Deep Learning Systems: Algorithms and Implementation

Normalization and Regularization

Fall 2022
J. Zico Kolter (this time) and Tianqi Chen
Carnegie Mellon University

Outline

Normalilzation

Regularization

Interaction of optimization, initialization, normalization, regularization

Outline

Normalilzation

Regularization

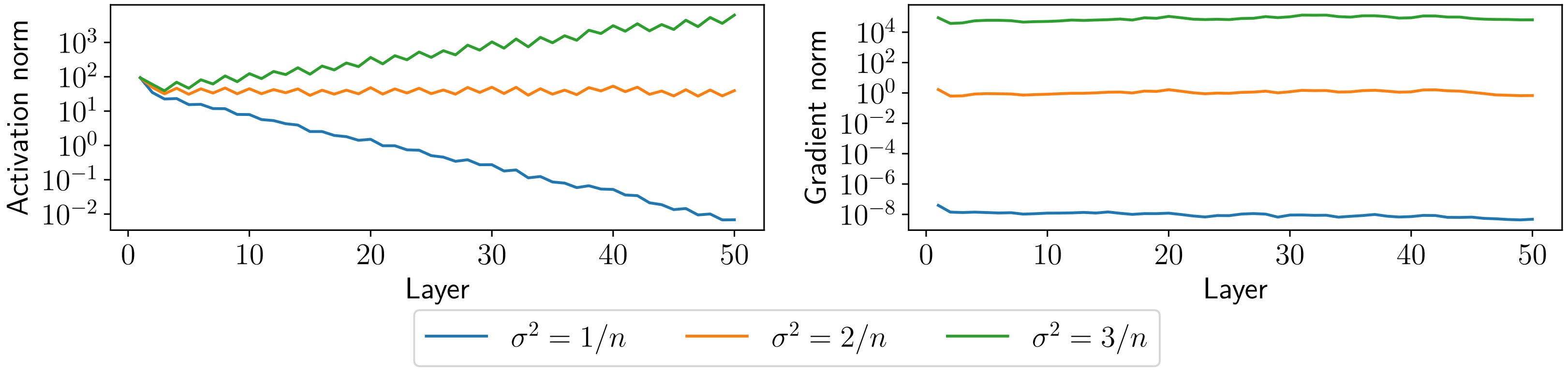
Interaction of optimization, initialization, normalization, regularization

Initialization vs. optimization

Suppose we choose $W_i \sim \mathcal{N}(0, \frac{c}{n})$, where (for a ReLU network) $c \neq 2\ldots$

Won't the the scale of the initial weights be "fixed" after a few iterations of optimization?

- No! A deep network with poorly-chosen weights will *never* train (at least with vanilla SGD)



