# Homework 2 — Training Multilayer Perceptrons

## 1 Scope

This assignment is on implementing backpropagation (BP) and training multilayer perceptrons (MLPs) with different optimizers.

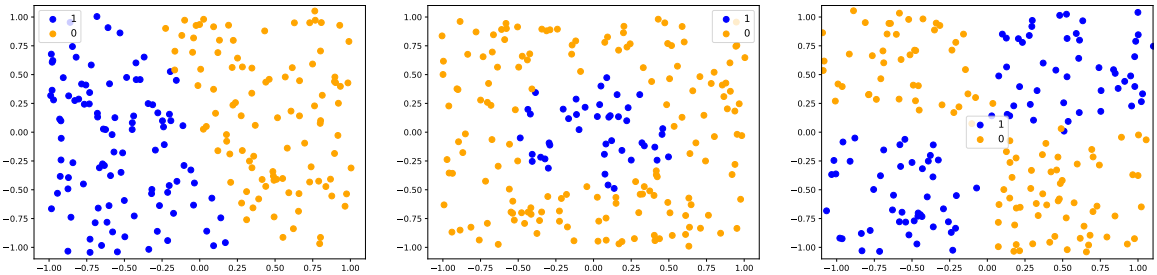
## 2 Instructions

- Deadline for this assignment is **February 9** via Gradescope
- The submission portal will remain open till end of February. Do keep in mind the late submission policy (Check slides from first lecture)
- Your submission has two parts: (i) a PDF report that details all the requested deliverables and (ii) the code for the assignment. You will upload the PDF report in gradescope, marking which parts of the report corresponds to which deliverable. The code will be uploaded separately as a collection of .py files, and we will run "software similarity" software on it to detect violations of the integrity policy.
- You can discuss this HW with others, but you are **not** allowed to share, borrow, copy or look at each other's codes.
- All code that you submit must be yours (except for the starter code that we provide). Specifically, you are **not** allowed to use any software packages or code from the internet or other resources, except for a basic python installation of `numpy, random` and `matplotlib`.
- **We are intentionally not disclosing achievable results for each part.** Use your intuition and what you have learnt from class to adapt. For example, vanilla gradient descent will take a long time to converge. Adam will converge a lot faster. More neurons in a network will often lead to better training results (at the risk of overfitting). I would encourage you to avoid posting on piazza to check if your answer is good enough. Instead, try a few different things and see what you get, and learn from it. Add it to your report.

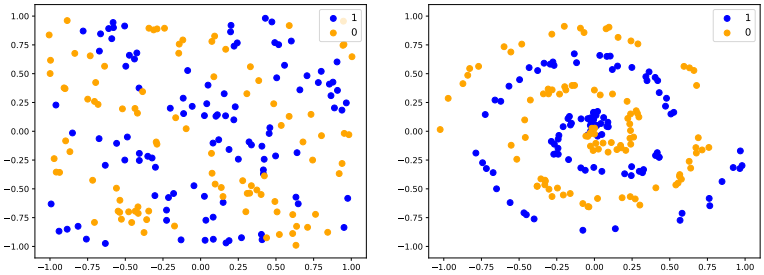
## 3 Problem Set

In this homework, you will implement training and testing of MLPs with numpy. Specifically, you will replicate several basic features of the demo at tensorflow-playground-website https://playground.tensorflow.org/.

*Training and validation dataset.* In the starter code, `numpyNN.py` we have provided some datasets for you to use. There are five sets: `linear-separable, circle, XOR, sinuoid, and swiss-roll.` The function `sample_dataset` in the starter code will allow you to obtain samples from each. This code will give you samples from a 2-dimensional input $(x_i, y_i)$ to a one-dimensional output $d_i$. The inputs $(x_i, y_i)$ are both real-valued with $(x_i, y_i) \in [-1, +1]^2$. The output $d_i$ is binary-valued, $d_i \in \{0, 1\}$. The Figure 1 shows the five datasets you will work with.



((a)) Linear-separable ((b)) Circle ((c)) XOR



((d)) Sinuoid ((e)) Swiss-roll

Figure 1: Datasets

**(Deliverable 1: Code)** Your first deliverable is an implementation of a codebase that trains a generic MLP. Your implementation should provide the following capabilities.

- Ability to handle any number of layers with different number of perceptrons per layer.
- The following activation functions: ReLU, Sigmoid, tanh, and linear
- Two possible loss functions to train with: $\ell_2^2$ (L2) and cross-entropy (CE)
- Intializing MLP parameters with Xavier or He initialization
- The forward pass of the MLP
- The backward pass (aka BP)
- Using basic gradient descent to train a network; this can have constant step size or a decaying step size
- Implementing gradient descent with momentum to train a network
- Implementing ADAM to train a network

The entry code to this should have switches to select across various options. You are encouraged to follow good software engineering practices; but this is not mandatory. For example, you can refer to the following architecture:

```
mlp = initialize_mlp(num_layers, num_width, opt_act, opt_init)
train_mlp(mlp, training_data, num_epoch, opt_loss, opt_optim)
test_mlp(mlp, test_data, opt_loss)

def train_mlp(mlp, training_data, num_epoch, opt_loss, opt_optim):
```

```
    for epoch in range(num_epoch):
        y, z = forward(mlp, training_data.x)
        y_L = y[num_layers-1]
        train_loss, loss_div = loss(y_L, training_data.y, opt_loss)
        dW, db = backward(mlp, training_data.x, y, z, loss_div)
        update_mlp(mlp, dW, db, opt_optim)
        ...

def test_mlp(mlp, test_data,  opt_loss):
    y, _ = forward(mlp, test_data.x)
    test_loss, _ = loss(y[num_layers-1], test_data.y, opt_loss)
    ...
```

