Weight Initialization in Deep Neural Networks

Florian Fröhlich

Bachelor Thesis

University of Osnabrück

Institute of Cognitive Science

First supervisor: MSc Leon René Sütfeld
Second supervisor: Prof. Dr. Gordon Pipa

Abstract

A neural network's *initial weights* dramatically impact its ability to learn. Chosen without care, they may cause gradients and activations to explode or vanish. The dynamics of activation and gradient flow depend on the choice of activation function, and weight initialization schemes specifically tuned to the activation function exist for linear and rectifier-based networks. Since the introduction of rectifiers, however, a wide range of alternative non-linearities has been introduced, and corresponding initialization schemes did not yet follow. Analogous methods have also not been proposed, thus far, for the primordial activation functions *tanh* and *sigmoid*. We attempt to derive such methods by adapting the existing identity- and rectifier-specific schemes to other activation functions. Failing at that for reasons that will be explained, we introduce several algorithms for finding suitable initial weights that can accommodate *any* activation function. We test these proposed algorithms on a range of models, obtaining, however, rather underwhelming results. We conjecture that these poor results are partially explained by the algorithms' disregarding other layer types, like Max-Pooling and Dropout, and therefore change course.

Successes have recently been achieved with *data-driven* weight initialization schemes that re-scale weights based on empirically obtained activation statistics rather than theoretical predictions. These methods can also countervail the influence of non-activation-layers on activation variances, and are suited for activation functions for which the above-mentioned methods fail, such as Maxout. One such *data-driven* method is *layer-sequential unit-variance* (LSUV) initialization which normalizes activations at each layer but disregards gradients. We show how LSUV can be adapted to preserve gradient variances, how one might compromise between preserving gradient and pre-activation variances, and, finally, how LSUV could be revised to stabilize variances of *weight gradients*. The latter two methods show promise in our experiments, performing better than regular LSUV.

Contents

Chapter 1

Introduction

Over the past few years, the field of deep learning has seen remarkable breakthroughs across a variety of domains, with successes in computer vision, natural language processing, and game-playing.

In vision, convolutional neural networks (CNNs; LeCun et al., 1989) have achieved recognition accuracies beyond the human benchmark (He et al., 2015), while in speech, hidden Markov models (HMMs; Huang et al., 1990), for long the leading paradigm, have lost their eminence to recurrent neural networks (RNNs) (Graves et al., 2013). More recently, deep nets have been used to estimate the value function in a reinforcement learning setting, notably yielding an agent that achieved superhuman performance in a range of Atari games (Mnih et al., 2015).

This progress is owing in part to advances in hardware that allow for deeper models to be trained in reasonable time. Many real world tasks are compositional in their nature, thus lending themselves naturally to hierarchical representations where lower-level features are combined to form more abstract ones at a higher-level of the pyramid. Under certain conditions, it can be proven that deeper nets are more expressive than their shallower counterparts, meaning they can represent more patterns using the same number of units (Telgarsky, 2016; Raghu et al., 2016; Lin et al., 2017; Poole et al., 2016; Sharir and Shashua, 2017; Eldan and Shamir, 2016). It comes as no surprise, then, that in a rough trend, added depth often goes along with improved accuracy.

Meanwhile, it has long been appreciated that, for all its benefits, greater depth makes training the network significantly more difficult (Glorot and Bengio, 2010; Bengio et al., 1994; Erhan et al., 2009; Sutskever et al., 2013). Various explanations have been ventured as to why this is the case, such as saturating nonlinearities, the presence of local minima, the presence of saddle-points in high-error plateaus, and exploding or vanishing gradients.

The latter phenomenon, described by Hochreiter in the context of RNNs (Hochreiter, 1991) but seen as well in feed-forward neural networks, refers to the exponential growth or decay of the back-propagated gradients as one moves further away from the output layer. This can slow down and effectively halt learning at a poor solution.

A second spur of breakthroughs has thus consisted in architectural and algorithmic innovations that address this problem.

One such innovation has been today's most popular activation function, the rectified linear unit (ReLU; Maas et al., 2013; Nair and Hinton, 2010). Compared to their sigmoidal counterparts, rectifier neurons do not suffer as much from vanishing gradients due to their non-contractive derivative of 1 in the positive domain and, surprisingly, despite the hard saturation for negative arguments (likely because gradients can still travel along paths of active neurons).

Optimization methods like AdaGrad (Duchi et al., 2011) mitigate the inconsistency of gradients across layers to some degree as they adjust the step size for each parameter according to a history of partial derivatives. In the presence of vanishing or exploding gradients there is no one step size that can accommodate gradients both at lower and at higher layers.

Batch normalization (Ioffe and Szegedy, 2015) shifts and scales layer outputs such that they have zero mean and unit variance, thereby preventing saturation of units in the following layer, and so, to some degree, vanishing gradients. In a similar vein, self-normalizing neural networks (SNNs; Klambauer et al., 2017) employ a special activation function, scaled exponential linear units (selu), for which activations are guaranteed to converge to zero mean and unit variance as they are propagated through the network. They hence avoid vanishing gradients in the same manner as batch normalization does but without explicit normalization. Further related approaches are weight normalization (Salimans and Kingma, 2016), which first scales weights in a data-dependent fashion such that activations have zero mean and unit variance[1], and subsequently decouples the weight vectors' direction and length such that the initial scaling is approximately preserved during training; and layer normalization (Ba et al., 2016), which, like Batch Normalization, normalizes activations but uses layer- instead of batch-statistics.

Residual Networks, specifically in their latest incarnation sporting identity mappings (He et al., 2016a;b), allow for gradients to pass directly through skip connections, bypassing any weight layers upstream such that vanishing

[1] This step is almost identical to Mishkin and Matas's LSUV (see below); the two methods were developed concurrently.

gradients are unlikely to occur[1]. Before that, Highway Networks (Srivastava et al., 2015) introduced gated shortcuts from earlier layers to later ones (and from later to earlier ones from the perspective of gradients). (Huang et al., 2017) escalate this course and present DenseNet, a convolutional neural network that features connections between any two layers that agree in feature-map size.

Concurrent with the approaches named thus far, important advances in training very deep neural networks have been spawned by considerations of *how to sensibly choose the network's initial weights*. Importantly, the question retains its relevance even in the light of all the above ideas — a poor initialization will stifle learning in a network armed with any of the named contrivances.

In its most basic function, the initialization should introduce randomness to break the symmetry between units and allow them to learn different things. It is therefore the standard to sample the initial weights from a normal or uniform distribution.

(Krizhevsky et al., 2012) initialized the weights of their ILSVRC-2012-classification-challenge-winning CNN with values sampled from a normal distribution with zero mean and variance 10^{-4}, uniformly for all eight weight-layers.

Such uniform initialization fails, however, for deeper models, which is why (Simonyan and Zisserman, 2014) pre-trained a smaller network of 11 weight-layers and afterwards added intermediate weight-layers to obtain VGG16 and VGG19 with 16 and 19 layers, respectively[2].

This strategy is reminiscent of older methods that split training into multiple stages to overcome the difficulties associated with training deep networks. Supervised and unsupervised pre-training (Erhan et al., 2010; Bengio et al., 2007; Hinton et al., 2006), for example, first train the network layer-by-layer according to an intermediate target in hope that this initializes the weights near a good local minimum.

The problem making uniform initialization unsuited for the *end-to-end* training of very deep models lies in the malignant scaling both forward- and backward-propagated signals can experience at each layer if initialization variances are not informed by layer-sizes and properties of the activation function. Repeated at every layer, the effect of scaling is exponential in network depth. A scaling factor smaller than one will see vanishing gradients as referenced above while scaling factors greater than one cause the

[1]Note that He et al. do not attribute the success of their residual architecture over the plain one to its rectifying pathological gradient flow; their plain network uses batch norm and they confirm the backward propagated gradients exhibit healthy norms.

[2]Weights for the smaller model as well as for the later added weight-layers were initialized with values from a zero-mean normal distribution with variance 10^{-2}.

gradients to explode. Correspondingly, the forward signal can suffer from incorrect scaling, leading, e.g., to overly linear activations, saturated units, or numerical problems.

(Glorot and Bengio, 2010) consider the variance of pre-activations and gradients at each layer and derive an initialization scheme that adapts weight initialization variances to layer sizes, commonly known as *Xavier* initialization. It keeps variances of pre-activations and gradients approximately constant across layers. Their derivation ignores the non-linearity but yields good results in many non-linear settings, nonetheless.

For very deep networks, though, the incorrect assumption becomes a problem. (He et al., 2015) adapt Glorot and Bengio's derivation for the ReLU activation. ReLU halves the variance of the pre-activation; ignoring this leads to a quickly diminishing forward-signal. Equivalently, using Glorot and Bengio's formula in combination ReLU will lead to a halving of the activation-gradient's variance at each layer. With their ReLU-specific initialization, He et al. are able to train a 30-layer CNN that stalls when initialized with *Xavier*. (This ReLU-specific initialization is officially called *MSRA*).

So far, to the best of our knowledge, there have been no attempts to generalize Glorot and Bengio's derivations to further activation functions[1].

Before (He et al., 2015), (Sussillo and Abbott, 2014) introduced Random Walk Initialization, which consists in a theoretically derived scaling scheme of the initial weight matrices that will effect the log-norm of the error gradient to take an unbiased random walk as it is propagated from layer to layer.

(Mishkin and Matas, 2015) dispense with theoretical derivations in favor of a *data-driven* approach that echoes Batch Normalization: they repeatedly pass mini-batches through the pre-initialized network, measure output variances at each layer, and scale weight matrices such that each layer's output achieves unit variance. This approach, which they call *layer-sequential unit-variance* (LSUV) initialization, is compatible with any activation function and can redress the influence of other layer-types, like Dropout (Srivastava et al., 2014), on activation variances, which *Xavier* and *MSRA* can not. Despite its simplicity, LSUV obtains near state-of-the-art results on MNIST, CIFAR, and ImageNet, outperforming more complex initialization schemes. While LSUV does not explicitly consider the back-propagated signal, it will also lead to relatively healthy gradients in the initial setting, since, like Batch Normalization, it avoids saturation of units in the following layer.

Shortly after (Mishkin and Matas, 2015), (Krähenbühl et al., 2015) introduced another data-dependent initialization that includes LSUV as a first step but

[1] (Hendrycks and Gimpel, 2016) present an initialization scheme that can deal with any activation function and that adjusts for the influence of Dropout. For reasons that will become apparent, our meditations converge in a conceptually similar approach.

then, in a second step, re-scales weights to yield constant parameter change rates throughout the network, that is, the ratio of expected weight gradient norm, on the one hand, and weight magnitude, on the other hand, is approximately preserved across layers.

Illustrated by the miscellany of methods, there is no definite agreement as to the precise goal a weight initialization should realize, emblematic of the oft-noted disconnect between theoretical understanding and empirical success seen in deep learning. *Xavier* and *MSRA* are heuristics. Further investigation to that effect might prove fruitful, as we will touch upon in the discussion.

In this thesis, we first attempt to generalize findings from (Glorot and Bengio, 2010) and (He et al., 2015) to the activation functions *elu* (Clevert et al., 2015), *selu*, *tanh*, *swish* (Ramachandran et al., 2018), and *sigmoid*. Given the improvement seen in (He et al., 2015) over (Glorot and Bengio, 2010), we expect these activation-specific initializations to yield an increase in model performance for networks based on the respective activation function, relative to non-specific initializations like *Xavier* and *MSRA*. It will become apparent that finding formulae analogous to (Glorot and Bengio, 2010) for the named activation functions is non-trivial due to some intractable integrals involved. Resorting to numerical integration, we propose several algorithms for obtaining weight initialization variances that preserve variances of pre-activations and gradients throughout the network, and that can deal with *any* activation function. We evaluate the proposed algorithms' performance on four different convolutional neural network architectures and with a variety of activation functions.
We then show how Mishkin and Matas' LSUV can be adapted to preserve gradient variances instead of activation variances. We further introduce a variant of LSUV that compromises between preserving gradient variances and pre-activation variances, and, finally, a variant that is intended to stabilize *weight gradient variances*, following a similar intuition as (Krähenbühl et al., 2015). We evaluate this second set of algorithms in the same manner as the first one.
In the subsequent analysis, we first confirm that the presented algorithms *"do what they are supposed to do"*, i.e., that they preserve activation, gradient, or weight gradient variances throughout the network. Secondly, we attempt to corroborate the intuitions that motivate the different algorithms empirically by establishing a link between variance-consistency and model performance.
Finally, we discuss the observed results, address some shortcomings of our approach, and offer an outlook for further research. We end with a brief conclusion.