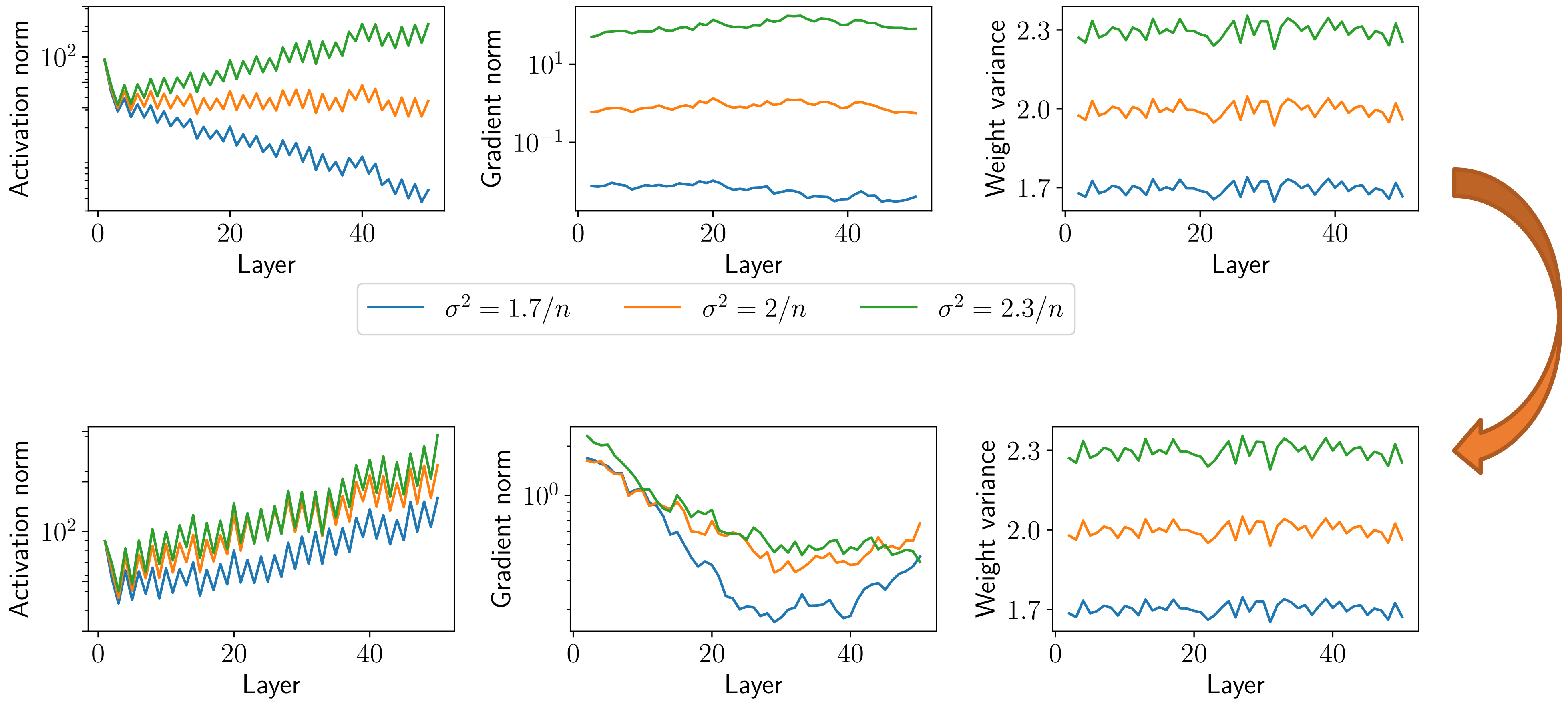
$\sigma^2 = 3/n \Longrightarrow$ NaN

$\sigma^2 = 2/n \Longrightarrow$ Works

$\sigma^2 = 1/n \Longrightarrow$ No progress

Initialization vs. optimization

The problem is even more fundamental, however: even when trained successfully, the effects/scales present at initialization *persist* throughout training



Train to 5% error on MNIST

Normalization

Initialization matters a lot for training, *and* can vary over the course of training to no longer be “consistent” across layers / networks

But remember that a “layer” in deep networks can be any computation at all…

…let’s just add layers that “fix” the normalization of the activations to be whatever we want!

Layer normalization

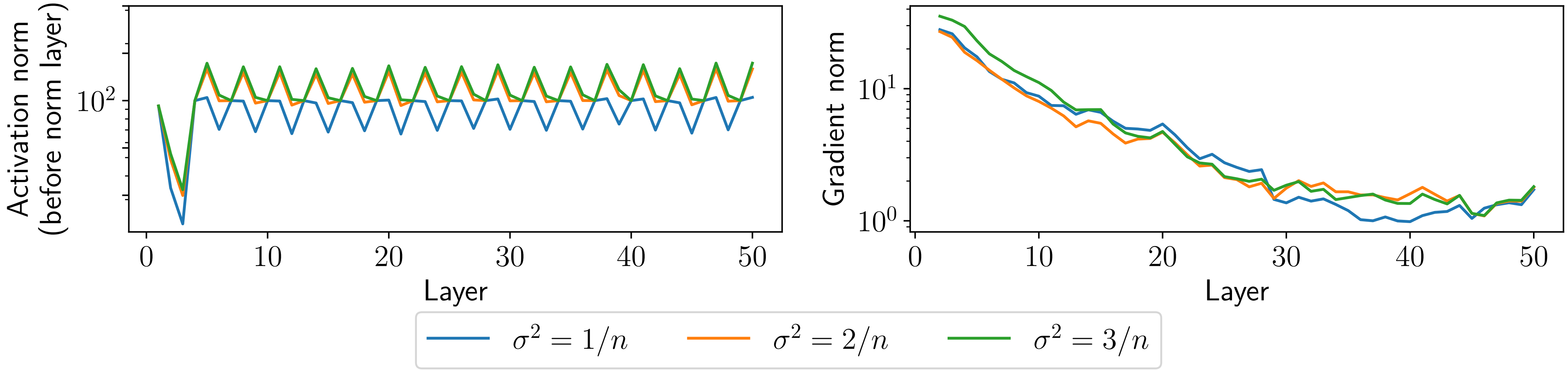
First idea: let’s normalize (mean zero and variance one) activations at each layer; this is known as *layer normalization*

