705.641.81: Natural Language Processing Self-Supervised Models

Homework 3: Building Your Neural Network!

For homework deadline and collaboration policy, please see our Canvas page.

Name: _____Jacob Cunningham_____

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.

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.

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**

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

4. Which of the following techniques does NOT prevent a model from overfitting?
   ▫ Data augmentation

   ▫ Dropout

   ▫ Early stopping

   ▫ **None of the above**

5. Why should dropout be applied during training? Why should dropout NOT be applied during evaluation?
   Dropout should be applied during training because it randomly deactivates neurons, and this acts as a regularization technique by preventing co-adaptation of

features and forcing the network to learn more robust, distributed representations. Dropout should not be applied during evaluation because we want to use the full network capacity to make stable and deterministic predictions. Applying dropout at test time would introduce unnecessary randomness and reduce performance, so instead the network should use the appropriately scaled weights learned during training.

6. Explain why initializing the parameters of a neural net with a constant is a bad idea.

   Initializing all parameters to the same constant is a bad idea because it creates a symmetry problem where neurons within the same layer start with identical weights and biases, receive identical gradients during backpropagation, and thus remain identical throughout training. In this scenario, neurons in a layer learn the same features and the network behaves like it has only one neuron per layer. This severely limits the representational power of the model and prevents it from learning complex patterns.

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.
   This is not a good idea as large positive weight initialization will push the inputs to the sigmoid activation into the saturation region near 1. This means that the gradient will be close to zero, causing vanishing gradients and very slow learning. As a result, backpropagation becomes ineffective, especially in deeper networks, and the model may fail to learn meaningful representations.

8. Explain what is the importance of "residual connections".

   Residual connections allow layers to learn residual functions (how the output should change relative to the input) rather than complete transformations, which makes deep neural networks much easier to train. They provide shortcut paths for information and gradients to flow directly across layers, thus mitigating vanishing and exploding gradient problems, enabling stable training of very deep networks, and helping to preserve useful low-level features while higher layers refine them.

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.

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.

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 $\mathbf{J}_\mathbf{f}(\mathbf{x})$, whose (i, j)th entry is $\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$, or:

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

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 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 y is a $n \times 1$ column vector and z is a $d \times 1$ column vector, the Jacobian of $\mathbf{z}$ with respect to y, or $\text{J}_\text{z}(\text{y})$ is a $d \times n$ matrix.
- Suppose A $\in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{l \times p \times q}$. Then the Jacobian $\text{J}_\text{A}(\text{B})$ 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)×(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 $X = [x_1, \dots, x_n]^\top$
- 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)

   The shape of the Jacobian is $\frac{\partial f}{\partial w} \in \mathbb{R}^{1\times d}$ as $f(w)$ is a scalar-valued function w.r.t a $d \times 1$ vector.

   Since $f(w)$ does not depend on w, $\frac{\partial f}{\partial w} = [0, \dots, 0] = 0_{1\times d}$

2. $f(w) = \| w \|^2$ (squared L2-norm)

   $f(w)$ is scalar as $f(w) = \left||w|\right|^2 = w^T w$, where $w \in \mathbb{R}^{d\times 1}$.

   So, we know $\frac{\partial f}{\partial w} \in \mathbb{R}^{1\times d}$.

   And $\frac{\partial f}{\partial w} = \left(\frac{\partial}{\partial w} w^T w\right) = 2w^T$

3. $f(w) = w^\top x_i$ (vector dot product)

   $f(w)$ is scalar valued, so the Jacobian w.r.t. w is again $\frac{\partial f}{\partial w} \in \mathbb{R}^{1\times d}$.

   And $\frac{\partial f}{\partial w} = x_i^T$

4. $f(w) = Xw$ (matrix-vector product)

   $f(w) \in \mathbb{R}^{n\times 1}, w \in \mathbb{R}^{d\times 1}$, so the Jacobian w.r.t w has shape $\frac{\partial f}{\partial w} \in \mathbb{R}^{n\times d}$.

   And because f is linear in w, $\frac{\partial}{\partial w}(Xw) = X$

5. $f(w) = w$ (vector identity function)

$f(w), w \in \mathbb{R}^{d\times 1}$, so $\frac{\partial f}{\partial w} \in \mathbb{R}^{d\times d}$

And we can say that each component $f_i(w) = w_i$, so:

$\frac{\partial f_i}{\partial w_j} = \begin{cases} 1 \; if \; i = j \\ 0 \; if \; i \neq j \end{cases}$, thus $\frac{\partial f}{\partial w} = I_d$, or the $d \times d$ Identity matrix

6. $f(w) = w^2$ (element-wise power)

   $f(w), w \in \mathbb{R}^{d\times 1}$ again, so $\frac{\partial f}{\partial w} \in \mathbb{R}^{d\times d}$
   $\frac{\partial f_i}{\partial w_j} = \begin{cases} 2w_i \; if \; i = j \\ 0 \; if \; i \neq j \end{cases}$, thus $\frac{\partial f}{\partial w} = diag(2w) = 2diag(w)$

7. **Extra Credit:** $f(W) = XW^\top$ (matrix multiplication)

   $f(W) \in \mathbb{R}^{n\times k}$, or $f: \mathbb{R}^{k\times d} \to \mathbb{R}^{n\times k}$

   So, the Jacobian of $f(W)$ w.r.t W has shape $\frac{\partial f}{\partial W} \in \mathbb{R}^{(nk)\times(kd)}$

   Now, using the vectorization identity we have:

   $vec(XW^T) = (I_k \otimes X)vec(W^T)$

   Since $vec(W^T) = Kvec(W)$, where K is the commutation matrix

   So, the Jacobian can be written as $\frac{\partial vec(XW^T)}{\partial vec(W)} = (I_k \otimes X)K$

Activations Per Layer, Keeps Linearity Away!

