Department of Computer Science, Rutgers University
Major in Computer Science - KM
Major in Computer Science - MM
Major in Computer Science - SP
Major in Computer Science - AS

Investigating Semantic Segmentation with Spiking Neural Networks

Kunal Mokashi
Moulindra Muchumari
Sandeep Panigrahy
Aravind Sivaramakrishnan

5/7/19
525 Brain Inspired Computing
Prof. Konstantinos Michmizos

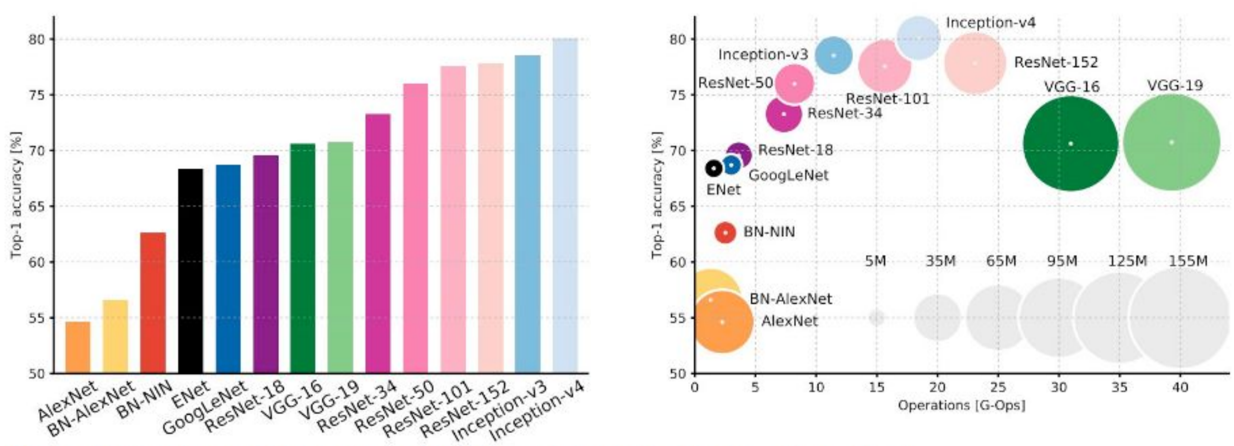
Abstract

We investigate the problem of segmenting an image semantically using spiking neural networks. We simulate a popular brain-inspired architecture for semantic segmentation using an SNN.

1. Introduction

Image segmentation is a fundamental process in many applications of image processing. The main goal of image segmentation is assigning a label to every pixel of the image such that pixels of same labels share similar characteristics. One of the primary applications of image segmentation is object detection, where the segmented parts of the image are processed by an object detection algorithm to assign labels differentiating different objects. This particular task is known as semantic segmentation and it is one of the key problems in computer vision.

There are many different image segmentation algorithms such as clustering methods, graph partition methods and most recently deep learning neural networks. Convolutional neural networks (CNN) have been used to perform image segmentation with remarkable accuracy as shown in the image below [4].



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

This paper investigates the application of spiking neural networks (SNN) to perform image segmentation. We used Nengo, a popular SNN library to simulate a state-of-the-art CNN architecture using SNNs rather than build an SNN from scratch. We chose to evaluate a VGG-16 FCN, which has high accuracy and high throughput. First, we created a set of images to train our neural network. We simulated the placement of twelve different household objects like books, etc. on a table and took images of the environment in various camera angles. We trained the VGG-16 FCN on these images and evaluated it. Then, we simulated VGG-16 FCN with an SNN using Nengo and evaluated on the same images.

2. Background

2.1 Convolutional Neural Networks (CNNs)

Convnets try to mimic the idea of *translation invariance*. *Invariance* refers to the human brain's property to recognize an object as an object, even when its appearance varies in some way. In particular, *translation invariance* is invariance when an object in an image is moved by some distance in a particular direction, while retaining its original shape and geometry.

