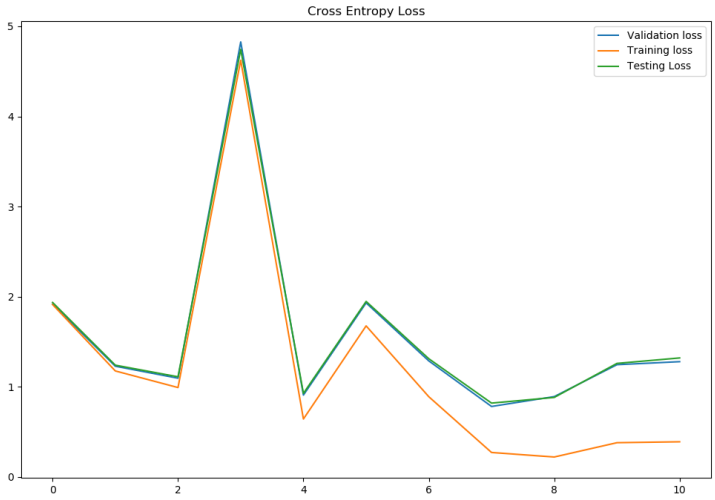
TDT4265 - Computer Vision and Deep Learning Assignment 3

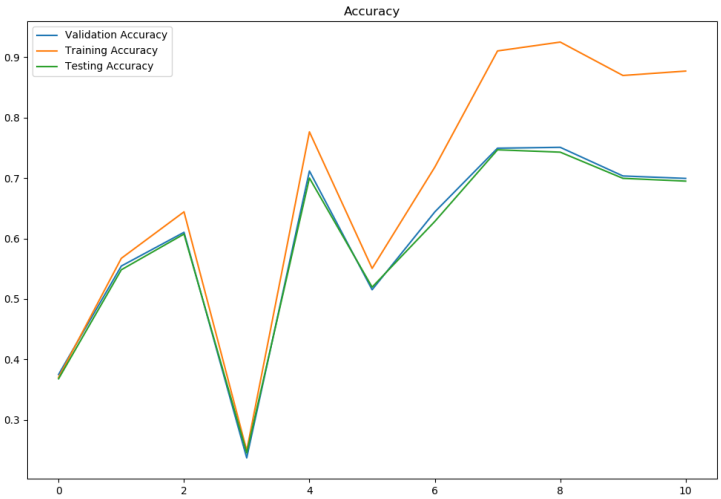
Thomas Aven, Lasse Eggen

February 26, 2019

1 - Convolutional Neural Networks

a) For this task we implemented the class `SimpleModel` as descried in Table 1. We evaluated the loss and accuracy at the end of every epoch, instead evaluating more than once for each epoch, as all the information we want is still contained within the images this way. It is very clear that the model is very unstable, which is due to poor weight initialization. For hyperparameters we used, the learning rate was set to $5 * 10^{-2}$, batch size to 64, and the optimizer in use was vanilla SGD.

