Data 188: Introduction to Deep Learning

Normalization and Regularization

Speaker: Eric Kim
Lecture 09 (Week 05)
2026-02-17, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Rev01: 2026-02-17 5:33 PM PST

Announcements

- HW1 due this Thursday!
  - Warning: this homework is substantially more work than HW0. Start early!
- Weekly course surveys
  - Thanks for filling these out!
  - "Course Survey (Week 05) (optional, extra credit)"

Midterm is coming up!

For more info, see course website.

Exams

There will be one midterm, and a final exam. Both exams will be on campus, in-person, paper and pencil, and proctored by course staff. Electronic devices are not allowed, including: calculators, smart phones, laptops.

For the midterm, you are allowed one page (double sided) of notes. For the final exam, you are allowed two pages (double sided) of notes.

Exams will cover all material covered in: lectures, discussion section, and assignments.

Name	Date	Time (PST)	Location
Midterm	Tuesday March 10th 2026	7:00 PM - ?	Genetics and Plant Biology 0100
Midterm (DSP)	Tuesday March 10th 2026	7:00 PM - ?	Anthropology & Art Practice 115
Alternate Midterm	Wednesday March 11th 2026	7:00 PM - ?	Genetics and Plant Biology 0100
Alternate Midterm (DSP)	Wednesday March 11th 2026	7:00 PM - ?	Anthropology & Art Practice 115
Final	Thursday May 14th 2026	11:30 AM - ?	?

Midterm scope

- We'll release more details on Ed, but in short:

- Midterm covers all material up to and including Week 07
    - Up to and including: Lecture 14 (Thursday March 5th, 2026)
    - Up to and including: Discussion 06 (Wednesday March 4th, 2026)

- If any content is not in scope for the midterm, we'll let you know on Ed.

Outline

Normalization

Regularization

Interaction of optimization, initialization, normalization, regularization

Outline

Normalization

Regularization

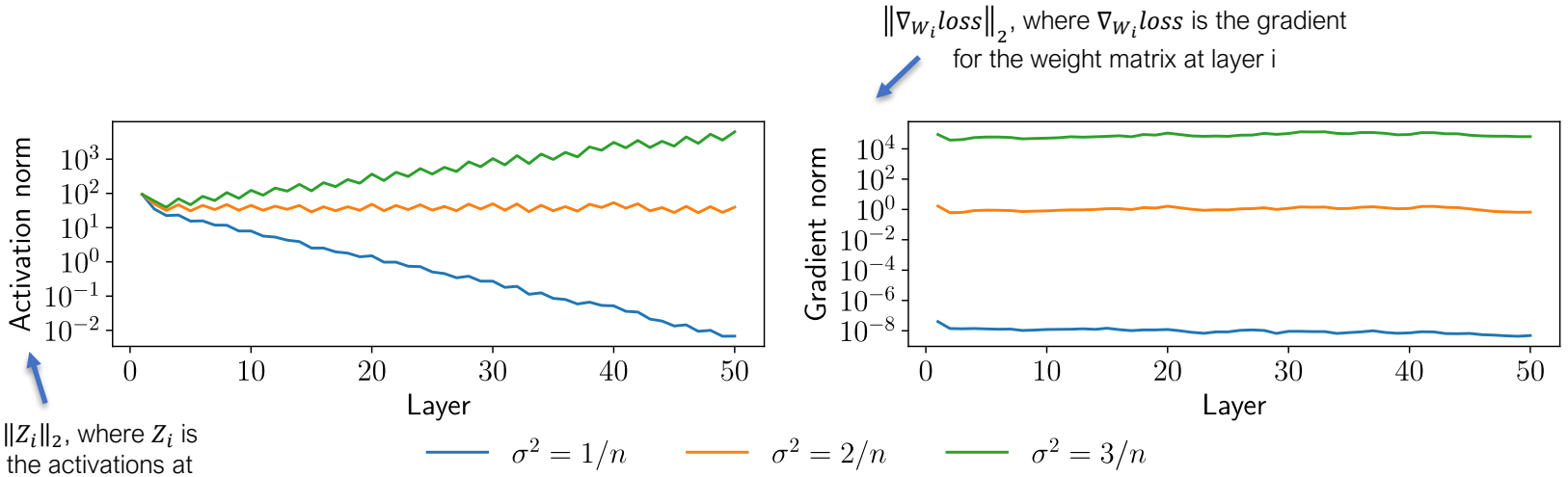
Interaction of optimization, initialization, normalization, regularization

Initialization vs. optimization (1/2)

