Homework 1: Backpropagation

DS-GA 1008 Deep Learning

Fall 2025

The goal of homework 1 is to help you understand the common techniques used in Deep Learning and how to update network parameters by the backpropagation algorithm.

Part 1 has two sub-parts, 1.1, 1.2, 1.3 majorly deal with the theory of backpropagation algorithm whereas 1.4 is to test conceptual knowledge on deep learning. For part 1.2 and 1.3, you need to answer the questions with mathematical equations. You should put all your answers in a PDF file and we will not accept any scanned hand-written answers. It is recommended to use LaTeX.

For part 2, you need to program in Python. It requires you to implement your own forward and backward pass without using autograd. You need to submit your `mlp.py` file for this part.

The due date of homework 1 is **23:55 ET of 09/21**. Submit the following files in a zip file `your_net_id.zip` through Brightspace course page:

- `theory.pdf`
- `mlp.py`
- `gd.py`

The following behaviors will result in penalty of your final score:

1. 5% penalty for submitting your files without using the correct format. (including naming the zip file, PDF file or python file wrong, or adding extra files in the zip folder, like the testing scripts from part 2).
2. 20% penalty for late submission within the first 24 hours. We will not accept any late submission after the first 24 hours.
3. 20% penalty for code submission that cannot be executed using the steps we mentioned in part 2. So please test your code before submit it.

1 Theory (50pt)

To answer questions in this part, you need some basic knowledge of linear algebra and matrix calculus. Also, you need to follow the instructions:

1. Every provided vector is treated as column vector.
2. **IMPORTANT:** You need to use the numerator-layout notation for matrix calculus. Please refer to Wikipedia about the notation. Specifically, $\frac{\partial y}{\partial x}$ is a row-vector whereas $\frac{\partial y}{\partial x}$ is a column-vector.[1]

[1] `https://en.wikipedia.org/wiki/Matrix_calculus#Numerator-layout_notation`

3. You are only allowed to use vector and matrix. You cannot use tensor in any of your answer.

4. Missing transpose are considered as wrong answer.

1.1 Two-Layer Neural Nets

You are given the following neural net architecture:

$$\text{Linear}_1 \;\rightarrow\; f \;\rightarrow\; \text{Linear}_2 \;\rightarrow\; g$$

where $\text{Linear}_i(x) = W^{(i)}x + b^{(i)}$ is the i-th affine transformation, and f, g are element-wise nonlinear activation functions. When an input $x \in \mathbb{R}^n$ is fed to the network, $\hat{y} \in \mathbb{R}^K$ is obtained as the output.

1.2 Regression Task

We would like to perform regression task. We choose $f(\cdot) = 5(\cdot)^+ = 5\,\text{ReLU}(\cdot)$ and g to be the identity function. To train this network, we choose MSE loss function $\ell_{\text{MSE}}(\hat{y}, y) = \|\hat{y} - y\|^2$, where y is the target output.

(a) (1pt) Name and mathematically describe the 5 programming steps you would take to train this model with PyTorch using SGD on a single batch of data.

Let B be the batch size.

1) Forward pass: generate prediction

$$\hat{Y} = g(\text{Linear}_2(f(\text{Linear}_1(X))))$$

2) Compute loss (single batch MSE)

$$\mathcal{L}(\theta; X, Y) \;=\; \tfrac{1}{B}\,\|\hat{Y} - Y\|_F^2.$$

3) Zero gradients

$$\nabla_\theta \mathcal{L} \;\leftarrow\; 0.$$

4) Compute & accumulate gradients

$$\nabla_\theta \mathcal{L}(\theta; X, Y) \quad \text{via chain rule (autograd).}$$

5) SGD update (step toward negative gradient)

$$\theta \;\leftarrow\; \theta - \eta\,\nabla_\theta \mathcal{L}(\theta; X, Y).$$

(b) (4pt) For a single data point (x, y), write down all inputs and outputs for forward pass of each layer. You can only use variables $x, y, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ in your answer. (note that $\text{Linear}_i(x) = W^{(i)}x + b^{(i)}$).

