

# Keras: Multiple Inputs and Mixed Data

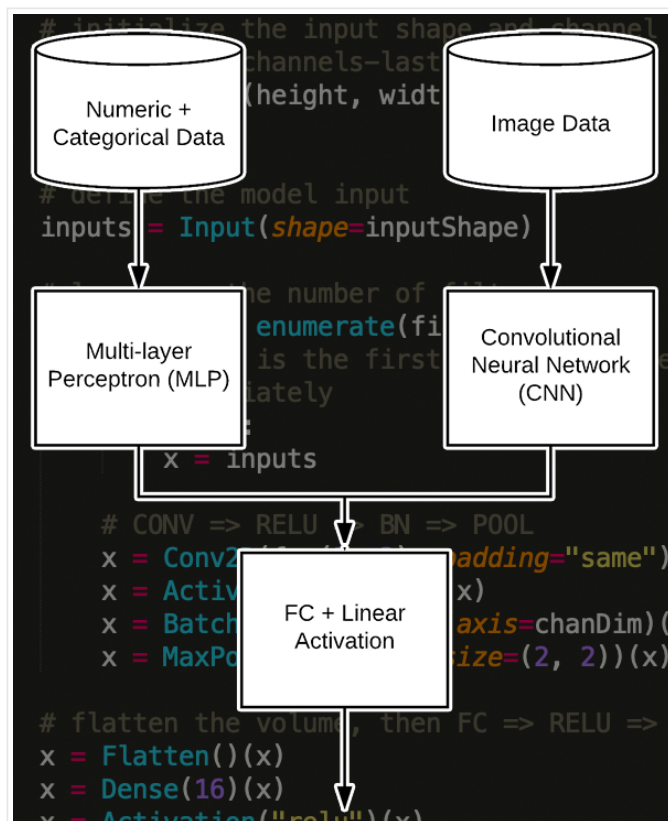
by **Adrian Rosebrock** on February 4, 2019 in **Deep Learning, Keras, Tutorials**



[Click here to download the source code to this post](#)

Like 35

G+



In this tutorial, you will learn how to use Keras for multi-input and mixed data.

You will learn how to define a Keras architecture capable of accepting multiple inputs, including numerical, categorical, and image data. We'll then train a single end-to-end network on this mixed data.

**Today is the final installment in our three part series on Keras and regression:**

1. [Basic regression with Keras](#)
2. [Training a Keras CNN for regression prediction](#)
3. Multiple inputs and mixed data with Keras (today's post)

In this series of posts, we've explored regression prediction in the context of *house price prediction*.

The house price dataset we are using includes not only **numerical and categorical data**, but **image data as well** — we call multiple types of data **mixed data** as our model needs to be capable of accepting our multiple inputs (that are not of the same type) and computing a prediction on these inputs.

In the remainder of this tutorial you will learn how to:

1. Define a Keras model capable of accepting multiple inputs, including numerical, categorical, and image data, **all at the same time**.
2. Train an end-to-end Keras model on the **mixed data inputs**.

3. Evaluate our model using the multi-inputs.

To learn more about multiple inputs and mixed data with Keras, *just keep reading!*

Looking for the source code to this post?

[Jump right to the downloads section.](#)

## Keras: Multiple Inputs and Mixed Data

In the first part of this tutorial, we will briefly review the concept of both **mixed data** and **how Keras can accept multiple inputs**.

From there we'll review our house prices dataset and the directory structure for this project.

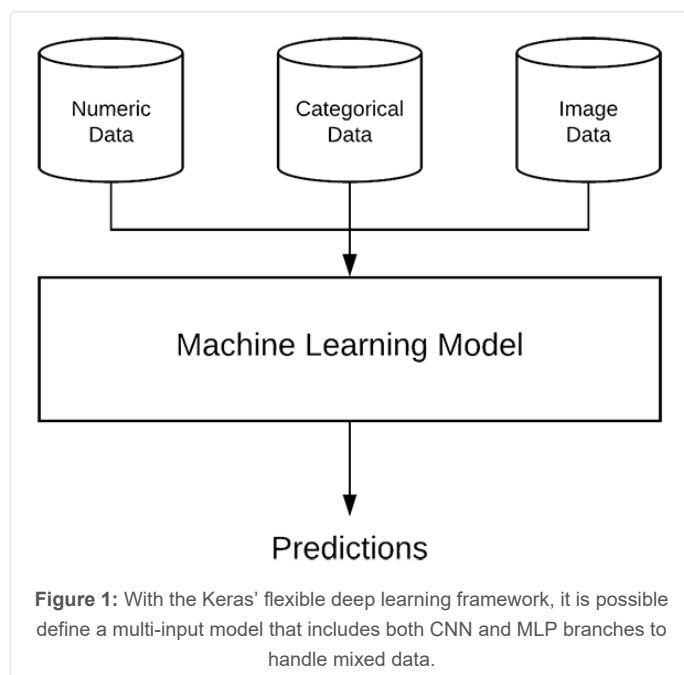
Next, I'll show you how to:

1. Load the numerical, categorical, and image data from disk.
2. Pre-process the data so we can train a network on it.
3. Prepare the mixed data so it can be applied to a multi-input Keras network.

Once our data has been prepared you'll learn **how to define and train a multi-input Keras model** that accepts multiple types of input data in a single end-to-end network.

Finally, we'll evaluate our multi-input and mixed data model on our testing set and compare the results to our previous posts in this series.

### What is mixed data?



In machine learning, mixed data refers to the concept of having multiple types of independent data.

For example, let's suppose we are machine learning engineers working at a hospital to develop a system capable of classifying the health of a patient.

We would have multiple types of input data for a given patient, including:

1. **Numeric/continuous values**, such as age, heart rate, blood pressure
2. **Categorical values**, including gender and ethnicity
3. **Image data**, such as any MRI, X-ray, etc.

All of these values constitute different data types; however, our machine learning model must be able to ingest this “mixed data” and make (accurate) predictions on it.

