Deep Learning & Neural Networks

A Comprehensive Guide to Modern Deep Learning

Contents

1 Introduction to Deep Learning **4**
1.1 Why Deep Learning? 4

2 Artificial Neural Networks (ANNs) **4**
2.1 The Perceptron 4
2.1.1 Mathematical Formulation 4
2.2 Activation Functions 4
2.2.1 Common Activation Functions 4
2.2.2 Python Implementation 5
2.3 Multi-Layer Perceptron (MLP) 5
2.3.1 Forward Propagation 5
2.3.2 Loss Functions 6
2.4 Backpropagation 6
2.4.1 Gradient Computation 6
2.4.2 Python Implementation from Scratch 6
2.4.3 Using TensorFlow/Keras 9

3 Optimization Algorithms **10**
3.1 Gradient Descent Variants 10
3.1.1 Batch Gradient Descent 10
3.1.2 Stochastic Gradient Descent (SGD) 10
3.1.3 Mini-Batch Gradient Descent 10
3.2 Advanced Optimizers 10
3.2.1 Momentum 10
3.2.2 RMSprop 10
3.2.3 Adam (Adaptive Moment Estimation) 11

4 Regularization Techniques **11**
4.1 L2 Regularization (Weight Decay) 11
4.2 Dropout 12
4.3 Batch Normalization 12

5 Convolutional Neural Networks (CNNs) **12**
5.1 Convolutional Layer 12
5.1.1 Convolution Operation 12
5.1.2 Properties of Convolution 13
5.2 Pooling Layer 13
5.3 CNN Architecture 13
5.3.1 Python Implementation 13
5.4 Famous CNN Architectures 14
5.4.1 LeNet-5 (1998) 14

5.4.2 AlexNet (2012) . . . 14
5.4.3 VGGNet (2014) . . . 15
5.4.4 ResNet (2015) . . . 15

6 Recurrent Neural Networks (RNNs) 16
6.1 Basic RNN . . . 16
6.1.1 Mathematical Formulation . . . 16
6.1.2 Backpropagation Through Time (BPTT) . . . 16
6.2 Long Short-Term Memory (LSTM) . . . 16
6.2.1 LSTM Equations . . . 16
6.3 Gated Recurrent Unit (GRU) . . . 17
6.3.1 GRU Equations . . . 17
6.3.2 Python Implementation . . . 17
6.4 Bidirectional RNNs . . . 18

7 Attention Mechanisms 18
7.1 Attention Score . . . 19
7.2 Multi-Head Attention . . . 19

8 Transformers 20
8.1 Transformer Architecture . . . 20

9 Autoencoders 21
9.1 Vanilla Autoencoder . . . 21
9.2 Variational Autoencoder (VAE) . . . 21

10 Generative Adversarial Networks (GANs) 24
10.1 GAN Objective . . . 24

11 Transfer Learning 26
11.1 Approaches . . . 26

12 Best Practices & Tips 27
12.1 Training Strategies . . . 27
12.2 Model Evaluation . . . 28
12.3 Hyperparameter Tuning . . . 29
12.4 Debugging Neural Networks . . . 30

13 Advanced Topics 30
13.1 Neural Architecture Search (NAS) . . . 30
13.2 Meta-Learning . . . 31
13.3 Neural ODEs . . . 31
13.4 Graph Neural Networks (GNNs) . . . 31

14 Practical Projects 31
14.1 Image Classification . . . 31
14.2 Text Generation with RNN . . . 32

15 Resources & Further Reading 34
15.1 Books . . . 34
15.2 Online Courses . . . 34
15.3 Research Papers . . . 34

16 Conclusion **34**

1 Introduction to Deep Learning

Deep learning is a subset of machine learning based on artificial neural networks with multiple layers. These networks can learn hierarchical representations of data, making them particularly effective for complex tasks like image recognition, natural language processing, and game playing.

1.1 Why Deep Learning?

- **Feature Learning**: Automatically learns features from raw data
- **Scalability**: Performance improves with more data
- **End-to-End Learning**: Can optimize entire pipelines jointly
- **Transfer Learning**: Pre-trained models can be adapted to new tasks

2 Artificial Neural Networks (ANNs)

2.1 The Perceptron

The perceptron is the simplest neural network unit, inspired by biological neurons.

2.1.1 Mathematical Formulation

Given input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ and weights $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$:

$$z = \mathbf{w}^T\mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b \tag{1}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{2}$$

2.2 Activation Functions

Activation functions introduce non-linearity, enabling networks to learn complex patterns.

2.2.1 Common Activation Functions

1. Sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z)) \tag{3}$$

2. Hyperbolic Tangent (tanh):

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \tanh'(z) = 1 - \tanh^2(z) \tag{4}$$

3. ReLU (Rectified Linear Unit):

$$\text{ReLU}(z) = \max(0, z), \quad \text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{5}$$

4. Leaky ReLU:

$$\text{LeakyReLU}(z) = \max(\alpha z, z), \quad \alpha \approx 0.01 \tag{6}$$

5. Softmax (for output layer):

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{7}$$

2.2.2 Python Implementation

```python
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def tanh(z):
    return np.tanh(z)

def relu(z):
    return np.maximum(0, z)

def leaky_relu(z, alpha=0.01):
    return np.where(z > 0, z, alpha * z)

# Visualization
z = np.linspace(-5, 5, 100)
plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)
plt.plot(z, sigmoid(z))
plt.title('Sigmoid')
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(z, tanh(z))
plt.title('Tanh')
plt.grid(True)

plt.subplot(2, 2, 3)
plt.plot(z, relu(z))
plt.title('ReLU')
plt.grid(True)

plt.subplot(2, 2, 4)
plt.plot(z, leaky_relu(z))
plt.title('Leaky ReLU')
plt.grid(True)

plt.tight_layout()
plt.show()
```

2.3 Multi-Layer Perceptron (MLP)

An MLP consists of an input layer, one or more hidden layers, and an output layer.

