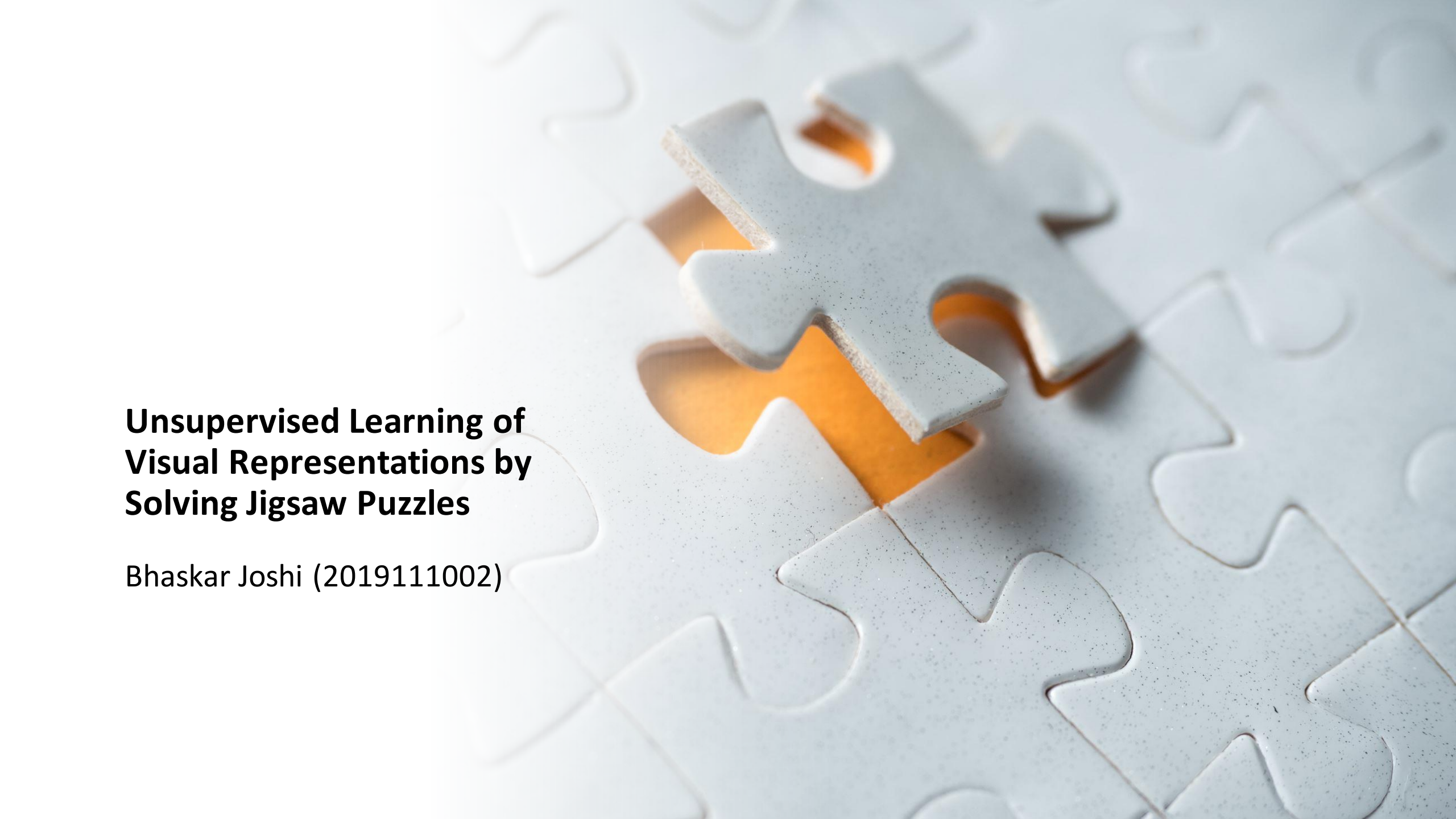
Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles

Bhaskar Joshi (2019111002)



Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles

- **Outline**
  - **Feature Learning by Solving Jigsaw Puzzles**
  - **Context Free Network (CFN): Network Architecture**
  - **Experimentation**

Brief Introduction of Paper

- Paper Objective:

The objective of this is to study the problem of image representation learning without human annotation. The paper aims to build a CNN that could be trained for solving Jigsaw puzzle without the help of manual labelling. It also solves object classification and detection task.

The paper talks about new self supervised task i.e. the jigsaw reassembly problem that is able to perform well when transferred to detection and classification tasks.

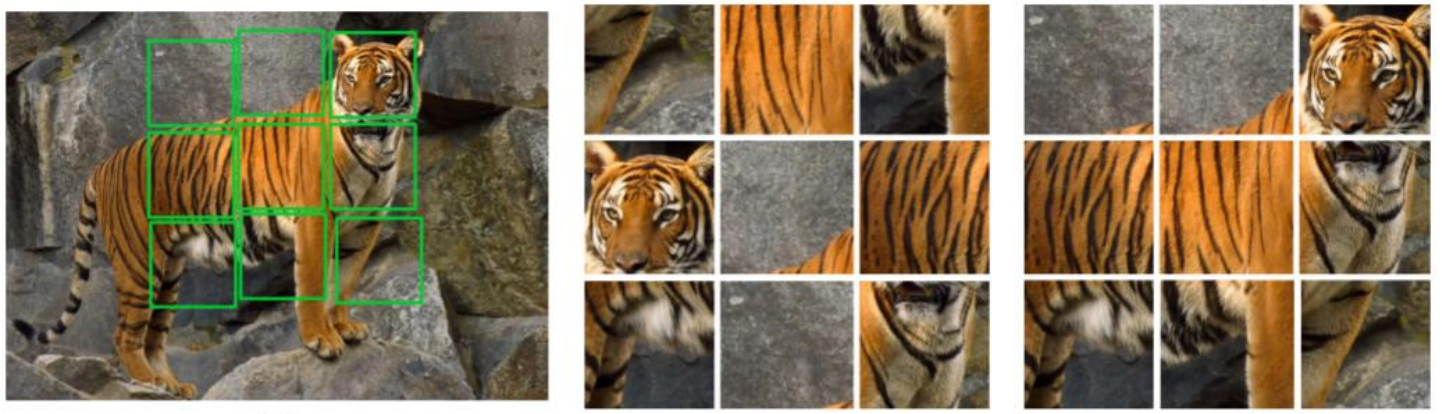
Brief Introduction of Paper

- The Context Free Network (CFN), is designed to take picture tiles as input and output the correct spatial arrangement. This way, useful feature representation is learned and applied to a variety of transfer learning benchmarks (downstream tasks).
- Conceptual Idea:

  Two different-colored cars and two dogs with various fur patterns. Because they are invariant to shared patterns, the characteristics learned to solve puzzles in one (car/dog) image will also apply to the second (car/dog) image.

Fig. 2: Most of the shape of these 2 pairs of images is the same (two separate instances within the same categories). However, some low-level statistics are different (color and texture). The Jigsaw puzzle solver learns to ignore such statistics when they do not help the localization of parts.

Brief Introduction of Paper



(a) (b) (c)

Fig. 1: Learning image representations by solving Jigsaw puzzles. (a) The image from which the tiles (marked with green lines) are extracted. (b) A puzzle obtained by shuffling the tiles. Some tiles might be directly identifiable as object parts, but others are ambiguous (*e.g.*, have similar patterns) and their identification is much more reliable when all tiles are jointly evaluated. In contrast, with reference to (c), determining the relative position between the central tile and the top two tiles from the left can be very challenging [10].

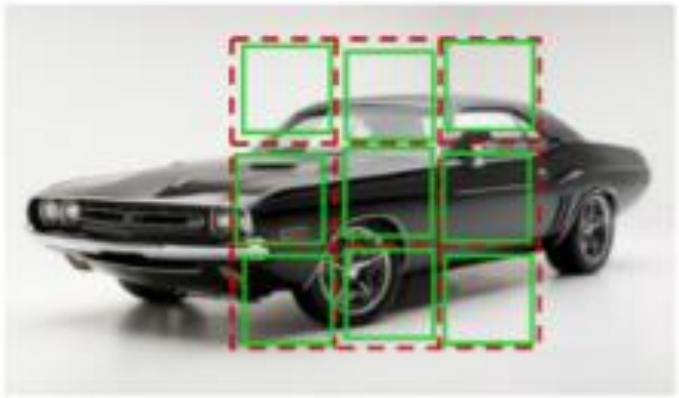
Brief Introduction of Paper **

- Brief description of how authors arrived at convolutional architecture that is capable to solve jigsaw puzzle is to stack the tiles of the puzzle along the channels (i.e., the input data would have 9×3=27 channels) and input these channels into a CNN to solve the Jigsaw puzzles. (**Naïve Stacked Patches NOT Working**)
- The problem with this design is that the network prefers to identify correlations between low-level texture statistics across tiles rather than between the high-level primitives. A CNN with only low-level features learnt is NOT what we want.

