COMP4211 Assignment 2 Report

Ng Chi Him
SID 20420921
chngax@connect.ust.hk

Results: CNN classifier from scratch

Code Screenshot

```python
class CnnFromScratch(nn.Module):
    def __init__(self, n_hidden):
        super(CnnFromScratch, self).__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, stride=2, padding=0),
            nn.ReLU(),
            nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2, padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=1, padding=0),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=0),
            nn.Sigmoid()
        )

        self.predictor = nn.Sequential(
            nn.Linear(in_features=32, out_features=n_hidden),
            nn.ReLU(),
            nn.Linear(in_features=n_hidden, out_features=47)
        )

        self.loss_func = nn.CrossEntropyLoss(reduction='sum')

    def forward(self, in_data):
        img_features = self.encoder(in_data).view(in_data.size(0), 32)
        logits = self.predictor(img_features)
        return logits

    def loss(self, logits, labels):
        return self.loss_func(logits, labels) / logits.size(0)

    def top_k_acc(self, logits, labels, k=1):
        _, k_labels_pred = torch.topk(logits, k=k, dim=1) # shape (n, k)
        k_labels = labels.unsqueeze(dim=1).expand(-1, k) # broadcast from (n) to (n, 1) to (n, k)
        # flatten tensors for comparison
        k_labels_pred_flat = k_labels_pred.reshape(1,-1).squeeze()
        k_labels_flat = k_labels.reshape(1,-1).squeeze()
        # get num_correct in float
        num_correct = k_labels_pred_flat.eq(k_labels_flat).sum(0).float().item()
        return num_correct / labels.size(0)
```

Please refer to code submission for other parts e.g. training/evaluation process.

Holdout Validation

Each candidate combination was tested for 10 epochs with shuffled batch size 32.

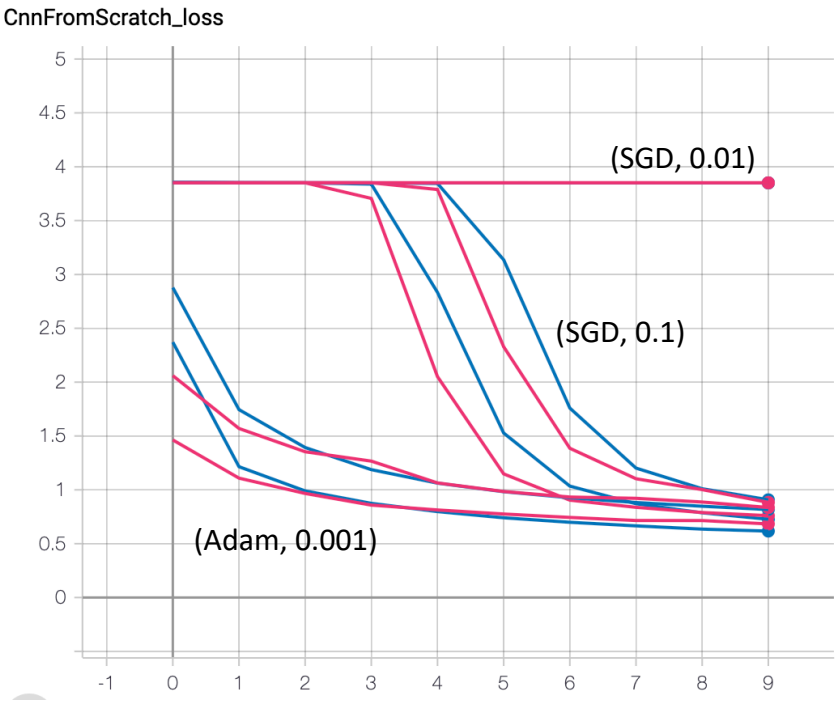
Minimal Validation Set Loss for each candidate combination:

	(Adam, 0.001)	(SGD, 0.1)	(SGD, 0.01)
H = 32	0.836	0.8832	3.851
H = 64	0.6833	0.7601	3.851

(Note: all candidates attained minimal loss at last epoch)

Comparing the two hidden layer sizes, H = 64 gave lower loss (~0.7) than H = 32 (~0.8) and was chosen for the final model.

