REPORT： “Neural Network and Deep Learning”

23300200022 滕勇功

Due to 8:00 AM, May 1, 2025.

model & codes & dataset link :

- https://github.com/tyyyyy333/Fdu_DS_DL24spring_collection/tree/main/project1

1. Project Overview

This project aims to implement a simple neural network framework using NumPy from scratch, and to train it on the MNIST dataset for image classification.

2. Implement Details

2.1 General MLP Structure:

$$\underbrace{Linear(m,n)}_{customiaed}$$
$$->activation('ReLU',' Sigmoid',' LeakyReLU')$$

$$->Linear(n_k, 10)->Loss\_fn$$

where m, n refer to units-in, units-out

2.2 CNN:

$$\underbrace{Conv2D(m, n, k, o, p, q)}_{customized}$$
$$->activation('ReLU', 'Sigmoid', 'LeakyReLU')$$
$$->flatten$$
$$->\underbrace{Linear(m^*, n^*)}_{customized}$$
$$->activation('ReLU', 'Sigmoid', 'LeakyReLU')$$
$$->Linear(n\_k, 10)->Loss\_fn$$

where m, n, k,o ,p , q respectively refer to the inchannel, outchannel, kernel_size, strides, paddings

2.3 Implement and Derivation:

2.3.1 Linear:

$$\frac{\partial \mathrm{L}}{\partial x} = \frac{\partial \mathrm{L}}{\partial y}\frac{\partial y}{\partial x} = grads \cdot W^T$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W}} = \frac{\partial \mathrm{L}}{\partial y}\frac{\partial y}{\partial \mathrm{W}} = x^T \cdot grads$$

$$\frac{\partial \mathrm{L}}{\partial b} = \frac{\partial \mathrm{L}}{\partial y}\frac{\partial y}{\partial b} = grads$$

- Here comes the codes:

```
|解释|添加注释|×
def backward(self, grad : np.ndarray):     "ndarray": Unknown word.
    """
    input: [batch_size, out_dim] the grad passed by the next layer.
    output: [batch_size, in_dim] the grad to be passed to the previous layer.
    This function also calculates the grads for W and b.
    """
    self.grads['W'] = np.dot(self.input.T, grad) / self.input.shape[0]
    self.grads['b'] = np.sum(grad, axis=0, keepdims=True) / self.input.shape[0]
    #print(self.input.T.shape, grad.shape)
    #print('-------------------------------------')
    out = np.dot(grad, self.W.T)

    return out

|解释|添加注释|×
def clear_grad(self):
    self.grads = {'W' : None, 'b' : None}
```

2.3.2 CNN:

$$\frac{\partial \mathrm{L}}{\partial X} = \sum_{o=1}^{C_{out}} \delta_o \star \mathrm{rot180}(W_o)$$

$$\frac{\partial \mathrm{L}}{\partial W^{(o)}} = \sum_{n=1}^{N} \sum_{i,j} \delta_o^{(n)}(i,j) \cdot X^{(n)}_{[:,i:i+K,j:j+K]}$$

$$\frac{\partial \mathrm{L}}{\partial b_o} = \sum_{n=1}^{N} \sum_{i,j} \delta_o^{(n)}(i,j)$$

- Here comes the codes:

```python
    grads : [batch_size, out_channel, new_h, new_w]
    """
    batch_size = grads.shape[0]
    X = self.input

    self.grads['W'] = np.zeros_like(self.W_)
    self.grads['b'] = np.zeros_like(self.b_)
    dX = np.zeros_like(X, dtype=np.float64)

    for i in range(grads.shape[2]):
        for j in range(grads.shape[3]):
            h_start = i * self.stride
            h_end = h_start + self.kernel_size
            w_start = j * self.stride
            w_end = w_start + self.kernel_size

            X_window = X[:, :, h_start:h_end, w_start:w_end]

            for o in range(grads.shape[1]):
                delta = grads[:, o, i, j][:, None, None, None]
                self.grads['W'][o] += np.sum(delta * X_window, axis=0)
                self.grads['b'][0, o, 0, 0] += np.sum(grads[:, o, i, j])
                dX[:, :, h_start:h_end, w_start:w_end] += delta * self.W_[o]

    self.grads['W'] /= batch_size
    self.grads['b'] /= batch_size

    if self.padding > 0:
        dX = dX[:, :, self.padding:-self.padding, self.padding:-self.padding]

    return dX
```

2.3.3 flatten:

```
To flatten the feature map into***[batch_size, features]*** shape to calculate by
linear layer.
```

