Project title:

Date:

What is LeNet-5?

- LeNet-5 is a convolutional neural network that was introduced by Yann LeCun et al. in their 1998 paper, Gradient-Based Learning Applied to Document Recognition.

- The LeNet-5 network architecture consists of the following layers:

- Input layer: The input layer consists of a 28x28 grayscale image.
- Convolutional layer: The first convolutional layer has 6 feature maps, each with a 5x5 kernel. The layer also has a ReLU activation function.
- Pooling layer: The first pooling layer performs max pooling with a 2x2 kernel and stride of 2.
- Convolutional layer: The second convolutional layer has 16 feature maps, each with a 5x5 kernel. The layer also has a ReLU activation function.
- Pooling layer: The second pooling layer performs max pooling with a 2x2 kernel and stride of 2.
- Fully connected layer: The first fully connected layer has 120 units and a ReLU activation function.
- Output layer: The output layer has 10 units and a softmax activation function, corresponding to the 10 classes in the MNIST dataset.

The LeNet-5 model is trained using stochastic gradient descent with backpropagation. The model is trained to minimize the cross-entropy loss function between the predicted class probabilities and the true class labels.

Code:

```
from tensorflow.keras import Model

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Create the input layer

input_layer = Input(shape=(28, 28, 1))

# First convolutional layer with max pooling

x = Conv2D(6, kernel_size=(5, 5), activation='relu')(input_layer)

x = MaxPooling2D(pool_size=(2, 2))(x)

# Second convolutional layer with max pooling

x = Conv2D(16, kernel_size=(5, 5), activation='relu')(x)

x = MaxPooling2D(pool_size=(2, 2))(x)

# Flatten the feature maps and add a fully connected layer

x = Flatten()(x)

x = Dense(120, activation='relu')(x)
```

```
# Add another fully connected layer

x = Dense(84, activation='relu')(x)

# Add a softmax output layer

output_layer = Dense(10, activation='softmax')(x)

# Create the model

model = Model(input_layer, output_layer)

# Compile the model

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(x_train, y_train, batch_size=128, epochs=10, validation_data=(x_val, y_val))
```

This code creates a LeNet-5 model with a single input layer and three convolutional/max pooling layers, followed by two fully connected layers and a softmax output layer. It then compiles the model using the Adam optimizer and categorical cross-entropy loss, and trains it using the provided training data.

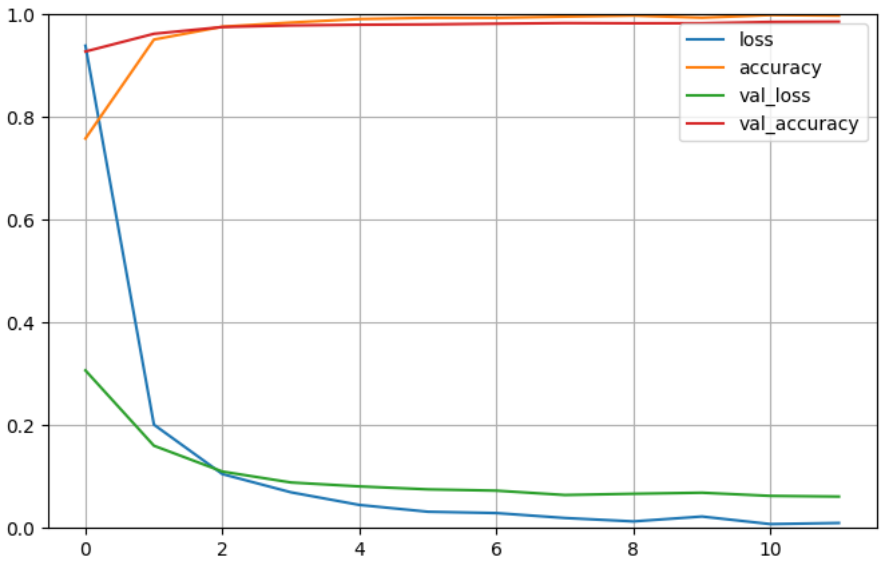
Summary for layers:

```
1  model.summary()

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         456

 average_pooling2d (AverageP  (None, 14, 14, 6)        0
 ooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 conv2d_2 (Conv2D)           (None, 1, 1, 120)         48120

 flatten (Flatten)           (None, 120)               0

 dense (Dense)               (None, 84)                10164

 dense_1 (Dense)             (None, 43)                3655

=================================================================
Total params: 64,811
Trainable params: 64,811
Non-trainable params: 0
_________________________________________________________________
```