Each layer of data in a convnet is a three-dimensional array of size $h \times w \times d$, where h and w are spatial dimensions, and d is the feature or channel dimension. The first layer is the image, with pixel size $h \times w$, and d color channels. Locations in higher layers correspond to the locations in the image they are path-connected to, which are called their *receptive fields*.

Convolutional neural networks are very loosely inspired by how the brain perceives visual information, which has been well studied in neuroscience literature. In their fundamental work [7], Hubel & Weisel suggested a new model for how mammals perceive the world visually. They showed that cat and monkey visual cortexes include neurons (analogous to convolutional filters) that exclusively respond to neurons in their direct environment. They also describe two basic types of visual neuron cells in the brain that each act in a different way: simple cells (S cells) and complex cells (C cells). The simple cells activate, for example, when they identify basic shapes as lines in a fixed area and a specific angle. The complex cells have larger receptive fields and their output is not sensitive to the specific position in the field. The complex cells continue to respond to a certain stimulus, even though its absolute position on the retina changes. The neocortex, which is the outermost layer of the brain, stores information hierarchically. Simon Thorpe's work [6] is often used as a justification for why convolutional neural networks work better than traditional computer vision approaches for most popular image-related talks.

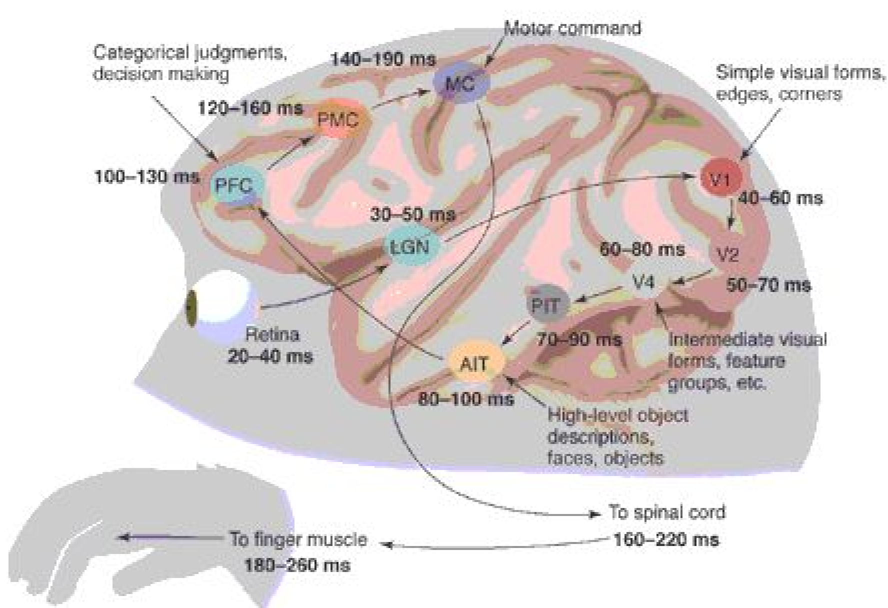


Figure 1: In [6], a plausible route between the retina and the muscles of the hand of a monkey during a categorization task is shown. Once the image is received by the retina, different neuronal cells are responsible for hierarchically constructing higher and higher level representations, which finally aid in task planning at the highest level, quickly followed by motor neurons being relayed so that the task is actually performed.

There are three basic components of traditional CNNs:

- **Convolution**: A convolution is traditionally a mathematical operation on two functions *f* and *g* that produces a function that expresses how the shape of one is modified by the other. In traditional image processing, *convolutional filters* are matrices that are designed for specific purposes such as line detection (similar to the SNN designed in Assignment 2). These filters "slide" across an image from top to bottom and left to right and are activated when they find a match to what they are looking for. Every entry in the 3D array described above can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter). It is possible to train such filters using learning rules like STDP [3], but for the purpose of this term project, we aim to only simulate them, which is possible due to the nature of their operation.
- **Pooling**: It is common to periodically insert a Pooling layer between convolutional layers to progressively reduce the spatial size of the representation. A common form of the pooling layer is the MaxPooling layer, which operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. Though pooling layers are commonly used to reduce the amount of parameters and computation in the network, they have no basis on neuroscience. As Geoff Hinton, the pioneer of the 2nd generation artificial neural network revolution has remarked, "The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster." [5]
- **Activation:** The *activation function* of a neuron in an artificial neural network defines the mapping between the input to the neurons and the output. In biologically inspired neural networks, the activation function is usually an abstraction representing the rate of action potential firing in the cell. The output of the activation function is calculated using the dot product of the neuron weights (W) and the neuron inputs (x). While a general fully connected artificial neural network computes a non-linear *function*, an activation function acting upon a convolutional layer allows us to compute a non-linear *filter*.

