Deep Learning

Homework 2

Ludovisi Linda
s278221

May 2020

Academic Year 2019/2020

Contents

1 Homework Description **3**

2 Data Preparation **3**
2.1 Creating the Dataset 3
2.1.1 _init_ method 3
2.1.2 _getitem_ and _len_ methods 4

3 AlexNet: Training from scratch **5**
3.1 Creating the validation set 5
3.2 Implement model selection with validation 5
3.3 Implement hyperparameter tuning 6

4 AlexNet: Transfer Learning **8**
4.1 Training the whole initialized network 8
4.2 Training only some layers of the network 9
4.2.1 Training only the fully connected layers 10
4.2.2 Training only the convolutional layers 10

5 Data Augmentation **11**

6 Beyond Alexnet **12**
6.1 VGG-11 with batch normalization 12

1 Homework Description

The aim of the project is to train a Convolutional Neural Network for image classification. In the first section of the homework the focus will be on the AlexNet neural network; at the end other models will be tried out.
The dataset on which the model is trained and evaluated is Caltech-101: it contains images from 101(+1) classes. The background class is filtered out.

2 Data Preparation

In image data, the data is big enough and it's not feasible to load all the images in memory. For this reason, we need to use dataloaders, which make it possible to load images as we pass through the data and will only load few images in memory.
The most important argument of DataLoader constructor is dataset, which indicates a dataset object to load data from. Our dataset is a map-style dataset: it implements the *__getitem__()* and *__len__()* protocols, and represents a map from indices to data samples. Our Caltech dataset, when accessed with dataset[idx], can read the idx-th image and its corresponding label.
The dataset folders, along with the custom class and the files for the split are located in the GitHub repository.

2.1 Creating the Dataset

A custom class *Caltech()* was created in order to build the dataset. The implementation of this class is in the *caltech_dataset.py* file.

2.1.1 __init__ method

In the constructor of this class, the *split* attribute specifies if we are preparing the train or the test dataset; in fact, the split is not random and the path of the images to load in the respective dataset is stored in a particular file, called *train.txt* or *test.txt*.
The attributes that are saved in the constructor are:

- *self.split*
  This defines the split we are going to use : it can be train or test
- *self.root*
  Root directory of the image dataset
- *self.transform*
  This attribute is for applying preprocessing to the image

- *self.categories*
  A sorted list with all the names of the image categories

- *self.index*
  A list containing integers: each integer is the 4-digits number that corresponds to the last 4 characters of the image name. It is the number of the image.

- *self.y*
  A list containing integers which values are in range 0, 100: different numbers correspond to different categories.

When loading the dataset, the constructor reads the content of the file *train.txt* or *test.txt*, according to the *split* attribute.
Then each line of the file is stored in a list: each element of the list has the format *<category>/image_<number>*. The list is then sorted by alphabetical order and images that are in BACKGROUND_Google category are subsequently filtered out, because they are not meant to be part of the dataset.
<number> field is stored in *self.index* as an integer.
<category> field is stored in *self.y* as an integer between 0 and 100 (mapping from category to number).
An important thing to notice is that len(self.index) = len(self.y).

2.1.2 _getitem_ and _len_ methods

In the implementation of the _getitem_ method, taking as parameter the index of the image, it retrieves the image and his label. The function *pil_loader()* provided in the original template was used in order to access to the image by index and retrieve a *PIL Image*. The image is then preprocessed, according to the *split.transform* attribute.
In the implementation of the _len_ method, it retrieves the length of the list *self.index*: its length corresponds to the total number of images of the specific dataset.

3 AlexNet: Training from scratch

Once the test and the train sets have been created, a validation set is required in order to perform the training procedure. For training, we need to create a single validation set for both hyperparameter tuning and model selection.
Then, the AlexNet model will be used to perform our task.

3.1 Creating the validation set

Next thing we need to do is to split our training set into *training* and *validation* set, taking into account the fact that we need to maintain balance among all classes: it means that we aim to have half samples of each class in train and the other half in val.
The split was performed in a simple way: first, a list containing all the indexes of the original train dataset (= *numbers going from 0 to len(train_dataset)-1*) was created, then indexes divisible by 2 were assigned to val_dataset and the others to train_dataset. The proportion between categories is maintained because the dataset is sorted by alphabetical order.
I thought it was legitimate to do the split in a deterministic way (avoiding using random functions) since the original split between the train and the test set was also deterministic.

