MDS 5122 Assignment 1

Guyuan Xu 224040074

A. Build a Neural Network Using PyTorch

1. reproducing example NeuralNetwork of example code A.

Results are summaried in subsection 2.2 together with other settings, codes check notebook

2. Factors to improve accuracy

In my experiment, I tried to use 1) Simple NN structure(which is the baseline model of example code A) and NN with more complex architecture (vgg16 and ResNet18/50); 2) adding BatchNorm layer and dropout layer; 3) different optimziers SGD and Adam with popular *lr* or momentum params settings; 4) with/without data augmentation.

We mainly use the model of example code-A as our baseline model to examine how the above factors might affect accuracy. And due to computing resource limitation, i did not conduct strict experiment with strict variable controls. To gain insight is what we want.

Experiments Setup

2.1 Baseline Convolutional Neural Network (C5L3)

- architecture: 5 conv layers, each followed by a dropout layer (rate=0.3); 3 linear layers, each followed by a dropout layer (rate=0.5).
- optimizer: Adam with a learning rate of 3e-4 and weight decay of 1e-6.
- dataset: Trained on the CIFAR-10 dataset. The training set has dimensions (60,000, 3, 32, 32), and the test set has dimensions (10,000, 3, 32, 32).
- Training settings: Trained for 150 epochs with a batch size of 64. Model accuracy was evaluated on the test set.

2.2 C5L3 architecture and data prep overview

```
# baseline model(C5L3) architecture:
net = nn.Sequential(
  # conv layer: 5
    nn.Conv2d(3, 128, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(256, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 256, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Flatten(),

  # linear layer: 3, excluding the last layer
```

```
      nn.Linear(256 * 4 * 4, 512), nn.ReLU(inplace=True), nn.Dropout(0.5),
      nn.Linear(512, 256), nn.ReLU(inplace=True), nn.Dropout(0.5),
      nn.Linear(256, 128), nn.ReLU(inplace=True), nn.Dropout(0.5),
      nn.Linear(128, 10),
  )
# ParamsCount: 3*128*3*3+128*256*3*3+256*512*3*2*3+512**2*3*3+256*16*512+512*256+256*128+1280=7.28M

# Using default data transformation in example code-A
  transformation = dict()
  for data_type in ("train", "test"):
    is_train = data_type=="train"
    transformation[data_type] = tv_transforms.Compose((
      [
      # default data augmenation
        tv_transforms.RandomRotation(degrees=15),
        tv_transforms.RandomHorizontalFlip(),
        tv_transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),

      # extra data augmentation:
        tv_transforms.ColorJitter(
          brightness=0.2,
          contrast=0.2,
          saturation=0.2,
          hue=0.1)
      ] if is_train else []) +
    [
        tv_transforms.ToTensor(),
        tv_transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
    ])
```

Example of data augmentation on CIFAR-10:

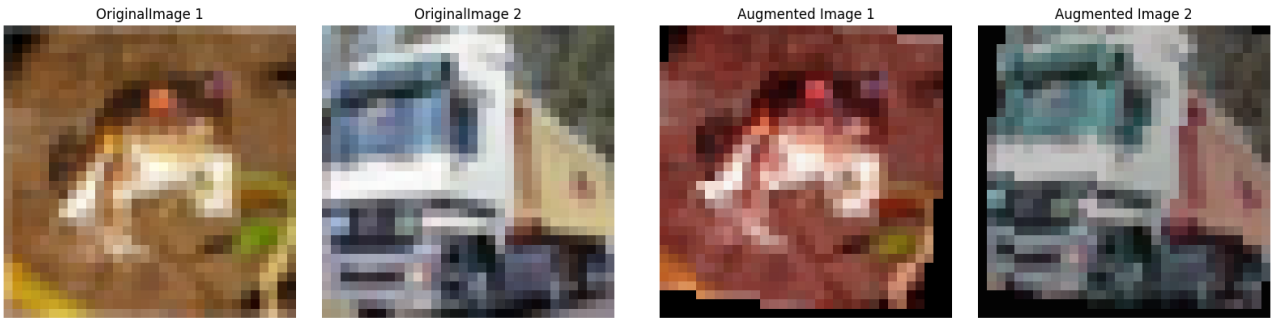

Figure 1: original(left), augmented(right):random filp/rotate/affine/colorjit

Training strategies We have 6 (basline models with different settings on CIFAR-10) + 3 (advanced network on CIFAR-10) + 1 (best model on MNIST) = 10 models in total to train. We decided to run 10 separates .ipynb on kaggle, 1 for each model with it's corresponding settings, insteading of packing all in one file for the following reasons:

- Parallel: kaggle allows running 2 notebook at the same time, save time.
- Flexibility: we can adjust each model at any time.
- Fault tolerance: If we run everything in 1 file, we will loose everything if 1 error pop up, wasting time quota of free gpu usage.

2.2 Baseline model + additional modifications beyond the Baseline(C5L3):

- Extra Data Augmentation: Aside from default techniques in baselline model including random flipping, random rotation, random affine transformation, we add random brightness adjustments, and saturation modifications.
- Optimizer Variant: Replaced Adam with SGD (learning rate=1e-3).
- Architectural Varaint: Removed dropout layers after convolutional layers (retained dropout for linear layers), or adding BatchNorm layers after convolutional layers.

Results on CIFAR-10 are summarized in the folloing table

Table 1: ModelSettings and Performance

Model	C5L3 base	C5L3 variant				
SettingNo.	1	2	3	4	5	6
Adam	√		√	√	√	√
SGD		√				
ExtraDataAug			√	√		√
dropout	all	all	linear	none	linear	linear
BatchNorm			conv	conv	conv	conv
MaxEpoch	150	150	150	150	150	256
BestTestAccu	85.11	84.95	89.31	89.69	88.34	89.73