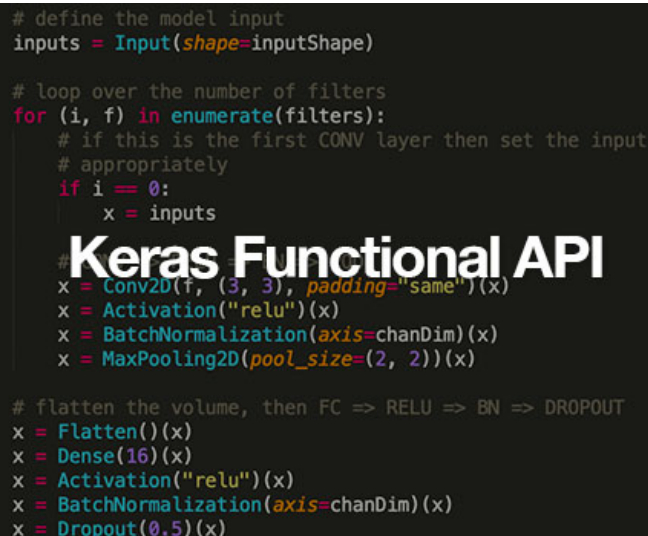
You will see the term “mixed data” in machine learning literature when working with multiple data modalities.

Developing machine learning systems capable of handling mixed data can be extremely challenging as each data type may require separate preprocessing steps, including scaling, normalization, and feature engineering.

Working with mixed data is still very much an open area of research and is often *heavily* dependent on the specific task/end goal.

We'll be working with mixed data in today's tutorial to help you get a feel for some of the challenges associated with it.

## How can Keras accept multiple inputs?



```
# define the model input
inputs = Input(shape=inputShape)

# loop over the number of filters
for (i, f) in enumerate(filters):
    # if this is the first CONV layer then set the input
    # appropriately
    if i == 0:
        x = inputs

    # Keras Functional API
    x = Conv2D(f, (3, 3), padding="same")(x)
    x = Activation("relu")(x)
    x = BatchNormalization(axis=chanDim)(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)

# flatten the volume, then FC => RELU => BN => DROPOUT
x = Flatten()(x)
x = Dense(16)(x)
x = Activation("relu")(x)
x = BatchNormalization(axis=chanDim)(x)
x = Dropout(0.5)(x)
```

**Figure 2:** As opposed to its Sequential API, Keras' functional API allows for much more complex models. In this blog post we use the functional API to support our goal of creating a model with multiple inputs and mixed data for house price prediction.

Keras is able to handle multiple inputs (and even [multiple outputs](#)) via its **functional API**.

The functional API, as opposed to the sequential API (which you almost certainly have used before via the [Sequential](#) class), can be used to define much more complex models that are non-sequential, including:

- Multi-input models
- Multi-output models
- Models that are both multiple input and multiple output
- Directed acyclic graphs
- Models with shared layers

**For example, we may define a simple sequential neural network as:**

Keras: Multiple Inputs and Mixed Data	Python
<pre>1 model = Sequential() 2 model.add(Dense(8, input_shape=(10,), activation="relu")) 3 model.add(Dense(4, activation="relu")) 4 model.add(Dense(1, activation="linear"))</pre>	

This network is a simple feedforward neural without with 10 inputs, a first hidden layer with 8 nodes, a second hidden layer with 4 nodes, and a final output layer used for regression.

**We can define the sample neural network using the functional API:**

Keras: Multiple Inputs and Mixed Data	Python
<pre>8 inputs = Input(shape=(10,)) 9 x = Dense(8, activation="relu")(inputs) 10 x = Dense(4, activation="relu")(x)</pre>	

```
11 x = Dense(1, activation="linear")(x)
12 model = Model(inputs, x)
```

Notice how we are no longer relying on the `Sequential` class.

To see the power of Keras' function API consider the following code where we create a model that accepts multiple inputs:

Keras: Multiple Inputs and Mixed Data	Python
<pre>16 # define two sets of inputs 17 inputA = Input(shape=(32,)) 18 inputB = Input(shape=(128,)) 19 20 # the first branch operates on the first input 21 x = Dense(8, activation="relu")(inputA) 22 x = Dense(4, activation="relu")(x) 23 x = Model(inputs=inputA, outputs=x) 24 25 # the second branch operates on the second input 26 y = Dense(64, activation="relu")(inputB) 27 y = Dense(32, activation="relu")(y) 28 y = Dense(4, activation="relu")(y) 29 y = Model(inputs=inputB, outputs=y) 30 31 # combine the output of the two branches 32 combined = concatenate([x.output, y.output]) 33 34 # apply a FC layer and then a regression prediction on the 35 # combined outputs 36 z = Dense(2, activation="relu")(combined) 37 z = Dense(1, activation="linear")(z) 38 39 # our model will accept the inputs of the two branches and 40 # then output a single value 41 model = Model(inputs=[x.input, y.input], outputs=z)</pre>	

Here you can see we are defining two inputs to our Keras neural network:

1. `inputA` : 32-dim
2. `inputB` : 128-dim

**Lines 21-23** define a simple `32-8-4` network using Keras' functional API.

Similarly, **Lines 26-29** define a `128-64-32-4` network.

