JOHANNES KEPLER
UNIVERSITY LINZ

Author
Pieter Dewyse

Submission
Institute for Machine Learning

Thesis Supervisor
Günter Klambauer

Assistant Thesis Supervisor
Richard Freinschlag

Month Year
(For Information: Month and year of submission to Examination and Recognition Services)

DESIGNING NORMALIZER-FREE EFFICIENTNETS

Master's Thesis
to confer the academic degree of

Master of Science
in the Master's Program

Artificial Intelligence

SWORN DECLARATION

I hereby declare under oath that the submitted Master's degree thesis has been written solely by me without any third-party assistance (except for proofreading). Information or aids other than in the sources indicated in this work have not been used and those used have been fully documented. I realize that the mere presence of a reference does not avoid plagiarism. Sources for literal, paraphrased and cited quotes have been accurately credited. If I have used the exact words, phrases, clauses, or sentences of someone else, I have enclosed that information in quotation marks. If I have paraphrased the opinions of someone else, I have not enclosed the paraphrase in quotation marks, but I have stated those opinions in my own words and put a reference at the end of it. I agree that the university uses the digital version of this work for an electronic plagiarism check using suitable software, and that for technical reasons my work may be stored in a database.

The submitted document here present is identical to the electronically submitted text document.

Place, Date

Signature

Abstract

As neural networks grow in complexity, Batch Normalization (BN) has become essential for stabilizing training and improving generalization. However, BN introduces drawbacks such as increased memory usage, computational overhead, and reduced effectiveness for small batch sizes. This study aims to modify EfficientNet to function without normalization while maintaining stable signal propagation.

Signal Propagation Plots (SPPs) were used to visualize activations, gradients, and weights, identifying instability sources and revealing BN’s influence on training dynamics. Baseline testing showed BN actively scales gradients using its learnable parameters, potentially explaining its regularization effects.

EfficientNet-B0 was redesigned without BN by mainly using Scaled Exponential Linear Units (SELU). While this stabilized signal propagation, a performance trade-off was observed, with variance fluctuations in Squeeze-and-Excitation blocks.

This study confirms stable training without normalization is possible, though alternative regularization techniques are needed to retain performance. future work could focus on designing novel regularization techniques that mimic how BN is able to influence both the forward and backward pass using its learnable parameters. The full code implementation is available at: https://github.com/PDewyse/MT_Normalizer_free_Neural_Nets

Contents

1. Checklist before handing in

- Sign your paper on second page and add date on title page
- Remove & resolve comments everywhere
- Add enters to make text/tables/figures nicely fit everywhere

2. Introduction

As neural networks are increasingly used for tasks with varying levels of complexity, the demand for larger and more sophisticated neural networks continues to grow. To meet these demands, neural networks are scaled by increasing the size of individual layers, stacking more layers, and implementing architectural improvements to enhance performance. These advancements allow neural networks to better fit complex datasets and produce more accurate predictions (Goodfellow et al., 2016).

However, with this continued development, certain limitations have become apparent. One of the early challenges of increasing the number of layers is the vanishing or exploding gradient problem (Bengio et al., 1994). While this issue has largely been mitigated through the use of appropriate activation functions to prevent signal degradation between layers (Glorot & Bengio, 2010)., it still poses difficulties when training extremely deep neural networks, such as those exceeding 100 layers.

To improve training in deep neural networks clever techniques are utilized. These techniques include but are not limited to, weight initialization to promote stable signal propagation (H. Zhang et al., 2019), normalization layers to maintain a consistent signal range during the forward pass (Qiao et al., 2019), and architectural modifications such as skip connections, which help preserve signal integrity (He et al., 2015a). The use of these techniques is vital in popular neural network architectures, including for example ResNets, DenseNets, and transformers.

A popular normalization technique is Batch Normalization (BN) due to its ability to stabilise training, accelerate convergence, and improve generalization (Ioffe & Szegedy, 2015). However, its use comes with side effects, including increased memory usage, added computational overhead, and reduced effectiveness for smaller batch sizes. Additionally, performance inconsistencies may arise between training and inference (Hoffer et al., 2018). Given these drawbacks, designing neural networks that inherently maintain stable signal propagation without relying on normalization would be more efficient.

This work aims to modify EfficientNet (M. Tan & Le, 2020), a neural network that heavily relies on BN, to function without normalization. To achieve this, Signal Propagation Plots (SPPs) will be used to visualize signal propagation during training and identify components responsible for instability (Brock, De, & Smith, 2021). This study seeks to answer the following questions:

- Can SPPs visualize BN’s impact on the forward and backward passes?
- Can a stable EfficientNet be designed without BN, and what is the performance trade-off?

2.1. Fundamentals of Neural Networks

Vital components for human decision-making are neurons. These biological cells are found in most species of the animal kingdom, with the exception of sponges and placozoans (Arendt et al., 2019). Neurons communicate through connections called synapses, and together they form a nervous system that is responsible for receiving, processing, and transmitting information throughout the body. While they primarily connect to other neurons, they can also connect to other types of cells, such as muscle cells and gland cells, to transmit signals and regulate various functions. In turn, this allows for complex behaviours associated with for example humans (Thau et al., 2024).

Artificial neural networks (ANNs) are machine learning algorithms resembling a nervous system or biological neural network. In ANNs, nodes or units translate to neurons, and the connections between them are represented by (synaptic) weights. These weights will function like synapses by adjusting the strength of signals passed between nodes (Baba, 2024).

While this architecture seems promising, and it theoretically bears a high resemblance to a biological nervous system. The actual implementation as a machine learning algorithm will need to address several key differences such as but not limited to:

- Complexity

Biological neural networks found in human brains approximates to about 100 billion neurons connecting to up to 1 quadrillion synapses (J. Zhang, 2019a). This is considerably more compared to for example ResNet-50, a classical ANN architecture, which has approximately 1 million neurons and 25 million weights (He et al., 2015a). Designing neural networks with a similar number of parameters to a human brain imposes challenges in terms of both hardware (storage and processing) and energy consumption (Alzubaidi et al., 2021)

- Energy efficiency

As mentioned in the previous paragraph, energy consumption can pose considerable problems. For example, the energy requirement of a human brain is around 20 watts of continuous power consumption (Balasubramanian, 2021). For context, operating a laptop consumes between 30 and 70 watts of power. While a direct comparison to the energy consumption of electronics to a human brain is not straightforward due to differences in how they operate and their purposes, it can be seen how there is still a large gap to be made up in order to create a computer or program capable of rivalling the versatility of a human brain. For example, the training of a large neural network such as OpenAI's ChatGPT-3 required 1064 MWh excluding continuous server costs (Patterson et al., 2022).

- Learning mechanisms

The biggest difference between biological neural networks and artificial neural networks (ANNs) lies in their learning processes. Biological networks will learn through a complex combination of electrical and chemical processes, including but not limited to synaptic plasticity, which involves altering the connections between neurons and synapses (Stampanoni Bassi et al., 2019) and neuromodulation, neurons that regulate other neurons using chemicals (Bazzari & Parri, 2019). In contrast, ANNs involve iterative mathematical algorithms, such as backpropagation and gradient descent, to gradually improve their performance.

- Adaptability

A Biological neural network is better able to adapt after it is created. Due to the Biological network being composed of cells, they can be replaced individually in case of for example damage or when the connection is no longer needed (Arcuschin et al., 2023)**.** On the other hand, ANNs are composed of numerous parameters that are generally fixed after the training is completed. This is because the impact of a single weight within a neural network is hard to estimate.

- Generalization

ANNs are trained on large sets of data that are usually be labelled individually (supervised). This step is time-consuming as increasingly large neural networks will also need increasingly large datasets. However, the human brain can learn from smaller datasets and is able to do this with less or no labelling on the data itself (un-supervised).

The following sections will give a brief introduction on basic Artificial Neural Networks. This includes the main design principles, the core components that are required for proper functioning, and finally the common problems associated with ANNs, especially with respect to deep neural networks. It should be noted that this overview is not a complete analysis for the workings of ANNs but serves to provide background that is necessary in subsequent chapters.

2.1.1. Basic Neural Network Architecture

A basic three-layered feedforward neural network (FNN) with two input units, a hidden layer of size three, and a final layer with one output unit is depicted in Figure 1 . Every neuron, except for the input neurons, will be equipped with an activation function $f(s)$ that will introduce non-linearity to the signal coming from lower layers before passing it to the next layer. The input and hidden layer in Figure 1 can be modelled using algebra with the following terms:

x is a 2×1 input feature column vector containing all the values from the input neurons
a is a 3×1 activation column vector containing all the values from the input neurons
W is a 2×3 weight matrix containing the weights that connect the input and hidden layer
b is a 3×1 bias vector containing the bias values for the hidden layer
σ is the activation function $f(s)$, here indicated as a sigmoid

The activation, **a** can then be calculated as follows:

$$\boldsymbol{a} = f(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) = \sigma(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$$

Using this formula, layers can be stacked and the activations of each layer can be calculated and passed to the next layer (Dai et al., 2017). Hence, the name feed-forward neural network (FFNN). The use of activation functions or non-linearities is two-fold. First, by using a nonlinear function the neural network will be able to capture more complex patterns in the data. Second, without an activation function, multiple layers would collapse into a single larger linear transformation, essentially reducing the network to a simple linear model and negating the benefits of having a multi-layered architecture.

The choice of activation function depends on both the layer and the task. In hidden layers, the ability to capture non-linear relationships is important. Common functions include: ReLU, Leaky ReLU, or Swish. For the output layer, the activation function depends on the machine learning task. In regression, no activation function is typically applied, allowing the output neurons to predict continuous values. For classification, a sigmoid or SoftMax activation function is used to generate probability outputs, with SoftMax ensuring the probabilities sum to 1 for multi-class problems (Dubey et al., 2022).

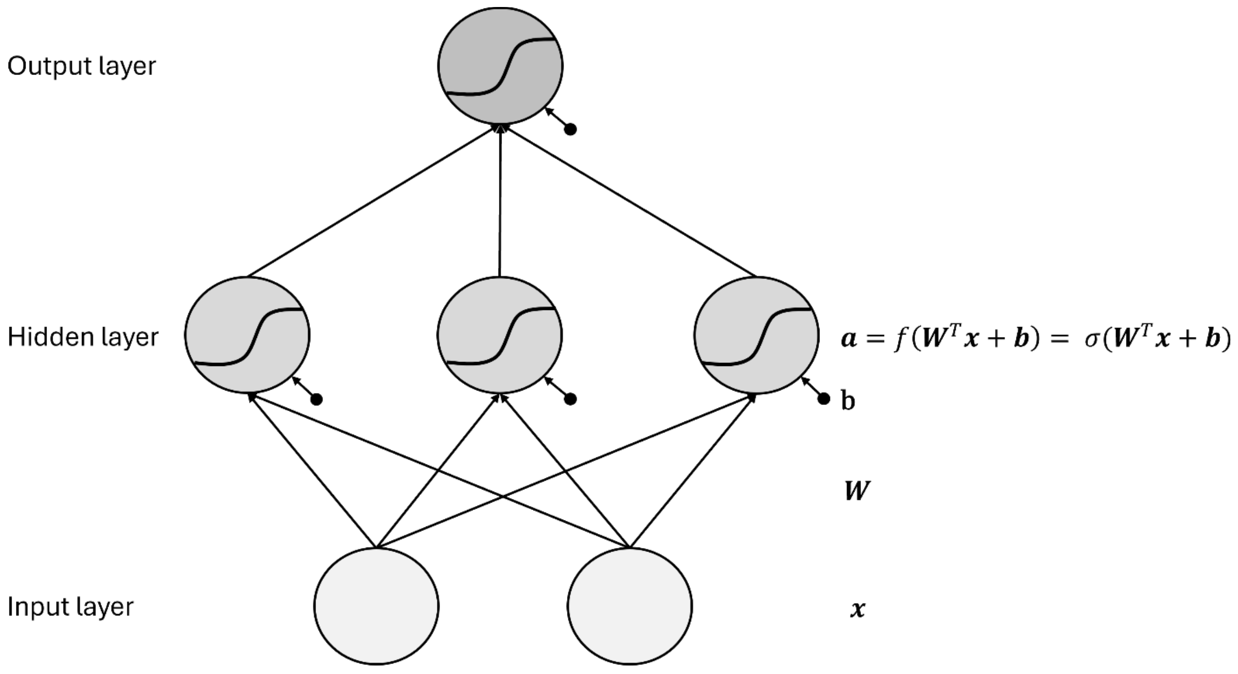

Output layer
Hidden layer
Input layer
$\boldsymbol{a} = f(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) = \sigma(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$
b
$\boldsymbol{W}$
$\boldsymbol{x}$

*Figure 1 Multi-layer Perceptron with three layers (**x** is a 2×1 input feature column vector containing all the values from the input neurons, **a** is a 3×1 activation column vector containing all the values from the input neurons, **W** is a 2×3 weight matrix containing the weights that connect the input and hidden layer, **b** is a 3×1 bias vector containing the bias values for the hidden layer, σ is the activation function f(s), here indicated as a sigmoid)*

It should be noted that that FFNNs are the general class describing networks where data flows in one direction. Therefore, the use of a non-linearity is not required, using an identity activation function is still a valid way of building FFNNs. The correct way to classify the example from Figure

1 would be as a Multi-layer Perceptron due to the use of multiple layers in combination with sigmoid non-linearities (Yamany et al., 2015).

2.1.2. Key Concepts in Neural Network Training

Neural network learning is achieved by adjusting the weights, also known as parameters, in such a way that the neural network will produce the desired output. To do this, a loss function will be needed to quantify how far the produced output or prediction is from the desired output or the true value, label, or ground truth. In addition, an algorithm will be required to take the result from the loss function or loss and use it to iteratively update the weights of the neural network. This weight update should be done in such a way that the loss will decrease (Schmidhuber, 2015).

2.1.2.1. Loss Function and Training Error

The main purpose of a loss function is to produce a single scalar value, called the loss, that quantifies how well a neural network is performing on a specific training sample. The choice of the loss function depends on the specific task that the neural network is designed to perform, such as regression or classification (Terven et al., 2024).

In regression tasks, a continuous value will be predicted by the neural network. For example, the average temperature per day. Thus, the loss function will take one scalar, e.g. the temperature, and produce a loss value. One common function to do this is the Mean Squared Error (MSE) with a formula as follows:

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widehat{y_i})^2$$

Here, N represents the number of training samples, y_i is the ground truth and $\widehat{y_i}$ is the value predicted by the model for the i^{th} training sample. Due to squaring the errors, large errors will be penalized more heavily compared to smaller errors. The MSE is taken over all samples and will also be known as the training error during training, or the empirical error (Terven et al., 2024).

For classification tasks, the model will categorize inputs into different classes. For example, an image is given, and the model must determine whether the image depicts a cat, dog, or car. Essentially, the neural network will be tasked with producing a probability distribution over all classes where every output neuron will have the probability that the input belongs to that class. A common loss function to quantify the prediction will be cross-entropy loss:

$$\text{Cross Entropy Loss} = -\sum_{c=1}^{C} p_c \log(\widehat{p_c})$$

C is the number of classes, p_c is the true probability of class c, and $\widehat{p_c}$ is the predicted probability of class C. It should be noted that the cross-entropy loss is only calculated for a single sample. To obtain the empirical error the formula needs to be adjusted as follows and will also be named the Categorical Cross-Entropy Loss:

$$Categorical\ Cross\ Entropy\ Loss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(p_{ij})$$

N is the number of samples, C is the number of classes, y_{ij} is the true probability for the ith sample and the jth class, and p_{ij} is the predicted probability that sample i belongs to the class j (Terven et al., 2024).

2.1.2.2. Learning Algorithm – Backpropagation

The empirical error calculated in the previous section will be used as an objective to be minimised during training. This minimization is achieved using the backpropagation algorithm combined with gradient descent. During backpropagation, gradients will be calculated and used to adjust the neural network's weights to reduce the loss. These gradients are calculated by taking the partial derivative of the loss with respect to the weights in the neural network (J. Zhang, 2019b).

Gradient descent will update the weights in the opposite direction of the gradient of the loss function. This essentially changes the weights in order to create a loss that is situated in a local minimum in the solution space. The formula for updating the weights will be as follows: $w \coloneqq w - \eta\nabla L(w)$ where w represents the weights, η is the learning rate, and $\nabla L(w)$ is the gradient of the loss function with respect to the weights. More advanced optimization algorithms include Stochastic gradient descent, RMSprop, and Adam (J. Zhang, 2019b).

Stochastic gradient descent (SGD) uses the same update rule as gradient descent but will update the weights for a subset or mini batch of the training data. This effectively smoothens the noise from the gradients, making the loss function less sensitive to outliers and providing more stable weight updates. More information on Root Mean Square Propagation (RMSprop) and Adam (Adaptive Moment Estimation) can be found in Section 2.3.3.3.

2.1.3. Common Problems in Deep Neural Networks

A major problem during the training of deep neural networks is the vanishing and exploding gradient problem. This occurs during backpropagation when the gradients used to update the network weights either become too small (vanishing) or too large (exploding) as they are propagated back through the layers. Vanishing gradients lead to very small weight updates, causing poor learning in deeper networks. In contrast, exploding gradients cause increasingly large weight updates, leading to instability during training (Bengio et al., 1994). Gradient problems are commonly attributed to improper weight initialization and the use of specific activation functions.

Using *activation functions* such as the sigmoid or tanh can lead to vanishing gradients, because their outputs are constrained to a small range. This results in derivatives that are less than 1 during backpropagation, which causes gradient magnitudes to decrease as they pass through multiple layers. In this situation, an activation function such as ReLU (Rectified Linear Unit) can be used as address the vanishing gradient problem (H. H. Tan & Lim, 2019).

During *weight initialization*, weights that are initialized too small can exacerbate the vanishing gradient problem. In the same way, initializing the weights too large can result in exploding gradients. This is especially important in deep neural networks where small deviations can grow uncontrollably through multiple layers. Techniques such as Xavier initialization and He initialization can be used to initialize weights so that they are neither too small nor too large, keeping gradients more stable (Glorot & Bengio, 2010; He et al., 2015b). This will be discussed in more detail in Section 2.3.2.1.

Other common problems are overfitting and underfitting of neural networks. In both cases, the model is not able to generalize well to new or unseen data. Overfitting occurs when a model is too complex compared to the data, causing it to learn the noise present in the training data rather than the underlying patterns. This results in poor generalization to new data. In case of underfitting, the model is not complex enough to capture all the nuances in the data. Proper training of a model requires finding a balance between a model that is complex enough to capture the important patterns in the data but not so complex that it will overfit (Aliferis & Simon, 2024).

Internal covariate shift refers to the change in the distribution of network activations due to updates in the parameters of previous layers during training. This shift forces the model to constantly readjust, slowing down training and making convergence more difficult. Internal covariate shift can be mitigated by using batch normalization, this technique normalizes the activations within each layer, leading to more stable and faster training (Ioffe & Szegedy, 2015). Batch normalization is discussed in chapter 2.3.1.1.

2.2. Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) is a neural network architecture designed to handle images and other grid-like data structures. CNNs can extract and learn the spatial structure of input data to make predictions. They are commonly used for computer vision tasks that involve image recognition, image segmentation, and object detection (Alzubaidi et al., 2021). Conventional CNNs consist of two major parts. (1) Convolutional layers and pooling operations that will serve as a feature extractor, and (2) fully connected layers with a final activation function acting as a classifier to produce an output (Y. Lecun et al., 1998).

2.2.1. Feature extraction

In CNNs, convolutional layers will play a vital role in feature extraction using so called kernels or filters on the input data. This is done by sliding or convolving the filters over grid-like data such an image represented by a 2D matrix. The filters detect spatial patterns, such as edges, textures, and more complex features in case of deeper networks. Mathematically, feature extraction on a 2D matrix can be done using a discrete convolution operation (Goodfellow et al., 2016):

$$s_{a,b} = (\boldsymbol{W} * \boldsymbol{x})_{a,b} = \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} w_{i,j} x_{a+i,b+j}$$

where,

$s_{a,b}$ is value in the feature map at position (a, b) after applying the convolution filter.

$\boldsymbol{W}$ is the kernel weight matrix of dimension $R \times R$.

$*$ denotes the convolution operation

$\boldsymbol{x}$ is the image or input feature map.

$w_{i,j}$ are the weights of the filter at position (i, j).

$x_{a+i,b+j}$ is the value from the input feature map at position $(a + i, b + j)$.

A notable feature of convolutional layers is the concept of weight-sharing, where each filter (or kernel) is applied across the entire input feature map. This leads to a reduction in the number of parameters compared to fully connected layers, where each input neuron would have its own weight (Goodfellow et al., 2016).

The concept of weight-sharing can also be explained through the concept of receptive fields. In convolutional layers, the neural network focuses on smaller regions in the input feature map, depending on the size of the kernel. As the kernel size increases, the number of parameters (weights) will also increase. If the kernel size is large enough to cover the entire input, every pixel in the feature map would have its own associated weight. In this situation, the convolutional layer would essentially act like a fully connected layer. Another way to achieve this is by using a number of 1×1 kernels that is equal to the number of pixels in the input (M. Tan & Le, 2020). In practice, using small kernels (i.e. small local receptive field) will allow the neural network to extract and focus on small patterns within the input feature map. In contrast, using larger kernels (larger receptive fields), the neural network will focus more on capturing global patterns in the data (Gregor & LeCun, 2010).

The total number of parameters for a convolution layer can be calculated using the following formula: $\left((R \cdot R \cdot C_l) + 1\right) \cdot C_{l+1}$ where R is the kernel size, C_l is the number of input channels, and C_{l+1} is the number of output channels (or filters) in the layer. One additional parameter is added to account for the bias term (Y. Lecun et al., 1998).

