Short Answers - Tiny ImageNet

Best Preforming Design

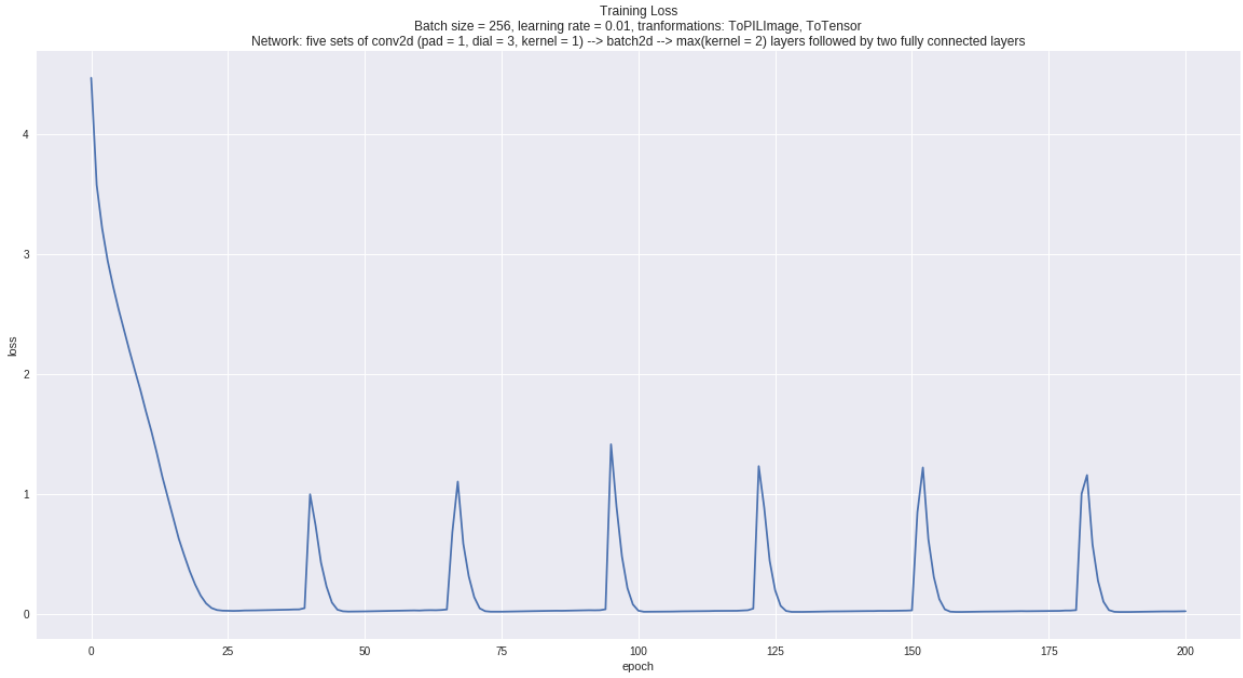
Network Design:

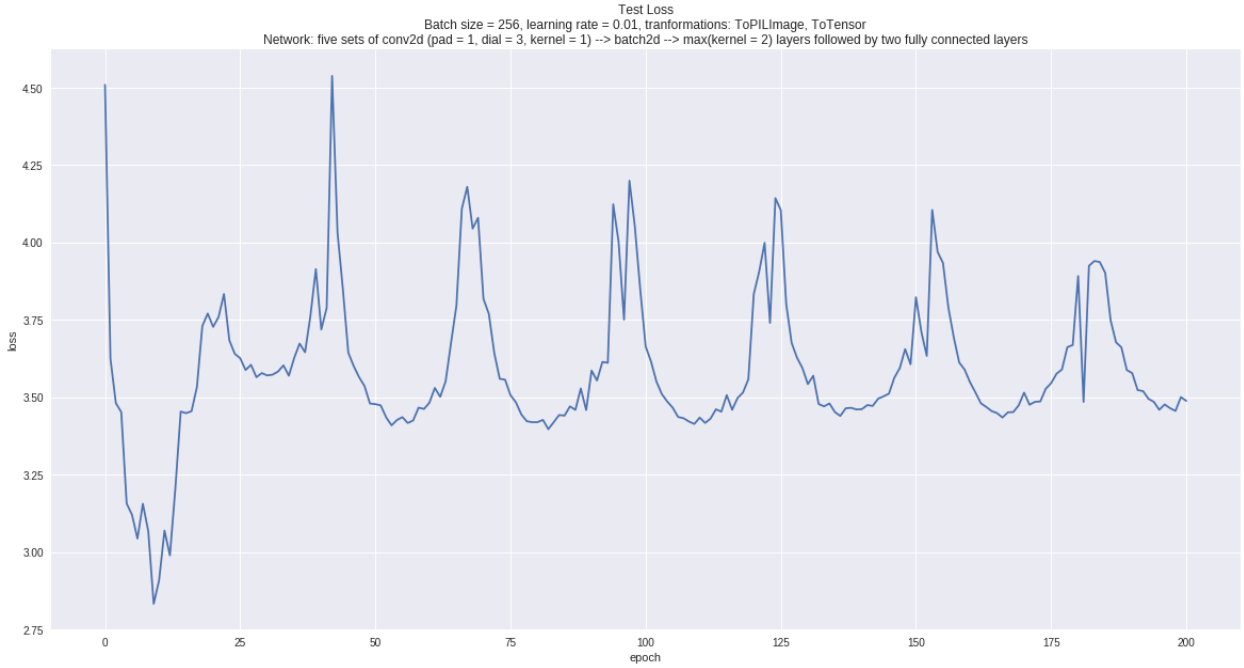
```
self.conv1 = nn.Conv2d(3, 32, kernel_size=KERNEL, padding=PADDING)
self.batch1 = nn.BatchNorm2d(32) # 64x64x32 -> 64x64x32

self.conv2 = nn.Conv2d(32, 64, kernel_size=KERNEL, padding=PADDING)
self.batch2 = nn.BatchNorm2d(64) # 32x32x64 -> 32x32x64

self.conv3 = nn.Conv2d(64, 128, kernel_size=KERNEL, padding=PADDING)
self.batch3 = nn.BatchNorm2d(128) # 16x16x128 -> 16x16x128

self.conv4 = nn.Conv2d(128, 256, kernel_size=KERNEL, padding=PADDING)
self.batch4 = nn.BatchNorm2d(256) # 8x8x256 -> 8x8x256

self.conv5 = nn.Conv2d(256, 512, kernel_size=KERNEL, padding=PADDING)
self.batch5 = nn.BatchNorm2d(512) # 4x4x512 -> 4x4x512

self.fc1 = nn.Linear(2 * 2 * 512, 1024)
self.fc2 = nn.Linear(1024, 200)
```

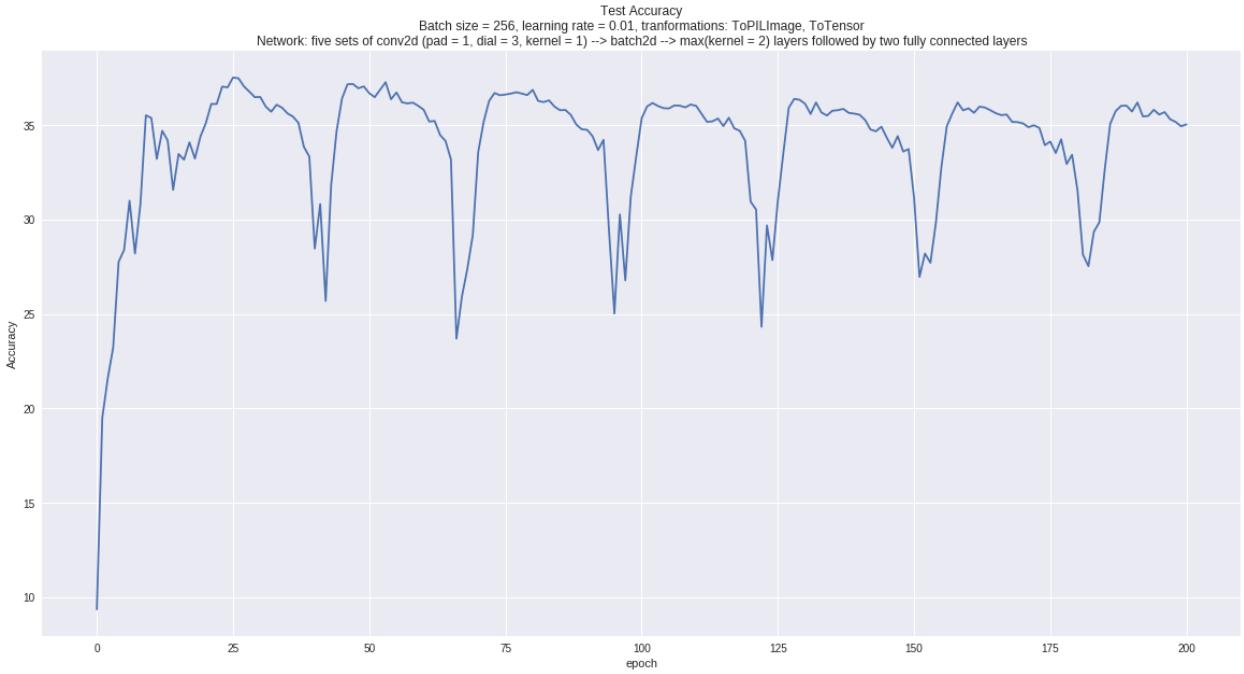
Each batch normalization layer is fed into a maxpool layer with kernel size = 2.

