

SqueezeNet

EECS 6699: Mathematics for Deep Learning Project Presentation

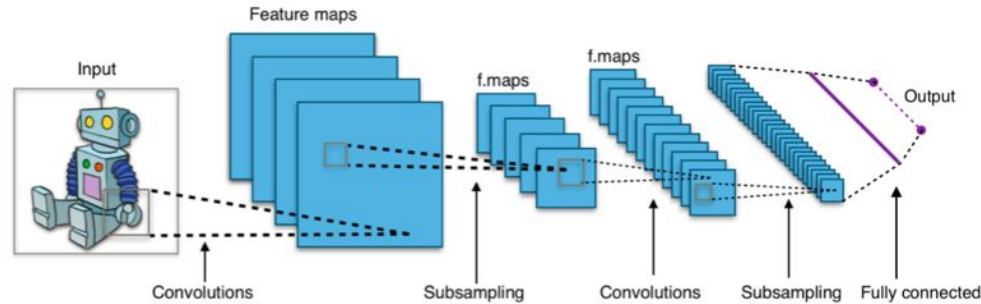
Apurva Patel (amp2365)
Harsh Benahalkar (hb2776)
Devika Gumaste (dg3370)

Spring 2024



COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Traditional Convolution Neural Networks (CNNs)



- **Complex Data Processing:** CNNs are designed to handle complex data structures like images, videos, and audio, which have high-dimensional data representations.
- **Feature Extraction:** CNNs excel at automatically learning hierarchical patterns and features from raw data. Through successive convolutional and pooling layers, CNNs can identify low-level features like edges and textures, gradually building up to higher-level features like shapes, objects, and semantic concepts.
- **Applications in Computer Vision:** CNNs have revolutionized the field of computer vision by enabling tasks such as image classification, object detection, facial recognition, and image segmentation with unprecedented accuracy.

Motivation

Why do we need smaller models?

- Lack of compute resources as the cloud and storage industry is not advancing as exponentially as the Deep Learning industry is.
- Increasing complexity of models to adapt to future use cases where autonomous systems have been integrated into society.
- Need to deploy models on edge/mobile devices to remove the barrier of Deep Learning as just a cloud/remote deployable system.

How does SqueezeNet solve this problem

- The need for squeezenet is justified by the model size which has just about 1.2M parameters.
- Comes handy when use cases demand of real time edge computing without significant performance drop and onboard memory is limited.

Model	No. of Layer	Parameters (Million)	Size
AlexNet	8	60	-
VGGNet-16	23	138	528 MB
VGGNet-19	26	143	549 MB
Inception-V1	27	7	-
Inception-V3	42	27	93 MB
ResNet-152	152	50	132 MB
ResNet-101	101	44	171 MB
InceptionResNetV2	572	55	215 MB
MobileNet-V1	28	4.2	16 MB
MobileNet-V2	28	3.37	14 MB
EfficientNet B0	-	5	-

Some models with their footprint size and number of parameters

Principal Theorems

1. Wiener-Khinchin theorem (cross-correlation):

Wiener-Khinchin theorem is applied to understand the frequency-domain representation of signals.

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

2. Universal Approximation theorem:

Universal Approximation Theorem assures that a neural network with at least one single hidden layer can approximate any continuous function.

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

3. Lipschitz Continuity of Neural Networks:

Lipschitz continuity ensures stability and smoothness of neural network outputs with respect to small changes in inputs.

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

4. Gradient Descent algorithm:

Gradient descent efficiently updates neural network parameters by iteratively minimizing a loss function.

