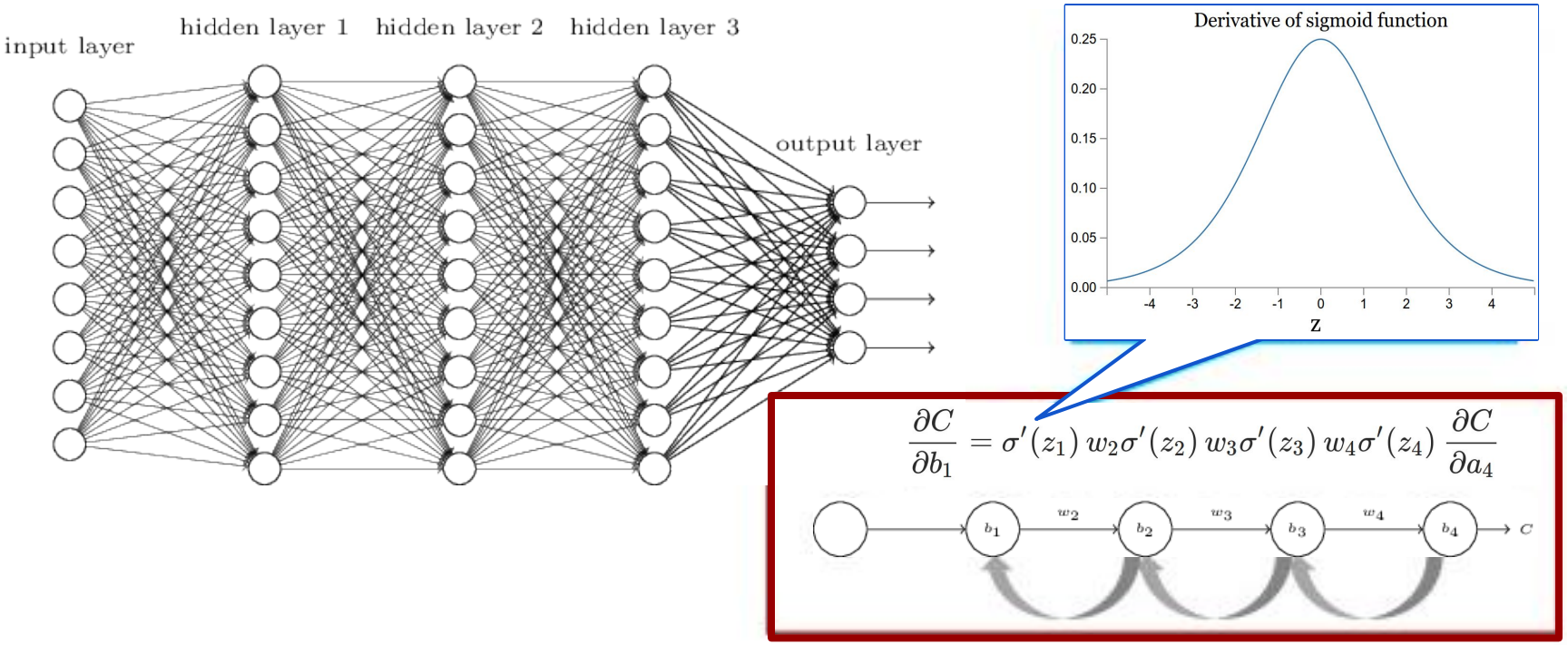


Image source: Michael A. Nielsen. Neural Networks and Deep Learning. Determination Press, 2015

Introduction: Why are Neural Nets hard to train?

Long-Term dependencies

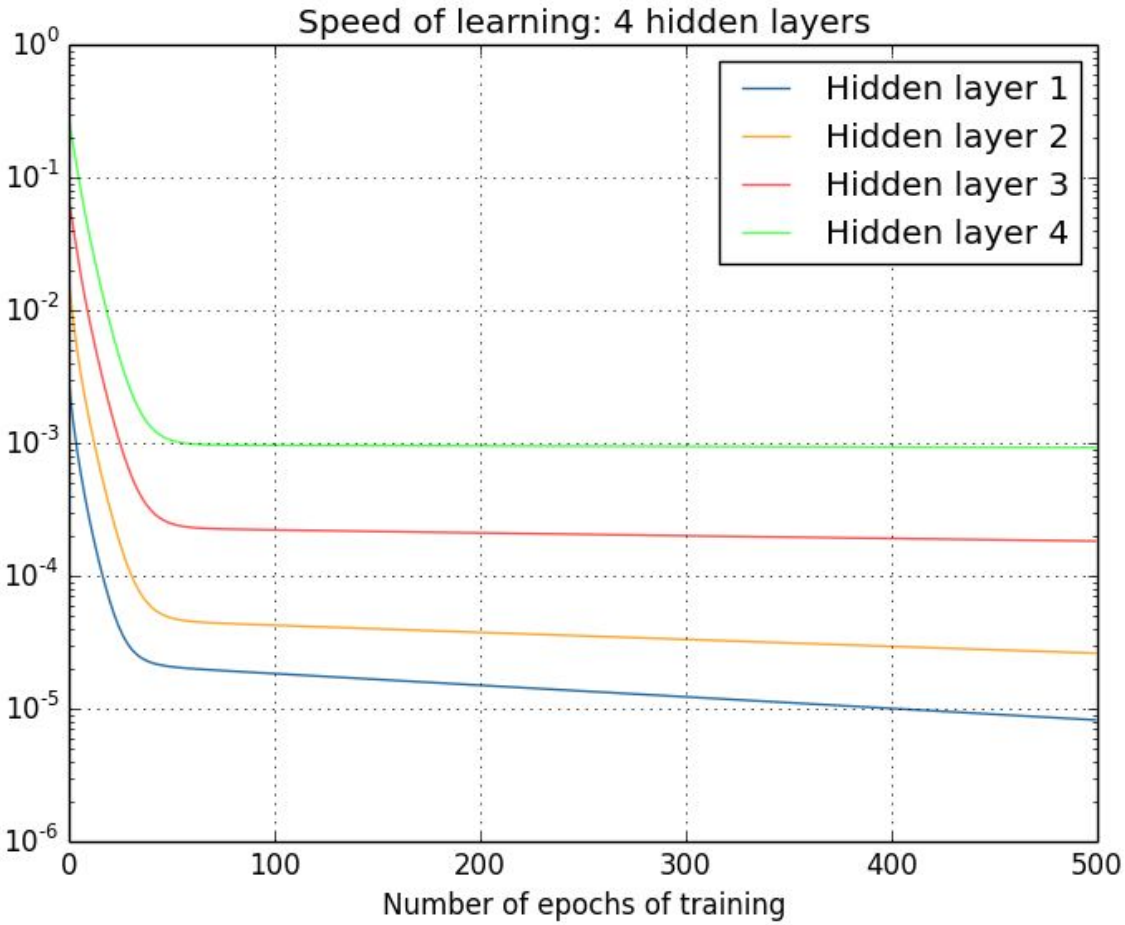


Image source: Michael A. Nielsen. Neural Networks and Deep Learning. Determination Press, 2015

Introduction: Why are Neural Nets hard to train?

Inexact Gradients

Optimization algorithms assume we have the exact gradients, but in practice we have only a **noisy or biased estimate when using a minibatch of training examples** to compute them.

The loss landscape changes at each iteration!

Introduction: Why are Neural Nets hard to train?

Local vs. Global structure

Sometimes the most improvement locally does not point toward regions with lower global cost.

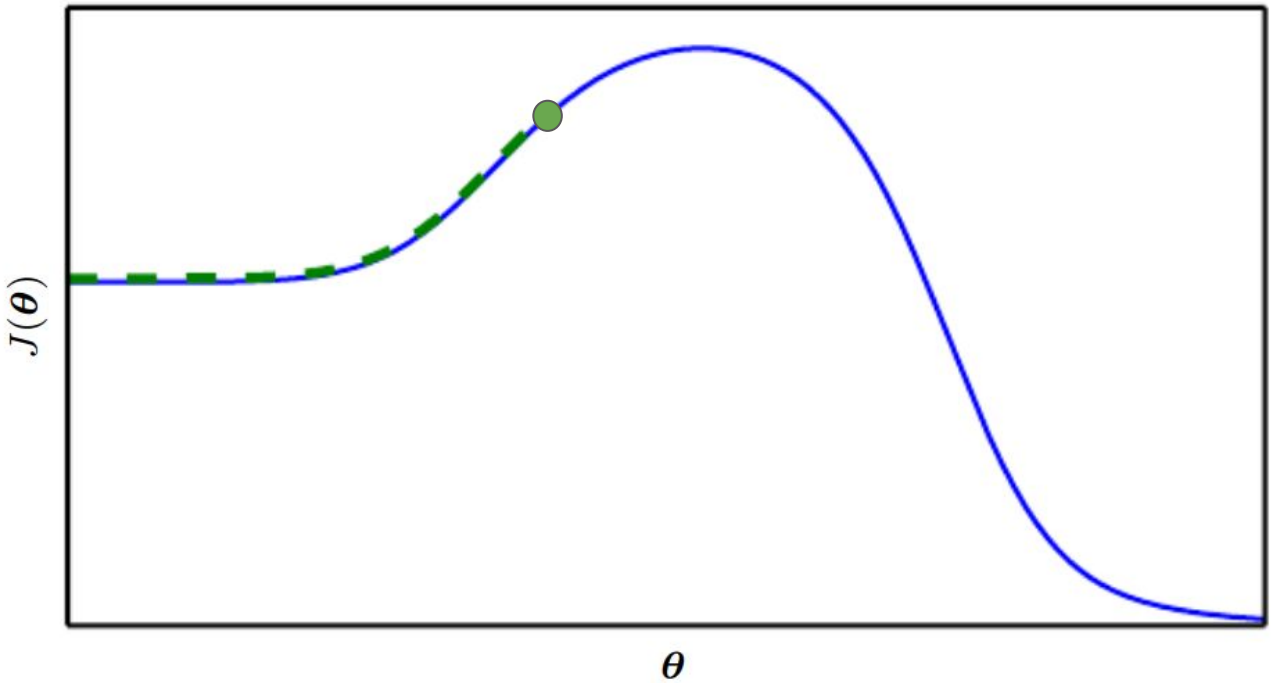
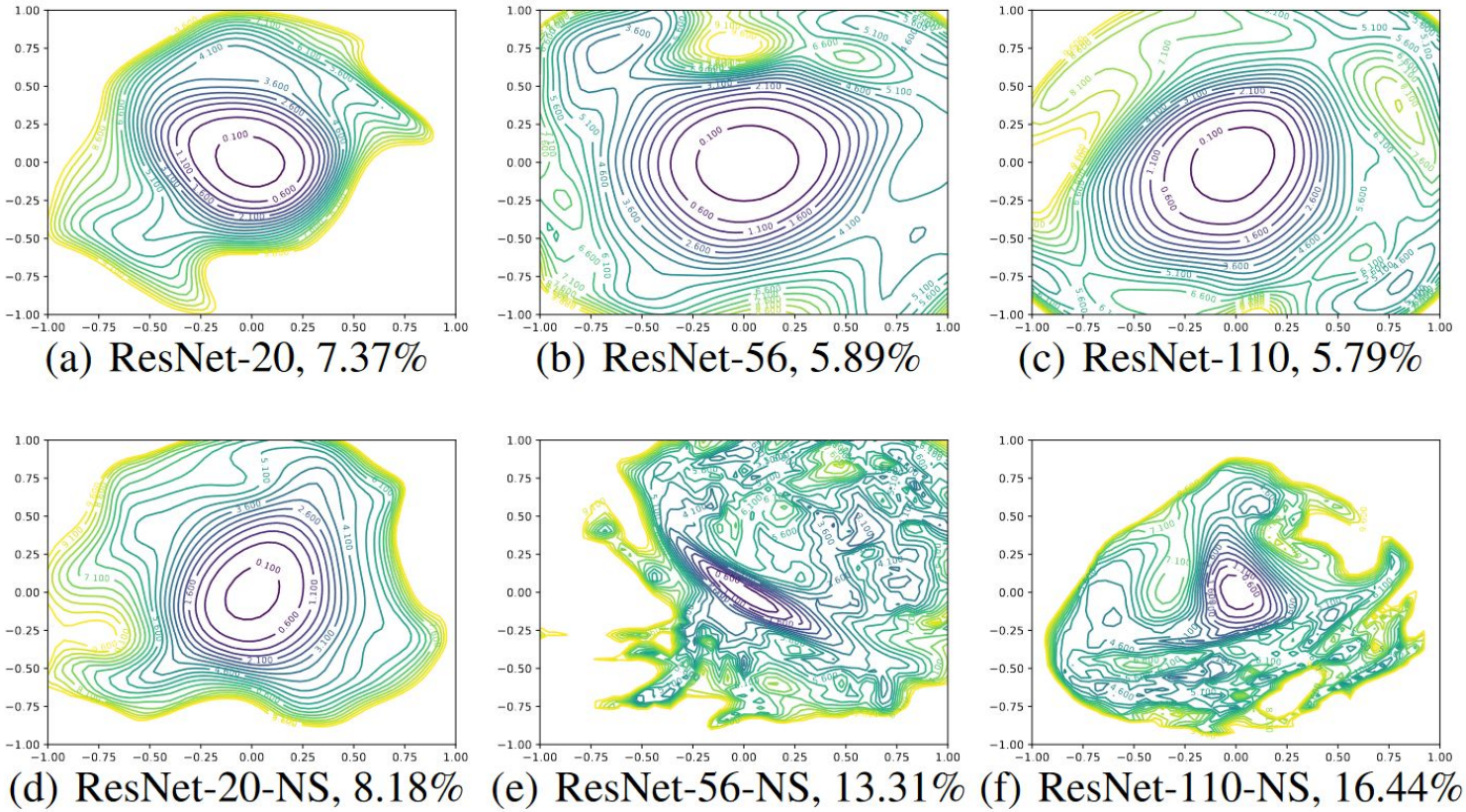


Image source: Goodfellow et al. Deep Learning. MIT Press 2016.

Introduction: Why are Neural Nets hard to train?



(a) ResNet-20, 7.37% (b) ResNet-56, 5.89% (c) ResNet-110, 5.79%

(d) ResNet-20-NS, 8.18% (e) ResNet-56-NS, 13.31% (f) ResNet-110-NS, 16.44%

2D visualization of the loss surface of ResNet and ResNet-noshort with different depth.