Chapter 2

Adapting Xavier & He

Glorot and Bengio study the variance of gradients and pre-activations in a neural network when the activation function is linear. Based on the idea that one would like to allow for information to flow through the network smoothly in either direction (activations flowing forward, gradients flowing backward), they derive a weight initialization scheme that compromises between keeping the variance of gradients and the variance of pre-activations constant across layers. Their initialization scheme, commonly known as *Xavier* initialization, yields good results even when the assumption of a linear activation function is violated (which is virtually always the case). Meanwhile, for very deep networks, *Xavier* can still hinder convergence. He et al. adapt Glorot and Bengio's formulae to accommodate the *ReLU* non-linearity and are able to train a 30-layer convolutional neural network. So far, to our knowledge, there have been no attempts to apply the same ideas to other non-linearities (a possible reason for that will become apparent later).

2.1 Deconstructing Xavier (Derivations)

Glorot and Bengio and He et al. study the variance of a single unit's pre-activation as well as that of the gradient of the error with respect to the unit's activation (across input/target pairs or, synonymously, across units in that layer).

We assume elements in W to be mutually independent and identically distributed. W has zero mean. The elements in $\mathbf{x}^l$ are also mutually independent and identically distributed. $\mathbf{x}^l$ and W^l are independent of each other.

2.1.1 Forward propagation

With y^l, x^l, and w^l representing the random variables of each element in $\mathbf{y}^l$, $\mathbf{x}^l$, and W^l, respectively, we obtain the pre-activation variance of a single

unit at layer l as

$$\mathrm{Var}[y^l] = n^l \mathrm{Var}[w^l]\, \mathbf{E}[f^2(y^{l-1})],$$

where n^l is the size of layer $l-1$ for fully-connected layers and the volume of a convolution kernel for convolutional layers (i.e., $n^l = k_l^2 c_l$, with k_l the width and height of the kernel, and c_l the number channels at layer $l-1$). How one arrives at this formula is detailed in the appendix. Since y^l is a sum of random variables, according to the central limit theorem it approaches a normal distribution. Therefore:

$$\mathbf{E}[f^2(y^l)] = \sqrt{2\pi\sigma^2}^{-1} \int_{-\infty}^{\infty} f^2(z) \exp\left(\frac{-(z-\mu)^2}{2\sigma^2}\right) \mathrm{d}z,$$

with

$$\sigma^2 = \mathrm{Var}[y^l],$$

and

$$\mu = \mathbf{E}[y^l] \; = n^l \, \mathbf{E}[w^l] \, \mathbf{E}[x^l] \; = 0.$$

For simplicity, we define:

$$\mathbf{g}(\sigma^2) := \sqrt{2\pi\sigma^2}^{-1} \int_{-\infty}^{\infty} f^2(z) \exp\left(\frac{-z^2}{2\sigma^2}\right) \mathrm{d}z \tag{2.1}$$

The variance of the pre-activation of a unit at layer l is then given by:

$$\mathrm{Var}[y^l] = n^l \mathrm{Var}[w^l] \mathbf{g}(\mathrm{Var}[y^{l-1}]). \tag{2.2}$$

2.1.2 Backward propagation

For convenience, we denote the gradient of the error with respect to activations and pre-activations as

$$\Delta \mathbf{x}^l = \frac{\partial E}{\partial \mathbf{x}^l},$$

and

$$\Delta \mathbf{y}^l = \frac{\partial E}{\partial \mathbf{y}^l},$$

respectively.

The gradient with respect to the pre-activation is given by

$$\Delta \mathbf{y}^l = f'(\mathbf{y}^l)\Delta \mathbf{x}^{l+1},$$

where

$$\Delta \mathbf{x}^l = \hat{W}^l \Delta \mathbf{y}^l.$$

$\hat{W}^l$ is obtained by rearranging elements in W^l. Its size is $n^{l-1} \times n^l$ for fully-connected layers and $c_l \times k_l^2 c_{l+1}$ for convolutional layers.

Then the variance of the gradient of the error with respect to a unit's activation at layer l is given by

$$\mathrm{Var}[\Delta x^l] = \hat{n}^l \mathrm{Var}\left[w^l\right] \mathbf{E}\left[(f'(y^l))^2\right] \mathrm{Var}\left[\Delta x^{l+1}\right].$$

$\hat{n}^l$ is n^{l+1} for fully-connected layers and $k_l^2 c_{l+1}$ for convolutional layers. How one arrives at this formula is again detailed in the appendix.

For simplicity, once again, we define:

$$\mathbf{h}(\sigma^2) := \sqrt{2\pi\sigma^2}^{-1} \int_{-\infty}^{\infty} (f'(z))^2 \exp\left(\frac{-z^2}{2\sigma^2}\right) \mathrm{d}z \tag{2.3}$$

Then the variance of the gradient of the error with respect to a unit's activation at layer l is given by:

$$\mathrm{Var}[\Delta x^l] = \hat{n}^l \mathrm{Var}[w^l]\mathbf{h}(\mathrm{Var}[y^l])\mathrm{Var}[\Delta x^{l+1}] \tag{2.4}$$

2.2 Applied to Activation Functions

2.2.1 Applied to *Identity*

Applying above formulae to the *identity* activation function yields

$$\mathbf{g}_{\text{ident}}(\sigma^2) = \sqrt{2\pi\sigma^2}^{-1} \int_{-\infty}^{\infty} z^2 \exp\left(\frac{-z^2}{2\sigma^2}\right) \mathrm{d}z \; = \sigma^2$$

and

$$\mathbf{h}_{\text{ident}}(\sigma^2) = \sqrt{2\pi\sigma^2}^{-1} \int_{-\infty}^{\infty} 1^2 \exp\left(\frac{-z^2}{2\sigma^2}\right) \mathrm{d}z \; = 1.$$

Inserting this into (2.2) and (2.4) we get

$$\mathrm{Var}[y^l] = n^l \mathrm{Var}[w^l]\mathrm{Var}[y^{l-1}]$$

and

$$\mathrm{Var}[\Delta x^l] = \hat{n}^l \mathrm{Var}[w^l]\mathrm{Var}[\Delta x^{l+1}].$$

Therefore, to preserve forward variances, we should set

$$\mathrm{Var}[w^l] = \frac{1}{n^l}. \tag{2.5}$$

To preserve backward variances, we should set

$$\mathrm{Var}[w^l] = \frac{1}{\hat{n}^l}. \tag{2.6}$$

Glorot and Bengio opt for the harmonic mean between (2.5) and (2.6) to compromise between preserving forward and backward variances:

$$\mathrm{Var}[w^l] = \frac{2}{\hat{n}^l + n^l}. \tag{2.7}$$

2.2.2 Applied to *ReLU*

For the *ReLU* activation, we get:

$$\mathbf{g}_{\text{relu}}(\sigma^2) = \sqrt{2\pi\sigma^2}^{-1} \int_{-\infty}^{\infty} (\max(0,z))^2 \exp\left(\frac{-z^2}{2\sigma^2}\right) \mathrm{d}z = \frac{1}{2}\sigma^2$$

and

$$\mathbf{h}_{\text{relu}}(\sigma^2) = \sqrt{2\pi\sigma^2}^{-1} \int_{0}^{\infty} 1^2 \exp\left(\frac{-z^2}{2\sigma^2}\right) \mathrm{d}z = \frac{1}{2}.$$

Insert into (2.2) and (2.4) to obtain

$$\mathrm{Var}[y^l] = \frac{1}{2} n^l \mathrm{Var}[w^l] \mathrm{Var}[y^{l-1}]$$

and

$$\mathrm{Var}[\Delta x^l] = \frac{1}{2} \hat{n}^l \mathrm{Var}[w^l] \mathrm{Var}[\Delta x^{l+1}].$$

To keep pre-activation variances constant across layers, then, for the *ReLU* activation we should set

$$\mathrm{Var}[w^l] = \frac{2}{n^l}; \tag{2.8}$$

to keep gradient variances constant, likewise, we should set

$$\mathrm{Var}[w^l] = \frac{2}{\hat{n}^l}. \tag{2.9}$$

He et al. advise to use either of the two equations since for a "typical" convolutional network, (2.8) will also lead to healthy backward variances and (2.9) to healthy forward variances.

2.2.3 Anticlimax: Applied to Other Activation Functions

To the best of our knowledge, thus far, no analogous activation-aware initializations have been introduced for non-linearities other than *ReLU*, such as the *hyperbolic tangent*, *sigmoid*, *swish*, *elu*, or *selu*. A reason for that might be that the integrals in (2.1) and (2.3) are far more involved for other activations and it is not possible to derive a formula as concise as (2.7) or (2.8) and (2.9).

For the *elu* activation, (2.1) and (2.3) are

$$\mathbf{g}_{\text{elu}}(\sigma^2) = \frac{\alpha^2}{2}\left(1 - 2\exp\left(\frac{\sigma^2}{2}\right)\text{erfc}\left(\sqrt{\frac{\sigma^2}{2}}\right) + \exp\left(2\sigma^2\right)\text{erfc}\left(\sqrt{2\sigma^2}\right) + \sigma^2\right)$$

and

$$\mathbf{h}_{\text{elu}}(\sigma^2) = \frac{\alpha^2}{2}\exp\left(2\sigma^2\right)\text{erfc}\left(\sqrt{2\sigma^2}\right) + \frac{1}{2}.$$

The integrals look similar for the closely related *selu* activation.

erfc is the *complementary error function*, defined as

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\mathrm{d}t.$$

It cannot be evaluated in closed form, though many math libraries (numpy, math for Python) include a function to approximate it. Still, the resultant weight initialization formulae are not quite as appealing as the ones for the *identity* and *ReLU*.
Similarly, for activation functions from the *sigmoid family*, like *tanh*, *swish*, and *sigmoid*, no closed form of the integrals in (2.1) and (2.3) can be found as they involve the *logistic-normal integral* for which, to our knowledge, no closed form is known to this day.[1] Approximations, of course, exist.

2.3 Generalizing to Arbitrary Activation Functions

While finding formulae analogous to (2.7) and (2.8) might not be possible or practically useful for all activation functions, we can of course resort to numerical methods to evaluate the integrals in (2.1) and (2.3).

Doing so, we generalize the ideas from (Glorot and Bengio, 2010) and (He et al., 2015) to suit arbitrary activation functions.

[1] Of course, it is difficult to substantiate the claim that an integral is *not* known. As a semblance of evidence, take this article from 2013, published in a reputable mathematics journal: (Pirjol, 2013). They (mathematicians) state: *"The case corresponds to the so-called logistic-normal integral, which appears in statistics in problems of logistic regression. To our knowledge no closed form result for this integral is known."*. *Wolfram Mathematica* does not have the answer either. I can at least say with certainty that I will not find this integral as part of my Bachelor thesis.

For convenience, we introduce the following changes to notation:

$$\begin{aligned} y^i &= \mathrm{Var}[y^i] \\ x^i &= \mathrm{Var}[\Delta x^i] \\ w^i &= \mathrm{Var}[w^i] \\ m^i &= \hat{n}^i. \end{aligned}$$

Further, L denotes the number of layers in our network, excluding layers without trainable weights.

Recall that we have:

$$y^i = n^i w^i g(y^{i-1})$$

and

$$x^i = m^i w^i h(y^i) x^{i+1}.$$

2.3.1 Idea 1: Algorithm 1

Generally, to preserve pre-activation variances across layers, we should set

$$w^i \leftarrow 1 \;/\; (\; n^i g(1) \;)$$

for all layers.

2.3.2 Idea 2: Algorithm 2

To preserve gradient variances, on the other hand, we should set

$$w^i \leftarrow 1 \;/\; (\; m^i h(y^i) \;)$$

for all layers. For *ReLU*, $h(y^i)$ conveniently evaluates to $\frac{1}{2}$ independently of y^i, for the identity to 1. For other activation functions we will have to use fixed-point iteration, since $h(y^i)$ is again a function of w^i:

$G \leftarrow g(1)$
for $i = 1...L$ **do**
  $y \leftarrow 1$
  for $k = 1...K$ **do**
    $w^i \leftarrow 1 \ / \ (\ h(y)\ m^i\ )$
    $y \leftarrow w^i n^i G$
  $G \leftarrow g(y)$

K can be a small number (~ 10).

2.3.3 Idea 3: Algorithm 3

We can adapt this algorithm to compromise between pre-activation and gradient variances using, for example, the harmonic mean.

