Master's Thesis

Generalization of SELU to CNN

Bach Ha

Submitted to Hochschule Bonn-Rhein-Sieg,
Department of Computer Science
in partial fullfilment of the requirements for the degree
of Master of Science in Autonomous Systems

Supervised by

Prof. Dr. Paul G. Plöger
Prof. Dr. Gerhard K. Kraetzschmar
Dr. Florian Zimmermann

Jan 2019

Abstract

Neural network based object detectors are able to automatize many difficult, tedious tasks. However, they are usually slow and/or require powerful hardware. One main reason is called Batch Normalization (BN) [1], which is an important method for building these detectors. Recent studies present a potential replacement called Self-normalizing Neural Network (SNN) [2], which at its core is a special activation function named Scaled Exponential Linear Unit (SELU). This replacement seems to have most of BN's benefits while requiring less computational power. Nonetheless, it is uncertain that SELU and neural network based detectors are compatible with one another. An evaluation of SELU incorporated networks would help clarify that uncertainty. Such evaluation is performed through series of tests on different neural networks. After the evaluation, it is concluded that, while indeed faster, SELU is still not as good as BN for building complex object detector networks.

Contents

List of Figures

List of Tables

Acronyms

AUC Area Under Curve. 9

BN Batch Normalization. 1, 2, 5–7, 9, 21–23, 25, 27–29, 33, 34, 42–44

CNN Convolutional Neural Network. 5–7, 10, 11, 13, 14, 21, 22, 24–27, 33, 38–41, 44, 45

CPU Central Processing Unit. 29, 30

DSSD Deconvolutional Single Shot Detector. 15

ELU Exponential Linear Unit. 7

FCN Fully Convolutional Network. 16

FNN Feed-forward Neural Network. 7, 9, 21

GD Gradient Descend. 19

GPU Graphics Processing Unit. 29, 30, 33, 34

LRELU Leaky Rectified Linear Unit. 7

MLP Multi Layer Perceptron. 2, 5, 6, 24, 25, 27, 30, 33, 36–41, 45

RELU Rectified Linear Unit. 7, 9, 22, 25

RoI Region of Interest. 13, 14

RPN Region Proposal Network. 13, 14

SELU Scaled Exponential Linear Unit. 1–3, 5, 7–9, 21–25, 27–31, 33–46

SGD Stochastic Gradient Descend. 19, 30, 31, 35–38, 40–44

SNN Self-normalizing Neural Networks. 1, 27

SSD Single Shot MultiBox Detector. 14–16, 18

YOLO You Only Look Once. 14, 16–18, 25, 27, 31–33, 42, 43

1 Introduction

1.1 Motivation

Deep convolutional neural network based object detectors are computer programs, which detect and label objects in images automatically, without human interaction. These detectors are a crucial part in automatizing many tedious tasks that normally require human, such as: maintenance, surveillance, driving.

For these networks to be operational, they have to go through the process of training. However, trainings are often unreliable, and time consuming. One method to partially deal with this problem is called normalization. This method usually enables faster and more stable training of deep networks [1, 5, 2, 6]. The most widely adopted normalization method, for deep convolutional networks, is Batch Normalization (BN) [1]. However, BN adds an unignorable amount of calculations to the network. This raises hardware requirements, and slows down inference speed significantly – which will be shown in section 7.1. Therefore, resulting detectors are not suitable for applications that require high processing speed, and/or applications, in which powerful hardwares are unavailable. An example of such application is autonomous driving.

In order to speed up detector while keeping the training process manageable, a replacement for BN is likely required. One potential candidate is Self-normalizing Neural Networks (SNN) [2], which is built using Scaled Exponential Linear Unit (SELU) activation function. Utilization of SELU seems to yield similar normalization effects to BN with only a fraction of BN's computational cost. Nevertheless, there are still problems. SELU is built specifically for MLP. Furthermore, it relies heavily on mathematical theorems, and assumptions. This means any change in its operating environment could strongly affect SELU's functionality. There is no concrete evidence to claim that SELU would function properly inside arbitrarily-configured convolutional neural networks. Although there are several successful SELU-Based convolutional networks [7, 8, 9, 10], these simply apply SELU without further considerations. These projects do not mention whether SELU's normalization effect actually happens within the hidden layers.

1.2 Problem Statement

With the long-term goal of speeding up detectors without abandoning the benefits of normalization, this thesis intends to evaluate SELU's normalization effect in convolutional neural networks, as well as look into the effects of different variables on SELU's functionality.

1.3 Challenges and Difficulties

For this work, several challenges and difficulties are expected. The first challenge is determining how to check whether SELU operates properly inside a network. Due to the black-box nature of neural networks – especially deeper networks – simply adding SELU to a network, and looking at its accuracy result is not enough to validate SELU's functional correctness. Those positive accuracy results could be produced by SELU functioning only as an activation function, without any of its normalization effect. They could also be entirely due to other parts of the network – in which case, SELU contributes little or even causes harm to the network.

The second challenge is finding out how different parameters are affecting SELU's operation. The black-box nature of neural networks is again a big difficulty in this task. Performance of networks, and by extension SELU, are determined by many parameters: layer amount, layer type, neuron amount, training duration, optimizer, etc. There are also other unknown parameters. Thus this project will certainly not produce an exhaustive list. For this task, it must be ensured that only one variable is different from the control configuration in each and every test.

1.4 Thesis Outline

This report is divided into 8 chapters:

- **Introduction**: This chapter is intended to give readers a general idea of reasons for doing this project, core tasks of this project, and expected challenges.
- **Background**: Background chapter provides necessary knowledge for understanding this work.
- **Related Work**: This chapter presents some earlier scientific researches that are relevant to SELU and this thesis. Their shortcomings are also discussed in this chapter.
- **Approach**: Chapter 4 describes approaches that are taken to properly evaluate SELU.
- **Network Architecture**: This chapter gives detailed information on different architectures, which are tools used for performing core tasks of this project.
- **Experiment Setup**: Chapter 6 describes experiments that are performed in the scope of this thesis.
- **Results**: This chapter presents outcomes of the aforementioned experiments.
- **Conclusions**: Final chapter is dedicated to presenting contributions, learned lessons, and potential future works.

2

Background

The purpose of this chapter is providing readers with the background and relevant scientific concepts which are necessary for understanding the work in this thesis. These concepts are presented in the following order: normalization methods, convolutional network building techniques, object detection networks, and optimizers.

2.1 Normalization Methods

In neural network, normalization means transforming a set of value so that it has a certain distribution. In most case, normal distribution with mean $\mu = 0$, and variance $\sigma^2 = 1$ is used.

Normalization is mainly used to tackle two problems: covariate shift [11], and exploding/vanishing gradient [12].

- **Covariate shift**: layers' input distribution changes during training. This is caused by changes of previous layers' variables. Covariate shift slows down neural network training because it forces networks to also adapt to input distribution changes.

- **Exploding/vanishing gradient**: scale of calculated gradients is too large/small in comparison to layers' variables. This is caused by the large difference in scalar scale between hidden layers' output. For example, layer 5, and layer 6 output are in the range of [0,1], and [0, 100], respectively. This causes the exploding gradient problem. Exploding gradient destabilizes training, and can lead to divergence. On the other hand, vanishing gradient slows down training, and can result in a complete halt.

Solving those problems allow networks to be trained reliably, and faster.

There are several methods of normalization: layer normalization [13], weight normalization [14], batch normalization (BN) [1], and self-normalizing neural network (SNN) [2]. They are different on their methods of transformation and/or the set of value, on which they apply transformations.

- **Layer normalization**: performs explicit transformation on each layer's output for each training sample – not batch.
- **Weight normalization**: performs normalization on each layer's weight by separating weight vectors' length, and direction. After that optimization is done directly on the length component, and direction component, instead of the weight vector itself.
- **Batch normalization**: performs explicit transformation feature-wise on layers' output across each training batch.
- **Self-normalizing neural network**: performs implicit normalization on layers' output using SELU activation function, and LeCun weight initialization [16].

This thesis focuses only on BN, and SELU. In which, BN is used as benchmark, and SELU is the main test subject. Reasons for this decision are the following. BN is currently the most successful and widely used normalization method [5, 2, 13, 14], thus it is a good choice for a benchmark. This is especially true for convolutional neural networks – which is the target network type of this thesis. On the other hand, SELU is a new method, which seems to be faster than BN without sacrificing much of the normalizing effect. Furthermore, SELU presents a novel way of implicitly normalizing layers' output, which is different from other normalization methods. Section 2.1.1 will provide more details on BN. Details on SELU can be found in section 2.1.2.

2.1.1 Batch Normalization

Batch normalization [1] is the current de facto standard in deep CNN [5, 2, 13, 14] for dealing with exploding/disappearing gradients. BN is a technique of explicitly performing normalization on the input of hidden layers, not just the network's first input. Normalization is performed over each mini-batch, or the entire dataset. BN is applied to each feature individually. A special property of BN is that its calculation during training, and inference are different. For training, BN consists of two steps. The first step, which is on the left, is normalization using mini-batch mean, and variance. On the right side is the second step: "scale and shift".

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad y_i \leftarrow \gamma \hat{x}_i + \beta$$

μ_B, σ_B^2: mean, and variance over mini-batch

γ, β: must-be-learned parameters

The "scale and shift" step allows layers to retain their representation after normalization. For inference, there is no mini-batch. Therefore, normalization has to be done differently. Running mean E[x], and running variance Var[x] are calculated using mean μ, and variance σ^2 of all mini-batches that were used during training.

$$E[x] \leftarrow \frac{1}{N} \sum_{n=0}^{N} \mu_n$$

$$Var[x] \leftarrow \frac{m}{m-1} \left(\frac{1}{N} \sum_{n=0}^{N} \sigma_n^2 \right)$$

$$y = \frac{\gamma}{\sqrt{Var[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma E[x]}{\sqrt{Var[x] + \epsilon}} \right)$$

N: number of trained mini-batch

m: mini-batch size

γ, β: learned during training

For CNN, normalization is not applied to each individual feature. Instead, it is applied feature-map wise. All activations in each feature-map are normalized together. Each feature-map is assigned one pair of (γ, β). A note on using BN, β essentially acts as bias, thus a separate bias in the convolutional layer is not necessary.

In the original BN paper, BN is placed before activation function, such as: Rectified Linear Unit (RELU), Leaky Rectified Linear Unit (LRELU), sigmoid. However, bench tests [15] suggest that placing BN after RELU, or LRELU leads to better result. BN improves the stability, and training speed of neural network, especially deep CNN [1]. Nonetheless, BN adds additional computations to the network. Specifically, two must-be-trained parameters for each and every feature-map in a CNN.

BN's performance in Feed-forward Neural Network (FNN) is not as good as its CNN counterpart. FNNs trained with BN suffer from perturbations and have high variance in the training error. This high variance hinders learning and slows it down [2].

2.1.2 Self-Normalizing Neural Networks

Self-Normalizing Neural Networks [2] are networks with SELU, which is a customized Exponential Linear Unit (ELU) activation function. Unlike RELU, and other sigmoid functions, an extra step must be done to correctly utilize SELU. Weights in the network must be initialized using LeCun normal distribution [16] with mean $\mu = 0$, and variance $\nu = \frac{1}{n}$. In which, n is the size of input. For example, in case of a convolutional layer with kernel of size 3x3, n = 9. There is another important requirement for SELU to operate correctly – Each neuron in a layer must be connected to a large number of inputs.

