

# The All Convolutional Net for CIFAR10 and Transfer Learning on CIFAR100

Yingying Ma  
SIST

mayy1@shanghaitech.edu.cn

## Abstract

*In the paper, we implement All Convolutional Network in and other 5 network structures on which it based. We compare performance of these six models and found out ConvPool-CNN-C performs better than other models instead of All Convolutional Net as expected.*

*Besides, we finish transfer learning on subset class1 and class 2 of CIFAR-100 datasets.*

## 1. Introduction

Most modern convolutional neural networks (CNNs) used for object recognition are built using the same principles: Alternating convolution and max-pooling layers followed by a small number of fully connected layers. In Paper [1], the authors discovered that max-pooling could simply be replaced by a convolutional layer with increased stride without loss in accuracy on several image recognition benchmarks. For instance, max-pooling with stride 2 can be replaced by another convolutional layer with stride 2.

In the paper, we construct three basic CNN models and three advanced models, in which Strided-CNN-C and ALL-CNN-C replace max-pooling layer with convolution layer with stride 2. We compare performance of these six models and found out ConvPool-CNN-C is the best model to classify CIFAR10 datasets.

## 2. Model description

### 2.1. Basic models

Model A is a traditional model for CNN with convolutional layer following by max-pooling layer except for the last two convolutional layers with the  $1 \times 1$  kernel, which is used to replace the fully convolution layer. If the image area covered by units in the topmost convolutional layer covers a portion of the image large enough to recognize its content.

### 2.2. Advanced models

The Advanced models are shown in 2 The higher layers are the same as in Table 1. ConvPool-CNN-C does not use

Model		
A	B	C
Input $32 \times 32$ RGB image		
$5 \times 5$ conv. 96 ReLU	$5 \times 5$ conv. 96 ReLU $1 \times 1$ conv. 96 ReLU	$3 \times 3$ conv. 96 ReLU $3 \times 3$ conv. 96 ReLU
$3 \times 3$ max-pooling stride 2		
$5 \times 5$ conv. 192 ReLU	$5 \times 5$ conv. 192 ReLU $1 \times 1$ conv. 192 ReLU	$3 \times 3$ conv. 192 ReLU $3 \times 3$ conv. 192 ReLU
$3 \times 3$ max-pooling stride 2		
$3 \times 3$ conv. 192 ReLU		
$1 \times 1$ conv. 192 ReLU		
$1 \times 1$ conv. 10 ReLU		
global averaging over $6 \times 6$ spatial dimensions		
10 or 100-way softmax		

Figure 1. Three basic models

Model		
Strided-CNN-C	ConvPool-CNN-C	ALL-CNN-C
Input $32 \times 32$ RGB image		
$3 \times 3$ conv. 96 ReLU $3 \times 3$ conv. 96 ReLU with stride $r = 2$	$3 \times 3$ conv. 96 ReLU $3 \times 3$ conv. 96 ReLU $3 \times 3$ conv. 96 ReLU	$3 \times 3$ conv. 96 ReLU $3 \times 3$ conv. 96 ReLU
$3 \times 3$ max-pooling stride 2		
$3 \times 3$ conv. 192 ReLU $3 \times 3$ conv. 192 ReLU with stride $r = 2$	$3 \times 3$ conv. 192 ReLU $3 \times 3$ conv. 192 ReLU $3 \times 3$ conv. 192 ReLU	$3 \times 3$ conv. 192 ReLU $3 \times 3$ conv. 192 ReLU
$3 \times 3$ max-pooling stride 2		
$3 \times 3$ conv. 192 ReLU with stride $r = 2$		

⋮

Figure 2. Three advanced models

convolution layer with bigger stride. Strided-CNN-C and ALL-CNN-C is new models brought up by [1]. ALL-CNN-C add another convolution layer compared with Strided-CNN-C.

## 3. Transfer learning

In this part we do classification task on subset class1 and class2 of CIFAR-100 datasets. We save the best ALL-CNN-C model and try to use it as a feature extractor. Replace and retrain the final FC classifier(which is the  $1 \times 1$  conv layer), while fine-tuning the parameters of other layers.

## 4. Experiments

### 4.1. Experiment environment

Our sever is based on Ubuntu 16.04 and has 2 GTX 1080Ti GPU. Pytorch is used for training deep learning net-

work. In order to speed up the training progress, we installed cuda9.0. Other than that, we install some packages like PIL for image loading.

