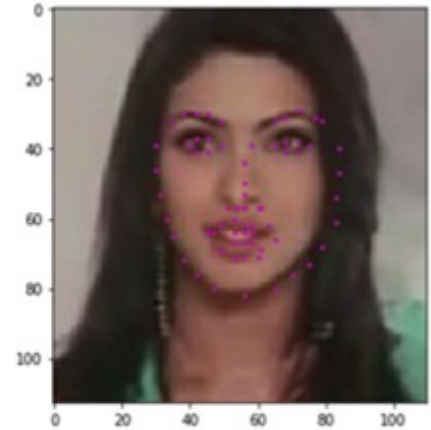


Introduction to Deep Learning (I2DL)

Tutorial 9: Facial Keypoint Detection

Overview

- Exercise 08: Case Study
- Fully Connected & Convolutional Layers
 - Recap
 - Changes to Dropout & BatchNorm
- Exercise 09: Facial Keypoint Detection



Exercise 8: Leaderboard

Exercise 1 Exercise 3 Exercise 4 Exercise 5 Exercise 6 Exercise 7 **Exercise 8** Exercise 9 Exercise 10 Exercise 11

#	User	Score	
1	u0741	100.00	CNN
2	u1289	100.00	MLP
3	u0736	99.00	
4	a0001	99.00	
5	u1770	96.00	
6	u1479	95.00	
7	u0922	94.00	
8	u0926	91.00	
9	u1662	90.00	
10	u1149	89.00	
11	u1625	88.00	
12	u0533	88.00	

Exercise 8: Case Study - Architecture

```
self.encoder = nn.Sequential(
    nn.Linear(input_size, num_hidden), # 784 -> 392
    nn.BatchNorm1d(num_hidden),
    nn.ReLU(),
    nn.Linear(num_hidden, int(num_hidden*0.5)),
    nn.BatchNorm1d(int(num_hidden*0.5)),
    nn.ReLU(),
    nn.Linear(int(num_hidden*0.5), int(num_hidden*0.25)),
    nn.BatchNorm1d(int(num_hidden*0.25)),
    nn.ReLU(),
    nn.Linear(int(num_hidden*0.25), int(num_hidden*0.125)),
    nn.BatchNorm1d(int(num_hidden*0.125)),
    nn.ReLU(),
    nn.Linear(int(num_hidden*0.125), latent_dim))
```

```
self.classifier = nn.Sequential(
    nn.Linear(latent_dim, num_hidden_c),
    nn.BatchNorm1d(num_hidden_c),
    nn.LeakyReLU(),
    nn.Dropout(p=0.2),
    nn.Linear(num_hidden_c, num_hidden_c),
    nn.BatchNorm1d(num_hidden_c),
    nn.LeakyReLU(),
    nn.Dropout(p=0.2),
    nn.Linear(num_hidden_c, num_classes))
```

```
self.decoder = nn.Sequential(
    nn.Linear(latent_dim, int(num_hidden*0.125)),
    nn.BatchNorm1d(int(num_hidden*0.125)),
    nn.ReLU(),
    nn.Linear(int(num_hidden*0.125), int(num_hidden*0.25)),
    nn.BatchNorm1d(int(num_hidden*0.25)),
    nn.ReLU(),
    nn.Linear(int(num_hidden*0.25), int(num_hidden*0.5)),
    nn.BatchNorm1d(int(num_hidden*0.5)),
    nn.ReLU(),
    nn.Linear(int(num_hidden*0.5), num_hidden),
    nn.BatchNorm1d(num_hidden),
    nn.ReLU(),
    nn.Linear(num_hidden, input_size))
```



