o n t o l o g y*

@alpinemoose

2021

*This is a small code-essay. Although it is originally intended to be read from the class in the code itself, here we present it in a notebook-like format, where the call of each method serves as the title of the section, and its outputs as the content.

o n t o l o g y fue creado por @alpinemoose en londres el 3 de abril del 2021 & editado en barcelona el 4 de julio de 2021 entre las 5:42 y las 6:07.

este libro se creó, maquetó y editó en el taller de hiperpublicación 100/24, un proyecto de investigación y experimentación artística de la escocesa

ed. artefactos nativos

```
import ontology
o = ontology.Ontology()
```

Any (human) Language has an implicit ontology.
Those that speak in thoughts have reality be a representation.
Those that speak in science have reality be a complex system.
What is reality.. (for me)
This essay tries to explore a trascendental understanding from within a neural network, by relating their machanisms to categories (`from_within`).

```
o.language_as_vector_retains_semantics()
```

What we cannot interpret can have meaning.
Are these numbers inside the network a model, a void approximation..
Yet they hold attributes we give to things that are.
For the following example we use the Universal Sentence Encoder [USE].
The (vector) Representation the USE encodes the sentences as retains (a sense of) meaning.

Take this Sentence: `Any language has an implicit ontology.`
This is how an Algorithm interprets its similarity with these other sentences:

```
All speech is subject to an understanding of reality.  | SIMILARITY: 0.32372928
Humans communicate through structures.  | SIMILARITY: 0.21987574
This essay is written in english.  | SIMILARITY: 0.14686193
Water beetles inhabit ponds accross the universe.  | SIMILARITY: 0.063178
```

```
o.the_vgg16_convolutional_network()
```

The VGG16 is a convolutional neural network oriented for image classification [VGG16]. It concatenates convolutional and pooling layers, with a flattening and dense layers as the standard classification suite. We are now interested on the kernels, or filters at each convolutional layer. During training these slide over the (numericalized) pixels of the original image.

The filters in VGG16 are 3 by 3. The convolution operation multiplies the weights of the filter `w_i` with the corresponding pixel of the image `p_i`. These are then summed and the process is repeated with the kernel shifted. This combines the information of the kernel with that of the image in a new representation. Please refer to many of the better explanations and visualizations of this mechanism.

We will mostly experiment with the first convolutional layer:


```
{
    "class_name": "Conv2D",
    "config": {
        "name": "block1_conv1",
        "trainable": true,
        "dtype": "float32",
        "filters": 64,
        "kernel_size": [3, 3],
        "strides": [1, 1],
        "padding": "same",
        "data_format": "channels_last",
        "dilation_rate": [1, 1],
        "groups": 1,
        "activation": "relu",
        "use_bias": true,
        "kernel_initializer": {
            "class_name": "GlorotUniform",
            "config": {
            "seed": null
        }
    },
    "bias_initializer": {
        "class_name": "Zeros",
        "config": {}
    },
    "kernel_regularizer": null,
    "bias_regularizer": null,
    "activity_regularizer": null,
    "kernel_constraint": null,
    "bias_constraint": null
    },
    "name": "block1_conv1",
    "inbound_nodes": [
        [
            ["input_1", 0, 0, {}]
        ]
    ]
}
```

```
o.what_is_the_porpuse_in_vgg16()
```

During training, a network mutates its internal states to minimize a Loss. A loss is a function measuring how good is the network performance on the data, given a task.
VGG16 is already trained, and it can classify, look:

Image:

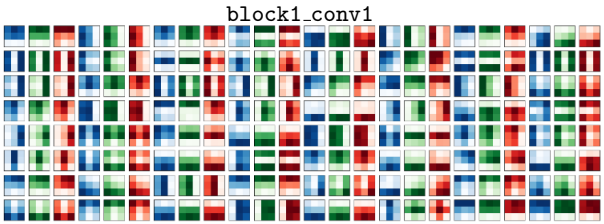
VGG16 classification:

```
('n03598930', 'jigsaw_puzzle', 0.7110014)
('n03291819', 'envelope', 0.037444476)
('n01930112', 'nematode', 0.029493595)
('n02840245', 'binder', 0.02353751)
('n03733281', 'maze', 0.020341983)
```

```
o.the_vgg16_kernels()
```

