Mathematical Exploration of SqueezeNet: CNN Architecture Design Strategies for Efficient Low-Cost Computing

Apurva Patel*, Harsh Benahalkar†, Devika Gumaste‡
Department of Electrical Engineering
Columbia University
New York, USA
Email: {*amp2365, †hb2776, ‡dg3370}@columbia.edu

Abstract

This paper provides a comprehensive quantitative analysis and mathematical exploration of various SqueezeNet architectures, emphasizing the enhancement of computational efficiency. SqueezeNet is renowned for its compact design and suitability for resource-constrained settings such as mobile and edge devices. Through meticulous investigation, we dissect the mathematical frameworks underlying architectural decisions across SqueezeNet variants, including vanilla configurations, those integrating residual connections, and those employing dense-sparse-dense training paradigms. Our inquiry elucidates the intricate trade-offs between model complexity, parameter reduction methodologies, and computational efficiency metrics. Leveraging advanced mathematical optimization techniques, we propose novel architectural design strategies tailored to further boost efficiency while preserving competitive performance benchmarks. Empirical validations corroborate the effectiveness of these strategies, showcasing tangible improvements in efficiency metrics without notable sacrifices in model accuracy. Overall, this study is to be used as a technical guide for future researchers and practitioners engaged in enhancing deep learning architectures for real-world deployment scenarios, particularly where computational efficiency is paramount. By rigorously dissecting the mathematical underpinnings and architectural nuances of SqueezeNet variants, this work contributes to the ongoing discourse on efficient neural network design for diverse deployment contexts.

Index Terms

SqueezeNet, parameter reduction, architecture design, mathematical analysis

I. Introduction

In the expansive domain of deep learning, Convolutional Neural Networks (CNNs) stand as one of the most influential innovations, particularly in tasks concerning image analysis, recognition, and understanding. These networks have been designed, inspired by the intricate organization of neurons in biological organisms, and have revolutionized computer vision by mimicking the hierarchical processing of visual information.

CNNs operate by employing layers of learnable filters or kernels that convolve over input data, progressively extracting and learning hierarchical features. These features become increasingly abstract and complex as they traverse deeper layers of the network, ultimately enabling the network to discern intricate patterns and objects within images.

While CNNs have proven incredibly effective across a spectrum of applications, their widespread adoption on low-compute devices, such as FPGAs, mobile phones, and embedded systems, has posed challenges. These devices often have limited computational power and memory, necessitating the development of architectures that strike a balance between accuracy and efficiency.

SqueezeNet emerges as a pioneering solution to this conundrum. Unlike traditional CNNs, which can be parameter-heavy, SqueezeNet achieves comparable performance while significantly reducing the number of parameters, thus rendering it lightweight and computationally efficient. This efficiency is pivotal for real-time applications and deployment on devices with restricted computational resources.

At the heart of SqueezeNet's efficiency lies its ingenious utilization of 1x1 convolutional filters. These filters, also known as pointwise convolutions, allow for dimensionality reduction by squeezing the input data channels before applying traditional convolutions. This reduction in channel dimensions drastically lowers the number of parameters while preserving crucial features, effectively compressing the model without sacrificing accuracy.

In this paper, we embark on an exhaustive exploration of SqueezeNet architectures, aiming to unravel the mathematical intricacies that underscore their design principles. We delve into various architectural variants of SqueezeNet, ranging from the original design to those enriched with residual connections and those trained using dense-sparse-dense methodologies.

Our objective is multifaceted. We seek to elucidate the delicate trade-offs between model complexity, parameter reduction techniques, and computational efficiency within SqueezeNet variants. Through meticulous analysis and empirical experimentation, we strive to provide insights into the underlying mechanisms driving the efficiency of the SqueezeNet model.

These insights hold profound implications for both researchers and practitioners in the field of deep learning. They offer actionable guidance on how to optimize neural network architectures for real-world deployment scenarios, where computational efficiency is of paramount importance. By demystifying the mathematical underpinnings of SqueezeNet, we aspire to catalyze advancements in efficient neural network design tailored to the diverse demands of modern deployment environments.

II. Previous Related Work

A. Convolution Neural Networks

A special branch of Neural Networks, known as Convolutional Neural Networks (CNNs) has become the backbone of modern machine learning and artificial intelligence, particularly in domains like computer vision. These deep neural networks are adept at automatically learning and extracting intricate patterns and features from visual data, making them usefull tools for image tasks such as classification, detection, and segmentation.

CNNs work on a fundamental concept inspired by the architecture of the neuro-visual cortex in animals. They consist of multiple layers, each comprising of small, trainable filters or kernels that convolve across the input data. Through this convolutional process, the network is trained to capture hierarchical representations of features in the input data, with lower layers detecting simple patterns like edges and textures, and deeper layers detecting more complex structures like objects and shapes.

The hierarchical and localized nature of convolution allows CNNs to efficiently capture spatial dependencies within the input data. Furthermore, CNNs often incorporate additional layers such as pooling layers to downsample feature maps, and fully connected multi-layer perceptrons to make predictions using the features learned.

The key advantage of using a CNN lies in it's ability to learn hierarchical representations automatically from raw data, without the need for synthesized features. This start-to-end learning process enables them to generalize well on new unseen data and achieve state-of-the-art performance metrics across many visual tasks.

Moreover, they exhibit a degree of parameter sharing, meaning that the same set of filters is applied across the entire input data, leading to significant parameter reductions compared to fully connected networks. This efficiency combined with their hierarchical feature learning capabilities, makes them well-suited for even noisy and large-scale datasets.

Convolutional Neural Networks have revolutionized the field of Deep Learning by enabling the automatic extraction of meaningful insights from raw data. Their hierarchical architecture, parameter efficiency, and end-to-end learning capabilities have propelled them to the forefront of machine learning research and applications, paving the way for advancements in various fields, from medical imaging to autonomous vehicles.

Wiener-Khinchin theorem (cross-correlation): In Convolutional Neural Networks (CNNs), convolution operations are often represented as cross-correlations between the input image (or feature map) and the filter (kernel or mask). This cross-correlation operation captures the similarity between the input and the filter at different spatial locations.

Let $x * y$ denote the cross-correlation of functions $x(t)$ and $y(t)$. Then

$$
\begin{aligned}
x * y &= \int_{-\infty}^{\infty} x^*(\tau) y(t+\tau) d\tau \qquad (1)\\
&= \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} X^*(\nu) e^{2\pi i \nu \tau} d\nu \right] \left[\int_{-\infty}^{\infty} Y(\nu') e^{-2\pi i \nu'(t+\tau)} d\nu' \right] d\tau \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} X^*(\nu) Y(\nu') e^{-2\pi i \tau(\nu'-\nu)} e^{-2\pi i \nu' t} d\tau d\nu d\nu' \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} X^*(\nu) Y(\nu') e^{-2\pi i \nu' t} \left[\int_{-\infty}^{\infty} e^{-2\pi i \tau(\nu'-\nu)} d\tau \right] d\nu d\nu' \\
&= \int_{-\infty}^{\infty} X^*(\nu) Y(\nu) e^{-2\pi i \nu t} d\nu \\
&= X\left[X^*(\nu) Y(\nu)\right] \qquad (2)
\end{aligned}
$$

where X denotes the Fourier transform and z^* is the complex conjugate, and

$$
\begin{aligned}
x(t) &= \mathcal{F}_\nu\left[X(\nu)\right](t) = \int_{-\infty}^{\infty} X(\nu) e^{-2\pi i \nu t} d\nu \\
y(t) &= \mathcal{F}_\nu\left[Y(\nu)\right](t) = \int_{-\infty}^{\infty} Y(\nu) e^{-2\pi i \nu t} d\nu
\end{aligned}
$$

Applying the Fourier on each side gives the cross-correlation theorem,

$$
x * y = X\left[X^*(\nu) Y(\nu)\right] \qquad (3)
$$

Now if $X = Y$, then the above cross-correlation theorem is the same as the Wiener-Khinchin theorem.

- **Mathematical Representation:** The cross-correlation operation, denoted as $x * g$, involves sliding the filter $y(t)$ across the input signal $x(t)$ and computing the integral of the product of the values at each overlapping point.
- **Relation to Fourier Transform:** According to the cross-correlation theorem, the cross-correlation operation of the input and the filter in the time domain is equivalent to the multiplication of the Fourier transforms of the input and the filter in the frequency domain.
- **Connection to CNNs:** In CNNs, this theorem improves the computational efficiency of convolution operations. By performing convolution in the frequency domain using the Fourier transforms of the input and filter, convolutional layers can leverage fast Fourier transform (FFT) algorithms for efficient computation.
- **Implementation in CNNs:** CNN frameworks often exploit this relationship by converting convolution operations into frequency domain multiplications, especially when dealing with large filters or images. This approach can lead to significant computational savings, particularly in deep networks with numerous convolutional layers.
- **Generalization:**While the original theorem applies to continuous signals, its principles can be extended to discrete signals and images, which are commonly encountered in CNNs. The Discrete Fourier transform (DFT) and the Fast Fourier transform (FFT) play a crucial role in this context.

So, the cross-correlation theorem provides a theoretical foundation for understanding convolution operations in CNNs and guides the development of efficient algorithms for implementing convolutions, thereby contributing to the computational effectiveness of CNN architectures.

*1) **CNN architecture and working**:* Convolutional Neural Networks are a category of Neural Networks specifically designed for processing structured grid-like data, such as images, video, and audio. Figure 1 shows a typical layout of a CNN implementation. From a mathematical perspective, CNNs comprise several key components:

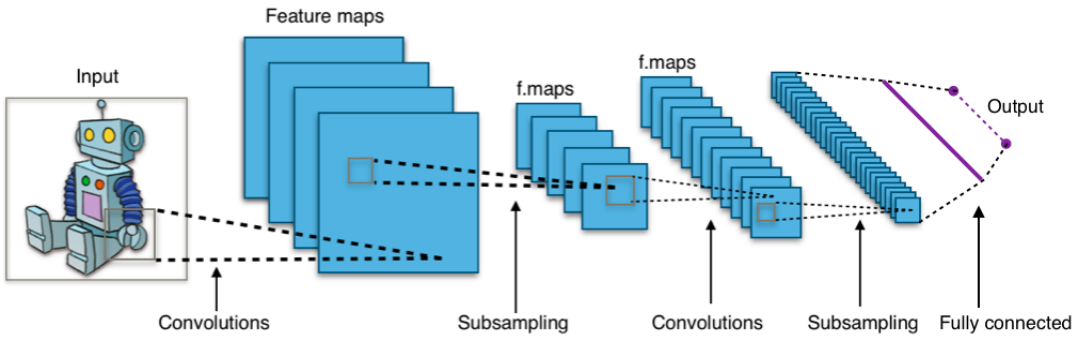


Fig. 1: Typical CNN architecture (figure adapted from Wikipedia [2])

1) **Convolutional Layers**: These layers apply the convolution operations on the input data using learnable kernels. Mathematically, this convolution operation is represented as:

$$(X * Y)(i,j) = \sum_{m=0}^{F-1} \sum_{n=0}^{F-1} X(i+m, j+n) \cdot Y(m,n)$$

where X represents the input, Y represents the filter, and F is the size of the filter.

2) **Activation Functions (ReLU)**: After convolution, an activation function is typically applied element-wise to introduce non-linearity into the network. The Rectified Linear Unit (ReLU) is a commonly used activation function, defined as:

$$\text{ReLU}(x) = \max(0, x)$$

3) **Pooling Layers**: Pooling layers downsample feature maps obtained from prior convolutional layers, thus reducing spatial dimensions while retaining information. The max pooling operation selects the maximum value from each patch of the feature map and is represented as:

$$\text{MaxPooling}(X)(i,j) = \max_{m,n} X(2i+m, 2j+n)$$

4) **Fully Connected Layers**: These layers receive flattened feature maps from convolution layers as input and make predictions on them. The output of a fully connected layer can be represented as:

$$\text{FC}(x) = Wx + b$$

where W represents the weight matrix, x represents the input, and b represents the bias.

These components work in sequence to enable CNNs to automatically learn features directly from input data, making them useful for various machine learning and computer vision tasks.

B. Alexnet variant of CNN

Building on the design principle of CNN, Alexnet was developed. The model, proposed by Alex Krizhevsky et al. 2012 [10], was a significant advancement in the field of Deep Learning and Computer Vision. It was built upon the principles of Convolutional Neural Networks (CNNs) while introducing several key architectural innovations that contributed to its high performance.

It presents a fundamental, straightforward, and efficient Convolutional Neural Network (CNN) architecture. It predominantly comprises sequential stages of 5 convolutional layers, followed by a pooling layer after the first four layers, and finally 3 fully connected layers.

In AlexNet's design, convolutional layers are learned through back-propagation, optimizing the cost function with the stochastic gradient descent algorithm. Convolutional layers process inputs using sliding kernels to produce

convolved outputs, while max pool layers aggregate information within specified neighborhood windows through operations like max pooling or average pooling.

AlexNet uses ReLU activation function and dropout regularization. ReLU, defined by the equation

$$f(x) = \max(x, 0),$$

and it works as a half-wave rectifier. Dropout regularization, a form of stochastic regularization, randomly sets a portion of input or hidden neurons to zero during training, reducing neuron co-adaptations, primarily applied in the fully connected layers to reduce over dependancy on features.

