Homework #2: Deep Learning

Simone Dutto

Description of the assignment:

The purpose of this work is to learn how to train correctly a deep neural network, putting emphasis on the tuning of hyperparameters.

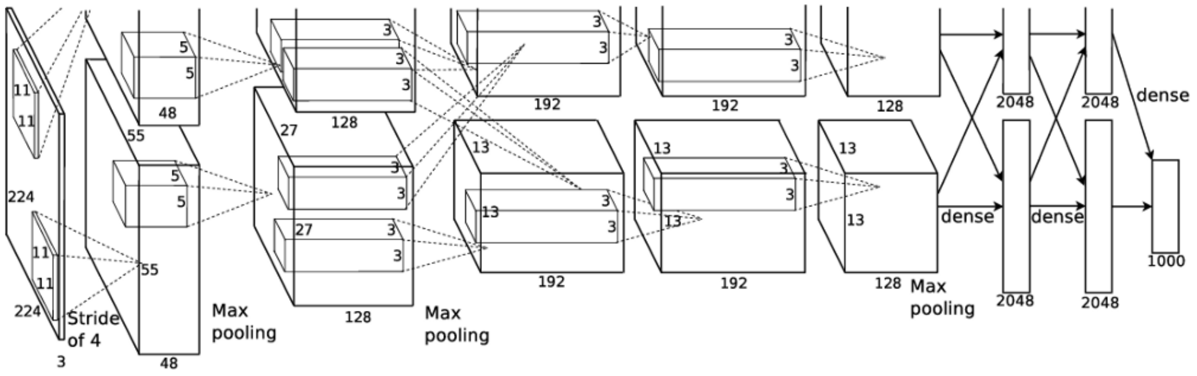
Dataset:

The dataset is the CalTech 101, it contains 9144 entries divided in 102 folders which correspond to 102 different classes.
For our assignment we had to ignore the BACKGROUND class. Easily done by prune the folder starting with BACKGROUND.
For validation and testing, I had to divide the dataset in train, validation and test sets.
The train/test split was already given by the repository, to further divide train into train/validation with a 1/3 – 2/3 proportion I implemented an ad hoc function with complexity O(N) `divideClasses()`.
It creates 3 lists of "path-label" to access images in the dataset and associate the correct label. Then, the list is passed as an argument to the `Caltech` constructor.
It's important to use the `Caltech` class because it implements some methods which are necessary for the `DataLoader`, for example `__len__()` and `__getitem__()`.
Another transformation we apply to the dataset is the resize and the central crop to make images suitable for the AlexNet input size.
Finally, after transforming the image into a tensor we normalize the dataset to better fit the model and avoid overfitting.

Architecture:

Model:

The AlexNet is the architecture chosen for this work.
AlexNet marked a breakthrough in deep learning where a CNN was used to reduce the error rate in ILSVRC 2012 substantially and achieve the first place of the ILSVRC competition.
AlexNet contains five convolutional and three fully-connected layers. The output of the last fully-connected layer is sent to a 1000-way softmax layer which correspondes to 1000 class labels in the ImageNet dataset.



The model was deep compared to others, therefore computational expensive, but it was made feasible using GPUs.
To run the code and my experiments I used Google Colab, which has 12GB VRAM GPU and it takes approximately 10 minutes to train the AlexNet on CalTech101.

The last fully connected layer is modified to output 102 classes of our dataset and not the 1000 classes, which is the ImageNet standard.

Loss function:
Loss functions are a method of evaluating how well specific algorithm models the given data. If predictions deviate too much from actual results, loss function would cough up a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction.
`CrossEntropyLoss()` is the loss for my work, it is computed as:

$$loss(x,class) = -x[class]+log(j\sum exp(x[j]))$$

And it is averaged after each minibatch and used to update weights.
Minibatch size is important: too high and the learning process converges accurately but slowly, too small and the learning process converges quickly but with a lot of noise in the training process.
Minibatch in my case is 256.

Optimizer:
The optimizer helps to minimize the Loss Function.
I choose:
```
SGD(parameters_to_optimize, lr=LR, momentum=MOMENTUM, weight_decay=WEIGHT_DECAY)
```
Learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration. It is a trade-off between the rate of convergence and the risk of overshooting updates.
Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.
Weight decay is used to prevent updates to be too small and overfit the train set.
In addition, we define a scheduler:
```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=STEP_SIZE, gamma=GAMMA)
```
GAMMA is a multiplicative factor for learning rate step-down after each STEP_SIZE number of epochs. It is helpful not to overshoot updates after the model has been trained for a while.

Regularization:
Train time is implemented a regularization technique: Dropout.
During training, some number of layer outputs are randomly ignored. This has the effect of making the layer be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "*view*" of the configured layer. This helps a lot in preventing overfitting and improving generalization.

Training:
For the training phase I tried different combination of learning rate and Step Size.
The procedure is to train the model on train set, test it on validation set, save the best model and use it on test set.

Model	LR	STEP SIZE	EPOCHS	Accuracy on Validation Set	Accuracy on Test Set	Number of Epochs

First model	0.001	20	30	0.161	0.092	16
Second model	0.01	20	30	0.432	0.425	29
Third model	0.05	25	30	0.537	0.550	29
Fourth model	0.03	20	30	0.528	0.532	30
Fifth model	0.03	24	30	0.551	0.529	30

We can notice the learning rate was too small in the first model. Even if the Loss was high, the weights update was not significative, and the model doesn't seem to train properly.
With the second model the learning rate plays a huge role in increasing the performance, and the loss keeps decreasing so I decided to higher the step size in order to have more epochs with a high learning rate.
A higher LR at start is good because it helps to make the first steps towards optima, in fact the best performing model is the third one.

