PJ1 Report

Chen Hao 22307110062

April 30, 2025

Abstract

Implementation can be found in the codes at `https://github.com/Laplx/DLPJ` and will also be gradually explained in the following answers. The *best_models* folder contains several model configurations, each corresponding to the specific number of question and the unsuffixed best_model is the final selected optimal version.

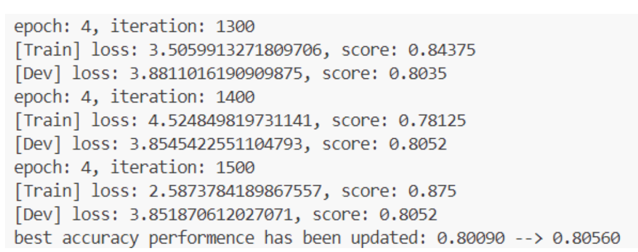
1 Question1: MLP Structure

A full implementation of the original setting of MLP model performs normally well and has a score over 0.80.

```
linear_model = nn.models.Model_MLP([train_imgs.shape[-1], 600, 10],
    'ReLU', [1e-4, 1e-4])
optimizer = nn.optimizer.SGD(init_lr=0.06, model=linear_model)
scheduler = nn.lr_scheduler.MultiStepLR(optimizer=optimizer,
    milestones=[800, 2400, 4000], gamma=0.5)
```

Some modifications are:

- Expand the dimension of hidden layer from 600 to 1024, which promote the accuracy slightly by enhancing the information processing.
- Deepen the network shows a better result, while we also change other hyperparameters including weight_decay and learning rate control.



```
epoch: 4, iteration: 1300
[Train] loss: 3.5059913271809706, score: 0.84375
[Dev] loss: 3.8811016190909875, score: 0.8035
epoch: 4, iteration: 1400
[Train] loss: 4.524849819731141, score: 0.78125
[Dev] loss: 3.8545422551104793, score: 0.8052
epoch: 4, iteration: 1500
[Train] loss: 2.5873784189867557, score: 0.875
[Dev] loss: 3.851870612027071, score: 0.8052
best accuracy performence has been updated: 0.80090 --> 0.80560
```

Figure 1

```
epoch: 4, iteration: 1300
[Train] loss: 3.9366823571447345, score: 0.8125
[Dev] loss: 3.374622380281188, score: 0.8323
epoch: 4, iteration: 1400
[Train] loss: 3.7820186742139996, score: 0.78125
[Dev] loss: 3.3713858138101087, score: 0.8327
epoch: 4, iteration: 1500
[Train] loss: 4.8604874421764634, score: 0.78125
[Dev] loss: 3.368807692862332, score: 0.8342
best accuracy performence has been updated: 0.83000 --> 0.83360
```

Figure 2

```
linear_model = nn.models.Model_MLP([train_imgs.shape[-1],
    512, 256, 10], 'ReLU', [5e-4, 5e-4, 5e-4])
optimizer = nn.optimizer.SGD(init_lr=0.1, model=linear_model)
scheduler = nn.lr_scheduler.MultiStepLR(optimizer=optimizer,
    milestones=[300, 900, 1800], gamma=0.3)
```

2 Question2: Different Optimizer

According to the momentum gradient descent, we implement MomentGD: (Note for CNN, 'self.W' may be overwritten by the latter layers so distinguish them.)

```
class MomentGD(Optimizer):
  def __init__(self, init_lr, model, mu=0.9):
  super().__init__(init_lr, model)
  self.mu = mu
  self.v = {}

  # Avoid same names of params in different layers
  for layer_idx, layer in enumerate(self.model.layers):
    if layer.optimizable:
      for key in layer.params.keys():
        unique_key = f"layer{layer_idx}_{key}"
        self.v[unique_key] = np.zeros_like(layer.params[key])

  def step(self):
    for layer_idx, layer in enumerate(self.model.layers):
      if layer.optimizable:
        for key in layer.params.keys():
          unique_key = f"layer{layer_idx}_{key}"
          self.v[unique_key] = self.mu * self.v[unique_key] -
              self.init_lr * layer.grads[key]
          if layer.weight_decay:
            layer.params[key] *= (1 - self.init_lr *
                layer.weight_decay_lambda)
            layer.params[key] = layer.params[key] +
```

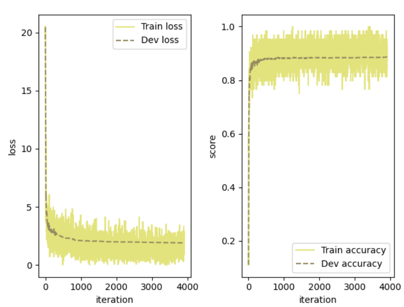
```
            self.v[unique_key]
        if hasattr(layer, 'sync_params'):
          layer.sync_params()
```

where we use previous settings and set *batch_size* to 64, under MomentGD, to get a score higher than 0.88.

