BMI 826 / CS 838 Homework Assignment 2 Writeup

Team Members and Contributions	
Name	Contribution
Deepan Das (ddas27@wisc.edu)	• 3.1 Forward and Backward Propagation • 3.2 Sections 1, 2, 3 • Writeup
Noor Mohamed Ghouse (mohamedghous@wisc.edu)	• 3.2 Sections 0, 1 • 3.3 Attention and Adversarial Samples • Writeup
Shashank Verma (sverma28@wisc.edu)	• 3.1 Forward and Backward Propagation • 3.2 Sections 1, 2, 3 • Writeup
NOTE: In effect, everyone had **equal** total contribution (If someone did not participate in direct coding for a section, then they participated in discussions, or in terms of understanding and code review)	

3. 1 Understanding Convolutions

3. 1. 1 Forward Propagation

In our implementation, the first step is to calculate the height and width of the output activation map using the input dimensions, kernel size, padding and stride values. These calculations will be used to fold the resultant tensor after the convolution. To implement the convolution operation, we first unfold both the input and the weight matrices, and then calculate the output activation map using a careful matrix multiplication operation. Note that the output activation map that we get here is in the unfolded form. We add the bias scalar individually in each of output activation maps throughout the depth of the output. This is done for the whole batch of images (Note that they share the same biases and filter bank). Finally, we fold the output to the desired shape as calculated in the first steps of our implementation. We also make sure to save important parameters in a PyTorch context object for the backward pass like the unfolded input tensor, weights and biases, stride, padding, and input height and width values.

3. 1. 2 Backward Propagation

Backward pass can also be implemented using convolution operations. Just like forward pass, we have used the trick of first unfolding and then doing a matrix multiplication to implement the convolution.

1. We obtain the unfolded grad_output tensor, and the unfolded weight tensor. Note that the weight tensor had been saved in the context, along with the unfolded input and biases during the forward pass. We obtain the gradients wrt input (grad_input) using a matrix multiplication between the transposed unfolded weight tensor and the unfolded grad_output. The output of this operation is then folded to the size of the input to get the final grad_input.
2. For the gradient wrt the weights (grad_weight), we simply do a matrix multiplication between the unfolded grad_output, and the unfolded transposed input tensor (from the context). The output of this operation can be folded to obtain the final grad_weight that we return.

NOTE: Using "test_conv.py", we have verified that all our tests for both forward and backward propagation are passing.

3. 2 Design and Train a Convolutional Neural Network

3. 2. 0 Training SimpleNet for 90 epochs

We trained the already provided SimpleNet for 90 epochs on GCP. As expected, we obtained a top-1 validation accuracy of 46.94% and top-5 validation accuracy of 75.82% at the end of 90 epochs. The accuracy plots are depicted in the figure below.

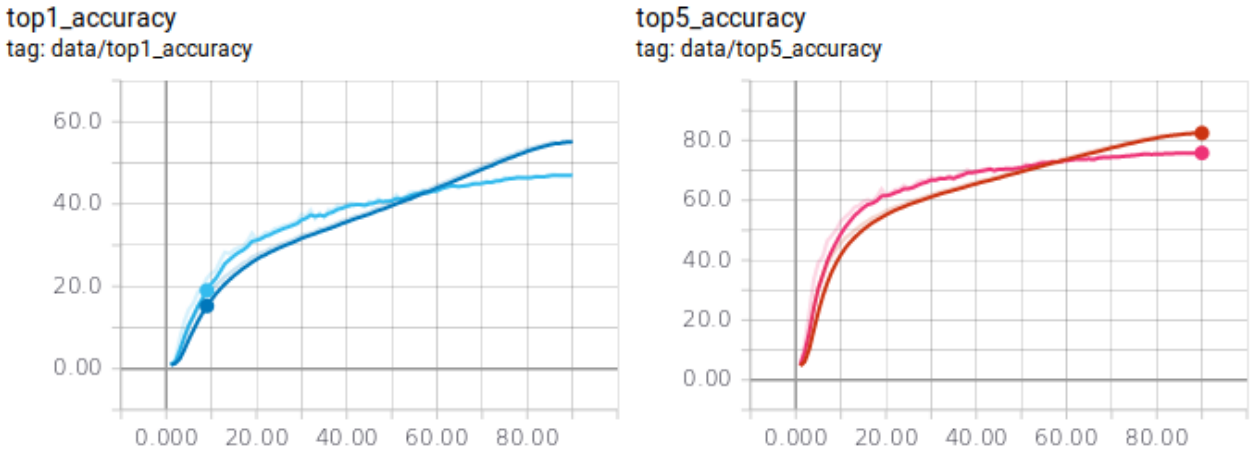


Figure: Top-1 (Sky Blue: Validation; Navy Blue: Training) and Top-5 (Pink: Validation; Red: Training) accuracy plots.

From the accuracy plots, we observe that in both the top-1 and top-5 cases, the validation accuracy is initially higher, and then it bypasses the training accuracy only to plateau. It seems like the SimpleNet model is probably overfitting the training data around 50 epochs of training. One thing we could do to prevent overfitting is to add more regularization like Batch Normalization, Dropout layers, weight decay, and add more augmented data.

3. 2. 1 Description of the training process

The model and the supporting code given has been used for training a deep neural network for scene recognition. The dataset used is the MiniPlaces dataset. The following are some key observations from the given code:

- The **Stochastic Gradient Descent** (SGD) optimizer is used. When the learning rate is scheduled to decay over time, SGD shows a similar convergence pattern like the standard gradient descent algorithm. By default, a momentum of 0.9 is used for training. Weight decay of 10^{-4} is used by default for regularization during optimization.
- The learning rate (by default, it starts at 0.1) decays over time based on a **Cosine based decay** function. This would help the model converge.
- The measure of loss is the **Cross-Entropy** Loss. Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is.
- For **regularization**, weight decay (L2) and data augmentation have been used. In addition, SGD would naturally cause noisy gradients, which could also be considered a form of regularization.
- Top-1 and Top-5 are the two accuracy measures used to gauge the model. **Top-K** accuracy basically gives us the credit for the right answer if our right answer appears in the top K guesses. Sometimes the predictive accuracy (top-1) can be a **misleading** accuracy measure in an unbalanced dataset. Some classes may contain much more data as compared to some other classes. The classifier can tend to be biased towards the majority class and hence perform poorly on the minority class. In these cases, top-5 accuracy can be a better measure, where we give the model a bit more leeway because the data might be biased.

