Homework 1: Neural Networks and Backpropogation

Deep Learning (84100343-0)
Spring 2022
Tsinghua University

1 Introduction

Multi-task learning (MTL) aims at exploiting 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 [10]. Another typical example in CV is named as Multi-task Network Cascades for instance-aware semantic segmentation (MNC) [2], 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.

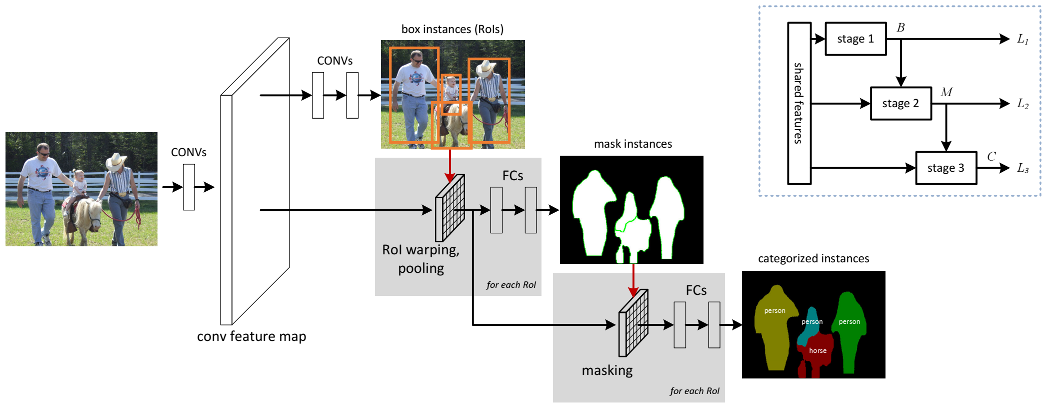

Figure 1: Multi-task Network Cascades for instance-aware semantic segmentation. At the top right corner is a simplified illustration.

Compared with common multi-task learning, this new kind of multi-task cascade mechanism further explores the dependences of several tasks, as 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 [2]. As reported by the paper, this method achieves a mean Average Precision (mAP) of 63.5% on the PASCAL VOC dataset, about 3.0% higher than the previous best results using the same VGG network.

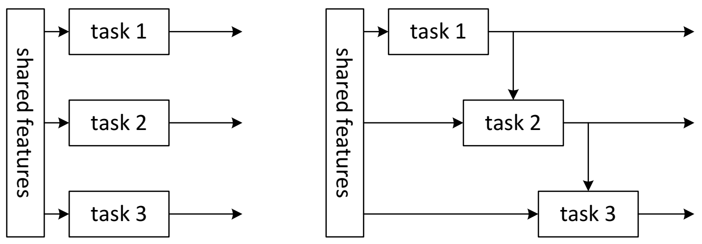

Figure 2: Illustrations of common multi-task learning (left) and multi-task cascade (right).

2 Part One: Code Implementation

2.1 Multilayer Perceptron (MLP)

Figure 3 presents a simplified Multilayer Perceptron (MLP) for each branch of the multi-task cascade network mentioned above. 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 the output is a vector of 10 probabilities (one for each digit).

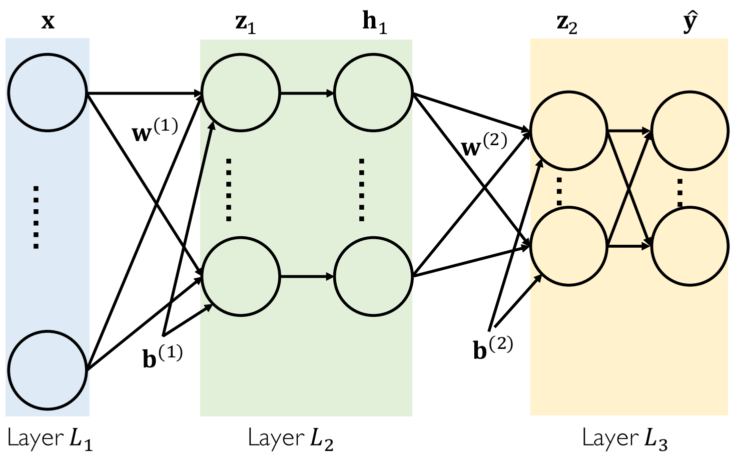

Figure 3: The network architecture of the Multilayer Perceptron (MLP).

