Tiny Imagenet Classification

I have tried several deep learning architectures, including my own sequential experiments and popular models imported (but not pretrained) from PyTorch, such as VGG, ResNet etc. They all gave me the same result of 0.5% accuracy, and I understood that the problem was not in architecture (this is why I missed the deadline btw). Nonetheless, I decided to stick with resnet18 from PyTorch and add one more fully connected layer to it to adapt it to our problem and provide the number of outputs equal to 200 – the probabilities of belonging to each of the classes. I chose resnet18 because I have googled some performance benchmarks for Tiny Imagenet (http://cs231n.stanford.edu/reports/2017/pdfs/931.pdf for example) and got the impression that resnet18 is the best choice for that dataset.

I have tried training my model with batch size equal to 128 (for both training and validation) but it showed poor performance in terms of calculational time. So, I changed batch size to 64. However, my accuracy was still 0.5% – exactly the result you would get for random guessing, because 1 / 200 * 100% = 0.5%. I also tried two optimization algorithms: Adam and SGD with momentum=0.9. Nothing was changing, the performance was still bad. It seemed like the model was not learning at all.

Finally, I understood the reason for bad performance. I was applying Softmax to the results of my model before feeding them to the loss function. And I used CrossEntropyLoss from PyTorch after that. But this loss function already implements something similar to Softmax under the hood. This is the screenshot from PyTorch documentation indicating that.

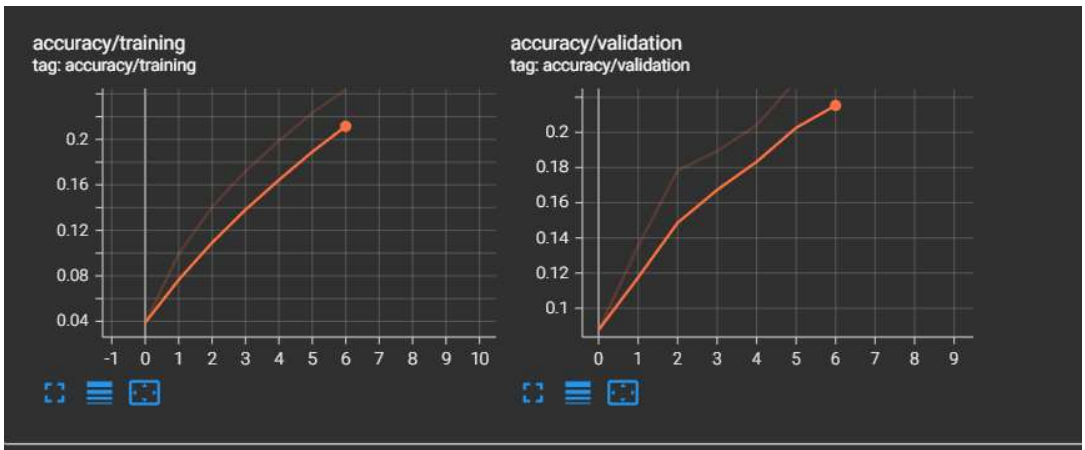
- Probabilities for each class; useful when labels beyond a single class per minibatch item are required, such as for blended labels, label smoothing, etc. The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

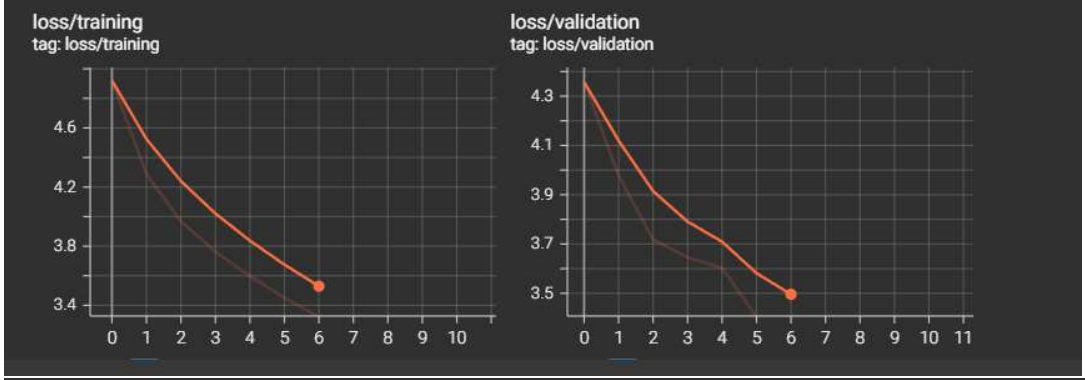
Thus, I found the reason for the bad performance of my model. It was experiencing a well-known vanishing gradient problem because of twice applied Softmax. That's why the loss function and accuracy were not improving.