3. 2. 2 Custom convolution training for 10 epochs

For this, we analyze the training loss of the two models.

1. **Convergence Rate**

It is seen that our implementation of the conv2d function is quite close in performance when it comes to converging to the local minimum. In terms of training loss (see figure), the graph for PyTorch conv2d does seem to drop faster initially but is soon caught up by our custom conv2D around 10 epochs.

2. Training Speed

The training loss curve is also useful in assessing the training speed of the model as we can see that our implementation converges to the same minimum as the PyTorch's implementation within 10 epochs. To reach 10 epochs, out custom conv2D took approximately 54 minutes, in comparison to the 40 minutes taken by the PyTorch conv2D. This indicates that the custom conv2D might have some scope for further improvement in terms of efficiency. One reason that our custom conv2d takes a bit more time could that even though we tried to avoid all loops in both the forward and backward propagation, we still have a loop for adding bias during the forward propagation (which we couldn't avoid because of some errors in tensor math). Additionally, we suspect some of the parts of the PyTorch conv could have been written in C/C++ giving it an edge in terms of both speed and memory efficiency.

3. Training Memory

On running nvidia-smi while the SimpleNet was training, using our custom conv2d, the total GPU memory consumption was consistently ~5400MB out of the available 11441MB for a Tesla K80. The PyTorch conv2d however seems to be more than twice as efficient, as the total GPU memory consumption was 2332MB / 11441MB. One reason that our custom conv2d takes more memory could be that we might have used more variables to capture intermittent calculations (fold, unfold, matrix multiplication) so that we have more clarity in code. We could probably compress the code even further by removing these extra intermittent variables.

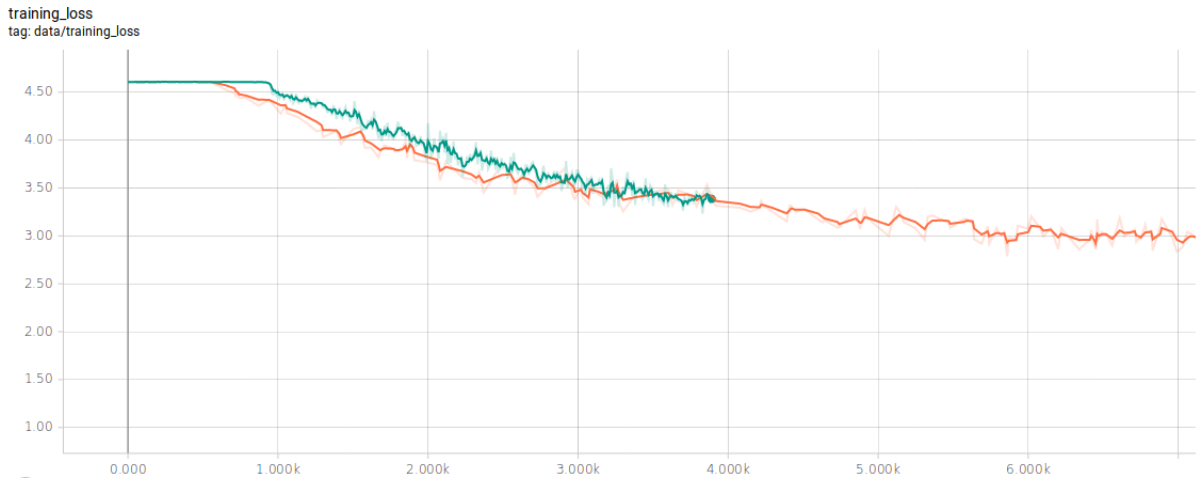
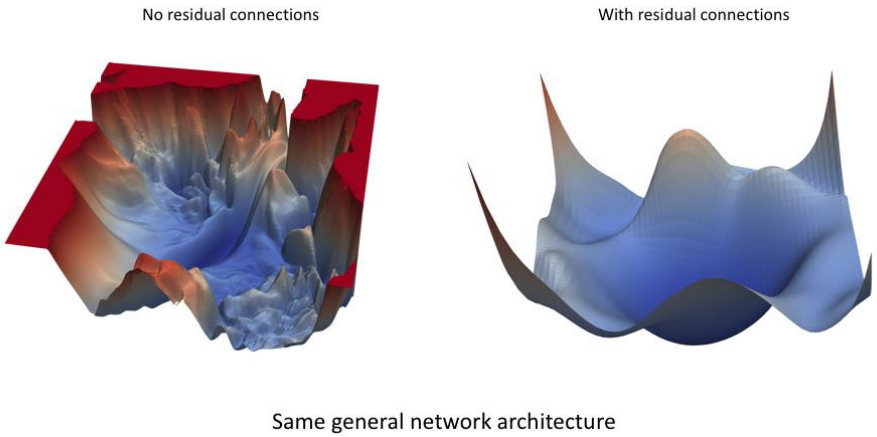


Figure: Training loss of SimpleNet trained using **custom conv2d** (Green) function for 10 epochs vs in-built **PyTorch conv2d** (Orange) function.

3. 2. 3 Customized CNN Architecture

The Mini-Places Challenge requires us to identify the scene category of an image. To address this classification problem, we take inspiration from two popular architectures in Deep Learning: Skip connections [2] and Inception modules [3]. We also take inspiration from the SimpleNet module provided by the instructor and base the framework of our modified architectures on that. Later, once we figure out a best performing model at the end of 4 hours of training, we use a lower learning rate scheduler to help the model learn better. In this section, we will be first explaining each model that we used and then show results corresponding to each model. In the last section, we explain our understanding and interpretation of the various results.

