Homework 1: Neural Networks and Backpropogation

Deep Learning (84100342-0)
Spring 2020
Tsinghua University

1 Introduction

Multi-task learning (MTL) aims to exploit the commonalities and differences across relevant tasks by learning them jointly. It can transfer useful information among various related tasks and has been applied to a wide range of areas such as natural language processing(NLP) and computer vision(CV). One example in NLP incorporates multiple subtasks and Directed Acyclic Graph (DAG) dependencies into judgment prediction [3]. Another typical example in CV is named as Multi-task Network Cascades for instance-aware semantic segmentation (MNC) [1], which consists of three networks, respectively differentiating instances, estimating masks, and categorizing objects, shown as Figure 1. These networks form a cascaded structure and are designed to share their convolutional features. In the MNC model, the network takes an image of arbitrary size as the input and outputs instance-aware semantic segmentation results. The cascade has three stages: proposing box-level instances, regressing mask-level instances, and categorizing each instance. Each stage involves a loss term, but a later stage's loss relies on the output of a previous stage, so these three loss terms are not independent. The entire network is trained end-to-end with a unified loss function.

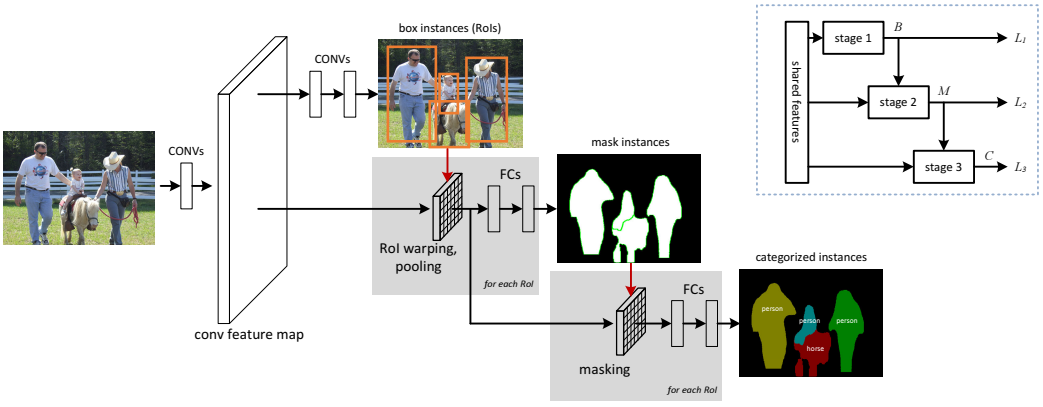

Figure 1: Multi-task Network Cascades for instance-aware semantic segmentation. At the top right corner is a simplified illustration.

Compared with common multi-task learning, this new kind of multi-task cascade mechanism further explores the dependencies of several tasks, shown in Figure 2. Thanks to the end-to-end training and the independence of external modules, the three sub-tasks and the entire system easily benefit from stronger features learned by deeper models [1]. As reported by the paper, this method achieves a mean Average Precision (mAP) of 63.5% on the PAS-CAL VOC dataset, about 3.0% higher than the previous best results using the same VGG network.

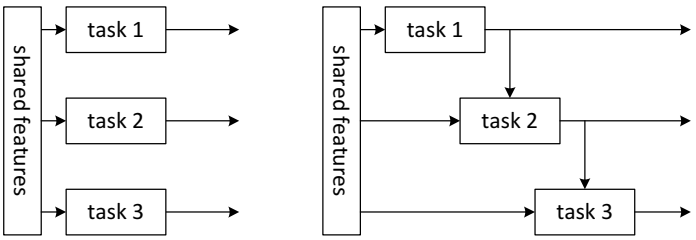

Figure 2: Illustrations of common multi-task learning (left) and multi-task cascade (right)

2 Part One: Back-propogation

Figure 3 presents a simplified multi-task cascade network for this homework. For simplicity, we only use fully connected layers and leave out convolutional and pooling layers.

