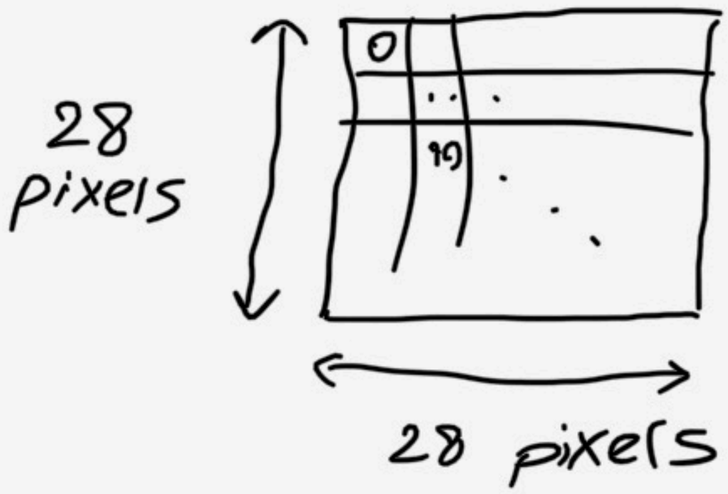
Interpreting Images as Numbers

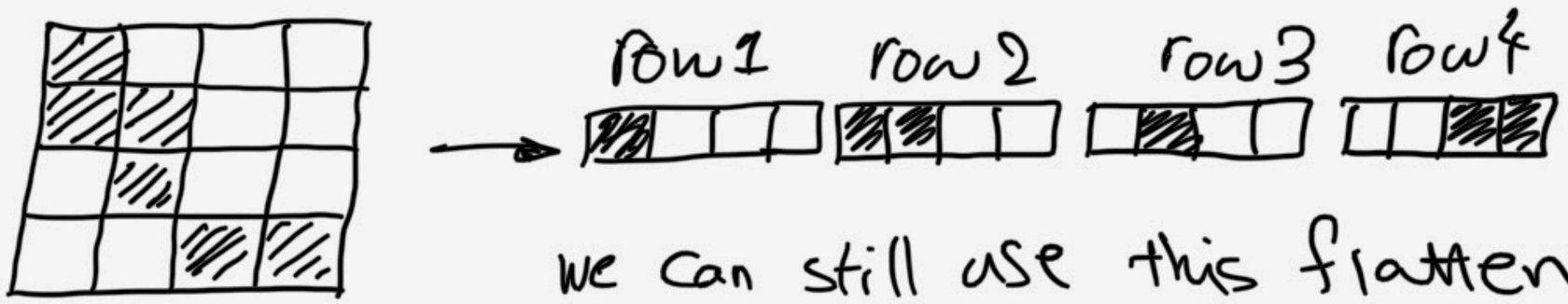


255 white

⋮ ⋮

0 black

Flattening Images:



we can still use this flattened image as the input to a MLP.

Image Normalization:

Typically done by subtracting the mean from each pixel and dividing the result by the std-dev, for each pixel.

In PyTorch → Normalize transformations

Model Validation

from torch.utils.data.sampler import SubsetRandomSampler

we use this to sample some validation data

train_Sampler = SubsetRandomSampler(train_idx)
valid_Sampler = SubsetRandomSampler(valid_idx)

train_loader(as before , sampler = train_Sampler)
we do the same for valid_Sampler.

Question: why do we need validation?
Answer: Knowing when to stop training to avoid overfitting (when valid_loss starts increasing but train_loss keep decreasing)

OpenCV

```
import cv2
import numpy as np
import matplotlib.image as mpimg
```

load → `image = mpimg.imread('img.png')`

make black/white → `gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)`

the filter →
```
filt = np.array([[-1, -2, -1],
                 [0, 0, 0],
                 [1, 2, 1]])
```

Convolute → `filtered_image = cv2.filter2D(gray, -1, filt)`

Some key terms and concepts

frequency in images

frequency in image is a rate of change.
images change in space.
e.g., when the level of brightness changes quickly from one pixel to the next.

most images have both.

High-frequency components
edges of objects in image

filters

used in images to filter out noise or amply some features

High-pass filters

a filter that sharpens an image
↓
enhance high frequency regions
it creates edges/boundaries

Convolution Kernels

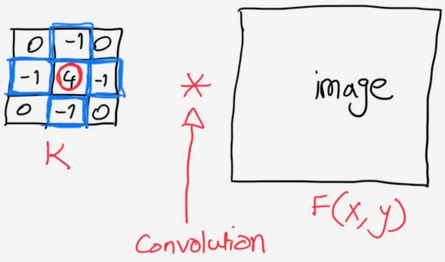
Conv. Kernel = a matrix of numbers that modifies image.

0	-1	0
-1	4	-1
0	-1	0

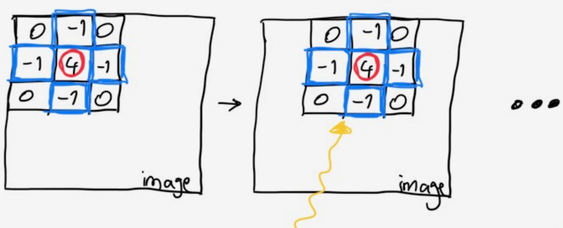
edge detection filter
Σ cell = 0

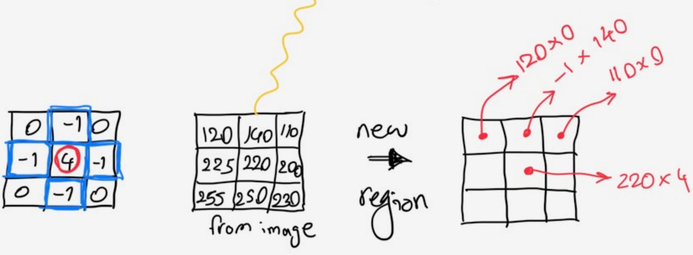
this filter, intuitively, wants to subtract the blue cells from the middel cell.

