

CNN: Plain vs. Batch Normalized

Convolutional networks for MNIST digit classification

Functional API, Conv2D internals, and why Batch Normalization changes everything

01 Why Convolutions Instead of Dense Layers

The MLP notebooks flattened images into 784-length vectors, throwing away all spatial structure -- a pixel at position (5,5) is treated no differently than one at (5,500) (if the image were that big). **Convolutional layers** instead keep the 2D grid shape and slide small filters across it, learning patterns that respect spatial locality -- edges, corners, strokes -- exactly the kind of structure handwritten digits actually have.

Data preparation difference vs. the MLP notebooks

```
X_train = train_images.reshape(60000, 28, 28, 1) # keep 2D shape + channel dim
X_test = test_images.reshape(10000, 28, 28, 1)
```

Instead of flattening to (784,), images are reshaped to **(28, 28, 1)** -- height, width, and a single channel (grayscale; would be 3 for RGB). Conv2D layers require this 3D shape per image.

02 The Functional API (vs. Sequential)

Both CNN notebooks use Keras' **Functional API** rather than Sequential -- each layer is called directly on the previous layer's output, like function composition:

```
myInput = layers.Input(shape=(28,28,1))
conv1 = layers.Conv2D(32, 3, activation='relu', padding='same', strides=2)(myInput)
conv2 = layers.Conv2D(64, 3, activation='relu', padding='same', strides=2)(conv1)
flat = layers.Flatten()(conv2)
out_layer = layers.Dense(10, activation='softmax')(flat)
mymodel = models.Model(myInput, out_layer)
```

DEEP DIVE -- Why Functional API here (and when you NEED it)

Sequential works fine for a simple stack like this one. The BatchNorm version uses Functional too, mostly for consistency, but Functional becomes **necessary** the moment you need multiple inputs/outputs, skip connections (like ResNet), or branching architectures. Each layer is a callable that takes a tensor and returns a new tensor -- this is the foundation for building anything more complex than a straight stack.

03 Conv2D Parameters -- What Each One Controls

```
layers.Conv2D(32, 3, activation='relu', padding='same', strides=2)
```

Argument	Value	What it controls
filters	32 (first), 64 (second)	Number of distinct learned feature detectors -- each filter learns to recognise a different pattern (edge, curve, etc.)
kernel_size	3	Each filter is a 3x3 window -- small enough to capture local patterns like edges and curves
activation	relu	Same as in the MLP -- introduces nonlinearity, zeroes negative values
padding	'same'	Pads the input with zeros so the output keeps roughly the same spatial size as the input (instead of shrinking at every layer)
strides	2	The filter window jumps 2 pixels at a time instead of 1 -- this halves the spatial resolution at each layer (a form of downsampling, like pooling but built into the conv)

DEEP DIVE -- How a filter actually 'sees' an image

A 3x3 filter is a small grid of learnable weights. It slides across the image, and at each position it computes a **weighted sum** of the 9 pixels under it (a dot product), producing a single output value. Sliding this across the whole image produces a 2D 'feature map' showing where that pattern was detected.

With 32 filters in the first Conv2D, you get 32 different feature maps -- 32 different patterns being searched for simultaneously, all learned automatically from data, not hand-designed.

Why strides=2 instead of strides=1 + pooling? Both approaches downsample (reduce spatial size), but strided convolutions fold the downsampling into the same operation that's already learning features, slightly more parameter-efficient than a separate MaxPooling layer.

04 THE KEY DIFFERENCE -- Batch Normalization

Side-by-side: what changed

Plain CNN	BatchNorm CNN
<code>Conv2D(32,3, activation='relu', ...)(myInput)</code>	<code>Conv2D(32,3, ...)(myInput) -- NO activation here</code> <code>BatchNormalization()(conv1) Activation('relu')(bn1)</code>

The BatchNorm version splits what used to be one step (Conv2D with activation built in) into three explicit steps: **convolve -> normalize -> activate**. This reordering is deliberate and important.

DEEP DIVE -- What Batch Normalization actually does, mathematically

For each batch of training examples, BatchNorm takes the raw outputs of the conv layer (before activation) and rescales them so that, across the current batch, they have **mean 0 and standard deviation 1**:

```
x_normalized = (x - batch_mean) / sqrt(batch_variance + epsilon)
```

Then it applies two small learnable parameters, gamma (scale) and beta (shift), so the network can still shift/stretch the normalized values if that helps:

```
output = gamma * x_normalized + beta
```

Why normalize between layers at all? As training progresses, the distribution of activations flowing into each layer keeps shifting (because earlier layers' weights are also changing) -- this is called **internal covariate shift**. Each layer has to constantly re-adapt to a moving target, which slows training and can make deep networks unstable. BatchNorm keeps that distribution stable, letting you train faster and with less sensitivity to weight initialization and learning rate choice.

Why activation comes AFTER BatchNorm, not before

The order convolve -> normalize -> activate is the standard, recommended pattern. BatchNorm normalizes the raw linear output of the conv layer; applying ReLU afterward then decides which of those normalized values to keep (positive) or zero out (negative). Putting BatchNorm after the activation would normalize an already-clipped (half-zeroed) distribution, which defeats some of its purpose.

Train-time vs. test-time behavior (an important subtlety)

During training, BatchNorm uses the **current batch's** mean and variance. But at test time, you might predict on a single image -- there's no 'batch' to compute statistics from. So Keras automatically tracks a **running average** of mean/variance during training and uses those fixed values at inference time. This happens automatically; you don't need to manage it yourself, but it's worth knowing it's happening under the hood -- similar to how Dropout also behaves differently between train and test.

05 Compiling and Training (identical in both)

```
mymodel.compile(
    optimizer=optimizers.SGD(learning_rate=0.001),
    loss=losses.categorical_crossentropy
)

history = mymodel.fit(X_train, y_train, batch_size=128, epochs=50, validation_split=0.2)
```

Same optimizer, same learning rate, same training configuration as the plain CNN -- the only architectural difference is the BatchNorm + separated Activation layers. This makes it a clean controlled comparison: any difference in training speed or stability you observe in the loss curves is attributable to BatchNorm alone.

06 Summary -- Plain CNN vs. BatchNorm CNN

Aspect	Plain CNN	BatchNorm CNN
Conv block	Conv2D(activation='relu')	Conv2D -> BatchNormalization -> Activation('relu')
Internal covariate shift	Not addressed	Actively normalized between layers
Typical effect on training	Can be slower / more sensitive to learning rate	Usually faster, more stable convergence
Extra learnable params	None	gamma and beta per BatchNorm layer (small overhead)
Test-time behavior	Same as training	Uses running average statistics instead of batch statistics

What to compare in your two loss plots

Look at how quickly loss drops in the first 5-10 epochs. BatchNorm models typically show a steeper initial drop and less jagged validation loss, since the optimization landscape each layer sees is more consistent from batch to batch. If both curves look almost identical, it may mean the model is small/simple enough (only 2 conv layers here) that the benefit of BatchNorm is modest -- it shows up more dramatically in much deeper networks (10+ layers).