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.

2.0 Experiments Setup: How to run

All training codes are packed in files located in `./kaggleFile`, what are those `.ipynb` files' names mean:

- All files start with `main_` are training files
- if not specify explicitly in file name, models are trained in CIFAR-10 dataset.
- `main_bsln_4` means "baseline model" with setting number 4 on CIFAR-10, `main_res18` means "ResNet18" model on CIFAR-10
- change "code_file" in `./kaggleFile/kernel-metadata.json` to the file name you want to run. Then push it to kaggle. All training results/weights, including visualization of train/test loss and test accuracy will be saved to /kaggle/output automatically.

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:

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.

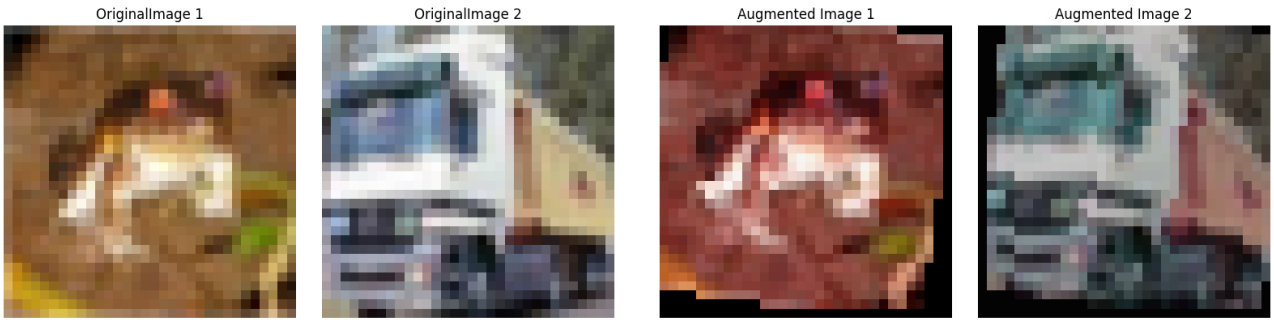

Figure 1: original(left), augmented(right):random filp/rotate/affine/colorjit

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
BestEpoch	134	146	118	147	84	236

We now interprate the results together with the curve of train/test loss + test accuracy.

1) Figure 2 shows the train/test results of training setting 1 and 2: same on model structure + no data aug + full dropout after all layer, only differ in optimizer, where setting 1 adopt Adam and setting 2 adopt SGD. It is obvious that Adam covnerge faster than SGD, while the best testing accuray is almost the same: 85%. And we can also find that Adam one might be over fitted: for the testing loss(tagged validation loss, although this is not validation techinically) goes up after epoch 40, while SGD's testing loss descent steadily.

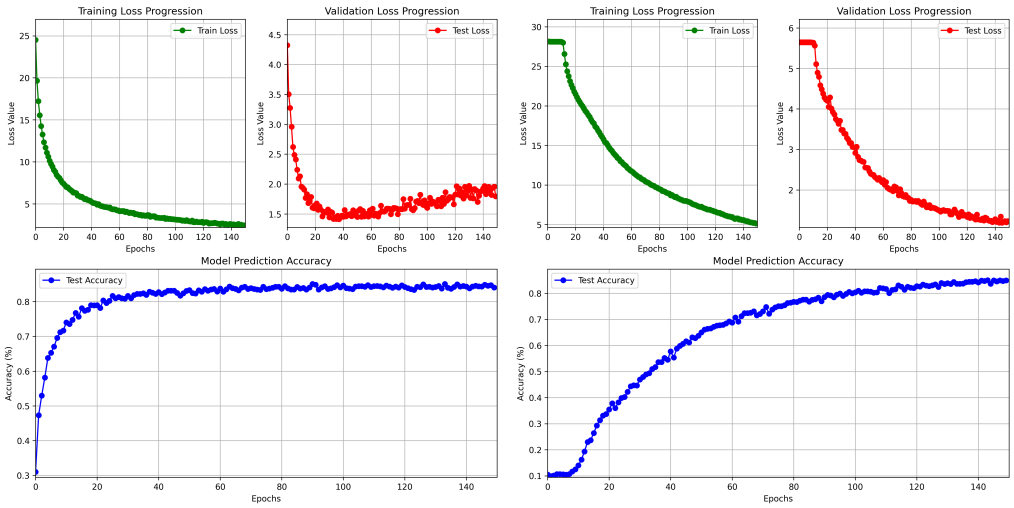

Figure 2: model settings 1(left):Adam 2(right):SGD

2) Figure 3 shows setting 3 and 3: same on model structure and other, only differ in whether using extra data augmentation (Color jitter), seems data augmentation is usefull cause the test accuracy is higher by 1%, and they have almost the same curves.

