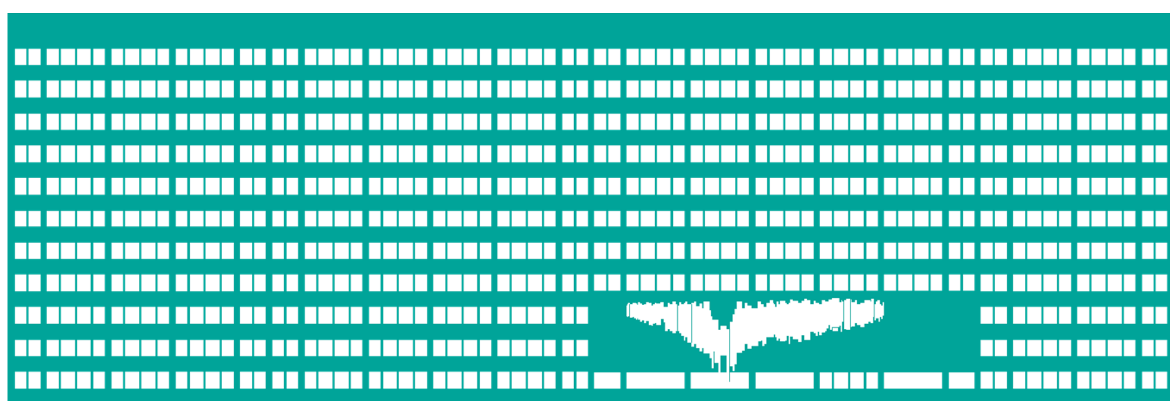
VŠB TECHNICKÁ
UNIVERZITA
OSTRAVA

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | VSB TECHNICAL UNIVERSITY OF OSTRAVA

