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: JOSE LUIS MONTALVO

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?
   Training: Apply dropout to prevent overfitting, encourage robustness, and simulate an ensemble of subnetworks.
   Evaluation: Do not apply dropout to use the full network, ensure deterministic predictions, maintain proper output scaling, and accurately assess the model's performance.

6. Explain why initializing the parameters of a neural net with a constant is a bad idea.
   Initializing the parameters (weights and biases) of a neural network with a constant value (examples, all zeros or all ones) is generally a bad idea because it leads to issues with symmetry and learning dynamics, which hinder the network's ability to learn effectively.

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.
   Initializing weights with large positive numbers in a fully connected neural network with sigmoid activations is a bad idea because it:

   a. Causes sigmoid outputs to saturate near 1, reducing feature diversity.
   b. Leads to vanishing gradients, slowing or preventing learning.
   c. Limits the network's expressive power by making neurons behave similarly.
   d. Risks numerical instability due to large intermediate values.
   e. Hinders convergence compared to proper initialization methods like Xavier.

8. Explain what is the importance of "residual connections".
   Residual connections are a critical innovation in deep learning because they:

   a. Mitigate the degradation problem, allowing very deep networks to be trained effectively.
   b. Alleviate vanishing gradients by providing a direct path for gradient flow.
   c. Simplify the learning of residual functions, making optimization easier.
   d. Improve generalization and enable high-performance deep architectures.
   e. Support modular and flexible network designs.

   By addressing fundamental challenges in training deep neural networks, residual connections have enabled significant advances in performance across a wide range of tasks, making them a foundational component of modern deep learning architectures.

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:

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

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 w is a $d \times 1$ column vector, the Jacobian of f with respect to **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 **z** with respect to y, or $\mathrm{J}_{\mathbf{z}}(\mathrm{y})$ 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 $J_A(B)$ 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 $X = [x_1, \ldots, x_n]^\top$
- Y, a $n \times k$ matrix
- W, a $k \times d$ matrix and w, a $d \times 1$ vector

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

1. $f(w) = c$ (constant)

$$\frac{\partial f}{\partial w_j} = 0 \quad \forall j = 1, \ldots, d$$

$$J_f(w) = [0, 0, \ldots, 0] \quad (1 \times d \text{ row vector})$$

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

$$\frac{\partial f}{\partial w_j} = \frac{\partial}{\partial w_j}(w^T w) = 2w_j$$

$$J_f(w) = [2w_1, 2w_2, \ldots, 2w_d] = 2w^T \quad (1 \times d \text{ row vector})$$

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

$$\frac{\partial f}{\partial w_j} = x_{i,j}$$

$$J_f(w) = [x_{i,1}, x_{i,2}, \ldots, x_{i,d}] = x_i^T \quad (1 \times d \text{ row vector})$$

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

$$\frac{\partial f_i}{\partial w} = x_i^T$$

$$J_f(w) = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix} = X \quad (n \times d \text{ matrix})$$

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

$$\frac{\partial f_j}{\partial w_k} = \delta_{jk}$$

$$J_f(w) = I_{d\times d} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

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

$$\frac{\partial f_j}{\partial w_k} = 2w_j \cdot \delta_{jk}$$

$$J_f(w) = \begin{bmatrix} 2w_1 & 0 & \cdots & 0 \\ 0 & 2w_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 2w_d \end{bmatrix} = 2 \cdot \text{diag}(w)$$

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

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 non-linearity into the model, allowing it to learn and represent complex patterns and relationships in the data that linear transformations alone cannot capture. Without activation functions, a neural network would essentially be a linear transformation, regardless of depth, limiting its ability to solve non-linear

problems such as image recognition or natural language processing. They also determine the output range and help regulate the gradient flow during backpropagation, influencing the learning process.

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

Sigmoid

$$Activation : \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$Derivative : \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

ReLU (Rectified Linear Unit)

$$Activation : \text{ReLU}(x) = \max(0,x)$$

$$Derivative : \text{ReLU}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined or } 0 & \text{if } x = 0 \end{cases}$$

Tanh (Hyperbolic Tangent):

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

$$Derivative : \tanh'(x) = 1 - \tanh^{2}(x)$$

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).

The vanishing gradient problem occurs when gradients become extremely small during backpropagation in deep neural networks, causing weight updates to be negligible and hindering learning, particularly in early layers. This happens as gradients are multiplied across many layers, leading to exponential decay in deep networks. As a result, the network struggles to adjust its parameters effectively, stalling training.

