CS 838 Assignment2

Bin Li, Jimmy Chang, Zhiyu Ji

November 4th, 2019

1 Understand Convolutions

Instead of just using fold or unfold function provied by Pytorch, We developed these function manually by ourselves. We used the provided function to do forward pass and after that we developed the fold and unfold function in the backward pass. The basic idea for implementing the convolution operation is to transfer feature map and filters into matrix form and do matrix multiplication,after that,transfer the result back.

1.1 Forward Pass

The basic idea to implement the convolution operation is to transfer the feature map into matrix form according to the filter size and the stride.To make it clearly, the stride in figure below is a bit large.

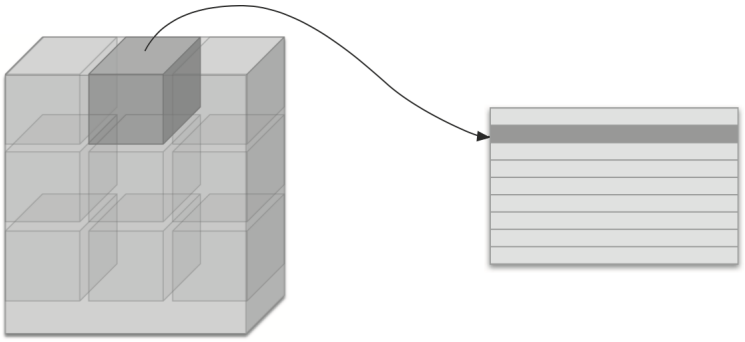

Figure 1: Unfold input data

The idea is the same in the filters,we need to span the filters into matrix form.

