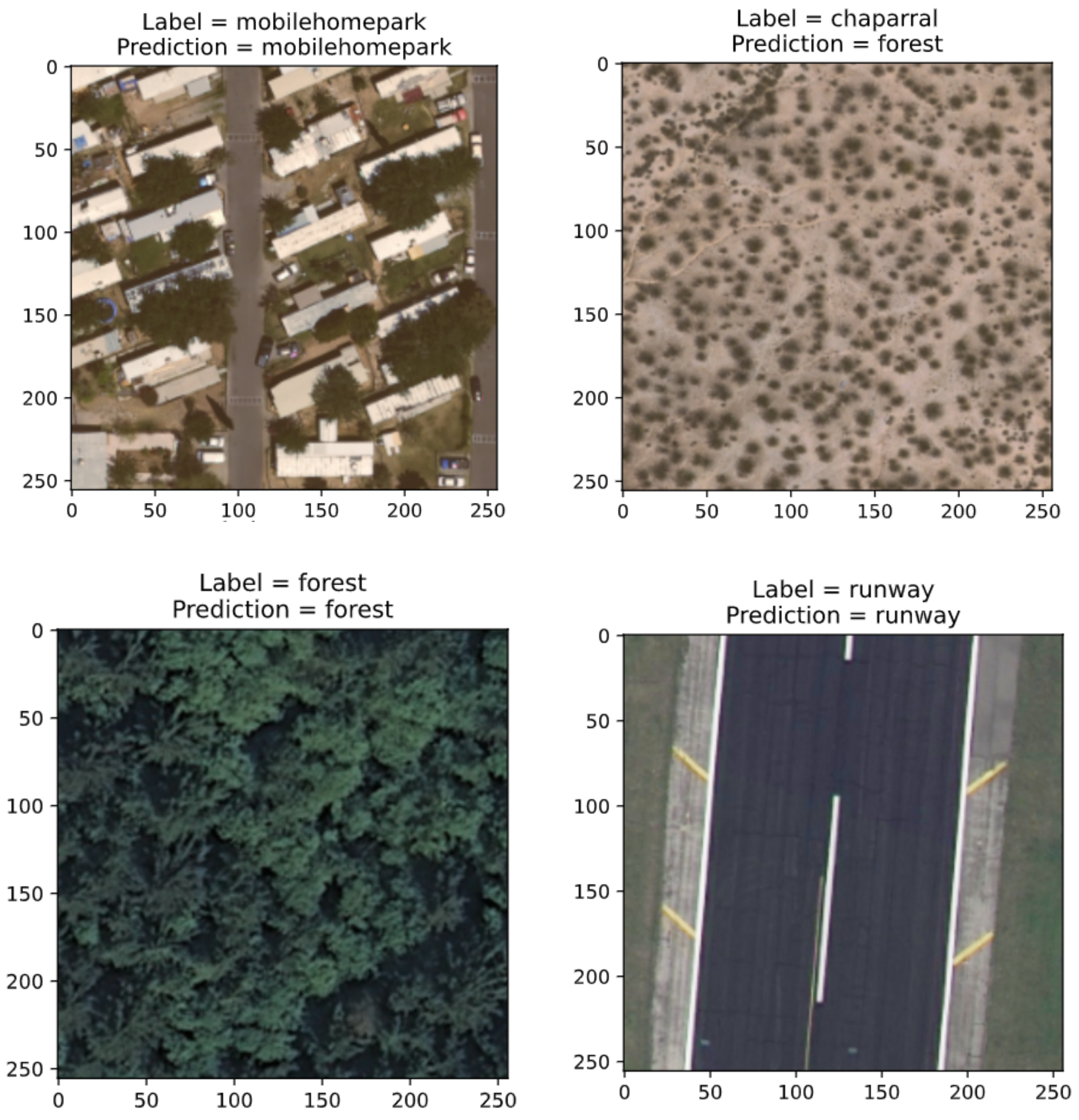
Single- and multi-label classification of UCM satellite images

This report applies the AlexNet CNN architecture on the UCM dataset and investigates how changes in hyperparameters can optimise our model and lower the test error rate.