*

*(Can be skipped)

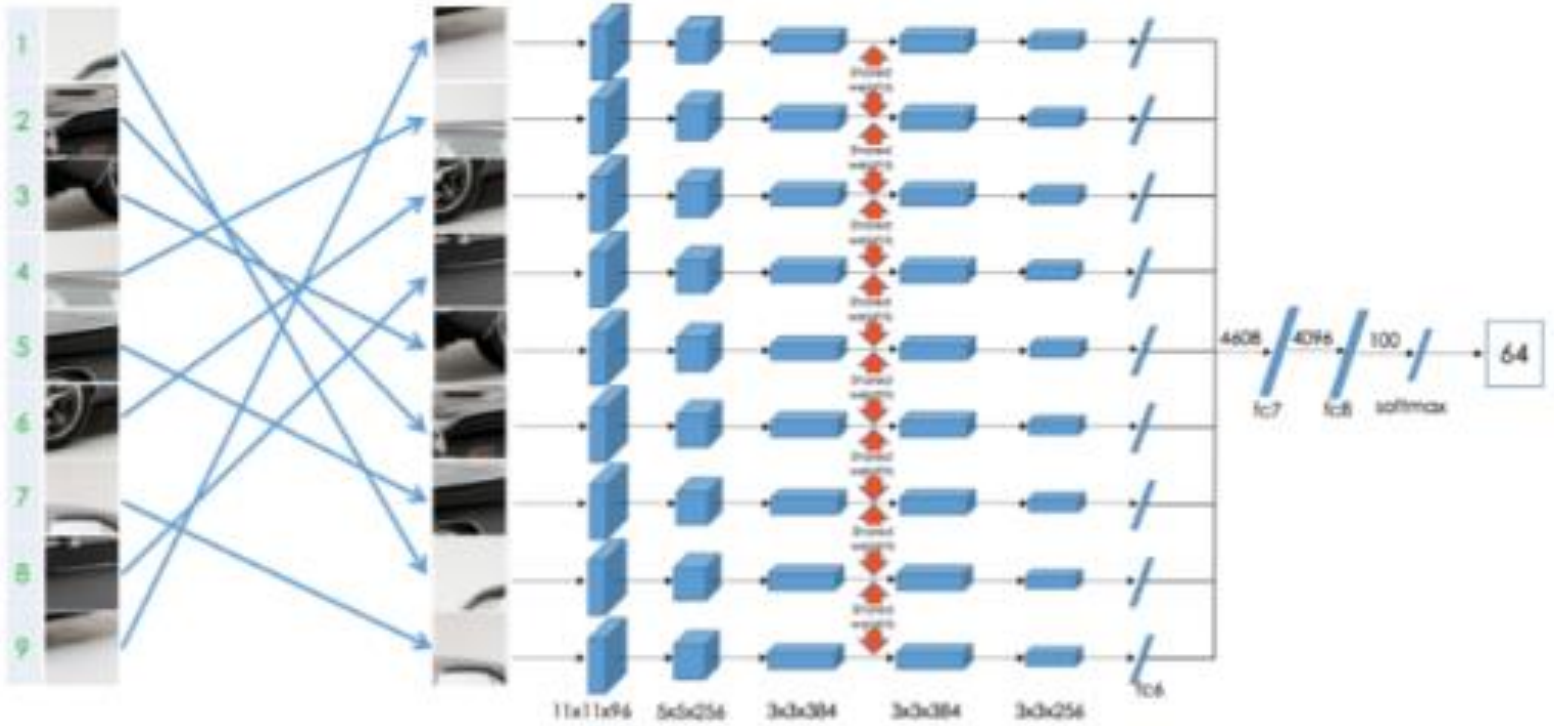
Paper Implementation Description

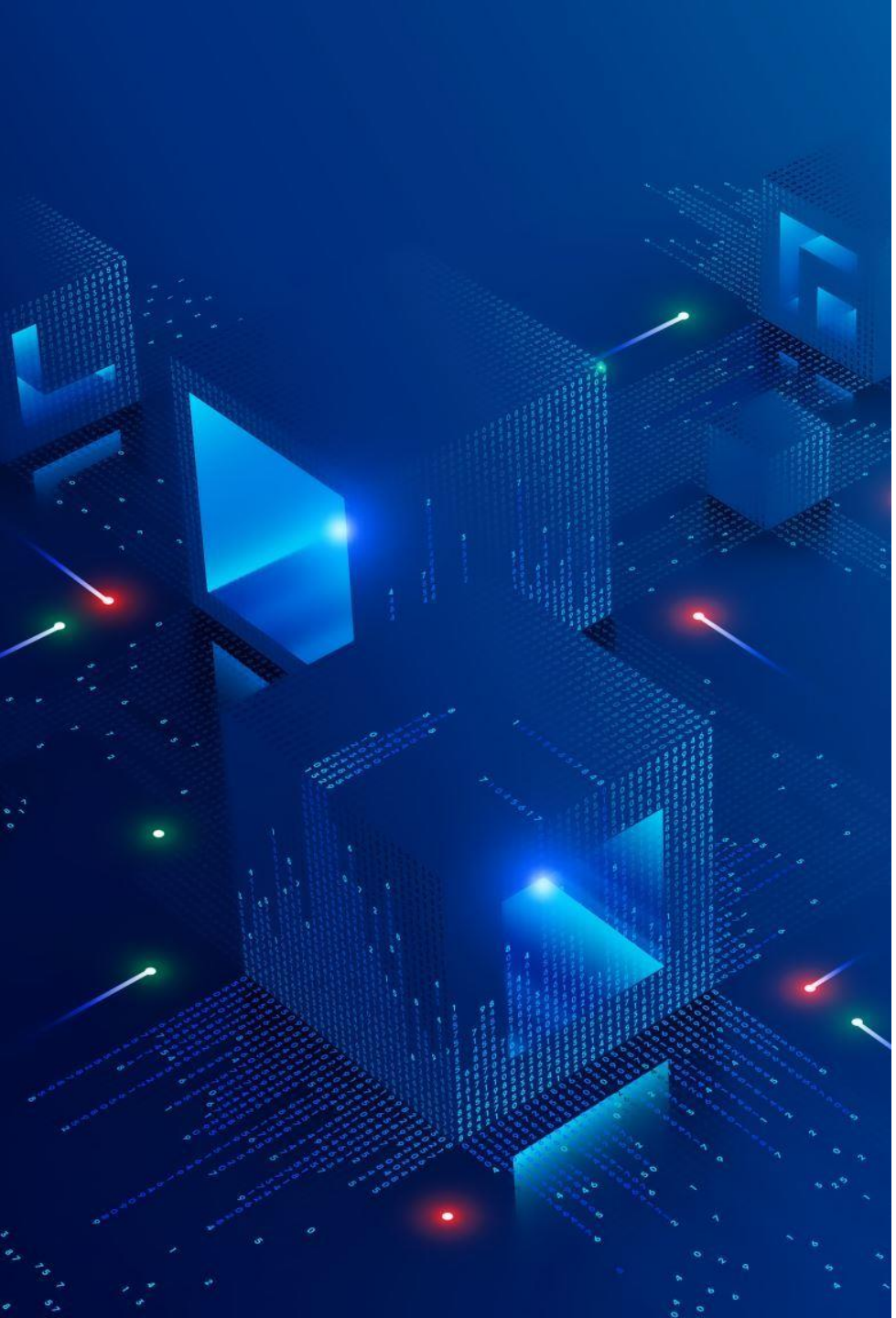


Permutation Set

index	permutation
64	9,4,6,8,3,2,5,1,7

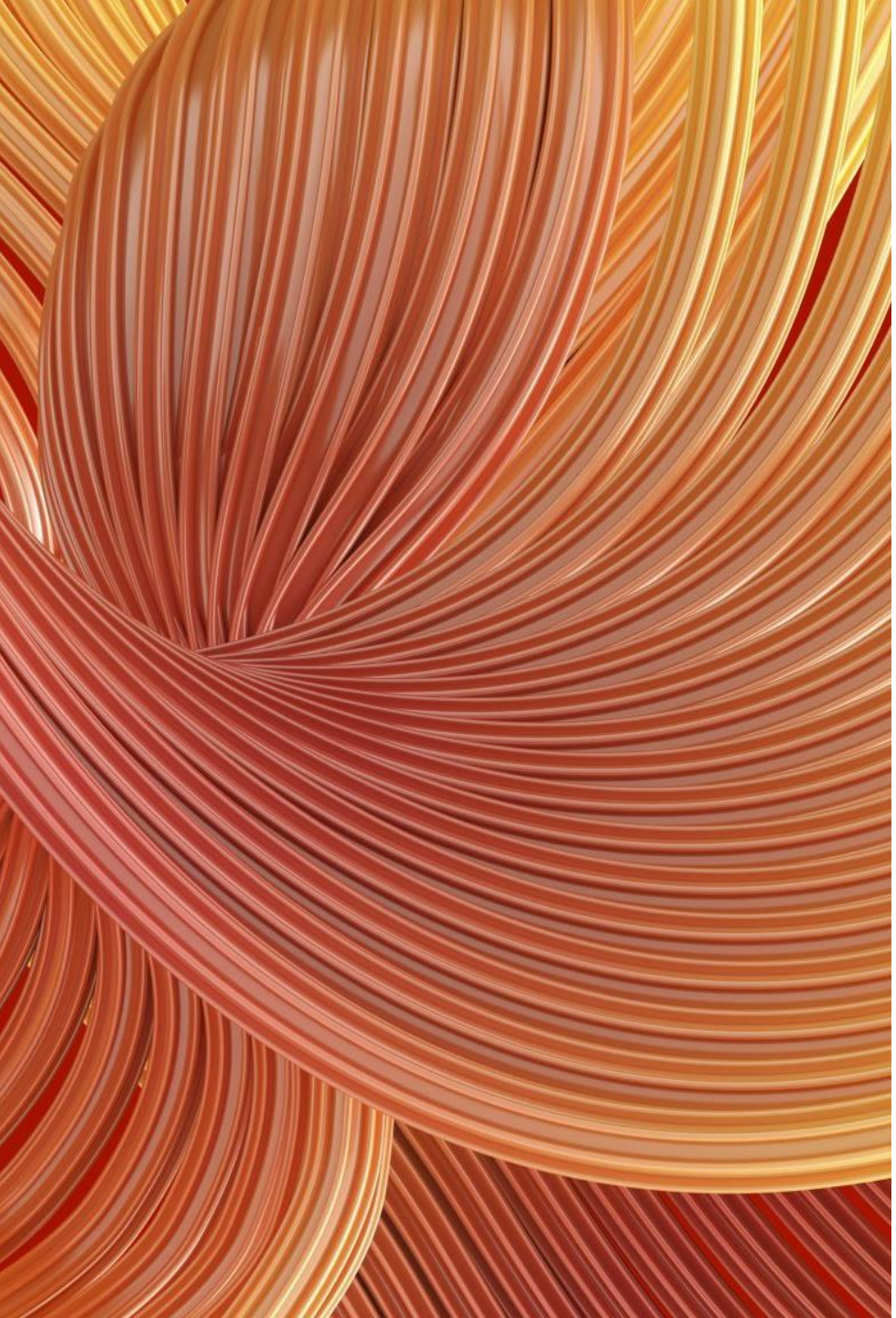
Reorder patches according to the selected permutation





Paper Implementation Description

- The given architecture is called context-free network (CFN) because the data flow of each patch is explicitly separated until the fully connected layer and context is handled only in the last fully connected layers.
- GitHub Code Link: https://github.com/BhaskarJoshi-01/JigSawPuzzle



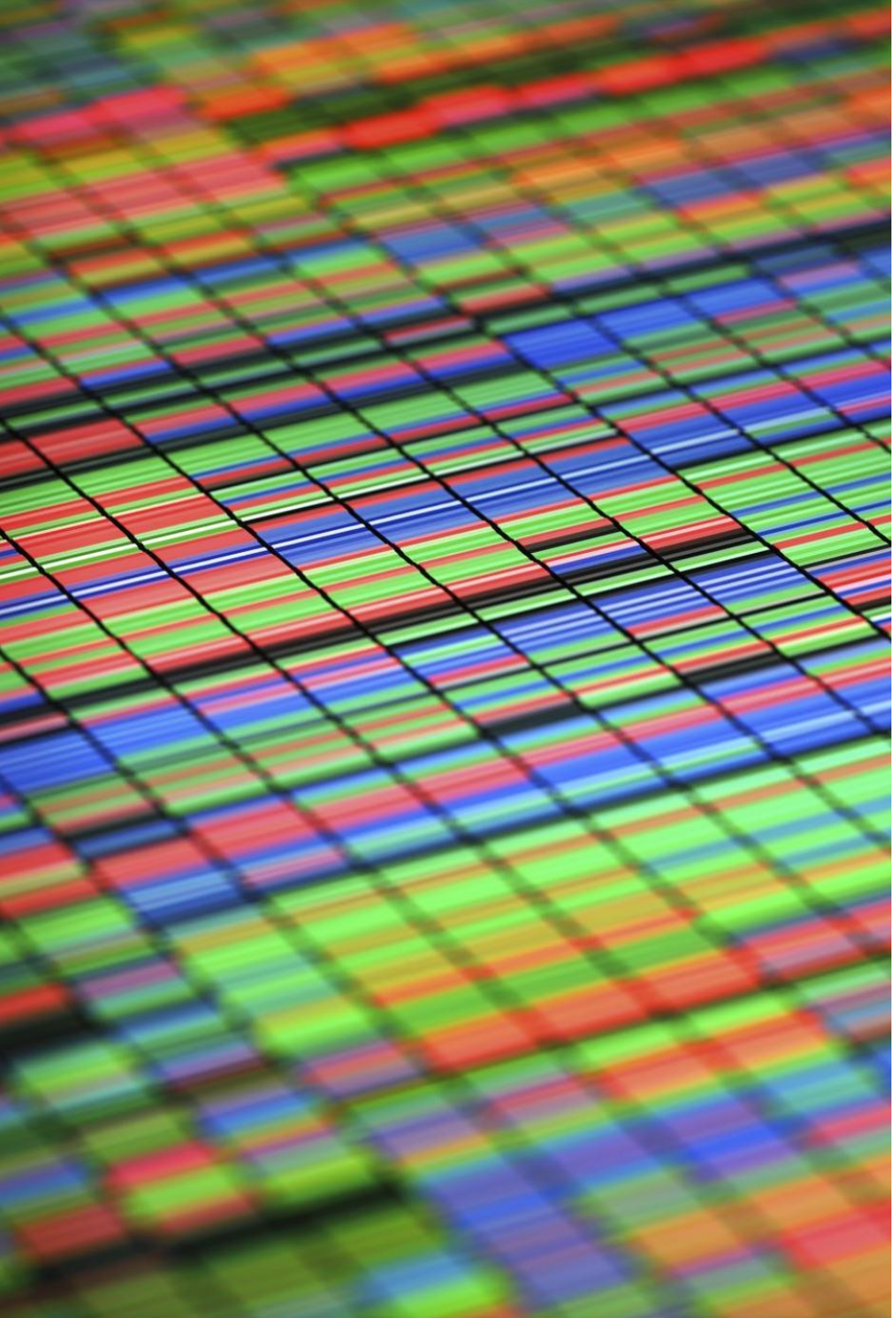
Network Framework Description

First, the Jigsaw puzzle is permuted before being fed into an AlexNet-like CNN.

Then, each patch/puzzle is processed by the CNN. In this case, puzzles in the network's early layers will NOT communicate with one another.

The weights are the same for each branch.

There are numerous permutation options. The goal is to correctly predict the index of the chosen permutation (technically, we define as output a probability vector with 1 at the 64-th location and 0 elsewhere).



Network Training Description

- During training, each input image is resized until either the height or the width matches 256 pixels and preserve the original aspect ratio. Then, a random region is cropped from the resized image of size 225×225 and is split into a 3×3 grid of 75×75 pixels tiles.
- This network can thus be used interchangeably for different tasks including detection and classification.
- This may sound simple but the paper presenters took 2.5 days to train on TitanX GPU. We trained on Ada by submitting it as batch job for 4 days.