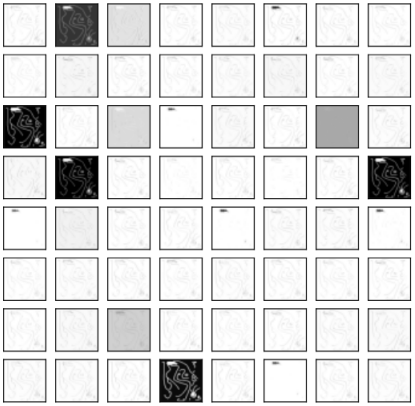
Visualization is translating numerical objects to spatial representations, we are as a species more fit to understand. The original image has three channels: RGB, which are changed to BGR during preprocessing.

Therefore each kernel will also have three channels. Let"s focus now on the first layer of the VGG16 network. The output dimension is (224,224,64).
This means there are 64 3x3 kernels in this layer.
Each kernel has 3 channels, each one for the RGB channels in the input image.



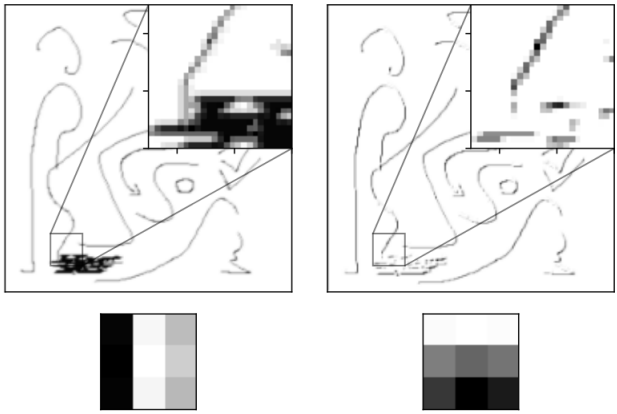

```
o.kernel_representation()
```

For each of the 64 kernels, we have a representation of the input image.



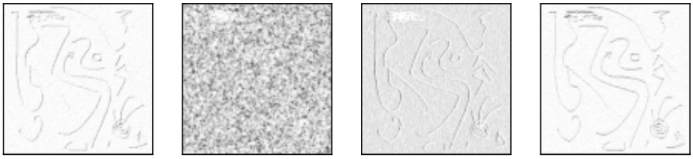
`o.elements_of_a_network_differ_in_representation()`

Close-up of representations after the first convolutional layer. Note the difference, which origins in the internal states. Here we have two examples highlighted.
It can be shown how different kernels focus on different patterns (features) on images. For example vertical or horizontal lines, corners, and many other geometrical elements.



```
o.dropout_experiments()
```

Dropout [DROPOUT] is a regularization technique used during training.
It randomly sets neurons of the layer to 0, to make the task harder to complete. Observers the image with pixels dropped out (turned to 0) with probability 0.1.
How this affects representations.



```
o.the_loss_function_as_inherent_porpuse()
```

Why is it so hard to untangle any pov, any understanding from Porpuse..
Evolution. Social. Movement..
This drives me crazy seriously.
A training network needs a hardcoded Loss function to train.
Training, reward, loss.. Porpu.. ARGH .!.....
For classification, the most used loss function is the Categorical Cross-Entropy. Usually a classifying network will yield a probability distribution (softmaxed logits). The ground truth will also be a probability distribution where labels are one-hot encoded.
The true value has probability 1 and all others 0. Watch how different predictions yield different losses.
At each training iteration, the internal states of the network are modified to try update the predictions produced as to decrease the loss.

```
loss = tf.keras.losses.CategoricalCrossentropy()
training_samples_true_label = [[0, 1, 0], [0, 0, 1]]

network_distributions_prediction = [[0.33, 0.34, 0.33], [0.33, 0.33, 0.34]]

loss(training_samples_true_label, network_distributions_prediction).numpy()
1.0788096

network_distributions_prediction = [[0.02, 0.95, 0.03], [0.005, 0.015, 0.98]]

loss(training_samples_true_label, network_distributions_prediction).numpy()
0.035747964
```

`o.a_self_referential_network()`

Consider now a self-referential network.

A network whose Inputs are its Own internal states.

Each neuron or element has a sensation (internal state).

This sensation makes its representations unique.