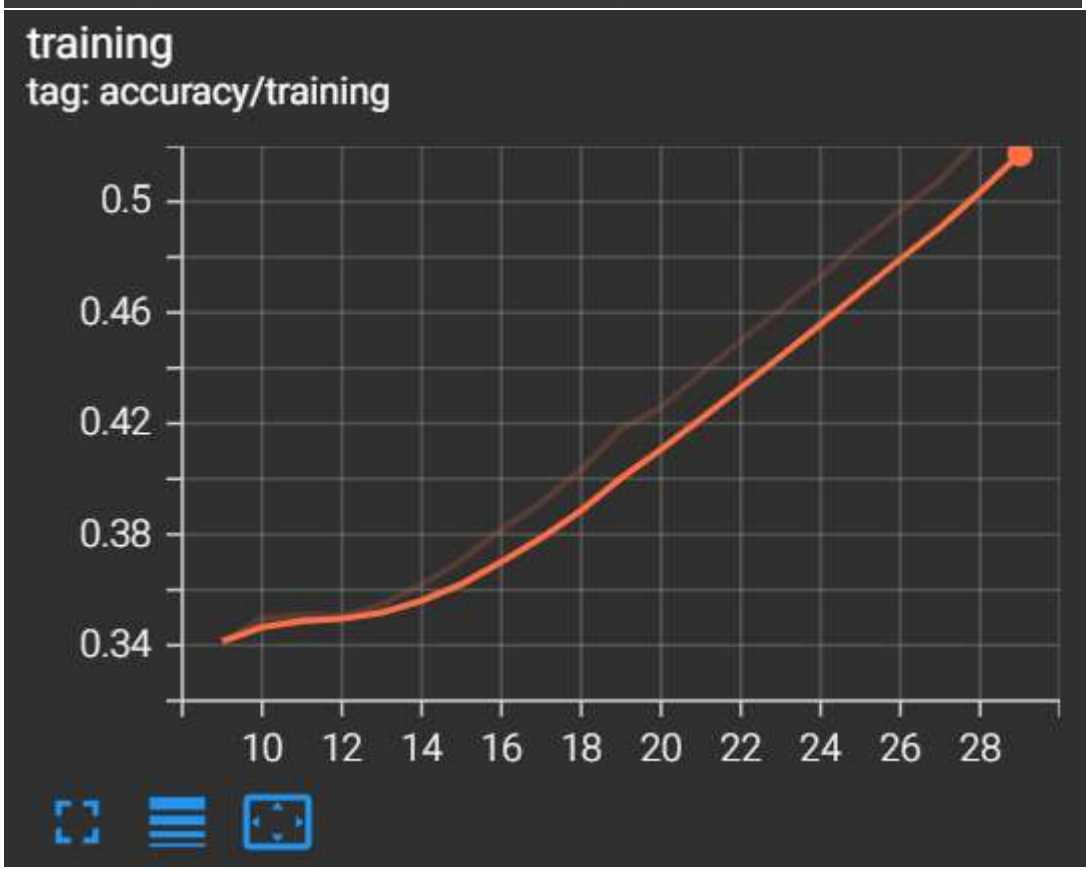
After I fixed that, the problem was solved, and the learning process went well. During training, the model achieved 42% accuracy on validation set. I attach the screenshots from TensorBoard with accuracy and loss curves for training and validation. There are two sets of them, because, after reaching the limit for GPU usage, the training process was interrupted by Colab. I had to reconnect to Colab and continue training from another account from the saved checkpoint file.

