Visual cortex decoding

Tianye Zhao

August 17, 2020

1 BOLD fMRI pre-processing

Due to disk storage and computation capacity, we used 8/12 runs of *LetterImage*, 17/20 runs of *ArrificalImage*, 23/24 runs of *NaturalImageTest* and 87/120 runs of *NaturalImageTraining* in *sub-01*.

After applied advised methods in *Final project part 1 DEMO* video to register all BOLD fMRI modalities to *sub-01_NaturalImageTraining01-run01*, we used `3dTproject` method in `AFNI` for bandpass filtering from 0.01 to 0.1 and smoothing. We also used `3dTstat` to calculate mean and standard deviation, then used `3dcalc` to normalize each voxel with mean zero and standard deviation 1.

2 Samples extraction

Applied methods in *final project final hint* video, we kept only back 1/3 ($96 \times 1/3 = 32$ voxels) brain along axial view, trim 8 voxels on both sides of sagittal view and pad coronal view to same size as sagittal view for easy manipulation when feeding to neural network. Reshaping samples as a list of size 55 according to each stimulus in each run, we could further average samples corresponding to each stimulus image. Finally, we got samples shape with $1300 \times 80 \times 32 \times 80$ which includes all images (1200 *NaturalImageTraining*, 50 *NaturalImageTest*, 40 *ArtificialImage*, 10 *LetterImage*).

We tried to use freesurfer's method for inflating and flattening the cortex, it was extreme time consuming. Thus we decided to use 3d representation.

We also converted images to grayscale and resized to 80×80 pixels, and normalized the pixel value between -1 and 1 before feeding to network, which would be easier for us ti optimize the network. As more color channels and larger image size led to more parameters in network and it would take long time for gradient descent convergence and sometimes lead to GPU memory exhaust on *Google Colab* during training (network couldn't successfully be built if *Colab* allocate less than 15GB for us).

3 U-shape deep convolution image reconstruction

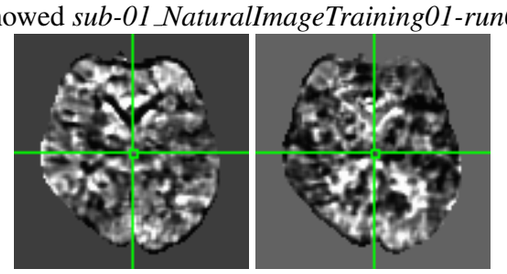
We adapted the idea from *3DUnetCNN* provided in slides. However, unlike U-net, the feature maps we extracted in down-sampling stage has no spatial relationship with output images, we simple dropped feature maps concatenation steps in up-sampling (up-convolution) stage.

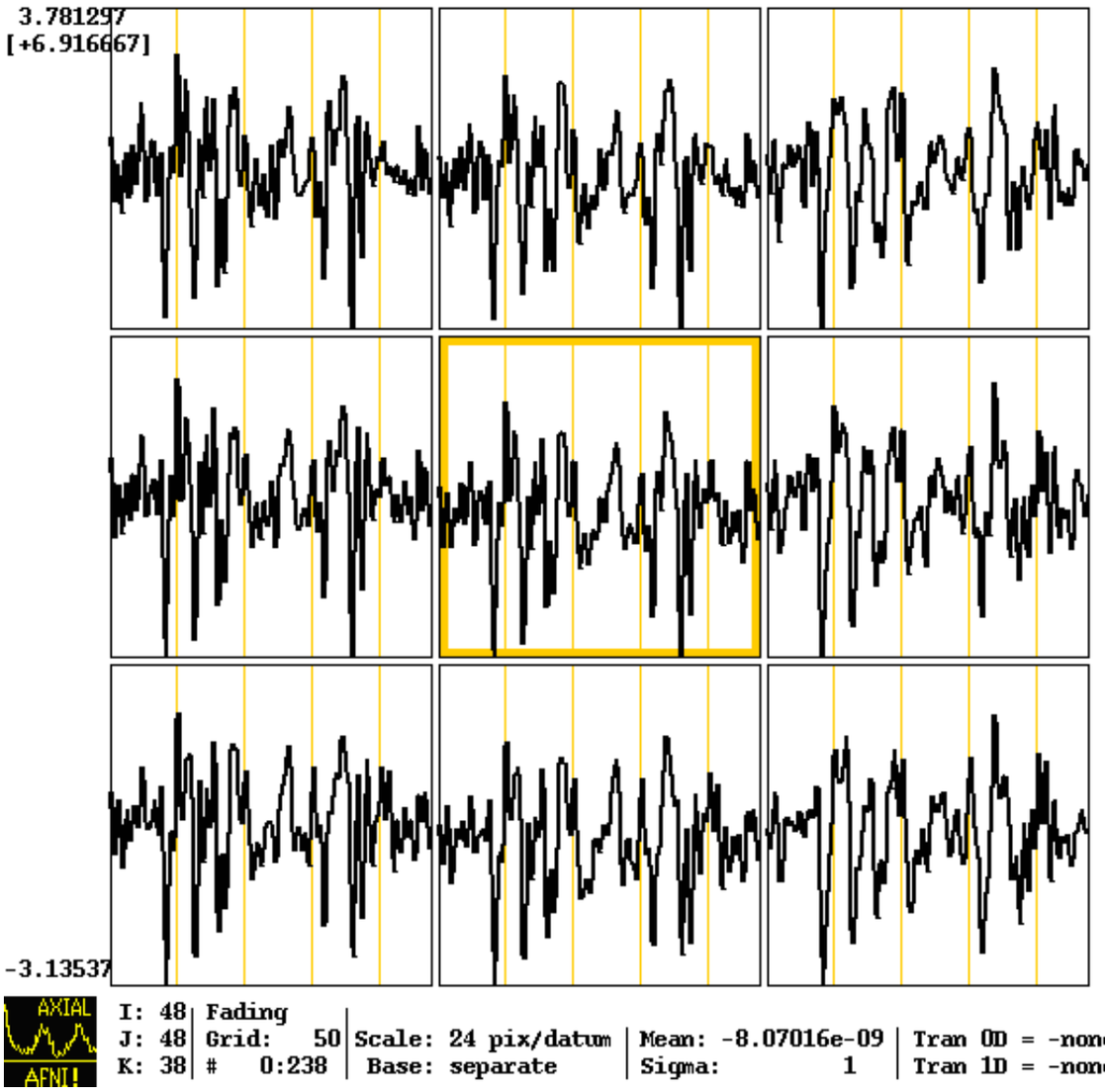
In feature extraction stage, we used two 2D convolution layers along coronal view and sagittal view instead of two 3D convolution layers. Comparing to 3D convolution, the proposed network architecture has only around 1/3 of parameters (20 million down to 7 million), it saved training time and avoided frequent GPU memory exhaust. Moreover, we showed that it performed well from results. Before up-sampling, we could get 2D features through reducing 3D features from $5 \times 2 \times 5$ to 5×5 using `tensorflow.math.reduce_max()`.

