Chem 277B Spring 2024 Tutorial 7

Outline

- Convolutional Neural Network (CNN):
    - Hyperparamters in CNN: channels, padding, stride, dilation
    - Pooling
    - CNN in PyTorch
- Residual Network
- Batch Normalization

HW6 - Helper function

You can use the following decorator to report time:

In [ ]:
```
import time

def timeit(f):

    def timed(*args, **kw):

        ts = time.time()
        result = f(*args, **kw)
        te = time.time()

        print(f'func:{f.__name__} took: {te-ts:.4f} sec')
        return result

    return timed

@timeit
def sleep(sec):
    return time.sleep(sec)

sleep(0.1)
```

```
func:sleep took: 0.1050 sec
```

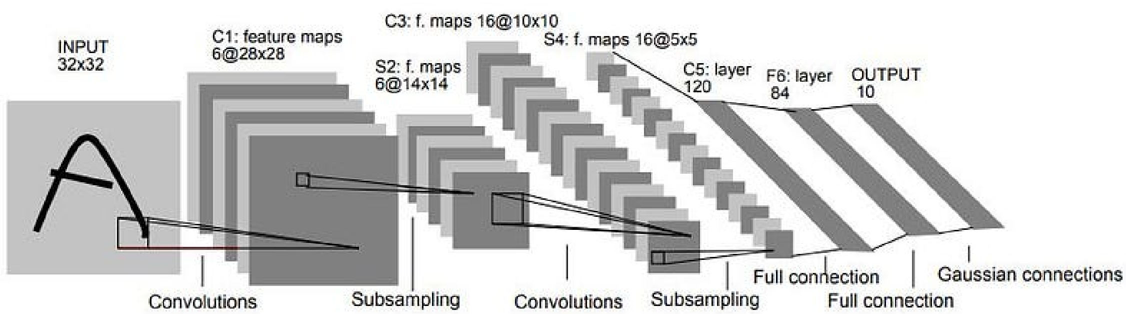
In [ ]:
```
class Trainer:

    def __init__(self, model, opt_method, learning_rate, batch_size, epoch,
        self.model = model

        if opt_method == "adam":
            self.optimizer = torch.optim.Adam(model.parameters(), learning_r
```

```
        else:
            raise NotImplementedError("This optimization is not supported")

        self.epoch = epoch
        self.batch_size = batch_size

    @timeit
    def train(self, train_data, val_data, early_stop=True, verbose=True, dra
        train_loader = DataLoader(train_data, batch_size=self.batch_size, sh

        train_loss_list, train_acc_list = [], []
        val_loss_list, val_acc_list = [], []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        loss_func = nn.CrossEntropyLoss()
        for n in tqdm(range(self.epoch), leave=False):
            # enable train mode
            self.model.train()
            epoch_loss, epoch_acc = 0.0, 0.0
            for X_batch, y_batch in train_loader:
                # batch_importance is the ratio of batch_size
                batch_importance = y_batch.shape[0]/len(train_data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)

                self.optimizer.zero_grad()
                batch_loss.backward()
                self.optimizer.step()

                epoch_loss += batch_loss.detach().cpu().item() * batch_impor
                batch_acc = torch.sum(torch.argmax(y_pred, axis=1) == y_batc
                epoch_acc += batch_acc.detach().cpu().item() * batch_importa


#             train_loss_list.append(epoch_loss)
#             train_acc_list.append(epoch_acc)
            # previous way to report might get low acc due to dropout
            train_loss, train_acc = self.evaluate(train_data)

            val_loss, val_acc = self.evaluate(val_data)
            val_loss_list.append(val_loss)
            val_acc_list.append(val_acc)

            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()

        if draw_curve:
            x_axis = np.arange(self.epoch)
            fig, axes = plt.subplots(1, 2, figsize=(10, 4))
            axes[0].plot(x_axis, train_loss_list, label="Train")
            axes[0].plot(x_axis, val_loss_list, label="Validation")
            axes[0].set_title("Loss")
            axes[0].legend()
            axes[1].plot(x_axis, train_acc_list, label='Train')
```

