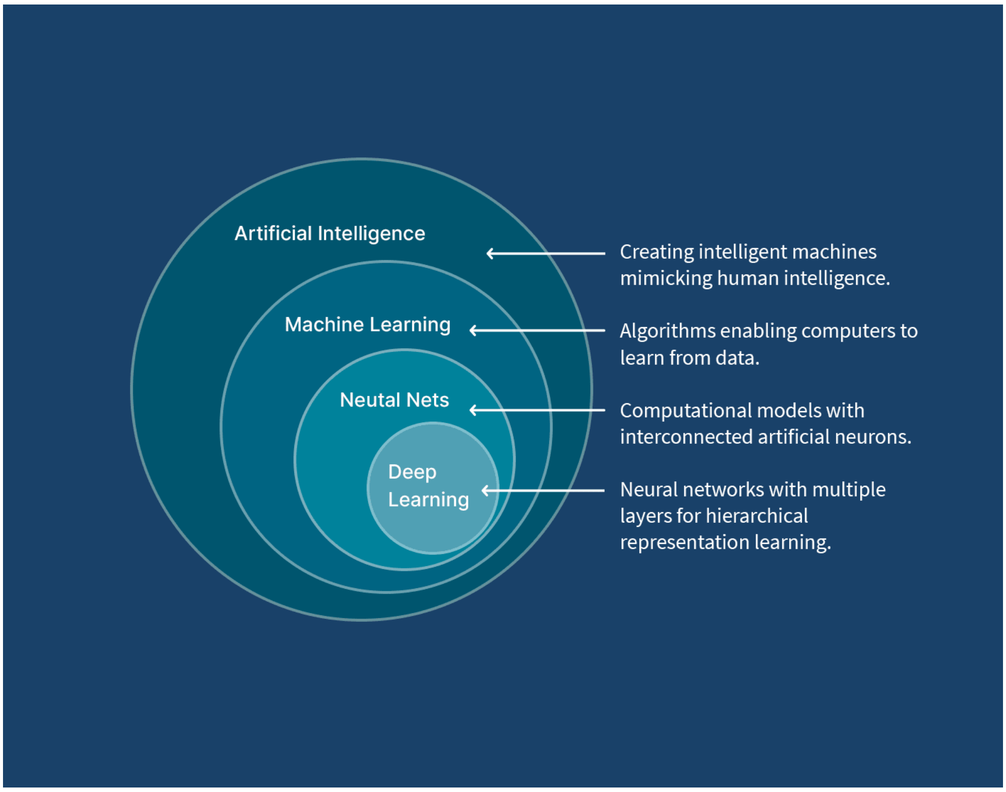
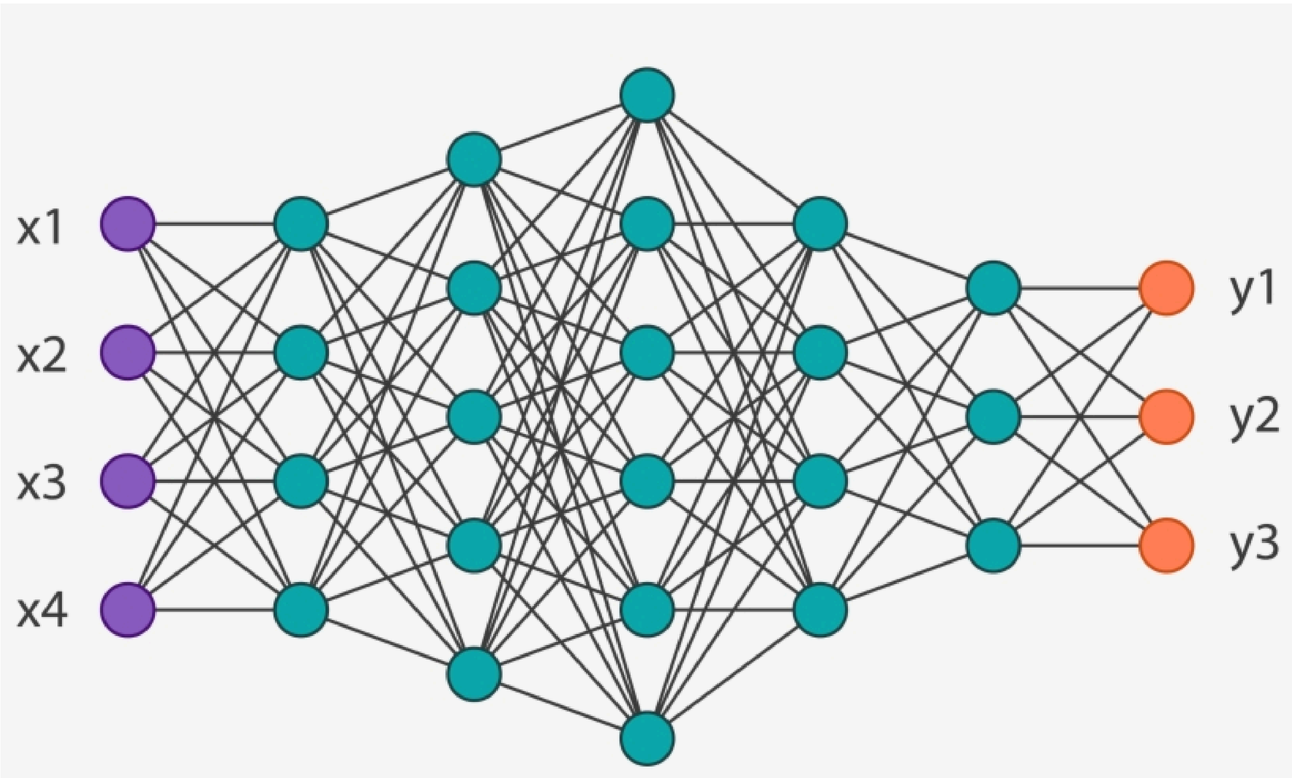
Neural Network Lesson Notes

1. Introduction

What is a Neural Network?





A **Neural Network (NN)** is a type of machine learning model inspired by how the **human brain** works. It is made of layers of small units called **neurons**, which take input, process it, and pass it forward to the next layer.

Neural networks are especially powerful for tasks where traditional algorithms struggle, such as recognizing images, understanding text, or making predictions from complex data.

