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 dataset, 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

2.1 Notebook setup

We import all the necessary libraries and we set the seed to remov randomness.

```
# Fix randomness and hide warnings
seed = 42

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

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

import numpy as np
np.random.seed(seed)

import logging

import random
random.seed(seed)

# Import tensorflow
import tensorflow as tf
from tensorflow import keras as tfk
from tensorflow.keras import layers as tfkl
tf.autograph.set_verbosity(0)
tf.get_logger().setLevel(logging.ERROR)
tf.random.set_seed(seed)
print(f"TensorFlow version {tf.__version__}")

# Import other libraries
import pandas as pd
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
plt.rc('font', size=14)
%matplotlib inline
import seaborn as sns
sns.set(font_scale=1.4)
sns.set_style('white')
from pandas.plotting import scatter_matrix
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import mean_squared_error
```

This time we decided to import the dataset about California housing.

2.2 Data analysis

We start our analysis by importing the California housing dataset and printing a description of it.

```
data = fetch_california_housing(as_frame=True)
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:

```
housing_dataset = data.frame
print('California Housing dataset shape', housing_dataset.shape)
housing_dataset.head(10)
```

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

```
print('California Housing dataset shape', housing_dataset.shape)
housing_dataset.describe()
```

We can check the structure of the datase:

```
# Set Seaborn theme with white grid style
sns.set_theme(style="whitegrid")

# Define a harmonious colour palette
palette = sns.color_palette("viridis", n_colors=9)

# Generate the figure and axes for subplots
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(16, 10), sharex=False,
    ↪ sharey='row')

# Flatten the axes array for easier access
axes = axes.flatten()

# List the columns of the dataset
columns = housing_dataset.columns
```

```
# Create histograms with customised colours for each feature
for i, column in enumerate(columns):
    if i < len(axes): # Ensure the number of subplots is not exceeded
        sns.histplot(
            data=housing_dataset,
            x=column,
            kde=True,
            ax=axes[i],
            color=palette[i],
            edgecolor='white',
            alpha=0.7
        )
        axes[i].set_title(column.capitalize(), fontsize=14, fontweight='bold',
            ↪ color='#333333')
        axes[i].set_xlabel('') # Remove x-axis labels for all subplots
        if i % 3 != 0:
            axes[i].set_ylabel('') # Remove y-axis labels except for the first
                ↪ column
        axes[i].tick_params(axis='both', which='major', labelsize=10)

        # Improve label readability by limiting the number of ticks
        axes[i].xaxis.set_major_locator(ticker.MaxNLocator(6))
        axes[i].yaxis.set_major_locator(ticker.MaxNLocator(6))

# Remove any extra subplots not used
for i in range(len(columns), len(axes)):
    fig.delaxes(axes[i])

# Enhance layout with proper padding
plt.tight_layout(pad=3.0, h_pad=2.5, w_pad=2.0)
fig.suptitle('Distribution of Housing Dataset Features', fontsize=20,
    ↪ fontweight='bold', y=1.02, color='#444444')
plt.show()
```

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

```
housing_dataset.plot(
    kind="scatter",
    x="Longitude",
    y="Latitude",
    c="MedHouseVal",
    cmap="jet",
    colorbar=True,
    legend=True,
    sharex=False,
    figsize=(10,7),
    s=housing_dataset['Population']/100,
    label="Population",
    alpha=0.7
    )
plt.show()
```

2.3 Data processing

Since we have only the data from the given dataset, 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.

```
features = housing_dataset.columns[:-1]
target = housing_dataset.columns[-1]

# 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(
    housing_dataset[features],
    housing_dataset[target],
    test_size = 0.1,
    random_state=seed
)

# 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 = len(X_test), # Ensure validation set size matches test set size
    random_state=seed
)

# 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()
max_target = y_train.max()
min_df = X_train.min()
min_target = y_train.min()

# Normalise the dataset splits in the range [0,1]
X_train_val = (X_train_val - min_df) / (max_df - min_df)
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)

# Normalise the dataset splits in the range [0,1]
y_train_val = (y_train_val - min_target) / (max_target - min_target)
y_train = (y_train - min_target)/(max_target - min_target)
```

```
y_val = (y_val - min_target)/(max_target - min_target)
y_test = (y_test - min_target)/(max_target - min_target)
```

2.4 Model definition

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

```
input_shape = X_train.shape[1:]
batch_size = 64
epochs = 1000
metadata = {}
```

We can finally build the model.

