2021/22 Frist Term

INM702 Programming and Mathematics for Artificial Intelligence

Report – Task 3

Fashion MNIST Image Classification

Report – Task 4

CIFAR-10 Image Classification

By Alex Collins and Suen Chi Hang

[Alex.Collins@city.ac.uk](mailto:Alex.Collins@city.ac.uk)

[Chi.Suen@city.ac.uk](mailto:Chi.Suen@city.ac.uk)

**Task 3**

**Building the Model**

The goal is to classify Fashion MNIST images by building and using neural networks (NN) with max. test accuracy. Data: <https://www.kaggle.com/zalando-research/fashionmnist>

We built a NN class to run build the neural net model. The model workflow is as follows:

The model is initialised and at that point learning rate, regulization (None, “L1” or “L2”), and regularization alphs (None or float in [0,1]) and random seed are set. The random seed ensures that all output for each NN object is reproduceable. It sets an attribute as np.random.default\_rng(random\_seed) and all distributions are driven by that.

We then add layers using the .add() method. Arguments are nodes and activation function. A NN method is passed as the activation function: options are NN.ReLu, NN.sigmoid, NN.softmax. The last layer has to have nodes = no. of classes and activation NN.softmax. For the final layer we left it as a manual input to keep the code flexible for other tasks.

The final part of preparing the model is to pass the data and other hyper-parameters used to set parameters. The .model() method accepts train and test features and labels, minibatch size (default None to use full dataset), scale used for initializing weights (options are “He” and “Xavier” – default is He as that is better for ReLu activation which we use more), and finally verbose. If verbose is set to True then we print out of NN architecture. .model() initialises the weights and biases matrices as empty matrices of the correct size.

*Fig. 1: example of initializing a model.* Graphical user interface, text, application, email

Description automatically generated

Finally we are ready to fit! We run the .fit() method with the following arguments: epochs – max number of epochs to run; min-epochs – min number of epochs to run regardless of stopping criteria; patience – number of epochs to wait until improvement for stopping; metric – either “train” or “valid” – to choose whether to use training or validation loss as stopping metric; verbose – if True we print out summary information for each epoch.

|  |  |
| --- | --- |
| Text  Description automatically generated | Chart, scatter chart  Description automatically generated |

*Fig. 2: Left - output of .fit(): fitting with early stop. For mini-bath SGD, the model fits well after first epoch but doesn’t improve much afterwards. Right - output of .plot\_error()*

Finally, we built above basic loss visualisation. The .plot\_loss() method shows a chart of each mini-batch loss through time. It’s interesting to see the variation by batch. In the case of our example model, there was not much improvement on an already really good result.

**Tuning Parameters Systematically**

**Number of hidden layers and nodes**

First, we need to decide the basic structure for tuning, i.e. number of layers. Then we build helper function based on the determined basic structure. As in earlier trials sigmoid activation performs poorly (see supplementary Fig. 1), we simply keep using ReLu activation. We will add hidden layer(s) of 1024 nodes with ReLu activation.

1: 

2: 

3: 

*Fig. 3 Results of final epoch by no. of hidden layers*

So let’s tune the no. of nodes for 2 hidden layers, which has highest test accuracy of 87.4%.

*Fig. 4* Table

Description automatically generated

2-hidden-layer NN with n1=2048, n2=256 has highest accuracy of 88.4%. Note that for n1=2048, accuracy seems to decrease with n2 beyond 256, which may justify us to stop the expensive looping thereafter. Yet, as 2-hidden-layer is much slower than 1-hidden-layer using python built functions for NN, we will also try different no. of nodes for 1-hidden-layer.

|  |  |
| --- | --- |
| Text  Description automatically generated with low confidence | *Fig. 5*  The best for 1-hidden-layer is 1024 nodes, with accuracy of 87.28%. As it is close to the optimal of 2-hidden-layer NN and beat most of them, we choose the 1-hidden-layer NN of 1024 nodes after computation time is also considered. |

### **Learning rate and batch size**

As learning rate and batch size directly affects update of parameters, we tune them together.

|  |  |
| --- | --- |
| Table  Description automatically generated  *Fig. 6a* Best learning rate 0.05, batch size 16  Larger batch size tends to go with higher learning rate to maximize accuracy. This is because for smaller batch size, the loss is more volatile and it is more suitable to use smaller learning rate for smaller batch size. | Chart, line chart  Description automatically generated  *Fig. 6b* |

**Regularization**

We built two types of regularization: L1, L2, where L1 uses the sum of absolute weight as penalty added to loss, while L2 uses the sum of squares of weights as penalty, each multiplied by lambda, which acts as extent of regularization. The rationale is to prevent weights from being too big. Usually L1 forces many weights to zero and L2 forces weights to move towards zero which helps prevent overfitting). We will tune extent of regularization.

