# Chapter 14 Report:
# Classifying Images with CNNs

Quan Pham

June 21, 2025

## Contents

# 1 Abstract

In this chapter, I studied convolutional neural networks (CNNs) and their application in image classification tasks. CNNs are designed to extract hierarchical features from images, which makes them highly effective for vision-related problems.

# 2 Key Concepts

## 2.1 Convolutional Neural Networks

- CNNs mimic the human visual cortex and automatically learn features from raw data.
- Feature hierarchies: low-level features $\rightarrow$ high-level representations.
- Key ideas: **sparse connectivity** and **parameter sharing**.
- Pooling layers reduce spatial dimensions and help with generalization.

## 2.2 Discrete Convolution

$$y[i] = \sum_{k=-\infty}^{+\infty} x[i-k] \cdot w[k]$$

$$Y[i,j] = \sum_{k_1} \sum_{k_2} X[i-k_1, j-k_2] \cdot W[k_1, k_2]$$

Important hyperparameters: padding (full, same, valid), stride.

## 2.3 Pooling Layers

- Max-pooling and mean-pooling.
- Improve robustness to noise, reduce overfitting.
- Can use overlapping or non-overlapping pooling.

## 2.4 Loss Functions

- Binary classification: `BCEWithLogitsLoss`, `BCELoss`
- Multiclass: `CrossEntropyLoss`, `NLLLoss`

# 3 Algorithms and Models Studied

## 3.1 Forward pass in convolution layer

$$Z^{\text{conv}}[:,:,k] = \sum_{c=1}^{C_{in}} W[:,:,c,k] * X[:,:,c]$$
$$Z[:,:,k] = Z^{\text{conv}} + b[k]$$
$$A[:,:,k] = \sigma(Z[:,:,k])$$

## 3.2 Adam optimization

**Adam** (Adaptive Moment Estimation) is a widely used optimization algorithm in training neural networks. It combines the advantages of **AdaGrad** (adaptive learning rates for individual parameters) and **RMSprop** (using a moving average of squared gradients). Adam is effective due to its ability to adaptively adjust the learning rate for each parameter based on estimates of the first-order moments (mean) and second-order moments (variance) of the gradients.

### Basic Steps:

Given a learning rate $\alpha$, decay rates $\beta_1, \beta_2 \in [0, 1)$, and a small constant $\epsilon$ to prevent division by zero.

1. Initialize moment vectors: $m_0 = \mathbf{0}$, $v_0 = \mathbf{0}$.
2. For each iteration $t = 1, 2, \ldots$:
   - Compute the gradient $g_t$ of the loss function at time $t$:
   $$g_t = \nabla_\theta J(\theta_{t-1})$$
   - Update biased first moment estimate (mean of gradients):
   $$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
   - Update biased second moment estimate (mean of squared gradients):
   $$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
   - Perform bias correction for the moments (to counteract initial values being zero):
   $$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
   $$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
   - Update model parameters:
   $$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

### Advantages:

- **Efficiency:** Performs well in practice across various machine learning tasks.
- **Adaptive:** Adjusts the learning rate for each parameter, leading to faster convergence.
- **Ease of Use:** Does not require extensive hyperparameter tuning; default values often yield good results.

## 3.3 Regularization

- L2 regularization (weight decay) and dropout help prevent overfitting.
- Dropout encourages robust feature learning.

## 3.4 Other Techniques

- **Data augmentation**: enhances generalization.
- **Global average pooling**: reduces parameters.

## 3.5 Example models in the book

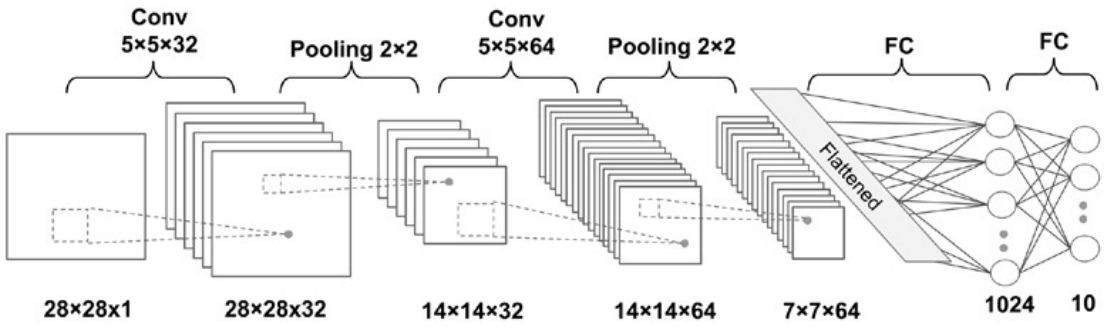


Figure 1: A deep CNN

# 4 Code Examples and Practical Exercises

In this chapter, I delve into two image classification problems presented in the book. I begin by carefully studying the provided code examples to grasp the fundamental concepts and implementation details. Following this, I engage in hands-on practice to solidify my understanding and explore potential modifications or extensions.

