_pdf_build

August 24, 2026

1 MNIST from Scratch

Two-layer MLP trained on MNIST using only numpy.

Architecture: `784 → 128 (ReLU) → 10 (Softmax)`

[1]:
```
# install dependencies (run once)
!pip install numpy matplotlib keras tensorflow
```

```
zsh:1: command not found: pip
```

[2]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist

np.random.seed(28)
```

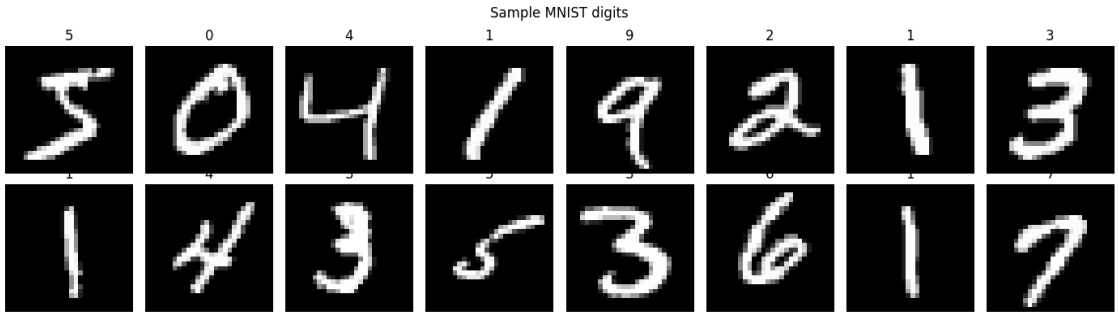
1.1 1. Data Loading & Exploration

[3]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

print(f"x_train: {x_train.shape}, dtype={x_train.dtype}")
print(f"y_train: {y_train.shape}, dtype={y_train.dtype}")
print(f"x_test:  {x_test.shape}")
print(f"y_test:  {y_test.shape}")
print(f"pixel range: [{x_train.min()}, {x_train.max()}]")
print(f"labels: {np.unique(y_train)}")
```

```
x_train: (60000, 28, 28), dtype=uint8
y_train: (60000,), dtype=uint8
x_test:  (10000, 28, 28)
y_test:  (10000,)
pixel range: [0, 255]
labels: [0 1 2 3 4 5 6 7 8 9]
```

[4]:
```
# visualize a sample of digits
fig, axes = plt.subplots(2, 8, figsize=(14, 4))
for i, ax in enumerate(axes.flat):
```

```
    ax.imshow(x_train[i], cmap='gray')
    ax.set_title(str(y_train[i]))
    ax.axis('off')
plt.suptitle('Sample MNIST digits')
plt.tight_layout()
plt.show()
```



1.2 2. Preprocessing

First we flatten it: - So that we can matrix multiply w our weights

Then we normalize it: - The weight update is W-= lr * gradient and the gradient is porportional to the input values. if the input are ~255 as opposed to ~1, the learning rate is extremely sensitve and its too easy to overshoot. - Our He initalization (discussed next) produces weights scaled to keep activations around 1 assuming that our (now normalized) inputs are around 1

[5]:
```
# flatten 28x28 → 784, normalize to [0, 1]
X_train = x_train.reshape(-1, 784).astype(np.float64) / 255.0
X_test  = x_test.reshape(-1, 784).astype(np.float64) / 255.0
```

[6]:
```
# one-hot encode labels
def one_hot(y, num_classes=10):
    out = np.zeros((len(y), num_classes))
    out[np.arange(len(y)), y] = 1.0
    return out

Y_train = one_hot(y_train)
Y_test  = one_hot(y_test)

print(f"X_train: {X_train.shape}, range [{X_train.min():.1f}, {X_train.max():.
↪1f}]")
print(f"Y_train: {Y_train.shape}, example label {y_train[0]} → {Y_train[0]}")
```

```
X_train: (60000, 784), range [0.0, 1.0]
Y_train: (60000, 10), example label 5 → [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

2 3. Network Architecture

Ai Generated - Prompt: explain why we need it w the relu activation, how we draw from normal distribution is mean 0 and var 2/fan_in , why we do so, how the standard normal would fail here, how variance changes/collapses through layers and other information that ties this explanation of He initalization together

2.1 Weight Initialization — Why It Matters

2.1.1 The core problem: variance through layers

A single layer computes `Z = X @ W`. Each output neuron is a sum of `n` terms:

```
z_i = w_1*x_1 + w_2*x_2 + ... + w_n*x_n
```

If weights and inputs are independent with zero mean, the variance of `z_i` is:

```
Var(z_i) = n * Var(w) * Var(x)
```

This means the variance **scales with fan-in (n)** (number of inputs). In a deep network, this compounds at every layer — activations either explode toward infinity or collapse toward zero before training even starts.

2.1.2 Why standard normal (std=1) fails

If `Var(w) = 1`, then after one layer `Var(z) = n * Var(x)`. With n=784, variance multiplies by 784 at the first layer alone. Activations explode — gradients become numerically unstable, and the network is untrainable from the start.

2.1.3 Why too-small init fails

If std=0.01, `Var(w) = 0.0001`. Variance collapses to near zero after the first layer. Every neuron outputs roughly the same near-zero value regardless of input — the network loses all signal and gradients vanish.

2.1.4 Xavier initialization (Glorot, 2010)

To keep variance stable: set `n * Var(w) = 1`, giving:

```
Var(w) = 1/n     →     std = √(1/n)
```

This works well for **symmetric activations** like tanh or linear, where the full output range is used. But it assumes the activation doesn't change the variance. This doesn't apply here since we're using ReLU.

2.1.5 The ReLU problem — and He's fix

ReLU zeros out all negative values. If `z ~ N(0, ²)`, roughly half the values get zeroed. This halves the variance at every layer:

```
Var(relu(z))   (1/2) * Var(z)
```

So with Xavier init under ReLU, variance halves at each layer — you get vanishing activations even though the init looked fine on paper.

He initialization (He et al., 2015) compensates by doubling the target variance:

```
(1/2) * n * Var(w) = 1
Var(w) = 2/n      →      std = √(2/n)
```

The 2 exactly cancels the halving ReLU introduces. Variance stays near 1 through the full forward pass.

In practice:

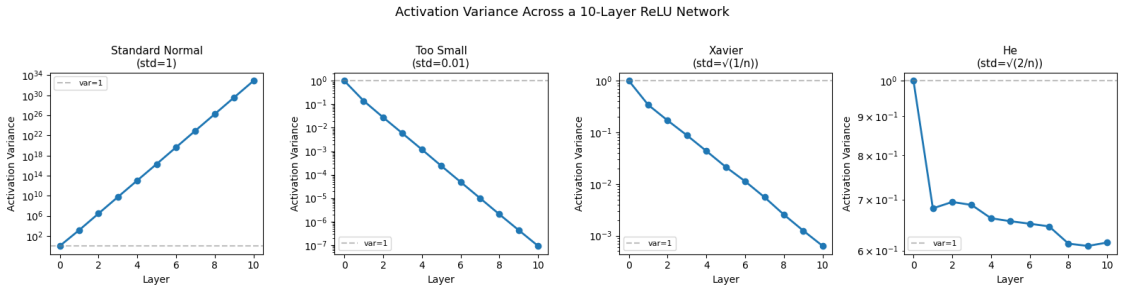
```
W = np.random.randn(fan_in, fan_out) * np.sqrt(2 / fan_in)
```

Mean 0, std = $\sqrt{(2/\text{fan\_in})}$. Biases are initialized to zero — they don't affect variance propagation.

[7]:
```
# simulate a 10-layer ReLU network under different initializations
# and track how activation variance changes at each layer
# to prove that He initialization is what we need with our ReLU activation

np.random.seed(42)

N_LAYERS  = 10
N_NEURONS = 4096
N_SAMPLES = 1000

x = np.random.randn(N_SAMPLES, N_NEURONS)# n_samples by n_neurons matrix of std␣
 ↪normal vals, var  1
print(f"variances: {x.var()}")

schemes = {
    'Standard Normal\n(std=1)':       lambda n: 1.0,
    'Too Small\n(std=0.01)':          lambda n: 0.01,
    'Xavier\n(std=√(1/n))':           lambda n: np.sqrt(1.0 / n),
    'He\n(std=√(2/n))':               lambda n: np.sqrt(2.0 / n),
}

fig, axes = plt.subplots(1, 4, figsize=(16, 4), sharey=False) # 4 subplots that␣
 ↪dont share y axis

for ax, (name, std_fn) in zip(axes, schemes.items()):
    variances = [float(x.var())]
    h = x.copy()
    for _ in range(N_LAYERS):
        W = np.random.randn(N_NEURONS, N_NEURONS) * std_fn(N_NEURONS) #␣
 ↪n_neuron by n_neuron rand std normal values and scales it to chosen schema␣
 ↪std
        h = np.maximum(0, h @ W)    # do forward pass and apply ReLU elementwise
```

