Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

CSEN 903: Advanced Computer Lab, Winter 2025
Lab 3 Manual: Basics of Neural Networks with Tensorflow

Part 1: Read and Watch Before the Lab

In the previous lab, we explored **supervised learning**, where models learn from labeled data to make predictions on new, unseen inputs. We used a traditional machine learning model for training and prediction. In this lab, our focus shifts to **neural networks**, moving beyond classic machine learning approaches.

Neural networks are powerful machine learning models inspired by the structure of the brain. In this lab, we introduce **Feedforward Neural Networks** using **TensorFlow** and **Keras**. You will learn how to:d

1. Define model architecture (input, hidden, and output layers)
2. Choose activation functions (e.g., ReLU, sigmoid)
3. Set loss functions and optimizers
4. Train the model and monitor training progress
5. Evaluate performance on unseen data

We will continue using the **Titanic Dataset**, treating the survival prediction task as a binary classification.

Required Readings:

1. TensorFlow Keras Guide: Getting Started
2. TensorFlow Tutorial: Basic Classification
3. Introduction to Neural Networks
4. Activation Functions Explained

Required Watch List:

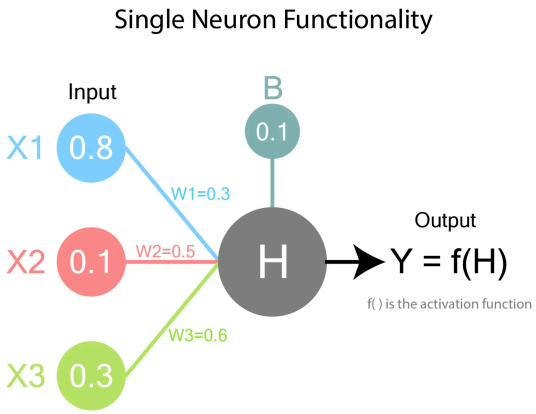
1. Neural Networks and Activation Functions
2. [Extra Material] Backpropagation

Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

Neural Network Model:

Neural network models are built from layers, and each layer is made up of neurons. Each neuron is influenced by the outputs of neurons from the previous layer, while neurons within the same layer are not directly connected. A neuron itself is a simple function. To see how this works, let's look at a real-life example using just one neuron.

In this simple example, the neuron is deciding whether a person will go to the cinema or not based on three main factors: **Weather (X1)**, **Company (X2)**, and **Proximity (X3)**. Each factor, however, does not contribute equally to the decision. Some may be more important than others. To reflect this, each input is multiplied by a weight (**W1, W2, W3**) that adjusts its influence. A bias term (**B**) is also added to give flexibility to the decision. All of these are combined into a single value **H**, calculated as:



$$H = X1*W1 + X2*W2 + X3*W2 + B$$

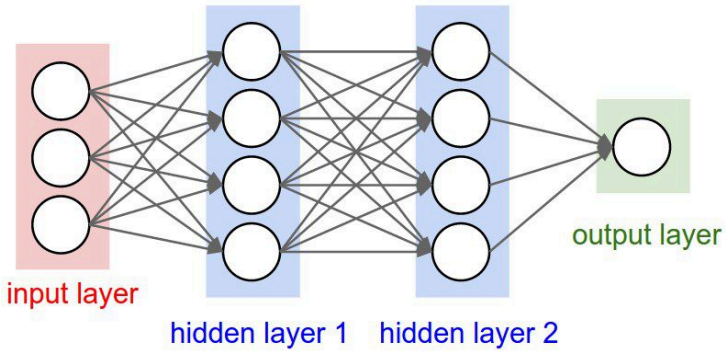
$$Y = f(H)$$

This value is then passed through a function **f(H)**, which outputs a clear decision: **Yes** (go to the cinema) or **No** (don't go). In this way, the neuron is essentially weighing the importance of weather, company, and proximity to decide the final outcome.