Here are some concepts you need to be familiar with.

2.1.1 Forward Propagation

The intermediate outputs $\mathbf{z}_1$, $\mathbf{h}_1$, $\mathbf{z}_2$, and $\widehat{\mathbf{y}}$ as the directed graph are shown below:

$$\begin{aligned} \mathbf{z}_1 &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{h}_1 &= \mathrm{ReLU}(\mathbf{z}_1) \\ \mathbf{z}_2 &= \mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)} \\ \widehat{\mathbf{y}} &= \mathrm{Softmax}(\mathbf{z}_2). \end{aligned} \tag{1}$$

2.1.2 Loss function

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

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

where m is the number of examples in each mini-batch.

2.1.3 Backward Propagation

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

$$\begin{aligned}
\frac{\partial f_{\mathrm{CE}}}{\partial \mathbf{W}^{(2)}} &= \frac{1}{m}\sum_{i=1}^{m}(\widehat{\mathbf{y}}^i - \mathbf{y}^i)(\mathbf{h}_1^i)^{\mathsf{T}} \\
\frac{\partial f_{\mathrm{CE}}}{\partial \mathbf{b}^{(2)}} &= \frac{1}{m}\sum_{i=1}^{m}(\widehat{\mathbf{y}}^i - \mathbf{y}^i) \\
\frac{\partial f_{\mathrm{CE}}}{\partial \mathbf{W}^{(1)}} &= \frac{1}{m}\sum_{i=1}^{m}\mathbf{W}^{(2)\mathsf{T}}(\widehat{\mathbf{y}}^i - \mathbf{y}^i)\circ \mathrm{sgn}(\mathbf{z}_1^i)(\mathbf{x}^i)^{\mathsf{T}} \\
\frac{\partial f_{\mathrm{CE}}}{\partial \mathbf{b}^{(1)}} &= \frac{1}{m}\sum_{i=1}^{m}\mathbf{W}^{(2)\mathsf{T}}(\widehat{\mathbf{y}}^i - \mathbf{y}^i)\circ \mathrm{sgn}(\mathbf{z}_1^i).
\end{aligned} \tag{3}$$

2.1.4 Task Descriptions

We provide the starter code in `mlp.py`. To get full scores, you **only** need to complete the code marked with **TODO**. Besides, you are free to modify other parts of the code for your convenience.

To get you comfortable with gradient derivation, in this part we will be using `numpy`. Frameworks that support auto-derivation are **not** allowed. We encourage you to adopt a **vectorized** implementation to speed up computation.

The concrete tasks and their scores are as follows:

- Implement forward propagation and backward propagation algorithms. Train the network and plot the change in loss during training. (**10 points**)
- Train the network using proper hyper-parameters (batch size, learning rate, etc), and report the training accuracy and test accuracy. The test accuracy should exceed 90%. (**10 points**)

2.2 Variational Autoencoder (VAE)

In this part, you will implement a variational autoencoder [5] using PyTorch [6]. In addition to understanding the principles of VAE, we expect you to get a better grasp of PyTorch. You can learn more about the usage of PyTorch through the PyTorch Tutorial. Next, we will walk you through the implementation of VAE.

2.2.1 Forward Propagation

Recall that VAE consists of two components, the encoder $q_\phi(z|x)$ and the decoder $p_\theta(x|z)$, which are both MLPs in this assignment.

For a given input x, we first forward it through the encoder $q_\phi(z|x)$ and get μ_z and $\log(\sigma_z)$ estimated by the encoder. We then sample initial random data ϵ from $\mathcal{N}(0,1)$ and compute z as

$$z = \mu_z + \sigma_z\epsilon. \tag{4}$$

In fact, we use the so called reparametrization trick above. With z, we can forward it through the decoder $p_\theta(x|z)$ and similarly get μ_x and $\log(\sigma_x)$.

