A Review of LeNet-5 and AlexNet Models

Shieladitya Basu

October 7, 2025

Part I
A Review of LeNet-5

1 Introduction

The paper "Gradient-Based Learning Applied to Document Recognition" was written in a period of significant transition within the field of pattern recognition during the late 1990s. The historical context is characterized by a major shift from traditional methods relying on hand-crafted features towards systems that depend more heavily on automatic learning from data.

1.1 Historical Context

Historically, pattern recognition systems were built with two distinct modules: a fixed hand-designed extractor and a separate trainable classifier. The success of these systems was heavily dependent on the designer's ability to manually create a good set of features, a task described as "daunting" and specific to each new problem. This approach was necessary because early learning techniques were limited and could handle only low-dimensional data.

The paper argues that this vision had been changed by three key factors over the preceding decade (roughly the late 1990s to late 1990s).

1. Increased Computational Power
2. Availability of Large Datasets
3. Powerful Machine Learning Techniques

The central message of the paper is that relying more on automatic learning and less on hand-designed heuristics leads to better pattern recognition systems.

1.2 The Rise of Gradient-Based Learning and Back-Propagation

The paper was written at the time when gradient-based learning to complex and nonlinear systems was very popular. Although the core ideas of gradient descent had been known since the 1950s, their widespread application was limited to linear systems. The surprising usefulness of these methods for complex tasks became widely recognized because of three key events that occurred in the mid-1980s:

1. The realization that local minima were not a major practical problem for training neural networks, contrary to earlier fears.

2. The popularization of the back-propagation algorithm by researchers like Rumelhart, Hinton, and Williams, provided an efficient way to compute gradients in multi-layer systems.

3. Successful demonstrations that back-propagation could be used to train multi-layer neural networks could be used to train multi-layer networks to solve complicated learning tasks.

By the time this paper was written, multi-layered neural networks trained with back-propagation were already being used in many commercial OCR systems.

1.3 Challenges in Handwritting Recognition

The paper uses handwritten character recognition as a case study to demonstrate its core ideas. While recognizing isolated characters was an early success for neural networks, a much harder problem was segmentation which is the process of separating individual characters from their neighbors within a word or sentence.

The standard technique at the time was Heuristic Over-Segmentation, which involved generating numerous potential character cuts and using a recognizer to score the resulting segments. A major challenge with this approach was the difficulty of training the recognizer to distinguish correctly segmented characters from incorrectly segmented ones, which required the tedious and costly creation of labeled databases of character fragments and errors.

The paper presents novel solutions to this long-standing problem through two main approaches:

- Global Training: The system is trained at the level of whole character strings, optimizing a single performance criterion for the entire system. This avoids the need for manually labeled segmented characters. The paper introduces *Graph Transformer Networks* (GTN) as a unified paradigm for designing and training such multi-module systems.
- Eliminating Segmentation: The Space Displacement Neural Network (SDNN) approach avoids segmentation heuristics altogether by sweeping a recognizer across all possible locations on the input image. This was made feasible by the efficiency and robustness of *Convolutional Neural Networks.*

The paper's own work, including the development of LeNet-5 and its commercial deployment in a check reading system that reads millions of checks per day, serves as a landmark application of these advanced, learning-centric techniques.

2 CNNs For Isolated Character Recognition

Multilayer networks trained with gradient descent to learn complex mappings from large collections of data makes them suitable for image recognition tasks. In the traditional model of pattern recognition a hand designed feature extractor gathers relevant information from the input and then a trainable classier (standard fully-connected multilayer networks can be used) then categorizes the resulting feature vectors into classes. A better way to approach the problem would be to rely as much as possible on learning in the feature extractor itself. In the case of character recognition a network could be fed with almost raw inputs e.g. size-normalized images.

While this can be done with an ordinary fully connected feedforward network with some success for tasks such as character recognition there are problems.

Firstly, typical images are large, often with several hundred variable pixels. A fully connected first layer with say one hundred hidden units in the first layer would already contain several tens of thousands of weights. Such a large number of parameters increases the capacity of the system and therefore requires a larger training set. In addition the memory requirement to store so many weights may rule out certain hardware implementations.

Fully-connected layers for image applications have no built-in invariance with respect to local distortions of the input images. Before being sent to the fixed-input layer of a neural net character images or other 2D or 1D signals must be approximately size-normalized and centered in the input field. Unfortunately no such pre-processing can be perfect: handwriting is often normalized at the word level which can cause size slant and position variations for individual characters. This combined with variability in writing style will cause variations in the position of distinctive features in input objects. In principle a fully connected network of sufficient size could learn to produce outputs that are in variant with respect to such variations. Learning these weight configurations requires a very large number of training instances to cover the space of possible variations. *In convolutional networks shift invariance is automatically obtained by forcing the replication of weight configurations across space.*

Secondly a deficiency of fully connected architectures is that the topology of the input is entirely ignored. The input variables can be presented in any fixed order without affecting the outcome of the training. On the contrary images or time-frequency representations of speech have a strong 2-D local structure: variables or pixels that are spatially or temporally nearby are highly correlated. Local correlations are the reasons for the well known advantages of extracting and combining local features before recognizing spatial or temporal objects because configurations of neighboring variables can be classified into a small number of categories e.g. edges corners. *Convolutional Networks force the extraction of local features by restricting the receptive fields of hidden units to be local.*

3 Model Architecture

3.1 Convolutional Neural Network

CNNs combine three architectural ideas to ensure some degree of shift, scale and distortion invariance: local receptive fields, shared weights, and spatial or temporal sub-sampling (now better known as pooling).

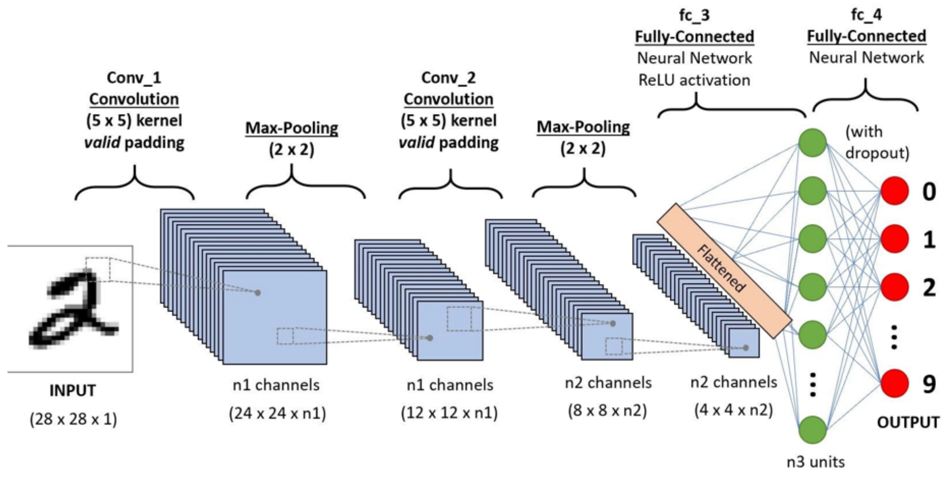


Figure 1: *CNN Architecture*

The input plane receives images of characters that approximately sized-normalized and centered. Each unit in the layer receives inputs form a set of units located in a small neighborhood, i.e. receptive fields in the previous layer. With local receptive fields, neurons can extract elementary visual features such as oriented edges, end-point, corners. These features are then combined by the subsequent layers in order to detect higher-order features. Elementary feature detectors that are useful on one part of the image are likely to be useful across the entire image. This knowledge can be applied by forcing a set of units, whose receptive fields are located at different places on the image, to have identical weight vectors

