IT496: Introduction to Data Mining

Lecture 25

Neural Network Examples Using Keras

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

Arpit Rana

17th October 2023

Deep Learning

- The word 'deep' in 'deep learning' does not mean profound.
- In deep learning, we have 'lots' of layers — tens or even hundreds.

Representations

One way of thinking about Machine Learning:

- It uses guidance from a feedback signal to automatically find transformations that turn input data into more useful representations.

  For example,

  - in the case of supervised learning, the feedback comes from the loss function and the algorithm seeks a representation that is closer to the target outputs.

Representations

Deep learning is about jointly finding successive layers of representations, usually in the form of the layers of a neural network.

- The network takes in vectors (examples).
- The first layer in some sense transforms the input vectors into new vectors — a different representation of the inputs examples.
- The second layer transforms again into new vectors — another representation.
- Since each layer produces a new representations, one way of thinking about this is, for the kinds of tasks on which it is successful, deep learning automates feature engineering.

Drivers of Deep Learning

Hardware:

- Faster CPUs but then highly-parallel Graphical Processing Units (GPus) and now specially-designed Tensor Processing Units (TPUs).

Data:

- Sensors and the Internet have made vast datasets available: text, images, video, …

Algorithmic advances:

- The core ideas have been around a long time: Perceptrons (1950s), backpropagation (1980s or earlier), convolutional networks (1980s), LSTMs (1990s), …
- But new ideas from 2010 onwards: better weight initialization, batch normalization, different activation functions, variants of SGD, numerous ways to avoid overfitting, new architectures,…

Freeware:

- Toolkits/APIs; Educational resources.

Money!

Applications of Deep Learning

It is excelling at 'perceptual' tasks, e.g.

- image classification;
- image segmentation;
- speech recognition;
- handwriting transcription.

But it is finding ever wider application:

- video recommendation;
- machine translation;
- text-to-speech;
- question-answering;
- autonomous driving;
- the protein folding problem (AlphaFold);
- superhuman game playing (e.g. AlphaGo).

Implementation

In this lecture:

- We will use layered, dense, feedforward neural networks for regression, binary classification and multi-class classification:
  - We'll use our two small datasets that contain **structured data** (sometimes called **tabular data**): not necessarily ideal for deep learning.
  - We'll see one example that uses images.
- This will illustrate some of the different activation functions we can use:
  - in the output layer: linear, sigmoid or softmax; and
  - in the hidden layers: sigmoid or ReLU.
- This will also introduce the Keras library.

The Keras Library

`scikit-learn` has very limited support for neural networks.

Tensorflow and PyTorch are the two main libraries that do support tensor computation, neural networks and deep learning in Python:

We will use Keras, which is a high-level API for Tensorflow, first released in 2015 by François Chollet of Google (*https://keras.io*):

- It is very high-level, making it easy to construct networks, fit models and make predictions.
- The downside is it gives less fine-grained control than TensorFlow itself. When fine-grained control is needed, you can mix in TensorFlow functions, methods and classes.
- This seems a suitable trade-off for us: our module is about AI, not the intricacies of TensorFlow.

Network Architecture: Number of Hidden Layers

- Neural network with no hidden layers is just a linear model.
- Hidden layers are needed when data is not linearly separable.
    - Try to avoid more than 2 hidden layers otherwise it will increase the model complexity.
    - For very large datasets, gradually ramp up the number of hidden layers until you start overfitting the training set.

Network Architecture: Number of Neurons in Hidden Layers

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Source: An Introduction to Neural Networks for Java, Second Edition by Jeff Heaton

Keras Concepts

Layers are the building blocks.

- To begin with, we will use **dense (*fully connected*) layers.**

The activation functions of hidden layers are open for you to choose, e.g. *sigmoid* or *ReLU*.

- But the *activation functions* of output layers are determined by the task:
  - **Regression**: linear activation function (default);
  - **Binary classification:** sigmoid activation function; and
  - **Multiclass classification:** softmax activation function.

Layers are combined into **networks:**

- Consecutive layers must be compatible: the shape of the input to one layer is the shape of the output of the preceding layer.