Think of a neural network like a group of **tiny decision-makers** working together:

- Each neuron makes a small, simple decision
- When combined, the network can solve **very complex problems**

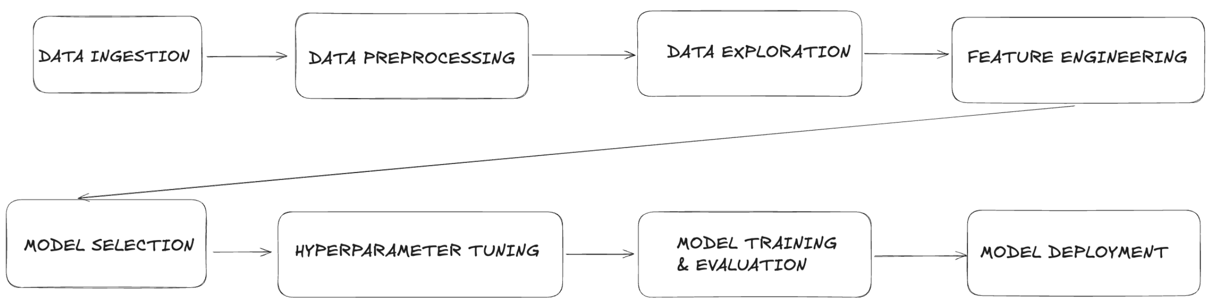
Why are Neural Networks Important?

- **Handle Complexity**: Can model very complicated, non-linear patterns
- **Adaptability**: Learn directly from raw data (like pixels, audio, or words)
- **Foundation of Deep Learning**: Power modern AI systems such as ChatGPT, image recognition, and recommendation engines

Traditional Machine Learning vs Neural Networks

Aspect	Machine Learning (ML)	Neural Networks (NN)
What it is	A way for computers to learn from data.	A special type of ML inspired by the brain.
Examples	Decision trees, regression, SVM.	CNNs, RNNs, Transformers.
Data need	Works with small or medium data.	Needs lots of data.
Speed	Trains faster, simpler.	Slower, more complex.
Features	Needs humans to pick features.	Learns features automatically.
Best for	Simple or structured data (numbers, tables).	Complex data (images, text, audio).

How Neural Networks Fit into the ML Pipeline



Step	Short Definition
1. Data Ingestion	Load data from files, APIs, or databases
2. Data Preprocessing	Clean and fix messy data

Step	Short Definition
3. Data Exploration	Look at data to understand patterns
4. Feature Engineering	Create better input variables
5. Model Selection	Pick the best algorithm
6. Hyperparameter Tuning	Adjust settings for better performance
7. Model Training and Evaluation	Train model and test accuracy
8. Model Deployment	Put model into real use

Neural Networks belong to **Step 5: Model Selection** in the ML pipeline.
They are one of the algorithms you can choose, just like regression, decision trees, or SVM.

Key differences when using Neural Networks:

- **Feature Engineering (Step 4):** NNs can automatically learn features from raw data (especially images, text, audio), so you may need less manual feature crafting.
- **Training (Step 7):** NNs often require more data and computational power compared to traditional ML models.

In short: Neural Networks don't change the overall pipeline, but they make **Step 5** more powerful and can reduce the effort in **Step 4**.

2. Glossary

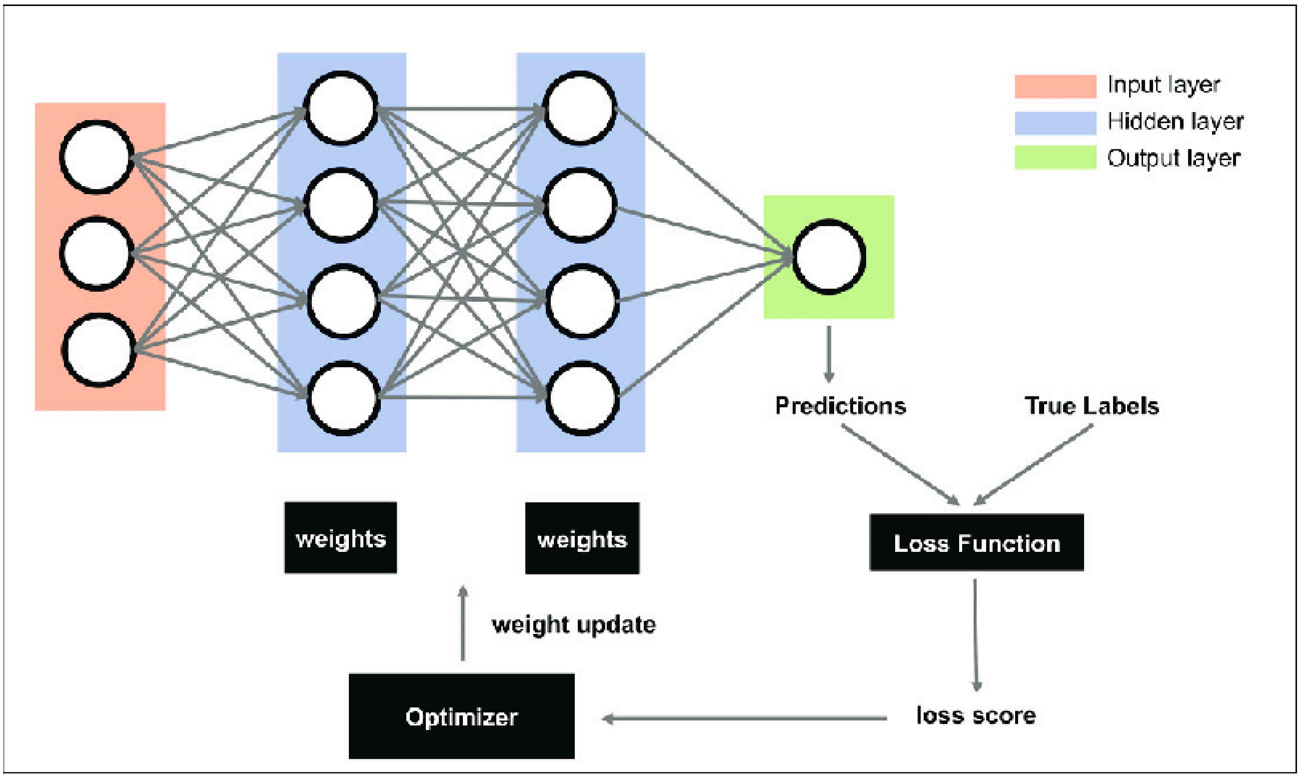
- **Neuron**: A tiny unit in neural network that; takes input, processes it, and outputs a value.
- **Layer**: A collection of neurons. Neural networks are built from multiple layers: input, hidden, output.
- **Input Layer**: Where the data enters the network (e.g., pixels of an image).
- **Hidden Layer**: Middle layers that transform data and learn patterns.
- **Output Layer**: Final predictions (e.g., "cat" vs "dog").
- **Weights (W)**: Numbers that decide how important each input is.
- **Bias (b)**: A number that helps shift the result so the neuron can work better.
- **Activation Function**: A rule that changes the neuron's output before passing it to the next layer (e.g., ReLU makes

negatives zero, Sigmoid turns numbers into 0–1).

