10707
Deep Learning: Spring 2020

Andrej Risteski

Machine Learning Department

Lecture 6: Advanced optimization

Supervised learning

Empirical risk minimization approach:
minimize a **training** loss l over a class of **predictors** $\mathcal{F}$:

$$\hat{f} = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{(x,y):\text{training samples}} l(f(x), y)$$

Three pillars:

(1) How expressive is the class $\mathcal{F}$? (**Representational power**)

(2) How do we minimize the training loss efficiently? (**Optimization**)

(3) How does $\hat{f}$ perform on unseen samples? (**Generalization**)

Recap:continuous optimization

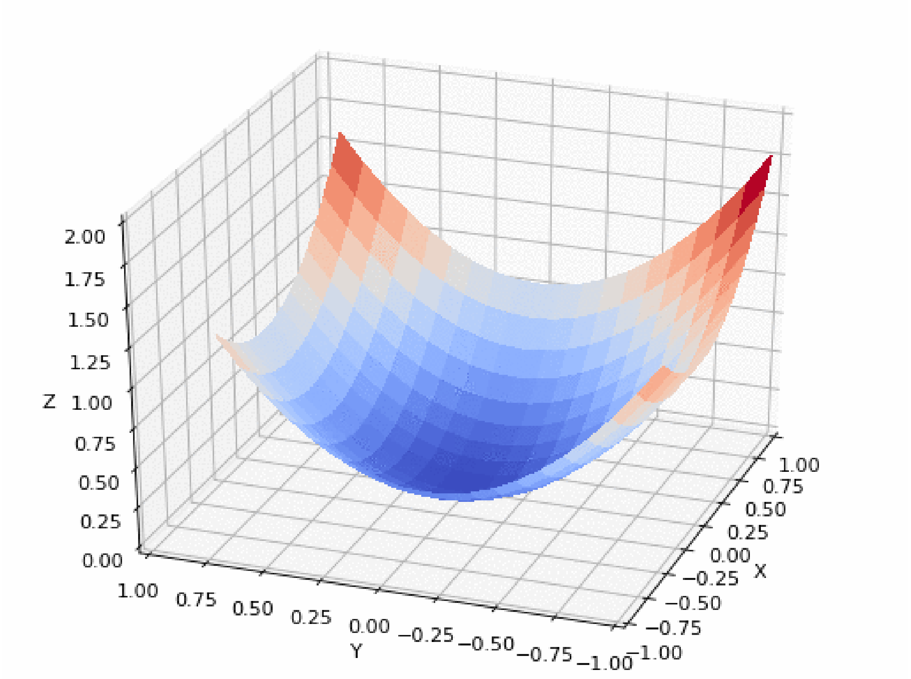
The typical training task in ML can be cast as: $\min_{x \in \mathbb{R}^d} \quad f(x)$

Most algorithms we will look at are iterative: they progressively pick points $x_1, x_2, \dots$ that are supposed to bring “improvement”.

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

Gradient descent

Recap:continuous optimization

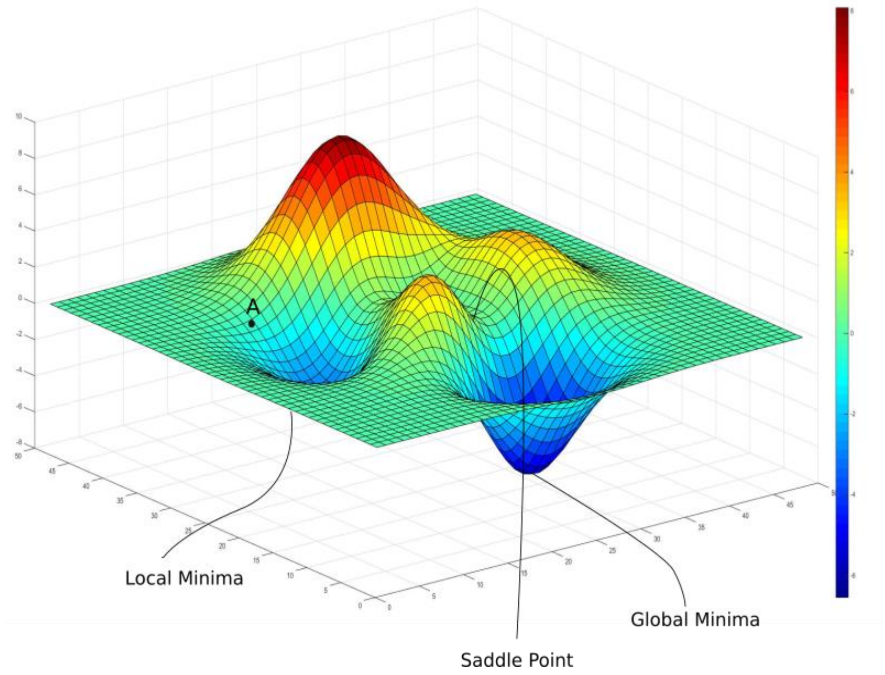


What can we hope for, in the case that $\eta \to 0$?

We stop moving when $\nabla f(x_t) \approx 0$: these care called **stationary points**.

What kinds of stationary points are there?

Recap:continuous optimization



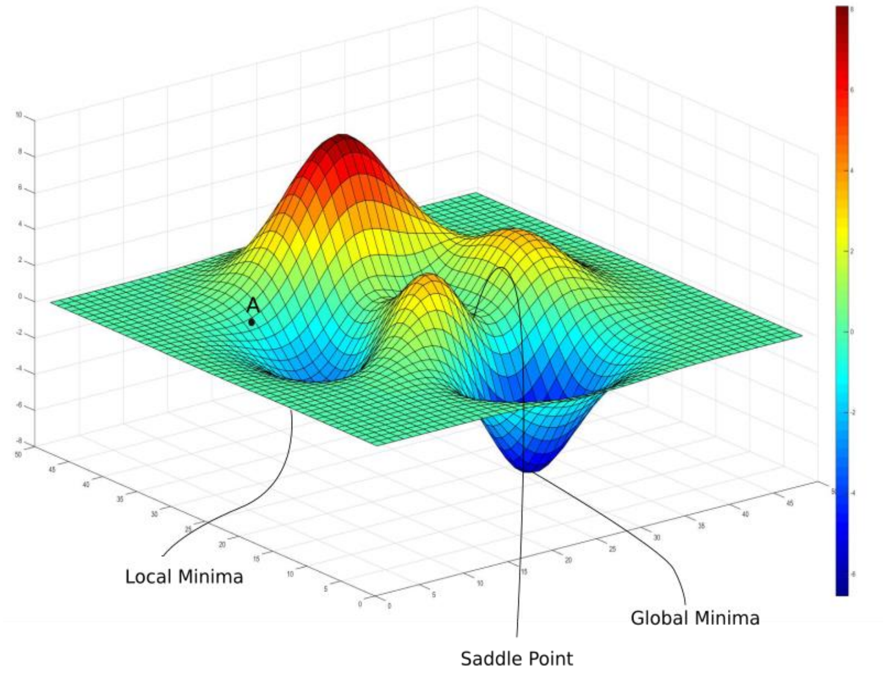
Global minimum: actual minimizer, namely $f(\hat{x}) \le f(x), \forall x \in \mathbb{R}^d$

Local minimum: $f(\hat{x}) \le f(x),\ \forall x \text{ s.t. } ||x - \hat{x}|| \le \epsilon$ for some $\epsilon > 0$

Local maximum: $f(\hat{x}) \ge f(x),\ \forall x \text{ s.t. } ||x - \hat{x}|| \le \epsilon$ for some $\epsilon > 0$

Saddle points: stationary point that is *not* a local min/max.