In addition to convolution layers, pooling layers are commonly used during feature extraction. Pooling operations, such as max pooling or average pooling, do not have learnable parameters but are used to reduce the spatial dimensions of feature maps. A max pooling operation will select the maximum value within the region of the input feature map covered by the filter as it slides over the feature map. Average pooling works similarly but computes the average value of the pixels within the window. In general, pooling operations help summarize information in the feature maps without adding more trainable parameters (Gholamalinezhad & Khosravi, 2020).

Figure 2 provides an example of a pooling operation on 4×4 input feature map with a stride of 2. The stride determines how many pixels (horizontally and vertically) the filter moves at a time.

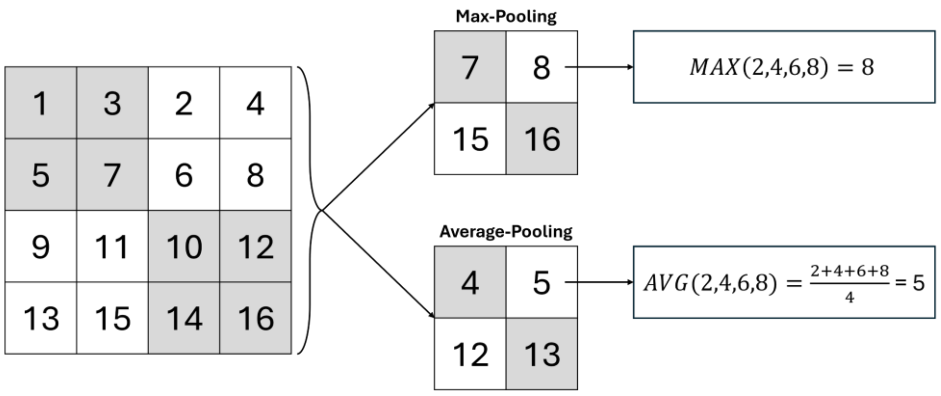


Figure 2 Max pooling and average pooling operations (on a 4×4 input feature map with a stride of 2)

2.2.2. Classification

Once the feature maps have been extracted and reduced in size through convolutional layers or pooling operations, they can be used for tasks such as classification or regression. Fully connected layers are a good candidate for this, as they take the flattened feature maps and connect them to neurons in deeper layers. This results in a decision-making process that weighs the different features to the output depending on their importance (Goodfellow et al., 2016). A comparison of the different layers in a Convolutional Neural network is summarized in Table 1 .

Table 1 Comparison between convolutional, pooling, and fully connected layers.

Layer type	Parameters	Purpose
Convolutional Layer	Low	Feature extraction
Pooling Layer	None	Dimensionality reduction
Fully Connected Layer	High	High-level decision-making

2.2.3. Popular CNN Architectures

2.2.3.1. LeNet-5

The LeNet architecture, developed by Yann LeCun (Y. Lecun et al., 1998), was one of the first successful implementations of Convolutional Neural Networks. It was specifically designed for digit recognition tasks in datasets such as the MNIST database (Y. Lecun et al., 1994). LeNet uses a combination of convolutional layers to extract features from input images, and average pooling (subsampling) layers to reduce dimensionality, then classifies the extracted features with fully connected layers. A detailed view of LeNet-5 is depicted in Figure 3 .

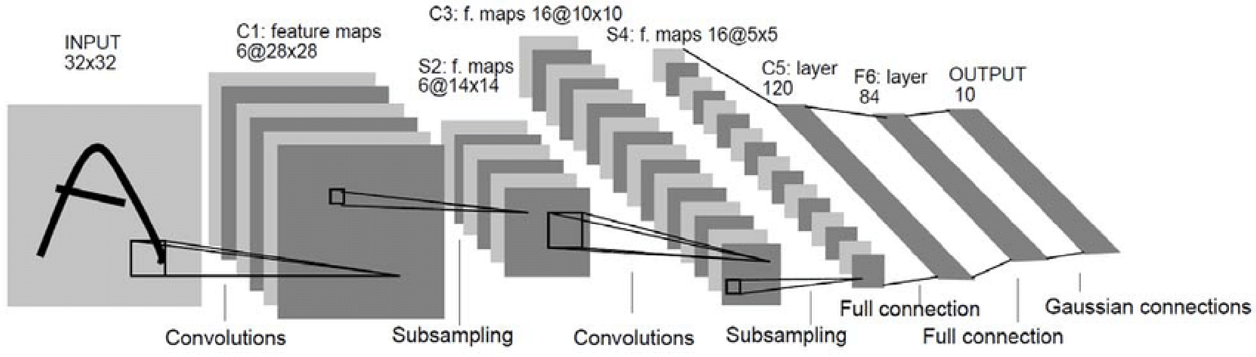


Figure 3 Architecture of LeNet-5, a Convolutional Neural Network, for digits recognition (Y. Lecun et al., 1998).

2.2.3.2. AlexNet

AlexNet, developed by Alex Krizhevsky (Krizhevsky et al., 2012), was built on the principles of LeNet, with modifications to improve performance on larger and more complex datasets such as ImageNet (J. Deng et al., 2009). An important difference is the increased depth of the network, with more filters and a larger receptive field. This allowed AlexNet to learn more complex features within the data, resulting in a model with 60 million parameters, compared to just 60 thousand in LeNet-5. Furthermore, the use of Rectified Linear Unit (ReLU) activations, max pooling layers, dropout, and data augmentation techniques were used to enhance the network's performance.

2.2.3.3. ResNet

ResNet, introduced by Kaiming He (He et al., 2015a) was designed to improve the training of very deep neural networks through the use of residual blocks with skip connections and batch normalization. In a residual block, a combination of convolutional operations, ReLU activations, and batch normalization are used on the main path. Then skip connections, which bypass one or more layers by connecting the input directly to the output, are used to create a shortcut around this section of the main path. These skip connections, or residual paths allow ResNets to learn residual functions more effectively by using the shortcuts as a baseline or reference. With these improvements, training of very deep networks, such as ResNet-152, became feasible, leading to a significant advancement in performance.

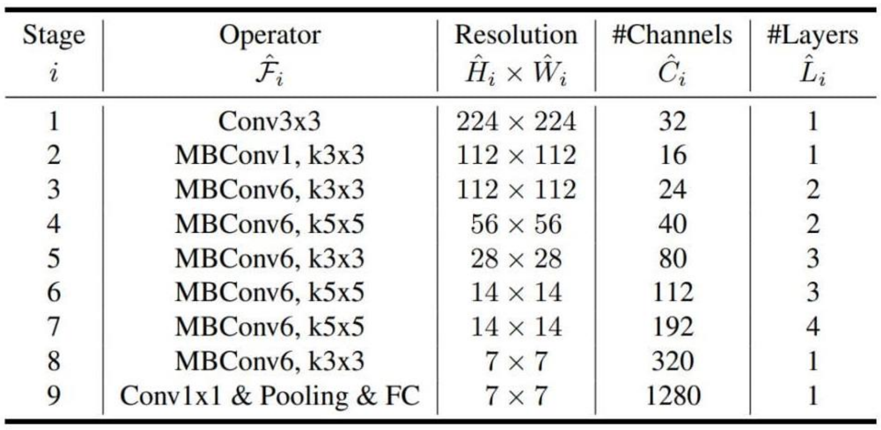
2.2.3.4. EfficientNet

EfficientNet was first developed by Mingxing Tan, Quoc V. Le in 2019 (M. Tan & Le, 2020). As the name suggests, this neural network focuses on efficiency while maintaining high performance. The architecture is inspired by MobilenNetV2 through the use of inverted residual blocks as a backbone (Sandler et al., 2019). However, the key innovation is the implementation of compound scaling.

Compound scaling is a method that allows the model to scale during training by tuning three factors: depth (number of layers), width (number of channels), and input resolution (input image size). During this process, a balance between the model size and the computational cost is maintained, leading to models that are finetuned during training without the need for manually adjusting the architecture.

EfficientNet comes in multiple variants, starting from the baseline model, EfficientNet-B0, up to the larger EfficientNet-B7 that is being upscaled from the baseline. Table 2 lists the different stages of the baseline model as used in the original paper (M. Tan & Le, 2020). The larger models will retain the same structural design, but the individual layers are scaled with a specific factor leading to models with more parameters. It should be noted that the starting size of EfficientNet-B0 was selected using Neural Architecture Search (NAS) (Zoph & Le, 2016). NAS functions similarly to reinforcement learning, where a controller (typically a recurrent neural network) proposes an architecture. The architecture is trained, and the resulting accuracy is then fed back to the controller to produce an improved architecture. Further details on this process can be found in the NAS paper (Zoph & Le, 2016).

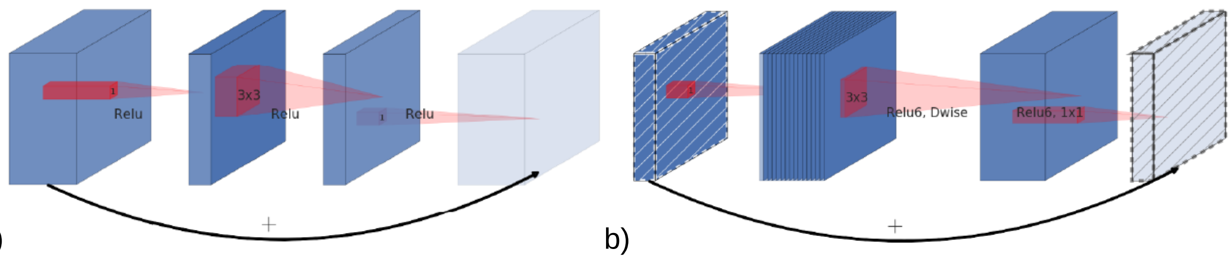
Table 2 EfficientNet-B0 baseline network – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $(\hat{H}_i, \hat{W}_i)$ and output channels $\hat{C}_i$ (M. Tan & Le, 2020). Each MobileNet Convolutional (MBConv) Block is a inverted residual block as defined in the paper introducing MobileNetV2 (Sandler et al., 2019).



Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

EfficientNet is based on the use of inverted residual blocks that were introduced in the MobileNetV2 architecture (Sandler et al., 2019). This block structure is a variant of the original residual blocks found in popular ResNets. Traditional residual blocks use a series of convolutional operations with batch normalization and ReLU activation functions, then each block is connected to a skip connection. Figure 4 depicts a regular residual block where the convolution operations follow a pattern of wide, narrow, wide. Here, the channels are squeezed and then expanded. Of note is that the skip connections start from an input feature with a high number of channels and are then connected to the next block's layer with the high number of channels.

This contrasts with the inverted residual blocks found in MobileNetV2 (Figure 4), where a pattern of narrow, wide, and narrow is followed. Inside the blocks, the network channels are increased using 1x1 convolutions followed by a depth-wise convolution (see below) and then again, a 1x1 convolution to squeeze and ensure that the skip connections at the start and the end have the same number of channels. An added benefit to this configuration is that the number of trainable parameters will also be reduced. The intuition explained by Sandler et al. (Sandler et al., 2019) is that all the necessary information is already compacted inside the squeezed layers so using skip connections here will act to pass along a more compact message between blocks. As the inverted residual blocks are attached via the skip connections between squeezed layers, the authors also termed these blocks as bottlenecks.



a) b)

Figure 4 The difference between a) a residual block (He et al., 2015a), and b) an inverted residual block with skip connections used in MobileNetV2 (Sandler et al., 2019). Note that both use Depth wise Separable Convolutions, and the lightly coloured blocks represent the first layer in the next block.

Depth-wise convolutions were introduced in the first iteration of MobileNet by Howard et al (A. G. Howard et al., 2017). This operation involves a two-step process aiming to reduce the computational cost associated with the standard convolutional operation. The total computational cost for a standard convolution (Figure 5), can be calculated as follows:

$$D_F \cdot D_F \cdot M \cdot N \cdot D_K \cdot D_K$$

where:

D_F is the spatial dimension of the input feature map.

D_K is the kernel size.

M is the number of input channels.

N is the number of output channels.

In contrast, the depth-wise followed by pointwise convolution (Figure 6 and Figure 6) results in a computational cost of: $D_F \cdot D_F \cdot M \cdot 1 \cdot D_K \cdot D_K + M \cdot N \cdot D_F \cdot D_F$. This reduction is achieved by using M kernels of size $D_K \times D_K$, applied to individual input channels (depth-wise convolution), which filters the input without creating new features. These filtered channels are then combined using a pointwise 1x1 convolution, where the output channels are created using a 1x1 kernel. Dividing the computational cost of the depth-wise separable convolution by that of the standard convolution gives:

$$\frac{D_F \cdot D_F \cdot M \cdot D_K \cdot D_K + M \cdot N \cdot D_F \cdot D_F}{D_F \cdot D_F \cdot M \cdot N \cdot D_K \cdot D_K} = \frac{1}{N} + \frac{1}{D_K^2}$$

This shows that, when using a 3x3 kernel and generating 128 output channels, a ratio of 0.112 is obtained. This means that the standard convolution requires approximately nine times the computational cost of the depth-wise separable convolution.

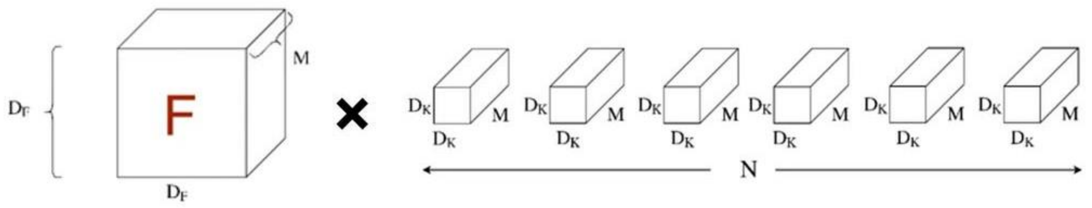
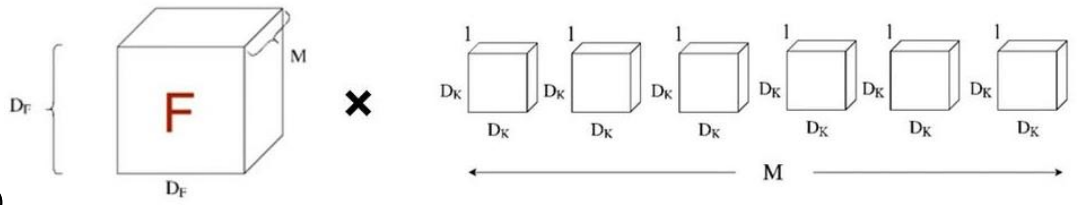


Figure 5 Depiction of the standard convolutional operation with: D_F input feature dimensions, D_K kernel dimensions, M number of input channels, and N number of output channels (PA, 2020).



a)

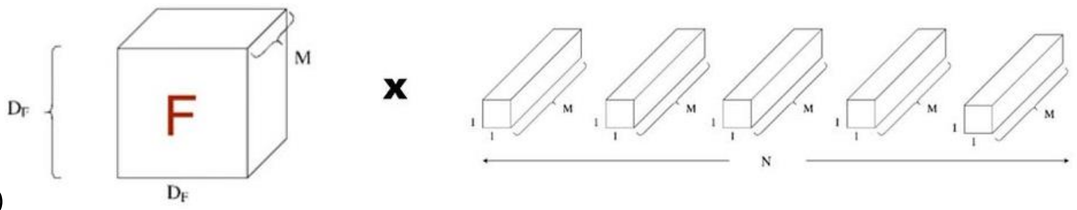


Figure 6 Depiction of a) Depth-wise convolutional operation and b) Point-wise convolutional with: D_F input feature dimensions, D_K kernel dimensions, M number of input channels, and N number of output channels (PA, 2020).

Another vital component contributing to the performance of EfficientNet is the use of Squeeze-and-Excitation (SE) modules (Hu et al., 2019). These modules are used in the inverted residual blocks (Figure 7) to improve the model's ability to pay attention to specific features in the data. SE takes the channels as input and performs a pooling operation followed by two fully connected layers, the first is Swish activated and the second is activated with a sigmoid. The key contribution of the squeeze-and-excite module is to place unequal attention on the different channels of the input. This also explains why a sigmoid non-linearity is used as this will produce values ranging from zero to one.

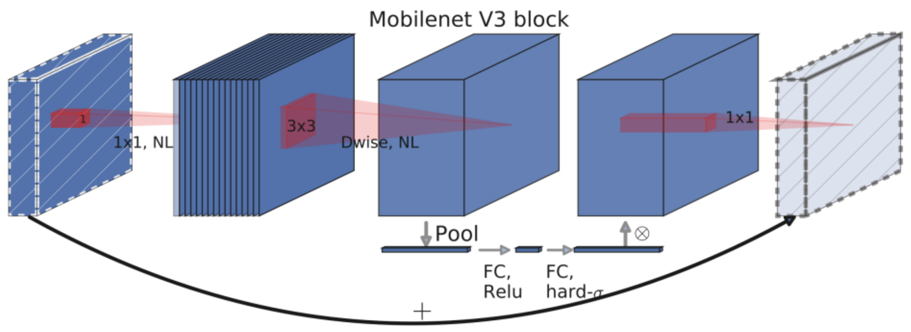


Figure 7 MobileNetV3 block. Similar to a block from MobileNetV2 but now with a squeeze-and-excite (SE) module (A. Howard et al., 2019). Note, within the EfficientNet architecture activation functions vary within SE the first activation function will be a Swish function followed by a regular sigmoid.

A final key element in EfficientNet is the use of Swish activations. The Swish function is defined as: $f(x) = x \cdot \sigma(x)$ where $\sigma(x)$ is the sigmoid function. An important property of Swish, compared to ReLU, is that this nonlinearity is smooth, allowing for better gradient propagation during training. Additionally, Swish can produce negative outputs, which may help the network explore a wider range of activations (Ramachandran et al., 2017).

2.3. Techniques for Stabilizing Neural Network Training

As mentioned in 2.1.3. training neural networks can lead to issues such as vanishing gradients, internal covariate shift, and overfitting or underfitting. In addition to these, other challenges such as the stability of signals related to (pre)activations, weights, and gradients may cause instability during training. For example, uncontrollable oscillations may occur when the network weights grow excessively large, while activations remain near zero. This can lead to the model failing to converge or possibly cause the weights to exceed the limits of bit storage, leading to numerical errors during training.

To avoid these issues during training, many techniques have been developed to improve or stabilise training. These will be described in more detail in the following chapters. To gain a better overview, the stabilization techniques will be divided into different categories depending on the specific aspect of the model they primarily target (e.g., (pre)activations, gradients, or weights). However, it is important to note that while certain methods may stabilise one aspect, such as activations, they may still have side effects on for example gradient propagation.

2.3.1. Stabilization of (pre)activations

Stabilizing activations is primarily achieved through normalization techniques. In general, these normalization techniques aim to whiten the data into having a zero mean and unit variance. This ensures that features during training are scaled appropriately, leading to faster and more stable convergence. The operations of each normalization and standardization technique on the activations are visualized in Figure 8 .

2.3.1.1. Batch Normalization (Batch Norm)

To normalize (pre)activations, Sergey Ioffe and Christian Szegedy (Ioffe & Szegedy, 2015) proposed batch normalization. By introducing two learnable parameters and calculating the mean and variance for every batch, the network will learn to scale and shift the data distribution keeping it centred around zero mean and unit variance per batch. This greatly mitigates the issue of internal covariate shift and accelerates convergence. The batch normalization operation can be represented as follows (A. and L. Zhang et al., 2023):

$$\mathrm{BN}(x) = \gamma \odot \frac{x - \boldsymbol{\mu}_B}{\boldsymbol{\sigma}_B} + \beta.$$

Where:

$\boldsymbol{x}$ are the inputs, or (pre)activations.

$\boldsymbol{\mu}_B$ is the sample mean of the batch B.

$\boldsymbol{\sigma_B}$ is the sample standard deviation of batch B.

$\boldsymbol{\gamma}$ is the learnable scale parameter, with the same shape as the input.

$\boldsymbol{\beta}$ is the learnable shift parameter, with the same shape as the input.

An Additional benefit of Batch Norm is that by computing the mean and variance over different batches, small noise will be generated. This noise, resulting from the batch-to batch variability, will act as a form of regularization during training. However, a significant drawback is that with smaller batch sizes, the benefits of Batch Norm will quickly disappear, as the estimation of mean and variance becomes less accurate. This is especially problematic when working with smaller datasets, where smaller batch sizes are often preferred. Another disadvantage is that the network becomes dependent on the use of mini-batches to produce outputs, which is not ideal for small datasets (Ba et al., 2016).

2.3.1.2. Layer Normalization (Layer Norm)

To address the issues associated with Batch Norm, other normalization techniques have been proposed that work on other different components of the activations. One such technique is Layer normalization (Ba et al., 2016). Instead of normalizing across batches, LayerNorm normalizes the inputs of each layer over the features or channels. This is done similarly to Batch Norm, but here the mean and variance are computed across the channels for each individual sample, rather than across a batch.

The main advantage of this technique is that it becomes independent of the batch size, enabling it to be used for online learning applications. However, A drawback is that the performance on CNNs may vary. This is because CNNs will extract patterns on the spatial information in the input feature maps. When layer norm normalizes across the channels, it can compromise these spatial relationships. Additionally, as layer normalization calculates the mean and standard deviation for individual samples, the operation will be computationally more expensive compared to for example batch normalization.

2.3.1.3. Instance Normalization (Instance Norm)

Another approach focused on reducing dependence on batch size is Instance Normalization (Ulyanov et al., 2016). Instance Norm will normalize the activations using the mean and variance across both the spatial dimensions and the channels of each individual input, effectively normalizing each instance or sample independently. It shares the advantages of Layer Norm by becoming independent of the batch size, but it will now also preserve more of the spatial information contained within the instances. This information is beneficial in applications such as, for example, object detection and image segmentation. The disadvantage here is that treating every sample independently may lead to a lack in the model’s ability to capture global information between instances or samples.

