Introduction to ANNs and Limitations in Image Processing

Course:
Deep Learning with Tensorflow & Kersa 2

Developed by:
Mohammad Noorchenarboo

May 6, 2025

Current Section

Biological Inspiration for ANNs

Artificial Neural Networks (ANNs) are inspired by how the human brain processes information. Just like biological neurons communicate through synapses, artificial neurons use mathematical operations to transmit and transform signals.

Key Analogies

- **Biological neuron** $\rightarrow$ Artificial neuron
- **Synapse weight** $\rightarrow$ Numeric weight parameter
- **Neuron firing** $\rightarrow$ Activation function
- **Brain learning** $\rightarrow$ Weight adjustment via optimization

What is an ANN?

An **Artificial Neural Network (ANN)** is a computational model made up of layers of interconnected neurons. Each neuron applies a mathematical transformation to its input and passes it to the next layer.

Basic Properties

- ANN learns to map input features to desired outputs
- Each connection has a weight and bias that is learned
- Layers apply nonlinear transformations
- Final layer generates predictions

ANN Structure: Input, Hidden, Output Layers

ANNs are organized in layers:

- **Input Layer:** Accepts raw data (e.g., pixel values, text vectors)
- **Hidden Layers:** Transform inputs through weighted sums and activations
- **Output Layer:** Produces final prediction (e.g., probability)

Example: 3-4-1 Network

- Input: 3 features (e.g., height, weight, age)
- Hidden Layer: 4 neurons (ReLU)
- Output Layer: 1 neuron (Sigmoid)

Total parameters:

$$\text{Weights: } (3 \times 4) + (4 \times 1) = 16$$

Biases: 4 + 1 = 5

$\Rightarrow$ **21 trainable parameters**

Current Section

How Does a Neuron Compute Output?

Each artificial neuron receives inputs, applies weights and a bias, then passes the result through an activation function:

Neuron Formula

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

$$a = \sigma(z)$$

Where:

- $\mathbf{x} \in \mathbb{R}^n$: Input vector
- $\mathbf{w} \in \mathbb{R}^n$: Weight vector
- $b \in \mathbb{R}$: Bias term
- $z \in \mathbb{R}$: Linear combination
- $\sigma(\cdot)$: Activation function

Numeric Example: Single Neuron

Let's compute the output of one neuron with:

- $\mathbf{x} = [2, -1, 3]$
- $\mathbf{w} = [0.5, -0.6, 0.8]$
- $b = 0.2$
- $\sigma(z) = \text{ReLU}(z) = \max(0, z)$

Step-by-Step Computation

$$z = (0.5)(2) + (-0.6)(-1) + (0.8)(3) + 0.2 = 1.0 + 0.6 + 2.4 + 0.2 = 4.2$$

$$a = \text{ReLU}(4.2) = 4.2$$

Vectorized Form: Multiple Neurons

For a layer of multiple neurons, we use matrix operations:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{b} \quad \Rightarrow \quad \mathbf{A} = \sigma(\mathbf{Z})$$

Where:

- $\mathbf{X} \in \mathbb{R}^{m \times n}$: Inputs for m samples, n features
- $\mathbf{W} \in \mathbb{R}^{n \times h}$: Weights for h neurons
- $\mathbf{b} \in \mathbb{R}^{1 \times h}$: Bias for each neuron
- $\mathbf{Z} \in \mathbb{R}^{m \times h}$, $\mathbf{A} \in \mathbb{R}^{m \times h}$: Outputs before and after activation

Current Section

Understanding ANN Architecture

Artificial Neural Networks are composed of **layers**, each responsible for transforming input data:

- **Input Layer:** Receives raw input features (e.g., pixel values)
- **Hidden Layers:** Perform transformations using weighted sums and activation functions
- **Output Layer:** Generates final predictions

Key Point

Each layer learns a feature representation that improves performance on the target task.

Types of Layers in Feedforward ANNs

1. **Dense (Fully Connected) Layer**
   - Each neuron receives input from every neuron in the previous layer
   - Expressive but computationally expensive
2. **Output Layer: Task-specific behavior**
   - **Binary Classification:** 1 neuron with sigmoid
   - **Multi-class Classification:** K neurons with softmax
   - **Regression:** 1 neuron with linear activation

ANNs Are Task-Agnostic

The architecture of an ANN can be reused across tasks. What defines the learning objective is the configuration of the final layer (activation + loss function).

Calculating Parameters in Dense Layers

Each dense (fully connected) layer has:

$$\text{Parameters} = (\text{inputs} \times \text{neurons}) + \text{neurons (biases)}$$

Example Calculation

A layer with 5 inputs and 4 neurons has:

$$\text{Weights} = 5 \times 4 = 20, \quad \text{Biases} = 4 \Rightarrow \text{Total Parameters} = \mathbf{24}$$

Python sanity check

```
from tensorflow.keras.layers import Dense
layer = Dense(4, input_shape=(5,))
layer.build((None, 5))
print(layer.count_params())  # Output: 24
```

Stacking Layers to Build a Deep Network

Deep networks are built by stacking multiple layers. Each layer extracts higher-level features.