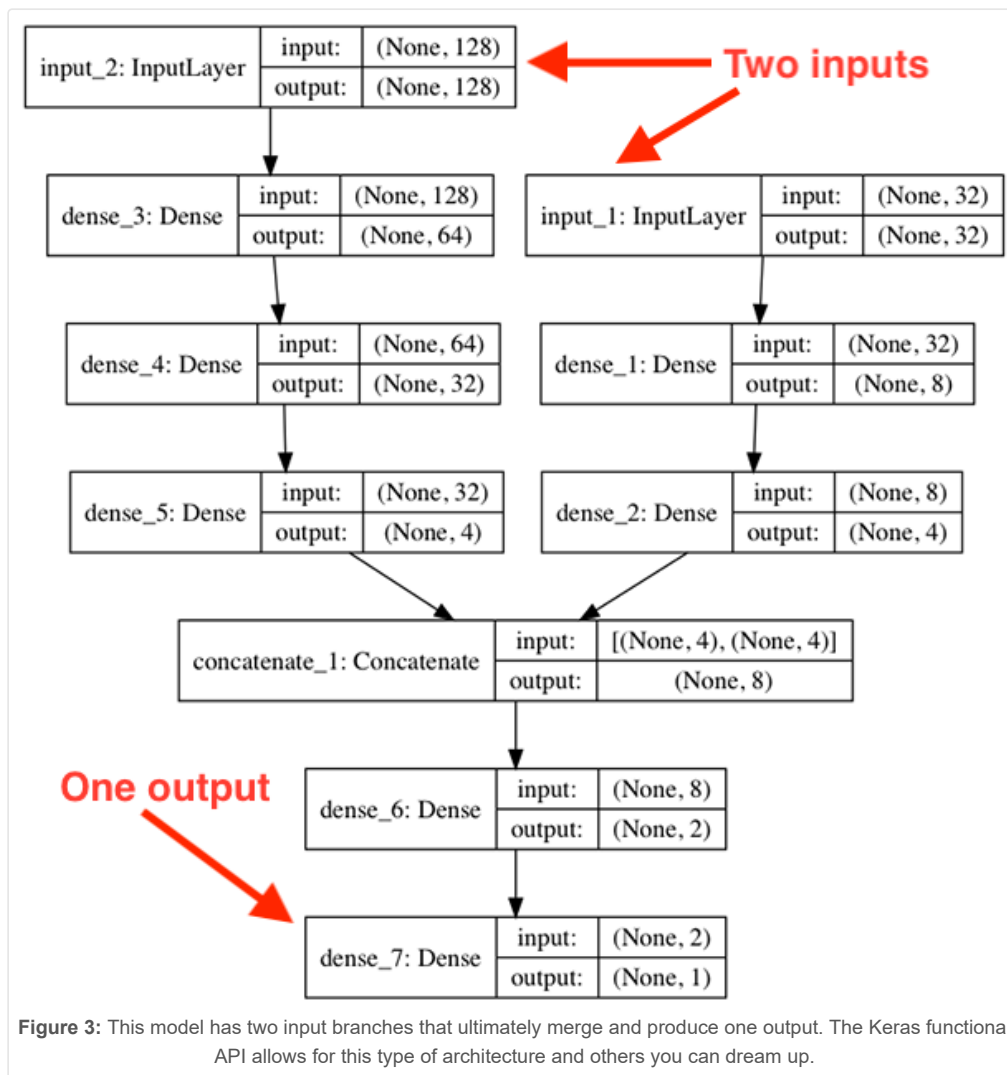
We then *combine* the outputs of both the `x` and `y` on **Line 32**. The outputs of `x` and `y` are both 4-dim so once we concatenate them we have a 8-dim vector.

We then apply two more fully-connected layers on **Lines 36 and 37**. The first layer has 2 nodes followed by a ReLU activation while the second layer has only a single node with a linear activation (i.e., our regression prediction).

The final step to building the multi-input model is to define a `Model` object which:

1. Accepts our two `inputs`
2. Defines the `outputs` as the final set of FC layers (i.e., `z`).

If you were to use Keras to visualize the model architecture it would look like the following:



Notice how our model has two distinct branches.

The first branch accepts our 128-d input while the second branch accepts the 32-d input. These branches operate independently of each other until they are concatenated. From there a single value is output from the network.

In the remainder of this tutorial, you will learn how to create multiple input networks using Keras.

## The House Prices dataset



**Figure 4:** The House Prices dataset consists of both numerical/categorical data and image data. Using Keras, we'll build a model supporting the multiple inputs and mixed data types. The result will be a Keras regression model which predicts the price/value of houses.

In this series of posts, we have been using the House Prices dataset from Ahmed and Moustafa's 2016 paper, *House price estimation from visual and textual features*.

This dataset includes both **numerical/categorical data** along with **images data** for each of the 535 example houses in the dataset.

**The numerical and categorical attributes include:**

1. Number of bedrooms
2. Number of bathrooms
3. Area (i.e., square footage)
4. Zip code

**A total of four images are provided for each house as well:**

1. Bedroom
2. Bathroom
3. Kitchen
4. Frontal view of the house

In the first post in this series, you learned [how to train a Keras regression network](#) on the numerical and categorical data.

Then, last week, you learned [how to perform regression with a Keras CNN](#).

**Today we are going to work with multiple inputs and mixed data with Keras.**

We are going to accept both the numerical/categorical data along with our image data to the network.

Two branches of a network will be defined to handle each type of data. The branches will then be combined at the end to obtain our final house price prediction.

In this manner, we will be able to leverage Keras to handle both multiple inputs and mixed data.

## Obtaining the House Prices dataset

To grab the source code for today's post, use the **"Downloads"** section. Once you have the zip file, navigate to where you downloaded it, and extract it:

```
Keras: Multiple Inputs and Mixed Data Shell
1 $ cd path/to/zip
2 $ unzip keras-multi-input.zip
3 $ cd keras-multi-input
```

And from there you can download the House Prices dataset via:

```
Keras: Multiple Inputs and Mixed Data Shell
1 $ git clone https://github.com/emanhamed/Houses-dataset
```

The House Prices dataset should now be in the `keras-multi-input` directory which is the directory we are using for this project.

## Project structure

Let's take a look at how today's project is organized:

```
Keras: Multiple Inputs and Mixed Data Shell
1 $ tree --dirsfirst --filelimit 10
2 .
3 |— Houses-dataset
4 |   |— Houses\ Dataset [2141 entries]
5 |   |— README.md
```



```

6 | pyimagesearch
7 |   |__init__.py
8 |   |datasets.py
9 |   |models.py
10 | mixed_training.py
11
12 3 directories, 5 files

```

The Houses-dataset folder contains our House Prices dataset that we're working with for this series. When we're ready to run the `mixed_training.py` script, you'll just need to provide a path as a command line argument to the dataset (I'll show you exactly how this is done in the results section).

Today we'll be reviewing three Python scripts:

- `pyimagesearch/datasets.py` : Handles loading and preprocessing our numerical/categorical data as well as our image data. We previously reviewed this script over the past two weeks, but I'll be walking you through it again today.
- `pyimagesearch/models.py` : Contains our Multi-layer Perceptron (MLP) and Convolutional Neural Network (CNN). These components are the input branches to our multi-input, mixed data model. We reviewed this script last week and we'll briefly review it today as well.
- `mixed_training.py` : Our training script will use the `pyimagesearch` module convenience functions to load + split the data and concatenate the two branches to our network + add the head. It will then train and evaluate the model.

## Loading the numerical and categorical data

```

>>> import pandas as pd
>>> cols = ["bedrooms", "bathrooms", "area", "zipcode", "price"]
>>> inputPath = "HousesInfo.txt"
>>> df = pd.read_csv(inputPath, sep=" ", header=None, names=cols)
>>> df.head()
   bedrooms  bathrooms  area  zipcode  price
0         4         4.0  4053   85255  869500.0
1         4         3.0  3343   36372  865200.0
2         3         4.0  3923   85266  889000.0
3         5         5.0  4022   85262  910000.0
4         3         4.0  4116   85266  971226.0
>>>

```

Figure 5: We use pandas, a Python package, to read CSV housing data.

