# Project 1: Backpropagation

CSE 849 Deep Learning (Spring 2025)

Gautam Sreekumar
Instructor: Zijun Cui

February 6, 2025

In this assignment, you will implement backpropagation and gradient descent (GD). Backpropagation is an efficient method to compute gradients in neural networks composed of modular units. GD is an optimization method that uses gradients to (typically) minimize some objective.

You will use the conda environment from Project 0 to complete this assignment. If you face any issues with using the `.yml` file from the previous assignment, you can run the following commands in your terminal.

```
conda create -n project-1
conda activate project-1
conda install -c pytorch matplotlib pytorch
```

We will refer to the lecture slides titled "04-Forward_and_Backward_Propagation" in this homework.

## 1 Gradient Descent

You will implement basic gradient descent optimization to a 2D vector variable $\boldsymbol{x}$. You are given starter codes `q1.py`, and the major steps are instructed below. Your task is to follow the steps and complete `q1.py`.

- Create a 2D PyTorch tensor $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ with initial value set to an uniformly sampled value in $[-5, 5] \times [-5, 5]$.
- Calculate the function $y = r^2(\sin^2(6\theta + 2r) + 1)$ where $\theta = \tan^{-1}(x_2/x_1)$ and $r = \sqrt{x_1^2 + x_2^2}$. This will result in a spiral with ridges and valleys.
- Calculate the gradient $\frac{dy}{d\boldsymbol{x}}$ analytically (pen-and-paper method) by applying chain rule. Using these gradients, update $\boldsymbol{x}$ to minimize $y$. $\boldsymbol{x}$ is updated at time step $t$ as $\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - \lambda \frac{dy}{d\boldsymbol{x}}$, where $\lambda$ is the learning rate.
- Save the gradient vectors at every time step and plot the resulting trajectory over the loss landscape using `contourf` and arrow functions from matplotlib. As you will see, the minima is attained when $\boldsymbol{x}$ reaches the origin.
- With a fixed starting position of $\boldsymbol{x}$ (by fixing the random seed), repeat this experiment for $\lambda = 10^{-4}, 1e-3, 1e-2, 1e-1, 1$.
- Include the plots in your report and describe what you observe for various step sizes.

## 2 Backpropagation

To apply gradient descent (above) to train a neural network, learnable parameters within the neural network become variables and we need to compute gradients for them. Backpropagation is an effective

way of calculating gradients in settings where the overall function is modeled as a combination of computational units with analytically tractable derivatives. Since neural networks are constructed using modular computational units (such as linear layers, activation functions, etc.), backpropagation is an obvious choice to train neural networks. In this task, we will implement backpropagation for a multi-layer perceptron.

## 2.1 Setup

Your task is to predict the annual rainfall $y$ (say, relative to average annual rainfall in Michigan) at some location in Michigan given by the 2D coordinates $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$. You must train a simple MLP with **three** stacked linear layers. The $i^{\text{th}}$ linear layer consists of a weight matrix $\boldsymbol{W}^{(i)} \in \mathbb{R}^{d_{i-1} \times d_i}$ and a bias vector $\boldsymbol{b}^{(i)} \in \mathbb{R}^{d_i}$ where $d_j$ denotes the dimensionality of the output from $j^{\text{th}}$ layer. For $j = 0$, $d_0 = 2$, the input dimension. For the output layer, $d_3$ is 1. Between each linear layer, we will have a non-linear activation function $\sigma$ to model non-linear functions. $\sigma$ acts elementwise on vectors, i.e., $\sigma(\boldsymbol{x}) = \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_d) \end{bmatrix}$. The model to predict $y$ from a given test sample $\boldsymbol{x}$ is given as,

$$\boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \tag{1}$$

$$\boldsymbol{h}^{(1)} = \sigma(\boldsymbol{a}^{(1)}) \tag{2}$$

$$\boldsymbol{a}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)} \tag{3}$$

$$\boldsymbol{h}^{(2)} = \sigma(\boldsymbol{a}^{(2)}) \tag{4}$$

$$\hat{y} = \boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)} + b^{(3)} \tag{5}$$

To find the optimal parameters for this model, we have access to a training dataset comprising $n$ data samples $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$. We will arrange the inputs from the samples into an input data matrix $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix} \in \mathbb{R}^{n \times d_0}$. The ground truth outputs are stacked to form a vector $\boldsymbol{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$.