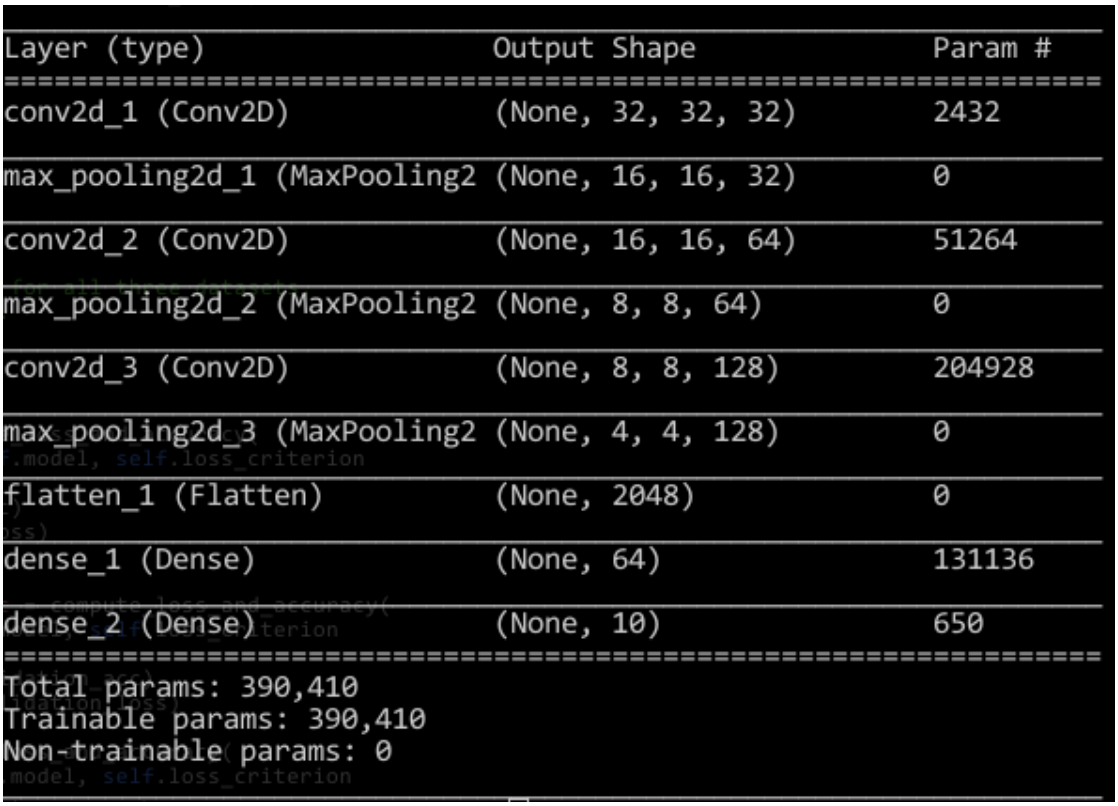




b) The final accuracies observed were 87.7% for the training set, 69.9% for the validation

set, and 69.5% for the test set.

c) We tried summing the parameters with both PyTorch and Keras, and found them both to give a total of 390,410 parameters. Keras gives a much prettier output with `model.summary()` (PyTorch may obviously have a prettier summary that we didn't figure out; we used `p.numel() for p in self.model.parameters() if p.requires_grad`):



Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 32)	2432
max_pooling2d_1 (MaxPooling2	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	51264
max_pooling2d_2 (MaxPooling2	(None, 8, 8, 64)	0
conv2d_3 (Conv2D)	(None, 8, 8, 128)	204928
max_pooling2d_3 (MaxPooling2	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 64)	131136
dense_2 (Dense)	(None, 10)	650

Total params: 390,410
Trainable params: 390,410
Non-trainable params: 0

2 - Deep Convolutional Network for Image Classification

a) For this task, we first focused on getting any network to reach 75% at all, which was not particularly hard – as can be seen in `GoodModel`. This is a fairly standard convolutional net (we found the best way to report the architecture of a network to simply use the `__str__` part of `nn.Module`, we hope that this is sufficient):

The optimizer in use is Adam, with default betas of $(0.9, 0.999)$, eps of $1 * 10^{-8}$ and a weight decay of 0.001. The learning rate used is $5 * 10^{-4}$, the batch size is set to 64, and the maximum amount of epochs is left at 10. Weights are initialized using Xavier initialization. Regularization is present in the image above – this should be enough to be able to replicate our results.

After reaching 79% with this model, we set out to improve it even further. We were seeing some extreme overfitting, and thus decided to try improving on this. `GoodestModel` is our attempt at this, which can be seen below:

Initially we wanted to pursue methods similar to what's described in a fairly recent paper on Super Convergence with CLR (cyclical learning rates), and a large maximum learning rate (taken from `https://arxiv.org/abs/1708.07120`), but instead ended up with just *expanding* the original `GoodModel` with extra layers and additional measures to improve accuracy. In hindsight, we realize that we should've split the model into self-contained layers, e.g. `conv1`, `conv2`, etc. to improve on readability.