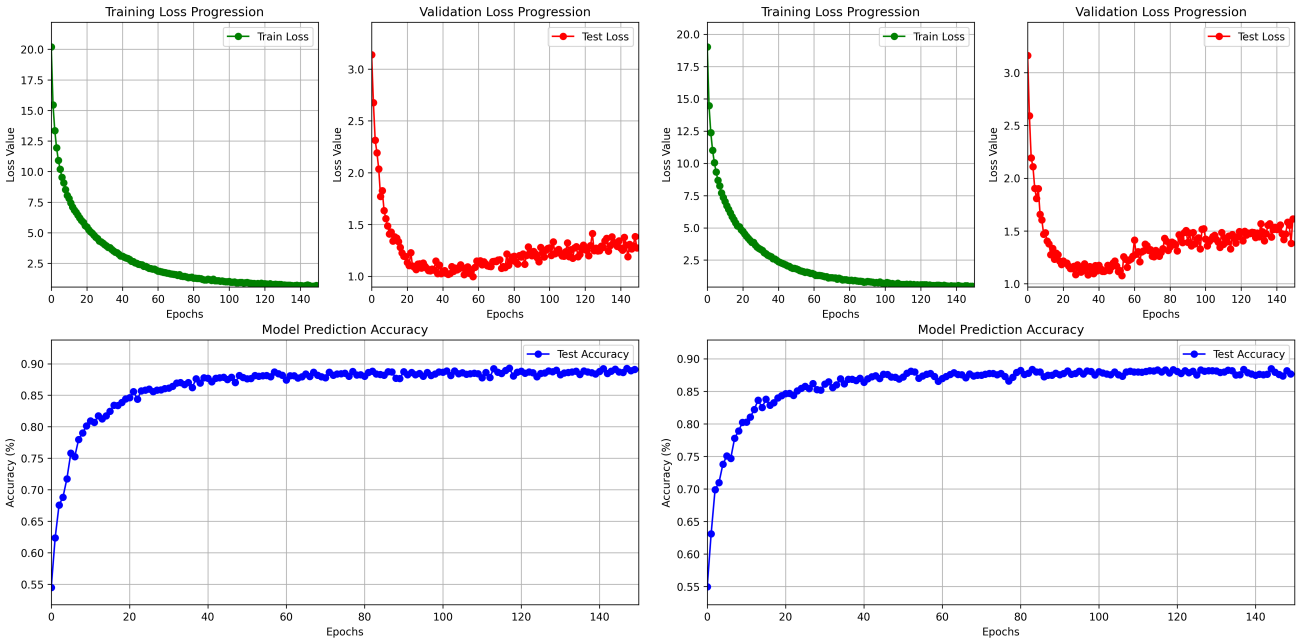

Figure 3: model settings 3(left): extra data aug 5(right): no extra data aug

3) Figure 4 shows setting 3 and 4: same in model structure and other, only differ in whether to use dropout after linear layer. It is surprising that the one without dropout has higher testing accuracy, because in common sense dropout layer will decrease overfitting. And there is no significant difference in loss curves eigther. Hence we can conclude that **dropout layer has no significant improve in our baseline model**

4) At Figure 5, we also add more training epochs to our BEST model: **Why we choose setting 3 to be the BEST? Although setting 3 has better accuracy, but only better in a margin, but dropoutput can accelerate training, because it "dropout" 50% params in linear layer.** the BEST model is the same with setting 3, only differ in number of training epoch, we add 100 more epoch to see if the testing accuracy can go higher. But find out

