Introduction to Deep Learning (I2DL)

Tutorial 9: Facial Keypoint Detection

Overview

- **Recap: Exercise 8 & Convolutional Layers**
  - exercise 8 case study — what worked on the leaderboard
  - fully connected vs convolutional layers
  - spatial batch normalization and dropout for convolutions
- **Exercise 9: Facial Keypoint Detection**
  - build and train your own convolutional network

Exercise 8: Leaderboard

Leaderboard

The leaderboard shows for each exercise the highest scoring submission from each user. Only valid submissions are displayed.

Exercise 1 | Exercise 3 | Exercise 4 | Exercise 5 | Exercise 6 | Exercise 7 | Exercise 8 | Exercise 9 | Exercise 10 | Exercise 11 | Exercise 12

#	User	Score
1	u1314	100.00
2	u1449	98.00
3	u1518	95.00
4	u0084	92.00
5	u0271	92.00
6	u1959	91.00
7	u0994	90.00
8	u1288	90.00
9	u0151	89.00
10	u0433	88.00

How the #1 Submission Scored 100

The Model — a Denoising MLP Autoencoder

- **Block** = Linear → LayerNorm → GELU → Dropout
- **Encoder:** 3 blocks, 784 → 1024 → 512 → **128** · Dropout **0.1** (last block omits it)
- **Decoder:** mirrors it, 128 → 512 → 1024 → 784 → Sigmoid (no dropout)
- **Classifier head:** 128 → **block** (Dropout 0.2) → 256 → Linear → 10
- **Model size:** ~1.4M parameters (under the 5M limit)

Step 1: Pretrain a Denoising Autoencoder

- **Data:** the **~59k unlabeled** images.
- **Make the task harder:** reconstruct the **clean** image from a **corrupted** one, not from itself.
- **Corrupt the input, in this order:** clean image → **mask 20% of pixels** → **then add Gaussian noise σ=0.15** → feed to the network.
- **Loss** BCE · **Adam** lr 8e-4 · weight decay 1e-4 · **cosine** scheduler.
- **batch** 512 · **15 epochs** · keep the **best-validation** encoder.

Step 2: Transfer — the PyTorch Pieces

```
# Discriminative LRs: low on the pretrained encoder, higher on the head
optimizer = torch.optim.AdamW([
    {"params": encoder.parameters(), "lr": 1.5e-4},
    {"params": head.parameters(),    "lr": 7.5e-4},
], weight_decay=1e-4)

# Class-balanced sampling (1000 samples / epoch, fine-tune batch = 32)
weights = torch.bincount(labels, minlength=10).float().reciprocal()[labels]
sampler = WeightedRandomSampler(weights, num_samples=1000, replacement=True)
loader  = DataLoader(train_set, batch_size=32, sampler=sampler)

# Label smoothing + Mixup (alpha = 0.2)
criterion = nn.CrossEntropyLoss(label_smoothing=0.05)
lam   = np.random.beta(0.2, 0.2)
perm  = torch.randperm(len(images))
mixed = lam * images + (1 - lam) * images[perm]
loss  = lam * criterion(model(mixed), labels) \
      + (1 - lam) * criterion(model(mixed), labels[perm])

# Strong augmentation (applied to the labeled images)
strong = T.Compose([
    T.RandomAffine(degrees=12, translate=(0.10, 0.10), scale=(0.90, 1.10), shear=5),
    T.RandomErasing(p=0.15, scale=(0.01, 0.06)),
])
# -> fine-tune for 50 epochs (grad-clip max-norm 5)
```

