"Deep Learning for Computer Vision (2019)"

HomeWork 5

Report by Oleg Dats

1. Paper review

Paper

Title: Dynamic Routing Between Capsules
Authors: Sara Sabour, Nicholas Frosst, Geoffrey E. Hinton
Link: https://arxiv.org/pdf/1710.09829
Tags: Neural Network, CNN, CapsuleNet
Year: 2017

Summary

What

There are few main problems with CNN:

1. Representation of structure is very simple. No explicit notion of entity.
2. Precise location of features is getting lost due to a pooling layer.

Invariance vs. Equivariance:
Invariance: the same result without depending on object location. Network predicts cat no matter where the cat is located.
Equivariance: system works equally well across all positions but in addition has location property (for example: heat map)
Pooling layer makes CNN to have translation invariance but it pays by losing some portion of information (precise location of features)

Capsules: groups of neurons that outputs: probability that objects exists and different features of this entity (angle, pose, scale, defloration, color...)
Capsules can form hierarchy and represent sophisticated objects: first layer of capsules identify eyes, nose… and the next layer identify cat. It is doing so by clustering features from previous level capsules in high dimensional space. Hinton calls it high dimensional coincidence and mentions that it does not come by chance. It was highly inspired by minicolumns in our brain.

Capsule Net is promising for image segmentation tasks. Where exact location is very important. It also robust for affine transformations.
Individual dimensions of a capsule are represented as vectors. Interpretation of these vectors is a very interesting topic. Authors mentioned that it can represent features like rotation angle, stroke thickness… Higher capsule layers have larger vector due to increasing complexity of the features.

How

Paper suggest new idea to use the overall length of the vector of instantiation parameters to represent the existence of the entity. For example after a few convolutional layers we have 18 feature maps that can be reshaped into 9 vectors to represent 2 entities for each location. To get object presence probabilities we run to nonlinear squashing function on this vectors to have a size between 0 and 1.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

Routing by agreement is the main topic of the paper. Capsules on the first layer try to predict the output of the capsules on the second layer using transformation matrices.

Regularization by reconstruction: they have also added reconstruction network. That take a final layer and draws the related image.
Loss = margin loss (error in class prediction) + Alpha * reconstruction loss (real image - reconstructed image)
Reconstruction loss makes the network to preserve all important information and improves the final result.

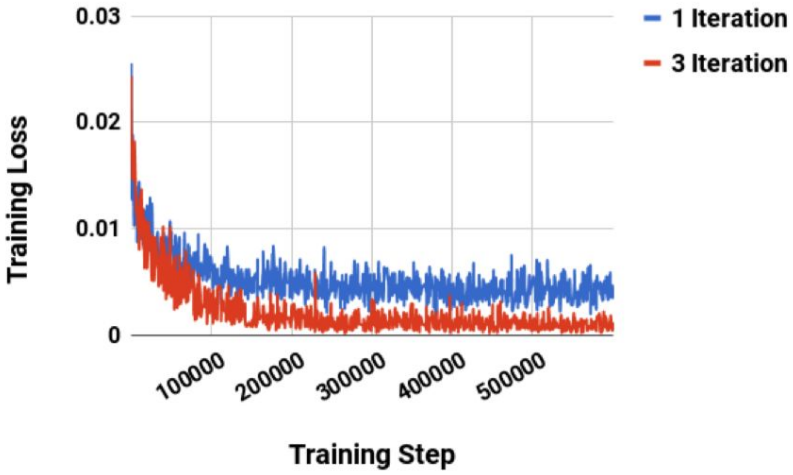
Results

Test error (0.25%) on MNIST data set.
Data augmentation: images that have been shifted by up to 2 pixels in each direction with zero padding.

Table 1: CapsNet classification test accuracy. The MNIST average and standard deviation results are reported from 3 trials.