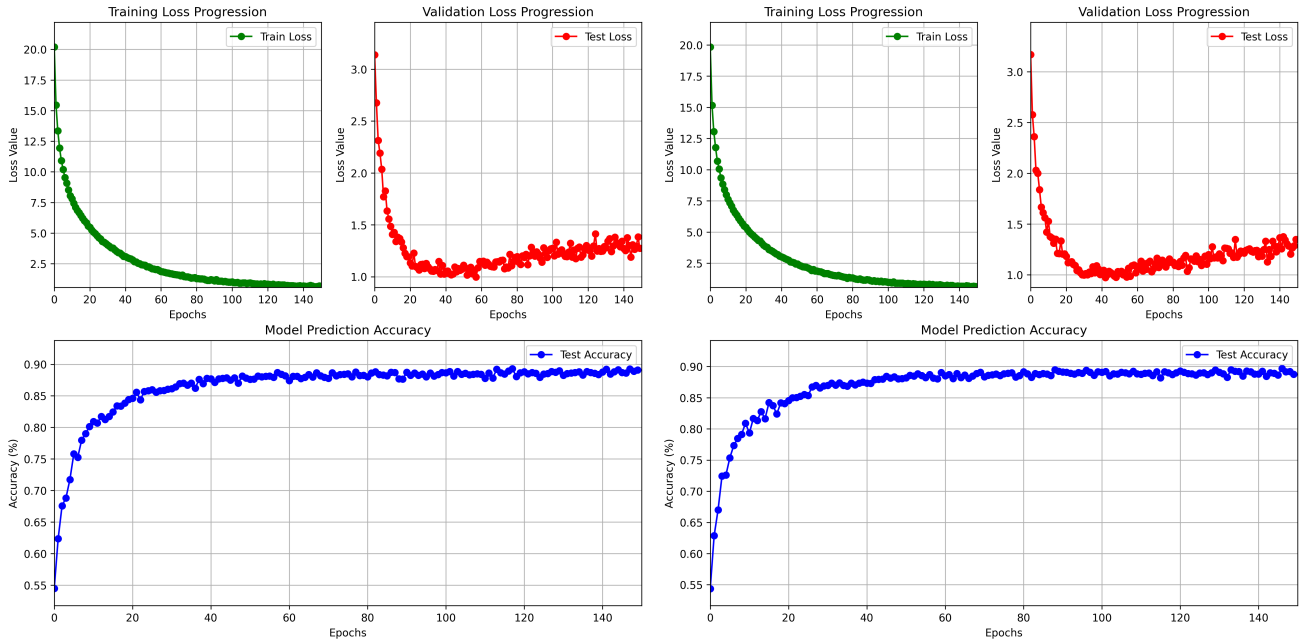

Figure 4: model settings 3(left): keep only dropout after linear layer, (right): no dropout

the best testing accuracy didn't improve, and never go beyond 90%, it converges to 89.5%.

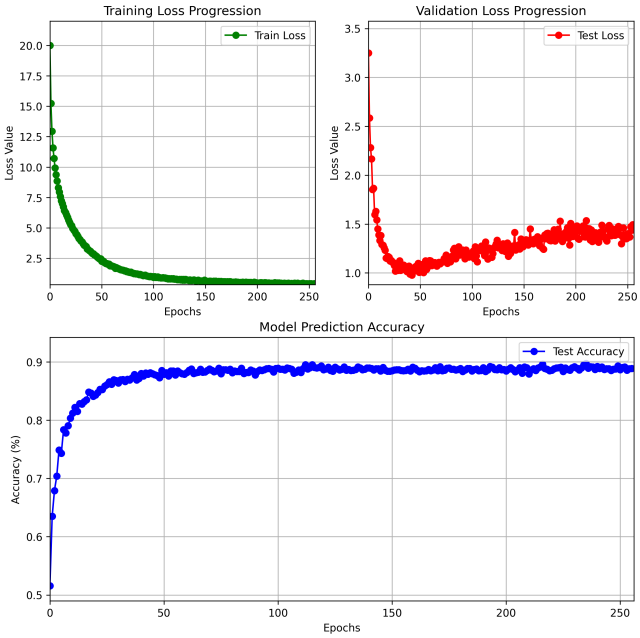

Figure 5: model settings 6: BEST model (same with setting 3) with more training epochs

5) At Figure 6: since dropout layer has no much difference to model performance, by comparing setting 1 and setting 3, we can conclude that BatchNorm has significant improvement to testing accuracy.

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

- Baseline Model Results Interpretation:

- BatchNorm considerably improve accuracy (+4.5%), compared with other techniques.