Training Neural Networks requires minimizing a high-dimensional non-convex loss function and we know that simple gradient methods often find global minimizers. However, this depends largely on the network architecture being used and various design choices affect the performance of the training. The primary reason for a neural network's performance can be attributed to its depth and a challenge for vanilla SGD on very deep networks is the problem of vanishing gradient. The much-celebrated work of skip connections in Residual networks has been found to diminish this problem a bit. The architecture titled **SkipNet** and its other versions presented here is basically inspired by such skip connections. We apply skip connections around the bottleneck layers. The **bottleneck** layers are an efficient way to reduce computation and has the same receptive field as intended. At the end of these bottleneck layers, we add fully connected layers without dropout layers for SkipNet. For modified versions of SkipNet (SkipNet V2), we implement one lesser bottleneck layer, thereby reducing the number of parameters in convolutional operations, and increase the number of fully connected layers with dropout. Each convolution operation has its own **batch normalization** and **ReLU** non-linearity. For the skip connections, the ReLU non-linearity is applied after the identity connection is added. It has also been shown in the work presented in [4] that Residual connections enable the network to minimize a smoother loss surface.



We modify SkipNet further by **adding Dilated Convolutions** and **removing Max-Pooling layers**. This is inspired by the work presented in [6]. As the image size is already really small(128x128), we thought it would be good to incorporate location invariance and a widened receptive field using a **dilated** convolution operation. The main advantage that dilated convolutions provide is that they support exponential expansion of receptive field without loss of resolution, with similar computational costs. Although pooling and strided convolutions implement similar ideas, they reduce the resolution. A reduction in resolution for scene images might not be desirable.

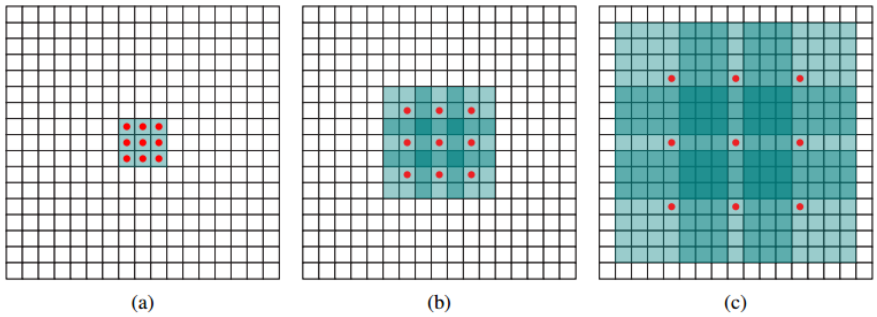
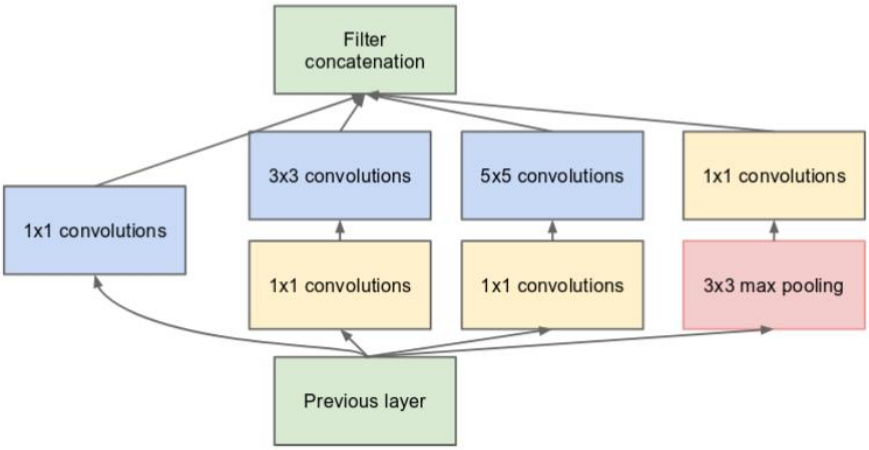


Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a) F_1 is produced from F_0 by a 1-dilated convolution; each element in F_1 has a receptive field of 3×3. (b) F_2 is produced from F_1 by a 2-dilated convolution; each element in F_2 has a receptive field of 7×7. (c) F_3 is produced from F_2 by a 4-dilated convolution; each element in F_3 has a receptive field of 15×15. The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

Another concept that we incorporated was inspired by the Inception net architecture. The most salient object in the image can have large variations in size, and each convolutional operation can be made to look at different sized receptive fields on the image to be scale-invariant. Furthermore, instead of having deeper networks, Inception Net makes use of a wider architecture and doesn't make the network too deep. This reduces chances of overfitting and also ensures that the gradient is backpropagated efficiently. In our model, we used an architecture that resembles InceptionNet_V4 that has skip connections around inception modules.



