exam 개인학습

Lec 2.5

AI summary Try on this page

Type something...

01) PyTorch on CIFAR-10

- 이번 과제는 deep learning framework를 사용하는 과제이다.
- framework를 사용하는 이유는 다음과 같다.

1. GPU를 사용해서 훈련을 더 빠르게 할 수 있다. (CUDA)
2. 이미 만들어져 있는 함수들을 쉽게 사용할 수 있다.

01_Part 1. Preparation

```python
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

- `Datasets`, `DataLoader` 등을 이용해서 `CIFAR-10` dataset을 다운로드할 수 있다.

02_Part 2. Barebones PyTorch

- `PyTorch` 를 이용해서 fully-connected ReLU network를 구현할 것이다.
- `hidden layers` 는 2개이고 `biases` 는 없다.
- `(N, C, H, W)` 데이터를 2D로 바꿔주자 → `(N, C x H x W)` (flatten)
- 행렬곱은 `.mm` 을 이용해서 계산한다.

Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

1. `convolutional layer(with bias)` = `(channel_1, KW1 x KH1)` , zero padding 2
2. `ReLU`
3. `convolutional layer(with bias)` = `(channel_2, KW2 x KH2)` , zero padding 1
4. `ReLU`
5. `fully-connected with bias` : scores for C classes

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
x = F.relu(F.conv2d(x, conv_w1, padding = 2, bias = conv_b1))
x = F.relu(F.conv2d(x, conv_w2, padding = 1, bias = conv_b2))
x = flatten(x)
scores = x.mm(fc_w) + fc_b

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                                 END OF YOUR CODE                             #
################################################################################
return scores
```

- `F.Conv2d` 와 `F.relu` 를 이용해서 구현하였다.

1. Barebones PyTorch: Initialization

- `random_weight` 와 `zero_weight` 를 이용해서 초기화를 진행하였다.
- Kaiming normalization을 사용하였다.

```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH, kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

2. Baebones PyTorch: Check Accuracy

```python
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * acc))
```

- gradient를 구할 필요가 없기 때문에 `torch.no_graad()` 안에서 진행한다.

3. Baebones PyTorch: Training Loop

- 이번 훈련 때는 momentum이 없는 SGD를 사용할 것이다.
- Loss는 cross_entropy로 계산하자.

```python
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss
        scores = model_fn(x, params)
        loss = F.cross_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational
        # graph has requires_grad=True and uses backpropagation to compute the
        # gradient of the loss with respect to these Tensors, and stores the
        # gradients in the .grad attribute of each Tensor.
        loss.backward()

        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the backward pass
                w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

- `loader train` 에서 `t, (x, y)` 를 꺼내주고 GPU에 올려둔다.
- score를 구하고 loss를 구한다.
- 그 후 backward를 진행한다.
- 업데이트가 끝나면 gradient를 0을 초기화 시켜준다.

4. Baebones PyTorch: Training a ConvNet

```python
conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight((channel_1, ))
# x = (N, 32, 32, 32)
conv_w2 = random_weight((channel_2, channel_1, 3, 3))
conv_b2 = zero_weight((channel_2, ))
# x = (N, 16, 32, 32)
fc_w = random_weight((channel_2 * 32 * 32, 10))
fc_b = zero_weight(10)
```

- Channel수만 변경 시켜주고 나머지는 함수에 있는 filter size를 이용했다.

03_Part 3. PyTorch Module API

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the [doc](doc) to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

- Module API를 통해서 model를 만들어보자.

1. `nn.Module` 를 상속받자.
2. `__init__()` 에 `nn.Linear + nn.Conv2d` 의 사이즈를 정하고 초기화 시켜주자.
3. `forward()` 에서 model를 완성하자.

```python
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

- 이 모델은 Linear 2개짜리 모델이다.
- Linear → relu → Linear

1. Module API: Three-Layer ConvNet

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classe

```python
def __init__(self, in_channel, channel_1, channel_2, num_classes):
    super().__init__()
    ########################################################################
    # TODO: Set up the layers you need for a three-layer ConvNet with the  #
    # architecture defined above.                                          #
    ########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size = 5, padding = 2)
    nn.init.kaiming_normal(self.conv1.weight)
    self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size = 3, padding = 1)
    nn.init.kaiming_normal(self.conv2.weight)
    self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)
    nn.init.kaiming_normal(self.fc.weight)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ########################################################################
    #                          END OF YOUR CODE                            #
    ########################################################################

def forward(self, x):
    scores = None
    ########################################################################
    # TODO: Implement the forward function for a 3-layer ConvNet. you      #
    # should use the layers you defined in __init__ and specify the        #
    # connectivity of those layers in forward()                            #
    ########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x = F.relu(self.conv1(x))
    x = F.relu(self.conv2(x))
    x = flatten(x)
    scores = self.fc(x)
```

- 코드로 구현할 때 특이한 점은 없었다.

2. Module API: Training Loop

- `optimizer.zero_grad()` → `loss.backward()` → `optimizer.step()` 3가지 과정을 반복한다.