```
        variances.append(float(h.var())) # variance for this layer

    ax.plot(range(N_LAYERS + 1), variances, marker='o', linewidth=2)
    ax.axhline(y=1.0, color='gray', linestyle='--', alpha=0.5, label='var=1')
    ax.set_title(name, fontsize=11)
    ax.set_xlabel('Layer')
    ax.set_ylabel('Activation Variance')
    ax.set_yscale('log')
    ax.legend(fontsize=8)

plt.suptitle('Activation Variance Across a 10-Layer ReLU Network', fontsize=13,␣
 ↪y=1.02)
plt.tight_layout()
plt.show()
```

```
variances: 1.0002173502609304
```



As discussed above, for one ReLU layer w n inputs:

$Var_{output} = Var_{Input} * n * std^2 * 0.5$ so we need our $k = n * std^2 * 0.5$ to be 1 since > 1 means explosion and < 1 means vanishing

for standard, the k factor is 128 $(256 * 1 * 0.5)$ and so explodes by x128 per layer for 0.01, the k factor is 0.0128 and so vanishes x0.0128 per layer for xavier, the k factor is 0.5 $(var = 1/256)$ and so x0.5 per layer finally for He, the k factor is 1 so its stable

From graph we see that Xavier panel shows variance dropping from 1 to $\approx 10^{-4}$. Other ones are explode/vanish. Interestingly, while He does stabilize around 0.6 (not 1) it still drops alot from 1. this is because the theoretical derivation assumes 50% actiavtion but in practice, this isn't exact. We just need to make sure that the first forward pass doesnt fail and then backprop can work even if var is not exactly 1 and we can achieve that using He. Extra: we could also use batch normalization to remedy this.

[8]:
```
### Weight initialization
W1 = np.random.randn(784, 128) * np.sqrt(2 / 784)    # He init
b1 = np.zeros((1, 128))

W2 = np.random.randn(128, 10) * np.sqrt(2 / 128)     # He init
```

```
b2 = np.zeros((1, 10))

print(f"W1: {W1.shape},  std={W1.std():.4f}  (expected  {np.sqrt(2/784):.4f})")
print(f"W2: {W2.shape},  std={W2.std():.4f}  (expected  {np.sqrt(2/128):.4f})")
```

```
W1: (784, 128),  std=0.0504  (expected  0.0505)
W2: (128, 10),  std=0.1267  (expected  0.1250)
```

2.2 4. Forward Pass

For input X:

```
Z1 = X @ W1 + b1     ← linear transform (input → hidden)
A1 = relu(Z1)        ← activation (hidden layer)
Z2 = A1 @ W2 + b2    ← linear transform (hidden → output)
A2 = softmax(Z2)     ← activation (output layer) → probabilities
```

2.2.1 ReLU

Applied element-wise to the hidden layer's pre-activations (h @ W).

```
relu(x) = max(0, x)
```

This is the non-linearity that lets the network learn non-trivial patterns (link)

Derivative for backprop:

```
relu'(x) = 1  if x > 0
           0  otherwise
```

Binary mask — 1 where the neuron was "on", 0 where it was "off".

[9]:
```
# will implement in numpy ourselves
def relu(Z):
    return np.maximum(0, Z)

def relu_deriv(Z):
    return (Z > 0).astype(np.float64)

# test
z = np.array([-3.0, -0.5, 0.0, 1.2, 4.0])
print("input:      ", z)
print("relu:       ", relu(z))
print("relu_deriv: ", relu_deriv(z))
```