2.2.2 Loss function

As shown below, the loss function for VAE contains two terms: A reconstruction loss term (left) and KL divergence term (right)

$$-\mathrm{E}_{Z_{q_\phi(z|x)}}[\log p_\theta(x|z)] + D_{KL}(q_\phi(z|x), p(z)). \tag{5}$$

Note that this is the negative of the evidence lower bound (ELBO). When we are minimizing this loss term, we're maximizing the ELBO.

With above variables, the reconstruction loss is calculated as

$$-\mathrm{E}_{Z_{q_\phi(z|x)}}[\log p_\theta(x|z)] = \frac{1}{2}\log(2\pi) + \log(\sigma_x) + \frac{(x-\mu_x)^2}{2{\sigma_x}^2}. \tag{6}$$

The KL divergence is calculated as

$$D_{KL}(q_\phi(z|x), p(z)) = -\log(\sigma_z) - \frac{1}{2} + \frac{1}{2}({\mu_z}^2 + {\sigma_z}^2). \tag{7}$$

2.2.3 Task Descriptions

We provide the starter code in `vae.py`. To get full scores, you **only** need to complete the code marked with **TODO**. Besides, you are free to modify other parts of the code for your convenience.

The concrete tasks and their scores are as follows:

- Implement forward propagation. Train the VAE to generate 2D samples that follow a specific distribution. (**10 points**)

3 Part Two: Back-propogation

Figure 4 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.

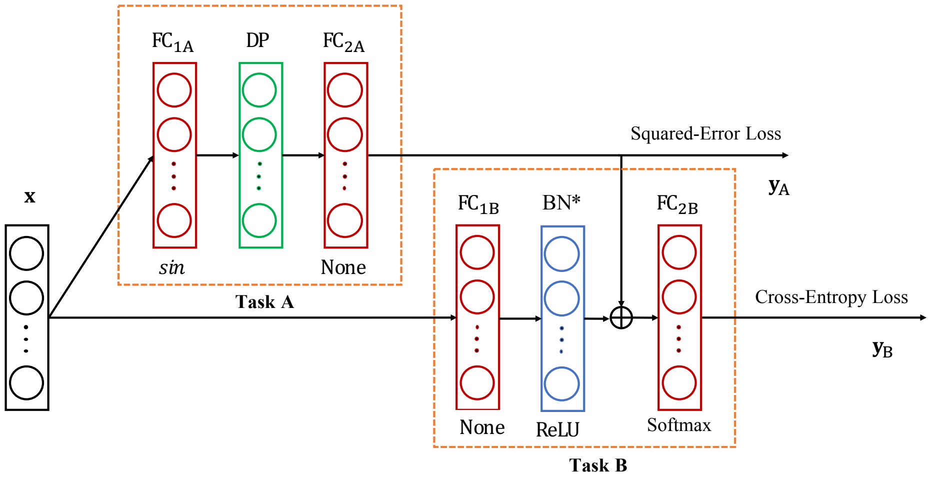

Figure 4: Schematic of the feed-forward network for multi-task learning in this homework.

3.1 Network Details

For a mini-batch of training samples ($\mathbf{x}$, $\mathbf{y}_\mathrm{A}$, $\mathbf{y}_\mathrm{B}$) with a batch size of m:

(i) For each input sample $\mathbf{x}^i \in \mathbb{R}^d$ in a mini-batch, the prediction outputs for task A and task B are $\mathbf{y}^i_\mathrm{A} \in \mathbb{R}^a$ and $\mathbf{y}^i_\mathrm{B} \in \mathbb{R}^b$ respectively. $\mathbf{y}^i_\mathrm{A}$ is usually a vector with continuous value for regression task, while $\mathbf{y}^i_\mathrm{B}$ is a one-hot vector for classification task.

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

(iii) Denote weight matrixs of all fully connected layers as $\theta_{1\mathrm{A}}$, $\theta_{2\mathrm{A}}$, $\theta_{1\mathrm{B}}$ and $\theta_{2\mathrm{B}}$, whose bias vectors are $\mathbf{b}_{1\mathrm{A}}$, $\mathbf{b}_{2\mathrm{A}}$, $\mathbf{b}_{1\mathrm{B}}$ and $\mathbf{b}_{2\mathrm{B}}$ respectively. Note that, $\mathrm{FC}_{1\mathrm{B}}$ is slightly different from the standard fully connected layer, detailed in (vi).

