# CS4910: Deep Learning for Robotics

David Klee
klee.d@northeastern.edu

T/F, 3:25-5:05pm
Behrakis Room 204

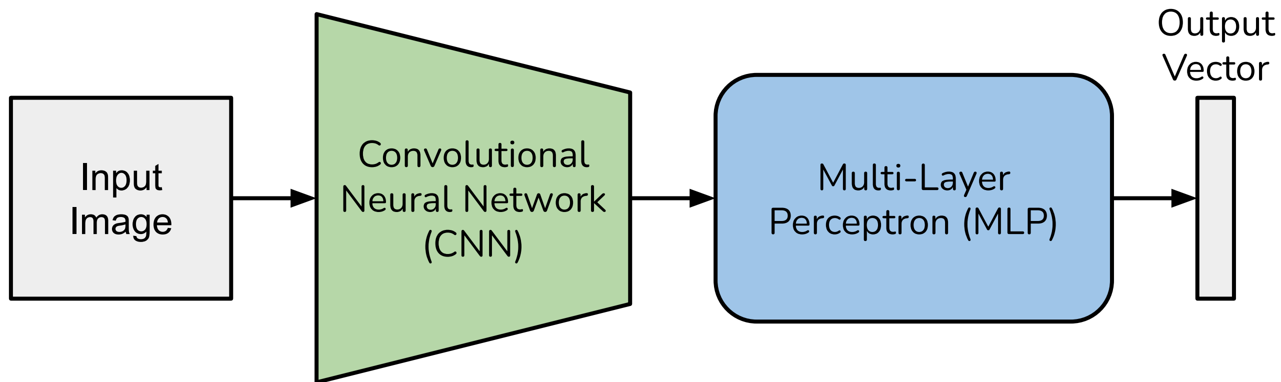
https://www.ccs.neu.edu/home/dmklee/cs4910_s22/index.html

https://piazza.com/northeastern/spring2022/cs4910a/home

# Neural Networks in Pytorch

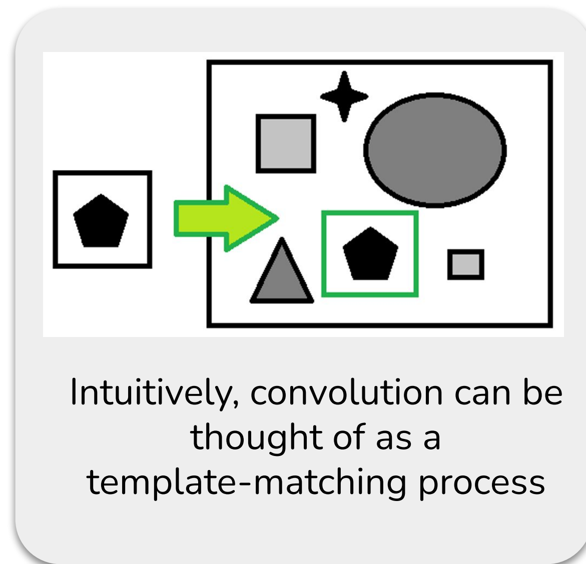
# Today’s Agenda

1. Review HW1
2. Using Convolutional layers in network
3. Normalizing functions
4. Datasets and DataLoaders
5. Training and Debugging
6. Introduce HW2
7. nuro.arm API


Input
Image
Convolutional
Neural Network
(CNN)
Multi-Layer
Perceptron (MLP)
Output
Vector