AE

Parameters: Your model has 0.824 mio. params



CLS

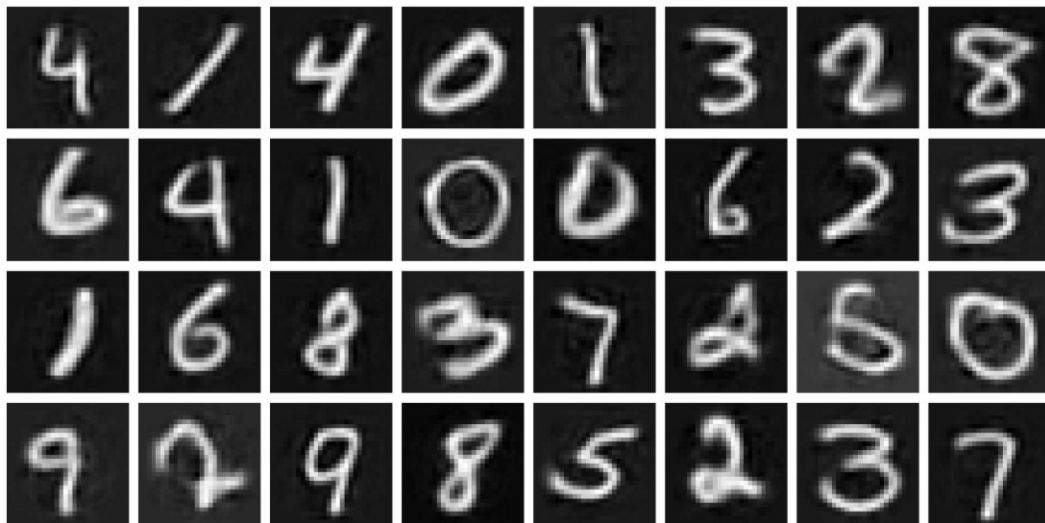
Parameters: Your model has 0.591 mio. params

Exercise 8: Case Study – Hyper-Parameters

```
transform = T.Compose([
    T.RandomApply([T.RandomRotation(degrees=30)], p=0.2),
    T.RandomApply([T.GaussianBlur(kernel_size=3, sigma=(0.1, 1.5))], p=0.2),
    T.RandomApply([T.RandomAffine(degrees=0, translate=(0.08, 0.08))], p=0.2),
])
```

```
hparams = {
    "n_hidden": 392,
    "latent_dim": 32,
    "n_hidden_C": 400,
    "learning_rate": 5e-4,
    "weight_decay": 1e-4,
    "epochs_ae": 5,
    "epochs_classifier": 50
}
```

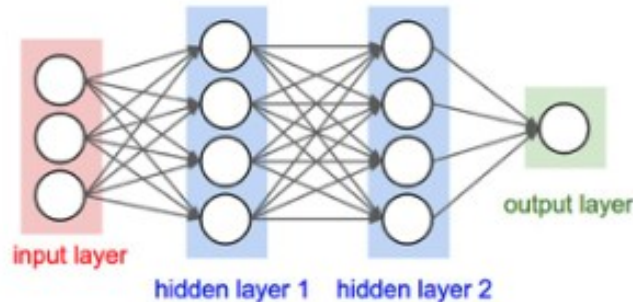
Auto-Encoder Reconstructions



Fully Connected vs Convolutional Layers

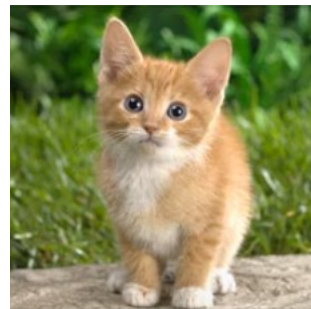
Recap: Fully Connected Layers

- Fully Connected (FC) networks / Multi-Layer Perceptron (MLP): Receive an input vector and transform it through a series of hidden layers (weights & activation functions).
- **Fully Connected layers:** Each layer is made up of a set of neurons, where each single neuron is connected to all neurons in the previous layer