A set of units in a layer are constrained to have identical weight vectors and biases. These units are organized into a plane called a feature map. The primary benefit is shift invariance: because all units in a feature map perform the same operation on different parts of the image, they detect the same feature regardless of its position. If the input image is shifted, the output on the feature map will be shifted by the same amount but will otherwise be unchanged. Another major benefit is a drastic reduction in the number of free parameters. This reduces the capacity of the network and improves its generalization ability. For example, the LeNet-5 network has over 300,000 connections but only 60,000 trainable parameters due to weight sharing. This operation is equivalent to a convolution with a kernel/filter defined by the shared weights, which gives the network its name. A complete convolutional layer is composed of several feature maps, each extracting a different type of feature at every location.

After a feature's presence is detected by a convolutional layer, its exact location becomes less important than its approximate position relative to other features. Sub-sampling layers reduce the spatial resolution of the feature maps, thereby reducing the sensitivity of the output to shifts and distortions. This is typically achieved by performing a local averaging over a neighborhood (e.g., 2x2) in the previous feature map and then sub-sampling the result. The resulting feature map has fewer rows and columns. Convolutional and sub-sampling layers are typically alternated, creating a structure where spatial resolution is progressively reduced while the number of feature maps is increased.

After the convolutional and sub-sampling layers extract hierarchical features, the resulting feature maps are flattened into a one-dimensional vector that serves as input to one or more fully connected (dense) layers. These dense layers act as a traditional multilayer perceptron classifier, where each neuron connects to all outputs from the previous layer, enabling the network to learn complex non-linear combinations of the extracted features. The final fully connected layer typically has a number of neurons equal to the number of classes in the classification task, with a softmax activation function that converts the raw output scores into class probabilities.

3.2 LeNet-5

LeNet-5 is a specific 7-layer Convolutional Neural Network designed for digit recognition that serves as the paper's primary example. The architecture is detailed layer by layer.

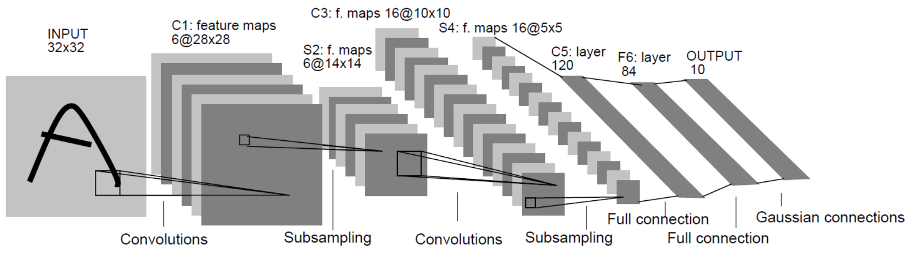


Figure 2: *LeNet-5 Architecture*

Input:

The input is a 32x32 pixel image. This is significantly larger than the largest character in the dataset (which is at most 20x20 pixels centered in a 28x28 field) to ensure that distinctive features like stroke endpoints can appear in the center of the receptive fields of the final convolutional layer's units.

LeNet-5 Architecture Layers:

The layers are labeled Cx for convolution, Sx for sub-sampling, and Fx for fully-connected.

C1: This layer has 6 feature maps. Each unit in each feature map is connected to a 5x5 local receptive field.The size of the feature maps is 28x28 which prevents connection from the input from falling off the boundary. This layer has 156 trainable parameters and 122304 connections.

S2: This layer has 6 feature maps of size 14x14. Each unit in each feature map is connected to a 2x2 non-overlapping neighborhood in the corresponding feature map in C1. It computes an average of its four inputs, applies a trainable coefficient and bias, and passes the result through a sigmoidal function. This layer has 12 trainable parameters adn 5880 connections.

C3: This layer has 16 feature maps, each of size 10x10. Each unit is connected to 5x5 neighborhoods in a subset of the feature maps from S2. This incomplete connection scheme is designed to break symmetry in the network, forcing different feature maps to extract different, complementary features. This layer has 1,516 trainable parameters adn 151600 connections.

S4: This layer has 16 feature maps, each 5x5. It functions similarly to S2, with each unit connected to a 2x2 neighborhood in the corresponding map in C3. It has 32 trainable parameters and 2000 connections.

C5: This layer has 120 feature maps, each 1x1. Each unit is connected to a 5x5 neighborhood across all 16 of S4's feature maps. Because the input maps (from S4) are also 5x5, this amounts to a full connection between S4 and C5. This layer has 48120 trainable connections.

F6: This layer contains 84 units and is fully connected to all 120 units of C5. It has 10,164 trainable parameters. As in classical neural networks, units in layers up to F6 compute a dot product of inputs and weights, to which a bias is added. This weighted sum, denoted by a_i for unit i, is then passed through a sigmoid squashing function to product the state of unit i, denoted by x_i:

$$x_i = f(a_i)$$

The squashing function is a scaled hyperbolic tangent:

$$f(a) = Atanh(S_a)$$

where A is the amplitude of the function and S determines its slope at the origin. The function f is off, with horizontal asymptotes at $+A$ and $-A$. The constant A is chosen to be 1.7159.

Output Layer (RBF): The final layer consists of Euclidean Radial Basis Function (RBF) units, one for each class with 84 inputs each. The outputs of each RBF unit y_i is computed as follows:

$$y_i = \sum_j (x_j - w_{ij})^2$$

- Each RBF unit computes the Euclidean distance between its input vector (from F6) and its own parameter vector. The output is a penalty measuring the fit between the input pattern and the class model.
- The RBF parameter vectors are not learned initially but are designed to represent stylized 7x12 bitmap images of the characters (hence the 84 units in F6).
- This design creates distributed output codes, where confusable characters (e.g., 'O', 'o', and '0') have similar target vectors. This helps if the system is combined with a linguistic post-processor. It is also better for rejecting non-characters than standard sigmoid outputs.
- The components of these target vectors are +1 or -1, which keeps the units in the preceding F6 layer from saturating and slowing down convergence.

