COSC490LLModels

Homework 3: Building Your Neural Network!

Name: __________Doug Hoover________________
Collaborators, if any: ____
Sources used for your homework, if any: _______________________________________

This assignment is focusing on understanding the fundamental properties of neural networks and their training.

Homework goals: After completing this homework, you should be comfortable with:

- thinking about neural networks
- key implementation details of NNs, particularly in PyTorch,
- explaining and deriving Backpropagation,
- debugging your neural network in case it faces any failures.

How to hand in your written work: via MyClasses.

Collaboration: Make certain that you understand the course collaboration policy, described on the course website. You may discuss the homework to understand the problems and the mathematics behind the various learning algorithms, but you are **not allowed to share problem solutions with any other students. You must write the solutions individually**.

1 Concepts, intuitions and big picture

1. Suppose you have built a neural network. You decide to initialize the weights and biases to be zero. Which of the following statements are True? (Check all that apply)
☐✓ Each neuron in the first hidden layer will perform the same computation. So even after multiple iterations of gradient descent each neuron will be computing the same thing as other neurons in the same layer.
☐ Each neuron in the first hidden layer will perform the same computation in the first iteration. But after one iteration of gradient descent they will learn to compute different things because we have "broken symmetry".
☐ Each neuron in the first hidden layer will compute the same thing, but neurons in different layers will compute different things, thus we have accomplished "symmetry breaking" as described in lecture.
☐ The first hidden layer's neurons will perform different computations from each other even in the first iteration; their parameters will thus keep evolving in their own way.
Answer:

2. Vectorization allows you to compute forward propagation in an L-layer neural network without an explicit for-loop (or any other explicit iterative loop) over the layers $l = 1, 2, \times, L$. True/False?
☐ True
☐✓ False
Answer:

3. The `tanh` activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer. True/False?
☐✓ True
☐ False
Answer:

4. Which of the following techniques does NOT prevent a model from overfitting?
☐ Data augmentation ☐ Dropout ☐ Early stopping ☐✓ None of the above
Answer:

5. Why should dropout be applied during training? Why should dropout NOT be applied during evaluation?
Answer: *Dropout should be applied during training to prevent over fitting by random deactivated neurons, This makes it learn more robust features. While dropout is not applied during evaluation because it would randomness and reduce the performance.*

6. Explain why initializing the parameters of a neural net with a constant is a bad idea.
Answer: *Initializing with constant leads to symmetry between the neurons in the same layer. They will have identical updates and never learn different features.*

7. You design a fully connected neural network architecture where all activations are sigmoids. You initialize the weights with large positive numbers. Is this a good idea? Explain your answer.
Answer: *No this is a bad idea because large positive weights push activation into the saturated region. This can lead to very small gradients slowing it down or preventing the model from learning.*

8. Explain what is the importance of "residual connections".
Answer: *residual connections allow gradients to flow easily by bypassing layers and allowing for deeper networks to train effectively.*

9. What is cached ("memoized") in the implementation of forward propagation and backward propagation?
☐✓ Variables computed during forward propagation are cached and passed on to the corresponding backward propagation step to compute derivatives.
☐ Caching is used to keep track of the hyperparameters that we are searching over, to speed up computation.
☐ Caching is used to pass variables computed during backward propagation to the corresponding forward propagation step. It contains useful values for forward propagation to compute activations.
Answer:

10. Which of the following statements is true?
☐✓ The deeper layers of a neural network are typically computing more complex features of the input than the earlier layers.
☐ The earlier layers of a neural network are typically computing more complex features of the input than the deeper layers.
Answer:

2 Revisiting Jacobians

Recall that Jacobians are generalizations of multi-variate derivatives and are extremely useful in denoting the gradient computations in computation graph and Backpropagation. A potentially confusing aspect of using Jacobains is their dimensions and so, here we're going focus on understanding Jacobian dimensions.

