Batch normalisation

How I Learned to Stop Worrying and Love the BN

WHAT is Batch Norm

Intuition of BN calculation

- Take a batch, and normalise its activations

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

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

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

x_i = activations of a neuron on sample *i*

m = batch size***

*** caveats ahead

The forgotten:

- BN does not JUST subtract mean and divide by STD. It also re-scales the normalised thing by a set of **trainable parameters, a pair for each activation (neuron). THIS IS DEFAULT IN MOST FRAMEWORKS!**

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

Why is this done? Shouldn't be necessary in most cases.
Note: original paper was NOT most cases

WHEN/WHERE is Batch Norm

When goes BN?

- Dunno?

madebyollin · 5 yr. ago · *edited 5 yr. ago*

From Francois Chollet (Keras author currently at Google):

> I can guarantee that recent code written by Christian [Szegedy, from the BN paper] applies relu before BN. It is still occasionally a topic of debate, though.

In my opinion, BN after ReLU makes much more sense - the weight matrix W then looks at mean-centered data. This is false because of training parameters 0?

```
# CONV => RELU => POOL
model.add(Conv2D(32, (3, 3), padding="same",input_shap
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.25))
```

From Ioffe and Szegedy (2015)'s point of view, only use BN in the network structure. Li et al. (2018) give the statistical and experimental analyses, that there is a variance shift when the practitioners use Dropout before BN. Thus, Li et al. (2018) recommend applying Dropout after all BN layers.

From Ioffe and Szegedy (2015)'s point of view, BN is located **inside/before** the activation function. However, Chen et al. (2019) use an IC layer which combines dropout and BN, and Chen et al. (2019) recommends use BN after ReLU.

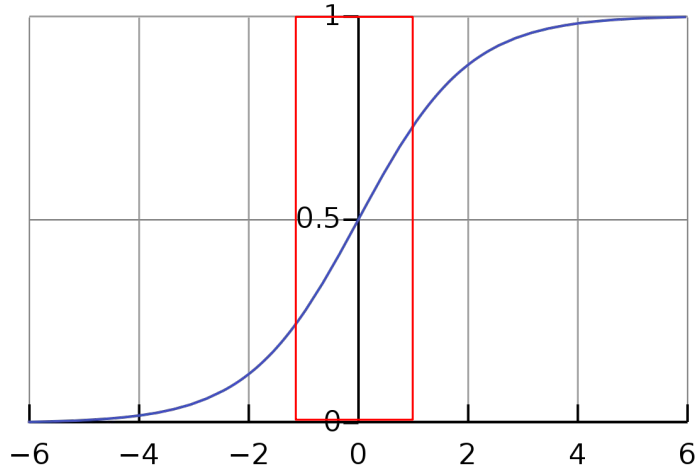
```
self.conv1 = conv3x3(inplanes, planes, stride)
self.bn1 = norm_layer(planes)
self.relu = nn.ReLU(inplace=True)
self.conv2 = conv3x3(planes, planes)
```

When goes BN: Original paper

- BN paper was defined to work on **FULLY CONNECTED, SIGMOID-ACTIVATION networks**
- The layer was located **before** the activation function **for a reason:** The trainable parameters were introduced to *preserve expressivity:*

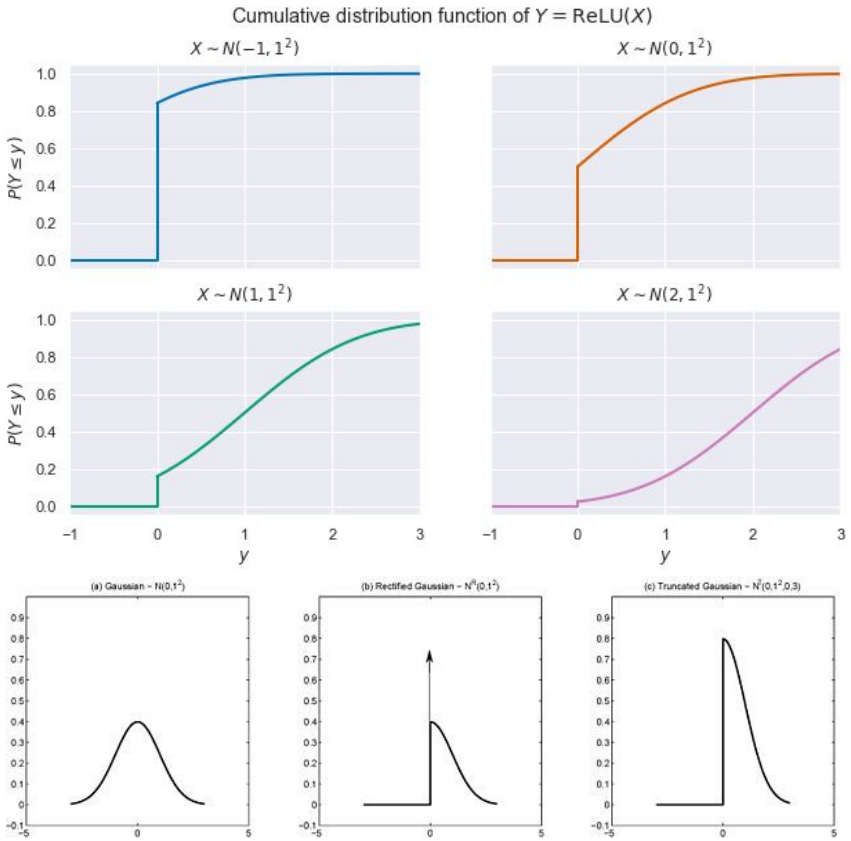
Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, we make sure that *the transformation inserted in the network can represent the identity transform.*

~ Original BN paper



In the context of ReLU

- **BN then ReLU** (or without BN) ->
  - output >= 0, obviously, with some (most) accumulated at 0. If BN learns gamma = 1, B = 0, top-right case. **Similar to no BN outs**.
  - Output distribution is obviously non-gaussian (mean > 0, with a mode at 0).
  - **NOT** a truncated gaussian! There's a point mass at 0.
- **ReLU then BN** ->
  - output: some shifted function from before
  - output distribution into next layer should be 'centered' to the B learnt by BN
  - To some, it 'makes more sense' because of it
  - Empirically better-ish? **Should be 'more expressive' at the very least? (wrt single neuron)**



In the context of ReLU

BN can be integrated into the weights of a layer (doing the same) if no non-linearity in-between.

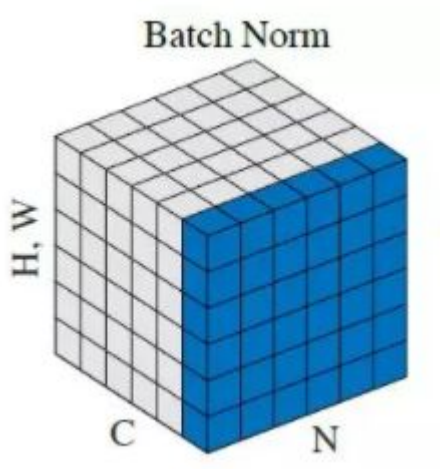
- **BN before ReLU**
  - BN+parameters **do not add** expressivity wrt **no BN**, you can achieve the same by integrating on the weights on the layer before
  - BN+parameters **add** expressivity wrt **BN only**, without them you always have 0 mean (so ‘’half’’ of activations die (not really, it’s the mean not the median, but close enough)), but the bias of neurons can be removed (aggregated in the mean and discarded). **Similar reason to why original paper included them for sigmoid**
- **BN after ReLU**
  - BN+parameters **do not add** expressivity wrt no BN, you can achieve the same by integrating all BN parameters in all neurons of the next layer; but **appears more expressive** for the neuron outputs
  - BN+parameters **do not add** expressivity wrt BN only, for the same reasons as previous point

So it becomes apparent BN is not about expressivity

HOW is Batch Norm (applied)

In the context of…

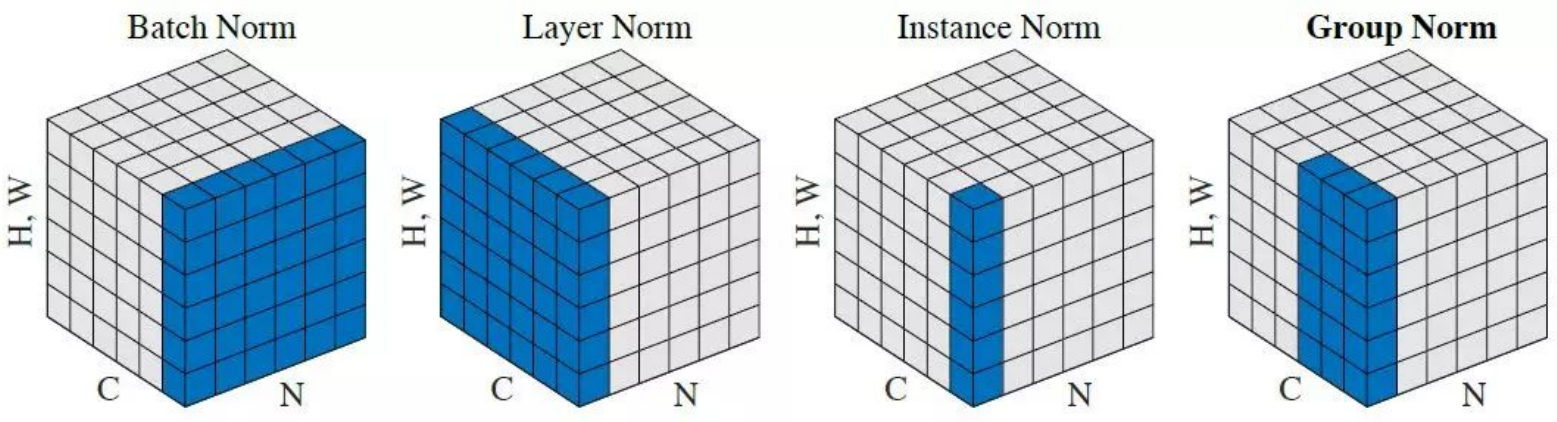
- Conv: normalise activations out of a feature: considering separately each feature: M = batch_size*H*W of output.
- If you put ReLU after BN (as in the original): forget biases of layers, they just get removed in the mean. Just don't add that parameter, (it may cause numerical instability).



