L193 - Explainable Artificial Intelligence
Practical #2: Concept Bottleneck Models

Mateja Jamnik, Zohreh Shams, Mateo Espinosa Zarlenga

Due date: March 11$^{\text{th}}$, 2025 at 14:00

1 Introduction

In our first practical, you got the chance to get a hands-on exposure to using feature attribution methods for debugging a Deep Neural Network (DNN). In this practical, you will instead explore a key method in another big chunk of the material we covered during the lectures: *Concept Learning*. Specifically, we will ask you to implement Concept Bottleneck Models (CBMs) [1] and to explore some of their interesting, and sometimes quirky, properties. This practical will therefore consist of a series of exercises that will take you from training a simple CBM to understanding how its training process, hyperparameters, and design affect its behaviour.

When to submit This completed practical will be assessed and will contribute to the 10% part of your final module mark for L193. The deadline for submitting completed work is **11$^{\text{th}}$ of March, 2025 at 14:00** via Moodle course page.

What to submit Although most of the questions on this practical will require you to write and execute some code, we want to emphasise that we are more interested in evaluating your understanding of the concepts you are working with (pun intended) rather than your ability to write readable code. With this in mind, for each question in this practical, we ask you to submit a short description of what you did to solve it (which may include a few key Python commands required for this purpose) and then proceed to emphasise the key takeaways of the results you obtained. These takeaways will very likely be best represented via plots or figures which will then become the main component of each solution. If possible, we ask you to write one to two paragraphs for each exercise's part, only exceeding this if it is really necessary.

Notice that we will base your **mark on only the non-optional parts** of the practical. The optional parts are there for you if you are interested and to have fun and learn more. Feel free to submit answers to those too and we will strive to provide feedback.

How to submit Once you have compiled all your answers in a single document, we ask you to please submit it via Moodle **as a PDF**. Besides LaTeX-generated PDFs, we will accept iPython notebooks exported as PDFs as valid solutions

as long as they are clearly organised and are focused more on the interpretation of results rather than on the code you ran to get to those results. If you opt to submit a PDF version of an iPython notebook, please make sure your answers to the practical's questions are written in Markdown and clearly marked (e.g., we should be able to quickly find the answer to each exercise and subexercise).

What to focus on This practical is about deeply understanding CBMs, their power, and their limitations. It is **not** about designing state-of-the-art models or training pipelines. Therefore, we ask you to please focus on the methodology itself when answering different questions rather than focus on optimising a CBM to perfection; it is perfectly ok if a CBM could have performed better if more time was put into its training but you chose not to do so. We hope that the models you will be using for the exercises on this practical can be trained to reasonable accuracy with at most a couple of minutes of GPU training in Colab. Unless you wish to use your own personal GPU for these exercises, we suggest that you use Google Colab as we did in our first practical.

2 Background

As introduced in the lectures, Concept Bottleneck Models (CBMs) [1] are a family of concept-based self-explaining neural network architectures. These models construct explanations for their predictions using high-level concepts learned from annotations in the training data. When deployed in supervised settings, CBMs learn to generate such explanations by splitting their architecture as a composition of two functions: (1) a *concept encoder* model $g : \mathcal{X} \to \mathcal{C}$ that maps input samples $\mathbf{x} \in \mathcal{X}$ to a set of k binary concepts $\hat{\mathbf{c}} \in \mathcal{C} \subseteq [0, 1]^k$ and (2) a *label predictor* model $f : \mathcal{C} \to \mathcal{Y}$ that maps predicted concepts scores $\hat{\mathbf{c}}$ to predicted output labels $\hat{y} \in \mathcal{Y} \subseteq \mathbb{N}$. This architectural design, shown in Figure 1, allows for test-time interventions where an intermediate concept prediction can be corrected by a domain expert and fed back into the model's label predictor to improve the end performance.

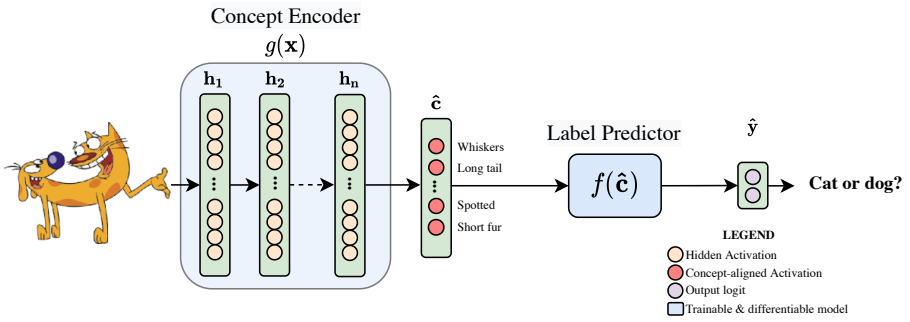

Figure 1: Depiction of a Concept Bottleneck Model used to predict whether an input image is a cat or a dog.

