ML4Science:
Week 2 Meeting

Calafà, M., Mescolini, G., Motta, P.

École Polytechnique Fédérale de Lausanne (EPFL)

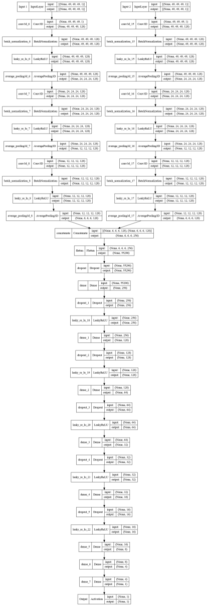
Machine Learning Project 2

30 Nov 2021

Tutor: Dr. Michele Bianco

Outlines

Structure of the CNN



As first attempt:

1. Complete structure as in picture (*left*, *right*, *center branches*)
2. Same parameters to avoid inventing from scratch

Main parameters

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # MAIN PARAMETERS
        self.kernel_size = 5
        self.padding = (self.kernel_size - 1)/2
        self.stride_conv = 1

        self.kernel_pooling = 2
        self.stride_pool = self.kernel_pooling
```

Main parameters

Choices of the parameters motivated by these reasons:

- $Kernel = 5$ as common, basic first choice (to be corrected in the future maybe)
- $Stride = 1$ to convolute at every step
- $Padding = \frac{Kernel-1}{2}$ to keep same sizes before and after the convolution
- For pooling, $Stride = Pooling$ to make steps in correspondence to the pooled areas

Left branch

```
        # LEFT BRANCH
        self.conv3d_6 = nn.Conv3d(in_channels=1, out_channels=128, kernel_size=self.kernel_size, stride=self.stride_conv, padding=self.padding)
        self.batch_normalization_6 = nn.BatchNorm3d(num_features=128)
        self.leaky_re_lu_6 = nn.LeakyReLU()
        self.average_pooling_3d_6 = nn.AvgPool3d(kernel_size = self.kernel_pooling)

        self.conv3d_7 = nn.Conv3d(in_channels=128, out_channels=128, kernel_size=self.kernel_size, stride=self.stride_conv, padding=self.padding)
        self.batch_normalization_7 = nn.BatchNorm3d(num_features=128)
        self.leaky_re_lu_7 = nn.LeakyReLU()
        self.average_pooling_3d_7 = nn.AvgPool3d(kernel_size = self.kernel_pooling)

        self.conv3d_8 = nn.Conv3d(in_channels=128, out_channels=128, kernel_size=self.kernel_size, stride=self.stride_conv, padding=self.padding)
        self.batch_normalization_8 = nn.BatchNorm3d(num_features=128)
        self.leaky_re_lu_8 = nn.LeakyReLU()
        self.average_pooling_3d_8 = nn.AvgPool3d(kernel_size = self.kernel_pooling)
```

Right branch

```
# RIGHT BRANCH
self.conv3d_15 = nn.Conv3d(in_channels=1, out_channels=128, kernel_size=
    self.kernel_size, stride=self.stride_conv, padding=self.padding)
self.batch_normalization_15 = nn.BatchNorm3d(num_features=128)
self.leaky_re_lu_15 = nn.LeakyReLU()
self.average_pooling_3d_15 = nn.AvgPool3d(kernel_size = self.
    kernel_pooling)

self.conv3d_16 = nn.Conv3d(in_channels=128, out_channels=128,
    kernel_size=self.kernel_size, stride=self.stride_conv, padding=self
    .padding)
self.batch_normalization_16 = nn.BatchNorm3d(num_features=128)
self.leaky_re_lu_16 = nn.LeakyReLU()
self.average_pooling_3d_16 = nn.AvgPool3d(kernel_size = self.
    kernel_pooling)

self.conv3d_17 = nn.Conv3d(in_channels=128, out_channels=128,
    kernel_size=self.kernel_size, stride=self.stride_conv, padding=self
    .padding)
self.batch_normalization_17 = nn.BatchNorm3d(num_features=128)
self.leaky_re_lu_17 = nn.LeakyReLU()
self.average_pooling_3d_17 = nn.AvgPool3d(kernel_size = self.
    kernel_pooling)
```

Central branch