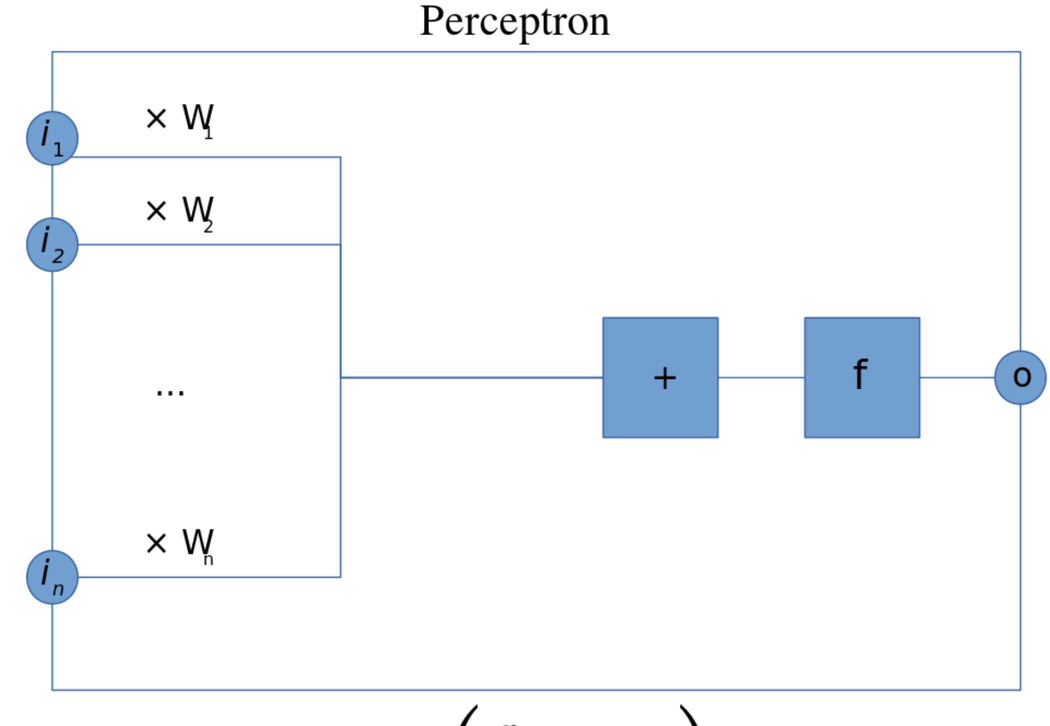


www.vsb.cz

Perceptron - Convolution - Polling - CNN - PyTorch

Radovan Fusek

Perceptron

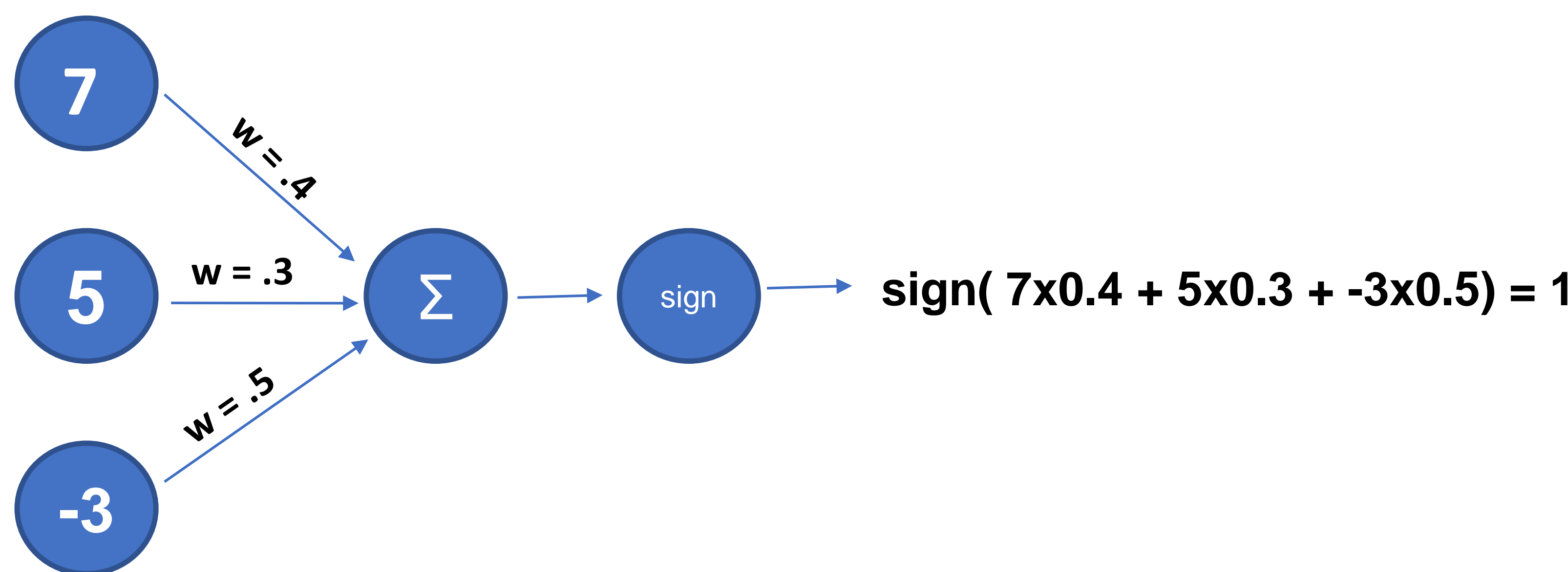


$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

The appropriate weights are applied to the inputs, and the resulting weighted sum passed to a function that produces the output o.

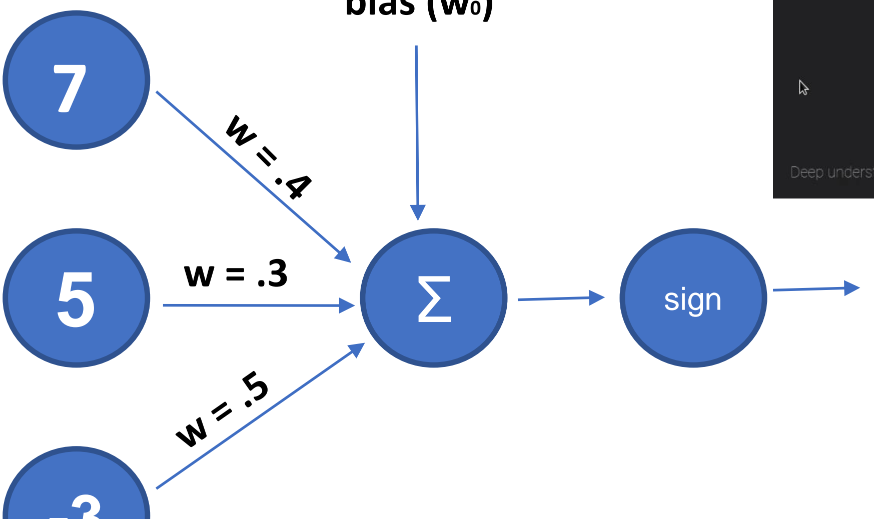
https://en.wikipedia.org/wiki/Perceptron

Perceptron



https://www.udemy.com/course/deeplearning_x/

https://en.wikipedia.org/wiki/Perceptron

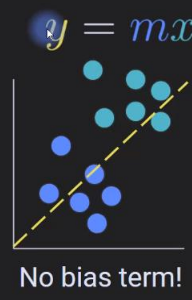


https://www.udemy.com/course/deeplearning_x/

Do you need the bias term?

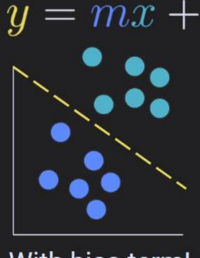
Goal: Separate the two colors

$y = mx$



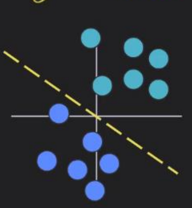
No bias term!

$y = mx + b$

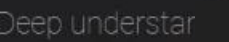


With bias term!

