Assignment 2

Arshad Kazi, Bhuyashi Deka, Sadman Sakib

October 26, 2024

1 Convolutional Neural Networks

1.1 Understand Convolutions

For the implementation of custom convolutions we have unfolded the weights and feature map in order to perform convolutions in the form of matrix multiplication. The output is then folded back before returning. This allows us to leverage optimized Basic Linear Algebra Subprograms (BLAS) libraries, which efficiently use CPU caches to improve data locality and reduce memory bandwidth bottlenecks and massive parallel architecture of GPUs.

For number of images in a batch (N), input channel (Ci), output channel (Co), height of image (H), width of image (W), kernel size (K), no of patches (P), we transform the dimensions as follows to compute matrix multiplication.

Forward-pass
Initial input dimension (X) = [N, Ci, H, W]
Initial weight dimension (W) = [Co, Ci, K, K]
Unfolded input dimension (X_u) = [N, Ci*H*W, P]
Unfolded weight dimension (W_u) = [Co, Ci*H*W]
Output dimension (Y) = [N, Co, Ho, Wo]

$$Y_u = W_u \times X_u$$

Backward-pass

$$dW_u = dY_u \times {X_u}^T$$
$$dX_u = {W_u}^T \times dY_u$$

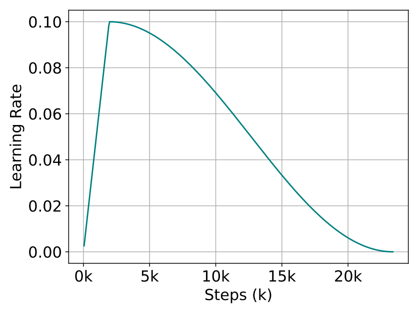
Comparison with PyTorch's Conv2d
When comparing with PyTorch's implementation both forward prop and backward prop tests pass, that is the results perfectly match. However, the PyTorch implementation is faster since it has been written in C++. Also, for the backward prop, we are using reshape function which adds to the time.

1.2 Design and Training Convolutional Neural Networks

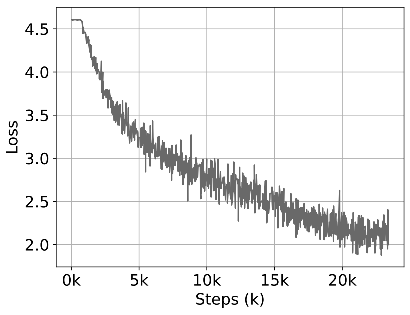
1.2.1 A Simple Convolutional Network

The `SimpleNet` implements a CNN with 3 convolution blocks.

During training, we have initial 5 epochs of warm-up during which we gradually increase the learning rate. This is because the network is initialized with random weights from a uniform distribution and would result in large gradients. Small learning rate helps to learn more stable gradients before learning rate can be increased. This behavior is evident in the learning rate plot. We are using the Cross-entropy Loss for the training. After 60 epochs the loss reduces from 4.6 to 2.12 (figure 1). This means that the model is predicting the correct class with less confidence and training for more epochs should help.



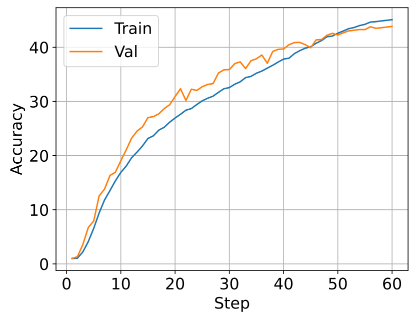
(a) Learning rate curve



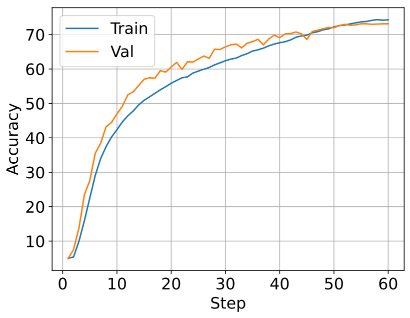
(b) Loss trend

Figure 1: Learning rate and loss for inbuilt convolution

Using PyTorch's convolution, the network can achieve 43.9% top-1 accuracy and 73.1% top-5 accuracy on the test set in 60 epochs (figure 2). As the network has to classify 100 classes, the model is performing quite good.



