VISUAL-RECOGNITION

Mini-Project-1 Report

Dhanvi Medha Beechu - IMT2020529

Sriram Munagala - IMT2020030

Pratham Dandale - IMT2020038

3a:

For this problem, as we had to use Pytorch extensively, I went through the videos posted by our TA and understood the necessities required. AlexNet has 5 Conv and 3 FC layers but as posted in the problem, initially we tried experimenting with 2 convolutional and 2 fully connected and then finalized with 2 convolutional and 3 fully connected layers.

The architecture is as follows.

- 2x2 Convolutional layer with stride = 2, and number of filters = 128.
- 2x2 max pool layer with stride = 2
- Batch normalization
- 2x2 Convolutional layer with stride = 2, and number of filters = 256.
- 2x2 max pool layer with stride = 1
- Batch normalization
- Linear Layer (output size = 1000)
- Linear Layer (output size = 120)
- Linear Layer (output size = 10)
- Softmax Classification
-

Convolutional layers have tanh activation while fully connected layers have ReLu activation.

For 40 epochs, this gave 67.54% accuracy.

Training time was around one hour (using GPU on kaggle) for 40 epochs, so for taking graphs and observations later, therefore for later observations we used 15 epochs.

Both training and validation plots have been included in this report

Batch Normalization:

Used batch normalization after every convolutional layer(after max pooling), as in the beginning layers there are many parameters and filters initially, leading to noisy gradients.

Activation functions:

- **Sigmoid**

  g(x) = 1/(1 + e^(-x))

- **Tanh**

  g(x) = 2/(1+ e^(-2x)) - 1

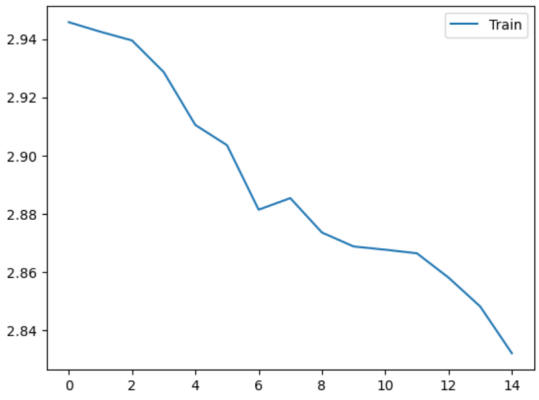
- **ReLu**

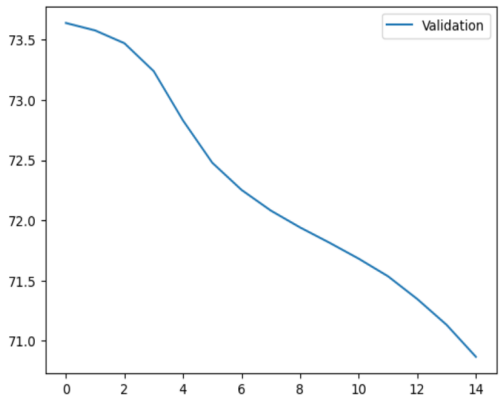
  g(x) = max(0,x)

Without momentum:

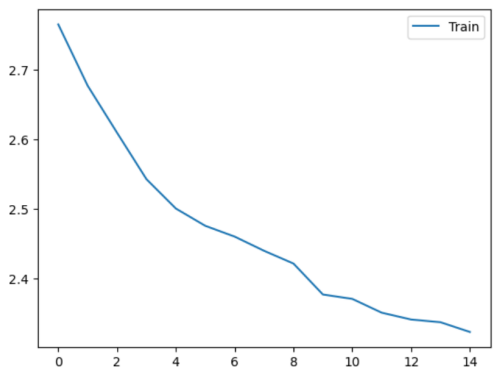
Without momentum, the network did not converge for all 3 activations listed below even for 30 epochs

