National Chiao Tung University
Spring 2019
Deep Learning
Instructor: Jen-Tsung Chien

Deep Learning HW2 Report

Alfons Hwu
Student ID: 0416324

alfons.cs04@g2.nctu.edu.tw
Dept of Computer Science
Writing with LaTeXon Overleaf

Deep Learning HW2 Report

May 2, 2019

1 Self-designed CNN for image classification

1.1 Preprocessing the images

In the preprocessing part, I merely resize the image to 256 x 256 and perform normalization. No gray scale transformation nor random crop and(or) rotate to form data augmentation.
Reason I only choose resize mainly due to my **hardware limitation** (the GPU of my machine is only GTX1070, too much data will increase my training time or cause **cuda out of memory error**
Second, the main core lies in resizing image to 256 x 256, since suppose we use the image of original size, the dimension of training data will explode (known as **Curse of Dimensionality**).
Another important part is to normalize image on accounting of reducing the effect from extremity and different image scale, now all the images are fitted to the same baseline (normalized and resized).
Both of the training set and testing set are under the same transformation from pytorch for the sake of fairness.

```
my_transform = transforms.Compose([transforms.Resize((256, 256)),
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

1.2 CNN architecture explanation and performance evaluation

1.2.1 Architecture

Main architecture is similar to that of famous VGG16, yet concerning the hardware limitation , I condense them into only 8 convolutional layers and two linear(normal dnn) layer in the end. The following figure show the VGG16 architecture.
With original architecture layer representing in list, M represents the maxpool layer and number is the channel size.
VGG16 = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']

And here is my condensed version

VGG16-small = [8, 'M', 16, 'M', 32, 'M', 64, 'M']

The filter size is reduced to 1/8 in total and for each stacked layer between two maxpool layer, I extract one from them since it is enough to represent the overall architecture, larger kernel size will only increase the computational complexity.

