LAB 4: CONVOLUTIONAL NEURAL NETWORKS (CNNs)

University of Washington, Seattle

Fall 2024

OUTLINE

Part 1: Introduction to CNNs

- Why do we need CNNs?
- Convolutional Layers
- Pooling Layers

Part 2: Applications of CNNs

- Image Segmentation
- Visual Recognition

Part 3: CNN Implementation in PyTorch

- MNIST Classification Example

Lab Assignment

- Fashion MNIST Dataset
- Fashion MNIST Classification with CNN

INTRODUCTION TO CNNs

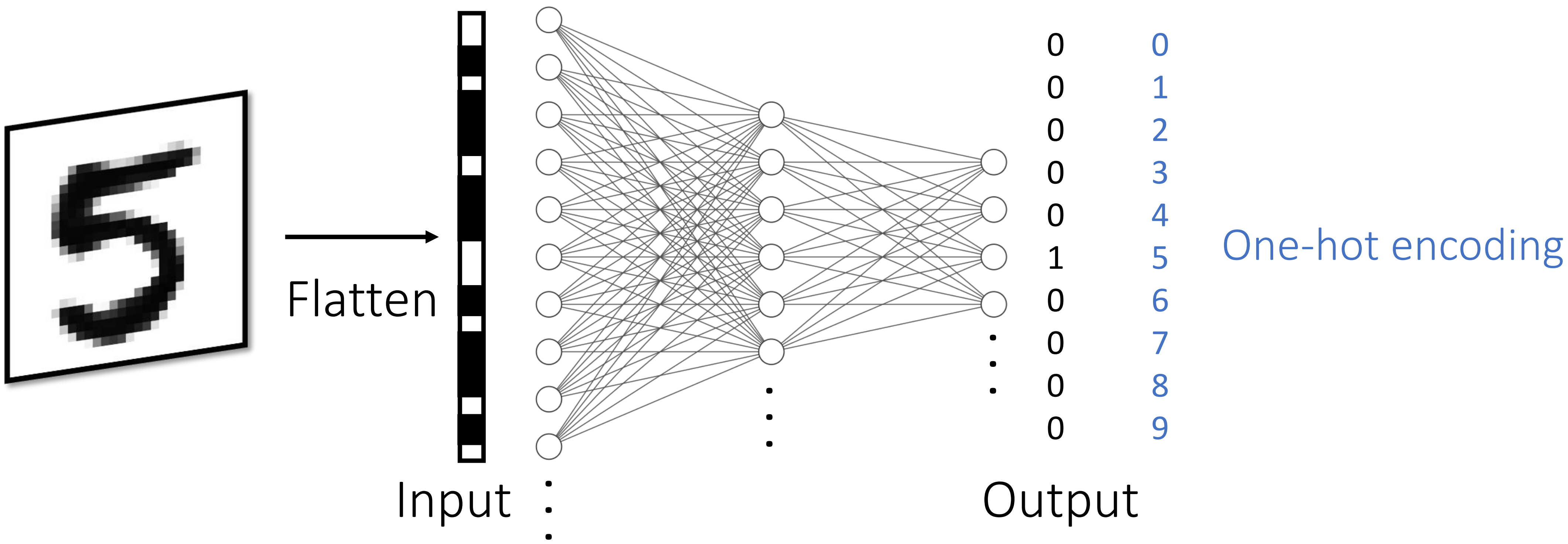
Why do we need CNNs?

Convolutional Layers

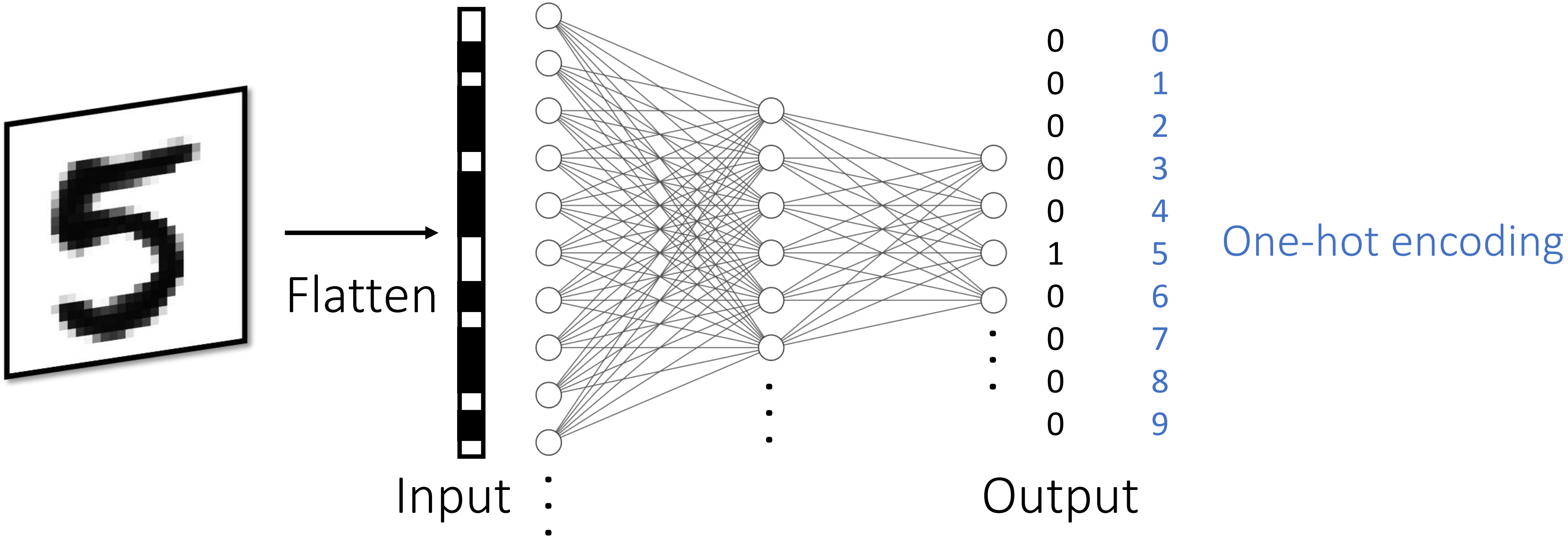
Pooling Layers

Why do we need CNNs?