# nn.Conv2d(in_channels: int, out_channels: int, kernel_size, stride, padding, dilation, bias: bool, ...)

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$



Intuitively, convolution can be thought of as a template-matching process

# Visualizing convolution

https://ezyang.github.io/convolution-visualizer/index.html

# Output shape after convolution

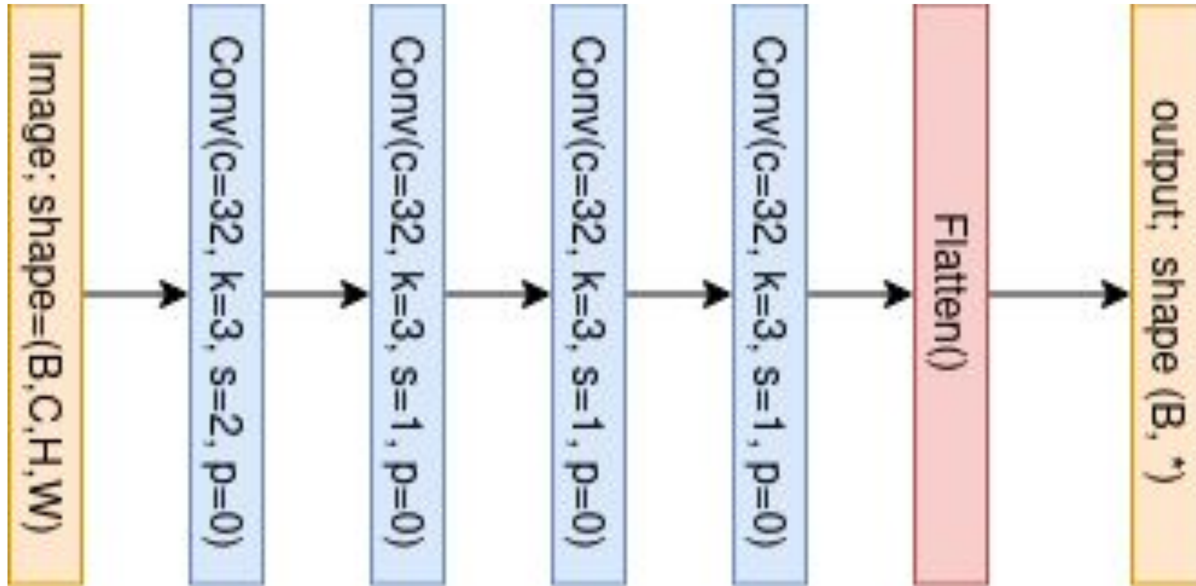
- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

# Example: Implement CNN (examples/cnn_module.py)




Add ReLU after every layer.  Make sure to calculate the output_size.

# Normalization layers

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

**nn.BatchNorm2d(num_features: int)**

- Expectation is taken over batch, height, and width dimensions, each image channel is normalized
- Increases learning speed for CNNs (bias is redundant in Conv2d)

**nn.LayerNorm**

- Normalizes values of each sample in batch
- Sometimes used on feature vectors between CNN and MLP

# Normalization layers

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

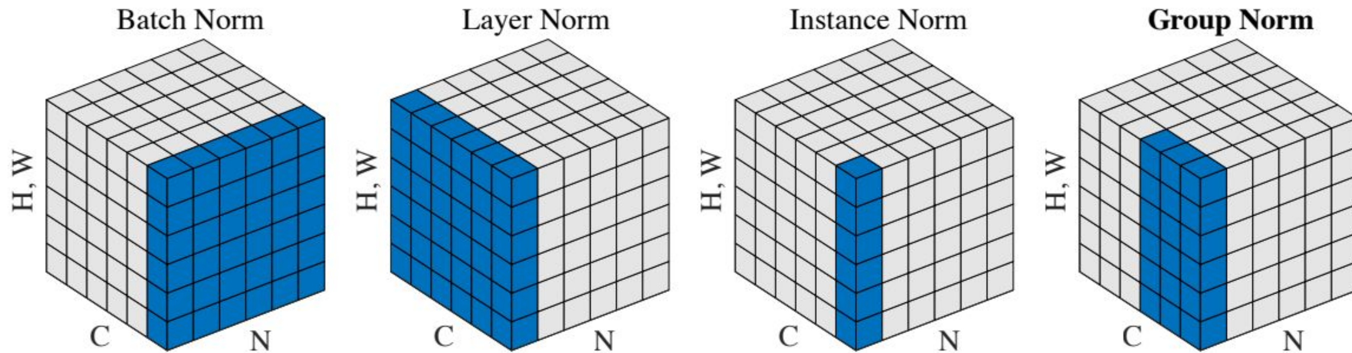



Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with $N$ as the batch axis, $C$ as the channel axis, and $(H, W)$ as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

There are two other well-known normalization layers, but they are not often used in robotic manipulation work.

# Eval vs train modes, and no_grad()

**network.train()**

- enables certain modules that keep running stats like BatchNorm
- add it after you initialize the network

**network.eval()**

- Disables these modules
- Make sure to call it before passing

**with torch.no_grad(): (...)**

- Gradient info is not stored in this context
- Use it whenever you don't need to backprop, it's faster

# from torch.utils.data import Dataset

```
class CustomDataset(Dataset):
    def __init__(self, *args):
        # reads data from file or stores paths to files

    def __len__(self):
        # provides number of data in the dataset

    def __getitem__(self, idx: int):
        # returns indexed data as a tuple of tensors or dict with tensor values
        # applies a transformation to data if applicable
        # tensors should be shaped appropriately, but kept on cpu
```

