Marked assignment 2

Setup You can **choose** whether you want to work on Option 1 or Option 2. Choose either of the two. Option 1 would be suited for those more interested in theory and not too much coding. Option 2 would be more suited for those interested in a more engineering-like assignment and who are prepared to spend more time running their codes.

As before, fix the random seed to your CID number.

Hand-in format This is the same as before. You will hand in the .ipynb file via the Blackboard (see Assessments and Mark Schemes folder and then upload it into 'Coursework 2 Drop Box Spring 23'). Optional is to have the Google Colab link inside the .ipynb. Make sure to save your notebook such that you show your results (i.e. I'll need to be able to check the values you report in your results).

Option 1: Neural Network Gaussian Processes

Part I: Proving the relationship between a Gaussian process and a neural network [10pt]

Setup We will follow here a more recent line of work that has made the connection between deep neural networks and Gaussian processes. Initially, the connection was made in [Neal, 1996] for neural networks with a single hidden layer. More recently this has been expanded upon by for example [Matthews et al., 2018, Lee et al., 2017]. Even more recently, the connection has been made that also the *training* process of a neural network under certain settings can be characterised through a kernel [Jacot et al., 2018].

One hidden layer Consider a fully-connected neural network. Suppose the input x is of dimensionality N_0 and we have N_1 hidden nodes in layer 1 and N_2 output nodes. The activation function is given by ϕ. The output for node i in the first layer is given by,

$$f_i^{(1)}(x) = \sum_{j=1}^{N_0} w_{ij}^{(1)} x_j + b_i^{(1)}, \tag{1}$$

where $w_{ij}^{(1)}$ is the weight in the first layer connecting node i in layer 1 with input j and $b_i^{(1)}$ is the bias we add in layer 1 to output node i. This is then passed through the non-linearity to obtain:

$$g_i^{(1)}(x) = \phi(f_i^{(1)}(x)). \tag{2}$$

The output layer is consequently given by,

$$f_i^{(2)}(x) = \sum_{j=1}^{N_1} w_{ij}^{(2)} g_j^{(1)}(x) + b_i^{(2)}. \tag{3}$$

Note that in the above notation we make explicit the dependence on x.

We will assume a distribution on the parameters of the network. Conditional on the inputs, this will induce a distribution on the other nodes. In particular, we will assume independent normal distributions on the weights and biases,

$$w_{ij}^{(l)} \sim \mathcal{N}(0, C_w^{(l)}), \text{ i.i.d.} \tag{4}$$

$$b_i^{(l)} \sim \mathcal{N}(0, \sigma_b^{(l)}), \text{ i.i.d.}, \tag{5}$$

where $l = \{1, 2\}$ and denotes the number of the layer, $C_w^{(l)}$ is the covariance for the weights in layer l and $\sigma_b^{(l)}$ is the covariance for the biases in layer l. Note that because the weight and bias parameters are taken to be i.i.d., the post-activations $g_j^{(1)}$, $g_{j'}^{(1)}$ are independent for $j \neq j'$.

Task 1: Proper weight scaling [1pt] We will be interested in the behavior of this network as the width N_1 becomes large (goes to infinity). In order to not have divergence in the variance of the output, we will scale the weight variances for $l \geq 2$ accordingly (in the one hidden layer case only $l = 2$ gets scaled). The appropriate scaling will be,

$$C_w^{(l)} = \frac{\sigma_w^{(l)}}{N_{l-1}}, \; l \geq 2. \tag{6}$$

Can you explain in your own words why such a scaling would make sense?

Task 2: Derive the GP relation for a single hidden layer [4pt] We are now interested in deriving the distribution of the outputs $f_i^{(2)}(x)$ (for different inputs). Specifically, we want to show it converges to a Gaussian process. Use the multivariate central limit theorem to show that $f_i^{(2)}$ follows a Gaussian process distribution and derive the mean μ^1 and covariance matrix K^1.

Hint: arrange the terms into vectors, show that the items we are summing are i.i.d. and apply the CLT. The covariance matrix will be a function of the activation functions and you can leave it as an expected value of the previous layer's output.

Multiple hidden layers Consider an L-hidden layer fully-connected neural network where the hidden layers are of width N_l (for layer l) and the activation function for each layer is given by ϕ. We would now like to extend the arguments in the previous section to the setting of multiple hidden layers, i.e. deep neural networks. We will take the limits $N_1 \to \infty$, $N_2 \to \infty$ and so on in *succession*.

Task 0: Why in succession? [1pt] Explain why we apply the limit in succession.

Task 1: The derivation [4pt] Suppose thus that $f_j^{(l-1)}$ is a GP (over the inputs) identical and independent for every j (and hence $g_j^{(l-1)}(x)$ are independent and identically distributed). The output in the l-th layer is,

$$f_i^{(l)}(x) = \sum_{j=1}^{N_l} w_{ij}^{(l)} g_j^{(l-1)}(x) + b_i^{(l)}, \;\; g_j^{(l-1)}(x) = \phi(f_j^{(l-1)}(x)). \tag{7}$$

Show that $f_i^{(l)}$ is a Gaussian process with mean μ^l and covariance matrix K^l. Make it clear why it is valid to apply the CLT in this case. Derive a recursive expression for the covariance K^l; the final expression will again contain an expected vaue over the previous layer's output.