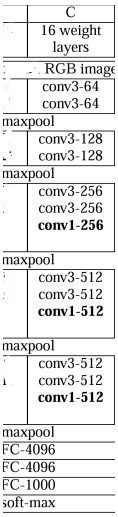
After that, data are fed into the classification layer, the output size of image will changed to 7 x 7 x 64. flatten to the dnn for the final classification.

The following is the architecture of classification layers, ReLU is applied here since it first prevents gradient vanishing, second it provides sparsity with an eye to preventing overfitting (and also reduce computational workload), last but not least, it bears a resemblance to **all or none law** similar to that of nature species. Drop out is used to prevent overfitting and finally the linear output of classification result.

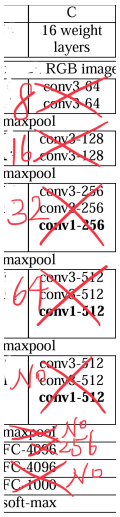
```
self.classifier = nn.Sequential(
                  nn.Linear(64 * 7 * 7, linear_size),
                  nn.ReLU(True),
                  nn.Dropout(),
                  nn.Linear(linear_size, len(classes)),
                  )
```

Compare my CNN with original VGG16,

Actually, the original VGG16 did perform a bit better but both of the test acc get stuck (overfitting), hence the depth size is not the main concern and further tuning will be discussed later(Due to VRAM limitation, original vgg16 can only be fitted with 40 images/batch)



(a) Original VGG16



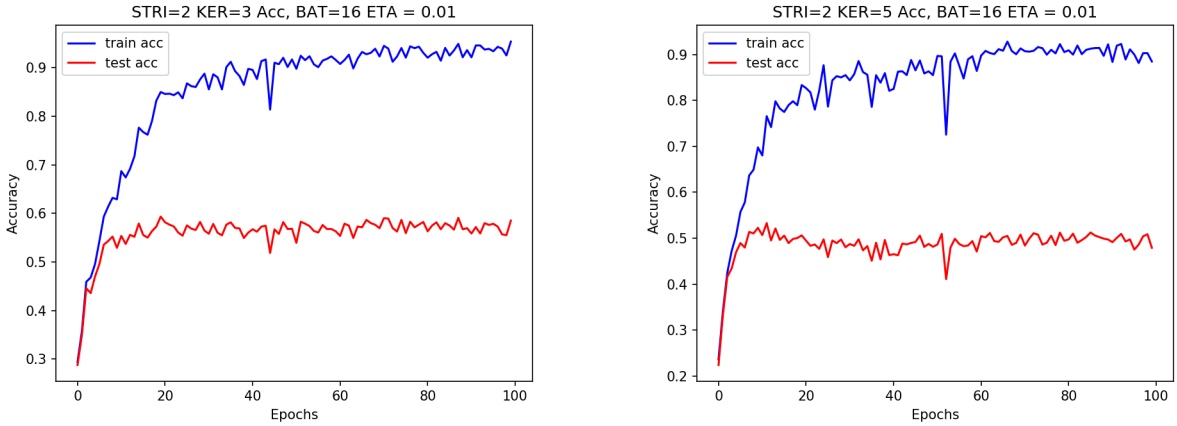
(b) My own VGG

Figure 1: VGG comparison

1.2.2 The learning curve vs kernel and stride size

Experiments to check the effect of kernel and stride size on result

First we compare the effect of kernel size, both of which use the same stride, optimizer, lr and batch size. We are able to clearly see that the small kernel size performed slightly better than large one. In my perspective, larger convolutional kernel probably will mix more pixel around the center pixel, causing more noise in training. (Reference: `https://zhuanlan.zhihu.com/p/41423739`), in this link, author mentioned about "receptive" filed, that is stacked some of the small convolutional layer may not only better preserve the properties of original image(less "mixing" effect) but also cost less parameters during computation.

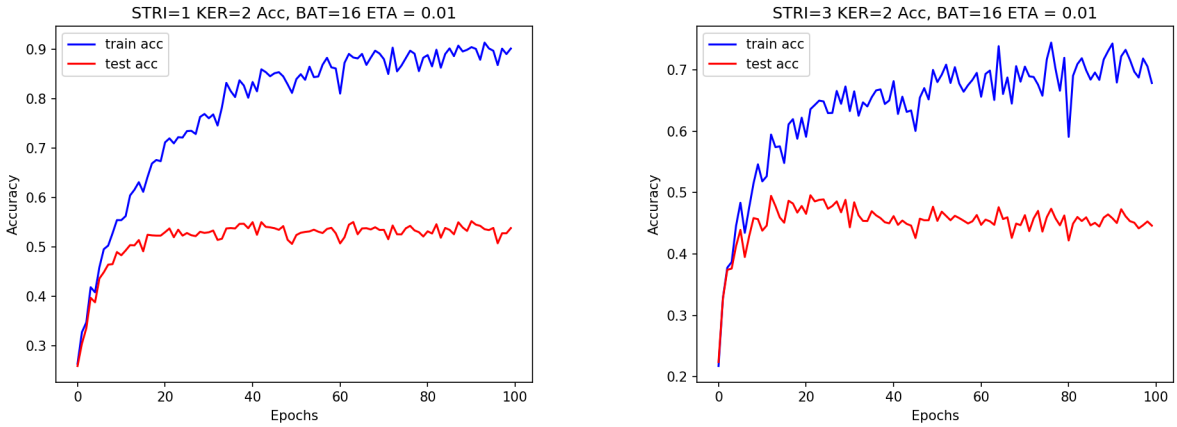


