STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVI**, Number 1, 2021
DOI: 10.24193/subbi.2021.1.01

AN ANALYSIS ON VERY DEEP CONVOLUTIONAL NEURAL NETWORKS: PROBLEMS AND SOLUTIONS

TIDOR-VLAD PRICOPE

Abstract. Neural Networks have become a powerful tool in computer vision because of the recent breakthroughs in computation time and model architecture. Very deep models allow for better deciphering of the hidden patterns in the data; however, training them successfully is not a trivial problem, because of the notorious vanishing/exploding gradient problem. We illustrate this problem on VGG models, with 8 and 38 hidden layers, on the CIFAR100 image dataset, where we visualize how the gradients evolve during training. We explore known solutions to this problem like Batch Normalization (BatchNorm) or Residual Networks (ResNets), explaining the theory behind them. Our experiments show that the deeper model suffers from the vanishing gradient problem, but BatchNorm and ResNets do solve it. The employed solutions slighly improve the performance of shallower models as well, yet, the fixed deeper models outperform them.

1. Introduction

We have witnessed a lot of breakthroughs in deep learning lately [15] and all of them had a certain thing in common: very large and deep neural networks. The network depth has played probably the most important role in these successes, just over a span of a few years, the top-5 image classification accuracy over the ImageNet dataset has increased from **84%** [12] to **95%** [20], [16] using deeper networks with rather small receptive fields [2]. There seems to be a general rule that deeper is better and other results in this area have also underscored the superiority of deeper networks [25] in terms of accuracy and/or performance.

However, to achieve the advancements we have today, challenging problems had to be solved. There is a fundamental problem that very deep CNNs (Convolutional Neural Networks) suffer from. It was showed [10] that training

---

Received by the editors: 26 January 2021.
2010 *Mathematics Subject Classification.* 68T45 .
1998 *CR Categories and Descriptors.* I.2.1 [**Artificial Intelligence**]: Learning – *Connectionism and neural nets*.
Key words and phrases. Deep Learning, Neural Network, Image Classification, Deep Convolutional Neural Network, Vanishing Gradient Problem, VGG.

becomes more difficult as we increase the number of layers of a NN (Neural Network), stacking many non-linear transformations typically results in poor propagation of activations and gradients [19]. This is caused by the well-known problem of **vanishing/exploding gradients** [7]. With a big model, as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small (or infinitely large) and a meaningful signal won't reach the input layers causing the network not to learn anything even after the first iterations.

In this paper, we are going to visualize and explore this problem, analyze and test proposed solutions like **BatchNorm** [10], Resnets [6] and DenseNets [9]. We experiment on VGG (Visual Geometry Group) architectures [16] which are based on convolutional layers and are still an inspiration for top models these days. The **motivation behind this work** is the fact that current and previous state-of-the-art technology in computer Vision AI does heavily rely on a very deep convolutional architecture. Therefore, it is **important** to know **how to detect problems** and how to **successfully fix them** when using such tools. We will confirm one of the statements that were thought about the VGG networks - going deeper without any change whatsoever is unacceptable, visualizing the gradients during training. We propose some intuition and a mathematical underpinning of the problem that causes this phenomenon and explore solutions.

Our main contribution is a throughout evaluation of VGG networks of increasing depth using different stabilization techniques on the CIFAR100 image dataset [11]. We show that a plain (traditional) VGG network with 7 convolutional layers outperforms a much deeper network that uses 37 convolutions on a same setup. We prove that this is caused by the vanishing gradient problem (analyzing the gradients with respect to the model parameters) and we fix it using BatchNorm and Resnets showing that deeper is better if proper techniques are used to stabilize the learning of such models.

For the purpose of this research, we have used one of the most powerful GPU machines openely available to public as of today: the Nvidia Tesla V100, which allowed for 60% decrease in training time compared to other solid GPU workstations like Tesla T4 or K80.

2. Identifying problems of a deep CNN

As a baseline model we have a VGG network with 7 convolutional layers and 1 flatten layer. After training for 100 epochs, this model gets a train accuracy of around 54% and a test accuracy of 49%. The learning stage of this model is healthy enough, the accuracy does not decrease after a certain point and the generalization gap analyzing the loss is not that big. The gradient flow -

mean absolute value of gradients with respect to the model parameters can be seen in Figure 1. However, these are not satisfactory results, good models on this dataset achieve consistently over 70% accuracy [9].

