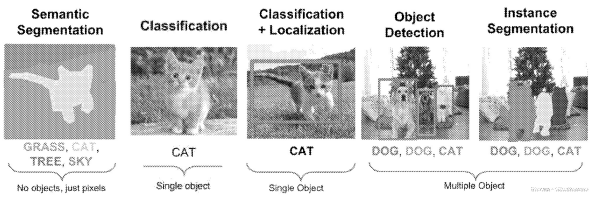
Classification tasks



No objects → Semantic segmentation

One object → Classification + localization

Multiply objects → Object detection and instance segmentation
↓
pixel level

Semantic segmentation: Classify each pixel into a category

Classification: Identify what is in the image as a whole

Classification + localization: Identify the object and provides its localization

Object detection: Detect and classify multiple objects in an image, providing their localizations

Instance segmentation: Detect and classify multiply objects and segment them at the pixel level

Object recognition

* Semantic gap (pixel vs labels):
  - Pixels values as descriptors are sensitive to small variations → same object might look different
* object recognition needs to be invariant
  - pose (rotation, translation)
  - illumination
  - deformation (eg. articulated objects)
  - occlusion
  - background
  - Natural variability (eg. cars)

Object recognition

* want:
  - generalization (invariance)
  - extend to other problems
* Need:
  - feature extraction (invariance)
  - classification algorithm

Approaches to Object recognition

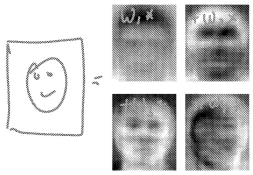
1) extract features (e.g. SIFT, Hog) and train a classifier.
2) Bag-of-words: extract patches and cluster them to form a "code book". Describe image using similarity to code words (with or without count).
3) Eigenfaces: map images to a lower dimensional space where less sensitivity to variations and classify in this domain
4) Convolutional Neural Networks (CNN): learn feature extraction and classification.

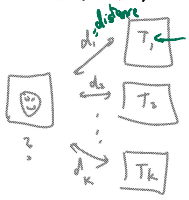
Models to extract features → SIFT: (Scale-invariant feature transform); HOG: Histogram of gradients)

This methods detect key points or gradients orientations in an image that can represents characteristics of objects

Object recognition

* Eigenfaces approach:
  - map image (e.g. 100x100) to lower dimensional vectors (e.g. 64) using PCA
  - measure similarity to templates in lower dimensional space (less sensitive to small variations)





Eigenface approach:

For 100×100 → 10,000 pixels
PCA reduces this to a smaller vector → Capture the most important features or variations

① Reduce dimensionally by PCA

② Measure similarity between new face and pre-stored template
↳ Using the distance between projected image and each template

PCA focus on significant variations, ignore minor changes

Close distance from input face is considered the closest match → Input will classify as that template

Object recognition

* Bag-of-words:
  - extract features (e.g. SIFT or Hog)
  - cluster features to create a codebook (dictionary)
  - Compute a distribution of code words in each class
  - classify using distribution of code words)

Cluster features → Codebook (each cluster represent a word which represents a group of similar features)

↓

Distribution of codewords (each image can be described by a distribution histogram of code words instead of raw pixels, each row represents the frequency of each word)

↓

Classification

Convolutional neural networks → From examples, not coding algorithms

* Convolutional Neural Networks (CNN): learn feature extraction and classification. (both points)
* Learn from examples instead of coding algorithms
* Architecture for one class can work for others by changing data → class generalization

Frameworks

- Keras (TensorFlow)
  - **High-level API** built on top of TensorFlow.
  - Simplifies model building and training.
  - Suitable for beginners and rapid prototyping.
  - Integrates seamlessly with TensorFlow for advanced features.
- TensorFlow (without Keras)
  - **Low-level API** developed by Google.
  - Provides fine-grained control over computations.
  - Suitable for researchers and advanced users who need custom operations and optimizations.

Frameworks → Set of tools that provide libraries to build ML models

- PyTorch
  - **Dynamic computation graph** framework developed by Facebook.
  - Allows flexibility and ease of debugging.
  - Popular in research and academia for its intuitive interface.
- PyTorch Lightning
  - **High-level API** built on top of PyTorch.
  - Simplifies code for training and validation.
  - Suitable for scalable and production-level code.
  - Provides a structured and organized approach to PyTorch projects.

