Deep Neural Networks

Pietro Michiardi

EURECOM

Outline i

Outline ii

Outline iii

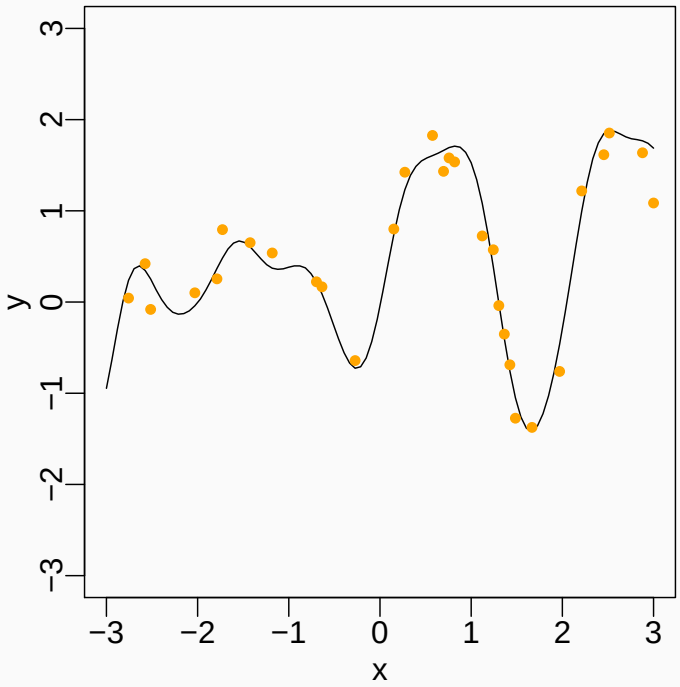
Fully Connected Nets

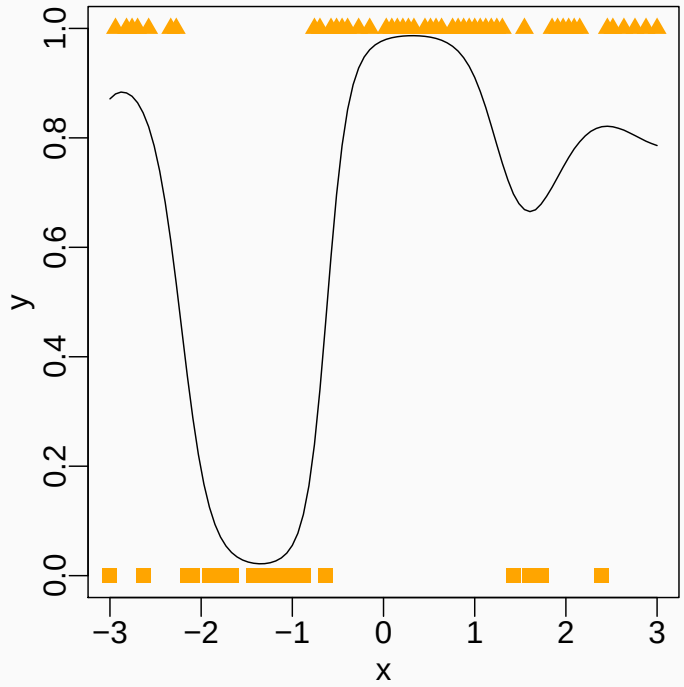
Introduction and Definitions

Learning from Data – Function Estimation

- We are interested in estimating a function $\mathbf{f}(\mathbf{x})$ from data
- Most problems in Machine Learning can be cast this way!

- Take these two simple examples





Definitions

- Many different names
  - (Deep) Feed-forward networks
  - Deep Neural Networks (Deep Nets)
  - Multilayer perceptrons (MLPs)
  - Fully connected networks (or layers)

- Structure of such models
  - Input, hidden, output layers
  - Depth and width of the network
  - Cost function

- Our goal is function approximation
  - Given a "true" function $\mathbf{y} = \mathbf{f}^*(\mathbf{x})$
  - Deep Nets approximate it with $\mathbf{f}(\mathbf{x}, W)$
  - Such that $\mathbf{f} \approx \mathbf{f}^*$

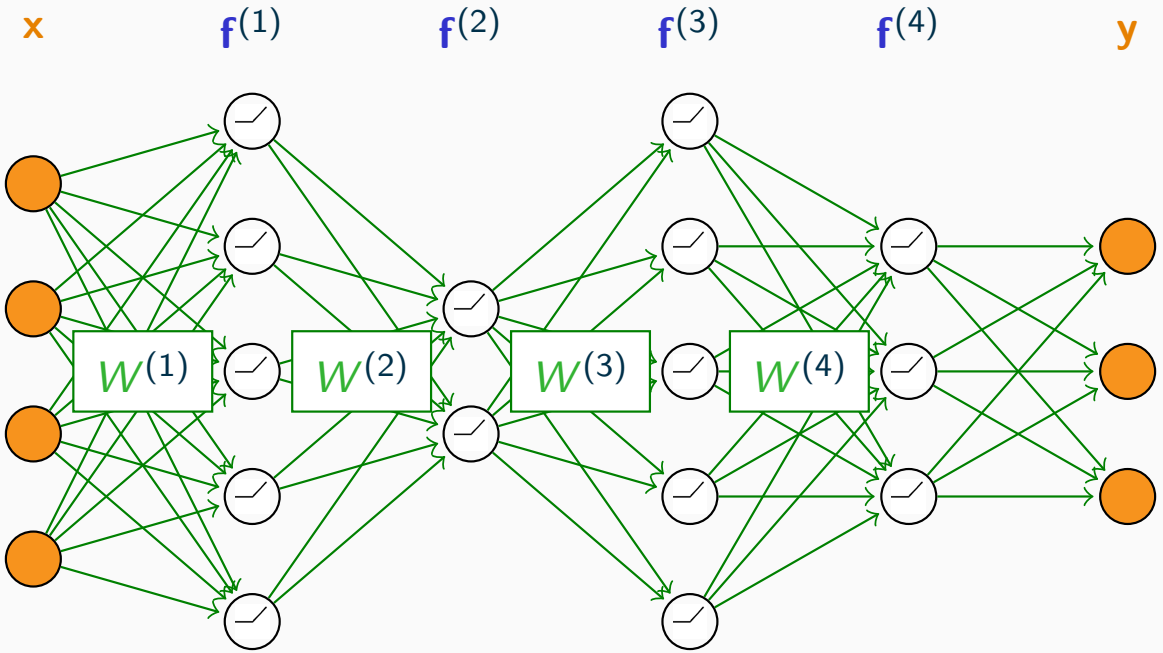
Deep Neural Networks

Deep Nets implement a composition of parametric functions

$$\mathbf{f}(\mathbf{x}, W) = \mathbf{f}^{(L)}\left(\mathbf{f}^{(L-1)}\left(\cdots \mathbf{f}^{(1)}\left(\mathbf{x}\right)\cdots\right)\right)$$

with (note: we include bias terms in the notation)

$$\mathbf{f}^{(l)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^{\top} W^{(l)}\right)$$



Notation and dimensions

Inputs : $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$

Labels : $Y = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$

Weights : $W = \{W^{(1)}, \ldots, W^{(L)}\}$

$X \in \mathbb{R}^{N \times D}$, with $\mathbf{x}_i \in \mathbb{R}^D, \forall i \in \{1, \cdots, N\}$

We call D the input dimensionality, and N the number of input samples

$Y \in \mathbb{R}^{N \times C}$, with $\mathbf{y}_i \in \mathbb{R}^C, \forall i \in \{1, \cdots, N\}$

We call C the number of label classes (for classification)

$W^{(i)} \in \mathbb{R}^{|h^{(i-1)}| \times |h^{(i)}|}$

Overall, weight W dimensions can be in the order of several millions!

Deep Nets approximate non-linear functions (*)

- Non-linearities distinguish Deep Nets from Linear Models
  - We can apply the linear model not to $\mathbf{x}$ itself but to a transformed input $\phi(\mathbf{x})$, where $\phi()$ is a non-linear transformation
  - Intuitively, $\phi()$ provides a new set of features describing $\mathbf{x}$, or alternatively, it provides a new representation for $\mathbf{x}$
- The idea in Deep Nets is to **learn** ϕ
  - We can write our model as $\mathbf{y} = \mathbf{f}(\mathbf{x}; \boldsymbol{\theta}; W) = \phi(\mathbf{x}; \boldsymbol{\theta})^\top W$
  - We now have parameters $\boldsymbol{\theta}$ that we use to learn $\phi()$ from a broad class of functions, and parameters W that map from $\phi(\mathbf{x})$ to the desired output.
  - This is an example of a Deep Net, with ϕ defining a single hidden layer.
- Note: there is a connection between infinite basis functions (that is, a Deep Net with infinite width), Kernel methods, and Gaussian Processes

Cost Functions

Motivations

- Objective: learn from observations to approximate the underlying data generating function

- Basic ingredients:
  - Optimization algorithm
  - Cost function
  - Model family

- Deep Nets are non-linear
  - Loss functions become non-convex
  - We use variants of the Gradient Descent algorithm
  - There are weaker guarantees for convergence (see Part 3)
  - Initialization is important

A probabilistic interpretation

- Our parametric model defines a function $p(\mathbf{y}|\mathbf{x}; W)$
  - This is called the likelihood
  - "Probability" of seeing the data, given the input and the model parameters
  - NOTE: the likelihood is un-normalized, so it's not really a probability

- We use maximum likelihood (ML) to compute model parameters
  - We focus on the cross-entropy loss
  - We will also discuss about regularization terms
  - NOTE: A principled approach to overcome the typical overfitting of ML is to use Bayesian Inference

From information to loss functions

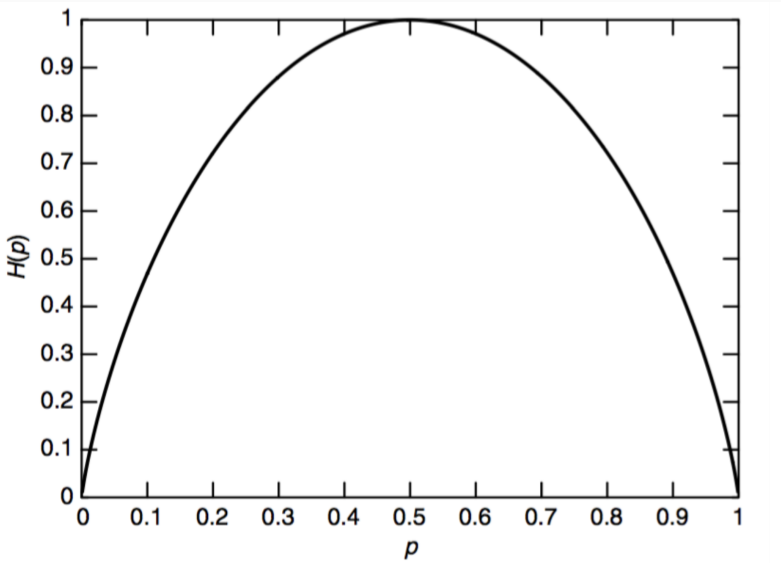
- Here's the outline for what we will see next:
  - Define Entropy
  - Define KL divergence between distributions
  - Define Cross-Entropy
  - Show the ML is equivalent to minimum Cross-Entropy

- We will see some examples applied to Image classification
  - We still do not know how we can output probabilities!

Information and Entropy

- Entropy is a measure of the uncertainty of a random variable
- Consider a discrete r.v., similar expressions for continuous ones
- Given X, with $p(x) = \Pr(X = x)$, we have that:

$$H(X) = -\sum_x p(x) \log p(x) = -\mathbb{E}_{X \sim p(x)} \log p(x) = \mathbb{E}_{X \sim p(x)} \frac{1}{\log p(x)}$$



Entropy vs. p distributed as Bernoulli

Kullback-Leibler Divergence

- Measures the divergence between two distributions

$$\mathrm{KL}\left[p(x)||q(x)\right] = \sum_x p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{X\sim p(x)}\left[\log \frac{p(x)}{q(x)}\right]$$
$$= \mathbb{E}_{X\sim p(x)}\left[\log \frac{1}{q(x)}\right] - \mathbb{E}_{X\sim p(x)}\left[\log \frac{1}{p(x)}\right]$$

Cross-Entropy

- Discrepancy between two distributions
- Say $p(x)$ is the "true" distribution, $q(x)$ is what you get from your Deep Net

$$H\left(p(x), q(x)\right) = \mathbb{E}_{X \sim p(x)}\left[\log \frac{1}{q(x)}\right]$$
$$= \mathbb{E}_{X \sim p(x)}\left[\log \frac{1}{p(x)}\right] + \mathrm{KL}\left[p(x) || q(x)\right]$$

- The Cross-Entropy is larger than the Entropy

ML and Cross-Entropy (1)

- Given $X \sim p(\mathbf{x})$: this is our data, distributed according to an unknown distribution
- Also, given $q(\mathbf{x}|W)$: this is our model, that will output predicted labels

$$W_{\mathrm{ML}} = \arg\max_W q(X|W) = \arg\max_W \prod_{i=1}^{N} q(\mathbf{x}_i|W)$$

$$= \arg\max_W \sum_{i=1}^{N} \log q(\mathbf{x}_i|W) = \arg\max_W \sum_{i=1}^{N} \frac{1}{N} \log q(\mathbf{x}_i|W)$$

$$= \arg\max_W \mathbb{E}_{X\sim p(\mathbf{x})} \log q(\mathbf{x}|W)$$

ML and Cross-Entropy (2)

$$
\begin{aligned}
W_{\mathrm{ML}} &= \arg\max_{W} \mathbb{E}_{X\sim p(\mathbf{x})} \log q(\mathbf{x}|W) \\
&= \arg\min_{W} -\mathbb{E}_{X\sim p(\mathbf{x})} \log q(\mathbf{x}|W) \\
&= \arg\min_{W} \mathbb{E}_{X\sim p(\mathbf{x})} \log \frac{1}{q(\mathbf{x}|W)} \\
&= \arg\min_{W} H(p(\mathbf{x}), q(\mathbf{x}|W))
\end{aligned}
$$

