VR Assignment 3

REPO (https://github.com/Gapes21/Assignment-3.git) - It is recommended to view this report on Github :)

Subhajeet Lahiri - IMT2021022 | Sai Madhavan G - IMT2021101 | Aditya Garg - IMT2021545

Part A

Problem Statement

The goal of this part of the assignment was to play around with CNNs. Particularly, we were to experiment with different optimization techniques and activation functions. We were to compare the different approaches using training time and classification performance. Based on our experiments, we were to recommend the best architecture.

Dataset

We performed these experiments on the CIFAR-10 (https://www.cs.toronto.edu/%7Ekriz/cifar.html) image classification dataset. It consisted of 50000 train images and 10000 test images of size 32x32x3. Each image had exactly one class label among 10 possible values.

Base model

We opted to work with a CNN with 3 convolutional layers followed by 2 fully connected layers. This model consisted of around 820k parameters. The model was implemented using the pytorch library. All the code used in this section can be accessed in the `part_a` directory of our github repo (https://github.com/Gapes21/Assignment-3).

Evaluation Metrics

We used the following evaluation metrics to compare the results of our experiments:

- **Accuracy:** The percentage of the test set predictions that matched the ground truth.
- **Macro f1:** The average f1 score of all classes.
- **Epochs until convergence:** The number of epochs the model took before it reached the maximum accuracy.
- **Time until convergence:** The time since the beginning of the training till the model reached the maximum accuracy.

Experiments on optimization strategies

Vanilla SGD

We started our experiments by optimizing our model using vanilla stochastic gradient descent (SGD). We dub this experiment and its resulting model as `CNN1`. Our implementation used the `SGD` optimizer of `torch.optim` and we used a constant learning rate of 10$^{-3}$.

Observations

- The model came close to convergence after 500 epochs and 7430 seconds (~2 hours).
- It reached the best test loss at around 200 epochs after which the test loss began rising

- Despite the increasing test loss, the test accuracy and f1 kept increasing and the train accuracy reached 99% at the end of 500 epochs.
- The best accuracy was **63.67%** and the best test f1 was **0.64** at **500** epochs (~**2 hours**)

SGD with momentum

We next tried introducing momentum to our optimization. We used a momentum of **0.9** with initial learning rate at \(10^{-3}\). This experiment and it's accompanying model was called `CNN1.1`.

Observations

- The least test loss was obtained in just 40 epochs (789 seconds).
- Similar to the previous experiment, despite the increasing test loss, the test accuracy and f1 were increasing up until the experiment concluded at 120 epochs.
- The best accuracy was **70.83%** and the best test f1 was **0.71** at **120** epochs (~**40 minutes**)

Adam

We then used the Adam optimizer which uses adaptive estimates of first order and second order moments. The initial learning rate was again \(10^{-3}\). This experiment and it's accompanying model was called `CNN1.2`.

Observations

- The least test loss along with the maximum accuracy was obtained in just 7 epochs (433 seconds).
- Contrary to the earlier experiments, the test accuracy and f1 was fluctuating with a downward trend following this, when the test loss started increasing.
- The best accuracy was **68.17%** and the best test f1 was **0.69** at **7** epochs (~**7 minutes**)

Conclusion

- The vanilla SGD approach was very slow and it wasn't able to reach the best accuracy score either.
- Using momentum made training considerably faster and the model reached a much better score too.
- Both of the above models exhibited an oddity wherein the test accuracy was increasing in spite of signs of over-fitting, something we weren't able to explain.
- Adam was the fastest to converge. However, it's best score was a bit lesser than the approach of SGD with momentum. This is probably due to the *"overshooting"* nature of the optimizer.
- We choose **Adam** to be our optimizer of choice due to it's fast convergence. We can counteract this overshooting nature in larger models and datasets using weight decay.

Experiments using various values of dropout

In the earlier experiments, we observed overfitting to be a persistent problem. To fix this, we use different values of dropout in the fully connected layers. We conducted experiments with 4 different values of dropout:

1. `CNN2.1`: **0.2**
2. `CNN2.2`: **0.4**
3. `CNN2.3`: **0.6**
4. `CNN2.4`: **0.8**

We trained these three models for 25 epochs each. All other factors except the dropout of these models are identical to model `CNN1.2`

Observations

- The best accuracy of `CNN2.1` was **71.44%** and the best test f1 was **0.71** at **9** epochs (**83 seconds**)
- The best accuracy of `CNN2.2` was **72.54%** and the best test f1 was **0.72** at **17** epochs (**150 seconds**)
- The best accuracy of `CNN2.3` was **72.00%** and the best test f1 was **0.72** at **14** epochs (**127 seconds**)
- The best accuracy of `CNN2.4` was **69.59%** and the best test f1 was **0.69** at **22** epochs (**200 seconds**)

Conclusion

- Using dropout increased performance considerably at the cost of a slight increase in convergence time
- Using a dropout value of **0.4** seemed to be ideal as increasing it too much is resulting in decrease in performance.