Step 3: Mean Teacher — the Deciding +1%

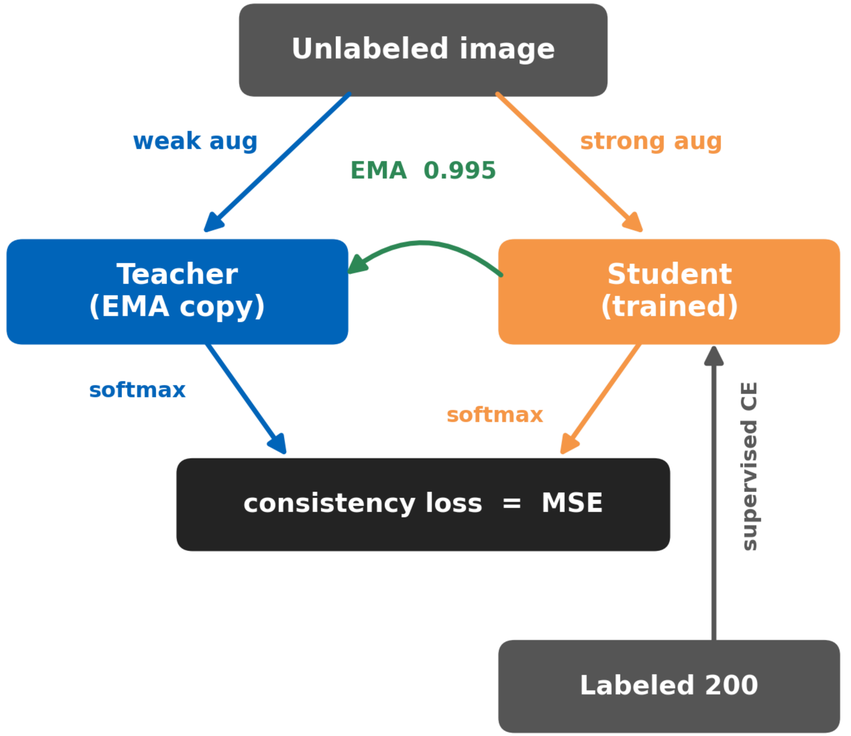
- **Student** trained by backprop · **teacher** = frozen **EMA** of the student (0.995).

$$\mathcal{L} = \mathcal{L}_{sup} + w(t) \cdot \mathcal{L}_{cons}$$

$$\mathcal{L}_{cons} = \mathrm{MSE}\big(\sigma(\mathrm{student}_{strong}),\, \sigma(\mathrm{teacher}_{weak})\big)$$

$$w(t) = 10\, \exp\big(-5\,(1 - \min(1,\, t/10))^2\big)$$

- **weak aug:** RandomAffine(5°, 4% shift, 0.97–1.03 scale)
- **strong aug:** RandomAffine(12°, 10% shift, 0.9–1.1, shear 5) + RandomErasing(0.15)
- **AdamW 5e-5** · **30 epochs** · submit the **teacher**



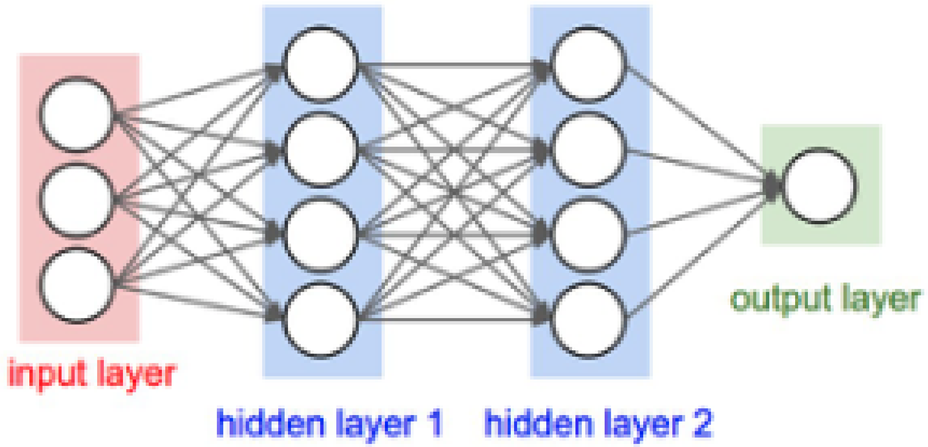
Further Reading — Techniques Used

- A few of these go beyond the exercise, handy if you want to push further:
- **Denoising autoencoder** — paper · blog
- **Label smoothing** — paper · blog
- **Mixup** — paper · blog
- **Mean Teacher** (semi-supervised) — paper · blog