2.3.1 Forward Propagation

For a network with L layers:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \tag{8}$$

$$\mathbf{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]}) \tag{9}$$

where:

- $\mathbf{z}^{[l]}$ is the weighted sum at layer l
- $\mathbf{a}^{[l]}$ is the activation at layer l
- $g^{[l]}$ is the activation function
- $\mathbf{W}^{[l]}$ is the weight matrix
- $\mathbf{b}^{[l]}$ is the bias vector

2.3.2 Loss Functions

Mean Squared Error (Regression):

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{10}$$

Cross-Entropy Loss (Classification):

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik}) \tag{11}$$

2.4 Backpropagation

Backpropagation computes gradients using the chain rule.

2.4.1 Gradient Computation

For layer l:

$$\delta^{[l]} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[l]}} \odot g'^{[l]}(\mathbf{z}^{[l]}) \tag{12}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \delta^{[l]}(\mathbf{a}^{[l-1]})^T \tag{13}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]} \tag{14}$$

Backpropagation recursion:

$$\delta^{[l]} = (\mathbf{W}^{[l+1]})^T\delta^{[l+1]} \odot g'^{[l]}(\mathbf{z}^{[l]}) \tag{15}$$

where $\odot$ denotes element-wise multiplication.

2.4.2 Python Implementation from Scratch

```python
import numpy as np

class NeuralNetwork:
    def __init__(self, layer_dims):
        """
        layer_dims: list containing dimensions of each layer
        Example: [784, 128, 64, 10] for MNIST
        """
        self.parameters = {}
        self.L = len(layer_dims) - 1  # number of layers

```

```
        # Initialize weights and biases
        for l in range(1, self.L + 1):
            self.parameters[f'W{l}'] = np.random.randn(
                layer_dims[l], layer_dims[l-1]
            ) * 0.01
            self.parameters[f'b{l}'] = np.zeros((layer_dims[l], 1))

    def relu(self, Z):
        return np.maximum(0, Z)

    def relu_derivative(self, Z):
        return (Z > 0).astype(float)

    def softmax(self, Z):
        exp_Z = np.exp(Z - np.max(Z, axis=0, keepdims=True))
        return exp_Z / np.sum(exp_Z, axis=0, keepdims=True)

    def forward_propagation(self, X):
        cache = {'A0': X}
        A = X

        # Hidden layers with ReLU
        for l in range(1, self.L):
            W = self.parameters[f'W{l}']
            b = self.parameters[f'b{l}']
            Z = np.dot(W, A) + b
            A = self.relu(Z)
            cache[f'Z{l}'] = Z
            cache[f'A{l}'] = A

        # Output layer with softmax
        W = self.parameters[f'W{self.L}']
        b = self.parameters[f'b{self.L}']
        Z = np.dot(W, A) + b
        A = self.softmax(Z)
        cache[f'Z{self.L}'] = Z
        cache[f'A{self.L}'] = A

        return A, cache

    def compute_cost(self, AL, Y):
        m = Y.shape[1]
        cost = -np.sum(Y * np.log(AL + 1e-8)) / m
        return cost

    def backward_propagation(self, cache, Y):
        gradients = {}
        m = Y.shape[1]

        # Output layer gradient
        dZ = cache[f'A{self.L}'] - Y
        gradients[f'dW{self.L}'] = np.dot(dZ, cache[f'A{self.L-1}'].T)
            / m
        gradients[f'db{self.L}'] = np.sum(dZ, axis=1, keepdims=True) /
            m

        # Hidden layers gradients
        for l in reversed(range(1, self.L)):
```

```
            dA = np.dot(self.parameters[f'W{l+1}'].T, dZ)
            dZ = dA * self.relu_derivative(cache[f'Z{l}'])
            gradients[f'dW{l}'] = np.dot(dZ, cache[f'A{l-1}'].T) / m
            gradients[f'db{l}'] = np.sum(dZ, axis=1, keepdims=True) / m

        return gradients

    def update_parameters(self, gradients, learning_rate):
        for l in range(1, self.L + 1):
            self.parameters[f'W{l}'] -= learning_rate * gradients[f'dW{l}']
            self.parameters[f'b{l}'] -= learning_rate * gradients[f'db{l}']

    def train(self, X, Y, epochs, learning_rate):
        costs = []
        for epoch in range(epochs):
            # Forward propagation
            AL, cache = self.forward_propagation(X)

            # Compute cost
            cost = self.compute_cost(AL, Y)
            costs.append(cost)

            # Backward propagation
            gradients = self.backward_propagation(cache, Y)

            # Update parameters
            self.update_parameters(gradients, learning_rate)

            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Cost: {cost:.4f}")

        return costs

    def predict(self, X):
        AL, _ = self.forward_propagation(X)
        return np.argmax(AL, axis=0)

# Example usage with MNIST-like data
# Generate dummy data
np.random.seed(42)
X_train = np.random.randn(784, 1000)  # 1000 samples, 784 features
Y_train = np.zeros((10, 1000))  # 10 classes
Y_train[np.random.randint(0, 10, 1000), np.arange(1000)] = 1

# Create and train network
nn = NeuralNetwork([784, 128, 64, 10])
costs = nn.train(X_train, Y_train, epochs=1000, learning_rate=0.01)

# Plot learning curve
plt.plot(costs)
plt.xlabel('Epoch')
plt.ylabel('Cost')
plt.title('Learning Curve')
plt.grid(True)
plt.show()
```