Phases of our project work

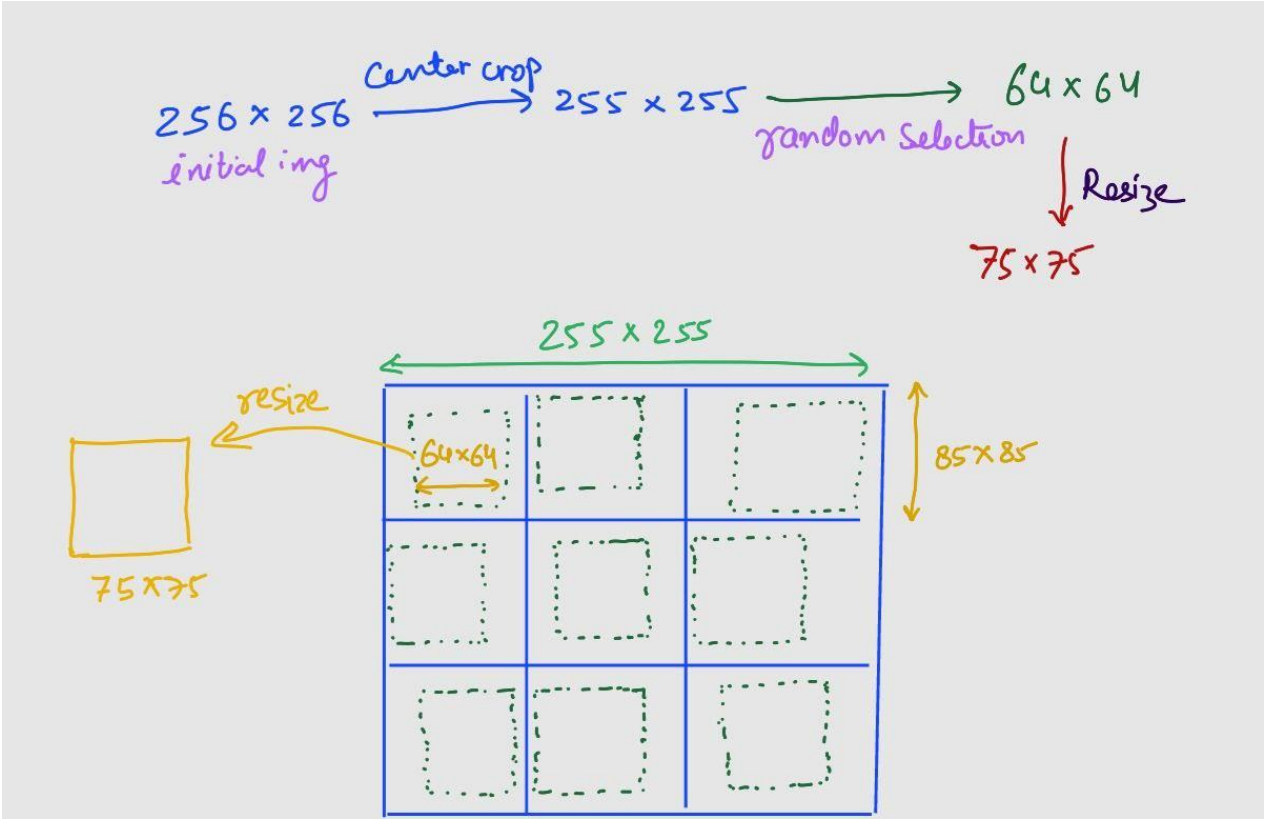
- Dataset loading : Trying out on small miniset and then on 170Gbs of ImageNet dataset. Yeah manipulating data was big problem!!
- Creating the framework: We created a CFN model, and added layers of convolution, ReLU, etc.
- Training: Setting the hyperparameters like learning rate, loss function, etc.

Dataset

- We have used the ImageNeT 2012 dataset for our project.
- It contains around 11 M images.
- Link: https://image-net.org/challenges/LSVRC/index.php

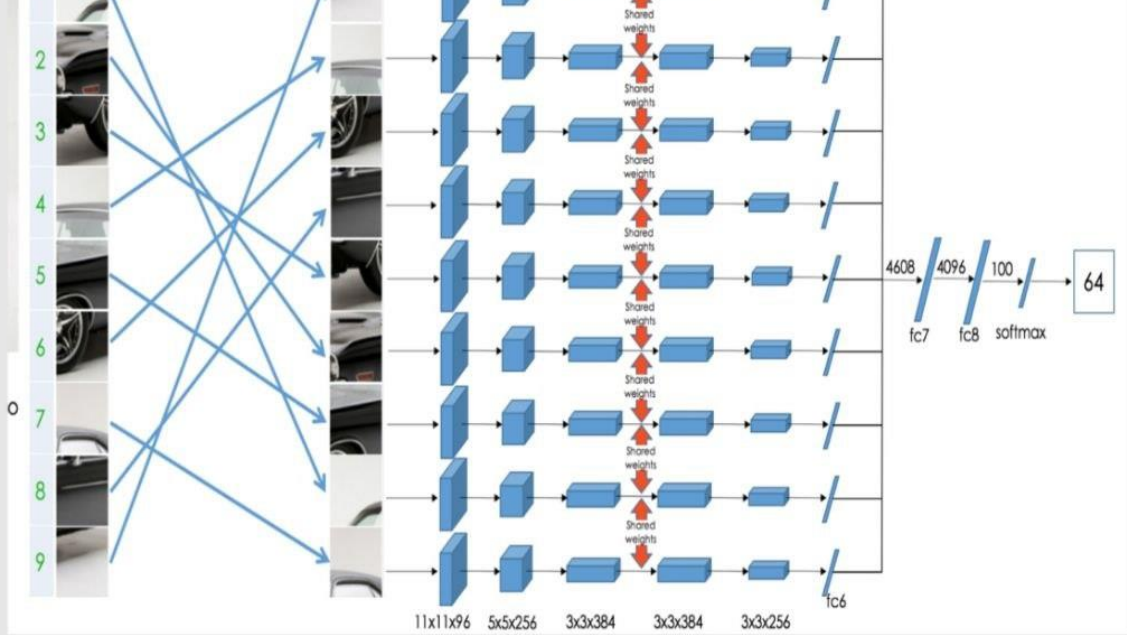
Generating Permutations

- We generated all 9! possible permutations (only done once). Then we picked the best 1000 permutations based on maximizing the metric of Hamming distance and give each one a unique index.
- Now, we permute the image randomly based on one of these 1000 permutations.



Framework

- We have implemented a Context Free Network (CFN) which is like an AlexNet CNN.
- Then we pass each patch/puzzle through our CFN.
- These patches will not communicate with each other within the network layers until the last 3 fully connected layers. This is done to ensure that patches do not learn from each other, preventing the model from learning patterns in the images (low-level feature).
- Each branch shares the same weights.
- There are a lot of permutation choices. The task is to predict the index of the chosen permutation (technically, we define as output a probability vector of size 1000 with 1 at the i^{th} location and 0 elsewhere, where i is the index of the vector predicted).



Picking vector from 1000 permutation set & they have unique index → Alexnet like network → Last 3 layers to get Representation Learning

getting output vector of

index
1
2
3
64
1000
1000x1

1 at index 64 means the vector that was picked corresponds to order of 75th index

JigsawNetwork.py

1

Define a network (add layers Convolution, ReLU, max-pool).

2

Load the model.

3

Weight Initialization (Xavier normalization).

4

Forward propagation, load/create Checkpoints (saving various states of model).

Training

Load train/test/validation data, define batch size = 128.

Learning rate (α) = 0.001

Loss function = Cross-Entropy Loss

Optimizer = Adam

No. of epochs = 70

Experiments/Analysis

- Since, the validation loss was not decreasing during the validation phase, we tried out a few different experiments to try and figure out the problem:

Hyperparameters

Learning Rate (α)

We experimented with various learning rates to see if something was working xD.

The learning rates we tried were: {1e-3,1e-4,1e-2,1e-1}

None of these values for the learning rate seemed to reduce the training loss.

Something fishy seems to be happening in the code ….

Also, batch sizes of 128 and 64 were tried and none seemed to show any significant positive result.

Dataset Change : Checking Data loader part

MNIST

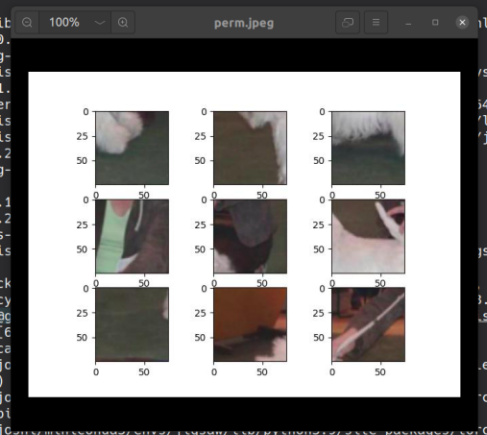
- The first dataset we tried was the MNIST dataset.
- However, the images from this dataset were stored in .csv format, so we had to read data of a particular buffer length from the file, and then manually reshape them, and then finally convert to .png format.
- We divided the dataset into about ~ 40000 images for the training set and ~ 10000 images for the validation set.

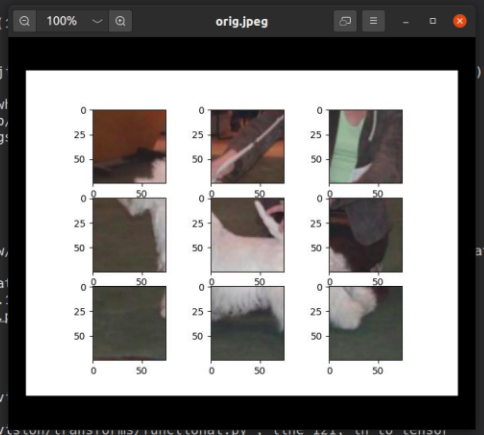
Dataset Change : Checking Data loader part

