Artificial Neural Networks And Deep Learning

Laboratory

Christian Rossi

Academic Year 2024-2025

Abstract

Neural networks have matured into flexible and powerful non-linear data-driven models, effectively tackling complex tasks in both science and engineering. The emergence of deep learning, which utilizes neural networks to learn optimal data representations alongside their corresponding models, has significantly advanced this paradigm.

In the course, we will explore various topics in depth. We will begin with the evolution from the Perceptron to modern neural networks, focusing on the feedforward architecture. The training of neural networks through backpropagation and algorithms like Adagrad and Adam will be covered, along with best practices to prevent overfitting, including cross-validation, stopping criteria, weight decay, dropout, and data resampling techniques.

The course will also delve into specific applications such as image classification using neural networks, and we will examine recurrent neural networks and related architectures like sparse neural autoencoders. Key theoretical concepts will be discussed, including the role of neural networks as universal approximation tools, and challenges like vanishing and exploding gradients.

We will introduce the deep learning paradigm, highlighting its distinctions from traditional machine learning methods. The architecture and breakthroughs of convolutional neural networks (CNNs) will be a focal point, including their training processes and data augmentation strategies.

Furthermore, we will cover structural learning and long-short term memory (LSTM) networks, exploring their applications in text and speech processing. Topics such as autoencoders, data embedding techniques like word2vec, and variational autoencoders will also be addressed.

Finally, we will discuss transfer learning with pre-trained deep models, examine extended models such as fully convolutional CNNs for image segmentation (e.g., U-Net) and object detection methods (e.g., R-CNN, YOLO), and explore generative models like generative adversarial networks (GANs).

Contents

CHAPTER 1

Feed Forward Neural Networks

1.1 Notebook setup

We start by setting a standard sees in order to have a replicable execution. We also set some environmental variables to disable warnings and adjust other modules settings.

```
seed = 42

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=Warning)

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
os.environ['PYTHONHASHSEED'] = str(seed)
os.environ['MPLCONFIGDIR'] = os.getcwd() + '/configs/'
```

Import the necessary modules used to handle data and numbers.

```
import logging
import random
import numpy as np

np.random.seed(seed)
random.seed(seed)
%matplotlib inline
```

Import tensorflow and keras to manage Neural Networks

```
import tensorflow as tf
from tensorflow import keras as tfk
from tensorflow.keras import layers as tfkl

tf.random.set_seed(seed)
tf.compat.v1.set_random_seed(seed)

tf.autograph.set_verbosity(0)
tf.get_logger().setLevel(logging.ERROR)
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

print(tf.__version__)
```

Lastly, import pandas (for tabular data), seaborn, and matplotlib (for data visualization).

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↪ f1_score, confusion_matrix
from sklearn.model_selection import train_test_split

sns.set(font_scale=1.4)
sns.set_style('white')
plt.rc('font', size=14)
%matplotlib inline
```

1.2 Data analysis

We start our analysis by importing the well-known Iris dataset and printing a description of it.

```
data = load_iris()
print(data.DESCR)
```

Instead of obtaining a general description of the dataset, we may want to inspect a table with the features of the elements in the dataset. We could do this as follows:

```
iris_dataset = pd.DataFrame(data.data, columns=data.feature_names)
print('Iris dataset shape', iris_dataset.shape)
iris_dataset.head(10)
```

We can now print all the statistical data of the given dataset with the following command:

```
print('Iris dataset shape', iris_dataset.shape)
iris_dataset.describe()
```

To find how the dataset has beed divided and classified, and also retrieve the labales of the elements in the dataset we can use the following command:

```
target = data.target
print('Target shape', target.shape)
unique, count = np.unique(target, return_counts=True)
print('Target labels:', unique)
for u in unique:
    print(f'Class {unique[u]} has {count[u]} samples')
```

Lastly, we can check the distribution of the samples over the space.

```
plot_dataset = iris_dataset.copy()
```

```
plot_dataset["Species"] = target
sns.pairplot(plot_dataset, hue="Species", palette="tab10", markers=["o", "s",
    ↪ "D"])
plt.show()
del plot_dataset
```

In particular, in this case we can see that there is a linear decision boundary in many dimensions.

1.3 Data processing

Since we have only the data from the given dataser, but after the training we have to test the genrated model to test the correctness, we need to split the dataset in multiple parts:. Also, we need a validation set to validate the given model.

```
# Split the dataset into a combined training and validation set, and a separate
# test set
X_train_val, X_test, y_train_val, y_test = train_test_split(
    iris_dataset,
    target,
    test_size=20,
    random_state=seed,
    stratify=target
)

# Further split the combined training and validation set into a training set and
# a validation set
X_train, X_val, y_train, y_val = train_test_split(
    X_train_val,
    y_train_val,
    test_size=20,
    random_state=seed,
    stratify=y_train_val
)

# Print the shapes of the resulting sets
print('Training set shape:\t', X_train.shape, y_train.shape)
print('Validation set shape:\t', X_val.shape, y_val.shape)
print('Test set shape:\t\t', X_test.shape, y_test.shape)
```

After dividing the dataset in training, validation, and test sets we need now to normalize them. To do so we use the entire dataset to find maximum and minimum, and then we normalize all other samples with respect to these values. The normalization is mainly used to speed up the training process. With the minmax applied in this case we have all values constrained between zero and one.

```
max_df = X_train.max()
min_df = X_train.min()

X_train = (X_train - min_df) / (max_df - min_df)
X_val = (X_val - min_df) / (max_df - min_df)
```

```
X_test = (X_test - min_df) / (max_df - min_df)

X_train.describe()
```

Then, since we are dealing with a classification task in which we want to know the exact class of each element, we may use perceptron with one hot encoding.