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

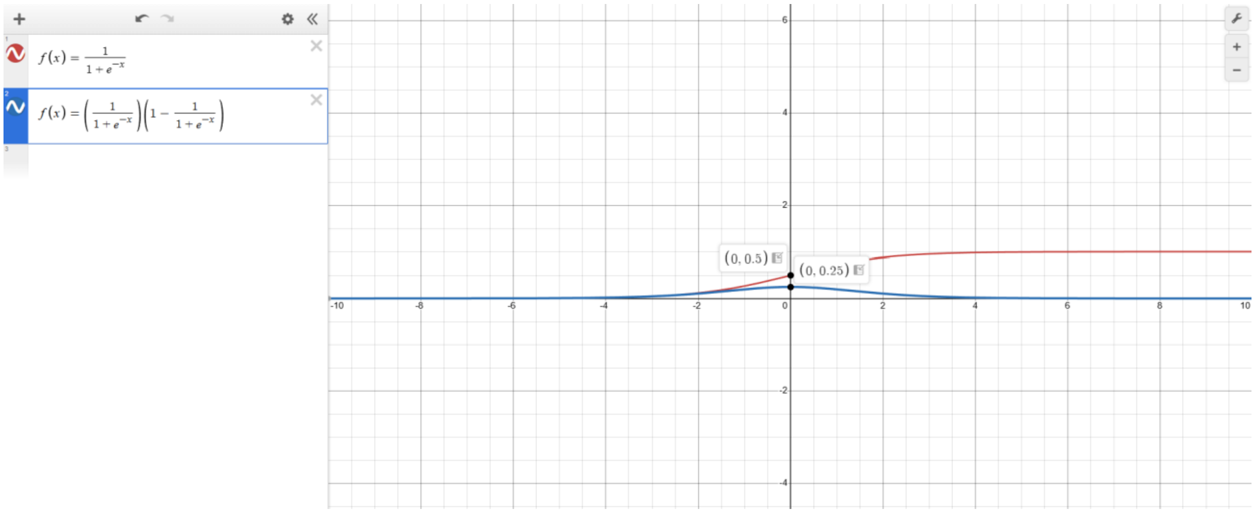
1. Why are activation functions used in neural networks?

   Activation functions are used in neural networks to introduce nonlinearity into the model. Without them, a neural network composed only of linear transformations would be equivalent to a single linear transformation, no matter how many layers it had, and thus could only model linear relationships. Nonlinear activation functions allow the network to learn and approximate complex, nonlinear mappings between inputs and outputs, enabling it to solve a wider range of tasks. Additionally, some activation functions (tanh or ReLU, for example) help control gradient flow and improve training dynamics.

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 $(-\infty, +\infty)$.

Note: In all plots, the activation function is in red with the derivative in blue

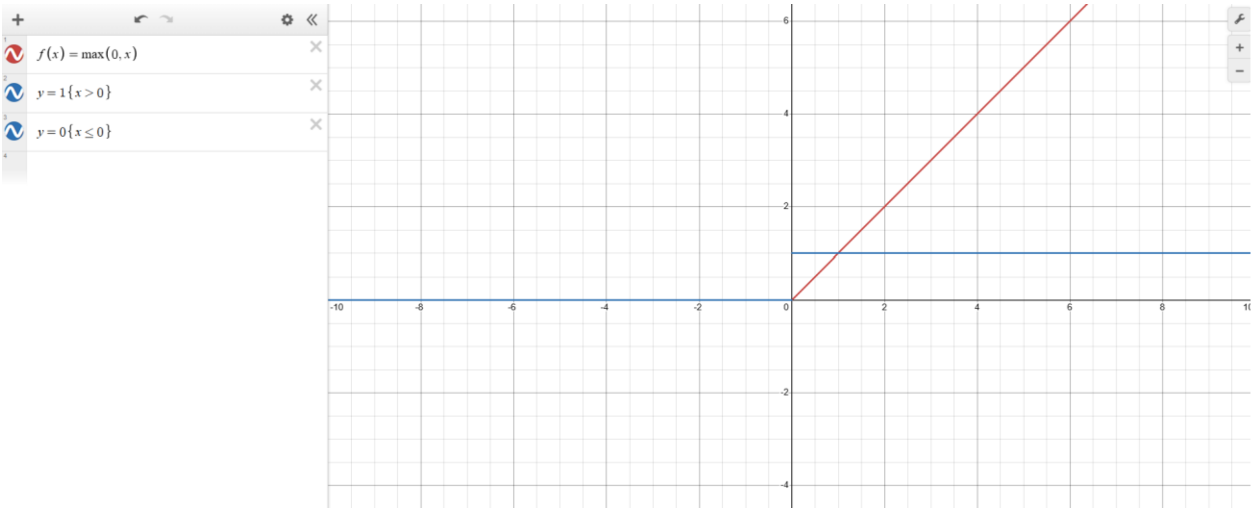
Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$, derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$



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

2 $f(x) = \left(\frac{1}{1+e^{-x}}\right)\left(1 - \frac{1}{1+e^{-x}}\right)$

ReLU: $ReLU(x) = \max(0, x)$, derivative: $ReLU'(x) = \begin{cases} 1 \; if \; x > 0 \\ 0 \; if \; x \leq 0 \end{cases}$