$$\begin{aligned}\hat{z}_{i+1} &= \sigma_i(W_i^T z_i + b_i) \\ z_{i+1} &= \frac{\hat{z}_{i+1} - \mathbf{E}[\hat{z}_{i+1}]}{(\mathbf{Var}[\hat{z}_{i+1}] + \epsilon)^{1/2}}\end{aligned}$$

Also common to add an additional scalar weight and bias to each term (only changes representation e.g., if we put normalization *prior to* nonlinearity instead)

LayerNorm illustration

“Fixes” the problem of varying norms of layer activations (obviously)



In practice, for standard FCN, harder to train resulting networks to low loss (relative norms of examples are a useful discriminative feature)

Batch normalization

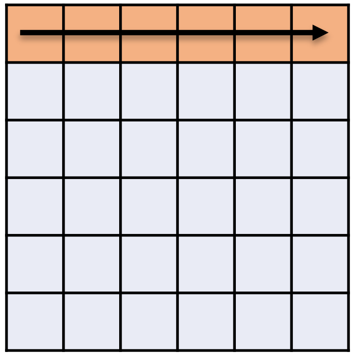
An odd idea: let’s consider the matrix form of our updates

$$\hat{Z}_{i+1} = \sigma_i(Z_i W_i + b_i^T)$$

then layer normalization is equivalent to normalizing the *rows* of this matrix

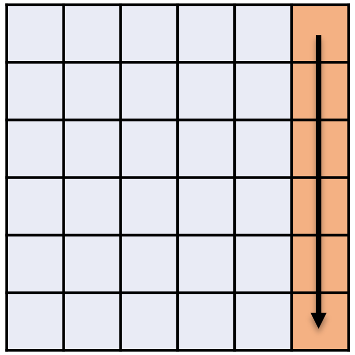
What if, instead, we normalize it’s columns? This is called *batch normalization*, as we are normalizing the activations *over the minibatch*

$$\hat{Z}_{(i+1)} =$$



Normalize (layer norm)

$$\hat{Z}_{(i+1)} =$$



Normalize (batch norm)

Minibatch dependence

One oddity to BatchNorm is that it makes the predictions for each example dependent on the entire batch

Common solution is to compute a running average of mean/variance for all features at each layer $\hat{\mu}_{i+1}, \hat{\sigma}^2_{i+1}$, and at test time normalize by these quantities

$$(z_{i+1})_j = \frac{(\hat{z}_{i+1})_j - (\hat{\mu}_{i+1})_j}{\left((\hat{\sigma}^2_{i+1})_j + \epsilon\right)^{1/2}}$$

Outline

Normalilzation

Regularization

Interaction of optimization, initialization, normalization, regularization

Regularization of deep networks

Typically deep networks (even the simple two layer network you wrote in the homework) are *overparameterized models:* they contain more parameters (weights) than the number of training examples

- This means (formally, under a few assumptions), that they are capable of fitting the training data *exactly*

In "traditional" ML/statistical thinking (with a number of big caveats), this should imply that the models will overfit the training set, and not generalize well

- … but they do generalize well to test examples
- … but not always (many larger models will often still overfit)

Regularization

Regularization is the process of “limiting the complexity of the function class” in order to ensure that networks will generalize better to new data; typically occurs in two ways in deep learning

Implicit regularization refers to the manner in which our existing algorithms (namely SGD) or architectures already limit functions considered