2.2 Image Segmentation with Spiking Neural Networks (SNNs)

SNNs use spiking neurons as the base computational unit of a neural network. The main advantage of the SNN is its ability to encode information based on timing of spikes which is not possible in previous generations of neurons. There has been very little research to develop SNN to perform image segmentation. This mainly due to the computational complexity involved in training multi-layered SNNs and the complexity of the problem.

Meftah [8] proposes to use unsupervised learning with SNNs to perform image segmentation. Meftah tackles the problem of segmenting images based on color. So, the first layer of the network are three nodes for red, green and blue values. These are fully connected to a hidden layer with radial-basis functions (RBF) as activation functions. The hidden layer transforms the input real RGB values to temporal values. All of the hidden layer neurons are fully connected to the output layer. They use Hebbian reinforcement learning to train the network. Meftah shows very high mean absolute error (MAE) in the segmented image.

Lin [9] developed an automatic image segmentation algorithm using SNNs. The SNN consists of three layers which use the leaky integrate-and-fire neuron model. The input and output of the network uses the time-to-first-spike coding to encode the image pixel value into timing of spikes. The first layer uses receptive fields to convert grayscale pixel values to spike trains. The middle layer contains one neuron per receptive field of input layer. The middle layer integrates the spike responses from the presynaptic neurons in the receptive field. The output layer stores the spike timing of middle layer in a matrix. This matrix represents the image segmentation through the segmentation threshold.

The major advancement that Lin builds on Meftah is the use of genetic algorithms. Lin augments the SNN training with a genetic algorithm to calculate the segmentation threshold of the output layer. They showed that the genetic algorithm results in the convergence of the segmentation threshold. However, the major drawback is that this genetic algorithm does not provide a generic segmentation threshold for all images. This results in running the genetic algorithm for every image and has too much delay for any practical use. It might be more optimal to approach image segmentation using SNNs by building upon already working solutions (in terms of accuracy and throughput) which are CNNs.

3. Experimental/Modeling Design

3.1 Creating Training Data

Figure 2: Some of the household objects found in our dataset. These objects are presented here in isolation.

For the training data, we use 3D models of different household objects (such as a box of crayons, a bar of soap, etc.) provided in [10]. To generate data for semantic segmentation, we physically simulate the effect of placing these 3D models in various settings, such as on a shelf, or a tabletop, using [11]. By placing a camera in the physics simulator, we are able to extract images of these physically simulated scenes, as well as ground truth per-pixel class labels. The different objects that we use in our dataset are: 'crayola_24_ct', 'expo_dry_erase_board_eraser', 'folgers_classic_roast_coffee', 'scotch_duct_tape', 'up_glucose_bottle', 'laugh_out_loud_joke_book', 'soft_white_lightbulb', 'kleenex_tissue_box', 'dove_beauty_bar', 'elmers_washable_no_run_school_glue' and 'rawlings_baseball'. Any pixel that does not belong to any of the object classes is assigned a consistent ground truth label of 'background'.

Figure 3: Example images from our dataset. Ground truth class labels are not shown.

3.2 Model used for semantic segmentation

We use the Fully Convolutional Networks (described in [1]) for semantic segmentation.