```python
for e in range(epochs):
    for t, (x, y) in enumerate(loader_train):
        model.train()  # put model to training mode
        x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
        y = y.to(device=device, dtype=torch.long)

        scores = model(x)
        loss = F.cross_entropy(scores, y)

        # Zero out all of the gradients for the variables which the optimizer
        # will update.
        optimizer.zero_grad()

        # This is the backwards pass: compute the gradient of the loss with
        # respect to each parameter of the model.
        loss.backward()

        # Actually update the parameters of the model using the gradients
        # computed by the backwards pass.
        optimizer.step()
```

3. Module API: Train a Three-Layer ConvNet

```
model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr = learning_rate)
```

```
Iteration 400, loss = 1.6046
Checking accuracy on validation set
Got 446 / 1000 correct (44.60)

Iteration 500, loss = 1.3930
Checking accuracy on validation set
Got 461 / 1000 correct (46.10)

Iteration 600, loss = 1.4830
Checking accuracy on validation set
Got 473 / 1000 correct (47.30)

Iteration 700, loss = 1.8258
Checking accuracy on validation set
Got 464 / 1000 correct (46.40)
```

- 46%의 정확도를 얻을 수 있다.

04_Part 4. PyTorch Sequential API

```
model = nn.Sequential(nn.Conv2d(3, channel_1, kernel_size = 5, padding = 2),
                      nn.ReLU(),
                      nn.Conv2d(channel_1, channel_2, kernel_size = 3, padding = 1),
                      nn.ReLU(),
                      nn.Flatten(),
                      nn.Linear(channel_2 * 32 * 32, 10))
# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                     momentum=0.9, nesterov=True)
```

- 아까 만들었던 model을 Sequential을 통해서 다시 재구축 하였다.

05_Part 5. CIFAR-10 open-ended challenge

- 이번 목표는 `nn.Sequential` or `nn.Module` 를 통해서 적어도 70%의 정확도를 얻는 것이다.
- 나는 `nn.Module` 를 통해서 클래스를 만들 것이다.
- `conv2d` 는 사이즈가 변하지 않는 선에서 만들어주었다.
- `conv1` : 64 x 7 x 7, padding = 3
- `conv2` : 64 x 7 x 7, padding = 3 → 너무 깊이가 깊어 생략
- `conv3` : 32 x 3 x 3, padding = 1
- `conv4` : 32 x 3 x 3, padding = 1
- `conv5` : 32 x 3 x 3, padding = 1
- `fc1` : 32 x 32 x 32, 32 → 너무 깊이가 깊어 생략
- `fc2` : 32 x 32, 10
- `batchNorm2d`
- `softmax`

`conv1` → `ReLU` → `BatchNorm` → `conv2` → `ReLU` → `BatchNorm` → `conv3` → `ReLU` → `BatchNorm` → `conv4` → `ReLU` → `BatchNorm` → `conv5` → `ReLU` → `Flatten` → `fc1` → `fc2`

```python
  def __init__(self):
    super(AlexNet, self).__init__()
    self.layers = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size= 7, padding = 3),
    nn.ReLU(),
    nn.BatchNorm2d(64),
    nn.Conv2d(64, 64, kernel_size= 7, padding = 3),
    nn.ReLU(),
    nn.BatchNorm2d(64),
    nn.Conv2d(64, 32, kernel_size= 3, padding = 1),
    nn.ReLU(),
    nn.BatchNorm2d(32),
    nn.Conv2d(32, 32, kernel_size= 3, padding = 1),
    nn.ReLU(),
    nn.BatchNorm2d(32),
    nn.Conv2d(32, 32, kernel_size= 3, padding = 1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(32 * 32 * 32, 32 * 32),
    nn.Linear(32 * 32, 10)
    )
  def forward(self , x):
    x = self.layers(x)
    return x
```

- 학습을 시키고 결과를 확인해보자.
- 생각보다 70프로를 넘기는건 힘들다 ㅎㅎ..