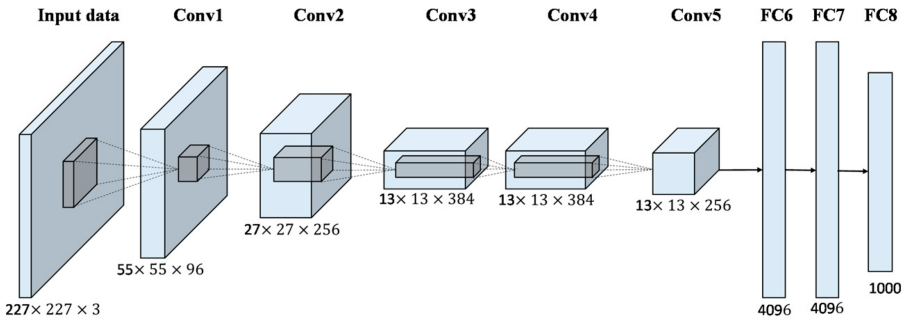


Fig. 2: Alexnet architecture (figure adapted from [5])

1) **Convolution Operation**:
In AlexNet's convolutional layers, the convolution function is applied to the input feature maps using learnable filters (convolutional kernels). Let $I \in \mathbb{R}^{H \times W \times 3}$ represent the input feature map, where H is the height W is the width, respectively, number of input channels is 3 for RGB channels. Let $K_1 \in \mathbb{R}^{11 \times 11 \times 3 \times 96}$ represent the filters in the first layer of convolution. The convolution operation at a specific spatial position (i, j, k) in the first layer can be represented as:

$$(I * K_1)(i, j, k) = \sum_{m=0}^{10} \sum_{n=0}^{10} \sum_{c=0}^{2} I(i + m, j + n, c) \cdot K_1(m, n, c, k)$$

Here, k represents the index of the output channel, and the summation is performed over all elements of the filter and corresponding input feature map region.

2) **Max Pooling Operation**:
After the convolutional layers, max pooling operations are applied to the feature maps. Let $I' \in \mathbb{R}^{H' \times W' \times C'}$ represent the output feature map after convolution, where H' is the height, W' is the width, and C' is the number of channels. The max pooling operation with a pool size of 3×3 and stride of 2 can be represented as:

$$(I')_{\text{pooled}}(i, j, c) = \max_{m,n} I'(2i + m, 2j + n, c)$$

3) **Fully Connected Layer**:
AlexNet includes three fully connected layers with ReLU activation functions. Let $x \in \mathbb{R}^N$ represent the input vector to the fully connected layer, where N is the number of neurons. The output of a fully connected layer with 4096 neurons and ReLU activation can be represented as:

$$\text{Output}(x) = \max(0, Wx + b)$$

Here, W represents the weight matrix of dimensions $4096 \times N$, and b represents the bias vector.

At the end we get just over 62M parameters to train to acheive a decent level accuracy. It is very difficult to deploy Alexnet on the edge and real time computation was limited to hardware capabilities, and this paved the way for researchers to develop a different model architecture and acheive those deficits.

III. SqueezeNet Overview

From the detailed parameter analysis (see I in Appendix A for parameter calculation) of AlexNet, it's evident that the model requires a substantial number, $>$ 62M parameters, which can lead to computational overhead, especially in low compute environments such as FPGAs, mobile devices, or embedded systems.

SqueezeNet, proposed by Iandola et al. in 2016 [7], addresses this challenge by introducing a highly efficient convolutional neural network architecture that significantly reduces the number of parameters without sacrificing performance. The key idea behind SqueezeNet is to replace traditional 3×3 kernels with 1×1 kernels, which reduces parameters significantly while still capturing meaningful information.

The primary objective behind the development of SqueezeNet was to create CNN architectures that could maintain competitive accuracy metrics with significantly less number of parameters. To achieve this, 3 strategies were employed:

1) **Strategy 1:** Replacing 3x3 kernels with 1x1 kernels to reduce the number of parameters while maintaining accuracy.
2) **Strategy 2:** Reducing channels in the input to 3x3 filters through squeeze layers, to further reduce computational complexity.
3) **Strategy 3:** Deferring down-sampling until later stages in the network results in larger activation maps, which can lead to higher classification accuracy.

A. Mathematical intuition / theorems involved

1) **Forward pass through a single layer:** In a multi-layer Deep Convolution Network, the ith activation in layer $l+1$ of the DCN can be expressed as follows:

$$a_{(l+1),i} = \sum_{j=1}^{N} w_{(l+1),i,j} \cdot a_{(l),j} + b_{(l+1),i}$$

where (l) represents the lth layer, N represents the number of additions, $w_{(l+1),i,j}$ represents the weight, and $b_{(l+1),i}$ represents the bias.

2) **Universal Approximation Theorem (UAT):** This theorem states that any feed-forward neural network with at least one single hidden layer containing a finite number of neurons and appropriate activation functions can approximate any continuous function on a compact input domain to arbitrary accuracy. Mathematically, it can be stated as follows:
Given a function $f : \mathbb{R}^n \to \mathbb{R}$ that is continuous on a compact subset S of $\mathbb{R}^n$, and a non-constant, bounded, and continuous activation function $\sigma : \mathbb{R} \to \mathbb{R}$, there exists a feedforward neural network with a single-layer, a sufficient number of neurons, and appropriate weights and bias values such that for any $\epsilon > 0$ and any x in S, the approximation error satisfies the condition:

$$|f(x) - F(x)| < \epsilon$$

where $F(x)$ is the output of the network.

3) **Lipschitz Continuity of Neural Networks:** It has been shown that neural networks, including CNNs, are Lipschitz continuous functions. This means that any small changes in the input result in bounded changes in the network's output. Mathematically, Lipschitz Continuity is expressed as:

$$\|f(x) - f(y)\| \leq L\|x - y\|$$

Here, f is the neural network function, x and y are input vectors, $\|\cdot\|$ is a norm, and L is the Lipschitz constant. This property is important for the stability and robustness of neural networks.

4) **Gradient Descent:** Gradient Descent is an optimization technique used to minimize any cost function $J(\theta)$. It does so by iteratively updating the parameters θ in the direction of the negative gradient of the cost function. The way gradient descent reduces the cost function is given by:

$$\theta := \theta - \alpha \nabla J(\theta)$$

where α is the learning rate, and $\nabla J(\theta)$ is the gradient of the cost function for the parameters θ.

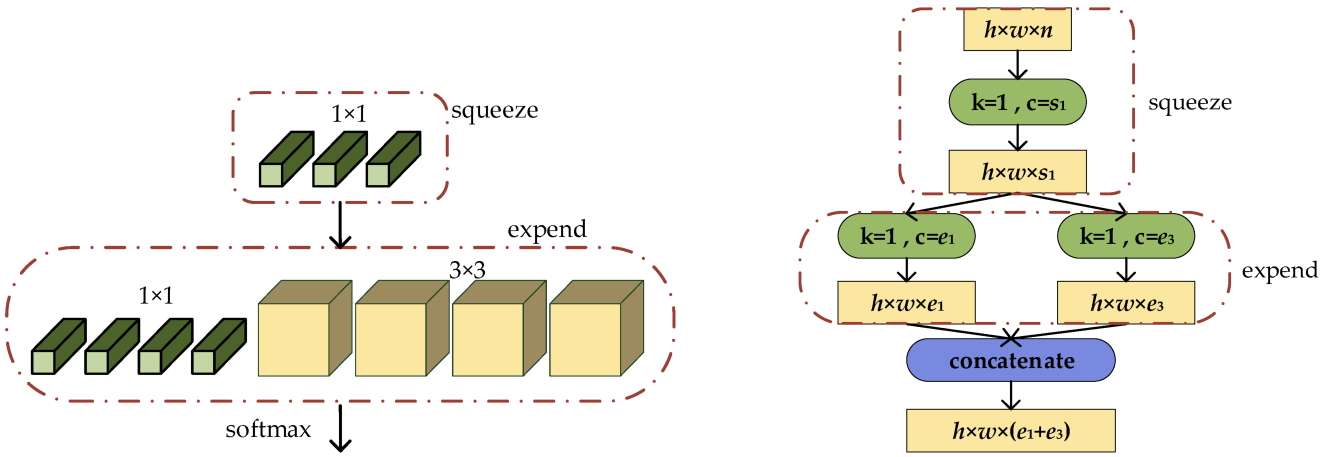
5) **Regularization:** Regularization techniques are used to prevent overfitting in neural networks. The common examples of regularization include L1 and L2 regularization. L2 regularization penalizes large weights by adding a term to the loss function which is proportional to the weight squared, while L1 regularization incorporates sparsity by adding a term which is proportional to the value of the weight.
Mathematically, the regularized loss function $J_{\text{reg}}(\theta)$ is given by:

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda \sum_{i=1}^{n} |\theta_i|^p$$

where $J(\theta)$ is the original loss function, λ is the regularization parameter, n is the number of parameters, and p is the regularization term (usually $p = 1$ for L1 regularization and $p = 2$ for L2 regularization).

B. Fire Modules

The core building blocks of SqueezeNet are Fire Modules. Every module consists of a squeeze layer which are then followed by expand layers. The squeeze layer consists of 1×1 convolutional filters, which aim to compress the input channels (reduce the number of feature maps). The expand layers consist of a mix of convolutional layers - 1×1 and 3×3.



(a) Fire module (b) Fire operation process

Fig. 3: Fire module in Squeezenet v1.0 and it's working (figures adapted from [12])

The squeeze layer decreases the number of input channels (e.g. from $128 \times 32 \times 32$ to $64 \times 32 \times 32$) to the 3×3 filters, while the expand layer helps in learning better representations by incorporating both 1×1 and 3×3 convolution filters, which aim to expand the feature maps in depth. The outputs of these layers are concatenated to enhance expressiveness.

In order to implement the Fire module, the authors utilized concatenation to connect layers with different filter resolutions (e.g., 1×1 and 3×3). This was done by implementing separate convolution layers for each filter resolution and concatenating their outputs.

In a precise way, let's denote the number of input channels as C_{squeeze}, the number of output channels as $C_{\text{squeeze\_out}}$, and the number of output channels from the expand layer as C_{expand}. Then, the total number of parameters in a Fire Module can be calculated as the sum of the parameters in the squeeze layer and the expand layer.

The squeeze layer performs 1×1 convolutions, resulting in $C_{\text{squeeze}} \times C_{\text{squeeze\_out}}$ parameters. The expand layer results in $C_{\text{squeeze\_out}} \times (C_{\text{expand}} + 3 \times C_{\text{expand}})$ parameters.

By carefully selecting the no. of output channels, SqueezeNet ensures that the total no. of parameters in each Fire Module is minimized while preserving representational capacity.

C. SqueezeNet Architecture

SqueezeNet consists of multiple Fire Modules interleaved with max-pooling layers. These Fire Modules are designed to efficiently capture both spatial and temporal features across different scales in the input data. The max-pool layers decreases the spatial dimension of the feature maps(convolution filter), further contributing to parameter reduction.

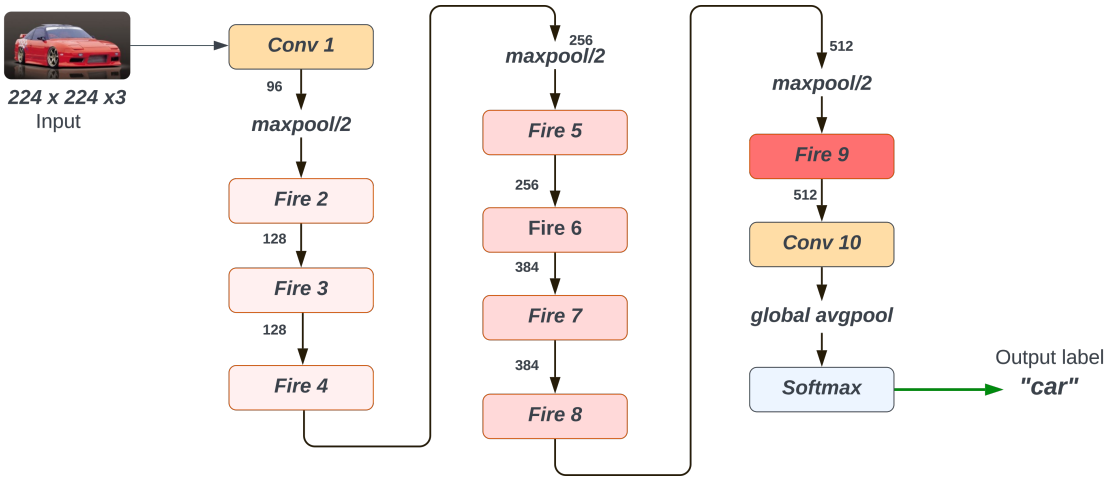


Fig. 4: Squeezenet v1.0 architecture (figure adapted from [1])

1) ***1x1 Convolutions****:* Mathematically, it's the the operation of a 1×1 convolution on an input X having dimensions $H \times W \times C_{\text{in}}$, where H represents height C_{in} represents the no. of channels, and W represents width in the input image. Let Y be the output feature map having dimensions $H \times W \times C_{\text{out}}$, where C_{out} represents the number of output channels in the image.