We covered how to load the numerical and categorical data for the house prices dataset in our [Keras regression post](#) but as a matter of completeness, we will review the code (in less detail) here today.

Be sure to refer to the [previous post](#) if you want a detailed walkthrough of the code.

Open up the `datasets.py` file and insert the following code:

```

Keras: Multiple Inputs and Mixed Data
Python
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.preprocessing import MinMaxScaler
4  import pandas as pd
5  import numpy as np
6  import glob
7  import cv2
8  import os
9
10 def load_house_attributes(inputPath):
11     # initialize the list of column names in the CSV file and then
12     # load it using Pandas
13     cols = ["bedrooms", "bathrooms", "area", "zipcode", "price"]
14     df = pd.read_csv(inputPath, sep=" ", header=None, names=cols)
15
16     # determine (1) the unique zip codes and (2) the number of data
17     # points with each zip code
18     zipcodes = df["zipcode"].value_counts().keys().tolist()
19     counts = df["zipcode"].value_counts().tolist()
20
21     # loop over each of the unique zip codes and their corresponding
22     # count
23     for (zipcode, count) in zip(zipcodes, counts):
24         # the zip code counts for our housing dataset is *extremely*
25         # unbalanced (some only having 1 or 2 houses per zip code)
26         # so let's sanitize our data by removing any houses with less

```

```

27     # than 25 houses per zip code
28     if count < 25:
29         idxs = df[df["zipcode"] == zipcode].index
30         df.drop(idxs, inplace=True)
31
32     # return the data frame
33     return df

```

Our imports are handled on **Lines 2-8**.

From there we define the `load_house_attributes` function on **Lines 10-33**. This function reads the numerical/categorical data from the House Prices dataset in the form of a CSV file via Pandas' `pd.read_csv` on **Lines 13 and 14**.

The data is filtered to accommodate an imbalance. Some zipcodes only are represented by 1 or 2 houses, therefore we just go ahead and `drop` (**Lines 23-30**) any records where there are fewer than `25` houses from the zipcode. The result is a more accurate model later on.

Now let's define the `process_house_attributes` function:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 35 def process_house_attributes(df, train, test): 36     # initialize the column names of the continuous data 37     continuous = ["bedrooms", "bathrooms", "area"] 38 39     # perform min-max scaling each continuous feature column to 40     # the range [0, 1] 41     cs = MinMaxScaler() 42     trainContinuous = cs.fit_transform(train[continuous]) 43     testContinuous = cs.transform(test[continuous]) 44 45     # one-hot encode the zip code categorical data (by definition of 46     # one-hot encoding, all output features are now in the range [0, 1]) 47     zipBinarizer = LabelBinarizer().fit(df["zipcode"]) 48     trainCategorical = zipBinarizer.transform(train["zipcode"]) 49     testCategorical = zipBinarizer.transform(test["zipcode"]) 50 51     # construct our training and testing data points by concatenating 52     # the categorical features with the continuous features 53     trainX = np.hstack([trainCategorical, trainContinuous]) 54     testX = np.hstack([testCategorical, testContinuous]) 55 56     # return the concatenated training and testing data 57     return (trainX, testX) </pre>	

This function applies **min-max scaling** to the *continuous features* via scikit-learn's `MinMaxScaler` (**Lines 41-43**).

Then, **one-hot encoding** for the *categorical features* is computed, this time via scikit-learn's `LabelBinarizer` (**Lines 47-49**).

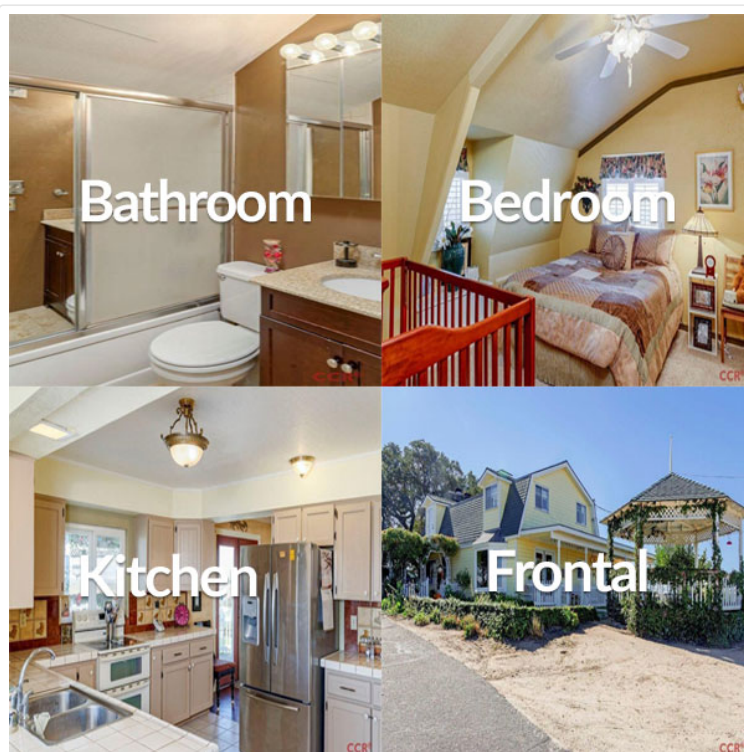
The *continuous* and *categorical* features are then **concatenated** and returned (**Lines 53-57**).

Be sure to refer to the previous posts in this series for more details on the two functions we reviewed in this section:

1. [Regression with Keras](#)
2. [Keras, Regression, and CNNs](#)

## Loading the image dataset





**Figure 6:** One branch of our model accepts a single image — a montage of four images from the home. Using the montage combined with the numerical/categorical data input to another branch, our model then uses regression to predict the value of the home with the Keras framework.

The next step is to define a helper function to load our input images. Again, open up the `datasets.py` file and insert the following code:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 59 def load_house_images(df, inputPath): 60     # initialize our images array (i.e., the house images themselves) 61     images = [] 62 63     # loop over the indexes of the houses 64     for i in df.index.values: 65         # find the four images for the house and sort the file paths, 66         # ensuring the four are always in the *same order* 67         basePath = os.path.sep.join([inputPath, "{}.*".format(i + 1)]) 68         housePaths = sorted(list(glob.glob(basePath))) </pre>	

The `load_house_images` function has three goals:

1. Load all photos from the House Prices dataset. Recall that we have **four photos** per house (**Figure 6**).
2. Generate a **single montage image** from the four photos. The montage will always be arranged as you see in the figure.
3. Append all of these home montages to a list/array and return to the calling function.

