Intro to Deep Learning

Lecture 7: Optimization III

GD for Convex Functions

- Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex function that is also L-smooth
- Let $w^\star := \arg\min_w f(w)$ — the minimizer we try to find
- Consider the following gradient descent update

$$w_{t+1} = w_t - \eta \nabla f(w_t)$$

Step I: The Descend

- If $\eta \leq 1/L$, we then have

$$f(w_{t+1}) \leq f(w_t) - \frac{\eta}{2}\|\nabla f(w_t)\|_2^2$$

Function value decreases strictly unless $\nabla f(w_t) = 0$

Step II: Convergence Rate

We have

- $f(w_{t+1}) - f(w^\star) \leq \frac{1}{2\eta}\left(\|w_t - w^\star\|_2^2 - \|w_{t+1} - w^\star\|_2^2\right)$

- $f(w_T) - f(w^\star) \leq \frac{\|w_0 - w^\star\|_2^2}{2\eta T}$

  - To achieve ε error, we need $O\left(\frac{1}{\eta\varepsilon}\right) = O\left(\frac{L}{\varepsilon}\right)$ number of iterations

GD for Linear regression

Optimization becomes

$$f(w) = \frac{1}{2} \left\| y - Xw \right\|_2^2$$

- f is convex (actually quadratic)
- It's gradient equals to

$$\nabla f(w) = X^\top Xw - X^\top y$$

GD for Linear regression

- It's gradient equals to

$$\nabla f(w) = X^\top X w - X^\top y$$

- $\nabla f(w)$ is $\lambda_{\max}\left(X^\top X\right)$-Lipschitz

$$|\nabla f(w_1) - \nabla f(w_2)| = |X^\top X(w_1 - w_2)| \le \lambda_{\max}(X^\top X) \cdot \|w_1 - w_2\|_2$$

Select the learning rate as $\eta = 1/\lambda_{\max}(X^\top X)$ and run for $O\left(\lambda_{\max}(X^\top X)/\varepsilon\right)$ iterations ensures convergence up to an ε-optimal minimizer.

Only $O(\log(1/\varepsilon))$ iterations are needed if $\lambda_{\min}(X^\top X) > 0$, i.e., under strong convexity

Stochastic Gradient Descent

- The empirical risk minimization with m examples solves

$$\mathscr{L}(w) = \frac{1}{m} \sum_{i=1}^{m} \ell(y_i, g(x_i; w))$$

- This is a special case of minimizing avg of functions

$$\mathscr{L}(w) = \frac{1}{m} \sum_{i=1}^{m} f_i(w)$$

where $f_i(w) = \ell(y_i, g(x_i; w))$.

- Per-iteration cost is proportional to m.
- Question: Efficient ways to minimize avg of functions?

Stochastic Gradient Descent

Consider minimizing an average of functions

$$\min_x \frac{1}{m} \sum_{i=1}^{m} f_i(x)$$

Stochastic Gradient Descent

Consider minimizing an average of functions

$$\min_x \frac{1}{m} \sum_{i=1}^m f_i(x)$$

As $\nabla \sum_{i=1}^m f_i(x) = \sum_{i=1}^m \nabla f_i(x)$, gradient descent would repeat:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{m} \sum_{i=1}^m \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

Stochastic Gradient Descent

Consider minimizing an average of functions

$$\min_x \frac{1}{m} \sum_{i=1}^m f_i(x)$$

As $\nabla \sum_{i=1}^m f_i(x) = \sum_{i=1}^m \nabla f_i(x)$, gradient descent would repeat:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{m} \sum_{i=1}^m \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

In comparison, stochastic gradient descent or SGD (or incremental gradient descent) repeats:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \nabla f_{i_k}(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

where $i_k \in \{1, \ldots, m\}$ is some chosen index at iteration k

Two rules for choosing index i_k at iteration k:

- Randomized rule: choose $i_k \in \{1, \ldots, m\}$ uniformly at random
- Cyclic rule: choose $i_k = 1, 2, \ldots, m, 1, 2, \ldots, m, \ldots$

Two rules for choosing index i_k at iteration k:

- Randomized rule: choose $i_k \in \{1, \ldots, m\}$ uniformly at random
- Cyclic rule: choose $i_k = 1, 2, \ldots, m, 1, 2, \ldots, m, \ldots$

Randomized rule is more common in practice. For randomized rule, note that

$$\mathbb{E}[\nabla f_{i_k}(x)] = \nabla f(x)$$

so we can view SGD as using an unbiased estimate of the gradient at each step

Two rules for choosing index i_k at iteration k:

- Randomized rule: choose $i_k \in \{1, \ldots, m\}$ uniformly at random
- Cyclic rule: choose $i_k = 1, 2, \ldots, m, 1, 2, \ldots, m, \ldots$

Randomized rule is more common in practice. For randomized rule, note that

$$\mathbb{E}[\nabla f_{i_k}(x)] = \nabla f(x)$$

so we can view SGD as using an unbiased estimate of the gradient at each step

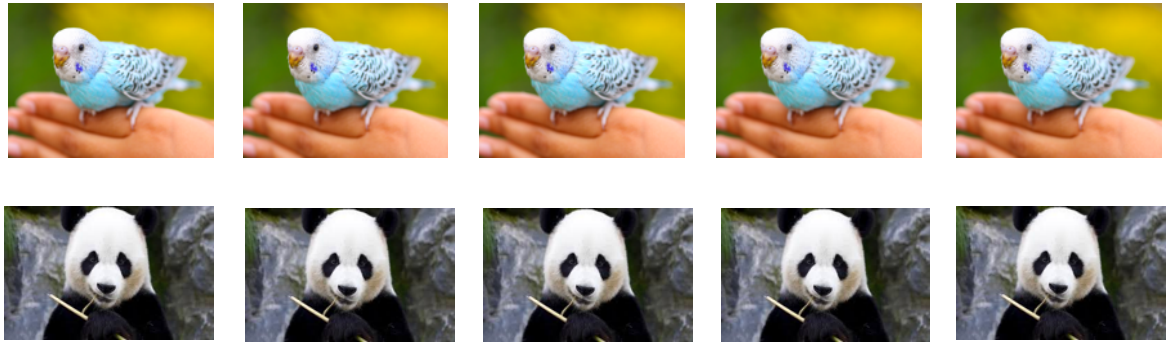
Main appeal of SGD:

- Iteration cost is independent of m (number of functions)
- Can also be a big savings in terms of memory useage

Intuition: Avoid Redundancy

- Example: Classify parrot vs monkey

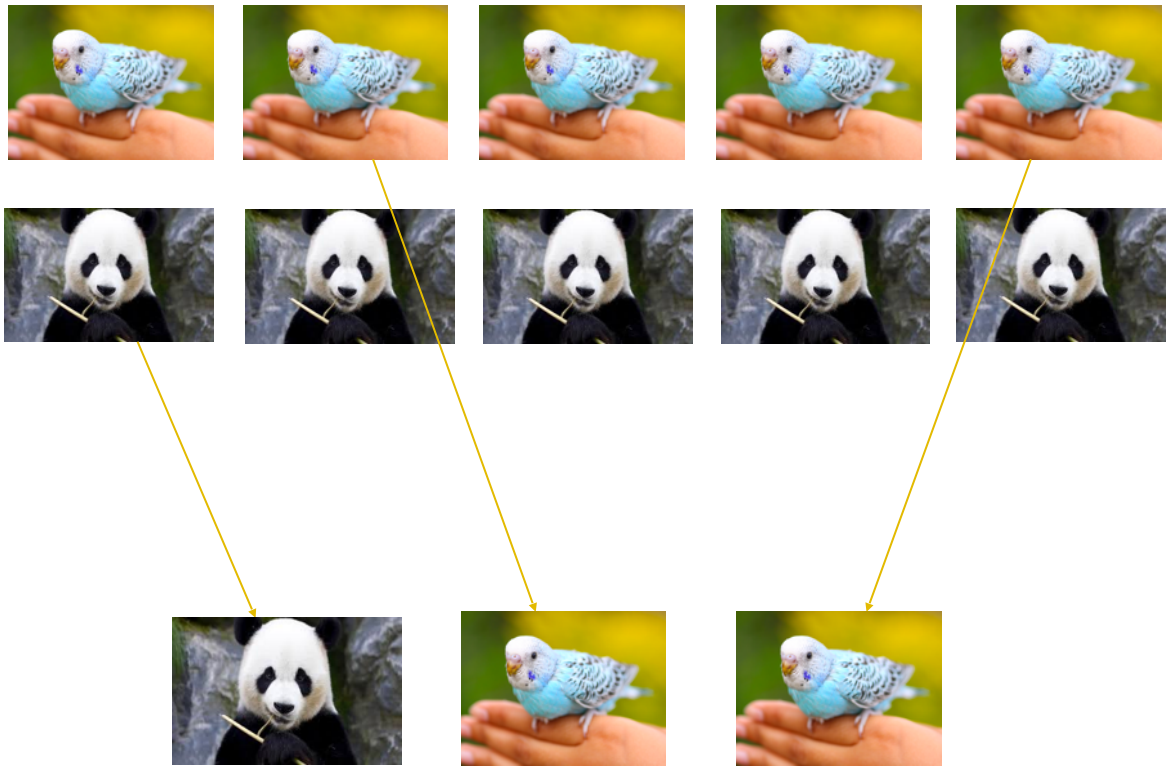
My Dataset



- Gradient descent: Use all 10 pictures
- ☹ These pictures are all duplicates
  - Duplicate gradients are redundant calculations

Making things efficient

My Dataset

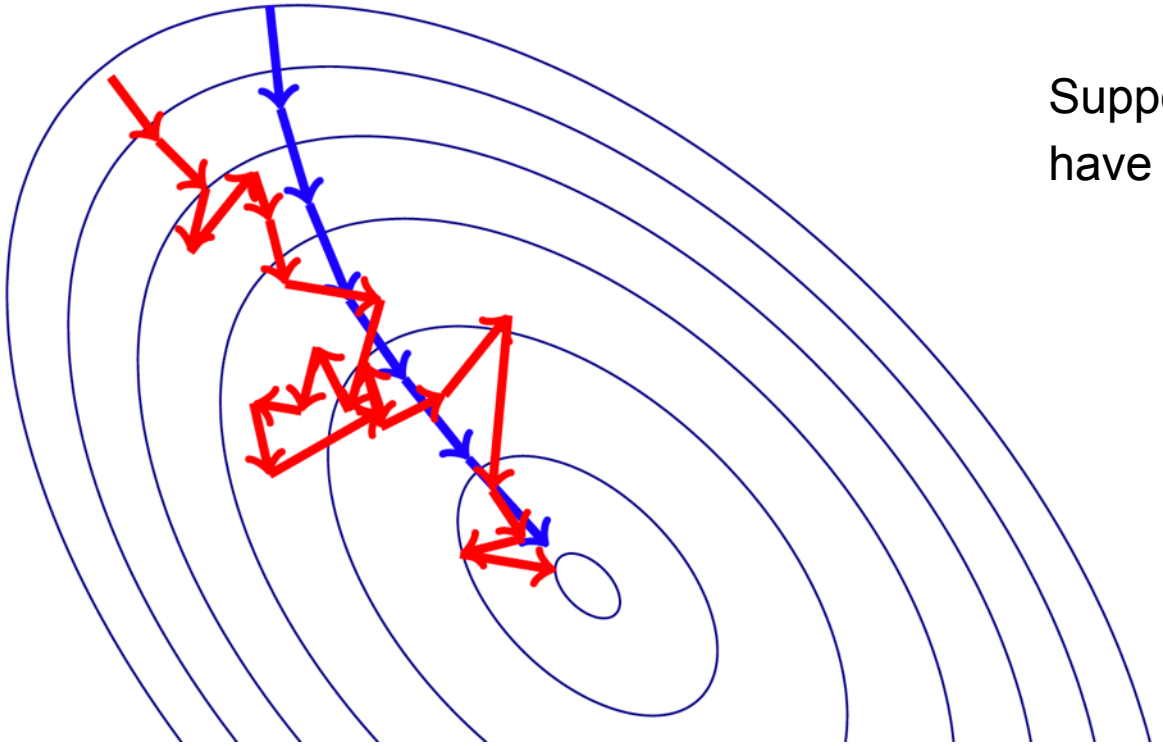


3 random samples

☺ These three samples contain all the info

Remark: In real data, examples are not duplicates but similar

Computation Cost



Suppose $x \in \mathbb{R}^d$ and we have m data points

- GD: $O(md)$
- SGD: $O(d)$

SGD can struggle when it gets close to the optimum

Mini-batch SGD

Also common is mini-batch stochastic gradient descent, where we choose a random subset $I_k \subseteq \{1, \ldots, m\}$, $|I_k| = b \ll m$, repeat:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{b} \sum_{i \in I_k} \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