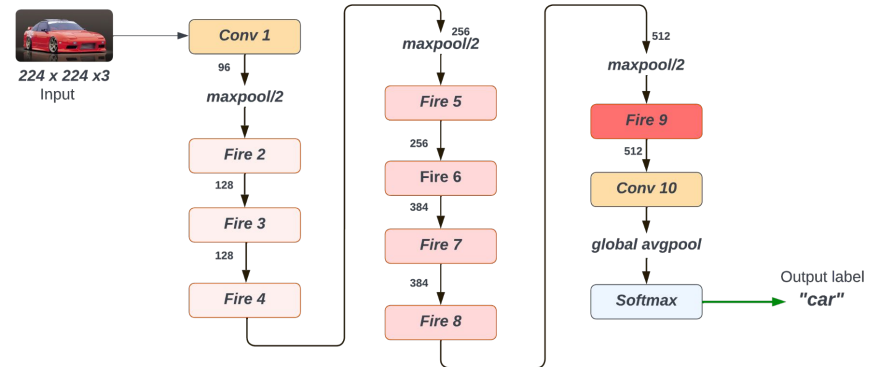
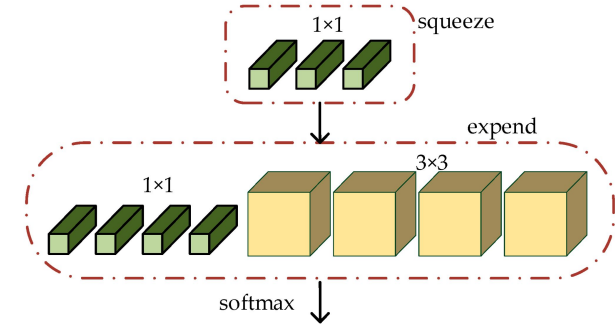
$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

SqueezeNet Architecture

Stages of the Fire module:

- 1. Compression**
 - Replacing 3x3 filters with 1x1 filters (squeeze layers).
 - Downsampling feature maps.
- 2. Expansion**
 - Applying 1x1 and 3x3 convolution on the input parallelly.
 - 1x1 keeps input dimension intact and the 3x3 filter captures spatial features.
- 3. Concatenation**
 - Channel-wise addition of 1x1 and 3x3 convolutions.
- 4. Activation**
 - Introduces non-linearity in the module.

Fire module representation



Vanilla squeezeNet architecture

Experiments

Optimizers used:

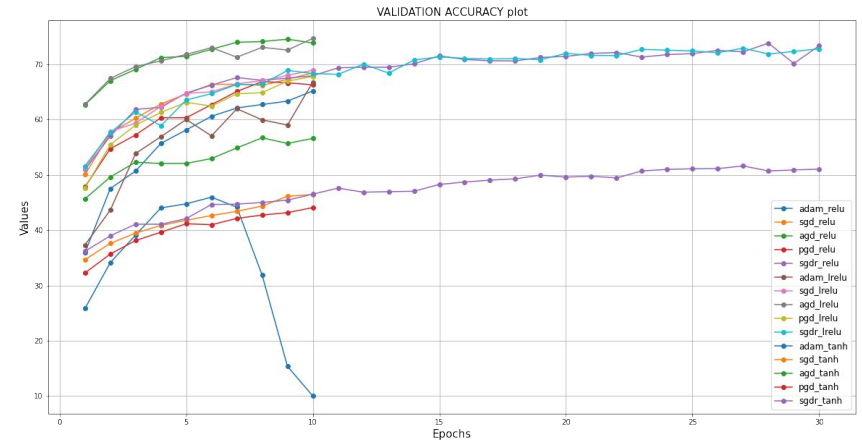
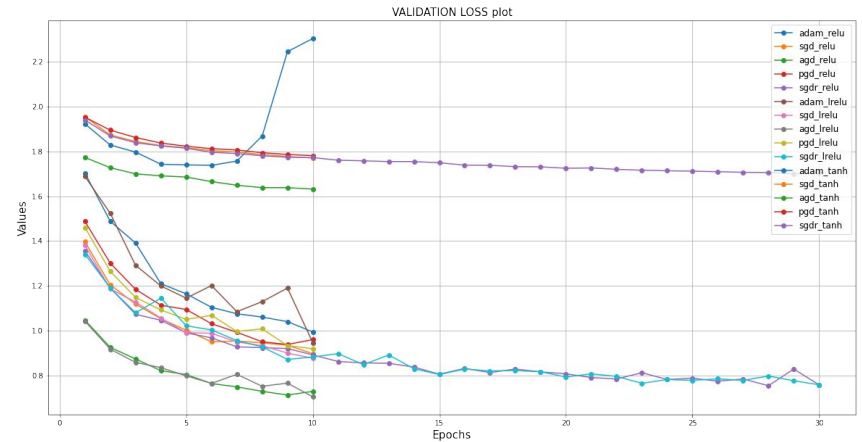
- Adaptive Moment (Adam)
- Stochastic Gradient Descent (SGD)
- Perturbed Gradient Descent (PGD)
- Accelerated Gradient Descent AGD (SGD with momentum)
- SGD with restart

