IT496: Introduction to Data Mining

Lecture 33-34

Training Convolutional Neural Networks

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana

9th / 10th November 2023

Convolution Layers: Hyperparameters

- Convolution layers have quite a few hyperparameters:
  - number of filters,
  - their height and width,
  - the strides,
  - padding type, and
  - the activation function
- Using cross-validation to find the right hyperparameter values is very time consuming.
- There are common CNN architectures: *LeNet5*, *AlexNet*, *GoogLeNet*, *VGGNet*, **ResNet**, *Xception*, *SENet*, etc.; these can give some idea of which hyperparameter values work best in practice.
- We will NOT discuss all of them in this course.

Convolution Layers: Memory Requirements

- Convolution layers require a huge amount of RAM
    - during training, the backward pass of backprop requires all the intermediate values computed during the forward pass.
- For example,

200 5

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="SAME", activation="relu")
```

Input (RGB) image dimensions: **150 x 100 x 3**

Number of parameters: **(5 x 5 x 3 + 1) x 200 = 15, 200**

Total float multiplications: **200 x 150 x 100 x 5 x 5 x 3**

Total Memory Required: **200 x 150 x 100 x 32 = 96 M bits ≅ 12 MB**

(just for an instance and for just one layer)

Convolution Layers: Memory Requirements

- During Training:

  Everything computed during the forward pass needs to be preserved for the backward pass, so the total amount of RAM required by all layers is the minimum requirement.

- What about during Inference (when making a prediction for a new instance)?

  The RAM occupied by one layer can be released as soon as the next layer has been computed, so we only need as much RAM as required by two consecutive layers.

Convolution Layers: Memory Requirements

- If training crashes because of an *out-of-memory error*, we have the following choices:
  - reducing the mini-batch size.
  - reducing dimensionality using a stride,
  - removing a few layers,
  - using 16-bit floats instead of 32-bit floats,
  - distribute the CNN across multiple devices.

Convolution Layers: Overfitting

Mainly two regularization techniques are used:

- Adding dropout layer
- Data Augmentation
    - Artificially increasing the size of the training set by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.
    - Simply adding white noise will not help; the modifications should be learnable.
    - However, this is not as good as additional real examples: these synthesized examples are correlated with each other and the originals from which they were generated.

Convolution Layers: Overfitting

- Data Augmentation
    - This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures.

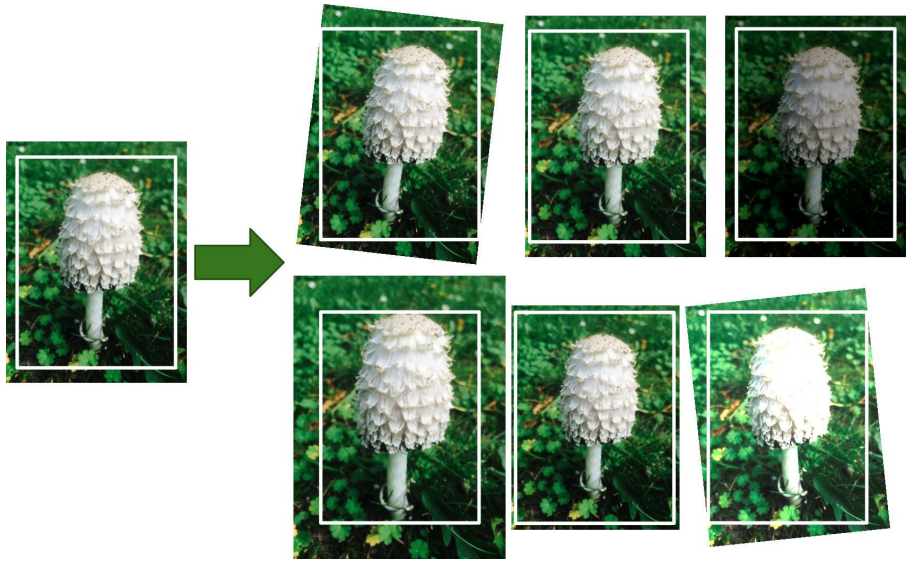


Image Recognition: Cats and Dogs

- There is a dataset, supplied by Microsoft researchers, for a Kaggle competition: Dogs vs. Cats | Kaggle
    - 12,500 medium-resolution JPEGs depicting cats and 12,500 depicting dogs.
- In the following implementation, we use a subset of the full dataset:
    - training set: 1000 cats and 1000 dogs;
    - validation set: 500 cats and 500 dogs;
    - test set: 500 cats and 500 dogs.

Image Recognition: Cats and Dogs - Data Preprocessing

- Keras gives us an extremely useful function: `image_dataset_from_directory` (see the documentation).
    - This function decodes images from one format into a grid with a certain number of channels. For example, if the raw images are JPEGs, it will decompress them.
    - Mostly, its default values for the arguments are what we want. Often we only need to think about the `directory` and the `label_mode`.