Mini-batch SGD

Also common is mini-batch stochastic gradient descent, where we choose a random subset $I_k \subseteq \{1, \ldots, m\}$, $|I_k| = b \ll m$, repeat:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{b} \sum_{i \in I_k} \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

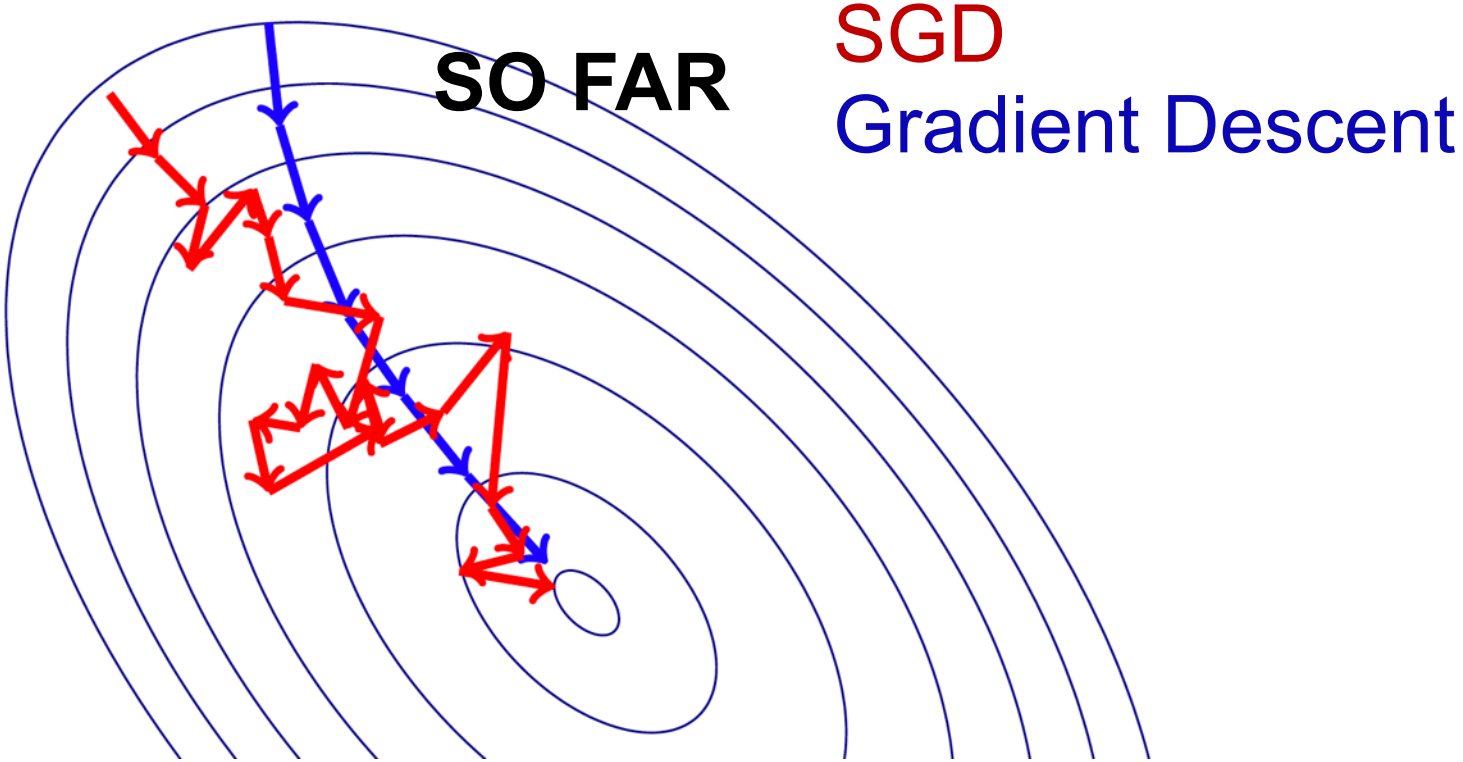
Again, we are approximating full gradient by an unbiased estimate:

$$\mathbb{E}\left[\frac{1}{b} \sum_{i \in I_k} \nabla f_i(x)\right] = \nabla f(x)$$

Computation Cost

- Suppose $x \in \mathbb{R}^d$, and we have m data points in total. Suppose the mini-match size is $b \in [m]$.
  - Gradient descent: $O(md)$
  - Stochastic gradient descent: $O(d)$
  - Mini-batch SGD: $O(bd)$

Going beyond SGD



How can we do better than minibatch SGD?

Optimization Algorithms

Many smart people are working on speeding up stuff!

Key ideas:

- Making the optimizer adapt to the data
- Take advantage of previous calculations

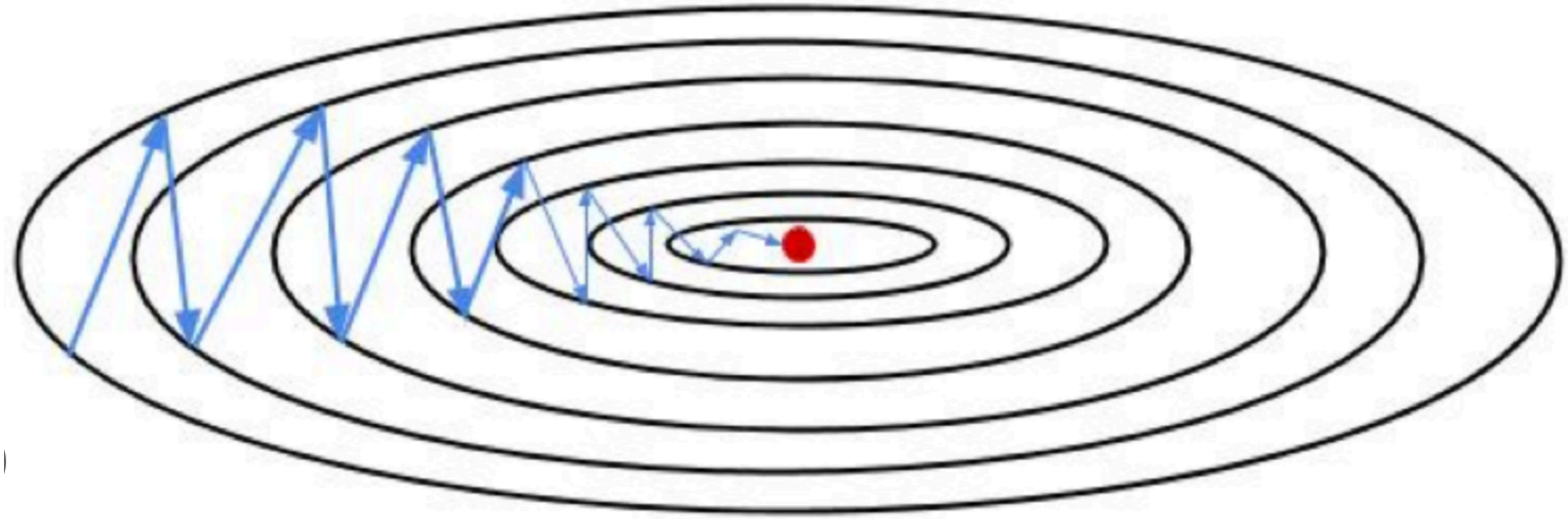
Optimizers:

- Momentum
- Adaptive Methods (AdaGrad, RMSProp, Adam)

Momentum

- The Momentum method is a method to accelerate learning using SGD

- In particular SGD suffers in the following scenarios:
  - Error surface has asymmetric curvature
  - The gradients are very noisy



(Stochastic) gradient descent can exhibit Zigzag behavior

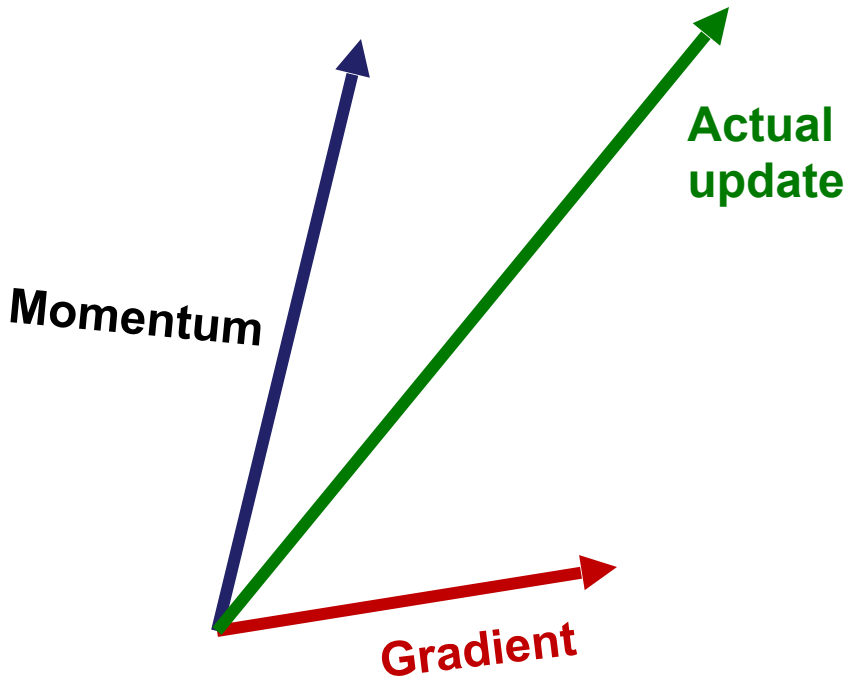
Momentum

- How do we solve this problem?
- Introduce a new variable m_t

- Think of m_t as the direction and speed by which the parameters (weights) move as the learning dynamics progresses

- Choose parameter $\beta \in (0,1)$. Denote $g_t = \nabla f_t$
  - $m_t = \beta m_{t-1} + g_{t-1}$
  - $\theta_t = \theta_{t-1} - \eta m_t$