Convolution



$K * F(x,y)$ = output image

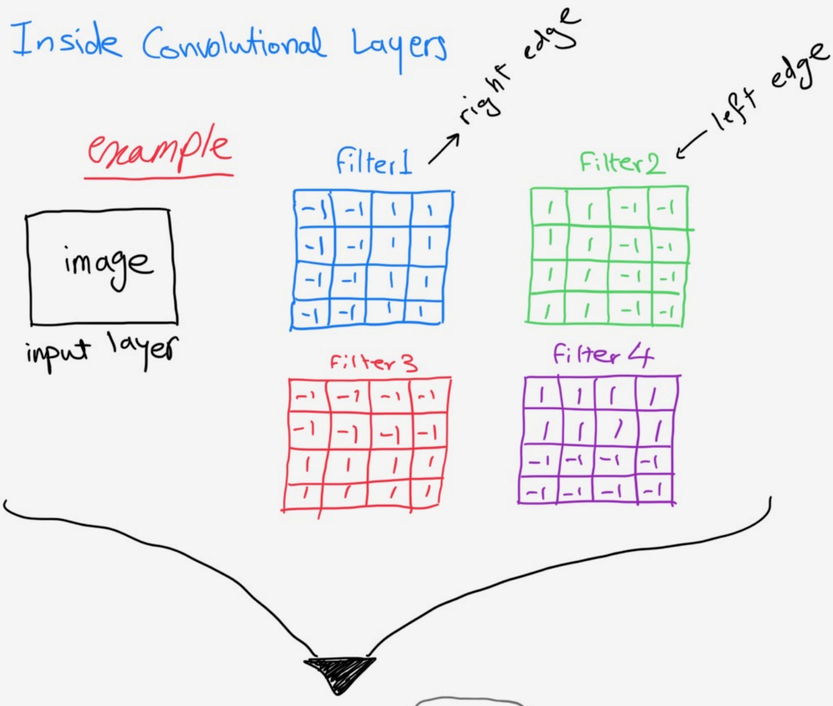


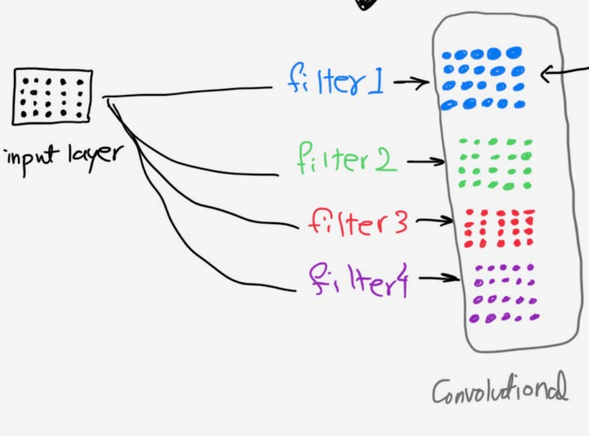


sum them up → $120 \times 0 + -1 \times 140 + \cdots$ = some number

values in the kernel matrix are called weights.

Inside Convolutional Layers





each of these sets of nodes are called feature maps or activation maps.

If we visualize each feature map, we see that each of them looks like the filtered image.

Color images:

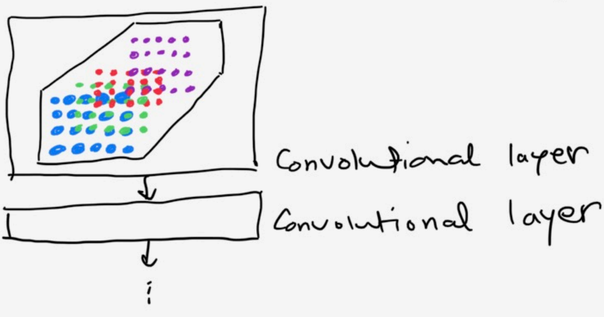
Just like GrayScale but 3D. Depth
width
height

Depth → RGB channel.

So, the filter has to be <u>3D</u>, too.
I.e., three 2D filters.

Stacking filtered Images:

We can take the four filtered images above, stack them, and use them as input to anoth Convolutional layer.



Similar to dense layer, we have weights, biases, and loss functions.

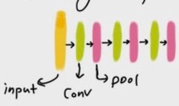
MNIST problem using PyTorch

```
from torchvision import datasets
import torchvision.transforms as transforms

transform = transforms.ToTensor()  → convert data to torch.FloatTensor

train_data = datasets.MNIST(root='data',
                            train=True,
                            transform=transform)
# similarly for test_data (train=False)

train_loader = torch.utils.data.DataLoader(
                            train_data,
                            batch_size=b_size,
                            num_workers=workers)

# Inspecting the data:
one batch of the data {
    dataiter = iter(train_loader)
    images, labels = dataiter.next()
    images = images.numpy()

one image ← img = np.squeeze(images[1])

# Defining the Model:
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 10)
to prevent overfitting ← self.dropout = nn.Dropout(0.2)
    def forward(self, x):
flatten ← x = x.view(-1, 28*28)
apply layer ← x = F.relu(self.fc1(x)); x = self.dropout(x)
        same for fc2.
        x = self.fc3(x)
        return x

#question: why do we need activation functions?
to scale the outputs of a layer so that they
are a consistent, small value.
model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
# Finally, train the model!
n_epochs = 30   # start small!

for epoch in range(n_epochs):
    train_loss = 0
    for data, target in data_loader:
        optimizer.zero_grad()  → clear all the gradients

        output = model(data)  forward pass
        loss = criterion(output, target)  compute loss

        loss.backward()  → backward propagation
        optimizer.step()  → parameter update

        train_loss += loss.item() * data.size(0)

    train_loss = train_loss/len(train_loader.dataset)
# test!   model.eval()
for data, target in test_loader:
    output = model(data)
    loss = criterion(output, target)
    test_loss += loss.item() * data.size(0)
    _, pred = torch.max(output, 1)
    correct =
        np.squeeze(pred.eq(target.data.view_as(pred)))

* Question: nn.CrossEntropy, does both of
softmax and NLLLoss. What if we want
the output of the model as class probabilities
instead of scores?
Answer: use Softmax and NLLLoss separately.

    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

test_loss = test_loss/len(test_loader.dataset)

for i in range(num_classes):
    if class_total[i] > 0:
        accuracy[i] = np.sum(class_correct[i])/
                      np.sum(class_total[i]))

overall_accuracy = np.sum(class_correct)/np.sum(class_total)
```