Fully Connected vs Convolutional Layers

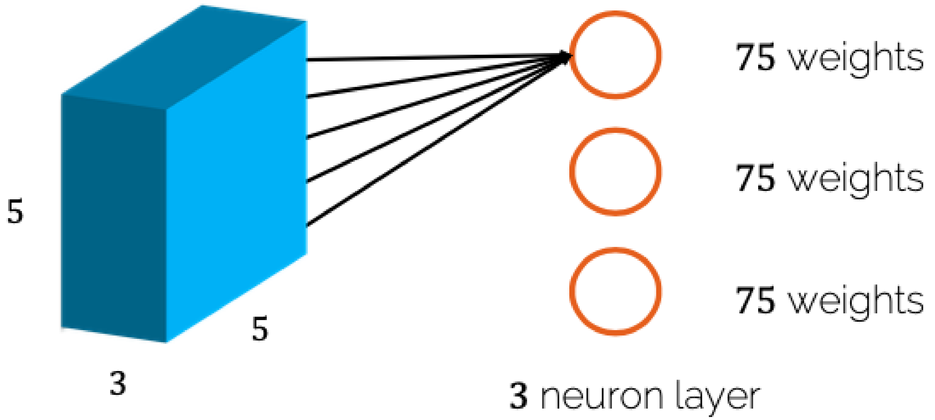
Recap: Fully Connected Layers

- A fully connected network, or **multilayer perceptron**, sends an input vector through a series of **hidden layers**.
- **Each neuron** connects to **every neuron** in the previous layer.
- Every connection is its own **weight** to learn.



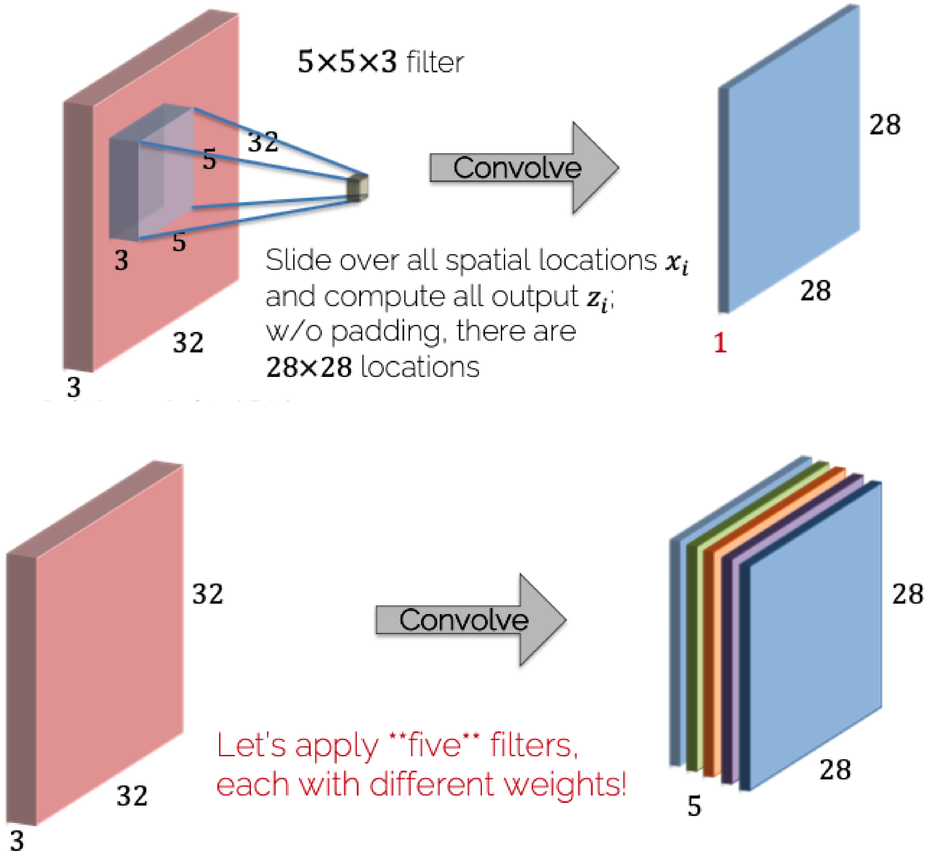
Computer Vision: Why Not Just an MLP?