Smaller ImageNet dataset

- We also tried training our model on a single class dataset of Dog's images obtained from Image NET consisting of ~ 16000 images for the training set, and ~ 2000 images for the validation set.
- Permutation and reordering of the images seemed to work well so it ensures that Dataset loader section was correct, and we need to verify the network.

```
bhaskar.joshi@gnode32:~/cv/JigSawCustom
bhaskar.joshi@gnode32:~/cv/JigSawCustom | bhaskar.joshi@gnode32:~/cv/JigSawCustom/mnist | bhaskar@bhaskar-VivoBook-S15-X530UN:~

aset Permutation :  [3 6 5 8 0 1 7 4 2]
gsaw) bhaskar.joshi@gnode32 ~/cv/JigSawCustom ᚠ main ± ls
pped.jpeg       JigsawNetwork.py  mnist.py       original3.jpeg  original7.jpeg        permuted0.jpeg  permuted4.jpeg  permuted8.jpeg
                Layers.py         original0.jpeg  original4.jpeg  original8.jpeg        permuted1.jpeg  permuted5.jpeg  __pycache__
aset_mnist.py   mnist             original1.jpeg  original5.jpeg  original.jpeg         permuted2.jpeg  permuted6.jpeg  train-images-idx3-ubyte.gz
aset.py         mnist.ipynb       original2.jpeg  original6.jpeg  permutations_1000.npy permuted3.jpeg  permuted7.jpeg  train.py
gsaw) bhaskar.joshi@gnode32 ~/cv/JigSawCustom ᚠ main ± pip install matplotlib
Collecting matplotlib
Using cached matplotlib
Collecting packaging>=20.
Using cached packaging-
Requirement already satis
Collecting kiwisolver>=1.
Using cached kiwisolver
Requirement already satis
Requirement already satis
Collecting pyparsing>=2.2
Using cached pyparsing-
Collecting cycler>=0.10
Using cached cycler-0.1
Collecting fonttools>=4.2
Using cached fonttools-
Requirement already satis
0)
Installing collected pack
Successfully installed cy
gsaw) bhaskar.joshi@g
aset Permutation :  [6
Traceback (most recent ca
File "/home2/bhaskar.jo
tile=transform(tile)
File "/home2/bhaskar.jo
return F.to_tensor(pi
File "/home2/bhaskar.jo
raise TypeError(f"pic should be PIL Image or ndarray. Got {type(pic)}")
TypeError: pic should be PIL Image or ndarray. Got <class 'torch.Tensor'>
gsaw) ✗ bhaskar.joshi@gnode32 ~/cv/JigSawCustom ᚠ main ± python dataset_mnist.py
aset Permutation :  [8 3 7 2 5 4 6 0 1]
gsaw) bhaskar.joshi@gnode32 ~/cv/JigSawCustom ᚠ main ±
```





"perm.jpeg" selected (26.9 kB)

Activities Image Viewer Apr 8 8:31 PM

dataset_mnist.py - cv [SSH: gnode] - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

OPEN EDITORS
- dataset.py JigSawCustom M
- dataset_mnist.py JigSawCust... 1
- original.jpeg JigSawCustom
- mnist.py JigSawCustom
- train.py JigSawCustom
- TrainingUtils.py JigsawPuzzlePytor...
- JigsawTrain.py JigsawPuzz... 5, M
- JigsawNetwork.py JigSawCu... U

CV [SSH: GNODE]
- mnist
- cropped.jpeg
- dataset_mnist.py 1
- dataset.py M
- JigsawNetwork.py U
- Layers.py U
- mnist.ipynb
- mnist.py
- orig.jpeg
- original.jpeg
- original0.jpeg
- original1.jpeg
- original2.jpeg
- original3.jpeg
- original4.jpeg
- original5.jpeg
- original6.jpeg
- original7.jpeg
- original8.jpeg
- perm.jpeg
- permutations_1000.npy

OUTLINE
TIMELINE

dataset.py M | dataset_mnist.py 1 | original.jpeg | mnist.py | train.py | TrainingUtils.py | JigsawTrain.py 5, M | JigsawNet

JigSawCustom › dataset_mnist.py › ...

original Aa ab .* 2 of 4

```python
    @staticmethod
    def rgb_jittering(im):
        im = np.array(im, 'int32')
        for ch in range(3):
            im[:, :, ch] += np.random.randint(-2, 2)
        im[im > 255] = 255
        im[im < 0] = 0
        return im.astype('uint8')


import matplotlib.pyplot as plt
if __name__ == '__main__':
    data_path = 'data/ILSVRC2012_img_train'
    dataset = ImageData(data_path)
    permuted,order,original  = dataset[0]
    print("Dataset Permutation : ",dataset.permutations[order])
    transform=T.ToPILImage()
    f, ax = plt.subplots(3, 3)
    for i,tile in enumerate(original):
        tile=transform(tile)
        ax[i//3][i%3].imshow(tile)
        # tile.save(f"original{i}.jpeg")
    plt.savefig("orig.jpeg")

    f, ax = plt.subplots(3, 3)
    for i,tile in enumerate(permuted):
        tile=transform(tile)
        ax[i//3][i%3].imshow(tile)
        # tile.save(f"permuted{i}.jpeg")
    plt.savefig("perm.jpeg")

```

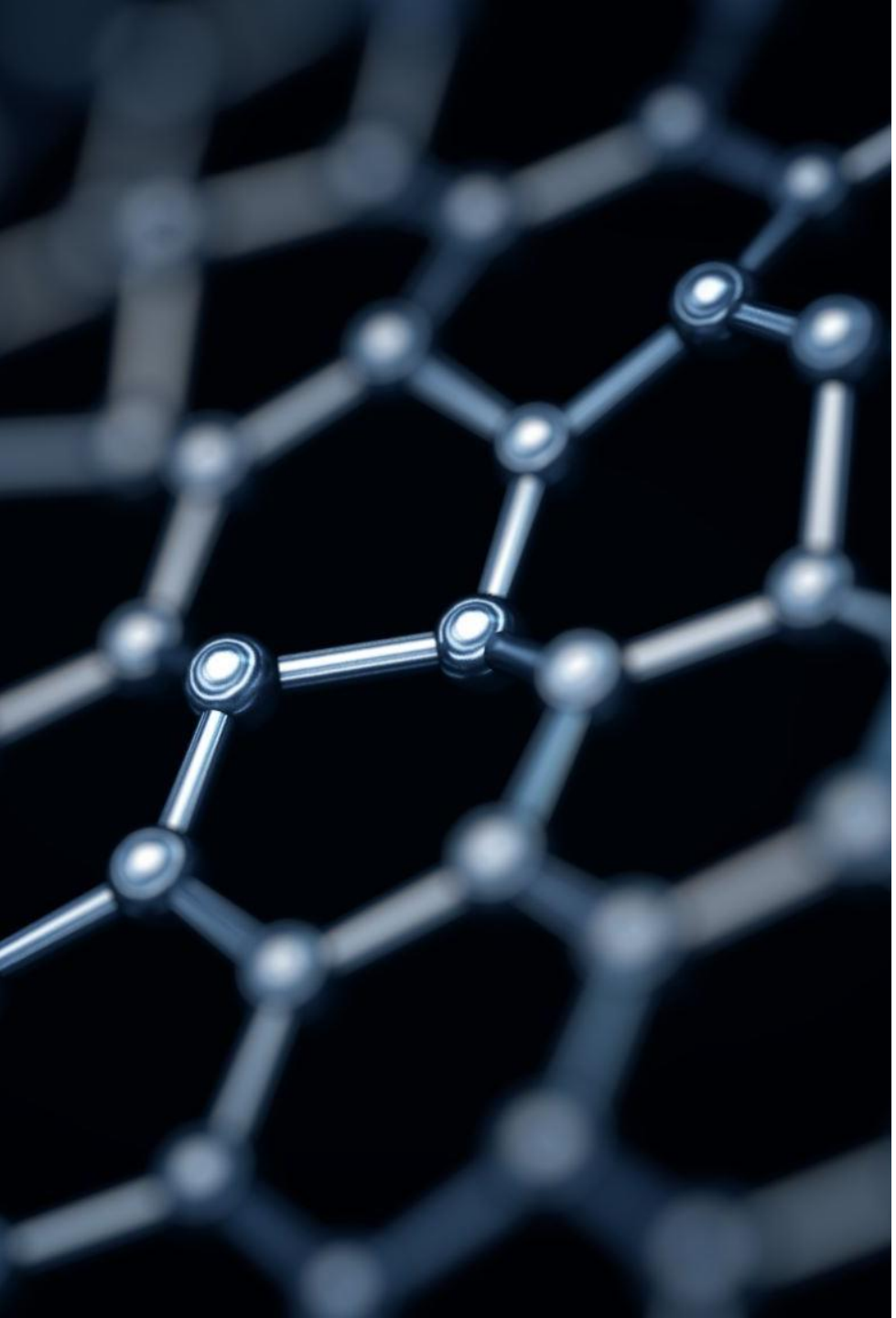
SSH: gnode main* ⊗0 ⚠6 0 Ln 101, Col 27 Spaces: 4 UTF-8 LF Python 3.9.7 ('jigsaw': conda)

LRN (Local Response Normalization) Layer: Changes to Network

The reason we may want to have normalization layers in our CNN is that we want to have some kind of lateral inhibition scheme, where an excited neuron subdues its neighbors.
This results in a significant peak so that we have a form of local maxima.

ReLU neurons have unbounded activations and we need LRN to normalize that. We want to detect high frequency features with a large response. If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors.

At the same time, it will dampen the responses that are uniformly large in any given local neighborhood, and boost the neurons with relatively larger activations.



LRN (Local Response Normalization) Layer

LRN can be performed within the same channel or across multiple channels.

In our code, we tried Response Normalization within the same channel.