Components of CNNs

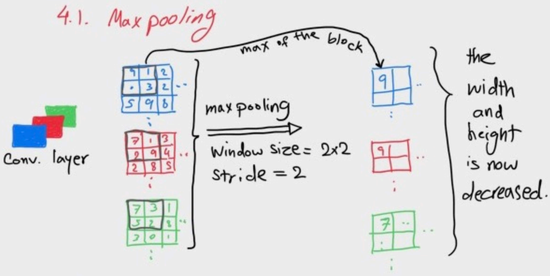
1. number of filters
2. size of each filter
3. stride → when sliding, we choose how much we slide at each step.

   when stride = 1, the filtered image will have the same size as the input image. The size of the filtered image decreases when we increase the stride.
4. pooling layers



Convolutional layers can get too large, i.e., high-param. we use pooling to deal with that.

4.1. Max pooling

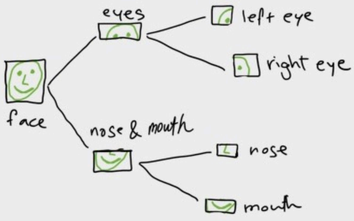


4.2. Average pooling

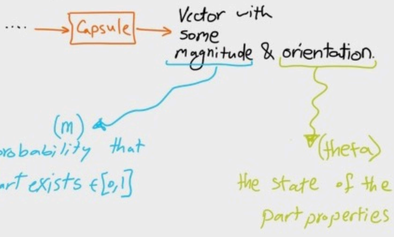
not usually for edge detection

4.3. Alternatives to pooling:

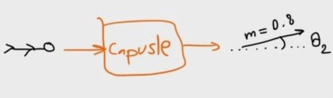
pooling throws away some info, i.e., discards some pixels.
In classification pooling is fine. But in something like face recognition it is bad.
Instead, some people use strategies to keep some spatial info. An example is Capsule Network, which learns spatial relationships between parts.



Capsule: Collection of nodes that contain info about a sub-part of the image (width, orientation, color, ...)



The output of these Capsules can be used to make a "tree".



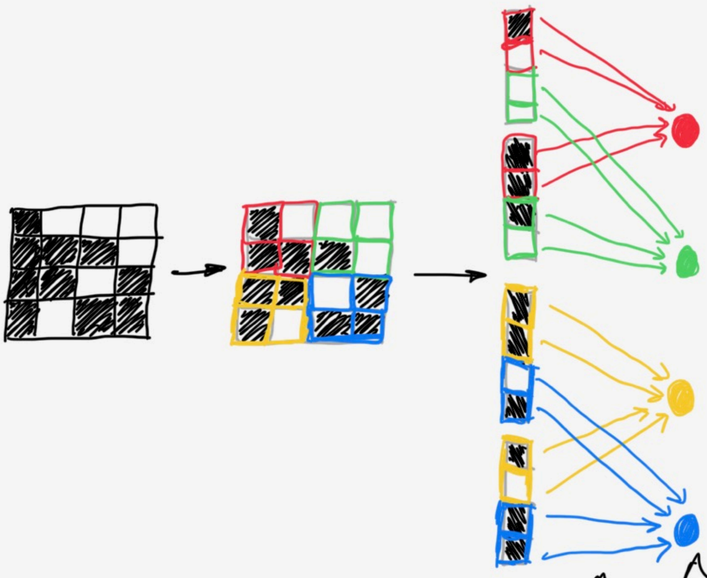
MLPs versus CNNs

MLP:

- only use fully-connected layers
- only accept vectors as input
- tosses the connections between vector elements

CNN:

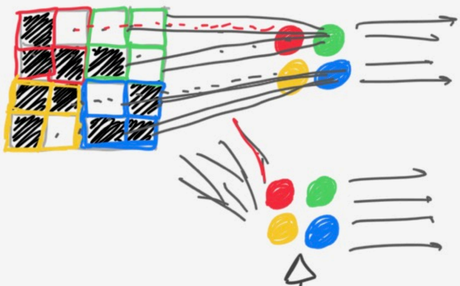
- no need for all-fully-connected layers



no longer FC.
also called locally connected layer.

every hidden node is in charge of one region!

How does CNN look like?



a second set of hidden nodes.

Convolutional Layers in PyTorch

Defining Filters:

```
filter_vals = np.array([-1,-1,1,1],
                       [-1,-1,1,1],
                       [-1,-1,1,1],
                       [-1,-1,1,1])
filters = np.array([
            filter_vals,
            -filter_vals,
            filter_vals.T,
            -filter_vals.T])
```

Defining Convolutional Layer:

```
class MyNN(nn.Module):

★ def __init__(self, weights):
      super(MyNN, self).__init__()
      h,w = weight.shape[2:]
      self.conv = nn.Conv2d(1,4,
                            kernel_size=(h,w),
                            bias=False)
```

input depth

output depth

define a Convolutional layer...

...that has four filters...

...all of which are grayscale.

```
      self.conv.weight = torch.nn.Parameter(weight)
      self.pool = nn.MaxPool2d(2,2)

★ def forward(self, x):
      conv_x = self.conv(x)
      activated_x = F.relu(conv_x)
      pooled_x = self.pool(activated_x)
      return conv_x, activated_x, pooled_x
```

→ Compute output of conv. layer.

pre-activation

post-activation

```
weight = torch.from_numpy(filters).unsqueeze(1).
                                   type(torch.TensorFloat)
model = MyNN(weight)
```

```
nn.Conv2d(in_channels,      → input depth
          out_channels,     → output depth
          kernel_size,      → commonly 3
          stride=1,
          padding=0)
```