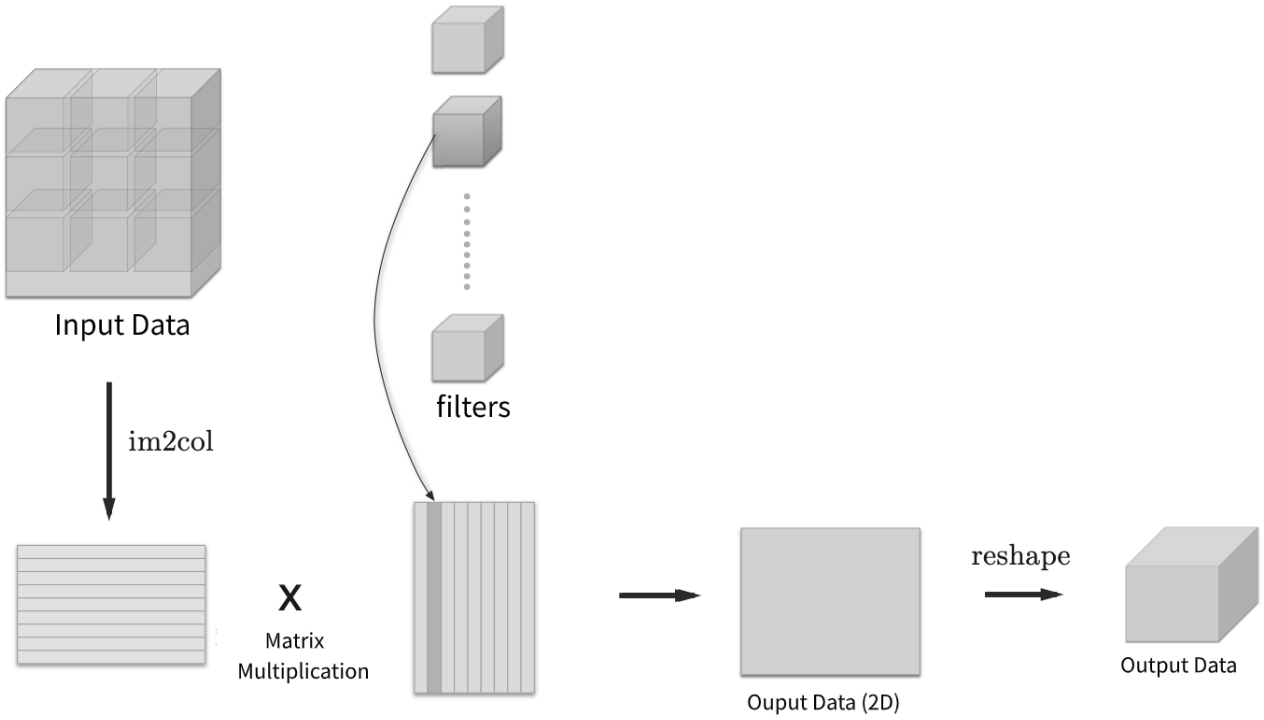

Figure 2: Convolution operation

The process is quite clearly in the picture, filters will be spanned into matrix by column,which means that each column in the matrix represent a filter. And the input data will be spanned by row.

1.2 Backward Pass

We implement the fold and unfold in backward pass by ourselves.There are two gradients need to calculate,gradient of weight and gradient of input features.
As for gradient of weight,we denote feature map as X, the shape of X is [N*OH*OW,C_{in}*K*K],gradient of output's shape is [N,C_{out},OH,OW],So gradient of weight is as follows:

$$dW = X^T dY$$

As for gradient of input, we denote kernek matrix as M, the shape of M is [C_in*K*K,C_{out}], We have the formula as follows:

$$dX^0 = dY * W^T$$

And then we have the formula:

$$dX = 0$$

$$dX[:,:,y:y_{max},x:x_{max}] += \sum dX^0[:,y,x,:,:,:]$$

After finishing that, we returned the gradients.

2 Design and Train a Convolutional Neural Network

2.1 Train SimpleNet

The SimpleNet network is trained using the following command:
python ./main.py ../data --epochs=60
The learning rate, top 5/1 training loss and validation loss is shown in **Fig. 3**