A normalization method's final goal is to ensure that activations have zero mean, and unit variance over a mini-batch, or the entire dataset. SELU's goal is similar. However, there are three differences:

- Output of SELU's activation are already normalized, thus requiring no explicit normalization operation like in BN.

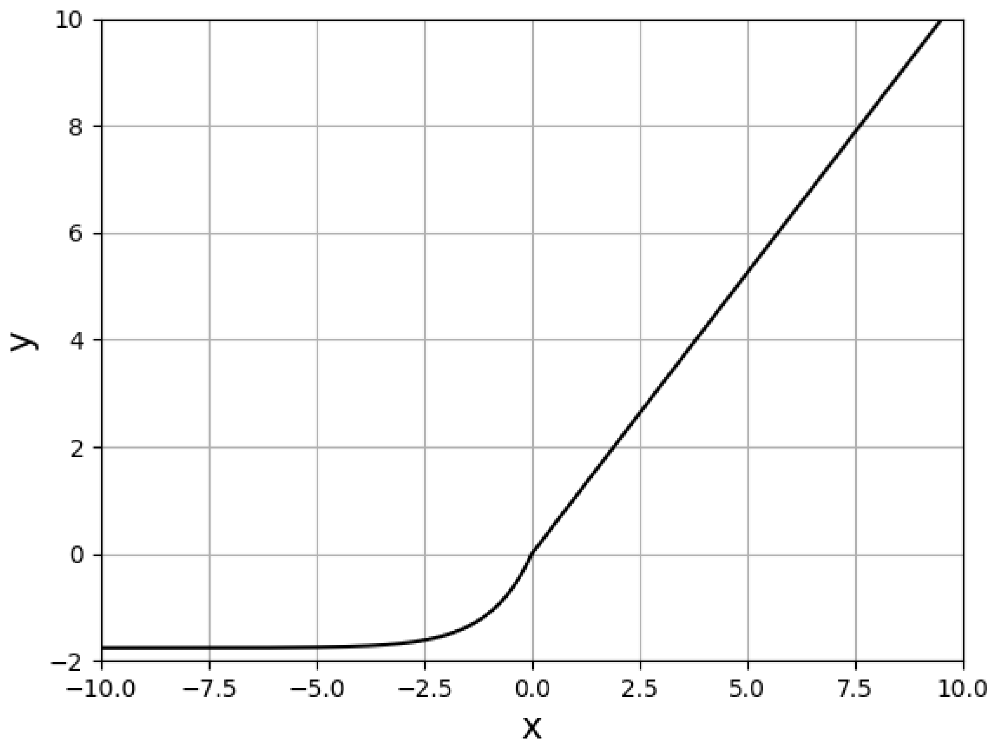

Figure 2.1: SELU activation function with $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$

- In SELU, desired activation's mean, and variance (μ, ν) can be in the range of $[-0.1, 0.1]$, and $[0.8, 1.5]$ respectively. Naturally, zero mean, and unit variance are still the most desired values.
- SELU does not ensure that earlier layers' activations have desired (μ, ν).

The core idea behind SELU is the following: Given that all weight matrices are initialized using LeCun normal distribution, and network inputs are well conditioned (normalized, and standardized) – definition in section 6.2. By applying SELU to each and every layer, the (μ, ν) of previous layers will push the (μ, ν) of later layers closer to a fixed point in the desired range. As a result, SELU's full potential can only be realized in networks with high number of layers. Here is the actual implementation of the above idea:

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

$$\alpha \approx 1.6733 \text{ and } \lambda \approx 1.0507$$

SELU's author provided a good explanation for this activation function:

> *"The activation function is required to have (1) negative and positive values for controlling the mean, (2) saturation regions (derivatives approaching zero) to dampen the variance if it is too large in the lower layer, (3) a slope larger than one to increase the variance if it is too small in the lower layer, (4) a continuous curve."* – Klambauer [2]

In the SELU paper [2], for empirical evaluation, SELU is compared against RELU (without BN), BN, layer normalization, weight normalization, highway networks, and residual networks. In term of Area Under Curve (AUC), SELU outperforms other techniques on FNN with high number of layers. For example, with the Tox21 challenge dataset, on 32 layer FNNs, SELU receives a score of 82.5 while BN gets 76. Since SELU only replaces other activation functions (i.e RELU), it adds negligible amount of additional computations to an existing network.

It is important to note that residual/skip connections might negatively affect SELU [17]. On a different note, the difference in accuracy between a SELU+BN combination and pure SELU is only marginal [9, 18]. This means that SELU's performance is not further improved by combining it with BN, and vice versa.

2.2 Convolutional Neural Network Building Techniques

Normalization methods by themselves are not useful. To properly evaluate these methods, they have to be placed inside a functioning neural network. To gather more data, each method should be tested with two different networks. The first network should be a smaller, carefully built, and controlled network for precision testing. The architecture of this network is described in section 5.2. The second network should be a complex, high performing network for competency testing. Details on such networks can be found in section 2.3. This section focuses on

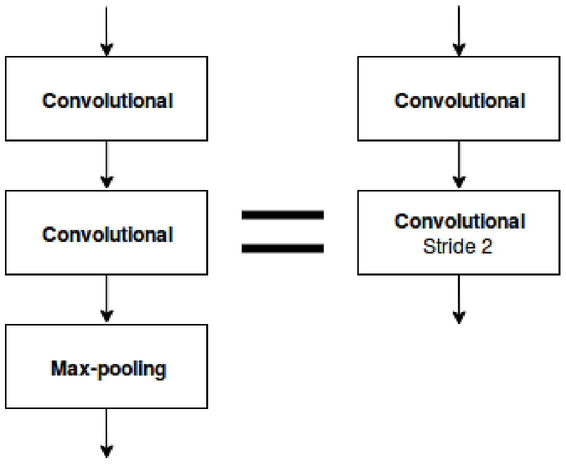

Figure 2.2: Explicit Max-pooling layer functions similarly to 2-stride Convolutional layer

introducing different techniques that are used for building the first network, as well as those that are used in complex CNNs. These techniques are listed below:

- Max-pooling with 2-stride convolution
- Kernel-doubling design
- Residual connection
- Anchor box
- Region proposal
- Region of interest pooling

Max-pooling with 2-stride convolution, and kernel-doubling design are used to build the first network.

2.2.1 Max-pooling with 2-stride Convolution

First is some information on pooling. Pooling means combining values from a cluster of neurons on a previous layer into a single neuron in the next layer. There are max-pooling, average-pooling, and min-pooling. However, max-pooling is the

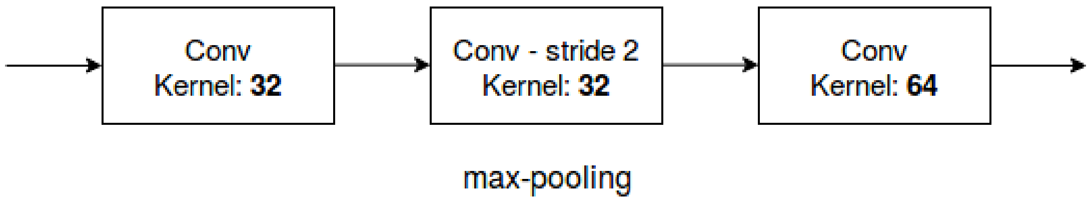

Figure 2.3: Example of kernel-doubling design

most useful and often used. It allows networks to converge faster, and also makes networks more tolerant to local changes in inputs [19, 20]. On the other hand, especially in deep CNNs, pooling leads to loss of information, which stops networks from learning deeper features [21].

Normally, max-pooling is explicitly performed by placing a max-pooling layer after a convolutional layer. However, it has been found that max-pooling's full benefit can be achieved by setting a convolutional layer's stride to 2 [22]. This design enables cleaner and easier-to-implement CNNs. Therefore, it is being used in several recent deep CNNs [4, 5, 23, 24].

2.2.2 Kernel-doubling Design

Kernel-doubling design means increasing the amount of kernels by 2 times after pooling is performed. This design is first used in VGG-16. The main benefit is mitigating the loss of information associated with pooling, as mentioned in section 2.2.1 [21]. The kernel-doubling design is used in many complex deep CNNs, especially those with VGG-16 as backbone [4, 5, 25, 26].

2.2.3 Residual Connection

Residual connection [24] – also called skip connection – is defined as taking input of a layer and perform element-wise addition to the output of that same layer, or another layer several levels deeper. Residual connection is designed to improve training of deep CNNs. It is shown that deep networks with residual connection converge better, and achieve higher accuracy [24]. This is due to the connection preventing a bad layer from ruining the whole network by skipping

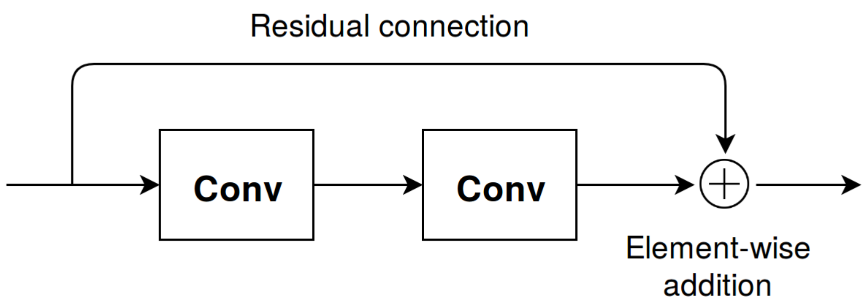

Figure 2.4: Example of a residual (skip) connection

it. Another advantage is that residual connections are easy to implement, and they adds an insignificant amount of computation to the network. However, since residual connection performs element-wise addition, two addends matrices must have the same size.

Residual connection is used in several large networks, such as: ResNet [24], Inception-resnet [27], RetinaNet [28], and YOLO v3 [5].

2.2.4 Anchor Box

Anchor box [25] is only applicable to bounding box detectors. Anchor boxes are rectangular boxes with different known sizes, and aspect ratios. Their sizes, and ratios are either chosen by hand [25], or dynamically using K-mean clustering [29] before the network is trained. These chosen sizes, and ratios of anchor boxes are not changed by the training. During training, transformations from these anchor boxes to object bounding boxes are learned. The main idea behind the anchor box technique is giving networks prior knowledge of objects. Therefore these networks do not have to learn object size, and aspect ratio from scratch, thus allowing training to be better.

2.2.5 Region Proposal

Region proposal consists of several techniques, its purpose is to give prediction on locations that have high chance of containing valid objects. Region proposal does not output class or bounding box predictions. Some examples are Selective Search

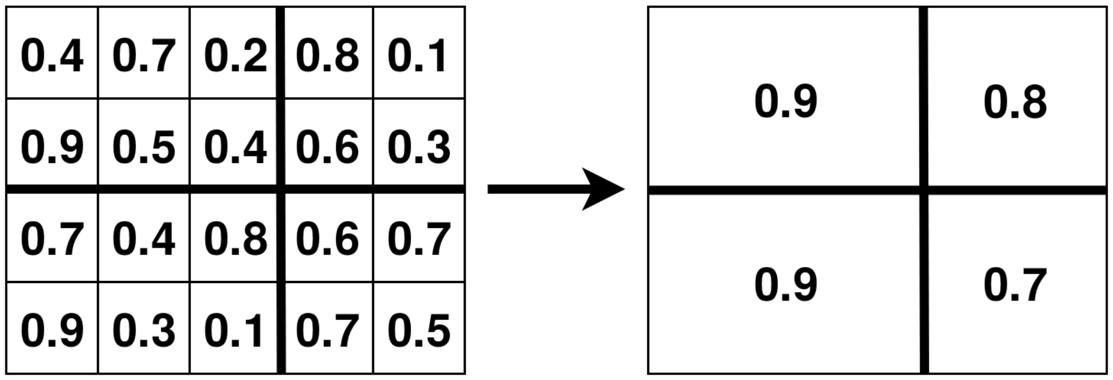

Figure 2.5: Example of a 4x5 to 2x2 RoI pooling

[30], SPPnet [31], and Region Proposal Network (RPN) [25]. In those example, RPN – which is introduced in the Faster-RCNN paper – is the fastest and simplest to use [25]. Unlike SPPnet – which has to be trained and used separately from a detector network – RPN can be trained together with the whole detector network. RPN is a fully convolutional neural network. Aside from regions, it also gives out two scores: object, and not-object. For a region, if the object score is higher, it is passed to the detector for classes and bounding boxes prediction. However, if the not-object score is higher, that region is discarded.