2.4.3 Using TensorFlow/Keras

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load data
digits = load_digits()
X, y = digits.data, digits.target

# Preprocess
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# Build model
model = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(64,)),
    layers.Dropout(0.3),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(10, activation='softmax')
])

# Compile
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Train
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.2,
    verbose=1
)

# Evaluate
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")

# Plot training history
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
```

```
57
58 plt.subplot(1, 2, 2)
59 plt.plot(history.history['accuracy'], label='Training Accuracy')
60 plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
61 plt.xlabel('Epoch')
62 plt.ylabel('Accuracy')
63 plt.legend()
64 plt.grid(True)
65 plt.tight_layout()
66 plt.show()
```

3 Optimization Algorithms

3.1 Gradient Descent Variants

3.1.1 Batch Gradient Descent

Updates parameters using the entire dataset:

$$\mathbf{W} := \mathbf{W} - \alpha \nabla_{\mathbf{W}} \mathcal{L} \tag{16}$$

3.1.2 Stochastic Gradient Descent (SGD)

Updates parameters using one sample at a time:

$$\mathbf{W} := \mathbf{W} - \alpha \nabla_{\mathbf{W}} \mathcal{L}^{(i)} \tag{17}$$

3.1.3 Mini-Batch Gradient Descent

Updates using small batches of size m:

$$\mathbf{W} := \mathbf{W} - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla_{\mathbf{W}} \mathcal{L}^{(i)} \tag{18}$$

3.2 Advanced Optimizers

3.2.1 Momentum

Accelerates gradient descent by accumulating velocity:

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1 - \beta) \nabla_{\mathbf{W}} \mathcal{L} \tag{19}$$

$$\mathbf{W} := \mathbf{W} - \alpha \mathbf{v}_t \tag{20}$$

Typical: $\beta = 0.9$

3.2.2 RMSprop

Adapts learning rate based on gradient magnitude:

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1 - \beta)(\nabla_{\mathbf{W}} \mathcal{L})^2 \tag{21}$$

$$\mathbf{W} := \mathbf{W} - \frac{\alpha}{\sqrt{\mathbf{s}_t + \epsilon}} \nabla_{\mathbf{W}} \mathcal{L} \tag{22}$$

3.2.3 Adam (Adaptive Moment Estimation)

Combines momentum and RMSprop:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla_{\mathbf{W}}\mathcal{L} \tag{23}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)(\nabla_{\mathbf{W}}\mathcal{L})^2 \tag{24}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \tag{25}$$

$$\mathbf{W} := \mathbf{W} - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t \tag{26}$$

Typical: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

```python
# Comparing optimizers in Keras
optimizers_list = [
    ('SGD', keras.optimizers.SGD(learning_rate=0.01)),
    ('Momentum', keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
        ),
    ('RMSprop', keras.optimizers.RMSprop(learning_rate=0.001)),
    ('Adam', keras.optimizers.Adam(learning_rate=0.001))
]

histories = {}
for name, optimizer in optimizers_list:
    model = keras.Sequential([
        layers.Dense(128, activation='relu', input_shape=(64,)),
        layers.Dense(64, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])

    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    history = model.fit(X_train, y_train, epochs=30,
                        batch_size=32, validation_split=0.2,
                        verbose=0)
    histories[name] = history

# Plot comparison
plt.figure(figsize=(12, 4))
for name, history in histories.items():
    plt.plot(history.history['val_accuracy'], label=name)
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.title('Optimizer Comparison')
plt.legend()
plt.grid(True)
plt.show()
```

4 Regularization Techniques

4.1 L2 Regularization (Weight Decay)

Add penalty to loss function:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2}\sum_{l=1}^{L}\|\mathbf{W}^{[l]}\|_F^2 \tag{27}$$

4.2 Dropout

Randomly deactivate neurons during training with probability p:

$$\mathbf{a}^{[l]} = \mathbf{a}^{[l]} \odot \mathbf{d}^{[l]} \tag{28}$$

where $\mathbf{d}^{[l]} \sim \text{Bernoulli}(1-p)$
During inference: $\mathbf{a}^{[l]} = (1-p)\mathbf{a}^{[l]}$

4.3 Batch Normalization

Normalize activations within mini-batch:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} z_i \tag{29}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(z_i - \mu_{\mathcal{B}})^2 \tag{30}$$

$$\hat{z}_i = \frac{z_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{31}$$

$$\tilde{z}_i = \gamma \hat{z}_i + \beta \tag{32}$$

where γ and β are learnable parameters.

```
# Regularization in Keras
model = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(64,),
                 kernel_regularizer=keras.regularizers.l2(0.01)),
    layers.BatchNormalization(),
    layers.Dropout(0.5),

