Deep Learning with Keras and Tensorflow (M3)

Created	@December 11, 2025 8:27 PM
Module Code	IBMM03: Deep Learning with Keras and Tensorflow

Module 1: Introduction to Deep Learning

Introduction to Advanced Keras

Sequential API vs Functional API

Sequential API

- Good for simple models where layers form a **straight line**.
- Example: Input → Dense → Dense → Output.
- Limited flexibility — cannot handle complex architectures.
-

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a Sequential model
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Functional API

- Much more flexible and powerful.
- Lets you build:
    - Models with **multiple inputs**

    - Models with **multiple outputs**
    - **Shared layers**
    - **Branches**, **merges**, **residual connections**, etc.
- You manually define how layers are connected:
    - Define input layer
    - Pass it through layers
    - Define output
    - Build Model(inputs, outputs)

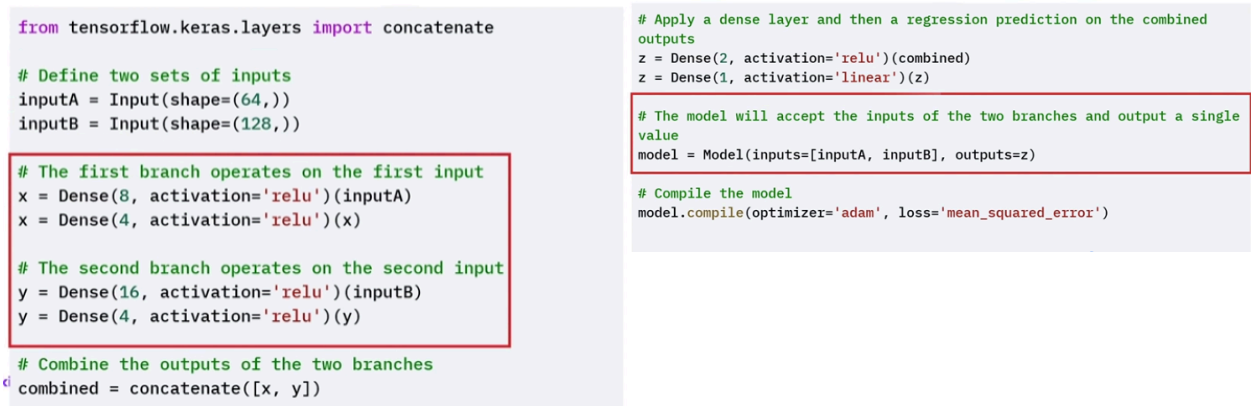
This gives better **clarity**, **control**, and **debugging ability**.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

# Define the input
inputs = Input(shape=(784,))

# Define layers
x = Dense(64, activation='relu')(inputs)
outputs = Dense(10, activation='softmax')(x)

# Create the model
model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy'
```

🔷 Example: Multi-Input Model

You can create two input layers, process each through separate paths, then **concatenate** them and pass through more layers to produce a single output.

This is not possible with the Sequential API.



```
from tensorflow.keras.layers import concatenate

# Define two sets of inputs
inputA = Input(shape=(64,))
inputB = Input(shape=(128,))

# The first branch operates on the first input
x = Dense(8, activation='relu')(inputA)
x = Dense(4, activation='relu')(x)

# The second branch operates on the second input
y = Dense(16, activation='relu')(inputB)
y = Dense(4, activation='relu')(y)

# Combine the outputs of the two branches
combined = concatenate([x, y])
```

```
# Apply a dense layer and then a regression prediction on the combined outputs
z = Dense(2, activation='relu')(combined)
z = Dense(1, activation='linear')(z)

# The model will accept the inputs of the two branches and output a single value
model = Model(inputs=[inputA, inputB], outputs=z)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

🔷 Where Functional API Is Used

It shines in real-world industries that need complex architectures:

Healthcare

- Medical image analysis
- Disease detection

Finance

- Market trend prediction

Autonomous Driving

- Object detection
- Lane detection
- Deep learning is a core area in data science, driving recent breakthroughs in various applications.
- Neural networks are the foundation of deep learning applications.

Keras Functional API and Subclassing API

What it does:

Functional API lets you build models as a **graph of layers**, not just a simple stack.

