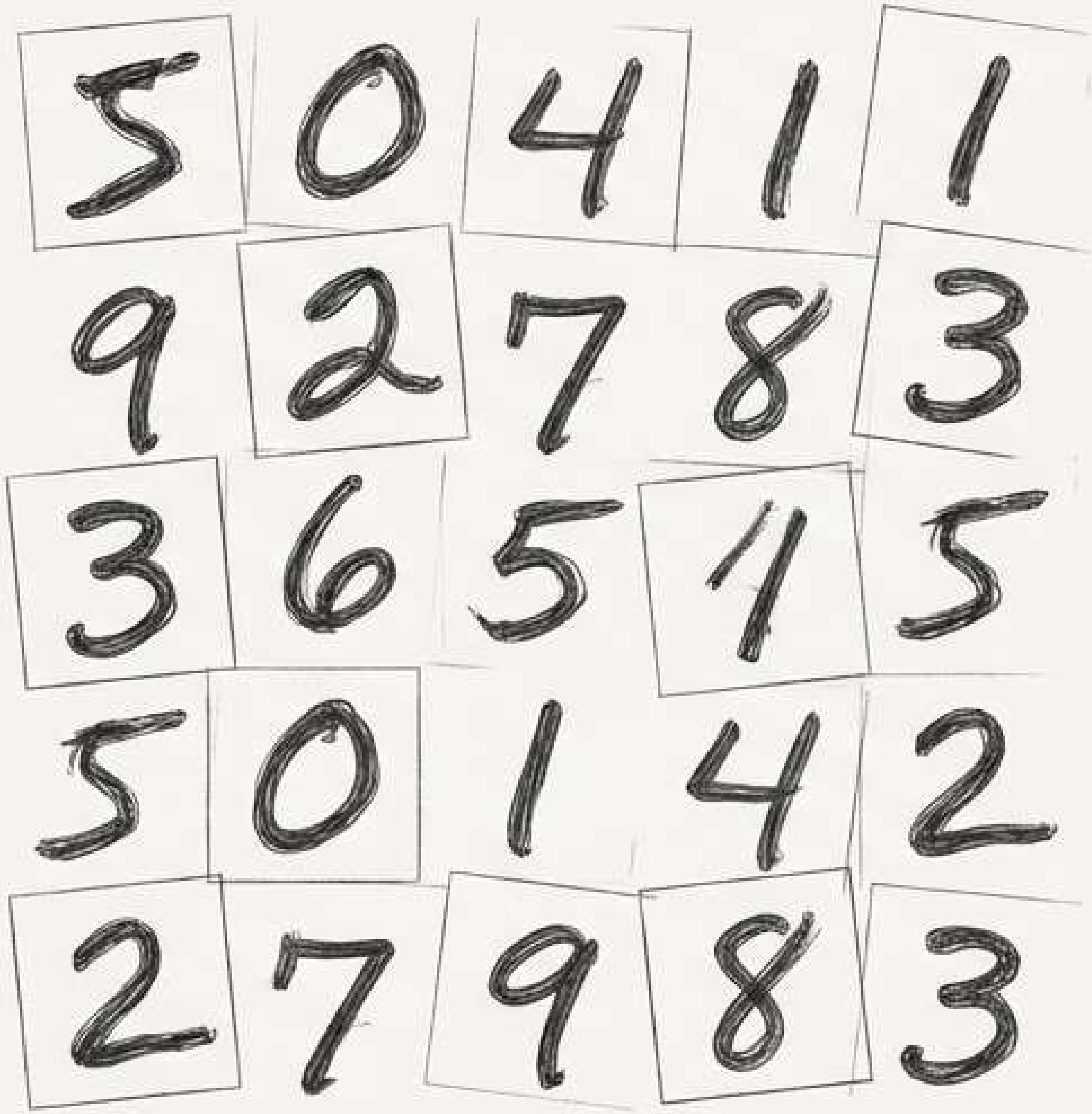
The 'Hello, World' of AI: A Magic Trick in 15 Lines

- **The problem:** Classify 28x28 pixel grayscale images of handwritten digits (0-9).
- **The tool:** Keras, a high-level deep learning library.
- **The dataset:** MNIST, a classic collection of 60,000 training images and 10,000 test images.