2.2.6 Region of Interest Pooling

Region of Interest (RoI) pooling is used in conjunction with region proposal. The purpose of RoI pooling is to make all proposed regions have the same size. This is done by dividing each region into a grid of desired size. After that, max pooling is performed on each cell of the grid.

2.2.7 Discussion

Techniques mentioned in this section have been used extensively in recent CNN. The first three techniques are compatible with any applications that makes use of CNN. Anchor box is limited only to bounding box predictor. However it has been proven to be effective, and leads to good results [25, 29]. Region proposal, and

RoI pooling are useful for detector networks that focus on accuracy. Nevertheless, networks with region proposal are often slower than other detectors [5, 25, 32]. Region proposal, and RoI pooling are often used together.

Regarding the small network for precision testing – which is mentioned at the start of this section – only the first two techniques (max-pooling with 2-stride convolution, and kernel-doubling design) are used to build it. Details on the reasoning can be found in section 5.2.

2.3 Object Detection Networks

This section introduces some successful object detection networks, which can be used for evaluating a normalization method's competency in complex, real application. Since this thesis focuses on detection speed, only bounding box detectors – which is the simplest, and fastest type – are considered. Thus, object mask detectors are not included in this work. These bounding box detection networks are: Faster R-CNN, Single Shot MultiBox Detector (SSD), and You Only Look Once (YOLO) version 3.

2.3.1 Faster R-CNN

Faster R-CNN [25] is a two-stages object detection network. These two stages are region proposal, and detection.

- Region proposal: this stage suggests areas in input feature-maps, where objects are likely to be located.
- Detection: this stage processes proposed regions to produce classes, and bounding boxes prediction.

The general architecture of Faster R-CNN is shown in figure 2.6. Input images are first fed into a CNN for features extraction. After that, produced feature-maps are given to the RPN [25]. RoI pooling is then performed on feature-maps based on proposed regions from the RPN. Finally, all resized regions are fed to the detector part of the network. Classes and bounding boxes are predicted separately by different sections of the detector part.

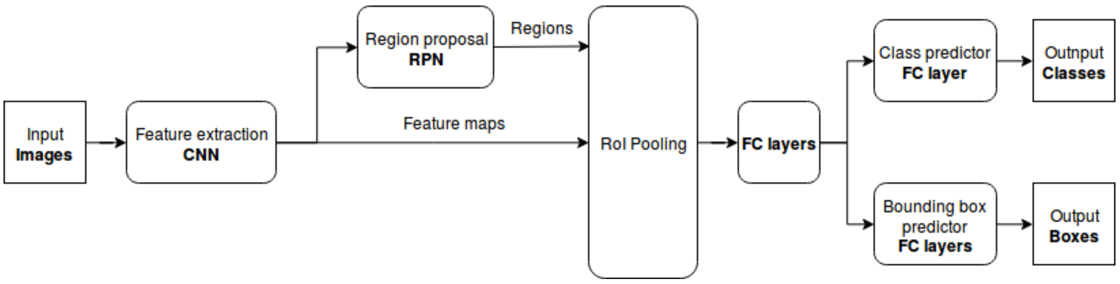

Figure 2.6: Faster R-CNN overall architecture [3].

Regarding further work, Faster R-CNN has been extended to detect masks instead of just bounding boxes [26]. Faster R-CNN was also used as baseline for discovering the problem of foreground-background imbalance in object detections [33].

Two-stages detectors, such as Faster R-CNN, focus on accuracy. While faster than other two-stages detector networks – such as R-CNN, Fast R-CNN[32] – the region proposal stage is known to be costly [4, 5, 34]. Therefore, these networks are usually slower than single-stage detectors. Single Shot MultiBox Detector (section 2.3.2), and You Only Look Once 2.3.3 are two examples of single-stage detectors.

2.3.2 Single Shot MultiBox Detector

Single Shot MultiBox Detector [4] is a single-stage object detection network. Unlike Faster R-CNN, SSD does not depend on region proposal. Input data is processed by the network only once. Furthermore, classes, and bounding boxes are predicted together. An overview of SSD can be found in figure 2.7. Input images are first given to VGG16 [21] for features extraction. Resulting feature-maps are then given to a series of convolutional layers. To detect objects with different sizes, predictions are made using individual output of each convolutional layer in the series. Early layers are responsible for detecting small objects. Meanwhile, later layers handle the detection of large objects.

SSD has been used as base for several other detectors. Deconvolutional Single Shot Detector (DSSD) [35] is one of those networks, which was built by combining SSD, and Residual-101 [24]. SSD has also been used for object pose prediction [36].

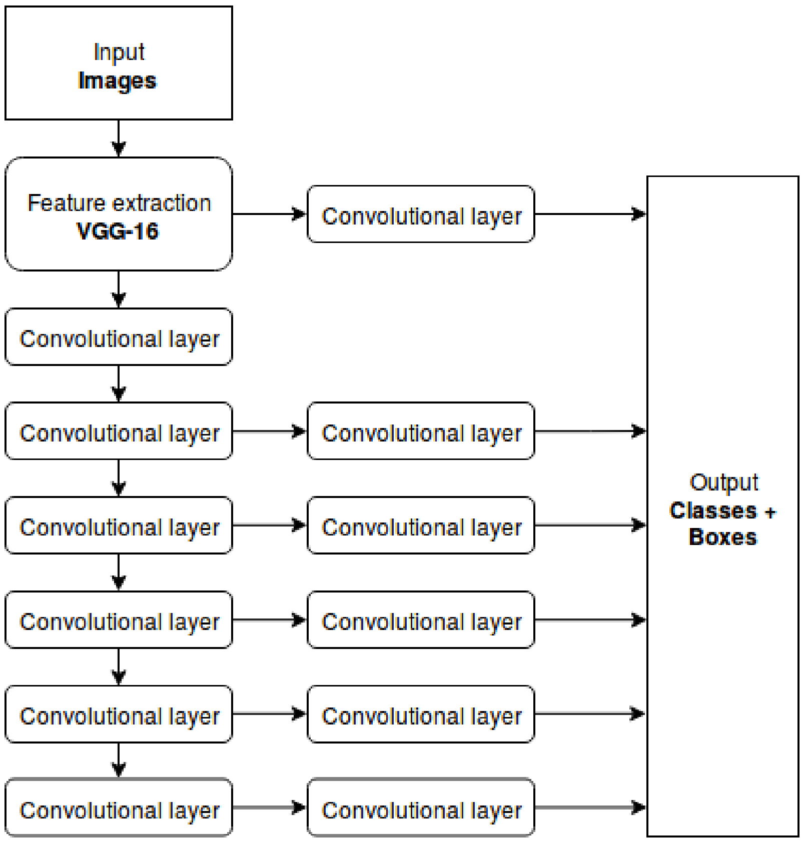

Figure 2.7: Single Shot MultiBox Detector overall architecture [4].

2.3.3 You Only Look Once

You Only Look Once version 3 [5] is a Fully Convolutional Network (FCN) for object detection. Similar to SSD, YOLO v3 is also a single-stage detection network. YOLO works by dividing input images into square cells, and utilizing multiple anchor boxes per cell to localize, and classify objects. Anchor boxes' size, and aspect ratio are picked using K-mean clustering. Due to its large amount of layers, YOLO is susceptible to covariate shift, and exploding/disappearing gradients problems. Batch Normalization is used to deal with this issue.

A YOLO [5] network has 2 parts: Darknet-53, and Yolo-heads. Darknet-53 is the backbone, which handles features extraction. This part consists of 52 convolutional layers. Regarding the 53rd layer, it is a fully-connected layer that exists only

	Type	Filters	Size	Stride
	Input			
	Convolutional	32	3x3	1
	Convolutional	64	3x3	2
1x	Convolutional	32	1x1	1
	Convolutional	64	3x3	1
	Residual			
	Convolutional	128	3x3	2
2x	Convolutional	64	1x1	1
	Convolutional	128	3x3	1
	Residual			
	Convolutional	256	3x3	2
8x	Convolutional	128	1x1	1
	Convolutional	256	3x3	1
	Residual			
	Convolutional	512	3x3	2
8x	Convolutional	256	1x1	1
	Convolutional	512	3x3	1
	Residual			
	Convolutional	1024	3x3	2
4x	Convolutional	512	1x1	1
	Convolutional	1024	3x3	1
	Residual			
	Output			

Table 2.1: Darknet 53 structure [5]

during pre-training. It is not a part of the final YOLO architecture. Darknet-53 is built using several techniques: pooling with 2-stride convolutional layers [22], kernel number doubling after pooling [21], residual/shortcut connections [24], and dimensionality reduction with 1x1 convolution [37]. It contains 5 large convolutional sections. Each is a sequence of several similar concatenated residual blocks. A block consists of two convolutional layers and a residual connection. The first layer halves the amount of kernels from its input using 1x1 convolution. Its job is to lower the number of operations the network has to perform, without sacrificing performance significantly. The second layer is a normal 3x3 convolutional layer with the same amount of kernels as the block's input. The residual connection performs element-wise addition between the second layer's output and the block's input to produce the block's output. In front of each sections is a 2-stride convolutional layer. It performs pooling, and doubling the amount of feature-maps.

Yolo-heads is the second part of the network. This part is for fine-tuning, and localizing detected objects within images. Its design focuses on allowing the network to detect objects of different scales. There are three heads: 13x13, 26x26, and 52x52. Each head's name denotes its output size, given the default input size of 416x416. Head 13x13 is a series of 7 convolutional layers. It takes the output of Darknet-53's last section as input. Head 26x26, and 52x52 are similarly constructed. However, 26x26's input comes from the fifth convolutional layer of 13x13, and the output of Darknet-53's second to last section. For 52x52, input comes from the fifth convolutional layer of 26x26, and the backbone's third to last section.

There are further projects based on YOLO. ROLO [38] is an object tracking system, which is built by combining YOLO with Long Short-Term Memory network. Another object tracking system is named MV-YOLO [39], which uses YOLO's detection and motion vector from input stream. YOLO is also used as a baseline for the "EuroCity Persons" [40] dataset.

2.3.4 Discussion

As stated in the introduction, this thesis focuses on network inference speed rather than accuracy. In order to push the speed limit, it would be better to work with the fastest network, which still has respectable classification performance. With a GeForce GTX TITAN X graphic card, inference time for R-CNN, SSD, and YOLO v3 are 85 ms, 61 ms, and 22 ms respectively [5]. It is clear that YOLO v3 is the fastest network of the three. As a result, YOLO v3 is chosen as the base for testing in this thesis.