Comparison of Frameworks

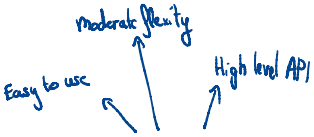
Feature	**TensorFlow/ Keras**	**TensorFlow (without Keras)**	**PyTorch**	**PyTorch Lightning**
API Level	High-level	Low-level	Mid-level	High-level
Ease of Use	Easy	Moderate	Moderate	Easy
Flexibility	Moderate	High	High	Moderate
Dynamic Computation Graph	No	Yes	Yes	Yes
Popularity in Research	Moderate	Moderate	High	High
Suitable for Beginners	Yes	No	Yes	Yes
Suitable for Production	Yes	Yes	Yes	Yes

Dynamic Computation Graphs

- **Dynamic computation graphs** are graphs that are built on-the-fly as operations are executed.
- Contrast with static computation graphs, which are defined before execution and cannot change during runtime.
- Benefits of Dynamic Computation Graphs
  - **Flexibility**: Easier to modify and debug.
  - **Intuitive**: More natural to use for models with varying behavior.
  - **Debugging**: Immediate feedback allows easier debugging
- Frameworks Supporting Dynamic Computation Graphs
  - **PyTorch**: Known for its dynamic computation graph
  - **PyTorch Lightning**: Inherits the dynamic nature of PyTorch
  - **TenforFlow**: Eager Execution - A mode in TensorFlow that allows operations to be executed immediately as they are called from Python.

Choosing a Framework

- **Ease of Use**
  - **TensorFlow/Keras**: High-level API, great for beginners, easy model building.
  - **PyTorch Lightning**: Simplifies PyTorch code, great for production-level code.
- **Flexibility**
  - **TensorFlow**: Low-level control, highly customizable, suitable for complex models.
  - **PyTorch**: Dynamic computation graphs, easy to modify and debug.
- **Popularity and Available Models**
  - **TensorFlow/Keras**: Extensive model zoo, large community, widely used in industry.
  - **PyTorch**: Popular in research, growing industry adoption, extensive pre-trained models.
- **Ease of Deployment on Mobile Platforms**
  - **TensorFlow Lite**: Optimized for mobile devices, supports both iOS and Android.
  - **PyTorch Mobile**: Streamlined for mobile deployment, supports both iOS and Android.
- **Ease of Serving Models**
  - **TensorFlow Serving**: Robust serving system, integrates with TensorFlow models
  - **TorchServe**: Easy to deploy and manage PyTorch models.



Keras Example

- **Load and Preprocess Data**: Load, split into training, validation, and test sets. Normalize if necessary.
- **Build the Model**: Define the architecture using Keras layers.
- **Initialize Weights**: Weights are initialized automatically by Keras
- **Compile the Model**: Specify the loss, optimizer, and metrics.
- **Train the Model**: Call the 'fit' method, which handles the forward pass, loss computation, backpropagation, and parameter updates internally.
- **Evaluate the Model**:
  - Use the 'evaluate' method to compute metrics on the test set.

for building CNNs
for stacking the neural network
imported libraries

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train[..., tf.newaxis] / 255.0, x_test[..., tf.newaxis] /
255.0

# Split the training data into train and validation subsets
x_train, x_val = x_train[:50000], x_train[50000:]
y_train, y_val = y_train[:50000], y_train[50000:]

# Build the model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

Reshape (28,28,1)
Pixel are rescaled to between 0 and 1 by dividing by 255
The pixel values of the images are 0 to 255
First 50.000 images for training
50.000 to the final for testing (10.000 images)

MNIST dataset: 28x28 grayscale images of handwritten digits

```
# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5, validation_data=(x_val, y_val))

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'\nTest accuracy: {test_acc}')
```

TensofFlow Example (without Keras)

