CS4619: Artificial Intelligence II

More Neural Network Architectures

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [ ]: %load_ext autoreload
%autoreload 2
%matplotlib inline

In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

Acknowledgments

- The diagrams and code are based on diagrams and code in: A. Géron: Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow (2nd edn), O'Reilly, 2019

Warning

- The code does not run. I'm providing snippets only
Introduction

- Mostly, we’ve been looking at quite simple neural network architectures: stacks of layers
  - In Keras, Sequential models are ideal for this
- But, the architecture could be more complicated
  - E.g. you might want multiple inputs
    - This might be useful if your dataset comprises images and text, which might best be handled by different subnetworks
  - E.g. you might want multiple outputs
    - This might be useful if, from pictures of faces, you want to classify the expression (smiling, surprised) but you also want to classify by eyewear (wearing glasses or not)
  - In Keras, the Model class used with the functional API can be used
- We’ll look at some examples. We won’t do all the code (just snippets), so don’t try to execute them. And you don’t need to learn the functional API

Classification and Localization

- You may want to locate and classify the main object in a picture
  - This is a classification task: what kind of object is in the image (cat, dog,…)
  - It is also a regression task. In fact, you will want to predict four numbers that describe a bounding box around the object:
    - \(x\)-coordinate of the centre of the bounding box
    - \(y\)-coordinate of the centre of the bounding box
    - width of the bounding box
    - height of the bounding box
- Hence, this requires a network with multiple outputs. How many outputs?

```python
In [ ]:
vgg16_base = VGG16(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
class_output = Dense(n_classes, activation="softmax") (vgg16_base)
loc_output = Dense(4, activation="linear") (vgg16_base)
model = Model(inputs=vgg16_base.input, outputs=[class_output, loc_output])

model.compile(optimizer=RMSprop(lr=0.00003),
loss=["sparse_categorical_cross_entropy", "mse"],
loss_weights=[0.7, 0.3])
```
In this snippet you can see the `Model` class but also the functional API

You can see the multiple outputs, and you can see that each output requires its own loss function
- By default, Keras sums the losses
- If you care more about one loss than another, you supply weights

Getting a labelled dataset is now even harder: every image must come with the class and bounding box of the main object in the image. We have talked about this previously

What we’ve been discussing is classification and localization of the main object in an image. But we can use convolutional neural networks for **object detection**, which refers to classifying and localizing multiple objects in an image, e.g. to say that an image contains a car and a pedestrian and to give bounding boxes for each

One way to do object detection:
- Train a convolutional neural network to classify and locate a single object
- Slide it across the image, i.e. predict and shift, e.g., for each $3 \times 3$ region
- Perhaps repeat this for each $4 \times 4$ region
- Post-process the results because you will have detected the same object multiple times

There are other clever ways of doing object detection, but we don’t have time to look at them (look up Fully Convolutional Networks and YOLO, if you are interested)

---

**Autoencoders**

- Autoencoders learn to copy their inputs to their outputs
  - They have the same number of outputs as inputs
  - Reconstruction loss: They are trained with a loss function that penalizes them if the output for each example is not the same as the input
  - The network will have constraints that prevent the autoencoder from simply copying inputs to outputs; the constraints force the autoencoder to learn an efficient representation, rather than just memorizing; the autoencoder must find features that capture the inputs and allow it to reconstruct them as outputs
- Stacked autoencoders, for example, have
  - An encoder: layers that convert the input to a more compact internal representation
  - A decoder: layers that convert the internal representation to the outputs

```python
In [ ]:
stacked_encoder = Sequential([  
    Dense(100, activation="relu", input=(200,)),
    Dense(30, activation="relu")
])
stacked_decoder = Sequential([  
    Dense(100, activation="relu", input_shape=[30]),
    Dense(200, activation="linear")
])
stacked_autoencoder = Sequential([stacked_encoder, stacked_decoder])
stacked_autoencoder.compile(optimizer=RMSprop(lr=0.00003),
                          loss="mse")
stacked_autoencoder.fit(X_train, X_train, epochs=10, validation_data=[X_valid, X_valid])
```

- In the snippet, we create two submodels (encoder and decoder) and combine
- I have pretended we have 200 numeric-valued features. Hence, I use linear as the activation function on the output layer. Hence, also `mse` is a suitable loss function
- Note that `X_train` is used for both the inputs and the targets
- Autoencoders have several uses, and we will look at one
Unsupervised Pretraining using Stacked Autoencoders