Input depth { RGB = 3 / BW = 1

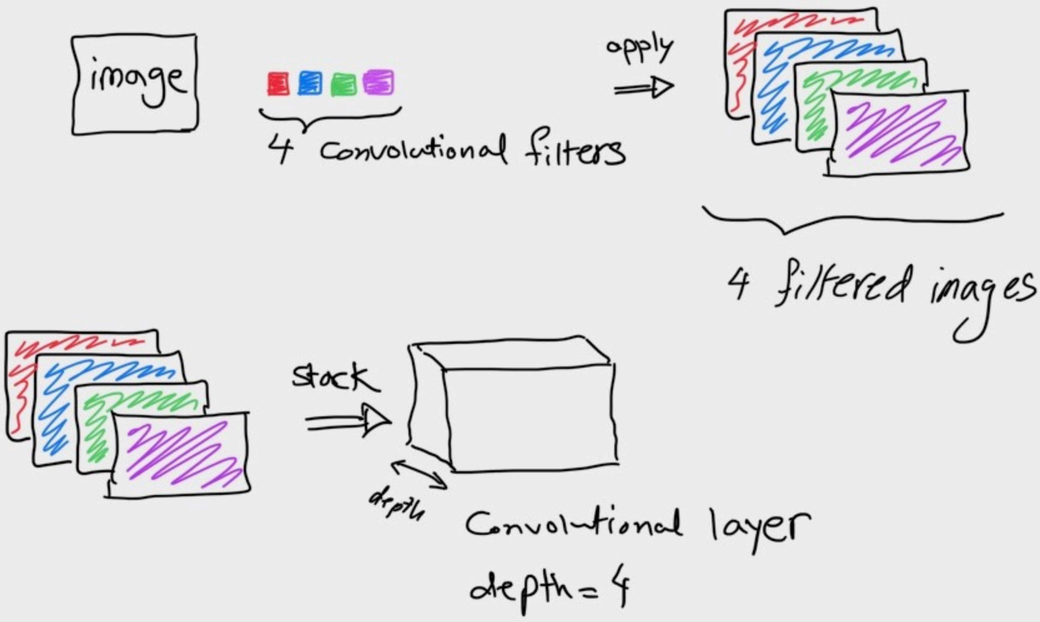
```
nn.MaxPool2d(kernel_size, stride)
  forward:
      x = F.relu(self.conv1(x))
      x = self.pool(x)
```

Sequential Models

There is another way in pytorch to create CNNs: Sequential wrapper

```
def __init__(self):
    super(ModelName, self).__init__()
    self.features = nn.Sequential(
        ① nn.Conv2d(1,16, 2, stride=2,
          nn.MaxPool2d(2,2),
          nn.ReLU(True),
        ② nn.Conv2d(16, 32,3, padding=1),
          nn.MaxPool2d(2,2),
          nn.ReLU(True)
    )
```

Overview of a Convolutional Layer



As we learn, our goal is to find good weights for filters.

Number of Parameters in a Conv layer

- out_channels = K
- kernel_size = F
- last value in input_shape tuple = D_in
  - 1 for B/W
  - 3 for RGB

$$\underbrace{K}_{\#\text{filters}} \times \underbrace{F \times F \times D_in}_{\text{weights per filter}} + \underbrace{K}_{\text{one bias term per filter}} \quad \text{parameters}$$

Shape of a Conv Layer

- other than K and F above:
- S = stride
- P = padding
- W_in = height or width of previous layer

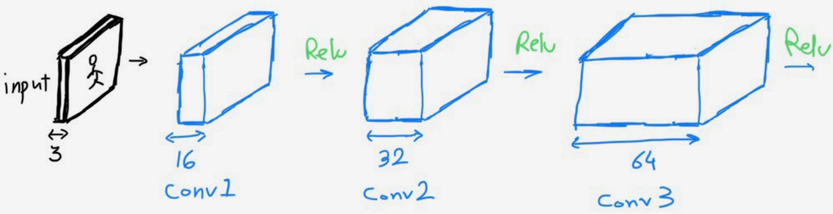
K → depth of the layer

$$(W_in - F + 2P)/S + 1$$

→ spatial dimentions of a layer

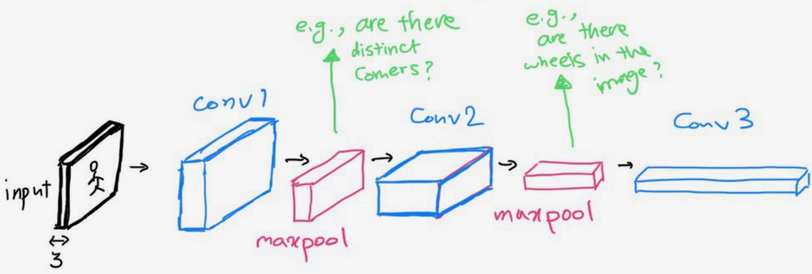
we compute this for the final level.

Overview of a full CNN model




```
self.Conv1 = nn.Conv2d(3,16,3, padding=1)
self.Conv2 = nn.Conv2d(16,32,3, padding=1)
self.Conv3 = nn.Conv2d(32,64,3, padding=1)
```

When we pass the image from these Conv. layers, the width and height does not change but the depth increases. This results in too many params. So, we use max pooling!




```
self.Conv1 = nn.Conv2d(3,16,3, padding=1)
self.Conv2 = nn.Conv2d(16,32,3, padding=1)
self.Conv3 = nn.Conv2d(32,64,3, padding=1)
self.maxpool = nn.MaxPool2d(2,2)
```

Kernel/filter size → 2; Stride → 2

We achieve a <u>deep</u> but with <u>small width and height</u> output.

regarding the size and stride in maxpool:

* if we want to see all pixels and down-sample an image by a factor of 4, then, MaxPool2d(4,4) is our best choice.

assume this is the final maxpool layer.
we can flatten it and give it to a fully-connected layer to make a binary classification.

Note: as we go through more and more layers, the representation becomes more and more specific, i.e., feature-level.

