Universiteit van Amsterdam

CNNs for Image Classification

Lab Project Part 2

October 24, 2021

Students:
Name: Nicole Sang-Ajang
UvAnetID: 10734155

Name: Haohui Zhang
UvAnetID: 14124262

Name: Xiaoyang Sun
UvAnetID: 13825453

Affiliation:
Master Artifical Intelligence

Lecturer:
Dr. Shaodi You

Course:
Computer Vision 1

Course code:
52041COV6Y

1 Introduction

Up till now several techniques for image recognition and classification problems have been discussed. However, as more challenging problems arise where the objects hold great variability such as recognizing handwritten letters or face recognition a more optimal approach is needed which in return requires a complex recognition system. This is where neural networks come forward which are supposed to model the human brain, i.e. a network that consists of neurons being modeled as so called units that have an input and output. Here we specifically model perceptron neurons which have a binary output. What makes this attractive is that neurons are elementary components thus they perform very simple computations. Each neuron can be connected to another neuron such that it provides the input for the other neuron. From a computational perspective this allows us to construct modular complex networks where information travels from input to output creating a mapping, which when connected correctly are able to do remarkable things.

The main challenge of this assignment is thus how do we construct a neural network and most importantly how do we train such a network to perform the correct mapping when given a task. One of the goals is to find the parameter configuration of the neural network that is gonna perform the mapping from the input to the output with high reliability and accuracy. During this assignment two neural network architectures are chosen to study: an ordinary Two-Layer neural network and a CNN (convolutional neural network) utilizing the LeNet-5 architecture[2]. For this we use the $CIFAR-10$ dataset which has already been pre-processed and prepared. Secondly the pytorch framework is used to implement our neural networks and everything required.

2 Session 1: Image Classification on CIFAR-10

CIFAR-10 for each existing class in the data set 4 example images, i.e. drawn from the training data set, are shown with their corresponding class description. See **figure 6** and **7** in the Appendix page 10 and 11.

2.1 Architecture understanding

TwoLayerNet A two layer fully connected neural network is constructed.

```
class TwoLayerNet(nn.Module):
    # assign layer objects to class attributes
    # nn.init package contains convenient initialization methods
    # http://pytorch.org/docs/master/nn.html#torch-nn-init
    def __init__(self,input_size ,hidden_size ,num_classes ):
        '''
        :param input_size: 3*32*32
        :param hidden_size: decide by yourself e.g. 1024, 512, 128 ...
        :param num_classes:
        '''
        self.input_size = input_size # Num neurons input layer
        self.hidden_size = hidden_size # Num neurons hidden layer
        self.num_classes = num_classes # Num neurons output layer

        # Construct neural network
        super(TwoLayerNet, self).__init__()
        # First fully connected layer
        self.fc1 = torch.nn.Linear(input_size, hidden_size)
        # Second fully connected layer
        self.fc2 = torch.nn.Linear(hidden_size, num_classes)

    # Forward function passing data through the network from input, hidden
    layers between, to output
    def forward(self,x):
        # Flatten x
        x = x.view(x.shape[0], -1)
        # Pass data to fc1
        # Use the rectified-linear (ReLu) activation function in the first
    hidden layer (fc1)
        # Pass the result through fc2
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores
```

Using this two layered neural network we can train our model by dividing the training set into batches. Starting with a model that has random parameters (i.e. weights and biases) assigned to it we predict the outcome of the given batch. We define our cost/loss function as the sum of squared errors. To find our best fit model our goal is to minimize the cost function. I.e. find our parameters such that the sum of squared errors is at its local minimum. For this we use the gradient descent method through which we converge into the local minimum. During each iteration gradients of all variables with regard to loss are computed using backpropagation since this is computationally less expensive. Using these gradients we can update the parameters of our model accordingly and move on to the next iteration, i.e. batch.