```
def build_model(input_shape, learning_rate=1e-3, l2_lambda=0, dropout_rate=0,
    ↪ name='', seed=seed):

    # Set random seed for reproducibility
    tf.random.set_seed(seed)

    # Initialise weights and regulariser
    initialiser = tfk.initializers.GlorotNormal(seed=seed)
    regulariser = tfk.regularizers.l2(l2_lambda)

    # Input layer
    input_layer = tfkl.Input(shape=input_shape, name='Input')

    # Hidden layers with ReLU activations
    x = tfkl.Dense(units=256, name='HiddenDense1',
        ↪ kernel_initializer=initialiser)(input_layer)
    x = tfkl.Activation('relu', name='HiddenActivation1')(x)
    x = tfkl.Dense(units=256, name='HiddenDense2',
        ↪ kernel_initializer=initialiser)(x)
    x = tfkl.Activation('relu', name='HiddenActivation2')(x)

    # Dropout layer if specified
    if dropout_rate > 0:
    x = tfkl.Dropout(dropout_rate, seed=seed, name='Dropout')(x)

    # Output layer with optional L2 regularisation
    if l2_lambda > 0:
    output_layer = tfkl.Dense(units=1, kernel_initializer=initialiser,
        ↪ kernel_regularizer=regulariser, name='Output')(x)
    else:
    output_layer = tfkl.Dense(units=1, kernel_initializer=initialiser,
        ↪ name='Output')(x)
```

```
    # Linear output activation
    output_activation = tfkl.Activation('linear',
        ↪ name='OutputActivation')(output_layer)

    # Connect input and output through the Model class
    model = tfk.Model(inputs=input_layer, outputs=output_activation, name=name)

    # Compile the model with Adam optimiser and MSE loss
    opt = tfk.optimizers.Adam(learning_rate)
    loss = tfk.losses.MeanSquaredError()
    mtr = ['mse']
    model.compile(loss=loss, optimizer=opt, metrics=mtr)

    # Return the compiled model
    return model
```

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

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

2.5 Auxiliary functions

We define some auxiliary function usefule for our analysis:

```
# Define a function for plotting training and validation Mean Squared Error (MSE)
    ↪ histories.
def plot_histories(metadata, training=False, baseline=False, show_all=False):

    print('VALIDATION MSE')
    plt.figure(figsize=(21, 9))

    # Iterate through different models stored in metadata.
    for model in list(metadata.keys()):

        # Skip the baseline model if baseline flag is False.
        if model == 'Baseline' and not baseline:
            continue

        # Retrieve relevant information from the metadata dictionary.
        history = metadata[model]['history']
        patience = metadata[model]['patience']
        color = metadata[model]['color']
        val_score = metadata[model]['val_score']
        print('%s: %.4f' % (model, val_score))

        # Plot training and validation MSE histories with or without patience
            ↪ handling.
        if patience != 0:
```

```
            if training:
                plt.plot(
                    history['mse'][:-patience],
                    alpha=0.5,
                    color=color
                )
                if show_all:
                    plt.plot(
                        np.arange(len(history['mse']) - patience - 1,
                            ↪ len(history['mse'])),
                        history['mse'][-patience - 1:],
                        alpha=0.2,
                        color=color
                    )

            plt.plot(
                history['val_mse'][:-patience],
                label=model,
                alpha=0.9,
                color=color
            )
            if show_all:
                plt.plot(
                    np.arange(len(history['val_mse']) - patience - 1,
                        ↪ len(history['val_mse'])),
                    history['val_mse'][-patience - 1:],
                    alpha=0.2,
                    color=color
                )

        else:
            if training:
                plt.plot(
                    history['mse'],
                    alpha=0.5,
                    color=color
                )
            plt.plot(
                history['val_mse'],
                label=model,
                alpha=0.9,
                color=color
            )