The optimal parameters are obtained by minimizing the mean squared prediction error on this training data. The training error for a given parameter vector $\boldsymbol{\Theta}$ (comprising all weight and bias terms) over the entire dataset is given by,

$$\ell(\boldsymbol{y}, \hat{\boldsymbol{y}}; \boldsymbol{\Theta}) = \frac{1}{n}\sum_i \ell(y_i, \hat{y}_i; \boldsymbol{\Theta}) = \frac{1}{n}\sum_i (y_i - \hat{y}_i)^2 \tag{6}$$

In eq. (6), $\hat{y}_i$ is a function of all parameters $\boldsymbol{\Theta}$ but we do not write it to keep the expression short and simple. We will optimize the parameter values using GD similar to the previous question. However, instead of analytically deriving the gradients like in the previous question, we will compute the gradients using backpropagation.

## 2.2 Overall Pipeline

We will build and train our MLP as follows. More detailed codes can be found in starter codes `q2.py`.

```
class MLP:
  def __init__(self, d_in, d_out, hid_dim, lr):
    self.layer1 = Linear(d_in, hid_dim, lr) # your Linear layer
    self.act1 = ReLU() # your ReLU layer
    self.layer2 = Linear(hid_dim, hid_dim, lr) # your Linear layer
    self.act2 = ReLU() # your ReLU layer
    self.layer3 = Linear(hid_dim, d_out, lr) # your Linear layer

  def forward(self, X):
```

```
    a1 = self.layer1.forward(X)
    h1 = self.act1.forward(a1)
    a2 = self.layer2.forward(h1)
    h2 = self.act2.forward(a2)
    yhat = self.layer3.forward(h2)

    return yhat

  def backward(self, grad):
    grad = self.layer3.backward(grad)
    grad = self.act2.backward(grad)
    grad = self.layer2.backward(grad)
    grad = self.act1.backward(grad)
    grad = self.layer1.backward(grad)


lr = # some learning rate
model = MLP(d_in, d_out, hid_dim, lr) # your MLP
loss_fn = MSELosS() # your loss function

for X, y in train_dataloader: # your data loader
  out = model.forward(X)
  loss = loss_fn(out, y)
  grad = loss_fn.backward()
  model.backward(grad)
```

## 2.3 Tasks

### 2.3.1 Design Your Functions

To complete the pipeline shown above, in this assignment, you will build three computational functions: `Linear`, `ReLU`, and `MSELoss`. The structure of each function class will be as follows:

```
class Layer:
  def __init__(self, lr, *args, **kwargs):
    self.weight = # initialize to some value from normal distribution
    self.lr = lr # learning rate
    self.eval_mode = False # are-we-training-or-evaluating flag

  def forward(self, x):
    out = # compute the output of this linear using self.weight and x
    if not self.eval:
      self.grad_comps = # compute the components required to calculate the gradients

    return out

  def backward(self, grad):
    # grad is the gradient of the loss w.r.t. the outputs of this layer's forward function
    self.grad = # compute the gradients for self.weight using grad and self.grad_comps

    # update the weights of this model using the estimated gradients.
    # self.weights -= self.lr * self.grad

    grad_for_next = # compute the gradients of this layer's input in self.forward

    return grad_for_next
```

```
# The remaining functions are included in the base class in layers.py
def eval(self):
  self.eval_mode = True

def train(self):
  self.eval_mode = False
```

**Meaning of the variables in `Layer` class:**

1. `forward` function is called to compute the output of this layer. For example, if this is a `Linear` layer, then the input is multiplied with the weights, and the bias terms are added to it.
2. In `forward` function, you will also compute `self.grad_comps` that stores the intermediate results that are needed to compute the gradients during the backward pass. This terms corresponds to $\frac{\partial z_j^{(N)}}{\partial w_{ij}^{(N)}}$ in page 22 of the lecture slides. You can also check section 6.1.
3. `backward` function computes the gradients `self.grad` for all parameters of the layer. It takes in as argument `grad`, which is the gradient of the loss terms w.r.t. `out` of the corresponding `forward` function. This is equivalent to $\frac{\partial Div}{\partial z_j^{(N)}}$ in page 22 of the lecture slides. This is provided by the succeeding layer in the forward pass.
4. `backward` method also calculates the gradients of the loss w.r.t. the inputs of the corresponding `forward` method and store it in `grad_for_next`. This will be used by the preceding unit to compute its gradients. In other words, `grad_for_next` from the current layer becomes `grad` for the preceding layer in the backward pass.
5. Be careful that forward and backward passes have opposite orders of computation.