ConvNet A convolutional neural network is constructed.

```
class ConvNet(nn.Module):
    # Complete the code using LeNet-5
    # reference: https://ieeexplore.ieee.org/document/726791
    def __init__(self, input_channels=3, num_classes=10):
        super(ConvNet, self).__init__()
        self.c1 = nn.Conv2d(input_channels, 6, 5) # The first convolutional
    layer
        self.s2 = nn.AvgPool2d(2, stride=2) # A subsampling/pooling layer
        self.c3 = nn.Conv2d(6, 16, 5) # The second convolutional layer
        self.s4 = nn.AvgPool2d(2, stride=2) # The second pooling layer
        self.c5 = nn.Linear(16*5*5, 120) # The last convolutional layer
        self.f6 = nn.Linear(120, 84) # The first fully-connected layer
        self.output = nn.Linear(84, num_classes) # The last dense layer

    # Forward data x through CNN
    def forward(self, x):
        x = torch.tanh(self.c1(x))
        x = self.s2(x)
        x = torch.tanh(self.c3(x))
        x = self.s4(x)
        x = x.view(-1, self.num_flat_features(x))
        x = torch.tanh(self.c5(x))
        x = torch.tanh(self.f6(x))
        x = torch.sigmoid(self.output(x))
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]
        num_features = 1
        # torch.Size([16, 5, 5])
        for s in size:
            num_features *= s
        return num_features
```

Our CNN utilizes the LeNet-5 structure. The LeNet-5 CNN architecture is made up of 7 layers. The layer composition consists of 3 convolutional layers, 2 subsampling layers and 2 fully connected layers[2]. For the hidden layers the tanh function is used as the activation function. For the last dense layer the sigmoid function is used as the activation function.

Architecture understanding Q2.2.2 The first convolutional layer has 6 kernels (each one for one feature) of size 5×5 and the stride of 1 given an one channel, i.e. grayscale, image. However, we feed this network color images. Assuming the terms kernel and filter are not used interchangeably the kernel size remains the same. Nevertheless, since we need to process a color image, i.e. a three channel image, the network requires a filter consisting of 3 kernels stacked together. Hence, we get a collection of kernels. Filter size is therefor size 5x5x3.

Parameters are the trainable elements in a model such as weights and biases which contribute to model's predictive power and can be changed during the back-propagation process. In neural networks the number of parameters of a layer can be determined according to the architecture of the layer inside the network. Using the preceding layer's neurons and the current layer's neurons we can determine the number of parameters (total weights and biases) of "F6" which comes down to 400 * 120 + 120 biases = 48120 parameters. Compared to the

other layers this layer has the highest number of parameters because it is fully connected, i.e. every neuron is connected to every other neuron.

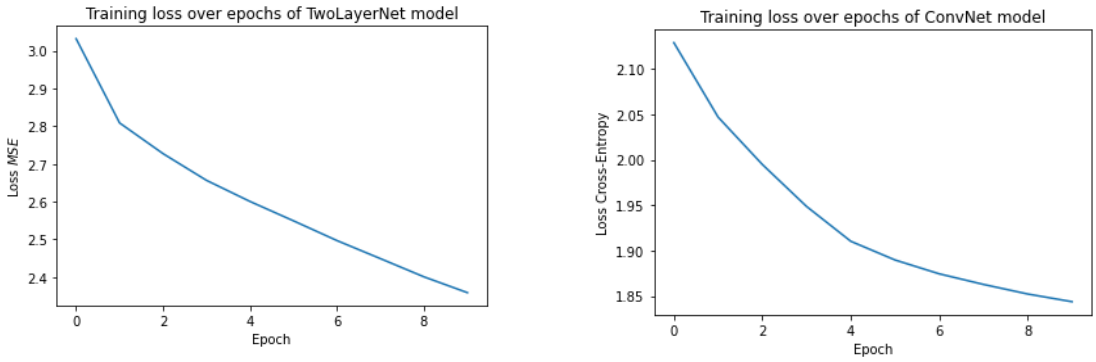
2.2 Preparation of training

We previously used the $CIFAR10$ data set in which an important step, i.e. data preparation, had already been done. In order to learn how to properly construct this we implement the processes of data preparation ourselves complementing $CIFAR10\_loader$, $transform$ and $optimizer$. According to $< train >$ being set to True or not we load in batches from the training or test data set. Secondly, we must be able to handle the following transformation:

```
transform_train = transforms.Compose([transforms.Resize((32, 32)), # To make
    sure of proper size
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5, 0.5, 0.5),
    (0.5, 0.5, 0.5))]) # To speed up learning process
```