- Now the input is an **image**.
- To carry enough information an image needs a **decent resolution**, so the input is very high-dimensional.
- Fully connecting a full-size image to a hidden layer needs **billions of weights**, impractical to train.
- **Can we reduce the number of weights?**



The Convolution Idea

- Instead of connecting every pixel, **slide a small filter** over the image.
- Take a 32×32 image with 3 channels and a **5×5×3 filter**, focused on one small region at a time.
- Slide it across the image to build a **feature map**, here 28×28.
- The **same filter weights** are reused everywhere — **weight sharing** — giving **translation-invariant** features.
- **Different filters** learn different features, each producing its own feature map.



Convolution by Example: Hand-Coded Filters

3x3 kernel

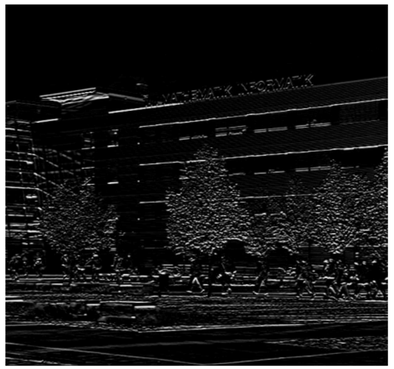
-1	-1	-1
0	0	0
1	1	1

*

*

3x3 kernel

-1	0	1
-1	0	1
-1	0	1

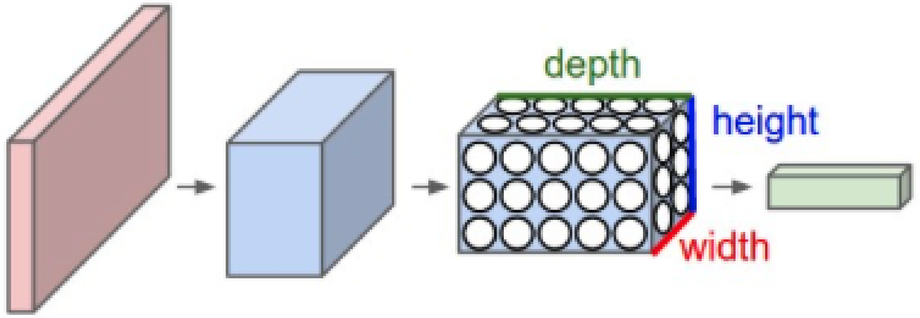


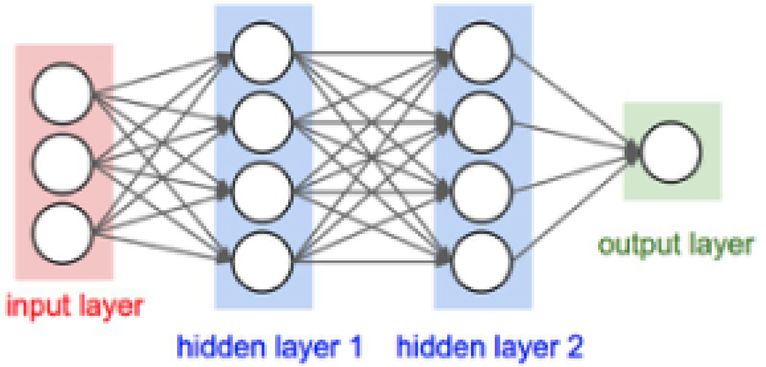
horizontal edges

vertical edges

Fully Connected vs Convolution

- Side by side, the difference is how the **output neurons** are arranged.
- **Fully connected:** one flat layer of neurons, each independent.
- **Convolution:** neurons arranged in **three dimensions** — width, height, and depth.
- Convolution keeps the image's **spatial structure**, with far fewer weights.