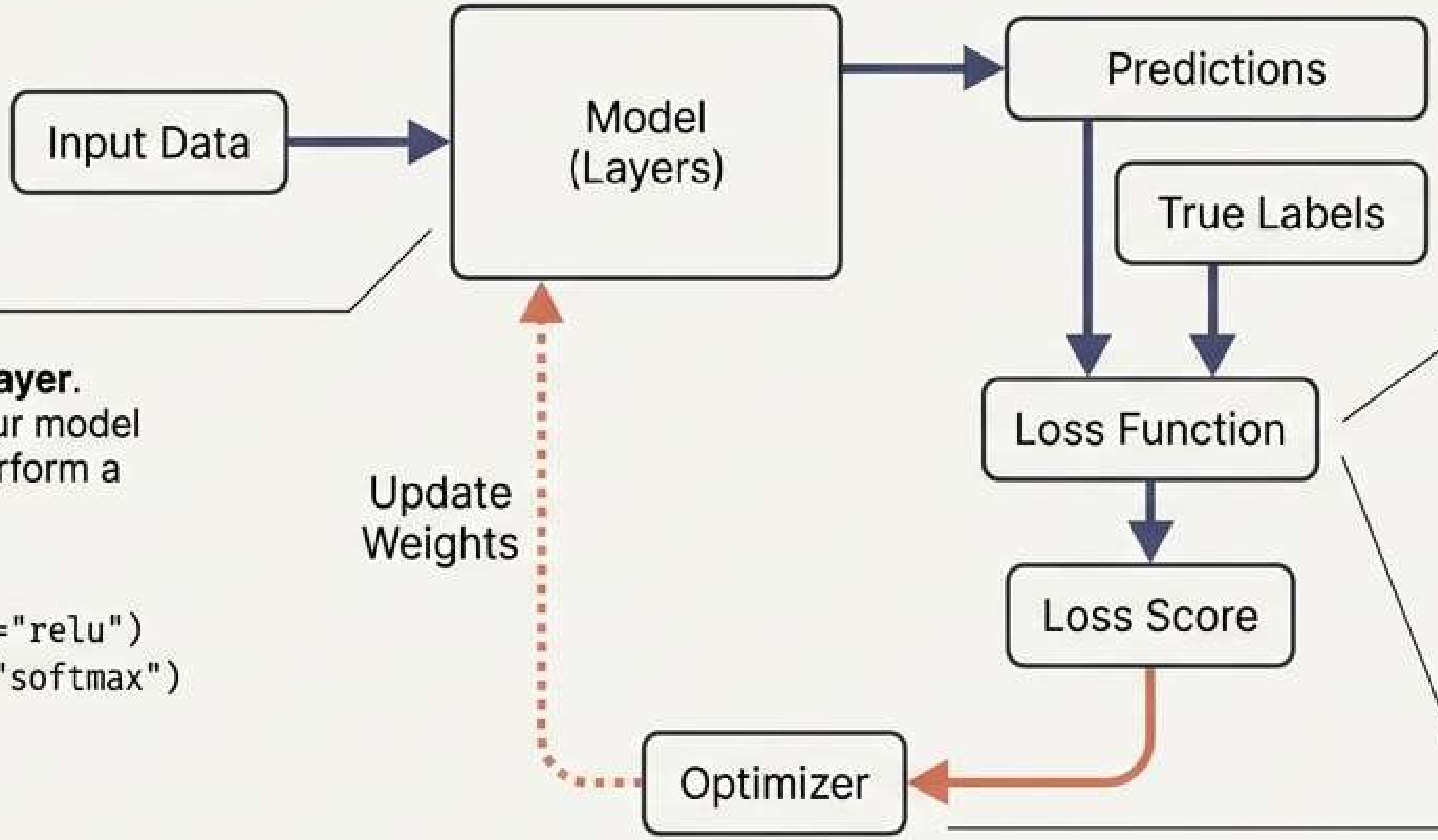
```python
import keras
from keras import layers
from keras.datasets import mnist

# Load and prep data (simplified for slide)
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28)).astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28)).astype("float32") / 255

# Build the model
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])

# Compile and train
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels, epochs=5, batch_size=128)

# Evaluate
test_loss, test_acc = model.evaluate(test_images, test_labels)
# >>> test_acc: 0.9785
```

Test Accuracy: 97.8%

The rest of this deck will explain exactly how these 15 lines achieve this.