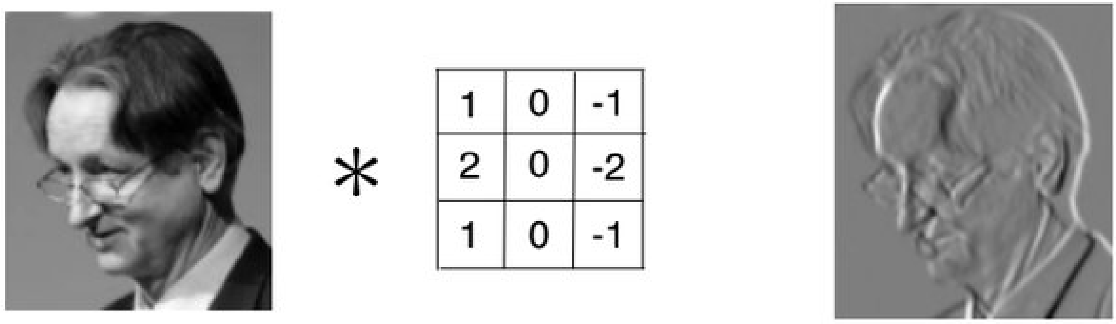
```
            axes[1].plot(x_axis, val_acc_list, label='Validation')
            axes[1].set_title("Accuracy")
            axes[1].legend()

        if early_stop:
            self.model.load_state_dict(weights)

        return {
            "train_loss_list": train_loss_list,
            "train_acc_list": train_acc_list,
            "val_loss_list": val_loss_list,
            "val_acc_list": val_acc_list,
        }

    def evaluate(self, data, print_acc=False):
        # enable evaluation mode
        self.model.eval()
        loader = DataLoader(data, batch_size=self.batch_size, shuffle=True)
        loss_func = nn.CrossEntropyLoss()
        acc, loss = 0.0, 0.0
        for X_batch, y_batch in loader:
            with torch.no_grad():
                batch_importance = y_batch.shape[0]/len(data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)
                batch_acc = torch.sum(torch.argmax(y_pred, axis=1) == y_batc
                acc += batch_acc.detach().cpu().item() * batch_importance
                loss += batch_loss.detach().cpu().item() * batch_importance
        if print_acc:
            print(f"Accuracy: {acc:.3f}")
        return loss, acc
```

Convolutional Neural Netwok (CNN)

CNN general architechture



Convolution Filters help extract features



Calculating convolution output shape

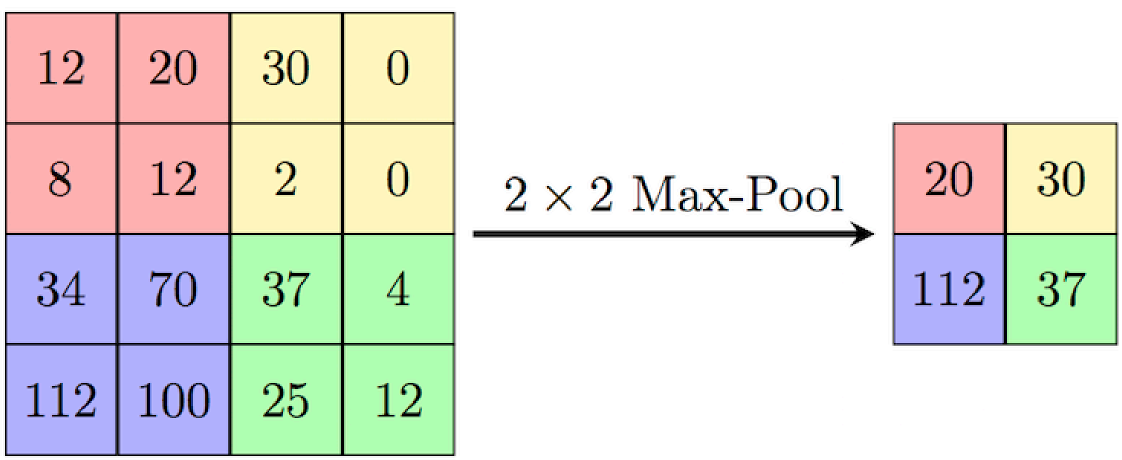
Here is a visualization for padding, stride and dilation