## 4.2. Experiment setup

All networks were trained using stochastic gradient descent with fixed momentum of 0.9.

In classification task, we set original learning rate to be 0.01, weight decay to be 0.001 and train the network for 350 epochs. In epoch [200, 250, 300], we multiply 0.1 to the learning rate. The dropout layer is added and the probabilities were 20% for dropping out inputs and 50% otherwise.

In Transfer learning task, first we construct our own class of CIFAR100 to load the given data and transform the data into dataloader in torch. Then we load the best model parameters and set the weight of the last layer to be random and the bias to be 0. We use a list of learning rate of [0.0025, 0.001, 0.0005] to re-train the net. The total epoch number is 100, weight decay is also set 0.001.

## 4.3. Classification results

In order that we could compare the performance of each network, we list the validation accuracy of each model in the table1. As we can see ALL-CNN-C is the second best model. ConvPool-CNN-C with convolutional max-pooling layer performs better than ALL-CNN-C.

During our training, the three advanced model is easy to stuck in the local-minimal and hard to converge compared with the three basic cnn models, especially ALL-CNN-C.

Model	Accuracy
Model A	0.864
Model B	0.844
Model C	0.869
Strided-CNN-C	0.864
ConvPool-CNN-C	0.891
ALL-CNN-C	0.869

Table 1. Accuracy Comparison

## 4.4. Transfer learning

We could see that transfer learning is much more better than random initializing the parameters and less likely to stuck in the local-minimal point.

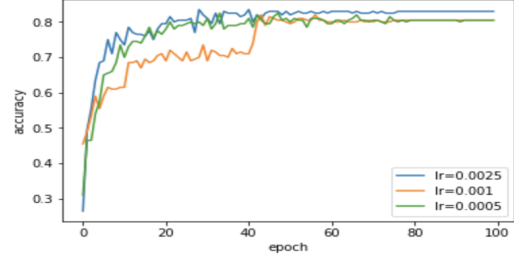


Figure 3. Transfer learning Class1 Accuracy

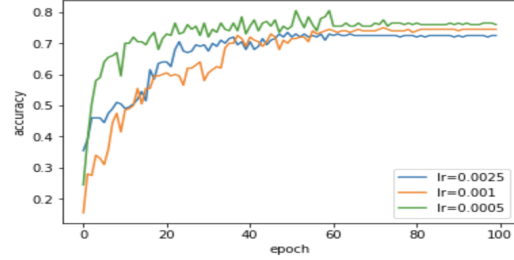


Figure 4. Transfer learning Class2 Accuracy

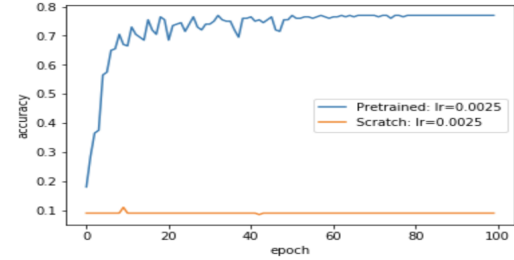


Figure 5. Class1 Transfer vs. Scratch learning Accuracy

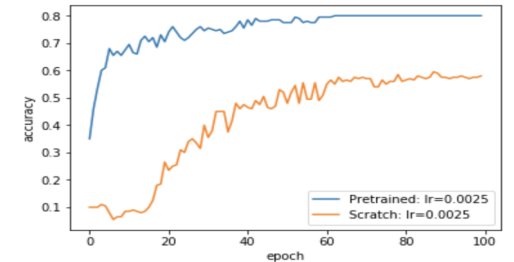


Figure 6. Class2 Transfer vs. Scratch learning Accuracy

## 5. Conclusion

In the paper, we construct the models that [1] mentioned and find that the new model is not robust and does not perform better than ConvPool-CNN-C with conventional max-pooling layer. So the function of CNN with higher stride needs to be discussed later. Transfer learning is a great trick to initialize network parameters. The network with transfer learning is more likely to converge and converge faster.