The output Y is computed as:

$$Y_{i,j,k} = \sum_{c=1}^{C_{\text{in}}} X_{i,j,c} \times W_{1,1,c,k} + b_k$$

Where:

- $X_{i,j,c}$ is the value of the input map in the c-th channel at position (i, j)
- $W_{1,1,c,k}$ is the weight of the convolutional kernel at position $(1, 1)$ in the c-th input channel and k-th output channel.
- b_k is the bias term for the k-th output channel.

By applying 1×1 convolutions, SqueezeNet is reducing the number of trainable parameters in the feature estimating convolutional layers.

2) ***Bottleneck Design****:* The bottleneck design in SqueezeNet involves using convolutions of size 1×1 to decrease the no. of input channels to the subsequent convolutions of size 3×3 in the expand layers. This design choice is motivated by the observation that a large no. of input channels to a convolutional layer can significantly increase the no. of parameters and computational cost without necessarily improving performance.

Mathematically, let's denote the no. of input channels to the bottleneck layer as C_{in}, the no. of output channels from the squeeze layer as C_{squeeze}, and the no. of output channels from the expand layer as C_{expand}. The bottleneck layer consists of convolutions of size 1×1, resulting in $C_{\text{in}} \times C_{\text{squeeze}}$ parameters. The expand layer then uses convolutions of size 1×1 to increase the no. of channels to C_{expand}, followed by 3×3 convolutions.

By reducing the no. of input channels to the 3×3 convolutions in the expand layer, the bottleneck design significantly decreases the number of parameters while still allowing the model to capture complex spatial features.

*3) **Global Average Pooling**:* Mathematically, global average pooling decreases the spatial dimensions of the input features to one value per channel by calculating the average of all values in each channel. Let's denote the input to the global average pooling layer as X with dimensions $H \times W \times C$.

The output Y is computed as:

$$Y_c = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} X_{i,j,c}$$

Where Y_c denoted the c-th channel of the output feature map.

This operation introduces no additional parameters, effectively reducing the overall parameter count in the network.

*4) **Model Compression Techniques**:* Model compression techniques such as network pruning and quantization involve identifying and removing redundant or less important parameters from the network while preserving performance.

- **Network Pruning (implementation below)**: In network pruning, parameters with low magnitudes are pruned or removed from the network. Mathematically, this can be achieved by setting parameters below a certain threshold to zero. Pruning reduces the parameter count and computational cost of the network without significantly affecting performance.
- **Quantization**: Quantization reduces the precision of weights and activations from float to lower bit-width integers or fixed-point numbers. This reduces the memory and cost of the network, making it more efficient for resource-constrained deployment.

These model compression techniques leverage mathematical concepts such as magnitude-based pruning and quantization to eliminate parameters that contribute minimally to the network's output, resulting in further parameter reduction.

Overall, SqueezeNet demonstrates that it is possible to achieve high performance on image classification tasks with a significantly smaller model size compared to traditional architectures like AlexNet. This makes SqueezeNet well-suited for deployment on edge devices where memory and compute resources are limited.

IV. SqueezeNet Design and Implementation

Squeezenet has a lot of variations/implementations considering it's popularity and model complexity. SqueezeNet, proposed by Iandola et al. in 2016 [7] implemented the first squeezenet architecture consisting of fire modules and pretrained on the Imagenet dataset [3].

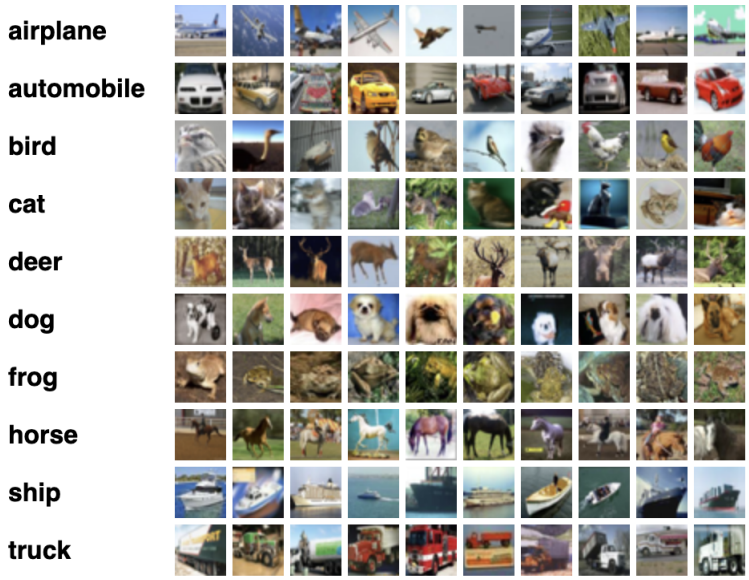


Fig. 5: Samples of all classes from the CIFAR-10 dataset (figure adapted from [9])

A. Vanilla SqueezeNet

SqueezeNet has been developed with efficiency in mind, aiming to achieve high-performance metrics while keeping the model and computation lightweight, so that accuracy is not sacrificed. The extensive use of novel fire modules which combine 1x1 and 3x3 convolutions, enable high accuracy while keeping the computation required in control. The breakdown of the functionality of the fire modules is as follows:

1) **Compression**: The fire module consists of 1x1 convolution layers. These layers first "squeeze" the input, effectively compressing the feature maps.
2) **Expansion**: Following the squeeze operation, the compressed features are passed through an expand operation. 1x1 and 3x3 convolutions are applied parallelly on the input. The purpose of using these layers is that the 1x1 convolution keeps the size constant and the 3x3 convolution captures spatial information.
3) **Concatenation**: The outputs of the 1x1 and 3x3 convolutions are concatenated along the channel dimension so that the output is now enriched spatially and channel-wise.
4) **Activation**: Finally, non-linearity is introduced by passing the output through an activation. the activation used in the fire module is Rectified Linear Unit (ReLU).

SqueezeNet has been implemented in 2 versions, SqueezeNet 1.0 and SqueezeNet 1.1. The major difference between the two versions is in the input layer and the conv layer where SqueezeNet 1.0 uses 96 channels with a kernel size of 7 and SqueezeNet 1.1 used 64 channels with a kernel size of 3.

SqueezeNet was originally trained using the ImageNet dataset leveraging stochastic gradient descent with momentum (AGD). To increase the diversity in the dataset, we applied data augmentation techniques such as random cropping, image resize, and horizontal flipping. However for our experiments, training the SqueezeNet model, using different Optimizers and Activation functions presented with an opportunity to explore the impact of these choices and the dataset, on the model performance. Optimizers selected include Adam (adam), Stochastic Gradient Descent (sgd), Perturbed Gradient Descent (pgd), Stochastic Gradient Descent with restarts (sgdr), and Accelerated Gradient Descent (agd). Activation functions selected for the classifier layers of the model include Rectified Linear Unit (relu), Leaky Rectified Linear Unit (leaky_relu), and Hyperbolic Tangent (tanh) activation functions. The choice of the dataset (CIFAR-10) for this purpose helps in analyzing the performance of all trainable layers in identifying global and local patterns in the image.

The activation functions used for experimenting with the vanilla architecture, and the intuition behind using them:

1) **TanH (Hyperbolic Tangent)**: This activation function zero-centers the input values by squashing them between -1 and 1. This property helps the network in learning symmetrical patterns and the smoothness of the gradient aids in gradient-based optimization techniques. Although it is very beneficial in gradient optimizations, it suffers from the issue of vanishing gradients, especially in deep networks. This activation function should also be used when the output range of the function is appropriate for the application.

$$\text{TanH}(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

2) **ReLU (Rectified Linear Unit)**: ReLU is a linear activation function that simply returns 0 for any negative inputs and the input value as it is for positive inputs. ReLU is extensively used as it is simple and computationally friendly. This also gets rid of the vanishing gradient problem seen with the Hyperbolic Tangent activation function. However they are prone to suffer from the dying ReLU problem, where neurons of the layer become inactive and don't learn if the output is zero consistently.

$$\text{ReLU}(x) = \max(0, x)$$

3) **Leaky ReLU**: Leaky ReLU is used to mitigate the problem of dying ReLU by allowing negative inputs to pass through. This creates the alpha parameter which needs to be optimized for the activation function to optimize properly. Hence this choice of the hyper-parameter alpha can affect network performance.

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

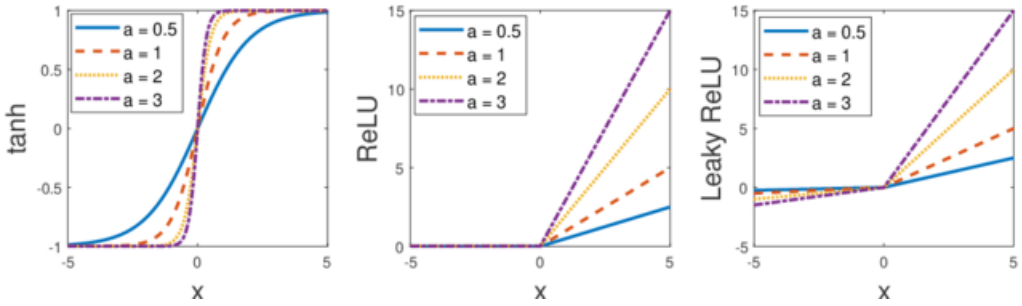


Fig. 6: Figure showing the graph of TanH, ReLU, and Leaky ReLU activation functions(figure adapted from [8])

The Optimizers used for experimenting with the vanilla architecture, and the intuition behind using them:

1) **Stochastic Gradient Descent (SGD)**: Stochastic Gradient Descent updates model weights in the direction of the minima. The factor by which the model weights are updated is called η and it is constant. This is generally used for large datasets, and it is computationally friendly and efficient. However, this optimizer can cause oscillations if there is high variance in the weight updates.

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t; x_i, y_i)$$

2) **Adam (Adaptive Moment Estimation)**: Adam combines the advantages of both AdaGrad and RMSProp. It dynamically adjusts the learning rate for each weight based on estimates of the first and second moments of the gradients. This makes Adam effective for optimizing non-convex objectives and noisy weights. And since Adam adjusts the weights according to their moments, the convergence is faster. However, this advantage comes at the cost of memory. Adam is also sensitive to hyperparameters, which makes it very tricky to use.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \cdot m_t$$

3) **Perturbed Gradient Descent**: Perturbed Gradient Descent deliberately introduces noise by adding a random noise value term to the weight during updates. This exploration of different directions in weights helps the model in escaping the local minima, and thus generalize better. Similar to Adam, this is also sensitive to the hyper-parameter, i.e. the pertubance value. The generation of additional perturbance also comes at the cost of computation and memory.

$$\theta_{t+1} = \theta_t - \eta(\nabla J(\theta_t) + \epsilon_t)$$

4) **Accelerated Gradient Descent**: This uses momentum to accelerate gradient descent updates, by accumulating past gradients to determine the direction and speed of parameter updates. This addition of momentum speeds up convergence and reduces oscillations. however, the momentum parameter makes this model sensitive to the hyperparameters, which when set incorrectly may cause the model to overshoot the minima, leading to instability.

$$v_{t+1} = \beta v_t + (1 - \beta) \nabla J(\theta_t)^2$$

5) **SGD with Restarts**: SGD with Restarts periodically restarts the learning rate schedule during training. This is done to stay away from saddle points and local minima. Resetting the learning rate parameter helps avoid convergence to sub-optimal solutions.

$$\eta_t = \eta_0 \cdot \left(\frac{1}{2}\right)^{\lfloor \frac{t}{T} \rfloor}$$

B. SqueezeNet with Residual connections

SqueezeNet with residual connections enhances the original SqueezeNet architecture by incorporating residual connections between Fire modules. Residual connections allow for easier training of deeper networks by facilitating the flow of gradients through the network and solve the issue of vanishing gradient problem.

The key mathematical concept of residual connections is the Residual Learning Framework, which was introduced in [6].

Residual Learning Framework is to give the underlying mapping as a residual mapping, instead of expecting the stacked layers to capture a desired underlying mapping. Specifically, if the expected underlying mapping is $H(x)$, the residual learning framework lets the layers capture another mapping $F(x) = H(x) + x$, where x is the input to the layer.

Mathematically, this can be expressed as:

$$H(x) = F(x) + x$$

The advantage of this formulation is that it makes the optimization process easier for the network. Instead of having the layers fit the complete mapping $H(x)$, they only need to fit the residual mapping $F(x)$, which is often simpler.

In the context of SqueezeNet, the residual connections are implemented between the "Fire" modules, as shown in the project code. These bypass connections, denoted as bypass_{23}, bypass_{45}, bypass_{67}, and bypass_{89}, enable the network to effectively increase its depth and capacity without significantly increasing the number of parameters, thereby improving the net performance and robustness of the model.

1) Architecture design:

- Each Fire module has a squeeze layer followed by expand layers, as in the original SqueezeNet architecture [7].
- Additionally, residual connections are introduced between Fire modules. These connections directly pass the output of one Fire module to the input of the next Fire module (see figure 7).
- The intuition for residual connections is that they help the network to capture residual functions, i.e., the difference of input and output of a layer, which can be easier to optimize and backpropogate error/loss.