Keras Concepts

Once the network is built, we compile it, specifying:

A ***loss function***:

- Regression, e.g. mean-squared-error (mse);
- Binary classification, e.g. (binary) cross-entropy (binary_crossentropy);
- Multiclass classification, e.g. (categorical) cross-entropy (sparse_categorical_crossentropy if the labels are encoded as integer labels or categorical_crossentropy if the integer labels are then also one-hot encoded).

An ***optimizer***, such as SGD — but see below.

A ***list of metrics*** to monitor during training and testing:

- Regression, e.g. mean-absolute-error (mae);
- Classification, e.g. accuracy (acc).

Keras Optimizers

We know about Gradient Descent: Batch, Mini-Batch, Stochastic.

Without going into details, many other variants of Gradient Descent have been devised (e.g. `RMSprop`, `Adam`, `Nadam`, `Adagrad`, ...):

- some may have better convergence behaviour in the case of local minima;
- some may converge more quickly.

although a disadvantage is that they typically introduce further hyperparameters (e.g. momentum) in addition to learning rate.

Keras Optimizers

We will use `RMSprop` below.

- Be aware, its default *learning rate* is 0.001. This is usually OK, but in some cases you may need to change it.

- Be aware too that there is an argument called `batch_size`. Assuming we set its value to somewhere between 1 and the size of the training set then we are getting Mini-Batch Gradient Descent.

A Neural Network for Regression

For regression on structured/tabular data, we might use a network with the following architecture:

- **Input layer:** one input per feature.
- **Hidden layers:** one or more hidden layers.
  - Activation function for neurons in hidden layers can be the sigmoid function or ReLU.
- **Output layer:** just one output neuron (assuming we're predicting a single number).
  - Activation function for the output neuron should be the linear function: $g(\boldsymbol{z}) = \boldsymbol{z}$

There are also biases in each layer except the output layer — Keras will give us these 'for free'.

Example: Property Rent Prediction

We don't want too many hidden layers, nor too many neurons in each hidden layer. Why?

Let's start with this:

- An input layer with three inputs (BHK, Size, Bathrooms);
- Two hidden layers, with 64 neurons in each, and ReLU activation function;
- An output layer with a single neuron and linear activation function.

We need to scale the features. But, since we are now not using scikit-learn's `ColumnTransformers` to create a preprocessor, we need to take care of the scaling.

Example: Property Rent Prediction

```
from sklearn.model_selection import train_test_split

hr_train_x, hr_test_x, hr_train_y, hr_test_y = train_test_split(df_reduced[['BHK', 'Size', 'Bathroom']].to_numpy(),
                                                                df_reduced[['Rent']].to_numpy(),
                                                                train_size=0.8, shuffle=True, random_state=25
                                                                )
hr_train_x.shape, hr_test_x.shape
```

```
((3796, 3), (950, 3))
```

```
from tensorflow import keras

hr_model = keras.models.Sequential([
    keras.layers.InputLayer(input_shape=(3,)),
    keras.layers.Normalization(),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dense(1, activation="linear")
])
```

Example: Property Rent Prediction

```
hr_model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.001),
                 loss="mse",
                 metrics=["mae"]
                 )
```

```
history = hr_model.fit(hr_train_x, hr_train_y,
                       epochs=30,
                       batch_size=64,
                       validation_split=0.1,
                       verbose="auto"
                       )
```

```
test_loss, test_accuracy = hr_model.evaluate(hr_test_x, hr_test_y)
test_accuracy
```

```
30/30 [==============================] - 0s 1ms/step - loss: 3025089536.0000 - mae: 27686.8379
27686.837890625
```

A Neural Network for Binary Classification

For binary classification, we might use a network with the following architecture:

- **Input layer**: one input per feature.
- **Hidden layers:** one or more hidden layers.
  - Activation function for neurons in hidden layers can be sigmoid or ReLU.
- **Output layer:** just one output neuron (for binary classification).
  - Activation function for the output neuron should be the sigmoid function also. Why?