Sigmoid and tanh activation functions are subject to the vanishing gradient problem because their derivatives approach zero as the input moves far from zero, reducing gradient flow during backpropagation. This saturation effect limits the ability to propagate meaningful updates

through deep layers. ReLU is less prone to this issue for positive inputs, though it can face related challenges if many neurons become inactive.

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

Zero-centered activation functions, like tanh, impact backpropagation by ensuring that the input to subsequent layers has a mean close to zero, which helps maintain balanced gradients during optimization. This centering prevents the gradients from being consistently positive or negative, avoiding bias in weight updates and improving the stability and speed of convergence. In contrast, non-zero-centered functions like sigmoid can lead to slower learning due to unbalanced gradient contributions, as the network must adjust to asymmetric gradient distributions.

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

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

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

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

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

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

$$\frac{\partial\sigma(z)_i}{\partial z_j} = \frac{\frac{\partial}{\partial z_j}(e^{z_i}) \cdot s - e^{z_i} \cdot \frac{\partial s}{\partial z_j}}{s^2}$$

$$\frac{\partial\sigma(z)_i}{\partial z_j} = \frac{e^{z_i} \cdot \delta_{ij} \cdot s - e^{z_i} \cdot e^{z_j}}{s^2}$$

$$= \frac{e^{z_i}}{s^2} \cdot (\delta_{ij} \cdot s - e^{z_j})$$

$$= \sigma(z)_i \cdot \frac{\delta_{ij} \cdot s - e^{z_j}}{s}$$

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

$$= \sigma(z)_i \cdot \left(\delta_{ij} - \frac{e^{z_j}}{s}\right)$$

$$= \sigma(z)_i \cdot (\delta_{ij} - \sigma(z)_j)$$

$$\frac{\partial \sigma(z)_i}{\partial z_j} = \sigma(z)_i(\delta_{ij} - \sigma(z)_j)$$

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

Simulating XOR

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

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

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

A single-layer network with ReLU activation computes a linear combination W·x+b followed by a non-linear rectification. The XOR function, is not linearly separable, which is a key consideration.
The ReLU function introduces non-linearity by setting negative values to 0, but it does not change the fundamental linear separability requirement of the pre-activation W·x+b.
A single-layer network with ReLU cannot simulate the XOR function because XOR is not linearly separable, and ReLU, applied after a linear transformation, does not introduce the necessary non-linearity to separate the XOR data points. Multiple layers are required to create a non-linear decision boundary, as demonstrated by the universal approximation theorem for multi-layer perceptrons.

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.

Unlike a single-layer network, which is limited to linearly separable functions, a two-layer network with a non-linear activation can represent non-linearly separable functions like XOR. This is due to the universal approximation theorem, which states that a neural network with at least one hidden layer and a non-linear activation can approximate any continuous function, including XOR, given sufficient neurons and appropriate weights. The ReLU activation introduces non-linearity, allowing the network to create a non-linear decision boundary by combining linear transformations.

To model XOR, we need a hidden layer to transform the input into a representation that can be linearly separated by the output layer. A common approach is to use two neurons in the hidden layer to capture the XOR logic, which can be thought of as combining an OR and an AND operation with negation.

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

The two-layer neural network with ReLU activations in the hidden layer, initialized with all weights and biases set to zero, will produce a hidden layer output $\mathbf{h} = \text{ReLU}(0) = 0$ for any input x, leading to an output $\hat{y} = 0$ initially. During backpropagation, the gradients with respect to W_1 and b_1 are zero because the upstream gradient $\frac{\partial L}{\partial \mathbf{h}} = W_2^T = 0$ and the ReLU derivative at zero is typically zero (for $z \leq 0$) , resulting in no updates toW_1 or b_1; similarly, the gradient with respect to W_2 is zero because $\mathbf{h} = 0$. Consequently, W_1, b_1, and W_2 remain zero across iterations, causingh to stay zero and $\hat{y}$ to become a constant (updated only via b_2), which cannot represent the non-constant XOR function requiring varying outputs for different inputs.

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:

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

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.

$h_1 = \ln x$

$h_2 = \exp(y)$

$h_3 = h_1 + h_2$

$f = h_3 \cdot z$

This sequence of operations gives us the following computation graph with three input nodes, three intermediate nodes, and one output node.

