Implementing Grad-CAM in PyTorch

Stepan Ulyanin · Follow
11 min read · Feb 22, 2019

Recently I have come across a chapter in François Chollet's "Deep Learning With Python" book, describing the implementation of Class Activation Mapping for the VGG16 network. He implemented the algorithm using Keras as he is the creator of the library. Hence, my instinct was to re-implement the CAM algorithm using PyTorch.

Grad-CAM

The algorithm itself comes from this paper. It was a great addition to the computer vision analysis tools for a single primary reason. It provides us with a way to look into what particular parts of the image influenced the whole model's decision for a specifically assigned label. It is particularly useful in analyzing wrongly classified samples. The Grad-CAM algorithm is very intuitive and reasonably simple to implement.

The intuition behind the algorithm is based upon the fact that the model must have seen some pixels (or regions of the image) and decided on what object is present in the image. Influence in the mathematical terms can be described with a gradient. On the high-level, that is what the algorithm does. It starts with finding the gradient of the most dominant logit with respect to the latest activation map in the model. We can interpret this as some encoded features that ended up activated in the final activation map persuaded the model as a whole to choose that particular logit (subsequently the corresponding class). The gradients are then pooled channel-wise, and the activation channels are weighted with the corresponding gradients, yielding the collection of weighted activation channels. By inspecting these

channels, we can tell which ones played the most significant role in the decision of the class.

In this post I am going to re-implement the Grad-CAM algorithm, using PyTorch and, to make it a little more fun, I am going to use it with different architectures.

VGG19

In this part I will try to reproduce the Chollet's results, using a very similar model — VGG19 (note that in the book he used VGG16). The main idea in my implementation is to dissect the network so we could obtain the activations of the last convolutional layer. Keras has a very straight forward way of doing this via Keras functions. However, in PyTorch I had to jump through some minor hoops.

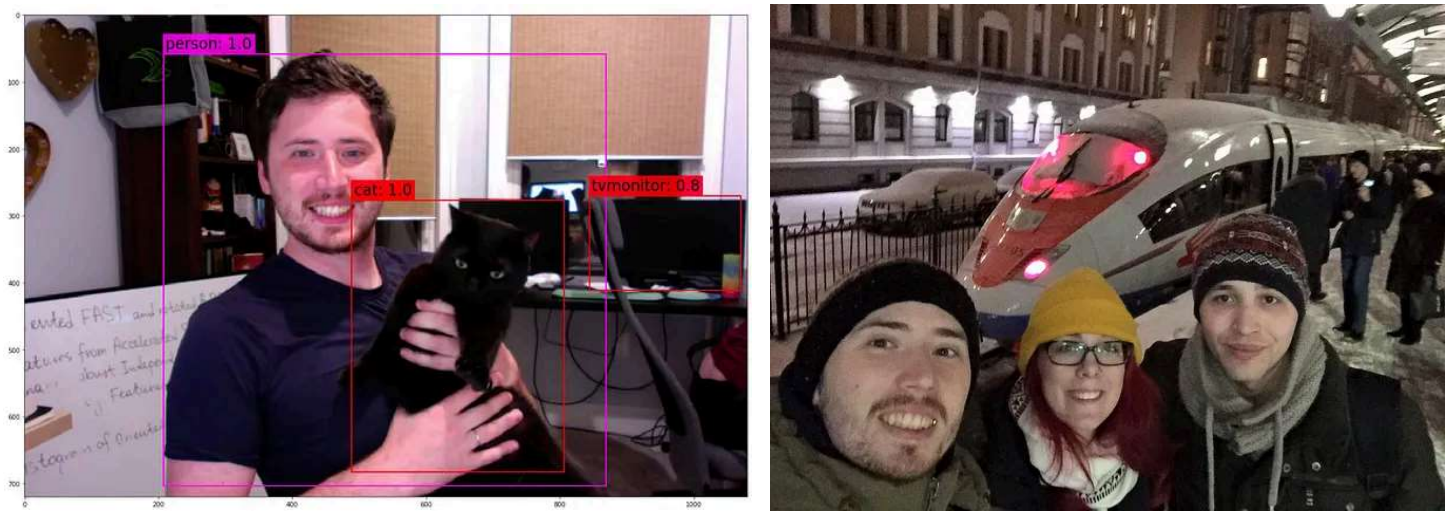
The strategy is defined as follows:

- Load the VGG19 model
- Find its last convolutional layer
- Compute the most probable class
- Take the gradient of the class logit with respect to the activation maps we have just obtained
- Pool the gradients
- Weight the channels of the map by the corresponding pooled gradients
- Interpolate the heat-map

I set aside a few images (including the images of the elephants Chollet used in his book) from the ImageNet dataset to investigate the algorithm. I also applied the Grad-CAM to some photographs from my Facebook to see how the algorithm works in the "field" conditions. Here are the original images we are going to be working with:

Left: the image of the elephants Chollet uses in his book. Center and right: white shark images from ImageNet

The images of iguanas from ImageNet dataset



Left: me holding my cat Luna while applying the YOLO model. Right: Me, my wife and my friend are taking a bullet train in Moscow