Figure source: Li, Hao, et al. "Visualizing the loss landscape of neural nets." *NeurIPS*. 2018.

Training Deep Nets: Outline

- Introduction
- **Stochastic Gradient Descent (Review)**
- Data Preprocessing
- Weight Initialization
- Algorithms with Adaptive Learning Rates
- Other optimization strategies
- Practical Methodology

Review: Stochastic Gradient Descent (SGD)

Algorithm: SGD update at training iteration k

Require: Learning rate ϵ_k
Require: Initial parameter $\boldsymbol{\theta}$

while stopping criterion not met **do**

Sample a minibatch of m samples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{g} \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

Apply update: $\theta \leftarrow \theta - \epsilon\hat{g}$

end while

Review: Stochastic Gradient Descent (SGD)

A crucial parameter for SGD is the learning rate ϵ.

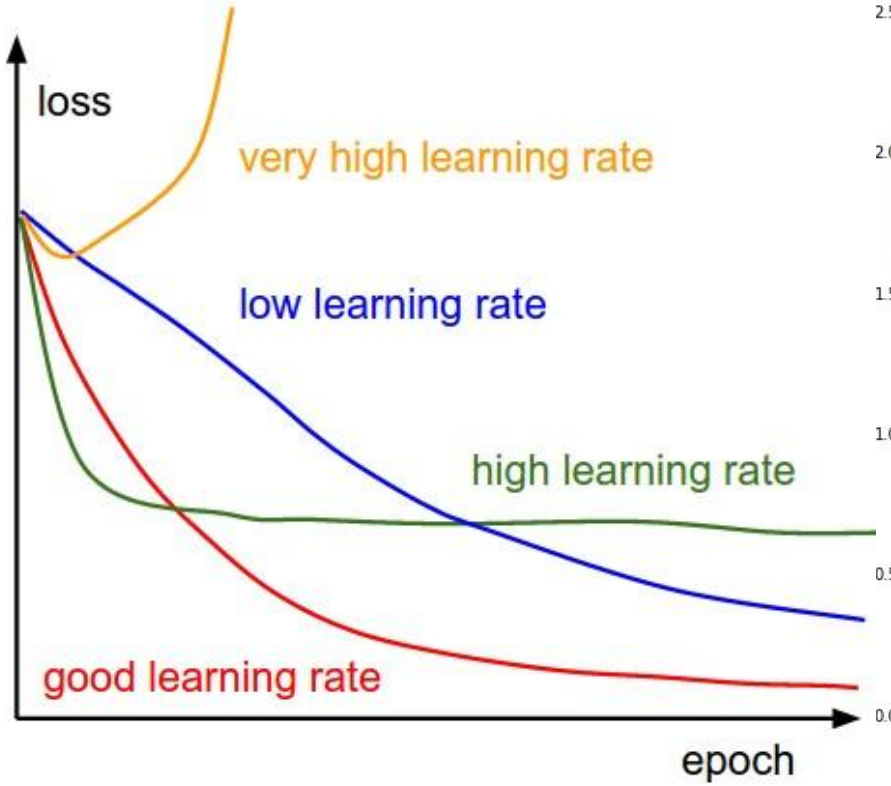
The standard SGD uses a fixed learning rate, but in practice it is necessary to decrease it over time. Learning rate at iteration k : ϵ_k

It is common to **decay** ϵ_k linearly until iteration τ

$$\epsilon_k = (1 - \frac{k}{\tau})\epsilon_0 + \frac{k}{\tau}\epsilon_\tau$$

Another common strategy when training deep models is to drop the learning rate in "steps" by a factor of gamma every stepsize iterations.

Review: Stochastic Gradient Descent (SGD)



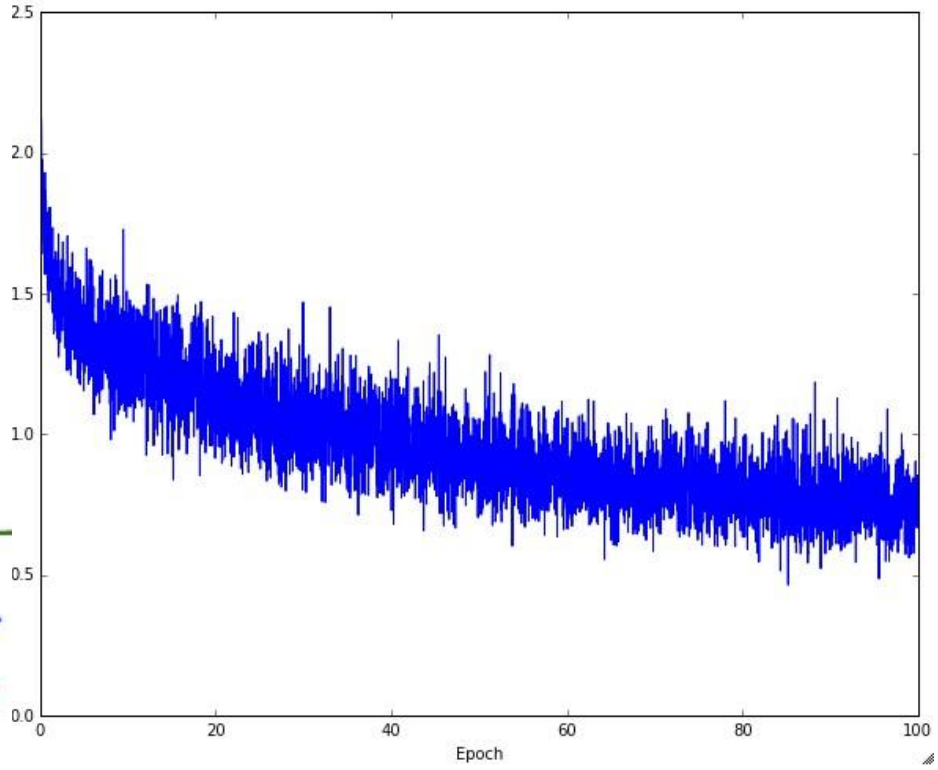


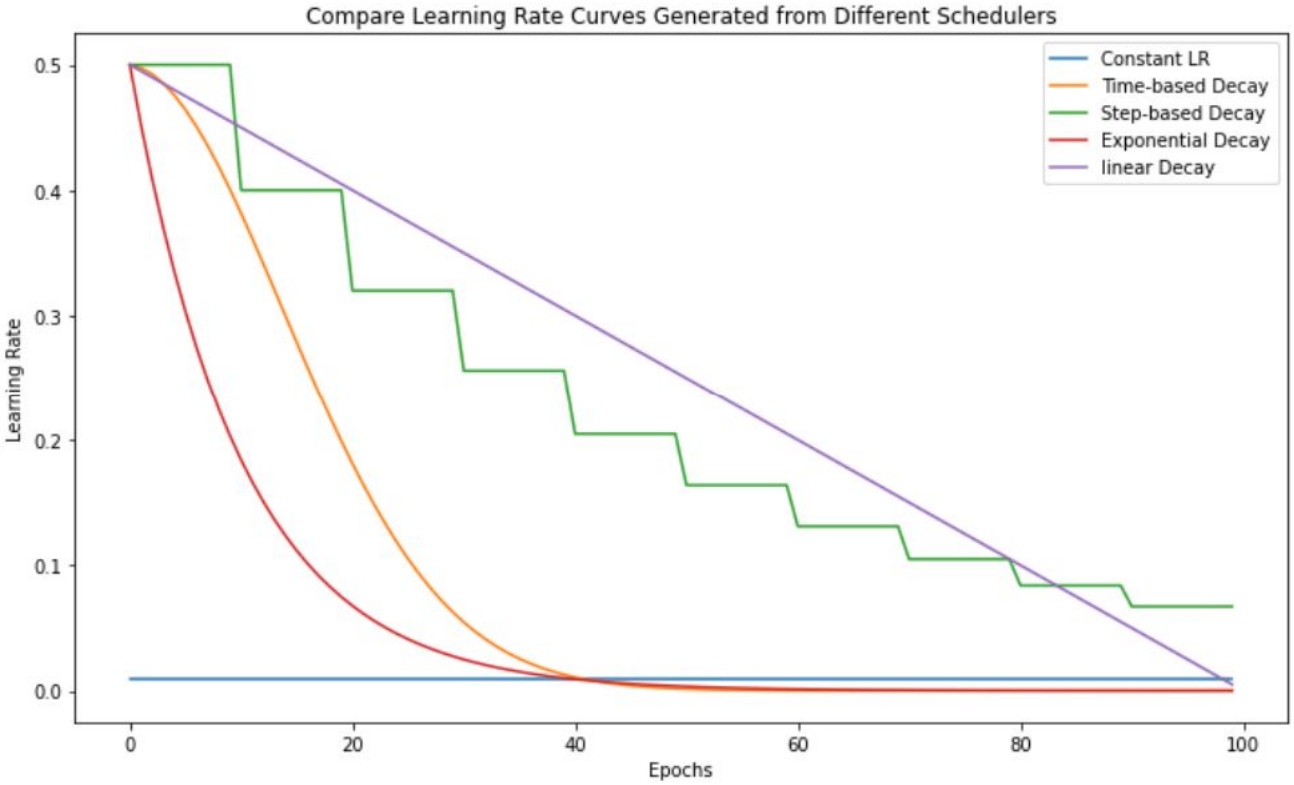
Image source: Stanford CS class [CS231n: Convolutional Neural Networks for Visual Recognition](#).

Learning Rate Scheduler in Keras

```
from keras import callbacks

def my_scheduler(epoch):
    if epoch == 5:
        model.lr.set_value(.02)
    return model.lr.get_value()

change_lr = callbacks.LearningRateScheduler(my_scheduler)

model.fit(X, Y, nb_epoch=10, batch_size = batch_size,
          show_accuracy=True, callbacks=[chage_lr])
```

https://keras.io/optimizers/

Common Learning Rate Schedulers

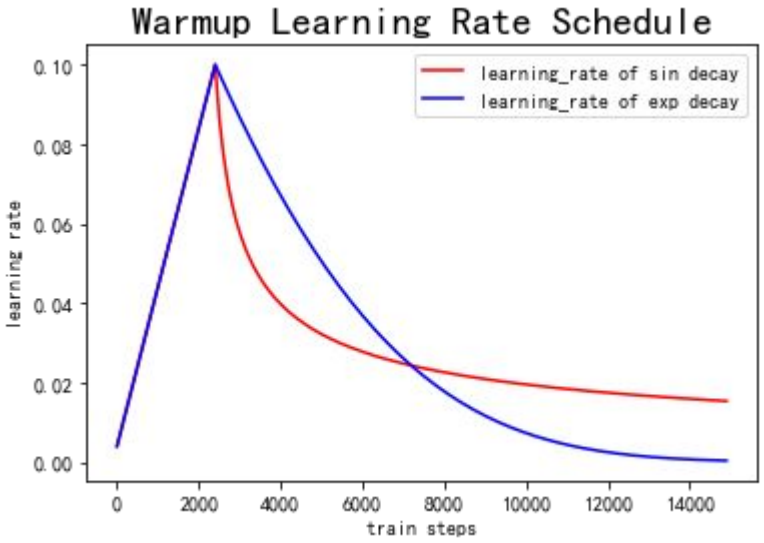


https://neptune.ai/blog/how-to-choose-a-learning-rate-scheduler

Learning rate warmup

In some cases (especially with large models or when using large batch-sizes) monotonically decreasing learning rate from an initial large value may lead to instability (oscillation) of the model.

Warmup strategies, i.e. using lower learning rates at the start of training [He et al., Goyal et al.], help to overcome early optimization difficulties.

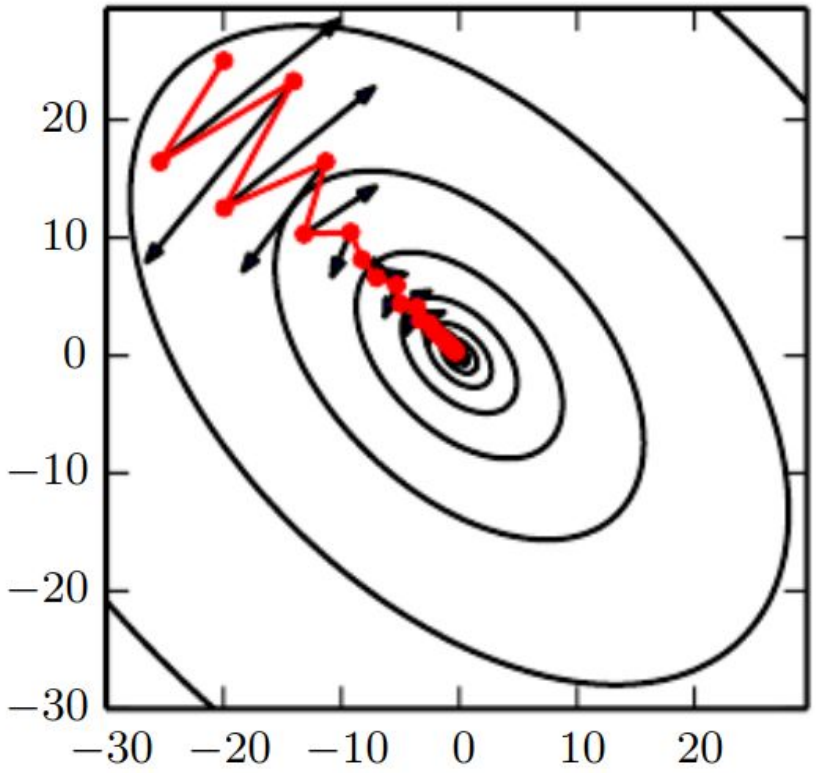


- **Constant warmup**: uses a low constant learning rate for the first few epochs of training.
- **Gradual warmup**: gradually ramps up the learning rate from a small to a large value.

He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In CVPR, 2016.
Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." *arXiv* (2017).

SGD with Momentum

The method of momentum (Polyak, 1964) aims to **accelerate learning of SGD**, especially in the face of high curvature, small but consistent gradients, or noisy gradients.



