Exercises

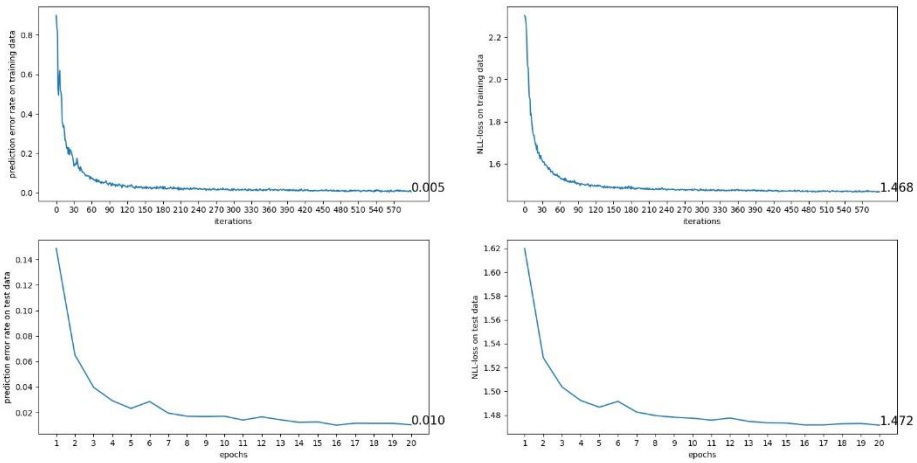
1. **The purpose of pooling** is to reduce the dimensions of the data so that the complexity of calculation and memory usage can be limited. Also, in most of the CNNs we require a sequence of features to represent the input data so that we can use these features for some operations like classification. The pooling layers reduce the number of pixels of each feature maps so that ultimately it will become a single scalar representing a certain feature that we can easily use for further operations. Morever, pooling reduce the complexity of the parameters, limiting the flexibility of the model itself and prevent overfitting to a certain extent. **The stride** is the distance (usually measured by pixels) that a kernel moves on a feature map. The control of strides is significant to achieve a balance between the precision of extraction and complexity of calculation. For example when the stride to set to one, then the extracted feature map is the size of the input, so the calculation requires a lot of time and we will require a pooling layer to reduce dimensions just as what is mentioned above. This is a waste of both time and memory when we don't need to focus on such pixel-level details especially when the size of kernel is fairly large. Whereas, when the stride is large, the size of the extracted feature map will be dramatically reduced so that both time and memory would be saved. That is to say, although we may lose some details intuitively, a single convolutional layer will realize the function of both extracting features and pooling, that is amazing providing precision is not that important.
2. A convolutional kernel is a filter, a feature detector, a group of neurons whose parameters are fixed when it is used on different parts of the previous feature map. After iterations of training, each convolutional kernel represents an extractor a certain feature. In the convolutional layers, the kernels slides across the feature map to detect if there is a feature corresponding to the feature that the kernel represents at a certain coordinate. Because the rule of convolutional calculation and activation function decides that the scalar we obtain would be pretty large when the two features are similar. All in all, convolutional kernels represent **an extractor that detect certain features**.
3. Intuitively, the features learned in early convolutional layers are **much more local** compared with those learned in later layers. In other words, features learned in early layers **contain less information, or information from smaller areas** from the input image. That is because, on a certain feature map, each pixel is calculated by convolution so it contains information from an area that is of same size to the convolutional kernel from the previous feature map. So we can imagine that, as the layer gets deeper, the learned features contain more and more information and detect a larger and larger area from the raw picture. So by the time the features become scalars (We usually obtain a vector comprised of different features finally), it will contain enough information so that we can easily use them to do further operations.
4. a. The dimensionality of the output of this layer is $\frac{n^{l-1}}{s^l} * \frac{n^{l-1}}{s^l} * m^l$

b. The dimensionality a single tensor of weight values is $k^l * k^l * m^{l-1}$, the same size as a kernel. There are m^l tensors in total because there are m^l kernels. The total number of weight values is $k^l * k^l * m^{l-1} * m^l$

c. Since the size of input is $n^{l-1} * n^{l-1} * m^{l-1}$ and the size of output is $\frac{n^{l-1}}{s^l} * \frac{n^{l-1}}{s^l} * m^l$ and there is a weight value between each input scalar and output scalar, the total number of weight values is $\frac{(n^{l-1})^2}{s^l} * \frac{(n^{l-1})^2}{s^l} * m^{l-1} * m^l$

Software Lab

5. The performance is pretty good that after 20 epochs of training, the prediction accuracy on the test set achieves 99.0%. It seems that it encountered no particular trouble.



6. First I tried different hyperparameters like batch sizes, number of epochs and learning rates. It seems that batch size doesn't matter a lot, providing it's not so small, if we do not take whether we train the model on GPU into account. Also, both the prediction loss on training set and test set tend to converge (around 1% error rate) after five epochs and more training attempt only gains marginal accuracy. In terms of learning rate, I tried several times and decided on 0.005 but in fact, if we only take account of the training results, it seems that there nearly no difference between each learning rate around 0.005, for example 0.01 or 0.002. But it is obvious that when the learning rate is higher, the decreasing progress of loss become more unstable, or in other words, the curve become more unsmooth. It is intuitive because we can easily imagine that the parameters would bounce here and there if the learning rate is to high. In general, the differences on the results with different hyperparameters is negligible in a certain range, for nearly all of them can achieve an accuracy of around 99.0%.
   Then I modified the model by **adding a fully connected layer of 64 neurons** for I thought feeding 576 features to ten classes directly is a little bit arbitrary. I also **added a normalization step** in preprocessing because I thought it will do good

to data processing. And it turned out that the finally result does become better. Both the loss and prediction error decrease a little bit more, although the training progress seems to be slower(which can be easily noticed from the charts).