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

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

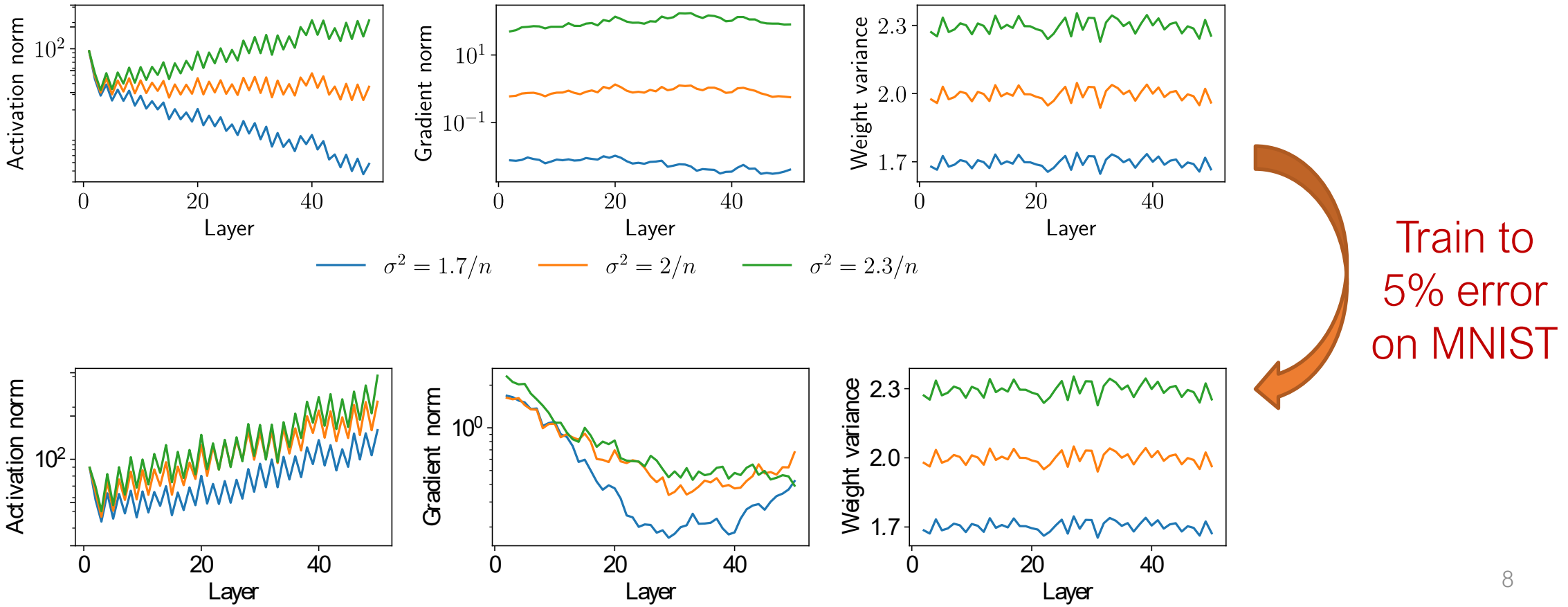
$\sigma^2 = 2/n \Longrightarrow$ Works

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

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Initialization vs. optimization (2/2)

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



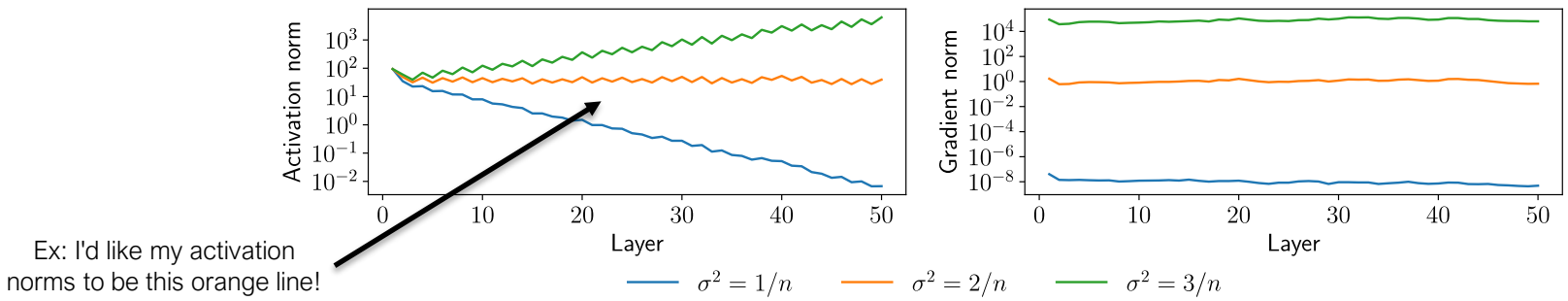
Normalization

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 add layers that “fix” the activations to be whatever we want!

Example: suppose wanted to add a layer that ensures that activations are always at a "healthy" scale. What would that look like?



Idea: activation normalization

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

$\hat{z}_{i+1}$ has shape=[batchsize, n], where n is the feature dim

$$\hat{z}_{i+1} = \sigma_i(z_i W_i + b_i)$$

$$z_{i+1} = \frac{\hat{z}_{i+1} - \mathbf{E}[\hat{z}_{i+1}]}{\sqrt{\mathbf{Var}[\hat{z}_{i+1}] + \epsilon}}$$

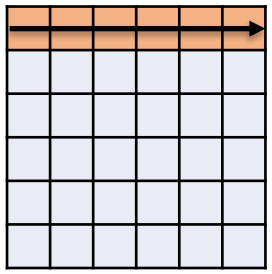
Mean, Var is calculated across feature dim

$$\mathbf{E}[x] = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\boldsymbol{Var}[x] = \frac{1}{n}\sum_{i=1}^{n}(x_i - \boldsymbol{E}[x])^2$$

Mean, var is calculated across the feature dim, and for each batch sample independently (eg each input sample uses different mean/var stats).

