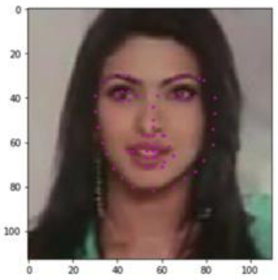
Introduction to Deep Learning (I2DL)

Tutorial 9: Facial Keypoint Detection

Overview

- Optional Exercise: CIFAR-10
  - Case study of two submitted solutions
- Convolutional Layers
  - Recap
  - Changes to Dropout & Batchnorm
- Submission: Facial Keypoint Detection
  - Deadline: 05.01.2022 15.59



Case Study:
Optional Exercise CIFAR-10

Optional Exercise: Summary

- Image classifier on CIFAR-10 dataset
- CIFAR-10: Ten classes ('plane', 'car', 'bird', 'cat', …)
- Pytorch Lightning
- Passing Criteria: 50%
- Restrictions in this exercise:

** The size of your final model must be less than 20 MB, which is approximately equivalent to 5 Mio. params. Note that this limit is quite lenient, you will probably need much less parameters!*

Also, don't use convolutional layers as they've not been covered yet in the lecture and build your network with fully connected layers (`nn.Linear()`)!

Case Study: Leaderboard

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

#	User	Score
1	u1051	86.30
2	u1048	80.80
3	u1120	64.89
4	u1552	61.66
5	u0924	59.94
6	u0449	59.86
7	u1458	57.94

Submission Leaderboard Optional Exercise: CIFAR10 (13.12.2021)

Case Study #1: Model

```
##############################################################################
# TODO: Initialize your model!                                              #
##############################################################################
self.model = nn.Sequential(
    nn.Conv2d(3, 6, 3, padding=1),
    torch.nn.BatchNorm2d(num_features=6),
    nn.ReLU(),
    nn.Conv2d(6, 9, 3, padding=1),
    torch.nn.BatchNorm2d(num_features=9),
    nn.ReLU(),
    nn.Conv2d(9, 9, 5, padding=2),
    torch.nn.BatchNorm2d(num_features=9),
    nn.ReLU(),
    nn.Conv2d(9, 9, 5, padding=2),
    torch.nn.BatchNorm2d(num_features=9),
    nn.ReLU(),
    nn.Conv2d(9, 18, 5, stride=2, padding=2),
    torch.nn.BatchNorm2d(num_features=18),
    nn.ReLU(),
    nn.Conv2d(18, 36, 5, stride=2, padding=2),
    torch.nn.BatchNorm2d(num_features=36),
    nn.ReLU(),
    nn.Conv2d(36, 72, 3, stride=1, padding=1),
    torch.nn.BatchNorm2d(num_features=72),
    nn.ReLU(),
    torch.nn.AvgPool2d((8,8)),
    Lambda(lambda x: torch.squeeze(x)),
    torch.nn.Linear(72, num_classes),
    torch.nn.Softmax(dim=1)
)
```

Also, don't use convolutional layers as they've not been covered yet in the lecture and build your network with fully connected layers (`nn.Linear()`)!

Case Study #2: Model

```
########################################################################
# TODO: Initialize your model!                                         #
########################################################################

modules = []
for _ in range(hparams['num_layers']-2):
    modules.append(nn.Linear(self.hparams['n_hidden'], self.hparams['n_hidden']))
    modules.append(nn.ReLU())

self.model = nn.Sequential(
    nn.Linear(input_size, self.hparams['n_hidden']),
    nn.ReLU(),
    *modules,
    nn.Linear(self.hparams['n_hidden'], num_classes),
)

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

Model Hyperparameters:

- Number of nodes in hidden layers
- Number of layers

Very Simple Network Architecture

- No BatchNorm Layers
- No Dropout Layers
- Finally: 3 Blocks (Linear (+ ReLu))

Default initialization for Linear Layers

Pytorch Default Weight Initialization

CLASS `torch.nn.Linear(in_features, out_features, bias=True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Xavier/2 Init
in comparison

$$Var(w_i) = \frac{2}{fan\_in}$$

Case Study #2: Model

```
############################################################################
# TODO: Initialize your model!                                             #
############################################################################

modules = []
for _ in range(hparams['num_layers']-2):
    modules.append(nn.Linear(self.hparams['n_hidden'], self.hparams['n_hidden']))
    modules.append(nn.ReLU())

self.model = nn.Sequential(
    nn.Linear(input_size, self.hparams['n_hidden']),
    nn.ReLU(),
    *modules,
    nn.Linear(self.hparams['n_hidden'], num_classes),
)

