MNIST Digit Classification + Interactive Draw→Predict Demo (Keras/TensorFlow)

I'm building this blog as a portfolio of machine learning engineering work. This post documents an end-to-end project we created in this chat: a **handwritten digit classifier trained on MNIST**, implemented in **Keras** with a **TensorFlow** backend, plus an interactive **"draw a digit with your mouse → run inference"** canvas embedded directly inside a Google Colab notebook.

The point is not "MNIST is hard." It is not. The point is that it is a compact environment where I can demonstrate the full lifecycle:

- data ingestion and preprocessing,
- model definition and training,
- disciplined evaluation on held-out data,
- error analysis (confusion matrix, precision/recall/F1),
- model persistence (save/reload),
- and a user-facing inference interface with real distribution shift issues.

If you want a single sentence summary: **I trained a deep learning model and then treated inference as a product problem.**

---

1. What We Built (Systems View)

The Colab notebook contains two "planes" of functionality.

1.1 Training plane (offline learning)

1. Load MNIST from `tensorflow.keras.datasets.mnist`.
2. Preprocess:
   - normalize pixel intensities to $[0, 1]$,
   - reshape into the input form the model expects,
   - one-hot encode the labels for cross-entropy.
3. Define a neural network classifier in Keras.
4. Train with minibatches, track training and validation curves.
5. Evaluate once on a held-out test set.
6. Report a confusion matrix and a classification report.
7. Save the model artifact and verify it reloads cleanly.

1.2 Inference plane (interactive)

1. Render an HTML canvas in the notebook output.
2. Capture the user's drawing as a PNG (base64 data URL).
3. Ship it back to Python via Colab's callback bridge.
4. Preprocess the drawing into an MNIST-like 28×28 image:
   - crop, resize, pad, center-of-mass alignment,

- normalize,
- reshape to the model's input tensor.

5. Run `model.predict` and show:
   - predicted class,
   - top-k probabilities,
   - the processed 28×28 image that the model actually saw.

That last bullet is the engineering punchline: **show the model input**, not just the UI drawing. Most inference failures are input-contract failures.

---

2. Data: MNIST and the Learning Problem

MNIST consists of 70,000 grayscale images of handwritten digits:

- Training set: 60,000 samples
- Test set: 10,000 samples (held out)

Each input is $x \in \mathbb{R}^{28\times 28}$ and label $y \in \{0, \ldots, 9\}$. We learn a conditional distribution $p_\theta(y \mid x)$ parameterized by θ.

From the standpoint of statistical learning, training is maximum likelihood:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{N} p_\theta(y_i \mid x_i)$$

Equivalently, we minimize negative log likelihood (NLL):

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \left(-\log p_\theta(y_i \mid x_i) \right)$$

The practical implementation is **softmax cross-entropy**.

2.1 Dataset splits and leakage discipline

A model's job is to generalize. The only honest measure of generalization is performance on data that was not used for training or hyperparameter selection.

Split	What it's for	What I use it for
Training	Parameter learning	Gradient updates
Validation	Model selection	Monitoring curves, early stopping decisions
Test	Final estimate	Once, at the end, for reporting

A common error is to "peek" at the test set while tuning. That converts the test set into an implicit validation set and biases performance upward.

3. From Pixels to Tensors: Preprocessing

The baseline model in the notebook is a dense network, so the image must be vectorized:

$$\mathrm{vec}(x) \in \mathbb{R}^{784}$$

3.1 Normalization

Original pixels are integers in $[0, 255]$. We scale:

$$x = \frac{x}{255}$$

This improves numerical conditioning and gradient scale. Without normalization, the network can still learn, but optimization is less stable and learning rate sensitivity increases.