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



Normalize (layer norm)

Layer Norm: definition

Finally, it's common to add an additional learnable weight and bias:

$$Z = \frac{X - \mathbf{E}[X]}{\sqrt{\mathbf{Var}[X] + \epsilon}} \circ \gamma + \beta$$

$$\mathbf{E}[x] = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\boldsymbol{Var}[x] = \frac{1}{n}\sum_{i=1}^{n} (x_i - \boldsymbol{E}[x])^2$$

Mean, Var is calculated across feature dim

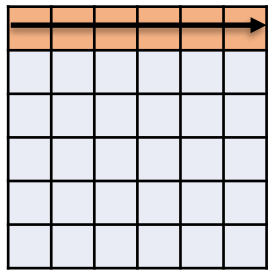
x has shape=[batchsize, n], where n is the feature dim

z has shape=[batchsize, n]

$\circ$ denotes elementwise multiplication

γ, β are the learnable affine params with shape=[1, n]

$$Z =$$



Normalize (layer norm)

$$\mathbf{E}[X] = \begin{bmatrix} E[X[0,:]] \\ E[X[1,:]] \\ \vdots \\ E[X[batchsize-1,:]] \end{bmatrix}$$

$$\mathbf{Var}[X] = \begin{bmatrix} Var[X[0,:]] \\ Var[X[1,:]] \\ \vdots \\ Var[X[batchsize-1,:]] \end{bmatrix}$$

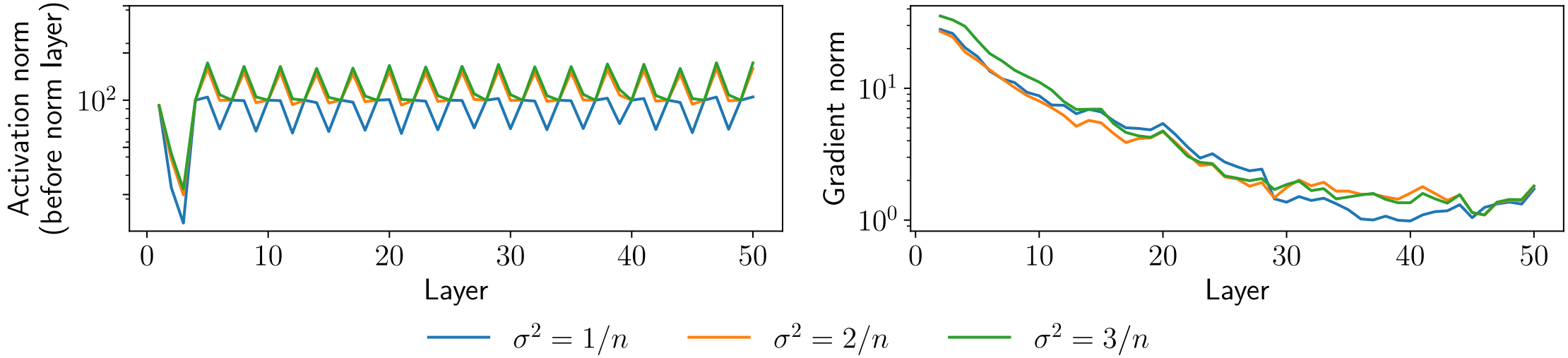
"Batched" mean/var calculates separate mean/var for each batch sample

Paper (2016): "Layer Normalization"

Pytorch: torch.nn.LayerNorm

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)

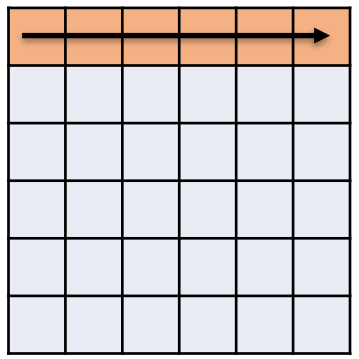
Batch Normalization

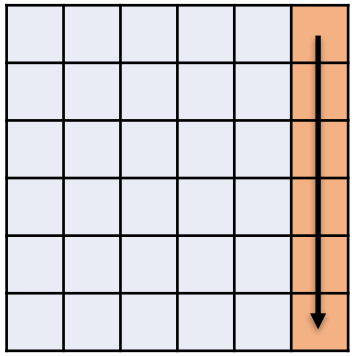
An odd idea: let's consider the matrix form of our updates

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

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

What if, instead, we normalize its 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)

BatchNorm: definition

Like LayerNorm, it's common to also add a learnable scale+shift (affine):

$$Z = \frac{X - \mathbf{E}[X]}{\sqrt{\mathbf{Var}[X] + \epsilon}} \circ \gamma + \beta$$

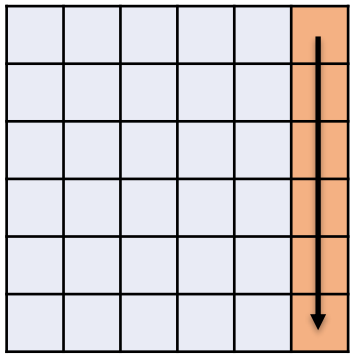
$$\mathbf{E}[x] = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\boldsymbol{Var}[x] = \frac{1}{n}\sum_{i=1}^{n} (x_i - \boldsymbol{E}[x])^2$$

Mean, Var is calculated across **batch dim**

x has shape=[batchsize, n], where n is the feature dim
z has shape=[batchsize, n]
$\circ$ denotes elementwise multiplication
γ, β are the learnable affine params with shape=[1, n]

$$Z =$$



Normalize (batch norm)

$$\mathbf{E}[X] = \begin{bmatrix} E[X[:,0]] \\ E[X[:,1]] \\ \vdots \\ E[X[:,n-1]] \end{bmatrix}$$

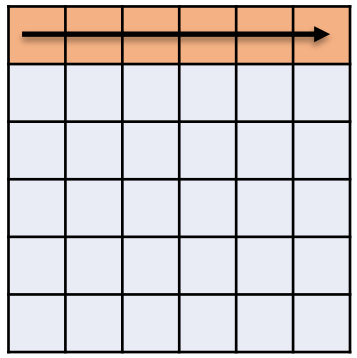
"Batched" mean/var calculates separate mean/var for each **input dim**

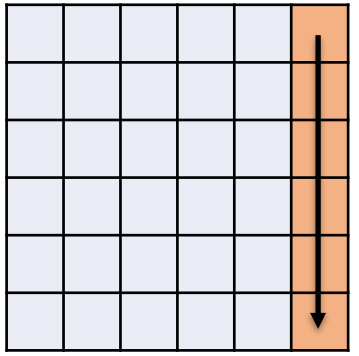
$$\mathbf{Var}[X] = \begin{bmatrix} Var[X[:,0]] \\ Var[X[:,1]] \\ \vdots \\ Var[X[:,n-1]] \end{bmatrix}$$

BatchNorm: intuition

Intuition: standardize each input dim (individually!) to have mean=0, var=1, using the current batch to calculate the population statistics (eg normalize each column).