2.3.1.4. Group Normalization (Group Norm)

Group normalization (Wu & He, 2018) normalizes activations by dividing the channels of each instance into smaller groups and then normalizing each group separately. When all channels are included in a single group, Group Norm effectively becomes equivalent to Layer Normalization. In a similar case, when each group contains only a single channel, it becomes Instance Norm.

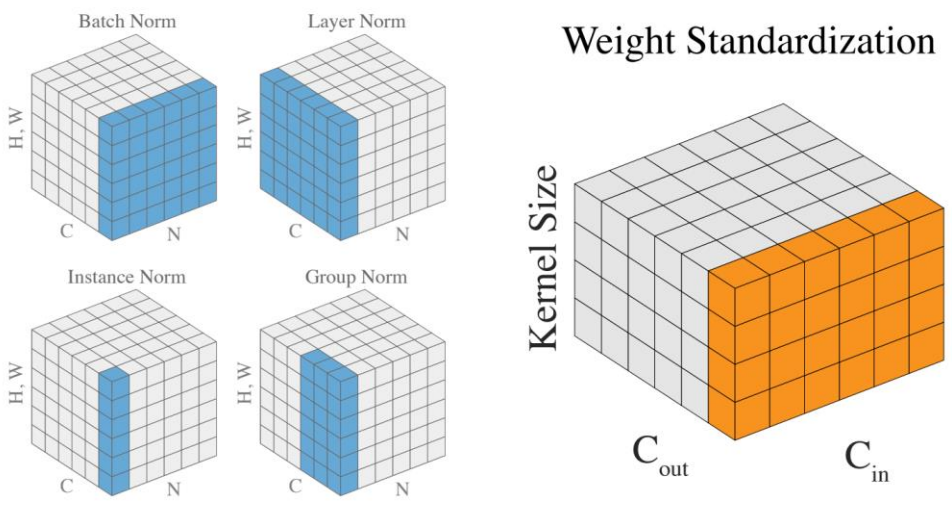


Figure 8 Comparing normalization methods on (pre)activations (blue) and Weight Standardization (orange). (Qiao et al., 2019)

2.3.2. Stabilization of Weights

An alternative approach to stabilise neural network training is by focusing on weight stabilization. These techniques aim to prevent the weights from becoming too small or too large, which may lead to issues such as exploding or vanishing gradients. Additionally, with careful management of the weights, these techniques can also indirectly stabilise activations, reducing training instabilities. The techniques to achieve this will be discussed below.

2.3.2.1. Weight Initialization Strategies

When stabilizing neural networks weights, the first important factor to consider is how the weights are initialized. The main purpose of weight initialization for stabilization will be to preserve the scale of activations and gradients across layers. This will allow the signal to propagate through the network without large fluctuations in magnitude. The following are common initialization strategies:

- **LeCun Initialization**

This initialization method was designed to improve training for networks using sigmoid or hyperbolic tangent (tanh) activation functions (LeCun et al., 1998). Weights are sampled from a

Gaussian distribution with zero mean and a variance of $\frac{1}{n_{fan-in}}$, where n_{fan-in} is the number of input connections to the neurons in that layer. This approach helps to maintain balanced activations as they propagate through the layers, preventing gradients from becoming too large or too small.

- **Xavier Initialization (Glorot Initialization)**

Xavier initialization, also known as Glorot initialization (Glorot & Bengio, 2010), builds on the ideas of LeCun initialization. It will sample the weights from a Gaussian distribution with zero mean but uses a variance of $\frac{2}{n_{fan-in} + n_{fan-out}}$, where n_{fan-in} is the number of input connections and $n_{fan-out}$ is the number of output connections. This technique will balance the input and output sizes, which ensures the variance will have less scaling and be more consistent across layers.

- **He Initialization (Kaiming Initialization)**

He initialization (He et al., 2015b) is specifically designed for networks using activations such as ReLU and other similar variants like Leaky ReLU. This technique will sample the weights from a Gaussian distribution with zero mean and variance $\frac{2}{n_{fan-in}}$, where n_{fan-in} is the number of input connections to that layer. This configuration will ensure variance preserving properties during the forward pass. Alternatively, the variance can be computed using fan-out (the number of output connections) for variance preservation during the backward pass. The factor 2 accounts for the fact that ReLU deactivates approximately half of the neurons, as only positive values are activated using ReLU.

It should be noted that these weight initialization techniques are particularly important in deep neural networks. As the signal propagates through many layers, any small error in weight distribution can be amplified or diminished during training, leading to exploding or vanishing activations and gradients. By sampling weights from Gaussian distributions with specific variances changes in the signal magnitude across layers can be prevented.

2.3.2.2. Weight Standardization

Weight standardization (Qiao et al., 2019) is a normalization technique that functions similar to normalization techniques such as batch normalization discussed in Section 2.3.1. However, instead of normalizing activations, weight standardization is applied directly to the weights of the neural network. This helps to ensure a stable mean and variance in the weights across different layers, which may improve training stability and convergence.

A visualization of weight standardization is shown in Figure 8 The main purpose of this method is to standardize the weights of each layer before use in forward or backward propagation. The standardized weights are computed as follows:

$$\widehat{\boldsymbol{W}} = \frac{\boldsymbol{W} - \mu_W}{\sigma_W}$$

where W is the original weight matrix, μ_W is the mean of the weights, and σ_W is the standard deviation of the weights. This centering and scaling of the weights will lead to a more stable distribution of weight values which may facilitate learning.

It should be noted that weight standardization is often combined with other stabilization techniques such as batch normalization to improve performance, especially in deeper neural networks where weights may introduce instability.

2.3.2.3. Weight Regularization Techniques

In addition to initialization strategies and normalization methods, weight regularization techniques offer a way to stabilise neural network training (Ng, 2004). Weight regularization penalizes large weights during training, for example, in stochastic gradient descent. In general, these techniques will lead to models that learn simpler patterns with less overfitting (Goodfellow et al., 2016). Two common weight regularization techniques are L1 and L2 regularization.

- **L1 regularization**

L1 regularization introduces a penalty proportional to the absolute value of the weights, and can be expressed as follows (Goodfellow et al., 2016):

$$L_{L1} = \lambda \sum_i |\boldsymbol{W_i}|$$

Where W_i are the weights and λ is a hyperparameter controlling the strength of regularization. In practice, this regularization term is added to the task-specific loss function (e.g., mean squared error or cross-entropy loss). The total loss is then calculated as $L_{\text{total}} = L_{\text{task}} + L_{L1}$ where L_{task} is the task-specific loss. L1 regularization will also modify the weight update rule in gradient descent as follows:

$$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} - \eta(\nabla L_{\text{task}} + \lambda \cdot sign(\boldsymbol{w_t}) \cdot \boldsymbol{w_t})$$

Where η is the learning rate, and $sign(\boldsymbol{w_t})$ refers to the sign of each weight element. Adding L1 regularization will increase sparsity by forcing many weights to zero. This leads to simpler models that rely on fewer, more important features and will be less prone to overfitting. However, a disadvantage is the addition of a hyperparameter that needs to be tuned carefully. For example, using a large λ may lead to excessive sparsity, causing underfitting.

- **L2 regularization (Weight Decay)**

L2 regularization, also known as Weight Decay, functions similarly to L1 regularization, but instead penalizes the square of the weights (Goodfellow et al., 2016):

$$L_{L2} = \lambda \sum_i \boldsymbol{W}_{\boldsymbol{i}}^2$$

As with L1, this penalty term is added to the task-specific loss function to create the total loss, but the weight update rule will also have to be adjusted:

$$\boldsymbol{w}_{\boldsymbol{t+1}} = \boldsymbol{w}_{\boldsymbol{t}} - \eta(\nabla L_{\text{task}} - 2\lambda \cdot \boldsymbol{w}_{\boldsymbol{t}}).$$

L2 regularization will penalize large weights more heavily than small ones, but in contrast to L1 regularization, will not have the effect of driving weights to zero. Instead, it will reduce the weights using the scaled magnitude of $\boldsymbol{w}_{\boldsymbol{t}}$, resulting in smoother, smaller weights across features.

The choice between L1 and L2 regularization depends on the features in the dataset. In the case where few features are expected to influence the prediction, L1 regularization can be used. This is because it promotes sparsity by setting less important features to zero. On the other hand, if many features are expected to influence predictions, L2 regularization may yield better results by reducing the magnitude of all weights without specifically forcing them to zero (Mazilu & Iria, 2011).

2.3.3. Stabilization of Gradients

Gradients are crucial for learning as they carry the signal during backpropagation. This signal is composed of both the magnitude and the direction of the gradients and will be used to update the weights. Therefore, a key component to prevent instabilities during training will be to control the gradients. Techniques to achieve this will be discussed in the following paragraphs.

2.3.3.1. Gradient Clipping

A straightforward method to control the gradient magnitude is to limit them to a specific threshold (Goodfellow et al., 2016; Pascanu et al., 2013; Sundermeyer et al., 2012). This can be implemented in two ways: *clipping by value* and *clipping by norm*.

Clipping by value will restrict the individual gradients to lie within a minimum and maximum threshold. This method is conceptually simple and is especially useful for preventing individual gradients from introducing instability. However, it may introduce bias as it can disrupt the gradient direction, which is critical for gradient descent.

Clipping by norm will rescale the gradient to a desired norm, effectively preserving the direction of the gradient by scaling it uniformly along that direction. Mathematically, this can be represented as:

$$\boldsymbol{g} = c \cdot \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|}$$

Where $\boldsymbol{g}$ is the gradient vector and c is the clipping threshold value. Clipping by norm will respect the gradient direction, but all components of the gradient will be scaled down together per layer. This method avoids distorting the gradient direction but may lead to slower due to possibly flattening important details that might allow the model to learn faster. Additionally, because this is applied per layer, it may unnecessarily rescale layers with already small gradients, further slowing down learning and potentially exacerbating vanishing gradient issues.

While both methods offer a way to control gradients, they may result in side effects. A main consideration when using these methods is that overuse can reduce the model's capacity to learn effectively. It should also be noted that applying these techniques requires careful tuning of hyperparameters to ensure that training remains stable without compromising performance.

2.3.3.2. Adaptive Gradient Clipping

Adaptive Gradient Clipping (AGC) (Brock, De, Smith, et al., 2021) builds on the concept of regular gradient clipping but will now leverage information from the weight magnitudes to adaptively scale the gradients. This will allow the threshold to change depending on the weights and make the hyperparameter tuning less challenging. Specifically, based on the current norms of both the gradients and weights:

$$\|\boldsymbol{g}_i\| > \lambda \|\boldsymbol{w}_i\|$$

Where $\boldsymbol{g_i}$ is the gradient matrix of the ith layer, $\boldsymbol{w_i}$ is the weight matrix of the ith layer, and the clipping threshold λ is a scalar hyperparameter. λ will ensure that the gradients don’t become disproportionally large compared to the weights.

2.3.3.3. Optimizer-Based Techniques

Optimizers are one of the core components for training neural networks as they are responsible for updating the model weights to minimize the loss function. The paragraphs below will discuss popular optimizers that extend on the classical gradient descent introduced in 2.1.2.2. In general, these methods will process the raw gradients or learning rate before using them in the weight update rule.

- **Normalized Gradient Descent (NGD)**

Normalized Gradient Descent (NGD) (Cortés, 2006) is a variation of classical gradient descent optimization method. Classical gradient descent will update the weights based on a fixed learning rate and the gradient of the loss with respect to the weights. However, a potential problem for stability is that the gradient controls both the direction and magnitude of the weight updates. This may result in update steps that are either too large or too small, which affects the effective learning rate, especially as gradients tend to shrink overall during training as the loss decreases. Although this shrinking is generally beneficial for convergence, it limits control over the actual weight update step.

Normalized Gradient Descent can be used to remedy this problem by normalizing the gradients before using them to update the weights. With this approach, the gradient will only provide the direction for the update, while the learning rate controls the update step size. The weight update rule for NGD is:

$$\boldsymbol{w_{t+1}} = \boldsymbol{w_t} - \eta \frac{\nabla L(\boldsymbol{w_t})}{\|\nabla L(\boldsymbol{w_t})\|}$$

Where η is the learning rate, $L(\boldsymbol{w_t})$ is the loss with respect to the weights, and $\|\nabla L(\boldsymbol{w_t})\|$ is the norm of the gradient. It should be noted that while this process allows for better control of the update step size it will result in a loss of information of the original gradients that have now been normalized. Additionally, the learning rate will now have to be more carefully tuned as it is responsible for all update step sizes. Poorly tuned learning rates can lead to longer convergence times as it might overshoot good local minima in the loss landscape.

- **Root Mean Square Propagation (RMSprop)**

Root Mean Square Propagation (RMSprop) is an optimization technique that will adapt the learning rate based on the magnitudes of previous gradients (Tieleman & Hinton, 2012). RMSprop calculates an exponentially weighted average of past squared gradients. The average is then used during weight updates to scale the learning rate, leading to more stable updates. Using this technique will allow the neural network to avoid large, sudden fluctuations in the gradients that may cause instabilities. The weight update rule is defined as follows:

$$E[\boldsymbol{g}^2]_t = \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma)\boldsymbol{g}_t^2$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{E[\boldsymbol{g}^2]_t + \epsilon}} \boldsymbol{g_t}$$

Where $E[\boldsymbol{g}^2]_t$ is the exponentially weighted moving average of the squared gradients, γ is a hyperparameter that weighs the importance of previous steps (e.g.: $\gamma = 0.9$), η is the learning rate, and ϵ is a constant for numerical stability. It should be noted that using RMSprop will result in a decreasing effective learning rate over time. While this generally aids convergence it may also slow down training when the learning rate inevitably becomes too small.

- **Adaptive Moment Estimation (Adam)**

Adaptive Moment Estimation (Adam) is another popular optimizer that combines momentum and adaptive learning rates to improve stability and convergence speed (Kingma & Ba, 2017). Similar to RMSprop, it will keep a running average and use this to adaptively scale the learning rate during weight updates. However, here the moving averages from both past gradients and past squared gradients to update the weights as follows:

$$\boldsymbol{m_t} = \beta_1 \boldsymbol{m_{t-1}} + (1 - \beta_1)\boldsymbol{g_t}$$
$$\boldsymbol{v_t} = \beta_2 \boldsymbol{v_{t-1}} + (1 - \beta_2)\boldsymbol{g_t^2}$$
$$\boldsymbol{w_t} = \boldsymbol{w_{t-1}} - \eta \frac{\boldsymbol{m_t}}{\sqrt{\boldsymbol{v_t}} + \epsilon}$$

Where $\boldsymbol{m_t}$ and $\boldsymbol{v_t}$ represent the exponentially decayed moving averages of gradients and squared gradients respectively, β_1 and β_2 are decay rates set close to 1, η is the learning rate, and ϵ is a constant for numerical stability. It should be noted that while Adam is a very good optimizer it is very sensitive to hyperparameter tuning.

2.3.4. Combined Techniques for Stabilization

To stabilise training, multiple stabilization methods as discussed above can be used in tandem. For example, proper weight initialization may result in good initial training, but small errors can accumulate over time, leading to instabilities later in the process. Combining different stabilization techniques can lead to overall improvements and potentially mitigate drawbacks from specific techniques.

In line with the primary topic of this work, some combined techniques can allow neural networks to reduce their reliance on more conventional techniques such as batch normalization. These architectures are generally referred to as normalizer-free neural networks. It should be noted that this term does not imply the absence of all normalization. Instead, it tries to explore the possibility of removing normalization methods that introduce significant drawbacks without impacting performance. In practise, the goal is to replace batch normalization with alternative techniques that maintain stability. However, as many deep neural networks depend heavily on batch normalization to ensure stable activations, and as batch normalization comes with significant advantages, developing normalizer-free architectures is challenging. This section will explore approaches to create neural networks that achieve stable performance without explicitly performing normalization through for example batch normalization.

2.3.4.1. Self-Normalizing Neural Networks

Self-Normalizing Neural Networks (SNNs) (Klambauer et al., 2017) stabilise training through a specialized activation function called the Scaled Exponential Linear Unit (SELU), and LeCun initialization. These changes will reduce and possibly remove the need for batch normalization, which avoids increased computational cost and other disadvantages. Another factor is that SNNs allow the architecture to be simplified as the normalization is now built into the activation function, hence the name self-normalization.

A core component of SNNs is the SELU activation function, defined as follows:

$$SELU(x) = \lambda \begin{cases} x & if\ x > 0 \\ \alpha e^x - \alpha & if\ x \leq 0 \end{cases}$$

Where λ is a scaling factor, and α a parameter ensuring that the activations stay around zero mean and unit variance. By having a constant distribution, instabilities such as an uncontrollable growth for the activations can be avoided. This is in contrast with other activation functions, such as ReLU, which cause a distribution shift with each application, leading to an unbounded increase in activation variance and potential instabilities.

While SELU is responsible for maintaining zero mean and unit variance for the activations, proper initialization of the network weights is vital for using SNNs. This is because the initialization must align with the variance preserving properties of the activation function. For this purpose, LeCun initialization (Section 2.3.2.1. is used, as it ensures a stable variance across layers. It should also be noted that for more complex architectures, such as EfficientNet, which includes squeeze-and-excitation layers, modifications to the architecture may be necessary to ensure stable training. In case of dropout, a popular regularization technique, the paper addresses this by employing alpha dropout. Alpha dropout will ensure that the input and output will maintain the original mean and standard deviation.

2.3.4.2. Un-Normalized ResNets

A Normalizer-Free ResNets (NF-ResNets) was designed in the paper “Characterizing signal propagation to close the performance gap in unnormalized ResNets” (Brock, De, & Smith, 2021). As the title suggests, the authors analysed how activations propagate in ResNets and tried to replicate this without batch normalization. Key tools were signal propagation plots (SPPs), which were used to track the activation mean and variance per layer during training. Based on these observations, the ResNet architecture was modified to remove batch normalization layers while achieving stable training. Two key changes to achieve stable training were (1) a redesign of the residual blocks and (2) the use of scaled weight standardization.

To control the increasing variance within each residual block, pre-activations were downscaled within each block as follows:

$$x_{l+1} = x_l + \alpha f_l\left(\frac{x_l}{\beta_l}\right)$$

Where $\beta_l = \sqrt{var(x_l)}$, to ensure that $\frac{x_l}{\beta_l}$ has unit variance and α is a hyperparameter that determines the linear growth between residual blocks.

While this redesign effectively controls the variance, it results in the rapid growth of mean activations in deeper layers. To address this, scaled weight standardization was used. This slightly modified version of weight standardization (2.3.2.2.) introduces a scalar to ensure unit variance and effectively solves the problem of mean-shift.

To bridge the performance gap between the classical ResNet and the NF-ResNet, the authors noted that batch normalization provides not only stability but also regularization. When stochastic depth and drop out were used, NF-ResNet achieved comparable performance to standard ResNet. However, it should be noted that while NF-ResNets obtained comparable performance to regular ResNets, this was not the case for architectures such as EfficientNet due to stability problems. The authors hypothesized that this was likely due to be poor interaction between weight standardization and depth-wise convolutions, which both constrain EfficientNet's expressivity by imposing strict limitations on the convolution weights.

3. Methods

This section will describe the setup and experiments in detail. The notable parts will be (1) model and data selection, (2) visualization of neural network training, and (3) the experimental steps to create models that are less reliant on normalization techniques such as batch normalization. As mentioned above (2.3.4.), these architectures will be referred to as Normalizer-Free neural networks. It should be noted that this term does not imply the absence of all normalization. Instead, it tries to explore the possibility of removing normalization methods that introduce significant drawbacks without impacting performance and stability.

The full hardware and software specifications used for this work can be found under appendix A. In short, python was used to write the full framework for the experiments, PyTorch and Cuda were used to train the neural networks and Matplotlib was used to create the visualizations of the experiments. The full code implementation is also accessible on GitHub with the following link: https://github.com/PDewyse/MT_Normalizer_free_Neural_Nets

3.1. Model and Data Selection

For this work, EfficientNet-B0 was used due to its balance between performance and relatively low hardware requirements (M. Tan & Le, 2020). Other architecture candidates like MobileNets (A. Howard et al., 2019) and ResNets (He et al., 2015a) were considered, but EfficientNet was preferred based on (1) the more advanced architectural modules such as Squeeze-and-Excitation and Depth wise Separable Convolutions, and (2) the scalability from its lightweight variant EfficientNet-B0 to the larger EfficientNet-B5. Additionally, EfficientNet achieves state-of-the-art accuracy with batch normalization playing a crucial role in stabilizing the network during training and improving performance. This creates a significant research incentive to investigate the removal of batch normalization, while retaining EfficientNet’s high performance while deepening the understanding of batch normalization's impact on training.

For data selection, this study used the CIFAR-100 dataset (Krizhevsky, 2008). This dataset is well-established in the field of computer vision and neural network training, as it has clear documentation and has widespread use in other studies. This offers a more stable and predictable baseline for testing, especially when exploring models that may introduce instabilities and other unseen behaviours. Between CIFAR-10 and CIFAR-100, the latter was chosen for its greater number of classes and higher complexity that aligns with the learning capacity of EfficientNet.

3.2. Visualizing Signal Propagation in Neural Networks

3.2.1. Signal Propagation Plots

Signal Propagation Plots (SPPs) are a method for visualizing neural network training by plotting the hidden activations at different depths within a neural network (Brock, De, & Smith, 2021). Of note is that these plots provide a purely empirical visualization of training in real-world scenarios. Although this empirical approach can reveal patterns that are unexpected or challenging to interpret theoretically, it serves as a practical tool for diagnosing signal propagation and debugging implementation errors.

SPPs can help identify patterns that could point to future instabilities during training. This is especially important for large and deep neural networks, as unstable signals can be diagnosed and resolved earlier on during training. Thus, reducing time and computational resources for training neural networks.