ML and Cross-Entropy: Example

Animal	Encoding
Dog	$[1, 0, 0, 0, 0]$
Fox	$[0, 1, 0, 0, 0]$
Horse	$[0, 0, 1, 0, 0]$
Eagle	$[0, 0, 0, 1, 0]$
Squirrel	$[0, 0, 0, 0, 1]$

- Given some W:

  $p = [1, 0, 0, 0, 0]$

  $q = [0.4, 0.3, 0.05, 0.05, 0.2]$

  $H(p, q) \approx 0.916$

- Given some other W:

  $p = [1, 0, 0, 0, 0]$

  $q = [0.98, 0.01, 0.0, 0.0, 0.01]$

  $H(p, q) \approx 0.02$

Output Units

Output Units

- The choice of output units is tightly coupled with cost functions
  - Often, we use the cross-entropy between the data distribution and the model distribution
  - The choice of how to represent the output determines the form of the cross-entropy function
  - We assume the Deep Net provides a set of hidden features:

$$\mathbf{f}^{(l)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^{\top} W^{(l)}\right)$$

- Types of output units we cover
  - Linear Units for Gaussian Output Distributions
  - Sigmoid Units for Bernoulli Output Distributions
  - Softmax Units for Multinoulli Output Distributions

Linear Units

- Given features $\mathbf{f}^{(l-1)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^\top W^{(l-1)}\right)$
- The last layer l, that is the output layer is

$$\hat{\mathbf{y}} = \mathbf{f}^{(l-1)}(\mathbf{h})^\top W^l$$

- Linear output layers are often used to produce the mean of a conditional Gaussian distribution

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I})$$

- ML is equivalent to minimizing the mean squared error
  - Linear units do not saturate, hence they work well with gradient based optimization

Sigmoid Units

- Sigmoid units are used in binary classification
  - We have a linear layer as before
  - A sigmoid activation function squashes the output in $[0, 1]$
- Given features $\mathbf{f}^{(l-1)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^\top W^{(l-1)}\right)$
- The last layer l, that is the output layer is

$$\hat{y} = \sigma\left(\mathbf{f}^{(l-1)}(\mathbf{h})^\top W^l\right), \quad \sigma(z) = \frac{1}{1+e^{-z}}$$

- The ML approach is to define a Bernoulli distribution over $\mathbf{y}$ conditioned on $\mathbf{x}$

Softmax Units

- Softmax units are used in multi-class classification
  - We have a linear layer as before
  - A softmax functions squashes the output in $[0, 1]$
- Given features $\mathbf{f}^{(l-1)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^\top W^{(l-1)}\right)$
- The last layer l, that is the output layer is

$$\hat{y} = \text{softmax}\left(\mathbf{f}^{(l-1)}(\mathbf{h})^\top W^l\right)$$

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\text{with } \hat{y}_i \in [0, 1] \quad \text{and} \quad \sum_i \hat{y}_i \in [0, 1]$$

- The ML approach is to define a Categorical distribution over **y** conditioned on **x**

Hidden Units

Hidden Units

- The design of hidden units is an active area of research
  - We will see some typical examples
  - More details are in Section 6.3 of the Deep Learning Book
  - Research often proceeds with trial and error

- Differentiable computations
  - During training, we need to compute gradients
  - Since we use SGD to optimize non-convex functions we have no guarantees to reach a global minimum
  - Additionally, we need to consider numerical precision limitations
  - We can settle to piece-wise differentiable units

Rectified Linear Units (RELU) (1)

- Most hidden units apply an element-wise non-linear function to an affine transformation of their inputs

$$\mathbf{f}^{(l)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^{\top} W^{(l)}\right)$$

where:

- $\mathbf{f}^{(l)}(\mathbf{h})$ is the hidden unit at layer l
- $\mathbf{h}$ is the input for the hidden unit
- $W^{(l)}$ are the weights for the linear transformation
- $\mathbf{g}(\cdot)$ is the non-linear function

Rectified Linear Units (RELU) (2)

- When we use RELUs, we set the non-linear function as follows:

$$\mathbf{g}(\mathbf{z}) = \max\{0, \mathbf{z}\}$$

- This is very similar to a linear layer
  - Except that we have a zero value for half domain
  - When the unit is active, the derivatives are large
  - Then, the gradient direction is useful
- The main drawback is that when the activation is zero, weights cannot be learned

Other types of units

- Extensions to RELUs

$$\mathbf{g}(\mathbf{z}, \alpha)_i = \max\{0, z_i\} + \alpha_i \min\{0, z_i\}$$

  - Absolute value rectification
  - Leaky RELUs
  - Parametric RELUs
  - MaxOut Units
- Logistic or Sigmoid non-linear function ($\approx$ output units)
- Radial Basis Functions
- ...

Architecture Design

Architecture Design

- Network architectures refer to structural properties
  - How many layers? $\rightarrow$ Depth
  - How many units per layer? $\rightarrow$ Width
  - How to connect units?
  - Additional "objects": normalization, pooling, ...

- Deep Nets design is an important research topic
  - In some cases, we proceed empirically
  - More recently, architecture search algorithms have emerged

- In general, deeper nets use fewer units per layer and fewer parameters than **wide** but **shallow** nets

Universal Approximation Properties and Depth

- How can we approximate any function underlying the data generation process?
  - Linear models can't
  - Do we need to specialize our model to every kind of non linearity we want to approximate?

- The universal approximation theorem
  - A feedforward network with a linear output layer,
  - at least one hidden layer with any "squashing" function,
  - can approximate any(*) function with a small error,
  - provided that it has enough hidden units

Universal Approximation Caveats: Theory

- Regardless of what function we are trying to learn, a large Deep Net will be able to represent it
  - We have no guarantees that the learning algorithm will **learn** the function
  - Given a function, there exists a feedforward network that approximates it
  - There is no universal procedure that, given a training set, will choose a function that will generalize

- Why can "learning" fail? It's the stochastic optimization algorithm!
  - It might not find the right parameters for the function
  - It might end up overfitting, thus not selecting the right function

Universal Approximation Caveats: Practice

- The universal approximation theorem says that a large enough network exists, but
  - It does not say how large this network will be
  - That is, it is not prescriptive, it's descriptive
  - In the worst case, we might need an exponential number of hidden units

- Infinitely wide Neural Nets (See also the ASI course)
  - A feedforward network with a single layer is sufficient to represent any function
  - But the layer may be infeasibly large and may fail to learn and generalize correctly

- In practice, using deeper models can reduce the number of units and can reduce the generalization error

Architectures: Final Remarks

- In practice, deep architectures are preferred
  - Fewer units, fewer parameters, better generalization
  - Statistical reasons: choosing a good prior

- Architectures beyond chaining of fully connected layers
  - Skip connections, forwarding the input at each layer, additional components
  - Practical examples of widely used architectures:
    ResNet
    VGG
    AlexNet
    Inception

Automatic Differentiation

Automatic Differentiation

- Differentiability and gradient computation are essential
  - Train a Deep Network to find parameters according to ML
  - Use stochastic gradient descent

  $\rightarrow$ The Deep Net "function" has to be differentiable

- How do we compute derivatives of the loss w.r.t. parameters
  - This problem applies to any model
  - Derive analytic expressions (by hand) for gradients

  $\rightarrow$ Today we use automatic differentiation

- Back-Propagation
  - Implementation of automatic differentiation concepts
  - Forward pass to compute output and loss
  - Backward pass to compute gradients, then update weights

Computational Graphs

- Abstract representation of operations in a model
  - Used to decide how to perform backprop
  - Operation "scheduling" for computational efficiency
- Ingredients
  - Node: a variable in our model, i.e. scalar, tensor
  - Edge: an operation on variables
- A note on Operations
  - "Allowed" operations, for which a derivative is known
  - Sub-graphs used to implement arbitrary/complex operations

Chain Rule of Calculus

- How backprop works
  - Uses the chain rule for derivatives
  - Compute the derivatives of functions by composing other functions whose derivatives are known

$$\mathbf{x} \in \mathbb{R}^m, \quad \mathbf{y} \in \mathbb{R}^n$$

$$g : \mathbb{R}^m \to \mathbb{R}^n, \quad f : \mathbb{R}^n \to \mathbb{R}$$

$$\mathbf{y} = g(\mathbf{x}), \quad z = f(\mathbf{y})$$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\mathbf{x}} \right)^\top \nabla_{\mathbf{y}} z$$

- Backprop: Jacobian-gradient product for each operation

Algorithmic considerations

- Backprop algorithm implementation requires thinking about
  - Computational complexity
  - Space complexity
  - Representation: e.g. symbol-to-symbol derivatives

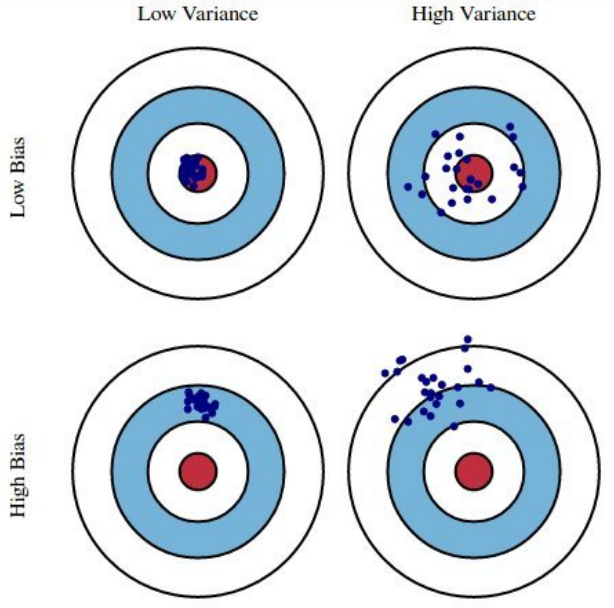
- Recursive application of the chain rule
  - Algebraic expression for the gradient of a scalar (e.g. loss)
  - Operations might be repeated
  - To store or to recompute?

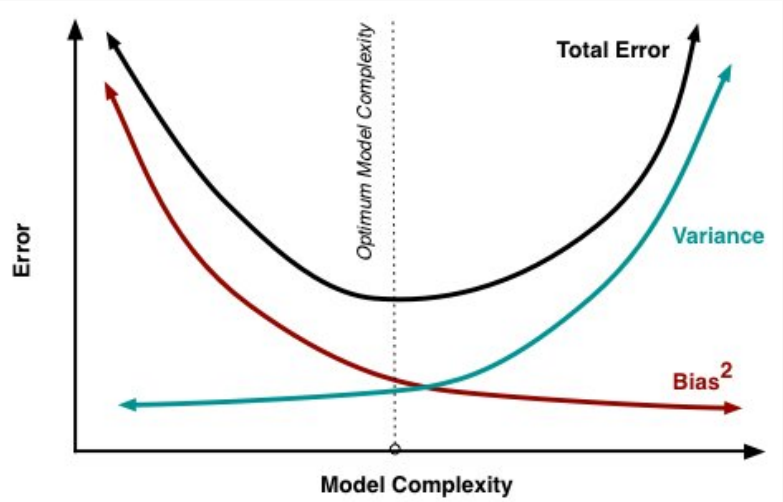
- In summary (ref. 6.5.3 Deep Learninng book)
  - Computational complexity linear in number of edges
  - Computation at each edge: partial derivative, one multiplication, one addition

Regularization

Motivations

- Any modification to a learning algorithm to reduce its generalization error but not its training error
  - Regularization trades increased bias for reduced variance
- The goal of Regularization is to prevent overfitting
  - Put extra constraints on a machine learning model
  - Add extra terms in the objective function
  - Impose ensemble method
  - Add noise





Vector Norms

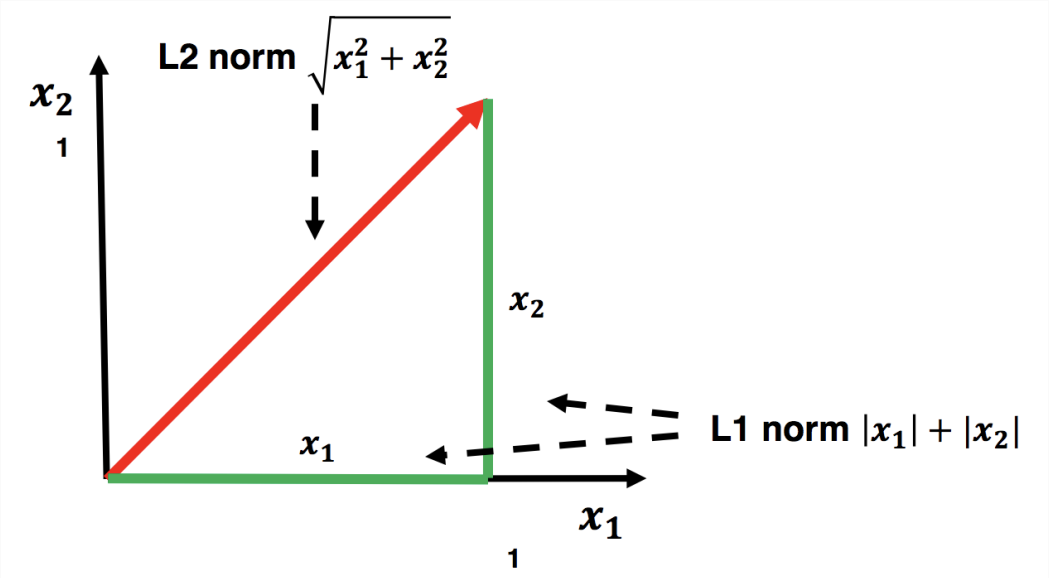
- Definition of norm: any measure of the "size" of a vector

$$||\mathbf{x}||_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

- L1 and L2 norms often used

$$||\mathbf{x}||_1 = \sum_i x_i$$

$$||\mathbf{x}||_2 = \sqrt{\sum_i |x_i|^2}$$



Squared L2 Norm

- Squared L2 norm is used instead of original L2 norm for regularization in machine learning task
  - All of the derivatives depend on the entire vector
  - The derivatives with respect to each element of $\mathbf{x}$ each depend only on the corresponding element of $\mathbf{x}$
- Example: $\mathbf{x} = [x_1, x_2]^\top$

$$||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2}$$

$$\frac{\partial ||\mathbf{x}||_2}{\partial x_1} = x_1(x_1^2 + x_2^2)^{-\frac{1}{2}}$$

$$||\mathbf{x}||_2^2 = x_1^2 + x_2^2$$

$$\frac{\partial ||\mathbf{x}||_2^2}{\partial x_1} = 2x_1$$

Regularization: parameter norm penalties

- Limiting the capacity of models by adding norm penalty

$$\underbrace{\tilde{\mathcal{L}}(W; X, y)}_{\text{Regularized loss}} = \underbrace{\mathcal{L}(W; X, y)}_{\text{Original loss}} + \underbrace{\alpha\Omega(W)}_{\text{Penalty term}}$$

