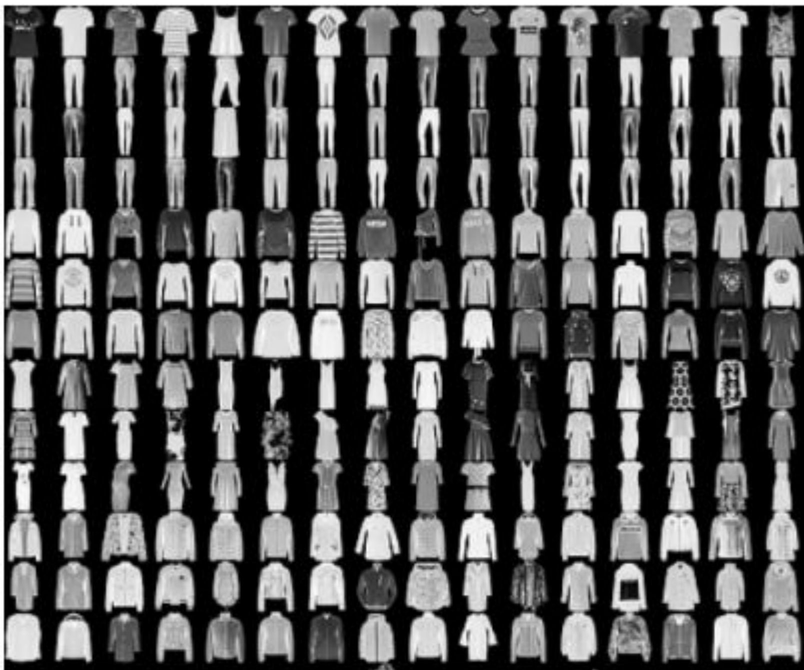
Problem 2 - Representation Learning



We will be using the Fashion MNIST dataset to showcase how self-supervised representation learning can be utilized for more efficient training in downstream tasks.

1. Train a classifier from scratch on the Fashion MNIST dataset and observe how fast and well it learns.
2. Train useful representations via predicting image rotations, rather than classifying clothing types.
3. Transfer our rotation pretraining features to solve the classification task with much less data than in step 1.

Data Preparation

Creating a Dataset in PyTorch

- Implement these 3 functions at the minimum
- __len__: directly return the size
- __getitem__:
  - Randomly choose an angle from the list to rotate
  - Rotate the image according to the chosen angle, using Image.rotate

```python
class MNISTDatasetWrapper(Dataset):
    """
    Comment
    """

    def __init__(self):
        """
            Store any information you'll need to create
            the dataset and sample from it
        """
        pass

    def __len__(self):
        """
            Return the size of the dataset
        """
        pass

    def __getitem__(self, idx):
        """
            Return the sample at idx
        """
        pass
```

Creating a Model

- __init__: define the model structure
- forward: pass the input through the model
- Using
  - nn.Conv2d
  - nn.MaxPool2d
  - nn.ReLU
  - nn.Linear
  - nn.Flatten
  - model.parameters() gives you a list of parameters of model

```
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out),
        ).to(device)
```

```
y_pred = model(x)
```