- **Loss Function**: A number that shows how far off the model's predictions are from the correct answers.
- **Epoch**: One full training cycle through the entire dataset.

- **Gradient**: The direction and size of change we need to make to weights and biases to reduce the loss.

3. Neural Network Workflow (Training Process)



1. Forward Pass

- **Input data** enters the network through the **Input Layer** (orange)
- Data flows through **Hidden Layers** (blue) where neurons process and transform the information
- Each connection has **weights** that determine how much

influence each input has

- Finally, the data reaches the **Output Layer** (green), which produces **predictions**

2. Loss Calculation

- Compare the **predictions** with the **true labels** (actual correct answers)
- The **loss function** calculates a single number showing how wrong our predictions are
- **Higher loss** = worse performance, **Lower loss** = better performance

3. Optimization

- The **loss score** goes to the **optimizer**
- The optimizer decides how to **adjust the weights** to reduce the loss

- Uses **gradients** (calculated through backpropagation) to determine the best direction to change each weight

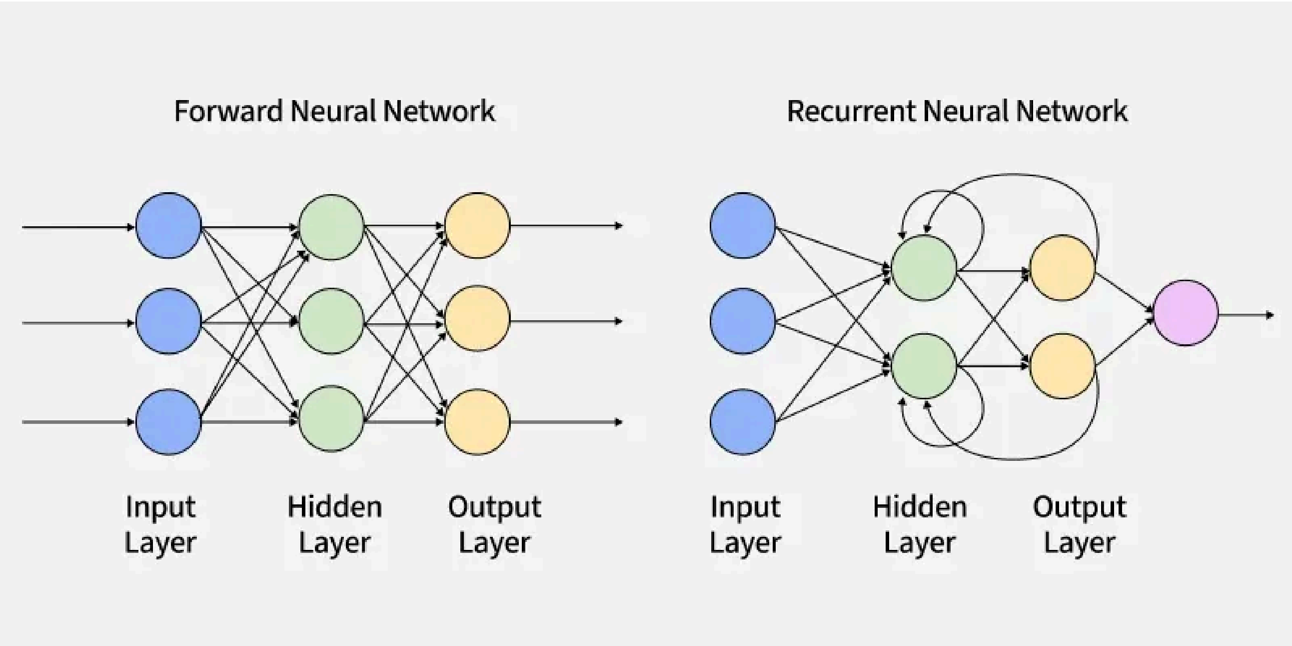
4. Weight Update

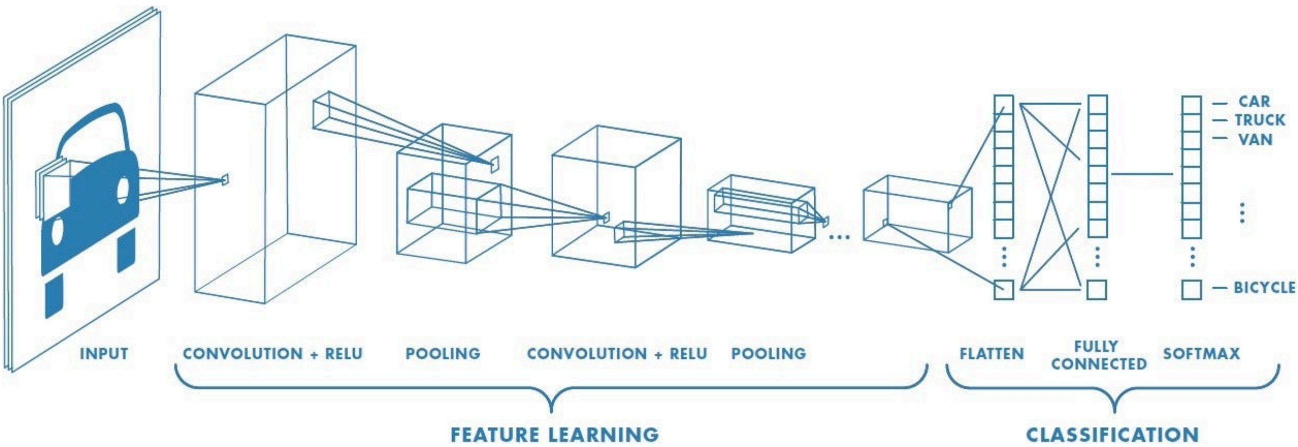
- The optimizer sends **update instructions** back to all the weights in the network
- All **weights** between layers get adjusted slightly
- These **small adjustments** should make the network perform better on the next attempt