$G \leftarrow g(1)$
for $i = 1...L$ **do**
  $y \leftarrow 1$
  for $k = 1...K$ **do**
    $w^i \leftarrow 2 \ / \ (\ m^i h(y) + n^i G\ )$
    $y \leftarrow w^i n^i G$
  $G \leftarrow g(y)$

2.3.4 Idea 4: Algorithm 4

We could further factor in the dependency of x^i on x^{i+1} like so:

$G \leftarrow g(1)$

$z \leftarrow 1$

for $i = 1...L$ **do**

  $y \leftarrow 1$

  for $k = 1...K$ **do**

    $w^i \leftarrow 2 \ / \ (\ m^i h(y) z + n^i G \ )$

    $y \leftarrow w^i n^i G$

  $G \leftarrow g(y)$

  $z \leftarrow w^i h(y) m^i z$

2.3.5 Idea 5: Algorithm 5

As a final refinement, we observe that the mean between the weight variance that preserves pre-activation variances and the weight variance that preserves gradient variances will not necessarily lead to pre-activation and gradient variances that are balanced in terms of quality. We therefore introduce a loss function to measure the gradient and pre-activation variance's quality at each iteration so we can adjust weight variances accordingly.

One possible loss function is:

$$\text{loss}(x) = \begin{cases} \frac{1}{x}, & \text{if } x < 1 \\ \exp(x-1), & \text{otherwise.} \end{cases}$$

Then Algorithm 5 could look like this:

$G \leftarrow g(1)$

$z \leftarrow 1$

for $i = 1...L$ **do**

  $y \leftarrow 1$

  for $k = 1...K$ **do**

    $y \leftarrow w^i n^i G$

    $x \leftarrow w^i m^i h(y) z$

    $E_y \leftarrow \text{loss}(y)$

    $E_x \leftarrow \text{loss}(x)$

    $w^i \leftarrow w^i \ (E_y + E_x) \ / \ (E_y y + E_x x)$

  $G \leftarrow g(y)$

  $z \leftarrow x$

2.4 Evaluation (1)

We evaluate the algorithms proposed in the previous section. To that end, we train four different convolutional neural networks, varying activation function and initialization method, and compare their performance.
Models were implemented in Keras (Chollet et al., 2015) and trained on the CIFAR-10 dataset. It consists of 60,000 32x32 color images from ten categories (6,000 images per category). Per-image z-transformation was applied as pre-processing step; 10,000 images were held out for validation.
All models were trained over 300 epochs on mini-batches of size 128, using the *Adam* optimizer (Kingma and Ba, 2014) with an initial learning rate of 10^{-3}. Biases were initialized to zero.
All convolutional layers used stride 1 and zero padding to preserve height and width of output maps from one convolutional layer to the next ("same" padding in Keras).

Model architectures are shown below (Table 2.4). SMCN and SMCN-10 are taken from (Sütfeld et al., 2018), FitNet-1 and FitNet-4 are taken from (Romero et al., 2014).

We tested the algorithms' performances for the activation functions ReLU, *elu*, *swish*, *tanh*, and *sigmoid*. We also recorded performances for *He* (MSRA)

FitNet-1 250K param	**FitNet-4** 2.5M param	**SMCN** 1.8M param	**SMCN-10** 2M param
conv 3x3x16 conv 3x3x16 conv 3x3x16	conv 3x3x32 conv 3x3x32 conv 3x3x32 conv 3x3x48 conv 3x3x48	conv 5x5x64 dropout .5 conv 3x3x64	conv 5x5x64 dropout .5 conv 3x3x64
pool 2x2	pool 2x2	pool 2x2	pool 2x2
conv 3x3x32 conv 3x3x32 conv 3x3x32	conv 3x3x80 conv 3x3x80 conv 3x3x80 conv 3x3x80 conv 3x3x80	conv 1x1x64 dropout .5 conv 5x5x64	conv 1x1x64 dropout .5 conv 5x5x64 conv 1x1x64 dropout .5 conv 5x5x64 conv 1x1x64 dropout .5 conv 5x5x64
pool 2x2	pool 2x2	pool 2x2	pool 2x2
conv 3x3x48 conv 3x3x48 conv 3x3x64	conv 3x3x128 conv 3x3x128 conv 3x3x128 conv 3x3x128 conv 3x3x128		
pool 8x8 fc-500	pool 8x8 fc-500	fc-384 dropout .5 fc-192	fc-384 dropout .5 fc-192
softmax-10	**softmax-10**	**softmax-10**	**softmax-10**

Table 2.1: Model architectures used in performance comparison. SMCN and SMCN-10 are taken from (Sütfeld et al., 2018), FitNet-1 and FitNet-4 from (Romero et al., 2014).

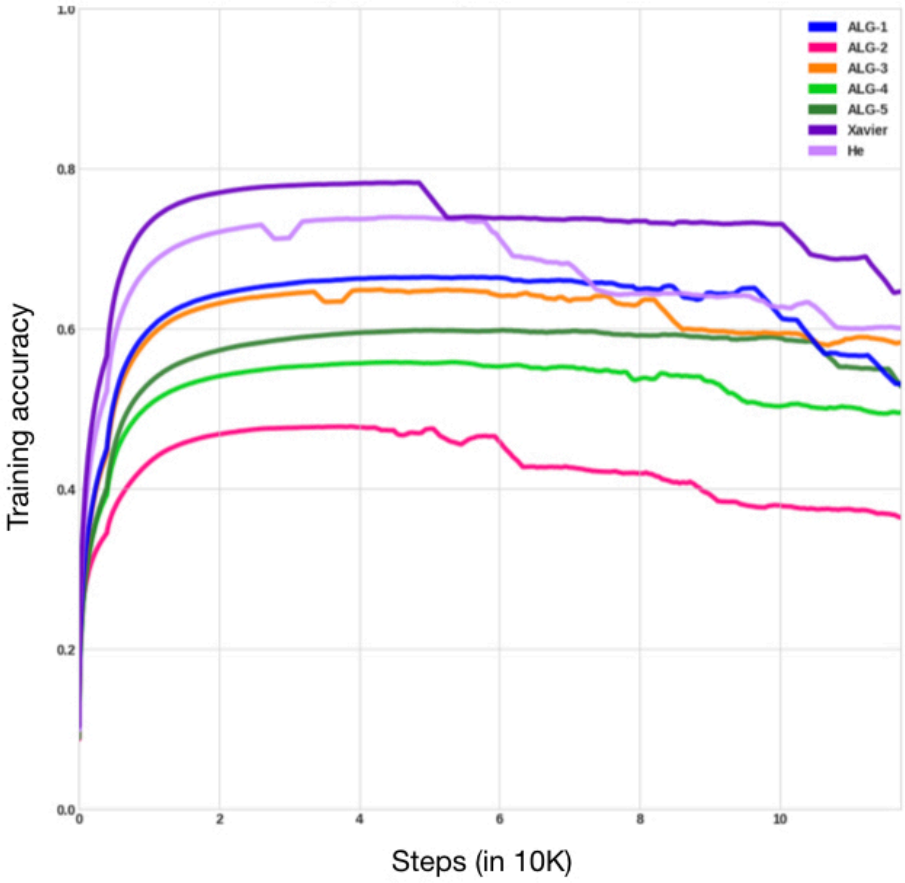

Figure 2.1: Training accuracy per algorithm, averaged over activation functions and models and smoothed over steps. Each line corresponds to one of the algorithms and is the average of 4x5 individual training runs, one run for each model/activation function setting. The y-axis scale is linear. A step consists of one mini-batch update.

and *Xavier* initialization as a baseline. Weights were drawn from a truncated normal distribution (values more than two standard deviations from the mean are discarded and re-drawn) with variances as dictated by the algorithms.

Figures 2.1, 2.2, and 2.3 show smoothed learning curves for each algorithm, averaged over models and activation functions (2.1); for each algorithm and model, averaged over activation functions (2.2); and for each algorithm and activation function, averaged over models (2.3). We can see that, when averaging over models and activations, *Xavier* initialization performs best,

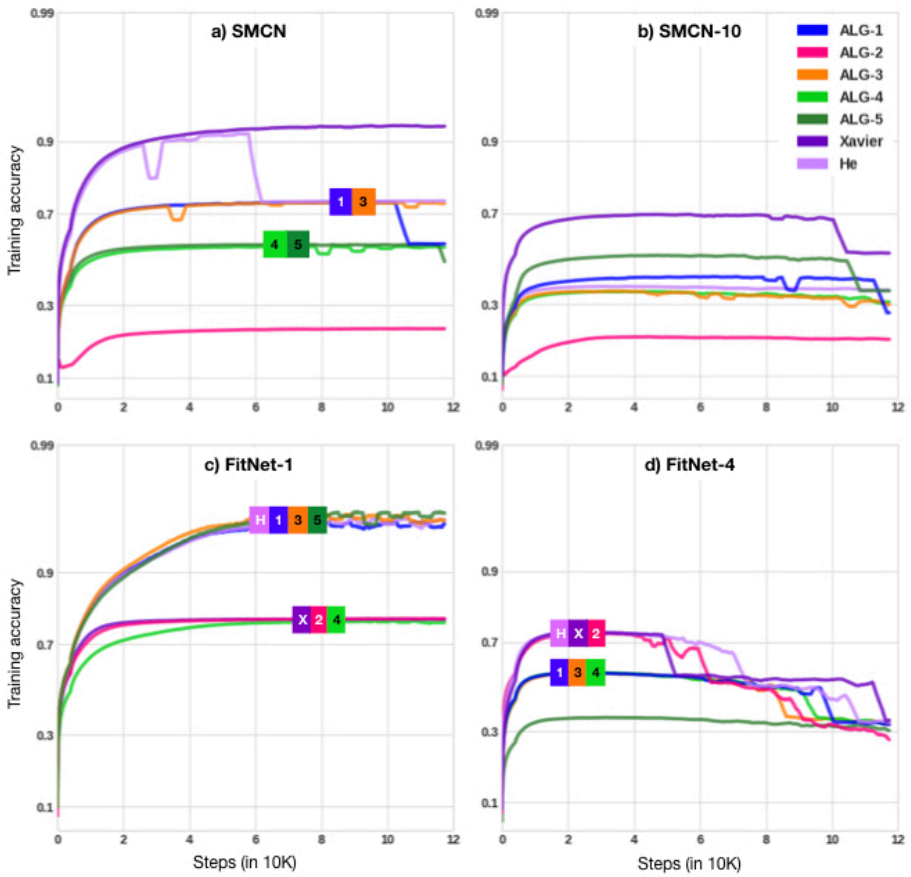

Figure 2.2: Training accuracy per algorithm for each of the four models, averaged over activation functions and smoothed over steps. Each line corresponds to one of the algorithms and is the average of five individual training runs, one run for each activation function. The y-axes are on a logit scale. Lines are labelled where there is strong overlap.

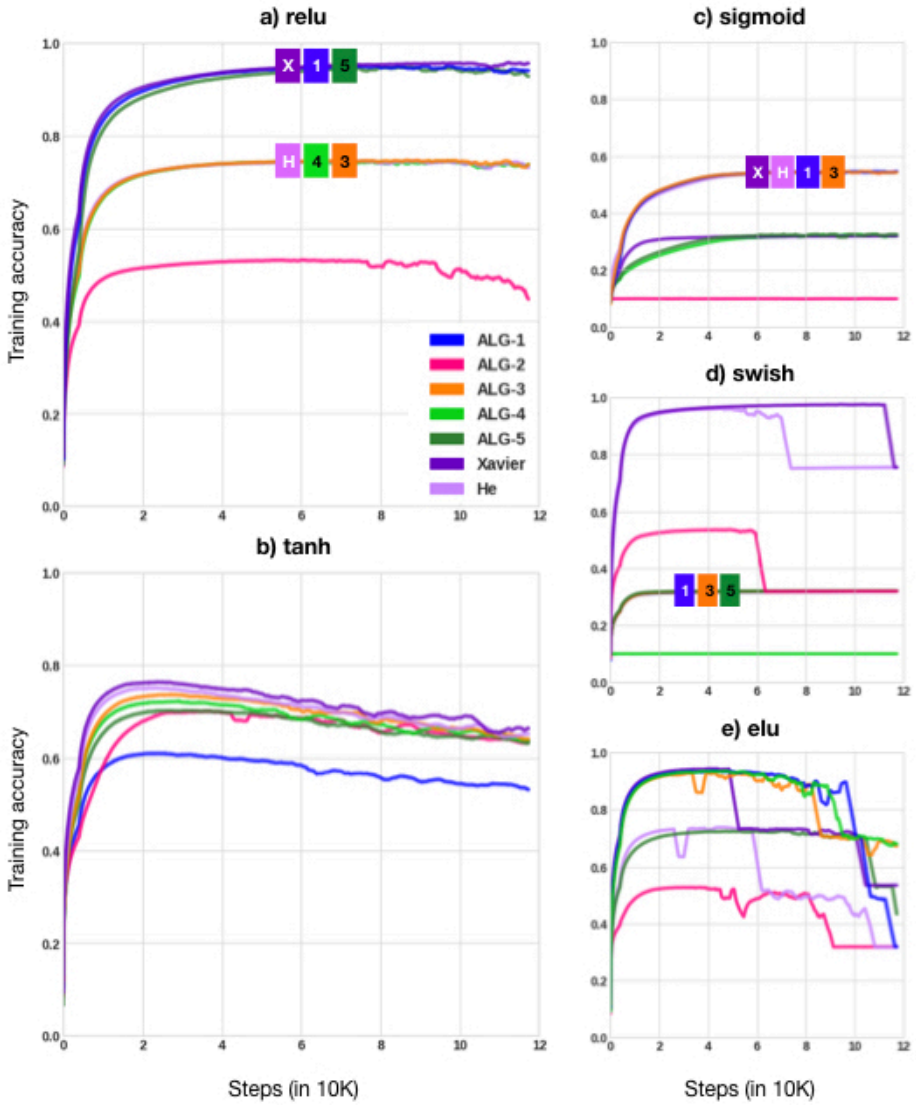

Figure 2.3: Accuracy per algorithm for each of the five activation functions, averaged over models. Each line corresponds to one of the algorithms and is the average of four individual training runs, one for each model. The y-axes are on a linear scale. Lines are labelled where there is strong overlap.

followed by *He* initialization. Looking at the algorithms' performance per model, we see that neither of the proposed algorithms can match *Xavier*'s and *He*'s performance for SMCN; *Xavier* outperforms all other methods by a wide margin for SMCN-10; Algorithms 1, 3, 5 and *He* yield the best average performance for FitNet-1. *Xavier*, *He* and Algorithm 2 perform best for FitNet-4.