    layers.Dense(64, activation='relu',
                 kernel_regularizer=keras.regularizers.l2(0.01)),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    layers.Dense(10, activation='softmax')
])
```

5 Convolutional Neural Networks (CNNs)

CNNs are specialized for processing grid-like data, particularly images.

5.1 Convolutional Layer

5.1.1 Convolution Operation

For input $\mathbf{I}$ and kernel $\mathbf{K}$:

$$(\mathbf{I} * \mathbf{K})(i,j) = \sum_m \sum_n \mathbf{I}(i+m, j+n)\mathbf{K}(m,n) \tag{33}$$

Output dimensions:

$$\text{Output size} = \left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1 \tag{34}$$

where:

- n: input size
- p: padding
- f: filter size
- s: stride

5.1.2 Properties of Convolution

- **Parameter Sharing**: Same filter applied across entire input
- **Translation Invariance**: Detects features regardless of position
- **Local Connectivity**: Each neuron connects to local region

5.2 Pooling Layer

Reduces spatial dimensions while retaining important features.

Max Pooling:

$$y_{i,j} = \max_{m,n \in \mathcal{R}} x_{i+m,j+n} \tag{35}$$

Average Pooling:

$$y_{i,j} = \frac{1}{|\mathcal{R}|} \sum_{m,n \in \mathcal{R}} x_{i+m,j+n} \tag{36}$$

5.3 CNN Architecture

Typical architecture:

$$\text{INPUT} \to [\text{CONV} \to \text{RELU} \to \text{POOL}]^* \to \text{FC} \to \text{SOFTMAX} \tag{37}$$

5.3.1 Python Implementation

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

# Load MNIST dataset
(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

# Build CNN
model = models.Sequential([
    # First convolutional block
    layers.Conv2D(32, (3, 3), activation='relu',
                  input_shape=(28, 28, 1), padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Second convolutional block
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Third convolutional block
```

```
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Flatten and fully connected layers
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

# Model summary
model.summary()

# Compile
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Train
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    validation_split=0.1,
    verbose=1
)

# Evaluate
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")

# Visualize filters
first_layer_weights = model.layers[0].get_weights()[0]
fig, axes = plt.subplots(4, 8, figsize=(12, 6))
for i, ax in enumerate(axes.flat):
    if i < 32:
        ax.imshow(first_layer_weights[:, :, 0, i], cmap='gray')
        ax.axis('off')
plt.suptitle('First Layer Filters')
plt.tight_layout()
plt.show()
```

5.4 Famous CNN Architectures

5.4.1 LeNet-5 (1998)

One of the earliest CNNs for digit recognition.

5.4.2 AlexNet (2012)

- 5 convolutional layers
- 3 fully connected layers
- ReLU activation

- Dropout regularization
- 60M parameters

5.4.3 VGGNet (2014)

- Very deep (16-19 layers)
- Small 3×3 filters
- Uniform architecture
- 138M parameters

5.4.4 ResNet (2015)

Introduced skip connections to enable very deep networks:

$$\mathbf{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x} \tag{38}$$

```
# ResNet block implementation
def residual_block(x, filters, kernel_size=3):
    # Save input
    shortcut = x

    # First conv layer
    x = layers.Conv2D(filters, kernel_size, padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # Second conv layer
    x = layers.Conv2D(filters, kernel_size, padding='same')(x)
    x = layers.BatchNormalization()(x)

    # Add shortcut
    x = layers.Add()([x, shortcut])
    x = layers.Activation('relu')(x)

    return x

# Build mini ResNet
inputs = layers.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, padding='same')(inputs)
x = layers.BatchNormalization()(x)
x = layers.Activation('relu')(x)

# Add residual blocks
x = residual_block(x, 32)
x = residual_block(x, 32)

x = layers.MaxPooling2D(2)(x)
x = layers.Flatten()(x)
x = layers.Dense(128, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

resnet_model = models.Model(inputs=inputs, outputs=outputs)
resnet_model.summary()
```

6 Recurrent Neural Networks (RNNs)

RNNs process sequential data by maintaining hidden states.

6.1 Basic RNN

6.1.1 Mathematical Formulation

At time step t:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{39}$$

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \tag{40}$$

where:

- $\mathbf{x}_t$: input at time t
- $\mathbf{h}_t$: hidden state at time t
- $\mathbf{y}_t$: output at time t
- $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$, $\mathbf{W}_{hy}$: weight matrices

6.1.2 Backpropagation Through Time (BPTT)

Gradients flow backward through time:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{W}} \tag{41}$$

Problem: Vanishing/exploding gradients

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \tag{42}$$

6.2 Long Short-Term Memory (LSTM)

LSTMs solve vanishing gradient problem using gating mechanisms.

6.2.1 LSTM Equations

Forget Gate:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \tag{43}$$

Input Gate:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \tag{44}$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \tag{45}$$

Cell State Update:

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \tag{46}$$

Output Gate:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \tag{47}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \tag{48}$$

6.3 Gated Recurrent Unit (GRU)

Simplified version of LSTM with fewer parameters.

6.3.1 GRU Equations

Reset Gate:

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t]) \tag{49}$$

Update Gate:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t]) \tag{50}$$

Candidate Hidden State:

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t]) \tag{51}$$

Hidden State:

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \tag{52}$$

6.3.2 Python Implementation

```python
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# Generate synthetic sequence data
def generate_sequence(length=50):
    X = np.random.randn(1000, length, 1)
    # Predict sum of sequence
    y = np.sum(X, axis=1)
    return X, y

X_train, y_train = generate_sequence()
X_test, y_test = generate_sequence()

# LSTM Model
lstm_model = models.Sequential([
    layers.LSTM(64, return_sequences=True, input_shape=(50, 1)),
    layers.Dropout(0.2),
    layers.LSTM(32),
    layers.Dropout(0.2),
    layers.Dense(1)
])

lstm_model.compile(optimizer='adam', loss='mse', metrics=['mae'])
lstm_model.summary()

# Train
history_lstm = lstm_model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    verbose=1
)

# GRU Model for comparison
gru_model = models.Sequential([
```

```
    layers.GRU(64, return_sequences=True, input_shape=(50, 1)),
    layers.Dropout(0.2),
    layers.GRU(32),
    layers.Dropout(0.2),
    layers.Dense(1)
])

gru_model.compile(optimizer='adam', loss='mse', metrics=['mae'])

history_gru = gru_model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    verbose=0
)

# Compare
plt.figure(figsize=(10, 5))
plt.plot(history_lstm.history['loss'], label='LSTM Train')
plt.plot(history_lstm.history['val_loss'], label='LSTM Val')
plt.plot(history_gru.history['loss'], label='GRU Train')
plt.plot(history_gru.history['val_loss'], label='GRU Val')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('LSTM vs GRU')
plt.legend()
plt.grid(True)
plt.show()
```

