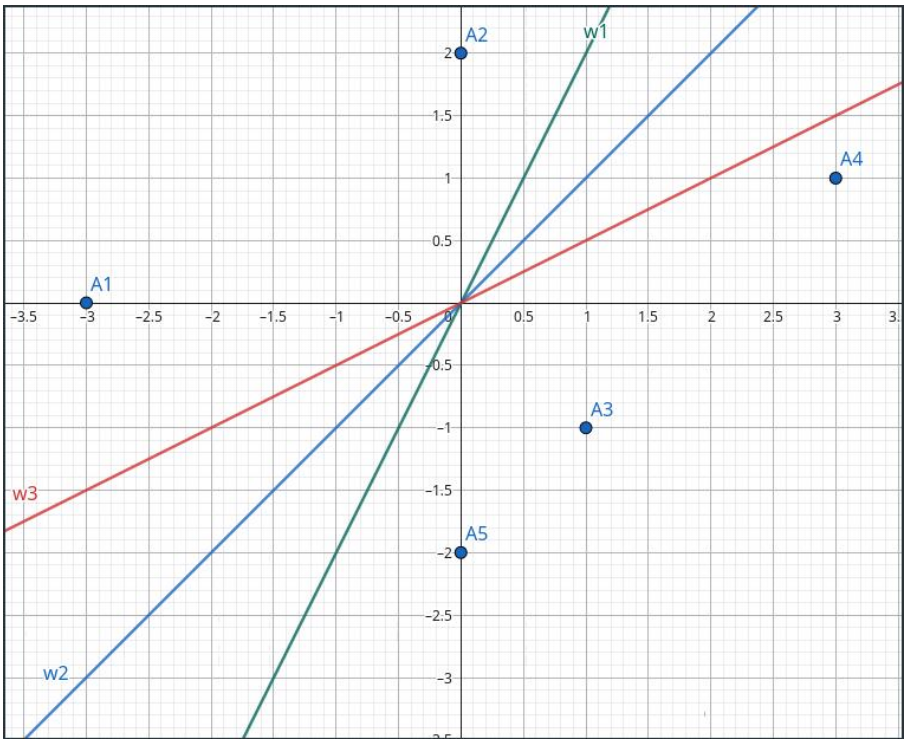
Project 5: Machine Learning

1 Binary Perceptron:

1.1 Phát biểu bài toán:

Xét không gian Euclid $\mathbb{R}^n$ với tích trong thông thường (tích chấm), cho tập hợp k vector $\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_k$ và gán tùy ý mỗi vector $\mathbf{x}_i$ với một nhãn (label) bất kỳ y_i (y_i chỉ nhận giá trị 1 hoặc -1). Hỏi có tồn tại một siêu mặt phẳng (hyperplane) nào đó có vector pháp tuyến $\mathbf{w}$ phân tách được các điểm A_i (ứng với vector $\mathbf{x}_i$) có giá trị trọng số khác nhau không?



Hình 1: Minh họa bài toán trong không gian $\mathbb{R}^2$

1.2 Hướng giải quyết bài toán:

Một siêu mặt phẳng tổng quát trong không gian vector $\mathbb{R}^n$ có phương trình dạng:

$$b + a_1x_1 + a_2x_2 + ... + a_{n-1}x_{n-1} + a_nx_n = 0$$

Vector pháp tuyến của mặt phẳng này chính là ma trận $\mathbf{w}$:

$$\mathbf{w} = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_{n-1} & a_n \end{bmatrix}^\top$$

Siêu mặt phẳng trên chia không gian $\mathbb{R}^n$ thành 2 nửa:

$$b + a_1x_1 + a_2x_2 + ... + a_{n-1}x_{n-1} + a_nx_n > 0 \quad (1)$$

$$b + a_1x_1 + a_2x_2 + ... + a_{n-1}x_{n-1} + a_nx_n < 0 \quad (2)$$

Xét một vector $\mathbf{x}$ bất kỳ có tọa độ:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_{n-1} & x_n \end{bmatrix}^\top$$

Từ định nghĩa tích trong thông thường, ta có:

$$\langle \mathbf{x}, \mathbf{w} \rangle = a_1 x_1 + a_2 x_2 + ... + a_{n-1} x_{n-1} + a_n x_n$$