Note that the above structure can be simplified for ReLU layer and MSELoss. For ReLU layer, weights related definitions are not needed. For MSELoss function, since the gradient computation starts with `loss_fn`, we will not pass any `grad` to it. Refer to starter codes `layers.py` for details.

### 2.3.2 Load The Data

Load the data from the file "`Project1_data.pt`" using the provided codes in the starter codes.

### 2.3.3 Train The Model

Train the model with $d_1 = d_2 = 100$. Plot training and validation losses using the codes provided in the starter codes. Save the prediction results as `q2_ytest.txt` file.

# 3 Important Tips

1. For this assignment, the accuracy of the gradients is more important than the speed or the efficiency of your code. **You are encouraged to start with element-wise gradient and feel free to use `for` loops to iterate over various indices. Once you ensure the gradients are accurate, you can try to make your code more efficient, i.e., vector formulations.**
2. To verify the accuracy of your algorithm, use simple test cases where the gradients are known. For example, if the training objective (output of the loss function) is to minimize the norm of the output of the linear layer, the gradients must direct towards the origin for all parameters.
3. Although our description calculated gradients for a single sample, you will compute gradients for mini-batches in the assignment. However, this results in only slight modification of your algorithm. For example, in calculating the mean squared error, you need to scale the loss for each sample by 1/batch_size.

4. **Do not** collapse the batch dimension when you pass the intermediate results for gradient computation to the preceding unit. Gradients are only averaged right before using them to update the parameters.

# 4 Submission Deliverables and Formats

## 4.1 Deliverables

- Question 1: Gradient Descent
  1. Completed Python file `q1.py`.
  2. Include the trajectories for all $\lambda$s and seeds in your report. Describe what you observe for various values of $\lambda$s. The report must be in **PDF format**. No handwritten or Word documents.
- Question 2: Backpropagation
  1. Completed Python files `layers.py` and `q2.py`.
  2. Plot training and validation losses similar to Project 0. Include them in your report.
  3. Similar to Project 0, run your best model based on validation error on the test data and include the results as a `txt` file.

## 4.2 Formats

Submit a folder named `first name_last name_project_1` using the same subfolders as below:

1. code
   (a) q1.py
   (b) q2.py
   (c) layers.py
2. results
   (a) report.pdf
   (b) q2_ytest.txt

# 5 Grading Policy

1. In total, 100 points.
2. 80 points for submission completeness and functionality. q1.py (10 pts), q2.py (10 pts), layers.py (30 pts), report (20.pt), q2_ytest.txt (10 pts). Your output is automatically graded using a Python evaluation script. Therefore, make sure to follow the details in the question accurately.
3. 20 points for the accuracy of your prediction. We have a testing error based on our implementation. You will be scored based on how close to/below your prediction error is to ours.

# 6 Further Reading: How does backpropagation work?

We will see how to write a general backpropagation system before applying it to a neural network. Let $\boldsymbol{x} \in \mathbb{R}^{d_x}$ denote the vector input of dimension $d_x$. Our output is given by a composition of $m$ functions $f_1$, $f_2$ and so on, parameterized by matrices $\boldsymbol{W}^{(1)}$, $\boldsymbol{W}^{(2)}$, etc. The output is a vector $\hat{\boldsymbol{y}} \in \mathbb{R}^{d_y}$, given by $\hat{\boldsymbol{y}} = f_m(f_{m-1}(\ldots f_2(f_1(\boldsymbol{x}; \boldsymbol{W}^{(1)}); \boldsymbol{W}^{(2)}) \ldots; \boldsymbol{W}^{(m-1)}); \boldsymbol{W}^m)$. Note that each parameter matrix only acts in its corresponding function. You can think of each function as a modular computational unit that calculates some intermediate result and passes it on to the next unit.