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{ padding } - \text{dilation} \times (\text{ kernel size } - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

In [ ]:
```
import pickle
import torch
import torch.nn as nn
```

In [ ]:
```
# init a Conv2d layer
conv = nn.Conv2d(1, 1, kernel_size=(2,2))
conv
```

Out[ ]:
```
Conv2d(1, 1, kernel_size=(2, 2), stride=(1, 1))
```



In [ ]:
```
# init a MaxPool layer
max_pool = nn.MaxPool2d(2)
max_pool
```

Out[ ]:
```
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

In [ ]:
```
# init a Average Pool layer
avg_pool = nn.AvgPool2d(2)
```

```
avg_pool
```

Out[ ]: AvgPool2d(kernel_size=2, stride=2, padding=0)

In [ ]:
```
def out_dim(in_dim, kernel_size, padding, stride, dilation):
    return (in_dim + 2 * padding - dilation * (kernel_size - 1) - 1) // stri


# data shape: (N, C, W, H)
data = torch.rand(1, 1, 2, 2)
conv(data)
```

Out[ ]: tensor([[[[-0.4292]]]], grad_fn=<ConvolutionBackward0>)

LeNet architecture

LeCun, Y.; Bottou, L.; Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition.Proceedings of the IEEE. 86(11): 2278 - 2324.

Layer No.	Layer type	#channels/#features	Kernel size	Stride	Activation
1	2D Convolution	6	5	1	tanh
2	Average pooling	6	2	2	\
3	2D Convolution	16	5	1	tanh
4	Average pooling	16	2	2	\
5	2D Convolution	120	5	1	tanh
6	Flatten				\
7	Fully connected	84			tanh
8	Fully connected	10			softmax

In [ ]:
```
def load_dataset(path):
    with open(path, 'rb') as f:
        train_data, test_data = pickle.load(f)

    X_train = torch.tensor(train_data[0], dtype=torch.float).unsqueeze(1)
    y_train = torch.tensor(train_data[1], dtype=torch.long)
    X_test = torch.tensor(test_data[0], dtype=torch.float).unsqueeze(1)
    y_test = torch.tensor(test_data[1], dtype=torch.long)
    return X_train, y_train, X_test, y_test

X_train, y_train, X_test, y_test = load_dataset("mnist.pkl")
```

In [ ]:
```
class LeNet(nn.Module):
    def __init__(self, in_channels=1):
        super().__init__()
        self.conv = nn.ModuleList([
            nn.Conv2d(in_channels, 6, kernel_size=5, stride=1),
            nn.Conv2d(6, 16, kernel_size=5, stride=1),
```

```
            nn.Conv2d(16, 120, kernel_size=5, stride=1)
        ])
        self.pool = nn.AvgPool2d(2)
        self.activation = nn.Tanh()
        self.fc = nn.ModuleList([
            nn.Linear(120, 84),
            nn.Linear(84, 10)
        ])

    def forward(self, x):
        for i in range(2):
            x = self.pool(self.activation(self.conv[i](x)))
        x = nn.Flatten()(self.activation(self.conv[2](x)))
        x = self.activation(self.fc[0](x))
        x = nn.Softmax(dim=-1)(self.fc[1](x))
        return x

net = LeNet()
net
```

Out[ ]:

```
LeNet(
  (conv): ModuleList(
    (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (2): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  )
  (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (activation): Tanh()
  (fc): ModuleList(
    (0): Linear(in_features=120, out_features=84, bias=True)
    (1): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

In [ ]:

```
# Use torchsummary to print the architecture
# ! pip install torchsummary
from torchsummary import summary

s = summary(net, (1, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             156
              Tanh-2            [-1, 6, 28, 28]               0
         AvgPool2d-3            [-1, 6, 14, 14]               0
            Conv2d-4           [-1, 16, 10, 10]           2,416
              Tanh-5           [-1, 16, 10, 10]               0
         AvgPool2d-6             [-1, 16, 5, 5]               0
            Conv2d-7            [-1, 120, 1, 1]          48,120
              Tanh-8            [-1, 120, 1, 1]               0
            Linear-9                   [-1, 84]          10,164
             Tanh-10                   [-1, 84]               0
           Linear-11                   [-1, 10]             850
================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.11
Params size (MB): 0.24
Estimated Total Size (MB): 0.35
----------------------------------------------------------------
```