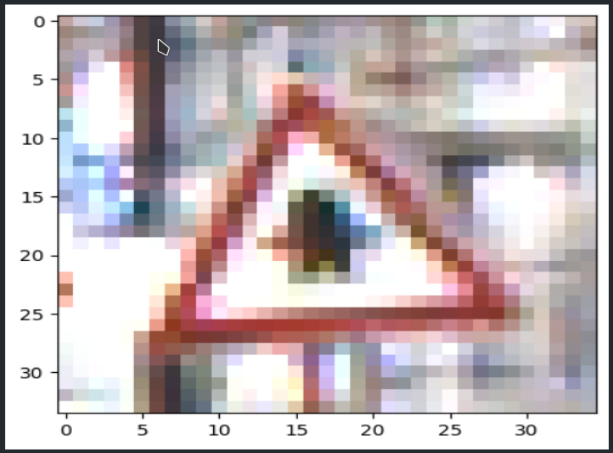
Classes that are used as an output in the model:

```
#dictionary to label all traffic signs class.
classes = { 0:'Speed limit (20km/h)',
        1:'Speed limit (30km/h)',
        2:'Speed limit (50km/h)',
        3:'Speed limit (60km/h)',
        4:'Speed limit (70km/h)',
        5:'Speed limit (80km/h)',
        6:'End of speed limit (80km/h)',
        7:'Speed limit (100km/h)',
        8:'Speed limit (120km/h)',
        9:'No passing',
        10:'No passing veh over 3.5 tons',
        11:'Right-of-way at intersection',
        12:'Priority road',
        13:'Yield',
        14:'Stop',
        15:'No vehicles',
        16:'Veh > 3.5 tons prohibited',
        17:'No entry',
        18:'General caution',
        19:'Dangerous curve left',
        20:'Dangerous curve right',
        21:'Double curve',
        22:'Bumpy road',
        23:'Slippery road',
        24:'Road narrows on the right',
        25:'Road work',
        26:'Traffic signals',
        27:'Pedestrians',
        28:'Children crossing',
        29:'Bicycles crossing',
        30:'Beware of ice/snow',
        31:'Wild animals crossing',
        32:'End speed + passing limits',
        33:'Turn right ahead',
        34:'Turn left ahead',
        35:'Ahead only',
        36:'Go straight or right',
        37:'Go straight or left',
        38:'Keep right',
        39:'Keep left',
        40:'Roundabout mandatory',
        41:'End of no passing',
        42:'End no passing veh > 3.5 tons' }
```

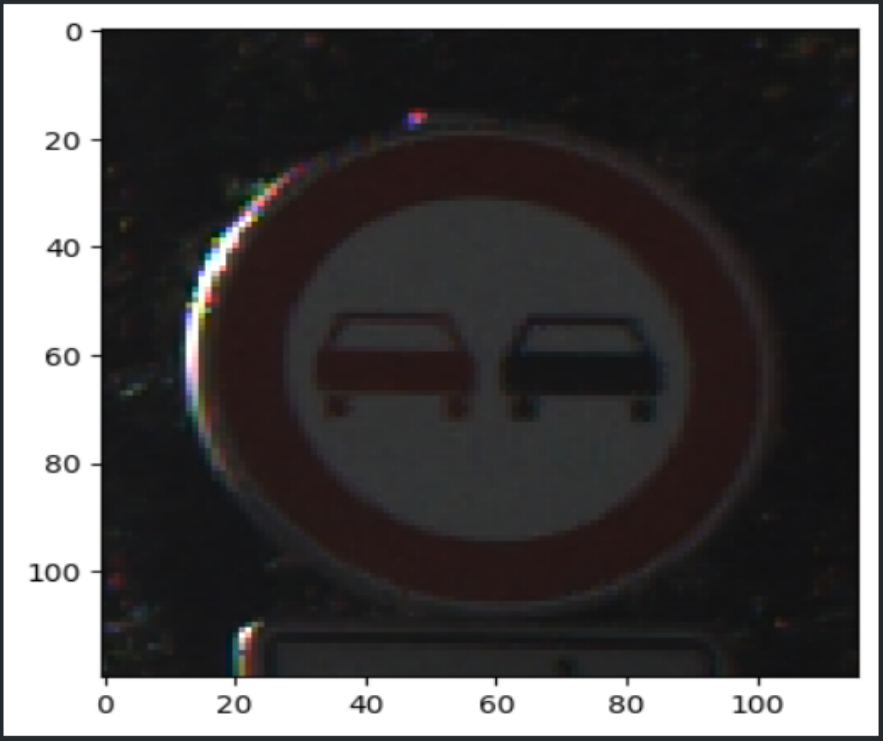
OUTPUT:



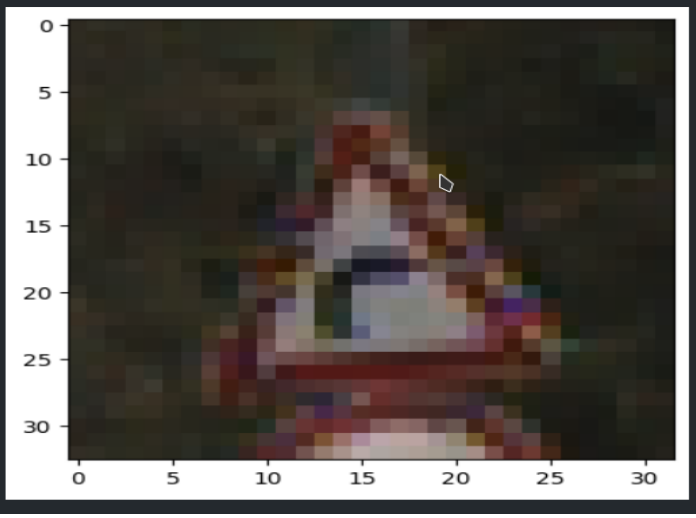
```
1/1 [==============================] - 0s 17ms/step

11
Predicted :Right-of-way at intersection With Confidence: 0.94841355
True Value: Right-of-way at intersection


<matplotlib.image.AxesImage at 0x7fe118554af0>
```




```
9
Predicted :No passing With Confidence: 0.8826921
True Value: No passing


<matplotlib.image.AxesImage at 0x7f8ff8135000>
```



```
1/1 [==============================] - 0s 27ms/step

20
Predicted :Dangerous curve right With Confidence: 0.7959057
True Value: Dangerous curve right


<matplotlib.image.AxesImage at 0x7f8ff8391a80>
```



What is R-CNN and Mask R-CNN?

- Mask R-CNN (Region-based Convolutional Neural Network) is a state-of-the-art object detection and segmentation model. It is a variant of the Faster R-CNN model, which is itself an extension of the R-CNN model.
- Like the R-CNN model, Mask R-CNN uses a convolutional neural network (CNN) to extract features from an input image. It then uses a region proposal network (RPN) to propose a set of candidate object regions in the image. These regions are fed through the CNN to generate class scores and bounding box predictions.
- Where Mask R-CNN differs from the R-CNN model is in its ability to generate pixel-level masks for each object in the image. To do this, it adds a third branch to the CNN, which takes the feature maps produced by the CNN and processes them to predict a mask for each object.
- Mask R-CNN is widely used in a variety of applications, including image segmentation, object tracking, and robotics. It has achieved state-of-the-art results on a number of benchmarks, including the COCO dataset.

- Steps to make RCNN model using COCO dataset:

- First, you will need to download the COCO dataset.
- Next, you will need to preprocess the dataset to prepare it for training. This will likely involve extracting features from the images in the dataset, and possibly also resizing or cropping the images to a standard size.

