# Deep Learning with Keras and Tensorflow (M3)

| | |
|---|---|
| Created | @December 12, 2025 10:55 AM |
| Module Code | IBMM03: Deep Learning with Keras and Tensorflow |

# Module 2: Advanced CNNs in Keras

## Advanced CNN Techniques Using Keras

### What are CNNs?

Convolutional Neural Networks (CNNs) process visual data (Image).

What is an Image?

- A grid of numbers (Pixels)
- Each Pixel has 3 values:  R, G, B
- Tensorflow represents this as a tensor:

Height × Width × Channels
e.g., 224 × 224 × 3

### ✍🏻 What are tensor then? - Data Values and Shape

- mathematical object that generalizes scalars, vectors and matrices into a single concept.
- Tensors act a Multidimensional arrays / Data structure in ML and DL as Tensorflow stores data as tensors.
- Eg. Image: 3D Tensor (h, w, channels)
- A batch of Images: 4D Tensors
- A video: 5D Tensors

They use:

- **Convolution layers** → extract features
- **Pooling layers** → reduce size
- **Fully connected layers** → classification

A basic CNN stacks multiple Conv → Pool → Dense layers.

---

## Basic CNN

Typical structure:

1. Conv2D (32 filters, 3×3, ReLU)
2. MaxPooling (2×2)
3. Conv2D (64 filters)
4. MaxPooling
5. Flatten
6. Dense(128, ReLU)
7. Output Dense(10, Softmax)

Compiled with Adam + categorical crossentropy.

```
# Create a Sequential model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D((2,  2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Raw Image (jpg/png)
↓
Decode (convert to tensor)
↓
Resize to model's required size
↓
Normalize (0-1 or -1 to 1)

↓
Apply augmentation (if training)
↓
Expand dims to batch shape
↓
Feed to CNN

---

## Advanced CNN Architectures

## VGG (Deep + Simple)

- Uses many stacked **3×3 convolution layers**
- Depth increases: 64 → 128 → 256 → 512
- Followed by MaxPooling after each block
- Ends with fully connected layers

**Idea:** small filters + many layers → strong feature extraction.

```
model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(256, (3, 3), activation='relu'),
    Conv2D(256, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
```

```
    Dense(512, activation='relu'),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
# Summary of the model
model.summary()
```

---

## ResNet (Residual Connections)

- Solves **vanishing gradient problem**
- Uses **skip connections**:
- Allow the network to learn identity mapping

output = F(x) + x

- Helps train very deep networks easily
- Residual block = Conv → Conv → Add shortcut → ReLU

**Key benefit:** stable training for deep models.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization,
Activation, Add, Flatten, Dense

def residual_block(x, filters, kernel_size=3, stride=1):
    shortcut = x
    x = Conv2D(filters, kernel_size, strides=stride, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = Conv2D(filters, kernel_size, strides=1, padding='same')( x)
    x = BatchNormalization()(x)
    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x
```

```
input = Input(shape=(64, 64, 3))
x = Conv2D(64, (7, 7), strides=2, padding='same')(input)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = residual_block(x, 64)
x = residual_block(x, 64)
x = Flatten()(x)
outputs = Dense(10, activation='softmax')(x)
```

---

# Data Augmentation Techniques

## What is Data Augmentation?

Data augmentation creates modified versions of training images (rotated, flipped, shifted, etc.) to make the model more robust and reduce overfitting.

