Positional Encoding (homework)

Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains

Matthew Tancik[1*] Pratul P. Srinivasan[1,2*] Ben Mildenhall[1*] Sara Fridovich-Keil[1]

Nithin Raghavan[1] Utkarsh Singhal[1] Ravi Ramamoorthi[3] Jonathan T. Barron[2] Ren Ng[1]

[1]University of California, Berkeley [2]Google Research [3]University of California, San Diego

Abstract

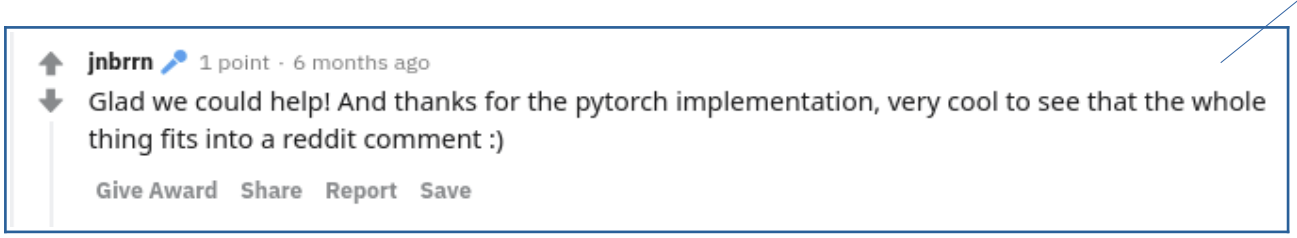
We show that passing input points through a simple Fourier feature mapping enables a multilayer perceptron (MLP) to learn high-frequency functions in low-dimensional problem domains. These results shed light on recent advances in computer vision and graphics that achieve state-of-the-art results by using MLPs to represent complex 3D objects and scenes. Using tools from the neural tangent kernel (NTK) literature, we show that a standard MLP fails to learn high frequencies both in theory and in practice. To overcome this spectral bias, we use a Fourier feature mapping to transform the effective NTK into a stationary kernel with a tunable bandwidth. We suggest an approach for selecting problem-specific Fourier features that greatly improves the performance of MLPs for low-dimensional regression tasks relevant to the computer vision and graphics communities.

Source: https://arxiv.org/pdf/2006.10739.pdf

Introduction

The 'positional encoding' trick has surprised many researchers in the Deep Learning community. It has already been used in multiple papers but the intuition behind it is not always immediately clear. So, in the spirit of Richard Feynmans "*What I cannot create, I do not understand*", we are going to reproduce some results from the paper "*Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains*" in the hope to better understand how positional encodings help in their research.

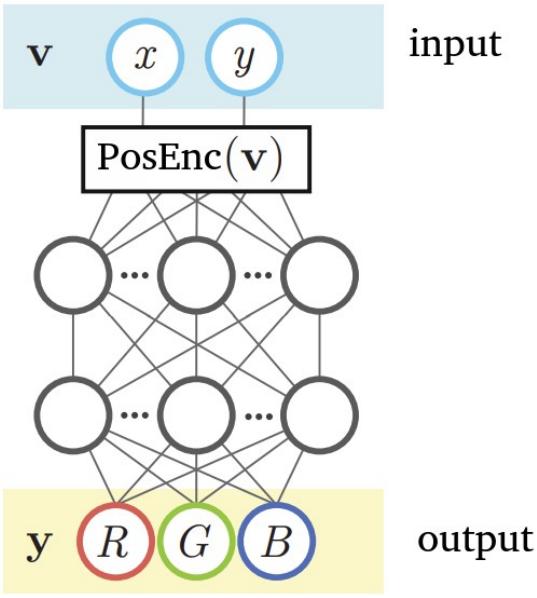
I choose this specific paper since recreating their work only requires a few lines of code (one of the authors agreed with me, see reaction below), the results can be visually inspected, it requires short training time (which allows for hyper-parameter exploration), an individual image/photo is already a complete dataset, we can practice with tensor operations, we can play with Fourier series, and we don't have to worry about over-fitting!



Papers goal

You can read the paper if you wish, but this is not necessary to complete the assignments.

One thing the paper explores is 2D image regression. In short, they train a whole neural network on a single image, and once trained, the neural network can recreate that image. Unlike other neural architectures (like auto-encoders) the pixel color values are not given as input but must be predicted by the network given the position of that pixel. The neural network learns a mapping from pixel-position to color. The neural architecture can be seen in the image on the right. It accepts two inputs (x and y coordinates), uses one or more fully connected hidden layers, and outputs three values (R G B = the color). At the bottom of this page you can find an example of an input with corresponding output which should clarify it a bit. There the input and output values are given for the highlighted pixel. For better performance, we will always normalize all values to values between 0 and 1. This means that the **y** and **x** will be divided by the image height and width respectively, and the RGB values by the maximum color value (255). The normalization of the coordinates gives this network unique properties which we will explore in the assignments. By forcing the network to predict the colors of an image given only the (normalized) coordinates as cue, the network has to learn to represent it as a smooth continuous mathematical function (instead of a set of individual pixels, as images are represented normally). With this implicit representation of the image, it has now become independent the number of pixels. You can therefore save this image in any size you want!

