LAB 3: CONVOLUTIONAL NEURAL NETWORKS (CNN)

Machine Learning Hardware Course

OVERVIEW

This lab focuses on Convolutional Neural Networks (CNNs), a specialized type of neural network designed for processing structured grid-like data, particularly images. Building on previous labs, you will implement increasingly complex CNN architectures, visualize learned features, experiment with different hyperparameters, and analyze the computational requirements and hardware efficiency of CNN models. By working through this lab, you will gain a deeper understanding of how CNN architecture choices affect model performance, memory usage, and computational demands.

LEARNING OBJECTIVES

By the end of this lab, you will be able to:

1. Implement and train CNNs with various architectures
2. Understand the impact of convolutional layers, pooling operations, and filter sizes on model performance
3. Visualize and interpret feature maps and filters learned by CNN layers
4. Measure and analyze computational requirements of different CNN architectures
5. Compare CNNs with Fully Connected Neural Networks (FCNNs) in terms of efficiency and performance
6. Apply transfer learning techniques with pre-trained CNN models
7. Make informed architectural decisions for CNN models based on performance-efficiency trade-offs

---

PREREQUISITES

- Completion of Labs 1 and 2
- Understanding of neural network basics
- Familiarity with TensorFlow/Keras
- Basic knowledge of image processing concepts

TIME ALLOCATION

Total time: 2 hours (120 minutes)

Activity	Duration
Environment Setup	10 minutes
CNN Fundamentals	10 minutes
Basic CNN Implementation	20 minutes
Architectural Exploration	30 minutes
Feature Visualization	20 minutes
Transfer Learning	20 minutes
Performance Analysis	10 minutes

REQUIRED MATERIALS

- Computer with internet access
- Google account for Colab
- This lab guide
- Graded worksheet (provided separately)

LAB SETUP INSTRUCTIONS

Accessing Google Colab

1. Open your web browser and navigate to https://colab.research.google.com
2. Sign in with your Google account
3. Create a new notebook by clicking on "New Notebook"
4. Rename your notebook to "Lab3_CNN_YourName"

Setting Up Environment

python

Copy

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import time
import tensorflow as tf
from tensorflow.keras.datasets import mnist, cifar10
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import (
    Dense, Dropout, Flatten, Input, Conv2D, MaxPooling2D, AveragePooling2D,
    BatchNormalization, GlobalAveragePooling2D
)
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.applications import VGG16, MobileNetV2
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
import psutil
import os

# Check TensorFlow version and GPU availability
print("TensorFlow version:", tf.__version__)
print("GPU Available: ", tf.config.list_physical_devices('GPU'))

# Mount Google Drive (optional, for saving results)
from google.colab import drive
drive.mount('/content/drive')
!mkdir -p "/content/drive/My Drive/ML_Hardware_Course/Lab3"

# Set random seed for reproducibility
tf.random.set_seed(42)
np.random.seed(42)
```

PART 1: CNN FUNDAMENTALS (10 minutes)

Let's start by understanding the key components of CNN architectures.

python

Copy

```python
# Helper function to display informative graphics
def display_cnn_fundamentals():
    """Display graphics explaining CNN fundamentals."""
    # Create a figure for explaining convolution operations
    plt.figure(figsize=(15, 10))
    plt.suptitle("CNN Fundamentals", fontsize=16)

    # 1. Convolution operation
    plt.subplot(2, 2, 1)
    plt.title("1. Convolution Operation")
    # Dummy image to represent convolution
    plt.text(0.5, 0.5, "Filter slides across image\ncomputing dot products",
             ha='center', va='center', fontsize=12)
    plt.axis('off')

    # 2. Pooling operation
    plt.subplot(2, 2, 2)
    plt.title("2. Pooling Operation")
    # Dummy image to represent pooling
    plt.text(0.5, 0.5, "Reduces spatial dimensions\nby taking max/average values",
             ha='center', va='center', fontsize=12)
    plt.axis('off')

    # 3. CNN Architecture
    plt.subplot(2, 2, 3)
    plt.title("3. Typical CNN Architecture")
    # Dummy image to represent architecture
    plt.text(0.5, 0.5, "Conv -> Pool -> Conv -> Pool -> Flatten -> Dense",
             ha='center', va='center', fontsize=12)
    plt.axis('off')

    # 4. Feature Hierarchy
    plt.subplot(2, 2, 4)
    plt.title("4. Hierarchical Feature Learning")
    # Dummy image to represent feature hierarchy
    plt.text(0.5, 0.5, "Early layers: Edges, textures\nLater layers: Shapes, objects",
             ha='center', va='center', fontsize=12)
    plt.axis('off')

    plt.tight_layout()
    plt.subplots_adjust(top=0.9)
    plt.show()
```

```
# Display CNN fundamentals
display_cnn_fundamentals()
```

Key Components of CNNs

1. **Convolutional Layers**: The core building block
   - Apply filters (kernels) to detect spatial patterns
   - Parameters: number of filters, filter size, stride, padding
2. **Pooling Layers**: Reduce spatial dimensions
   - Max Pooling: Take maximum value in a region
   - Average Pooling: Take average of values in a region
   - Parameters: pool size, stride
3. **Activation Functions**: Introduce non-linearity
   - ReLU: Most common for CNNs
   - Others: LeakyReLU, ELU, etc.
4. **Fully Connected Layers**: Usually at the end of the network
   - Connect to all activations in the previous layer
   - Used for final classification/regression

PART 2: DATASET PREPARATION (10 minutes)

We'll use both the MNIST and CIFAR-10 datasets for our CNN experiments.

python

Copy

```python
def load_and_prepare_mnist():
    """
    Load and prepare MNIST dataset for CNN training.