Vậy hiển nhiên, nếu $b + \langle \mathbf{x}, \mathbf{w} \rangle > 0$ thì vector này thuộc nửa không gian (1), và $b + \langle \mathbf{x}, \mathbf{w} \rangle < 0$ thì nó thuộc nửa không gian (2).
Chọn các hệ số b và a_k ngẫu nhiên, siêu mặt phẳng luôn tồn tại với trường hợp chỉ có một vector $\mathbf{x}$. Không mất tính tổng quát ta giả sử $\mathbf{x}$ thuộc nửa không gian (1), với giá trị $y = +1$; ta quy ước gán cố định nhãn $+1$ cho nửa không gian (1), và -1 cho nửa không gian (2).
Xét trường hợp tổng quát, giả sử ta có hệ k vector $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k$ với các nhãn tương ứng $y_1, y_2, \cdots, y_k$. Sau khi thực hiện quy trình trên với vector $\mathbf{x}_1$, ta lần lượt xét các bất phương trình:

$$y_i(b + \langle \mathbf{w}, \mathbf{x}_i \rangle) > 0 \quad (i = 2, 3, \cdots, k) \tag{3}$$

Nếu (3) đúng, hiển nhiên $\mathbf{w}$ đã được chọn đúng; còn nếu (3) sai, ta cần phải điều chỉnh lại một vài tham số nào đó trong $\mathbf{w}$ cho đến khi bất phương trình trên đúng. Dễ dàng nhận thấy rằng nếu $k > n$, hệ bất phương trình có số phương trình nhiều hơn số ẩn nên tồn tại khả năng các miền nghiệm không giao nhau, dẫn đến $\mathbf{w}$ có thể không xác định.

1.3 Thuật toán học máy: Perceptron

Algorithm 1 Binary Linear Regression

Require: Training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, with $x_i \in \mathbb{R}^n$, $y_i \in \{+1, -1\}$
Ensure: Vector w will classify all of the data points correctly (if exists), and b is a bias value

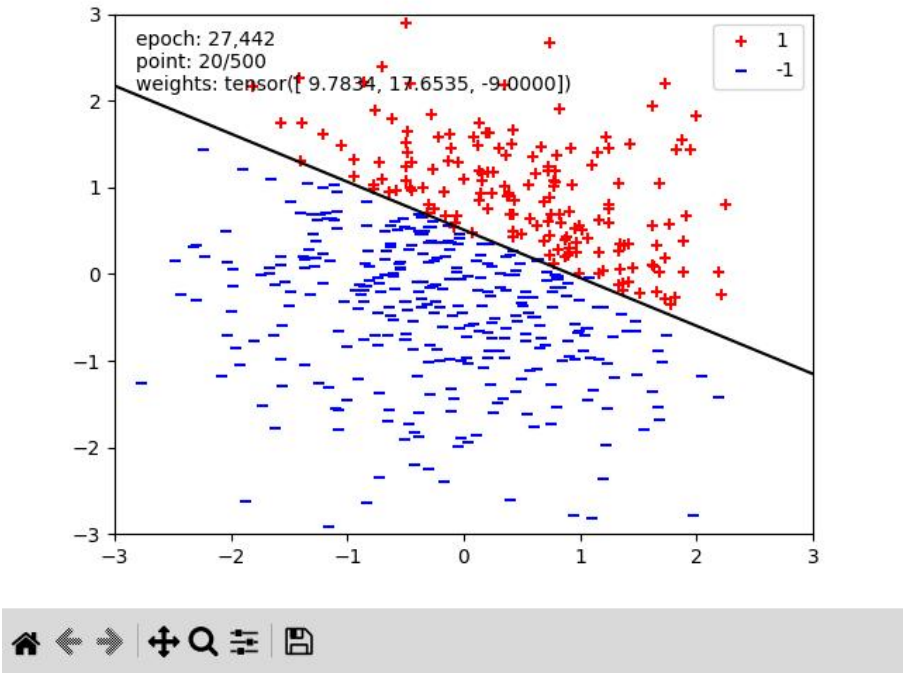
```
Initialize w ← 0
Initialize b ← 0
while true do
    mistake ← 0
    for each (x_i, y_i) ∈ D do
        if y_i · ⟨w, x_i⟩ ≤ 0 then
            w ← w + y_i x_i
            b ← b + y_i
            mistake ← mistake + 1
        end if
    end for
    if mistake = 0 then
        break
    end if
end while
return (w, b)
```

