CS838 Homework Assignment 2

Akshata Bhat (abhat6@wisc.edu)
Rohit Sharma (rsharma@cs.wisc.edu)

2 November 2019

3.1 Understanding Convolutions

In this section, we implemented forward and backward propagation in a 2D convolution operation from scratch with a configurable kernel size, stride and padding. The details of the implementation are mentioned below:

Forward Propagation

The forward propagation code is inspired by the slide 49 of Convolutional Neural Networks lecture slides. The basic idea is to unfold the kernel parameters, unfold the input feature maps (with the given stride) and multiply the resulting matrices. The resulting matrix is folded with the given stride to obtain the activations of the corresponding convolution layer.

Before doing this, we first need to calculate the dimensions of the output convolution map so that we can fold the multiplied matrix using dimensions of the input feature map, kernel size, stride and padding. The formula is:

$$(H_{out}, W_{out}) = (\frac{H_{in} + 2P - K}{S} + 1, \frac{W_{in} + 2P - K}{S} + 1)$$

We used the inbuilt PyTorch functions for fold, unfold and matrix operations to perform the forward propagation.

Additionally, we also add the bias term to each output activation map per each channel.

Finally, we save some parameters in the PyTorch context object like the input feature maps and weights for backward propagation.

Backward Propagation

The backward propagation code for computing gradients with respect to parameters and inputs is inspired by slides 51 and 52 of Convolutional Neural Networks lecture slides respectively.

We first load the saved parameters (input feature map and weights) from the PyTorch context object that has been saved during forward propagation.

The compute the gradients with respect to parameters, we unfold the output gradients flowing in (backward direction), unfold and transpose the input feature maps and multiply the resulting matrices. The resulting matrix is folded to obtain the gradients with respect to parameters. To compute the gradients with respect to inputs, we unfold and transpose the weights, unfold the backward gradients and multiply the resulting matrices. The result is

folded to get the input gradients. The implementation for computing gradient for the bias terms was already provided to us.

Again, we used the inbuilt PyTorch functions for fold, unfold and matrix operations to perform the backward propagation.

Finally, we used `test_conv.py` to verify that our custom implementations for forward and backward propagation for the 2D convolution operation are correct.

3.2 Design and Train a Convolutional Neural Network

In this part, we designed and trained a convolutional neural network for scene classification. We used the MiniPlaces dataset [1] for the same which was developed by MIT. There are 120k images and 100 scene categories in this dataset.

All the experiments from here on are performed on a Tesla K80 GPU on a Deep Learning VM deployed in Google Cloud Platform. In each experiment, the training is done using only the training set and the weights were initialized randomly (we did not use for example ImageNet pre-trained weights) except for the last component where we fine-tune pre-trained resnet 18 model for comparison.

Training a Simple Network

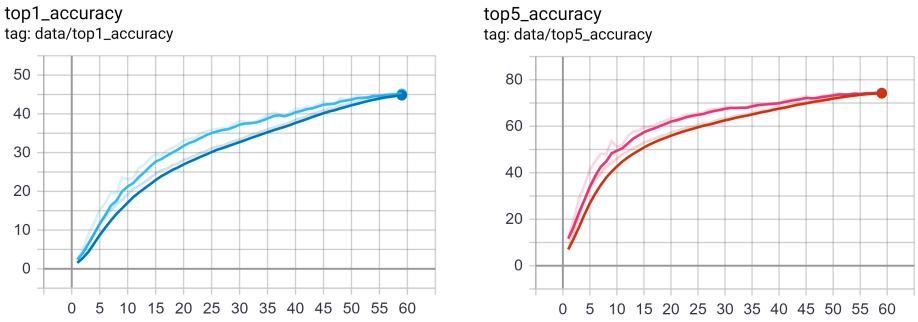
We first trained the provided convolutional neural network architecture called **SimpleNet** for 60 epochs using only the trained set. This model is a simple single branch convolutional neural network with some convolution layers and max pooling layers. It uses ReLU activation for non-linearity. The last layers involve global average pooling and a fully connected layer. The model parameters are initialized according to the method described in [He+15] using a normal distribution.