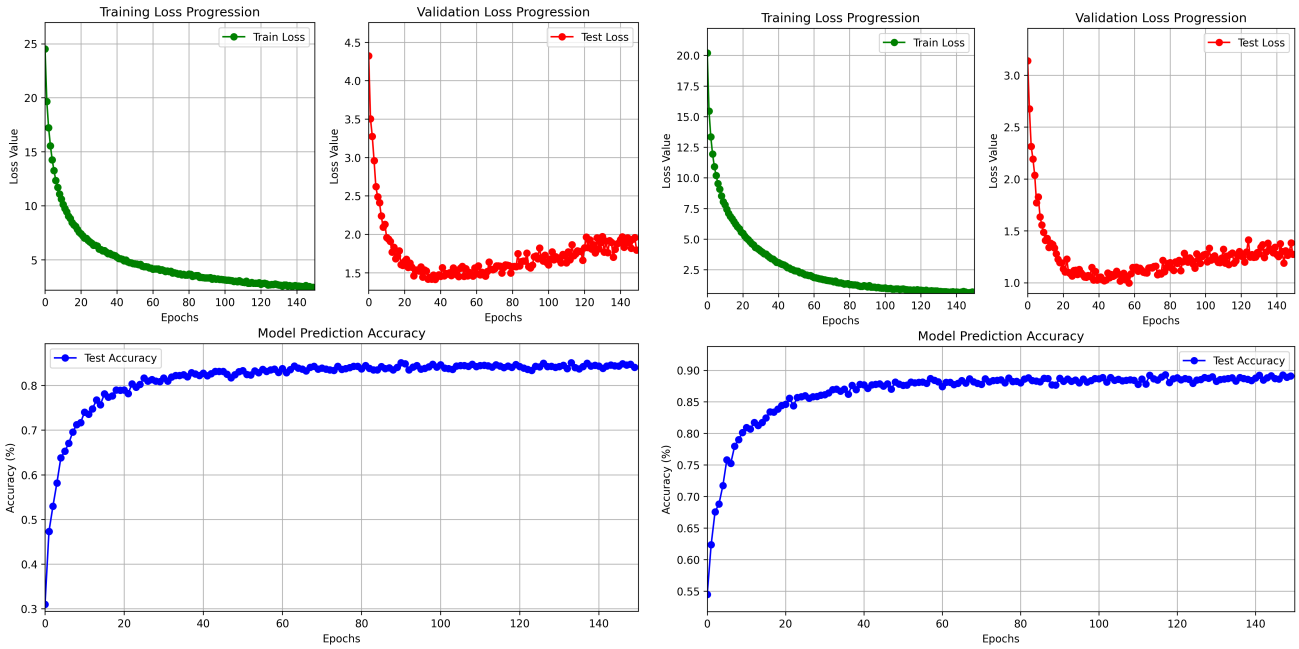

Figure 6: model settings 1(left): no Batchnorm, setting 3(right): with Batchnorm

- 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. (almost no improvement)

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.

The training and testing results are shown in below Table 2 + Figure 7.

Table 2: ModelSettings and Performance on cifar-10

Model	vgg16	ResNet18	ResNet50
TestAcc	91.01	92.51	92.29

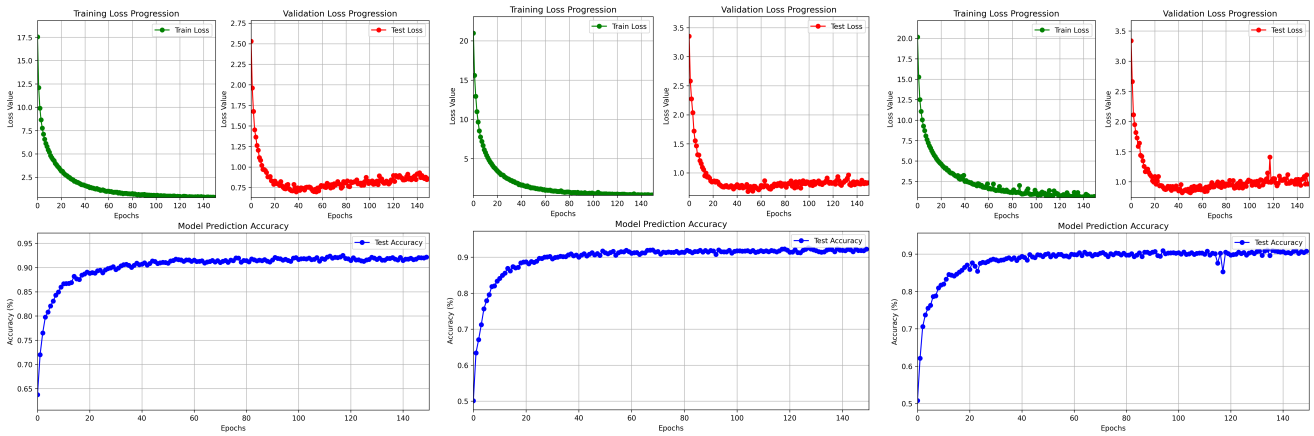

Figure 7: ResNet18(left), ResNet50(middle), vgg16(right)

We can terpretate from the results:

- deeper neural networks are more capable of learning picture pattern than lower ones
- we can see from the loss curves that, these 3 deep neural network do not grow as much as basline model in testing loss as epoch goes up: they are less likely to be overfitting than our baseline model, and hence get a higher testing accuracy: 92%+.
- ResNets are better than vgg16 because of the skip connections between ResidualBlocks so the gradients propagate in a more stable manner.
- VGG16 aggressively reduces resolution via max-pooling (5 maxpooling layer, causing the receptive field to shrink to 1x1), causing severe spatial information loss in later layers. While ResNet preserves details through gradual downsampling.

3 Train and eval on MNIST dataset

We decide to test 2 models in MNIST dataset: 1) Baseline model with BEST settings (setting 3: Adam+dropout after linear + batchnorm), 2) ResNet18. Notice that MNIST's image has only 1 channel instead of 3, the extra data augmentation–Color jit will automatically idle: doing nothing to the figure, an example of origianl image and image after transformed are like:
Results are shown in Table 3 and Figure 8: We can interprate from the result:

- We can see diminishing returns of depth here: ResNet18's residual connections and deeper layers (18 layers) are designed to solve vanishing gradients and feature degradation in very deep networks (e.g., 50+ layers).

  For MNIST's shallow feature hierarchy, these mechanisms provide negligible benefits, as even simple CNN structures avoid gradient issues at this depth.
- The dataset's 60,000 training samples are sufficient to regularize both simple and moderately complex models, preventing overfitting. ResNet18's parameters (11M) are not excessively large relative to MNIST's size, unlike scenarios with smaller datasets (e.g., CIFAR-10 with 5k samples per class).
- MNIST's simplicity creates a "performance ceiling" where even modest models achieve near-perfect accuracy, rendering architectural refinements largely unnecessary. The observed parity underscores the importance of matching model complexity to task difficulty.