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

$$\hat{g} \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$$

$$v \leftarrow \alpha v - \epsilon \hat{g}$$

$$\theta \leftarrow \theta + v$$

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:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta [\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta + \alpha v), y^{(i)})]$$

$$\theta \leftarrow \theta + v$$

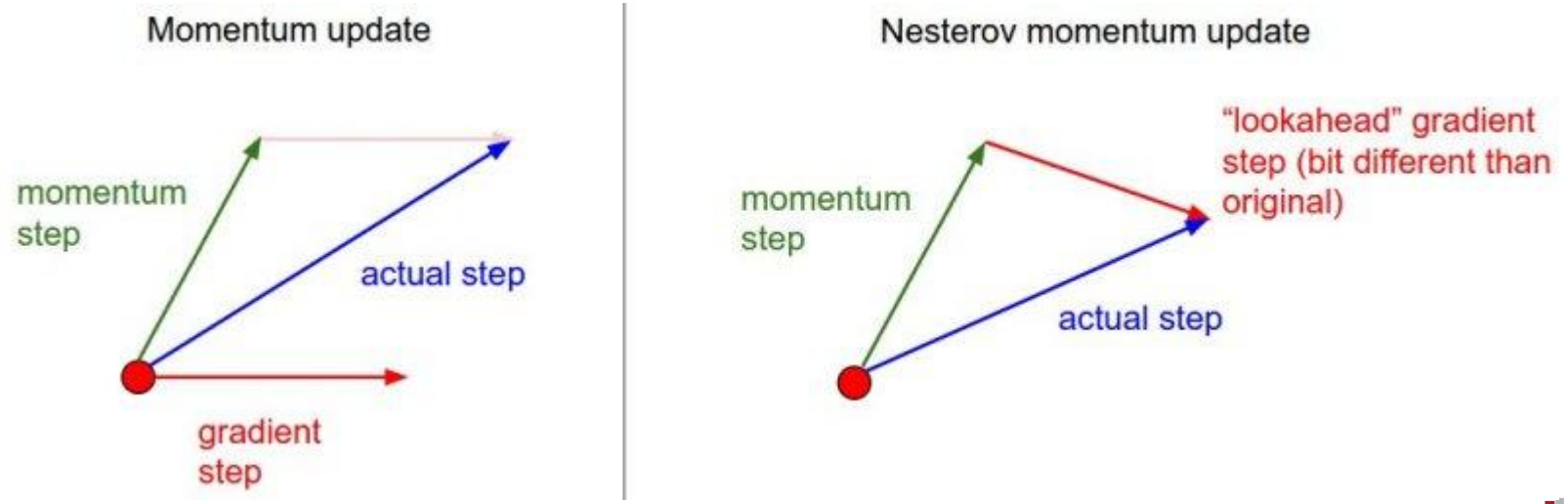


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

SGD with Momentum in Keras

```
from keras import optimizers

model = Sequential()
model.add(Dense(64,input_shape=(10,)))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

sgd=optimizers.SGD(lr=0.01,decay=1e-6,momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

https://keras.io/optimizers/

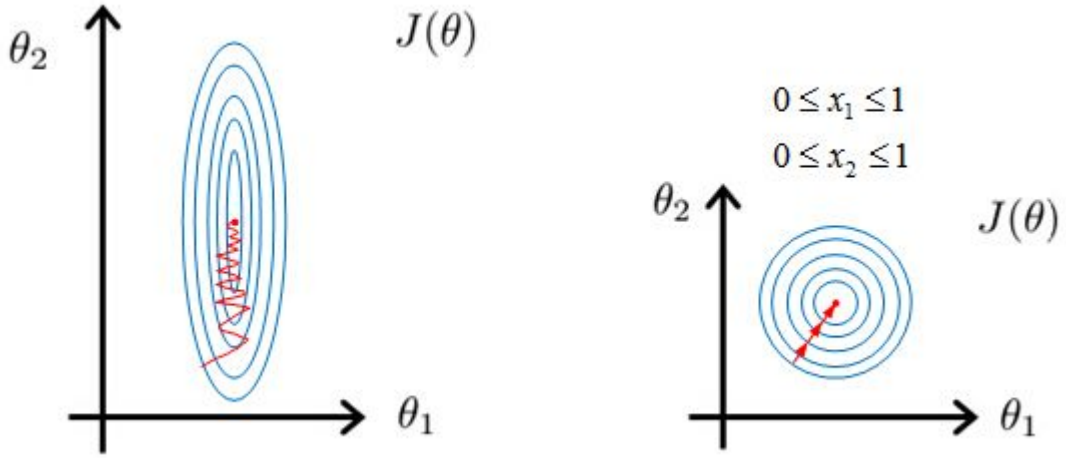
Training Deep Nets: Outline

- Introduction
- Stochastic Gradient Descent (Review)
- **Data Preprocessing**
- Weight Initialization
- Algorithms with Adaptive Learning Rates
- Other optimization strategies
- Practical Methodology

Data Preprocessing

Data preprocessing plays a very important role in many deep learning algorithms. In practice, many machine learning algorithms work best/learn faster after the data has been **normalized** and **whitened**.

SGD is not scale-free: most efficient on whitened data!



When approaching a dataset, the first thing to do is to look at the data itself and observe its properties.

Effect of feature scaling on the contour plot of the cost function.
Image source: https://machinelearningmedium.com/2017/08/23/multivariate-linear-regression/

Data Preprocessing

There are four common forms of data preprocessing a data matrix X, where we will assume that X is of size [N x D] (N is the number of data, D is their dimensionality).

- Simple rescaling
- Mean subtraction
- Normalization
- PCA and Whitening

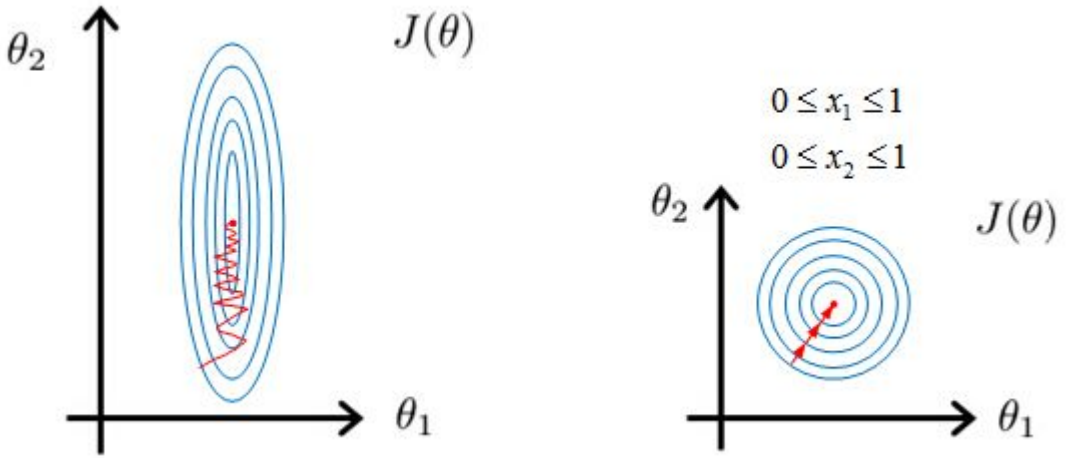


Image source: https://machinelearningmedium.com/2017/08/23/multivariate-linear-regression/

Data Preprocessing: Simple rescaling

In simple rescaling, our goal is to **rescale the data along each data dimension** (possibly independently) so that the final data vectors lie in the range [0,1] or [−1,1] (depending on your dataset). This is useful for later processing as many *default* parameters (e.g., epsilon in PCA-whitening) treat the data as if it has been scaled to a reasonable range.

- **Example:** When processing natural images, we often obtain pixel values in the range [0,255]. It is a common operation to rescale these values to [0,1]by dividing the data by 255.

```
X /= 255
```

Data Preprocessing: Mean subtraction

Mean subtraction is the most common form of preprocessing. It involves subtracting the mean across every individual *feature* in the data, and has the geometric interpretation of **centering the cloud of data around the origin along every dimension**. In numpy, this operation would be implemented as:

```
import numpy as np
X -= np.mean(X, axis=0)
```

With images specifically, for convenience it can be common to **subtract a single value from all pixels** (e.g. `X -= np.mean(X)`), and/or to do so separately across the three color channels.

Data Preprocessing: Normalization

Normalization refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization.

One is to divide each dimension by its standard deviation, once it has been zero-centered:

```
import numpy as np
X /= np.std(X, axis=0)
```

Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively. In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step.

Data Preprocessing: Normalization

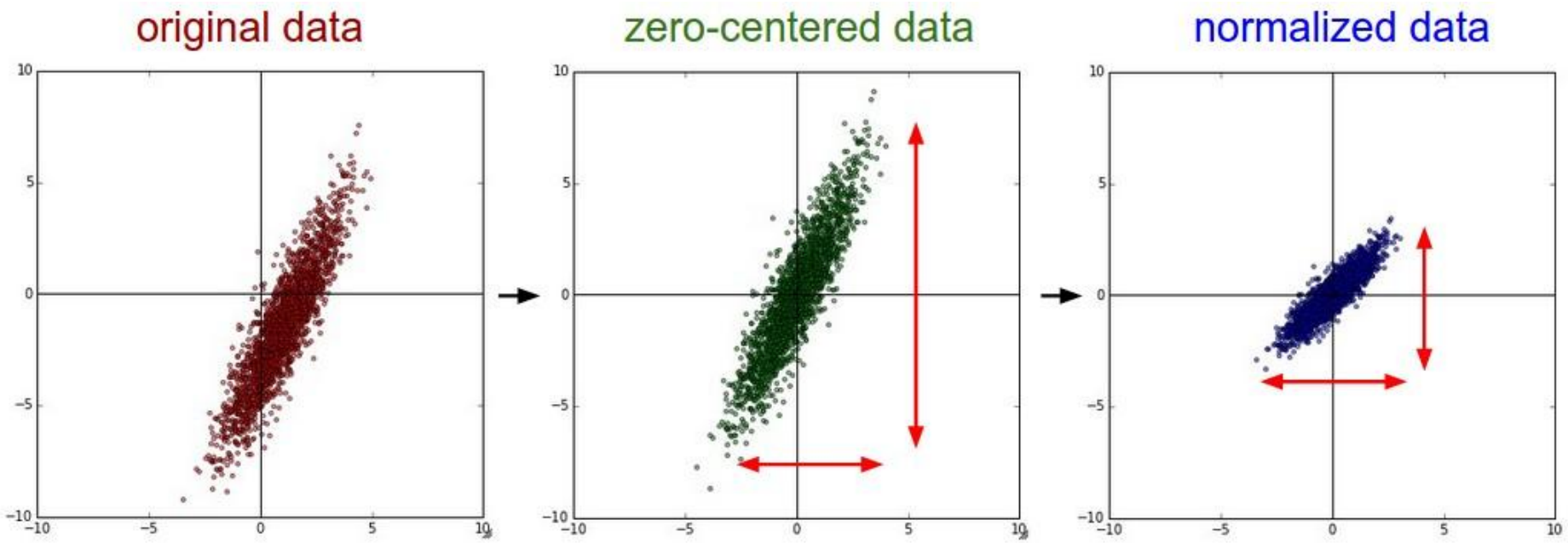


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

Data Preprocessing: PCA/ZCA Whitening

Whitening is another preprocessing technique that has been traditionally used to help ML models training converge faster. Many early deep learning algorithms relied on whitening to learn good features. Whitened data has zero mean, unit variance, and is **decorrelated**.

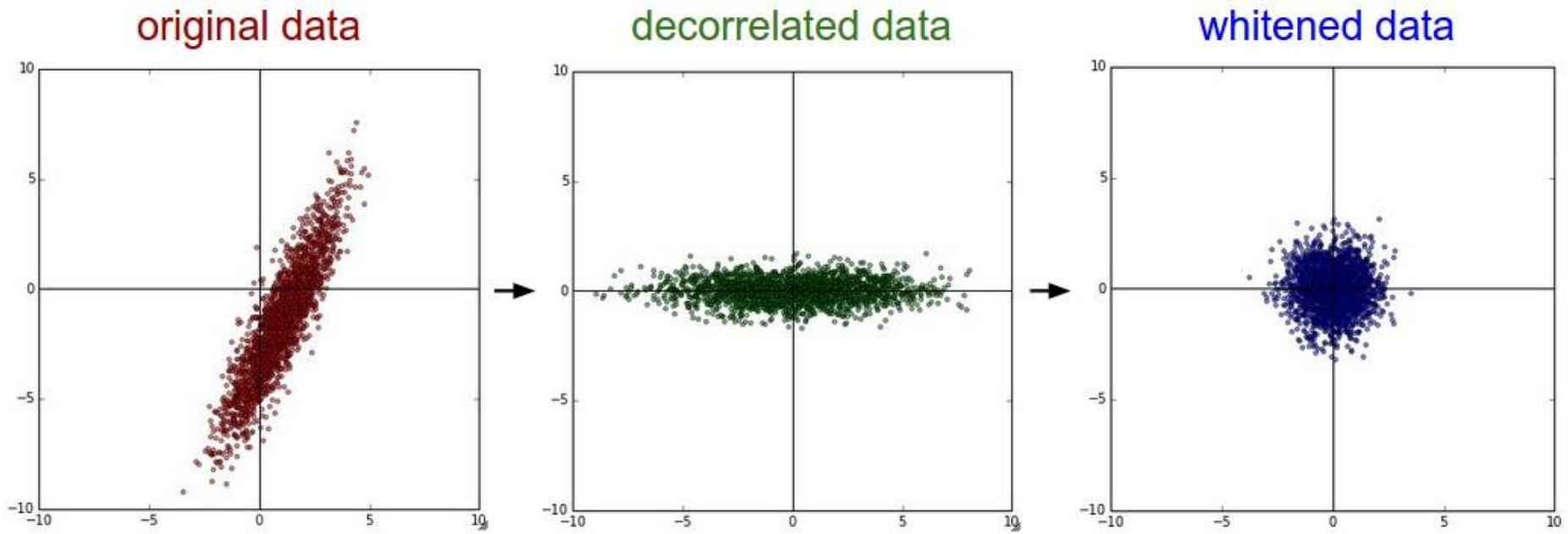


Image source: Stanford CS class [CS231n: Convolutional Neural Networks for Visual Recognition](#).

Data Preprocessing: PCA/ZCA Whitening

The data is first centered as described before. Then, we can compute the covariance matrix that tells us about the correlation structure in the data:

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

The (i,j) element of the data covariance matrix contains the *covariance* between i-th and j-th dimension of the data. In particular, the diagonal of this matrix contains the variances.

Data Preprocessing: PCA/ZCA Whitening

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

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

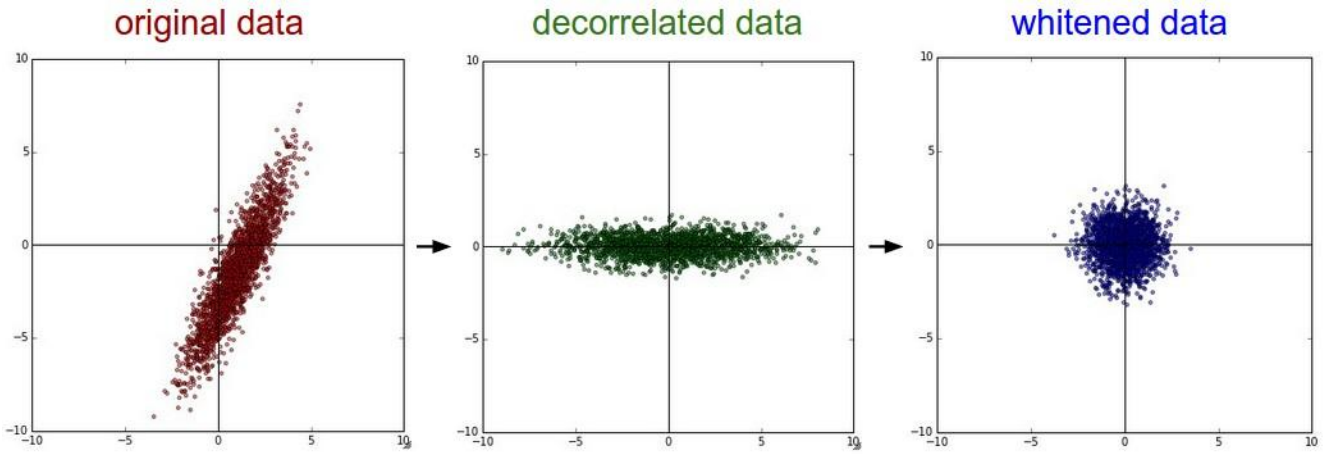
where the columns of U are the eigenvectors and S is a 1-D array of the singular values. To decorrelate the data, we project the original (but zero-centered) data into the eigenbasis:

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

Finally, the **whitening** operation takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale. This step would take the form:

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

Data Preprocessing



In practice PCA/Whitening are not used with Convolutional Neural Networks. However, **it is very important to zero-center the data**.

Common pitfall. An important point to make about the preprocessing is that any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation / test data.

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

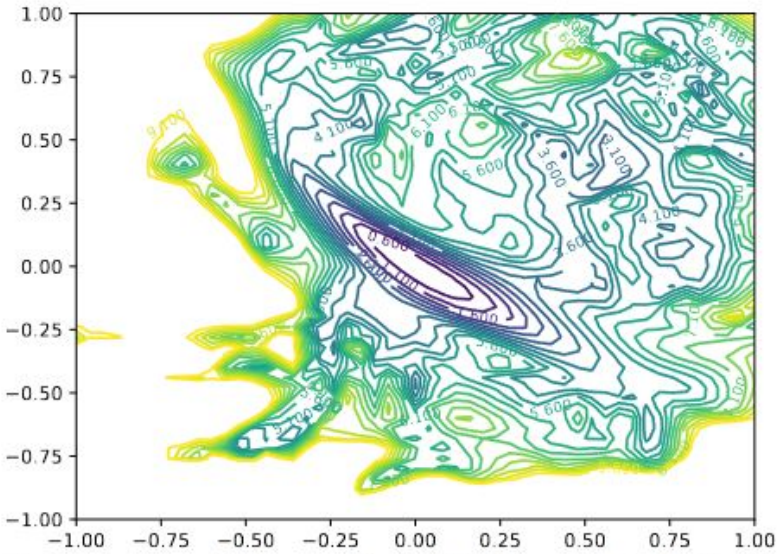
Training Deep Nets: Outline

- Introduction
- Stochastic Gradient Descent (Review)
- Data Preprocessing
- **Weight Initialization**
- Algorithms with Adaptive Learning Rates
- Other optimization strategies
- Practical Methodology

Parameter Initialization

Deep learning training algorithms are iterative and thus require the user to specify some initial point from which to begin iterating.

Most algorithms are strongly affected by the choice of initialization.



- The initial point can determine whether the algorithm converges or not, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.
- When learning does converge, the initial point may determine how quickly it does, and whether it converges to a point with high or low cost.
- Also the initialization may affect the generalization error as well, even for convergence points with similar cost.

Figure source: Li, Hao, et al. "Visualizing the loss landscape of neural nets." *NeurIPS*. 2018.

Parameter Initialization

Intuitions:

- If the weights in a network start too small, then the signal shrinks as it passes through each layer until it’s too tiny to be useful.

- If the weights in a network start too large, then the signal grows as it passes through each layer until it’s too massive to be useful.

Parameter initialization makes sure the weights are ‘just right’, keeping the signal in a reasonable range of values through many layers.

Parameter Initialization

Pitfall: all zero (or close to zero) initialization.

(Let's start with what we should not do)

A reasonable-sounding idea might be to **set all the initial weights to zero**, which we expect to be the "best guess" in expectation.

This turns out to be a mistake, because if **every neuron in the network computes the same output**, then they will also all **compute the same gradients** during backpropagation and **undergo the exact same parameter updates.**

There is no source of asymmetry between neurons if their weights are initialized to be the same!

Parameter Initialization

Symmetry breaking.

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

This motivates random initialization of the parameters.

We still want the weights to be very close to zero, but not identically zero.

The idea is that the neurons are all random and unique in the beginning, so **they will compute distinct updates** and integrate themselves as diverse parts of the full network.

We could search for a large set of basis functions that are all mutually different from each other, but this would be computationally expensive.

Parameter Initialization

Symmetry breaking.

We almost always initialize the weights in the model to values drawn from a Gaussian or uniform distribution. The implementation for one weight matrix with D inputs and H neurons might look like this:

```
import numpy as np
W = 0.01 * np.random.randn(D, H)
```

where randn samples from a **zero mean, unit standard deviation gaussian**.

Typically we **set the biases to heuristically chosen constants**, and initialize only the weights randomly.

Parameter Initialization

Calibrating the variances.

One problem with the suggested initialization is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.

We can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its *fan-in* (i.e. its number of inputs):

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

This is known as the **Xavier/Glorot initialization algorithm**..

Xavier Glorot & Yoshua Bengio's *Understanding the difficulty of training deep feedforward neural networks*, JMLR 2010.

Parameter Initialization

Calibrating the variances.

For the more recent rectifying nonlinearities, that doesn't hold, and in a recent paper (He et al 2015) they build on Glorot & Bengio and suggest using:

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

Which makes sense: **a rectifying linear unit is zero for half of its inputs**, so you need to double the size of weight variance to keep the signal's variance constant.

He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", CVPR 2015.

Parameter Initializers in Keras

```
from keras import initializers

model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))

