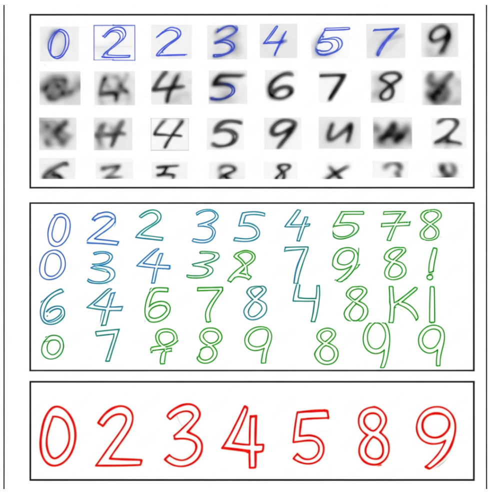
Minor in AI

FNN and RNN

May 8, 2025



1 The Magic Behind Your Phone's Handwriting Recognition

Imagine writing the sequence "1, 2, 3" on your smartphone screen. Like magic, it suggests "4" as the next number. This predictive capability mirrors two fundamental neural network paradigms:

- **Feedforward Neural Networks (FNNs)**: Analyze complete patterns at once (e.g., recognizing handwritten digits in the MNIST dataset).
- **Recurrent Neural Networks (RNNs)**: Process sequential data step-by-step (e.g., predicting the next number in a sequence).

Real-World Analogy: Just as FNNs power the U.S. Postal Service's automated mail sorting (processing 20,000 addresses/hour with 98% accuracy), RNNs enable Google's Smart Compose feature that predicts your next email sentence. Both architectures solve different problems using the same core principle: learning patterns from data.

Why This Matters

Understanding these networks helps build systems that can:

- Diagnose diseases from X-rays (FNNs)
- Predict stock market trends (RNNs)
- Translate languages in real-time (RNNs)

2 Feedforward Neural Networks: The Pattern Detectives

2.1 How Your Brain Processes Images (Simplified)

When you glance at a handwritten "5", your brain instantly:

1. Breaks it into edges and curves
2. Matches these patterns to known digit features
3. Confidently identifies it as “5”

FNNs mimic this process computationally:

Listing 1: MNIST Image Processing

```
# Original image: 28x28 grid of pixel brightness (0=black to 255=white)
[[0, 0, 255, ..., 0],
 [0, 255, 0, ..., 0],
 ...]

# Flatten to 784 pixels (28*28) for FNN input
[0, 0, 255, ..., 0, 0, 255, 0, ..., 0]
```

Key Insight: Flattening converts spatial relationships into linear patterns the network can learn, similar to how you might describe an image’s features left-to-right, top-to-bottom.

2.2 Building the Digital Brain

Listing 2: FNN Architecture for MNIST

```
model = Sequential([
    Dense(128, activation='relu', input_shape=(784,)), # Hidden Layer 1
    Dense(64, activation='relu'),                      # Hidden Layer 2
    Dense(10, activation='softmax')                    # Output Layer
])
```

Layer-by-Layer Explanation:

- **Input Layer (784 neurons)**: Each neuron represents one pixel’s brightness. A value of 0.9 means “very white”, 0.1 means “dark”.
- **Hidden Layer 1 (128 neurons)**: Learns basic shapes. Neuron 1 might activate for curves, Neuron 2 for diagonal lines.
- **Hidden Layer 2 (64 neurons)**: Combines basic shapes into digit parts. Neuron 1 detects “loops” (common in 0,6,8), Neuron 2 detects “straight lines” (1,4,7).
- **Output Layer (10 neurons)**: Each neuron represents a digit (0-9). The highest value determines the prediction.

Why ReLU?

The Rectified Linear Unit (ReLU) activation $f(x) = \max(0, x)$ acts like a filter:

- Allows positive signals through unchanged (”Yes, this looks like a curve!”)
- Blocks negative signals (”No relevant pattern here”)

This mimics how biological neurons fire or stay quiet.

