Challenge #2: An empirical study on the learnability of functions by *NNs*

In the following *challenge exercises*, we will empirically investigate the behaviour of deep neural networks with respect to the learning of specific classes of functions, and in specific training regimes.

A. The effect of *under-* and *over-*parameterisation in the *Teacher/Student* setup

In this exercise, we will train deep neural networks (*students*), supervisedly, on input/output pairs produced by another deep neural network with frozen weights (*teacher*).[1] Given our ability to modulate the expressivity of both networks, this setup allows us to disentangle the effects of task hardness, model expressivity, and training dynamics.

We will monitor the training and test loss of the *students* during training, as well as the final distribution of weights. We will do so in three regimes: when the *student* has much less (*under-parameterisation*), much more (*over-parameterisation*) or exactly the same learnable parameters (within a fixed given structure) as those frozen in the *teacher*.

What to do?

1. Instantiate the *teacher* model $\mathcal{T}$, a fully-connected feedforward neural network mapping a 100-dimensional input to a single output scalar. Use 3 hidden layers of sizes, respectively: 75, 50, 10. Use the ReLU activation function after all neurons, except for the output. Weights and biases should be initialised as *i.i.d.* samples from the Standard Normal distribution. Keep those parameters fixed for the rest of the exercise.
2. Generate the **test set** for the learning task, by repeatedly querying the *teacher* model. Inputs $\boldsymbol{x}_i \in \mathbb{R}^{100}$ should be obtained as samples from the multivariate Uniform distribution in the interval $[0, 2]^{100}$, whereas the outputs as $y_i = \mathcal{T}(\boldsymbol{x}_i)$. Generate at least 6×10^4 datapoints and keep them fixed for the rest of the exercise. Such points will be used as a way to quantify generalisation error by the *student* model.
3. Instantiate the *student* model $\mathcal{S}$, a fully-connected feedforward neural network mapping a 100-dimensional input to a single output scalar – as in the case of the teacher. Repeat the steps that follow with (at least) three different *student* models, architecturally identical to the *teacher* but with different number and width of the hidden layers.
   - S_u: one hidden layer of size 10;
   - S_e: as for the *teacher*;
   - S_o: 4 hidden layers of sizes 200, 200, 200, 100;
4. Train the *student* model on the *MSE* loss for a sufficiently large number of iterations, as to allow for the training and test loss to reach a quasi-stationary behaviour. To actually perform the training, harvest a fresh sample of B inputs (*i.e.* $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_\mathrm{B}\}$) per iteration, label each of them using the *teacher* model, and train the *student* on the given batch. Use an optimizer of your choice, taking care to tune (at least) its learning rate to minimize time to convergence. Do not use default learning rates assuming they are already optimal! Do not optimize batch-size (as it is scarcely effective when tuning also the learning rate[2]): you can use $\mathrm{B} = 128$ (or less, if you cannot make it fit into memory).
   As the training progresses, log the training loss (every batch, if you can). Additionally, log also the test-set loss every given number of batches (of your choice).
5. Once the training is over, evaluate the *student* model on the test set one last time. Additionally, collect (separately) weights and biases for each layer of the *student* network, and compare their distribution to that of the *teacher* network. Do the same for the collection of all weights and biases of the network (*i.e.* not on a layer-wise basis).

Comment on the results collected, specifically in terms of: number of learnable parameters, trainability, generalisation, distributional convergence to target parameters. Do so individually in each case, as well as in comparison across the different *student* models.

B. Function learning and hierarchical structure

In this exercise, we will train a particular kind of *deep residual network*, supervisedly, on examples generated by two specific polynomials. Although their monomials share most of their respective properties, one polynomial shows a strongly hierarchical structure[3], whereas the other does not. The hierarchical polynomial is B_6, *i.e.* the *sixth-order multivariate complete Bell polynomial*, which is defined as follows.

$$B_6(x_1, x_2, x_3, x_4, x_5, x_6) = x_1^6 + 15x_2x_1^4 + 20x_3x_1^3 + 45x_2^2x_1^2 + 15x_2^3 + 60x_3x_2x_1 + 15x_4x_1^2 + 10x_3^2 + 15x_4x_2 + 6x_5x_1 + x_6.$$

We will analyse the generalisation error of the same model trained to reproduce each of the two polynomials, both in terms of general input/output mapping ability and sensitivity with respect to the variation of individual input components.