```
y_train = tfk.utils.to_categorical(y_train, num_classes=len(unique))
y_val = tfk.utils.to_categorical(y_val, num_classes=len(unique))
y_test = tfk.utils.to_categorical(y_test, num_classes=len(unique))

print('Training set target shape:\t', y_train.shape)
print('Validation set target shape:\t', y_val.shape)
print('Test set target shape:\t\t', y_test.shape)
```

1.4 Model definition

We need to find the number of features and the number of classis for our Neural Network.

```
input_shape = X_train.shape[1:]
output_shape = y_train.shape[1]
```

We set also the other parametes such as: batch size (number of samples processed in each training iteration), number of epochs (times the entire dataset is passed through the network during training), and learning rate (step size for updating the model's weights).

```
batch_size = 16
epochs = 500
learning_rate = 0.001
```

We can finally build the model.

```
def build_model(input_shape=input_shape, output_shape=output_shape,
    ↪ learning_rate=learning_rate, seed=seed):

# Fix randomness
tf.random.set_seed(seed)

# Build the neural network layer by layer
inputs = tfkl.Input(shape=input_shape, name='Input')

# Add hidden layer with ReLU activation
x = tfkl.Dense(units=16, name='Hidden')(inputs)
x = tfkl.Activation('relu', name='HiddenActivation')(x)

# Add output layer with softmax activation
x = tfkl.Dense(units=output_shape, name='Output')(x)
outputs = tfkl.Activation('softmax', name='Softmax')(x)

# Connect input and output through the Model class
model = tfk.Model(inputs=inputs, outputs=outputs, name='FeedforwardNeuralNetwork')
```

```
# Compile the model with loss, optimizer, and metrics
loss = tfk.losses.CategoricalCrossentropy()
optimizer = tfk.optimizers.Adam(learning_rate)
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optimizer, metrics=metrics)

# Return the model
return model
```

Now we can finally display the data about the new model we have created.

```
model = build_model()
model.summary(expand_nested=True, show_trainable=True)
tfk.utils.plot_model(model, expand_nested=True, show_trainable=True,
    ↪ show_shapes=True, dpi=70)
```

1.5 Model training

We can now train the model.

```
# Train the model and store the training history
history = model.fit(
    x=X_train,
    y=y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_data=(X_val, y_val)
).history

# Calculate the final validation accuracy
final_val_accuracy = round(history['val_accuracy'][-1] * 100, 2)

# Save the trained model to a file with the accuracy included in the filename
model_filename = f'Iris_Feedforward_{final_val_accuracy}.keras'
model.save(model_filename)

# Delete the model to free up memory resources
del model
```

We can now plot the results of the training of the saved model.

```
# Create a figure with two vertically stacked subplots
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(15, 6), sharex=True)

# Plot training and validation loss
ax1.plot(history['loss'], label='Training loss', alpha=.8)
ax1.plot(history['val_loss'], label='Validation loss', alpha=.8)
ax1.set_title('Loss')
ax1.legend()
ax1.grid(alpha=.3)
```

```
# Plot training and validation accuracy
ax2.plot(history['accuracy'], label='Training accuracy', alpha=.8)
ax2.plot(history['val_accuracy'], label='Validation accuracy', alpha=.8)
ax2.set_title('Accuracy')
ax2.legend()
ax2.grid(alpha=.3)

# Adjust the layout and display the plot
plt.tight_layout()
plt.subplots_adjust(right=0.85)
plt.show()
```

1.6 Model prediction

To make prediction with a model we start by loading it into memory.

```
# Load the saved model
model = tfk.models.load_model('Iris_Feedforward_95.0.keras')

# Display a summary of the model architecture
model.summary(expand_nested=True, show_trainable=True)

# Plot the model architecture
tfk.utils.plot_model(model, expand_nested=True, show_trainable=True,
    ↪ show_shapes=True, dpi=70)
```

The, we can finally make prediction and plot the confusion matrix.

```
# Predict class probabilities and get predicted classes
train_predictions = model.predict(X_train, verbose=0)
train_predictions = np.argmax(train_predictions, axis=-1)

# Extract ground truth classes
train_gt = np.argmax(y_train, axis=-1)

# Calculate and display training set accuracy
train_accuracy = accuracy_score(train_gt, train_predictions)
print(f'Accuracy score over the train set: {round(train_accuracy, 4)}')

# Calculate and display training set precision
train_precision = precision_score(train_gt, train_predictions, average='weighted')
print(f'Precision score over the train set: {round(train_precision, 4)}')

# Calculate and display training set recall
train_recall = recall_score(train_gt, train_predictions, average='weighted')
print(f'Recall score over the train set: {round(train_recall, 4)}')

# Calculate and display training set F1 score
train_f1 = f1_score(train_gt, train_predictions, average='weighted')
print(f'F1 score over the train set: {round(train_f1, 4)}')
```

```

# Compute the confusion matrix
cm = confusion_matrix(train_gt, train_predictions)

# Create labels combining confusion matrix values
labels = np.array([f"{num}" for num in cm.flatten()]).reshape(cm.shape)

# Plot the confusion matrix with class labels
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=labels, fmt='', xticklabels=['Setosa', 'Versicolor',
    ↪ 'Virginica'], yticklabels=['Setosa', 'Versicolor', 'Virginica'],
    ↪ cmap='Blues')
plt.xlabel('True labels')
plt.ylabel('Predicted labels')
plt.show()
```

CHAPTER 2

Overfitting and regularization