11
Confronting Vanishing Gradients: Batch Normalization and ResNets

The remarkable success of deep learning is built on a simple premise: with more layers, a network can learn more complex and hierarchical features, leading to better performance. While this "deeper is better" philosophy was always appealing, training very deep nets had not worked. Starting 2012 ("AlexNet moment") it became clthat more data and better engineering could get over this obstacle, and a flurry of followup work (notably the VGG models) took this further. However, in the mid-2010s, researchers hit a wall. Simply stacking more layers on top of existing architectures led not to better models, but to ones that were impossible to train. There was a fundamental obstacle that seemed not be overcome with more data: **the vanishing gradient problem**. This chapter tells the story of how this problem was diagnosed and overcome, leading to landmark architectures like ResNet that form the backbone of many modern deep learning systems. Some innovations also influenced subsequent architectures such as transformers.

11.1 The Vanishing Gradient: Error Signal Fades to Nothing

At its core, training a neural network is about communication: the loss is computed at the final layer and converted via the back-propagataion algorithm to an error signal that is communicated backward (with suitable modifications) through the network, layer by layer, telling each parameter how to adjust itself to lower the error.

The problem arises from the repeated application of the chain rule through many layers. Each layer's activation function has a derivative that acts as a multiplicative factor in the gradient calculation. This is the signal being propagated, and it can get weaker as it propagates through the layers. Let's consider a simple, deep network composed of a chain of functions: $\widehat{y} = f_L(\dots f_2(f_1(x)))$. The gradient for a

weight in an early layer, say w_1 in f_1, will be proportional to a product of derivatives from all the subsequent layers:

$$\frac{\partial \text{Loss}}{\partial w_1} \propto \frac{\partial f_L}{\partial f_{L-1}} \cdot \frac{\partial f_{L-1}}{\partial f_{L-2}} \cdots \frac{\partial f_2}{\partial f_1}$$

Many traditional activation functions, like the sigmoid function $\sigma(z) = 1/(1+e^{-z})$, have derivatives that are always small. The maximum value of the sigmoid's derivative, $\sigma'(z)$, is 0.25. When we multiply many numbers that are significantly less than 1 together, the product shrinks exponentially fast.

Example 11.1.1 (The Fading Whisper)**.** *Imagine a simple network where each layer just consists of a single neuron with a sigmoid activation and a weight $w_i = 1$. The output of layer i is $z_i = \sigma(z_{i-1})$. To update the first neuron, the gradient signal must pass backward through all the other neurons. The chain rule dictates that we multiply the derivatives of each activation function:*

$$\textit{Gradient signal for layer 1} \propto \sigma'(z_{L-1}) \cdot \sigma'(z_{L-2}) \cdots \sigma'(z_1)$$

Even in the best-case scenario where every neuron is in its most sensitive state (i.e., its input is 0, so $\sigma'(z_i) = 0.25$), the gradient signal for the first layer in a 10-layer network would be scaled by $(0.25)^9 \approx 3.8 \times 10^{-6}$. In a 20-layer network, this factor plummets to the order of 10^{-12}.

The error signal from the output, which might have started as a clear instruction, becomes a faint, imperceptible whisper by the time it reaches the early layers. As a result, these early layers, which are responsible for learning the most fundamental features, receive virtually no update signal, and remain frozen at or near their random initializations.

Problem 11.1.2. *Suppose the network uses ReLU non-linearity instead of sigmoid. ($ReLU(z) = \max(0, z)$, has a derivative that is either 0 or 1). How does using ReLU instead of sigmoid affect the vanishing gradient problem? Does it solve it completely? What new problem might it introduce?*

11.1.1 Concrete manisfestation: Deeper is Worse!

The issue of vanishing gradients issue was known for decades but around 2014-2015, it became known as a bottleneck in obtaining further improvements from deep learning. Researchers found that when training very deep networks for tasks like image recognition, performance would saturate and then, as more layers were added, rapidly degrade.