2.4 Optimizers

Architecture alone is not enough to bring a neural network into operational state. Training procedure, and hyperparameters have major effects on a network's performance. In order to be tested properly, a network should be trained with different configurations. One of the most important choices for training is optimizer.

This section focuses on introducing some optimizers, which are used for testing in this work.

Optimizers are cores of the back-propagation process, which trains neural networks. The basic idea behind optimizer is calculating gradients of weight matrices and then update weights to make loss value as small as possible. There are several different optimizers. The only significant difference between them is how they calculate update values for weights after gradients are found. Some of those optimizers are Gradient Descend (GD), Stochastic Gradient Descend (SGD), SGD with momentum [41], RMSProp [42], AdaDelta [43], and Adam[44].

- **Gradient Descend**: It is the base of all other optimizers. However, GD is not always suitable for neural network applications. GD needs to process the entire dataset to produces one update. If the dataset is large, training will take a large amount of time. Furthermore, GD assumes that the dataset never change.

- **Stochastic Gradient Descend**: SGD is a trade-off between accuracy and speed. Instead of calculating gradients from the entire dataset, SGD does it with a single sample, or a small batch of samples. Although gradients are less accurate, SGD is much faster and can handle ever-changing dataset. Update is controlled by a hyperparameter: learning rate η. Recommended value for η is in from 0.1 to 1 [45].

- **Stochastic Gradient Descend with momentum**: SGD with momentum extends SGD by taking the last update value into consideration. The idea is analogue to momentum in physics. The more weight matrices change in one direction, the larger the next update in that direction will be. Furthermore, sudden changes in direction would be slowed down. This momentum is controlled by a hyperparameter: momentum γ. Recommended value for γ is either 1.0 [46] or 0.9 [47].

- **RMSProp**: It is another extension of SGD. Instead of just looking at gradients from the last iteration, it considers all past gradients. This is done by keeping a moving average v of squared gradients. New average is a weighted sum of current average and the newest gradient squared. The weight in this sum is controlled by a hyperparameter called decay rate ρ. Recommended value for ρ is 0.9 [42]. Update values are calculated using gradients, moving average v, and learning rate η.

- **AdaDelta**: AdaDelta is similar to RMSProp. It also maintains a moving average of past squared gradients. However, it is different in its method of calculating update value. AdaDelta does not depend on learning rate η. Instead, updates are calculated with gradients, updates from last iteration, moving average v, and a smoothing value ϵ. $\rho = 0.95$, $\epsilon = 10^{-8}$ are recommended values for these hyperparameters [43].
- **Adam**: Adam is built upon both RMSProp, and AdaDelta. This optimizer calculates two moving averages: one for squared gradients, and another for gradients. They are controlled by hyperparameter β_1, and β_2, respectively. Moving averages, learning rate η, and smoothing value ϵ are used to calculate updates. Empirical evidences show that $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ are recommended values [44].

There are also other existing optimizers, such as: AdaMax, Nadam, AMSGrad. However, this thesis will only focus on those aforementioned ones, because they are often used, and have been shown to give good results [48]. Regarding difference in performance, optimizers with adaptive learning rate usually function better [49]. For example, Adam, AdaDelta, and RMSProp give faster convergence, and allow deep networks to be trained successfully [43, 44, 48].

3

Related Work

This chapter briefly introduces some related works, and their shortcomings in investigating SELU's usage in convolutional neural network.

3.1 SELU-applied projects

SELU has already been employed for several machine learning projects. Linked neuron paper [18] shown that reliable training on deep CNN, such as AllCNN and ResNet, can be achieved with SELU and special neuron configurations. Integration of SELU in CNN has also produced good result on point cloud segmentation [7], radio/microwave time series classification [8, 9], Omics data classification [10]. Regarding FNN, some deep networks get better result with SELU [50, 51]. SELU has also been applied to reinforcement learning [52, 53], auto-encoders [54, 55]. There is also project that used SELU together with residual connection. However, the result was not positive [17]. On another note, several works [9, 18] have tried to use SELU together with BN. The difference in accuracy is, however, only marginal. This means that SELU's performance is not further improved by combining it with BN, and vice versa.

3.2 Limitations of previous works

The SELU paper only focuses on Feed-forward Neural Network. The paper does not explore the effects of SELU applied into deep CNN. Furthermore, the original SELU network was evaluated using a medical drug dataset, whose nature can be different from that of an image dataset. From the SELU paper alone, the effects of SELU on a CNN cannot be clearly determined. Later, SELU's authors did perform some testing with CNN, which is available on their Github repository[1]. However, these networks have only two convolutional layers. Consequently, SELU's effects on much deeper CNN are not clearly demonstrated.

Linked neuron paper [18] did several tests between RELU, SELU, Swish [56], and their customized activation functions. However, batch normalization is used together with SELU in most of these tests. This complicates the test, and makes it difficult to correctly evaluate SELU. There are tests without BN, but these are only between SELU and customized versions of activation functions. The comparison between RELU+BN and pure SELU is not clearly mentioned in this paper.

Most SELU-applied projects in section 2.1.2 work on different type of networks or non-image data, thus their results may not remain the same for an image processing deep CNN. Furthermore, some projects either make no comparison between SELU and BN [7, 10], or only use it once before the last fully-connected layer [8].

[1]https://github.com/bioinf-jku/SNNs

4

Approach

This chapter gives readers a general overview of approaches, which are used to explore SELU's performance, and characteristics. The two main focuses are SELU's inference speed, and SELU's reaction to configuration changes.

4.1 Inference Speed

For the first task, SELU's main advantage over BN – namely calculation speed – is tested. In this context, calculation speed is defined as how much time a network takes to calculate outputs from inputs. This is done by performing training and inference on two almost identical networks. Training configurations are also identical. The only difference between these two networks is the normalization method – either SELU or BN. Although these tests are not concrete enough to draw any final conclusion on whether SELU's normalization effect is working properly or not. For example, accuracy, and loss are not carefully controlled, thus these data are not trustworthy. These speed tests would still provide credible and tangible statistics on how fast SELU-based networks are in comparison to BN-based networks during inference. Information on calculation speed is reliable, because the amount of computation is still the same, even if SELU is not functioning correctly.

4.2 Parameters' Effects on SELU

The second task is finding out different variables' effects on SELU's functional correctness. This task consists of two parts. The first part is a series of tests on a

Multi Layer Perceptron. The second part is made up of similar tests, but they are performed on a CNN. For MLP tests, a control network is first built using nearly identical configurations from SELU's author [2]. For each test network, only one configuration is deviated from the control network. This ensures the link between a specific change in configuration, and a specific change in network behavior is most likely legit, and not coincidental. Hidden layers' outputs distributions from these modified networks are compared to that of the control. Tests on CNN follow the same procedure.

5

Network Architecture

The purpose of this chapter is providing information on network architectures, which are used for the two tasks mentioned in chapter 4. More specifically, this chapter gives detailed information on "Normalization Unit", "Generic CNN", "Control MLP", and "YOLO".

5.1 Normalization unit

A normalization unit is responsible for normalizing hidden activations within a network. Although it is not a complete architecture, it is an important building block for all architectures in this chapter. Normalization units are placed before all hidden layers. There are two types of normalization unit: BN with RELU (BN+RELU), and SELU. BN+RELU consists of Batch Normalization, and leaky RELU – a variant of RELU – with $\alpha = 0.1$. This alpha value of 0.1 was chosen based on YOLO v3's configurations, which has been empirically proven to function well [5]. As a side note, $\alpha = 0.01$ – which is recommended in the original leaky RELU paper [57] – could also be a good value. Meanwhile, SELU's parameters is configured for Gaussian distribution of zero mean and unit variance ($\alpha \approx 1.6733$ and $\lambda \approx 1.0507$). Each network will have only one type of normalization unit.

5.2 Generic Convolutional Neural Network

The generic CNN architecture is trained with the MNIST dataset [58] – an often used image dataset. This architecture is used for both task 1 and 2. In task 1,

	Type	Filters	Size	Stride
	Input			
5x	Convolutional	32	3x3	1
	Normalization			
	Convolutional	32	3x3	2
	Normalization			
5x	Convolutional	64	3x3	1
	Normalization			
	Convolutional	64	3x3	2
	Normalization			
5x	Convolutional	128	3x3	1
	Normalization			
	Convolutional	128	3x3	2
	Normalization			
	Dense		256	
	Normalization			
	Dense		10	
	Output			

Table 5.1: Generic CNN architecture

SELU-based variants and BN-based variants are used to compare calculation speed. In task 2, only SELU-based variants are used to investigate variables' impact on SELU. This architecture is used to make sure that changes in its behavior are most likely due to changes of specifically chosen variables during tests. A larger, more complex network would likely introduce unaccountable disturbances to test results. Details on generic CNN's structure is introduced below.

The generic base network consists of 18 consecutive convolutional layers and 2 final fully-connected layers. An overview of its structure can be seen in table 5.1. Except for the last fully-connected layer, each layer is followed by a normalization unit (see section 5.1). Regarding the choice of network depth, in the SELU paper [2], the author performed tests on 3 different depth: 8, 16, 32. In order to reduce the amount of time needed for each test, while still leaving the network deep enough for SELU's effects to be observable, depth of 20 layers is chosen. The convolutional section is divided into 3 smaller convolution blocks. Each block has 5 identical convolutional layers, and a final convolution layer with stride of 2. This last layer can replace explicit convolution and max-pooling combination without loss in

	Type	Size
	Input	
20x	Dense	784
	Normalization	
	Dense	10
	Output	

Table 5.2: Control MLP architecture

accuracy [22]. It is being favored in recent deep CNNs, therefore it is included in the generic architecture [4, 5, 23, 24].

Kernel size is always 3x3, which is chosen for its ubiquitousness in CNN. Number of kernels is doubled after each pooling, starting at 32 – this choice is based on YOLO v3 network. The kernel-doubling design, which is proposed in VGG-16 [21], allows networks to learn deeper features without losing information. This design is used in the generic network because it is being used to build many networks, especially those with VGG-16 as backbone [4, 5, 25, 26]. A fully-connected layer with 256 neurons comes after the convolutional section. Finally, the last fully-connected (detection) layer has 10 neurons, which is the same as the number of classes. Outputs of the detection layer are not normalized.

5.3 Control Multi Layer Perceptron

The Control MLP is a Multi Layer Perceptron/dense network, which is built using similar configurations from the SELU paper [2]. Like the generic CNN, it is also trained with the MNIST dataset. This architecture is only used for task 2. Therefore, it only has the SELU-based version, and no BN-based version. This architecture serves two purposes. First, it acts as a benchmark for verifying other SELU-based networks' functional correctness. Second, it is used to figure out which variables affect SELU's functionality the most. Because this architecture is close to the original SNN, causes of any deviation in SELU's behavior can be determined easier. The Control MLP's structure can be found in table 5.2.

5.4 You Only Look Once

The last network architecture is based on YOLO v3, which is summarized in section 2.3.3. For testing, BN is replaced with SELU. Variances of this architecture, with either SELU or BN, are trained with the COCO dataset [59]. Because of its complexity, and highly customized nature, this architecture is not suitable for both main tasks. However, it is still included to demonstrate SELU's performance in a complex, high-performing network.

6

Experiment Setup

This chapter presents information on the setup and hyperparameter setting of all experiments. This thesis consists of three large experiments – each contains several smaller tests.

- Experiment 1: Correspond with task 1 (section 4.1). This experiment focuses on comparing inference speed.
- Experiment 2: Correspond with task 2 (section 4.2). This experiment focuses on the relationship between SELU's behavior and hyperparameters.
- Experiment 3: This experiment demonstrates SELU's performance when it is näively applied into a complex convolutional network.