Ok, let's load up the VGG19 model from the `torchvision` module and prepare the transforms and the dataloader:

```python
import torch
import torch.nn as nn
from torch.utils import data
from torchvision.models import vgg19
from torchvision import transforms
from torchvision import datasets
import matplotlib.pyplot as plt
import numpy as np

# use the ImageNet transformation
transform = transforms.Compose([transforms.Resize((224, 224)),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.

# define a 1 image dataset
dataset = datasets.ImageFolder(root='./data/Elephant/', transform=transform)

# define the dataloader to load that single image
dataloader = data.DataLoader(dataset=dataset, shuffle=False, batch_size=1)
```

load_vgg19.py hosted with ❤️ by GitHub view raw

Here I import all the standard stuff we use to work with neural networks in PyTorch. I use the basic transform needed to use any model that was trained on the ImageNet dataset, including the image normalization. I am going to feed one image at a time, hence I define my dataset to be the image of the elephants, in attempt to obtain similar results as in the book.

Here comes the tricky part (trickiest in the whole endeavor, but not too tricky). We can compute the gradients in PyTorch, using the `.backward()` method called on a `torch.Tensor`. This is exactly what I am going to do: I am going to call `backward()` on the most probable logit, which I obtain by performing the forward pass of the image through the network. **However, PyTorch only caches the gradients of the leaf nodes in the computational graph, such as weights, biases and other parameters.** The gradients of the output with respect to the activations are merely intermediate values and are discarded as soon as the gradient propagates through them on the way back. So what are our options?

Hook 'Em

If you graduated from the University of Texas at Austin as I did you will like this part. There is a callback instrument in PyTorch: hooks. Hooks can be used in

different scenarios, ours is one of them. This part of the PyTorch documentation tells us exactly how to attach a hook to our intermediate values to pull the gradients out of the model before they are discarded. The documentation tells us:

> *The hook will be called every time a gradient with respect to the Tensor is computed.*

Now we know that we have to register the backward hook to the activation map of the last convolutional layer in our VGG19 model. Let's find where to hook.

We can easily observe the VGG19 architecture by calling the `vgg19(pretrained=True)` :

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Pretrained models in PyTorch heavily utilize the `Sequential()` modules which in most cases makes them hard to dissect, we will see the example of it later.

In the image we see the whole VGG19 architecture. I highlighted the last convolutional layer in the feature block (including the activation function). Well, now we know that we want to register the backward hook at the 35th layer of the

feature block of our network. This is exactly what I am going to do. Also, it is worth mentioning that it is necessary to register the hook inside the `forward()` method, to avoid the issue of registering hook to a duplicate tensor and subsequently losing the gradient.

As you can see there is a remaining max pooling layer left in the feature block, not to worry, I will add this layer in the `forward()` method.

This looks great so far, we can finally get our gradients and the activations out of the model.

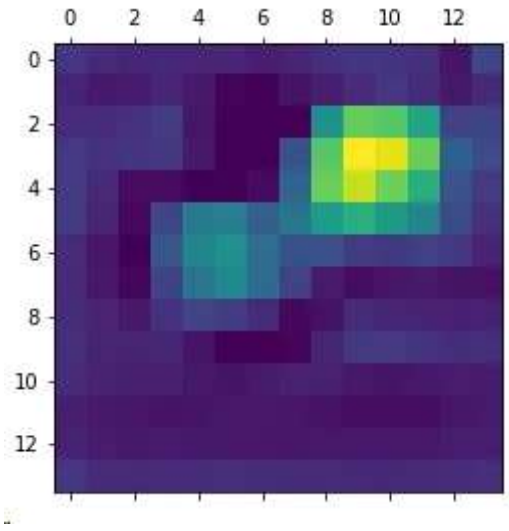
Drawing CAM

First, let's make the forward pass through the network with the image of the elephants and see what the VGG19 predicts. Don't forget to set your model into the evaluation mode, otherwise you can get very random results:

As expected, we get same results as Chollet gets in his book:

```
Predicted: [('n02504458', 'African_elephant', 20.891441),
('n01871265', 'tusker', 18.035757), ('n02504013', 'Indian_elephant',
15.153353)]
```

Now, we are going to do the back-propagation with the logit of the 386th class which represents the ‘African_elephant’ in the ImageNet dataset.



Heat-map for the elephant image

Finally, we obtain the heat-map for the elephant image. It is a 14x14 single channel image. **The size is dictated by the spacial dimensions of the activation maps in the last convolutional layer of the network.**

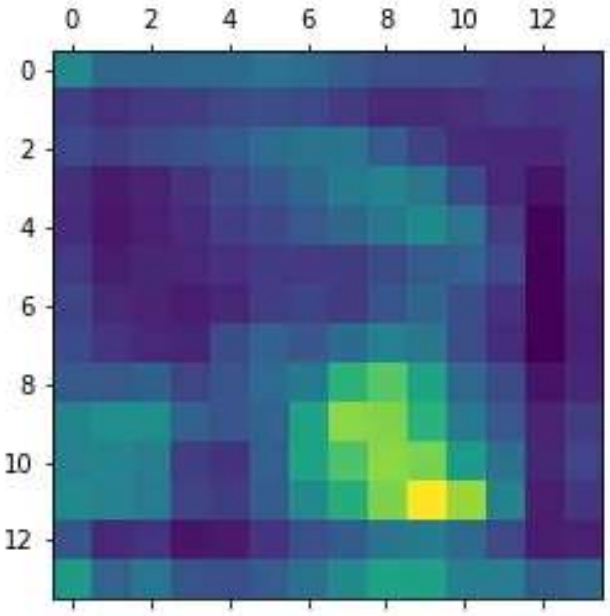
Now, we can use OpenCV to interpolate the heat-map and project it onto the original image, here I used the code from the Chollet's book:

The original code is written by François Chollet

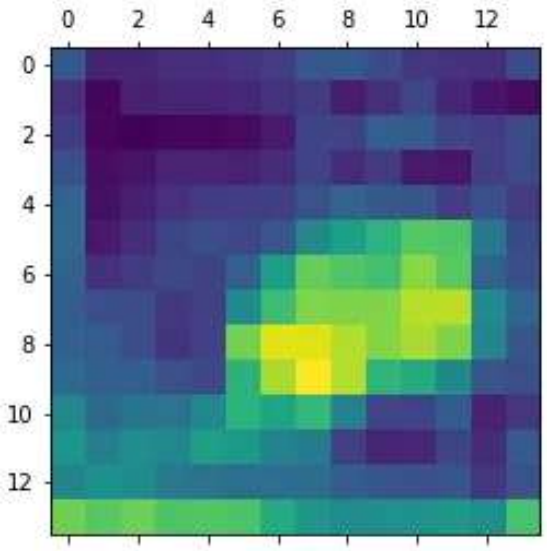
In the image bellow we can see the areas of the image that our VGG19 network took most seriously in deciding which class (‘African_elephant’) to assign to the image. We can assume that the network took the shape of the head and ears of the elephants as a strong sign of the presence of an elephant in the image. What is more interesting, the network also made a distinction between the African elephant and a Tusker Elephant and an Indian Elephant. I am not an elephant expert, but I suppose the shape of ears and tusks is pretty good distinction criterion. In general, this is exactly how a human would approach such a task. An expert would examine the ears and tusk shapes, maybe some other subtle features that could shed light on what kind of elephant it is.

Grad-CAM heat-map projected onto the original elephants image

Ok, let's repeat the same procedure with some other images.



Left: the shark image with projected CAM heat-map (Right).



Another shark image with the corresponding CAM heat-map

The sharks are mostly identified by the mouth/teeth area in the top image and body shape and surrounding water in the bottom image. Pretty cool!

Going Beyond VGG

VGG is a great architecture, however, researchers since came up with newer and more efficient architectures for image classification. In this part we are going to investigate one of such architectures: DenseNet.

There are some issues I came across while trying to implement the Grad-CAM for the densely connected network. First, as I have already mentioned, the pretrained models from the PyTorch model zoo are mostly built with nested blocks. It is a great choice for readability and efficiency; however it raises an issue with the dissection of such nested networks. Notice that VGG is formed with 2 blocks: feature block and the fully connected classifier. DenseNet is made of multiple nested blocks and trying to get to the activation maps of the last convolutional layer is impractical. There are 2 ways we can go around this issue: we can take the last activation map with the corresponding batch normalization layer. This yields pretty good results as we will see shortly. The second thing we could do is to build the DenseNet from scratch and repopulate the weights of the blocks/layers, so we could access the layers directly. The second approach seems too complicated and time consuming, so I avoided it.

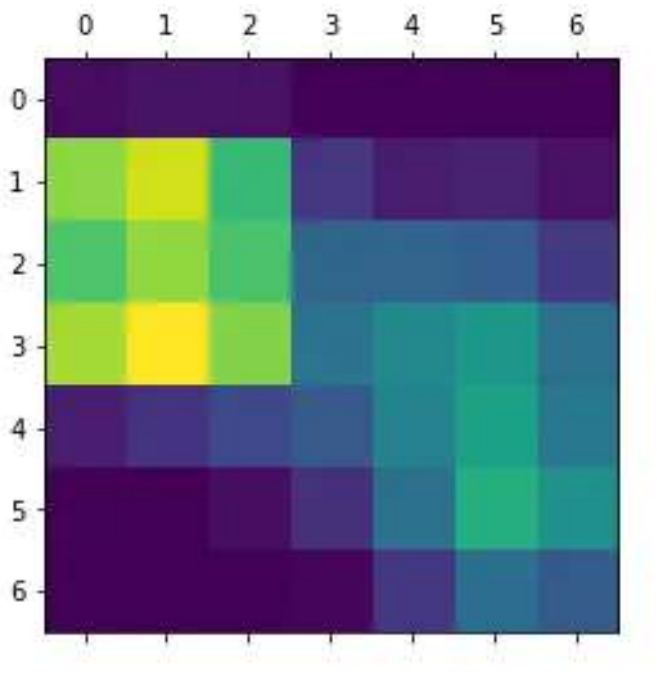
The code for the DenseNet CAM is almost identical to the one I used for the VGG network, the only difference is in the index of the layer (block in the case of the DenseNet) we are going to get our activations from:

It is important to follow the architecture design of the DenseNet, hence I added the global average pooling to the network before the classifier (you can always find these guides in the original papers).