In the per-activation comparison, *Xavier* performs at least as well as all other algorithms for all activation functions. For *relu*, Algorithms 1 and 5 perform roughly on par with *Xavier*. For the *sigmoid* activation, *Xavier* is matched by *He* and Algorithms 1 and 3. *Xavier* and *He* perform equally well for *swish* and much better than all other algorithms (though for *He*, learning collapses towards the midpoint of training). For *elu*, Algorithms 1, 3, 4, and *Xavier* perform best.
Overall, the proposed algorithms did not bring the increase in performance one might expect, seeing as they adapt weight variances specifically for each activation function and given the improvement achieved in (He et al., 2015) over (Glorot and Bengio, 2010). In the same vein, it is surprising that *Xavier*, on average, outperformed *He* for the ReLU activation. One has to keep in mind that the algorithms adapt weight variances to the activation function but do not take into account other layer types that affect activation and gradient variances, like Dropout or Max-Pooling, such that an activation-aware initialization might still yield relatively worse forward or backward variances compared to an activation-agnostic initialization in a network that uses these layer types. We address this in the following section.

Chapter 3

Adapting LSUV

The initialization methods discussed in the previous section adjust weight variances so as to preserve variances of pre-activations or gradients across layers. They can do so for any layer type where the same transformation is applied in an element-wise fashion to each unit's pre-activation. This excludes activation functions like Maxout (Goodfellow et al., 2013), and layers like Max-Pooling, Batch Normalization, and Dropout.
While theoretically possible, deriving analogous methods for every conceivable layer type seems impractical.

3.1 Layer-sequential unit-variance initialization

(Mishkin and Matas, 2015) introduce *layer-sequential unit-variance initialization* (LSUV) which dispenses with theoretical derivations in favor of a *data-driven* approach. They repeatedly pass mini-batches through the network, measure output variances at each layer, and re-scale the weights accordingly. The same approach was proposed concurrently by (Salimans and Kingma, 2016) and (Krähenbühl et al., 2015).

pre-initialize network

for each layer **do**

  for $k = 1...K$ **do**

    do forward-pass with mini-batch

    $v \leftarrow \text{Var}[\text{layer-output}]$

    $W \leftarrow W / \sqrt{v}$

W is the weight matrix of the current layer. Following (Saxe et al., 2013), Mishkin and Matas pre-initialize the network with orthogonal unit-variance Gaussian matrices. Instead of the second for-loop, the original algorithm uses a while-loop conditioned on the output variance achieved in the last iteration. They state that the algorithm reached the desired variance in few (<5) iterations in all experiments. They achieve good results across a variety of models, data sets, and activation functions.

3.2 Adapting LSUV

3.2.1 Idea 6: G-LSUV

LSUV normalizes output variances at each layer but does not consider the variance of back-propagated gradients. We adapt LSUV to keep gradient-variances constant.

Our goal is to keep $\text{Var}\left[\frac{\partial E}{\partial \mathbf{y}^l}\right]$ constant across layers. We have:

$$\text{Var}\left[\frac{\partial E}{\partial \mathbf{y}^l}\right] = \text{Var}\left[\frac{\partial E}{\partial \mathbf{y}^L}\frac{\partial \mathbf{y}^L}{\partial \mathbf{y}^{L-1}}\dots\frac{\partial \mathbf{y}^{l+1}}{\partial \mathbf{y}^l}\right]. \tag{3.1}$$

In keeping with deep learning tradition, we deliberately make an incorrect assumption and let the $\frac{\partial \mathbf{y}^{l+1}}{\partial \mathbf{y}^l}$ be independent. We will see that this assumption plays out well in practice. Then, because

$$\mathbf{E}\left[\frac{\partial \mathbf{y}^{l+1}}{\partial \mathbf{y}^l}\right] = \mathbf{E}[W^{l+1}]\mathbf{E}[T'(\mathbf{y}^l)] = 0,$$

where T is whatever transformation is applied to $\mathbf{y}^l$ before it is passed on to the next weight layer (e.g., pooling, an activation function, Dropout), we can write

$$\text{Var}\left[\frac{\partial E}{\partial \mathbf{y}^l}\right] = \mathbf{E}\left[\left(\frac{\partial E}{\partial \mathbf{y}^L}\right)^2\right]\prod_{k=l}^{L-1}\text{Var}\left[\frac{\partial \mathbf{y}^{k+1}}{\partial \mathbf{y}^k}\right]. \tag{3.2}$$

To keep $\text{Var}\left[\frac{\partial E}{\partial \mathbf{y}^l}\right]$ constant across layers, it is sufficient to set

$$\text{Var}\left[\frac{\partial \mathbf{y}^{l+1}}{\partial \mathbf{y}^l}\right] = 1$$

for all layers, or alternatively

$$\text{Var}\left[\frac{\partial \mathbf{y}^l}{\partial \mathbf{y}^1}\right] = 1.$$

Keras offers a function to obtain $\frac{\partial \mathbf{y}^{l+1}}{\partial \mathbf{y}^1}$, and other deep learning libraries likely do as well.

To make notation simpler, let

$$\nabla\text{layer} = \frac{\partial \text{ layer's pre-activation}}{\partial \text{ first layer's pre-activation}}.$$

Then **G-LSUV** could look like this:

pre-initialize network

for each layer **do**

  for $k = 1...K$ **do**

    do forward-pass with mini-batch

    $v \leftarrow \text{Var}[\nabla\text{layer}]$

    $W \leftarrow W/\sqrt{v}$

3.2.2 Idea 7: C-LSUV

Like in the previous section, we might want to strike a compromise between preserving forward and backward variances.

Here goes. First, normalize the first layer's pre-activation (as in regular LSUV, except LSUV is applied *after* the activation):

pre-initialize network

for $k = 1...K$ **do**

  do forward-pass with mini-batch

  $v \leftarrow \text{Var}[\text{first layer's pre-activation}]$

  $W \leftarrow W/\sqrt{v}$

Then, for the following layers, compromise between preserving forward and backward variances, using a loss function like Algorithm 5 does:

for each layer **do**

  for $k = 1...K$ **do**

    do forward-pass with mini-batch

    $v_1 \leftarrow \text{Var}[\nabla \text{layer}]$

    $v_2 \leftarrow \text{Var}[\text{layer's pre-activation}]$

    $\text{loss}_1 \leftarrow \text{loss}\, v_1$

    $\text{loss}_2 \leftarrow \text{loss}\, v_2$

    $W \leftarrow W \cdot (\text{loss}_1 + \text{loss}_2) \;/\; (\text{loss}_1 \sqrt{v_1} + \text{loss}_2 \sqrt{v_2})$

We can choose between preserving activation or *pre*-activation variances, and equivalently, between preserving variances of gradients on activations or on *pre*-activations. The version used in our experiments preserved pre-activation and pre-activation-gradient variances. With an eye towards the *gradients on the weights*, preserving activation variances and pre-activation-gradient variances might be the most sensible choice (see below).

3.2.3 Idea 8: W-LSUV

So far, the discussion has focused on preserving variances of pre-activations and variances of gradients on pre-activations. However, what is intuitively a more crucial quantity is the *variance of the gradients on the weights*. This intuition also motivates (Krähenbühl et al., 2015) but their approach differs from the one we propose below.

We write Δw for $\frac{\partial E}{\partial w}$. The variance of the gradient of the error with respect to weight w is given by

$$\text{Var}[\Delta w^l] = m^l \text{Var}[\Delta y^l]\, \mathbf{E}[(x^l)^2]. \tag{3.3}$$

Here, m^l is the product of width and height of the output map for convolutional layers and 1 for fully-connected layers. We adapt C-LSUV to preserve weight gradient variances and call it **W-LSUV**. The goal is to strike a compromise between setting

$$\mathbf{E}[(x^l)^2] = \frac{1}{m^l}$$

and

$$\text{Var}\left[\frac{\partial \mathbf{y}^l}{\partial \mathbf{y}^1}\right] = 1$$

for each layer.

Before adjusting weight variances:

pre-initialize network

in **Y**, store all layers with trainable weights

in **X**, for each layer in **Y**, store the layer preceding it

in **M**, for each layer in **Y**, store m for that layer

Additionally, it is advisable to scale input images by $\sqrt{\mathbf{M}_1}$. For the first layer:

for $k = 1...K$ **do**

do forward-pass with mini-batch

$v \leftarrow \mathbf{M}_2 \cdot \text{mean}[(\mathbf{X}_2\text{-output})^2]$

$W \leftarrow W / \sqrt{v}$

For the intermediate layers:

for $i = 2...L-1$ **do**

for $k = 1...K$ **do**

do forward-pass with mini-batch

$v_1 \leftarrow \mathbf{M}_{i+1} \cdot \text{mean}[(\mathbf{X}_{i+1}\text{-output})^2]$

$v_2 \leftarrow \text{Var}[\nabla \mathbf{Y}_i]$

$\text{loss}_1 \leftarrow \text{loss}\, v_1$

$\text{loss}_2 \leftarrow \text{loss}\, v_2$

$W \leftarrow W \cdot (\text{loss}_1 + \text{loss}_2) \;/\; (\text{loss}_1 \sqrt{v_1} + \text{loss}_2 \sqrt{v_2})$

For the final layer:

for $k = 1...K$ **do**

do forward-pass with mini-batch

$v \leftarrow \text{Var}[\nabla \mathbf{Y}_L]$

$W \leftarrow W / \sqrt{v}$

3.3 Evaluation (2)

We evaluate the newly proposed algorithms in the same way as we did in section 2.4, using the same models, data set, and training parameters. We only test activation functions *ReLU* and *tanh*. We also record the performance for regular LSUV to allow for comparison. Note that, for all algorithms, we pre-initialized networks with unit-variance Gaussian matrices omitting the orthonormalization step. For W-LSUV, input images were scaled by $\sqrt{\mathbf{M}_1}$.

Figures 3.1, 3.2, and 3.3 show learning curves for the four LSUV variants. Figures 3.1 shows accuracies averaged over models and activation functions. It seems that, on average, C-LSUV performs best, followed by W-LSUV. G-LSUV performs poorly. Both C-LSUV and W-LSUV outperform regular LSUV. Looking at individual performances across the different model/activation settings (Figures 3.2 and 3.3), W-LSUV appears to perform best in all cases except FitNet-4 together with *tanh*. (For FitNet-1 together with ReLU it seems like W-LSUV approaches a slightly higher accuracy than the other algorithms towards the end of training.)