Experiment with addition of Batch Normalization

In this experiment, we introduced 2-dimensional batch normalization after each convolutional layer. We called this model `CNN3` All other factors were identical to the model `CNN2.2`.

Observations

- The best accuracy of `CNN3` was **75.89%** and the best test f1 was **0.76** at **25** epochs (**227 seconds**)

Conclusion

- Using batch normalization further increased the score. We speculate this is because it helps in solving the vanishing/exploding gradients problem, thereby making more neurons "active".

Experiments with various activation functions

Finally, we experimented with different activation functions that were used after each convolutional and fully connected layers. All models upto `CNN3` used the ReLU activation function. All other factors except the activation function were identical to `CNN3`. We conducted experiments with the following activation functions:

1. `CNN4.1`: Sigmoid
2. `CNN4.2`: tanh
3. `CNN4.3`: leaky ReLU (negative_slope = 0.01)
4. `CNN4.4`: GELU

Observations

- The best accuracy of `CNN4.1` was **72.67%** and the best test f1 was **0.73** at **25** epochs (**230 seconds**)
- The best accuracy of `CNN4.2` was **72.84%** and the best test f1 was **0.73** at **23** epochs (**211 seconds**)
- The best accuracy of `CNN4.3` was **76.77%** and the best test f1 was **0.77** at **11** epochs (**105 seconds**)
- The best accuracy of `CNN4.4` was **77.21%** and the best test f1 was **0.77** at **22** epochs (**203 seconds**)

Conclusion

- Based on the above observations, we concluded that the **GELU** activation function was the best.

Conclusion

Model	Optimization Strategy	Dropout	Batch Normalization	Activation Function	Test Accuracy (%)	Test f1	Epochs till convergence	Time till convergence (s)
CNN1	SGD	0	No	ReLU	63.67	0.64	500	7430
CNN1.1	SGD with momentum	0	No	ReLU	70.83	0.71	120	2464
CNN1.2	Adam	0	No	ReLU	68.17	0.68	*7*	433
CNN2.1	Adam	0.2	No	ReLU	71.44	0.71	9	*83*
CNN2.2	Adam	0.4	No	ReLU	72.54	0.72	17	150
CNN2.3	Adam	0.6	No	ReLU	72.00	0.72	14	127
CNN2.4	Adam	0.8	No	ReLU	69.59	0.69	22	200
CNN3	Adam	0.4	Yes	ReLU	75.89	0.76	25	227
CNN4.1	Adam	0.4	Yes	Sigmoid	72.67	0.73	25	230
CNN4.2	Adam	0.4	Yes	tanh	72.84	0.73	23	211
CNN4.3	Adam	0.4	Yes	Leaky ReLU	76.77	*0.77*	11	105
CNN4.4	Adam	0.4	Yes	GELU	*77.21*	*0.77*	22	203

Based on the results of our experiments, we recommend the configuration of the model `CNN4.4`, due to it's performance and efficiency.

Part B

Problem Statement

This part delves into the effectiveness of CNNs for object recognition.

1. **Dataset Selection:** We needed to choose a suitable object detection dataset beyond the widely used CIFAR-10/MNIST.
2. **CNN Feature Extraction** : Utilizing a pre-trained CNN architecture as a feature extractor. The report will explain the chosen architecture and its key components.
3. **Classification on Top**: Employing a ML model on top of the extracted CNN features for classification. We also had to report accuracies for the Bikes vs Horses dataset.

In order to do this, we experimented with the bike vs horse dataset, SVHN dataset and the FashionMNIST dataset. Despite this, most of the observations and the accuracies would be made on the dataset FashionMNIST.

Procedure

Initially, we set up the building blocks of the network using sequential modules.

- Three convolutional blocks (block_1 to block_3) perform feature extraction. Each block uses a 2D convolution with 3x3 kernel and "same" padding to preserve image dimensions.
- Batch normalization (BatchNorm2d) helps with training stability.
- ReLU activation introduces non-linearity.

- Max pooling downsamples the feature maps, reducing spatial dimensions but capturing essential features.
- In the next layer, (Flatten Layer) transforms the multi-dimenaional feature maps into a 1D vector.
- A fully connected layer (block_4) with ReLU activation further processes the features.
- The final linear layer (last_layer) maps the features to 10 output units, representing 10 possible clothing categories.