Recap:continuous optimization



Global minimum: finding these in general is very hard (both in theory – NP-hard, as well as in practice)

Local minimum: seem to work quite well often. Some theoretical understanding of why in very restricted cases.

Saddle points: typically bad, arise from invariances in input. Want to avoid these. (Stay tuned.)

Why are local minima good?

In short: we don't know.

Some results:

(Kawaguchi '16): **Linear** neural networks (that is, neural networks with linear activation functions): all local minima are also global minima.

(Hardt-Ma '18): **Linear** residual networks (that is, ResNets with linear activation functions): all stationary points are also global minima.

Evidence for the other side:

(Safran-Shamir '18, Yun-Sra-Jadbabie '19): **Non-linear** nets with:

- Virtually any non-linearity.
- ***Even*** Gaussian inputs (about as simple as it gets)
- Labels ***generated*** by a neural net of the same architecture we are training can have **many** bad local minima.

Overparametrization (training larger net) breaks these examples.

Why are local minima good?

In short: we don't know.

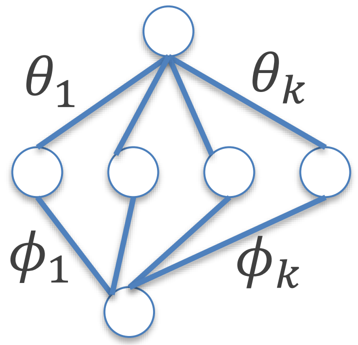
Gradient descent w/ overparametrization:

(Allen-Zhu et al '18, Du et al '18): **Sufficiently overparametrized** neural networks (that is neural nets with # parameters >> # of training samples, training from random initialization, converge to small training error solutions.

Interaction with generalization is still evolving/unclear.

Intuition: saddle points arise due to symmetry

Warning: intuition only, not formal in any way!



Permutation symmetry: loss function is

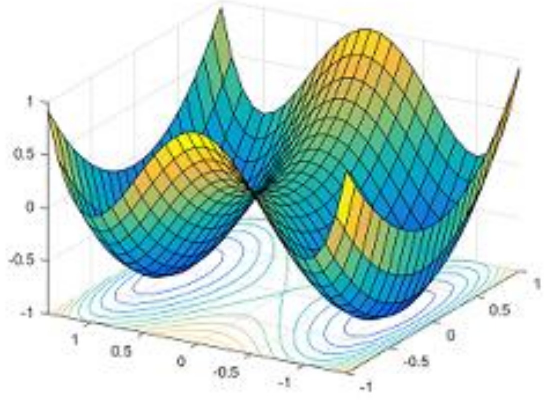
$$f(\theta_1, \theta_2, \ldots, \theta_k, \phi_1, \ldots, \phi_k).$$

Clearly same loss if we permute neurons, e.g.

$$f(\theta_3, \theta_1, \theta_2 \ldots, \theta_k, \phi_3, \phi_1, \phi_2 \ldots, \phi_k)$$

However, averaging likely will give different (larger) loss

$$f\left(\frac{\theta_1+\theta_3}{2}, \frac{\theta_2+\theta_1}{2}, \frac{\theta_3+\theta_2}{2} \ldots, \theta_k, \frac{\phi_1+\phi_3}{2}, \frac{\phi_2+\phi_1}{2}, \frac{\phi_3+\phi_2}{2}, \ldots, \phi_k\right).$$



Isolated minima corresponding to permutations, connected via a “flat” region inbetween. (We’ll see later this intuition may be deficient in some ways.)

First order methods on the cheap: stochastic gradient descent

In the supervised learning setup, function we are optimizing has the form of a sum:

$$\min_{\theta} \sum_{(x,y):\text{training samples}} l\left(f_\theta(x), y\right)$$

Stochastic gradient descent

```
- Initialize: $\theta_0 := \{W^{(1)}, b^{(1)}, \dots, W^{(L+1)}, b^{(L+1)}\}$
- For t=1 to T
    - Pick a uniformly random training example $(x, y)$:
        - Set $\theta_{t+1} = \theta_t - \eta \nabla_\theta l(f_\theta(x), y))$
```

Random variable w/ expectation true gradient

First order methods on the cheap: stochastic gradient descent

In the supervised learning setup, function we are optimizing has the form of a sum:

$$\min_{\theta} \sum_{(x,y):\text{training samples}} l\left(f_\theta(x), y\right)$$

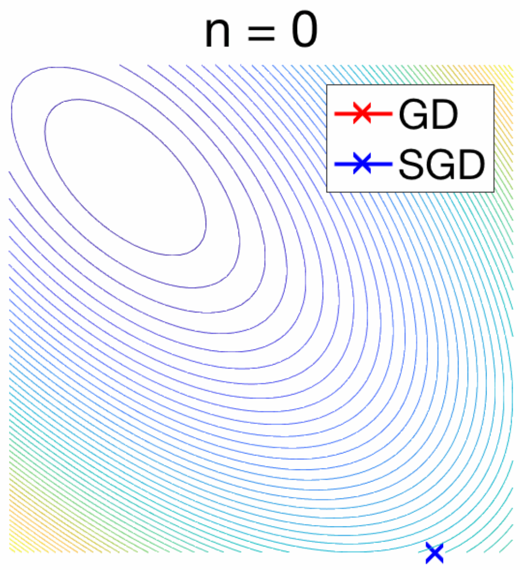
Stochastic gradient descent

```
- Initialize: $\theta_0 := \{W^{(1)}, b^{(1)}, \dots, W^{(L+1)}, b^{(L+1)}\}$
- For t=1 to T
    - Pick a uniformly random training example $(x, y)$:
        - Set $\theta_{t+1} = \theta_t - \eta \nabla_\theta l(f_\theta(x), y))$
```

*Actually, typically, one takes a **mini-batch** of B samples, and calculates gradient over those samples*

First order methods on the cheap: stochastic gradient descent

Due to randomness, more jittery than gradient descent:



Bug? Longer convergence time

Feature? Regularization effect / better generalization.

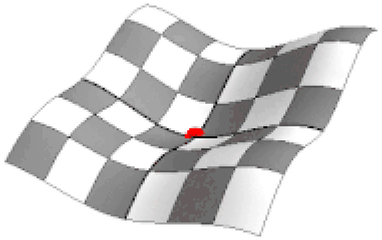
Feature? Can help escape saddles.

Recap: local minima

Second order checks: Hessian approximates a function to second order

Taylor's thm: $f(x+\Delta) \approx f(x) + \Delta^T \nabla f(x) + \frac{1}{2}\Delta^T \nabla^2 f(x)\Delta + O(||\Delta||^3)$

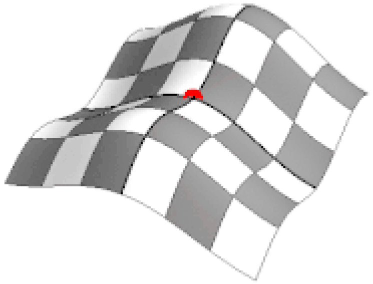
$$\approx f(x) + \frac{1}{2}\Delta^T \nabla^2 f(x)\Delta + O(||\Delta||^3)$$



If $\nabla^2 f(x) \succ 0$: for any direction Δ, and small enough $||\Delta||$

$\Delta^T \nabla^2 f(x)\Delta + O(||\Delta||^3) \geq 0$, so $f(x+\Delta) > f(x)$

Local minimum! (Flipped for local maximum)



If $\nabla^2 f(x)$ has both positive and negative eigenvalues:

Saddle point (not a local minimum/maximum)

If neither of these attains, test is inconclusive!