        # Set y-axis limits and add labels, legends, and grid.
        if training:
            plt.ylim(0.004, 0.0225)
        else:
            plt.ylim(0.01, 0.025)
        plt.title('Mean Squared Error')
        plt.legend(loc='upper right')
```

```
        plt.grid(alpha=0.3)

    # Display the plot.
    plt.show()


# Define a function for plotting residuals and model predictions
def plot_residuals(model, data, labels):
    # Sort the data and labels based on the 'labels' column
    data['sort'] = labels
    data = data.sort_values(by=['sort'])
    labels = np.expand_dims(data['sort'], 1)
    data.drop(['sort'], axis=1, inplace=True)

    # Make predictions using the model
    y_pred = model.predict(data, verbose=0)

    # Calculate squared errors and mean squared error (MSE)
    squared_errors = (labels - y_pred)**2
    mse = np.mean(squared_errors).astype('float32')
    print('MSE: %.4f' % mse)

    # Set up plotting styles and create a scatter plot
    mpl.rcParams.update(mpl.rcParamsDefault)
    sns.set(font_scale=1.1, style=None, palette='Set1')
    plt.figure(figsize=(21, 5))

    # Plot true labels in red and model predictions in blue
    plt.scatter(np.arange(len(labels)), labels, label='True', color='#d62728',
        ↪ alpha=0.7, s=8)

    # Add vertical lines to represent residuals
    for i in range(len(labels)):
        if labels[i] >= y_pred[i]:
            plt.vlines(i, y_pred[i], labels[i], alpha=0.2, linewidth=0.5)
        else:
            plt.vlines(i, labels[i], y_pred[i], alpha=0.2, linewidth=0.5)

    plt.scatter(np.arange(len(y_pred)), y_pred, label='Prediction',
        ↪ color='#1f77b4', s=8)

    # Add legends, grid, and set y-axis limits
    plt.legend()
    plt.grid(alpha=0.3)
    plt.ylim((-0.1, 1.1))
    plt.show()


def evaluate_and_plot_model(model, X_val, y_val, X_test, y_test, metadata,
    ↪ history, patience, model_name, color, plot_baseline=False):
    # Calculate validation Mean Squared Error (MSE) for the model
```

```
    val_predictions = np.squeeze(model.predict(X_val, verbose=0))
    val_squared_errors = (y_val - val_predictions)**2
    val_mse = np.mean(val_squared_errors).astype('float32')

    # Calculate test Mean Squared Error (MSE) for the model
    test_predictions = np.squeeze(model.predict(X_test, verbose=0))
    test_squared_errors = (y_test - test_predictions)**2
    test_mse = np.mean(test_squared_errors).astype('float32')

    # Add model metadata to the dictionary
    metadata[model_name] = {
        'model': model,
        'history': history,
        'color': color,
        'patience': patience,
        'val_score': val_mse,
        'test_score': test_mse
    }

    # Plot histories
    plot_histories(metadata, baseline=plot_baseline)

    return metadata
```

2.6 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,
    validation_data = (X_val, y_val),
    batch_size = batch_size,
    epochs = epochs,
    verbose=0
).history
```