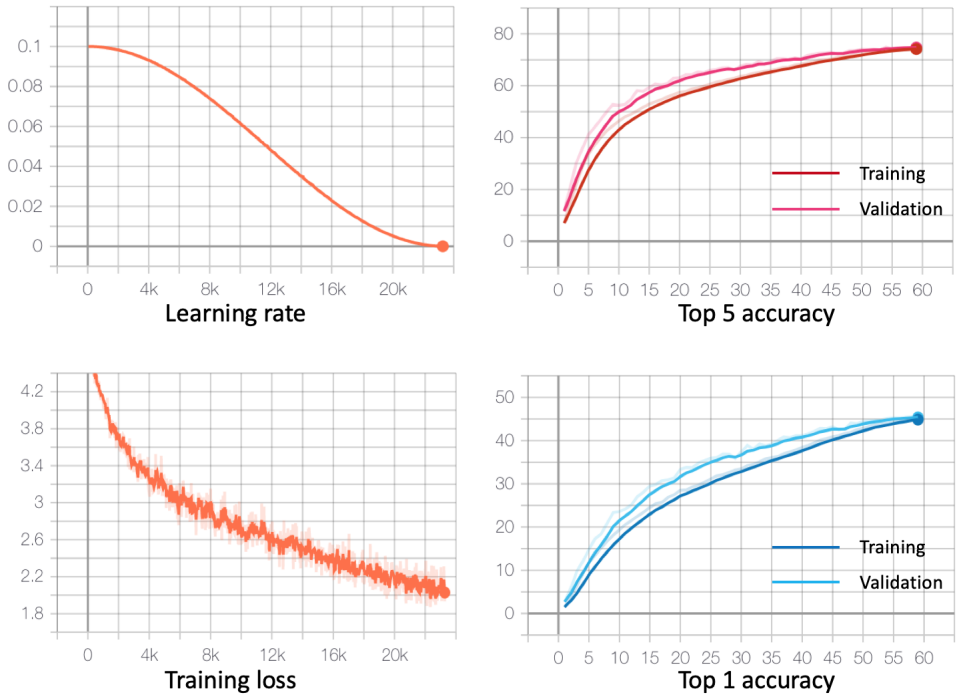

Figure 3: Tensorboard plot of SimpleNet

Top 5 validation accuracy: 74.74%; top 1 validation accuracy: 45.37%

2.2 Understand the training process of SimpleNet

2.2.1 Training process overview

- Set device: use GPU or CPU
- Define loss function

- Define model
- Transfer loss function and model to GPU if available
- Construct transform pipeline for dataloader
- Construct dataloader, specify mini-batch size
- Load pre-trained weight to model if available
- Set model to training mode (default)
- Feed mini-batch to model and forward-prop
- Obtain model output and compute loss
- Compute gradient and back-prop, do optimizer step
- Compute validation accuracy

2.2.2 Loss function

Cross Entropy (CE) loss in Pytorch combines LogSoftmax operation and Negative Log Likelihood (NLL) loss measurement.

$$loss(x, class) = -log(\frac{exp(x[class])}{\sum_j^N exp(x[j])}) = -x[class] + log(\sum_j^N exp(x[j])) \tag{1}$$

Where x is a vector containing the probabilities of each class in the prediction and $class$ is the true class label, N is the total number of classes. We can consider the Softmax operation produces a normalize probability vector where each element corresponds to the predicted probability of the input belonging to that class. Combined with NLL, this loss function measures the distance between two distributions and it is widely used in classification problems.

2.2.3 Optimization method

The optimization method is stochastic gradient descent (SGD) with momentum and weight decay. The weights updating can be described as:

$$v = \rho v + lr\nabla(loss + \lambda||w||_{\ell_2}) \tag{2}$$

$$w^{i+1} = w^i - v \tag{3}$$

Where w^i denotes the weights of the model in ith iteration and $w^(i+1)$ denotes the updated weights. ρ is the momentum, v is the velocity, λ is the ℓ_2 penalty coefficient, lr is the learning rate.

2.2.4 Learning rate scheduling

If the stage is set to "warmup" in the options, then the learning rate will be computed as:

$$lr_c = lr_0 \frac{n_e n_b + 1}{n_t n_b} \tag{4}$$

if the option is not set, then the learning rate will be computed as:

$$lr_c = \frac{1}{2} lr_0 (1 + cos \frac{n_e n_b + i}{n_t n_b} \pi) \tag{5}$$

Tweaking the learning as the training progresses using this cosine annealing fashion has been demonstrated to facilitate the convergence of deep networks training.

2.2.5 Evaluation metric

Validation set top-k accuracy is reported for $k = 1$ and $k = 5$. Top-k accuracy means that the result is considered to be correct if the correct class falls in the top k probabilities of prediction. Top-k allows tolerance to similar classes that are hard to distinguish even for human beings and such tolerance is adjustable through the parameter k.
Average per-class accuracy computes the accuracy of each class and then produces the averaged value across each class. This is useful when the dataset is imbalanced such that the number of instances in each class is very different. For example, for a disease screening task where the natural occurrence rate is 1%, a fully negative classifier will have 99% accuracy but 50%.
Mean average precision (mAP) is useful for information retrieval our bounding box evaluation for object detection. In information retrieval the average precision is computed as:

$$AP_k = \frac{1}{TP} \sum_{i}^{k} \frac{tp}{i} \tag{6}$$

Where k is the number of instances in each retrieval, TP is the total number of true instances in the sorted result, tp is the index of true instance detected. mAP averages this value of multiple retrieval.
In object detection average precision is computed using Intersection over Union (IoU):

$$IoU = \frac{Area\,of\,overlap}{Area\,of\,union} \tag{7}$$

Where $Area\,of\,overlap$ stands for the area of overlap between detected bounding box and label bounding box, $Areaofunion$ stands for the union of detected bounding box and label bounding box. The IoU would be used to determine if a predicted bounding box is TP, FP or FN. The TN is not evaluated as each image is assumed to have an object in it. Then we can obtain precision using:

$$p = \frac{TP}{TP + FP} \tag{8}$$

The mAP for object detection is the average of the AP calculated for all the classes.