FCN for Image Classification (Lab 3)



FCN for Image Classification (Lab 3)

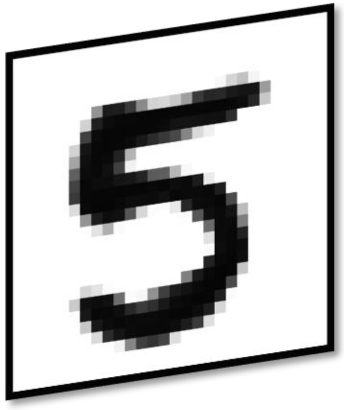


Great at Classification

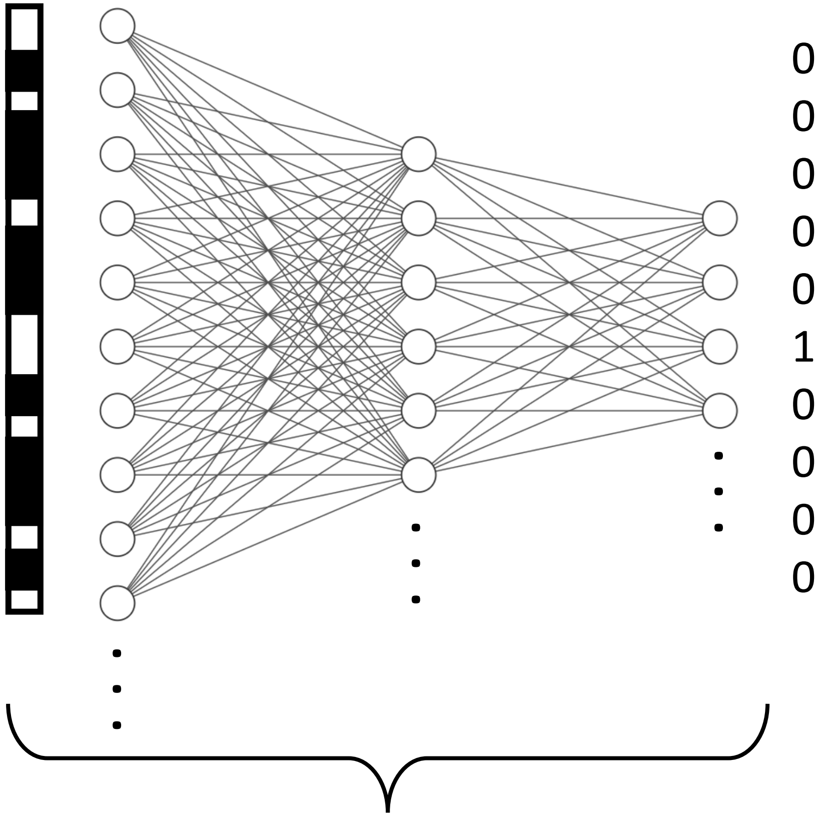
Not as good with Extracting image features

Too many parameters when Flattening images

Specialized Layers for Feature Extractions

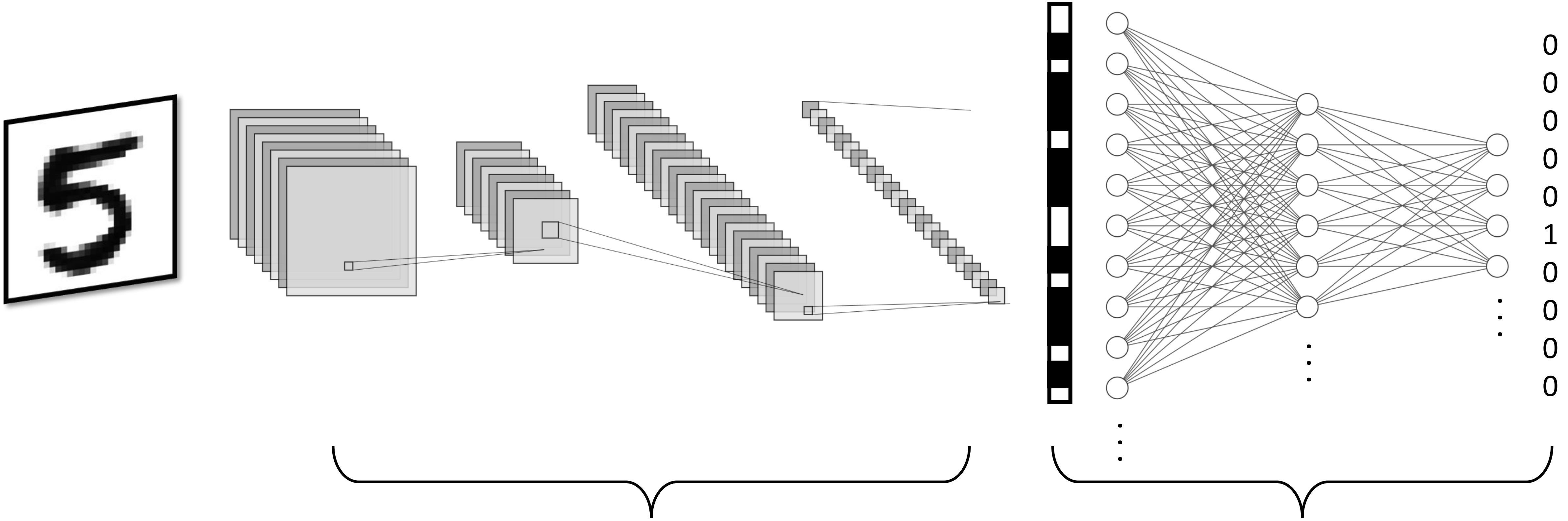


Specialized Layers for
Image Feature Extraction



Fully connected layers
(Classifier)

Full CNN Architecture

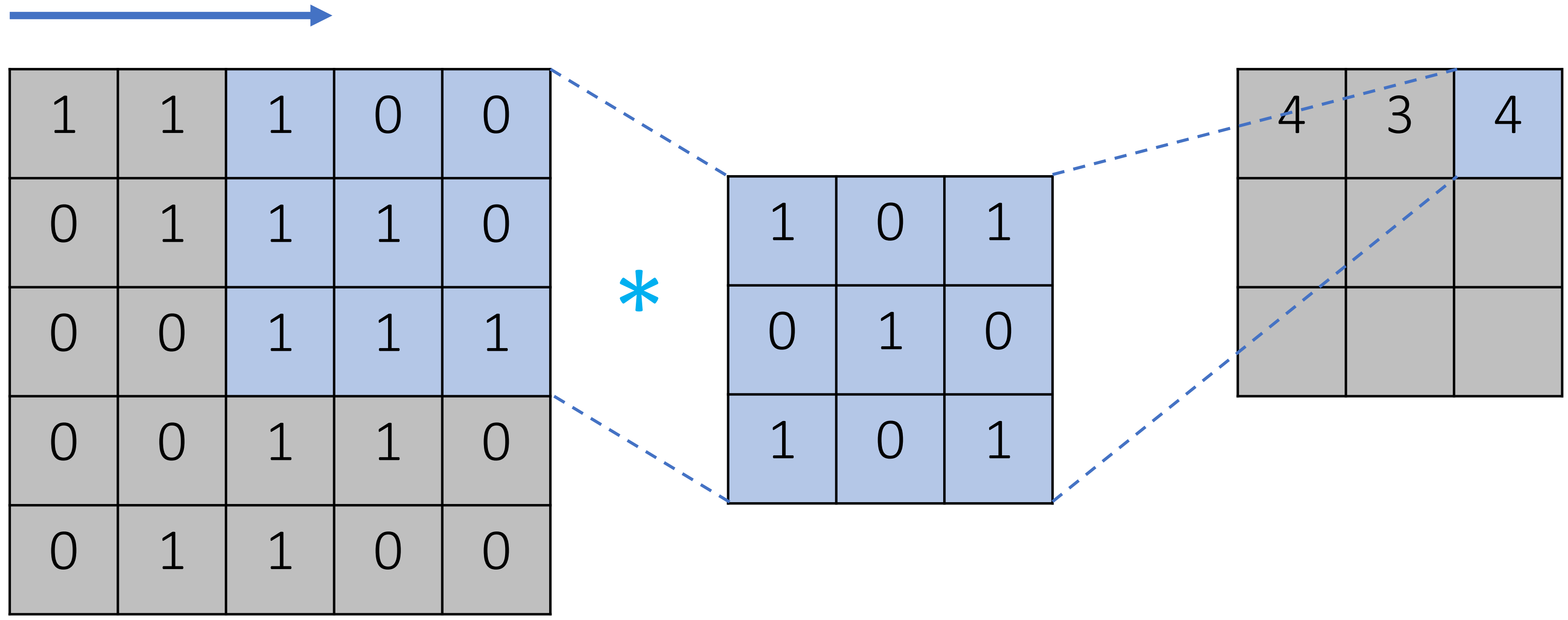


Convolution Layers + Pooling Layers
(Image feature extraction)