5. Repeat

- This entire process **repeats many times** (thousands of iterations)
- Each cycle: **Forward Pass → Loss Calculation → Optimization → Weight Update**
- Gradually, the network gets **better at making accurate predictions**

4. Types of Neural Networks

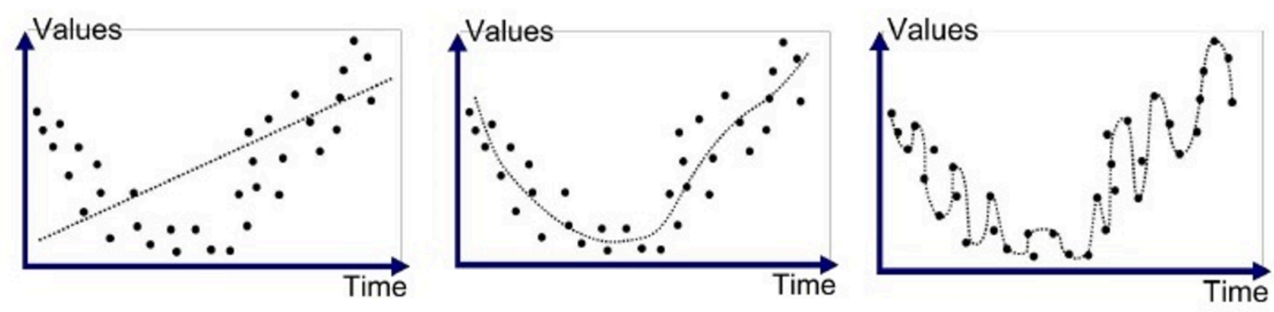




	Feedforward NN	Convolutional NN (CNN)	Recurrent NN (RNN)
Definition	Basic neural network where data flows in one direction	Specialized for image/visual data with filters	Can remember previous information for sequences
Best For	- Simple classification - Basic regression - Tabular data	- Image recognition - Computer vision - Medical imaging	- Text processing - Time series - Speech recognition
Architecture	Input → Hidden → Output	Convolution + Pooling layers	Hidden states that loop back
Example Use	Predict house prices from features	Detect cats in photos	Translate languages

5. Common Neural Network Problems & Solutions

1. Overfitting vs Underfitting



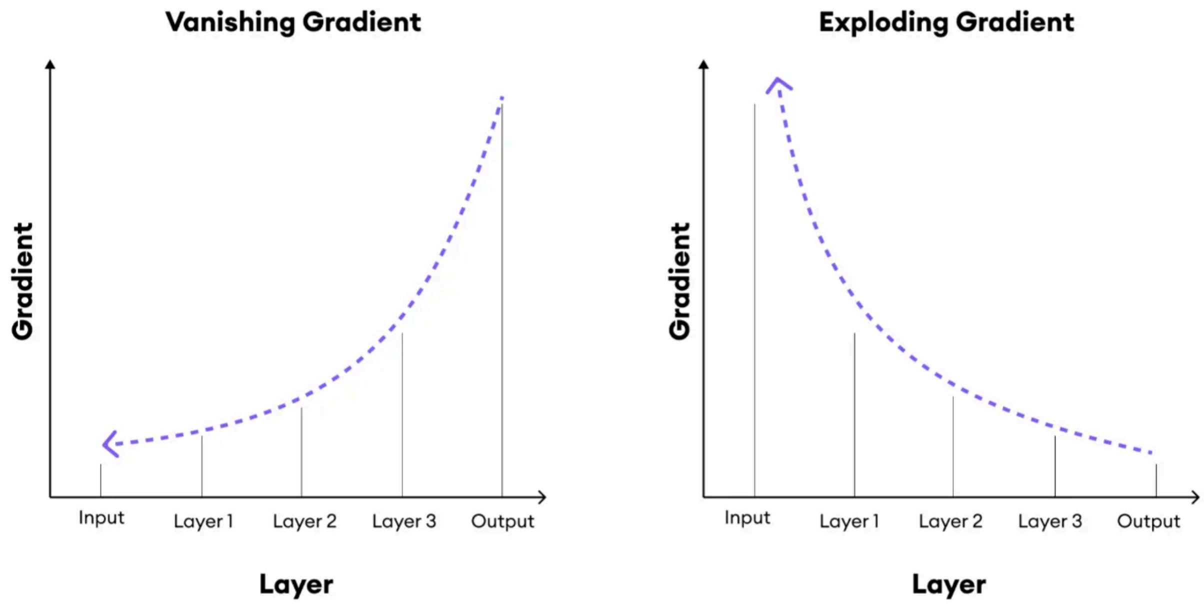
Underfitted   Good Fit/Robust   Overfitted

	Underfitting	Overfitting	Good Fit
Training Loss	High	Very Low	Moderate
Validation Loss	High	High (much higher than training)	Similar to training
Symptoms	Network is too simple	Network memorizes training data	Generalizes well
Solutions	- Add more layers - Add more neurons - Train longer	- Add dropout - Use regularization - Get more data - Reduce network size	- Monitor validation loss - Use early stopping

2. Vanishing vs Exploding Gradient Problems

Gradient is essentially the **direction and strength** of the steepest change in a function. In neural networks, it tells us:

1. **Which direction** to adjust each weight to reduce the error.
2. **How much** to change each weight.