2.3 Train with your own convolutions

The model is trained with the default conv2d and our conv2d for 10 epochs, training curves are shown in **Fig. 4** and **Fig. 5**

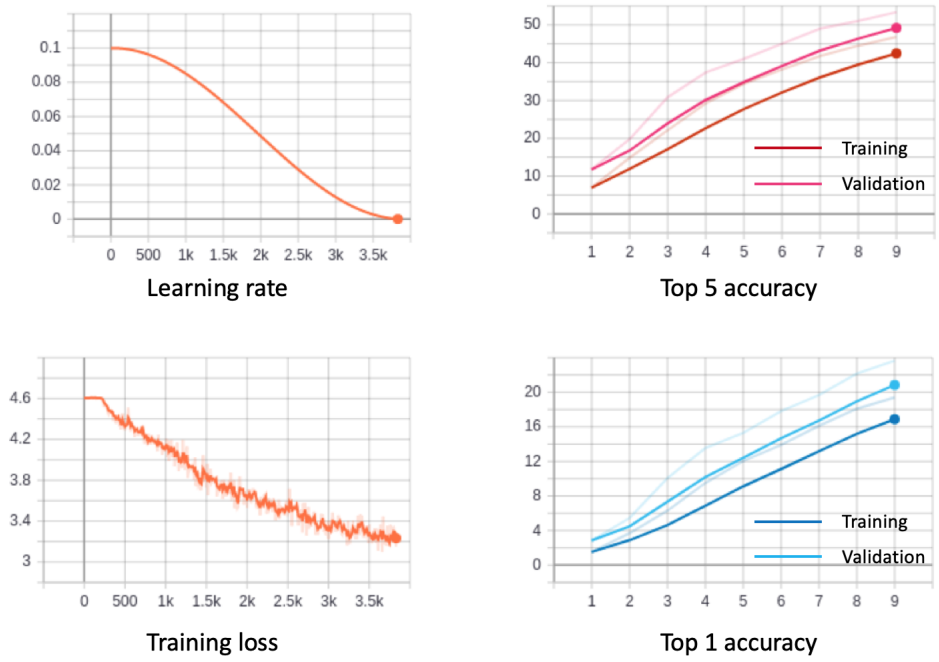

Figure 4: Tensorboard plot of SimpleNet trained with custom conv2d

Figure 5: Tensorboard plot of SimpleNet trained with default conv2d

The custom implementation of conv2d is significantly slower compared to the default conv2d and also uses a lot more GPU memory. This could be due to that default conv2d is written in parallel broadcasting fashion and uses low-level language interface that communicates directly with GPU and has better data buffering strategies. The comparison is summarized in **Table 2.3**.

Convolution	batch time	GPU usage	accuracy top1/5
Default conv2d	1.58	7.5	23.79/54.09%
Custom conv2d	0.123	3.1	23.87/53.58%

Table 1: Summary of training details of default conv2d and custom conv2d

2.4 Our network

We name our network as multi-scale net (MSN), the model is coded in model.py and need to be imported to student_code.py. The network is trained using the following command:
python ./main.py ../data --epochs=60
The learning rate, top 5/1 training loss and validation loss is shown in **Fig. 6**