    Returns:
        tuple: Training, validation, and test data
    """
    # Load MNIST dataset
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Reshape for CNN (add channel dimension): (samples, height, width, channels)
    x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
    x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)

    # Normalize pixel values
    x_train = x_train.astype('float32') / 255.0
    x_test = x_test.astype('float32') / 255.0

    # One-hot encode labels
    y_train_encoded = to_categorical(y_train, 10)
    y_test_encoded = to_categorical(y_test, 10)

    # Create validation set
    val_size = 6000
    x_val = x_train[-val_size:]
    y_val = y_train_encoded[-val_size:]
    x_train_final = x_train[:-val_size]
    y_train_final = y_train_encoded[:-val_size]

    print(f"MNIST shapes:")
    print(f"  Training set: {x_train_final.shape}")
    print(f"  Validation set: {x_val.shape}")
    print(f"  Test set: {x_test.shape}")

    return (x_train_final, y_train_final), (x_val, y_val), (x_test, y_test_encoded), y_te

def load_and_prepare_cifar10():
    """
    Load and prepare CIFAR-10 dataset for CNN training.

    Returns:
        tuple: Training, validation, and test data
    """
```

```
    # Load CIFAR-10 dataset
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    # Normalize pixel values
    x_train = x_train.astype('float32') / 255.0
    x_test = x_test.astype('float32') / 255.0

    # One-hot encode labels
    y_train = y_train.squeeze()
    y_test = y_test.squeeze()
    y_train_encoded = to_categorical(y_train, 10)
    y_test_encoded = to_categorical(y_test, 10)

    # Create validation set
    val_size = 5000
    x_val = x_train[-val_size:]
    y_val = y_train_encoded[-val_size:]
    x_train_final = x_train[:-val_size]
    y_train_final = y_train_encoded[:-val_size:]

    print(f"CIFAR-10 shapes:")
    print(f"  Training set: {x_train_final.shape}")
    print(f"  Validation set: {x_val.shape}")
    print(f"  Test set: {x_test.shape}")

    return (x_train_final, y_train_final), (x_val, y_val), (x_test, y_test_encoded), y_te

# Load datasets
mnist_data = load_and_prepare_mnist()
cifar10_data = load_and_prepare_cifar10()

# Unpack the datasets
(mnist_train, mnist_y_train), (mnist_val, mnist_y_val), (mnist_test, mnist_y_test), mnist_
(cifar_train, cifar_y_train), (cifar_val, cifar_y_val), (cifar_test, cifar_y_test), cifar_

# Define class names
mnist_class_names = [str(i) for i in range(10)]
cifar10_class_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
                      'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

# Display sample images
def display_sample_images(dataset, labels, class_names, title, num_samples=5):
    """Display sample images from a dataset."""
    plt.figure(figsize=(15, 3))
    indices = np.random.choice(range(len(dataset)), num_samples, replace=False)
```

```
    for i, idx in enumerate(indices):
        plt.subplot(1, num_samples, i+1)
        img = dataset[idx]
        if img.shape[-1] == 1:  # Grayscale
            plt.imshow(img.squeeze(), cmap='gray')
        else:  # RGB
            plt.imshow(img)

        if labels.ndim > 1:  # One-hot encoded
            label_idx = np.argmax(labels[idx])
        else:
            label_idx = labels[idx]

        plt.title(f"{class_names[label_idx]}")
        plt.axis('off')

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

# Display samples
display_sample_images(mnist_train, mnist_y_train, mnist_class_names, "MNIST Samples")
display_sample_images(cifar_train, cifar_y_train, cifar10_class_names, "CIFAR-10 Samples"
```

PART 3: BASIC CNN IMPLEMENTATION (20 minutes)

Now, let's implement a basic CNN architecture for the MNIST dataset.

python

Copy

```
def create_basic_cnn(input_shape, num_classes=10):
    """
    Create a basic CNN model for image classification.