- **Load and Preprocess Data**: Load, split, and normalize.
- **Build the Model**:Define the architecture manually using TensorFlow operations.
- **Initialize Weights**:Initialize weights using TensorFlow variables.
- **Define Loss and Optimizer**:Specify the loss function and optimizer.
- **Train the Model**:
  - **Forward Pass**: Pass the inputs through the model to get predictions.
  - **Compute Loss**: Calculate the loss between predictions and labels.
  - **Backpropagation**: Compute loss gradients w.r.t. model parameters.
  - **Update Parameters**: Apply gradients to update parameters (optimizer)
- **Evaluate the Model**:Manually compute metrics on the test set.

```
import tensorflow as tf

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train[..., tf.newaxis] / 255.0, x_test[..., tf.newaxis] /
255.0

# Split the training data into train and validation subsets
x_train, x_val = x_train[:50000], x_train[50000:]
y_train, y_val = y_train[:50000], y_train[50000:]

# Build the model
class SimpleCNN(tf.Module):
    def __init__(self):
        self.conv1 = tf.Variable(tf.random.normal([3, 3, 1, 32]),
name='conv1')
        self.conv2 = tf.Variable(tf.random.normal([3, 3, 32, 64]),
name='conv2')
        self.fc1_w = tf.Variable(tf.random.normal([7*7*64, 64]),
name='fc1_w')
        self.fc1_b = tf.Variable(tf.zeros([64]), name='fc1_b')
        self.fc2_w = tf.Variable(tf.random.normal([64, 10]), name='fc2_w')
        self.fc2_b = tf.Variable(tf.zeros([10]), name='fc2_b')
```

```
    def __call__(self, x):
        x = tf.nn.conv2d(x, self.conv1, strides=[1, 1, 1, 1], padding='SAME')
        x = tf.nn.relu(x)
        x = tf.nn.max_pool2d(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')

        x = tf.nn.conv2d(x, self.conv2, strides=[1, 1, 1, 1], padding='SAME')
        x = tf.nn.relu(x)
        x = tf.nn.max_pool2d(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')

        x = tf.reshape(x, [-1, 7*7*64])
        x = tf.nn.relu(tf.matmul(x, self.fc1_w) + self.fc1_b)
        x = tf.matmul(x, self.fc2_w) + self.fc2_b
        return x

# Instantiate the model
model = SimpleCNN()

# Loss function and optimizer
loss_fn = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.optimizers.Adam()
```

```
# Training and validation functions
def train_step(model, x, y):
    with tf.GradientTape() as tape:
        logits = model(x)
        loss = loss_fn(y, logits)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

def validate(model, x, y):
    logits = model(x)
    loss = loss_fn(y, logits)
    return loss

# Training loop
for epoch in range(5):
    for batch in range(len(x_train) // 32):
        x_batch = x_train[batch*32:(batch+1)*32]
        y_batch = y_train[batch*32:(batch+1)*32]
        train_step(model, x_batch, y_batch)

    val_loss = validate(model, x_val, y_val)
    print(f'Epoch {epoch}, Validation Loss: {val_loss}')

# Evaluate the model on the test set
logits = model(x_test)
predictions = tf.argmax(logits, axis=1)
accuracy = tf.reduce_mean(tf.cast(predictions == y_test, tf.float32))
print(f'Test accuracy: {accuracy.numpy()}')
```

PyTorch Example

- **Load and Preprocess Data**:Load, split, normalize

- **Build the Model**: Define the architecture by subclassing `nn.Module`.
- **Initialize Weights**:Weights initialized automatically, but can be customized
- **Define Loss and Optimizer**: Specify the loss function and optimizer.

- **Train the Model**:
  - **Forward Pass**: Pass the inputs through the model to get predictions.
  - **Compute Loss**: Calculate loss between predictions and actual labels.
  - **Backpropagation**: Compute gradients of the loss with respect to model parameters using `loss.backward()`.
  - **Update Parameters**: Apply gradients to update the model parameters using the optimizer's `step` method.

- **Evaluate the Model**:
  - Switch the model to evaluation mode and compute metrics on test set.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split

# Data loading and preprocessing
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
dataset = datasets.MNIST('.', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST('.', train=False, transform=transform,
download=True)
train_dataset, val_dataset = random_split(dataset, [50000, 10000])
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
# Define the model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(64*5*5, 64)
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(-1, 64*5*5)

        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
# Instantiate the model, define the loss function and the optimizer
model = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training Loop
for epoch in range(5):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    model.eval()
    val_loss = 0
    with torch.no_grad():
        for data, target in val_loader:
            output = model(data)
            val_loss += criterion(output, target).item()

    print(f'Epoch {epoch}, Validation Loss: {val_loss / len(val_loader)}')
```

```
# Evaluate the model on the test set
model.eval()
correct = 0
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

print(f'Test accuracy: {correct / len(test_loader.dataset)}')
```

PyTorch Lightning Example

- **Load and Preprocess Data**: Load, split, normalize
- **Define the Lightning Module**:
  - Define the architecture by subclassing `pl.LightningModule`.
  - Implement the `training_step`, `validation_step`, and `configure_optimizers` methods.
- **Initialize Weights:** initialized automatically, but can be customized
- **Train the Model**:
  - Use the `Trainer` class to handle the training loop, forward pass, loss computation, backpropagation, and parameter updates.
- **Evaluate the Model**:
  - Use the `Trainer` class to evaluate the model on the test set.