Fully connected layers
(Classifier)

Convolutional Layers

Image Convolution



Input Image

Filter

Convoluted Feature

$$(1*1) + (0*0) + (0*1) +$$
$$(0*1) + (1*1) + (0*0) +$$
$$(1*1) + (0*0) + (1*1) +$$

Stride

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image

*

1	0	1
0	1	0
1	0	1

Filter

4		

Convoluted Feature

Stride = 1

Stride

Input Image

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

*

Filter

1	0	1
0	1	0
1	0	1

Convoluted Feature

4	3	

Stride = 1

Stride

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image

*

1	0	1
0	1	0
1	0	1

Filter

4	3	4

Convoluted Feature

Stride = 1

Stride

Input = 5 x 5

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image

*

1	0	1
0	1	0
1	0	1

Filter

Output = 3 x 3

4	3	4
3		

Convoluted Feature

Stride = 1

Stride

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image

*

1	0	1
0	1	0
1	0	1

Filter

4	

Convoluted Feature

Stride = 2

Stride

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image

*

1	0	1
0	1	0
1	0	1

Filter

4	4

Convoluted Feature

Stride = 2

Stride

Input = 5 x 5

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image

*

1	0	1
0	1	0
1	0	1

Filter

Output = 2 x 2

4	4
2	

Convoluted Feature

Stride = 2

Padding

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image
(5x5)

Padding = 1

0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	0	0	0	0	0

Padded Image
(7x7)

Padding

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input Image
(5x5)

Padding = 2

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	1	0	0
0	0	0	0	1	1	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Padded Image
(9x9)

Convolution Layer in PyTorch

torch.nn.Conv2d(	Parameter description	Data type
- in_channels	# of channels of input	int
- out_channels	# of channels of output	int
- kernel_size	Size of the convolving Filter	int or tuple
- stride	Stride of the convolution	int or tuple (default = 1)
- padding	Padding added to input	int, tuple or str (default = 0)
)		

Convolution Layer in PyTorch

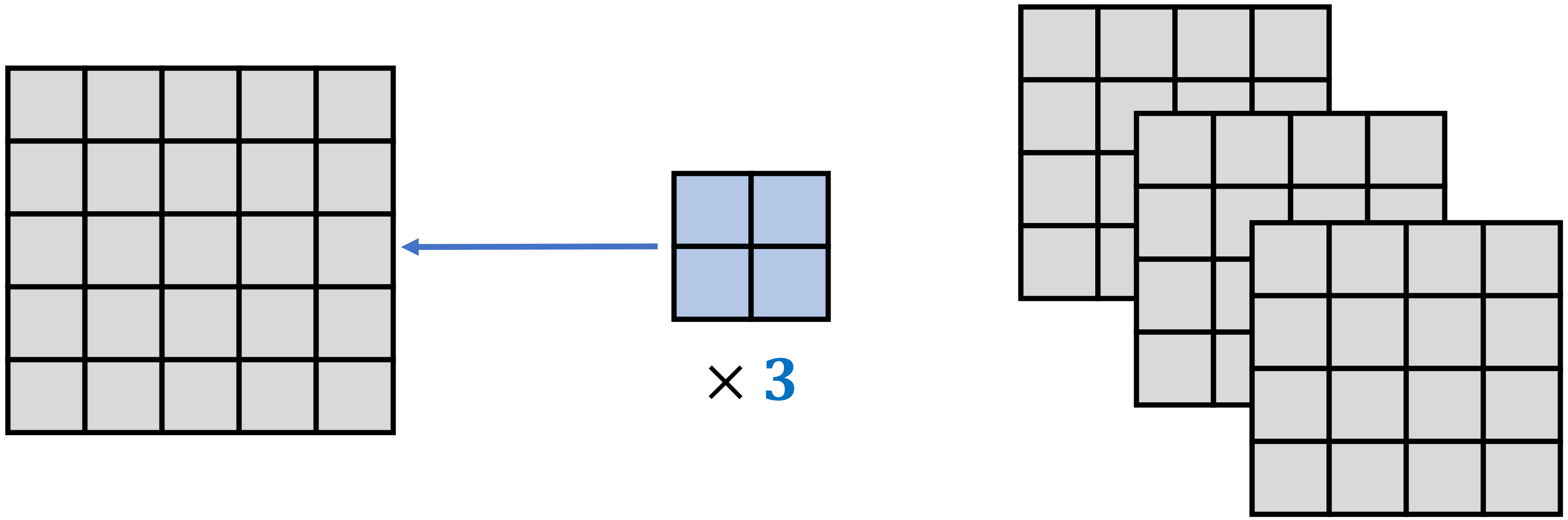
torch.nn.Conv2d(

in_channels: 1

out_channels: 3

kernel_size: 2

)



Input
(1x1x5x5)

Filters
(3x1x2x2)

Output
(1x3x4x4)

(N x Channels x Height x Width)