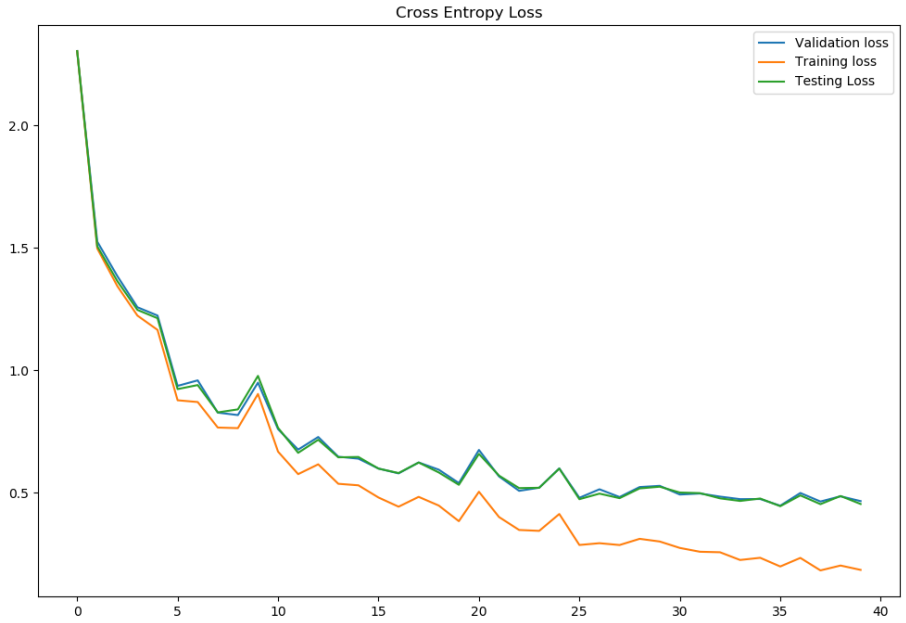
The differences in convolutional layers are not particularly big from the previous model; it is longer and wider, and some extra measures like dropout are added. This model uses the same hyperparameters as for `GoodModel` described above, the only difference being a learning rate of $6 * 10^{-4}$ instead. This shows how changes to the actual model also improved the success, such that it is not just fine tuning of hyperparameters that showed improvement. Note that the model ended up closely resembling the VGG-type network (e.g. VGG-16), which happened quite naturally as we slowly expanded the network. The

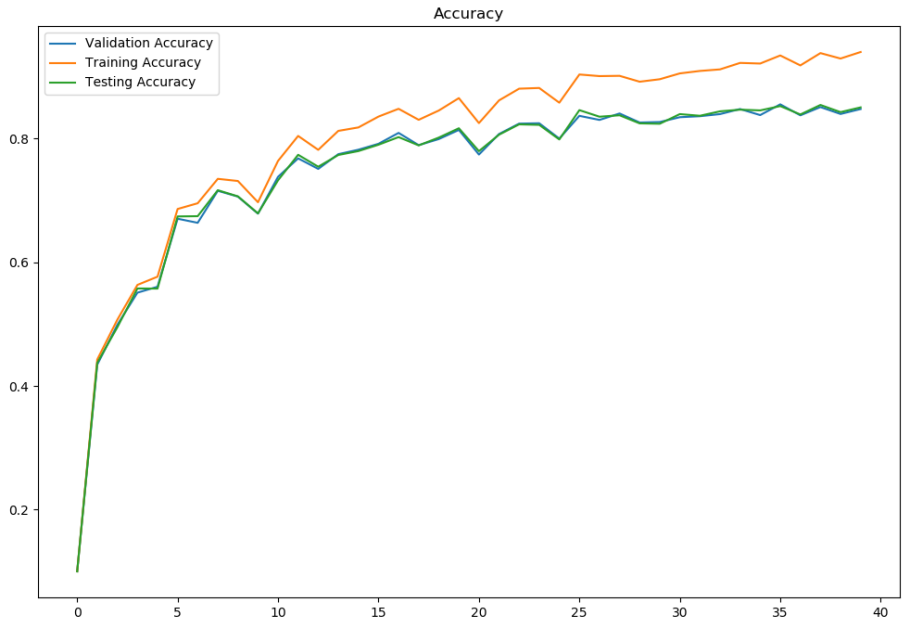
size of the layers of the feedforward are set to 1024, which is the mean of the output of the last CNN layer and the output of the FFNN itself, giving $(2048 + 10)/2$ neurons. This is a common size chosen for this type of network.

b) Shown below are the observed values for our network models.

