WIDS- Final Report

IMAGE CAPTIONING USING DEEP LEARNING

Abhiruchi Kotamkar (210070043)

Mentors: Kartikey Anand, Abhinav Anurag

The aim of the project was to develop an image captioning model in python using deep learnings and convolutional neural networks.

In the first week, the project focused on learning different libraries of python necessary for the project like Numpy, Pandas, Data visualization using matplotlib,etc. In the first week I also learned the basics of machine learning and learnt to predict data for tabular data and learnt to work with missing values, categorical variables and cross-validation.

In the second week, I learnt about PyTorch and TorchVision to learn how to make neural networks using this. I also learnt the basics of neural networks and learnt about the MNIST dataset.

In the third week I learnt about Pytorch transforms and learnt how to preprocess images and learnt about convolutional neural networks and deep learning. Here is a summary of the concepts used in the project:

Convolutional Neural Networks

First we start with the convolution layer which consists of the input layer and filter. The input layer has the width, height and depth dimensions (here the depth of a volume must not be confused with the depth of a network). Next the filter which is small spatially but has the sane depth as the input layer.

We now 'convolve' the filter with the image which is basically sliding them through the input volume and taking the dot product along the way($w^T x + b$). If the input layer is 32x32x3 and the filter is 5x5x3, the dot product taken will be 75-dimensional + bias and the activation map will be 28x28x1. There are 28 unique positions in which the filter can be placed when the stride is 1. Now this process is done for multiple filters (set of filters say 6) and the activation maps are stacked together along the depth dimension. This then becomes the input volume for the next convolution layer (28x28x6). If the next convolution layer here is of 10 5x5x6 layers, the new input volume becomes 24x24x10. A problem noticed at this stage is that the input volume is shrinking.

Note: At this point it is important to understand that the convolution in the sequential layers occurs on the new input layer and not the original image.

The three core building blocks should be the convolution layer, the rectifier layer and the pool layer that carries out the pooling operations.

A convolution layer should have a layout that is similar to the following:

$$CONV->RELU->CONV->RELU->POOL->CONV->RELU->CONV->RELU->$$

$$POOL->CONV->RELU->CONV->RELU->POOL->FC$$

The output size for the convolution layer is given by: $N - F/stride + 1$.

It is common to pad the input layer to prevent 'shrinking'. Padding the input layer so that the output size is the same as the input size is called Zero-padding the border. For a filter of size FxF you want the zero-padding to be done with $(F-1)/2$ border.

$$size_{output} = \frac{height_{i/p} + 2 * pad - height_{filter}}{stride} + 1$$

Number of parameters:

Consider the following example:

Input volume: 32x32x3

10 5x5 filters with stride 1, pad 2.

Each filter has 5*5*3 + 1 = 76 params

Therefore # of params = 76*10 = 760.

The convolution layer takes 4 hyperparameters (Number of filters K, the spatial extent F, the stride S and the amount of zero-padding P). The depth is the number of filters. K is usually taken in the powers of 2 and F is usually odd.

Question: How does a 1x1 filter make sense?

Answer: Consider an input layer of 56x56x64 and 32 1x1 filter layers. Each 1x1 filter will do a 64-dimentional dot product with the input cause the depth of the input is 64.

Question: Why is the size of F always odd?
Answer: You can do 2x2 filters but odd filters make sense cause they will always have equal on both sides, makes sense cause you apply the filter around a specific point.

Question: Are we always working with squares?
Answer: yes.

The brain/neuron view of CONV layer:

The filter at a certain position is interpreted as a neuron that is fixed in space at its center that takes the dot product in the FxF space around it. The receptive field is the FxF space around them and all of them share the same parameters.

Pooling Layers:

The conv layers preserve the size of the input layer the pool layer takes care of reducing the size and making them more manageable. It operates on each activation map independently.

Max pooling- takes max of the corresponding sectioned sectioned single depth slice.