    Args:
        input_shape: Input shape (height, width, channels)
        num_classes: Number of output classes

    Returns:
        model: Compiled Keras model
    """
    model = Sequential([
        # First convolutional block
        Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same', input_shape=inpu
        MaxPooling2D(pool_size=(2, 2)),

        # Second convolutional block
        Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),

        # Flatten and fully connected layers
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(num_classes, activation='softmax')
    ])

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

    # Print model summary
    model.summary()

    return model

def train_and_evaluate_model(model, train_data, val_data, test_data, model_name, batch_si
    """
    Train and evaluate a model, measuring performance metrics.
```

```
    Args:
        model: Compiled Keras model
        train_data: Tuple of (x_train, y_train)
        val_data: Tuple of (x_val, y_val)
        test_data: Tuple of (x_test, y_test)
        model_name: Name for the model
        batch_size: Batch size for training
        epochs: Maximum number of epochs
        patience: Early stopping patience

    Returns:
        dict: Results including metrics
    """
    x_train, y_train = train_data
    x_val, y_val = val_data
    x_test, y_test = test_data

    # Early stopping
    early_stopping = EarlyStopping(
        monitor='val_accuracy',
        patience=patience,
        restore_best_weights=True
    )

    # Measure training time
    start_time = time.time()

    # Train the model
    history = model.fit(
        x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(x_val, y_val),
        callbacks=[early_stopping],
        verbose=1
    )

    training_time = time.time() - start_time

    # Evaluate on test set
    test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)

    # Measure inference time
    start_time = time.time()
    _ = model.predict(x_test[:1000], verbose=0)
    inference_time = (time.time() - start_time) / 1000  # per sample
```

```
# Count parameters
trainable_params = np.sum([np.prod(v.get_shape()) for v in model.trainable_weights])
non_trainable_params = np.sum([np.prod(v.get_shape()) for v in model.non_trainable_we
total_params = trainable_params + non_trainable_params

# Calculate efficiency metrics
params_per_second = total_params / training_time
accuracy_per_million_params = test_accuracy * 100 / (total_params / 1e6)

# Save results
results = {
    'model_name': model_name,
    'history': history,
    'training_time': training_time,
    'test_accuracy': test_accuracy * 100,  # convert to percentage
    'test_loss': test_loss,
    'inference_time': inference_time * 1000,  # convert to milliseconds
    'total_params': total_params,
    'trainable_params': trainable_params,
    'params_per_second': params_per_second,
    'accuracy_per_million_params': accuracy_per_million_params,
    'epochs_trained': len(history.history['accuracy']),
    'batch_size': batch_size
}

print(f"\n--- {model_name} Results ---")
print(f"Test Accuracy: {results['test_accuracy']:.2f}%")
print(f"Training Time: {results['training_time']:.2f} seconds")
print(f"Inference Time: {results['inference_time']:.4f} ms")
print(f"Total Parameters: {results['total_params']:,}")
print(f"Epochs Trained: {results['epochs_trained']}")

# Plot training history
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.title(f'{model_name} - Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train')
```

```
    plt.plot(history.history['val_loss'], label='Validation')
    plt.title(f'{model_name} - Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    return results, model

# Create and train a basic CNN on MNIST
mnist_input_shape = mnist_train.shape[1:]  # (28, 28, 1)
basic_cnn_model = create_basic_cnn(mnist_input_shape)
basic_cnn_results, basic_cnn_model = train_and_evaluate_model(
    model=basic_cnn_model,
    train_data=(mnist_train, mnist_y_train),
    val_data=(mnist_val, mnist_y_val),
    test_data=(mnist_test, mnist_y_test),
    model_name="Basic CNN (MNIST)"
)

# Create and train a basic CNN on CIFAR-10
cifar_input_shape = cifar_train.shape[1:]  # (32, 32, 3)
cifar_cnn_model = create_basic_cnn(cifar_input_shape)
cifar_cnn_results, cifar_cnn_model = train_and_evaluate_model(
    model=cifar_cnn_model,
    train_data=(cifar_train, cifar_y_train),
    val_data=(cifar_val, cifar_y_val),
    test_data=(cifar_test, cifar_y_test),
    model_name="Basic CNN (CIFAR-10)"
)

# Generate confusion matrices
def plot_confusion_matrix(model, x_test, y_test_true, class_names, title):
    """
    Plot confusion matrix for model predictions.

    Args:
        model: Trained Keras model
        x_test: Test inputs
        y_test_true: True labels (not one-hot encoded)
        class_names: List of class names
        title: Plot title
    """
```

```
    # Generate predictions
    y_pred = model.predict(x_test, verbose=0)
    y_pred_classes = np.argmax(y_pred, axis=1)

    # Create confusion matrix
    cm = confusion_matrix(y_test_true, y_pred_classes)

    # Normalize confusion matrix
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    # Plot confusion matrix
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm_normalized, annot=True, fmt='.2f', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)
    plt.title(title)
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.show()

    # Find most confused pairs
    np.fill_diagonal(cm, 0)  # Ignore correct classifications
    max_confusion = np.unravel_index(np.argmax(cm), cm.shape)
    print(f"Most confused pair: {class_names[max_confusion[0]]} mistaken for {class_names

# Generate confusion matrices
plot_confusion_matrix(
    basic_cnn_model, mnist_test, mnist_y_test_raw,
    mnist_class_names, "MNIST - Basic CNN Confusion Matrix"
)

plot_confusion_matrix(
    cifar_cnn_model, cifar_test, cifar_y_test_raw,
    cifar10_class_names, "CIFAR-10 - Basic CNN Confusion Matrix"
)
```

PART 4: ARCHITECTURAL EXPLORATION (30 minutes)

Now, let's explore different CNN architectures and hyperparameters.

python

Copy

```python
def create_cnn_with_config(input_shape, config, num_classes=10):
    """
    Create a CNN model with the specified configuration.