As for optimizers, with reference to the graph of loss over epochs below,



(SGD, 0.01) failed to converge in 10 epochs and was not considered. While both (SGD, 0.01) and (Adam, 0.001) converged, consider that Adam was able to decrease the loss early on and gave lower loss at the end, it was chosen for the final model.

Final model

5 runs were executed with configuration below:

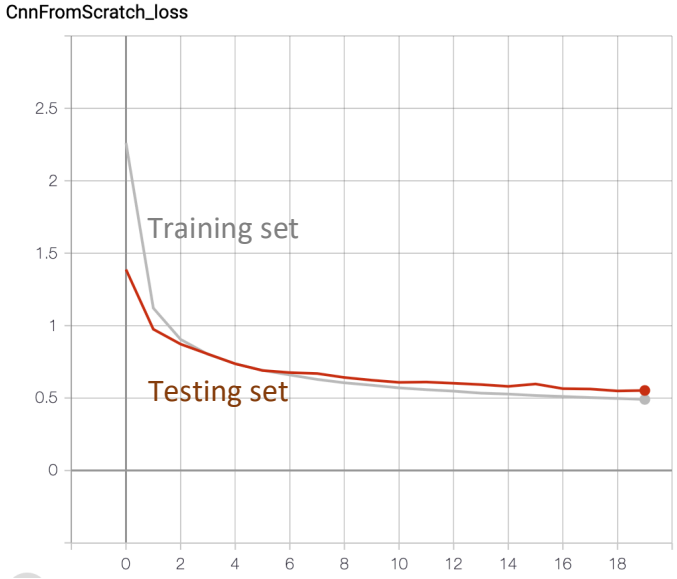
Hidden layer size: 64
Optimizer: Adam, learning rate 0.001
Batch size: 32, shuffled
Number of epochs: 20

Optimal validation metrics recorded:

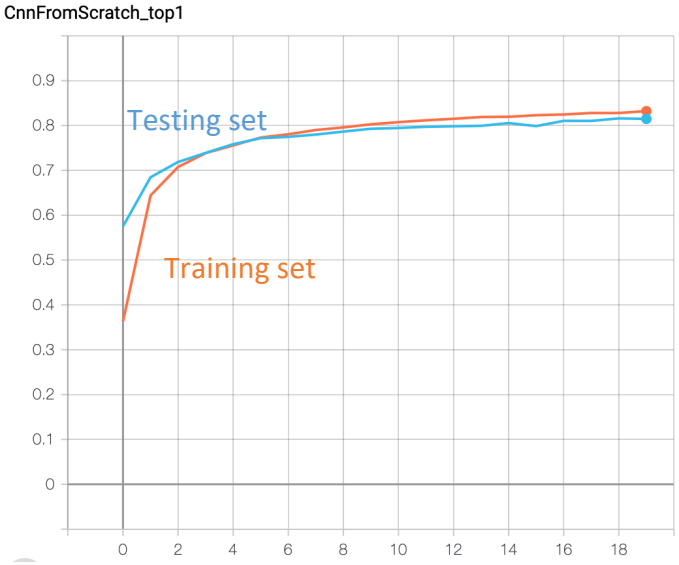
	Run 0 epoch 19	Run 1 epoch 16	Run 2 epoch 18	Run 3 epoch 18	Run 4 epoch 19	Mean	Std. Dev.
Loss	0.5804	0.6645	0.615	0.5489	0.5796	0.5977	0.0441
Top-1 Acc.	0.8117	0.7826	0.7972	0.8159	0.8039	0.8023	0.0131
Top-3 Acc.	0.9631	0.9546	0.9593	0.966	0.9651	0.9616	0.0047

