TOPIC FOUR

COMPUTER VISION

SUMMARY NOTES (ET0732)

Lecture Notes: Introduction to Computer Vision Fundamentals with Google Cloud

---

1. Overview of Computer Vision and Its Rapid Growth

- **Definition**: Computer Vision is a subset of Machine Learning (ML) and Artificial Intelligence (AI) that enables computers to interpret and understand visual data from images and videos.
- **Historical Context**:
    - Origin of digital imaging in the 1960s.
    - Significant advancements over the last few decades, driven by cloud computing and specialized hardware.
- **Growth Factors**:
    - Increasing high-resolution image data, thanks to the proliferation of smartphones.
    - Massive amounts of visual data generated, e.g., 79 zettabytes of data in 2021.
    - Internet usage: 93% of people accessed the web through mobile devices.
- **Scale of Data**:
    - Every second: 8.5 hours of videos uploaded to YouTube.
    - Google Photos stored 4 trillion photos by November 2020, with 28 billion new photos and videos uploaded weekly.

---

2. Understanding Computer Vision Problems

- **Types of Problems**:
    - **Image Classification**: Assigns labels to whole images (e.g., identifying an object category).
    - **Semantic Segmentation**: Labels each pixel, segmenting an image into different categories.
    - **Instance Segmentation**: Differentiates objects in an image by identifying boundaries and coloring pixels uniquely.
    - **Image Classification with Localization**: Labels an image and provides bounding boxes for object locations.

- **Object Recognition**: Identifies objects and provides class labels with probabilities but not locations.
- **Object Detection**: Detects and locates multiple objects, providing bounding boxes.
- **Pattern Recognition**: Identifies repeated shapes, colors, and patterns (e.g., facial recognition, OCR).
- **Edge Detection**: Finds object boundaries by detecting changes in brightness and intensity.
- **Feature Matching**: Compares features across images, enabling applications like object tracking and 3D reconstruction.

---

3. Historical Developments in Computer Vision

- **Early Research**:
  - 1960s: Studies on cat vision laid the foundation.
  - 1974: First robust Optical Character Recognition (OCR) system developed, shifting focus to practical applications.
- **2000s Onwards**: Advances in handling complex tasks like image segmentation, object detection, and facial recognition.

---

4. Business Applications of Computer Vision

- **Examples**:
  - **The New York Times**: Uses machine learning for digitizing and categorizing vast photo archives.
  - **Box**: Employs image labeling technology for efficient searching and content analysis.
  - **Harte Research Institute at Texas A&M University**: Uses computer vision to classify shorelines from aerial imagery for environmental analysis.

---

5. Google Cloud's Machine Learning Tools

- **Pre-Built Vision API**:
  - Analyzes images to detect objects, text, emotions, and more.
  - Outputs ranked labels, bounding boxes, and even generates descriptive captions.
- **Capabilities**:
  - Classifies and segments images, detects facial attributes, recognizes text, and automates tasks.

- Example: Describes images (e.g., "Two hockey players are fighting over a puck") and may occasionally make errors.

---

6. Potential and Limitations of Computer Vision Models

- **Applications**:
  - Object detection, disease diagnosis, autonomous driving, etc.
- **Performance**: Some models outperform humans in specific tasks.
- **Challenges**: Even advanced models can misinterpret images, highlighting the need for continuous improvement.

Lecture Notes: Custom Training with Linear, Neural Network, and Deep Neural Network Models

Overview

- **Goal**: Classify images using machine learning techniques, exploring their limitations with image datasets.
- **Focus**: Create custom image classification models from scratch without pre-trained weights, starting with the 5-flowers dataset.

Image Dataset: 5-Flowers

- **Dataset**: 3,700 labeled photographs of five types of flowers.
- **Image Classification Problem**: Classify images into one of the five categories (multiclass classification).

Image Representation

- **Images**: Represented as pixels (2D arrays of numbers) and fed into models as inputs.
- **Classes**: Five possible output classes for classification.

Training and Evaluation

- **Data Split**: Training set and test set.
- **Storage**: Data is stored in Cloud Storage and accessed using TensorFlow datasets.

Techniques for Model Training

1. **Linear Models**: Assumes a linear relationship between input and output.
   - Uses softmax/sigmoid for classification.
   - Computes a weighted sum of inputs plus a bias term.

2. **Neural Network Models**:
    - Extends linear models with multiple layers.
    - Incorporates regularization techniques like dropout to prevent overfitting.
    - Batch normalization to stabilize and speed up training.
3. **Deep Neural Networks**:
    - Multiple hidden layers for learning complex representations.
    - Dropout and batch normalization enhance model performance.

