Introduction To Deep Learning & Neural Networks with keras (M2) (2)

Created	@December 10, 2025 7:12 PM
Module Code	IBMM02: Introduction to Deep Learning

Module 4 : Deep Learning Models

Shallow vs Deep Neural Networks

Type	Layers	What it handles	Use cases
Shallow Neural Network	1–2 hidden layers	Input must be preprocessed into numeric vectors	Simple problems, structured data
Deep Neural Network	3+ hidden layers (often many more)	Can take **raw data** (images, text, audio) and learn features automatically	Computer vision, NLP, speech, advanced ML tasks

Convolutional Neural Networks

A **CNN** is a special type of neural network designed specifically for **images**.

A normal neural network takes this kind of input:

```
[ x1, x2, x3, x4, ... ]
```

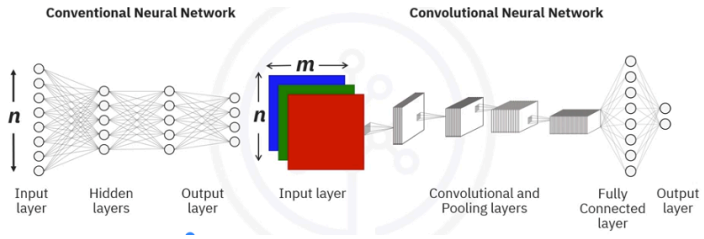
But images are **2D (or 3D)**:

Height × Width × Channels (RGB)

So CNNs **keep the spatial structure** of images instead of flattening everything at the start.

If we flatten an image (e.g., 128×128×3 → 49,152 numbers) and feed it into a fully connected network:

- Too many parameters → **slow training**, **high memory usage**
- Very easy to **overfit**



CNN Architecture

A CNN usually follows this order:

1️⃣ **Convolution Layer**

2️⃣ **ReLU Activation**

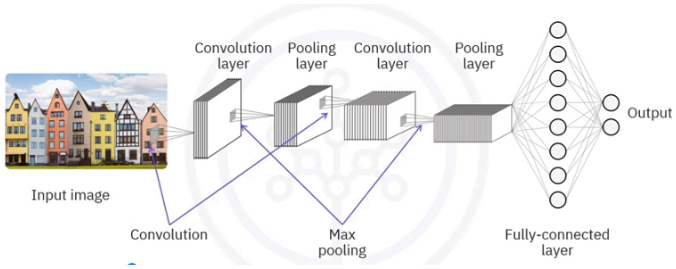
3️⃣ **Pooling Layer**

(repeat 1–3 several times)

4️⃣ **Flatten**

5️⃣ **Fully Connected (Dense) Layers**

6️⃣ **Output Layer (Softmax)**



1. Convolution Layer — Feature extractor

This layer applies **filters (kernels)** to the image.

A filter is a tiny matrix, like:

```
[1 0]
[-1 1]
```

The filter slides over the image → performs dot product → creates a **feature map**.

This helps the network learn:

- Edges
- Corners
- Colors
- Patterns
- Shapes

Using convolution dramatically **reduces number of parameters**.

Each filter = new feature map.

Using more filters → detect more types of features.

2. ReLU Activation

ReLU simply means:

```
if x < 0 → 0
else → x
```

This introduces **non-linearity** and makes training faster/less likely to have vanishing gradients.

---

3. Pooling Layer — Reduces Size

Pooling compresses image size:

Max Pooling (most common)

Keeps only the maximum value in each region.

```
[1, 3]
[2, 5]   → max = 5
```

Why?

- Simpler computation
- Reduces overfitting
- Makes model recognize objects even if position changes a bit → **spatial invariance**

Average Pooling

Takes the average of values in each region.

---

4. Fully Connected Layer

Once features are extracted, they are flattened into a long vector (1D) :

```
[ f1, f2, f3, ... ]
```

Then fed into:

- Dense layer(s)

- Output layer (softmax for classification)

This part works just like a normal neural network.

When are CNNs used?

