UCSB, Physics 129L, Computational Physics Lecture notes, Week 9

Zihang Wang (UCSB), zihangwang@ucsb.edu

March 9, 2025

Contents

1 Multilayer neural network, optimal transport, and deep learning

Deep Learning is a subfield of machine learning that employs neural networks to model complex patterns in data. Unlike traditional machine learning approaches, deep learning models automatically learn hierarchical representations from raw data. In other words, we do not need to impose any pre-known models and assumptions.

When data is **labeled**, we call it **supervised deep learning**. $\{X, y\}$ Both X and y can be scalar or vector. On the other hand, if the data is **unlabeled**, X, it is **unsupervised deep learning**.

In supervised deep learning, the task is to learn the relationship between y and X, such the prediction $\hat{y}$ from the model $\hat{y} = T(x)$ is as close as possible to true value. T is analogous to the transport map (plan) that represents large amount of hyperparameters that give the deep learning enough of degree's of freedom. We should note that the x and $\hat{y}$ does not need to have the same dimension. This is similar to the **Kantorovich problem**. Number of parameters in state-of-the-art neural networks: **GPT-3: 175 billion GPT-4: 1.8 trillion**.

The performance of a neural network is measured by a **loss function** E. The loss function defined in the neural network shares the same idea as we have in the **optimal transport**: **we want to find the optimal way of designing a transport map T^* that minimizes the cost of transporting a source distribution μ (model output) to a target distribution ν (training set that contains true labels).**

The question becomes, how to find the optimal transport plan T^* such that the predicted $\hat{y} = T(x)$ can match the true value y: **The best map T^* is the ones that give the smallest prediction error on training data**.

In the perspective of optimal transport, the map is defined by a scalar potential gradient $\nabla\varphi = T$ that minimizes the **transport cost**,

$$C(T) = \int_X c[x, T(\mathbf{x})] d\mu(\mathbf{x}) = \int_X c[\mathbf{x}, T(\mathbf{x})]\mu(\mathbf{x}) d\mathbf{x},$$

where in supervised learning, it is defined as the **loss function** quantifies the discrepancy between predicted values and actual values. The choice of loss function depends on the type of learning problem: **regression** and **classification**.

1.1 Regression Problem: maximum likelihood estimation

In regression tasks, the target variable y is continuous, meaning $y \in \mathbb{R}$. The most commonly used loss functions are: **Mean Squared Error** (MSE),

$$C(T) = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2,$$

where $\hat{y}_i = f_\theta(X_i)$ is the predicted value based on the transport plan T. MSE penalizes larger errors more heavily due to the squared term. There is also the **Mean Absolute Error** (MAE),

$$C(T) = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|,$$

MAE gives equal weight to all errors, making it more robust to outliers compared to MSE.

Minimizing MSE corresponds to maximizing the likelihood under the assumption that errors are **normally distributed** (refer to lecture notes on stochastic calculus and log likelihood):

$$y_i = T(X_i) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

Recall in previous lecture notes, the MSE objective function is derived from the **negative log-likelihood** (χ-square) of the Gaussian distribution (with

Frequentist statistics or Bayesian inference with uniform prior),

$$C(T) = -\sum_{i=1}^{n} \log P(y_i|X_i, \theta) \sim \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.$$

This shows that minimizing MSE is equivalent to performing maximum likelihood estimation (MLE) under a Gaussian noise assumption.

1.2 Classification Problem: Cross entropy

In **classification problems**, the target variable y is discrete and belongs to a finite set of classes. For a classification problem with K classes, the model assigns a raw score (**logit**) z_k to each class k. For a given input X_i, the logits are computed as,

$$z_k = w_k^T x_i + b_k,$$

where w_k is the weight for class k, b_k is the bias for class k, x_i is the input feature vector for sample i. The logits z_k do not represent probabilities. Instead, **they are arbitrary scores that can be positive or negative**. To convert the logits into probabilities, we apply the **softmax function**:

$$p_T(y_i = k|X_i) = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)}$$

This ensures the probability $p_T(y_i = k|X_i)$ is always in the range $(0, 1)$, and the probabilities sum to 1 across all classes. This is very similar to the Boltzmann weights we discussed previously.