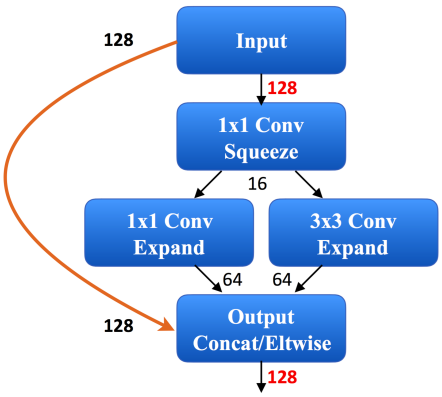


Fig. 7: Fire module with residual connections

Mathematically, let's denote the output of i-th Fire module as y_i. The output of i-th Fire module with residual connection y_i^{res} can be expressed as the sum of outputs of the Fire module y_i and the input to the next Fire module x_{i+1}. This can be represented as:

$$y_i^{res} = y_i + x_{i+1}$$

The intuition goes as follows:

- By adding the residual connection, the network can learn to adjust the input to the next Fire module based on the difference between the expected output and the output of the current Fire module.
- This facilitates the training process, especially in deeper networks, by allowing gradients to flow more easily through the network.

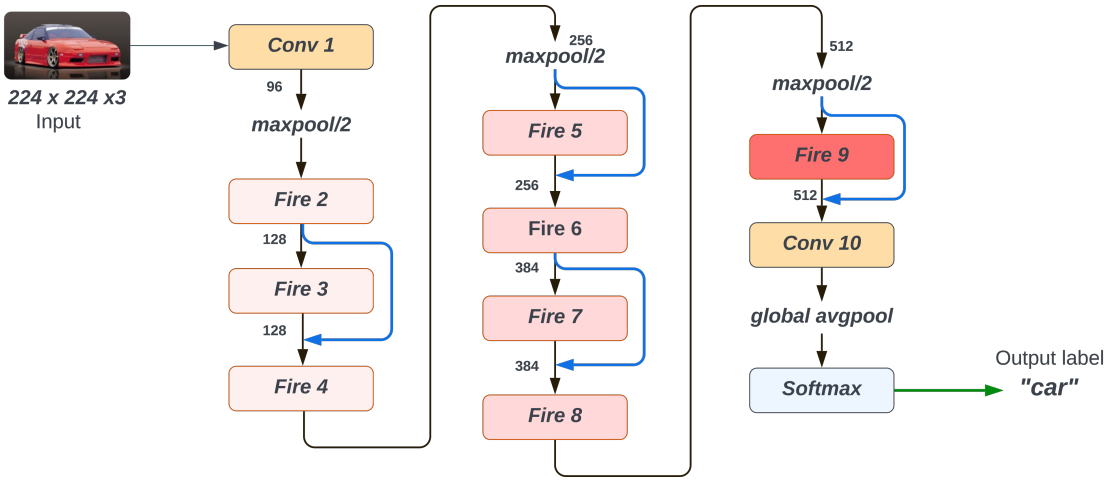


Fig. 8: Squeezenet architecture with residual connections

The loss function used in the SqueezeNet implementation is the widely used Cross-Entropy Loss in classification tasks. The mathematical formulation of the Cross-Entropy Loss is:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log(p_{i,j}) \tag{4}$$

Where, N represents the total samples in the batch, C represents the total number of classes, $y_{i,j}$ represents the label(one-hot encoded) of ground truth for the i-th sample and j-th class, and $p_{i,j}$ represents the estimated probability for the i-th sample and j-th class

Backpropagation Equations

The backpropagation equations for the residual connections in our model can be derived using the chain rule. Let's denote the input to the l-th layer as x_l and the output as y_l. The residual connection is represented as $x_{l+1} = y_l + x_l$.

The partial derivatives with respect to the weights in the l-th layer can be calculated as:

$$\frac{\partial \mathcal{L}}{\partial W_l} = \frac{\partial \mathcal{L}}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{\partial x_l}{\partial W_l}$$

And the gradients w.r.t. the input x_l can be calculated as:

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

Where:

- $\frac{\partial \mathcal{L}}{\partial y_l}$ is the gradient from the upper layers, propagated through the network
- $\frac{\partial y_l}{\partial x_l}$ is the gradient of the activation function in layers
- $\frac{\partial x_l}{\partial W_l}$ is the gradient of the output from layers w.r.t. its weights
- $\frac{\partial x_{l+1}}{\partial x_l}$ is the gradient of the residual connection, which is simply 1 (since $x_{l+1} = y_l + x_l$)

The key advantage of the residual connections is that they provide an alternative path for the gradients to flow during backpropagation (see figure 8 with blue lines as residual connections), which can help solve the diminishing gradient problem and positively change the training of DNNs like SqueezeNet with a significant reduction in model training time.

C. SqueezeNet with DSD traing approach

- Introduction:
  Deep neural networks have become prominent in different fields in today's world. From speech processing, natural language recognition, to vision applications - these networks have proved to be very powerful. The availability of more potent hardware facilitates the training of intricate models with significant capacities. The benefit of large complicated models lies in their ability to expressively capture the nonlinear and intricate relationships between features and outputs. However, the disadvantage of such large models is their vulnerability to capturing background noise rather than the intended patterns in the training data, leading to overfitting and high variance that does not generalize well to new datasets. Conversely, reducing model capacity may lead to underfitting and high bias, causing the machine learning system to overlook relevant feature-output relationships. Balancing bias and variance concurrently poses a challenge in optimization.
  To address this issue, researchers from Stanford, Nvidia, Baidu and Facebook propose a novel training strategy called dense-sparse-dense training flow (DSD). [4] This approach begins with training a dense model conventionally, followed by regularization with optimization using sparsity constraints, and then, increasing model capacity by recovering, re-initialising and retraining the weights which are pruned in the earlier stage. Throughout inference, the resultant model derived from DSD maintains identical architecture and dimensions to the original dense model, without any added overhead.

- Training Flow:
  DSD training methodology is a three step process: dense, sparse, re-dense. Each phase is shown in Figure 5. The distribution of weights throughout the process is shown in Figure 6.
  - **Phase-1:**
    Initial Dense Training - in this method the model learns the weights and the important connections using normal network training. Unlike the conventional training mechanism, here we have a dual objective. We want to not only to learn the values of the weights; but also which connections are important. The initial research employs a straightforward heuristic to measure the significance of weights by assessing their absolute values.
  - **Phase-2:**
    The pruning step - sparse training - involves the removal of low-weight connections and the training of a sparse network. Uniform sparsity is applied across all layers, using a hyperparameter called sparsity percentage which indicates the proportion of weights that are pruned to zero. For each layer, the parameters are sorted, and the k-th highest ($\lambda = S_k$) is selected to be the threshold, where $k = N \times (1 - \text{sparsity})$. A binary mask is then generated to eliminate weights smaller than λ. This pruning of small weights is motivated by the Taylor expansion. To minimize the escalation in loss during hard thresholding on weights, it's essential to reduce the influence of both the first and second terms in Equation (2). As weights are set to zero, ΔW_i becomes simplified to $W_i - 0 = W_i$. At the local minimum where $\frac{\partial \text{Loss}}{\partial W_i} \approx 0$ and $\frac{\partial^2 \text{Loss}}{\partial W_i^2} > 0$, the significance lies primarily with the second-order term. Considering the computational complexity associated with computing second-order gradients and the inherent squared nature of W_i, the absolute value of W_i, denoted as $|W_i|$, serves as the criterion for pruning. A smaller $|W_i|$ indicates a lesser increase in the loss function.

    $$\text{Loss} = f(x, (W_i, i = 1, 2, 3...) \ldots)$$

    $$\Delta\text{Loss} = \frac{\partial \text{Loss}}{\partial W_i}\Delta W_i + \frac{1}{2}\frac{\partial^2 \text{Loss}}{\partial W_i^2}(\Delta W_i)^2 + \ldots$$

    Through retraining while adhering to the binary mask in each iteration, a transformation of a dense network into a sparse one with a predefined sparsity support is achieved, capable of fully recovering or potentially enhancing the original accuracy of the initial dense model under the sparsity constraint. The uniform sparsity across layers can be fine-tuned using validation. Generally, in experiments, a sparsity value between 25% and 50% yields satisfactory results.
  - **Phase-3:**

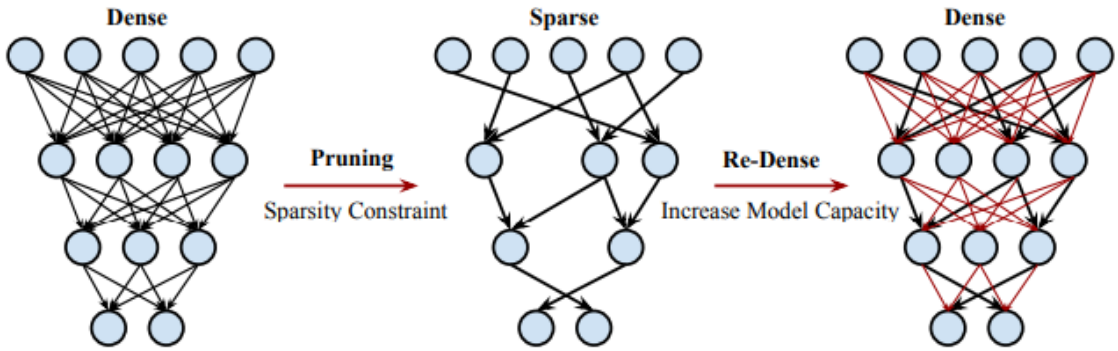


Fig. 9: Training Flow for Dense-Sparse-Dense. Sparse training acts as a regularization method, while the final dense training phase reinstates the pruned weights (indicated in red), thereby enhancing model capacity without risking overfitting. [4]

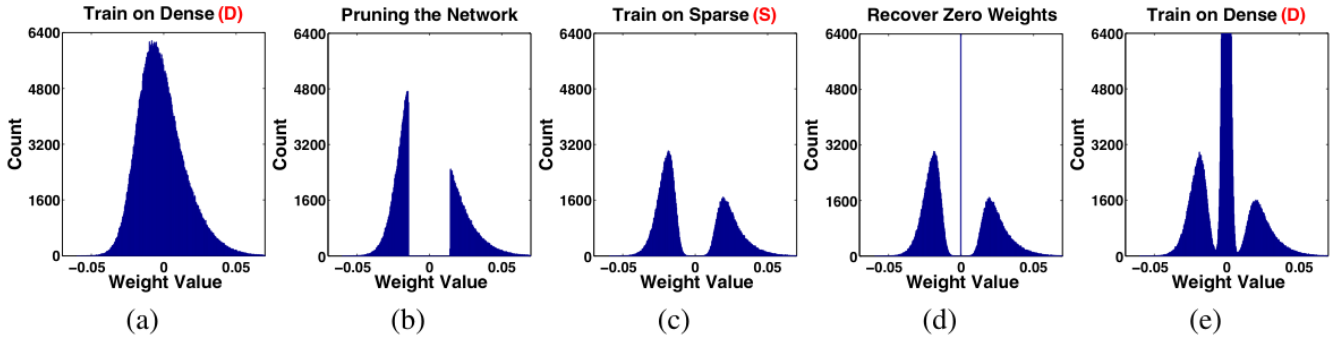


Fig. 10: Weight Distribution throughout the DSD training process. (a) Original Model (b) Pruned Model (c) After sparsity training phase (d) recovering the pruned weights (e) after re-training the entire dense network [4]

The final D step involves the recovery of previously pruned connections, thereby restoring the network to a dense state. These connections, which were previously pruned, are initialized to zero, and the entire network is retrained with a learning rate reduced to 1/10 of the original value (given the sparse network's proximity to a favorable local minimum). Other parameters remain unchanged throughout this process. By reintroducing the zero connections, the final phase increases the capacity of the model, potentially leading to an improved local minimum compared to the constrained model resulting from the phase-2.

To illustrate the training flow of DSD, we examined the evolution of weight distribution, as shown in Figure 10. Initially, weight distribution is distributed around zero. Pruning results in the removal of a significant portion of the central region. During the subsequent retraining phase, the unpruned network parameters readjust, leading to a softened boundary and the emergence of a bimodal distribution, as illustrated in (c). As the re-dense training phase commences in (d), all previously cut-off weights are reinstated and initialized to zero. Lastly, in (e), the pruned and unpruned weights undergo retraining, using consistent learning hyperparameters (e.g., weight decay, learning rate). A comparison between Figures (d) and (e) reveals that while the distribution of unpruned weights remains largely unchanged, the distribution of pruned weights extends further around zero, resulting in a reduction in the mean absolute value. This aligns with the hypothesis that selecting the smallest vector to solve the machine learning problem restrains irrelevant components of the weight vector.