    Args:
        input_shape: Input shape (height, width, channels)
        config: Dictionary with model configuration
        num_classes: Number of output classes

    Returns:
        model: Compiled Keras model
    """
    model = Sequential()

    # Add convolutional blocks
    for i, block in enumerate(config['conv_blocks']):
        # First layer in block (with input shape if it's the first layer)
        if i == 0:
            model.add(Conv2D(
                filters=block['filters'],
                kernel_size=block['kernel_size'],
                activation=config['activation'],
                padding=block.get('padding', 'same'),
                input_shape=input_shape
            ))
        else:
            model.add(Conv2D(
                filters=block['filters'],
                kernel_size=block['kernel_size'],
                activation=config['activation'],
                padding=block.get('padding', 'same')
            ))

        # Add batch normalization if specified
        if config.get('batch_norm', False):
            model.add(BatchNormalization())

        # Add pooling layer if specified
        if 'pool_size' in block:
            if block.get('pool_type', 'max') == 'max':
                model.add(MaxPooling2D(pool_size=block['pool_size']))
            else:
                model.add(AveragePooling2D(pool_size=block['pool_size']))
```

```python
    # Flatten and add dense layers
    model.add(Flatten())

    for units in config['dense_units']:
        model.add(Dense(units, activation=config['activation']))
        if config.get('dropout_rate', 0) > 0:
            model.add(Dropout(config['dropout_rate']))

    # Output layer
    model.add(Dense(num_classes, activation='softmax'))

    # Compile model
    model.compile(
        optimizer=config.get('optimizer', 'adam'),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    # Print model summary
    model.summary()

    return model

# Define different CNN configurations to explore
cnn_configurations = [
    {
        "name": "ShallowCNN",
        "config": {
            "conv_blocks": [
                {"filters": 16, "kernel_size": (3, 3), "pool_size": (2, 2)},
                {"filters": 32, "kernel_size": (3, 3), "pool_size": (2, 2)}
            ],
            "activation": "relu",
            "dense_units": [64],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    },
    {
        "name": "DeepCNN",
        "config": {
            "conv_blocks": [
                {"filters": 32, "kernel_size": (3, 3), "pool_size": (2, 2)},
                {"filters": 64, "kernel_size": (3, 3), "pool_size": (2, 2)},
                {"filters": 128, "kernel_size": (3, 3), "pool_size": (2, 2)}
```

```python
            ],
            "activation": "relu",
            "dense_units": [128],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    },
    {
        "name": "WideCNN",
        "config": {
            "conv_blocks": [
                {"filters": 64, "kernel_size": (3, 3), "pool_size": (2, 2)},
                {"filters": 128, "kernel_size": (3, 3), "pool_size": (2, 2)}
            ],
            "activation": "relu",
            "dense_units": [256],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    },
    {
        "name": "TinyCNN",
        "config": {
            "conv_blocks": [
                {"filters": 8, "kernel_size": (3, 3), "pool_size": (2, 2)},
                {"filters": 16, "kernel_size": (3, 3), "pool_size": (2, 2)}
            ],
            "activation": "relu",
            "dense_units": [32],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    }
]

# Train all configurations on MNIST
mnist_architecture_results = []
mnist_architecture_models = []

for config_info in cnn_configurations:
    name = config_info["name"]
    config = config_info["config"]

    print(f"\nTraining {name} on MNIST...")
    model = create_cnn_with_config(mnist_input_shape, config)
```

```python
    results, trained_model = train_and_evaluate_model(
        model=model,
        train_data=(mnist_train, mnist_y_train),
        val_data=(mnist_val, mnist_y_val),
        test_data=(mnist_test, mnist_y_test),
        model_name=f"{name} (MNIST)"
    )

    mnist_architecture_results.append(results)
    mnist_architecture_models.append(trained_model)

# Experiment with different filter sizes on MNIST
filter_size_configs = [
    {
        "name": "SmallFilters",
        "config": {
            "conv_blocks": [
                {"filters": 32, "kernel_size": (2, 2), "pool_size": (2, 2)},
                {"filters": 64, "kernel_size": (2, 2), "pool_size": (2, 2)}
            ],
            "activation": "relu",
            "dense_units": [128],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    },
    {
        "name": "LargeFilters",
        "config": {
            "conv_blocks": [
                {"filters": 32, "kernel_size": (5, 5), "pool_size": (2, 2)},
                {"filters": 64, "kernel_size": (5, 5), "pool_size": (2, 2)}
            ],
            "activation": "relu",
            "dense_units": [128],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    },
    {
        "name": "MixedFilters",
        "config": {
            "conv_blocks": [
                {"filters": 32, "kernel_size": (3, 3), "pool_size": (2, 2)},
                {"filters": 64, "kernel_size": (5, 5), "pool_size": (2, 2)}
            ],
```

```
            "activation": "relu",
            "dense_units": [128],
            "dropout_rate": 0.3,
            "optimizer": "adam"
        }
    }
]