Linear$_1$:	Input: x,	Output: $W^{(1)}x + b^{(1)}$
f**:**	Input: $W^{(1)}x + b^{(1)}$,	Output: $5\big(W^{(1)}x + b^{(1)}\big)^+$
Linear$_2$:	Input: $5\big(W^{(1)}x + b^{(1)}\big)^+$,	Output: $W^{(2)}\,5\big(W^{(1)}x + b^{(1)}\big)^+ + b^{(2)}$
g**:**	Input: $W^{(2)}\,5\big(W^{(1)}x + b^{(1)}\big)^+ + b^{(2)}$,	Output: $W^{(2)}\,5\big(W^{(1)}x + b^{(1)}\big)^+ + b^{(2)}$

(c) (6pt) Write down the gradients calculated from the backward pass. You can only use the following variables: $x, y, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \frac{\partial \ell}{\partial \hat{y}}, \frac{\partial z_2}{\partial z_1}, \frac{\partial \hat{y}}{\partial z_3}$ in your answer, where $z_1, z_2, z_3, \hat{y}$ are the outputs of $\text{Linear}_1, f, \text{Linear}_2, g$.

(Chain rule to z_3) $$\frac{\partial \ell}{\partial z_3} = \frac{\partial \ell}{\partial \hat{y}}\,\frac{\partial \hat{y}}{\partial z_3}$$

(Params of Linear_2) $$\nabla_{W^{(2)}}\ell = \left(\frac{\partial \ell}{\partial z_3}\right)^\top z_2^\top, \qquad \nabla_{b^{(2)}}\ell = \left(\frac{\partial \ell}{\partial z_3}\right)^\top$$

(Chain rule to z_2) $$\frac{\partial \ell}{\partial z_2} = \frac{\partial \ell}{\partial z_3}\,\underbrace{\frac{\partial z_3}{\partial z_2}}_{=W^{(2)}} = \frac{\partial \ell}{\partial z_3}\,W^{(2)}$$

(Chain rule through nonlinearity) $$\frac{\partial \ell}{\partial z_1} = \frac{\partial \ell}{\partial z_2}\,\frac{\partial z_2}{\partial z_1}$$

(Params of Linear_1) $$\nabla_{W^{(1)}}\ell = \left(\frac{\partial \ell}{\partial z_1}\right)^\top x^\top, \qquad \nabla_{b^{(1)}}\ell = \left(\frac{\partial \ell}{\partial z_1}\right)^\top$$

(Optional, input gradient) $$\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial z_1}\,\underbrace{\frac{\partial z_1}{\partial x}}_{=W^{(1)}} = \frac{\partial \ell}{\partial z_1}\,W^{(1)}.$$

(d) (2pt) Show us the elements of $\frac{\partial z_2}{\partial z_1}$, $\frac{\partial \hat{y}}{\partial z_3}$ and $\frac{\partial \ell}{\partial \hat{y}}$ (be careful about the dimensionality)?

$$\left[\frac{\partial z_2}{\partial z_1}\right]_{ij} = \frac{\partial (z_2)_i}{\partial (z_1)_j} = \begin{cases} 5, & i = j \text{ and } (z_1)_i > 0, \\ 0, & \text{otherwise}, \end{cases} \qquad \frac{\partial z_2}{\partial z_1} \in \mathbb{R}^{m \times m}.$$

$$\left[\frac{\partial \hat{y}}{\partial z_3}\right]_{ij} = \frac{\partial \hat{y}_i}{\partial (z_3)_j} = \begin{cases} 1, & i = j, \\ 0, & i \neq j, \end{cases} \qquad \frac{\partial \hat{y}}{\partial z_3} = I_K \in \mathbb{R}^{K \times K}.$$

$$\left[\frac{\partial \ell}{\partial \hat{y}}\right]_j = \frac{\partial \ell}{\partial \hat{y}_j} = 2(\hat{y}_j - y_j), \qquad \frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y)^\top \in \mathbb{R}^{1 \times K}.$$

1.3 Classification Task

We would like to perform multi-class classification task, so we set $f = \tanh$ and $g = \sigma$, the logistic sigmoid function $\sigma(z) \doteq (1 + \exp(-x))^{-1}$.

(a) (2pt + 3pt + 1pt) If you want to train this network, what do you need to change in the equations of (b), (c) and (d), assuming we are using the same MSE loss function.