CIFAR problem Using PyTorch

```
# to check if CUDA is available:
train_on_gpu = torch.cuda.is_available()

# Load the data
transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((1/2,1/2,1/2),(1/2,1/2,1/2))
        ])

train_data = datasets.CIFAR10('data', train=True,
                              download=True,
                              transform=transform)
test_data = datasets.CIFAR10('data', train=False,
                             download=True,
                             transform=transform)

num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=batch_size,
                                           sampler=train_sampler,
                                           num_workers=nw)
valid_loader = torch.utils.data.DataLoader(valid_data,
                                           batch_size=batch_size,
                                           sampler=valid_sampler,
                                           num_workers=nw)
test_loader = torch.utils.data.DataLoader(test_data,
                                          batch_size=batch_size,
                                          num_workers=nw)

# it is a classification task
classes = ['airplane', ...., 'truck']

# model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3,16,3, padding=1)
        self.conv2 = nn.Conv2d(16,32,3, padding=1)
        self.conv3 = nn.Conv2d(32,64,3, padding=1)
        self.pool = nn.MaxPool2d(2,2)
        self.fc1 = nn.Linear(64*4*4, 500)
        self.fc2 = nn.Linear(500, 10)
        self.dropout = nn.Dropout(.25)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(-1, 64*4*4)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# use the defined model
model = Net()
if train_on_gpu: model.cuda()

# Training step

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

valid_loss_min = np.Inf

for epoch in range(epochs):
    train_loss = valid_loss = 0.0

    model.train()
    for data, target in train_loader:
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)

    model.eval()
    for data, target in valid_loader:
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        output = model(data)
        loss = criterion(output, target)
        valid_loss += loss.item() * data.size(0)

    # Compute the final values for losses
    train_loss /= len(train_loader.dataset)
    valid_loss /= len(valid_loader.dataset)

    # save only if there is an improvement
    if valid_loss <= valid_loss_min:
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

torchvision

data to validate

0.2

to make batches

sees 32x32x3

sees 16x16x16

sees 8x8x32

move tensors to Cuda

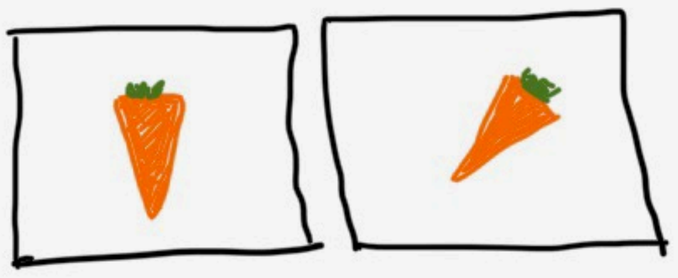
train mode

eval mode

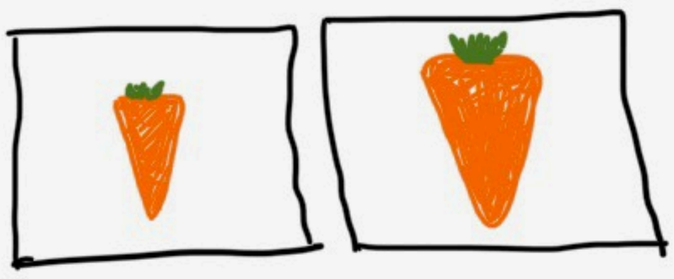
Summary of the Common Strategy:

1. After each conv layer, we do relu and then maxpooling.
2. Then, we flatten the data.
3. Pass through dropouts

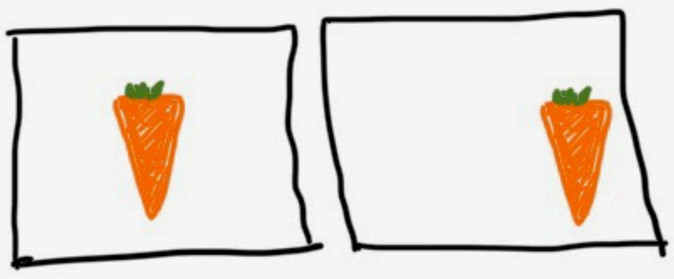
Invariant Representation



→ Rotation Invariance



→ Scale invariance



→ translation invariance

★ Ideally, a model should understand the above.

CNNs have translation invariance (partial):

5	2	8	5
7	3	9	2
2	1	4	6

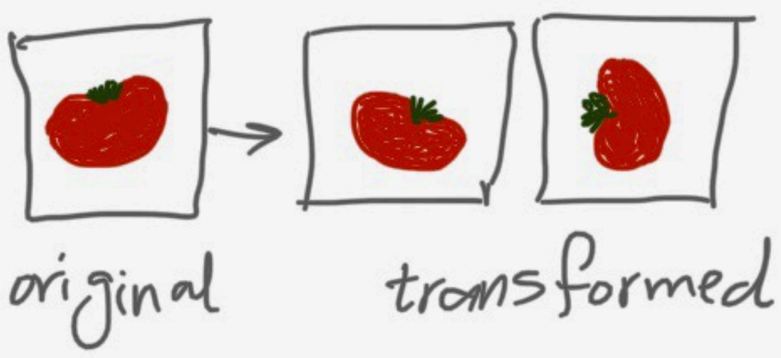
5	2	8	5
7	3	9	2
2	1	4	6

Reason:

a little shifting around does not change maxpool result!

An strategy to make models invariant:

Data Augmentation



Benefits: avoiding overfitting

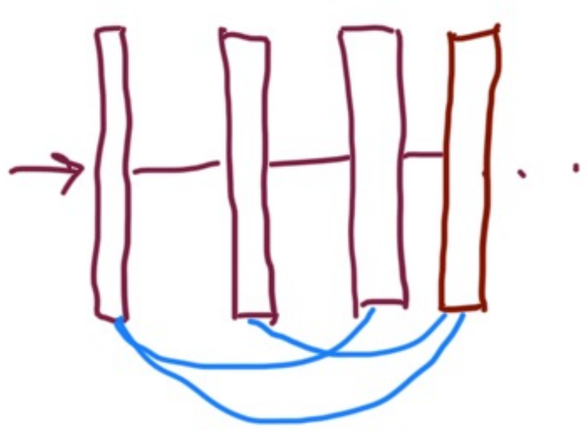
increasing the accuracy

ResNet Architechture

- up to 152 layers

- they observed that adding more layers increases the performance up to a point after which it decreases.
- the problem is the "vanishing gradient" phenomenon.
- Simply put, as the number of layers increases, during the backward pass the loss gets smaller and smaller and lose its effectiveness.
- So, in ResNet, they added shortcuts between layers to avoid that.

ordinary architecture



ResNet

Google's Deep Dreams

this filter is the key. It can be some specific (deeper) filters that detect houses.

Transfer Learning Cheatsheet

small dataset
similar images

1. Remove end of the NN
2. Add a FC layer that matches my # of classes.
3. Randomize weights of FC and freeze the pre-trained weights.
4. train to update FC weights.

* why freeze the pretrained model? avoiding overfitting.

small dataset
different images

1. Only keep some of the first few layers in the pretrained model.
2. add a fc network that matches my # of classes.
3. Randomize fc weights and freeze layers from pretrained one.
4. train to update weights of FC.

* why freeze the pretrained model? avoiding overfitting

large dataset
similar images

1. Remove the last FC in the pretrained model.
2. Replace it with a layer that matches my # classes.
3. randomly init FC weights.
4. Use the pretrained model to initialize the rest of weights.
5. Retrain the whole thing.

large dataset
different images

1. Remove the last FC from the pretrained network.
2. Replace it with a new FC that matches your # classes.
3. Retrain the whole network from scratch with random init weights.

Alternatively, one can do the approach on the left.

Initializing weights with pretrained model may speed up training

Transfer Learning in PyTorch

Case: freeze all the weights and only train a new last layer (FC)