2.3 Training: The Learning Process

Listing 3: Training Configuration

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5)
```

Step-by-Step Learning:

1. **Epoch 1**: The network makes random guesses. Accuracy 30% (worse than chance!).
2. **Epoch 3**: Discerns basic shapes. Accuracy jumps to 85%.
3. **Epoch 5**: Refines edge cases (e.g., distinguishing 5 from 6). Final test accuracy: 92.69%.

Behind the Scenes: - The loss function calculates prediction errors: Loss $= -\log$(Probability of Correct Class) - The Adam optimizer adjusts weights using calculus (derivatives), like tuning radio knobs for clearer signal

3 Recurrent Neural Networks: The Time Travelers

3.1 Predicting the Future (One Step at a Time)

Consider daily temperature data: [18°C, 20°C, 22°C]. An RNN detects the upward trend to predict 24°C. This temporal reasoning makes RNNs ideal for:

- Stock price prediction
- Speech recognition ("Play" → "Play some" → "Play some music")
- Writing assistance (predicting next word)

Listing 4: RNN for Sequence Prediction

```
model = Sequential([
    SimpleRNN(10, input_shape=(3, 1)),  # 3 time steps, 1 feature
    Dense(1)                            # Predict next value
])
```

Data Structure Deep Dive:

- **FNN Input**: [1, 2, 3] (order doesn't matter)
- **RNN Input**: [[1], [2], [3]] (sequential processing)

Code Walkthrough: Temperature Prediction

```
# Training data: Sequence of 3 days → predict day 4
X_train = [[[18], [20], [22]]] # Shape: (1 batch, 3 timesteps, 1
feature)
y_train = [24]
# After training, predict next temperature:
test_input = [[[20], [22], [24]]]
prediction = model.predict(test_input) # Output: 2̃6°C
```

3.2 The Memory Mechanism

Recurrent Neural Networks (RNNs) are designed to process sequential data by maintaining a "hidden state" that carries information across time steps. This hidden state acts as the network's memory and is updated at each step based on the current input and the previous hidden state:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Where:

- h_t: *Current hidden state* — This represents the memory of the network at time step t. It captures relevant information from both the current input and the accumulated past, allowing the network to make context-aware predictions.
- h_{t-1}: *Previous hidden state* — This is the hidden state from the previous time step. It carries forward the historical information learned so far, effectively forming a memory chain across time.
- x_t: *Current input* — This is the new data point at time step t. For example, in a temperature prediction task, x_t could be today's temperature reading. This input is used along with the memory to update the state.
- W_{xh}, W_{hh}: *Learnable weight matrices* — These matrices determine how much influence the current input and the previous hidden state have on the new hidden state. They are adjusted during training to minimize prediction error.
- b_h: *Bias term* — This helps the model make better predictions by allowing for a trainable offset in the transformation.

The activation function tanh introduces non-linearity, ensuring that the network can learn complex patterns in the data.

Think of reading a novel:

- Your brain (the hidden state) stores key plot points and character developments from previous chapters.
- As you read a new chapter (the current input), your brain updates this understanding, integrating new information with what you already know.

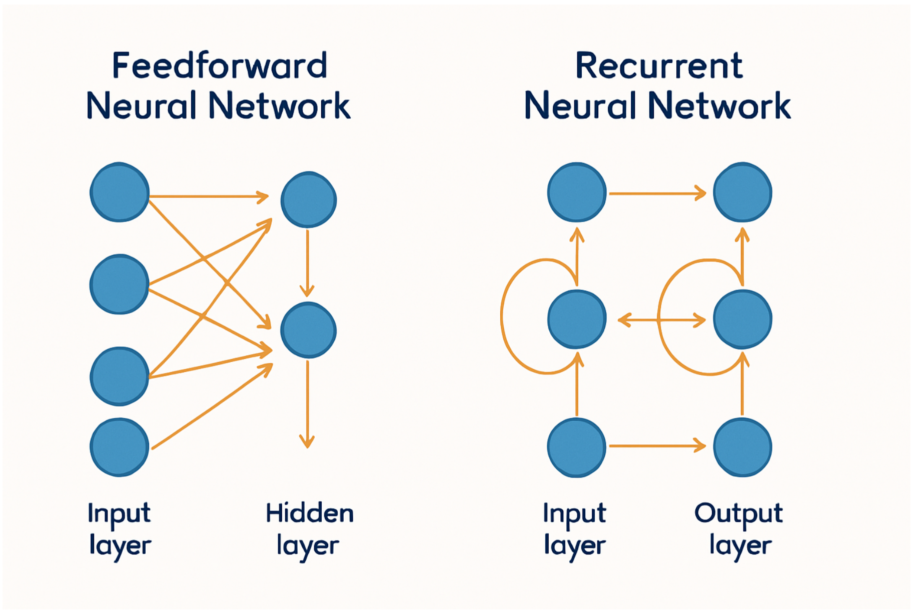
4 MNIST in Action: From Pixels to Predictions

4.1 Data Preparation: The Unsung Hero

Listing 5: MNIST Data Pipeline