Method	Routing	Reconstruction	MNIST (%)	MultiMNIST (%)
Baseline	-	-	0.39	8.1
CapsNet	1	no	$0.34_{\pm 0.032}$	-
CapsNet	1	yes	$0.29_{\pm 0.011}$	7.5
CapsNet	3	no	$0.35_{\pm 0.036}$	-
CapsNet	3	yes	$\mathbf{0.25}_{\pm 0.005}$	**5.2**

They also trained separate CNN to achieve the same results. CNN has 35.4M while CapsNet has 8.2M parameters



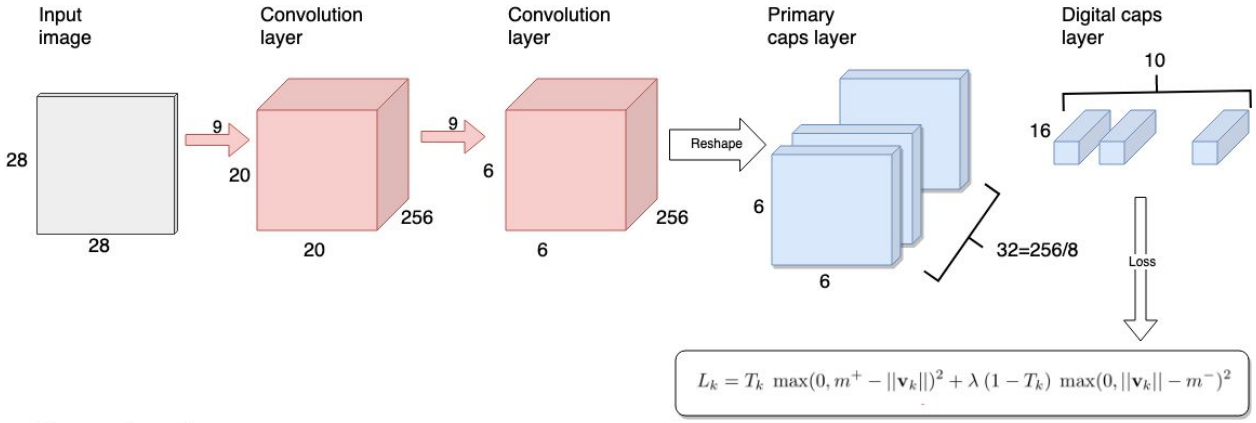
Training loss of CapsuleNet on cifar10 dataset

Training loss of CapsuleNet on cifar10 dataset: 10.6%. It is far away from state of the art CNN. The main problem is due to background. The next versions of CapsuleNet should learn to model the clutter.

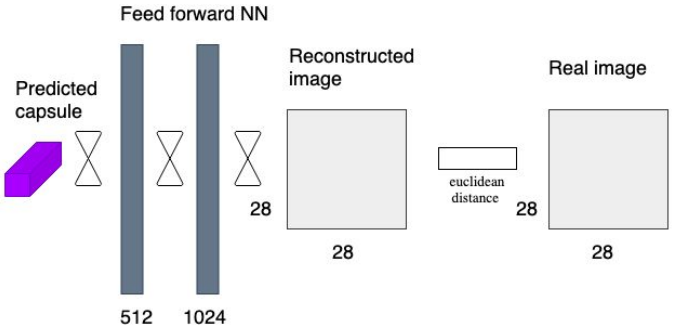
Training loss of CapsuleNet on smallNORB dataset: 2.7%. Almost the same as state-of-the-art. But this data set contains objects without background.

2. Network visualization

CapsuleNet:



Reconstruction process:



3. Experiment summary

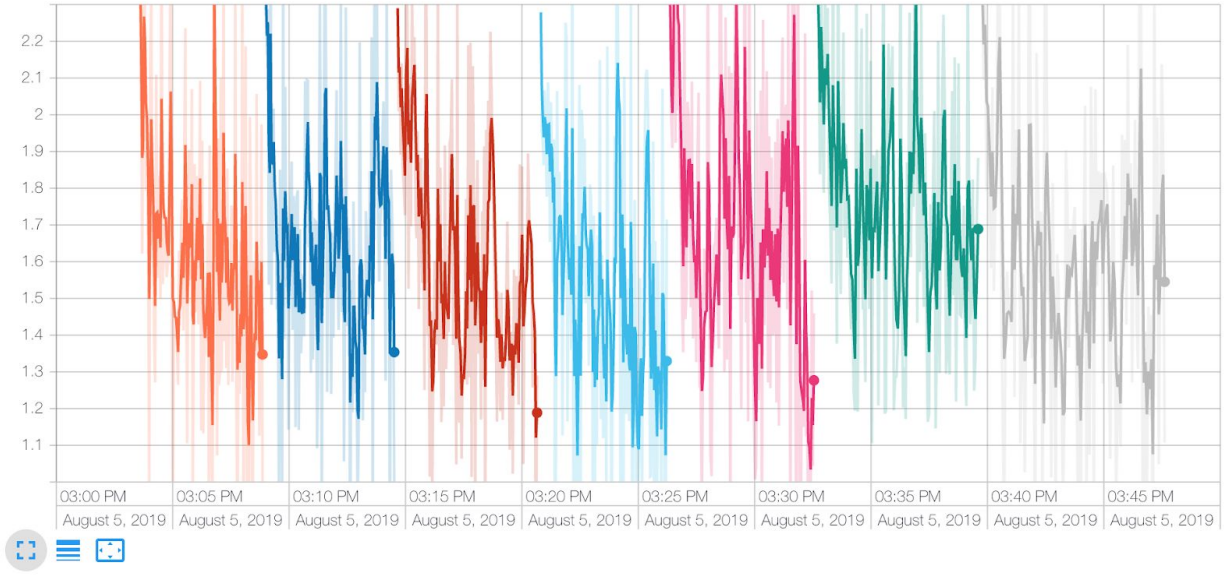
Compare architectures

From left to right

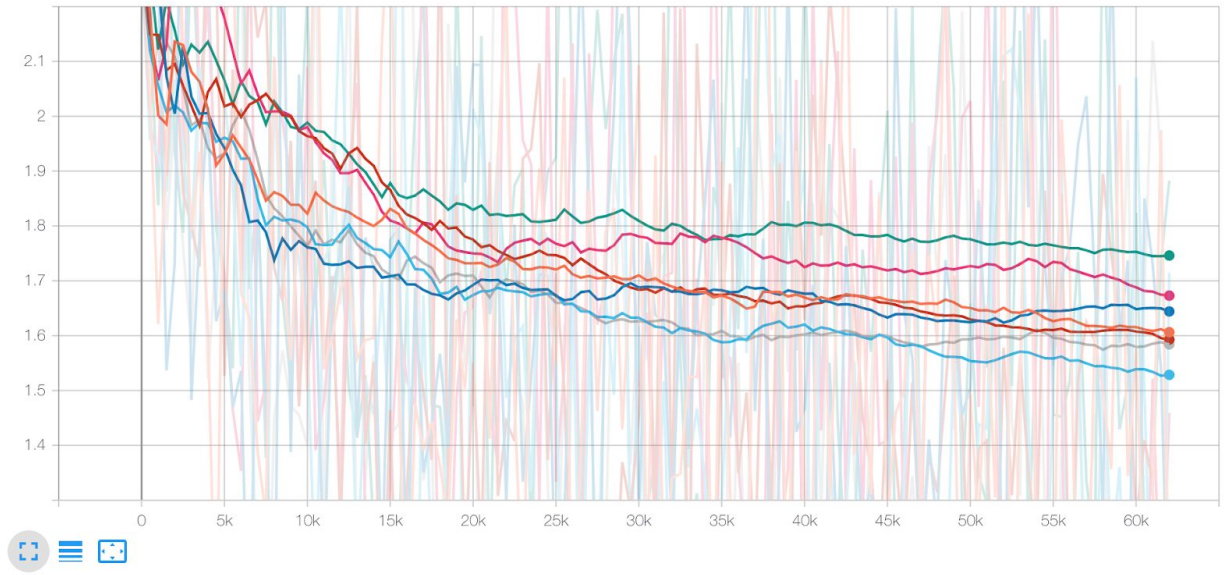
1. Default architecture, [conv-relu-pool]XN - [FC]XM
2. The same as #1. Increased filter size from 5 to 7
3. # [conv-relu-pool]xN - conv - relu - [FC]xM
4. The same as #1. Increased number of filters: 2x
5. # [conv-relu-conv-relu-pool]xN - [FC]xM (N=3)
6. The same as #5. Integrated bach norm.
   `self.conv1 = nn.Conv2d(3, 32, 7, padding=2)`
   `self.conv1_bn = nn.BatchNorm2d(32)`
   `self.pool1 = nn.MaxPool2d(2, 2)`
7. The same as #6. Increased number of filters: 2x

Summary:
I have tested 7 Neural Networks with small changes to architecture: last layer is pooling or convolution, with or without bach norm, different combinations of number and sizes of filters. My main outcome: deepest, wider (more filters) and more complicated NN shows better results. The main intuition: more layers can represent much more complicated function and the higher number of filters can capture more features.