# Train filter size configurations on MNIST
for config_info in filter_size_configs:
    name = config_info["name"]
    config = config_info["config"]

    print(f"\nTraining {name} on MNIST...")
    model = create_cnn_with_config(mnist_input_shape, config)

    results, trained_model = train_and_evaluate_model(
        model=model,
        train_data=(mnist_train, mnist_y_train),
        val_data=(mnist_val, mnist_y_val),
        test_data=(mnist_test, mnist_y_test),
        model_name=f"{name} (MNIST)"
    )

    mnist_architecture_results.append(results)
    mnist_architecture_models.append(trained_model)

# Compare results of different architectures
architecture_df = pd.DataFrame([
    {
        'Model': result['model_name'],
        'Accuracy (%)': result['test_accuracy'],
        'Training Time (s)': result['training_time'],
        'Inference Time (ms)': result['inference_time'],
        'Parameters': result['total_params'],
        'Params/Second': result['params_per_second'],
        'Accuracy/Million Params': result['accuracy_per_million_params']
    }
    for result in mnist_architecture_results
])

print("\nMNIST Architecture Comparison:")
print(architecture_df)

# Visualize the results
plt.figure(figsize=(12, 10))
```

```
# Accuracy vs Parameters
plt.subplot(2, 2, 1)
plt.scatter(architecture_df['Parameters'], architecture_df['Accuracy (%)'], s=100)
for i, row in architecture_df.iterrows():
    plt.annotate(row['Model'].replace(' (MNIST)', ''),
                 (row['Parameters'], row['Accuracy (%)']),
                 xytext=(5, 5), textcoords='offset points')
plt.title('Accuracy vs Parameters')
plt.xlabel('Number of Parameters')
plt.ylabel('Accuracy (%)')
plt.grid(True)

# Inference Time vs Parameters
plt.subplot(2, 2, 2)
plt.scatter(architecture_df['Parameters'], architecture_df['Inference Time (ms)'], s=100)
for i, row in architecture_df.iterrows():
    plt.annotate(row['Model'].replace(' (MNIST)', ''),
                 (row['Parameters'], row['Inference Time (ms)']),
                 xytext=(5, 5), textcoords='offset points')
plt.title('Inference Time vs Parameters')
plt.xlabel('Number of Parameters')
plt.ylabel('Inference Time (ms)')
plt.grid(True)

# Accuracy vs Training Time
plt.subplot(2, 2, 3)
plt.scatter(architecture_df['Training Time (s)'], architecture_df['Accuracy (%)'], s=100)
for i, row in architecture_df.iterrows():
    plt.annotate(row['Model'].replace(' (MNIST)', ''),
                 (row['Training Time (s)'], row['Accuracy (%)']),
                 xytext=(5, 5), textcoords='offset points')
plt.title('Accuracy vs Training Time')
plt.xlabel('Training Time (s)')
plt.ylabel('Accuracy (%)')
plt.grid(True)

# Accuracy/Million Parameters
plt.subplot(2, 2, 4)
plt.bar(architecture_df['Model'].str.replace(' (MNIST)', ''), architecture_df['Accuracy/M
plt.title('Accuracy per Million Parameters')
plt.xticks(rotation=45, ha='right')
plt.ylabel('Accuracy/Million Params')
plt.grid(axis='y')
```

```
plt.tight_layout()
plt.show()
```

PART 5: FEATURE VISUALIZATION (20 minutes)

Let's visualize what our CNN models are learning in their convolutional layers.

python

Copy

```
def visualize_filters(model, layer_name=None):
    """
    Visualize the filters/kernels in a convolutional layer.

    Args:
        model: Trained Keras model
        layer_name: Name of layer to visualize (if None, use first Conv2D layer)
    """
    # Get the layer of interest
    if layer_name is None:
        # Find the first convolutional layer
        for layer in model.layers:
            if isinstance(layer, tf.keras.layers.Conv2D):
                layer_name = layer.name
                break

    # Get the weights
    for layer in model.layers:
        if layer.name == layer_name:
            weights, biases = layer.get_weights()
            break

    # Normalize weights for better visualization
    weights_min, weights_max = np.min(weights), np.max(weights)
    weights = (weights - weights_min) / (weights_max - weights_min)

    # Number of filters and their dimensions
    n_filters, _, filter_height, filter_width = weights.shape

    # Compute grid size
    grid_size = int(np.ceil(np.sqrt(n_filters)))

    # Create figure with subplot grid
    plt.figure(figsize=(15, 15))

    # Plot each filter
    for i in range(n_filters):
        if i < grid_size * grid_size:  # Ensure we don't exceed the grid
            plt.subplot(grid_size, grid_size, i+1)
            # Depending on the input shape, filters might have different channel dimensio
            if weights.shape[1] == 1:  # Grayscale
                plt.imshow(weights[i, 0], cmap='viridis')
            else:  # RGB - take mean across channels for visualization
```

```python
            plt.imshow(np.mean(weights[i], axis=0), cmap='viridis')
        plt.axis('off')

    plt.suptitle(f'Filters in layer: {layer_name}')
    plt.tight_layout()
    plt.subplots_adjust(top=0.95)
    plt.show()

def visualize_feature_maps(model, image, layer_names=None):
    """
    Visualize feature maps from convolutional layers for a given input image.

