MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

CENG 403

Spring 2024

Take-Home Exam 2

REGULATIONS

Due date: 23:59, 8 May, 2024, Wednesday *(Not subject to postpone)*

Submission: Electronically. You should save your Jupyter notebook without clearing the outputs as a ipynb file named `the2.ipynb`. Check the announcement on the ODTUCLASS course page for the submission procedure.

Team: There is **no** teaming up. This is an EXAM.

Cheating: Source(s) and Receiver(s) will receive zero and be subject to disciplinary action. You are NOT allowed to use code pieces from the Internet or AI tools.

1 OBJECTIVE

The main objective of the this take-home exam (THE) is to extend the capabilities of the CerGen tensor library, developed initially in THE1, by integrating functionalities crucial for training Multi-Layer Perceptrons (MLPs). This THE is designed to provide hands-on experience of the inner workings and optimization techniques of neural networks, including forward propagation, backpropagation, and gradient descent, without the aid of high-level deep learning frameworks like TensorFlow, PyTorch or NumPy.

2 TASK DESCRIPTION

In THE1, we implemented some crucial n-dimensional element-wise mathematical operations. Specifically, we utilized the Operation class as a base class for the operations in the Gergen class and used the `ileri()` functions for forward pass. In THE2, we will extend THE1 to train a simple MLP classifier with one hidden layer on the MNIST dataset. The MNIST dataset is a image collection of handwritten digits. These images have been normalized to a fixed size of 28×28 pixels and centered to ensure consistency.

To solve THE2, we need to follow a step-by-step procedure, starting by implementing our linear layers, followed by our MLP, activation, and loss functions. In addition, to apply gradient descent to our model, we will extend our Operation class implementation and implement the `geri()` function to apply backpropagation for our MLP (which works as a backward pass).

Once we have completed this THE, we can create and train our own MLP. Keep in mind that an MLP is a type of feedforward neural network in which all the layers are fully connected, utilizing a non-linear activation function. By completing this assignment, we can train our own MLP on a small portion of the MNIST dataset for a classification task without using any additional libraries.

2.1 Revisiting THE1

In THE1, we implemented several essential functions for, e.g., adding two Gergens, calculating the mean of a Gergen's elements, and finding the logarithm of a Gergen. To accomplish this, we employed an Operation class with an `ileri()` pass, which provides us with the necessary building blocks to perform a simple forward pass for a MLP. For instance, by using `ic_carpim()` and addition, we can compute $\mathbf{xW} + \mathbf{b}$, which corresponds to the output of the linear layer.

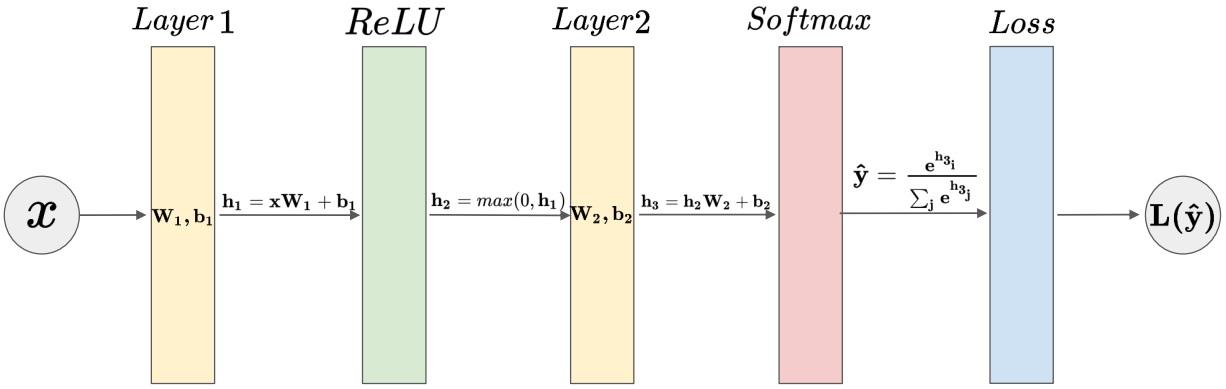


Figure 1: The forward pass of the two-layer MLP considered in THE2.