Effective learning rate $\frac{\eta}{1-\beta}$


Actual
update
Momentum
Gradient

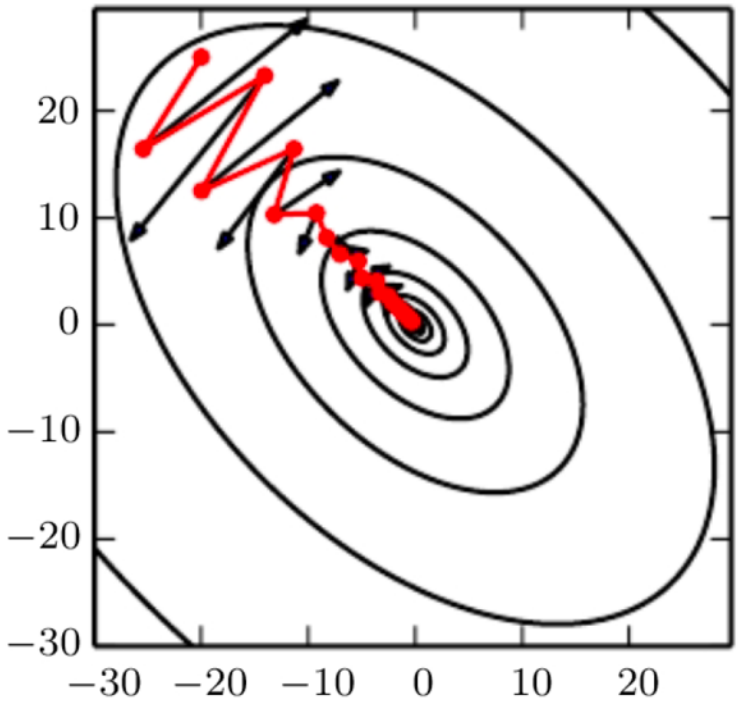
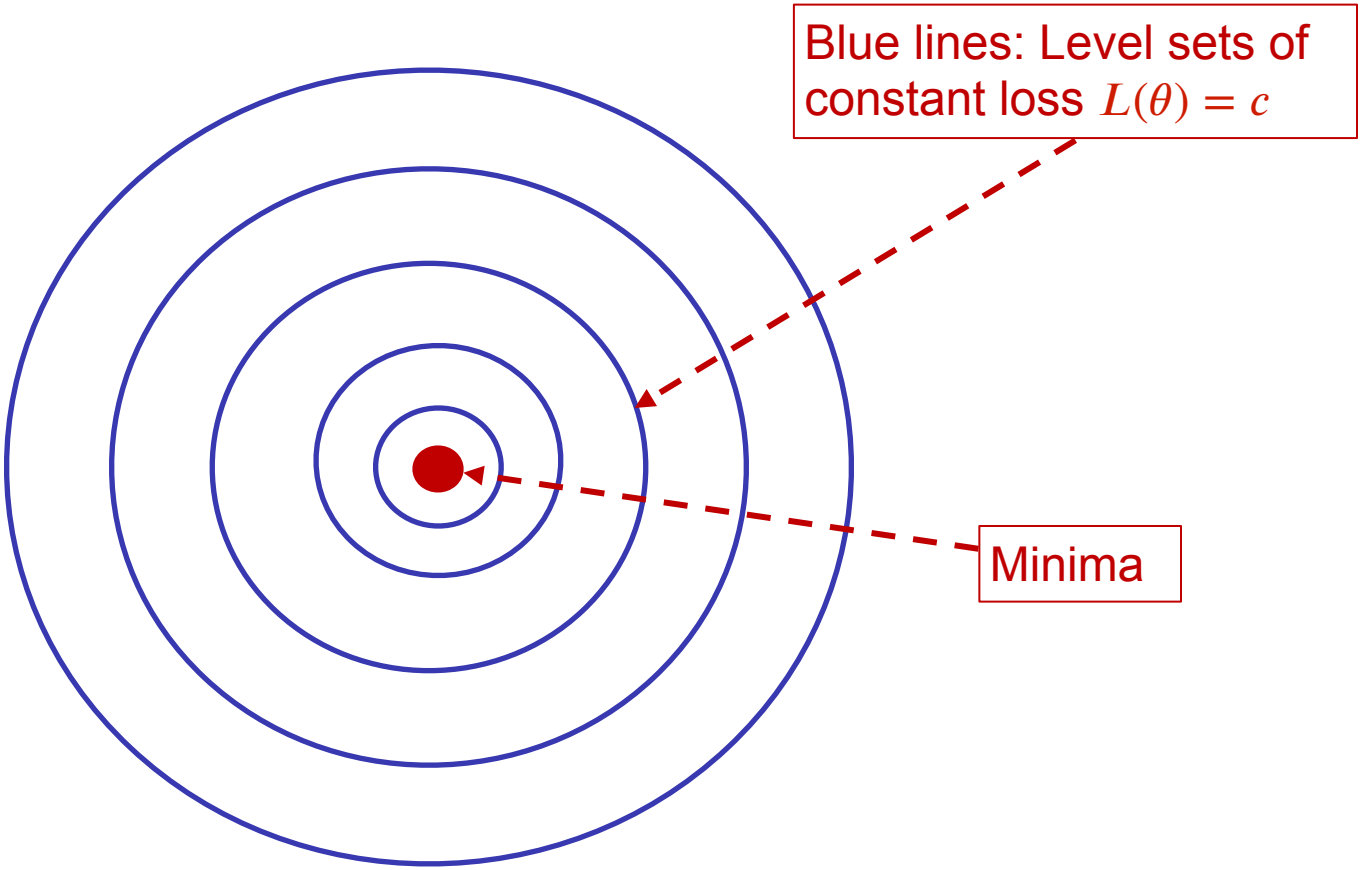


Illustration of how momentum traverses such an error surface better compared to Gradient Descent

Motivation

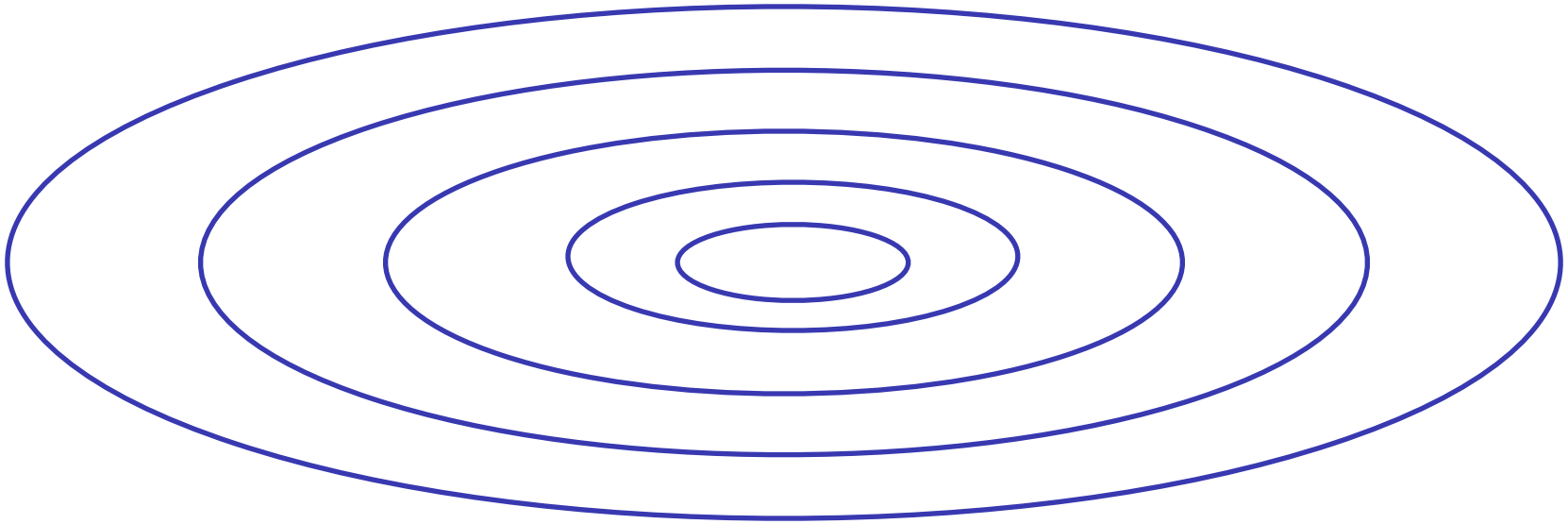
- Till now we assign the same learning rate to all features
- If the features vary in importance and frequency, why is this a good idea?
- It's probably not!

Loss landscape



Nice (all features are equally important)

Loss landscape



Not as nice!
Wish to have small learning rate on y axis
but large learning rate on x axis

AdaGrad (Adaptive Gradient)

- **Idea:** Downscale a model parameter by square-root of sum of squares of all its historical values

- Parameters that have large partial derivative of the loss: learning rates for them are rapidly declined

AdaGrad (Adaptive Gradient)

- For any $k \in [d]$, set $s_{t,k} = \sum_{i=1}^{t} g_{i,k}^2$
- Update $\theta_{t+1,k} = \theta_{t,k} - \eta \frac{1}{\sqrt{s_{t,k} + \varepsilon}} g_{t,k}$
- In matrix format:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{\sqrt{s_t + \varepsilon}} \odot g_t$$

Break

RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient
- Currently has about 7000+ citations on scholar, but was proposed **in a slide** in Geoffrey Hinton's Coursera course

RMSProp

- Set $s_{t,k} = \beta s_{t-1,k} + (1-\beta) g_{t-1,k}^2$

Exponentially weighted moving average

- Update $\theta_{t+1,k} = \theta_{t,k} - \eta \frac{1}{\sqrt{s_{t,k} + \varepsilon}} g_{t,k}$
- In matrix format:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{\sqrt{s_t + \varepsilon}} \odot g_t$$

Adam: ADAptive Moments

- We could have used RMSProp with momentum
- Use of Momentum with rescaling is not well motivated
- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

Adam: A method for stochastic optimization

DP Kingma, J Ba - arXiv preprint arXiv:1412.6980, 2014 - arxiv.org Paperpile

… **Adam** works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss AdaMax, a variant of **Adam** … Overall, we show that **Adam** is a versatile …

☆ Save 99 Cite Cited by 157542 Related articles All 27 versions Import into BibTeX »

Adam

- Set two parameters $\beta_1, \beta_2 \in (0,1)$
- Set
  - $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ Momentum
  - $s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$ Adaptive gradient
- Update $\theta_{t+1,k} = \theta_{t,k} - \eta \frac{1}{\sqrt{s_{t,k}} + \varepsilon} m_{t,k}$ for any $k \in [d]$
- In matrix format:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{\sqrt{s_t} + \varepsilon} \odot m_t$$

Finding the best model

- Statistical learning: Data is generated from an unknown distribution

$$(x, y) \sim \mathscr{D}$$

- The model f has the performance

$$\mathscr{L}(f) = \mathbb{E}_{\mathscr{D}}\left[\ell(f(x), y)\right]$$

Observation: $\mathbb{E}_{\mathscr{D}}\left[\mathbb{I}\left(y \neq \operatorname{sign}(f(x))\right)\right] = \mathbb{P}\left(y \neq \operatorname{sign}(f(x))\right)$

- We can find best model from a hypothesis set $\mathscr{F}$ via

$$f^\star = \arg\min_{f \in \mathscr{F}} \mathscr{L}(f)$$

Training Loss

Our aim is typically finding an accurate model i.e.

$$f^\star = \arg\min_{f \in \mathcal{F}} \mathscr{L}(f) \quad \text{where} \quad \mathscr{L}(f) = \mathbb{E}_{\mathscr{D}}\left[\ell(f(x), y)\right]$$

- **Challenge 1:** We don't have $\mathscr{D}$. Instead, we only have training data sampled from $\mathscr{D}$

$$\left(x_i, y_i\right)_{i=1}^n \sim \mathscr{D}$$

- **Solution:** Minimize the training loss to approximate $\mathscr{L}(f)$

$$\hat{f} = \arg\min_{f \in \mathcal{F}} \widehat{\mathscr{L}}(f) \quad \text{where} \quad \widehat{\mathscr{L}}(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(x), y)$$

Hopefully $\hat{f}$ will be a good proxy for $f^\star$! Rigorous proof via concentration inequalities!