Transfer Learning

- We tested model by importing AlexNet model from Pytorch as it was already per-trained on the existing ImageNet dataset.
- The next thing we tried was to freeze the weights of AlexNet model and then try training.
- But this also didn’t give us any useful output.

Trying out GitHub Code

- Next thing was to find some online implementation of the paper.
- Their code was in python 2.7 and there was some conflict in version of pytorch and tensorflow while making conda environment.
- There was an issue with dataloader part of github code as it was choosing some custom and existing dataset images so we needed to tweak it accordingly.
- Next, we trained out on small dataset of github's implemendation and found that their code was not converging and so we trained it for 4 days on Imagenet dataset.

Trying out GitHub Code

Code

Edit

facebookresearch/vissl Quickstart in Colab	★ 2,599	PyTorch
bbrattoli/JigsawPuzzlePytorch	★ 132	PyTorch
gitlimlab/Representation-Learning-b...	★ 107	TensorFlow
Confusezius/selfsupervised_learning	★ 9	PyTorch
virtualgraham/sc_patch	★ 9	PyTorch
ferrannoguera/MLMI-Transfromers	★ 2	PyTorch
SharadGitHub/Self-Supervised-Autoen...	★ 1	PyTorch

Collapse 7 implementations

Datasets


ImageNet

Author's Train Loss

```
Using GPU 0
Process number: 36256
Images: train 16464, validation 2058
Start training: lr 0.000100, batch size 128, classes 1000
Checkpoint: checkpoints/
Learning Rate 0.000100
[ 1/70]     0) [batch load  4.356sec, net 0.28sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 1/70]    20) [batch load  0.227sec, net 0.03sec], LR 0.00010, Loss:  6.906, Accuracy  0.78%
[ 1/70]    40) [batch load  0.130sec, net 0.02sec], LR 0.00010, Loss:  6.908, Accuracy  0.00%
[ 1/70]    60) [batch load  0.094sec, net 0.02sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 1/70]    80) [batch load  0.075sec, net 0.02sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 1/70]   100) [batch load  0.020sec, net 0.02sec], LR 0.00010, Loss:  6.908, Accuracy  0.78%
[ 1/70]   120) [batch load  0.017sec, net 0.01sec], LR 0.00010, Loss:  6.906, Accuracy  0.00%
Learning Rate 0.000100
[ 2/70]   140) [batch load  0.043sec, net 0.02sec], LR 0.00010, Loss:  6.906, Accuracy  0.00%
[ 2/70]   160) [batch load  0.042sec, net 0.01sec], LR 0.00010, Loss:  6.906, Accuracy  0.00%
[ 2/70]   180) [batch load  0.043sec, net 0.02sec], LR 0.00010, Loss:  6.906, Accuracy  0.78%
[ 2/70]   200) [batch load  0.044sec, net 0.02sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 2/70]   220) [batch load  0.047sec, net 0.02sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 2/70]   240) [batch load  0.017sec, net 0.01sec], LR 0.00010, Loss:  6.907, Accuracy  0.78%
Learning Rate 0.000100
[ 3/70]   260) [batch load  0.045sec, net 0.01sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 3/70]   280) [batch load  0.045sec, net 0.02sec], LR 0.00010, Loss:  6.905, Accuracy  0.00%
[ 3/70]   300) [batch load  0.046sec, net 0.02sec], LR 0.00010, Loss:  6.906, Accuracy  1.56%
[ 3/70]   320) [batch load  0.045sec, net 0.02sec], LR 0.00010, Loss:  6.906, Accuracy  0.00%
[ 3/70]   340) [batch load  0.047sec, net 0.02sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
[ 3/70]   360) [batch load  0.020sec, net 0.01sec], LR 0.00010, Loss:  6.906, Accuracy  0.78%
[ 3/70]   380) [batch load  0.016sec, net 0.01sec], LR 0.00010, Loss:  6.907, Accuracy  0.00%
```

```
Learning Rate 0.000100
[70/70]  8960) [batch load  0.049sec, net 0.01sec], LR 0.00010, Loss:  6.900, Accuracy  0.00%
[70/70]  8980) [batch load  0.048sec, net 0.01sec], LR 0.00010, Loss:  6.901, Accuracy  0.00%
[70/70]  9000) [batch load  0.050sec, net 0.01sec], LR 0.00010, Loss:  6.902, Accuracy  0.78%
[70/70]  9020) [batch load  0.049sec, net 0.01sec], LR 0.00010, Loss:  6.902, Accuracy  0.00%
[70/70]  9040) [batch load  0.053sec, net 0.01sec], LR 0.00010, Loss:  6.903, Accuracy  0.78%
[70/70]  9060) [batch load  0.025sec, net 0.01sec], LR 0.00010, Loss:  6.902, Accuracy  0.00%
[70/70]  9080) [batch load  0.023sec, net 0.01sec], LR 0.00010, Loss:  6.904, Accuracy  0.78%
```

Author's Validation Loss

logs_val_0.0001_.txt

```
1  Using GPU 0
2  Process number: 1409
3  Images: train 16464, validation 2058
4  ('Starting from: ', 'jps_007_001000.pth.ta
5  Evaluating network.......
6  TESTING: 0), Accuracy 0.05%
7
```

train_logs_custom_0.001.txt

```
Using device:  cuda
Iteration: 20, Epoch: 1, Loss: 6.9091
Iteration: 40, Epoch: 1, Loss: 6.9132
Iteration: 60, Epoch: 1, Loss: 6.9106
Iteration: 80, Epoch: 1, Loss: 6.9055
Iteration: 100, Epoch: 1, Loss: 6.9062
Iteration: 120, Epoch: 1, Loss: 6.9080
Iteration: 140, Epoch: 1, Loss: 6.9078
Iteration: 160, Epoch: 1, Loss: 6.9151
Iteration: 180, Epoch: 1, Loss: 6.9023
Iteration: 200, Epoch: 1, Loss: 6.9165
Iteration: 220, Epoch: 1, Loss: 6.9078
Iteration: 240, Epoch: 1, Loss: 6.9053
Epoch [1/70], Loss: 6.9049
Accuracy of the network on the 2058 test images: 0.04295432458697766

Iteration: 20, Epoch: 2, Loss: 6.9083
Iteration: 40, Epoch: 2, Loss: 6.9096
Iteration: 60, Epoch: 2, Loss: 6.9020
Iteration: 80, Epoch: 2, Loss: 6.9048
Iteration: 100, Epoch: 2, Loss: 6.9097
Iteration: 120, Epoch: 2, Loss: 6.9052
Iteration: 140, Epoch: 2, Loss: 6.9075
Iteration: 160, Epoch: 2, Loss: 6.9065
Iteration: 180, Epoch: 2, Loss: 6.9075
Iteration: 200, Epoch: 2, Loss: 6.9022
Iteration: 220, Epoch: 2, Loss: 6.9112
Iteration: 240, Epoch: 2, Loss: 6.9041
Epoch [2/70], Loss: 6.9233
Accuracy of the network on the 2058 test images: 0.04859086491739553

```

Our Train and Validation Loss (initial)

- implemented without the AlexNet pre-trained model

```
Iteration: 20, Epoch: 69, Loss: 6.9064
Iteration: 40, Epoch: 69, Loss: 6.9058
Iteration: 60, Epoch: 69, Loss: 6.9080
Iteration: 80, Epoch: 69, Loss: 6.9108
Iteration: 100, Epoch: 69, Loss: 6.9101
Iteration: 120, Epoch: 69, Loss: 6.9065
Iteration: 140, Epoch: 69, Loss: 6.9168
Iteration: 160, Epoch: 69, Loss: 6.9159
Iteration: 180, Epoch: 69, Loss: 6.9064
Iteration: 200, Epoch: 69, Loss: 6.9084
Iteration: 220, Epoch: 69, Loss: 6.9064
Iteration: 240, Epoch: 69, Loss: 6.9029
Epoch [69/70], Loss: 6.8996
Accuracy of the network on the 2058 test images: 0.0457725947521866

Iteration: 20, Epoch: 70, Loss: 6.9059
Iteration: 40, Epoch: 70, Loss: 6.9130
Iteration: 60, Epoch: 70, Loss: 6.9089
Iteration: 80, Epoch: 70, Loss: 6.9034
Iteration: 100, Epoch: 70, Loss: 6.9100
Iteration: 120, Epoch: 70, Loss: 6.9035
Iteration: 140, Epoch: 70, Loss: 6.9083
Iteration: 160, Epoch: 70, Loss: 6.9031
Iteration: 180, Epoch: 70, Loss: 6.9063
Iteration: 200, Epoch: 70, Loss: 6.9087
Iteration: 220, Epoch: 70, Loss: 6.9053
Iteration: 240, Epoch: 70, Loss: 6.9061
Epoch [70/70], Loss: 6.8889
Accuracy of the network on the 2058 test images: 0.04859086491739553
```