(b) Inception module with dimension reductions

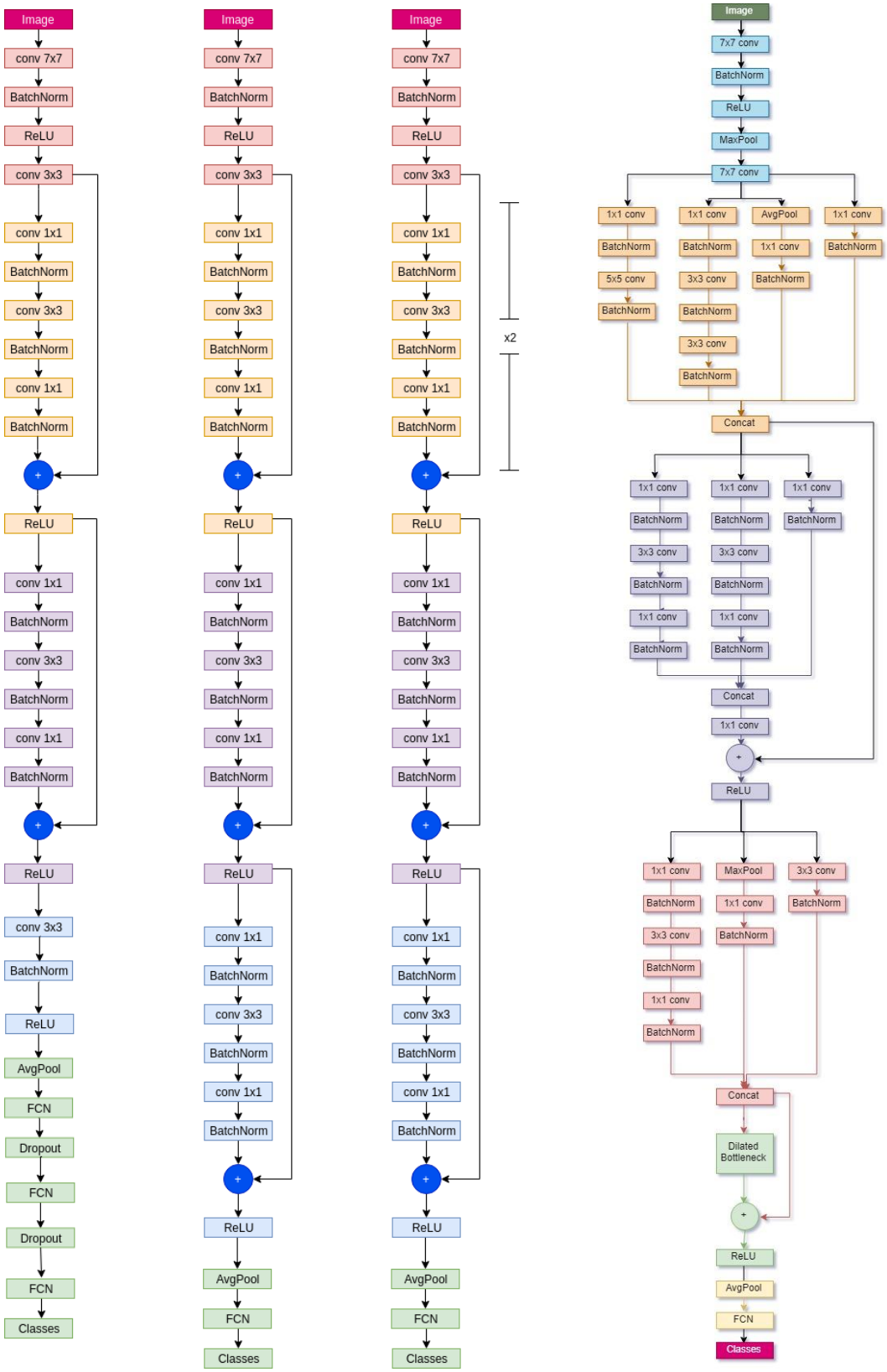


Figure: Different networks that we implemented. From Left: SkipNet_v2, DresNet, SkipNet, MixNet

Results

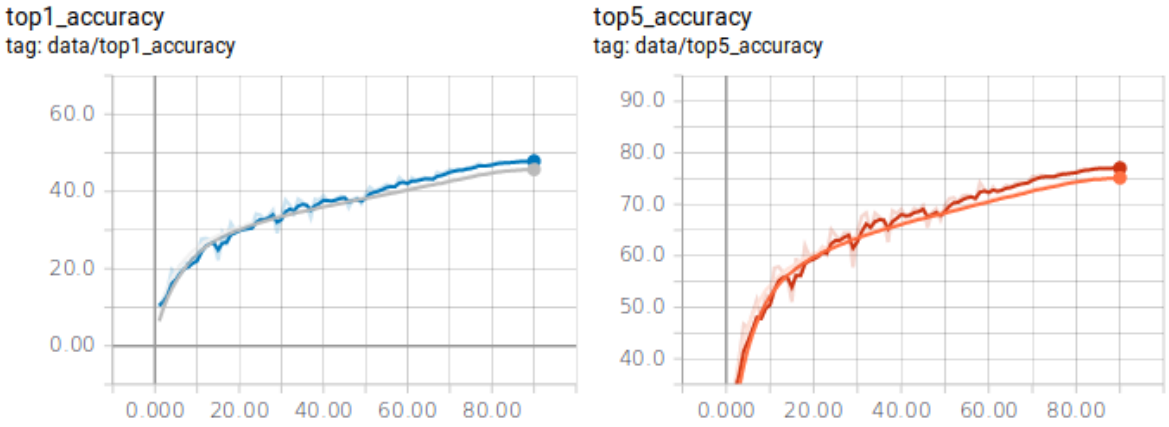
Simplicity is key: We find that **SkipNet_v2** performs better than all other models. We also tweaked the learning rate, once SkipNet_v2 beat the baseline of SimpleNet at the 70th epoch. We lowered the learning rate scheduler at this point and let SkipNet_v2 run as two different models. In the end, the scheduled model performs better in terms of Validation Accuracy@1. The proposed architecture outperforms all other architectures and only generates a slightly smaller Top-5 Validation accuracy than the non-scheduled SkipNet. The table summarizing the results is shown as below:

Model	**Training Accuracy**		**Validation Acc**		**LRate**	**Loss**
	@1	@5	@1	@5		
SimpleNet	55.13	82.47	46.96	75.82	8.01e	1.71
SkipNet	45.71	75.15	47.91	77.04	9.1e-7	2.05
SkipNet_v2	55.24	83.06	51.34	79.59	1.1e-7	1.58
SkipNet_v2_Scheduled	54.08	82.347	51.47	79.35	3.0e-7	1.68
MixNet	42.03	72.05	40.40	70.32	0.06	2.14
DresNet	69.83	90.27	44.54	73.94	0.02	0.89

Accuracy Curves

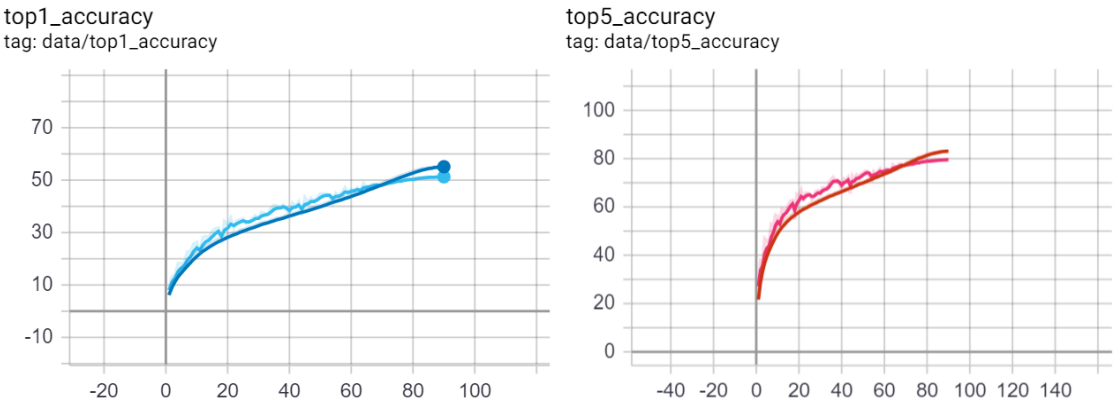
We present accuracy curves for all the models. We explain the model performance based on these curves.