## 6.1. Appendix 1

Each ipynb file in the code structure construct one model. Each file consists of basic load data, class Model, train func-

File Name	Size	Time
ALL_CNN_C.py	11 hours	
Base CNN - model A-Copy1.py	6 hours	
Base CNN - model A.py	18 hours	
Base CNN - model B.py	18 hours	
Base CNN - model C.py	18 hours	
ConvPool CNN-C.py	Running 17 hours	
Enriched CNN-C.py	18 hours	
transfer_learning_class1.py	Running 25 minutes	
transfer_learning_class2.py	Running 25 minutes	
model_a.log	18 hours	
model_all_cnn_c.log	11 hours	
model_b.log	18 hours	
model_c.log	18 hours	
model_convpool_cnn_c.log	17 hours	
model_enriched_cnn_c.log	18 hours	
tf_class1.log	an hour	
tf_class1_ac.log	25 minutes	
tf_class2.log	an hour	
tf_class2_ac.log	an hour	

Figure 7. Code Structure

tion. The flow is shown as following:

```
[ 21] NUM_TRAIN = 47000

transform = Transform.Compose([
    transforms.Randomize(),
    transforms.Normalize((0.4914, 0.4822, 0.4464), (0.2023, 0.1994, 0.2010))
])

cifar10_train = torchvision.datasets.CIFAR100(root='./data', train=True,
                                              transform=transform, download=True)
loader_train = torch.utils.data.DataLoader(cifar10_train, batch_size=128,
                                             num_workers=4, shuffle=True)
cifar10_val = torchvision.datasets.CIFAR100(root='./data', train=False,
                                             transform=transform, download=True)
loader_val = torch.utils.data.DataLoader(cifar10_val, batch_size=128,
                                          num_workers=4, shuffle=True)

cifar10_test = torchvision.datasets.CIFAR100(root='./data', train=False,
                                              transform=None, download=True)
loader_test = torch.utils.data.DataLoader(loader_train.loader, batch_size=1)

Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified

In [3]: train_images, train_labels = loader_train.next()
print('train', train_images.shape, train_labels.shape)
val_images, val_labels = loader_val.next()
print('val', val_images.shape, val_labels.shape)
test_images, test_labels = loader_test.next()
print('test', test_images.shape, test_labels.shape)

train_xi = torch.index(train_images[4, 32, 321], train_y=torch.index(
val_xi = torch.index(val_images[4, 32, 321], val_y=torch.index(
test_xi = torch.index(test_images[4, 32, 321], test_y=torch.index(
```

Figure 8. 1\_Load Data

```

class Model(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2,
                 channel_3, channel_4, num_classes):
        """
        Default initialization:
        model = Model(1, channel=3, channel_1=96,
                     channel_2=192, channel_4=32, num_classes=10)
        """
        super().__init__()
        self.conv1 = nn.Conv2d(in_channel, channel_1, 3, padding=1) # (3, 96, 3)
        self.conv2 = nn.Conv2d(channel_1, channel_1, 3, padding=1) # (3, 96, 3)
        self.conv3 = nn.Conv2d(channel_1, channel_1, 3, padding=1, stride=2) # (3, 96, 3)
        self.conv2 = nn.Conv2d(channel_1, channel_2, 3, padding=1) # (3, 192, 3)
        self.conv3 = nn.Conv2d(channel_2, channel_2, 3, padding=1, stride=2) # (3, 192, 3)
        self.conv4 = nn.Conv2d(channel_2, channel_3, 3, padding=1) # (3, 192, 3)
        self.conv5 = nn.Conv2d(channel_3, num_classes, 1) # (192, 10, 1)

    def forward(self, x):
        # shape: (4, 32, 32, 3)
        x = x.shape[0]
        # N = shape[0]
        x_drop = F.dropout(x, 0.5) # (4, 32, 32, 3)
        layer_out = F.relu(self.conv1(x_drop)) # (4, 96, 32, 32)
        layer_out = F.relu(self.conv2(layer_out)) # (4, 96, 32, 32)
        layer_out_drop = F.dropout(layer_out, 0.5) # (4, 96, 32, 32)
        layer_out = F.relu(self.conv3(layer_out_drop)) # (4, 192, 16, 16)
        layer_out_drop = F.dropout(layer_out, 0.5) # (4, 192, 16, 16)
        layer_out = F.relu(self.conv4(layer_out_drop)) # (4, 192, 16, 16)
        layer_out_drop = F.dropout(layer_out, 0.5) # (4, 192, 16, 16)
        layer_out = F.relu(self.conv5(layer_out_drop)) # (4, 10, 1, 1)
        return layer_out

#
def test_model(device, model):
    layer_out = layer_out.cpu().detach().numpy()
    layer_out = layer_out.cpu().detach().numpy()
    layer_out = layer_out.cpu().detach().numpy()
    return layer_out

#