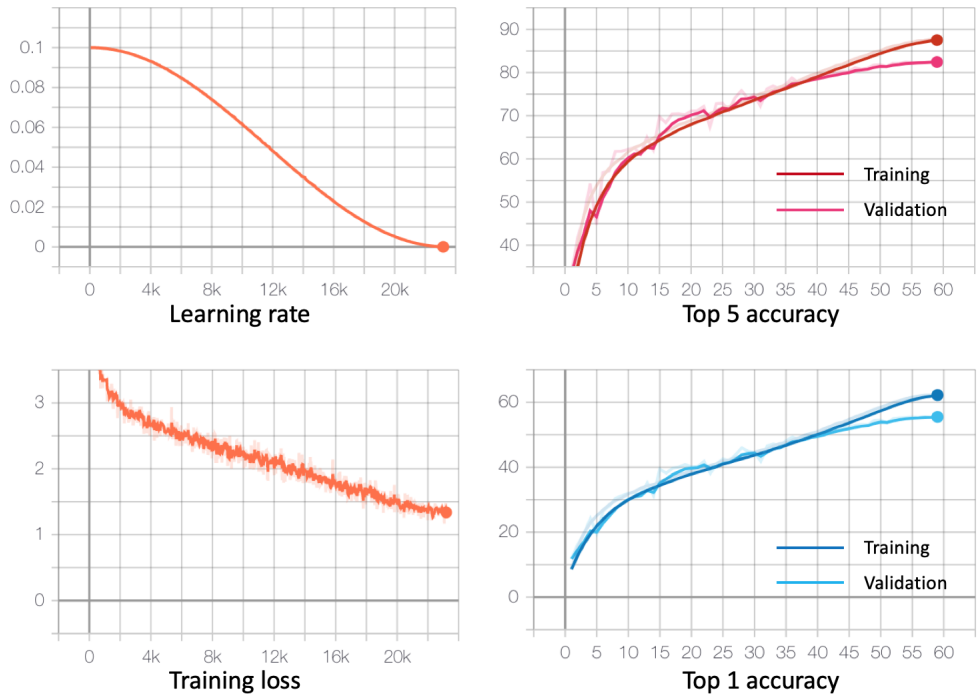

Figure 6: Tensorboard plot of MSN

Top 5 validation accuracy: 82.45%; top 1 validation accuracy: 55.48%
Our network consists of a combination of convolutions with 4 scales (*i.e.* $2\times, 4\times, 8\times, 16\times$ downsampling). Downsampling is done by convolution with stride of 2. Residuals from the first downsampling are passed to next levels. Outputs features of 4 hierarchical scales are concatenated and averagely pooled, resulting in 2048 features which form the inputs of a fully connected layer with 100 output classes. The network is designed in this way such that multi-scale features can be learned from the images. This makes use of a strong prior that image features are hierarchical in scale and spatially connected. Our network significantly outperforms the baseline network but has more parameters and presents sign of over-fitting at the late stage of training.

