lab08_fine_tune_partial

April 23, 2018

Lab 8: Transfer Learning with a Pre-Trained Deep Neural Network

As we discussed earlier, state-of-the-art neural networks involve millions of parameters that are prohibitively difficult to train from scratch. In this lab, we will illustrate a powerful technique called *fine-tuning* where we start with a large pre-trained network and then re-train only the final layers to adapt to a new task. The method is also called *transfer learning* and can produce excellent results on very small datasets with very little computational time.

This lab is based partially on this excellent blog. In performing the lab, you will learn to: * Build a custom image dataset * Fine tune the final layers of an existing deep neural network for a new classification task. * Load images with a `DataGenerator`.

You may run the lab on a CPU machine (like your laptop) or a GPU. See the notes on setting up a GPU instance on Google Cloud Platform. The GPU training is much faster (< 1 minute). But, even the CPU machine training time will be less than 20 minutes.

0.1 Create a Dataset

In this example, we will try to develop a classifier that can discriminate between two classes: `cars` and `bicycles`. One could imagine this type of classifier would be useful in vehicle vision systems. The first task is to build a dataset.

TODO: Create training and test datasets with: * 1000 training images of cars * 1000 training images of bicylces * 300 test images of cars * 300 test images of bicylces * The images don't need to be the same size. But, you can reduce the resolution if you need to save disk space.

The images should be organized in the following directory structure:

```
./train
    /car
       car_0000.jpg
       car_0001.jpg
       ...
       car_0999.jpg
    /bicycle
       bicycle_0000.jpg
       bicycle_0001.jpg
       ...
       bicycle_0999.jpg
./test
    /car
       car_0000.jpg
```

```
      car_0001.jpg
      ...
      car_0299.jpg
   /bicycle
      bicycle_0000.jpg
      bicycle_0001.jpg
      ...
      bicycle_0299.jpg
```

The naming of the files within the directories does not matter. The `ImageDataGenerator` class below will find the filenames. Just make sure there are the correct number of files in each directory.

A nice automated way of building such a dataset if through the FlickrAPI.

0.2 Loading a Pre-Trained Deep Network

We follow the VGG16 demo to load a pre-trained deep VGG16 network. We first load the appropriate Keras packages.

```
In [1]: import keras
```

```
/usr/local/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the se
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
In [2]: from keras import applications
        from keras.preprocessing.image import ImageDataGenerator
        from keras import optimizers
        from keras.models import Sequential
        from keras.layers import Dropout, Flatten, Dense
```

We also load some standard packages.

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Clear the Keras session.

```
In [4]: # TODO
        keras.backend.clear_session()
```

Set the dimensions of the input image. The sizes below would work on a CPU machine. But, if you have a GPU image, you can use a larger image size, like 150 x 150.

```
In [5]: # TODO:  Set to larger values if you are using a GPU.
        nrow = 64
        ncol = 64
```

Now we follow the VGG16 demo and load the deep VGG16 network. Alternatively, you can use any other pre-trained model in keras. When using the `applications.VGG16` method you will need to: * Set `include_top=False` to not include the top layer * Set the `image_shape` based on the above dimensions. Remember, `image_shape` should be height x width x 3 since the images are color.

```
In [6]: # TODO:  Load the VGG16 network
        input_shape = (nrow, ncol, 3)
        base_model = applications.VGG16(weights='imagenet',
                                        input_shape = input_shape,
                                        include_top=False)
```

To create now new model, we create a Sequential model. Then, loop over the layers in `base_model.layers` and add each layer to the new model.

```
In [7]: # Create a new model
        model = Sequential()

        # TODO:  Loop over base_model.layers and add each layer to model
        for layer in base_model.layers:
            model.add(layer)
```

Next, loop through the layers in `model`, and freeze each layer by setting `layer.trainable = False`. This way, you will not have to *re-train* any of the existing layers.

```
In [8]: # TODO
        for layer in model.layers:
            layer.trainable = False
```

Now, add the following layers to `model`: * A `Flatten()` layer which reshapes the outputs to a single channel. * A fully-connected layer with 256 output units and `relu` activation * A final fully-connected layer. Since this is a binary classification, there should be one output and `sigmoid` activation.

```
In [9]: # TODO
        model.add(Flatten())
        model.add(Dense(256, activation='relu'))
        model.add(Dropout(0.5))
        model.add(Dense(1, activation='sigmoid'))
```