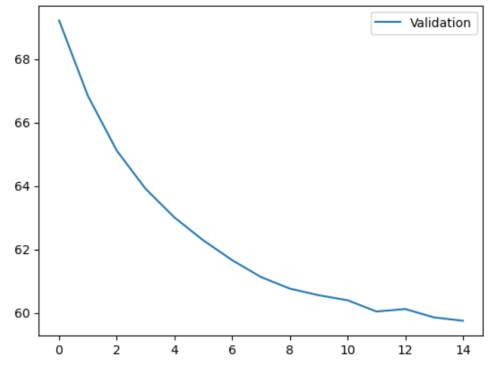
Sigmoid



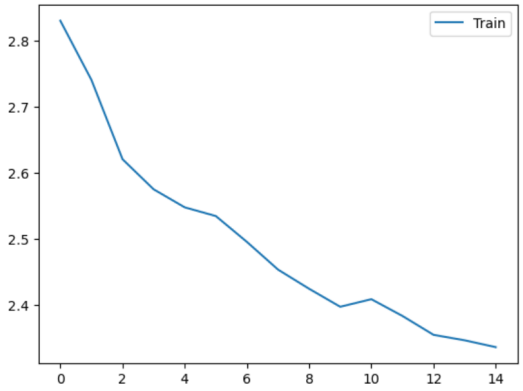


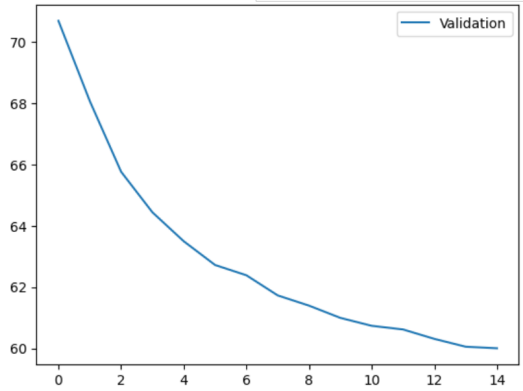
tanh





ReLu





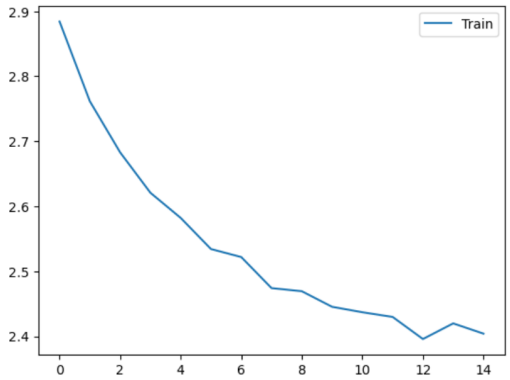
With momentum

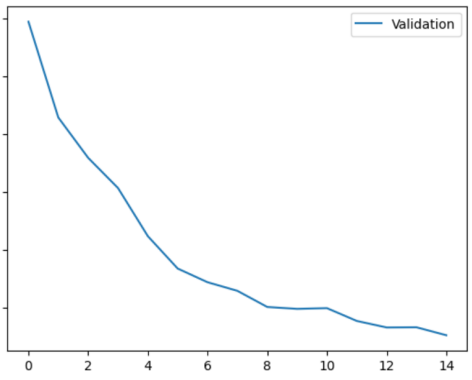
With momentum in the range (0.8 to 1) all (sigmoid,tanh,relu) converged within 40 epochs,

Took more than an hour each to run the training.

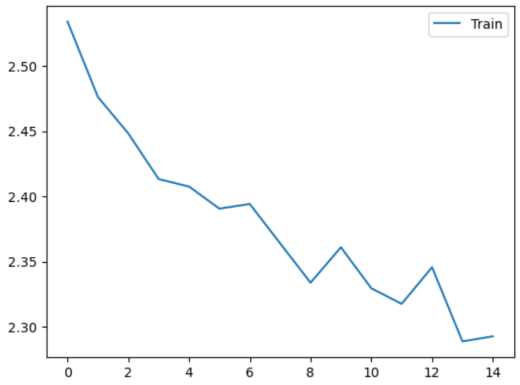
Accuracies of 54.84 for sigmoid, 59.31 for tanh and **67.54 for ReLu (best)** were reported.

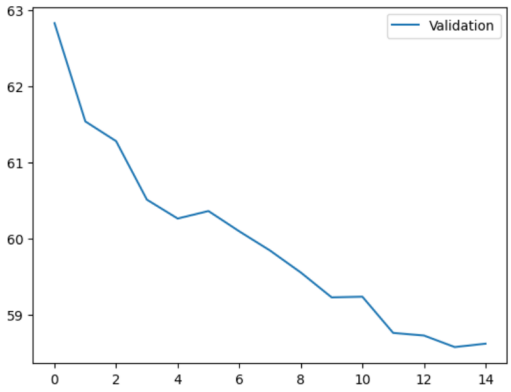
Sigmoid:


Train
2.9
2.8
2.7
2.6
2.5
2.4
0
2
4
6
8
10
12
14

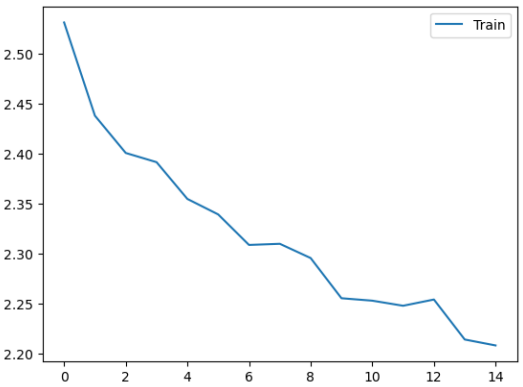

Validation
72
70
68
66
64
62
0
2
4
6
8
10
12
14

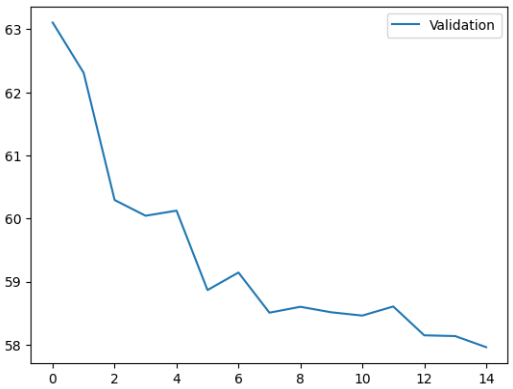
Tanh


Train
2.50
2.45
2.40
2.35
2.30
0
2
4
6
8
10
12
14


Validation
63
62
61
60
59
0
2
4
6
8
10
12
14

ReLu


Train
2.50
2.45
2.40
2.35
2.30
2.25
2.20
0
2
4
6
8
10
12
14

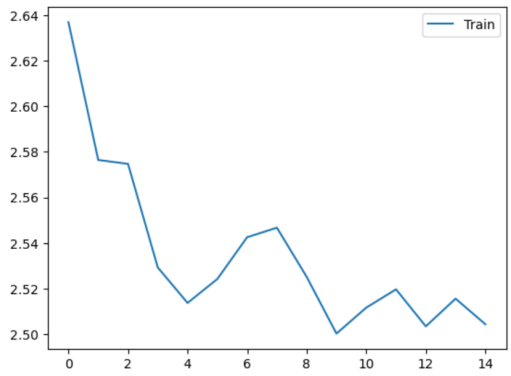

Validation
63
62
61
60
59
58
0
2
4
6
8
10
12
14

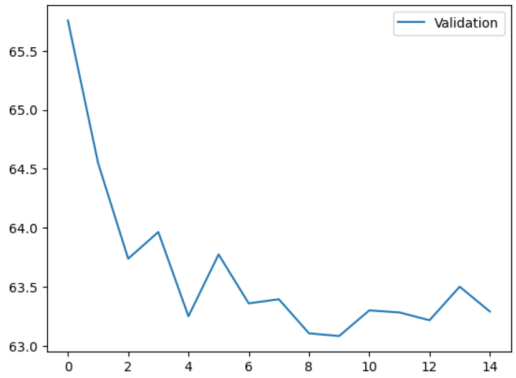
Adam: Adaptive learning

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1) \nabla_\theta J(\theta)$$
$$\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \quad \theta \leftarrow \theta - \eta \boldsymbol{m} \oslash \sqrt{\boldsymbol{s} + \epsilon}$$

The Adam optimizer is an optimization method combining RMSProp and SGD with momentum. The exponential moving average of the last n gradients, described by the beta parameter (tuple of beta1 and beta2) is used in place of the current gradient.

If momentum is like a ball running a slope, Adam behaves like a heavy ball with friction, which helps in preventing flat minima in the error surface.





As seen from the above graph, there was significant noise. Optimizer did not converge even at 40 epochs , therefore SGD with momentum was largely preferred.

3b:

In this question, we are asked to pick an object recognition dataset other than CIFAR/MNIST. We have ended up choosing the intel Image classification dataset from Kaggle. This dataset is basically a collection of images of locations and there are six categories in it.

The entire dataset contains about 14k images spread over all categories which seem like enough data to extract most of the features. There are also 3k images included in the test dataset too. Each image has a size of 150x150. All the images were extracted in a numpy array using standard python libraries.

I have used the library function in PyTorch to generate a pre-trained alexnet model and generated a feature vector for each image and trained the images on a logistic Regression network using the sklearn library.

In order to use the data loader function in torch.utils, I also had to modify the dataset into a particular format which is mentioned in the documentation. I made a custom dataset class and defined some functions in it. The model is also finetuned for my dataset.

For this model, I have added 3 extra layers also. I have added two Linear Layers and one activation layer. I have used the sigmoid function in the activation layer.