- Here comes the codes:


```python
    grads : [batch_size, out_channel, new_h, new_w]
    """
    batch_size = grads.shape[0]
    X = self.input

    self.grads['W'] = np.zeros_like(self.W_)
    self.grads['b'] = np.zeros_like(self.b_)
    dX = np.zeros_like(X, dtype=np.float64)

    for i in range(grads.shape[2]):
        for j in range(grads.shape[3]):
            h_start = i * self.stride
            h_end = h_start + self.kernel_size
            w_start = j * self.stride
            w_end = w_start + self.kernel_size

            X_window = X[:, :, h_start:h_end, w_start:w_end]

            for o in range(grads.shape[1]):
                delta = grads[:, o, i, j][:, None, None, None]
                self.grads['W'][o] += np.sum(delta * X_window, axis=0)
                self.grads['b'][0, o, 0, 0] += np.sum(grads[:, o, i, j])
                dX[:, :, h_start:h_end, w_start:w_end] += delta * self.W_[o]

    self.grads['W'] /= batch_size
    self.grads['b'] /= batch_size

    if self.padding > 0:
        dX = dX[:, :, self.padding:-self.padding, self.padding:-self.padding]

    return dX
```


2.3.4 MutiCrossEntropy:

```
softmax :$\hat{y}_i = \frac{e^{z_i - z_{max}}}{\sum_{j=1}^{n} e^{z_j - z_{max}}}$

- forward:
```

$$L = -\frac{1}{B}\sum_{j=1}^{B}\sum_{i=1}^{n_{class}} y_i^j * \log(softmax(\hat{y}_i^j) + \epsilon)$$

```
- backward:
```

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{1}{B}\sum_{j=1}^{B}(softmax(\hat{y}_i^{(j)}) - y_i^{(j)})$$

- Here comes the codes:

```python
    def forward(self, predicts, labels):
        """
        predicts: [batch_size, D]
        labels : [batch_size, ]
        This function generates the loss.
        """
        # / ---- your codes here ----/

        self.labels = labels
        if self.has_softmax:
            self.pred = softmax(predicts)
        else:
            self.pred = predicts
        one_hot = np.eye(self.max_classes)[self.labels]
        eps = 1e-8
        loss = -np.mean(np.sum(one_hot * np.log(self.pred + eps), axis=1))

        return loss

    | 解释 | 添加注释 | ×
    def backward(self):
        # first compute the grads from the loss to the input
        # / ---- your codes here ----/
        one_hot = np.eye(self.max_classes)[self.labels]
        self.grads = (self.pred - one_hot)
        # Then send the grads to model for back propagation
        self.model.backward(self.grads)

    | 解释 | 添加注释 | ×
    def cancel_soft_max(self):
        self.has_softmax = False
        return self
```

2.3.5 sigmoid:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

```
- backward:
```

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = grads * y * (1 - y)$$

- Here comes the codes:

```python
def forward(self, X):
    self.input = X
    output = 1 / (1 + np.exp(-X))
    return output

def backward(self, grads):
    assert self.input.shape == grads.shape
    y = self.forward(self.input)

    output = grads * y * (1 - y)
    return output
```

2.3.6 Other functions' implement codes:

```
- L2 penalty:
```

```python
def forward(self, loss):
    w_sum = 0
    for layer in self.model.layers:
        if layer.optimizable:
            w_sum += np.sum(layer.params['W']**2)
    loss += self.lambda_ * w_sum
    return loss
```

```
- backward of L2 Loss:
```

```
    <∕>∨ | 解释 | 添加注释 | ×
    def forward(self, predicts, labels):
        if self.one_hot:
            self.labels = np.eye(self.max_classes)[labels]
        else:
            self.labels = labels

        if self.has_softmax:
            self.pred = softmax(predicts)
        else:
            self.pred = predicts

        loss = np.mean((self.pred - self.labels)**2)

        return loss

    <∕>∨ | 解释 | 添加注释 | ×
    def backward(self):
        self.grads = 2 * (self.pred - self.labels)
        self.model.backward(self.grads)
```

3. Experiment Setup

```
- learning rate : 1e-3 (default)
- optimizer : Adam / Momentum SGD / SGD (default)
- learning rate scheduler : None(default) / MultiStep / Exponential
- scaler : min-max-scaler
- loss function : MultiCrossEntropy(default) / MseLoss
- batch size : 128(default)
- dataset : Mnist
- random seed : 309
- running epoch : 10
- Weight decay : shut down(default)
```

4. Result

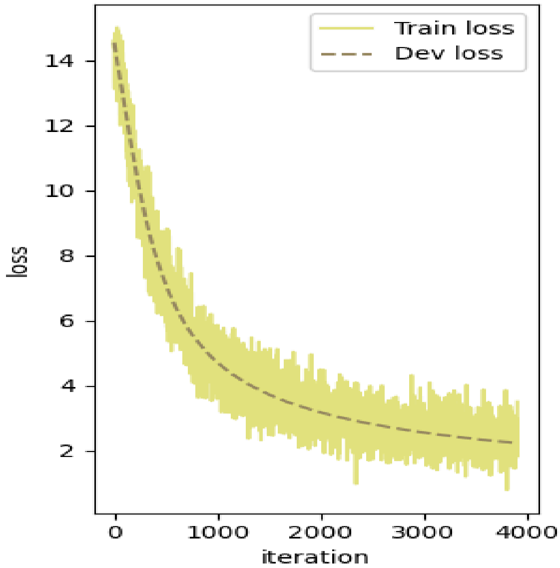
Unless otherwise specified, all the following results are obtained from the default settings in 3.