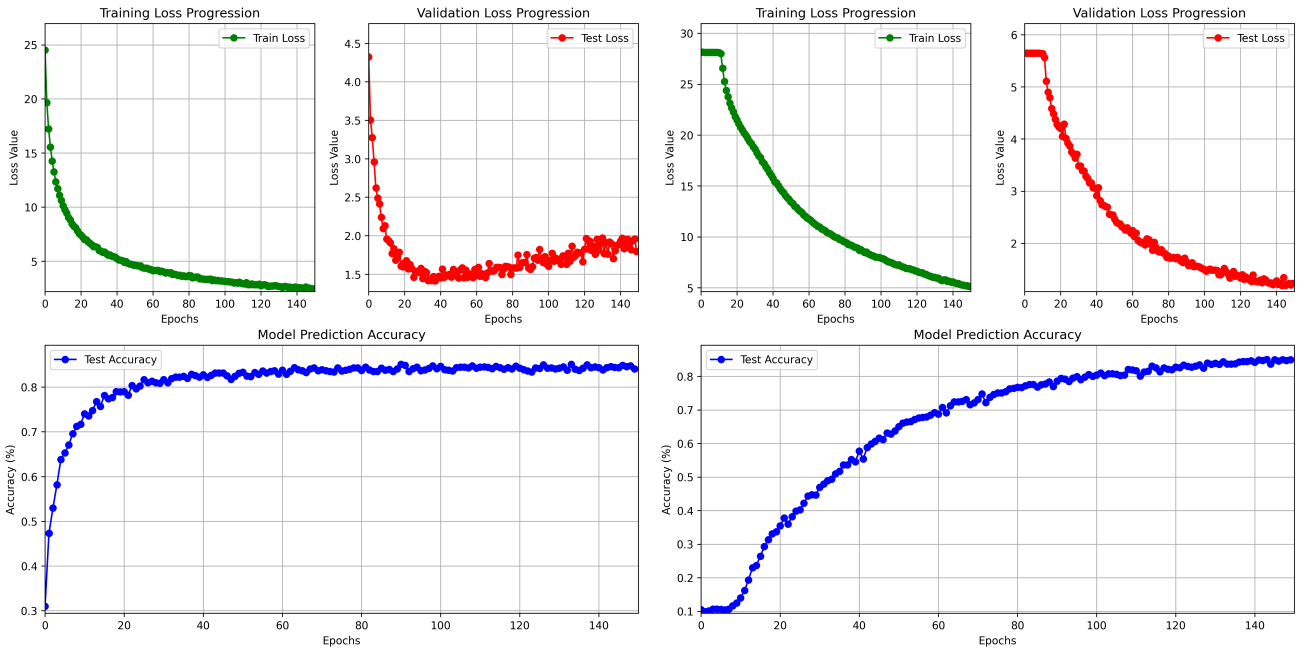

Figure 2: model settings 1(left) 2(right)

Note: bacthsize = 64, Epoch num = 150, dropout(conv) indicate whether we use dropout layer in convlution layer.

- Conclusion:

- BatchNorm considerably improve accuracy (+4.5%), compared with other techniques.
- choosing SGD or Adam will not significantly affect accuracy, they just converge to almost the same accuracy at different speeds (in terms of epochs). attach figure of loss curve

- dropout layers have almost no effect to the performance of model, since we have tested 2 scenarios: 1) keep dropout layers only in Linear layer (popular method) 2) removing all dropout layers; and they do not differ much in terms of test accu.
- Notice that the last column of the table, we re-do the 3rd settings for 256 epochs, and the test accuracy only improve for a margin.

2.3 Deeper Neural Network

Although our basline model can achieve around 90% accuracy on test set, deeper neural network can always (at a high prob) handle complicated features better, here, cifar-10 images have complex features. So we decided to try deeper networks `vgg16, ResNet18, ResNet50`.

`vgg16` stacking 5 blocks of conv->batchnorm->relu->conv->batchnorm->relu->maxpool, hoping to extract more abstract representation of features.

`ResNet18` stacking 4 basic ResidualBlocks, by doing so it allows gradient to propagate without exploding or disappearing.

`ResNet50` stack 4 special ResidualBlock (named BottleNeck Block) and allows gradient to propagate in even deeper network.

Table 2: ModelSettings and Performance

Model	vgg16	ResNet18	ResNet50
TestAcc	*	*	*

2.4 MNIST dataset

We first use basline model together with our best params on MNIST dataset. Check notebook codes.

Interesting findings and reflections

1) visualize data after data augmentation 2) what have i learnt. 3) due to curiosity，可以用 batchsize 32 以及 256 epoch 对 best model 进行验证，看看 accu 能否上 90

B. Build a Neural Network From Scratch

1. AutoGrad: To Learn from the better

When attempting to build a deep learning framework from scratch, I referenced numerous materials, including Example-Code-Part-B and PyTorch. However, PyTorch is an immensely large framework that integrates numerous heavily optimized modules. The implementation logic of most modules has become unrecognizable under massive speed and compatibility optimizations, which makes it akin to a "rocket science project" for a beginner. At least for me, it is not a good reference material.

In Example-Code-Part-B, the implementation of backpropagation relies on MANUALLY deriving gradients and writing them into .backward() methods, which are then called in a nested way to perform backward propagation when gradients are needed. This raised concerns for me: For more complex network architectures (e.g., ResidualBlock in ResNet), manual gradient derivation would be nearly impossible. Does this mean that after painstakingly "implementing" the simple functionalities required for the assignment, I still cannot gain much insight into the model DL framework (especially AutoGrad and Computing Graph), or to use these components to build more sophisticated networks?

This concern drove me to explore further, leading me to discover the Chainer framework. Its core innovation is automatic differentiation via the dynamic computational graph (Define-by-Run), which allows users to construct the computation graph on-the-fly during the forward propagation process then attain gradients automatically, rather than predefining a static structure. Chainer was the first framework to propose dynamic computational graphs and inspired PyTorch's design. However, Chainer's high complexity still made it unsuitable for a novice learner.