  Moving ahead, the "forward" method defines the data flow of the network.

For further training, we use the cross entropy loss with Adam optimizer (learning rate was set as 0.00032. We trained the model with 2 epoch values. One of the value was set as 128 and the other was set as 32. After this step, we form a new dataset by taking the training dataset, and passing it to the network.

Classifiers

This dataset will serve as the input for the following classifiers:

1. Logistic Regression
2. Random Forest
3. Bernoulli NB
4. Multinomial NB
5. Gaussion NB
6. KNN
7. Linear SVC
8. Extra Trees Classifiers
9. Linear SVC with L1 Regularization

Following are the accuracies achieved using these models:

Note that unless mentioned, we are using ReLU activations.

FashionMNIST

128 epochs

i. Logistic Regression: 0.9137 (max_iter=1024)
ii. Random Forest - 0.9147
iii. Gaussian NB - 0.9119
iv. Multinomial NB - 0.9106
v. Linear SVC - 0.9125 (max_iter=5000)
vi. Gradient Boosting - 0.8287 (n_estimators, learning rate=1.0, max_depth=3)
vii. KNeighbours - 0.9148 (n_neighbours=50)

32 epochs

i. Logistic Regression - 0.9123

ii. Random Forest - 0.9124
iii. Gaussian NB - 0.9087
iv. Multinomial NB - 0.9011
v. Linear SVC - 0.9143
vi. Gradient Boosting - 0.8998
vii. Bernoulli NB - 0.82

Bike-Horse

The below values are for the Bike-Horse dataset.

i. Logistic Regression - 100
ii. Random Forest - 100
iii. Gaussian NB - 100
iv. Multinomial NB - 100
v. Gradient Boosting - 100
vi. LinearSVC - 100
vii. K-Nearest Neighbours - 100
viii. Extra Trees Classifiers - 100
ix. Bernoulli NB - 85.7142
x. LinearSVC with L1 regularization - 100

SVHN (Street View House Numbers) Dataset

SVHN is obtained from house numbers in Google Street View images - similar to MNIST but significantly more real-world.

ReLU activation

i. Logistic Regression - 90.16
ii. Random Forest - 90.24
iii. Gaussian NB - 87.60
iv. Multinomial NB - 89.67
v. LinearSVC - 89.80
vi. K-Nearest Neighbours - 90.51
vii. Extra Trees Classifiers - 90.53
viii. Bernoulli NB - 80.50
ix. LinearSVC with L1 regularization - 89.74

Sigmoid activation

i. Logistic Regression - 90.18
ii. Random Forest - 90.31
iii. Gaussian NB - 89.42
iv. Multinomial NB - 89.48
v. LinearSVC - 90.16
vi. K-Nearest Neighbours - 90.58
vii. Extra Trees Classifiers - 90.29
viii. Bernoulli NB - 19.58
ix. LinearSVC with L1 regularization - 90.15

Part C

Improvements made by YOLOv2 over YOLOv1 :

1. Architecture - Darknet 19

Darknet 19 is a smaller and a more efficient CNN. It has 19 Conv. layers compared to YOLOv1's 24. It also contains 5 Max-Pooling layers. It is a smaller model - 5.5 billion parameters compared to v1's 8.5 billion. A significant change in v2 was the **omission of the FC layers** which led to the drop in model size. This step also resulted in better spatial information preservation - crucial for accurate bounding box prediction.

2. Anchor Boxes

YOLOv1 had poor localization performance due to its system of dividing the image into a fixed grid. It was specially difficult for v1 to assign accurate bboxes to objects appearing close together within a cell.

v2 addressed this using **Anchor boxes**. A set of anchor boxes with different sizes and aspect ratios are predefined. These boxes act as references for predicting the actual bounding boxes of objects. For each grid cell, v2 predicts offsets relative to the most appropriate bounding box to get the final output - along with the confidence score.

YOLOv2 also predicts multiple bounding boxes per grid cell, each associated with a different anchor box, allowing it to handle the multiple object situation.

3. Batch Norm

YOLOv2 incorporates batch norm as a technique to improve training efficiency and converge faster. It addresses the issue of internal covariate shift - where the distribution of activations within a layer changes during training, which hinders the learning process of the subsequent layers due to inconsistent input distributions. Batch norm layers - placed after conv layers and before activation functions - keep track of the mean and std. devn. of activations for each mini-batch and then use them to normalize activations. This not only leads to faster training convergence but results in improved gradient flow.

4. Increase sizes for input and feature maps

v2 raises the input resolution to 416x416 from v1's 224x224 - allowing it to capture finer details and accurately detect and differentiate between smaller objects. It also produces larger feature maps which retain more information about the image and allowing better localization.