General information

Names:	Jorrit van Gils
Student nr.:	1006511
Course:	Deep learning
Project:	Geo-information science

WAGENINGEN
UNIVERSITY & RESEARCH

Introduction

In big data analysis, deep learning is the fastest-growing trend and widely applied in image analysis tasks, segmentation and object detection. Also in the field of remote sensing, where one of the main tasks is to learn the representation of an image, the popularity of deep learning is growing quickly. It is therefore no surprise that many remote sensing scientists are embracing deep learning techniques to classify their satellite images (Zhu et al., 2017).

In particular, deep convolutional neural networks (CNNs), which apply multiple convolution and pooling layers, are able to detect features of different scales and therefore are hopeful in land-use predictions (Hu, F., Xia, Hu, J., & Zhang, 2015). Our objective was to build a single- and multi-label classification model that can characterize land-use. To get optimal results we had to optimise our parameters by choosing initial values for our hyperparameters. The UCM dataset, used in this task was obtained from the USGS National Map Urban Area Imagery collection. As a CNN we tested Alexnet which was known for its low test error (Zhu et al., 2017).

Methods

Dataset

We are working with a UCM dataset that contains 2100 images (Yang & Newsam, 2010). The shape of the images is 3x256x256. Three layers, one for each colour in RGB and the image resolution is 256x256 pixels. These images are evenly divided into 21 different categories. These categories are:

agricultural	forest	overpass
airplane	freeway	parkinglot
baseballdiamonds	golfcourse	river
beach	harbor	runway
building	intersection	sparseresidential
chaparral	mediumresidential	sotragetanks
denseresidential	mobilehomepark	tenniscourt

Besides this categorical division, each image is also classified as one or more labels. The labels are similar to the categories, but we have 17 labels instead of 21 categories. The labels are also more generalised than the categories. The 17 labels are:

airplane	dock	sea
bare-soil	field	ship
building	grass	tanks
cars	mobile-home	trees
chaparral	pavement	water
court	sand	

In this paper we will try to tackle two different cases. We will build a neural network that predicts the classes for the images. We will look at a single-label classification and multi-label classification. For the single-label classification we will use the 21 categories and for the multi-label classification we will use the 17 labels.

Packages

To get a full understanding of our script, we decided to avoid auto machine learning packages and packages directly implemented from the book ‘Dive into Deep Learning’ (Zhang et al., 2020). Basically, we did not use functions that were an abbreviation of a larger function, in particular when we trained and evaluated our model. Three examples of functions that we included in our script are the try_gpu(), accuracy() and evaluate_accuracy_gpu() functions. We also included an Accumulator class. The current script only includes d2l packers when it links to functions that can be considered minor functions. This boils down to the animator and lambda functions from d2l.

Single-label classification

For the first part of the project, the single-label classification, we tried to implement AlexNet and made slight changes to fit our image sizes. We chose AlexNet, because AlexNet achieved excellent performance in the 2012 ImageNet challenge (Zhu et al., 2017) and it was also discussed in the book, Dive into Deep Learning, that was used for the course. The main structure of AlexNet consists of five convolutional layers and three fully connected layers (figure 1). We had to change some minor parameters in the network, due to a difference in image size. Where AlexNet is made for 224x224 images, we work with 256x256 images. We didn't resize our images, because according to the course book, it's not considered a smart practice.

Since our images are different sizes and we also have only 21 classes, we have to change some layers in the network. After the convolutional layers, the network flattens the image. Due to the difference in input size the images in our model have a (1x256x6x6) shape instead of the (1x256x5x5) shape from AlexNet. Therefore we have to change our input features for our first fully connected layer from 6400 to 9216. Since the output layer corresponds to the amount of classes we have to change the output from (1,1000) to (1,21) (figure 2).

Now we need to prepare the images so the network can handle them. To load the data with torch.utils.data.DataLoader, we first split the images and labels into a training and test dataset. Our first step is to extract the correct label from the image names. The second step was a division of index numbers for the training and test dataset. We used a 80%-20% distribution for the training and test data respectively. This means we have 80 images for each class in our training dataset and 20 images for each class in our test dataset. The next step was splitting our images and our labels according to the training and test index numbers. Since we used the same indexes to split the labels and images, they correspond to each other in different dataframes. We implemented a Dataset class to combine our images and the labels together. In this class we also made sure to Resize our images to 256x256, because some images came out in slightly different shapes. Four our final step we needed to load the data and make the data iterable. Therefore we used the previously mentioned DataLoader. Here we also specified our batchsize, which is 128 just like AlexNet. Now the data is ready for the network. To make sure our images had the correct label, we plotted some images for several batches and confirmed the data was loaded correctly (figure 5).