Activations used:

- ReLU
- Leaky ReLU
- TanH

No. of parameters: 740554

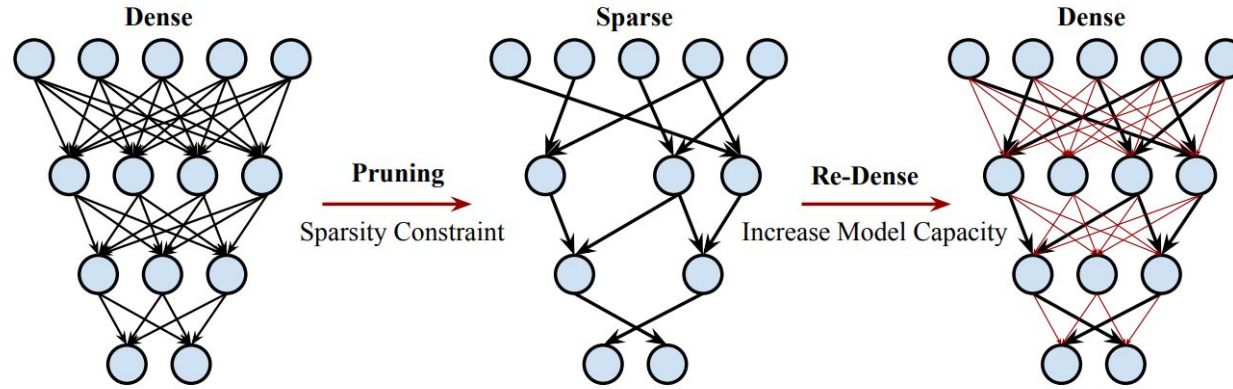
Model Size after training: 11.3 Mb



Our Implementations

- 1. Dense Sparse Dense Training**
- 2. Squeezenet with Residual Connections**

Dense-Sparse-Dense Training



- **Motivation:** Large complicated models are prone to capturing noise, rather than finding the intended pattern in the dataset, leading to **overfitting** and high variance. Simply reducing the model capacity leads to **underfitting** and high bias.
- **Approach:** The authors of the parent paper propose a new solution - “**Dense Sparse Dense**” approach - a novel training strategy that starts with conventional training, then regularizes the model with sparsity- constrained optimization, and finally increases the model capacity by restoring and re-training the pruned weights.

DSD Algorithm Overview

Three step process -

- Initial Dense Training
- Regularization with sparse training
- Final Dense Training by restoring and re-training the pruned weights

Algorithm 1: Workflow of DSD training

Initialization: $W^{(0)}$ with $W^{(0)} \sim N(0, \Sigma)$

Output : $W^{(t)}$.

Initial Dense Phase

while not converged do

$W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)});$
 $t = t + 1;$

end

Sparse Phase

// initialize the mask by sorting and keeping the Top-k weights.

$S = \text{sort}(|W^{(t-1)}|); \lambda = S_{k_i}; \text{Mask} = \mathbb{1}(|W^{(t-1)}| > \lambda);$

while not converged do

$W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)});$
 $W^{(t)} = W^{(t)} \cdot \text{Mask};$
 $t = t + 1;$