To construct SPPs, Brock et al. (Brock, De, & Smith, 2021) used inputs and outputs in the shape of *NCHW* as typically seen when training CNNs. Here, *N* denotes the batch dimension, *C* denotes the channels, and *H* and *W* denote the height and width of the feature maps after convolution. The signal is then recorded using the following three metrics that are calculated from the (pre)activations.

- Average Channel Squared Mean, computed as the square of the mean across the NHW axes, and then averaged across the C axis.
- Average Channel Variance, computed by taking the channel variance across the NHW axes, and then averaging across the C axis.
- Average Channel Variance on the end of the residual branch and before merging with the skip path.

For layers such as fully connected layers, that do not have specific channels and process data in the format NF (where F represents for example flattened features), the signal is calculated by squaring the F values and then taking the mean.

It should also be noted, as stated in the paper (Brock, De, & Smith, 2021), that these statistics focus specifically on the activations and are designed to visualize the forward pass of the network.

3.2.2. Enhancing Signal Propagation Plots

For a more comprehensive view on signal stability during training, this work expanded on the design of SPPs to include not only the forward pass, but also the parameters and backward pass of the model. In practice, this involved recording data from: (1) the (pre)activations, (2) the gradients and (3) the weights. For simplicity only the Average Channel Squared Mean and the Average Channel Variance metrics for every network layer were recorded during training.

In the original framework for SPPs, the metrics were calculated and plotted per layer and for every batch. While this allowed for a very granular view of the data it was hard to generalize observations for training over the course of e.g., multiple batches or a complete epoch. This was resolved by keeping a running average that updates the statistics per layer for every batch. The running averages were then recorded and visualized in two ways:

- The signal for every layer throughout one epoch
- The signal at one specific layer during the whole training

It should be noted that the term signal refers to the calculated metrics (Average Channel Squared Mean or Variance) represented as running averages.

The implementation to obtain SPPs is split into two parts. The first is a class that can be used in a classical PyTorch training loop and handles gathering and calculating the Average Channel Squared Mean and Variance. The second part involves logging, storing, and/or plotting the statistics. This last step will vary depending on the application, in this work the data was written to text files and stored together with other data that was saved during training.

3.2.3. Testing Signal Propagation plots

To test and gain a general understanding of SPPs and their implementation, an initial test was done to view the impact of different learning rates on the stability of more simple neural network architectures. The first model was a standard Feed-Forward Neural Network (FFNN) with 10 fully connected layers and activated by ReLU. The second model was a Convolutional Neural Network (CNN) with a feature extractor containing three convolutional layers and a classifier with three fully connected layers. ReLU was used as an activation function and in the feature extractor a max pool operation was applied after the activation function.

The models were trained for 20 epochs on the CIFAR-100 dataset at the following learning rates: 0.1, 0.01, and 0.001. The full training setup included an Adam optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.999$), Kaiming initialization (fan-out), and a batch size of 512. No regularization was used. Additionally, both models were trained with and without batch normalization (BN). BN was inserted before the activation function as follows: fully connected/convolution → BN → ReLU. This configuration for BN was chosen to stay consistent with its use in the EfficientNet architecture. The motivation for testing BN was to gain a better understanding of how it can affect training in simpler models before analysing more advanced models such as EfficientNet.

After training, the recorded activations, weights, and gradients of both models will be visualized using SPPs. These results will be further discussed under Section 4.1.

3.3. Designing Normalizer-Free Neural Networks

To ensure good signal propagation in a neural network, normalization helps by regularly normalizing the signal to prevent it from growing or vanishing. However, instead of relying on normalization to constrain the signal, neural networks can be designed so that the individual components or modules do not drastically transform the signal. This removes the need for normalization at the source.

This section presents the experimental setup for two methods that implement this concept to create neural networks without explicit use of batch normalization.

3.3.1. SN-EfficientNet based on Self-Normalizing Neural Networks

Self-Normalizing EfficientNet (SN-EfficientNet) was designed based on the use of the SELU activation function and LeCun initialization, as introduced in the paper Self-normalizing neural networks (Klambauer et al., 2017). The main concepts behind this approach are discussed in more detail in Section 2.3.4.1.

However, when analysing the EfficientNet architecture certain modules still did not preserve the signal. The first was when merging the skip connections with the main path via summation. The second came from the Squeeze-and-Excitation (SE) layers, which learns to scale the activations channel-wise between convolution operations.

To limit the extent to which this sum and multiplication impacted the signal propagation, magnitude preserving modules were used. These modules were inspired by research on improving diffusion models through controlled signal propagation (Karras et al., 2024). Additionally, batch normalization was removed entirely from the architecture.

To ensure stable signal propagation, a magnitude preserving sum (MP-sum) between a skip connections and the main path is defined as follows:

$$MP - sum(\boldsymbol{x}_{main}, \boldsymbol{x}_{skip}, w) \ = \frac{(1-w)\cdot \boldsymbol{x}_{main} + w \cdot \boldsymbol{x}_{skip}}{\sqrt{(1-w)^2 + w^2}}$$

Where $\boldsymbol{x}_{skip}$ and $\boldsymbol{x}_{main}$ are the inputs from the main path and skip connection, respectively, and w is a scaler that determines the contribution of the skip connection. The denominator is the norm of the weights and will ensure that the output (pre)activations stay bounded and do not increase or decrease drastically.

To preserve the magnitude of after SE layers the scaling operation can be modified as follows:

$$\mathrm{MP-multiplication}(\boldsymbol{x}, \boldsymbol{w_{SE}}) = \frac{\boldsymbol{x} \cdot \boldsymbol{w_{SE}}}{\sqrt{(1 - \boldsymbol{w_{SE}})^2 + \boldsymbol{w_{SE}}^{\boldsymbol{2}}}}$$

Where $\boldsymbol{x}$ are the input features in a convolution operation and $\boldsymbol{w_{SE}}$ are the learned weights that scale the channels of $\boldsymbol{x}$. Again, the denominator normalizes the computed weights to prevent excessive changes in (pre)activation magnitudes.

3.3.2. UN-EfficientNet based on Un-Normalized ResNets

Un-Normalized EfficientNet (UN-EfficientNet) was designed based on Un-Normalized ResNets, as introduced by (Brock, De, & Smith, 2021). In short, the residual blocks were re-designed to scale the (pre)activations and scaled weight standardization was applied for additional stability (Section 2.3.4.2.). Additionally, as described by the authors the output of the SE-layers was also multiplied by a factor of two to maintain signal propagation.

3.3.3. Training Setup

SN-EfficientNet and UN-EfficientNet were tested against a vanilla EfficientNet-B0 under similar training conditions as described in Section 3.2.3. Both models were trained for 20 epochs on the CIFAR-100 dataset using learning rates of 0.1, 0.01, and 0.001. The full training setup included an Adam optimizer (β_1 = 0.9, β_2 = 0.999), and a batch size of 512. No regularization was used (stochastic depth or drop out). EfficientNet was also configured to train with and without BN to verify its impact. Additionally, SN-EfficientNet was trained with and without the magnitude preserving adjustments in the SE-layers and skip connections.

After training, the recorded activations, weights, and gradients of both models will be visualized using SPPs. These results will be further discussed in Section 4.2.

4. Results and Discussion

4.1. Visualizing Signal Propagation in Neural Networks

This section contains the results of the experiments described in 3.2.3. 3.2.3. A simple feedforward neural network (FFNN) and a convolutional neural network (CNN), both with and without batch normalization (BN), were trained using varying learning rates over 20 epochs. The CIFAR-100 top-1 accuracies of both models can be found in Table 3 . It should be noted that accuracies close to 1% indicate that the models did not to learn effectively, as 1% corresponds to randomly selecting the right class among the dataset's 100 classes.

Learning rate	**FFNN (no BN)**	**FFNN (BN)**	**CNN (no BN)**	**CNN (BN)**	**CNN (BN in FE)**	**CNN (BN in classifier)**
0.1	1.0 ± 0.0	8.6 ± 4.0	1.0 ± 0.0	30.1 ± 3.1	1.4 ± 0.7*	23.9 ± 2.6
0.01	2.2 ± 2.4*	19.5 ± 0.7	11.9 ± 9.1*	49.1 ± 1.2	25.9 ± 1.0	48.1 ± 1.0
0.001	19.5 ± 0.5	23.9 ± 0.3	45.8 ± 0.3	**49.9 ± 0.4**	40.1 ± 0.4	48.3 ± 0.4

*Table 3 CIFAR-100 Top-1 accuracy (%) for a Feed-Forward Neural Network (FFNN) and a Convolutional Neural Network (CNN), trained at different learning rates with and without batch normalization (BN) for 20 epochs. CNNs trained with BN only in the feature extractor (FE) or classifier are also included. Results are presented as the mean accuracy ± standard deviation across 5 random seeds. *Unstable training, one or more out of five seeds did not train.*

The experiments with different learning rates highlighted the conditions where training became unstable or failed to learn. At a relatively high learning rate of 0.1 models without BN didn't converge. In contrast, the FFNN with BN did train, but with larger variability across random seeds, suggesting unstable convergence. Reducing the learning rate to 0.01 enabled all models to train, however, FFNNs without BN showed instability, with one or more runs failing to converge. Further reducing the learning rate to 0.001 allowed all models to train stably. Of note is that using BN in the FFNN increased accuracies significantly but the opposite held for the CNN, where using BN in the feature extractor (FE) decreased the accuracy from 45.8 ± 0.3% to 40.1 ± 0.4%. As described in the experimental setup, BN was applied to the feature extractor only.

To further investigate the impact of BN on CNN performance, additional experiments were conducted using two modified configurations: (1) BN applied throughout the whole network, including the classifier, and (2) BN applied exclusively to the classifier. This led to accuracies of 49.9 ± 0.4% and 48.3 ± 0.4% respectively. Both configurations showed similar behaviours to those observed previously, but with overall better results and no failed runs for the learning rates. It should also be noted that for this architecture, BN was most effective when applied across the whole network, with BN layers in the classifier being the main contributor of the observed performance improvement.

A possible explanation for this behaviour could be that the classifier was the least stable part of the model, and thus benefitted the most from using BN. Additionally, BN may be inherently more effective for fully connected layers in this specific training scenario.

In many projects, this would be where BN is declared the overall winner despite its potential drawbacks if implemented incorrectly. However, in this experiment a deeper analysis was done using Signal Propagation Plots to explore the underlying causes of these observations. During training, the Average Channel Squared Mean and Variance of the (pre)activations, weights and gradients were recorded for every layer. This data was then used to create SPPs for the tested learning rates.

The next section will present detailed Signal Propagation Plots of the model trained with a learning rate of 0.001 (Figure 9). This serves a dual purpose of (1) establishing a stably trained baseline for observing SPPs in more advanced architectures, and (2) facilitating the interpretation and navigation of SPPs due to their high informational density. Subsequent sections will then explore the effects of adding of BN, or changing the learning rate, as observed through SPPs. For complete SPPs from all experiments, readers are referred to Appendix A.

4.1.1. Feed-Forward Neural Network

4.1.1.1. Detailed Signal Propagation Plot Analysis

This section contains a detailed analysis of training a FFNN at a learning rate of 0.001. The complete SPPs can be found in Figure 9 .

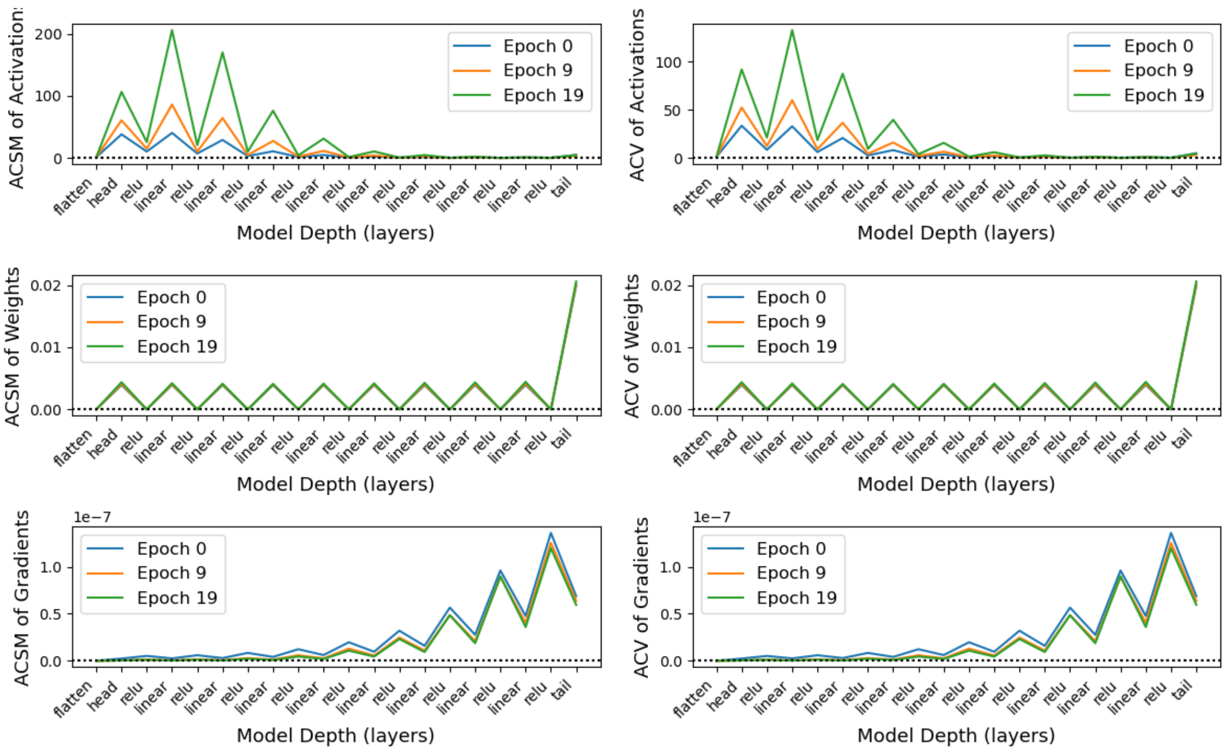


Figure 9 Signal Propagation Plots during training of the (pre)activations, weights, and gradients of a Feed-Forward Neural Network with10 fully connected layers. The signals are calculated using the Average Channel Squared Mean (ACSM) and Average Channel Squared Variance (ACV) per epoch.

- **Analysis of (Pre)activations**

The ACSM and ACV of the (pre)activations in Figure 9 (top left and right) showed that fully connected layers, denoted by *'linear'*, caused the signal to spike before it was brought down by ReLU activations. The signal also became weaker in deeper layers but did not get extinguished. Comparing the ACSM and ACV showed a strong similarity in their patterns but with different magnitudes.

The spiking behaviour of the fully connected layers can be explained by the change in weights that resulted in a general increase in (pre)activation magnitudes. For example, the increased magnitude of the second peak (Figure 9 top left) can be caused by the model *'focusing'* or increasing the weights to specific neurons. This then led to an overall increase in ACSM but also an increase ACV between neurons in that layer. The scaling down at activation layers was caused specifically by ReLU as it maps pre-activations to zero or leaves positive numbers. When the squared mean is subsequently taken for plotting purposes, ReLU layers will appear to reduce the signal. This is because computing the squared mean of the pre-activations will allow negative values to contribute to the overall magnitude whereas negative values are set to zero after a ReLU layer, resulting in a smaller magnitude.

- **Analysis of Weights**

The ACSM of the weights (Figure 9 middle left and right) stayed constant across fully connected layers and went to zero for activation layers as these do not have weights. Then on the final fully connected layer or '*tail'*, the weights spiked up. A second observation was that the ACSM and ACV showed a very similar pattern and magnitude, but comparing the exact values does reveal slight differences

All the observations for the weights indicated proper weight initialization according to Kaiming initialization with the fan-out configuration (2.3.2.1.). This means that the variance of the weights in the hidden layers at the start of training can be calculated as $\frac{2}{n_{fan-out}} = \frac{2}{n_{hidden}} = \frac{2}{512} \approx 0.004$, which is reflected in the hidden layers of Figure 9 . It also explained the spike to approximately 0.02 ($= \frac{2}{100}$) in the final layer as the model reduced its neurons to 100, each corresponding to an output class of the CIFAR100 dataset.

- **Analysis of Gradients**

Investigating ACSM and ACV of the gradients in the lower right and left plot of Figure 9 revealed an upward trend in deeper layers. This trend, when interpreted from the opposite direction, suggested that the signal diminished in the earlier layers, providing a visual representation of the vanishing gradient problem. Of note is that the signal in the earlier layers was not entirely extinguished.

- **Analysis of layers throughout training**

Up to this point, the signal has been analysed for each layer over a single epoch. However, to investigate stability, it is vital to monitor the signal throughout the whole training process, i.e. for multiple epochs. For this purpose, every plot in Figure 9 features the signal per layer recorded at three points: the start of training (epoch 0), the middle of training (epoch 9) and at the end of training (epoch 19).This revealed that for both the (pre)activations and the weights, the ACSM and ACV increased during training. In contrast, the gradients tended to decrease over the course of training.

To further investigate the extent to which these activations can grow, additional training was done for 100 epochs. During the last epoch, the largest ACSM values for the (pre)activations and weights were approximately 6550 and 0.028 respectively. While these values were well below thresholds that might cause numerical issues, larger (pre)activations could have led to reduced sensitivity during training. The larger gradients observed at the start of training are a common phenomenon as the model is initialized with random parameters, leading to larger errors and bigger updates steps.

4.1.1.2. Effect of Batch Normalization

This section contains a detailed analysis of training a FFNN with batch normalization at a learning rate of 0.001. The complete SPPs can be found in Appendix A.1.1.2.

Using BN between the fully connected layers and the non-linearities preserved the sawtooth pattern of ACSM (pre)activations observed in the absence of BN. However, the spike magnitudes in fully connected layers remained constant per epoch rather than decreasing in deeper layers.

For the weights, the signal consistently jumped to approximately 1 in all BN layers. This corresponds to the initialized value of the scale parameter $\gamma = 1$ (2.3.1.1.). Other layers saw a similar oscillating behaviour and magnitude as previously observed without BN. However, the gradient signal retained its sawtooth pattern without diminishing in earlier layers, a contrast to training without BN where the gradients decayed.

In general, BN layers affected the ACSM and ACV of (pre)activations, weights, and gradients by scaling or setting them to specific values. In the forward pass the fully connected layers were allowed to act *'freely'* before being scaled down using batch normalization. In the backward pass, the gradient signal was scaled down. These repetitive patterns consistently prevented vanishing gradients and dampened (pre)activations, which were commonly observed in the absence of BN (Figure 9).

When further examining how BN affected the backward pass (Figure 10), it was observed that during the first epoch, the ACSM and ACV decreased in deeper layers. However, in later stages during training (e.g., epochs 9 or 19) the corresponding signal magnitudes between fully connected layers was more constant. This suggested that BN might be influencing the backward pass. To investigate whether this behaviour resulted from the learnable parameters (γ and β) scaling and shifting the backward pass, the FFNN with BN was trained with a fixed $\gamma = 1$ and $\beta = 0$ (non-learnable) under the same training conditions used in experiment 3.2.3. with a learning rate of 0.001.

The experiment with fixed BN parameters resulted in an accuracy of 18.4 ± 0.2%. Compared to the accuracy results in Table 3, it still scored 1.1 ± 0.5% higher compared to the FFNN trained without BN but saw an overall decrease of 5.5 ± 0.4% compared to training with regular BN. The ACSM and ACV of (pre)activations in this experiment retained the familiar sawtooth pattern, but with larger spike magnitudes in the first layer and smaller spikes in deeper layers. While the weight signals in the fully connected layers remained unchanged, the gradient signals showed higher overall magnitudes, with dampened spikes in deeper layers. Figure 10 depicts a detailed comparison of the gradient ACSM for BN with and without learnable γ and β. During the first epoch, where the learnable parameters were still largely untrained, both configurations showed a similar decrease in ACSM. However, in later epochs with regular BN (Figure 10 left), the gradient signal was scaled down and stabilised across layers. This was not the case for BN without learnable parameters (Figure 10 right), where the magnitude of the gradients was overall higher and spiked higher in earlier layers. The complete SPP for this experiment can be found in in appendix A.1.1.2 in Figure 19 .

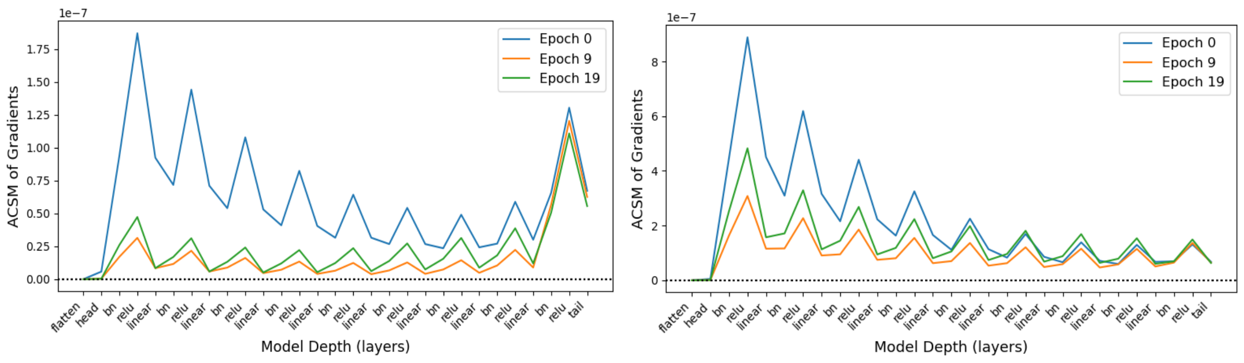


Figure 10 Signal Propagation Plots during training of the gradients of a 10-layered Feed-Forward Neural Network with BN. Left, training with batch normalization and right, with batch normalization without learnable γ and β. The signals are calculated using the Average Channel Squared Mean (ACSM) per epoch.