SkipNet



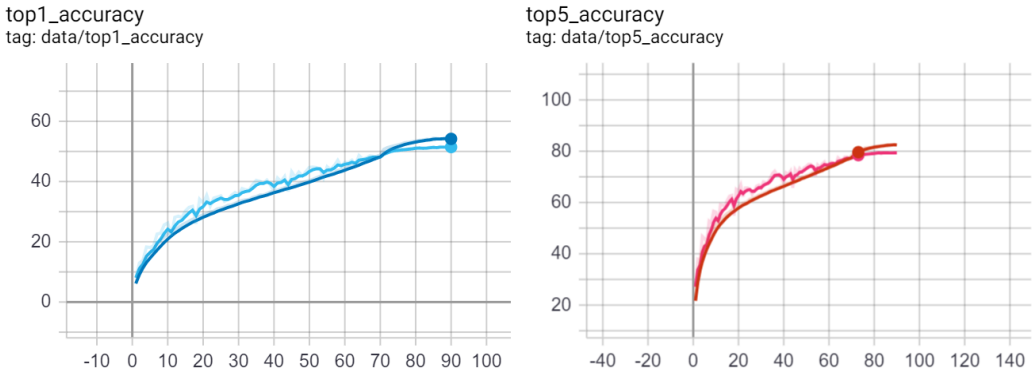
The training and validation accuracy curves for the SkipNet shows good convergence and the model is expected to overfit if training was continued after that. **The darker colored curves are the training accuracy in all cases.** Validation Accuracy @1 reaches 47.91% and reaches 77.04% @5.

SkipNet_v2

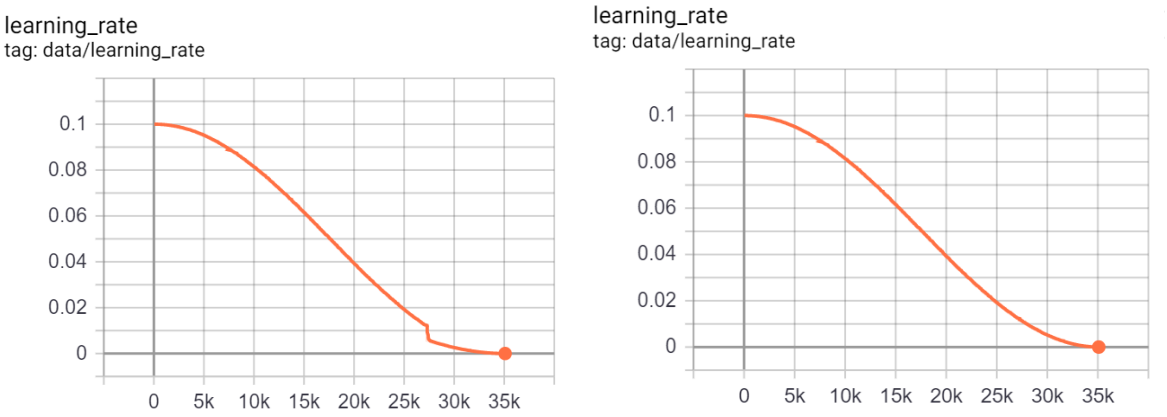


The training and validation accuracy curves for the SkipNet_v2 shows equally good convergence and the model is expected to overfit if training was continued after that. The darker colored curves are the training accuracy in all cases. Validation Accuracy @1 reaches 51.34% and reaches 79.59% @5. This model is lighter than the original SkipNet and has one less layer of bottleneck operations. We believe this model performed better due to the extra fully connected layers and the added regularization of dropout layers. Moreover, we reduce the number of parameters in the model, thereby reducing any overfitting even further.

SkipNet_v2_scheduled

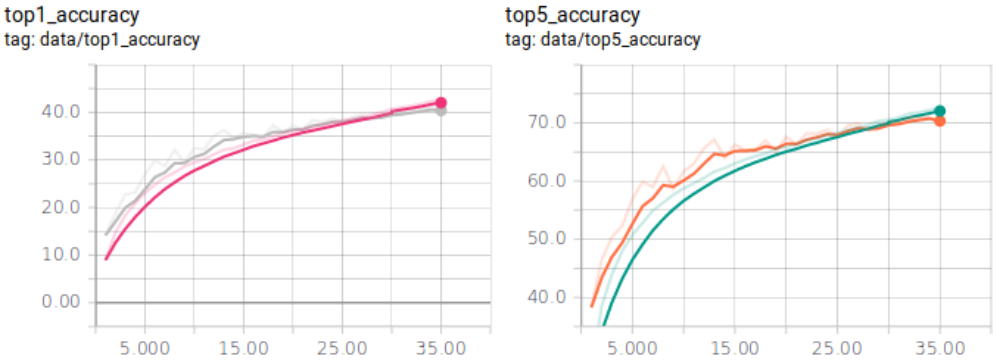


Once SkipNet_v2 reaches the baseline accuracy figures set by SimpleNet, we lower the learning rate scheduler and train the model differently. We reach at a better Validation Accuracy@1: 51.47%. We believe that this enables the network to learn better and maneuver better to reach the minima.