3.2 One-hot labels

We represent labels as $\mathbf{y} \in \{0, 1\}^{10}$ so cross-entropy can be written compactly:

$$\mathbf{y} = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0) \quad \text{for digit } 3.$$

4. Neural Networks: Theory and Mechanics

A feedforward neural network is a composition of affine maps and nonlinearities. For layer ℓ:

$$a^{(\ell)} = W^{(\ell)} h^{(\ell-1)} + b^{(\ell)}, \qquad h^{(\ell)} = \phi^{(\ell)}(a^{(\ell)})$$

with $h^{(0)} = x$. The final layer produces logits $z \in \mathbb{R}^{10}$. Softmax maps logits to a probability simplex.

4.1 Expressivity, depth, and inductive bias

Universal approximation results tell us "a sufficiently wide network can approximate many functions." But engineering practice cares about:

- **sample efficiency** (how much data is required),
- **generalization** (how well it performs on unseen data),
- **optimization** (whether it trains at all).

Architectures that encode the right inductive bias learn more efficiently. CNNs encode locality and translation structure; dense networks do not. This is why CNNs dominate vision even when dense nets can, in principle, represent the same functions.

4.2 Activation functions and gradient flow

Training deep networks is a gradient propagation problem. Saturating nonlinearities like sigmoid can cause vanishing gradients when $|x|$ is large. ReLU avoids saturation for positive inputs and yields piecewise-linear behavior, which is generally easier to optimize.

The derivative matters:

- Sigmoid: $\sigma(x) = \sigma(x)(1 - \sigma(x))$ (can be tiny)
- ReLU: $\mathrm{ReLU}(x) = \mathbf{1}_{x>0}$ (stable, but "dead ReLUs" exist)

---

5. Optimization: Backpropagation, Cross-Entropy, and Calculus

5.1 Backpropagation as chain rule

Backprop is the chain rule applied systematically through the computation graph. For $a = Wx + b$, $h = \phi(a)$, loss $\mathcal{L}(h)$:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial a} x^T, \quad \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a}, \quad \frac{\partial \mathcal{L}}{\partial x} = W^T \frac{\partial \mathcal{L}}{\partial a}$$

and

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial h} \odot \phi(a)$$

The elegance is that gradients localize: each module only needs local derivatives and upstream gradients.

5.2 Softmax

Given logits $z \in \mathbb{R}^K$ (for MNIST, $K = 10$):

$$\mathrm{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Softmax is invariant to adding a constant to all logits:

$$\mathrm{softmax}(z) = \mathrm{softmax}(z + c\mathbf{1})$$

Numerically, we stabilize with $z - \max(z)$ to prevent overflow.

5.3 Cross-entropy / negative log likelihood

For one-hot $\mathbf{y}$ and predicted probabilities $\hat{\mathbf{p}}$:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{p}}) = -\sum_{k=1}^{K} y_k \log \hat{p}_k$$

If y is the correct class, this simplifies to:

$$\mathcal{L} = -\log \hat{p}_y$$

A very useful identity for softmax + cross-entropy:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{\mathbf{p}} - \mathbf{y}$$

5.4 Stochastic gradient descent and minibatches

In practice we compute gradients on minibatches B:

$$\nabla \mathcal{L}(\theta) \approx \frac{1}{|B|} \sum_{i \in B} \nabla \ell_i(\theta)$$

This reduces compute cost and injects noise that can act as implicit regularization (especially in overparameterized regimes).

5.5 Geometry and curvature intuition

Curvature matters. A local second-order approximation is:

$$\mathcal{L}(\theta + \Delta) \approx \mathcal{L}(\theta) + \nabla \mathcal{L}(\theta)^T \Delta + \frac{1}{2} \Delta^T H(\theta) \Delta$$

where $H(\theta)$ is the Hessian. The learning rate η interacts with $\lambda_{\max}$, the largest eigenvalue of H. If η is too large relative to curvature, you diverge or bounce.

---

6. Regularization: Why Generalization Happens

Regularization is not optional in modern deep learning: it's how we turn high-capacity models into generalizing systems.