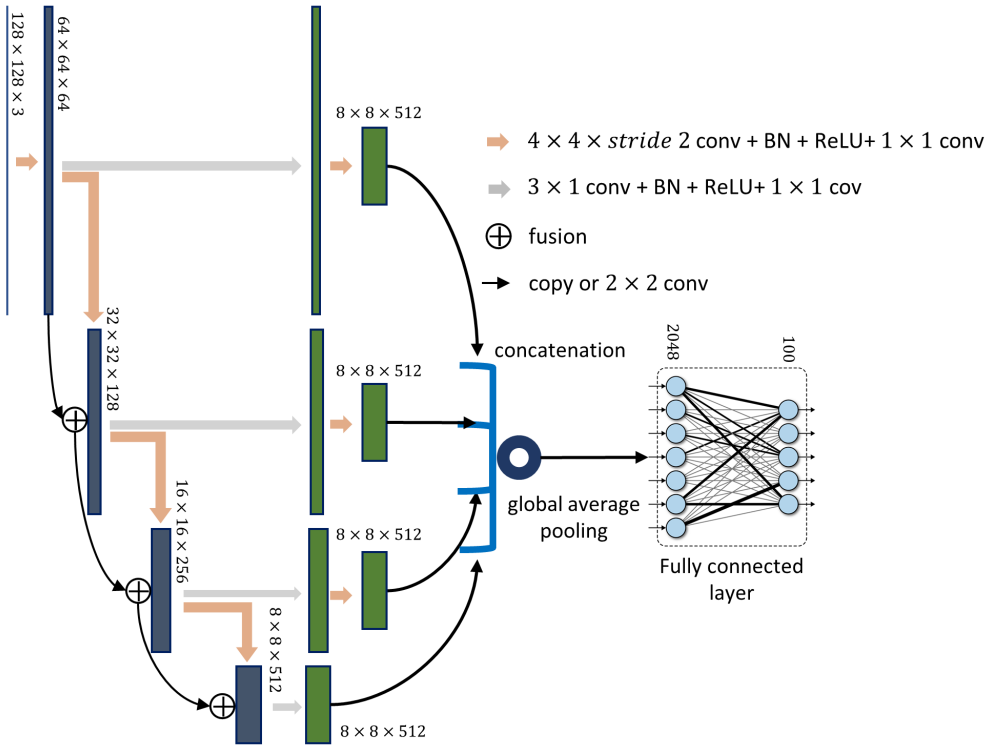

Figure 7: Architecture of multi-scale net

2.5 Fine-Tune a Pre-trained Model

The ResNet18 model is tuned using the following command:
python ./main.py ../data --epochs=60 --use-resnet18
The learning rate, top 5/1 training loss and validation loss is shown in **Fig. 8**

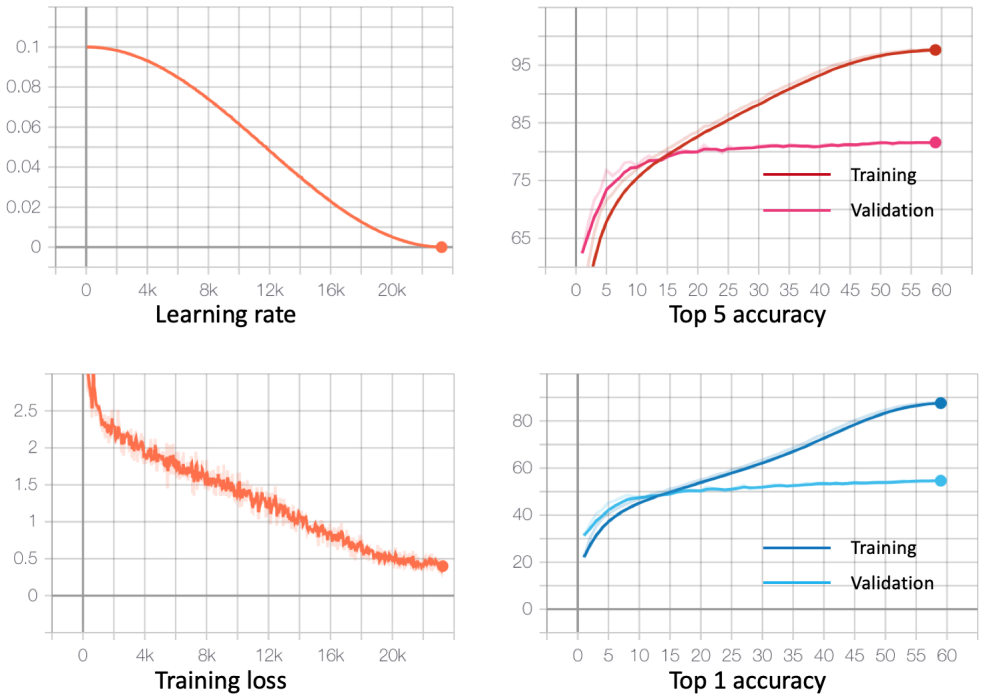

Figure 8: Tensorboard plot of fine tuning ResNet18

Top 5 validation accuracy: 81.61%; top 1 validation accuracy: 54.62% The pre-trained ResNet18 model converges faster than both the baseline network and MSN. This is patially due to the fact that common low level features are already learned in the pre-trained model and the model lies at a point in the optimization surface that are close to the new minima. However, ResNet18 presents more over-fitting on the training set, as the training accuracy continuously grows but the validation accuracy does not improve much. This could be due to the lack of sufficient training data and the overly high capacity of the model.