I am going to pass both iguana images through our densely connected network in order to find the class that was assigned to the images:

```
Predicted: [('n01698640', 'American_alligator', 14.080595),
('n03000684', 'chain_saw', 13.87465), ('n01440764', 'tench',
13.023708)]
```

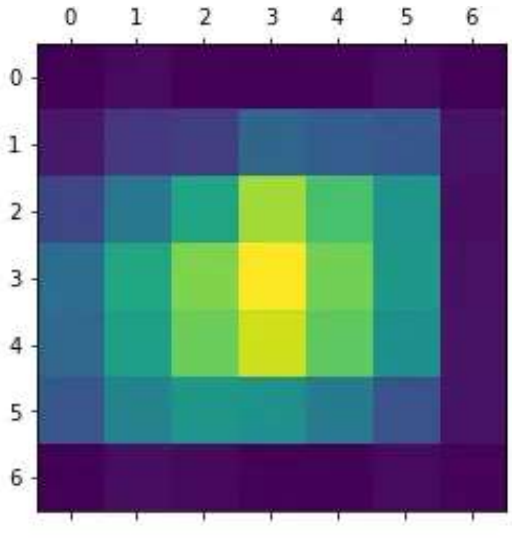
Here, the network predicted that this is the image of an ‘American Alligator’. Hmm, let’s run our Grad-CAM algorithm against the ‘American Alligator’ class. In the images below I show the heat-map and the projection of the heat-map onto the image. We can see that the network mostly looked at the “creature”. It is evident that alligators may look like iguanas since they both share body shape and overall structure.

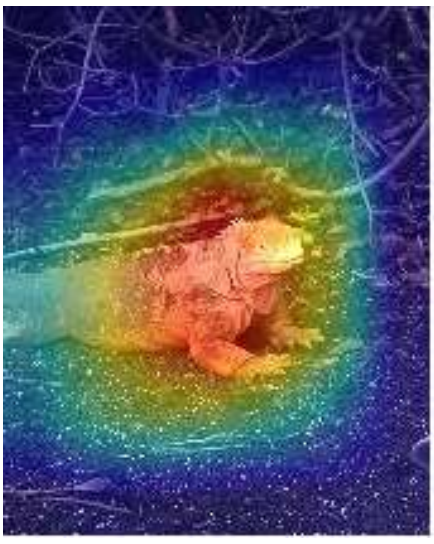


The common iguana was misclassified as an American alligator.

However, notice that there is another part of the image that has influenced the class scores. The photographer in a picture may throw the network off with his position and pose. The model took both the iguana and the human in consideration while making the choice. Let's see what will happen if we crop the photographer out of the image. Here are the top-3 class predictions for the cropped image:

```
Predicted: [('n01677366', 'common_iguana', 13.84251), ('n01644900',
'tailed_frog', 11.90448), ('n01675722', 'banded_gecko', 10.639269)]
```