✔️ Image classification

✔️ Object detection

✔️ Medical imaging

✔️ Face recognition

✔️ Traffic sign detection

✔️ OCR (text from images)

✔️ Satellite image analysis

CNNs handle **raw images** without manual feature engineering.

```
model = Sequential()
model.add(Conv2D(16, (2,2), activation='relu', input_shape=(128,128,3)))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(32, (2,2), activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

"""16 filters → then 32 filters
2×2 pooling
Dense layer with 100 units
Final softmax output"""
```

Recurrent Neural Networks

Traditional neural networks assume:

> Each input is independent.

But many real-world problems involve **sequences**, where the order matters, for example:

- Words in a sentence
- Scenes in a movie
- Stock prices over time
- DNA sequences
- Handwriting strokes

A normal neural network cannot understand sequence order.

RNNs solve this.

---

1. What Problem Do RNNs Solve?

RNNs are built to handle **sequential data**.

They add a **memory** mechanism, meaning:

✔️ They take the current input

✔️ AND they also use information from previous inputs

This lets the network understand **context** and **temporal patterns**.

---

2. How RNNs Work (Architecture)

Imagine a normal neural network, but with a loop:

At time **t = 0**

- Input: x_o
- Output: a_o

At time **t = 1**

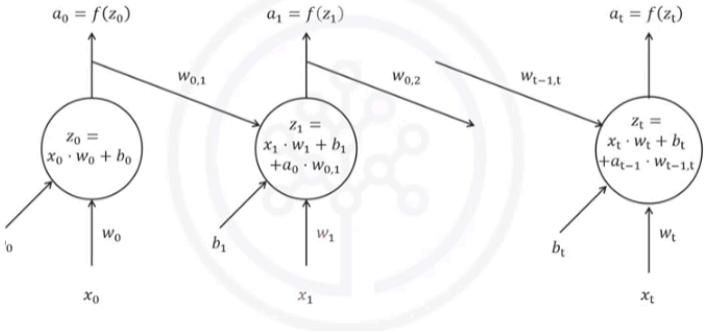
- Because it's sequential, the network takes:

- New input: x_1
- Previous output: a_o

So at each step:

```
output_t = f(input_t + previous_output_t-1)
```

This “loop” lets information flow from previous time steps.



3. What Are RNNs Used For?

RNNs are good at anything involving **time**, **order**, or **dependencies**.

Common uses:

- Text processing
- Language modeling
- Predicting next word
- Genomics
- Handwriting recognition
- Speech recognition
- Stock market prediction
- Audio generation

Basically anything where the meaning depends on what came before.

4. The LSTM (Long Short-Term Memory)

Basic RNNs have a problem:

They **forget long-term information** and suffer from **vanishing gradients**.

LSTMs fix this.

An LSTM has special **gates** that control:

- What to remember
- What to forget
- What to output

So LSTMs can capture **long-range dependencies** (like remembering the subject of a sentence many words later).

---

5. Real-World Applications of LSTMs

LSTMs are widely used for:

✔️ Image generation

Using sequential pixel patterns

✔️ Handwriting generation

Reproducing realistic handwritten strokes

✔️ Automatic image captioning

CNN extracts features → LSTM generates sentence

✔️ Video description

LSTM learns sequences of visual features over time

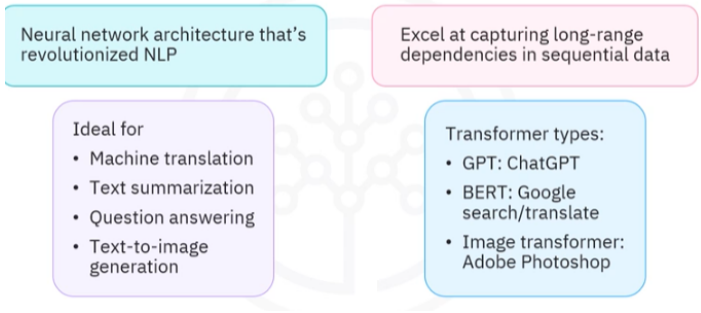
They are extremely powerful in tasks involving long sequences.

Transformers

Transformers are a type of deep learning architecture that changed the entire field of **NLP, vision, and generative AI**. They are used in today's powerful AI systems—like ChatGPT, Gemini, BERT, and image generators such as DALL·E.