Beginning on **Line 59**, we define the function which accepts a Pandas dataframe and dataset `inputPath`.

From there, we proceed to:

- Initialize the `images` list (**Line 61**). We'll be populating this list with all of the montage images that we build.
- Loop over houses in our data frame (**Line 64**). Inside the loop, we:
  - Grab the paths to the four photos for the current house (**Lines 67 and 68**).

Let's keep making progress in the loop:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 70     # initialize our list of input images along with the output image 71     # after *combining* the four input images 72     inputImages = [] 73     outputImage = np.zeros((64, 64, 3), dtype="uint8") 74 75     # loop over the input house paths </pre>	

```

76     for housePath in housePaths:
77         # load the input image, resize it to be 32 32, and then
78         # update the list of input images
79         image = cv2.imread(housePath)
80         image = cv2.resize(image, (32, 32))
81         inputImages.append(image)
82
83     # tile the four input images in the output image such the first
84     # image goes in the top-right corner, the second image in the
85     # top-left corner, the third image in the bottom-right corner,
86     # and the final image in the bottom-left corner
87     outputImage[0:32, 0:32] = inputImages[0]
88     outputImage[0:32, 32:64] = inputImages[1]
89     outputImage[32:64, 32:64] = inputImages[2]
90     outputImage[32:64, 0:32] = inputImages[3]
91
92     # add the tiled image to our set of images the network will be
93     # trained on
94     images.append(outputImage)
95
96 # return our set of images
97 return np.array(images)

```

The code so far has accomplished the first goal discussed above (grabbing the four house images per house). Let's wrap up the `load_house_images` function:

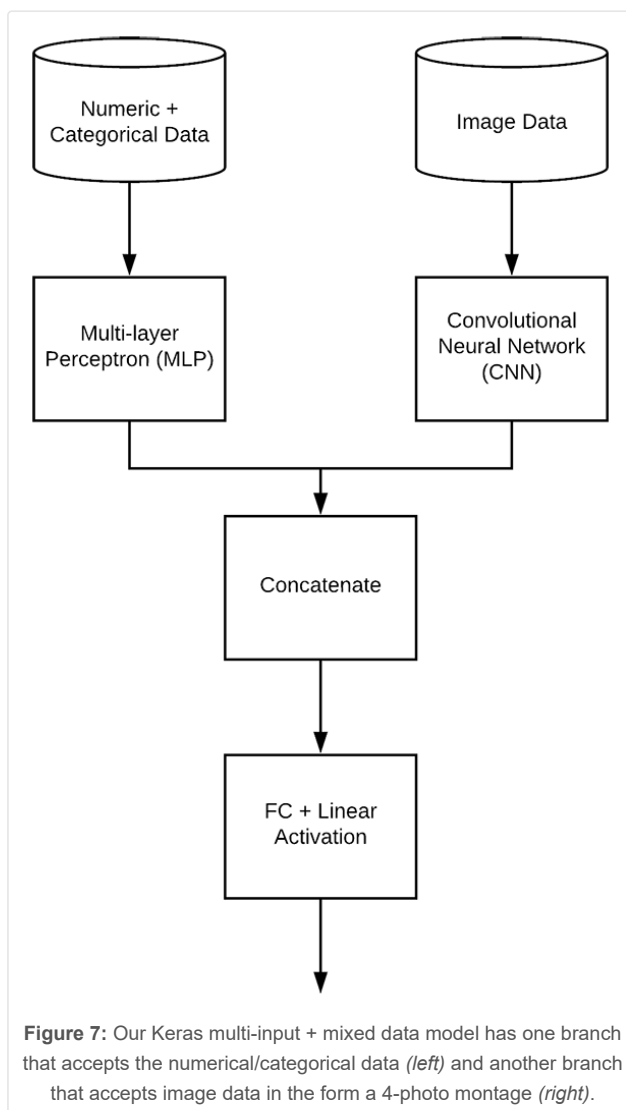
- Still inside the loop, we:
  - Perform initializations (**Lines 72 and 73**). Our `inputImages` will be in list form containing four photos of each record. Our `outputImage` will be the montage of the photos (like **Figure 6**).
  - Loop over 4 photos (**Line 76**):
    - Load, resize, and append each photo to `inputImages` (**Lines 79-81**).
  - Create the tiling (a montage) for the four house images (**Lines 87-90**) with:
    - The bathroom image in the *top-left*.
    - The bedroom image in the *top-right*.
    - The frontal view in the *bottom-right*.
    - The kitchen in the *bottom-left*.
  - Append the tiling/montage `outputImage` to `images` (**Line 94**).
- Jumping out of the loop, we `return` all the `images` in the form of a NumPy array (**Line 57**).

We'll have as many `images` as there are records we're training with (remember, we dropped a few of them in the `process_house_attributes` function).

Each of our tiled `images` will look like **Figure 6** (without the overlaid text of course). You can see the four photos therein have been arranged in a montage (I've used larger image dimensions so we can better visualize what the code is doing). Just as our numerical and categorical attributes represent the house, these four photos (tiled into a single image) will represent the visual aesthetics of the house.

If you need to review this process in further detail, be sure to refer to [last week's post](#).

## Defining our Multi-layer Perceptron (MLP) and Convolutional Neural Network (CNN)



As you've gathered thus far, we've had to massage our data carefully using multiple libraries: Pandas, scikit-learn, OpenCV, and NumPy.

We've organized and pre-processed the two modalities of our dataset at this point via [datasets.py](#) :

- Numeric and categorical data
- Image data

The skills we've used in order to accomplish this have been developed through experience + practice, machine learning best practices, and behind the scenes of this blog post, a little bit of debugging. Please don't overlook what we've discussed so far using our data massaging skills as it is key to the rest of our project's success.

Let's shift gears and discuss our multi-input and mixed data network that we'll build with Keras' functional API.

**In order to build our multi-input network we will need two branches:**

- The first branch will be a **simple Multi-layer Perceptron (MLP)** designed to handle the **categorical/numerical inputs**.
- The second branch will be a **Convolutional Neural Network** to operate over the **image data**.
- These **branches** will then be **concatenated** together to form the final **multi-input Keras model**.

We'll handle building the final concatenated multi-input model in the next section — our current task is to define the two branches.