**Note:** The above construction leads to a computational graph that takes the form of a simple chain graph: $f_1 \to f_2 \to \dots f_m$. However, not every neural network has such simple computational graph. As an example, it is possible that $f_m$ takes the intermediate results from $f_1$ and $f_{m-1}$ as its input, resulting in an additional edge from $f_1$ to $f_m$.

We also have a vector-to-scalar function $\ell : \mathbb{R}^{d_y} \to \mathbb{R}$ that gives the loss value for our output $\hat{\boldsymbol{y}}$ (usually w.r.t. some reference/ground truth). Our objective is to calculate the gradient of this loss function w.r.t. various parameter matrices. We will use $\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}$ as the example.

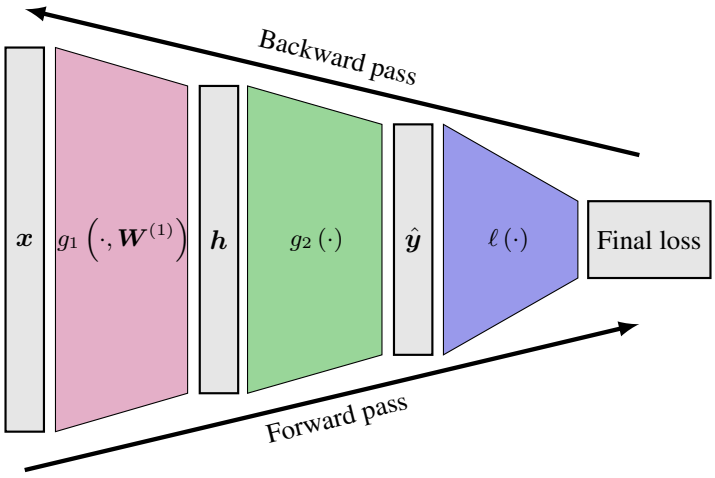


Figure 1: A general framework for understanding backpropagation

Since we are interested in gradients w.r.t. $\boldsymbol{W}^{(1)}$, we can write $\hat{\boldsymbol{y}} = g_2(\boldsymbol{h})$ where $\boldsymbol{h} = g_1(\boldsymbol{x}; \boldsymbol{W}^{(1)}) \in \mathbb{R}^{d_h}$ where $g_1$ is equivalent to $f_1$ and $g_2$ is the composition of the functions $f_2, \dots, f_m$ that do not involve $\boldsymbol{W}^{(1)}$. See fig. 1. This formulation lets us focus on $g_1$ as a computational unit unaffected by other units. We choose to keep $g_2$ and $\ell$ to highlight the distinction between the neural network and the loss function, although this distinction is mathematically not important. Now we can write the expression for $\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}$ using this formulation.

$$\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}} = \sum_i \frac{\partial \ell}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial \boldsymbol{W}^{(1)}}$$

This above equation is due to $\boldsymbol{W}^{(1)}$ affecting $\ell$ through possibly all elements of $\hat{\boldsymbol{y}}$. Then,

$$\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}} = \sum_i \frac{\partial \ell}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial \boldsymbol{W}^{(1)}} = \sum_i \frac{\partial \ell}{\partial \hat{y}_i} \sum_j \left( \frac{\partial \hat{y}_i}{\partial h_j} \frac{\partial h_j}{\partial \boldsymbol{W}^{(1)}} \right) \tag{7}$$

Here again, we assume that every element of $\hat{\boldsymbol{y}}$ is affected by every element of $\boldsymbol{h}$. The colors denote that these gradients are within the polygons of corresponding colors in fig. 1.