## 4.1 Datasets

In this study, two distinct datasets are employed: the MNIST dataset, a widely recognized benchmark for handwritten digit classification, and the CelebA dataset, a large-scale face attributes dataset. The MNIST dataset serves as an initial case study for implementing and evaluating a basic CNN architecture, while the CelebA dataset is utilized for a more complex task: smile classification from face images.

## 4.2 Implementing a deep CNN using PyTorch

In the previous chapter, I achieved 95.6% accuracy in handwritten digit recognition using a neural network with two hidden layers. In this chapter, I implement a CNN to explore whether it can achieve superior predictive performance on the same task.

### 4.2.1 Loading and preprocessing the MNIST dataset

```
import torchvision
from torchvision import transforms
from torch import nn

image_path = '../data/'
transform = transforms.Compose([
    transforms.ToTensor()
])
mnist_dataset = torchvision.datasets.MNIST(root=image_path, train=True,
 ↪  transform=transform, download=False)

from torch.utils.data import Subset
mnist_valid_dataset = Subset(mnist_dataset, torch.arange(10000))
mnist_train_datset = Subset(mnist_dataset, torch.arange(10000,
 ↪  len(mnist_dataset)))
mnist_test_dataset = torchvision.datasets.MNIST(root=image_path,
 ↪  train=False, transform=transform, download=False)

from torch.utils.data import DataLoader
batch_size = 64
torch.manual_seed(1)
train_dl = DataLoader(mnist_train_datset, batch_size, shuffle=True)
valid_dl = DataLoader(mnist_valid_dataset, batch_size, shuffle=False)
```

### 4.2.2 Construct the CNN model

```
model = nn.Sequential()
model.add_module(
    'conv1',
```

```
    nn.Conv2d(
        in_channels=1, out_channels=32, kernel_size=5, padding=2
    )
)
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
model.add_module(
    'conv2',
    nn.Conv2d(
        in_channels=32, out_channels=64, kernel_size=5, padding=2
    )
)
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
model.add_module('fc1', nn.Linear(3136, 1024))
model.add_module('relu3', nn.ReLU())
model.add_module('dropout', nn.Dropout(p=0.5))
model.add_module('fc2', nn.Linear(1024, 10))
```

#### 4.2.3 Train and evaluate the model

I used cross-entropy loss and Adam optimizer with learning rate 0.001.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

This is the output of training progress:

```
Epoch 1 accuracy: 0.9496 val_accuracy: 0.9792
...
Epoch 10 accuracy: 0.9964 val_accuracy: 0.9902
...
Epoch 20 accuracy: 0.9981 val_accuracy: 0.9909
```

We observe that the CNN model achieves a validation accuracy of 99.09% on the MNIST dataset. This result demonstrates a significant improvement in predictive performance compared to the 95.6% accuracy achieved with the simple neural network in the previous chapter, highlighting the effectiveness of CNNs for image classification tasks.