When we trained our model we had to choose a *loss function*, which measured the performance of our classification model. Because we were dealing with a classification task, we chose the binary cross-entropy loss function from Pytorch that analysed probability values ranging from 0 to 1 and compared these with the actual label. Binary because we predict the probability of an image belonging to one single class. In our single-classification task we calculated the *accuracy* by applying a hard prediction boundary by taking the highest prediction probability (y_hat) and compared this with the true label (y). The large dataset with 2100 images required us to send our data to the GPU, so that pixels could be processed in parallel which reduced training time.

We also wanted to add batch normalization in our model to see if we could make our model faster and more stable (Santurkar et al., 2018; Ioffe & Szegedy, 2015). In general batch normalization is done before the activation functions, but we found a lot of discussion in blogs and fora about the difference between batch normalization before or after activation function. We looked for some scientific literature about this difference, but couldn't find clear research on this specific issue. Therefore we decided to extend our basic model two times. Once with batch normalization before the activation function (figure 3) and once with batch normalization after the activation function (figure 4). We focus on the networks with batch normalization because it learns faster, and is more stable compared to our basic model.

Multi-label classification

The second part of the project, the multi-label classification has a comparable approach compared to the single-label classification task. Instead of predicting one class out of the 21 categories per image the model now should be able to predict multiple classes out of the 17 labels per image (Chaudhuri et al., 2017). We will use the same network structure as the single class prediction. The output layer has to be changed from (1,21) to (1,17) to deal with the different amount of classes.

We can use the same image distribution, but not the same labels. The input for multi-labels has to be changed to a tensor with ones for the classes present in the image and zero's for the classes not present in the image. It's also important to make sure all the values in the label tensor are float values. We can use the same index numbers that were used for the training and test data split. Now we can use the same Dataset class to combine our images and the new labels together and load the data with a DataLoader.

For our multi-label classification task, a limitation of our cross entropy loss function was that it could only handle single label outputs. To deal with this we chose the loss function nn.BCE that allowed us to apply a multi-class classification.

Result and discussion

Single-label classification

We trained our basic model with a learning rate of 0.01 and 0.05 for 100 epochs each. At first it looked like a learning rate of 0.05 would give better predictions at a faster rate in comparison to the model being trained with a learning rate of 0.01. However, we quickly found out that a learning rate of 0.05 gave some problems. Where the model starts learning immediately with a learning rate of 0.01 (figure 6), it can take quite some time before the model starts learning anything with a learning rate of 0.05 (figure 7). We shortly looked into a dynamic learning rate that increases after a set amount of epochs, but training the model with a 0.05 learning rate also resulted in catastrophic interference (figure 8). Sometimes the model was able to quickly recover from the event, but most of the time the model continued with a flat line at roughly 0.05 accuracy, the guessing rate. We didn't check if the model was stuck infinitely or it was just because the model sometimes takes a lot of epochs before it starts learning something. Due to these errors we continued with a learning rate of 0.01. The basic model trained for 200 epochs can reach a test accuracy around 0.6 (figure 9). For the first 100 epochs the test accuracy keeps up with the training accuracy, but in the second half of the 200 epochs it's clear that the test accuracy converges while the network keeps improving on the training accuracy till it gets really close to 1.0. We didn't experience real overfitting, because the test accuracy didn't go down, but stays around 0.6.

Our second model is similar to our first model except for the extra batch normalization after the convolutional layer and before the activation function, ReLU. Conveniently batch normalization is used to make models able to learn at a more stable and faster rate. This directly plays into our shortcomings of the basic model. It's unstable or slow. Since it looked like a learning rate of 0.05 would give better predictions for our images, we first tested our updated model with a learning rate of 0.05. The model learns at a much quicker rate now the batch normalization is added (figure 10). The training accuracy skyrockets and approaches 1.0 fast. Sadly, the model loses quite a bit of accuracy. The newer model with batch normalization seems to converge at a test accuracy in a range of 0.2 to 0.4 (figure 10&11. This is significantly lower than our basic model where we can reach accuracies of around 0.6. Therefore we also tested our batch normalization model with a learning rate of 0.01. The updated learning rate resulted in a very similar model with slight changes (figure 12). The models training accuracy goes even faster to 1.0 and the training and test accuracy are more stable. The test accuracy has some peaks above the previous range, it's hard to say if this is the model being better or just random change. We think a test accuracy can be expected to be in the same range of the test accuracy of the model with a learning rate of 0.05. It's also good to note that, just as the basic model, our batch normalization model doesn't really overfit. The test accuracy doesn't decline and keeps itself in most cases within a certain range.