5. Multi Scale Training

YOLOv1 struggled with objects of varying sizes in images. To address the same, YOLOv2 used multi-scale training - during training the model is not given images of fixed size but the images are randomly resized and the rest is padded in black. Aspect ratios are preserved during such an operation. This allows the model to learn to detect objects in various scales and aspect ratios and achieve a more robust and generalizable object detectoin performance.

YOLO9000 also incorporates the concept of hierarchical classfication - but this is outside our current scope.

Part D

Problem Statement

In this part of the assignment, we were required to create a **car counter** and use it on the junction videos recorded during our previous outing to brigade road. The system comprises of two crucial components :

1. **Detector** : The detector spits out the bounding boxes around the cars that we need to count. We were instructed to experiment with **FasterRCNN** and **YOLOv2**.
2. **Tracker** : The tracking algorithm accepts the bounding boxes detected earlier and matches tracked objects across frames so that we can *track* them as they move. We were instructed to experiment with the **SORT** and **DeepSORT** algorithms.

We were also asked to document the difference between SORT and DeepSORT and create ground-truths for evaluation.

System Overview

As mentioned in the problem statement, the car counting system needs two main components - the detector and the tracker. As such we have two base classes `Detector` and `Tracker`. We pass instances of both to the `VehicleTracker` class which contains the complete flow combining the frame-by-frame detection and tracking logic.

1. `DetectorFasterRCNN` and `DetectorYOLO` inherit the `Detector` class and implement the base methods :
   - `__init__` Apart from fetching the model with its pre-trained weights, the init method also contains the list of vehicle classes that we need to detect. This is done inside the detector itself, as the class labels vary model-to-model and dataset-to-dataset. Both our FasterRCNN and YOLO implementations were trained on MS COCO and the labels have been used accordingly.

     It is to be noted that we decided to count not just cars, but all available vehicle classes in MS COCO - cars, bikes, trucks and buses.
   - `getDetectionsSORT` This method returns the detections in the following format :

     \([x_1, y_1, x_2, y_2, score]\)

     i.e. the ltrb coordinates of the bounding box and the confidence score.

     The detections are returned after filtering out the detected objects which do not belong to the vehicle classes and those which have a confidence score below `0.7`.
   - `getDetections` This method returns the detections in the following format :

     \((box, score, label)\)
2. `TrackerSORT` and `TrackerDeepSORT` inherit the `Tracker` class and implement the following base methods :
   - `getTrackedObjects` : Given the detections in a frame, the tracker returns its own set of bounding boxes after assigning an `id` to each tracked object. This id is all that is required to track an object across frames.
   - `id_and_bbox` : This is just a utility method to unpack the tracker's output to get the bounding box and the tracking id.
3. `VehicleTracker` is initialized with instances of the desired `Detector` and `Tracker` objects and then one can use the `getVideo` method to obtain bounding boxes around the tracked vehicles and a vehicleCount that is updated after each frame.
   - Vehicles are counted once they cross a vertical line drawn approximately halfway across the screen. We agreed upon such a strategy so that we could count vehicles which were not just on the screen but those which *crossed* the intersection.

- A set (Data structure without duplicate elements) of crossing ids is maintained. The vehicle count is the size of this set and is displayed on the top right corner of the video.

Implementations used

We used pretrained models and trackers throughout the system as training full scale detectors and trackers was not possible on our systems with the limited and unstructured data in our possession.

1. FasterRCNN

Faster R-CNN — Torchvision 0.16 documentation (https://pytorch.org/vision/0.16/models/faster_rcnn.html)

We used the Faster RCNN implementation with a ResNet-50-FPN backbone available in `torchvision.models`. We used the weights that come with the trained model.

2. YOLOv2

https://pjreddie.com/darknet/yolo/ (https://pjreddie.com/darknet/yolo/)

The detector was used with already available weights. The implementation is contained in the `yolo` and the `yolo_gpu` folders. A YOLO compatible version of our tracker.ipynb file can be found inside the latter.

3. SORT

GitHub - abewley/sort: Simple, online, and realtime tracking of multiple objects in a video sequence. (https://github.com/abewley/sort?tab=readme-ov-file)

We have utilized the SORT implementation written by one of the authors of the paper itself - Alex Bewley.