(a) Kernel size 3

(b) Kernel size 5

Figure 2: Convolutional layer kernel size comparison

Then we compare the effect of stride size, both of which use the same kernel size, optimizer, lr and batch size. We can see that the small stride performed slightly better than large one. In my perspective, the higher the stride size is, the less image detail it will be filtered out(or preserved) by the kernel.



(a) Stride size 1

(b) Stride size 3

Figure 3: Convolutional layer stride size comparison

Conclusion

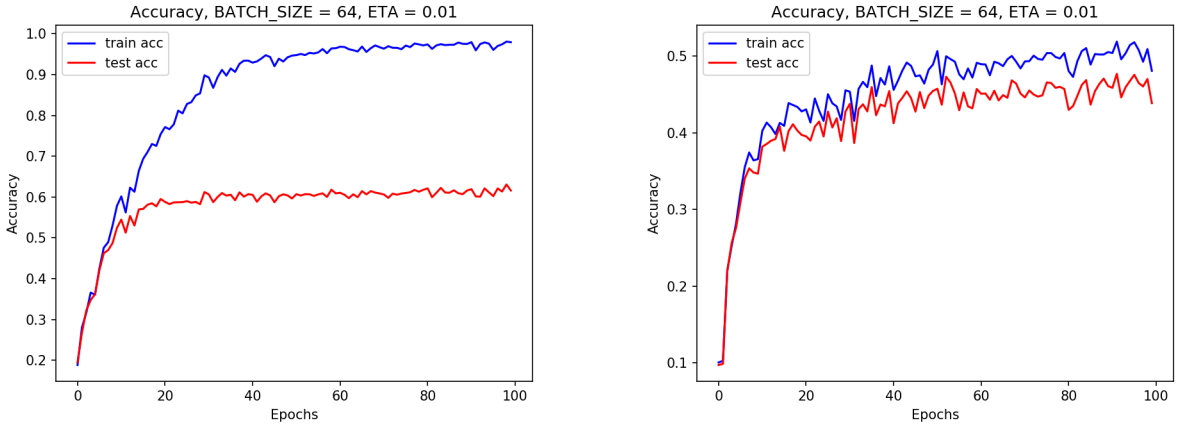
Kernel size 3 and stride 2(no overlapping for maxpool size 2) will be enough.

Experiments to find the best optimizing method

In this project, I have tried using three method to tune my model.

First, by using SGD with momentum and L2 penalty to prevent overfitting

Second, by using adam L2 penalty to prevent overfitting



(a) SGD with momentum and L2 penalty

(b) Adam with L2 penalty

Figure 4: Optimizing method comparison

We can see that the accuracy of testing set in SGD with momentum got stuck in about 30th epoch. Although the curve of adam kept damping, it did not stuck as severe as SGD with momentum.

Conclusion

Extend the epoch to see if one outperformed the other.

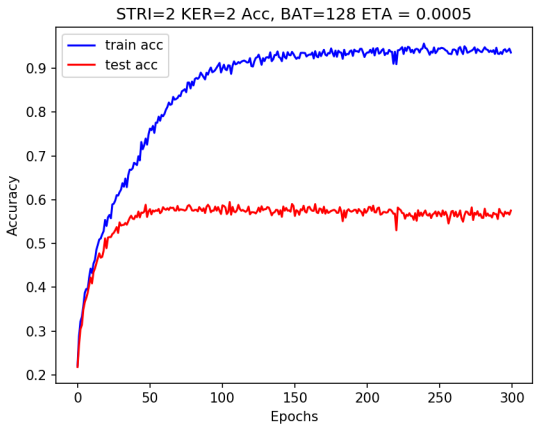
Finally the `batch_size 128, adam optimizer(initial learning rate 5e-4), stride_size 2` performs the best and here is the overall best result.

1.3 The accuracy results and related discussion

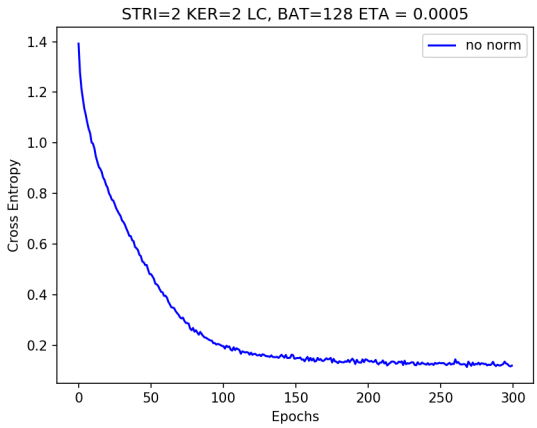
```
Accuracy on  train set of 10000 images is 0.930300
Accuracy on train set of  butterfly class with is 0.973
Accuracy on train set of        cat class with is 0.941
Accuracy on train set of    chicken class with is 1.000
Accuracy on train set of        cow class with is 0.972
Accuracy on train set of        dog class with is 1.000
Accuracy on train set of   elephant class with is 0.882
Accuracy on train set of      horse class with is 1.000
Accuracy on train set of      sheep class with is 0.938
Accuracy on train set of     spider class with is 0.906
Accuracy on train set of   squirrel class with is 1.000