3 Attention and Adversarial Samples

3.1 Attention/Saliency Map

In order to summarize the reasons for neural network behavior or produce insights about the causes of their decisions, we need to know which parts of the data actually have influence on the network output. Here we implemented the saliency map algorithm to highlight input regions that cause the most changes in the output by computing the gradient of output with respect to input image. All the positive gradient values tell us that a small change to that pixel will increase the output value and vice versa. Thus, visualizing these gradients should provide some intuition of attention. Note that we used our multi-scale net (MSN) as described previously

for this experiment.

Implementation. For every image, we first feed forward the image through the network and then obtain its output prediction values. We compute the gradient for it with respect to the input image using *backward* function in PyTorch. Then, we take the absolute value of these gradients and find the maximum value across channels. By leveraging the helper function *vis_grad_attention* to visualize the saliency maps.

Figure 9: Saliency maps of the validation data using our MSN model.

3.2 Adversarial Samples

Deep neural networks have been shown to be easily fooled into misclassifying a image to any target class by making imperceptible changes to the pixels, which is called *adversarial samples*. Many algorithms have been proposed to generate this kind of adversarial samples, such as Project Gradient Descent (PGD) attack. The PGD attack performs arbitrary steps of gradient descent of a fixed size, while always staying within the range of ϵ (under L_∞) from the input.

Implementation. First, we pass an image through the network and then obtain its output prediction values. Secondly, compute the gradient for it with respect to the input image using *backward* function in PyTorch. Thirdly, take the sign of the gradient, and based on the sign, we tweak the image by a small step size set at **0.01**. Repeat the above three steps for **10** times and get the adversarial sample. For this we used the MSN model as described previously. The generated adversarial samples and its corresponding original images are shown in Figure 10. The performance evaluation of our MSN model over adversarial samples is summarized in Table 2.

Figure 10: The left figure is the original image and the right one is the adversarial image.

Table 2: The performance evaluation over adversarial samples on our MSN model.

	Top-1 accuracy	Top-5 accuracy
Original Images	55.48 %	82.45 %
Adversarial Images	34.60 %	59.23 %
Difference	-20.88 %	-23.22 %