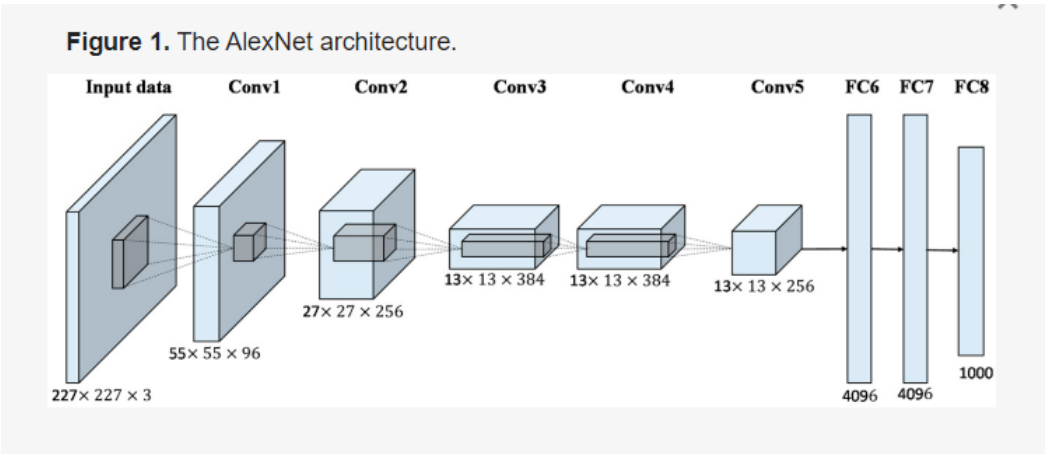
It requires three hyperparameters: the spatial extent F and stride S. Output depth is preserved.

Produces volume of size:

$$W_2 = \frac{W_1 - F}{S} + 1$$

$$H_2 = \frac{H_1 - F}{S} + 1$$

AlexNet Architecture:



For the Conv3, Conv4 and Conv5 layers the size has reduced to 13x13 and so we keep it for a few steps at the same size.

Localization and Detection

Task to be completed: Classification + Localization

Cat

Classification just tells you the image is of a cat Localization puts a box around the image of the cat (x, y, w, h).

Localization as Regression:

Localization can just be framed as a localization problem. An image is coming in and that image goes through some processing and eventually we produce 4 real valued numbers that parametrize the image.

When the ground parameters are given we compute the loss between the output and the ground parameters, back propagate and update the network.

Simple Recipe for Classification + Localization:

1. Train a classification model.
2. Attach new fully-connected “regression head” to the network.

3. Train the regression head only with SGD and L2 loss.
4. At test time use both heads.

Per-class regression generates Cx4 numbers (one box per class) and class agnostic regression outputs 4 numbers (one box).

Human Pose Estimation: Using a zoomed in image of the human the pose of the human can be predicted. A person has say K joints, we run the image through a convolution network and regress (x , y) coordinates for each of the joints.

Sliding Window

- The idea becomes that you run the classification-localization dual headed network at multiple positions of the image and then aggregate across those positions.

Overfeat Architecture: It basically starts off with an alexnet and then a classification head and a regression head. The classification head outputs Class scores and the regression head outputs the boxes. Because it is an alexnet architecture the input expected is 221x221 but actually we can use larger images say 257x257. For this you can run it on the top corner of the larger image and obtain the class scores and later on all 4 corners of the image. We end up with 4 boxes from each of the corners together with the classification scores. Now to arrive at a single box some heuristics are used.

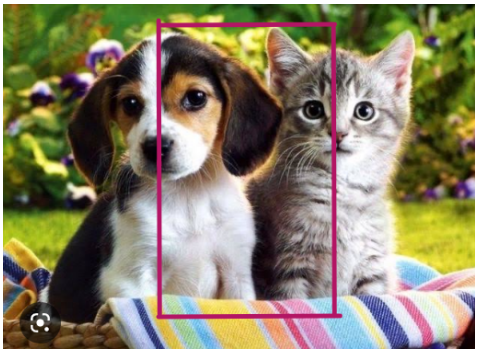
In practice we use many sliding window locations and multiple scales (not just 4). One problem at this stage is low efficiency and higher cost.