$y = mx$

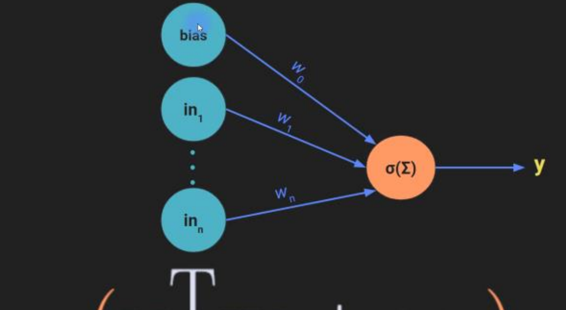


No bias term
Mean-centered data



Deep understar

The full perceptron model



$$\sigma(\mathbf{x}^T\mathbf{w} + w_0) = \hat{\mathbf{y}}$$

Deep understanding of deep learning — MX Cohen — sincxpress.com

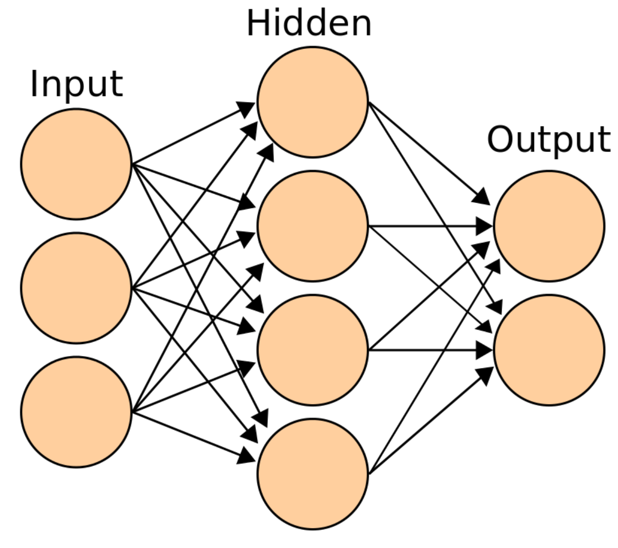
Multi Layer Perceptron

Training Process (forward/backward propagation):

1. Guess some solution

2. Compute the error of this solution

3. Modify the parameters based on this error – We need to find the minimum of the error "landscape" - Gradient Descent

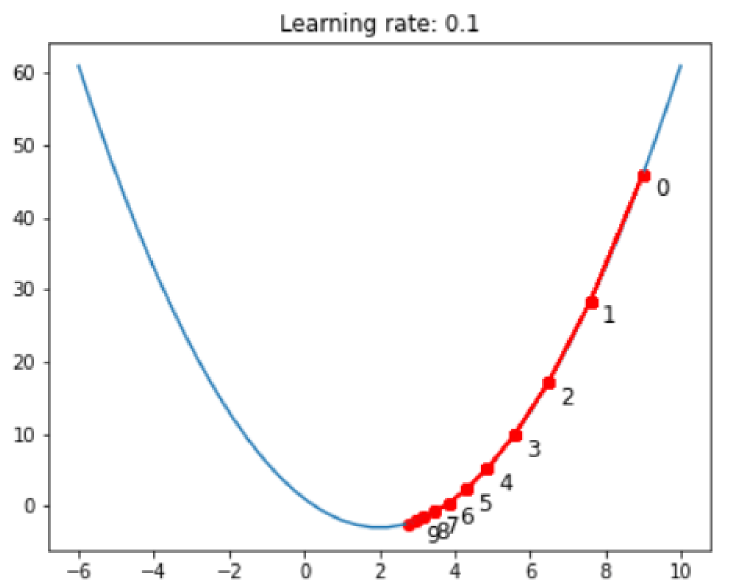


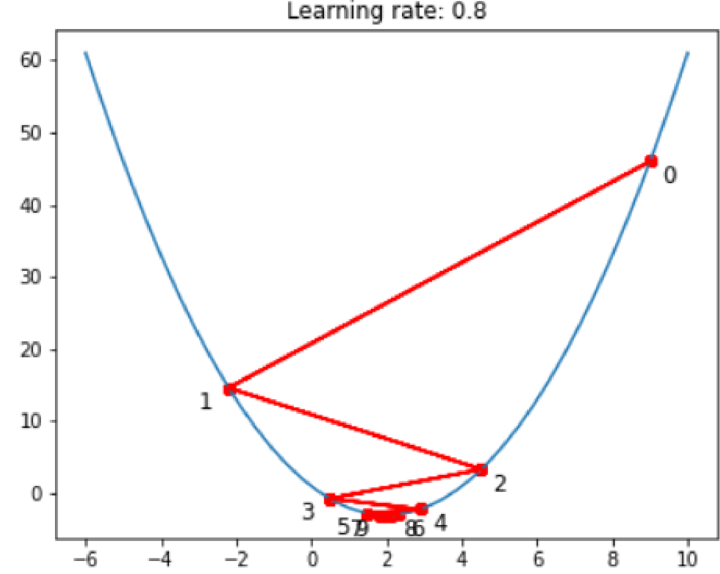
https://en.wikipedia.org/wiki/Perceptron

https://upload.wikimedia.org/wikipedia/commons/4/46/Colored_neural_network.svg

Gradient Descent

1. Random initialization of starting point
2. Based on gradient (derivation) decided the step way
3. Learning rate – scaling factor for step sizes
4. Repeat the process (number of iterations/epochs)