    Args:
        model: Trained Keras model
        image: Input image to visualize feature maps for
        layer_names: List of layer names to visualize (if None, use all Conv2D layers)
    """
    # If no layer names provided, find all convolutional layers
    if layer_names is None:
        layer_names = []
        for layer in model.layers:
            if isinstance(layer, tf.keras.layers.Conv2D):
                layer_names.append(layer.name)

    # Create intermediate models for each layer
    feature_maps = []
    for layer_name in layer_names:
        intermediate_model = Model(inputs=model.input,
                                   outputs=model.get_layer(layer_name).output)
        feature_maps.append((layer_name, intermediate_model.predict(np.expand_dims(image,

    # Plot feature maps
    for layer_name, feature_map in feature_maps:
        # Get the feature map from the batch
        feature_map = feature_map[0]

        # Number of features in this layer
        n_features = feature_map.shape[-1]

        # Compute grid size
        grid_size = int(np.ceil(np.sqrt(n_features)))

        # Limit to max 64 feature maps for visibility
        display_features = min(n_features, 64)
        display_grid_size = int(np.ceil(np.sqrt(display_features)))
```

```
        # Create figure with subplot grid
        plt.figure(figsize=(15, 15))

        # Plot each feature map
        for i in range(display_features):
            if i < display_grid_size * display_grid_size:  # Ensure we don't exceed the g
                plt.subplot(display_grid_size, display_grid_size, i+1)
                plt.imshow(feature_map[:, :, i], cmap='viridis')
                plt.axis('off')

        plt.suptitle(f'Feature Maps in layer: {layer_name} (showing {display_features} of
        plt.tight_layout()
        plt.subplots_adjust(top=0.95)
        plt.show()

# Visualize filters from our trained CNN models
print("Visualizing filters from Basic CNN (MNIST):")
visualize_filters(basic_cnn_model)

# Visualize feature maps for a sample image
sample_idx = np.random.randint(0, len(mnist_test))
sample_image = mnist_test[sample_idx]
sample_label = np.argmax(mnist_y_test[sample_idx])

plt.figure(figsize=(4, 4))
plt.imshow(sample_image.squeeze(), cmap='gray')
plt.title(f"Sample Image (Digit: {sample_label})")
plt.axis('off')
plt.show()

print("Visualizing feature maps for the sample image:")
visualize_feature_maps(basic_cnn_model, sample_image)

# Visualize activations in different CNN architectures
best_model_idx = np.argmax([r['test_accuracy'] for r in mnist_architecture_results])
best_arch_model = mnist_architecture_models[best_model_idx]
best_arch_name = mnist_architecture_results[best_model_idx]['model_name']

print(f"Visualizing feature maps for the best architecture ({best_arch_name}):")
visualize_feature_maps(best_arch_model, sample_image)
```

PART 6: TRANSFER LEARNING (20 minutes)

Let's explore transfer learning by adapting pre-trained models to our datasets.

python

Copy

```python
def create_transfer_learning_model(base_model, input_shape, num_classes=10, freeze_base=T
    """
    Create a transfer learning model using a pre-trained base model.

    Args:
        base_model: Pre-trained model to use as base
        input_shape: Input shape for the model
        num_classes: Number of output classes
        freeze_base: Whether to freeze the base model weights

    Returns:
        model: Compiled Keras model
    """
    # Freeze base model if requested
    base_model.trainable = not freeze_base

    # Create input layer with required input shape
    inputs = Input(shape=input_shape)

    # Resize input if needed (e.g., for MNIST)
    # Most pre-trained models expect RGB images of specific sizes
    resized_inputs = inputs
    if input_shape != base_model.input_shape[1:]:
        # Add preprocessing to match expected input
        if input_shape[-1] == 1:  # Grayscale to RGB
            # Repeat the single channel three times
            resized_inputs = tf.keras.layers.Lambda(
                lambda x: tf.repeat(x, 3, axis=-1)
            )(inputs)

        # Resize to expected dimensions
        resized_inputs = tf.keras.layers.Lambda(
            lambda x: tf.image.resize(x, base_model.input_shape[1:3])
        )(resized_inputs)

