

MNIST MLP: Plain vs. Dropout

Comparing two fully-connected networks

What changed, why it matters, and what every parameter does

01 Setup -- Data Preparation (identical in both)

Loading and flattening MNIST

Both notebooks load the same MNIST dataset (60,000 train / 10,000 test images of handwritten digits, 28x28 pixels). Since a plain Dense (fully-connected) layer expects a 1D vector input, each image is **flattened** from a 28x28 grid into a 784-length vector ($28 \times 28 = 784$).

```
X_train = train_images.reshape(60000, 784)
X_test = test_images.reshape(10000, 784)
```

Normalizing pixel values

Raw pixel values range 0-255. Dividing by 255 rescales them to 0-1. Neural networks train faster and more stably on small, centered input ranges -- large raw pixel values can cause unstable gradients.

```
X_train = X_train.astype('float') / 255
```

One-hot encoding the labels

Labels are digits 0-9 (a single integer). For a softmax output layer with categorical cross-entropy loss, labels need to be **one-hot vectors** -- e.g. digit 3 becomes [0,0,0,1,0,0,0,0,0]. `utils.to_categorical()` does this conversion automatically.

02 Model Architecture -- Plain MLP

```
mymodel = models.Sequential()
mymodel.add(layers.Dense(500, activation='relu', input_shape=(784,)))
mymodel.add(layers.Dense(100, activation='relu'))
mymodel.add(layers.Dense(10, activation='softmax'))
```

What each layer does

Layer	Units	Activation	Role
Dense #1	500	ReLU	First hidden layer -- learns low-level pixel patterns
Dense #2	100	ReLU	Second hidden layer -- combines patterns into higher-level features
Dense #3	10	Softmax	Output layer -- one probability per digit class, summing to 1

DEEP DIVE -- Why ReLU, and why softmax only at the end

ReLU(x) = max(0, x) -- it passes positive values through unchanged and zeroes out negatives. It's used in hidden layers because it's cheap to compute and doesn't suffer from the 'vanishing gradient' problem that older activations (sigmoid, tanh) have in deep networks -- its gradient is exactly 1 for any positive input, so signal doesn't shrink as it flows backward through many layers.

Softmax is reserved for the output layer because it converts raw scores into a true probability distribution (all values between 0 and 1, summing to exactly 1) -- exactly what's needed to interpret the output as 'P(digit is 0), P(digit is 1), ... P(digit is 9)'.

Why 500 then 100? (the funnel shape)

Going from 784 inputs to 500 to 100 to 10 outputs is a common 'funnel' pattern: each layer compresses information into fewer, more abstract features. There's no hard rule for these exact numbers -- they're hyperparameters found by experimentation -- but the general principle (start wide, narrow toward the output) tends to work well for classification.

03 Compiling the model

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

Argument	Value	Meaning
optimizer	SGD(lr=0.001)	Stochastic Gradient Descent -- updates weights using small steps in the gradient's negative direction
learning_rate	0.001	Step size per update -- small enough to avoid overshooting, but means slower convergence
loss	categorical_crossentropy	Standard multiclass loss -- penalises confident wrong predictions much more than uncertain ones

04 Training -- model.fit() arguments

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

Argument	Value	Meaning
batch_size	128	Number of samples processed before each weight update (mini-batch gradient descent)
epochs	50	Number of full passes through the entire training set
validation_split	0.2	Holds out 20% of training data (not used for weight updates) to monitor generalisation each epoch

What 'history' captures

fit() returns a History object recording loss (and validation loss) at the end of every epoch. This is what gets plotted afterward to see whether the model is still improving, plateauing, or overfitting.

05 THE KEY DIFFERENCE -- Dropout

What changed in the dropout version

```
mymodel.add(layers.Dense(500, activation='relu', input_shape=(784,)))
mymodel.add(layers.Dropout(0.2)) # NEW
mymodel.add(layers.Dense(100, activation='relu'))
mymodel.add(layers.Dropout(0.2)) # NEW
mymodel.add(layers.Dense(10, activation='softmax'))
```

A **Dropout(0.2)** layer is inserted after each hidden Dense layer. During training, it randomly 'turns off' (zeroes out) 20% of that layer's neurons for each batch -- a different random 20% every time.

DEEP DIVE -- Why randomly disabling neurons helps

Without dropout, neurons can become **co-dependent** -- neuron A learns to only work correctly if neuron B produces a specific value, and the network ends up memorising fragile, overly-specific patterns from the training data (overfitting).

By randomly dropping different neurons every batch, no single neuron can rely on any other specific neuron always being present. This forces each neuron to learn features that are **independently useful**, which generalises better to unseen data.

Important: dropout is automatically turned OFF during evaluation/prediction (`model.evaluate()` and `model.predict()`) -- Keras handles this for you. All neurons are active at test time; dropout is purely a training-time regularisation trick.

The 0.2 parameter is the fraction of neurons dropped, not kept. 0.2 = 20% dropped, 80% kept per batch. Common values range 0.2-0.5; higher dropout = stronger regularisation but risks underfitting if too aggressive.

The other change: learning rate

Notebook	Optimizer	Learning rate
Plain MLP	SGD	0.001
Dropout MLP	SGD	0.01 (10x higher)

Since dropout adds noise/regularisation and effectively makes the network 'fight itself' a bit during training, a slightly higher learning rate (0.01 vs 0.001) is reasonable to compensate and keep training moving at a similar pace despite the added regularisation.

06 Evaluation and Visualizing Misclassifications

Both notebooks finish identically: evaluate test loss, get predictions, convert softmax outputs to class labels with **`np.argmax()`** (picks the index of the highest probability), then loop through test images to find and display the first 10 the model got wrong.

```
test_labels_p = mymodel.predict(X_test)
test_labels_p = np.argmax(test_labels_p, axis=1) # softmax probs -> class index

# Loop finds first 10 mismatches between predicted and true label
if (test_labels_p[i] != test_labels[i]):
    axs[n].imshow(test_images[i], cmap='gray')
    axs[n].set_title(f'{test_labels[i]} -> {test_labels_p[i]}')
```

axis=1 explained

`predict()` returns a (10000, 10) array -- one row per test image, 10 probabilities per row. **axis=1** tells `argmax` to find the max along each row (across the 10 class probabilities), not down each column. The result is a (10000,) array of predicted digit labels.

07 Summary -- Plain MLP vs. Dropout MLP

Aspect	Plain MLP	Dropout MLP
Architecture	784->500->100->10	Same + Dropout(0.2) after each hidden layer
Regularization	None	Dropout -- randomly zeroes 20% of neurons per batch
Learning rate	0.001	0.01
Overfitting risk	Higher on more epochs	Lower -- validation loss should track training loss more closely
Test-time behavior	Same	All neurons active (dropout disabled automatically)

What to look for in your loss plots

In the plain MLP, watch for the validation loss curve starting to rise while training loss keeps falling -- that's the overfitting signature. In the dropout version, the gap between train and validation loss should be noticeably smaller, even if both numbers are slightly higher than the plain model's training loss alone.