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

$x_i = \ldots + b_i \Rightarrow x_i' = x_i - \ldots - b_i$

Other contenders, may reappear later



WHY Batch Norm (works)

Short answer

- No one quite knows yet. Original hypotheses have been proven false, new ones are a WIP
  - **ICS is** a thing, the inspiration for BN, but it seems **unrelated** by further works
  - BN, WeightN, NormProp (not quite for LayerN) seem to be doing the same: reparametrisation, and the effect is in the loss landscape mostly
  - BN is most likely **serendipity**

Original Hypothesis Internal Covariate Shift

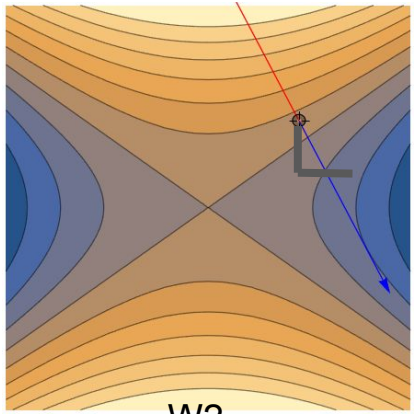
- ICS term: originally in paper *Improving predictive inference under covariate shift by weighting the log-likelihood function (1998-2000)*
- Reborn in *Batch Normalization: Accelerating … by Reducing Internal Covariate Shift*
- When training, gradients update **all** layers of an NN
- Updating weights of layer L changes the distribution of the activations coming out of it, and informally they say this is why vanishing gradients are a thing with sigmoids.
- At the same time, we update weights of layer L+1, which learn from the now changed layer L. This is ‘obviously’ bad for optimisation, so we can ‘fix it’ with Batch Normalization, keeping distribution ‘known’ (mean and std)

Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift

ICS explained II

$$\ell = F_2(F_1(\mathrm{u}, \Theta_1), \Theta_2)$$

$$\Theta_2 \leftarrow \Theta_2 - \frac{\alpha}{m} \sum_{i=1}^{m} \frac{\partial F_2(\mathrm{x}_i, \Theta_2)}{\partial \Theta_2}$$



2-layer function

Gradient update for 2nd function does not consider weights of previous function (besides the output acts)

Updating Theta1 **changes the input** to Theta2, so it changes the weights

BN does not ‘unchange’ the input, but tries to keep a ‘similar distribution’, which ‘**should help’ (no cite/proof of this, beside one tangentially related)**

New paper: *How does Batch Norm Help Optimization?*

Among many contributions, following are the most interesting:

- BN **increases** internal covariate shift (opposite of what was said originally!)
- **BN +** a **random** perturbation (mean and std != 0 and 1, and changing through time thus forcing **even MORE ICS**) improves the same as BN
- What BN seems to be doing is **‘smooth’** the loss landscape by improving its Lipschitzness and Beta-smoothness
- Paper also introduces interesting tools for analysing effects of architectural changes on the loss landscape

How Does Batch Normalization Help Optimization?

Scrutiny of How does BN help Optimization

BN vs Noisy BN

Algorithm 1 "Noisy" BatchNorm

% **For constants** n_m, n_v, r_m, r_v.

for each layer at time t **do**

  $a^t_{i,j} \leftarrow$ *Batch-normalized activation for unit j and sample i*

  for each j **do**

    $\mu^t \sim U(-n_\mu, n_\mu)$

    $\sigma^t \sim U(1, n_\sigma)$

  for each i **do**

    for each j **do**

      $m^t_{i,j} \sim U(\mu - r_\mu, \mu + r_\mu)$

      $s^t_{i,j} \sim \mathcal{N}(\sigma, r_\sigma)$

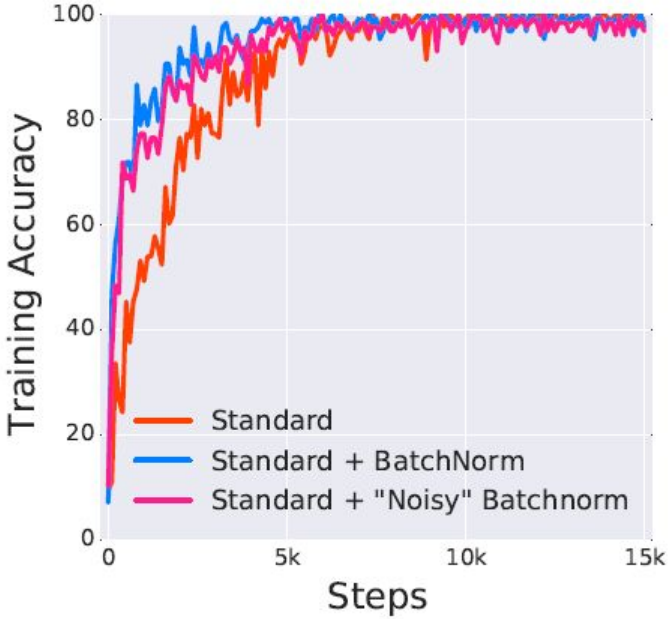
      $a^t_{i,j} \leftarrow s^t_{i,j} \cdot a_{i,j} + m^t_{i,j}$

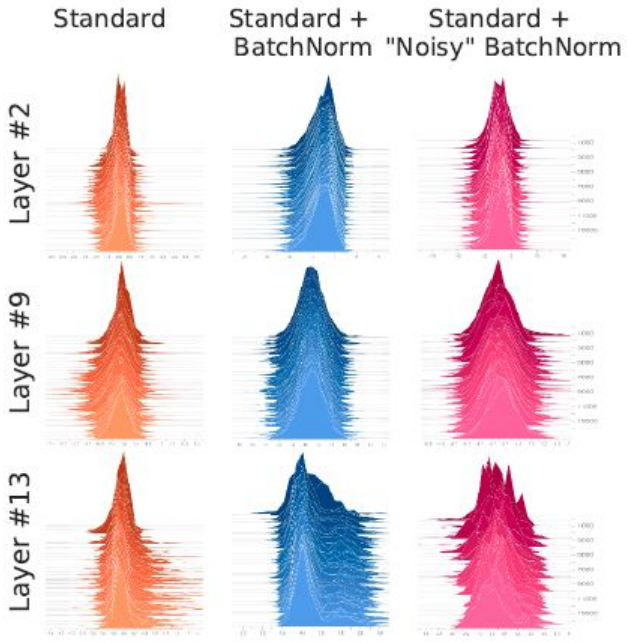
Create a perturbation generator for each neuron. Values are 0.5, 1.25 respectively. Each neuron gets a different mean and STD

Each neuron activation (for each img i) gets perturbed by two random numbers drawn from the above generator for its neuron.

Expected large covariance shift.

BN vs Noisy BN





Each stacked histogram is the ‘activation distribution across timesteps’, ‘closer’ = ‘more steps’

Despite the covariance shift being enormous (pink peak miss-alignment), esp. on layer 13, training follows mostly the same tendency.