A FC is just a vector of 4096 numbers. We can think of it as jut another convolution map (4096x1x1). We can convert the fully connected layer to a convolutional layer (turns into a 5x5 convolution). Now the network consists only of convolutional layers and pooling operations.

Detection as Regression? : It is kind of hard to treat detection as regression because of the problem of variable sized outputs.

Detection as Classification:

Cat? - Yes Dog? - No



Cat? - No Dog? -No

So we run a classifier at different regions of the image and then this will solve our variable output problem. Now we need to try windows of different sizes and at different positions but this is expensive.

Region Proposals: Really fast class agnostic object detector. Not very accurate but give the results quickly. They Find "blobby" image regions that are likely to contain objects. The most popular region proposal method is Selective Search.

RCNN: Region based CNN

We have an input image on which we run a region proposal method such as Selective search to get about 2000 boxes. We crop and warp the image region for each of the boxes and run it through a CNN to classify.

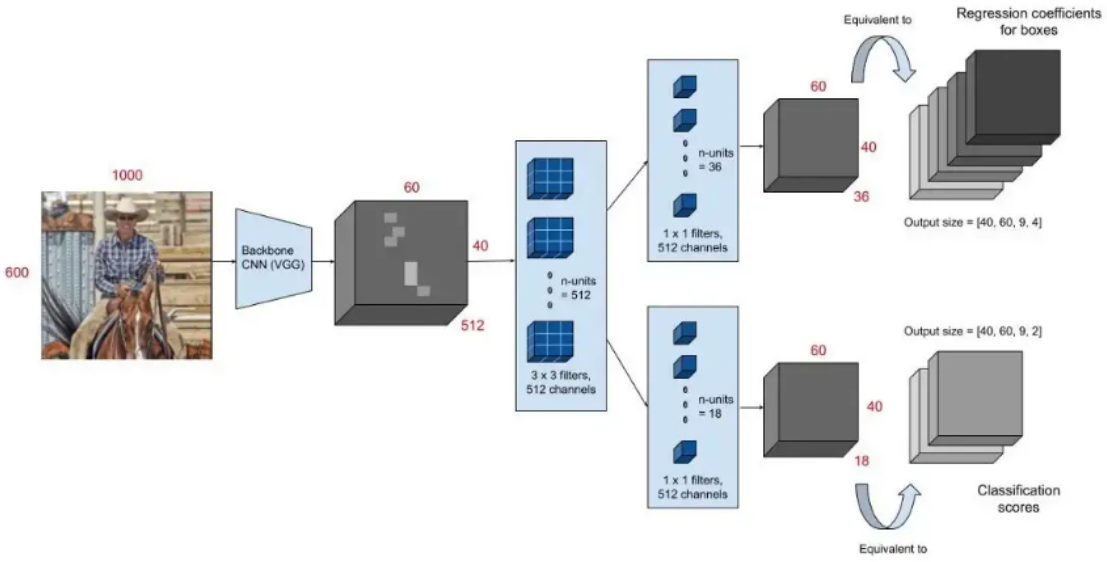
R-CNN Training:

1. Train a classification model for ImageNet.
2. Fine-tune the model for detection.
3. Extract features.
4. Train one binary SVM per class to classify region features.
5. Bbox regression.

Problems for R-CNN : It is pretty slow and has a larger test time. It also has a complicated pipeline.

Fast R-CNN:

The solution to the problems of R-CNN is solved by just swapping the order of extracting regions and running the CNN. The problem of a complex training pipeline is solved by just training the whole system end-to-end at once!



Fast R-CNN

Visualization, Deep Dream, Neural Style, Adversarial Examples

We first looked into understanding ConvNets by visualizing patches that maximally activate neurons and also by visualizing weights. For visualizing weights we look at the first Conv layer which consists of a filter bank that we slide over the image. It only makes sense to use this method on the first layer as the weights in the above layers are not very interpretable.

Another way to study ConvNets is by visualizing the representation. The FC7 layer is a 4096 dimensional “code” for an image. t-SNE visualization- two images are placed nearby if their CNN codes are close.