With a big enough sample size (eg batchsize), this "sounds reasonable".

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

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

BatchNorm: test time

One oddity to BatchNorm is that it makes the predictions for each example dependent on the entire batch (even at test time!)

Ex: would be very strange to use a model with BatchNorm where batchsize=1 is impossible/"unsupported".

Solution: during training, keep track of (on the side) a running average of mean/variance (via exponential moving average, similar to Momentum/Adam).

At **test time:** use the precomputed ("frozen") mean/variance running averages (rather than batch-computed mean/variances):

```
def batch_norm_fwd_train(
        inp: Tensor,
        running_mean: Tensor,
        running_var: Tensor,
        running_weight: float = 0.1,
        eps: float = 1e-5
):
    # inp.shape=[batchsize, dim_feat]
    # running_mean, running_var shape=[dim_feat]
    # calculate batch mean/var stats
    feat_mean = mean(inp, axis=0)  # shape=[dim_feat]
    feat_var = var(inp, axis=0)  # shape=[dim_feat]
    out = (inp - feat_mean) / sqrt(feat_var + eps)

    # update running avg of mean/var (weighted by running_weight
    running_mean.data = (
        running_weight * running_mean + (1 - running_weight) * feat_mean
    )
    running_var.data = (
        running_weight * running_var + (1 - running_weight) * feat_var
    )
    return out

def batch_norm_fwd_test(
        inp: Tensor,
        running_mean: Tensor,
        running_var: Tensor,
        eps: float = 1e-5
):
    # inp.shape=[batchsize, dim_feat]
    # running_mean, running_var shape=[dim_feat]
    return (inp - running_mean) / sqrt(running_var + eps)
```

BatchNorm: train vs test

Training time: calculate and update mean/var running average. Use minibatch mean/var stats for doing training normalizing.

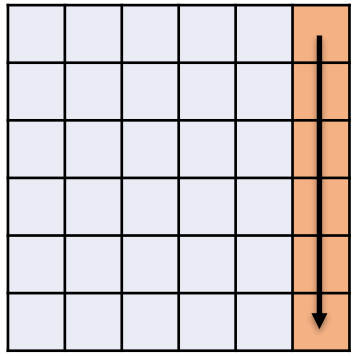
Test time: freeze mean/var running average. Use mean/var running average for doing test-time normalizing.

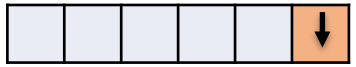
Lots of stories where people forgot to freeze the batch norm running mean/var average. "Why is my model getting worse after serving it for a week??"

Verdict: BatchNorm is peculiar and sort of annoying to use. But, in practice tends to work reasonably well (perhaps frustratingly so? haha)

BatchNorm: sensitive to batchsize

What if you're training a model and your batchsize is too small? As an extreme example, batchsize=1?

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

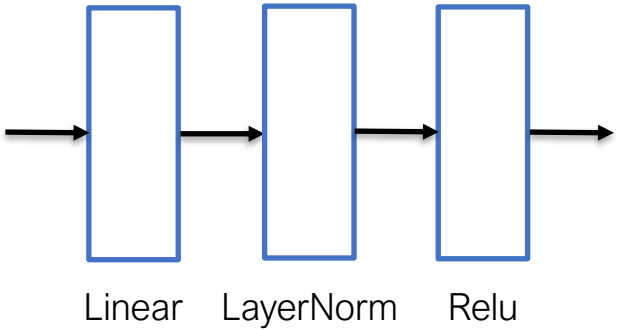
$\hat{Z}_{(i+1)} =$  Normalize (batch norm)???

Naive batchnorm fails (can't calculate the variance of a single sample). There are workarounds, but it's very strange that our batchnorm has a strict requirement on batchsize.

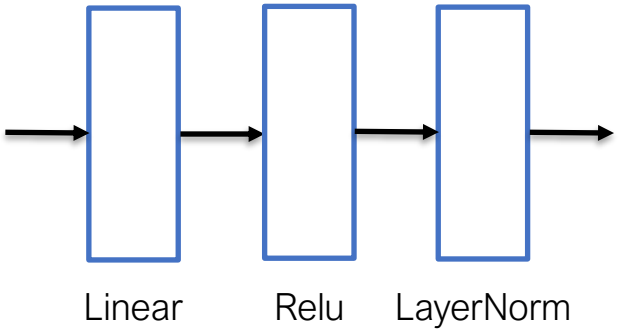
Activation normalization

In practice, we typically add the activation normalization **before** the nonlinearity:

Ex: `BasicBlock` in torchvision.models.resnet does conv -> batchnnorm -> relu

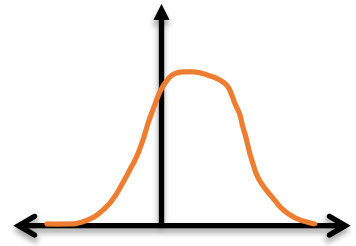


...but this will also "work":

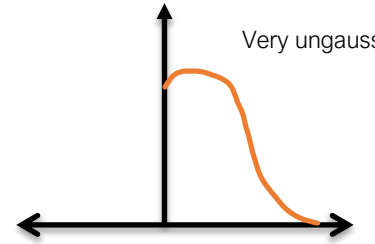


Question: why might adding the activation normalization **before** the nonlinearity work better than **after**?