keras.initializers.Zeros()
keras.initializers.Ones()
keras.initializers.Constant(value=0)
keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
keras.initializers.RandomUniform(minval=-0.05, maxval=0.05,
seed=None)
glorot_normal(seed=None)
glorot_uniform(seed=None)
he_normal(seed=None)
...
```

https://keras.io/initializers/

The Lottery Ticket Hypothesis

A technique for training pruned deep nets that **highlights the importance of weights' initialization.**

> *<<**Lottery ticket hypothesis**: a dense, randomly-initialized, feed-forward network contain subnetworks (winning tickets) that—when trained in isolation— reach test accuracy comparable to the original network in a similar number of iterations. >>*

1. Randomly initialize a neural network $f(x; \theta_0)$ (where $\theta_0 \sim \mathcal{D}_\theta$).
2. Train the network for j iterations, arriving at parameters θ_j.
3. Prune $p\%$ of the parameters in θ_j, creating a mask m.
4. Reset the remaining parameters to their values in θ_0, creating the winning ticket $f(x; m \odot \theta_0)$.

Prune the weights with smallest magnitude

The pruned model only reaches comparable (or better) performance if its weights are initialized

Frankle, Jonathan, and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks." *International Conference on Learning Representations*. 2018.

The Lottery Ticket Hypothesis

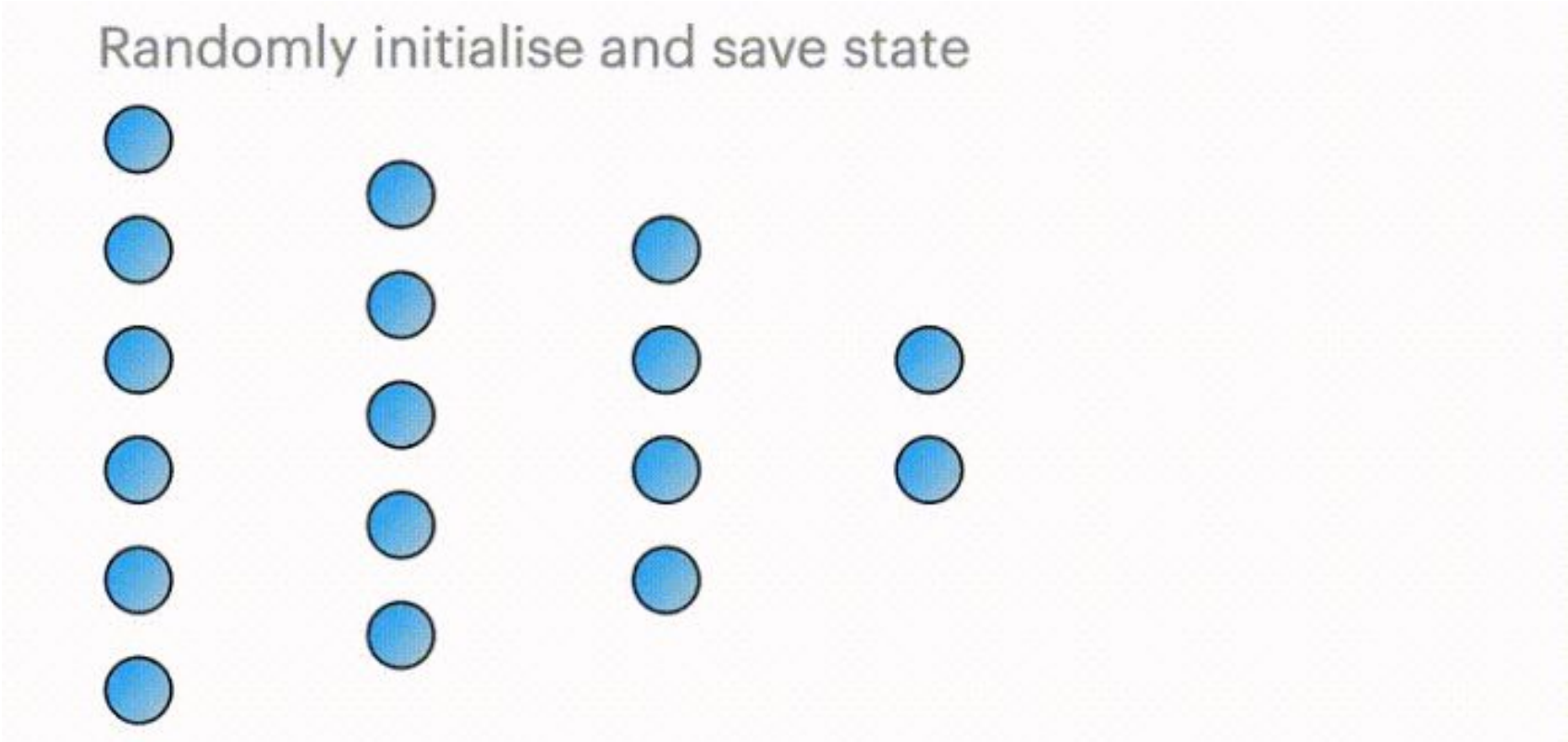


Image source: https://medium.com/dair-ai/the-lottery-ticket-hypothesis-7cd4eae3faaa

Frankle, Jonathan, and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks." *International Conference on Learning Representations*. 2018.

Training Deep Nets: Outline

- Introduction
- Stochastic Gradient Descent (Review)
- Data Preprocessing
- Weight Initialization
- **Algorithms with Adaptive Learning Rates**
- Other optimization strategies
- Practical Methodology

Algorithms with Adaptive Learning Rates

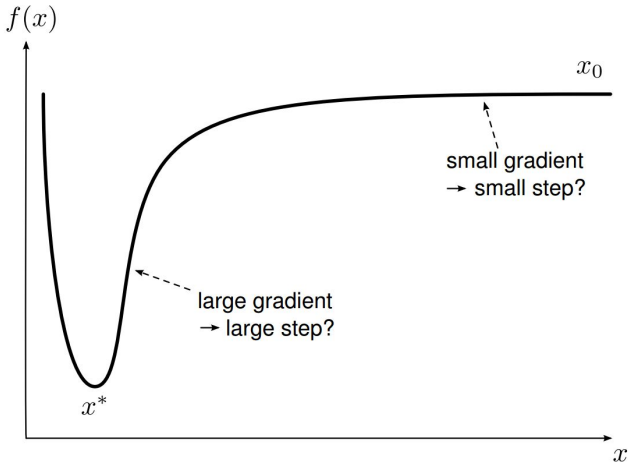
The learning rate is one of the most difficult hyperparameters to set.

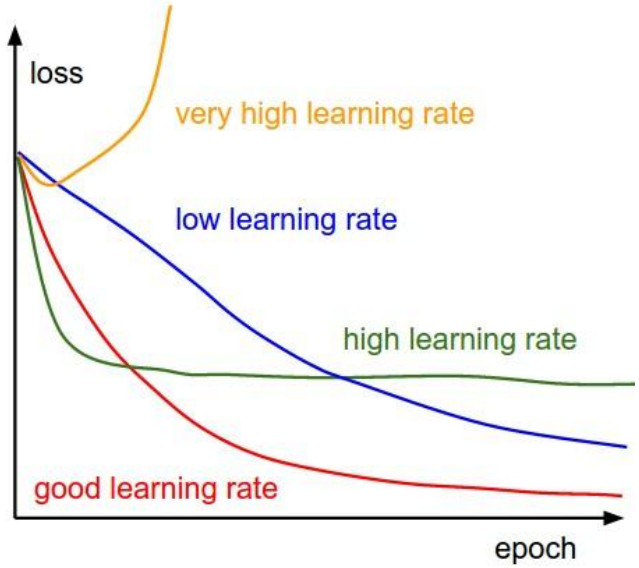
It significantly affects the performance of the model.

As we have seen 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.

A number of methods have been introduced that **adapt the learning rates of model parameters.**





AdaGrad

The AdaGrad algorithm (Duchi et al. 2011) **individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient.**

The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.

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

The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant $\boldsymbol{\delta}$, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r = 0}$

while stopping criterion not met **do**

Sample a minibatch of m samples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

Accumulate squared gradient: $r \leftarrow r + g \odot g$

Compute update: $\Delta\theta \leftarrow \frac{\epsilon}{\delta + \sqrt{r}} \odot g$

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

RMSProp

The RMSProp algorithm (Hinton, 2012) modifies AdaGrad to perform better in the nonconvex setting by **changing the gradient accumulation into an exponentially weighted moving average**.

AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a nonconvex function to train a neural network the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl. 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 so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

Accumulate squared gradient: $r \leftarrow \rho r + (1 - \rho) g \odot g$

Compute update: $\Delta\theta \leftarrow \frac{\epsilon}{\delta + \sqrt{r}} \odot g$

Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

Adam

The name of the Adam algorithm (Kingma and Ba, 2014) derives from the phrase “adaptive moments.”

In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with some distinctions.

First, in Adam, **momentum is incorporated directly as an estimate of the first-order moment** (with exponential weighting) of the gradient.

Second, 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.

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$

AdamW

L2 regularization and weight decay regularization are equivalent for standard stochastic gradient descent (when rescaled by the learning rate), but this is not the case for adaptive gradient algorithms, such as Adam.

AdamW is a simple modification of Adam to **recover the original formulation of weight decay regularization** by decoupling the weight decay from the optimization steps taken w.r.t. the loss function.

Improves Adam's generalization performance in a number of image classification benchmarks.

Loshchilov, Ilya, and Frank Hutter. "Decoupled Weight Decay Regularization." *International Conference on Learning Representations*. 2018.

AdamW

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

Loshchilov, Ilya, and Frank Hutter. "Decoupled Weight Decay Regularization." *International Conference on Learning Representations*. 2018.

RAdam

- Learning rate warmup heuristics work well to **stabilize training** in combination with adaptive learning rate algorithms**.**
- These heuristics also work well to **improve generalization.**
- Liu et al. decided to **study the theory** behind learning rate warmup.
- They found a problem with adaptive learning rates — during the first few updates **the gradients had very high variance**.
- The authors studied the problem in detail and concluded that the issue can be resolved/mitigated by:
  - 1. Applying warm up with a **low initial learning rate.**
  - 2. Or, simply **turning off the momentum term** for the first few sets of input batches.
- As training continues, the variance will stabilize, and from there, the **learning rate can be increased** and the **momentum term can be added back in**.

Liu, Liyuan, et al. "On the Variance of the Adaptive Learning Rate and Beyond." *International Conference on Learning Representations*. 2020.

LARS and LAMB

When training with **large batch sizes** we need to increase the learning rate to converge in the same number of epochs. However as learning rate increases, training becomes more unstable.

Layerwise Adaptive Rate Scaling (LARS): uses a layer-wise learning rate to stabilize learning. The layer-wise learning rate λ^l is the global learning rate η times the ratio of the norm of the layer weights to the norm of the layer gradients. They call this ratio the “trust ratio”.

$$\lambda^l = \eta \times \frac{||w^l||}{||\nabla L(w^l)||}$$

Layer-wise Adaptive Moments optimizer for Batch training (LAMB): makes some changes to LARS, among them it uses Adam update rule instead of SGD.

https://towardsdatascience.com/an-intuitive-understanding-of-the-lamb-optimizer-46f8c0ae4866

Large Batch Size training

(Out of the scope of this lecture)

Goyal, Priya, et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour." *arXiv preprint arXiv:1706.02677* (2017).

Hoffer, Elad, Itay Hubara, and Daniel Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks." *arXiv preprint arXiv:1705.08741* (2017).

Smith, Samuel L., et al. "Don't decay the learning rate, increase the batch size." *arXiv preprint arXiv:1711.00489* (2017).

Choosing the right optimization algorithm

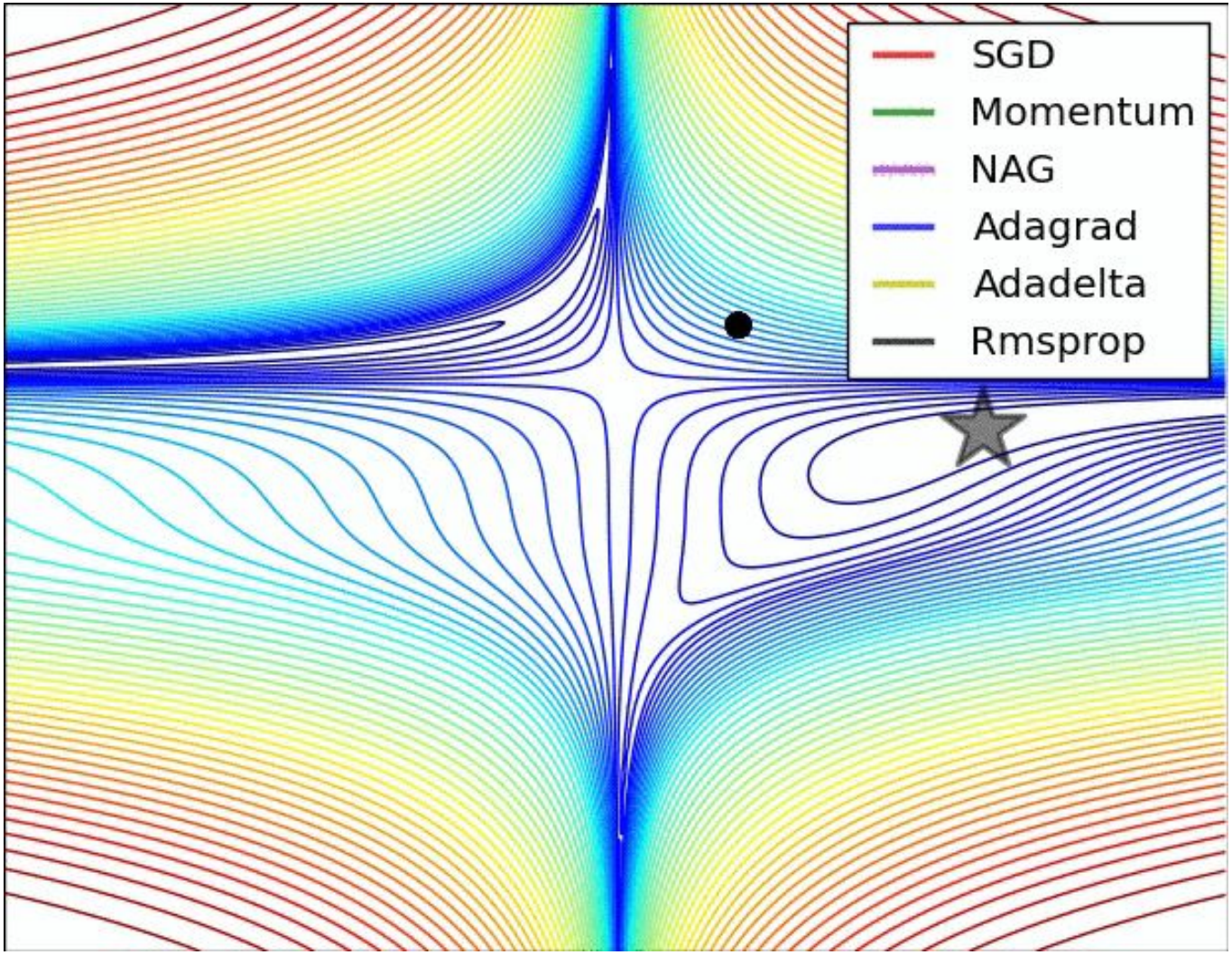


Image source: Alec Radford.

Choosing the right optimization algorithm

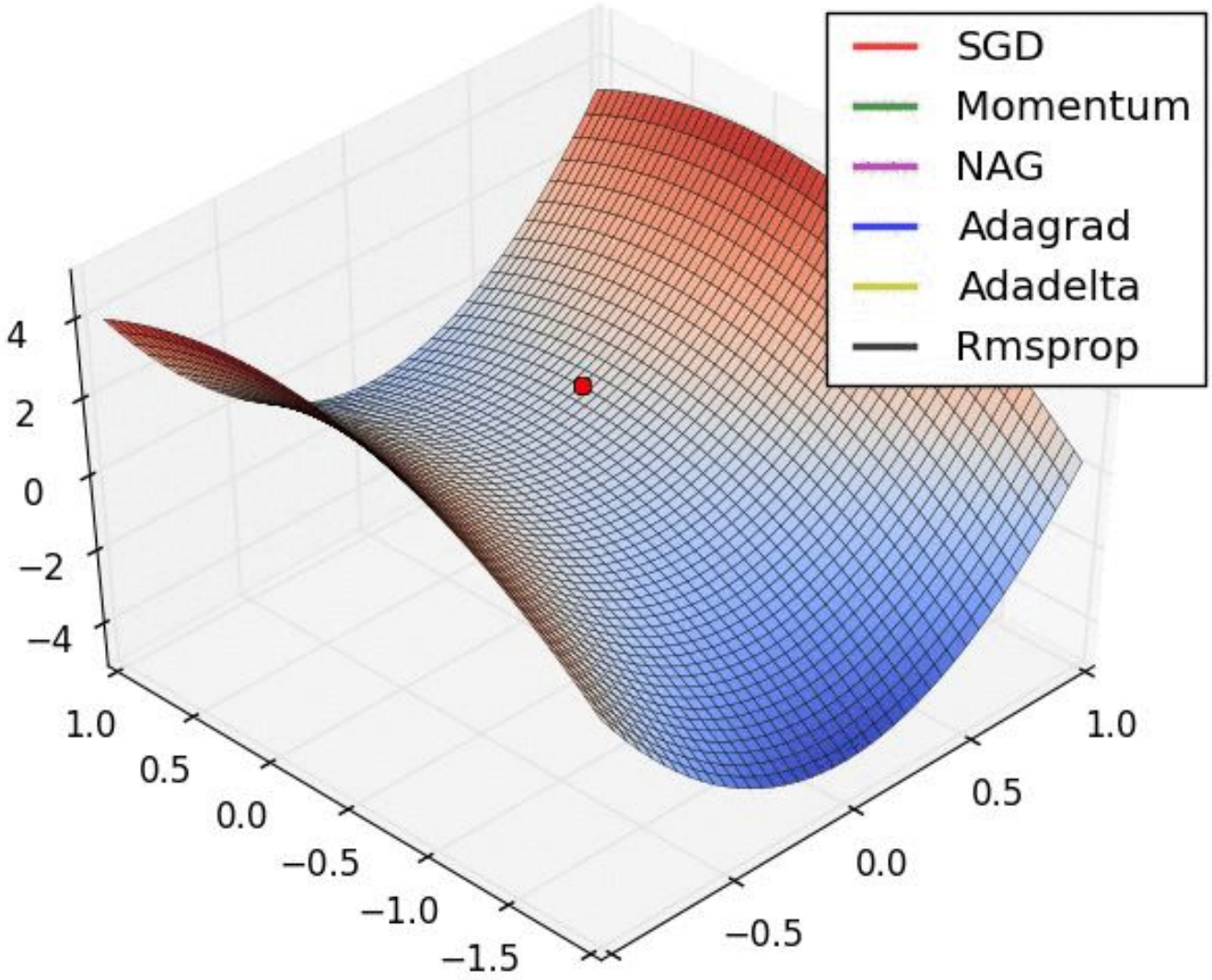


Image source: Alec Radford.

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. While the results suggest that the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta) performed fairly robustly, no single best algorithm has emerged.

The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).

Algorithms with Adaptive Learning Rates in Keras