Transformation abilities comprise of: reshaping images to 3x32x32, transforming data structure to tensor and normalization of the input. Normalization in neural networks is desired to ensure that the batch of images have a mean of 0 and a standard deviation of 1. Normalizing the data generally leads to faster convergence thus speeding up learning.

Learning curve Just for observation the $TwoLayNet$ en $ConvNet$ their learning curve is shown are the number of epochs ran increases[1]:



(a) TwoLayerNet model

(b) ConvNet model

Figure 1: Learning curve, i.e. minimizing the cost function, on the whole training data set set as epochs ran increases.

Also interesting to see is how during gradient descent method the model adapts increasingly to the sample. As the sample is shown increasingly to the network it becomes more accurate at predicting the output of the sample **(see figure 2)**. However, one can image that this is not desirable since training it too specifically causes overfitting and the network to not be generalizable, i.e. have a high accuracy on unseen data.

[1]Due to computation bottlenecks we used the PyTorch tools implementation instead.

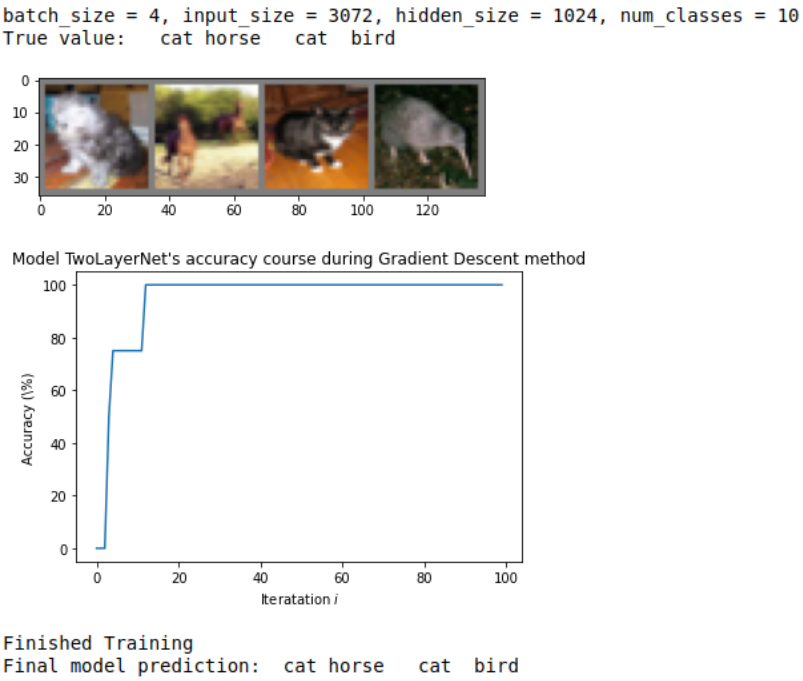
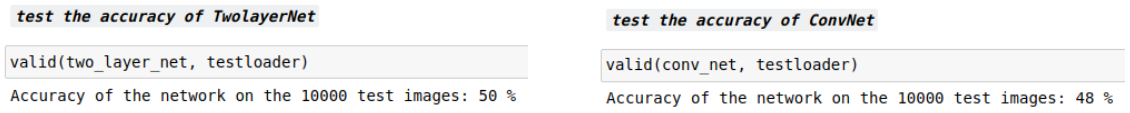


Figure 2: Accuracy of our model on training batch applying the gradient descent method iteratively.