	GoodModel	GoodestModel
Train loss	0.107576	0.185432
Val loss	0.716263	0.466123
Test loss	0.744095	0.454339
Train acc	96.8%	94.0%
Val acc	80.1%	84.8%
Test acc	79.4%	85.1%

c, d) Below are the losses and accuracies of our *best model*. For these plots, we validated our model four times every epoch, giving a total of 40 validation runs (which actually took as much time as the training itself, which only takes about a minute for each epoch). Overfitting is still present, but to a lesser extent.





e) Changing from `MaxPool2d` to strided convolutions did not work, and actually showed a significant drop in performance (it's hard to pinpoint exactly why – as it seems that researchers are quite divided on the topic as well). We changed the filter size from 5x5 to 3x3, as this is a very common setup for convolutional nets. The most significant change for our model was to include batch normalization, which was the first step to reach the high 70% (batch normalization normalizes the outputs of a layer before forwarding it forward to the next layer, such that we force the input of all layers to have approximately the same distribution).

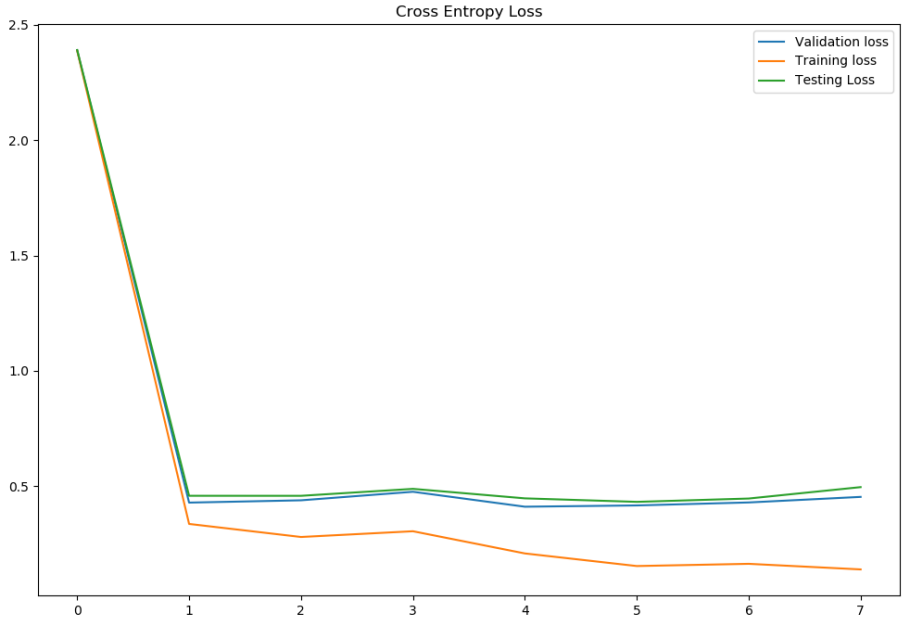
Changing from SGD to Adam also showed a very significant improvement, which took us from about 60% in 10 epochs to mid 70%, and fine-tuning the learning rate seems to be one of the most efficient ways to improve learning. Fine-tuning dropout was the last addition, which moved our network towards the 85% accuracy observed. Here we increase the dropout from `0.1` to `0.5` as we move along the convolutional layers, such that the value used for the feed forward net is fairly high for regularization purposes. Xavier intialization of weights improved the stability of training quite a lot, and also improved the convergence rate substantially.

