Robot Imagination

Shaowei Sun and Quan Xiong

Abstract: This work is designed to have a deep insight into robot imagination with the tools of neural networks. The main assignment is to sort different hand-written numbers or flower pictures to realize the goal of ‘imagination’. In this process, this work establishes basic two-layer CNNs from scratch and employ the pre-trained ResNet152 model for transfer learning. By adjusting different hyperparameters and applying various modifications, this work shows good performances in the real-life tasks.
Keywords: robot imagination; CNNs; ResNet.

1. Introduction

The term 'robot imagination' encompasses a range of interpretations. However, when deconstructed to its literal meaning, 'imagination,' it becomes closely associated with smart robots or artificial intelligence (AI). Specifically, ‘imagination’ refers to the ability of deducing strategies and leveraging past experiences to generate decisions, which is just the core idea of neural networks.

Generally, there are several types of traditional neural networks, namely KNNs, SOM, RBFNs, etc. However, the most popular one is CNNs, which are widely applied in several areas, such as machine vision, deep learning, self-driving automobiles, and more. Basically, CNNs rely on convolutional layers to extract the information contained in arrangements of pixels, which generates classification predictions with the cooperation of other functional layers. This makes CNNs not only a robust tool but also the foundational basis for several renowned deep learning structures.

Visual Graphics Group Networks, also known as VGG, are a family of convolutional neural networks developed and popularized by Karen Simonyan and Andrew Zisserman from the University of Oxford in 2014 [1]. The VGG models are characterized by their simplicity, using only 3x3 convolutional layers stacked on top of each other in increasing depth, with ReLU activation functions, and 2x2 pooling layers to reduce the dimensionality. The fully connected layers at the end of the network have 4096 units each and use dropout for regularization, and the final layer performs classification. VGG was one of the first models to show that the depth of the network is a critical component for good performance. Residual Networks, also known as ResNet, is another type of neural network architecture that facilitates training very deep neural networks with hundreds or even thousands of layers, which was previously practically unattainable due to the vanishing gradient problem [2]. The key innovation in ResNet is the introduction of "skip connections" or "shortcuts" that allow the input of one layer to bypass one or more intermediate layers and be added to the output of those layers.

In this assignment, CNNs with two convolutional layers are constructed from scratch. After deploying the Python codes on the CPU, a computer needs quite a long time to reach the minimum error state. Concrete configurations would be discussed in the following section, including Cross-Entropy loss, SoftMax layer, etc. The final accuracy of predictions based on the MNIST dataset is 97.4%, which is the same as that on the official website. Further, transfer learning is utilized to tackle the realistic problem, which is the classification of various flowers. These flowers’ pictures are downloaded from the internet and share similarity within the same category, as judged by human eyes. Because it would take lots of resources to train the deep learning networks from scratch, especially when training the classic ResNet152 with more than 152 hidden layers, Pytorch provides a trained model for transfer learning, contributed by previous academic

research. Detailed information will be presented in the following section. In the end, ResNet152 reached more than 85% accuracy on the training dataset and about 75% accuracy on the testing dataset.

2. Theoretical Analysis

Foundational functions of each layer include Forward Propagation and Backward Propagation. In forward propagation, each layer makes inferences based on its current set of weights and biases, receiving input from the previous layer's outcomes or the original input images, and carries forward computation in a forward direction through the network. Backward propagation is a supervised learning algorithm used for training artificial neural networks. The main goal of backward propagation is to minimize the error between the network's prediction and the actual target values by adjusting the network's weights and biases.

There are several types of optimizers whose functions are to adjust the attributes of a neural network to reduce losses. When establishing CNNs from scratch, a two-layer model is not impeded by the processes of updating weights and learning rates, which conversely is a prominent problem in deep learning networks. Hence, Stochastic Gradient Descent (SGD) is employed in this assignment because it is proven to be the most efficient and simplest method in this situation. SGD works by randomly selecting a training sample, computing the gradient of the loss function with respect to the model parameters, and then updating the parameters in the opposite direction of the gradient. This process is repeated until the algorithm converges to the optimal set of parameters that minimize the loss function. The learning rate is a hyperparameter that determines the size of the steps taken towards the minimum. While SGD can sometimes converge to a local minimum or oscillate around the minimum, variations and improvements such as learning rate schedules, momentum, and weight decay can help mitigate these issues.