```
train_dataset = image_dataset_from_directory(directory=train_dir, label_mode="binary", image_size=(224, 224))
val_dataset = image_dataset_from_directory(directory=val_dir, label_mode="binary", image_size=(224, 224))
test_dataset = image_dataset_from_directory(directory=test_dir, label_mode="binary", image_size=(224, 224))
```

```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
```

Image Recognition: Cats and Dogs - Creating the ConvNet

- Since the images are bigger than the Fashion MNIST ones, we use a network with more layers

```
inputs = Input(shape=(224, 224, 3))
x = Rescaling(scale=1./255)(inputs)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=64, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dense(512, activation="relu")(x)
outputs = Dense(1, activation="sigmoid")(x)
convnet = Model(inputs, outputs)
convnet.compile(optimizer=RMSprop(learning_rate=0.0001), loss="binary_crossentropy", metrics=["accuracy"])
```

Image Recognition: Cats and Dogs - Creating the ConvNet

```
convnet.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 rescaling (Rescaling)       (None, 224, 224, 3)       0

 conv2d (Conv2D)             (None, 222, 222, 128)     3584

 max_pooling2d (MaxPooling2D  (None, 111, 111, 128)    0
 )

 conv2d_1 (Conv2D)           (None, 109, 109, 128)     147584

 max_pooling2d_1 (MaxPooling  (None, 54, 54, 128)      0
 2D)

 conv2d_2 (Conv2D)           (None, 52, 52, 64)        73792

 max_pooling2d_2 (MaxPooling  (None, 26, 26, 64)       0
 2D)

 conv2d_3 (Conv2D)           (None, 24, 24, 32)        18464

 max_pooling2d_3 (MaxPooling  (None, 12, 12, 32)       0
 2D)

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 512)               2359808

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 2,603,745
Trainable params: 2,603,745
Non-trainable params: 0
_________________________________________________________________
```

Image Recognition: Cats and Dogs - Training and Testing

- We will use more epochs than before, but still with early stopping.

```
convnet_history = convnet.fit(train_dataset, epochs=30,
                validation_data=val_dataset,
                callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                verbose=0)
```

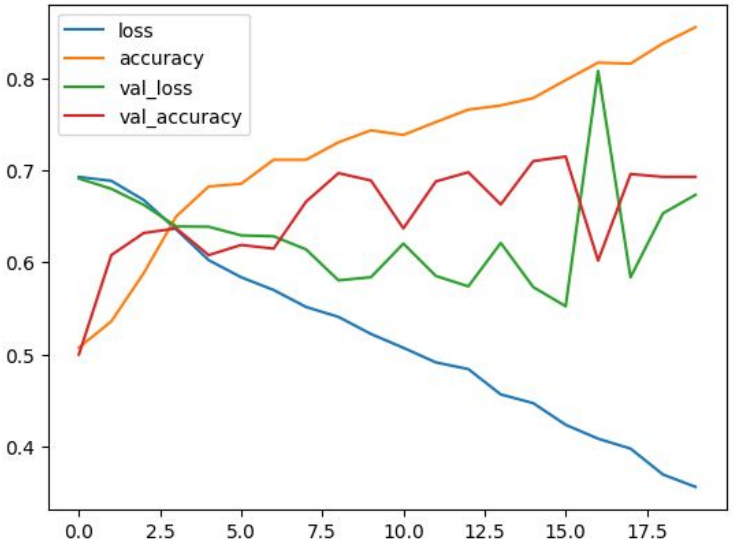


Image Recognition: Cats and Dogs - Data Augmentation

- In Keras, we add various layers to perform data augmentation.
- These layers will only augment the training data, not the validation or test data.
- We can *flip*, *rotate*, *zoom*, and *shift* the images.

```
augmentation_layers = Sequential([
    Input(shape=(224, 224, 3)),
    RandomFlip(mode="horizontal"),
    RandomRotation(factor=0.1),
    RandomZoom(height_factor=(-0.2, 0.2)),
    RandomTranslation(height_factor=0.2, width_factor=0.2)
])
```

Image Recognition: Cats and Dogs - Data Augmentation

- So now we add those layers to our network.