Example Architecture:

- Input Layer: 784 features (flattened 28×28 image)
- Hidden Layer 1: 128 neurons, ReLU
- Hidden Layer 2: 64 neurons, ReLU
- Output Layer: 10 neurons, Softmax

Model declaration in Keras

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(128, activation='relu', input_shape=(784,)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Current Section

What is an Activation Function?

An **activation function** introduces non-linearity into the neural network, allowing it to learn complex patterns.

Purpose

Without non-linear activation functions, stacked layers would behave like a single linear transformation.

Mathematical Form:

$$a = \sigma(z), \quad \text{where } z = \mathbf{w}^\top \mathbf{x} + b$$

Common Activation Functions

1. Sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Use Case

Binary classification (output layer).

2. Tanh:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \in [-1, 1]$$

3. ReLU (Rectified Linear Unit):

$$\text{ReLU}(z) = \max(0, z)$$

Use Case

Most popular for hidden layers due to simplicity and efficiency.

Softmax Activation

For multi-class classification (output layer):

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \ldots, K$$

Use Case

Used in the output layer of classifiers with more than 2 classes to produce probability distribution over classes.

Visual Comparison (Verbal Description)

- **Sigmoid:** Smooth, S-shaped curve in range (0, 1); prone to vanishing gradients
- **Tanh:** S-shaped but centered at 0; still suffers from vanishing gradient at extremes
- **ReLU:** Outputs 0 for negative inputs, linear for positive; sparse activation
- **Softmax:** Converts raw scores into probability vectors

Best Practices

- Use ReLU in hidden layers
- Use sigmoid for binary classification output
- Use softmax for multi-class classification output

Python Example: ReLU and Sigmoid

Simple example using TensorFlow

```
import tensorflow as tf

x = tf.constant([-2.0, -0.5, 0.0, 1.0, 2.0])
relu_out = tf.nn.relu(x)
sigmoid_out = tf.nn.sigmoid(x)

print("ReLU:", relu_out.numpy())
print("Sigmoid:", sigmoid_out.numpy())

# Output:
# ReLU: [0. 0. 0. 1. 2.]
# Sigmoid: [0.119 0.378 0.5 0.731 0.881]
```

Interpretation

ReLU filters out negative inputs; sigmoid squashes values to [0, 1].

Current Section

What is an Optimizer?

An **optimizer** updates the model's weights to minimize the loss function. It computes gradients via backpropagation and adjusts weights accordingly.

General Weight Update Rule

$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

Where:

- η: learning rate
- $\frac{\partial L}{\partial w}$: gradient of the loss w.r.t. weight

Adam Optimizer (Adaptive Moment Estimation)

Adam combines the ideas of Momentum and RMSprop:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(w) \quad \text{(momentum)}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(w))^2 \quad \text{(adaptive rates)}$$

$$w \leftarrow w - \frac{\eta}{\sqrt{v_t} + \varepsilon} \cdot m_t$$

Advantages of Adam

- Combines benefits of momentum + adaptive learning
- Works well for most tasks without tuning
- Default in TensorFlow/Keras

Choosing an Optimizer in Practice

Optimizer	When to Use
SGD	Simple models, theoretical analysis, interpretable convergence
SGD + Momentum	Faster convergence on convex surfaces or deeper networks
RMSprop	Good for RNNs or non-stationary objectives
Adam	Default for deep learning tasks; balances speed and robustness

Python Example: Optimizer in Keras

Switching optimizers in Keras

```
from tensorflow.keras.optimizers import Adam, SGD

# Adam with default settings
model.compile(optimizer=Adam(),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# SGD with momentum
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Tip

Try Adam first. If unstable or overfitting, experiment with learning rate or switch to SGD with decay.

Current Section

Steps to Build an ANN in Keras

Workflow:

1. Define model architecture
2. Compile model (specify loss, optimizer, metrics)
3. Train the model with training data
4. Evaluate and predict on test data

Keras API: High-level and beginner-friendly

Ideal for prototyping, testing, and deep learning pipelines.

Step 1: Define the Model

Simple MLP for classification

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(128, activation='relu', input_shape=(784,)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Explanation

`Sequential` builds the model layer by layer. `Dense` is a fully connected layer.

Step 2: Compile the Model

Specifying training configuration

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

Explanation

`sparse_categorical_crossentropy` is used when targets are integer-encoded.

Step 3: Train the Model

Fitting the model to training data

```
history = model.fit(
    X_train, y_train,
    validation_split=0.1,
    epochs=10,
    batch_size=64
)
```

Output

Trains over 10 epochs using 90% of data and validates on 10%. Returns a history object with metrics.

Step 4: Evaluate and Predict

Test evaluation and inference

```
loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.3f}")