NotebookLM

Anatomy of a Neural Network



1. The Network Architecture

The core building block is the **Layer**. Think of it as a filter for data. Our model chains two `Dense` layers to perform a progressive 'data distillation'.

```
layers.Dense(512, activation="relu")
layers.Dense(10, activation="softmax")
```

2. The Compilation Step

To make the model trainable, we need three more pieces:

Loss Function: `sparse_categorical_crossentropy` — Measures the model's performance. It's the feedback signal that steers the model in the right direction.

Optimizer: `adam` — The mechanism that updates the model based on the data it sees and its loss score.

Metrics: `accuracy` — What we monitor to judge performance.

3. The Training Loop

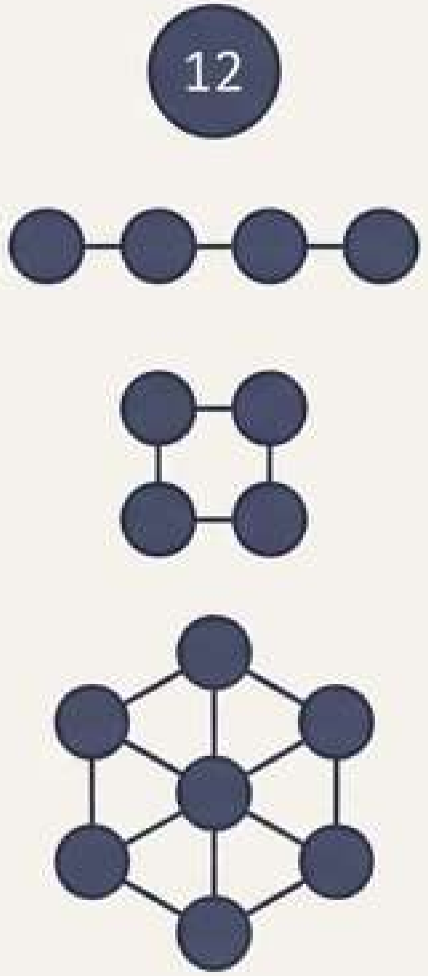
The `fit()` method executes the training. The model iterates on the data, learning to associate images with labels.

NotebookLM

Deconstructing the Magic, Part 1: The Ingredients

All data in neural networks is stored in Tensors.

At its core, a tensor is a container for numerical data. It's a generalization of matrices to an arbitrary number of dimensions (or "axes").