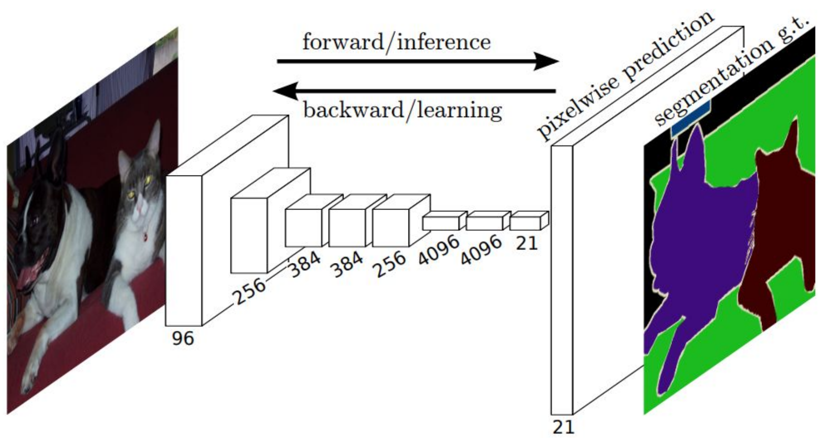


Figure 3: Architecture of a FCN for semantic segmentation. The fundamental idea behind the FCN is that transforming fully connected layers into convolution layers enables a classification net to output a heatmap, provided the network is trained with an appropriate spatial loss function. The numbers indicate the number of filters that are trained at each layer of the CNN. The filter size is 3x3. After each convolutional layer, a MaxPooling layer of size 2x2 is applied.

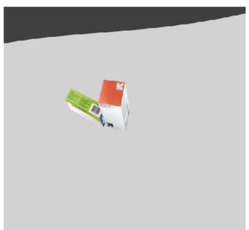
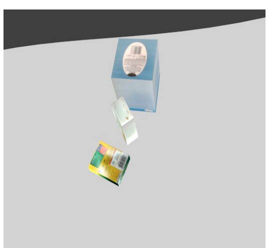
3.2 Training and simulation of the model

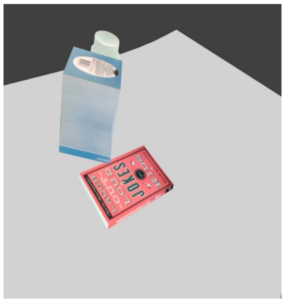
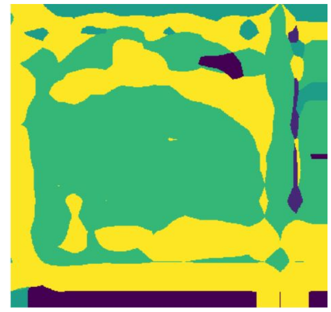
We implemented the FCN model using Keras and trained it end-to-end using the AdamOptimizer. We convolutionalized the popular VGG16 architecture, which has achieved remarkable success in object recognition tasks.

We simulated the trained model using nengo-dl [2]. Unlike traditional neural network libraries like TensorFlow or Keras, where the model's computational graph simply represents an abstract set of computations, Nengo's abstraction of the model's computational graph represents a stateful neural simulation, thus making it fundamentally temporal in nature. The network is simulated using leaky integrate-and-fire (LIF) neurons.

4. Results and Discussion

We provide some qualitative results of our simulation. The raw images used as input to the network are provided in the left-most column, and the SNN predictions generated by Nengo are on the right-most column. In the middle column, we provide the segmentation outputs of the CNN trained using Keras.

Original Image	CNN Image Segmentation	SNN Image Segmentation
		
		
		
		

We can see that in all the above cases, although the Keras FCN network does a reasonably good job of predicting the class of each pixel in the image, the FCN simulated by Nengo is unable to accurately predict the per-pixel class of each pixel. However, upon closer observation, we can see that all the images in the right-most column contain as many colors as there are classes present in the image (including a color that is assigned for the background). So we can conclude that at least qualitatively, the SNN is able to reason correctly about the classes of objects present in the image (including the background), but it is unable to appropriately cluster the pixels present in the image and assign the class labels per pixel correctly.