RunningLoss
tag: Train/RunningLoss



RunningLoss
tag: Train/RunningLoss



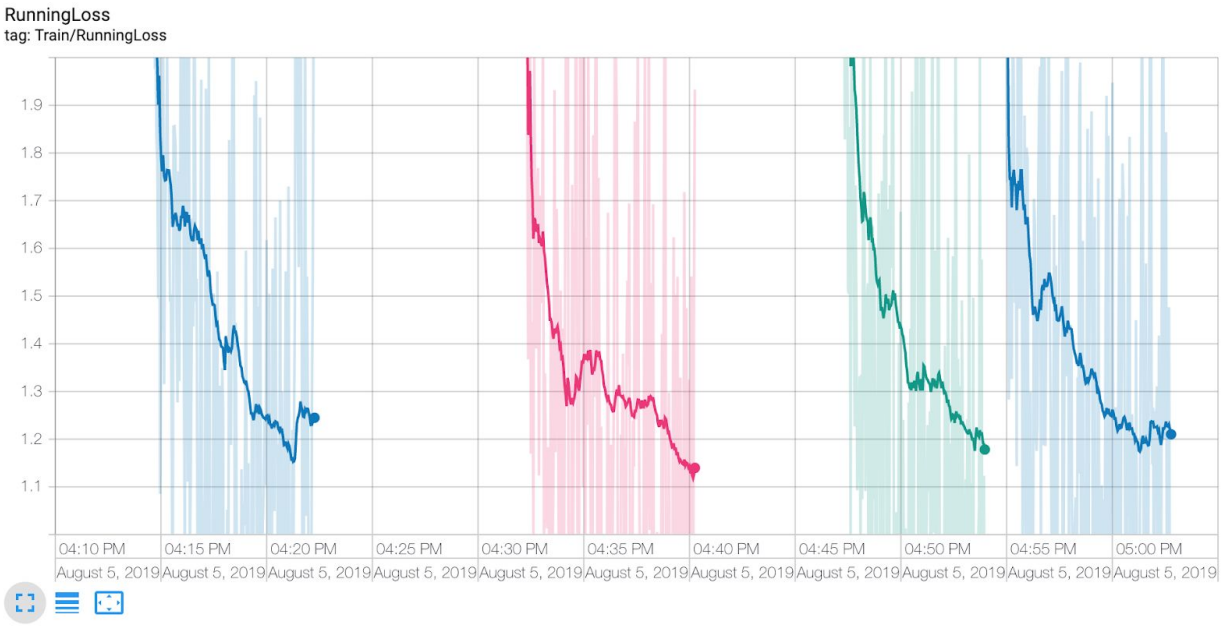
Alternative update steps

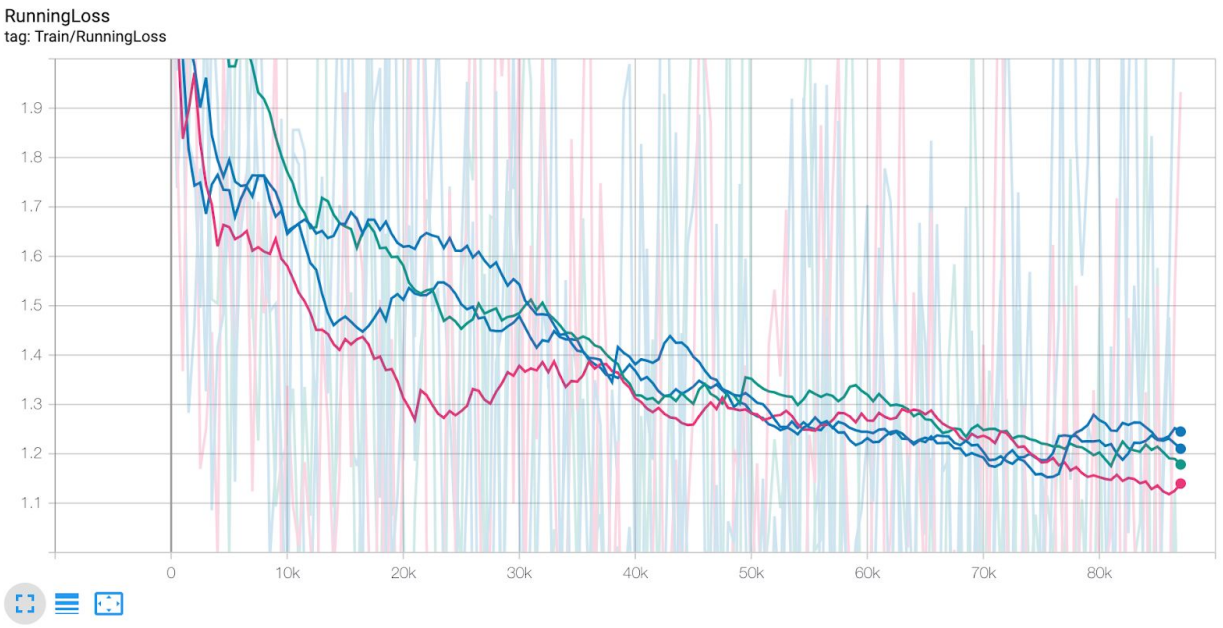
From left to right

1. Adam:
   `torch.optim.Adam(net.parameters(), lr=1e-4)`
2. Adam + learning rate decay:
   `torch.optim.Adam(net.parameters(), lr=1e-4, lr_decay=0.1)`
3. Adam + weight decay:
   `torch.optim.Adam(net.parameters(), lr=0.001, weight_decay=5e-4`
4. SGD + scheduler. Every 2 epoch the scheduler will decrease learning rate by 10%
   `torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)`
   `torch.optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.1)`
   `for epoch in range(1, n_epochs + 1):`
   `    scheduler.step()`

http://cs231n.github.io/neural-networks-3/

Summary:
I have tried different gradient update procedures. Including weight decay, learning rate decay, scheduler… I can not confirm the great advantage one approach over another. Maybe the main reason is that my Neural Network is relatively small. (less than 10 layers).