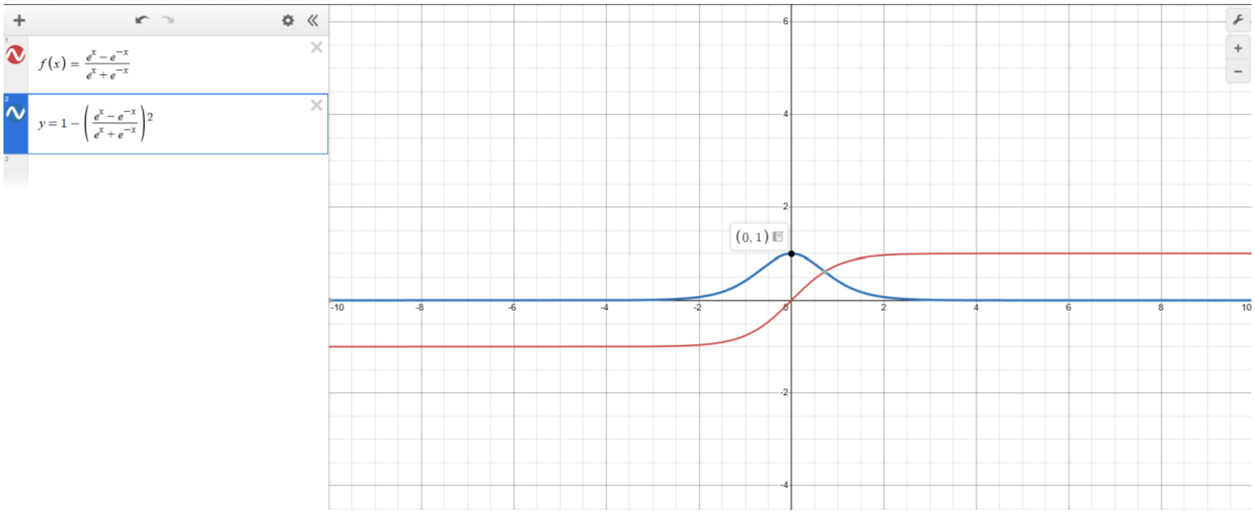


1 $f(x) = \max(0, x)$

2 $y = 1\{x > 0\}$

3 $y = 0\{x \leq 0\}$

Tanh: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$, derivative $tanh'(x) = 1 - \tanh^2(x)$


$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
$y = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2$
(0, 1)
6
4
2
-2
-4
-10
-8
-6
-4
-2
0
2
4
6
8
10

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).

   The vanishing gradient problem occurs when gradients become extremely small as they are backpropagated through many layers, causing the earlier layers to learn very slowly or not at all. This often happens in deep networks where repeated multiplication by small derivatives shrinks the gradient exponentially.

   Activation functions like sigmoid and tanh are prone to this issue because their derivatives are less than 1 in magnitude for most inputs, so repeated multiplications during backpropagation lead to vanishing gradients.

4. Why zero-centered activation functions impact the results of Backprop?

   Zero-centered activation functions impact backpropagation because they produce outputs with a mean around zero, which helps keep the gradients balanced during weight updates. If activations are not zero-centered (like sigmoid), the gradients for all weights in a layer tend to have the same sign, which can cause inefficient zigzagging in gradient descent and slow convergence. Zero-centered activations allow positive and negative gradients to cancel out appropriately, leading to faster and more stable learning.

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

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

where δ_{ij} is the Kronecker delta function.[1]

First, let $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_k e^{z_k}}, S = \sum_k e^{z_k}$, and as a result $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{S}$

We can differentiate w.r.t. z_j using the quotient rule:

$\frac{\partial \sigma_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{S}\right) = \frac{\delta_{ij}e^{z_i}S - e^{z_i}\frac{\partial S}{\partial z_j}}{S^2}$, where $\frac{\partial e^{z_i}}{\partial z_j} = \delta_{ij}e^{z_i}, \frac{\partial S}{\partial z_j} = e^{z_j}$

and so $\frac{d\sigma_\mathrm{i}}{dz_j} = \frac{\delta_{ij}e^{z_i}S - e^{z_i}e^{z_j}}{S^2} = \frac{e^{z_i}}{S}(\delta_{ij} - \frac{e^{z_j}}{S})$

but $\frac{e^{z_i}}{S} = \sigma_i$ and $\frac{e^{z_j}}{S} = \sigma_j$

Therefore $\frac{\partial \sigma(\mathbf{z})_i}{\partial z_j} = \sigma(\mathrm{z})_i\left(\delta_{ij} - \sigma(\mathrm{z})_j\right)$

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

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

where $\mathrm{diag}(.)$ turns a vector into a diagonal matrix. Also, note that $\mathbf{J}_\sigma(\mathbf{z}) \in \mathbb{R}^{K\times K}$..

We know the Jacobian has entries $J_\sigma(z)_{ij} = \frac{\partial \sigma_\mathrm{i}}{\partial z_j} = \sigma_i\delta_{ij} - \sigma_i\sigma_j$ (this uses the previous result)

So, in matrix form the term $\sigma_i\delta_{ij}$ corresponds to a diagonal matrix with entries $\sigma_1, \dots, \sigma_K$, or $diag(\sigma(z))$.

And the term $\sigma_i\sigma_j$ corresponds to the outer product: $\sigma(z)\sigma(z)^T$

Thus, combining both terms:

$$\mathrm{J}_\sigma(\mathrm{z}) = \mathrm{diag}\big(\sigma(\mathrm{z})\big) - \sigma(\mathrm{z})\sigma(\mathrm{z})^\top$$

Simulating XOR

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

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

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

Explain your reasoning using the following single-layer network definition: $\hat{y} = \text{ReLU}(W \cdot \mathbf{x} + b)$

No, a single layer network in the form of $\hat{y}$ cannot represent the XOR function. This is because $\hat{y}$ is a linear function of $x = [x_1, x_2]$ which defines a single linear decision boundary (line). Applying ReLU does not change the fact that the model is performing a linear separation followed by a monotonic transformation. This is a problem because the XOR function is not linearly separable. Therefore, a single-layer ReLU network cannot represent XOR.