First order methods on the cheap: stochastic gradient descent

Feature: can help escape (second-order) saddle points

- Consider $f(x) = \frac{1}{2} x^\top A x$, A is *strict saddle*: has **negative** eval
- Gradient descent updates: $x_{t+1} = x_t - \eta \nabla f(x_t) = (I - \eta A) x_t$
- Write $x_t = \sum_i \mu_i^t v_i$, for the eigenbasis $\{(v_i, \lambda_i)\}_i$ of A.
- Then, $x_{t+1} = \sum_i (1 - \eta \lambda_i) \mu_i^t v_i$
- In other words, $x_T = \sum_i (1 - \eta \lambda_i)^T \mu_i^0 v_i$

In other words: component in direction of negative eigenvalue grows – we don't get stuck!

Stochastic noise helps no component to get too small.

[Can be formalized for fresh noise, *Jin, Ge, Netrapalli, Jordan, Kakade '17*]

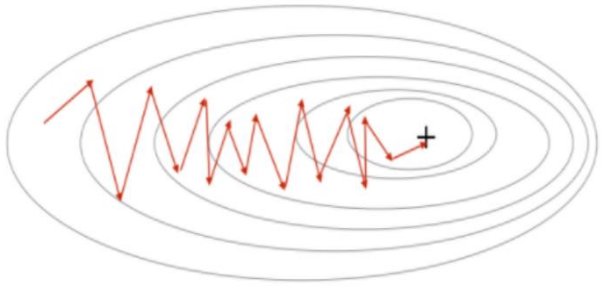
Recap:continuous optimization

The typical training task in ML can be cast as: $\min\limits_{x \in \mathbb{R}^d} \quad f(x)$

Most algorithms we will look at are iterative: they progressively pick points $x_1, x_2, \ldots$ that are supposed to bring "improvement".

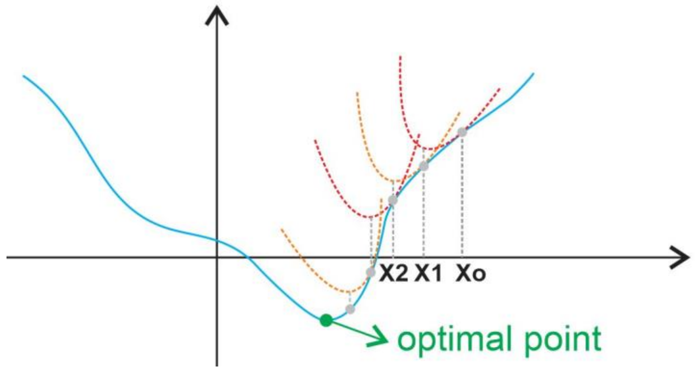
$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

Gradient descent



Set $x_{t+1} = x_t - \eta\left(\nabla^2 f(x_t)\right)^{-1} \nabla f(x_t)$

Newton's method.



Second order methods on the cheap: AdaGrad

Recall the problem with Newton: inverting large matrices

Set $x_{t+1} = x_t - \eta\left(\nabla^2 f(x_t)\right)^{-1} \nabla f(x_t)$

Newton's method.

AdaGrad (simplified): have a separate learning rate for every parameter

Set $x_{t+1} = x_t - \eta G_t^{-1} \nabla f(x_t)$,

$$(G_t)_{ii} = \sqrt{\sum_{j=1}^{t-1} (\nabla f(x_t)_{ii})^2}\ , \qquad \text{else } 0$$

$$(G_t)^{-1}{}_{ii} = \frac{1}{\sqrt{\sum_{j=1}^{t-1} (\nabla f(x_t)_{ii})^2}}, \qquad \text{else } 0$$

Second order methods on the cheap: RMSProp

AdaGrad (simplified): have a separate learning rate for every parameter

$$\text{Set } x_{t+1} = x_t - \eta G_t^{-1} \nabla f(x_t), \qquad (G_t)_{ii} = \sqrt{\sum_{j=1}^{t-1} (\nabla f(x_t)_{ii})^2}\ , \qquad \text{else } 0$$

RMSProp: since we keep summing the norms of the gradient, the learning rate could keep getting smaller. [**NOTE**: For convex functions, this is standard and in fact works.] Instead, keep a weighted exponential avg.

$$(g_{t+1})_i = \beta (g_t)_i + (1-\beta)(\nabla f(x_t)_{ii})^2$$

$$(x_{t+1})_i = (x_t)_i - \frac{\eta \nabla f(x_t)_i}{\sqrt{(g_{t+1})_i}}$$

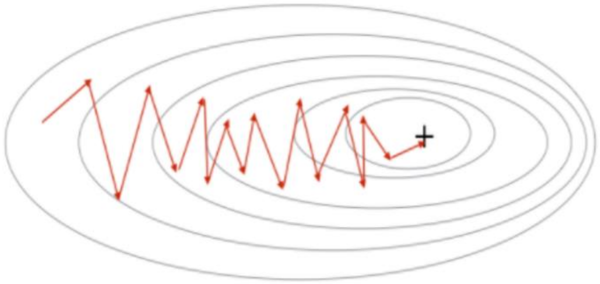
Recap:continuous optimization

The typical training task in ML can be cast as: $\min_{x \in \mathbb{R}^d} \quad f(x)$

Most algorithms we will look at are iterative: they progressively pick points $x_1, x_2, \ldots$ that are supposed to bring "improvement".

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

Gradient descent



Momentum (Polyak and Nesterov)

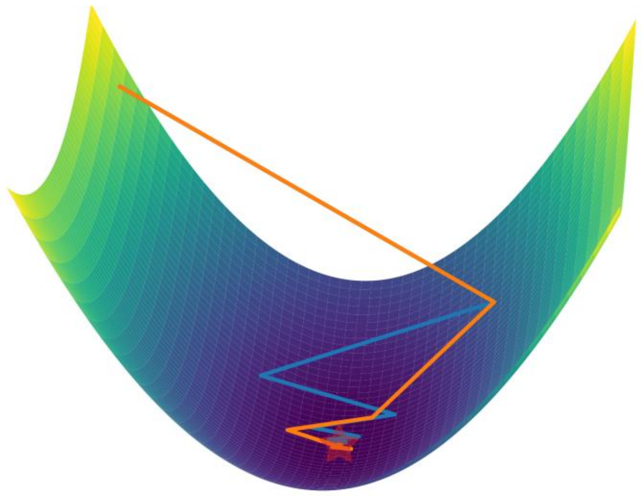
Linear combination of prior gradients + current one

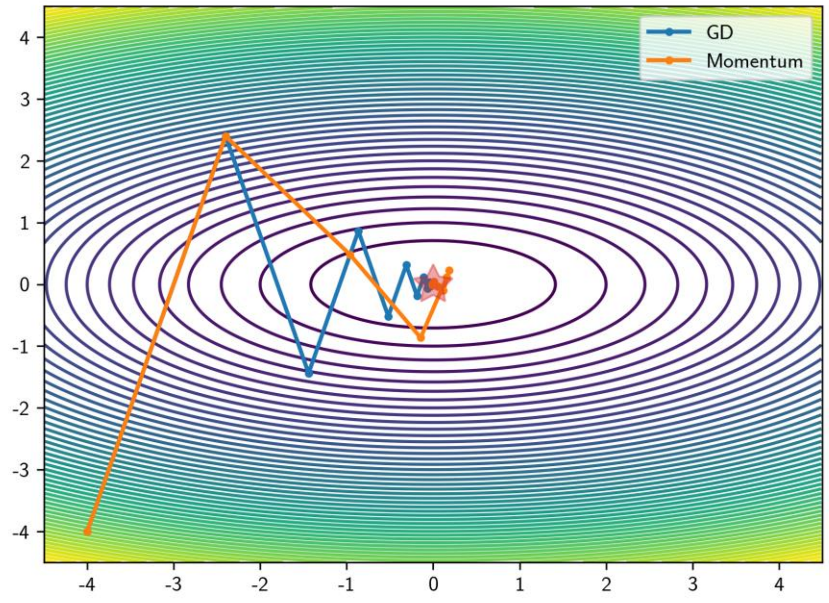
$$v_{t+1} = -\nabla f(x_t) + \beta v_t$$
$$x_{t+1} = x_t + \eta v_{t+1}$$