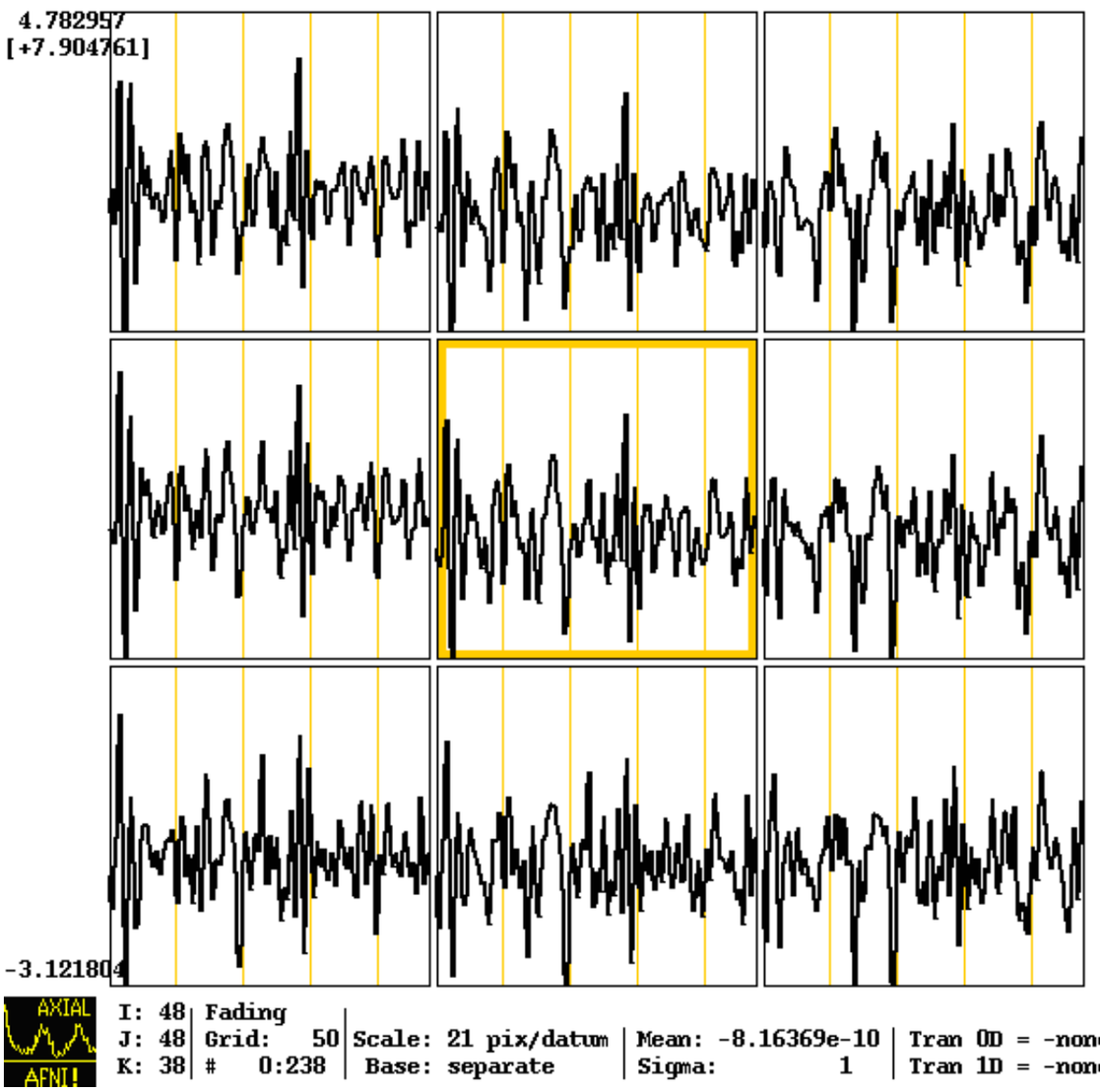
In up-sampling (up-convolution) stage, we used the idea of creating a image generator in *Deep Convolution Generative Adversarial Network* (*DCGAN*). Instead of feeding random noise in *DCGAN*, we fed 2d features extracted from previous stage and tried to generate meaningful images.