This was not a problem of overfitting! In a typical overfitting scenario, the training loss continues to decrease while the validation (or

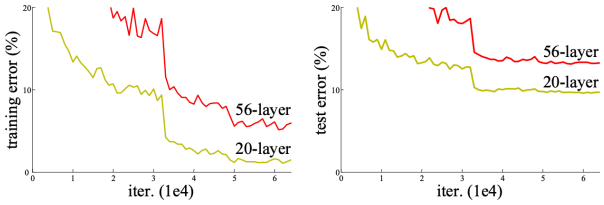


Figure 11.1: Degradation problem arising in training vanilla CNNs on CIFAR10 dataset using of 20 and 56 layers respectively. (From ResNet paper (He et al 2015.)

test) loss begins to rise. Here, something far stranger was happening: the **training loss** of a deeper network (e.g., 56 layers) was *higher* than that of its shallower counterpart (e.g., 20 layers) as shown in Figure 11.1. Called the *degradation problem*, it pointed to fundamental failure of (Stochastic Gradient Descent) to find a good solution once depth increases too much.

Vanishing gradients went from a theoretical curiosity to practical impediment; they were actively preventing deeper models from learning even basic functions that shallower models could master with ease. This sparked a wave of innovation, which is the subject of this chapter.

11.2 *Batch Normalization: Taming the Internal Dynamics*

The degradation problem made it clear that as networks grew deeper, the process of propagating the training signal became unstable. The activations in deep layers could drift to extreme values, or their distributions could shift so dramatically during training that the subsequent layers were forced to constantly adapt to a moving target. The first major breakthrough in solving this was not an architectural change, but a clever normalization technique: **Batch Normalization** (BN) [1].

At its heart, the idea is simple and direct: if the distribution of a layer's inputs is causing problems, why not fix it? Batch Normalization forces the activations at the input of each layer to have a consistent, predictable distribution throughout training.

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pages 448–456, 2015

11.2.1 *The Mechanism*

For a mini-batch of data during training, BN computes the mean and variance of the activations for each feature (or channel, in a CNN). It then normalizes these activations to have a mean of zero and a variance of one. The procedure for an activation x_i in a mini-batch B is:

1. **Calculate mini-batch statistics:**

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{and} \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

2. **Normalize the activation:**

$$\widehat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

   where ϵ is a small constant for numerical stability.

3. **Scale and shift:**

$$y_i = \gamma \widehat{x}_i + \beta$$

The crucial final step introduces two new learnable parameters per feature, γ (scale) and β (shift). This is vital. By forcing all activations to a standard normal distribution, we might limit the network's expressive power. These parameters allow the network to learn the optimal distribution for each layer's activations. In the extreme, it could learn $\gamma = \sqrt{\sigma_B^2 + \epsilon}$ and $\beta = \mu_B$, recovering the original activation if that were optimal.

At inference, when there is no mini-batch, BN uses a running average of the means and variances collected during training to perform the normalization. It is customary to maintain an exponential moving average of these parameters, and it suffices to use those values from the end of training.

Problem 11.2.1. *In BN after normalizing activations to have zero mean and unit variance, we apply a learnable scale (γ) and shift (β). What expressive power would the network lose if we omitted them and simply used the normalized activations $\widehat{x}_i$? (Note: you have to think through whether the network could learn to adapt, or not.)*

11.2.2 *The Diagnosis: From Internal Covariate Shift to a Smoother Landscape*

The original motivation for Batch Normalization was to combat what the authors called **Internal Covariate Shift**. The idea is that as the weights in early layers are updated, the distribution of inputs to

later layers is constantly changing. This forces the later layers to continuously adapt to a non-stationary input distribution, much like trying to learn a function while someone is constantly shifting and scaling the coordinate axes. By stabilizing the mean and variance of each layer's inputs, BN was thought to reduce this shift and make learning easier.

While this intuition is appealing, subsequent research has shown that the story is more nuanced. The beneficial effects of BN seem to be less about fixing covariate shift and more about its profound impact on the optimization landscape itself. The key insight is that Batch Normalization makes the loss landscape significantly **smoother**.