Recap: Let's first recap the formal definition of Jacobian. Suppose $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is a function takes a point $\mathbf{x} \in \mathbb{R}^n$ as input and produces the vector $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$ as output. Then the Jacobian matrix of $\mathbf{f}$ is defined to be an $m \times n$ matrix, denoted by $\mathbf{J_f}(\mathbf{x})$, whose (i, j)th entry is $\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$, or:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^\top f_1 \\ \vdots \\ \nabla^\top f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Examples: The shape of a Jacobian is an important notion to note. A Jacobian can be a vector, a matrix, or a tensor of arbitrary ranks. Consider the following special cases:

- If f is a scalar and $\mathbf{w}$ is a $d \times 1$ column vector, the Jacobian of f with respect to $\mathbf{w}$ is a row vector with $1 \times d$ dimensions.
- If $\mathbf{y}$ is a $n \times 1$ column vector and $\mathbf{z}$ is a $d \times 1$ column vector, the Jacobian of $\mathbf{z}$ with respect to y, or $\mathbf{J_z}(\mathbf{y})$ is a $d \times n$ matrix.
- Suppose $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{l \times p \times q}$. Then the Jacobian $\mathbf{J_A}(\mathbf{B})$ is a tensor of shape $(m \times n) \times (l \times p \times q)$. More broadly, the shape of the Jacobian is determined as (shape of the output)$\times$(shape of the input).

Problem setup: Suppose we have:

- X, an $n \times d$ matrix, $x_i \in \mathbb{R}^{d \times 1}$ correspond to the rows of $X = [x_1, \ldots, x_n]^\top$
- Y, a $n \times k$ matrix
- W, a $k \times d$ matrix and w, a $d \times 1$ vector

For the following items, compute (1) the shape of each Jacobian, and (2) an expression for each Jacobian:

1. $f(w) = c$ (constant)
   Answer: *The shape is 1 X d. The expression is*

$$J_f(w) = [0\ 0\ \ldots\ 0] \in \mathbf{R}^{1 \times d}$$

2. $f(w) = \|w\|^2$ (squared L2-norm)
   Answer: *The shape is 1 X d The expression is*

$$J_f(w) = 2w^t$$

3. $f(w) = w^\top x_i$ (vector dot product)
   Answer: *The shape is 1 x d. The expression is*

$$J_f(w) = x_i^T$$

4. $f(w) = Xw$ (matrix-vector product)
   Answer: *The shape is n x d. The expression is*

$$J_f(w) = X$$

5. $f(w) = w$ (vector identity function)
Answer: *The shape is d x d. The expression is*
$$J_f(w) = I_d$$

6. $f(w) = w^2$ (element-wise power)
Answer: *The shape is d x d. The expression is*
$$J_f(w) = 2diag(w)$$

7. **Extra Credit:** $f(W) = XW^\top$ (matrix multiplication)
Answer: *The shape is*
$$(n * k) * (k * d)$$
The expression is
$$\frac{\partial(XW^\top)}{\partial W} = X$$

3 Activations Per Layer, Keeps Linearity Away!

Based on the content we saw at the class lectures, answer the following:

1. Why are activation functions used in neural networks? Answer: *Activation functions allow for non linearity into a neural network. This means its able to learn complex patterns and represent non-trivial relationships. Without them a neural network would essentially behave as a linear model/*

2. Write down the formula for three common action functions (sigmoid, ReLU, Tanh) and their derivatives (assume scalar input/output). Plot these activation functions and their derivatives on $(-\infty, +\infty)$. Answer: *sigmod:*

$$\partial(x) = \frac{1}{1+e^{-x}}$$

Derivative:

$$\partial'(x) = \partial(x)(1-\partial(x))$$

ReLU:

$$f(x) = max(0, x)$$

Derivative:

$$f'(x) = 1, x > 0, 0, x <= 0$$

Tanh:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Derivative:

$$tanh'(x) = 1 - tanh^2(x)$$

3. What is the "vanishing gradient" problem? (respond in no more than 3 sentences) Which activation functions are subject to this issue and why? (respond in no more than 3 sentences).
Answer: *The vanishing gradient problem happens when gradients become extremely small during backpropagation, causing weights in earlier layers to update slowly. This is most common in deep networks like sigmoid or tanh, where their derivatives saturate to near zero. This makes the networks become stalled.*