5. Conclusions

We present a way to use SNNs for image segmentation through utilizing highly accurate CNNs as a starting point. The results show that SNNs are able to perform object detection very accurately as evidenced by using correct colors to label the image. However, the pixel labelling is not that accurate. We suspect that the LIFs used by Nengo to approximate the operation of the FCN is unable to preserve the 2D structure of images in its simulation of CNNs with SNNs. This results in the decoder part of the SNN unable to correctly label the pixels with the colors given by the encoder. However, in the future, we would like to restructure the SNN architecture to use receptive fields to preserve the 2D structure of the images.

Acknowledgments

We would like to acknowledge the developers of Keras, keras-fcn, and nengo_dl, which helped us develop the code base for our project. We would also like to thank Prof. Konstantinos Michmizos along with Neelesh Kumar and Guangzhi Tang for their valuable support and feedback.

References

[1] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel

Rasmussen, Xuan Choo, Aaron Russell Voelker, and Chris Eliasmith. Nengo: a Python tool for building large-scale functional brain models. Frontiers in Neuroinformatics, 2014.

[2] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3431-3440).

[3] Lee C, Panda P, Srinivasan G, Roy K. Training Deep Spiking Convolutional Neural Networks With STDP-Based Unsupervised Pre-training Followed by Supervised Fine-Tuning. Front Neurosci. 2018;12:435. Published 2018 Aug 3. doi:10.3389/fnins.2018.00435

[4] Canziani, Alfredo et al. "An Analysis of Deep Neural Network Models for Practical Applications." *CoRR* abs/1605.07678 (2017): n. pag.

[5] https://www.reddit.com/r/MachineLearning/comments/2lmo0l/ama_geoffrey_hinton/clyj4jv/

[6] Thorpe, Simon J., and Michele Fabre-Thorpe. "Seeking categories in the brain." Science 291.5502 (2001): 260-263.

[7] Hubel, David H., and Torsten N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex." The Journal of physiology 160.1 (1962): 106-154.

[8] Boudjelal Meftah, A. Benyettou, Olivier Lezoray, M. Debakla. Image Segmentation with Spik-ing Neuron Network. 1st Mediterranean Conference on Intelligent Systems and Automation,2008, Algeria. 1019, pp.15-19, 2008.

[9] Lin X., Wang X., Cui W. (2014) An Automatic Image Segmentation Algorithm Based on Spiking Neural Network Model. In: Huang DS., Bevilacqua V., Premaratne P. (eds) Intelligent Computing Theory. ICIC 2014. Lecture Notes in Computer Science, vol 8588. Springer, Cham

[10] Singh, Arjun, et al. "Bigbird: A large-scale 3d database of object instances." 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2014.

[11] Chaitanya Mitash, Kostas E. Bekris and Abdeslam Boularias, A Self-supervised Learning System for Object Detection using Physics Simulation and Multi-view Pose Estimation. In Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS), Vancouver, Canada, 2017