1.4 Thực thi giải thuật bằng Python

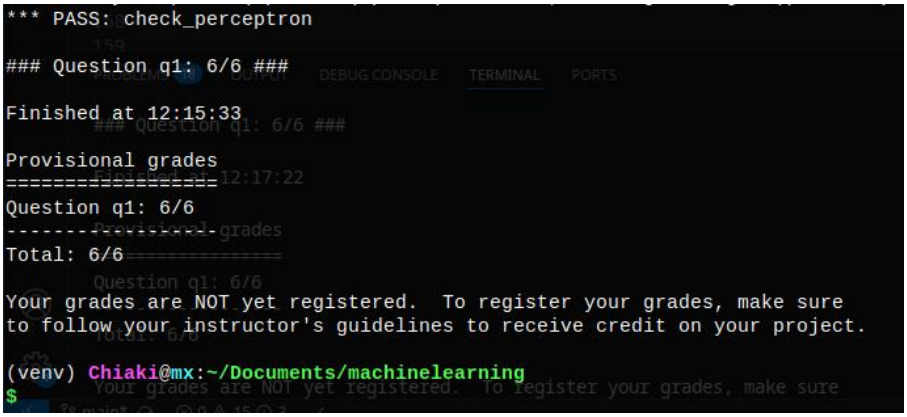
```
class PerceptronModel(Module): # class definition
    def __init__(self, dimensions): # dimensions = n
```

```
#  this function allows only one parameter, so we just ignore bias b from now on.

        super(PerceptronModel, self).__init__()
        self.w = Parameter(torch.ones(1, dimensions)) # w in R^n, x has shape (1,n)
                                                      # w is a learnable parameter

    def get_weights(self):
        return self.w   # return learned vector w

    def run(self, x):
        return (x*self.w).sum() # return inner product <w, x>

    def get_prediction(self, x):
        score = self.run(x)
        if score.item() >= 0:
            return 1
        else:
            return -1

        # checking the score item value of <w, x> positive or negative

    def train(self, dataset):   # uploading training data
        with no_grad():
            dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
            while True: # implement the algorithm above
                mistakes = 0
                for sample in dataloader:
                    x = sample['x']
                    y = sample['label'].item()

                    score = self.run(x)
                    if y*score.item() <= 0:
                        self.w += y*x
                        mistakes += 1
                if mistakes == 0:
                    break
```

1.5 Kết quả



Hình 2: Quá trình thực thi code



Hình 3: Final result, 6/6 testcases passed

2 Non-linear regression:

2.1 Phát biểu bài toán:

Ước lượng xấp xỉ hàm $\sin(x)$ trên đoạn $[-2\pi, 2\pi]$ sử dụng neural network từ một tập dữ liệu rời rạc cho trước dưới dạng $(\mathbf{x}, \mathbf{y})$.
Định nghĩa: một mạng lưới neural đơn giản là một hàm ước lượng xấp xỉ gồm 2 lớp (layers), lớp phi tuyến và lớp tuyến tính. Lớp tuyến tính được sử dụng để thực hiện các phép toán tuyến tính, còn lớp phi tuyến được dùng để ước lượng xấp xỉ.

$$\mathbf{f}(\mathbf{x}) = \text{relu}(\mathbf{x} \cdot \mathbf{W}_1 + \mathbf{b}_1) \cdot \mathbf{W}_2 + \mathbf{b}_2 \tag{4}$$
$$\text{relu}(x) = \max(x, 0)$$

Ta có thể thêm nhiều lớp để ước lượng chính xác hơn:

$$\mathbf{f}(\mathbf{x}) = \text{relu}(\text{relu}(\mathbf{x} \cdot \mathbf{W}_1 + \mathbf{b}_1) \cdot \mathbf{W}_2 + \mathbf{b}_2) \cdot \mathbf{W}_3 + \mathbf{b}_3 \tag{5}$$