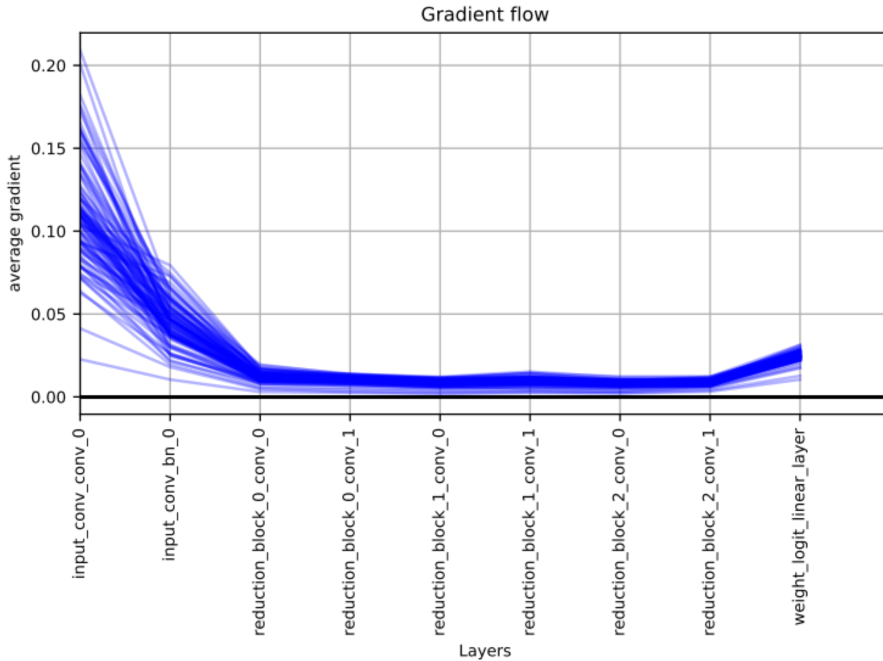

FIGURE 1. Gradient Flow in each layer for the healthy VGG 08 network.

Therefore, we tried repeating convolutional blocks over and over until we ended up with a VGG neural network with 37 convolutional layers and 1 flatten layer. Unsurprisingly, training this network, in its current form, did not yield great results, the test and train accuracy remained steady at 1% during the whole training stage and the loss did not decrease almost at all. It seems that the more shallow architecture beat the deeper one in this experiment. In 1989, Cybenko proved [3] that a network with a large enough single hidden layer of sigmoid units can approximate any decision boundary. Empirical work, however, suggests that it can be difficult to train shallow nets to be as accurate as deep nets. Moreover, for vision tasks, multiple studies suggest that deeper models are **preferred** under a **parameter budget** [4], [19], [16].

So why is it not the case that we get better performance with higher number of hidden units?

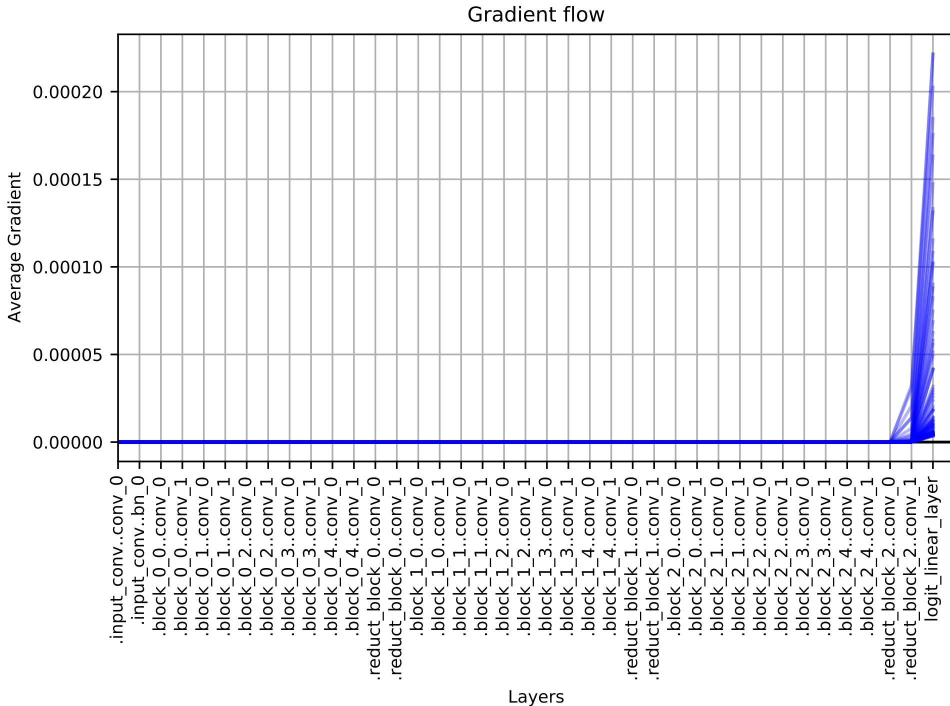

FIGURE 2. Gradients vanishing when training a VGG model with 37 convolutional layers (VGG 38 network). Simply stacking layers does not work.

Well, increasing network depth does not work by simply stacking layers together. Very deep networks are hard to train because of the vanishing gradient problem (Figure 2). An **intuition** for that happening is that, when the network is too deep, the gradients from where the loss function is calculated easily shrink to zero after several applications of the **chain rule**, so gradients aren't really back-propagated sufficiently to the initial layers of the network. This can be clearly seen in the **Figure 2** that shows the mean absolute value of the gradients at each epoch. The gradients quickly turn very close to 0 after just 2 layers during backpropagation from output layer to input layer.

This is just an intuition, but neural networks haven't been regarded as uninterpretable black-boxes for no reason, can we somehow explain this phenomenon mathematically?

In a way, yes. In very deep architectures, the variance of the data changes at each activation and the idea that earlier layers influence later layers in complex

ways is not new. The problem is to understand why and how these high order interactions between layers are an issue for learning.