Computer Vision – MLP

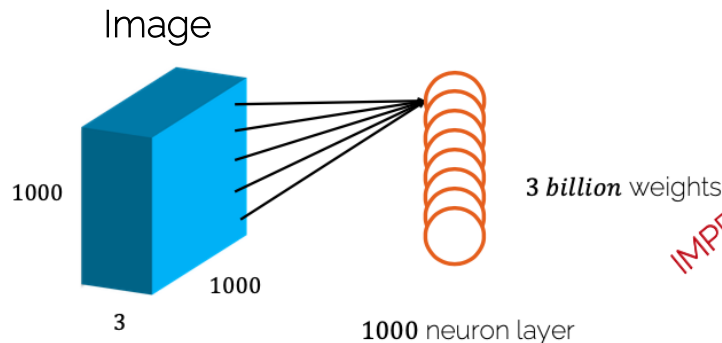
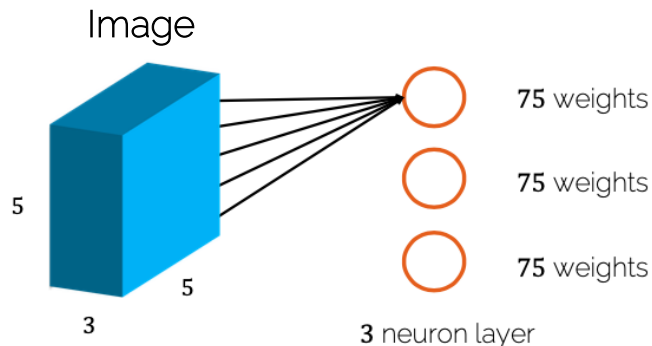
- **Assumption:** Input to the network are images
- **Disadvantage:** Images need to have a certain resolution to contain enough information



238x238



5x5

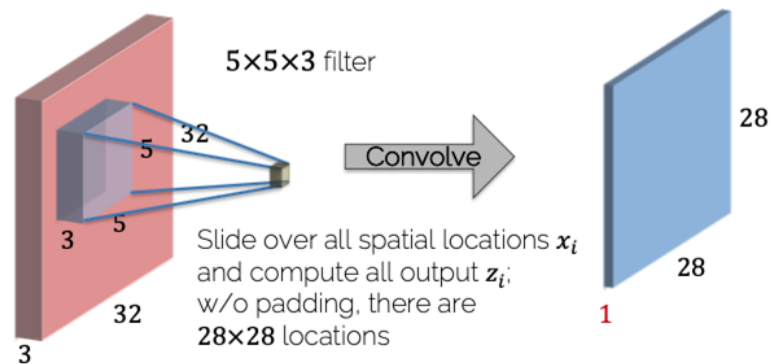


IMPRACTICAL

Can we reduce the number of weights in our architecture?

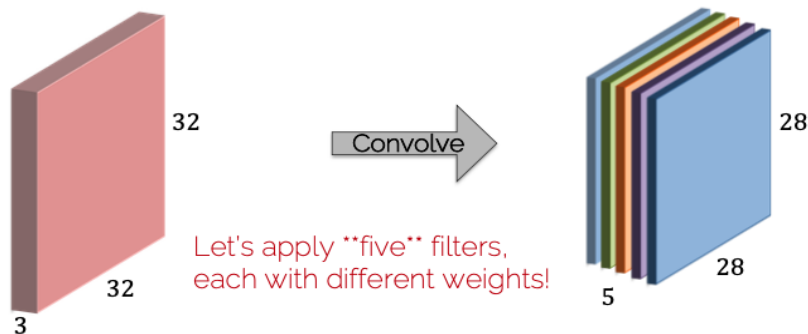
Computer Vision - CNN

- **Assumption:** Input to the network are images
- **Idea:** Sliding filter over the input image (convolution) instead of passing the entire image through all neurons individually



Computer Vision - CNN

- **Assumption:** Input to the network are images
- **Filters:** Sliding window with the same filter parameters to extract image features
- **Advantage:** Learn translation-invariant “concepts” and weight sharing



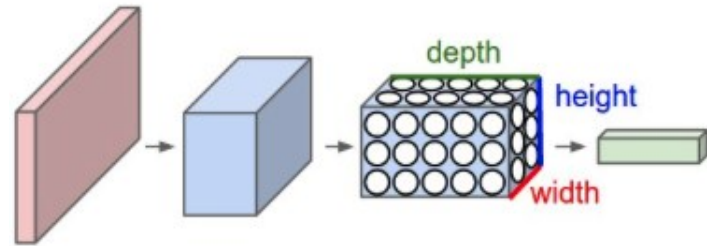
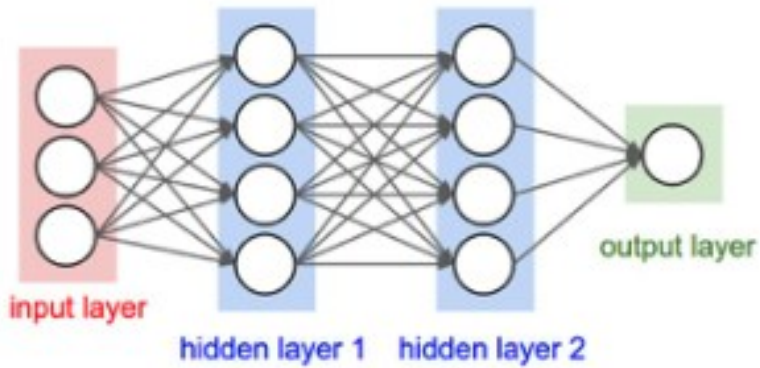
Convolution: Hard-coded