3.3 Loss Function

While the mean squared error (MSE) is simple and commonly used, it is not ideal for classification tasks since it does not enforce strong separation between classes. To overcome this, the authors adopt a likelihood-based criterion (negative log-likelihood), which directly maximizes the probability of the correct class while penalizing incorrect ones. This shift ensures clearer decision boundaries, faster convergence, and better classification performance—an insight that foreshadowed the later dominance of cross-entropy loss in deep learning.

4 LeNet-5 Keras Implementation

Below is the code for python implementation of the LeNet-5 CNN model on the MNIST dataset. The model has a test accuracy of 98.62%. You will also find plots of Accuracy vs. Epochs for training and test data, Loss vs. Epochs for training and test data along with the test confusion matrix.

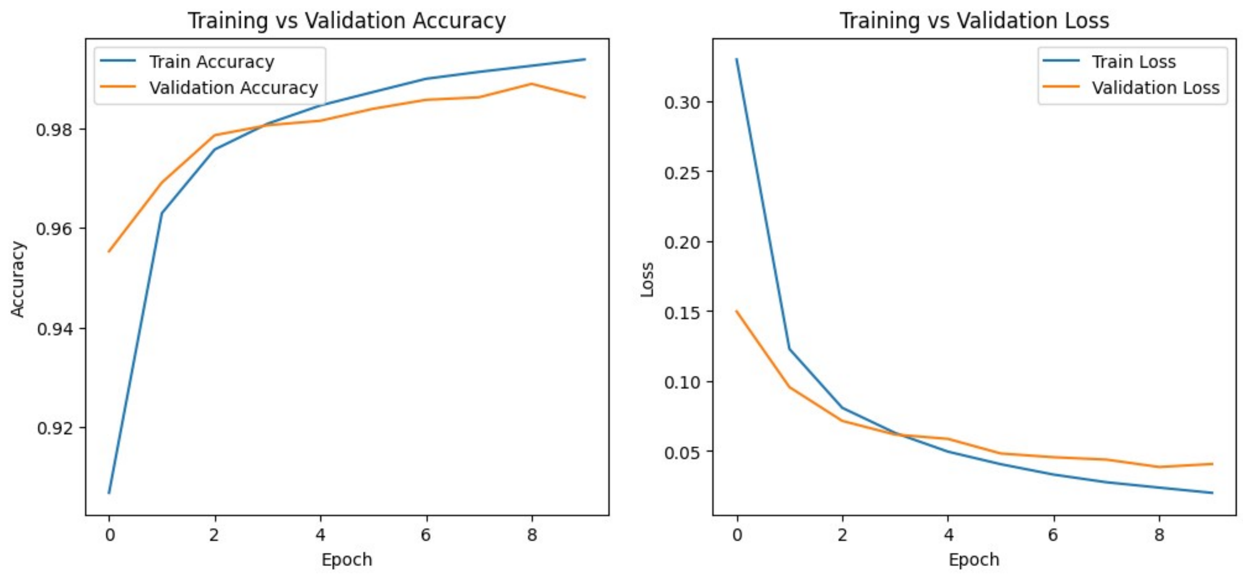
```python
import tensorflow
import numpy as np

from tensorflow import keras
from keras.layers import Dense,Conv2D,Flatten,AveragePooling2D
from keras import Sequential
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize
X_train = X_train.astype('float32') / 255.0
X_test  = X_test.astype('float32') / 255.0

# Add channel dimension
X_train = np.expand_dims(X_train, -1)  # (60000,28,28,1)
X_test  = np.expand_dims(X_test, -1)   # (10000,28,28,1)

# Pad to 32x32 (mnist images are 28x28 but the original LeNet-5 used
32x32)
X_train = np.pad(X_train, ((0,0),(2,2),(2,2),(0,0)), 'constant')
X_test  = np.pad(X_test, ((0,0),(2,2),(2,2),(0,0)), 'constant')

model = Sequential()

model.add(Conv2D(6, kernel_size=(5,5), padding='valid',
activation='tanh', input_shape=(32,32,1)))
model.add(AveragePooling2D(pool_size=(2,2), strides=2,
padding='valid'))

model.add(Conv2D(16, kernel_size=(5,5), padding='valid',
activation='tanh'))
model.add(AveragePooling2D(pool_size=(2,2), strides=2,
padding='valid'))

# C5 (Conv with 120 filters of size 5x5 → outputs 1x1x120)
model.add(Conv2D(120, kernel_size=5, activation='tanh'))

model.add(Flatten())
model.add(Dense(84, activation='tanh'))
model.add(Dense(10, activation='softmax'))
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/
convolutional/base_conv.py:113: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```python
model.summary()
```