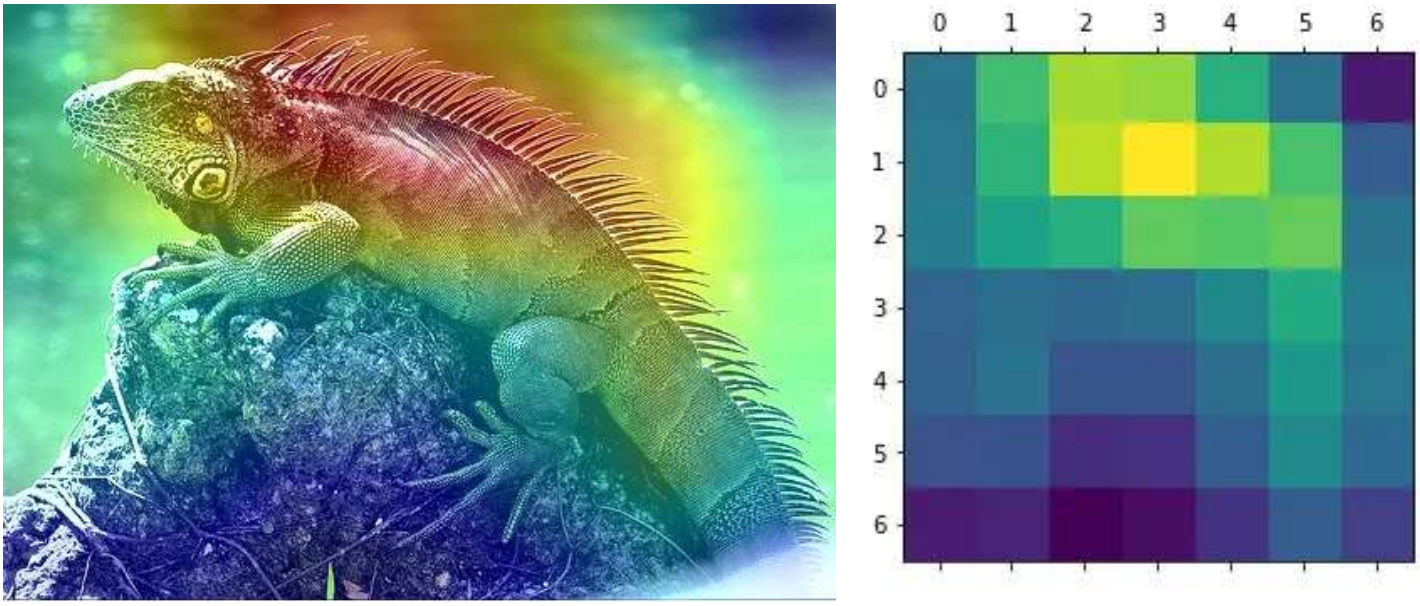




Cropped iguana image is now classified as the common iguana

We now see that cropping the human from the image actually helped to obtain the right class label for the image. **This is one of the best applications of the Grad-CAM:** being able to obtain information of what possibly could go wrong in misclassified images. Once we figure out what could have happened we can efficiently debug the model (in this case cropping the human helped).

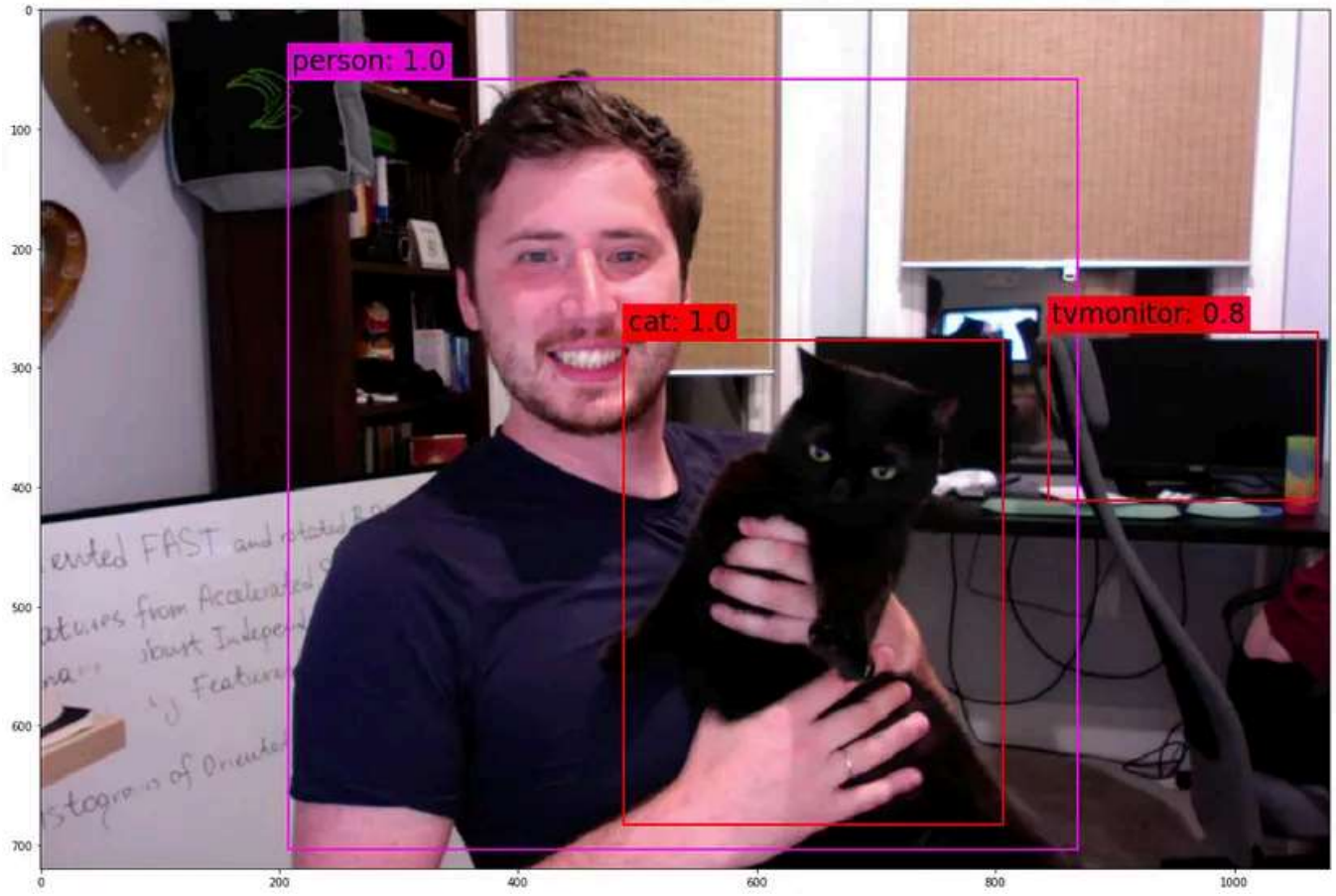
The second iguana was classified correctly and here is the corresponding heat-map and projection.



The second iguana was identified by the pattern of spikes on its back

Going Beyond ImageNet

Let's try some of the images I have downloaded from my Facebook page. I am going to use our DenseNet201 for this purpose.