Answer: to be honest there's no "correct" answer. But, here's one answer: the goal of LayerNorm is to make activation distributions look "nice" (eg gaussian-like with mean=0, var=1).
The outputs of Linear are (likely) more gaussian-like than Relu outputs. So, it makes sense to try to apply LayerNorm to the Linear outputs rather than the Relu outputs.



Linear output value distribution



Linear->Relu output value distribution

(optional) For an interesting discussion about whether to put the norm fn before or after the nonlinearity:
https://forums.fast.ai/t/where-should-i-place-the-batch-normalization-layer-s/56825/6
https://discuss.pytorch.org/t/batch-normalization-of-linear-layers/20989/22?u=shirui-japina

"actual" answer: try out both, see which works better!

Activation normalization takeaways

Activation normalization is a Good Thing, and is widely used.

Popular techniques: LayerNorm, BatchNorm.

Normalizing the intermediate activations leads to a "better behaved" optimization, which is more robust to things like: parameter initialization, learning rate, hyperparameters, the time of day, etc.

This is the (one) "art" of deep learning: a series of techniques and best practices to make it easy to train effective deep learning models on your task.

Outline

Normalization

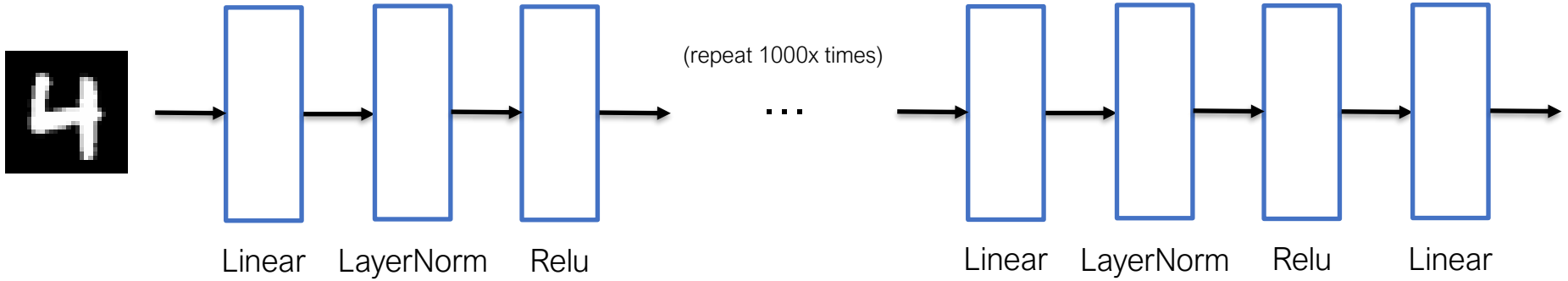
Regularization

Interaction of optimization, initialization, normalization, regularization

Model design: keep stacking layers?

So far in this class, we've trained a single-layer NN, a two-layer NN...

What's stopping us from training a 50-layer NN? A million?

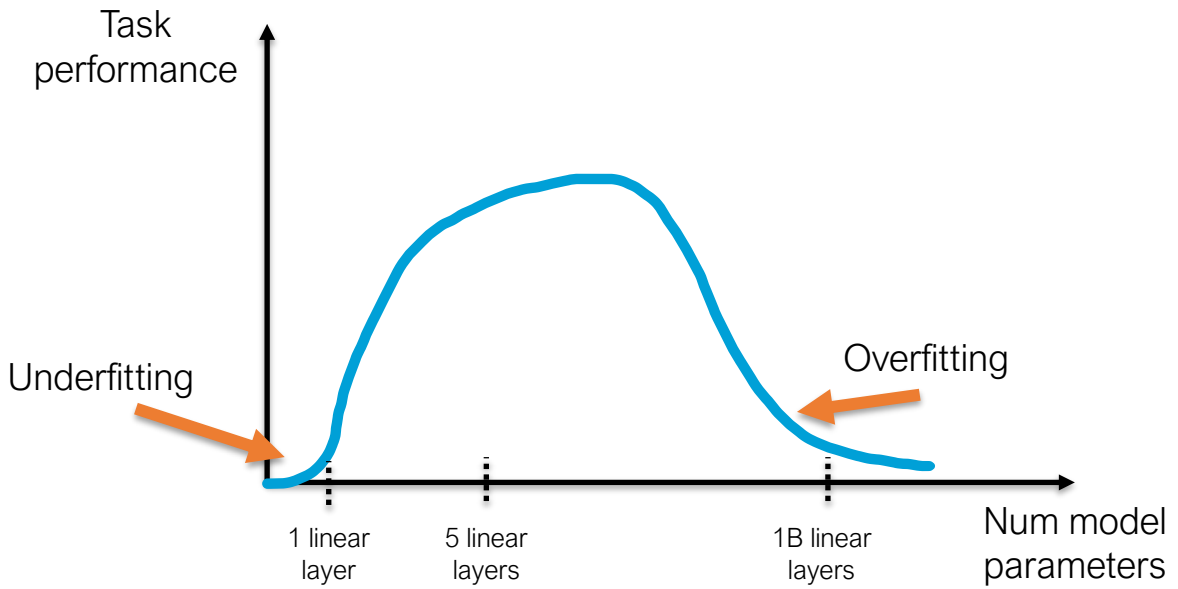


Model capacity

Intuition: as we add more layers to our NN, we should initially see an **increase** in task performance.