3.2 Implement model selection with validation

The AlexNet model was created, using a custom function:

initialize_model(model_name, num_classes, feature_extract, use_pretrained=True)

It instantiates the *model_name* network, modifying the last fully connected layer with *num_classes* nodes, it is also able to load the pretrained model from ImageNet with the *use_pretrained* parameter and it can also freeze some layers of the network using the *feature_extract* parameter.
Since we want to perform a training from scratch, this function was called using:

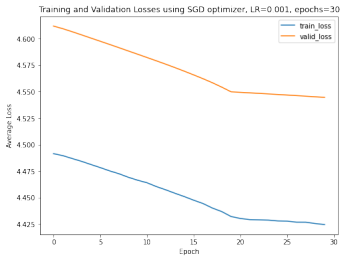
net = initialize_model("alexnet", NUM_CLASSES, False, use_pretrained=False)

The Cross Entropy loss function was then instantiated , then the optimizer was created along with the scheduler.
For training, at each epoch the network is trained on the training set and evaluated on the validation set. The *average train loss* and the *average validation loss* are saved for each epoch, they are finally plotted at the end of the training procedure. The model selection is implemented saving the network with the highest accuracy score on the validation set.

3.3 Implement hyperparameter tuning

For hyperparameter tuning different combinations were tried out:

- The first experiment was performed using the default parameters: SGD with momentum for 30 epochs with an initial learning rate (LR) of 0.001 and a decaying policy (STEP_SIZE) after 20 epochs. The obtained results are:
  Validation Accuracy: 0.1635
  Test Accuracy: 0.1631



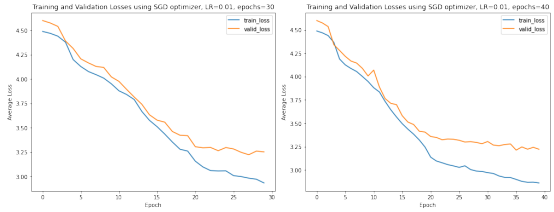
  The plot of train_loss, valid_loss shows that as the training loss decreases, the validation loss continues to decrease. This is good, though the accuracy is still too low. Also, the divergence in losses shows that the model is clearly overfitting.

Next experiments have been performed changing *LR*, *optimizer* and *NUM_EPOCHS* paramaters.

Experiments performed using as optimizer **SGD with momentum** :

- A learning rate LR = 0.01 and NUM_EPOCHS = 30 was set.
  The obtained results are the following:
  Validation Accuracy: 0.3018
  Test Accuracy: 0.3000
  The plot figure below (a) shows that as the NUM_EPOCHS increases, the valid_loss continues to decrease: a useful thing to do could be changing the number of the epochs from 30 to 40, to see the behavior of the average loss.

- A learning rate LR = 0.01 and NUM_EPOCHS = 40 was set.
  The obtained results are the following:
  Validation Accuracy: 0.3118
  Test Accuracy: 0.3041



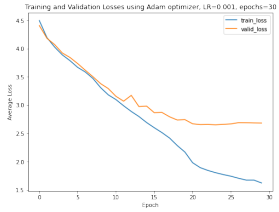
(a) NUM_EPOCHS = 30 (b) NUM_EPOCHS = 40

As the training loss continues to decrease with further training, the validation loss doesn't seem to decrease much; plus, both the models seem to overfit on the training data. Also, the accuracy of model (a) and (b) is quite the same.

Experiments performed using as optimizer **Adam** :

- A learning rate LR = 0.01 and NUM_EPOCHS = 30 was set.
  The obtained results are the following:
  Validation Accuracy: 0.0923
  Test Accuracy: 0.0919
  The accuracy is low, compared to the previous ones. Maybe a lower learning rate could help reach better results: next experiment is performed with LR = 0.001

- A learning rate LR = 0.001 and NUM_EPOCHS = 30 was set.
  The obtained results are the following:
  Validation Accuracy: 0.4336
  Test Accuracy: 0.4244
  Changing the LR parameter caused an increase in accuracy, compared to the previous configuration.



The plot shows that in the first few epochs the validation loss and the train loss decrease one along with the other. But, after a certain epoch, the validation loss reaches a minimum and plateaus, leading the model to overfit on the training data. At a certain epoch, there is no return to further training. Our model will only start to memorize the training data and will not be able to generalize to testing data. A solution for this

problem could be the *early stopping*: in this project it is implemented with the logic of the model selection, in fact every epoch in which the validation accuracy decreases, we save the parameters so we can later retrieve those with the best validation performance.