2.2 Hướng giải quyết bài toán:

Để đơn giản, ta chỉ xem xét hướng giải cho trường hợp neural network đơn giản nhất, từ đó tổng quát hóa bài toán với trường hợp mạng có nhiều lớp hơn.
Ta định nghĩa hàm loss như sau:

$$\mathscr{L} = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i))^2 \tag{6}$$

Điều kiện lý tưởng nhất để $\mathscr{L}$ nhỏ nhất là từng hàm loss của các neuron con trong mạng lưới cũng phải đạt giá trị cực tiểu.
Ta xét hàm loss của một neural con:

$$L = \frac{1}{2}(y - f(x))^2 \tag{7}$$

Với:

$$f(z) = \text{relu}(z)w_2 + b_2$$
$$z = xw_1 + b_1$$

Ta tìm gradient descend của từng biến trong hàm L:

$$\nabla_{w_1} L = \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}\frac{\partial z}{\partial w_1} = (y - f(x))w_2 x 1_{z \geq 0}$$
$$\nabla_{b_1} L = \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}\frac{\partial z}{\partial b_1} = (y - f(x))w_2 1_{z \geq 0}$$
$$\nabla_{w_2} L = \frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_2} = (y - f(x))\text{relu(z)}$$
$$\nabla_{b_2} L = \frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial b_2} = (y - f(x))$$

Vector ∇L chỉ hướng có độ dốc lớn nhất trong không gian $\mathbb{R}^4$, khi ta cập nhật từng giá trị:

$$\begin{aligned} w_1 &\leftarrow w_1 - \eta \nabla_{w_1} L \\ b_1 &\leftarrow b_1 - \eta \nabla_{b_1} L \\ w_2 &\leftarrow w_2 - \eta \nabla_{w_2} L \\ b_2 &\leftarrow b_2 - \eta \nabla_{b_2} L \end{aligned} \tag{8}$$

Khi đó L dần tiến về 0, với η là một hằng số cho trước (learning rate).
Tương tự như vậy, ta có thể tổng quát hóa bài toán cho mạng n lớp.

2.3 Thuật toán học máy: Non-linear regression

Algorithm 2 Training a One-Hidden-Layer Neural Network

Require: Dataset $\{(x_i, y_i)\}_{i=1}^{N}$, learning rate $\eta > 0$
Ensure: Learned parameters W_1, b_1, W_2, b_2
  Initialize W_1, b_1, W_2, b_2 randomly
  repeat
    Forward pass:
    for $i = 1$ to N **do**
      $h_i \leftarrow x_i W_1 + b_1$
      $a_i \leftarrow \text{relu}(h_i)$
      $\hat{y}_i \leftarrow a_i W_2 + b_2$
    end for
    Compute loss:

$$L \leftarrow \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

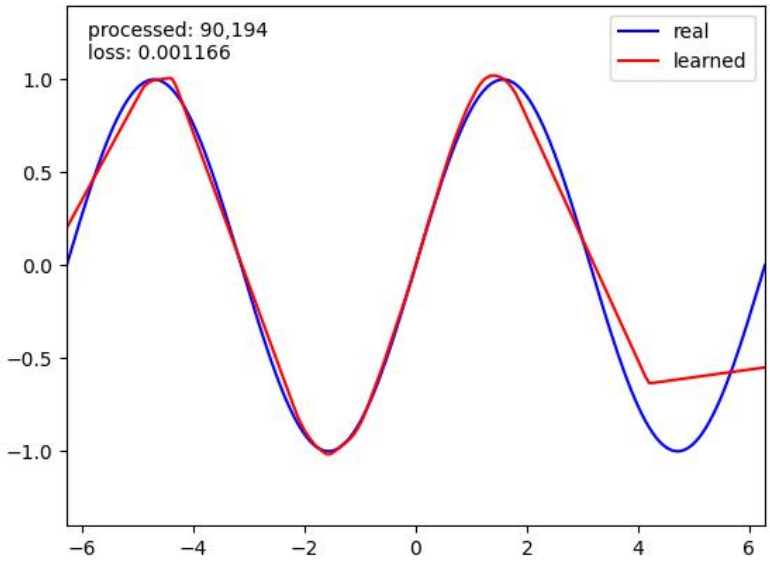
    Backward pass (compute gradients):
    Compute $\nabla_{W_2} L$, $\nabla_{b_2} L$
    Compute $\nabla_{W_1} L$, $\nabla_{b_1} L$ using chain rule
    Gradient descent update:
    $W_1 \leftarrow W_1 - \eta \nabla_{W_1} L$
    $b_1 \leftarrow b_1 - \eta \nabla_{b_1} L$
    $W_2 \leftarrow W_2 - \eta \nabla_{W_2} L$
    $b_2 \leftarrow b_2 - \eta \nabla_{b_2} L$
  until L converges (or $L < \varepsilon$)