Measuring ICS

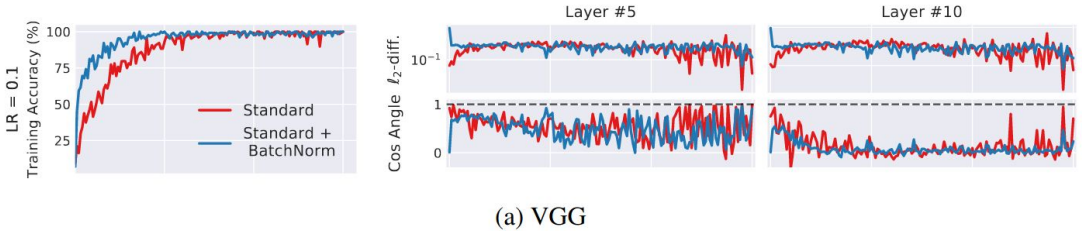
$$G_{t,i} = \nabla_{W_i^{(t)}} \mathcal{L}(W_1^{(t)}, \ldots, W_k^{(t)}; x^{(t)}, y^{(t)})$$

$$G'_{t,i} = \nabla_{W_i^{(t)}} \mathcal{L}(W_1^{(t+1)}, \ldots, W_{i-1}^{(t+1)}, W_i^{(t)}, W_{i+1}^{(t)}, \ldots, W_k^{(t)}; x^{(t)}, y^{(t)}).$$

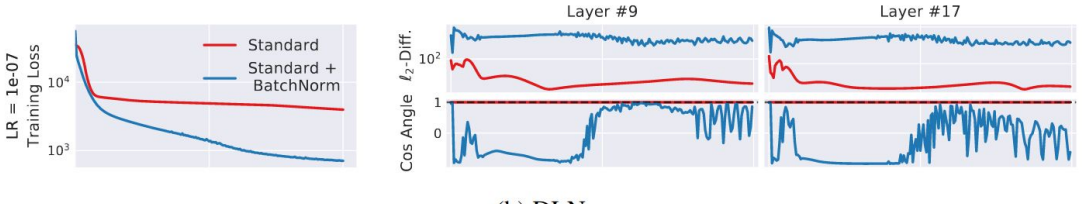
Compare gradient when network is updated ‘up to a layer *i*’ with original gradient

2 archs (each with and without BN)
2 measures (l2, cos)

BN has **more** ICS (at least on DLN or VGG in the ‘train’ part)



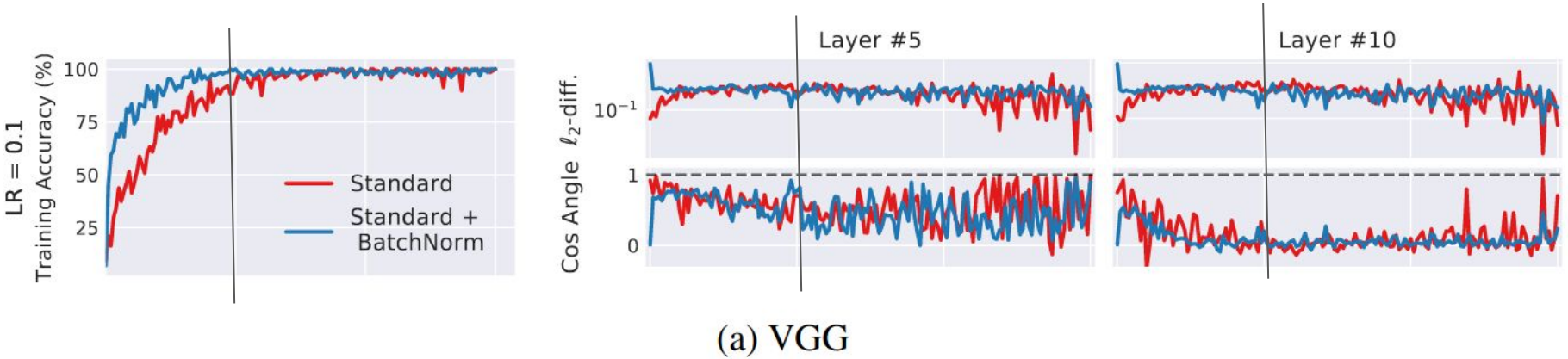
(a) VGG



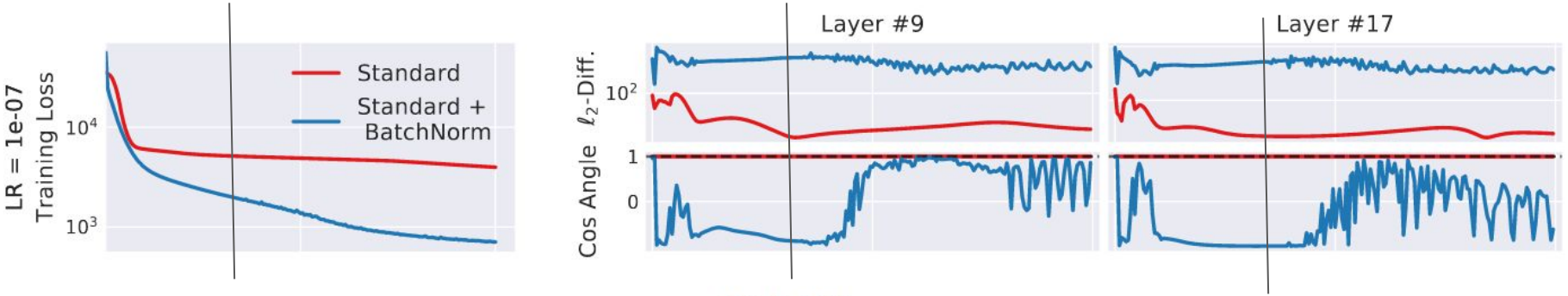
(b) DLN

25 FC layers, full batch descent, hell to train

Close up



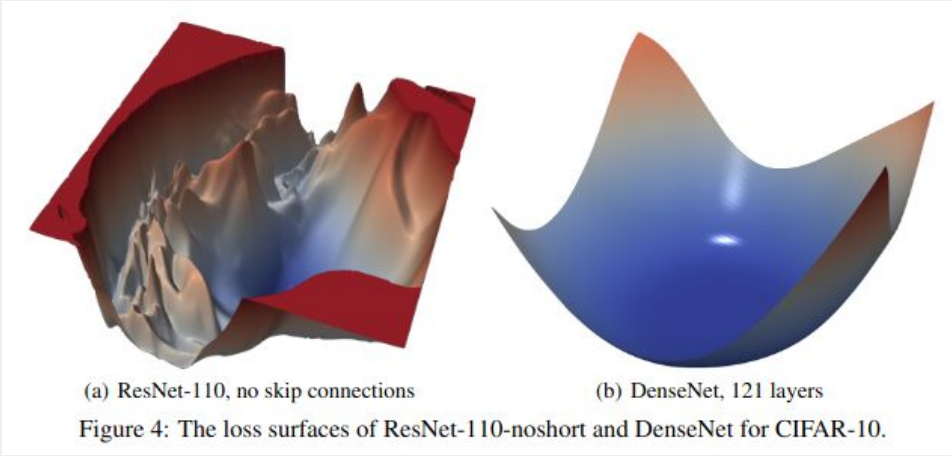
(a) VGG



(b) DLN

Then WHY Batch Norm?

Loss landscape, probably
(+ healthy noise?)

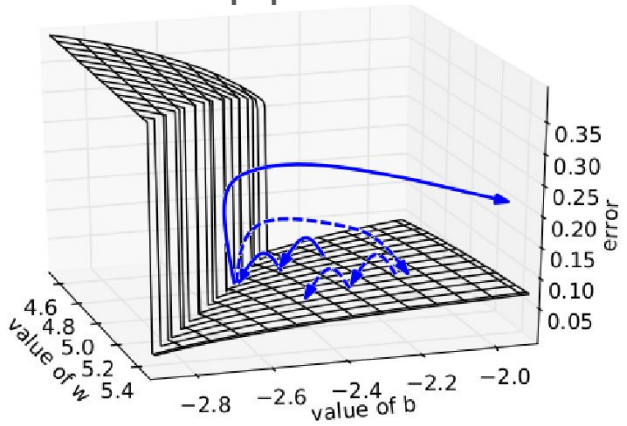


(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

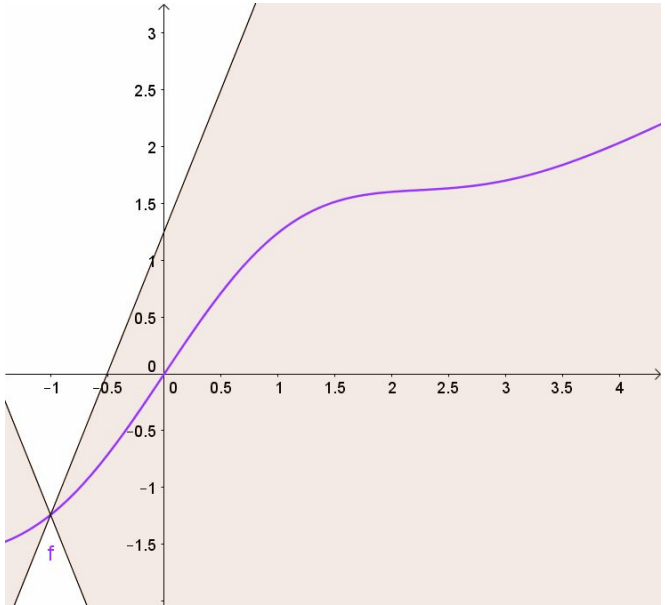
Figure 4: The loss surfaces of ResNet-110-noshort and DenseNet for CIFAR-10.

