Notes on MIT Introduction to Deep Learning

Tianzong Cheng

February 9, 2024

Preface

Many thanks to Professor Ban for introducing this lecture to me.

Tianzong Cheng

February 9, 2024

Contents

1 Intro to Deep Learning

1.1 Perceptron

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} x_i w_i)$$

x_i is the input, w_i is the weight of each input, w_0 is the bias term, and g is a non-linear activation function.

Or we can rewrite the equation in linear algebra language.

$$\hat{y} = g(w_0 + X^T W)$$

where: $X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

An example of the activation function: sigmoid function. It can be interpreted as a continuous version of the threshold function.

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

1.2 Building Neural Networks with Perceptrons

Perceptron: dot product, bias, non-linearity.

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers.

The first layer, which is a hidden layer can be expressed as follows:

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}$$

Then, the second layer, which calculates the final outputs, can be expressed as follows:

$$\hat{y}_i = g(w_{0,i}^{(2)} + \sum_{j=1}^{m} g(z_j) w_{j,i}^{(2)})$$

By stacking these layers on top of each other, we create a sequential model.

1.3 Loss Functions and Gradient Descent

The **empirical loss** measures the total loss over our entire dataset. Empirical loss is also known as objective function, cost function and empirical risk.

$$J(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; \boldsymbol{W}), y^{(i)})$$

In this equation, J is a function of the weights of the neuron network. **Cross entropy loss** can be used with models that output a probability between 0 and 1. **Mean squared error loss** can be used with regression models that output continuous real numbers.

Next, we want to find the network weights that **achieve the lowest loss**.

First, we randomly pick an initial (w_0, w_1). Then, by taking a small step in the opposite direction of gradient at this point, we can get closer to where the loss is the lowest. Repeat this step until convergence.

1.4 Backpropagation

Backpropagation is the process of computing gradients with respect to the weights and biases. The process is given the name because gradients are calculated layer by layer, starting from the output layer.

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

Repeat this for **every weight in the network** using gradients from later layers. We can see backpropagation is simply an instantiation of the chain rule.

1.5 Optimization

$$W \leftarrow W - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

The equation shows how the weights are optimized through gradient descent. Note that *eta* indicates the learning rate. A small learning rate converges slowly and gets stuck in false local minima while large learning rates overshoot, become unstable and diverge. So we need some kind of algorithms that adapts to the landscape.

1.6 Batched Gradient Descent

Compute the gradient of a batch of (usually around a hundred) data points, rather than a single data point or all the data points. This is a fast and accurate way of estimating the gradient.

1.7 Regularization

The problem of overfitting describes that the model is too complex and does not generalize well.

Regularization is a technique that constrains our optimization problem to discourage complex models. It helps to improve generalization of our model on unseen data.

The idea of **dropout** is randomly selecting some (typically half) activations to 0.

The idea of **early stopping** is to stop training before we have a chance to overfit. To be more specific, as the number of iterations increase, the loss of training dataset gets smaller. However, the loss of testing dataset gets smaller first and starts to increase again at certain point. The training process should stop before this happens.

2 Deep Sequence Modeling

2.1 Neurons with Recurrence

$$\hat{y}_t = f(x_t, h_{t-1})$$

The output is a function of the input and **a past memory**, represented by the symbol h. This is the intuitive foundation behind **Recurrent Neural Networks** (RNNs).

$$h_t = f_W(x_t, h_{t-1})$$

The cell state is a function of the input and the old state. Note that the same function and set of parameters are used at every time step.

2.2 Encoding Language for a Neural Network

- Indexing: each word is given a number index
- Embedding: Index to fixed-sized vector
  - One-hot embedding: $[0, \cdots, 0, 1, 0, \cdots, 0]$

  - Learned embedding: Words that have close meanings are put near to each other

2.3 Dealing with Gradient Issues

Computing the gradient with respect to h_0 involves many factors of W_{hh} and repeated gradient computation. Consider the W_{hh} matrix:

- Many values > 1: exploding gradients. Solution: Gradient clipping
- Many values < 1: vanishing gradients.

Solving the vanishing gradient problem:

- Activation Functions: Using ReLU prevents f' from shrinking the gradients when $x > 0$ because the derivative of ReLU function is 1.
- Parameter Initialization: Initialize weights to identity matrix and initialize biases to zero.
- Gated Cells, Long Short Term Memory (LSTMs): Use gates to selectively add or remove information within each recurrent unit. This is **the most robust method** to solve the vanishing gradient problem.

2.4 Long Short Term Memory (LSTMs)

- Maintain a cell state
- Use gates to control the flow of information
  - **Forget** gate gets rid of irrelevant information
  - **Store** relevant information from current input

  - Selectively **update** cell state
  - **Output** gate returns a filtered version of the cell state
- Backpropagation through time with partially uninterrupted gradient flow

There are three gates, which are all functions of the input and the last output, controlling the information flow.

$$\text{gate} = \text{sigmoid}(W[x_t, h_{t-1}] + b)$$

- The old cell state goes through **the forget gate** to get rid of the irrelevant information.
- The input goes through **the input gate**, getting rid of the irrelevant information, and is added upon the filtered old cell state to obtain the current cell state.
- The current cell state goes through **the output gate** to get the output at current time.