In our lecture on in-model concept-interpretable models (lecture 4) we discussed how these models are naturally constrained to have a "bottleneck" layer with as

many activations as provided training concepts. In this practical, we will explore how this can be both a blessing and a curse for CBMs: a blessing because it allows meaningful interventions and a curse because it constrains their capacity when one does not have meaningful-enough concepts in the training set. Before moving forward, we ask you to familiarise yourself with CBMs, how they are trained, and how can they be used by reading the original paper found here. Once you have read over this paper and understood its core components, you should be ready to proceed and complete the following parts of this practical.

3 Setup

Similar to our practical, for this practical we will focus on using an MNIST-based dataset in which every sample is a pair of handwritten digits from 0 to 5 (inclusive) that have been annotated with their sum as a label.[1] To get this dataset, you can download the `numpy` files from this folder to your working directory (later in Section 4, **we also give you some boilerplate code that does this to a Colab directory if you wish to use Colab for this practical**). These two files are our test and training splits and you should be able to access and load them as we did in the practical session as follows:

```
import numpy as np

train_data = np.load('train_sum_mnist_concepts.npz')
x_train = train_data['x_train']
y_train = train_data['y_train']
c_train = train_data['c_train']

test_data = np.load('test_sum_mnist_concepts.npz')
x_test = test_data['x_test']
y_test = test_data['y_test']
c_test = test_data['c_test']
```

Each of these datasets has (a) a set of samples $\mathbf{x} \in \{0, 1, \cdots, 255\}^{B \times 28 \times 56 \times 1}$ of grey-scale images, (b) a set of categorical labels $y \in \{0, 1, \cdots, 10\}^{B}$ for those samples representing the result of adding their digits, and (c) a vector of concept annotations $\mathbf{c} \in \{0, 1\}^{B \times 12}$ where the first 6 entries are a one-hot encoding of the digit in the left image of the input and the last 6 digits are a one-hot-encoding of the digit in the right image of the input (thus, $\mathbf{c} =$ [left_0, left_1, left_2, left_3, left_4, left_5, right_0, right_1, right_2, right_3, right_4, right_5]).

For example, if image $\mathbf{x}$ is formed by a handwritten 4 next to a handwritten 2 then its corresponding label y will be $y = 6$ and its concept annotation vector will be $\mathbf{c} = [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]$. Notice then that this set of concept annotations is capable of perfectly explaining a sample's label (as one can extract the exact values of the left and right digits from them), implying that they can be used to train a CBM to produce meaningful concept-based explanations for the additive task we worked on during our first practical.

[1] No tricks this time, we promise! No spurious correlation has been added to this training set.

4 Exercises

We now ask you to complete the following exercises using the dataset described above. Please make sure to **fully read each exercise's instructions before jumping into implementing it** as some details may come at later points in the instructions.

Boilerplate Helper Code To help you with some guidance in this practical, we provide some very simple boilerplate code in this Colab Notebook. This code is to help you with getting started with this practical. However, it is far from complete and you are free to edit, use, or ignore as much of it as you see fit/prefer. This practical is more hands-on than the previous one and will require you to get a bit deeper into the details of implementing, running, and testing models. As such, it is important that you have the freedom to approach these tasks using your preferred path and having more say in how you design your experiments. Again, this is because we will grade you based on your analysis and presentation of the results, rather than on your ability to code.

Model Architectures For simplicity, when asked to implement and train CBMs, we would like you to use the following DNN architectures for the CBM's concept encoder and label predictor:

- **Concept Encoder:** For the concept encoder, please use a Convolutional Neural Network with the following architecture: the model will consist of three convolutional layers with each having four 3×3 filters and using "same" padding. The convolutional layers will then be followed by a single linear layer whose outputs will be in $[0, 1]$ (using a sigmoidal activation) and whose size will be the same as the number of training concepts. Between each of the layers, please use a Leaky ReLU non-linear activation function. You may find PyTorch's Conv2D, Linear, and Sequential layers useful for this.
- **Label Predictor:** For the label predictor, please use a simple linear layer mapping the bottleneck to the output classification logits. You may find PyTorch's Linear layer useful for this.

We ask you to please implement and train these models using PyTorch. For a refresher of how PyTorch can be used, feel free to reference this tutorial or the instructions from our first practical on how to train a PyTorch model. When building your CBM, you may find it useful to encapsulate all of its components into a single PyTorch module object.