3.1 BOLD co-registered runs

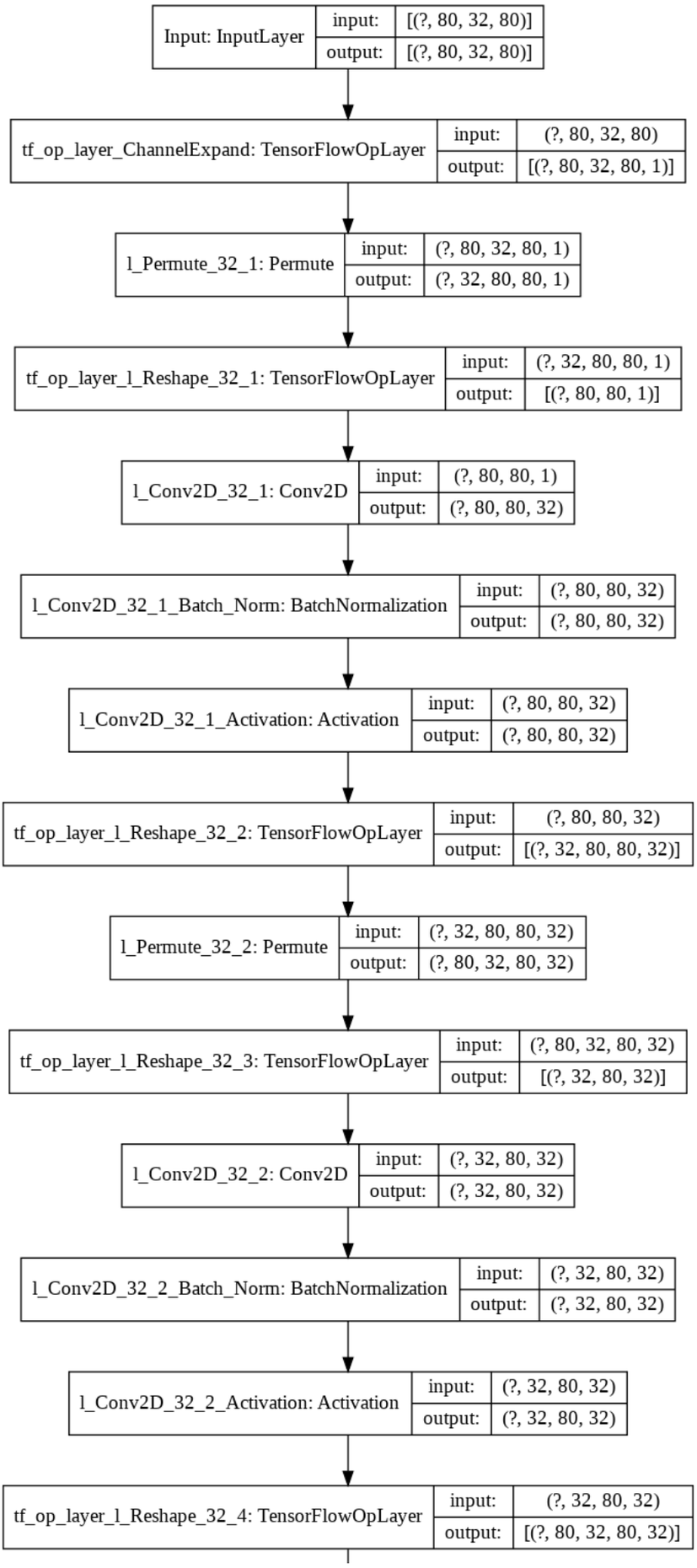
Showed *sub-01_NaturalImageTraining01-run01* and *sub-01_NaturalImageTraining01-run03* as following:

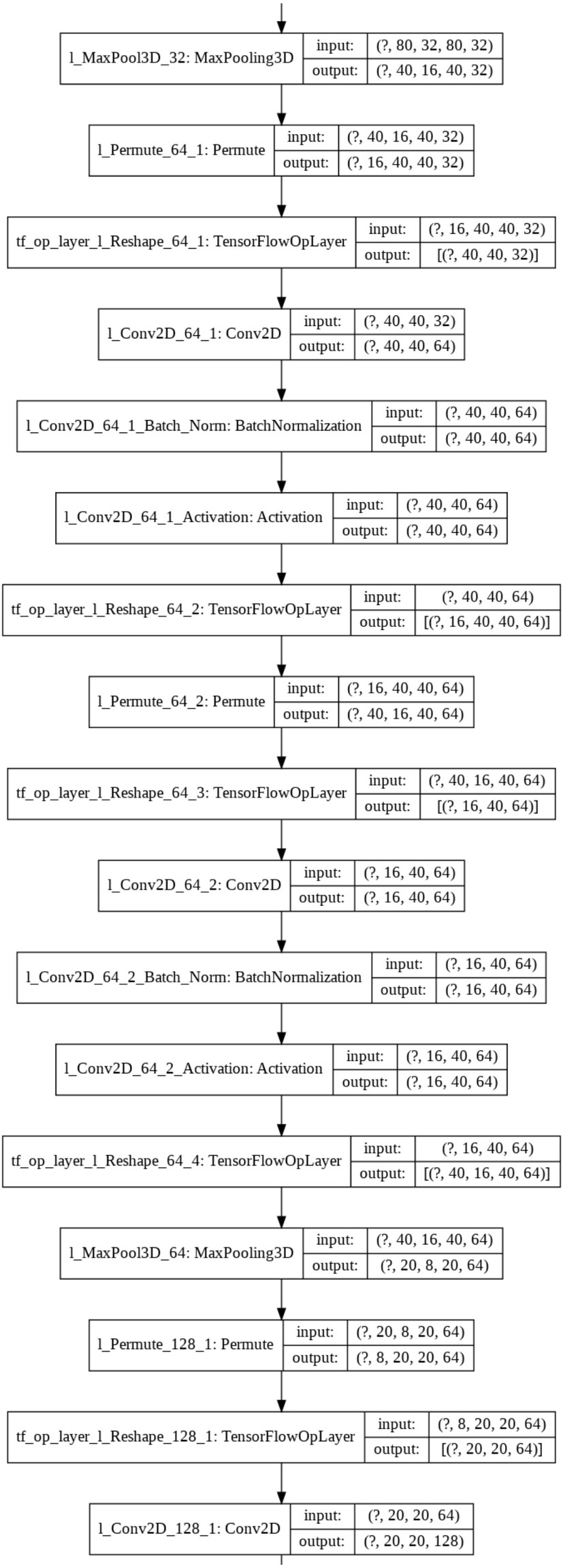


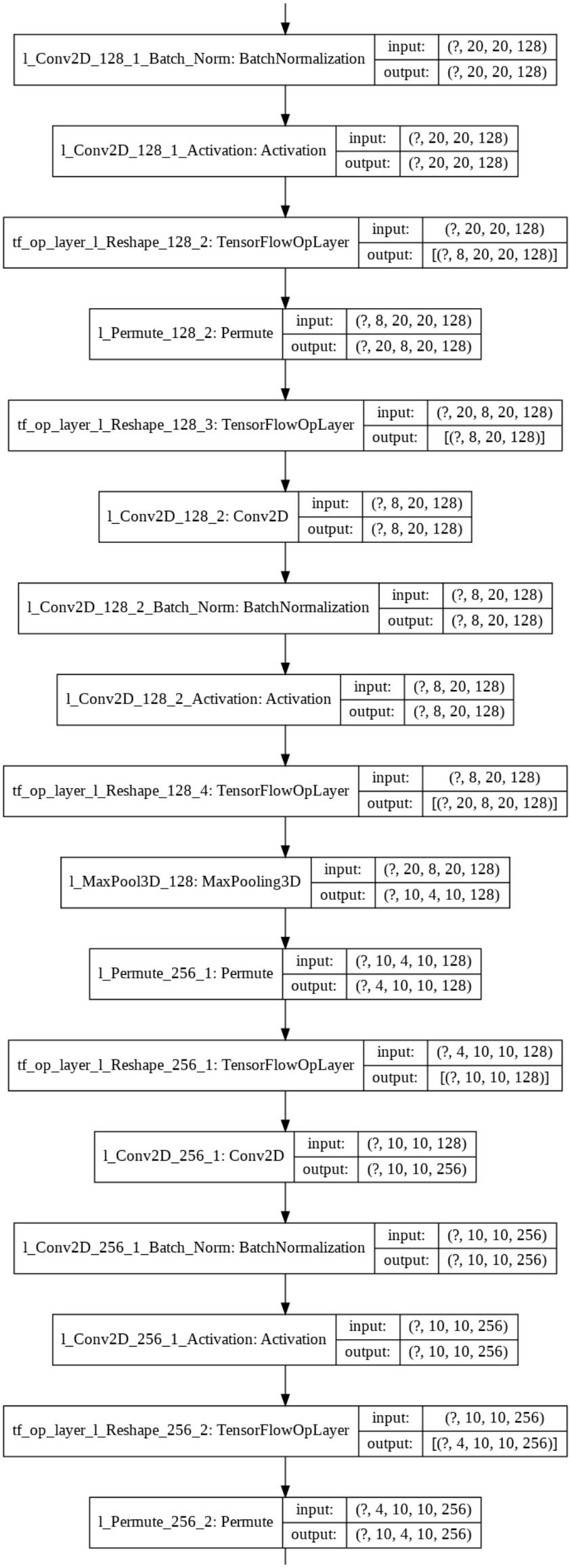

3.781297
[+6.916667]
-3.13537
AXIAL
AFNI
I: 48 J: 48 K: 38
Fading
Grid: 50
0:238
Scale: 24 pix/datum
Base: separate
Mean: -8.07016e-09
Sigma: 1
Tran 0D = -non
Tran 1D = -non