```
        # CENTRAL BRANCH

        self.dropout = nn.Dropout(p=0.8)

        self.dense = nn.Linear(in_features=55296, out_features=256)
        self.dropout_1 = nn.Dropout(p=0.5)
        self.leaky_re_lu_18 = nn.LeakyReLU()

        self.dense_1 = nn.Linear(in_features=256, out_features=128)
        self.dropout_2 = nn.Dropout(p=0.5)
        self.leaky_re_lu_19 = nn.LeakyReLU()

        self.dense_2 = nn.Linear(in_features=128, out_features=64)
        self.dropout_3 = nn.Dropout(p=0.5)
        self.leaky_re_lu_20 = nn.LeakyReLU()

        self.dense_3 = nn.Linear(in_features=64, out_features=32)
        self.dropout_4 = nn.Dropout(p=0.5)
        self.leaky_re_lu_21 = nn.LeakyReLU()

        self.dense_4 = nn.Linear(in_features=32, out_features=16)
        self.dropout_5 = nn.Dropout(p=0.5)
        self.leaky_re_lu_22 = nn.LeakyReLU()

        self.dense_5 = nn.Linear(in_features=16, out_features=8)
        self.dense_6 = nn.Linear(in_features=8, out_features=4)
        self.dense_7 = nn.Linear(in_features=4, out_features=1)
```

get_neighborhood

```python
def get_neighborhood(T,x0,y0,z0,r):

    edge = T.shape[3]

    idx = list(range(x0-r, min(edge,x0+r+1))) + list(range(max(edge,x0+r+1) % edge))
    idy = list(range(y0-r, min(edge,y0+r+1))) + list(range(max(edge,y0+r+1) % edge))
    idz = list(range(z0-r, min(edge,z0+r+1))) + list(range(max(edge,z0+r+1) % edge))

    Tm = T[0,0,:,:,:]
    neigh = Tm[idx,:,:][:,idy,:][:,:,idz]
    neigh = np.expand_dims(neigh, axis=0)
    neigh = np.expand_dims(neigh, axis=0)

    return neigh
```

get_neighborhood

To get convinced from the code, take the simpler 1D case.
$\{x_0 - r, \ldots, x_0 + r\}$ inside $\{0, \ldots, edge - 1\}$. Three possible cases:

- Fully inside the larger interval
  $\Rightarrow idx = \{x_0 - r, \ldots, x_0 + r\} \cup \{\emptyset\}$
- $x_0 - r$ exceeds from left
  $\Rightarrow$ same as before (see negative indices in `numpy`)
- $x_0 + r$ exceeds from right
  $\Rightarrow idx = \{x_0 - r, \ldots, edge - 1\} \cup \{0, \ldots, (x_0 + r) \mod edge\}$

Choice of the Loss

In our implementation, we opted for a **MSE Loss**.
We did not choose the Cross Entropy Loss as it is suitable only for classification problems. In addition, we initially got dimension incompatibilities when we applied it to our net.

Questions

1. Parameters choice discussion:
   - *"Main parameters"* (i.e. kernel size, padding ...)
   - Set vs default parameters in *PyTorch* modules

     ```
     self.batch_normalization_6 = nn.BatchNorm3d(..?..)
     ```

   - Flattening dimension

     ```
     out = torch.cat((out1, out2), dim = 0)
     ```

   - Choice of the Loss (*MSE Loss*)
2. Should we divide our dataset in training and test set or do we have other test data?
3. About February conference
4. Issues (see next slide)

Issues

- Problem when passing the padding number as parameter
  $\Rightarrow$ solved by passing directly the number
- Need of more efficient code; due to the extremely high computational time, we actually do not have an output