```
inputs = Input(shape=(224, 224, 3))
x = RandomFlip(mode="horizontal")(inputs)
x = RandomRotation(factor=0.1)(x)
x = RandomZoom(height_factor=(-0.2, 0.2))(x)
x = RandomTranslation(height_factor=0.2, width_factor=0.2)(x)
x = Rescaling(scale=1./255)(x)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=128, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=64, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Conv2D(filters=32, kernel_size=(3, 3), activation="relu")(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dropout(0.5)(x)
x = Dense(512, activation="relu")(x)
outputs = Dense(1, activation="sigmoid")(x)
augmented_model = Model(inputs, outputs)
augmented_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="binary_crossentropy", metrics=["accuracy
```

Image Recognition: Cats and Dogs - Training and Testing

- We will use more epochs than before, but still with early stopping.

```
augmented_model_history = augmented_model.fit(train_dataset, epochs=60,
                validation_data=val_dataset,
                callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                verbose=0)
```

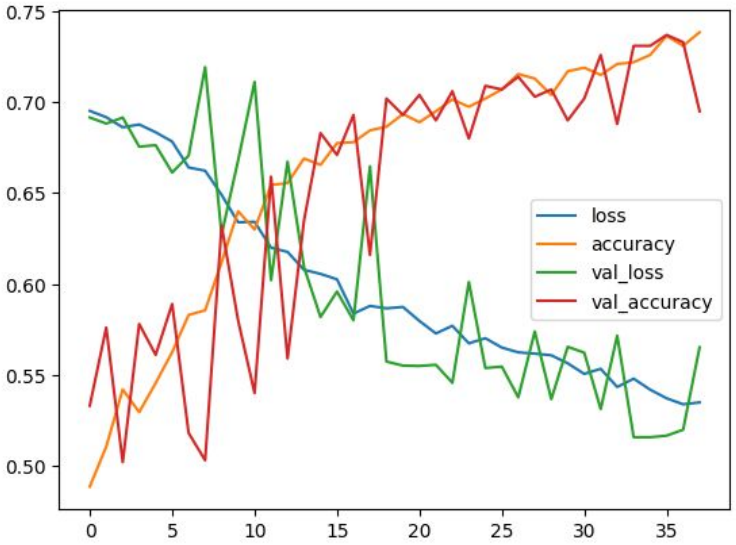


Image Recognition: Cats and Dogs - Training and Testing

- See the prediction on the following four images (`demo_dataset`)




```
augmented_model.predict(demo_dataset) # Class 0 is cat and class 1 is dog (alphabetic)
```

```
1/1 [==============================] - 1s 600ms/step
array([[0.11105712],
       [0.9930074 ],
       [0.6058035 ],
       [0.65952003]], dtype=float32)
```

Pretrained Convolutional Neural Network

- A pretrained network is a saved network that was trained, usually on a large dataset.
- These are increasingly being made available for image classification, speech recognition and other tasks.
- Consider the ImageNet dataset (http://www.image-net.org/):
    - 1.4 million images, each manually labeled with one class per image;
    - thousands of classes, mostly animals and everyday objects;
    - annual competitions (ImageNet Large Scale Visual Recognition Challenges, ILSVRC), now hosted on Kaggle.

Pretrained Convolutional Neural Network

- We will see how to use a pretrained network: ResNet50

```
resnet50 = ResNet50(weights="imagenet", include_top=True, input_shape=(224, 224, 3))
```

- `include_top`: whether to include the fully-connected layer at the top of the network.