6.1 L2 regularization (weight decay)

Add a penalty to discourage large weights:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \lambda \|\theta\|_2^2$$

Gradient update becomes:

$$\theta \leftarrow \theta - \eta \left(\nabla \mathcal{L}(\theta) + 2\lambda \theta \right)$$

Interpretation: shrinkage favors smoother functions and reduces sensitivity to pixel-level noise.

6.2 Dropout

Dropout samples a Bernoulli mask m and applies it to activations. With inverted dropout:

$$\tilde{h} = \frac{m \odot h}{1 - p}, \qquad m_j \sim \text{Bernoulli}(1 - p)$$

Dropout can be seen as approximate model averaging over exponentially many subnetworks; it often improves robustness for dense nets.

6.3 Early stopping

Early stopping halts training when validation performance stops improving. In a sense, it limits the effective capacity used during optimization.

6.4 Implicit regularization of SGD/Adam

Optimization dynamics themselves can bias solutions. SGD often prefers “flatter minima,” which correlate empirically with better generalization in many regimes. It’s not a single theorem that explains deep learning, but it is a real phenomenon that influences practice.

---

7. CNNs: Why Convolution Wins for Images

Dense networks throw away geometry by flattening pixels. CNNs preserve geometry and exploit it.

7.1 Convolution operator

For input X and kernel K, the (valid) 2D convolution is:

$$(Y = X * K)[i,j] = \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} X[i+u, j+v] K[u,v]$$

Deep learning libraries often implement cross-correlation (no kernel flip). The learned kernel removes the practical distinction.

7.2 Locality, weight sharing, and equivariance

- **Locality:** early layers learn edges/corners from small neighborhoods.
- **Weight sharing:** the same detector is applied everywhere.
- **Equivariance:** translating input translates feature maps.

These properties match images, so CNNs learn faster and generalize better under translation/handwriting variation.

7.3 Parameter efficiency

Dense 784→512: $784 \cdot 512 \approx 401k$ weights.

32 conv filters of 3×3: $32 \cdot 3 \cdot 3 = 288$ weights (per input channel, plus biases).

CNNs spend parameters on **patterns**, not on **positions**.

---

8. Keras with TensorFlow Backend (Implementation Notes)

Keras is a high-level API that compiles a computation graph and runs it efficiently on CPU/GPU via TensorFlow.

8.1 What model.fit() actually does

Even though it is one call, it orchestrates:

6.3 Early stopping

Early stopping halts training when validation performance stops improving. In a sense, it limits the effective capacity used during optimization.

6.4 Implicit regularization of SGD/Adam

Optimization dynamics themselves can bias solutions. SGD often prefers “flatter minima,” which correlate empirically with better generalization in many regimes. It’s not a single theorem that explains deep learning, but it is a real phenomenon that influences practice.

---

7. CNNs: Why Convolution Wins for Images

Dense networks throw away geometry by flattening pixels. CNNs preserve geometry and exploit it.

7.1 Convolution operator

For input X and kernel K, the (valid) 2D convolution is:

$$(Y = X * K)[i, j] = \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} X[i+u, j+v] K[u, v]$$

Deep learning libraries often implement cross-correlation (no kernel flip). The learned kernel removes the practical distinction.

7.2 Locality, weight sharing, and equivariance

- **Locality:** early layers learn edges/corners from small neighborhoods.
- **Weight sharing:** the same detector is applied everywhere.
- **Equivariance:** translating input translates feature maps.

These properties match images, so CNNs learn faster and generalize better under translation/handwriting variation.

7.3 Parameter efficiency

Dense 784→512: $784 \cdot 512 \approx 401k$ weights.

32 conv filters of 3×3: $32 \cdot 3 \cdot 3 = 288$ weights (per input channel, plus biases).

CNNs spend parameters on **patterns**, not on **positions**.

---

8. Keras with TensorFlow Backend (Implementation Notes)

Keras is a high-level API that compiles a computation graph and runs it efficiently on CPU/GPU via TensorFlow.