	Vanishing Gradients	Exploding Gradients
What happens	Gradients become very small as they flow backward	Gradients become very large as they flow backward
Effect	Early layers learn very slowly or stop learning	Weights change wildly, training becomes unstable
Symptoms	- Training loss decreases very slowly - Early layers barely update - Network seems "stuck"	- Training loss jumps around wildly - Weights become very large - Network performance degrades suddenly
Common in	- Deep networks with Sigmoid/Tanh - Very deep networks - Vanilla RNNs	- Very deep networks - Poor weight initialization - High learning rates
Solutions	- Use ReLU activation - Proper weight initialization (He, Xavier) - Batch normalization - Residual connections - LSTM/GRU for RNNs	- Gradient clipping - Lower learning rates - Better weight initialization - Batch normalization - Regularization

Think of it like this:

- **Vanishing**: Like playing telephone - the message gets weaker as it passes through more people
- **Exploding**: Like an avalanche - small changes become massive as they roll downhill

6. Tips for Better Neural Networks

Training Tips

- **Start simple**: Begin with a basic network, then add complexity
- **Monitor both training and validation loss**: Watch for overfitting
- **Use appropriate learning rates**: Too high = unstable, too low = slow

- **Batch normalization**: Helps with training stability
- **Early stopping**: Stop training when validation loss stops improving

Architecture Tips

- **Hidden layer size**: Often between input and output size
- **Depth vs Width**: Deeper networks can learn more complex patterns
- **Activation functions**: ReLU for hidden layers, Sigmoid/Softmax for output
- **Regularization**: Dropout, L1/L2 regularization to prevent overfitting

Data Tips

- **Normalize inputs**: Scale features to similar ranges
- **Augment data**: Create more training examples (especially for images)
- **Balance classes**: Ensure equal representation of different categories
- **Quality over quantity**: Clean, relevant data is better than lots of noisy data

7. Useful References

1. Learning Resources

- **3Blue1Brown Neural Networks Series**
  YouTube: Neural Networks
  Best visual explanation of how neural networks work

- **Fast.ai**
  Website: course.fast.ai
  Practical deep learning for coders

2. Frameworks & Tools

- **TensorFlow/Keras**: Most popular deep learning framework
- **PyTorch**: Research-friendly framework with dynamic graphs
- **Scikit-learn**: Good for simple neural networks
- **Google Colab**: Free GPU access for training

3. Datasets for Practice

- **MNIST**: Handwritten digits (beginner-friendly)

- **CIFAR-10**: Small images with 10 categories
- **ImageNet**: Large-scale image recognition
- **IMDB Reviews**: Text sentiment analysis
- **Boston Housing**: Regression problem

🎯 Let's Code: Fashion Image Classification with Neural Networks

In this notebook, we'll build a complete **neural network from scratch** using TensorFlow to classify fashion items from images.

What We'll Cover:

1. **Data Ingestion** – Load Fashion-MNIST dataset from CSV files containing 28x28 grayscale images
2. **Data Preprocessing** – Normalize pixel values, shuffle data, and split into train/validation/test sets
3. **Data Exploration** – Visualize sample images, check data distribution, and understand the 10 fashion categories
4. **Network Architecture Design** – Build a simple feedforward neural network with input, hidden, and output layers
5. **Model Compilation** – Configure Adam optimizer, sparse categorical crossentropy loss, and accuracy metrics
6. **Model Training & Evaluation** – Train for 20 epochs with validation monitoring and plot learning curves
7. **Model Deployment** – Save the trained model and load it for making predictions on new fashion images
8. **Results Visualization** – Display predicted vs. actual labels with user-friendly category names

We'll also use:

- **TensorFlow/Keras** for neural network implementation
- **NumPy/Pandas** for data manipulation
- **Matplotlib** for visualization and plotting training curves
- **Gradio** for creating interactive interfaces (setup included)

Fashion Categories We'll Classify:

- T-shirt/top, Trouser, Pullover, Dress, Coat
- Sandal, Shirt, Sneaker, Bag, Ankle boot

Learning Goals:

- Understand the complete neural network workflow

- Experience forward pass, loss calculation, and backpropagation
- Learn proper model saving and loading techniques
- Practice with real image classification tasks

Pipeline 1 - Data Preparation

Step 1: Install Dependencies

Install specific version of Gradio for creating interactive interfaces. Colab already includes numpy, pandas, matplotlib, pillow, and tensorflow, so we only need to add Gradio and verify all library versions.

In [ ]:
```
# ─── Cell 1: Install only Gradio & verify existing packages ──────────────────
!pip install gradio==3.36.0 --quiet

# Verify versions of all key libraries
import numpy as np, pandas as pd, matplotlib, PIL, tensorflow as tf, gradio
print(f"▶ numpy      {np.__version__}")
print(f"▶ pandas     {pd.__version__}")
print(f"▶ matplotlib {matplotlib.__version__}")
print(f"▶ Pillow     {PIL.__version__}")
print(f"▶ tensorflow {tf.__version__}")
print(f"▶ gradio     {gradio.__version__}")
```

```
▶ numpy      2.0.2
▶ pandas     2.2.2
▶ matplotlib 3.10.0
▶ Pillow     11.3.0
▶ tensorflow 2.18.0
▶ gradio     3.36.0
```

Step 2: Import Libraries & Set Random Seeds

Import core libraries for neural networks (tensorflow), data manipulation (numpy, pandas), visualization (matplotlib), image processing (PIL), and interactive interfaces (gradio). Also sets random seeds for reproducibility across numpy, tensorflow, and random modules, plus defines user-friendly label names for Fashion-MNIST categories.

In [3]:
```
# ─── Cell 2 (UPDATED): Imports, Seeding & Label Names ────────────────────────
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw
import gradio as gr
import random
from tensorflow.keras.models import load_model

# Set random seeds for reproducibility
SEED = 42
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

```python
random.seed(SEED)

# User-friendly class names for FashionMNIST labels 0–9
LABEL_NAMES = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]

print(f"▶ Random seed set to {SEED}")
print("▶ Label mapping:", {i: name for i, name in enumerate(LABEL_NAMES)})
```

```
▶ Random seed set to 42
▶ Label mapping: {0: 'T-shirt/top', 1: 'Trouser', 2: 'Pullover', 3: 'Dress', 4: 'Coa
t', 5: 'Sandal', 6: 'Shirt', 7: 'Sneaker', 8: 'Bag', 9: 'Ankle boot'}
```

Pipeline 2 - Data Ingestion

Step 3: Load & Inspect Fashion-MNIST Dataset

Load the Fashion-MNIST training and testing datasets from CSV files, then perform comprehensive data inspection including:

- Dataset shapes (60,000 training samples, 10,000 test samples with 785 columns each)
- Column structure examination (1 label + 784 pixel values)
- Data quality checks (missing values, label distribution, pixel intensity ranges 0-255)
- Visualization of sample image to verify correct data loading

In [4]:
```python
# —— Cell 3: Load Data & Quick Inspect ——————————————————————————————
# Load the CSV files from Colab's local filesystem
train_df = pd.read_csv('fashion-mnist_train.csv')
test_df  = pd.read_csv('fashion-mnist_test.csv')
```

In [5]:
```python
# 1) Print dataset shapes
print(f"▶ train_df.shape = {train_df.shape}")
print(f"▶ test_df.shape  = {test_df.shape}")
```