For our third model we made a slight change in computation order. Instead of batch normalization after a convolutional layer and before an activation layer, we put it after the activation layer. We expected rather similar results to our second model. Training the model at a learning rate of 0.01 we see the same pattern as our second model (figure 13). It's a lot more stable than our basic model and also the learning accuracy approaches 1.0 really quick. A big difference however, was the test accuracy. The test accuracy seemed to be structurally higher than our model with batch normalization before the activation function. Training our third model with a learning rate of 0.05 resulted in an even higher test accuracy (figure 14). With a learning rate of 0.01 our third model can easily reach a test accuracy in a range of 0.5 to 0.6. With a learning rate of 0.05 we mainly get values for the test accuracy higher than 0.6 and even crossing 0.7 sometimes. We were surprised by these last results, since we expected the models to behave similarly and have similar results. The models do behave similar, but the only difference is the test accuracy being a bit higher.

Multi-label classification

For the single-label, after fine tuning our model we ended up with quite accurate prediction results for classifications. However, implementing multi-label classification was more challenging. We knew that we had to change our input data and also our loss function. Besides this we weren't sure what we had to change exactly and our time was already limited, so we had to go with trial and error. After we combined our images with the new multi-label tensor, which had float32 values inside, the data was ready for input. We also changed our loss function from nn.CrossEntropyLoss to nn. BCEloss, to deal with the multiple labels. And we also made sure our output layer gave 17 values instead of 21 values. After some small tweaks, we ran into a dimension error in the accuracy calculation. This is where we are still stuck to this day. We checked the shapes of y and y_hat. y has a shape of (128, 17) and y_hat has a shape of (128, 17). The Runtime Error we get is "The size of tensor a (128) must match the size of tensor b (17) at non-singleton dimension 1". It seems that our y_hat does not have the same size as our output y. But we can't figure out how this happened. After this project we would like to find out why this issue occurred and hope to solve it and learn from it so that we know how to perform both methods in the future.

Discussion

We successfully look back on our single-label classification task. We were able to put together several models and make them learn with different parameters and achieved a test accuracy of up to 70%. In this project, we have put a great effort in finding out which parameters were mostly contributing to a lower test error. We found out that learning rate, batch size, nr. of epochs play an important role. Because our project was limited to two weeks, we only experimented with a learning rate of 0.1, 0.05 and 0.1. If we would have had more time we could spend more time on further fine tuning these initial values.

Spending a lot of time on the single label task helped us to really understand what we did in every step. However, because of this we had less time to make the multi-label model work. Unfortunately we also experienced computer issues on tuesday and wednesday of week 2.One computer in particular seemed to struggle a lot with the colab environment. This was already the case with some practicals, but wasn't fixable. Running the exact same code gave CUDA errors 80% of the time on one computer but not on the other computer. This surprised us, but at the same time was also expected since it also happened with some practicals.

To summarise, we have learned a lot of skills mainly because for problems that we ran into during our project, we did not have assistance like we had in the lectures. This stimulated us to find solutions ourselves when we were stuck via Stack overflow, Google or via the help from Google Collab. The next time when we are going to build a CNN we would also implement the image augmentation technique. Including variation in the dataset can help our model to learn better from our images. Because of the challenge to link the correct image to the label, when duplicating the image we decided to not use this method for this script. Other improvements on our current network can be to implement the architecture neural networks that have been designed after AlexNet in 2012 and therefore potentially result in more accurate prediction results.

Implementation

Here we provide the shareable link to our code:
https://drive.google.com/file/d/1a2ZT4Ss7UgTzX7MprgrG8ygxSoh4IP4T/view?usp=sharing

References

Chaudhuri, B., Demir, B., Chaudhuri, S., & Bruzzone, L. (2017). Multilabel remote sensing image retrieval using a semisupervised graph-theoretic method. *IEEE Transactions on Geoscience and Remote Sensing, 56*(2), 1144-1158. https://doi.org/10.1109/TGRS.2017.2760909

Hu, F., Xia, G. S., Hu, J., & Zhang, L. (2015). Transferring deep convolutional neural networks for the scene classification of high-resolution remote sensing imagery. *Remote Sensing*, *7*(11), 14680-14707. https://doi.org/10.3390/rs71114680

Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448-456). PMLR.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. *arXiv preprint arXiv:1805.11604*.