Without BN, the loss can be a chaotic function of the parameters, with sharp valleys, flat plateaus, and steep cliffs. This forces the use of small learning rates to avoid "overshooting" a minimum and diverging. BN reparameterizes the network in a way that smooths out many of these pathological curvatures. This is like turning a jagged, treacherous mountain range into a landscape of smooth, rolling hills, which is much easier for an optimizer like SGD to navigate. This smoothing effect allows for much higher learning rates, which is the primary reason for the dramatic speed-up in training observed with BN-equipped networks.

"Regularization effect:" While BN was invented to improve optimization, it is also widely observed to have a regularizing effect, often reducing the need for other noise-based regularizers like Dropout. This effect is thought to arise from the stochastic nature of its training-time computation. For any single training example, its normalized output depends on the statistics of the other examples randomly chosen to be in its mini-batch. This added noise is thought to prevent the network from becoming overly confident in the features of any individual example, pushing it to learn more robust and generalizable representations.

11.2.3 A Theoretical Surprise: Scale Invariance and the Effective Learning Rate

Further study has revealed that BN reveal fundamentally alters the relationship between different parts of the optimization process. One of the most important properties of a weight layer followed by BN is **scale invariance**.

Consider a linear layer with weights W. Its output is Wx. If we scale the weights by a positive constant c, the output becomes $c(Wx)$. When this output is fed into a BN layer, the normalization step will

calculate a mean proportional to c and a standard deviation proportional to c. The normalization step, $\frac{c(Wx)-c\mu}{c\sigma}$, cancels out the scaling factor c. The final output is unchanged.

This means that the function computed by the network is invariant to the norm of the weight matrices preceding a BN layer. This turns out to place strict constraints on how other common techniques, like L_2 regularization (Weight Decay), operate. Normally, weight decay penalizes large weights, pulling their norms towards zero to prevent overfitting. But in a BN network, shrinking the norm of W to $0.5W$ has no effect on the final output, because the BN layer will rescale it back.

So what does weight decay actually do? A rigorous analysis [2] reveals a surprising equivalence: training a BN network with a fixed learning rate η and weight decay λ is equivalent to training the same network *without* weight decay, but with an **exponentially increasing learning rate**.

The intuition is that weight decay tries to decrease the norm of the weights at each step. To preserve the function it has learned (due to scale invariance), the optimization process must compensate by taking larger steps in the direction of the gradient. This implicit dynamic creates an effective learning rate that grows over time. This insight helps explain why BN is so robust to hyperparameter choices and why it enables such stable and rapid training. It shows that the elements of deep learning optimization—normalization, regularization, and learning rate schedules—are deeply intertwined, and their interactions can lead to emergent behaviors that are very different from how they behave in isolation. This is a prime example of how developing mathematical insight can reveal the hidden mechanics behind our most successful empirical methods.

11.2.4 A Deeper Dive into the Mathematics of Scale Invariance

The smoothing effect of Batch Normalization is a consequence of a deeper mathematical property: scale invariance. Understanding this property not only explains why BN works but also reveals a surprising connection between normalization, regularization, and the learning rate schedule itself.

Let's consider a single linear layer with weights W and input x, followed by a Batch Normalization layer. The output of the linear layer is the pre-activation $z = Wx$. The BN layer then normalizes these pre-activations.