2.3 Setting up the hyperparameters

Initially the accuracy on unseen data is as follows:



(a) TwoLayerNet model (b) ConvNet model

Figure 3: Initial, i.e. with initial hyperparameters, performance.

With hyperparameters we can fine-tune our model, i.e. control the learning process by determining the network structure or determining how the model is trained. Whether optimal parameters are chosen can determined by validating our model using unseen data of a validation set until we reach an acceptable level. We aim to achieve high accuracy here. Our models provide us with the following hyperparameters:

Two Layer Neural Network

lr (learning rate) η is the scalar with which the gradient descent is multiplied. In other words it determines how much you are going to descent and consequently how fast learn. However, setting this value too high might cause the cost function to overshoot bouncing around and not ending up in the correct local minimum, affecting the accuracy of our resulting model. The optimal value would therefor be not too low (since it then converges slow) and not too high.

num_layers increasing the number of layers can increase accuracy depending on the problem. But this should be done in consideration e.g. the sigmoid and hyperbolic tangent activation function are affected by the networks with too many layers.

hidden_size is the number of units in the hidden layer. A too little number of units causes underfitting whereas a too high number of units causes overfitting. Hence, it makes sense to increase the number of units until accuracy no further improves.

epochs is the number of times the whole training data set is shown to the network while training. Increasing epochs of course increases accuracy, however the model increasingly adapts to the training data which at some point will cause overfitting and thus decreases accuracy on the validation data set. This turning point of increasing to decreasing accuracy of the validation data makes a good threshold to determine the optimal number of epochs.

CNN

lrt (learning rate)

num_layers

hidden_size

epochs

batch_size a small batch, e.g. within a size range of 16 to 128, is usually preferable in the learning process of a CNN. We should note that CNN's are sensitive to batch size.

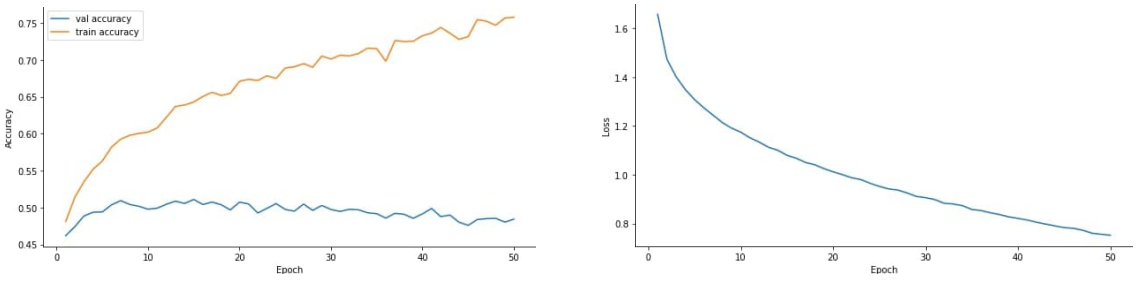
activation function can change that but since we adhere the LeNet-5 structure[2] it is not desirable besides the tanh function is preferred over the logistic sigmoid since this speeds up training. The ReLu function can replace a sigmoid function as the activation function, however, the question remains: "Should we?" but might be interesting to experiment with.

dropout for regularization dropout is a preferable regularization technique to avoid overfitting. The method simply drops out units in neural network according to a desired probability.

Of course we can manually tune the hyperparameters and empirically find our optimal values. But as computer science states: "Everything that can be automated should be automated.". There are two generic approaches to sampling search candidates. Grid search exhaustively search all parameter combinations for given values. This however seems more like a brute force approach in terms of computational efficiency. Although grid search is useful in many machine learning algorithms, it is not desirable for hyperparameter tuning in deep neural network because as the number of parameters increases computation grows exponentially. Random search samples a given number of candidates from a parameter space with a specified distribution. It is also helpful to combine it with some manual hyperparameter tuning based on prior experience. Random search is hereby preferred[1] and can be implemented.