Part II: Analysing the performance of the Gaussian process and a neural network [10pt]

Consider the ReLU activation function. In this case it can be derived that the recurrence relation for the kernel is given by,

$$K^l(\mathbf{x}, \mathbf{x}') = \sigma_b^2 + \frac{\sigma_w^2}{2\pi}\sqrt{K^{l-1}(\mathbf{x}, \mathbf{x})K^{l-1}(\mathbf{x}', \mathbf{x}')}\left(\sin\theta_{\mathbf{x},\mathbf{x}'}^{l-1} + (\pi - \theta_{\mathbf{x},\mathbf{x}'}^{l-1})\cos\theta_{\mathbf{x},\mathbf{x}'}^{l-1}\right), \tag{8}$$

$$\theta_{\mathbf{x},\mathbf{x}'}^{l} = \cos^{-1}\left(\frac{K^l(\mathbf{x}, \mathbf{x}')}{\sqrt{K^l(\mathbf{x}, \mathbf{x})K^l(\mathbf{x}', \mathbf{x}')}}\right), \tag{9}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are two datapoints of dimension N_0. For K^0 we would have,

$$K^0(\mathbf{x}, \mathbf{x}') = \sigma_b^2 + \sigma_w^2\left(\frac{\mathbf{x}\cdot\mathbf{x}'}{N_0}\right). \tag{10}$$

We will use this recurrence in the implementations.

Dataset We will use the CIFAR10 dataset. However, we will select just two classes of the dataset so that we are working with binary classification. You are free to select the classes you want. Also subsample 1000 train samples (i.e. we will not work with the full dataset).

Task 0: Formulate classification as a regression problem [1pt] We will perform classification using a regression-type output. Change the class labels to -0.5 and +0.5. How can you use the Gaussian process to perform classification and how can you formulate your decision rule to perform the classification?

Task 1: Implement the kernel of the Gaussian process [3pt] Use relation (9) to implement a function that defines the GP kernel. This function should take as input the number of layers L, σ_w^2, σ_b^2 and two datasets X^1, X^2 of size $M_1 \times N_0$ and $M_2 \times N_0$ where N_0 is, as before, the dimension of the input and M_1 and M_2 are the number of datapoints in the first and second dataset, respectively. The output from this function should be a matrix K^L where each element K^L_{ij} is the kernel function applied to element i from X^1 and element j from X^2.

Task 2: Implement a method to compute the mean and covariance of the Gaussian process [2pt] Implement a method to compute the mean and covariance of the Gaussian process posterior using:

$$f^*|X, y, X^* \sim \mathcal{N}(K^L(X^*, X)[K^L(X,X) + \sigma^2 I)]^{-1}\mathbf{y}, \tag{11}$$

$$K^L(X^*, X^*) - K^L(X^*, X)[K^L(X,X) + \sigma^2 I]^{-1} K^L(X, X^*)), \tag{12}$$

respectively, where $(X, \mathbf{y})$ is the train dataset and X^* is the dataset for which we want to obtain the prediction, σ is the noise level, and K^L is the kernel matrix for layer L. This function should take as input the train dataset $(X, \mathbf{y})$, the test points X^*, the noise σ, the weight and bias variance parameters σ_b^2 and σ_w^2 and the number of layers L. The output from the function should be the posterior mean and covariance. Hint: if you get NaN values, you may need to clip the outputs in the kernel function before passing it into the acos; you can do this using *torch.clamp*.

Task 3: Analyse the performance of the Neural Network Gaussian process [2.5pt] Let us measure the performance in terms of the accuracy (the proportion of samples predicted correctly) on a test set. Compute the accuracy over 1000 test samples using the posterior mean. You will also need to set the σ_b^2 and σ_w^2 in an appropriate manner. One option is to define a grid of possible values and choose the values which achieve highest test performance. Present results on the performance of the NNGP changes as you increase the number of layers L from 1 to 10 (in a table of in a figure). Discuss your results: what is the impact of depth on the performance? What σ_w^2 and σ_b^2 are optimal? Any other observations on the performance? Note: the σ_b^2 and σ_w^2 can thus differ for each L.

Task 4: Analyse the uncertainty [1pt] For each test point you can also compute the posterior covariance. Present the 2 test samples for which the uncertainty (i.e. the covariance) is highest and the 2 test samples for which it is lowest.

Task 5: Computational cost analysis [0.5pt] Can you identify where the computational cost of the NNGP model lies? What would happen if you increased your dataset size? Discuss.

Option 2: A DeepDream-like model

Setup In this task we will implement a DeepDream model in order to generate 'dreams': images enhanced by the neural network.

Part 1: A convolutional neural network, dropout and batchnorm [6pt]

Task 1: Implement and train a CNN [2pt] Load the CIFAR10 train and test dataset. Implement a convolutional neural network (follow the standard structure for a neural network class and implement a training loop in whichever way you wish). Train the CNN on the CIFAR10 dataset. Present the train and test accuracy (print these out in the usual way). You are free to choose the model architecture as you wish, but use only convolutional and fully-connected layers. Make sure to achieve an accuracy of above 60% on the test set.