Now, consider scaling the weights by some positive constant $c > 0$, resulting in a new weight matrix $W' = cW$. The new pre-activations are $z' = W'x = c(Wx) = cz$. When these are fed into the BN layer

over a mini-batch B, the statistics are:

$$\mu'_B = \frac{1}{m}\sum_{i\in B} z'_i = \frac{1}{m}\sum_{i\in B} cz_i = c\mu_B$$
$$(\sigma'_B)^2 = \frac{1}{m}\sum_{i\in B}(z'_i - \mu'_B)^2 = \frac{1}{m}\sum_{i\in B}(cz_i - c\mu_B)^2 = c^2\sigma_B^2$$

The normalized activation is therefore:

$$\widehat{z}'_i = \frac{z'_i - \mu'_B}{\sqrt{(\sigma'_B)^2 + \epsilon}} = \frac{cz_i - c\mu_B}{\sqrt{c^2\sigma_B^2 + \epsilon}}$$

For a sufficiently large c such that $c^2\sigma_B^2 \gg \epsilon$, this simplifies to:

$$\widehat{z}'_i \approx \frac{c(z_i - \mu_B)}{\sqrt{c^2\sigma_B^2}} = \frac{c(z_i - \mu_B)}{c\sigma_B} = \frac{z_i - \mu_B}{\sigma_B} = \widehat{z}_i$$

The normalized output is identical. Since the subsequent learnable scale and shift parameters γ and β are independent of the weights W, the final output of the BN layer remains unchanged. Thus, the function computed by the '(Linear + BN)' block is invariant to the norm of the weight matrix W.

Problem 11.2.2. *The proof of scale invariance for BN relies on the fact that scaling the input to the normalization layer, $z' = cz$, also scales the mean and standard deviation by c. This works for a linear layer. Now consider a (ReLU + Linear + BN) block. The input to the linear layer is now $ReLU(x)$. If we scale the weights of the layer before the ReLU, does the scale invariance property still hold? Why or why not?*

Geometric Consequence 1: The Gradient is Orthogonal to the Weights. This scale invariance has a powerful geometric consequence. If the loss $L(W)$ does not change when we scale W, it means that moving in the "radial" direction of W in the parameter space does not change the loss. The directional derivative of the loss in the direction of W must be zero.

The directional derivative of L in the direction of a vector v is given by $\langle \nabla_W L, v\rangle / \|v\|$. Setting $v = W$, we have:

$$\text{Directional Derivative along } W = \frac{\langle \nabla_W L(W), W\rangle}{\|W\|} = 0$$

This implies that the inner product of the gradient and the weight vector itself must be zero:

$$\langle \nabla_W L(W), W\rangle = 0$$

The gradient is always orthogonal to the weight vector. This is a rigid constraint on the direction of the updates. Intuitively, any component

of the gradient that was parallel to W would only serve to change its norm, but since the norm doesn't affect the output, the optimization process learns to produce gradients with no such component.

Geometric Consequence 2: The Norm of the Weights Increases During Training. The orthogonality property directly explains why weights in BN-equipped networks tend to grow in magnitude during training (if not regularized). Consider a standard SGD update without weight decay:

$$W_{t+1} = W_t - \eta \nabla_W L(W_t)$$

Let's analyze the squared norm of the updated weights:

$$\begin{aligned} \|W_{t+1}\|^2 &= \|W_t - \eta \nabla_W L(W_t)\|^2 \\ &= \langle W_t - \eta \nabla_W L(W_t), W_t - \eta \nabla_W L(W_t) \rangle \\ &= \|W_t\|^2 - 2\eta \langle W_t, \nabla_W L(W_t) \rangle + \eta^2 \|\nabla_W L(W_t)\|^2 \end{aligned}$$

Because of the orthogonality property, the cross-term $\langle W_t, \nabla_W L(W_t) \rangle$ is zero. The update equation for the norm simplifies to:

$$\|W_{t+1}\|^2 = \|W_t\|^2 + \eta^2 \|\nabla_W L(W_t)\|^2$$

This shows that the norm of the weights is a non-decreasing quantity. At every step, the update moves the weight vector in a direction perpendicular to itself, which, by the Pythagorean theorem, necessarily increases its distance from the origin. This leads to the ever-growing weights observed in practice and highlights the need for some form of regularization.