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

```
# Set the number of initial data points to ignore
ignore = 0

# Create a figure for loss visualization
plt.figure(figsize=(21, 4))

# Plot training and validation loss
plt.plot(history['loss'][ignore:], label='Training loss', alpha=.2,
    ↪ color='#1f77b4')
plt.plot(history['val_loss'][ignore:], label='Validation loss', alpha=.8,
    ↪ color='#1f77b4')
```

```
plt.title('Loss')
plt.legend()
plt.grid(alpha=.3)

# Create a figure for Mean Squared Error visualization
plt.figure(figsize=(21, 4))

# Plot training and validation MSE
plt.plot(history['mse'][ignore:], label='Training MSE', alpha=.2, color='#1f77b4')
plt.plot(history['val_mse'][ignore:], label='Validation MSE', alpha=.8,
    ↪ color='#1f77b4')
plt.title('Mean Squared Error')
plt.legend()
plt.grid(alpha=.3)

# Display the plots
plt.show()

metadata = evaluate_and_plot_model(
    model,
    X_val,
    y_val,
    X_test,
    y_test,
    metadata,
    history=history,
    patience=0,
    model_name='Baseline',
    color='#1f77b4',
    plot_baseline=True
    )

# Calculate the final validation mse
final_val_mse = round(history['val_mse'][-1], 4)

# Save the trained model to a file with the mse included in the filename
model_filename = f'Feedforward_{final_val_mse}.keras'
model.save(model_filename)

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

2.7 Regularization

2.7.1 Early stopping

Early stopping may be applied in the following way:

```
es_model = build_model(input_shape)
es_model.summary(expand_nested=True, show_trainable=True)
```

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

# Define the patience value for early stopping
patience = 100

# Create an EarlyStopping callback
early_stopping = tfk.callbacks.EarlyStopping(
    monitor='val_mse',
    mode='min',
    patience=patience,
    restore_best_weights=True
)

# Store the callback in a list
callbacks = [early_stopping]

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

# Set the number of initial data points to ignore
ignore = 0

# Create a figure for loss visualization
plt.figure(figsize=(21, 4))

# Plot training and validation loss for early-stopped model
plt.plot(es_history['loss'][ignore:], label='Training loss', alpha=.2,
    ↪ color='#ff7f0e')
plt.plot(es_history['val_loss'][ignore:], label='Validation loss', alpha=.8,
    ↪ color='#ff7f0e')
plt.title('Loss')
plt.legend()
plt.grid(alpha=.3)

# Create a figure for Mean Squared Error visualization
plt.figure(figsize=(21, 4))

# Plot training and validation MSE for early-stopped model
plt.plot(es_history['mse'][ignore:], label='Training MSE', alpha=.2,
    ↪ color='#ff7f0e')
plt.plot(es_history['val_mse'][ignore:], label='Validation MSE', alpha=.8,
    ↪ color='#ff7f0e')
plt.title('Mean Squared Error')
```

```
plt.legend()
plt.grid(alpha=.3)

# Display the plots
plt.show()

metadata = evaluate_and_plot_model(
    es_model,
    X_val,
    y_val,
    X_test,
    y_test,
    metadata,
    history=es_history,
    patience=patience,
    model_name='Baseline (es)',
    color='#ff7f0e',
    plot_baseline=True
    )


# Calculate the final validation mse
final_val_mse = round(es_history['val_mse'][-(patience+1)], 4)
print(final_val_mse)

# Save the trained model to a file with the mse included in the filename
es_model_filename = f'Feedforward_es_{final_val_mse}.keras'
es_model.save(es_model_filename)

# Delete the model to free up memory resources
del es_model

plot_histories(metadata, baseline=False)
```