(b) Forward pass for a single point (x, y) (*changed where noted*)

Linear$_1$:

Input: x

Output: $W^{(1)}x + b^{(1)}$

f **(=** $\tanh$**):**

Input: $W^{(1)}x + b^{(1)}$

Output: $\tanh\big(W^{(1)}x + b^{(1)}\big)$ (*changed*)

Linear$_2$:

Input: $\tanh\big(W^{(1)}x + b^{(1)}\big)$

Output: $W^{(2)} \tanh\big(W^{(1)}x + b^{(1)}\big) + b^{(2)}$

g **(=** σ**):**

Input: $W^{(2)} \tanh\big(W^{(1)}x + b^{(1)}\big) + b^{(2)}$

Output: $\hat{y} = \sigma\Big(W^{(2)} \tanh\big(W^{(1)}x + b^{(1)}\big) + b^{(2)}\Big)$ (*changed*)

(c) Backward pass (chain rule form) (*same structure; local Jacobians changed*)

$$\frac{\partial \ell}{\partial z_3} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3}, \qquad \nabla_{W^{(2)}} \ell = \Big(\frac{\partial \ell}{\partial z_3}\Big)^\top z_2^\top, \qquad \nabla_{b^{(2)}} \ell = \Big(\frac{\partial \ell}{\partial z_3}\Big)^\top,$$

$$\frac{\partial \ell}{\partial z_2} = \frac{\partial \ell}{\partial z_3} W^{(2)}, \qquad \frac{\partial \ell}{\partial z_1} = \frac{\partial \ell}{\partial z_2} \frac{\partial z_2}{\partial z_1},$$

$$\nabla_{W^{(1)}} \ell = \Big(\frac{\partial \ell}{\partial z_1}\Big)^\top x^\top, \qquad \nabla_{b^{(1)}} \ell = \Big(\frac{\partial \ell}{\partial z_1}\Big)^\top.$$

Local Jacobians now are:

$$\frac{\partial z_2}{\partial z_1} = \operatorname{diag}\big(1 - \tanh^2(z_1)\big) \ \text{(changed)}, \qquad \frac{\partial \hat{y}}{\partial z_3} = \operatorname{diag}\big(\hat{y} \odot (1 - \hat{y})\big) \ \text{(changed)}$$

With MSE, $\dfrac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y)^\top$ (unchanged). $\odot$ is the Hadamard (element-wise) product.

(d) Elementwise forms (with dimensions) (*changed where noted*)

$$\Big[\tfrac{\partial z_2}{\partial z_1}\Big]_{ij} = \frac{\partial (z_2)_i}{\partial (z_1)_j} = \begin{cases} 1 - \tanh^2\big((z_1)_i\big) = 1 - (z_2)_i^2, & i = j, \\ 0, & i \neq j, \end{cases} \quad \in \mathbb{R}^{m \times m} \ \text{(changed)}.$$

$$\left[\frac{\partial \hat{y}}{\partial z_3}\right]_{ij} = \frac{\partial \hat{y}_i}{\partial (z_3)_j} = \begin{cases} \hat{y}_i(1-\hat{y}_i), & i=j, \\ 0, & i \neq j, \end{cases} \quad \in \mathbb{R}^{K\times K} \;\; (\textit{changed}).$$

$$\left[\frac{\partial \ell}{\partial \hat{y}}\right]_j = 2(\hat{y}_j - y_j), \qquad \frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y}-y)^\top \in \mathbb{R}^{1\times K} \quad \text{(unchanged)}.$$

(b) (2pt + 3pt + 1pt) Now you think you can do a better job by using a Binary Cross Entropy (BCE) loss function

$$\ell_{\text{BCE}}(\hat{y}, y) = \frac{1}{K}\sum_{i=1}^{K}\Big(- \big[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\big]\Big).$$

What do you need to change in the equations of (b), (c) and (d)?

(b) Forward pass: same as in (a) (no change).