Evaluate gradient at a "lookahead" point

$$v_{t+1} = -\nabla f(x_t + \beta\, v_t) + \beta\, v_t$$
$$x_{t+1} = x_t + \eta\, v_{t+1}$$

Recap:continuous optimization





Momentum vs grad. descent
https://tangbinh.github.io/01/04/Optimizers.html

Momentum + AdaGrad = Adam

Adam: Frankenstein momentum and AdaGrad together:

$$v_{t+1} = \beta_1 v_t + (1-\beta_1)\nabla f(x_t) \longrightarrow \textit{Momentum}$$

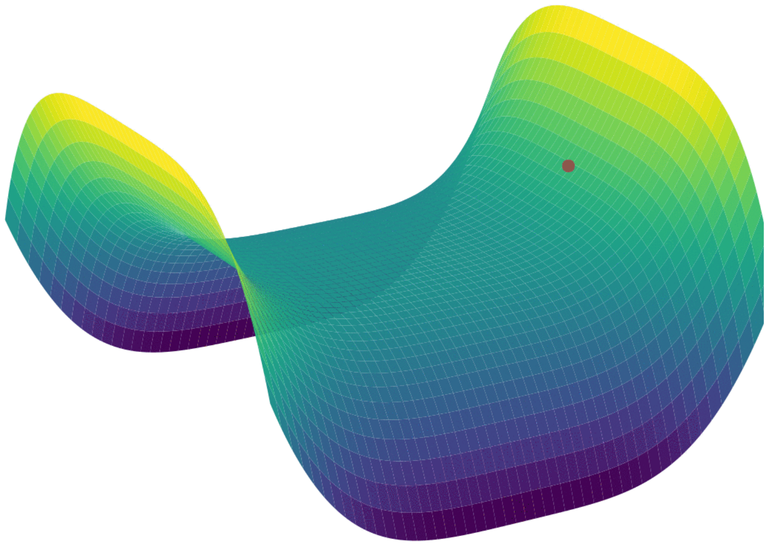
$$(g_{t+1})_i = \beta_2 (g_t)_i + (1-\beta_2)(\nabla f(x_t)_{ii})^2 \longrightarrow \textit{AdaGrad}$$

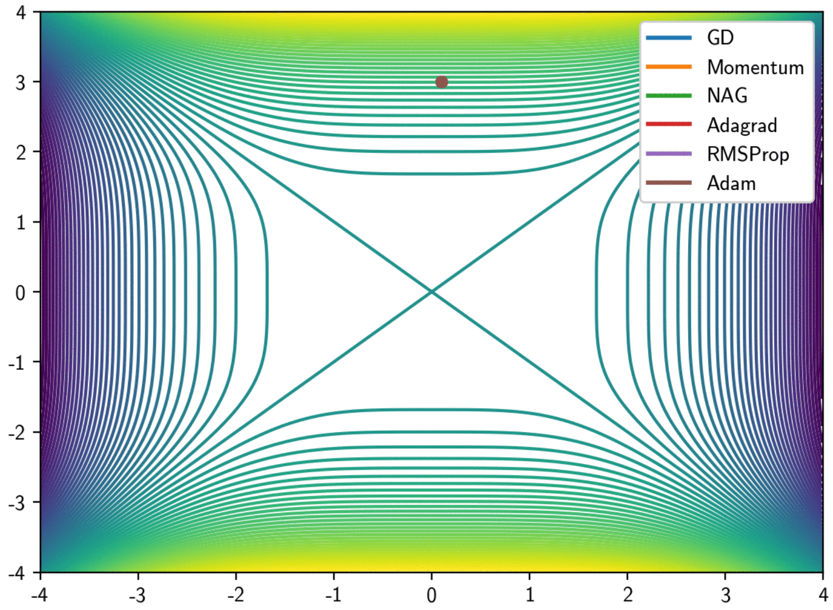
$$(x_{t+1})_i = (x_t)_i - \frac{\eta v_{t+1}}{\sqrt{(g_{t+1})_i}} \longrightarrow \textit{Combine the two}$$

Default choice nowadays.

It actually does **not** converge, even on convex instances without some modification of updates. (*Reddi, Kale, Kumar, Best paper at ICLR '18*)

Moving around saddles with various methods

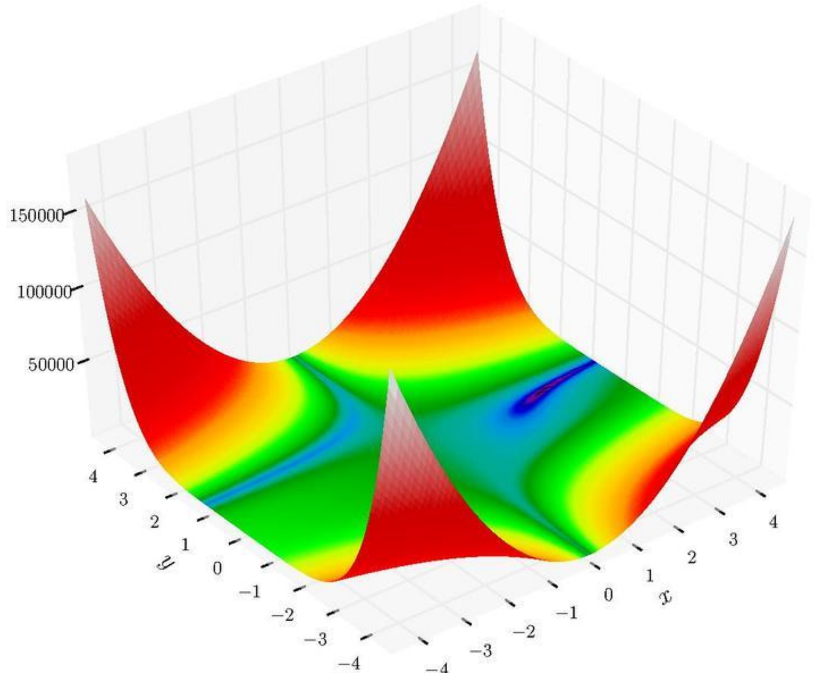


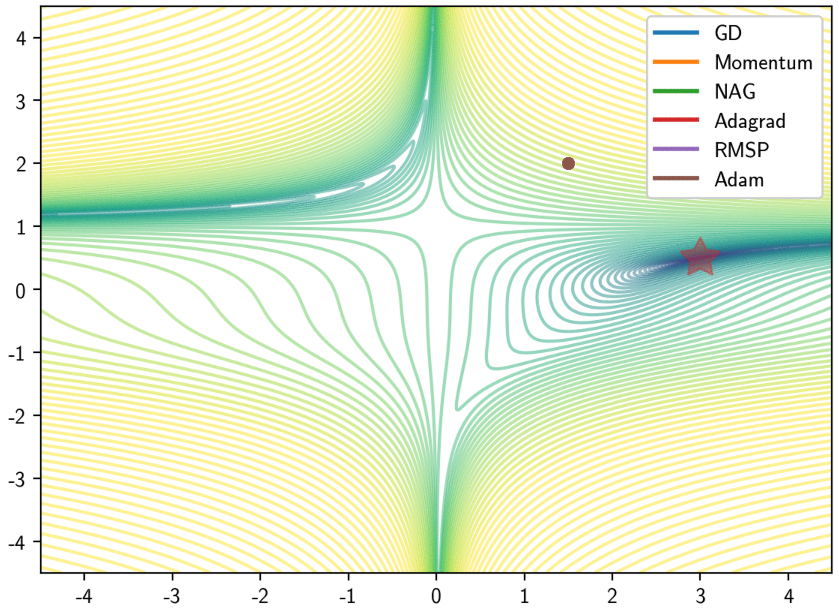


https://tangbinh.github.io/01/04/Optimizers.html

Loss function $f(x,y) = y^4 - x^4$, saddle point at (0,0), minimized at x=∞.