  - $\alpha \to 0$ little regularization
  - $\alpha > 0$ regularization
  - α becomes an hyper parameter

- Doesn't affect inference, penalties applied only to training

- Also known as weight decay

Squared L2 Regularization

- A more compact notation

$$\tilde{\mathcal{L}}(W; X, y) = \mathcal{L}(W; X, y) + \frac{\alpha}{2} W^\top W$$
$$\nabla_W \tilde{\mathcal{L}}(W; X, y) = \nabla_W \mathcal{L}(W; X, y) + \alpha W$$

- Preview: SGD with regularized updates

$$W = (1 - \lambda\alpha) W - \lambda \nabla_W \mathcal{L}(W; X, y)$$

- Note: Only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact

L1 Regularization

- Explicit notation

$$\tilde{\mathcal{L}}(W; X, y) = \mathcal{L}(W; X, y) + \alpha ||W||_1$$
$$\nabla_W \tilde{\mathcal{L}}(W; X, y) = \nabla_W \mathcal{L}(W; X, y) + \alpha \operatorname{sign}(W)$$

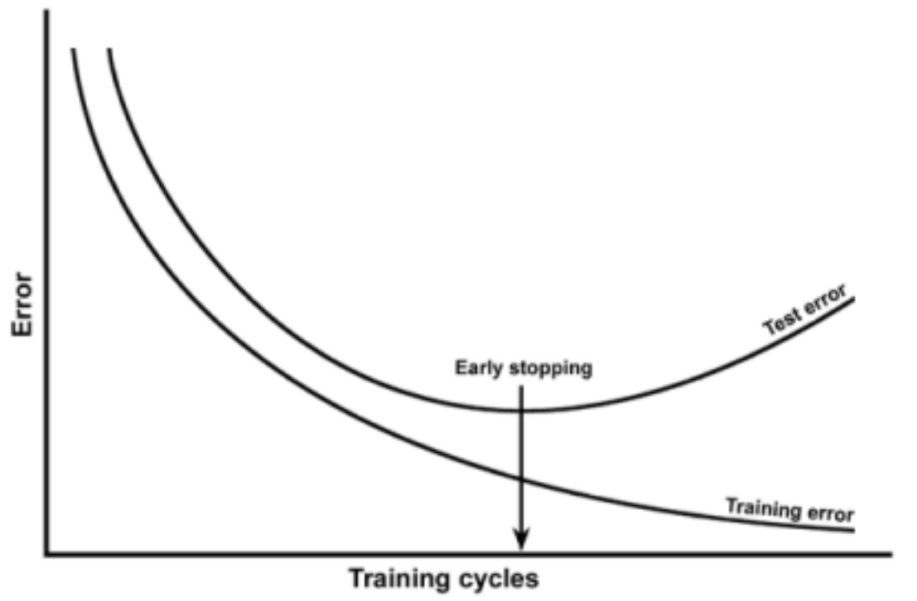
- L1 vs. squared L2 norm regularization
  - L1 is commonly used when the difference between zero and nonzero elements is very important
  - L2 does not cause the parameters to become zero
  - L1 may cause the parameters to become zero for large α

Regularization by Noise Injection

- Data augmentation (not really noise injection)
- Noise injection to improve robustness to perturbations
  - Theoretical understanding often more involved
  - Either see ASI course, or we might cover some aspects later on
- Where do we inject noise?
  - In input samples, e.g. random noise filters
  - In weights of the Deep Net
  - In labels

Regularization by Early Stopping

- Training for too many iterations can causes overfitting



- Early stopping regards iteration budget as hyper parameter and find optimal value of it
  - Can be parallelized
  - Doesn't affect the cost function expression
  - Theoretical understanding can be rather involved

Regularization by Dropout

- A randomized technique suggested by G. Hinton [2012]
  - Hinton, Geoffrey E., et al., "Improving neural networks by preventing co-adaptation of feature detectors", arXiv:1207.0580
- Popularized in the AlexNet architecture
  - Krizhevsky, Alex et al., "Imagenet classification with deep convolutional neural networks", NIPS 2012
  - CNN model with ReLU, Dropout, Data augmentation
  - Applied dropout at Fully Connected layer
- Reinforced by Srivastava
  - Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014
  - Strengthen the theoretical background
  - Extend dropout to convolutional layer

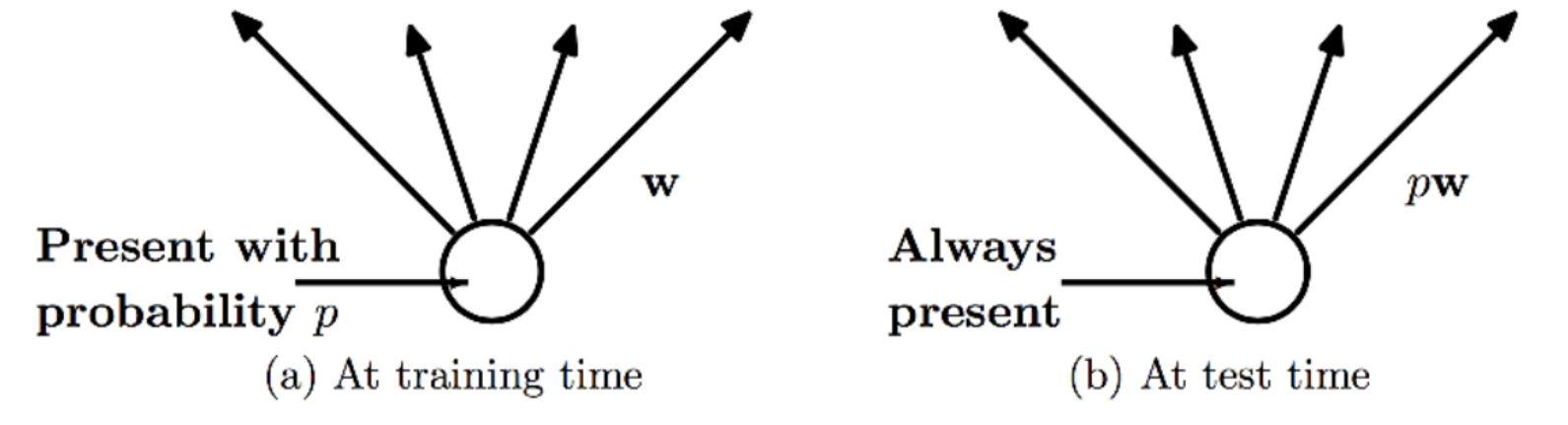
Dropout Basics

- Very simple randomization idea
  - A technique that omits a portion of the network
  - Addresses some limitation of other regularization methods

- Strengths
  - Mimics the voting in an ensemble technique
  - Dropout probability to multiply weights at test time

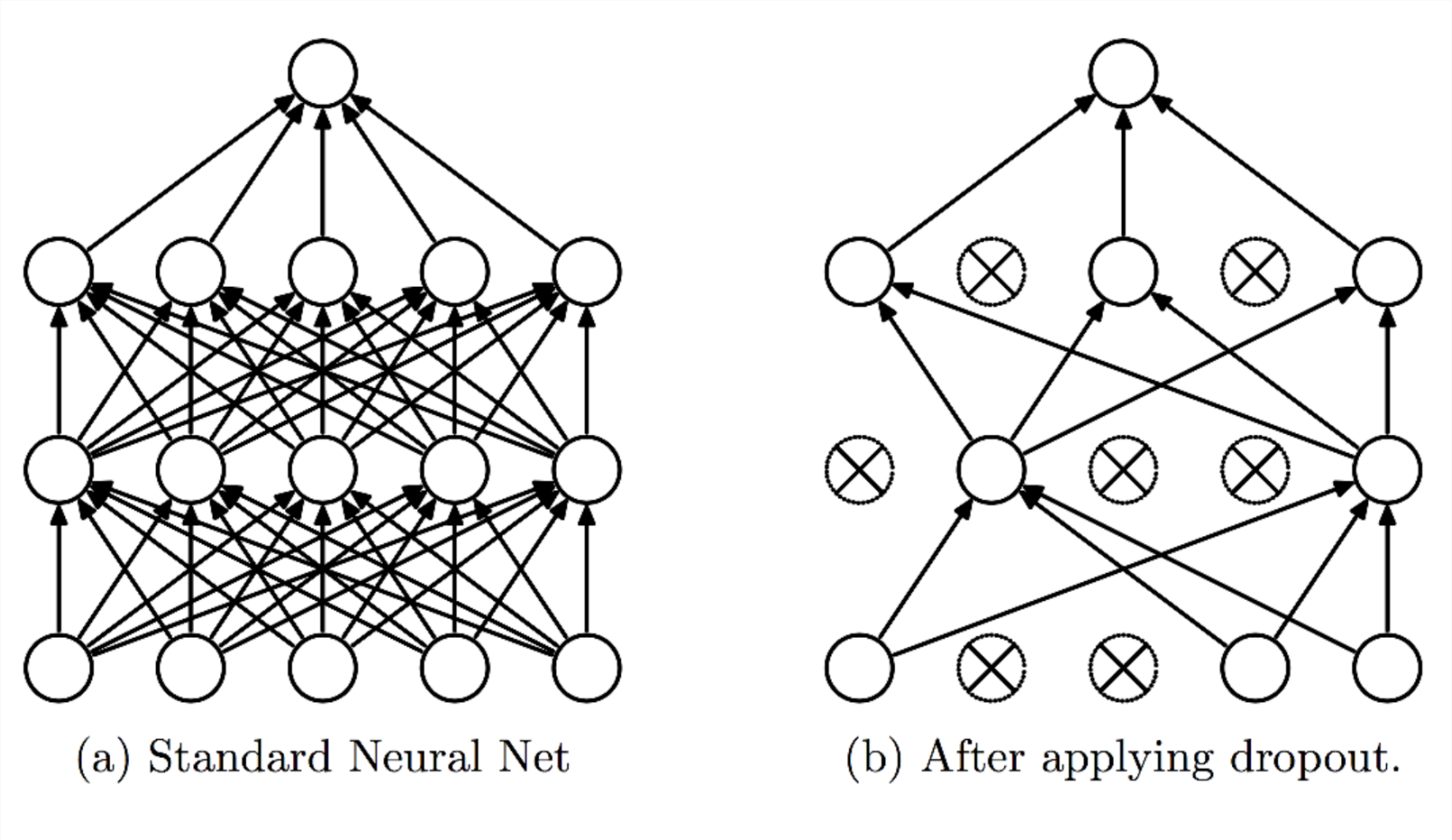
  $\rightarrow$ It is more computational efficient than e.g. bagging

  $\rightarrow$ Avoids co-adaptation: no neurons represent similar features



(a) At training time (b) At test time

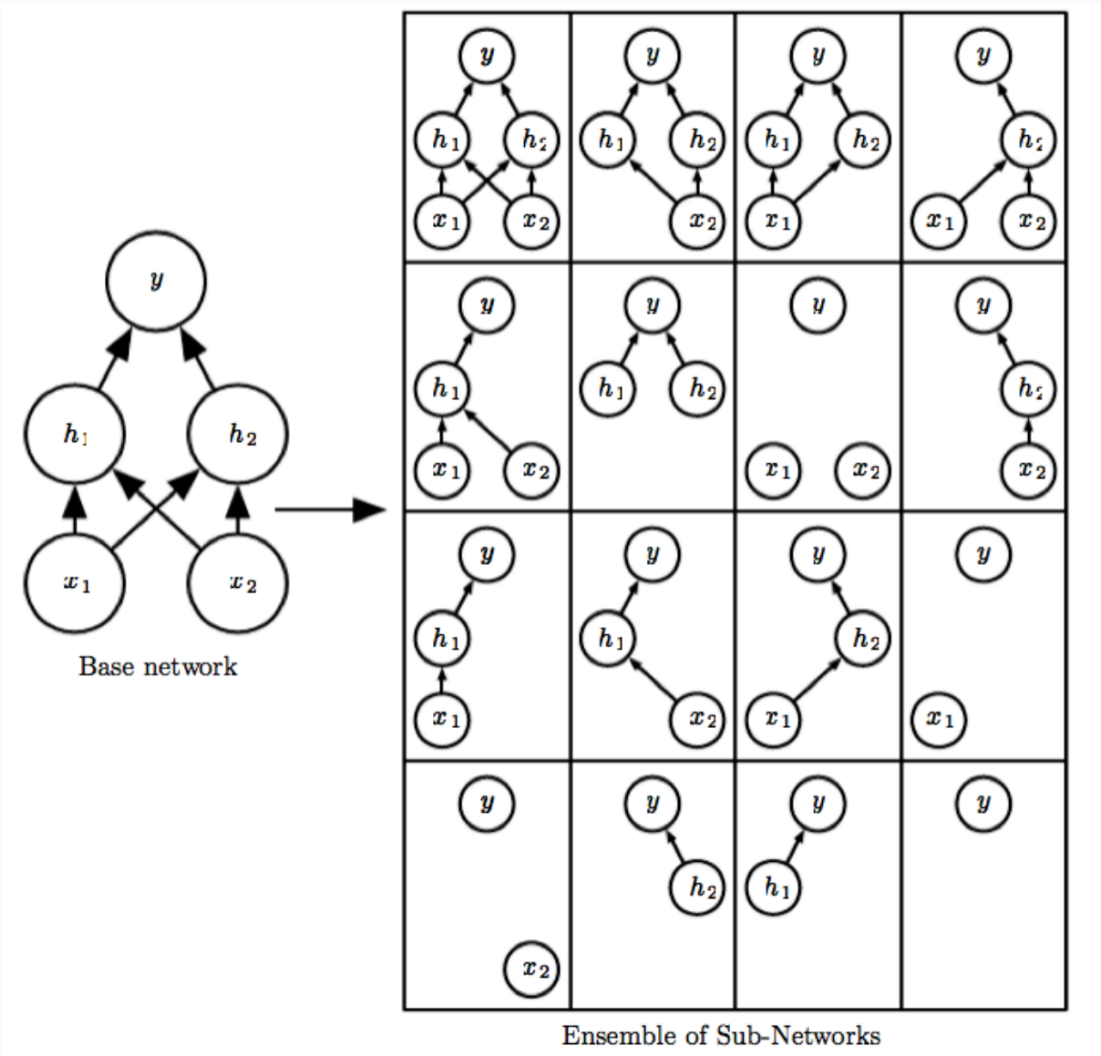
Dropout Illustration (1)



(a) Standard Neural Net

(b) After applying dropout.