Scalar (Rank-0 Tensor)
`np.array(12)`

Vector (Rank-1 Tensor)
`np.array([12, 3, 6, 14])`

Matrix (Rank-2 Tensor)
`np.array([[5, 78], [6, 79]])`

Rank-3 Tensor

Connecting to the Example

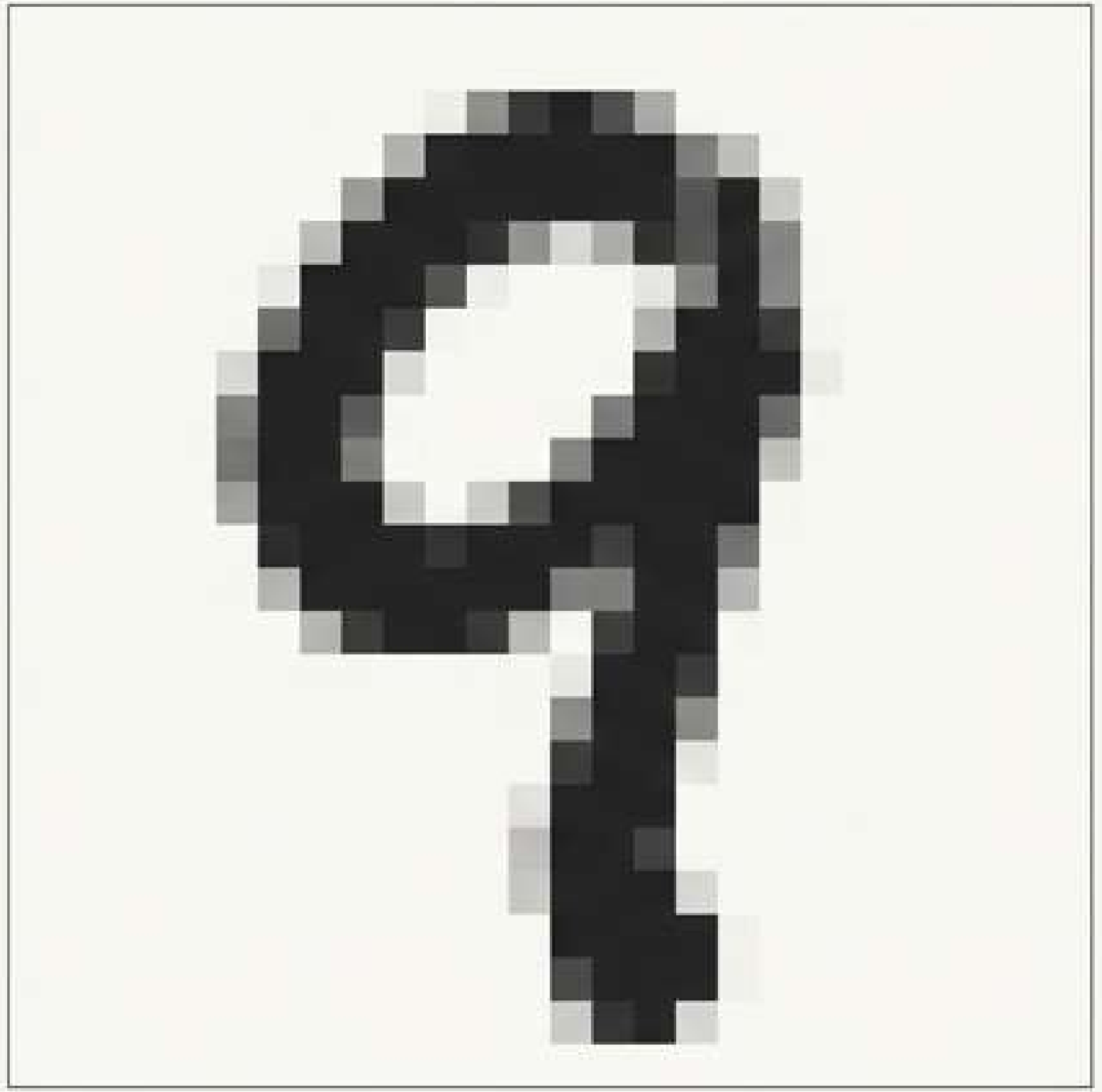
Let's look at our MNIST training data:

```
>>> train_images.shape
(60000, 28, 28)
```

This is a **Rank-3 Tensor**. It's a container holding 60,000 matrices, where each matrix is a 28x28 grid of integers representing a single handwritten digit.

NotebookLM

A Tensor's Key Attributes



```
import matplotlib.pyplot as plt
digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

1. **Number of Axes (Rank):** How many dimensions the tensor has.

```
>>> train_images.ndim
```

```
3
```

2. **Shape:** A tuple of integers describing the tensor's dimensions along each axis.

```
>>> train_images.shape
```

```
(60000, 28, 28)
```

(samples, height, width)

3. **Data Type (dtype):** The type of data contained in the tensor.

```
>>> train_images.dtype
```

```
uint8
```

(8-bit integers from 0 to 255)

NotebookLM

Deconstructing the Magic, Part 2: The Toolkit

All transformations learned by neural networks are chains of simple tensor operations.

A **layer** is a function that takes a tensor as input and returns a new, more useful representation of that tensor.

```
keras.layers.Dense(512, activation="relu")
```

This layer implements a simple but powerful formula:

```
output = relu( matmul(input, W) + b )
```

Let's unpack the three tensor operations at play:

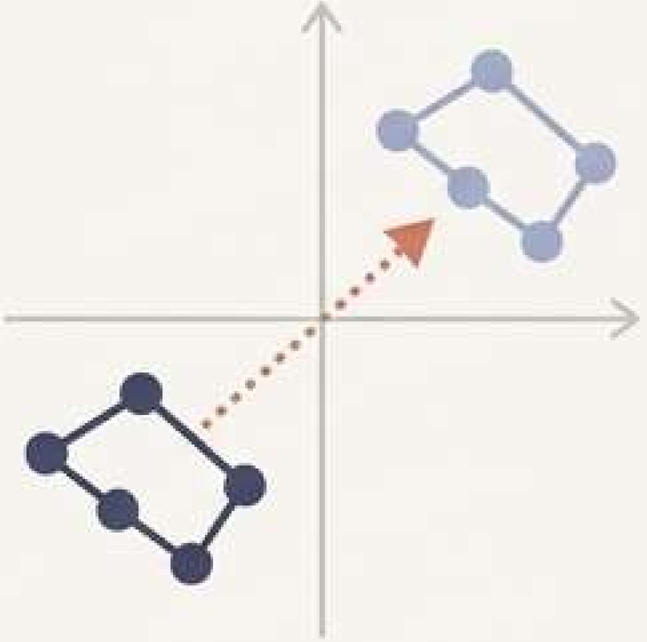
1. `matmul(input, W)`: A **tensor product** between the input data and the layer's internal weight matrix `W`.
2. `+ b`: An **addition** between that result and the layer's internal bias vector b.
3. `relu(...)`: An element-wise **activation function**. `relu(x)` is simply `max(x, 0)`.

NotebookLM

Tensor Operations are Geometric Transformations

Because tensors can represent points in a geometric space, tensor operations can be interpreted as geometric transformations of that space.

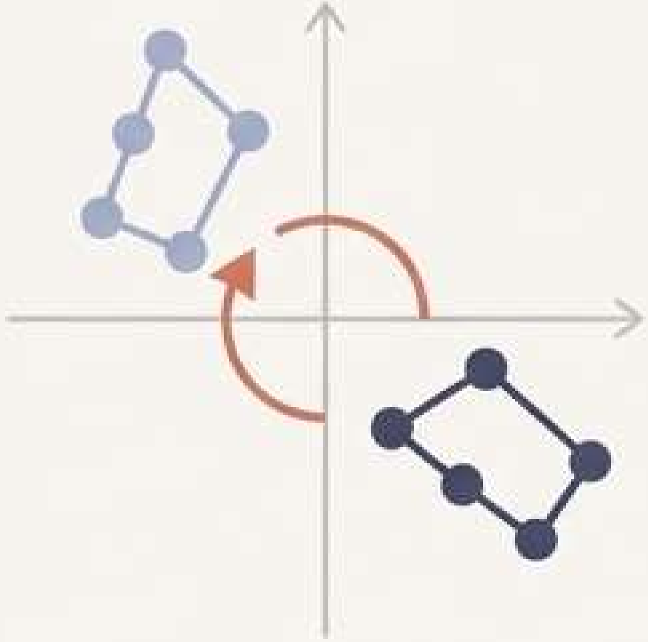
Translation



`point + translation_vector`

Adding a vector translates an object.

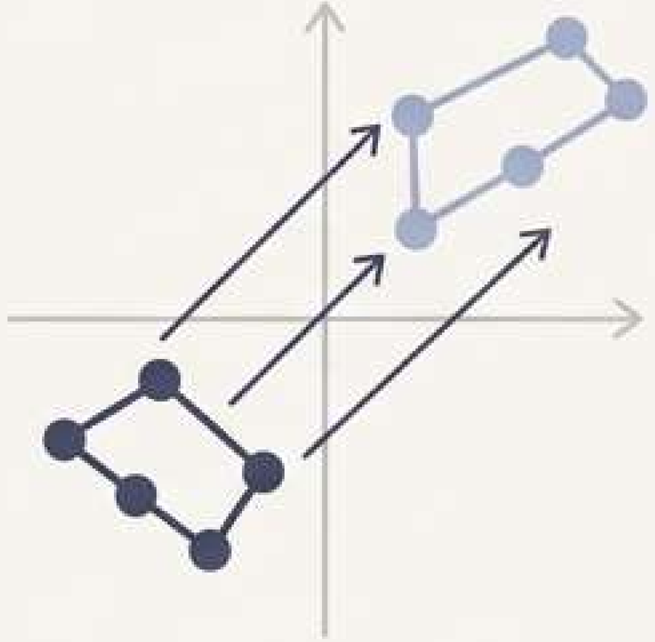
Rotation



`rotation_matrix @ point`

Multiplying by a specific matrix rotates an object.

Affine Transform



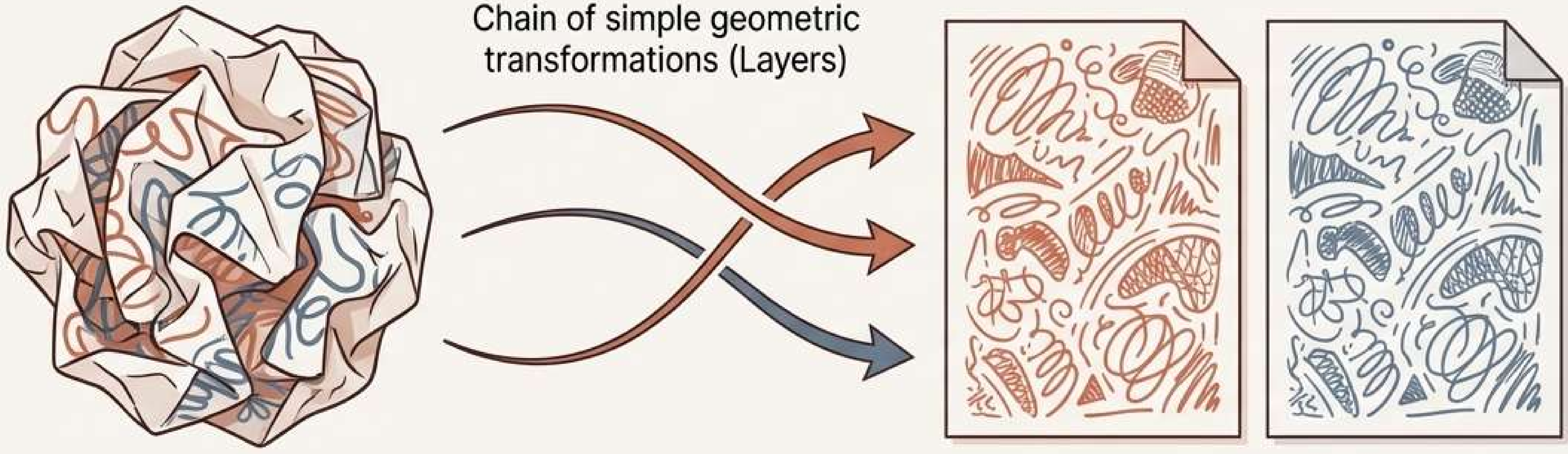
`W @ x + b`

A linear transform plus a translation.

A `Dense` layer without an activation function is just an **affine transform**. Chaining them together just results in another affine transform. This is why we need activation functions!

NotebookLM

The Goal of Deep Learning: Uncrumpling Data



Imagine your data is a crumpled paper ball, with each class (e.g., 'red' or 'blue') being a sheet of paper.

A neural network learns a complex geometric transformation to **uncrumple the ball**.

Each layer in a deep network applies one simple transformation that disentangles the data a little.
A deep stack of layers makes an extremely complicated disentanglement process tractable.