The model took around 3 hrs and 22 min to train for 60 epochs. Figures 1a and 1b plot top-1 and top-5 train and validation accuracy achieved by *SimpleNet* on the MiniPlaces dataset. As can be seen from the plots, this model doesn't perform very well on the dataset (final top-1 and top-5 validation accuracies are 45.2% and 74.34%). This may be because of the simplicity of the model and the inability to capture advanced features that can help in classifying complex sceneries in the MiniPlaces dataset effectively. The convergence of loss during training is depicted in figure 2. In a below section, we described how we trained another model to improve the accuracy on this dataset.

Some important things to note based on the PyTorch implementation of the SimpleNet model are as follows:

- Cross-Entropy Loss is used for calculating the loss with respect to the target labels.
- Stochastic Gradient Descent(SGD) with *momentum* and *weight decay* is used for optimization.
- Learning Rate
  - The default initial value of learning rate is set to 0.1.
  - During warm-up, the learning rate is adjusted with linear scaling. This is to find a good initial start on the parameter landscape quicker.

[1] https://github.com/CSAILVision/miniplaces



(a) Top-1 accuracy (b) Top-5 accuracy

Figure 1: SimpleNet - 60 epochs

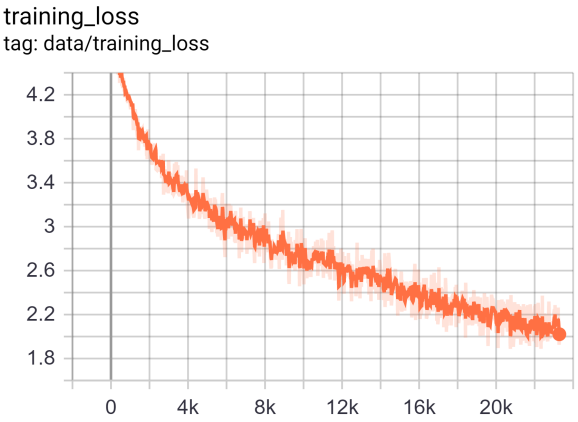


Figure 2: SimpleNet - 60 epochs - Convergence

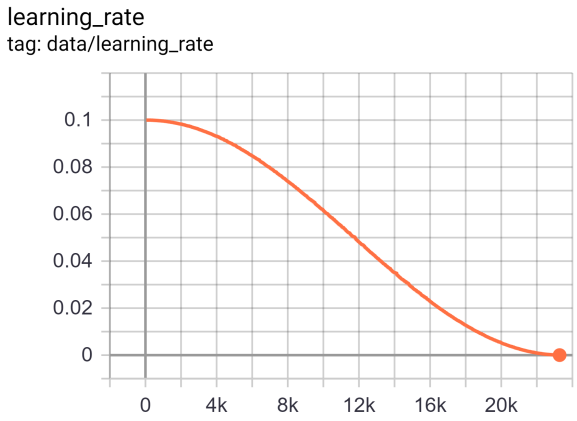


Figure 3: Learning Rate Cosine Decay

  - During training, the learning rate decays over time using cosine-based decay as shown in figure 3. The decay in learning rate helps the model make finer steps as the training progresses.

- For regularization, *weight decay* is used with a default value of 0.0001. Also, to make the model robust, data augmentation is used which transforms the input images using transformations like - scale, random horizontal flip, random color, random rotate, random sized crop and normalization.

Evaluation Metric

In this assignment, we used Top-1 and Top-5 accuracies to evaluate the models. Some other metrics which we could have used for image classification are *average per-class accuracy* and *mean average precision (mAP)*

Compare others with Top-k accuracy TODO []

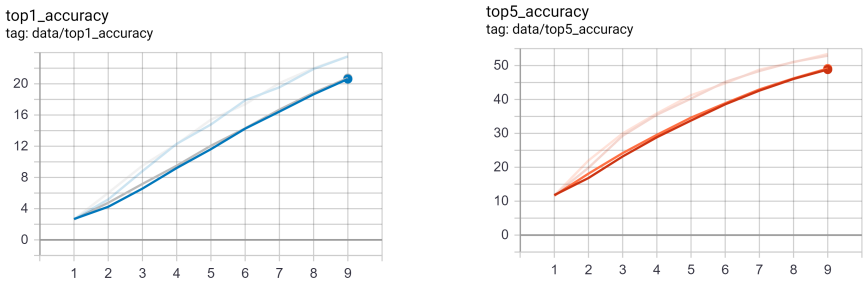
Training with our Custom Convolutions

In this part, we used the custom convolution operation developed in section 3.1 to train the neural network instead of the PyTorch's inbuilt version.