Dropout Illustration (2)



Expressing Dropout in Math

$$\mathbf{f}(\mathbf{x}, W) = \mathbf{f}^{(L)}\left(\mathbf{f}^{(L-1)}\left(\cdots \mathbf{f}^{(1)}\left(\mathbf{x}\right)\cdots\right)\right)$$

$$\mathbf{f}^{(l)}(\mathbf{h}) = \mathbf{g}\left(\left(\mathbf{h}^{\top} W^{(l)}\right) \odot \mathbf{m}^{(l)}\right)$$

- We term $\mathbf{m}^{(l)}$ the dropout mask applied at layer l

$$\mathbf{m}_i^{(l)} \sim \text{Bernoulli}(p)$$

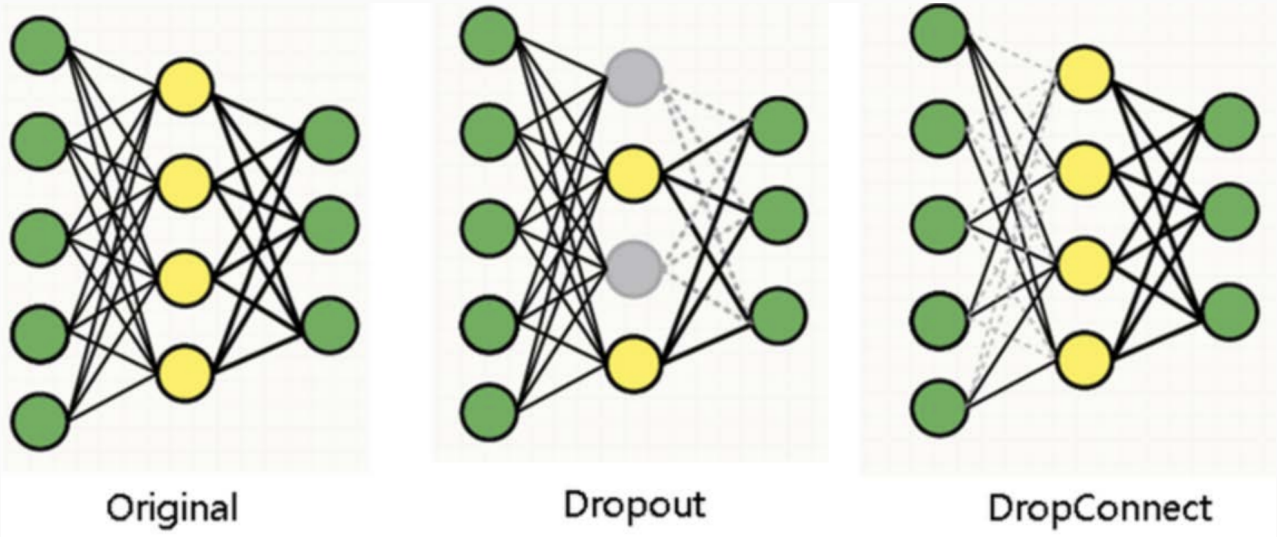
- In other words, the mask is a random vector of 0s and 1s
- p is the dropout probability, whereby we can chose
  - Fixed number of units, variable p
  - Fixed value of pn, with n number of units

Regularization by DropConnect

- DropConnect generalizes dropout
  - Wan, Li, et al. "Regularization of neural networks using dropconnect." ICML 2013
  - Dropout omits all connections to dropped units
  - DropConnect only omits connections, all units are alive

$$\mathbf{f}(\mathbf{x}, W) = \mathbf{f}^{(L)}\left(\mathbf{f}^{(L-1)}\left(\cdots \mathbf{f}^{(1)}(\mathbf{x})\cdots\right)\right)$$

$$\mathbf{f}^{(l)}(\mathbf{h}) = \mathbf{g}\left(\mathbf{h}^{\top}\left(W^{(l)} \odot \mathbf{M}^{(l)}\right)\right), \quad \mathbf{M}^{(l)}_{i,j} \sim \text{Bernoulli}(p)$$



Loss Landscapes (*)

Motivations and Questions

Why studying losses?

- The "trainability" of Deep Nets is hard to understand
  - Highly dependent on network architecture design choices
  - The choice of optimizer affects efficiency
  - Parameter initialization is crucial
  - Many other considerations, e.g. learning rate schedule, ...
- Training Deep Nets requires minimizing a high-dimensional non-convex loss function
  - Hard in theory, sometimes easy in practice
  - Simple gradient methods often find global minimizers
- Important questions
  - Why can we minimize highly non-convex loss functions?
  - Why do the resulting minima generalize?
  - How different architectures affect the loss landscape?
  - How the geometry of minimizers (sharpness/flatness) affects generalization?

Methodology

- The material from this section is from

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

- You can use the following link for an interactive 3D visualization of some pre-built network architectures

  `http://www.telesens.co/loss-landscape-viz/viewer.html`

- IMPORTANT: take with a grain of salt the following results!

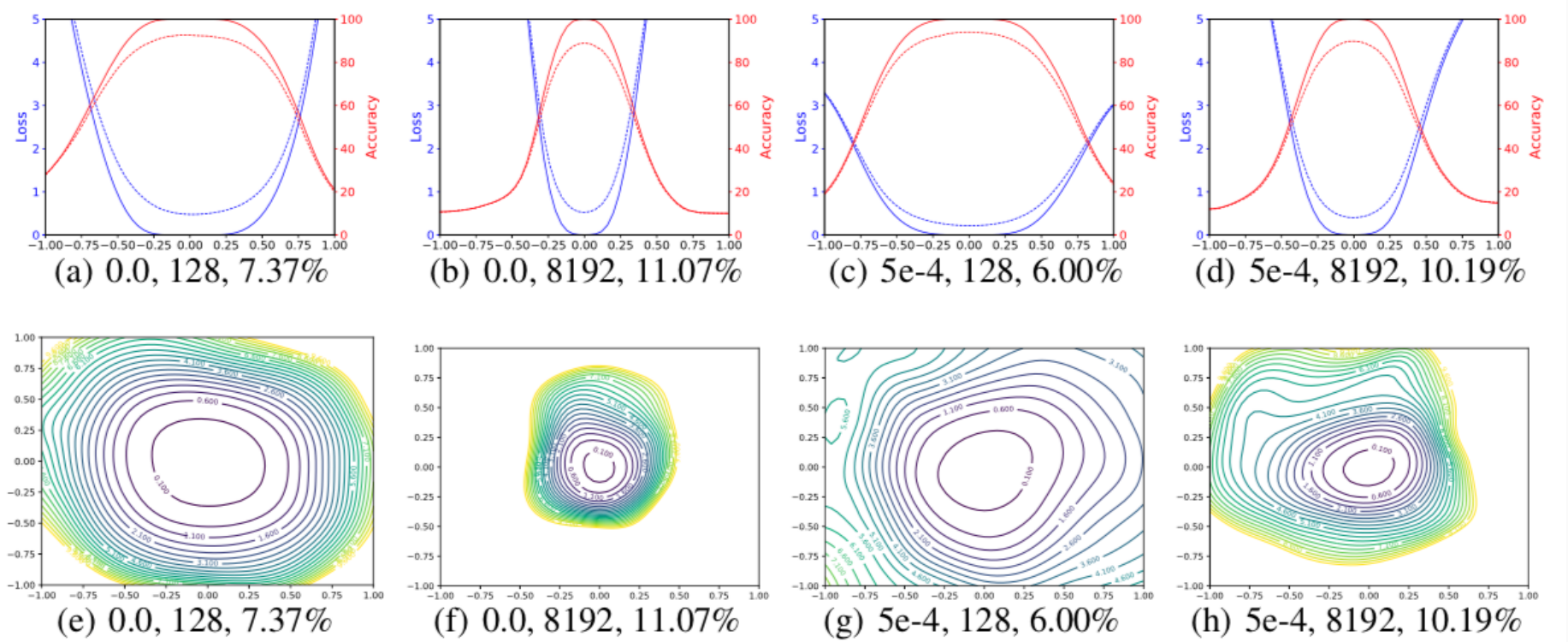
The Sharp vs. Flat Dilemma

Shall We Prefer Sharp or Flat minimizers?

- Do sharp minimizers generalize better than flat ones?
  - Previous studies: small-batch SGD produces "flat" minimizers that generalize well, while large batches produce "sharp" minima with poor generalization
  - Recent studies: generalization is not directly related to the curvature of loss surfaces

- Using appropriate visualization tools
  - Large batches produce visually sharper minima (although not dramatically so)
  - Sharper minima seems to be correlated with higher test error

Shall We Prefer Sharp or Flat minimizers?



Weight decay; Batch size; test error

CIFAR-10 classifier using a 9-layer VGG network

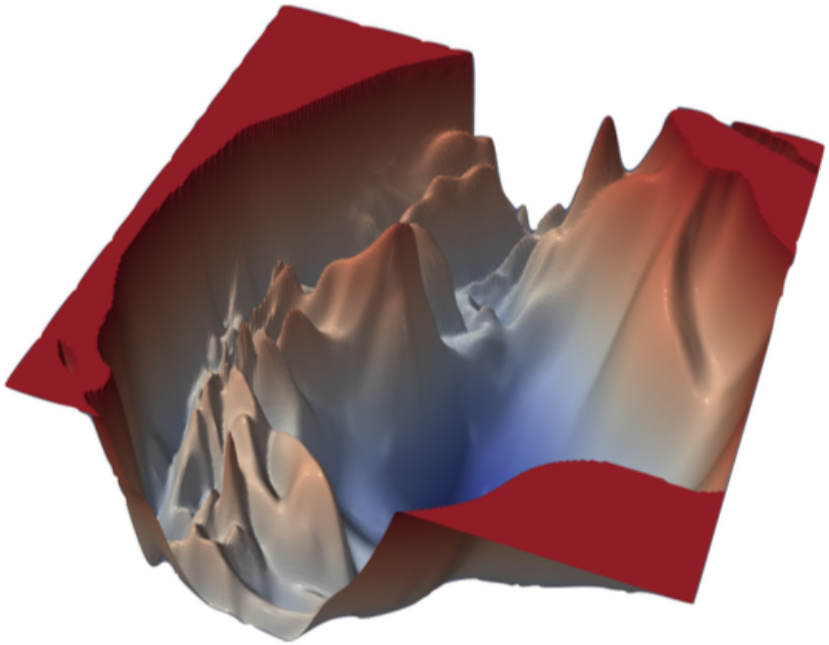
The Structure of the Loss Function

What makes Deep Nets Trainable?

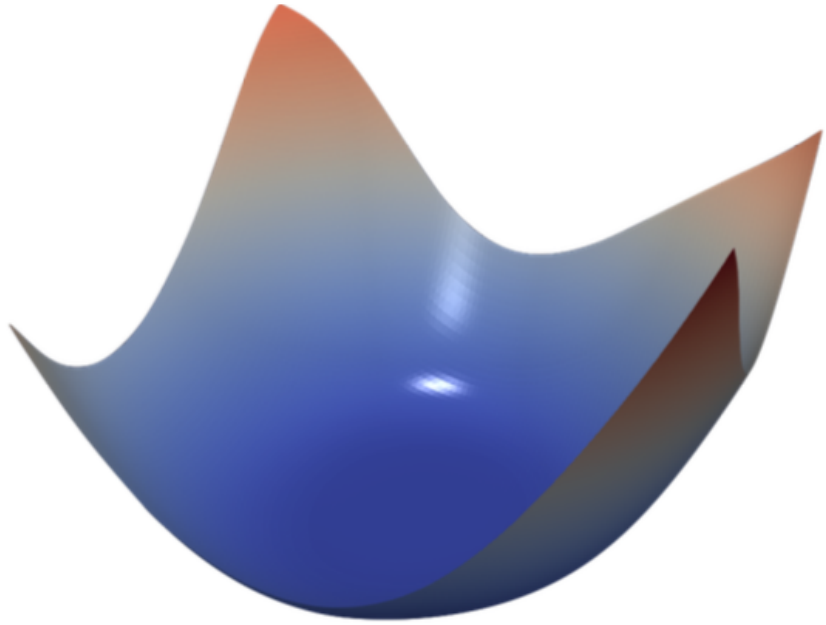
- Easily finding loss minimizers is not universal
  - Some neural architectures are easier to minimize than others
  - Our ability to train seems to depend strongly on initialization

- Questions
  - Do loss functions have significant non-convexity at all?
  - If prominent non-convexities exist, why are they not problematic in all situations?
  - Why are some architectures easy to train?
  - Why are results so sensitive to the initialization?