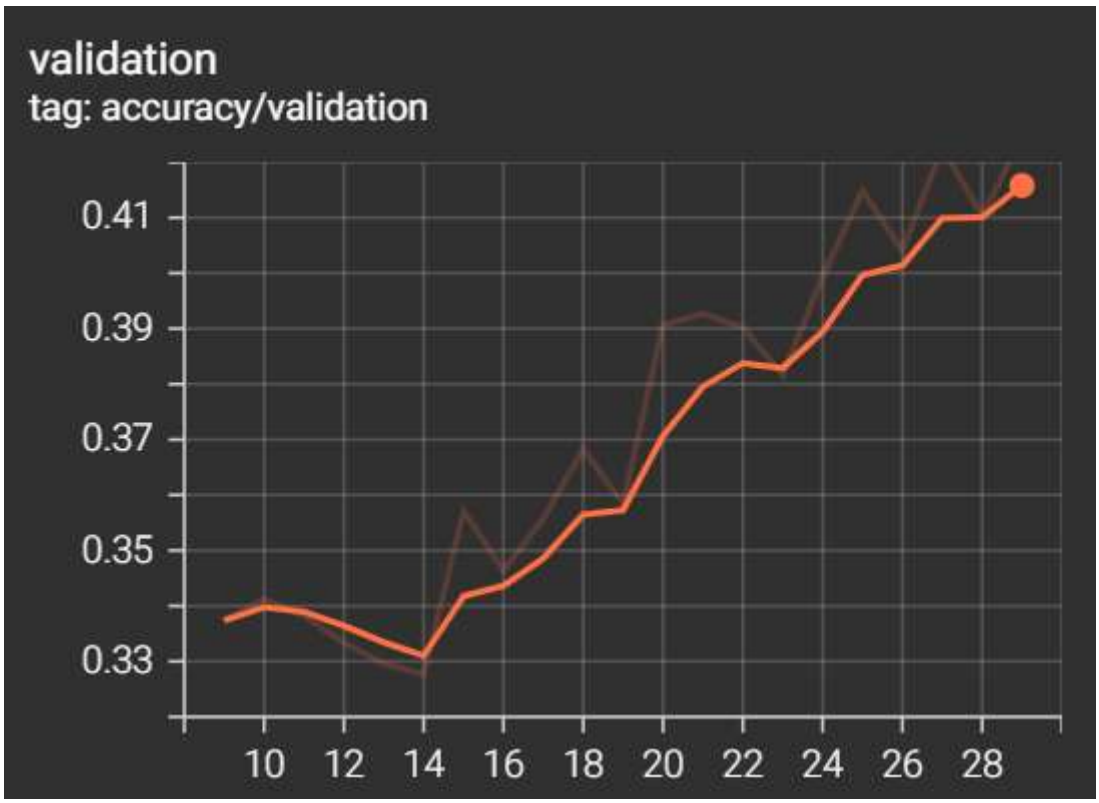
First, these are screenshots of the first 30 epochs.