############################################################################
#                              END OF YOUR CODE                            #
############################################################################
```

Model Hyperparameters:

- Number of nodes in hidden layers
- Number of layers

Default initialization for Linear Layers

- ReLu kills half of the data, so Xavier/2 initiliazation could be beneficial (see Lecture 7)
- Pytorch: torch.nn.init.xavier_normal_ (takes in_f and out_f into consideration)

Case Study #2: Transforms

```
my_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std),
    transforms.RandomHorizontalFlip(p=0.5),
    RandomTranslation(prob=0.5),
    transforms.RandomChoice([
        RandomSpeckle(std=0.2, prob=0.5),
        SaltandPepper(prob=0.5),
        #transforms.GaussianBlur(kernel_size=5, sigma=(0.1, 2.0)),
        #transforms.RandomRotation((90, 90), resample=False, expand=False, center=None, fill=None),
        #transforms.RandomRotation((-90, -90), resample=False, expand=False, center=None, fill=None),
        #transforms.RandomRotation((180, 180), resample=False, expand=False, center=None, fill=None),
        #transforms.RandomErasing(p=0.2, scale=(0.02, 0.2), ratio=(0.3, 3.3), value=0.05, inplace=False)
    ])
])
```

Case Study #2: Hyperparameter Tuning

Random Search

```python
from exercise_code.MyPytorchModel import MyPytorchModel
from exercise_code.Util import printModelInfo
from exercise_code.Util import test_and_save
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
import random
from math import log10

for i in range(100):
    early_stop_callback = EarlyStopping(
        monitor='val_loss',
        min_delta=0.0,
        patience=7,
        verbose=False,
        mode='min'
    )

    hparams = {}
    sample = random.uniform(log10(2.5e-4), log10(9e-4))#random.uniform(
    lr = 10**(sample)
    num_layers = random.choice([2, 3, 4, 5])
    if num_layers == 2:
        hidden_end = 1600
    if num_layers == 3:
        hidden_end = 1170
    if num_layers == 4:
        hidden_end = 980
    if num_layers == 5:
        hidden_end = 870
    hparams = {
        'lr': lr,
        'decay': 0.0,#random.uniform(0.0, 0.3),
        'num_layers': 3,#num_layers,
        'n_hidden': 725,#random.choice([703, 725]),#random.randint(700,
        'batch_size': 2048,#random.choice([512, 1024, 2048]),#random.ch
        'num_workers': 3
    }
    print(hparams)
```

```python
model = MyPytorchModel(hparams)
model.prepare_data()
_ = printModelInfo(model)

if hparams['batch_size'] == 512:
    epochs = 57
if hparams['batch_size'] == 1024:
    epochs = 50
if hparams['batch_size'] == 2048:
    epochs = 65

trainer = None
trainer = pl.Trainer(
            #callbacks=[early_stop_callback],
            precision=16,
            weights_summary=None,
            max_epochs=65,#epochs,
            #profiler='simple',
            progress_bar_refresh_rate=10,
            gpus=1
)
trainer.fit(model)
test_and_save(model)
```

Case Study #2: Final Hyperparameters