Learning curves: (Run 3 was chosen for showcase)

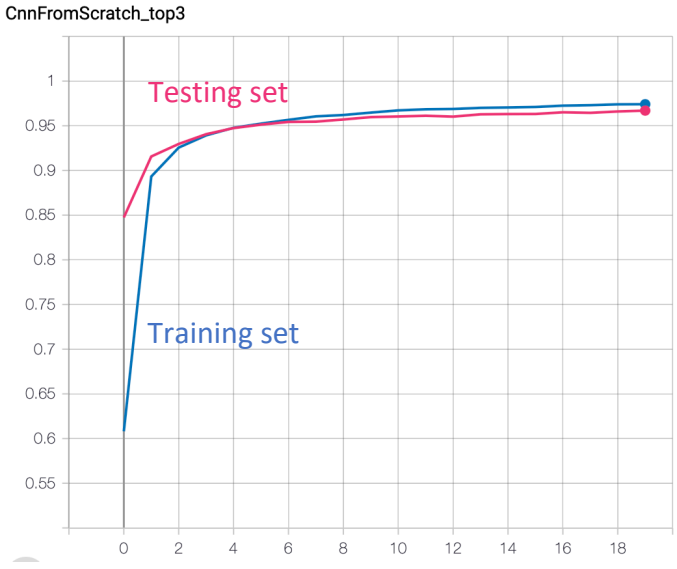
Loss over epoch



Top-1 Accuracy over epoch



Top-3 Accuracy over epoch



Results: CNN classifier with Pretrained Encoder

Code Screenshot

```
class CnnFromPretrained(nn.Module):
    def __init__(self, n_hidden):
        super(CnnFromPretrained, self).__init__()

        self.encoder = torch.load('pretrained_encoder.pt')['model']

        self.predictor = nn.Sequential(
            nn.Linear(in_features=32, out_features=n_hidden),
            nn.ReLU(),
            nn.Linear(in_features=n_hidden, out_features=47)
        )

        self.loss_func = nn.CrossEntropyLoss(reduction='sum')

    def forward(self, in_data):
        img_features = self.encoder(in_data).view(in_data.size(0), 32)
        logits = self.predictor(img_features)
        return logits

    def loss(self, logits, labels):
        return self.loss_func(logits, labels) / logits.size(0)

    def top_k_acc(self, logits, labels, k=1):
        _, k_labels_pred = torch.topk(logits, k=k, dim=1) # shape (n, k)
        k_labels = labels.unsqueeze(dim=1).expand(-1, k) # broadcast from (n) to (n, 1) to (n, k)
        # flatten tensors for comparison
        k_labels_pred_flat = k_labels_pred.reshape(1,-1).squeeze()
        k_labels_flat = k_labels.reshape(1,-1).squeeze()
        # get num_correct in float
        num_correct = k_labels_pred_flat.eq(k_labels_flat).sum(0).float().item()
        return num_correct / labels.size(0)
```

Please refer to code submission for other parts e.g. training/evaluation process.

Holdout Validation

Each candidate combination was tested for 10 epochs with shuffled batch size 32.