Accuracy on   test set of 4000 images is 0.591000
Accuracy on  test set of  butterfly class with is 0.875
Accuracy on  test set of        cat class with is 0.750
Accuracy on  test set of    chicken class with is 0.500
Accuracy on  test set of        cow class with is 0.417
Accuracy on  test set of        dog class with is 0.500
Accuracy on  test set of   elephant class with is 0.583
Accuracy on  test set of      horse class with is 0.667
Accuracy on  test set of      sheep class with is 0.917
Accuracy on  test set of     spider class with is 0.688
Accuracy on  test set of   squirrel class with is 0.667
```

(a) Accuracy by class



(b) Accuracy curve



(c) Loss curve

Figure 5: Overall best result

Following are the example of incorrectly-classfied images (the filename format are used to represent the incorrectly format: label_predicted_errorcount.png)

(a) Filename format

(b) Label chicken, predict butterfly

(c) Label cow, predict dog

(d) Label dog, predict cat

Figure 6: Incorrectly-classified images

Further discussions (My deductions)

Reason for class `chicken cow dog` perform poor mainly lies in

- Chicken: The side-view of chicken may bear resemblance butterfly.
- Cow: The side-view of cow may bear resemblance to either dog or cat.
- Dog: The front and side view of dog may bear resemblance to cat, since in Taxonomy they both lie under Animalia/Mammalia/Carnivora

2 Self-designed RNN for NLP

2.1 Conventional RNN

2.1.1 Preprocessing

In the preprocessing part, I crop (or padding) the sentence to make each of them in the length of 10 and each word corresponding to the vector length of 10 as well. Finally, each sentence can be represented to the matrix size of 10x10. The preprocessing code is here `https://github.com/Alfons0329/DL_Spring_2019/blob/master/HW2/RNN/preprocessing_2.py`

For example `This is not an apple` in the embedding process, it will be converted to `This is not an apple XXX XXX XXX XXX XXX` since `XXX` is the 0th element in the while dictionary and thus being used for

padding the sentence.

And finally `1 2 3 4 5 0 0 0 0 0` and be converted to matrix representation of size 10 x 10 by the rule of word embedding.

Reference tutorial: `https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html`

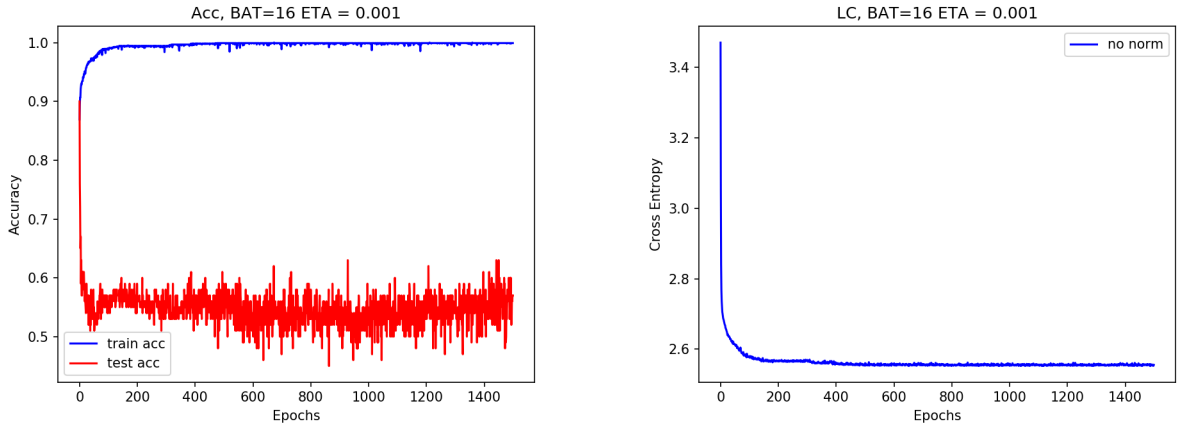
2.1.2 Architecture explanation

The RNN architecture is as follows: The input size is a 10-dimension wordvector and time step is set as 10 as we output the binary classification result based on the sentence of length 10.