Optimization on Training Set

- Gradient descent iterations

$$f_{i+1} = f_i - \eta \nabla \mathcal{L}(f_i)$$

New model | Current model | Learning rate | Gradient

- Gradient descent for convex functions:
  - Provable guarantees for convergence
- Other variants for neural networks:
  - AdaGrad, RMSProp, Adam…

Statistical Learning Guarantees

- Suppose $(x_i, y_i)_{i=1}^n \sim \mathscr{D}$.
- Fix $\mathscr{F}$. Denote
- $f^\star = \arg\min_{f \in \mathscr{F}} \mathscr{L}(f), \hat{f} = \arg\min_{f \in \mathscr{F}} \widehat{\mathscr{L}}(f)$
- We have
- $\text{err}(\hat{f}) - \text{err}(f^\star) \propto \sqrt{\frac{\text{complexity}(\mathscr{F})}{n}}$

Learning-Approx. Error Trade-off

- Let f_{Bayes} denote the optimal classifier wrt the unknown distribution $\mathscr{D}$.
- We have

$$\text{err}(\hat{f}) - \text{err}(f_{\text{Bayes}}) = \text{err}(\hat{f}) - \text{err}(f^\star) + \text{err}(f^\star) - \text{err}(f_{\text{Bayes}})$$

- $$\propto \sqrt{\frac{\text{complexity}(\mathscr{F})}{n}} + \left(\text{err}(f^\star) - \text{err}(f_{\text{Bayes}})\right)$$

Learning error

Approximation error

Inversely proportional to complexity($\mathscr{F}$)

Learning-Approx. Error Trade-off

- We have
- $\text{err}(\hat{f}) - \text{err}(f_{\text{Bayes}}) \propto \sqrt{\frac{\text{complexity}(\mathcal{F})}{n}} + \left(\text{err}(f^\star) - \text{err}(f_{\text{Bayes}})\right)$

Learning error Approximation error

  - For complex $\mathcal{F}$, the learning error dominates
  - For simple $\mathcal{F}$, the approximation error dominates

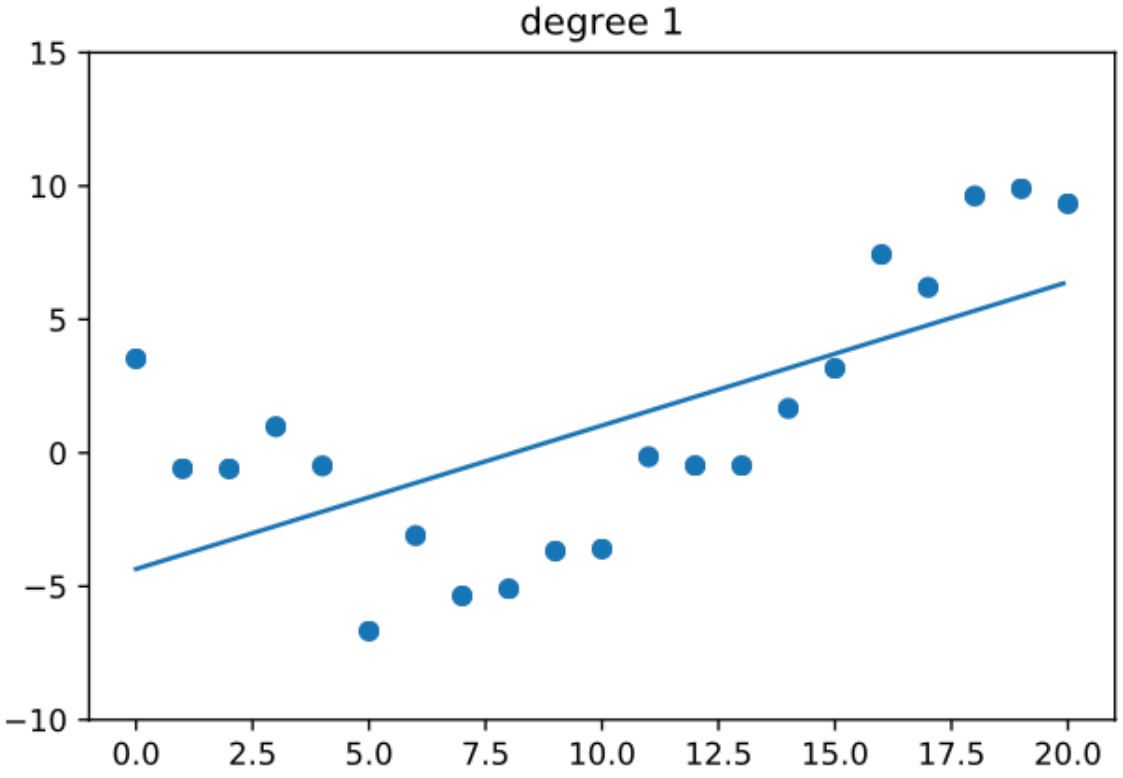
An Illustration

- Randomly generate $(x_i, y_i)_{i=1}^{21} \sim \mathscr{D}$; where $x_i, y_i \in \mathbb{R}$, and $y_i = \tilde{f}(x_i) + \varepsilon_i$ for a quadratic function $\tilde{f}$.
- For any $x \in \mathbb{R}$, define

$$\phi_d(x) = [1, x, \ldots, x^d]$$

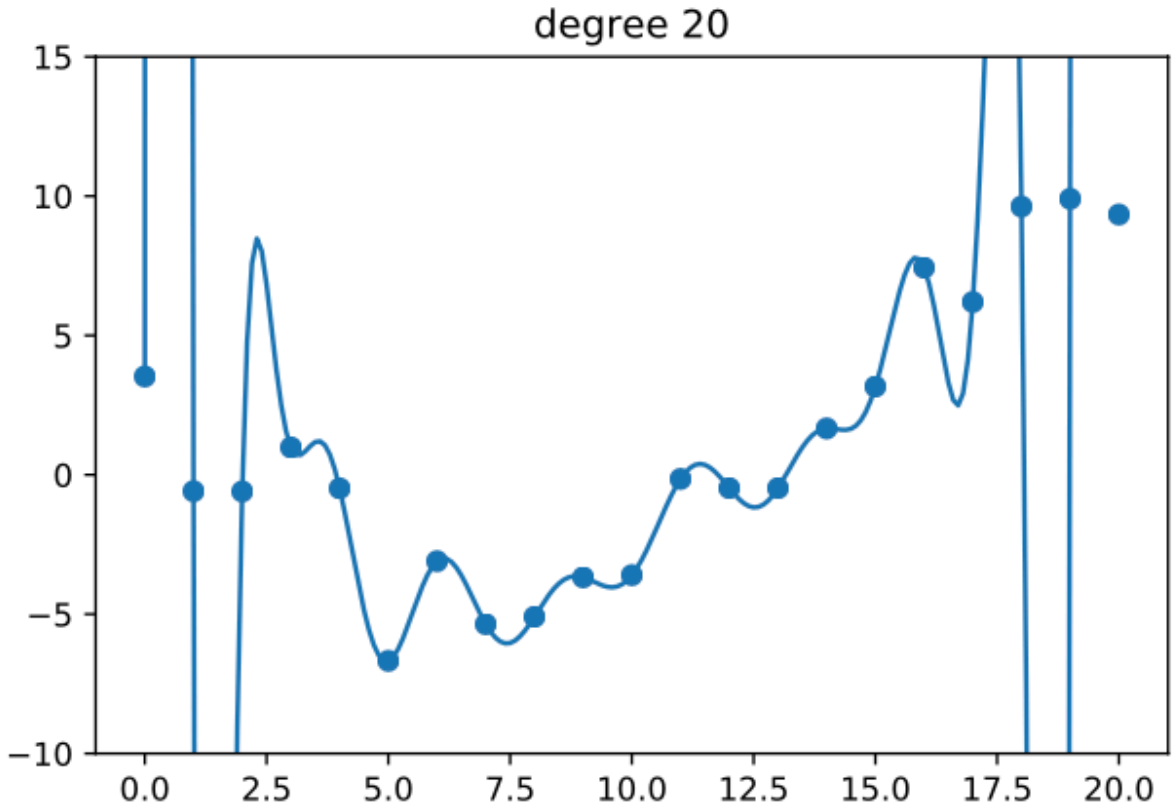
- Fit $(x_i, y_i)_{i=1}^{21}$ with $f_d(x) = \phi_d(x)^\top w$.
- Vary d

