Basic Deep Learning Techniques

Although I've studied many topics related to deep learning, I still don't know how to contruct a neural network and never get into the common DL tools such as TensorFlow and PyTorch, so I started a simple tutorial in Kaggle: Intro to Deep Learning. This notes mainly focus on how to use TensorFlow/PyTorch and how to build a model.

1. A Single Neuron

To create a linear neuron, in **TensorFlow** we use

```
model = keras.Sequential(
layers.Dense(unit=1, input_shape=[?])
)
```

(**Don't forget []!**). The input shape means the dimension of the vector which **does not count the target**. The code in **PyTorch** is a little bit more complicated. For the input $X_{n\times i}$, `torch.nn.Linear` applies a linear transformation $Y_{n\times o}=X_{n\times i}W_{i\times o}+b$ is the input.

```
torch.nn.Linear(in_features, #num of input neurons
out_features, #num of output neurons
bias=True)
```

Each element w_{lm} is a neuron, so `in_features` and `out_features` are just the size of weight matrix $W_{i\times o}$. Note that when we use `Linear`, **a tensor is viewed as a 2-dimensional matrix**. Thus for example,

```
X = torch.Tensor([1.,2.,3.]) #1*3
model = torch.nn.Linear(3,4) # 3*4 matrix
output = model(X)
output
>> tensor([-1.4166], grad_fn=<AddmmBackward0>) # 1*1 tensor
```

Suppose now

```
X = torch.Tensor([
    [0,1,0.2,0.3,0.4,0.5],
    [0.1,0.1,0.1,0.3,0.5],
    [0.1,0.4,0.2,0.3,0.2]
])
```

a 3×5 matrix (**Don't forget [] !**). The `in_features` should be 5. Let's say we want the output be of the size 3×10. Then

```
model = torch.nn.Linear(5,10)
output = model(X)
output
>> tensor([[ 0.3844,  0.4926, -0.0800,  0.1387, -0.5622, -0.1287,  0.1424,
0.2119,
          0.2945,  0.1583],
        [ 0.3084,  0.3935, -0.0992,  0.0705, -0.4386, -0.1054,  0.2088,
0.2399,
          0.3493,  0.0843],
        [ 0.3878,  0.4486,  0.0828, -0.0193, -0.4876, -0.1795,  0.2149,
0.1331,
          0.2172,  0.3481]], grad_fn=<AddmmBackward0>)
```

2. Deep Neural Networks

Now we build a NN by ourself. In DNN, we usually apply activation functions to our neurons. In **TensorFlow**, when we want to use the activation function, for example Relu, then we need to specify it by `activation='relu'`. Let's say we want to build a neural network with 3 hidden layers and input shape [8], where each layer has 512 neurons and uses Relu. Also, the output layer has one unit and is linear. Then our network should be

```
model = keras.Sequential([
    layers.Dense(units=512,activation='relu',input_shape=[8]),
    layers.Dense(units=512,activation='relu'),
    layers.Dense(units=512,activation='relu'),
    layers.Dense(unit=1)
])
```

You can also specify the `Activation` layer in each layer by

```
layers.Dense(units=512,input_shape=[8]),
layers.Activation('relu'),
```

In **PyTorch** we need to be more careful. Let

```
X = torch.Tensor([
    [0,1,0.2,0.3,0.4,0.5],
    [0.1,0.1,0.1,0.3,0.5],
    [0.1,0.4,0.2,0.3,0.2]
]).
```

The network should be

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = torch.nn.Sequential(
    torch.nn.Linear(5,512),
    torch.nn.ReLU(),
    torch.nn.Linear(512,512),
    torch.nn.ReLU(),
    torch.nn.Linear(512,512),
    torch.nn.ReLU(),
    torch.nn.Linear(512,1),
).to(device)
```

(**No []!**). The first layer is `torch.nn.Linear(5,512)` since X is a 3×5 tensor. The output is