The key reason they became dominant:

> Transformers can capture long-range relationships in data extremely well and can be trained in parallel, making them fast and scalable.



---

1. Why Transformers?

Before transformers:

- **RNNs** could handle sequences but were slow (trained step by step) and struggled with long dependencies.
- **CNNs** could understand patterns but were not built for text or long sequences.

Transformers solve both problems:

They understand **context across the entire sequence**, not just nearby words

They process all tokens **in parallel**, making training **much faster**

---

2. The Core Idea: Attention

Transformers rely on a mechanism called **attention**.

Instead of reading tokens one by one, the model:

- Looks at the whole sequence at once
- Decides which words are important for understanding each token

Example:

In the sentence *"The dog ran because **it** was scared,"*

the word **"it"** should be linked to **"dog"**.

Attention makes this possible.

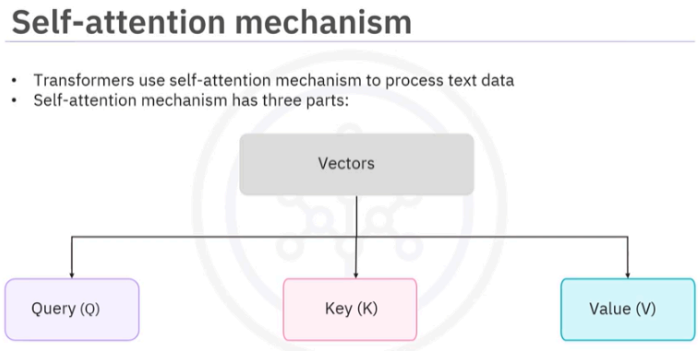
---

3. Self-Attention: How It Works

Self-attention is used when processing text.

For each token, the model generates:

- **Query (Q)** → the token we are focusing on
- **Key (K)** → signals how relevant each token is
- **Value (V)** → the information carried by each token

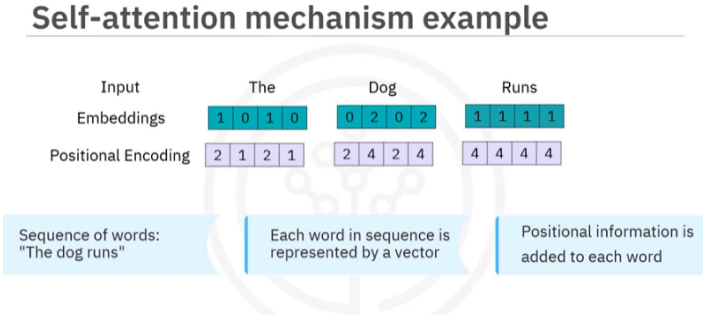


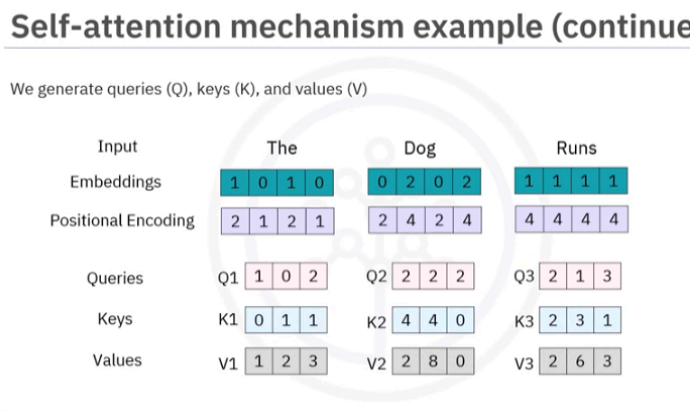
Steps:

1. **Q, K, V vectors are created** for every token
2. The model computes **attention scores** → Q · K
3. Scores are scaled and passed through **Softmax** to get probabilities
4. The model computes a **weighted sum of V** values

   → this produces a *contextualized meaning* of the token

This process helps each word understand the entire sentence.