This analysis highlighted the dual role of the learnable parameters γ and β used by batch normalization to affect and stabilise not only the activations but also the gradients and improve performance. Training without the scale and shift parameters affected the accuracy in FFNNs and led to visual differences during the backward pass when observed through SPPs.

4.1.1.3. Effect of Learning Rates

This section contains a detailed analysis of training a FFNN with and without batch normalization at learning rates: 0.1, 0.01, and 0.001. Training results for the different configurations are summarized in Table 3 , and the complete SPPs for all configurations can be found in Appendix A.1.1.1 and A.1.1.2.

- **Training without Batch Normalization**

Training the FFNNs without BN, learning rates of 0.1 resulted in no convergence and for a learning rate of 0.01 led to inconsistent convergence. At these learning rates, the ACSM and ACV of the *(pre)activations* increased explosively, reaching values exceeding one million.

For both learning rates, the ACSM and ACV *gradient* spiked in early layers. At a learning rate of 0.1, these spikes reached magnitudes exceeding one million before eventually dampening. Towards the output layers, the signal showed a slight increase, reaching values comparable to stable training at a learning rate of 0.001 (Figure 9). At a learning rate of 0.01, a similar pattern

was observed, but the magnitude of the spike in the early layers was greatly reduced and fell within the range observed during stable training runs.
The *weights* still maintained the familiar sawtooth pattern. However, at a learning rate of 0.1, their magnitudes increased from the Kaiming-initialized ACSM of approximately 0.004 to values fluctuating around 0.2.

The *(pre)activation* signals were a useful tool for identifying instability. Extremely large or small magnitudes generally indicated problems such as exploding gradients or improper initialization. However, for the purpose of stability, this information could also have been inferred through careful monitoring of the training loss curve.
The ACSM and ACV of the *gradients* had a similar role as the (pre)activation signals, but with a focus on the backward pass instead. These signals effectively visualized the gradient flow and highlighted problematic layers, but they were not able to identify specific causes for instability.
The most reliable metrics for evaluating stability were the ACSM and ACV of the *weights*. These metrics directly influenced both the activations during the forward pass and the gradients during the backward pass. Strong deviations from the initialized weights were observed when training failed. However, this behaviour could also be attributed to the larger learning rate causing larger weight updates.

In summary, stability during training without batch normalization can be evaluated using the ACSM and ACV of the (pre)activations for the forward pass and the gradients for the backward pass. However, neither metric revealed root causes of instability. The ACSM and ACV of the weights were the most reliable stability metrics due to their central role in both the forward and backward passes.

It is important to note that in this experiment, the weights and their initialization had a significant impact on training dynamics. However, in more real-world scenarios, training would also be influenced by other hyperparameters and design choices. In this experiment, hyperparameters were intentionally kept constant to ensure that the visualization was not affected by additional confounding factors.

- **Training with Batch Normalization**

Training FFNNs with BN succeeded for all learning rates, but convergence was not consistent at a learning of 0.1. The familiar sawtooth pattern of the (pre)activation ACSM and ACV was observed for all configurations, but the magnitudes increased with higher learning rates. This behaviour was also noted in Section 4.1.1.2. , where the fully connected layers between BN could still cause the activations to spike.

Training at a learning rate of 0.01 resulted in *gradient* signals similar to those observed at a learning rate of 0.001. In contrast, training at a learning rate of 0.1 revealed smaller, irregular spikes during the early stages of training, followed by very small gradients in later stages.

Consistent with training without BN, the *weight* ACSM and ACV became more irregular at higher learning rates. The SPPs with this behaviour are depicted in Figure 11 . Notably, there was a clear growth in the ACSM and ACV for both the learnable scale parameter γ and the weights in the fully connected layers at a learning rate of 0.1 (Figure 11 , top). Lowering the learning rate to 0.01 (Figure 11 , middle), stabilised the ACSM of γ. However, the ACV still showed similar patterns, albeit with peaks of lower magnitude. In the best-performing configuration, at a learning rate of 0.001 (Figure 11 , bottom), both ACSM and ACV resulted in SPPs with the familiar sawtooth pattern. The full SPPs can be found in A.1.1.2.

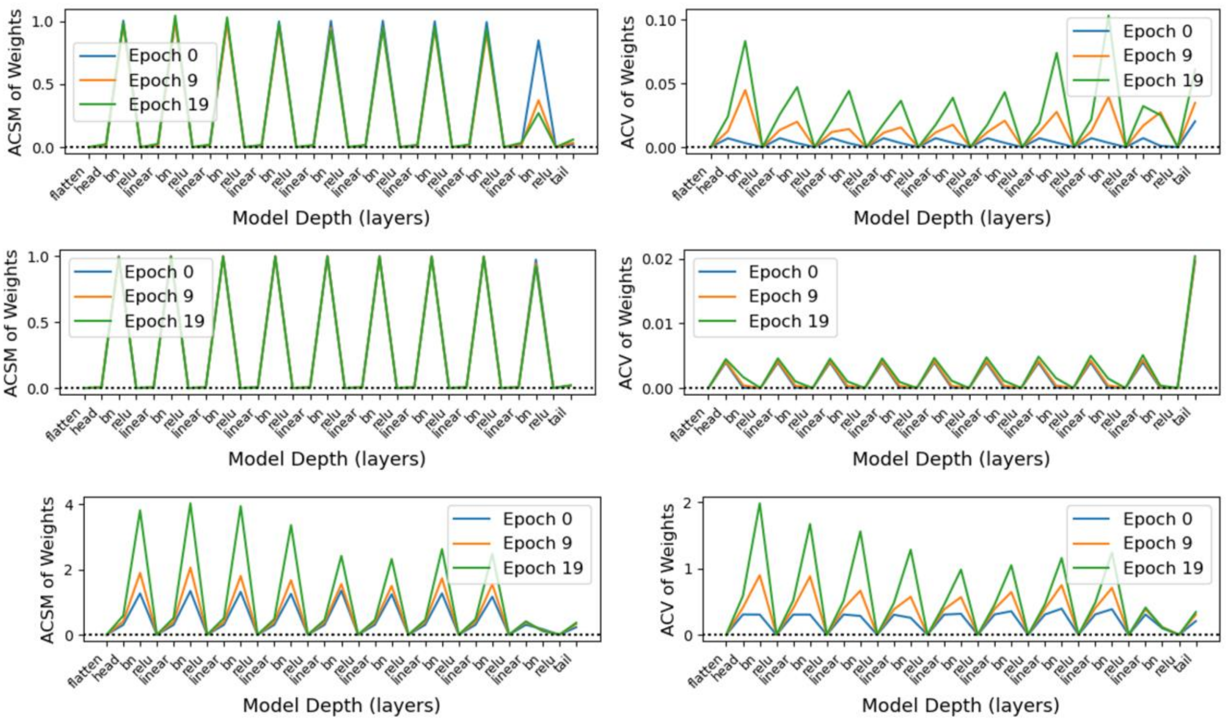


Figure 11 Signal Propagation Plots during training of the weights of a 10-layered Feed-Forward Neural Network with BN. Top, middle, and bottom plots are trained at learning rates 0.001, 0.01, and 0.1 respectively. The signals are calculated per epoch using the Average Channel Squared Mean (ACSM) on the left and Average Channel Variance (ACV) on the right.

The ACSM and ACV patterns of the (pre)activations during the forward pass were consistent with expectations of batch normalization. Regardless of the learning rate, each BN layer kept the pre-activations normalized. This effect may be especially important in preventing activations from growing excessively large, which could reduce the sensitivity of the model. For stability, tracking the (pre)activations was a suboptimal choice due to the frequent normalization of BN potentially supressing symptoms of instability. This was also the case for the ACSM and ACV of the gradients, where BN was shown to impact the gradient flow (Figure 10).

Similar to training without BN, the weights were an important indicator of stability due to their behaviour of growing faster at a learning rate of 0.1. Analysing the γ parameter provided additional insights into BN's role in stabilization. The ACV of γ increased along with the weights of the fully connected layers, and in more extreme cases, the ACSM of γ was also affected. This behaviour could indicate that BN was attempting to stabilise training by counteracting weight growth or scaling gradients.

In summary for training with BN, the forward and backward passes were visualized using the ACSM and ACV of the (pre)activations and gradients. However, these metrics did not reveal the root causes of instability. In contrast, the ACSM and ACV of the weights in fully connected layers can be used to identify problems such as large deviations from the initialized weights. The γ parameter in BN layers was the most indicative metric of instability due to its fluctuations in ACV and ACSM during training.

4.1.2. Convolutional Neural Networks

The Signal Propagation plots (SPPs) for training the different Convolutional Neural Networks (CNNs) at a learning rate of 0.001 are presented in Appendix A.1.2.

The SPPs effectively visualized signal propagation in CNNs and reflected patterns similar to those observed in Feedforward Neural Networks (FFNNs). This included the general profile of the recorded signals and their behaviour when increasing the learning rate, both with and without batch normalization. However, in case of CNNs, when linear layers and convolutional layers were plotted together, the measured ACSM and ACV signal ranges varied substantially due to the inherently different number of learnable parameters. Of note was that this variability is a potential drawback for SPPs when visualizing models that use different layer types or an increasing number of layers.

For stability analysis, the SPPs were used similarly to the experiments on FFNNs. Larger spikes or abrupt fluctuations were strong indicators of instability. However, comparing SSP profiles between different architectures was more complex. Generally, models with similar structures exhibited recognizable patterns, but identifying the exact causes of these patterns proved difficult due to largely unknown influence of batch normalization on signal propagation.

Overall, SPPs effectively visualized the training process of CNNs. However, with increasing model complexity, identifying key behaviours for instability in neural networks became more challenging.

4.2. Designing Normalizer-Free Neural Networks

This section contains the results of the experiments described in 3.2.3. SN-EfficientNet and UN-EfficientNet were tested against a vanilla EfficientNet-B0. All models were trained using varying learning rates over 20 epochs. EfficientNet-B0 was trained with and without batch normalization (BN), while SN-EfficientNet was trained without and without the magnitude preserving modules (MP) as described in Section 3.3.1. The CIFAR-100 top-1 accuracies of both models can be found in Table 3 . It should be noted that accuracies close to 1% indicate that the models did not to learn effectively, as 1% corresponds to randomly selecting the right class among the dataset's 100 classes.

Learning rate	**EfficientNet (No BN)**	**EfficientNet (BN)**	**SN-EfficientNet (No MP)**	**SN-EfficientNet (MP)**	**UN-EfficientNet**
0.1	1.0 ± 0.0	1.2 ± 0.4	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
0.01	1.0 ± 0.0	13.9 ± 3.4	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
0.001	1.0 ± 0.0	**33.5 ± 0.8**	30.0 ± 0.2	29.2 ± 0.5	1.0 ± 0.0

Table 4 CIFAR-100 Top-1 accuracy (%) after training EfficientNet-B0 (with and without Batch Normalization (BN)), Self-Normalizing (SN) EfficientNet (With and without magnitude preserving modules (MP))), and Un-Normalized (UN) EfficientNet for 20 epochs without regularization. Results are presented as the mean accuracy ± standard deviation across 5 random seeds.

Training EfficientNet-B0 without batch normalization was unsuccessful regardless of the learning rate. However, adding BN allowed training, with the highest top-1 accuracy reaching 33.5 ± 0.8% at a learning rate of 0.001. SN-EfficientNet, using the first normalization-free configuration with SELU activations, trained successfully both with and without the MP modules, achieving top-1 accuracies of 29.2 ± 0.5 and 30.0 ± 0.2 respectively, at a learning rate of 0.001.

The second normalization-free architecture, UN-EfficientNet, using the un-normalized principles, failed to train even after extensive fine tuning of the training hyperparameters, the model specific hyperparameter α, and adjustments based on the original paper’s recommendations for adapting EfficientNets (Brock, De, & Smith, 2021).

The performance difference between EfficientNet-B0 with BN and SN-EfficientNet can be attributed to the inherent regularization effects of BN described above and the regularization due to batch statistics. For SN-EfficientNet, using the MP modules resulted in a small accuracy loss from 30.0 ± 0.2% to 29.2 ± 0.5. One possible explanation for this was that the MP modules constrained the original signal from the layers, which may have been crucial for proper learning. Signal Propagation Plots (SPPs) will be used to visualize and potentially explain this behaviour in more detail.

The following sections will present a different look on training models using Signal Propagation Plots (SPPs). For completeness, additional SPPs for all training configurations can be found in Appendix A.

4.2.1. Self-Normalizing EfficientNet

This section contains the analysis of training Self-Normalizing (SN) EfficientNet at a learning rate of 0.001. The full SPPs for training vanilla EfficientNet and SN-EfficientNet (without MP modules) can be found in Figure 12 and Figure 13 below.

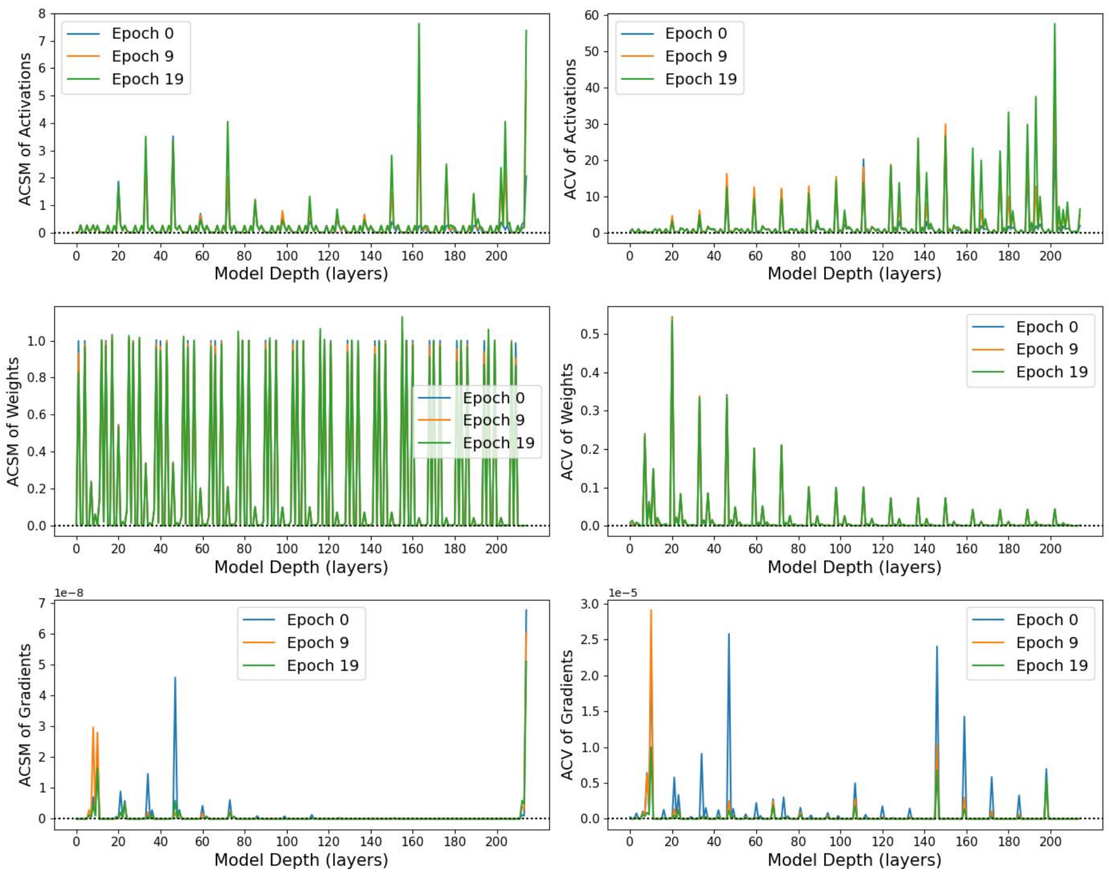


Figure 12 Signal Propagation Plots during training of EfficientNet. The signals are calculated using the Average Channel Squared Mean (ACSM) and Average Channel Squared Variance (ACV) per epoch.

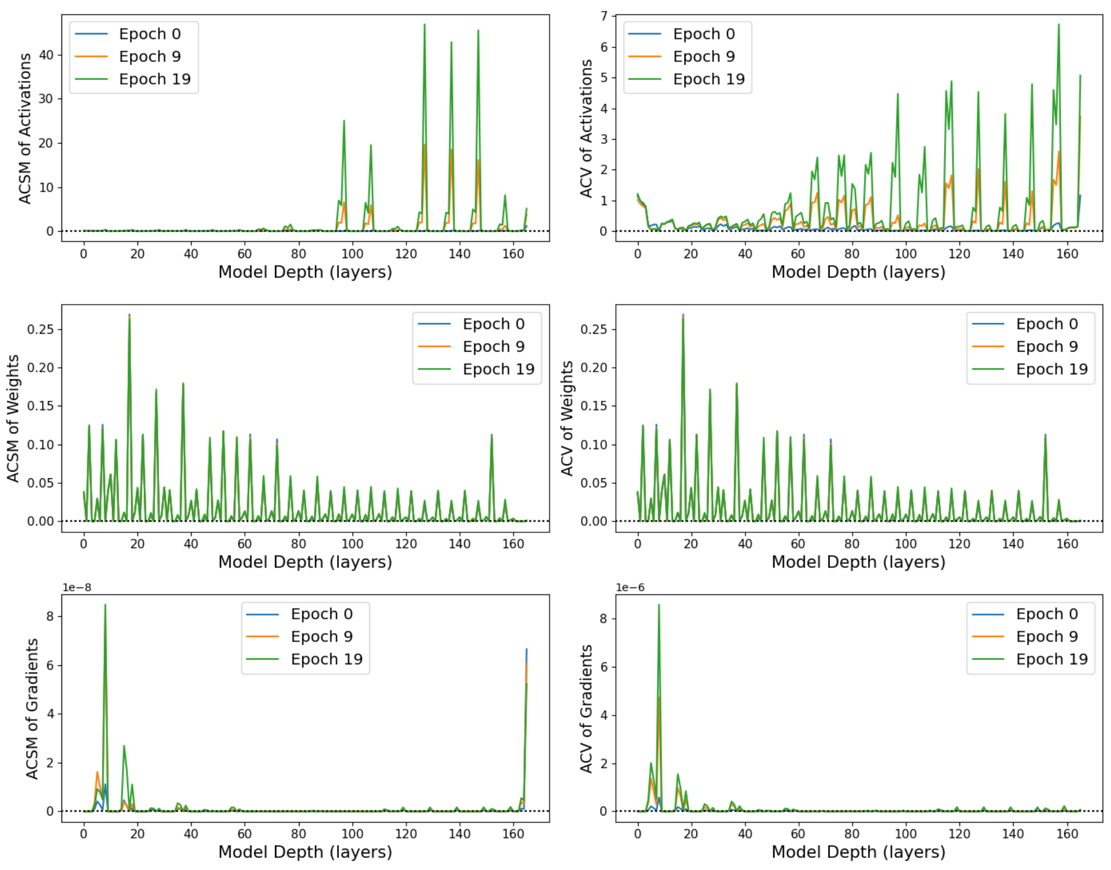


Figure 13 Signal Propagation Plots during training of Self-Normalizing (SN) EfficientNet. The signals are calculated using the Average Channel Squared Mean (ACSM) and Average Channel Squared Variance (ACV) per epoch.

4.2.1.1. Detailed Signal Propagation Plots Analysis

To improve the SPP readability for larger models, layers are now numbered. For a more detailed layer-by-layer analysis of shallower neural networks, the reader is referred to Section 4.1.

- **Analysis of (Pre)activations**

For vanilla EfficientNet, no trend of exploding or vanishing signals was observed in the ACSM of the (pre)activations, with the signal magnitudes remaining between zero and eight. Notably, consistent spikes occurred during squeeze-and-excitation (SE), and more specifically, in the reduction layer which acted to squeeze the signal or convolution channels. This spike can be attributed to the reduction layer's increased activity when mapping multiple channels to a smaller number of output units. In general, signal magnitudes remained well within control posing no risk of numerical instability. Additionally, BN brought the (pre)activation ACSM close to zero, as expected.

When comparing the (pre)activation ACV and ACSM, signal magnitudes showed a general upward trend in deeper layers and were generally larger, ranging between zero and sixty. Again, spikes were present in the SE modules. However, a clear turning point was observed around layer

90, after which spikes also appeared in the convolutional layers following the SE modules. While these additional spikes did not cause severe problems such as numerical problems, they could indicate an early sign of instability. Additionally, it should be noted that BN sets the ACV signal to one as expected.

Analysing the SPPs for SN-EfficientNet (without MP modules) revealed a different behaviour. The ACSM of the (pre)activations remained close zero, as expected, due to the self-normalizing principles discussed in Section 2.3.4.1. However, in earlier layers, small spikes appeared due to the sigmoid function at the end of the squeeze-and-excitation (SE) module. This occurred because a zero input is mapped to 0.5 by a sigmoid function and to 0.25 when measured as ACSM. In deeper layers, the entire SE module exhibited elevated signal magnitudes, indicating that the self-normalizing properties of SELU non-linearities may fail when the input signal deviates too far from zero.

Comparing the overall ACSM (pre)activation signal magnitudes of the spikes between vanilla EfficientNet and SN-EfficientNet showed that while SN-EfficientNet stayed closer to zero, the actual spikes were larger, ranged from 0 to 50.

For the (pre)activation ACV in SN-EfficientNet, the overall signal increased in deeper layers, but its magnitude ranged between 0 and 7, which was lower compared to vanilla EfficientNet. The largest spikes were measured within the SE modules, but did not propagate to the convolutional layers outside SE, unlike in vanilla EfficientNet. This suggested that avoiding BN for signal stabilization can be effective, especially since the layers between BN operations were now inherently designed for stable signal propagation. This was in contrast to regular EfficientNet, where no measures were taken to regulate the signal between BN layers.