In [ ]:

```
net(X_train[:10])
```

Out[ ]:
```
tensor([[0.0852, 0.0939, 0.0924, 0.0957, 0.0943, 0.1224, 0.0921, 0.0951, 0.
1162,
         0.1127],
        [0.0886, 0.0982, 0.0959, 0.0981, 0.0989, 0.1172, 0.0908, 0.0948, 0.
1115,
         0.1060],
        [0.0857, 0.0923, 0.0987, 0.0914, 0.0947, 0.1241, 0.1011, 0.0948, 0.
1085,
         0.1088],
        [0.0866, 0.0932, 0.0929, 0.1049, 0.0901, 0.1173, 0.1016, 0.0951, 0.
1060,
         0.1124],
        [0.0814, 0.0931, 0.0989, 0.1023, 0.0948, 0.1180, 0.0936, 0.0964, 0.
1132,
         0.1083],
        [0.0857, 0.0949, 0.0938, 0.1034, 0.0925, 0.1185, 0.0992, 0.0949, 0.
1068,
         0.1102],
        [0.0846, 0.0943, 0.0928, 0.0976, 0.0920, 0.1193, 0.0957, 0.1011, 0.
1092,
         0.1133],
        [0.0861, 0.0948, 0.0940, 0.0980, 0.0971, 0.1200, 0.0924, 0.0957, 0.
1109,
         0.1110],
        [0.0840, 0.0929, 0.0944, 0.0982, 0.0943, 0.1147, 0.1005, 0.1016, 0.
1086,
         0.1107],
        [0.0867, 0.0931, 0.0925, 0.1038, 0.0950, 0.1199, 0.0986, 0.0930, 0.
1104,
         0.1069]], grad_fn=<SoftmaxBackward0>)
```

Residual Network (ResNet)

An example of residual block:

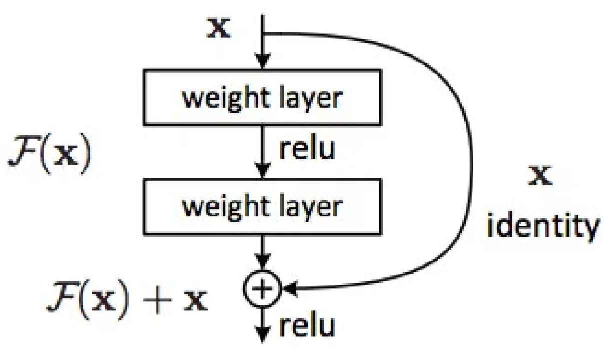


Figure 2. Residual learning: a building block.

In [ ]:
```
class ResBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
```

```python
        self.fc = nn.ModuleList([nn.Linear(dim, dim), nn.Linear(dim, dim)])
        self.activation = nn.ReLU()

    def forward(self, x):
        out = self.activation(self.fc[0](x))
        out = self.fc[1](out)
        out += x
        out = self.activation(out)
        return out
```

In [ ]:

```python
class LeNetRes(nn.Module):
    def __init__(self, in_channels=1):
        super().__init__()
        self.conv = nn.ModuleList([
            nn.Conv2d(in_channels, 6, kernel_size=5, stride=1),
            nn.Conv2d(6, 16, kernel_size=5, stride=1),
            nn.Conv2d(16, 120, kernel_size=5, stride=1)
        ])
        self.pool = nn.AvgPool2d(2)
        self.activation = nn.Tanh()
        self.fc = nn.ModuleList([
            nn.Linear(120, 120),
            nn.Linear(120, 84),
            nn.Linear(84, 10)
        ])

    def forward(self, x):
        for i in range(2):
            x = self.pool(self.activation(self.conv[i](x)))
        x = nn.Flatten()(self.activation(self.conv[2](x)))
        x = self.activation(x + self.fc[0](x))
        x = self.activation(self.fc[0](x))
        x = nn.Softmax(dim=-1)(self.fc[1](x))
        return x

net = LeNetRes()
net
```