- Related Work:
  While **Dropout** introduces random sparse patterns at each stochastic gradient descent iteration, DSD adheres

to a sparsity determined by data-driven methods' pattern consistently. **Model distillation** offers an alternative avenue for improving neural network performance without architectural modifications. By transferring knowledge from a larger model to a smaller, more deployable one, model distillation facilitates performance enhancements.DSD training and **model compression** share the common strategy of network pruning. However, while model compression primarily focuses on maintaining accuracy, DSD training advances further by significantly enhancing accuracy levels. Notably, DSD training achieves this without necessitating aggressive regularization; instead, a moderately pruned network can deliver robust performance. Conversely, model compression often requires aggressive pruning to achieve high compression rates.The theoretical underpinnings of **sparsity regularization and hard thresholding** have been extensively explored, particularly in the context of learning statistical models in high-dimensional spaces. Additionally, similar training strategies involving iterative hard thresholding and connection restoration have been proposed independently. The application of sparsity regularized optimization, notably in Compressed Sensing, further underscores its relevance in finding optimal solutions to inverse problems within highly underdetermined systems, leveraging the sparsity assumption.

V. Results and Experiments

A. *Vanilla SqueezeNet*

We experimented with 3 activation functions and 5 optimizers, totalling to 15 experiments. The model architecture was designed in PyTorch, and the re-trained weights were loaded into the model. Since we had to replace the activation function in the classification layer and train it on our own task, the final layers of the model were unfrozen to enable them to train. The CIFAR-10 dataset with 10 classes was used for training and validation of the models. The dataset for each experiment was augmented using random resize crops and normalizations. Every experiment was run for 10 epochs, with a CosineAnnealingLR applied on the optimizer. Metrics recorded include train & validation accuracy, train & validation loss, and computation time per epoch. Apart from the aforementioned metrics, we wanted to see how the weights in the final layer are adjusted, so they were also recorded and visualized.