A dropout mechanism defines our category of space.

An instance of iterative training defines our category of time.

A loss function entangles porpuse as in change with tendency.

The neuron cannot acces the origin of such change, just infer a direction.

Careful because in this framework causality is not trivial!

Where is the unit? The network, the neuron? The weight? The ensembled networks? Is there a unit?

`o.final_note()`

This is an extremely naive essay.
I wanted to touch the idea of code as theory.
And explore a trascendental study of neural networks from the most contradictory empirical point of view.
Critique from within the Network.

Love xxx,

@alpinemoose

0.0.1 Source!

```
import json
import numpy as np
np.random.seed(619)
import jsbeautifier
import pandas as pd
import tensorflow as tf
import tensorflow_hub as hub
import matplotlib.pyplot as plt
tf.get_logger().setLevel('ERROR')
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions

'''
Meta comment:
This piece is intended to be read from the raw code of the Ontology class,
while in parallel the methods are executed sequentially in order to illustrate it.
Otherwise, explore the exposition through the .ipynb, .html or .pdf you will probably find attached.
'''

class Ontology:
    '''
    Any (human) Language has an implicit ontology.
    Those that speak in thoughts have reality be a representation.
    Those that speak in science have reality be a complex system.
    What is reality.. (for me)

    This essay tries to explore a trascendental understanding from within a neural network,
    by relating their machanisms to categories (from_within).

    '''

    def __init__(self):
        #self.embed = hub.load('https://tfhub.dev/google/universal-sentence-encoder/4')
        self.embed = hub.load('/home/gus/Desktop/TFM/finding-process-descriptions/models/universal-sentence-encoder_4')
        print(self.__doc__)

    def language_as_vector_retains_semantics(self):
        '''
        What we cannot interpret can have meaning.
        Are these numbers inside the network a model, a void approximation..
        Yet they hold attributes we give to things that are.
        For the following example we use the Universal Sentence Encoder [USE].
        The (vector) Representation the USE encodes the sentences as retains (a sense of) meaning.
        '''
        print(self.language_as_vector_retains_semantics.__doc__)
        sentence = "Any language has an implicit ontology."
        other_sentences = [
                'All speech is subject to an understanding of reality.',
                'Humans communicate through structures.',
                'This essay is written in english.',
                'Water beetles inhabit ponds accross the universe.'
            ]
        sentence_embeddings = self.embed([sentence]+other_sentences)
        print('Take this Sentence: ' + sentence, end = '\n')
        print('This is how an Algorithm interprets its similarity with these other sentences:')
        for idx, other_sentence in enumerate(other_sentences):
            similarity = np.inner(sentence_embeddings[0], sentence_embeddings[idx+1])
            print(other_sentence + ' | SIMILARITY: ' + str(similarity))

    def the_vgg16_convolutional_network(self):
        '''
        The VGG16 is a convolutional neural network oriented for image classification [VGG16].
        It concatenates convolutional and pooling layers, with a flattening and dense layers as the standard classification suite.
        We are now interested on the kernels, or filters at each convolutional layer.
        During training these slide over the (numericalized) pixels of the original image.
        The filters in VGG16 are 3 by 3.
        The convolution operation multiplies the weights of the filter w_i with the corresponding pixel of the image p_i.
        These are then summed and the process is repeated with the kernel shifted.
        This combines the information of the kernel with that of the image in a new representation.
        Please refer to many of the better explanations and visualizations of this mechanism.
        We will mostly experiment with the first convolutional layer:
        '''
```

```
        print(self.the_vgg16_convolutional_network.__doc__)
        self.vgg16 = VGG16()
        vgg16_config = self.vgg16.get_config()
        opts = jsbeautifier.default_options()
        opts.indent_size = 2
        print(jsbeautifier.beautify(json.dumps(vgg16_config['layers'][1]), opts))
        #print(self.vgg16.summary())

    def what_is_the_porpuse_in_vgg16(self):
        '''
        During training, a network mutates its internal states to minimize a Loss.