Concepts

- **Dropout**: Reduces overfitting by randomly setting activations to zero during training.
- **Batch Normalization**: Normalizes inputs to layers, improving stability and training speed.

Image Data Processing Workflow

1. **Input Pipeline Setup**:
    - Use tf.io and tf.image for processing images.
    - Read, decode, convert, and resize images to prepare them for the model.
2. **Data Conversion**:
    - Images are read as byte tensors and decoded into 3D uint8 tensors.
    - Convert RGB values to floats and scale between 0 and 1.
3. **Data Resizing**:
    - Resize images as needed using tf.image.resize.
    - Consider padding or cropping to maintain aspect ratio.
4. **Visualization**:
    - Use Matplotlib to visualize images and understand data distribution.

Efficient Data Handling

- **Full Dataset Reading**: Use tf.data.Dataset API to create efficient input pipelines.
    - **Transformations**: Preprocess data in a streaming fashion for memory efficiency.
    - **Batching**: Batch data for model optimization and efficient computation.

Data Extraction and Labeling

- **Class Names**: Extracted from image filenames or provided CSV files.

- **Data Parsing**: Use TensorFlow functions to streamline data preparation and label extraction.

Creating the Dataset

- **Methods**: Use TextLineDataset or TFRecordDataset for structured data processing.
- **Function Definition**: Parse CSV lines, extract filenames, and read images with labels.

Lecture Notes on Implementing Linear Models and Deep Neural Networks with Keras

---

Introduction to Linear Models

- **Start Simple**: Implement simpler models first, introducing complexity only if necessary to meet performance criteria.
    - Simple models are less prone to overfitting and are easier to interpret and maintain.
- **Alternative Approach**: Some techniques use large models with strong regularization from the start.

Implementing Linear Models Using Keras

- **Keras**: High-level API of TensorFlow, designed for efficient and intuitive model building, focusing on modern deep learning.
- **Core Concepts**:
    - **Layers and Models**: Fundamental structures in Keras.
    - **Sequential Model**: A straightforward stack of layers where each layer feeds into the next. Suitable for simple, linear models but not for models needing multiple inputs/outputs or complex topologies.

Steps to Create and Train a Model in Keras

1. **Define and Create Model**: Use the Sequential class and stack layers.
2. **Compile Model**: Configure optimizer, loss function, and metrics.
3. **Optional**: Use model.summary() to display the model architecture.
4. **Train Model**: Use model.fit() with training data, specifying the number of epochs.
5. **Evaluate Model**: Plot training and validation loss/accuracy to assess performance.
6. **Make Predictions**: Use model.predict() and convert logits to probabilities using softmax.

Example Model Architecture

```python
python

Copy code

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    tf.keras.layers.Dense(len(CLASS_NAMES))
])
```

- **Flatten Layer**: Converts a 3D image (e.g., 224x224x3) into a 1D array. It doesn't learn parameters; it only reshapes data.
- **Dense Layer**: Computes the weighted sum of inputs and applies an activation function like softmax to output probabilities.

Model Compilation

- **Settings**:
  - **Optimizer**: E.g., Adam, adjusts model weights to minimize loss.
  - **Loss Function**: Measures the model's error; SparseCategoricalCrossentropy is used for multi-class classification.
  - **Metrics**: E.g., accuracy, to monitor performance.

Model Training and Evaluation

- **Training**: Use model.fit() with both training and validation datasets.
- **History and Callbacks**: Record and plot loss and accuracy to analyze the model's convergence and detect overfitting.

Making Predictions and Understanding Outputs

- **Model Outputs**: Logits transformed into probabilities via softmax.
- **Confidence Values**: Higher confidence indicates stronger model prediction for a class.

Deep Neural Networks (DNNs) and Regularization Techniques

- **Universal Approximation Theorem**: A single hidden layer can theoretically solve any problem, but large models are impractical.
- **Challenges with Large Models**:
  - Increased memory usage, slower training, and higher risk of overfitting.
- **Overfitting Prevention**:

    - **Regularization**: Dropout, L1, and L2.
    - **Dropout**: Randomly deactivates neurons during training to prevent overfitting.
    - **Batch Normalization**: Normalizes inputs of each layer to speed up training and improve stability.

Dropout and Batch Normalization

- **Dropout**: Applies a dropout layer with a probability p to deactivate neurons during training.
- **Batch Normalization**: Normalizes activations using batch statistics, stabilizing and accelerating learning.