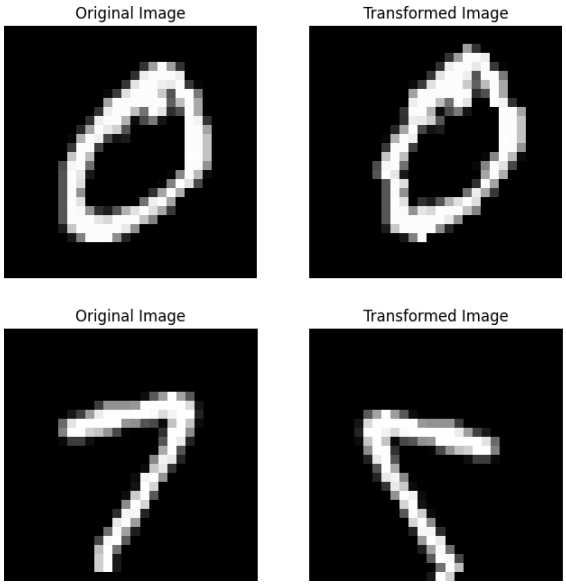

Figure 8: MNIST image: original 0/7 (left coolumn), transformed 0/7 (right column)

Table 3: ModelSettings and Performance on cifar-10

Model	BaseLine Best	ResNet18
TestAcc	99.35	99.50

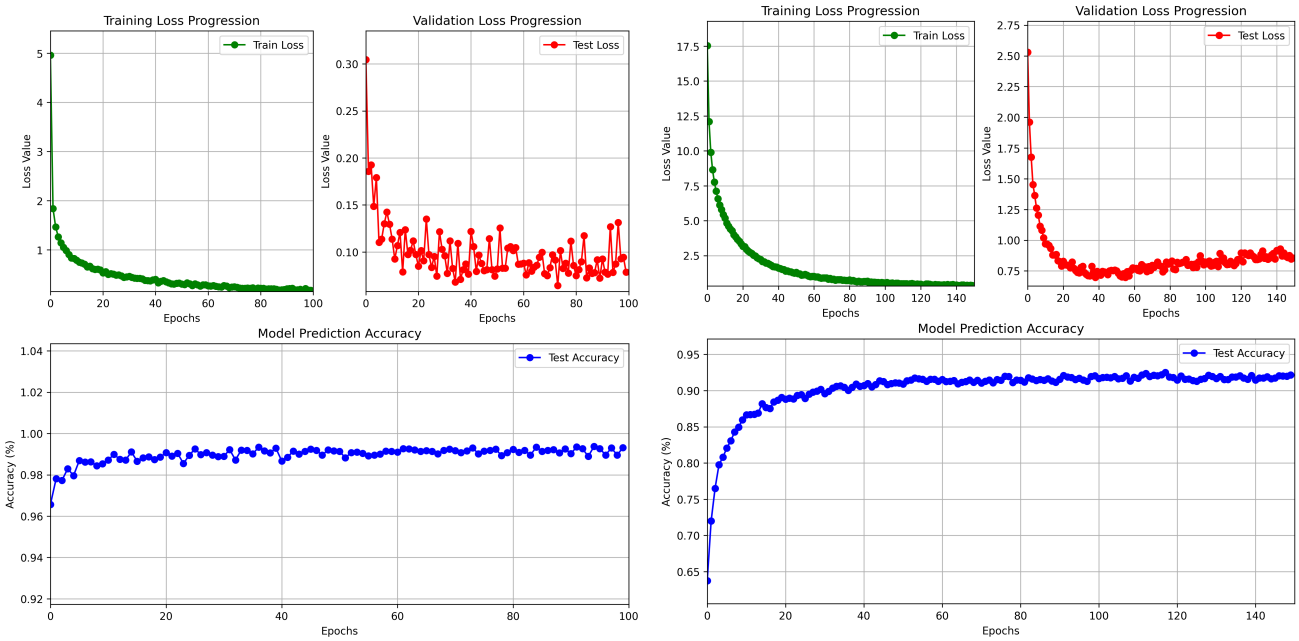

Figure 9: Basline(left), ResNet18(right) on MNIST

Interesting findings and reflections(What have i learnt)

1) I wasted significant amount of time tuning hyperparameters because the original training framework had messy dependencies between variables – adjusting a single parameter often required modifying multiple places. This ultimately drove me to encapsulate the entire workflow (data preprocessing, model selection, optimizer configuration, training loop, result saving, and visualization) into a massive main function, exposing only hyperparameters via an API.

2) Avoid stucking all experiments into a single `.ipynb` file. If any error occurs midway (which is common due to discrepancies between the training environment and local debugging environment), all training results could be lost. Therefore, I chose to execute each experiment in separate files, even though this approach appears less organized.