Convolution Layer in PyTorch

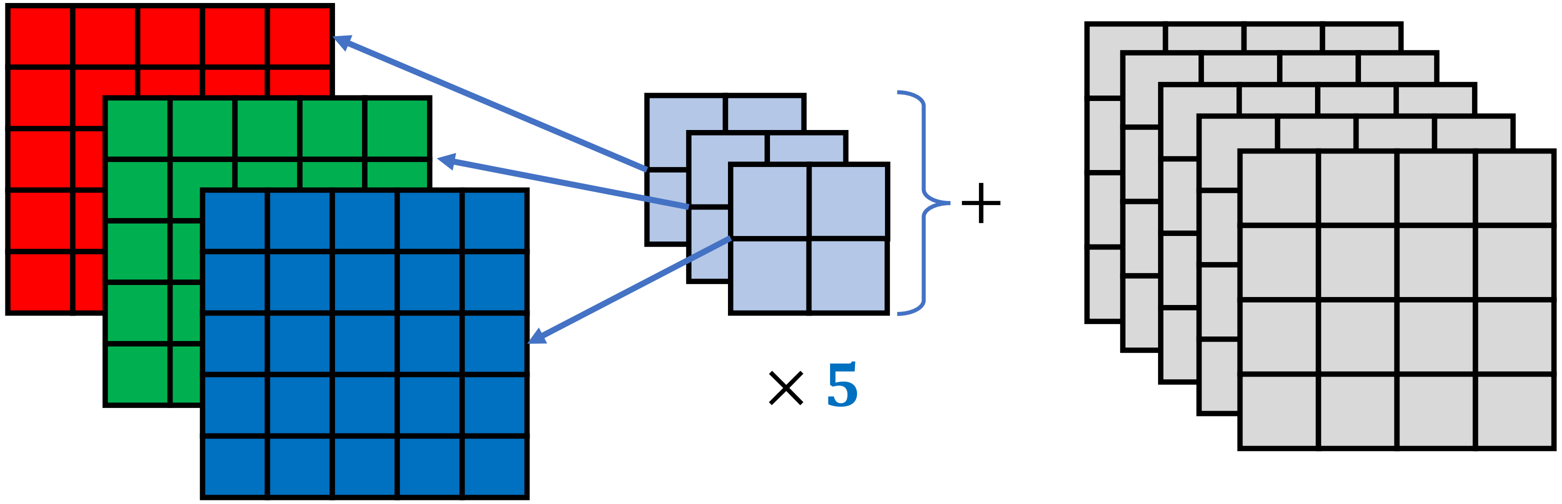
torch.nn.Conv2d(

in_channels: 3

out_channels: 5

kernel_size: 2

)



Input
(1x3x5x5)

Filters
(5x3x2x2)

Output
(1x5x4x4)

(N x Channels x Height x Width)

More detailed explanation: https://d2l.ai/chapter_convolutional-neural-networks/channels.html

Pooling Layers

Max Pooling and Average Pooling

4	9	2	5
5	6	2	4
2	4	5	4
5	6	8	4

9	5
6	8

```
torch.nn.MaxPool2D(
kernel_size = 2,
stride = 2
)
```

4	9	2	5
5	6	2	4
2	4	5	4
5	6	8	4

6.0	3.3
4.3	5.3

```
torch.nn.AvgPool2D(
kernel_size = 2,
stride = 2
)
```

APPLICATIONS OF CNNs

Image Segmentation

Visual Recognition

Image Segmentation

segmentation →

Human
Bike

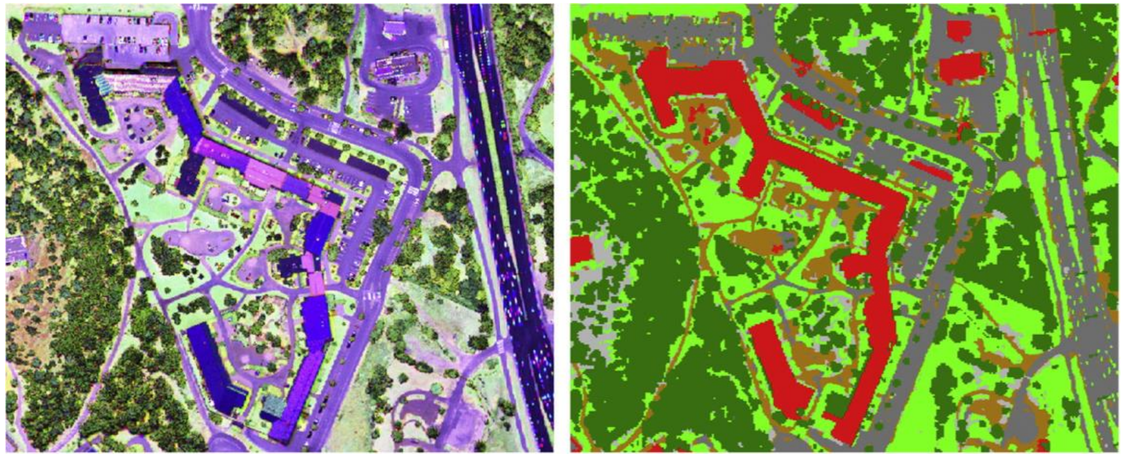
Assigns each pixel into a class

Image credit:
https://learnopencv.com/pytorch-for-beginners-semantic-segmentation-using-torchvision/

Image Segmentation Applications

Facial Segmentation

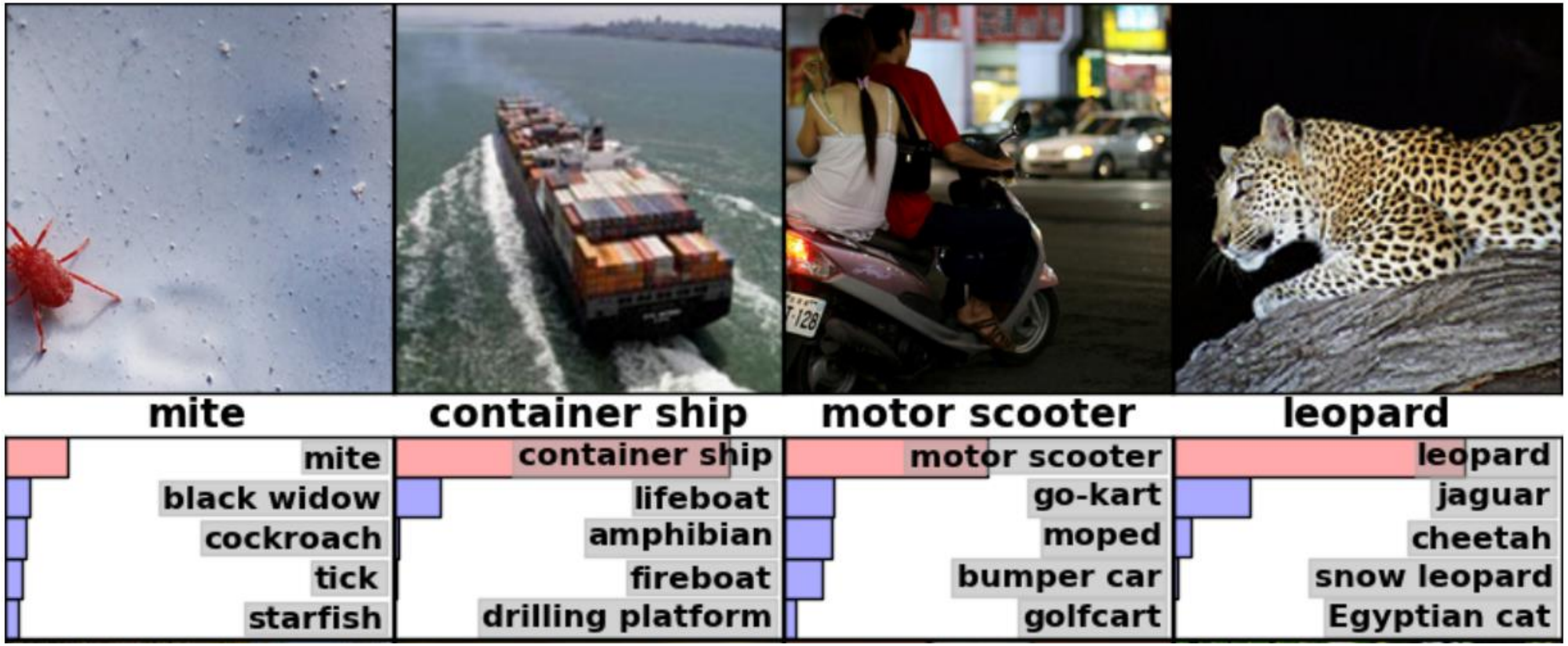
Autonomous Driving



Geo Land Sensing

Image credit: https://learnopencv.com/pytorch-for-beginners-semantic-segmentation-using-torchvision/