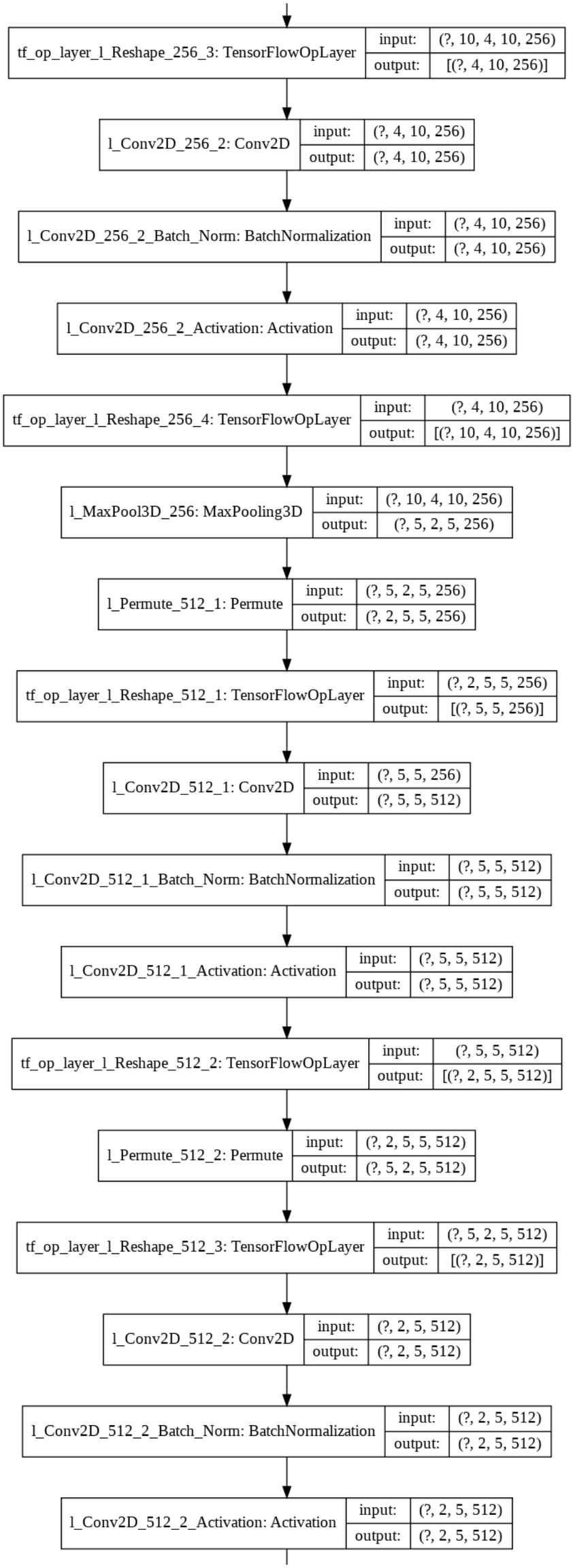

4.782957
[+7.904761]
-3.121804
AXIAL
AFNI
I: 48 J: 48 K: 38
Fading
Grid: 50
0:238
Scale: 21 pix/datum
Base: separate
Mean: -8.16369e-10
Sigma: 1
Tran 0D = -non
Tran 1D = -non