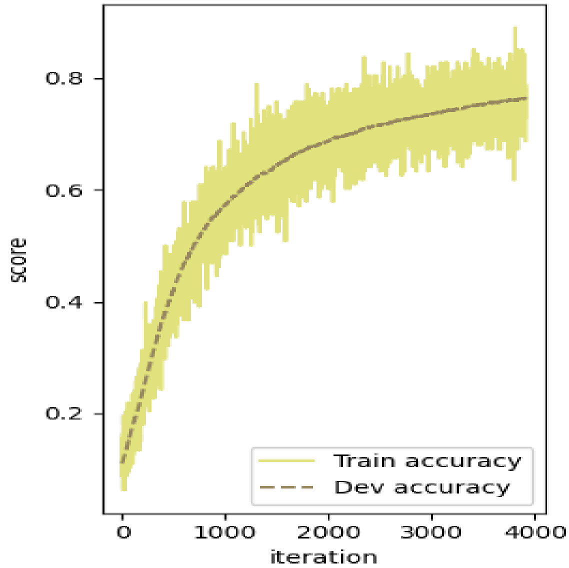
4.1 MLP

4.1.1 Metrics:

Defaulted setting

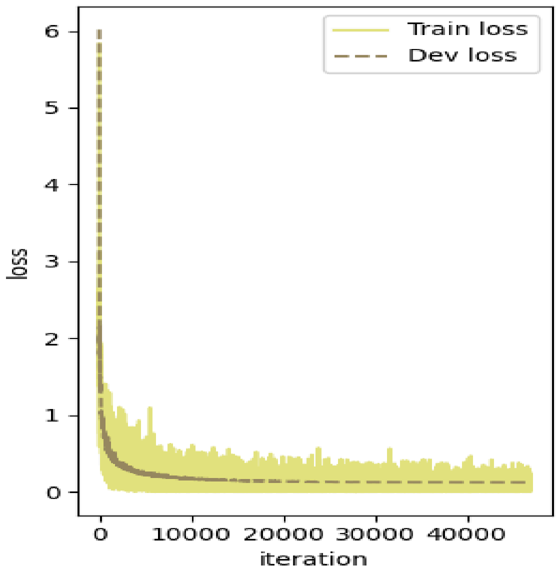
- valid Accuracy: 0.7634
- valid loss: 2.234
- loss graph as follows:

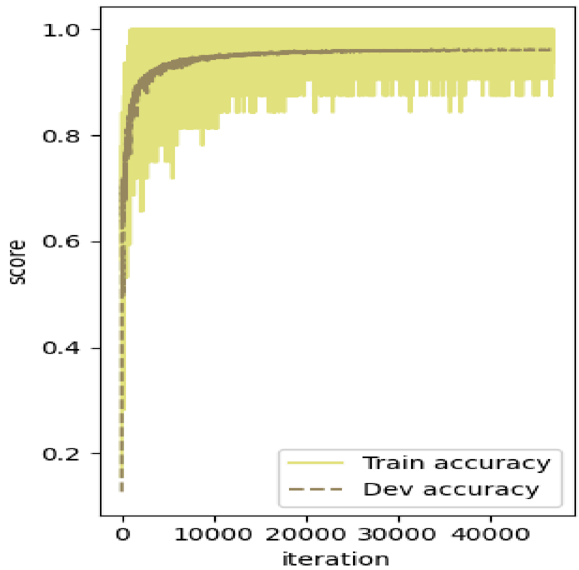




Developed setting

- valid Accuracy: 0.9535
- valid loss: 0.1541
- Setting seen in Appendix.
- loss graph as follows:



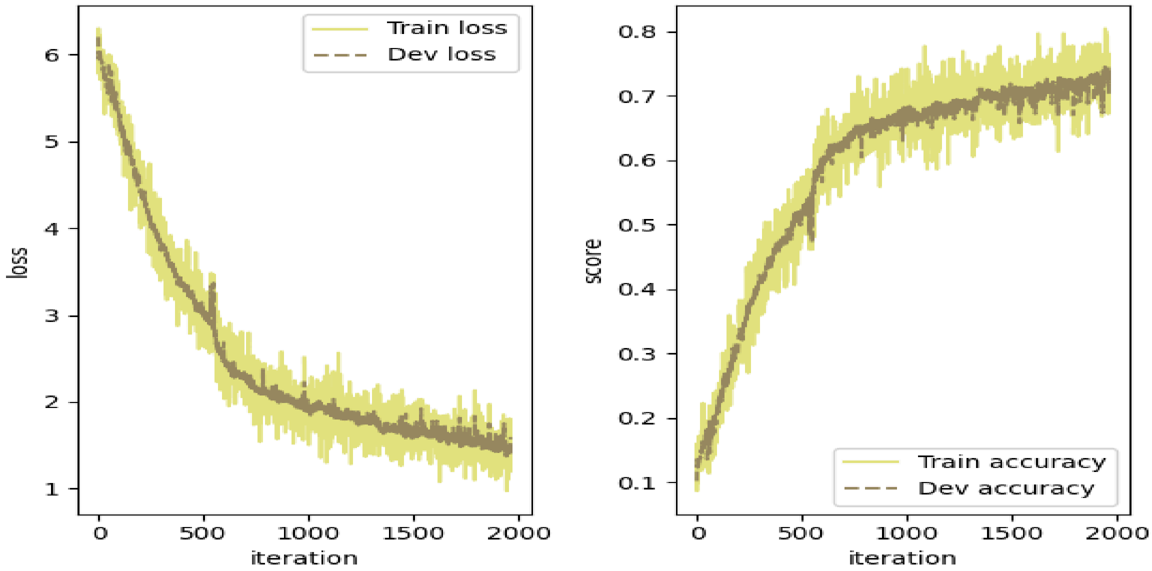


4.2 CNN

4.2.1 Metrics:

Defaulted setting

- Note : The accuracy can be worse than mlp when the cnn model is not well trained and designed.
- valid Accuracy: 0.7023
- valid loss: 1.5977
- loss graph as follows:



4.2.2 A note for the result of cnn:

```
Due to the efficiency of the CNN layer implementation, this CNN model cannot achieve
too many and too deep cnn kernel settings(seen as Appendix). Similarly, the
operation time required to ensure convergence is also too long. Therefore, the
accuracy is lower than that of MLP. Under ideal conditions, CNN can carry more
economical parameters, and the training effect should be better than MLP ideally.
```

4.3 Result of changing hyperparameters:

For convenience, following experiments are based on the defaulted MLP model to save the calculating time. Both defaulted details of MLP and CNN will be written in Appendix.

4.3.1 modifying layers:

```
Modify the layer units into {[784,500,100,10], [784,100,10], [784,500,10]}
while shut down the weight decay.
The activations are all 'ReLU' and the loss function is 'MultiCrossEntropy'.
[784,500,100,10] :
    valid accuary : 0.8541
    valid loss : 1.8914
[784,100,10] :
```

```
    valid accuracy : 0.6799
    valid loss : 1.967

[784,500,10] :
    valid accuracy : 0.7898
    valid loss : 2.262
```

4.3.2 learning rate:

```
Modify the learning rate into {1e-4, 1e-2, 1, 10}
using the lr_scheduler of 'MultiStep', 'Exponential' and 'None'.
All the valid accuracy and loss are based on the final epoch.
[1e-4, None] :
    valid accuracy : 0.3678
    valid loss : 8.1202
[1e-2, None] :
    valid accuracy : 0.8898
    valid loss : 0.7872
[1, None] :
    valid accuracy : 0.9584
    valid loss : 0.1660
Here the training accuracy is around 0.95
[10, None] :
    valid accuracy : 0.0998
    valid loss : 2.3954
[1, MultiStep] :
    The milestones are [1000, 2000] and the gamma is 0.1
    valid accuracy : 0.9439
    valid loss : 0.2012
However, the lowest training loss is around 0.05 and the training accuracy is nearly
0.98
[1, Exponential] :
    THE gamma is 0.999 and the lowest learning rate is 1e-5
    valid accuracy : 0.8558
    valid loss : 1.8909
However, the lowest training loss is around 0.08 and the training accuracy is nearly
0.98
```

4.3.3 optimizers:

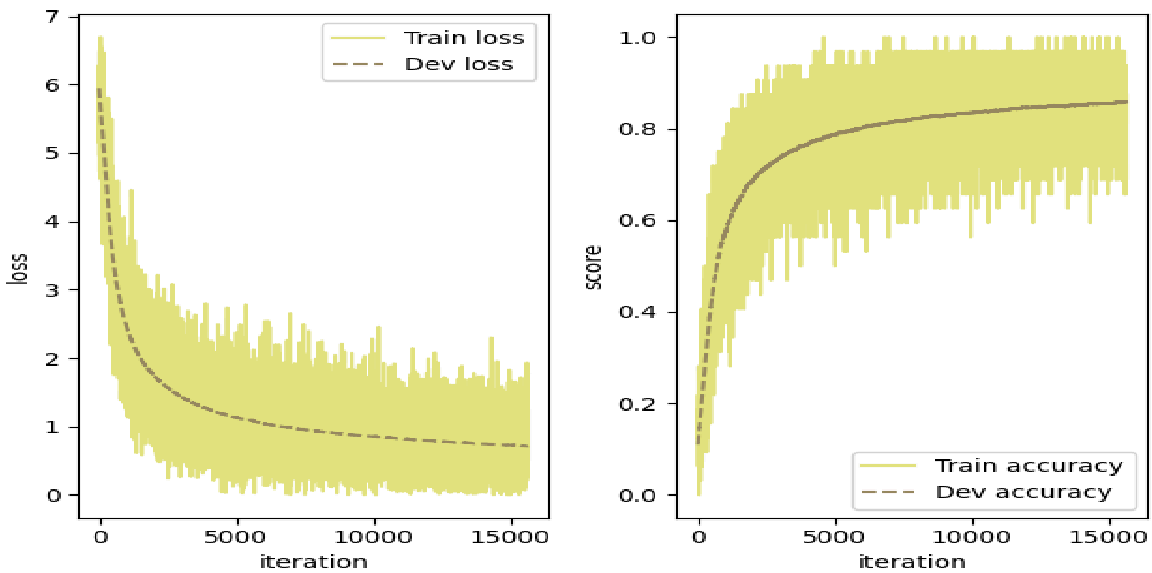
```
Modify the optimizer into {'Adam', 'Momentum SGD', 'SGD'}
learning rate is 1e-3 as defaulted.
SGD :
    valid Accuracy: 0.7634
    valid loss: 2.234
Momentum SGD :
The mu is 0.9.
    valid Accuracy: 0.7644
```

```
    valid loss: 2.2335
Adam :
The beta_1 is 0.9, beta_2 is 0.999, epsilon is 1e-8.
    valid Accuracy: 0.9327
    valid loss: 0.4679

Actually, losses optimized by SGD and MomentumGD are not converged because of learning rate.
The Adam optimizer is the best one among them, for its self-adapting learning rate.
```

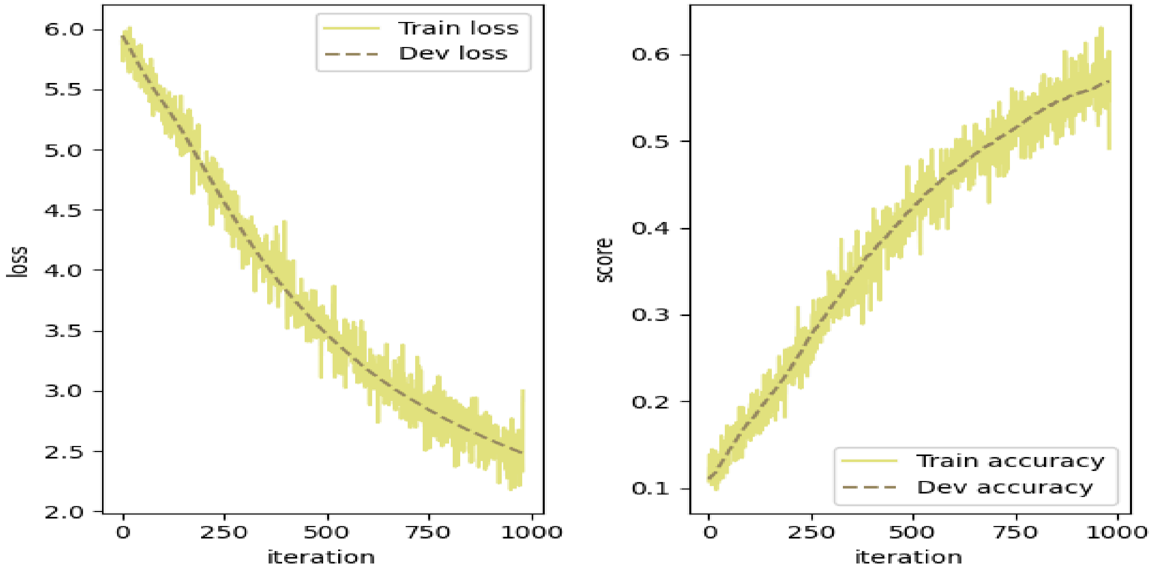
4.3.4 batch size:

```
Modify the batch size with {32, 128, 512}
others are defaulted
32 :
    valid Accuracy: 0.8573
    valid loss: 0.7167
Here comes the loss tendency:
```




```
256(Defaulted) :
    valid Accuracy: 0.7634
    valid loss: 2.234

512 :
    valid Accuracy: 0.5688
    valid loss: 2.4836
Here comes the loss tendency:
```




```
The batch size of 32 is the best one, for it has the lowest loss and the highest
accuracy.
The batch size of 512 is the worst one, for it has the highest loss and the lowest
accuracy.
But with respect to the running time, the batch size of 512 is the best one, for it
has the lowest running time.
```

4.3.5 loss_fn:

```
Modify the loss function with {'MultiCrossEntropy', 'MseLoss'}
others are defaulted
MultiCrossEntropy(Defaulted) :
    valid Accuracy: 0.7634
    valid loss: 2.234
MseLoss :
    valid Accuracy: 0.7641
    valid loss: 0.0426(mse)
```

4.3.6 data augmentation:

```
Modify the data augmentation a sequence of {'translate', 'rotate', 'random resize'}
Honestly, data augmentation is not so suited for MLP, but the manual CNN is too
slow...
So here put the result as an illustration.
others are defaulted
with augmentation :
    valid Accuracy: 0.6704
    valid loss: 1.8192
without augmentation :
```

```
    valid Accuracy: 0.6846
    valid loss: 1.7559
```