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

$$\mathbf{h} = \text{ReLU}(W_1 \cdot \mathbf{x} + \mathbf{b}_1)$$
$$\hat{y} = W_2 \cdot \mathbf{h} + b_2$$

Note that this model has an additional layer compared to the earlier question: an input layer x ∈ $\mathbb{R}^2$, a hidden layer 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 b_1 and b_2 such that this model can accurately model the XOR problem.

For this, we use a network with 2 hidden units

First, let $W_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, b_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

then $h = ReLU(W_1 x)$, and the two hidden units compute $h_1 = ReLU(x_1 - x_2)$ and $h_2 = ReLU(x_2 - x_1)$

And now let $W_2 = [1 \quad 1], b_2 = 0$

so $\hat{y} = h_1 + h_2$.

This has the following input and outputs and matches the XOR function:

x_1	x_2	h_1	h_2	$\hat{y}$
0	0	0	0	0
1	0	1	0	1
0	1	0	1	1
1	1	0	0	0

3. Consider the same network as above (with ReLU activations for the hidden layer), with an arbitrary differentiable loss function $\ell: \{0,1\} \times \{0,1\} \rightarrow \mathbb{R}$ 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.

Using the network with initializations described, we start with forward pass.

For any input x: $W_1x + b_1 = 0$, so $h = ReLU(0) = 0$

Then $\hat{y} = W_2h + b_2 = 0$. The model outputs 0 for every input, regardless of x.

Next, we find gradients for W_2:

$\frac{\partial \hat{y}}{\partial W_2} = h = 0$ since $h = 0$

Thus, the gradient of the loss w.r.t. W_2 is 0 and W_2 never updates.

To find the gradient for W_1, we start by backpropagating $\frac{\partial \hat{y}}{\partial h} = W_2 = 0$ since $W_2 = 0$.

Thus, all gradients flowing back to W_1 are 0. Also, we know $ReLU'(0) = 0$ in practice, so the gradient is also blocked locally.

Therefore, $\nabla w_1 = 0$

All gradients for $W_1$and $W_2$are zero at initialization, so gradient descent makes no updates at all. The network remains stuck outputting 0 for every input and therefore cannot learn XOR

4. **Extra Credit:** Now let's consider a more general case than the previous question: we have the same network with an arbitrary hidden layer activation function:

$$\mathbf{h} = f(W_1 \cdot \mathbf{x} + \mathbf{b}_1)$$
$$\hat{y} = W_2 \cdot \mathbf{h} + b_2$$

Show that if the initial weights are any uniform constant, then gradient descent will not learn the XOR function from this initialization.

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
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

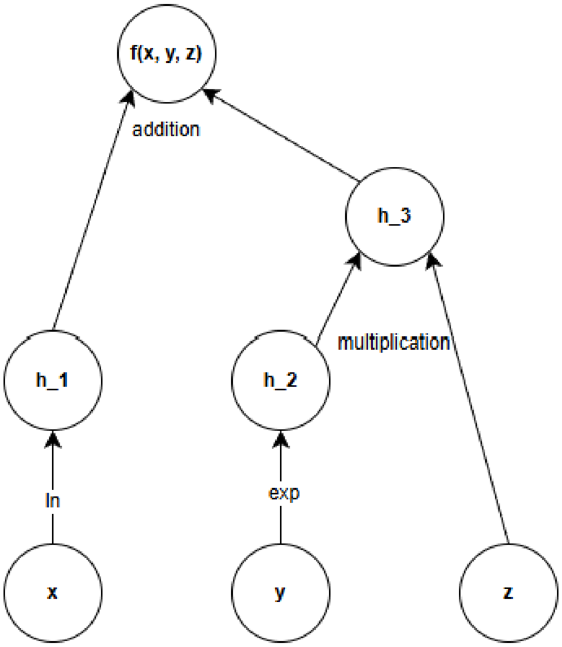
Backpropagation, it enables with grace
Making deep learning, a beautiful race

–ChatGPT Feb 3 2023

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.

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.



2. Run the forward propagation and evaluate f and h_i $(i = 1,2,\ldots)$ at $[x,y,z] = [1,3,2]$.

$$h_1 = \ln(1) = 0$$

$$h_2 = \exp(3) = e^3 \approx 20.085$$

$$h_3 = e^3 * 2 = 2e^3 \approx 40.171$$

$$f = 0 + 2e^3 = 2e^3 \approx 40.171$$

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 $[x, y, z] = [1,3,2]$.

   First, doing the local derivatives:

   $\mathrm{h}_1 = \ln(x), \frac{\partial h_1}{\partial x} = \frac{1}{x}$, at $x = 1$, $\frac{\partial h_1}{\partial x} = 1$

   $h_2 = \exp(y), \frac{\partial h_2}{\partial y} = \exp(y)$, at $y = 3$, $\frac{\partial h_2}{\partial y} = \mathrm{e}^3$

   $h_3 = h_2 z, \frac{\partial h_3}{\partial h_2} = z, \frac{\partial h_3}{\partial z} = h_2$, at $z = 2, h_2 = e^3$ we have $\frac{\partial h_3}{\partial h_2} = 2, \frac{\partial h_3}{\partial z} = e^3$

   For the final node,

   $f = h_1 + h_3, \frac{\partial f}{\partial h_1} = 1, \frac{\partial f}{\partial h_3} = 1$

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.

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

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial h_3} * \frac{\partial h_3}{\partial h_2} * \frac{\partial h_2}{\partial y} = 1 * 2 * e^3 = 2e^3$$

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial h_3} * \frac{\partial h_3}{\partial z} = 1 * e^3 = e^3$$

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 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

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).

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).

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.