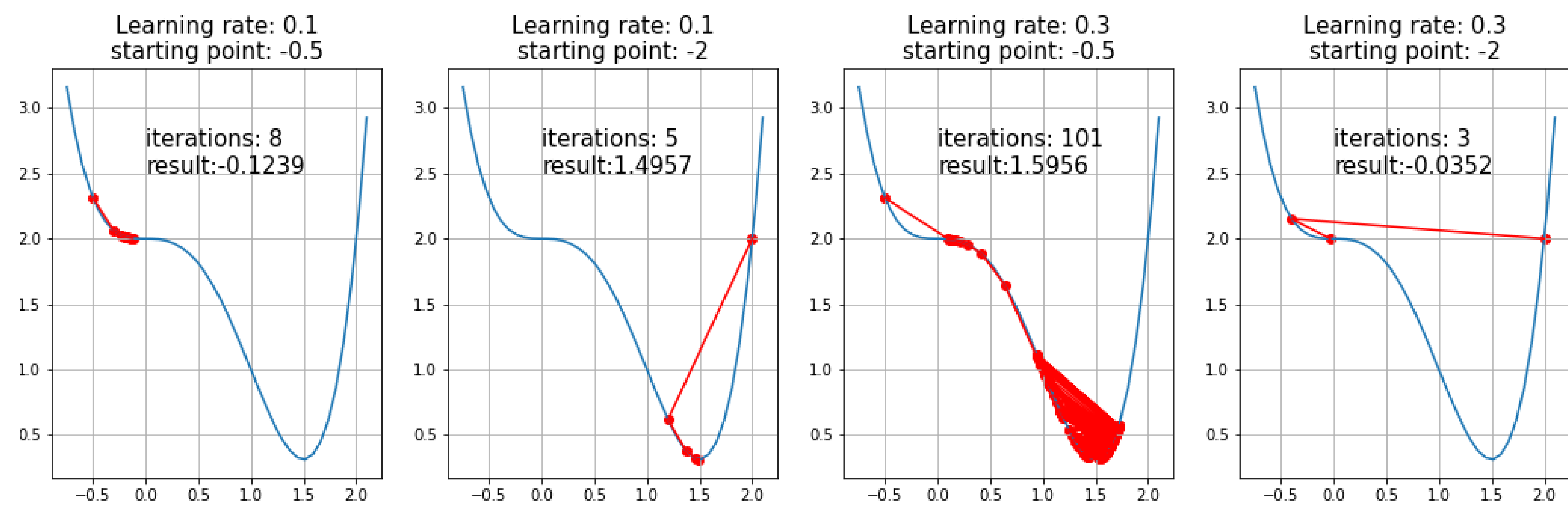




https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21

Gradient Descent

Problems?



https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21

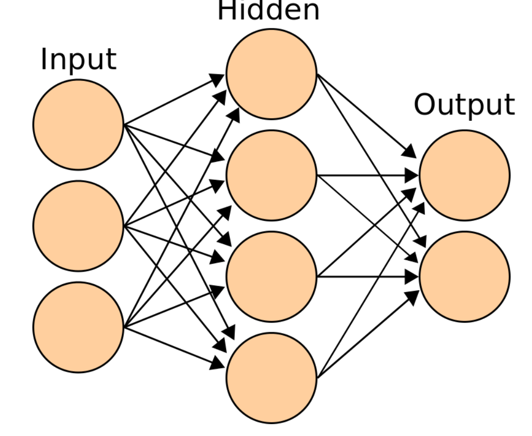
Gradient Descent

Python Example

```
import numpy as np
import matplotlib.pyplot as plt

def fx(x):
    return x**2 - 6*x + 1

def deriv(x):
    return ???

x = np.linspace(-14, 20, 2000)

localmin = 18 #np.random.choice(x,1)
grad = deriv(localmin)
print("localmin", localmin)
print("grad", grad)
plt.plot(x, fx(x))
plt.plot(localmin, fx(localmin), 'ro')
plt.show()
```

```
learning_rate = 0.05
training_epochs = 1000

for i in range(training_epochs):
    grad = deriv(localmin)
    move = learning_rate*grad
    localmin = localmin - move
print(localmin)
```


https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21



convolutional neural network (CNN):

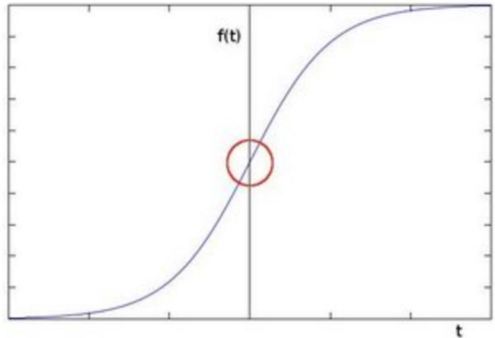
CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION

Edge Detection

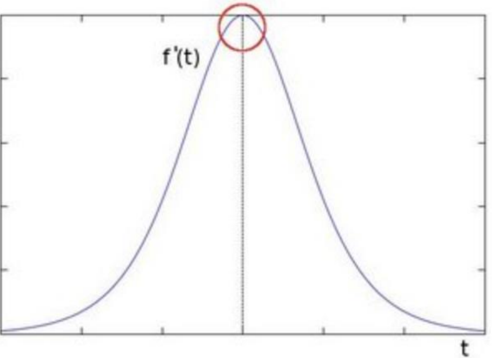
- Edge detection is the process for finding structure and properties of the object (i.e. edges in an image)
- Edges are one of the most important features
- Edges can be described by sudden changes in pixel intensity
- Locations with extreme differences in brightness of pixels indicate an edge
- We need to examined changes in the neighbouring pixels
- We can use operation that is called **convolution**
    - We need input image and kernel
    - Multiply the image pixels by pixels of the filter, then sum the results

Edge Detection

To be more graphical, let's assume we have a 1D-image. An edge is shown by the "jump" in intensity in the plot below:



The edge "jump" can be seen more easily if we take the first derivative (actually, here appears as a maximum)



https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html

Edge Detection

Assuming that the image to be operated is I:

1. We calculate two derivatives:

   a. **Horizontal changes**: This is computed by convolving I with a kernel G_x with odd size. For example for a kernel size of 3, G_x would be computed as:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

   a. **Vertical changes**: This is computed by convolving I with a kernel G_y with odd size. For example for a kernel size of 3, G_y would be computed as:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

2. At each point of the image we calculate an approximation of the *gradient* in that point by combining both results above:

$$G = \sqrt{G_x^2 + G_y^2}$$

Although sometimes the following simpler equation is used:

$$G = |G_x| + |G_y|$$

https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html

Edge Detection

Simple (naive) explanation of convolution steps:

1. Center of the kernel is positioned over a specific pixel in an input image.
2. Each element in the kernel is multiplied with the corresponding pixel element in the input image.
3. Sum the result of multiplications
4. This result can be stored in our new image (edge map)

100	100	200	200
100	100	200	200
100	100	200	200
100	100	200	200

-1	0	1
-2	0	2
-1	0	1

-100
-200
-100
200
400
+200
=400

Kernel Convolution: The bigger the value at the end, the more noticeable the edge will be.

https://www.projectrhea.org/rhea/index.php/An_Implementation_of_Sobel_Edge_Detection
https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411

Edge Detection OpenCV

```
1  import cv2 as cv
2  import numpy as np
3
4  img_input = cv.imread('image.jpg', 0)
5
6  dx_kernel = np.array([[-1, 0, 1]])
7  dy_kernel = dx_kernel.T
8
9  dx_img = cv.filter2D(img_input, cv.CV_32FC1, dx_kernel)
10 dy_img = cv.filter2D(img_input, cv.CV_32FC1, dy_kernel)
11
12 #this formula or abs
13 #G = (dx_img**2 + dy_img**2)**0.5
14 abs_x = np.absolute(dx_img)
15 abs_y = np.absolute(dy_img)
16
17 G = abs_x + abs_y
18
19 cv.imshow("img_input", img_input)
20 cv.imshow("abs_x", np.uint8(abs_x))
21 cv.imshow("abs_y", np.uint8(abs_y))
22 cv.imshow("G", np.uint8(G))
23 cv.waitKey()
```

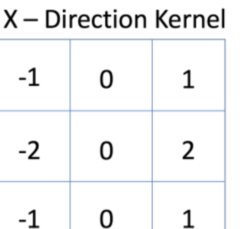
$$G = \sqrt{G_x^2 + G_y^2}$$

$$G = |G_x| + |G_y|$$

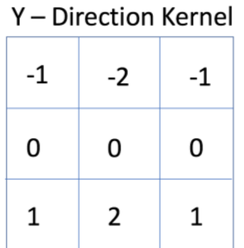
https://docs.opencv.org/4.x/d5/d0f/tutorial_py_gradients.html

Edge Detection

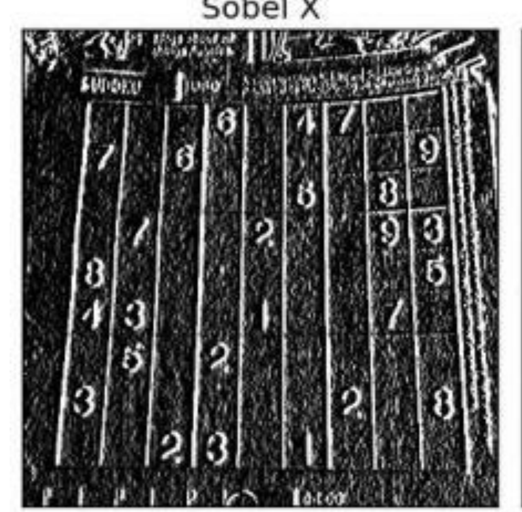
For example, if we talk about Sobel edge detection, we can use two 3x3 kernels:

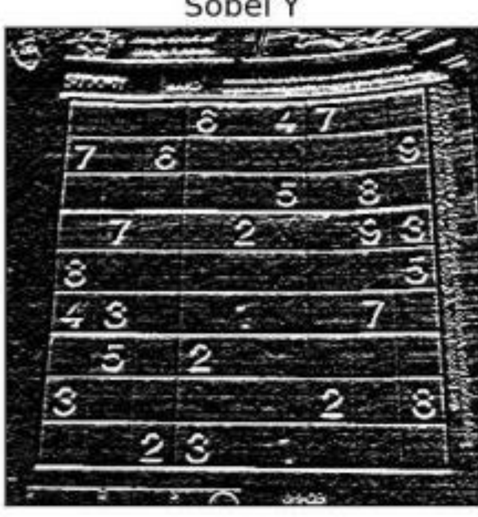

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1


Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1



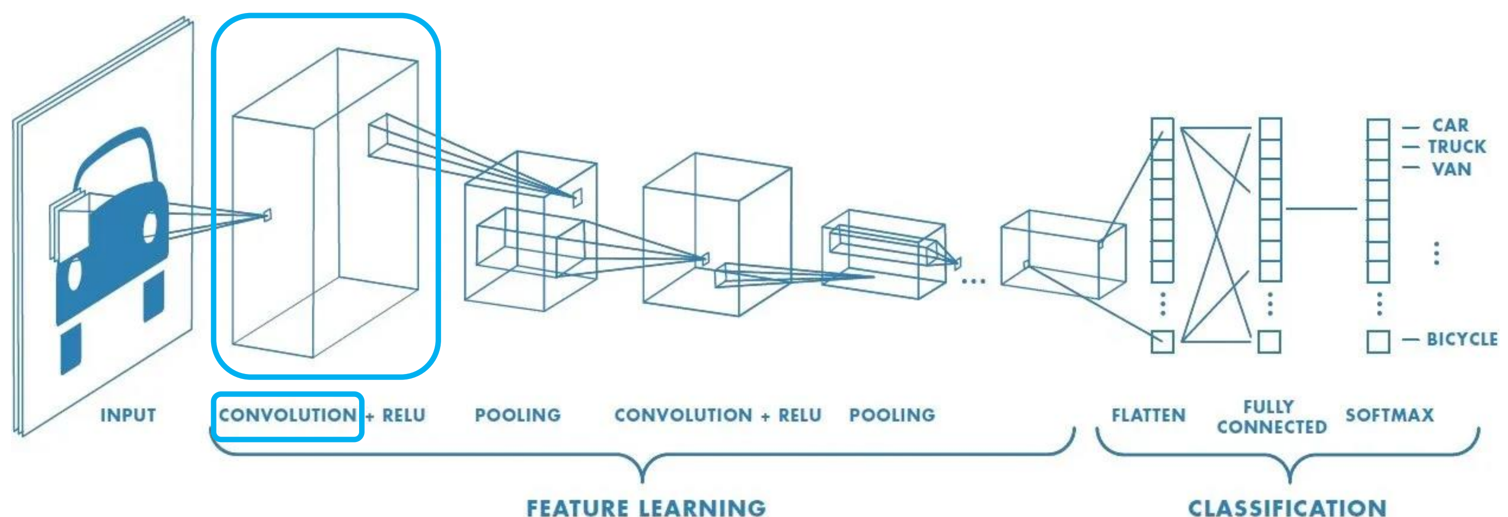


In this example, we have two edge (feature) maps (two kernels). The **convolutional** kernels are hand crafted with the help of engineers (e.g. Sobel, Prewitt, Roberts). What about **convolutional** networks? How does it work in **convolutional** networks?

https://docs.opencv.org/4.x/d5/d0f/tutorial_py_gradients.html

CNN - kernels

- In CNNs, the **kernels are used for feature extraction.**
- The size of kernels is generally small (e.g. 3x3, 5x5, 7x7).
- At the beginning of training, the kernels have **random** values.
- The **kernels are learned** during training phase (through gradient descent).
- This is the opposite process that were described in the previous slides with **hand crafted** kernels.



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

CNN – Conv - PyTorch

CONV2D PyTorch

CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

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

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

This module supports TensorFloat32.

Parameters:

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int* or *tuple*) – Size of the convolving kernel
- **stride** (*int* or *tuple*, *optional*) – Stride of the convolution. Default: 1
- **padding** (*int*, *tuple* or *str*, *optional*) – Padding added to all four sides of the input. Default: 0
- **padding_mode** (*str*, *optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int* or *tuple*, *optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int*, *optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool*, *optional*) – If `True`, adds a learnable bias to the output. Default: `True`

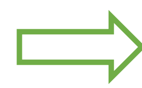
https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

CNN – Conv - PyTorch

```
1  import torch
2  import cv2 as cv
3
4  torch_conv = torch.nn.Conv2d(
5                  in_channels  = 1,
6                  out_channels = 6,
7                  kernel_size  = (3, 3),
8                  stride       = 1,
9                  padding      = 1,
10                 padding_mode = 'zeros')
11
12 print("torch_conv", torch_conv, torch_conv.weight)
13
14 torch_rand_image = torch.randn(1, 1, 10, 10)
15
16 output = torch_conv(torch_rand_image)
17 print("output", output, output.shape)
```

Parameters:

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int, tuple or str, optional*) – Padding added to all four sides of the input. Default: 0
- **padding_mode** (*str, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`



$$(N, C_{\text{in}}, H, W)$$

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

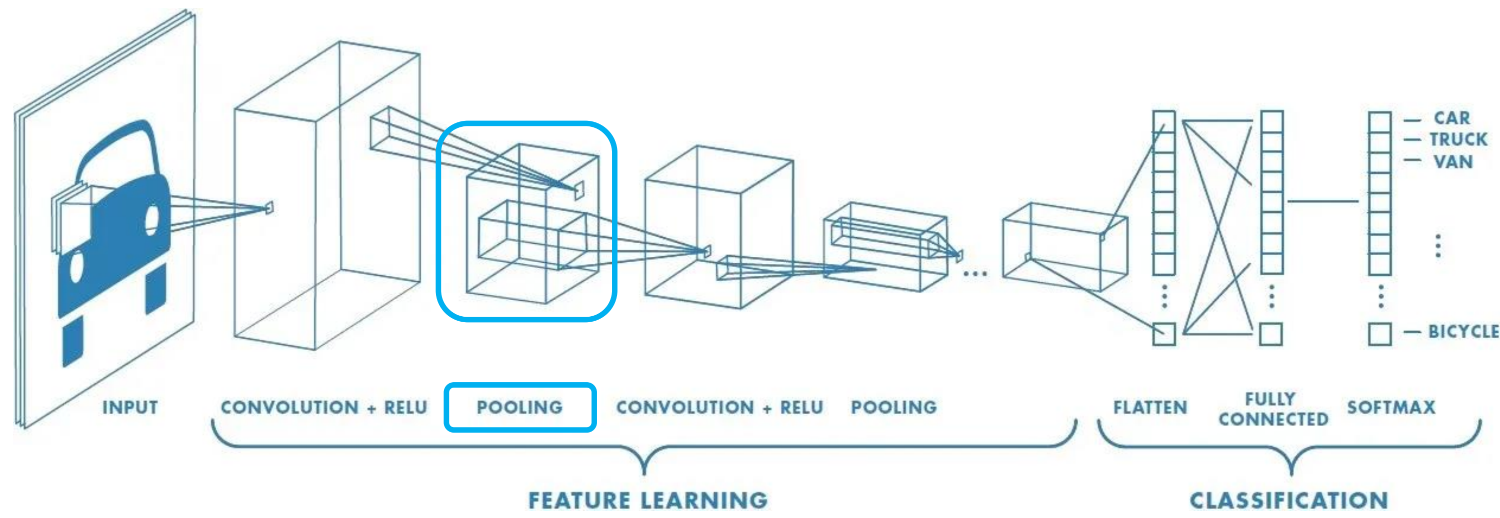
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$$