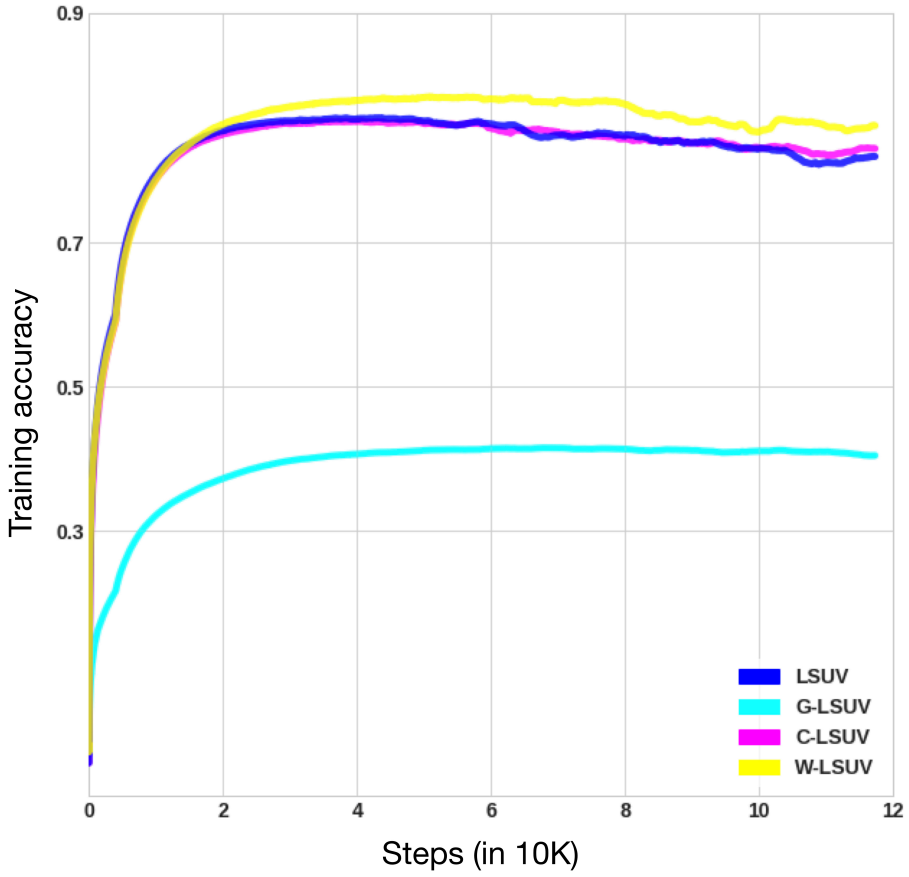

Figure 3.1: Training accuracy per algorithm, averaged over activations and models. Y-axis is on a logit scale. Each line corresponds to one of the four algorithms (LSUV, G-LSUV, C-LSUV, W-LSUV) and is the average of 4x2 individual training runs, one run for each model/activation function setting (the LSUV variants were tested with tanh and relu, but not elu, swish, sigmoid).

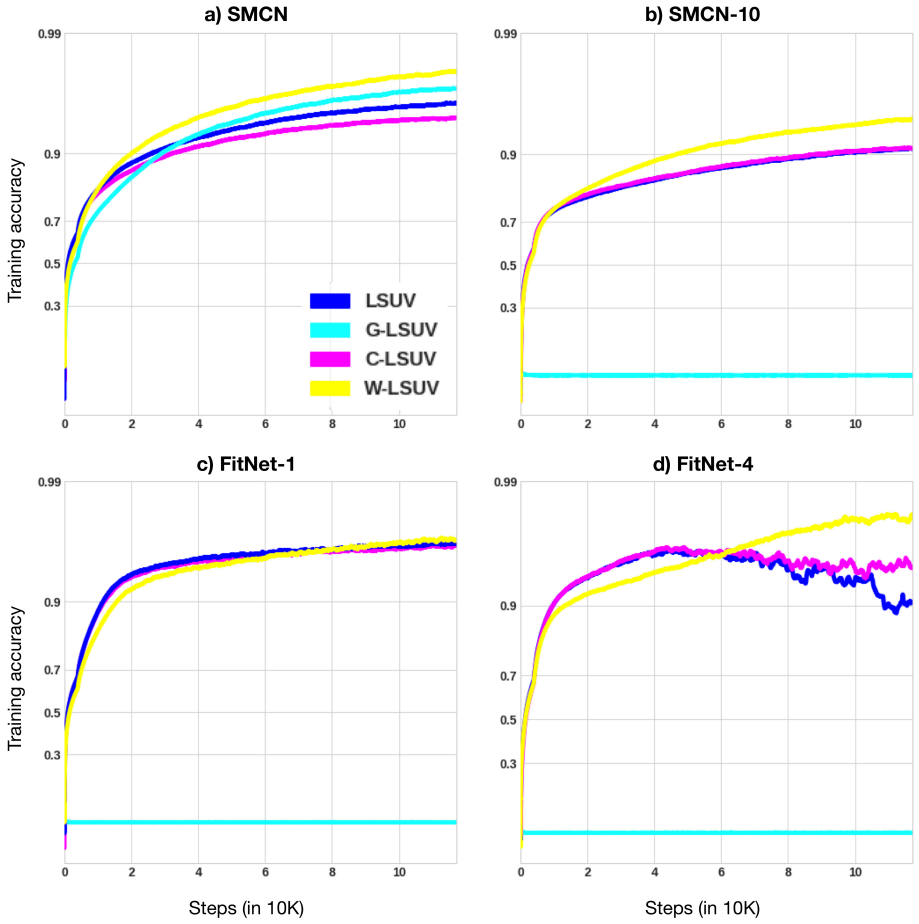

Figure 3.2: Training accuracy per algorithm for each model, with ReLU activation (logit scale).

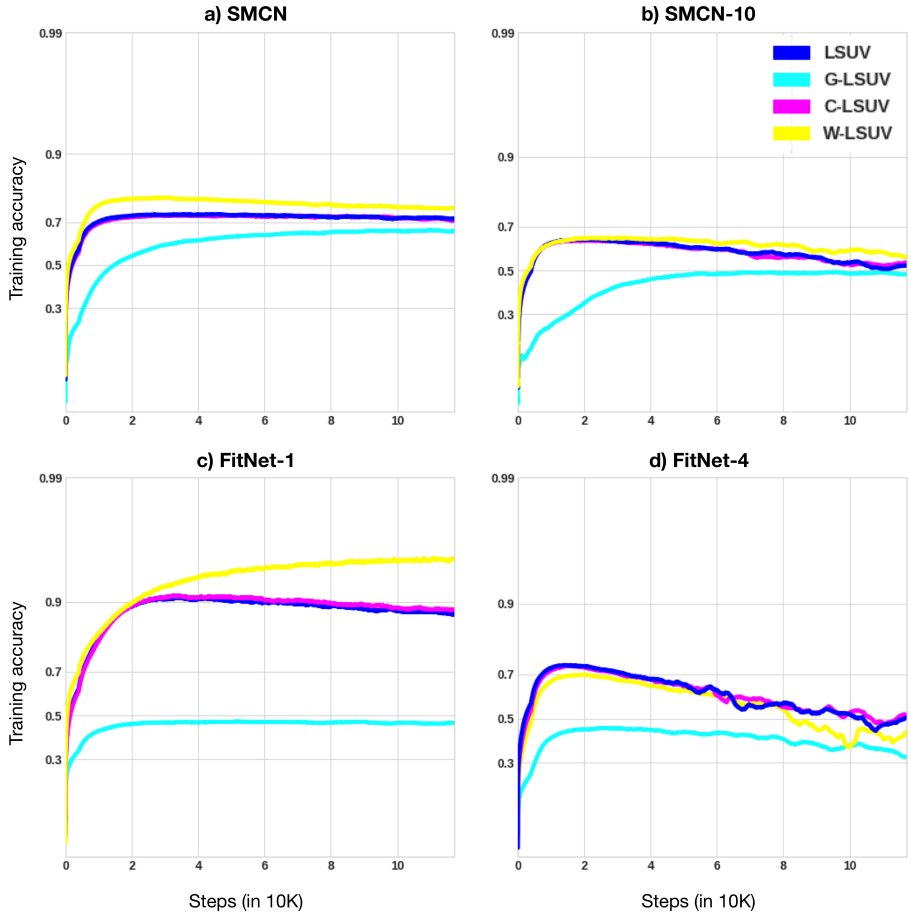

Figure 3.3: Training accuracy per algorithm for each model, with *tanh* activation (linear scale).

Chapter 4

Analysis

Do our algorithms do what they are supposed to do? That is, does, for example, W-LSUV actually stabilize weight-gradient variances throughout the network? How good is each algorithm at keeping other variances like those of pre-activations and gradients constant across layers?
Further, ultimately, how predictive are these capacities of model performance? The algorithms were conceived based on the expectation that stabilizing weight-gradient and other variances would benefit training, but does this supposition hold up when correlating model performances with some notion of stability?
To study the first question, we take a look at the *mean normalized variance variance* achieved by each algorithm; for the second question, we correlate *normalized variance variances* with a summary measure for model performance.

4.1 Do our algorithms do what they are supposed to do?

We want to gauge the stability of variances achieved by each algorithm. To that end, we look at the *mean normalized variance variance* for each algorithm, that is, the variance of the normalized vector of weight-gradient variances (or pre-activation variances, etc.) per layer, averaged over models and activation functions. The normalization step is necessary because error-gradients are scaled by the error at the final layer and so is the vector's variance without normalization. For pre-activations it is technically not necessary to normalize the vector but since raw pre-activation variance variances differ by orders of magnitude we still include this step to ease comparison.

Weight-gradients Figure 4.1 shows relative mean normalized weight-gradient variance variances for each algorithm. For *He*, *Xavier*, and *Algorithms 1-5*, the transparent bar on top shows the average when including those acti-

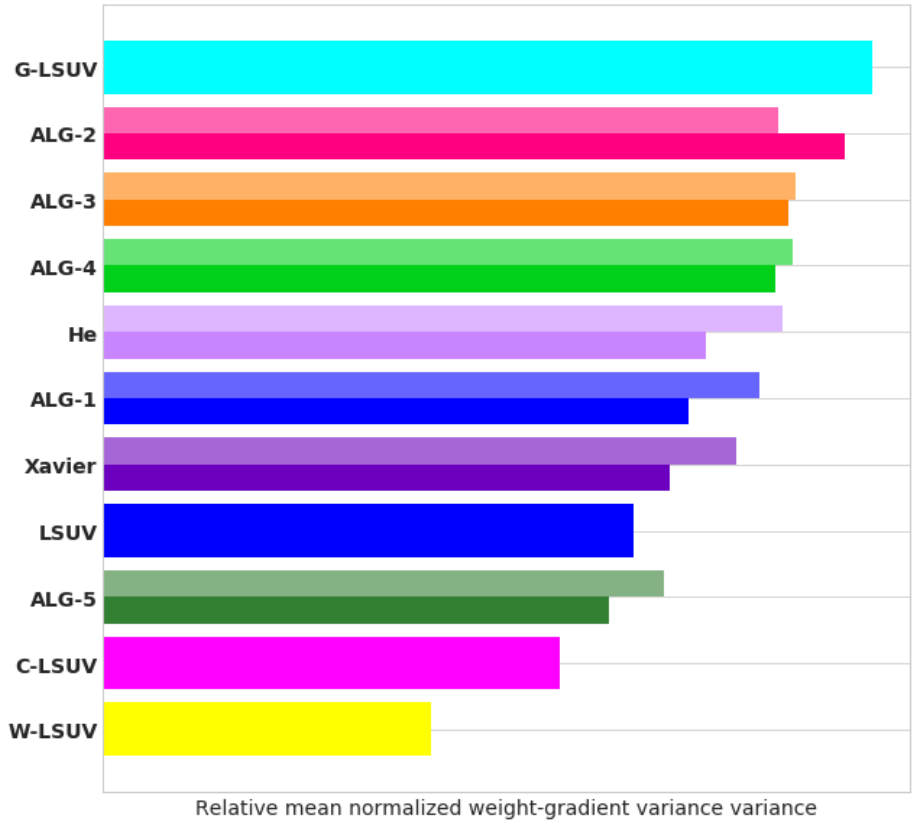

Figure 4.1: Relative mean normalized weight-gradient variance variance per algorithm, i.e., the variance of the normalized vector of weight-gradient variances per layer, averaged over models and activation functions. W-LSUV *humiliatingly* out-stabilizes all other algorithms. The transparent bar on top shows the average when including those activation functions that the LSUV variants were not tested on (*sigmoid*, *swish*, *elu*). Data is missing for Algorithm 5 with *sigmoid* on models *SMNC*, *SMNC-10*, and *FitNet-4*.