(a) top-1 accuracy trend



(b) top-5 accuracy trend

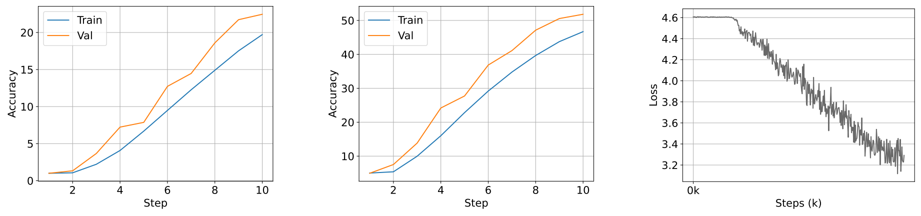
Figure 2: Accuracy of SimpleNet model with PyTorch Conv2d convolution

1.2.2 Train with Your Own Convolutions

Table 1 below compares the test accuracy after 10 epochs of `SimpleNet` model with our implementation of convolution (`CustomConv`) and PyTorch's implementation (`Conv2d`). Figure 3 shows the accuracy and loss plots of `SimpleNet` model with custom convolution.

	top-1 accuracy	top-5 accuracy
CustomConv	19.7%	46.7%
Conv2d	16.9%	46.9%

Table 1: Comparison of accuracy after 10 epochs



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

Figure 3: Accuracy and Loss for SimpleNet model with custom convolution

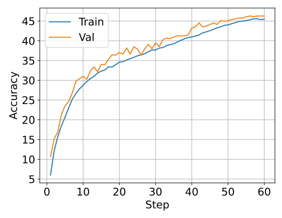
1.2.3 Design Your Own Convolutional Network

We designed two convolution networks by improving on the `SimpleNet` model.

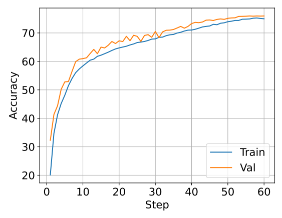
In `CustomNet1` model, we added batch normalization layer after each convolution layer in `SimpleNet` model. So, this model also has three convolution blocks, where first block has one convolution layer, and second and third blocks each has three convolution layers. The number of feature maps are- 64 (1st and 2nd block), 128 (3rd block) and 512 (3rd block output).

In `CustomNet2` model, we made the model deeper than `CustomNet1` model. We kept only one max pooling layer at the very beginning. This model has five convolution blocks, where first block has one convolution layer and all other blocks have three convolution layers. In all blocks other than first block, the middle layer has stride 2 to downsample. The number of feature maps are increased gradually- 64 (1st and 2nd block), 128 (3rd block), 256 (4th block) and 512 (5th block).

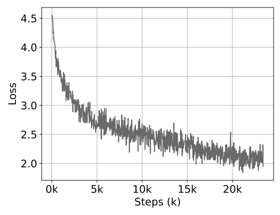
After 60 epochs, `CustomNet1` model achieves ˜46% top-1 validation accuracy, about 2% improvement over `SimpleNet` model. After 60 epochs, `CustomNet2` model achieves ˜49% top-1 validation accuracy, about 5% improvement over `SimpleNet` model. Figure 4 and 5 show the accuracy and loss of `CustomNet` and `CustomNet2` models.



(a) Top-1 accuracy

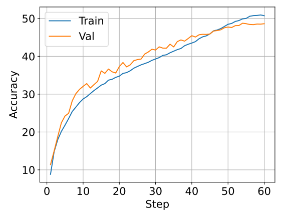


(b) Top-5 accuracy

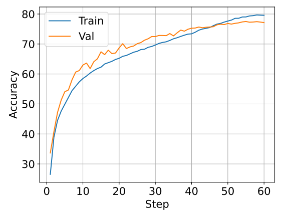


(c) Loss

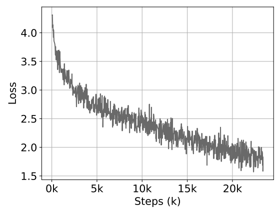
Figure 4: Accuracy and Loss for CustomNet model