Regarding software and hardware setup, all networks and scripts are implemented in python 3. Machine learning aspects are handled with Tensorflow [60]. Experiments using Graphics Processing Unit (GPU) are performed on a cluster with a GeForce GTX TITAN X graphic card. CPU-only experiments are ran on an Intel i7-6700 Central Processing Unit (CPU) with 8GB RAM.

6.1 Experiment 1

First experiment includes a series of näive timing, and performance tests. These tests show results of applying SELU or BN to a generic architecture and training them with generic settings. In these tests, the generic network's variances are trained on different hardware settings, optimizers, and random seeds. Batch size is, however, fixed at 5. There are two variances of the generic network: SELU-Based, and BN-Based. Each of them is trained on GPU, or only CPU. This is done to show normalization choice's impact on network in different hardware settings. Furthermore, 4 different optimizers are tested. This is performed to observe reactions of normalization methods to different optimizers. Each training setting is repeated 5 times with 5 different random seeds. In total, there are 40 runs for each variance. During training, for every 100 iterations, the network is evaluated on the MNIST test set (10000 samples). Duration of each evaluation, and network's accuracy are logged. Duration of an evaluation is logged instead of each detection because of hardware reasons. With the aforementioned hardware, each detection is calculated very quickly, thus collected timing data is vulnerable to errors. Finally, Tensorflow's summary functionalities are used to record distributions of several hidden layers' outputs.

6.2 Experiment 2

Second experiment goes into SELU-Based network's reactions to different training settings. Instead of looking at the end performance of the network, these tests look at hidden layers' outputs to determine whether SELU is working as intended. The control of these tests is the Control MLP (table 5.2), which is built and trained using similar configurations from the SELU paper. Each hidden layer has 784 neurons, which is the number of pixels in each MNIST sample. Input dataset is preprocessed to have zero mean and unit variance. Optimizer is SGD with momentum. The original SELU network is trained with SGD, however preliminary tests show that momentum helps stabilizing the network without sabotaging SELU. Without momentum, some configurations diverge, and cannot be trained. The control is trained with batch size of 100 for 2 epochs. The first test subject is the same MLP network, however its training configuration is modified. The second test subject is the SELU-Based generic network, which is described in section 5.2. For each test, a change is made to one of the following settings: data preprocessing, batch size, or optimizer. Two methods of data preprocessing are used. The first (original) method consists of two steps: normalization and standardization. First, normalization means rescaling the dataset to $[0, 1]$ range. Second, standardization means transforming data so that the dataset has zero mean, unit variance. The second method is performing only normalization. This method is tested, because it is more suitable for large convolutional networks. These networks usually work with large image datasets, for which calculation of global mean and variance are resource intensive. Furthermore, these datasets are regularly-updated, thus calculated global values become obsolete quickly. Batch size of 5, 16, and 32 are used in comparison to the original 100. This test is done to see how important batch size is to SELU. Finally, optimizers' effects on SELU are examined. These optimizers are SGD-Momentum [41], RMSProp [42], AdaDelta [43], and Adam [44]. With the recommendation from Fu [61], SGD-Momentum's learning rate and momentum are set to 0.001 and 0.9, respectively. RMSProp, AdaDelta, and Adam optimizer are set up using recommended parameters from their respective authors. These tests are done to see whether more complex optimizers are compatible with SELU. Hidden layers' output distributions from each test are compared to that of the control. All networks are trained for 2 epochs.

6.3 Experiment 3

Third experiment is about testing SELU in a real, complex, high-performing network. For benchmarking, a BN-based YOLO v3 network is trained from scratch with only two labels. Pre-trained weight is not used to for training. Because SELU requires special initialization – and it is unsure whether SELU can be trained using pre-trained weight – this ensures the fairness of the experiment. Only two labels are used, so that training and testing cycles are shorter. The test subject is a SELU-based YOLO v3 network. Training configurations for the two YOLO variances are decided from the result of experiment 2. Two configuration conditions are drawn after experiment 2, namely: use large batch size, and use only SGD-Momentum. Details are presented in section 7.2. With those conditions, for this experiment, two batch sizes are chosen – 8 and 32. Regarding batch size 32, this is an often used large batch size in image processing. Larger batch sizes are not used due to hardware limitations – Out of memory error. Batch size 8 is another often used size, which is included for comparison's sake. As for optimizer, AdaDelta and SGD-Momentum are chosen. AdaDelta is used to observe the differences, which optimizer can make to large and complex network. In total there are 4 different training configurations for each YOLO's variant. All networks are trained for 1000 epoches. A checkpoint is saved every 20 epoches.

7 Results

This chapter presents the results of several experiments, which are done to evaluate SELU. Here is a quick summary of the following sections:

- Experiment 1: During inference, SELU is 1.55 to 2.83 times faster than BN, especially on weaker hardware. SELU also requires less physical memory. However, SELU does not function with all optimizers.
- Experiment 2: SELU retains its normalization property when it is applied into a CNN. However, this normalization effect needs more layers to function properly in comparison to SELU in MLP. SELU's normalization effect slows down when input data is ill-formed, or when input batch size is not optimal. Furthermore, SELU's normalization fails completely with complex optimizers, namely: RMSProp, AdaDelta, Adam.
- Experiment 3: SELU-based YOLO, even when configured properly, is not yet trainable. Batch Normalization in combination with complex optimizers is still better than SELU at training complex network.

7.1 Experiment 1

The first criteria is inference speed. A quick summary is shown in figure 7.1. This criteria is evaluated by measuring the amount of time a network required to process 10000 samples at batch size of 5. Without a GPU, BN-Based network

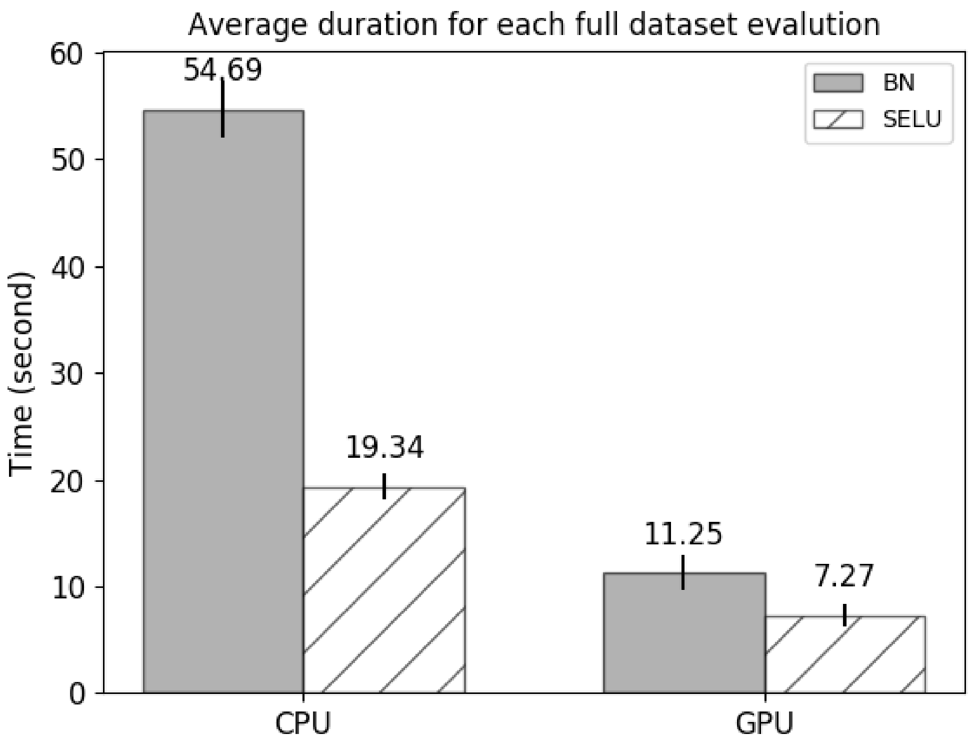

Figure 7.1: Average duration for each full dataset evaluation

requires on average 54.69 second. SELU-Based network needs 19.34 second on average. In this case, SELU-Based network is 2.83 times faster. With a GPU, BN-Based and SELU-Based takes 11.25 second, and 7.27 second respectively. The speed ratio is 1.55. It can be seen that the additional computations from BN has significant negative effect on networks' inference speed. SELU, which is only an activation function, is always faster than BN. Regarding the difference in speed ratio between two hardware setups, this might be due to better parallel computing capability, and larger physical memory offered by GPU. In summary, SELU is much faster than BN, especially in case of weaker, non-specialized hardware.

Regarding convergence, in comparison to BN-Based networks, SELU-Based networks require around 2 times more steps to reach its top test accuracy.

The last criteria is accuracy and reliability. Accuracy chart is shown in figure 7.2. Batch normalization outperforms SELU in this case. BN-based networks are able to

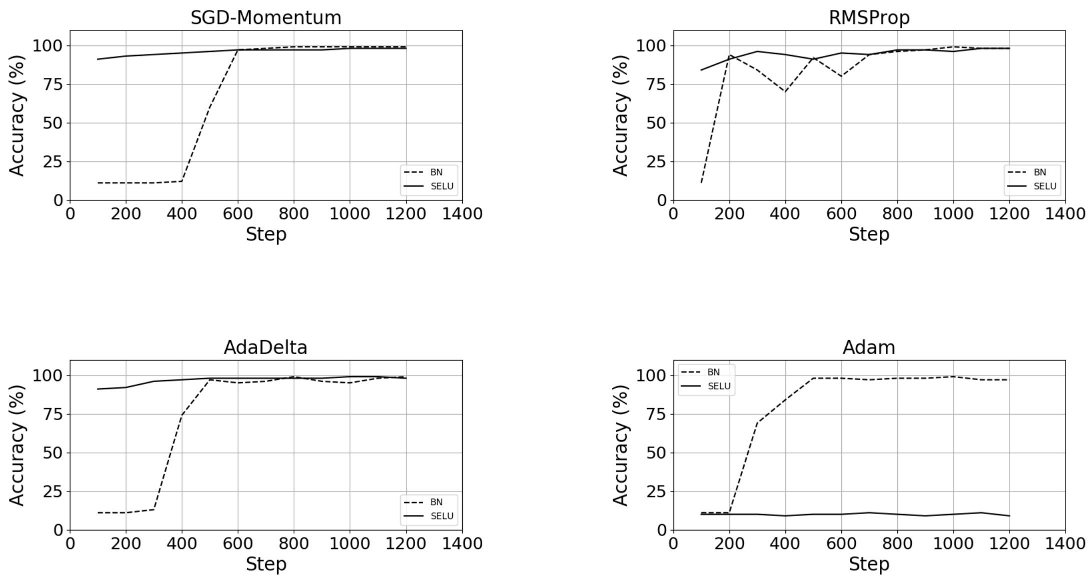

Figure 7.2: Accuracy during training with different optimizers

reliably reach 99% test accuracy with all 4 different optimizers: SGD-Momentum, RMSProp, AdaDelta, and Adam. For SELU-Based networks, only SGD-Momentum, and AdaDelta yield reliable 99% test accuracy. Networks trained with RMSProp can diverge during training. In case they do not diverge, the maximum accuracy is only 96%. Adam diverges, and fails in all runs. This indicates that SELU might not work with most complex optimizers. In case of AdaDelta, while networks are properly trained, it is unclear whether SELU is functioning correctly inside these networks.