end

Final Dense Phase

while not converged do

$W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)});$
 $t = t + 1;$

end

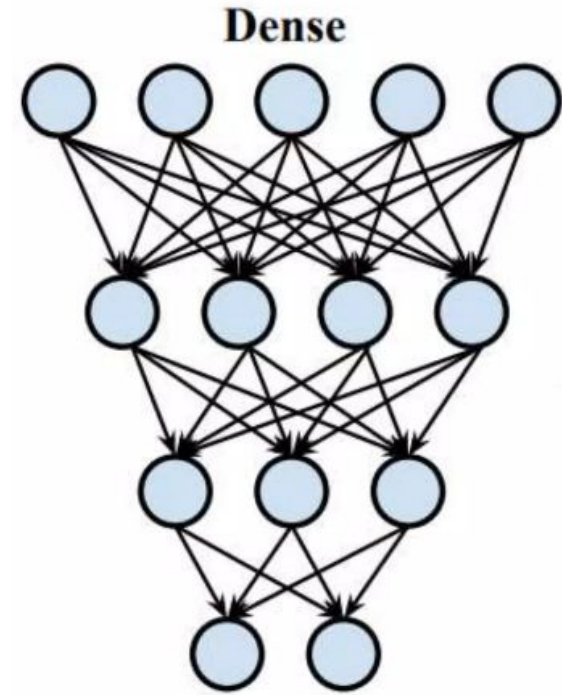
goto *Sparse Phase* for iterative DSD;

Phase-1 Dense Training

Dual Objective

- The first step focuses on training the network using all connections.
- The network learns both the values of the weights and identifies the important connections.
- The importance of connections is quantified by simply taking the absolute value of the weights - weights with higher absolute values are considered important.

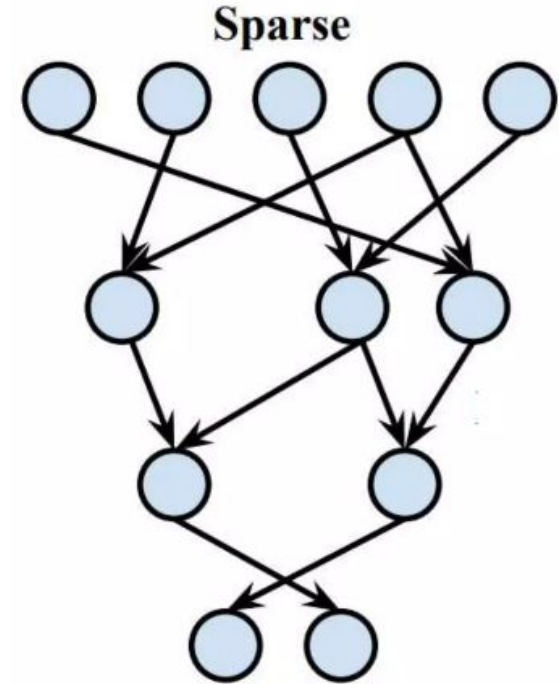
```
while not converged do  
     $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)});$   
     $t = t + 1;$   
end
```



Phase-2 Sparse Training

- Connecting with low weights are pruned to induce sparsity. The degree of sparseness is controlled by a hyperparameter “sparsity”.
- Threshold = kth largest weight, where $k=N(1 - \text{sparsity})$.
- Binary Mask is generated for each layer to prune weights.
- Network is trained while enforcing the binary mask