(a) Top-1 accuracy



(b) Top-5 accuracy

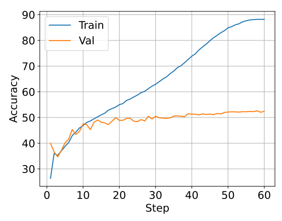


(c) Loss

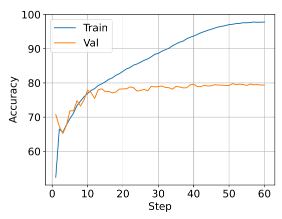
Figure 5: Accuracy and Loss for CustomNet2 model

1.2.4 Fine-Tune a Pre-trained Model

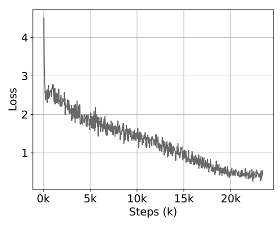
We fine-tuned a pre-trained ResNet18 model on the MiniPlaces dataset. The fine-tuned model achieves 52% top-1 accuracy after 60 epochs. Figure 6 shows the accuracy and loss of the model.



(a) Top-1 accuracy



(b) Top-5 accuracy



(c) Loss

Figure 6: Accuracy and Loss for CustomNet2 model

2 Vision Transformers

2.1 Understanding Self-Attention

We implemented the self-attention mechanism in `Attention` class. The self-attention was calculated with

$S = softmax\left(\frac{QK^T}{\sqrt{D_q}}\right)V$

The `TransformerBlock` module implments local self-attention. So the input and output dimension of the `Attention` module is $[B, H, W, C]$, where
B = batch size
H = number of patches along image height
W = number of patches along image width
C = number of channels (embedding size)

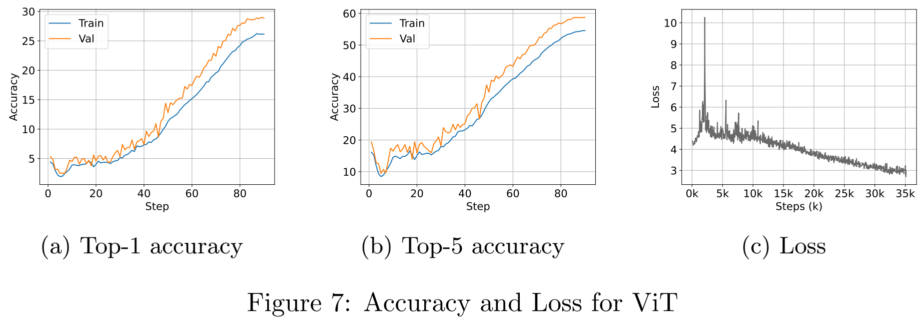
2.2 Design and Implement Vision Transformers

The vision transformer model we designed has 4 transformer blocks, followed by two linear layers. We perform average pooling of the output embeddings from the last transformer block. The hyperparameters we used for the model are described in table 2.

Hyperparameter	Value
Epochs	90
Patch size	16
Embedding size	192
Number of blocks	4
Number of heads	4
Batch size	256
Optimizer	AdamW
Weight decay	0.05
Learning rate	0.01

Table 2: ViT Training Hyperparameters

The training and validation accuracy, and loss during training the ViT training is provided in figure 7. The model reaches 28% top-1 accuracy after 90 epochs. We tried increasing the embedding size to 768 and increasing number of blocks to 6. However, the accuracy did not improve.



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

Figure 7: Accuracy and Loss for ViT

3 Adversarial Samples

PGD Attack deals with iteratively adding noise to input images to fool the model and make false predictions. The magnitude of this noise is very small in magnitude and is barely noticeable. The addition of this noise is guided by the gradient of the Loss with respect to the input image. The model is attacked iteratively, taking small steps in each iteration. Every pixel of the image is updated either in positive or negative direction based on the direction of its corresponding gradient.