Yang, Y., & Newsam, S. (2010). Bag-of-visual-words and spatial extensions for land-use classification. In *Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems* (pp. 270-279). https://doi.org/10.1145/1869790.1869829

Zhu, X. X., Tuia, D., Mou, L., Xia, G. S., Zhang, L., Xu, F., & Fraundorfer, F. (2017). Deep learning in remote sensing: A comprehensive review and list of resources. *IEEE Geoscience and Remote Sensing Magazine*, *5*(4), 8-36. https://doi.org/10.1109/MGRS.2017.2762307

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2020). Dive into Deep Learning. 2020. *URL https://d2l. ai.*

Figures

Figure 1: AlexNet basic structure

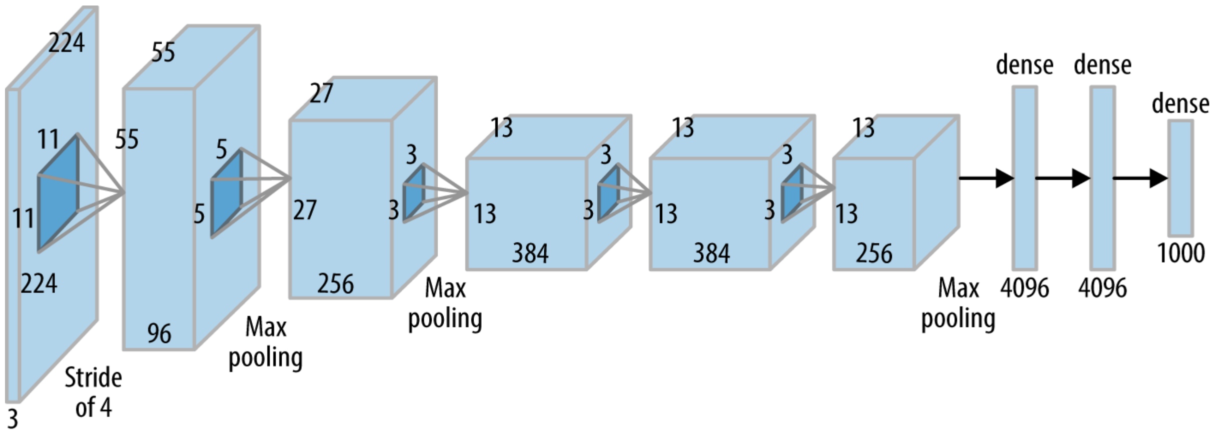
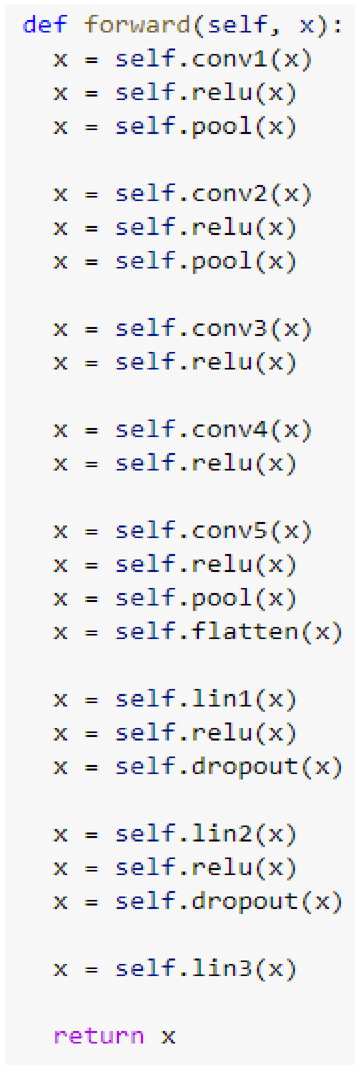
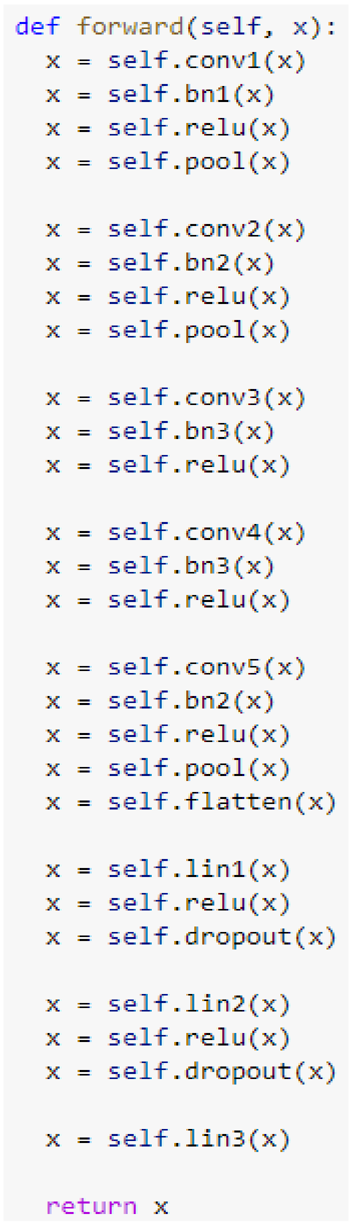


