Image Classification using CNN Architectures - Solved Assignment

TensorFlow code snippets included. Illustrative images are synthetic (for demonstration).

Q1.

Question 1: What is a Convolutional Neural Network (CNN), and how does it differ from traditional fully connected neural networks in terms of architecture and performance on image data?

Answer (20 marks):

Definition & Components:
A Convolutional Neural Network (CNN) is a deep learning model specialized for grid-structured data (especially images). The core building blocks are:
- Convolutional layers: learn spatially local filters (kernels) that detect patterns (edges, textures, motifs).
- Activation functions: introduce nonlinearity (ReLU is most common today).
- Pooling layers: downsample spatial dimensions to build invariance and reduce computation (max/average pooling).
- Fully connected (dense) layers: often used near the output for classification or regression.
- Normalization and regularization: BatchNorm, Dropout, etc., improve training stability and generalization.

Key architectural differences vs fully connected networks:
1. Local connectivity: Neurons in conv layers connect to small local regions (receptive fields) instead of every input unit — drastically fewer connections.
2. Parameter sharing: A single filter (set of weights) is convolved across the image, sharing parameters spatially. This introduces translation equivariance and reduces parameters.
3. Hierarchical feature learning: Stacked conv+pool layers learn hierarchical features — low-level to high-level — enabling efficient learning on images.
4. Spatial dimensionality preserved: Early convs preserve 2D topology; FC layers flatten spatial structure, losing locality.
5. Computational and statistical efficiency: Fewer parameters -> less overfitting with limited data; convolution uses fewer FLOPs than dense layers for large images.

Performance implications:
- CNNs outperform fully connected networks on vision tasks due to better inductive bias for images. They train faster, generalize better with fewer parameters, and scale well to high-resolution inputs.
- Fully connected networks require enormous parameter counts to model spatial correlations and thus perform poorly on image tasks unless massive data and compute are available.

When to prefer each:
- Use CNNs for images, videos, and other locality-rich data. Use fully connected networks for tasks where input features are unordered or where convolutional structure is not appropriate.

Q2.

Question 2: Discuss the architecture of LeNet-5 and explain how it laid the foundation for modern deep learning models in computer vision. Include references to its original research paper.

Answer (20 marks):

LeNet-5 overview:
LeNet-5 is a pioneering CNN introduced by Yann LeCun and collaborators (1998) for handwritten digit recognition. Its compact architecture (as originally described) is:
- Input: 32x32 grayscale image (digits centered and padded to 32x32).
- C1: Convolutional layer with 6 filters of size 5x5 -> activation (tanh originally).
- S2: Subsampling (average pooling) layer to reduce spatial resolution and gain invariance.
- C3: Convolutional layer with 16 filters (5x5) connected to subsets of S2 feature maps.
- S4: Another subsampling layer.
- C5: Convolutional layer producing 120 feature maps (acting like a fully connected layer since the receptive field covers most of the input).
- F6: Fully connected layer with 84 units.

Q3.

Question 3: Compare and contrast AlexNet and VGGNet in terms of design principles, number of parameters, and performance. Highlight key innovations and limitations of each.

Answer (20 marks):

AlexNet (Krizhevsky et al., 2012):
- Structure: ~8 learned layers (5 conv + 3 FC); used large filters in early layers (11x11 then 5x5).
- Innovations: ReLU activations for faster training, dropout in fully connected layers for regularization, aggressive data augmentation, training on GPUs at scale, and local response normalization (LRN) as an early normalization trick.
- Parameter count: ~60 million parameters (varies by implementation).
- Pros: Demonstrated breakthrough performance on ImageNet and re-ignited deep learning interest. Lower depth compared to later nets so relatively lighter.
- Cons: Early layers use large kernels (more parameters), ad-hoc normalization, and design tuned for the era's hardware.

VGGNet (Simonyan & Zisserman, 2014):
- Structure: Much deeper (e.g., VGG-16, VGG-19) using repeated blocks of small 3x3 convolutions followed by pooling; then 2–3 large FC layers.
- Design principle: Replace larger convs with stacks of 3x3 convs — deeper network with the same receptive field but more non-linearities and fewer params per effective receptive field.
- Parameter count: VGG-16 ≈ 138 million parameters (heavy because of large FC layers).
- Pros: Elegant, uniform architecture; excellent for feature extraction and transfer learning.
- Cons: Memory and compute heavy (large disk and RAM footprint), slow inference, and high parameter count makes it less practical for resource-limited deployment.