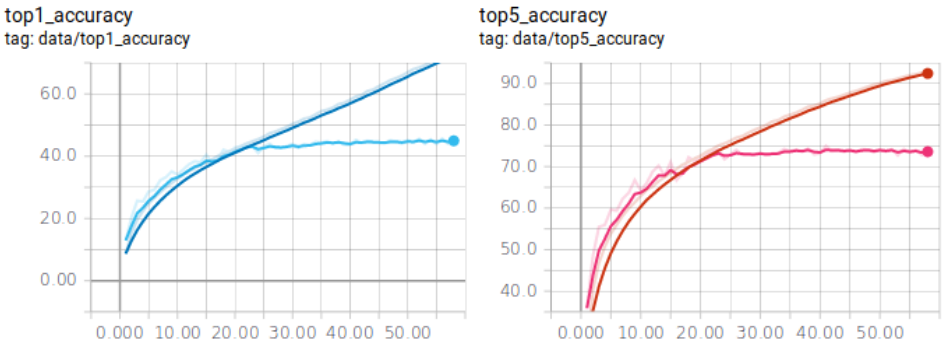
Left: Learning rate for SkipNet_v2_scheduled. **Right:** Learning rate for SkipNet_v2

MixNet

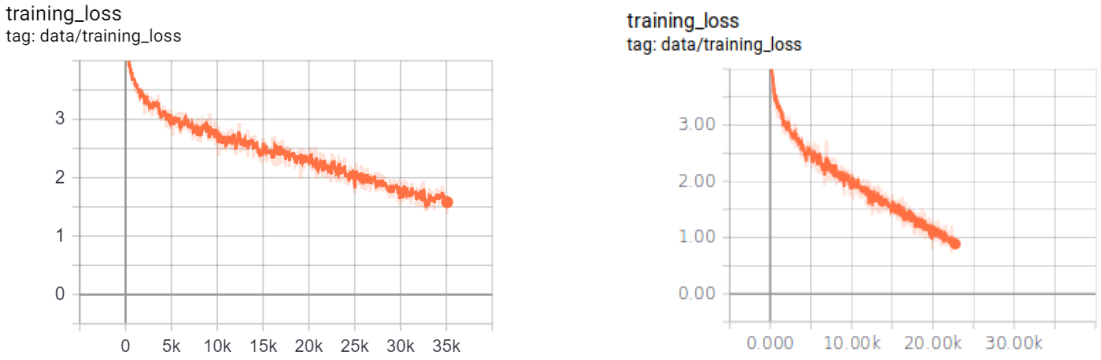


In MixNet, we trained a model that has skip connections added over inception_v4 like modules. We believe that this doesn't perform as well as SkipNet as the inception modules have different receptive field sizes. For an image of size 128x128, we expect that some of the larger receptive fields capture just too large a variance in information and consequently, fail to model the salient features of a given scene category. This in addition to the fact that the network is parameter-heavy but is fed with a very small amount of data, leads to lower accuracy metrics.

DresNet



DresNet performs poorly when compared to the expected theoretical reasoning. We can see that the model starts to overfit very badly and validation accuracy stagnates after a point. It is not able to improve its accuracy on unseen cases. This is probably again consistent with our belief that by increasing the receptive field of the convolution operation on smaller images, we restrict the trainability of the parameters and the model can learn a very constrained set of variations from the images. The model starts to overfit by the 20th epoch itself, mainly because of this. Even though the training accuracy keeps on increasing steadily, the architecture is not able to model finer unseen variations in the validation set.



Left: Training loss for SkipNet_v2_scheduled and **Right:** for DresNet.

The loss curves for the models reveal that almost all models converge almost uniformly. DresNet is able to minimize training loss very quickly and reaches at a value of about 0.89 at the end of 25 epochs. However, this model overfits by a large margin for reasons discussed above. The best performing model, SkipNet_v2 reaches a value of about 1.58 at the end of 90 epochs. The behavior is expected as training accuracy keeps improving for all models.

Apart from the architectural changes and learning rate scheduling, we also introduce several regularization techniques in our architectures. like Batch-Norm in the convolutional operations, and dropout after the Fully connected layers. It enables the SkipNet model to perform better than the SimpleNet baseline and eventually we are able to achieve am improvement of about 5% on Validation Accuracy @1 and a 4% improvement on Validation Accuracy @ 5.

Overall, we believe that the Mini-Places dataset has an upper bound on the accuracy metric, because of its smaller size. Moreover, some of the images show similar variations and features even when they belong to different classes. This leads to poorer classification. The key takeaway from the architectures is that simpler models are able to perform better and even though regularized models which are very large in size achieve higher training accuracy, these simpler models are able to learn parameters that are able to classify unseen data better.

NOTE: All additional models that we implemented and not used for the final results have also been provided with the code and are present in Additional_Models.py. They can be imported and used directly if required for review.