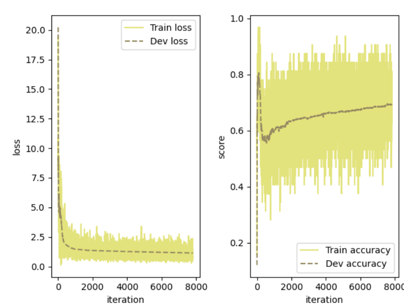
```
epoch: 4, iteration: 500
[Train] loss: 1.1199157386450245, score: 0.890625
[Dev] loss: 1.9135549208739124, score: 0.8849
epoch: 4, iteration: 600
[Train] loss: 0.6454234913735916, score: 0.9375
[Dev] loss: 1.9108263360708964, score: 0.8849
epoch: 4, iteration: 700
[Train] loss: 1.7268473106006428, score: 0.90625
[Dev] loss: 1.9065082148794459, score: 0.8859
best accuracy performence has been updated: 0.88460 --> 0.88560
```

Figure 3

We also exam the case when *batch_size* is 32 and it yields a unsatisfactory outcome. The following two loss graphs show the difference clearly. (Under momentum, small batches may be too noisy to learn.)



(a)



(b)

3 Question3: Regularization

L2 regularization(equivalent to *weight_decay*) and dropout layer has been fulfilled.

```
class L2Regularization(Layer):
  ...
  def forward(self, predicts, labels):
    loss = self.loss_fn(predicts, labels)
    for layer in self.model.layers:
    if hasattr(layer, 'weight_decay'):
      if layer.weight_decay:
```

```
            loss += np.sum(layer.W ** 2) * self.lambda_ / 2.0
    return loss

    def backward(self):
        self.loss_fn.backward()
        for layer in self.model.layers:
            if hasattr(layer, 'weight_decay'):
                if layer.weight_decay:
                    layer.grads['W'] += layer.W * self.lambda_

class Dropout(Layer):
    ...
    def forward(self, X):
        self.mask = np.random.binomial(1, 1-self.p, size=X.shape) /
            (1-self.p)
        return X * self.mask

    def backward(self, grads):
        return grads * self.mask
```

L2 penalty has already been used in the model training process and the effect of dropout method(realized by class *Model_MLP_Dropout* that adds dropout layers after each activation functions despite the last one) see bleow(with same params). Its loss graph has the similar pattern and can be found in *figs* folder.

```
epoch: 4, iteration: 1300
[Train] loss: 1.4391156831212786, score: 0.9375
[Dev] loss: 2.5230513562264347, score: 0.8872
epoch: 4, iteration: 1400
[Train] loss: 2.1586735246819178, score: 0.90625
[Dev] loss: 2.516930501990438, score: 0.8884
epoch: 4, iteration: 1500
[Train] loss: 2.878231366242557, score: 0.875
[Dev] loss: 2.5266903356960224, score: 0.8872
best accuracy performence has been updated: 0.88540 --> 0.88700
```

Figure 4

Schedulers(MultiStepLR, ExponentialLR) are also implemented and used in the codes.

4 Question4: Cross-Entropy Loss

Refer to the annotation in the following codes for the calculation formulas(obtained by taking derivatives of loss function wrt. p_{i,y_i}).

```
class MultiCrossEntropyLoss(Layer):
    ...
    self.eps = 1e-10 # to avoid log(0)
    ...
```

```
  def forward(self, predicts, labels):
    assert predicts.shape[0] == labels.shape[0], "The batch size of
        predicts and labels should be the same."
    assert predicts.shape[1] == self.max_classes, "The number of
        classes should be the same."
    self.grads = np.zeros_like(predicts)
    if self.has_softmax:
      predicts = softmax(predicts)
      self.predicts = predicts
      self.labels = labels

    selected_probs = predicts[np.arange(predicts.shape[0]), labels]
    loss = np.sum(-np.log(np.clip(selected_probs, self.eps, 1.0))) /
        predicts.shape[0]

    return loss

  def backward(self):
    batch_size = self.predicts.shape[0]

    if self.has_softmax:
    # Softmax + CrossEntropy: grads = (p - one_hot(labels)) /
        batch_size
      one_hot_labels = np.zeros_like(self.grads)
      one_hot_labels[np.arange(batch_size), self.labels] = 1
      self.grads = (self.predicts - one_hot_labels) / batch_size
    else:
    # CrossEntropy only: grads = (-1/p[labels]) / batch_size for
        correct class, 0 otherwise
      self.grads.fill(0)
      self.grads[np.arange(batch_size), self.labels] = -1.0 /
          np.clip(self.predicts[np.arange(batch_size), self.labels],
          self.eps, 1.0)
      self.grads /= batch_size

    self.model.backward(self.grads)

  def cancel_soft_max(self):
    self.has_softmax = False
    return self
```