Print the model summary. This will display the number of trainable parameters vs. the non-trainable parameters.

```
In [10]: # TODO
         print(model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
input_1 (InputLayer)         (None, 64, 64, 3)         0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 64, 64, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 64, 64, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 32, 32, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 32, 32, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 32, 32, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 16, 16, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 16, 16, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 16, 16, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 16, 16, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 8, 8, 256)         0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 8, 8, 512)         1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 4, 4, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 2, 2, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               524544
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 15,239,489
Trainable params: 524,801
```

```
Non-trainable params: 14,714,688
_________________________________________________________________
None
```

0.3 Using Generators to Load Data

Up to now, the training data has been represented in a large matrix. This is not possible for image data when the datasets are very large. For these applications, the `keras` package provides a `ImageDataGenerator` class that can fetch images on the fly from a directory of images. Using multi-threading, training can be performed on one mini-batch while the image reader can read files for the next mini-batch. The code below creates an `ImageDataGenerator` for the training data. In addition to the reading the files, the `ImageDataGenerator` creates random deformations of the image to expand the total dataset size. When the training data is limited, using data augmentation is very important.

```
In [11]: train_data_dir = './train'
         batch_size = 32
         train_datagen = ImageDataGenerator(rescale=1./255,
                                            shear_range=0.2,
                                            zoom_range=0.2,
                                            horizontal_flip=True)
         train_generator = train_datagen.flow_from_directory(
             train_data_dir,
             target_size=(nrow,ncol),
             batch_size=batch_size,
             class_mode='binary')
```

```
Found 2000 images belonging to 2 classes.
```

Now, create a similar `test_generator` for the test data.

```
In [12]: # TODO
         # test_generator = ...
         test_data_dir = './test'
         batch_size = 32
         test_datagen = ImageDataGenerator(rescale=1./255,
                                           shear_range=0.2,
                                           zoom_range=0.2,
                                           horizontal_flip=True)
         test_generator = test_datagen.flow_from_directory(
             test_data_dir,
             target_size=(nrow,ncol),
             batch_size=batch_size,
             class_mode='binary')
```

```
Found 600 images belonging to 2 classes.
```

The following function displays images that will be useful below.

```
In [13]: # Display the image
         def disp_image(im):
             if (len(im.shape) == 2):
                 # Gray scale image
                 plt.imshow(im, cmap='gray')
             else:
                 # Color image.
                 im1 = (im-np.min(im))/(np.max(im)-np.min(im))*255
                 im1 = im1.astype(np.uint8)
                 plt.imshow(im1)

             # Remove axis ticks
             plt.xticks([])
             plt.yticks([])
```

To see how the `train_generator` works, use the `train_generator.next()` method to get a minibatch of data `X,y`. Display the first 8 images in this mini-batch and label the image with the class label. You should see that bicycles have `y=0` and cars have `y=1`.

```
In [14]: # TODO
         X, y = train_generator.next()
         nplot = 8
         for i in range(nplot):
             plt.subplot(2, 4, i+1)
             disp_image(X[i])
             plt.title(y[i])
             plt.xticks([])
             plt.yticks([])
         plt.show()
```

0.4 Train the Model

Compile the model. Select the correct `loss` function, `optimizer` and `metrics`. Remember that we are performing binary classification.

```
In [15]: # TODO.
         model.compile(loss='binary_crossentropy',
                       optimizer=optimizers.adam(lr=1e-3),
                       metrics=['accuracy'])
```

When using an `ImageDataGenerator`, we have to set two parameters manually: * `steps_per_epoch = training data size // batch_size` * `validation_steps = test data size // batch_size`

We can obtain the training and test data size from `train_generator.n` and `test_generator.n`, respectively.

```
In [16]: # TODO
         steps_per_epoch = train_generator.n // batch_size
         validation_steps = test_generator.n // batch_size
```

Now, we run the fit. If you are using a CPU on a regular laptop, each epoch will take about 3-4 minutes, so you should be able to finish 5 epochs or so within 20 minutes. On a reasonable GPU, even with the larger images, it will take about 10 seconds per epoch. * If you use `(nrow,ncol) = (64,64)` images, you should get around 90% accuracy after 5 epochs. * If you use `(nrow,ncol) = (150,150)` images, you should get around 96% accuracy after 5 epochs. But, this will need a GPU.