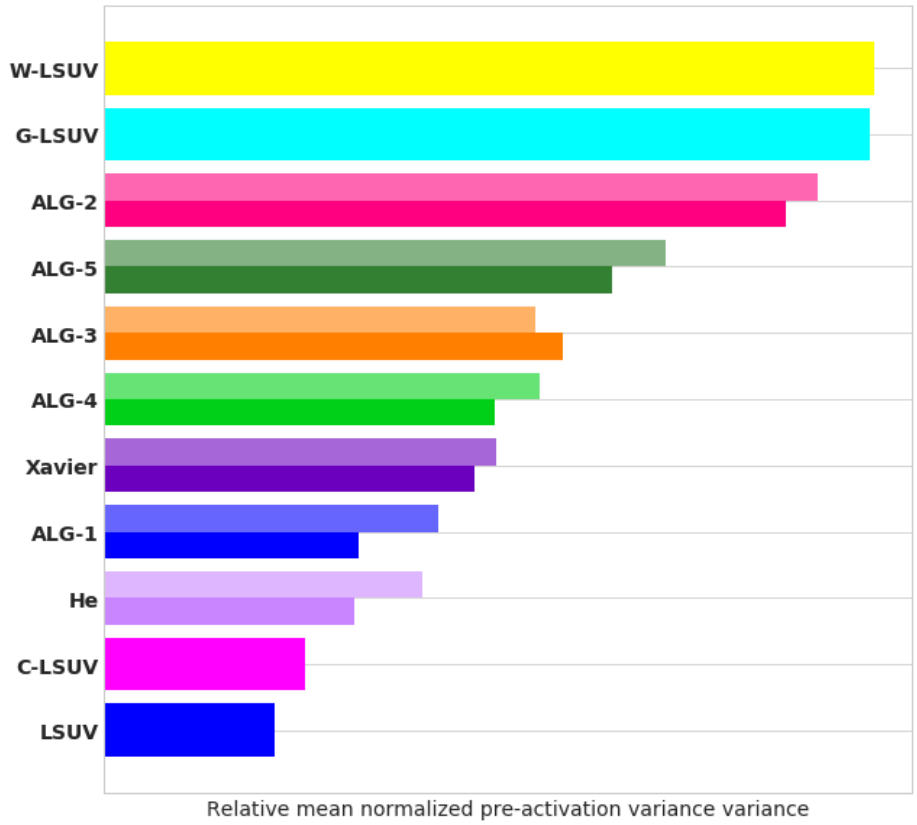

Figure 4.2: Relative mean normalized pre-activation variance variance per algorithm. The transparent bar on top shows the average when including those activation functions that the LSUV variants were not tested on (*sigmoid*, *swish*, *elu*).

vation functions that the LSUV variants were not tested on (*sigmoid*, *swish*, and *elu*). Data is missing for *Algorithm 5* with *sigmoid* on models *SMNC*, *SMNC-10*, and *FitNet-4*. We can see that, among those algorithms tested, W-LSUV is indeed the one that best stabilizes weight-gradient variances, followed by C-LSUV and Algorithm 5. The latter two were not designed specifically to stabilize weight-gradient variances but to strike a compromise between keeping forward- and backward variances constant across layers, that is, variances of pre-activations and gradients on pre-activations. Stabilizing *both* these quantities seemingly also leads to stable weight-gradient variances. Stabilizing pre-activation-gradient variances alone (G-LSUV, Algorithm 2) leads to unstable weight-gradient variances. Meanwhile, stabilizing pre-activations alone (LSUV, Algorithm 1) also seems to yield relatively stable weight-gradient variances.

Pre-activations Moving on to mean normalized pre-activation variance variances (Figure 4.2), we see that, unsurprisingly, LSUV yields the most stable pre-activation variances. It is followed by C-LSUV, *He*, and Algorithm 1. Like LSUV, He and Algorithm 1 are designed specifically to keep pre-activation variances constant. It might be surprising, initially, that Algorithm 1 in that respect does not perform at least as well as *He* even though it adapts to the activation function used while *He* is tailored only to the *ReLU* activation. This is explained, however, by other layer types that influence pre-activation variances and are not addressed by Algorithm 1 nor *He*, as mentioned in the evaluation section. Pre-activation variances are most volatile with W-LSUV. W-LSUV does not address pre-activation- but activation-variances (or, more precisely, variances of outputs of layers preceding layers with weights; this includes, e.g., outputs of pooling layers). If the transformations applied to the pre-activations vary, keeping the variances of outputs of those transformations constant will result in variant pre-activation variances. Further, for conv-layers, W-LSUV adapts the preceding layer's output-variance to the area of the conv-layer's output map which also varies from layer to layer.

Pre-activation-gradients Finally, we look at mean normalized pre-activation-gradient variance variances (Figure 4.3). G-LSUV achieves the highest stability, followed by W-LSUV and Algorithm 2. This is as expected. Both G-LSUV and Algorithm 2 normalize pre-activation gradients disregarding the forward signal, the difference being that Algorithm 2 scales weights assuming only activation layers where G-LSUV can deal with any layer type. What is interesting is the significant lead W-LSUV has over C-LSUV in terms of stabilizing pre-activation-gradient variances given that they both compromise between keeping pre-activation-gradient variances and one other quantity stable: for C-LSUV, this quantity is the pre-activation variance; for W-LSUV, it is the expectation of the squared output of layers preceding layers with weights (which is, in most cases, identical to the variance of activations), scaled by the area of the following layer's output map if that layer is a conv-layer. Apparently, the compromise sought in W-LSUV is easier to achieve than that in C-LSUV.

In summary, comparing mean normalized variance variances did not hold any surprises, with LSUV, W-LSUV, and G-LSUV showing superiority in preserving pre-activation-, weight-gradient-, and pre-activation-gradient variances, respectively. One moderately interesting observation is that LSUV and especially C-LSUV also attain steady weight-gradient variances.

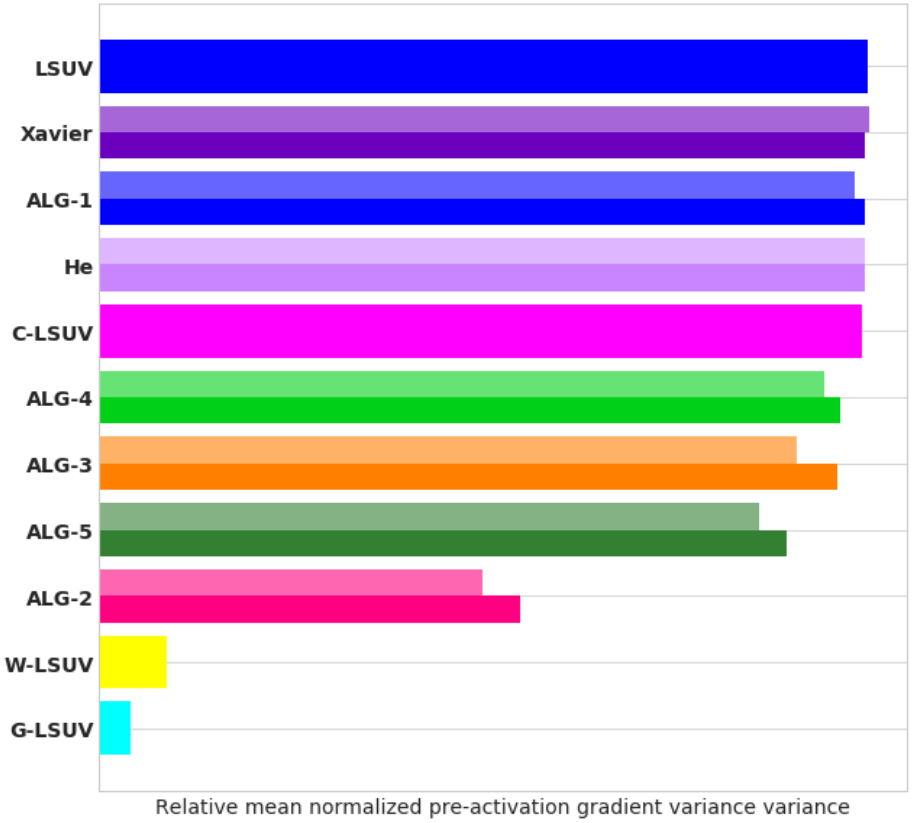

Figure 4.3: Relative mean normalized pre-activation-gradient variance variance per algorithm. The transparent bar on top shows the average when including those activation functions that the LSUV variants were not tested on (*sigmoid*, *swish*, *elu*).

4.2 Does stability predict performance?

Every algorithm discussed in here is motivated by the notion that stabilizing some quantity in the network would benefit training. Can this notion be justified empirically?
In testing the algorithms with various models and activation functions we collected a host of data that we can now use to investigate this question. To explore the connection between stability and performance, we define a summary measure for model performance:

$$\text{performance} = \frac{49}{50} \cdot \text{max-accuracy} + \frac{1}{50} \cdot \frac{\text{total-steps} - \text{steps-to-max}}{\text{total-steps}},$$

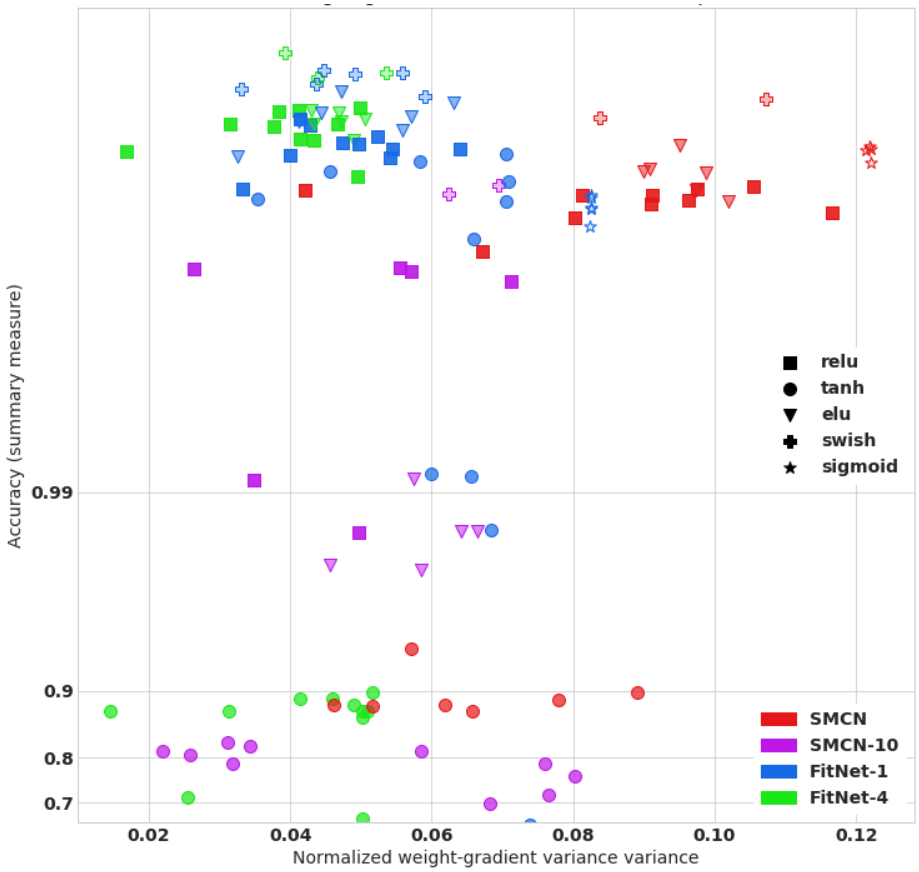

Figure 4.4: Normalized weight-gradient variance variance and performance. Performances are on a logit-scale.

i.e., the model achieving the highest accuracy performs best but if two networks achieve near identical accuracies, the one that achieves it earlier is considered to perform better. To define stability with respect to weight-gradient variances, etc., we again use normalized variance variances.

Weight-gradient variances The scatter plot in 4.4 shows performances according to above formula on the y-axis and normalized weight-gradient variance variances on the x-axis. Each dot represents one of the model/algorithm/activation function settings. Marker color encodes model, marker shape activation function; we do not encode the algorithm since it is irrelevant to our question. Performance is on a logit-scale. If there is a correlation between weight-gradient variance and performance, it is not obvious from this plot.

We look at the correlation between performance and normalized weight-gradient variance variance for each model/activation-function pair individually, as well as for each model including all activation functions, for each activation function including all models, and, finally, for all models and all activation functions together (Table 4.1). We use Pearson's r as correlation coefficient. The value in brackets is the correlation when excluding failed runs that resulted in a performance lower than 0.5. It is not shown if this leaves less than two runs or if there were no such runs (if the latter is the case the value is marked with an asterisk). Since model architecture and activation function have a vast influence on model performance, raw joint correlations are mostly meaningless. The mean of the coefficients was computed by applying Fisher z-transform to r values, taking the average, and applying the inverse transform.
Much like the scatter plot, the coefficients do not unequivocally suggest a relationship between weight-gradient variance stability and model performance with correlations varying greatly in direction and magnitude both within models and within activation functions, and contingent on whether or not failed runs were taken into account. The mean correlation over all models and activation functions is insignificantly negative when including all runs and strongly positive when only successful runs are considered.

Pre-activation variances We examine pre-activation variances in the same manner. Figure 4.5 shows a scatter plot of performances and normalized pre-activation variance variances; table 4.2 shows correlations between normalized pre-activation variance variance and performance analogous to table 4.1. For pre-activation variances, the pertinence to model performance is more prominent. The correlation is negative for all but one model/activation function setting (when excluding failed runs); the mean correlation is strongly negative irrespective of whether or not failed runs were included.