Machine learning is about finding neat representations for complex, highly folded data **manifolds**.

NotebookLM

Deconstructing the Magic, Part 3: The Engine

How a network adjusts its weights to learn from data.

Initially, the layer's weights (**W** and **b**) are filled with small random values. The network's output is meaningless.

Learning is the gradual adjustment of these weights based on a feedback signal. This happens in a **training loop.**

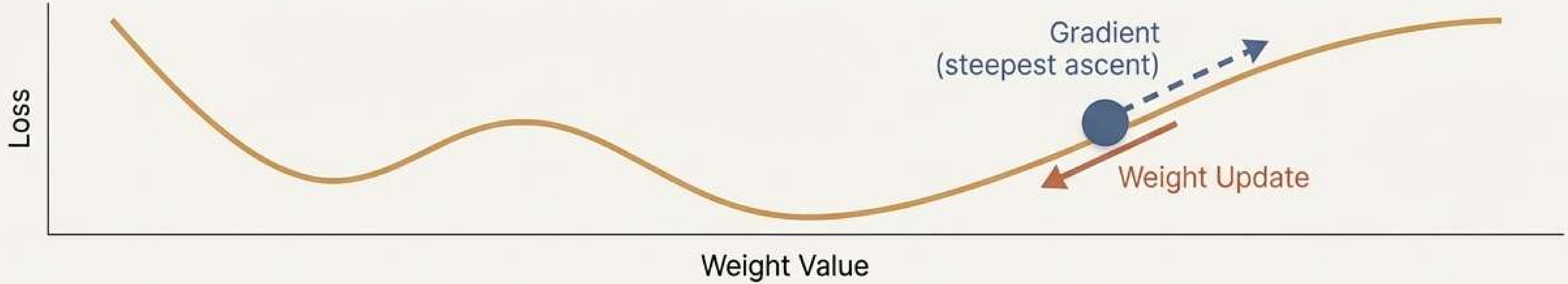
The Training Loop

1. Draw a **batch** of training samples **x** and corresponding targets y_true.
2. **Forward Pass**: Run the model on **x** to get predictions y_pred.
3. **Compute Loss**: Calculate the mismatch between y_pred and y_true.
4. **Update Weights**: Adjust the model's weights to slightly reduce the loss on this batch.

Step 4 is the key. How do we compute *how* to change each weight, and by how much? The answer is **Gradient Descent**.

NotebookLM

Finding the Bottom of the Hill with Gradient Descent



The Loss Surface

Imagine a surface where every possible set of weight values corresponds to a point, and the height of that point is the loss for the training data. Our goal is to find the lowest point on this surface.

What is a Gradient?

The functions in our network are "differentiable". This means we can compute a **gradient**. The gradient is a tensor that describes the **slope (or curvature) of the loss curvature) of the loss surface** at our current location. It points in the direction of steepest ascent.

How Gradient Descent Works