## Why is it important?

- Increases data variety
- Prevents overfitting
- Improves generalization on unseen data

## Basic Augmentation Techniques

Using **ImageDataGenerator**, you can apply:

- **Rotation**
- **Width/height shifts**
- **Zoom**
- **Shear**
- **Horizontal flip**
- **Rescale**

These create new image variations during training.

```
from tensorflow.keras.preprocessing.image
import ImageDataGenerator

# Create an instance of ImageDataGenerator with augmentation options
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Load a sample image and reshape it
from tensorflow.keras.preprocessing
import image
img = image.load_img('sample.jpg')
x = image.img_to_array(img)
x = x.reshape((1,) + x.shape)
```

---

## Advanced Augmentation

### Feature-wise normalization

- Dataset mean = 0
- Dataset std = 1

### Sample-wise normalization

- Each image mean = 0
- Each image std = 1

```
# Create an instance of ImageDataGenerator with advanced options
datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    samplewise_center=True,
    samplewise_std_normalization=True
)
# Compute the mean and standard deviation on a dataset of images
datagen.fit(training_images)
# Generate batches of normalized images
i = 0
for batch in datagen.flow(training_images, batch_size=32):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()
```

### Custom Augmentation

You can write your own preprocessing function (e.g., adding random noise) and pass it to:

`ImageDataGenerator(preprocessing_function=your_function)`

```python
import numpy as np
def add_random_noise(image):
    noise = np.random.normal(0, 0.1, image.shape)
    return image + noise
# Create an instance of ImageDataGenerator with custom augmentation
datagen = ImageDataGenerator(preprocessing_function=add_random_noise)
# Generate batches of augmented images
i = 0
for batch in datagen.flow(training_images, batch_size=32):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()
```

### Why do CNNs overfit?

Because they memorize patterns from LIMITED data.

So we trick the model by making **fake variations** of the same image.

TensorFlow can rotate, zoom, flip, crop, etc.

### Intuition:

Model sees:

- The same cat rotated
- The same cat zoomed
- The same cat with brightness changed

It learns:

👉 *"This is still a cat. I shouldn't overfit to exact pixels."*

Mathematically, all these transformations are just **function operations on pixel coordinates**, but you don't need formula-level details right now.

## Transfer Learning in Keras

### What is Transfer Learning?

Using a model **pre-trained on a large dataset** (e.g., ImageNet) and reusing its learned features for a **new but related task**.

---

## Why use Transfer Learning?

- **Reduced training time** – model already knows useful features.
- **Better performance** – pre-trained on huge datasets.
- **Works well with small datasets** – especially useful in medical imaging, NLP, etc.
- **Less computation needed.**

---

## How it works

1. Load a **pre-trained model** (e.g., VGG16).
2. Remove its top layers (`include_top=False`).
3. Freeze its convolutional layers (`layer.trainable = False`).
4. Add new custom layers for your task (Flatten + Dense layers).
5. Compile and train.

---

## ⚙️ Code Breakdown

## 1. Load Pre-trained Model

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

## 2. Freeze layers

```
for layer in base_model.layers:
    layer.trainable = False
```

### 3. Add new classifier

```
model = Sequential()
model.add(base_model)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

### 4. Compile

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

### 5. Data preprocessing

```
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
    directory,
    target_size=(224,224),
    batch_size=32,
    class_mode='binary'
)
```

### 6. Train

```
model.fit(train_generator, epochs=10)
```

---

## 🔧 Fine-tuning

Unfreeze top layers to adapt the model more deeply:

```
for layer in base_model.layers[-4:]:
    layer.trainable = True
```

Then recompile + train again.

# Using Pre-Trained Models

## 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

```python
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

```python
        # 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=(2
24, 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.

## TensorFlow for Image Processing

Image processing means **transforming or analyzing images** to extract useful information.

Used in:

- Medical imaging
- Autonomous vehicles
- Facial recognition
- Computer vision tasks

---

### Why Use TensorFlow for Image Processing?

- **Easy to use** — high-level APIs simplify complex operations
- **Pretrained models available** (ResNet, Inception, etc.)
- **Scalable** from mobile devices to cloud clusters
- **Strong community & documentation**

---

### Basic Image Processing in TensorFlow

Common steps:

1. **Load image**
2. **Resize** (e.g., 224×224)
3. **Convert to NumPy or tensor**
4. **Add batch dimension** → required for model predictions

   `img = tf.expand_dims(img, axis=0)`

```python
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
img_to_array, load_img

