UCSB, Physics 129L, Computational Physics
Lecture notes, Week 10

Zihang Wang (UCSB), zihangwang@ucsb.edu

March 11, 2025

Contents

1 Backpropagation

Backpropagation is the algorithm used to compute gradients of the transport map for training deep networks. It uses the **chain rule** of calculus to propagate errors backward from the output to the input layers. Then, it updates the weights and biases via the gradient descent and push forward the network again. We need to update those weights and biases layer-by-layer backwards, i.e. we need,

$$\frac{\partial C(T)}{\partial \theta_1^{(l)}}, \quad \frac{\partial C(T)}{\partial \theta_0^{(l)}}, \quad \forall\, l \in [0, L], \tag{1}$$

such that,

$$\theta_{1,\mathrm{new}}^{(l)} = \theta_{1,\mathrm{old}}^{(l)} - \alpha_1 \frac{\partial C(T)}{\partial \theta_1^{(l)}}, \quad \theta_{0,\mathrm{new}}^{(l)} = \theta_{0,\mathrm{old}}^{(l)} - \alpha_0 \frac{\partial C(T)}{\partial \theta_0^{(l)}}.$$

Therefore, we must calculate those gradients.

Let's first look at the change in the output layer,

$$\frac{\partial C}{\partial \theta_1^{(L)}} = \frac{\partial C}{\partial z^{(L)}} \otimes \frac{\partial z^{(L)}}{\partial \theta_1^{(L)}} = \frac{\partial C}{\partial z^{(L)}} \left[a^{(L-1)}\right]^T, \tag{2}$$

where in the last line, we use the following,

$$z^{(l)} = \theta_1^{(l)} a^{(l-1)} + \theta_0^{(l)}. \tag{3}$$

We should note that outer product is defined such that,

$$\left[\frac{\partial C}{\partial \theta_1^{(L)}}\right]_{ij} = \frac{\partial C}{\partial z^{(L)}}_i \left[a^{(L-1)}\right]_j, \quad \frac{\partial C}{\partial z^{(L)}} \in \mathbb{R}^{n_L \times 1}, \quad a^{(L-1)} \in \mathbb{R}^{n_{L-1} \times 1}. \tag{4}$$

The transpose ensures the dimensionality.

Similarly, we have,

$$\frac{\partial C}{\partial \theta_0^{(L)}} = \frac{\partial C}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial \theta_0^{(L)}} = \frac{\partial C}{\partial z^{(L)}} = \delta^{(L)}, \tag{5}$$

since the coefficient is a constant 1 (1×1) matrix. The above expression can be extended to hidden layer calculations: We define the error term at layer l as,

$$\delta^{(l)} \equiv \frac{\partial C(T)}{\partial z^{(l)}} = \frac{\partial C(T)}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} = \frac{\partial C(T)}{\partial a^{(l)}} \circ f'\big(z^{(l)}\big), \quad \delta^{(l)} \in \mathbb{R}^{n_l \times 1}$$

where $\circ$ is element-wise product. The error above is **specific with respect to a transport cost and activation**.

For a general hidden layer, we can also propagate backward for the weight $\theta_1^{(l)}$,

$$\frac{\partial C}{\partial \theta_1^{(l)}} = \frac{\partial C}{\partial z^{(l)}} \otimes \frac{\partial z^{(l)}}{\partial \theta_1^{(l)}} = \frac{\partial C}{\partial z^{(l)}} \left[a^{(l-1)}\right]^T, \tag{6}$$

For hidden layers, the error is propagated backward:

$$\frac{\partial C(T)}{\partial a^{(l-1)}} = \frac{\partial C(T)}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial a^{(l-1)}}, \tag{7}$$

and since we know,

$$z^{(l)} = \theta_1^{(l)} a^{(l-1)} + \theta_0^{(l)}, \quad a^{(l)} = f(z^{(l)}), \tag{8}$$

we have,

$$\frac{\partial C(T)}{\partial a^{(l-1)}} = (\theta_1^{(l)})^T \delta^{(l)}, \quad \delta^{(l)} \in \mathbb{R}^{n_l \times 1}, \quad \theta_1^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}. \tag{9}$$

