MODELS

Logan Lieou

November 11, 2021

1 Neural Networks

1.1 Convolutional Neural Network

A convolutional neural network uses convolutions, that's why it's called a convolutional neural network. Instead of normal matrix operations we're instead using a convolution, in a normal feed forward network, there's an input layer, hidden layers and an output layer. Instead of the hidden layers being typical fully connected layers they are instead convolutional layers. A convolution can be defined as an operation on two functions that produces a third function that expresses how the shape of one is modified by the other. [1]

1.1.1 Convolutions

A convolution of the functions f and g can be written as $f * g$ the symbol $*$ denotes a convolution. A convolution is a particular kind of integral transform:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

There's explaination to this function to give a more intuitive understanding of it but I'm really lazy right now.

1.1.2 Network Architecture

Convolutional networks are similar to a normal feed forward network with the main difference being that we use convolutions instead of fully connected layers, a typical network may look something like:

- Input Layer of size (B, W, H, C)
- Convolution
- Max Pool
- Convolution
- Max Pool

Repeat these layers a few times, then ending with a few Dense layers to map to outputs. In code this may look something like this:

```
import torch.nn as nn

class CNN(nn.Module):
    def __init__(self, in_channels, height, width, n_classes):
        super(CNN, self).__init__()
        # activation function
        self.activation = nn.ReLU()

        # first convolutional layer
        self.conv1 = nn.Conv2d(in_channels, 16, (5, 5), padding=(2, 2))
        self.pool1 = nn.MaxPool2d((2, 2), stride=2)

        # second convolutional layer
        self.conv2 = nn.Conv2d(16, 8, (2, 2), padding=(2, 2))
        self.pool2 = nn.MaxPool2d((2, 2), stride=2)

        # fully connected layer
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(height*width*8, 120)
        self.fc2 = nn.Linear(120, 84)

        # output to number of classes predicts 1 of n_classes
        self.output = nn.Linear(84, n_classes)

    def forward(self, x):
        # first convolutional layer
        out = self.conv1(x)
        out = self.activation(out)
        out = self.pool1(out)

        # second convolutional layer
        out = self.conv2(out)
        out = self.activation(out)
        out = self.pool2(out)

        # fully connected layers
        out = self.flatten(out)
        out = self.fc1(out)
        out = self.fc2(out)

        return self.output(out)
```

1.2 UNet

Made up of this encoder – decoder architecture combined with residual connections. UNet was specifically made for the task of medical image segmentation. [2] Which is perfect for us because that's exactly what we're trying to do. UNet starts with an convolutional encoding layer that looks very similar to a standard feed forward convolutional network but instead of flattening out and passing to Dense layers, the model is then decoded by a residual convolutional decoder, which outputs a mask that can be threshold to produce binary mask for segmentation. So it's all the same but instead of flattening the output of a convolutional layer there's instead a decoding layer that outputs a mask instead of Dense output of $1 \times n$ probability matrix for classification you get a $n \times m$ mask to overlay on image. In addition you also have to implement the residual connections:

1.2.1 Residual Connections

Residual connections are added to combat vanishing gradient by adding outputs of previous layers to later layers

```
import torch.nn as nn

class Block(nn.Module):
    def __init__(self):
        super(Block, self).__init__()
        self.inp = nn.Linear(1, 18)
        self.act = nn.Sigmoid()
        self.fc1 = nn.Linear(18, 1)

    def forward(self, x):
        out = self.inp(x)
        out = self.act(out)
        out = self.fc1(out)

        return out

class ResNet(nn.Module):
    def __init__(self, height, width):
        super(ResNet, self).__init__()
        self.block1 = Block()
        self.block2 = Block()
        self.block3 = Block()

    def forward(self, x):
        rout = self.block1(x)
        out = self.block2(rout)
        # residual connection
        out = self.block(rout) + rout

        return out
```

1.2.2 Encoder Decoder

Implementation of a very simple convolutional encoder decoder architecture:

```
import torch.nn as nn

class EncDec(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, (2, 2), padding=(1, 1)),
            nn.Conv2d(16, 32, (2, 2), padding=(1, 1)),
            nn.Conv2d(32, 64, (2, 2), padding=(1, 1))
        )
        self.decoder = nn.Sequential(
            nn.Conv2d(64, 32, (2, 2), padding=(1, 1)),
            nn.Conv2d(32, 16, (2, 2), padding=(1, 1)),
            nn.Conv2d(16, 8, (2, 2), padding=(1, 1))
        )

    def forward(self, x):
        out = self.encoder(x)
        out = self.decoder(out)

        return out
```

2 Statistical Models

2.1 Support Vector Classifier

Draw hyper planes to seperate data try to maximize distance between support vectors and plane.

2.2 XGBoost

Gradient boost decision tree

2.3 Data Extraction

2.3.1 GraphicalLassoCV

Calculate sparse inverse covariance for time series data in order to produce connection information.

2.3.2 KMeans

Cluster brain regions based on criteria

2.4 Noisy OR

"If there are n independent causes for $\{X_1, ..., X_n\}$ for a random variable Y and assuming simplicity that Y is binary, then the target distrobution $P(Y = 1|X_1 = x_1, ..., X_n)$ is given by the following eq:" [3]

$$P(Y = 1|X_1 = x_1, .., X_n = x_n) = 1 - \Pi_i P(Y = 0|X_i = x_i)$$

References

[1] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, Ieee, 2017.

[2] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[3] S. Yang and S. Natarajan, "Knowledge intensive learning: Combining qualitative constraints with causal independence for parameter learning in probabilistic models," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 580–595, Springer, 2013.