Problem formulation

This problem involves predicting a price range for new listers on Airbnb. Several factors constitute an 'acceptable' price such as various amenities, location, size, and even pictures. For this problem we want to create a model that uses this information to predict a price, which is categorized into beginner, plus and premium as 0,1,2 respectively. In this problem we have both text and image data being used as input for our independent variables. However, for our dependent or target variable, we actually have two instead of one making it a multi-target classification problem. The output of this problem, however, only predicts for one of those target variables which is price, the other target variable is type which refers to the type of the listing (house, apartment…). So even though the model is trained and predicted with two target variables, only one is used for file submission. There are some advantages of doing this; usually single output models take longer to train and are more computationally expensive than multi-output ones. Data mining functions requires include natural language processing techniques such as tokenizing and other preprocessing methods. Image mining techniques will also be used such as padding, feature extraction, etc. It also requires techniques used to build a model, which in this case starts with a neural net, libraries such as pandas, keras, etc. can be used to formulate these models.

Adding an attention layer

I first ran the model with the template setting to establish a "baseline" to compare against later on. This model mainly produced a result in the low to mid-sixties and the call back (early stopping) function ended the epochs around 13-25 with a patience of 5. I then increased the batch number and disabled early stopping which consistently lowered performance. In the next step I added an attention layer, applying this layer will let the model know what to focus on, for example when there are no signals to narrow down the search, the attention API will iteratively attend to every part of the input data and it will only do this to regions of the input with the most important content. Since attention layer also focuses on local regions, it is less prone and sensitive to global distortion that may skew learning. Applying the attention layer ended up lowering the model performance to a certain degree and also increased training time. It makes sense that training took longer as the attention mechanism adds more weight parameters to the model.

Customizing image layers

To customize the image layers, I first left the CONV2D with the pre-set template parameters and just added a Dense (64) and Dropout (0.5) after the Flatten () layer (keep in mind that my text part did not contain the attention mechanism as it lowered the performance previously). Doing so increased the model performance significantly and I noticed that the model ran for more epochs. Perhaps the problem was more complex than initially assumed, therefore adding these layers helped with image training. Since I noticed a very stable and consistent improvement in the model performance, I added another dense and dropout layer which improved performance slightly therefore I avoided adding more. I then decided to play around with the convolution layer parameters, initially the Conv2D had a filter of 32 and kernel size (16,16), the kernels in the convolutional layer are the convolutional filters so kernel size refers to the weight * height of the filter. Since the problem appears to be more complex than I initially assumed, first I decided to increase the filter number to increase the number of abstractions that the network is able to extract from the data, doing this did not significantly change performance so I decided to add

another Conv2D with a higher filter number than the first one. The reason for this is because the first layer (with a lower number of filters than second) will catch a few features (edge, shape, etc…) of the image and next layers will obtain more complex features based on simple ones and hopefully will avoid more noise. I added another Conv2D (with more filters) and another Maxpooling layer; a max pooling layer progressively reduces the spatial size of the representation to reduce the number of parameters and computation (controls overfitting). This is because the exact location of the feature is less important than its approximate location relative to other features. I also added the activation function 'reLU' to my conv2D layers. I used 'reLU' because it slows the network to converge very quickly and allows for back propagation. Adding these two layers slightly improved model performance.

Fusion layer

I decided to replace the concatenate (tf.concat) function with stack (tf.stack). The only difference between concatenate and stack function is that Concatenating joins a sequence of tensors along an existing axis, and stacking joins a sequence of tensors along a new axis, this means that stack will produce errors in the model training stage if new a new dimension is added. To avoid this problem, I also added a Flatten () layer right after the stack layer to join the two models. In the picture below you can see that a new dimension is added after the stack layer and then removed in the Flatten () layer. Using stack resulted in the best model performance so far.

Layer (type)	Output Shape	Param #	Connected to
input_86 (InputLayer)	[(None, 64, 64, 2)]	0	
conv2d_24 (Conv2D)	(None, 49, 49, 32)	16416	input_86[0][0]
max_pooling2d_24 (MaxPooling2D)	(None, 3, 3, 32)	0	conv2d_24[0][0]
flatten_29 (Flatten)	(None, 288)	0	max_pooling2d_24[0][0]
dense_31 (Dense)	(None, 64)	18496	flatten_29[0][0]
input_85 (InputLayer)	[(None, 100)]	0	
dropout_30 (Dropout)	(None, 64)	0	dense_31[0][0]
embedding_41 (Embedding)	(None, 100, 100)	1548000	input_85[0][0]
dense_32 (Dense)	(None, 100)	6500	dropout_30[0][0]
tf_op_layer_Mean_23 (TensorFlow	[(None, 100)]	0	embedding_41[0][0]
dropout_31 (Dropout)	(None, 100)	0	dense_32[0][0]
tf_op_layer_stack_8 (TensorFlow	[(None, 100, 2)]	0	tf_op_layer_Mean_23[0][0] dropout_31[0][0]
flatten_30 (Flatten)	(None, 200)	0	tf_op_layer_stack_8[0][0]
price (Dense)	(None, 3)	603	flatten_30[0][0]
type (Dense)	(None, 24)	4824	flatten_30[0][0]

Total params: 1,594,839
Trainable params: 1,594,839
Non-trainable params: 0

Advanced Tokenizer

Here I replaced the tokenizer module with SentencePiece and observed a small decrease in performance.