```
import pytorch_lightning as pl
import torch
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms

# Define the LightningModule
class LitModel(pl.LightningModule):
    def __init__(self):
        super(LitModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(64*5*5, 64)
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(-1, 64*5*5)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self(x)
        loss = F.cross_entropy(y_hat, y)
        return loss

    def configure_optimizers(self):
        return optim.Adam(self.parameters(), lr=0.001)
```

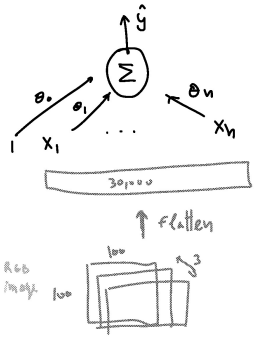
```
# Data preparation
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
dataset = datasets.MNIST('', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST('', train=False, transform=transform,
download=True)
train_dataset, val_dataset = random_split(dataset, [50000, 10000])
train_loader = DataLoader(train_dataset, batch_size=32)
val_loader = DataLoader(val_dataset, batch_size=32)
test_loader = DataLoader(test_dataset, batch_size=32)

# Train the model using PyTorch Lightning
model = LitModel()
trainer = pl.Trainer(max_epochs=5)
trainer.fit(model, train_loader, val_loader)

# Evaluate the model on the test set
trainer.test(model, test_loader)
```

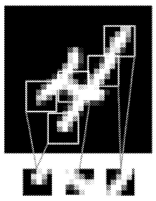
Flattening an image

- **Problems:**
  - **Numerous coefficients for each unit**
  - **Loss of spatial context**
  - **Loss of shift invariance (same object at different locations)**

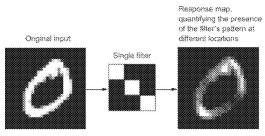


Convolution layers

- **Solution:**
  - **Convolve image with small filters (e.g. 3×3 or 5×5)**
  - **Share weights of filter between locations (shift invariance)**
  - **Sharing weights effectively increases data**

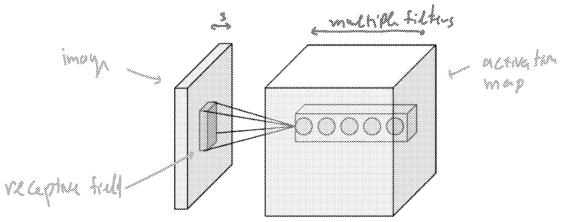


filter size
3×3 conv filter → (3×3+1) coefficients
FC layer → (100×100+1) coefficients
image size

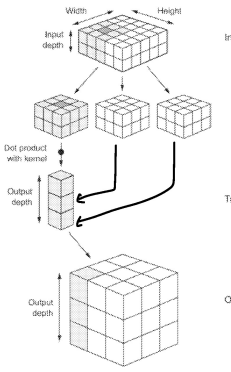


Convolution layers

- Extract image patches (windows)
- Vectorize image window and filter (dot product + bias)
- Filter extends full depth of image
- Multiple convolution filters per location (e.g. oriented edges)
- Use stride to move filter and so activation map maybe smaller



Convolution layers



5×5×2 image

3×(3×3×2) convolution filters

Convolution output at one location

Assembled outputs (no padding) 3×3×3

Convolution layers

(batch size, img height, img width, img channels)

e.g. (128, 27, 28, 1)

Convolution layer with 10 filters

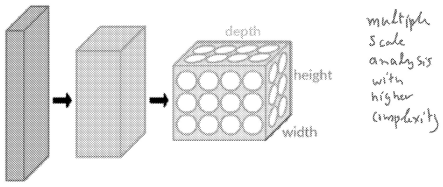
(batch size, height, width, channels)

(128, 28, 28, 10)

assume zero padding

Convolution layers

- **Multiple layers: spatial dimensions decrease and depth increases to compensate for reduced coefficients (keep the same number of coefficients)**
- **Nonlinear activation and sampling between layers**
- **Final FC layers perform classification after feature extraction**



multiple scale analysis with higher complexity

Convolution layers

Alexnet 2012