- E.g., we aren’t actually optimizing over “all neural networks”, we are optimizing over all neural networks considered by SGD, with a given weight initialization

Explicit regularization refers to modifications made to the network and training procedure explicitly intended to regularize the network

ℓ_2 Regularization a.k.a. weight decay

Classically, the magnitude of a model's parameters are often a reasonable proxy for complexity, so we can minimize loss while also keeping parameters small

$$\underset{W_{1:D}}{\text{minimize}} \ \frac{1}{m}\sum_{i=1}^{m} \ell\left(h_{W_{1:D}}(x^{(i)}), y^{(i)}\right) + \frac{\lambda}{2}\sum_{i=1}^{D}\|W_i\|_2^2$$

Results in the gradient descent updates:

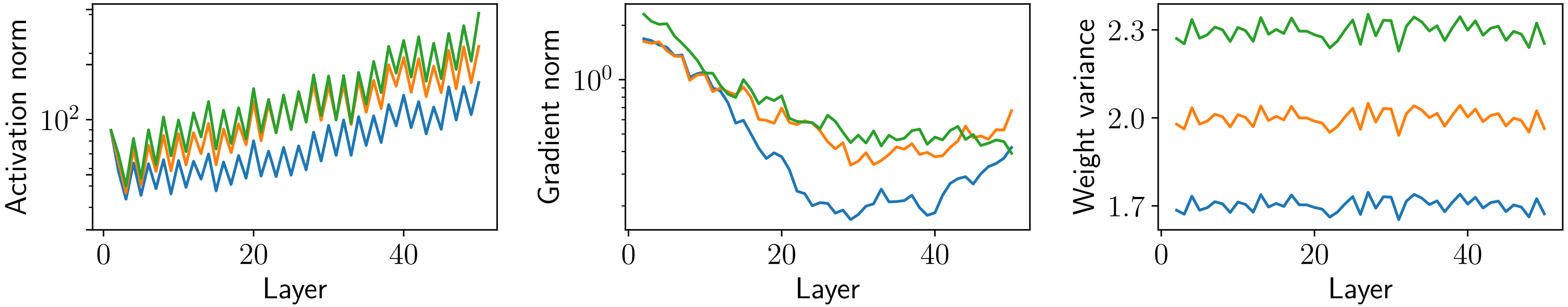
$$W_i := W_i - \alpha \nabla_{W_i} \ell(h(X), y) - \alpha\lambda W_i = (1-\alpha\lambda)W_i - \alpha\nabla_{W_i}\ell(h(X), y)$$

I.e., at each iteration we *shrink* the weights by a factor $(1-\alpha\lambda)$ before taking the gradient step

Caveats of ℓ_2 regularization

ℓ_2 regularization is exceedingly common deep learning, often just rolled into the optimization procedure as a “weight decay” term

However, recall our optimized networks with different initializations:



... Parameter magnitude may be a bad proxy for complexity in deep networks

Dropout

Another common regularization strategy: randomly set some fraction of the activations at each layer to zero

$$
\begin{aligned}
\hat{z}_{i+1} &= \sigma_i(W_i^T z_i + b_i) \\
(z_{i+1})_j &= \begin{cases} (\hat{z}_{i+1})_j/(1-p) & \text{with probability } 1-p \\ 0 & \text{with probability } p \end{cases}
\end{aligned}
$$

(Not unlike BatchNorm) seems very odd on first glance: doesn't this massively change the function being approximated?

Dropout as stochastic approximation

Dropout is frequently cast as making networks “robust” to missing activations (but we don’t apply it at test time? … and why does this regularize network?)