```
hparams = {}
sample = random.uniform(log10(2.5e-4), log10(9e-4))#random.uniform(log10(5e-6), log10(6e-3))
lr = 10**(sample)
```

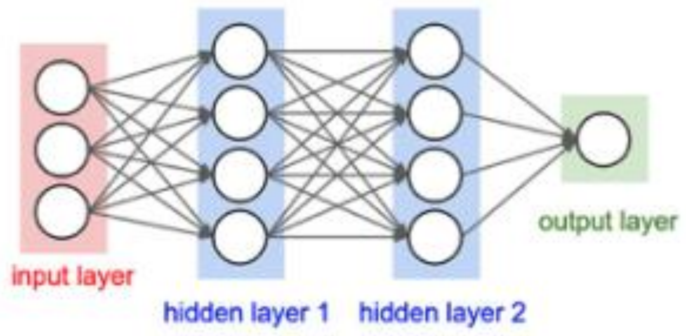
```
hparams = {
    'lr': lr,
    'decay': 0.0,#random.uniform(0.0, 0.3),
    'num_layers': 3,#num_layers,
    'n_hidden': 725,#random.choice([703, 725]),#random.randint(700,800),#random.randint(100, hidden_end),
    'batch_size': 2048,#random.choice([512, 1024, 2048]),#random.choice([32, 64, 128, 256, 512, 1024, 2048]),
    'num_workers': 3
}
```

Take away: Always start with simple networks, you can already achieve quite good results

Convolutional Layers

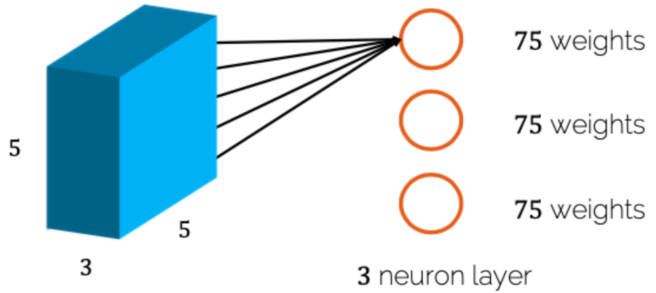
Recap: Fully-Connected Layers

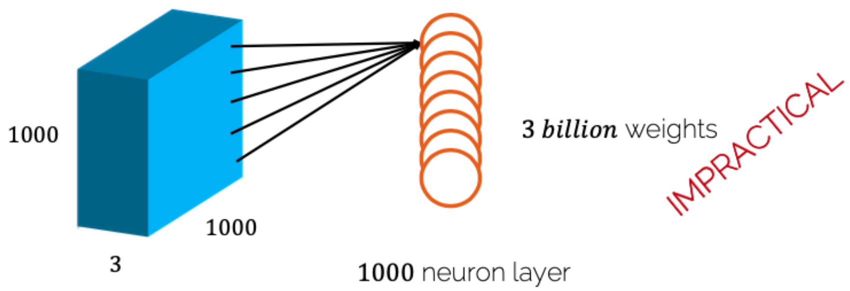
- **Regular Neural Networks:** Receive an input vector and transform it through a series of hidden layers.
- **Fully connected layers:** Each layer is made up of a set of neurons, where each single neuron is connected to all neurons in the previous layer



Convolutional Layers

- **Assumption:** Input to our Network are images
- **Disadvantage:** Normal sized images are more likely to produce the right situation

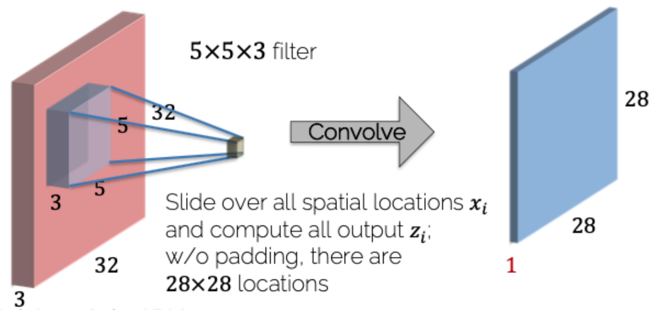




Can we reduce the number of weights in our architecture?

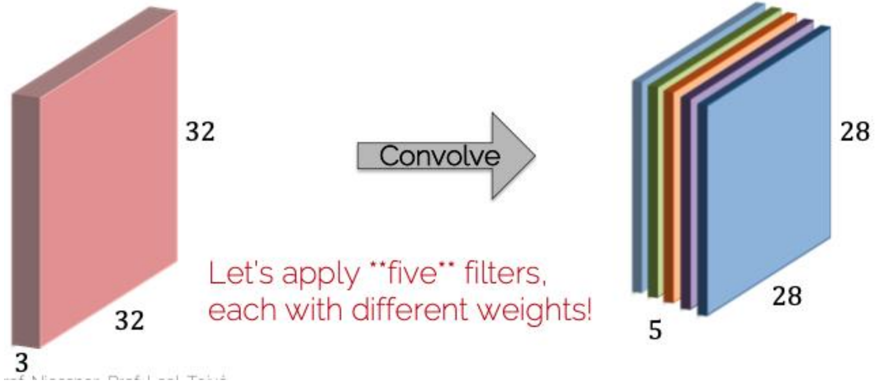