2.4 Thực thi giải thuật bằng Python

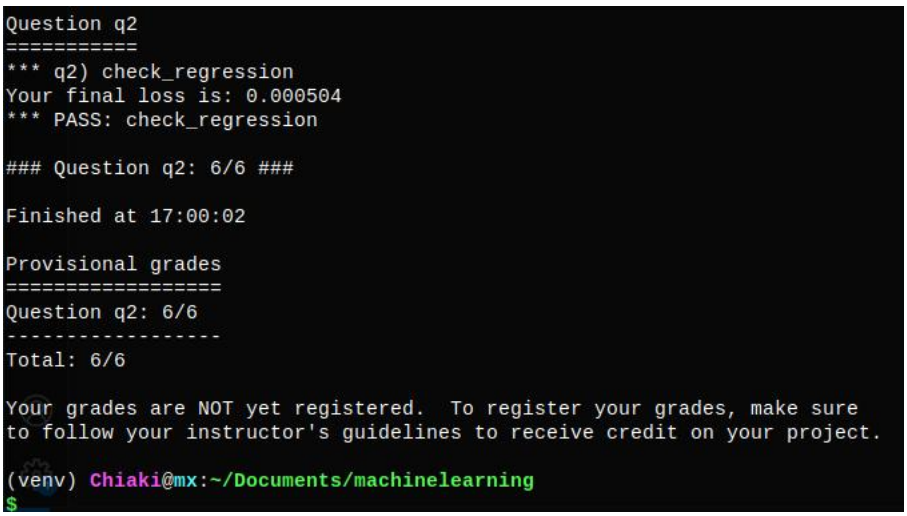
```
class RegressionModel(Module):
    def __init__(self):
        super().__init__()
        hidden_size = 200 # x in R^200
        self.fc1 = Linear(1, hidden_size) # initialize W1, b1 in R^200
                                          # and assign x = x*W1 + b1

        self.fc2 = Linear(hidden_size, 1) # initialize W2 in R^(200*1), b2 in R

    def forward(self, x):
        return self.fc2(relu(self.fc1(x))) # f(x) = relu(xw1 + b1)w2 + b2

    def get_loss(self, x, y): # loss function
        pred = self.forward(x)
        return mse_loss(pred, y)



    def train(self, dataset):
        dataloader = DataLoader(dataset, batch_size=32, shuffle=True) # 32 samples
```

```
optimizer = optim.Adam(self.parameters(), lr=0.001) # theta<-theta - eta*nabla
while True:
    total_loss = 0.0
    for sample in dataloader:
        x = sample['x']
        y = sample['label']
        optimizer.zero_grad() # assign zero to grad at first
        loss = self.get_loss(x, y) # core algorithm here
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    if total_loss < 0.02:
        break
```