(c) Backward pass: same chain-rule structure as above; *only* the upstream row-gradient changes to

$$\left[\frac{\partial \ell}{\partial \hat{y}}\right]_j = \frac{1}{K}\left(-\frac{y_j}{\hat{y}_j} + \frac{1-y_j}{1-\hat{y}_j}\right), \qquad \frac{\partial \ell}{\partial \hat{y}} \in \mathbb{R}^{1\times K} \;\; (\textit{changed}).$$

Local Jacobians remain those of (a): $\dfrac{\partial \hat{y}}{\partial z_3} = \operatorname{diag}\big(\hat{y}\odot(1-\hat{y})\big)$ and $\dfrac{\partial z_2}{\partial z_1} = \operatorname{diag}\big(1-\tanh^2(z_1)\big)$. *(Note: combining these gives the familiar simplification* $\frac{\partial \ell}{\partial z_3} = \frac{1}{K}(\hat{y}-y)$*.)*

(d) Elementwise: replace only $\dfrac{\partial \ell}{\partial \hat{y}}$:

$$\left[\frac{\partial \ell}{\partial \hat{y}}\right]_j = \frac{1}{K}\left(-\frac{y_j}{\hat{y}_j} + \frac{1-y_j}{1-\hat{y}_j}\right) \quad (\textit{changed});$$

the elementwise forms of $\dfrac{\partial \hat{y}}{\partial z_3}$ and $\dfrac{\partial z_2}{\partial z_1}$ are the same as in (a).

(c) (1pt) Things are getting better. You realize that not all intermediate hidden activations need to be binary (or soft version of binary). You decide to use $f(\cdot) = (\cdot)^+$ but keep g as tanh. Explain why this choice of f can be beneficial for training a (deeper) network.

Using $(\cdot)^+$ (ReLU) in hidden layers avoids saturation on positive inputs: its derivative is 1 (not $\ll 1$ as with tanh), so gradients do not vanish as quickly with depth. This yields sparser activations, better gradient flow, and often faster, more stable training for deeper networks, while the final tanh keeps outputs bounded.

1.4 Deriving Loss Functions

Derive the loss function for the following algorithms based on their common update rule

$$w_i \leftarrow w_i + \eta\,(y-\hat{y})\,x_i.$$

Show the steps of the derivation given the following inference rules (simply stating the final loss function will receive no points).

We want to find the loss functions whose gradient descent update rule matches

$$w \leftarrow w + \eta(y - \hat{y})x, \qquad b \leftarrow b + \eta(y - \hat{y}).$$

This means the gradient of the loss must satisfy

$$\frac{\partial L}{\partial w} = -(y - \hat{y})x^\top, \qquad \frac{\partial L}{\partial b} = -(y - \hat{y}).$$

1. (4 pt) Perceptron $\quad \hat{y} = \text{sign}\Big(b + \sum_{i=1}^{d} w_i x_i\Big)$

   Here $\hat{y} = \text{sign}(b + w^\top x)$. The perceptron loss is

   $$L = \max\{0, -y(b + w^\top x)\}, \quad y \in \{-1, +1\}.$$

   If $y(b + w^\top x) > 0$, the derivative is zero. If $y(b + w^\top x) \leq 0$, then

   $$\frac{\partial L}{\partial w} = -yx^\top, \qquad \frac{\partial L}{\partial b} = -y.$$

   This leads to the update $w \leftarrow w + \eta yx$, which matches the given rule up to a constant factor (absorbed into η).

2. (4 pt) Adaline / Least Mean Squares $\quad \hat{y} = b + \sum_{i=1}^{d} w_i x_i$

   Here $\hat{y} = b + w^\top x$. We take the squared error loss:

   $$L = \tfrac{1}{2}(y - \hat{y})^2 = \tfrac{1}{2}(y - (b + w^\top x))^2.$$

   Then

   $$\frac{\partial L}{\partial w} = -(y - \hat{y})x^\top, \qquad \frac{\partial L}{\partial b} = -(y - \hat{y}).$$

   So the gradient descent update is exactly

   $$w \leftarrow w + \eta(y - \hat{y})x, \qquad b \leftarrow b + \eta(y - \hat{y}).$$

3. (4 pt) Logistic Regression $\quad \hat{y} = \tanh\Big(b + \sum_{i=1}^{d} w_i x_i\Big)$