You will get full credit for either version. With more epochs, you may get slightly higher, but you will have to play with the damping.

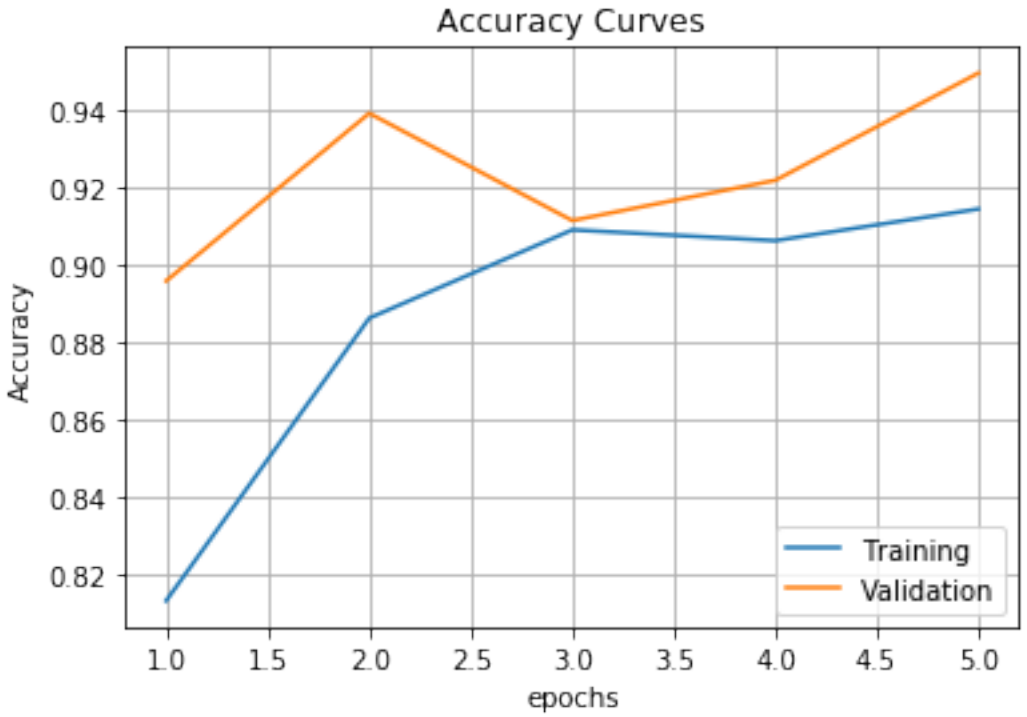
Remember to record the history of the fit, so that you can plot the training and validation accuracy curve.

```
In [17]: nepochs = 5  # Number of epochs

         # Call the fit_generator function
         hist = model.fit_generator(
             train_generator,
             steps_per_epoch=steps_per_epoch,
             epochs=nepochs,
             validation_data=test_generator,
             validation_steps=validation_steps)
```

```
Epoch 1/5
62/62 [==============================] - 68s 1s/step - loss: 0.4064 - acc: 0.8130 - val_loss: 
Epoch 2/5
62/62 [==============================] - 67s 1s/step - loss: 0.2676 - acc: 0.8856 - val_loss: 
Epoch 3/5
62/62 [==============================] - 64s 1s/step - loss: 0.2395 - acc: 0.9098 - val_loss: 
```

```
Epoch 4/5
62/62 [==============================] - 65s 1s/step - loss: 0.2319 - acc: 0.9062 - val_loss: 
Epoch 5/5
62/62 [==============================] - 66s 1s/step - loss: 0.2167 - acc: 0.9143 - val_loss: 
```

In [18]:
```
# Plot the training accuracy and validation accuracy curves on the same figure.

# TO DO
epochsn = np.arange(1, 6)
plt.plot(epochsn, hist.history['acc'])
plt.plot(epochsn, hist.history['val_acc'])

plt.xlabel('epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy Curves')
plt.legend(['Training', 'Validation'])
plt.grid()
plt.show()
```