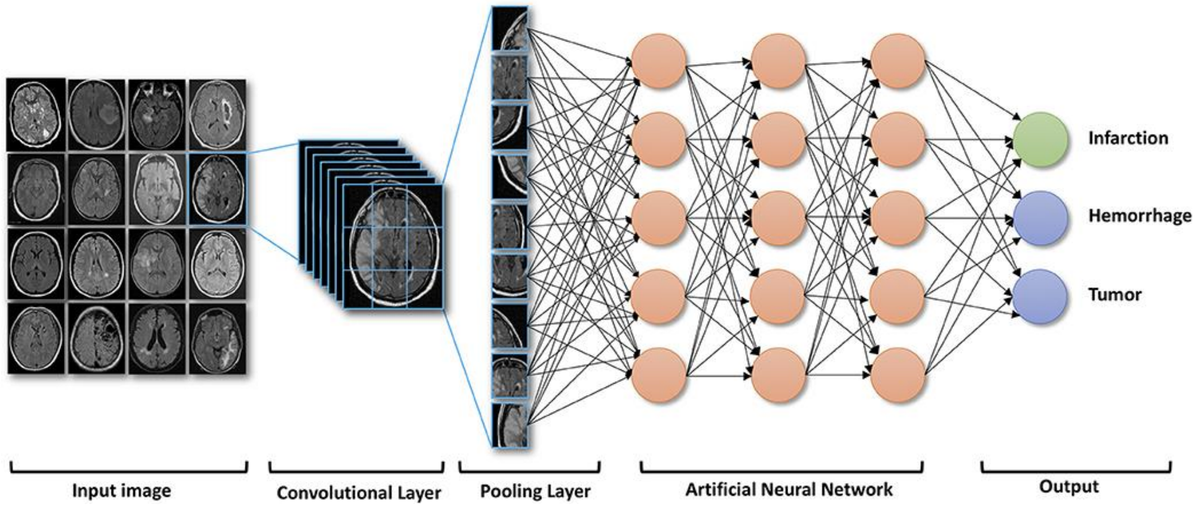
Visual Recognition Task



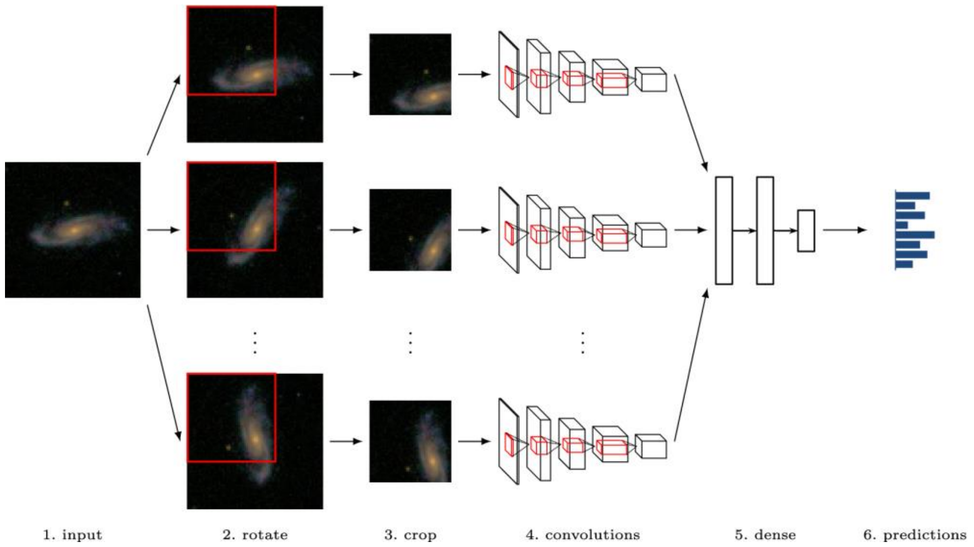
Assigns a whole image into a class

Image credit:
https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

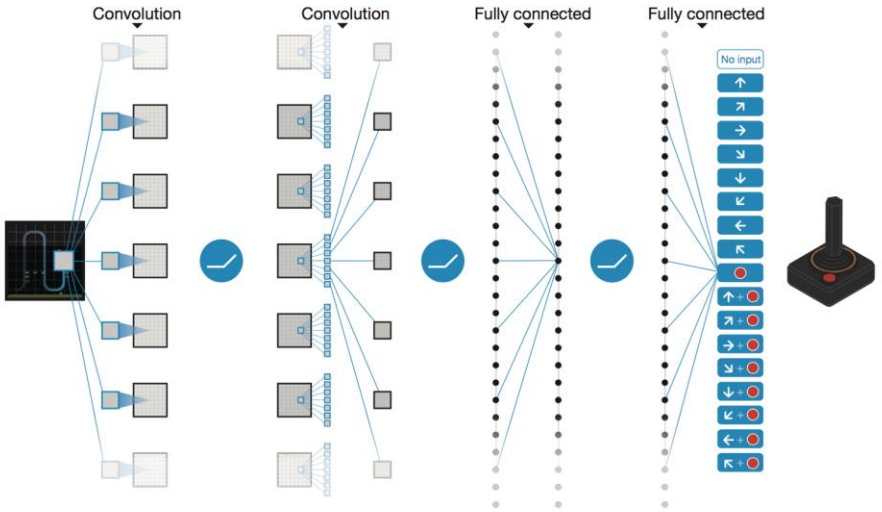
Visual Recognition Applications



Medical Diagnosis from Images



Astronomical Image Analysis



AI Controlled Gaming

Image credits:
https://www.frontiersin.org/articles/10.3389/fneur.2019.00869/full
https://jwuphysics.github.io/blog/galaxies/astrophysics/
https://www.nature.com/articles/nature14236

