CS 6.5: Theories of Deep Learning
Problem Sheet 1

Prof. Jared Tanner

October 19, 2020

The questions marked with an asterix should be submitted for marking. The remaining questions entail computational experiments which should be attempted based on the available notebook[1] but need not be submitted; they will be discussed in tutorials.

Coding setup:

1. For the practical exercises in the upcoming tutorials, you can complete them in whatever programming language you prefer. When code is provided to you, however, it will be in Python (Google Colab) notebooks.
2. If you use Python on your own machine, make sure you have key libraries that you will require for machine learning development, specifically: Tensorflow/Pytorch, SciPy, NumPy, Matplotlib, and Scikit-learn.
3. When working on deep neural networks, if/when you need to use a deep learning package, you are permitted to use whichever you prefer. However, do note that when code (and solutions) are provided to you, this will be in Tensorflow 2, and as such we recommend sticking to TF.
4. If you want to use Python but prefer not to install everything on your laptop, you can create a Google CoLab notebook, which will already have most of the packages you will need installed and ready to use. Just click 'File' → 'New Notebook'. Your notebooks get saved to your Google drive. Google Colab will be the most straightforward option for completing any coding excercises given in the problem sheets. Simply navigate to the notebook provided, click File → Save a Copy in Drive, and this will make a copy which you can edit directly.

---

1. Training Nets with Backpropagation
   (a) (*) Consider a standard feedforward network defined by $h_1 = x$ and $h_{j+1} = \sigma(W^{(j)}h_j + b^{(j)})$ for $j = 1, \ldots, N-1$, and define the loss as the sum of squared errors, $\sum_{i=1}^{n} \|h_N(x_i) - y_i\|_2$. Derive the formulae used for 'backpropagation', which would be used to update the weights and biases in the network when optimising the loss function using gradient descent.
   (b) Consider a XOR problem, i.e. data points $x_i \in \{0,1\}^2$ for all i, with points (0,0), (1,1) labelled 0, and (1,0), (0,1) labelled 1.
      i. Can a linear classifier correctly classify these points? Sketch the domain and the desired decision boundary of your classifier to motivate your answer.
      ii. Next, you will train a single-hidden-layer network classifer with sigmoid activations from scratch (without help from Tensorflow or Pytorch) to solve XOR. In the notbook attached, calculate the required gradient updates and use these to fill in the `NeuralNetwork.backprop()` function, which will allow you to then train the network, and visualise the results. Take note of how the learnt decision boundary compares with what you expected.

[1] `https://colab.research.google.com/drive/1Yoktyk35O5WRaaxYfyHK0bHPk-SQ1RmS?usp=sharing`

(c) The "Hello World" of supervised deep learning: MNIST digit classification
Now that you have trained a network from scratch yourself, you are unlikely to ever do this again. Deep learning libraries have developed high level APIs which make it very easy to quickly build and train models. The ability to use these APIs to quickly build and investigate small models is a very useful skill in (even theoretical) deep learning research, allowing you to run experiments and probe hypotheses.

MNIST is a benchmark dataset of handwritten digits, that has approximately 70,000 example images from 10 classes. Each 2-dimentional image is of size '28×28', and belongs to one of the categories '0-9'.

Fill in the code for the missing steps in the attached notebook to prepare the data, add layers to the model, compile the model with a loss function (which loss function will you use?), and then train, and finally evaluate, your model.

2. Expressivity and depth

(a) (*) In Yarotsky (2016) it is proven that $f(x) = x^2$ on $[0, 1]$ can be approximated with any error $\varepsilon > 0$ by a ReLU network having the depth and the number of weights and computation units $\mathcal{O}(\log(1/\varepsilon))$. The proof relies on the following statement:

Let f_m be the piece-wise linear interpolation of $f(x) = x^2$ with $2^m + 1$ uniformly distributed breakpoints $\frac{k}{2^m}$, $k = 0, \ldots, 2^m$. The function f_m approximates f with the error $\varepsilon_m = 2^{-2m-2}$ (in the ℓ_∞ norm).
Prove this statement.

(b) (*) Recall from lectures that the 'sawtooth' function can be created by iteratively composing the single-hidden-layer network $f(x) = \sigma(2\sigma(x) - 4\sigma(x - 1/2))$, where $\sigma(x) = \max(x, 0)$. Can the same be achieved with a network of the same width and depth if hard-tanh activations are used instead of ReLU? If so, write down the corresponding function.

(c) (*) Consider n-ap problem (the n-alternating-point problem). Dataset consisting of a set of n uniformly spaced points within $[0, 1 - 2^{-n}]$ with alternating labels, i.e., the pairs $((x_i, y_i))$ with $x_i = i2^{-n}$, and $y_i = 0$ when i is even. How many layers does a width-2 ReLU network need in order to have the capacity to solve the n-ap problem? How wide would the layers need to be if there were only two layers?

(d) Create an n-ap dataset for $n = 3$ and visualize the points by plotting. Implement and train a network of the minimum necessary depth (identified in the previous question) to solve the n-ap problem for $n = 8$ (see the code outline in the attached notebook). How well does it perform?

(e) Now, hardcode the weights which would solve the n-ap problem exactly, (e.g. specifying the weights of the NN, or compose the function $f(x) = \sigma(2\sigma(x) - 4\sigma(x - 1/2))$ multiple times) and plot the result. Now add small Gaussian noise ($\sigma = 0.1$) to the weights/coefficients. What happens to the function? Does this tell us anything about the loss function which was optimised to train this network in the previous question?