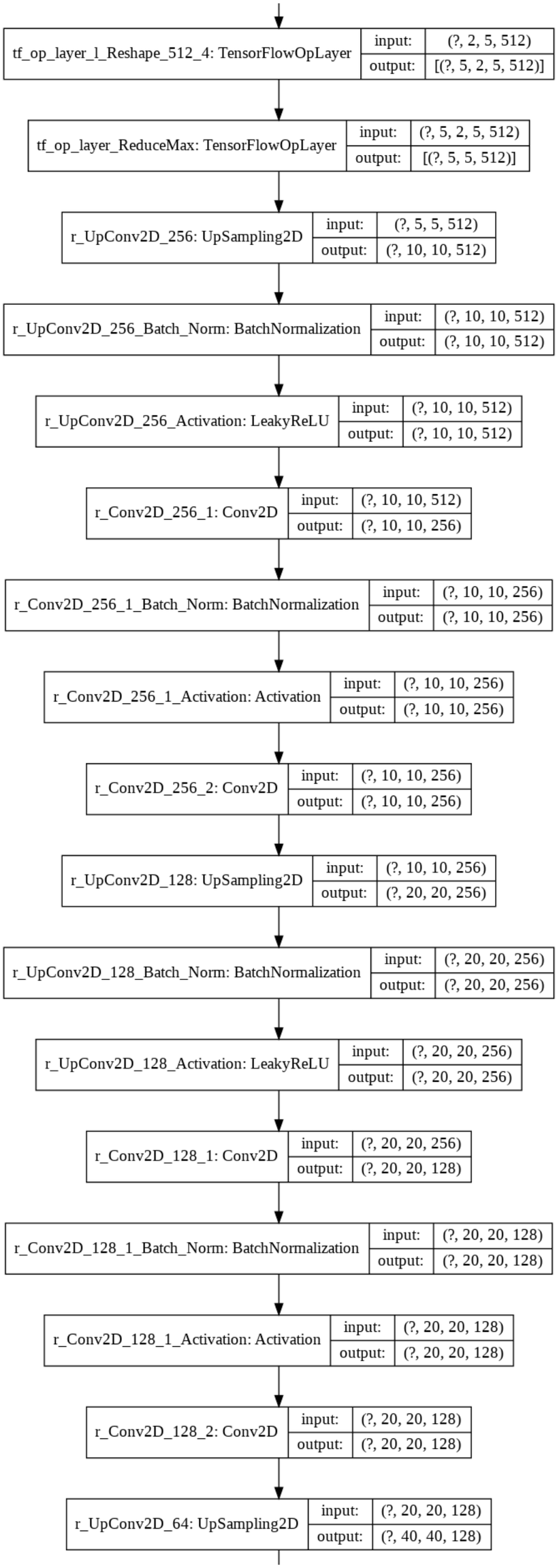
3.2 Neural network diagram

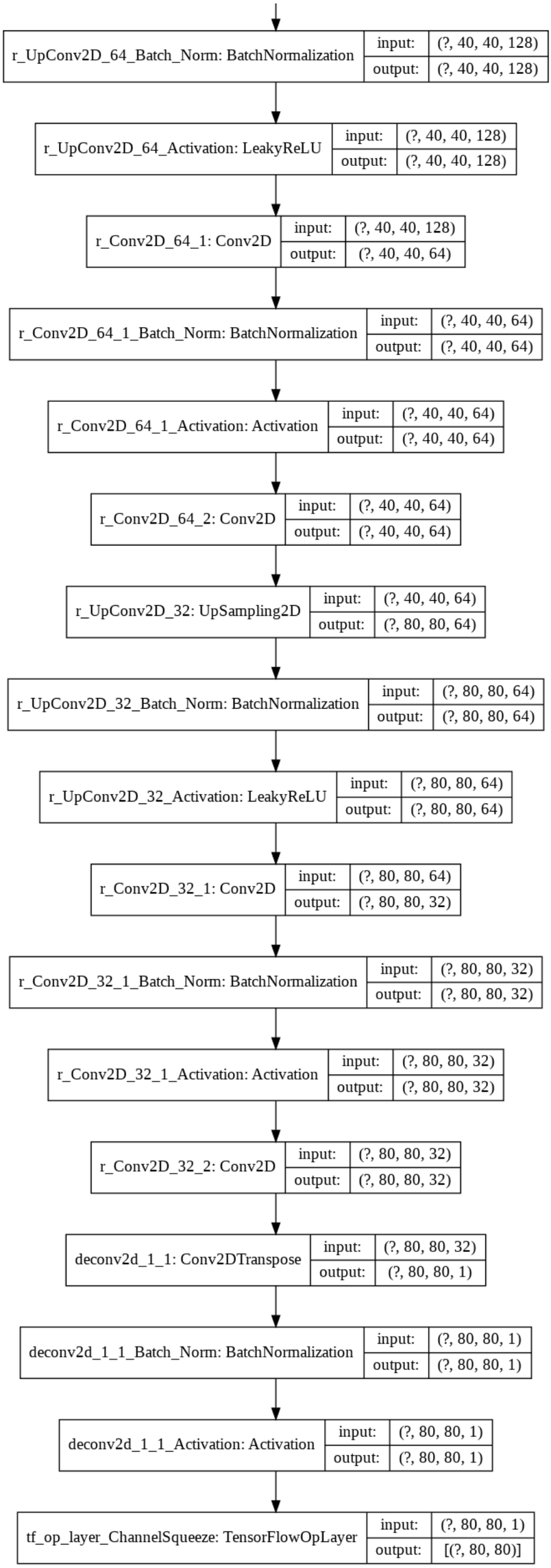



l_MaxPool3D_32: MaxPooling3D
input: (?, 80, 32, 80, 32)
output: (?, 40, 16, 40, 32)
l_Permute_64_1: Permute
input: (?, 40, 16, 40, 32)
output: (?, 16, 40, 40, 32)
tf_op_layer_l_Reshape_64_1: TensorFlowOpLayer
input: (?, 16, 40, 40, 32)
output: [(?, 40, 40, 32)]
l_Conv2D_64_1: Conv2D
input: (?, 40, 40, 32)
output: (?, 40, 40, 64)
l_Conv2D_64_1_Batch_Norm: BatchNormalization
input: (?, 40, 40, 64)
output: (?, 40, 40, 64)
l_Conv2D_64_1_Activation: Activation
input: (?, 40, 40, 64)
output: (?, 40, 40, 64)
tf_op_layer_l_Reshape_64_2: TensorFlowOpLayer
input: (?, 40, 40, 64)
output: [(?, 16, 40, 40, 64)]
l_Permute_64_2: Permute
input: (?, 16, 40, 40, 64)
output: (?, 40, 16, 40, 64)
tf_op_layer_l_Reshape_64_3: TensorFlowOpLayer
input: (?, 40, 16, 40, 64)
output: [(?, 16, 40, 64)]
l_Conv2D_64_2: Conv2D
input: (?, 16, 40, 64)
output: (?, 16, 40, 64)
l_Conv2D_64_2_Batch_Norm: BatchNormalization
input: (?, 16, 40, 64)
output: (?, 16, 40, 64)
l_Conv2D_64_2_Activation: Activation
input: (?, 16, 40, 64)
output: (?, 16, 40, 64)
tf_op_layer_l_Reshape_64_4: TensorFlowOpLayer
input: (?, 16, 40, 64)
output: [(?, 40, 16, 40, 64)]
l_MaxPool3D_64: MaxPooling3D
input: (?, 40, 16, 40, 64)
output: (?, 20, 8, 20, 64)
l_Permute_128_1: Permute
input: (?, 20, 8, 20, 64)
output: (?, 8, 20, 20, 64)
tf_op_layer_l_Reshape_128_1: TensorFlowOpLayer
input: (?, 8, 20, 20, 64)
output: [(?, 20, 20, 64)]
l_Conv2D_128_1: Conv2D
input: (?, 20, 20, 64)
output: (?, 20, 20, 128)