This means you can create:

- Multi-input models
- Multi-output models
- Shared layers
- Non-linear architectures (branches, merges)
- Complex models like Siamese networks
- Models for multitask learning

🌟 How it works:

- Start with an `Input()` layer.
- Pass it through layers like functions:
  `x = Dense(...)(input)`
- Combine inputs/branches using operations like:
  - `Concatenate()`
  - `Add()`
- Define final output.
- Build model using:
  `model = Model(inputs, outputs)`

```python
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
# Define the input
inputs = Input(shape=(784,))
# Define a dense layer
x = Dense(64, activation='relu')(inputs)
# Define the output layer
outputs = Dense(10, activation='softmax')(x)

# Create the model
model = Model(inputs=inputs, outputs=outputs)
```

🌟 Example capabilities:

1. **Multiple Inputs and Outputs**
   - Each input is processed separately.

- Their outputs are merged.
- Multiple outputs can be generated for multitask learning.

2. **Shared Layers**
   - Same layer is reused across multiple inputs.
   - Used in Siamese networks (compare two images/texts).
3. **Complex Architectures**
   - Multiple CNN branches
   - Merging using concat/add
   - Final dense layers for output

Functional API gives **clarity**, **structure**, and **high flexibility**, making it great for research and real-world ML systems.

🔷 3. Subclassing API (Most Advanced)

🌟 What it is:

Subclassing API gives **maximum control** by letting you create your own model class.

Implements subclassing of Model class and call() method

```
import tensorflow as tf
# Define your model by subclassing
class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        # Define layers
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(10,
activation='softmax')
   def call(self, inputs):
        # Forward pass
        x = self.dense1(inputs)
        return self.dense2(x)
# Instantiate the model
model = MyModel()
# Define loss function and optimizer
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
```

Example:

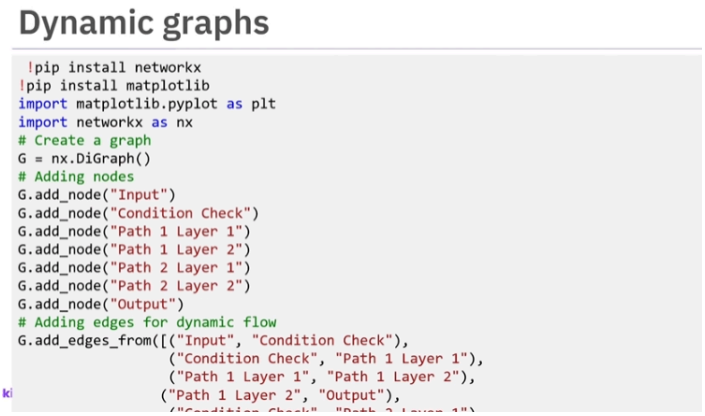
```
class MyModel(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.d1 = Dense(64, activation="relu")
        self.d2 = Dense(10, activation="softmax")

    def call(self, inputs):
        x = self.d1(inputs)
        return self.d2(x)
```

🌟 When to use Subclassing API:

- When model architecture must change **dynamically (Reinforcement)**When forward pass can't be described as a static graph.
- When implementing **custom training loops** using:
    - `tf.GradientTape`

🌟 Useful for:

- Reinforcement learning
- Dynamic graph models



- Research prototypes

- Custom layers & exotic architectures
- Variable-length loops or conditional execution

Subclassing provides **ultimate flexibility**, but:

- Harder to debug
- Harder to visualize
- No automatic shape inference

🔷 4. Custom Training Loops (tf.GradientTape)

Instead of using `model.fit()`, subclassing allows you to define:

```
with tf.GradientTape() as tape:
    predictions = model(x)
    loss = loss_fn(y, predictions)
```

This gives full control over:

- Loss computation
- Gradient updates
- Custom training behavior

Creating Custom Layers in Keras

Standard Keras layers (Dense, Conv, LSTM, etc.) handle most tasks, but sometimes you need more control.

✔️ When to Use Custom Layers:

- **Novel research ideas**

  Implement new neural network techniques not yet available in Keras.

- **Performance optimization**

Tailor operations for your dataset or computational constraints.

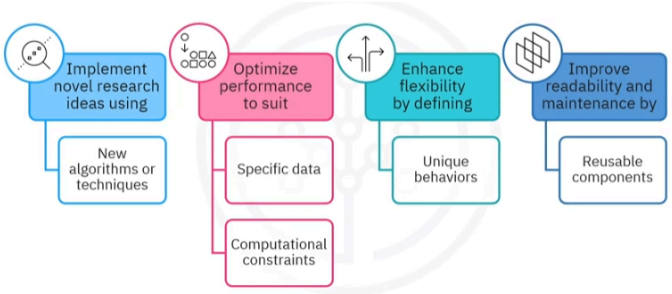
- **Special functionality**

  Create behavior not possible with standard layers.