Pre-activation-gradient variances Finally, we investigate pre-activation gradient variances (scatter plot in figure 4.6; table 4.2). From the scatter plot we first observe that NPAGVVs are highly contingent on model architecture[1]. This renders across-model correlations meaningless but we can still audit the mean of within-model/activation function correlations. While meaningful, the values shown in table 4.3 are also unexpected, suggesting a positive correlation between NPAGVV and model performance, both when including and excluding failed runs. The values are more varied than for NPAVVs.

[1]More precisely on the number of layers with trainable weights in the model since the variance of normalized pre-activation-gradient variances was not corrected for sample size; both SMCN-10 and FitNet1 have 11 layers with trainable weights.

	SMCN	**SMCN-10**	**FitNet-1**	**FitNet-4**	**all**	**mean**
relu	-0.324 (0.218)	-0.532 (0.209)	-0.644 (0.307)	-0.448 (0.031)	-0.226 (-0.041)	-0.496 (0.193)
tanh	-0.684*	0.074 (-0.694)	-0.333*	0.115*	0.13 (0.009)	-0.243 (-0.447)
elu	-0.832 (-0.621)	-0.442 (0.616)	0.6*	-0.148 (-0.385)	-0.161 (0.012)	-0.274 (0.07)
swish	-0.261 (1.0)	-0.004 (1.0)	0.582 (-0.023)	0.004 (-0.606)	-0.264 (-0.315)	0.099 (0.994)
sigmoid	0.634 (-0.346)	0.349	0.656 (0.328)	0.132	0.562 (0.913)	0.468 (-0.01)
all	-0.175 (0.175)	-0.38 (-0.025)	-0.158 (-0.218)	-0.145 (0.085)	-0.099 (0.054)	
mean	-0.36 (0.707)	-0.126 (0.913)	0.203 (0.194)	-0.076 (-0.236)		**-0.093** **(0.529)**

Table 4.1: Correlation: normalized weight-gradient variance variance and performance.

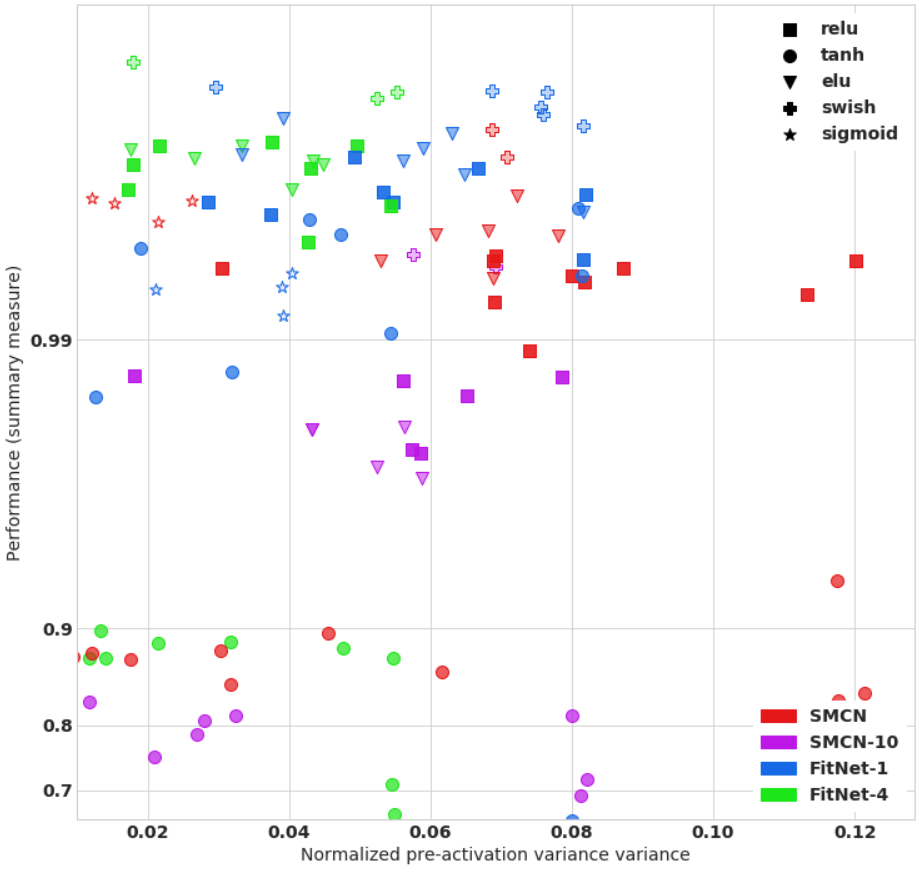

Figure 4.5: Normalized pre-activation variance variance and performance. Performances on a logit-scale.

	SMCN	**SMCN-10**	**FitNet-1**	**FitNet-4**	**all**	**mean**
relu	-0.452 (0.013)	-0.386 (-0.145)	-0.375 (-0.232)	-0.399 (-0.209)	-0.306 (-0.236)	-0.404 (-0.145)
tanh	-0.106*	0.129 (-0.634)	-0.392*	-0.592*	-0.03 (-0.194)	-0.261 (-0.452)
elu	-0.924 (0.354)	-0.698 (-0.614)	-0.724*	-0.628 (-0.513)	-0.441 (-0.039)	-0.775 (-0.428)
swish	-0.791 (-1.0)	-0.57 (-1.0)	-0.219 (-0.627)	-0.422 (-0.964)	-0.353 (-0.174)	-0.536 (-0.878)
sigmoid	-0.636 (-0.325)	-0.289	-0.603 (-0.478)	-0.068	-0.112 (-0.555)	-0.425 (-0.404)
all	-0.36 (0.134)	-0.043 (0.201)	-0.268 (-0.168)	-0.144 (-0.117)	-0.145 (0.039)	
mean	-0.668 (-0.015)	-0.395 (-0.49)	-0.484 (-0.511)	-0.44 (-0.699)		**-0.505** **(-0.46)**

Table 4.2: Correlation: Normalized pre-activation variance variance and performance.

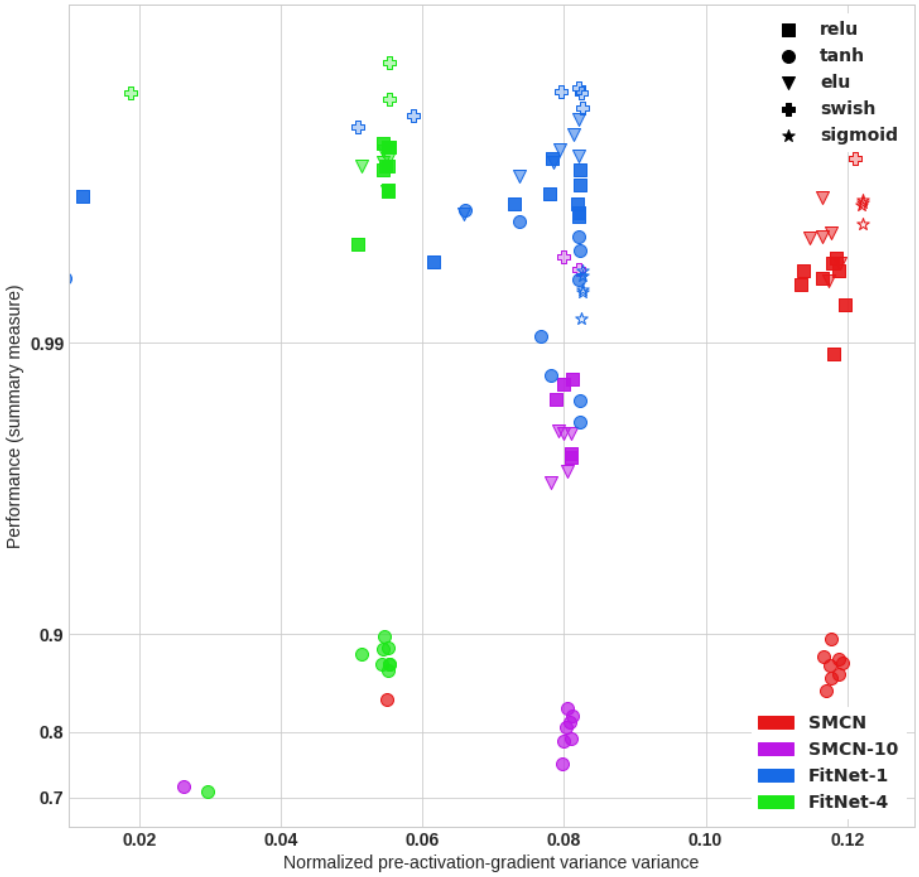

Figure 4.6: Normalized pre-activation-gradient variance variance and performance. Performances on a logit-scale.

	SMCN	**SMCN-10**	**FitNet-1**	**FitNet-4**	**all**	**mean**
relu	0.696 (-0.148)	0.297 (-0.393)	0.699 (0.145)	-0.046 (0.355)	0.318 (-0.199)	0.46 (-0.012)
tanh	0.047*	-0.014 (0.682)	0.651*	0.672*	0.198 (0.341)	0.385 (0.55)
elu	0.998 (-0.177)	0.652 (0.484)	0.957*	0.449 (0.078)	0.5 (-0.034)	0.932 (0.525)
swish	0.676 (-1.0)	0.841 (-1.0)	0.273 (0.878)	-0.036 (0.317)	0.084 (-0.226)	0.517 (0.69)
sigmoid	0.63 (-0.326)	0.349	-0.133 (0.495)	0.132	0.541 (0.914)	0.269 (0.102)
all	0.375 (0.276)	0.17 (0.492)	0.299 (0.524)	0.104 (0.455)	0.22 (0.293)	
mean	0.833 (-0.154)	0.487 (0.305)	0.628 (0.739)	0.263 (0.378)		**0.595** **(0.393)**

Table 4.3: Correlation: Normalized pre-activation-gradient variance variance and performance.

Chapter 5

Discussion

Some disjointed remarks before beginning the discussion:

- The results presented here are based on too little data to draw definite conclusions from. To get a proper sense of the presented algorithms' merit we should average performances over many runs instead of considering just a single run. Moreover, we should consider a wider range of model architectures, learning rates, optimization methods, data-sets, etc.
- It would be interesting to see how the algorithms fare on a more challenging task than CIFAR-10, and when using deeper networks.
- We did not test the algorithms on any models that included Batch Normalization or residual connections.
- For testing W-LSUV, which is meant to stabilize weight-gradient variances, the Adam optimizer with its adaptive learning rate might have been an unfortunate choice. We used it due to resource constraints, to avoid a lengthy parameter search.
- In W-LSUV, inputs are re-scaled depending on the area of the first conv-layer's output map. It would be important to probe the impact of re-scaling alone, which could partially explain W-LSUV's superiority in our tests. We should compare performances when inputs are re-scaled for all algorithms.
- In all our experiments (including performance comparisons), we erroneously used a version of LSUV that normalizes *pre-activations*. LSUV, as described in the paper, normalizes *activations*. However, this should not make too great a difference for the models we used in our tests. For *tanh*, normalizing pre-activations corresponds to scaling activations such that they have a variance of 0.39. For *ReLU*, normalizing

pre-activations corresponds to scaling activations such that they have a variance of 0.34.

- For further tests, we advise to also include a version of W-LSUV that omits the scaling factor m.
- Similarly, for testing C-LSUV, it would be interesting to vary between preserving pre-activation-variances vs. activation-variances, and pre-activation-gradient-variances vs. activation-gradient-variances.
- A better loss function for the balancing algorithms is

$$\text{loss}(x) = x^{-1} \quad \text{if} \quad x < 1 \quad \text{else} \quad x.$$

- Further, the balancing algorithms work much better if a *learning rate* is included, i.e., if the last line is expanded like this:

$$\widetilde{W} \leftarrow W \cdot (\text{loss}_1 + \text{loss}_2) \;/\; (\text{loss}_1\sqrt{v_1} + \text{loss}_2\sqrt{v_2})$$
$$W \leftarrow (1-\alpha)W + \alpha\widetilde{W}$$

  with α small, contingent on the number of iterations. In the version that was used in the performance comparisons and analyses, the weight variance often does not actually converge to one that balances between preserving forward and backward variances but rather oscillates between two variances, one that preserves forward variances and one that preserves backward variances. By virtue of randomness, this defect averages out in a network with many layers.
- We came across (Krähenbühl et al., 2015) only while compiling the introduction, after experiments and analyses were done and the body of this work was written. Their approach, like W-LSUV, is motivated by the intuition that all layers should learn at the same *rate*. However, their conception of the *rate at which a layer learns* consists in the ratio of the magnitude of the weights to that of the weights' gradients, which they hence aim to keep approximately constant. Their approach differs substantially from ours. This idea, that weight gradient magnitudes should stand in some relation to weight magnitudes seems compelling and future renditions of W-LSUV could attempt to incorporate it.