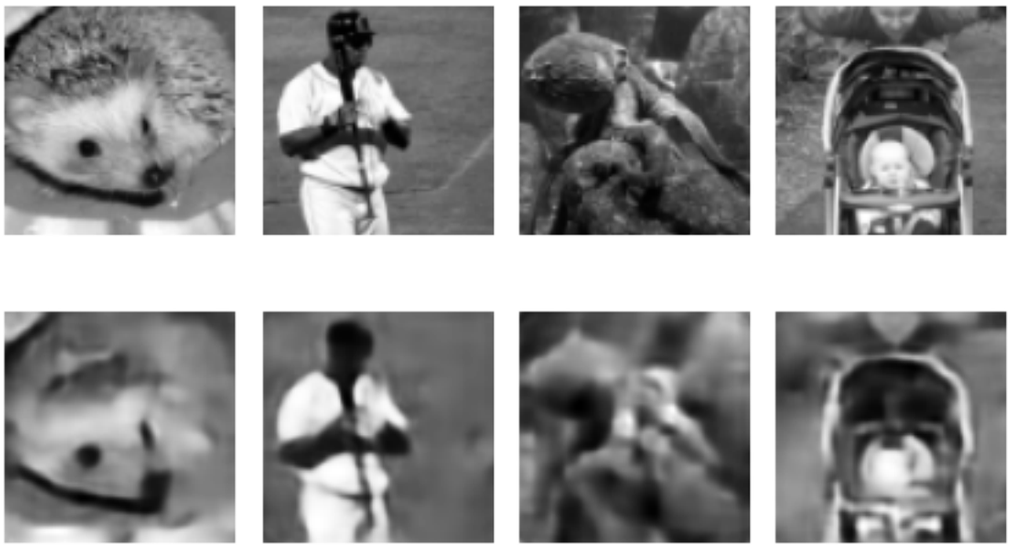

l_Conv2D_128_1_Batch_Norm: BatchNormalization
input: (?, 20, 20, 128)
output: (?, 20, 20, 128)
l_Conv2D_128_1_Activation: Activation
input: (?, 20, 20, 128)
output: (?, 20, 20, 128)
tf_op_layer_l_Reshape_128_2: TensorFlowOpLayer
input: (?, 20, 20, 128)
output: [(?, 8, 20, 20, 128)]
l_Permute_128_2: Permute
input: (?, 8, 20, 20, 128)
output: (?, 20, 8, 20, 128)
tf_op_layer_l_Reshape_128_3: TensorFlowOpLayer
input: (?, 20, 8, 20, 128)
output: [(?, 8, 20, 128)]
l_Conv2D_128_2: Conv2D
input: (?, 8, 20, 128)
output: (?, 8, 20, 128)
l_Conv2D_128_2_Batch_Norm: BatchNormalization
input: (?, 8, 20, 128)
output: (?, 8, 20, 128)
l_Conv2D_128_2_Activation: Activation
input: (?, 8, 20, 128)
output: (?, 8, 20, 128)
tf_op_layer_l_Reshape_128_4: TensorFlowOpLayer
input: (?, 8, 20, 128)
output: [(?, 20, 8, 20, 128)]
l_MaxPool3D_128: MaxPooling3D
input: (?, 20, 8, 20, 128)
output: (?, 10, 4, 10, 128)
l_Permute_256_1: Permute
input: (?, 10, 4, 10, 128)
output: (?, 4, 10, 10, 128)
tf_op_layer_l_Reshape_256_1: TensorFlowOpLayer
input: (?, 4, 10, 10, 128)
output: [(?, 10, 10, 128)]
l_Conv2D_256_1: Conv2D
input: (?, 10, 10, 128)
output: (?, 10, 10, 256)
l_Conv2D_256_1_Batch_Norm: BatchNormalization
input: (?, 10, 10, 256)
output: (?, 10, 10, 256)
l_Conv2D_256_1_Activation: Activation
input: (?, 10, 10, 256)
output: (?, 10, 10, 256)
tf_op_layer_l_Reshape_256_2: TensorFlowOpLayer
input: (?, 10, 10, 256)
output: [(?, 4, 10, 10, 256)]
l_Permute_256_2: Permute
input: (?, 4, 10, 10, 256)
output: (?, 10, 4, 10, 256)


tf_op_layer_l_Reshape_256_3: TensorFlowOpLayer
input: (?, 10, 4, 10, 256)
output: [(?, 4, 10, 256)]
l_Conv2D_256_2: Conv2D
input: (?, 4, 10, 256)
output: (?, 4, 10, 256)
l_Conv2D_256_2_Batch_Norm: BatchNormalization
input: (?, 4, 10, 256)
output: (?, 4, 10, 256)
l_Conv2D_256_2_Activation: Activation
input: (?, 4, 10, 256)
output: (?, 4, 10, 256)
tf_op_layer_l_Reshape_256_4: TensorFlowOpLayer
input: (?, 4, 10, 256)
output: [(?, 10, 4, 10, 256)]
l_MaxPool3D_256: MaxPooling3D
input: (?, 10, 4, 10, 256)
output: (?, 5, 2, 5, 256)
l_Permute_512_1: Permute
input: (?, 5, 2, 5, 256)
output: (?, 2, 5, 5, 256)
tf_op_layer_l_Reshape_512_1: TensorFlowOpLayer
input: (?, 2, 5, 5, 256)
output: [(?, 5, 5, 256)]
l_Conv2D_512_1: Conv2D
input: (?, 5, 5, 256)
output: (?, 5, 5, 512)
l_Conv2D_512_1_Batch_Norm: BatchNormalization
input: (?, 5, 5, 512)
output: (?, 5, 5, 512)
l_Conv2D_512_1_Activation: Activation
input: (?, 5, 5, 512)
output: (?, 5, 5, 512)
tf_op_layer_l_Reshape_512_2: TensorFlowOpLayer
input: (?, 5, 5, 512)
output: [(?, 2, 5, 5, 512)]
l_Permute_512_2: Permute
input: (?, 2, 5, 5, 512)
output: (?, 5, 2, 5, 512)
tf_op_layer_l_Reshape_512_3: TensorFlowOpLayer
input: (?, 5, 2, 5, 512)
output: [(?, 2, 5, 512)]
l_Conv2D_512_2: Conv2D
input: (?, 2, 5, 512)
output: (?, 2, 5, 512)
l_Conv2D_512_2_Batch_Norm: BatchNormalization
input: (?, 2, 5, 512)
output: (?, 2, 5, 512)
l_Conv2D_512_2_Activation: Activation
input: (?, 2, 5, 512)
output: (?, 2, 5, 512)