### 4.3 Smile classification from face images using CNN

```
get_smile = lambda attr: attr[18]

transform_train = transforms.Compose([
    transforms.RandomCrop([178, 178]),
```

```
    transforms.RandomHorizontalFlip(),
    transforms.Resize([64, 64]),
    transforms.ToTensor(),
])

transform = transforms.Compose([
    transforms.CenterCrop([178, 178]),
    transforms.Resize([64, 64]),
    transforms.ToTensor(),
])
```

```
celeba_train_dataset = torchvision.datasets.CelebA(image_path,
 ↪  split='train', target_type='attr', download=False,
 ↪  transform=transform_train, target_transform=get_smile)
celeba_train_dataset = Subset(celeba_train_dataset, torch.arange(16000))
celeba_valid_dataset = Subset(celeba_valid_dataset, torch.arange(1000))
```

```
batch_size = 32
torch.manual_seed(1)
train_dl = DataLoader(celeba_train_dataset, batch_size, shuffle=True)
valid_dl = DataLoader(celeba_valid_dataset, batch_size, shuffle=False)
test_dl = DataLoader(celeba_test_dataset, batch_size, shuffle=False)
```

```
model = nn.Sequential()
model.add_module('conv1', nn.Conv2d(in_channels=3, out_channels=32,
 ↪  kernel_size=3, padding=1))
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(kernel_size=2))

model.add_module('conv2', nn.Conv2d(in_channels=32, out_channels=64,
 ↪  kernel_size=3, padding=1))
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(kernel_size=2))

model.add_module('conv3', nn.Conv2d(in_channels=64, out_channels=128,
 ↪  kernel_size=3, padding=1))
model.add_module('relu3', nn.ReLU())
model.add_module('pool3', nn.MaxPool2d(kernel_size=2))

model.add_module('conv4', nn.Conv2d(in_channels=128, out_channels=256,
 ↪  kernel_size=3, padding=1))
model.add_module('relu4', nn.ReLU())

model.add_module('pool4', nn.AvgPool2d(kernel_size=8))
```

```
model.add_module('flatten', nn.Flatten())
model.add_module('fc', nn.Linear(256, 1))
model.add_module('sigmoid', nn.Sigmoid())
```

```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
Epoch 1 accuracy: 0.6350 val_accuracy: 0.7100
...
Epoch 15 accuracy: 0.8899 val_accuracy: 0.8990
...
Epoch 30 accuracy: 0.9139 val_accuracy: 0.9030
```

### 4.4 Modularizing the code

The notebook files contain a substantial amount of code. It would be advantageous to divide the code into smaller, more manageable modules for easier maintenance and modification.

There are three processes that I can modularize: loading and processing data, building model classes, and building trainer classes. The files created are `MyData.py`, `MyModels.py`, and `MyTrainer.py`.

### 4.5 Practice on CelebA dataset

To further enhance my understanding and skills, I engaged in a series of practical exercises using the CelebA dataset. These exercises focused on several key areas:

**Data Loading and Balancing:** I implemented a data loading process that specifically addressed the class imbalance between 'smile' and 'not smile' instances within the training dataset. The training dataset was configured to use 10,000 examples. A validation dataset of 1,000 examples was also prepared.

**CNN Architecture Exploration:** I experimented with different CNN architectures to evaluate their impact on smile classification performance. These architectures were named v1, v2, v3, and v4, each representing a unique configuration of convolutional layers, pooling layers, and fully connected layers. The goal was to identify the architecture that yielded the highest accuracy and generalization ability.

**Model Persistence:** I learned how to save trained models and checkpoints, enabling the resumption of training from a specific point and the reuse of trained models for inference. This involved using PyTorch's model saving and loading functionalities.

## 5 Learnings and Challenges

### 5.1 Key Takeaways

- CNNs are highly effective for vision-ralated problems.

- Convolution layer, pooling layer.
- Regularization for CNN: L1, L2, dropout.
- Data augmentation: flipping, rotating, scaling, and cropping.
- Building CNN classes in Pytorch with `nn.Sequential` or `nn.Module`
- Save and load models

### 5.2 Challenges Encountered

Despite experimenting with various models, the highest accuracy achieved was not significantly greater than the initial accuracy.

From this chapter onwards, the network architectures will be very complex, and the data will be very large. This creates a training time problem.

This is the first time writing a chapter report, so my writing skills are still lacking, and it takes time to learn each part.

### 5.3 Opening Questions

- What strategies can be used to identify the optimal architecture and hyperparameters for a specific problem?
- What techniques can be applied to further enhance model accuracy?
- What are the best practices for managing and cleaning dirty data in real-world scenarios?

## 6 Conclusion and Furture work

Through this chapter, I have gained a comprehensive understanding of Convolutional Neural Networks (CNNs). I have learned how to utilize Python to implement CNNs for image classification tasks. Compared to neural networks consisting solely of fully connected layers, CNNs demonstrated significantly improved performance.

In subsequent chapters, I aim to delve into more advanced models, such as recurrent neural networks and transformers.

## References

[1] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[2] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, Richard Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2, 1989.

[3] Raschka Sebastian, (Hayden) Liu Yuxi, and Mirjalili Vahid. *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, Birmingham, UK, 2022.

[4] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.