In chapter 2, we attempted to extend ideas from (Glorot and Bengio, 2010) and (He et al., 2015) to the activation functions *tanh*, *elu*, *selu*, *swish*, and *sigmoid*. We found that, due to some intractable integrals involved, deriving formulae analogous to those in (Glorot and Bengio, 2010) and (He et al., 2015) is either not possible or not practically useful for the named activation

functions. Resorting to numerical integration, then, we proposed several algorithms for finding suitable initial weights, compatible with any activation function. We tested the proposed algorithms on a range of model architectures and activation functions. Their performance fell short of our expectations. Despite doing essentially the same as *Xavier* and *MSRA* initialization but adapted to the activation function they brought little if any improvement over the activation-unspecific initializations in most tests. A possible explanation, as mentioned in the evaluation section, is that neither of the proposed algorithms (nor *He* or *Xavier*) take into account other layer types that influence activation and gradient variances, like Dropout and Max-Pooling, such that an activation-agnostic initialization might, haphazardly, result in more desirable activation and gradient variances than an activation-aware initialization.

In chapter 3, we presented some variations of Mishkin and Matas's LSUV. We first adapted LSUV to normalize pre-activation gradients instead of activations (G-LSUV), leading to poor results. We then presented a version that compromises between preserving pre-activation-gradient variances and pre-activation variances (C-LSUV) and obtained more promising results. Finally, we introduced a variant that is intended to stabilize variances of weight-gradients; this version (W-LSUV) outperformed all other variants in seven out of eight model/activation function settings.

We confirmed that W-LSUV actually stabilizes weight-gradient variances by comparing *mean normalized weight-gradient variance variances* across algorithms, finding that W-LSUV yields by far the lowest mean normalized weight-gradient variance variance. In the same manner, we confirmed that LSUV best stabilizes pre-activation variances, and G-LSUV best stabilizes pre-activation-gradient variances.

W-LSUV was inspired by the intuition that consistent weight-gradient variances would benefit training. All layers should learn at approximately the same rate so units can effectively co-adapt[1]. We sought to substantiate this intuition by correlating normalized weight-gradient variance variances with a summary measure for model performance. The results we obtained from that are perplexing as they fail to demonstrate a link between stable weight-gradient variances and improved performance. This is contrary to what we expected given W-LSUV's superior performance and the fact that it best stabilizes weight-gradient variances.
Can we infer from this that the intuition that motivated W-LSUV is misguided? The sparsity of our data does not permit this conclusion; a myriad factors impact a neural network's training success and the normalized vari-

[1] "Co-adaptation", in the context of neural networks, has multiple meanings, one being an undesirable phenomenon (Srivastava et al., 2014); what is meant here is that units should make effective use of each other.

ance is just one, and possibly not the best, metric for stability. Nevertheless, the results give us pause, illustrating that we do not know *precisely* which quantity to care for when we want to ensure a neural network's successful training. Besides the question *which* variance to consider (that of pre-activations? of weight-gradients? of activations? of activation-gradients? of pre-activation-gradients?) we should also contemplate whether variance *per se* is the ideal metric to consult. Regarding exploding gradients, (Philipp et al., 2018) note:

> «There is no well-accepted metric for determining the presence of pathological exploding gradients. Should we care about the length of the gradient vector? Should we care about the size of individual components of the gradient vector? Should we care about the eigenvalues of the Jacobians of individual layers? [...] The problem is that it is unknown whether exploding gradients, when defined according to any of these metrics, necessarily lead to training difficulties.»

Tending to the *variance* of activations and gradients as a heuristic has proven its merit in practice but other properties of values flowing through the network, and how they relate to model performance, might be worth studying as well. This could be done empirically, by training a wide range of models over a wide range of hyper-parameters, tasks, etc., collecting values like gradients and activations, and correlating model performances with a variety of indices over these values (i.e., essentially repeating the experiments we presented in the analysis but on a larger scale). Based on those results, then, one could formulate new goals for weight initialization methods and devise better algorithms.

With W-LSUV, we tried to stabilize weight-gradient variances following the intuition that all layers should learn at the same rate. In a similar vein, optimization methods like *AdaGrad* adjust the magnitude of parameter updates according to the history of gradients. This should offset irregularities in the magnitude of gradients across layers that might arise through an unfortunate choice of initial weights. The difference between adaptive learning rate and weight initialization methods (besides the two being two entirely different domains wherefore the comparison might seem nonsensical) is that, while the former can make adjustments continuously throughout training, with weight initialization we are limited to finding an adequate setting *ex ante*. This entails that adaptive learning rates can make adjustments for single parameters where weight initialization can adjust weights only in a layer-wise fashion. Parameter-wise adjustments allow for larger updates being made for sparse features, and smaller updates for more frequent ones. Since, before training, we do not know which role each unit will play in the network—which feature it will encode and how that feature is distributed

in the data—we cannot make such parameter-wise adjustments at the time of initialization. Crucially, the choice of initial weights still has a considerable impact on training even when using adaptive learning rates (see performance comparisons; we used the *Adam* optimizer (Kingma and Ba, 2014)).

Chapter 6

Conclusion

In this thesis, we demonstrated how existing weight initialization schemes can be reworked to be compatible with any and all activation functions. With the increment in performance this brought about falling short of our expectations we were cautioned that the influence of non-activation-layers should not be neglected. For very deep networks it might be critical that a weight initialization take into account such layers' impact on activations and gradients.

This impelled us to turn our focus to *data-driven* initialization methods. We adapted an existing data-driven method, LSUV, which in its original form addresses only activation variances, to also tend to gradient variances. We introduced three novel algorithms based on LSUV: G-LSUV, C-LSUV, and W-LSUV. G-LSUV preserves only pre-activation-gradient variances; C-LSUV compromises between preserving pre-activation-gradient and pre-activation variances; and W-LSUV is an attempt at preserving weight-gradient variances. We achieved promising results with the latter two methods, though further testing is required to establish their merit with certainty. Especially the approach taken in W-LSUV deserves further attention; some refinements might yield additional improvements.

We were unable to demonstrate a link between model performance and the consistency of weight-gradient variances across layers. This was surprising, as the presumption of such a link motivated W-LSUV which appears to work well practice. While we have far too little data for this result to be conclusive it is inspiring insofar as it acts as a token of our ignorance with respect to what precisely a weight initialization is supposed to accomplish. Existing methods are, more or less, heuristics. This points to a potentially extremely fruitful avenue for future research: much could be gained from studying empirically the values that flow through a neural network during training and correlating model performances with a variety of indices over these values, much like we did on a small scale in our analyses. Insights

from this could motivate, e.g., better optimization and weight initialization algorithms.

Appendix A

Appendix

A.1 Preliminaries

- The variance of a random variable X can be expressed in terms of its expectation:

$$\mathrm{Var}[X] = \mathbf{E}[X^2] - (\mathbf{E}[X])^2.$$

- The expectation of the product of two independent random variables is the product of their expectations:

$$\mathbf{E}[XY] = \mathbf{E}[X]\,\mathbf{E}[Y].$$

- The variance of the sum of two independent random variables is the sum of their respective variances:

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y].$$

- The expectation of a function of a random variable X with probability density function p is given by:

$$\mathbf{E}[g(X)] = \int_{-\infty}^{\infty} g(x)p(x)\mathrm{d}x.$$

A.2 Arriving at the formula for pre-activation variances

The variance of a single unit's pre-activation can generally be obtained like so:

$$
\begin{aligned}
\mathrm{Var}[y_i^l] &= \mathrm{Var}\left[\sum_{j=1}^{n^l} W_{ij}^l x_j^l + b_i^l\right] && n^l \text{ is the size of layer } l-1 \\
&= \mathrm{Var}\left[\sum_{j=1}^{n^l} W_{ij}^l x_j^l\right] && \text{biases are 0 at initialization} \\
&= \sum_{j=1}^{n^l} \mathrm{Var}\left[W_{ij}^l x_j^l\right] && \text{variance of sum of i. r. v.} \\
&= n^l \mathrm{Var}\left[W_{ij}^l x_j^l\right] && \text{same for all j (elements in W, x i.i.d.)} \\
&= n^l \mathrm{Var}\left[w^l x^l\right] && \text{might as well drop indices} \\
&= n^l \left(\mathbf{E}[(w^l x^l)^2] - (\mathbf{E}[w^l x^l])^2\right) && \text{variance formula} \\
&= n^l \left(\mathbf{E}[(w^l)^2]\mathbf{E}[(x^l)^2] - (\mathbf{E}[w^l]\mathbf{E}[x^l])^2\right) && \text{expectation of product i. r. v.} \\
&= n^l \mathbf{E}[(w^l)^2]\mathbf{E}[(x^l)^2] && W \text{ has zero-mean} \\
&= n^l \left(\mathbf{E}[(w^l)^2] - (\mathbf{E}[w^l])^2\right)\mathbf{E}[(x^l)^2] && \text{subtracting zero} \\
&= n^l \mathrm{Var}[w^l]\mathbf{E}[(x^l)^2] \\
&= n^l \mathrm{Var}[w^l]\mathbf{E}[f^2(y^{l-1})].
\end{aligned}
$$

A.3 Arriving at the formula for activation-gradient variances

The variance of the gradient of a single unit's activation can generally be obtained like so:

$$
\begin{aligned}
\mathrm{Var}[\Delta x_i^l] &= \mathrm{Var}\left[\sum_{j=1}^{\hat{n}^l} \hat{W}_{ij}^l \Delta y_j^l\right] \\
&= \sum_{j=1}^{\hat{n}^l} \mathrm{Var}\left[\hat{W}_{ij}^l \Delta y_j^l\right] && \text{sum of i. r. v.} \\
&= \hat{n}^l \mathrm{Var}\left[w^l \Delta y^l\right] && \text{same for all j} \\
&= \hat{n}^l \mathrm{Var}\left[w^l\right] \mathbf{E}\left[(\Delta y^l)^2\right] && \text{see above} \\
&= \hat{n}^l \mathrm{Var}\left[w^l\right] \mathbf{E}\left[(f'(y^l)\Delta x^{l+1})^2\right] \\
&= \hat{n}^l \mathrm{Var}\left[w^l\right] \mathbf{E}\left[(f'(y^l))^2\right] \mathbf{E}\left[(\Delta x^{l+1})^2\right] && \text{exp. of prod. of i. r. v.} \\
&= \hat{n}^l \mathrm{Var}\left[w^l\right] \mathbf{E}\left[(f'(y^l))^2\right] \left(\mathbf{E}\left[(\Delta x^{l+1})^2\right] - (\mathbf{E}\left[\Delta x^{l+1}\right])^2\right) && \text{subtract zero} \\
&= \hat{n}^l \mathrm{Var}\left[w^l\right] \mathbf{E}\left[(f'(y^l))^2\right] \mathrm{Var}\left[\Delta x^{l+1}\right].
\end{aligned}
$$

Bibliography

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2): 157–166, 1994.

Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

F. Chollet et al. Keras. https://keras.io, 2015.

D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

R. Eldan and O. Shamir. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pages 907–940, 2016.

D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Artificial Intelligence and Statistics*, pages 153–160, 2009.

D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016a.

K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016b.

D. Hendrycks and K. Gimpel. Adjusting for dropout variance in batch normalization and weight initialization. *arXiv preprint arXiv:1607.02488*, 2016.

G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

X. D. Huang, Y. Ariki, and M. A. Jack. Hidden markov models for speech recognition. 1990.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pages 971–980, 2017.

P. Krähenbühl, C. Doersch, J. Donahue, and T. Darrell. Data-dependent initializations of convolutional neural networks. *arXiv preprint arXiv:1511.06856*, 2015.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

H. W. Lin, M. Tegmark, and D. Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017.

A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

D. Mishkin and J. Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

G. Philipp, D. Song, and J. G. Carbonell. The exploding gradient problem demystified-definition, prevalence, impact, origin, tradeoffs, and solutions. *arXiv preprint*, 2018.

D. Pirjol. The logistic-normal integral and its generalizations. *Journal of Computational and Applied Mathematics*, 237(1):460–469, 2013.

B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein, and S. Ganguli. Exponential expressivity in deep neural networks through transient chaos. In *Advances in neural information processing systems*, pages 3360–3368, 2016.

M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein. On the expressive power of deep neural networks. *arXiv preprint arXiv:1606.05336*, 2016.

P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. 2018.

A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

O. Sharir and A. Shashua. On the expressive power of overlapping architectures of deep learning. *arXiv preprint arXiv:1703.02065*, 2017.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

D. Sussillo and L. Abbott. Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558*, 2014.

L. R. Sütfeld, F. Brieger, H. Finger, S. Füllhase, and G. Pipa. Adaptive blending units: Trainable activation functions for deep neural networks. *arXiv preprint arXiv:1806.10064*, 2018.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

M. Telgarsky. Benefits of depth in neural networks. *arXiv preprint arXiv:1602.04485*, 2016.

Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

_____

signature

_____

city, date