The best hyperparameter configuration for training from scratch is:
Optimizer: *Adam* LR: *0.001* NUM_EPOCHS: *30*
Validation Accuracy: 0.4336 Test Accuracy: **0.4244**

4 AlexNet: Transfer Learning

The maximum accuracy obtained on the test set performing training from scratch was quite low. To improve accuracy, the best solution is performing *Transfer Learning*.

4.1 Training the whole initialized network

The network was instantiated using the function:

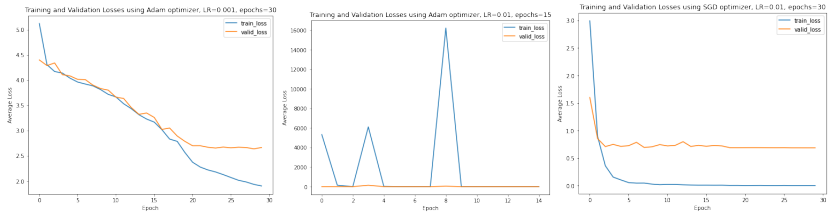
net = initialize_model("alexnet" , NUM_CLASSES, feature_extract=False, use_pretrained = True)

It instantiates the alexnet model pretrained on ImageNet, by setting the *use_pretrained* parameter to *True*.
Also, the Normalize function of data preprocessing has to be changed using ImageNet's mean and standard deviation:

transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

Now the model is ready for training. Different experiments have been done:

- Optimizer=Adam LR=0.001 NUM_EPOCHS=30
  Validation Accuracy: 0.4363
  Test Accuracy: 0.4317
- Optimizer=Adam LR=0.01 NUM_EPOCHS=15
  I expect this configuration to be worse than the previous one: we saw in the last experiments that Adam optimizer doesn't perform well with LR=0.01 in our scenario.
  Validation Accuracy: 0.0923
  Test Accuracy: 0.0916
- Optimizer=SGD LR=0.01 NUM_EPOCHS=30
  Validation Accuracy: 0.8530
  Test Accuracy: 0.8447
  This is by now the best performing model. Next steps will be performed with this hyperparameter configuration.



(a) Adam with LR=0.001 (b) Adam with LR=0.01 (c) SGD with LR=0.01

In Figure (a) the training loss decreases continually with epochs. At a certain point however (around epoch 20), the validation loss stops decreasing. With the divergence in losses, there is likely not much more to gain from further training. However, results obtained in this configuration are very similar to the ones obtained with the same configuration performing training from scratch.
Figure (b) shows what is like using Adam optimizer with a LR=0.001. This model is not good for our task, and it is evident because the training loss reaches very high peaks.
Figure (c) shows the train/validation loss obtained using a SGD optimizer: this is the best model, even if it could be improved (maybe reducing the number of epochs) because it's clearly overfitting. In fact the train loss reaches very low values (close to 0) and the validation loss does quite the same. Both the losses decrease very fast: they both reach their "min" value around epoch 10. Further training won't help to gain better results.

4.2 Training only some layers of the network

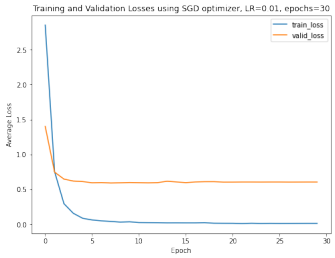
The alexnet model was created using the function:

input_size = initialize_model("alexnet", NUM_CLASSES, feature_extract="conv", use_pretrained=True)

It instantiates the alexnet model pretrained on ImageNet, by setting the *use_pretrained* parameter to *True*.
It is also able to "freeze" some layers of the network due to the parameter *feature_extract* : if it is set to *"conv"* it freezes the convolutional layers, if it is set to *"fc"* it freezes the fully connected layers. The function that performs the "freezing" operates on the *requires_grad* attribute of the parameters of the network, setting it to False (his default value is True) in the parameters related to the "freezed" layers .