Out[ ]:

```
LeNetRes(
  (conv): ModuleList(
    (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (2): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  )
  (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (activation): Tanh()
  (fc): ModuleList(
    (0): Linear(in_features=120, out_features=120, bias=True)
    (1): Linear(in_features=120, out_features=84, bias=True)
    (2): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

In [ ]:

```python
s = summary(net, (1, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             156
              Tanh-2            [-1, 6, 28, 28]               0
         AvgPool2d-3            [-1, 6, 14, 14]               0
            Conv2d-4           [-1, 16, 10, 10]           2,416
              Tanh-5           [-1, 16, 10, 10]               0
         AvgPool2d-6             [-1, 16, 5, 5]               0
            Conv2d-7            [-1, 120, 1, 1]          48,120
              Tanh-8            [-1, 120, 1, 1]               0
            Linear-9                  [-1, 120]          14,520
             Tanh-10                  [-1, 120]               0
           Linear-11                  [-1, 120]          14,520
             Tanh-12                  [-1, 120]               0
           Linear-13                   [-1, 84]          10,164
================================================================
Total params: 89,896
Trainable params: 89,896
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.11
Params size (MB): 0.34
Estimated Total Size (MB): 0.46
----------------------------------------------------------------
```

Batch Normalization (BN)

For a 4-D input data X with shape (N, C, W, H). For each channel, the data is normalized by:

$$\hat{X}_{ijkl} = \frac{X_{ijkl} - \text{mean}(X_j)}{\sqrt{\text{var}(X_j) + \epsilon}} * \gamma_j + \beta_j$$

where

$$\text{mean}(X_j) = \frac{1}{NWH} \sum_i^N \sum_k^W \sum_l^H X_{ikl}$$

$$\text{var}(X_j) = \frac{1}{NWH} \sum_i^N \sum_k^W \sum_l^H (X_{ikl} - \text{mean}(X_j))^2$$

ϵ is a small number (say, 10^{-5}) to avoid numerical instability. $\boldsymbol{\gamma}, \boldsymbol{\beta}$ are learnable parameters

In [ ]:
```
batch_norm = nn.BatchNorm2d(120)
batch_norm
```

Out[ ]: BatchNorm2d(120, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

In [ ]:
```
class LeNetResNorm(nn.Module):
    def __init__(self, in_channels=1):
        super().__init__()
        self.conv = nn.ModuleList([
            nn.Conv2d(in_channels, 6, kernel_size=5, stride=1),
            nn.Conv2d(6, 16, kernel_size=5, stride=1),
            nn.Conv2d(16, 120, kernel_size=5, stride=1)
        ])
        self.bn = nn.ModuleList([
            nn.BatchNorm2d(6),
            nn.BatchNorm2d(16),
        ])
        self.pool = nn.AvgPool2d(2)
        self.activation = nn.Tanh()
        self.fc = nn.ModuleList([
            nn.Linear(120, 120),
            nn.Linear(120, 84),
            nn.Linear(84, 10)
        ])

    def forward(self, x):
        for i in range(2):
            x = self.bn[i](self.pool(self.activation(self.conv[i](x))))
        x = nn.Flatten()(self.activation(self.conv[2](x)))
        x = self.activation(x + self.fc[0](x))
        x = self.activation(self.fc[0](x))
        x = nn.Softmax(dim=-1)(self.fc[1](x))
        return x

net = LeNetResNorm()
net
```

Out[ ]:
```
LeNetResNorm(
  (conv): ModuleList(
    (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (2): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  )
  (bn): ModuleList(
    (0): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (activation): Tanh()
  (fc): ModuleList(
    (0): Linear(in_features=120, out_features=120, bias=True)
    (1): Linear(in_features=120, out_features=84, bias=True)
    (2): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

In [ ]:

```

```