4. Why zero-centered activation functions impact the results of Backprop?
Answer: *Zero-centered activation function help prevent biases in weight updates by ensuring that gradient updates are more balanced. This reduces oscillation and leads to faster convergence.*

5. Remember the Softmax function $\sigma(\mathbf{z})$ and how it extends sigmoid to multiple dimensions? Let's compute the derivative of Softmax for each dimension. Prove that:

$$\frac{d\sigma(\mathbf{z})_i}{dz_j} = \sigma(\mathbf{z})_i(\delta_{ij} - \sigma(\mathbf{z})_j)$$

where δ_{ij} is the Kronecker delta function.[*] Answer: *by taking the derivative with respect to*

$$z_j$$

we get

$$\frac{d\partial(z)_i}{dz_j} = \partial(z)_i(s_{ij} - \partial(z)_j)$$

where

$$s_{ij}$$

is the Kronecker delta function.

6. Use the above point to prove that the Jacobian of the Softmax function is the following:

$$\mathbf{J}_\sigma(\mathbf{z}) = \text{diag}(\sigma(\mathbf{z})) - \sigma(\mathbf{z})\sigma(\mathbf{z})^\top$$

where diag(.) turns a vector into a diagonal matrix. Also, note that $\mathbf{J}_\sigma(\mathbf{z}) \in \mathbb{R}^{K\times K}$..
Answer: *This shows that the jacobian consists of a diagonal matrix of softmax values minus the outer product of the softmax vector with itself. The results is K X K matrix, which shows how each output component of the softmax function affects every other component.*

[*] https://en.wikipedia.org/wiki/Kronecker_delta

4 Simulating XOR

1. Can a single-layer network simulate (represent) an XOR function on $\mathbf{x} = [x_1, x_2]$?

$$y = \mathrm{XOR}(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} = (0,1) \text{ or } \mathbf{x} = (1,0) \\ 0, & \text{if } \mathbf{x} = (1,1) \text{ or } \mathbf{x} = (0,0). \end{cases}$$

   Explain your reasoning using the following single-layer network definition: $\hat{y} = \mathrm{ReLU}(W \cdot \mathbf{x} + b)$ Answer: *No a single layer network cannot show the XOR function. The XOR function is not linearly separable meaning no single hyperplane can separate the two classes.*

2. Repeat (1) with a two-layer network:

$$\begin{aligned} \mathbf{h} &= \mathrm{ReLU}(W_1 \cdot \mathbf{x} + \mathbf{b}_1) \\ \hat{y} &= W_2 \cdot \mathbf{h} + b_2 \end{aligned}$$

   Note that this model has an additional layer compared to the earlier question: an input layer $\mathbf{x} \in \mathbb{R}^2$, a hidden layer $\mathbf{h}$ with ReLU activation functions that are applied component-wise, and a linear output layer, resulting in scalar prediction $\hat{y}$. Provide a set of weights W_1 and W_2 and biases $\mathbf{b}_1$ and b_2 such that this model can accurately model the XOR problem. Answer: *A two layer network can represent XOR by creating intermediate features that separate the XOR outputs into linearly separable regions. We define this with the functions above. We can choose the appropriate weights and biases. The firs layer is the hidden which can be written as:*

$$W_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{b}_1 \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

   The second layer is the output neuron:

$$W_2 = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad b_2 = -0.5$$

3. Consider the same network as above (with ReLU activations for the hidden layer), with an arbitrary differentiable loss function $\ell : \{0,1\}\ \{0,1\} \to \mathbb{R}$ which takes as input $\hat{y}$ and y, our prediction and ground truth labels, respectively. Suppose all weights and biases are initialized to zero. Show that a model trained using standard gradient descent will not learn the XOR function given this initialization.
   Answer: *When all weights and biases are initialized to zero, every neuron in a given layer will compute the same output during forward propagation. This means that during backprop., all neurons will receive identical gradients. As a result the network will remain symmetric throughout training and preventing the differentiation to learn the XOR function. The issues happen because gradient descent updates are the same for all neurons, so the network fails to break symmetry.*

5 Neural Nets and Backpropagation

Draw the computation graph for $f(x,y,z) = \ln x + \exp(y) \cdot z$. Each node in the graph should correspond to only one simple operation (addition, multiplication, exponentiation, etc.). Then we will follow the forward and backward propagation described in class to estimate the value of f and partial derivatives $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ at $[x,y,z] = [1,3,2]$. For each step, show you work.