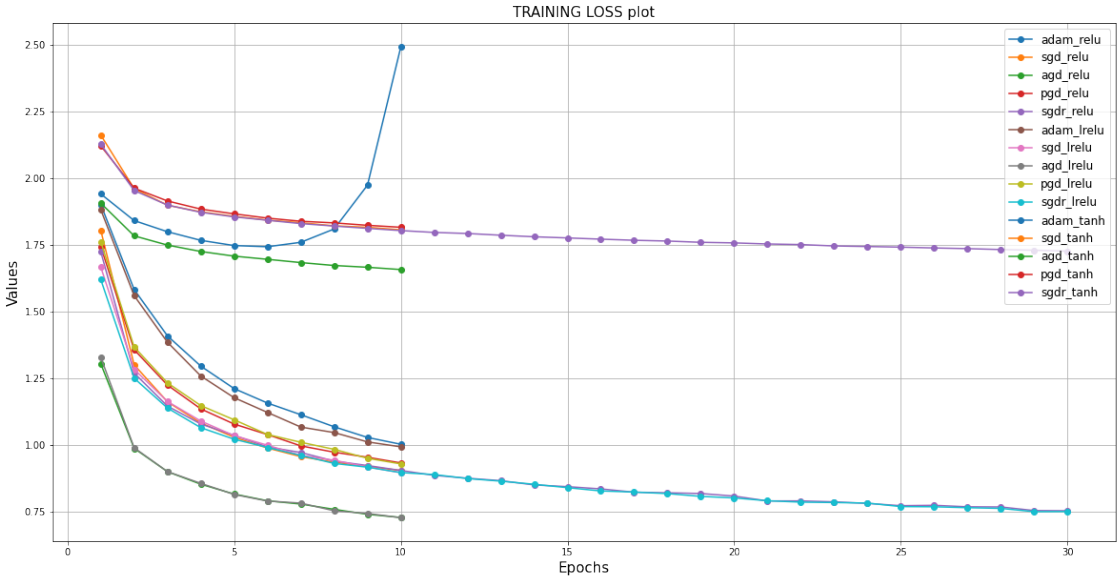


Fig. 11: Training loss values seen for all 15 experiments

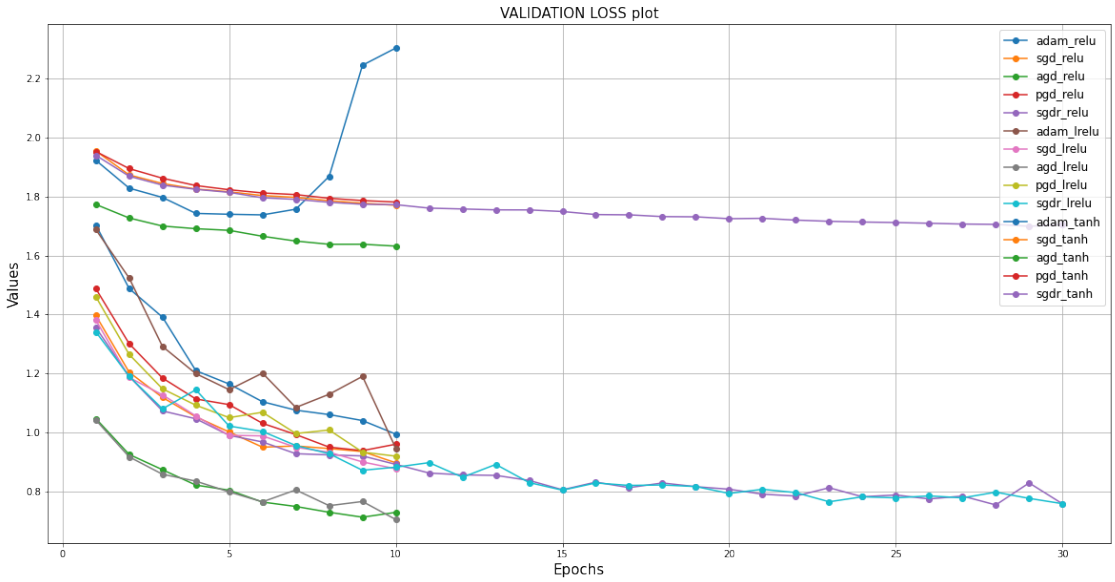


Fig. 12: Validation loss values seen for all 15 experiments

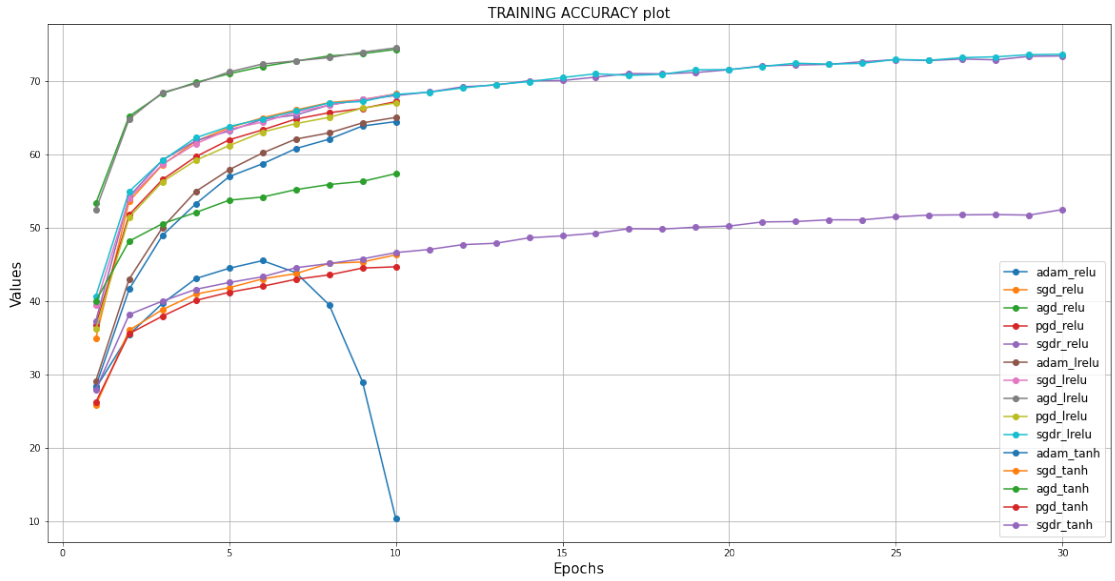


Fig. 13: Training accuracy values (%) seen for all 15 experiments

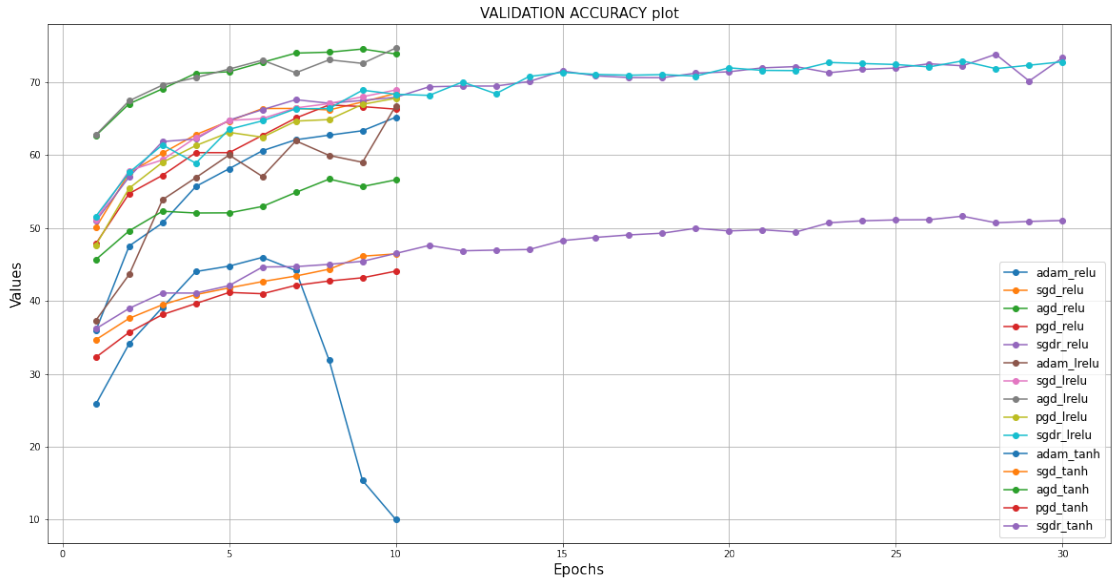


Fig. 14: Validation accuracy values (%) seen for all 15 experiments

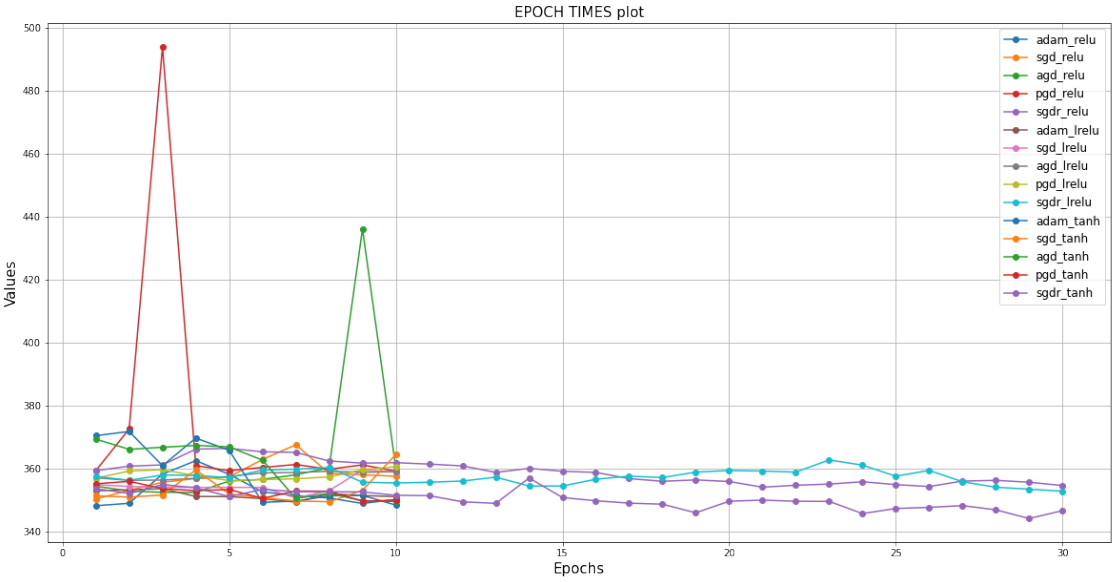


Fig. 15: Epoch times (seconds) for each cycle of training

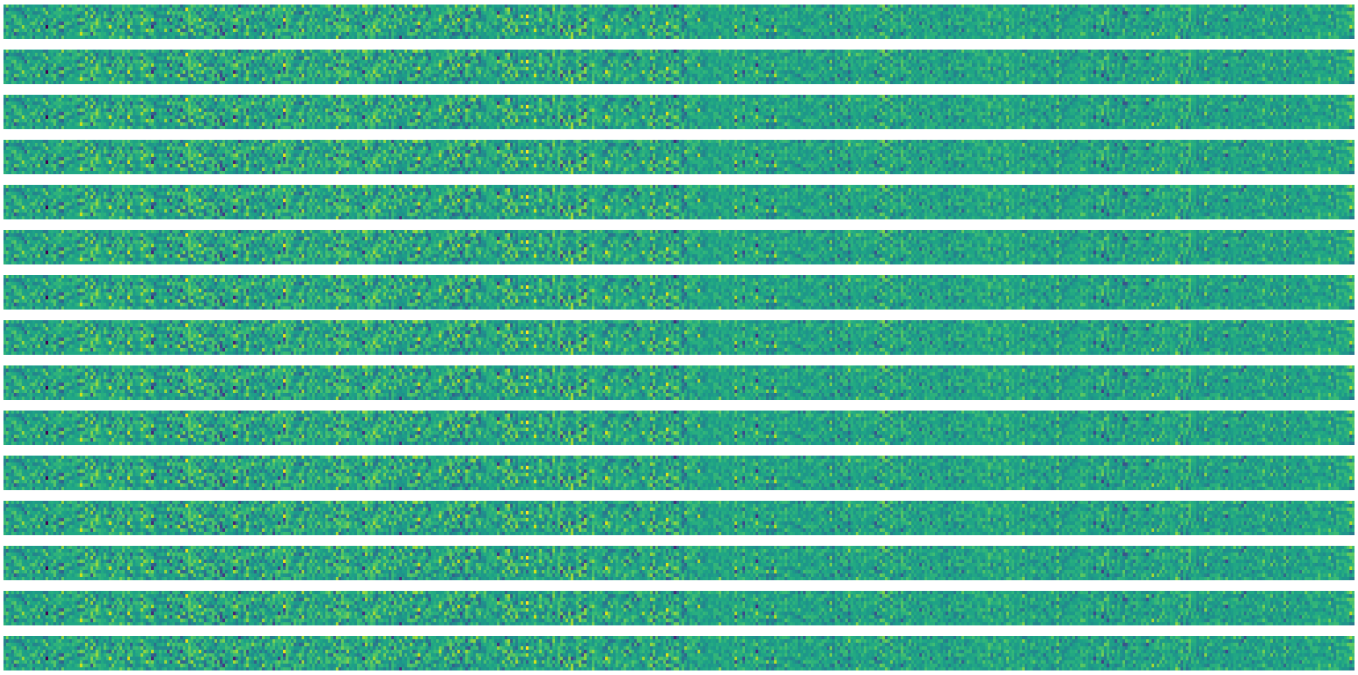


Fig. 16: Last layer classifier weights for all 15 experiments

From the results above, it is evident that out of the five optimizers used Stochastic Gradient Descent has performed well as compared to other optimizers, when all other parameters are kept consistent. This observation is a slight deviation from the norm where the Adam optimizer is expected to generalize better than most of the optimizers. This can be concluded when it is seen that for most of the experiments there is no overfitting observed, i.e. the training and validation losses are consistently decreasing while the accuracies are consistently increasing.

Also seen from the classifier weights of the last layer, there are no descernable differences in the weight values and all weights are similar. This can be further compounded by observing the loss and accuracy patterns of all models.

Thus we can say that SqueezeNet trains and generalizes very well on the dataset, and the performance observed even with the worst optimizers and activation functions is acceptable for a lot of situations. Also considering the model's RAM footprint and number of parameters, it can be said that this model is suited for a lot of applications that may be deployed on edge and embedded devices, since it requires a smaller memory footprint when compared to other models, while keeping the sacrificing on model accuracy.

B. SqueezeNet with Residual connections

This residual connection helps to mitigate the diminishing gradient issue that might occur in DNNs. By providing a direct path for the gradients to flow from the final to the previous layers, the residual connections facilitate better gradient propagation during training.

Additionally, the residual connections allow the network to reuse and combine features from different layers, which can lead to more expressive and informative representations. This feature reuse can be particularly beneficial for small-scale models like SqueezeNet, where the network capacity is limited compared to larger models

1) ***Solving vanishing gradients problem in SqueezeNet****:* In our deep neural network with L (here 10) layers, where the input to the l-th layer is x_l and the output is y_l. In a traditional feedforward network, the mapping can be represented as:

$$y_l = F_l(x_l)$$

where F_l is the transformation performed by the l-th layer.

The diminishing gradient issue arises when the gradients of the loss function $\mathcal{L}$ with respect to the input x_l become exponentially small as l decreases. This can be expressed mathematically as:

$$\frac{\partial \mathcal{L}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial y_L} \frac{\partial y_L}{\partial y_{L-1}} \cdots \frac{\partial y_{l+1}}{\partial y_l} \frac{\partial y_l}{\partial x_l}$$

As the total layer count L increases, the product of the partial derivatives will become exponentially small, leading to the diminishing gradient problem.

In our case of a residual connection in squeezenet as implemented in figure 8, where the mapping is represented as:

$$y_l = F_l(x_l) + x_l$$

This is the key idea behind residual connections, where the network learns the residual mapping $F_l(x_l)$ instead of the complete mapping y_l.

The partial derivative of the loss function w.r.t the input x_l can be given as:

$$\frac{\partial \mathcal{L}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial y_L} \frac{\partial y_L}{\partial y_{L-1}} \cdots \frac{\partial y_{l+1}}{\partial y_l} \frac{\partial y_l}{\partial F_l(x_l)} \frac{\partial F_l(x_l)}{\partial x_l} + \frac{\partial \mathcal{L}}{\partial y_{l+1}} \frac{\partial y_{l+1}}{\partial x_l}$$

The key difference is the addition of the second term, $\frac{\partial \mathcal{L}}{\partial y_{l+1}} \frac{\partial y_{l+1}}{\partial x_l}$, which represents the gradient flowing through the residual connection, and the residual connection ensures that the gradients can be shown as a sum of two terms, where one term is not subject to the exponential decay caused by the product of derivatives.

This residual connection provides an alternative path for the gradients to flow, bypassing the potentially vanishing gradients through the layers. As a result, the gradients can be maintained and effectively propagated to the earlier layers, mitigating the vanishing gradient problem.

*2) **Results for default configuration**:* The default configuration of squeezenet as mentioned in [7] is used to train a residual squeezenet architecture on CIFAR-10 data [9] from scratch. Although the output obtained in figure 17 are nowhere close to the original paper which used Imagenet dataset [3] to pretrain the model, but it's important to note that how fast the model learns with fewer parameters (precisely 1.24M parameters (see II in Appendix B for parameter calculation in residual squeezeent) evern when started without any weight initialization.

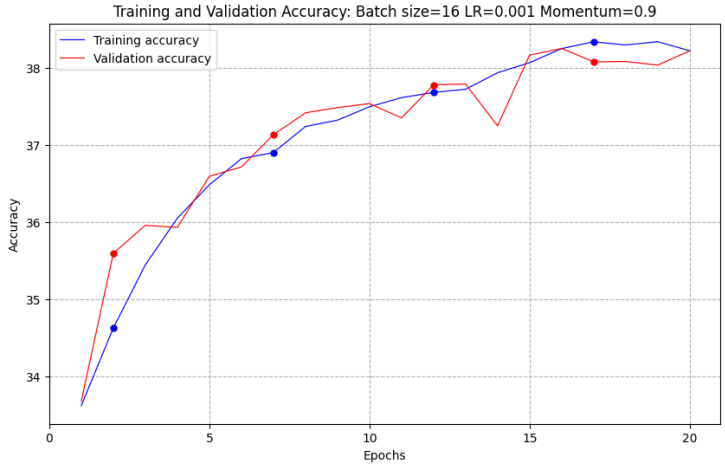


Fig. 17: Default configuration (Batch size: 32, Optimizer: SGD with 0.9 momentum & learning rate α: 0.001)

*3) **Experiments with different Batch size and Optimizers**:* Apart from the aforementioned configuration we experimented with 2 different batch sizes and 3 different optimizers. The choices of batch sizes were made by taking into consideration the GPU capabilities, estimated epoch time, memory and dataset limitations. The choices of batch sizes, 64 and 128, were made considering the GPU capabilities, estimated epoch time, memory, and dataset limitations. With the NVIDIA V100 GPU and 64GB memory, both batch sizes can be accommodated. A reduced batch size like 64 promotes quick updation to the model parameters but may cause noisy gradients. On the contrary, a bigger batch size like 128 gives stability while updating but requires more memory.

***Optimizers**:*

1) **SGD with Nesterov Momentum**: This optimizer includes momentum to accelerate convergence. It helps SGD to converge faster and navigate through plateaus and local minima more efficiently.

$$\begin{aligned} &\text{Momentum term:} && v_{t+1} = \mu \cdot v_t - \alpha \cdot \nabla f(x_t + \mu \cdot v_t) \\ &\text{Update rule:} && x_{t+1} = x_t + v_{t+1} \end{aligned} \tag{1}$$

2) **Adam (Adaptive Moment Estimation)**: Adam is an adapting learning rate optimization algorithm that calculates unique adaptive LR for each parameters. It adapts the learning rate for each parameter, allowing for a speedy convergence towards a minima and better performance on a wide range of deep learning tasks.

$$\begin{aligned} &\text{Moment estimates:} && \mu_t = \beta_1 \cdot \mu_{t-1} + (1 - \beta_1) \cdot g_t \\ & && \nu_t = \beta_2 \cdot \nu_{t-1} + (1 - \beta_2) \cdot g_t^2 \\ &\text{Bias correction:} && \hat{\mu}_t = \frac{\mu_t}{1 - \beta_1^t} \\ & && \hat{\nu}_t = \frac{\nu_t}{1 - \beta_2^t} \\ &\text{Update rule:} && \theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{\mu}_t}{\sqrt{\hat{\nu}_t} + \epsilon} \end{aligned} \tag{2}$$

3) **RMSProp (Root Mean Square Propagation)**: RMSProp is an adapting learning rate optimization method that changes the learning rate by a factor of an exponentially decaying mean of squared gradients. It helps to adjust the learning rates dynamically based on the gradients, allowing for faster convergence and improved performance on non-stationary objectives.

$$\begin{aligned} &\text{Exponentially decaying average:} && \nu_t = \beta \cdot \nu_{t-1} + (1 - \beta) \cdot (g_t)^2 \\ &\text{Update rule:} && \theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\nu_t} + \epsilon} \cdot g_t \end{aligned} \tag{3}$$

Here, α represents the learning rate, μ is the momentum parameter, β_1 and β_2 are the exponential decay rates for the moment estimates, ϵ is a small constant to prevent division by zero, g_t is the gradient at time step t, v_t represents the moving average of squared gradients, and θ_t represents the parameters at time step t.

***Mathematical Reasoning for Adam Optimizer**:* Adam optimizer combines the advantages of AdaGrad and RMSProp. It calculates adaptive learning rates for each parameter by considering both the first and second moments of the gradients.

The mathematical explanation for Adam optimizer is given below:

$$\mu_t = \beta_1 \cdot \mu_{t-1} + (1 - \beta_1) \cdot g_t$$

$$\nu_t = \beta_2 \cdot \nu_{t-1} + (1 - \beta_2) \cdot g_t^2$$

$$\hat{\mu}_t = \frac{\mu_t}{1 - \beta_1^t}$$

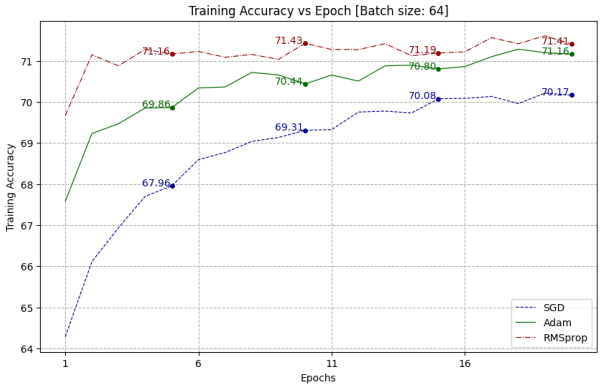
$$\hat{\nu}_t = \frac{\nu_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{\mu}_t}{\sqrt{\hat{\nu}_t} + \epsilon}$$

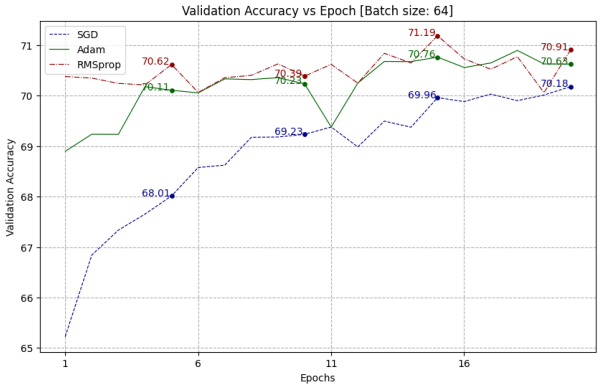
Here, μ_t and ν_t are estimates of the first and second moments of the gradients, β_1 and β_2 are the exponentially decaying rates for the moment estimate, α is the LR, ϵ is a negligible constant to add numerical stability, and θ_t represents the parameters at time step t.

The adaptive learning rate computed by Adam optimizer allows for efficient updates of the model parameters, leading to faster convergence and improved generalization performance.