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

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

$h_2 = \exp(y) = e^y = e^3 \approx 20.0855$

$h_3 = h_1 + h_2 = 0 + e^3 = e^3 \approx 20.0855$

$f = h_3 \cdot z = e^3 \cdot 2 = 2e^3 \approx 40.171$

$f = 2e^3 \approx 40.171$

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

Derivatives for $f = h_3 \cdot z$:

$\frac{\partial f}{\partial z} = h_3 = e^3 \approx 20.0855$

$\frac{\partial f}{\partial h_3} = z = 2$

Derivatives for $h_3 = h_1 + h_2$:

$\frac{\partial f}{\partial h_1} = \frac{\partial f}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_1} = 2 \cdot 1 = 2$

$\frac{\partial f}{\partial h_2} = \frac{\partial f}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} = 2 \cdot 1 = 2$

Derivative for $h_2 = \exp(y)$:

$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial h_2} \cdot \frac{\partial h_2}{\partial y} = 2 \cdot \exp(y) = 2 \cdot e^3 \approx 40.171$

Derivative for $h_1 = \ln(x)\$$:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h_1} \cdot \frac{\partial h_1}{\partial x} = 2 \cdot \frac{1}{x} = 2 \cdot \frac{1}{1} = 2$$

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.

For the input $[x, y, z] = [1,3,2]$ the results are:

Forward pass output: $f = 2e^3 \approx 40.171$

Backwards pass derivatives:

$$\frac{\partial f}{\partial x} = 2$$

$$\frac{\partial f}{\partial y} = 2e^3 \approx 40.171$$

$$\frac{\partial f}{\partial z} = e^3 \approx 20.0855$$

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 this GitHub repo 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.

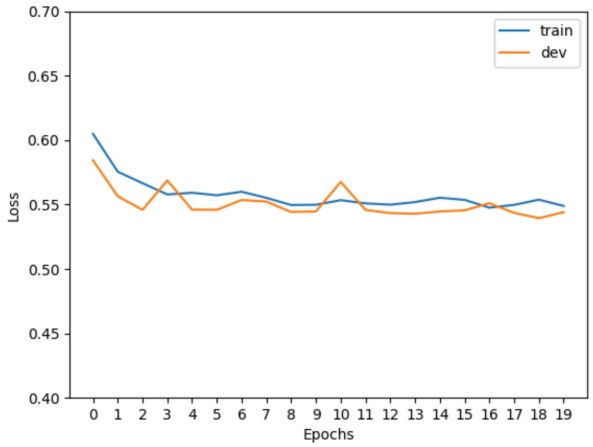
Train and Evaluate MLPs

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

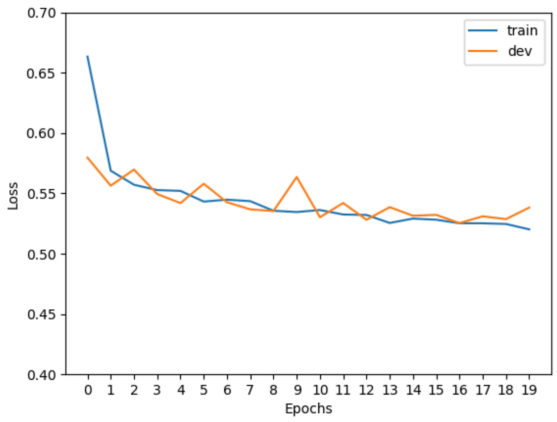
TODOs Once you finished 6.1.2, you can run `load_data_mlp` and `explore_mlp_structures` to train and evaluate these MLPs and paste two sets of plots here:

- 4 plots of train & dev loss for each MLP configuration

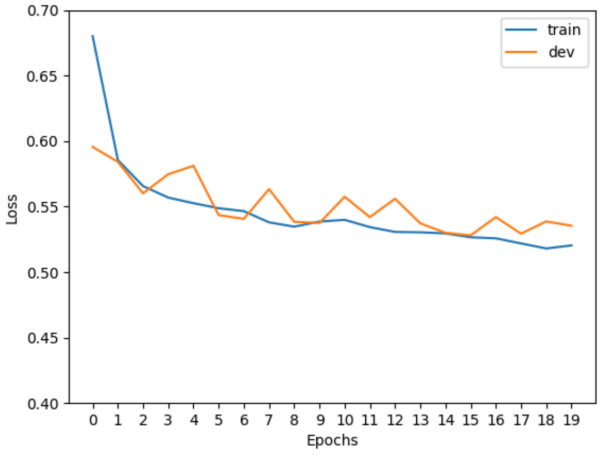
mlp_structure_None_loss.