3. 3 Attention and Adversarial Samples

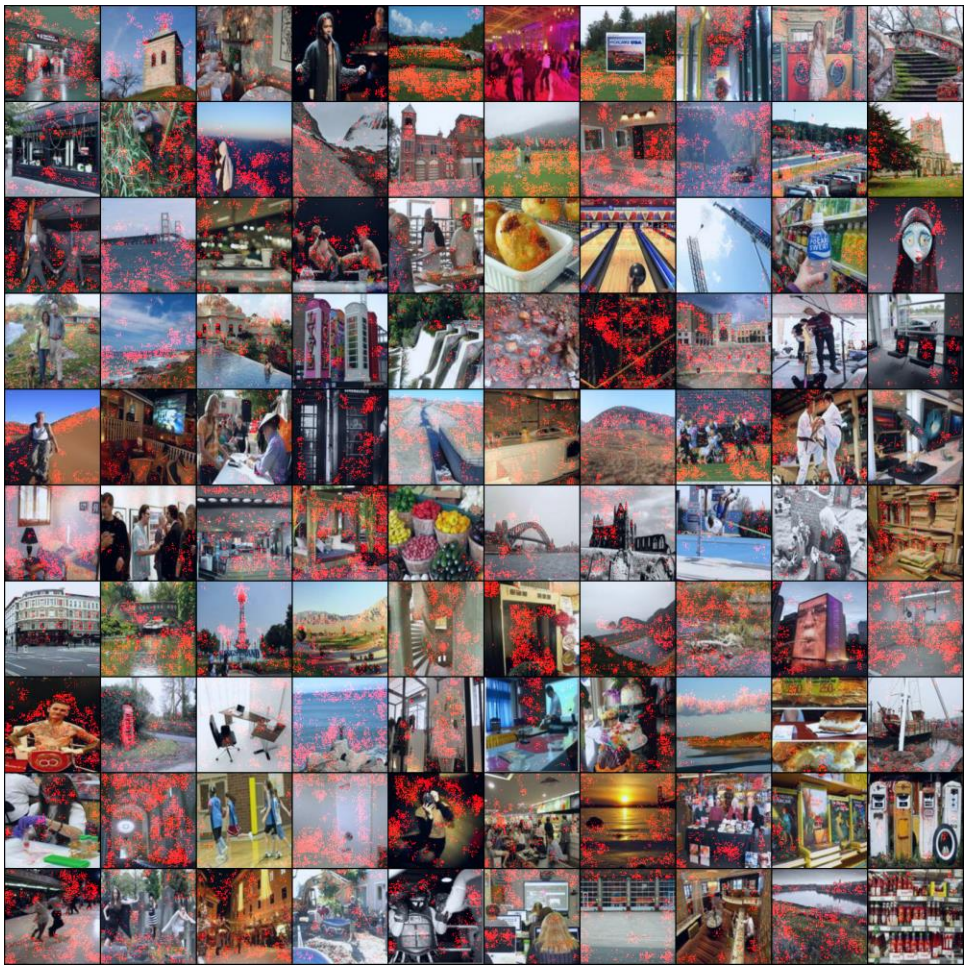
3. 3. 1 Attention / Saliency Maps

Having obtained the CNN classifier, we would like to learn about the spatial support of a class in a given image. Given an image, a class, and a classification ConvNet with the class score function, we would like to rank the pixels of the image based on their influence on the score. The idea of Saliency map is to determine the extent to which each image pixel contributes to the class prediction.

The implementation would require us to minimize the loss of the predicted label and then compute its gradient all the way back up to the image pixels. For we collect the score of the most confident label and minimize the loss of this predicted label. We then compute the gradient of the loss with respect to the input. Based on the observations in [1] we can see that these pixel-wise gradients correspond to the importance of the respective pixel in the given classification decision. A processing step at the end is to find the maximum gradient values across all the three channels of the image.

Implementation using PyTorch

We first compute the prediction scores for each image and track and extract the class score with the maximum value. We then compute the gradients for this with respect to the input using the backward() function provided by PyTorch. We also pass a CUDA-enabled Tensor with all 1’s to the backward() function to minimize the loss. By computing the gradient on the input, we can easily find the saliency map for the image and find the maximum across channels. The helper code in vis_grad_attention helps us visualize the saliency maps for a test set. We used our SkipNet model as described previously.



Visualization of Saliency Maps using Tensorboard

One can observe that the red marked pixels correspond to the most important parts of the image. These regions would intuitively contribute the most in nudging the classifier to decide for its said class. Let's consider the example of the image with the tower and almost all the pixels corresponding to the tower are marked with red.

3. 3. 2 Adversarial Samples

Part I

In the first part of this section, we are required to generate Adversarial Samples of the validation images and observe how our trained model responds to an adversarial attack.

Generating Adversarial Samples

[1] shows how adversarial images can be generated and used for training. They introduce several fast gradient methods to add adversarial examples for training such as the Fast-Gradient-Sign-Method (FGSM) and multiple variations of it. In [4], we see an optimization view of adversarial robustness. Here they allow the adversary to perturb the input first. This gives rise to a saddle point problem where; the inner maximization problem aims to find an adversarial version of a given data point that achieves a high loss. This is precisely the problem of attacking a given neural network. On the other hand, the goal of the outer minimization problem is to find model parameters so that the "adversarial loss" given by the inner attack problem is minimized. This is precisely the problem of training a robust classifier using adversarial training techniques.

PyTorch Implementation

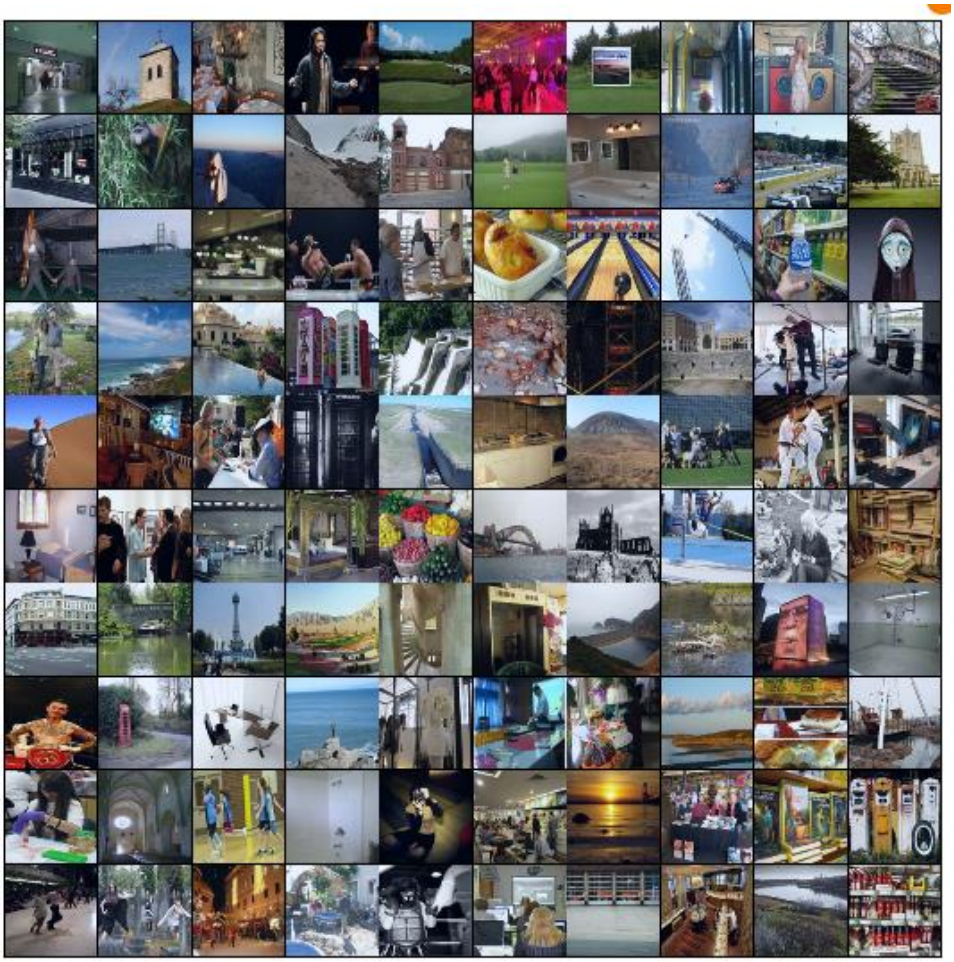
We first compute the prediction scores for each image and track and extract the class score with the minimum value. We then compute the gradients for this with respect to the input using the backward() function provided by PyTorch. We also pass a CUDA-enabled Tensor with all 1's to the backward() function to minimize the loss. We then take several steps of FGSM by replacing the image with its FGSM perturbed counterpart in each step. We also clamp this perturbed image within epsilon-neighborhood of the input image (PyTorch clamp function does not take Tensors in the 'min','max' arguments and required a roundabout way to implement it). For this we used our SkipNet model as described previously.