This gives the following recursion formula,

$$\delta^{(l-1)} = \frac{\partial C(T)}{\partial z^{(l-1)}} = \frac{\partial C(T)}{\partial a^{(l-1)}} \frac{\partial a^{(l-1)}}{\partial z^{(l-1)}} = (\theta_1^{(l)})^T \delta^{(l)} \circ f'\big(z^{(l)}\big).$$

We then have gradients with respect to the weights and biases, and can be solved iteratively, as we only need to calculate the gradient at the output layer,

$$\frac{\partial C(T)}{\partial \theta_1^{(l)}} = \delta^{(l)} \big(a^{(l-1)}\big)^T, \quad \frac{\partial C(T)}{\partial \theta_0^{(l)}} = \delta^{(l)}.$$

1.1 Loss function

For **multi-class classification**, the derivative of the cross-entropy loss with respect to the logits z_i is,

$$\frac{\partial C(T)}{\partial z_i} = \hat{y}_i - y_i, \quad \frac{\partial C(T)}{\partial z_i} \in \mathbb{R}^{D\times 1}, \tag{10}$$

where D is the dimension of the output layer, and the expression results from the softmax function. In general, we need additional batch size dimension G. Instead, it becomes $\frac{\partial C(T)}{\partial z} \in \mathbb{R}^{D\times G}$.

For **regression** problems, the mean squared error (MSE) loss, using the sigmoid function:

$$\frac{\partial C(T)}{\partial z} = (\hat{y} - y) \cdot \hat{y} \cdot (1 - \hat{y}), \quad \frac{\partial C(T)}{\partial z} \in \mathbb{R}, \tag{11}$$

and we should note that we are working with the unit batch size. In general, we need additional batch size dimension G. Instead, it becomes $\frac{\partial C(T)}{\partial z} \in \mathbb{R}^{1\times G}$.

To optimize the network, we use **gradient descent** methods. The parameters (weights and biases) are updated as follows,

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \cdot \nabla_\theta C,$$

where θ represents the parameters, α is the **learning rate**, and $\nabla_\theta C$ is the gradient of the loss with respect to θ. In practice, this gradient is used in **stochastic gradient descent (SGD)** to update model weights efficiently. As we discussed previously, SGD only samples the gradients at different points from the previous neurons (since it is the input of the current neuron), instead of using gradients of all at once.

1.2 Concept in convolutional neural network

We first have the **Input Layer**, which takes an input image of shape (H, W, C), where H is the height, W is the width, and C is the number of color channels (RGB).

The input is then processed by the **convolutional layer**, where a set of K **filters** (kernels) of size (k, k, C) slides over the input with **step size** s to produce K feature maps of size (H', W', K). Sometimes, we do not want to keep the image dimension without changing the kernel. We can add **padding** P. The relation between dimensions becomes,

$$K' = \frac{K - k + 2p}{s}, \quad H' = \frac{H - k + 2p}{s}. \tag{12}$$

The reduced input after the convolution becomes,

$$z_{i,j,k}^{(1)} = \sum_{m=1}^{C}\sum_{p=1}^{k}\sum_{q=1}^{k}\theta_{p,q,m,k}^{(1)} x_{i+p,j+q,m} + \theta_0^{(1)}, \quad \theta^{(1)} \in \mathbb{R}^{k\times k\times C\times K}, \quad \theta_0^{(1)} \in \mathbb{R}^K \tag{13}$$

$$a^{(1)} = f\big(z^{(1)}\big), \quad a^{(1)} \in \mathbb{R}^{H'\times W'\times K}. \tag{14}$$

