Xi Liu

robot hand state prediction
State prediction given RGBD (RGB + Depth) images. The input is RGBD images of top view robot hand, after the use of several supervised learning algorithms, the output is vertex positions of each finger in meters. Each sample is made of three images from three different views. A custom dataset class is defined for lazy loading to deal with out of memory issue to load only when the training iteration started to use the portion of the data.

method
data preprocessing
A custom data processing class *Data_Preprocessing* is implemented in *CNN_model.py*

```
dp = CNN_model.Data_Preprocessing()
```

load *data_train* and *data_test* using custom data loader *load_images*()

```
data_train = CNN_model.load_images(path = './lazydata/', isTrain = True)
data_test = CNN_model.load_images(path = './lazydata/', isTrain = False)
```

convert *data_train* and *data_test* from tensor to array using *tensorToArray*()

```
img0_array_test, img1_array_test, img2_array_test, depth_array_test, field_id_array = dp.
    tensorToArray(data = data_test, isTrain = False)
img0_array_train, img1_array_train, img2_array_train, depth_array_train, y_array = dp.
    tensorToArray(data = data_train, isTrain = True)
```

normalize depth arrays and image arrays using *depth_normalization*(), *img_normalization*(), *combine_image_depth*()

```
normalized_depth_train = dp.depth_normalization(depth = depth_array_train)
normalized_img0_train = dp.img_normalization(img = img0_array_train)
new_img_train = dp.combine_image_depth(img = normalized_img0_train, depth =
    normalized_depth_train, whichImg = 0)
ready_img_train = dp.reshape_data(new_img_train)
```

combine *ready_img_train* and *y_array* into one array, and save that data persistently as a *.joblib* file using *joblib.dump*()

```
train_img0 = [ready_img_train, y_array]
dump(train_img0, 'lx_preprocessed_data0.joblib')
```

train model
start to train the model by calling *cnn_model.main*(), many residual neural network architectures are used, including ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152.

```
cnn_model = cnn_model.main(loadname = 'lx_preprocessed_data0.joblib', pre_trained_model =
    None)
model_scripted = torch.jit.script(cnn_model)
model_scripted.save('res50_pretrained_model.pt') # save model
sub = submission.Submission()
df = sub.submit(filename = 'preprocessed_testX.joblib', modelname = '
    res50_pretrained_model.pt)
```

let $\mathcal{F}$ be the class of functions the the network architecture can satisfy
$\forall f \in \mathcal{F}, \exists$ weights, biases that can be obtained through training

let f^* be the truth function to be find, usually $f^* \notin \mathcal{F}$, so find a $f^*_{\mathcal{F}} \in \mathcal{F}$ that is close to f
L is loss function, $f^*_{\mathcal{F}} := arg\,min_f L(X, y, f)$ subject to $f \in \mathcal{F}$
weights of neural network
$\{\alpha_{0m}, \alpha_m; m = 1, 2, ..., M \in \mathbb{N}\}$ $M(p+1)$ weights
$\{\beta_{0k}, \beta_k; k = 1, 2, ..., K \in \mathbb{N}\}$ $K(M+1)$ weights
θ is full set of weights, sum of squares error is $R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2$
suppose there is a architecture with a larger function class $\mathcal{F}'$,
$\sup\{|f^*_{\mathcal{F}'}| - |f^*|\} < \sup\{|f^*_{\mathcal{F}}| - |f^*|\}$ is not guaranteed
if $\mathcal{F} \not\subset \mathcal{F}'$, $f^*_{\mathcal{F}'}$ move closer to f^* is not guaranteed with a larger function class
so use nested function classes $\mathcal{F}_1 \subset ... \subset \mathcal{F}_{n \in \mathbb{N}}$
so if smaller function classes are subsets of the larger function classes,
we can obtain more closeness to f^* as we increase $\mathcal{F}$
so train the new layers into identity function $f(\mathbf{x}) = \mathbf{x}$
as one of the elements in each additional layer using a residual block
let $\mathbf{x}$ be input, $f(\mathbf{x})$ be the underlying mapping,
that we desire to be learned as the input to the top activation function $f(\mathbf{x}) = g(\mathbf{x}) + \mathbf{x}$
suppose a function class $\mathcal{F}$ need to learn the mapping $f(\mathbf{x})$,
the larger function class $\mathcal{F}'$ need to learn the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$
if the desired mapping is the identity $f(\mathbf{x}) = \mathbf{x}$,
then the residual mapping becomes $g(\mathbf{x}) = 0$
then only need to make the weights and biases of convolutional layer
and fully connected layer to zero.
through the residual connections across layers of residual blocks,
inputs can have a faster forward propagation.

```
class Block(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, i_downsample=None, stride=1):
        super(Block, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,
            stride=stride, bias=False)
        self.batch_norm1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1,
            stride=stride, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(out_channels)

        self.i_downsample = i_downsample
        self.stride = stride
        self.relu = nn.ReLU()
```