Instructive to consider Dropout as bringing a similar stochastic approximation as SGD to the setting of individual activations

$$\frac{1}{m}\sum_{i=1}^{m}\ell(h(x^{(i)}), y^{(i)}) \implies \frac{1}{|B|}\sum_{i\in B}\ell(h(x^{(i)}), y^{(i)})$$

$$z_{i+1} = \sigma_i\left(\sum_{j=1}^{n} W_{:,j}(z_i)_j\right) \implies z_{i+1} = \sigma_i\left(\frac{n}{|\mathcal{P}|}\sum_{j\in\mathcal{P}}^{n} W_{:,j}(z_i)_j\right)$$

Outline

Normalilzation

Regularization

Interaction of optimization, initialization, normalization, regularization

Many solutions … many more questions

Many design choices meant to ease optimization ability of deep networks

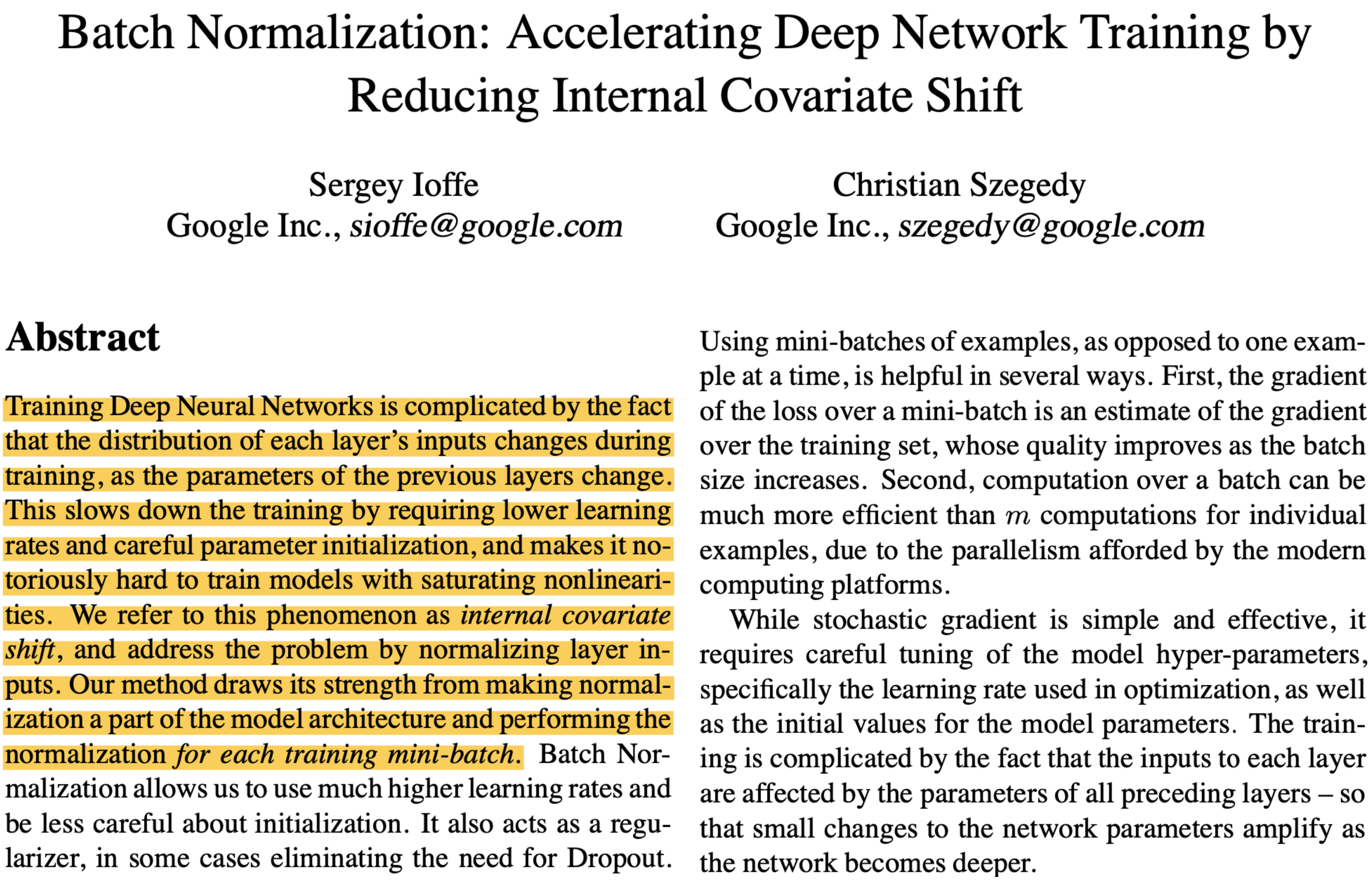
- Choice of optimizer learning rate / momentum
- Choice of weight initialization
- Normalization layer
- Reguarlization

These factors all (of course) interact with each other

And these don't even include many other "tricks" we'll cover in later lectures: residual connections, learning rate schedules, others I'm likely forgetting

…you would be forgiven for feeling like the practice of deep learning is all about flailing around randomly with lots of GPUs

BatchNorm: An illustrative example



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization *for each training mini-batch*. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout.