8.1 What model.fit() actually does

If $W_{ij} \sim \mathcal{N}(0, \sigma^2)$ and inputs are roughly i.i.d., then $\mathrm{Var}(Wx) \approx n\sigma^2 \mathrm{Var}(x)$. For ReLU networks, maintaining variance suggests **He initialization**:

$$\sigma^2 \approx \frac{2}{n}$$

---

10. Measuring Model Performance and Accuracy

10.1 Accuracy

Accuracy is the fraction correct on the test set. It is necessary but not sufficient.

10.2 Confusion matrix

The confusion matrix C_{ij} counts predictions j when the true label is i. Normalizing rows yields per-class confusion probabilities. This reveals systematic confusions like 4 vs 9 or 3 vs 5 (depending on handwriting style).

10.3 Precision / recall / F1

For a class k:

$$\mathrm{precision}_k = \frac{TP_k}{TP_k + FP_k}, \qquad \mathrm{recall}_k = \frac{TP_k}{TP_k + FN_k}$$

$$\mathrm{F1}_k = 2 \cdot \frac{\mathrm{precision}_k \cdot \mathrm{recall}_k}{\mathrm{precision}_k + \mathrm{recall}_k}$$

These metrics become essential when costs are asymmetric or classes are imbalanced.

10.4 Calibration (advanced but valuable)

Softmax probabilities are not guaranteed to be calibrated. A model can be accurate but overconfident. For production systems, calibration matters because downstream decisions use confidence thresholds. A useful extension is Expected Calibration Error (ECE) or reliability diagrams.

---

11. Interactive Inference: Draw a Digit with the Mouse

The interactive section uses an HTML canvas. When the user clicks **Predict**, JavaScript captures the drawing as a PNG data URL and calls back into Python:

```js
google.colab.kernel.invokeFunction("mnist.predict_digit", [dataUrl], {});
```

11.1 The real work: preprocessing the user's sketch

A trained model expects training-like inputs. User drawings differ from MNIST (stroke width, centering, scale, anti-aliasing). The notebook performs a small but essential pipeline:

1. Convert to grayscale.

Appendix A: Information Theory, KL Divergence, and Proper Scoring Rules

Cross-entropy is not just a convenient engineering choice; it is a principled objective for probability estimation.

Let the true conditional label distribution be $p(y \mid x)$ and the model be $q_\theta(y \mid x)$. The expected negative log likelihood under the data distribution is:

$$\mathbb{E}_{(x,y)\sim p}\left[-\log q_\theta(y \mid x)\right]$$

Condition on x and expand:

$$\mathbb{E}_{x\sim p}\left[\sum_y p(y \mid x)\left(-\log q_\theta(y \mid x)\right)\right]$$

Now relate it to conditional KL divergence:

$$\mathrm{KL}\left(p(\cdot \mid x) \,\|\, q_\theta(\cdot \mid x)\right) = \sum_y p(y \mid x) \log \frac{p(y \mid x)}{q_\theta(y \mid x)}$$

Rearrange:

$$\sum_y p(y \mid x)\left(-\log q_\theta(y \mid x)\right) = H\left(p(\cdot \mid x)\right) + \mathrm{KL}\left(p(\cdot \mid x) \,\|\, q_\theta(\cdot \mid x)\right)$$

The entropy term does not depend on θ. Therefore minimizing cross-entropy is equivalent to minimizing expected KL divergence. This is why cross-entropy is a **proper scoring rule**: it incentivizes truthful probability estimates under the assumed model class.

Appendix B: Generalization and Capacity

Classical theory relates generalization error to hypothesis class complexity (VC dimension, Rademacher complexity, margin bounds). Deep learning lives in a regime where parameter counts can exceed sample counts, yet models generalize. Two complementary explanations matter in practice:

1. **Inductive bias of the architecture.** CNNs restrict the functional form in a way that matches images (locality, weight sharing). Even with many parameters, the effective function class is structured.
2. **Implicit bias of optimization.** Gradient methods do not explore parameter space uniformly; they preferentially find certain kinds of solutions (often low-norm or flat-minima solutions).