Comparing Custom Convolution with PyTorch's version

For comparison, we trained SimpleNet using both the custom convolution operation and the default convolution in PyTorch for 10 epochs. Here are the results:

- **Training time:**
  - Training with default convolution took around 36 min to train while the custom convolution took 1 hr 57 min.
  - This may be due to the inefficient implementation in the custom convolution. For example, we add the bias term in a loop during forward propagation which is not efficient.



(a) Top-1 accuracy (Gray: Custom, Blue: Default)

(b) Top-5 accuracy (Red: Custom, Orange: Default)

Figure 4: CustomConv vs DefaultConv - 10 epochs - Accuracy

  - Another reason could be due to the implementation of the default convolution operation in low level libraries such as C/C++ which is more efficient.

- **Memory consumption:**
  - The average GPU memory consumption with the default convolution observed was 2398 MiB, whereas for custom convolution, it was 5267 MiB.
  - This is because the default convolution code is written in a very efficient way to exploit the GPU performance well.

- **Convergence:**
  - The top-1 and top-5 validation accuracies achieved using the default convolution operation are 23.82% and 53.82% respectively. See figures 4a and 4b.
  - The top-1 and top-5 validation accuracies achieved using the custom convolution operation are 23.96% and 53.35% respectively. See figure 4a and 4b.
  - During training, the accuracies on validation set using either convolution operation is approximately the same as can be seen from the plots.
  - The convergence plots of loss vs epochs for both the scenarios is shown in figure 5. The blue and green lines almost consistently overlap with each other to the extent that they are not distinguishable.
  - As we can see, using custom convolution doesn't affect the convergence behavior as the difference is not statistically significant. The custom convolution only affects the system performance as it uses an inefficient implementation of the exact same code.

Designing a Custom Network - OurBestNet

To design a custom convolutional neural network, we took inspiration from the Google's Inception v3 model [Sze+15]. The basic idea was to use advanced CNN techniques such as Batch Normalization, Dropout, multi-branch architecture, residual connections, etc., to let the model learn more complex features that can be used to classify more easily in the high-dimensional non-convex space. The Inception module (see figure 6) from the Inception v3 architecture uses Batch Normalization [IS15] and multi-branched layers.

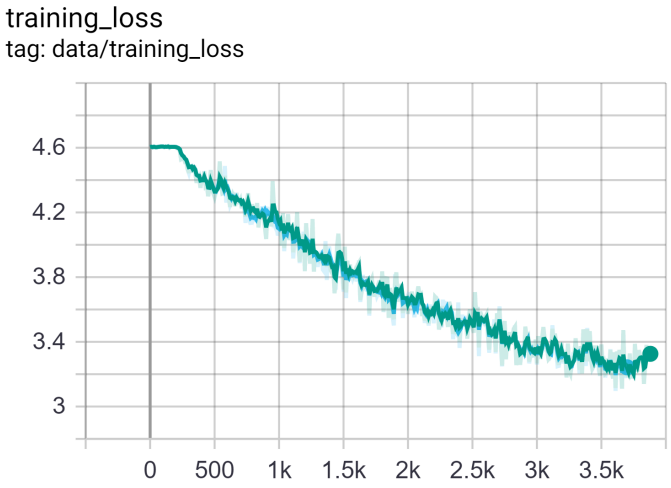


Figure 5: CustomConv vs DefaultConv - 10 epochs - Convergence (Blue: Custom, Green: Default)