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

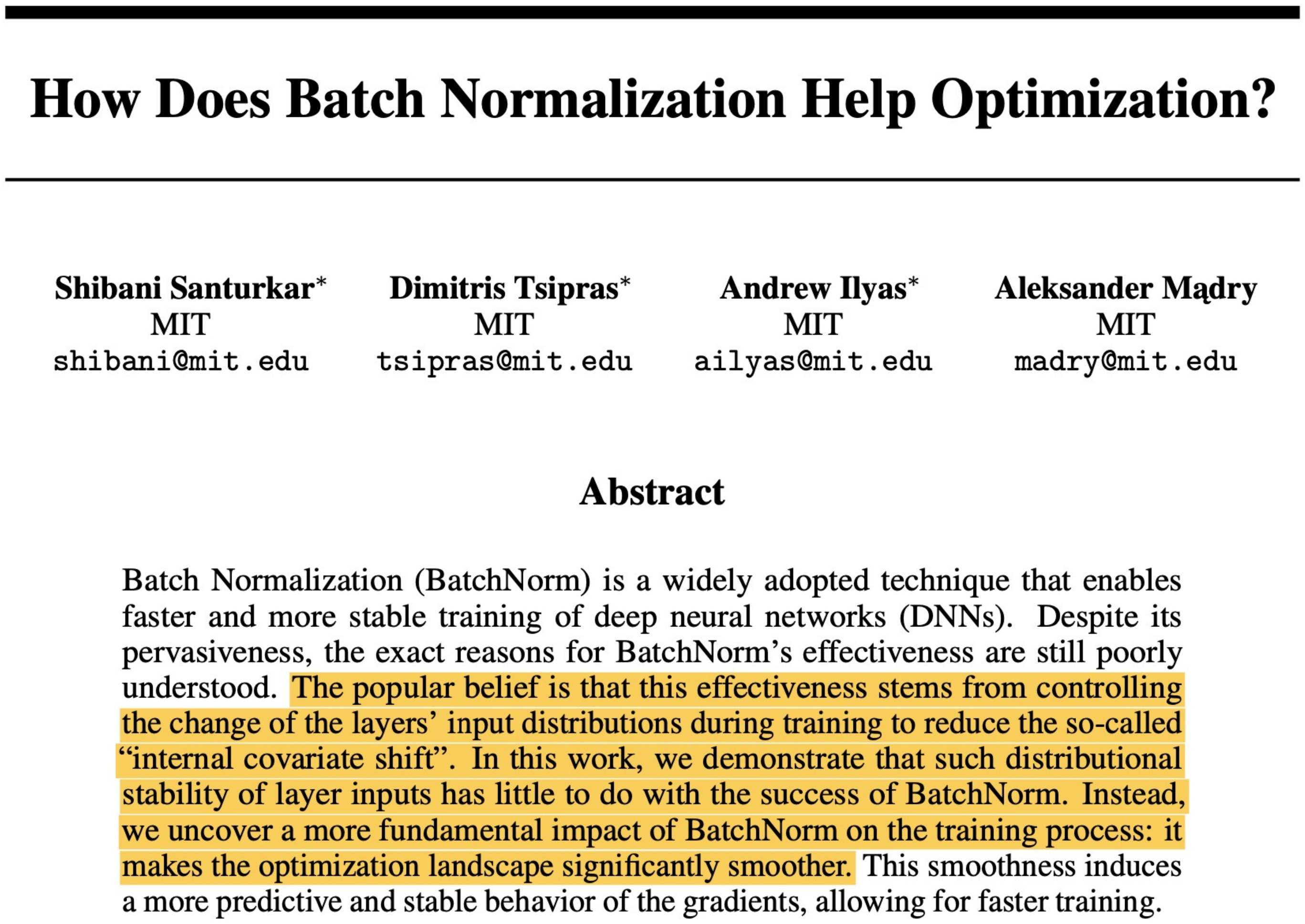
While stochastic gradient is simple and effective, it requires careful tuning of the model hyper-parameters, specifically the learning rate used in optimization, as well as the initial values for the model parameters. The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters amplify as the network becomes deeper.