6.4 Bidirectional RNNs

Process sequences in both forward and backward directions:

$$\overrightarrow{\mathbf{h}}_t = \text{RNN}(\mathbf{x}_t, \overrightarrow{\mathbf{h}}_{t-1}) \tag{53}$$

$$\overleftarrow{\mathbf{h}}_t = \text{RNN}(\mathbf{x}_t, \overleftarrow{\mathbf{h}}_{t+1}) \tag{54}$$

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] \tag{55}$$

```
# Bidirectional LSTM
bi_model = models.Sequential([
    layers.Bidirectional(layers.LSTM(64, return_sequences=True),
                         input_shape=(50, 1)),
    layers.Bidirectional(layers.LSTM(32)),
    layers.Dense(1)
])

bi_model.compile(optimizer='adam', loss='mse')
bi_model.summary()
```

7 Attention Mechanisms

Attention allows models to focus on relevant parts of the input.

7.1 Attention Score

Scaled Dot-Product Attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{56}$$

where:

- $\mathbf{Q}$: Query matrix
- $\mathbf{K}$: Key matrix
- $\mathbf{V}$: Value matrix
- d_k: dimension of keys

7.2 Multi-Head Attention

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{57}$$

where:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{58}$$

```
# Simple attention layer implementation
class AttentionLayer(layers.Layer):
    def __init__(self, units):
        super(AttentionLayer, self).__init__()
        self.W1 = layers.Dense(units)
        self.W2 = layers.Dense(units)
        self.V = layers.Dense(1)

    def call(self, query, values):
        # query: (batch, hidden)
        # values: (batch, seq_len, hidden)

        # Expand query to match values shape
        query_with_time = tf.expand_dims(query, 1)

        # Compute attention scores
        score = self.V(tf.nn.tanh(
            self.W1(query_with_time) + self.W2(values)
        ))

        # Compute attention weights
        attention_weights = tf.nn.softmax(score, axis=1)

        # Compute context vector
        context = attention_weights * values
        context = tf.reduce_sum(context, axis=1)

        return context, attention_weights

# Example usage in sequence model
inputs = layers.Input(shape=(50, 1))
lstm_out = layers.LSTM(64, return_sequences=True)(inputs)
query = layers.LSTM(64)(inputs)
context, weights = AttentionLayer(64)(query, lstm_out)
output = layers.Dense(1)(context)
```

```

attention_model = models.Model(inputs=inputs, outputs=output)
attention_model.compile(optimizer='adam', loss='mse')
```

8 Transformers

Transformers rely entirely on attention mechanisms, eliminating recurrence.

8.1 Transformer Architecture

Positional Encoding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{59}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{60}$$

Encoder Block:

$$\mathbf{Z} = \text{MultiHeadAttention}(\mathbf{X}, \mathbf{X}, \mathbf{X}) \tag{61}$$

$$\mathbf{Z}' = \text{LayerNorm}(\mathbf{X} + \mathbf{Z}) \tag{62}$$

$$\mathbf{Y} = \text{FFN}(\mathbf{Z}') \tag{63}$$

$$\mathbf{Y}' = \text{LayerNorm}(\mathbf{Z}' + \mathbf{Y}) \tag{64}$$

```
# Simplified Transformer implementation
def positional_encoding(length, depth):
    positions = np.arange(length)[:, np.newaxis]
    depths = np.arange(depth)[np.newaxis, :] / depth

    angle_rates = 1 / (10000**depths)
    angle_rads = positions * angle_rates

    pos_encoding = np.zeros(angle_rads.shape)
    pos_encoding[:, 0::2] = np.sin(angle_rads[:, 0::2])
    pos_encoding[:, 1::2] = np.cos(angle_rads[:, 1::2])

    return tf.cast(pos_encoding, dtype=tf.float32)

class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim):
        super(TransformerBlock, self).__init__()
        self.att = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.ffn = keras.Sequential([
            layers.Dense(ff_dim, activation='relu'),
            layers.Dense(embed_dim)
        ])
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(0.1)
        self.dropout2 = layers.Dropout(0.1)

    def call(self, inputs, training):
        attn_output = self.att(inputs, inputs)
```

```
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)

        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

# Build transformer model
vocab_size = 10000
maxlen = 100
embed_dim = 128
num_heads = 8
ff_dim = 512

inputs = layers.Input(shape=(maxlen,))
embedding_layer = layers.Embedding(vocab_size, embed_dim)(inputs)
positions = positional_encoding(maxlen, embed_dim)
x = embedding_layer + positions

transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.1)(x)
x = layers.Dense(20, activation='relu')(x)
outputs = layers.Dense(2, activation='softmax')(x)

transformer_model = models.Model(inputs=inputs, outputs=outputs)
transformer_model.summary()
```

9 Autoencoders

Autoencoders learn compressed representations of data.

9.1 Vanilla Autoencoder

Encoder:

$$\mathbf{z} = f_{\text{enc}}(\mathbf{x}; \boldsymbol{\theta}_{\text{enc}}) \tag{65}$$

Decoder:

$$\hat{\mathbf{x}} = f_{\text{dec}}(\mathbf{z}; \boldsymbol{\theta}_{\text{dec}}) \tag{66}$$

Loss:

$$\mathcal{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 \tag{67}$$

9.2 Variational Autoencoder (VAE)

VAE learns probabilistic latent representations.

