https://arxiv.org/pdf/1607.06450.pdf

$a_i^l = w_i^{l^T} h^l$
$h_i^{l+1} = f\left(a_i^l + b_i^l\right)$
w_i^l is the weight to the ith hidden unit in layer l
one of the challenges of deep learning is that hte gradients wrt the weights in a layer are highly dependent on the outputs of the neurons in the previous layer. Especially if these outputs change in a highly correlated way

- Q: why is this is a challenge?

Batch normization was proposed to reduce such undesirable "covariate shift". The method normalizes a_i^l over the training cases.

- https://arxiv.org/pdf/1603.09025.pdf

BN(x) = $\gamma\widehat{x} + \beta$, learn γ and β
where $\widehat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ for a given activation (column) k
where $\mu_B = \Sigma x_i / m$ across the mini-batch
and $\sigma_B^2 = \Sigma(x_i - \mu_B)^2$

Batch Normalization: Acceleratin Deep Network Training by Reducing Internal Covariate Shift: https://arxiv.org/pdf/1502.03167.pdf

- Coviarate shift is a change in the distribution of the inputs to a model.
- This occurs continuously during the training of feed forward networks, since changing the parameters of layer l changes the distribution of the inputs to layers l+1, l+2, ...
- This degrades the efficiency of training
  - have to think very carefully about init params
  - have to set low learning rates
- Q: Why?
  - My intuition is that
  - isnt this somethign you can control in the input data though? As long as the data in each batch are not correlated somehow, and their correlation is representative of the entire dataset, you should not be able to do any better...?
  - you need lower learning rates

Calculating gradients over mini batches, over indiviual samples one at a time, is helpful in several ways
1. the gradient loss over a mini batch is a better estimate of the gradient over the training set

than a single sample
2. one computation over a batch of size m is more efficient than m computations over m samples

In training neural networks, small changes in a layer's parameters have large impacts on subsequent layer

- this is because the outputs from layer become inputs to layer 2, etc...

Even within more local subnetworks of a large network, it is generally advantageous for the distribution of inputs to be fixed over time

- Q: Why?

Fixed distributoin of inputs to a subnetwork would have positive consequences for hte layers outside the subnetwork as well.
- Imagine we have

$$z = g(Wu + b),\ g(x) = softmax(x) = \frac{1}{1 + \exp(-x)} = \frac{\exp(x)}{\exp(x) + 1},\ g'(x) = g(x)\,(1 - g(x))$$

as $|x| \to \inf,\ g'(x) \to 0$
so for all dimensions of $x = Wu + b$ that are not small absolute values, the gradient flowing to u will vanish and the model will train slowly

However, since x is affected by $W,\ b$ and the parameters of preceding layers, changes to these parameters during training will likely move many dimensions of x into the saturated regime of the non linearity and slow down the convergence

- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z}\left(\frac{\partial z}{\partial g} = 1\right)\frac{\partial g}{\partial f(W) = Wu + b}\frac{\partial f(W)}{\partial W}$
- Q: Why likely?
- Assuming the layer input $u = g(x_1)$, i.e. it is the output from a previous layer, then we know $u \in \{0,\ 1\}$
- so there is not significant chance of Wu + b going towards inf if you init W and b nicely...?

In practice, mitigate this by

- using Relu: $Relu(x) = \max(x,\ 0)$
- careful initialization
- small learning rates

Internal covariate shift = change in the distributoin of network activations due to the change in network parameters during training

- to improve training, reduce internal covariate shift
- expec the training speed to improve if we fix the distribution of hte layer inputs as the

training progresses
- whiten the data i.e. zero mean unit variance?

The idea is to, as far as possible, 'fix' the distribution of the every input feature to a layer.

Can you use a whitening activation at every training step? Ether by modifying the netowrk directly, or by changing the parameters of the optimzation algorithm to depend on the network activation values?
A. If these normalization modifications are interspersed with optimization steps, then the gradient descent step may update the parameters in a way that requries the normalization to be updated, which reduces the effect of the gradient step!
Say we had
$x = u + b$ where we try and learn b, with some loss $L(x)$
$e.g.\ u = [3, 4, 5, 6, 7],\ b = [-7, -9, -11, \dots]$
$x = [-4, -5, -6, -7, -8]$
$E[x] = -6 = \frac{1}{N}\Sigma(u_i + b_i)$
$\hat{x} = [2, 1, 0, -1, -2]$
$L = \Sigma\widehat{x^2} = 10$
$\frac{\partial L}{\partial b_1} = \partial(x_1 - E[x])^2 - 2x_2E[x] - 2x_3E[x] \dots + 4E[x]^2)/\partial b_1$
$= \partial\left(x_1^2 + nE[x]^2 - 2E[x](x_1 + x_2 + \dots + x_n)\right)/\partial b_1$
$= 2x_1 + n.2E[x].\frac{1}{n}.1 - 2E[x].1$
$= 2x_1$