# Load and preprocess the image
img = load_img('/content/path_to_image.jpg', target_size=(224, 224))
img_array = img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Add batch dimension

# Display the original image
import matplotlib.pyplot as plt
plt.imshow(img)
plt.show()
```

---

### Data Augmentation with TensorFlow

Used to increase dataset diversity → reduces overfitting.

Typical augmentations:

- Rotation
- Shifts (width/height)
- Shear
- Zoom
- Flip

TensorFlow can generate **batches of augmented images automatically**.

# Transpose Convolution

# What is Transpose Convolution?

Transpose convolution (also called **deconvolution**) is a technique used to **increase the spatial resolution** of feature maps.

It performs the **inverse of standard convolution** → used for **up-sampling**.

---

### Why Standard Convolution Is Not Enough?

- Normal convolution **reduces** spatial size.

- Tasks like **image generation**, **super-resolution**, and **semantic segmentation** need **larger** output images.
- Transpose convolution expands the image.

---

## How Transpose Convolution Works?

- Inserts **zeros between pixels** of the input feature map.
- Applies a convolution filter over this expanded grid.
- Produces a **larger** (up-sampled) feature map.

---

## Applications

- **GANs:** generate images from a latent vector
- **Super-resolution:** enhance image resolution
- **Semantic segmentation:** create pixel-wise output maps

---

## 🧩 Keras Implementation

- `Conv2DTranspose(filters=32, kernel_size=3, strides=2, activation='relu')`

  → upsamples by factor of 2
- Output layer often uses:
  - `sigmoid` (binary masks)
  - `softmax` (multi-class segmentation)
- Compile with:
  - Optimizer: `Adam`
  - Loss: e.g., `mse`, or segmentation-specific loss

```python
import os
import logging
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2DTranspose

# Set environment variables to suppress TensorFlow warnings
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'  # Ignore INFO, WARNING, and ERROR
messages
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'  # Turn off oneDNN custom
operations

# Use logging to suppress TensorFlow warnings
logging.getLogger('tensorflow').setLevel(logging.ERROR)

# Define the input layer
input_layer = Input(shape=(28, 28, 1))
```

```python
# Add a transpose convolution layer
transpose_conv_layer = Conv2DTranspose(filters=32, kernel_size=(3, 3),
strides=(2, 2), padding='same', activation='relu')(input_layer)

# Define the output layer
output_layer = Conv2DTranspose(filters=1, kernel_size=(3, 3),
activation='sigmoid', padding='same')(transpose_conv_layer)

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

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

# Display the model summary
model.summary()
```

---

# Issue: Checkerboard Artifacts

Caused by uneven kernel overlap during transpose convolution.

## How to Fix

Use:

- **UpSampling2D** → simple nearest-neighbor/bilinear upsampling
- How to fill pixel values in new positions:

## a) Nearest Neighbor (most intuitive)

“Pick the closest pixel.”

Works like zooming into a Minecraft block → blocky but fast.

Math (simple):

`new_pixel = old_pixel at nearest coordinate`

---

## b) Bilinear (smoother)

“Blend the 4 nearest pixels.”

Imagine mixing colors from nearby areas.

Math (concept version):

TensorFlow takes:

- top-left pixel
- top-right pixel
- bottom-left pixel
- bottom-right pixel

Then blends them depending on distance

- Followed by **Conv2D** → smooths & refines output