The Main Result: Weight Decay as an Exponential Learning Rate Schedule. Now, let's introduce L_2 regularization, or weight decay. The update rule becomes:

$$W_{t+1} = W_t - \eta \left(\nabla_W L(W_t) + \lambda W_t \right) = (1 - \eta\lambda) W_t - \eta \nabla_W L(W_t)$$

This update has two components: a "decay" step that shrinks the weights by a factor of $(1 - \eta\lambda)$, and a gradient step. The decay step counteracts the norm explosion derived above. But what is the combined effect?

The analysis from [3] reveals the true dynamic. Let's reason about the update step by step. [3]

1. At step t, weight decay first proposes to shrink the weights to $W'_t = (1 - \eta\lambda) W_t$.
2. This shrinking has *no effect* on the function computed by the network due to scale invariance.

3. However, this scaling *does* affect the gradient. Because $\nabla_W L(cW) = \frac{1}{c}\nabla_W L(W)$, the gradient at the shrunken weights W_t' is larger than the gradient at the original weights W_t:

$$\nabla_W L(W_t') = \frac{1}{1-\eta\lambda}\nabla_W L(W_t)$$

4. The SGD update is then applied using this amplified gradient.

This means that weight decay is not truly acting as a regularizer in the traditional sense of penalizing complexity by keeping weights small. Instead, it creates an internal dynamic that effectively **increases the learning rate**.

More formally, let's track the "direction" of the weights, represented by the unit vector $u_t = W_t/\|W_t\|$, and the "magnitude" $\|W_t\|$. The weight decay term $(1-\eta\lambda)$ only affects the magnitude, while the gradient $\nabla_W L(W_t)$ is orthogonal to W_t and thus primarily affects the direction u_t. The effective size of the update to the direction of the weights is controlled by what is called the **effective learning rate**, $\eta_{\text{eff}} = \eta/\|W_t\|^2$.

When both BN and weight decay are used, the system settles into an equilibrium. The weight decay term shrinks the norm, which in turn increases the gradient magnitude, leading to a more aggressive update. This larger update causes the norm to grow again via the Pythagorean effect. This feedback loop stabilizes, but the net result is that the optimization proceeds as if it had a much larger learning rate.

The paper proves a precise equivalence: the training trajectory of a network with BN, weight decay λ, and learning rate η is the same (in function space) as a network with BN, *no* weight decay, and a learning rate schedule $\widetilde{\eta}_t$ that increases exponentially at a rate related to λ:

$$\widetilde{\eta}_t \approx (1-\eta\lambda)^{-2t}\eta_0$$

This result shows that Batch Normalization fundamentally changes the role of weight decay from a regularizer into a mechanism for implicitly tuning the learning rate upwards. This helps understand the remarkable stability of BN networks during training.

11.3 ResNets: Creating a Superhighway for the Gradient

Batch Normalization provided a powerful tool for stabilizing the training of deep networks, but the degradation problem hinted at a more fundamental issue. The core challenge remained: how can we ensure that a deeper model can represent at least the same functions as a shallower one? If a 20-layer model is optimal, a 56-layer model

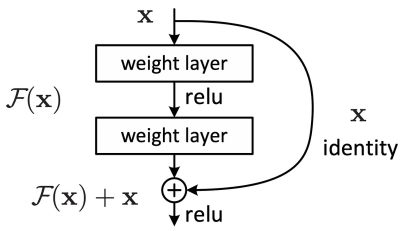
ought to be able to learn to replicate those first 20 layers and then pass the signal through the remaining 36 layers untouched. The fact that optimizers struggled to even learn this simple *identity mapping* suggested that our network architectures were inherently difficult to optimize.

The breakthrough came from a brilliantly simple architectural change proposed in the **Residual Network (ResNet)** [4], which allowed easy training of with dozens or even hundreds of layers, and with fewer total parameters than previous shallower architectures. Instead of forcing a stack of layers to learn a complex transformation $H(x)$ from scratch, ResNets reformulate the problem.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016

11.3.1 *Learning the Residual*

Figure 11.2: Skip Connection



A standard "plain" network block takes an input x from the previous layer and applies a series of convolutions, activations, etc., to produce an output $H(x)$. A residual block does something different. It computes a non-linear transformation $F(x)$ (e.g., two convolutional layers) and then adds its input x back to the output via a "shortcut" or "skip connection."

$$H(x) = F(x, \{W_i\}) + x$$

This small change has a profound implication. The network is no longer learning the target function $H(x)$; it is learning the **residual function** $F(x) = H(x) - x$. It is learning *what to change* about the input x that it received from the previous layer.