Convolutional Layers: BatchNorm and Dropout

Fully Connected vs Convolution

- Output Fully-Connected layer: One layer of neurons, independent
- Output Convolutional Layer: Neurons arranged in 3 dimensions



Recap: Batch Normalization

- Batch norm for **FC** neural networks
 - Input size (N, D)
 - Compute minibatch mean and variance across N (i.e. we compute mean/var for each feature dimension)

Input: $x : N \times D$

Learnable params:

$$\gamma, \beta : D$$

Intermediates: $\mu, \sigma : D$
 $\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Recap: Batch Normalization

- Batch norm for **FC** neural networks
 - Input size (N, D)
 - Compute minibatch mean and variance across N (i.e. we compute mean/var for each feature dimension)

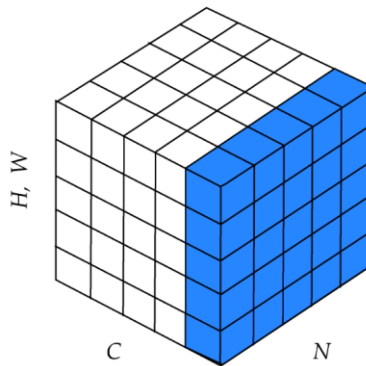
Batch Normalization for
fully-connected networks

$$\begin{array}{l} \mathbf{x}: \mathbf{N} \times \mathbf{D} \\ \text{Normalize} \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{1} \times \mathbf{D} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: \mathbf{1} \times \mathbf{D} \\ \mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

Spatial Batch Normalization

- Batchnorm for **convolutional** NN = spatial batchnorm
 - Input size (N, C W, H)
 - Compute minibatch mean and variance across N, W, H (i.e. we compute mean/var for each channel C)

$$\begin{aligned} \mathbf{x}: & \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} & \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: & \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: & \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = & \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{aligned}$$



Spatial Batch Normalization

Fully Connected

- Input size (N, D)
- Compute minibatch mean and variance **across** N (i.e. we compute mean/var for each feature dimension)

$$\begin{array}{l} \mathbf{x}: \mathbf{N} \times \mathbf{D} \\ \text{Normalize} \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{1} \times \mathbf{D} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: \mathbf{1} \times \mathbf{D} \\ \mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

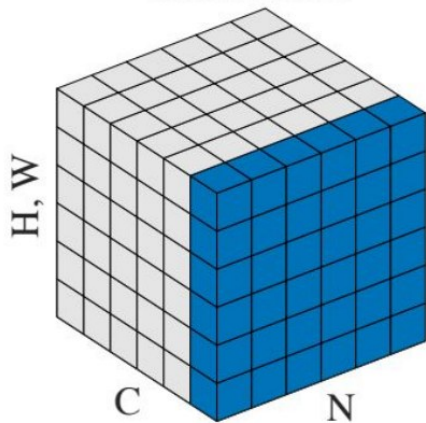
Convolutional = spatial BN

- Input size (N, C, W, H)
- Compute minibatch mean and variance **across** N, W, H (i.e. we compute mean/var for each channel C)

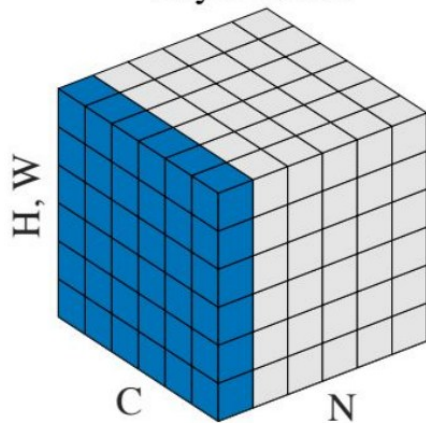
$$\begin{array}{l} \mathbf{x}: \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