As mentioned earlier, please do not focus too much on over-optimising these models. Instead, fix the number of epochs (e.g., to $T = 50$), batch size (e.g., to $B = 128$), and the optimizer (e.g., ADAM optimizer with learning rate 10^{-3}) throughout all experiments. There is **no need to be extremely fair when evaluating all models** by supplying equal "resources" to all of them. Using these values, we could train the CBMs below in 1-2 minutes each in a standard Colab GPU.

Exercise 1 Train a CBM on the MNIST addition task defined above using the three different training schemes defined in the paper: (1) sequential, (2) independent, and (3) joint training. For joint training, fix the concept loss weight to $\lambda = 1$ for now and for all models use a sigmoidal activation in the bottleneck (i.e., that way all concept predictions are between 0 and 1). Once you have trained all three variations, please answer the following questions:

1. How do the test task accuracies compare between CBMs trained with these three training regimes?
2. If you set $\lambda = 0$ when training the joint CBM, what does your model reduce to?
3. How do each variation's task accuracy fare against that of a black-box DNN with the same architecture?
4. How do the test concept accuracies[2] compare between CBMs trained with these three training regimes?
5. What do you think explains the results observed above?

Exercise 2 Evaluate the effect of test-time concept interventions in the test task accuracy on all three CBMs above by changing an increasing number of activations in the CBM's bottleneck using the ground-truth concepts known during testing. Then answer the following questions:

1. How does the test task accuracy of all three models change as the number of intervened concepts changes? Why do you think the results look the way they do?
2. If you would have to use one of these models in a real-world situation where an expert is given access and full control over the concept explanations, which model would you choose and why?

Exercise 3 In the lecture we discussed that CBMs may struggle if the set of training concepts is not fully descriptive of the task. In this exercise, we ask you to explore this question by training **only** the joint CBM (set $\lambda = 1$ for now) using only 50% of the annotated concepts during training (selected at random). This means that before training your CBM, you must randomly chose a fixed subset of all k concepts $\mathcal{S} \subset \{1, 2, \cdots, k\}$ such that $|\mathcal{S}| = k/2$ and each training sample $\mathbf{x}^{(i)}$ is given only concept annotations $\mathbf{c}_{\mathcal{S}}^{(i)}$ during training. Once you do this, please answer the following questions:

1. How does this model's task and test accuracy compare to that of its counterpart trained with the whole set of concepts? Can you explain why you observe this difference, if any?
2. How does the CBM react to concept interventions at test time in this variation of the dataset? Why do you think you observe this difference w.r.t. our results in Exercise 2 if any?

[2]For this practical, we will consider the "concept accuracy" as the **mean concept accuracy** for all concepts.

3. **(Optional)** How does the Joint CBM's task and concept accuracy vary as we change the number of concepts used during training to be $\{20, 40, \cdots, 100\}\%$ of the total number of training concepts? Would you expect a vanilla DNN with the same architecture as these models to exhibit this same behaviour as we change the size of its bottleneck?

Exercise 4 When using jointly trained CBMs, it is crucial to select a proper value λ, the weight assigned to the concept loss during training. Run a simple ablation study on the modified dataset from the previous exercise where you plot the test concept and task accuracies of a jointly trained CBM as we change the λ over $\{0.001, 0.1, 1, 10\}$. What does this study tell you about the importance of λ in tasks where concept annotations are not complete?

Exercise 5 (Optional) It may be very tempting to solve the issue we observed in Exercise 3 by increasing the capacity of a jointly-trained CBM's bottleneck by adding a few unsupervised activations to it (i.e., we add k' new neurons to the bottleneck which will not be aligned to any training concepts). Train such a joint CBM by adding $k' = 32$ unsupervised neurons to its bottleneck and answer the following questions:

1. **(Optional)** How do its test and task accuracies compare to the joint CBM trained in exercise 3? Can you explain the observed differences, if any?

2. **(Optional)** How does this model behave as you intervene on an increasing number of concepts? Do you have any intuition to explain the observed results?

3. **(Optional)** Would you observe the same intervention results if you vary k' across multiple values (both above and below 32)? Do you have an intuition as to why this may or may not be the case?

4. **(Optional and open)** If you let k' be equal to the number of concepts we removed during training of the joint CBM in exercise 3 and you train a joint CBM with k' unsupervised neurons in its bottleneck, do any of its unsupervised neurons naturally "align" with concepts that we did not include during training? Why do you think this is or is not the case? For this part, you may use any metric to define "alignment" between a learnt neuron and an unseen concept (e.g., Pearson correlation coefficient may work). This is an open question so feel free to be creative as you wish!

References

[1] Pang Wei Koh, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. Concept Bottleneck Models. In *International Conference on Machine Learning*, pages 5338–5348. PMLR, 2020.