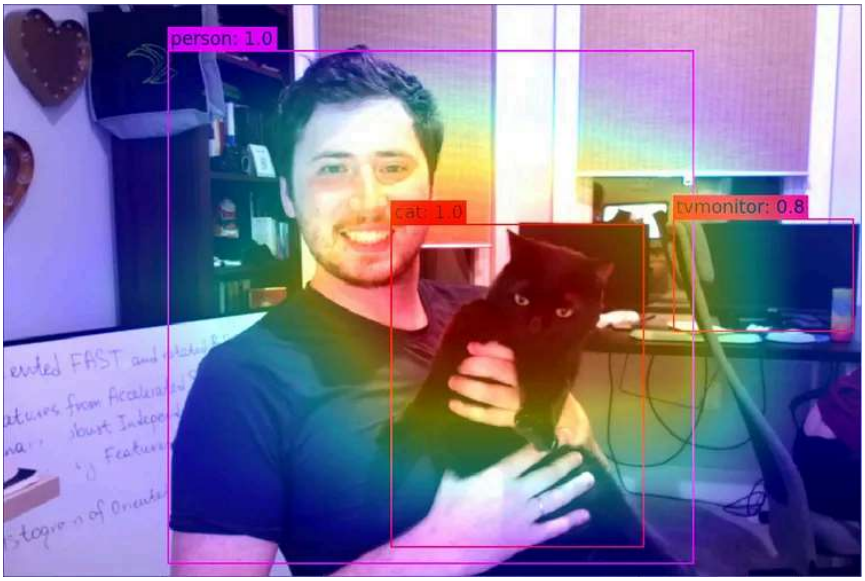
YOLO applied to Luna

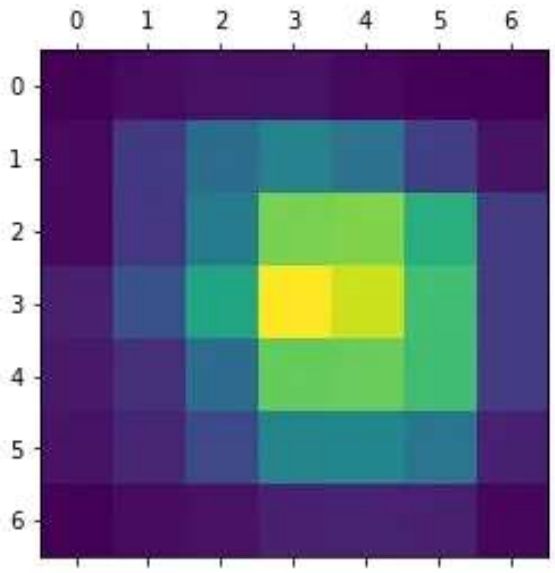
The image of me holding my cat is classified as follows:

```
Predicted: [('n02104365', 'schipperke', 12.584991), ('n02445715',
'skunk', 9.826308), ('n02093256', 'Staffordshire_bullterrier',
8.28862)]
```

Let's look at the class activation map for this image.

In the images below we can see that the model is looking in the right place.





Let's see if cutting myself out will help with the classification.

Cropping myself out helped dramatically:

```
Predicted: [('n02123597', 'Siamese_cat', 6.8055286), ('n02124075',
'Egyptian_cat', 6.7294292), ('n07836838', 'chocolate_sauce',
6.4594917)]
```

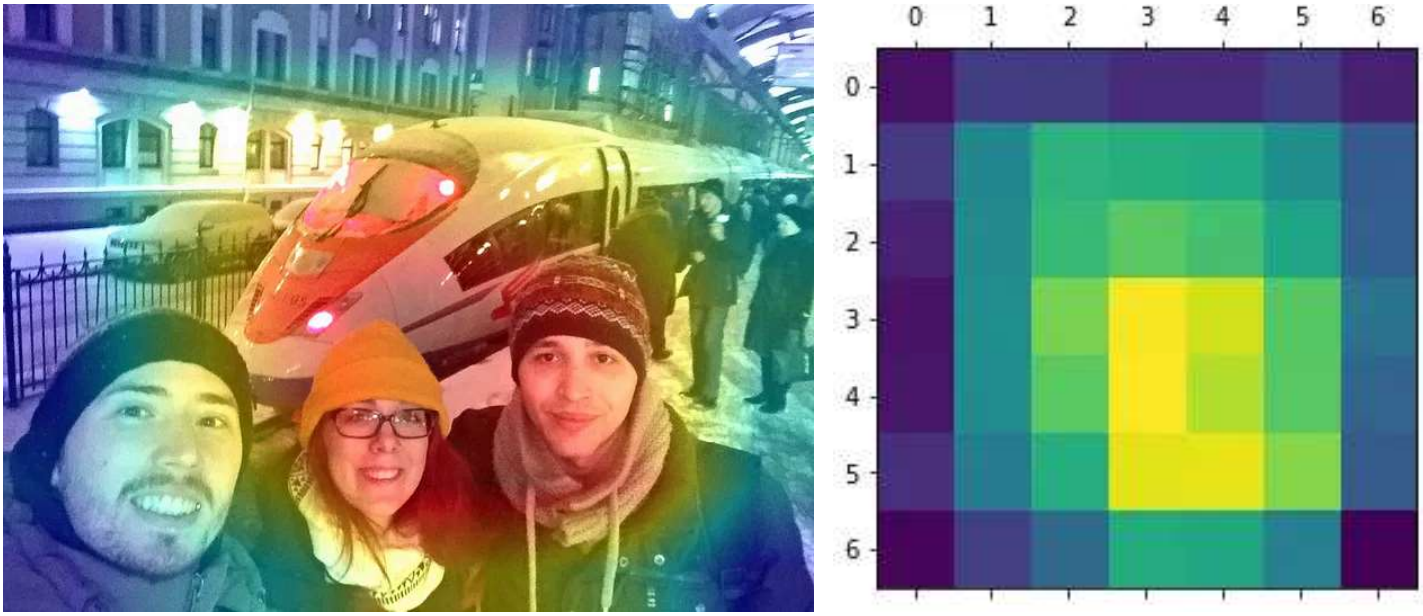
Now Luna is predicted at least as a cat, which is much closer to the real label (which I don't know because I don't know what kind of cat she is).