4.2.1 Training only the fully connected layers

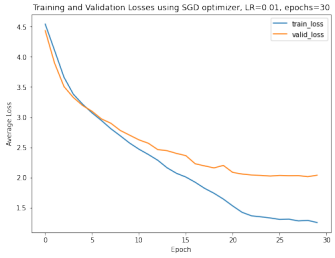
The experiment was done using the best configuration of the previous step:
Optimizer=SGD LR=0.01 NUM_EPOCHS=30
Validation Accuracy: 0.8506
Test Accuracy: 0.8472



This results are very similar to the ones obtained in the previous step, using the same hyperparameter configuration. So there is not so much difference between *re-training* the whole pretrained network and training only the (last) FC layers for our dataset. This behavior could be justified by the fact that the Caltech-101 and the ImageNet dataset are very similar so the first layers of the network learn quite similar features.

4.2.2 Training only the convolutional layers

The experiment was done using the best configuration of the previous step:
Optimizer=SGD LR=0.01 NUM_EPOCHS=30
Validation Accuracy: 0.5708
Test Accuracy: 0.5568



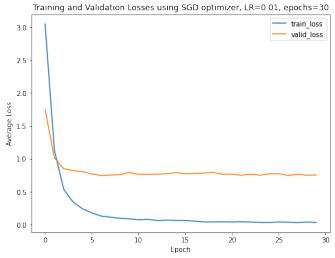
It is clear that this model doesn't perform well, with respect to the previous one. This is because when performing pre-training generally the early convolutional layers of a CNN extract features that are relevant for many image recognition tasks. The later, fully-connected layers, *specialize* to the specific dataset by learning higher-level features. Therefore it is a bad idea to "freeze" the FC layers, leaving convolutional ones "unfreezed".

5 Data Augmentation

Data augmentation was used in order to increase the size of the original dataset. It was performed creating new sets of transformations that extend the original ones.
Augmentation is generally done during training and these random transformations are applied on the training set. This means that during each epoch a different random transformation is applied to each training image.
Different experiments were carried out. Notice that every proposed set of transformations is an extension of the original one, with the *CentralCrop(224)* in order to fit the AlexNet's standards.

- With the best hyperparameter set found on previous steps, a training was performed, freezing the conv layers of the network.
  The image transformation set was extended with:
  RandomRotation(degrees=15), *ColorJitter()*, *RandomHorizontalFlip()*.

  The obtained results are:
  Validation Accuracy: 0.8115
  Test Accuracy: 0.8441

  The results obtained at this point do not differ very much from the ones obtained without all these transformations.

- Then a new set of transformations was tried. This set extends the original one with:
  Grayscale(num_output_channels=3), *RandomHorizontalFlip()*.

  The obtained results are:
  Validation Accuracy: 0.8191
  Test Accuracy: 0.8247
  The results are very similar with respect to the previous point, the plot was also quite the same.

6 Beyond Alexnet

As suggested, also the *VGG-11* model was tried out. It means that training, validation and testing are performed in the same way as before: the difference is that the alexnet model is replaced with the VGG model.

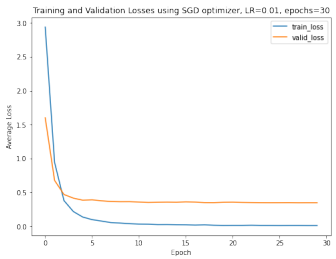
6.1 VGG-11 with batch normalization

The network was instantiated using the function:

net = initialize_model("vgg" , NUM_CLASSES , feature_extract = 'conv' , use_pretrained = True)

It instantiates the *VGG-11 with batch normalization* model pretrained on ImageNet, by setting the *use_pretrained* parameter to *True*.
It also "freezes" the convolutional layers of the network: *vgg-11bn* is a larger model, compared to alexnet, so it consumes more GPU memory and time. Performing the weight update only on the parameters of the last layers could be a reasonable choice for limiting these problems.
The input image size of this net (224x224) was the same as before: there was no need to change it.
Also, the BATCH_SIZE was reduced to 128, due to the higher consume of the CoLab GPU memory.

Then the training was performed, using:
Optimizer=SGD LR=0.01 NUM_EPOCHS=30
Even if the training phase took longer, it indeed produced better results:
Validation Accuracy: 0.9083
Test Accuracy: **0.9014**



The plot shows the trend of classification and validation losses. We can see that, from a certain epoch, the validation loss along with the training loss, do not seem to decrease, in fact they reach a minimum and then plateau: in this case the model is overfitting.
Even if the results obtained in this configuration were the best performing one, they could be improved performing an hyperparameter tuning.