Transfer learning:

Transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task. With PyTorch is possible to use `alexnet(pretrained=True).`
In this case, the net is already trained on ImageNet, so it is already able to extract general features from images, therefore there is a huge improvement in the training phase.
This technique is very powerful when the dataset is little, like CalTech101, and it is important to finetune the model without loosing a lot of time training the net to extract general features (lines, edges, etc.).
In fact, already at the first epoch the accuracy was 0.72.

Model	LR	STEP SIZE	EPOCHS	Accuracy on Validation Set	Accuracy on Test Set	Number of Epochs
First model	0.03	24	30	0.849	0.834	30
Second model	0.03	27	30	0.851	0.841	30
Third model	0.01	10	20	0.878	0.864	16
Fourth model	0.01	10	30	0.879	0.871	29
Fifth model	0.01	15	20	0.877	0.866	20

The model performs good from the start and the Loss in the final steps was very low (0.0005). so I decided to cut the training time by lowering the number of epochs to 20, because the training time was higher, but the performance remained the same.
The third model is better in less epochs, because the LR decreases after 10 epochs and it is possible to make more precise weights updates and finetune the model.
We can notice that the Forth model is a little bit better, but it took almost double the training time.
The Fifth is as good as the Thirds but it took a little bit more to tune.

Freeze layers:

In some case to make the model converge faster we can freeze some layers, in the case of a pretrained network freezing the first layer is a good choice because the extraction of the general features is already good and doesn't need more train.
Freezing the fully connected layers, indeed, is not a smart choice because those are the most specialized layers. In our case they are trained on ImageNet and they must "adapt" to CalTech classification.
I took the best performing model from the table above and trained freezing layers.

With Convolutional Layer freeze it converged a lot fast, it reached 0.86 in the 11th epochs of training. With FC Layers freeze the model was not accurate on CalTech101 and converged really slow.

Model	LR	STEP SIZE	EPOCHS	Accuracy on Validation Set	Accuracy on Test Set	Number of Epochs
Freeze Conv	0.01	10	30	0.860	0.856	11
Freeze FC	0.01	10	30	0.491	0.489	29

Data Augmentation:

Having a large dataset is crucial for the performance of the deep learning model. However, even if our dataset is small we can improve the performance of the model by augmenting the data we already have. Data augmentation helps the generalization of the model and prevent overfitting.
I have used: HorizontalFlip, ColorJitter and RandomRotation.
It is important to notice those techniques doesn't higher the size of a dataset, but at each epoch there is a certain probability (I set 50%) each entry can be modified, so the dataset remains the same size but the number of different entries the net trains on increases.

Model	LR	STEP SIZE	EPOCHS	Accuracy on Validation Set	Accuracy on Test Set	Number of Epochs
Horizontal Flip	0.01	10	30	0.873	0.861	14
ColorJitter	0.01	10	30	0.872	0.872	24
Rotation 20°	0.01	10	30	0.861	0.853	22
Horizontal Flip | ColorJitter | Rotation 20°	0.01	15	30	0.857	0.856	28

ColorJitter helps a little accuracy in test time, improvement are not huge and this could be caused by the fact that train set and test set are very similar.

TenCrop:

I also tried TenCrop at test time, this technique consists in: apply 10 different crops(four corners and the central crop plus the flipped version of these) to the same image, evaluate the 10 crops with the net, average the results and get the solution.
It is computational expensive, but it increases performances test time.

Model	LR	BATCH SIZE TEST	STEP SIZE	EPOCHS	Accuracy on Validation Set	Accuracy on Test Set	Number of Epochs
TenCrop	0.01	32	10	30	0.868	0.872	29

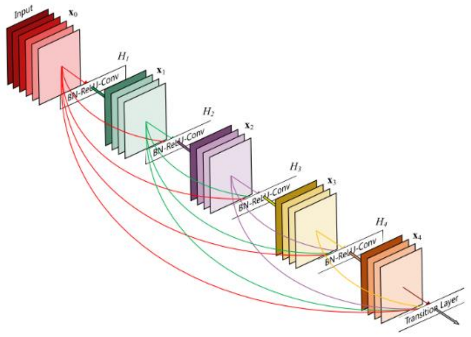
Beyond AlexNet:

I tried also another architecture, this one is deeper, and it took longer to train but it was way more accurate.

ResNet is different from common deep CNN.

The failure of the deep CNN could be blamed on the optimization function, initialization of the network, or the famous vanishing/exploding gradient problem. For this reason, the author of ResNet introduced the Skip Connections between layers, they add the outputs from previous layers to the outputs of stacked layers.

This results in the ability to train much deeper networks than what was previously possible.



Model	LR	BATCH SIZE	STEP SIZE	EPOCHS	Accuracy on Validation Set	Accuracy on Test Set	Number of Epochs
ResNet50	0.01	32	10	20	0.954	0.947	12

The train time was almost the triple of AlexNet but the performance is way better in much less epochs.
This demonstrates the power of Skip Connection modules; these modules can make a highway for gradient to flow in each layer.
This permits a good training procedure where each layer can gain from the train phase. In other deep architectures the first layers were almost not updated in train phase.