7.2 Experiment 2

This experiment looks into SELU's functional correctness regarding changes in network's hyperparameters. To verify each network's functional correctness, shape of histogram, mean, and variance at the end of training are examined and compared to a reference. This reference, which is shown in figure 7.3, is created using SELU's original configurations. For comparison, activation distribution after batch normalization is also shown in figure 7.4. Next paragraph is a description of these figures.

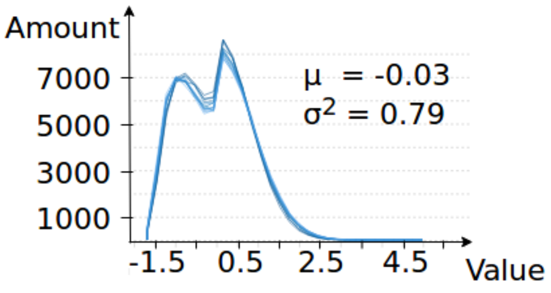

Figure 7.3: SELU reference

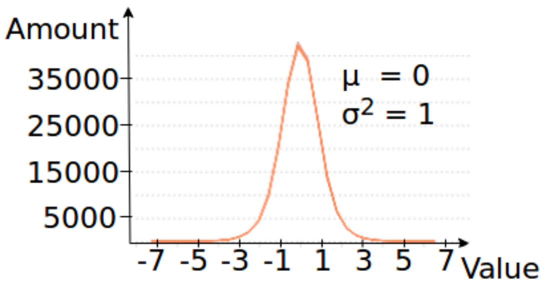

Figure 7.4: BN distribution

These histograms are automatically generated by Tensorflow [60]. They are histograms of all data at each layer's output while calculating a mini-batch – not a single image/sample. The x-axis represents data value. The y-axis shows the number of data points in each bin. Each plot does not consist of only one histogram but several overlapped curves, which are drawn every 100 training steps. The fuzzier the figure, the more output distribution shifted during training. There is an option to show these histograms in 3D – thus curves are separated, and sorted in temporal order – however, these figures does not render well on printed paper. For better visibility, 2D overlapped histograms are used. Mean and variance values in each figure are taken from the last histogram at the end of training.

As a side note, it is clear that SELU's reference variance is not 1. However, that is expected and discussed below.

The first series of tests are done on the Control MLP architecture (section 5.3). Default setting is batch size 100, SGD-Momentum optimizer, and data normalization + standardization preprocessing. Activation distributions of this reference configuration is always shown in the first row in figure 7.5, 7.6, and 7.7. These figures only show distributions of the first (1), middle (10), and last (20) layer. However they are good representatives for the whole network, as layers' behavior across the network is fairly consistence. With these visualizations, it can be confirmed that given the correct configuration, SELU functions exactly as its authors claimed. Histograms' shapes are consistent across all layers, and they change less the deeper the layer. Mean values stay around 0 for most of the time. Except for the first layer – which has unexpectedly good variance value – the deeper the network, the closer layers' variance values move to 1. Therefore

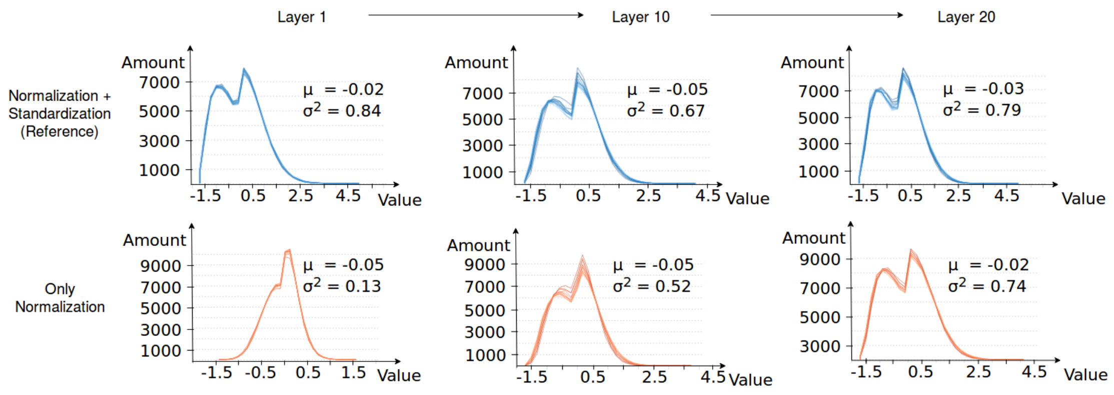

Figure 7.5: MLP: Activation distributions with different preprocessing techniques

the reference curve in figure 7.3 does not need to have variance of 1. From this point on, it is fairly safe to conclude that SELU is functioning correctly if any SELU-based network exhibits similar behaviors to the reference.

For the first test in the MLP series, data preprocessing method is changed from normalization and standardization to only normalization. As shown in figure 7.5, distributions of earlier layers are not well formed. However, the network quickly recovers, and becomes similar to the reference in deeper layers. Mean, and variance also move closer to their desired values. In this case, SELU is able to function properly and adjust network's activations if the network is deep enough.

The second MLP test is done by changing the batch size. From figure 7.6, it is shown that at smaller batch size, early layers' distributions are not stable. This also causes a lot of distribution shift during training. The lower the batch size the more layers SELU needs to adjust the output activation. For example, the network with batch size of 5 only starts to behave better at around layer 20. To sum up, SELU still functions with small batch size, however it is recommended to train SELU-based network with large batch size.

The third MLP test is done by changing training optimizers. These are SGD-Momentum, RMSProp, AdaDelta, and Adam. Looking at hidden layers' activations (figure 7.7), only SGD-Momentum-trained SELU network is working as intended. RMSProp, and Adam's internal activations' exploded in scale. AdaDelta's activations also increased, but not in the magnitude of the other two. This could be a reason why AdaDelta functions better than RMSProp, and Adam in experiment 1

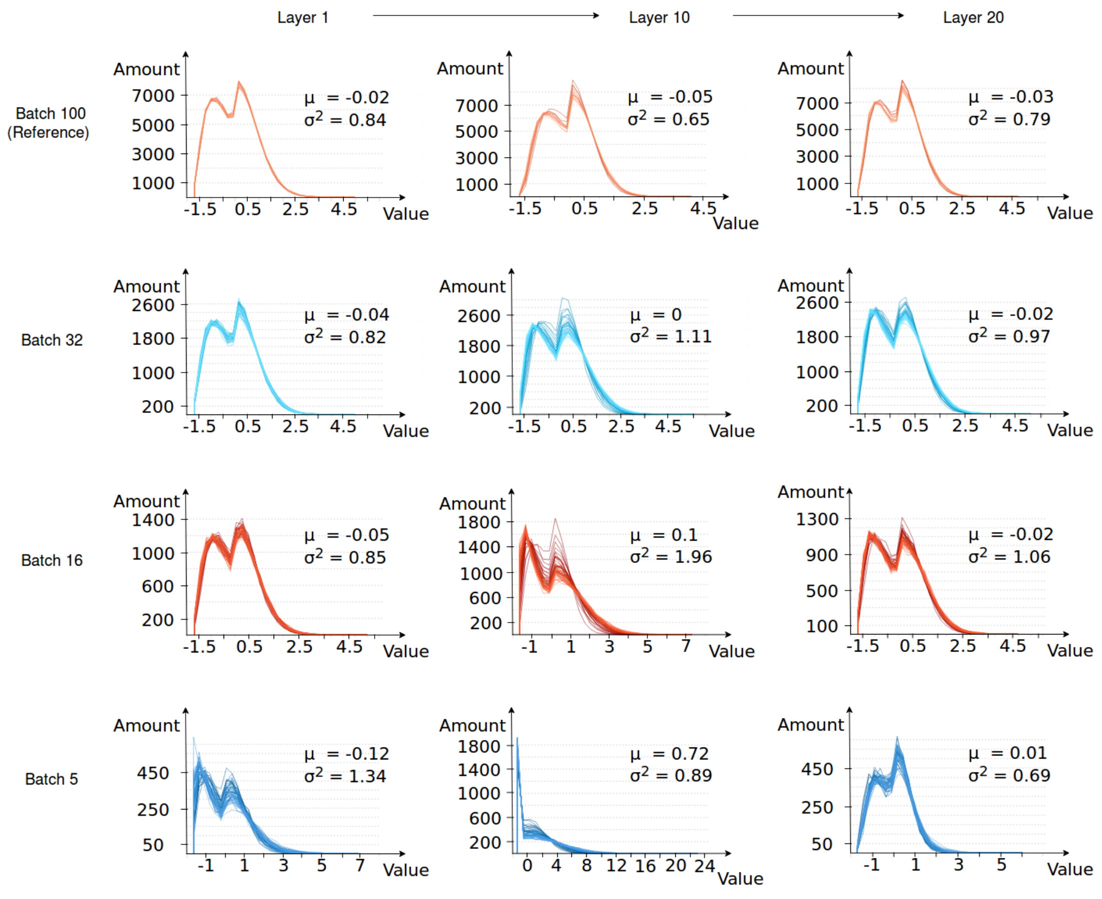

Figure 7.6: MLP - Activation distributions with different batch sizes

(section 7.1). According to this test, SELU does not seem to work with complex optimizers.

The second series of tests are done on the generic CNN architecture (section 5.2). Default setting is batch size 100, SGD-Momentum optimizer, and data normalization + standardization preprocessing. Figure 7.8 shows the comparison between the CNN's reference and the MLP's reference – which is the main one. It is important to note that CNN's figures in this report show histograms from layer 1, layer 6, and layer 17 (last convolutional layer). From experiments, it is seen that these layers represent the entire network's behavior well. Regarding the CNN reference, it is clear that SELU struggles inside convolutional networks. Although, variance values fluctuate around 1, especially in early layers, SELU still functions properly in this case. Mean values hover around 0. Variance values start to stabilize