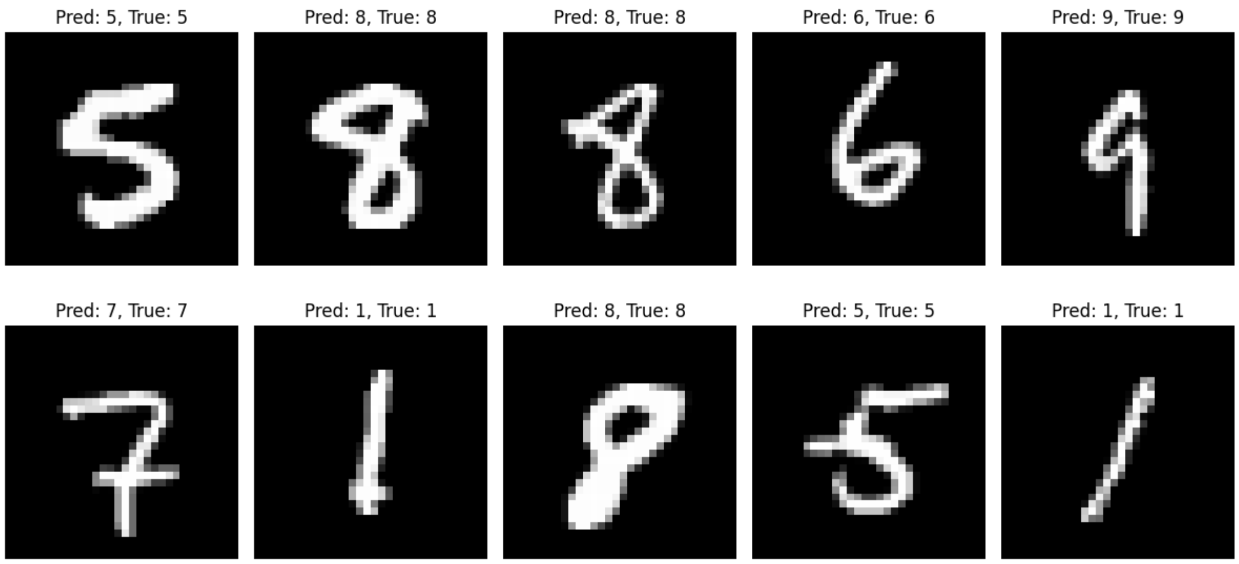
```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv2d (Conv2D)                  │ (None, 28, 28, 6)       │
156 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ average_pooling2d                │ (None, 14, 14, 6)       │
0 │
│ (AveragePooling2D)               │                         │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ conv2d_1 (Conv2D)                │ (None, 10, 10, 16)      │
2,416 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ average_pooling2d_1              │ (None, 5, 5, 16)        │
0 │
│ (AveragePooling2D)               │                         │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ conv2d_2 (Conv2D)                │ (None, 1, 1, 120)       │
48,120 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ flatten (Flatten)                │ (None, 120)             │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dense (Dense)                    │ (None, 84)              │
10,164 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dense_1 (Dense)                  │ (None, 10)              │
850 │
└─────────────────────────────────┴────────────────────────┴──────────
─────┘

 Total params: 61,706 (241.04 KB)

 Trainable params: 61,706 (241.04 KB)

 Non-trainable params: 0 (0.00 B)
```

```
model.compile(optimizer="adam",
loss="sparse_categorical_crossentropy", metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=10, batch_size=128,
validation_data=(X_test, y_test))

Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 34s 68ms/step - accuracy: 0.8398 - loss:
0.5965 - val_accuracy: 0.9553 - val_loss: 0.1498
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 40s 66ms/step - accuracy: 0.9584 - loss:
0.1366 - val_accuracy: 0.9691 - val_loss: 0.0956
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 31s 66ms/step - accuracy: 0.9736 - loss:
0.0863 - val_accuracy: 0.9786 - val_loss: 0.0715
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 32s 68ms/step - accuracy: 0.9799 - loss:
0.0661 - val_accuracy: 0.9806 - val_loss: 0.0617
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 40s 65ms/step - accuracy: 0.9845 - loss:
0.0509 - val_accuracy: 0.9815 - val_loss: 0.0587
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 30s 65ms/step - accuracy: 0.9871 - loss:
0.0395 - val_accuracy: 0.9839 - val_loss: 0.0482
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 30s 64ms/step - accuracy: 0.9896 - loss:
0.0332 - val_accuracy: 0.9857 - val_loss: 0.0455
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 43s 67ms/step - accuracy: 0.9915 - loss:
0.0275 - val_accuracy: 0.9862 - val_loss: 0.0438
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 30s 64ms/step - accuracy: 0.9930 - loss:
0.0232 - val_accuracy: 0.9889 - val_loss: 0.0385
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 41s 64ms/step - accuracy: 0.9940 - loss:
0.0197 - val_accuracy: 0.9862 - val_loss: 0.0407

test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_acc*100:.2f}%")

Test Accuracy: 98.62%

import matplotlib.pyplot as plt

# Plot Accuracy
plt.figure(figsize=(12,5))

plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training vs Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plot Loss
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training vs Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

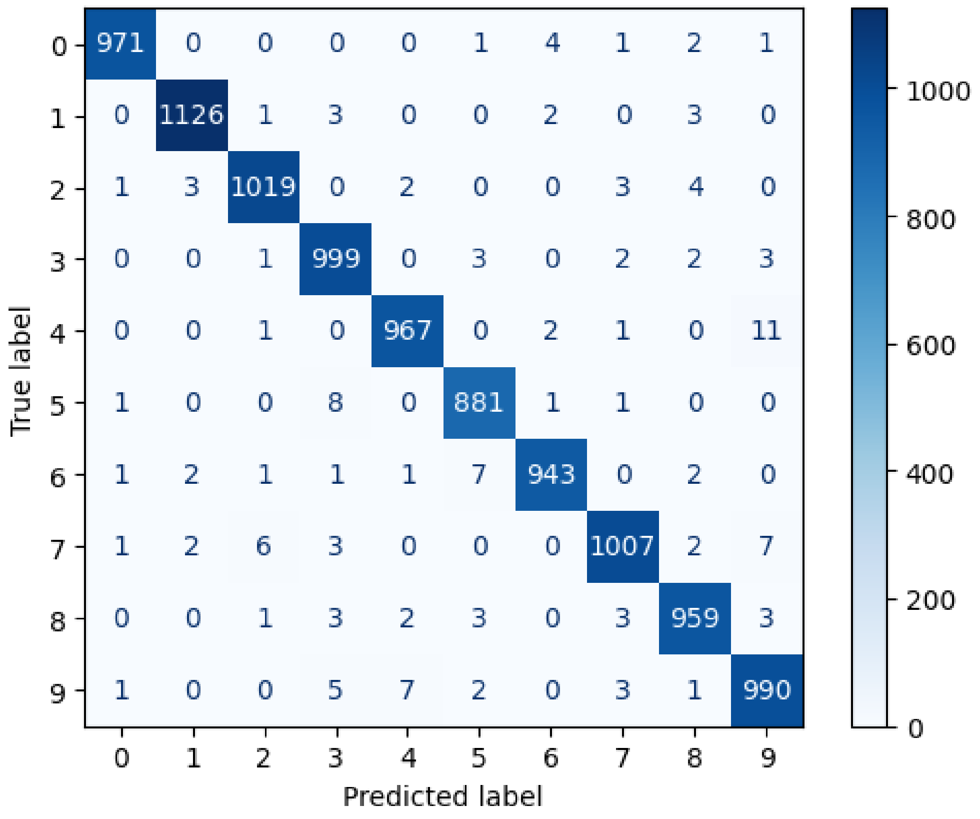


```
import numpy as np

# Get predictions
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Show some test images with predicted labels
plt.figure(figsize=(12,6))
for i in range(10):
    idx = np.random.randint(0, X_test.shape[0])
    plt.subplot(2,5,i+1)
    plt.imshow(X_test[idx].reshape(32,32), cmap='gray')
    plt.title(f"Pred: {y_pred_classes[idx]}, True: {y_test[idx]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step
```



```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = y_test

cm = confusion_matrix(y_true, y_pred_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=np.arange(10))
disp.plot(cmap=plt.cm.Blues)
plt.show()