Suppose that we are minimizing a loss function $f(w)$ using gradient descent, where w are the weights. We consider what happens when we take a step in the direction of the gradients from the current weights w_0 . Of course, we don't know the form of function f yet, however, recall the **Weierstrass Approximation Theorem** in R^n which states that every real-valued continuous function in a closed n dimensional subspace can be uniformly approximated as closely as desired by a **polynomial function**. Note that this does not contradict our context here, as it is generally assumed NNs do provide differentiable, well-behaved functions (as gradients are backpropagated through the layers), so it is a reasonable assumption to consider f continuous (as a consequence). To approximate f, usually a second order **Taylor** polynomial expansion is taken (around the current weights w_0): $f(w) \approx f(w_0) + (w - w_0)^T g + \frac{1}{2}(w - w_0)^T H(w - w_0)$ where g and H are the gradient and Hessian matrix of $f(w)$ at w_0 . When we take a step in the direction of the gradient with size ϵ , the loss function becomes :

$$f(w_0 - \epsilon g) \approx f(w_0) - \epsilon g^T g + \frac{1}{2}\epsilon^2 g^T H g \quad (*)$$

This is actually a well known formula in convex optimization as it was used in old papers that were not even Deep Learning related [21]. Notice the third term on the right-hand side of the equation: $\frac{1}{2}\epsilon^2 g^T H g$. If this term was **0**, the loss function would strictly decrease. This happens when the model has no second-order terms - i.e. when it is a strictly linear model. On the other hand, if this term was sufficiently large, it may exceed the absolute value of $\epsilon g^T g$ so the loss might actually increase. This happens when the **second-order** effects outweigh the **first-order** effects. It is regarded that the last term (the one that contains the Hessian and the gradient) represents the effect of the curvature of the loss function [14]. If the **curvature is small**, the **gradient is mostly constant**, meaning we can take a large step-size ϵ and decrease the loss. On the other hand, when the curvature is large, the gradient changes quickly, meaning a large step-size poses a risk of increasing the loss. In the worst case, the gradient is the eigenvector of H with the largest eigenvalue.

The mathematical background presented above was needed as solutions to the vanishing gradient problem do refer to this problem of **conditioning**, to be more precise, the *ill-conditioning* of the Hessian matrix. The only way to ensure that the curvature does not cause the loss to increase is by decreasing the step-size ϵ -making it extremely small. Using a very small learning rate (lr) with **VGG 38** is just not practical, though. Of course, we analyzed what happens only for second-order effects, but this gives tremendous insight into the behavior of deeper neural networks. We are confident that this translates

to higher order effects caused by very deep NN architectures, that need higher order **Taylor** series for a good approximation, where there are third, fourth, and even higher-degree effects between the weights. This means that gradient updates can be even more unpredictable because the higher order interactions complicate the gradient update, and the only way to ensure that these effects do not adversely affect the loss is to make the step-size extremely small, **or to incorporate techniques that allow higher learning rates to be used**.

3. Background Literature

3.1. Batch Normalization.

Batch normalization (BN) [10] is a technique to normalize activations in intermediate layers of deep neural networks. BN has become a staple in state-of-the-art models because of its tendency to speed up training and improve performance. The main idea is to normalize the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. It is empirically proved that this solves the **vanishing gradient** problem in very deep CNNs possibly due to more controlled activations and well-behaved gradient updates.

To our understanding, the **motivation** comes from the fact that we always knew input normalization is needed for a healthy learning; if the input layer is benefiting from it, why not do the same for the values in the hidden layers, as the distribution of each layer's inputs changes all the time during training? We normalize the input layer by **adjusting** and **scaling** the activations. This way, it reduces the amount by what the hidden unit values **shift around**. The authors refer to this phenomenon as **internal covariate shift**.

However, after this shift/scale of activation outputs by some randomly initialized parameters, the weights in the next layer are no longer optimal. To address this problem, the authors introduced, for each activation, **a pair of trainable parameters** γ, β, which scale and shift the normalized value: $y = \gamma x + \beta$. BN lets the gradient descent do the denormalization by changing only these **two weights** (γ, β) for each activation, instead of losing the stability of the network by changing **all the weights**.

It comes as a consequence that BN allows each layer of a network to learn by itself a little bit more independently of other layers, but, intuitively, **why does that help?** Recall formula $(*)$. With BN, the mean and variance of the activations of each layer are independent by the values themselves, **they are not decided by complex interactions between multiple layers, but rather by two simple parameters**. This means that the **magnitude**

of the higher order interactions are likely to be **suppressed**, allowing **larger learning rates** to be used.

Among other **advantages** of BN, it helps bypass local minima and makes the training more resilient to weights initialization and the authors show that it is invariant to parameter scale. It is a form of **regularization**, the networks with BN usually do not require Dropout [17]. BN also enabled the training of deep neural networks with sigmoid activations that were previously deemed too difficult to train due to the vanishing gradient problem.

Nevertheless, as any method in machine learning, there are some **limitations**. Convergence is necessary for generalizing well, but if a network converges without normalization, BN does not add further improvement in generalization [1]. It was also showed that BN strongly depends on how the batches are constructed during training, and it may not converge to a desired solution if the statistics on the batch are not close to the statistics over the whole dataset. Moreover, it was shown [13] that BN fails/overfits when the mini-batch size is 1 and are in general, very sensitive to the mini-batch size.

