Multilayer networks trained with backpropagation

Train a two layer network to classify hand written digits (the famous MNIST dataset). Use the matlab code and the dataset that are available on the website. The parameters are explained in the .m file. If the computer that is used is slow, it is possible to reduce the size of the dataset.

- Choose the size of the two hidden layers, the learning rate and the number of epochs to get a cross-validated performance of at least 96% (4% error). A performance of 98% is not too difficult to achieve. The initial parameters in the code on the website are bad, you need to find the good ones.
- Once the network works, plot the training and the test error as a function of the number of epochs.
- Variability from run to run: Run the program 10 times, and plot the training and test errors for all the runs on the same plot. Determine the minimum and the maximum number of iterations needed to achieve a 96% performance. The variability that will be observed is one of obstacles to study deep networks.
- Choose the size of the two hidden layers to minimize the training error and to maximize the test error (overfitting situation). Plot the errors as a function of the number of epochs.

Autoencoders (optional)

Reproduce the famous figure of Kingma and Welling (2017) [1], shown below. Since some details are obscured in their paper, try the following hyperparameters:

Encoder network (a.k.a. approximate posterior): $784 \to 500$ (ReLU) $\to 2$ (Sigmoid).

Latent distribution: Gaussian with diagonal covariance, i.e. $z_i \sim \mathcal{N}(\mu_i, \sigma_i)$

Decoder network (a.k.a. generative model): $2 \to 500$ (ReLU) $\to 784$ (Sigmoid).

Output distribution: 784 Bernoulli variables, i.e. $pixel_i \sim Bern(p_i) = p_i = decoder(z_i)$. (Essentially, each pixel is a value b/t 0 and 1, which is interpreted as a probability. That is how we get the 'cross-entropy' loss.)

[1] I **don't** necessarily recommend reading the Kingma and Welling paper; it was a very important and influential paper, but it's remarkably obscure, even to those in the field. I **do** recommend this blog post explaining VAEs; it's very clearly written, and the very first equation will provide the loss function of the VAE. (How to implement that loss function is a separate issue.)

You can of course play with these hyperparameters. If this was a simple exercise for you, try convolutional/deconvolutional networks as the encoder and decoder (respectively). Alternatively, consider adding a coefficient (β) to the D_{KL} term of the loss, and slowly increase/decrease it during training.

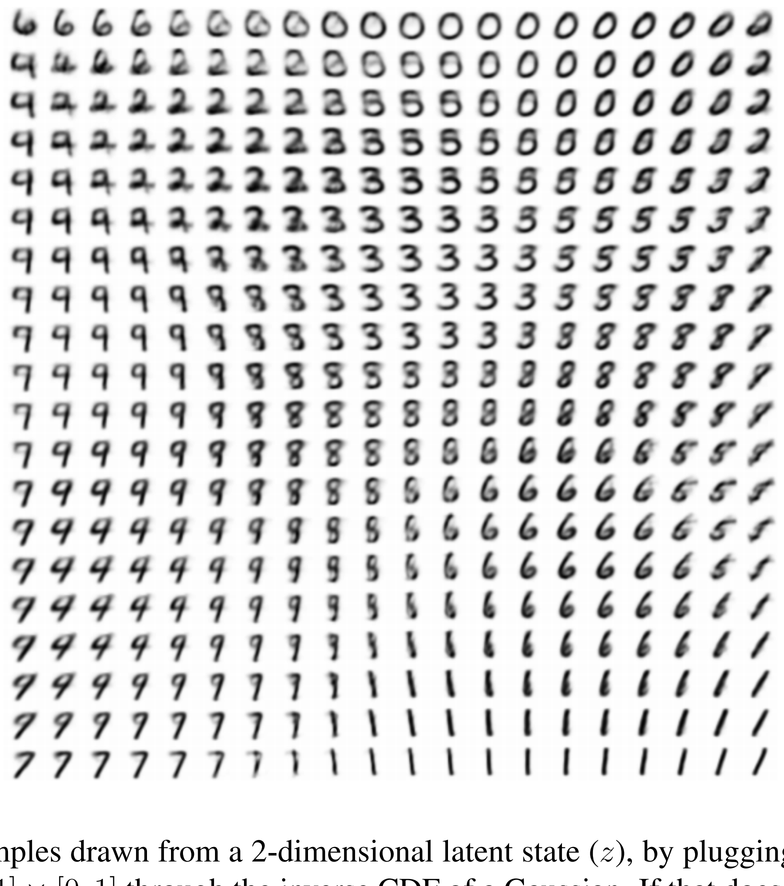

Figure 1: Samples drawn from a 2-dimensional latent state (z), by plugging uniform samples from $[0, 1] \times [0, 1]$ through the inverse CDF of a Gaussian. If that doesn't make sense, try different ways of sampling randomly from z.