Given a dataset with n samples, the cross-entropy loss for multi-class classification is:

$$C(T) = -\sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}(y_i = k) \log p_T(y_i = k|X_i),$$

$\mathbb{1}(y_i = k)$ is an indicator function that equals 1 if the true class for X_i is k, otherwise 0 (**one-hot encoded**). - $p_T(y_i = k|X_i)$ is the predicted probability for class k. Since y_i is the true value, it only selects a single value out of this sum by the indicator function $\mathbb{1}(y_i = k)$.

Consider a classification problem with $K = 3$ classes, where $y \in \{0, 1, 2\}$. Suppose for an input X_i, the model predicts the logits:

$$z_0 = 1.5, \quad z_1 = 2.0, \quad z_2 = 0.5.$$

Applying the **softmax function**:

$$p_\theta(y_i = 0|X_i) = \frac{e^{1.5}}{e^{1.5} + e^{2.0} + e^{0.5}} \approx 0.29$$

$$p_\theta(y_i = 1|X_i) = \frac{e^{2.0}}{e^{1.5} + e^{2.0} + e^{0.5}} \approx 0.49$$

$$p_\theta(y_i = 2|X_i) = \frac{e^{0.5}}{e^{1.5} + e^{2.0} + e^{0.5}} \approx 0.22$$

If the true label is $y_i = 1$, then the loss for this sample is:

$$L_1 = -\log p_\theta(y_i = 1|X_i) = -\log(0.49) \approx 0.71$$

If the true label was $y_i = 2$, the loss would be:

$$L_2 = -\log p_\theta(y_i = 2|X_i) = -\log(0.22) \approx 1.51$$

The cross-entropy loss directly corresponds to the negative log-likelihood function in maximum likelihood estimation,

$$C(T) = -\sum_{i=1}^{n} L_i.$$

Thus, **minimizing cross-entropy loss is equivalent to maximizing the likelihood of the observed data.**

In **unsupervised deep learning**, how well the model captures or reconstructs the inherent structure of the input data, rather than comparing its output to an externally provided label.

1.3 Transport cost function (loss function)

To update the weights in the network, we compute the gradient of the loss with respect to the network outputs.

Cross entropy measures the dissimilarity between two probability distributions. In **classification**, we often have a one-hot encoded target distribution (delta function) and a predicted probability distribution. **In regression, however, you're predicting continuous values rather than a distribution**.

Regression models typically output real numbers without the normalization constraint (i.e., they don't sum to 1 like probabilities). **Using cross entropy would require interpreting these outputs as probabilities, which usually doesn't align with the nature of regression tasks.**

In binary classification, even though the problem is about predicting a class (0 or 1), the model typically outputs a continuous value that represents a probability after applying a sigmoid function.

In binary classification, we use a single output neuron with a sigmoid activation function, which converts the logit z into a probability:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

Since there are only two classes (0 and 1), one probability $\hat{y}$ is enough because:

$$P(y=1|X)=\hat{y}, \quad P(y=0|X)=1-\hat{y}$$

The two probabilities are complementary and always sum to 1. However, **in multi class case, we do not have the complementary property.**

Therefore, if we want to interpret the output as probabilities, we need to use cross entropy method. On the other hand, if we look for prediction in values, we need to use mean-square.

1.4 Linear Regression as a Single-Layer Supervised Learning Algorithm

Linear regression can be viewed as a simple supervised learning algorithm with one layer,

$$\hat{y}_i = T(X_i) = \theta_0 + \theta_1 X_i,$$

where $\theta = (\theta_0, \theta_1)$ are the model parameters that generate transport map from X_i to y_i. These parameters can be obtained via the stationary condition of the negative log-likelihood: It minimizes the total sum of squared residuals (χ square),

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

This corresponds to minimizing the MSE loss.

For optimization, the gradient of the MSE loss function with respect to θ is,

$$\frac{\partial C_\theta}{\partial \theta_1} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)X_i, \quad \frac{\partial C_\theta}{\partial \theta_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i).$$

which leads to updates in gradient descent:

$$\theta^{(l)}_{1,\text{new}} = \theta^{(l)}_{1,\text{old}} - \alpha_1 \frac{\partial C(T)}{\partial \theta^{(l)}_1}, \quad \theta^{(l)}_{0,\text{new}} = \theta^{(l)}_{0,\text{old}} - \alpha_0 \frac{\partial C(T)}{\partial \theta^{(l)}_0}$$

where α is the learning rate.