A computation graph, so elegantly designed
With nodes and edges, so easily combined
It starts with inputs, a simple array
And ends with outputs, in a computationally fair way

Each node performs, an operation with care
And passes its results, to those waiting to share
S The edges connect, each node with its peers
And flow of information, they smoothly steer

It's used to calculate, complex models so grand
And trains neural networks, with ease at hand
Backpropagation, it enables with grace
Making deep learning, a beautiful race

–ChatGPT Feb 3 2023

1. Draw the computation graph for $f(x,y,z) = \ln x + \exp(y) \cdot z$. The graph should have three input nodes for x, y, z and one output node f. Label each intermediate node h_i.
   Answer: *To make the graph we can break the function into operations such as*

$$h_1 = ln(x), h_2 = exp(y), h_3 = h_2 * z, f = h_1 + h_3$$

   h1 is natural logarithm of x, h2 is the exponential of y, h3 is the product of h2 and z and f is the sum of h1 and h3

2. Run the forward propagation and evaluate f and h_i $(i = 1, 2, \dots)$ at $[x,y,z] = [1,3,2]$.
   Answer: *by running the forward propagation with the values 1,3,2 we get*

$$h_1 = ln(x) = ln(1) = 0, h_2 = exp(y) = exp(3) = 20.0855, h_3 = h_2 * z = 20.0855 * 2 = 40.171, f = h_1 + h_3 = 0 + 40.171 = 40.$$

3. Run the backward propagation and give partial derivatives for each intermediate operation, i.e., $\frac{\partial h_i}{\partial x}, \frac{\partial h_j}{\partial h_i}$, and $\frac{\partial f}{\partial h_i}$. Evaluate the partial derivatives at $[x,y,z] = [1,3,2]$.
   Answer: *Using backpropagation we find:*

$$\frac{\partial f}{\partial h_1} = 1, \frac{\partial f}{\partial h_3} = 1, \frac{\partial h_3}{\partial h_2} = z = 2, \frac{\partial h_2}{\partial z} = h_2 = 20.0855, \frac{\partial h_2}{\partial y} = exp(y) = 20.8500, \frac{\partial h_1]}{\partial x} = \frac{1}{x} = \frac{1}{1} = 1$$

4. Aggregate the results in (c) and evaluate the partial derivatives $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$ with chain rule. Show your work.
   Answer: *First we find the partial derivative of f with respect to x*

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h_1} * \frac{\partial h_1}{\partial x} = 1 * 1 = 1$$

   Next is the find the partial derivative of f with respect to y:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial h_3} * \frac{\partial h_3}{\partial h_2} * \frac{\partial h_2}{\partial y} = 1 * 2 * 20.0855 = 40.171$$

   lastly the partial derivative of f with respect to z:

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial h_3} * \frac{\partial h_3}{\partial z} = 1 * 20.0855 = 20.855$$

6 Programming

In this programming homework, we will

- implement MLP-based classifiers for the sentiment classification task of homework 1.

Skeleton Code and Structure: The code base for this homework can be found at MyClasses Files under the `hw3` directory. Your task is to fill in the missing parts in the skeleton code, following the requirements, guidance, and tips provided in this pdf and the comments in the corresponding .py files. The code base has the following structure:

- `mlp.py` reuse the sentiment classifier on movie reviews you implemented in homework 1, with additional requirements to implement MLP-based classifier architectures and forward pass .
- `main.py` provides the entry point to run your implementations `mlp.py`
- `hw3.md` provides instructions on how to setup the environment and run each part of the homework in `main.py`

TODOs — Your tasks include 1) generate plots and/or write short answers based on the results of running the code; 2) fill in the blanks in the skeleton to complete the code. We will explicitly mark these plotting, written answer, and filling-in-the-blank tasks as **TODOs** in the following descriptions, as well as a `# TODO` at the corresponding blank in the code.