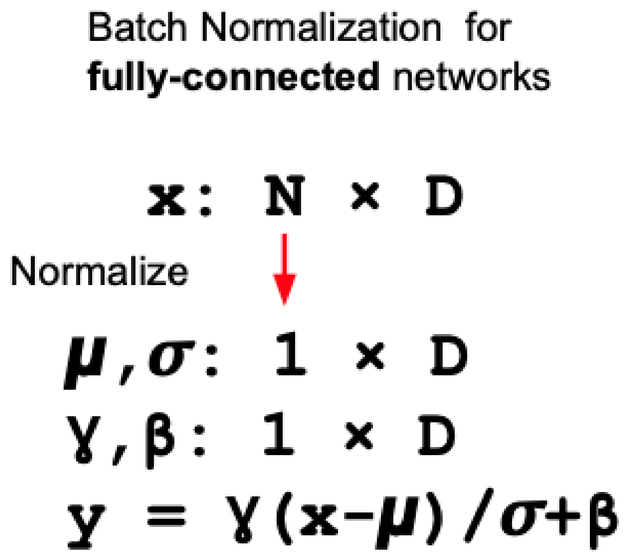




Batch Normalization & Dropout for Convolutions

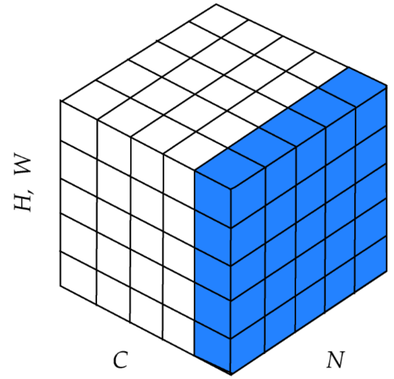
Recap: Batch Normalization

- For a fully connected network the input is shape **(N, D)** — N samples, D features.
- Compute the **mean and variance across the batch**, separately for **each feature dimension**, then normalize.



Spatial Batch Normalization

- For a **convolutional** network we use **spatial batch normalization**.
- The input is now **(N, C, H, W)** — N samples, **C channels**, height H, width W.
- Compute the **mean and variance over N, H and W** — the batch and both spatial dimensions — **for each channel C**.
- Same idea as before — just normalized **per channel (feature map)** instead of per feature.