Impact of Network Architecture



(a) ResNet-110, no skip connections

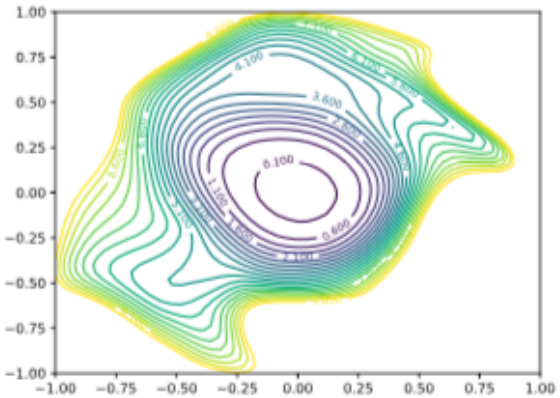


(b) DenseNet, 121 layers

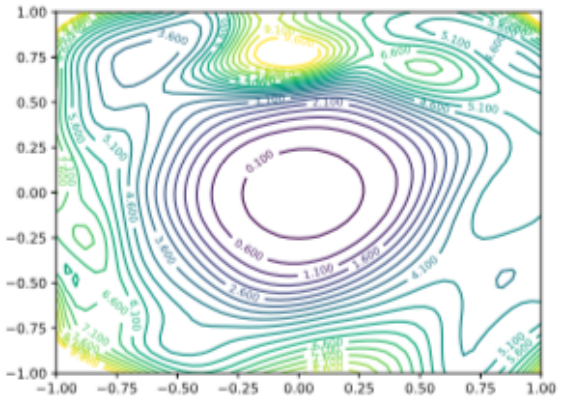
The Effect of Depth

- Network depth has a dramatic effect on the loss surfaces
  - This is especially true when skip connections are not used
  - Skip connections are useful for deeper architectures
- Forward reference: what are skip connections?
  - We forward the input of a layer to the next layer
  - That is, we can skip some hidden layers
- From (nearly) convex to chaotic landscapes
  - As network depth increases, VGG-like nets transition
  - Deep nets without skip connections are highly non-convex
  - Regions where gradients do not point to minimizers
  - Loss function can become extremely large

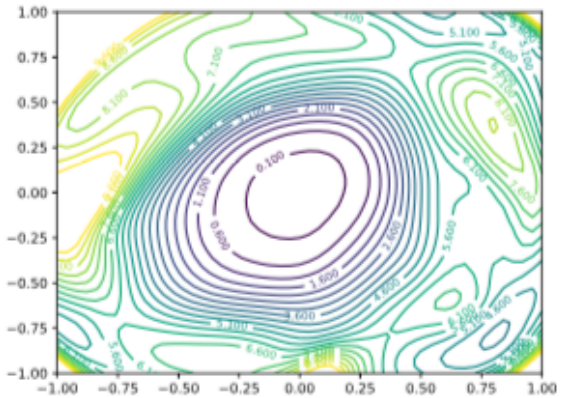
Shortcut Connections to the Rescue



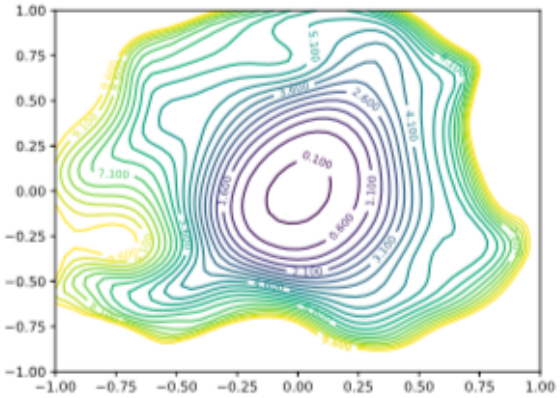
(a) ResNet-20, 7.37%



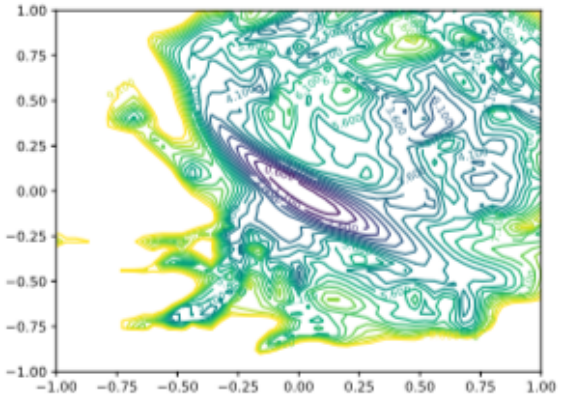
(b) ResNet-56, 5.89%



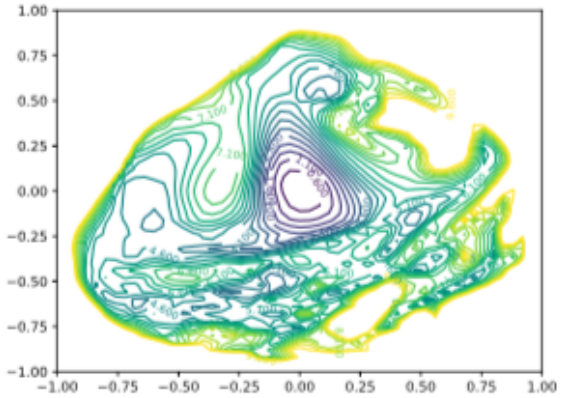
(c) ResNet-110, 5.79%



(d) ResNet-20-NS, 8.18%



(e) ResNet-56-NS, 13.31%



(f) ResNet-110-NS, 16.44%

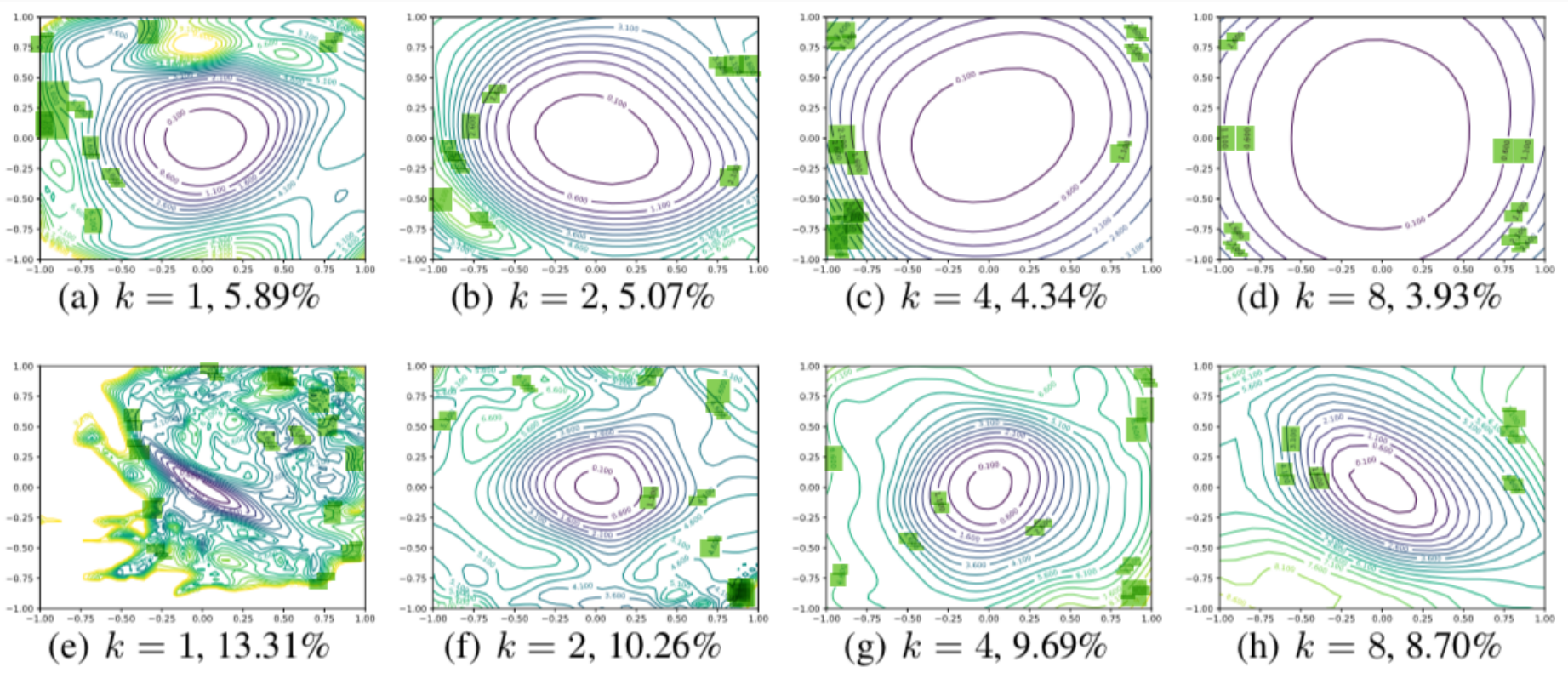
The Effect of Depth

- Number of (filter) units per layer is important too
  - Well studied in theory for simple feed forward nets
  - Empirical studies for convolutional filters

- Take home messages
  - Wider models are well behaved, no chaotic loss landscape
  - Flat minima with wide convex regions
  - Minimizer sharpness correlates with test error
  - Shortcuts very useful
  - If no shortcuts are used, width should increase

WARNING: wider $\Rightarrow$ more weights $\Rightarrow$ computationally expensive

Wider Layers to the Rescue



(a) $k = 1$, 5.89% (b) $k = 2$, 5.07% (c) $k = 4$, 4.34% (d) $k = 8$, 3.93%

(e) $k = 1$, 13.31% (f) $k = 2$, 10.26% (g) $k = 4$, 9.69% (h) $k = 8$, 8.70%

Top: skip connections, Bottom: no skip connections

k: multiplicative factor for number of filters per layer

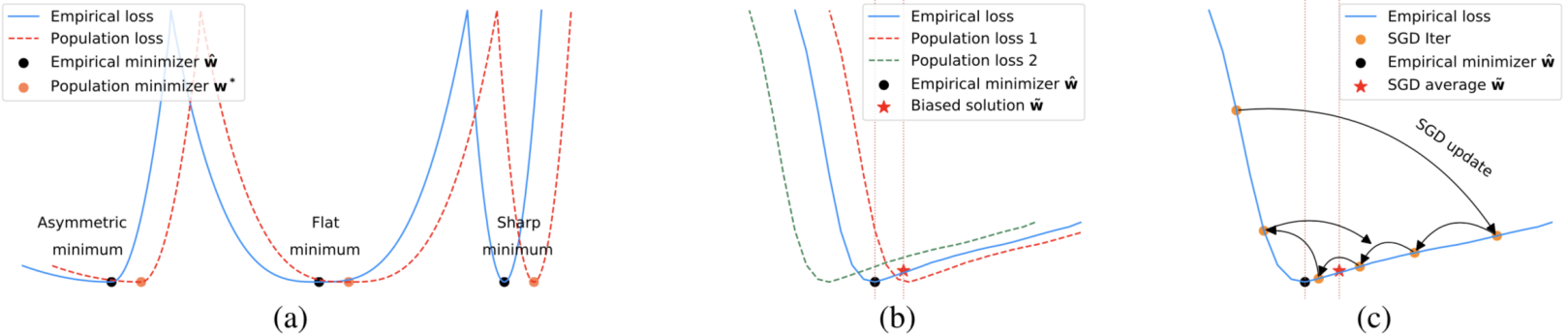
Final Considerations (1)

- Additional aspects to be considered
  - Initialization, i.e., starting point in the landscape, is crucial
  - Combined effects of architecture, optimizer and initialization only partially understood

- Cautionary notes
  - Previous figures obtained under dramatic dimensionality reduction
  - Compute principal curvatures using eigenvalues of the Hessian
  - Convex or near-convex loss when Hessian is positive semi-definite
  - Principal curvatures of a dimensionality reduced plot correspond to weighted average on the full surface

Final Considerations (2)

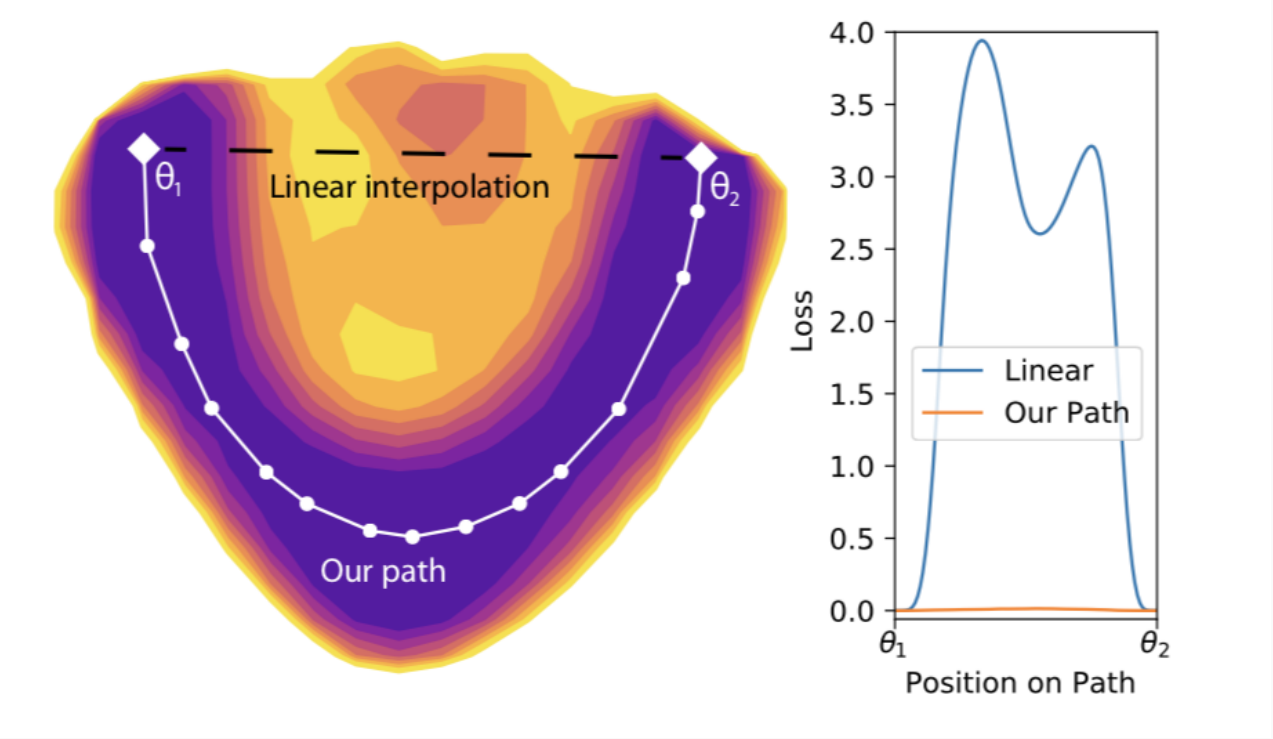
- Loss surfaces have **Asymmetric Valleys**



(a) Three kinds of local minima: asymmetric, flat and sharp. If there exists a shift from empirical loss to population loss, flat minimum is more robust than sharp minimum. **(b)** For asymmetric valleys, if there exists a random shift, the solution $\tilde{\boldsymbol{w}}$ biased towards the flat side is more robust than the minimizer $\hat{\boldsymbol{w}}^*$. **(c)** SGD tends to stay longer on the flat side of asymmetric valleys, therefore SGD averaging automatically produces the desired bias.