3.2. Residual Neural Networks.

Residual Neural Networks (Resnets) [6] are a family of neural networks with a specific common trait: they use skip connections in their architecture to fit the input from the previous layer to the next layer without any modification of the input. Resnets solve the vanishing gradient problem by letting the gradients flow directly through the skip connections backwards from later layers to initial filters. Other problems that these NNs solve is the **shattered gradients** problem in which we get gradients that are not correlated within samples in any way.

The **motivation** behind the authors' work is the fact that adding multiple layers to an already defined NN architecture shouldn't come at any performance cost if the layers that we add are identity mappings - that don't do anything. It should be easy for a NN (which is a good function approximator) to learn the identity map $f(x) = x$. The authors also took inspiration from other sources as Resnet was not the first one to use skip connections. **LSTMs** [8] have a similar mechanism with their parametrized forget gate that controls how much information will flow to the next time step and there is also **Highway Networks** [18] which actually contain Resnets in their solution space and yet they perform no better than them.

It is said that the problem of training very deep CNN models has largely been overcome via carefully constructed initializations and BN, however, architectures incorporating skip-connections such as highway and resnets perform much better than standard feedforward architectures despite BN. **But why**

wasn't BN enough to train very deep models, what is it that these deep residual models do better? In short, when training deep networks there comes a point where an increase in depth causes accuracy to saturate, then degrade rapidly - the **degradation problem** caused by **shattered gradients**. Shattered gradients resemble white noise and cancel each other out, making training more difficult. Shallow networks have unshattered gradients. However, for deeper networks, training them with batch norm leads to shattered gradients, while training them without it leads to the vanishing gradient problem. ResNets help ameliorate both problems, one of the arguments is that they resemble an ensemble of shallow networks.

The authors tested a 152-layered NN for ImageNet classification. It is really impressive that this was **8x** bigger than **VGG** nets, but it does require less computation according to the no of Flops.

Limitations of the Resnet concern a mathematical underpinning of the empirical research. Moreover, a study [22] found out that Resnet and variants of Resnet extremely vulnerable to adversarial examples (or attacks) [5] , which are input examples slightly perturbed with an intention to fool the network to make a wrong classification.

3.3. **Densely connected neural networks.**

Densely connected neural networks (Densenets) [9] extend on the idea of shortcut connections present in Resnets, connecting all the layers directly with each other. In this novel architecture, the input of each layer consists of the feature maps of all earlier layer, and its output is passed to each subsequent layer. The feature maps are aggregated with **depth-concatenation** and not with **summation** using identity mappings like Resnets. These connections form a dense circuit of pathways that allow better gradient-flow, thereby solving the vanishing gradient problem.

A key insight in this architecture is that each layer has direct access to the gradients of the loss function and the original input signal, the model requires fewer layers, as there is no need to learn redundant feature maps, allowing the *collective knowledge* to be **reused** - feature reuse, making the network highly **parameter-efficient**. Fewer and narrower layers means that the model has fewer parameters to learn, making them easier to train. The authors also talk about the importance of variation in input of layers as a result of concatenated feature maps, which prevents the model from over-fitting the training data which makes sense.

The full architecture proposed in the paper makes use of dense blocks and transition blocks. The dense blocks, as we mentioned before, are composed of interconnected dense layers (that here are 1x1 conv + 3x3 conv). A term that

is frequently brought up in the paper is the growth rate k that dictates how many channel features are concatenated and fed as input to the next dense layer. Transition blocks are used between dense blocks and use Convs and average pooling for dimensionality reduction.

Densenet models without hyper-parameter tuning are **compared** to **Resnet** with optimal hyper-parameters over the ImageNet dataset and it turns out that the Densenet model has a **significantly lower validation error** than the ResNet model with the **same number of parameters**. Moreover, another experiment showed that a Densenet model with **20M** parameters model yields similar validation error as a 101-layer ResNet with more than **40M** parameters.

Therefore, it seems that Densenet is a **clear improvement** over the previous state-of-the-art Resnet, granted the two architectures have the main concept of **skipping layers in common**, the execution being different. In relationship with **Batch Normalization**, both of them do use it which shows how important this technique still is.

4. Solution Overview

In order to solve the vanishing gradient problem, we have chosen **Batch Normalization** as a first-hand solution. The motivation behind this is the fact that all the solutions from the literature review section had this technique in common, so it comes as natural to apply it.

Implementing BatchNorm is like applying pre-processing but for hidden layers. The idea is to normalize the output coming from a previous hidden layer (likely Conv), restricting the amount by what a hidden unit value can shift around. An idea that we have brought up in the literature review as well is the reduction of the internal covariate shift. **Covariance shift** is directly linked to the different distributions that can appear in the data: if it changes between training data (for example, we train the model on greyscale images) and test data (we test it on RGB images), our algorithm would be, of course, pretty poor; BN tries to solve that.