Lipschitzness L is provably lower with Batch Norm

- Recurrent concept appearing on papers talking ‘optimisation speeds’
- L-Lipschitz <-> $|f(x_1)-f(x_2)| < L^*|x_1 - x_2|$
- In our case, x are parameters, f is loss. Loss should change ‘less’ than weights at all points (factor of L)
- Means no sharp pit such as below



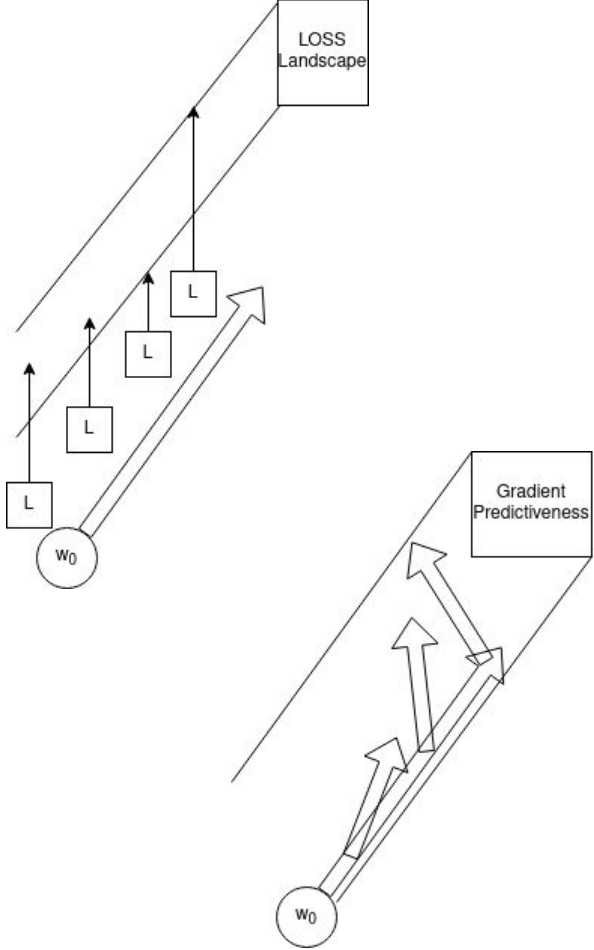
Source: [Gradient clipping](#) original paper



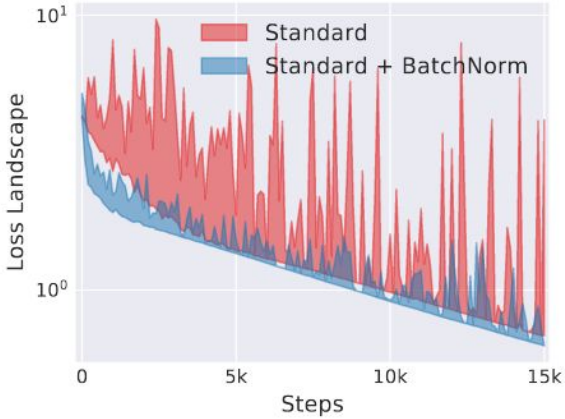
How does BN help Optimization (metrics)

- **Loss landscape**
  Each step, compute **losses** along gradient direction, compute **variability**
- **Gradient predictiveness**
  Each step compute **gradients** along gradient direction, compute **‘difference’**
- **Beta-smoothness**
  Gradient predictiveness, but **weigh each difference by the distance** travelled (the less distance, the most weight)

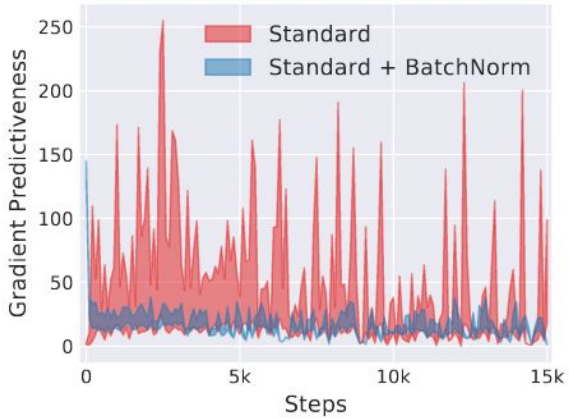
Bonus: custom mathematical proof for BN only about the loss Lipschitzness (*softness*)



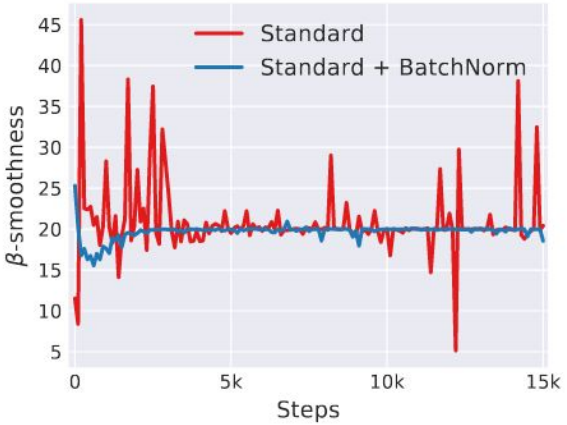
Batch Norm vs no Batch Norm



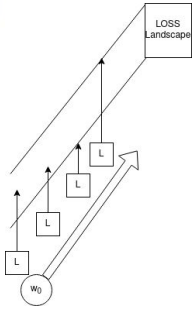
(a) loss landscape

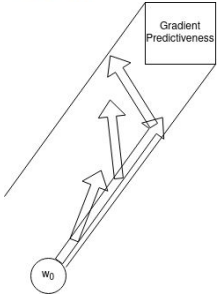


(b) gradient predictiveness

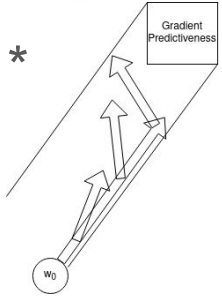


(c) "effective" β-smoothness





*

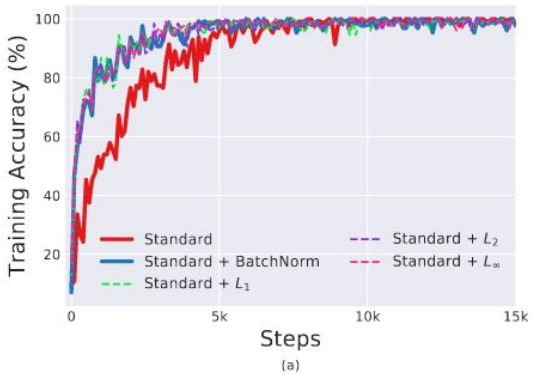


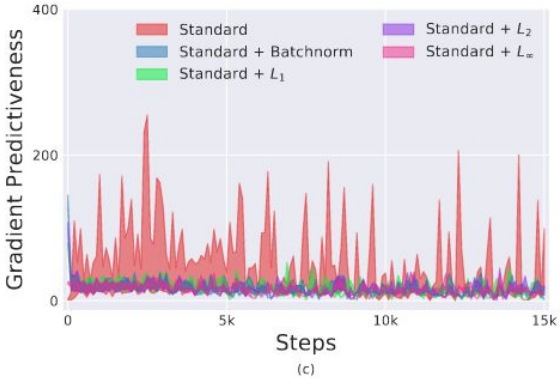
Similar things that also work? Norm scaling

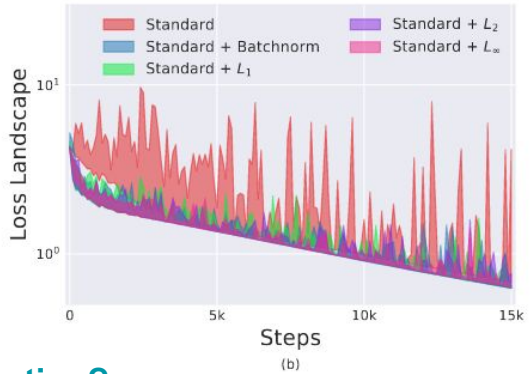
Take activations, divide by LP norm, after that shift mean. (all done before ReLU)

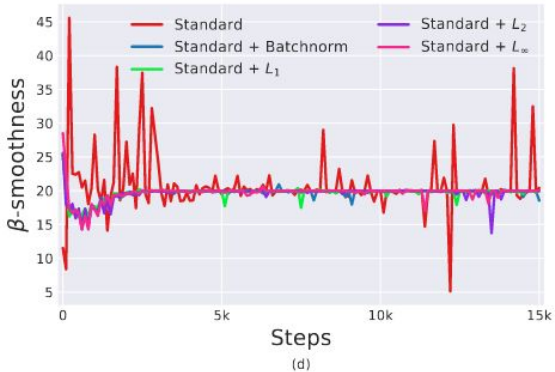
Architecture VGG-like.

All methods are similar, different from no-scale









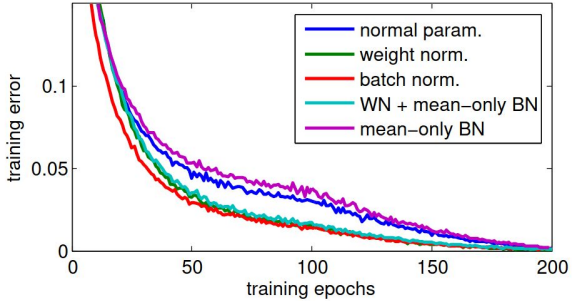
[How Does Batch Normalization Help Optimization?]