What to do?

1. Define the non-hierarchical counterpart of B_6, which we will call $\tilde{B}_6 : \mathbb{R}^6 \to \mathbb{R}$, with a *scrambled* monomial structure. In detail, start from the definition of B_6 and iteratively replace the x_{i_k} from each i^{th} monomial with a different $x_{i_{\sigma_i(k)}}$ so that:
   - $\tilde{B}_6$ still depends non-trivially on all six input variables $x_1, x_2, x_3, x_4, x_5, x_6$.
   - No two monomials of $\tilde{B}_6$ share the same permutation σ_i of indices.
   - No two monomials (regardless of their coefficient), although not sharing the same permutation σ_i of indices, can be rearranged as such by means of the commutative property of sums and/or products. This should be a concern only for the two terms $15x_4x_2$ and $6x_5x_1$.
2. Generate both a **training set** and a **test set** associated with B_6 and $\tilde{B}_6$. In particular:
   - Harvest *i.i.d.* input vectors $\boldsymbol{x} = (x_1, \ldots, x_6) \in \mathbb{R}^6$ from the multivariate Uniform distribution in the interval $[0, 2]^6$;
   - Compute the associated output scalar $y \in \mathbb{R}$ as $y = B_6(\boldsymbol{x})$ or $y = \tilde{B}_6(\boldsymbol{x})$. You do not need to share the same inputs $\boldsymbol{x}$ among the datasets generated by the two polynomials.

   Each training set should contain at least 10^5 datapoints, whereas each test set at least 6×10^4.
3. Instantiate the model as a *fully-connected residual* deep neural network, *i.e.* a fully-connected feedforward neural network where each layer is endowed with a ResNet-style skip connection. Layers of different sizes share no skip connection, which is simply dropped.
   Use a network with 9 layers (1 input layer, 8 hidden layers, 1 output layer), where hidden layers have all size 50. Use the ReLU activation function after all neurons, except for the output.
4. Train the model (in each of the two cases, *i.e.* B_6 and $\tilde{B}_6$) on the *MSE* loss for at least 30 epochs, using a batch size $\mathsf{B} = 20$. Use the Adam optimiser, with a learning rate tuned to minimise training error (independently across the two datasets). As the training progresses, log the training and test losses (at least once at the end of each epoch).
5. Once the training is over, evaluate the model one last time on the test set. The result of such evaluation will be used as the *final* generalisation error.
6. Investigate how the trained network models the dependency of the output on each input variable separately. To do so, proceed as follows.
   - Harvest a new input vector $\boldsymbol{x} = (x_1, \ldots, x_6) \in \mathbb{R}^6$ from the multivariate Uniform distribution in the interval $[0, 2]^6$;
   - For each of the input components, keep all the others fixed to their sampled value, whereas the one considered is evaluated on a fine uniform 1D grid $\mathcal{G} \subset [0, 2]$.
   - Evaluate both the target polynomial and the trained model on all resulting input points so generated, grouping their results according to the variable that is swept along the input interval. Compare the results of the sweeps (it is better to do so graphically!).
7. **Optional:** Repeat step 6 some more times, and present results in an aggregate form. This is useful to average over the randomness introduced when evaluating the input/output dependency of single input components. Indeed, such evaluation strongly depends on the specific input values sampled in the first place before generating the sweeps.

Comment on the results. How does the hierarchical structure of the function to be learnt influence the learning process and/or the *final* learnt model?

1. E. Gardner and B. Derrida; Three unfinished works on the optimal storage capacity of networks. Journal of Physics A: Mathematical and General, 22(12):1983–1994, 1989. ↩
2. V. Godbole, G.E. Dahl, J. Gilmer, C.J. Shallue and Z. Nado; Deep Learning Tuning Playbook, 2023. ↩
3. T. Poggio and M. Fraser; Compositional sparsity of learnable functions. Bulletin of the American Mathematical Society, 61:438-456, 2024. ↩