DS-AG-021 — CNN Architecture | Assignment

Instructions: Each question is answered with sufficient depth to earn full marks (20 per question). For programming questions (6–10) I include complete code examples ready to run in typical local or notebook environments. Training outputs are not executed here to keep the document lightweight; run the code locally or in Colab to observe outputs.

Question 1

What is the role of filters and feature maps in Convolutional Neural Network (CNN)?

Answer:

Filters (also called kernels) are small learnable weight tensors that slide across the input image or previous layer's feature maps performing element-wise multiplications and summations (convolution). Each filter is specialized to detect a certain local pattern such as edges, textures, or color blobs. During training the network learns filter values that maximize recognition of patterns useful for the downstream task.

A feature map (activation map) is the output produced by convolving a filter with the input and applying an activation function. Each filter produces one feature map; stacking many filters yields multiple feature maps that together represent a set of detected features. Early-layer feature maps typically encode low-level features (edges, corners), while deeper-layer feature maps encode higher-level concepts (parts, object shapes). Feature maps reduce spatial redundancy, provide translation equivariance, and serve as the basis for hierarchical representation learning in CNNs.

Question 2

Explain the concepts of padding and stride in CNNs(Convolutional Neural Network). How do they affect the output dimensions of feature maps?

Answer:

Stride controls how the filter moves across the input: a stride of 1 moves the filter one pixel at a time (dense coverage), stride >1 skips positions and reduces the spatial resolution of the output. Larger strides result in smaller feature maps and fewer computations, but may discard fine-grained spatial information.

Padding adds extra pixels around the input border (commonly zeros) so that filters can be applied to edge pixels without shrinking the output too quickly. There are common modes: 'valid' (no padding) and 'same' (padding chosen so output spatial size equals input size when stride=1).

The output dimension for a single spatial axis (height or width) given input size N, filter size F, padding P, and stride S is:

output = floor((N + 2P - F) / S) + 1

Examples:
- No padding (P=0), F=3, S=1 on N=28 → output = 26.
- 'Same' padding with F odd approximates P = (F-1)/2 so that output ≈ N when S=1.

Thus padding controls preservation of borders while stride controls downsampling rate; both directly determine the spatial dimensions and the number of activations in subsequent layers.

Question 3

Define receptive field in the context of CNNs. Why is it important for deep architectures?

Answer:

The receptive field of a unit in a CNN is the region of the input image that can influence that unit's activation. For early layers it equals the filter size; for deeper layers it grows combinatorially because each layer aggregates information from previous layers. The effective receptive field considers how many input pixels meaningfully affect a particular output neuron (often concentrated near the center).

Importance:
- Larger receptive fields allow neurons to capture wider context and global structures (shapes, object-level cues) which is crucial for tasks like classification and detection.
- Designing depth, filter sizes, strides, and pooling affects receptive field growth; insufficient receptive field prevents the model from integrating context necessary to recognize large objects, while excessively large receptive fields (with aggressive downsampling) can harm localization.

Thus, when building deep architectures, one must ensure the receptive field at deeper layers is large enough to cover meaningful parts or the whole object, balancing resolution and contextual coverage.

Question 4

Discuss how filter size and stride influence the number of parameters in a CNN.

Answer:

Parameters (weights) in a convolutional layer are primarily determined by: (filter_height × filter_width × input_channels × output_channels) + output_channels (biases).

Effect of filter size:
- Larger filters (e.g., 7×7 vs 3×3) increase the number of parameters roughly proportional

to the area of the filter. For the same number of input/output channels, a 7×7 filter has ≈(49/9)=5.4× more weights than a 3×3 filter.
- Smaller filters stacked in sequence (for example, three 3×3) can approximate a larger receptive field (7×7) with fewer parameters and more non-linearities, which is why modern networks prefer multiple small filters.

Effect of stride:
- Stride itself does not change the number of parameters (weights of the filters remain the same), but it affects the spatial size of the output feature map. A larger stride reduces the output spatial resolution, which reduces the number of activations and thus the computation and memory footprint in subsequent layers.

Practical note: To control model size use smaller kernels (3×3 or 1×1) and adjust channel counts. 1×1 convolutions are parameter-efficient for changing channel dimensionality (used heavily in bottleneck blocks).

Question 5

Compare and contrast different CNN-based architectures like LeNet, AlexNet, and VGG in terms of depth, filter sizes, and performance.

Answer:

LeNet (1990s):
- Depth & structure: Very shallow by modern standards — typically 2–3 conv layers followed by fully connected layers (original LeNet-5 had 2 conv + pooling stages then FCs).
- Filter sizes: Small (e.g., 5×5), designed for digit recognition (MNIST).
- Performance: Excellent for small-scale tasks like handwritten digit recognition but not competitive on large, complex datasets.