In our formulation, $\hat{\boldsymbol{y}} = g_2(g_1(\boldsymbol{x}; \boldsymbol{W}^{(1)}))$, $g_1$ is usually an atomic/standalone computational unit such as a multiplication with a parameter matrix, non-linear activation, bias addition, convolution, etc. and $g_2$ denotes everything else that follows $g_1$ until the output is generated. Therefore, the analytic expression for $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}^{(1)}}$ is known a priori and can be calculated during the forward pass itself. From eq. (7), we can observe that once we have calculated $\boldsymbol{h}$, we do not require $\boldsymbol{W}^{(1)}$ to estimate $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}}$ or $\frac{\partial \ell}{\partial \hat{\boldsymbol{y}}}$. That is, the function $g_1$ is no longer required once its output $\boldsymbol{h}$ is ready. Therefore, the remaining terms $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}}$ and $\frac{\partial \ell}{\partial \hat{\boldsymbol{y}}}$ can be calculated by the computational units that form $g_2$ using $\boldsymbol{h}$. This modularity is the key principle of backpropagation and in this assignment, we will implement computational units that will handle both forward pass and backpropagation.

## 6.1 Computing the required components during the forward pass

To demonstrate how to compute $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}^{(1)}}$, we will work out a specific case where $f_1$ is a linear function that multiplies the input vector $\boldsymbol{x}$ with its parameter matrix $\boldsymbol{W}^{(1)}$. Since $g_1$ is equivalent to $f_1$, we have

$\boldsymbol{h} = \boldsymbol{W}^{(1)}\boldsymbol{x}$, i.e., $h_i = \sum_k W_{ik}x_k$. Since $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}^{(1)}}$ is a tensor gradient of order 3 (due to vector-over-matrix derivative), three indices are required to denote each element of the gradient.

$$\left[\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}^{(1)}}\right]_{ijk} = \frac{\partial h_i}{\partial W^{(1)}_{jk}}$$

$$\frac{\partial h_i}{\partial W^{(1)}_{jk}} = \begin{cases} x_k, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \quad [\text{since } h_i = \sum_k W_{ik}x_k] \tag{8}$$

For visualization purposes, $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}^{(1)}}$ is a mostly empty cube with a diagonal plane. This tensor can be computed during the forward pass. Similarly, the components for calculating gradients for various computational units can be calculated using their corresponding analytic expressions during the forward pass itself.

**Computing the gradients during backward:** In addition to $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{W}^{(1)}}$ calculated during the forward pass, $g_1$ also needs to use $\frac{\partial \ell}{\partial \boldsymbol{y}}$ and $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{h}}$ to calculate $\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}$. As mentioned before, these terms are provided to $g_1$ by $g_2$. eq. (7) can be rewritten as follows,

$$\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}} = \sum_i \frac{\partial \ell}{\partial y_i} \sum_j \left( \frac{\partial y_i}{\partial h_j} \frac{\partial h_j}{\partial \boldsymbol{W}^{(1)}} \right) = \sum_j \left( \sum_i \frac{\partial \ell}{\partial y_i} \frac{\partial y_i}{\partial h_j} \right) \frac{\partial h_j}{\partial \boldsymbol{W}^{(1)}} = \sum_j \frac{\partial \ell}{\partial h_j} \frac{\partial h_j}{\partial \boldsymbol{W}^{(1)}} \tag{9}$$

That is, for the computational unit $g_1$ to calculate the gradient, it requires the gradient of the loss term w.r.t. each of its output $\frac{\partial \ell}{\partial h_j}$. The required gradient $\frac{\partial \ell}{\partial \boldsymbol{W}^{(1)}}$ is the dot product between gradient $\frac{\partial \ell}{\partial h_j}$ passed on from $g_2$ and the gradient $\frac{\partial h_j}{\partial \boldsymbol{W}^{(1)}}$ calculated during the forward pass by $g_1$ along the output dimension of $g_1$.

**Helping preceding units to compute their gradients:** Now consider a more general case where $g_1$ is not the first computational unit. In this case, it must pass on the gradients w.r.t. its input to its preceding computational units so that they may compute the gradients for their parameters. In this case, $g_1$ must calculate $\frac{\partial \ell}{\partial \boldsymbol{x}}$ since $\boldsymbol{x}$ is its input (and therefore the preceding unit's output). Following eq. (9),

$$\frac{\partial \ell}{\partial \boldsymbol{x}} = \sum_j \frac{\partial \ell}{\partial h_j} \frac{\partial h_j}{\partial \boldsymbol{x}}$$

where $\frac{\partial h_j}{\partial \boldsymbol{x}}$ is a vector whose $i^{\text{th}}$ element is given by $\frac{\partial h_j}{\partial x_i} = W^{(1)}_{ji}$.