Task 2: Dropout [2pt] Explain what dropout does. Then, add dropout layers to your model using *torch.nn.Dropout.* Use the dropout layers in your forward call. What kind of results do you observe? Does dropout help with generalisation? Does the location where you place the dropout layer matter?

Task 3: BatchNorm [2pt] Explain what BatchNorm does. Then, add batchnorm layers to your model using e.g. *torch.nn.BatchNorm2d.* Similar to dropout, analyse the performance.

Part 2: The DeepDream method [14pt]

Here we will work with a pre-trained model (Inception-V3). The idea is as follows: we will load some image (you can later on choose to load your own image if you want). We will load the Inception-V3 model. We will pass it an input and obtain the activation outputs from a certain layer. We will 'ask' the neural network to maximise the values of the activation, i.e. enhance the image with what those layer activations 'like' to see. For this we will implement a gradient ascent method in order to maximise a norm of the activations over that input. This will change our input into a 'dream'.

Task 1: Load an image [1pt] We will begin with loading an image. Make sure you install all the needed libraries. We will use the below code. Add code that will display the loaded input image.

```
url, filename = ("https://github.com/pytorch/hub/raw/master/images/dog.jpg",
"dog.jpg")
try: urllib.URLopener().retrieve(url, filename)
except: urllib.request.urlretrieve(url, filename)
# sample execution (requires torchvision)
from PIL import Image
from torchvision import transforms
input_image = Image.open(filename)
preprocess = transforms.Compose([
    transforms.Resize(299),
    transforms.CenterCrop(299),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]),
])
input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0) # create a mini-
batch as expected by the model

# Print initial image
... TO ADD ...
```

Task 2: Load the pre-trained model and write code to allow for a 'hook' [2pt] We will load the Inception-V3 model using the following code:

```
    model = torch.hub.load('pytorch/vision:v0.10.0', 'inception_v3',
    pretrained=True)
    model.eval()
```

Then, we will want to also get intermediate activation outputs from this model. If we run

```
model(input_batch)
```

this would just give us an output from the final layer, so we need to change some things. We will get the intermediate activation by registering a hook. We will use the following code:

```
outputs= []
def hook(module, input, output):
    outputs.append(output)
```

Can you explain in your own words what this code does? Hint: you may need to look on e.g. stackoverflow or in the PyTorch documentation.

Task 3: Create a hook [**1pt**] To obtain the values of the activations when passing a certain input, we need to know which layer's outputs we want. You can check the different layers here: `https://github.com/pytorch/vision/blob/main/torchvision/models/inception.py`. Create a hook for a layer of your choice using

```
h = model.[some layer].register_forward_hook(hook)
```

Task 4: Implement the deepdream optimisation loop [**4.5pt**] We will use gradient ascent to optimise for the norm of the activation. Implement a function that performs this gradient ascent. This function will take as input the number of iterations to perform, the learning rate, and the start image (i.e. the image which the model will enhance). Define in the function a for loop over the number of iterations, obtain the model output, get the output of the hook (hint: use something like *outputs[-1]*), compute the loss which will be L2 norm of this output (i.e. the L2 norm of the activations from your chosen layer), compute the gradients of this L2 norm loss, scale the gradients by their absolute average (hint: scale using *torch.abs(g).mean()*) and define a gradient *ascent* step over the image. Do not forget to zero out gradients where needed.

Task 5: Present comparisons of outputs [**3pt**] Implement code to display the output from your optimisation loop (using e.g. numpy). Present the generated images for different choices of layer activations and different number of optimisation steps. Discuss your results. What is the impact of using different layers? What is the impact of the number of optimisation steps?

Task 6: Improve the results by allowing to optimise for multiple layer activations [1.5pt] In our previous setting we registered a single hook and allowed for optimising for a single layer's activation. Can you change the method such that it will allow to compute the loss (L2 norm) of multiple layers' outputs? You will need to change both the creation of the hook and you will need to change something in the gradient ascent loop.

Task 7: Play around with the model and create your own dream [1pt] Load your own image of choice. Play around with the hyperparameters. Present the dream you like the most. You are free to also improve the method in whichever way you think is useful (e.g. using the octaves method presented in the original method `https://github.com/eriklindernoren/PyTorch-Deep-Dream/blob/master/deep_dream.py` and `https://www.tensorflow.org/tutorials/generative/deepdream`).

References

[Jacot et al., 2018] Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31.

[Lee et al., 2017] Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., and Sohl-Dickstein, J. (2017). Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165.*

[Matthews et al., 2018] Matthews, A. G. d. G., Rowland, M., Hron, J., Turner, R. E., and Ghahramani, Z. (2018). Gaussian process behaviour in wide deep neural networks. *arXiv preprint arXiv:1804.11271.*

[Neal, 1996] Neal, R. M. (1996). Priors for infinite networks. In *Bayesian Learning for Neural Networks*, pages 29–53. Springer.