```
self.rnn = nn.RNN(
                input_size = 10,
                hidden_size = N_HID_SIZE,
                num_layers = 1,
                dropout = 0.5,
                batch_first = True,
                bidirectional = False
                )
self.classifier = nn.Sequential(
                nn.Linear(N_HID_SIZE, 1),
                nn.Sigmoid(),
                )
```

```
inputs = inputs.view(N_BATCH_SIZE, N_RNN_STEP, N_VEC_WORD) # reshape
```

subsubsectionResults



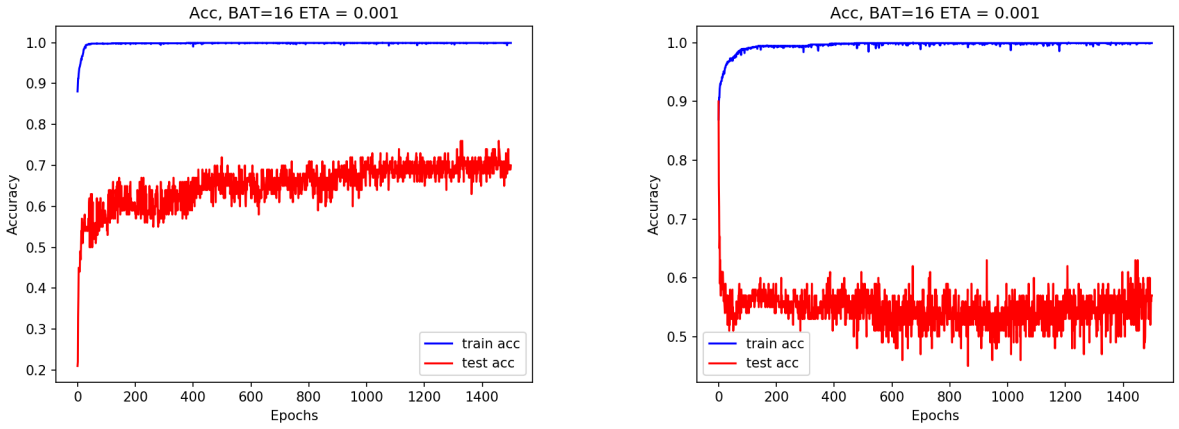
(a) Accuracy curve

(b) Loss curve

Figure 7: RNN results

2.1.3 LSTM results vs RNN results

By merely changed the network from RNN to LSTM (all the rest are remain unchanged, including the optimization method).
Loss curve are about the same, let us check the difference between the accuracy curve.



(a) Accuracy curve of LSTM

(b) Accuracy curve of RNN

Figure 8: Accuracy comparison

As the figures shown, RNN may suffer from the gradient vanishing/exploding in the latter epochs during training.

2.2 Discussion: LSTM vs RNN

Reference `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`
In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

2.2.1 Gradient vanishing and exploding

The back propogation is defined as: $w^{t+1} = wt - \eta \Delta E(w^t)$
The gradients coming from the deeper layers have to go through continuous matrix multiplications because of the the chain rule, and as they approach the earlier layers, if they have small values (<1), they shrink exponentially until they vanish and make it impossible for the model to learn , this is the **vanishing gradient problem**. While on the other hand if they have large values (>1) they get larger and eventually blow up and crash the model, this is the **exploding gradient problem**

Conclusion: The advantage of LSTM

The memory cell remembers the first input as long as the forget gate is open and the input gate is closed while output gate provides finer control to switch the output layer on or off without altering the cell contents, thus solve the gradient vanishing/exploding problem.