Train and Evaluate MLPs

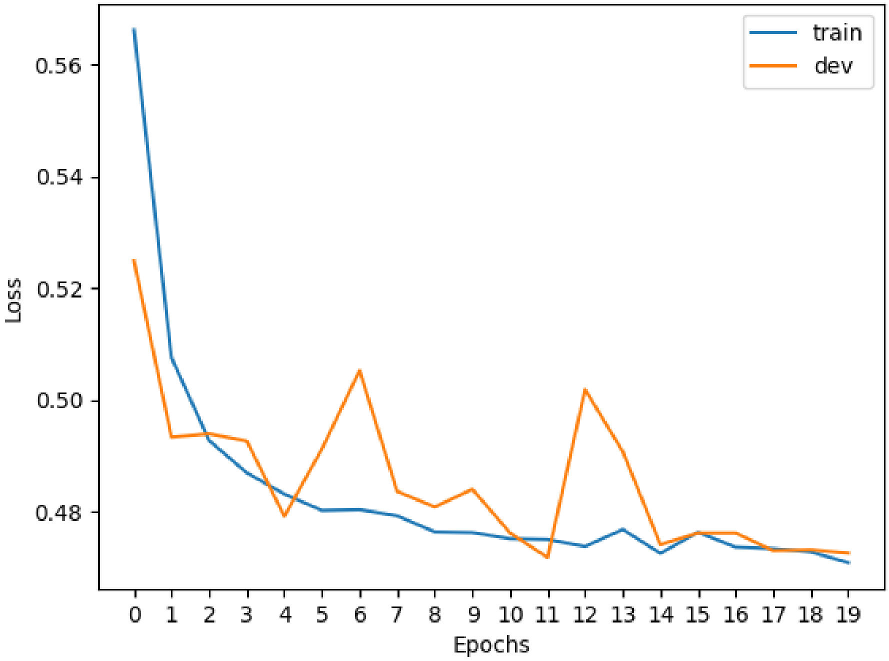
We provide in `main.py` several MLP configurations and corresponding recipes for training them.

TODOs Once you finished 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:

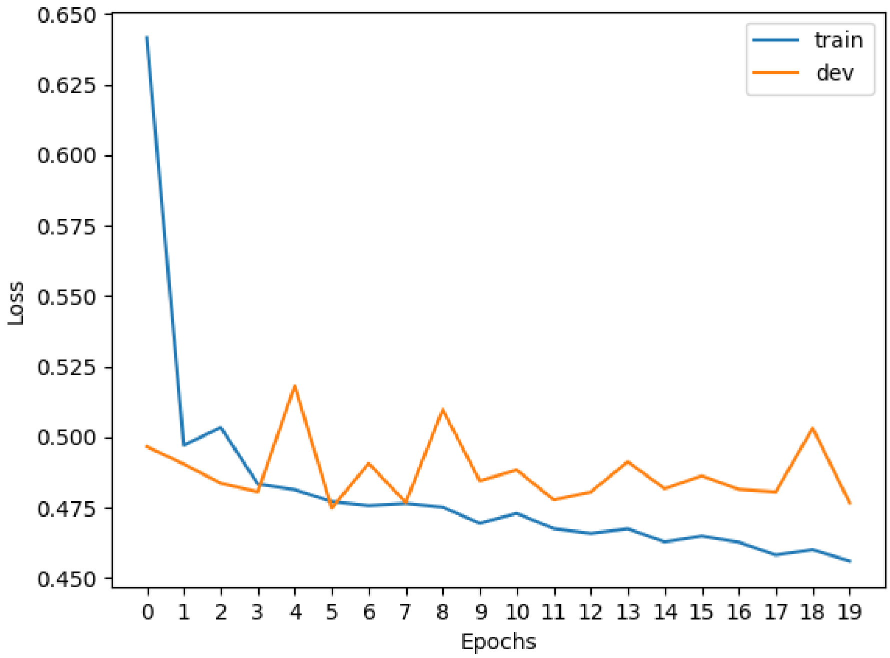
- 4 plots of train & dev loss for each MLP configuration
- 2 plots of dev losses and accuracies across MLP configurations

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

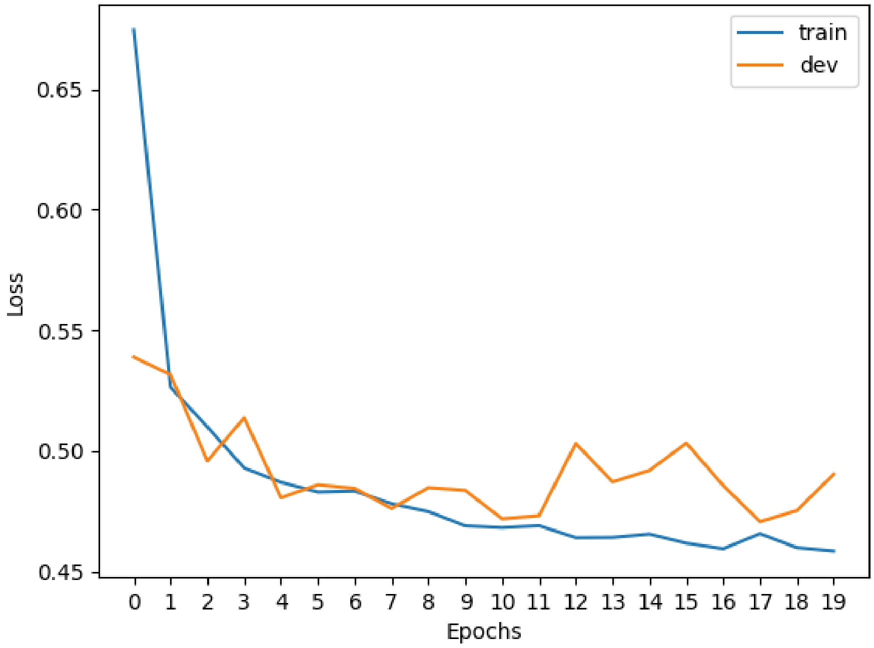
Loss plot for configuration with no hidden layers:



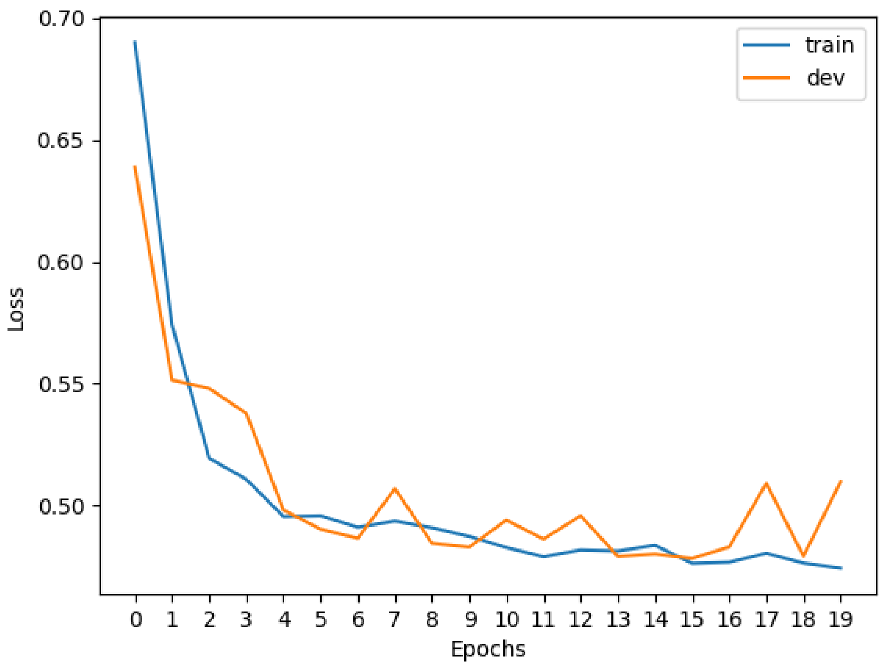
Loss for config with 1 hidden layer:



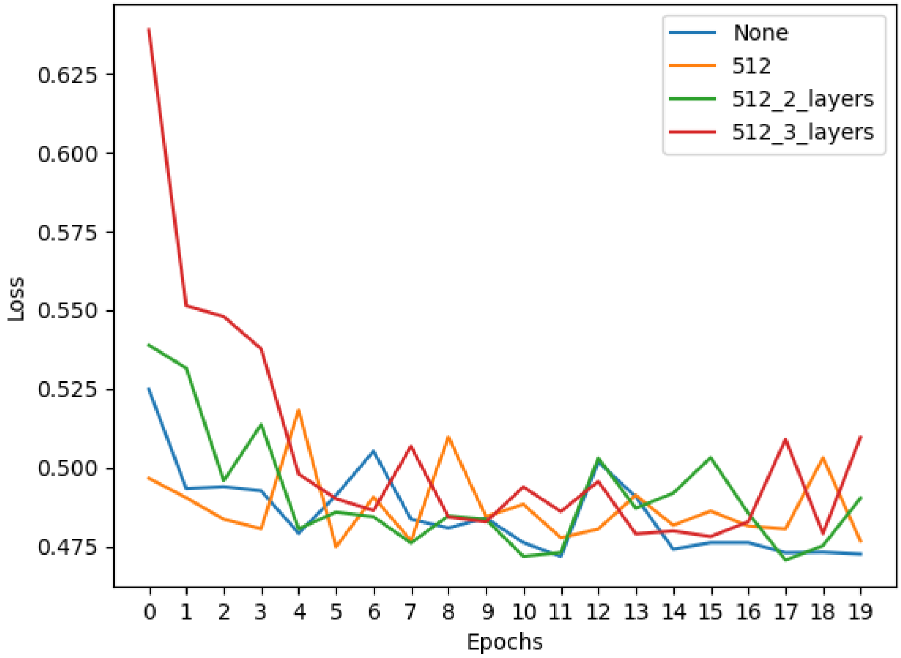
Loss for config with 2 hidden layers:



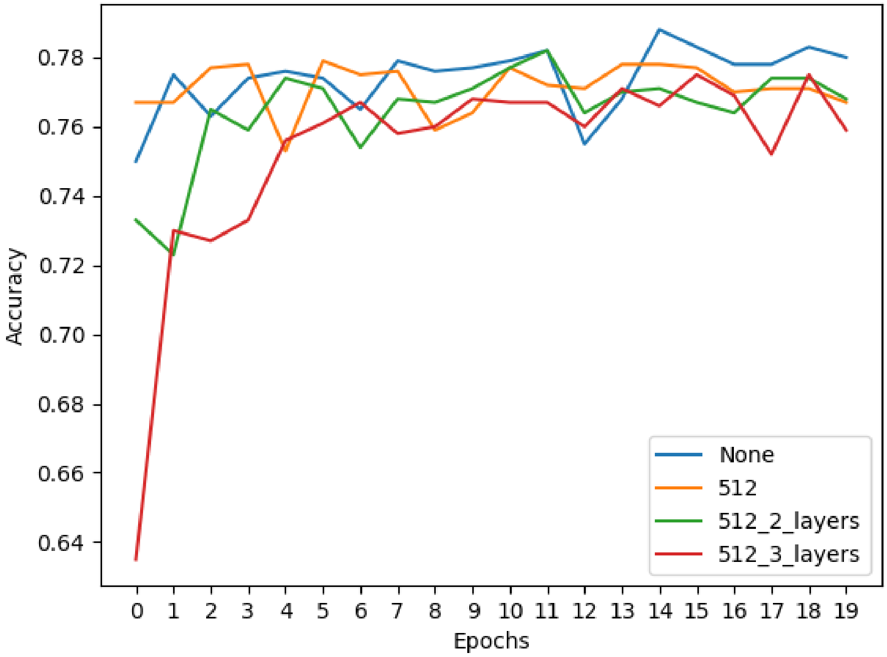
Loss for config with 3 hidden layers:



Dev loss for all configurations:



Accuracy for all configurations:



Based on these plots, I find it interesting that the network with no hidden layers achieves lower loss than deeper, more complex models. This may suggest that the task does not require highly complex feature hierarchies and that the underlying patterns are relatively simple or linearly separable. It could also indicate that the deeper models are overfitting the training data. Additionally, the more complex networks may be harder to optimize effectively, leading to poorer generalization compared to the simpler model. More testing would be required to know for sure what led to these results.