2.2 The Katman Class

First, you should implement the fundamental building block of neural networks: the Katman(Layer) class. Remember that a linear layer is a layer where all neurons in the layer have a connection to all the neurons in the previous layer. A linear layer can be formally described as a simple linear operation: $\mathbf{xW}+\mathbf{b}$, where the layer's input ($\mathbf{x}$), weights ($\mathbf{W}$) and biases ($\mathbf{b}$) are gergen objects.

The Katman class takes input size, output size, activation as arguments:

- `input_size` (int): The number of input features to the layer.
- `output_size` (int): The number of output features in the layer.
- `activation` (str, optional): The activation function to use. Supported values are 'relu' and 'softmax'. If None, no activation is applied.

With the given input and output sizes, you should initialize the layer's weighs and biases with appropriate shapes using the `rastgele_gercek()` function. You can choose the mean and the standard deviation of the initial parameters as you wish, e.g., using Xavier or He initialization methods.

For the forward pass, your Katman class should compute the multiplication of the inputs and weights then sum with biases, followed by the application of an activation function if one is specified (default is `None`). You can implement your own Matrix Multiplication operation in your gergen class in order to finalize the forward pass of Linear Layer.

2.3 MLP Class

The `MLP` class is designed to represent a basic Multilayer Perceptron (MLP) model. The class is initialized with three parameters: `input_size`, `hidden_size`, and `output_size`, which correspond to the dimensions of the input layer, hidden layer, and output layer, respectively.

Within the `__init__` method of the class, two layers are instantiated using the `Layer` class:

- `self.hidden_layer`: This is the hidden layer of the network that takes inputs from the input layer and applies a ReLU (Rectified Linear Unit) activation function to each neuron's output.

- `self.output_layer`: This is the output layer of the network that takes inputs from the hidden layer. The activation function used here is softmax, which is appropriate for multi-class classification tasks. Softmax converts the outputs (raw scores) to a probability distribution over the output classes. In cases where a MLP is used for regression or binary classification, the activation function may be omitted or replaced with a sigmoid function, respectively.

The class also includes a `ileri` method that defines the forward pass of the MLP. As illustrated in Figure 1, the forward pass of our MLP model is methodically outlined. Starting with an input gergen $\mathbf{x}$, it is processed through the first layer to yield the output $h_1 = \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1$. This output is then passed through a ReLU activation function, generating $h_2 = \max(0, h_1)$, to introduce non-linearity into the model. Subsequently, h_2 is transformed by the second layer to produce $h_3 = h_2\mathbf{W}_2 + \mathbf{b}_2$. A softmax function is then applied to h_3 to derive the predictive probability vector, which gives the probability distribution across the classes. Finally, the loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ is computed, which measures the error between the predicted output $\hat{\mathbf{y}}$ and the true labels $\mathbf{y}$, guiding parameter optimization through backpropagation.

2.4 Activation Functions

To be able to learn non-linear functions, you need to use two activation functions: (1) ReLU (Rectified Linear Unit), $relu(x) = \max(0, x)$, which is a simple piece-wise linear function. (2) Softmax, which is commonly used in the output layer of a classifier to obtain a probability distribution over classes: $\hat{y}_i = \frac{\exp(h_i)}{\sum_j \exp(h_j)}$. You are recommended to implement both of these functions as a subclass of the Operation class.

2.5 Loss Functions

To evaluate the prediction of our MLP and use gradient descent, we will use a loss function, specifically the Cross-Entropy Loss in this THE. Using $\hat{y}$ to denote the prediction probabilities obtained from the softmax function of the output Layer, $\mathbf{y}$ the correct class probabilities ($y_i = 1$ for the correct class and $y_i = 0$ for other classes), and assuming that there are n classes, we can define the Cross-Entropy Loss as follows for one sample:

$$\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i). \tag{1}$$

You should implement the loss function using the operations we already implemented in the Gergen library.