Visualizing Activations: The professor at this point shows a video that shows 2 separate kinds of visualizations for neurons 1) Deconv and 2) optimization based. The video focused on what happens inside of neural nets. It uses the very popular AlexNet convolution network. In the first layer one neuron responds strongly to light to dark edges, its neighbour responds strongly to the opposite. By the time we get to the 5th conv layer the features being computed represent an abstract concept.

Deconv approaches:

Question: Suppose we feed an image into the net how can we compute the gradient of any arbitrary neuron in the network?

Answer: We forward until some layer and then we have our activations at that layer, if we are interested in some specific neuron we'll zero out all the gradients in that layer except for the specific neurons gradient. Then we backprop the image.

The input image goes through a series of layers and we get an activation feature map at one point. On backprop we get the reconstructed image. We are going to meddle with the ReLu layer in the backprop. For a particular forward pass the ReLu operation thresholds the negative activations to 0. For the backward pass the ReLu layer finds what the gradient from above is and blocks all the gradients for all the parts that had negative activations. The ReLu neuron acts as a switch.

In guided backpropagation we backprop only the parts that had positive gradients. We end up with very clear images.

Optimization to Image:

Question: Can we find an image that maximizes some class score?

Answer: We start out with a zero-image, we feed it into a conv-net and get some scores out, then we set the gradient of the scores vector to be [0,0….1,…..0], then backprop to image. Then we do a small “image update” and then forward the image through the network.

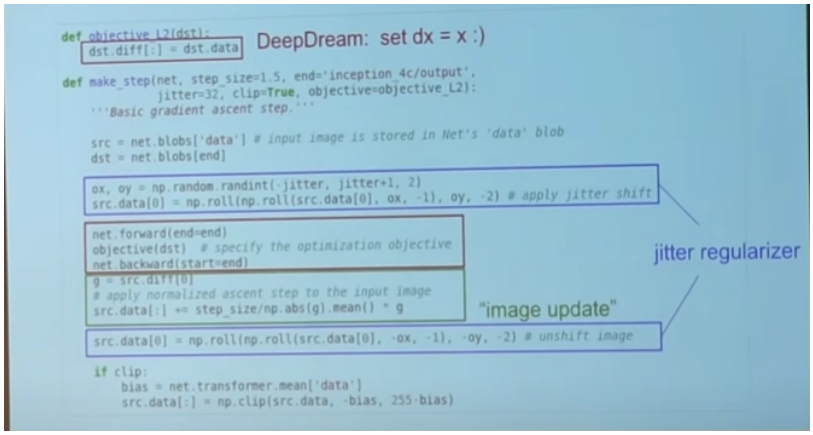
$$argmax S_c(I) - \lambda ||I||_2^2$$

A different form of regularizing the image:

Update the image x with gradient from some unit of interest and then blur x a bit and then take any pixel with small norm to 0.

Deep Dream

The make_step function is called repeatedly as we optimize the image. The variable end is a string denoting exactly what layer we want to dream at. This is an inception network and one of the inception networks is called 4c.



Using net.forward we are forwarding the network then we are calling the objective on dst which is the layer we want to dream in. The blobs have a diff field and a data field.The data field holds the raw activations and the diff field golds the gradients.

You are amplifying the features that are maximally activated in the network.

Neural Style

A painting in the style of the starry night by Vincent Van Gogh

Neural style transfer(**NST**) refers to a class of software algorithms that manipulate digital images, or videos, in order to adopt the appearance or visual style of another image.

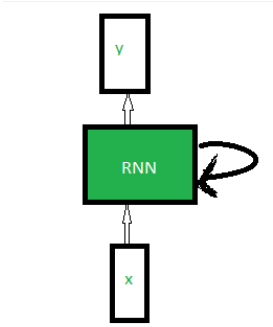
Here we have a content image and a style image. We want to transfer that style onto the content image. We pass both through a ConvNet. While passing through the content image we recall all the raw activations in the ConvNet. We call these raw activations the content of the image.

Recurrent neural Networks

Recurrent neural networks offer a lot of flexibility in how you wire up the neural network architectures. We can operate on sequences, sequences at the input, output or both at the same time. In image captioning we input a fixed size image and through the recurrent neural networks we get a sequence of words as the output. For Sentiment Classification we input a sequence of words and the output is the sentiment of the sentence- whether the sentence is positive or negative.