```

## Figure 9. 2\_Classifier

```

14 [In]: def test_model(device, model):
15     if isinstance(device, str):
16         print('Checking accuracy on validation set')
17         # print('Checking accuracy on test set')
18         num_correct = 0
19         num_samples = 0
20         for i in range(10):
21             with torch.no_grad():
22                 for x, y in loader:
23                     x = x.to(device=device, dtype=torch.float) # move to device, e.g. GPU
24                     y = y.to(device=device, dtype=torch.long)
25                     scores = model(x)
26                     scores = scores.detach().cpu().numpy()
27                     num_correct += np.sum(scores == y.numpy())
28                     num_samples += scores.size()[0]
29         print('Test set: %d %d correct (%.2f) %d num. current, num. samples, 100 = acc)'
30         return acc

17 [In]: def plot_loss, y_hat, path, name, labels, values):
31     plt.plot(loss, y_hat)
32     plt.xlabel('loss')
33     plt.ylabel('y_hat')
34     plt.title(name)
35     plt.show()

```

Figure 10.3: Utility functions

```

to [x] print: every = 100

def train(model, optimizer, scheduler, weight_decay=0.01, epochs=1, best_acc=0.):
    loss_list = []
    epoch_list = []
    acc_list = []

    model = model.to(device=device)
    m = m.to(device=device, device_id=-1)

    for e in range(epochs):
        scheduler.step()
        for step, (x, y) in enumerate(loader_train):
            model.train()
            x = x.to(device=device, device_id=-1)
            y = y.to(device=device, device_id=-1)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            optimizer.zero_grad()
            loss.backward()

            optimizer.step()

        if step % every == 0:
            print(f'Epoch {e}, iteration {0}, loss = {loss.item():.4} | w, step, loss(item)')
            acc = torch.argmax(torch.max(scores, dim=-1), dim=-1).item()
            if acc > best_acc:
                best_acc = acc
            best_model = copy.deepcopy(model.state_dict())
            # format: loss, all_acc, acc, step, iteration, w, step, loss(item), accuracy, F1 score
            # format: (step, loss(item), acc)
            # format: (step, loss(item), acc)

        # format: (step, loss(item), acc)
        # format: (step, loss(item), acc)

    epoch_list.append(e)
    loss_list.append(loss.item())
    acc_list.append(acc)

    plot(epoch_list, loss_list, path='../results/figure', name='Model All CHN C_loss accuracy Epoch',
         loss_list, acc_list, path='../results/figure', name='Model All CHN C_accuracy accuracy Epoch',
         label='epoch', ylabel='loss')

    print('Best accuracy:', best_acc)
    print('Model All CHN C_loss:', loss_list[-1])
    # format: loss, all_acc, acc, step, iteration, w, step, loss(item), accuracy, F1 score
    # format: loss, all_acc, acc, step, iteration, w, step, loss(item), accuracy, F1 score
    model.load_state_dict(best_model.state_dict())
    return model

```

Figure 11. 4\_train function

[illegible]

Figure 12. 5\_train model

```
In [11]: PATH = "../best_model/"
model_name = "best_model_all_cnn_e.pt"
if not os.path.exists(PATH):
    os.mkdir(PATH)
torch.save(best_model.state_dict(), PATH + model_name)
```

Figure 13. 6\_save the best model

## 6.2. Appendix 2

Appendix 2 includes the screen shot of the evaluation and training log. The last line is the validation accurate for each model From the log file we could see that all model converge. But during our training, the three advanced model is hard to converge compared with the three basic cnn models.