Convolutional Layers

- **Assumption:** Input to our Network are images
- **Advantage:** We can analyze the image by looking at different region instead of looking at the whole image
- **Idea:** Sliding filter over the input image (convolution) instead of matrix multiplication

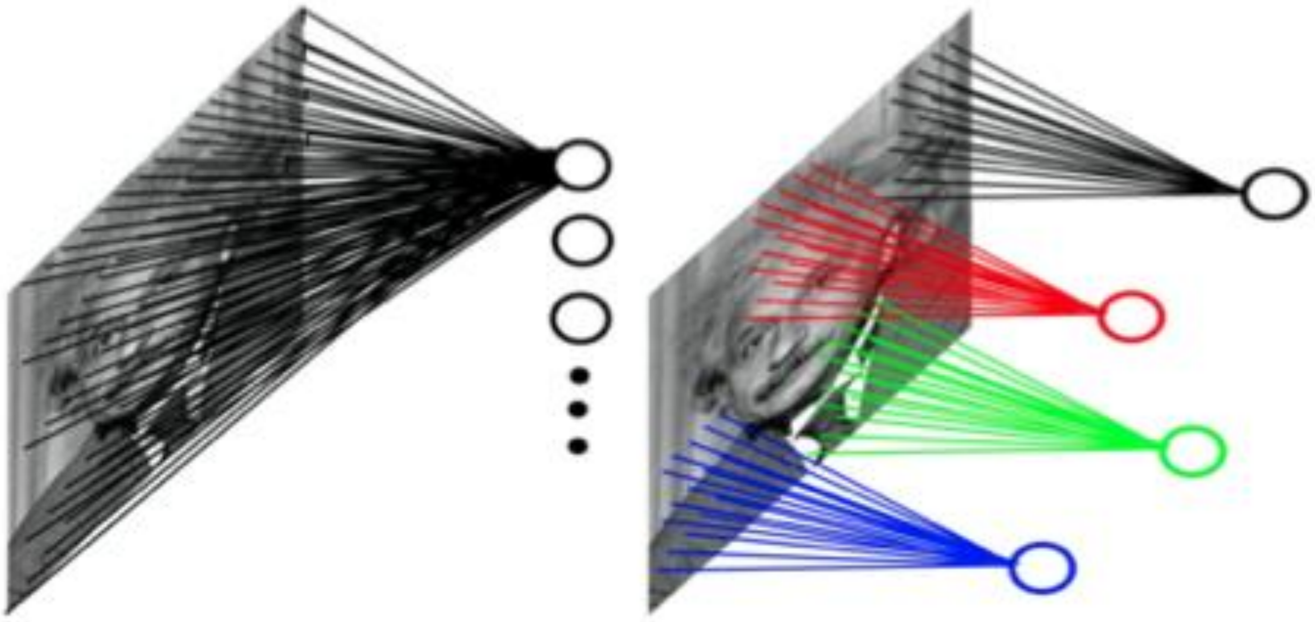


Convolutional Layers

- **Assumption:** Input to our Network are images
- **Filters:** Sliding window with the same filter parameters to extract image features
  - Concept of weight sharing
  - Extract same features independent of location



Fully Connected vs Convolution

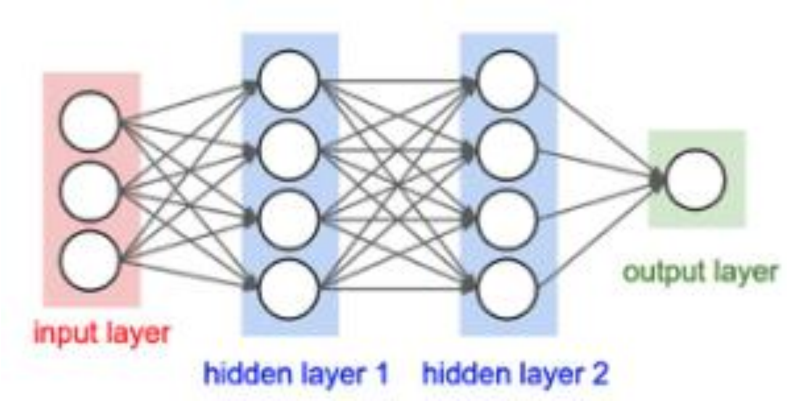


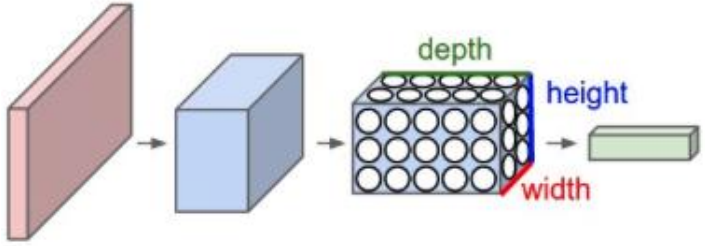
http://graphics.stanford.edu/courses/cs468-17-spring/LectureSlides/L10%20-%20intro_to_deep_learning.pdf

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





http://graphics.stanford.edu/courses/cs468-17-spring/LectureSlides/L10%20-%20intro_to_deep_learning.pdf

Recap: Batch Normalization

- Batch norm for regular 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 regular 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

$\mathbf{x}$: N × D

Normalize $\downarrow$

μ, σ: 1 × D

γ, β: 1 × D

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

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)