Convolutional layer

A convolutional layer is a core building block of CNNs, where the input image is processed using a set of learnable filters, also known as kernels. This operation allows the network to be sensitive to spatial hierarchies of features in the input data, making it well-suited for processing data with a grid-like topology, such as images. The formula for Forward Propagation goes as follows:

$$\mathrm{Y_i} = \mathrm{B_i} + \sum_{\mathrm{j=1}}^{\mathrm{n_c}} \mathrm{X_j} \star \mathrm{K_{ij}} \quad \mathrm{i} = 1, \dots, \mathrm{d}$$

where $\mathrm{X_j}$ stands for a single channel of the original input, which could also be inheritted from the previous layer with a certain depth; $\mathrm{K_{ij}}$stands for a corresponding kernel, which is implemented with the valid cross-correlation operation, denoted by $\star$; $\mathrm{Y_i}$ stands for the output; d stands for the depth of the output. The formulas of Backward Propagation go as follows:

$$\frac{\partial \mathrm{E}}{\partial \mathrm{X_j}} = \sum_{\mathrm{i=1}}^{\mathrm{d}} \frac{\partial \mathrm{E}}{\partial \mathrm{Y_i}} * \partial \mathrm{K_{ij}}$$

where $*$ is the sign of the full convolution operation.

By applying the SGD method, updating weights and biases follows the classic "Gradient Descent" strategy:

$$\frac{\partial \mathrm{E}}{\partial \mathrm{K_{ij}}} = X_j \star \frac{\partial \mathrm{E}}{\partial Y_i}$$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{B_i}} = \frac{\partial \mathrm{E}}{\partial \mathrm{Y_i}}$$

Hence,

$$\mathrm{K_{ij}} \leftarrow \mathrm{K_{ij}} - \alpha \frac{\partial \mathrm{E}}{\partial \mathrm{K_{ij}}}$$

$$\mathrm{B_i} \leftarrow \mathrm{B_i} - \alpha \frac{\partial \mathrm{E}}{\partial \mathrm{B_i}}$$

Max-pooling layer

Max-pooling is a sample-based discretization process that aims to down-sample the input representation by reducing its spatial dimensions, which helps both in reducing computational costs and controlling overfitting. It operates on each feature map separately to produce a pooled

feature map. Max-pooling kernel is more efficient than the average-pooling kernel, because the feature maps would only retain the significant features and eliminate unimportant information, which is just what the neural network needs to learn. Define $\Omega(y_{mn})$ as the set of input pixels that contribute to output y_{mn}.

$$\Omega(y_{mn}) = \begin{bmatrix} x_{mn} & x_{(m+1)n} & \cdots & x_{(m+i)n} \\ x_{m(n+1)} & x_{(m+1)(n+1)} & \cdots & x_{(m+i)(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m(n+j)} & x_{(m+1)(n+j)} & \cdots & x_{(m+i)(n+j)} \end{bmatrix}$$

where $I \times J$ is the size of the pooling kernel. The formula of Forward Propagation goes as follows:

$$y_{mn} = P_{max}[\Omega(y_{mn})] = \max \Omega(y_{mn})$$

The formula of Backward Propagation goes as follows:

$$\frac{\partial E}{\partial x_{ab}} = \begin{cases} \frac{\partial E}{\partial y_{mn}}, & x_{ab} = P_{max}[\Omega(y_{mn})] \\ 0, & otherwise \end{cases}$$

Dense layer

The dense layer, also called the fully connected layer, takes the flattened one-column vector from the last convolution-pooling layer as input, reshaping it into a N-dimensional vector, where N is the number of categories. For the dense layer, Forward Propagation and Backward Propagation are the same as for a single-layer perceptron. The formula of Forward Propagation goes as follows:

$$Y = W \cdot X + B$$

The formula of Backward Propagation goes as follows:

$$\frac{\partial E}{\partial X} = W^T \cdot \frac{\partial E}{\partial Y}$$

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_j \end{bmatrix} W = \begin{bmatrix} \omega_{11} & \omega_{21} & \cdots & \omega_{i1} \\ \omega_{12} & \omega_{22} & \cdots & \omega_{i2} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{1j} & \omega_{2j} & \cdots & \omega_{ij} \end{bmatrix} X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \end{bmatrix}$$