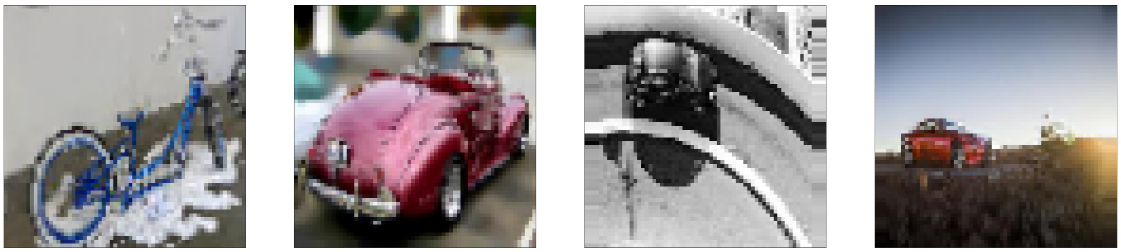
0.5 Evaluate your model on some test images

Apply your trained model to some specific test data and evaluate the accuracy. Also, display the test image and classified label to see whether the prediction makes sense. Note that the result

depends on the test images you use. You may not get very accurate prediction for a particular minibatch. You can run this several times to see the effect on different minibatches.

- Generate a mini-batch `Xtest,ytest` from the `test_generator.next()` method
- Predict the labels using the `model.predict( )` method and compute predicted labels `yhat`.
- Display the images with their ground truth labels and predicted labels
- Look at the images and their predicted labels, to see why the prediction is wrong in some cases.
- If you did not get any prediction error in one minibatch, run it multiple times.

```
In [19]: # TO DO
         Xtest, ytest = test_generator.next()
         ypred = model.predict(Xtest)
         yhat = (ypred>0.5).reshape(batch_size).astype(int)
         Idx = np.where(y != yhat)[0]
         Xerr = []
         yerr = []
         yhaterr = []
         for i in Idx:
             Xerr.append(Xtest[i])
             yerr.append(ytest[i])
             yhaterr.append(yhat[i])

         plt.figure(figsize=(40,40))

         nerr = 4 if len(Xerr) > 4 else len(Xerr)
         for i in range(nerr):
             plt.subplot(1, nerr, i+1)
             disp_image(Xerr[i])
```



0.6 Train the model with Batch Normalization and Dropout

Repeat the previous process with batch normalization and dropout for training.

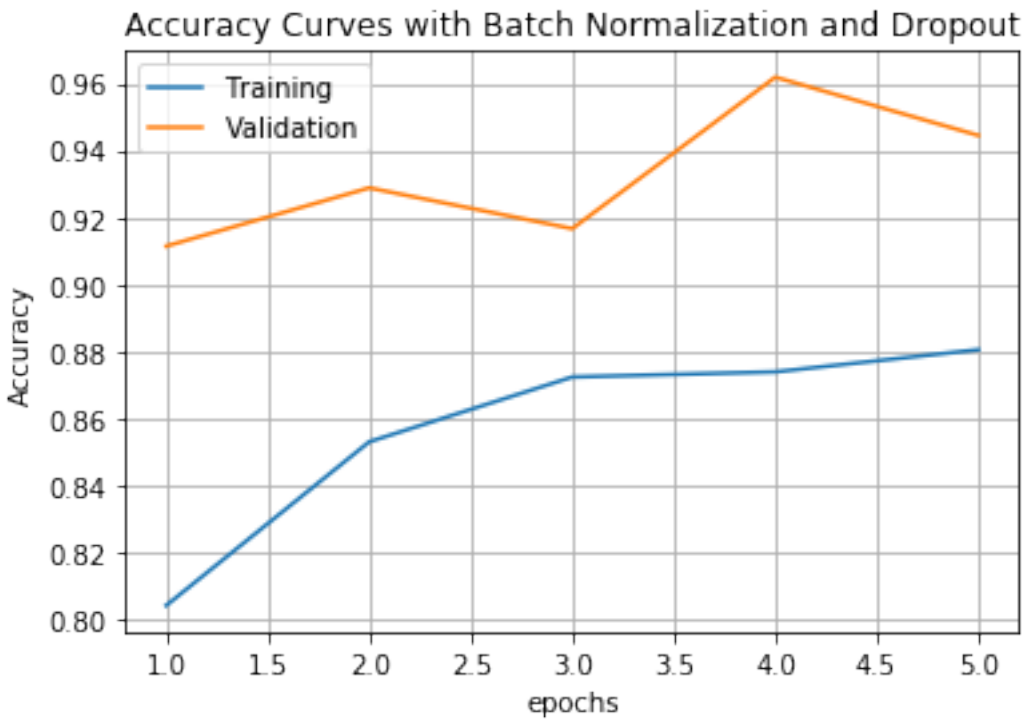
Make sure that you clear the session and name your model with a different name, so that it will not continue to train from the previous model, and you can plot the performance of this model separately.