3.1 Adversarial Samples

Adversarial samples are generated using SimpleNet on the validation dataset by back propagating the loss with respect to the class having the least confidence. A batch of images is passed through the trained SimpleNet to get the output predictions. Loss is computed considering the least confident class as the ground truth and the error is back propagated to the input image to calculate the gradients for each pixel of input data. All the pixels on the input batch are updated by a small step either in positive or negative direction based on the sign of the gradient of that pixel. This updated image is detached from the computational graph and passed though the model again and the entire process is repeated multiple times. To keep the noise under control, the pixel differences between the adversarial image and original image is clipped and bounded between a tolerance value so that the magnitude of noise remains barely noticeable. Figure 8 shows some generated adversarial samples. The adversarial samples are generated with the following command

```
python ./main.py ../data --resume=model_path -a -v
```

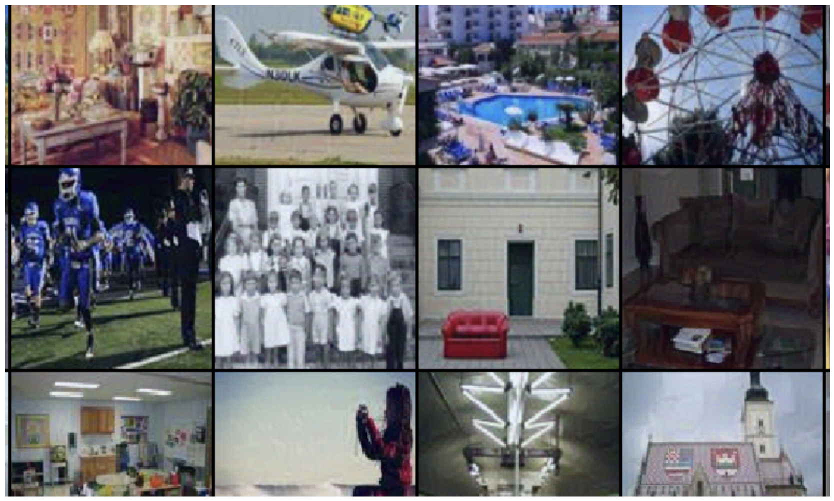


Figure 8: Adversarial samples

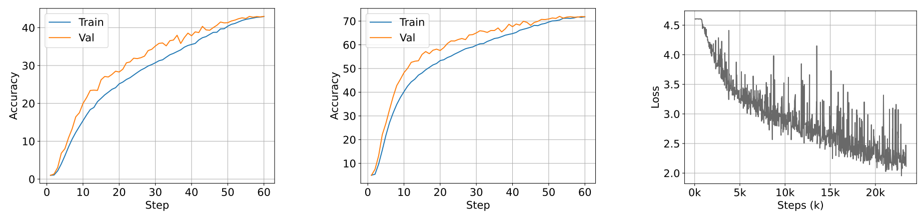
Observations
We compared the accuracy of the model test dataset on clean and perturbed images by changing the number of steps and epsilon. We found that adversarial attacks decrease the accuracy. Further we find that epsilon shows a greater impact on accuracy compared to the number of steps. Increasing the tolerance value gives the scope for more noise addition, increasing the chances of creating false predictions.

3.2 Adversarial Training

To make the model robust to such adversarial attacks, we design a training strategy to handle these noise. During training, the input batch first is perturbed by a PGD attack using the same model. These manipulated images are concatenated with clean images to generate the final batch that goes for training. The batch contains both, clean as well as perturbed images which then goes for training. Keeping both, clean and perturb images in the same batch can help in stable training as the weights are updated based on clean and perturbed images simultaneously.

To increase robustness of the model, we vary number of steps, epsilon value between a range. We also generate adversarial samples 20% times during training to keep the model stable. Figure 9 shows the accuracy and loss of the model trained with this method. The adversarial training can be run with the following command

```
python main.py ../data --epochs=60 --pgd=True
```



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

Figure 9: Accuracy and Loss for the model with adversarial training

The model trained with adversarial samples performs slightly better against adversarial samples. The `SimpleNet` model without adversarial training achieves 40.2% accuracy on adversarial samples, while the model with adversarial training achieves 41.2% accuracy on adversarial samples.