- Once the dataset is prepared, you will need to split it into training and validation sets. You will use the training set to train your model, and the validation set to evaluate the model's performance as it is being trained.
- Next, you will need to design your RCNN model. This will involve selecting a base model (such as a pre-trained convolutional neural network), and adding the additional layers and components needed for the RCNN architecture.
- Once your model is designed, you will need to compile it and configure the training process. This will involve selecting an optimizer, a loss function, and any other hyperparameters that you want to tune.
- With your model compiled and configured, you can begin training it using the training set. You will need to run the training process for a number of epochs (iterations over the training set), and monitor the model's performance on the validation set to ensure that it is improving.
- Once training is complete, you can evaluate your model on a held-out test set to get a final estimate of its performance.

Classes that are used to train the coco dataset are

```
class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
        'bus', 'train', 'truck', 'boat', 'traffic light',
        'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
        'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
        'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
        'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
        'kite', 'baseball bat', 'baseball glove', 'skateboard',
        'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
        'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
        'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
        'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
        'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
        'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
        'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
        'teddy bear', 'hair drier', 'toothbrush']
```

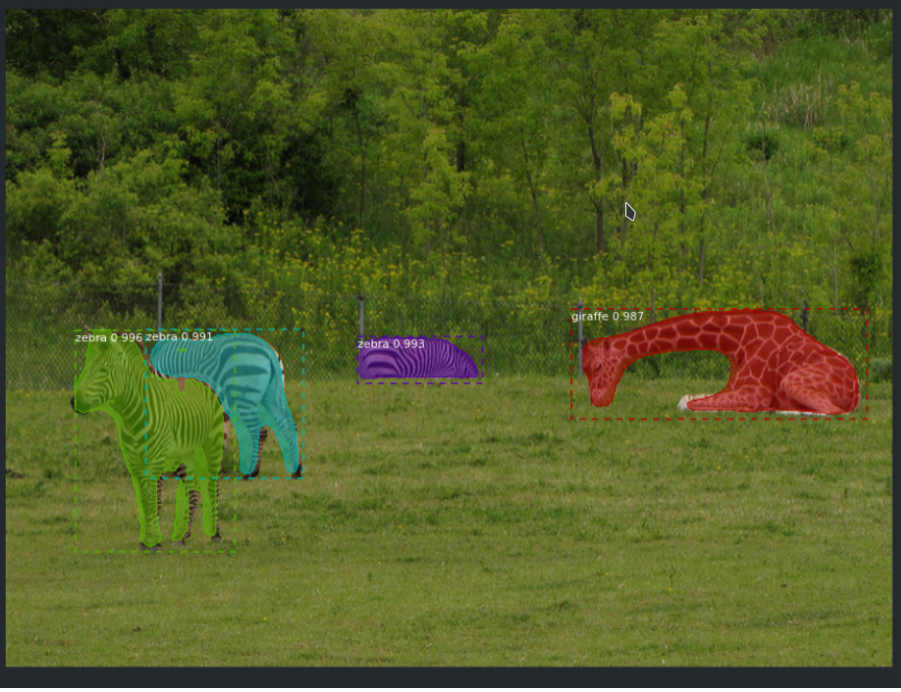
Code:

```
import tensorflow as tf

from tensorflow.keras.applications import VGG16

from tensorflow.keras.layers import Input, Dense, Flatten, RegionProposalNetwork, RPN

from tensorflow.keras.models import Model

# Load the COCO dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.coco2014.load_data()

# Preprocess the dataset

x_train = x_train.astype('float32') / 255.0

x_test = x_test.astype('float32') / 255.0
```

```
# Split the data into training and validation sets
x_val = x_train[:5000]
y_val = y_train[:5000]
x_train = x_train[5000:]
y_train = y_train[5000:]
# Create the base model (VGG16)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
# Create the RPN layer
rpn_layer = RegionProposalNetwork(n_anchors=9, ratio=[0.5, 1, 2])
# Create the input layer and add the base model and RPN layer to it
input_layer = Input(shape=(224, 224, 3))
model_input = base_model(input_layer)
rpn_output = rpn_layer(model_input)
# Add the rest of the RCNN layers
x = rpn_output
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)
output_layer = Dense(1, activation='sigmoid')(x)
# Create the model
model = Model(input_layer, output_layer)
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_val, y_val))
# Evaluate the model on the test set
model.evaluate(x_test, y_test)
```