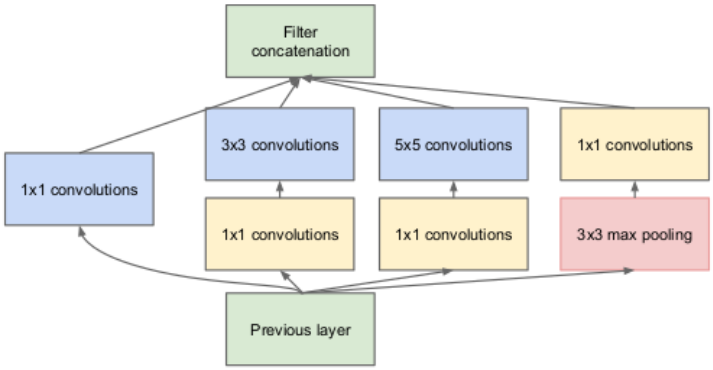


Figure 6: The Inception Module with multiple branches

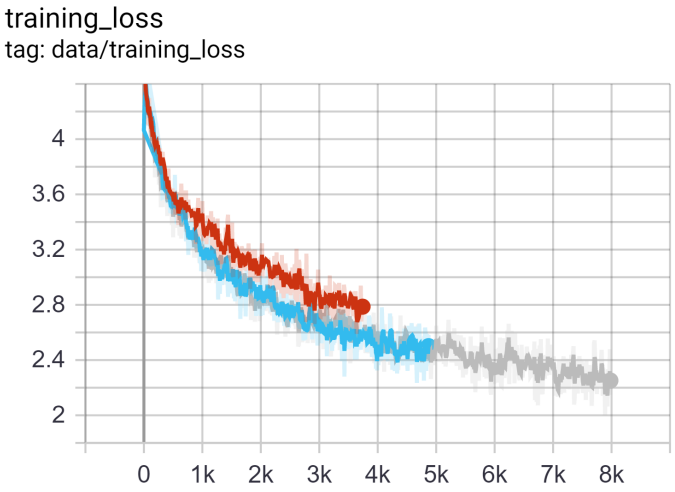
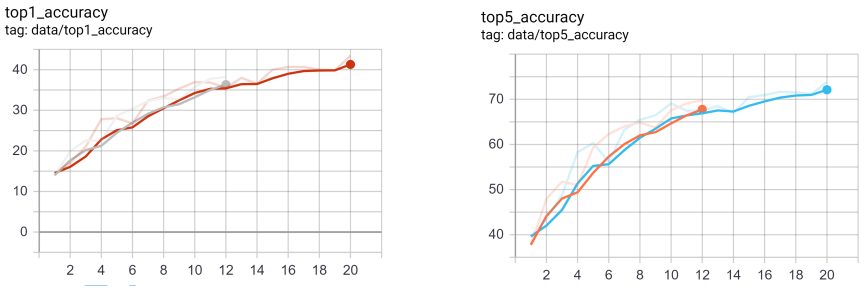


Figure 7: OurBestNet - Convergence (Red: 2 blocks, Gray: 3 blocks, Blue: 4 blocks)



(a) Top-1 accuracy (Gray: 4 blocks, 13 epochs, Orange: 3 blocks, 21 epochs)

(b) Top-5 accuracy (Orange: 4 blocks, 13 epochs, Blue: 3 blocks, 21 epochs)

Figure 8: OurBestNet - Accuracies

Inception network generates scale-invariant reception fields on the input image by using various kernel sized convolution operations in each branch. Additionally, instead of increasing the depth of the network, the wider layers helps in backpropagation through multiple branches more effectively, alleviating the gradient vanishing problem.

We modified the SimpleNet model by adding Inception module and trained on top of that. We experimented the impact of adding 2, 3 and 4 Inception modules on the model accuracy and training time. The model with 3 inception modules seemed to make a good trade-off in the training time and accuracy, hence we chose to implement it in as **OurBestNet**.

Results and Discussion

In this section, a block refers to one Inception module as described above. As it can be seen in figure 7, the convergence is slower in the case of 2 blocks (red curve), where as it is almost similar in the case of 3 blocks and 4 blocks (gray and blue). On the other hand, the time taken to train until 13 epochs with 3 blocks was 2 hrs 3 min, whereas the model with 4 blocks took 2 hrs 33 min to reach 13 epochs.

Because the convergence rate for the models with 3 blocks and 4 blocks is almost the same and because the model with 3 blocks was taking lesser time to train, we decided to kill the

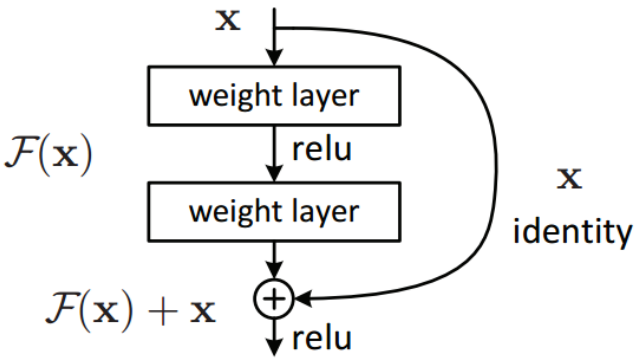


Figure 9: The Resnet Module with Skip Connection

training for the model with 4 blocks and decided to proceed with the model with 3 Inception modules as our best model.