```
# load the pretrained model
vgg16 = models.vgg16(pretrained=True)

# we can access components of model
print(vgg16.classifier[6].in_features

# freeze all but last layers in model
for param in vgg16.features.parameters():
    param.requires_grad = False

# change the final classifier layer
vgg16.classifier[6] = nn.Linear(4096, 5)

# loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(vgg16.classifier.parameters(), lr=0.01)
```

we only want to change the last classifier group

```
# send to gpu if possible
if train_on_gpu: vgg16.cuda()

# train loop

for epoch in range(n_epochs):
    train_loss = 0.0
    for batch_i, (data, target) in enumerate(train_loader):
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        optimizer.zero_grad()
        output = vgg16(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    train_loss /= len(train_loader)
```

small; because we are using a pretrained model.

Implementing Style transfer in images

Content Image → VGG19 Arch. → the output of the deepest Conv. layer in vgg19 is the content representation.

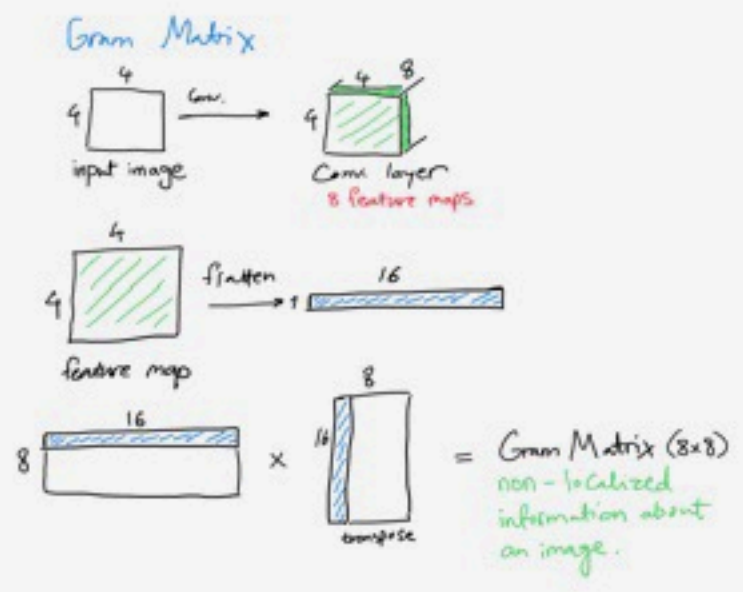
Style Image → VGG19 Arch. → we get outputs of multiple Conv. layers

Content representation (content image) C_c ≈ Content representation (target image) T_c

ideally

$$\underbrace{\mathcal{L}_{Content}}_{\text{Content loss}} = \frac{1}{2} \sum (T_c - C_c)^2$$

Gram Matrix



List of Gram Matrices of style image (S_s)
List of Gram Matrices of target image (T_s)

$$\underbrace{\mathcal{L}_{style}}_{\text{Style loss}} = \alpha \sum_i \omega_i \left(T_{s_i} - S_{s_i}\right)^2$$

$$\text{Total loss} = \underbrace{\alpha}_{\text{Content weight}} \mathcal{L}_{Content} + \underbrace{\beta}_{\text{style weight}} \mathcal{L}_{style}$$

- Style weight is usually much larger than the Content weight.
- to add more style to target image, decrease $\frac{\alpha}{\beta}$.

In PyTorch:

```
from PIL import Image  → load any image
from io import BytesIO
import numpy as np

import torch
import torch.optim as optim
import requests
from torchvision import transforms, models
```

Conv and pooling layers

```
# get feature layers of vgg19:
vgg = models.vgg19(pretrained = True).features

# freeze the parameters:
for param in vgg.parameters():
    param.requires_grad_(False)

# move to device
device = torch.device("cuda" if torch.cuda.is_available()
                      else "cpu")
vgg.to(device)

# Load Content/Style images
def load_image(img_path, max_size=400,
               shape=None):
    if "http" in img_path:
        response = requests.get(img_path)
        image = Image.open(BytesIO(response.content))
                 .Convert('RGB')
    else:
        image = Image.open(img_path).Convert('RGB')

    if max(image.size) > max_size:
        size = max_size
    else:
        size = max(image.size)

    if shape is not None:
        size = shape

    in_transform = transforms.Compose([
                transforms.Resize(size),
                transforms.ToTensor(),
                transforms.Normalize(
                    (0.485, 0.456, 0.406),
                    (0.229, 0.224, 0.225))])

    image = in_transform(image)[:3,:,:].unsqueeze(0)
    return image
```

actual loading

resize all to a manageable size

discard alpha channel

```
content = load_image('img1.jpg').to(device)
style = load_image('img2.jpg', shape=content.shape[-2:]).
                                 to(device)
```

to align easier

```
def im_Convert(tensor):
    image = tensor.to('cpu').clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225))
            + np.array((0.485, 0.456, 0.406))
    image = image.clip(0,1)
    return image
```

un-normalize an image