BatchNorm: An illustrative example

“Here is what we know about batch norm as a field. It works because it reduces internal covariant shift. Wouldn't you like to know why reducing internal covariant shift speeds up gradient descent? Wouldn't you like to see a theorem or an experiment? Wouldn't you like to know, wouldn't you like to see evidence that batch norm reduces internal covariant shift? Wouldn't you like to know what internal covariant shift is? Wouldn't you like to see a definition of it?”

- Ali Rahimi (NeurIPS 2017 Test of Time Talk)

BatchNorm: An illustrative example



How Does Batch Normalization Help Optimization?

Shibani Santurkar*
MIT
shibani@mit.edu

Dimitris Tsipras*
MIT
tsipras@mit.edu

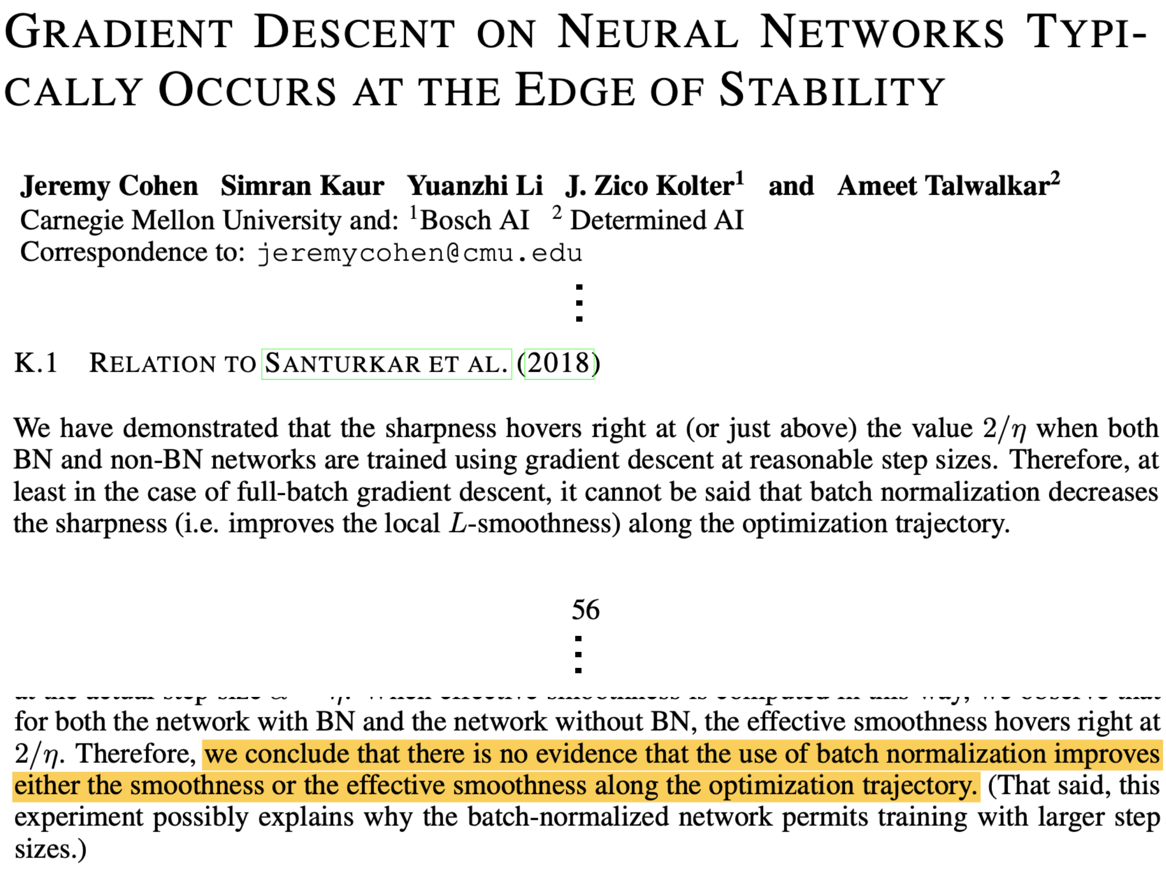
Andrew Ilyas*
MIT
ailyas@mit.edu

Aleksander Mądry
MIT
madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

BatchNorm: An illustrative example



GRADIENT DESCENT ON NEURAL NETWORKS TYPICALLY OCCURS AT THE EDGE OF STABILITY

Jeremy Cohen Simran Kaur Yuanzhi Li J. Zico Kolter[1] and Ameet Talwalkar[2]
Carnegie Mellon University and: [1]Bosch AI [2] Determined AI
Correspondence to: `jeremycohen@cmu.edu`

⋮

K.1 RELATION TO SANTURKAR ET AL. (2018)

We have demonstrated that the sharpness hovers right at (or just above) the value $2/\eta$ when both BN and non-BN networks are trained using gradient descent at reasonable step sizes. Therefore, at least in the case of full-batch gradient descent, it cannot be said that batch normalization decreases the sharpness (i.e. improves the local L-smoothness) along the optimization trajectory.

56

⋮