Even if there aren't sequences at the inputs or outputs we can use recurrent neural networks because you can process your fixed size inputs or outputs sequentially.

The RNN has a state and it receives input vectors through time and we can base some predictions on to[p of it.



We can process a sequence of vectors x by applying a recurrence formula at every step:

$$h_t = f_W(h_{t-1}, x_t)$$

h_t: new state.

x_t: input vector.

f_w: some function with parameters W.

The same function is used at every time step.

Character-level language model example:

We feed a sequence of characters into the neural network and at every single time step we ask the neural net to predict the character in the sequence.

At this point the professor takes us through the code for the character level language model.

Image Captioning

The full network is just made of Convolutional Neural net and a recurrent neural net. We are conditioning the RNN by the output of the CNN. We place the image in the CNN and then begin the generation of the RNN with a special start vector.

We then compute the recurrence using the formula:

$$h = tanh(Wxh * x + Whh * h + Wih * v)$$

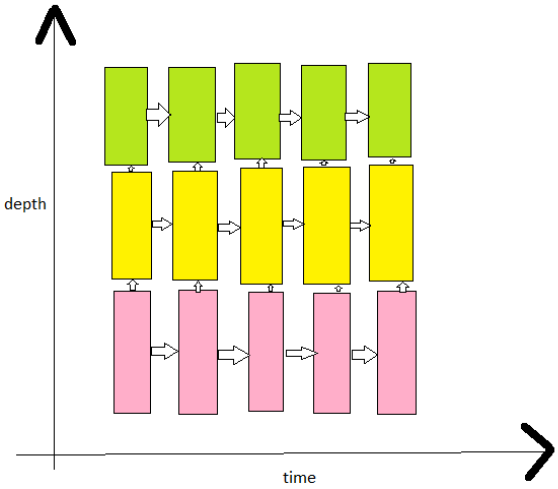
At the very first step the y0 vector is the distribution over the first word of the sequence. The RNN from now on has to juggle 2 tasks- predict the next word in the sequence in this case and it has to remember the image information. Then we continue sampling and predicting and in the end we ample an <END> token which correspond to the period (.) in the sentence. Every word has 300 numbers associated with it and the embeddings are 300 dimensional regardless of the image.

In the training data the correct sequence that we expect from the RNN is the first word, second word, third word, end. Every training data has the end token in it. When we train this model we train it on Image Sentence Datasets, an example of which is Microsoft COCO which has roughly 120K images with 5 sentences each.

Preview of fancier architectures:

In the current model we only feed the image once at the beginning and one way to play with this is to allow the RNN to look back to the image and reference parts of the image while its describing the words.

If we want to make RNNS more complex one of the ways of doing so is to stack them up in layers. One of the ways is given below:



The input for one RNN is the hidden vector of the other RNN.

$$h_t^l = tanhW^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

The following weeks were devoted to the final project.

Final Project

We are using the Flickr_8k dataset to develop the image captioning model.

We first started by preprocessing the data and defined the function that took in the images folder and saved the features extracted in the features.pkl file.

We are defined the model as VGG16 (VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION) and then we removed the last layer from the loaded model, as this is the model used to predict a classification for a photo. We also reshaped the loaded photo into the preferred size of the model. The function returns a dictionary of image identifier to image features.

```
In [3]: from PIL import Image
def extract_features(directory):
#load the model
    model = VGG16()
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    print(model.summary())
    features = dict()
    for name in listdir(directory):
        filename = directory + '/'+ name
        image = load_img(filename, target_size=(224,224))
        image = img_to_array(image)
        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[
        image = preprocess_input(image)
        feature = model.predict(image, verbose=0)
        image_id = name.split('.')[0]
        features[image_id] = feature
        print('>%s'% name)
    return features

directory = 'flickr8k/images'
features = extract_features(directory)
print('extracted features: %d' % len(features))
dump(features, open('features.pkl','wb'))
```

Preparing the Text Data:

First we load the captions.txt document using a function.