3) When constructing neural networks, one must consider the input image's channel nums. While using lazy-initialized layers like `nn.LazyConv2d` can automatically adapt to input channel dimensions, this approach introduces a limitation: you cannot calculate or print the model's total parameter count before feeding data to the model. Instead, you must first pass a dummy data batch to trigger lazy initialization, after which the layers will have initialized parameters.

B. Build a Neural Network From Scratch

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.

File Structure of the self-made framework: How to run

- All necessary codes for Task B are packed in `./dezero-master`
- `./dezero-master/dezero` contains all source code of the self-made framework, it can be imported like a normal python packages just like pytorch.
- `./dezero-master/dezero/dezero_bsln.ipynb & dezero_res18.ipynb` is the training files, push it to kaggle like before to start training. All training results/weights, including visualization of train/test loss and test accuracy will be saved to /kaggle/output automatically.

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.

```
dezero-master: .
    core.py --------# base class: Function and Variable(a torch.tensor like data structure), core modules for define-by run AutoGrad
    core_copy_beforeCuPY.py --------# ignore, testing module
    core_simple.py --------# ignore, testing module
    cuda.py --------# cupy adaptation
    dataloaders.py
    datasets.py
    functions.py --------# Sigmoid/LeakyReLU/SELU/Dropout/FocalLoss
    functions_conv.py --------#conv modules: F.Conv2dV function
    layers.py --------# Linear/Conv2dV(Conv2d)/BatchNorm(BatchNorm2d)
    models.py --------# C5L3/vgg16/ResNet18/50
    optimizers.py --------# SGD/Adam
    utils.py
    __init__.py
```

1. Implementations

My implementation codes path:

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

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

c) **SELU**: ../dezero-master/dezero/functions.py - line 670

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

e) **Conv2d/Conv2dV**: ../dezero-master/dezero/layers.py - line 221, the standard conv2d implementation is complicated, i also wrote a new one (named Conv2dV) with im2col in forward and col2im in backward, almost as fast as the original one. And the reproduction of the results from task A using my framework adopts Conv2dV as convolution layers.

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

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

h) **SigmoidFocalLoss_manual/SigmoidFocalLoss_simple**: ../dezero-master/dezero/functions.py - Line 570, 2 ways of implementation: 1) manually: `SigmoidFolcaLoss` implementation adopt `.forward(), .backward()`, which will be slightly faster; 2) automatically: `SigmoidFocalLoss_simple` only calculate the forward procedure, but the backward calculation can be done automatically via AutoGrad

i) **SGD**: ../dezero-master/dezero/optimizers.py - line 38

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

1.2 Testing implementations that not be used

Since `LeakyReLU, SELU, SigmoidFocalLoss` will not be used by my reproduction, we first test them and compare the results to `torch.nn.functions`'s implementation to see if they actually do the right things: all test codes are packed in `../dezero-master/DeZero_test_notes.ipynb`

```
# test self-made framework's sigmoid focal loss
from imp import reload
from dezero import Variable
import dezero.functions as F
reload(F)
import numpy as np

# input a random variable into  sigmoid focalloss
x = Variable(np.array([1.0, -3.0,2], dtype=np.float32))
target = Variable(np.array([0.0, 1.0, 0.0], dtype=np.float32))

# cal focal loss (forward test)
loss = F.sigmoid_focal_loss_manual(x, target, alpha=0.25, gamma=3)
print("Loss:", loss.data)
# cal gradient (backward test)
x.cleargrad()
loss.backward()
print(x.grad)

# loss and gradients:
>>> Loss: [0.7112133]
>>> gradients: variable([ 0.17490506 -0.099857    0.28040347])


# verify using torchvision.ops.sigmoid_focal_loss
from torchvision.ops import sigmoid_focal_loss as focal
import torch

x = torch.tensor(np.array([1.0, -3.0, 2], dtype=np.float32), requires_grad=True)
target = torch.tensor(np.array([0.0, 1.0, 0.0], dtype=np.float32), requires_grad=False)

loss_torch = focal(x, target, alpha = 0.25, gamma = 3,reduction='mean')
print("loss(torch):", loss_torch)

loss_torch.backward()
print("gradients(torch):",x.grad)

>>> loss(torch): tensor(0.7112, grad_fn=<MeanBackward0>)
>>> gradients(torch): tensor([ 0.1749, -0.0999,  0.2804])
```

```
# self-made framework SELU test
import dezero.functions as F
from imp import reload
reload(F)
from dezero import Variable
import torch
import numpy as np
```

```

x = Variable(np.array([1.0, -3.0,2.0, 5], dtype=np.float32))
x_np = np.array([1.0, -3.0,2])
y = F.sum(F.selu(x))
y.cleargrad()
y.backward()
print("loss:", y)
print("grads:", x.grad)

>>> loss: variable(6.7350397)
>>> grads: variable([1.0507009873554805 variable(0.08753060945898154) 1.0507009873554805
1.0507009873554805])

# verify using torch.nn.functional.selu
x_t = torch.tensor(np.array([1.0, -3.0, 2.0, 5]), requires_grad = True,  dtype = torch.float32)

y_t = torch.nn.functional.selu(x_t).sum()
y_t.backward()
print("loss(torch): ",y_t)
print("grads(torch): ", x_t.grad)

>>> loss(torch):  tensor(6.7350, grad_fn=<SumBackward0>)
>>> grads(torch):  tensor([1.0507, 0.0875, 1.0507, 1.0507])
```

```

# self-made framework's leaky relu test
x = Variable(np.array([1.0, -3.0,2.0, 5], dtype=np.float32))
y = F.sum(F.leaky_relu(x))
y.cleargrad()
y.backward()
print("activation: ",y)
print("grad: ", x.grad)

>>> activation:  variable(7.9700003)
>>> grad:  variable([1.   0.01 1.   1.  ])

# verify with torch.nn.functional.leaky_rely
x_t = torch.tensor(np.array([1.0, -3.0, 2.0, 5]), requires_grad = True,  dtype = torch.float32)
y_t = torch.nn.functional.leaky_relu(x_t,negative_slope=0.01).sum()
y_t.backward()
print("activation(torch): ", y_t)
print("grads(torch): ", x_t.grad)

>>> activation(torch):  tensor(7.9700, grad_fn=<SumBackward0>)
>>> grads(torch):  tensor([1.0000, 0.0100, 1.0000, 1.0000])
```