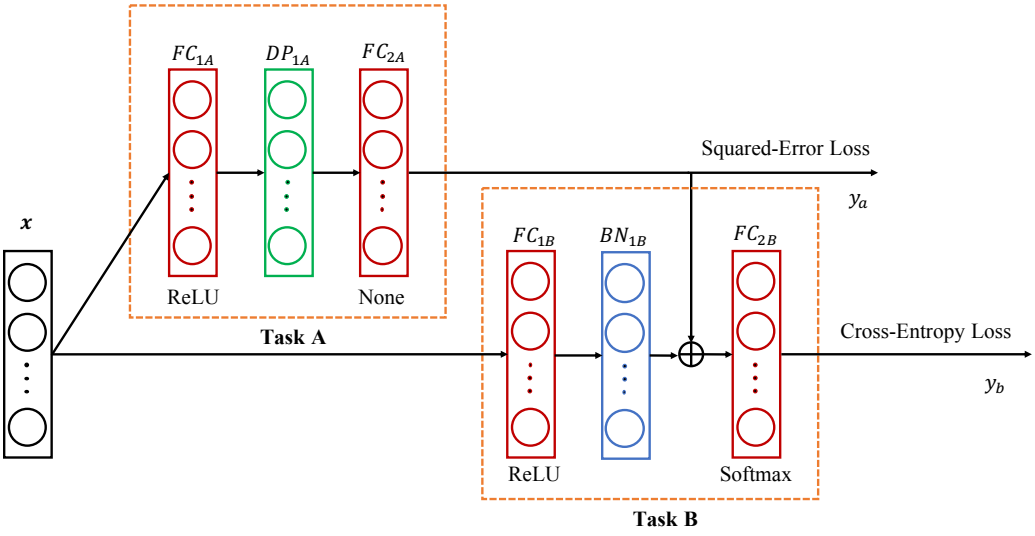

Figure 3: Schematic of the feed-forward network for multi-task learning in this homework.

2.1 Network Details

For a mini-batch of training samples ($\boldsymbol{x}$, y_a, y_b) where the batch size is equal to m:

(i) $\boldsymbol{x}_i \in \mathbb{R}^{n_x}$, the i^{th} sample of $\boldsymbol{x}$, indicates the input vectors that are fed into fully-connected layers. y_{ai}, y_{bi}, the i^{th} sample of y_a, y_b respectively, are n_{ya}, n_{yb} dimensional one-hot vectors denoting the ground-truth labels on Task A and Task B.

(ii) As illustrated by the boxes colored in red, $\mathrm{FC}_{1A}, \mathrm{FC}_{2A}, \mathrm{FC}_{1B}, \mathrm{FC}_{2B}$ are all fully-connected layers, in which $\mathrm{FC}_{1A}, \mathrm{FC}_{1B}$ serve as hidden layers with n_{1a}, n_{1b} neurons, while $\mathrm{FC}_{2A}, \mathrm{FC}_{2B}$ are output layers with n_{ya}, n_{yb} neurons respectively.

(iii) We use ReLU as the activation function of layers FC_{1A} and FC_{1B}, and Softmax for FC_{2B}. Note that FC_{2A} does not have any activation function. Let us use $\theta_{1a}, \theta_{2a}, \theta_{1b}, \theta_{2b}$ to indicate the weight matrix of each fully connected layer, $b_{1a}, b_{2a}, b_{1b}, b_{2b}$ to indicate the bias vectors, and $g(z)$ to denote activation functions. For simplicity, **the computations for the gradient of bias terms are not required.**

(iv) As illustrated by the green box, DP_{1A} is the dropout layer with a random mask vector $\mathbf{M}$. In the training stages, the probability that a neuron of FC_{1A} will be dropped is denoted as p.