Beale's function using various methods





https://tangbinh.github.io/01/04/Optimizers.html

Beale's function minimized at (3,0).

Are these actually helpful?

Remember, what we are trying to do is minimize *test error*.
There's still the question of whether the solution generalizes well.

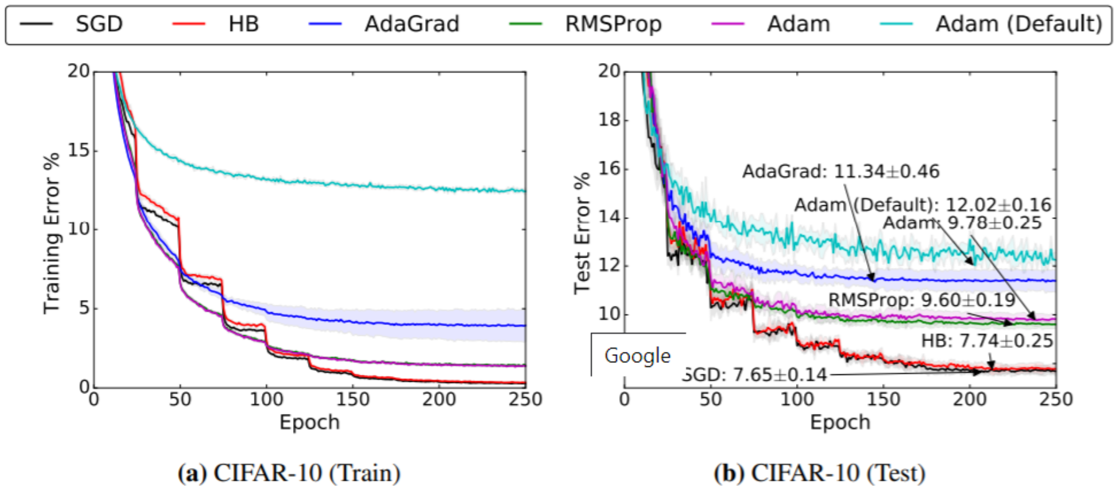


Figure 1: Training (left) and top-1 test error (right) on CIFAR-10. The annotations indicate where the best performance is attained for each method. The shading represents ± one standard deviation computed across five runs from random initial starting points. In all cases, adaptive methods are performing worse on both train and test than non-adaptive methods.

(Wilson, Roelofs, Stern, Srebro, Recht '18)

Batch Normalization

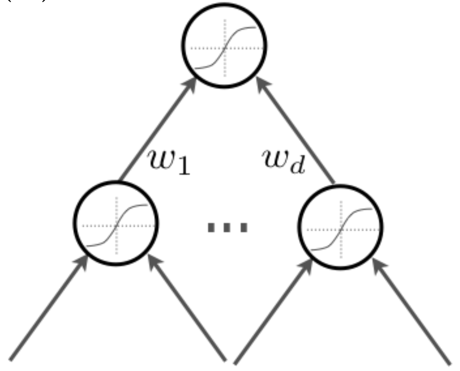
Normalizing/whitening (mean = 0, variance = 1) the inputs speeds up training (*Lecun et al. 1998*)

- Could normalization be useful at the level of the hidden layers?

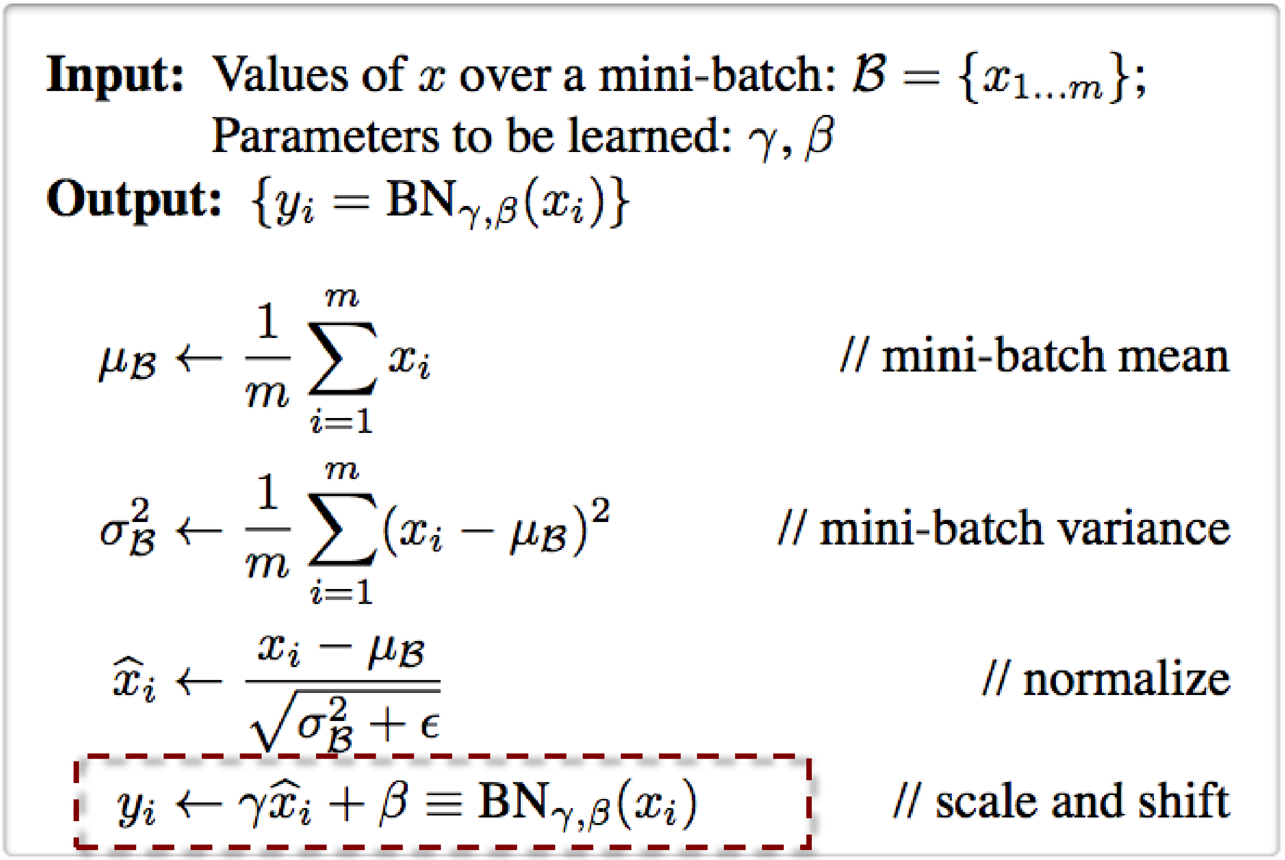
Batch normalization is an attempt to do that (*Ioffe and Szegedy, 2014*)