# Predict probability for one sample
probs = model.predict(X_test[:1])
predicted_class = probs.argmax()
```

Typical Output

- `Test Accuracy: 0.975`
- `Predicted Class: 7`

Current Section

What is MNIST?

MNIST (Modified National Institute of Standards and Technology) is a benchmark dataset for handwritten digit classification.

- 70,000 grayscale images of handwritten digits (28×28 pixels)
- 60,000 training samples, 10,000 test samples
- Classes: digits from 0 to 9 (10 total)

Why MNIST?

`MNIST` is widely used as an entry point for deep learning and computer vision models due to its simplicity and visual interpretability.

Loading and Exploring the MNIST Dataset

Load and inspect MNIST

```python
from tensorflow.keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("Train shape:", X_train.shape)
print("Labels:", set(y_train))

# Output:
# Train shape: (60000, 28, 28)
# Labels: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Data Shape

Each image is 28×28 pixels, and labels are integer class indices (0–9).

Preprocessing MNIST for a Neural Network

Flatten and normalize the images

```
# Flatten 28x28 -> 784 vector and normalize to [0, 1]
X_train = X_train.reshape(-1, 784).astype("float32") / 255.0
X_test = X_test.reshape(-1, 784).astype("float32") / 255.0
```

Why normalize?

Neural networks converge faster when features are on a common scale (typically [0, 1]).

Training an ANN on MNIST

Train and Test the model

```
model.fit(
    X_train, y_train,
    validation_split=0.1,
    epochs=10,
    batch_size=64
)

test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test Accuracy:", round(test_acc * 100, 2), "%")
```

Typical Output

Training accuracy may reach around 90%–92%, but validation accuracy may plateau or even decrease due to poor generalization.

Current Section

Why Go Beyond Accuracy?

Accuracy is often used to evaluate classification performance, but it may be misleading when:

- Data is imbalanced
- Certain types of errors (e.g., false negatives) are more critical

Warning

High accuracy does not guarantee good performance across all classes or use cases.

Confusion Matrix: Foundation of Evaluation

For multi-class classification, the confusion matrix summarizes prediction outcomes:
Each cell [i, j] represents how often class `i` was predicted as class `j`.

Key Terms (Binary Classification)

- **TP (True Positive)**: Correctly predicted positive
- **TN (True Negative)**: Correctly predicted negative
- **FP (False Positive)**: Incorrectly predicted as positive
- **FN (False Negative)**: Missed positive instance

Precision, Recall, and F1-Score

Precision:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad \text{(How many predicted positives are correct?)}$$

Recall (Sensitivity):

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{(How many actual positives were captured?)}$$

F1-Score:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Use Case

F1 is ideal when classes are imbalanced or when both FP and FN have costs.

Example: Classification Evaluation

Compute metrics with sklearn

```
from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score

y_true = [1, 0, 1, 1, 0, 1, 0]
y_pred = [1, 0, 0, 1, 0, 1, 1]

print("Accuracy:", accuracy_score(y_true, y_pred))
print("Precision:", precision_score(y_true, y_pred))
print("Recall:", recall_score(y_true, y_pred))
print("F1 Score:", f1_score(y_true, y_pred))
# Output:
# Accuracy: 0.71
# Precision: 0.75
# Recall: 0.75
# F1 Score: 0.75
```

Interpretation

Precision, recall, and F1 offer a more complete picture than accuracy alone.

Current Section

Why ANNs Struggle with Image Inputs

ANNs require vectorized inputs – for example, flattening a 28×28 image into a 784-dimensional vector. This causes several problems:

Key Limitations

- **Loss of spatial structure:** Pixel positions and neighborhoods are ignored
- **Large parameter count:** High-dimensional inputs cause exponential growth in weights
- **Overfitting risk:** Large models memorize training data, especially with few samples
- **No translation invariance:** Same object in different positions looks "different"

Example: ANN on a 100×100 RGB Image

Input dimension: $100 \times 100 \times 3 = 30,000$

Parameter Explosion

A fully connected layer with 512 neurons:

$$\text{Weights} = 30,000 \times 512 = 15,360,000$$

Plus 512 biases = **15.36M+ parameters**

Warning

Huge models are inefficient, overfit easily, and require massive compute.

Translation and Spatial Awareness

- ANNs do not “see” images as grids – shuffling pixels produces the same input vector length
- They have no built-in concept of **local patterns** like edges or shapes
- A “7” shifted 5 pixels to the right will activate entirely different neurons

Result

ANNs fail to generalize well on image tasks with rotation, scaling, or translation.

The Need for CNNs in Image Processing

To address the structural inefficiencies of ANNs in vision:

Convolutional Neural Networks (CNNs)

- Preserve spatial relationships via 2D filters
- Use local receptive fields and shared weights
- Automatically detect hierarchical patterns (edges -> shapes -> objects)
- Scale better with image size

Next Week

We explore CNNs and how they address the challenges introduced in this section.