2.5 Kết quả



Hình 4: Quá trình thực thi code



Hình 5: Final result

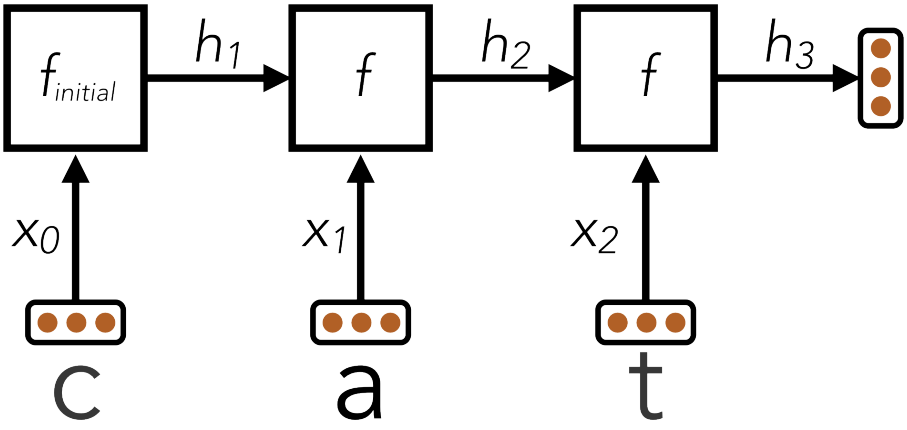
3 Language Identification:

3.1 Phát biểu bài toán:

Cho một tập dữ liệu như sau:

Word	Language
discussed	English
enternidad	Spanish
paileis	Dutch
...	...

Thiết kế một mạng lưới neural hồi quy (RNN - Recurrent Neural Network) để phân loại các từ vựng theo đúng ngôn ngữ của chúng.
Định nghĩa: mạng lưới RNN có kiến trúc như sau:



Hình 6: RNN architecture

Kiến trúc mạng RNN có thể "nhớ" (lưu trữ) dữ liệu từ các trạng thái trước thông qua cơ chế hồi quy.

3.2 Hướng giải quyết bài toán

Dữ liệu từ vựng (words) được biểu diễn dưới dạng một ma trận khối có cấu trúc như sau với tổng số từ k:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \cdots & \mathbf{x}_k \end{bmatrix}_{1\times k}$$

Mỗi từ $\mathbf{x}_j$ trong dữ liệu là một ma trận khối của các chữ cái cấu thành với độ dài L:

$$\mathbf{x}_j = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_L \end{bmatrix}_{1\times L}$$

Mỗi chữ cái x_j là một ma trận mã hóa one-hot nhị phân được mã hóa theo quy tắc bit "1" ở vị trí có kí tự trong bảng chữ cái và "0" ở các vị trí còn lại:

$$x_j = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix}^\top_{V\times 1}$$

Ta chọn thiết kế mạng neural từ hàm khởi tạo:

$$h_1 = f_{\text{intial}}(x_1) = \text{relu}(x_1\mathbf{W}_x + \mathbf{b}_h) \tag{9}$$

Hàm f là một hàm hợp đệ quy của các hàm trước đó:

$$h_i = f(x_i, h_{i-1}) = \text{relu}(x_i\mathbf{W}_x + h_{i-1}\mathbf{W}_h + \mathbf{b}_h) \tag{10}$$

Chính vì cơ chế đặc biệt này nên mạng RNN có khả năng ghi nhớ thông tin, không bị mất mát trong quá trình học. Output của quá trình này là một vector h_L (đã tổng hợp lại được toàn bộ thông tin của từ vựng được nhập) có kích thước $1 \times d$ với d là một hằng số chọn trước (kích thước của hidden layer).
h_L tiếp tục được xử lý qua lớp tuyến tính (linear layer):

$$\hat{y} = h_L\mathbf{W}_{out} + \mathbf{b}_{out} \tag{11}$$

$\hat{y}$ chính là **vector xác suất dự đoán đầu ra** y của từ vựng. Ta xác định hàm cross-entropy loss giữa giá trị dự đoán và thực tế:

$$\text{L}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log \hat{y}_i \tag{12}$$

Ta sử dụng thuật toán lan truyền ngược (backpropagation) để tìm ra các giá trị gradient rồi cập nhật các tham số:

$$\theta \leftarrow \theta - \eta\nabla_\theta L \tag{13}$$

Ràng buộc về kích thước của các ma trận được cho trong bảng sau:

Ma trận	Kích thước
x	$1 \times V$
$\mathbf{W}_x$	$V \times d$
$\mathbf{b}_h$	$1 \times d$
h_i	$1 \times d$
$\mathbf{W}_h$	$d \times d$
$\mathbf{b}_{out}$	$1 \times C$
$\mathbf{W}_{out}$	$d \times C$