Adding more layers (adding more model parameters) increases our model's complexity, aka "**model capacity**". ("capacity to learn")

But! At a certain point, we expect task performance to **drop**. "Overfitting"



Model capacity: curve fitting

Example: suppose we have a regression dataset of points in $\mathbb{R}^2$. Let's restrict our model to the space of polynomial functions:

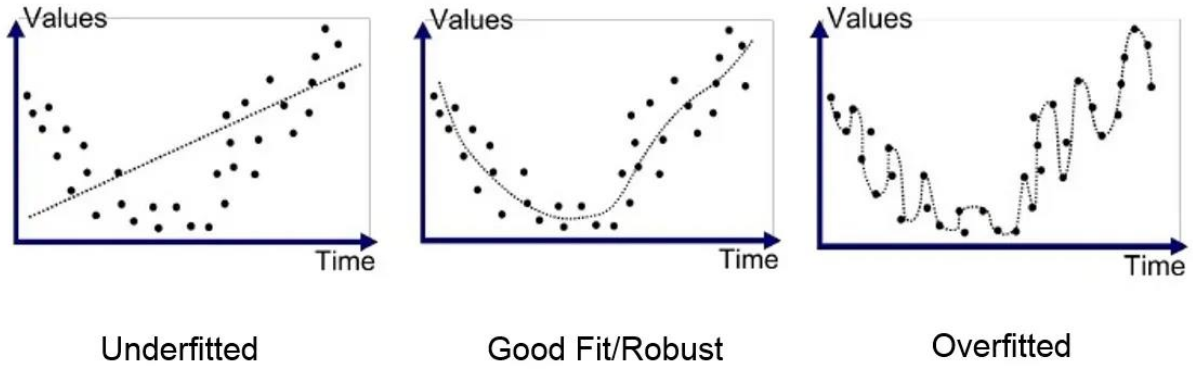
- $f(x) = a_0x + a_1x^2 + a_2x^3 + \ldots$

A "high capacity" model can fit to many kinds of phenomenon

- Ex: high-degree polynomials.

A "low capacity" model can represent only a limited amount of phenomenon

- Ex: degree-1 polynomial (aka straight line)



Underfitted | Good Fit/Robust | Overfitted

Low degree polynomial (eg linear function)

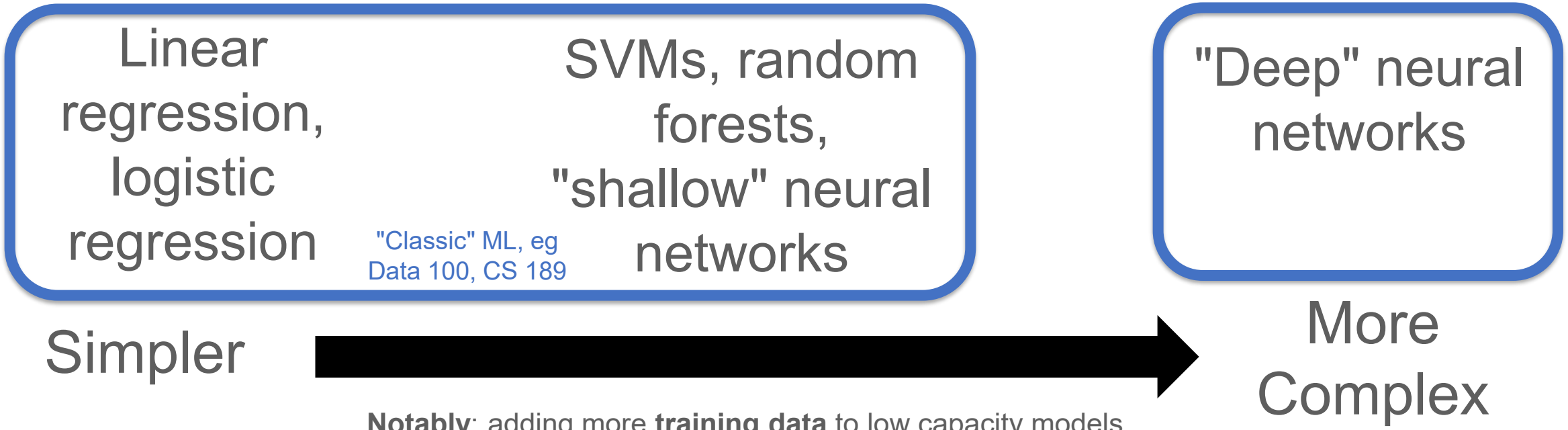
Very high degree polynomial

Simpler → More Complex

Source: https://medium.com/swlh/machine-learning-how-to-prevent-overfitting-fdf759cc00a9

Model capacity: model classes

Similar to the n-degree polynomial regression example, we can make similar statements when comparing different ML model approaches



Notably: adding more **training data** to low capacity models stops helping after a certain point.
But, for high capacity models one can keep adding more training data, and performance will continue to get better.

(a simplified caricature, but useful for intuitions)

Model capacity: deep learning

…and even within DNNs, there's stark differences in model capacity.

While not 100% precise, a model's "parameter count" is an often-used measure for a model's capacity.