$d = 1$



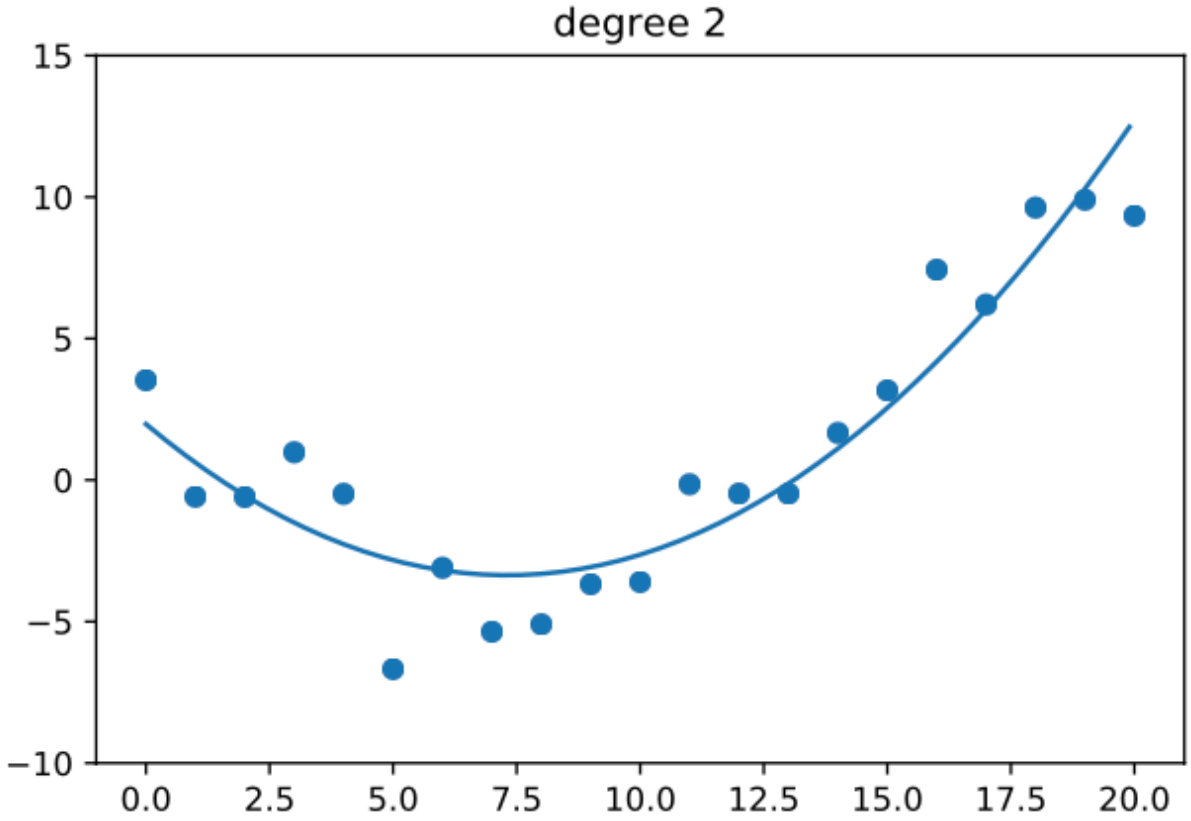
Approximation error dominates

$d = 20$



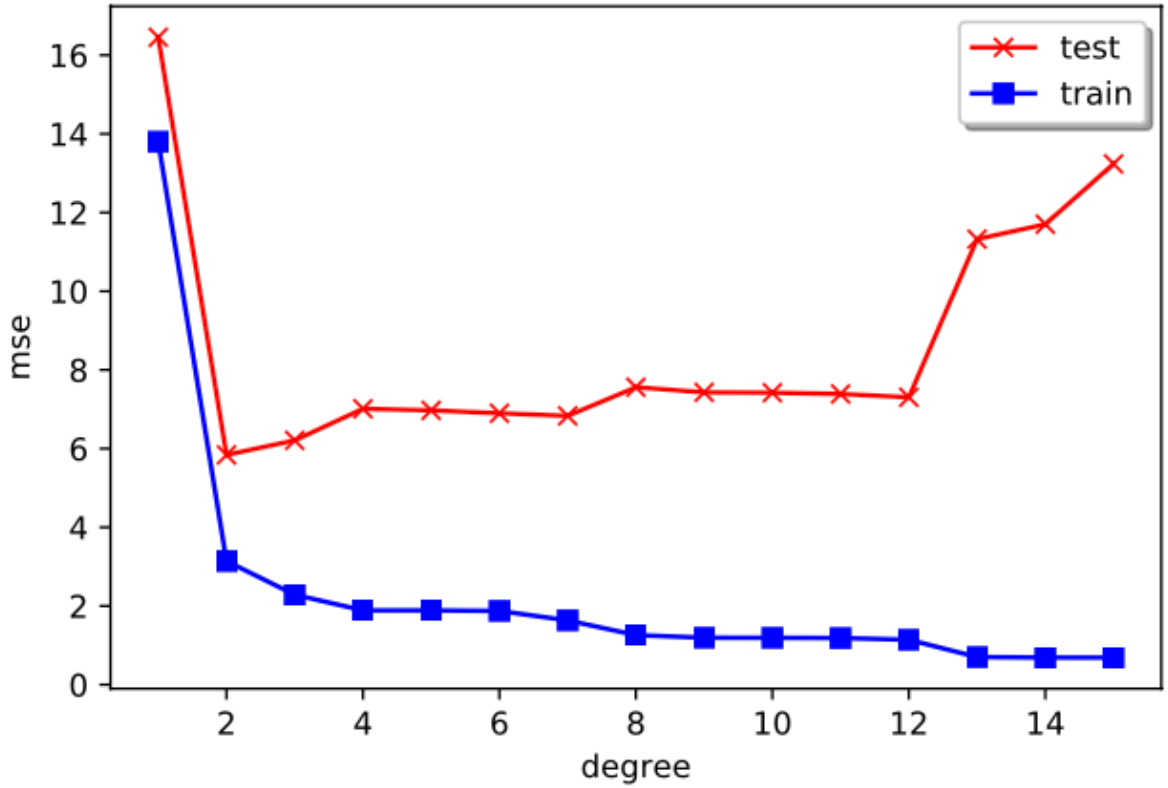
Learning error dominates

$d = 2$



Balance learning error and approx. error

Vary d



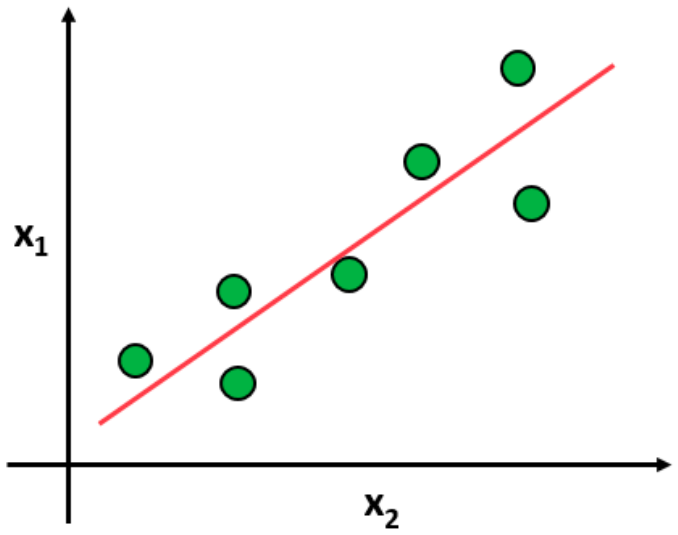
Learning-Approx. Error Trade-off

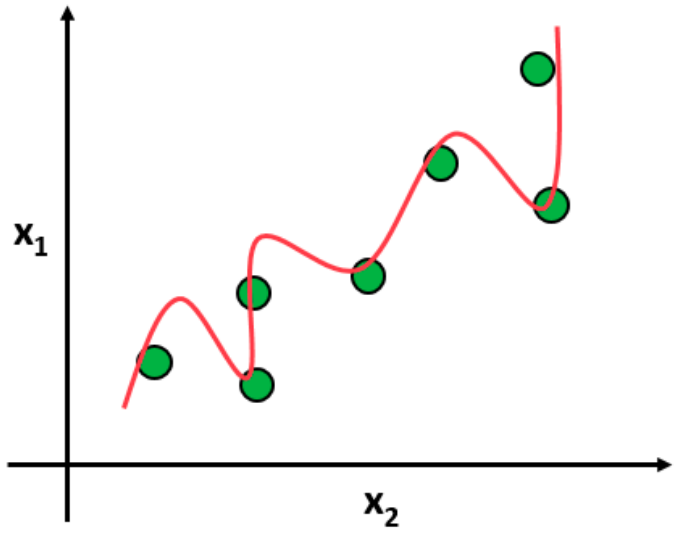
- We have
- $\text{err}(\hat{f}) - \text{err}(f_{\text{Bayes}}) \propto \sqrt{\frac{\text{complexity}(\mathscr{F})}{n}} + \left(\text{err}(f^{\star}) - \text{err}(f_{\text{Bayes}})\right)$

Learning error Approximation error

- Difficulty: Hard to choose the "right" $\mathscr{F}$ in hindsight
- Common practice: Choose a complex $\mathscr{F}$ (e.g., neural nets), but add regularization during learning

Regularization (and other tricks)





Regularization

- Idea: control learning capacity of ML models by searching over a smaller set of models
  - Hope: Finding models that generalize better

- Regularization is "bag of tricks" that help with generalization by adding information to the optimization

Regularization

- Regularized empirical risk minimization with $\lambda > 0$:

$$\mathscr{L}_\lambda(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(x), y; w) + \lambda R(w)$$

Standard loss function $\mathscr{L}(w)$ Regularization

- ℓ_2 regularization: $R(w) = \|w\|_2^2$

  Implication: Model weights are small

- ℓ_1 regularization: $R(w) = \|w\|_1$

  Implication: Model weights are sparse

Ridge Regression

- Regularized empirical risk minimization with $\lambda > 0$:

$$\mathscr{L}_\lambda(w) = \mathscr{L}(w) + \lambda R(w)$$

- Gradient update:

$$\begin{aligned} w_{t+1} &= w_t - \eta \nabla \mathscr{L}_\lambda(w_t) \\ &= w_t - \eta \Big(\nabla \mathscr{L}(w_t) + \lambda \nabla R(w_t) \Big) \end{aligned}$$

Ridge Regression

- Given data matrix X and label y, we have

$$\mathscr{L}_\lambda(w) = \|Xw - y\|_2^2 + \lambda \|w\|_2^2$$

- Differentiating:

$$\frac{1}{2}\nabla \mathscr{L}_\lambda(w) = X^\top X w - X^\top y + \lambda w$$

$$= \left(X^\top X + \lambda I\right) w - X^\top y$$

- Setting the gradient to 0:

$$\widehat{w} = \left(X^\top X + \lambda I\right)^{-1} X^\top y$$

Ridge Regression

- Given data matrix X and label y, we have

$$\mathscr{L}_\lambda(w) = \|Xw - y\|_2^2 + \lambda\|w\|_2^2$$

- Global minimizer:

$$\widehat{w} = \left(X^\top X + \lambda I\right)^{-1} X^\top y$$

- Observations:
  - As λ grows, $\| \widehat{w} \|_2$ decreases
  - Unique solution exists since $X^\top X + \lambda I$ is positive definite

Ridge regression

- Also known as weight decay (why?)
- General form: $\mathscr{L}_\lambda(w) = \mathscr{L}(w) + \lambda \|w\|_2^2$
- Gradient update:

$$w_{t+1} = w_t - \eta\left(\nabla \mathscr{L}(w_t) + 2\lambda w_t\right)$$

$$= \left(1 - 2\eta\lambda\right) w_t - \eta \nabla \mathscr{L}(w_t)$$

Weights are decaying

Weight Decay Benefits

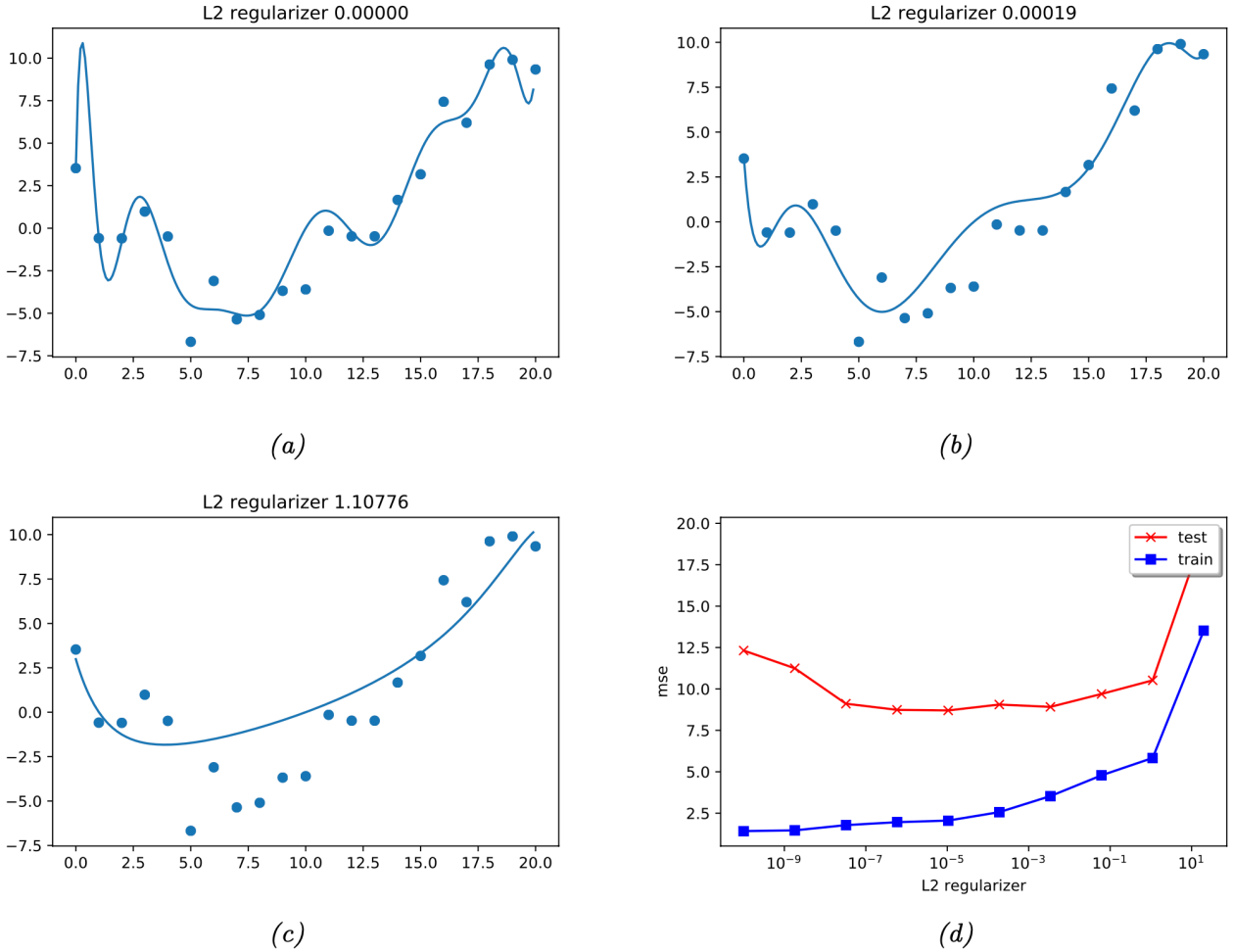
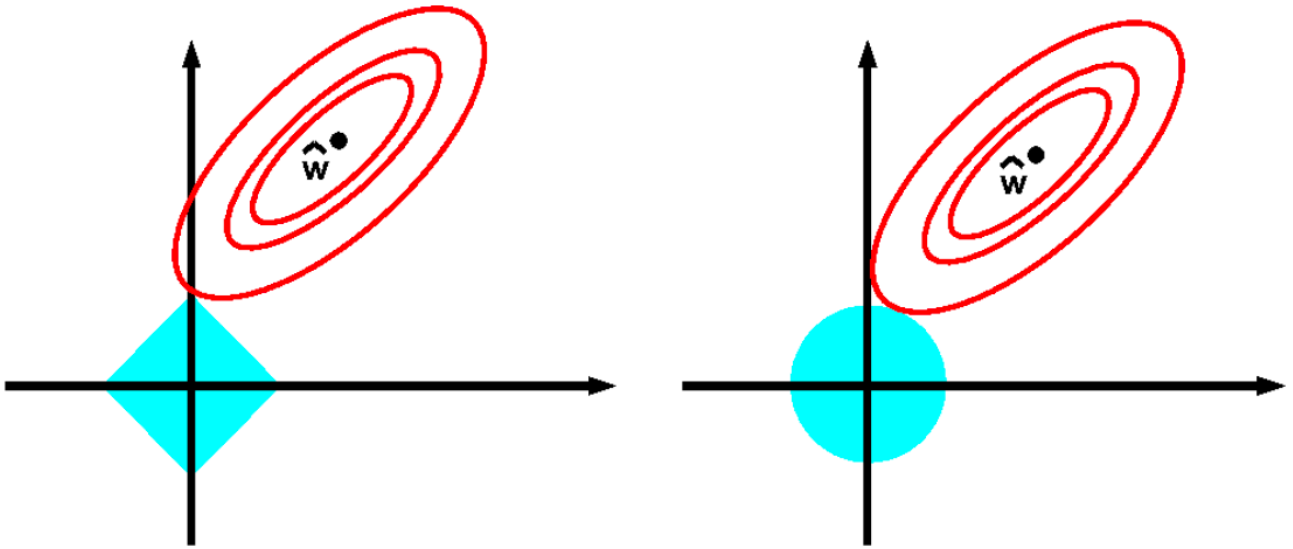


Figure 4.5: (a-c) Ridge regression applied to a degree 14 polynomial fit to 21 datapoints. (d) MSE vs strength of regularizer. The degree of regularization increases from left to right, so model complexity decreases from left to right. Generated by linreg_poly_ridge.ipynb.