This elegantly solves the identity mapping problem. If the optimal transformation for a given block is the identity (i.e., $H(x) = x$), the network can easily achieve this by learning to set the weights in the layers of $F(x)$ to zero. Driving weights to zero is a much easier task for an optimizer than learning a perfect identity transformation through a complex non-linear stack. The network now has a "zero-effort" path to maintaining the information from the layers below. It

only needs to expend its capacity on learning how to *improve* upon the identity.

11.3.2 The Uninterrupted Flow of the Gradient

The true power of the residual connection is revealed when we examine its effect on backpropagation. The skip connection creates a direct, uninterrupted path for the gradient to flow from deeper layers to shallower ones.

Let's consider the gradient of the loss L with respect to the input of a residual block, x_l. The output of this block is $x_{l+1} = F(x_l, W_l) + x_l$. Using the chain rule, the gradient is:

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_{l+1}} \frac{\partial x_{l+1}}{\partial x_l}$$

Now, let's compute the local derivative $\frac{\partial x_{l+1}}{\partial x_l}$:

$$\frac{\partial x_{l+1}}{\partial x_l} = \frac{\partial}{\partial x_l}(F(x_l, W_l) + x_l) = \frac{\partial F(x_l, W_l)}{\partial x_l} + 1$$

Substituting this back into the chain rule gives us the key equation for gradient flow in a ResNet:

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial x_{l+1}} \left(1 + \frac{\partial F(x_l, W_l)}{\partial x_l}\right) \tag{11.1}$$

The crucial term here is the '$+ 1$'. It ensures that the gradient from a deeper layer, $\frac{\partial L}{\partial x_{l+1}}$, has a direct path back to layer l. Even if the weights in the residual branch $F(x_l)$ are small and their derivative $\frac{\partial F}{\partial x_l}$ approaches zero (which can happen during vanishing gradients), the '$+ 1$' term guarantees that the gradient signal can still pass through the block unmodified.

If we unroll this recursion over the entire network, the gradient at a very shallow layer k receives a signal from a very deep layer L:

$$\frac{\partial L}{\partial x_k} = \frac{\partial L}{\partial x_L} \prod_{i=k}^{L-1} \left(1 + \frac{\partial F(x_i, W_i)}{\partial x_i}\right)$$

Unlike a plain network, where the gradient is a product of many matrix multiplications that can shrink the signal to zero, the ResNet gradient is a sum of terms. The expanded product above contains a term $\frac{\partial L}{\partial x_L} \cdot 1 \cdot 1 \cdots 1$, which represents the gradient flowing cleanly through the identity path. This "gradient superhighway" ensures that all layers receive a meaningful training signal, effectively mitigating the vanishing gradient problem from an architectural standpoint.

Problem 11.3.1 (Interplay of BN and ResNet)**.** *Consider a standard ResNet block defined by the equation $H(x) = x + F(x)$. The residual*

function, $F(x)$, is composed of a sequence of layers. Let's define it as:

$$F(x) = Conv_2(ReLU(BN_1(Conv_1(x))))$$

Let the weights of the first convolutional layer be W_1. As we established in the chapter, the output of the BN_1 layer is invariant to scaling the weights W_1 (i.e., replacing W_1 with cW_1 for some scalar $c > 0$).

Does this property imply that the final output of the entire block, $H(x)$, is also invariant to scaling the weights W_1? Explain your reasoning by tracing how the scaling of W_1 propagates through the output of BN_1, then through the rest of the residual function $F(x)$, and finally affects the sum in $H(x)$.