4.3.7 regularization:

```
Modify the regularization with 'l2' , dropout, and weight_decay
others are defaulted
Consider that applying penalty may cause worse behavior, here we modify the learning
rate to the best "1" as observed and apply the ExponentialLR with lowest lr of 1e-4.
we modify the epoch to 20 for sufficient training.

with l2 :
The l2 lambda will be 1e-3
    valid Accuracy: 0.9644
    valid loss: 1.3579
with dropout :
    The drop rate is 0.1
    valid Accuracy: 0.8884
    valid loss: 0.3245
This may be a potential issue.
Dropout essentially introduces the noise, which obstructs the convergence.
with weight_decay :
    lambda will be 1e-3
    valid Accuracy: 0.962
    valid loss: 0.1277
defaulted :
    valid Accuracy: 0.9442
    valid loss: 0.1831
```

5.Error Analysis

5.1

```
The model is not accessible for the gradient update because of the following
code:`layer.params[key] = layer.params[key] - self.init_lr *
self.v[layer.layer_name][key]`
in the `optimizer.py` due to the fact that the assignment creates a new object
reference rather than modifying the original parameter in place.

As a result, the`layer.params[key]` in the model is disconnected from the
optimizer's update mechanism.
```

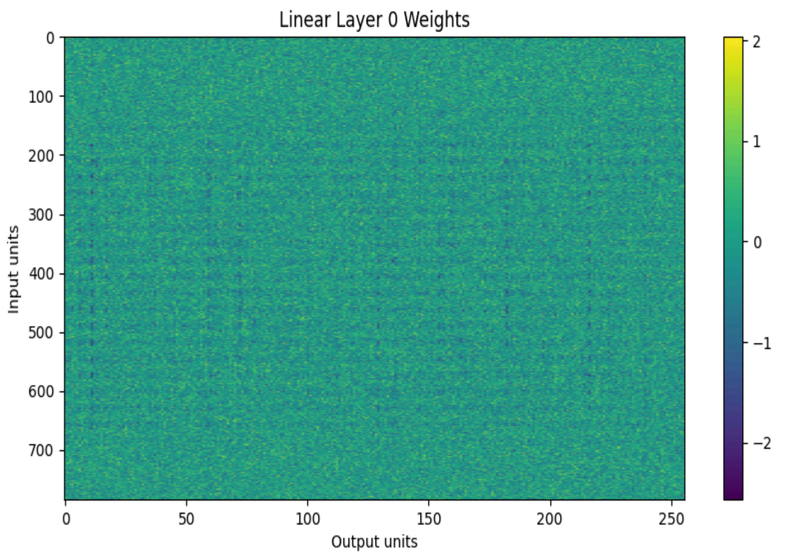
This leads to no actual parameter update during training, even though the gradients are correctly computed.

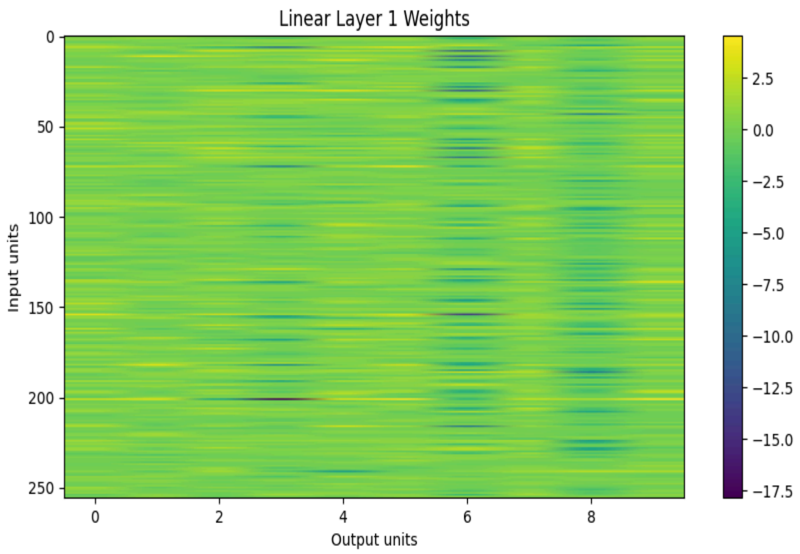
```
To fix this issue, the in-place update should be used instead:`layer.params[key] -= self.init_lr * self.v[layer.layer_name][key]`
```

