Lab3 Additional Notes

September 7, 2018

Lab3

tensorflow.js examples
source code of teachable machine
pretrained model

Since I am a bit curious about how exactly does the Google Teachable Machine works, so I did some digging. As it turns out, the teachable machine is actually a project built with deeplearn.js library.

As I see it, it seems like a transfer learning process. The machine feed the webcam images collected into a pre-trained model of SqueezeNet (a light weight neural network with AlexNet level accuracy but with 50x fewer parameters and <0.5MB model size) trained on the ImageNet dataset. So for the webcam images captured, first it does some standard ImageNet pre-processing before inferring through the model. This method returns pre-softmax logits, which then get feed into a KNN (k-nearest neighbors) classifier (with k set to 10) to recognize the ImageNet classes (as we see from the demo to classify the 3 gifs).

So with some basic understanding of the model, we can see that actually all the training examples we feed into the teachable machine through webcam first go through a pre-trained SqueezeNet Model to extract some high level features (it's quite deep), like smiles, hands up, hands down, gestures, etc., and then this feature set is feed into the KNN for final Gif classification.

Therefore regarding the 3rd question of lab3, of course the machine with more training images captured performs the best, but actually the difference is quite small. If the gathered training images number reaches to a certain level, say 20-30 (because k is set to 10 in the KNN classifier), then the classification result could be pretty accurate, even compared to the machine with more than 100+ images, if the features you displayed in the image are apparent, like hands up and down. The training phrase is actually with the KNN, not the SqueezeNet.

javascript source code of SqueezeNet pulled from github

```
In [ ]: /**
          * Infer through SqueezeNet, assumes variables have been loaded. This does
          * standard ImageNet pre-processing before inferring through the model. This
          * method returns named activations as well as pre-softmax logits. The user
          * needs to clean up namedActivations after inferring.
```