We then have a **pooling layer** (e.g., max-pooling or average) with filter size (p, p) and stride s, reducing the spatial dimensions to (H'', W'', K):

$$a^{(2)} = \text{Pooling}(a^{(1)}), \quad a^{(2)} \in \mathbb{R}^{H''\times W''\times K}. \tag{15}$$

After multiple convolutional and pooling layers, **we flatten the output into a vector of size** $(d = H''W''K)$ and pass it to a **fully connected layer** (e.g. MLP), mapping it to an r-dimensional hidden layer:

$$z^{(3)} = \theta_1^{(3)}\text{flatten}(a^{(2)}) + \theta_0^{(3)}, \quad \theta_1^{(3)} \in \mathbb{R}^{r\times d}, \quad \theta_0^{(3)} \in \mathbb{R}^r \tag{16}$$

$$a^{(3)} = f\big(z^{(3)}\big), \quad a^{(3)} \in \mathbb{R}^r. \tag{17}$$

Finally, we have the **output layer**, mapping the r-dimensional hidden representation to K class scores:

$$z^{(4)} = \theta_1^{(4)}a^{(3)} + \theta_0^{(4)}, \quad \theta_1^{(4)} \in \mathbb{R}^{K\times r}, \quad \theta_0^{(4)} \in \mathbb{R}^K \tag{18}$$

$$\hat{y} = \text{softmax}\big(z^{(4)}\big), \quad \hat{y} \in \mathbb{R}^K. \tag{19}$$

where $\hat{y}$ represents the predicted class probabilities for K classes. You should note that in the output layer, we need softmax for classification.

1.3 Apllication with CNN: MNIST Dataset

MNIST (Modified National Institute of Standards and Technology) is a widely used dataset in machine learning and computer vision. It consists of 70,000 grayscale images of handwritten digits (0-9), each of size 28x28 pixels. The dataset is commonly used for training and testing image classification models, particularly for deep learning applications like convolutional neural networks (CNNs). It is a benchmark dataset for evaluating new algorithms and techniques in pattern recognition. In this week's section, you are asked to train a multilayer perceptron network and a convolutional neural network that solve this classification problem.

2 Recurrent Neural Networks

Feedforward Neural Networks (such as MLP and CNN we discussed previously) process data in one direction from input to output without retaining information from previous inputs. This makes them suitable for tasks with independent inputs like image classification. However it struggle with sequential data since all information are passing to the network at once.

If we want to interpret a time series where the sequential arrangements are critical in drawing conclusions, we must modify the neural network structure.

An recurrent neural network processes sequential data (such as text, audio, and video) by keeping a hidden state and passing it to a future step.

Consider an input sequence:

$$X = \{x^{(1)}, x^{(2)}, \ldots, x^{(T)}\}.$$

At each time step t, the **RNN cell** updates its hidden state $h^{(t)}$ and produces an output $y^{(t)}$. The recurrence is defined by two equations: the first equation blends past information (h_{t-1}) with the new input (x_t); The second equation activates the hidden layer that produces the current RNN output (but it is optional),

$$h^{(t)} = a_h\left(z_{xh}^{(t)}\right), \quad z_{xh}^{(t)} = \theta_{1x}\, x^{(t)} + \theta_{1h}\, h^{(t-1)} + \theta_0, \tag{20}$$

$$y^{(t)} = a_y\left(z_y^{(t)}\right), \quad z_y^{(t)} = \theta_{1y}\, h^{(t)} + \theta_{0y}, \tag{21}$$

where,

- θ_{1h} = Weight matrix for the previous hidden state.
- θ_{1x} = Weight matrix for the current input.
- θ_0 and θ_{0y} are bias vectors.
- $a_h = \tanh$ = Activation function ranging from $[-1, 1]$. This range ensures that information carried by the hidden state $h^{(t)}$ can be add/removed in the next layer, where information does not continuous accumulate in an uncontrolled way (centered at zero).
- θ_{1y} is the weight matrix for hidden-to-output,
- $a_y = \sigma(\cdot)$ is the sigmoid function. For classification tasks, y_t might pass through a **softmax** function to produce probabilities.