I then discovered the book Building Deep Learning Frame, which introduces a framework called DeZero—a simplified version of Chainer. It is sufficiently complex and flexible, yet not overly daunting for learners. The book provides the complete source code for DeZero, but simply copying the code would not have benefited me. Since I started this project early (immediately after the assignment was released), I spent two full weeks studying every line of DeZero's code character by character. I abandoned the author' s original implementation of the Conv2d module and reimplemented it from scratch, and I independently implemented FocalLoss module, which were not provided by the author, as tests to verify my understanding. All of this was made possible by the concepts of automatic differentiation and dynamic computational graphs, freeing me from the agony of manual gradient derivation and debugging, and preparing me to build complex deep networks like ResNet18/50 in the future.

2. File Structure of the framework

The core of my framework is Define-by-Run AutoGrad, it builds connections between variables for later back-propagation when doing calculation. The DeZero module is composed of a series of `.py` files, which can be imported like a standard Python package using `import`. Below I will outline the module's file structures and map the assignment-required functionalities to their corresponding implementations position. For certain noteworthy implementation approaches, I will briefly discuss them here.

I want to emphasize again: I did not simply copy and submit the DeZero module as-is. The annotations and drafts within the module serve as evidence of this—I invested significant time and effort, ultimately not to deceive myself, but to deeply understand and internalize the concepts.

```
dezero-master: .
```

```
core.py --------# base class: Function and Variable(a torch.tensor like data structure)
core_copy_beforeCuPY.py --------# ignore
core_simple.py --------# ignore
cuda.py
dataloaders.py
datasets.py
functions.py --------# Sigmoid/LeakyReLU/Dropout/FocalLoss/
functions_conv.py --------#conv modules: F.Conv2dV function
layers.py --------# Linear/Conv2dV/BatchNorm2d/
models.py --------# C5L3/vgg16/ResNet18/50
optimizers.py --------# SGD/Adam
utils.py
__init__.py
```

a) Sigmoid: ../dezero-master/dezero/functions.py - line 457

b) LeakyReLU: ../dezero-master/dezero/functions.py - line 509

c) SELU: not implemented yet

d) Linear: ../dezero-master/dezero/layers.py - line 118

e) Conv2dV: ../dezero-master/dezero/layers.py - line 221, the original conv2d method is complicated, i wrote a new one (named Conv2dV) with im2col in forward and col2im in backward, almost as fast as the original one.

f) Dropout: ../dezero-master/dezero/functions.py - line 691

g) BatchNorm2d: ../dezero-master/dezero/layers.py - line 311, automatically detect num_features, so no need to add this config

h) SigmoidFocalLoss: ../dezero-master/dezero/functions.py - Line 570

i) SGD: ../dezero-master/dezero/optimizers.py - line 38, 2 ways of implementation, 1)manual: forward+backward, marginally faster 2)auto: simply define the forward caculation, and get gradients through auto grad, almost as fast as 1).

j) Adam: ../dezero-master/dezero/optimizers.py - line 84

做完 1 的试验后获取 best 参数，接着用 dezero 堆叠一个相同结构。

Reflections

From BackwardPropagation to AutoGrad: In example code part-B we can learn how forward pass and backward pass work together in a neural network, but implementing backward pass is complicated and sometimes impossible (especially in complicated computation). Meanwhile we know that all computation in deep learning are basically consist of few basic functions like exponents, add, subtract, multiplication etc, so we can utilizie chain rule to calculate gradient no matter how complicated a computation might be. This is how AutoGrad comes into play, and this is the core of all deep learning frameworks, including pytorch.

Understand by creating: During the process of building a framework, i learn extensively and experience moments of revelation like ”ah, this is how a neural network works!” or ”so this algorithm can be implemented this way!” These insights are unattainable through merely using existing tools like pytorch. Some concepts can only be understood by creating, some truths can only be seen by building.

For instance, some may view deep learning frameworks as mere libraries pieced together with layers and functions. In reality, frameworks go far beyond that. A framework can be considered a programming language—specifically, one with automatic differentiation capabilities (recently termed a ”differentiable programming language”).