1.5 Layers in neural network

A **neural network** comprises layers of interconnected nodes (neurons) that are usually fully-connected.

The main components include:

- **Input Layer:** Receives the raw data, denoted as $\mathbf{x} \in \mathbb{R}^n$.
- **Hidden Layers:** Intermediate layers that perform non-linear transformations. A network can have one or multiple hidden layers.
- **Output Layer:** Produces the final output, such as a class label or a regression value.

The term **deep** in deep learning refers to **the number of layers (or depth)** in the neural network. A neural network is considered deep when it has multiple hidden layers (beyond just an input and output layer). The **width of a neural network** refers to the **number of neurons per layer**.

Each neuron receives an n dimension data $\mathbf{z_1}$ produced by the previous layer, and preforms a linear transformation with **weight matrix** θ_1^l and **bias** θ_0^l, that acts on previous layer data $z^{(l-1)}$. **In this case, we set the batch number to be 1.**

$$z^{(l)} = \theta_1^{(l)} a^{(l-1)} + \theta_0^{(l)}, \quad \theta_1^{(1)} \in \mathbb{R}^{m\times n}, \quad \theta_0^{(1)} \in \mathbb{R}^m \tag{1}$$

Since joint linear transformations can be written effectively as a single linear equation, we must introduce some non-linearity. To do that, we introduce the **activation function**, $f()$, that acts on the current layer output,

$$a^{(l)} = f\big(z^{(l)}\big). \tag{2}$$

Common activation functions $f()$ include:

- **Sigmoid:** $\sigma(z) = \dfrac{1}{1+e^{-z}}$
- **ReLU:** $\mathrm{ReLU}(z) = \max(0, z)$
- **Tanh:** $\tanh(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$

1.6 Multilayer Perceptron (MLP)

Multilayer Perceptron (MLP) is a simple example of the **feedforward network** with one hidden layer (no loop). Let the input vector be $\mathbf{x} \in \mathbb{R}^n$, the hidden layer contain m neurons, and the output be a scalar y. **We say they are fully connected: all neurons are connected to the neurons in previous and forward layers.**

The forward propagation is described as follows: We first have the **input layer** that takes n inputs and processed by n neurons. therefore, it outputs m dimensional vector,

$$z^{(1)} = \theta_1^{(1)} \mathbf{x} + \theta_0^{(1)}, \quad \theta_1^{(1)} \in \mathbb{R}^{m\times n}, \quad \theta_0^{(1)} \in \mathbb{R}^m \tag{3}$$

$$a^{(1)} = f\big(z^{(1)}\big). \tag{4}$$

We then pass the vector to the activation function. The result will be pass on to the **hidden layer**. The input of the hidden layer becomes m dimensional and the output becomes r dimensional,

$$z^{(2)} = \theta_1^{(2)} a^{(1)} + \theta_0^{(2)}, \quad \theta_1^{(2)} \in \mathbb{R}^{r\times m}, \quad \theta_0^{(2)} \in \mathbb{R}^r \tag{5}$$

$$a^{(2)} = f\big(z^{(2)}\big). \tag{6}$$

Then, we have the **output layer:**

$$z^{(3)} = \theta_1^{(3)} a^{(2)} + \theta_0^{(3)}, \quad \theta_1^{(3)} \in \mathbb{R}^{1\times r}, \quad \theta_0^{(3)} \in \mathbb{R} \tag{7}$$

$$\hat{y} = a^{(3)} = \tilde{f}\big(z^{(3)}\big). \tag{8}$$

In this case, $\tilde{f}$ is usually the softmax for classification problem and activation function if in regression problem.

2 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,

$$\frac{\partial C(T)}{\partial \theta_1^{(l)}}, \quad \frac{\partial C(T)}{\partial \theta_0^{(l)}}, \quad \forall\, l \in [0, L], \tag{9}$$

such that,

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

Therefore, we must calculate those gradients.