Graphically, each RNN cell takes the previous RNN cell's hidden state $h^{(t-1)}$, and a new input $x^{(t)}$. It outputs $y^{(t)}$ by activating $h^{(t)}$. Then, $h^{(t)}$ is passing to the next time step.

We should note a very important property: When an RNN is unrolled over time, each time step is conceptually like a "layer" in a deep network, but **all these layers share the same weights and biases**. This parameter sharing is fundamental to RNNs because **it allows the network to generalize across different time steps** and significantly reduces the number of parameters that need to be learned.

2.1 Backpropagation Through Time (BPTT)

We define the total transport cost for the sequence by summing all transport costs,

$$C = \sum_{t=1}^{T} C^{(t)},$$

where $C^{(t)}$ is the loss at time step t (e.g., cross-entropy loss for classification tasks). The goal is to compute the gradients of the total loss C with respect to the network parameters,

For the output weight matrix W_{hy} and bias b_y, the gradients are computed as,

$$\frac{\partial L}{\partial \theta_{1y}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial \theta_{1y}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial a_y} \frac{\partial a_y}{\partial z_y^{(t)}} \frac{\partial z_y^{(t)}}{\partial \theta_{1y}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial z_y^{(t)}} \left(h^{(t)}\right)^T = \sum_{t=1}^{T} \delta_y^{(t)} \left(h^{(t)}\right)^T, \tag{22}$$

$$\frac{\partial L}{\partial \theta_{0y}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial z_y^{(t)}} = \sum_{t=1}^{T} \delta_y^{(t)}, \tag{23}$$

and we can see that this is basically the same as the backpropagation $a \to h$ we discussed in the MLP previously. The only difference is that it contains a sum over contributions from all times.

The important difference between BPTT and backpropagation in MLP is to compute the gradients with respect to the hidden states $h^{(t)}$ by applying the chain rule over time. Let's consider the following error term,

$$\delta_{xh}^{(t)} = \frac{\partial L^{(t)}}{\partial z_{xh}^{(t)}} = D_h^{(t)} \frac{\partial h^{(t)}}{\partial z_{xh}^{(t)}}, \quad D_h^{(t)} \equiv \frac{\partial L}{\partial h^{(t)}},$$

where it is a **derivative** on the total transport cost with respect to the hidden state. **We should note that it is not a derivative with respect to the time independent variable.** At each time step t, $D_h^{(t)}$ receives contributions from two sources:

- The direct effect on the loss at time t.
- The indirect effect through hidden states at future times.

The recursion becomes:

$$D_h^{(t)} = \frac{\partial L}{\partial h^{(t)}} = \frac{\partial L^{(t)}}{\partial h^{(t)}} + \frac{\partial L}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(t)}} = \frac{\partial L^{(t)}}{\partial h^{(t)}} + D_h^{(t+1)} \frac{\partial h^{(t+1)}}{\partial h^{(t)}}. \tag{24}$$

Since we have the expression,

$$h^{(t+1)} = a_h(z_{xh}^{(t+1)}) = a_h\Big(\theta_{1x}\, x^{(t+1)} + \theta_{1h}\, h^{(t)} + \theta_0\Big),$$

and its derivative with respect to $h^{(t)}$ is:

$$\frac{\partial h^{(t+1)}}{\partial h^{(t)}} = \frac{\partial h^{(t+1)}}{\partial z_{xh}^{(t+1)}} \frac{\partial z_{xh}^{(t+1)}}{\partial h^{(t)}} = \theta_{1h}^T \circ \sigma'\big(z_{xh}^{(t)}\big),$$

where $\circ$ is element-wise multiplication. Using the above expression, we have,

$$D_h^{(t)} = \frac{\partial L^{(t)}}{\partial h^{(t)}} + \theta_{1h}^T D_h^{(t+1)} \circ \sigma'\big(z_{xh}^{(t)}\big). \tag{25}$$

With this expression, we can calculate the gradient,

$$\frac{\partial L}{\partial \theta_{1h}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial z_{xh}^{(t)}} \frac{\partial z_{xh}^{(t)}}{\partial \theta_{1h}} = \sum_{t=1}^{T} \delta_{xh}^{(t)} (h^{(t)})^T. \tag{26}$$

We can see that all back propagations follow identical structures as those in MLP, but the detail calculations are different.