```
from keras import optimizers

keras.optimizers.RMSprop(lr=0.001,rho=0.9,epsilon=None,decay=0.0)

keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)


keras.optimizers.Adadelta(lr=1.0,rho=0.95,epsilon=None, decay=0.0)


keras.optimizers.Adam(lr=0.001,beta_1=0.9,beta_2=0.999,
                      epsilon=None, decay=0.0, amsgrad=False)

keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999,
                      epsilon=None, schedule_decay=0.004)
```

https://keras.io/optimizers/

Training Deep Nets: Outline

- Introduction
- Stochastic Gradient Descent (Review)
- Data Preprocessing
- Weight Initialization
- Algorithms with Adaptive Learning Rates
- **Other optimization strategies**
- Practical Methodology

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

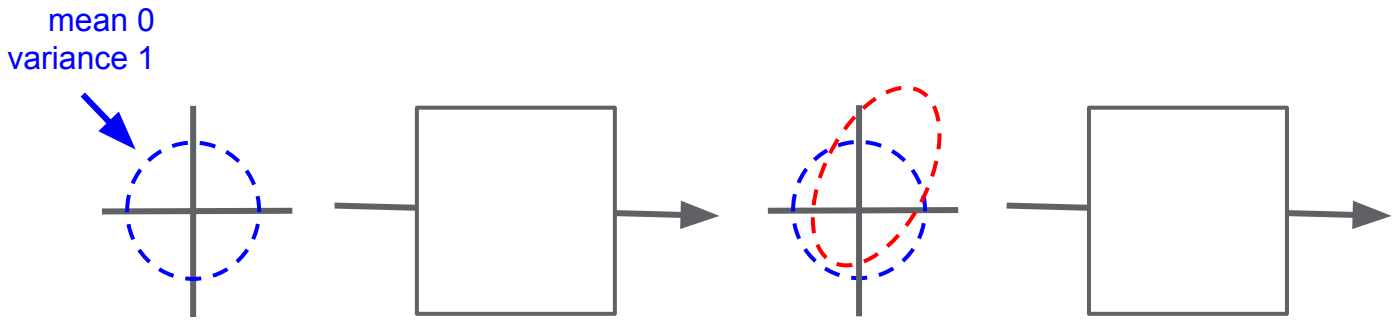
10x speedup in training, 2% improvement in performance on ImageNet!

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

Batch Normalization

SGD is not scale-free: most efficient 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

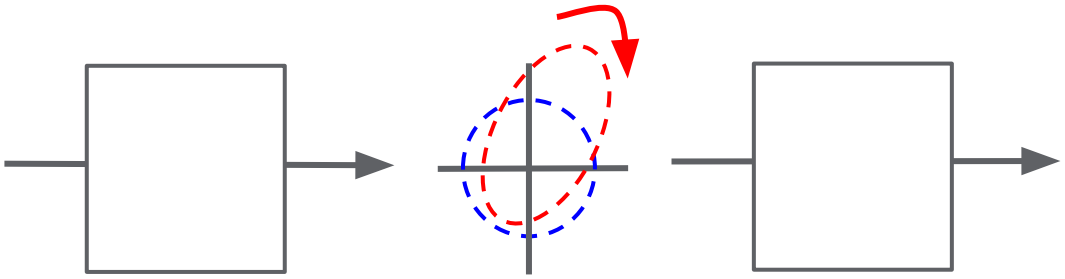
Excerpt from the Batch Norm paper:

> we seek to reduce the internal covariate shift. By fixing the distribution of the layer inputs x as the training progresses, we expect to improve the training speed. It has been long known (LeCun et al., 1998b; Wiesler & Ney, 2011) that the network training converges faster if its inputs are whitened – i.e., linearly transformed to have zero means and unit variances, and decorrelated. As each layer observes the inputs produced by the layers below, it would be advantageous to achieve the same whitening of the inputs of each layer. By whitening the inputs to each layer, we would take a step towards achieving the fixed distributions of inputs that would remove the ill effects of the internal covariate shift.

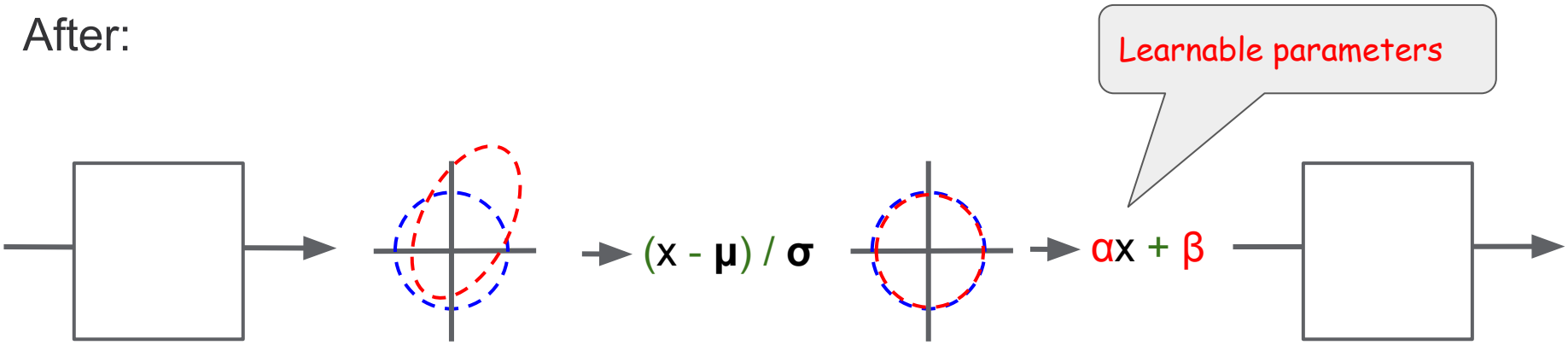
Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *ICML* 2015.

Batch Normalization

Before:



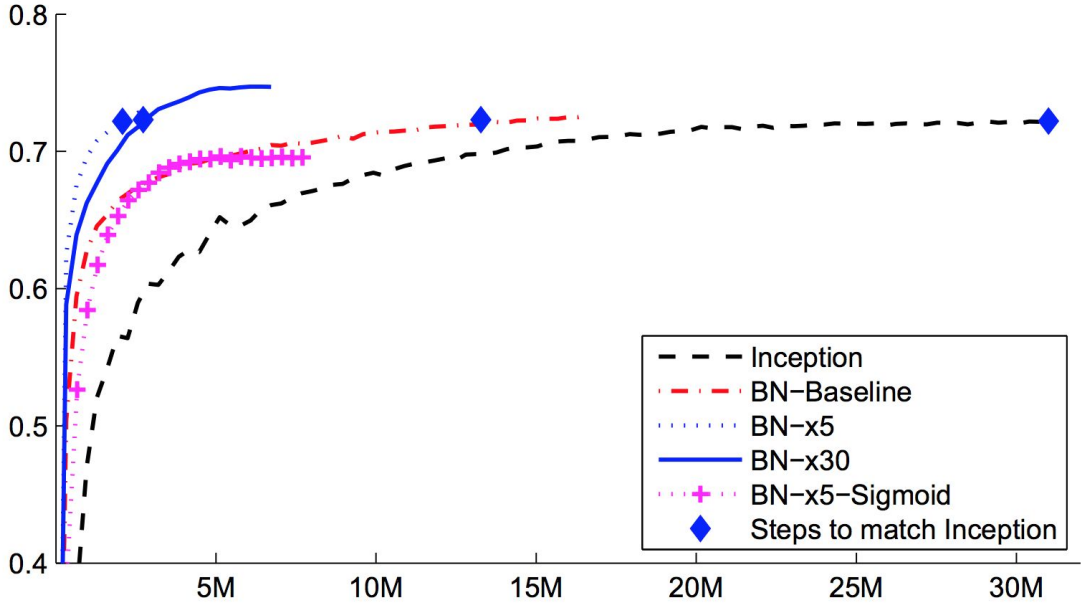
After:



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

Results



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

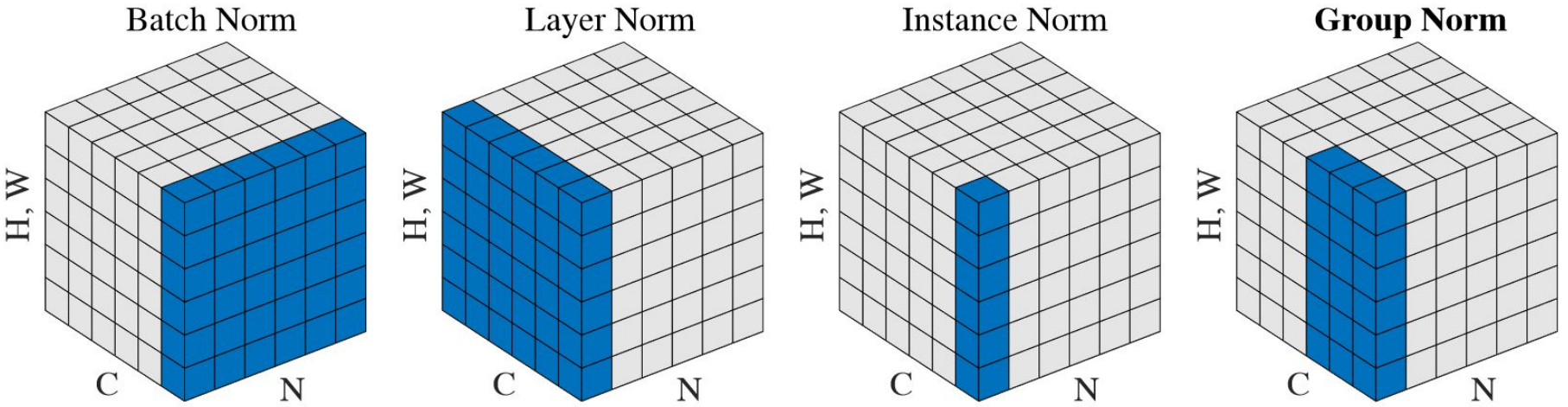
Batch Normalization in Keras

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99,
            epsilon=0.001, center=True, scale=True,
            beta_initializer='zeros', gamma_initializer='ones',
            moving_mean_initializer='zeros',
            moving_variance_initializer='ones',
            beta_regularizer=None, gamma_regularizer=None,
            beta_constraint=None, gamma_constraint=None)
```

https://keras.io/layers/normalization/

Normalization methods

Visual comparison of different normalization methods:

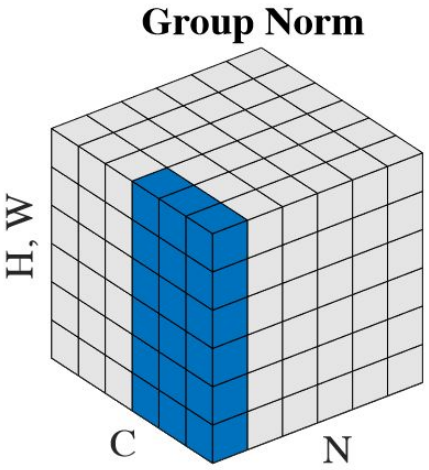


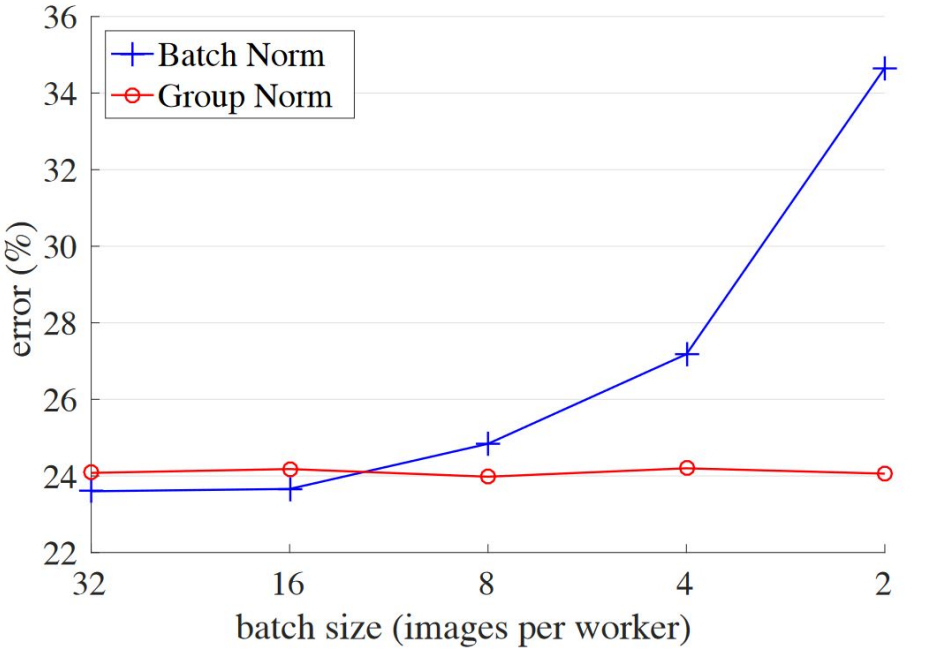
Where, In the case of 2D images, N is the batch axis, C is the channel axis, and H and W are the spatial height and width axes.

Figure source: Wu, Yuxin, and Kaiming He. "Group normalization." *Proceedings of the European conference on computer vision (ECCV)*. 2018.

Group Normalization

BN exhibits drawbacks that are also caused by its distinct behavior of normalizing along the batch dimension. In particular, it is required for BN to work with a sufficiently large batch size.





ImageNet classification error *vs.* batch sizes.

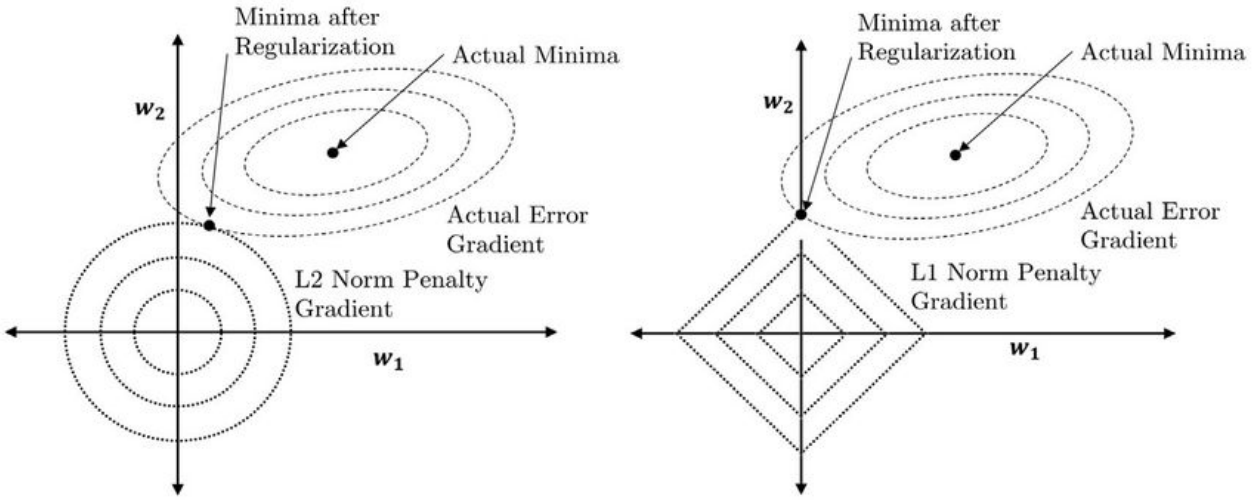
Wu, Yuxin, and Kaiming He. "Group normalization." *Proceedings of the European conference on computer vision (ECCV)*. 2018.