   Let $x \in \mathbb{R}^d$ and $w \in \mathbb{R}^d$ be column vectors, $b \in \mathbb{R}$, and define

   $$z := b + w^\top x, \qquad \hat{y} := \tanh(z), \qquad y \in \{-1, +1\}.$$

   We use the negative log-likelihood (logistic loss) corresponding to $P(y \mid z) = \sigma(2yz)$:

   $$L(w, b) \;=\; \log\big(1 + e^{-2yz}\big).$$

Step 1: Differentiate L w.r.t. z (numerator-layout). Let $u := -2yz$. Then

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial u}\,\frac{\partial u}{\partial z} = \frac{e^u}{1+e^u}\cdot(-2y) = -\,\frac{2y}{1+e^{2yz}}.$$

Step 2: Relate $\dfrac{\partial L}{\partial z}$ to $(y-\hat{y})$. Using $\hat{y} = \tanh(z)$ and $y \in \{\pm 1\}$, we have the identity

$$1 - y\,\tanh(z) = 1 - \tanh(yz) = \frac{2}{1+e^{2yz}},$$

hence

$$y - \hat{y} = y - \tanh(z) = y\big(1 - y\,\tanh(z)\big) = \frac{2y}{1+e^{2yz}}.$$

Therefore

$$\frac{\partial L}{\partial z} = -\,(y-\hat{y}).$$

Step 3: Apply the chain rule to get gradients w.r.t. (w, b) (numerator-layout). Since $\dfrac{\partial z}{\partial w} = x^\top$ and $\dfrac{\partial z}{\partial b} = 1$, we obtain

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z}\,\frac{\partial z}{\partial w} = -\,(y-\hat{y})\,x^\top, \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\,\frac{\partial z}{\partial b} = -\,(y-\hat{y}).$$

Step 4: Gradient descent update matches the required rule. Using $w \leftarrow w - \eta\big(\frac{\partial L}{\partial w}\big)^\top$ and $b \leftarrow b - \eta\frac{\partial L}{\partial b}$,

$$w \leftarrow w + \eta\,(y-\hat{y})\,x, \qquad b \leftarrow b + \eta\,(y-\hat{y}),$$

which is exactly the given update.

1.5 Conceptual Questions

(a) (1pt) Why is softmax actually softargmax?

Given scores $s \in \mathbb{R}^K$, the temperature-scaled softmax is

$$\mathrm{softmax}_\tau(s)_i = \frac{\exp(s_i/\tau)}{\sum_{j=1}^K \exp(s_j/\tau)}, \qquad \tau > 0.$$

- As $\tau \to 0^+$, the distribution concentrates on the index of the largest score:

  $$\lim_{\tau\to 0^+} \mathrm{softmax}_\tau(s) = \text{one-hot vector at } \arg\max_i s_i,$$

  so it behaves like the discrete arg max operator.
- As $\tau \to \infty$, all exponentials become nearly equal, giving

  $$\lim_{\tau\to\infty} \mathrm{softmax}_\tau(s) = \Big(\tfrac{1}{K}, \tfrac{1}{K}, \ldots, \tfrac{1}{K}\Big),$$

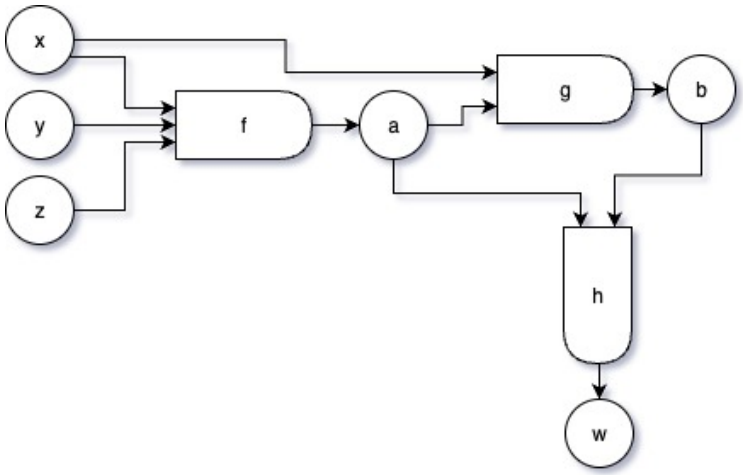
  the uniform distribution.