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

Pooling

- The goal of this step is to reduce the dimensionality of each feature map but preserve important informations
- Operations: e.g. Sum, Average, Max



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Pooling - PyTorch

AVGPOOL2D

CLASS torch.nn.AvgPool2d(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*) [SOURCE]

Applies a 2D average pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

MAXPOOL2D

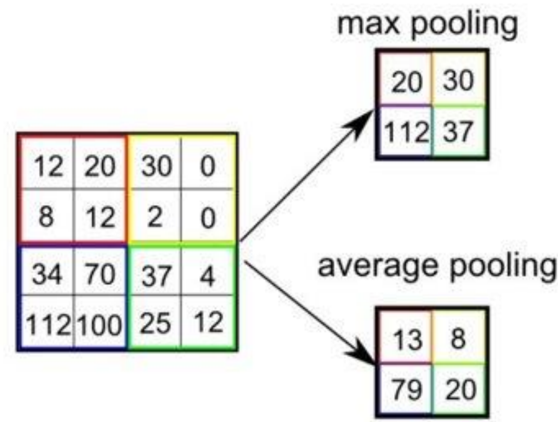
CLASS torch.nn.MaxPool2d(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*) [SOURCE]

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

Common way is a pooling layer with filters of size 2x2 applied with a stride of 2



Mean: smooths images

Max: highlights sharp features

https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d

Pooling- PyTorch

```
1  import torch
2
3  my_pool = torch.nn.MaxPool2d(kernel_size = 2, stride=2, padding=0)
4  print("my_pool", my_pool)
5
6  torch_rand_image = torch.randn(1, 1, 6, 6)
7  print("torch_rand_image", torch_rand_image, torch_rand_image.shape)
8
9  output = my_pool(torch_rand_image)
10 print("output", output, output.shape)
```

https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d

Simple CNN - PyTorch

```
import torch

first_simple_cnn = torch.nn.Sequential(
                    #input shape ([1, 1, 32, 32])
                    torch.nn.Conv2d(1, 6, (3,3), 1, 1),
                    #shape after Conv2d ([1, 6, 32, 32])
                    torch.nn.ReLU(),
                    torch.nn.AvgPool2d(2, 2),
                    #shape after AvgPool2d ([1, 6, 16, 16])

                    torch.nn.Flatten(),
                    #we have 16*16*6 features
                    torch.nn.Linear(16*16*6, 1),
                    torch.nn.Sigmoid() )

print(first_simple_cnn)
torch_rand_image = torch.randn(1, 1, 32, 32)
output = first_simple_cnn(torch_rand_image)
print("output", output)
```

CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION

PyTorch Sequential vs. Module

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 6, (3,3), 1, 1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(16*16*6, 1)

    def forward(self, x):
        x = self.pool(F.relu(self.conv(x)))
        print("x.shape", x.shape)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        print("x.shape", x.shape)
        x = self.fc(x)
        x = torch.sigmoid(x)
        return x

model = Net()
print(model)
torch_rand_image = torch.randn(1, 1, 32, 32)
output = model(torch_rand_image)
print("output", output)
```

```python
import torch

first_simple_cnn = torch.nn.Sequential(
                #input shape ([1, 1, 32, 32])
                torch.nn.Conv2d(1, 6, (3,3), 1, 1),
                #shape after Conv2d ([1, 6, 32, 32])
                torch.nn.ReLU(),
                torch.nn.AvgPool2d(2, 2),
                #shape after AvgPool2d ([1, 6, 16, 16])

                torch.nn.Flatten(),
                #we have 16*16*6 features
                torch.nn.Linear(16*16*6, 1),
                torch.nn.Sigmoid() )

print(first_simple_cnn)
torch_rand_image = torch.randn(1, 1, 32, 32)
output = first_simple_cnn(torch_rand_image)
print("output", output)
```

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

PyTorch – Example - 1

```
import torch
import torchvision
import torchvision.transforms as transforms
import cv2
import torch.nn as nn
import torch.optim as optim
import glob

BATCH_SIZE = 8
EPOCHS = 16
LR = 0.001

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
     ])

trainset = torchvision.datasets.ImageFolder(root='train_simple', transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=2)
dataset_size = len(trainloader.dataset)
print(trainset.classes)
classes = trainset.classes
print(dataset_size)
```

Hyperparameters

Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates (read more about hyperparameter tuning)

We define the following hyperparameters for training:

- **Number of Epochs** - the number times to iterate over the dataset
- **Batch Size** - the number of data samples propagated through the network before the parameters are updated
- **Learning Rate** - how much to update models parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training.

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 2

```
import torch
import torchvision
import torchvision.transforms as transforms
import cv2
import torch.nn as nn
import torch.optim as optim
import glob

BATCH_SIZE = 8
EPOCHS = 16
LR = 0.001

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
     ])

trainset = torchvision.datasets.ImageFolder(root='train_simple', transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=2)
dataset_size = len(trainloader.dataset)
print(trainset.classes)
classes = trainset.classes
print(dataset_size)
```

A generic data loader where the images are arranged in this way by default:

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/[...]/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/[...]/asd932_.png
```

https://pytorch.org/vision/main/generated/torchvision.datasets.ImageFolder.html

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 3

```
26  model = nn.Sequential(
27              nn.Conv2d(3,8,5),
28              nn.ReLU(),
29              nn.Conv2d(8,16,5),
30              nn.ReLU(),
31              nn.Flatten(),
32              nn.LazyLinear(120),
33              nn.ReLU(),
34              nn.LazyLinear(84),
35              nn.ReLU(),
36              nn.Linear(84, 2))
37
38  print(model)
```

3 - channel images
2 - classes

LAZYLINEAR

CLASS torch.nn.LazyLinear(*out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

A `torch.nn.Linear` module where *in_features* is inferred.