Other normalizations

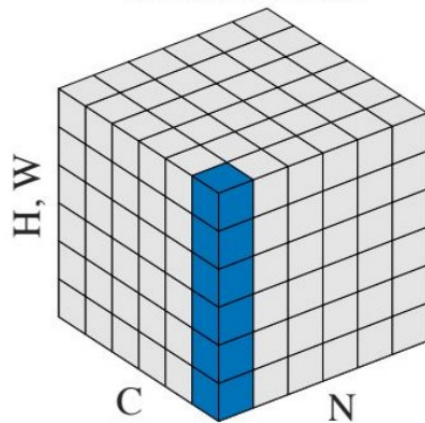
Batch Norm



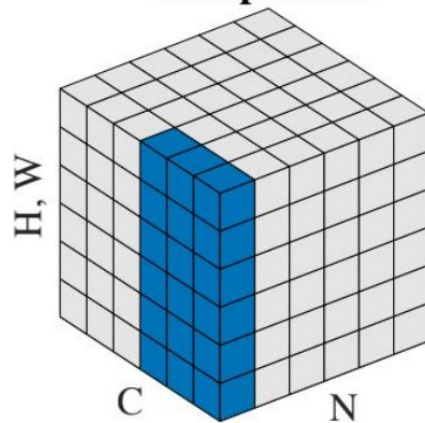
Layer Norm



Instance Norm



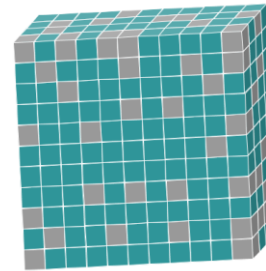
Group Norm



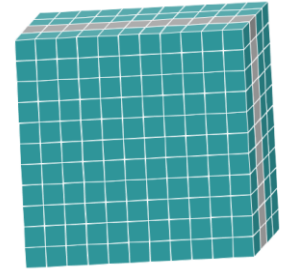
Dropout for convolutional layers

- **Regular Dropout:** Deactivating specific neurons in the networks (one neuron “looks” at whole image)
- **Dropout Convolutional Layers:** Standard neuron-level dropout (i.e. randomly dropping a unit with a certain probability) does not improve performance in convolutional NN
- **Spatial Dropout** randomly sets entire feature maps to zero

Standard Dropout



Spatial Dropout



Dropout for convolutional layers

```
def dropout_mlp():  
    m = nn.Dropout(p=0.5)  
    batch_size = 1  
    inputs = torch.randn(batch_size, 3 * 5 * 5)  
    outputs = m(inputs)  
  
    print(outputs)  
  
    tensor([[  
      -0.89,  0.37, -0.00,  0.00, -0.08, -0.00,  
       0.00, -3.55,  0.00,  0.47, -0.00,  5.08,  
      -0.00, -0.00,  2.63,  0.00,  0.00,  0.00,  
       2.18,  1.92, -0.00,  0.66,  1.96,  0.00,  
      -0.00, -0.00,  0.00,  1.31, -1.95, -0.00,  
       0.00, -4.44,  0.00, -1.07, -0.90, -0.07,  
      -3.81,  0.00,  0.23,  2.38, -2.27, -0.51,  
      -3.32, -0.00, -0.65,  0.00, -0.00, -0.00,  
      -0.00, -0.00, -0.61,  0.00,  0.00,  0.00,  
      -1.85, -0.40,  0.00,  0.68, -0.00, -1.96,  
      -0.00, -1.65,  0.00, -0.66,  3.10,  0.00,  
      -0.00,  1.89,  0.00, -1.28, 1.62, -0.56,  
      -0.00, -0.00, -0.99]])
```