Encoder produces distribution parameters:

$$\boldsymbol{\mu} = f_{\mu}(\mathbf{x}) \tag{68}$$

$$\log \boldsymbol{\sigma}^2 = f_{\sigma}(\mathbf{x}) \tag{69}$$

Sampling (reparameterization trick):

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}) \tag{70}$$

Loss (ELBO):

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{x}|\mathbf{z})] - D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})) \tag{71}$$

```
# Variational Autoencoder
latent_dim = 32

# Encoder
encoder_inputs = layers.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation='relu', strides=2, padding='same')(
    encoder_inputs)
x = layers.Conv2D(64, 3, activation='relu', strides=2, padding='same')(
    x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation='relu')(x)

z_mean = layers.Dense(latent_dim, name='z_mean')(x)
z_log_var = layers.Dense(latent_dim, name='z_log_var')(x)

# Sampling layer
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

z = Sampling()([z_mean, z_log_var])

encoder = models.Model(encoder_inputs, [z_mean, z_log_var, z], name='
    encoder')

# Decoder
latent_inputs = layers.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation='relu')(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation='relu', strides=2, padding
    ='same')(x)
x = layers.Conv2DTranspose(32, 3, activation='relu', strides=2, padding
    ='same')(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation='sigmoid',
    padding='same')(x)

decoder = models.Model(latent_inputs, decoder_outputs, name='decoder')

# VAE model
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = keras.metrics.Mean(name='total_loss')
        self.reconstruction_loss_tracker = keras.metrics.Mean(name='
            reconstruction_loss')
        self.kl_loss_tracker = keras.metrics.Mean(name='kl_loss')

    @property
    def metrics(self):
```

```
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker
        ]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)

            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data,
                       reconstruction),
                    axis=(1, 2)
                )
            )

            kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.
               exp(z_log_var))
            kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))

            total_loss = reconstruction_loss + kl_loss

        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.
           trainable_weights))

        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(
           reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)

        return {
            'loss': self.total_loss_tracker.result(),
            'reconstruction_loss': self.reconstruction_loss_tracker.
               result(),
            'kl_loss': self.kl_loss_tracker.result()
        }

vae = VAE(encoder, decoder)
vae.compile(optimizer='adam')

# Train on MNIST
(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
x_train = np.expand_dims(x_train, -1).astype('float32') / 255.0
x_test = np.expand_dims(x_test, -1).astype('float32') / 255.0

vae.fit(x_train, epochs=30, batch_size=128)

# Generate new images
n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-3, 3, n)
grid_y = np.linspace(-3, 3, n)[::-1]
```

```

for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi] + [0] * (latent_dim - 2)])
        x_decoded = decoder.predict(z_sample, verbose=0)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.title('VAE Generated Digits')
plt.axis('off')
plt.show()
```

10 Generative Adversarial Networks (GANs)

GANs consist of two networks competing in a minimax game.

10.1 GAN Objective

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))] \tag{72}$$

where:

- G: Generator network
- D: Discriminator network
- $\mathbf{z}$: Random noise vector

```
# Simple GAN implementation
latent_dim = 100

# Generator
generator = models.Sequential([
    layers.Dense(7 * 7 * 128, input_dim=latent_dim),
    layers.Reshape((7, 7, 128)),
    layers.BatchNormalization(),
    layers.LeakyReLU(alpha=0.2),

    layers.Conv2DTranspose(128, 4, strides=2, padding='same'),
    layers.BatchNormalization(),
    layers.LeakyReLU(alpha=0.2),

    layers.Conv2DTranspose(64, 4, strides=2, padding='same'),
    layers.BatchNormalization(),
    layers.LeakyReLU(alpha=0.2),

    layers.Conv2D(1, 3, padding='same', activation='tanh')
], name='generator')

# Discriminator
discriminator = models.Sequential([
    layers.Conv2D(64, 3, strides=2, padding='same',
                  input_shape=(28, 28, 1)),
    layers.LeakyReLU(alpha=0.2),
```

```
    layers.Dropout(0.3),

    layers.Conv2D(128, 3, strides=2, padding='same'),
    layers.LeakyReLU(alpha=0.2),
    layers.Dropout(0.3),

    layers.Flatten(),
    layers.Dense(1, activation='sigmoid')
], name='discriminator')

# GAN model
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]

        # Generate fake images
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )
        generated_images = self.generator(random_latent_vectors)

        # Combine with real images
        combined_images = tf.concat([generated_images, real_images],
            axis=0)
        labels = tf.concat([
            tf.ones((batch_size, 1)),
            tf.zeros((batch_size, 1))
        ], axis=0)

        # Add noise to labels
        labels += 0.05 * tf.random.uniform(tf.shape(labels))

        # Train discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)

        grads = tape.gradient(d_loss, self.discriminator.
            trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Train generator
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
```

```
        )
        misleading_labels = tf.zeros((batch_size, 1))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(
                self.generator(random_latent_vectors)
            )
            g_loss = self.loss_fn(misleading_labels, predictions)

        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(
            zip(grads, self.generator.trainable_weights)
        )

        return {'d_loss': d_loss, 'g_loss': g_loss}

# Create and compile GAN
gan = GAN(discriminator=discriminator,
          generator=generator,
          latent_dim=latent_dim)

gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    loss_fn=keras.losses.BinaryCrossentropy()
)

# Train
(x_train, _), _ = keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 28, 28, 1).astype('float32')
x_train = (x_train - 127.5) / 127.5  # Normalize to [-1, 1]

gan.fit(x_train, epochs=50, batch_size=128)

# Generate images
noise = tf.random.normal([16, latent_dim])
generated_images = generator(noise, training=False)

fig, axes = plt.subplots(4, 4, figsize=(8, 8))
for i, ax in enumerate(axes.flat):
    ax.imshow(generated_images[i, :, :, 0], cmap='gray')
    ax.axis('off')
plt.suptitle('GAN Generated Images')
plt.tight_layout()
plt.show()
```

11 Transfer Learning

Transfer learning leverages pre-trained models for new tasks.