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{bmatrix}$$

By applying the SGD method, updating weights and biases follows the classic "Gradient Descent" strategy:

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} \cdot X^T$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

Hence,

$$W \leftarrow W - \alpha \frac{\partial E}{\partial W}$$

$$B \leftarrow B - \alpha \frac{\partial E}{\partial B}$$

Activation layer

In neural networks, activation functions, which are often represented by activation layers, play a crucial role. The most essential role of an activation function is to introduce non-linearity into the network. Without non-linearity, even a deep neural network would behave like a single-layer linear model because the composition of linear functions is linear. Non-linearity allows the network to capture and model more complex relationships in the data. In this assignment, only the ReLU activation function is introduced in the activation layer. The formula of ReLU function goes as follows:

$$ReLU(x) = \max(0, x)$$

The derivative of ReLU function with respect to the input goes as follows:

$$ReLU'(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Softmax layer with cross-entropy loss

Cross entropy is a commonly used loss function in CNNs and is often used in conjunction with softmax activation function in the output layer. This concept comes from Information Theory. The cross-entropy from P to Q, denoted $H(P, Q) = \sum_j -P(j) \log Q(j)$, is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according

to probabilities P. The lowest possible cross-entropy is achieved when $\mathrm{P} = \mathrm{Q}$.

Considering cross-entropy loss with softmax activation function, the formula goes as follows:

$$\mathrm{l}(\mathrm{Y}, \hat{\mathrm{Y}}) = -\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{y}_\mathrm{i} \log \frac{\exp(\mathrm{o}_\mathrm{i})}{\sum_{\mathrm{k}=1}^{\mathrm{n}} \exp(\mathrm{o}_\mathrm{k})}$$

$$= \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{y}_\mathrm{i} \log \sum_{\mathrm{k}=1}^{\mathrm{n}} \exp(\mathrm{o}_\mathrm{k}) - \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{y}_\mathrm{i} \mathrm{o}_\mathrm{i}$$

$$= \log \sum_{\mathrm{k}=1}^{\mathrm{n}} \exp(\mathrm{o}_\mathrm{k}) - \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{y}_\mathrm{i} \mathrm{o}_\mathrm{i}$$

Hence, the derivative of cross-entropy loss with respect to any logit o_j goes as follows:

$$\partial_{\mathrm{o}_\mathrm{j}} \mathrm{l}(\mathrm{Y}, \hat{\mathrm{Y}}) = \frac{\exp(\mathrm{o}_\mathrm{i})}{\sum_{\mathrm{k}=1}^{\mathrm{n}} \exp(\mathrm{o}_\mathrm{k})} - \mathrm{y}_\mathrm{i} = \mathrm{softmax}(\mathrm{o}_\mathrm{j}) - \mathrm{y}_\mathrm{j}$$

The derivative with respect to any logit o_j is the difference between the probability assigned by the model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector. In this sense, it is very similar to the loss in regression, where the gradient is the difference between the observation y and estimate $\hat{y}$. This is not a coincidence. In any exponential family model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Dropout Layer

A Dropout layer in a Convolutional Neural Network (CNN) is used to prevent overfitting during the training process. It randomly sets a fraction of input units to 0 at each update during training, which helps to prevent overfitting. The dropout rate defines the probability at which inputs are dropped out. In this assignment, a dropout rate of 0.5 means that there is a 50% probability that any given unit will be dropped out.

3. Outcomes and Discussions

Two-layer CNNs are comprised of one convolutional layer, one ReLU layer, one max-pooling layer, followed by another convolutional layer, ReLU layer, and max-pooling layer, and finally, a reshaping layer, a dense layer, and a softmax layer. The learning rate is set to decay throughout the training process. As a result, the loss decreases quickly in the initial stage and gradually reaches a minimum state towards the end. Ultimately, the accuracy of predictions, based on a testing dataset of 300 pictures, reaches 97.3%.