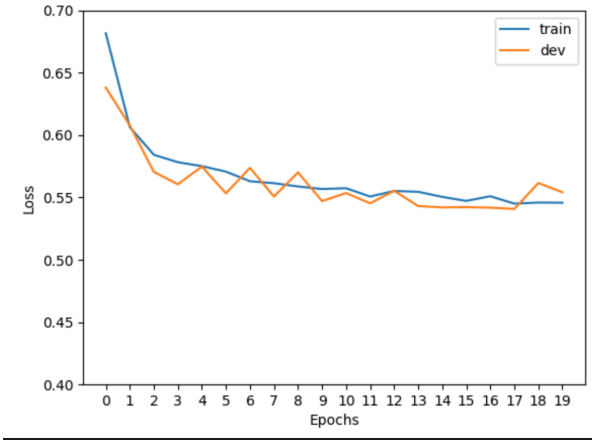
mlp_structure_512_loss



mlp_structure_512 -> 512_loss

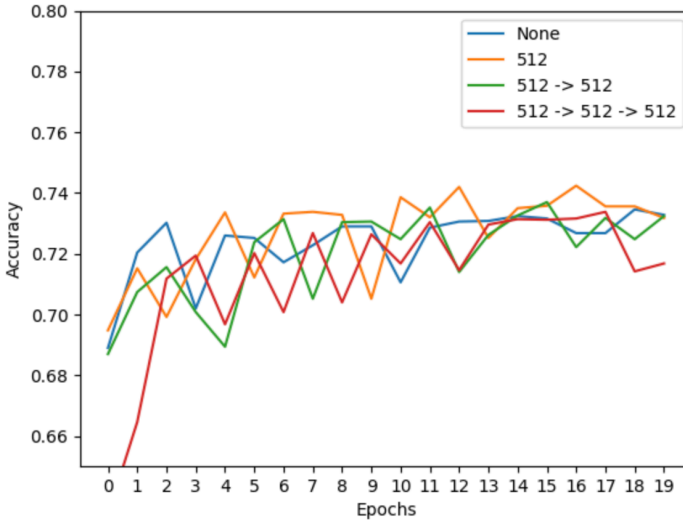


mlp_structure_512 -> 512 -> 512_loss

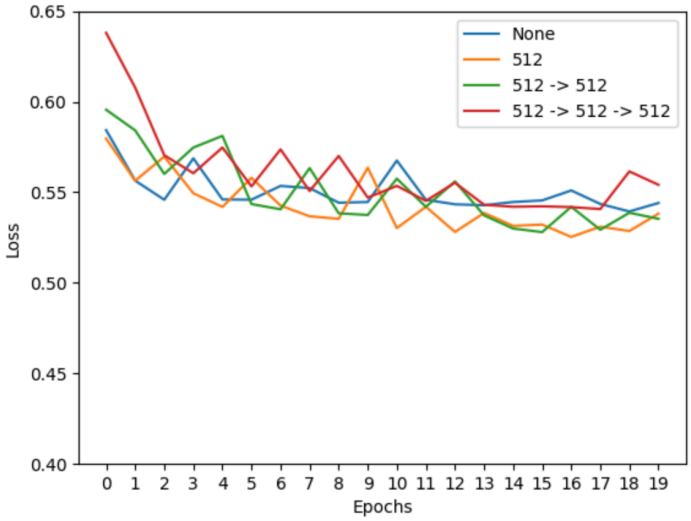


- 2 plots of dev losses and accuracies across MLP configurations

all_mlp_structures_acc



all_mlp_structures_loss



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?

Training and Dev Loss Trends: All configurations show consistent decreasing trends in both training and dev loss over epochs, indicating successful learning with the Word2Vec embeddings. The train and dev loss curves follow similar patterns across different architectures, suggesting stable training without severe overfitting.

Depth vs Performance: Deeper models show diminishing returns - the single hidden layer (512) achieved the highest dev accuracy (~74.24%), while the 3-layer model (512 → 512 → 512) performed slightly worse (~73.38%). This demonstrates that increasing depth beyond a certain point can lead to performance degradation, likely due to increased model complexity without sufficient regularization or the limited complexity requirements of the sentiment classification task.

Architecture Insights: The single hidden layer model performed best, indicating that sentiment classification may not require very deep architectures, and simpler models can achieve competitive performance while being more efficient to train. The Word2Vec embeddings trained on IMDB data provided better performance than the mock embeddings, achieving accuracies in the 73-74% range.