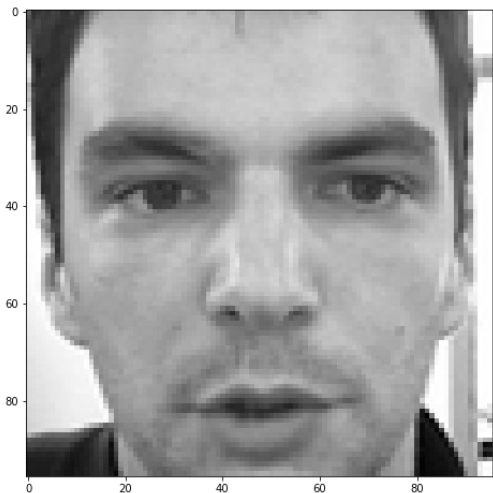
```
def dropout_cnn():  
    m = nn.Dropout2d(p=0.5)  
    batch_size = 1  
    inputs = torch.randn(batch_size, 3, 5 * 5)  
    outputs = m(inputs)  
  
    print(outputs)  
  
    tensor([[  
      [ 0.03,  1.40,  1.76, -4.34, -0.63,  
        -0.31,  2.80,  2.72, -3.00,  2.67,  
        -2.31, -3.45,  0.95,  1.18,  1.18,  
        -1.05,  0.74,  3.56,  0.55, -1.19,  
        -0.28,  0.89, -3.36, -2.00, -0.29],  
      [ 0.00, -0.00, -0.00, -0.00, -0.00,  
        0.00, -0.00, -0.00, -0.00,  0.00,  
        -0.00,  0.00,  0.00, -0.00, -0.00,  
        0.00, -0.00,  0.00,  0.00, -0.00,  
        -0.00,  0.00, -0.00,  0.00,  0.00],  
      [ 0.00, -0.00, -0.00, -0.00,  0.00,  
        0.00,  0.00,  0.00, -0.00, -0.00,  
        -0.00, -0.00,  0.00, -0.00, -0.00,  
        0.00,  0.00,  0.00, -0.00,  0.00,  
        -0.00, -0.00,  0.00,  0.00, -0.00]])
```

Exercise 9: Facial Keypoints Detection

Submission: Facial Keypoints

Input:

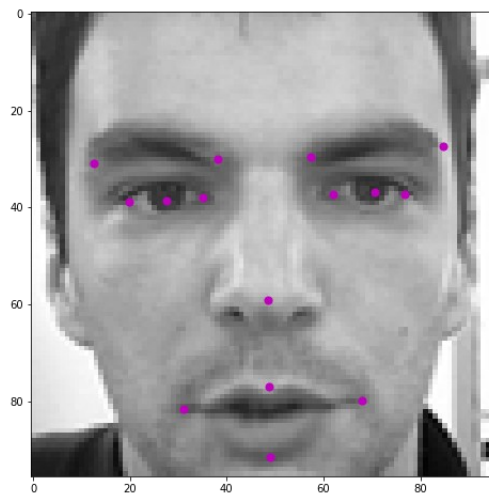
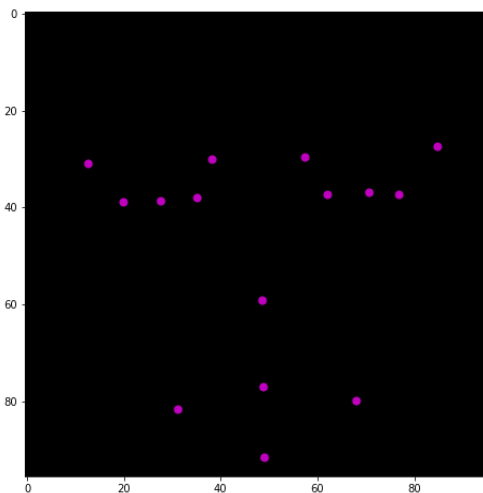
(1, 96, 96) grayscale image



CNN

Output:

(2, 15) keypoint coordinates



Dataset:

- train: 1546 images
- validation: 298 images

Submission: Metric

Accuracy (Classification) → Score (Regression)

```
def evaluate_model(model, dataset):  
    model.eval()  
    criterion = torch.nn.MSELoss()  
    dataloader = DataLoader(dataset, batch_size=1, shuffle=False)  
    loss = 0  
    for batch in dataloader:  
        image, keypoints = batch["image"], batch["keypoints"]  
        predicted_keypoints = model(image).view(-1,15,2)  
        loss += criterion(  
            torch.squeeze(keypoints),  
            torch.squeeze(predicted_keypoints)  
        ).item()  
    return 1.0 / (2 * (loss/len(dataloader)))  
  
print("Score:", evaluate_model(dummy_model, val_dataset))
```

Submission Requirement: Score ≥ 100

Good luck &
see you next week