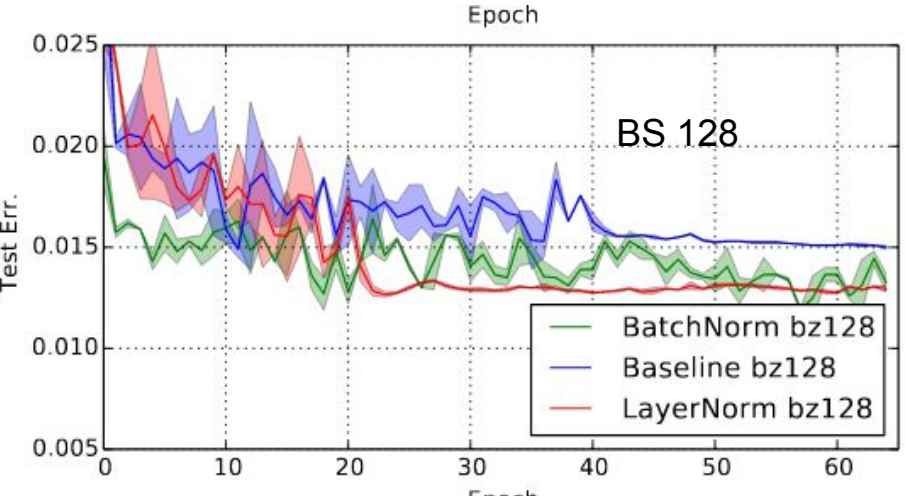
Weight Norm (top) vs layer norm (bot)

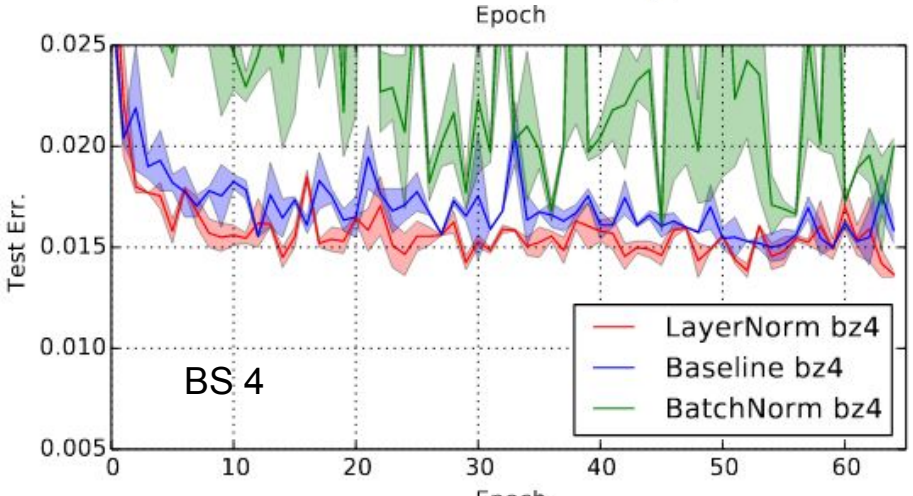
Scale:

Weight norm (Kingma): weights by L2 norm of weights

Layer norm (Hinton): acts by variance of LAYER







Considerations on similarity

Weight Norm: assuming that activations in layer L-1 are N(0,1), multiplying by a weight vector W results in activations that are N(0, ||W||) [Kingma] (before multiplying by parameter). So this is preserved for the 1 layer case. Multi-layer? Since [Kingma] **multiplies by a parameter v,** we can't know much about the output.

Layer norm: instead of normalising by output of a neuron, try to keep 'globally normal' by scaling all neurons for each image (separately). Opposed to WN and BN, **it is not a reparametrisation** (scaling the weights cannot achieve the same ops to scaling individual 'image acts'). Similar to L2 scaling in Feature extraction. Paper compares itself with BN in 1 traditional problem, marginal improvement. **Recommended for recurrent tasks** since BN is inapplicable there.

Norm propagation vs Weight Norm

$$\mathbf{o}_i = \frac{1}{\sqrt{\frac{1}{2}\left(1 - \frac{1}{\pi}\right)}} \left[\mathrm{ReLU}\left(\frac{\gamma_i (\mathbf{W}_i * \mathbf{x})}{\|\mathbf{W}_i\|_F} + \beta_i \right) - \sqrt{\frac{1}{2\pi}} \right]$$

- **Same as with WN**, but using “”””math””” to deduce expected values of the mean and std that result; theoretically more stable than BN since no moving mean/std. Assumes crazy things (whitened input always, no covariance, real world data is gaussian… and also that data is variance scaled)
- Empirically decent: CIFAR100 wins against BN by about 1-3% depending on if data augmented (no data augmentation -> 3%)

Weight Normalisation vs Batch Normalisation

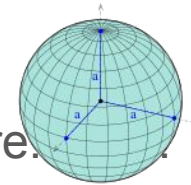
- BN: $\tilde{\mathbf{w}} := g\frac{\mathbf{w}}{\|\mathbf{w}\|_{\mathbf{S}}}$ where $\|\mathbf{w}\|_{\mathbf{S}} := \left(\mathbf{w}^\top \mathbf{S}\mathbf{w}\right)^{1/2}$ (S: **covar** x^*x^T).

  S is never computed explicitly, but appears due to activation values.
- WN: Same, but S = Identity (so w^Tw divisor, norm 2) -> **cannot model correlated values**).

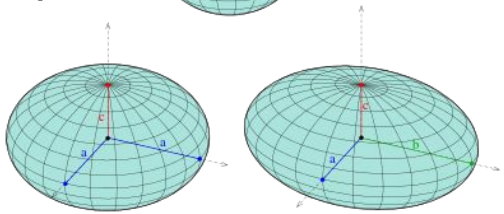
  Each neuron learns ‘a direction’

  A point in the sphere surface
- BN constraints to an **S**-sphere (ellipsoid), WN on a unit-sphere. **BN** also illustrates **covariances** between inputs

WN ->



BN ->



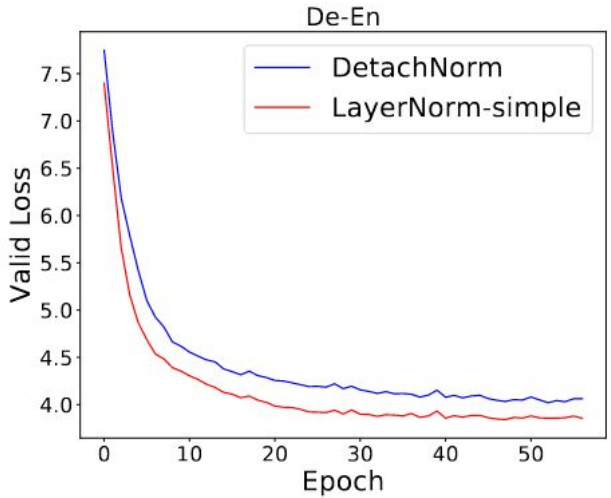
[Riemannian approach to batch normalization] / [Exponential convergence rates]

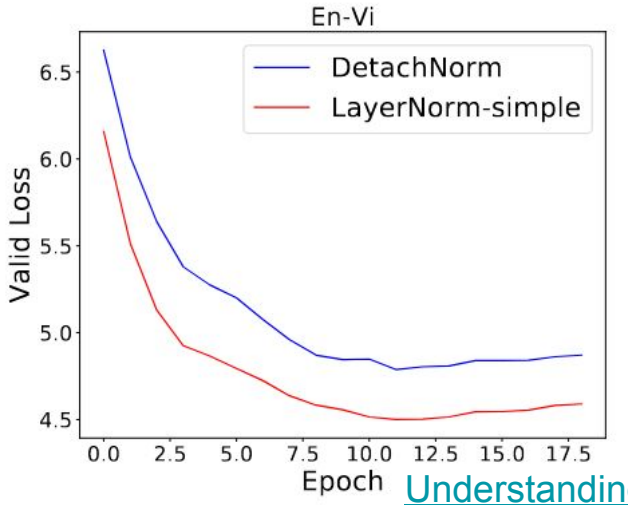
Detach-Norm vs Layer Norm

Is the ‘normalisation’ part important? Or the way the gradient changes?