```
# Define a model with up-sampling followed by convolution to avoid
checkerboard artifacts
x = UpSampling2D(size=(2, 2))(input_layer)
output_layer = Conv2D(filters=64, kernel_size=(3, 3), padding='same')(x)
```

This produces cleaner, artifact-free images.

Glossary :

| Term | Definition |
|---|---|
| **Activation function** | A mathematical function used in neural networks to determine the output of a neuron. |
| **Adam optimizer** | An optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data. |
| **Augmentation** | A process of increasing the diversity of training data by applying various transformations like rotation, scaling, and so on. |
| **Binary cross-entropy** | A loss function used for binary classification tasks, measuring the performance of a classification model whose output is a probability value between 0 and 1. |
| **Convolution** | A mathematical operation used in deep learning, especially in convolutional neural networks (CNNs), for filtering data. |

| Term | Definition |
| --- | --- |
| **Custom augmentation function** | A user-defined function that applies specific transformations to images during data augmentation, providing full control over the augmentation process. |
| **Data augmentation** | Techniques used to increase the diversity of training data by applying random transformations such as rotation, translation, flipping, scaling, and adding noise. |
| **Deconvolution** | Also known as transpose convolution, this is a technique used to up-sample an image, often used in generative models. |
| **Dense layer** | A fully connected neural network layer, where each input node is connected to each output node, commonly used in the final stages of a network. |
| **Feature map** | A set of features generated by applying a convolution operation to an image or data input. |
| **Feature-wise normalization** | A technique to set the mean of the data set to 0 and normalize it to have a standard deviation of 1. |
| **Fine-tuning** | The process of unfreezing some of the top layers of a pre-trained model base and jointly training both the newly added layers and the base layers for a specific task. |
| **Flatten layer** | A layer that converts the output of a convolutional layer to a 1D array, allowing it to be passed to a fully connected layer. |
| **Generative adversarial networks (GANs)** | A class of machine learning frameworks where two neural networks compete with each other to create realistic data samples. |
| **Height shift range** | A data augmentation parameter that randomly shifts an image vertically, altering its position to improve model robustness to vertical translations. |
| **TensorFlow Hub** | A repository of reusable machine learning modules, which can be easily integrated into TensorFlow applications to accelerate development. |
| **TensorFlow.js** | A library for training and deploying machine learning models in JavaScript environments, such as web browsers and Node.js. |

| Term | Definition |
|---|---|
| **Horizontal flip** | A data augmentation technique where the image is flipped horizontally, creating a mirror image to increase data diversity. |
| **ImageDataGenerator** | A Keras class used for generating batches of tensor image data with real-time data augmentation. |
| **ImageNet** | A large visual database designed for use in visual object recognition software research, often used as a data set for pre-training convolutional neural networks. |
| **Image processing** | The manipulation of an image to improve its quality or extract information from it. |
| **Kernel** | A small matrix used in convolution operations to detect features such as edges in images. |
| **Latent vector** | A vector representing compressed data in a lower-dimensional space, often used in generative models. |
| **Pre-trained model** | A model previously trained on a large data set, which can be used as a starting point for training on a new, related task. |
| **Random noise** | A type of custom augmentation that adds random noise to images, simulating different lighting conditions and sensor noise to make models more robust. |
| **Rotation range** | A data augmentation parameter that randomly rotates an image within a specified range of degrees, enhancing model robustness to rotations. |
| **Sample-wise normalization** | A technique to set the mean of each sample to 0 and normalize each sample to have a standard deviation of 1. |
| **Semantic segmentation** | A deep learning task that involves classifying each pixel in an image into a predefined class. |
| **Shear range** | A data augmentation parameter that applies a shear transformation to an image, slanting it along one axis to simulate different perspectives. |
| **Stride** | A parameter in convolution that determines the step size of the kernel when moving across the input data. |
| **TensorFlow** | An open-source machine learning library used for various tasks, including deep learning and image processing. |

| Term | Definition |
|---|---|
| **Transfer learning** | A method where a pre-trained model is adapted to a new, related task by adjusting its weights, allowing it to perform well even with limited data for the new task. |
| **Transpose convolution** | An operation that reverses the effects of convolution, often used for up-sampling in image processing. |
| **VGG16** | A convolutional neural network model pre-trained on the ImageNet data set, commonly used in transfer learning for tasks involving image classification. |
| **Width shift range** | A data augmentation parameter that randomly shifts an image horizontally, altering its position to improve model robustness to horizontal translations. |
| **Zoom range** | A data augmentation parameter that randomly zooms in or out on an image, altering its scale during training. |