Training

- Define model, loss, and optimizer
- In a loop, do
  - Pass input through model
  - Compute loss using output and label
  - Zero out gradients in the optimizer
  - Do a backward pass
  - Update the weights by stepping the optimizer

```
# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Construct our model by instantiating the class defined above.
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
loss_fn = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
  # Forward pass: Compute predicted y by passing x to the model
  y_pred = model(x)

  # Compute and print loss
  loss = loss_fn(y_pred, y)
  print(t, loss.item())

  # Zero gradients, perform a backward pass, and update the weights.
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

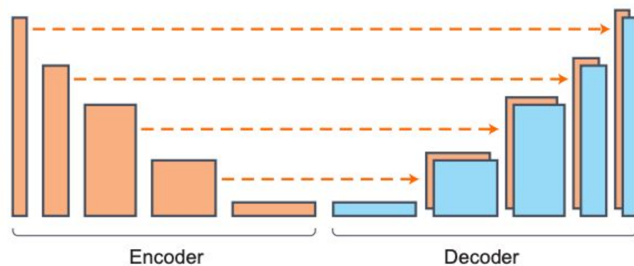
Problem 3 - Supervised Monocular Depth Estimation

High Quality Monocular Depth Estimation via Transfer Learning

Ibraheem Alhashim
KAUST
ibraheem.alhashim@kaust.edu.sa

Peter Wonka
KAUST
pwonka@gmail.com

Input

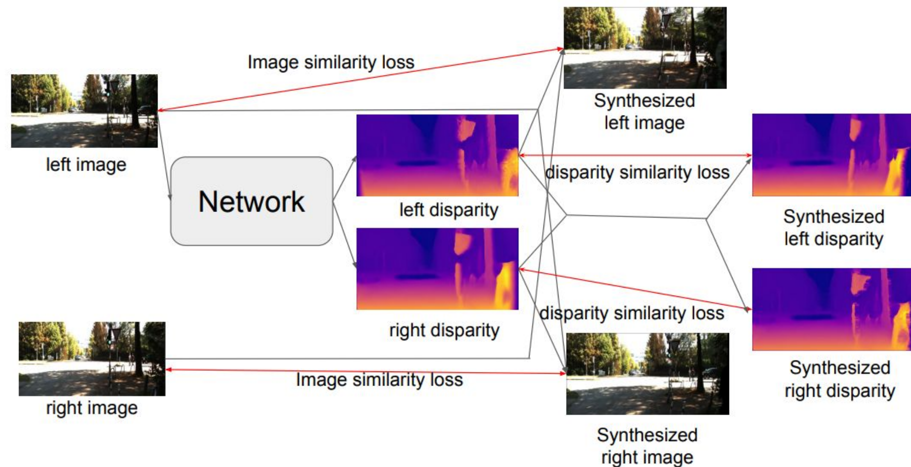


Output

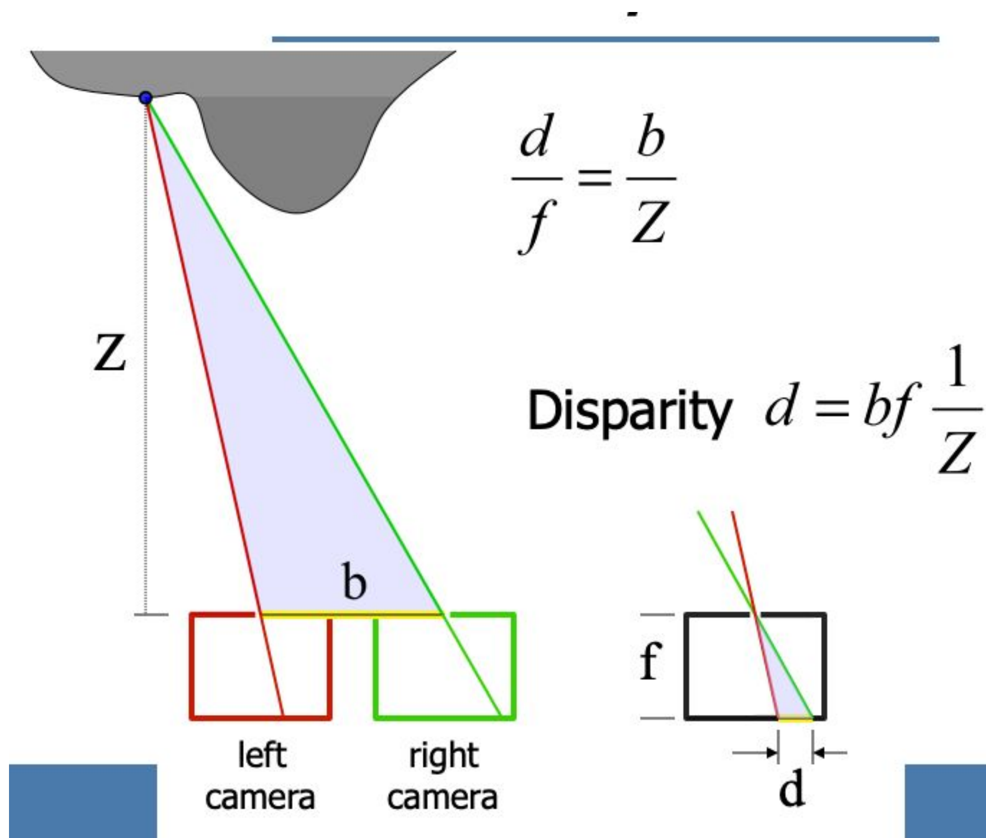
- Encoder pretrained on ImageNet1k, train decoder for depth prediction, supervised by ground truth depth
- Same as before, implement in model.py, training.py, losses.py, and report the results in your submission.

Problem 4 - Unsupervised Monocular Depth Estimation

- Both images from a stereo pair during training
- Model takes in left image, predicts left and right disparity
- Use predicted disparity to synthesize left and right images
- Also synthesize disparity using prediction to encourage cycle consistency
- Goal is to reconstruct right image



Disparity Recap



$$\frac{d}{f} = \frac{b}{Z}$$

Disparity $d = bf\frac{1}{Z}$

©Michael J. Black

Synthesize Images and Disparity

Then, assuming that the input images are rectified, we can generate left and right images from the predicted disparities. To be more concrete, using the left disparity $disp_l$, we synthesize left image and disparity as following: $img'_l = \text{generate_image_left}(img_r, disp_l)$ and $disp'_l = \text{generate_image_left}(disp_r, disp_l)$. Similarly, using the right disparity $disp_r$, we synthesize right image and disparity as following: $img'_r = \text{generate_image_right}(img_l, disp_r)$ and $disp'_r = \text{generate_image_right}(disp_l, disp_r)$. We will ask you to implement generate_img_left and generate_image_right in this pset.

- Image Reconstruction loss
  - L1 + SSIM between synthesized and ground truth
- Disparity Loss
  - Cycle consistency between synthesized disparity maps
  - Smoothness term as regularization

Data Augmentation

- Perform reasonable transformations on the data to generate more data for training
  - Flipping the input gives us 2x the amount of data
  - In some cases, flipping doesn’t make sense (training to classify letters, AO, but E?)
- torchvision.transforms.RandomHorizontalFlip
  - p=1: either flip both of them, or don’t flip.
  - Apply self.transform to left_image and right_image, and return appropriately
  - We have a stereo pair, are there any details to take care of before returning?

Bilinear Sampling

- Shift input image horizontally according to disparity
- Use torch.linspace and torch.meshgrid to generate a grid, each location storing the location of the pixel we want (between 0-1)
- Shift the meshgrid using disparity in the x-dimension
- Scale the range of the meshgrid to [-1, 1]
- Perform bilinear sampling (built-in interpolation method in nn.functional.grid_sample)
    - Generate output image by sampling the corresponding pixel from input image based on the mesh grid

Image Generator

- Given predicted disparity, synthesize left and right images using the bilinear_sampler from part b
- Note that the disparity values from the model are between [0, 1], whereas our bilinear_sampler implementation expects [-1, 1]
    - Need to flip the sign in one of them to get the correct result