Figures 8a and 8b plot the top-1 and top-5 validation accuracies for both 3 block model (run for 21 epochs in 3 hrs 31 min) and for 4 block model (run for 13 epochs in 2 hrs 33 min). The final top-1 and top-5 validation accuracies achieved by OurBestNet model are 43.47% and 73.83% respectively. In contrast, SimpleNet achieved 33.59% and 63.37% top-1 and top-5 validation accuracies respectively when trained for 21 epochs (see figure 1a and 1b). Therefore, our model performs significantly better than the SimpleNet model.

Comparison of OurBestNet with Resnet 18

Finally, we compare our custom architecture with that of the Resnet18. Resnet 18 uses skip connections which help with the gradient vanishing problem by flowing the gradients directly during backpropagation. The skip connection in the Resnet module is shown in figure 9.

Training Resnet from scratch

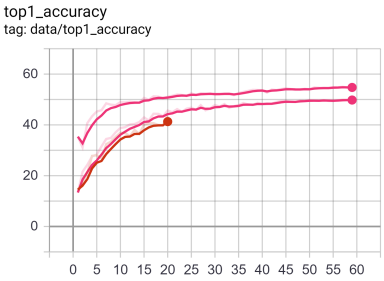
In this section, we initialized the parameters of the Resnet model randomly and trained it from scratch. It took 3 hrs 50 min to train the model for 60 epochs.

Fine-tuning pre-trained Resnet

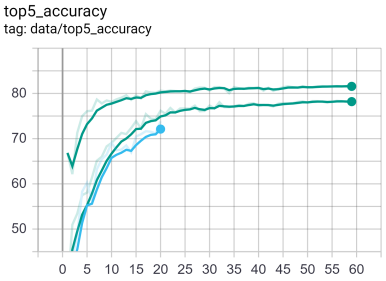
Here, we initialized the model parameters for Resnet 18 that have been pre-trained on the ImageNet classification task. We then fine-tuned the model by training on the MiniPlaces dataset. It took 3 hrs 54 min to train the model for 60 epochs.

Results

The convergence for fine-tuned ResNet 18 is better than that of random initialization (see figure 11). The initial convergence of OurBestNet is similar to randomly initialized ResNet18, but with time, ResNet converges better. The top-1 and top-5 accuracies are better for fine-tuned ResNet 18 compared to the random one (see figures 10a and 10b). The accuracies for OurBestNet and random ResNet are almost comparable with ResNet performing slightly



(a) Top-1 accuracy (Pink: ResNet FT, Orange: ResNet NonFT, Red: OurBestNet)



(b) Top-5 accuracy (Green1: ResNet FT, Green2: ResNet NonFT, Blue: OurBestNet)

Figure 10: Resnet 18 vs OurBestNet - Accuracies

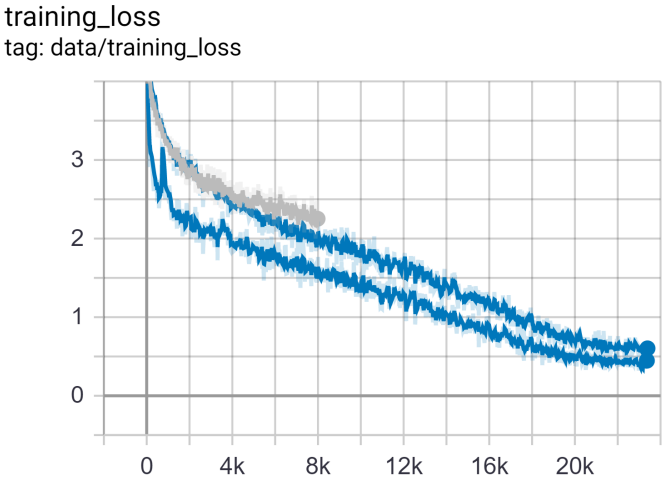


Figure 11: Resnet 18 vs OurBestNet - Convergence (Blue1: ResNet NonFT, Blue2: ResNet FT, Gray: OurBestNet)

better. We account the better performance of ResNet models to the more advanced skip connections which leads to faster convergence.