```
predictions = resnet50.predict(demo_dataset)
decode_predictions(predictions, top=3)
```

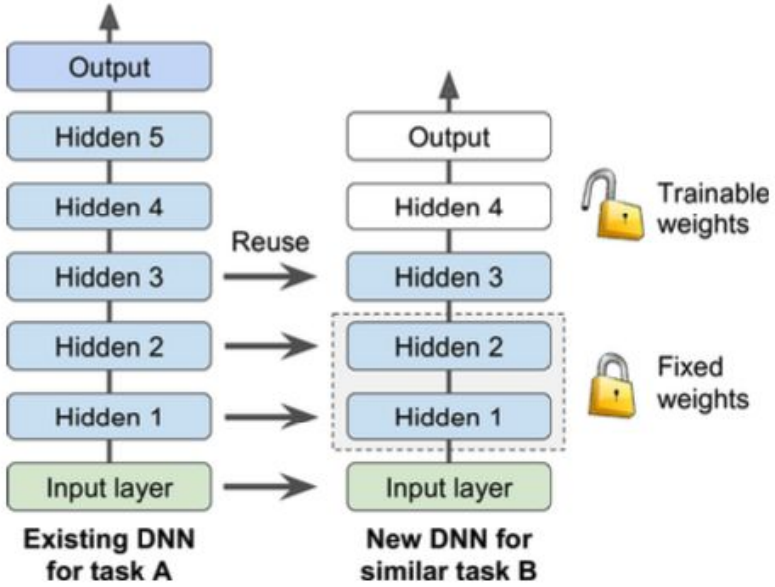
```
[[('n02123159', 'tiger_cat', 0.38074818),
  ('n02123045', 'tabby', 0.3423185),
  ('n02124075', 'Egyptian_cat', 0.12753917)],
 [('n02105412', 'kelpie', 0.41296554),
  ('n02099712', 'Labrador_retriever', 0.19615647),
  ('n02099849', 'Chesapeake_Bay_retriever', 0.10481451)],
 [('n02412080', 'ram', 0.8103329),
  ('n02444819', 'otter', 0.040895212),
  ('n02396427', 'wild_boar', 0.025922885)],
 [('n04398044', 'teapot', 0.99999344),
  ('n03063689', 'coffeepot', 6.545879e-06),
  ('n04560804', 'water_jug', 1.7925926e-08)]]
```

Transfer Learning

- Transfer learning:
  - taking a model that was learned when solving one problem and re-using it for solving a different but related problem.
- Advantages:
  - it speeds-up training for the new problem;
  - it means that less training data may be needed for the new problem.

Transfer Learning

- Deep neural networks are more amenable to transfer learning than many other machine learning techniques:
  - take a pre-trained network;
  - re-use its lower layers, even *freezing* their weights.



Transfer Learning

- For example:
    - We have several networks that are pre-trained on ImageNet, including ResNet50:
    - trained to classify images into 1000 classes (various animals, vehicles, etc.).
    - We can re-use the lower layers in a new network that is trained to classify images of just cats and dogs, or different types of vehicles, or for face recognition, or perhaps even facial expression recognition.
- Of course, this will only work well if the original and new tasks share similar low-level features.
    - The more similar the new problem is to the original problem, the more layers we may want to re-use.

Re-using the Convolutional Base of a Pretrained ConvNet

- Convolutional Neural Networks typically comprise two parts:
  - the *convolutional base*: the convolutional and pooling layers;
  - the densely-connected top layers for, e.g. classification.
- We want to reuse the convolutional base:
  - the features learned by these layers are likely to be more generic;
  - the features learned by the top layers will be more specific to the original task.

Image Recognition: Cats and Dogs - Using Transfer Learning

- We *freeze* the weights in the layers of the convolutional base. If we did not, then the features that ResNet50 learned previously would be lost.

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
```

```
resnet50_base = ResNet50(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
```

```
resnet50_base.trainable = False
```

Image Recognition: Cats and Dogs - Using Transfer Learning

We will re-use the convolutional base of the ResNet50 model within a new network for classifying cats and dogs.

- We create our new network but we add the pre-trained convolutional base of the ResNet50 model, just as we would add a layer.
- We can precede it by a function to *preprocess* the data into a form that ResNet50 expects. This even allows us to exclude the Rescaling layer.
- We could even add the augmentation layers if we wanted.

```
inputs = Input(shape=(224, 224, 3))
x = preprocess_input(inputs)
x = resnet50_base(x)
x = Flatten()(x)
outputs = Dense(1, activation="sigmoid")(x)
transfer_model = Model(inputs=inputs, outputs=outputs)
```

Image Recognition: Cats and Dogs - Using Transfer Learning

```
transfer_model.summary()
```

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_6 (InputLayer)        [(None, 224, 224, 3)]     0

 tf.__operators__.getitem (S  (None, 224, 224, 3)      0
 licingOpLambda)

 tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

 resnet50 (Functional)       (None, 7, 7, 2048)        23587712

 flatten_2 (Flatten)         (None, 100352)            0

 dense_4 (Dense)             (None, 1)                 100353

=================================================================
Total params: 23,688,065
Trainable params: 100,353
Non-trainable params: 23,587,712
_________________________________________________________________
```

Image Recognition: Cats and Dogs - Using Transfer Learning

```
transfer_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="binary_crossentropy", metrics=["accuracy"
```