11.4 *Fixup Initialization: Isolating the Architectural Contribution*

The original ResNet architecture was presented as a package deal: residual connections plus Batch Normalization in every block. This combination was incredibly effective, but it left a crucial question unanswered: was the success due to the residual architecture itself, or was BN doing most of the heavy lifting? Could a deep ResNet be trained without any normalization layers?

Simply removing BN from a standard ResNet causes training to fail. The reason lies in the signal variance. The output of a residual block is $x_{l+1} = x_l + F(x_l)$. At initialization, the weights are typically drawn from a distribution[5] designed to preserve the variance of the signal through a single layer. However, in a residual block, the variances add. Assuming the input x_l and the residual output $F(x_l)$ are uncorrelated, the variance of the block's output is:

$$\text{Var}(x_{l+1}) = \text{Var}(x_l) + \text{Var}(F(x_l))$$

Even if $\text{Var}(F(x_l))$ is small, this sum is strictly greater than $\text{Var}(x_l)$. As the signal passes through hundreds or thousands of such blocks, its variance explodes, leading to numerical instability and making training impossible.

The authors of **Fixup Initialization**[6] diagnosed this problem and proposed a solution that addresses it without any normalization layers. Their insight was to design an initialization scheme that ensures the network behaves as a near-identity function at the beginning of training, thus preventing the variance from exploding. This is achieved through a few precise rules:

1. **Initialize most layers conventionally.** All weight layers except for the last layer of each residual branch and the final classification layer are initialized using a standard method like He initialization. All bias layers are initialized to zero.

[5] A frequent choice is *He initialization*: if a node has d incoming edges, each edge is initialized via $\mathcal{N}(0, 2/\sqrt{d})$.

[6] Fixup Initialization: Residual Learning without Normalization. Zhang, Y. Dauphin, T. Ma. ICLR 2019.

2. **Downscale the residual branch output.** The primary cause of the variance explosion, $\text{Var}(F(x_l))$, is controlled by setting the weights of the *final layer* in each residual branch to zero at initialization. This forces $F(x_l) = 0$ at the start of training, ensuring that $\text{Var}(x_{l+1}) = \text{Var}(x_l)$ and the signal propagates perfectly. To allow for learning, instead of exact zeros, the weights are drawn from a distribution whose standard deviation is scaled by a factor of $L^{-1/(2m-2)}$, where L is the number of residual blocks in the network and m is the number of layers in a block. This carefully chosen factor, which depends on the total network depth, ensures that the contribution of the residual branches is appropriately small at initialization but non-zero, allowing training to commence.

3. **Introduce learnable scalar multipliers.** This is the key to providing the dynamic re-scaling that Batch Normalization offers. Fixup introduces two learnable scalar parameters per block (initialized to 1): one just before the residual branch, and one just before the final addition.

$$H(x) = x + \text{bias}_2 + \text{scale}_2 \cdot F(\text{scale}_1 \cdot x + \text{bias}_1)$$

These scalars allow the network to learn the appropriate magnitude for the signal flowing through both the identity path and the residual path. The scalar before the addition, in particular, plays a role analogous to BN's learnable γ parameter, letting the optimizer determine the appropriate contribution of the residual update at each block.

Problem 11.4.1. *The Fixup paper addresses the variance explosion where $Var(x_{l+1}) = Var(x_l) + Var(F(x_l))$. Consider an alternative residual block defined as $x_{l+1} = \frac{1}{\sqrt{2}}(x_l + F(x_l))$. Assuming x_l and $F(x_l)$ are uncorrelated and He initialization ensures $Var(F(x_l)) \approx Var(x_l)$, would this architecture solve the variance explosion problem? What might be a potential downside of this formulation compared to the standard ResNet block?*

The success of Fixup is a powerful demonstration. It proves that Batch Normalization, while immensely helpful, is not fundamental to why ResNets work. The core principle is the residual connection. BN is one (very effective) way to manage the signal dynamics within that architecture. Fixup is another.

Hopefully, this chapter has shown you the rich interplay between "architectural principle" and an "optimization aid"!