```
▶ train_df.shape = (60000, 785)
▶ test_df.shape  = (10000, 785)
```

In [ ]:
```python
# 2) Peek at column names and first row
print("▶ train_df columns:", train_df.columns.tolist())
print("▶ train_df first row:\n", train_df.head(1))
```

In [7]:
```python
# 3) Check for any missing values
missing_train = train_df.isnull().sum().sum()
print(f"▶ Missing values in train_df: {missing_train}")
```

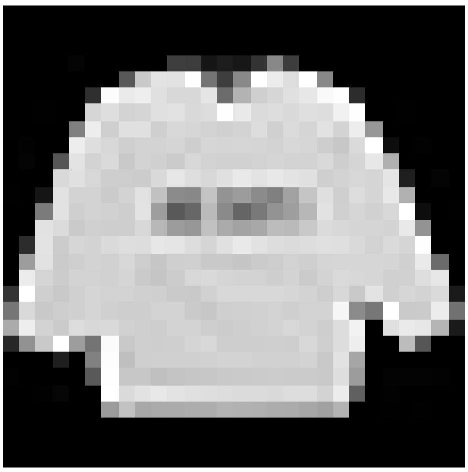
```
# 4) Label distribution in the training set
label_counts = train_df['label'].value_counts().sort_index()
print("▶ Label distribution:\n", label_counts)

# 5) Pixel intensity range across all pixels
pixel_min = train_df.iloc[:,1:].values.min()
pixel_max = train_df.iloc[:,1:].values.max()
print(f"▶ Pixel value range: [{pixel_min}, {pixel_max}]")
```

```
▶ Missing values in train_df: 0
▶ Label distribution:
 label
0    6000
1    6000
2    6000
3    6000
4    6000
5    6000
6    6000
7    6000
8    6000
9    6000
Name: count, dtype: int64
▶ Pixel value range: [0, 255]
```

In [8]:
```
# 6) Visualize one example image
example = train_df.iloc[0]
img_arr  = example.values[1:].astype(np.uint8).reshape(28,28)
plt.figure(figsize=(3,3))
plt.imshow(img_arr, cmap='gray')
plt.title(f"Label = {example[0]}")
plt.axis('off')
plt.show()
```

```
/tmp/ipython-input-594164400.py:6: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as l
abels (consistent with DataFrame behavior). To access a value by position, use `ser.il
oc[pos]`
  plt.title(f"Label = {example[0]}")
```

Label = 2



Pipeline 3 - Data Preprocessing

Step 4: Normalize Data & Create Train/Validation Split

Normalize pixel values from 0-255 range to 0-1 range for better neural network training. Separate features (pixels) from labels, shuffle the training data, and split into 90% training and 10% validation sets.

In [9]:
```
# ─── Cell 4: Preprocess & Split ──────────────────────────────────────────
# Normalize pixel values to [0,1] and separate features/labels
X        = train_df.iloc[:,1:].values.astype('float32') / 255.0
y        = train_df.iloc[:,0].values.astype('int32')
X_test   = test_df.iloc[:,1:].values.astype('float32') / 255.0
y_test   = test_df.iloc[:,0].values.astype('int32')

# Shuffle the training data
perm = np.random.permutation(len(X))
X, y = X[perm], y[perm]
print(f"▶ First 5 labels after shuffling: {y[:5]}")

# Split into 90% train / 10% validation
split_idx    = int(0.9 * len(X))
X_train, y_train = X[:split_idx], y[:split_idx]
X_val,   y_val   = X[split_idx:], y[split_idx:]
print(f"▶ Split sizes → train: {len(X_train)}, val: {len(X_val)}, test: {len(X_test)}
# ─── End of Cell 4 ───────────────────────────────────────────────────────
```

```
▶ First 5 labels after shuffling: [7 8 8 5 9]
▶ Split sizes → train: 54000, val: 6000, test: 10000
```

Pipeline 4 - Network Architecture Design

Step 5: Build & Compile Neural Network Model

Create a simple feedforward neural network with:

- Input layer (784 neurons for flattened 28x28 images)
- Hidden layer (128 neurons with ReLU activation)
- Output layer (10 neurons with softmax for 10 fashion categories)

Compile with Adam optimizer, sparse categorical crossentropy loss, and accuracy metric.

In [10]:
```
# ─── Cell 5: Build & Compile TensorFlow Model ────────────────────────────
# Define a simple feed-forward neural network with one hidden layer
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(784,)),      # 28x28 flattened
    tf.keras.layers.Dense(128, activation='relu'),       # Hidden layer
    tf.keras.layers.Dense(10, activation='softmax'),     # Output layer
])

# Compile with Adam optimizer and cross-entropy loss
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
```

```
)

# Display the model summary
model.summary()
# ── End of Cell 5 ──────────────────────────────────────────────────────────
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/input_layer.py:27: UserWarning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

Total params: 101,770 (397.54 KB)

Trainable params: 101,770 (397.54 KB)

Non-trainable params: 0 (0.00 B)

