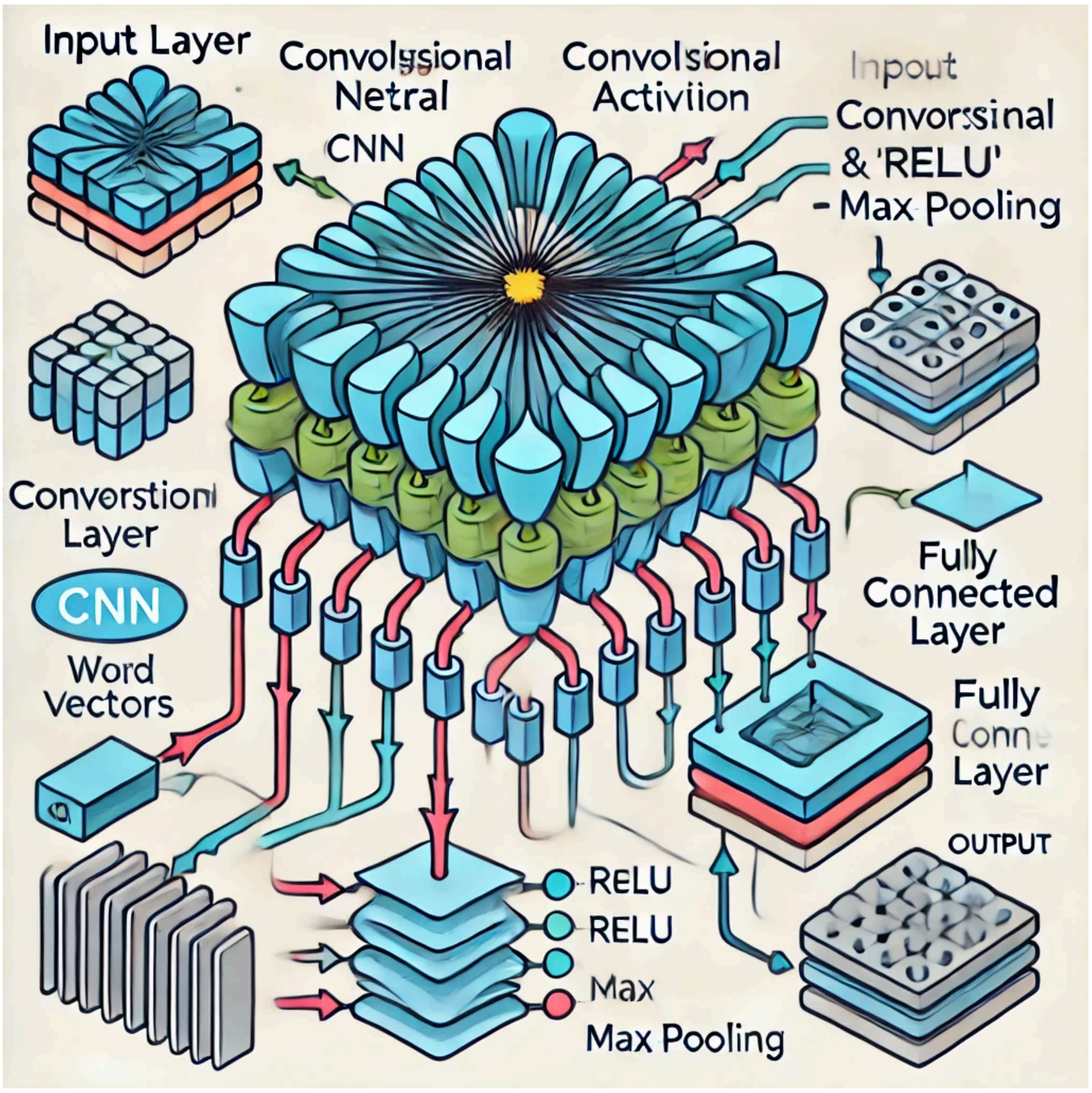
005.5 CNN



Convolutional Neural Networks (CNNs) are a class of deep learning models primarily used for processing data with grid-like topology, such as images and time-series data. CNNs are designed to automatically and adaptively learn spatial hierarchies of features from input data, making them particularly effective in tasks involving images, video, and even certain Natural Language Processing (NLP) tasks.

Convolutional Neural Networks (CNNs) are highly specialized neural networks that excel at processing grid-like data such as images and time-series data. They consist of convolutional layers for feature extraction, pooling layers for downsampling, and fully connected layers for final classification or regression tasks. CNNs are powerful because of their ability to learn spatial hierarchies of features, making them the go-to model for many computer vision and image-related tasks.