2788	Epoch: 348	Step: 0	loss: 0.0118	accuracy: 0.85
2789	Epoch: 348	Step: 100	loss: 0.0175	accuracy: 0.85
2790	Epoch: 348	Step: 200	loss: 0.0151	accuracy: 0.85
2791	Epoch: 348	Step: 300	loss: 0.0244	accuracy: 0.85
2792	Epoch: 348	Step: 400	loss: 0.0217	accuracy: 0.85
2793	Epoch: 348	Step: 500	loss: 0.0229	accuracy: 0.85
2794	Epoch: 348	Step: 600	loss: 0.0113	accuracy: 0.85
2795	Epoch: 348	Step: 700	loss: 0.0218	accuracy: 0.85
2796	Epoch: 349	Step: 0	loss: 0.0195	accuracy: 0.85
2797	Epoch: 349	Step: 100	loss: 0.0211	accuracy: 0.85
2798	Epoch: 349	Step: 200	loss: 0.0155	accuracy: 0.85
2799	Epoch: 349	Step: 300	loss: 0.0157	accuracy: 0.85
2800	Epoch: 349	Step: 400	loss: 0.0190	accuracy: 0.85
2801	Epoch: 349	Step: 500	loss: 0.0247	accuracy: 0.85
2802	Epoch: 349	Step: 600	loss: 0.0211	accuracy: 0.85
2803	Epoch: 349	Step: 700	loss: 0.0188	accuracy: 0.85
2804	Best accuracy: 0.864			

Figure 14. Model A Log

Figure 10. 3\_Utility functions

```

2788 Epoch: 348→ Step: 0→ loss: 0.0100→ accuracy: 0.84
2789 Epoch: 348→ Step: 100→ loss: 0.0141→ accuracy: 0.84
2790 Epoch: 348→ Step: 200→ loss: 0.0062→ accuracy: 0.84
2791 Epoch: 348→ Step: 300→ loss: 0.0197→ accuracy: 0.84
2792 Epoch: 348→ Step: 400→ loss: 0.0147→ accuracy: 0.84
2793 Epoch: 348→ Step: 500→ loss: 0.0104→ accuracy: 0.84
2794 Epoch: 348→ Step: 600→ loss: 0.0172→ accuracy: 0.84
2795 Epoch: 348→ Step: 700→ loss: 0.0134→ accuracy: 0.84
2796 Epoch: 349→ Step: 0→ loss: 0.0127→ accuracy: 0.84
2797 Epoch: 349→ Step: 100→ loss: 0.0172→ accuracy: 0.84
2798 Epoch: 349→ Step: 200→ loss: 0.0097→ accuracy: 0.84
2799 Epoch: 349→ Step: 300→ loss: 0.0113→ accuracy: 0.84
2800 Epoch: 349→ Step: 400→ loss: 0.0170→ accuracy: 0.84
2801 Epoch: 349→ Step: 500→ loss: 0.0082→ accuracy: 0.84
2802 Epoch: 349→ Step: 600→ loss: 0.0157→ accuracy: 0.84
2803 Epoch: 349→ Step: 700→ loss: 0.0135→ accuracy: 0.84
2804 Best accuracy: 0.844

```

Figure 15. Model B Log

```

2788 Epoch: 348→ Step: 0→ loss: 0.0070→ accuracy: 0.87
2789 Epoch: 348→ Step: 100→ loss: 0.0062→ accuracy: 0.87
2790 Epoch: 348→ Step: 200→ loss: 0.0072→ accuracy: 0.87
2791 Epoch: 348→ Step: 300→ loss: 0.0099→ accuracy: 0.87
2792 Epoch: 348→ Step: 400→ loss: 0.0091→ accuracy: 0.87
2793 Epoch: 348→ Step: 500→ loss: 0.0065→ accuracy: 0.87
2794 Epoch: 348→ Step: 600→ loss: 0.0053→ accuracy: 0.87
2795 Epoch: 348→ Step: 700→ loss: 0.0111→ accuracy: 0.87
2796 Epoch: 349→ Step: 0→ loss: 0.0064→ accuracy: 0.87
2797 Epoch: 349→ Step: 100→ loss: 0.0057→ accuracy: 0.87
2798 Epoch: 349→ Step: 200→ loss: 0.0075→ accuracy: 0.87
2799 Epoch: 349→ Step: 300→ loss: 0.0076→ accuracy: 0.87
2800 Epoch: 349→ Step: 400→ loss: 0.0088→ accuracy: 0.87
2801 Epoch: 349→ Step: 500→ loss: 0.0098→ accuracy: 0.87
2802 Epoch: 349→ Step: 600→ loss: 0.0066→ accuracy: 0.87
2803 Epoch: 349→ Step: 700→ loss: 0.0082→ accuracy: 0.87
2804 Best accuracy: 0.877