Figure 1: predicted labels compared with original labels by transfer learning in ResNets.

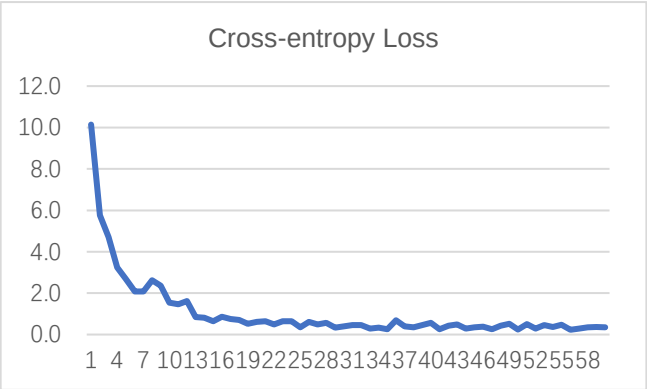

Figure 2: variances of loss in the training process of two-layer CNNs.

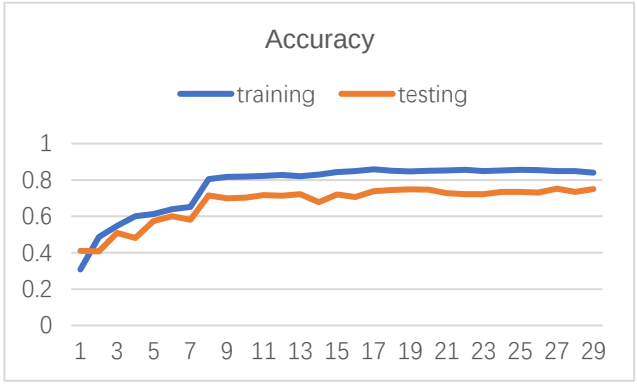

Figure 4: variances of accuracy in the training process of ResNets152.

ResNet152 is adapted according to the task at hand. Initially, only the dense layer at the end of ResNet152 is trained, leveraging the potential benefits of the pre-trained model. The results show that within the first 20 epochs, cross-entropy decreases significantly, and the accuracy of the training dataset reaches its peak. In the subsequent step, the entire ResNet152 model is fine-tuned for a more stable version. In conclusion, the last 10 epochs result in a gradual increase in the accuracy of predictions, stabilizing all parameters in the ResNet152 model. In the figure 1, out of eight examples, only one picture, marked in red, is misclassified.

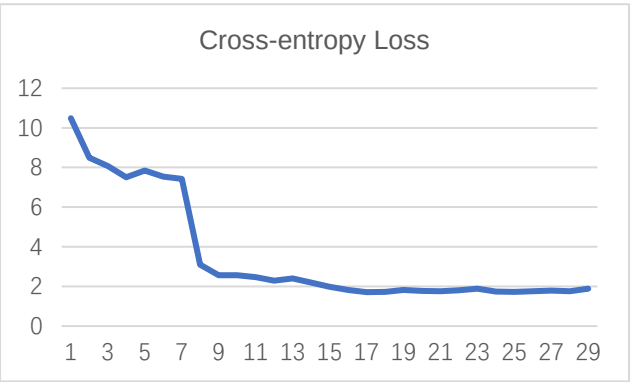

Figure 3: variances of loss in the training process of ResNets152.

4. Conclusion

In conclusion, utilizing transfer learning to train the ResNet152 model in PyTorch has proven to be an effective approach for flower classification. By leveraging the pre-trained weights of ResNet152, we were able to harness the power of a deep neural network without the need for extensive computational resources and training time. This approach enabled the model to achieve high accuracy in distinguishing between various flower species, highlighting the potential of transfer learning in efficiently solving specific classification problems. Future work can include further fine-tuning of the model, experimenting with different hyperparameters, and expanding the dataset to include a wider variety of flower species, ultimately improving the model's performance and robustness.

5. Code

GitHub link:
https://github.com/egvalley/Robot_Imagination

[1]. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
[2]. He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.