- each unit's pre-activation is normalized (mean subtraction, stddev division)
- during training, mean and stddev is computed for each minibatch
- backpropagation takes into account the normalization
- at test time, the global mean / stddev is used

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$



Batch Normalization



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

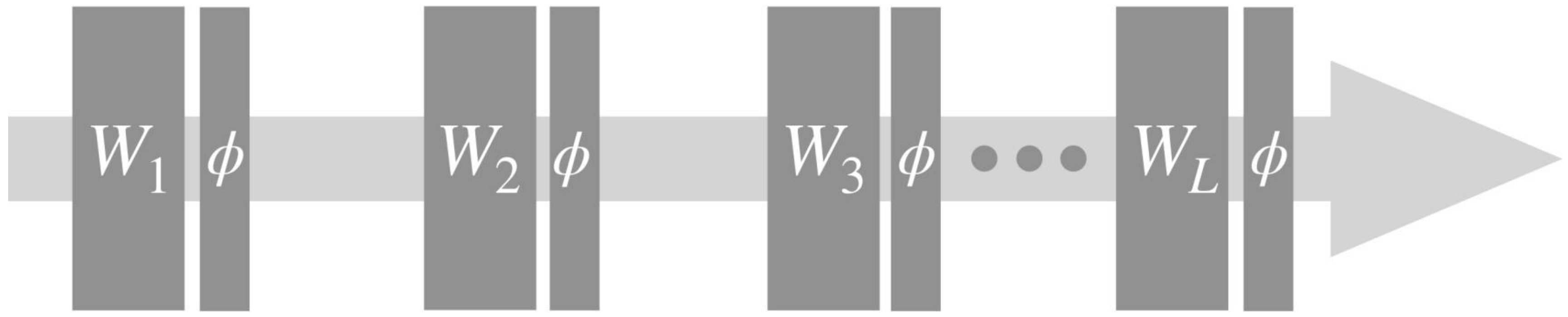
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

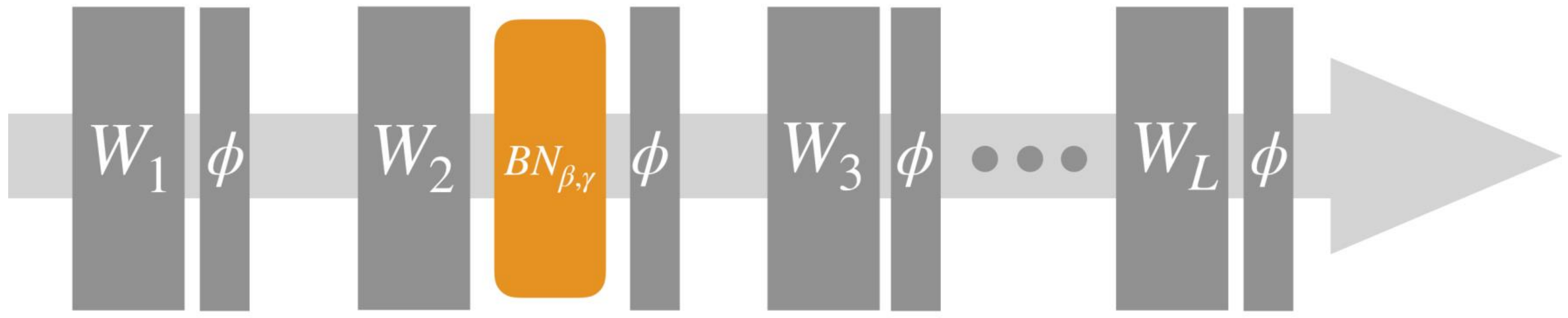
Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

Batch Normalization

Standard Network

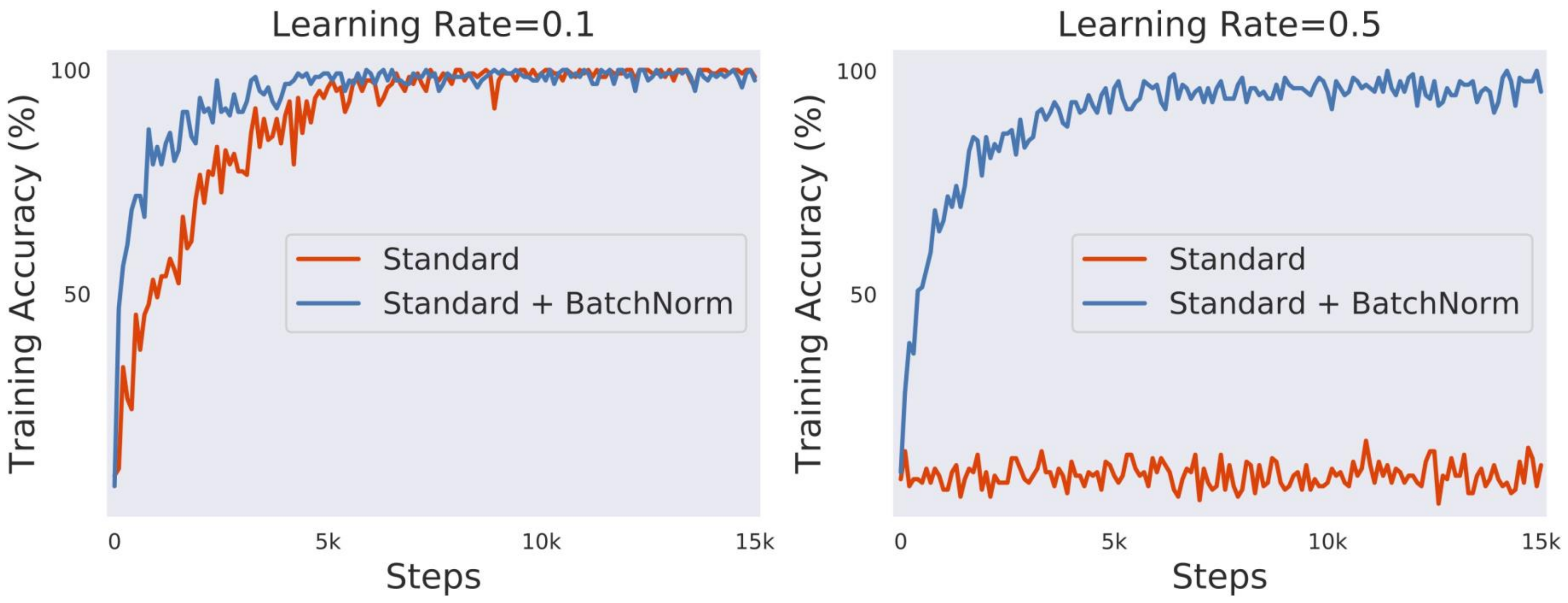


Adding a BatchNorm layer (between weights and activation function)



http://gradientscience.org/batchnorm/

Batch Normalization



http://gradientscience.org/batchnorm/

Batch Normalization

Why normalize the pre-activation?