figure 2: basic model



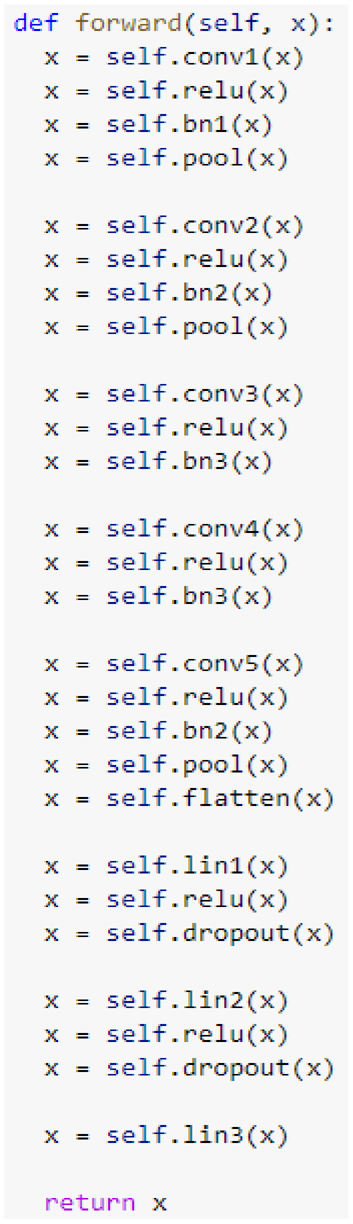
```
def forward(self, x):
    x = self.conv1(x)
    x = self.relu(x)
    x = self.pool(x)

    x = self.conv2(x)
    x = self.relu(x)
    x = self.pool(x)

    x = self.conv3(x)
    x = self.relu(x)

    x = self.conv4(x)
    x = self.relu(x)

    x = self.conv5(x)
    x = self.relu(x)
    x = self.pool(x)
    x = self.flatten(x)

    x = self.lin1(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.lin2(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.lin3(x)

    return x
```

figure 3: batchnorm before ReLU



```
def forward(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.pool(x)

    x = self.conv2(x)
    x = self.bn2(x)
    x = self.relu(x)
    x = self.pool(x)

    x = self.conv3(x)
    x = self.bn3(x)
    x = self.relu(x)

    x = self.conv4(x)
    x = self.bn3(x)
    x = self.relu(x)

    x = self.conv5(x)
    x = self.bn2(x)
    x = self.relu(x)
    x = self.pool(x)
    x = self.flatten(x)

    x = self.lin1(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.lin2(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.lin3(x)

    return x
```

