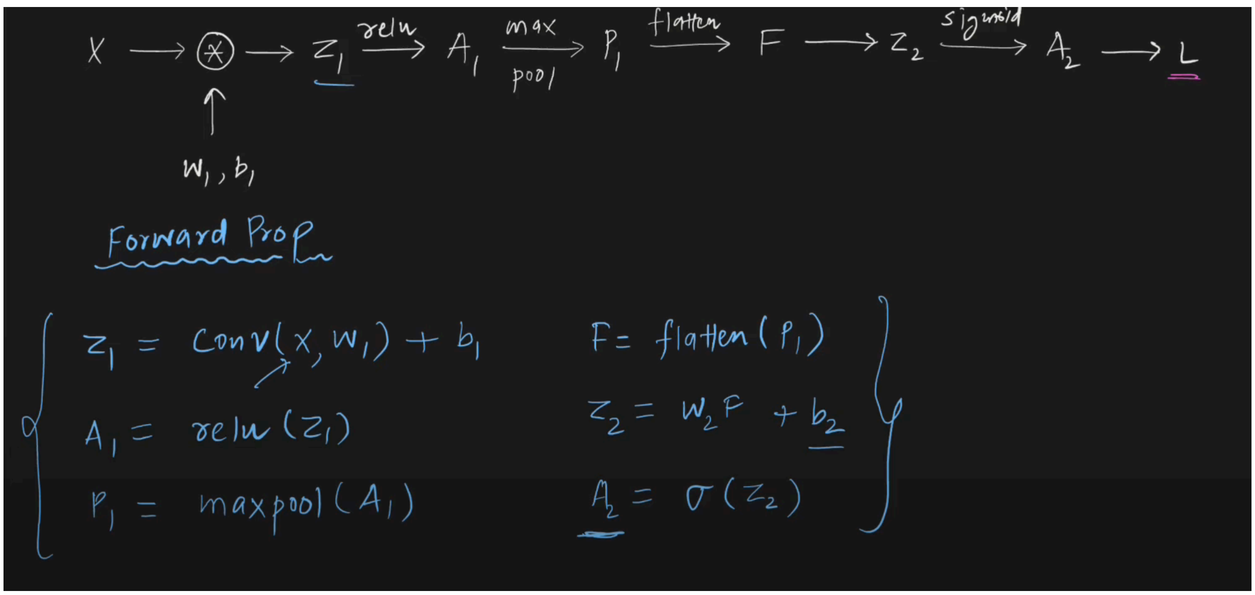
Backpropagation in CNN

In CNN (or any neural network), backpropagation is the **core learning algorithm** that:

1. **Finds the error** in prediction
2. **Traces it backward** to each layer
3. **Adjusts the weights** so that the error becomes smaller next time



🔁 Backpropagation = 2 Steps

Step Name	What Happens
Forward Pass	Input flows through CNN, predicts output
Backward Pass	Error is sent **backward** to update weights

👉Backpropagation only updates weights in:

- **Convolution layers** (filters)
- **Dense layers** (neurons)

> 📌 Pooling layers do **not** have weights, so they **just pass gradients backward**.

🔬 Mathematically, What Happens?

Forward:

```
Output = Activation(Weighted_Sum_of_Inputs)
```

Backward:

```
Weight = Weight - LearningRate × Gradient_of_Error
```

- The “gradient” tells the **direction** and **size** of change needed to reduce the error.

Steps:

1. **Forward Pass**
    - Input flows forward
    - Filters extract features
    - Final Dense layer makes prediction
2. **Calculate Loss (Error)**
    - Compare prediction vs actual (e.g., using cross-entropy)
3. **Backward Pass (Backpropagation)**

    🔁 From output to input:

    - Output Layer → update dense weights
    - Dense → send gradients to Flatten
    - Conv Layer 2 → update filters
    - Conv Layer 1 → update filters

    - Pooling layers: pass gradients only (no update)
4. **Update Weights**
    - Using gradients and learning rate

📦 Example with Keras

Keras handles backprop **automatically**:

```
model.compile(optimizer='adam', loss='categorical_crossentropy')
model.fit(X_train, y_train)
```

Here:

- `loss`: used to compute the error
- `optimizer`: applies backprop and updates weights

Behind the scenes:

- TensorFlow computes all gradients
- Applies backpropagation
- Adjusts weights layer-by-layer

Key Concepts in CNN Backpropagation

(A) Chain Rule

Backpropagation uses the **chain rule** to compute gradients layer by layer:

$$\frac{\partial \text{Loss}}{\partial W} = \frac{\partial \text{Loss}}{\partial \text{Output}} \cdot \frac{\partial \text{Output}}{\partial W}$$

(B) CNN-Specific Challenges

1. **Weight Sharing**:

- The same kernel (filter) is applied across the entire image → gradients are summed over all locations.

2. **Pooling Layers**:
    - Max Pooling only backpropagates gradients to the **winning neuron** (the one with the max value).
    - Average Pooling distributes gradients equally.
3. **Convolutional Layers**:
    - Gradients are computed for both **kernels** and **input feature maps**.

Backpropagation Steps in CNNs

Step 1: Forward Pass

Compute predictions using:

- Convolutions (`Conv2D`).
- Pooling (`MaxPooling2D`).
- Fully connected layers (`Dense`).

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(10, activation='softmax')
])
```

Step 2: Compute Loss

Compare predictions (`y_pred`) with true labels (`y_true`):

$$\text{Loss} = \text{CategoricalCrossentropy}(y_{\text{true}}, y_{\text{pred}})$$

Step 3: Backward Pass (Gradient Calculation)

(A) Fully Connected (Dense) Layer Gradients

- Standard backpropagation (like in regular neural networks):

$$\frac{\partial \text{Loss}}{\partial W} = \text{Error} \times \text{Activation}^T$$

(B) Convolutional Layer Gradients

1. **Gradient w.r.t. Kernels (Filters)**:
    - Sum gradients over all input locations where the kernel was applied.

$$\frac{\partial \text{Loss}}{\partial K} = \text{Input Patch} \times \text{Upstream Gradient}$$

2. **Gradient w.r.t. Input Feature Maps**:
    - Propagate gradients back to the input using **full convolution** (flipped kernel).

$$\frac{\partial \text{Loss}}{\partial X} = \text{Upstream Gradient} \star \text{Rot180}(K)$$

(where `⋆` is cross-correlation and `Rot180` flips the kernel 180°).

(C) Pooling Layer Gradients

- **Max Pooling**: Only the neuron with the max value gets the gradient.
- **Average Pooling**: Distributes gradient equally to all neurons in the pooling window.

```
# Max Pooling Gradient (Pseudocode)
if neuron_was_max:
    gradient = upstream_gradient
else:
    gradient = 0
```

Step 4: Update Weights

Use **optimizers** (e.g., SGD, Adam) to adjust weights:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \frac{\partial \text{Loss}}{\partial W}$$

Keras Implementation (Automatic Backpropagation)

You don't need to manually implement backprop—Keras handles it:

```
model.compile(optimizer='adam', loss='categorical_crossentropy')
model.fit(X_train, y_train, epochs=10)
```

❌ Disadvantages / Challenges

Disadvantage	Why it Happens
❗ Vanishing gradients	Small gradients → weights stop updating
❗ Overfitting	Learns too well → performs badly on new data
❗ Computational cost	Backprop on large CNNs = expensive
❗ Sensitive to learning rate	Too high → unstable, too low → very slow
❗ Needs labeled data	Supervised learning needs correct answers