Our Train and Validation Loss (initial)

train_logs_alex_0.0001.txt

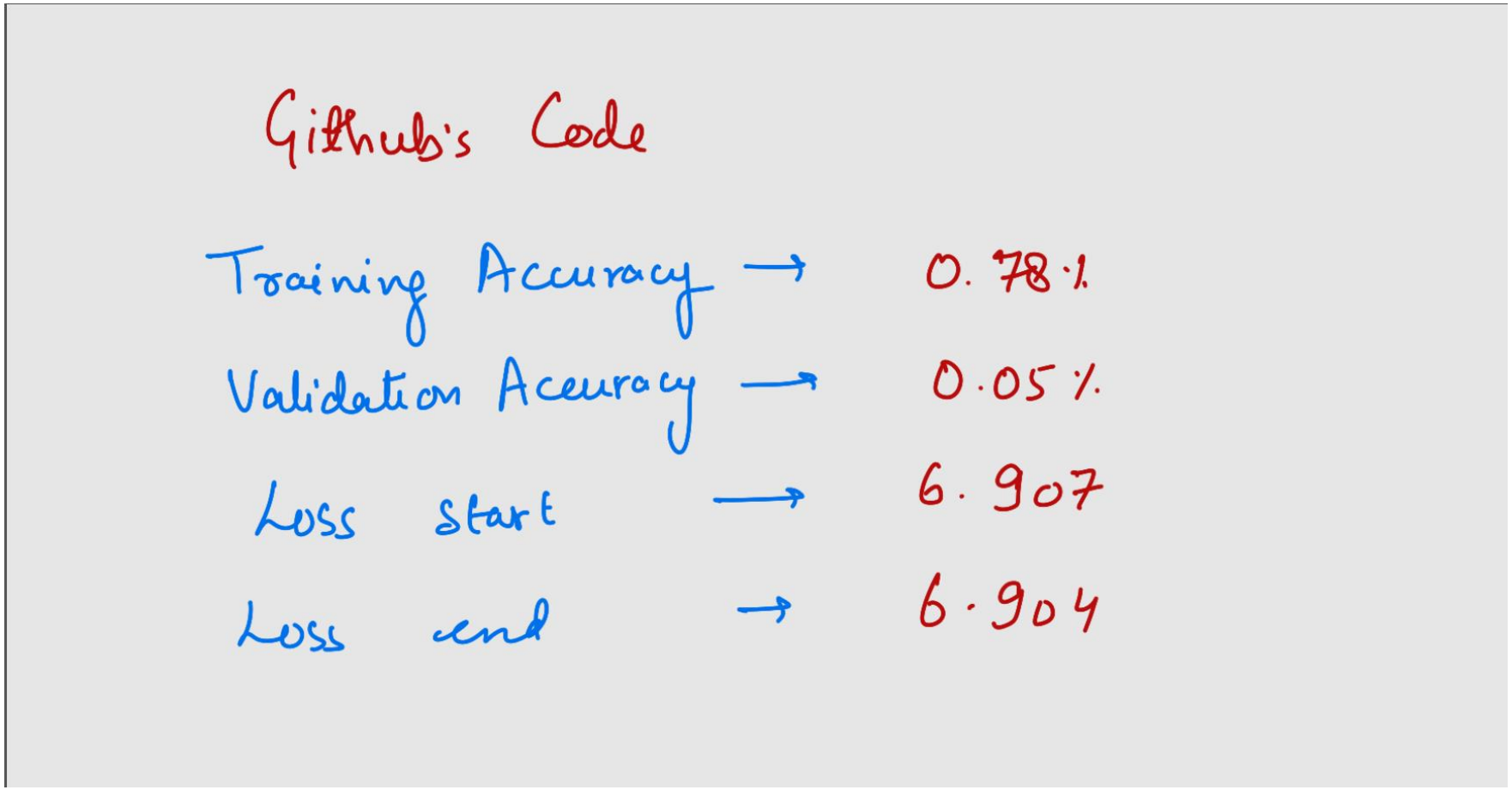
```
Using device:  cpu
Iteration: 20, Epoch: 1, Loss: 6.9102
Iteration: 40, Epoch: 1, Loss: 6.9109
Iteration: 60, Epoch: 1, Loss: 6.9085
Iteration: 80, Epoch: 1, Loss: 6.9053
Iteration: 100, Epoch: 1, Loss: 6.9065
Iteration: 120, Epoch: 1, Loss: 6.9129
Iteration: 140, Epoch: 1, Loss: 6.9091
Iteration: 160, Epoch: 1, Loss: 6.9039
Iteration: 180, Epoch: 1, Loss: 6.8944
Iteration: 200, Epoch: 1, Loss: 6.9036
Iteration: 220, Epoch: 1, Loss: 6.9057
Iteration: 240, Epoch: 1, Loss: 6.9091
Epoch [1/70], Loss: 6.9199
Accuracy of the network on the 2058 test images: 0.1457725947521866

```

AlexNet Training Loss and Accuracy

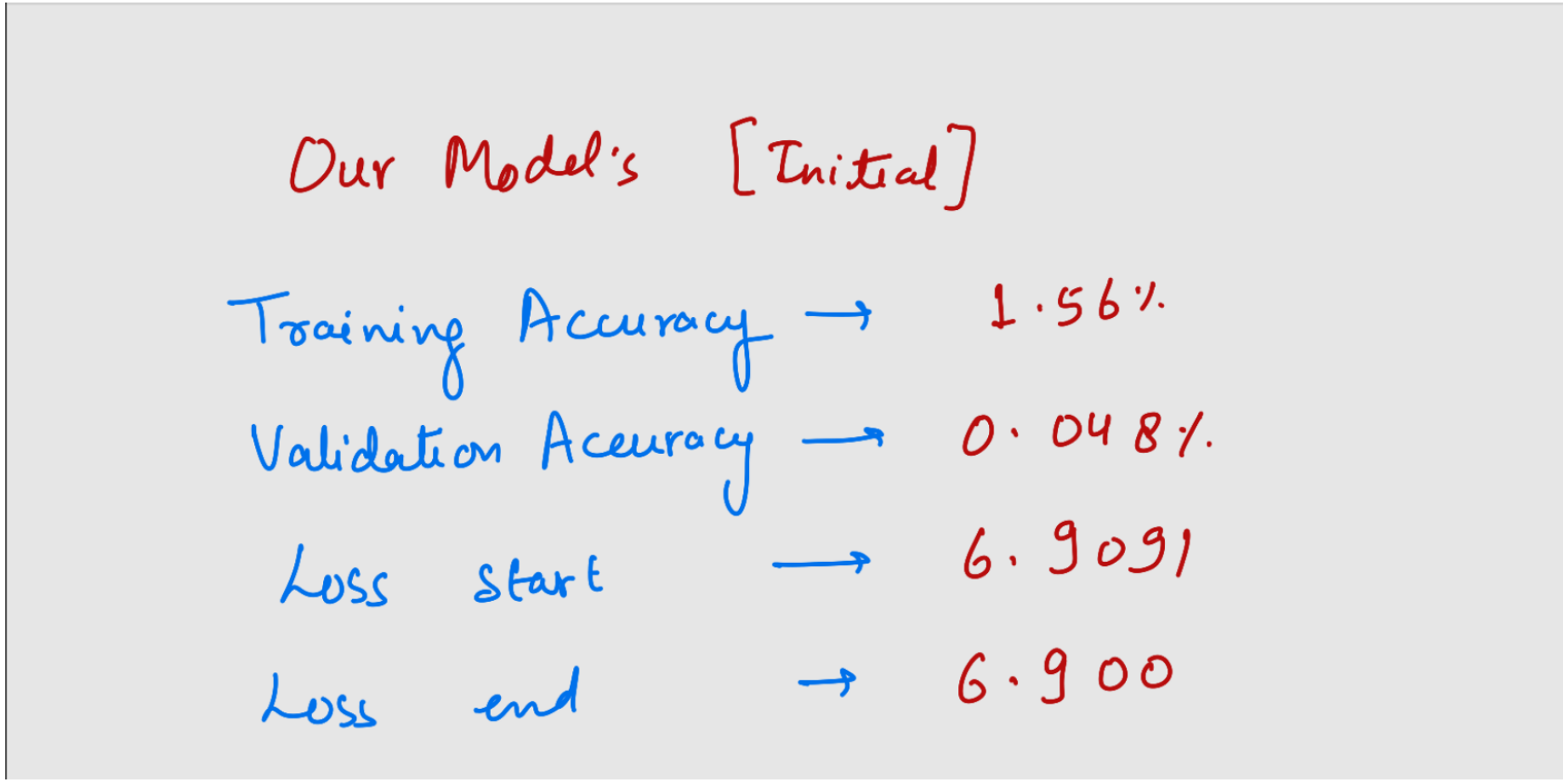
```
Iteration: 20, Epoch: 2, Loss: 6.9015
Iteration: 40, Epoch: 2, Loss: 6.9072
Iteration: 60, Epoch: 2, Loss: 6.9112
Iteration: 80, Epoch: 2, Loss: 6.9063
Iteration: 100, Epoch: 2, Loss: 6.9052
Iteration: 120, Epoch: 2, Loss: 6.9092
Iteration: 140, Epoch: 2, Loss: 6.9067
Iteration: 160, Epoch: 2, Loss: 6.9085
Iteration: 180, Epoch: 2, Loss: 6.9060
Iteration: 200, Epoch: 2, Loss: 6.9075
Iteration: 220, Epoch: 2, Loss: 6.9068
Iteration: 240, Epoch: 2, Loss: 6.9068
Epoch [2/70], Loss: 6.9052
Accuracy of the network on the 2058 test images: 0.048590864917395534

Iteration: 20, Epoch: 3, Loss: 6.9113
Iteration: 40, Epoch: 3, Loss: 6.9080
Iteration: 60, Epoch: 3, Loss: 6.9068
Iteration: 80, Epoch: 3, Loss: 6.9065
Iteration: 100, Epoch: 3, Loss: 6.9041
Iteration: 120, Epoch: 3, Loss: 6.9080
Iteration: 140, Epoch: 3, Loss: 6.9088
Iteration: 160, Epoch: 3, Loss: 6.9088
Iteration: 180, Epoch: 3, Loss: 6.9062
Iteration: 200, Epoch: 3, Loss: 6.9054
Iteration: 220, Epoch: 3, Loss: 6.9027
Iteration: 240, Epoch: 3, Loss: 6.9047
Epoch [3/70], Loss: 6.9040
Accuracy of the network on the 2058 test images: 0.09718172983479106
```