11.1 Approaches

1. **Feature Extraction**: Freeze pre-trained layers, train new classifier
2. **Fine-Tuning**: Unfreeze some layers and retrain

3. **Full Training**: Use pre-trained weights as initialization

```
# Transfer learning with VGG16
from tensorflow.keras.applications import VGG16
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Load pre-trained VGG16 (without top layers)
base_model = VGG16(
    weights='imagenet',
    include_top=False,
    input_shape=(224, 224, 3)
)

# Freeze base model
base_model.trainable = False

# Add custom classifier
inputs = keras.Input(shape=(224, 224, 3))
x = base_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)

model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

# Fine-tuning: unfreeze last few layers
base_model.trainable = True
for layer in base_model.layers[:-4]:
    layer.trainable = False

model.compile(
    optimizer=keras.optimizers.Adam(1e-5),  # Lower learning rate
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

12 Best Practices & Tips

12.1 Training Strategies

1. **Learning Rate Scheduling**

$$\alpha_t = \alpha_0 \cdot \text{decay\_factor}^{t/\text{decay\_steps}} \tag{73}$$

2. **Early Stopping**: Monitor validation loss, stop if no improvement

3. **Gradient Clipping**: Prevent exploding gradients

$$\mathbf{g} := \min\left(1, \frac{\text{threshold}}{\|\mathbf{g}\|}\right) \cdot \mathbf{g} \tag{74}$$

4. **Data Augmentation**: Artificially expand training data

5. **Batch Size Selection**: Larger batches = faster but less generalization

```python
# Learning rate scheduling
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=5,
    min_lr=1e-7,
    verbose=1
)

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True,
    verbose=1
)

# Data augmentation for images
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    zoom_range=0.1,
    shear_range=0.1,
    fill_mode='nearest'
)

# Training with callbacks
history = model.fit(
    datagen.flow(X_train, y_train, batch_size=32),
    epochs=100,
    validation_data=(X_val, y_val),
    callbacks=[reduce_lr, early_stop]
)
```

12.2 Model Evaluation

Classification Metrics:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{75}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{76}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{77}$$

$$\text{F1-Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{78}$$

```python
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

# Predictions
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Classification report
print(classification_report(y_test, y_pred_classes))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

12.3 Hyperparameter Tuning

```python
from keras_tuner import RandomSearch

def build_model(hp):
    model = keras.Sequential()

    # Tune number of layers
    for i in range(hp.Int('num_layers', 1, 4)):
        model.add(layers.Dense(
            units=hp.Int(f'units_{i}', 32, 512, step=32),
            activation='relu'
        ))
        model.add(layers.Dropout(
            hp.Float(f'dropout_{i}', 0, 0.5, step=0.1)
        ))

    model.add(layers.Dense(10, activation='softmax'))

    # Tune learning rate
    model.compile(
        optimizer=keras.optimizers.Adam(
            hp.Float('learning_rate', 1e-4, 1e-2, sampling='log')
        ),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model

tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=10,
    directory='tuning',
    project_name='mnist'
)
```

```
tuner.search(X_train, y_train, epochs=10, validation_split=0.2)
best_model = tuner.get_best_models(num_models=1)[0]
```

12.4 Debugging Neural Networks

Common Issues:

- **Vanishing Gradients**: Use ReLU, batch normalization, skip connections
- **Exploding Gradients**: Use gradient clipping, lower learning rate
- **Overfitting**: Add dropout, L2 regularization, more data
- **Underfitting**: Increase model capacity, train longer
- **Dead ReLU**: Use Leaky ReLU, lower learning rate

```
# Gradient monitoring
class GradientLogger(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        for layer in self.model.layers:
            if hasattr(layer, 'kernel'):
                weights = layer.get_weights()[0]
                print(f"{layer.name}: mean={np.mean(weights):.4f}, "
                      f"std={np.std(weights):.4f}")

# Check for dead neurons
def check_dead_neurons(model, X_sample):
    activations = {}
    temp_model = keras.Model(
        inputs=model.input,
        outputs=[layer.output for layer in model.layers]
    )

    outputs = temp_model.predict(X_sample)

    for i, output in enumerate(outputs):
        if len(output.shape) == 2:  # Dense layer
            dead = np.sum(output == 0, axis=0) / output.shape[0]
            print(f"Layer {i}: {np.sum(dead > 0.9)} dead neurons "
                  f"({100*np.mean(dead > 0.9):.1f}%)")

check_dead_neurons(model, X_train[:1000])
```

13 Advanced Topics

13.1 Neural Architecture Search (NAS)

Automatically discover optimal architectures using:

- Reinforcement Learning
- Evolutionary Algorithms
- Gradient-based methods (DARTS)

13.2 Meta-Learning

Learning to learn: models that adapt quickly to new tasks with few examples.