Train / Test Performance Figures

I used the default learning rate (0.01), batch size (256), number of epochs (200), and optimization parameters (momentum = 0.9, weight decay = 0.0005). I only applied typing transformations as data augmentation (ToPILImage and ToTensor).







My model is clearly bouncing around a local minima. I suspect that gradually decreasing the learning rate would help with this issue. Unfortunately, I did not have time to try this out.

Worst Performing Design

Unfortunately, I did not have time to try different settings. But compared to my original design, I suspect a network with similar convolutional and maxpooling layers but no batch normalization would encounter vanishing gradients issue and cannot be trained.

Short Answers – Full ImageNet

Design and Performance

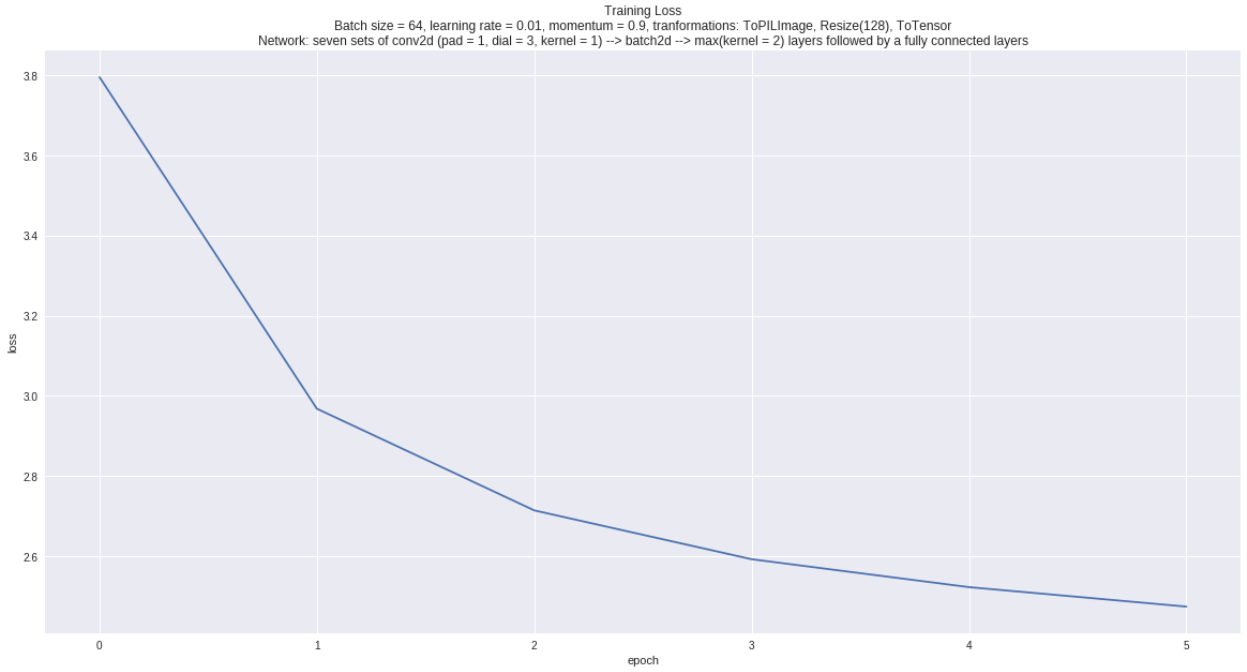
Network Design

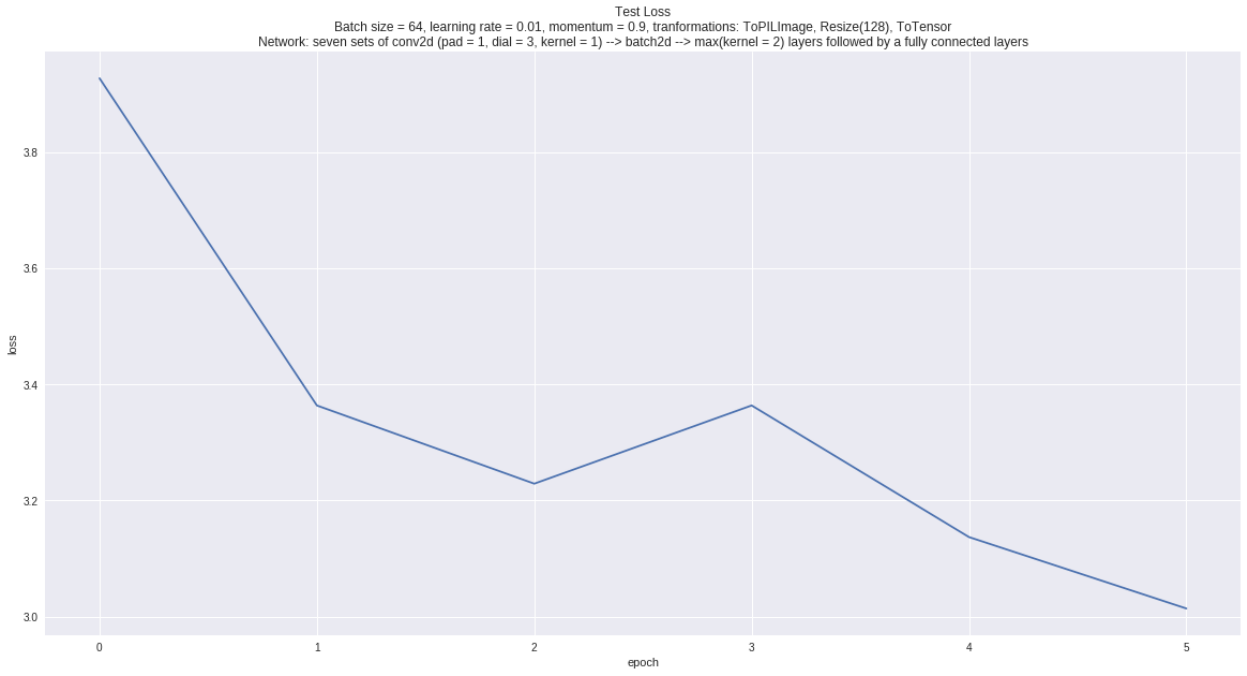
```
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
self.batch1 = nn.BatchNorm2d(32)

self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
self.batch2 = nn.BatchNorm2d(64)

self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
self.batch3 = nn.BatchNorm2d(128)

self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
self.batch4 = nn.BatchNorm2d(256)

self.conv5 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
self.batch5 = nn.BatchNorm2d(512)

self.conv6 = nn.Conv2d(512, 1024, kernel_size=3, padding=1)
self.batch6 = nn.BatchNorm2d(1024)

self.conv7 = nn.Conv2d(1024, 2048, kernel_size=3, padding=1)
self.batch7 = nn.BatchNorm2d(2048)

self.fc1 = nn.Linear(2048, 1000)
```