Ex: ResNet50 (a successful ConvNet from 2015) has **25.6M** parameters ([link])

GPT-3 (OpenAI, 2020) has **20B – 175B** parameters ([link] [link])

GPT4+ models rumored to be >1 trillion params!

Two layer NNs

Convolutional networks, lightweight text encoders, ...

Large language models (LLMs), generative AI

Simpler

More Complex

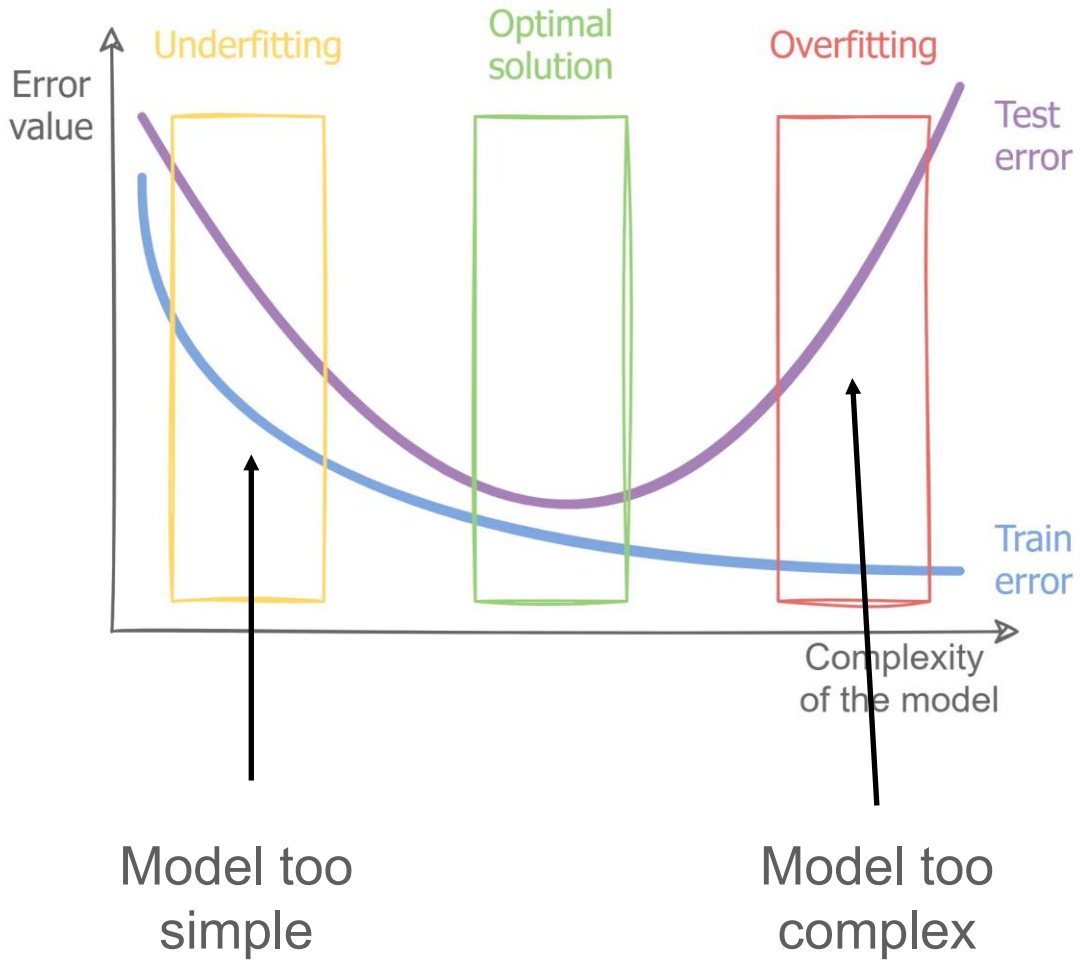
(a simplified caricature, but useful for intuitions)

Model capacity and under/overfitting

One useful knob for controlling underfitting vs overfitting is: model complexity (aka "capacity")

For a fixed training dataset, we have the following rule of thumb:

- **If you're underfitting**: often means that the model is too simple (ex: trying to fit a linear model to a nonlinear data distribution)
- **If you're overfitting**: often means that your model is too complex and is fitting to the noise in the data.



Source: https://towardsdatascience.com/overfitting-and-underfitting-principles-ea8964d9c45c

Regularization: linear regression

In machine-learning: regularization is an effective technique to avoid overfitting.

Idea: constrain your model (in some way) to avoid overfitting.

Classic example: L1, L2 regularization, eg for linear regression:

$$\min_{w}(w^T x - y)^2 + \lambda_1 \|w\|_2^2 + \lambda_2 \|w\|_1$$

Reconstruction error

L2 regularization

"Ridge Regression"

L1 regularization

"Lasso Regression"

x shape: [n, 1]
w shape: [n, 1]
y is scalar

λ_1, λ_2 is scalar weight controlling regularization importance.

Soft constraint: don't allow model parameters w to grow too large in magnitude.
Balance task performance (reconstruction error) vs regularization terms via λ_1, λ_2

Tip: L1 reg encourages model sparsity ("feature selection")

Regularization of deep networks

Typically deep networks are *overparameterized models:* they contain more parameters (weights) than the number of training examples

The ML conventional wisdom: a model with more parameters than training dataset examples can fit the training data *exactly* (eg egregious overfitting)

In deep learning, it's especially important to avoid overfitting, eg via regularization

Let's explore some popular methods of regularizing deep learning models!