3.3 Attention and Adversarial Samples

In this section, we analyse the interpretability and robustness of convolutional neural networks that we developed in the previous sections.

For analysing the interpretability, we generate *saliency maps* as described in the below subsection. For analysing the robustness, we generate *adversarial samples* to fool our neural networks. We then perform *adversarial training* to make the model robust to such attacks as described in the sections below.

Saliency Maps

Saliency maps or *Attention maps* help us interpret the reason for a neural network's prediction by indicating which pixels are most important for the prediction.

To compute the *saliency map* for an input image, we minimize the loss of the predicted label and compute the gradient of the loss with respect to the input. The magnitude of each pixel's gradient indicates the importance of that pixel for prediction. We take the maximum value of the gradient across each input image channel. This technique is described in [SVZ14].

In PyTorch, we first set the `requires_grad` attribute to `True` for the input tensor so that gradient values are computed for input image. Then we compute the loss of the most confident prediction and backward propagate on the loss using `loss.backward()`. We then produce the *saliency map* which is the maximum value of input gradient across all the channels for each pixel.

The results in figure 12 show saliency maps drawn on top of each input image. The red annotations on each image correspond to the regions where the model focuses the most to determine the class labels.

Adversarial Samples Generation

After getting a sense of which regions in the image are most important for the model predictions based on *saliency maps*, we then create adversarial samples to confuse the model. We perturb the input images such that the loss of a wrong prediction is minimized by computing gradient of the loss computed with target label as the least confident label. Specifically, we implemented the Projected Gradient Descent algorithm under l_∞ norm to modify the input image as described in [Mad+17].

In our implementation, we first clone the input tensor and set the `requires_grad` attribute to `True` for the clone. We then perform several fast gradient sign method steps. In each step, we compute the loss of least confident label and backward propagate to find the gradient of the input image. We then perturb the input image by a small step size, η in the direction of the sign of this gradient (gradient ascent). Finally, we clamp each pixel value in the ϵ-neighborhood of the original input image (one copy of the clone). We implemented this using PyTorch's *max* and *min* functions which selects a maximum or a minimum value for each data in the tensors provided. For more details, please see our code.

We now demonstrate some of the adversarial samples generated using this approach in figures 13a and 13b. On comparing with the original image, we can barely see any differences.

Figure 12: Saliency/Attention Maps

(a) Adversarially Perturbed Images (b) Original Images

Figure 13: Adversarial Samples vs Original Images

Since the images look almost the same, the model is expected to still perform well. But as mentioned in the next paragraph, the model fails to classify these adversarial examples correctly.

We then attack our model by generating adversarial samples on the validation set. This leads to a reduction of accuracy from [] to [] indicating that our adversarial attack was successful.

Adversarial Training

To alleviate the problem of performance reduction when a model is attacked by an adversary by perturbing the input images, we perform adversarial training. Specifically, we make the model robust to adversarial samples by feeding these samples as input data during training as described in [GSS14; Mad+17].

In PyTorch, we modify the `forward()` function of the `SimpleNet` model to generate adversarial samples by perturbing the input image by calling the function developed in the previous section. This makes the training much slower. To improve the training time, we only [@Akshata confirm this:] train for xx epochs and perform only xx FGSM steps to generate adversarial samples.

The results of adversarial training are shown in the table []. As we can see, comparing to the table [], the model is more robust to adversarial attacks.

Team Members and Contribution

In general, both the team members contributed equally by collaborating in all of the sections while studying the reference papers and implementing. On a high level, the individual contributions can be broken down as follows:

Name	**Contributions**
Rohit Kumar Sharma	• 3.1 Backward Propagation • 3.2 Training Simple Network, Comparison with Custom Convolutions, Comparison with Resnet18 • 3.3 Saliency Maps, Adversarial Samples Generation • Writeup
Akshata Bhat	• 3.1 Forward Propagation • 3.2 Contrasting Evaluation Metrics, Designing Custom Network • 3.3 Adversarial Training • Writeup

References

[GSS14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2014. arXiv: `1412.6572 [stat.ML]`.

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[Mad+17] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. *Towards Deep Learning Models Resistant to Adversarial Attacks*. 2017. arXiv: 1706.06083 [stat.ML].

[SVZ14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*. 2014. URL: http://arxiv.org/abs/1312.6034.

[Sze+15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.