then $E[x] = \Sigma x_i / N$ over the whole training data set
and normalize by $\hat{x} = x - E[x] = u + b - E[u + b]$, which we then use in the feed-forward calculation
then $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial \hat{b}} = \frac{\partial L}{\partial \hat{x}}\frac{\partial(u + b - E[x])}{\partial \hat{b}} = \frac{\partial L}{\partial \hat{x}}\left[\frac{\partial b}{\partial b} - \frac{\partial E[x]}{\partial b}\right]$
However, if the gradient step ignores that the normalization E[x] is also dependent on b, then we will get an update rule $b \leftarrow b + \Delta b$ where $\Delta b \propto \frac{\partial L}{\partial \hat{x}}$
then $\widehat{x_{new}} = u + (b + \Delta b) - E[u + (b + \Delta b)] = u + b - E[u + b] = \hat{x}$?
- presumably this is only true if $\Delta b = E[\Delta b\}$?

So the combination of the update to b AND the subsequent normalization led to no change in the output of the layer! Or the loss

- so as training progresses, b grows indefinitely but with no change to the loss function...
- Takeaway: the gradient descent optimization does not take into account the fact that the normalization takes place!

Models blow up when the normalization parameters are computed outside the gradient descent step.

- Q: What do you really want to do here instead?

The issue with the approach is that the gradient descent optimization does not take into account the fact that the normalization takes place.

Instead, you want to ensure that for any parameter values, the network always produces activations with the desired distribution.

- This way, the gradient of loss wrt model params will account for the normalization and for its dependence on the model params

For a layer with d dimestional input $\mathbf{x} = \{x_1, x_2, ..., x_d\}$
normalize each dimension

So you want a stable distribution on the inputs to the non-linearities

$$\overline{a}_i^l = \frac{g_i^l}{\sigma_i^l}\left(a_i^l - \mu_i^l\right)$$

$$\mu_i^l = E_{\mathbf{x} \sim P(\mathbf{x})}\left[a_i^l\right]$$

$$\sigma_i^l = \sqrt{E_{\mathbf{x} \sim P(\mathbf{x})}\left[\left(a_i^l - \mu_i^l\right)^2\right]}$$

where g_i^l is a gain parameter scaling the normalized activation before the non-linear activation
The expectation is under the whole training data distribution, but it is typically impractial to copute this exactly, since it would require running the entire training set through the network with the current set of weights.
Instead, μ and σ are estimated ising the empirical samples from the current mini-batch

- Q: what does this mean? for a given 'snapshot' of weights during training, standardize
- (during testing/inference you are just using the weights that alreayd exist, so you dont think about this)

But this puts constraints on the size of a mini-batch. and it is hard to apply to recurrent neural networks.
- Q; Why is there a constraint on the mini batch size? In theory if you wanted to, you _could_ run the entire training set as one batch?

Q: Why is this a problem for recurrent neural networks?

- in part this just feels like a problem of nomenclature/semantics: in an RNN, the same 'layer' = shared weights gets at multiple timesteps, but each timestep captures unique characterisitcs of the input, and you will likely lose signal by standardizing at each time step wrt means/sds *across all the time steps*!
- in an RNN a layer shares the weights W_{hh}, W_{xh}, W_{hy} across time
  - if you were to unroll the layer, it would be the same layer feeding itself n times
- the same batch goes through all the time steps from 0 till T
- but at each time step t, we capture the influence of the previous time steps t-1, t-2, ... 0
  - and in a bidirectional RNN, possibly future time steps as well
- However, if you were normalize a_i^{l,t_1} using $\mu(a_i^{l,t_1}, a_i^{l,t_2}, ..., a_i^{l,T})$, you will dilute the specific time specific signal captured by a_i^{l,t_i}
- The only meaningful normalization is with respect to the current timestep
- Something like this is outlined here: https://stackoverflow.com/questions/45493384/is-it-normal-to-use-batch-normalization-in-rnn-lstm