Q2.4.2 You can also modify the architectures of these two Nets. When we speak in terms of performance as the accuracy of the model on unseen data, generally speaking more layers means the neural network can describe the problem better, i.e. handle more complex problems. Nevertheless, increasing it too much in relation to the complexity of the problem (i.e. data being presented) might cause overfitting, however since we only add two layers here it probably will improve performance.

TwoLayNet hyperparameter tuning.



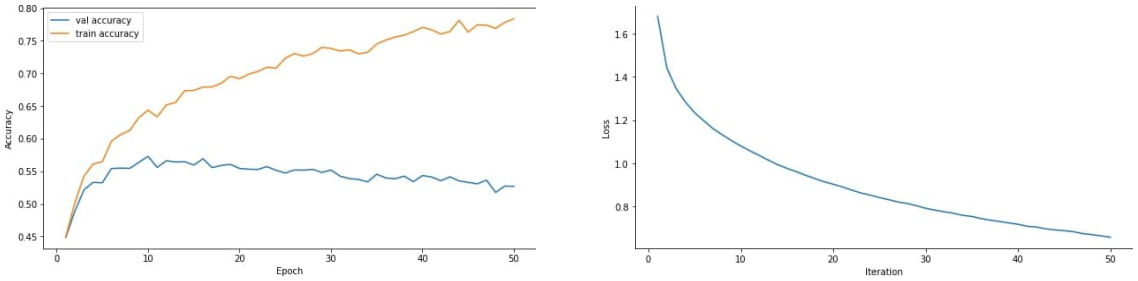
(a) Accuracy (b) Loss on test set

Figure 4: TwoLayNet

```
Accuracy of the network on the 10000 test images: 51 %
Accuracy of plane : 58 %
Accuracy of   car : 63 %
Accuracy of  bird : 38 %
Accuracy of   cat : 24 %
Accuracy of  deer : 40 %
Accuracy of   dog : 43 %
Accuracy of  frog : 65 %
Accuracy of horse : 53 %
Accuracy of  ship : 69 %
Accuracy of truck : 53 %
```

```
(lr 0.0004, batch_size 64, hidden_size 52): val_acc 0.474 train_acc 0.494
(lr 0.0002, batch_size 4, hidden_size 60): val_acc 0.477 train_acc 0.507
(lr 0.0061, batch_size 32, hidden_size 64): val_acc 0.361 train_acc 0.381
(lr 0.0002, batch_size 16, hidden_size 87): val_acc 0.486 train_acc 0.518
(lr 0.0004, batch_size 32, hidden_size 70): val_acc 0.477 train_acc 0.508
(lr 0.0003, batch_size 8, hidden_size 65): val_acc 0.483 train_acc 0.518
(lr 0.0032, batch_size 4, hidden_size 97): val_acc 0.319 train_acc 0.328
(lr 0.0014, batch_size 32, hidden_size 99): val_acc 0.458 train_acc 0.494
(lr 0.0024, batch_size 64, hidden_size 62): val_acc 0.470 train_acc 0.494
(lr 0.0011, batch_size 16, hidden_size 67): val_acc 0.468 train_acc 0.500
(lr 0.0091, batch_size 64, hidden_size 89): val_acc 0.348 train_acc 0.355
(lr 0.0007, batch_size 16, hidden_size 52): val_acc 0.459 train_acc 0.486
(lr 0.0001, batch_size 64, hidden_size 54): val_acc 0.452 train_acc 0.461
(lr 0.0007, batch_size 4, hidden_size 91): val_acc 0.455 train_acc 0.485
(lr 0.0003, batch_size 4, hidden_size 57): val_acc 0.479 train_acc 0.509
(lr 0.0008, batch_size 16, hidden_size 96): val_acc 0.481 train_acc 0.516
(lr 0.0022, batch_size 4, hidden_size 61): val_acc 0.353 train_acc 0.364
(lr 0.0002, batch_size 4, hidden_size 94): val_acc 0.485 train_acc 0.522
(lr 0.0002, batch_size 8, hidden_size 87): val_acc 0.486 train_acc 0.524
(lr 0.0022, batch_size 32, hidden_size 85): val_acc 0.451 train_acc 0.485
```

ConvNet hyperparameter tuning.