Algorithm 3 Train an RNN to classify words by language

Require:
Dataset of word sequences $X = [x_0, x_1, ..., x_{L-1}]$ with labels y
d = hidden state size, η = learning rate, *epochs* = number of training epochs
Ensure: Trained parameters of RNN: $W_x, W_h, b_h, W_{out}, b_{out}$
Initialize RNN parameters $W_x, W_h, b_h, W_{out}, b_{out}$ randomly
for *epoch* $= 1$ **to** *epochs* **do**
for each batch of words and labels (X, y) in Dataset **do**
Initialize hidden states $h_0 = 0$
for $i = 0$ **to** $L - 1$ **do** ▷ iterate through letters in word
if $i = 0$ **then**
$h_0 \leftarrow f_{\text{initial}}(x_0) = \text{ReLU}(x_0 W_x + b_h)$
else
$h_i \leftarrow f(h_{i-1}, x_i) = \text{ReLU}(x_i W_x + h_{i-1} W_h + b_h)$
end if
end for
$y_{\text{pred}} \leftarrow h_{L-1} W_{\text{out}} + b_{\text{out}}$ ▷ Linear layer to produce scores
$loss \leftarrow \text{CrossEntropy}(y_{\text{pred}}, y)$
Backpropagate *loss* to compute gradients $\nabla W_x, \nabla W_h, \nabla b_h, \nabla W_{\text{out}}, \nabla b_{\text{out}}$
Update parameters:
$W_x \leftarrow W_x - \eta \nabla W_x$
$W_h \leftarrow W_h - \eta \nabla W_h$
$b_h \leftarrow b_h - \eta \nabla b_h$
$W_{\text{out}} \leftarrow W_{\text{out}} - \eta \nabla W_{\text{out}}$
$b_{\text{out}} \leftarrow b_{\text{out}} - \eta \nabla b_{\text{out}}$
end for
end for

Algorithm 4 Backpropagation Through Time (BPTT)

Require: Sequence $(x_1, \ldots, x_L)$, label y, parameters θ
Forward compute $h_1, \ldots, h_L$
Compute output loss L
$\delta_L \leftarrow \nabla_{h_L} L$
for $t = L$ down to 1 **do**
$\delta_t \leftarrow (W_h^\top \delta_{t+1}) \odot \phi'(z_t)$
$\nabla W_x \mathrel{+}= \delta_t x_t^\top$
$\nabla W_h \mathrel{+}= \delta_t h_{t-1}^\top$
$\nabla b_h \mathrel{+}= \delta_t$
end for
Update $\theta \leftarrow \theta - \eta \nabla \theta$