CNNs are Leading Algorithms in Visual Recognition

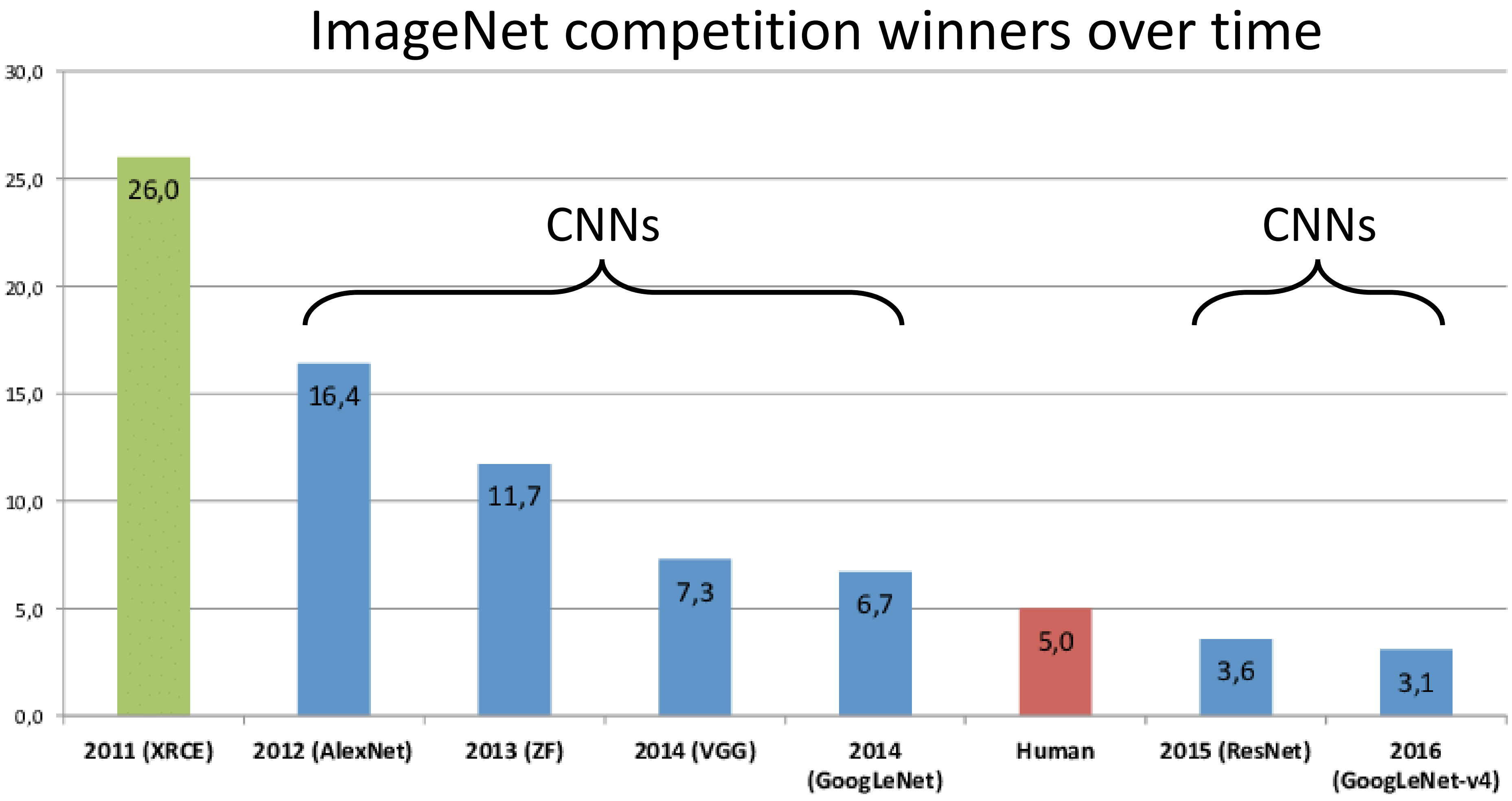


Image credit: Edge ai + Vision Alliance

CNN IMPLEMENTATION IN PYTORCH

MNIST Classification Example

Prepare Data

```
from sklearn.preprocessing import StandardScaler

train_features = np.load('mnist_train_features.npy')
train_targets = np.load('mnist_train_targets.npy')

test_features = np.load('mnist_test_features.npy')
test_targets = np.load('mnist_test_targets.npy')

train_features_flat = train_features.reshape((1000, 28 * 28))
test_features_flat = test_features.reshape((100, 28 * 28))

scaler = StandardScaler()
train_features = scaler.fit_transform(train_features_flat).reshape((1000, 28, 28))
test_features = scaler.fit_transform(test_features_flat).reshape((100, 28, 28))
```

We will use Standard Scaler from scikit-learn

Load train and testing dataset for MNIST

Flatten the features so we can scale them as 2D array

Scale train/test features and reshape them back to 28 x 28 arrays

Prepare Data

```
validation_features = train_features[:100]
validation_targets = train_targets[:100]

train_features = train_features[100:]
train_targets = train_targets[100:]
```

Take first 100 train samples as validation set

Take the remaining samples (900) as training set

```
train_features = np.reshape(train_features, (900, 1, 28, 28))
validation_features = np.reshape(validation_features, (100, 1, 28, 28))
test_features = np.reshape(test_features, (100, 1, 28, 28))
```

Reshape Train/Validation/Testing sets according to (N, Channels, H, W) format supported by PyTorch

Define Model

```python
class CNNModel(torch.nn.Module):

    def __init__(self):

        super(CNNModel, self).__init__()

        self.cnn1 = torch.nn.Conv2d(in_channels=1, out_channels=16,
                                    kernel_size=5, stride=1, padding=2)

        self.maxpool1 = torch.nn.MaxPool2d(kernel_size=2)

        self.cnn2 = torch.nn.Conv2d(in_channels=16, out_channels=32,
                                    kernel_size=5, stride=1, padding=2)

        self.maxpool2 = torch.nn.MaxPool2d(kernel_size=2)

        self.fc1 = torch.nn.Linear(32 * 7 * 7, 10)
```

Convolution

- In-channel # : 1
- Out-channel # : 16
- Kernel size : 5
- Stride = 1
- Padding = 2

Max Pool

Kernel size : 2

Convolution

- In-channel # : 16
- Out-channel # : 32
- Kernel size : 5
- Stride = 1
- Padding = 2

Max Pool

Kernel size : 2

FCN Layer

1568 → 10 neurons

Define Model

```python

    def forward(self, x):

        conv1_out = torch.nn.functional.relu(self.cnn1(x))
        pool1_out = self.maxpool1(conv1_out)

        conv2_out = torch.nn.functional.relu(self.cnn2(pool1_out))
        pool2_out = self.maxpool2(conv2_out)

        fcn_input = pool2_out.view(pool2_out.size(0), -1)

        output = self.fc1(fcn_input)

        return output
```

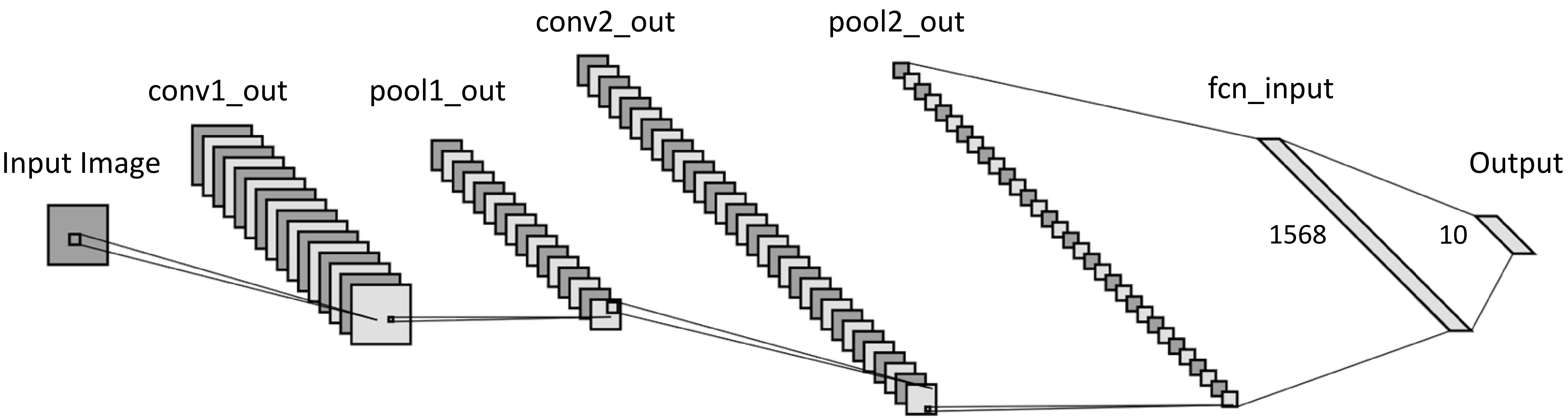
Input image → self.cnn1 → ReLU → self.maxpool1