We show some examples of the adversarial images here

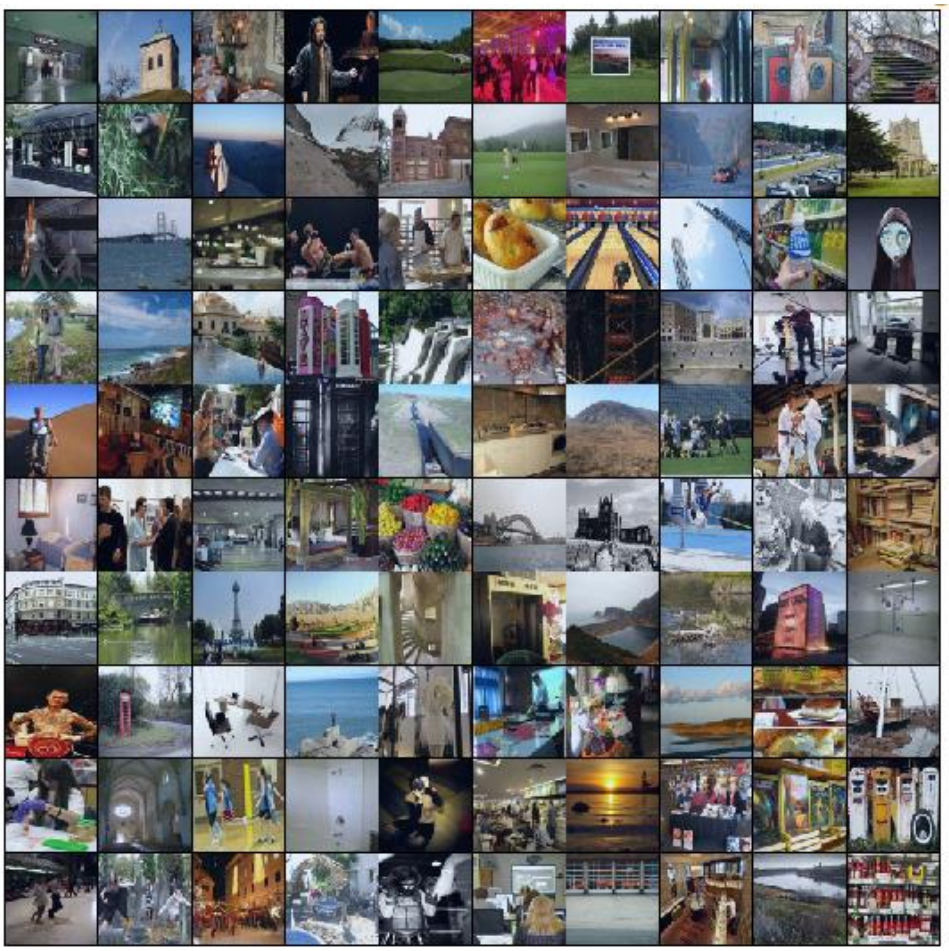
Original Images

Adversarial Images

Original Images:



Adversarial Images:



The adversarial samples look very much alike to the human eye. We can see some slight perturbations around the adversarial images. However, we expect the neural network to be able to classify these images with similar performance rates.

Attacking the Model using Adversarial Examples

We now attack our model "SkipNet_v2" with these adversarial examples. As expected from the literature, the performance of our model drops. The performance numbers are as follows:

Testing Condition	**Top-1 Accuracy (%)**	**Top-5 Accuracy (%)**
Without Adversarial Attacks	51.65	79.35
With Adversarial Attacks	43.05	71.34

3. 3. 2 [Bonus] Adversarial Training

In this section, we try to build a robust classifier and observe its performance on adversarial attacks. For this purpose, we use the SkipNet-v2 architecture to understand the effect of adding adversarial samples to the training procedure.

Experiment:

- Baseline Model: SkipNet-V2 Model trained with training examples for 10 epochs.
- We then attach our PGD Attack code to the forward function in the SimpleNet architecture.
- We set the number of steps for the PGD Attack to 5, since 10 steps would take much longer time to train.
- We then train this model using adversarial training for another 10 epochs and then record the performance of the classifier on the adversarial attacks.
- The performance table is as shown:

	Validation Accuracy - During Training		**Validation Accuracy - Under Adversarial Attack**	
	top-1 accuracy	**top-5 accuracy**	**top-1 accuracy**	**top-5 accuracy**
Training without adversarial samples	29.95	61.87	8.93	26.14
Training with adversarial samples	32.710	64.3	27.12	56.84

We can see from the table, that when the model is not trained with adversarial samples and is attacked by adversarial examples, the performance drops. However, the performance of the model does not degrade much when it was trained with adversarial examples during training.

References

[1] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing

adversarial examples. In ICLR, 2015.

[2] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[3] Szegedy, Christian, et al. "Inception-v4, inception-resnet and the impact of residual connections on learning." *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[4] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep

learning models resistant to adversarial attacks. In ICLR, 2018.

[5] Li, Hao, et al. "Visualizing the loss landscape of neural nets." *Advances in Neural Information Processing Systems*. 2018.

[6] Yu, Fisher et al. “Dilated Residual Networks.” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017): 636-644.