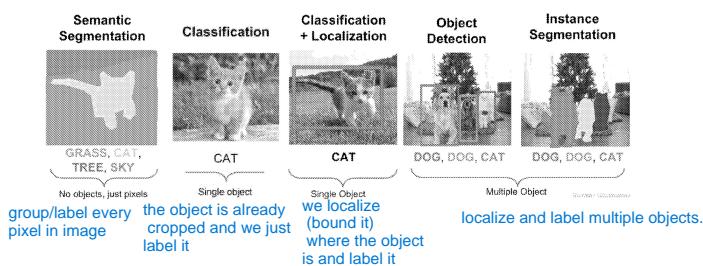


Classification tasks



instance vs semantic--->instance only focuses on the objects and also gives diff colors to diff instances of the object....the dogs are given diff colors, in case of semantics, it would have been same color for bit the dogs making it difficult to distinguish

Refer notes image for panoptic seg

Object recognition

*semantic gap (pixel vs labels):

- pixels values as descriptors are sensitive to small variations

*object recognition needs to be invariant

- pose (rotation, translation)
- illumination
- deformation (e.g. articulated objects) a person arm/legs in diff position in diff images
- occlusion only a part of the object seen
- background
- Natural variability (e.g. cars) diff ways/colors

Object recognition

*Want:

- Generalization (invariance)
- extend to other problems

use the same network/model to train diff data and use it for other problems

*Need:

- feature extraction (invariance)
- classification algorithm

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.

mostly used in NLP-->where the words are groups based on topic and place in code book. Then while detecting, each word is mapped to the corresponding group/topic. Similarly in CV, we extract patches which are the visual words and put them as code book to represent the groups and label for each later while detection/recognition

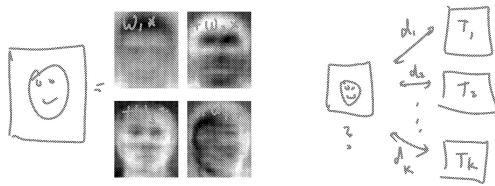
patch-->take a bunch of pixels in same neighborhood and make some transformations as an entity

Refer images

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}_{+ small variations})



use only important components from pca and discard others. the lower dimensional vectors represent the weights of the important principal components

1. do PCA
2. represent as lower dimension
3. represent image as the weighted sum of the components
4. measure distance b/w the weights

Adv:

when we take images of the same person, there may be changes in lightings but still the same person...so when we compare image pixels it may fail, but here we compare the weights of the pca which works well

disadv: facial expressions/glasses may change the key feature, not suitable for non-face objects

Object recognition

* Bag-of-words:

- extract features (e.g. SIFT or HOG)
- cluster features to create a code book (dictionary)
- Compute a distribution of code words in each class
- classify using distribution of code words

Convolutional neural networks

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

[View image](#)

Feature extraction in the convolution layers and classification in the final fully connected layers.

we keep reducing the size of the image at every convolution compensating with increasing the channels

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

- 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
 - **TensorFlow:** 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.

```

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.dewaxis(x_train[...], axis=2) / 255.0, x_test[...], tf.dewaxis(x_test[...], axis=2) / 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')
])

# Compile the model
model.compile(optimizer='sgd', 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'test accuracy: {test_acc}')

```

what is validation set? We cannot use test to help provide feedback to our model because then it would no longer be test set as we have incorporated its inputs. So, test is used only at the end to evaluate the model. In the process of model development, if we wanted feedback on how good the model is and improve, we use the validation set which provides feedback, we then update the model and again validate it before we go for the generalization testing on unknown input (Test set)

loss vs metric → loss can be used as metric as well to understand the model performance. the gradient of loss is used in the model optimization process but accuracy is not. metric could/could not be something who does not have gradient involvement in process

TensorFlow 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'Loss 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 processing
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((0.3387,), (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, [56000, 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)