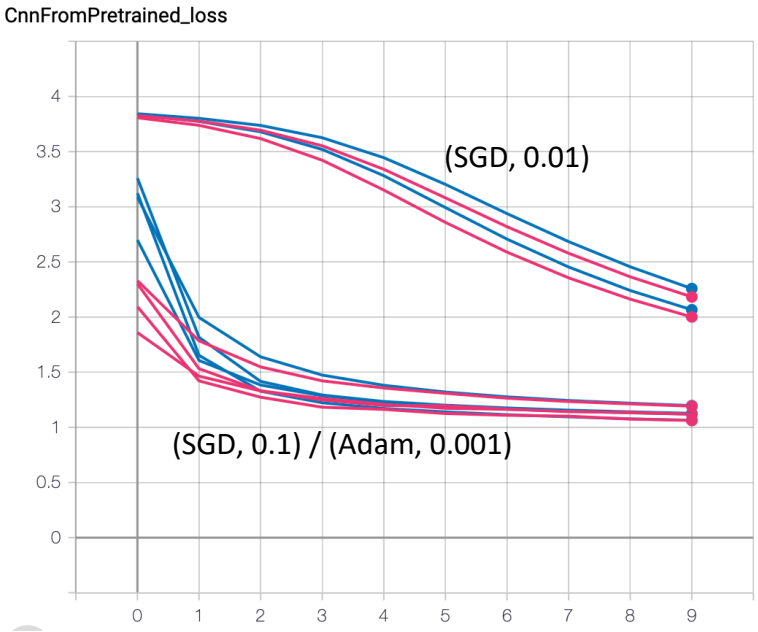
Minimal Validation Set Loss for each candidate combination:

	(Adam, 0.001)	(SGD, 0.1)	(SGD, 0.01)
H = 32	1.192	1.119	2.183
H = 64	1.121	1.063	2.001

(Note: all candidates attained minimal loss at last epoch)

For hidden layer size, H = 64 was chosen as it yielded slightly better loss across all three optimizers.

For optimizers, with reference to the graph of loss over epoch below,



We can see that (SGD, 0.01) had a slower loss decrease while (SGD, 0.1) and (Adam, 0.001) performed similarly with a steep curve. (SGD, 0.1) was chosen for the final model as it yielded the lowest loss after 10 epochs.

Final model

5 runs were executed with configuration below:

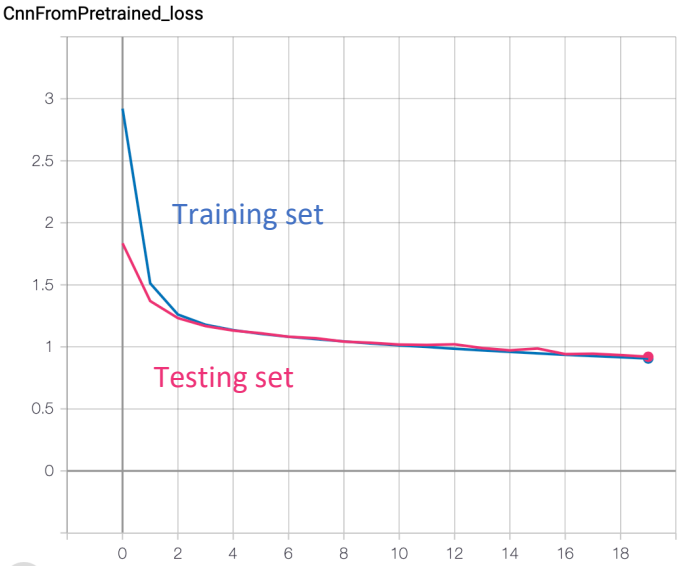
Hidden layer size: 64
Optimizer: SGD, learning rate 0.1
Batch size: 32, shuffled
Number of epochs: 20

Optimal validation metrics recorded:

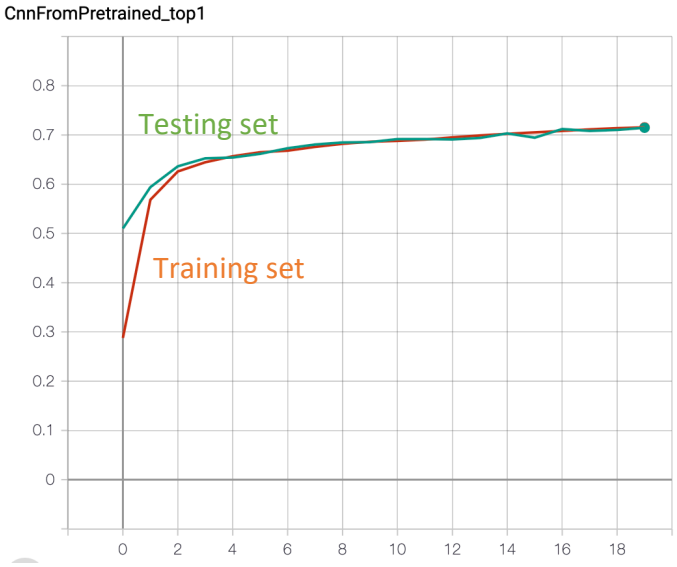
	Run 0 epoch 18	Run 1 epoch 19	Run 2 epoch 19	Run 3 epoch 18	Run 4 epoch 19	Mean	Std. Dev.
Loss	0.961	0.9483	0.9204	0.9362	0.9254	0.9383	0.0166
Top-1 Acc.	0.7064	0.7134	0.7147	0.7131	0.7147	0.7125	0.0035
Top-3 Acc.	0.9072	0.9108	0.9144	0.9114	0.914	0.9116	0.0029

Learning curves: (Run 2 chosen for showcase)

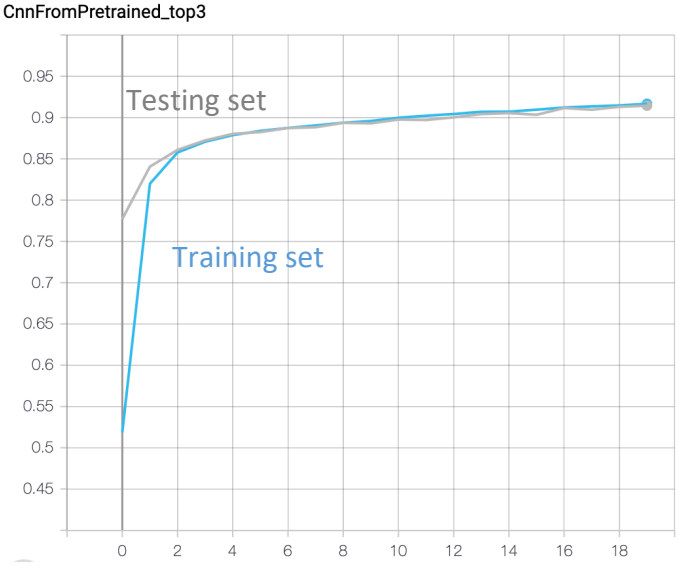
Loss over epoch



Top-1 accuracy over epoch



Top-3 accuracy over epoch



Discussion: Scratch vs Pretrained

	Mean over 5 runs		Standard Deviation over 5 runs	
	CNN from Scratch	CNN with Pretrained	CNN from Scratch	CNN with Pretrained
Training Time	6:25	3:50	N/A	
Number of epochs	20		N/A	
Loss	0.5977	0.9383	0.0441	0.0166
Top-1 Accuracy	0.8023	0.7125	0.0131	0.0035
Top-3 Accuracy	0.9616	0.9116	0.0047	0.0029

For training time, CNN from scratch took significantly more time to finish 20 epochs compared to that of CNN with pretrained encoder. This is because the number of parameters to be trained is much reduced in pretrained case thanks to the frozen encoder.

For performance, CNN from scratch gave slightly better result (around 9% higher top-1 accuracy) versus CNN with pretrained encoder. This is probably due to the flexibility gained from trainable encoder weights. To reduce this performance gap we could consider to allow training of ending layers of the imported encoder.

For consistency, CNN with pretrained encoder had lower deviation on the performance metrics over CNN from scratch. This could be due to the fact that the number of trainable weights are lower, thus the effect of random weight initiation is less prominent.

To conclude, despite the minor performance gap, building CNN with Pretrained Encoder could be a better alternative to building CNN from scratch as it saves a lot of training time (about half in this experiment) and gives higher consistency in resultant models. As suggested above the performance drawbacks can be reduced by enable training of some imported layers, i.e. trade some training time saved.