- **Analysis of Weights**

The weights for vanilla Efficient and SN-EfficientNet were initialized differently, resulting in different patterns in the SPPs. However, in both models, deeper layers had smaller ACSM and ACV values for the weights due to an increasing number of parameters, which led to smaller average weights as determined by their initialization. For a more detailed analysis of this behaviour, and the common patterns of weights in SPPs, the reader is referred to Section 4.1.1.1. Additionally, BN layers caused spikes in the ACSM of weights, reaching one, due to the scale parameter being initialized at $\gamma = 1$ (2.3.1.1.).

- **Analysis of Gradients**

Compared to earlier experiments on simpler neural networks, the ACSM and ACV of the gradient signals showed the biggest differences. In contrast, to the clear patterns typically associated with vanishing gradients, no consistent trend was observed in vanilla EfficientNet or SN-EfficientNet. However, the spikes in the SE modules were reflected by the spikes in the SE layers during the forward pass.

In the SPPs, both models had numerous layers with ACSM and ACV values of zero, indicating poor backpropagation and possibly inefficient use of those layers. However, this issue was more prevalent in SN-EfficientNet, suggesting a possible cause for its reduced performance compared to vanilla EfficientNet.

- **Analysis of layers throughout training**

During training, (pre)activations grew but showed no signs of exploding or causing severe side effects such as numerical instability. Weights fluctuated but remained stable and deviated only slightly from their initialization in both models. For the ACSM and ACV gradient signals, vanilla exhibited a decrease in magnitude during the later stages of training, whereas SN showed an increase.

The behaviour for the (pre)activation and weight signals throughout training reinforced the observations made in smaller neural networks and was discussed in depth in Section 4.1.1.1. However, for the gradient signals, identifying the causes of these changes proved challenging due to significant differences between the two models. The most notable difference was the absence of batch normalization (BN) in SN-EfficientNet and its impact on the backward pass (Section 4.1.1.2.). Additionally, the models used different non-linearities and weight initialization strategies, resulting in distinct starting points within the loss landscape.

Overall, signal propagation in both vanilla EfficientNet and SN-EfficientNet was stable, with no layers exhibiting exploding, vanishing, or highly unstable signals. However, some layers showed weak gradient activity, suggesting that there is still room for improvement in optimizing layer utilization in these models. Additionally, SPPs were able to accurately depict the signal propagation, a major limitation was that architectural differences between models led to significant variations in signal behaviour, particularly for the gradient signals.

4.2.1.2. Magnitude Preserving Modules

The use of Magnitude Preserving (MP) modules generally did not affect signal propagation for (pre)activations, weights, or gradients. Signal magnitudes changed slightly, but general patterns observed without MP modules remained. This was expected, as MP modules mainly served to bound extreme fluctuations, which were not present during training.

4.2.2. Un-Normalized EfficientNet

As discussed in Table 4 Un-Normalized EfficientNet failed to train. This was also reflected in the SPPs, where (pre)activations signals quickly converged to zero ACV. The weights remained at their initialized values and the gradient signals also vanished quickly starting from the last layer. For reference, the full SPPs of Un-Normalized EfficientNet have been added in Appendix A.1.5.

5. Conclusion

Signal Propagation Plots (SPPs) first introduced by Brock et. al (Brock, De, & Smith, 2021), were adapted to capture and visualize the forward pass through (pre)activations, the backward pass through the gradients and the parameters through the weights of the model layers. This allowed for the identification of potentially problematic layers contributing to instability, as well as the visualization of how common deep learning techniques, such as Batch Normalization (BN), behaved and interact with the forward and backward passes during training.

During baseline testing of SPPs on a simple feed-forward neural network, BN was observed to actively scale gradients in the backward pass during training using its learnable parameters. This finding provided a potential explanation for the inherent regularization effect of BN during training.

EfficientNet-B0 was successfully designed without BN using the self-normalizing principles described by Klambauer et. al (Klambauer et al., 2017). This approach utilized SELU non-linearities to maintain a signal with zero mean and unit variance. However, this came at a performance cost.

Using SPPs, the signal mean was observed to remain near zero as expected, but the variance fluctuated between zero and one. Exceptions to this were found in Squeeze and excitations blocks, where spikes exceeding zero mean and unit variance were observed.

SPPs have considerable room for improvement, particularly in the field of explainable AI by providing insight into important layers and aiding the research of novel architectures or regularization techniques. Additionally, SPPs may also be useful for investigating signal propagation between model training and during inference. While stable training without normalization is possible, regularization is critical for competitive performance. For this purpose, future work could focus on designing novel regularization techniques that mimic how BN is able to influence both the forward and backward pass using its learnable parameters.

6. References

Aliferis, C., & Simon, G. (2024). Overfitting, Underfitting and General Model Overconfidence and Under-Performance Pitfalls and Best Practices in Machine Learning and AI. In G. J. Simon & C. Aliferis (Eds.), *Artificial Intelligence and Machine Learning in Health Care and Medical Sciences: Best Practices and Pitfalls* (pp. 477–524). Springer International Publishing. https://doi.org/10.1007/978-3-031-39355-6_10

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1), 53. https://doi.org/10.1186/s40537-021-00444-8

Arcuschin, C. D., Pinkasz, M., & Schor, I. E. (2023). Mechanisms of robustness in gene regulatory networks involved in neural development. *Frontiers in Molecular Neuroscience*, *16*. https://doi.org/10.3389/fnmol.2023.1114015

Arendt, D., Bertucci, P. Y., Achim, K., & Musser, J. M. (2019). Evolution of neuronal types and families. *Current Opinion in Neurobiology*, *56*, 144–152. https://doi.org/10.1016/j.conb.2019.01.022

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016, July 21). *Layer Normalization*. arXiv.Org. https://arxiv.org/abs/1607.06450v1

Baba, A. (2024). Neural networks from biological to artificial and vice versa. *Biosystems*, *235*, 105110. https://doi.org/10.1016/j.biosystems.2023.105110

Balasubramanian, V. (2021). Brain power. *Proceedings of the National Academy of Sciences of the United States of America*, *118*(32), e2107022118. https://doi.org/10.1073/pnas.2107022118

Bazzari, A. H., & Parri, H. R. (2019). Neuromodulators and Long-Term Synaptic Plasticity in Learning and Memory: A Steered-Glutamatergic Perspective. *Brain Sciences*, *9*(11), 300. https://doi.org/10.3390/brainsci9110300

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166. IEEE Transactions on Neural Networks. https://doi.org/10.1109/72.279181

Brock, A., De, S., & Smith, S. L. (2021). *Characterizing signal propagation to close the performance gap in unnormalized ResNets* (arXiv:2101.08692). arXiv. http://arxiv.org/abs/2101.08692

Brock, A., De, S., Smith, S. L., & Simonyan, K. (2021). *High-Performance Large-Scale Image Recognition Without Normalization* (arXiv:2102.06171). arXiv. http://arxiv.org/abs/2102.06171

Cortés, J. (2006). Finite-time convergent gradient flows with applications to network consensus. *Automatica*, *42*(11), 1993–2000. https://doi.org/10.1016/j.automatica.2006.06.015

Dai, D., Tan, W., & Zhan, H. (2017). *Understanding the Feedforward Artificial Neural Network Model From the Perspective of Network Flow* (arXiv:1704.08068). arXiv. https://doi.org/10.48550/arXiv.1704.08068

Dubey, S. R., Singh, S. K., & Chaudhuri, B. B. (2022). *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark* (arXiv:2109.14545). arXiv. https://doi.org/10.48550/arXiv.2109.14545

Gholamalinezhad, H., & Khosravi, H. (2020). *Pooling Methods in Deep Neural Networks, a Review* (arXiv:2009.07485). arXiv. https://doi.org/10.48550/arXiv.2009.07485

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–256. https://proceedings.mlr.press/v9/glorot10a.html

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org

Gregor, K., & LeCun, Y. (2010). *Emergence of Complex-Like Cells in a Temporal Product Network with Local Receptive Fields* (arXiv:1006.0448). arXiv. https://doi.org/10.48550/arXiv.1006.0448

He, K., Zhang, X., Ren, S., & Sun, J. (2015a). *Deep Residual Learning for Image Recognition* (arXiv:1512.03385). arXiv. http://arxiv.org/abs/1512.03385

He, K., Zhang, X., Ren, S., & Sun, J. (2015b, February 6). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv.Org. https://arxiv.org/abs/1502.01852v1

Hoffer, E., Hubara, I., & Soudry, D. (2018). *Train longer, generalize better: Closing the generalization gap in large batch training of neural networks* (arXiv:1705.08741). arXiv. http://arxiv.org/abs/1705.08741

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017, April 17). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv.Org. https://arxiv.org/abs/1704.04861v1

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., & Adam, H. (2019). *Searching for MobileNetV3* (arXiv:1905.02244; Version 5). arXiv. http://arxiv.org/abs/1905.02244

Hu, J., Shen, L., Albanie, S., Sun, G., & Wu, E. (2019). *Squeeze-and-Excitation Networks* (arXiv:1709.01507). arXiv. http://arxiv.org/abs/1709.01507

Ioffe, S., & Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* (arXiv:1502.03167). arXiv. http://arxiv.org/abs/1502.03167

J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li, & Li Fei-Fei. (2009). ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

Karras, T., Aittala, M., Lehtinen, J., Hellsten, J., Aila, T., & Laine, S. (2024). *Analyzing and Improving the Training Dynamics of Diffusion Models* (arXiv:2312.02696). arXiv. http://arxiv.org/abs/2312.02696

Kingma, D. P., & Ba, J. (2017). *Adam: A Method for Stochastic Optimization* (arXiv:1412.6980). arXiv. https://doi.org/10.48550/arXiv.1412.6980

Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). *Self-Normalizing Neural Networks* (arXiv:1706.02515). arXiv. http://arxiv.org/abs/1706.02515

Krizhevsky, A. (2008). *Learning Multiple Layers of Features from Tiny Images*. 60.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, *25*. https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K.-R. (1998). Efficient BackProp. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural Networks: Tricks of the Trade: Second Edition* (pp. 9–48). Springer. https://doi.org/10.1007/978-3-642-35289-8_3

Mazilu, S., & Iria, J. (2011). L1 vs. L2 Regularization in Text Classification when Learning from Labeled Features. *2011 10th International Conference on Machine Learning and Applications and Workshops*, *1*, 166–171. https://doi.org/10.1109/ICMLA.2011.85

Ng, A. Y. (2004). Feature selection, L1 vs. L2 regularization, and rotational invariance. *Proceedings of the Twenty-First International Conference on Machine Learning*, 78. https://doi.org/10.1145/1015330.1015435

PA, S. (2020, June 11). An Overview on MobileNet: An Efficient Mobile Vision CNN. *Medium*. https://medium.com/@godeep48/an-overview-on-mobilenet-an-efficient-mobile-vision-cnn-f301141db94d

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). *On the difficulty of training Recurrent Neural Networks* (arXiv:1211.5063). arXiv. http://arxiv.org/abs/1211.5063

Patterson, D., Gonzalez, J., Hölzle, U., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., & Dean, J. (2022, April 11). *The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink*. arXiv.Org. https://arxiv.org/abs/2204.05149v1

Qiao, S., Wang, H., Liu, C., Shen, W., & Yuille, A. (2019, March 25). *Micro-Batch Training with Batch-Channel Normalization and Weight Standardization*. arXiv.Org. https://arxiv.org/abs/1903.10520v2

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). *Searching for Activation Functions* (arXiv:1710.05941). arXiv. https://doi.org/10.48550/arXiv.1710.05941

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2019). *MobileNetV2: Inverted Residuals and Linear Bottlenecks* (arXiv:1801.04381). arXiv. http://arxiv.org/abs/1801.04381

Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. *Neural Networks*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Stampanoni Bassi, M., Iezzi, E., Gilio, L., Centonze, D., & Buttari, F. (2019). Synaptic Plasticity Shapes Brain Connectivity: Implications for Network Topology. *International Journal of Molecular Sciences*, *20*(24), 6193. https://doi.org/10.3390/ijms20246193

Sundermeyer, M., Schlüter, R., & Ney, H. (2012). *LSTM neural networks for language modeling*. 194–197. https://doi.org/10.21437/Interspeech.2012-65

Tan, H. H., & Lim, K. H. (2019). Vanishing Gradient Mitigation with Deep Learning Neural Network Optimization. *2019 7th International Conference on Smart Computing & Communications (ICSCC)*, 1–4. https://doi.org/10.1109/ICSCC.2019.8843652

Tan, M., & Le, Q. V. (2020). *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks* (arXiv:1905.11946). arXiv. https://doi.org/10.48550/arXiv.1905.11946

Terven, J., Cordova-Esparza, D. M., Ramirez-Pedraza, A., Chavez-Urbiola, E. A., & Romero-Gonzalez, J. A. (2024). *Loss Functions and Metrics in Deep Learning* (arXiv:2307.02694). arXiv. https://doi.org/10.48550/arXiv.2307.02694

Thau, L., Reddy, V., & Singh, P. (2024). Anatomy, Central Nervous System. In *StatPearls*. StatPearls Publishing. http://www.ncbi.nlm.nih.gov/books/NBK542179/

Tieleman, T., & Hinton, G. (2012). *Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude.* https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2016, July 27). *Instance Normalization: The Missing Ingredient for Fast Stylization*. arXiv.Org. https://arxiv.org/abs/1607.08022v3

Wu, Y., & He, K. (2018, March 22). *Group Normalization*. arXiv.Org. https://arxiv.org/abs/1803.08494v3

Y. Lecun, C. Cortes, & J. C. Burges Christopher. (1994). *MNIST handwritten digit database*. THE MNIST DATABASE of Handwritten Digits. https://yann.lecun.com/exdb/mnist/

Y. Lecun, L. Bottou, Y. Bengio, & P. Haffner. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Yamany, W., Fawzy, M., Tharwat, A., & Hassanien, A. E. (2015). Moth-flame optimization for training Multi-Layer Perceptrons. *2015 11th International Computer Engineering Conference (ICENCO)*, 267–272. https://doi.org/10.1109/ICENCO.2015.7416360

Zhang, A. and L., Zachary, C. and L., Mu and Smola, & Alexander, J. (2023). *Dive into Deep Learning*. Cambridge University Press. https://D2L.ai

Zhang, H., Dauphin, Y. N., & Ma, T. (2019). *Fixup Initialization: Residual Learning Without Normalization* (arXiv:1901.09321). arXiv. http://arxiv.org/abs/1901.09321

Zhang, J. (2019a). *Basic Neural Units of the Brain: Neurons, Synapses and Action Potential* (arXiv:1906.01703). arXiv. https://doi.org/10.48550/arXiv.1906.01703

Zhang, J. (2019b). *Gradient Descent based Optimization Algorithms for Deep Learning Models Training* (arXiv:1903.03614). arXiv. https://doi.org/10.48550/arXiv.1903.03614

Zoph, B., & Le, Q. V. (2016, November 5). *Neural Architecture Search with Reinforcement Learning*. arXiv.Org. https://arxiv.org/abs/1611.01578v2

7. Appendix

Contents

A. Additional Signal Propagation Plots

A.1.1. Feed-Forward Neural Networks

A.1.1.1. Without Batch Normalization

The reader can find the SPPs for training a 10-layered FFNN for 20 epochs at learning rate 0.001 as Figure 9 under Section 4.1.1.1.

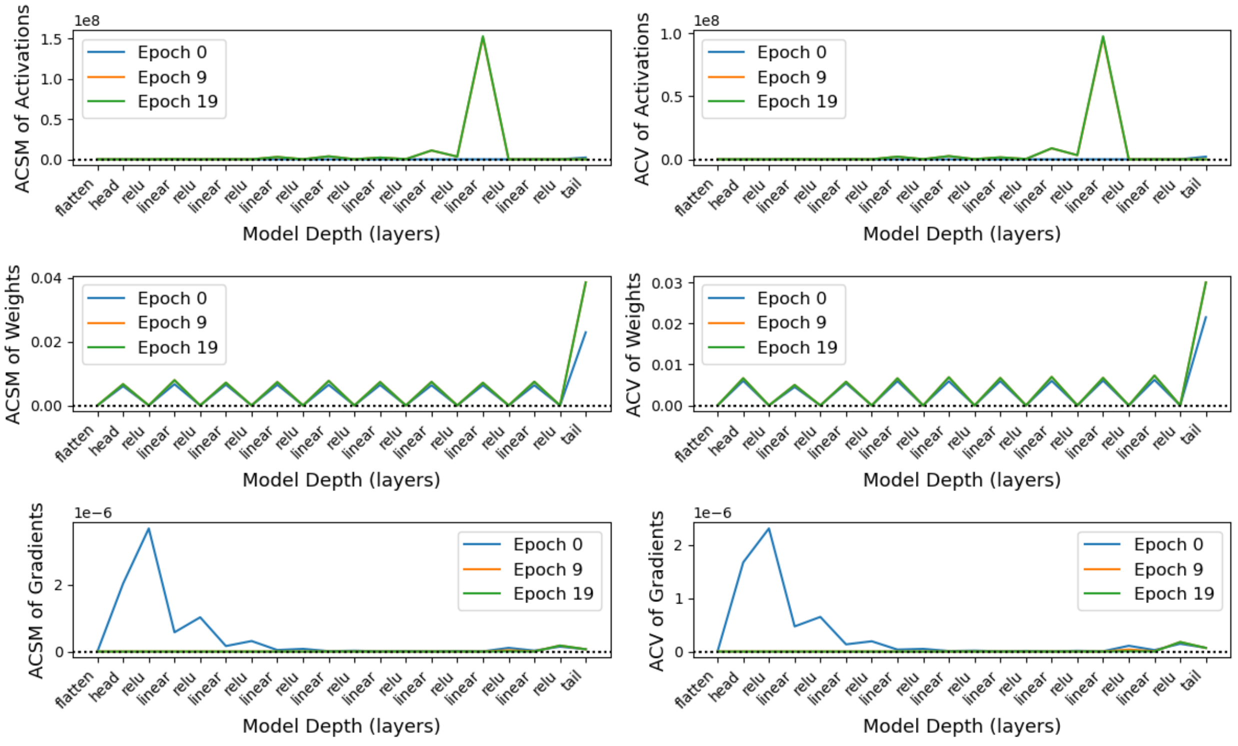


Figure 14 SPPs for training a 10-layered FFNN for 20 epochs at a learning rate of 0.01

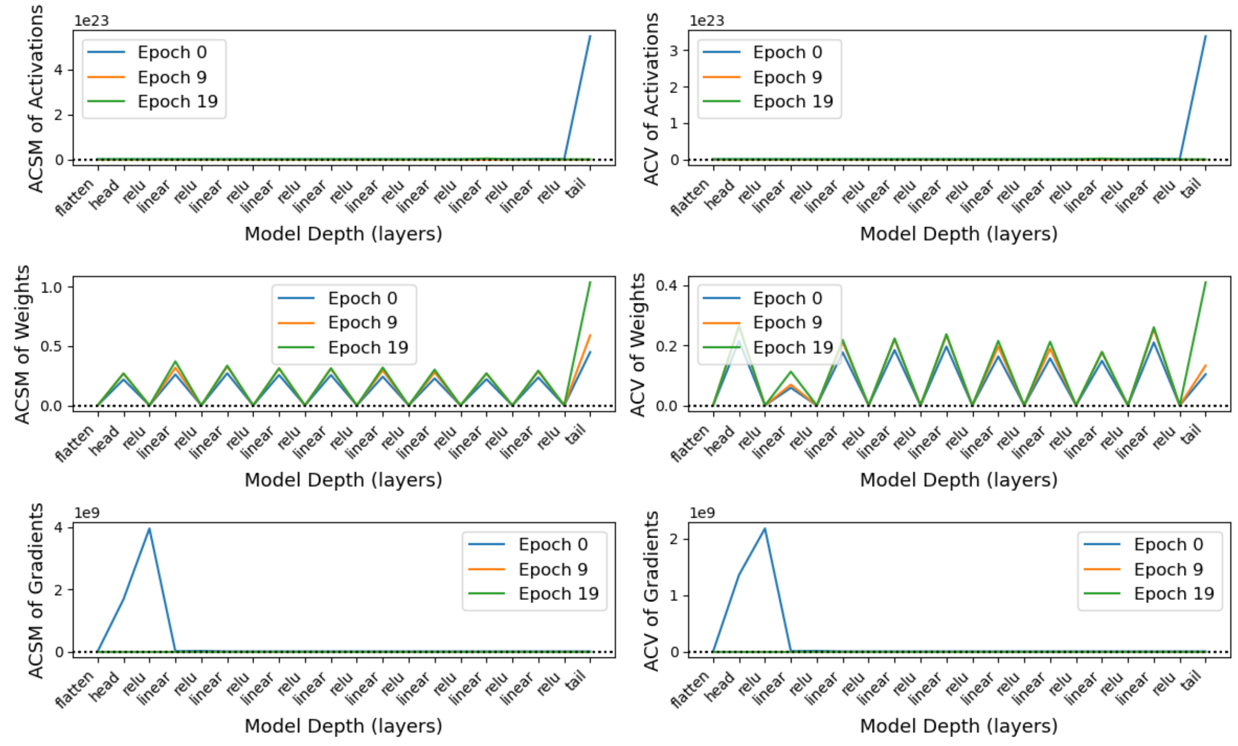


Figure 15 SPPs for training a 10-layered FFNN for 20 epochs at a learning rate of 0.1