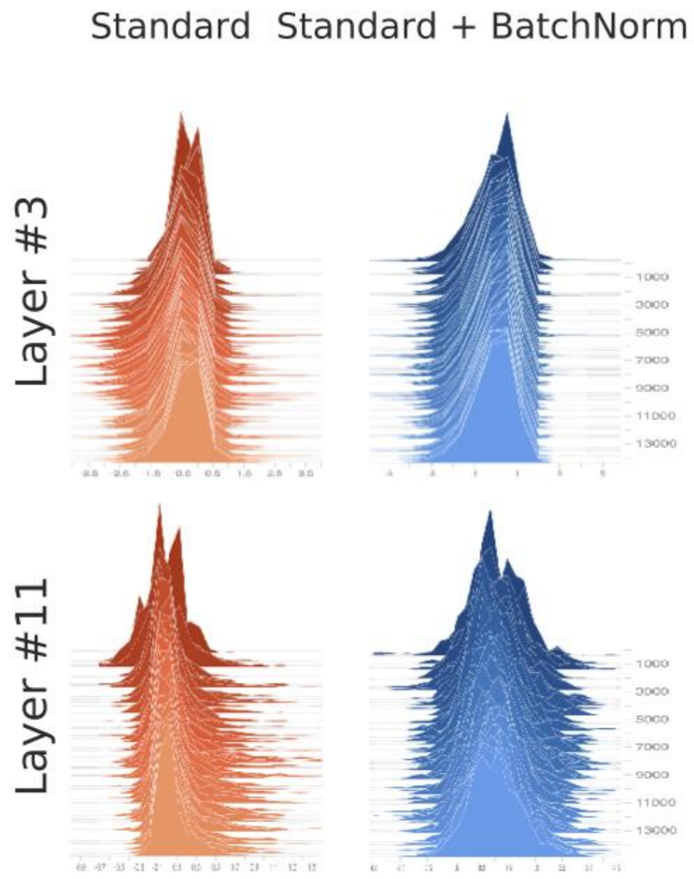
- Can help keep the pre-activation in a non-saturating regime
- Could condition Hessian better (in linear 1-layer nets, *Lecun et al. 1998)*

Why use minibatches?

- since hidden units depend on parameters, can't compute mean/stddev once and for all
- adds stochasticity to training, which might regularize

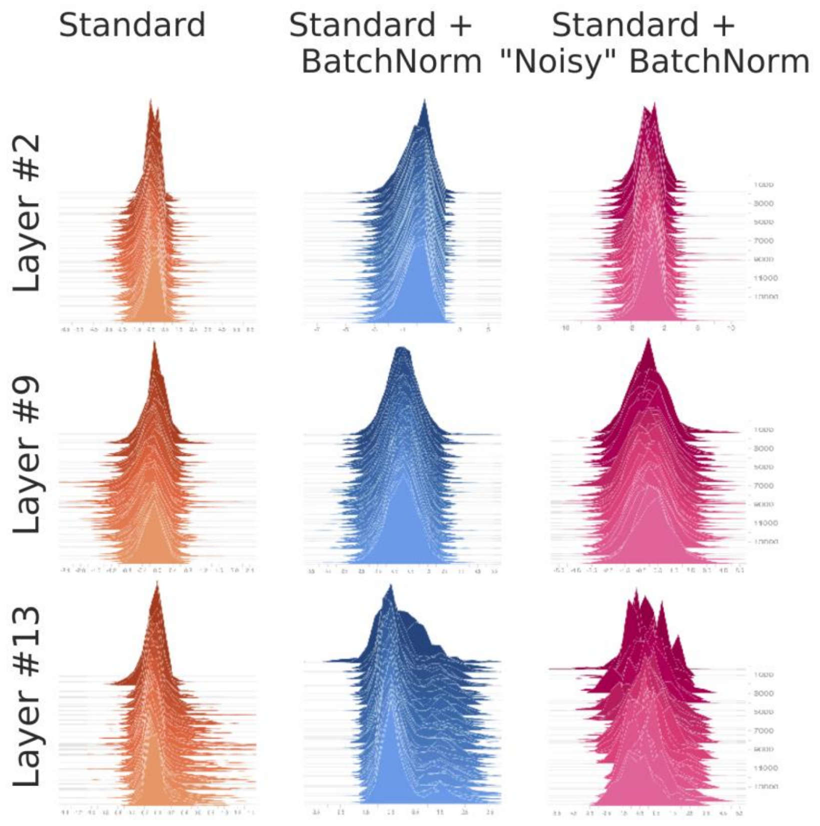
Batch Normalization

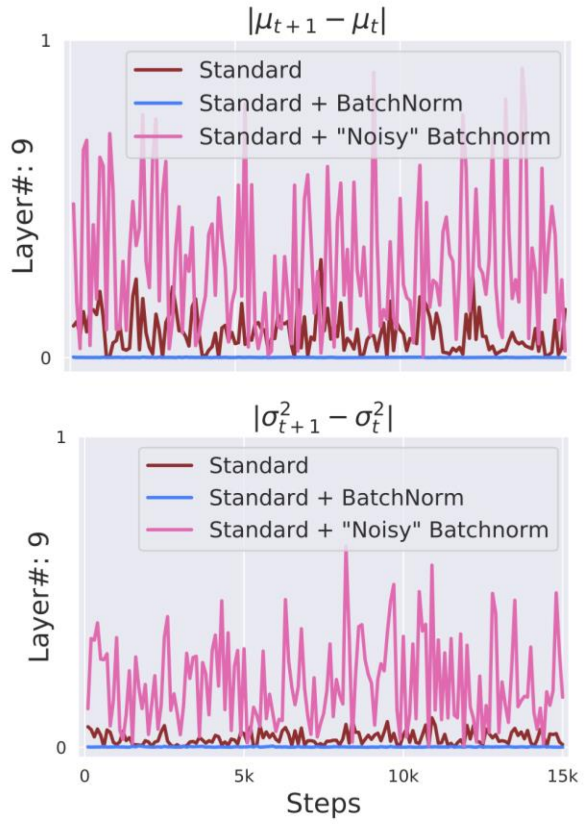
Recent observations cast some doubts on distribution shift explanation (*Santurka, Tspiras, Ilyas, Madry '18*)



Batch Normalization

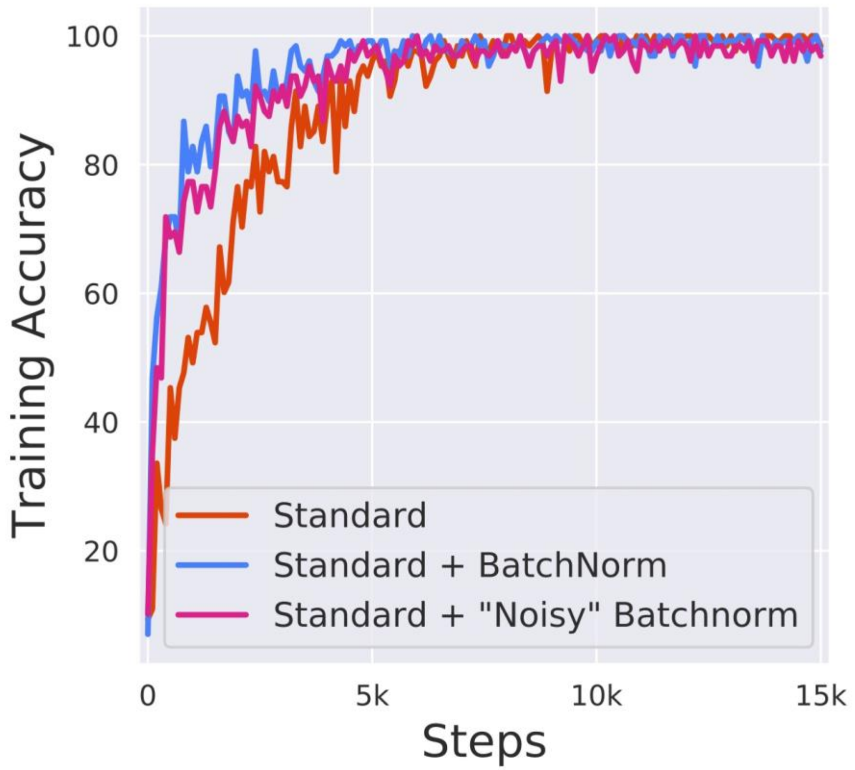
Simple experiment: inject non-stationary Gaussian noise after BN layer.
This removes the alleged whitening effect of BN layers.