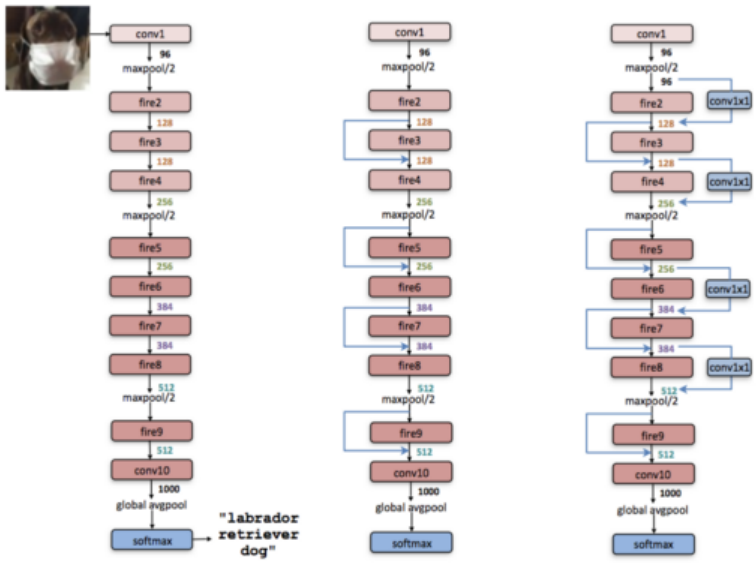

Figure 2: Macroarchitectural view of our SqueezeNet architecture. Left: SqueezeNet (Section 3.3); Middle: SqueezeNet with simple bypass (Section 6); Right: SqueezeNet with complex bypass (Section 6).

```
 *
 * @param preprocessedInput preprocessed input Array.
 * @return Named activations and the pre-softmax logits.
 */
infer(preprocessedInput) {
  const namedActivations = {};

  const avgpool10 = this.math.scope((keep) => {
    const conv1 = this.math.conv2d(
        preprocessedInput, this.variables['conv1_W:0'],
        this.variables['conv1_b:0'], 2, 0);
    const conv1relu = keep(this.math.relu(conv1));

    namedActivations['conv_1'] = conv1relu;

    const pool1 = keep(this.math.maxPool(conv1relu, 3, 2, 0));
    namedActivations['maxpool_1'] = pool1;

    const fire2 = keep(this.fireModule(pool1, 2));
    namedActivations['fire2'] = fire2;

    const fire3 = keep(this.fireModule(fire2, 3));
    namedActivations['fire3'] = fire3;
```

```
    // Because we don't have uneven padding yet, manually pad the ndarray on
    // the right.
    const fire3Reshape2d =
        fire3.as2D(fire3.shape[0], fire3.shape[1] * fire3.shape[2]);
    const fire3Sliced2d = this.math.slice2D(
        fire3Reshape2d, [0, 0],
        [fire3.shape[0] - 1, (fire3.shape[1] - 1) * fire3.shape[2]]);
    const fire3Sliced = fire3Sliced2d.as3D(
        fire3.shape[0] - 1, fire3.shape[1] - 1, fire3.shape[2]);
    const pool2 = keep(this.math.maxPool(fire3Sliced, 3, 2, 0));
    namedActivations['maxpool_2'] = pool2;

    const fire4 = keep(this.fireModule(pool2, 4));
    namedActivations['fire4'] = fire4;

    const fire5 = keep(this.fireModule(fire4, 5));
    namedActivations['fire5'] = fire5;

    const pool3 = keep(this.math.maxPool(fire5, 3, 2, 0));
    namedActivations['maxpool_3'] = pool3;

    const fire6 = keep(this.fireModule(pool3, 6));
    namedActivations['fire6'] = fire6;

    const fire7 = keep(this.fireModule(fire6, 7));
    namedActivations['fire7'] = fire7;

    const fire8 = keep(this.fireModule(fire7, 8));
    namedActivations['fire8'] = fire8;

    const fire9 = keep(this.fireModule(fire8, 9));
    namedActivations['fire9'] = fire9;

    const conv10 = keep(this.math.conv2d(
        fire9, this.variables['conv10_W:0'],
        this.variables['conv10_b:0'], 1, 0));
    namedActivations['conv10'] = conv10;

    return this.math.avgPool(conv10, conv10.shape[0], 1, 0).as1D();
  });

  return {namedActivations, logits: avgpool10};
}

fireModule(input, fireId) {
  const y1 = this.math.conv2d(
      input, this.variables['fire' + fireId + '/squeeze1x1_W:0'],
      this.variables['fire' + fireId + '/squeeze1x1_b:0'], 1, 0);
```

```
    const y2 = this.math.relu(y1);
    const left1 = this.math.conv2d(
        y2, this.variables['fire' + fireId + '/expand1x1_W:0'],
        this.variables['fire' + fireId + '/expand1x1_b:0'], 1, 0);
    const left2 = this.math.relu(left1);

    const right1 = this.math.conv2d(
        y2, this.variables['fire' + fireId + '/expand3x3_W:0'],
        this.variables['fire' + fireId + '/expand3x3_b:0'], 1, 1);
    const right2 = this.math.relu(right1);

    return this.math.concat3D(left2, right2, 2);
  }

export default SqueezeNet;
```