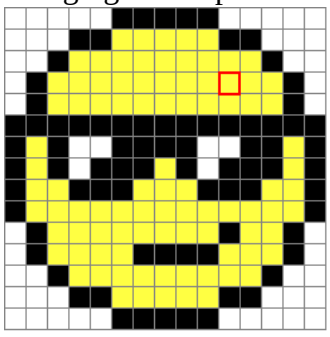


If you would add a third coordinate, this trick can also work for representing 3D objects (x, y, z) and even short videos (x, y, frame_nmb). But that is beyond the scope of what we are going to do here.

Positional encoding trick

In theory this all sounds quite doable, after all, neural networks are known to be general function approximators. But in practice when using just a standard MLP, the results are very disappointing. The 'positional encoding' trick changed this, it made the training time go down from weeks to mere seconds while also requiring far less neurons!

The main difference with earlier work is that the authors of the paper introduced a 'positional encoding block' between the input ((**x,y**) coordinates) and the neural network. This block contains multiple (predefined by a hyper-parameter) **2 dimensional sine** and **cosine** functions with predefined wavelengths (also hyper-parameter). Using the coordinates as input for the sin/cos functions, the output of all these waves are fed to the neural network. What the neural network basically learns is to sample the wavelengths by adjusting how important each wave is (basically changing the amplitudes of the waves). And since we are using a fully connected multi-layer neural network, each neuron in the first layer can learn to use a different set of waves! Basically the 'positional encoding block' allows the image to be described as multiple (limited) fourier series. And given that we learned during the AF2 tutorial that almost all functions can be approximated with fouries series, learning to represent an image should also not be that far fetched.



(y,x) = (4,11)
(R,G,B) = 255,255,66

And now the 'positional encoding block' details

Below you can find a screenshot of the formula to calculate the 'encodings' given the pixel coordinates. It takes the input coordinate/vector **v** and runs the values over multiple 2D sines and cosines with random wavelengths and returning *n* predefined number of outputs. The vector **v** is just the normalized **x** and **y** position of a pixel (normalized → values are between **0** and **1**). Since we can have only a limited number of waves, and some wavelengths are better suited than others for image data, we have to sample wavelengths from a wavelength distribution we think fits well for image representation. The authors of the paper choose to sample wavelengths form a normal distribution with mean 0 and a hyper-parameter standard deviation which they called the **scale**. By modifying the **scale** we can sample more smaller or larger wavelengths. The values are sampled only once during initializing and stored in matrix **B** (see formula below). The shape of **B** is just the number of inputs (**2** in our case, since **x** & **y**) times the number of desired encodings (a hyper-parameter). For example, to generate 128 sin/cos encodings the shape of **B** would be: (2, 128). Note that these sampled values are not optimized during training, only the neural network parameters are updates.

This Fourier feature mapping is very simple. For an input point **v** (for the example above, (x, y) pixel coordinates) and a random Gaussian matrix **B**, where each entry is drawn independently from a normal distribution $N(0, \sigma^2)$, we use

$$\gamma(\mathbf{v}) = [\cos(2\pi\mathbf{B}\mathbf{v}), \sin(2\pi\mathbf{B}\mathbf{v})]^{\mathrm{T}}$$

to map input coordinates into a higher dimensional feature space before passing them through the network.

Assignment 1, data data data

A single image will always be the complete train dataset with the pixel coordinates as input and the RGB values as labels. Lets first make the functions to create the dataset.

Assignment 1a, creating the input values

Since we are using x and y coordinates as input for the positional encoding block, we need to have the (normalized) x and y coordinates for each pixel in the image.
To get these, finish the code below:

```
def createCoordinates(imageShape):
    # x = create a tensor with same width and height as the image
    #     for every position fill in the normalized x coordinate
    #     belonging to that pixel location
    # y = create a tensor with same width and height as the image
    #     for every position fill in the normalized y coordinate
    #     belonging to that pixel location
    x = x.unsqueeze(0)
    y = y.unsqueeze(0)
    coordinates = torch.cat((x,y)).permute(1,2,0).view(-1,2)
    return coordinates
```

Note that with this function you can create the coordinate vectors for any resolution image!