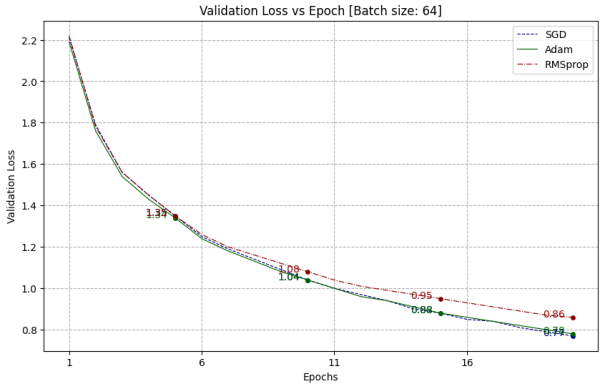
For Batch size: 64*:* The train and validation accuracy graphs for every optimizer (refer to Figures 18a and 18b) reveal the fluctuations and inflection points across epochs. Additionally, Figure 18c illustrates the validation loss trends for a batch size of 64 across different optimizers. Notably, it's intriguing to note that similar loss values across optimizers don't necessarily imply identical validation or test accuracy. This disparity stems from the loss function's dependence on the optimizer output rather than the predicted label.



(a) Training accuracy vs. Epoch



(b) Validation accuracy vs. Epoch



(c) Validation Loss vs. Epoch

Fig. 18: Performance characteristics for batch size of 64

Time per epoch comparison:
The main advantage of residual connection is the reduced time per epoch to train the model. This is achieved by backpropogating the error via the residual connection and hence reducing the time to compute the chain rule otherwise which is encountered during vanilla backpropogation algorithm.
Figure 19 shows the time per epoch for a batch size of 64. It is important to note that the implementation is achieved in PyTorch which uses the parallel computing power of GPU and hence the sequence of instruction execution might determine the time for epoch for each optimizer. To better understand the time we can look at the average time for the entire experiment period; which implied that the average time per epoch follows this order: SGD $>$ RMSProp $>$ ADAM. This can be concluded by having a look at the update rule for each optimizers in equations 1, 2, 3; which outlines the total computations necessary for each optimizer and the difference with global minima. Essentially its a linear combination of no of computations and the "distance" from global minima for that optimization landscape.

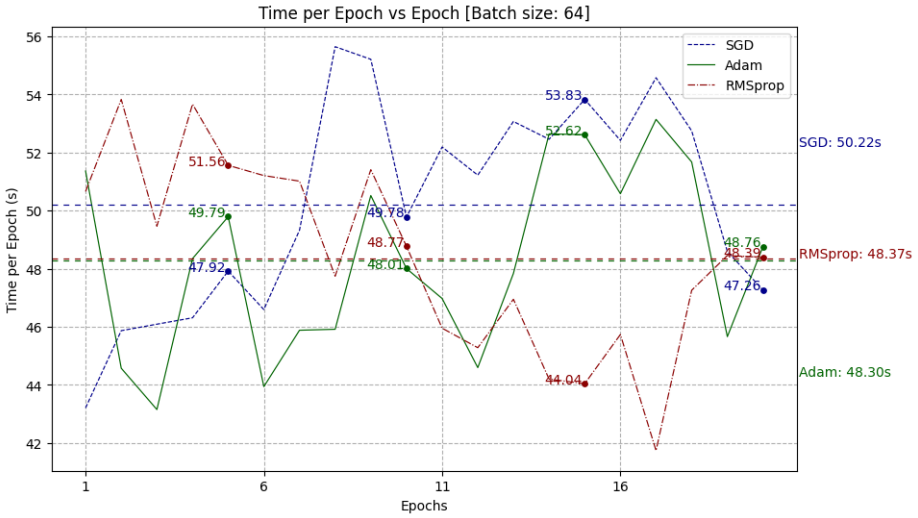


Fig. 19: Time per epoch(s) vs. Epoch for batch size of 64

Comparative study of LR Scheduler for Batch size: 128 *:* We experimented with *"ReduceLROnPlateau"* Learning rate scheduler. To illustrate the functionality of the `ReduceLROnPlateau` scheduler using mathematics, let's denote the validation loss at epoch t as val_loss_t, the lr at epoch t as lr_t, the patience as P, and the multiplier by which the lr is reduced as factor.

The scheduler observes the validation loss over consecutive epochs. If the validation loss does not reduce for a predefined period of epochs (patience value), it updates the learning rate by a factor. The process to update the learning rate can be expressed as:

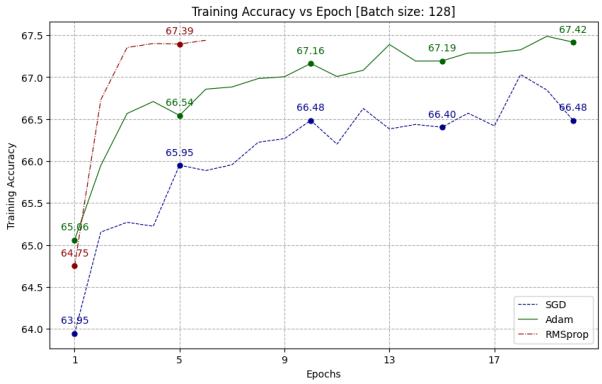
$$\text{new\_lr}_{t+1} = \begin{cases} \text{old\_lr}_t \times \text{factor} & \text{if } \text{val\_loss}_{t-P} \geq \text{val\_loss}_{t-P+1} \geq \ldots \geq \text{val\_loss}_{t-1} \geq \text{val\_loss}_t \\ \text{old\_lr}_t & \text{otherwise} \end{cases}$$

where:

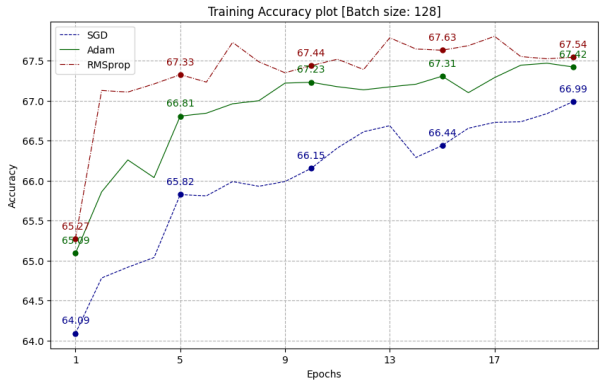
- new_lr_{t+1} is the updated LR for epoch $t+1$.
- old_lr_t is the LR at epoch t.
- val_loss_{t-P} is the validation loss at epoch $t-P$.
- P represents the patience, i.e., the number of epochs to pause before updating the learning rate.
- factor is the multiplier by which the LR is updated.

It shows that the learning rate parameter is reduced by the factor factor only if the validation loss does not improve (i.e., does not decrease) over the specified number of epochs (patience). Otherwise, the learning rate remains unchanged.

Figures 20a & 20b shows the difference in training accuracy when a higher batch size is given with and without a learning rate scheduler. In our experimentation we noticed a strange yet mathematically explainable event of RMSProp model training crashing despite numerous attempts. This can be comprehended by large already stored values in the memory and also the high batch size which fills up the memory and not allowing space allocation for incoming computation sequence and hence leading to a system crash.



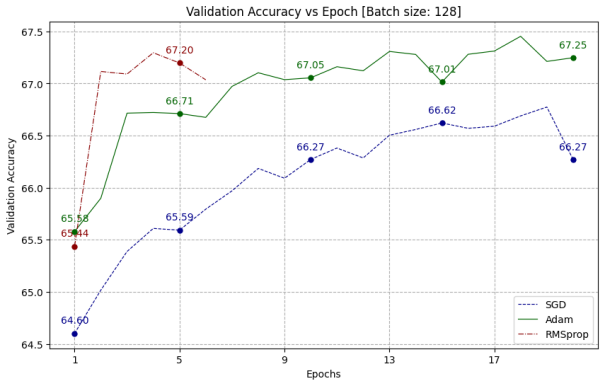
(a) Without Learning Rate (lr) scheduler



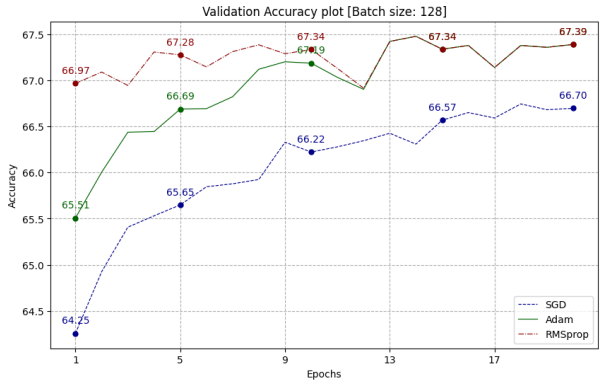
(b) With LR scheduler

Fig. 20: Training Accuracy comparison with 20 epochs

Validation accuracy is almost similar for all optimizers but we see a slight bump in values when a learning rate scheduler is employed. Figures 21a & 21b shows the validation accuracy metric plot for the model.



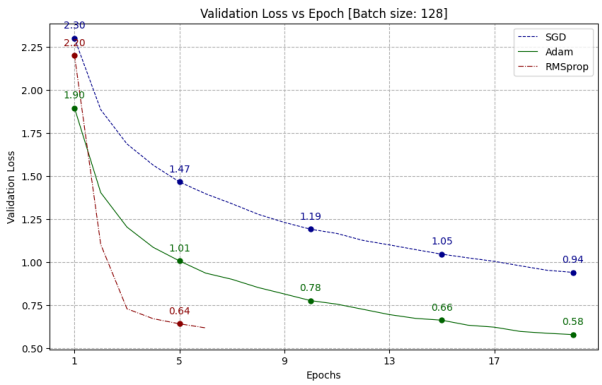
(a) Without Learning Rate (lr) scheduler



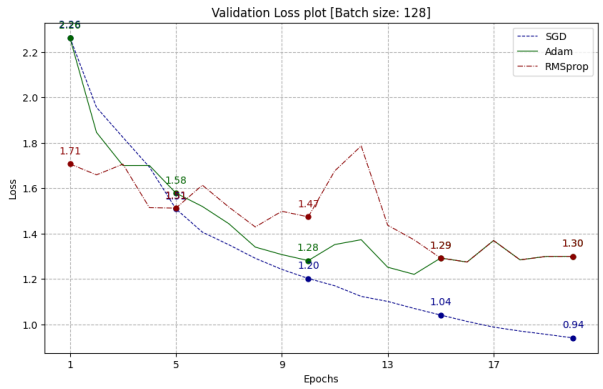
(b) With LR scheduler

Fig. 21: validation Accuracy comparison with 20 epochs

Through our experiments we found out that when a learning rate scheduler is employed we see a better overall convergence and this is because of the update steps in the optimizer governed by the scheduler's output.



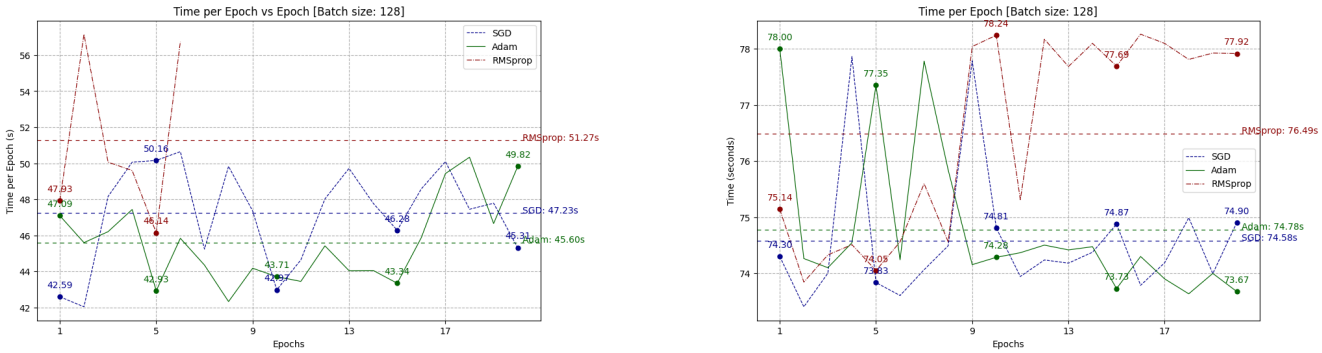
(a) Without Learning Rate (lr) scheduler



(b) With Learning Rate (lr) scheduler

Fig. 22: Validation loss comparison with 20 epochs

The main difference in performance is observed in training time per epoch. This additional time is accounted by the numerous extra computation steps required to find the optimal learning rate each time we start an epoch. Also those computations don't support GPU implementation and hence a CPU is used to perform that which increases its time for each epoch by a significant margin.