The Accuracies I got after running the AlexNet + Logistic Regression model are as follows:

- Intel Image classification dataset: 94 %
- Horse/Bike Dataset: 99%

I seem to have gotten high accuracies for both the datasets because there were a lot of training examples available for each category and there were very less

categories (1 and 6 for Horse/Bike and Intel respectively) compared to other datasets.

I had approximately 3000 images for every category in the Intel Dataset. Standardizing the dataset and resizing the images to 256x256 have also seemed to have improved the accuracy of the model. I have tried testing with some other classification models too but logistic regression gave the highest accuracy of them all.

3c:

5 additional features of YOLO V2:

1. **Batch Normalization**

    - **Adding a batch normalization layer in all of the convolutional layers in YOLO v2 improved the map (mean average precision) by 2%.**
    - **Improved the network training convergence and eliminated the need for other regularization techniques like dropout without the network getting overfitted on the training data.**

2. High Resolution Classifier

- **In YOLO v1, image classification task was performed as a pre-training step on the ImageNet dataset at input resolution**

224x 224, later upscaled to 448x448 for object detection.

Because of this, the network had to simultaneously switch to learning object detection and adjust to the new input resolution. That could have been a problem for the network weights to adapt to this new resolution while learning the detection task.

- **In YOLO v2, the pretraining classification step with 224x224.**

Still, they fine tuning of the classification network is carried out at the upscaled 448x448 resolution for ten epochs on the same ImageNet data.

By doing this, the network got time and adapted its filters to work better on the upscaled resolution since it had already seen that resolution in the fine-tuning classification.

- **This led to a 4% increase in mAP.**

3. Dimension Clusters

- **Like in Faster R-CNN, the sizes and scales of Anchor boxes were pre-defined without getting any prior information.**

- **Standard Euclidean distance based k-means clustering was not good enough as larger boxes generate more error than smaller boxes**
- **YOLOv2 uses k-means clustering with a new metric, which leads to good IOU scores:**

$$d(box, centroid) = 1 - IOU(box, centroid)$$

- **k = 5 is the best value with good tradeoff between model complexity and high recall.**
- **IOU based clustering with 5 anchor boxes (61.0%) has similar results with the one in Faster R-CNN with 9 anchor boxes (60.9%), while IOU based clustering with 9 anchor boxes got 67.2%.**

4. Convolution with Anchor Boxes

- **YOLO v2 removes all fully connected layers and uses anchor boxes to predict bounding boxes.**
- **One pooling layer is removed to increase the resolution of output.**
- **416×416 images are used for training the detection network now.**
- **13×13 feature map output is obtained, implying they are 32× downsampled**

- **Without anchor boxes, the intermediate model got 69.5% mAP and recall of 81% and with anchor boxes, 69.2% mAP and recall of 88% are obtained.**

5. **Fine-Grained Features**

- **YOLO v2 predicts detections over a 13x13 feature map, which works well for large objects, but detecting smaller objects can benefit from fine-grained features. Fine-grained features refer to feature maps from the earlier layers of the network. YOLO v2 predicts detections over a 13x13 feature map, which works well for large objects, but detecting smaller objects can benefit from fine-grained features. Fine-grained features refer to feature maps from the earlier layers of the network.**
- **Instead of using a proposal network at various layers (feature maps) in the network for multiple resolutions, YOLOv2 adds a passthrough layer.**
- **Since the higher resolution feature map spatial dimensions mismatch with the low-resolution feature map, the high-resolution map 26x26x512 is turned into a 13x13x2048, which is then concatenated with the original 13x13x1024 features.**
- **This concatenation expanded the feature map space to 13x13x3072, providing access to fine-grained features.**
- **This led to a 1% increase in mAP.**

3d

Difference between SORT and DeepSORT

Both the Sort and DeepSORT algorithms are used in object tracking, but they have several key distinctions.

SORT (Simple Online and Realtime Tracking) uses the Kalman filter to track objects in a video stream. The Hungarian algorithm, which resolves the assignment issue between detections and tracks in a frame, is the basis of SORT. Because SORT works frame-by-frame, it simply keeps track of objects in the current frame and ignores their past behavior. In real-time applications like surveillance systems, SORT can be utilized to track things.

DeepSORT (Deep Learning for Multi-Object Tracking) is a buildup on SORT that uses a deep neural network to extract visual features of the object which aids in handling occlusions, missed detections, and false alarms. In order to re-identify missed objects, or objects lost because of occlusions, DeepSORT additionally keeps track of histories of the objects. DeepSORT is capable of handling more complicated situations including tracking many objects at once, across multiple cameras, and for a longer period of time.