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 3. Now it is time to try them out in our MLPs!
Note: for the following TODO and the TODO in 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.

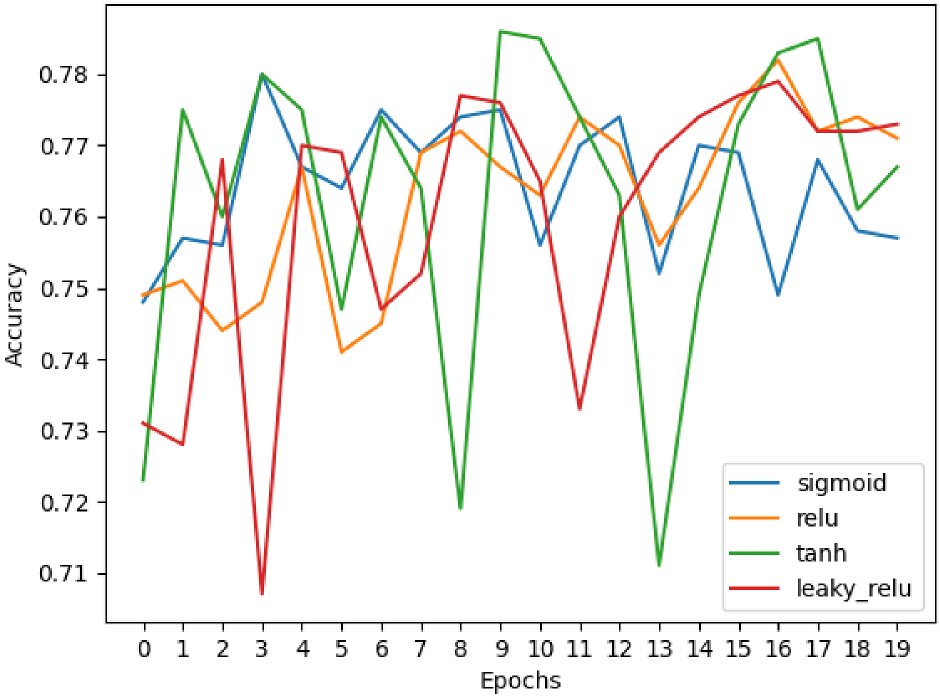
TODOs: Read and complete the missing lines of the two following functions:

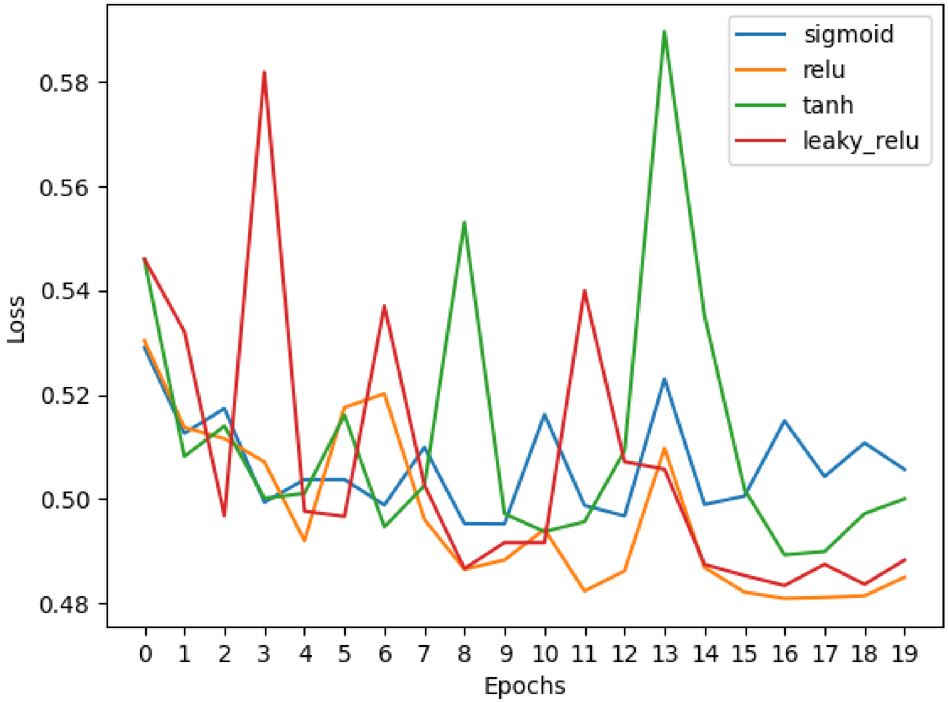
- `__init__` function of the `SentimentClassifier` class: define different activation functions given the input `activation` type.

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.





I found that all the alternative activation functions that I tested ended up achieving better results than the sigmoid activation function in the last epochs of training (16-19). ReLU achieved slightly better results than the other functions, and ReLU did not have the high spikes in loss during training that tanh and leaky ReLU did. The success of ReLU might suggest that the task benefits from sparse, piecewise-linear representations and does not require strong output centering like tanh provides.