> *SORT is a barebones implementation of a visual multiple object tracking framework based on rudimentary data association and state estimation techniques. It is designed for online tracking applications where only past and current frames are available and the method produces object identities on the fly. While this minimalistic tracker doesn't handle occlusion or re-entering objects its purpose is to serve as a baseline and testbed for the development of future trackers.*

SORT works in the following steps :

1. **Detection** : An external detector provides the SORT tracker with the bounding boxes and confidence scores for detected objects.
2. **Kalman Filter** : An algorithm used for state estimation in dynamic systems - it predicts the positions of objects in the current frame based on their previous positions and velocities.
3. **Matching** : The tracker associates the predictions with the detections using a matching algorithm.
4. **Tracking** : The tracker keeps track of the objects by assigning a unique ID to each detected object and then continuously updates its state - position and velocity. It also initializes new tracks for unassociated detection

The `sort.py` file in Bewley's implementation contains the `Sort` tracker being used by us.

4. DeepSORT

deep-sort-realtime · PyPI (https://pypi.org/project/deep-sort-realtime/)

We used this readily available (as a pip package!), real-time implementation of the DeepSORT algorithm.

DeepSORT follows a flow similar to the one used by SORT with a few additional steps :

1. **Feature Extraction** : The tracker extracts features from the detected bounding boxes of objects using a pre-trained CNN.

2. **Association** : DeepSORT associates detections across frames based on learned embeddings. It computes embeddings for both the detected objects and existing tracks, typically using a deep neural network trained specifically for this purpose.

   This makes DeepSORT better at handling challenges such as appearance variations, occlusions and temporary disappearances.

3. **Matching** : Hungarian algorithm or similar for matching detections to existing tracks.

4. **Tracking** : On top of the usual track management, DeepSORT incorporates mechanisms to confirm the existence of tracks by considering track quality metrics, such as track consistency and appearance consistency. Tracks that do not meet certain criteria are deleted to avoid cluttering the tracking output with false positives..

SORT v/s DeepSORT

In the previous section, we delved deep into the workings of SORT and DeepSORT. On the basis of the above, we tabulate the following differences :

1. **Feature Representation** : As we saw DeepSORT uses deep learning techniques to extract features from bounding boxes. These features capture more complex details about object appearance and characteristics as compared to the simple bounding box coordinates, confidence scores and velocity used in SORT.

2. **Matching** : Here too, DeepSORT uses learned embeddings to associate detections and existing tracks, while SORT uses algorithmic techniques like Kalman filtering.

3. **Performance** : SORT is computationally more efficient and light-weight due to its reliance on algorithms rather than learning.

Here, we can judge that SORT is more focussed on algorithms and feature engineering while DeepSORT employs feature learning.

Results

The results of the experiments can viewed on OneDrive (%5BIntersectionVideos%5D(https://iiitbac-my.sharepoint.com/:f:/g/personal/subhajeet_lahiri_iiitb_ac_in/Er_kDN_YTHxJm6l9yPzVegQBSDYXb0x0vwFqGAlQo7bbdw?e=Kgapvf)).

We counted the vehicles manually to generate the ground truths. The results are tabulated as follows :

Video	Autos	Cars	2-Wheelers	Trucks	Buses	Vehicle Count (GT)	FasterRCNN + SORT	FasterRCNN + DeepSORT	YOLO + SORT	YOLO + DeepSORT
vid1	10	5	11	3	5	34	31	44	18	29
vid2	13	31	46	2	3	95	95	177	43	82
vid4	15	17	28	3	2	65	82	178	32	75
vid5	0	8	14	0	0	22	18	53	8	3
vid6	17	31	44	2	1	95	102	217	43	70
vid7	7	26	9	0	0	42	43	90	35	39

Here are our notes on the results observed :

1. SORT counts lesser objects than DeepSORT. This is true across detectors and is most likely due to the advanced feature representations and embeddings used in DeepSORT which enables it to bypass occlusion and similar effects and keep track of objects - sometimes even extraneous ones.

On a similar note, FasterRCNN detects more objects than YOLO. This is obvious when we remind ourselves that the former was geared towards accuracy while the latter towards speed.

2. The best performing combo is - **FasterRCNN + SORT**. The temperance of SORT perfectly complements the accurate detections of FasterRCNN - leading not only to more accurate vehicle counts but also more pleasing and clean detections.
3. DeepSORT goes wild when used with FasterRCNN - it grossly overcounts and hallucinates bounding boxes out of nowhere. Some of the mistakes can be attributed to the unsteady camera which made the tracker create new tracks for existing objects while keeping ones along their old trajectories.
4. DeepSORT + YOLO is the second most decent as in this case the rowdiness of DeepSORT is neutralized by the temperance (or ineptitude) of YOLO.
5. SORT + YOLO is another extreme as **two modest cooks make a bland broth**.
6. We did experience sir's in-class prediction come true irl as neither detector could reliably detect an auto.