Detaching the mean/std before application worsens performance

(note these results are in LN, but this may/should apply to BN too)



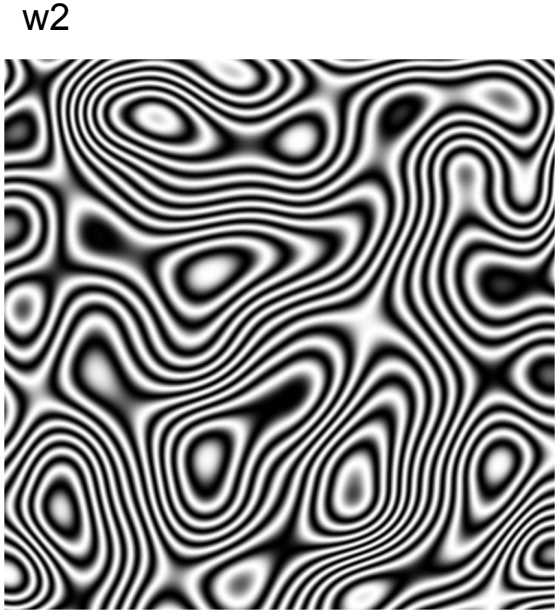


[Understanding & Improving Layer Normalisation]

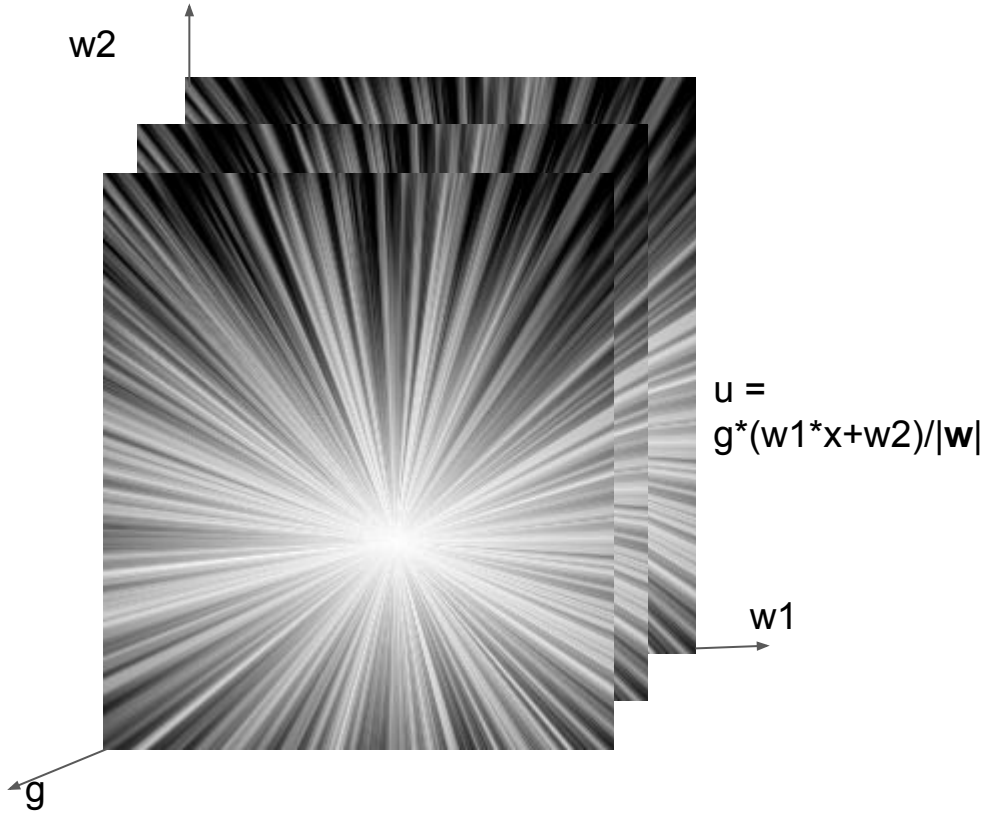
General idea: what re-parametrisation does

- Easier to optimize if neuron ‘direction’ is decoupled from ‘intensity’
- What does it mean ‘easier to optimize’? Different optics
  - Smoother loss landscape
  - Easier to learn some ‘particular functions’, and harder to learn others
  - Resilience to bad-luck batching
- Other things that improve smoothness? EG: Skip connections, regularisation (weight decay). Dropout ‘improves smoothness when looking at mean loss space through time’

Reparametrization visualised



u = w1*x+w2



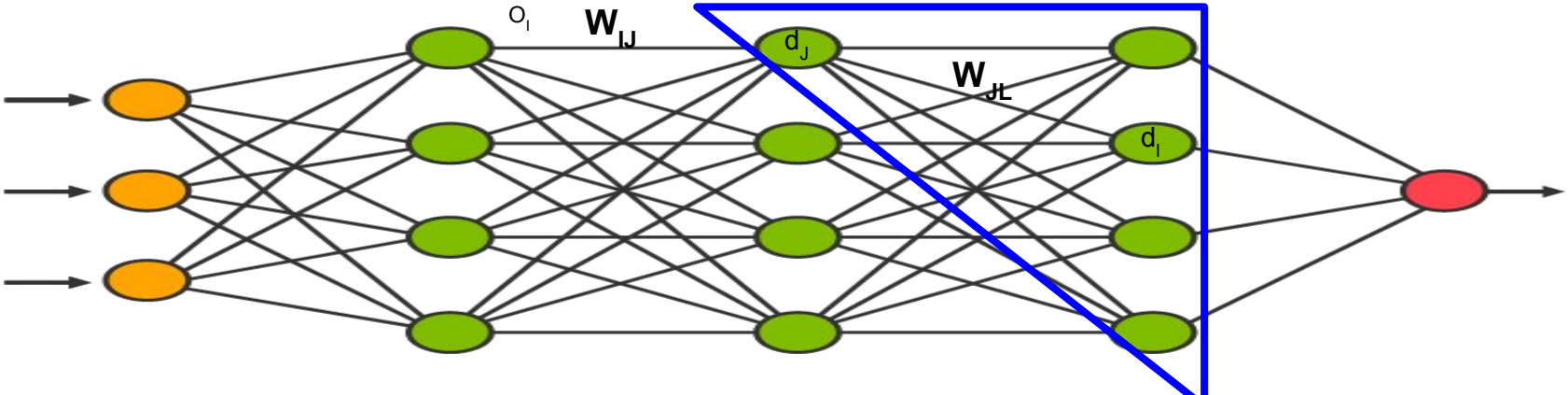
derivative of activation function

Recall the step rule (no BN):

$$\frac{dL}{dw_{ij}} = o_i * \delta_j = o_i * \left(\sum_{l \in L} w_{jl}\delta_l\right) * \frac{d\phi(net_j)}{dnet_j}$$

w_{ij} = weight feeding from neuron i (layer L) to j (L+1)

d_l = desire tendency of the neuron l (>0 wants increased, <0 decreased)



On complexity on learning functions

How would one learn to ‘shut down neuron j?’: must learn to lower either all w_{jl} (to exactly 0) or most w_{ij}(up to -inf).

Hard to guess what ‘we need’ for this, but it’s XOR(negative?(w,d,o)) = 1

wij	dj	oi	tendency	goes to 0
+	+	+	+	n
+	+	-	-	y
+	-	+	-	y
+	-	-	+	n
-	+	+	+	y
-	+	-	-	n
-	-	+	-	n
-	-	-	+	y

(Odd number of negatives)

tendency = - if different(dj, oi)

Each depends on tendency of a neuron and output of previous layer, probability of managing to do this several steps in a row is not high.

What about ‘identity?’ (recall resnets are good for this): must learn to lower all w_{ij} but one. Hard, too.

Reparametrisation helps: step rule for new params

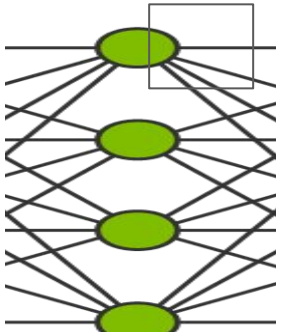
$\hat{x}_i$: out after normalisation. y_i: out after scaling (dl/dy_i as before = δ / desire)

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

$$\delta_j = \left(\sum_{l \in L} w_{jl}\delta_l\right) * \frac{d\phi(net_j)}{dnet_j}$$

Key: gamma and delta are independent of previous layer activations, only of this neuron's. Shutting it is now easy. ""Independent"" of 'neuron direction' in backward pass



x -> gamma&bias -> d

Furthermore: gradient of input before BN (with BN!)