A function *load_descriptions() is defined that* given the loaded document text, will return a dictionary of photo identifiers to descriptions. Each photo identifier maps to a list of one or more textual descriptions.

Next the *clean_descriptions()* function: given the dictionary of image identifiers to descriptions, steps through each description and cleans the text.

```
In [19]: def load_clean_descriptions(filename, dataset):
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        tokens = line.split()
        image_id, image_desc = tokens[0], tokens[1:]
        if image_id in dataset:
            if image_id not in descriptions:
                descriptions[image_id] = list()
                desc = 'startseq'+' '.join(image_desc)+' endseq'
                descriptions[image_id].append(desc)
    return descriptions
```

We move on to summarizing the size of the vocabulary.

We can save the dictionary of image identifiers and descriptions to a new file named *descriptions.txt*, with one image identifier and description per line.

Develop Deep Learning Model:

We start by loading the data. We must first load the prepared photo and text data so that we can use it to fit the model. The train and development dataset have been predefined in the *Flickr_8k.trainImages.txt*and *Flickr_8k.devImages.txt* files respectively, that both contain lists of photo file names.

The load_set() function will load a pre-defined set of identifiers given a filename.

In [18]:
```
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    for line in doc.split('\n'):
        if len(line) < 1:
            continue
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)
```

In [19]:
```
def load_clean_descriptions(filename, dataset):
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        tokens = line.split()
        image_id, image_desc = tokens[0], tokens[1:]
        if image_id in dataset:
            if image_id not in descriptions:
                descriptions[image_id] = list()
                desc = 'startseq'+' '.join(image_desc)+' endseq'
                descriptions[image_id].append(desc)
    return descriptions
```

We use ‘startseq’ and ‘endseq’, to kick-off the generation process and to signal the end of the caption.

The *Tokenizer* class that can learn this mapping from the loaded description data. The *to_lines()* to convert the dictionary of descriptions into a list of strings and the *create_tokenizer()* function that will fit a Tokenizer given the loaded photo description text.

In [23]:
```
from keras.preprocessing.text import Tokenizer
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)

def Max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)
```

Defining the Model:

The model has three parts the Photo feature extractor, The Sequence Processor and the Decoder.

In [17]:
```
def define_model(vocab_size, max_length):
 # feature extractor model
 inputs1 = Input(shape=(1000,))
 fe1 = Dropout(0.5)(inputs1)
 fe2 = Dense(47, activation='relu')(fe1)
 # sequence model
 inputs2 = Input(shape=(max_length,))
 se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
 se2 = Dropout(0.5)(se1)
 se3 = LSTM(47)(se2)
 # decoder model
 decoder1 = add([fe2, se3])
 decoder2 = Dense(47, activation='relu')(decoder1)
 outputs = Dense(vocab_size, activation='softmax')(decoder2)
 # tie it together [image, seq] [word]
 model = Model(inputs=[inputs1, inputs2], outputs=outputs)
 model.compile(loss='categorical_crossentropy', optimizer='adam')
 # summarize model
 print(model.summary())
 plot_model(model, to_file='model.png', show_shapes=True)
 return model
```

The following layers are present in the CNN model:

```
==================================================================================================
 input_4 (InputLayer)           [(None, 29)]         0           []

 input_3 (InputLayer)           [(None, 1000)]       0           []

 embedding_1 (Embedding)        (None, 29, 256)      1070592     ['input_4[0]
[0]']

 dropout_2 (Dropout)            (None, 1000)         0           ['input_3[0]
[0]']

 dropout_3 (Dropout)            (None, 29, 256)      0           ['embedding_1
[0][0]']

 dense_3 (Dense)                (None, 47)           47047       ['dropout_2[0]
[0]']

 lstm_1 (LSTM)                  (None, 47)           57152       ['dropout_3[0]
[0]']

 add_1 (Add)                    (None, 47)           0           ['dense_3[0]
[0]',
                                                                  'lstm_1[0]
[0]']

 dense_4 (Dense)                (None, 47)           2256        ['add_1[0]
[0]']

 dense_5 (Dense)                (None, 4182)         200736      ['dense_4[0]
[0]']
```