313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step
```



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	971	0	0	0	0	1	4	1	2	1
1	0	1126	1	3	0	0	2	0	3	0
2	1	3	1019	0	2	0	0	3	4	0
3	0	0	1	999	0	3	0	2	2	3
4	0	0	1	0	967	0	2	1	0	11
5	1	0	0	8	0	881	1	1	0	0
6	1	2	1	1	1	7	943	0	2	0
7	1	2	6	3	0	0	0	1007	2	7
8	0	0	1	3	2	3	0	3	959	3
9	1	0	0	5	7	2	0	3	1	990

Part II

A Review of AlexNet

1 Context

The introduction of AlexNet (Krizhevsky, Sutskever, and Hinton, 2012) represented a turning point in computer vision and deep learning. Before this breakthrough, visual recognition systems faced significant limitations due to the small scale of available image datasets and restricted computational power. The benchmarks available at the time—such as contained only tens of thousands of labeled images, which prevented models from effectively generalizing to the complexity of real-world visual tasks. However, two key developments changed this landscape: the creation of the ImageNet dataset, which provided millions of high-resolution images spanning thousands of object categories, and the growing computational capabilities of graphics processing units (GPUs), which made it feasible to train much deeper neural networks.

Against this backdrop, AlexNet showed that deep convolutional neural networks (CNNs) could dramatically outperform conventional machine learning approaches in visual object classification when trained on large-scale datasets using GPUs. Its decisive win in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) firmly established CNNs as the leading approach for image recognition and sparked widespread adoption of deep learning methods throughout the computer vision research community.

2 Dataset

AlexNet was trained and evaluated on the ILSVRC datasets from 2010 and 2012. The dataset comprises approximately 1.2 million training images, 50,000 validation images, and 150,000 test images, distributed across 1,000 object categories. All images were rescaled to a uniform spatial resolution of 256×256 pixels, from which random 224×224 crops were extracted for model training. No extensive pre-processing was performed beyond mean subtraction from each pixel channel. Model performance was reported using two standard metrics: the top-1 error rate, denoting the proportion of images where the highest-probability class prediction was incorrect, and the top-5 error rate, representing the fraction where the correct label was not among the five most probable outputs.

3 Model Architecture

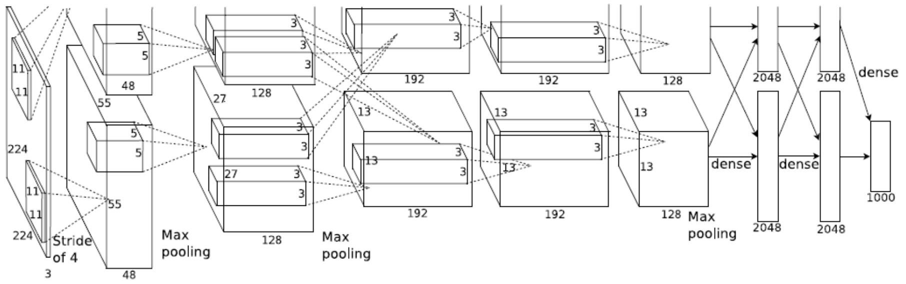


Figure 3: *AlexNet Architecture*

The AlexNet architecture comprises eight learnable layers, including five convolutional layers followed by three fully connected layers, culminating in a 1,000-way softmax output. The network contained approximately 60 million parameters and 650,000 neurons. The architecture introduced several key innovations that collectively contributed to its superior performance:

- Rectified Linear Units (ReLU): The model employed non-saturating ReLU activation functions (f(x) = max(0, x)), which significantly accelerated convergence compared to traditional sigmoid or tanh activations.
- GPU Parallelization: The network was distributed across two NVIDIA GTX 580 GPUs, each handling a subset of the model's kernels. This parallelization enabled training of a network size that exceeded the memory capacity of a single GPU.
- Local Response Normalization (LRN): A biologically inspired lateral inhibition mechanism was introduced to promote competition among neuron activations, improving generalization and reducing error rates.
- Overlapping Max-Pooling: Unlike prior approaches that used non-overlapping pooling windows, AlexNet used overlapping regions (z = 3, s = 2), which helped reduce overfitting slightly and improved feature robustness.
- The first convolutional layer employed 96 kernels of size 11×11×3 with a stride of 4; subsequent convolutional layers used progressively smaller filters (5×5 and 3×3) with increasing depth (up to 384 channels). The final two fully connected layers each consisted of 4,096 neurons, followed by the softmax output layer.

4 Regularization and Overfitting Mitigation

Given the network's substantial capacity, the authors employed multiple strategies to mitigate overfitting:

1. Data Augmentation: Two augmentation strategies were used.
   - Geometric Transformations: Random translations and horizontal reflections were applied to 224×224 patches extracted from the 256×256 images.
   - Color Perturbation: Principal Component Analysis (PCA) was applied to RGB pixel intensities, and random noise was added along the principal components to simulate variations in lighting and color.

   These transformations effectively expanded the size of the training dataset without requiring additional storage or preprocessing.
2. Dropout Regularization: Dropout was applied to the two fully connected layers, randomly disabling 50% of the neurons during each training iteration. This prevented complex co-adaptations among neurons and improved the model's generalization capacity, albeit doubling the number of iterations required for convergence.
3. Weight Decay: A small L2 regularization term (lambda = 0.0005) was used to penalize large weight magnitudes, thereby constraining model complexity and stabilizing learning.

5 Results

AlexNet achieved a top-1 error rate of 37.5% and a top-5 error rate of 17.0% on the ILSVRC-2010 test set—representing a dramatic improvement over the previous state-of-the-art (top-5 error of 25.7%). In the ILSVRC-2012 competition, a single AlexNet model achieved a top-5 test error of 18.2%, while an ensemble of seven models, including those pre-trained on a larger ImageNet subset, reduced this to 15.3%. The second-best entry in that competition recorded a 26.2% top-5 error rate, underscoring the substantial performance gain achieved by AlexNet.

The study conclusively demonstrated that deep, hierarchical architectures trained on large-scale data could significantly outperform traditional computer vision pipelines based on hand-engineered features. This result not only won the ILSVRC-2012 competition but also established the empirical foundation for subsequent architectures such as VGGNet, GoogLeNet, and ResNet.

Implementation of AlexNet on a Dogs and Cats dataset

```
import tensorflow_datasets as tfds
```

Loading the dataset

```
(train_dataset, test_dataset), info = tfds.load(
    'cats_vs_dogs',
    split = ('train[:80%]', 'train[80%:]'),
    with_info = True,
    as_supervised=True)
```

```
WARNING:absl:Variant folder
/root/tensorflow_datasets/cats_vs_dogs/4.0.1 has no dataset_info.json

Downloading and preparing dataset Unknown size (download: Unknown
size, generated: Unknown size, total: Unknown size) to
/root/tensorflow_datasets/cats_vs_dogs/4.0.1...
```

{"model_id":"4bfa634a99414663ad9dc50564608fbb","version_major":2,"version_minor":0}

{"model_id":"c4b752d687d14c7c8a63485e0a7cc607","version_major":2,"version_minor":0}

{"model_id":"df8166ec96e14da8beaba47ba2b2f933","version_major":2,"version_minor":0}

{"model_id":"7998d650ccf4440b9ff2c0cf55752060","version_major":2,"version_minor":0}

```
WARNING:absl:1738 images were corrupted and were skipped
```

{"model_id":"6d032c81bf974b1587a9be9f6dd1461d","version_major":2,"version_minor":0}

```
Dataset cats_vs_dogs downloaded and prepared to
/root/tensorflow_datasets/cats_vs_dogs/4.0.1. Subsequent calls will
reuse this data.
```

```
train_dataset
```

```
<_PrefetchDataset element_spec=(TensorSpec(shape=(None, None, 3),
dtype=tf.uint8, name=None), TensorSpec(shape=(), dtype=tf.int64,
name=None))>
```

This is a tensorflow dataset and not a numpy arrays.