Appendix

We provide final code for defining our model, loading a set of pre-trained weights, and simulating the model using nengo_dl.

```
"""
CS-525 Project - Image Segmentation : Simulation using Nengo DL.
"""

import sys
import os
from urllib.request import urlopen
import io
import shutil
import stat
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from PIL import Image
import tensorflow as tf
from tensorflow import keras
import nengo
import nengo_dl
from pylab import *
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
from tensorflow.keras.layers import *
from tensorflow.python.keras.engine import Layer
from tensorflow.keras.applications.vgg16 import *
from tensorflow.keras.models import *
import tensorflow.keras.backend as K
from tensorflow.keras.preprocessing import image

def resize_images_bilinear(X, height_factor=1, width_factor=1, target_height=None, target_width=None,
data_format='default'):
    if data_format == 'default':
        data_format = K.image_data_format()
    if data_format == 'channels_first':
        original_shape = K.int_shape(X)
        if target_height and target_width:
            new_shape = tf.constant(
                np.array((target_height, target_width)).astype('int32'))
        else:
```

```
            new_shape = tf.shape(X)[2:]
            new_shape *= tf.constant(
                np.array([height_factor, width_factor]).astype('int32'))
        X = permute_dimensions(X, [0, 2, 3, 1])
        X = tf.image.resize_bilinear(X, new_shape)
        X = permute_dimensions(X, [0, 3, 1, 2])
        if target_height and target_width:
            X.set_shape((None, None, target_height, target_width))
        else:
            X.set_shape(
                (None, None, original_shape[2] * height_factor, original_shape[3] * width_factor))
        return X
    elif data_format == 'channels_last':
        original_shape = K.int_shape(X)
        if target_height and target_width:
            new_shape = tf.constant(
                np.array((target_height, target_width)).astype('int32'))
        else:
            new_shape = tf.shape(X)[1:3]
            new_shape *= tf.constant(
                np.array([height_factor, width_factor]).astype('int32'))
        X = tf.image.resize_bilinear(X, new_shape)
        if target_height and target_width:
            X.set_shape((None, target_height, target_width, None))
        else:
            X.set_shape(
                (None, original_shape[1] * height_factor, original_shape[2] * width_factor, None))
        return X
    else:
        raise Exception('Invalid data_format: ' + data_format)


class BilinearUpSampling2D(Layer):
    def __init__(self, size=(1, 1), target_size=None, data_format='default', **kwargs):
        if data_format == 'default':
            data_format = K.image_data_format()
        self.size = tuple(size)
        if target_size is not None:
            self.target_size = tuple(target_size)
        else:
            self.target_size = None
        assert data_format in {
            'channels_last', 'channels_first'}, 'data_format must be in {tf, th}'
        self.data_format = data_format
        self.input_spec = [InputSpec(ndim=4)]
        super(BilinearUpSampling2D, self).__init__(**kwargs)

    def compute_output_shape(self, input_shape):
```

```
        if self.data_format == 'channels_first':
            width = int(self.size[0] * input_shape[2]
                        if input_shape[2] is not None else None)
            height = int(self.size[1] * input_shape[3]
                         if input_shape[3] is not None else None)
            if self.target_size is not None:
                width = self.target_size[0]
                height = self.target_size[1]
            return (input_shape[0],
                    input_shape[1],
                    width,
                    height)
        elif self.data_format == 'channels_last':
            width = int(self.size[0] * input_shape[1]
                        if input_shape[1] is not None else None)
            height = int(self.size[1] * input_shape[2]
                         if input_shape[2] is not None else None)
            if self.target_size is not None:
                width = self.target_size[0]
                height = self.target_size[1]
            return (input_shape[0],
                    width,
                    height,
                    input_shape[3])
        else:
            raise Exception('Invalid data_format: ' + self.data_format)

    def call(self, x, mask=None):
        if self.target_size is not None:
            return resize_images_bilinear(x, target_height=self.target_size[0], target_width=self.target_size[1],
data_format=self.data_format)
        else:
            return resize_images_bilinear(x, height_factor=self.size[0], width_factor=self.size[1],
data_format=self.data_format)

    def get_config(self):
        config = {'size': self.size, 'target_size': self.target_size}
        base_config = super(BilinearUpSampling2D, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))