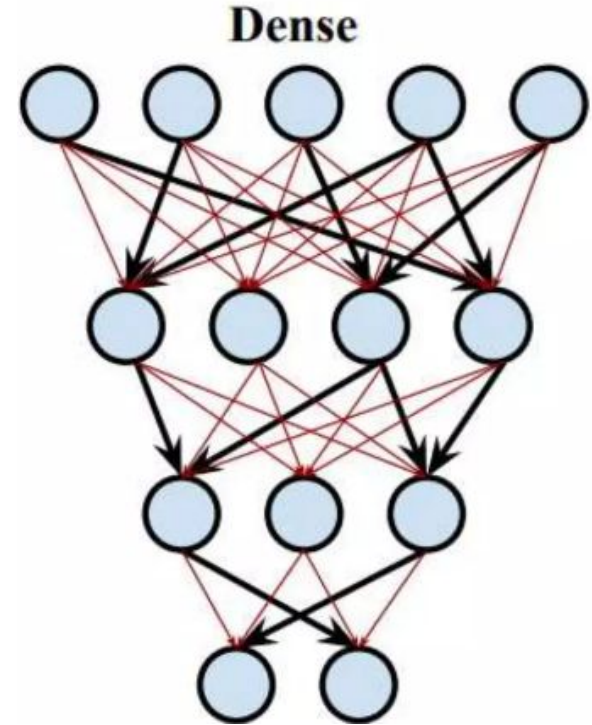
```
// initialize the mask by sorting and keeping the Top-k weights.  
 $S = \text{sort}(|W^{(t-1)}|)$ ;  $\lambda = S_{k_i}$ ;  $\text{Mask} = \mathbb{1}(|W^{(t-1)}| > \lambda)$ ;  
while not converged do  
     $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$ ;  
     $W^{(t)} = W^{(t)} \cdot \text{Mask}$ ;  
     $t = t + 1$ ;  
end
```



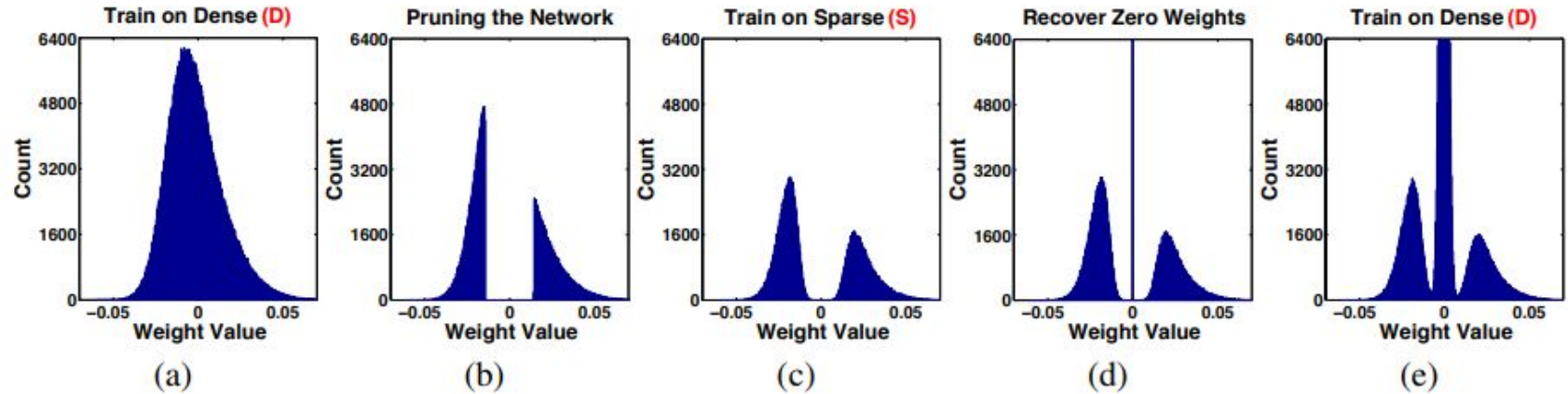
Phase-3 Dense Training

- Initially pruned connections are recovered and initialized anew
- Entire network is trained on a reduced learning rate (1/10th) of the original rate - since the sparse network is already close to a good local minimum.
- Other hyperparameters remain unchanged to maintain consistency and stability in the training process.
- Final dense training enhances capacity of the model - allowing the model to reach to a better local minimum.

```
while not converged do  
     $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)});$   
     $t = t + 1;$   
end  
goto Sparse Phase for iterative DSD;
```



DSD Weights Visualization



Weight distribution is initially centered around zero with rapid tail decay.

Pruning based on absolute value truncates the central region, leading to a bimodal distribution during retraining.

Pruned weights are then re-initialized to zero at the start of re-dense training and eventually retrained alongside un-pruned weights.