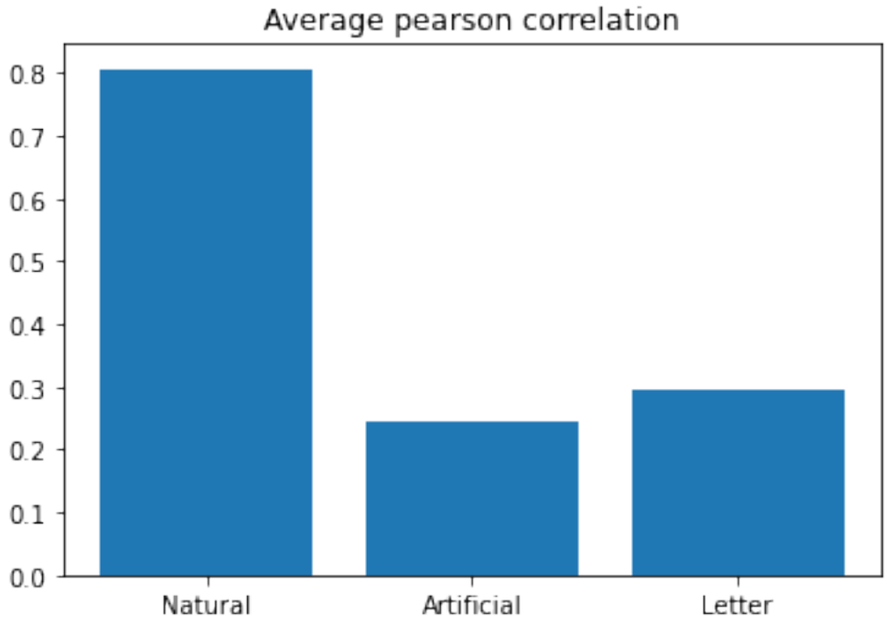

tf_op_layer_l_Reshape_512_4: TensorFlowOpLayer
input: (?, 2, 5, 512)
output: [(?, 5, 2, 5, 512)]
tf_op_layer_ReduceMax: TensorFlowOpLayer
input: (?, 5, 2, 5, 512)
output: [(?, 5, 5, 512)]
r_UpConv2D_256: UpSampling2D
input: (?, 5, 5, 512)
output: (?, 10, 10, 512)
r_UpConv2D_256_Batch_Norm: BatchNormalization
input: (?, 10, 10, 512)
output: (?, 10, 10, 512)
r_UpConv2D_256_Activation: LeakyReLU
input: (?, 10, 10, 512)
output: (?, 10, 10, 512)
r_Conv2D_256_1: Conv2D
input: (?, 10, 10, 512)
output: (?, 10, 10, 256)
r_Conv2D_256_1_Batch_Norm: BatchNormalization
input: (?, 10, 10, 256)
output: (?, 10, 10, 256)
r_Conv2D_256_1_Activation: Activation
input: (?, 10, 10, 256)
output: (?, 10, 10, 256)
r_Conv2D_256_2: Conv2D
input: (?, 10, 10, 256)
output: (?, 10, 10, 256)
r_UpConv2D_128: UpSampling2D
input: (?, 10, 10, 256)
output: (?, 20, 20, 256)
r_UpConv2D_128_Batch_Norm: BatchNormalization
input: (?, 20, 20, 256)
output: (?, 20, 20, 256)
r_UpConv2D_128_Activation: LeakyReLU
input: (?, 20, 20, 256)
output: (?, 20, 20, 256)
r_Conv2D_128_1: Conv2D
input: (?, 20, 20, 256)
output: (?, 20, 20, 128)
r_Conv2D_128_1_Batch_Norm: BatchNormalization
input: (?, 20, 20, 128)
output: (?, 20, 20, 128)
r_Conv2D_128_1_Activation: Activation
input: (?, 20, 20, 128)
output: (?, 20, 20, 128)
r_Conv2D_128_2: Conv2D
input: (?, 20, 20, 128)
output: (?, 20, 20, 128)
r_UpConv2D_64: UpSampling2D
input: (?, 20, 20, 128)
output: (?, 40, 40, 128)


r_UpConv2D_64_Batch_Norm: BatchNormalization
input: (?, 40, 40, 128)
output: (?, 40, 40, 128)
r_UpConv2D_64_Activation: LeakyReLU
input: (?, 40, 40, 128)
output: (?, 40, 40, 128)
r_Conv2D_64_1: Conv2D
input: (?, 40, 40, 128)
output: (?, 40, 40, 64)
r_Conv2D_64_1_Batch_Norm: BatchNormalization
input: (?, 40, 40, 64)
output: (?, 40, 40, 64)
r_Conv2D_64_1_Activation: Activation
input: (?, 40, 40, 64)
output: (?, 40, 40, 64)
r_Conv2D_64_2: Conv2D
input: (?, 40, 40, 64)
output: (?, 40, 40, 64)
r_UpConv2D_32: UpSampling2D
input: (?, 40, 40, 64)
output: (?, 80, 80, 64)
r_UpConv2D_32_Batch_Norm: BatchNormalization
input: (?, 80, 80, 64)
output: (?, 80, 80, 64)
r_UpConv2D_32_Activation: LeakyReLU
input: (?, 80, 80, 64)
output: (?, 80, 80, 64)
r_Conv2D_32_1: Conv2D
input: (?, 80, 80, 64)
output: (?, 80, 80, 32)
r_Conv2D_32_1_Batch_Norm: BatchNormalization
input: (?, 80, 80, 32)
output: (?, 80, 80, 32)
r_Conv2D_32_1_Activation: Activation
input: (?, 80, 80, 32)
output: (?, 80, 80, 32)
r_Conv2D_32_2: Conv2D
input: (?, 80, 80, 32)
output: (?, 80, 80, 32)
deconv2d_1_1: Conv2DTranspose
input: (?, 80, 80, 32)
output: (?, 80, 80, 1)
deconv2d_1_1_Batch_Norm: BatchNormalization
input: (?, 80, 80, 1)
output: (?, 80, 80, 1)
deconv2d_1_1_Activation: Activation
input: (?, 80, 80, 1)
output: (?, 80, 80, 1)
tf_op_layer_ChannelSqueeze: TensorFlowOpLayer
input: (?, 80, 80, 1)
output: [(?, 80, 80)]

3.3 Natural image reconstruction



3.4 Artificial image reconstruction



As we can see from above, our model performed well on natural images, it basically gave us the sketch of geometrical shapes with rough details in image. But it failed on artificial images. This is because there are much more training samples of natural images than artificial images.

4 Further work

4.1 3D features projection

Instead of simply reduce maximum value along an axis to get 2D feature, a dimension reduction technique like PCA or SVD may produce better feature representation from 3D to 2D.

4.2 SSIM loss function

In our model, we simply used `keras.losses.MeanSquaredError()` during training phase. There are better options like `tensorflow.image.ssim()` which would be more accurate to measure structural similarity between images. However, we failed to train the network with custom *ssim* loss function, it led to GPU memory exhaust on our training data.

4.3 L-BFGS optimizer

We used `keras.optimizers.Adam()` in our model. As suggested in slides, L-BFGS is used. There exists `tensorflow_probability.optimizer.lbfgs_minimize()`. It also caused memory exhaust when we used our custom *l-bfgs* optimizer.

4.4 Discriminator and DCGAN

Since we already adapted the idea of generator in *DCGAN*, we may further create a discriminator to examine the output from our network. Further formalize a *DCGAN* model with a U-shape generator and a discriminator. We have implemented code, but haven't successfully optimized the whole *DCGAN* model due to time-constraint and computation capacity.

4.5 Color image prediction

Instead of predict RGB channels, we could transform RGB to HSV which is closer to how human perceive color, and run whole network to predict HSV values.