Example: Class Performance Dataset

Let's start with this:

- An input layer with 3 inputs (lec, lab, cao).
- Two hidden layers, with 64 neurons in each, and ReLU activation function.
- An output layer with a single neuron and sigmoid activation function.

We'll scale using a `Normalization` layer.

Example: Class Performance Dataset

```
inputs = Input(shape=(3,))
x = Normalization()(inputs)
x = Dense(64, activation="relu")(x)
x = Dense(64, activation="relu")(x)
outputs = Dense(1, activation="sigmoid")(x)
cs1109_model = Model(inputs, outputs)

cs1109_model.compile(optimizer=RMSprop(learning_rate=0.001), loss="binary_crossentropy", metrics=["accuracy"])
```

```
cs1109_model.fit(train_cs1109_X, train_cs1109_y, epochs=40, batch_size=32, verbose=0)
```

```
test_loss, test_acc = cs1109_model.evaluate(test_cs1109_X, test_cs1109_y)
test_acc
```

```
// 0.6666666865348816
```

Feel free to edit the code, e.g. add or remove hidden layers, change the number of neurons in the hidden layers, change ReLU to sigmoid, change from RMSprop to another optimizer, change the learning rate, change the number of epochs, or change the batch size.

A Neural Network for Multiclass Classification

For multi-class classification, we might use a network with the following architecture:

- **Input layer:** one input per feature.
- **Hidden layers:** one or more hidden layers.
  - Activation function for neurons in hidden layers can be sigmoid or ReLU.
- **Output layer:** one output neuron per class.
  - Activation function for the output neurons should be the softmax function.

Example: Iris Dataset

Let's start with this:

- An input layer with 4 inputs (petal width and length, and sepal width and length).
- Two hidden layers, with 64 neurons in each, and ReLU activation function.
- An output layer with three neurons (one for Setosa, Versicolor and Virginica) and softmax activation function.

Example: Iris Dataset

```
iris = load_iris()

iris_train, iris_test, iris_train_y, iris_test_y = train_test_split(iris.data, iris.target, train_size=0.8, random_state=25)

iris_train[0], iris.target_names[iris_train_y[0]]
# iris_train.shape, iris_test.shape
```

```
((120, 4), (30, 4))
```

```
iris_model = keras.models.Sequential([
    keras.layers.InputLayer(input_shape=(4,)),
    keras.layers.Normalization(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(3, activation="softmax")
])
```

```
Layer (type)                 Output Shape              Param #
=================================================================
normalization_4 (Normaliza   (None, 4)                 9
tion)

dense_12 (Dense)             (None, 64)                320

dense_13 (Dense)             (None, 64)                4160

dense_14 (Dense)             (None, 3)                 195

=================================================================
Total params: 4684 (18.30 KB)
Trainable params: 4675 (18.26 KB)
Non-trainable params: 9 (40.00 Byte)
_________________________________________________________________
```

Example: Iris Dataset

```
hidden1 = iris_model.layers[1]
weights, biases = hidden1.get_weights()
weights.shape, biases.shape
```

```
((4, 64), (64,))
```

```
iris_model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=0.001),
    loss="sparse_categorical_crossentropy",
    metrics="accuracy"
)
```

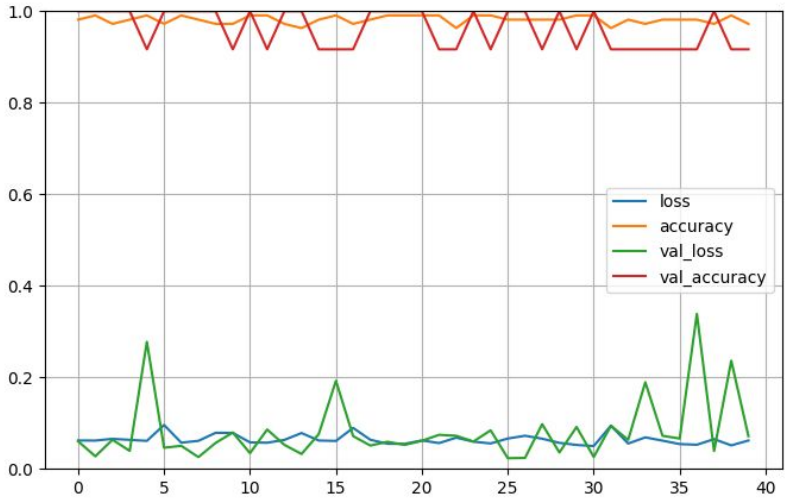
```
history = iris_model.fit(iris_train, iris_train_y,
                         epochs=40,
                         batch_size=32,
                         validation_split=0.1,
                         verbose=0
                         )
```

```
test_loss, test_accuracy = iris_model.evaluate(iris_test, iris_test_y)
test_accuracy
```

```
1/1 [==============================] - 0s 52ms/step - loss: 0.0959 - accuracy: 0.9333
0.9333333373069763
```