Text

Description automatically generatedText

Description automatically generated

*Fig. 7 L1 and L2 show no improvement.*

**Stopping Criteria**

Stopping criteria is used for saving computation time and more importantly, also avoids over-fitting which occurs with excessive training. We design the NN to stop when there is no significant improvement (default threshold 2.5%) of loss compared with previous p epochs, where p is patience in our code. We allow setting min and max no. of epochs, and whether to use training or validation loss for monitoring improvement. Results: test accuracy is unchanged at 88.76% for patience 1 to 9. It also remains unchanged at 86.23% for different improvement threshold for short patience of 2. (see supplementary Fig. 2 & 3)

**Conclusion**

|  |  |
| --- | --- |
| The optimal neural network structure is 2-layer(1024/10) and optimal parameters are: learning rate 0.05, batch size 16, ReLu activation for hidden layers Softmax activation for output layer, no regularization L1 or L2. As shown by loss curve, there is some degree of over-fitting, though regularization L1, L2 and stopping criteria don’t help. The test accuracy is 88.76%. | Chart, line chart  Description automatically generated  *Fig. 8 Loss curve* |

Code: <https://github.com/alexxcollins/AI_INM702_collab/tree/master/task3>

**Task 4**

**Implement a Neural Net**

Before we even started training neural nets, it is important to have a good benchmark to judge results against. Picking randomly would achieve accuracy of 10%, but can we do better than that? We decided to compute the “average” images of each class in the training data: simply an tensor of size (32, 32, 3) with mean values of all class pixel values. To classify test data, we calculated the mean square distance from each image RGB pixel value to our 10 “average” images. We picked the minimum distance as our classification.

|  |  |  |
| --- | --- | --- |
|  | The benchmark model achieved an accuracy of 28% on the test set. It correctly identified this frog as a frog. In fact, it predicted frog or airplain 45% of the time so everything looks like a frog or an airplane! |  |
| *Test image: frog* |  | *“Average” frog* |

Text

Description automatically generatedTo implement the Neural Net (and all CNNs afterwards), we created a train and test function, and used DataLoader objects to manage the minibatches. (Lab code and PyTorch documentation helped us build these.) We built a three layer NN, with activation function ReLu. The architecture is to the right. 40% accuracy was very disappointing: only a bit better than our benchmark. An advantage of this model is that over-fitting wasn’t a problem as early as many of the CNNs we trained.

**Improvements**

Convolutional Neural Network (CNN)

CNN is a popular technique for image classification. Unlike traditional fully connected (FC) NN, CNN shares same weights for each pixel and somehow retains the spatial relationship among features (i.e. pixels) of images. We will keep the previous NN layers in the end, but add CNN layers in the beginning. As the image is small, we will use kernel size 3x3, stride 1, max pooling (2, 2), with suitable padding generating same output shape.

Text

Description automatically generated Chart, line chart

Description automatically generated

Fig. 1 First CNN coding structure and results

The test accuracy increases significantly from 40.87% to 56.70%.

**Number of layers**

To decide the basic structure, try adding convolutional layers and cutting FC layers.

|  |  |
| --- | --- |
| 1 CNN, 3 FC: 56.70%  2 CNN, 3 FC: 64.77%  1 CNN, 2 FC: 60.66%  1 CNN, 1 FC: 56.10%  2 CNN, 2 FC: 67.62% 🡪best accuracy, see Fig. in right  \* The dimension of output of CNN layers can be kept by adjusting no. of CNN channels to fit to FC layers. | Chart, line chart  Description automatically generated |

**Dropout**

From above charts, validation loss isn’t following training loss to decrease, implying over-fitting. We try to overcome it by dropout, which is randomly dropping certain portions of nodes to avoid co-adaptation of weights and enhance generalization.

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

2CNN, 3FC, dropout 0.2 3CNN, 3FC, dropout 0.2 3CNN, 3FC, dropout 0.4

3CNN, 3FC, dropout 0.4 has highest test accuracy 70.75%, and the chart validation loss curve is much closer to training loss, implying over-fitting is much reduced.

Number of parameters

We can tune no. of parameters by no. of channels of CNN layers and no. of nodes of FC layers. Denote k as no. of channels of first CNN layer, and n as no. of nodes of first FC layer. Set 4k, 16k as no. of channels of second, third CNN layer respectively to keep dimension.

|  |  |
| --- | --- |
| Text  Description automatically generated | The model k=24, n=512 has highest test accuracy: 72.51% |

**Learning rate and dropout**

Dropout is first proposed by Hinton G., in which he suggested learning rate be adjusted appropriately with the use of dropout. Hence, we now tune them together.

|  |  |
| --- | --- |
| Text  Description automatically generated | The model lr=0.2, p=0.4 has highest test accuracy: 69.32%  \*lr: learning rate; p: dropout rate |

**Conclusion**

So after above tuning, the optimal model is: (k=24, n=512), i.e. 3 convolutional layers of 24/96/384 channels (each with kernels size 3, stride 1, padding 1, with Relu and Max Pool(2,2) ), followed by 2 hidden fully connected layers (each 512 nodes, RELU) and 1 output layer (10 classes) using Pytorch SGD optimizer with batch size 100, 10 epochs and cross-entropy loss function, at learning rate 0.2 and dropout rate 0.4.