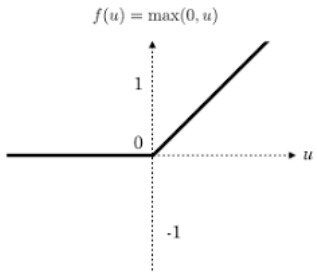
Function f(H) is referred to as an **activation function.** Activation functions introduce non-linearity, which is crucial because it allows the model to move beyond simple linear relationships and learn complex patterns in the data. More details about activation functions are to follow.

Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

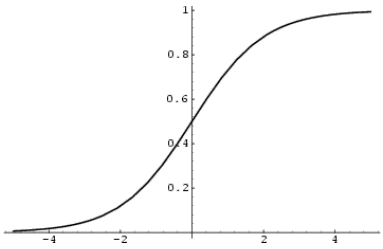
A single neuron makes simple decisions, but stacking neurons into **layers** allows the model to learn more complex patterns and make smarter decisions. The layers in a neural network are generally categorized into three types: the **input layer**, one or more **hidden layers**, and the **output layer**. When a network contains more than two hidden layers, it is commonly referred to as a **deep neural network**.



- **Input Layer:** the first layer of neurons in the model, which consists of *n* neurons, where *n* is equal to the number of features (i.e., the dimensionality) of the input data. In other words, the input layer has one neuron per feature in your dataset.
- **Hidden Layer(s):** middle layer(s) in a neural network where inputs are transformed into more useful features for learning.
- **Output Layer:** contains the neurons that produce the model's final predictions. Its size depends on the task: one neuron for binary classification, one per class for multi-class classification, or a single neuron with linear activation for regression.
- **Activation Function:** is like a switch that turns a neuron's input into an output. It lets neural networks learn complex patterns instead of just simple, straight-line relationships. Below, we will discuss the most commonly used activation functions:
    - **ReLU:** outputs the input if it's positive, otherwise sets it to 0, producing outputs in the range of [0, ∞[. It helps models learn faster, and works well when you just want to pass along positive values; thus, it is commonly used in hidden layers.



- **Sigmoid:** converts the input values into an S-shaped curve, producing outputs in the range [0, 1], which makes it mainly used for **binary classification problems**.



- **Softmax:** converts a vector of values into probabilities that sum to 1, with each output in the range [0, 1], which makes it suitable for **multi-class classification problems**, when you need to assign probabilities across several categories.

Part 2: In-Lab Task [Draft] In Lab Notebook

1. Set Up Your Environment.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras import Sequential, Input
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.metrics import Precision, Recall
```

2. Load and prepare the data. You can refer to the Lab Manual 1 for more information about Data Cleaning.

```
df = pd.read_csv('titanic.csv')
# Basic cleaning
df['Age'] = df['Age'].fillna(df['Age'].median())
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
df = pd.get_dummies(df, columns=['Embarked'], drop_first=True)
# Drop irrelevant columns
df = df.drop(columns=['Cabin', 'Name', 'Ticket'])
# Feature and target
X = df[['Pclass', 'Sex', 'Age', 'Fare', 'SibSp', 'Parch']]
y = df['Survived']
```

We usually **scale features** before training machine learning models because raw features can have very different ranges and units, which can cause problems for many algorithms.

Think about features in our Titanic dataset. One of the features is **Age (years, range ~0.1–80),** and another is **Fare (dollars, range ~20–500)**.

Without scaling, algorithms that depend on distances or weights may treat *fare* as much more important than *age*, just because the numbers are bigger. Standardization puts them on the same scale. Standardization means transforming each feature so that it has:

- **Mean = 0**
- **Standard deviation = 1**

```
# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Split the dataset into train and test sets:

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)
```

The **train_test_split** function basically splits the data into training and testing sets (both data and labels). Since the test_size=0.2, the data will be split with 80% used to train the model and 20% reserved to test its performance.

We want to shuffle the dataset before splitting, as it ensures both train and test sets represent the overall dataset fairly (non-biased dataset). Imagine a dataset of student grades sorted from lowest to highest. If we don't shuffle, the training set may contain only low grades, and the test set may contain only high grades. The model would not learn properly.

Adding the **random_state=42** parameter makes sure the data is shuffled in the same way every time, so the train-test split is consistent and reproducible.

Without a fixed random_state, every run might give a slightly different split, making results harder to compare. If you're testing different models, using the same split ensures a fair comparison since each model learns from the same training data and is tested on the same test data.