Pipeline 5 - Model Training & Evaluation

Step 6: Train Neural Network & Plot Training Curves

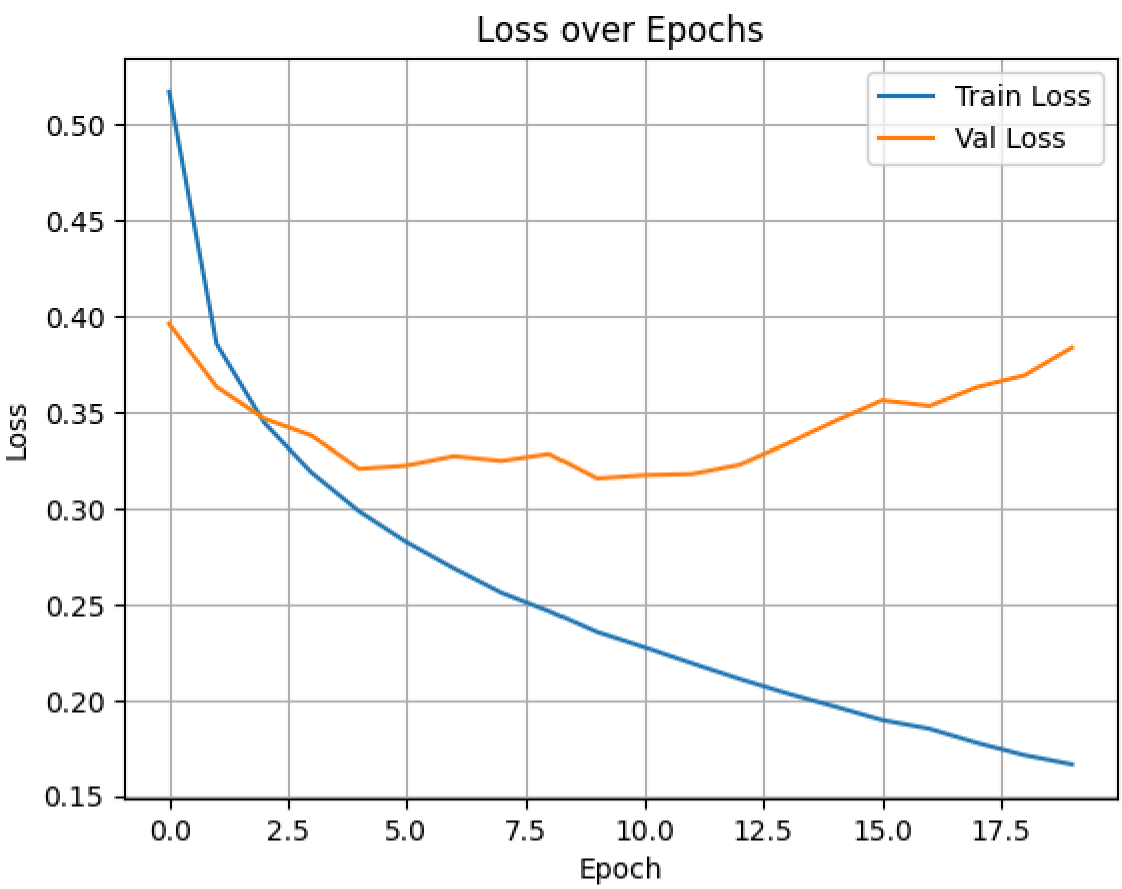
Train the model for 20 epochs using training data with validation monitoring. Use batch size of 32 and track both training and validation loss/accuracy throughout the training process. Then visualize training and validation curves over epochs to monitor learning progress and detect potential overfitting or underfitting issues.

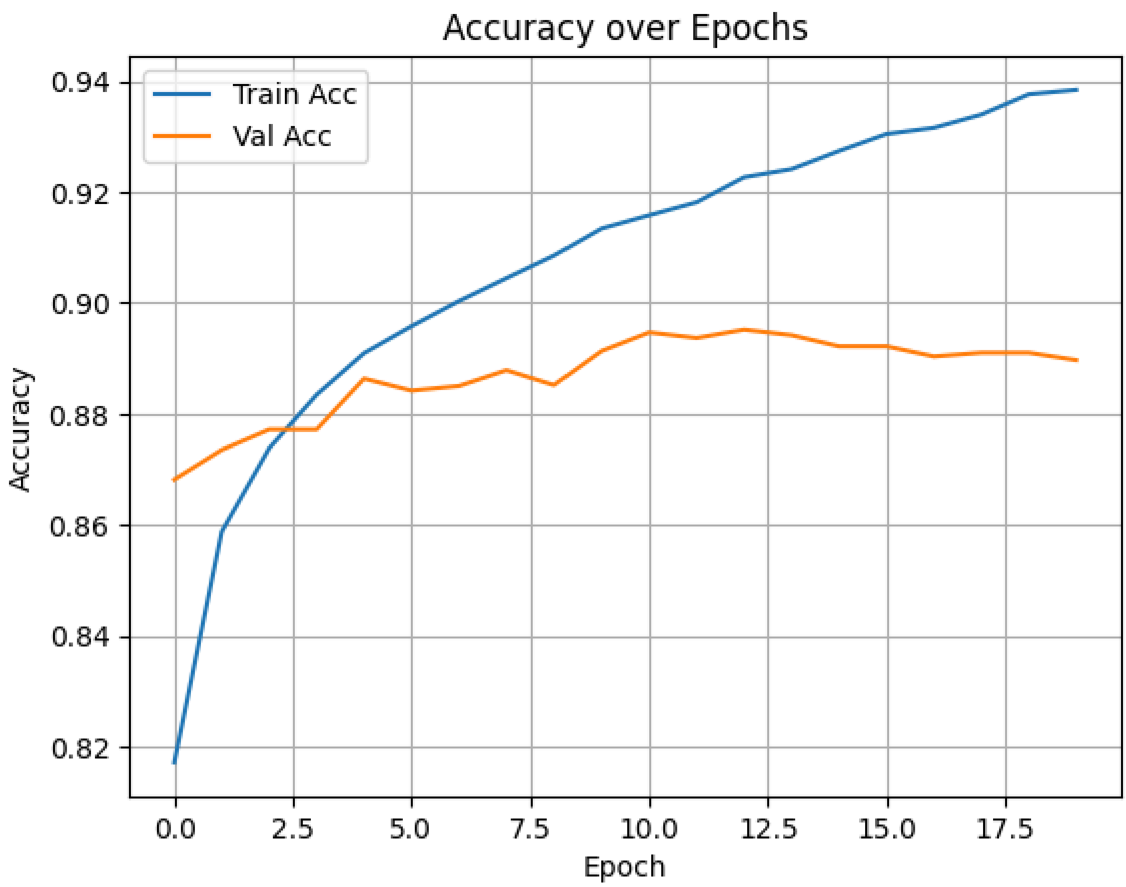
In [11]:

```
# ── Cell 6: Train & Plot Curves ────────────────────────────────────────────
# Train for 20 epochs with validation
EPOCHS = 20
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=EPOCHS,
    batch_size=32,
    verbose=2
)
```

```
Epoch 1/20
1688/1688 - 7s - 4ms/step - accuracy: 0.8171 - loss: 0.5171 - val_accuracy: 0.8682 - v
al_loss: 0.3961
Epoch 2/20
1688/1688 - 4s - 2ms/step - accuracy: 0.8588 - loss: 0.3856 - val_accuracy: 0.8735 - v
al_loss: 0.3633
Epoch 3/20
1688/1688 - 3s - 2ms/step - accuracy: 0.8738 - loss: 0.3446 - val_accuracy: 0.8772 - v
al_loss: 0.3467
Epoch 4/20
1688/1688 - 3s - 2ms/step - accuracy: 0.8834 - loss: 0.3187 - val_accuracy: 0.8772 - v
al_loss: 0.3379
Epoch 5/20
1688/1688 - 5s - 3ms/step - accuracy: 0.8909 - loss: 0.2986 - val_accuracy: 0.8863 - v
al_loss: 0.3207
Epoch 6/20
1688/1688 - 3s - 2ms/step - accuracy: 0.8958 - loss: 0.2825 - val_accuracy: 0.8842 - v
al_loss: 0.3224
Epoch 7/20
1688/1688 - 5s - 3ms/step - accuracy: 0.9003 - loss: 0.2688 - val_accuracy: 0.8850 - v
al_loss: 0.3273
Epoch 8/20
1688/1688 - 6s - 3ms/step - accuracy: 0.9045 - loss: 0.2560 - val_accuracy: 0.8878 - v
al_loss: 0.3248
Epoch 9/20
1688/1688 - 3s - 2ms/step - accuracy: 0.9086 - loss: 0.2464 - val_accuracy: 0.8852 - v
al_loss: 0.3284
Epoch 10/20
1688/1688 - 5s - 3ms/step - accuracy: 0.9134 - loss: 0.2356 - val_accuracy: 0.8913 - v
al_loss: 0.3157
Epoch 11/20
1688/1688 - 4s - 2ms/step - accuracy: 0.9158 - loss: 0.2276 - val_accuracy: 0.8947 - v
al_loss: 0.3174
Epoch 12/20
1688/1688 - 4s - 3ms/step - accuracy: 0.9182 - loss: 0.2192 - val_accuracy: 0.8937 - v
al_loss: 0.3180
Epoch 13/20
1688/1688 - 3s - 2ms/step - accuracy: 0.9227 - loss: 0.2111 - val_accuracy: 0.8952 - v
al_loss: 0.3228
Epoch 14/20
1688/1688 - 5s - 3ms/step - accuracy: 0.9242 - loss: 0.2035 - val_accuracy: 0.8942 - v
al_loss: 0.3336
Epoch 15/20
1688/1688 - 3s - 2ms/step - accuracy: 0.9274 - loss: 0.1967 - val_accuracy: 0.8922 - v
al_loss: 0.3453
Epoch 16/20
1688/1688 - 3s - 2ms/step - accuracy: 0.9305 - loss: 0.1897 - val_accuracy: 0.8922 - v
al_loss: 0.3562
Epoch 17/20
1688/1688 - 3s - 2ms/step - accuracy: 0.9316 - loss: 0.1851 - val_accuracy: 0.8903 - v
al_loss: 0.3533
Epoch 18/20
1688/1688 - 4s - 3ms/step - accuracy: 0.9340 - loss: 0.1778 - val_accuracy: 0.8910 - v
al_loss: 0.3631
Epoch 19/20
1688/1688 - 6s - 4ms/step - accuracy: 0.9377 - loss: 0.1713 - val_accuracy: 0.8910 - v
al_loss: 0.3692
Epoch 20/20
1688/1688 - 4s - 3ms/step - accuracy: 0.9385 - loss: 0.1665 - val_accuracy: 0.8897 - v
al_loss: 0.3837
```