Rarely, but sometimes one procedure can be better than all others. I mean it converges faster during the first 10 echoes. I am not sure this performance will stay dominant on all other datasets. My main outcome is that performance of update procedure and related hyper parameters is highly correlated with datasets and structure of Neural Network.

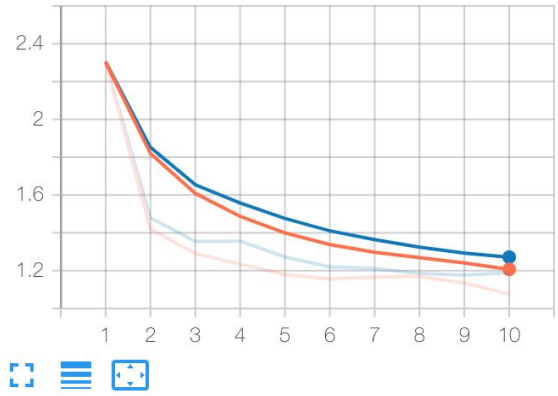




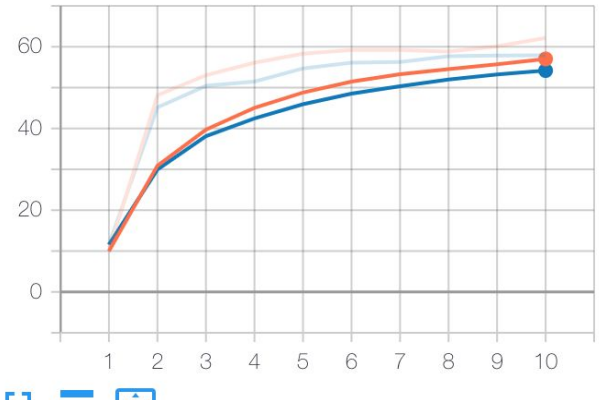
The charts display: y axis: learning loss (cross entropy loss), x axis: number of iterations.

MORE Tensorboard:

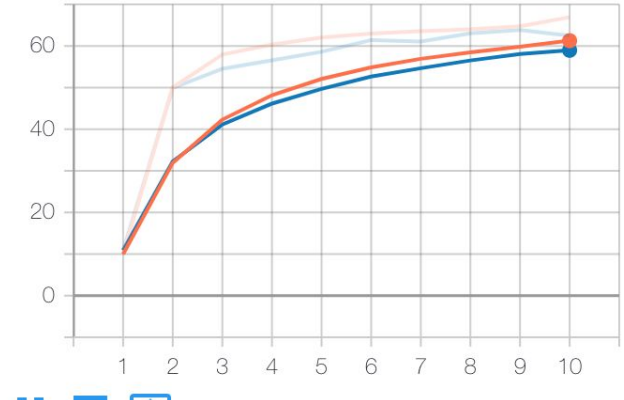
epoch_vs_loss
tag: Train/epoch_vs_loss



epoch_vs_accuracy_train_
tag: Train/epoch_vs_accuracy_train_



epoch_vs_accuracy_test_
tag: Test/epoch_vs_accuracy_test_



Data augmentation

Artificially extend data source to prevent overfitting and improve generalization.