Open up the [models.py](#) file and insert the following code:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 1 # import the necessary packages 2 from keras.models import Sequential 3 from keras.layers.normalization import BatchNormalization </pre>	

```

4 from keras.layers.convolutional import Conv2D
5 from keras.layers.convolutional import MaxPooling2D
6 from keras.layers.core import Activation
7 from keras.layers.core import Dropout
8 from keras.layers.core import Dense
9 from keras.layers import Flatten
10 from keras.layers import Input
11 from keras.models import Model
12
13 def create_mlp(dim, regress=False):
14     # define our MLP network
15     model = Sequential()
16     model.add(Dense(8, input_dim=dim, activation="relu"))
17     model.add(Dense(4, activation="relu"))
18
19     # check to see if the regression node should be added
20     if regress:
21         model.add(Dense(1, activation="linear"))
22
23     # return our model
24     return model

```

Lines 2-11 handle our Keras imports. You'll see each of the imported functions/classes going forward in this script.

Our **categorical/numerical data** will be processed by a simple Multi-layer Perceptron (MLP).

The MLP is defined by `create_mlp` on Lines 13-24.

Discussed in detail in the [first post in this series](#), the MLP relies on the Keras `Sequential` API. Our MLP is quite simple having:

- A fully connected ( `Dense` ) input layer with ReLU `activation` (Line 16).
- A fully-connected hidden layer, also with ReLU `activation` (Line 17).
- And finally, an **optional** regression output with linear activation (Lines 20 and 21).

While we used the regression output of the MLP in the first post, it **will not be used** in this multi-input, mixed data network. As you'll soon see, we'll be setting `regress=False` explicitly even though it is the default as well. **Regression will actually be performed later on the head of the *entire* multi-input, mixed data network** (the bottom of Figure 7).

The MLP *branch* is returned on Line 24.

Referring back to Figure 7, we've now built the *top-left* branch of our network.

Let's now define the *top-right* branch of our network, a CNN:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 26 def create_cnn(width, height, depth, filters=(16, 32, 64), regress=False): 27     # initialize the input shape and channel dimension, assuming 28     # TensorFlow/channels-last ordering 29     inputShape = (height, width, depth) 30     chanDim = -1 31 32     # define the model input 33     inputs = Input(shape=inputShape) 34 35     # loop over the number of filters 36     for (i, f) in enumerate(filters): 37         # if this is the first CONV layer then set the input 38         # appropriately 39         if i == 0: 40             x = inputs 41 42         # CONV =&gt; RELU =&gt; BN =&gt; POOL 43         x = Conv2D(f, (3, 3), padding="same")(x) 44         x = Activation("relu")(x) 45         x = BatchNormalization(axis=chanDim)(x) 46         x = MaxPooling2D(pool_size=(2, 2))(x) </pre>	

The `create_cnn` function handles the **image data** and accepts five parameters:

- `width` : The width of the input images in pixels.
- `height` : How many pixels tall the input images are.
- `depth` : The number of channels in our input images. For RGB color images, it is three.

- `filters` : A tuple of progressively larger filters so that our network can learn more discriminate features.
- `regress` : A boolean indicating whether or not a fully-connected linear activation layer will be appended to the CNN for regression purposes.

The `inputShape` of our network is defined on **Line 29**. It assumes “channels last” ordering for the TensorFlow backend.

The `Input` to the model is defined via the `inputShape` on (**Line 33**).

From there we begin looping over the filters and create a set of `CONV => RELU > BN => POOL` layers. Each iteration of the loop appends these layers. Be sure to check out Chapter 11 from the *Starter Bundle* of *Deep Learning for Computer Vision with Python* for more information on these layer types if you are unfamiliar.

Let's finish building the CNN branch of our network:

Keras: Multiple Inputs and Mixed Data	Python
48	<code># flatten the volume, then FC =&gt; RELU =&gt; BN =&gt; DROPOUT</code>
49	<code>x = Flatten()(x)</code>
50	<code>x = Dense(16)(x)</code>
51	<code>x = Activation("relu")(x)</code>
52	<code>x = BatchNormalization(axis=chanDim)(x)</code>
53	<code>x = Dropout(0.5)(x)</code>
54	
55	<code># apply another FC layer, this one to match the number of nodes</code>
56	<code># coming out of the MLP</code>
57	<code>x = Dense(4)(x)</code>
58	<code>x = Activation("relu")(x)</code>
59	
60	<code># check to see if the regression node should be added</code>
61	<code>if regress:</code>
62	<code>    x = Dense(1, activation="linear")(x)</code>
63	
64	<code># construct the CNN</code>
65	<code>model = Model(inputs, x)</code>
66	
67	<code># return the CNN</code>
68	<code>return model</code>

We `Flatten` the next layer (**Line 49**) and then add a fully-connected layer with `BatchNormalization` and `Dropout` (**Lines 50-53**).

Another fully-connected layer is applied to match the four nodes coming out of the multi-layer perceptron (**Lines 57 and 58**). Matching the number of nodes is not a requirement but it does help balance the branches.

On **Lines 61 and 62**, a check is made to see if the regression node should be appended; it is then added in accordingly. **Again, we will not be conducting regression at the end of this branch either.** Regression will be performed on the head of the multi-input, mixed data network (the very bottom of **Figure 7**).

Finally, the model is constructed from our `inputs` and all the layers we've assembled together, `x` (**Line 65**).

We can then `return` the CNN **branch** to the calling function (**Line 68**).

Now that we've defined **both branches of the multi-input Keras model**, let's learn how we can combine them!

## Multiple inputs with Keras

We are now ready to build our final Keras model capable of handling both multiple inputs and mixed data. This is where the **branches come together** and ultimately where the “magic” happens. Training will also happen in this script.

Create a new file named `mixed_training.py`, open it up, and insert the following code:

Keras: Multiple Inputs and Mixed Data	Python
1	<code># import the necessary packages</code>
2	<code>from pyimagesearch import datasets</code>
3	<code>from pyimagesearch import models</code>
4	<code>from sklearn.model_selection import train_test_split</code>
5	<code>from keras.layers.core import Dense</code>
6	<code>from keras.models import Model</code>
7	<code>from keras.optimizers import Adam</code>
8	<code>from keras.layers import concatenate</code>
9	<code>import numpy as np</code>

```

10 import argparse
11 import locale
12 import os
13
14 # construct the argument parser and parse the arguments
15 ap = argparse.ArgumentParser()
16 ap.add_argument("-d", "--dataset", type=str, required=True,
17     help="path to input dataset of house images")
18 args = vars(ap.parse_args())

```

Our imports and command line arguments are handled first.