6.Feature Visualization

```
Here comes the visualization weight.
```

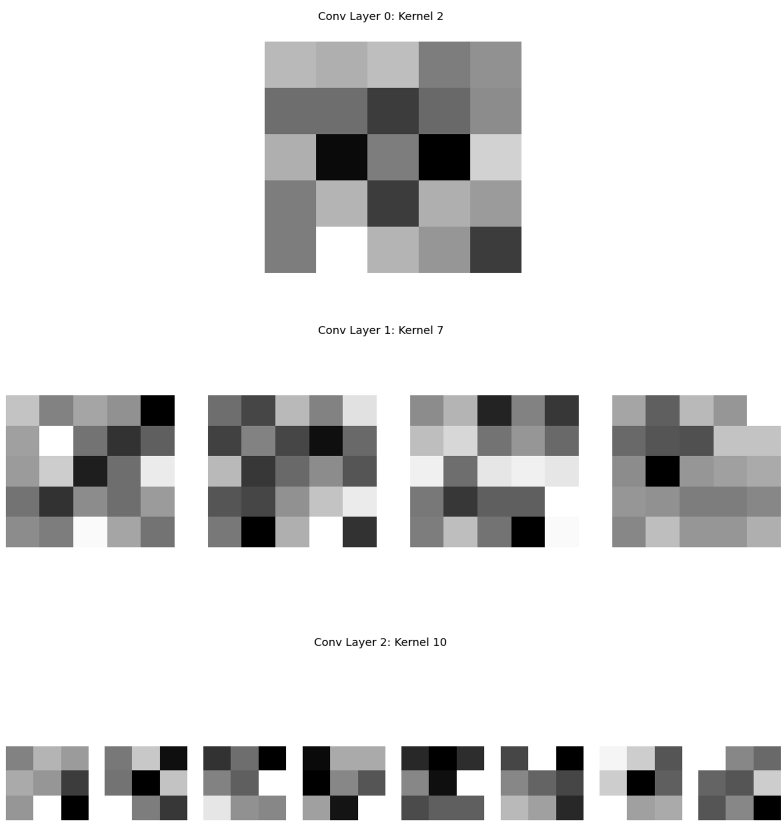
6.1 MLP:

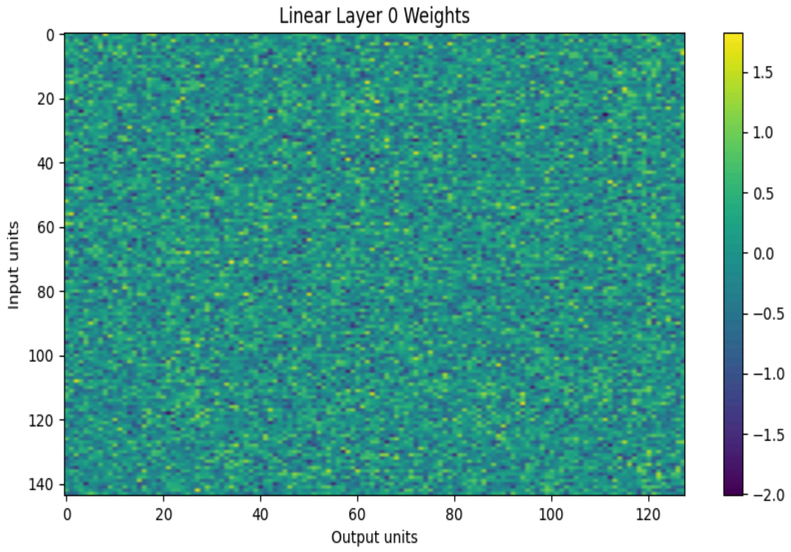


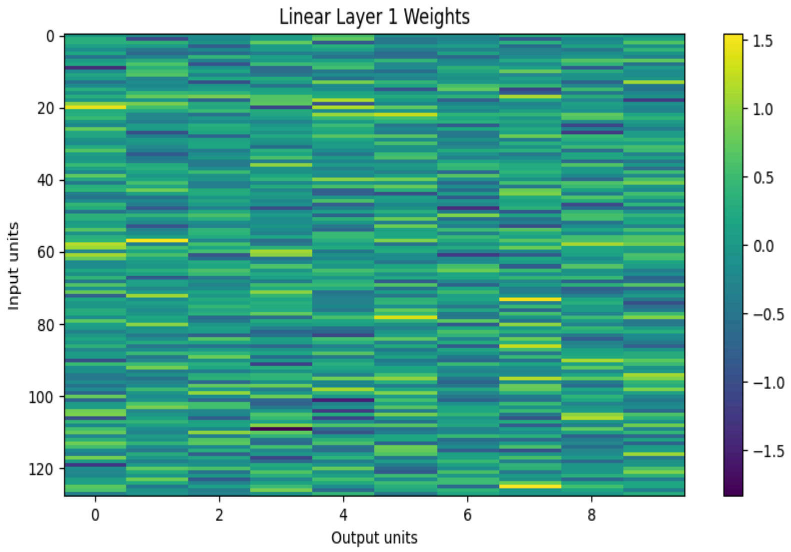


6.2 CNN:

Here shows some visualization of the CNN kernels. A complete visualization of the kernels is too large to be displayed. So here we only show a few kernels of each layer. Others will be uploaded in github.