the tensorflow version of SqueezeNet model from cs231n Stanford

```
In [ ]: import tensorflow as tf

NUM_CLASSES = 1000
def fire_module(x,inp,sp,e11p,e33p):
    with tf.variable_scope("fire"):
        with tf.variable_scope("squeeze"):
            W = tf.get_variable("weights",shape=[1,1,inp,sp])
            b = tf.get_variable("bias",shape=[sp])
            s = tf.nn.conv2d(x,W,[1,1,1,1],"VALID")+b
            s = tf.nn.relu(s)
        with tf.variable_scope("e11"):
            W = tf.get_variable("weights",shape=[1,1,sp,e11p])
            b = tf.get_variable("bias",shape=[e11p])
            e11 = tf.nn.conv2d(s,W,[1,1,1,1],"VALID")+b
            e11 = tf.nn.relu(e11)
        with tf.variable_scope("e33"):
            W = tf.get_variable("weights",shape=[3,3,sp,e33p])
            b = tf.get_variable("bias",shape=[e33p])
            e33 = tf.nn.conv2d(s,W,[1,1,1,1],"SAME")+b
            e33 = tf.nn.relu(e33)
        return tf.concat([e11,e33],3)

class SqueezeNet(object):
    def extract_features(self, input=None, reuse=True):
        if input is None:
            input = self.image
        x = input
        layers = []
        with tf.variable_scope('features', reuse=reuse):
            with tf.variable_scope('layer0'):
```

```
                W = tf.get_variable("weights",shape=[3,3,3,64])
                b = tf.get_variable("bias",shape=[64])
                x = tf.nn.conv2d(x,W,[1,2,2,1],"VALID")
                x = tf.nn.bias_add(x,b)
                layers.append(x)
            with tf.variable_scope('layer1'):
                x = tf.nn.relu(x)
                layers.append(x)
            with tf.variable_scope('layer2'):
                x = tf.nn.max_pool(x,[1,3,3,1],strides=[1,2,2,1],padding='VALID')
                layers.append(x)
            with tf.variable_scope('layer3'):
                x = fire_module(x,64,16,64,64)
                layers.append(x)
            with tf.variable_scope('layer4'):
                x = fire_module(x,128,16,64,64)
                layers.append(x)
            with tf.variable_scope('layer5'):
                x = tf.nn.max_pool(x,[1,3,3,1],strides=[1,2,2,1],padding='VALID')
                layers.append(x)
            with tf.variable_scope('layer6'):
                x = fire_module(x,128,32,128,128)
                layers.append(x)
            with tf.variable_scope('layer7'):
                x = fire_module(x,256,32,128,128)
                layers.append(x)
            with tf.variable_scope('layer8'):
                x = tf.nn.max_pool(x,[1,3,3,1],strides=[1,2,2,1],padding='VALID')
                layers.append(x)
            with tf.variable_scope('layer9'):
                x = fire_module(x,256,48,192,192)
                layers.append(x)
            with tf.variable_scope('layer10'):
                x = fire_module(x,384,48,192,192)
                layers.append(x)
            with tf.variable_scope('layer11'):
                x = fire_module(x,384,64,256,256)
                layers.append(x)
            with tf.variable_scope('layer12'):
                x = fire_module(x,512,64,256,256)
                layers.append(x)
        return layers

    def __init__(self, save_path=None, sess=None):
        """Create a SqueezeNet model.
        Inputs:
        - save_path: path to TensorFlow checkpoint
        - sess: TensorFlow session
```

```
        - input: optional input to the model. If None, will use placeholder for input.
        """
        self.image = tf.placeholder('float',shape=[None,None,None,3],name='input_image')
        self.labels = tf.placeholder('int32', shape=[None], name='labels')
        self.layers = []
        x = self.image
        self.layers = self.extract_features(x, reuse=False)
        self.features = self.layers[-1]
        with tf.variable_scope('classifier'):
            with tf.variable_scope('layer0'):
                x = self.features
                self.layers.append(x)
            with tf.variable_scope('layer1'):
                W = tf.get_variable("weights",shape=[1,1,512,1000])
                b = tf.get_variable("bias",shape=[1000])
                x = tf.nn.conv2d(x,W,[1,1,1,1],"VALID")
                x = tf.nn.bias_add(x,b)
                self.layers.append(x)
            with tf.variable_scope('layer2'):
                x = tf.nn.relu(x)
                self.layers.append(x)
            with tf.variable_scope('layer3'):
                x = tf.nn.avg_pool(x,[1,13,13,1],strides=[1,13,13,1],padding='VALID')
                self.layers.append(x)
        self.classifier = tf.reshape(x,[-1, NUM_CLASSES])
```

Other than the parameters like filter numbers, kernel weight, kernel height, we can see that the structure pretty much remains the same. The SqueezeNet used in Teachable Machine does not make much changes regarding the its architecture. Except when it later feed the pre-softmax logits into KNN:

```
In [ ]: const knn = this.math.scope((keep) => {
          const frameLogits = this.captureFrameSqueezeNetLogits();

          if (this.trainLogitsMatrix === null) {
            let newTrainLogitsMatrix = null;

            for (let index = 0; index < CLASS_COUNT; index += 1) {
              newTrainLogitsMatrix = this.concat(
                newTrainLogitsMatrix, this.trainClassLogitsMatrices[index]);
            }

            this.trainLogitsMatrix = keep(this.math.clone(newTrainLogitsMatrix));
          }

          return this.math.matMul(
            this.trainLogitsMatrix.as2D(numExamples, 1000),
            frameLogits.as2D(1000, 1)).as1D();
```

```
      });

      const computeConfidences = () => {
        const values = knn.getValues();
        const kVal = Math.min(TOPK, numExamples);
        const topK = this.mathCPU.topK(knn, kVal);
        knn.dispose();

const indices = topK.indices.getValues();
```