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

- Gradient of **before** BN (so just after weight multiplication, case of BN then activation function) is different than without BN (obv)
- Gradient now depends on mean and std computed (μ_B, σ_B), which ‘rescales it and shifts in backward propagation’ (just as activations are in the forward)
- Brings us forth to the previous comment on LN detachnorm: "the **derivatives of the mean and variance are more important** than forward normalization by **re-centering and re-scaling backward gradients**" (at least in **LN**)

Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift

Understanding & Improving Layer Normalisation

What about in WN? Does not have ‘gradient shifting’

- Gradient now depends on mean and std computed (μ_B, σ_B), which ‘rescales it and shifts in backward propagation’ (just as activations are in the forward) in the case of BN
- In the case of WN, the re-parametrisation is w = **g*v/||v||** (Note how gradient of w is unchanging from not doing WN, it’s just that the gradient needs ‘postprocessing to apply it to the real parameters g & v)
- Hence, previous layer gradients are unchanged by reparametrisation in WN’s case, since **v weights are independent when w is known.**

$$\nabla_g L = \frac{\nabla_{\mathbf{w}} L \cdot \mathbf{v}}{||\mathbf{v}||},$$

$$\nabla_{\mathbf{v}} L = \frac{g}{||\mathbf{v}||} \nabla_{\mathbf{w}} L - \frac{g \nabla_g L}{||\mathbf{v}||^2} \mathbf{v},$$

Informally: g increases with how ‘aligned’ is gradient of w with v’s direction, & magnitude of gradient w

Weight norm

Exponential convergence rates for Batch Normalization: The power of length-direction decoupling in non-convex optimization

Reparametrisation on Least Squares Logistic

$$\min_{\tilde{\mathbf{w}}\in\mathbb{R}^d} \left(f_{\text{OLS}}(\tilde{\mathbf{w}}) := \mathbf{E}_{\mathbf{x},y}\left[\left(y-\mathbf{x}^\top\tilde{\mathbf{w}}\right)^2\right]\right) \qquad (11)$$

$$\overset{\text{(A1)}}{\Leftrightarrow} \min_{\tilde{\mathbf{w}}\in\mathbb{R}^d} \left(2\mathbf{u}^\top\tilde{\mathbf{w}} + \tilde{\mathbf{w}}^\top\mathbf{S}\tilde{\mathbf{w}}\right).$$

Least Squares original problem (convex) (remember: **u**, **S** are mean E(x*y), covar $E(xx^t)$)

$$\min_{\mathbf{w}\in\mathbb{R}^d\setminus\{0\},g\in\mathbb{R}} \left(f_{\text{OLS}}(\mathbf{w},g) := 2g\frac{\mathbf{u}^\top\mathbf{w}}{\|\mathbf{w}\|_{\mathbf{S}}} + g^2\right). \qquad (12)$$

Least Squares when 'weight-normalised'. Problem becomes nonconvex. Convergence rate is still linear (fast)

Exponential convergence rates for Batch Normalization: The power of length-direction decoupling in non-convex optimization

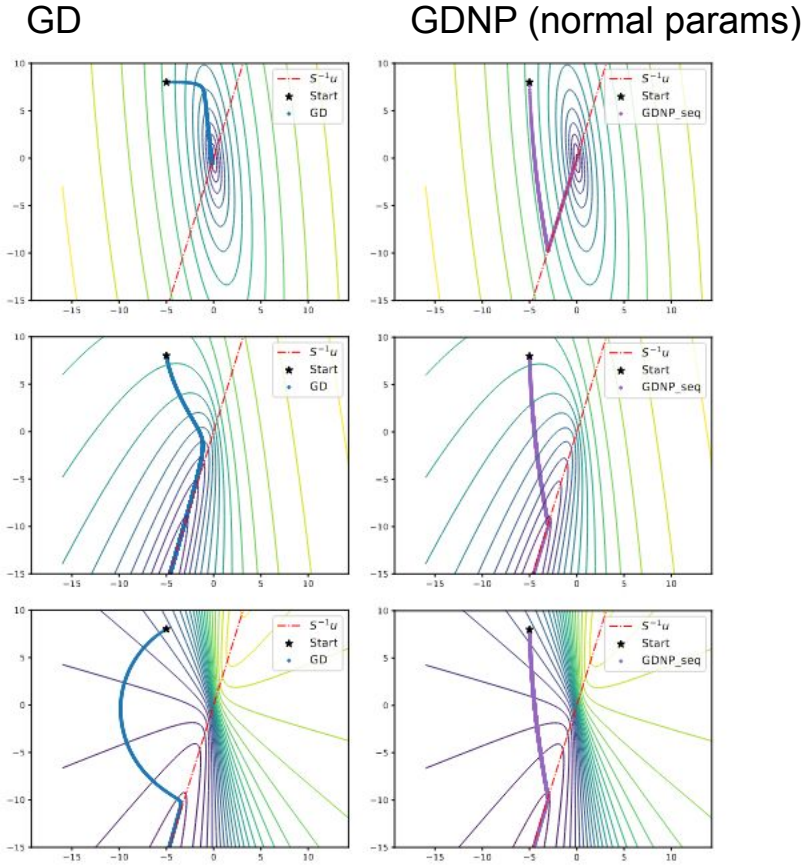
Learning Half-spaces (W*X > threshold)

On simple log-regression-like problems, path weights take through the loss. Left: traditional, right: optimise first ‘direction’ (until convergence: red line) then ‘scale’ (move on red line).

By row: least squares, logistic, sigmoidal regression; **same problem**. Note how right one always takes the ‘same path’ until the line.

Regardless of ‘row’, weight direction ‘should be the same always, but on regular GD, it takes different paths.

Later empirically show this works too if optimising direction & scale separately



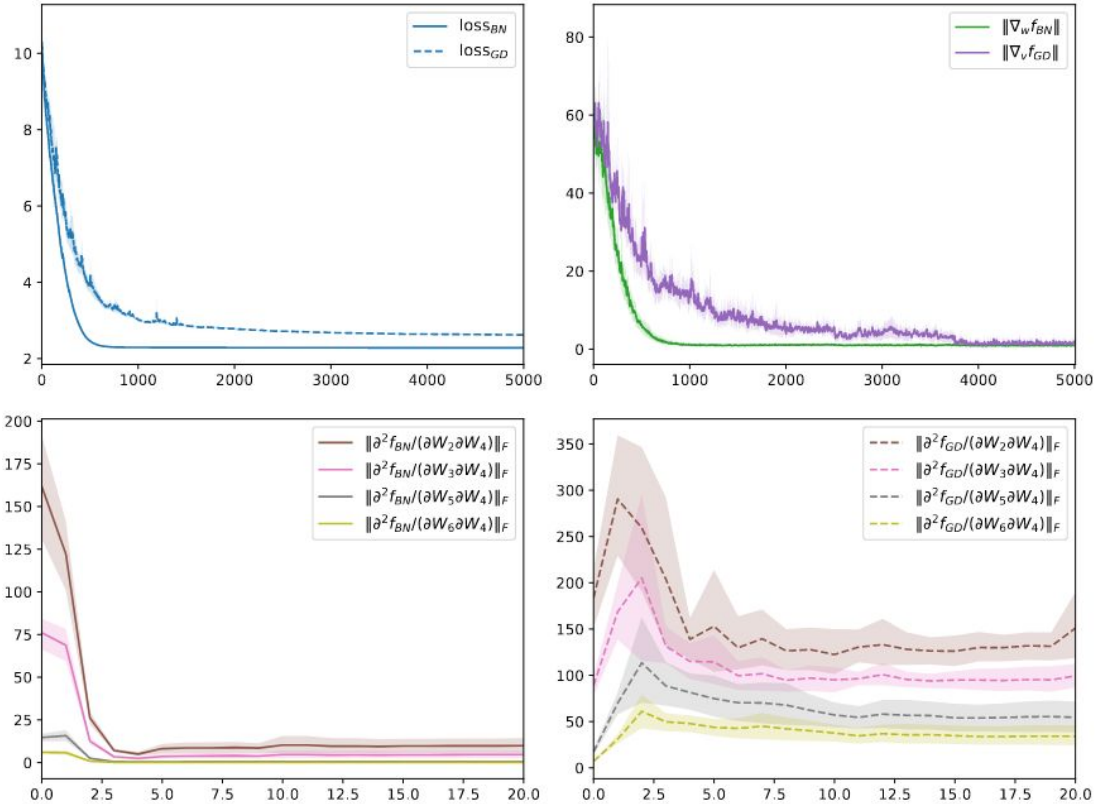
Learning Half-spaces result

- For S-spheres, there's proof that weights are optimisable 'linearly' with respect to target loss (L<= eps). For unit spheres (WN), no proof yet.
- Proof relies on data gaussianity (but empirically that doesn't matter(?))
- Proof relies on separate training direction and scale training (they claim: mainly to simplify the theoretical analysis)
- Proof does NOT rely on convexity of loss (cause it ain't, duh)

A = accelerated

Method	Assumptions	Complexity	Rate	Reference
GD	Smoothness	$\mathcal{O}(T_{\text{gd}}\epsilon^{-2})$	Sublinear	(Nesterov, 2013)
AGD	Smoothness	$\mathcal{O}(T_{\text{gd}}\epsilon^{-7/4}\log(1/\epsilon))$	Sublinear	(Jin et al., 2017)
AGD	Smoothness+convexity	$\mathcal{O}(T_{\text{gd}}\epsilon^{-1})$	Sublinear	(Nesterov, 2013)
GDNP	1, 2, 3, and 4	$\mathcal{O}(T_{\text{gd}}\log^2(1/\epsilon))$	Linear	This paper