ℓ_1 Regularization Produces Sparse Solutions

- Write the optimization problem in its constrained form

$$\min_{w} \mathcal{L}(w) \quad \text{s.t.} \quad R(w) \leq B$$

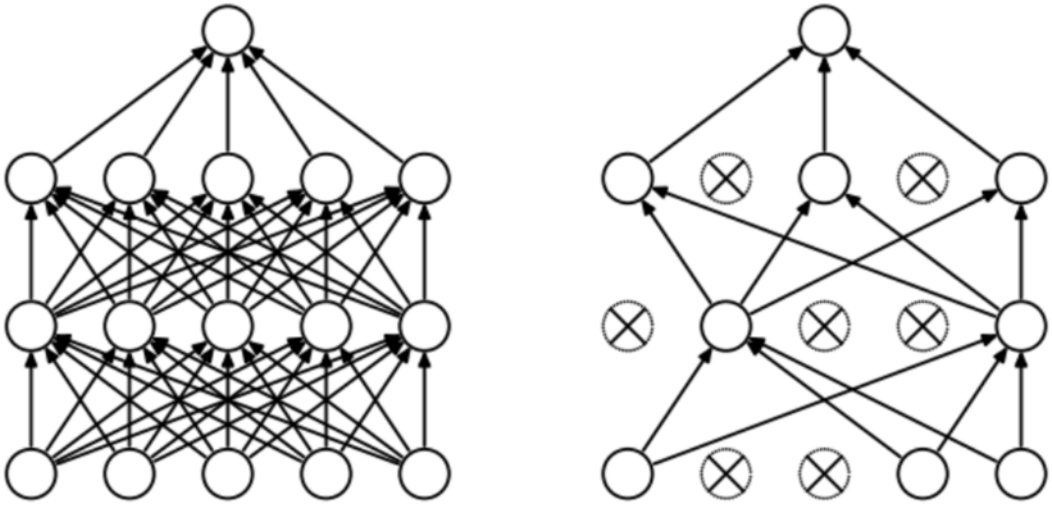


Dropout

- Hope: Average the prediction of multiple independently trained neural networks
- Difficulty: Computationally expensive
- Idea: Train small/thin neural networks and then average the results

Dropout

- At every SGD iteration: disable some of the neurons at random(with probability p) and train with others only.

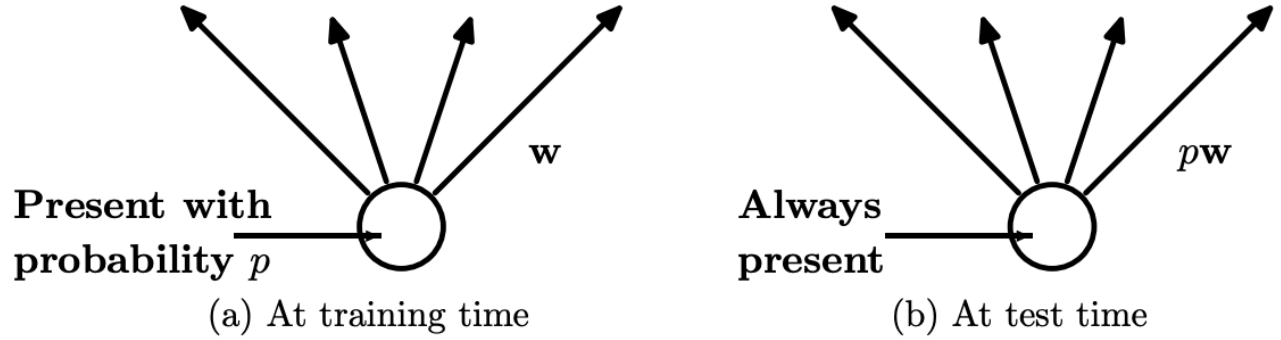


(a) Standard Neural Net (b) After applying dropout.

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

Dropout

- At inference time, multiple the learned weight by the dropout probability $p \in [0,1]$



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

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Dropout

- **Goal:** Prevent overfitting (how?)
  - Average the performance of multiple "thin" neural networks
  - Learn good and robust features. Enforce even a random subset of neurons predicts well!

- Dropout rate p: Probability to keep a node
  - $p = 1$: Keep all nodes (no regularization)
  - $p = 0$: Drop all nodes (layer has no output)
  - Typical values: $0.5 \leq p \leq 0.8$

Dropout Performance

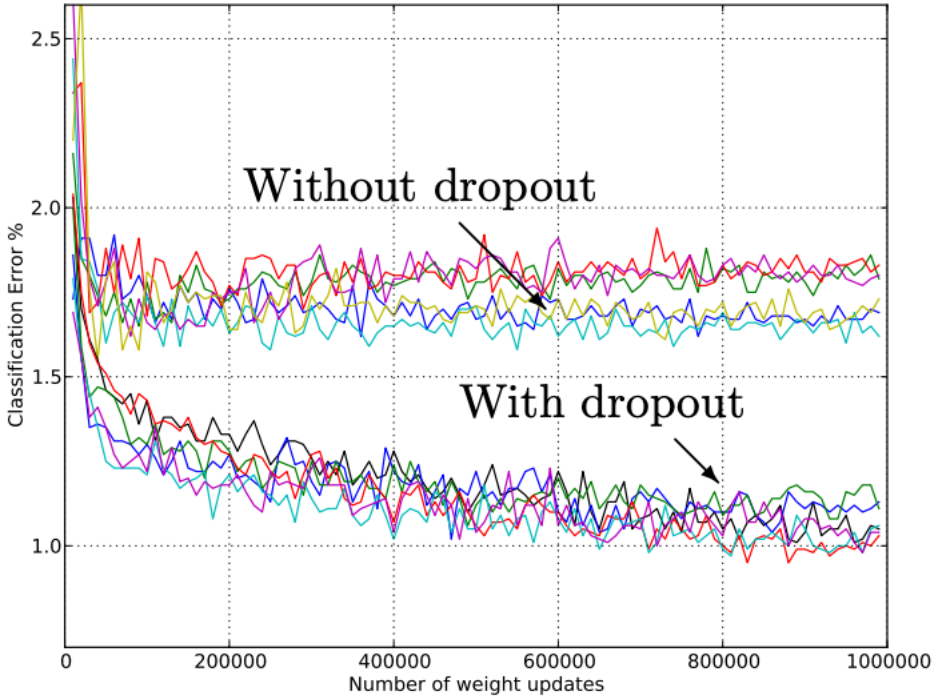


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Performance on MNIST

Method	Test Classification error %
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	**1.05**

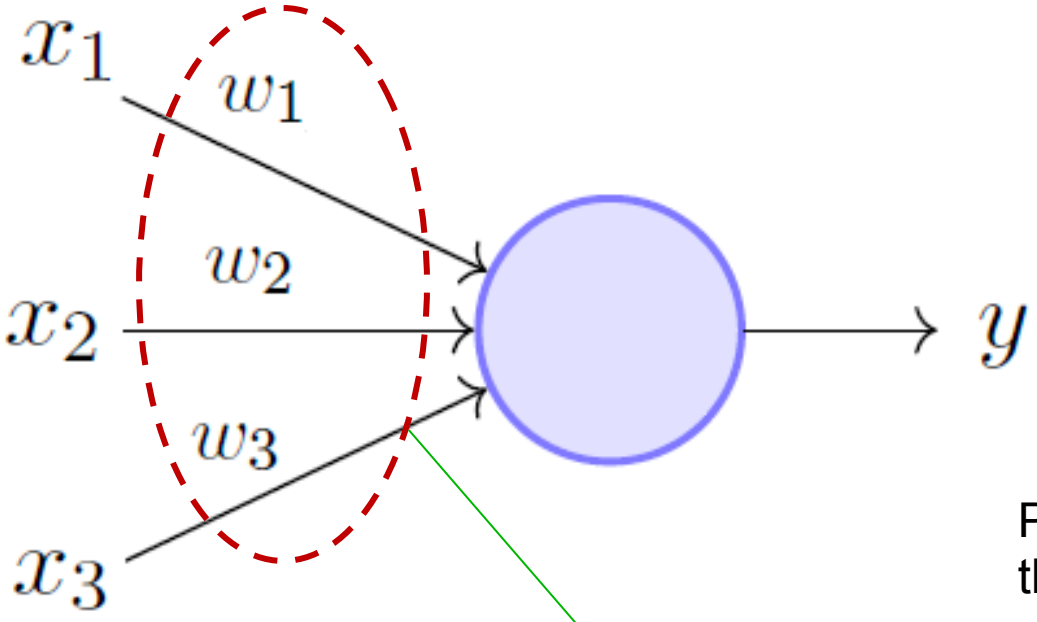
Table 9: Comparison of different regularization methods on MNIST.

Source: A paper titled "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Max-Norm regularization

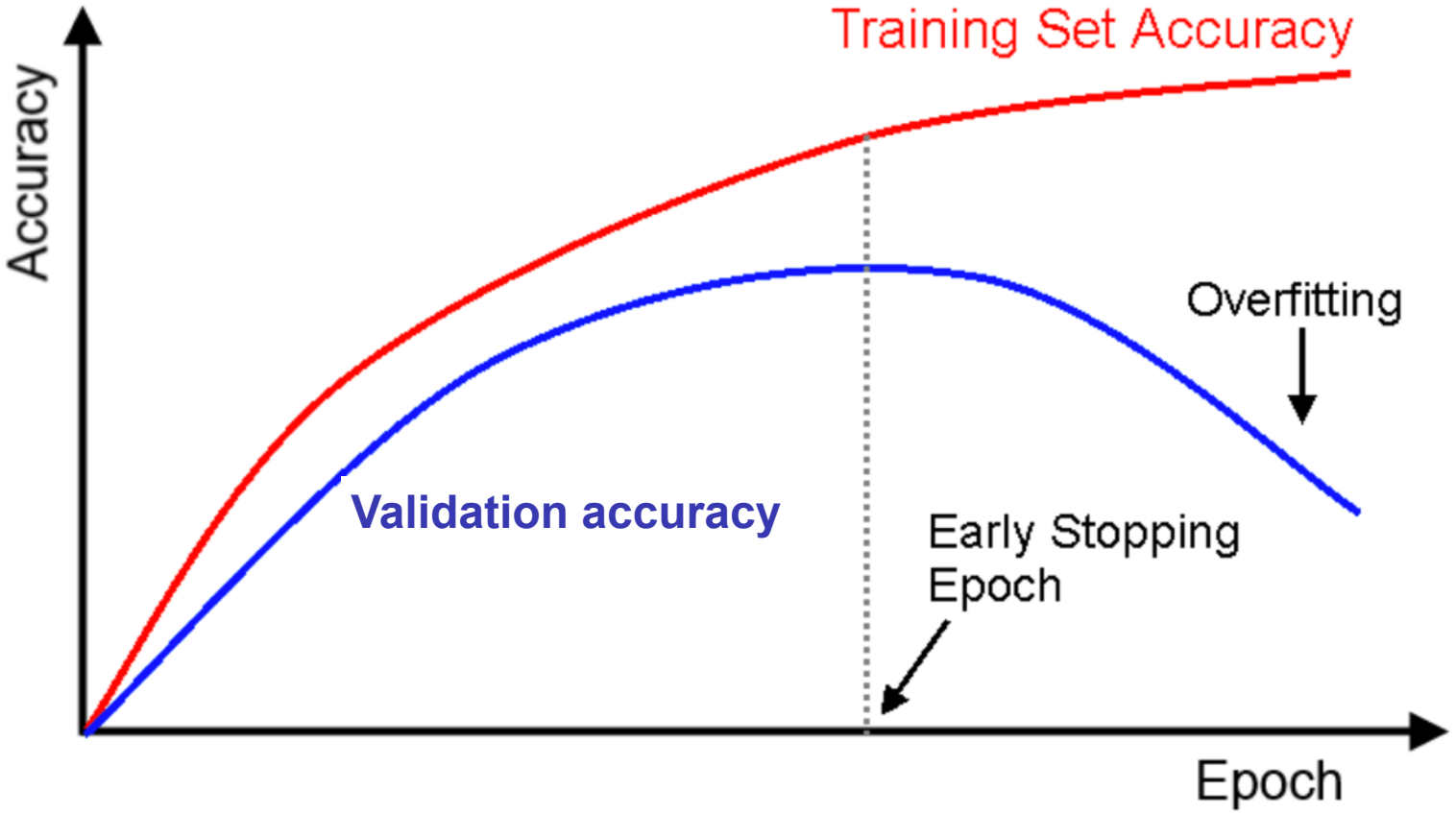
Max-Norm: enforces bounded incoming weights