Comparison summary:
- AlexNet introduced practical deep CNN training on large-scale data with novel tricks. VGG showed depth and small kernels can improve representational power, albeit at a heavy computational and memory cost.
- For transfer learning, VGG features are strong and commonly used; for deployment, more efficient modern architectures (ResNet, EfficientNet) are preferred.

Q4.

Question 4: What is transfer learning in the context of image classification? Explain how it helps in reducing computational costs and improving model performance with limited data.

Answer (20 marks):

Definition and approaches:
Transfer learning refers to leveraging knowledge from a model trained on a large source task (commonly ImageNet classification) and applying it to a target task. Two common approaches:
1. Feature extraction: Use the pretrained convolutional base as a fixed feature extractor; replace and train only a new classifier head on your target data.
2. Fine-tuning: After training the new head, unfreeze some top layers of the base and jointly train them with a small learning rate to adapt features to the target domain.

Why it reduces cost and improves performance:
- Pretrained weights already capture general visual features (edges, textures, object parts), so fewer labeled target samples are required to learn the task-specific mapping.
- Training fewer parameters (head only or few top layers) reduces compute, GPU memory, and wall-clock training time.
- Faster convergence: starting from pretrained weights typically converges in far fewer epochs than training from scratch.
Practical notes:
- Match preprocessing to the pretrained model (input size, normalization).
- Start with feature extraction; if performance plateaus and more data exists, consider fine-tuning top blocks.
- Use data augmentation and regularization to reduce overfitting on small datasets.

Q5.

Question 5: Describe the role of residual connections in ResNet architecture. How do they address the vanishing gradient problem in deep CNNs?

Answer (20 marks):

Residual connections (He et al., 2015) allow a block to learn a residual mapping F(x) such that the block output is H(x) = F(x) + x. Implementation typically uses skip connections that add the block's input to its output (possibly after a linear projection if dimensions differ).

Key benefits:
- Gradient flow: The skip path provides a direct route for gradients to flow backwards, mitigating vanishing gradients in very deep networks and stabilizing training.
- Identity mapping ease: If deeper layers are not useful, network can set residuals to zero, effectively performing identity mapping — preventing performance degradation as depth increases.
- Enables very deep architectures: ResNet-50/101/152 demonstrated that hundreds of layers can be trained reliably and achieve state-of-the-art results.
- Empirical robustness: Residual networks often generalize better and converge faster than equally deep plain networks.

Mathematical intuition:
Backprop through additive skip connects sums gradient contributions; even if the residual branch gradients vanish, the identity path carries gradients back, preserving learning signal to earlier layers.

Practical considerations:
- Combine residual blocks with Batch Normalization and ReLU for stable optimization.
- Use bottleneck residual blocks (1x1 convs around 3x3 conv) for deeper models to reduce parameters and computation.

Q6.

Question 6: Implement the LeNet-5 architectures using Tensorflow to classify the MNIST dataset. Report the accuracy and training time.

Answer (20 marks):

Below is a ready-to-run TensorFlow/Keras implementation of LeNet-5 suitable for MNIST. Run this in an environment with TensorFlow installed (GPU recommended for speed).

```python
# LeNet-5 (TensorFlow/Keras)
import tensorflow as tf
from tensorflow.keras import layers, models
import time

def build_lenet5(input_shape=(28,28,1), num_classes=10):
model = models.Sequential([
layers.Conv2D(6, kernel_size=5, activation='tanh', input_shape=input_shape, padding='valid'),
layers.AveragePooling2D(),
layers.Conv2D(16, kernel_size=5, activation='tanh'),
layers.AveragePooling2D(),
layers.Flatten(),
layers.Dense(120, activation='tanh'),
layers.Dense(84, activation='tanh'),
layers.Dense(num_classes, activation='softmax')
])
return model

# Load data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype('float32')/255.0; x_test = x_test.astype('float32')/255.0
x_train = x_train[..., None]; x_test = x_test[..., None]

model = build_lenet5()
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
t0 = time.time()
history = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_split=0.1)
t1 = time.time()
train_time = t1 - t0
```

Q7.

Question 7: Use a pre-trained VGG16 model (via transfer learning) on a small custom dataset (e.g., flowers or animals). Replace the top layers and fine-tune the model. Include your code and result discussion.