1.3 Model structures under self-made framework

Check models file at: ../dezero-master/dezero/models.py. Here is an example of `Baseline` model under self-made:

```

# C5L3, ignore below C5L4, wrong name
class C5L4(Model):
    def __init__(self):
```

```
        super().__init__()
        self.conv1 = L.Conv2d(128,kernel_size=3, stride=1, pad=1)
        self.bn1 = L.BatchNorm2d()

        self.conv2 = L.Conv2d(256,kernel_size=3, stride=1, pad=1)
        self.bn2 = L.BatchNorm2d()

        self.conv3 = L.Conv2d(512,kernel_size=3, stride=1, pad=1)
        self.bn3 = L.BatchNorm2d()

        self.conv4 = L.Conv2d(512,kernel_size=3, stride=1, pad=1)
        self.bn4 = L.BatchNorm2d()

        self.conv5 = L.Conv2d(256,kernel_size=3, stride=1, pad=1)
        self.bn5 = L.BatchNorm2d()

        self.fc1 = L.Linear(512)
        self.fc2 = L.Linear(256)
        self.fc3 = L.Linear(128)
        self.fc4 = L.Linear(10)
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.pooling(x, 2, 2)
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.bn4(self.conv4(x)))
        x = F.relu(self.bn5(self.conv5(x)))
        x = F.pooling(x, 2, 2)

        # flatten, x.shape[0] is N(batchsize)
        x = F.reshape(x, (x.shape[0], -1))
        # print(x.shape)
        x = F.dropout(F.relu(self.fc1(x)), dropout_ratio=0.3)
        x = F.dropout(F.relu(self.fc2(x)), dropout_ratio=0.3)
        x = F.dropout(F.relu(self.fc3(x)), dropout_ratio=0.3)
        x = self.fc4(x)
        return x
```

2. Reproducing Task A on MNIST

Table 4: Reproducing Task A on MNIST using self-made framework

Model	BaseLine Best	ResNet18
TestAcc(self-made)	98.972	98.903
TestAcc(torch)	99.35	99.50