```

Figure 16. Model C Log

```

2788 Epoch: 348→ Step: 0→ loss: 0.0139→ accuracy: 0.85
2789 Epoch: 348→ Step: 100→ loss: 0.0102→ accuracy: 0.85
2790 Epoch: 348→ Step: 200→ loss: 0.0126→ accuracy: 0.85
2791 Epoch: 348→ Step: 300→ loss: 0.0165→ accuracy: 0.85
2792 Epoch: 348→ Step: 400→ loss: 0.0115→ accuracy: 0.85
2793 Epoch: 348→ Step: 500→ loss: 0.0152→ accuracy: 0.85
2794 Epoch: 348→ Step: 600→ loss: 0.0117→ accuracy: 0.85
2795 Epoch: 348→ Step: 700→ loss: 0.0195→ accuracy: 0.85
2796 Epoch: 349→ Step: 0→ loss: 0.0138→ accuracy: 0.85
2797 Epoch: 349→ Step: 100→ loss: 0.0105→ accuracy: 0.85
2798 Epoch: 349→ Step: 200→ loss: 0.0197→ accuracy: 0.85
2799 Epoch: 349→ Step: 300→ loss: 0.0127→ accuracy: 0.85
2800 Epoch: 349→ Step: 400→ loss: 0.0172→ accuracy: 0.85
2801 Epoch: 349→ Step: 500→ loss: 0.0251→ accuracy: 0.85
2802 Epoch: 349→ Step: 600→ loss: 0.0120→ accuracy: 0.85
2803 Epoch: 349→ Step: 700→ loss: 0.0209→ accuracy: 0.85
2804 Best accuracy: 0.864

```

Figure 17. Model Strided-CNN-C Log

```

2788 Epoch: 348→ Step: 0→ loss: 0.0061→ accuracy: 0.88
2789 Epoch: 348→ Step: 100→ loss: 0.0065→ accuracy: 0.88
2790 Epoch: 348→ Step: 200→ loss: 0.0038→ accuracy: 0.88
2791 Epoch: 348→ Step: 300→ loss: 0.0062→ accuracy: 0.88
2792 Epoch: 348→ Step: 400→ loss: 0.0084→ accuracy: 0.88
2793 Epoch: 348→ Step: 500→ loss: 0.0071→ accuracy: 0.88
2794 Epoch: 348→ Step: 600→ loss: 0.0064→ accuracy: 0.88
2795 Epoch: 348→ Step: 700→ loss: 0.0052→ accuracy: 0.88
2796 Epoch: 349→ Step: 0→ loss: 0.0066→ accuracy: 0.88
2797 Epoch: 349→ Step: 100→ loss: 0.0027→ accuracy: 0.88
2798 Epoch: 349→ Step: 200→ loss: 0.0084→ accuracy: 0.88
2799 Epoch: 349→ Step: 300→ loss: 0.0030→ accuracy: 0.88
2800 Epoch: 349→ Step: 400→ loss: 0.0044→ accuracy: 0.88
2801 Epoch: 349→ Step: 500→ loss: 0.0068→ accuracy: 0.88
2802 Epoch: 349→ Step: 600→ loss: 0.0087→ accuracy: 0.88
2803 Epoch: 349→ Step: 700→ loss: 0.0056→ accuracy: 0.88
2804 Best accuracy: 0.891

```

Figure 18. Model ConvPool-CNN-C Log

```

11170 Epoch 349, Iteration 0, loss = 0.0094
11171 Checking accuracy on validation set
11172 Got 850 / 1000 correct (85.00)
11173
11174 Epoch 349, Iteration 100, loss = 0.0160
11175 Checking accuracy on validation set
11176 Got 850 / 1000 correct (85.00)
11177
11178 Epoch 349, Iteration 200, loss = 0.0160
11179 Checking accuracy on validation set
11180 Got 850 / 1000 correct (85.00)
11181
11182 Epoch 349, Iteration 300, loss = 0.0094
11183 Checking accuracy on validation set
11184 Got 850 / 1000 correct (85.00)
11185
11186 Epoch 349, Iteration 400, loss = 0.0153
11187 Checking accuracy on validation set
11188 Got 850 / 1000 correct (85.00)
11189
11190 Epoch 349, Iteration 500, loss = 0.0088
11191 Checking accuracy on validation set
11192 Got 850 / 1000 correct (85.00)
11193
11194 Epoch 349, Iteration 600, loss = 0.0106
11195 Checking accuracy on validation set
11196 Got 850 / 1000 correct (85.00)
11197
11198 Epoch 349, Iteration 700, loss = 0.0125
11199 Checking accuracy on validation set
12200 Got 849 / 1000 correct (84.90)
12201
12202 Best accuracy: 0.869