To verify robustness against randomness, we had 2 new trainings, see supplementary fig. 1.

Average test accuracy among 3 trainings is 72.9%, +/-0.9% within 95% confidence interval.

Disclaimer: we cannot guarantee it has similar accuracy in every random state, though we have tested multiple random states for initialization of weights and training. Please note that according to PyTorch, completely reproducible results are not guaranteed across PyTorch releases, individual commits, different platforms, or between CPU and GPU executions. (<https://pytorch.org/docs/stable/notes/randomness.html>).

**Code:** <https://github.com/alexxcollins/AI_INM702_collab/tree/master/task4>

**Reflection**

Task 3: Suen firstly built basic neural network with both forward and backward propagation, with *He* and *Xavier* initialization, sigmoid and softmax activation, and parameterizable to change number of nodes and layers in NN.py. Collins improved by adding ReLu, optimizer, stopping criteria and visualization such as scatter plot of losses in new version NN2.py, which is re-built in new style. Upon the work and many trials of Collins, Suen trained and tuned the neural networks systematically, and evaluated hyperparameter performance, tailor-made charts analyzing results and selected the optimized model. Collins wrote *Building the Model* and Suen wrote *Tuning Parameters Systematically* to *Conclusion* in report.

The code in task 4 was written with the help of lab lectures (Riaz, A. 2021) and PyTorch documentation. The first section on creating a benchmark was inspired by the fast.ai MOOC (Howard and Gugger, 2020). Collins set the notebook up, did the benchmark model, wrote the training and testing functions and the first neural net. Suen led the CNN development and parameter testing. Collins wrote *Implement a neural net* and Suen wrote *Implement CNN* to *Conclusion.*

For the project, each reviewed and amended the work of each other and contributed equally 50%. We were both very involved with all parts of the project: whether leading them or not.

**Task 3 - Supplementary Figures**

Graphical user interface, text, application, email

Description automatically generated Table

Description automatically generated

S-Fig. 1 Poor performance of Sigmoid S-Fig. 2 Tuning patience of stopping criteria

Text

Description automatically generated

S-Fig. 3 Tuning improvement threshold

**Task 4 - Supplementary Figures**

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

S-Fig. 1 New trainings of final model

A close-up of a document

Description automatically generated with medium confidence

S-Fig 2. PyTorch print out of final model. We applied dropout at every layer in the Linear stage.

**References**

# Bendersky, E, 2016, “The softmax function and its derivative”, *Eli Bendersky’s website*. Available at <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>. Accessed Dec 2021.

Dellinger J, 2019, towardsdatascience.com, accessed Dec 2021, < <https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79>>

# Howard, J, Gugger, S, 2020, “Chapter 4: MNIST Basics”, *Deep Learning for Coders with fastai and PyTorch: AI applications without a PhD*. Available at <https://github.com/fastai/fastbook/blob/master/04_mnist_basics.ipynb>. Accessed Nov 2021.

# Kerbiel, T, 2021, “Derivative of the Softmax Function and the Categorical Cross-Entropy Loss”*, Towards Data Science*. Available at <https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1>. Accessed Dec 2021.

Ng A., 2018, DeepLearning.AI, accessed 2018, < <https://www.coursera.org/specializations/deep-learning>>

# Oppermann, A, 2020, “Regulaization in Deep Learning, L1, L2 and Dropout”, *Towards Data Science.* Available at: <https://towardsdatascience.com/regularization-in-deep-learning-l1-l2-and-dropout-377e75acc036>. Accessed Dec 2021.

Riaz A., 2021, *Lecture 6: Neural Networks, Lecture 7: Regularisation and Optimization, Lecture 8: Pytorch, Lecture 9: CNN & RNN,* lecture notes, Programming and Mathematics for Artificial Intelligence INM702, City, University of London, delivered 23 Nov 2021, 30 Nov 2021, 7 Dec 2021

StackOverflow, editors/developers, 2021. Available at <https://stackoverflow.com>. Accessed Dec 2021.

PyTorch editors/developers, © 2021, PyTorch, accessed Dec 2021, <<https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html>>

PyTorch editors/developers, © 2021, PyTorch, accessed Dec 2021, <<https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module>>

Stanford University, 2021, CS231n: Convolutional Neural Networks for Visual Recognition, accessed Dec 2021, <<https://cs231n.github.io/convolutional-networks/>>