at the actual step size $\alpha = \eta$. When effective smoothness is computed in this way, we observe that for both the network with BN and the network without BN, the effective smoothness hovers right at $2/\eta$. Therefore, we conclude that there is no evidence that the use of batch normalization improves either the smoothness or the effective smoothness along the optimization trajectory. (That said, this experiment possibly explains why the batch-normalized network permits training with larger step sizes.)

BatchNorm: Other benefits?

TENT: FULLY TEST-TIME ADAPTATION BY ENTROPY MINIMIZATION

Dequan Wang[1]*, Evan Shelhamer[2]*†, Shaoteng Liu[1], Bruno Olshausen[1], Trevor Darrell[1]
dqwang@cs.berkeley.edu, shelhamer@google.com
UC Berkeley[1] Adobe Research[2]

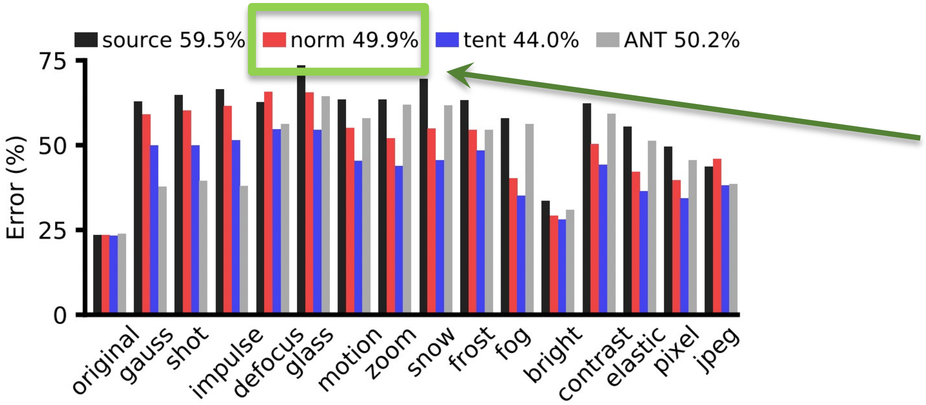


Figure 5: Corruption benchmark on ImageNet-C: error for each type averaged over severity levels. Tent improves on the prior state-of-the-art, adversarial noise training (Rusak et al., 2020), by fully test-time adaptation *without altering training*.

Running batch norm at test time (what we told you not to do, because it induces minibatch dependence), improves model performance on out-of-distribution data

The ultimate takeaway message

I don’t want to give the impression that deep learning is all about random hacks: there have been a lot of excellent scientific experimentation with all the above

But it is true that we don’t have a complete picture of how all the different empirical tricks people use really work and interact

The “good” news is that in many cases, it seems to be possible to get similarly good results with wildly different architectural and methodological choices