        A loss is a function measuring how good is the network performance on the data, given a task.
        VGG16 is already trained, and it can classify, look:
        '''
        print(self.what_is_the_porpuse_in_vgg16.__doc__)
        resized_image = load_img('test224.png', target_size=(224, 224))
        image_array = img_to_array(resized_image)
        image_reshaped = image_array.reshape((1, image_array.shape[0], image_array.shape[1], image_array.shape[2]))
        self.image = preprocess_input(image_reshaped)
        softmax_probability = self.vgg16.predict(self.image)
        label = decode_predictions(softmax_probability)
        print("Image:")
        display(resized_image)
        print("VGG16 classification:")
        for match in label[0]:
            print(match)

    def the_vgg16_kernels(self):
        '''
        Visualization is translating numerical objects to spatial representations, we are as a species more fit to understand.
        The original image has three channels: RGB, which are changed to BGR during pre-processing.
        Therefore each kernel will also have three channels.
        Let"s focus now on the first layer of the VGG16 network.
        The output dimension is (224,224,64).
        This means there are 64 3x3 kernels in this layer.
        Each kernel has 3 channels, each one for the RGB channels in the input image.
        '''
        print(self.the_vgg16_kernels.__doc__)
        layer = self.vgg16.layers[1]
        print(layer.name)
        w, b = layer.get_weights()
        self.detail_kernels = [np.array(w[:,:,:,5]), np.array(w[:,:,:,42])]
        fig, axes = plt.subplots(8, 24, sharex=True, figsize=(24,8), )
        for k in range(len(axes.flat)):
            channel = k%3
            ax = axes.flat[k]
            kernel = np.array(w[:,:,:,int(k/3)][channel])
            ax.imshow(kernel,cmap=['Blues','Greens','Reds'][channel])
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)
        plt.show()

    def kernel_representation(self):
        '''
        For each of the 64 kernels, we have a representation of the input image.
        '''
        print(self.kernel_representation.__doc__)
        input_layer = self.vgg16.layers[0]
        first_convolution = self.vgg16.layers[1]
        representations = first_convolution.call(input_layer.call(self.image))
        fig, axes = plt.subplots(8, 8, sharex=True, figsize=(6,6), )
        self.detail_representations = [np.array(representations[0,:,:,5]),np.array(representations[0,:,:,42])]
        for ax,k in zip(axes.flat,range(64)):
            representation = np.array(representations[0,:,:,k])
            ax.imshow(representation,cmap='Greys')
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)

    def elements_of_a_network_differ_in_representation(self):
        '''
        Close-up of representations after the first convolutional layer.
        Note the difference, which origins in the internal states.
        Here we have two examples highlighted.
        It can be shown how different kernels focus on different patterns (features) on images.
        For example vertical or horizontal lines, corners, and many other geometrical elements.
        '''
        print(self.elements_of_a_network_differ_in_representation.__doc__)
```

```
        dk = self.detail_kernels
        dr = self.detail_representations
        fig, axis = plt.subplots(2, 2, gridspec_kw={'height_ratios': [3, 1]}, figsize=(6,4))
        axis[0][0].imshow(dr[0],cmap='Greys', origin="lower")
        axis[1][0].imshow(dk[0][0],cmap='Greys')
        axis[0][1].imshow(dr[1],cmap='Greys', origin="lower")
        axis[1][1].imshow(dk[1][0],cmap='Greys')
        for a in [0,1]:
            ax = axis[0][a]
            axins = ax.inset_axes([0.5, 0.5, 0.5, 0.5])
            axins.imshow(dr[a], extent=(0,224,0,224), origin="lower",cmap='Greys')
            x1, x2, y1, y2 = 35, 60, 20, 45
            axins.set_xlim(x1, x2)
            axins.set_ylim(y1, y2)
            axins.set_xticklabels('')
            axins.set_yticklabels('')
            ax.indicate_inset_zoom(axins, edgecolor="black")
        for a in axis.flat:
            a.get_xaxis().set_visible(False)
            a.get_yaxis().set_visible(False)
        fig.tight_layout()

    def dropout_experiments(self):
        '''
        Dropout [DROPOUT] is a regularization technique used during training.
        It randomly sets neurons of the layer to 0, to make the task harder to complete.
        Observers the image with pixels dropped out (turned to 0) with probability 0.1.
        How this affects representations.
        '''
        print(self.dropout_experiments.__doc__)