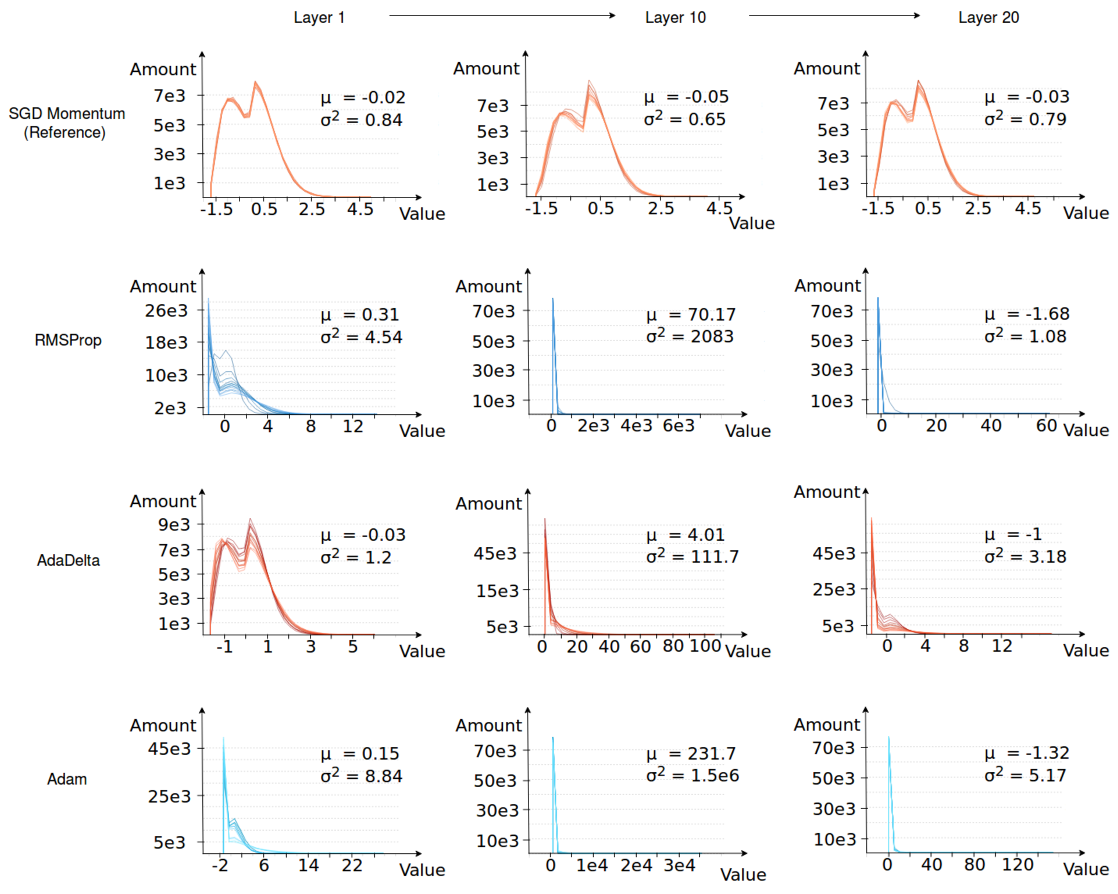

Figure 7.7: MLP - Activation distributions with different optimizers

around layer 15. Deep hidden layers' activations converge to the shape and values in the MLP reference. An example of this is layer 17's activation. In summary, the CNN reference network does contain correctly working SELU components – although not as good as the FNN reference network. Therefore, it is safe to use this network as benchmark for all other SELU-based CNN tests.

First, data preprocessing methods are compared (figure 7.9) – namely normalization with standardization, and only normalization. Hidden activations show that normalization-only method requires more layers for SELU to converge to the desirable distribution. This test result is similar to MLP. Additionally, raw data is fed to the network in one case. As a result, earlier layers' activations are not desirable. However, they move closer to the correct distribution as the network gets deeper. This further indicates that deep SELU-Based networks are capable of

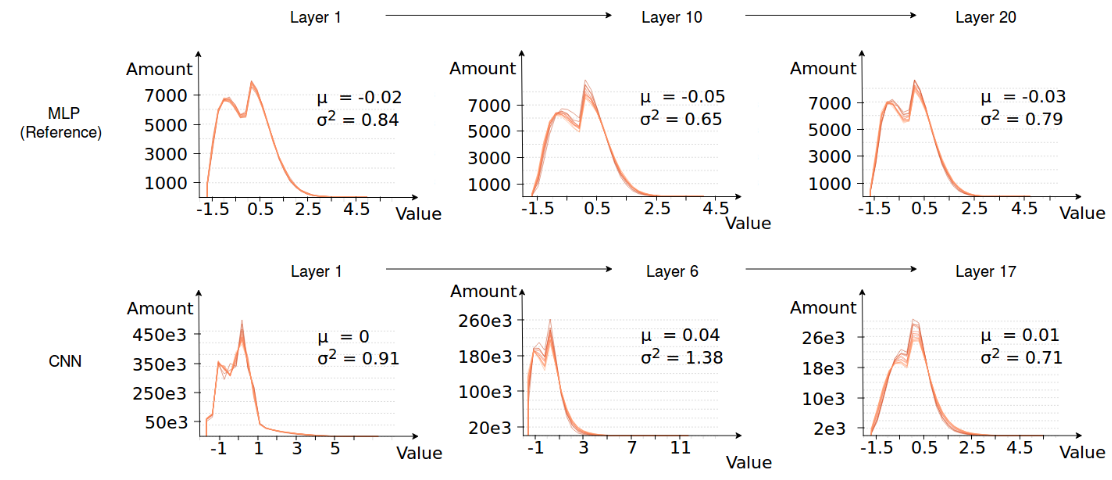

Figure 7.8: Activation distributions of reference settings on MLP and CNN

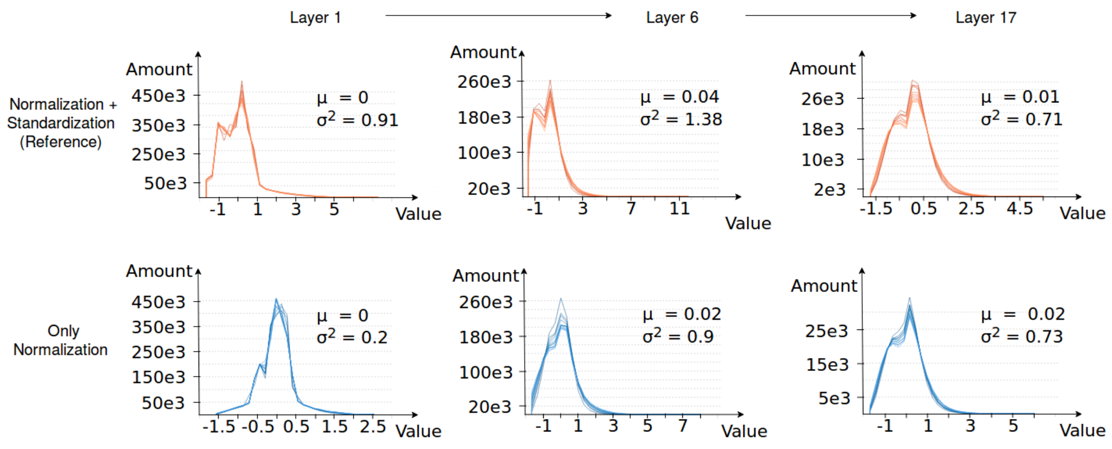

Figure 7.9: CNN - Activation distributions with different preprocessing techniques

handling ill-formated input data.

The second test (figure 7.10) is about changing batch size. This test leads to similar conclusion as the MLP's batch size test. Higher batch sizes produce better activation distributions. In this case, a batch size of 16 is enough for SELU to function properly in a 20-layers-network.

The third test (figure 7.11) is done by changing training optimizers, namely: SGD-Momentum, RMSProp, AdaDelta, and Adam. The result is similar to that of the MLP test. RMSProp and Adam's hidden activations still explode. However, in this case, scalar values' magnitude is smaller than in the MLP optimizer test.

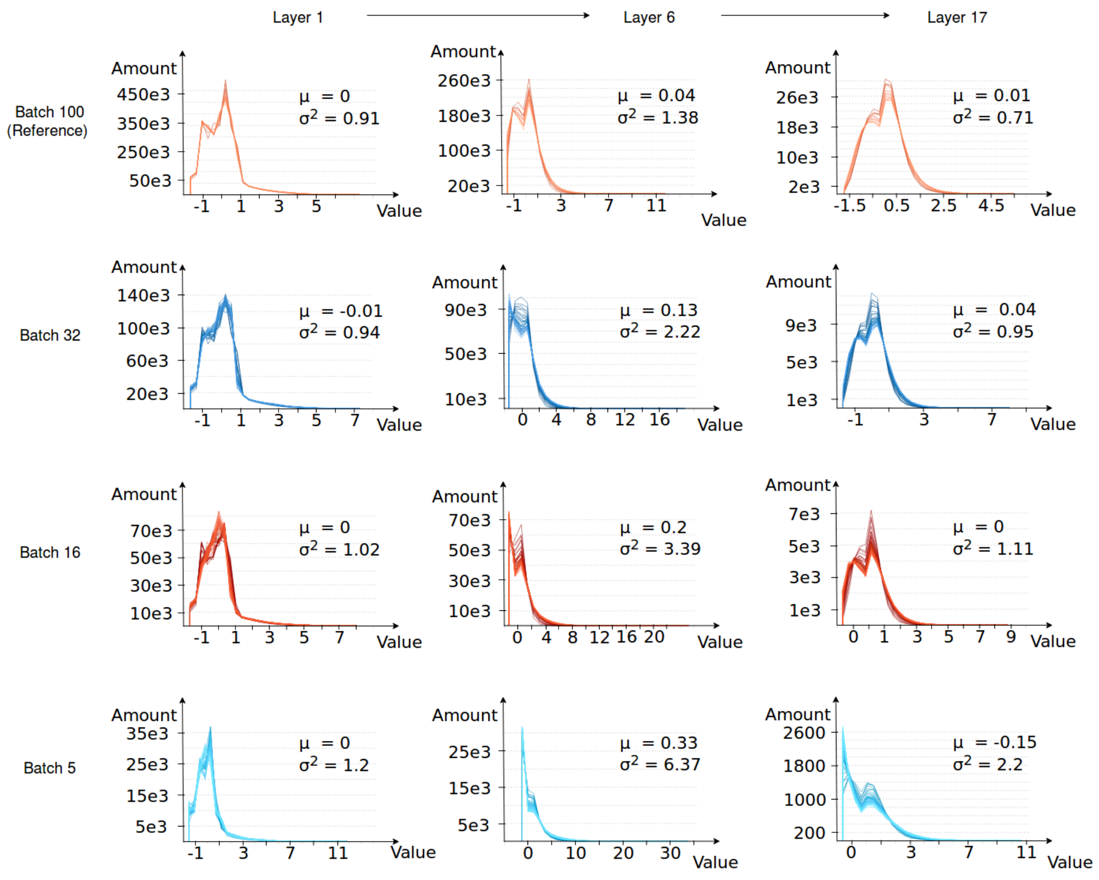

Figure 7.10: CNN - Activation distributions with different batch sizes

AdaDelta is able to hold a relatively controlled mean, variance. However it still is not able to reach an adequate state. The conclusion for this test is that SELU still only functions with SGD-Momentum in CNN.

In summary, these tests confirm that, under correct configurations, SELU still retains its self-normalizing capability in convolutional networks. However, SELU exhibits several problems, which exist for both MLP and CNN. SELU only functions as intended with either SGD or SGD-Momentum optimizer. This is a big downside, because it has been shown that other complex optimizers usually allow faster, more robust training, and simpler hyperparameters choice [48, 43, 44]. Empirical results from section 7.3 further confirm this. Additionally, these tests show another one of SELU's weaknesses – Earlier layers do not output well normalized activations. This weakness can be partly mitigated by using larger batch size, and better input