5 Question5: CNN Structure

```
class conv2D(Layer):
  ...
  def sync_params(self):
    self.W = self.params['W']
```

```
        self.b = self.params['b']

    def forward(self, X):
        self.input = X
        batch_size, in_channels, H, W = X.shape

        new_H = (H - self.k_H) // self.stride + 1 #! no padding
        new_W = (W - self.k_W) // self.stride + 1
        output = np.zeros((batch_size, self.out_channels, new_H, new_W))

        for i in range(new_H):
            for j in range(new_W):
                output[:, :, i, j] = np.tensordot(X[:, :, i*self.stride :
                    i*self.stride+self.k_H, j*self.stride :
                    j*self.stride+self.k_W], self.W, axes=([1, 2, 3], [1, 2,
                    3])) + self.b[0, :, 0, 0] #! no padding

        return output

    def backward(self, grads):

        batch_size, in_channels, H, W = self.input.shape
        _, out_channels, new_H, new_W = grads.shape

        dW = np.zeros_like(self.W)
        db = np.zeros_like(self.b)
        dX = np.zeros_like(self.input)

        for i in range(new_H):
            for j in range(new_W):
                input_slice = self.input[:, :, i*self.stride :
                    i*self.stride+self.k_H, j*self.stride :
                    j*self.stride+self.k_W]

        dW += np.tensordot(grads[:, :, i, j], input_slice, axes=([0], [0]))
        db += np.sum(grads[:, :, i, j], axis=0).reshape(self.b.shape)
        dX[:, :, i*self.stride : i*self.stride+self.k_H, j*self.stride :
            j*self.stride+self.k_W] += np.tensordot(grads[:, :, i, j],
            self.W, axes=([1], [0]))
        dW /= batch_size
        db /= batch_size
        dX /= batch_size

        # Apply L2 regularization to dW if enabled
        if self.weight_decay:
        dW += self.weight_decay_lambda * self.W

        self.grads['W'] = dW
        self.grads['b'] = db
        return dX
```

Notable points here: *sync_params* to update params during optimizer.step(), avoiding losing reference. And a good way to the gradient computation is checking dimensions of all tensors at each step.

Our CNN configuartion is (only one convolution layer with out_channel of 16, RELU between all layers):

```
linear_model = nn.models.Model_CNN(size_list=[1, 16, 256,
    10],lambda_list=[5e-4, 5e-4, 5e-4, 5e-4],kernel_size=[3, 3])
optimizer = nn.optimizer.SGD(init_lr=0.1,model=linear_model)
scheduler =
    nn.lr_scheduler.MultiStepLR(optimizer=optimizer,milestones=[800,
    2400, 4000],gamma=0.3)
runner = nn.runner.RunnerM(linear_model, optimizer, nn.metric.accuracy,
    loss_fn, batch_size=64, scheduler=scheduler)
```

Limited to the efficiency of cpu, long time is required for getting a trivial outcome. (The loss on test set decreases at first but gradually increases afterwards.)

```
epoch: 4, iteration: 100
[Train] loss: 3.4866668708589916, score: 0.125
[Dev] loss: 3.6152529524141603, score: 0.1157
epoch: 4, iteration: 200
[Train] loss: 3.542613226372711, score: 0.1328125
[Dev] loss: 3.564520317372663, score: 0.1157
epoch: 4, iteration: 300
[Train] loss: 3.4385684799182097, score: 0.109375
[Dev] loss: 3.514404609098541, score: 0.1159
```