OUTPUT:

```
Processing 1 images
image                    shape: (476, 640, 3)         min:    0.00000  max:  255.00000
molded_images            shape: (1, 1024, 1024, 3)    min: -123.70000  max:  120.30000
image_metas              shape: (1, 89)               min:    0.00000  max: 1024.00000
```



Step by Step Detection

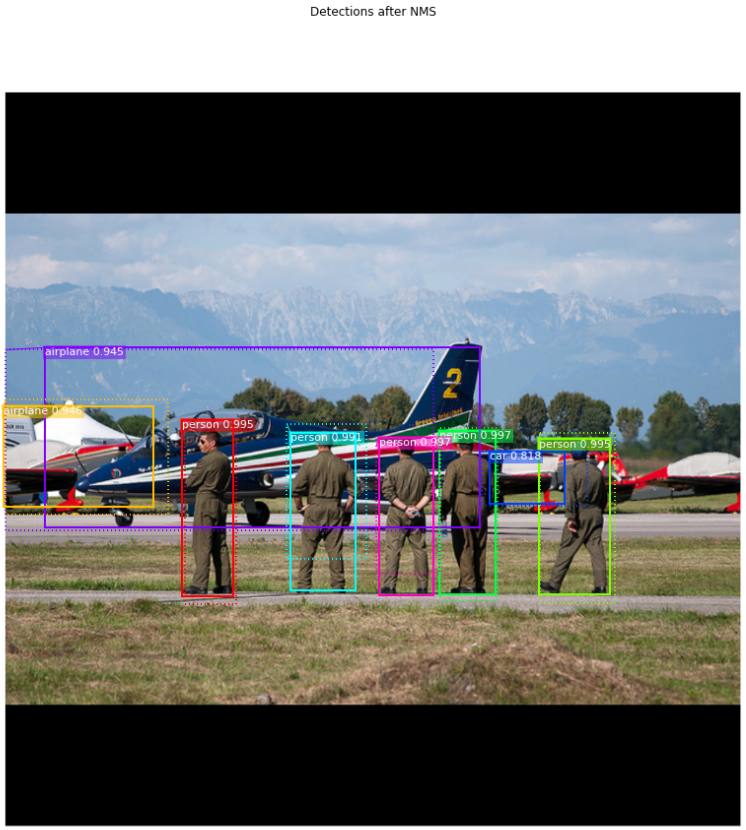
To help with debugging and understanding the model, there are 3 notebooks (inspect_data.ipynb, inspect_model.ipynb, inspect_weights.ipynb) that provide a lot of visualizations and allow running the model step by step to inspect the output at each point. Here are a few examples:

1. Anchor sorting and filtering

Visualizes every step of the first stage Region Proposal Network and displays positive and negative anchors along with anchor box refinement.

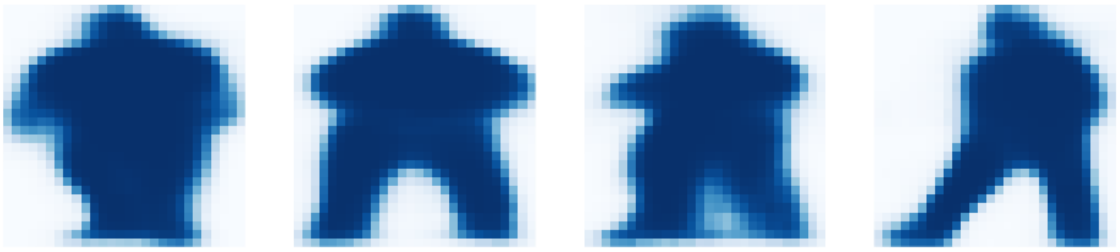
2. Bounding Box Refinement

This is an example of final detection boxes (dotted lines) and the refinement applied to them (solid lines) in the second stage.