Batch Normalization

Simple experiment: inject non-stationary Gaussian noise after BN layer. This removes the alleged whitening effect of BN layers.



Performance doesn't suffer…

Batch Normalization

Authors suggestions (can be formalized):

Lipschitz constant of the loss improves.

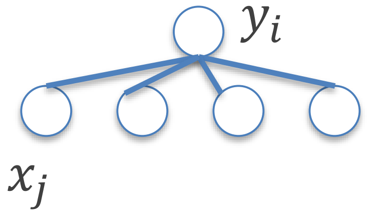
The **Hessian** evaluated in the direction of the gradient (i.e. second-order term in the Taylor expansion *decreases* in magnitude.
[i.e. first-order term gets comparatively bigger]

Initialization

Warning: lots of handwaiving, intuitions

Typically, the goal of initialization is to make sure the non-linearities are in the “active” (linear) regime.

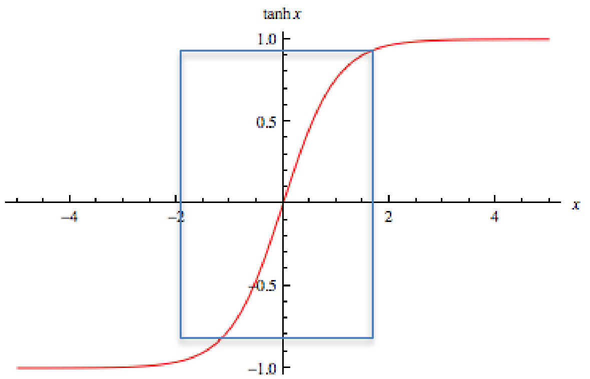
LeCun initialization: $\mathbb{E}[w_{ij}] = 0, \ \mathrm{Var}[w_{ij}] = \frac{1}{\text{fan−in}}$



If x_i’s are 0-mean, var-1, and $y_i = \sum_j w_{ij} x_j, \mathbb{E}[y_i] = 0, Var(y_i) = \sqrt{w_{ij}^2}$

Gradient of tanh is around 1 in the linear regime.

So, $\tanh(y_i)$ has variance 1.



Initialization

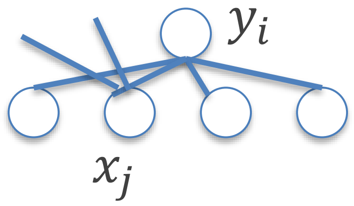
Warning: lots of handwaiving, intuitions

Typically, the goal of initialization is to make sure the non-linearities are in the "active" (linear) regime.

Xavier initialization: $\mathbb{E}[w_{ij}] = 0, \;\; \mathrm{Var}[w_{ij}] = \frac{2}{\text{fan-in}+\text{fan\_out}}$

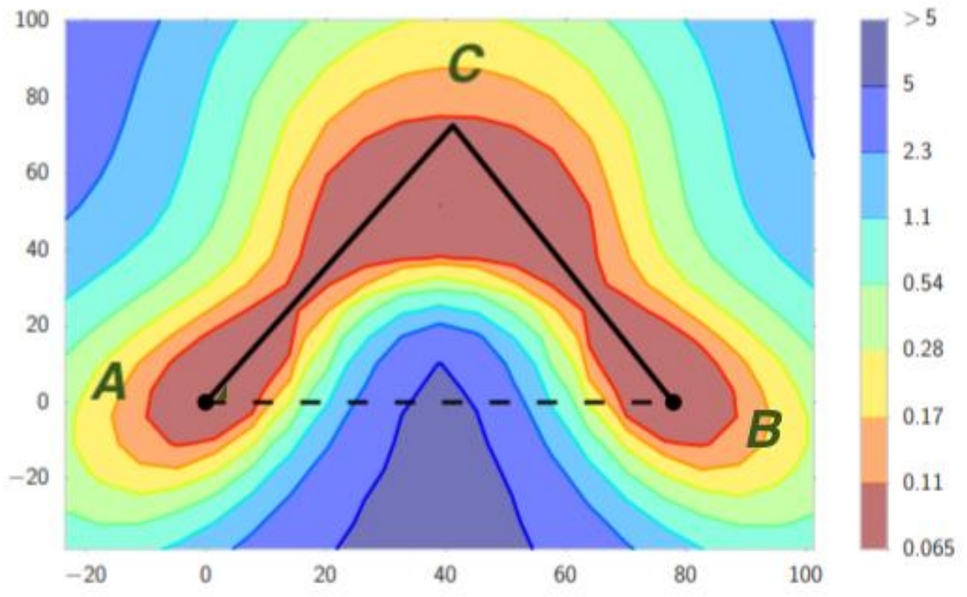
(e.g.: $N\left(0, \frac{2}{\text{fan-in}+\text{fan}_{\text{out}}}\right)$, unif$\left(\left[\frac{-\sqrt{6}}{\sqrt{\text{fan-in}+\text{fan-out}}}, \frac{\sqrt{6}}{\sqrt{\text{fan-in}+\text{fan-out}}}\right]\right)$,

Same argument as before, except when doing backprop, we take transposes of matrices. So, this keeps "backproped" activations also in the active regime.

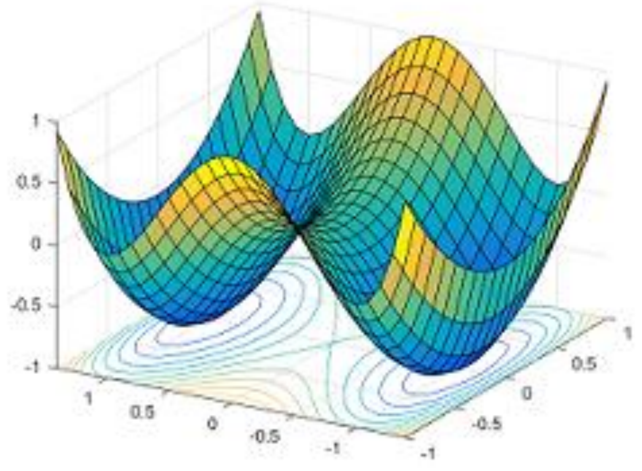


Surprises: mode connectivity

(*Freeman and Bruna, 2016*, *Garipov et al. 2018*, *Draxler et al. 2018*): Local minima can be connected by simple paths of near-same cost.



Surprising!
Remember our intuition, permutations should give rise to isolated solutions



Some explanations due to [*Kuditipudi et al '19*]: robustness of the network to "dropping out" some neurons can ensure this.

Surprises: lottery hypothesis

(*Frankle, Carbin ICLR '20, Best paper award):* Typical neural nets have **much** smaller subnetworks that "could have been trained" in isolation to give comparable results.

More precisely, the following works reliably:

1. Initialize a network and train it.
2. "Prune" superfluous structure. (e.g. smallest weights)
3. Reset unpruned weights to values in 1.
4. Repeat 2+3.

B/w 15% and 1% of the original network!

The small network won the "**initialization lottery**" – it could've been trained to a good network w/o rest of weights.

We're training networks that are way larger than in principle necessary!