- **Varying Sparsity Levels**

- Sparsity Level is systematically varied to observe it's effects on performance.
- Optimal sparsity level between 25-50%
- For our set-up, optimal sparsity turned out to be 40%
- Trade-off between sparsity and accuracy

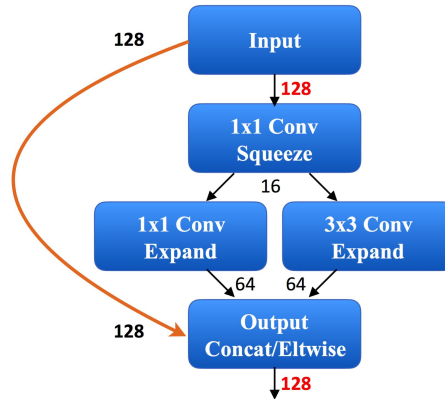
- **Different Epochs Distribution**

- Distribution of dense and sparse epochs
- Instead of allocating equal number of epochs to each phase, tested out different combinations.
- Differences observed in accuracy and training time
- 5 - 3 - 5 resulted in best performance

Residual connections

SqueezeNet with residual connections

- **Motivation:** Helps reformulate the underlying mapping as a residual mapping, rather than expecting the stacked layers to directly fit a desired underlying mapping.
- If the desired underlying mapping is $H(x)$, the residual learning framework lets the layers fit another mapping $F(x) = H(x) + x$, where x is the input to the layer.

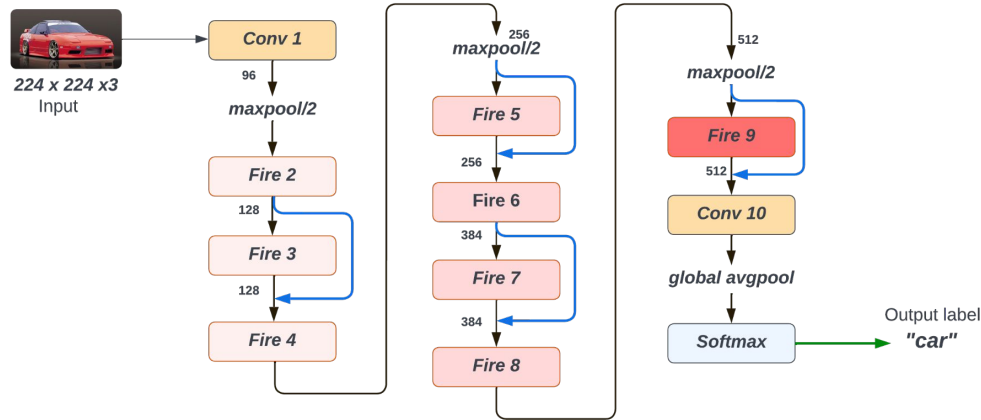


$$y_i^{res} = y_i + x_{i+1}$$

Fire module with residual connections between layers to improve backpropagation of error.