        input_layer = self.vgg16.layers[0]
        first_convolution = self.vgg16.layers[1]
        dropout_layer = tf.keras.layers.Dropout(0.1)
        dropout_image = dropout_layer(input_layer.call(self.image),training=True)
        out = first_convolution.call(dropout_image)
        fig, ax = plt.subplots(1, 1, sharex=True, figsize=(4,4))
        ax.imshow(dropout_image[0,:,:,0],cmap='gray')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        plt.show()

        fig, axes = plt.subplots(1, 4, sharex=True, figsize=(8,2))
        for ax,k in zip(axes.flat,range(4)):
            representation = np.array(out[0,:,:,k])
            ax.imshow(representation,cmap='Greys')
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)

    def the_loss_function_as_inherent_porpuse(self):
        '''
        Why is it so hard to untangle any pov, any understanding from Porpuse..
        Evolution. Social. Movement..
        This drives me crazy seriously.
        A training network needs a hardcoded Loss function to train.
        Training, reward, loss.. Porpu.. ARGH .!.......
        For classification, the most used loss function is the Categorical Cross-Entropy.
        Usually a classifying network will yield a probability distribution (softmaxed logits).
        The ground truth will also be a probability distribution where labels are one-hot encoded.
        The true value has probability 1 and all others 0.
        Watch how different predictions yield different losses.
        At each training iteration,
        the internal states of the network are modified to try update the predictions produced as to decrease the loss.
        '''
        print(self.the_loss_function_as_inherent_porpuse.__doc__)
        print('loss = tf.keras.losses.CategoricalCrossentropy()')
        print('training_samples_true_label = [[0, 1, 0], [0, 0, 1]]')
        print('')
        print('network_distributions_prediction = [[0.33, 0.34, 0.33], [0.33, 0.33, 0.34]]')
        print('')
        print('loss(training_samples_true_label, network_distributions_prediction).numpy()')
        print('1.0788096')
        print('')
        print('network_distributions_prediction = [[0.02, 0.95, 0.03], [0.005, 0.015, 0.98]]')
        print('')
        print('loss(training_samples_true_label, network_distributions_prediction).numpy()')
        print('0.035747964')
```

```
def a_self_referential_network(self):
    '''
    Consider now a self-referential network.
    A network whose Inputs are its Own internal states.
    Each neuron or element has a sensation (internal state).
    This sensation makes its representations unique.
    A dropout mechanism defines our category of space.
    An instance of iterative training defines our category of time.
    A loss function entangles porpuse as in change with tendency.
    The neuron cannot acces the origin of such change, just infer a direction.
    Careful because in this framework causality is not trivial!
    Where is the unit? The network, the neuron? The weight? The ensembled networks? Is there a unit?
    '''
    print(self.a_self_referential_network.__doc__)

def final_note(self):
    '''
    This is an extremely naive essay.
    I wanted to touch the idea of code as theory.
    And explore a trascendental study of neural networks from the most contradictory empirical point of view.
    Critique from within the Network.
    Love xxx,
    @alpinemoose
    '''
    print(self.final_note.__doc__)

def _references(self):
    '''
    [USE] Cer, Daniel & Yang, Yinfei & Kong, Sheng-yi & Hua, Nan & Limtiaco, Nicole & John, Rhomni & Constant,
     Noah & Guajardo-Cespedes, Mario & Yuan, Steve & Tar, Chris & Sung, Yun-Hsuan & Strope, Brian & Kurzweil, Ray.
     (2018). Universal Sentence Encoder.
    [VGG16] Simonyan, Karen & Zisserman, Andrew.
    (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.
    [DROPOUT] Srivastava, Nitish & Hinton, Geoffrey & Krizhevsky, Alex & Sutskever, Ilya & Salakhutdinov, Ruslan.
    (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting.
    Journal of Machine Learning Research. 15. 1929-1958.
    '''
    print(self._references.__doc__)
```

Artefactos Nativos es un laboratorio editorial especializado en internet en el que se explora la poética de la web y se imprimen pantallas en forma de fanzines, libros experimentales, ediciones piratas y obras de escritura no-creativa.

Hiperpublicación 100/24 es un encuentro de 24 horas en el que se crean, maquetan y editan 100 libros utilizando técnicas de escritura no-creativa y material apropiado de internet.

Un libro (según la UNESCO) es justamente eso, ni más, ni menos.

Barcelona, 2021.