The algorithm can be seen below, note the two model parameters introduced by BN (γ, β) that help the optimizer undo the normalization if it's a way for it to minimize the loss function. We add these two trainable parameters to each layer, so the normalized output (that has 0 mean and 1 standard deviation) is multiplied by a *standard deviation* parameter γ and add a *mean* parameter β. In practice, restricting the activations of each layer to be strictly 0 mean and unit variance can limit the expressive power of the network. Therefore, in practice, batch normalization allows the network to learn parameters γ and β that can convert the mean and variance to any value that the network desires.

Algorithm 1 Batch Normalization

Input: Values of X over a mini-batch after a Conv Layer: $x_i, i \in 1, 2, ..., n$.
Parameters to be learnt: γ, β.
$\mu \leftarrow \frac{1}{n} \sum_i^n x_i$ //mini-batch mean
$\sigma^2 \leftarrow \frac{1}{n} \sum_i^n (x_i - \mu)^2$ //mini-batch variance
$\hat{x_i} \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \forall i \in 1, 2, ..., n$ // normalization
$y_i \leftarrow \gamma \hat{x_i} + \beta, \forall i \in 1, 2, ..., n$ // scale and shift
Output: y

It comes as a natural question, though, if we should apply BN before and after activations. We have researched this issue quite thoroughly and there is not a clear definite answer to it. Although the proposed approach in the original paper used BN before activations, many empirical experiments have been conducted by the community [23] and great results are showed applying BN after ReLu activations. However, **our understanding is that BN helps more by reducing the high-order realationships between parameters of different layers than reducing the covariate shift, therefore the order might not really matter**. We will apply BN before activation and possible pooling layer as described by the authors of the original paper.

It is said that BN also **regularizes** the model. The **intuition** for this is that BN adds extra sources of noise so that every layer has to learn to be robust to account for the variations in its inputs: because the data points are randomly chosen to form a minibatch, the standard deviation randomly fluctuates and BN multiplies each hidden unit by such randomly fluctuating standard deviations and also subtracts the randomly fluctuating means of the minibatch data points.

BatchNorm will likely solve the problem, however, to get even more improvement in the performance of the model, we also chose to implement **Residual Blocks**.

Implementing **Resnet** is straight-forward: to construct a **skip connection** over a layer that applies transformation F to an input vector x, we modify the output of the whole block to another map $H(x) = F(x) + x$.

The idea is that **even if** there is **vanishing gradient** for the weight layers, we always still have the identity x to transfer back to earlier layers. The weight layers have to learn this kind of residual mapping: $F(x) = H(x) - x$. Intuitively, if we bypass the input to the first layer of the model to be the output of the last layer of the model, the network should be able to predict whatever function it was learning before with the input added to it.

However, this does **not** work if $F(x)$ changes the dimensions of x, so we need to be careful when implementing it. We have to find a **linear projection** $g = Wx$ such that we preserve the features in x but we reduce its dimensionality. This is addressed in the original paper: if we have to increase the dimensions of x to match $F(x)$, then **padding** is recommended as it does not add any more model parameters and is quite efficient. If we need to reduce it, we can apply any **pooling** transformation or, a **1x1 convolution** with an appropriate stride - this is what the authors use in their experiments. We can view the pooling reduction as a direct scaling without adding extra parameters, however, the **1x1 convolution** approach would work better in theory because, intuitively, this is like a *learnt* scaling.

5. Experiments

We base our experiments using the **CIFAR100** dataset which contains **60k**, 32x32 colored natural images. For all our experiments, we train on **100** epochs, having **47.5k** of the images as training data, **2.5k** as validation data and the rest (**10k**) as test data. Note that at each experiment we shuffle the samples and apply some basic data augmentation: random crop, horizontal flip and gaussian noise on all 3 RGB channels having the *mean* $(0.4914, 0.4822, 0.4465)$ and the *std* $(0.2023, 0.1994, 0.2010)$.

The architectures we are going to use are quite similar at a base level. We have **convolutional processing blocks** that are repeated in the network. Such a block is a cascade of **2** convolutional layers, each followed by a **Leaky ReLu** activation function. We also have 1 to 3 **reduction blocks** that are used to downgrade the units in terms of width and height through pooling. Each such block is a cascade of 2 convolutional layers with an average pooling layer in the middle. All these consecutive blocks and with a flatten and a softmax layer. Denote **VGG 08** being such a NN with 7 convolutional layers + 1 flatten layer and **VGG 38** a NN with 37 convolutional layers + 1 flatten layer.

The motivation behind using leaky ReLu is that it's more unlikely to suffer from vanishing gradients than other non-linear activation functions (sigmoid/tanh). Plus, we use the leaky version because we want to better account for the negative values that come through the layers. Average pooling was used in image classification by previous state-of-the-art models like **Densenets**. By default, we use **Adam** with **lr=0.001** and a batch-size of **100**.