Assignment 1b, creating the labels

Finish code below. This function creates your complete dataset given the location of an image file as input.

```
def photo2dataset(imageLocation):
    # Read image -> use imageio.imread
    # Normalize values to 0-1 range
    # Get coordinates using 'createCoordinates'
    # Make sure the labels have the shape -> (nmb_pixels, 3)
    # return the input, labels and imageshape
```

Assignment 2, creating the positional encoding block

Assignment 2a, creating the 'Fourier' module
We will design the positional encoding block as a torch module. Using the description and formula given before, please finish the code below. In the __init__ function initialize the **B** matrix, the 'dimension' input is the number of wavelengths you sample (self.B = ….). The forward function takes the coordinates tensor **v** as input (shape: number_of_pixels x 2) and uses the **B** and the standard torch sine and cosine functions to calculate the outputs (shape: number_of_pixels x number_of_waves*2).
Good luck!

```
import torch.nn.functional as F
import torch.nn as nn
import torch

class FourierModule(nn.Module):
    def __init__(self, dimension, scale):
        super(FourierModule, self).__init__()
        # insert code here
    def forward(self, v):
        # insert code here
```

Assignment 2b, two dimensional sine and cosine?
We are all familiar with the basic sine and cosine functions, but what do their two-dimensional versions look like?
1) Create a Fourier module with dimension=25 and scale=10.
2) Create the coordinate input tensor for an image of shape (100 x 100) that can be used as input for the module using the 'createCoordinates' funtion (we do not need labels for this part).
3) Run the coordinate tensor through the module. Each column of the output tensor is the output of a single sin/cos function with unique wavelength.
4) Save each column as an image of 100x100 pixels using the imageio.imwrite function.
5) Do the same for Fourier modules with scale=1 and dimension=100.

Question: How does the scale parameter affect the outputs?

Assignment 3, the fun part!

Assignment 3a, real images!
Lets create the whole model. Please transform the pseudo-code below into actual pytorch code. Use 256 neurons for each of the hidden layers.

```
neuralNetwork -> InputShape: (n, 2)

                 FourierModule,
                 LinearLayer, ReLU
                 LinearLayer, ReLU
                 LinearLayer, Sigmoid

                 OutputShape: (n, 3)
```

Assignment 3b
Create a train and test function to train and test your model. Use a standard Adam optimizer. Although you can probably dream the train function by now, please finish the code below:

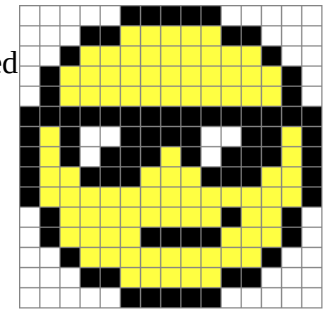
```
def trainOneEpoch(positions, labels, miniBatchSize):
    # Random shuffle the trainset
    # for miniBatch in trainSet:
    #       predictedLabels = neuralNetwork( miniBatch )
    #       loss = binaryCrossEntropy (predictedLabels, labels)
    #       loss.backward()
    #       apply gradients
    #       clear gradients
```

The test function takes as input: the model, a coordinate input tensor, the image shape. I t does not return anything but saves the output of the model as an image file (again with imageio.imwrite).
Tip: use torch.no_grad() to save memory during testing.

Assignment 3c, train and test on an image
Lets see how well our model performs. On our DL-tutorial github you can find the image 'img1.jpeg' to test on. Train a model for 50 epochs using a positional encoding module with dimension=256 and scale=25 and use the test function to see how well the model performs.
Question: Did the model perform well?

Assignment 3d, playing with the scale hyper-parameter
Train a new model on the same picture but this time with scale=1.
Question: What is the major difference in output when using a lower scale?
Also train a new model with scale = 300
Question: What is the major difference in output when using a higher scale?

Assignment 3e, change size
As mentioned earlier, the model learns to represent an image as a smooth continuous mathematical function. Once trained, the same model can be used to output the learned image in any desired resolution. Lets take an extreme example to illustrate this effect. The image on the right (on github: 'img2.png') has a shape of only 15 x 15 pixels. We will use this image to train our model on. (training will go much faster here!)



1) Train a model on this image (dimension=128, scale=1) for 500 epochs.
2) Save a test image with an resolution of 500 x 500 pixels.
Question: does it look like what you expected?
3) do the same for scale=0.5 and scale=2 and scale=10
Question: Which of the four model outputs do you prefer?

Assignment 3f, the power of the positional encoding trick
For this assignment choose any photo/image you like.
1) Train a model on this image. You might need to play with the 'scale' hyper-parameter.
2) Train a model without the 'positional encoding block' by directly giving the coordinates as input to the neural network.
3) Be amazed how well this simple 'position encoding trick' improves the performance.
4) Share the results with the group!