7.Conclusion

7.1. Comparison of Optimizers

```
* We compared several optimizers including SGD, SGD with momentum, and Adam. We
found that Adam consistently provides faster convergence due to its adaptive
learning rate mechanism, especially in the early stages of training. While SGD with
momentum can also accelerate convergence,but it requires more careful tuning. Pure
SGD, although simple, is more sensitive to the learning rate and generally slower.
For our task, Adam achieved the best trade-off between speed and stability. *
```

7.2 Impact of Network Depth and Neurons

```
* Deeper networks or those with more neurons per layer have greater expressive
power, but they are also more prone to overfitting and harder to train without
regularization. In our experiments, increasing the number of hidden units in MLP
```

```
improved accuracy up to a point, after which diminishing returns.In contrast, deeper
net is more effective for optmizing. *
```

7.3 Effectiveness of Regularization

```
* We applied both L2 regularization, dropout and weight decay to prevent
overfitting. L2 and weight decay help to keep the weights from growing too large,
while dropout forces the network to develop more robust, redundant representations.
Both methods improves generalization, especially for MLPs, which otherwise easily
overfit the MNIST dataset. However, excessive regularization can lead to
underfitting, especially when using shallow networks. As a sample, dropout here
plays a worse role in training for its offensive behavior, which is adding noise in
essence. *
```

7.4 Benefits of Cross-Entropy Loss

```
* Using softmax combined with cross-entropy loss provided a probabilistic
interpretation of the model's output and a smooth, numerically stable loss surface.
We confirmed that this formulation simplifies gradient computation and ensures
better class separation during training compared to mean squared error (MSE), which
performed worse in early experiments due to less sharp gradient signals. (although
we have not been observed the difference of training loss from mse yet in
experiments...due to the fault of the configuration for hyperparameters) *
```

7.5 Convolutional Network Performance and Limitations

```
* The convolution layers are able to extract local and hierarchical patterns such as
edges and textures, which the MLP could not learn effectively. However, our self-
built CNN performs slightly worse than reference implementations, likely due to a
limited number of convolutional filters. This constraint is imposed due to running
time limitations for the CPU. A smaller number of filters restricts the network's
feature representation capacity, especially in deeper layers. *
```

7.6 Insights from Visualization

```
* Visualizing the learned convolutional filters revealed that early layers typically
detect edges or gradients, while deeper layers captured more abstract shapes.
Feature map visualization helped confirm that the CNN progressively distilled the
input image into more discriminative representations. *
```

7.7 Hyperparameter Tuning Effects

```
* Through systematic tuning of hyperparameters such as learning rate, batch size,
number of filters, and dropout rate, we observed several important dynamics *:

- Too large a learning rate led to oscillations or divergence, while too small a
rate slowed convergence significantly.

- Small batch sizes improved generalization but made the training process noisier.

- Increasing the number of filters or hidden units enhanced accuracy but also raised
computational cost and overfitting risk.

- regularization methods helped regularize the model, but if set too high, harmed
learning by removing too much information.
```

Overall Learning Outcome

```
* Building a deep learning model from scratch provided us with invaluable insights
into the architecture and training mechanics of neural networks. We learned how each
component—from activation functions to gradient updates—affects the final
performance. While our CNN was relatively shallow and constrained by resources, the
implementation process deepened our understanding of feature extraction, training
dynamics, and model interpretability. This hands-on experience clarified not just
the "how," but also the "why" behind deep learning models' behavior. *
```

8.Appendix

8.1 Details of MLP:

Default:
Linear 1 : (784, 256)
->ReLU 1
->Linear 2 : (256, 10)
->Softmax + MultiCrossEntropy
Default setting

Developed:
Linear 1 : (784, 512) weight_decay=0.001
->ReLU 1
->Linear 2 : (512, 256) weight_decay=0.001

->ReLU 2
->Linear 3 : (256, 10) weight_decay=0.001
->Softmax + MultiCrossEntropy

```
learning_rate = 1e-1
batch_size = 32
epochs = 15
optimizer = MomentumGD
lr_scheduler = ExponentialLR(gamma=0.9999,lowest_lr=7e-4)
```

8.2 Details of CNN:

Default:

```
Conv 1 : (1 → 4), kernel_size=5, stride=2, padding=1
```

->ReLU
->Conv 2 : (4 → 8), kernel_size=5, stride=2, padding=1
->ReLU
->Conv 3 : (8 → 16), kernel_size=3, stride=2, padding=1
->ReLU
→Flatten
->Linear 1 : (16 * 3 * 3 → 128)
->ReLU
->Linear 2 : (128 → 10)
->Softmax + MultiCrossEntropy
Default setting

8.3 statement of scores and weight availability

Consider that multiple parameters change in the experiment, the model weights in the netdisk or github will be reserverd only for a basic version and a seemingly best one(only for mlp...) as declaration of model availability, and all the valid scores documented in this markdown are the behavior for the final epoch of a training process, rather than representing the behavior of the model weight, which documents the best one. All the setting will be defaulted unless declaring.