(a) Accuracy (b) Loss on test set

Figure 5: ConvNet

```
Accuracy of the network on the 10000 test images: 57 %
Accuracy of plane : 69 %
Accuracy of   car : 70 %
Accuracy of  bird : 39 %
Accuracy of   cat : 37 %
Accuracy of  deer : 45 %
Accuracy of   dog : 47 %
Accuracy of  frog : 76 %
Accuracy of horse : 62 %
Accuracy of  ship : 67 %
Accuracy of truck : 56 %
```

```
(lr 0.0005, batch_size 32, hidden_size 83): val_acc 0.519 train_acc 0.544
(lr 0.0010, batch_size 64, hidden_size 69): val_acc 0.550 train_acc 0.576
(lr 0.0008, batch_size 64, hidden_size 56): val_acc 0.522 train_acc 0.538
(lr 0.0011, batch_size 32, hidden_size 65): val_acc 0.544 train_acc 0.579
(lr 0.0001, batch_size 64, hidden_size 57): val_acc 0.433 train_acc 0.435
(lr 0.0001, batch_size 16, hidden_size 79): val_acc 0.480 train_acc 0.488
(lr 0.0037, batch_size 64, hidden_size 95): val_acc 0.525 train_acc 0.566
(lr 0.0003, batch_size 4, hidden_size 87): val_acc 0.536 train_acc 0.576
(lr 0.0005, batch_size 64, hidden_size 99): val_acc 0.516 train_acc 0.539
(lr 0.0002, batch_size 4, hidden_size 63): val_acc 0.534 train_acc 0.553
(lr 0.0003, batch_size 4, hidden_size 76): val_acc 0.551 train_acc 0.588
(lr 0.0037, batch_size 64, hidden_size 68): val_acc 0.508 train_acc 0.539
(lr 0.0002, batch_size 8, hidden_size 74): val_acc 0.523 train_acc 0.543
(lr 0.0014, batch_size 16, hidden_size 63): val_acc 0.551 train_acc 0.595
(lr 0.0014, batch_size 8, hidden_size 70): val_acc 0.534 train_acc 0.578
(lr 0.0001, batch_size 64, hidden_size 73): val_acc 0.454 train_acc 0.451
(lr 0.0002, batch_size 4, hidden_size 56): val_acc 0.524 train_acc 0.551
(lr 0.0002, batch_size 4, hidden_size 86): val_acc 0.529 train_acc 0.558
(lr 0.0008, batch_size 16, hidden_size 91): val_acc 0.541 train_acc 0.574
(lr 0.0050, batch_size 8, hidden_size 98): val_acc 0.100 train_acc 0.100
```

Two Layer Neural Network vs. CNN LeNet-5 Looking at the architecture it is noticable that the Two Layer Neural network contains two hidden layers which are fully connected. Contrary the CNN consists of convolutional layers. The asset of convolutional layers over fully connected layers is that they describe a narrower range of features than fully-connected layers. An unit in a fully connected layer is connected to every unit in the

preceding layer, and consequently can change when any of the preceding layer's units changes. On the other hand an unit in a convolutional layer is only connected to "local" neurons from the preceding layer within the width of the convolutional kernel. Hence, the units from a convolutional layer are insensitive to the activations of most of the neurons from the preceding layer. This is useful when most of the information we are interested in is local and "non-local" information is considered noise. In image classification this is the case. We can therefor presume the CNN will outperform the $TwoLayNet$ regardless.