Taken from: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

# DataLoaders load batches from the dataset for training

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

```
# to get single sample
batch = next(iter(train_dataloader))

# to use it for training, it will iterate over entire dataset
for batch in train_dataloader:
	pass
```

Dataloader will stack your data samples along a new batch dimension, so you do not need to unsqueeze your samples



Taken from: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

# from torchvision.transforms import ToTensor, ...

A transform is a Callable object that modifies some part of your data

Preprocessing Data

- ToTensor() : converts from (H,W,C) np.uint image array to (C,H,W) float tensor
- Normalize(px_range) : normalizes pixel values to be within pixel range

Augmenting Data

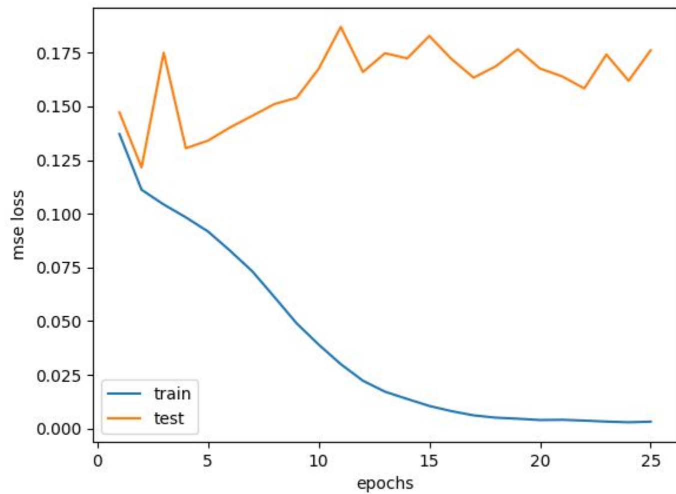
- RandomAffine : applies random affine transformation to image
- RandomHorizontalFlip

https://pytorch.org/tutorials/beginner/basics/transforms_tutorial.html, https://pytorch.org/vision/0.8/transforms.html

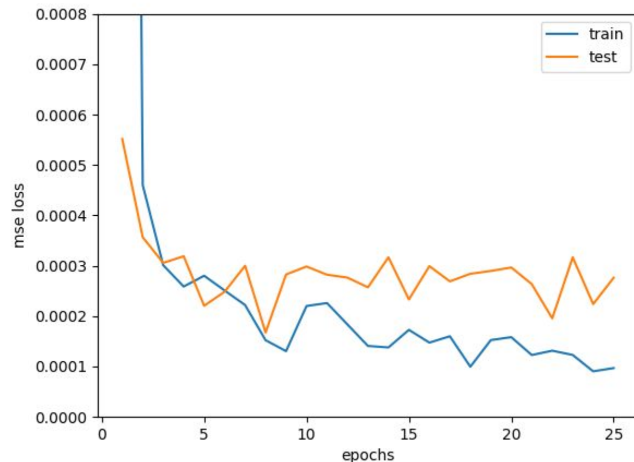
# Ensure that entire data sample is augmented

Come back to this with HW2

# Debugging Training with Loss Curves




**Overfitting:**
collect larger training dataset, add data augmentation, regularize, early-stoppage




**Instability/Jitter:**
Adjust learning rate, add gradient clipping,

While checking loss curves is useful, you should also find a way to visualize predictions/actions.
This can be immensely useful for understanding what might be off

# The training process is stochastic, so it is important that your results are statistically significant

Train multiple times and plot the mean, standard deviation of the losses

- Use different random seed every run
- Use cross-validation if needed

Use a holdout set to evaluate to double check that your network is not cheating on the test set.

# Saving and loading models

```
torch.save(model.state_dict(), PATH)      # models are saved with .pt  extension

model = ModelClass(...)
model.load_state_dict(torch.load(PATH))
```

# My rules of thumb

A lot can be done with 3-layer MLP with 256 units

Conv kernels should be 3x3

Number of channels in CNN should be inversely proportional to downsampling

Check that you can overfit on easy dataset/task first

There should be a non-linearity (activation function) between all layers

Double check tensor shapes are consistent when computing loss

# HW2

Make sure they pull again

Show training loop

# Survey to provide feedback

https://forms.gle/iP5mXYhU1Wpk8JP67

# Using xArm with nuro.arm API

https://dmklee.github.io/nuro-arm/

$ git clone https://github.com/dmklee/nuro-arm.git

$ cd nuro-arm

$ pip install .

# Calibrating xArm

Make sure it is plugged in and connected to computer

```
$ python nuro_arm/robot/calibrate.py
```

# Programming arm

```
$ python nuro_arm/examples/record_movements.py
```