Therefore, the random mask for each node j can be calculated as follows:

$$\mathbf{M}_j = \begin{cases} 0, & r_j < p \\ 1/(1-p), & r_j \geq p \end{cases},$$

where r_j is a random value between 0 and 1 for the j^{th} neuron.

(v) As illustrated by the blue box, BN_{1B} is the batch normalization layer parameterized by γ_{1b}, β_{1b}, which have the same definition as the BatchNorm Algorithm [2] shown in Figure 4.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 4: Batch Normalizing Transform, applied to activation x over a mini-batch

(vi) For each sample, the notation $\bigoplus$ in Task B means **element-wise addition** of two vectors and the output is a vector with the same size. To enable element-wise addition, n_{ya} is equal to n_{1b}.

(vii) The final predictions of the network for the i^{th} sample are $\widehat{y}_{ai}$ and $\widehat{y}_{bi}$. The overall loss function are combined by the squared-error loss of Task A and the cross-entropy loss of Task B.

$$L(\boldsymbol{x}, y_a, y_b; \theta) = \frac{1}{m}\sum_{i=1}^{m}\left[\frac{1}{2}\|(\widehat{y}_{ai} - y_{ai})\|_2^2 - \sum_{j=1}^{n_{yb}} y_{bi}^j \log(\widehat{y}_{bi}^j)\right] \tag{1}$$

2.2 Task Descriptions

You are required to finish the following tasks. When dealing with the first block of tasks: **Gradients of some basic layers**, you don't need to care about the predefined multi-task network because these tasks are designed to help you master the skill of calculating gradients of some basic layers. On the other hand, when tackling the second block of tasks: **Feed-forward and back-propagation of the predefined multi-task network**, you'd better have a good command of the predefined multi-task network and give a detailed computations.

Block One: Gradients of some basic layers (30 points)

(i) Given a BatchNorm layer, please calculate the gradients of the output $y_i = \mathbf{BN}_{\gamma,\beta}(x_i)$ with respect to the parameters of γ, β shown in Figure 4. (**10 points**)

(ii) Given a dropout layer, please calculate the gradients of **the output of a dropout layer** with respect to **the input of a dropout layer**. (**10 points**)

(iii) Given a Softmax function, please calculate the gradients of **the output of a Softmax function** with respect to **the input of a Softmax function**. (**10 points**)

Block Two: Feed-forward and back-propagation of the multi-task network (30 points)

(i) Finish the detailed **feed-forward computations** of a batch samples ($\boldsymbol{x}$, y_a, y_b) during a training iteration, coming with final predictions ($\widehat{y}_a$, $\widehat{y}_b$) of Task A, Task B. (**10 points**)

(ii) Use the back-propagation algorithm we have learned in class and give **the gradients of the overall loss function with respect to the parameters at each layer** corresponding to a batch of samples. (**20 points**)

Note that:

(i) Please show necessary derivations of the gradients.

(ii) Please attach variable notations in the gradients expressions.

(iii) A vectorial form of gradients expressions are highly encouraged.

3 Part Two: Code Implementation

3.1 Multilayer Perceptron (MLP)

Figure 5 presents a typical Multilayer Perceptron (MLP) for each branch of the multi-task cascade network in Section 2 with little modification. In this part, you are required to implement and train this 3-layer neural network to classify images of **hand-written digits** from the MNIST dataset. For each example, the input to the network is a 28×28-pixel image, which is converted into a 784-dimensional vector and then output a vector of 10 probabilities (one for each digit).

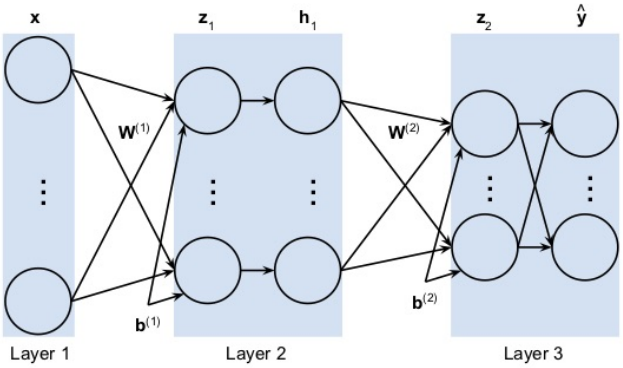

Figure 5: The network architecture of the Multilayer Perceptron (MLP).