(b) (3pt) Draw the computational graph defined by this function, with inputs $x, y, z \in \mathbb{R}$ and output $w \in \mathbb{R}$. You make use symbols x, y, z, o, and operators $*, +$ in your solution. Be sure to use the correct shape for symbols and operators as shown in class.

$$a = x * y + z, \qquad b = (x + x) * a, \qquad w = a * b.$$

We can state

$$\begin{aligned} a &= f(x, y, z) = x * y + z \\ b &= g(x, a) = (x + x) * a \\ w &= h(a, b) = a * b \end{aligned}$$



(c) (2pt) Draw the graph of the derivative for the following functions?

- ReLU()
- LeakyReLU(negative_slope = 0.01)
- Softplus($\beta = 1$)
- GELU()

$$\text{ReLU:} \quad f(x) = \max(0, x), \qquad f'(x) = \begin{cases} 0, & x < 0, \\ \text{undefined (choose 0 or 1)}, & x = 0, \\ 1, & x > 0 \end{cases}$$

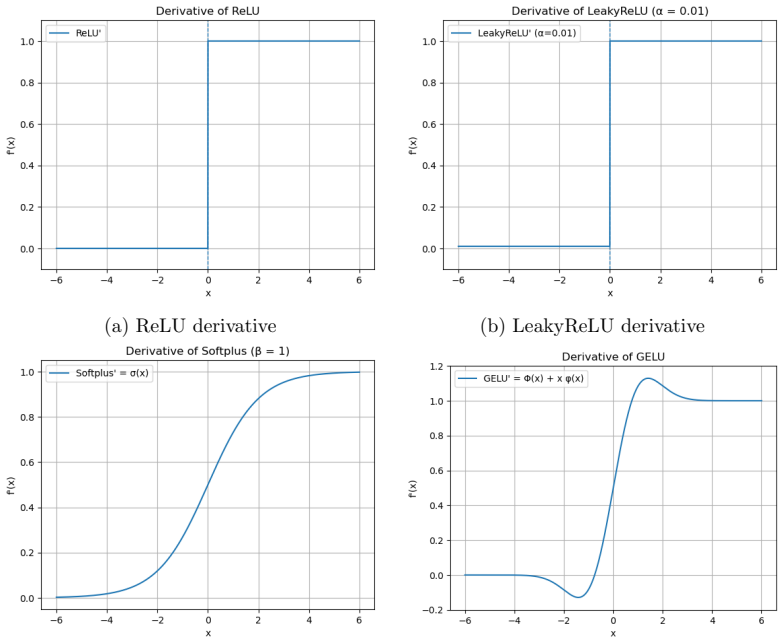
$$\text{LeakyReLU } (\alpha = 0.01): \quad f(x) = \max(\alpha x, x), \quad f'(x) = \begin{cases} \alpha, & x < 0, \\ \text{undefined (choose } \alpha \text{ or 1)}, & x = 0, \\ 1, & x > 0 \end{cases}$$

$$\text{Softplus } (\beta = 1): \quad f(x) = \log(1 + e^x), \quad f'(x) = \frac{1}{1 + e^{-x}} = \sigma(x)$$

$$\text{GELU:} \quad f(x) = x\,\Phi(x), \qquad f'(x) = \Phi(x) + x\,\phi(x)$$

Where:

- $\alpha = 0.01$ is the negative slope for LeakyReLU.
- $\sigma(x) = \dfrac{1}{1+e^{-x}}$ is the logistic sigmoid.
- $\Phi(x)$ is the standard normal cumulative distribution function (CDF).
- $\phi(x) = \dfrac{1}{\sqrt{2\pi}} e^{-x^2/2}$ is the standard normal probability density function (PDF).



(a) ReLU derivative

(b) LeakyReLU derivative

(c) Softplus derivative

(d) GELU derivative

(d) (3pt) What are 4 different types of linear transformations? What is the role of linear transformation and nonlinear transformation in a neural network?