Implementing Regularization in Keras

- **Dropout**: tf.keras.layers.Dropout(rate), used only during training.
- **Batch Normalization**: tf.keras.layers.BatchNormalization(), applied before the activation function.

Summary

- **Model Building**: Learned how to represent and train linear models, and progressively more complex neural networks for image classification.
- **Regularization Techniques**: Dropout and batch normalization to improve model generalization.

Lecture Notes: Introduction to Convolutional Neural Networks (CNNs)

---

1. Overview of CNNs

- CNNs specialize in detecting visual patterns within images.
- They differ from traditional neural networks by employing **convolutions** to extract features, sliding filters over image pixels to imitate how the human brain processes visual data.

---

2. Key Differences and Concepts

- **Feature Extraction**: Convolutions mimic the hierarchical approach of the visual cortex, emphasizing locally correlated features.

- CNNs use **filters** (kernels) that operate on images to create **feature maps**.
- **Parameters to Consider**:
    - **Filters**: The number and type affect feature extraction.
    - **Channels**: Define input data (e.g., RGB for color images).
    - **Kernel Size**: Determines the receptive field of filters.
    - **Strides & Padding**: Control how filters slide and manage border effects.
    - **Activation Functions**: Introduce non-linearity, such as ReLU.

---

3. Pooling Layers

- **Purpose**: Reduce the spatial dimensions of feature maps, making computations more efficient and reducing sensitivity to object location.
- Common types include **Max Pooling** (selects maximum value) and **Average Pooling** (computes average value).

---

4. Evolution and History

- **1980**: Kunihiko Fukushima developed the **Neocognitron**, inspired by simple and complex cells in the human brain.
- **Late 1980s-1990s**: CNNs were formalized by **Yann LeCun**, leading to models like **LeNet**, which was pivotal for tasks like handwriting recognition.
- **2012**: **AlexNet**'s win in the ImageNet competition marked a breakthrough, establishing CNNs as a key framework in computer vision.

---

5. CNN Applications and Beyond

- Widely used for **image classification** and **object detection**.
- Can also be applied to non-image data (e.g., audio, time series) but rely heavily on local feature hierarchies.
- Prior to CNNs, feature engineering required manual image preprocessing, whereas CNNs learn relevant features autonomously.

---

6. Dense Layers Recap

- **Dense Layers**: Every input is connected to every neuron, leading to vast parameter counts for high-resolution images.
- Dense layers lack hierarchical structuring, making pixel order irrelevant for classification tasks.

- CNNs contrast this by preserving the local structure, vital for visual tasks.

---

7. Feature Engineering and Modern CNNs

- Traditional models relied on engineered features (e.g., using Gabor filters) for pattern recognition.
- **Post-2012**: CNNs simplify design by learning features directly from images, with models like **AlexNet** and **Inception** setting standards.

---

8. Detailed CNN Layers

- **Convolutional Layers**: Use kernels to detect spatial patterns, with shared weights promoting efficiency.
- **Example**: A 5x5 image processed with a 3x3 kernel results in a smaller feature map, reduced by kernel size minus one.
- CNNs stack multiple filters, each detecting different features.

---

9. Specialized CNN Types

- **1D CNNs**: For sequential data (e.g., audio, time series).
- **3D CNNs**: For 3D data (e.g., videos, MRI scans).
- **Input Representation**: Grayscale images use 2D tensors; color images use 3D tensors (RGB channels).

---

10. Kernel Operations

- Kernels detect edges and patterns by computing the weighted sum of pixels.
- Example: Edge detection kernels highlight areas with intensity changes.
- **Parameter Sharing**: Kernels have consistent weights, enhancing efficiency.

---

11. Implementing CNNs

- **Keras** simplifies CNN construction, automating layer setup and enabling efficient feature learning.
- **Edge Detection Example**: Using weighted kernels to identify horizontal and vertical features.
- CNNs learn filters to progressively build complex representations, from edges to entire objects.

---

12. Practice and Quiz Insights

- **Quiz Concept**: Understanding kernel operations and how convolutions identify features.
- Practical implementation involves configuring filters, activation functions, and other parameters.

Lecture Notes: CNN Model Parameters and Operations

1. ML Model Parameters Recap

- **Definition**: Parameters are values learned during training to transform input data into the predicted output.
- **Complexity and Parameters**: As a model's complexity increases, so does the number of trainable parameters.
    - *Example*: Simple neural networks have weights and biases; CNNs have additional parameters.