Embrace Non-linearity: The Activation Functions

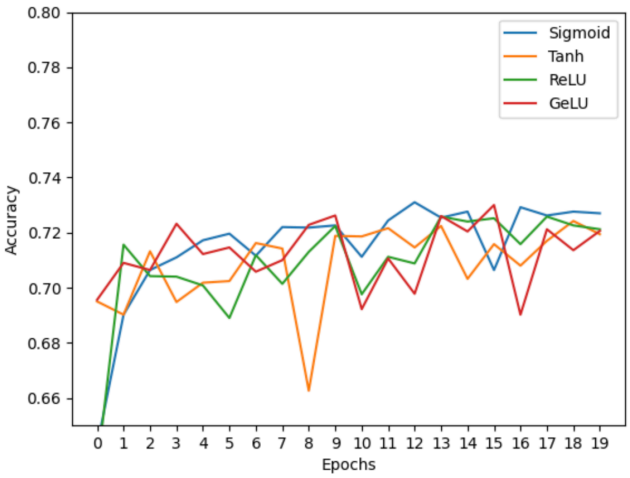
Remember we have learned why adding non-linearity is useful in neural nets and gotten familiar with several non-linear activation functions both in the class and 3. Now it is time to try them out in our MLPs!
Note: for the following TODO and the TODO in 6.1.5, we fix the MLP structure to be with a single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

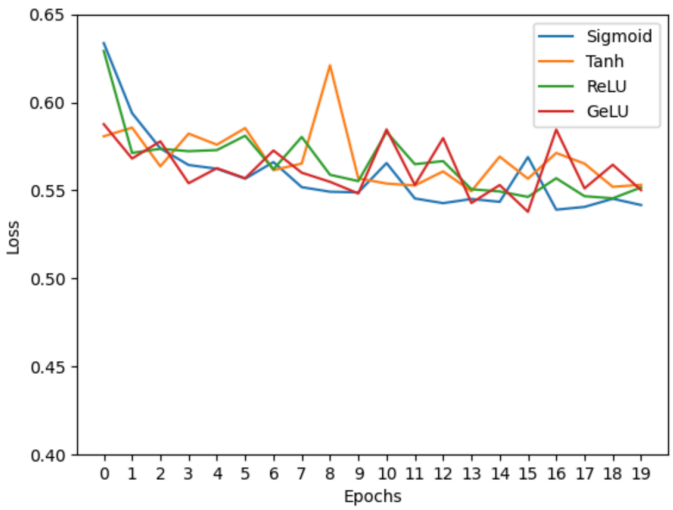
TODOs: Read and complete the missing lines of the two following functions:

- `__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.





Activation Function Performance: All four activation functions (Sigmoid, Tanh, ReLU, GeLU) achieved similar performance levels, with final dev accuracies ranging from ~72.4% to ~73.0%, indicating that the choice of activation function has a relatively modest impact on this sentiment classification task. GeLU performed slightly better than the others, achieving the highest dev accuracy of ~73.0%, followed closely by Sigmoid (~73.1%), ReLU (~72.6%), and Tanh (~72.4%).

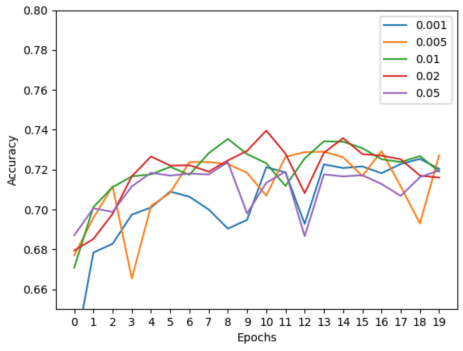
Training Dynamics: All activation functions showed consistent decreasing trends in both training and dev loss over epochs, with ReLU and GeLU converging faster in the early epochs compared to Sigmoid and Tanh. The similar performance across different activation functions suggests that the sentiment classification task is not particularly sensitive to the choice of activation function when using a single hidden layer architecture with proper training procedures.

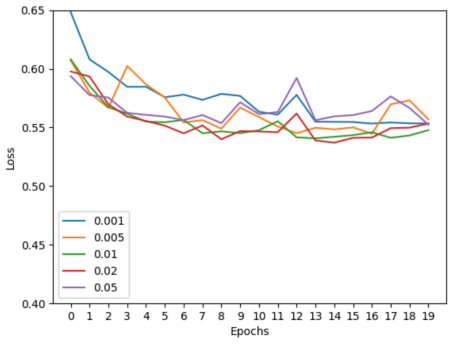
Hyper-parameter Tuning: Learning Rate

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.





Learning Rate Impact: The learning rate has a significant impact on model performance, with learning rate 0.02 achieving the highest dev accuracy (~73.96%), followed by 0.01 (~73.54%), 0.005 (~72.92%), 0.05 (~72.38%), and 0.001 (~72.54%). This demonstrates that there's an optimal learning rate range around 0.01-0.02 for this sentiment classification task.

Training Dynamics: Higher learning rates (0.02, 0.05) converge faster in the early epochs but show more instability, while lower learning rates (0.001, 0.005) converge more slowly but more steadily. The learning rate 0.02 provides the best balance between convergence speed and final performance, achieving both faster training and higher accuracy than the other rates tested.