    # Pass through base model
    x = base_model(resized_inputs, training=False)

    # Add classification head
    x = GlobalAveragePooling2D()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.3)(x)
    outputs = Dense(num_classes, activation='softmax')(x)
```

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

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

    # Print model summary
    model.summary()

    return model

# Load pre-trained MobileNetV2 model (without top layers)
mobilenet_base = MobileNetV2(
    weights='imagenet',
    include_top=False,
    input_shape=(224, 224, 3)
)

# Create transfer learning model for CIFAR-10
print("\nCreating transfer learning model for CIFAR-10...")
mobilenet_cifar = create_transfer_learning_model(
    base_model=mobilenet_base,
    input_shape=cifar_input_shape,
    num_classes=10,
    freeze_base=True
)

# Train the transfer learning model on CIFAR-10
mobilenet_results, mobilenet_model = train_and_evaluate_model(
    model=mobilenet_cifar,
    train_data=(cifar_train, cifar_y_train),
    val_data=(cifar_val, cifar_y_val),
    test_data=(cifar_test, cifar_y_test),
    model_name="MobileNetV2 Transfer (CIFAR-10)",
    epochs=10  # Faster convergence with transfer learning
)

# Let's try some fine-tuning by unfreezing the top layers
def create_fine_tuned_model(base_model, input_shape, num_classes=10, unfreeze_layers=5):
    """
    Create a fine-tuned model by unfreezing some layers of a pre-trained model.
```

```python
    Args:
        base_model: Pre-trained model to use as base
        input_shape: Input shape for the model
        num_classes: Number of output classes
        unfreeze_layers: Number of top layers to unfreeze

    Returns:
        model: Compiled Keras model
    """
    # Freeze all layers initially
    base_model.trainable = True
    for layer in base_model.layers:
        layer.trainable = False

    # Unfreeze the last few layers
    for layer in base_model.layers[-unfreeze_layers:]:
        layer.trainable = True

    # Create input layer with required input shape
    inputs = Input(shape=input_shape)

    # Resize input if needed
    resized_inputs = inputs
    if input_shape != base_model.input_shape[1:]:
        # Add preprocessing to match expected input
        if input_shape[-1] == 1:  # Grayscale to RGB
            resized_inputs = tf.keras.layers.Lambda(
                lambda x: tf.repeat(x, 3, axis=-1)
            )(inputs)

        # Resize to expected dimensions
        resized_inputs = tf.keras.layers.Lambda(
            lambda x: tf.image.resize(x, base_model.input_shape[1:3])
        )(resized_inputs)

    # Pass through base model
    x = base_model(resized_inputs, training=True)

    # Add classification head
    x = GlobalAveragePooling2D()(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.3)(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    # Create model
```

```
    model = Model(inputs=inputs, outputs=outputs)

    # Compile model with lower learning rate for fine-tuning
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    # Count trainable vs non-trainable parameters
    trainable_params = np.sum([np.prod(v.get_shape()) for v in model.trainable_weights])
    non_trainable_params = np.sum([np.prod(v.get_shape()) for v in model.non_trainable_we
    print(f"Trainable parameters: {trainable_params:,}")
    print(f"Non-trainable parameters: {non_trainable_params:,}")

    return model

# Create fine-tuned model for CIFAR-10
print("\nCreating fine-tuned model for CIFAR-10...")
fine_tuned_cifar = create_fine_tuned_model(
    base_model=mobilenet_base,
    input_shape=cifar_input_shape,
    num_classes=10,
    unfreeze_layers=10
)

# Train the fine-tuned model on CIFAR-10
fine_tuned_results, fine_tuned_model = train_and_evaluate_model(
    model=fine_tuned_cifar,
    train_data=(cifar_train, cifar_y_train),
    val_data=(cifar_val, cifar_y_val),
    test_data=(cifar_test, cifar_y_test),
    model_name="MobileNetV2 Fine-tuned (CIFAR-10)",
    epochs=5,  # Fewer epochs to prevent overfitting
    batch_size=32  # Smaller batch size for fine-tuning
)

# Compare the transfer learning results
transfer_df = pd.DataFrame([
    {
        'Model': result['model_name'],
        'Accuracy (%)': result['test_accuracy'],
        'Training Time (s)': result['training_time'],
        'Inference Time (ms)': result['inference_time'],
        'Parameters': result['total_params'],
        'Trainable Parameters': result['trainable_params'],
```

```
        'Accuracy/Million Params': result['accuracy_per_million_params']
    }
    for result in [cifar_cnn_results, mobilenet_results, fine_tuned_results]
])

print("\nTransfer Learning Comparison:")
print(transfer_df)

# Visualize predictions from transfer learning model
def display_predictions(model, images, true_labels, class_names, title, num_samples=5):
    """Display sample images with predictions."""
    indices = np.random.choice(range(len(images)), num_samples, replace=False)

    plt.figure(figsize=(15, 3))
    for i, idx in enumerate(indices):
        plt.subplot(1, num_samples, i+1)
        plt.imshow(images[idx])

        # Get prediction
        pred = model.predict(np.expand_dims(images[idx], axis=0), verbose=0)
        pred_class = np.argmax(pred)
        true_class = true_labels[idx]

        # Color based on correct/incorrect
        color = 'green' if pred_class == true_class else 'red'

        plt.title(f"True: {class_names[true_class]}\nPred: {class_names[pred_class]}", co
        plt.axis('off')

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

# Display predictions from the fine-tuned model
display_predictions(
    fine_tuned_model,
    cifar_test,
    cifar_y_test_raw,
    cifar10_class_names,
    "Fine-tuned MobileNetV2 Predictions"
)
```

PART 7: PERFORMANCE ANALYSIS (10 minutes)

Let's compare the performance of different models across our experiments.