→ self.cnn2 → ReLU → self.maxpool2

Flatten pool2_out (32, 7, 7) → 1568

Return raw output of fc1

Define Model



self.cnn1
- In-channel # : 1
- Out-channel # : 16
- Kernel size : 5
- Stride = 1
- Padding = 2
- ReLU

self.maxpool1
Kernel size : 2

self.cnn2
- In-channel # : 16
- Out-channel # : 32
- Kernel size : 5
- Stride = 1
- Padding = 2
- ReLU

self.maxpool2
Kernel size : 2

Flatten

self.fc1
1568 → 10

Select Hyperparameter

```
torch.manual_seed(25)

model = CNNModel()

learning_rate = 0.003
epochs = 15
batchsize = 100

loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

```

Set random seed so that results are reproducible

Initialize the model

Using learning rate of 0.003 and 15 as epochs

Using mini-batch size of 100 for mini-batch gradient

Using Cross Entropy Loss + Adam optimizer

NOTE: CrossEntropyLoss() implements Softmax to the output layer

Identify Tracked Values

```
train_loss_list = []
validation_accuracy_list = np.zeros((epochs,))
```

Create empty list or NumPy arrays to hold training loss and validation accuracy

NOTE: If using mini-batch gradient, training loss can be tracked per iteration instead of epoch

Train Model

```python
import tqdm

train_inputs = torch.from_numpy(train_features).float()
train_targets = torch.from_numpy(train_targets).long()

validation_inputs = torch.from_numpy(validation_features).float()
validation_targets = torch.from_numpy(validation_targets).long()

testing_inputs = torch.from_numpy(test_features).float()
testing_targets = torch.from_numpy(test_targets).long()

train_batches_features = torch.split(train_inputs, batchsize)
train_batches_targets = torch.split(train_targets, batchsize)

batch_split_num = len(train_batches_features)
```

tqdm to visualize the progress of your training

Convert training/validation/testing datasets into PyTorch Tensors

Use torch.split() to split the train features/targets according to mini-batch size

Total number of mini-batches in split training set

torch.split() documentation: https://pytorch.org/docs/stable/generated/torch.split.html

Train Model

```
# Training Loop ---------------------------------------------------------------

for epoch in tqdm.trange(epochs):

    for k in range(batch_split_num):

        optimizer.zero_grad()

        train_batch_outputs = model(train_batches_features[k])

        loss = loss_func(train_batch_outputs, train_batches_targets[k])

        train_loss_list.append(loss.item())

        loss.backward()

        optimizer.step()

    # Compute Validation Accuracy ------------------------------------

    with torch.no_grad():

        validation_outputs = model(validation_inputs)

        correct = (torch.argmax(validation_outputs, dim=1) ==
                   validation_targets).type(torch.FloatTensor)

        validation_accuracy_list[epoch] = correct.mean()
```

Within epoch, grab k_{th} mini-batch (training features/targets) from training dataset

Training loop

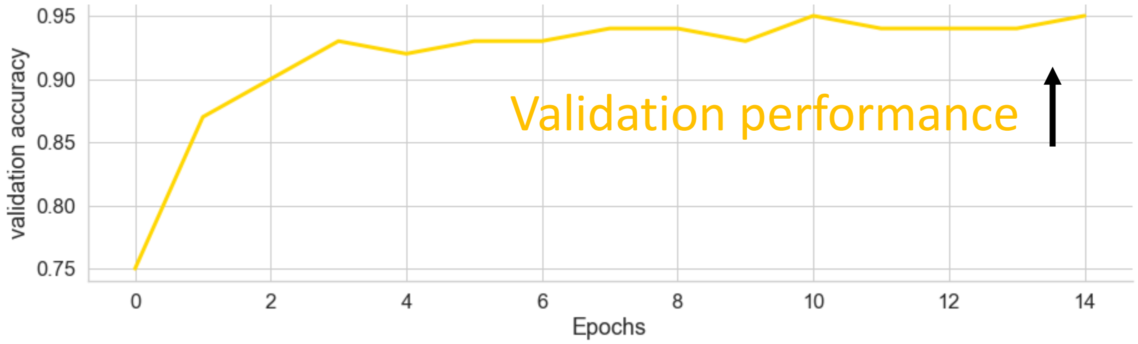
Compute Validation accuracy per epoch

Visualization and Evaluation

```python
plt.figure(figsize = (15, 9))

plt.subplot(2, 1, 1)
plt.plot(train_loss_list, linewidth = 3)
plt.ylabel("training loss")
plt.xlabel("Iterations")
sns.despine()
```

```python
plt.subplot(2, 1, 2)
plt.plot(validation_accuracy_list, linewidth = 3,
         color = 'gold')
plt.ylabel("validation accuracy")
plt.xlabel("Epochs")
sns.despine()
```




```python
with torch.no_grad():

    y_pred_test = model(testing_inputs)

    correct = (torch.argmax(y_pred_test, dim=1) ==
               testing_targets).type(torch.FloatTensor)

    print("Testing Accuracy: " + str(correct.mean().numpy()))
```