Notable imports include:

- `datasets` : Our three convenience functions for loading/processing the CSV data and loading/pre-processing the house photos from the Houses Dataset.
- `models` : Our MLP and CNN input branches which will serve as our multi-input, mixed data.
- `train_test_split` : A scikit-learn function to construct our training/testing data splits.
- `concatenate` : A special Keras function which will accept multiple inputs.
- `argparse` : Handles parsing command line arguments.

We have one command line argument to parse on **Lines 15-18**, `--dataset`, which is the path to where you downloaded the House Prices dataset.

Let's load our numerical/categorical data and image data:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 20 # construct the path to the input .txt file that contains information 21 # on each house in the dataset and then load the dataset 22 print("[INFO] loading house attributes...") 23 inputPath = os.path.sep.join([args["dataset"], "HousesInfo.txt"]) 24 df = datasets.load_house_attributes(inputPath) 25 26 # load the house images and then scale the pixel intensities to the 27 # range [0, 1] 28 print("[INFO] loading house images...") 29 images = datasets.load_house_images(df, args["dataset"]) 30 images = images / 255.0 </pre>	

Here we've loaded the House Prices dataset as a Pandas dataframe (**Lines 23 and 24**).

Then we've loaded our `images` and scaled them to the range `[0, 1]` (**Lines 29-30**).

Be sure to review the `load_house_attributes` and `load_house_images` functions above if you need a reminder on what these functions are doing under the hood.

Now that our data is loaded, we're going to construct our training/testing splits, scale the prices, and process the house attributes:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 32 # partition the data into training and testing splits using 75% of 33 # the data for training and the remaining 25% for testing 34 print("[INFO] processing data...") 35 split = train_test_split(df, images, test_size=0.25, random_state=42) 36 (trainAttrX, testAttrX, trainImagesX, testImagesX) = split 37 38 # find the largest house price in the training set and use it to 39 # scale our house prices to the range [0, 1] (will lead to better 40 # training and convergence) 41 maxPrice = trainAttrX["price"].max() 42 trainY = trainAttrX["price"] / maxPrice 43 testY = testAttrX["price"] / maxPrice 44 45 # process the house attributes data by performing min-max scaling 46 # on continuous features, one-hot encoding on categorical features, 47 # and then finally concatenating them together 48 (trainAttrX, testAttrX) = datasets.process_house_attributes(df, 49     trainAttrX, testAttrX) </pre>	

Our training and testing splits are constructed on **Lines 35 and 36**. We've allocated 75% of our data for training and 25% of our data for testing.



From there, we find the `maxPrice` from the training set (**Line 41**) and scale the training and testing data accordingly (**Lines 42 and 43**). Having the pricing data in the range `[0, 1]` leads to better training and convergence.

Finally, we go ahead and process our house attributes by performing min-max scaling on continuous features and one-hot encoding on categorical features. The `process_house_attributes` function handles these actions and concatenates the continuous and categorical features together, returning the results (**Lines 48 and 49**).

### Ready for some magic?

Okay, I lied. There isn't actually any "magic" going on in this next code block! But we will `concatenate` the branches of our network and finish our multi-input Keras network:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 51 # create the MLP and CNN models 52 mlp = models.create_mlp(trainAttrX.shape[1], regress=False) 53 cnn = models.create_cnn(64, 64, 3, regress=False) 54 55 # create the input to our final set of layers as the *output* of both 56 # the MLP and CNN 57 combinedInput = concatenate([mlp.output, cnn.output]) 58 59 # our final FC layer head will have two dense layers, the final one 60 # being our regression head 61 x = Dense(4, activation="relu")(combinedInput) 62 x = Dense(1, activation="linear")(x) 63 64 # our final model will accept categorical/numerical data on the MLP 65 # input and images on the CNN input, outputting a single value (the 66 # predicted price of the house) 67 model = Model(inputs=[mlp.input, cnn.input], outputs=x) </pre>	

Handling multiple inputs with Keras is quite easy when you've organized your code and models.

On **Lines 52 and 53**, we create our `mlp` and `cnn` models. Notice that `regress=False` — our regression head comes later on **Line 62**.

We'll then `concatenate` the `mlp.output` and `cnn.output` as shown on **Line 57**. I'm calling this our `combinedInput` because it is the input to the rest of the network (from **Figure 3** this is `concatenate_1` where the two branches come together).

The `combinedInput` to the final layers in the network is based on the output of both the MLP and CNN branches' `8-4-1` FC layers (since each of the 2 branches outputs a 4-dim FC layer and then we concatenate them to create an 8-dim vector).

We tack on a fully connected layer with four neurons to the `combinedInput` (**Line 61**).

Then we add our `"linear"` `activation` regression head (**Line 62**), *the output of which is the predicted price*.

Our `Model` is defined using the `inputs` of both branches as our multi-input and the final set of layers `x` as the `output` (**Line 67**).

Let's go ahead and compile, train, and evaluate our newly formed `model` :

Keras: Multiple Inputs and Mixed Data	Python
<pre> 69 # compile the model using mean absolute percentage error as our loss, 70 # implying that we seek to minimize the absolute percentage difference 71 # between our price *predictions* and the *actual prices* 72 opt = Adam(lr=1e-3, decay=1e-3 / 200) 73 model.compile(loss="mean_absolute_percentage_error", optimizer=opt) 74 75 # train the model 76 print("[INFO] training model...") 77 model.fit( 78     [trainAttrX, trainImagesX], trainY, 79     validation_data=([testAttrX, testImagesX], testY), 80     epochs=200, batch_size=8) 81 82 # make predictions on the testing data 83 print("[INFO] predicting house prices...") 84 preds = model.predict([testAttrX, testImagesX]) </pre>	

Our `model` is compiled with `"mean_absolute_percentage_error"` `loss` and an `Adam` optimizer with learning rate `decay` (Lines 72 and 73).

Training is kicked off on **Lines 77-80**. This is known as fitting the model (and is also where all the weights are tuned by the process known as backpropagation).

Calling `model.predict` on our testing data (**Line 84**) allows us to grab predictions for evaluating our model. Let's perform evaluation now:

Keras: Multiple Inputs and Mixed Data	Python
<pre> 86 # compute the difference between the *predicted* house prices and the 87 # *actual* house prices, then compute the percentage difference and 88 # the absolute percentage difference 89 diff = preds.flatten() - testY 90 percentDiff = (diff / testY) * 100 91 absPercentDiff = np.abs(percentDiff) 92 93 # compute the mean and standard deviation of the absolute percentage 94 # difference 95 mean = np.mean(absPercentDiff) 96 std = np.std(absPercentDiff) 97 98 # finally, show some statistics on our model 99 locale.setlocale(locale.LC_ALL, "en_US.UTF-8") 100 print("[INFO] avg. house price: {}, std house price: {}".format( 101     locale.currency(df["price"].mean(), grouping=True), 102     locale.currency(df["price"].std(), grouping=True))) 103 print("[INFO] mean: {:.2f}%, std: {:.2f}%".format(mean, std)) </pre>	

To evaluate our model, we have computed absolute percentage difference (**Lines 89-91**) and used it to derive our final metrics (**Lines 95 and 96**).