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

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

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

(v) $\mathrm{FC}_{1\mathrm{A}}$ uses the *sine* as a periodic activation function, proposed by a recent paper [8] published in NeurIPS 2020 as a oral presentation. Neural networks with this loss are demonstrated to fit complicated signals, such as natural images and 3D shapes, and their derivatives robustly.

(vi) As illustrated by the blue box, BN^* is a mean-only batch normalization layer [7] for $\mathrm{FC}_{1\mathrm{B}}$. Different from the standard batch normalization [4], inputs are only subtracted by the mini-batch means, without being divided by the mini-batch standard deviations. Formally,

$$\begin{aligned} \mathbf{x}_{1\mathrm{B}} &= \theta_{1\mathrm{B}}\mathbf{x} \\ \mu &= \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_{1\mathrm{B}}^{i} \\ \mathbf{x}_{\mathrm{BN}} &= \mathbf{x}_{1\mathrm{B}} - \mu + \mathbf{b}_{1\mathrm{B}}. \end{aligned} \tag{8}$$

After that, $\mathbf{x}_{\mathrm{BN}}$ is fed into a ReLU activation function.

(vii) For each sample, the notation $\bigoplus$ in Task B means *element-wise addition* of two vectors. To enable element-wise addition, a is equal to s.

(viii) For each input sample $\mathbf{x}^i$, the final predictions of two tasks are $\widehat{\mathbf{y}}_{\mathrm{A}}^{i}$ and $\widehat{\mathbf{y}}_{\mathrm{B}}^{i}$. The overall loss functions include the squared-error loss of Task A and the cross-entropy loss of Task B:

$$\mathcal{L} = \frac{1}{m}\sum_{i=1}^{m}\left[\frac{1}{2}\|(\widehat{\mathbf{y}}_{\mathrm{A}}^{i} - \mathbf{y}_{\mathrm{A}}^{i})\|_2^2 - \sum_{k=1}^{b}\mathbf{y}_{\mathrm{B},k}^{i}\log\widehat{\mathbf{y}}_{\mathrm{B},k}^{i}\right]. \tag{9}$$

3.2 Homework Description

Block One: gradients of some basic layers: (**30 points**)

(i) Given a standard 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 5. (**10 points**)

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

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

Block Two: feed-forward and backpropagation of the multi-task network: (**40 points**)

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

(ii) Use the backpropagation algorithm we have learned in class and give **the gradients of the overall loss in a mini-batch with respect to the parameters at each layer**. (**30 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.

4 What you should submit

For Part One, you should only submit the python source code and ensure that it is runable. No report is required in this part. For Part Two, you should submit a report with detailed computations.

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

    $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                   // mini-batch mean
    $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$          // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$                  // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$               // scale and shift
```

Figure 5: Batch Normalization Transform, applied to activation x over a mini-batch.

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 (shuffle the dataset) is important for training neural networks.
- Know how to babysit a simple neural network by adjusting the hyper-parameters, including learning rate, batch size etc.
- Know the variational inference and how it is used to design Bayesian deep models.
- Get an overview of how to define and train models with PyTorch.

6 Resources

6.1 Various VAEs

- Variational Lossy Autoencoder (VLAE [1]).
- beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework ($\beta-$VAE [3])
- Neural Discrete Representation Learning (VQ-VAE [9]).
- Awesome-VAEs (github repository).

References

[1] X. Chen, D. P. Kingma, T. Salimans, Y. Duan, P. Dhariwal, J. Schulman, I. Sutskever, and P. Abbeel. Variational lossy autoencoder. In *ICLR*, 2017.

[2] J. Dai, K. He, and J. Sun. Instance-aware semantic segmentation via multi-task network cascades. In *CVPR*, 2016.

[3] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *ICLR*, 2017.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[5] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[7] T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *NeurIPS*, 2016.

[8] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions. In *NeurIPS*, 2020.

[9] A. Van Den Oord, O. Vinyals, et al. Neural discrete representation learning. In *NeurIPS*, 2017.

[10] H. Zhong, Z. Guo, C. Tu, C. Xiao, Z. Liu, and M. Sun. Legal judgment prediction via topological learning. In *EMNLP*, 2018.