In [20]:
```
# TO DO
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.models import load_model #save and load models
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization

keras.backend.clear_session()
input_shape = (nrow, ncol, 3)
base_model = applications.VGG16(weights='imagenet',
                                input_shape = input_shape,
                                include_top=False)
model_dropout = Sequential()
for layer in base_model.layers:
    model_dropout.add(layer)
for layer in model_dropout.layers:
    layer.trainable = False

# Set up the model and fit
# TO DO
model_dropout.add(Flatten())
model_dropout.add(BatchNormalization())
model_dropout.add(Dropout(0.5))
model_dropout.add(Dense(256,activation = "relu"))
model_dropout.add(BatchNormalization())
model_dropout.add(Dropout(0.5))
model_dropout.add(Dense(1,activation = "sigmoid"))
print(model_dropout.summary())

model_dropout.compile(loss='binary_crossentropy',
                  optimizer = optimizers.adam(lr=1e-3),
                  metrics=['accuracy'])

hist_dropout = model_dropout.fit_generator(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=nepochs,
    validation_data=test_generator,
    validation_steps=validation_steps)

# Plot the training performance
# TO DO

epochsn=np.arange(1, 6)
plt.plot(epochsn,hist_dropout.history['acc'])
plt.plot(epochsn,hist_dropout.history['val_acc'])
plt.xlabel('epochs')
```

```
        plt.ylabel('Accuracy')
        plt.title('Accuracy Curves with Batch Normalization and Dropout')
        plt.legend(['Training', 'Validation'])
        plt.grid()
        plt.show()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 64, 64, 3)         0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 64, 64, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 64, 64, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 32, 32, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 32, 32, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 32, 32, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 16, 16, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 16, 16, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 16, 16, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 16, 16, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 8, 8, 256)         0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 8, 8, 512)         1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 8, 8, 512)         2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 4, 4, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 4, 4, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 2, 2, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
```

```
_________________________________________________________________
batch_normalization_1 (Batch (None, 2048)              8192
_________________________________________________________________
dropout_1 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               524544
_________________________________________________________________
batch_normalization_2 (Batch (None, 256)               1024
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 15,248,705
Trainable params: 529,409
Non-trainable params: 14,719,296
_________________________________________________________________
None
Epoch 1/5
62/62 [==============================] - 69s 1s/step - loss: 0.5139 - acc: 0.8039 - val_loss: 0
Epoch 2/5
62/62 [==============================] - 65s 1s/step - loss: 0.3770 - acc: 0.8533 - val_loss: 0
Epoch 3/5
62/62 [==============================] - 65s 1s/step - loss: 0.3135 - acc: 0.8735 - val_loss: 0
Epoch 4/5
62/62 [==============================] - 65s 1s/step - loss: 0.3085 - acc: 0.8745 - val_loss: 0
Epoch 5/5
62/62 [==============================] - 66s 1s/step - loss: 0.2986 - acc: 0.8815 - val_loss: 0
```



Question: Observe the difference in the training and validation accuracy curves with and without using batch normalization and dropout. Are the results as expected? Explain.

Answer:

Generally, the curves with batch normalization and dropout should obtain a more accurate result within a shorter time period, since it can avoid overfitting and make the inputs have the same bounds. Now move on to the cases above. It is obvious that the accuracy curve with normalization and dropout has a smoother training curve and higher validation curve, which is caused by the dropout layer.