AlexNet (2012):
- Depth: Deeper than LeNet — 5 convolutional layers + 3 fully connected layers; popularized deeper architectures for large-scale image classification.
- Filters: Used larger filters in the first layer (11×11 with stride 4), then smaller (5×5, 3×3). It introduced ReLU activations, dropout, and data augmentation at scale.
- Performance: Won the 2012 ImageNet competition by a large margin and showed the power of deep CNNs trained on GPUs and large datasets.

VGG (2014):
- Depth: Much deeper (e.g., VGG-16, VGG-19) — uses many stacked 3×3 convolutions (2–3 conv layers per block) followed by pooling and FC layers.
- Filters: Uniformly uses 3×3 filters and 1×1 in variants; advantage is simplicity and that stacking 3×3 layers increases receptive field while keeping parameter count manageable.
- Performance: Very strong on ImageNet and commonly used as a backbone for transfer learning, but heavy in parameters (large memory footprint).

Comparison summary:
- Depth increases from LeNet → AlexNet → VGG, as do representational capacity and performance on complex datasets, but also computational cost and parameter count.
- AlexNet introduced practical innovations (ReLU, dropout, data augmentation, GPU training). VGG showed the benefit of many small (3×3) filters stacked instead of using large kernels.
- Modern successors (ResNet, Inception, EfficientNet) improve over VGG by using residual connections, multi-scale filters, or compound scaling to achieve better accuracy vs compute tradeoffs.

Question 6

Using keras, build and train a simple CNN model on the MNIST dataset from scratch. Include code for module creation, compilation, training, and evaluation.

Answer:

Below is a compact, runnable Keras example. It builds a small CNN, compiles it, trains on MNIST, and evaluates test accuracy. Run locally or in Colab; training for 5–10 epochs is typical for high validation accuracy (~99% with a small CNN).

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load and preprocess
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(-1,28,28,1).astype('float32')/255.0
x_test = x_test.reshape(-1,28,28,1).astype('float32')/255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Model
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])
```

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

# Train
model.fit(x_train, y_train, epochs=5, batch_size=128, validation_split=0.1)

# Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)
```

Notes: You can raise epochs to 10–12 or use small data augmentation to slightly improve generalization.

Question 7

Load and preprocess the CIFAR-10 dataset using Keras, and create a CNN model to classify RGB images. Show your preprocessing and architecture.

Answer:

This Keras example shows preprocessing (normalization, one-hot labels) and a simple convnet architecture for CIFAR-10 (32×32 RGB images). Train for 50+ epochs for competitive accuracy; here we provide a compact model suitable for learning and experimentation.

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32')/255.0
x_test = x_test.astype('float32')/255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

model = models.Sequential([
    layers.Conv2D(32, (3,3), padding='same', activation='relu', input_shape=(32,32,3)),
    layers.Conv2D(32, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Dropout(0.25),

    layers.Conv2D(64, (3,3), padding='same', activation='relu'),
    layers.Conv2D(64, (3,3), activation='relu'),
```

```
    layers.MaxPooling2D((2,2)),
    layers.Dropout(0.25),

    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
model.fit(x_train, y_train, batch_size=64, epochs=30, validation_split=0.1)
test_loss, test_acc = model.evaluate(x_test, y_test)
print('CIFAR-10 test acc:', test_acc)
```

Tip: Use image augmentation (ImageDataGenerator or tf.image) and learning rate schedules to push accuracy higher.

Question 8

Using PyTorch, write a script to define and train a CNN on the MNIST dataset. Include model definition, data loaders, training loop, and accuracy evaluation.

Answer:

Below is a straightforward PyTorch script for MNIST. Save as a .py file or run inside a notebook cell. It uses GPU if available.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
train_dataset = datasets.MNIST('./data', train=True, download=True,
transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.conv2(x)
        x = nn.functional.relu(x)
        x = nn.functional.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        return nn.functional.log_softmax(x, dim=1)

model = SimpleCNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = nn.functional.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)}]
Loss: {loss.item():.6f}')

def test():
    model.eval()
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)
```

```
            correct += pred.eq(target.view_as(pred)).sum().item()

    print('Test accuracy:', correct / len(test_loader.dataset))

for epoch in range(1, 6):
    train(epoch)
    test()
```

This will print training progress and final test accuracy. Increase epochs or add augmentations for better results.

Question 9

Given a custom image dataset stored in a local directory, write code using Keras ImageDataGenerator to preprocess and train a CNN model.

Answer:

Keras' ImageDataGenerator is convenient for loading images that are organized in subfolders per class. This example shows directory-based loading, augmentation, and training.