To reduce the loss, we simply move the weights a little in the **opposite direction of the gradient**. This is like taking a small step downhill.

$$W_{new} = W_{old} - \text{learning\_rate} * \text{gradient}$$

We don't do this for the entire dataset at once. We calculate the gradient on small, random batches of data (mini-batches). This is called **Mini-batch Stochastic Gradient Descent**.

NotebookLM

The Engine's Blueprint: Backpropagation

The Challenge: How do we efficiently calculate the gradient of the loss with respect to potentially millions of weights in a deep network?

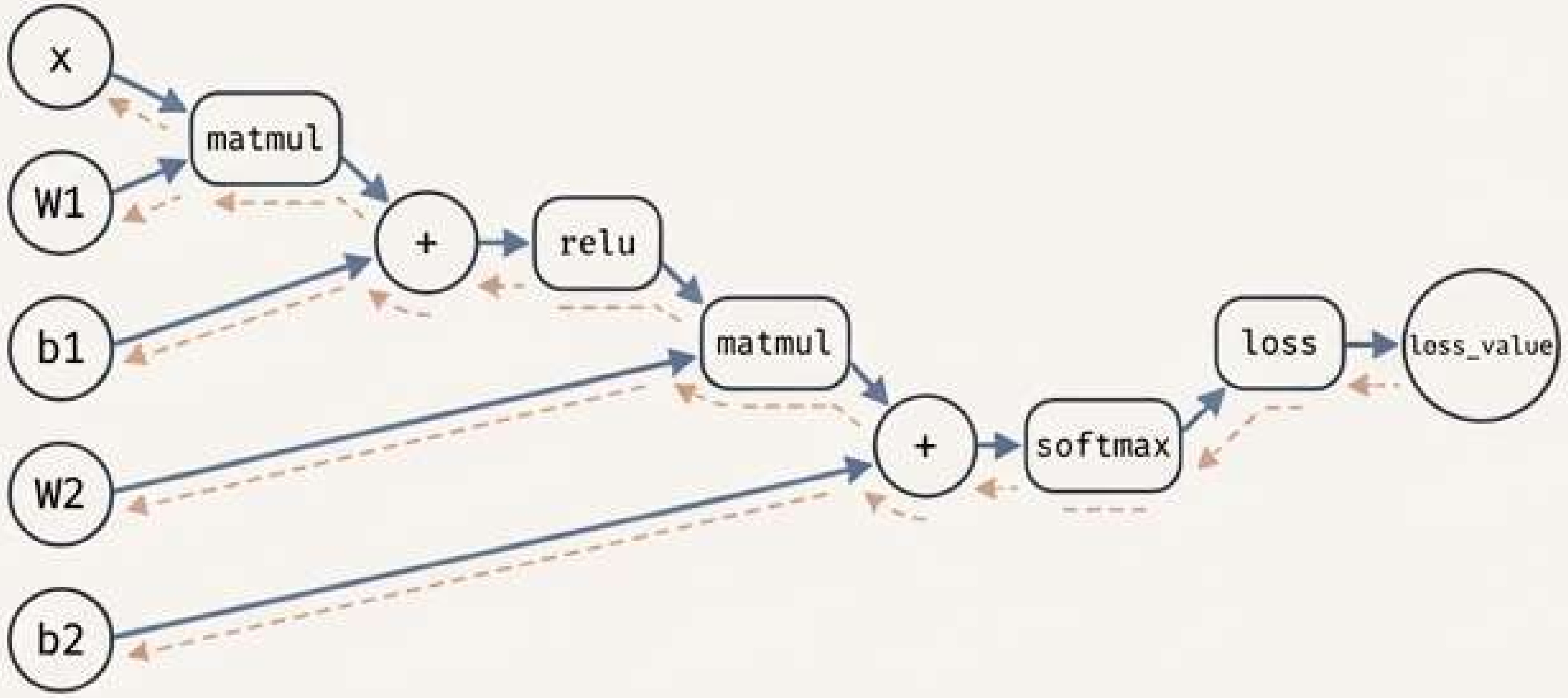
The Solution: The Chain Rule

Calculus gives us the **chain rule**, a way to compute the derivative of a chain of functions.

If `y = f(g(x))`, then the derivative of y with respect to x is `(derivative of y wrt g) * (derivative of g wrt x)`.

Backpropagation Explained

Backpropagation is simply the application of the chain rule to this computation graph. It starts with the final loss value and works **backward** through the graph, calculating the contribution that each parameter had to the final loss.



We can represent our network as a **computation graph** of chained operations.

Modern frameworks like TensorFlow and PyTorch perform this **automatic differentiation** for you. You'll never have to implement backpropagation by hand.

NotebookLM

The Magic Revealed: Let's Look Again

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])

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

model.fit(train_images, train_labels,
          epochs=5, batch_size=128)
```