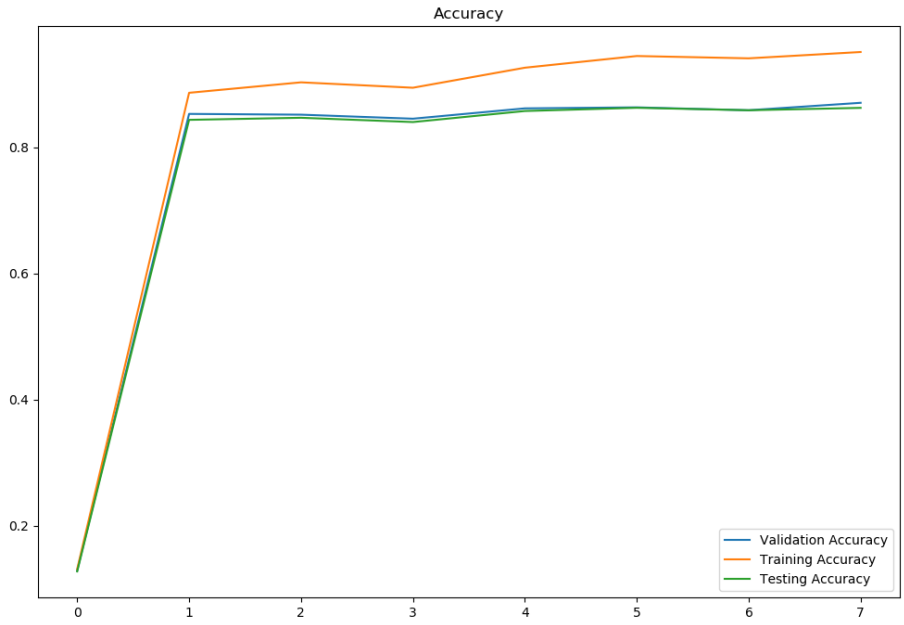
3 - Transfer Learning with ResNet

a) The ResNet transfer learning is implemented in the class `ResNet18`. Hyperparameters used are a batch size of 32, the Adam optimizer in the same manner we used it for

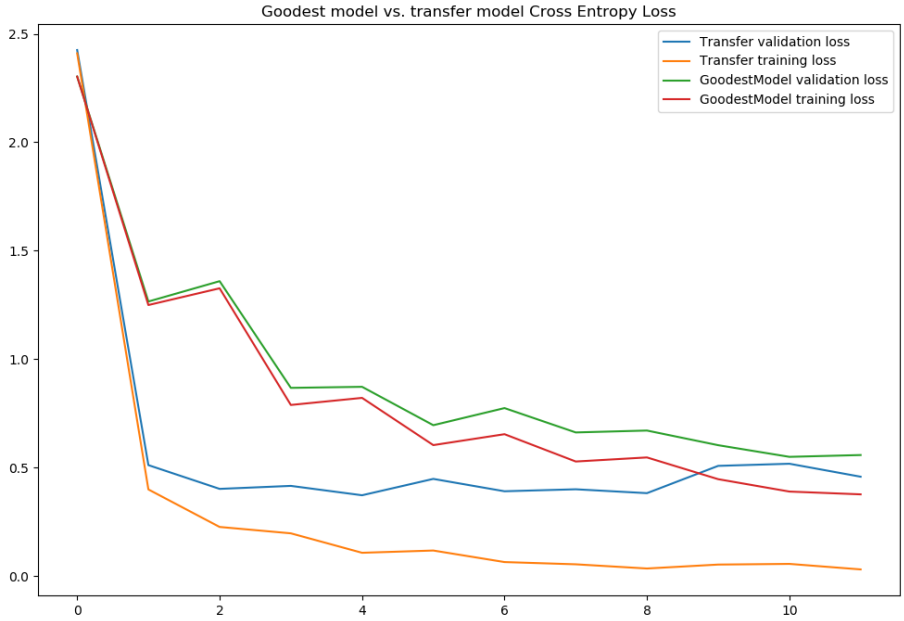
previous training (described above) with a learning rate of $5 * 10^{-4}$. No data augmentation was used during training, but the training examples are chosen at random by using `SubsetRandomSampler`, which was already present in `dataloaders.py`.

b, c) Below are losses and accuracies for the ResNet18-transferred model. We hit early stopping quite quickly. Note also that the network converges extremely fast (so fast that we should perhaps have evaluated the network even more often just to get the gist of how fast it's actually happening). After converging quite fast, the improvement flattens out. This is as we expected from transfer learning, as we are only fine tuning an already well-trained network.