- Suppose you want to build, e.g., an image classifier
- Suppose you have lots of unlabeled data and a little labeled data
  - E.g., you’ve downloaded millions of images from the web, but you’ve manually labeled only a small subset
- First train a stacked autoencoder on all your data: hopefully you learn an autoencoder that is good at detecting features in the images
- Then reuse its lower layers (the encoder) like we reused lower layers of a pretrained network in the previous lecture
  - Create a network that has the lower layers of the autoencoder and then a few additional layers that implement a classifier
  - Train this new network on the labeled data, but with all or most of the encoder’s weights frozen

Some Other Autoencoders

- We force stacked autoencoders to learn useful features by giving the internal layers lower dimensionality
- In a denoising autoencoder,
  - All the layers might have the same number of neurons
  - To force it to learn useful features, during training noise is added to the inputs
  - But the network is trained to reconstruct the noise-free inputs (a bit like dropout)
- In a sparse autoencoder,
  - A regularization term is added to the loss function to reduce the number of active neurons
  - This forces the autoencoder to represent each input as a combination of a small number of activations
- Variational encoders can generate new instances that look like they were sampled from the training set. However, a new kind of network has become more popular for this …

Generative Adversarial Networks

- Generative Adversarial Networks (GANs) allow for the generation of fairly realistic synthetic images
- They comprise:
  - A generator network: takes a random vector (think of it as if it were samples from the coded representations in the middle of an autoencoder) and decodes it into a synthetic image
  - A discriminator network: takes an image (which might be real (from the training set) or might be synthetic (from the generator)) and predicts whether it is real or synthetic
    Analogy: a forger and an art expert
- Training:
  - The generator is trained to fool the discriminator, so it must produce ever more realistic images
  - The discriminator is trained to tell synthetic from real images with high accuracy
  - As one network gets better, the other will have to get better
    Because it’s a dynamic system, there isn’t a fixed minimum
    Instead of seeking a minimum, we are seeking an equilibrium between two adversaries
    Hence, very difficult to train successfully
Training (in more detail)

- In the first phase, we train the discriminator
  - Sample real images from the training set, labeled 1
  - Generate an equal number of synthetic images, labeled 0
  - Use an epoch of backprop on the discriminator only with binary cross-entropy as the loss function
- In the second phase, we train the generator
  - Generate some synthetic images, label them all 1 (yes, 1!)
  - Use an epoch of backprop on the whole GAN with binary cross-entropy as the loss function but with the weights of the discriminator frozen within the GAN

This code snippet assumes that the coded representation has 30 features and that the images are $28 \times 28$

```python
In [ ]:

codings_size = 30
generator = Sequential([
    Dense(100, activation="relu", input_shape=[codings_size]),
    Dense(150, activation="relu"),
    Dense(28 * 28, activation="sigmoid"),
    Reshape([28 * 28])
])
discriminator = Sequential([
    Flatten(input_shape=[28, 28]),
    Dense(150, activation="relu"),
    Dense(100, activation="relu"),
    Dense(1, activation="sigmoid")
])
gan = Sequential([generator, discriminator])

In [ ]:
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")

In [ ]:
# The regular fit method cannot be used

for epoch in range(n_epochs):
    for X_batch in dataset:
        # phase 1
        noise = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator(noise)
X_synthetic_and_real = tf.concat([generated_images, X_batch], axis=0)
y1 = tf.constant([0.]) * batch_size + [[1.]] * batch_size
discriminator.trainable = True
discriminator.train_on_batch(X_synthetic_and_real, y1)

        # phase 2
        noise = tf.random.normal(shape=[batch_size, codings_size])
y2 = tf.constant([[1.]] * batch_size)
discriminator.trainable = False
gan.train_on_batch(noise, y2)
```

The code snippet is just a rough idea. As already said, successful training is hard, and would require a lot of tweaks, which you can look up if you're interested
GANs are now used for these and other purposes:

- increasing the resolution of images
- colorization
- image editing, e.g. replacing photo-bombers with realistic backgrounds
- turning sketches into photo-like images
- augmenting image datasets
- ...

```
In [ ]:
```