Regularization: L2/L1 Parameter regularization

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

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

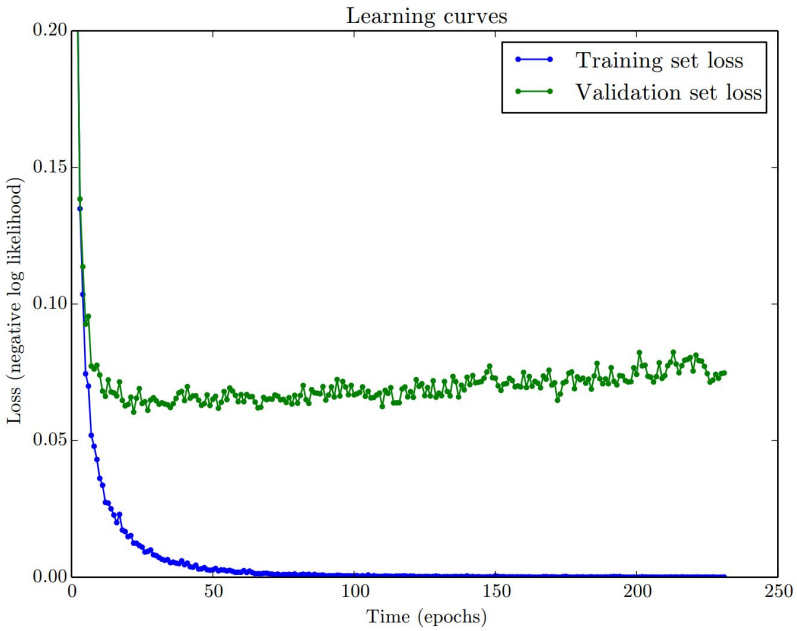
$\boldsymbol{\alpha}$ = weight decay parameter!



Regularization: Early stopping

Training algorithms do not usually halt at a local minimum.

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



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

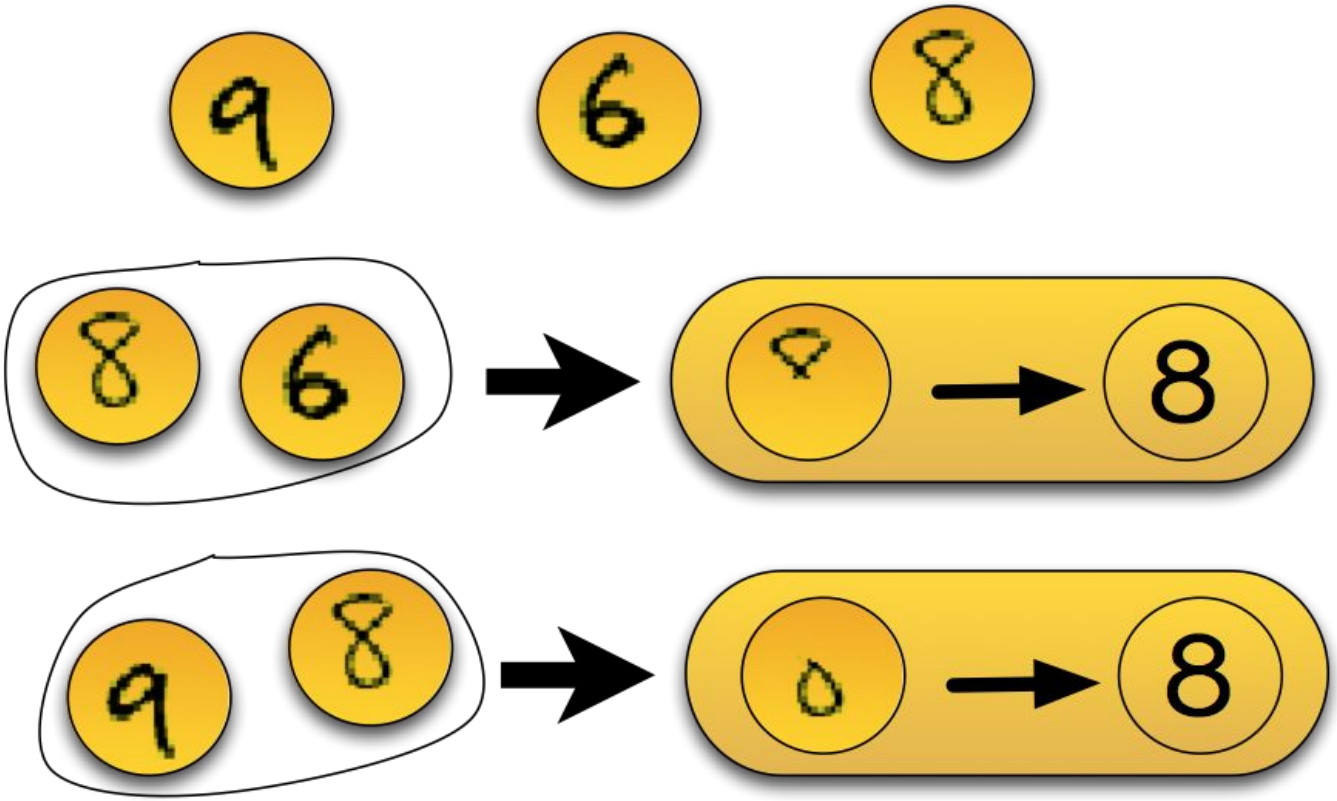
It is designed to halt whenever overfitting begins to occur. This not necessarily means when the gradients became very small.