Example: Iris Dataset

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



Example: Iris Dataset

Note the loss function above:

- `sparse_categorical_crossentropy` for multi-class classification when the classes are integers, e.g. 0 = one kind of Iris, 1 = another kind, 2 = a third kind (which is what we have in the Iris dataset).
- `categorical_crossentropy` for multi-class classification when the classes have been one-hot encoded.
- As we've seen, `binary_crossentropy` for binary classification, where the classes will be 0 or 1.

Below, an alternative, is code that illustrates one-hot encoding the target values using the Keras function called `to_categorical`, and then using `categorical_crossentropy` for the loss function.

Example: Iris Dataset

```
train_iris_y = to_categorical(train_iris_y)
test_iris_y = to_categorical(test_iris_y)
```

```
inputs = Input(shape=(4,))
x = Normalization()(inputs)
x = Dense(64, activation="relu")(x)
x = Dense(64, activation="relu")(x)
outputs = Dense(3, activation="softmax")(x)
iris_model = Model(inputs, outputs)

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

```
iris_model.fit(train_iris_X, train_iris_y, epochs=40, batch_size=32, verbose=0)
```

```
test_loss, test_acc = iris_model.evaluate(test_iris_X, test_iris_y)
test_acc
```

```
// 0.8999999761581421
```

Example: Iris Dataset

Observations:

- Neural networks are often not the best-performing approaches for structured data.
- And, sure enough, the results here are not great. Of course, there is a lot we can tweak to see if we can improve the results.
- But, instead, let's switch to an image processing example.

Example: Fashion MNIST Dataset

Fashion MNIST is also a classic dataset for multi-class classification.

- The task is classification of fashion items.
  - Features: 28 pixel by 28 pixel grayscale images of fashion items. The values are integers in [0, 255].
  - Classes: ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"].

- Dataset: 70,000 images, so we can safely use holdout, and it is already partitioned:
  - 60,000 training images; 10,000 test images.

Example: Fashion MNIST Dataset

We don't really need scikit-learn pipelines this time.

But we do need to reshape:

- Our training data is in a 3D array of shape (60000, 28, 28).
- We change it to a 2D array of shape (60000, 28 * 28).
    - This 'flattens' the images.
    - When working with images, it is often better not to do this.
- Similarly, the test data.

Example: Fashion MNIST Dataset

We will do a three-layer network:

- One hidden layer with 300 neurons, using the ReLU activation function.
- Second hidden layer with 100 neurons, using the ReLU activation function.
- The output layer will have 10 neurons, one per class, and will use the softmax activation function.

The features (pixel values) are all in the same range [0, 255], so we do not need to standardize using a Normalization layer.

But it is a bad idea to feed into a neural network values that are much larger than the initial weights, so we will rescale to by dividing by 255. We can do this using a ***Rescaling*** layer.

Example: Fashion MNIST Dataset

```
(X_train_full, Y_train_full), (X_test, Y_test) = keras.datasets.fashion_mnist.load_data()
X_train_full.shape, X_test.shape
```

```
((60000, 28, 28), (10000, 28, 28))
```

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
class_names[Y_train_full[0]]
```

```
'Ankle boot'
```