Key Concepts of CNNs:

1. Convolutional Layer

The core building block of a CNN is the **convolutional layer**, which performs a mathematical operation called convolution. Convolutional layers apply filters (also known as kernels) to the input data to extract important features like edges, textures, or patterns.

- **Filter/Kernels**: A filter is a small matrix of weights that slides (convolves) across the input data. The result of applying the filter is called the **feature map** or **activation map**. Filters are learned during training and help detect specific patterns in the input data.
- **Stride**: This refers to how much the filter moves at each step as it slides across the input. A stride of 1 means the filter moves one pixel at a time, while a stride of 2 moves two pixels at a time.
- **Padding**: Sometimes, the input data is padded with extra pixels around the edges to control the output size. Padding ensures that the convolution operation does not shrink the output size significantly.

The convolution operation can be mathematically expressed as:

$$(I * K)(i,j) = \sum_{m} \sum_{n} I(i+m, j+n) \cdot K(m,n)$$

Where:

- I is the input matrix (e.g., an image).
- K is the kernel (filter) applied.
- (i,j) represents the spatial coordinates in the output.

2. Activation Function

After the convolution operation, an **activation function** is applied to introduce non-linearity. The most commonly used activation function in CNNs is the **Rectified Linear Unit (ReLU)**:

$$\mathrm{ReLU}(x) = \max(0, x)$$

ReLU ensures that the network can capture complex relationships in the data by allowing only positive values to pass through.

3. Pooling Layer

The **pooling layer** is used to reduce the spatial dimensions (width and height) of the feature maps, making the network computationally efficient and reducing the risk of overfitting.

- **Max Pooling**: The most common pooling operation, where the maximum value from a small region of the feature map is taken.

- **Average Pooling**: Takes the average of values in the small region, although it is less commonly used than max pooling.

Pooling layers are important for downsampling the data while preserving important features. For instance, if max pooling is applied over a 2x2 region, the feature map size is reduced by a factor of 2 in both dimensions.

4. Fully Connected Layer (Dense Layer)

After a series of convolutional and pooling layers, the feature maps are flattened into a vector and passed through one or more **fully connected layers** (also called **dense layers**). These layers perform the final classification or regression tasks.

- **Flattening**: Converts the 2D feature maps into a 1D vector.
- **Classification**: In tasks like image classification, the fully connected layer typically outputs a probability distribution over classes using a **softmax** activation function for multiclass classification or a **sigmoid** activation function for binary classification.

5. CNN Architecture

The typical architecture of a CNN consists of multiple layers stacked in the following order:

1. **Input Layer**: Raw data like images or sequences.
2. **Convolutional Layers**: Multiple layers of convolution with filters to learn different feature hierarchies.
3. **Pooling Layers**: Downsampling layers to reduce dimensionality.
4. **Fully Connected Layers**: Dense layers that perform the final classification or prediction task.
5. **Output Layer**: For classification, this outputs the probability for each class.

6. Training CNNs

CNNs are trained using backpropagation and gradient descent, just like other neural networks:

- **Forward Propagation**: Input data is passed through the network layer by layer, applying convolutions, activation functions, and pooling operations.
- **Loss Function**: A loss function (e.g., cross-entropy for classification) measures the difference between the predicted and actual output.
- **Backpropagation**: The error is propagated backward through the network to update the weights of the filters and fully connected layers.
- **Optimization**: Gradient descent (or a variant like Adam) is used to optimize the weights during training.

7. Key Advantages of CNNs

- **Spatial Invariance**: CNNs are highly effective at recognizing patterns regardless of their position in the input (translation invariance). This is crucial in tasks like image recognition.

- **Parameter Sharing**: The same filters are applied across the entire input, reducing the number of parameters and making CNNs more efficient.
- **Automatic Feature Learning**: CNNs can automatically learn relevant features from data during training without requiring manual feature engineering.

8. Applications of CNNs

CNNs are widely used in various fields:

- **Image Classification**: Identifying objects or scenes in images (e.g., facial recognition, object detection).
- **Object Detection**: Locating and classifying objects in an image.
- **Natural Language Processing (NLP)**: CNNs can be applied to text classification, sentence modeling, and machine translation tasks by treating text sequences like 1D grids.
- **Medical Imaging**: Diagnosing diseases from X-rays, MRI scans, etc.
- **Video Processing**: Action recognition or event detection in videos.

Example in Python

Here is an example of how to implement a **Convolutional Neural Network (CNN)** for a text classification task, such as sentiment analysis, using **PyTorch**. CNNs can be applied to NLP tasks by treating text as a sequence of word embeddings and applying convolutional filters to extract features.