figure 4: batchnorm after ReLU

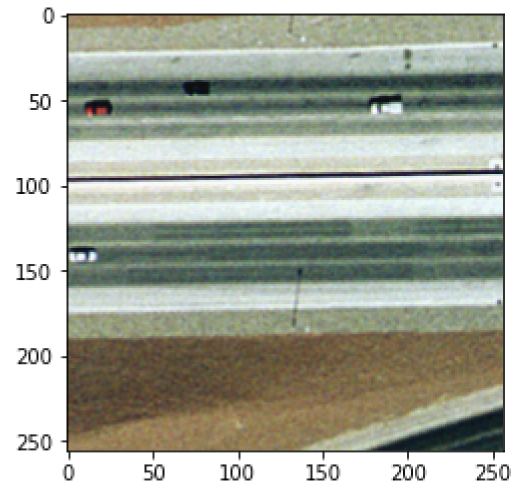


```
def forward(self, x):
    x = self.conv1(x)
    x = self.relu(x)
    x = self.bn1(x)
    x = self.pool(x)

    x = self.conv2(x)
    x = self.relu(x)
    x = self.bn2(x)
    x = self.pool(x)

    x = self.conv3(x)
    x = self.relu(x)
    x = self.bn3(x)

    x = self.conv4(x)
    x = self.relu(x)
    x = self.bn3(x)

    x = self.conv5(x)
    x = self.relu(x)
    x = self.bn2(x)
    x = self.pool(x)
    x = self.flatten(x)

    x = self.lin1(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.lin2(x)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.lin3(x)

    return x
```

Figure 5: Visualize image with label. Check if data is loaded properly

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

sample_id = 0
image = X[sample_id].permute(1,2,0)
imgplot = plt.imshow(image)
print("label = " + str(classes[y[sample_id]]))
```

```
label = freeway
```



Figures, basic model

Figure 6: Basic Model, lr = 0.01

```
loss 1.496, train acc 0.518, test acc 0.395
667.6 examples/sec on cuda:0
```

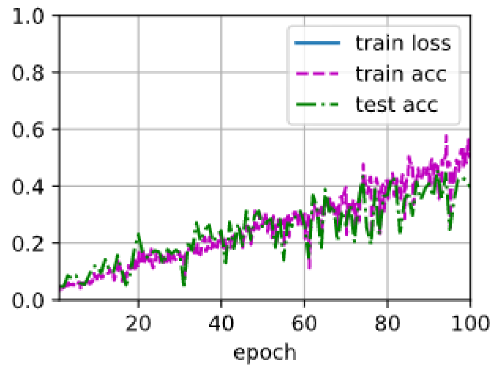


Figure 7: Basic model, lr = 0.05

```
loss 2.164, train acc 0.318, test acc 0.188
689.3 examples/sec on cuda:0
```

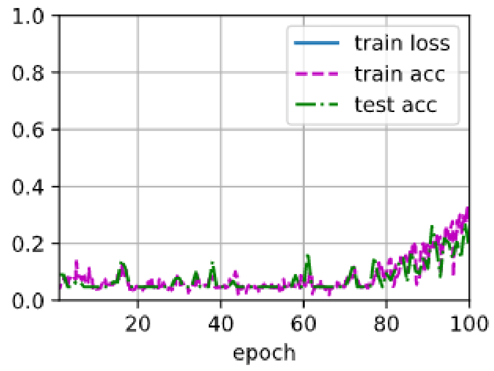


Figure 8: Basic model, lr = 0.05. Catastrophic interference

```
loss 1.693, train acc 0.535, test acc 0.407
336.1 examples/sec on cuda:0
```

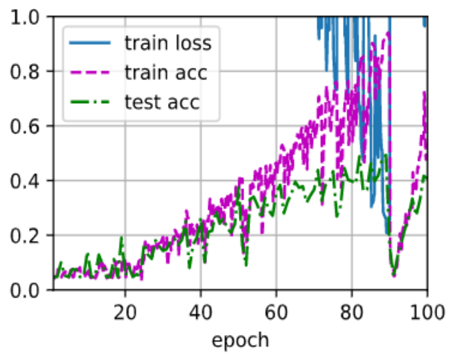
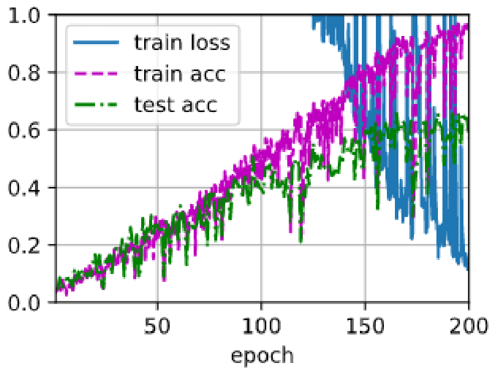


Figure 9: Basic model, lr = 0.01, Converged network