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

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

Here, $\|W\|_F$ means "Frobenius norm", aka sum of squared entries. Not to be confused with other matrix norms like spectral norm $\|W\|_2$

Results in the gradient descent updates:

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

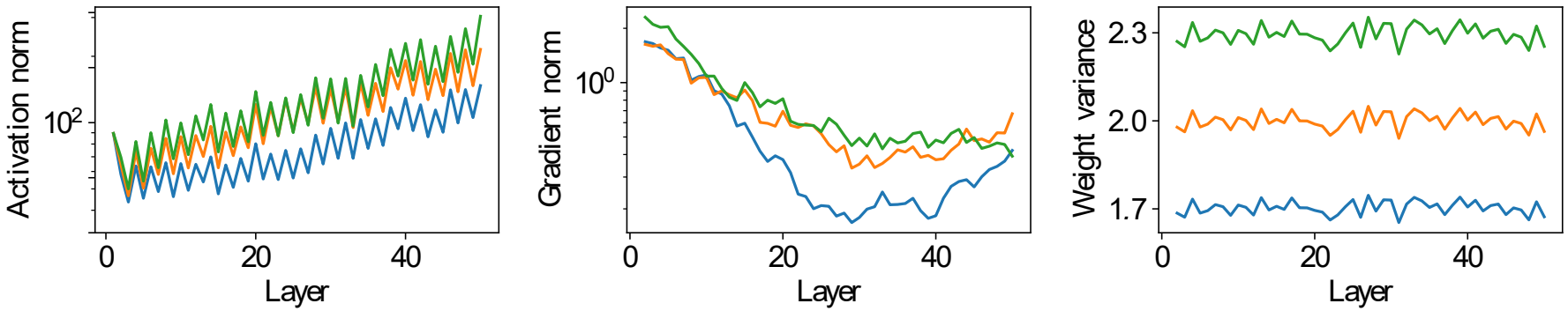
I.e., at each iteration we *shrink* the weights by a factor $(1 - \alpha\lambda)$ before taking the gradient step

α is learning rate

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. Especially when you consider activation normalization etc.

Solution: validate empirically!

Dropout

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

$$y_i = \begin{cases} \frac{x_i}{1-p} & \text{with probability } 1-p \\ 0 & \text{with probability } p \end{cases}$$

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

Dropout: train vs test

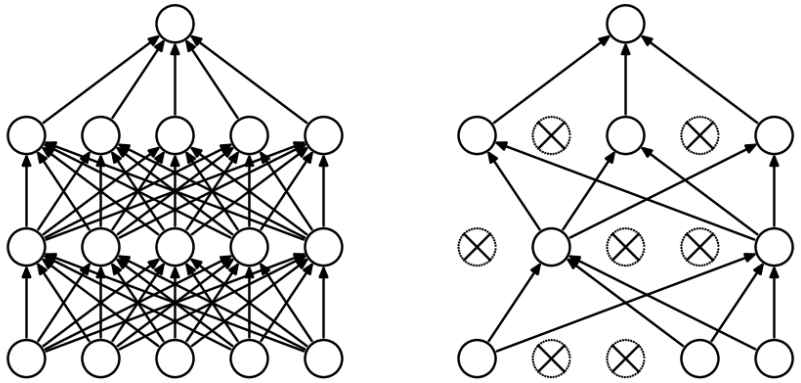
Training time: randomly drop input value with probability p (set to 0). If we don't drop the value, scale it by $\frac{1}{1-p}$

Test time: don't drop any input values. Return value unmodified ("no-op")

Tip: if we didn't scale by $\frac{1}{1-p}$ during training, then at test time, the total magnitude of the "no-op" output would be larger than during training because we no longer drop input values. This likely will make results worse, as we've changed the feature distribution from training to testing.

Dropout: intuition

One interpretation: dropout avoids overfitting because it forces the model to learn strong, generalizable representations that are robust to noise.



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

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Paper (2014): "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

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?)

Idea: 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

Normalization

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
- Regularization

These factors all 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: Ali Rahimi's thoughts

“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: One investigation...

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: Another investigation...

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.

⋮

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]

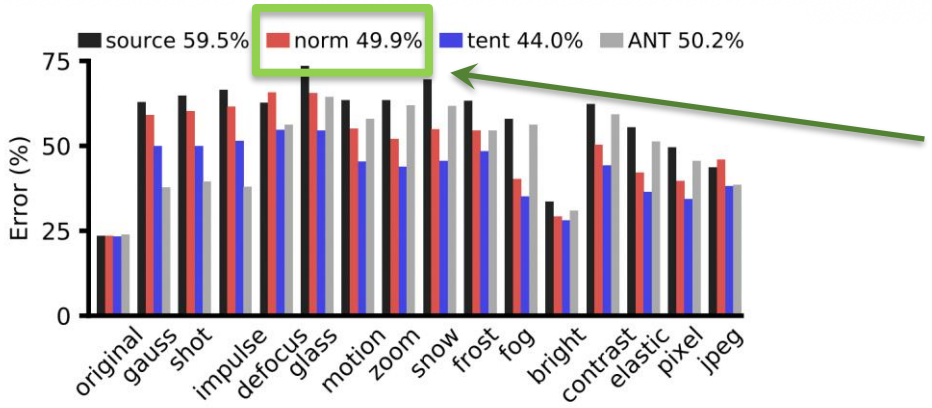


Figure 5: Corruption benchmark on ImageNet-C: error for each type averaged over severity levels. Tent improves on the prior state-of-the-art, adversarial noise training (Rusak et al., 2020), by fully test-time adaptation *without altering training*.

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

My advice: start off with well-studied, popular model architectures that work well for a variety of problems. Then, iterate from there as a starting point!