2.7.2 Ridge regression

Ridge regression may be applied in the following way:

```
l2_lambda = 5e-4

l2_model = build_model(input_shape, l2_lambda=l2_lambda)
l2_model.summary(expand_nested=True, show_trainable=True)
tfk.utils.plot_model(l2_model, expand_nested=True, show_trainable=True,
    ↪ show_shapes=True, dpi=70)

# Create an EarlyStopping callback
early_stopping = tfk.callbacks.EarlyStopping(
    monitor='val_mse',
    mode='min',
    patience=patience,
    restore_best_weights=True
```

```
)

# Store the callback in a list
callbacks = [early_stopping]

# Train the model and store the training history
l2_history = l2_model.fit(
    x = X_train,
    y = y_train,
    validation_data = (X_val, y_val),
    batch_size = batch_size,
    epochs = epochs,
    callbacks = callbacks
).history

# Set the number of initial data points to ignore
ignore = 0

# Create a figure for loss visualization
plt.figure(figsize=(21, 4))

# Plot training and validation loss for L2-regularized model
plt.plot(l2_history['loss'][ignore:], label='Training loss', alpha=.2,
    ↪ color='#2ca02c')
plt.plot(l2_history['val_loss'][ignore:], label='Validation loss', alpha=.8,
    ↪ color='#2ca02c')
plt.title('Loss')
plt.legend()
plt.grid(alpha=.3)

# Create a figure for Mean Squared Error visualization
plt.figure(figsize=(21, 4))

# Plot training and validation MSE for L2-regularized model
plt.plot(l2_history['mse'][ignore:], label='Training MSE', alpha=.2,
    ↪ color='#2ca02c')
plt.plot(l2_history['val_mse'][ignore:], label='Validation MSE', alpha=.8,
    ↪ color='#2ca02c')
plt.title('Mean Squared Error')
plt.legend()
plt.grid(alpha=.3)

# Display the plots
plt.show()

metadata = evaluate_and_plot_model(
    l2_model,
    X_val,
    y_val,
    X_test,
    y_test,
```

```
    metadata,
    history=l2_history,
    patience=patience,
    model_name='Ridge',
    color='#2ca02c'
    )

# Calculate the final validation mse
final_val_mse = round(l2_history['val_mse'][-(patience+1)], 4)

# Save the trained model to a file with the mse included in the filename
l2_model_filename = f'Feedforward_l2_{final_val_mse}.keras'
l2_model.save(l2_model_filename)

# Delete the model to free up memory resources
del l2_model
```

2.7.3 Dropout

Dropout may be applied in the following way:

```
dropout_rate = 1/2

dropout_model = build_model(input_shape, dropout_rate=dropout_rate)
dropout_model.summary(expand_nested=True, show_trainable=True)
tfk.utils.plot_model(dropout_model, expand_nested=True, show_trainable=True,
    ↪ show_shapes=True, dpi=70)

# Create an EarlyStopping callback
early_stopping = tfk.callbacks.EarlyStopping(
    monitor='val_mse',
    mode='min',
    patience=patience,
    restore_best_weights=True
)

callbacks = [early_stopping]

dropout_history = dropout_model.fit(
    x = X_train,
    y = y_train,
    validation_data = (X_val, y_val),
    batch_size = batch_size,
    epochs = epochs,
    callbacks = callbacks
).history

# Set the number of initial data points to ignore
ignore = 0

# Create a figure for loss visualization
```

```
plt.figure(figsize=(21, 4))

# Plot training and validation loss for model with dropout layers
plt.plot(dropout_history['loss'][ignore:], label='Training loss', alpha=.2,
    ↪ color='#9467bd')
plt.plot(dropout_history['val_loss'][ignore:], label='Validation loss', alpha=.8,
    ↪ color='#9467bd')
plt.title('Loss')
plt.legend()
plt.grid(alpha=.3)

# Create a figure for Mean Squared Error visualization
plt.figure(figsize=(21, 4))

# Plot training and validation MSE for model with dropout layers
plt.plot(dropout_history['mse'][ignore:], label='Training MSE', alpha=.2,
    ↪ color='#9467bd')
plt.plot(dropout_history['val_mse'][ignore:], label='Validation MSE', alpha=.8,
    ↪ color='#9467bd')
plt.title('Mean Squared Error')
plt.legend()
plt.grid(alpha=.3)

# Display the plots
plt.show()

metadata = evaluate_and_plot_model(
    dropout_model,
    X_val,
    y_val,
    X_test,
    y_test,
    metadata,
    history=dropout_history,
    patience=patience,
    model_name='Dropout',
    color='#9467bd'
    )


# Calculate the final validation mse
final_val_mse = round(dropout_history['val_mse'][-(patience+1)], 4)

# Save the trained model to a file with the mse included in the filename
dropout_model_filename = f'Feedforward_do_{final_val_mse}.keras'
dropout_model.save(dropout_model_filename)

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

2.7.4 Comparison

We can compare the models in the following way.