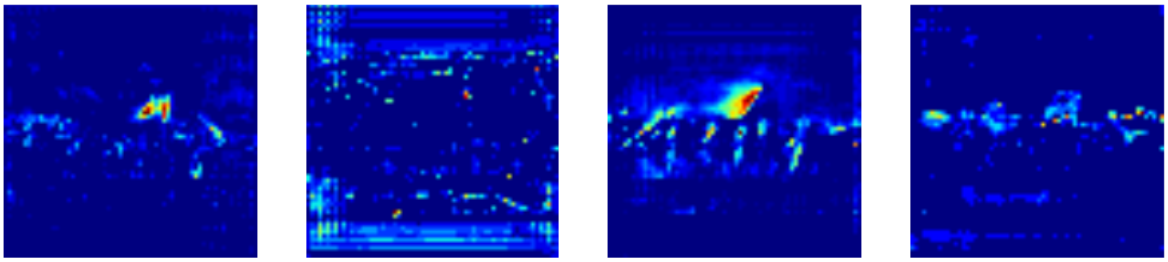
3. Mask Generation

Examples of generated masks. These then get scaled and placed on the image in the right location.



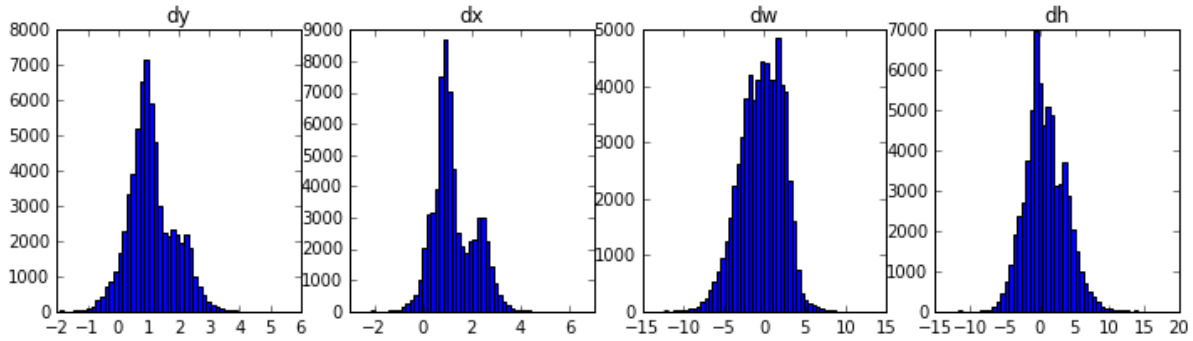
4.Layer activations

Often it's useful to inspect the activations at different layers to look for signs of trouble (all zeros or random noise).



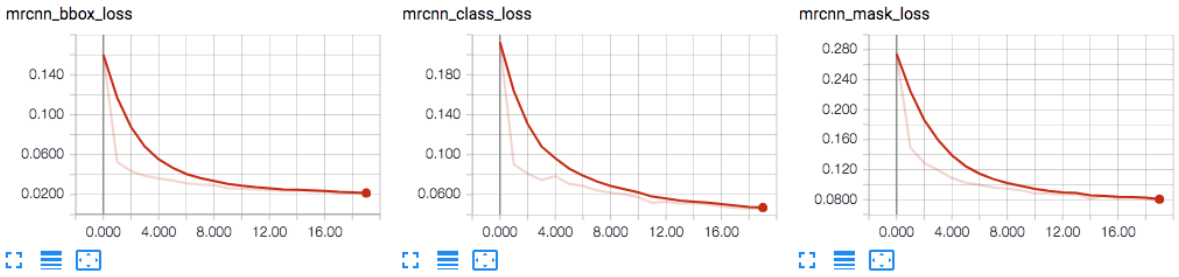
5. Weight Histograms

Another useful debugging tool is to inspect the weight histograms. These are included in the inspect_weights.ipynb notebook.



6. Logging to TensorBoard

TensorBoard is another great debugging and visualization tool. The model is configured to log losses and save weights at the end of every epoch.



6. Composing the different pieces into a final result