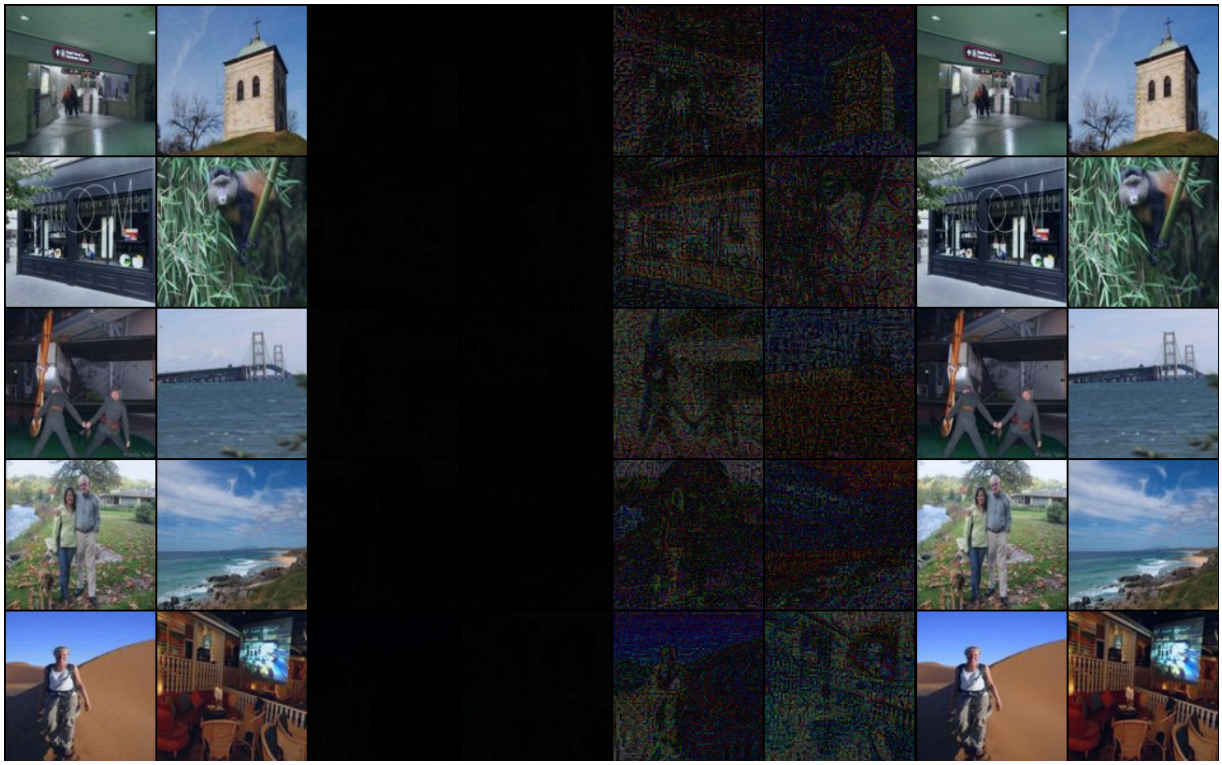

Figure 11: The figures from left to right are (i) the original images, (ii) adversarial noise, (iii) 10x adversarial noise, and (iv) the generated adversarial images. Note that we multiply adversarial noise by 10 to show more details in (iii).

We take a closer look at the difference between the original and adversarial images as shown in Figure 11. We found that the adversarial noise is really hard to be perceived for humans but that is enough to confuse our MSN model and result in significant classification performance drops from 55.48% to 34.60% in top-1 accuracy and from 82.45% to 59.23% in top-5 accuracy.

Table 3: Comparsion between SimpleNet and AdvSimpleNet (with adversarial training).

	Top-1 Accuracy			Top-5 Accuracy		
Image Source	Original	Adversarial	Difference	Original	Adversarial	Difference
SimpleNet	**47.41** %	32.16 %	-15.35 %	**77.23** %	59.45 %	-17.78 %
AdvSimpleNet	44.98 %	**43.58** %	**-1.40** %	74.17 %	**71.70** %	**-2.47** %

3.3 Adversarial Training

Adversarial examples are generated by perturbing inputs to fool machine learning models. To increase model robustness against adversarial attacks, researchers propose adversarial training that injects adversarial examples into training data and learns a more robust classifier.

Implementation. We create a new model class *AdvSimpleNet*, which inherits from the given class *SimpleNet*, to train the network on adversarial examples generated by the PGD attack. We overwrite the *forward* function by incorporating the PGD attack into the forward pass so that the network will only learn from adversarial samples. However, adding the PGD attack in the loop would slow down the training process, and thus we reduce the number of steps from 10 to 5 for speedup.

As demonstrated in Table 3, we observed that the AdvSimpleNet model had much better robustness against adversarial samples and, at the same time, maintained comparable performance over the original samples, where the performance difference was less than 1.4/2.47% in top-1/5 accuracy. On the other hand, the SimpleNet model, without adversarial training, had a significant performance drop in both top-1 and top-5 accuracy while evaluating on adversarial samples. Hence, we believe adversarial training is an effective way to fight against adversarial samples. However, due to training on adversarial samples, the AdvSimpleNet's performance over the original images slightly went down.

4 Team Members and Contributions

Name	Contributions
Bin Li	• Section 2: Design and Train a Convolutional Neural Network - Conducted extensive experiments on training SimpleNet - Designed and trained our multi-scale net (MSN) - Fine-tune a ResNet model for comparison • Write up Section 2 • Help trouble shooting for custom_conv in Section 1
Jimmy Chang	• Section 3: Attention and Adversarial Samples - Implemented *GradAttention* and *PGDAttack* - Visualized saliency maps - Generated and analyzed adversarial samples - Conducted experiments on adversarial training • Write up Section 3
Zhiyu Ji	• Section 1:Understand Convolution -Implemented forward pass -Implemented backward pass • Write up Section 1