```
len(train_dataset), len(test_dataset)
```

```
(18610, 4652)
```

There are 18610 train set images and 4652 test set images

```
for X, y in train_dataset:
  print(X.shape, y.numpy())
  image_1 = X.numpy()
  break
```

```
(262, 350, 3) 1
```

For the first image: X is (262, 350, 3) and the label y is 1 which denotes that the image is a dog.

```
import matplotlib.pyplot as plt
plt.imshow(image_1)
```

```
<matplotlib.image.AxesImage at 0x7a1027e034a0>
```

```
import tensorflow as tf

def normalize_img(image, label):
  return tf.cast(image, tf.float32) / 255.0, label

def resize(image, label):
  return (tf.image.resize(image, (224, 224)), label)
```

The normalize function divides each value by 255. The resize function resizes the images to a 224 by 224 image. These functions have been made so we can map them.

```
train_dataset = train_dataset.map(resize, num_parallel_calls =
tf.data.AUTOTUNE)
train_dataset = train_dataset.map(normalize_img, num_parallel_calls =
tf.data.AUTOTUNE)
```

we (lazily) mapped the two functions. the two lines basically says to resize and normalise all the images.

```
SHUFFLE_VAL = len(train_dataset) // 1000
BATCH_SIZE = 4

train_dataset = train_dataset.shuffle(SHUFFLE_VAL)
train_dataset = train_dataset.batch(BATCH_SIZE)

train_dataset = train_dataset.prefetch(tf.data.AUTOTUNE)

test_dataset = test_dataset.map(resize, num_parallel_calls =
tf.data.AUTOTUNE)
test_dataset = test_dataset.map(normalize_img, num_parallel_calls =
tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE)
test_dataset = test_dataset.prefetch(tf.data.AUTOTUNE)
```

A pipeline for train and test images ^^

```
for (img, label) in train_dataset:
  print(img.numpy().shape)
  break
```

```
(4, 224, 224, 3)
```

Model Training begins here

```
from tensorflow.keras import layers
from tensorflow.keras.models import Model

def AlexNet():
  inp = layers.Input((224, 224, 3))
  x = layers.Conv2D(96, 11, strides = 4, activation = 'relu')(inp)
  x = layers.BatchNormalization()(x)
  x = layers.MaxPool2D(3, strides = 2)(x)
  x = layers.Conv2D(256, 5, 1, activation = 'relu')(x)
  x = layers.BatchNormalization()(x)
  x = layers.MaxPool2D(3, strides = 2)(x)
  x = layers.Conv2D(384, 3, 1, activation='relu')(x)
  x = layers.Conv2D(384, 3, 1, activation='relu')(x)
  x = layers.Conv2D(256, 3, 1, activation='relu')(x)
  x = layers.MaxPool2D(3, 2)(x)
  x = layers.Flatten()(x)
  x = layers.Dense(4096, activation='relu')(x)
  x = layers.Dropout(0.5)(x)
  x = layers.Dense(4096, activation='relu')(x)
  x = layers.Dropout(0.5)(x)
  x = layers.Dense(1, activation='sigmoid')(x)

  model = Model(inp, x)
  return model

model = AlexNet()
model.summary()
```

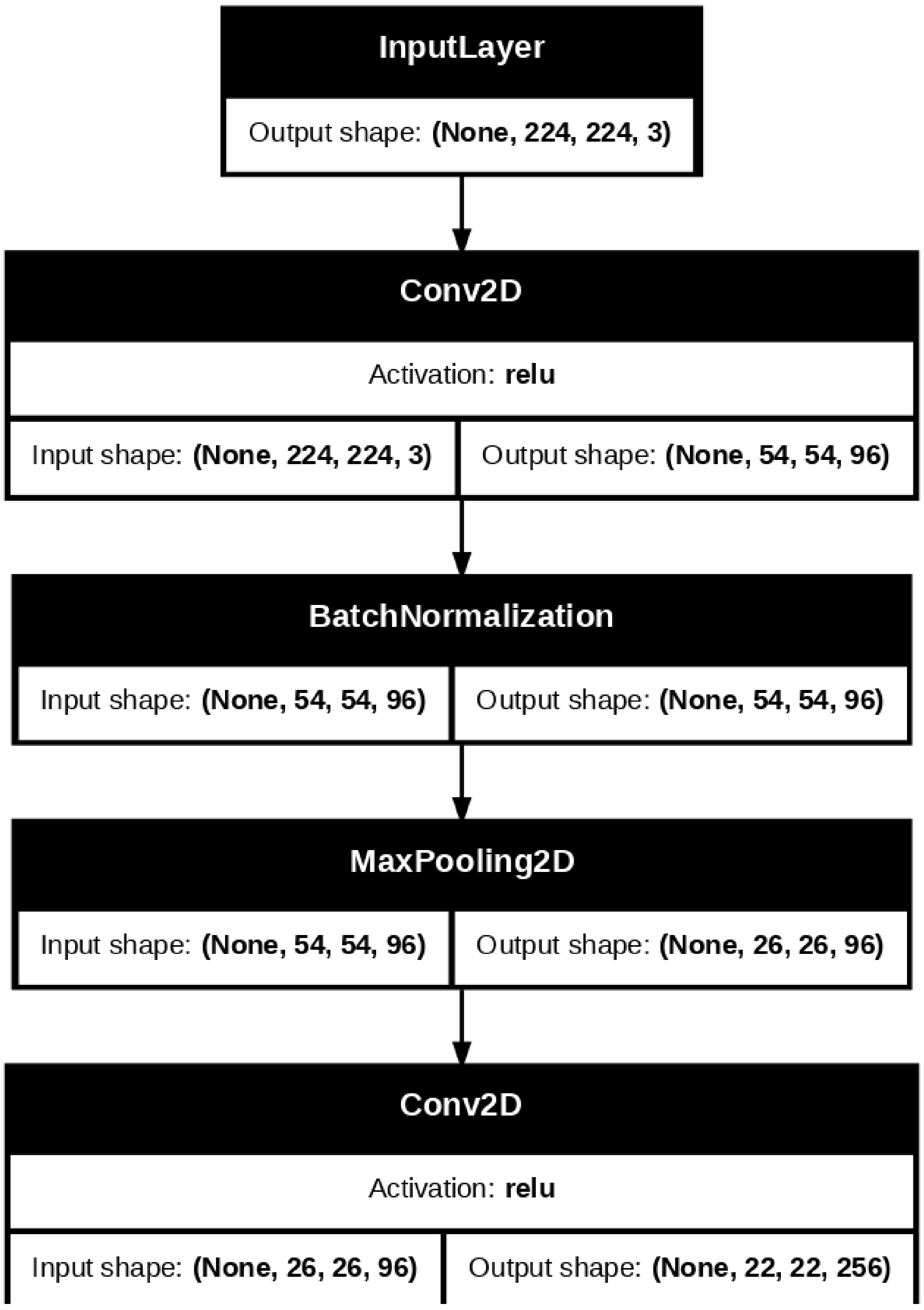
```
Model: "functional"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer (InputLayer)         │ (None, 224, 224, 3)     │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ conv2d (Conv2D)                  │ (None, 54, 54, 96)      │
34,944 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ batch_normalization              │ (None, 54, 54, 96)      │
384 │
│ (BatchNormalization)             │                         │
│
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ max_pooling2d (MaxPooling2D)     │ (None, 26, 26, 96)      │
0 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ conv2d_1 (Conv2D)                │ (None, 22, 22, 256)     │
614,656 │
├──────────────────────────────────┼─────────────────────────┼──────────
─────┤
│ batch_normalization_1            │ (None, 22, 22, 256)     │
1,024 │
│ (BatchNormalization)             │                         │
│
├──────────────────────────────────┼─────────────────────────┼──────────
```