1. Randomly change the brightness, contrast and saturation of an image.
2. Horizontally flip.
3. Crop the given image at a random location.
4. Rotate the image by angle.

```
transform_train = transforms.Compose(
  [
    torchvision.transforms.ColorJitter(hue=.04, saturation=.05),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.RandomCrop(32, padding=4),
    torchvision.transforms.RandomRotation(25),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
  ])
```

Compare time of training for CPU, GPU for different models

CPU times: user 25min 6s, sys: 2min 10s, total: 27min 17s
Wall time: 28min 41s

CPU times: user 9min 3s, sys: 1min, total: 10min 4s
Wall time: 13min 45s

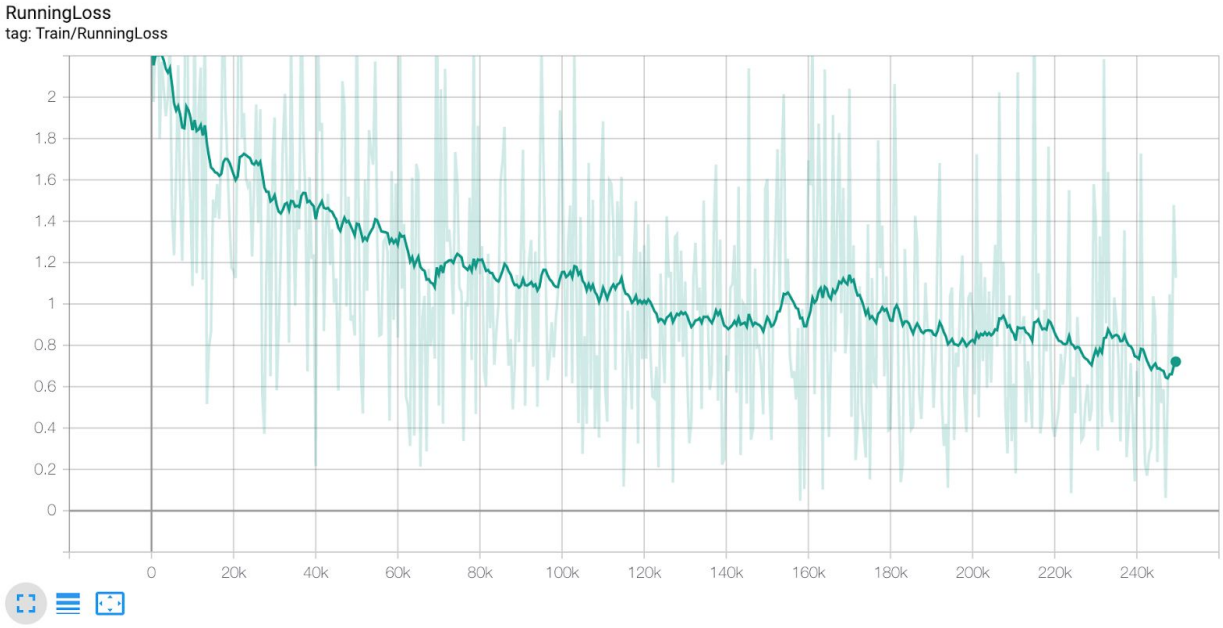
Summary:
Training NN with less than 10 layers took almost 3x less time on GPU then on CPU. This gain will increase while increasing the complexity of NN.

Final results and next steps

My final model:

```
class Net3b(nn.Module):
    def __init__(self):
        super(Net3b, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool3 = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(128*4*4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


Net3b(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=2048, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Final score on validation set:
Train Epoch: 20
Accuracy of the network on the 10000 test images: 75 %
CPU times: user 32min 25s, sys: 2min 23s, total: 34min 48s
Wall time: 36min 49s



How the model can be improved:

1. Add few more layers
2. Transfer learning: reuse pretrained model (VGG) without few last layers
3. Model ensembles: train few NN and combine their prediction to improve results and decrease variance.