Hyper-parameter Tuning: Learning Rate

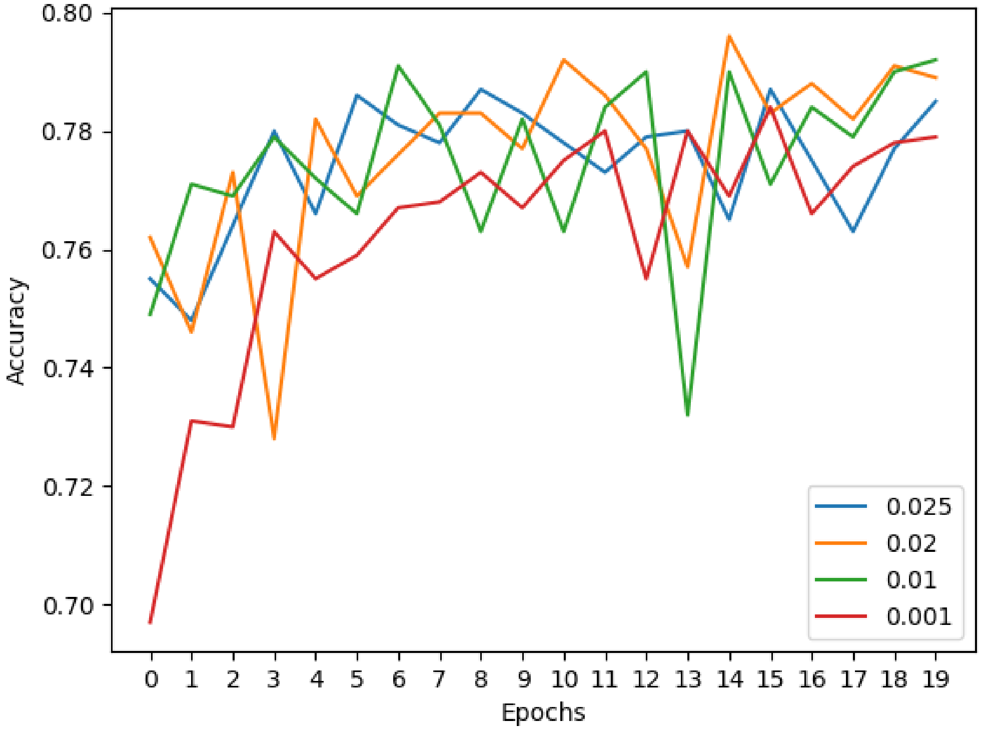
The training process mostly involves learning model parameters, which are automatically performed by gradient-based methods. However, certain parameters are “unlearnable" through gradient optimization while playing a crucial role in affecting model performance, for example, learning rate and batch size.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

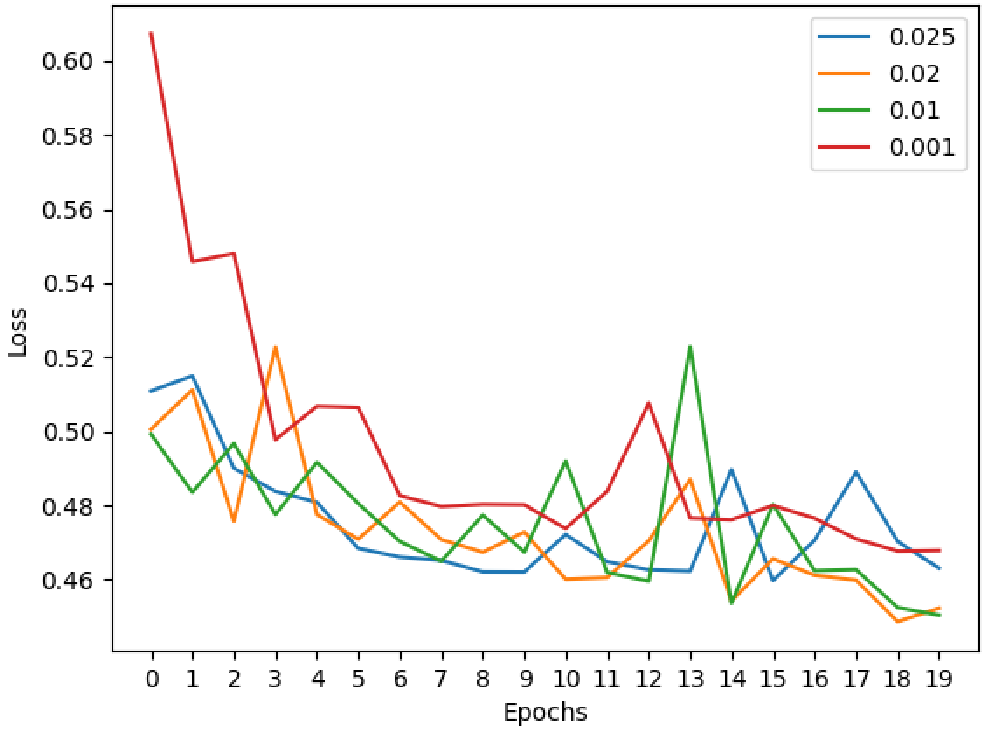
We will now take the first step to tune these hyper-parameters by exploring the choices of one of the most important one - learning rate, on our MLP. (There are lots of tutorials on how to tune the learning rate manually or automatically in practice, for example this note can serve as a starting point.)

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.





The plot shows that moderate learning rates (0.02 and 0.01) achieve the lowest dev loss and converge relatively smoothly, suggesting they provide a good balance between convergence speed and stability. The very small learning rate (0.001) decreases loss more slowly and remains higher overall, indicating slower learning rather than improved generalization. The largest learning rate (0.025) converges but exhibits more fluctuation, implying it is closer to instability, while overall the similar final losses suggest the task is not overly complex and is well-behaved.