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Rescaling(scale=1./255),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 rescaling_4 (Rescaling)     (None, 784)               0

 dense_12 (Dense)            (None, 300)               235500

 dense_13 (Dense)            (None, 100)               30100

 dense_14 (Dense)            (None, 10)                1010

=================================================================
Total params: 266610 (1.02 MB)
Trainable params: 266610 (1.02 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Example: Fashion MNIST Dataset

```
hidden1 = model.layers[2]
weights, biases = hidden1.get_weights()
weights.shape, biases.shape
```

```
((784, 300), (300,))
```

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="SGD",
              metrics=["accuracy"]
              )
```

```
history = model.fit(X_train_full, Y_train_full,
                    epochs=30,
                    validation_split=0.1   # by defalut shuffle is set to True
                    )
```

Example: Fashion MNIST Dataset

```
test_loss, test_accuracy = model.evaluate(X_test, Y_test)
test_accuracy
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3565 - accuracy: 0.8846
0.8845999836921692
```

```
y_proba = model.predict(X_test[0:5])
y_proba.round(2)
```

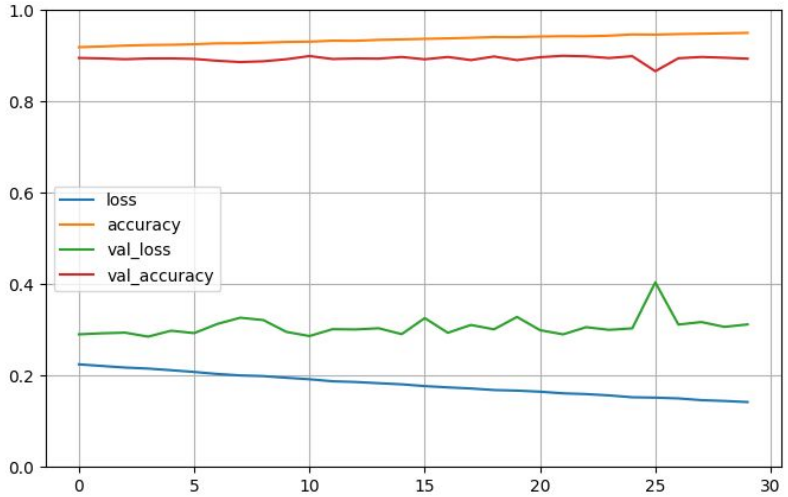
```
1/1 [==============================] - 0s 34ms/step
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.81, 0.  , 0.  , 0.  , 0.  , 0.  , 0.19, 0.  , 0.  , 0.  ]],
      dtype=float32)
```

```
import numpy as np

np.array(class_names)[np.argmax(model.predict(X_test[0:3]), axis=-1)]
```

```
1/1 [==============================] - 0s 21ms/step
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Example: Fashion MNIST Dataset

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



Remarks on Computer Vision Problems

In the 1960s, 70s, 80s and to some extent 90s, the typical pipeline for a computer vision (or image processing) system was as follows:

- There would be a module that would extract features from the images.
    - These features would have been carefully hand-designed.
    - They might include edges detected by some edge detection algorithm, for example. (If you are interested, look up SIFT or SURF or HOG.)
- Then these features would be fed into a typical learning algorithm, e.g. logistic regression.

Remarks on Computer Vision Problems

Notice how different life is now — when using neural networks.

- There's no extraction of hand-crafted features. We feed in the raw pixel values (or, lightly-processed pixel values, e.g. scaled values).
- It is the layers of the neural network that automatically discover the features, and the final layer that makes the classification.
- The dense layers are only one possibility.
    - Computer vision (image processing) more often also uses convolutional layers, pooling layers, batch normalization layers, and so on. We may study them in coming lectures.

Concluding Remarks

- A few decisions are constrained: number of inputs; number of output neurons; activation function of output neurons; and (to some extent) loss function.

- But there are numerous hyperparameters (and even more to come!)

    - Even making a good guess at them is more art than science, although this is changing.

    - On the other hand, grid search or randomized search will make things even slower than they already are — and we still have to specify some sensible values for them to search through.

- There is a considerable risk of overfitting.

Next lecture

Training Deep Neural Network

19th October 2023