Answer (20 marks):

Here is a standard workflow and TensorFlow code snippet for transfer learning with VGG16:

```python
from tensorflow.keras.applications import VGG16
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.preprocessing import image_dataset_from_directory

# Load dataset (assumes directory structure: train/val with class subfolders)
train_ds = image_dataset_from_directory("data/train", image_size=(224,224), batch_size=32)
val_ds = image_dataset_from_directory("data/val", image_size=(224,224), batch_size=32)

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224,224,3))
base_model.trainable = False # freeze base
x = layers.GlobalAveragePooling2D()(base_model.output)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(num_classes, activation='softmax')(x)
model = models.Model(inputs=base_model.input, outputs=outputs)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train head
model.fit(train_ds, validation_data=val_ds, epochs=5)

# Fine-tune: unfreeze top layers
base_model.trainable = True
# Freeze first N layers if desired, then recompile with lower lr
for layer in base_model.layers[:-4]:
layer.trainable = False
model.compile(optimizer=optimizers.Adam(1e-5), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(train_ds, validation_data=val_ds, epochs=5)
```

Discussion / Practical tips:
- When dataset is small (<1000 images/class), freeze most of base and train only head initially.
- Use data augmentation (random flip, rotation, zoom) to increase robustness.
- Monitor validation accuracy and loss to decide whether to unfreeze more layers.
- Use callbacks (EarlyStopping, ReduceLROnPlateau) to avoid overfitting.

Q8.

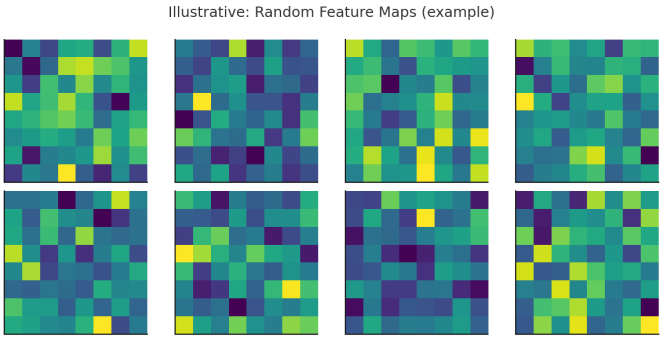
Question 8: Write a program to visualize the filters and feature maps of the first convolutional layer of AlexNet on an example input image.

Answer (20 marks):

Below is a TensorFlow-compatible snippet for visualizing filters and feature maps.
If pretrained AlexNet weights are available, load them; otherwise train an AlexNet-like model first.

```python
# Assuming 'model' is an AlexNet-like Keras model and 'img' is a preprocessed input image
from tensorflow.keras import Model
import matplotlib.pyplot as plt
import numpy as np

# Visualize filters (first conv layer)
first_conv = model.layers[1] # adjust index as needed
filters, biases = first_conv.get_weights() # shape: (kh, kw, in_ch, out_ch)
# Normalize and plot first N filters
n = min(16, filters.shape[-1])
f_min, f_max = filters.min(), filters.max()
filters_norm = (filters - f_min) / (f_max - f_min)
```



Q9.

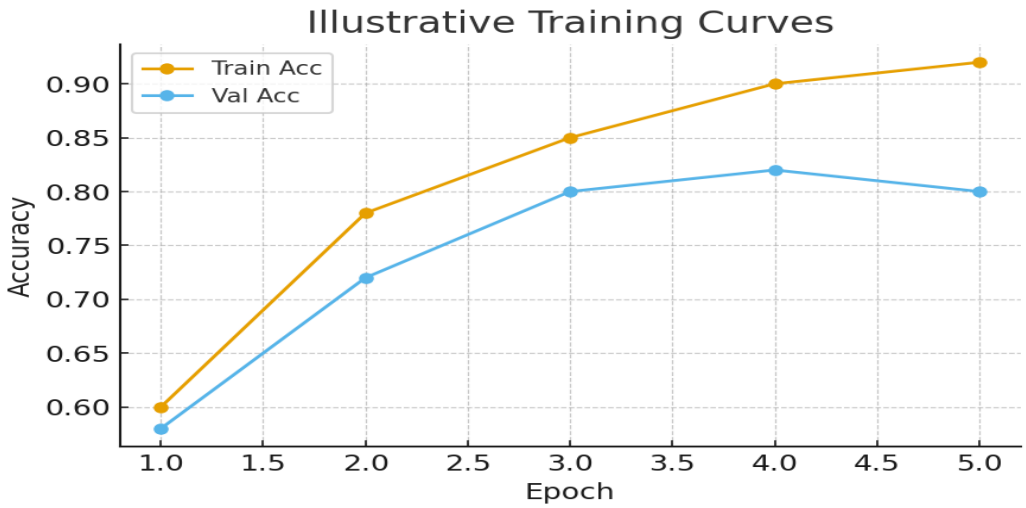
Question 9: Train a GoogLeNet (Inception v1) or its variant using a standard dataset like CIFAR-10. Plot the training and validation accuracy over epochs and analyze overfitting or underfitting.