The first experiment is to test the effectiveness of **BatchNorm** to solve the **vanishing gradient** problem. To do that, we have applied BN directly after every convolutional layer and let the **VGG 38** train with the same hyper-parameters as before. The mean absolute values of the gradients at each epoch

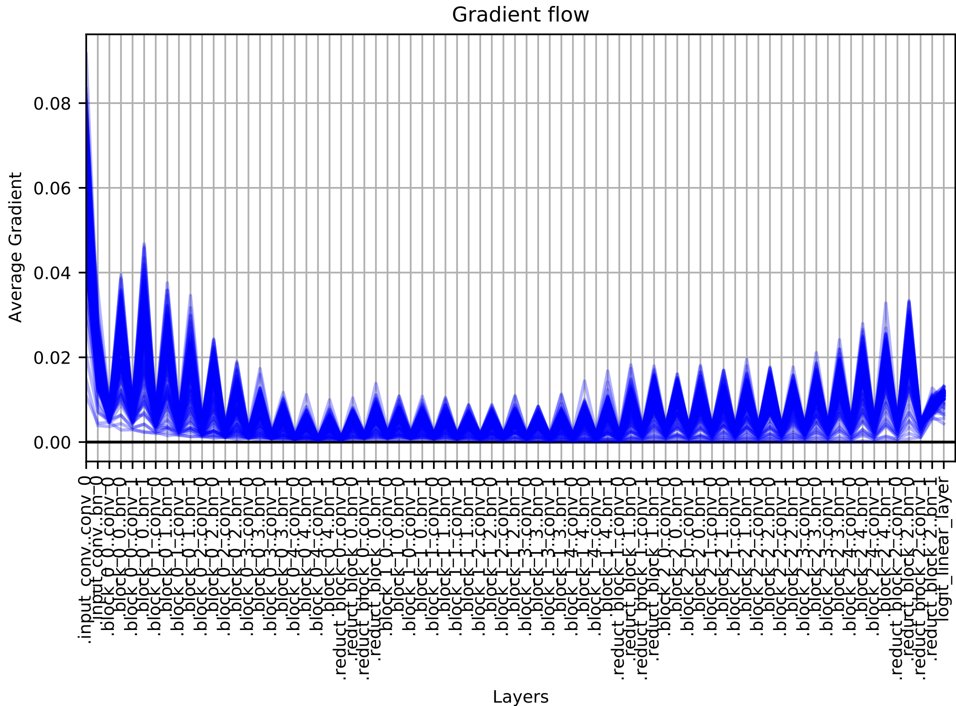

FIGURE 3. Mean abosolute values of gradients w.r.t model parameters at each epoch for VGG 38 with BN.

are displayed in **Figure 3**. We have also applied BN to the **VGG 08** model to see if it improves the results even for more shallow networks and it did (about 4%). All these results with the accuracy can be seen in **Table 1**. Note that for all the experiments, we have conducted tests on different seeds (for the initial random weights initialization) to check the robustness of the results and solidy our claims. We used 5 different seeds (0, 100, 550, 1000, 40000) and we recorded a standard deviation of the accuracy results of 0.499.

It is clear (Figure 3) that BN **solves** the main problem we are dealing with in this paper. However, the performance at the moment of **VGG 38** does not justify its complexity as **VGG 08** still has **similar performance with less number of model parameters**. That's why we now investigate if we can improve the performance with an intuitive change in hyper-parameters.

As we noted multiple times in this paper, BN allows training with higher learning rates, so this is the first thing we are going to try out. We try

0.01 and 0.1, however, the results are not that successful (Table 1), **0.001** still seems to be the appropriate learning rate to be used. We have chosen these learning rates because successful models from the literature that use BN typically choose higher learning rates [6] [9], however, they do use a **decaying factor**, so that might explain why it doesn't work that well in our case.

We have also tried different batch sizes, as we have stated in the literature review that this is a factor that can influence BN in a big way. We choose to experiment with **256** as it was used in other studies [6]. We have also tried **512** because it is more time efficient to use large batch sizes and that also approximates the **gradient** of the whole dataset a little bit better. The computational time decreases by at least 41.66% when we change the batch size from **100** to **256** or **512**, which is impressive, that's why we move on from using **100** as batch size for the next experiments.

The original paper that introduces BatchNorm claims that it solves the training issues even with **sigmoid**-like activation functions (which are notorious for the vanishing gradient problem) and this seems to be universally accepted by the community. We tested that out to confirm it (Table 1, Figure 4).

Results so far with BN still can't yet justify the need of a deeper model (the test accuracies are similar between VGG 38 and VGG 08), so we conduct experiments with residual blocks added into the models. Firstly, we wanted to try out a default version with skip connections but without BN just to prove that residual blocks alone can solve the vanishing gradient problem and indeed, our experiment was successful (Table 1 - VGG 38 Resnet).

Model	*Batchsize*	*Lr*	*Weightdecay*	*Testacc*	*Trainacc*
VGG08 Baseline	*100*	*0.001*	*0*	*49.95%*	*55.24%*
VGG08 BN	100	0.001	0	53.89%	59.65%
VGG08 BN + Resnet	100	0.001	0	54.45%	60.74%
VGG38 baseline	100	0.001	0	1%	1%
VGG38 BN	100	0.001	0	46.78%	54.20%
VGG38 BN	100	0.01	0	44.14%	48.33%
VGG38 BN	100	0.1	0	22.46%	24.86%
VGG38 BN	256	0.001	0	47.22%	58.05%
VGG38 BN	512	0.001	0	46.49%	60.43%
VGG38 Sigmoid BN	512	0.001	0	26.88%	28.29%
VGG38 Resnet (only)	100	0.001	0	44.77%	52.20%
VGG38 BN + Resnet	512	0.001	0	58.84%	78.65%
VGG38 BN + Resnet	512	0.1	0.0001	30.22%	29.70%
VGG38 BN + Resnet	256	0.01	0.0006	57.67%	72.81%
VGG38 BN + Resnet	256	0.01	0.0001	58.25%	67.50%
VGG38 BN + Resnet	**256**	**0.001**	**0.0001**	**61.81** %	**85.21%**

TABLE 1. VGG08, VGG38 models - varying hyper-parameters. Statistical error: +/-0.49.