Results: CAE with Pretrained Encoder

Code Screenshot

```
class CaeFromPretrained(nn.Module):
    def __init__(self):
        super(CaeFromPretrained, self).__init__()

        self.encoder = torch.load('pretrained_encoder.pt')['model']

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(in_channels=32, out_channels=16, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=8, out_channels=8, kernel_size=3, stride=2, padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=8, out_channels=4, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=4, out_channels=1, kernel_size=4, stride=2, padding=0),
            nn.Sigmoid()
        )

        self.loss_func = nn.MSELoss(reduction='sum')

    def forward(self, in_data):
        img_features = self.encoder(in_data)
        logits = self.decoder(img_features)
        return logits

    def loss(self, logits, in_data):
        return self.loss_func(logits, in_data) / logits.size(0)
```

Please refer to code submission for other parts e.g. training/evaluation process.

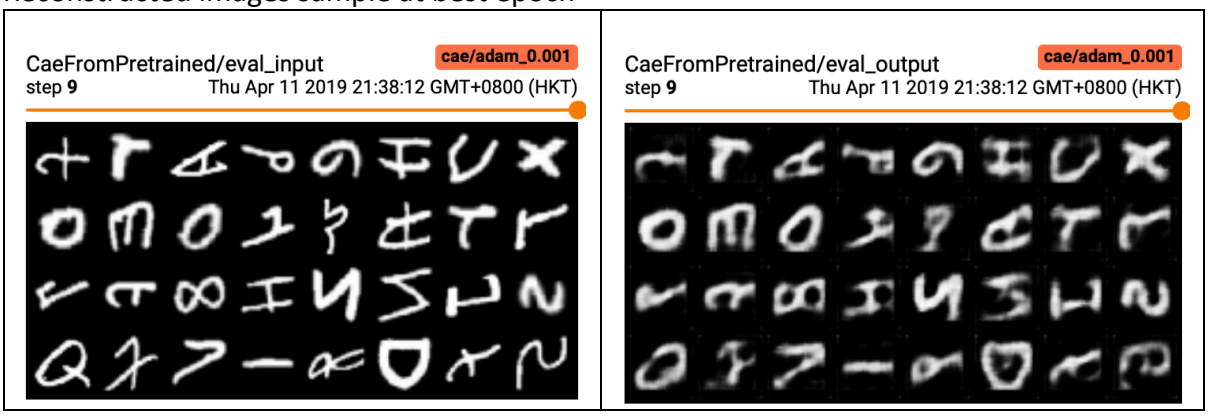
Holdout Validation

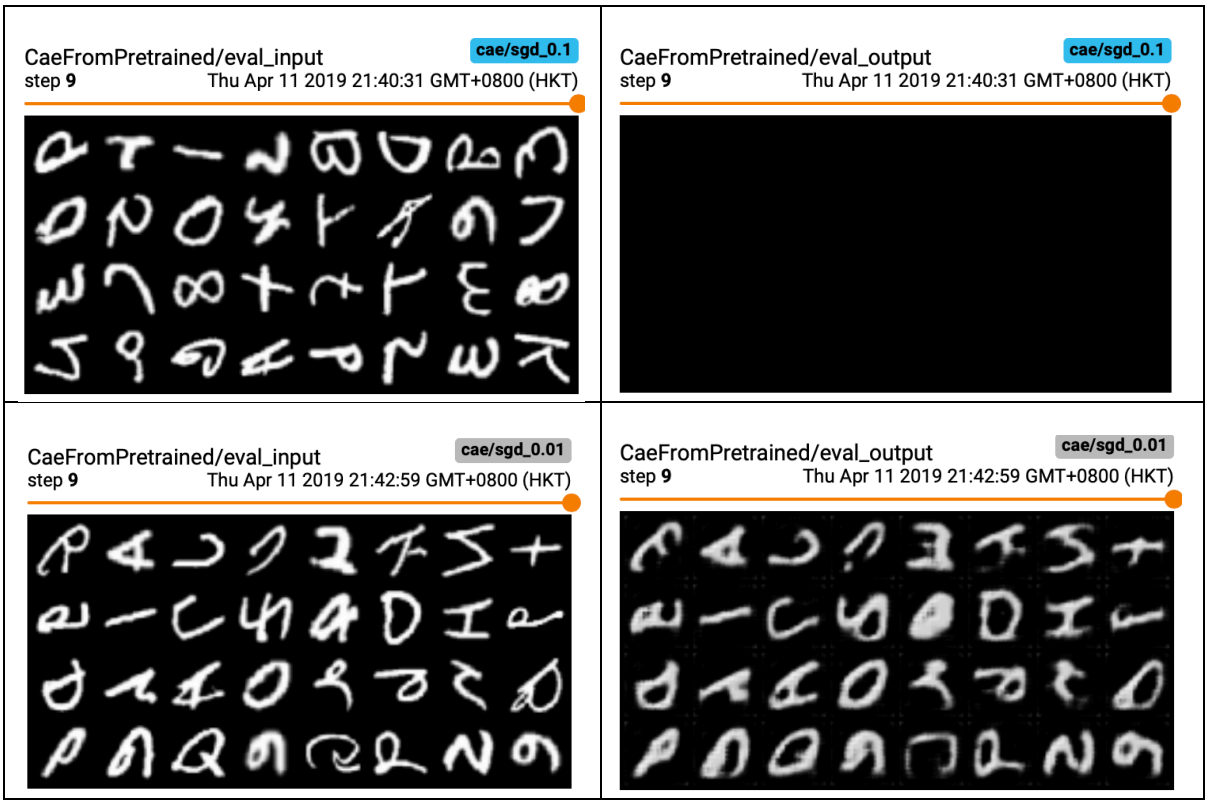
Each candidate combination was tested for 10 epochs with shuffled batch size 32.

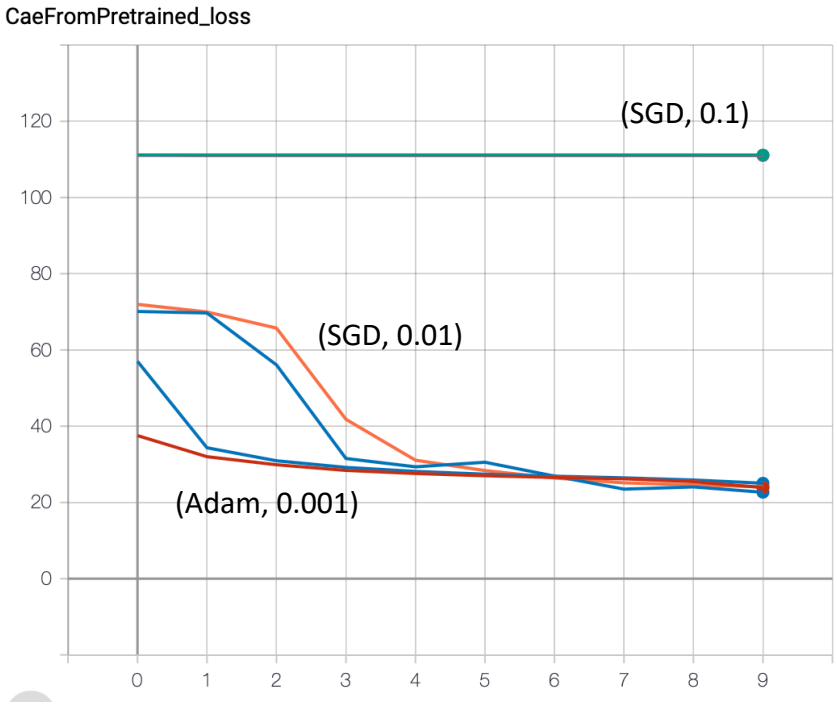
Minimal Validation Set Loss for each candidate combination:

	(Adam, 0.001) epoch 9	(SGD, 0.1) epoch 9	(SGD, 0.01) epoch 9
Loss	23.9	111.1	22.67

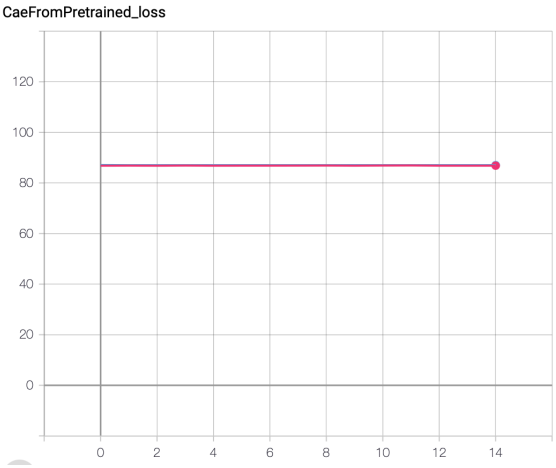
Reconstructed images sample at best epoch

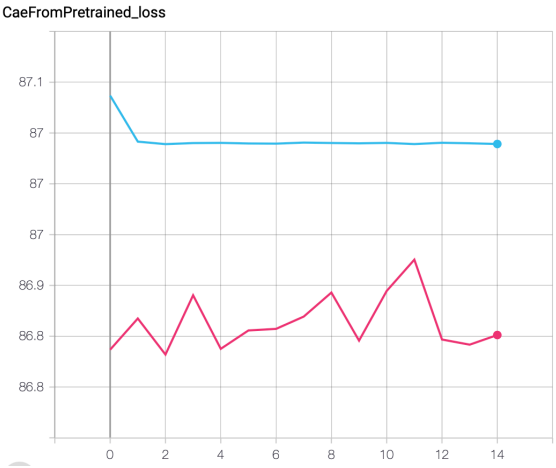






(SGD, 0.1) failed to converge in 10 epochs and was not considered. Although (Adam, 0.001) were able to converge faster than (SGD, 0.01), its resultant images are blurry and may contain visual artifacts or missing parts. (SGD, 0.01) was therefore initially selected for the final model, but its behaviour was inconsistent and failed to converge on final run. This is probably because momentum was not considered and thus the algorithm was stuck on local minima. See loss graph below captured during the supposedly “final” run:





Ultimately, (Adam, 0.001) was selected instead.

Final Model

Optimizer: Adam, learning rate 0.001
Batch size: 32, shuffled
Number of epochs: 15

Lowest testing set loss attained at epoch 14 with value 21.6:

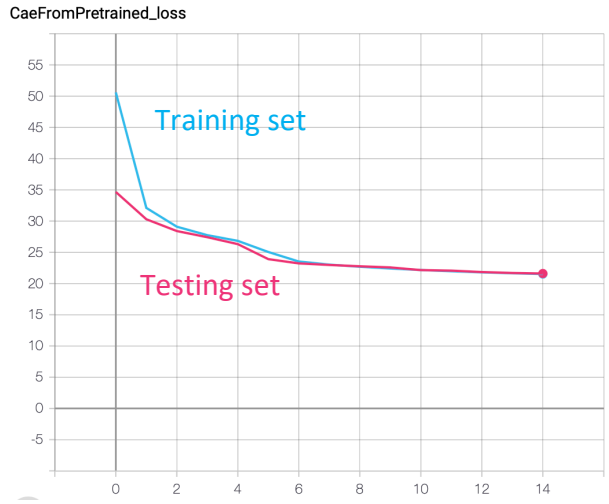


Image sample at epoch 14 (best epoch):