Regularization: Early stopping in Keras

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
                              patience=0, verbose=0, mode='auto')
```

https://keras.io/callbacks/

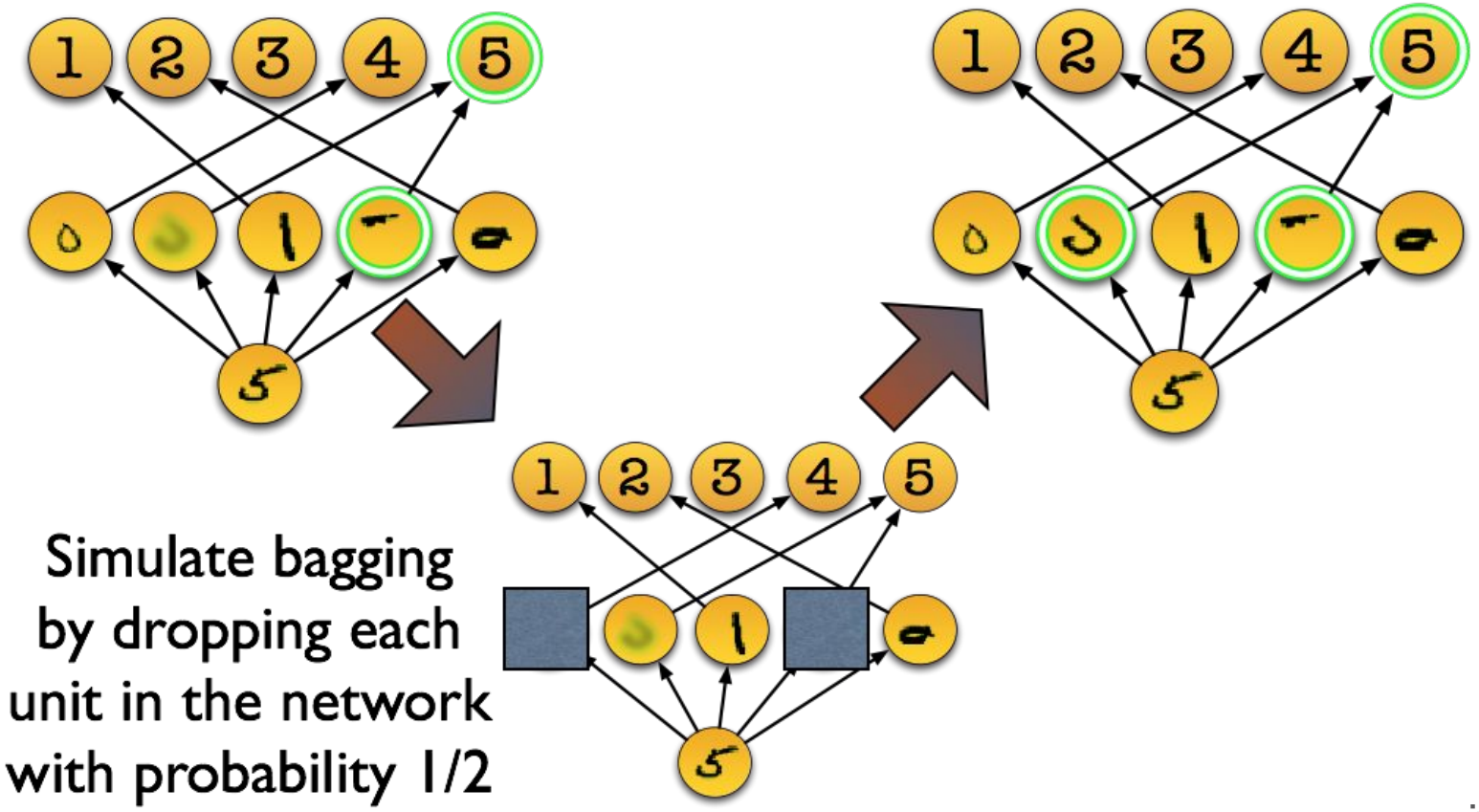
Regularization: Bagging



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

Regularization: Dropout

Dropout training



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

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]

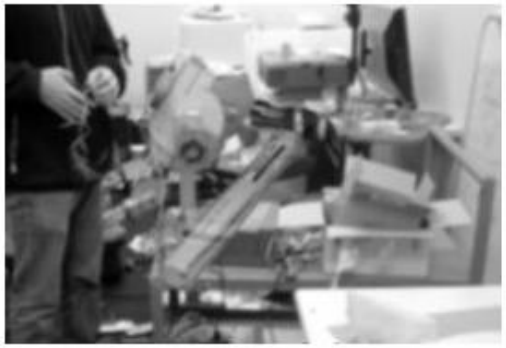
Data Augmentation in Keras

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

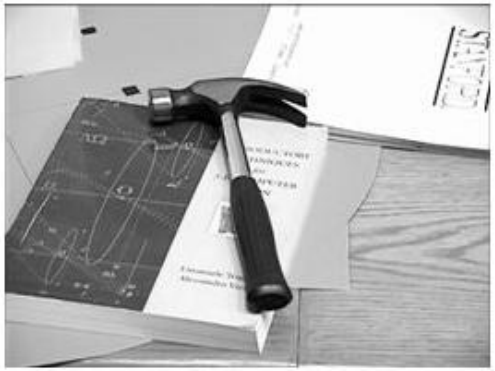
Building powerful image classification models using very little data

Synthetic Data

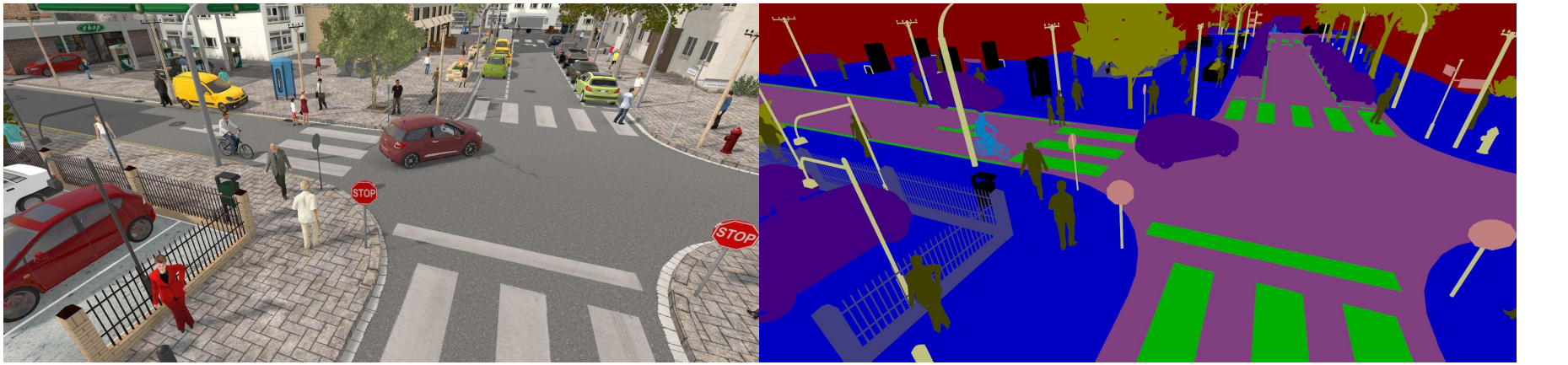
Synthetic Examples (Training set)


Real Examples (Test set)



Synthetic Data



The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes. German Ros et al. CVPR 2016.

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...

Transfer Learning

You will be able to recognize...

Image source: Yannis Ghazouani, 2016.

Training Deep Nets: Outline

- Introduction
- Stochastic Gradient Descent (Review)
- Data Preprocessing
- Weight Initialization
- Algorithms with Adaptive Learning Rates
- Other optimization strategies
- **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

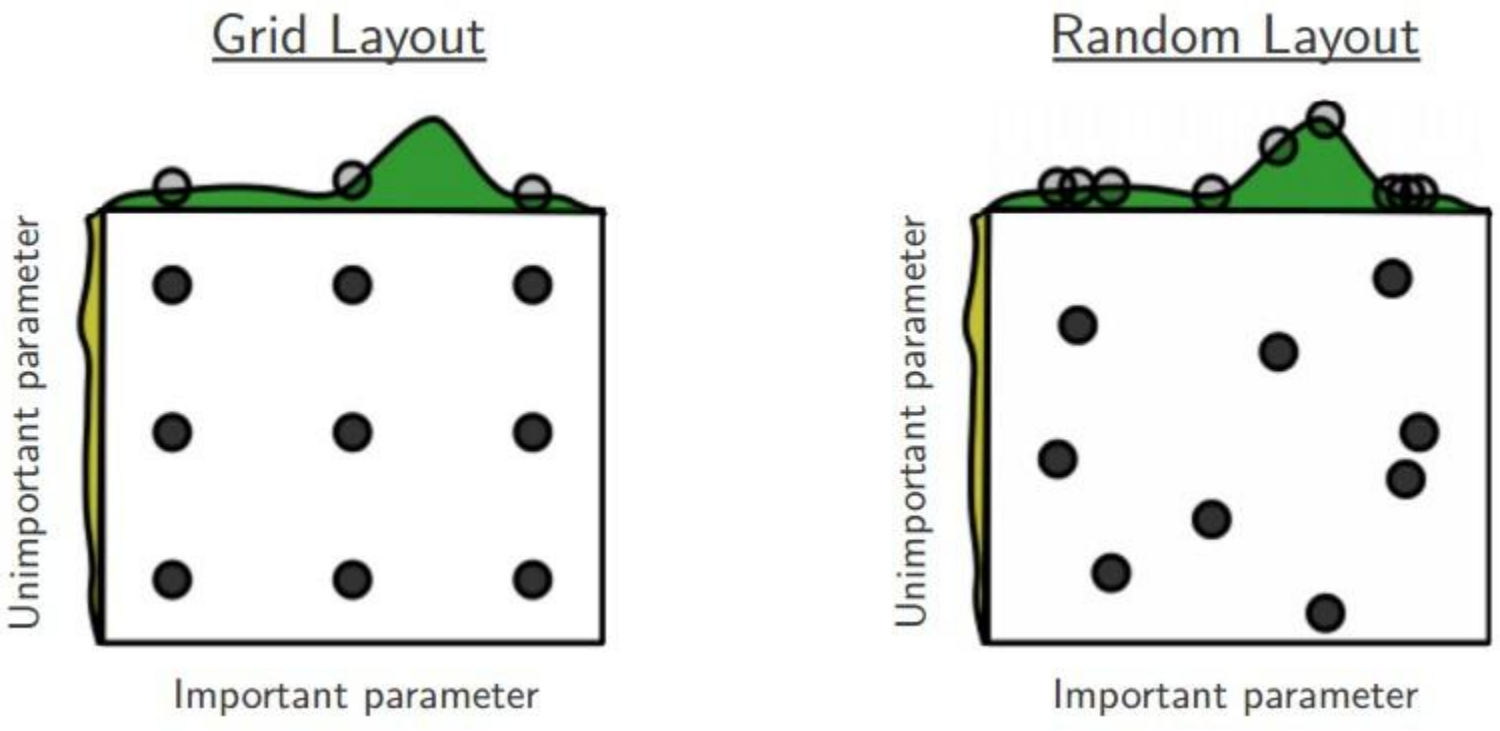


Image source: (Bergstra and Bengio, 2012).

Practical Methodology

Quora Home Answer Notifications Search Quora

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

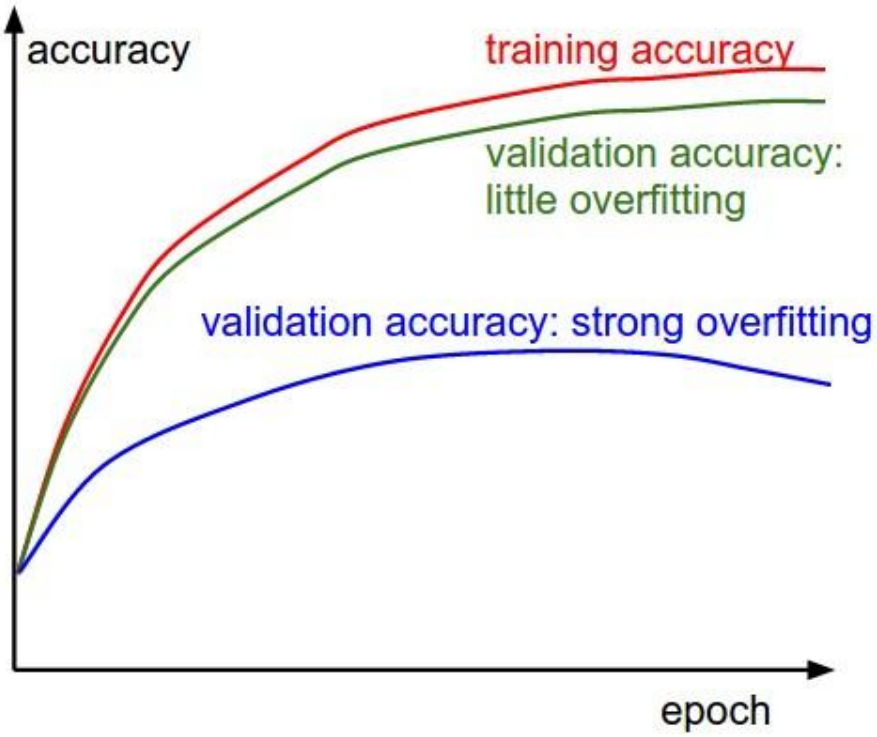
Ian Goodfellow, AI Research Scientist
Answered Jul 17, 2017

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!



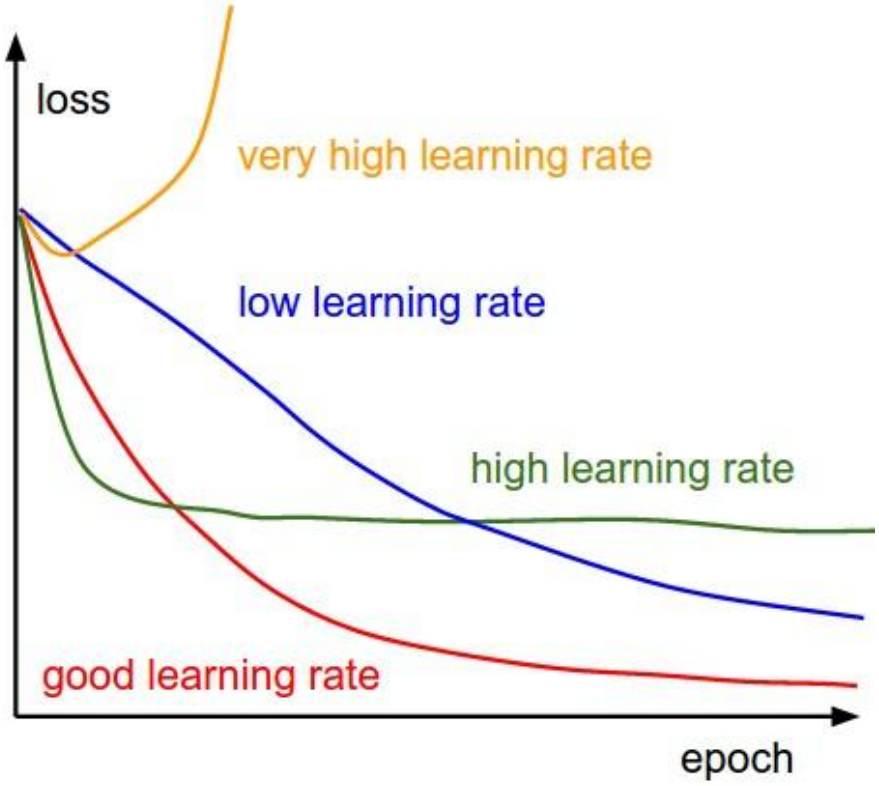


Image source: Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.

Practical Methodology

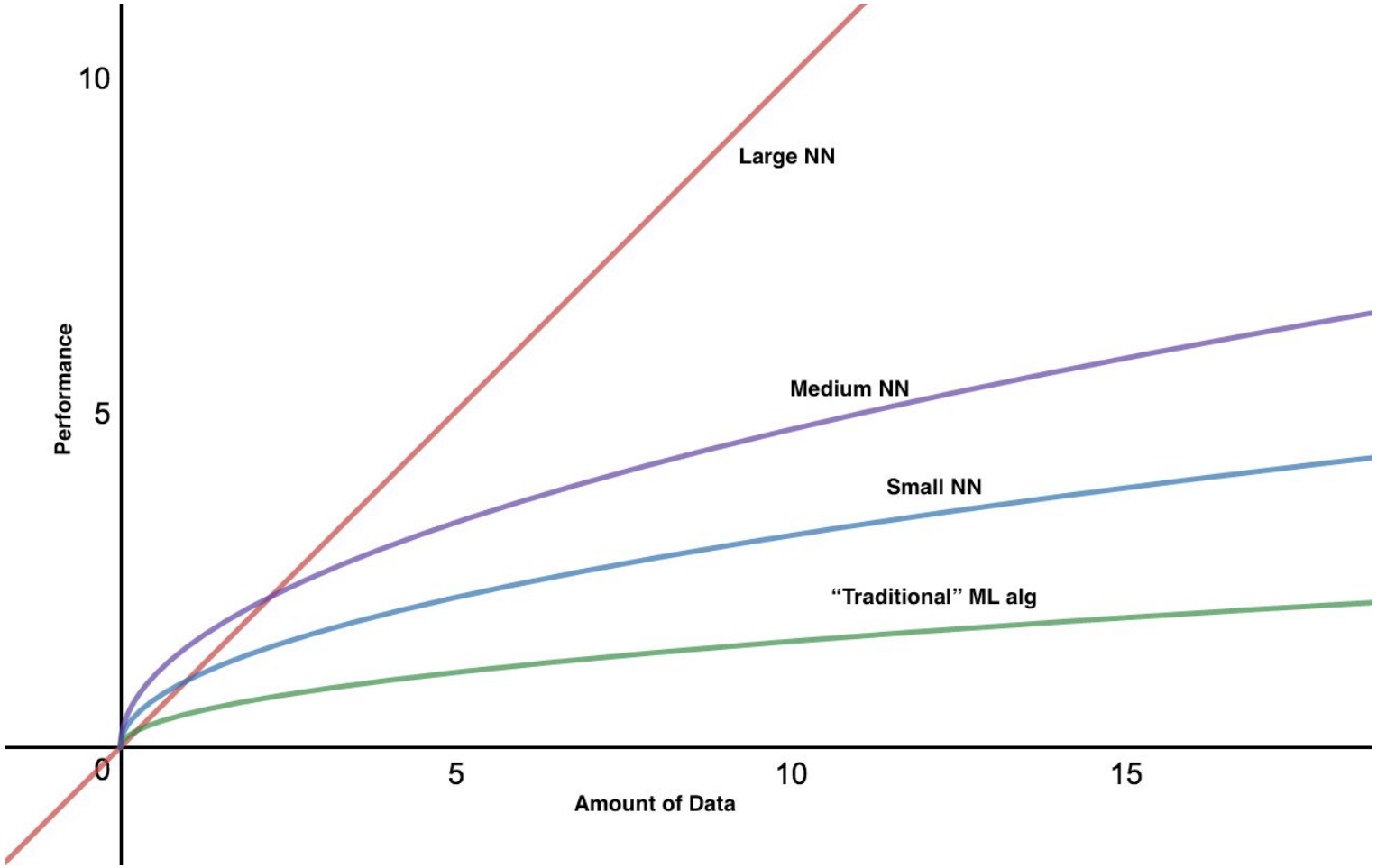
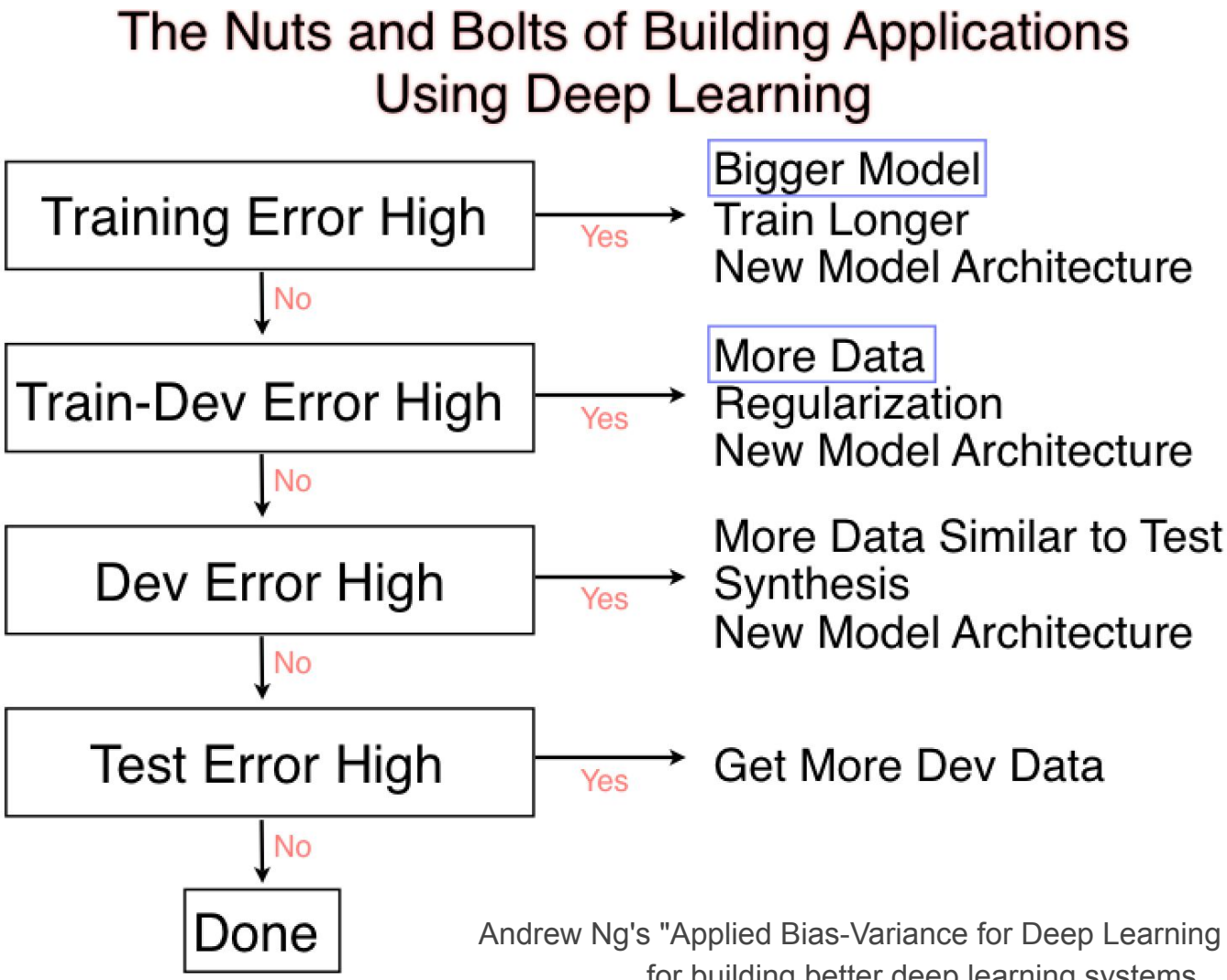


Image source: https://kevinzakka.github.io/2016/09/26/applying-deep-learning/

Practical Methodology



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

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.
- 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.


KEEP
CALM
AND LOWER YOUR
LEARNING
RATE

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.*