```
#Explore the size and dimension of the train and test data after splitting:
print("Train data shape:",X_train.shape, "Train Label shape", y_train.shape)
print("Test data shape:",X_test.shape, "Test Label shape", y_test.shape)
```

3. Build a Neural Network Model:

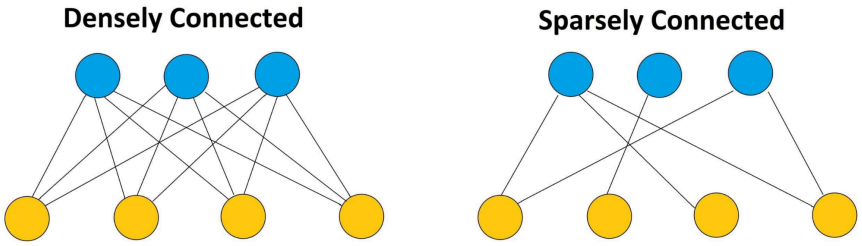
```
model = Sequential([
    Input(shape=(X_train.shape[1],)),
    layers.Dense(16, activation='relu'),
    layers.Dense(8, activation='relu'),
    layers.Dense(1, activation='sigmoid')  # binary classification
])
model.summary()   # displays the summary of your Keras model
```

In the above code snippet, **Input(shape=(X_train.shape[1],))** creates the model's input layer with one neuron for each feature in the training data.

The first hidden layer is a fully connected (dense) one with **16 neurons**. It means the layer learns 16 features from the previous layer's (i.e., input layer) outputs.

While the activation **ReLU** makes the 16 neurons output either the input value (if positive) or 0 (if negative).

A fully connected network means every neuron in one layer is connected to **all** neurons in the next layer. Each connection has its own weight, allowing the network to learn complex relationships between features.



The second hidden layer is also fully connected (dense) with **8 neurons**, meaning it learns 8 new features from the outputs of the first hidden layer. The **ReLU** activation makes these 8 neurons output either the input value (if positive) or 0 (if negative).

The output layer has one neuron with a sigmoid activation function that gives a value between 0 and 1, for example, 0.8, meaning an 80% chance of survival. Sigmoid is better than ReLU since it outputs probabilities and better than softmax since this is a binary rather than a multi-class problem.

After defining the model architecture, the next step is to compile it.

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy',
                       Precision(name='precision'),
                       Recall(name='recall')])
```

Compiling the model sets up how it will learn by choosing an **optimizer** (how weights update), a **loss function** (what the model tries to minimize), and evaluation **metrics** (Refer to Lab 2 for revisiting other evaluation metrics that can be used). How this works is as follows:

1. The model makes predictions.
2. The loss function measures the error (the smaller the loss value, the better our model is performing). It gives a single number representing "how bad"

the model is doing on training data by matching the model's predictions against the actual ground truths. We will discuss common loss functions and when to use each:

   a. **Binary Cross-Entropy:**
      i. Used for binary classification tasks (yes/no, spam/not spam, disease/no disease).
      ii. If the true label is `1` (positive) and the model predicts `0.9`, the loss will be small. If it predicts `0.1`, the loss will be large.
   b. **Categorical Cross-Entropy:**
      i. Used for multi-class classification (classifying images of cats, dogs, and birds).
      ii. It compares the predicted probability distribution with the true class (one-hot encoded).
   c. **Mean Squared Error (MSE), Mean Absolute Error (MAE):**
      i. Used for regression problems (predicting continuous values, like house prices, temperature).