These metrics (price mean, price standard deviation, and mean + standard deviation of the absolute percentage difference) are printed to the terminal with proper currency locale formatting (**Lines 100-103**).

## Multi-input and mixed data results



**Figure 8:** Real estate price prediction is a difficult task, but our Keras multi-input + mixed input regression model yields relatively good results on our limited House Prices dataset.

Finally, we are ready to train our multi-input network on our mixed data!

Make sure you have:

1. Configured your dev environment according to the [first tutorial in this series](#).
2. Used the **“Downloads”** section of this tutorial to download the source code.
3. Downloaded the house prices dataset using the instructions in the **“Obtaining the House Prices dataset”** section above.

From there, open up a terminal and execute the following command to kick off training the network:

Keras: Multiple Inputs and Mixed Data	Shell
<pre> 1 \$ python mixed_training.py --dataset Houses-dataset/Houses\ Dataset/ 2 [INFO] training model... 3 Train on 271 samples, validate on 91 samples 4 Epoch 1/200 5 271/271 [=====] - 2s 8ms/step - loss: 240.2516 - val_loss: 118.1782 6 Epoch 2/200 7 271/271 [=====] - 1s 5ms/step - loss: 195.8325 - val_loss: 95.3750 8 Epoch 3/200 9 271/271 [=====] - 1s 5ms/step - loss: 121.5940 - val_loss: 85.1037 10 Epoch 4/200 11 271/271 [=====] - 1s 5ms/step - loss: 103.2910 - val_loss: 72.1434 </pre>	

```

12 Epoch 5/200
13 271/271 [=====] - 1s 5ms/step - loss: 82.3916 - val_loss: 61.9368
14 Epoch 6/200
15 271/271 [=====] - 1s 5ms/step - loss: 81.3794 - val_loss: 59.7905
16 Epoch 7/200
17 271/271 [=====] - 1s 5ms/step - loss: 71.3617 - val_loss: 58.8067
18 Epoch 8/200
19 271/271 [=====] - 1s 5ms/step - loss: 72.7032 - val_loss: 56.4613
20 Epoch 9/200
21 271/271 [=====] - 1s 5ms/step - loss: 52.0019 - val_loss: 54.7461
22 Epoch 10/200
23 271/271 [=====] - 1s 5ms/step - loss: 62.4559 - val_loss: 49.1401
24 ...
25 Epoch 190/200
26 271/271 [=====] - 1s 5ms/step - loss: 16.0892 - val_loss: 22.8415
27 Epoch 191/200
28 271/271 [=====] - 1s 5ms/step - loss: 16.1908 - val_loss: 22.5139
29 Epoch 192/200
30 271/271 [=====] - 1s 5ms/step - loss: 16.9099 - val_loss: 22.5922
31 Epoch 193/200
32 271/271 [=====] - 1s 5ms/step - loss: 18.6216 - val_loss: 26.9679
33 Epoch 194/200
34 271/271 [=====] - 1s 5ms/step - loss: 16.5341 - val_loss: 23.1445
35 Epoch 195/200
36 271/271 [=====] - 1s 5ms/step - loss: 16.4120 - val_loss: 26.1224
37 Epoch 196/200
38 271/271 [=====] - 1s 5ms/step - loss: 16.4939 - val_loss: 23.1224
39 Epoch 197/200
40 271/271 [=====] - 1s 5ms/step - loss: 15.6253 - val_loss: 22.2930
41 Epoch 198/200
42 271/271 [=====] - 1s 5ms/step - loss: 16.0514 - val_loss: 23.6948
43 Epoch 199/200
44 271/271 [=====] - 1s 5ms/step - loss: 17.9525 - val_loss: 22.9743
45 Epoch 200/200
46 271/271 [=====] - 1s 5ms/step - loss: 16.0377 - val_loss: 22.4130
47 [INFO] predicting house prices...
48 [INFO] avg. house price: $533,388.27, std house price: $493,403.08
49 [INFO] mean: 22.41%, std: 20.11%

```

Our mean absolute percentage error starts off very high but continues to fall throughout the training process.

By the end of training, we are obtaining of **22.41%** mean absolute percentage error on our testing set, implying that, on average, our network will be ~22% off in its house price predictions.

**Let's compare this result to our previous two posts in the series:**

1. Using *just* an MLP on the numerical/categorical data: **26.01%**
2. Using *just* a CNN on the image data: **56.91%**

**As you can see, working with mixed data by:**

1. Combining our numerical/categorical data along with image data
2. And training a multi-input model on the mixed data...

**...has led to a better performing model!**

## Summary

In this tutorial, you learned how to define a Keras network capable of accepting multiple inputs.

You learned how to work with mixed data using Keras as well.

To accomplish these goals we defined a multiple input neural network capable of accepting:

- Numerical data
- Categorical data
- Image data

The numerical data was min-max scaled to the range  $[0, 1]$  prior to training. Our categorical data was one-hot encoded (also ensuring the resulting integer vectors were in the range  $[0, 1]$ ).

The numerical and categorical data were then concatenated into a single feature vector to form the first input to the Keras network.

Our image data was also scaled to the range  $[0, 1]$  — this data served as the second input to the Keras network.

One branch of the model included strictly fully-connected layers (for the concatenated numerical and categorical data) while the second branch of the multi-input model was essentially a small Convolutional Neural Network.

The outputs of both branches were combined and a single output (the regression prediction) was defined.

**In this manner, we were able to train our multiple input network end-to-end, resulting in *better accuracy* than using just one of the inputs alone.**

I hope you enjoyed today's blog post — if you ever need to work with multiple inputs and mixed data in your own projects definitely consider using the code covered in this tutorial as a template.

From there you can modify the code to your own needs.

**To download the source code, and be notified when future tutorials are published here on PyImageSearch, *just enter your email address in the form below!***

🔖 **cnn, convolutional neural network, deep learning, keras, multi-input, neural nets, regression**