```
transfer_model_history = transfer_model.fit(train_dataset, epochs=30,
                validation_data=val_dataset,
                callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                verbose=0)
```

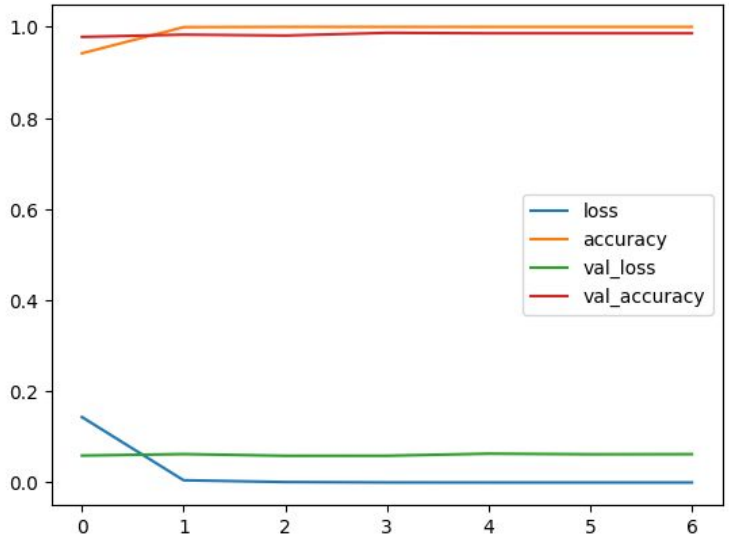


Image Recognition: Cats and Dogs - Using Transfer Learning

Now that our new top layers are well-trained, we can *unfreeze* all layers in the base (or just the top ones in the base) and continue training.

- Simplest is to unfreeze all of them, like this: `# resnet50_base.trainable = True`
- But, for reasons we will not unfreeze `BatchNormalization` layers:

```
for layer in resnet50_base.layers:
    if isinstance(layer, BatchNormalization):
        layer.trainable = False
    else:
        layer.trainable = True
```

Image Recognition: Cats and Dogs - Using Transfer Learning

- In Keras, re-compilation is needed at this point. (We re-compile, but we do not build a new model as the whole idea is to continue to tune the one we have already built.)

```
transfer_model.compile(optimizer=RMSprop(learning_rate=0.001), loss="binary_crossentropy", metrics=["accuracy"]
```

```
transfer_model_history = transfer_model.fit(train_dataset, epochs=30,
                validation_data=val_dataset,
                callbacks=[EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)],
                verbose=0)
```

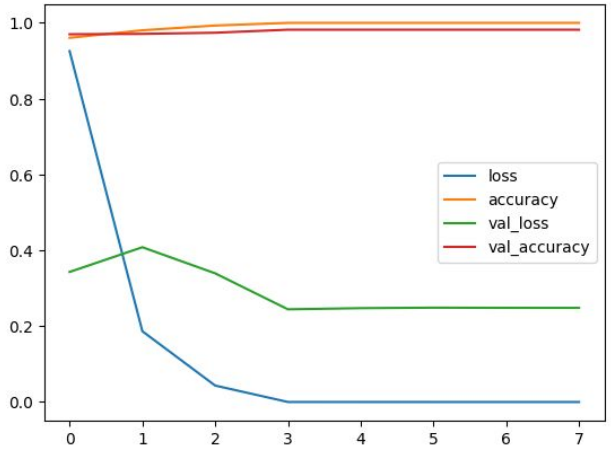


Image Recognition: Cats and Dogs - Using Transfer Learning

- Now, we test and compare all the version of our convnets on the test set.

```
# The original convnet
test_loss, test_acc = convnet.evaluate(test_dataset)
test_acc
```

```
32/32 [==============================] - 138s 3s/step - loss: 0.5667 - accuracy: 0.7280
0.7279999852180481
```

```
# The convnet with dropout, trained using data augmentation
test_loss, test_acc = augmented_model.evaluate(test_dataset)
test_acc
```

```
32/32 [==============================] - 3s 70ms/step - loss: 0.5555 - accuracy: 0.7150
0.7149999737739563
```

```
# The convnet that was trained and tuned using transfer learning
test_loss, test_acc = transfer_model.evaluate(test_dataset)
test_acc
```

```
32/32 [==============================] - 4s 96ms/step - loss: 0.2624 - accuracy: 0.9790
0.9789999723434448
```

Concluding Remarks on CNNs

- We saw that transfer learning helps us in cases where we have more limited data.
- Consider tasks that involve audio, text and time series data. For these tasks, Recurrent Neural Networks would be the obvious choice (we will see in IT492: Recommendation Systems).
    - But, in some cases, 1D (one-dimensional) convolutional networks can be used successfully instead.
    - Although, to be fair, these days Transformers (also will be covered in IT492) are displacing Recurrent Neural Networks and 1D convnets in many cases.
- Consider tasks that involve graphs (nodes and edges). We can process these using Graph Neural Networks. There are graph convolutions that can be used within these neural networks.

Next lecture

Clustering

21st November 2023