Project VIII Part 2: Classification

1 Linear Classifier for Multiple Classes

By far, we treat the age prediction as a regression problem. In fact, we can also model it as a multi-class classification problem. For the purpose, we suppose that age is an integer between the range $[0, 100]$. Thus, there are totally 101 possibilities (classes) for age. To extend the previous linear perceptron for the 101 classes, a linear module is first used.

$$Z = XW + b$$

where W is a matrix of 2048×101 and b is the bias. Then Z is a matrix of $N \times 101$ where N is the number of samples. The Z is usually called the **logit**. Here, the arg max of Z can be used to index the class for each sample.

$$\text{Class}_i = \arg\max_j Z_{ij}$$

To find the optimal paramters W, we work from the probabilistic pespective.

1.1 Classification for age prediction

With logit, we can caluclate ,

$$P_{ij} = \frac{\exp(Z_{ij})}{\sum_j \exp(Z_{ij})}$$

Here, the probability gives the confidence for that the i-th sample's age is j. We usually use the cross entropy to supervise the learning of the classification model. Let $\mathbf{y}$ denote the ground truth class, then $y_i \in \{0, 1, \cdots, 100\}$ is the actual age of the i-th sample. Then, cross entropy (CE) loss for i-th sample is as follows.

$$CE_i(W) = -\log(P_{i,y_i}) = -Z_{i,y_i} + \log\left(\sum_j \exp(Z_{ij})\right)$$

By minimizing the losses of all samples,

$$J(W, b) = \sum_i CE_i(W, b)$$

we can search for the optimal weights W by gradient descent.

1.2 Layer for linear classifier and its loss

In the part, you need to implement the linear module and the cross entropy for our linear classifier.

- **Linear**
- **CrossEntropyLosswithSoftmax**

The two modules need to be implemented with pure python code with numpy in 'pynetwork.ipynb'.
In the part, each module is implemented as a python class which contains four functions,

- **__init__**
- **init_param**(optional): layers without parameters, like ReLU don't have this function.
- **forward**: forward pass for gradient calculation
- **backward**: backward pass for gradient calculation

We have given the previous two functions. In the assignment, you need to implement both forward and backward computation.

1.3 Linear layer

The Linear layer (also called fully connected layer) performs linear transformation on input data,

$$Y = XW + b$$

This layer has two learn-able parameters, weight of shape (input_channel, output_channel) and bias of shape (output_channel). Both parameters are specified and initialized in **init_param**() function. For initialization, bias b can be set to zeros. The weight W can be initialized with the normalized Gaussian noise,

$$W \leftarrow \frac{\text{Gaussian(input\_channel,output\_channel)}}{\sqrt{\frac{\text{input\_channel+output\_channel}}{2}}}$$

Protocols for the forward and backward function can be found in the supplied file. The **forward** of linear module calculates the output Y from the input X. For **backward**, you need to calculate the gradients w.r.t the input X, the weights W and the bias b given the gradient of the loss w.r.t the output Y. Let $\frac{\partial J}{\partial Y}$ denote the gradient of the loss function w.r.t the output, the **backward** calculate the gradients w.r.t inputs and parameters via,

$$\frac{\partial J}{\partial X} = \frac{\partial J}{\partial Y}\frac{\partial Y}{\partial X} \qquad \frac{\partial J}{\partial W} = \frac{\partial J}{\partial Y}\frac{\partial Y}{\partial W} \qquad \frac{\partial J}{\partial b} = \frac{\partial J}{\partial Y}\frac{\partial Y}{\partial b}$$

To combine these gradients, you should take care of their sizes.

1.4 Cross Entropy Loss with Softmax

In classification task, we usually firstly apply softmax to map class-wise prediction into the probability distribution then we use cross entropy loss to maximise the likelihood of ground truth class's prediction. In this assignment, you will implement both forward and backward computation. (Detailed description about softmax and cross entropy can be found in section1.1.) The **backward** of cross entropy takes into the gradient $\frac{\partial J}{\partial CE}$, and outputs the gradient w.r.t its input. The input to the backward function $\frac{\partial J}{\partial CE}$ is a $N \times 1$ vector of all ones.

1.5 Provided functions

In 'pynetwork.ipynb', we supplied the framework for training with the two layers. All your implementations should make the file run correctly and generate results like that in Fig. 1.

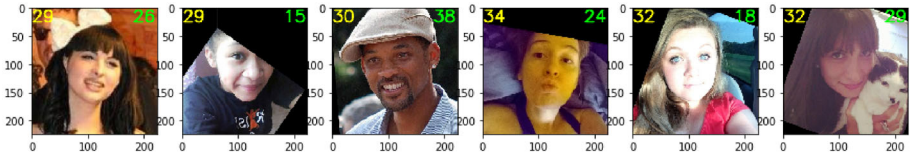

Figure 1: Result from the pynetwork.ipynb.

2 Model immigration

Now you have built the connection between the layers in PyTorch with the forward/backward pass used in your linear regression project. It's time for us to immigrate the model used in linear regression to PyTorch. In the part, you will *immigrate the linear classifier model for age prediction to PyTorch.* In 'network.ipynb', we have supplied the framework for the immigration.

2.1 Manipulation of the data

In linear regression, we load in the data and use the numpy array to manipulate it. In PyTorch, this job is done by module of 'Dataset' and 'Dataloaer'. We have supplied the codes, you should modify the 'base_dir' of the 'FaceNPDataset' to make it work. To understand the behaviours of 'Dataset' and 'Dataloader', you can consult the document of PyTorch (`https://pytorch.org/`).

2.2 Model immigration

Besides of the 'Dataset' and 'Dataloader', to train a model, you need to:

1. Construct a model, define the criterion for loss and choose an optimizer.
2. Specify device to train on (you can use GPU in Colab).
3. Implement training code and testing code including saving and loading of models.
4. Report validation accuracy and save the test result.

To set up the model, the torch.nn module is the cornerstone of designing neural networks in PyTorch. This class can be used to implement a layer like a linear layer, a convolutional layer, a pooling layer, an activation function, and also an entire neural network. The nn.Module class has two methods that you have to override.

- **__init__ function**. This function is invoked when you create an instance of the nn.Module. Here you will define the various parameters of a layer. Take the linear layer for example, you can define its input and output dimension, whether the bias is used.
- **forward function**. In the function, you define how your output is computed. This function doesn't need to be explicitly called, and can be run by just calling the nn.Module instance like a layer with the input as it's argument.