Conclusion

During this assignment we learned that when working in this field it is worth investing in hardware specifications that can run CUDA or either invest in cloud computing such as AWS. My laptop did not like this assignment at all. However, in principle and theoretically we did like the assignment. CNN's are a very interesting area especially in image classification since they are able to handle complex problems. Besides, it is very useful to experiment and play around with neural networks to fundamentally understand the architecture throughout and maybe eventually construct our own appropriate neural network to solve a certain complex problem which has not been solved yet.

References

[1] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[2] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

Appendices

A Results

Figure 6: 4 example images, i.e. drawn from train set, for the classes 1-5 per existing class.

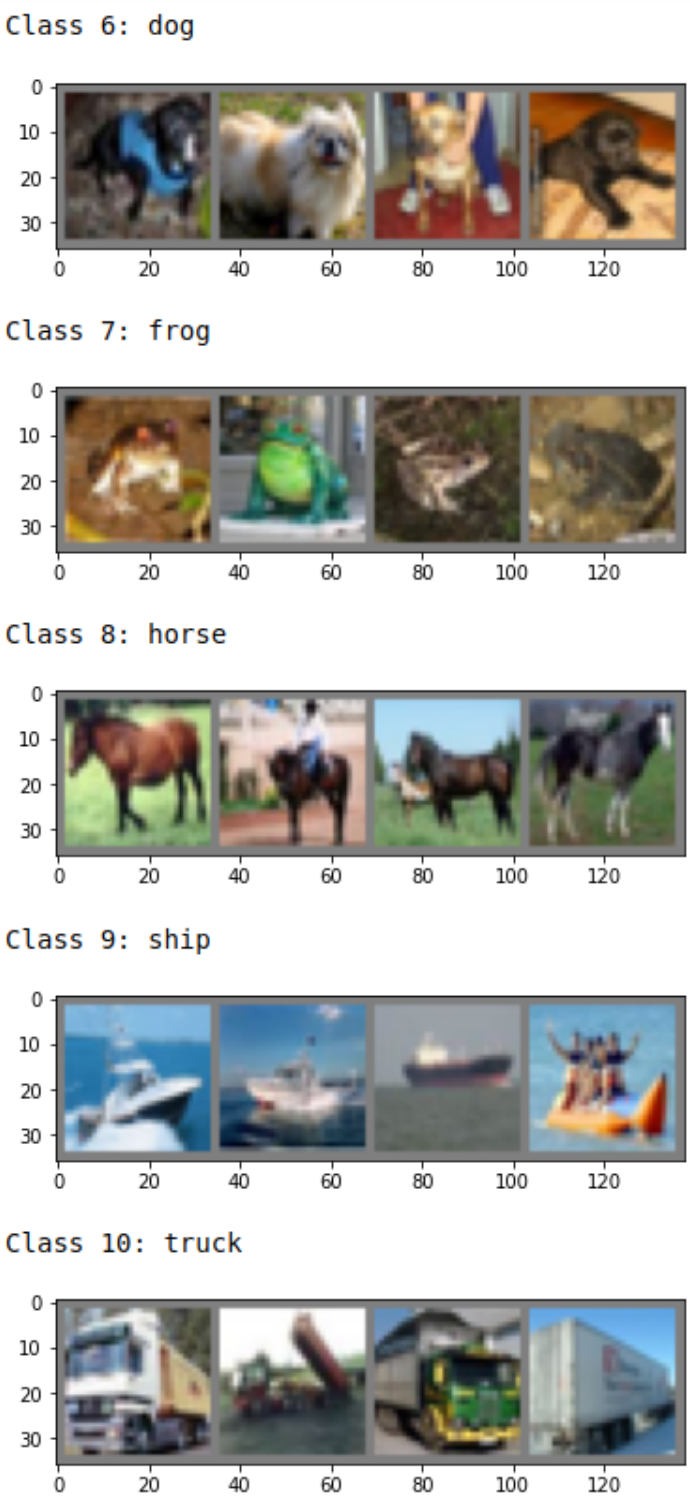


Figure 7: 4 example images, i.e. drawn from train set, for the classes 6-10 per existing class.