Requirements:

You'll need to install the following packages:

```
pip install torch torchtext spacy
python -m spacy download en_core_web_sm
```

Python Code: CNN for Text Classification (IMDb Dataset)

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchtext.legacy import data, datasets
import random

# Set the random seed for reproducibility
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

# Define fields for text and labels
```

```
TEXT = data.Field(tokenize='spacy',
tokenizer_language='en_core_web_sm', batch_first=True)
LABEL = data.LabelField(dtype=torch.float)

# Load IMDb dataset
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

# Build vocabulary using pre-trained GloVe embeddings
TEXT.build_vocab(train_data, max_size=25_000,
vectors="glove.6B.100d", unk_init=torch.Tensor.normal_)
LABEL.build_vocab(train_data)

# Create iterators for batching
BATCH_SIZE = 64

train_iterator, test_iterator = data.BucketIterator.splits(
      (train_data, test_data),
      batch_size=BATCH_SIZE,
      sort_within_batch=True,
      device=torch.device(
            'cuda' if torch.cuda.is_available() else 'cpu'
      )
)

# Define the CNN model
class CNN(nn.Module):
      def __init__(
            self, vocab_size, embedding_dim,
            n_filters, filter_sizes, output_dim,
            dropout, pad_idx
      ):
            super(CNN, self).__init__()

            # Embedding layer
            self.embedding = nn.Embedding(
                  vocab_size, embedding_dim, padding_idx=pad_idx
            )

            # Convolutional layers (one for each filter size)
            self.convs = nn.ModuleList([
                  nn.Conv2d(
                        in_channels=1,
                        out_channels=n_filters,
                        kernel_size=(fs, embedding_dim)
                  )
                  for fs in filter_sizes
            ])
```

```
        # Fully connected output layer
        self.fc = nn.Linear(
            len(filter_sizes) * n_filters, output_dim
        )

        # Dropout layer
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        # text: [batch_size, sentence_length]

        embedded = self.embedding(text)
        embedded = embedded.unsqueeze(1)

        # Apply convolution + ReLU + max pooling
        conved = [
            torch.relu(conv(embedded)).squeeze(3)
            for conv in self.convs
        ]
        pooled = [torch.max(conv, dim=2)[0] for conv in conved]

        # Concatenate pooled feature maps and apply dropout
        cat = self.dropout(torch.cat(pooled, dim=1))

        # Fully connected layer
        return self.fc(cat)

# Hyperparameters
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100  # Must match the GloVe embedding size
N_FILTERS = 100
FILTER_SIZES = [3, 4, 5]
OUTPUT_DIM = 1
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

# Initialize the model
model = CNN(INPUT_DIM, EMBEDDING_DIM, N_FILTERS, FILTER_SIZES,
OUTPUT_DIM, DROPOUT, PAD_IDX)

# Load pre-trained embeddings
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)

# Zero the embedding weights for the padding token
model.embedding.weight.data[PAD_IDX] = torch.zeros(EMBEDDING_DIM)

# Training setup
```

```
optimizer = optim.Adam(model.parameters())
criterion = nn.BCEWithLogitsLoss()

# Move the model and criterion to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
model = model.to(device)
criterion = criterion.to(device)

# Function to calculate binary accuracy
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    return correct.sum() / len(correct)

# Training function
def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:
        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)
        loss = criterion(predictions, batch.label)
        acc = binary_accuracy(predictions, batch.label)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

# Evaluation function
def evaluate(model, iterator, criterion):
    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():
        for batch in iterator:
            predictions = model(batch.text).squeeze(1)
            loss = criterion(predictions, batch.label)
```

```
                acc = binary_accuracy(predictions, batch.label)

                epoch_loss += loss.item()
                epoch_acc += acc.item()

     return epoch_loss / len(iterator), epoch_acc / len(iterator)

# Training loop
N_EPOCHS = 5

for epoch in range(N_EPOCHS):
     train_loss, train_acc = train(
          model, train_iterator, optimizer, criterion
     )
     test_loss, test_acc = evaluate(model, test_iterator,
criterion)

     print(f'Epoch {epoch+1}')
     print(f'\tTrain Loss: {train_loss:.3f} '
          f'| Train Acc: {train_acc*100:.2f}%')
     print(f'\tTest Loss: {test_loss:.3f} '
          f'| Test Acc: {test_acc*100:.2f}%')

# Save the model
torch.save(model.state_dict(), 'cnn_model.pth')
```