Residual connections

SqueezeNet architecture with residual connections



Cross-Entropy loss function

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

Error computation for backpropagation

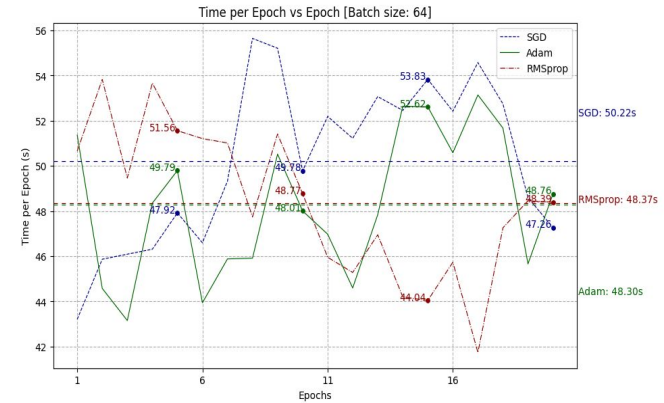
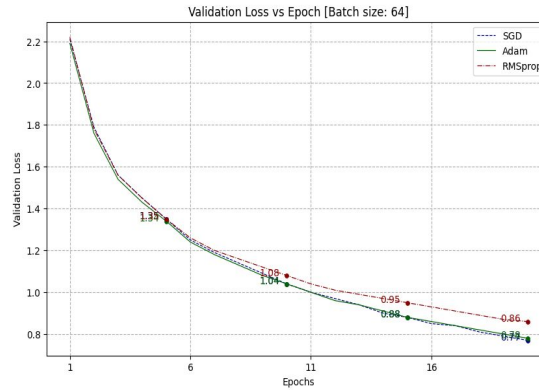
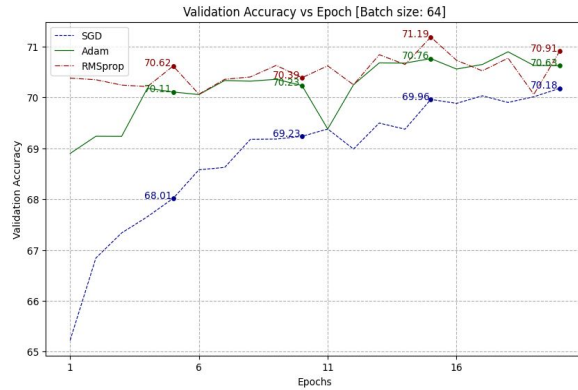
$$\frac{\partial \mathcal{L}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial y_L} \frac{\partial y_L}{\partial y_{L-1}} \dots \frac{\partial y_{l+1}}{\partial y_l} \frac{\partial y_l}{\partial F_l(x_l)} \frac{\partial F_l(x_l)}{\partial x_l} + \frac{\partial \mathcal{L}}{\partial y_{l+1}} \frac{\partial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

- The key difference is the addition of the second term which represents the gradient flowing through the residual connection, and the residual connection ensures that the gradients can be expressed 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

Results & Discussions

Batch Size: 64

Learning rate: 0.001



for RMSProp:

Exponentially decaying average:

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot (g_t)^2$$

Update rule:

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

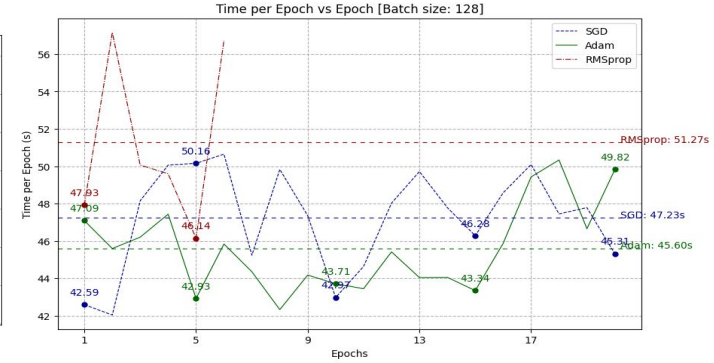
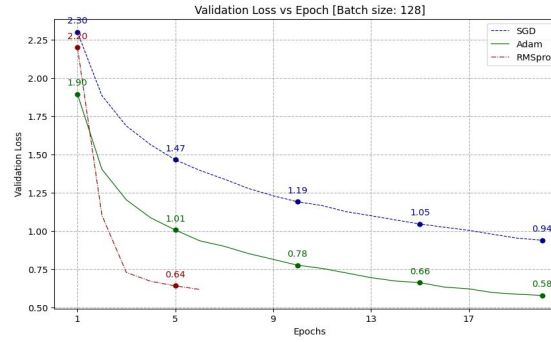
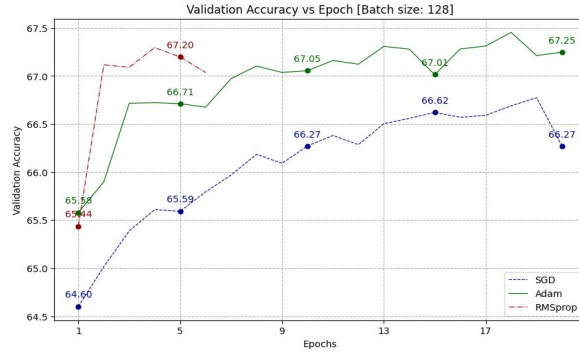
- RMSprop adapts the learning rates for each parameter based on the root mean square (RMS) of recent gradients. This adaptiveness allows it to converge faster and more robustly, especially in scenarios where the gradients exhibit large variations or are sparse.
- Average time(s) per epoch is highest for SGD as the no of updations required in that during the update rule is slightly higher than that ADAM and RMSProp where update takes place without momentum instead utilizing adaptive gradient.