2.5 Limitations of RNNs

Limitations of RNNs:

- Encoding bottleneck
- Slow, no parallelization
- Not long memory

Idea 1: Feed everything into dense network:

- Not scalable
- No order
- No long memory

Idea 2: Identify and attend to what's important

2.6 Attention

- Encode position information
- Extract **query**, **key**, **value** for search: we use neural network layers to extract **query**, **key**, **value**.
- Compute attention weighting: $\text{softmax}(\frac{Q \cdot K^T}{\text{scaling}})$, the softmax function constrains the values to be between 0 and 1
- Extract features with high attention

$$A(Q, K, V) = \text{softmax}(\frac{Q \cdot K^T}{\text{scaling}}) \cdot V$$

3 Deep Computer Vision

3.1 Learning Visual Features

If we use a fully connected neural network to deal with images, there are two main problems. Firstly, since the image is flattened down into one dimension, all the spatial information is lost. Secondly, there are too many parameters.

The idea of convolution:

- Connect patch in input layer to a single neuron in subsequent layer.
- Use a **sliding window** to define connections.

The convolution operation is element-wise multiplication between the input image and the filter and the result is a feature map.

3.2 Convolutional Neural Networks

- Convolution: Apply filters to generate feature maps.
- Non-linearity: Often ReLU.
- Pooling: Down-sampling operation on each feature map.

The idea of output volume introduces the dimension of depth, representing the number of filters. Thus, the network can learn a number of different features.

3.2.1 Non-linearity

ReLU function replaces all negative values with zero. The computational expense is really low while introducing non-linearity.

3.2.2 Pooling

Pooling is a way of down-sampling the feature map so that the CNN can deal with bigger images. Max pooling with 2×2 filters and stride 2 is taking the maximum value in each 2×2 block thus reducing the scale by 2 in each dimension. Note that the space information is preserved.

3.3 Applications

3.3.1 Basic Architecture

Use CNNs to learn features in the input image first and then feed the result into downstream networks.

3.3.2 Classification

After the features are learned, the arrays are flattened and fed into a fully connected network. Remember to use the softmax function to normalize the result between 0 and 1, representing the probability.

3.3.3 Object Detection

Faster R-CNN learns region proposals. A region proposal network learns candidate regions and feed them into downstream networks.

3.3.4 Semantic Segmentation

Semantic segmentation is a fully convolutional network that classify every pixel of the input image. The network is designed with both down-sampling and up-sampling.

4 Deep Reinforcement Learning

There are three classes of learning problems:

- Supervised Learning: learn a function to map $x \rightarrow y$
- Unsupervised Learning: learn the underlying structure
- Reinforcement Learning: maximize future rewards over many steps

4.1 Key Concepts

- Agent: take actions
- Environment: the world in which the agent exists and operates
- Action space: the set of possible actions an agent can make in the Environment
- Observations: of the environment after taking actions
- State: a situation which the agent perceives
- Reward: feedback that measures the success or failure of the agent's action
- Total reward: $R_t = \sum_{i=t}^{\infty} r_i$
- Discounted total reward: $R_t = \sum_{i=t}^{\infty} \gamma^i r_i$, where γ is the discount factor, $0 < \gamma < 1$
- **Q-function**: The Q-function captures the expected total future reward an agent in state, s, can receive by executing a certain action, a. $Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$
- **Policy**: The policy function infers the best action to take at its state. The policy should choose an action that maximizes future reward. $\pi^*(s) = \text{argmax} Q(s, a)$

4.2 Deep Reinforcement Learning Algorithms

- Value learning: Find $Q(s, a)$, $a = \text{argmax} Q(s, a)$
- Policy learning: Find $\pi(s)$

4.3 Deep Q Networks (DQN)

The Q-Loss function is:

$$\mathcal{L} = \mathbb{E}[||(r + \gamma \text{max} Q(s', a')) - Q(s, a)||^2]$$

Downsides of Q-learning

- Complexity:
  - Can model scenarios where the action space is discrete and small
  - Cannot handle continuous action spaces
- Flexibility:
  - Policy is deterministically computed from the Q function by maximizing the reward, thus it cannot learn stochastic policies.

4.4 Policy Learning

Function P represents the probability of selecting the corresponding action will lead to the highest reward. Note that $\sum_{a_i \in A} P(a_i|s) = 1$, the sum of all values of is 1.

The P function can be a continuous function, so it can deal with continuous action space, such as how fast should the agent move.

Policy Gradients Training Algorithm

```
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted
   in low rewards
```

```
5. Increase probability of actions that resulted
   in high rewards
```

The loss function is defined as the negative multiplication of log-likelihood of action and the reward.

$$\text{loss} = -\log P(a_t|s_t R_t)$$