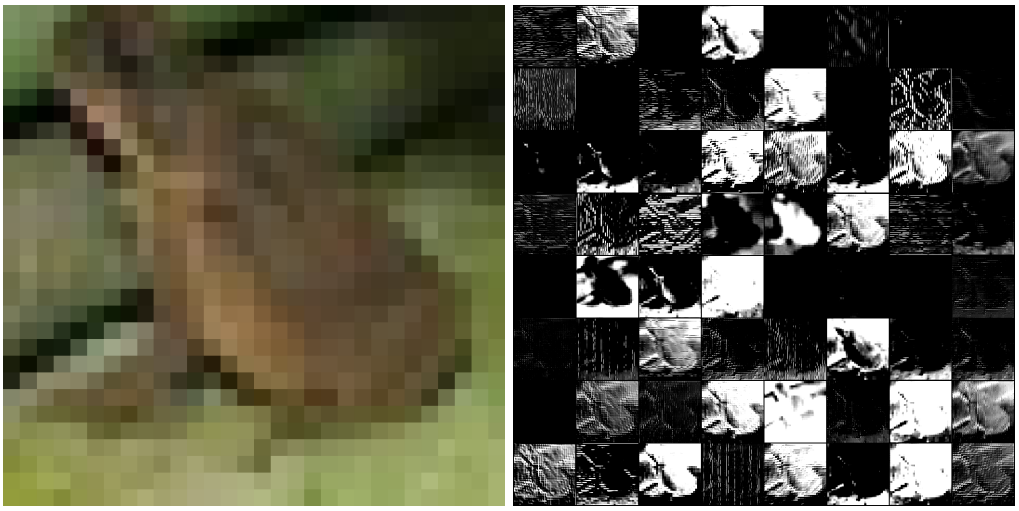




d) Observed below is the plot for loss of `GoodestModel` vs. the ResNet transfer taught model. The primary difference is their convergence rate, as already mentioned above. In addition, the ResNet model is overfitting completely, reaching as high as about 99% on the training set (the validation loss is also tanking slightly as the overfitting becomes more pronounced).



e) For this task, we used one of the frog images from the CIFAR10 set to visualize the activations of the filters of the CNN. We also used the code from the assignment lecture to produce the visualiation.



It is not always clear what you are seeing when looking at filter activations within a CNN, but it may often give indications as to what is being weighted. In this case, we are seeing that the first layer of the CNN works as expected; as a feature extractor. It is possible to notice that the filter is segmenting features (e.g. the contour) of the frog, and

attempting to extract them from the images. It is also possible to see that some features are extracted as horizontal features, while others are more vertical.

f) For this task we decided to visualize the last convolutional layer of the already trained ResNet18 for 1000 classes, instead of the model we did transfer learning on. This is because we wanted to do the visualization while we were waiting for training, and suspected that the small 8x8 pixel activations would be quite hard to interpret anyway. The 512 different 8x8 activations of this filter are all quite obscure to us, but if you see something interesting please let us know! (Really, we would love to understand more of what's going on here.)

Below are visualizations from some of the filters, zoomed in such that they are at least a little bit presentable.

g) Below are the 7x7 weights for all 64 filters of the first convolutional layer (for all three color channels). There are quite a few kernels in play here, so what would be interesting to look at? As the weights are a big part of determining the activations we saw in task **e)**, we might try to look for filters that provide the somewhat horizontal and vertical feature extractors. We could perhaps look for kernels that look similar to a Sobel filter. In general we are looking at kernels that are segmenting the images as seen before, seemingly to provide the more general characteristics of the image (so that later layers will zoom in on more specific features).