CHAPTER 10

Introduction to Artificial Neural Networks with Keras

Chapter 10 introduces **artificial neural networks (ANNs)** and shows how to implement multilayer perceptrons (MLPs) for regression and classification using **Keras** (tf.keras) in TensorFlow 2.

From neurons to MLPs

The chapter starts from **biological neurons** and builds to **artificial neurons** that compute a weighted sum of inputs plus bias, passed through an activation function. Single neurons can implement logic gates (AND, OR) and simple decision boundaries, but stacking layers yields networks that can approximate highly complex functions. The **Perceptron** algorithm (Rosenblatt) learns a linear decision boundary by updating weights on misclassified examples; it converges only if classes are linearly separable and is now mostly of historical interest.

An **MLP** is a feedforward network with one or more hidden layers; with non-linear activations (e.g., ReLU), an MLP is a universal approximator for continuous functions on compact sets. Training uses **backpropagation** (reverse-mode automatic differentiation) to compute gradients layer by layer and **gradient descent** (or variants) to update weights, minimizing a loss such as MSE (regression) or cross-entropy (classification).

Regression and classification MLPs

For **regression**, typical architectures use:

- One or more dense hidden layers with ReLU and a single linear output neuron for scalar regression, or multiple linear outputs for multivariate regression.
- Loss: usually **MSE** or MAE; metrics may include MAE or RMSE.

For **classification**:

- Binary: one output neuron with sigmoid activation and binary cross-entropy loss.
- Multilabel: one sigmoid neuron per label, binary cross-entropy.
- Multiclass single-label: one softmax neuron per class and sparse categorical cross-entropy (for integer labels) or categorical cross-entropy (for one-hot labels).

Building MLPs with Keras

Using **tf.keras**, you typically:

1. **Define the model**:
    - With the **Sequential API** for simple stacks (e.g., Flatten → Dense(300, ReLU) → Dense(100, ReLU) → Dense(10, softmax) for MNIST).
    - With the **Functional API** for more complex graphs, e.g., wide & deep networks that concatenate raw inputs and deep features, multiple inputs, or multiple outputs.
    - With the **Subclassing API** when you need fully dynamic architectures or custom behavior by subclassing keras.Model and implementing call.
2. **Compile** the model with a loss, optimizer, and metrics, e.g.
    - loss="sparse_categorical_crossentropy", optimizer=keras.optimizers.SGD(lr=...), metrics=["accuracy"] for MNIST classification.
3. **Train** using fit(...), passing training data, number of epochs, and either a validation set or validation_split; Keras returns a History object whose .history dict can be plotted to inspect learning curves.
4. **Evaluate and predict** using evaluate(...) to get test loss/metrics and predict(...) for class probabilities or regression outputs.

The MNIST example builds a Sequential classifier, compiles with sparse categorical cross-entropy and SGD, trains for 30 epochs with validation data, and reaches around 88–89% validation accuracy before evaluating on the test set.

Functional, multi-input, and multi-output models

The **Functional API** lets you define arbitrary DAGs of layers by treating layers as callable objects on tensors. Examples include:

- **Wide & Deep networks**: one input goes through several dense layers (deep path), then is concatenated with the original input (wide path), followed by an output layer; this combines memorization (wide features) with generalization (deep features).
- **Multiple inputs**: e.g., send features 0–4 through the wide path and features 2–7 through the deep path using two Input layers and concatenation, then define a single output.

- **Multiple outputs**: e.g., a main regression output plus an auxiliary output from a hidden layer for regularization; you compile with a list of losses and optional loss_weights to emphasize the main task (e.g., [0.9, 0.1]) and pass label tuples (y_main, y_aux) to fit.

These patterns are important for building production architectures such as recommendation models, multitask learning, and models with auxiliary heads to stabilize training.

Saving models, callbacks, and hyperparameters

Keras lets you **save and restore** models via model.save(...) and keras.models.load_model(...), including architecture, weights, and optimizer state. **Callbacks** like ModelCheckpoint, EarlyStopping, ReduceLROnPlateau, and TensorBoard can be passed to fit to automatically save best weights, stop training when validation loss stops improving, adapt learning rates, and log metrics for visualization.

Hyperparameter tuning focuses on:

- **Depth and width**: number of hidden layers and neurons per layer; too small underfits, too large overfits or becomes harder to train.
- **Learning rate** (most critical), optimizer choice, batch size, activation functions, and regularization (weight decay, dropout, etc., previewed here and detailed in Chapter 11).