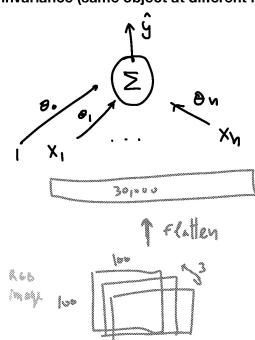
# Evaluate the model on the test set
trainer.test(model, test_loader)
...

```

We may consider that we can just convert the image into 1 vector(flatten) and decide for classification. Flatten means --->take an RGB image, place all the orws of the 3 channels in 1 single row.....
R1R2R3...RnG1G2G3...GnB1B2...Bn --->1 single row

Flattening an image

- Problems:
 - Numerous coefficients for each unit
 - Loss of spatial context ---the relationship between the pixels of the image becomes more difficult to understand
 - Loss of shift invariance (same object at different locations)



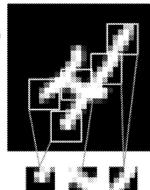
for 100x100 RGB image, after flattening it would be 30000 columns in row and then each one needs a weight parameter while training the neuron/model. this increases the computation needed and also when the model is built complex, training this flattened data would be even more difficult given the no.of parameters it needs

it is difficult for the model to learn the object and becomes complex for classification

Convolution layers

- Solution:
 - Convolve image with small filters (e.g. 3x3 or 5x5)
 - Share weights of filter between locations (shift invariance)
 - Sharing weights effectively increases data

filter size
3x3 conv filter $\rightarrow (3 \times 3 \times 1)$ coefficients
FC layer $\rightarrow (100 \times 100 + 1)$ coefficients
image size

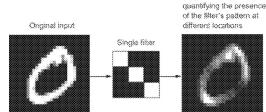


In convolution, we do not explicitly provide the conv filter weights, it is learnt by the model. How is conv solving the issues of the flattened image??

the same conv filter size say 3x3 is used across the images...9 weights +1 bias = 10 weights for every filter. If we need more filters, then say 30 filter--->total 300 weights making the number of parameters needed less.

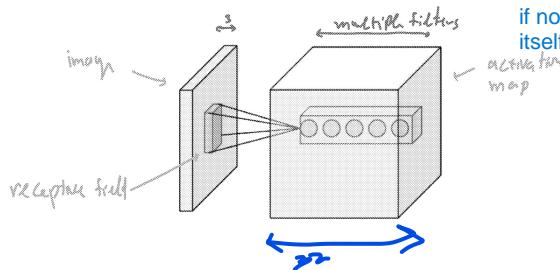
Since we move the filter throughout the image we get the spatial/shift invariance as they share the same weights

reduce dimension-->simple model-->less data/shared weights-->easy to train



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

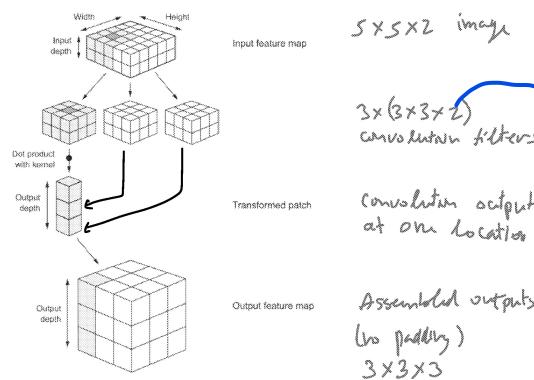


if the input image has 3 channels(RGB), the filter goes to the depth of the channel.
Meaning--it convolves through the 3 channels of the image and all together one weight will be learnt/generated.

We can do the same for multiple filters(say 32) because each filter may learn diff feature.. Remember that the filter slides throughout the image , thereby every pixel would have the weights in the dimension as $1 \times 1 \times 32 \rightarrow$ thereby forming the 3d result of weights.

if no sampling(pooling) or varied strides used, it will result in the height and breadth of the image itself.

Convolution layers



filter should always match the depth of the input

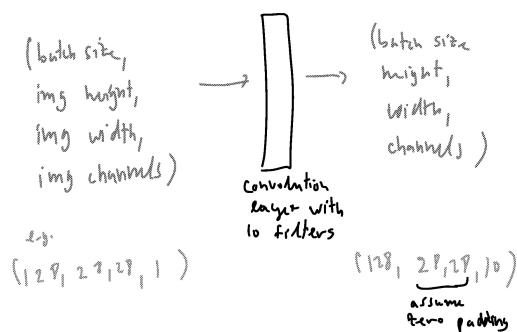
Same location, 3 filters-->3 weights

depth/height-->number of filters used

width/breadth--->based on the stride and the padding

it has become 3×3 from 5×5 input because the stride was 1 and there was no zero-padding done to maintain the input size. the filter was placed wherever it just fits-->losing 1 pixel from each side for a 3×3 filter

Convolution layers

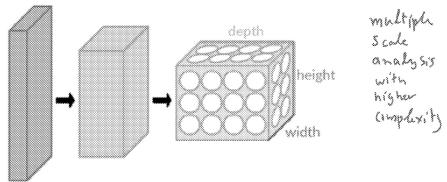


when we do convolutions for multiple images, the input itself would then be 4d and so is the output.

we can even do 5d in case of doing multiple videos each video having 10 frames and are of RGB channels

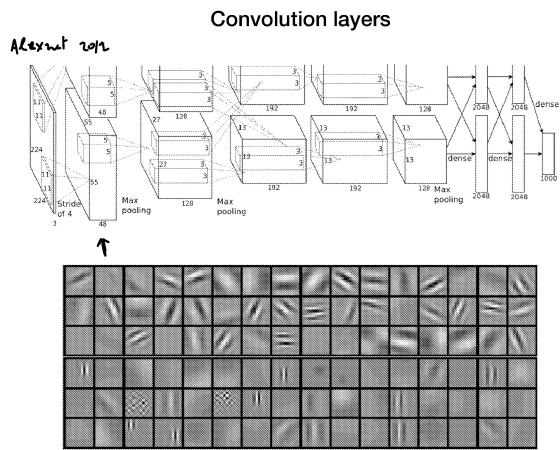
Convolution layers

- 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



nonlinear activation- introduce non-linearity into the model, allowing it to learn complex patterns and relationships in the data.

we reduce the spatial dimension so that we have an overview/pyramid view



Refer image