for every single neuron



Project back to the constrained region if the weights go out

Early Stopping

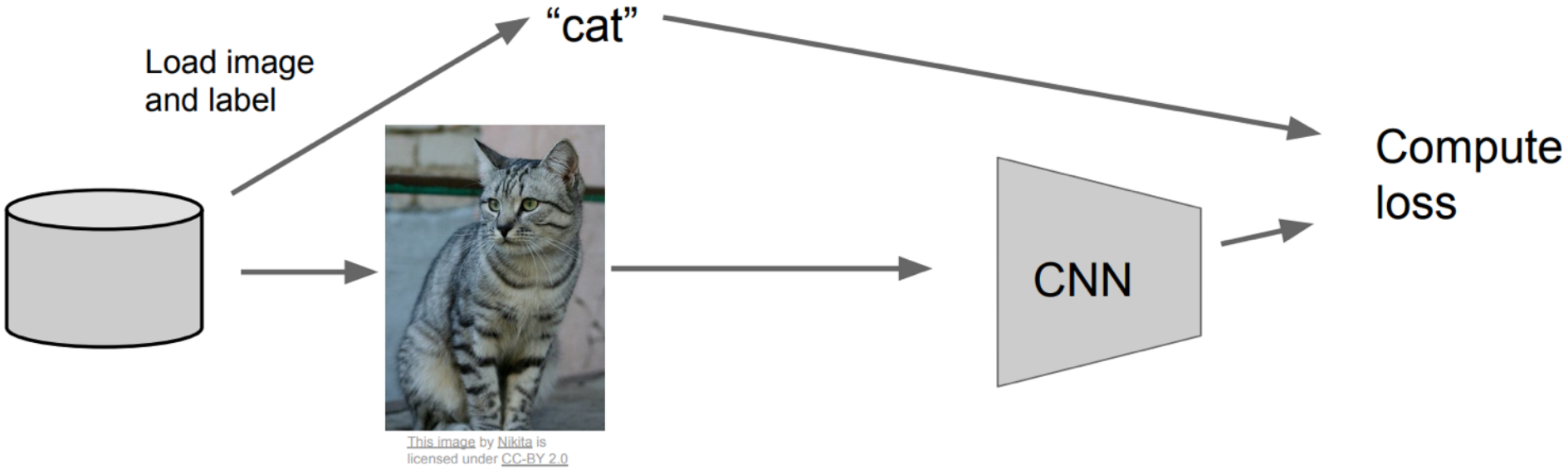


i) Keep track of validation accuracy.
ii) Stop training when validation acc is maximized

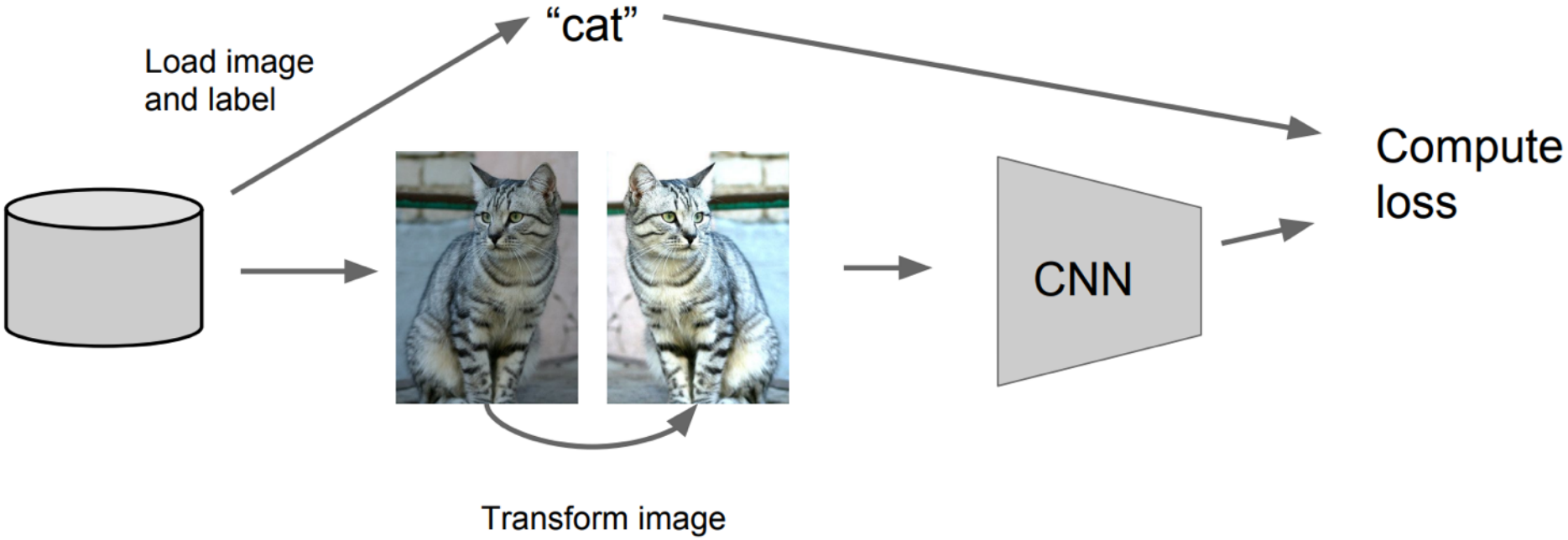
Data Augmentation

- One of the most important tools
  - Can increase model accuracy **10-20%**
- Helps you extract the most out of your data
- Found huge success in deep learning
  - Can make DNNs much less data-intensive