2.6 Backpropogation via the Operation and Gergen Classes

2.6.1 Extending the Operation Class by Implementing `geri()`

Recall that the `Operation` class serves as a base class for operations in the `gergen` class. In THE1, we defined it as:

```
class Operation:
    def __call__(self, *operands):
        self.operands = operands
```

```
        self.outputs = None
        return self.ileri(*operands)

    def ileri(self, *operands):
        raise NotImplementedError
```

With this structure, we can implement the forward pass of our network. However, to train our network, we need to extend our classes to support backpropagation. To achieve this, we need to add the `geri()` function. In `geri()`, you need to implement the calculation of gradients related to a given operation and pass the gradients for each operand for their gradient calculation.

```
def geri(self, grad_input):
    # grad_input: Gradient of the loss wrt the output of this operation.
    # Task: Calculate and pass the gradient
    #       to each operand correctly
```

We can give an example with an addition operation. In the Add operation, we have two operands, let us say **a** and **b**. The `ileri()` function returns the result as $\mathbf{r} = \mathbf{a}+\mathbf{b}$. In the `geri()` function, we need to calculate:

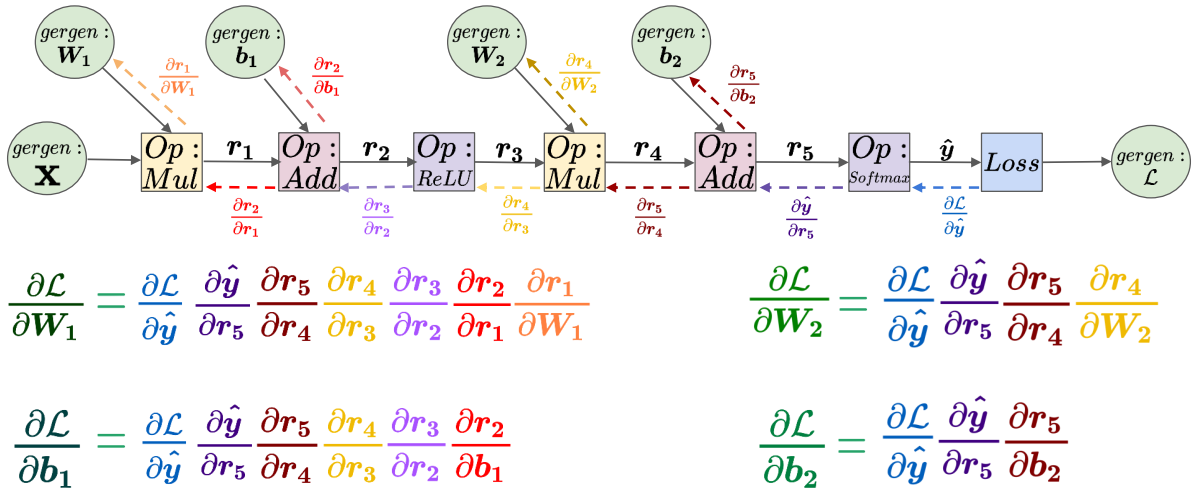
$$\frac{\partial \mathcal{L}}{\partial \mathbf{r}}\frac{\partial \mathbf{r}}{\partial \mathbf{a}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \mathbf{r}}\frac{\partial \mathbf{r}}{\partial \mathbf{b}}, \tag{2}$$

where $\frac{\partial \mathcal{L}}{\partial \mathbf{r}}$ is essentially `grad_input` of the `geri()` function. $\frac{\partial \mathcal{L}}{\partial \mathbf{r}}\frac{\partial \mathbf{r}}{\partial \mathbf{a}}$ is the gradient input that should be passed to gergen **a** and $\frac{\partial \mathcal{L}}{\partial \mathbf{r}}\frac{\partial \mathbf{r}}{\partial \mathbf{b}}$ is to gergen **b**. (Explanation in Section 2.6.2)

To complete this THE, you are expected to develop only the `geri()` functions that are essential for training our MLP. You can implement the `geri()` calculations for other operations, which, however, will not be graded.

Please note that in THE1, we have implemented our operations for several cases, such as 1-D Gergens, 2-D Gergens or N-D Gergens. However, in this take-home exam, we will mainly be working with 2-D Gergens. To simplify the process, you may keep only the necessary implementations and modify your `ileri()` functions accordingly.