```
───────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 10, 10, 256)    │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ conv2d_2 (Conv2D)               │ (None, 8, 8, 384)      │
885,120 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ conv2d_3 (Conv2D)               │ (None, 6, 6, 384)      │
1,327,488 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ conv2d_4 (Conv2D)               │ (None, 4, 4, 256)      │
884,992 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ max_pooling2d_2 (MaxPooling2D)  │ (None, 1, 1, 256)      │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ flatten (Flatten)               │ (None, 256)            │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ dense (Dense)                   │ (None, 4096)           │
1,052,672 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ dropout (Dropout)               │ (None, 4096)           │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ dense_1 (Dense)                 │ (None, 4096)           │
16,781,312 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ dropout_1 (Dropout)             │ (None, 4096)           │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
───────┤
│ dense_2 (Dense)                 │ (None, 1)              │
4,097 │
└─────────────────────────────────┴────────────────────────┴──────────
───────┘

 Total params: 21,586,689 (82.35 MB)

 Trainable params: 21,585,985 (82.34 MB)
```

```
Non-trainable params: 704 (2.75 KB)
```

Batch Normalisation does not exist in the original AlexNet but it improves the AlexNet model. We also used a sigmoid function instead of a 1000-softmax since we just classifying 2 classes.

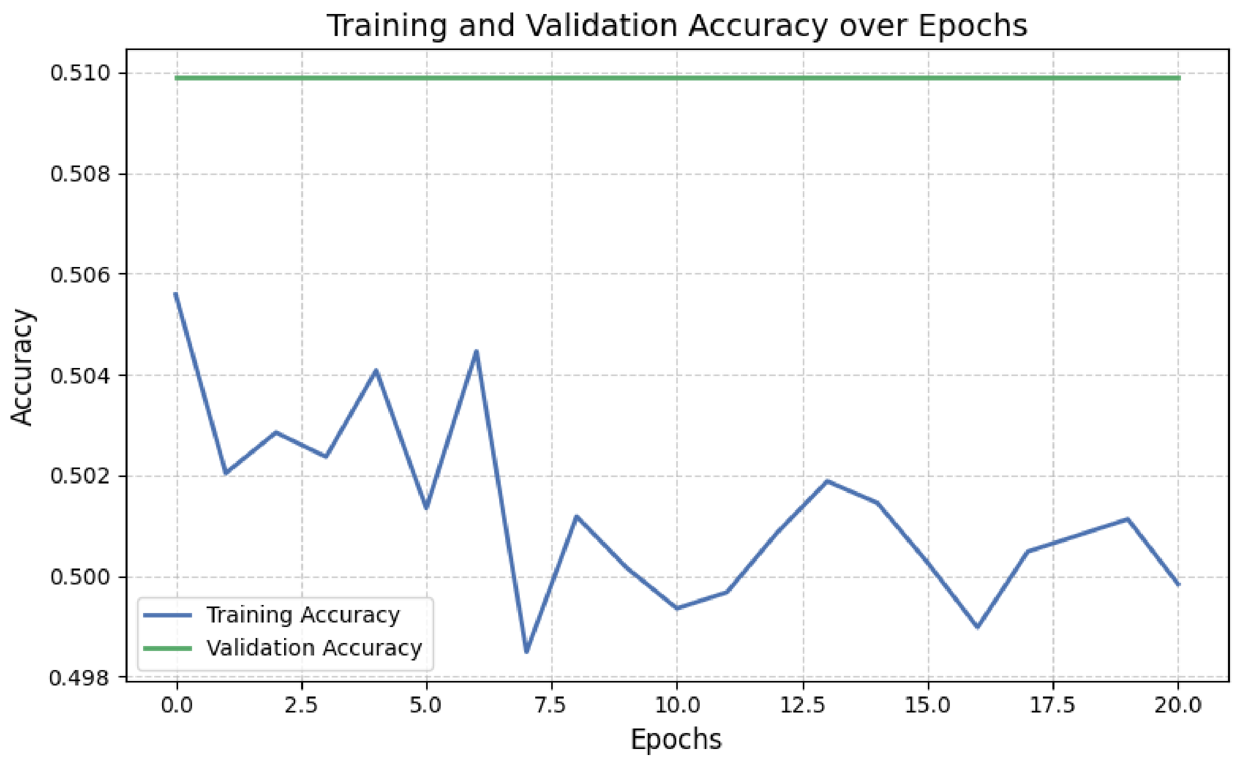
```
tf.keras.utils.plot_model(
    model,
    to_file='model.png',
    show_shapes=True,
    show_dtype=False,
    show_layer_names=False,
    show_layer_activations=True,
    dpi=100
)
```


InputLayer
Output shape: (None, 224, 224, 3)
Conv2D
Activation: relu
Input shape: (None, 224, 224, 3)
Output shape: (None, 54, 54, 96)
BatchNormalization
Input shape: (None, 54, 54, 96)
Output shape: (None, 54, 54, 96)
MaxPooling2D
Input shape: (None, 54, 54, 96)
Output shape: (None, 26, 26, 96)
Conv2D
Activation: relu
Input shape: (None, 26, 26, 96)
Output shape: (None, 22, 22, 256)

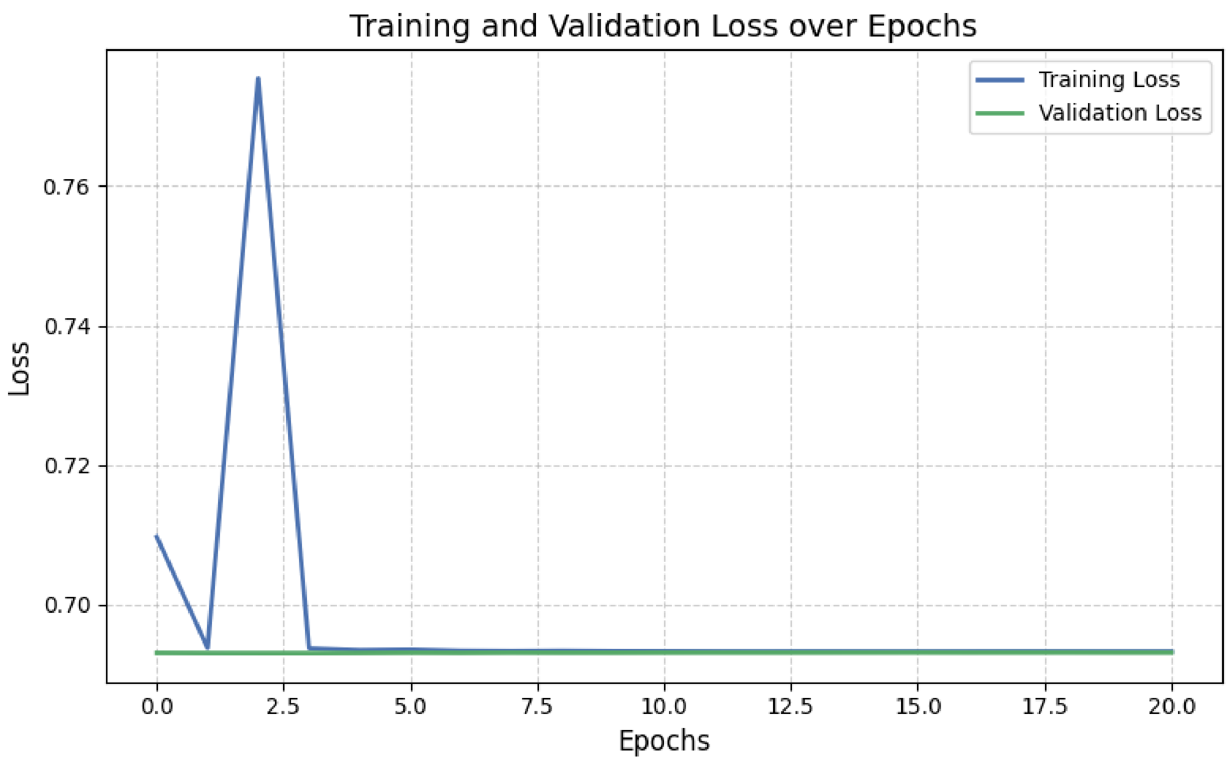
```
for (img, label) in train_dataset:
  print(model(img).numpy().shape, label.numpy())
  break