def FCN_Vgg16_32s(input_shape=(640, 640, 3), weight_decay=0., batch_momentum=0.9, batch_shape=(1,) +
(640, 640) + (3, ), classes=12):
    if batch_shape:
        img_input = Input(batch_shape=batch_shape)
        image_size = batch_shape[1:3]
    else:
        img_input = Input(shape=input_shape)
```

```
    image_size = input_shape[0:2]

# Block 1
x = Conv2D(64, (3, 3), activation='relu', padding='same',
        name='block1_conv1', kernel_regularizer=l2(weight_decay))(img_input)
x = Conv2D(64, (3, 3), activation='relu', padding='same',
        name='block1_conv2', kernel_regularizer=l2(weight_decay))(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

# Block 2
x = Conv2D(128, (3, 3), activation='relu', padding='same',
        name='block2_conv1', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same',
        name='block2_conv2', kernel_regularizer=l2(weight_decay))(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

# Block 3
x = Conv2D(256, (3, 3), activation='relu', padding='same',
        name='block3_conv1', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same',
        name='block3_conv2', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same',
        name='block3_conv3', kernel_regularizer=l2(weight_decay))(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

# Block 4
x = Conv2D(512, (3, 3), activation='relu', padding='same',
        name='block4_conv1', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
        name='block4_conv2', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
        name='block4_conv3', kernel_regularizer=l2(weight_decay))(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

# Block 5
x = Conv2D(512, (3, 3), activation='relu', padding='same',
        name='block5_conv1', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
        name='block5_conv2', kernel_regularizer=l2(weight_decay))(x)
x = Conv2D(512, (3, 3), activation='relu', padding='same',
        name='block5_conv3', kernel_regularizer=l2(weight_decay))(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

# Convolutional layers transfered from fully-connected layers
x = Conv2D(4096, (7, 7), activation='relu', padding='same',
        name='fc1', kernel_regularizer=l2(weight_decay))(x)
x = Dropout(0.5)(x)
x = Conv2D(4096, (1, 1), activation='relu', padding='same',
```

```
            name='fc2', kernel_regularizer=l2(weight_decay))(x)
    x = Dropout(0.5)(x)

    # classifying layer
    x = Conv2D(classes, (1, 1), kernel_initializer='he_normal', activation='linear',
            padding='valid', strides=(1, 1), kernel_regularizer=l2(weight_decay))(x)

    x = BilinearUpSampling2D(size=(32, 32))(x)

    # flatten the output
    x = Flatten()(x)

    model = Model(img_input, x)
    return model


# Create Keras Model - Already defined in Keras. Train in nengo_dl
model = FCN_Vgg16_32s()
print(type(model))
model_weights = "C:\\Kunal\\Rutgers\\CS525\\Project\\apc_weights.hdf5"
# model.load_weights(model_weights)


class KerasNode:
    def __init__(self, keras_model):
        self.model = keras_model

    def pre_build(self, *args):
        self.model = keras.models.clone_model(self.model)

    def __call__(self, t, x):
        return self.model.call(image_array, training=False)

    def post_build(self, sess, rng):
        self.model.load_weights(model_weights)

# CREATE TENSOR NODE
class_names = ['background', 'crayola_24_ct', 'expo_dry_erase_board_eraser', 'folgers_classic_roast_coffee',
'scotch_duct_tape', 'up_glucose_bottle',
                'laugh_out_loud_joke_book', 'soft_white_lightbulb', 'kleenex_tissue_box', 'dove_beauty_bar',
'elmers_washable_no_run_school_glue', 'rawlings_baseball']

# For each pixel it has 12 classes and we have 640 * 640 pixels.
num_classes = 640 * 640 * 12

img1 = image.load_img("C:\\Users\\Kunal\\Downloads\\images\\image_00001.png")
image_array = image.img_to_array(img1)
```

```
# Preprocessing of the input image.
image_shape = image_array.shape
image_size = (640, 640)
img_h, img_w = image_array.shape[0:2]
pad_w = max(image_size[1] - img_w, 0)
pad_h = max(image_size[0] - img_h, 0)
image_array = np.lib.pad(image_array, ((pad_h//2, pad_h - pad_h//2),
                                       (pad_w//2, pad_w - pad_w//2), (0, 0)), 'constant', constant_values=0.)
image_array = np.expand_dims(image_array, axis=0)
image_array = preprocess_input(image_array)


# Create Nengo network using Keras Node.
with nengo.Network() as net:
    input_node = nengo.Node(output=image_array.flatten())
    keras_node = nengo_dl.TensorNode(KerasNode(model), size_in=np.prod(
        image_array.shape), size_out=num_classes)

    # connect up our input to our keras node
    nengo.Connection(input_node, keras_node, synapse=None)
    keras_input = nengo.Probe(input_node)
    keras_p = nengo.Probe(keras_node)

minibatch_size = 1


# Simulate using nengo dl
with nengo_dl.Simulator(net) as sim:
    sim.step()

tensornode_output = sim.data[keras_p]

output = tensornode_output[-1].reshape((640, 640, 12))
output = np.argmax(np.squeeze(output), axis=-1).astype(np.uint8)
sim.close()
```