2.6.2 Extending Gergen class with `turev_al()`



$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W_1}} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{r_5}}\frac{\partial \boldsymbol{r_5}}{\partial \boldsymbol{r_4}}\frac{\partial \boldsymbol{r_4}}{\partial \boldsymbol{r_3}}\frac{\partial \boldsymbol{r_3}}{\partial \boldsymbol{r_2}}\frac{\partial \boldsymbol{r_2}}{\partial \boldsymbol{r_1}}\frac{\partial \boldsymbol{r_1}}{\partial \boldsymbol{W_1}} \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W_2}} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{r_5}}\frac{\partial \boldsymbol{r_5}}{\partial \boldsymbol{r_4}}\frac{\partial \boldsymbol{r_4}}{\partial \boldsymbol{W_2}}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b_1}} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{r_5}}\frac{\partial \boldsymbol{r_5}}{\partial \boldsymbol{r_4}}\frac{\partial \boldsymbol{r_4}}{\partial \boldsymbol{r_3}}\frac{\partial \boldsymbol{r_3}}{\partial \boldsymbol{r_2}}\frac{\partial \boldsymbol{r_2}}{\partial \boldsymbol{b_1}} \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b_2}} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{r_5}}\frac{\partial \boldsymbol{r_5}}{\partial \boldsymbol{b_2}}$$

Figure 2: Forward and Backward Passes in terms of Gergen and Operation classes.

In order to update the weights and biases in our network, we will use backpropagation with the chain rule. Figure 2 shows our general MLP forward and backward pass visualized in detail by using the Operation and Gergen classes. The forward pass relies on passing information through Operation instances (Matrix multiplication 'Mul', addition 'Add', ReLU, and Softmax).

For the backward pass (i.e., for calculating the gradients), the chain rule is employed:

- The gradient of the loss with respect to $\boldsymbol{W_2}$ is $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W_2}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \boldsymbol{r_5}} \cdot \frac{\partial \boldsymbol{r_5}}{\partial \boldsymbol{r_4}} \cdot \frac{\partial \boldsymbol{r_4}}{\partial \boldsymbol{W_2}}$.
- Similarly, the gradient for $\boldsymbol{b_2}$ is computed as $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b_2}}$.
- The gradients for $\boldsymbol{W_1}$ and $\boldsymbol{b_1}$ are determined through the propagation of partial derivatives backward through each operation in the graph.

These gradients are utilized to update the network parameters $\boldsymbol{W_1}, \boldsymbol{W_2}, \boldsymbol{b_1}$ and $\boldsymbol{b_2}$ during the training via optimization algorithms like gradient descent, aiming to minimize the loss $\mathcal{L}$.

To do so, after completing the `geri()` function for calculating the gradients based on their operations, we now need to implement `turev_al()` to compute the derivative of our Gergen concerning all gergens that were involved in computing the current Gergen, leveraging the chain rule. To effectively use this method, you must understand that each gergen operation should be kept as part of a computational graph. The computational graph tracks how elementary operations modify data throughout the function. We need to extend our `gergen` class to keep track of this computational graph.

We introduce three new attributes for this THE:

- `operation`: This attribute holds a reference to an Operation object. The operation represents the computational operation that produces this Gergen's value when performing the forward pass. For example, this could be an addition, multiplication, or an activation function.
- `turev`: This attribute holds the derivative (gradient) of the node concerning a particular variable. Initially, it is set to `None`, indicating that the gradient has not been calculated yet. During the backward pass, this attribute should be modified and updated.
- `turev_gerekli`: This boolean flag determines whether the gradient for this Gergen needs to be calculated. If the value of `turev_gerekli` is True, the system will compute the gradients for this Gergen during the backward pass. If the user does not specify a value for `turev_gerekli`, it should be set to True.

When invoking `turev_al`, you should ensure that the computational graph is properly constructed, meaning that all operations leading to the gergen on which `turev_al` is called are recorded. This includes operations like addition and multiplication. The method then systematically calculates the derivative of each operation with respect to its inputs and accumulates these derivatives using the chain rule.

In practice, using `turev_al` requires keeping track of the operations history for each tensor within the computational graph. This history is used to apply the chain rule correctly, propagating gradients from the output back to the inputs.