```

Figure 19. Model ALL-CNN-C Log

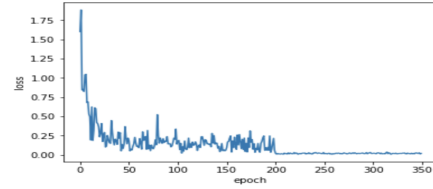


Figure 20. Model A Loss

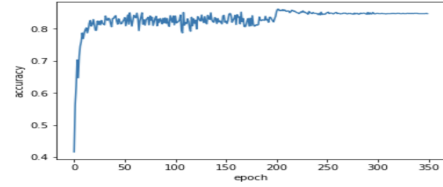


Figure 21. Model A Accuracy

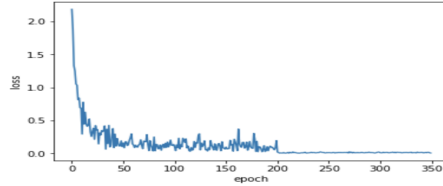


Figure 22. Model B Loss

### 6.3. Appendix 3

In classification task, the loss and accuracy figure of the six model is show as followed. In addition, we capture the log file.

### References

- [1] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

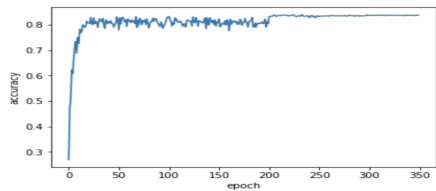


Figure 23. Model B Accuracy

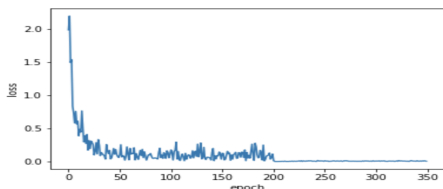


Figure 24. Model C Loss

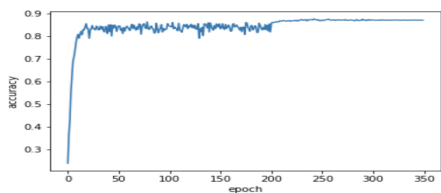


Figure 25. Model C Accuracy

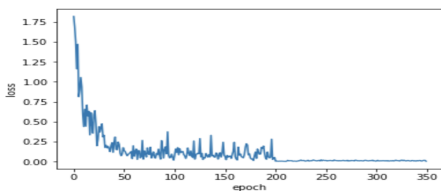


Figure 26. Model Strided-CNN-C Loss

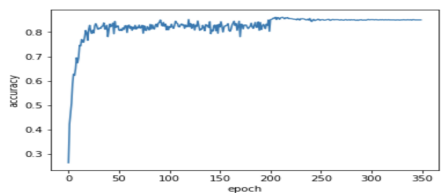


Figure 27. Model Strided-CNN-C Accuracy

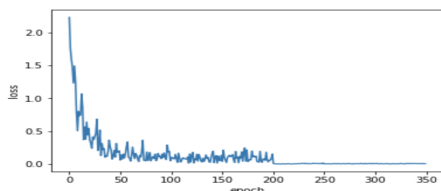


Figure 28. Model ConvPool-CNN-C Loss

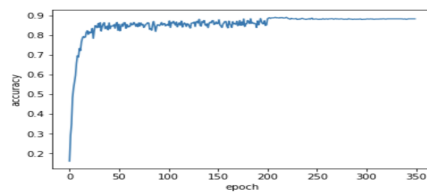


Figure 29. Model ConvPool-CNN-C Accuracy

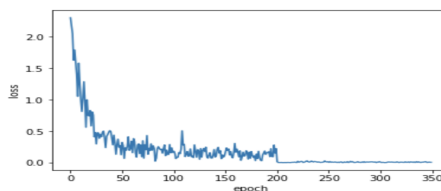


Figure 30. Model All-CNN-C Accuracy

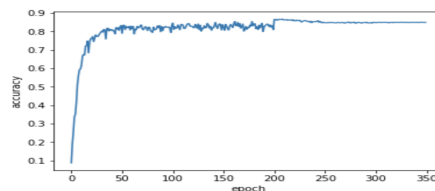


Figure 31. Model All-CNN-C Accuracy