The last image we are going to look at is the image of me, my wife and my friend taking a bullet train from Moscow to Saint-Petersburg.

The image is classified correctly:

```
Predicted: [('n02917067', 'bullet_train', 10.605988), ('n04037443',
'racer', 9.134802), ('n04228054', 'ski', 9.074459)]
```

We are indeed in front of a bullet train. Let's look at the class activation map just for fun then.



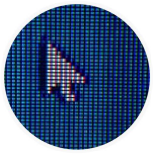
Me and my friends in front of a bullet train in Moscow

It is important to note that the DenseNet's last convolutional layer yields 7x7 spacial activation maps (in contrast to 14x14 in the VGG network), hence the resolution of the heat-map may be a little exaggerated when projected back into the image space (corresponds to the red attention color on our faces).

Another potential question that can arise is why wouldn't we just compute the gradient of the class logit with respect to the input image. Remember that a convolutional neural network works as a feature extractor and deeper layers of the network operate in increasingly abstract spaces. We want to see which of the features actually influenced the model's choice of the class rather than just individual image pixels. That is why it is crucial to take the activation maps of deeper convolutional layers.

I hope you enjoyed this article, thank you for reading.

Machine Learning | Computer Vision | Pytorch | Convolutional Network | Towards Data Science



Follow

Written by Stepan Ulyanin

207 Followers

Software engineer @ Apple

More from Stepan Ulyanin

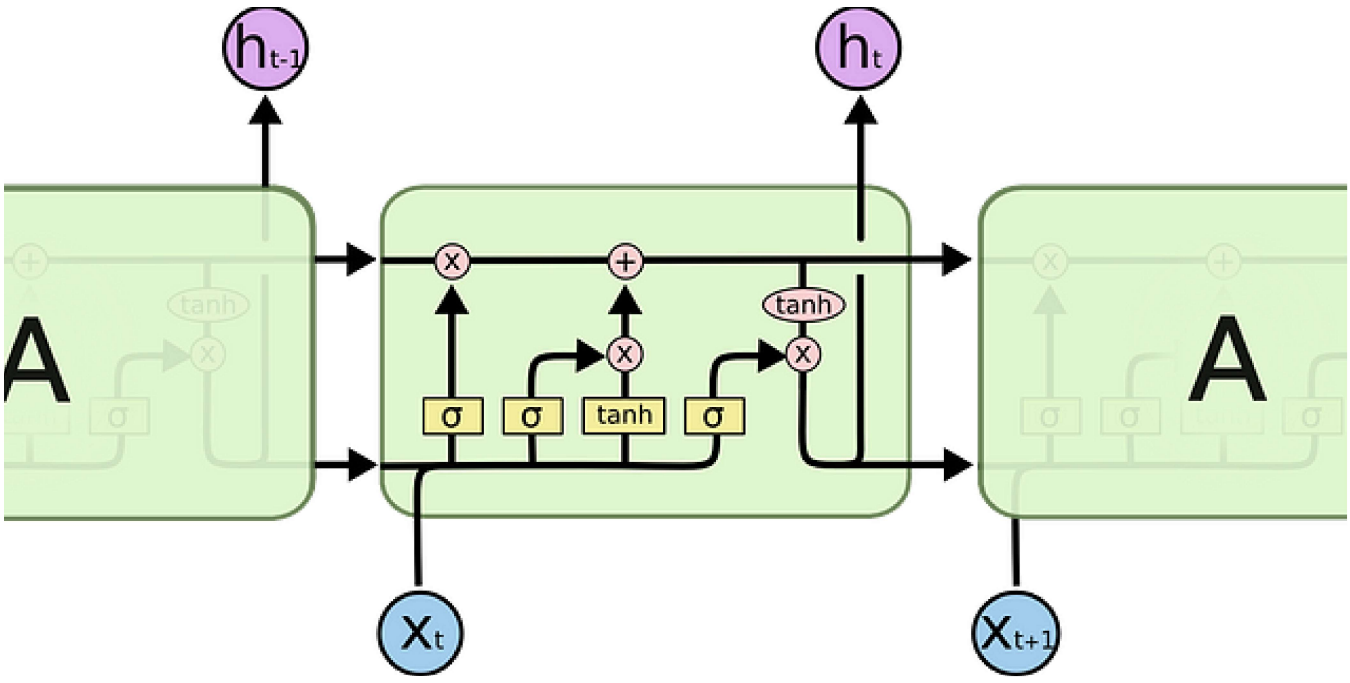
Stepan Ulyanin

Captioning Images with PyTorch

A concise write up of implementation of the Image Captioning model in PyTorch with aftertaste of a tutorial.