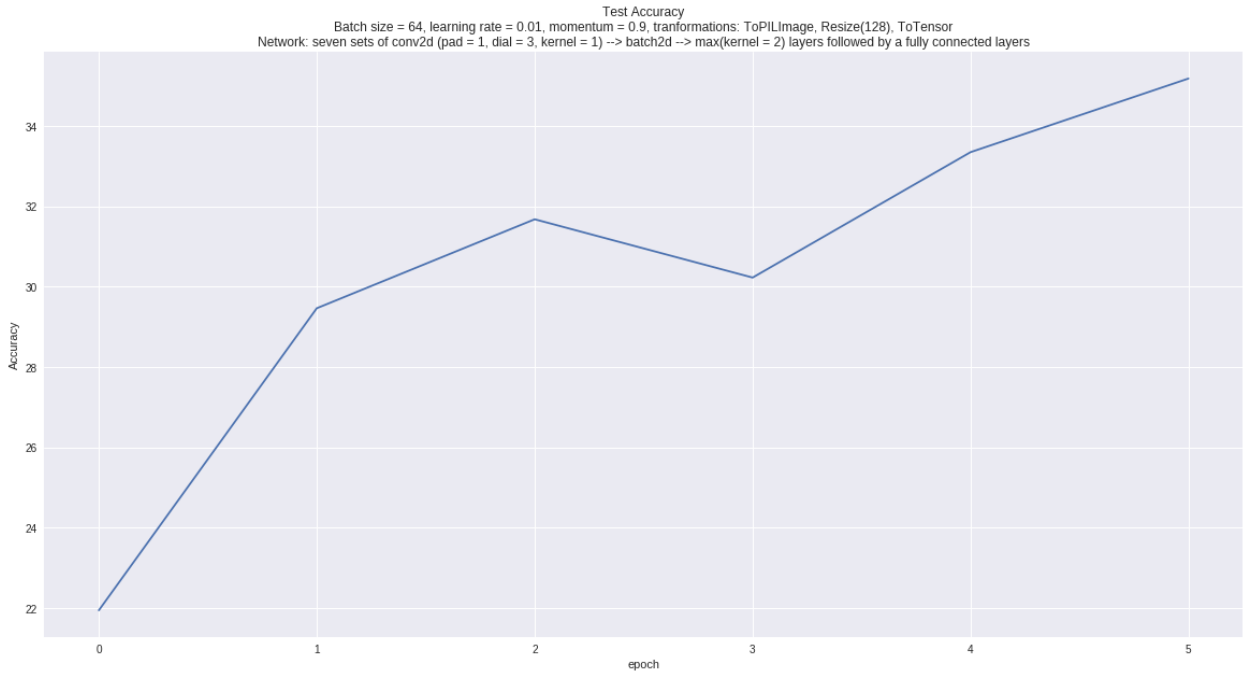
Similar to my Tiny ImegeNet design, each batch normalization layer is fed into a maxpool layer with kernel size = 2.

Train / Test Performance Figures

I used the default learning rate (0.01), batch size (64), and optimization parameters (momentum = 0.9, weight decay = 0.0005). I only applied typing transformations and resizing as data augmentation (ToPILImage, Resize([128, 128]), ToTensor). As I did not have time I only trained the model for 5 epochs. For the same reason I did not try different designs or learning settings.


Training Loss
Batch size = 64, learning rate = 0.01, momentum = 0.9, tranformations: ToPILImage, Resize(128), ToTensor
Network: seven sets of conv2d (pad = 1, dial = 3, kernel = 1) --> batch2d --> max(kernel = 2) layers followed by a fully connected layers
loss
3.8
3.6
3.4
3.2
3.0
2.8
2.6
0
1
2
3
4
5
epoch


Test Loss
Batch size = 64, learning rate = 0.01, momentum = 0.9, tranformations: ToPILImage, Resize(128), ToTensor
Network: seven sets of conv2d (pad = 1, dial = 3, kernel = 1) --> batch2d --> max(kernel = 2) layers followed by a fully connected layers
loss
3.8
3.6
3.4
3.2
3.0
0
1
2
3
4
5
epoch



Comparisons between Tiny ImageNet and Full ImageNet

I used a deeper network for ImageNet. Unfortunately I was unable to train the ImageNet network for the same number of epochs so I cannot really compare accuracy for the two. I suspect my deeper network is stronger yet it does not suffer from overfitting.

Larger/Smaller Images

To handle larger images I would try the following with the goal of learning to resize through convolution:

- adding extra convolution-batchnorm-max layers, which capture the most relevant information for the next layers to process
- convolutional layers with filters that also down sample (e.g. for a kernel of size 3 at pixel (i, j) impacts rows (i-2) and (i+2) and columns (j-2) and (j+2) in addition to max pooling, which hopefully capture the relationships across larger locality
- convolutional layers with larger kernels and striding > 1, which again capture the relevant information while also down sampling.

To handle smaller images I follow a similar approach: learning to resize through convolution in a way done for image segmentation by the deconvolution network.

We might end up overfitting if we are introducing too many parameters for resizing (lowered accuracy). But in general, I expect no loss of accuracy for a properly designed network.