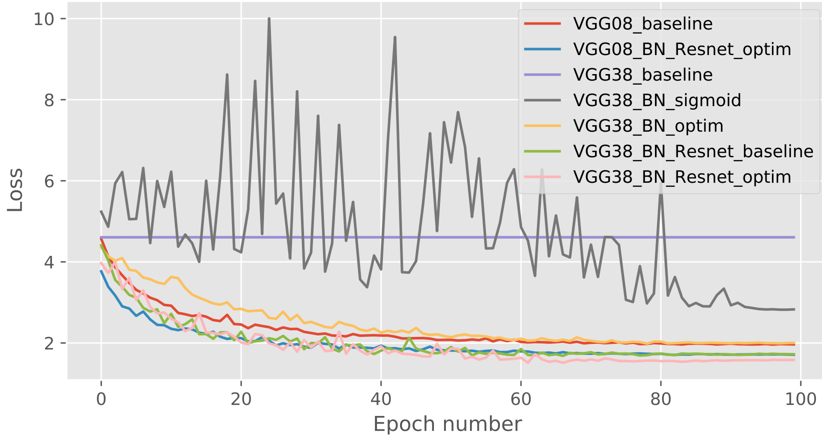

FIGURE 4. Validation loss during training for different models tested.

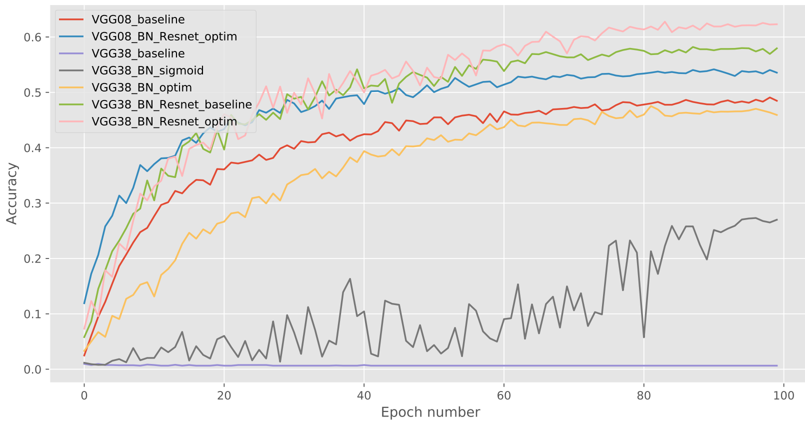

FIGURE 5. Validation accuracy during training for different models tested.

However, for the next experiments, we used **Resnet + BN** as it was recommended in the original paper [6]. We first tested it on VGG 08 to see if it adds any improvement to the previous version and indeed, the test accuracy is slightly higher. However, when training VGG 38 with BN and Residual Blocks, we got much higher results. Nevertheless, it was clear (Table 1) that

these models were overfitting on the training data (huge generalization gap), so we tried adding L2 regularization - varying weight penalty, which also used in [6]. This has helped, the training evolution summaries can be observed in Figure 4 and 5.

6. Discussion

Looking at **Figure 3**, the gradients seem healthy as they do not come really close to **0** and the scale is big enough, BN solved the vanishing gradient problem. Note that we expect small gradients for this problem, however, the scale from **Figure 2** was in the $O(10^{-3})$ region and now it is in $O(10^{-1})$. This should mean that the network is indeed learning and the gradient flow provides meaningful signal backwards to the input layers and indeed, it achieves an accuracy of **46.78%**, similar to what **baseline VGG 08** got. Note that there is a **zig-zagg** phenomenon that can be observed, this also happens when training NNs with sigmoid activations (not zero-centered) - the data coming into a neuron is always positive, then the gradient on the weights (during backprop) will become either all be positive or all negative - this leads to zig-zagg dynamics in the gradient updates for the weights. We suspect a similar cause is in our case.

It seems **256** as batch size gives the best test accuracy, which is not surprising as other studies like [6] for image classification also use that.

Sigmoid as activation function for the hidden units lead to a **noisy** behaviour in training (fig. 4&5), however, the model does somewhat learn and achieves an accuracy of **26.88%** on test, which is not great but **proves the hypothesis that BN makes even deep networks with sigmoid work**.

Using **Residual Blocks** and **BN** for **VGG 08** lead to an about **1%** increase in acc, which is not really justifiable. However, when training **VGG 38** with the same architecture, we can begin to see why **deeper is better**, the best model achieving over 60% accuracy on test and 85% on train (**Table 1**), granted it overfits much harder than **VGG 08**. The architecture of this best model can be seen in **Figure 6**, note that we display the **convolutional processing** and **reduction blocks** that we mentioned in the previous section, there are such **5** processing blocks followed by **1** reduction block repeated **3** times in a **VGG 38** model.