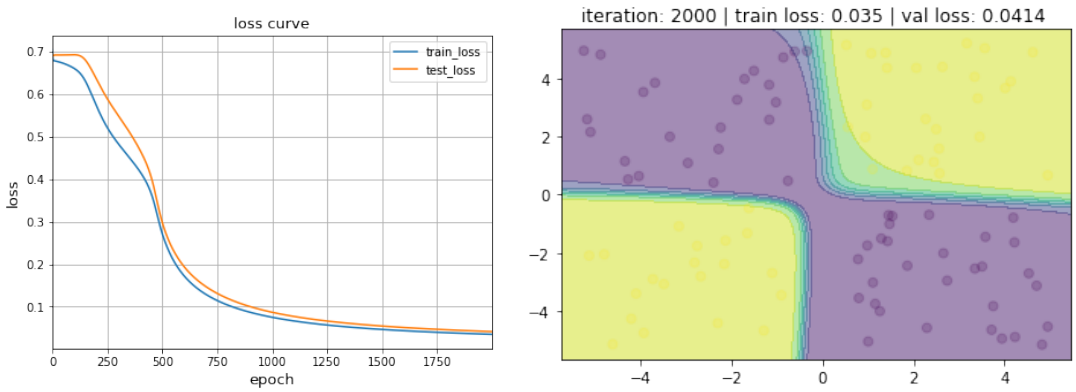
The deliverable for this part is the code itself. This is the only deliverable that requires you to submit your code.

Your PDF report **need not** contain this code. As noted earlier, you will upload your code separately.

**(Deliverable 2: Linear separable dataset)** Train an MLP on the `linear-separable` dataset. Sample 200 points for training and 200 points for validation. *Use basic gradient descent to train an MLP with a single hidden layer that has a single perceptron under L2 loss.* Choose the activation functions on each layer carefully!

Your deliverable should include a plot of training/validation error with epochs as well as a visualization of the final decision boundary you obtain.

Specifically, you need to implement a codebase to produce a visualization similar to Figure 2. A similar visualization can also be found at tensorflow-playground-website https://playground.tensorflow.org/. The starter code provided in `numpyNN.py` has functions `plot_loss` and `plot_decision_boundary` that provide a template for implementing these.




((a)) Loss curve ((b)) Decision boundary

Figure 2: Example results for XOR dataset

**(Deliverable 3: XOR problem)** Train an MLP on the `XOR` dataset. Sample 200 points for training and 200 points for validation. You have complete freedom in the design of the MLP including the number of layers, the number of perceptrons per layer, the activations for each layer, loss function; however the input has to be two-dimensional $(x, y)$ and output is one-dimensional. You are welcome to use any of the optimizers you have implemented. Figure 2 shows a result on this XOR dataset.

As with Deliverable 2, you should include a plot of training/validation error with epochs as well as a visualization of the final decision boundary you obtain. You should also detail the architecture of the specific network used (layers, activations, cost function).

**(Deliverable 4: Differences in cost function)** This deliverable looks at the difference in training a regressor vs. a classifier. Both MLPs should share the same architecture, except for activations (which you need to select carefully) and the loss function—which is L2 for one and CE for the other. Train both MLPs on the `circle` dataset, sampling 200 points for training and 200 points for validation.

Show a plot of training/val error with epochs as well as final decision boundaries for each of the two MLPs you have trained. You should also detail the architecture of specific network used (layers, activations, cost function).

**(Deliverable 5: Differences in optimizers)** This deliverable looks at the differences in training with vanilla gradient descent, gradient descent with momentum, and ADAM. You will use the the `sinusoid` dataset, sampling 200 points for training and 200 points for validation. Design an MLP for this dataset; you have full freedom in network parameters and loss functions but the network must be at least four layers deep. You will train it thrice, once for each of the three optimization strategies.

For each of the three optimizers, show a plot of training/val error with epochs as well as final decision boundaries. It is normal that vanilla gradient descent will either not converge to a good solution (ditto for descent with momentum); this is as expected. Explore various options (size of the network, for example) to get each of the optimizers to work as well as you can.

**(Deliverable 6: Swiss roll)** Train an MLP to classify on the `swiss-roll` dataset, sampling 200 points each for training and validation. You have complete freedom in designing the network, but it must at least be four layers deep. (Adam will likely get you the best results).

In addition to detailing the specifics of the network you used, show a plot of training/val error with epochs as well as final decision boundaries.

**(Deliverable 7: Non-linear embeddings)** In machine learning, embedding the input in a non-linear space often dramatically simplifies the learning problem. For example, the `circles` dataset becomes linearly separable if we include $x_i^2 + y_i^2$ as an input variable. (Do test this!)

For the the `XOR` dataset, design an MLP with the fewest possible perceptrons along with a single additional non-linear function of inputs. Implement the code to test this. Show training/validation error, as well as the decision boundary (visualized in $(x, y)$ as before). Repeat for `swiss-roll` dataset. Feel free to use more than one nonlinear function to the input.

For each dataset, in addition to detailing the specifics of the network you used, show a plot of training/val error with epochs as well as final decision boundaries. For plotting the decision boundaries, feel free to modify the `plot_decision_boundary` function.

PS: We dont have an optimal solution here. More interested in seeing what you come up. So feel free to explore.