4. Cross-Attention: For Text-to-Image Generation

Cross-attention is used when **one type of data guides another**, such as:

- Text → influences image generation

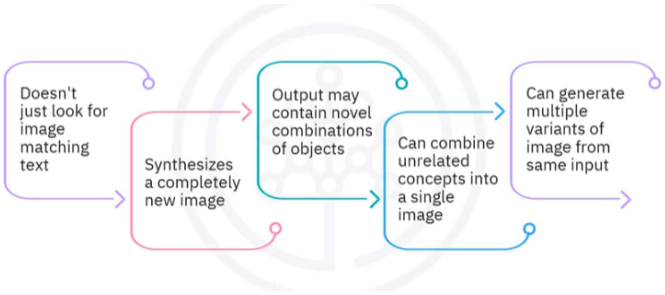
Example:

Prompt: *"a two-story house with a red roof"*

Process:

1. The text is converted into embeddings
2. The transformer's encoder produces a sequence of Q (from text)
3. The image-generation model (e.g., DALL·E) uses cross-attention so image tokens respond to text tokens
4. The model autoregressively generates image patches based on:
    - Text meaning
    - Previously generated image parts

This allows the model to **synthesize new images**, not just retrieve existing ones.



5. Power of Transformers

Transformers are used in:

📝 NLP

- Translation
- Summarization
- Answering questions
- Chatbots

🎨 Creativity

- Text-to-image generation
- Style transfer
- Novel concept generation

🔍 Other fields

- Search (Google uses BERT)
- Photoshop’s image editing features
- Code generation (Copilot)

They handle extremely long and complex patterns better than older models.

6. Limitations of Transformers

Despite their power, transformers have some drawbacks:

❗ They require huge datasets

Without massive training data, they don’t generalize well.

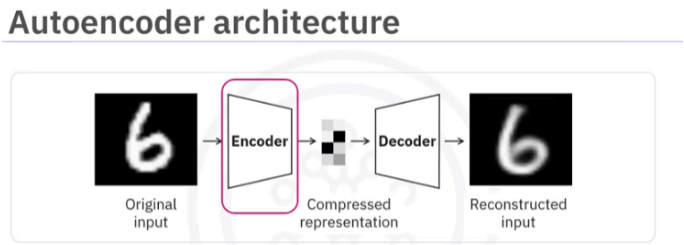
❗ They inherit bias from training data

If the data contains stereotypes or errors, the model can learn them.

❗ They need large compute resources

Transformers are computationally expensive.

Autoencoder



What are Autoencoders?

Autoencoders are **unsupervised neural networks** used mainly for:

- **Compressing data** (encoding)
- **Reconstructing data** (decoding)

They learn to recreate the input **as output**, so the target = input.

Think of them as a machine that learns:

> "How can I represent this data in a smaller form... and bring it back?"

How They Work (Architecture)

1. Encoder

- Takes the input (image/text/etc.)
- Compresses it into a smaller representation (called *latent vector*)

2. Bottleneck

- The compressed "code"

- Forces the network to keep only the most important features

3. Decoder

- Expands the compressed representation back to original shape

Training Objective

Minimize:

```
Original Input  ≈  Reconstructed Output
```

So, the network learns meaningful patterns.

Why Autoencoders are Useful

1. Data Denoising

Feed a noisy image → autoencoder learns to remove noise.

2. Dimensionality Reduction

Similar to PCA, but **non-linear**, so more powerful.

3. Feature Learning

Automatically extracts patterns in data.

Restricted Boltzmann Machines (RBMs)

RBMs are a special type of autoencoder-like model (older, but powerful):

Applications:

✔️ Fixing **imbalanced datasets**

- RBMs learn the distribution of the minority class
- Then they **generate synthetic samples**
- Helps balance class counts

✔️ **Estimating missing values**

- RBMs learn correlations in features
- Can fill missing entries

✔️ **Automatic feature extraction**

- Especially useful for unstructured data (e.g., images)

Using Pretrained Model

What is a Pretrained Model?

A **pretrained model** is a neural network that has already been trained on a large dataset (like **ImageNet**, with 14M images / 1000 classes).

Examples:

- **VGG16**
- **ResNet**
- **MobileNet**
- **EfficientNet**

These models have already learned:

- Edges
- Textures
- Shapes
- Object parts

...so you don't need to train them from scratch.