---

2. Creating a Convolution Layer in Keras

- **Conv2D Layer**: tf.keras.layers.Conv2D method creates a 2D convolutional layer in Keras.
- **Input/Output Format**: CNNs handling image recognition expect 4D tensors: [batch, height, width, channels].
    - *Example*: For 256x256 RGB images, the input shape is [256, 256, 3]. With a batch of 16, it becomes [16, 256, 256, 3].

---

3. CNN Model Parameters

- **Number of Filters**: Determines the number of independent filters applied. Output channels = number of filters.
- **Input Channels**: Based on the input image's channel count. For a 256x256 RGB image: 3 channels.
- **Kernel Size**: Defines the dimensions of each filter (e.g., 3x3, 5x5). Smaller kernels with multiple layers are efficient.
- **Strides**: Step size for the filter sliding across the image. Default is 1. Larger strides reduce output size but can skip information.
- **Padding**: Adds borders to maintain input-output dimensions. Methods:
    - same: Keeps output size the same as input.

- valid: No padding; reduces output size.

---

4. Calculating Parameters in CNNs

- **Convolution Layer**: Parameters = (width * height * input channels + 1) * number of filters.
- **Pooling Layers**: No learnable parameters; used to reduce dimensionality and computation.
- **Fully Connected Layers**: High parameter count. Parameters = (neurons in current layer * neurons in previous layer + 1) * current neurons.

---

5. Pooling Operations

- **Purpose**: Reduce feature map dimensions and computations.
- **Max Pooling**: Returns the maximum value within a filter window.
  - *Example*: 4x4 input reduced to 2x2 using a 2x2 filter with a stride of 2.
- **Average Pooling**: Computes average values in the filter window.
- **Global Pooling**: Used for summarizing information from the feature map.
- **Implementation in Keras**: tf.keras.layers.MaxPooling2D(pool_size=2, strides=1)

---

6. Convolution vs. Dense Layers

- **Dense Layers**: Connect all input pixels to every neuron, requiring a large number of parameters.
- **Convolution Layers**: Use kernels to detect patterns (edges, textures) with fewer weights, improving efficiency and reducing training time.
- **Example**: MNIST dataset - using convolution reduces parameter count compared to dense layers.

---

7. Key Takeaways

- CNNs use fewer parameters than dense layers, enhancing efficiency in image processing tasks.
- Convolution and pooling layers recognize and compress patterns, passing a flattened feature vector to a fully connected network.
- Use model.summary() in Keras to check parameter details easily.

Quiz Question Review

- **Advantage of Convolution Kernels**: Reduces parameters by processing small patches and sharing weights, speeding up training.

Lecture Notes: Dealing with Image Data in Computer Vision

---

1. Overview of Section:

- **Objective:** Learn how to preprocess image data for reproducibility in production.
- **Hands-On Labs:** Implement preprocessing in Keras using TensorFlow datasets.
- **Key Topics:** Relationship between model parameters and data scarcity, data augmentation, transfer learning, and efficient data storage formats.

---

2. Creating and Reading Image Datasets:

- **Creating a Dataset:** Collect and annotate image data using internal resources or third-party services.
- **Validating Data:** Ensure image data is in formats like JPEG/PNG but avoid inefficiencies (e.g., CSV files) for large datasets.
- **Efficient Data Format:** Use tf.records for better I/O operations and storage efficiency, essential for optimized training on frameworks like TPUs.

---

3. TensorFlow Data Handling:

- **Using tf.data API:** Construct efficient data pipelines with TFRecordDataset for fast processing.
- **Example Pipeline:** Aggregate data, apply random transformations, and batch images for training.

---

4. Image Preprocessing:

- **Purpose:** Prepare raw images for model training by resizing, color conversion, and other transformations.
- **Techniques:** Resizing (using tf.image.resize), flipping, rotating, and cropping images to match model input shapes.
- **Aspect Ratio Management:** Use tf.image.resize_with_pad to maintain the original aspect ratio and avoid distortion.
- **Learned Resizers:** Enhance performance using custom resizers beyond traditional methods like bilinear.

---

5. Keras Preprocessing Layers:

- **API Usage:** Keras provides layers like Resizing and Rescaling to standardize inputs.

- **Integration:** Combine preprocessing layers with models to streamline training and prediction.

---

6. Handling Data Scarcity:

- **Problem Overview:** Modern models require vast labeled datasets, but data can be limited.
- **CNNs' Needs:** Models with more parameters demand more data for effective training.