python

Copy

```
# Combine all results (CNN architectures and transfer learning)
all_results = mnist_architecture_results + [basic_cnn_results, cifar_cnn_results, mobilene

# Create comprehensive results table
all_df = pd.DataFrame([
    {
        'Model': result['model_name'],
        'Dataset': 'MNIST' if 'MNIST' in result['model_name'] else 'CIFAR-10',
        'Accuracy (%)': result['test_accuracy'],
        'Training Time (s)': result['training_time'],
        'Inference Time (ms)': result['inference_time'],
        'Parameters': result['total_params'],
        'Trainable Params': result.get('trainable_params', result['total_params']),
        'Params/Second': result['params_per_second'],
        'Accuracy/Million Params': result['accuracy_per_million_params']
    }
    for result in all_results
])

print("\nAll Models Comparison:")
print(all_df)

# Compare CNN vs FCNN from Lab 2 (we'll simulate this here)
fcnn_results = {
    'model_name': 'Best FCNN (MNIST)',
    'test_accuracy': 98.5,
    'training_time': 45.0,
    'inference_time': 0.06,
    'total_params': 270000,
    'params_per_second': 6000,
    'accuracy_per_million_params': 365.0
}

# Find best CNN for MNIST
best_mnist_cnn = all_df[all_df['Dataset'] == 'MNIST'].sort_values('Accuracy (%)', ascendi

# Compare CNN vs FCNN
comparison_df = pd.DataFrame([
    {
        'Model': fcnn_results['model_name'],
        'Accuracy (%)': fcnn_results['test_accuracy'],
        'Training Time (s)': fcnn_results['training_time'],
        'Inference Time (ms)': fcnn_results['inference_time'],
```

```
        'Parameters': fcnn_results['total_params'],
        'Accuracy/Million Params': fcnn_results['accuracy_per_million_params']
    },
    {
        'Model': best_mnist_cnn['Model'],
        'Accuracy (%)': best_mnist_cnn['Accuracy (%)'],
        'Training Time (s)': best_mnist_cnn['Training Time (s)'],
        'Inference Time (ms)': best_mnist_cnn['Inference Time (ms)'],
        'Parameters': best_mnist_cnn['Parameters'],
        'Accuracy/Million Params': best_mnist_cnn['Accuracy/Million Params']
    }
])

print("\nCNN vs FCNN Comparison:")
print(comparison_df)

# Visualize the comparison
plt.figure(figsize=(12, 8))

# Normalized metrics for better comparison (relative to each other)
metrics = ['Accuracy (%)', 'Training Time (s)', 'Inference Time (ms)', 'Parameters', 'Accu
models = comparison_df['Model'].tolist()

# Normalize values
normalized_df = comparison_df.copy()
for metric in metrics:
    if metric in ['Accuracy (%)', 'Accuracy/Million Params']:  # Higher is better
        normalized_df[metric] = comparison_df[metric] / comparison_df[metric].max()
    else:  # Lower is better
        normalized_df[metric] = comparison_df[metric].min() / comparison_df[metric]

# Plot normalized metrics
bar_width = 0.35
index = np.arange(len(metrics))

plt.bar(index, normalized_df.iloc[0][metrics], bar_width, label=models[0])
plt.bar(index + bar_width, normalized_df.iloc[1][metrics], bar_width, label=models[1])

plt.xlabel('Metric')
plt.ylabel('Normalized Score (higher is better)')
plt.title('CNN vs FCNN Comparison')
plt.xticks(index + bar_width / 2, metrics, rotation=45, ha='right')
plt.legend()
plt.tight_layout()
plt.show()
```

```
# Identify the best model for different criteria
best_accuracy = all_df.loc[all_df['Accuracy (%)'].idxmax()]
print(f"\nBest Model for Accuracy: {best_accuracy['Model']} ({best_accuracy['Accuracy (%)

fastest_inference = all_df.loc[all_df['Inference Time (ms)'].idxmin()]
print(f"Best Model for Inference Speed: {fastest_inference['Model']} ({fastest_inference[

most_efficient = all_df.loc[all_df['Accuracy/Million Params'].idxmax()]
print(f"Most Parameter-Efficient Model: {most_efficient['Model']} ({most_efficient['Accur
```

CONCLUSION

In this lab, you learned about Convolutional Neural Networks (CNNs) and their applications in image classification. You implemented various CNN architectures, visualized learned features, experimented with transfer learning, and analyzed the computational efficiency of different models.

Key takeaways:

1. CNNs are highly effective for image processing tasks, outperforming FCNNs with fewer parameters
2. Architectural choices (depth, width, filter sizes) significantly impact model performance and efficiency
3. Transfer learning allows leveraging pre-trained models for superior results with less training
4. Visualizing feature maps helps understand what patterns the network is learning
5. Hardware efficiency metrics are crucial for deployment considerations

---

ADDITIONAL CHALLENGES (Optional)

If you complete the lab early, try these extensions:

1. Implement data augmentation (rotation, flipping, etc.) to improve model performance
2. Experiment with different pooling strategies (max vs. average)
3. Try implementing a more complex CNN architecture like ResNet or Inception
4. Apply your CNN models to a different dataset (e.g., CIFAR-100, STL-10)
5. Implement Grad-CAM for visualizing which parts of the image are important for classification
6. Explore model quantization to reduce model size and improve inference speed

---

REFERENCES

1. Convolutional Neural Networks: https://www.tensorflow.org/tutorials/images/cnn

2. Transfer Learning: https://www.tensorflow.org/tutorials/images/transfer_learning
3. Visualizing CNN Features: https://distill.pub/2017/feature-visualization/
4. CNN Architectures: https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5
5. CIFAR-10 Dataset: https://www.cs.toronto.edu/~kriz/cifar.html