```
tensor([[0.0196],
        [0.0184],
        [0.0352]], grad_fn=<AddmmBackward0>)
```

which is output of the linear transformation $Y_{3\times 1}=X_{3\times 5}W^1_{5\times 512}W^2_{512\times 512}W^3_{512\times 1}+b$.

3. Stochastic Gradient Descent

In deep learning, the **optimizer** is an algorithm that adjusts the weights to minimize the loss. The **batch size** is the number of training data in a single step of training. The **learning rate** and **batch size** are the most important factors that effect how the SGD training proceeds. In **TensorFlow**, it's very simple to train the model:

```
model = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=input_shape),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(1),
]) #input_shape=[50]
model.compile(optimizer='adam',loss='mae')
```

and

```
model.fit(
    X, y,
    batch_size=128,
    epochs=200,
)
```

We can let `history=model.fit(...)` to plot the chart with `history`. In **PyTorch**, we have to write the code explicitly.

```
model = torch.nn.Sequential(
    torch.nn.Linear(50,128),
    torch.nn.ReLU(),
    torch.nn.Linear(128,128),
    torch.nn.ReLU(),
    torch.nn.Linear(128,64),
    torch.nn.ReLU(),
    torch.nn.Linear(64,1),
)

batch_size=128,
epochs=200,
learning_rate=0.05

loss_list = [] #record loss
optimizer = torch.optim.SGD(model.parameters(),lr=learning_rate)
criterion = torch.nn.MSELoss()
for epoch in range(epochs):
    y_pred = model(X)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print('epoch {}, loss {}'.format(epoch, loss.item()))
```

4. Overfitting

To prevent overfitting, we can can simply stop the training, called **ealy stopping**. In TensorFlow, the function is very simple:

```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(
    min_delta=0.001,  #minimium amount of change to count as an
improvement
    patience=20,  #how many epochs to wait before stopping
    restore_best_weights=True,
)
```

It's used in `model.fit`:

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=512,
    epochs=50,
    callbacks=[early_stopping]
)
```

In **PyTorch**, we have to define the function ourself. It's a good convention to use `class`.

```
class EarlyStopper:
    def __init__(self, patience=1, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_validation_loss = np.inf

    def early_stop(self, validation_loss):
        if validation_loss < self.min_validation_loss:
            self.min_validation_loss = validation_loss
            self.counter = 0
        elif validation_loss > (self.min_validation_loss +
self.min_delta):
            self.counter += 1
            if self.counter >= self.patience:
                return True
        return False
```

This is the way to use:

```
early_stopper = EarlyStopper(patience=3, min_delta=10)
for epoch in np.arange(n_epochs):
    train_loss = train_one_epoch(model, train_loader)
    validation_loss = validate_one_epoch(model, validation_loader)
    if early_stopper.early_stop(validation_loss):
        break
```

5. Dropout and Batch Normalization

Dropout originally comes from this paper.

`dropout=0.2` means that for each neuron in the next layer, its input is dropped out by $20%$ randomly, i.e. the output of $20%$ neurons in the current layer are not put in this neuron. To achieve this in **TensorFlow**, we only need to add a line to `model`:

```
layers.Dropout(0.2)
```

after the activation function. In **PyTorch**, we instead use

```
torch.nn.Dropout(0.2)
```

Batch normalization can alleviate gradient vanishing and accelerate the convergence. Let ${x_1,...,x_m}$ be a batch of training data. Batch normalization is simply normalize each x_i:

- $\mu_B=\frac{1}{m}\sum_{i=1}^mx_i$
- $\sigma_B=\frac{1}{m}\sum_{i=1}^m(x_i-\mu_B)^2$
- $\hat{x_i}=\frac{x_i-\mu_B}{\sigma_B+\varepsilon}$

The ε term is to prevent the situation where σ_B is close to 0. We introduce two trainable parameters γ and β so that

- $\hat{y_i}=\gamma\hat{x_i}+\beta$.

We can of course not train γ and β. In **TensorFlow**, we simply add

```
layers.BatchNormalization()
```

after each layer. In **PyTorch**, we need to specify the neurons, for example:

```
...
torch.nn.Linear(5,7),
torch.nn.BatchNorm1d(7),
...
```