Final Considerations (3)

- Loss minimizers are connected through a path



Stochastic Optimization

Motivations and Challenges

Focus on theoretical results:

Stochastic gradient methods for machine learning

Francis Bach, ENS

`https://www.di.ens.fr/~fbach/fbach_mlss_2018.pdf`

Optimization for training Deep Models

- (Deep) Learning involves optimization algorithms in many contexts
  - Huge investments in terms of hardware and time
  - Statistical and computational efficiency are key

- Our focus is on computing the parameters W of a Deep Net
  - We do so by minimizing a loss function $\mathcal{J}(X, \mathbf{y}, W)$
  - Typically we include regularization terms

- What is the difference between optimization and learning?
  - Explicit vs. implicit optimization
  - Challenges related to loss surfaces

Explicit vs. implicit optimization

- Machine learning algorithms act indirectly
  - Performance measure P defined on the **test set**
  - Training happens on the **training set**

- Expected generalization error vs. Empirical risk

$$\mathcal{J}^*(W) = \mathbb{E}_{(X,\mathbf{y})\sim p_{\text{data}}} \mathcal{L}\left(f(X, W), \mathbf{y}\right)$$

$$\mathcal{J}(W) = \mathbb{E}_{(X,\mathbf{y})\sim \hat{p}_{\text{data}}} \mathcal{L}\left(f(X, W), \mathbf{y}\right) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f(\mathbf{x}_i, W), \mathbf{y}_i\right)$$

- Empirical risk minimization
  - Like traditional optimization
  - We hope that this will generalize

Challenges related to loss surfaces

- Differentiability
  - Some loss functions might not be differentiable, e.g. 0-1 loss
  - Minimize surrogate function, e.g. negative log likelihood
  - Halt conditions based on early stopping rather than (local) optimizer

  $\rightarrow$ Gradients might not be zero

Computational challenges

- Computational cost
  - Finite sum loss functions

$$\mathbb{E}_{(X,\mathbf{y})\sim\hat{p}_{\text{data}}} \log p_{\text{model}}\left(\mathbf{y}|X, W\right)$$

$$\Leftrightarrow$$

$$W_{\text{ML}} = \text{argmax}_W \sum_{i=1}^{n} \log p_{\text{model}}\left(\mathbf{y}_i|\mathbf{x}_i, W\right)$$

  - Unbiased estimators

$$\nabla_W \mathcal{J}(W) = \mathbb{E}_{(X,\mathbf{y})\sim\hat{p}_{\text{data}}} \nabla_W \log p_{\text{model}}\left(\mathbf{y}_i, \mathbf{x}_i, W\right)$$

$\rightarrow$ We can use a subset of the training set, called a mini-batch

Additional Challenges (1)

Ill-conditioning

$$f(\mathbf{x}) \approx$$
$$f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \nabla f(\mathbf{x}^{(0)}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(0)})^\top H(\mathbf{x} - \mathbf{x}^{(0)})$$
$$f(\mathbf{x}^{(0)} - \lambda \nabla f(\mathbf{x}^{(0)})) \approx$$
$$f(\mathbf{x}^{(0)}) - \lambda \nabla f(\mathbf{x}^{(0)})^\top \nabla f(\mathbf{x}^{(0)}) + \frac{1}{2}\lambda^2 \nabla f(\mathbf{x}^{(0)})^\top H \nabla f(\mathbf{x}^{(0)})$$

If $\frac{1}{2}\lambda^2 \nabla f(\mathbf{x}^{(0)})^\top H \nabla f(\mathbf{x}^{(0)}) > \nabla f(\mathbf{x}^{(0)})^\top \nabla f(\mathbf{x}^{(0)})$

- Learning is slow despite a strong gradient
- λ must be shrunk to compensate for stronger curvature

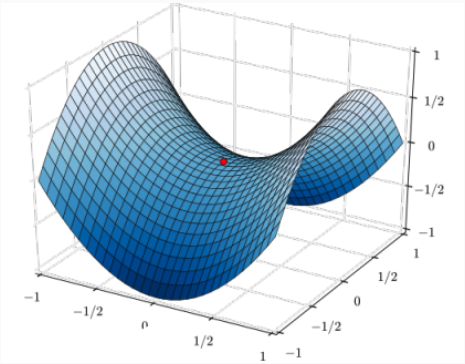
Additional Challenges (2)

- Model identifiability
  - A sufficiently large training set can rule out all but one setting of the model's parameters
- Loss surfaces for Deep Nets are non-convex
  - Many (eventully infinite) equivalent local minima
  - Model is not identifiable: weight space symmetry
- Approaches, but this is still "research"
  - Chose better initialization
  - Check decrease of gradient norm

Additional Challenges (3)

- Saddle points
  - Gradient is zero
  - Hessian matrix has both positive and negative eigenvalues
  - High-dimensional spaces: saddles are more frequent than local minima

- What are the implications?
  - First order methods seem "immune": SGD escapes saddle points
  - Second order methods suffer more, but there are "patches"



Additional Challenges (4)

- Cliffs and exploding gradients
  - Loss surfaces can induce large gradients
  - Heuristic: gradient clipping, as we're interested in gradient direction

- Exploding, vanishing gradients
  - Occur for deep architectures
  - E.g.: repeated multiplication by W
    Assume we have the eigendecomposition:

$$W = V\mathrm{diag}(\lambda)V^{-1}$$
$$W^t = V\mathrm{diag}(\lambda)^t V^{-1}$$

  - If λ_i are not around 1, they either explode or vanish
  - Difficult to use gradient directions

Basic Algorithms and Extensions

Disclaimer

- Quick overview of the basic algorithms
  - Focus on update rules, not full pseudo-code
- Quick overview of recent extensions
  - Focus on automatic tuning of optimization parameters
- Theoretical understanding out of scope
  - Very important and active area of research
  - Some notions discussed in the Optim course
  - Convergence rate is not the only interesting topic
  - Connections with sampling methods and Bayesian inference
- Notation warning!

Finite-sum Non-convex Loss Functions

$$\mathcal{J}(W, X, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f(\mathbf{x}_i, W), \mathbf{y}_i\right)$$

where:

W: the Deep Net weights (a.k.a. θ)

$\mathbf{x}_i$: individual data sample, the input

$\mathbf{y}_i$: associated label or response, the output

$i \in \{1, \cdots, n\}$, index for the training set of n total samples

The basics: λ = Learning Rate

- Gradient Descent: all samples

$$W_{t+1} = W_t - \lambda_t \nabla \mathcal{J}(W_t, X, \mathbf{y})$$

- Stochastic Gradient Descent (SGD): one sample at the time

$$W_{t+1} = W_t - \lambda_t \nabla \mathcal{J}(W_t, \mathbf{x}_i, \mathbf{y}_i)$$

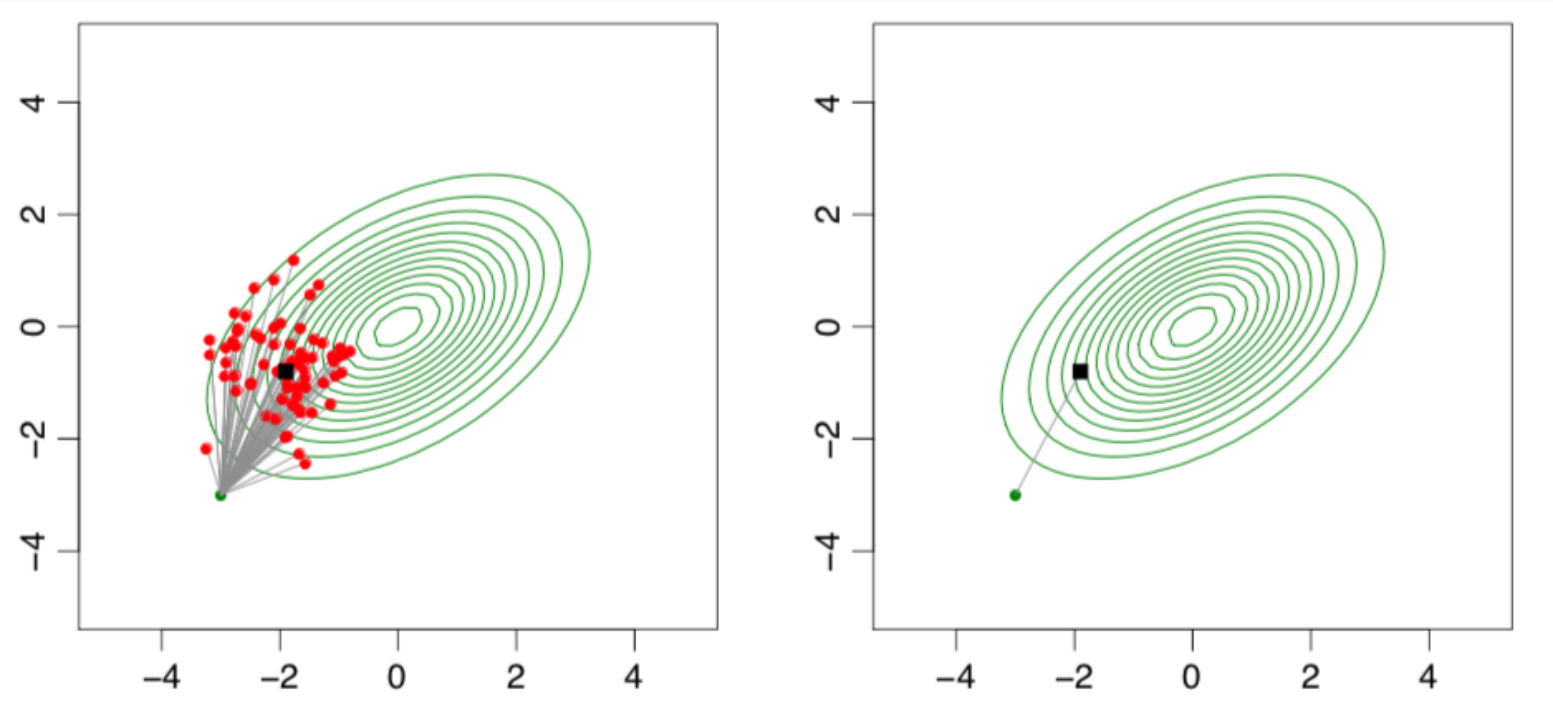
- Mini-batch Stochastic Gradient Descent:
  a randomized (mini) batch m of samples

$$W_{t+1} = W_t - \lambda_t \frac{1}{m} \nabla \sum_{i=1}^{m} \mathcal{J}(W_t, \mathbf{x}_i, \mathbf{y}_i)$$

Unbiased Estimator

$$\mathbf{g}_t = \frac{1}{m} \nabla \sum_{i=1}^{m} \mathcal{J}(W_t, \mathbf{x}_i, \mathbf{y}_i)$$

$$\mathbb{E}_{X,\mathbf{y}} [\mathbf{g}_t] = \nabla \mathcal{J}(W_t, X, \mathbf{y})$$



Robbins-Monro Conditions on λ

H. Robbins, S. Monro, "A stochastic approximation method.", Annals of mathematical statistics, 1951

- Sufficient conditions to guarantee convergence of SGD

$$\sum_{i=1}^{\infty} \lambda_t = \infty$$

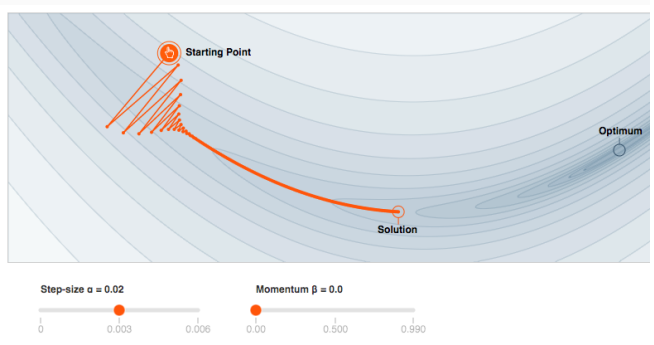
$$\sum_{i=1}^{\infty} \lambda_t^2 < \infty$$

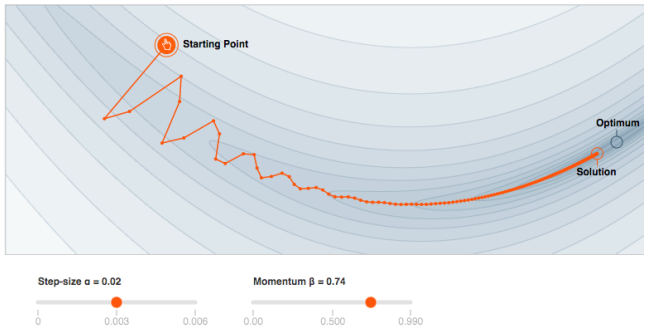
SGD with Momentum (1)

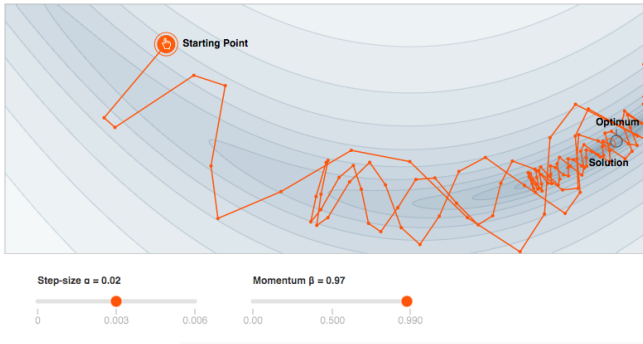
- Momentum is designed to accelerate learning
  - Accumulates an exponentially decaying moving-average of past gradients
  - Continues to move in their direction
  - Momentum allows a larger range of step-sizes to be used, and creates its own oscillations