```
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,10))
ax1.imshow(im_Convert(content))
ax2.imshow(im_Convert(style))
```

pass image through layers

```
def get_features(image, model, layers=None):
    if layers is None:
        layers = {'0': 'Conv1-1',
                  '5': 'Conv2-1',
                  '10': 'Conv3-1',
                  '19': 'Conv4-1',
                  '21': 'Conv4-2',
                  '28': 'Conv5-1'}

    features = {}
    x = image
```

Contains all the modules of a model

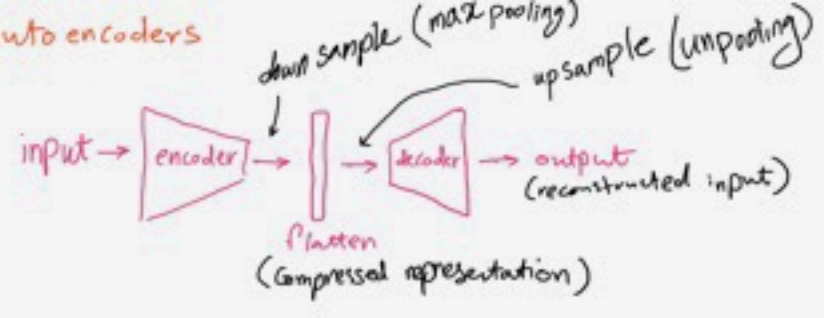
```
    for name, layer in model._modules.items():
        x = layer(x)
        if name in layers:
            features[layers[name]] = x
    return features

def gram_matrix(tensor):
    b, d, h, w = tensor.size()
    tensor = tensor.view(b*d, h*w)
    gram = torch.mm(tensor, tensor.t())
    return gram

# Finally!
content_features = get_features(content, vgg)
style_features = get_features(style, vgg)

style_grams = {layer: gram_matrix(style_features[layer])
               for layer in style_features}
```

better initialize with content img

```
target = content.clone().requires_grad_(True).to(device)

style_weights = {'Conv1-1': 1.0,
                 'Conv2-1': 0.75,
                 'Conv3-1': 0.2,
                 'Conv4-1': 0.2,
                 'Conv5-1': 0.2}
content_weight = 1  → alpha
style_weight = 1e6  → beta

optimizer = optim.Adam([target], lr=0.003)

for _ in range(steps):
    target_features = get_features(target, vgg)
    content_loss = torch.mean(
                (target_features['conv4-2']
                - content_features['conv4-2'])
                **2)
    style_loss = 0
    for layer in style_weights:
        target_feature = target_features[layer]
        target_gram = gram_matrix(target_feature)
        _, d, h, w = target_feature.shape
        style_gram = style_grams[layer]
        layer_style_loss =
                style_weights[layer] *
                torch.mean((target_gram -
                            style_gram)**2)
        style_loss += layer_style_loss/(d*h*w)

    total_loss = content_weight *
                 content_loss +
                 style_weight *
                 style_loss

optimizer.zero_grad()
total_loss.backward()
optimizer.step()
```

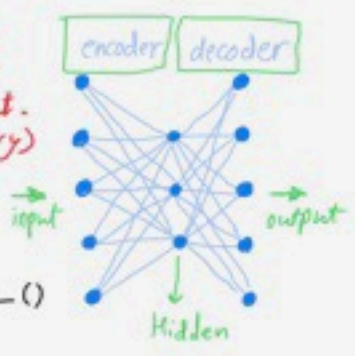
update the target image

Auto encoders



In PyTorch: option 1: MLP

* the output image and input image are kinda diffent. (blurry)



$\boxed{\text{data handling as before}}$

```
class AutoEncoder(nn.Module):
    def __init__(self, encoding_dim):
        super(AutoEncoder, self).__init__()
        # encoder:
        self.fc1 = nn.Linear(28x28, encoding dim)
        # decoder
        self.fc2 = nn.Linear(encoding_dim, 28x28)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        return x

encoding_dim = 32
model = AutoEncoder(encoding_dim)

★ criterion = nn.MSELoss() → good for pixels comparison not probabilities
optimizer = optim.Adam(model.parameters(), lr = 0.001)
```

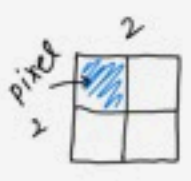
$\boxed{\text{train loop as before; but we are not interested in labels here. We just want to compare the input image with reconstructed image.}}$

↳ at each step: loss = criterion(outputs, images)
this tells us how good of a reconstruction our model does

option 2: Conv. layers

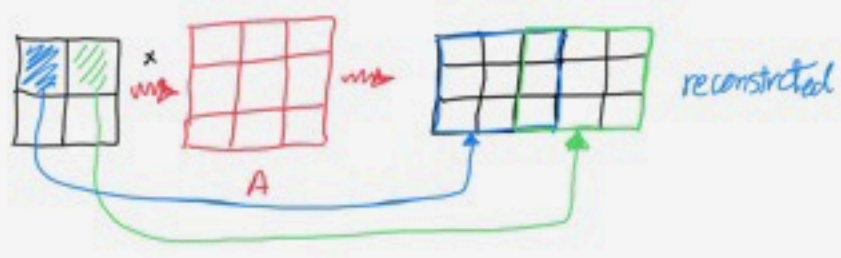
* a better solution but still has issue of "blocks" in the image.

Key point: transpose Conv. layers

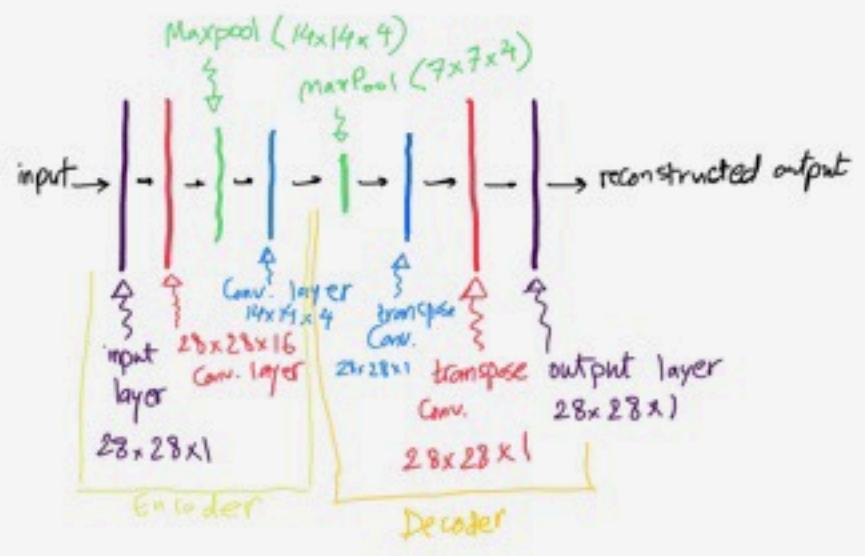


→ we take each pixel, and use a kernel for upsampling. e.g., we use a 3x3 kernel for a 2x2 image. (let's call this A)

For each pixel in the image, we multiply the value of it with each element of the kernel.



Convolutional Autoencoders



In pytorch, transpose Conv Layers are nn.ConvTranspose2d.