3.2 Forward Propagation

Compute the intermediate outputs $\mathbf{z}_1$, $\mathbf{h}_1$, $\mathbf{z}_2$, and $\widehat{\mathbf{y}}$ as the directed graph shown below. Specifically, the network you create should implement a function $g : \mathbb{R}^{784} \to \mathbb{R}^{10}$, where:

$$\mathbf{z}_1 = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\mathbf{h}_1 = \mathrm{ReLU}(\mathbf{z}_1)$$

$$\mathbf{z}_2 = \mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)}$$

$$\widehat{\mathbf{y}} = g(\mathbf{x}) = \mathrm{Softmax}(\mathbf{z}_2)$$

3.3 Loss function

After forward propagation, you should use the cross-entropy loss function:

$$f_{\mathrm{CE}}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{10}\mathbf{y}_k^{(i)}\log\widehat{\mathbf{y}}_k^{(i)}$$

where n is the number of examples.

3.4 Backward Propagation

The individual gradient for each parameter term can be shown as follows:

$$\frac{\partial f_{CE}}{\partial \mathbf{W}^{(2)}} = \frac{1}{n}\sum_{i=1}^{n}(\widehat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)})(\mathbf{h_1}^{(i)})^T$$

$$\frac{\partial f_{CE}}{\partial \mathbf{b}^{(2)}} = \frac{1}{n}\sum_{i=1}^{n}(\widehat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)})$$

$$\frac{\partial f_{CE}}{\partial \mathbf{W}^{(1)}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{W}^{(2)^T}(\widehat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}) \circ \mathrm{sgn}(\mathbf{z_1}^{(i)})(\mathbf{x}^{(i)})^T$$

$$\frac{\partial f_{CE}}{\partial \mathbf{b}^{(1)}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{W}^{(2)^T}(\widehat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}) \circ \mathrm{sgn}(\mathbf{z_1}^{(i)})$$

3.5 Task Descriptions

- Implement stochastic gradient descent for the network shown above with the help of the starter code. Specially, you need to finish the code of three functions: ***fCE*** (X, Y, w), ***gradCE*** (X, Y, w), ***train***(trainX, trainY, testX, testY, w). (**20 points**)
- Train the network using proper hyper-parameters (batch size, learning rate etc), and report the train accuracy and test accuracy. (**20 points**)

4 What you should submit

For Part One: Back-propogation, you should submit a *pdf* fomat report with detailed computations. For Part Two: Code Implementation, you should only submit the *python* source code and ensure that it is runable, but no report is required.

5 Knowledge Checklist

After finishing this homework, we hope you mater the following knowledges or technique skills:

- Know how to calculate the gradients of basic layers, including fully connected layer, Softmax, Batch Normalization, Dropout etc.
- Master the feed-forward and back-propagation of a neural network.
- Know how to implement the stochastic gradient descent in python, including three typical steps: feed-forward, back-propagation and parameter update.
- Know the difference between epoch and iteration.
- Know that stochasticity is important for training neural networks and shuffle the dataset for each epoch while training.
- Know how to babysit a simple neural network by adjusting the hyper-parameters, including learning rate, batch size etc.

References

[1] J. Dai, K. He, and J. Sun. Instance-aware semantic segmentation via multi-task network cascades. *Computer Vision and Pattern Recognition*, pages 3150–3158, 2016.

[2] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[3] H. Zhong, Z. Guo, C. Tu, C. Xiao, Z. Liu, and M. Sun. Legal judgment prediction via topological learning. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3540–3549, 2018.