```
loss 0.124, train acc 0.964, test acc 0.571
686.0 examples/sec on cuda:0
```



Figures, batch normalization added before ReLU function model

Figure 10: Batch normalization before ReLU function test accuracy range, lr = 0.05

```
loss 0.025, train acc 0.995, test acc 0.217
606.6 examples/sec on cuda:0
```

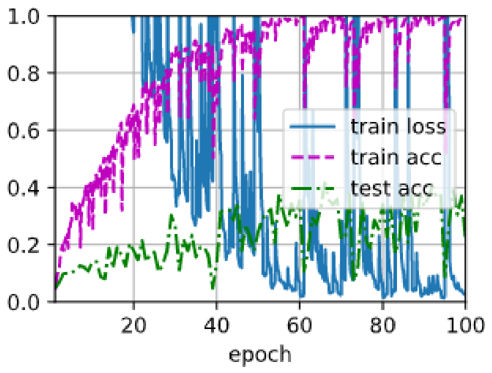


Figure 11: Batch normalization before ReLu function test accuracy range, lr = 0.05

```
loss 0.007, train acc 0.999, test acc 0.376
606.2 examples/sec on cuda:0
```

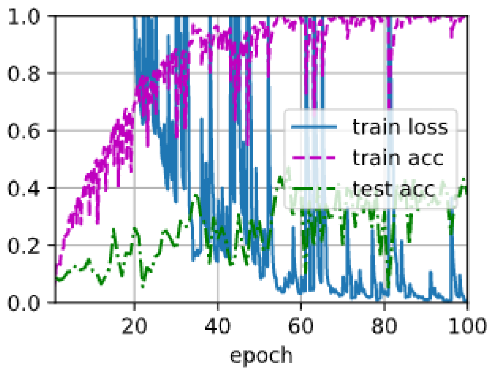
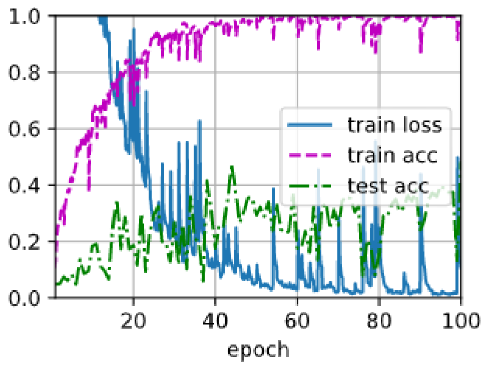


Figure 12: Batch normalization before ReLU function trained model, lr = 0.01

```
loss 0.106, train acc 0.980, test acc 0.483
621.5 examples/sec on cuda:0
```



Figures, batch normalization added after ReLU function model

Figure 13: Batch normalization after ReLU function trained model, lr = 0.01

```
loss 0.018, train acc 0.998, test acc 0.548
636.1 examples/sec on cuda:0
```

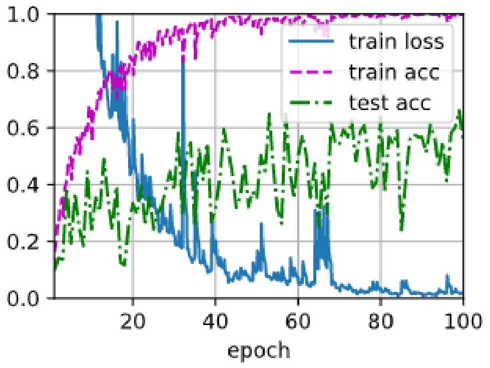
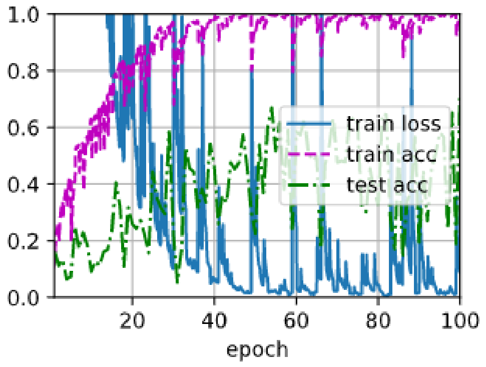


Figure 14: Batch normalization after ReLU function trained model, lr = 0.05

```
loss 0.087, train acc 0.982, test acc 0.710
607.7 examples/sec on cuda:0
```


train loss
train acc
test acc
epoch