Data Augmentation

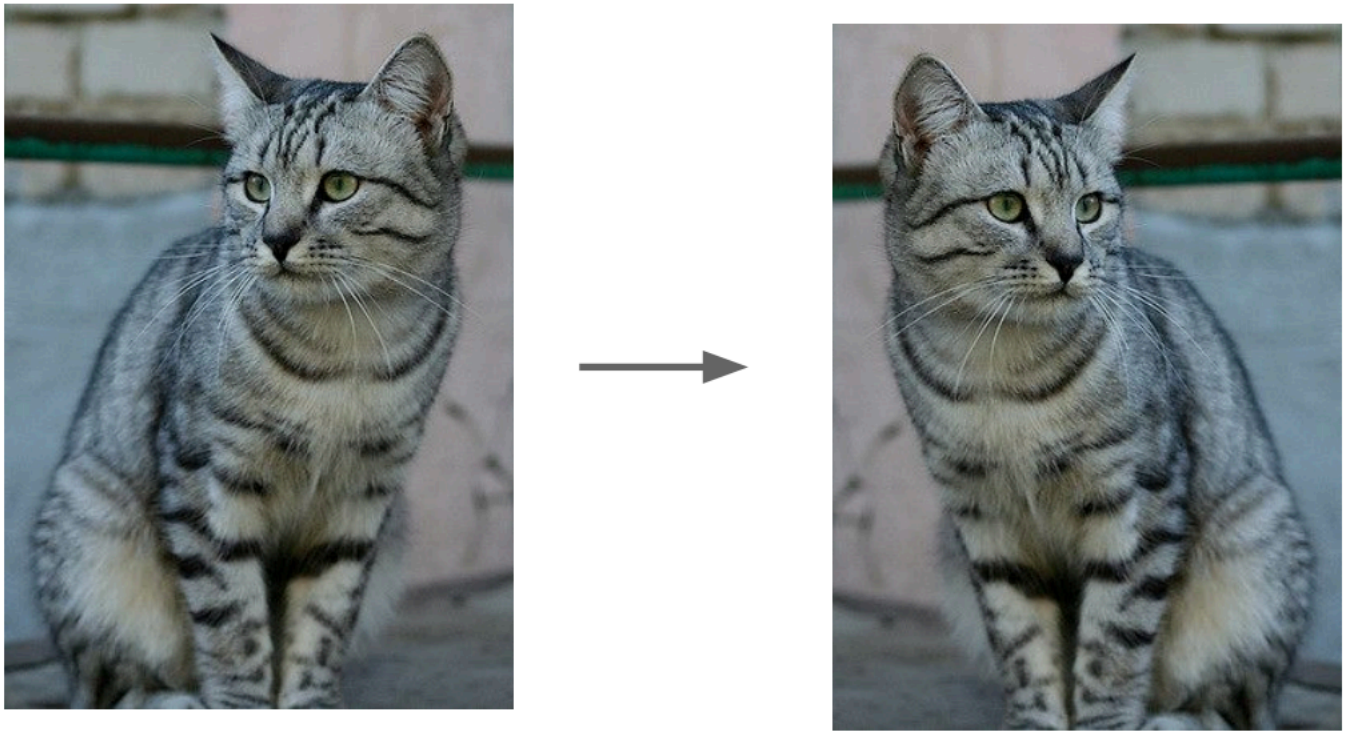


Data Augmentation



Data Augmentation

Horizontal Flips



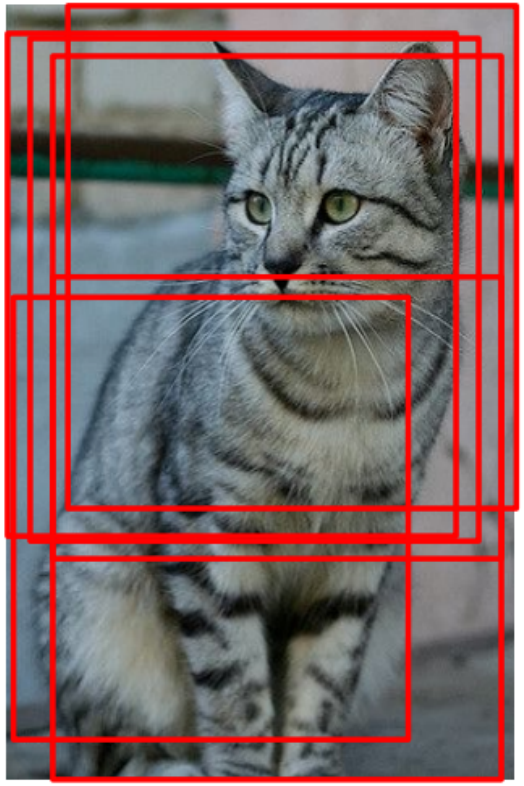
Data Augmentation

Random crops and scales

Training: sample random crops / scales

CNN with 224 x 224 input

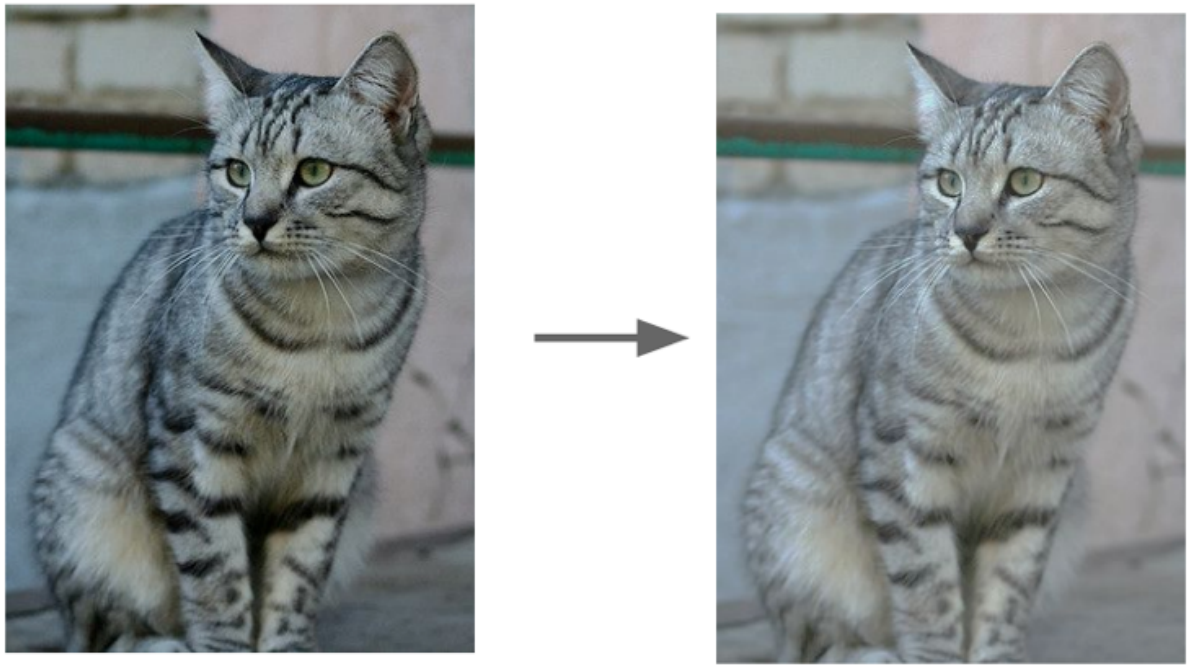
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Data Augmentation

- Random mix/combination of:
  - Translation
  - Rotation
  - Stretching
  - Len distortion
  - …
- Get Creative for your own problem!

Data Augmentation

- Augmentation applies a transformation T on input x
  - $x \rightarrow T(x)$ where T can be potentially random
    - E.g., randomly select which pixels we crop the image

- Key idea: x and $T(x)$ should have the same label

- Dataset grows
  - From (x, y) we generated a new example $(T(x), y)$

Data Augmentation

- Risk minimization

$$\mathscr{L}(f) = \mathbb{E}\left[\ell(f(x), y)\right]$$

$$= \int \ell(f(x), y) p^\star(x, y)\, dx dy$$

- We approximate the unknown $p^\star$ with the empirical distribution

$$\hat{p}(x, y) = \frac{1}{n} \sum_{i=1}^{n} \delta(x - x_i)\delta(y - y_i)$$

Data Augmentation

- Risk minimization

$$\mathscr{L}(f) = \mathbb{E}\left[\ell(f(x), y)\right]$$

$$= \int \ell(f(x), y) p^{\star}(x, y)\, dxdy$$

- With data augmentation, we approximate the unknown $p^{\star}$ with the smoothed empirical distribution

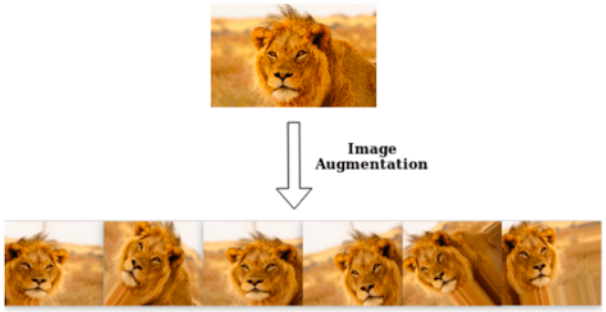
$$\hat{p}(x, y) = \frac{1}{n} \sum_{i=1}^{n} p(x \mid x_i, T) \delta(y - y_i)$$

$p(x \mid x_i, T)$ don't just concentrate on a single data point

Data Augmentation - Training

SGD with data augmentation in practice:

- There can be infinitely many augments
  - e.g. rotation or crop can be continuous
  - We don't have enough time for that!
  - **Simple Idea:** Randomly sample augmentations



1. Draw a sample (x, y)
2. Get (randomized) augmented data $\bar{x} = T(x)$
3. Update model parameters

$$w_{t+1} = w_t - \eta \nabla \ell(f(\bar{x}; w_t)y)$$

Initialization

- How to initialize neural network weights?

- Idea: Initialize all the weights to 0?
  - Problem 1: Training stops since all gradients are 0.
    - Recall that $\frac{\partial f}{\partial w} = \sigma\left(Ux\right)$ and $\frac{\partial f}{\partial U} = \left(w \odot \sigma'(Ux)\right)x^\top$

Initialization

- How to initialize neural network weights?

- Idea: Initialize all the weights to 0?
  - Problem 1: Training stops
  - Problem 2: Lack of diversity in weights—learn the same thing multiple times

Initialization

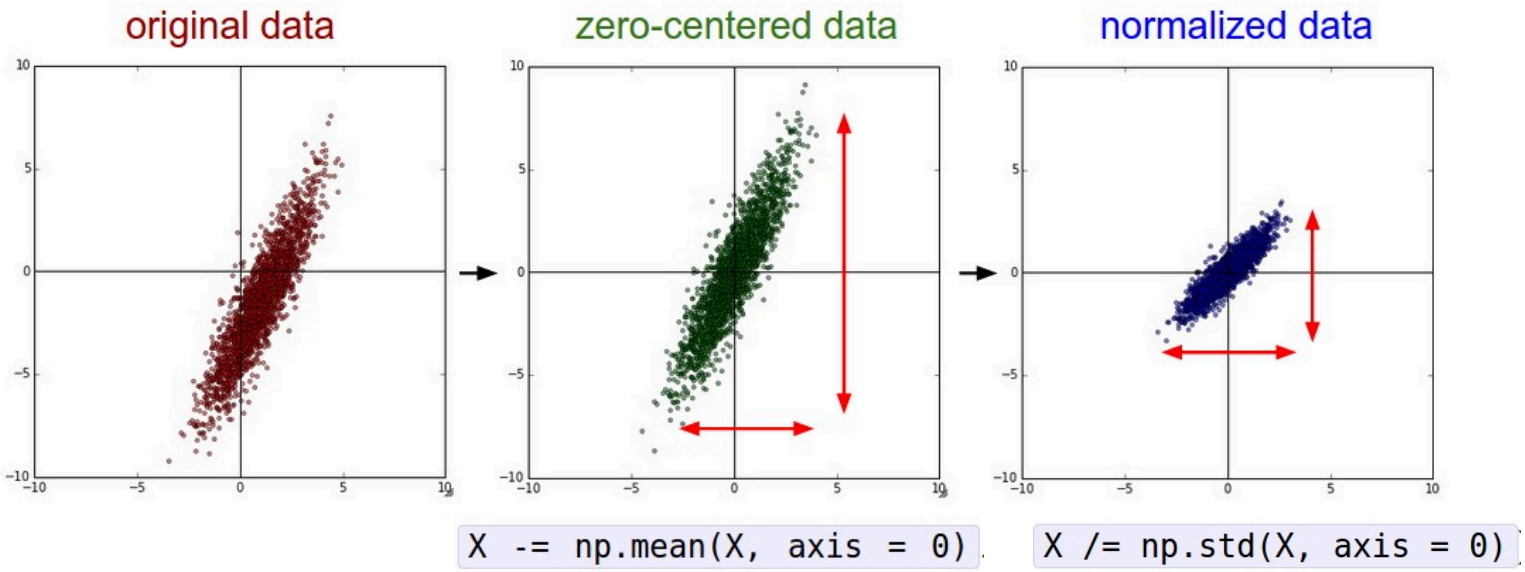
- Idea: How about initialize $w_i \sim \mathcal{N}(0, \sigma^2)$
- Problem: Variance blows up
- Consider $o = \sum_{i=1}^{n_{in}} w_i x_i$, where $\mathbb{E}[x_i] = 0, \mathbb{V}[x_i] = \gamma^2$ and x_i, w_i are independent
- We have $\mathbb{E}[o] = 0$ and $\mathbb{V}[o] = n_{in}\sigma^2\gamma^2$

Initialization

- LeCun initialization (1990s): $w_i \sim \mathcal{N}\left(0, \frac{1}{n_{in}}\right)$
- Xavier initialization (2010): $w_i \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right)$
- He initialization (2015): $w_i \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$

Data Preprocessing

Data Preprocessing



(Assume X [NxD] is data matrix,
each example in a row)

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance hidden features? just make them so."

consider a batch of hidden features at some layer. To make each dimension zero-mean unit-variance, apply:

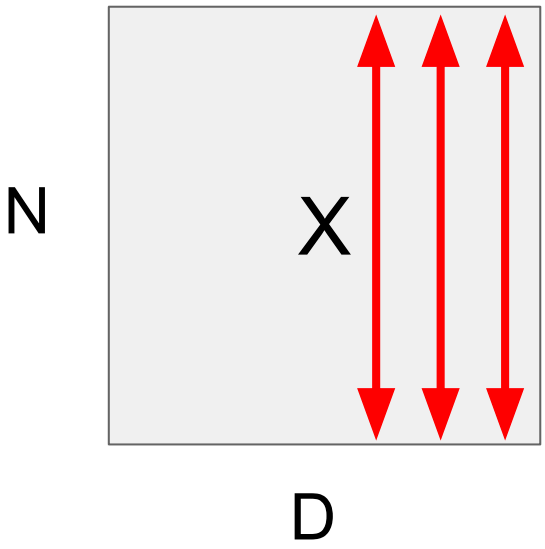
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance hidden features? just make them so."



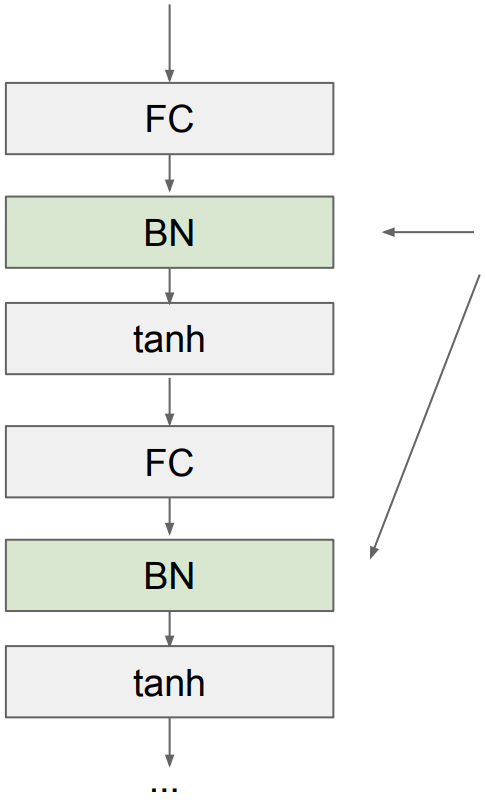
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

A linear transformation on $\hat{x}$ with learnable parameters γ, β

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

At inference time, batch statistics (save running avg. mean/std) are used

Acknowledgements

A lot of content from online resources

– Fei Fei Li's slides
http://cs231n.stanford.edu/slides/2019/

– Justin Johnson's slides
https://web.eecs.umich.edu/~justincj/slides/eecs498/