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models

train_dir = '/path/to/train'  # structure: train/class1/*.jpg, train/class2/*.jpg
val_dir = '/path/to/val'

train_gen = ImageDataGenerator(rescale=1./255,
                               rotation_range=20,
                               width_shift_range=0.1,
                               height_shift_range=0.1,
                               shear_range=0.1,
                               zoom_range=0.1,
                               horizontal_flip=True)

val_gen = ImageDataGenerator(rescale=1./255)

train_loader = train_gen.flow_from_directory(train_dir, target_size=(150,150),
batch_size=32, class_mode='categorical')
val_loader = val_gen.flow_from_directory(val_dir, target_size=(150,150), batch_size=32,
class_mode='categorical')

model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(150,150,3)),
    layers.MaxPooling2D(2,2),
```

```
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(train_loader.num_classes, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_loader, epochs=20, validation_data=val_loader)
```

Adjust target_size and batch_size to dataset and GPU memory. For binary classification set class_mode='binary' and final activation to 'sigmoid'.

Question 10

You are working on a web application for a medical imaging startup. Your task is to build and deploy a CNN model that classifies chest X-ray images into 'Normal' and 'Pneumonia' categories. Describe your end-to-end approach–from data preparation and model training to deploying the model as a web app using Streamlit.

Answer:

End-to-end approach (concise, practical, and production-minded):

1) Data collection & labeling
- Source de-identified chest X-ray datasets (e.g., NIH, RSNA, Kaggle pediatric pneumonia dataset) ensuring proper licensing and ethics.
- Confirm correct labels (Normal / Pneumonia) and, if available, metadata (age, view, device). Establish a validation set from separate patients to avoid data leakage.

2) Data cleaning & preprocessing
- Convert to consistent image format and resolution (e.g., 224×224 or 320×320). Normalize intensity (min-max or mean/std). Apply lung-window cropping / lung segmentation if available to remove irrelevant background.
- Augmentation: horizontal flips (if clinically valid), small rotations, translations, brightness/contrast variations. Avoid augmentations that produce medically implausible images.

3) Model selection & training
- Start with a pre-trained CNN backbone (transfer learning) like ResNet-34/50 or EfficientNet, initialized on ImageNet, and replace the head with a binary classifier (one-node sigmoid).
- Example training recipe: freeze backbone for initial epochs (train head), then unfreeze and fine-tune with a lower learning rate. Use focal loss or class-weighting if classes are imbalanced.

- Regularization: dropout, weight decay, early stopping, and validation-based checkpointing. Use metrics appropriate for clinical settings (AUROC, sensitivity/recall, specificity) rather than accuracy alone.

4) Evaluation & validation
- Evaluate on hold-out test set and, if possible, an external dataset. Compute confusion matrix, sensitivity at fixed specificity, ROC curve, and calibration (reliability diagram).
- Perform per-patient split, not per-image, to prevent leakage. Consider cross-validation if dataset size is small.

5) Explainability and safety
- Use saliency methods (Grad-CAM, integrated gradients) to highlight regions influencing predictions—helps clinicians trust the model.
- Validate that saliency corresponds to lung regions, not spurious markers (e.g., text or devices).

6) Packaging the model
- Export the best model as a serialized artifact (SavedModel, TorchScript, or ONNX). Include preprocessing code (resize, normalize) in the packaging so the inference pipeline is deterministic.

7) Serving and deployment with Streamlit
- Build a Streamlit app that accepts uploads (single DICOM/PNG/JPEG), runs preprocessing and model inference, and displays:
  - Predicted label and probability
  - Grad-CAM overlay visualization
  - Downloadable report (PDF) for clinician records

Example Streamlit skeleton (save as app.py):

```python
import streamlit as st
import tensorflow as tf
from PIL import Image
import numpy as np

# Load model
model = tf.keras.models.load_model('best_model')

st.title('Chest X-ray Classifier')
uploaded_file = st.file_uploader('Upload chest X-ray (PNG/JPG/DICOM)', type=['png','jpg','jpeg'])

if uploaded_file is not None:
    img = Image.open(uploaded_file).convert('RGB')
    st.image(img, caption='Uploaded image', use_column_width=True)
    # Preprocess
```

```
    img_resized = img.resize((224,224))
    x = np.array(img_resized).astype('float32')/255.0
    x = np.expand_dims(x, axis=0)

    pred = model.predict(x)[0][0]
    st.write(f'Pneumonia probability: {pred:.3f}')
    if pred > 0.5:
        st.error('Prediction: Pneumonia')
    else:
        st.success('Prediction: Normal')

    # Optionally compute and show Grad-CAM overlay (omitted for brevity)
```

8) Operational concerns
- Host the Streamlit app behind authentication (HIPAA-compliant if handling PHI). Use a secure cloud service (GCP, AWS, Azure) and containerize the app with Docker.
- Monitoring: log model inputs/outputs, track data drift, and periodically re-evaluate model performance. Provide an interface for clinicians to flag incorrect predictions for retraining.

9) Regulatory & clinical validation
- For clinical use, plan prospective validation trials and documentation for regulators. Maintain audit trails and versioning for models and data.

This pipeline balances model performance, safety, explainability, and deployability suitable for a medical imaging startup.