- Why Momentum Really Works

  `https://distill.pub/2017/momentum/`







SGD with Momentum (2)

$$\mathbf{g}_t = \frac{1}{m} \nabla \sum_{i=1}^{m} \mathcal{J}(W_t, \mathbf{x}_i, \mathbf{y}_i)$$

$$\mathbf{v}_{t+1} = \alpha \mathbf{v}_t + \lambda \mathbf{g}_t$$

$$W_{t+1} = W_t - \mathbf{v}_{t+1}$$

- $\mathbf{v}$: direction and speed at which W move
- Hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay
  - Larger α w.r.t. λ, previous gradients count more
  - In SGD, step size was $||\mathbf{g}||$ multiplied by λ
  - Now, step size depends on how large and how aligned a sequence of gradients are
  - Easier to think of $\frac{1}{1-\alpha}$

Adaptive Algorithms

AdaGrad

- Similar to SGD, but with per-parameter learning rate
  - Increases λ for parameters with small gradient
  - Decreases λ for parameters with large gradient

$$\mathbf{g}_t = \frac{1}{m} \nabla \sum_{i=1}^{m} \mathcal{J}(W_t, \mathbf{x}_i, \mathbf{y}_i)$$

$$\mathbf{r}_{t+1} = \mathbf{r}_t + \mathbf{g}_t \odot \mathbf{g}_t$$

$$W_{t+1} = W_t - \frac{\lambda}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}_t$$

where δ is a small constant

RMSProp

- Like AdaGrad, but with exponentially weighted moving average

$$\mathbf{g}_t = \frac{1}{m} \nabla \sum_{i=1}^{m} \mathcal{J}(W_t, \mathbf{x}_i, \mathbf{y}_i)$$

$$\mathbf{r}_{t+1} = \rho\, \mathbf{r}_t + (1-\rho)\mathbf{g}_t \odot \mathbf{g}_t$$

$$W_{t+1} = W_t - \frac{\lambda}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}_t$$

where $0 < \rho < 1$

- In AdaGrad accumulation of squared gradients $\Rightarrow$ excessive decrease of learning rate
- RMSProp is better in non-convex settings

Adam

- RMSProp + Momentum
- Accumulates first and second order estimates

$$\mathbf{s}_{t+1} = \rho_1 \mathbf{s}_t + (1 - \rho_1)\mathbf{g}_t$$
$$\mathbf{r}_{t+1} = \rho_2 \mathbf{r}_t + (1 - \rho_2)\mathbf{g}_t \odot \mathbf{g}_t$$

- Corrects bias

$$\hat{\mathbf{s}} = \frac{\mathbf{s}_t + 1}{1 - \rho_1^t} \quad \hat{\mathbf{r}} = \frac{\mathbf{r}_t + 1}{1 - \rho_2^t}$$

- Applies the parameter update

$$W_{t+1} = W_t - \lambda \frac{\hat{\mathbf{s}}}{\delta + \sqrt{\hat{\mathbf{r}}}}$$

How to Chose an Optimization Algorithm

- As of today, there's no consensus
- Trial and error
- Good luck!

Parameter Initialization

How to Choose Appropriate Parameter Initialization

- Determining appropriate initial parameters for a deep model is one of the most important part of optimization
- Initial parameters can determine
  - Whether the algorithm converges or not
  - Whether the model converges to a point with a high or low cost
  - How quickly learning converges
- Initialization strategies are still heuristic
  - Random initialization
  - Samples from Gaussian or Uniform Distributions

Initialization with Appropriate Scale

- What happens with the scale of parameters
  - If too small $\Rightarrow$ signal shrinks as it passes through each layer
  - If too large $\Rightarrow$ signal explodes as it passes through each layer

- Heuristics to address vanishing or exploding gradients
  - Xavier/Glorot Initialization
  - He-Normal Initialization

Xavier/Glorot Initialization

- Initialization with zero mean and 0.01 standard deviation Gaussian noise very popular
  - Works well in many situations
  - Doesn't work with deeper architectures, e.g. modern ConvNets
  - Activation (and/or) gradient magnitude in final layers is problematic
    - If each layer scales input by k, the final scale would be k^L, where L is the number of layers
    - $k > 1 \Rightarrow$ exploding output values
    - $k < 1 \Rightarrow$ vanishing output values
- Xavier init estimates the std dev to be used for the Gaussian
  - Based on the number of input/output channels of each layer
  - Under the assumption of no non-linearity

$$\mathrm{Var}(W^{(l)}) = \frac{1}{\mathrm{Fan\_in} + \mathrm{Fan\_out}}$$

He-Normal Initialization

- Similar in principle to Xavier
  - Estimates the std dev for the Gaussian init
  - Weights initialized keeping in mind the size of the previous layer
  - Helps attaining a better minimum of the cost function, faster and more efficiently
  - Works for RELU activations

$$\mathrm{Var}(W^{(l)}) = \frac{2}{\mathrm{Fan\_in}}$$

- Intuitively: a RELU is zero for half of its input, so you need to double the size of weight variance to keep the signal's variance constant

[From our partners in CTU Prague!] Dmytro Mishkin, Jiri Matas, "All you need is a good init", ICLR 2016

Normalization Methods

Motivations and Misconceptions

Based on:

Santurkar, S. et al, "How Does Batch Normalization Help Optimization?", NIPS, 2018

Why Batch Normalization?

- A practical, important technique for Deep Learning
  - Improves training by stabilizing input distributions of each layer
  - Additional layers that control the first two moments of these distributions

- Original motivations
  - Remedy to internal covariate shift (ICS)
  - ICS: change in the input distribution of each layer caused by updates to preceding layers
  - Conjecture: such continual change negatively impacts training

- Our current understanding
  - ICS effects are negligible
  - BatchNorm makes the loss landscape significantly more smooth

  $\rightarrow$ Gradients are more predictive

  $\rightarrow$ Larger range of learning rates

  $\rightarrow$ Faster convergence

BatchNorm and Internal Covariate Shift

BatchNorm in a Nutshell

- Sets mean and variance of each activation to be zero and one
- Batch norm inputs scaled and shifted based on trainable parameters
- Applied before the non-linearity of the previous layer
- i: sample index, k: dimension index of a sample

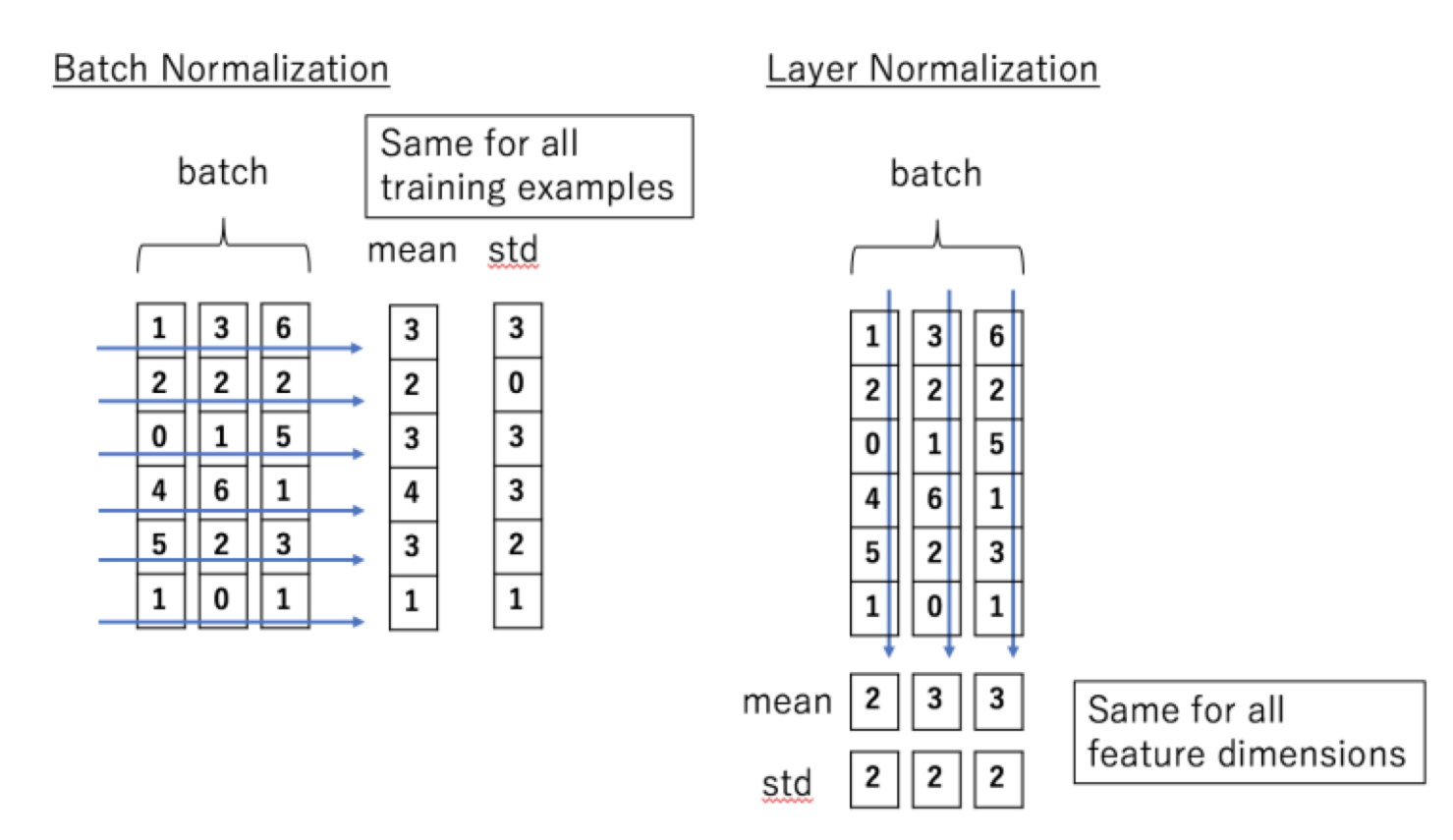
$$\mu_B^{(k)} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i^{(k)}$$

$$\sigma_B^{(k)} = \frac{1}{m}\sum_{i=1}^{m} \left(\mathbf{x}_i^{(k)} - \mu_b^{(k)}\right)^2$$

$$\hat{\mathbf{x}_i}^{(k)} = \frac{\mathbf{x}_i^{(k)} - \mu_B^{(k)}}{\sqrt{{\sigma^{(k)}}_B^2 + \epsilon}}$$

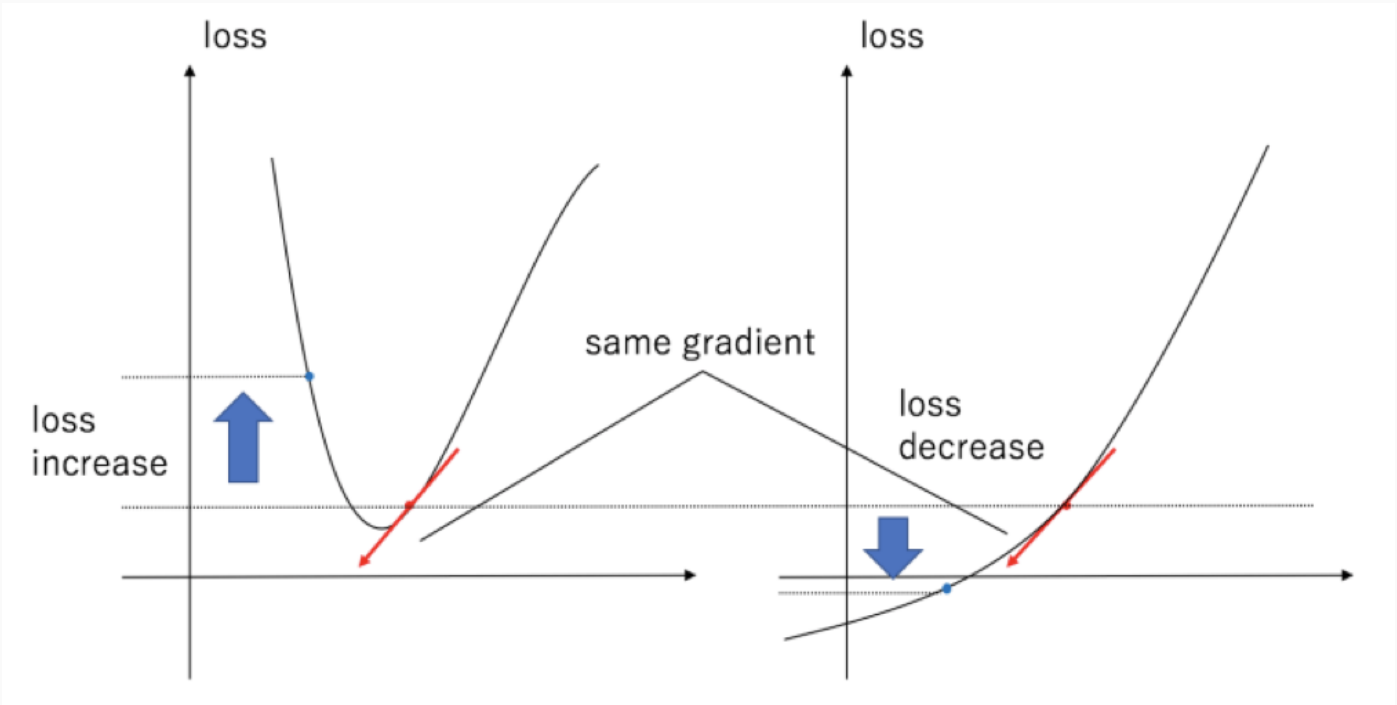
$$\mathbf{y}_i^{(k)} = \gamma \hat{\mathbf{x}_i}^{(k)} + \beta$$

Intuition: BatchNorm (vs. LayerNorm)