```
class ConvAutoencoder(nn.Module):
    def __init__(self):
        super(ConvAutoencoder, self).__init__()

        # encoder layers
depth: 1→16     self.conv1 = nn.Conv2d(1,16, 3, padding=1)   → number of feature maps
depth: 16→4     self.conv2 = nn.Conv2d(16,4,3, padding=1)
                self.pool = nn.MaxPool2d(2,2)   size, stride

        # decoder layers
        self.t_conv1 = nn.ConvTranspose2d(4,16,2, stride=2)
        self.t_conv2 = nn.ConvTranspose2d(16,1,2, stride=2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)

        x = F.relu(self.t_conv1(x))
        x = F.sigmoid(self.t_conv2(x))
        return x

model = ConvAutoencoder()
Criterion = nn.MSELoss()
```

$\boxed{\text{most of the train loop is as before. just note following statements: outputs = model(images), loss = Criterion(outputs, images). i.e., we want outputs and images be ideally identical.}}$

option 3: upsampling

smoother and more similar images

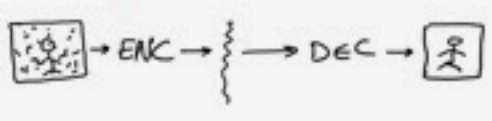
We change option 2's decoder by adding a few layers. (upsampling)

* idea: replace a transpose Conv. layer with with a conv. layer + an upsample layer.

The change with option 2's code is just in the "forward" portion of the module.

```
#encoder
    ⋮
#decoder
x = F.upsample(x, scale_factor=2, mode='nearest')
x = F.relu(self.conv4(x))
x = F.upsample(x, scale_factor=2, mode='nearest')
x = F.sigmoid(self.conv5(x))
```

Denoising Autoencoders



In PyTorch

```
class ConvDenoiser(nn.Module):
    def __init__(self):
        super(ConvDenoiser, self).__init__()

Encoder
        self.conv1 = nn.Conv2d(1,32, 3, padding=1)
        self.conv2 = nn.Conv2d(32,16,3, padding=1)
        self.conv3 = nn.Conv2d(16,8,3, padding=1)
        self.pool = nn.MaxPool2d(2,2)

Decoder
        self.t_conv1 = nn.ConvTranspose2d(8,8,3, stride=2)
        self.t_conv2 = nn.ConvTranspose2d(8,16,2, stride=2)
        self.t_conv3 = nn.ConvTranspose2d(16,32,2, stride=2)
        self.conv_out = nn.Conv2d(32,1,3, padding=1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)

        x = F.relu(self.t_conv1(x))
        x = F.relu(self.t_conv2(x))
        x = F.relu(self.t_conv3(x))
        x = F.sigmoid(self.conv_out(x))
        return x
```

Choosing Initial weights

- **All zeros or all ones** NOT GOOD

  inside the init function of the Module:

```
for m in self.modules():
    if isinstance(m, nn.Linear):
        nn.init.constant_(m.weight, const_w)
        nn.init.constant_(m.bias, 0)
```

Comparison: model_zero vs. model_one

both have low accuracy

- **Uniform random weights**

  e.g., np.random.uniform(−3, 3, [1000])

  values between 3, −3 — generate 1000 numbers

Writing an initialization function:

```
def weight_init_uniform(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != 1:
        m.weight.data.uniform_(0.0, 1.0)
        m.bias.data.fill_(0)
```

Module

for every Linear layer in a model...

still not the best

uniform distribution for weights

set bias to zero

```
model = Net()
model.apply(weight_init_uniform)
```

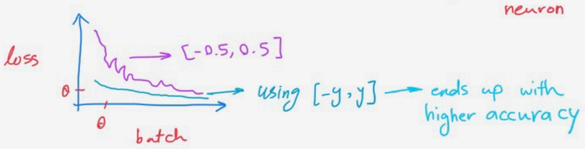
- **Relationship between #inputs and uniform range**

  more input ; smaller range

  Rule of thumb:

  choose uniform random weights from the range $[-y, +y]$. $y = \frac{1}{\sqrt{n}}$

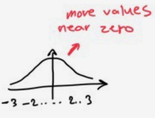
  n → #inputs of the neuron



- **Normal Distribution**

  e.g., np.random.normal(0, 1, [1000])

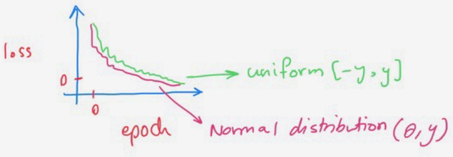
  mean — std dev



  typically, mean = 0 and stddev = $1/\sqrt{n}$.

  in weight_init function, we do the following,

```
def weight_init_normal(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != 1:
        n = m.in_features
        y = 1.0 / np.sqrt(n)
        m.weight.data.normal_(0, y)
        m.bias.data.fill_(0)
```



* In PyTorch, the default behavior for initialization of weights is Cool! Actually, it is usually the "uniform" distribution. However, there are cases where "normal" distribution will be better.