(4, 1) [0 0 1 1]

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import BinaryCrossentropy

model.compile(loss=BinaryCrossentropy(),
              optimizer=Adam(learning_rate=0.001),
metrics=['accuracy'])

from tensorflow.keras.callbacks import EarlyStopping

es = EarlyStopping(patience=5,
                   monitor='loss')

model.fit(train_dataset, epochs=100, validation_data=test_dataset,
          callbacks=[es])

Epoch 1/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 94s 18ms/step - accuracy: 0.5029 -
loss: 0.7819 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 2/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5039 -
loss: 0.6936 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 3/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 80s 17ms/step - accuracy: 0.5019 -
loss: 0.7189 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 4/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 79s 17ms/step - accuracy: 0.5039 -
loss: 0.6935 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 5/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 82s 18ms/step - accuracy: 0.5052 -
loss: 0.6932 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 6/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5027 -
loss: 0.6935 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 7/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5065 -
loss: 0.6933 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 8/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 80s 17ms/step - accuracy: 0.5000 -
loss: 0.6933 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 9/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 80s 17ms/step - accuracy: 0.5062 -
loss: 0.6932 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 10/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5014 -
loss: 0.6932 - val_accuracy: 0.5099 - val_loss: 0.6930
```

```
Epoch 11/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 79s 17ms/step - accuracy: 0.4998 -
loss: 0.6932 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 12/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 80s 17ms/step - accuracy: 0.4983 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 13/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 79s 17ms/step - accuracy: 0.4996 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 14/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5021 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 15/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 80s 17ms/step - accuracy: 0.5021 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 16/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.4992 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 17/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5000 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 18/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 83s 18ms/step - accuracy: 0.5013 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 19/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.4992 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 20/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 81s 17ms/step - accuracy: 0.5001 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930
Epoch 21/100
4653/4653 ━━━━━━━━━━━━━━━━━━━━ 80s 17ms/step - accuracy: 0.4978 -
loss: 0.6931 - val_accuracy: 0.5099 - val_loss: 0.6930

<keras.src.callbacks.history.History at 0x7a0faf035880>
```

```python
history = model.history

import matplotlib.pyplot as plt

plt.style.use('seaborn-v0_8-deep')  # modern clean style
plt.figure(figsize=(8, 5))
plt.plot(model.history.history['accuracy'], label='Training Accuracy',
linewidth=2)
plt.plot(model.history.history['val_accuracy'], label='Validation
Accuracy', linewidth=2)
plt.title('Training and Validation Accuracy over Epochs', fontsize=14)
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Accuracy', fontsize=12)
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```



```
plt.style.use('seaborn-v0_8-deep')  # modern clean style
plt.figure(figsize=(8, 5))
plt.plot(model.history.history['loss'], label='Training Loss',
linewidth=2)
plt.plot(model.history.history['val_loss'], label='Validation Loss',
linewidth=2)
plt.title('Training and Validation Loss over Epochs', fontsize=14)
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Loss', fontsize=12)
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```


Training and Validation Loss over Epochs
Training Loss
Validation Loss
Loss
0.76
0.74
0.72
0.70
Epochs
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] D. Rumelhart, G. Hinton, and R. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, pp. 533–536, 1986.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems (NIPS)*, vol. 25, pp. 1097–1105, 2012.