Testing performance

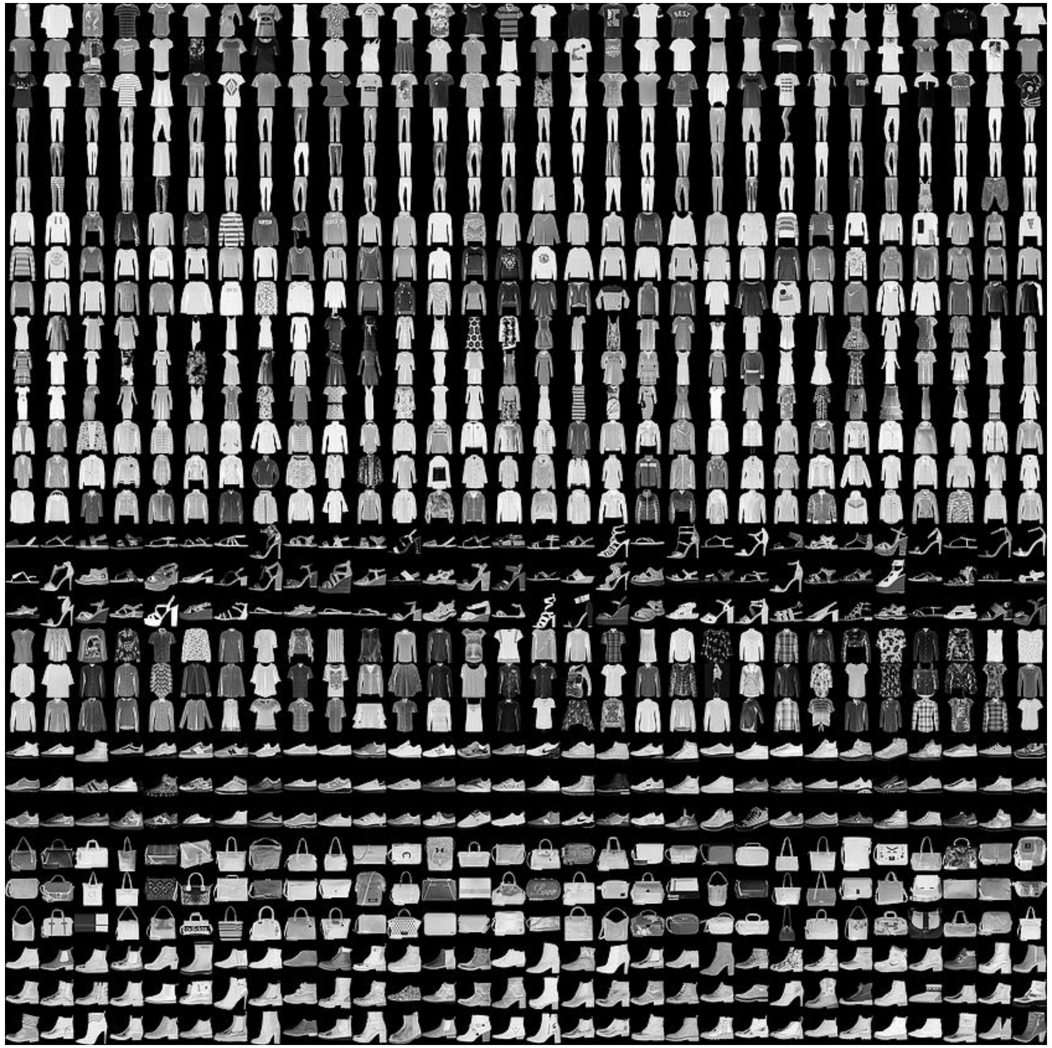
```
Testing Accuracy: 0.94
```

94% classification accuracy

LAB 4 ASSIGNMENT:

Surpass Human Performance in Fashion MNIST Classification

Fashion MNIST



0 – T-shirt/top
1 – Trouser
2 – Pullover
3 – Dress
4 – Coat
5 – Sandal
6 – Shirt
7 – Sneaker
8 – Bag
9 – Ankle Boot

Images of 10 classes of fashion items

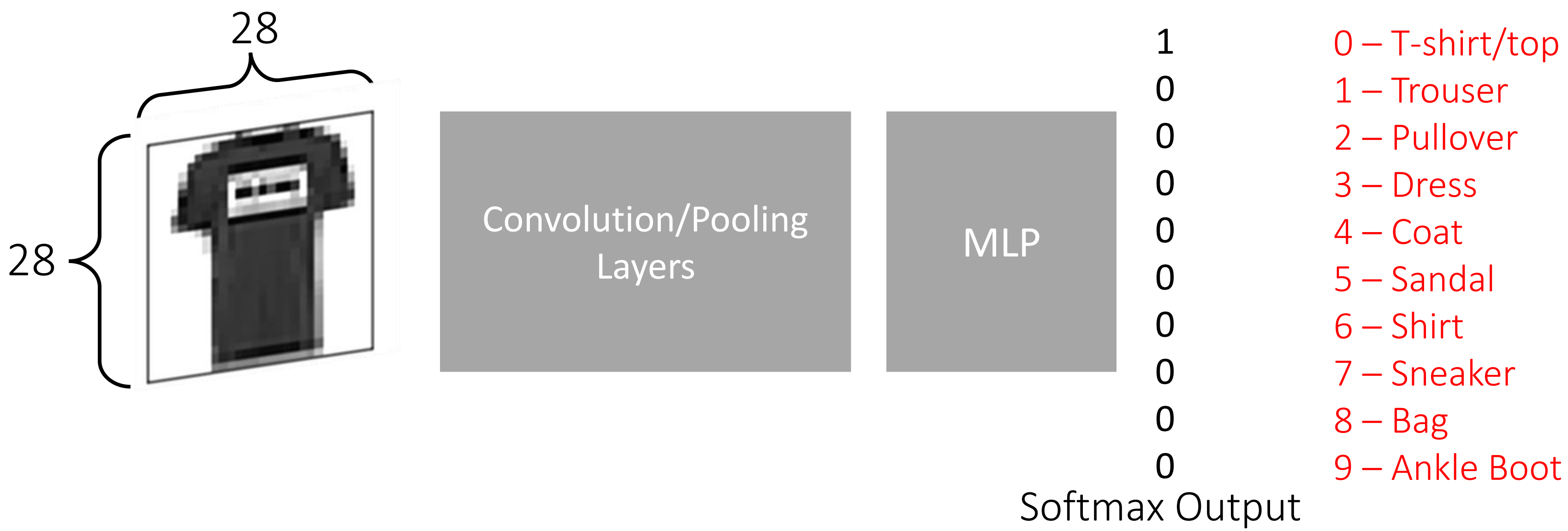
Target labels are the correct clothes types

Data consists of grayscale images of fixed size (28x28)

More complex features than MNIST

10000 training samples, 1000 testing samples

Surpass Human Performance in Fashion MNIST Classification



In this exercise, you will classify fashion item class (28 x 28) using your own **Convolutional Neural Network Architecture.**

Prior to training your neural net, 1) Normalize the dataset using standard scaler and 2) Split the dataset into train/validation/test.

Design your own CNN architecture with your choice of Convolution/Pooling/FCN layers, activation functions, optimization method etc.

Your goal is to **achieve a testing accuracy of >89%**, with no restrictions on epochs **(Human performance: 83.5%).**

Demonstrate the performance of your model via plotting the **training loss, validation accuracy** and printing out the **testing accuracy.**

After your model has reached the goal, print the accuracy in each class. What is the class that your model performed the worst?

Tips for Training Your CNN

First things to decide

- Number of Convolution/Pooling layers
- Out-channels, kernel size, stride, padding for each layer
- Number of FC layers
- Number of neurons per each FC layer
- Activation function
  (ReLU, Tanh, sigmoid)
- Training batch size
  (SGD, Mini-batch, Batch Gradient)
- Learning rate
- Optimizer
  (SGD, Adam, RMS Prop etc)
- Number of training epochs

Additional tips

- Start with a small baseline network
- Try different mini-batch sizes and learning rate before changing network architecture
- Add additional network component one at a time (e.g., additional convolution/pooling layer, FC layer)
- For optimizer, recommend fine tuning SGD or Adam with lr = 3e-4 as baseline.
- Stop training early if it shows sign of overfitting
- Reference known architectures for design ideas
  - LeNet-5 and AlexNet
- More tips by Andrej Karpathy