Figure 5

Below is a modified CNN adding maxpool layer(kernel_size=(2, 2), stride=2) after all convolution layers, which raises the best score up to 0.37. Bottleneck structure is also implemented in the codes.

```
epoch: 4, iteration: 100
[Train] loss: 6.375962132075712, score: 0.3671875
[Dev] loss: 7.102077837661858, score: 0.3354
epoch: 4, iteration: 200
[Train] loss: 8.356719106550287, score: 0.3515625
[Dev] loss: 7.002302519633833, score: 0.3245
epoch: 4, iteration: 300
[Train] loss: 6.588466531498078, score: 0.3203125
[Dev] loss: 6.838307183642651, score: 0.3224
```

Figure 6

6 Question6: Data Augmentation

We adopt a variety of tranformation including shift, rotation and scale.

```
def augment_images(images, labels, num_augmentations=2):
  augmented_images = []
  augmented_labels = []

  for img, label in zip(images, labels):
    img_2d = img.reshape(28, 28)

    augmented_images.append(img)
    augmented_labels.append(label)

    for _ in range(num_augmentations):
      shift_x = np.random.uniform(-2, 2)
      shift_y = np.random.uniform(-2, 2)
      shifted_img = shift(img_2d, [shift_y, shift_x],
          mode='nearest')

      angle = np.random.uniform(-10, 10)
      rotated_img = rotate(shifted_img, angle, reshape=False,
          mode='nearest')

      scale = np.random.uniform(0.9, 1.1)
      scaled_img = zoom(rotated_img, scale, mode='nearest')

      if scaled_img.shape != (28, 28):
        scaled_img = zoom(scaled_img, (28 / scaled_img.shape[0], 28
            / scaled_img.shape[1]), mode='nearest')

      augmented_images.append(scaled_img.flatten())
      augmented_labels.append(label)

  return np.array(augmented_images), np.array(augmented_labels)
```

with the following typical setting:

```
linear_model = nn.models.Model_MLP([train_imgs.shape[-1], 512, 256,
    10], 'ReLU', [5e-4, 5e-4, 5e-4])
optimizer = nn.optimizer.MomentGD(init_lr=0.1, model=linear_model,
    mu=0.9)
scheduler = nn.lr_scheduler.MultiStepLR(optimizer=optimizer,
    milestones=[300, 900, 1800], gamma=0.3)
loss_fn = nn.op.MultiCrossEntropyLoss(model=linear_model,
    max_classes=train_labs.max()+1)
runner = nn.runner.RunnerM(linear_model, optimizer,
    nn.metric.accuracy, loss_fn, batch_size=64, scheduler=scheduler)
```

```
[Dev] loss: 1.80130754079332, score: 0.7727
epoch: 2, iteration: 900
[Train] loss: 2.2432868075449512, score: 0.703125
[Dev] loss: 1.7925035625743577, score: 0.7722
```

Figure 7

SGD yields a probably worse result since it was stuck in a local optimum.

```
epoch: 4, iteration: 2100
[Train] loss: 10.308857531315414, score: 0.546875
[Dev] loss: 7.283265513055703, score: 0.6792
epoch: 4, iteration: 2200
[Train] loss: 7.776414847028029, score: 0.625
[Dev] loss: 7.275523380016096, score: 0.6787
epoch: 4, iteration: 2300
[Train] loss: 10.415179997511956, score: 0.546875
[Dev] loss: 7.26817721611492, score: 0.6793
best accuracy performence has been updated: 0.66880 --> 0.67930
```

Figure 8

7 Question7: Visualization of Kernel Weights

Rewrite visualization code in *weight_visualization.py* and get the following result for CNN implemented in Sec.5.

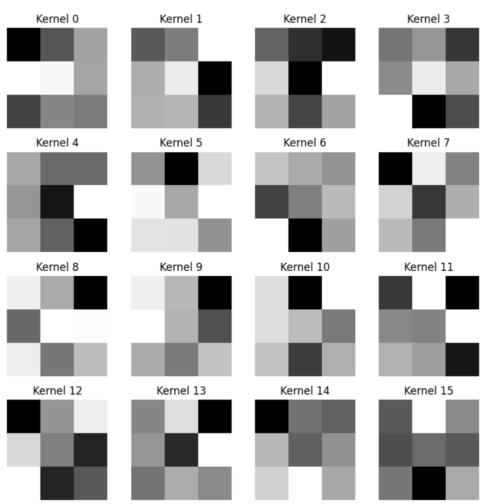


Figure 9

Here we can discover some patterns in writing strokes of numbers.