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

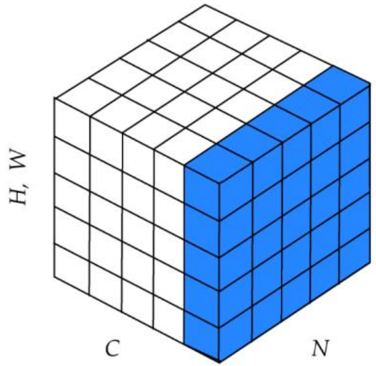
$\mathbf{x}$: N×C×H×W

Normalize

$\boldsymbol{\mu}, \boldsymbol{\sigma}$: 1×C×1×1

γ, β: 1×C×1×1

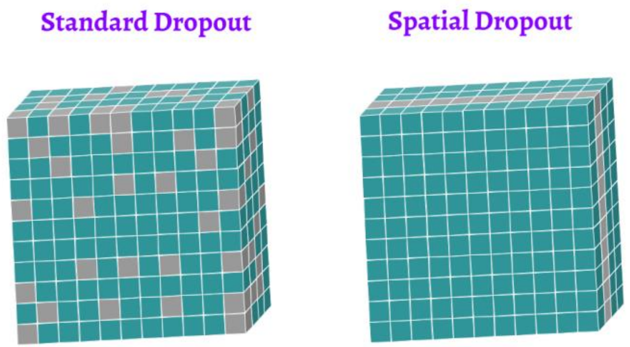
$\mathbf{y} = \gamma(\mathbf{x}-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$



http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture07.pdf

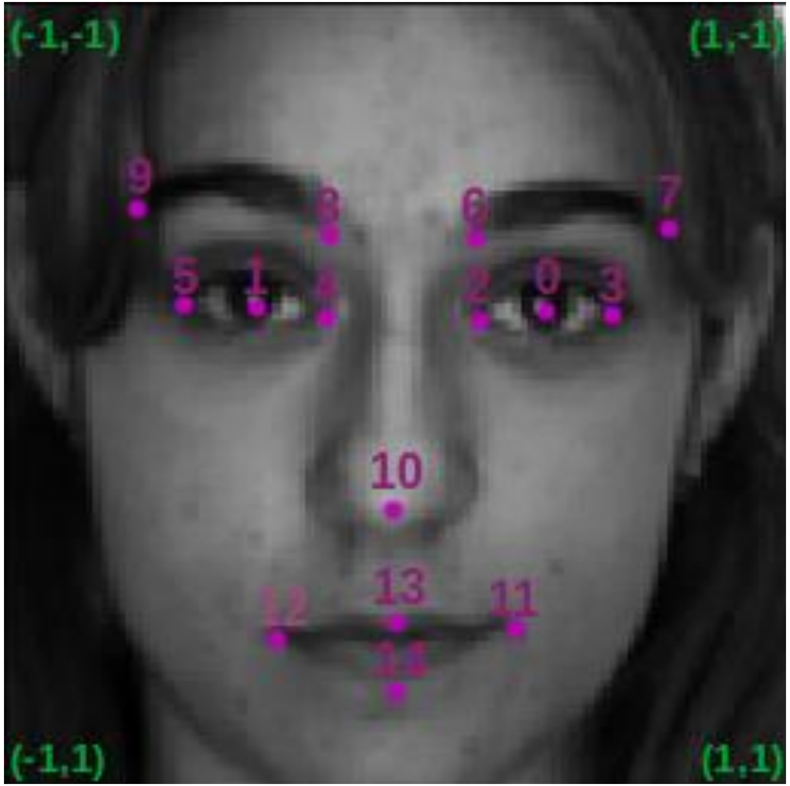
Dropout for convolutional layers

- 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
- Variant: Spatial Dropout randomly sets entire feature maps to zero



Exercise 9: Facial Keypoints Detection

Submission: Facial Keypoints



Input:
(1, 96, 96) grayscale image

Output:
(2, 15) keypoint coordinates

Submission: Metric

```
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: Details

- Submission **Start:** 16.12 13.00
- Submission **Deadline** : 05.01.2022 15.59
- Your model's **evaluation score** is all that counts!
  - Evaluation score: 1 / (2 * MSE)
  - A **score of at least 100** to pass the submission

Summary

- Monday 20.12: Watch Lecture 10
  - Popular CNN Architectures
- Wednesday 05.01.2022: Submit exercise
- Thursday 06.01.2022: Tutorial 10
  - Semantic Segmentation

Good luck &
see you next year
☺