MAML (Model-Agnostic Meta-Learning):

$$\boldsymbol{\theta}^* = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{T}_i}(f_{\boldsymbol{\theta}}) \tag{79}$$

13.3 Neural ODEs

Model dynamics as continuous transformations:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \boldsymbol{\theta}) \tag{80}$$

13.4 Graph Neural Networks (GNNs)

Process graph-structured data:

$$\mathbf{h}_v^{(k+1)} = \sigma\left(\mathbf{W}^{(k)}\mathbf{h}_v^{(k)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(k)}\right) \tag{81}$$

14 Practical Projects

14.1 Image Classification

```python
# Complete image classification pipeline
import tensorflow as tf
from tensorflow.keras.applications import EfficientNetB0

# Load and preprocess data
def load_dataset(data_dir):
    train_ds = tf.keras.preprocessing.image_dataset_from_directory(
        data_dir + '/train',
        validation_split=0.2,
        subset='training',
        seed=123,
        image_size=(224, 224),
        batch_size=32
    )

    val_ds = tf.keras.preprocessing.image_dataset_from_directory(
        data_dir + '/train',
        validation_split=0.2,
        subset='validation',
        seed=123,
        image_size=(224, 224),
        batch_size=32
    )

    return train_ds, val_ds

# Build model with transfer learning
def build_classifier(num_classes):
    base_model = EfficientNetB0(
        include_top=False,
        weights='imagenet',
```

```
        input_shape=(224, 224, 3)
    )
    base_model.trainable = False

    inputs = keras.Input(shape=(224, 224, 3))
    x = tf.keras.applications.efficientnet.preprocess_input(inputs)
    x = base_model(x, training=False)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)

    model = keras.Model(inputs, outputs)
    return model

# Train with mixed precision
from tensorflow.keras import mixed_precision
mixed_precision.set_global_policy('mixed_float16')

model = build_classifier(num_classes=10)
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Callbacks
callbacks = [
    keras.callbacks.ModelCheckpoint('best_model.h5', save_best_only=
        True),
    keras.callbacks.ReduceLROnPlateau(patience=3),
    keras.callbacks.EarlyStopping(patience=10),
    keras.callbacks.TensorBoard(log_dir='./logs')
]

# Train
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=50,
    callbacks=callbacks
)
```

14.2 Text Generation with RNN

```
# Character-level text generation
class TextGenerator:
    def __init__(self, text, seq_length=100):
        self.text = text
        self.seq_length = seq_length
        self.chars = sorted(list(set(text)))
        self.char_to_idx = {c: i for i, c in enumerate(self.chars)}
        self.idx_to_char = {i: c for i, c in enumerate(self.chars)}

    def prepare_data(self):
        sequences = []
        next_chars = []

```

```
        for i in range(0, len(self.text) - self.seq_length):
            sequences.append(self.text[i:i + self.seq_length])
            next_chars.append(self.text[i + self.seq_length])

        X = np.zeros((len(sequences), self.seq_length, len(self.chars))
            )
        y = np.zeros((len(sequences), len(self.chars)))

        for i, seq in enumerate(sequences):
            for t, char in enumerate(seq):
                X[i, t, self.char_to_idx[char]] = 1
            y[i, self.char_to_idx[next_chars[i]]] = 1

        return X, y

    def build_model(self):
        model = keras.Sequential([
            layers.LSTM(128, input_shape=(self.seq_length, len(self.
                chars)),
                        return_sequences=True),
            layers.Dropout(0.2),
            layers.LSTM(128),
            layers.Dropout(0.2),
            layers.Dense(len(self.chars), activation='softmax')
        ])

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

        return model

    def generate_text(self, model, seed_text, length=400, temperature
        =1.0):
        generated = seed_text

        for _ in range(length):
            x = np.zeros((1, self.seq_length, len(self.chars)))
            for t, char in enumerate(seed_text):
                x[0, t, self.char_to_idx[char]] = 1

            preds = model.predict(x, verbose=0)[0]
            preds = np.log(preds + 1e-7) / temperature
            exp_preds = np.exp(preds)
            preds = exp_preds / np.sum(exp_preds)

            next_idx = np.random.choice(len(self.chars), p=preds)
            next_char = self.idx_to_char[next_idx]

            generated += next_char
            seed_text = seed_text[1:] + next_char

        return generated

# Usage
with open('shakespeare.txt', 'r') as f:
```

```
    text = f.read().lower()

gen = TextGenerator(text, seq_length=100)
X, y = gen.prepare_data()
model = gen.build_model()
model.fit(X, y, batch_size=128, epochs=50)

# Generate text
seed = text[1000:1100]
generated = gen.generate_text(model, seed, length=500, temperature=0.5)
print(generated)
```

15 Resources & Further Reading

15.1 Books

- Deep Learning by Goodfellow, Bengio, and Courville
- Neural Networks and Deep Learning by Michael Nielsen
- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

15.2 Online Courses

- Stanford CS231n: Convolutional Neural Networks
- Stanford CS224n: Natural Language Processing
- Fast.ai Practical Deep Learning
- DeepLearning.AI Specialization

15.3 Research Papers

- ImageNet Classification with Deep CNNs (AlexNet)
- Deep Residual Learning (ResNet)
- Attention Is All You Need (Transformer)
- Generative Adversarial Networks (GANs)
- Auto-Encoding Variational Bayes (VAE)

16 Conclusion

Deep learning has revolutionized artificial intelligence, enabling breakthroughs in computer vision, natural language processing, and beyond. Key takeaways:

1. **Architecture matters**: Choose appropriate architectures (CNNs for images, RNNs/-Transformers for sequences)
2. **Regularization is crucial**: Use dropout, batch normalization, and data augmentation
3. **Optimization techniques**: Adam is often a good default, but experiment with others
4. **Transfer learning**: Leverage pre-trained models when possible

5. **Hyperparameter tuning**: Systematic search can significantly improve performance

6. **Monitor training**: Use visualization and callbacks to detect issues early

The field continues to evolve rapidly with new architectures, training techniques, and applications emerging constantly. Stay curious, experiment, and keep learning!