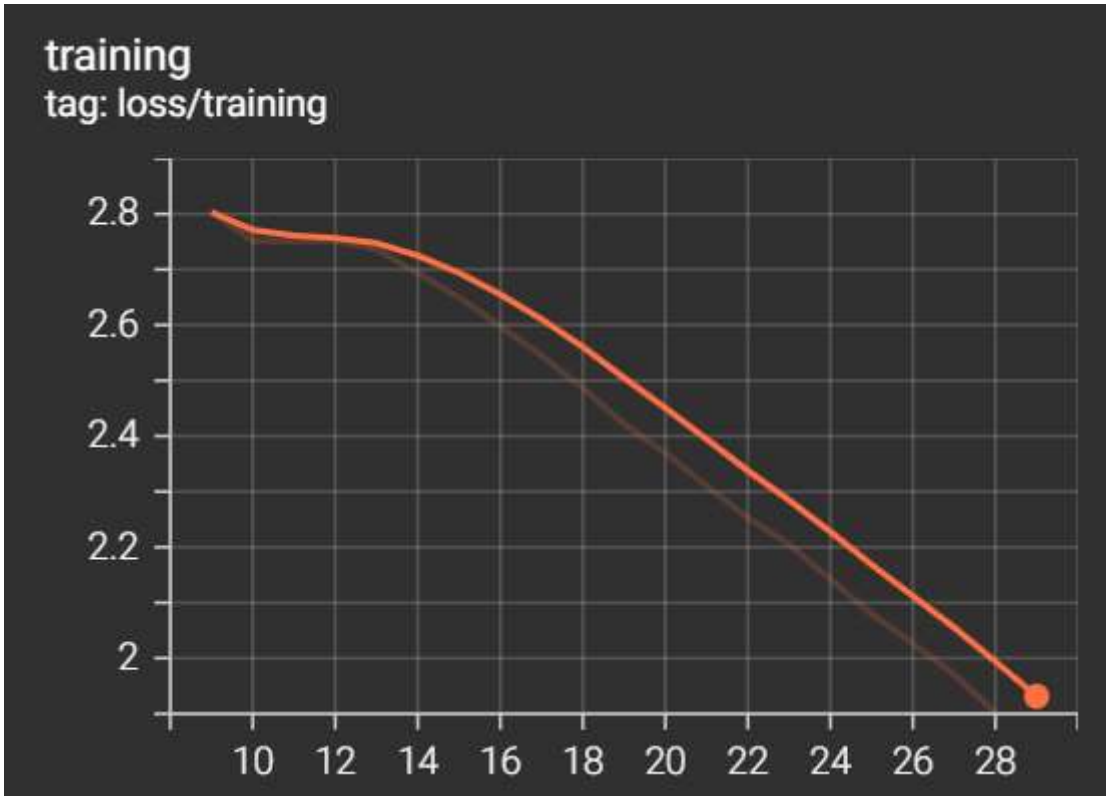

accuracy/training
tag: accuracy/training
0.2
0.16
0.12
0.08
0.04
-1 0 1 2 3 4 5 6 7 8 9 10
accuracy/validation
tag: accuracy/validation
0.2
0.18
0.16
0.14
0.12
0.1
0 1 2 3 4 5 6 7 8 9