TODOs (Copy from your HW1). We are reusing most of the `model.py` from homework 1 as the starting point for the `mlp.py` - you will see in the skeleton that they look very similar. Moreover, in order to make the skeleton complete, for all the `# TODO (Copy from your HW1)`, please fill in the blank below them by copying and pasting the corresponding implementations you wrote for homework 1 (i.e. the corresponding `# TODO` in homework 1.)

Submission: Your submission should contain two parts: 1) plots and short answers under the corresponding questions below; and 2) your completion of the skeleton code base, in a `.zip` file

6.1 MLP-based Sentiment Classifier

In both homework 1 & 2, our implementation of the `SentimentClassifer` is essentially a single-layer feedforward neural network that maps input features directly to 2-dimensional output logits. In this part of the programming homework, we will expand the architecture of our classifier to multi-layer perceptron (MLP).

6.1.1 Reuse Your HW1 Implementation

TODOs (Copy from your HW1): for all the `# TODO (Copy from your HW1)` in `mlp.py`, please fill in the blank below them by copying and pasting the corresponding implementations you wrote for homework 1 (i.e. the corresponding `# TODO` in the `model.py` in homework 1).

6.1.2 Build MLPs

Remember from the lecture that MLP is a multi-layer feedforward network with perceptrons as its nodes. A perceptron consists of non-linear activation of the affine (linear) transformation of inputs.

TODOs: Complete the `__init__` and `forward` function of the `SentimentClassifier` class in `mlp.py` to build MLP classifiers that supports custom specification of architecture (i.e. number and dimension of hidden layers)
Hint: check the comments in the code for specific requirements about input, output, and implementation. Also, check out the document of nn.ModuleList about how to define and implement forward pass of MLPs as a stack of layers.

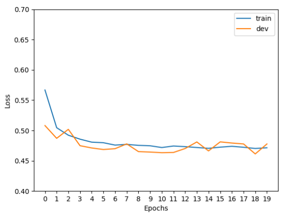
6.1.3 Train and Evaluate MLPs

We provide in `main.py` several MLP configurations and corresponding recipes for training them.

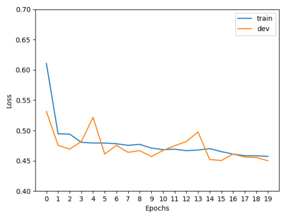
TODOs Once you finished subsubsection 6.1.2, you can run `load_data_mlp` and `explore_mlp_structures` to train and evaluate these MLPs and paste two sets of plots here:

- 4 plots of train & dev loss for each MLP configuration
- 2 plots of dev losses and accuracies across MLP configurations

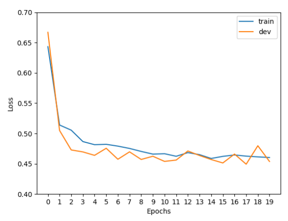
and describe in 2-3 sentences your findings.
Hint: what are the trends of train & dev loss and are they consistent across different configurations? Is deeper models always better? Why?
your plots and answer:



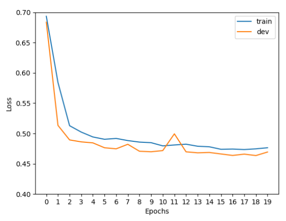
(a) No Hidden Layer



(b) 1 Hidden Layer

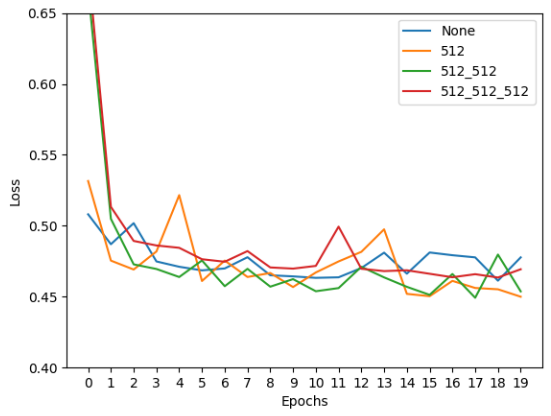


(c) 2 Hidden Layers

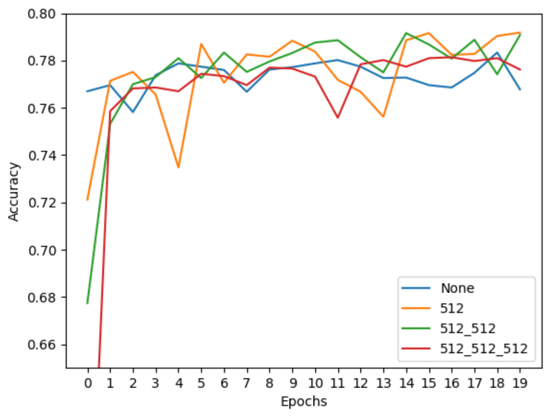


(d) 3 Hidden Layers

Figure 1: loss of different MLP architectures



(a) dev loss of different MLP architectures



(b) dev acc of different MLP architectures

Figure 2: loss and acc of different MLP architectures

Answer: *The train and dev loss generally decreases over time which shows the model is learning but the rate is decreasing. The final loss value can vary across different configurations. The trends are mostly consistent but some configurations show over fitting where the train loss continues to decrease while dev loss starts to increase. By testing deeper models they do not always do better and are more prone to over fitting if the dataset is large enough.*

6.1.4 Embrace Non-linearity: The Activation Functions

Remember we have learned why adding non-linearity is useful in neural nets and gotten familiar with several non-linear activation functions both in the class and section 3. Now it is time to try them out in our MLPs!
Note: for the following TODO and the TODO in subsubsection 6.1.5, we fix the MLP structure to be with a single 512-dimension hidden layer, as specified in the code. You only need to run experiments on this architecture.

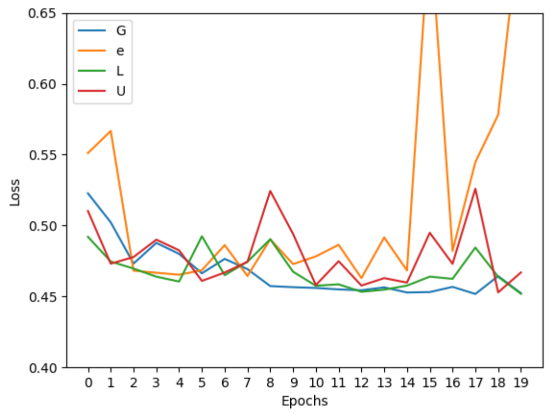
TODOs: Read and complete the missing lines of the two following functions:

- `__init__` function of the `SentimentClassifier` class: define different activation functions given the input `activation` type.
  Hint: we have provided you with a demonstration of defining the Sigmoid activation, you can search for the other `nn.<activation>` in PyTorch documentation.

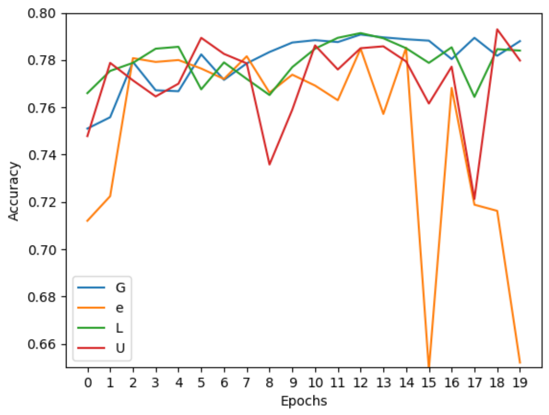
- `explore_mlp_activations` in `main.py`: iterate over the activation options, define the corresponding training configurations, train and evaluate the model, and visualize the results. Note: you only need to generate the

plots of dev loss and dev acc across different configurations, by calling `visualize_configs`, you **do not** need to plot the train-dev loss curves for each configuration (i.e. no need to call `visualize_epochs`). We provide you with a few choices of common activation functions, but feel free to try out the others.
Hint: You can refer to `explore_mlp_structure` as a demonstration of how to define training configurations with fixed hyper-parameters & iterate over hyper-parameters/design choices of interests (e.g. hidden dimensions, choice of activation), and plot the evaluation results across configurations.