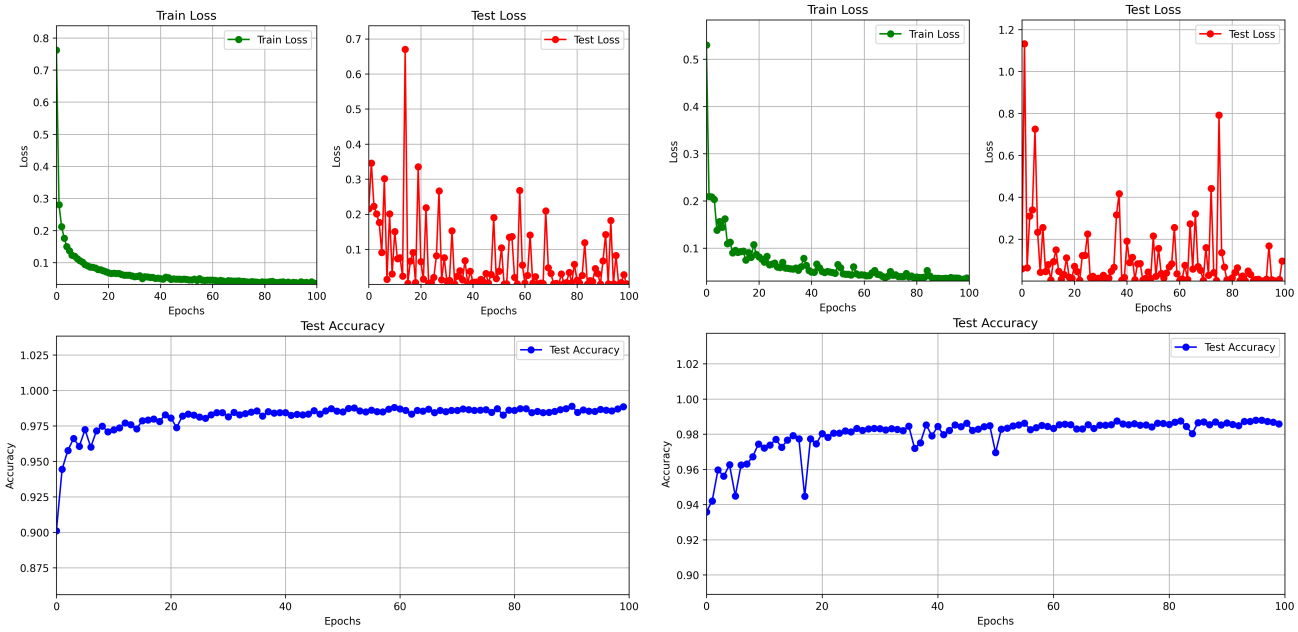

Figure 10: Basline(left), ResNet18(right) on MNIST using self-made framework

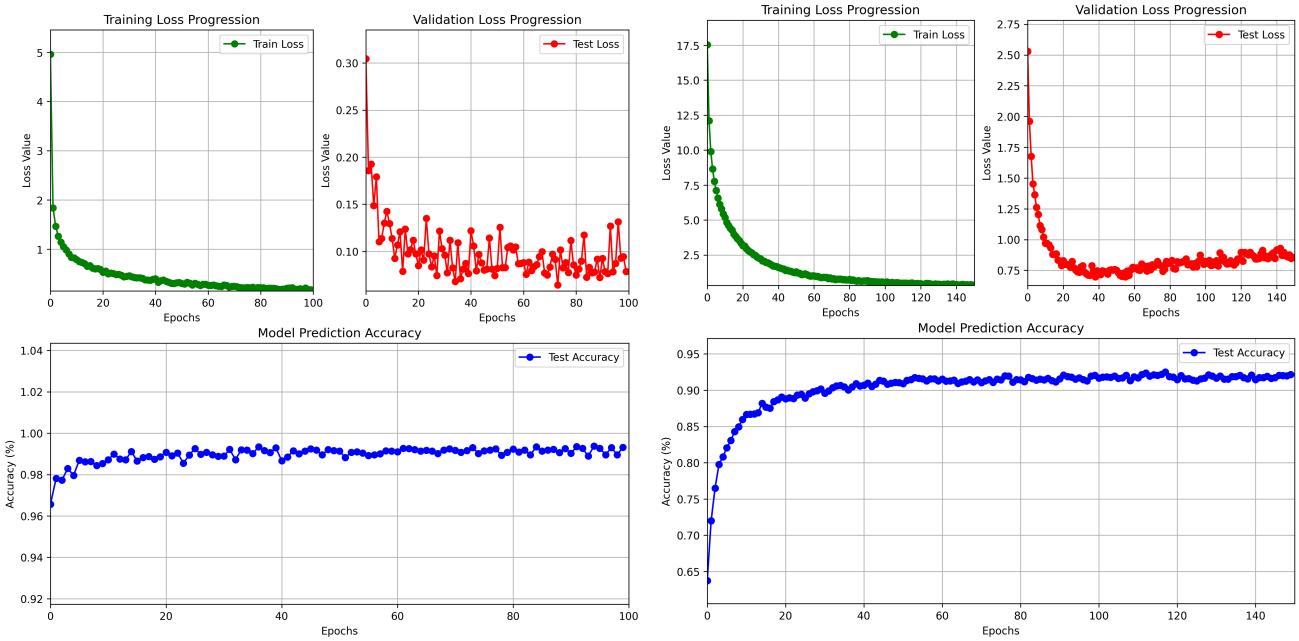

Figure 11: Basline(left), ResNet18(right) from Task A

Conclusion we can see from Table 4. that the reproduced training and testing on MNIST of our self-made framework has converge to a fairly close testing accuracy: 98.9%+ to that of Task A (torch).

However, see Figure 10 and Figure 11, the testing loss of our framewrok is more volitile that Task A, i think this might because:

- Differences of default weights initialization: PyTorch's convolutional layers default to Kaiming Normal initialization, while my self-made one simply using `np.random.rand`
- Differences in hardware and parallelization algo: PyTorch may leverage more optimized cuDNN algorithms for convolutions, while our framework simply replace `numpy` with `cupy`

What have i learnt

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”).