In [12]:
```
# Plot training & validation loss
plt.figure()
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss over Epochs')
plt.grid(True)
plt.show()

# Plot training & validation accuracy
plt.figure()
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy over Epochs')
plt.grid(True)
plt.show()
```





Step 7: Evaluate Final Performance

Test the trained model on the unseen test dataset to get final performance metrics (test loss and test accuracy).

In [13]:
```
# ─── Cell 7 (UPDATED): Evaluate on Test Set & Visualize Samples ──────────────────
# Evaluate final performance on the test set
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss = {test_loss:.4f}, Test Accuracy = {test_acc:.4f}")
```

```
Test Loss = 0.3966, Test Accuracy = 0.8821
```

Pipeline 6 - Model Deployment

Step 8: Save Trained Model

Save the trained neural network model to disk using TensorFlow's native .keras format for later use and deployment.

In [14]:
```
# ─── Cell 8: Save Trained Model with Proper Extension ──────────────────────
# TensorFlow requires a file extension; use the native Keras format (.keras)
MODEL_PATH = 'fashion_mnist_saved_model.keras'

# Save the model
model.save(MODEL_PATH)
```

```
print(f"▶ Model successfully saved to '{MODEL_PATH}'")
# Use: loaded_model = tf.keras.models.load_model(MODEL_PATH) to reload later
# ─── End of Updated Cell ──────────────────────────────────────────────
```

```
▶ Model successfully saved to 'fashion_mnist_saved_model.keras'
```

Step 9: Load Saved Model & Visualize Predictions

Demonstrate how to load the saved model from disk and verify it works correctly by showing the model architecture summary. Then display 5 random test images with their predicted vs. actual fashion category labels using human-readable category names to showcase model performance in action.Retry

In [15]:

```
# ─── Cell 9: Load Saved Model for Inference ──────────────────────────────────

# Define path to the saved model directory
MODEL_DIR = 'fashion_mnist_saved_model.keras'

# Load the model
trained_model = load_model(MODEL_DIR)
print(f"▶ Model loaded from '{MODEL_DIR}'")

# (Optional) Verify by showing its architecture
trained_model.summary()
# ─── End of New Cell ─────────────────────────────────────────────────────
```

```
▶ Model loaded from 'fashion_mnist_saved_model.keras'
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

Total params: 305,312 (1.16 MB)

Trainable params: 101,770 (397.54 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 203,542 (795.09 KB)

In [ ]:

```
# Visualize 5 random test images with Predicted vs. Actual labels (using names)
idxs = np.random.choice(len(X_test), 5, replace=False)

plt.figure(figsize=(10,2))

for i, idx in enumerate(idxs):
    # Reshape image for visualization
    img = X_test[idx].reshape(28,28)

    # Predict label for the image
    pred = np.argmax(trained_model.predict(img.reshape(1,784)), axis=1)[0]

    # Get predicted and true label names
    pred_name = LABEL_NAMES[pred]
    true_name = LABEL_NAMES[y_test[idx]]
```

```
    # Plot image with predicted vs true labels
    plt.subplot(1,5,i+1)
    plt.imshow(img, cmap='gray')
    plt.title(f"Pred: {pred_name}\nTrue: {true_name}")
    plt.axis('off')

plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
```