- **Code readability**

  Wrap complex logic into a reusable layer.

Custom layers = **flexibility + innovation + cleaner code**.



🧩 Basic Structure of a Keras Custom Layer

To create a custom layer, you subclass from `tensorflow.keras.layers.Layer`

You typically implement **three key methods**:

1️⃣ **`__init__(self, ...)`**

- Define layer attributes (units, activation, etc.)
- No weights or shapes yet.

2️⃣ **`build(self, input_shape)`**

- Create and initialize trainable weights.
- Called automatically the first time layer receives input.
- Example: `self.kernel = self.add_weight(...)`

3️⃣ **`call(self, inputs)`**

- Defines the **forward pass** logic.
- All operations on the input happen here.

🛠️ Example: Custom Dense Layer With ReLU

Here is what the implementation (conceptually) looks like:

```
From tensorflow.keras.layers
import Layer
class CustomDense(tf.keras.layers.Layer):
    def __init__(self, units):
        super(CustomDense, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True
        )
        self.b = self.add_weight(
            shape=(self.units,),
            initializer="zeros",
            trainable=True
        )

    def call(self, inputs):
        x = tf.matmul(inputs, self.w) + self.b
        return tf.nn.relu(x)

a = tf.constant([1,2,3])
b = tf.constant([4,5,6])
print(tf.add(a,b))
```

Using the Custom Layer in a Model

You can integrate it like any other Keras layer:

```
model = tf.keras.Sequential([
    CustomDense(32),
    CustomDense(10)
])
```

No special treatment required — Keras handles everything.

Key Takeaway

Creating custom layers:

- Gives **complete control** over computations.
- Lets you **experiment with new neural architectures**.
- Helps in **research-heavy + specialized tasks**.
- Makes complex logic **clean and reusable**.

By practicing custom layers, you also gain a **deep understanding of how neural networks perform computations internally**.

Key Features of TensorFlow 2.x

1. **Eager Execution (default)**
    - Runs operations immediately
    - Easier debugging, simpler code, more intuitive
2. **High-Level API: Keras (tf.keras)**
    - Easy model building
    - Modular, user-friendly, great documentation
3. **Multi-platform Support**
    - Works on CPU, GPU, TPU, mobile, web, embedded systems
4. **Scalability & Performance**

- Efficient training on large datasets and hardware

5. **Rich Ecosystem**
    - **TensorFlow Lite** → mobile/embedded
    - **TensorFlow.js** → browser/JavaScript
    - **TFX** → production pipelines
    - **TensorFlow Hub** → reusable pretrained modules
    - **TensorBoard** → training visualization

Eager Execution

Executes code line-by-line like normal Python → immediate outputs, easier to debug, ideal for learning and experimentation.

Old TensorFlow versions required you to build a graph first, THEN run it. TF 2.x removes this complexity.

Old TF:

Write program → Build graph → Run graph → See output

TF 2.x:

Write code → Get output immediately

It behaves like normal Python → super easy for beginners.

Keras Integration

Simplifies building neural networks using clear, concise, high-level Python code.

Glossary :

Term	Definition
Build	A method that creates the layer's weights, called once during the first invocation of the layer.
Call	A method that defines the forward pass logic of the layer.

Term	Definition
Custom layer	A user-defined layer that allows customization of operations in a neural network, providing flexibility for specific tasks and experimentation.
Eager execution	A TensorFlow feature that executes operations immediately, making it more intuitive and useful for debugging and interactive programming.
Init	A method that initializes the layer's attributes.
Input layer	The first layer in a model that defines the input shape.
Keras	A high-level neural network API written in Python that can run on top of TensorFlow, Theano, and CNTK.
Keras Functional API	A powerful API for creating complex models with multiple inputs and outputs, shared layers, and non-sequential data flows.
Keras Sequential API	Creates models with layers in a linear stack.
ReLU	An activation function that outputs the input directly if positive; otherwise, it outputs zero. Commonly used in hidden layers.
Shared layer	Helpful when applying the same transformation to multiple inputs.
Softmax	An activation function suitable for classification tasks.