Chaining together layers. Each layer is a geometric transformation of the data.

Implements `relu(matmul(input, W) + b)`. It learns a set of weights `W` and biases `b` to transform the data.

Configuring the learning process.

The specific algorithm for gradient descent (an advanced variant of SGD).

Defines the 'loss surface' we want to descend.

Executes the training loop: forward pass -> compute loss -> backward pass (backpropagation) -> update weights. Repeats for 5 epochs over mini-batches of 128 images.

NotebookLM

From Magic to Mastery: Building It Ourselves

To prove we understand, let's reimplement a simplified version of our model from scratch.

A Simple Dense Layer

```
class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        # ... initialize W and b as trainable variables ...

    def __call__(self, inputs):
        # The core transformation
        return self.activation(ops.matmul(inputs, self.W) +
self.b)
```

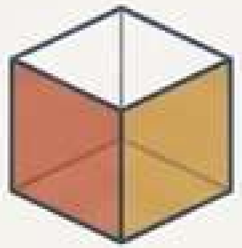
A Single Training Step

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        # 1. Forward pass
        predictions = model(images_batch)
        # 2. Compute loss
        loss = ops.mean(ops.sparse_categorical_crossentropy(
            labels_batch, predictions))
    # 3. Compute gradients (Backpropagation)
    gradients = tape.gradient(loss, model.weights)
    # 4. Update weights
    update_weights(gradients, model.weights)
    return loss
```

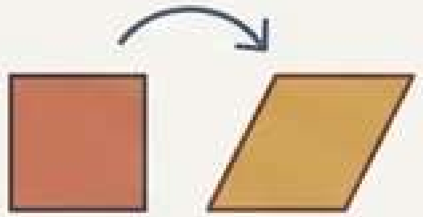
While high-level frameworks like Keras are incredibly powerful, understanding the low-level mechanics—tensors, operations, and gradients—is what separates a user from a practitioner.

NotebookLM

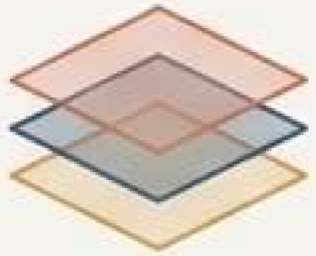
The Core Ideas of Deep Learning



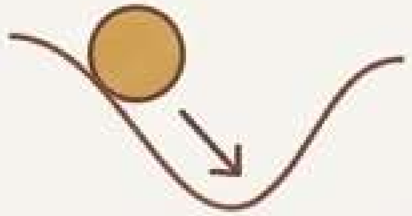
Tensors: The fundamental data structures of machine learning. They are multidimensional arrays that hold the data.



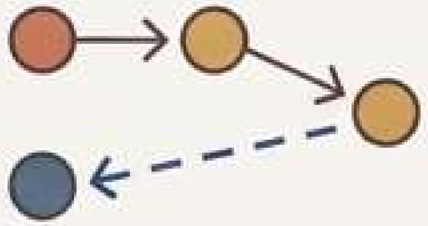
Tensor Operations: The building blocks of models. Simple functions like `matmul`, `+`, and `relu` that geometrically transform tensors.



Layers: Modules that chain tensor operations together to perform useful data transformations. A deep model is a deep stack of these layers.



Gradient Descent: The optimization "engine" that allows a model to learn. It iteratively adjusts model weights by stepping down the 'slope' of a loss function.



Backpropagation: The algorithm that makes gradient descent efficient. It applies the chain rule to a computation graph to calculate the gradient for all model weights.

NotebookLM