2.7 Data Handling and Training Process

2.7.1 Implementing the Training Process with `egit()`

After finalizing the implementation of our MLP, the subsequent step involves the development of the training function. The `egit()` function implements the MLP's training process, necessitating several parameters for its operation:

- `mlp`: Our custom MLP model.
- `inputs` (list or generator): Inputs provided to the MLP.
- `targets` (list or generator): Correct / target values.
- `epochs` (int): Total epochs for training.
- `learning_rate` (float): Learning rate (step size).

Our MLP model should have an input size that is the product of the image height and width (for MNIST, it should be $28 \times 28 = 784$) and an output size of 10 (for 10 digits). You can define the hidden layer size as desired.

The training function executes on a per-epoch basis, handling the training data in single-instance batches (batch size is defined as one). Each batch is processed through the following sequence of steps:

1. **Forward pass:** The `mlp.ileri` function is subsequently invoked, propelling the input gergen through the network to obtain output predictions.
2. **Loss computation:** Utilizing the `cross_entropy` function, resulting in the loss gergen computation.
3. **Backward pass:** The `loss.turev_al` method is activated. This engenders backpropagation by computing the gradients of the loss in relation to all model parameters marked with `turev_gerekli` set as `True`.
4. **Parameter update:** Adjust the model's weights and biases by subtracting the scaled gradients with learning rate.
5. **Cleanup:** Do not forget to reset the weights and biases' operational attributes, and their gradients after the completion of the update.

To optimize your model configuration, you can define a grid that varies your hyperparameters within a small predefined range. For each combination of hyperparameters, train your model and observe the resulting loss curve. The goal is to identify the set of hyperparameters that minimizes the loss and produces the most effective model. To do this, you should try given different values for the learning rate and hidden layer size:

1. `learning_rate`: 0.01, 0.001, 0.001, 0.0001.
2. `hidden_size`: $5, 10, 30$.

For each setting, you will train your model for 10 epochs. You should plot the loss curve that achieves the best model performance. You can also define more tuning to achieve better performance.

2.7.2 Training and Testing Data

You have been provided with two files named **train_data.csv** and **test_data.csv** that contain training and testing data, respectively. Your task is to read these files and convert them into Python lists first, and then to initialize your input and target gergens. The first column of the CSV files represents the target value of the given image, while the remaining columns represent the pixel values. Keep in mind that you should implement an iterator for the Gergen class for employing a for loop.

3 MLP with PyTorch

Lastly, we will define and train an MLP class using the PyTorch library, ensuring it has the same architecture as our custom MLP implementation.

To do this, we need to define a new MLP class using the PyTorch library and train it with the same hyperparameters as our custom MLP implementation. We should implement the `MLP_torch` class, which takes the same arguments as our custom MLP implementation, namely `input_size`, `hidden_size`, and `output_size`. In the notebook provided, you will find functions related to torch implementation that require completion. You need to complete these functions.

4 GRADING/SPECIFICATIONS

- You will receive a Jupyter Notebook (.ipynb) file that contains the skeleton of the tensor library. Additionally, a solution to THE1 is provided for you. You can use your own Gergen implementations, solutions from your friends, or the provided one.
- It's okay if your MLP is slow. You don't have to worry about time complexity.
- Your task is to complete the implementation of the provided classes and functions.
- You should use Python 3.6+. You can use Matplotlib for plotting the loss curve, Pandas for reading data and Sklearn for one hot encoding.
- You should save your class and method implementations as a ipynb file named `the2.ipynb`.
- Please note that some parts of THE2 are purposely left unspecified, such as how you handle the backpropagation process or handle cases in your 'ileri' functions. It is important to mention that the Gergen class might be unclear in certain instances, especially when it comes to shapes or operations. However, feel free to modify or implement the Gergen class in a way that benefits you. During the process of implementing forward and backward passes, there may be inconsistencies with shapes, but you are free to address and fix these issues as you see fit. You can choose your own design options when implementing these model training extensions. Please use comment lines in your code to explain your choices. Small differences in your implementations are not important, and the main goal is to train an MLP from scratch.