Note that we have chosen to implement the linear projection to make the identity smaller as a **1x1 conv** with appropriate stride. We have done this because using max-pooling instead lead to less impressive results, moreover, this was the recommended way in [6] and we personally believe this adaptive *learnt scaling* is better than an absolute one.

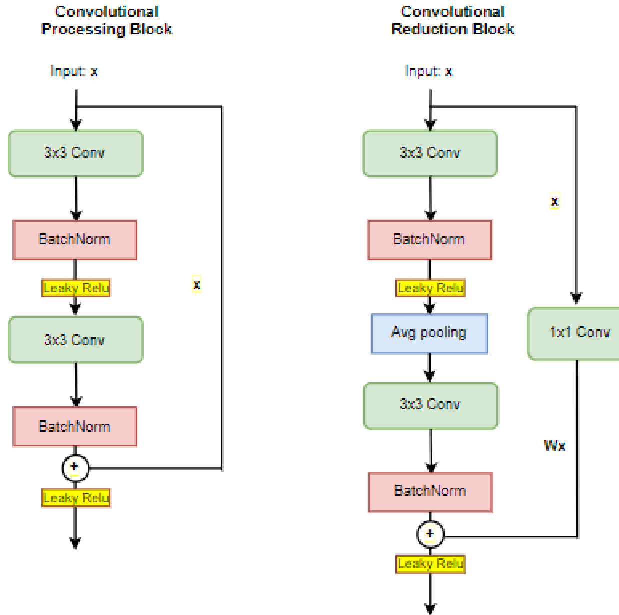

Figure 6. Building blocks for our best model VGG38 BN+Resnet

7. Conclusions and Further Work

Very deep CNNs are responsible for the recent *quantum leaps* in AI, not only in computer vision but also in Reinforcement Learning, for example, AlphaGo [15]. Training such models is a serious problem and different techniques need to be applied to get very good results. Simply stacking convolutional layers does not work for VGG models as we have seen in Figure 2. However, this can be solved by either BatchNorm or Resnets. Combining these two together is much better than using them separately. These methods can also improve more shallow networks (VGG 08); however, the very deep models completely outperform shallower methods in this case (over 7% increase), which is expected. However, there is still room for improvement regarding the hyper-parameters, furthermore, we use only 3x3 convolutions which can be very restrictive. Models similar to ours, like *Wide ResNet*, *ResNeXt*, [24] achieve much higher accuracy: 79.5%, 82.3%, but they also have much more **trainable parameters**, further experimentation on that can be beneficial.

References

[1] Bjorck, N., Gomes, C. P., Selman, B., and Weinberger, K. Q. Understanding batch normalization. In *Advances in Neural Information Processing Systems* (2018), pp. 7694–7705.

[2] Ciregan, D., Meier, U., and Schmidhuber, J. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition* (2012), IEEE, pp. 3642–3649.

[3] Cybenko, G. Mathematics of control. *Signals and Systems 2* (1989), 303.

[4] Eigen, D., Rolfe, J., Fergus, R., and LeCun, Y. Understanding deep architectures using a recursive convolutional network. *arXiv preprint arXiv:1312.1847* (2013).

[5] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

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

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

[8] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[9] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 4700–4708.

[10] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015* (2015), F. R. Bach and D. M. Blei, Eds., vol. 37 of *JMLR Workshop and Conference Proceedings*, JMLR.org, pp. 448–456.

[11] Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. *University of Toronto* (2009).

[12] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM 60*, 6 (2012), 84–90.

[13] Lian, X., and Liu, J. Revisit batch normalization: New understanding and refinement via composition optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics* (2019), pp. 3254–3263.

[14] Martens, J. Deep learning via hessian-free optimization. In *ICML* (2010), vol. 27, pp. 735–742.

[15] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *nature 550*, 7676 (2017), 354–359.

[16] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[17] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[18] Srivastava, R. K., Greff, K., and Schmidhuber, J. Highway networks. *CoRR abs/1505.00387* (2015).
[19] Srivastava, R. K., Greff, K., and Schmidhuber, J. Training very deep networks. In *Advances in neural information processing systems* (2015), pp. 2377–2385.
[20] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.
[21] Thacker, W. C. The role of the hessian matrix in fitting models to measurements. *Journal of Geophysical Research: Oceans 94*, C5 (1989), 6177–6196.
[22] Wu, D., Wang, Y., Xia, S.-T., Bailey, J., and Ma, X. Skip connections matter: On the transferability of adversarial examples generated with resnets. unpublished, 2020.
[23] XalosXandrez. Batch normalization before or after relu? https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/. Published: 2017-04-25.
[24] Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 1492–1500.
[25] Yu, D., Seltzer, M. L., Li, J., Huang, J.-T., and Seide, F. Feature learning in deep neural networks-studies on speech recognition tasks. *arXiv preprint arXiv:1301.3605* (2013).

The University of Edinburgh, School of Informatics, 10 Crichton St, Newington, Edinburgh EH8 9AB, United Kingdom
Email address: T.V.Pricope@sms.ed.ac.uk