experimental results
image 0, 1, 2 represent 3 images of 3 different views in each sample
Using ResNet50 with image 0 with 20 epochs received a root mean square error score of 0.00707.
Using ResNet50 with image 0 with 25 epochs received a root mean square error score of 0.01072.
Using ResNet50 with image 0 with 22 epochs received a root mean square error score of 0.0149.
Using ResNet50 with image 0 with 15 epochs received a root mean square error score of 0.01815.

Using ResNet50 with image 2 with 20 epochs received a root mean square error score of 0.02464.
Using ResNet50 with image 0 with 30 epochs received a root mean square error score of 0.0269.
Using ResNet50 with image 0 with 40 epochs received a root mean square error score of 0.03257.
Using ResNet50 with image 0, 1, 2 combined using image 0 as test x received a root mean square error score of 0.03367.
Using ResNet50 with image 1 with 20 epochs received a root mean square error score of 0.03531.

discussion
Using ResNet50 with image 0 with 20 epochs received a root mean square error score of 0.00707. Unexpectedly, using image 0 performed better than using all of image 0, 1, 2 at the same time. Intuitively, one might think that using three images together would have a lower error since it learned from not only one view of the robot hand, but three different views, but it seems that optimization is not so straightforward. After a threshold of epochs, the root mean square error increased, this might be due to overfitting. The reason why using only image 0 performed better than using all of image 0, 1, 2 at the same time might be due to the implementation of the convolutional neural network (an implementation in the fragment shader is shown below) since the there is a high amount of variability among the different views of the robot hand. When using three images at the same time, the original information locality among adjacent pixels are disturbed as compared with using only one view from one image in each sample, since the convolution layer performs convolution operation each time between a small portion of the input matrix with a kernel matrix and store the sum of products into each entry of the output matrix.

let $f(x, y)$ be the original input image,
ω be the weight filter kernel,
$g(x, y)$ be the output filtered image

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x-dx, y-dy)$$

```
vec3 conv2d(mat3 kernel, sampler2D s, vec2 uv)
{
    vec3 frag;
    mat3 region[] = region3x3(s, uv);
    for(int i = 0; i < n_channel, ++i)
    {
        mat3 region_channel = region[i],
        c = matrix_component_multiply(kernel, region_channel);
        frag[i] = c[0][0] + c[0][1] + c[0][2]
                + c[1][0] + c[1][1] + c[1][2]
                + c[2][0] + c[2][1] + c[2][2];
    }
    return frag;
}
```

future work
It seems there are a lot of performance penalty in terms of time taken due to the implementation of the language and the Torch library. For example, it took a lot of time to preprocess the data and traverse through the dataset. It would be a lot faster if the data is organized better and stored in memory in a way that have better spatial and temporal locality. In the future, a combination of C++, CUDA, and GLSL would be a better choice for training and preprocessing, in which training the convolutional neural network would be done inside the fragment shader exploiting the parallelism of GPUs.