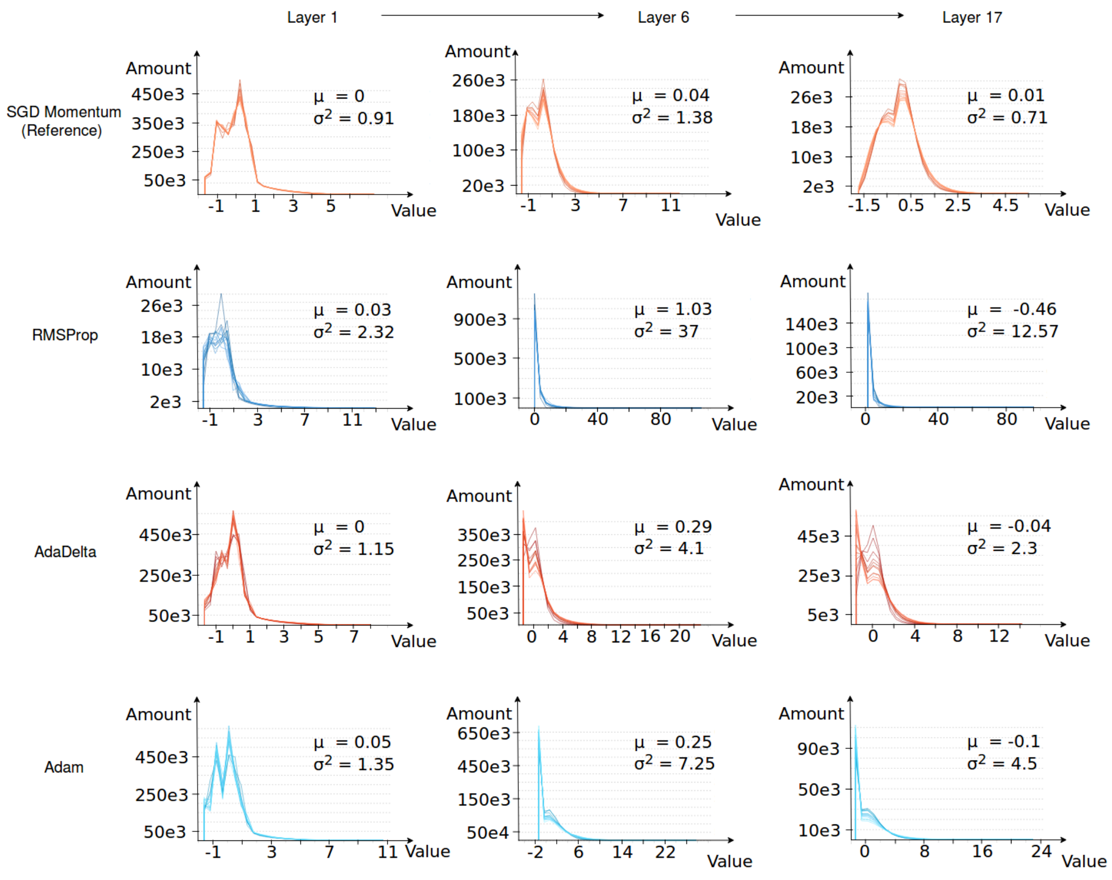

Figure 7.11: CNN - Activation distributions with different optimizers

preprocessing, which are sometimes difficult to fulfill.

7.3 Experiment 3

Table 7.1 shows conclusions of all parameter combinations. For SGD-momentum, no network is trained successfully. BN-based YOLO with batch size 32's failure is caused by hardware limitation. Thus, at this moment, no verdict can be given to this parameter combination. However, this clearly shows that BN has higher hardware requirements in comparison to SELU. Additionally SGD with momentum is not capable of training a complex, highly customized network without careful hyperparameters choices. AdaDelta was able to keep SELU-based networks from diverging. However, without proper normalization, these networks are only able to converge at local minimums. At these local minimums, network's performance is

	SGD - Momentum		**AdaDelta**	
	Batch 8	**Batch 32**	**Batch 8**	**Batch 32**
SELU	Diverged	Diverged	Local minimum	Local minimum
BN	Diverged	Out of Memory	***Trained***	Out of Memory

Table 7.1: Experiment 3 result

mediocre in comparison to a successfully trained network. Nonetheless, this has proven AdaDelta's advantages over SGD-momentum. In conclusion, a complex optimizer with batch normalization is still the most reliable combination for training complex deep neural network. The AdaDelta-trained BN-based network performs relatively well in comparison the original YOLO v3 network. Unless SELU is able to properly work with these complex optimizers, training SELU-based complex networks will still be difficult.

8 Conclusions

This chapter presents contributions, and learned lessons. Furthermore, possible future works are also discussed.

8.1 Contributions

This project presents three main contributions, and one byproduct.

- **Contribution 1**: Providing empirical proofs of SELU's computational advantages over BN. Speed test experiments show that, during inference, SELU-based networks are between 1.5 and 2.8 times faster than BN-based network. Furthermore, SELU-based networks require less physical memory.

- **Contribution 2**: Finding hyperparameter conditions for SELU. From experiments, three conditions are found for SELU to function properly. SELU requires large batch size, and well-conditioned data. Those two are soft conditions, meaning they can be relaxed if the network is deep. The last condition is that the optimizer must be SGD (with or without momentum). This is a strong condition. SELU is currently unable to function properly with any other complex optimizers, namely: RMSProp, AdaDelta, Adam.

- **Contribution 3**: Showing empirical proofs of SELU functioning correctly within CNNs. With several experiments – by comparing mean, variance value, and histogram shape – it can be concluded that SELU is still able to function properly in a convolutional neural network. However, a SELU-based CNN must be deeper than a SELU-based MLP for SELU's normalizing effect to fully manifest. All conditions from contribution 2 also applied to SELU in a CNN.

- **Byproduct**: Creating a complete YOLO v3 network with Tensorflow. For the purpose of testing SELU's capability in a complex, high-performing object detection network, YOLO v3 is recreated from scratch in Python 3 with Tensorflow – the original network is built in C.

8.2 Lessons learned

Several lessons are learned by doing this project:

- Timing tests should be set up carefully, and not be done in a shared cluster.
- Loss and accuracy are not always indicators of whether a technique is functioning correctly.
- New techniques should be tested with simple networks and datasets first.
- New techniques should be tested with multiple optimizers.
- For each test, only one hyperparameter should deviate from reference setting.
- AdaDelta seems to be the most robust optimizer.
- A neural network layer should not contain bias if it is followed by batch normalization.
- Tensorflow's graphs are only suitable for quick debugging, not deep analysis.
- External image processing library (i.e. Pillow) should be used instead of Tensorflow's built-in library for more accurate results.
- Codes should be tested immediately after they are written.

8.3 Future work

In its current state, SELU is not suitable for usage in complex image processing problems. The main reason is SELU's incompatibility with complex optimizers. To deal with this, the first step is finding out the cause of this incompatibility. The next step would be creating SELU-compatible version of those optimizers. Vice versa, SELU could also be modified to support more complex optimizers. To further improve SELU, it is promising to look into relaxing SELU's batch size and data preprocessing requirements. Another potential topic would be finding out about other hyperparameters, or network structures' effects on SELU.

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[2] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017.

[3] Jonathan Hui. What do we learn from single shot object detectors (ssd, yolov3), fpn, and focal loss (retinanet)?, Mar 2018.

[4] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[5] Joseph Redmon and Ali Farhadi. YOLOv3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[6] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?(no, it is not about internal covariate shift). *arXiv preprint arXiv:1805.11604*, 2018.

[7] Binh-Son Hua, Minh-Khoi Tran, and Sai-Kit Yeung. Point-wise convolutional neural network. *CoRR*, abs/1712.05245, 2017.

[8] Timothy J. O'Shea, Tamoghna Roy, and T. Charles Clancy. Over the air deep learning based radio signal classification. *CoRR*, abs/1712.04578, 2017.

[9] J. Zhang and Z. Shi. Deformable deep convolutional generative adversarial network in microwave based hand gesture recognition system. *CoRR*, abs/1711.01968, 2017.

[10] Diego Fioravanti Ylenia Giarratano Isotta Landi Margherita Francescatto Claudio Agostinelli Marco Chierici Manlio De Domenico Cesare Furlanello Giuseppe Jurman, Valerio Maggio. Convolutional neural networks for structured omics: Omicscnn and the omicsconv layer. *CoRR*, abs/1710.05918, 2017.

[11] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 90(2):227–244, 2000.

[12] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[13] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[14] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *CoRR*, abs/1602.07868, 2016.

[15] Dmytro Mishkin, Nikolay Sergievskiy, and Jiri Matas. Systematic evaluation of convolution neural network advances on the imagenet. *Computer Vision and Image Understanding*, 2017.

[16] Yann Le Cun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag, 1998.

[17] Redouane Lguensat, Miao Sun, Ronan Fablet, Evan Mason, Pierre Tandeo, and Ge Chen. Eddynet: A deep neural network for pixel-wise classification of oceanic eddies. *CoRR*, abs/1711.03954, 2017.

[18] Oriol Pujol Vila Carles Roger Riera Molina. Solving internal covariate shift in deep learning with linked neurons. *CoRR*, abs/1712.02609, 2017.

[19] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237. Barcelona, Spain, 2011.

[20] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Artificial Neural Networks–ICANN 2010*, pages 92–101. Springer, 2010.

[21] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[22] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[23] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[25] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[26] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[27] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[28] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.

[29] Joseph Redmon and Ali Farhadi. YOLO9000: Better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[30] Koen EA Van de Sande, Jasper RR Uijlings, Theo Gevers, and Arnold WM Smeulders. Segmentation as selective search for object recognition. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1879–1886. IEEE, 2011.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *CoRR*, abs/1406.4729, 2014.

[32] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[33] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.

[34] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016.

[35] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C. Berg. DSSD : Deconvolutional single shot detector. *CoRR*, abs/1701.06659, 2017.

[36] Patrick Poirson, Phil Ammirato, Cheng-Yang Fu, Wei Liu, Jana Kosecka, and Alexander C. Berg. Fast single shot detection and pose estimation. *CoRR*, abs/1609.05590, 2016.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[38] Guanghan Ning, Zhi Zhang, Chen Huang, Zhihai He, Xiaobo Ren, and Haohong Wang. Spatially supervised recurrent convolutional neural networks for visual object tracking. *CoRR*, abs/1607.05781, 2016.

[39] Saeed Ranjbar Alvar and Ivan V. Bajic. MV-YOLO: motion vector-aided tracking by semantic object detection. *CoRR*, abs/1805.00107, 2018.

[40] Markus Braun, Sebastian Krebs, Fabian Flohr, and Dariu M. Gavrila. The eurocity persons dataset: A novel benchmark for object detection. *CoRR*, abs/1805.07193, 2018.

[41] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back propagating errors. *Nature*, 323:533–536, 10 1986.

[42] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[43] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[44] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[45] Li-Ming Fu. *Neural networks in computer intelligence*. Tata McGraw-Hill Education, 2003.

[46] J Henseler. Back propagation. In *Artificial neural networks*, pages 37–66. Springer, 1995.

[47] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[48] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[49] Imad A Basheer and Maha Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, 43(1):3–31, 2000.

[50] Anand Avati, Kenneth Jung, Stephanie Harman, Lance Downing, Andrew Y. Ng, and Nigam H. Shah. Improving palliative care with deep learning. *CoRR*, abs/1711.06402, 2017.

[51] Joose Rajamäki and Perttu Hämäläinen. An iterative closest points approach to neural generative models. *CoRR*, abs/1711.06562, 2017.

[52] Zhiguang Wang, Chul Gwon, Tim Oates, and Adam Iezzi. Automated cloud provisioning on AWS using deep reinforcement learning. *CoRR*, abs/1709.04305, 2017.

[53] Zhewei Huang, Shuchang Zhou, BoEr Zhuang, and Xinyu Zhou. Learning to run with actor-critic ensemble. *CoRR*, abs/1712.08987, 2017.

[54] Mohammad Malekzadeh, Richard G. Clegg, and Hamed Haddadi. Replacement autoencoder: A privacy-preserving algorithm for sensory data analysis. *CoRR*, abs/1710.06564, 2017.

[55] Thomas Blaschke, Marcus Olivecrona, Ola Engkvist, Jürgen Bajorath, and Hongming Chen. Application of generative autoencoder in de novo molecular design. *CoRR*, abs/1711.07839, 2017.

[56] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017.

[57] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[58] Bottou L. Bengio Y. LeCun, Y. and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[59] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[60] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur,

Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[61] Li-Ming Fu. *Neural networks in computer intelligence.* Tata McGraw-Hill Education, 2003.