Similarly, we have,

$$\frac{\partial L}{\partial \theta_{1x}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial z_{xh}^{(t)}} \frac{\partial z_{xh}^{(t)}}{\partial \theta_{1x}} = \sum_{t=1}^{T} \delta_{xh}^{(t)} (x^{(t)})^T. \tag{27}$$

$$\frac{\partial L}{\partial \theta_{1x}} = \sum_{t=1}^{T} \frac{\partial L^{(t)}}{\partial z_{xh}^{(t)}} \frac{\partial z_{xh}^{(t)}}{\partial \theta_{0}} = \sum_{t=1}^{T} \delta_{xh}^{(t)}. \tag{28}$$

We can see that the gradient at a given time step depends on all future time steps. This accumulation can lead to numerical problems such as **vanishing gradients** (where gradients shrink exponentially), particularly when T is large. This is because that the recursion problem involving products that leads to exponential decay or growth as the depth T increases.

Long Short-Term Memory (LSTM) networks are a variant of RNN designed to mitigate the vanishing gradient problem. Unlike standard RNNs, LSTMs introduce a **cell state** $C^{(t)}$ that allows gradients to flow more effectively over long time steps.

In an LSTM, we define the following gates and states:

$$a_{\text{in}}^{(t)} = \sigma(z_{1,xh}^{(t)}) \quad \text{(Input Gate)} \tag{29}$$

$$a_{\text{forget}}^{(t)} = \sigma(z_{2,xh}^{(t)}) \quad \text{(Forget Gate)} \tag{30}$$

$$a_{\text{out}}^{(t)} = \sigma(z_{3,xh}^{(t)}) \quad \text{(Output Gate)} \tag{31}$$

$$\tilde{C}^{(t)} = \tanh(\theta_{1x}^C x^{(t)} + \theta_{1h}^C h^{(t-1)} + \theta_0^C) \quad \text{(Candidate Cell updates)} \tag{32}$$

$$C^{(t)} = a_{\text{forget}}^{(t)} \circ C^{(t-1)} + a_{\text{in}}^{(t)} \circ \tilde{C}^{(t)} \quad \text{(Cell State Update)} \tag{33}$$

$$h^{(t)} = a_{\text{out}}^{(t)} \circ \tanh(C^{(t)}) \quad \text{(Hidden State)} \tag{34}$$

3 Gradient Flow in LSTMs

To understand how LSTMs avoid the vanishing gradient problem, we compute the gradient of the loss with respect to the cell state:

$$\frac{\partial L}{\partial C^{(t)}} = \frac{\partial L}{\partial C^{(t+1)}} \frac{\partial C^{(t+1)}}{\partial C^{(t)}}. \tag{35}$$

From the cell state update equation, we differentiate:

$$\frac{\partial C^{(t+1)}}{\partial C^{(t)}} = a^{(t)}_{\text{forget}}. \tag{36}$$

Expanding over multiple time steps, the gradient flow is:

$$\frac{\partial L}{\partial C^{(t)}} = \left(\prod_{k=t+1}^{T} a^{(t)}_{\text{forget}} \right) \frac{\partial L}{\partial C^{(T)}}. \tag{37}$$

the model learns to keep $a^{(t)}_{\text{forget}}$ high when it is important for the task, ensuring that the memory (and thus the gradients) is preserved across many time steps. This is very different from the simple RNN: RNN cells only connect via the hidden state $h^{(t)}$.