(a) Without Learning Rate (lr) scheduler

(b) With Learning Rate (lr) scheduler

Fig. 23: Epoch time comparison with 20 epochs

Weights from convolution layer: After experimenting on each computationally and practically possible configuration we get the weights in convolution layers and other modules. These weights are 7×7 filters and we have displayed weights from *Conv1* layer in figure 24

This layer is just one of the many convolutional

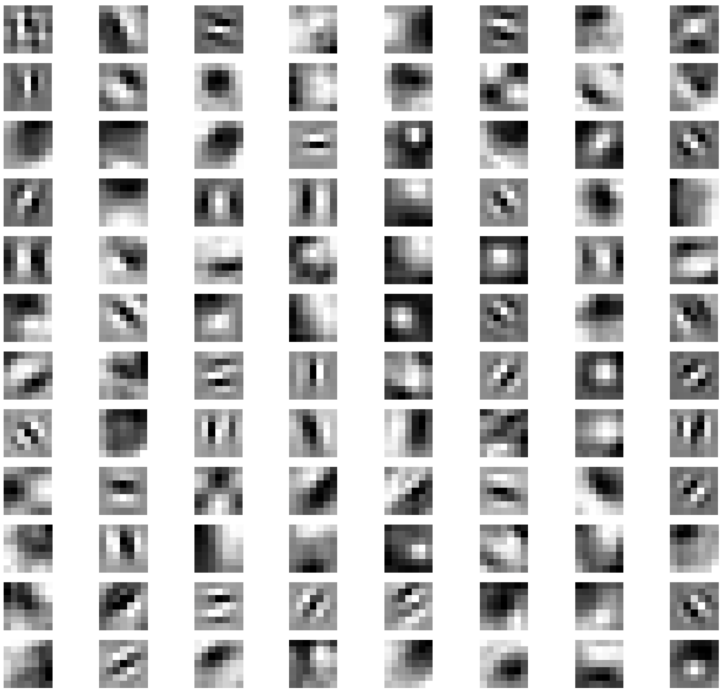


Fig. 24: Weights from Conv1 layer

These weights clearly are much more distributed than the original implementation all thanks to residual connections which helped backpropagate the error and not vanish it. The introduction of residual connections has proven to be instrumental in mitigating the issue of vanishing gradients during training. By facilitating the propagation of error signals through the network more effectively, residual connections helped the network to capture rich and more nuanced features of CIFAR-10 data.

C. SqueezeNet with DSD traing approach

We implemented the Dense-Sparse-Dense (DSD) training approach from scratch for the SqueezeNet architecture on the CIFAR-10 dataset, leveraging insights from existing implementations such as the codebase for applying DSD to the LeNet architecture on the MNIST dataset. [11] Then, we conducted numerous experiments to investigate the impact of varying sparsity levels and dense-sparse-dense epoch distributions on the performance and efficiency of the trained SqueezeNet model.

- **Varying Sparsity Levels:** In our experimental investigation on varying sparsity levels, we explored the difference of varying levels of sparsity on the quantifiable performance of the SqueezeNet network trained using Dense-Sparse-Dense (DSD) training. By systematically adjusting the sparsity parameter, we performed numerous experiments to quantify the model using its train and validation loss over a range of sparsity levels. Our results demonstrate a clear relationship between sparsity and model performance, with higher sparsity levels leading to increased training and validation losses. Furthermore, we observed that while sparser models tend to exhibit reduced computational complexity, a trade-off is made between network sparsity and accuracy. These findings underscore the importance of carefully selecting the optimal sparsity level based on the desired balance between model size, computational efficiency, and performance metrics. Moreover, it is important that since our experiments were conducted using the SqueezeNet architecture, which, due to its design complexity, posed challenges when trained from start on the CIFAR-10 data. CIFAR-10, being a relatively small dataset compared to ImageNet, presented difficulties in achieving competitive results with SqueezeNet.
- **Different Epoch Distributions:**In our investigation of epoch distributions, we sought to understand the effect of varying the distribution of dense and sparse training epochs within the Dense-Sparse-Dense (DSD) training paradigm. By manipulating the number of dense and sparse epochs, we explored how the model's convergence behavior and final performance were influenced. Through a series of experiments with different epoch distributions, ranging from predominantly dense epochs to predominantly sparse epochs, we observed distinct patterns in the train and validation loss over the duration of training. Specifically, we found that increasing the proportion of sparse epochs resulted in slower convergence rates and higher final losses, indicating the importance of maintaining a balance between dense and sparse training phases for effective model optimization. These findings provide valuable insights into the interplay between epoch distributions and training dynamics within the DSD framework, offering guidance for optimizing training strategies for neural network compression and efficiency.

VI. Conclusion & Discussions

In conclusion, our experiments into the vanilla SqueezeNet architecture and its variant incorporating residual connections and DSD training paradigm has provided valuable insights into the effectiveness of architectural enhancements in deep neural networks.

For the vanilla SqueezeNet, we observed its simple design and operational efficiency, proving it worthy in environments with resource crunch like mobile and edge devices. Despite its simplicity, SqueezeNet achieved competitive performance on various tasks, demonstrating its potential for real-world deployment.

On the other hand, the integration of residual connections in SqueezeNet significantly enhanced its representational capacity. By enabling the network to learn residual mappings, these connections facilitated the training of deeper architectures while mitigating the vanishing gradient problem. Consequently, the SqueezeNet model with residual connections exhibited improved convergence properties and achieved higher accuracy on complex datasets.

The Dense-Sparse-Dense (DSD) training methodology, characterized by the iterative process of pruning and re-densing, is shown to significantly enhance optimization performance, particularly in the context of SqueezeNet. This approach effectively mitigates the challenge of navigating saddle points encountered during the optimization of deep neural networks by leveraging pruning to perturb the learning dynamics, facilitating movement away from these points. Similar to Simulated Annealing, DSD strategically deviates from converged solutions, inducing sparsity to enable escape from sub-optimal solutions. Furthermore, DSD fosters the attainment of superior minima by decreasing both training and validation losses, thus enhancing the robustness of SqueezeNet. The regularization inherent in sparse training shifts optimization to a smoother, lower-dimensional space, further augmenting the model's robustness against noise. Additionally, DSD offers robust re-initialization opportunities, mitigating the impact of weight initialization, and effectively breaking symmetry among hidden units, thereby reducing co-adaptation

risks. In conclusion, the application of DSD to SqueezeNet presents a compelling framework for neural network regularization, demonstrating remarkable optimization performance and underscoring its potential to significantly improve the accuracy and robustness of the network.

In summary, our analysis underscores the importance of architectural innovations in shaping the performance and efficiency of deep neural networks. While the vanilla SqueezeNet offers a lightweight solution for deployment in resource-constrained scenarios, the incorporation of residual connections and DSD training methodology empowers the model to learn more expressive representations, leading to superior performance across various tasks.

VII. Future Work & Scope for Business

In the sense of future developments and business applications, SqueezeNet holds significant potential, particularly now that industries need models that are lightweight. For potential future applications in Deep learning, computational efficiency, and model size play pivotal roles. As the demand for such intelligent systems continues to surge exponentially across domains such as autonomous vehicles, IoT devices, and mobile applications, SqueezeNet's lightweight architecture presents a promising solution. Looking ahead, advancements in SqueezeNet, to make it even more efficient could focus on enhancing its adaptability to specialized hardware accelerators, optimizing its performance for low compute environments.

Moreover, the business scope for this model extends beyond traditional machine learning applications and reaches into sectors where real-time processing and minimal computational overhead are extremely crucial. Its competitive efficiency enables faster inference even on embedded devices with limited resources, making it an attractive choice for businesses seeking to deploy AI-powered solutions at scale. For example, in the health sector, SqueezeNet's ability to run efficiently on low and very-low power devices could facilitate the development of wearable technology such as health monitors, and assistive technologies for remote patient care. Similarly, in the automobile industry, its small footprint could enable integration of intelligent features into vehicles, enhancing driver safety without compromising performance. As SqueezeNet continues to evolve, its role in shaping the landscape of embedded AI applications is poised to expand, driving innovation and bringing new avenues for growth.

APPENDIX A
ALGORITHMS

Algorithm 1: Procedure for Dense Sparse Dense training

Initialize: $\theta^{(0)}$ with $\theta^{(0)} \sim \mathcal{N}(0, 1)$
Output: $\theta^{(t)}$

Phase-1: Dense Phase
while not converged **do**
$\theta^{(t)} = \theta^{(t-1)} - \eta^{(t)} \nabla f(\theta^{(t-1)}; x^{(t-1)})$
$t = t + 1$
end while

Phase-2: Sparse Phase
$S = \text{sort}(|\theta^{(t-1)}|)$; $\lambda = S_k$
$Mask = \mathbb{1}(|\theta^{(t-1)}| > \lambda)$
while not converged **do**
$\theta^{(t)} = \theta^{(t-1)} - \eta^{(t)} \nabla f(\theta^{(t-1)}; x^{(t-1)})$
$\theta^{(t)} = \theta^{(t)} \odot$ Mask
$t = t + 1$
end while

Phase-3: Dense Phase
while not converged **do**
$\theta^{(t)} = \theta^{(t-1)} - \eta(t) \nabla f(\theta^{(t-1)}; x^{(t-1)})$
$t = t + 1$
end while
Go to *Sparse Phase* for more iterations of Dense Sparse Dense;

APPENDIX B
DERIVATION OF PARAMETERS IN ALEXNET

TABLE I: Parameter computation for AlexNet

Layer name/type	Filter size	Bias	Total parameters
Conv. layer 1	$11 \times 11 \times 3 \times 96$	96	34,944
Conv. layer 2	$5 \times 5 \times 96 \times 256$	256	614,656
Conv. layer 3	$3 \times 3 \times 256 \times 384$	384	885,120
Conv. layer 4	$3 \times 3 \times 384 \times 384$	384	1,327,488
Conv. layer 5	$3 \times 3 \times 384 \times 256$	256	884,992
FC Layer 1	$6 \times 6 \times 256 \times 4096$	4096	37,752,832
FC Layer 2	4096×4096	4096	16,781,312
FC Layer 3	4096×1000	1000	4,097,000
Total	-	-	62,380,346

Total Parameters: 62,380,346 (over 62 million parameters)

APPENDIX C
DERIVATION OF PARAMETERS IN SQUEEZENET

TABLE II: Parameter computation for SqueezeNet

Layer type	Output dimension	Filter dim/stride	Depth	sq1x1	exp1x1	exp3x3	Total parameters (nor pruned)	Total parameters (pruned)
Input image	224x224x3	-	-	-	-	-	-	-
conv1	111x111x96	7x7/2 (x96)	1	1	6-bit	14,208	14,208	
maxpool1	55x55x96	3x3/2	0	-	-	-	-	-
fire2	55x55x128	2	16	64	64	6-bit	11,920	5,746
fire3	55x55x128	2	16	64	64	6-bit	12,432	6,258
fire4	55x55x256	2	32	128	128	6-bit	45,344	20,646
maxpool4	27x27x256	3x3/2	0	-	-	-	-	-
fire5	27x27x256	2	32	128	128	6-bit	49,440	24,742
fire6	27x27x384	2	48	192	192	6-bit	104,880	44,700
fire7	27x27x384	2	48	192	192	6-bit	111,024	46,236
fire8	27x27x512	2	64	256	256	6-bit	188,992	77,581
maxpool8	13x12x512	3x3/2	0	-	-	-	-	-
fire9	13x13x512	2	64	256	256	6-bit	197,184	77,581
conv10	13x13x1000	1x1/1 (x1000)	1	1	6-bit	513,000	103,400	
avgpool10	1x1x1000	13x13/1	0	-	-	-	-	-
Total	-	-	-	-	-	-	1,248,424	421,098

Total Parameters: 1,248,424 (about 1.25 million parameters)

ACKNOWLEDGMENT

The entire project team wish our deepest appreciation to Dr. Predrag Jelenkovic and Aidan Therien who have been extremely supportive in helping us to complete this project, thereby improving our knowledge experience in the domain of Statistical Deep Learning.

REFERENCES

[1] Urja Banati et al. "Soft Biometrics and Deep Learning: Detecting Facial Soft Biometrics Features Using Ocular and Forehead Region for Masked Face Images". In: (Dec. 2021). DOI: 10.21203/rs.3.rs-1174842/v1.

[2] Wikimedia Commons. *File:Typical cnn.png — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:Typical_cnn.png&oldid=671717156. [Online; accessed 4-May-2024]. 2022.

[3] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: (2009), pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[4] Song Han et al. "DSD: Dense-Sparse-Dense Training for Deep Neural Networks". In: *Published as a conference paper at ICLR 2017* (2017). URL: https://arxiv.org/pdf/1607.04381.

[5] Xiaobing Han et al. "Pre-Trained AlexNet Architecture with Pyramid Pooling and Supervision for High Spatial Resolution Remote Sensing Image Scene Classification". In: *Remote Sensing* 9.8 (2017). ISSN: 2072-4292. DOI: 10.3390/rs9080848. URL: https://www.mdpi.com/2072-4292/9/8/848.

[6] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[7] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size*. https://doi.org/10.48550/arXiv.1602.07360. 2016. arXiv: 1602.07360 [cs.CV].

[8] Ameya D. Jagtap. *Adaptive activation functions accelerate convergence in deep and physics-informed neural networks*. https://www.researchgate.net/figure/Left-to-right-Sigmoid-or-logistic-tanh-ReLU-and-Leaky-ReLU-activation-functions-for_fig2_333632820. [Online; accessed 4-May-2024]. 2019.

[9] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. Technical Report, University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[11] Ferdinand Mom. *DSD Training Repository*. https://github.com/3outeille/DSD-training/blob/master/src/mnist_dsd_pytorch.ipynb. 2021.

[12] Aili Wang et al. "A Dual Neural Architecture Combined SqueezeNet with OctConv for LiDAR Data Classification". In: *Sensors* 19.22 (2019), p. 4927. DOI: 10.3390/s19224927. URL: https://www.mdpi.com/1424-8220/19/22/4927.