Examples of linear transformations in $\mathbb{R}^n$:

- **Scaling**: multiply each coordinate by a constant factor (diagonal matrix).
- **Rotation**: preserve length and angle while rotating vectors (orthogonal matrix with $\det = 1$).
- **Reflection**: flip across a hyperplane (orthogonal matrix with $\det = -1$).
- **Shear / Projection**: shift or collapse dimensions while preserving linearity.

Role in neural networks:

- *Linear transformation*: mixes input features and changes their representation (e.g. $Wx + b$).

- *Nonlinear transformation*: introduces nonlinearity so that stacking layers does not collapse to a single linear map. This allows the network to approximate complex, non-linear functions (universal approximation property).

(e) (3pt) Given a neural network F parameterized by parameters θ, denoted F_θ, dataset $D = \{x_1, x_2, \ldots, x_N\}$, and labels $Y = \{y_1, y_2, \ldots, y_N\}$, write down the mathematical definition of training a neural network with the MSE loss function $\ell_{\text{MSE}}(\hat{y}, y) = \|\hat{y} - y\|^2$.

Training corresponds to minimizing the empirical risk defined by the MSE loss:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell_{\text{MSE}}\big(F_\theta(x_i),\, y_i\big), \qquad \ell_{\text{MSE}}(\hat{y}, y) = \|\hat{y} - y\|^2.$$

Thus the optimization problem is

$$\theta^* = \arg\min_\theta L(\theta).$$

In practice, this is done by gradient descent:

$$\theta \;\leftarrow\; \theta - \eta \left(\frac{\partial}{\partial \theta} \frac{1}{B} \sum_{i \in \mathcal{B}} \|F_\theta(x_i) - y_i\|^2 \right)^\top,$$

where $\mathcal{B}$ is a mini-batch of size B.

2 Implementation (50pt)

2.1 Backpropagation (35pt)

You need to implement the forward pass and backward pass for Linear, ReLU, Sigmoid, MSE loss, and BCE loss in the attached `mlp.py` file. We provide three example test cases `test1.py`, `test2.py`, `test3.py`. We will test your implementation with other hidden test cases, so please create your own test cases to make sure your implementation is correct.

Recommendation: Go through this PyTorch tutorial to have a thorough understanding of Tensors. `https://pytorch.org/tutorials/beginner/basics/intro.html`

Extra instructions:

1. Please use Python version $\geq$ 3.7 and PyTorch version 1.7.1. We recommend you to use Miniconda to manage your virtual environment.
2. We will put your `mlp.py` file under the same directory of the hidden test scripts and use the command `python hiddenTestScriptName.py` to check your implementation. So please make sure the file name is `mlp.py` and it can be executed with the example test scripts we provided.
3. You are not allowed to use PyTorch autograd functionality in your implementation.
4. Be careful about the dimensionality of the vector and matrix in PyTorch. It is not necessarily follow the the Math you got from part 1.

2.2 Gradient Descent (15pt + 5pt)

In DeepDream, the paper claims that you can follow the gradient to maximize an energy with respect to the input in order to visualize the input.[2] We provide some code to do this. Given an image classifier, implement a function that performs optimization on the input (the image), to find the image that most highly represents the class. You will need to implement the `gradient_descent` function in `gd.py`. You will be graded on how well the model optimizes the input with respect to the labels.

Extra hints:

1. We try to minimize the energy of the class, e.g. maximize the class logit. Make sure you are following the gradient in the right direction.
2. A reasonable starting learning rate to try is 0.01, but depending on your implementation, make sure to sweep across a few magnitudes.
3. Make sure you use `normalize_and_jitter`, since the neural network expect a normalized input. Jittering produces more visually pleasing results.

You may notice that the images that you generate are very messy and full of high frequency noise. Extra credit (5 points) can be had by generating visually pleasing images, and experimenting with visualizing the middle layers of the network. There are some tricks to this:

1. Blur the image at each iteration, which reduces high frequency noise
2. Clamp the pixel values between 0 and 1
3. Implement weight decay
4. Blur the gradients at each iteration
5. Implement gradient descent at multiple scales, scaling up every so often

[2] See Simonyan et al., 2013, and Mordvintsev, Olah, and Tyka, 2015. `https://arxiv.org/abs/1312.6034` `https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html`