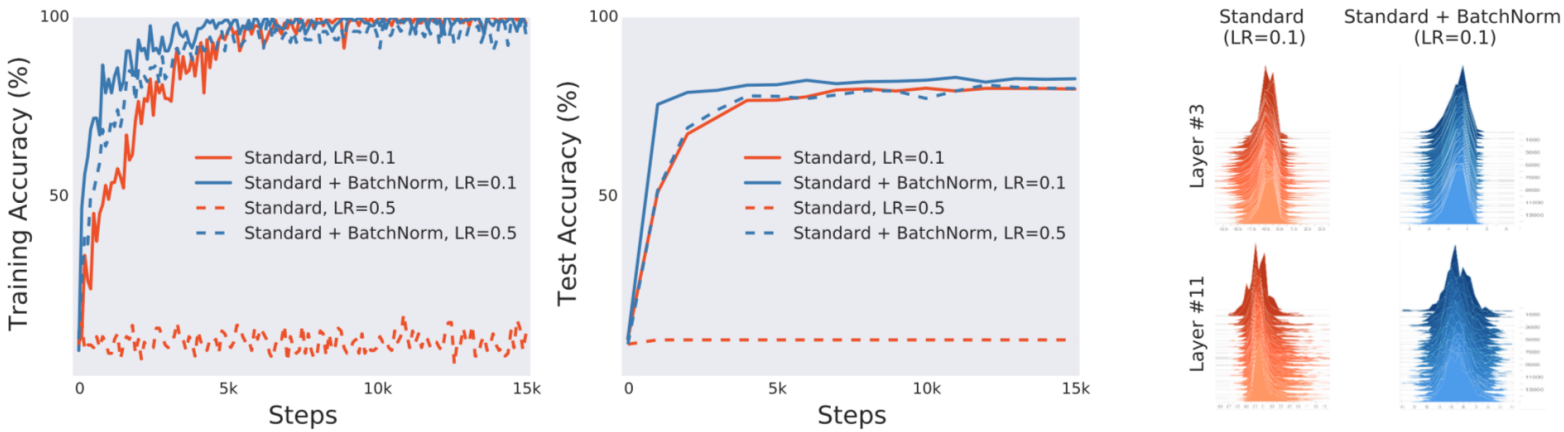
BatchNorm: high order effects

$$f(W_0 - \lambda \mathbf{g}_0) \approx f(W_0) - \lambda \mathbf{g}^\top \mathbf{g} + \frac{1}{2}\lambda^2 \mathbf{g}^\top H \mathbf{g}$$



BatchNorm and Internal Covariate Shift

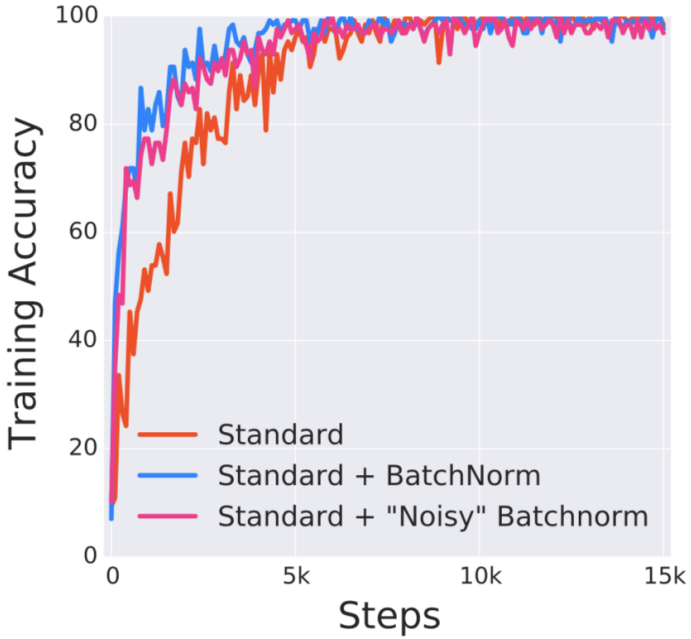
- Experimental findings
  - Drastic improvement of optimization and generalization performance
  - Difference in distributional stability (change in the mean and variance) in networks with and without BatchNorm layers seems marginal

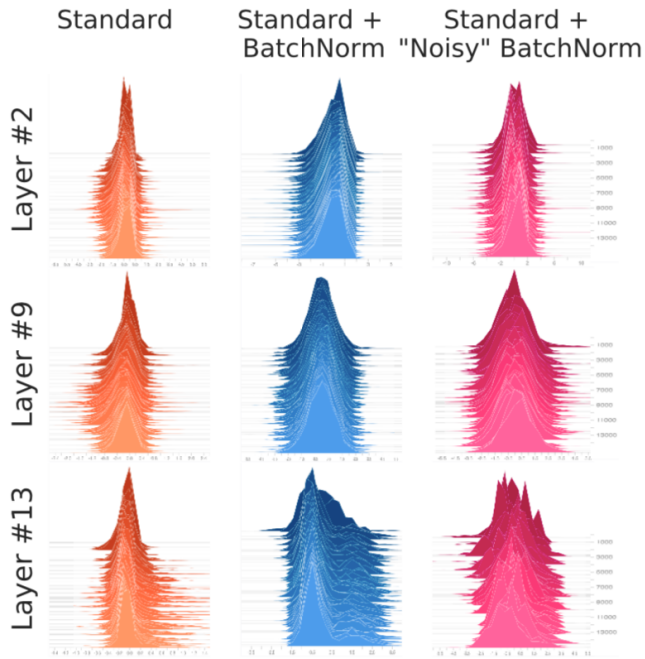


Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm
There is a consistent gain in training speed in models with BatchNorm layers. (c) Even though the gap between the performance of the BatchNorm and non-BatchNorm networks is clear, the difference in the evolution of layer input distributions seems to be much less pronounced. (Here, we sampled activations of a given layer and visualized their distribution over training steps.)

Does BatchNorm really mitigates Internal Covariate Shift?

- Experimental setup
  - Inject i.i.d. noise sampled from a non-zero mean and non-unit variance distribution
  - Perturb each activation for each sample in the batch to exacerbate ICS





Connections between distributional stability and BatchNorm performance: We compare VGG networks trained without BatchNorm (Standard), with BatchNorm (Standard + BatchNorm) and with explicit "covariate shift" added to BatchNorm layers (Standard + "Noisy" BatchNorm).

Why Does BatchNorm Work?

The Smoothing effect of BatchNorm

- BathNorm reparametrizes the underlying optimization problem
  - It make its landscape significantly more smooth
  - Lipschitzness of the loss function improves
  - Gradients of the loss more Lipschitz too

  $\rightarrow$ The loss changes at a smaller rate

  $\rightarrow$ The magnitudes of the gradients are smaller

- f is L-Lipschitz if

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2|, \ \forall x_1, x_2$$

$\rightarrow$ A larger step in a gradient direction is less risky

Model Compression (*)

Motivations

Why Deep Net Compression?

- Modern DeepNets are highly over-parametrized
  - Not very clear how this is not problematic!
  - Generalization improves

- Practical consequences

  `https://github.com/albanie/convnet-burden`

  - Model storage not negligible: $O(\mathrm{GB})$
  - Training computationally demanding
  - Testing too
    - VGG-16 model: 138.34 M parameters, 500MB storage space, 30.94 B FLOPs to classify a single image

- Resource constrained devices pose several challenges

Literature

- Pruning
  - E.g., Optimal Brain Damage, Optimal Brain Surgeon
  - Methodology
    - Focus on benefits for testing
    - Iterative procedure involving several training steps
    - Retain only some DeepNet elements (e.g.) weights for testing

    $\rightarrow$ Reduce parameter-counts by 90+% without harming accuracy

- Information-theoretic Compression
  - Quantization
  - Low-precision representation

- Can we make training more efficient as well?

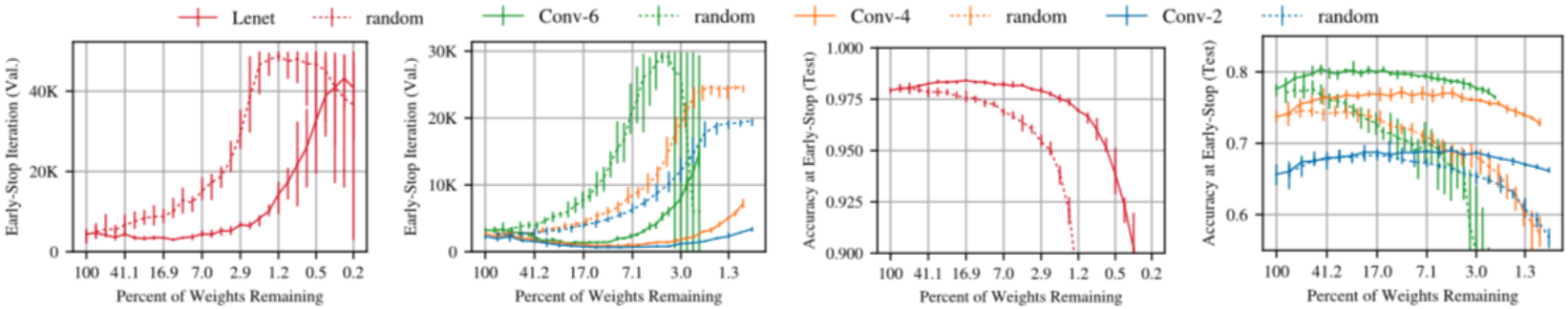
The Lottery Ticket Hypothesis

Based on:

Frankle, J., Carbin, M., "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks", ICLR, 2019

Randomization Alone is not sufficient

- Take a fully-connected Net trained on CIFAR10
  - Randomly sample subnets (similar to pruning)
  - Learning speed: iteration at which an early-stopping intervenes
  - Early-stopping: minimum validation loss during training



The iteration at which early-stopping would occur (left) and the test accuracy at that iteration (right) of the Lenet architecture for MNIST and the Conv-2, Conv-4, and Conv-6 architectures for CIFAR10 when trained starting at various sizes. Dashed lines are randomly sampled sparse networks (average of ten trials). Solid lines are winning tickets (average of five trials).

Lottery Tickets: Formal Definition

- Fully-connected Net
  - Let $f(\mathbf{x}, W)$
  - Let $W = W_0 \sim \mathcal{D}_W$
  - f reaches a minimum validation loss l at iteration j with accurcay a
- Masked Net
  - Let $f(\mathbf{x}, \mathbf{m} \odot W_0)$
  - **Fix** $\mathbf{m} \in \{0,1\}^{|W|}$, and initialization $\mathbf{m} \odot W_0$
  - f reaches a minimum validation loss l' at iteration j' with accurcay a'
- Then: $\exists \mathbf{m}\, \text{s.t.}\, j' \leq j,\ a' \geq a,\ ||\mathbf{m}||_0 << |W$

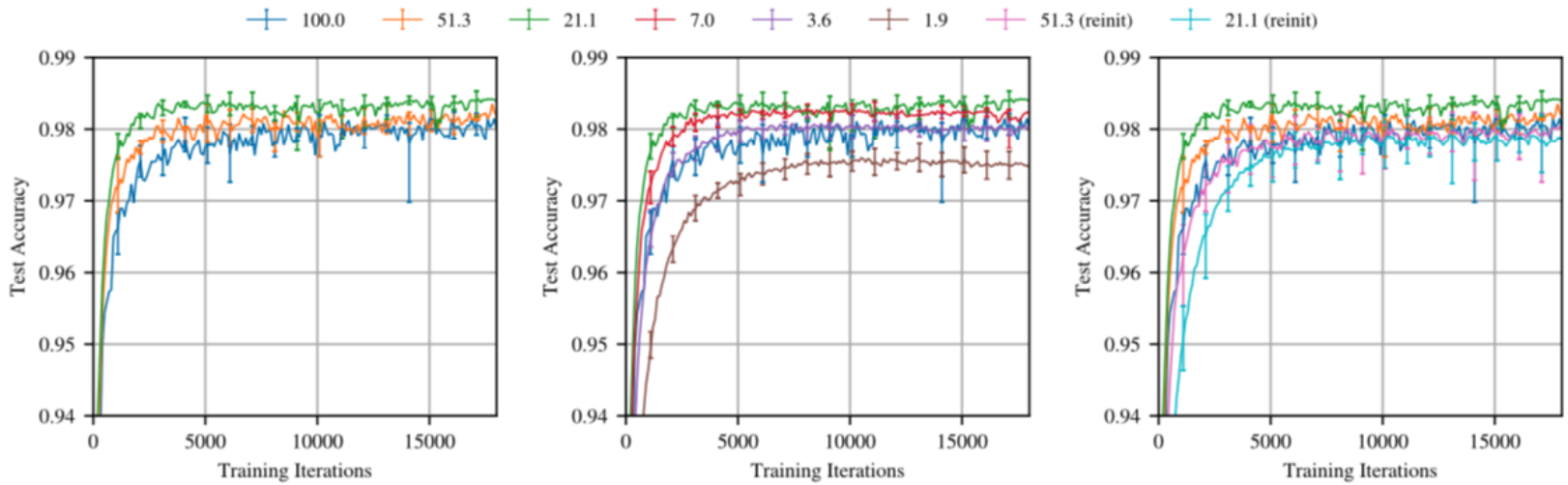
Note: If parameters are randomly reinitialized $f(\mathbf{x}, \mathbf{m} \odot W'_0),\ W'_0 \sim \mathcal{D}_W$, performance is subpar w.r.t original network

Identifying Winning Tickets

- In words
  - Winning ticket: train a network and prune its smallest-magnitude weights
  - Unpruned connection values reset to initialization in the original net before training
- Algorithmically (one-shot version)
  - Randomly initialize a neural net: $f(\mathbf{x}, W_0)$
  - Train the network for j iterations, arriving at parameters W_j
  - Prune $p\%$ of the parameters in W_j, creating a mask $\mathbf{m}$
  - Reset the remaining parameters to their values in W_0
  - Winning ticket: $f(\mathbf{x}, \mathbf{m} \odot W_0)$
- Iterative pruning
  - Trains, prunes, and resets the network over n rounds; each round prunes $p^{\frac{1}{n}}\%$ of the weights that survive the previous round

  $\rightarrow$ smaller mask size than one-shot pruning

Results: Fully-Connected Nets



Test accuracy on Lenet (iterative pruning) as training proceeds. Each curve is the average of five trials. Labels are P_m—the fraction of weights remaining in the network after pruning. Error bars are the minimum and maximum of any trial.

- Layer-wise pruning heuristic: remove a percentage of the weights with the lowest magnitudes within each layer
- Connections to outputs are pruned at half of the rate of the rest of the network

Connections to Dropout

- Dropout improves accuracy by randomly disabling a fraction of the units
  - Random sampling a subnetwork
  - This happens on **each training iteration**

- Additional interesting directions
  - Gomez, A., et. al., "Learning Sparse Networks Using Targeted Dropout", https://arxiv.org/abs/1905.13678