10 min read · Jan 25, 2019

580 8



Stepan Ulyanin in Coinmonks

Character-To-Character RNN With Pytorch's LSTMCell

I looked over a few tutorials on Recurrent Neural Networks, using LSTM; however I couldn't find the one that uses the LSTMCellclass, many...

7 min read · Aug 7, 2018

165 1

Stepan Ulyanin

Notes On Deep Learning Theory Part 1: Data-Generating Process

I have come across a wonderful book by Terrence Sejnowski called The Deep Learning Evolution. The book describes the early struggles of...

7 min read · Feb 1, 2019

181

Stepan Ulyanin

DCGAN Adventures with Cifar10

I have been tempted to build a Deep Convolutional Generative Adversarial Network for a while now since I have taken the course on Udacity...

5 min read · Apr 18, 2018

199

See all from Stepan Ulyanin

Recommended from Medium

Hakan Ateşli in Hepsiburada Data Science and Analytics

Explainable AI With SHAP

From Complexity to Clarity: Exploring AI Transparency with SHAP

13 min read · Dec 22, 2023

233

Rabia Gondur

Feature Extraction Using CNNs via PyTorch

In this article, we will explore CNN feature extraction using a popular deep learning library PyTorch. We will go over what is feature...

6 min read · Dec 6, 2023

Lists

Predictive Modeling w/ Python

20 stories · 1049 saves

Practical Guides to Machine Learning

10 stories · 1256 saves

Natural Language Processing

1335 stories · 821 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 350 saves

Shubham Bhandari in Towards AI

Explainable AI: GradCAM using PyTorch

The articles sheds light on Explainable AI. It explains and illustrates GradCAM method using PyTorch on ResNet model.

6 min read · Feb 20, 2024

57 1

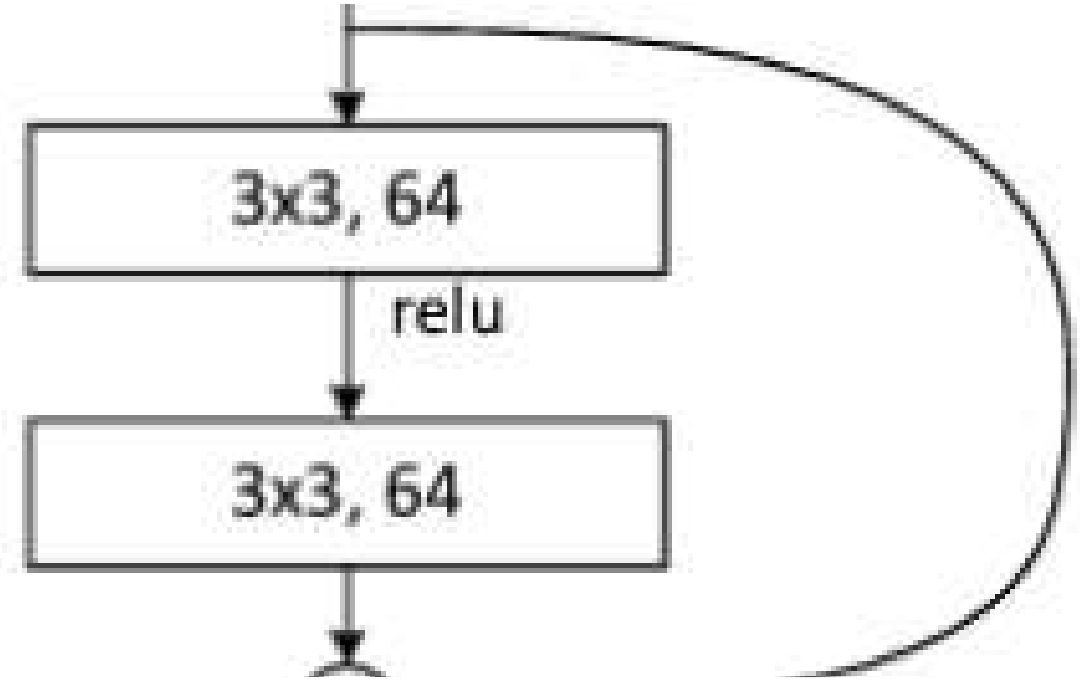
btd

Explainable AI (XAI) Tools and Libraries

Explainable AI (XAI) tools and libraries are essential components for developing, evaluating, and deploying machine learning models with...

· 3 min read · Nov 23, 2023

1 2



Chen-Yu Chang

Building a Customized Residual CNN with PyTorch

Residual networks (ResNets) have revolutionized the field of deep learning by enabling the construction of much deeper networks than was...

3 min read · Nov 5, 2023

Azeem - I

Understanding ResNet Architecture: A Deep Dive into Residual Neural Network

Residual Network is a deep Learning model used for computer vision applications. The ResNet (Residual Neural Network) architecture was...

6 min read · Nov 14, 2023

 101