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

$$\frac{\partial C}{\partial \theta_1^{(L)}} = \frac{\partial C}{\partial z^{(L)}} \otimes \frac{\partial z^{(L)}}{\partial \theta_1^{(L)}} = \frac{\partial C}{\partial z^{(L)}} \left[a^{(L-1)}\right]^T, \tag{10}$$

where in the last line, we use the following,

$$z^{(l)} = \theta_1^{(l)} a^{(l-1)} + \theta_0^{(l)}. \tag{11}$$

We should note that outer product is defined such that,

$$\left[\frac{\partial C}{\partial \theta_1^{(L)}}\right]_{ij} = \frac{\partial C}{\partial z^{(L)}}_i \left[a^{(L-1)}\right]_j, \quad \frac{\partial C}{\partial z^{(L)}} \in \mathbb{R}^{n_L \times 1}, \quad a^{(L-1)} \in \mathbb{R}^{n_{L-1}\times 1}. \tag{12}$$

The transpose ensures the dimensionality.

Similarly, we have,

$$\frac{\partial C}{\partial \theta_0^{(L)}} = \frac{\partial C}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial \theta_0^{(L)}} = \frac{\partial C}{\partial z^{(L)}}, \tag{13}$$

since the coefficient is a constant 1 (1×1) matrix.

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 hidden layers, the error is propagated backward:

$$\frac{\partial C(T)}{\partial a^{(l-1)}} = \frac{\partial C(T)}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial a^{(l-1)}}, \tag{14}$$

and since we know,

$$z^{(l)} = \theta_1^{(l)} a^{(l-1)} + \theta_0^{(l)}. \tag{15}$$

we have,

$$\frac{\partial C(T)}{\partial a^{(l-1)}} = (\theta_1^{(l)})^T \delta^{(l)}, \quad \delta^{(l)} \in \mathbb{R}^{n_l \times 1}, \quad \theta_1^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}. \tag{16}$$

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

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,

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

2.1 Loss function

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

$$\frac{\partial C(T)}{\partial z_i} = \hat{y}_i - y_i, \quad \frac{\partial C(T)}{\partial z_i} \in \mathbb{R}^{D \times 1}, \tag{17}$$

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:

$$\frac{\partial C(T)}{\partial z} = (\hat{y} - y) \cdot \hat{y} \cdot (1 - \hat{y}), \quad \frac{\partial C(T)}{\partial z} \in \mathbb{R}, \tag{18}$$

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.

2.2 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,

$$K' = \frac{K - k + 2p}{s}, \quad H' = \frac{H - k + 2p}{s}. \tag{19}$$

The reduced input after the convolution becomes,

$$z^{(1)}_{i,j,k} = \sum_{m=1}^{C}\sum_{p=1}^{k}\sum_{q=1}^{k} \theta^{(1)}_{p,q,m,k} x_{i+p,j+q,m} + \theta^{(1)}_0, \quad \theta^{(1)} \in \mathbb{R}^{k\times k\times C\times K}, \quad \theta^{(1)}_0 \in \mathbb{R}^K \tag{20}$$

$$a^{(1)} = f\big(z^{(1)}\big), \quad a^{(1)} \in \mathbb{R}^{H'\times W'\times K}. \tag{21}$$

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):

$$a^{(2)} = \text{Pooling}(a^{(1)}), \quad a^{(2)} \in \mathbb{R}^{H''\times W''\times K}. \tag{22}$$

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:

$$z^{(3)} = \theta^{(3)}_1 \text{flatten}(a^{(2)}) + \theta^{(3)}_0, \quad \theta^{(3)}_1 \in \mathbb{R}^{r\times d}, \quad \theta^{(3)}_0 \in \mathbb{R}^r \tag{23}$$

$$a^{(3)} = f\big(z^{(3)}\big), \quad a^{(3)} \in \mathbb{R}^r. \tag{24}$$

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

$$z^{(4)} = \theta^{(4)}_1 a^{(3)} + \theta^{(4)}_0, \quad \theta^{(4)}_1 \in \mathbb{R}^{K\times r}, \quad \theta^{(4)}_0 \in \mathbb{R}^K \tag{25}$$

$$\hat{y} = \text{softmax}\big(z^{(4)}\big), \quad \hat{y} \in \mathbb{R}^K. \tag{26}$$

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

2.3 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.