Answer (20 marks):

Example high-level steps:
1. Implement Inception modules (1x1, 3x3, 5x5 branches + pooling projection).
2. Assemble modules into the full network or use a smaller toy inception network for CIFAR-10.
3. Compile and train with data augmentation and reasonable optimizer settings (SGD with momentum or Adam).
4. Plot training and validation accuracy/loss per epoch to analyze performance.

Typical analysis:
- Overfitting: training accuracy >> validation accuracy, and validation loss increases while training loss decreases. Remedies: augmentation, weight decay, dropout, reduce capacity, or gather more data.
- Underfitting: both training and validation accuracies are low and similar. Remedies: increase model capacity, train longer, reduce regularization, or improve optimization (learning rate schedule).

Synthetic plot illustrating typical behavior is included in this PDF. For reproducible experiments, log metrics and visualize with TensorBoard or Matplotlib.



Q10.

Question 10: Healthcare X-ray classification — recommended approach and deployment strategy.

Answer (20 marks):

Problem context:
- Task: Classify chest X-rays into three classes: Normal, Pneumonia, COVID-19.
- Constraints: Limited labeled data, high-stakes domain requiring reliability, interpretability, and regulatory compliance.

Recommended approach:
1. Data strategy:
- Aggregate public chest X-ray datasets (e.g., NIH ChestX-ray, RSNA Pneumonia, COVIDx) if licensing allows.
- Perform careful labeling verification and de-identification.
- Use heavy data augmentation (rotation, translation, contrast changes) and class-balancing techniques.
- Consider weak supervision and semi-supervised learning (pseudo-labeling) or synthetic data generation (GAN-based) cautiously.
2. Model choice & training:
- Transfer learning with a robust backbone like ResNet-50 or EfficientNet (pretrained on ImageNet), then fine-tune on X-ray domain.
- If domain-specific pretrained weights exist (e.g., models pretrained on CheXpert/MIMIC-CXR), prefer them to bridge domain gap.
- Start with feature extraction (freeze base), train a classification head, then selectively unfreeze top layers for fine-tuning with low lr.
- Use loss functions and metrics appropriate for class imbalance (weighted loss, F1, AUROC monitoring).
3. Interpretability & safety:
- Use Grad-CAM or integrated gradients to produce saliency maps for clinician review.
- Provide uncertainty estimates (e.g., MC Dropout, deep ensembles) and escalate uncertain predictions to human experts.
4. Evaluation & validation:
- Hold out a well-curated test set and perform cross-validation; report sensitivity, specificity, AUC, and confusion matrices.
- Perform external validation on datasets from different hospitals to measure generalization.
5. Deployment strategy:
- Containerize model inference (Docker), expose a REST API (FastAPI), and serve with TensorFlow Serving or ONNX Runtime for scalable inference.
- Preprocessing: DICOM handling, adaptive windowing, resizing, and normalization steps must be deterministic and logged.
- Monitoring: track input data drift, model performance metrics, and enable human-in-the-loop feedback and retraining pipelines.
- Compliance: Ensure data governance (PHI handling), clinical validation, and regulatory approvals before clinical use.
6. Example inference server skeleton (TensorFlow):
```python
from fastapi import FastAPI, File, UploadFile
import tensorflow as tf
from PIL import Image
import numpy as np
app = FastAPI()
model = tf.keras.models.load_model("xray_model.h5")
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
contents = await file.read()
img = Image.open(io.BytesIO(contents)).convert("RGB").resize((224,224))
arr = np.array(img)/255.0
```