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

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 4

```
40 loss_fun = nn.CrossEntropyLoss()
41 optimizer = optim.SGD(model.parameters(), lr=LR)
```

Loss Function

When presented with some training data, our untrained network is likely not to give the correct answer. **Loss function** measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training. To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value.

Common loss functions include nn.MSELoss (Mean Square Error) for regression tasks, and nn.NLLLoss (Negative Log Likelihood) for classification. nn.CrossEntropyLoss combines `nn.LogSoftmax` and `nn.NLLLoss`.

We pass our model’s output logits to `nn.CrossEntropyLoss`, which will normalize the logits and compute the prediction error.

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 5

```
40 loss_fun = nn.CrossEntropyLoss()
41 optimizer = optim.SGD(model.parameters(), lr=LR)
```

Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed (in this example we use Stochastic Gradient Descent). All optimization logic is encapsulated in the `optimizer` object. Here, we use the SGD optimizer; additionally, there are many different optimizers available in PyTorch such as ADAM and RMSProp, that work better for different kinds of models and data.

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 6

```
43 for epoch in range(EPOCHS):
44     print("epoch", epoch+1)
45 
46     running_loss = 0.0
47     for i, data in enumerate(trainloader, 0):
48         inputs, labels = data
49         #print(inputs.shape)
50         #print(labels.shape)
51         optimizer.zero_grad()
52         outputs = model(inputs)
53         loss = loss_fun(outputs, labels)
54         loss.backward()
55         optimizer.step()
56         running_loss += loss.item()
57         if i % 20 == 19:
58             print(f'minibatch: {i+1} loss: {running_loss / 20}')
59             running_loss = 0
60 
61 torch.save(model.state_dict(), "net.pth")
62 print('Finished Training')
```

Inside the training loop, optimization happens in three steps:

- Call `optimizer.zero_grad()` to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to `loss.backward()`. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call `optimizer.step()` to adjust the parameters by the gradients collected in the backward pass.

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 7

```
13  transform = transforms.Compose(
14      [transforms.ToTensor(),
15       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
16       ])
```

```
64  test_images = [img for img in glob.glob("test_images/*.png")]
65  for test_img in test_images:
66      image = cv2.imread(test_img, 1)
67      tensor = transform(image)
68      print(tensor.shape)
69      #add batch size
70      tensor = tensor.unsqueeze(0)
71      outputs = model(tensor)
72      _, predicted = torch.max(outputs, 1)
73      print("predicted", predicted)
74      print("final", classes[predicted])
75
76      cv2.imshow("image", image)
77      cv2.waitKey(0)
```

Load images from folder
The same transformations as in training
Convert OpenCV numpy > PyTorch tensor
Prediction

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

PyTorch – Example - 8

SAVE AND LOAD THE MODEL

In this section we will look at how to persist model state with saving, loading and running model predictions.

```
import torch
import torchvision.models as models
```

Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called `state_dict`. These can be persisted via the `torch.save` method:

```
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using `load_state_dict()` method.

```
model = models.vgg16() # we do not specify weights, i.e. create untrained model
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```

• NOTE

be sure to call `model.eval()` method before inferencing to set the dropout and batch normalization layers to evaluation mode. Failing to do this will yield inconsistent inference results.

https://pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html

PyTorch – Example - 9

```
transform = torchvision.transforms.Compose([
                            torchvision.transforms.ToPILImage(),
                            torchvision.transforms.Resize(size=(40, 40)),
                            torchvision.transforms.ColorJitter(brightness=.5, hue=.3),
                            torchvision.transforms.RandomRotation(degrees=(0, 5)),
                            torchvision.transforms.GaussianBlur(kernel_size=(5, 5), sigma=(0.1, 5)),
                            torchvision.transforms.RandomAdjustSharpness(sharpness_factor=2),
                            torchvision.transforms.ToTensor(),
                            torchvision.transforms.Normalize((0.5,), (0.5,))
                          ])
```

Random transforms

The following transforms are random, which means that the same transfomer instance will produce different result each time it transforms a given image.

ColorJitter

The `ColorJitter` transform randomly changes the brightness, saturation, and other properties of an image.

```
jitter = T.ColorJitter(brightness=.5, hue=.3)
jitted_imgs = [jitter(orig_img) for _ in range(4)]
plot(jitted_imgs)
```

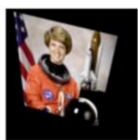
Original image

RandomPerspective

The `RandomPerspective` transform (see also `perspective()`) performs random perspective transform on an image.

```
perspective_transformer = T.RandomPerspective(distortion_scale=0.6, p=1.0)
perspective_imgs = [perspective_transformer(orig_img) for _ in range(4)]
plot(perspective_imgs)
```

Original image



https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py

https://pytorch.org/vision/0.9/transforms.html#torchvision.transforms.Compose