---

7. Strategies for Data Scarcity:

- **Data Augmentation:** Increase dataset size by applying transformations (e.g., cropping, flipping, color adjustments).
- **Transfer Learning:** Use pre-trained models to leverage knowledge from similar tasks, reducing data requirements.

---

8. Data Augmentation Techniques:

- **Common Transformations:** Blurring, sharpening, resizing, rotating, flipping, and color adjustments.
- **Considerations:** Ensure transformations do not compromise the model's learning. E.g., orientation is critical in cases like distinguishing flags or species.
- **Task-Specific Augmentation:** Tailor techniques to the data's characteristics and domain requirements.

---

9. Implementing Data Augmentation:

- **Using tf.image:** Write augmentation pipelines with methods like flip_left_right or rgb_to_grayscale.
- **Random Transformations:** Use APIs like tf.image.stateless_random_brightness for controlled randomness.
- **Parallelization with tf.data:** Use Dataset.map to parallelize preprocessing.

---

10. Keras Image Augmentation Layers:

- **Built-In Layers:** RandomTranslation, RandomRotation, RandomZoom, etc., for augmenting images during training.
- **Inference Handling:** Preprocessing is only applied during training, simplifying the prediction phase.
- **Training Considerations:** Training with augmented data often requires longer training times due to increased data size.

---

11. Quiz Questions (Examples):

- **Parameter Count Calculations:** Analyze the parameter count in models, e.g., linear and convolutional layers.
- **Practical Applications:** Discuss scenarios where specific augmentations would or would not improve model performance.

Lecture Notes on Transfer Learning

---

1. Introduction to Transfer Learning

- **Purpose**: A method to address data scarcity by decreasing the need for a large amount of labeled data.
- **Approach**: Instead of creating more data, transfer learning initializes model parameters with values from a pre-trained model, enhancing efficiency.

---

2. Optimization and Training

- **Optimization Journey**: The process of finding optimal weights for the model to minimize data and time expense.
- **From Scratch vs. Transfer Learning**:
  - *Training from Scratch*: Time-consuming and resource-intensive.
  - *Transfer Learning*: Utilizes a pre-trained model trained on similar tasks, saving both time and data.

---

3. Core Concept of Transfer Learning

- **How It Works**: Knowledge from a source model trained on a large dataset is transferred to a new, related task.
- **Benefit**: Significantly reduces training time compared to starting from scratch, especially when source and target tasks are similar.

---

4. Example: Using ImageNet for Transfer Learning

- **ImageNet**: A large dataset with 14 million labeled images across 20,000 categories.
- **Naive Transfer Learning**: Using the ImageNet model directly for predictions may not work if the target classes differ in number or specificity.
- **Advanced Transfer Learning**: Involves modifying the source model:
  - Replace parts closely tied to the source task (e.g., output layers).

- Retain generalized parts (e.g., convolutional layers) for feature extraction.

---

5. Model Layers and Task Dependence

- **Input Layer**: Task-independent and can handle general input, like any RGB image.
- **Convolutional Layers**: Generally task-independent, used for feature extraction.
- **Output Layers**: Highly task-dependent, aligned with the specific output requirements of the source task.
- **Feature Hierarchy**:
  - CNNs learn general to specific features.
  - Early layers detect simple patterns; later layers become more specific and task-dependent.

---

6. Adjusting the Source Model

- **Where to Cut the Network**: No clear point due to distributed representations in neural networks.
- **Standard Practice**: Cut after the convolutional layers and add fully connected layers suited to your task.
- **Weight Training Decisions**:
  - *Constant Weights*: Use the source model as a feature extractor, recommended for small datasets to avoid overfitting.
  - *Trainable Weights*: Adjust weights if your dataset is large enough, reducing overfitting risks.

---

7. Implementing Transfer Learning

- **Pre-trained Models**: Models trained on extensive datasets (e.g., ImageNet) that are readily available for use.
- **MobileNet Example**:
  - Pre-trained on ImageNet with 1-4 million parameters.
  - Efficient at compressing visual information, making it suitable for related image classification tasks.
  - Available on TensorFlow Hub for easy integration with Keras.

---

8. Recap and Applications

- **Preprocessing for Reproducibility**: Important for consistent performance in production.

- **Data Augmentation vs. Transfer Learning**:
  - *Data Augmentation*: Increases data variety and model robustness.
  - *Transfer Learning*: Leverages existing knowledge to minimize data needs.
- **Next Steps**: Reviewing key concepts and applying them to practical problems.