AlexNet Training Loss and Accuracy (with frozen weights)

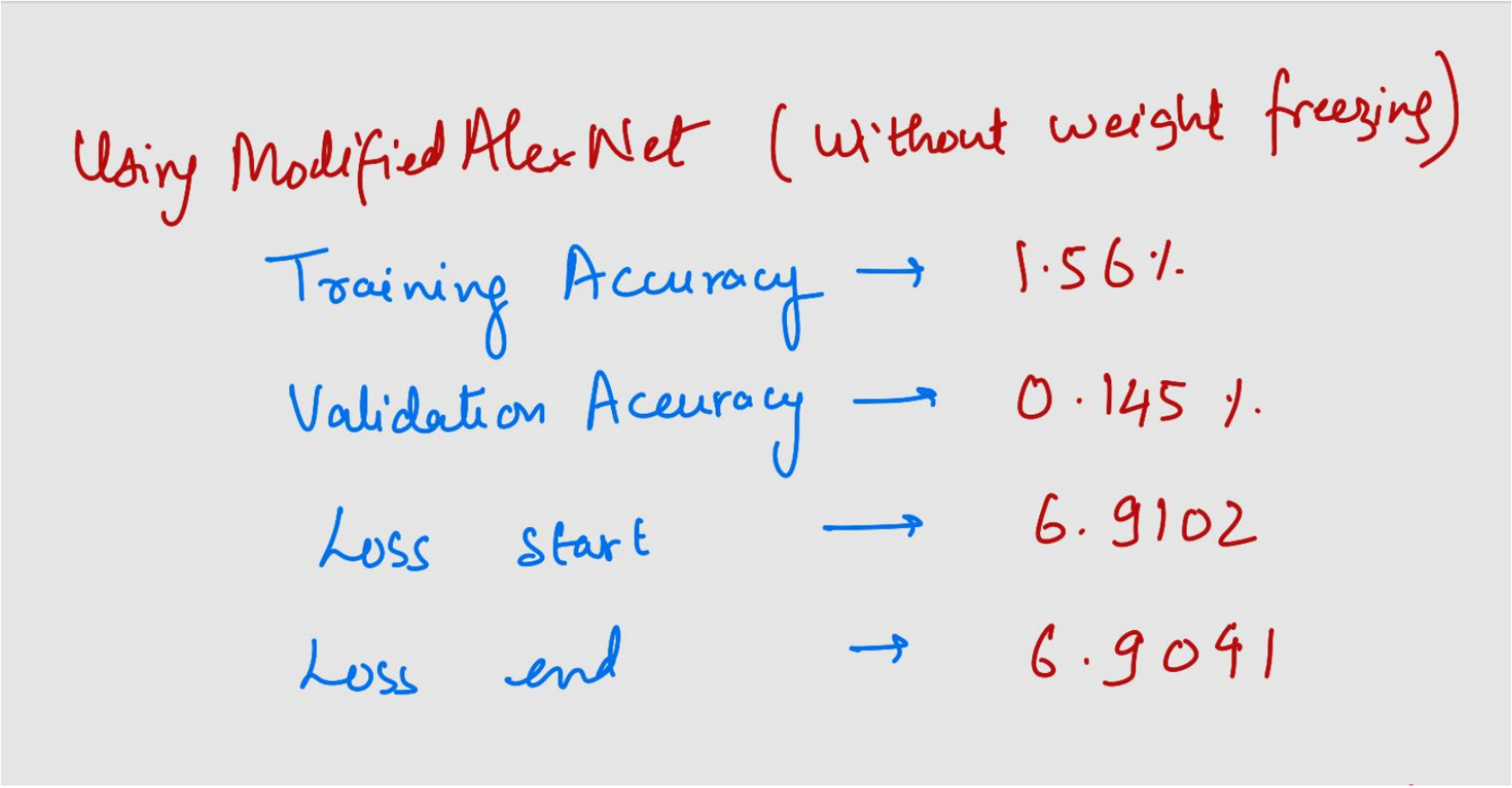
Comparisons



Comparisons



Comparisons



Comparisons

Using Modified AlexNet (with weight freezing)

Training Accuracy → 1.56%

Validation Accuracy → 0.097%

Loss start → 6.9015

Loss end → 6.9017

Avoiding Shortcuts

- Feature association with position: The CFN may accidentally learn to associate features with absolute positions if only 1 Jigsaw puzzle is generated per image.

$$p(S|A_1, A_2, \ldots, A_9) = p(S|F_1, F_2, \ldots, F_9) \prod_{i=1}^{9} p(F_i|A_i) \qquad (1)$$

- This shortcut can be avoided by feeding the CFN multiple Jigsaw puzzles of the same image, making sure that tiles are shuffled as much as possible and have adequately large average Hamming distance, ensuring that the same tile can be assigned to multiple positions within the grid and avoiding the correlation of an absolute position.

Algorithm 1. Generation of the *maximal* Hamming distance permutation set

Input: N \\ *number of permutations*
Output: P \\ *maximal permutation set*
$\bar{P} \leftarrow$ all permutations $[\bar{P}_1, \ldots, \bar{P}_{9!}]$ \\ $\bar{P}$ *is a* $9 \times 9!$ *matrix*
$P \leftarrow \emptyset$
$j \sim \mathcal{U}[1, 9!]$ \\ *uniform sample out of* 9! *permutations*
$i \leftarrow 1$
repeat
    $P \leftarrow [P\ \bar{P}_j]$ \\ *add permutation* $\bar{P}_j$ *to* P
    $\bar{P} \leftarrow [\bar{P}_1, \ldots, \bar{P}_{j-1}, \bar{P}_{j+1}, \ldots]$ \\ *remove* $\bar{P}_j$ *from* $\bar{P}$
    $D \leftarrow \text{Hamming}(P, P')$ \\ D *is an* $i \times (9! - i)$ *matrix*
    $\bar{D} \leftarrow \mathbf{1}^T D$ \\ $\bar{D}$ *is a* $1 \times (9! - i)$ *row vector*
    $j \leftarrow \arg\max_k \bar{D}_k$ \\ $\bar{D}_k$ *denotes the k-th entry of* $\bar{D}$
    $i \leftarrow i + 1$
until $i \leq N$

Avoiding Shortcuts

- Low-Level Statistics: These can include the mean and start deviation of the pixel intensities, allowing the model to determine the patch arrangement. As a result, the mean and standard deviation are normalized for each patch.
- Edge Continuity: Another shortcut is for the CFN to become aware of edge continuity and pixel intensity distribution. By addressing this issue, the CFN does not learn low-level statistics. This includes resizing input images and cropping a random region to leave random gaps between tiles ranging in size from 0 to 22 pixels. A 21-pixel gap between tiles is achieved by randomly selecting 64x64 pixel tiles from the 85x85 pixel cells.

Avoiding Shortcuts

- Chromic Aberration: A third shortcut can be attributed to chromatic aberration, which is a relative spatial shift between colour channels that increases from the center images to the borders, assisting the network in estimating the tiles. This is resolved by cropping and resizing the original image's central square, training with both colour and grayscale images, jittering colour channels, and using greyscale images.

Some References

- https://arxiv.org/pdf/1603.09246v3.pdf
- https://paperswithcode.com/paper/unsupervised-learning-of-visual-1#code
- https://towardsdatascience.com/difference-between-local-response-normalization-and-batch-normalization-272308c034ac
- https://towardsdatascience.com/transfer-learning-using-pre-trained-alexnet-model-and-fashion-mnist-43898c2966fb

Thank You !

We would like to thank Samartha Sir for being available for meet without prescheduling and providing idea about how to try to improve our paper implementation.