Dependencies between layers



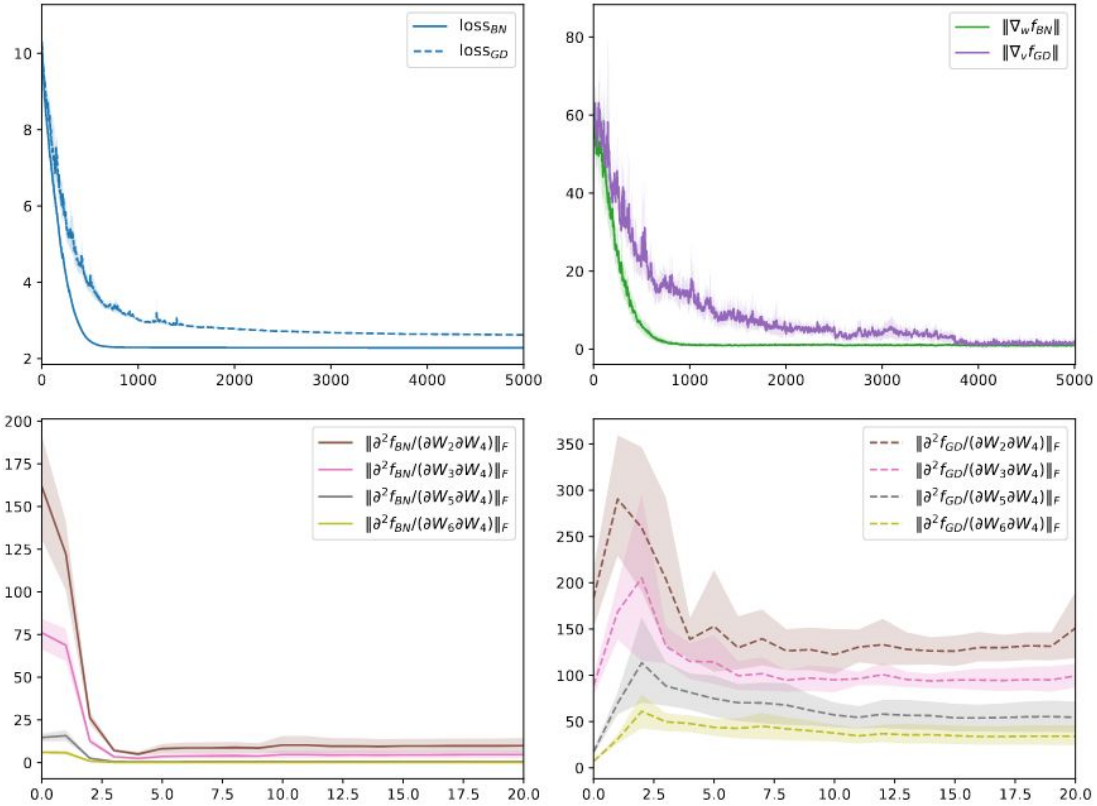
Loss (1) & gradient magnitude (2)

Dependencies between weight updates (layer {2,3,5,6} wrt layer 4) (BN left, no BN right). BN reduces dependencies as training goes

Proves: Given Gaussian inputs – the optimal direction of a given layer is independent of all downstream layers (**required** if want to prove fast training without training direction/scale separately)

“Batch Normalization layers indeed simplifies the networks curvature structure in w”

Dependencies between layers



ICS is defined as ‘how gradient of layer X’ changes when applying the gradient of the previous layer(s).

Counterpoint to previous paper, the ones below shows ‘how gradient of layer 2/3/5/6’ changes as we move in direction of gradient of layer 4.

How does Batch norm help **GENERALISATION?**

The idea is within original paper

3.4 Batch Normalization regularizes the model

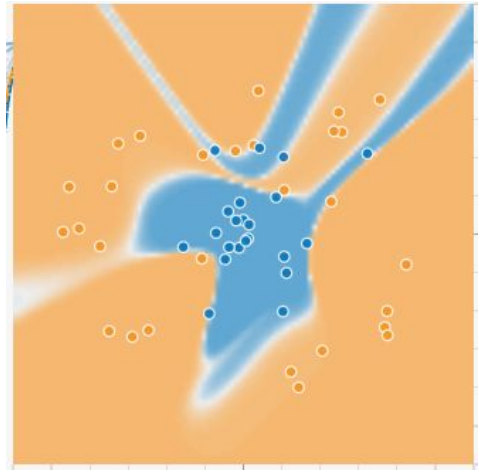
When training with Batch Normalization, a training example is seen in conjunction with other examples in the mini-batch, and the training network no longer producing deterministic values for a given training example. In our experiments, we found this effect to be advantageous to the generalization of the network. Whereas Dropout (Srivastava et al., 2014) is typically used to reduce overfitting, in a batch-normalized network we found that it can be either removed or reduced in strength.

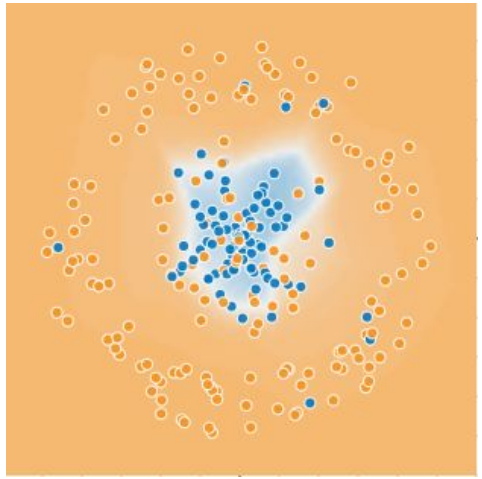
Healthy noise

Assume manifold such as right

- Adding noise may enhance the size of data and improve generalisation (data augmentation)
- Adding noise may ‘join’ parts of the manifold (**different manifold than real one**), changing the topology, changing what must be learnt, so **how much noise is good?**

- Not adding noise permits orange stragglers in the center to be **overfitted** more easily





How much Noise does BN cause?

- Let μ, σ be the ‘true’ mean and std of a neuron computed on the dataset
- Let μ’, σ’ be the mean and std of a neuron computed on a batch of *n* elements (pre-normalisation). What is the ‘expected noise?’
- The variance of the estimator μ’ **$\mathbf{Var(\mu') = \sigma^2/n}$**. As *n* increases (larger batches) it is less ‘wrong’, proportional to variance
- The variance of the estimator **sigma’** depends on the kurtosis (4th moment μ_4) and σ^4; (also tends to 0 with *n* -> infinity)

$$\mathrm{Var}(S^2) = \frac{\mu_4}{n} - \frac{\sigma^4 (n-3)}{n(n-1)}.$$

- **‘Running statistics’** slowly **reduce the noise** (increase *n*)
- **Summary: mean and variance estimators used have a ‘measurable’ noise, depends on BatchSize (n) (layernorm plot confirms it)**

Activation noise

- Small perturbations in an activation can have a similar effect to input perturbations
- But now, activation correlation may mean that noise is ‘discardable’ if two other neurons compute something similar
  - More formally, it can learn linear combinations of neurons to cancel added/multiplied noise
- Small perturbation in most neurons in general could work the same, achieving de-overfitting impact in the loss landscape (reduction of sharp minima through a better approximation of p(X)
- Another example: Dropout

Hypothesis: BN is length-direction decoupling + Healthy noise

Activation noise

- Only one example seen of this, in Parametric Noise Injection
- Paper introduces noise as a way to reduce adversarial effectivity, to a very impactful degree, at the cost of accuracy
- Paper does not show noise in training is effective for generalisation, but it does show **it is effective for adversarial attacks**
- It is possible both effects are related

Summary

- BN implicitly reparametrises the network, like WN or NP
- BN reparametrises differently than WN, taking covariances into account
- Reparametrisation causes length and direction of weights to be decoupled, provably improving convergence speeds
- The important part is the scaling, not the biases
- When is BN a good idea? It's pretty much free, so wherever you want to put it
- Should I use the additional parameters? In most cases they don't add expressivity, but they help with optimising, so yes
- Before or after activation function? If before, make sure to put the additional parameters. Do what you want otherwise.

Parametric Noise Injection

Idea: use noise like this to prevent adversarial attacks.

How much? Let network decide: add a parameter (a_i, one for each layer) which we multiply by gaussian noise (std proportional-ish to the activation std), and add to activations. Noise being 0-centered means that it is random whether a_i increases or decreases in each activation. With momentum, it tends to 0.

They add ‘robust optimisation’, adding noise to input in a specific way, so that a_i does not converge to 0 (beyond scope of this presentation)

	Test with PNI			Test without PNI		
	Clean	PGD	FGSM	Clean	PGD	FGSM
Vanilla adv. train [32]	-	-	-	83.84	39.14±0.05	46.55
PNI-W	84.89±0.11	**45.94±0.11**	**54.48±0.44**	85.48	31.45±0.07	42.55
PNI-I	85.10±0.08	43.25±0.16	50.78±0.16	84.82	34.87±0.05	44.07
PNI-A-a	85.22±0.18	43.83±0.10	51.41±0.08	85.20	33.93±0.05	44.32
PNI-A-b	84.66±0.16	43.63±0.20	51.26±0.09	83.97	33.53±0.05	43.37
PNI-W+A-a	85.12±0.10	43.57±0.12	51.15±0.21	84.88	33.23±0.05	43.59
PNI-W+A-b	84.33±0.11	43.80±0.19	51.14±0.07	84.42	33.30±0.05	43.43

V:Noise in loss?

What if we train with a similar approach: get noise and add it/multiply it by activations. Noise is pondered by some weights gamma/beta (each layer, neuron, or global), which in the loss are set to maximise (with a hyperparam on how much we care).

NN should be BN’d or isomorphised before to ensure activations cannot grow too much (else noise optimisation is trivial)

Effect on adversarial attacks? Potentially interesting.