A.1.1.2. Batch Normalization in all layers

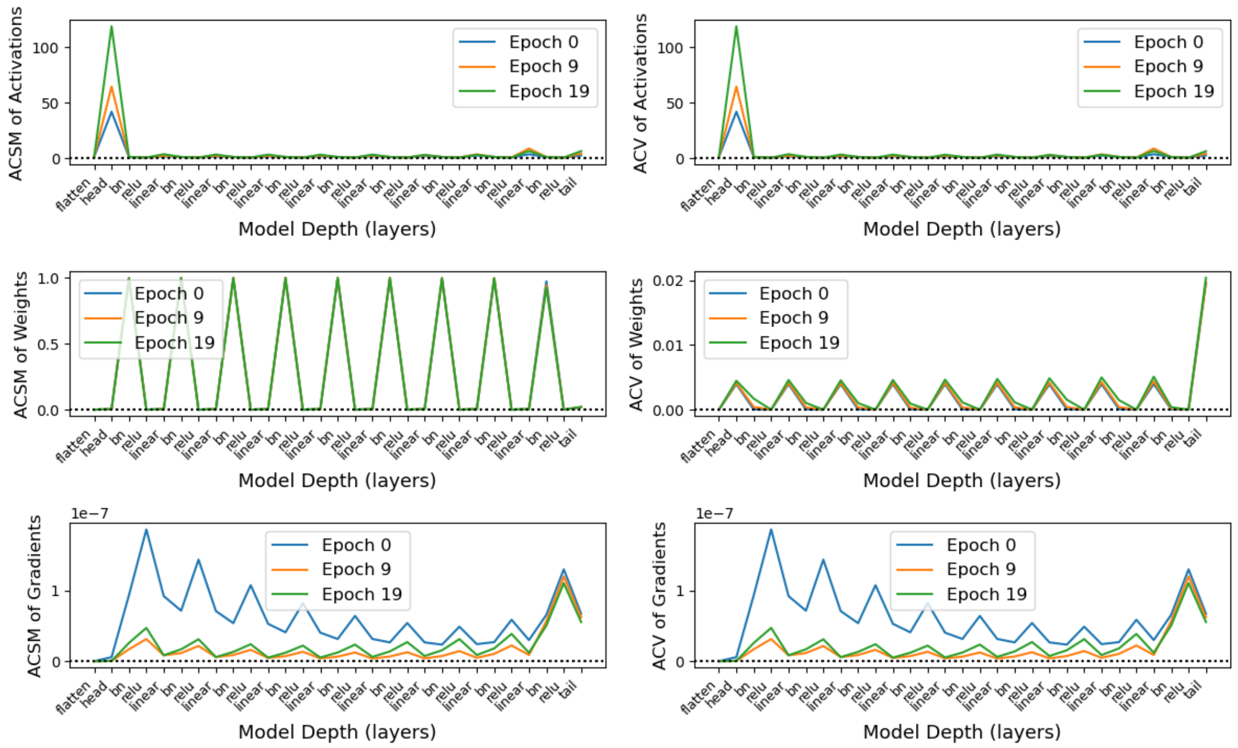


Figure 16 SPPs for training a 10-layered FFNN with BN for 20 epochs at a learning rate of 0.001

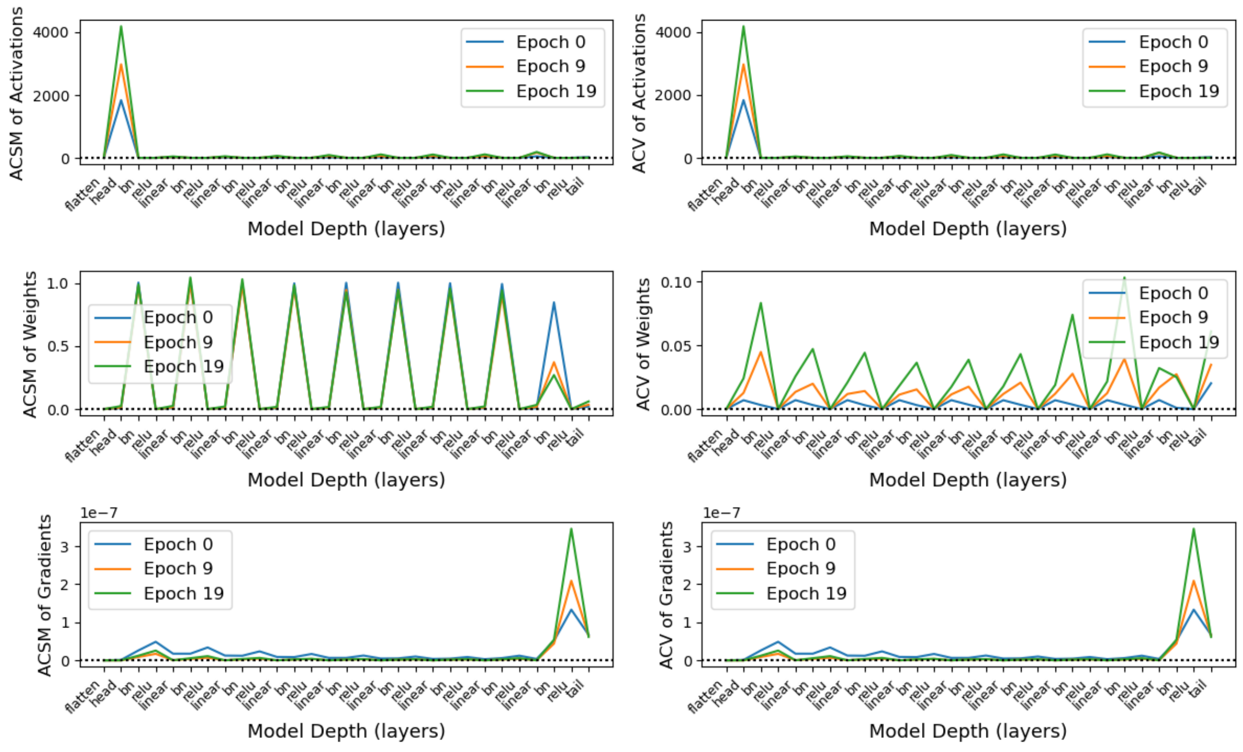


Figure 17 SPPs for training a 10-layered FFNN with BN for 20 epochs at a learning rate of 0.01

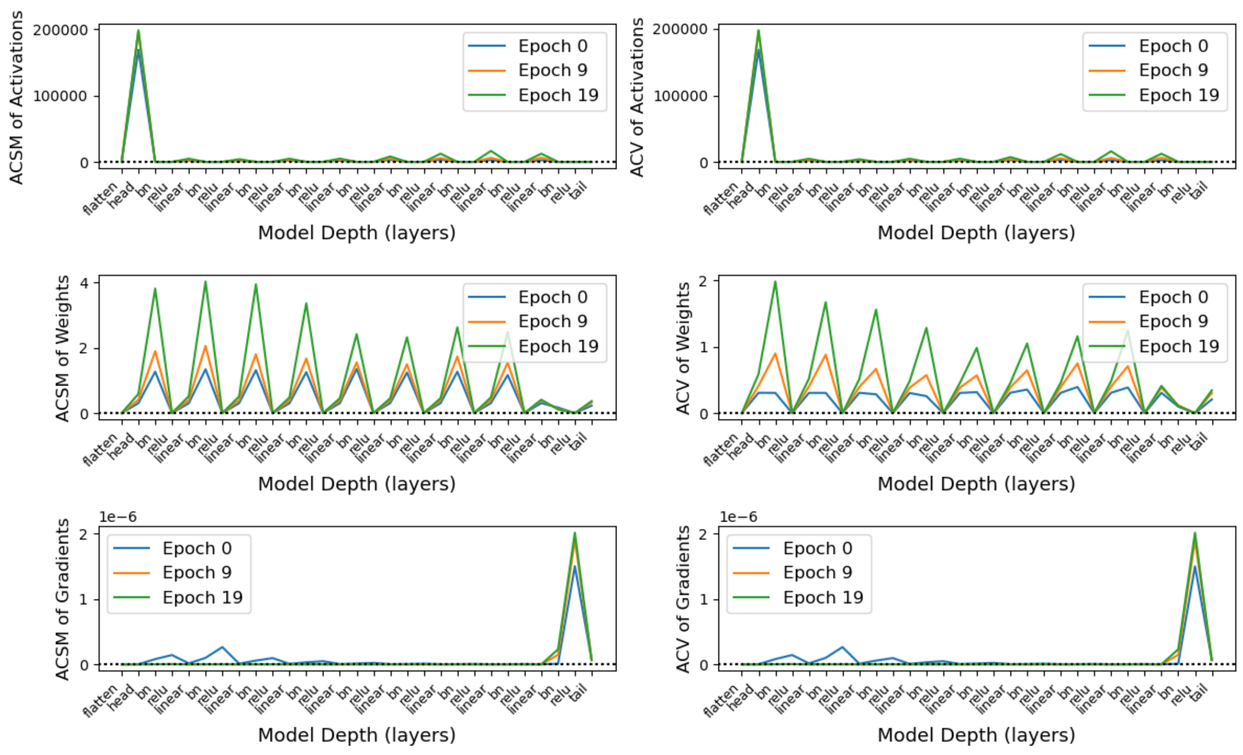


Figure 18 SPPs for training a 10-layered FFNN with BN for 20 epochs at a learning rate of 0.1

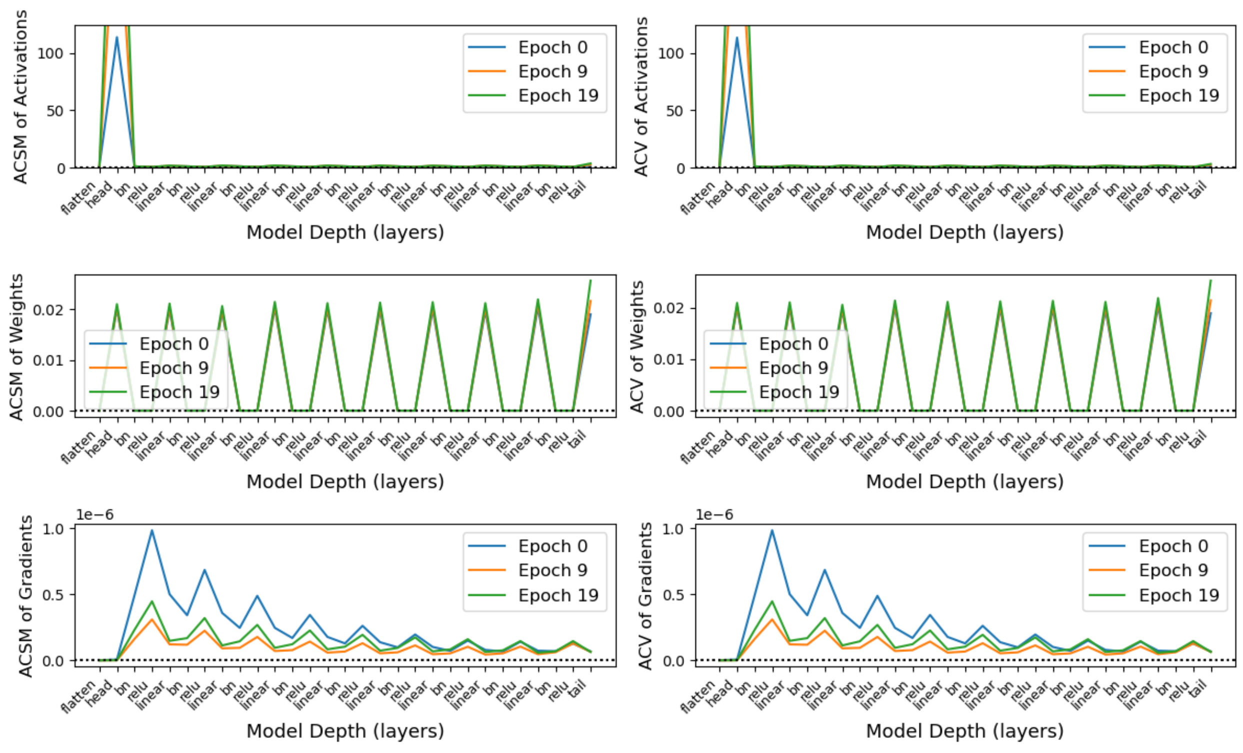


Figure 19 SPPs for training a 10-layered FFNN with BN with fixed parameters γ and β for 20 epochs at a learning rate of 0.001. Note the bigger initial spike and lower spikes for the hidden layers for the ACSM of Activations.

A.1.2. Convolutional Neural Networks

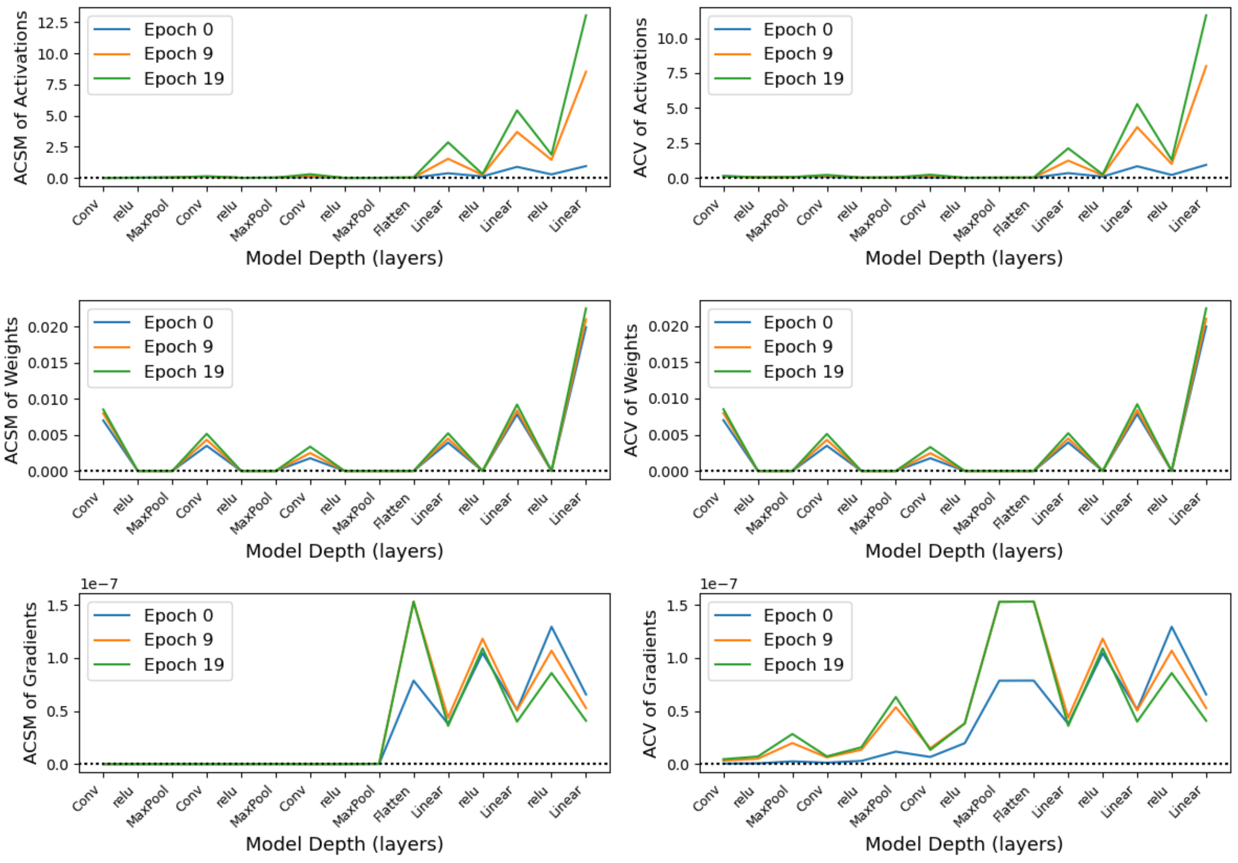


Figure 20 SPPs for training a CNN for 20 epochs at a learning rate of 0.001.

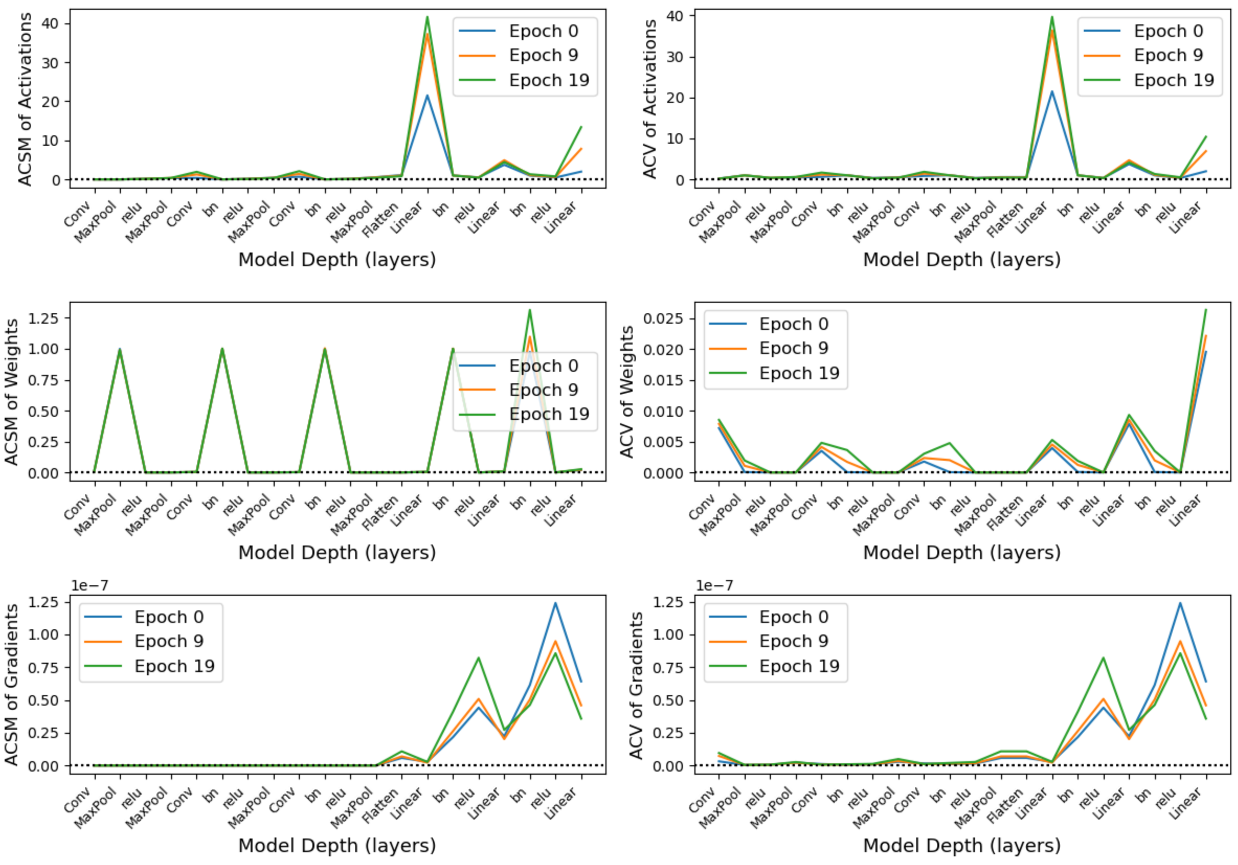


Figure 21 SPPs for training a CNN with BN in the feature extractor and classifier for 20 epochs at a learning rate of 0.001.

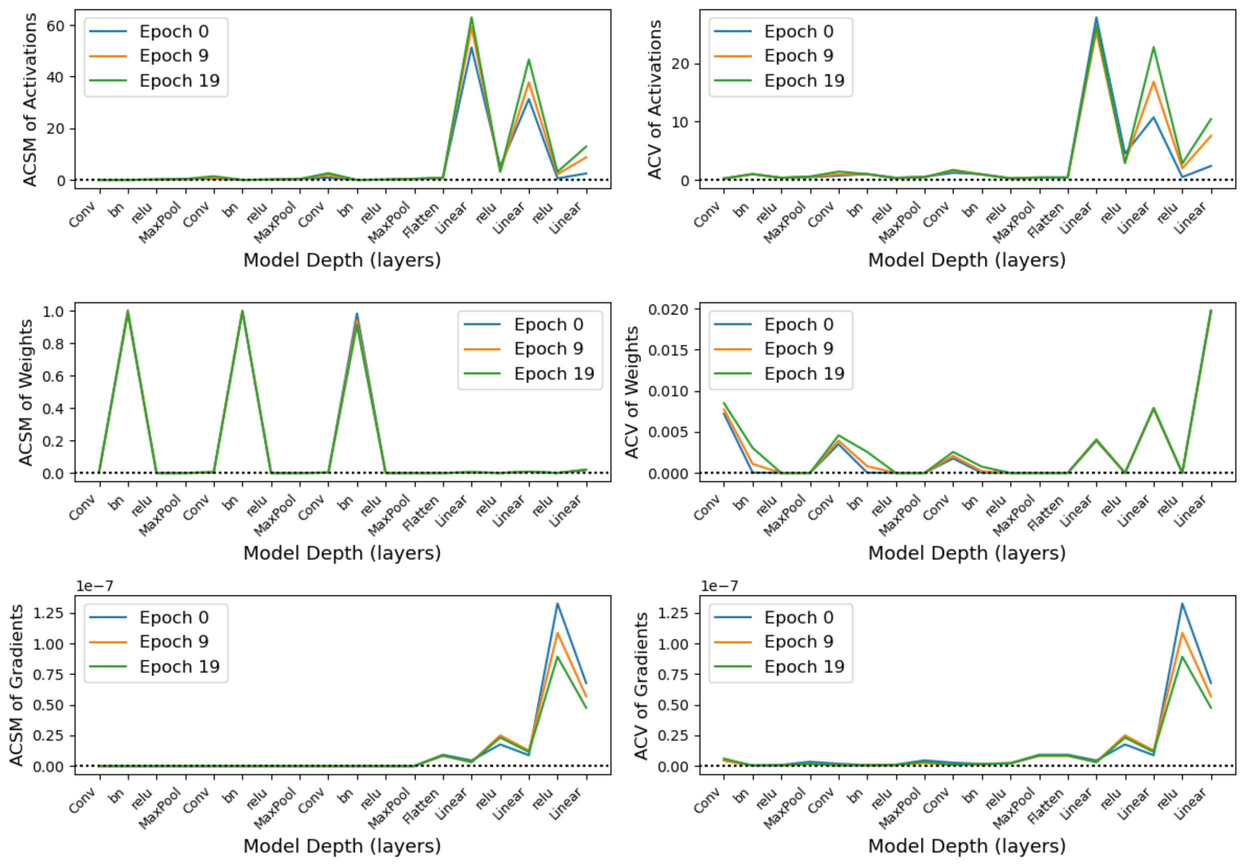


Figure 22 SPPs for training a CNN with BN in the feature extractor for 20 epochs at a learning rate of 0.001.