3. The optimizer uses this error to update the weights in the right direction. The optimizer needs a target to minimize. The loss tells the optimizer which direction to adjust the weights to make predictions more accurate.

4. Train the Neural Network Model.

```
history = model.fit(X_train, y_train, epochs=20, validation_split=0.2, verbose=1)
```

- A **validation_split** divides your training data into two parts. Earlier, the dataset was split into training and testing sets. Here, we further split the training data into training and validation sets, where the training set is used to train the model, and the validation set helps monitor how well it is learning. The testing set remains separate for final evaluation.
- We use **validation** during training to make sure the model isn't just memorizing the training data but can also perform well on new, unseen data. We will split our data into 80% for training and 20% for validation.
- An **epoch** is one full pass through the entire training dataset, where the model sees and learns from every example once. While in theory, more epochs could improve learning, in practice, there are limitations. Training for too many epochs can be computationally expensive, especially with large datasets or data types like images that require significant processing power. Moreover, using an excessively large number of epochs can lead

to overfitting, where the model learns the training data too well and fails to generalize to new, unseen data.

- **verbose=1** means it will show a progress bar with details of the training process.

To avoid overfitting when training with many epochs, we use **early stopping**, which automatically stops training once the validation performance stops improving.

In the code below, we added **early stopping** to prevent overfitting when training the model. It monitors the **validation accuracy** and stops training if it doesn't improve for 3 consecutive epochs (`patience=3`). The parameter `restore_best_weights=True` ensures the model keeps the weights from the epoch with the highest validation accuracy. This way, we avoid wasting time on unnecessary epochs and keep the best-performing version of the model.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='val_accuracy', patience=3,
restore_best_weights=True)

history = model.fit(X_train, y_train,
                    epochs=50,
                    validation_split=0.2,
                    verbose=1,
                    callbacks=[early_stop])
```

Rather than feeding the entire dataset through the network in a single step, the training data is divided into smaller subsets called **batches**. The model processes one batch at a time and updates its weights after each batch. When all batches have been processed, the model has completed one full pass through the dataset, which is known as an **epoch**.

```
history = model.fit(X_train, y_train, epochs=20, batch_size=32,
validation_split=0.2, verbose=1)
```

For example, with `batch_size=32`, the model takes **32 samples**, runs them through the network, calculates the error, and updates weights. Then it takes the next 32 samples, and so on, until it covers the whole dataset (that's one epoch).

5. Visualize the training progress.

```
# Plot accuracy, precision, and recall from training history
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')

plt.plot(history.history['precision'], label='Train Precision')
plt.plot(history.history['val_precision'], label='Val Precision')

plt.plot(history.history['recall'], label='Train Recall')
plt.plot(history.history['val_recall'], label='Val Recall')

plt.xlabel('Epoch')
plt.ylabel('Score')
plt.title('Training Progress (Accuracy, Precision, Recall)')
plt.legend()
plt.show()
```

6. Evaluate on the test set.

```
test_loss, test_accuracy, test_precision, test_recall = model.evaluate(X_test,
y_test, verbose=0)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
```

This line evaluates the trained model on the test data and returns its loss and accuracy.

Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

Part 3: Graded Task

Task Description:

1. Build a **neural network classifier** using TensorFlow to predict survival.
2. Experiment with at least **two different architectures** (e.g., number of layers, activation functions).
3. Train and evaluate both models.
4. Plot training accuracy curves for both architectures and compare.
5. Report **final test accuracy** for both models.

Deliverables:

Submit an .ipynb notebook with all the completed tasks above to the form. The deadline for submission is **@11:59 PM the day before your lab during the week starting 11/10**. The evaluation will be held in the labs running during the aforementioned week.