```python
# Load 70,000 handwritten digits
(train_images, train_labels), (test_images, test_labels) = mnist.
    load_data()

# Flatten 28x28 images to 784 pixels each
train_images = train_images.reshape((60000, 784))
```



```

# Normalize pixel values (0-255 -> 0-1)
train_images = train_images.astype('float32') / 255
```

Why Normalize? Imagine comparing weights in kilograms and grams without conversion. Normalization ensures equal feature scaling so the network doesn't mistakenly prioritize brighter pixels.

4.2 Architecture Decisions Decoded

- **128 Hidden Neurons**: Enough to capture basic shapes without memorizing noise. Too few (e.g., 32) underfit; too many (512) overfit.
- **ReLU vs Sigmoid**: ReLU's simplicity speeds up training. Sigmoid's smooth curve (0-1) is better for probability outputs.
- **Output Layer Design**: 10 neurons with softmax create a probability distribution. For digit "7", output might be [0.01, 0.02, ..., 0.87, ...].

5 Key Takeaways: Neural Networks Demystified

- **FNNs for Static Data**: Feedforward Neural Networks (FNNs) are best suited for data where the order of features does not carry inherent meaning — such as images. For instance, in the MNIST digit classification task, each image is treated as a fixed set of pixel values. FNNs can achieve high accuracy (e.g., 92.69%) by learning hierarchical representations — first recognizing edges, then shapes, and finally digits.
- **RNNs for Sequences**: Recurrent Neural Networks (RNNs) are designed for sequential or time-dependent data, where the order of inputs matters. They maintain a hidden state (memory) that evolves over time. For example, given a temperature sequence like [2, 3, 4], an RNN can learn to predict the next value (5) by recognizing temporal patterns and retaining context across steps.

- **Data Preparation**: Proper preprocessing is essential for model performance:
  - *Flattening*: Converts structured data like images into 1D vectors so they can be input into a neural network. For example, a 28x28 image becomes a 784-length vector.
  - *Normalization*: Scales features to a standard range (commonly 0–1). This ensures consistent gradient flow and speeds up convergence during training.
- **Activation Functions**: Non-linearities enable neural networks to learn complex functions:
  - *ReLU (Rectified Linear Unit)*: Efficiently introduces non-linearity by zeroing out negative values, allowing for sparse and fast computations.
  - *Softmax*: Used in classification tasks to convert raw output scores into probabilities that sum to 1, making interpretation straightforward.
- **Start Simple**: In practice, simpler models often generalize better than unnecessarily deep ones. Overly complex networks can overfit the training data, learning noise rather than signal. In tasks like MNIST, a well-tuned 2-layer FNN can outperform deeper networks that lack proper regularization or sufficient data.

From Classroom to Real World

Try These Improvements:

- **Learning Rate Tuning**: Change optimizer to SGD with lr=0.01. Accuracy may drop initially but converge better.
- **Add Dropout**: Insert Dropout(0.2) after hidden layers. Reduces overfitting for noisy real-world data.
- **RNN Depth**: Stack two SimpleRNN layers. Improves sequence prediction but requires more data.