$$x: \ N \times C \times H \times W$$

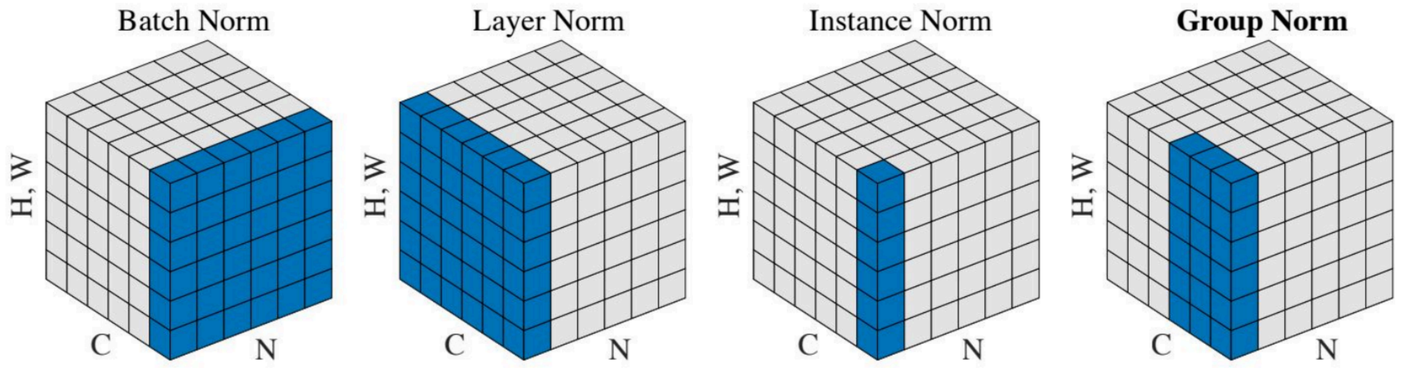
Normalize

$$\mu, \sigma: \ 1 \times C \times 1 \times 1$$

$$\gamma, \beta: \ 1 \times C \times 1 \times 1$$

$$y = \gamma(x-\mu)/\sigma+\beta$$

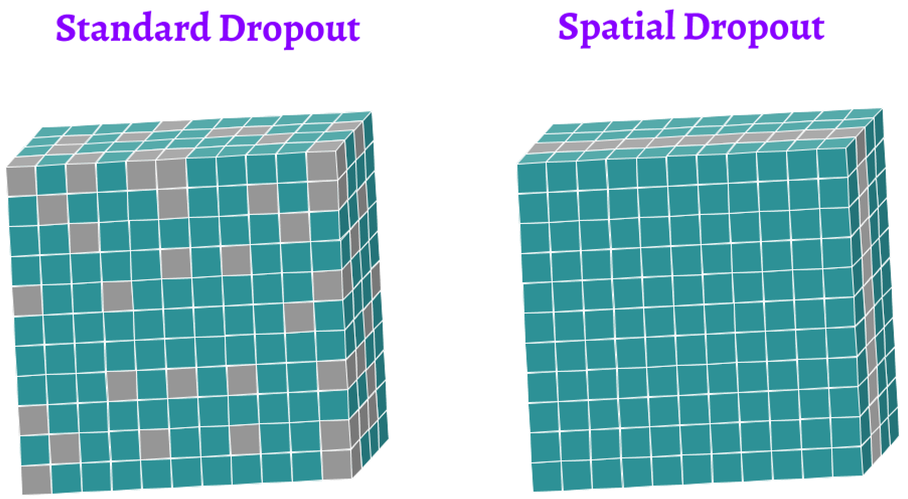
Other Normalizations



- Batch norm is not the only choice.
- **Layer, Instance, and Group Norm** differ in **which dimensions they normalize across**.
- Each one reduces the network's dependence on the **mini-batch**.

Dropout for Convolutional Layers

- **Dropout** also needs a twist for convolutions.
- Regular dropout zeros **individual neurons**, but in a conv layer that **barely helps**.
- **Spatial dropout** instead zeros **entire feature maps** at random (PyTorch's *Dropout2d*).
- Dropping a whole feature map makes more sense than scattering zeros across the image.



Dropout vs Dropout2D: An Example

```
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]])
```

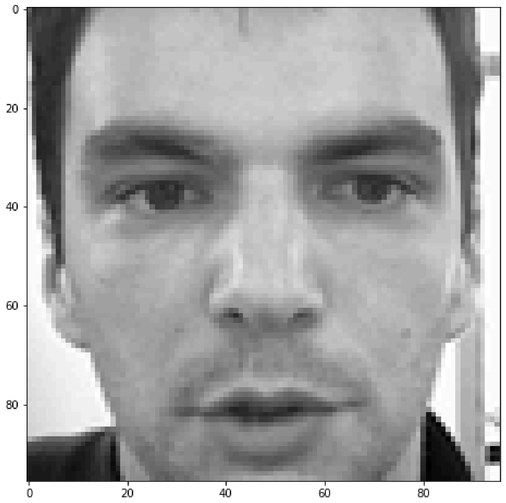
```
def dropout_cnn():
    m = nn.Dropout2d(p=0.5)
    batch_size = 1
    inputs = torch.randn(batch_size, 3, 5 * 5)
    outputs = m(inputs)

    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]]])
```

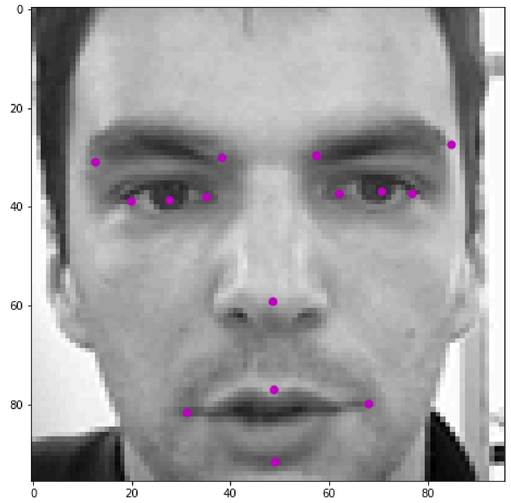
Exercise 9: Facial Keypoint Detection

The Task



Input: (1, 96, 96)

CNN



Output: 15 keypoints (x, y)

Dataset: 1546 training images, 298 for validation, fully labeled.

The Metric

- Keypoint detection is a **regression** problem, so the loss is **mean squared error**.

$$\text{score} = \frac{1}{2\,\overline{L}}$$

- The score comes from the **average loss** $\overline{L}$ over the dataset.
- You need **score ≥ 100** to pass — an average loss below **0.005**.

```python
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))
```

Good Luck & See You Next Week!