Note: Time per epoch is a very lucrative measure as it is highly dependant on instruction pipeline and so we consider average time per epoch

Results & Discussions

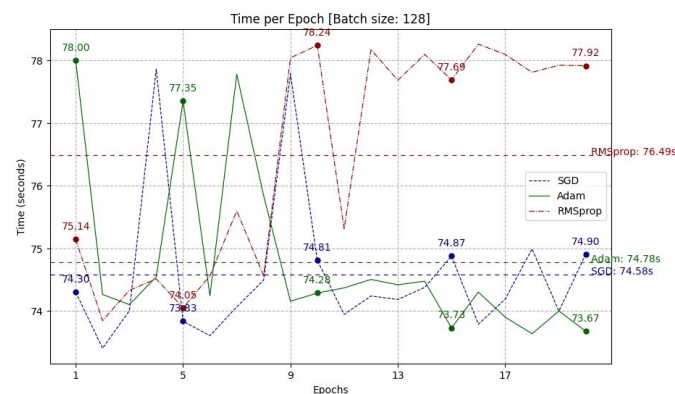
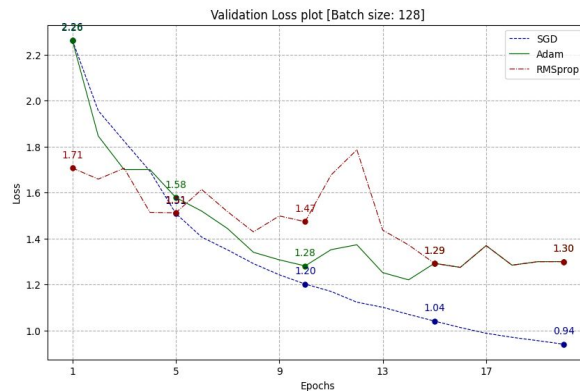
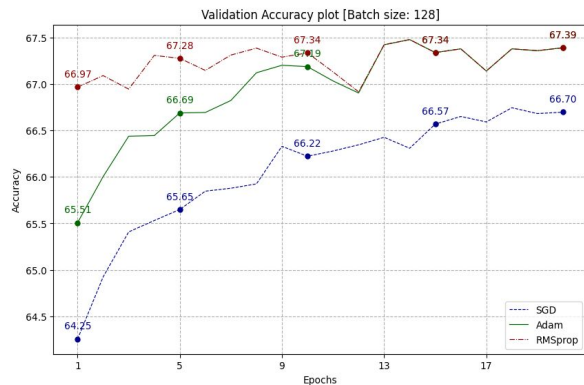
Batch Size: 128

Learning rate: 0.001



Results & Discussions

Batch Size: 128 with LR scheduler



for SGD with Nesterov momentum: $\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t) + \alpha \cdot \Delta\theta_{t-1}$

- In complex landscapes like CIFAR10, the momentum term will exacerbate oscillations or hinder convergence in narrow valleys, leading to suboptimal solutions compared to adaptive methods like Adam and RMSprop, which adjust learning rates based on gradient magnitudes.
- It is noteworthy to note that despite this the validation loss is lowest, which can be explained by the momentum being taken into consideration and making the “distance” closest to global minima.
- Also varying learning rate doesn't help in RMSProp in our case because moment updation will be costly

Results & Discussions

- Weights clearly are much more distributed than the vanilla squeezenet implementation.
- Residual connections helped backpropagate the error and not vanish the gradients with lower values.
- Propagation of error signals through the network more effectively, residual connections have enabled the model to learn richer and more nuanced representations of the input data.



Sample weights in conv1 layer

Thank You :)