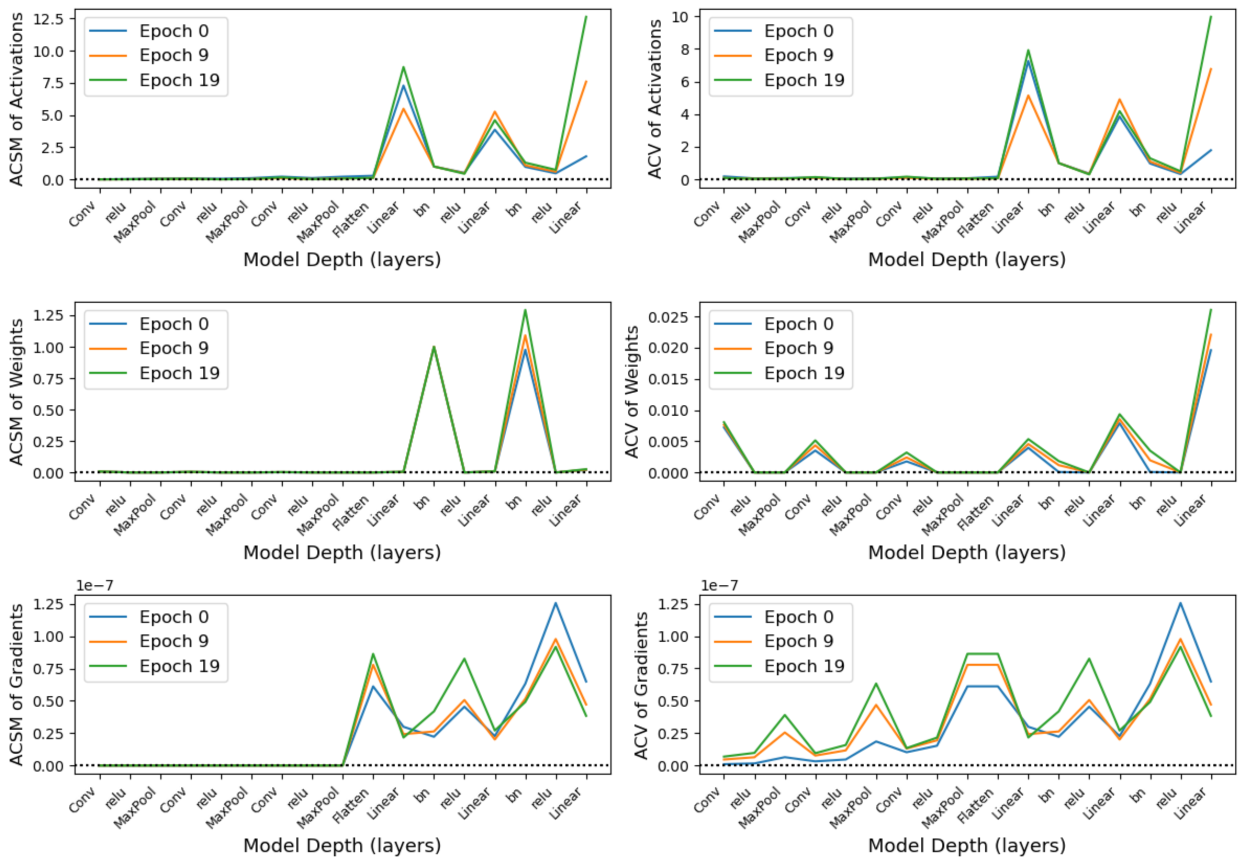


Figure 23 SPPs for training a CNN with BN in the classifier for 20 epochs at a learning rate of 0.001.

A.1.3.EfficientNet-B0

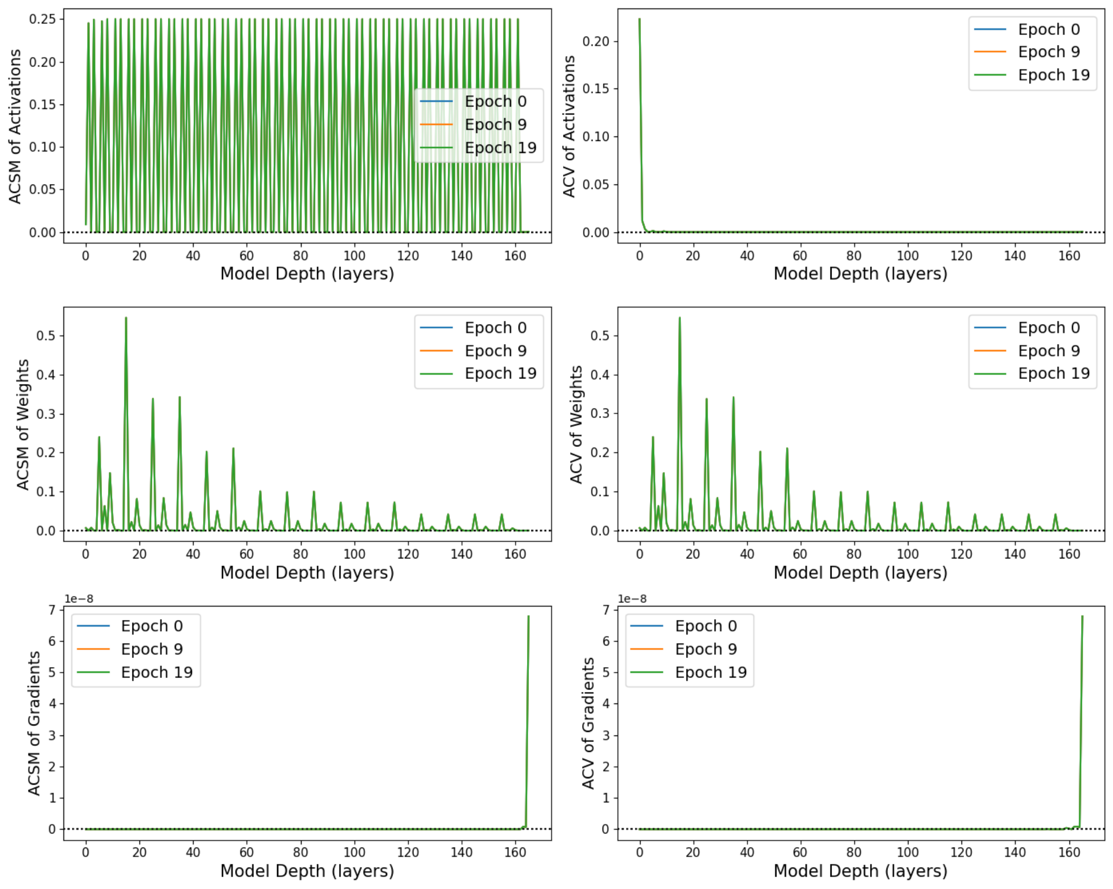


Figure 24 SPPs for training EfficientNet-B0 (vanilla) without BN for 20 epochs at a learning rate of 0.001.

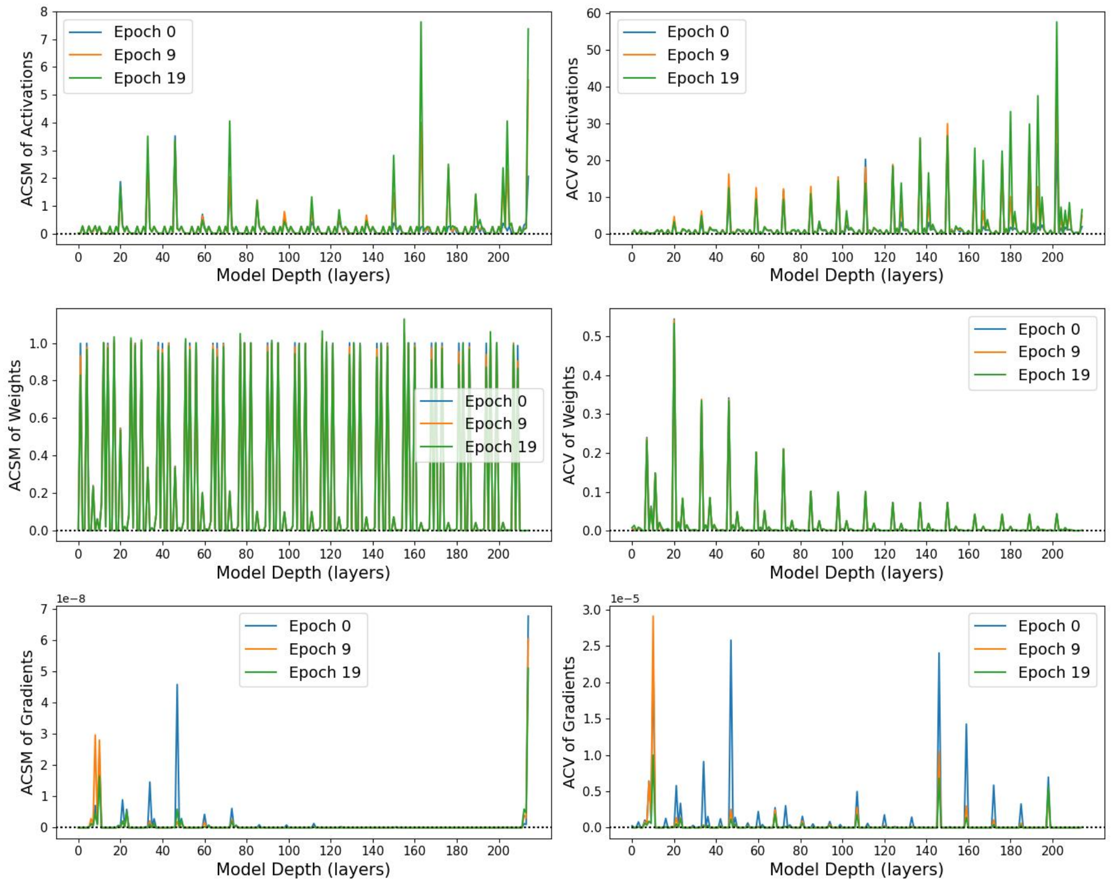


Figure 25 SPPs for training EfficientNet-B0 (vanilla) for 20 epochs at a learning rate of 0.001.

A.1.4. Self-Normalized EfficientNet

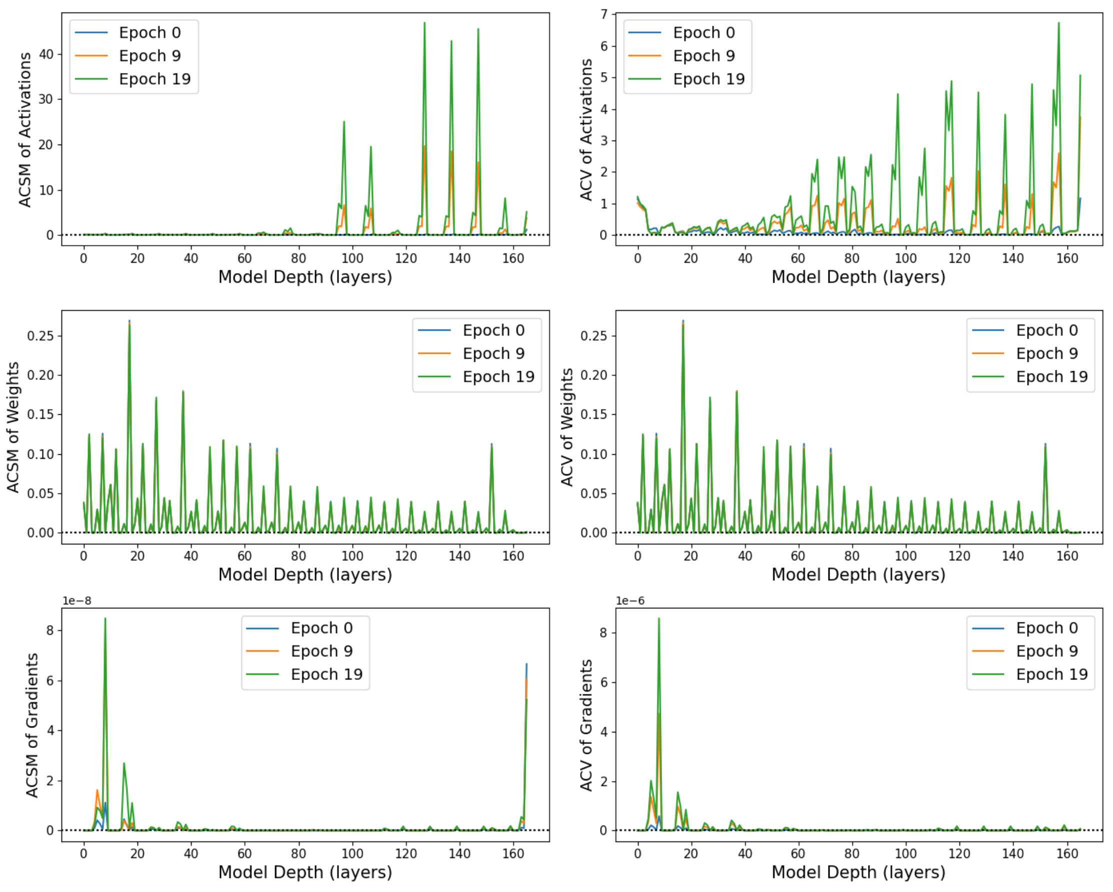


Figure 26 SPPs for training SN-EfficientNet for 20 epochs at a learning rate of 0.001.

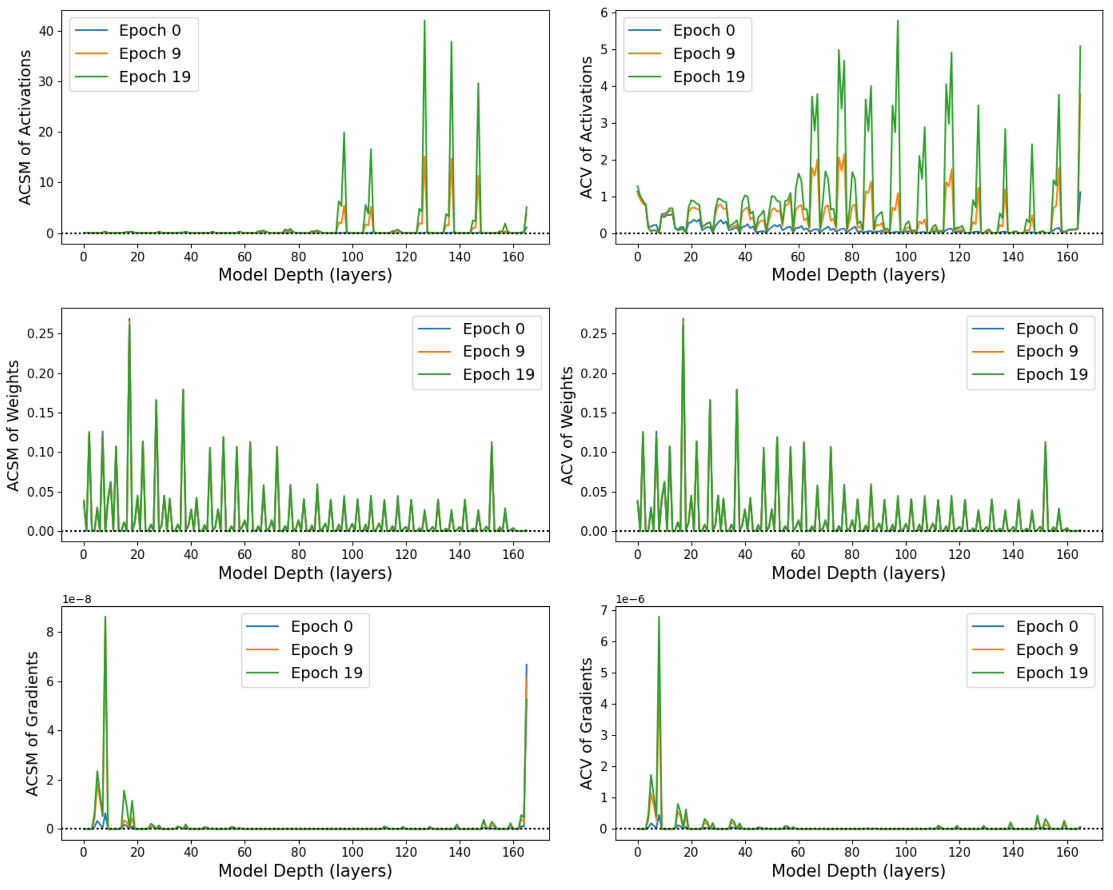


Figure 27 SPPs for training SN-EfficientNet and Magnitude-Preserving for 20 epochs at a learning rate of 0.001.

A.1.5. Un-Normalized EfficientNet

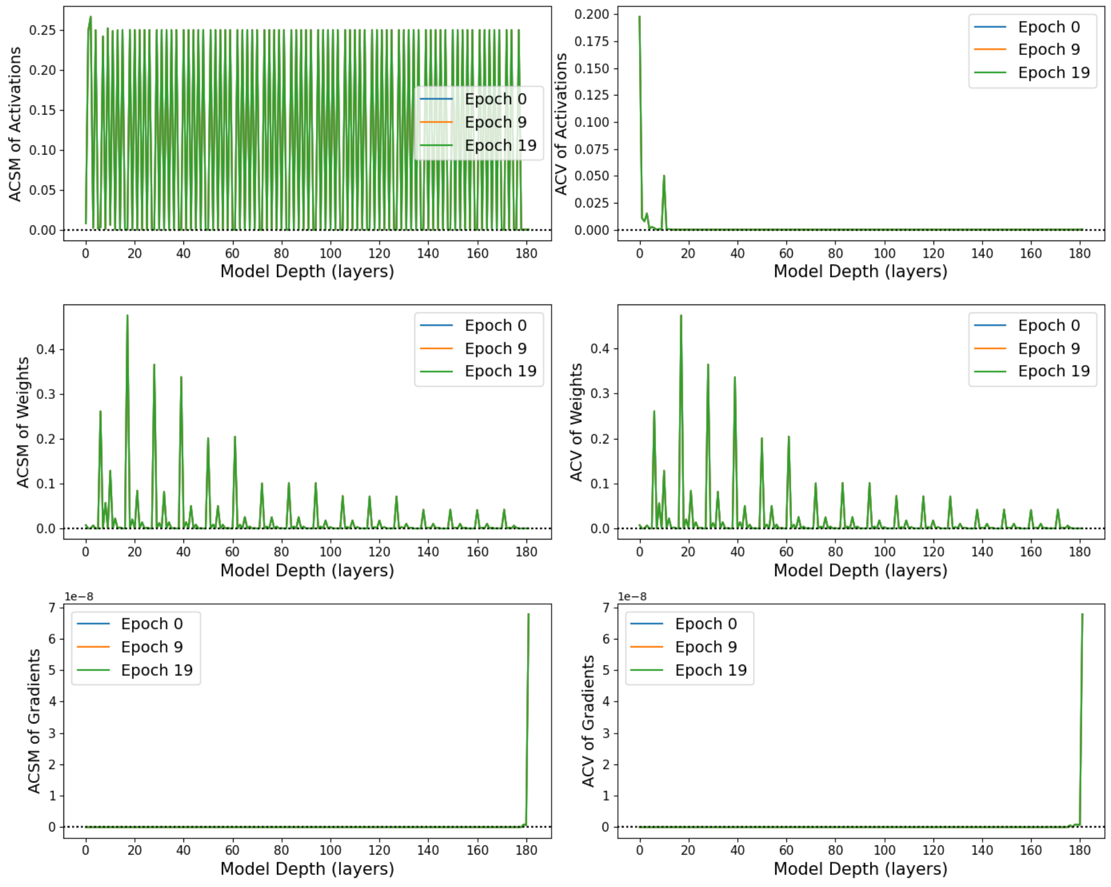


Figure 28 SPPs for training UN-EfficientNet for 20 epochs at a learning rate of 0.001.

B. Hardware and software specifications

- **System Information**

Operating System:	Windows 11 Pro 64-bit (Build 22631)
System Manufacturer:	Micro-Star International Co., Ltd.
System Model:	MS-7C91
BIOS-Version:	1.83 (UEFI)
Processor:	AMD Ryzen 5 5600X (6-Core, 12 Threads) @ 3.7 GHz
Installed Memory:	32 GB RAM
DirectX Version:	DirectX 12

- **Display Information**

Graphics Card:	NVIDIA GeForce RTX 3070 Ti
Display Memory:	24 GB (8 GB Dedicated, 16 GB Shared)

- **Software information**

Python:	3.12.4
PyTorch:	2.3.1 (CUDA 12.1, cuDNN 8)
NumPy:	1.26.4
SciPy:	1.13.1
scikit-learn:	1.4.2
Matplotlib:	3.8.4
Pandas:	2.2.2
TQDM:	4.66.4
W&B (Weights and Biases):	0.17.3