Two Ways to Use Pretrained Models

1. Use as a Fixed Feature Extractor

- You **freeze** all layers → no retraining.
- Pass new images through the network.

- The model outputs **feature maps** (high-level patterns).
- You use these features for:
  - Clustering
  - Visualization
  - Dimensionality reduction
  - Feeding into ML models (SVM, RF, etc.)

Benefits

✔ No training needed

✔ Very fast

✔ Works well with small datasets

✔ Uses very powerful learned features

Limitations

✘ Cannot adapt to your dataset

✘ Performance limited if your data is very different from ImageNet

```
import os
import shutil
from PIL import Image
import numpy as np

# Define the base directory for sample data
base_dir = 'sample_data'
class1_dir = os.path.join(base_dir, 'class1')
class2_dir = os.path.join(base_dir, 'class2')

# Create directories for two classes
os.makedirs(class1_dir, exist_ok=True)
os.makedirs(class2_dir, exist_ok=True)

# Function to generate and save random images
def generate_random_images(save_dir, num_images):
    for i in range(num_images):
        # Generate a random RGB image of size 224x224
```

How to use pretrained models as feature extractors in Keras

```
        # Generate a random RGB image of size 224x224
        img = Image.fromarray(np.uint8(np.random.rand(224, 224, 3) * 255))
        # Save the image to the specified directory
        img.save(os.path.join(save_dir, f'image_{i}.jpg'))

# Number of images to generate for each class
num_images_per_class = 100  # You can increase this to have more training
data

# Generate random images for class 1 and class 2
generate_random_images(class1_dir, num_images_per_class)
generate_random_images(class2_dir, num_images_per_class)

print(f'Sample data generated at {base_dir} with {num_images_per_class}
images per class.')
```

Example: Pretrained model for extracting features

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam

# Load the VGG16 model pre-trained on ImageNet
base_model = VGG16(weights='imagenet'
, include_top=False, input_shape=(224, 224, 3))

# Freeze all layers initially
for layer in base_model.layers:
    layer.trainable = False

# Create a new model and add the base model and new layers
model = Sequential([
    base_model,
    Flatten(),
```

Example: Pretrained model for extracting features

```
# Create a new model and add the base model and new layers
model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dense(1, activation='sigmoid')

# Change to the number of classes you have
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
loss='binary_crossentropy', metrics=['accuracy'])
```

Example: Pretrained model for extracting features

```
# Load and preprocess the dataset
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    '/content/sample_data',
    target_size=(224, 224),
    batch_size=32,
    class_mode='binary'
)

# Train the model with frozen layers
model.fit(train_generator, epochs=10)

# Gradually unfreeze layers and fine-tune
for layer in base_model.layers[-4:]:  # Unfreeze the last 4 layers
    layer.trainable = True

# Compile the model again with a lower learning rate for fine-tuning
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='binary_crossentropy', metrics=['accuracy'])
```

```
# Fine-tune the model
model.fit(train_generator, epochs=10)
```

2.Fine-Tuning (Transfer Learning)

- You **unfreeze some top layers** of the pretrained model.
- Add new layers for your task.
- Train everything together (usually with low learning rate).

Why Fine-Tune?

Because the pretrained model may not perfectly match your new dataset.

For example:

- ImageNet → dogs, cars, buildings
- Your dataset → retinal fundus images

So fine-tuning helps the model adjust.

Benefits

✔️ Much better accuracy

✔️ Model adapts to your domain

✔️ Still requires *far less* data than training from scratch

Keras Workflow (Conceptual Steps)

Step 1: Load pretrained model

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

Step 2: Freeze its layers

```
base_model.trainable = False
```

Step 3: Add your own classifier

Flatten → Dense(256, relu) → Dense(1, sigmoid)

Step 4: Train your model (feature extraction mode)

This only trains the new layers you added.

Step 5: (Optional) Fine-tune

Unfreeze top layers from VGG16 and train again with smaller LR.

When Should You Use Feature Extraction vs Fine-Tuning?

Scenario	Best Choice
Very small dataset	Feature extractor
Low compute	Feature extractor
Dataset very different from ImageNet	Fine-tuning
Want highest accuracy	Fine-tuning
Quick prototype	Feature extractor