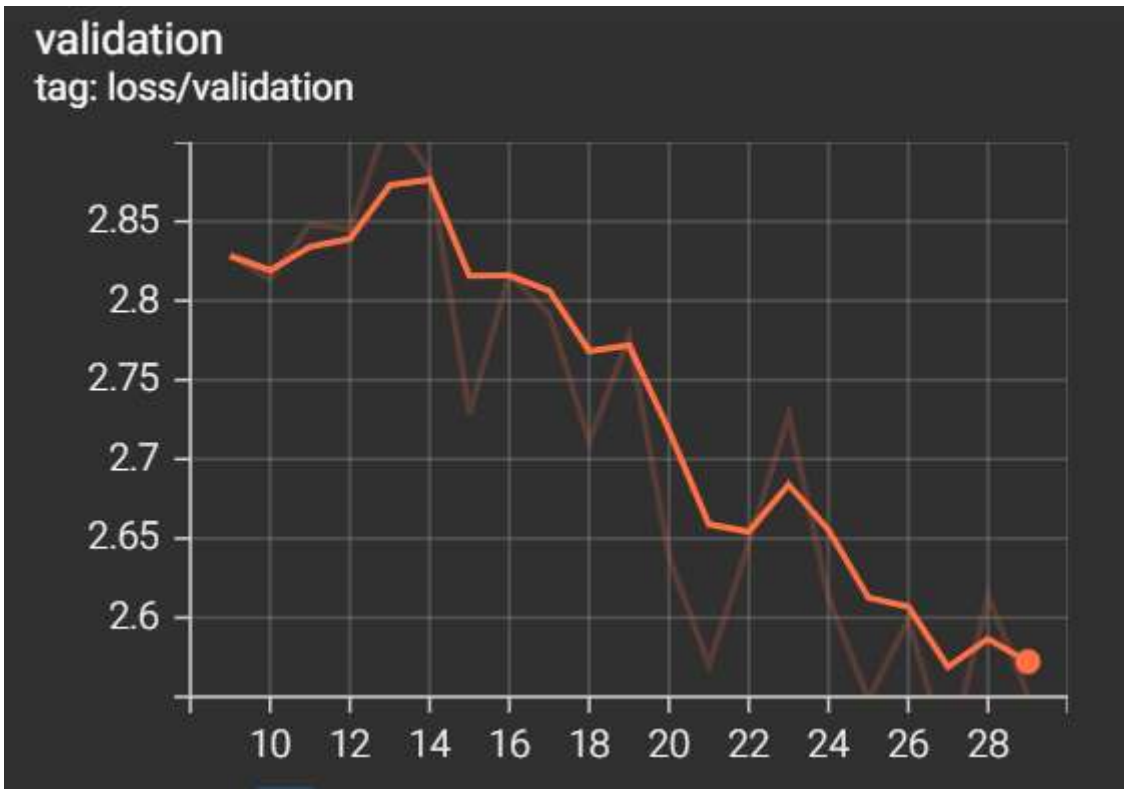
loss


loss/training
tag: loss/training
4.6
4.2
3.8
3.4
0 1 2 3 4 5 6 7 8 9 10
loss/validation
tag: loss/validation
4.3
4.1
3.9
3.7
3.5
0 1 2 3 4 5 6 7 8 9 10 11


training
tag: accuracy/training
0.5
0.46
0.42
0.38
0.34
10 12 14 16 18 20 22 24 26 28


validation
tag: accuracy/validation
0.41
0.39
0.37
0.35
0.33
10
12
14
16
18
20
22
24
26
28


training
tag: loss/training
2.8
2.6
2.4
2.2
2
10
12
14
16
18
20
22
24
26
28



After that I conducted a few more epochs but the model started to overfit.

The training loop is quite simple: for each epoch, we iterate over train data and optimize the weights. In the end of each epoch, I run `validate` function to calculate validation loss and accuracy. I also save a checkpoint of the weights in the end of each epoch. I also tried different LR schedulers: ReduceLROnPlateau, OneCycleLR. For my problem, OneCycleLR was giving better results.

For anti-overfitting, I used batch normalization layers (they are already implemented in resnet18). I also applied following transformations to the images: RandomResizedCrop, RandomHorizontalFlip, ToTensor, Normalize((.485, .456, .406), (.229, .224, .225)) for training data and ToTensor, Normalize((.485, .456, .406), (.229, .224, .225)) for test data. The interesting effect was that the accuracy and the loss on the validation set were better than on the train set (but only for some specific epochs), which was quite unusual for me. Seems like data augmentations decrease the model accuracy on the train set making it harder for the model to classify the images. But it gives better results on the validation set. Hard learning – easy inference.

From this exercise, I will remember for the rest of my life never to use Softmax with CrossEntropyLoss.

Another insight: it is a good idea to set pin_memory=True in data loaders when working with CUDA. Loading the data takes less time then.

Sources of code (also marked in my code in comments in the corresponding places).

Transforms:
https://pytorch.org/tutorials/beginner/basics/transforms_tutorial.html#transforms
https://pytorch.org/vision/stable/transforms.html

Model architecture:
http://cs231n.stanford.edu/reports/2017/pdfs/931.pdf
https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

Training and validation loop:
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-

pages/_downloads/361c46b0182e04464775765192096219/optimization_tutorial.ipynb#scrollTo=i9lSncaIXYCW