3.3 RNN for Word Language Classification

3.4 Thực thi giải thuật bằng Python

```
class LanguageIDModel(Module):
    """
    A model for language identification at a single-word granularity.

    (See RegressionModel for more information about the APIs of different
    methods here. We recommend that you implement the RegressionModel before
    working on this part of the project.)
    """
    def __init__(self):
        # Our dataset contains words from five different languages, and the
        # combined alphabets of the five languages contain a total of 47 unique
        # characters.
        # You can refer to self.num_chars or len(self.languages) in your code
        self.num_chars = 47
        self.languages = ["English", "Spanish", "Finnish", "Dutch", "Polish"]
        super(LanguageIDModel, self).__init__()
        "*** YOUR CODE HERE ***"
        self.hidden_size = 1024
        self.output_size = len(self.languages)

        # RNN parameters
        self.Wx = Linear(self.num_chars, self.hidden_size, bias=True)
        self.Wh = Linear(self.hidden_size, self.hidden_size, bias=False)
        self.Wout = Linear(self.hidden_size, self.output_size)

    def run(self, xs):
        """
        Runs the model for a batch of examples.

        Although words have different lengths, our data processing guarantees
        that within a single batch, all words will be of the same length (L).

        Here `xs` will be a list of length L. Each element of `xs` will be a
        tensor with shape (batch_size x self.num_chars), where every row in the
        array is a one-hot vector encoding of a character. For example, if we
        have a batch of 8 three-letter words where the last word is "cat", then
        xs[1] will be a tensor that contains a 1 at position (7, 0). Here the
        index 7 reflects the fact that "cat" is the last word in the batch, and
        the index 0 reflects the fact that the letter "a" is the inital (0th)
        letter of our combined alphabet for this task.

        Your model should use a Recurrent Neural Network to summarize the list
        `xs` into a single tensor of shape (batch_size x hidden_size), for your
        choice of hidden_size. It should then calculate a tensor of shape
        (batch_size x 5) containing scores, where higher scores correspond to
        greater probability of the word originating from a particular language.
```

```
        Inputs:
            xs: a list with L elements (one per character), where each element
                is a node with shape (batch_size x self.num_chars)
        Returns:
            A node with shape (batch_size x 5) containing predicted scores
                (also called logits)
        """
        "*** YOUR CODE HERE ***"
        batch_size = xs[0].shape[0]
        h = torch.zeros(batch_size, self.hidden_size)

        # Process each character sequentially
        for x_t in xs:
            # Update hidden state: h_t = ReLU(x_t * Wx + h_{t-1} * Wh + b)
            h = relu(self.Wx(x_t) + self.Wh(h))

        # After the last character, use final hidden state to predict language
        out = self.Wout(h)  # shape: batch_size x output_size
        return out

    def get_loss(self, xs, y):
        """
        Computes the loss for a batch of examples.

        The correct labels `y` are represented as a node with shape
        (batch_size x 5). Each row is a one-hot vector encoding the correct
        language.

        Inputs:
            xs: a list with L elements (one per character), where each element
                is a node with shape (batch_size x self.num_chars)
            y: a node with shape (batch_size x 5)
        Returns: a loss node
        """
        "*** YOUR CODE HERE ***"
        logits = self.run(xs)
        return cross_entropy(logits, y)  # PyTorch cross_entropy expects logits, not sof

    def train(self, dataset, batch_size=32, lr=0.001, max_epochs=50):
        """
        Trains the model.

        Note that when you iterate through dataloader, each batch will returned as its c
        (batch_size x length of word x self.num_chars). However, in order to run multipl
        get_loss() and run() expect each batch to be in the form (length of word x batch
        that you need to switch the first two dimensions of every batch. This can be don
        as follows:

        movedim(input_vector, initial_dimension_position, final_dimension_position)
```

```
        For more information, look at the pytorch documentation of torch.movedim()
        """
        "*** YOUR CODE HERE ***"
        dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
        optimizer = optim.Adam(self.parameters(), lr=lr)

        for epoch in range(max_epochs):
            total_loss = 0.0
            for batch in dataloader:
                batch_x = batch['x']       # shape: batch_size x L x num_chars
                batch_y = batch['label']   # shape: batch_size x output_size
                # xs: list of length L, each element shape: batch_size x num_chars
                xs = [movedim(batch_x[:, i, :], 0, 0) for i in range(batch_x.shape[1])]
                y = batch['label']  # shape: batch_size x output_size

                optimizer.zero_grad()
                loss = self.get_loss(xs, y)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()

            print(f"Epoch {epoch+1}/{max_epochs}, Loss: {total_loss:.4f}")



def Convolve(input: tensor, weight: tensor):
    """
    Acts as a convolution layer by applying a 2d convolution with the given inputs and w
    DO NOT import any pytorch methods to directly do this, the convolution must be done
    already imported.

    There are multiple ways to complete this function. One possible solution would be to
    If you would like to index a tensor, you can do it as such:

    tensor[y:y+height, x:x+width]

    This returns a subtensor who's first element is tensor[y,x] and has height 'height,
    """
    input_tensor_dimensions = input.shape
    weight_dimensions = weight.shape
    Output_Tensor = tensor(())
    "*** YOUR CODE HERE ***"
    H_out = H_in - H_k + 1
    W_out = W_in - W_k + 1
    for i in range(H_out):
        for j in range(W_out):
            patch = input[i:i+H_k, j:j+W_k]
            Output_Tensor[i, j] = (patch * weight).sum()

    return Output_Tensor
```