Once you complete the above functions, run `explore_mlp_activations` and paste the two generated plots here. Describe in 2-3 sentences your findings.
your plots and answer:



(a) dev loss of different activations



(b) dev acc of different MLP architectures

Figure 3: loss and acc of different activations

Answer: *Different activation functions can lead to different performances. In the ReLU and GeLU perform better in terms of convergence speed and final accuracy. ReLU and its variants are probably preferred for deep networks due to its ability to migrate the vanishing gradient problem.*

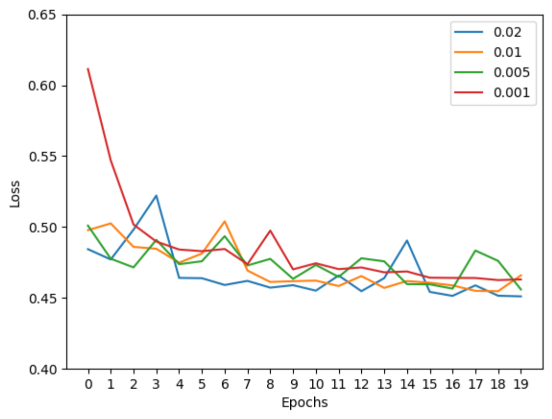
6.1.5 Hyper-parameter Tuning: Learning Rate

The training process mostly involves learning model parameters, which are automatically performed by gradient-based methods. However, certain parameters are "unlearnable" through gradient optimization while playing a crucial role in affecting model performance, for example, learning rate and batch size. We typically refer to these parameters as *Hyper-parameters*.

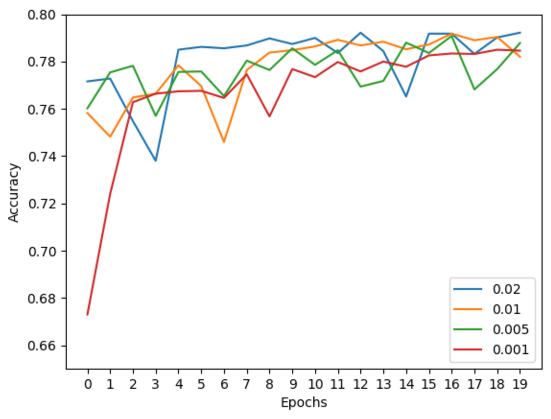
We will now take the first step to tune these hyper-parameters by exploring the choices of one of the most important one - learning rate, on our MLP. (There are lots of tutorials on how to tune the learning rate manually or automatically in practice, for example this note can serve as a starting point.)

TODOs: Read and complete the missing lines in `explore_mlp_learning_rates` in `main.py` to iterate over different learning rate values, define the training configurations, train and evaluate the model, and visualize the results. Note: same as above, you only need to generate the plots of dev loss and dev acc across different configurations, by calling `visualize_configs`, you **do not** need to plot the train-dev loss curves for each configuration (i.e. no need to call `visualize_epochs`). We provide you with the default learning rate we set to start with, and we encourage you to add more learning rate values to explore and include in your final plots curves of **at least 4 different representative learning rates.**
Hint: again, you can checkout `explore_mlp_structure` as a demonstration for how to perform hyper-parameter search.
Once you complete the above functions, run `explore_mlp_learning_rates` and paste the two generated plots here. Describe in 2-3 sentences your findings.
your plots and answer:

Answer: *The learning rate has a substantial impact on the training process. A learning rate that is too high can cause the model to diverge, while a learning rate that is too low can cause slow convergence.*



(a) dev loss of different learning rates



(b) dev acc of different MLP architectures

Figure 4: loss and acc of different learning rates

References