```
# Compare all the trainings
plot_histories(metadata, baseline=True)

# Create a bar chart for validation MSE of different models
plt.figure(figsize=(21, 6))
for m in metadata.keys():
    plt.bar(m, metadata[m]['val_score'], color=metadata[m]['color'], alpha=.8)
plt.ylim(0.01, .02)
plt.title('Validation MSE')
plt.grid(alpha=.3, axis='y')
plt.show()
```

2.8 Model prediction

To make prediction with a model we start by loading it into memory and plot the residuals.

```
dropout_model = tfk.models.load_model('Feedforward_do_0.0105.keras')

# Evaluate and plot performance on the training data
print('Train Performance')
plot_residuals(dropout_model, X_train.copy(), y_train.copy())

# Evaluate and plot performance on the validation data
print('Validation Performance')
plot_residuals(dropout_model, X_val.copy(), y_val.copy())

# Evaluate and plot performance on the test data
print('Test Performance')
plot_residuals(dropout_model, X_test.copy(), y_test.copy())
```

2.9 Cross validation

We can validate the models with k-fold cross validation:

```
# Define the number of folds for cross-validation
num_folds = 10

# Initialize lists to store training histories, scores, and best epochs
histories = []
scores = []
best_epochs = []

# Create a KFold cross-validation object
kfold = KFold(n_splits=num_folds, shuffle=True, random_state=seed)
```

```
# Loop through each fold
for fold_idx, (train_idx, valid_idx) in enumerate(kfold.split(X_train_val,
    ↪ y_train_val)):

    print("Starting training on fold num: {}".format(fold_idx+1))

    # Build a new dropout model for each fold
    k_model = build_model(input_shape, dropout_rate=dropout_rate)

    # Create an EarlyStopping callback
    early_stopping = tfk.callbacks.EarlyStopping(
        monitor='val_mse',
        mode='min',
        patience=patience,
        restore_best_weights=True
    )

    callbacks = [early_stopping]

    # Train the model on the training data for this fold
    history = k_model.fit(
    x = X_train_val.iloc[train_idx],
    y = y_train_val.iloc[train_idx],
    validation_data=(X_train_val.iloc[valid_idx], y_train_val.iloc[valid_idx]),
    batch_size = batch_size,
    epochs = epochs,
    callbacks = callbacks,
    verbose = 0
    ).history

    # Evaluate the model on the validation data for this fold
    score = k_model.evaluate(X_train_val.iloc[valid_idx],
        ↪ y_train_val.iloc[valid_idx], verbose=0)
    scores.append(score[1])

    # Calculate the best epoch for early stopping
    best_epoch = len(history['loss']) - patience
    best_epochs.append(best_epoch)

    # Store the training history for this fold
    histories.append(history)

    # Define a list of colors for plotting
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b',
    ↪ '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']

# Print mean and standard deviation of MSE scores
print("MSE")
print(f"Mean: {np.mean(scores).round(4)}\nStd: {np.std(scores).round(4)}")

# Create a figure for MSE visualization
```

```
plt.figure(figsize=(15,6))

# Plot MSE for each fold
for fold_idx in range(num_folds):
    plt.plot(histories[fold_idx]['val_mse'][:-patience], color=colors[fold_idx],
        ↪ label=f'Fold N{fold_idx+1}')
    plt.ylim(0.009, 0.02)
    plt.title('Mean Squared Error')
    plt.legend(loc='upper right')
    plt.grid(alpha=.3)

# Show the plot
plt.show()

# Calculate the average best epoch
avg_epochs = int(np.mean(best_epochs))
print(f"Best average epoch: {avg_epochs}")

# Build the final model using the calculated average best epoch
final_model = build_model(input_shape, dropout_rate)

# Train the final model on the combined training and validation data
final_history = final_model.fit(
    x = X_train_val,
    y = y_train_val,
    batch_size = batch_size,
    epochs = avg_epochs
).history

# Evaluate and plot the performance of the final model on the test data
print('Final Model Test Performance')
plot_residuals(final_model, X_test.copy(), y_test.copy())
```