```
input:       [-3.  -0.5  0.   1.2  4. ]
relu:        [0.  0.  0.  1.2 4. ]
relu_deriv:  [0. 0. 0. 1. 1.]
```

relu_deriv shows us that $z_{1..3}$ didnt affect output, as such relu_deriv acts as a mask: neurons that dired during forwrd pass are the only ones that will be affected in backward pass. during backward

pass/backprop the error sugnal (gradient) from the layer ahead is multiplied by the relu_deriv so that the gradient update only applies to "active" neurons.

2.2.2 Softmax

Applied row-wise to the output layer's 10 raw logits. Converts them to a probability distribution — all values positive, summing to 1:

$softmax(z)_i = exp(z_i)/\Sigma exp(z_j)$

Numerical stability: `exp(z)` overflows for large z (e.g. `exp(1000) = inf`). Fix: subtract the row maximum before exponentiating. This doesn't change the output mathematically — the max cancels in numerator and denominator — but it clamps the largest exponent to `exp(0) = 1`:

$softmax(z)_i = exp(z_i - max(z))/\Sigma exp(z_j - max(z))$

[10]:
```
def softmax(Z):
    Z_shifted = Z - Z.max(axis=1, keepdims=True)
    # `keepdims=True` is needed so the subtraction and division broadcast␣
 ↪correctly across
    # all 10 columns. for our example below it gives [[2.0], [2.5]] instead of␣
 ↪[2.0,2.5]

    exp_Z = np.exp(Z_shifted)
    return exp_Z / exp_Z.sum(axis=1, keepdims=True)

# quick test
z = np.array([[2.0, 1.0, 0.1],
              [0.5, 2.5, 1.0]])
out = softmax(z)
print("softmax output:\n", out.round(4))
print("row sums:       ", out.sum(axis=1))   # must be [1. 1.]
```

```
softmax output:
 [[0.659  0.2424 0.0986]
 [0.0996 0.7361 0.1643]]
row sums:        [1. 1.]
```

2.2.3 Z1 = X @ W1 + b1 — first linear transform

Each of the 128 hidden neurons computes a weighted sum of all 784 inputs, plus its bias. For a batch of m examples this is one matrix multiply:

```
X:  (m, 784)
W1: (784, 128)
b1: (1, 128)
Z1: (m, 128)
```

Each hidden neuron has a direct weighted connection ($W1[:, 0]$) to every one of the 784 input pixels. After training, a neuron might have learnt to fire when certain pixels look a certain way but recieves all as input.

[11]:
```
X_batch = X_train[:5]   # 5 images for inspection
print(f"X_batch shape: {X_batch.shape}")

print(f"W1 shape: {W1.shape}")
Z1 = X_batch @ W1 + b1 # b1 is all zeros

print(f"Z1 shape: {Z1.shape}")
print(f"Z1[:, :6]:\n {Z1[:, :6].round(4)}")  # sample - mix of positive and␣
 ↪negative
```

```
X_batch shape: (5, 784)
W1 shape: (784, 128)
Z1 shape: (5, 128)
Z1[:, :6]:
 [[ 0.2259  0.0268 -0.4166  0.0116  0.7327 -0.1666]
 [ 0.5531  0.146  -0.0483 -0.111   1.4601  0.0978]
 [ 0.4741 -0.3063 -0.4869 -0.675  -0.3209  0.3973]
 [-0.3798  0.1897 -0.2642 -0.1936  0.6418  0.0894]
 [-0.1574  0.5124 -0.5667 -0.5088  0.2762  0.0987]]
```

2.2.4 A1 = relu(Z1) — hidden layer activations

ReLU applied element-wise to Z1.

```
Z1: (m, 128)  →  A1: (m, 128)   (same shape, element-wise)
```

As explained previously, using He init and normalized inputs, roughly half the neurons will be active (>0) at initialization.

[12]:
```
A1 = relu(Z1)
print(f"A1 shape: {A1.shape}")
print(f"A1[:, :6]:\n {A1[:, :6].round(4)}")           # negatives are gone
print(f"fraction active: {(A1 > 0).mean():.2f}")    # should be ~0.5
```

```
A1 shape: (5, 128)
A1[:, :6]:
 [[0.2259 0.0268 0.     0.0116 0.7327 0.    ]
 [0.5531 0.146  0.     0.     1.4601 0.0978]
 [0.4741 0.     0.     0.     0.     0.3973]
 [0.     0.1897 0.     0.     0.6418 0.0894]
 [0.     0.5124 0.     0.     0.2762 0.0987]]
fraction active: 0.49
```

2.2.5 Z2 = A1 @ W2 + b2 — second linear transform

The 128 hidden activations are now combined into 10 output values — one per digit class.

```
A1: (m, 128)
W2: (128, 10)
b2: (1, 10)
Z2: (m, 10)   ← 10 raw logits per example
```

These logits have no probabilistic meaning yet — they're unbounded real numbers. The relative magnitudes matter (larger = more confident), but they don't sum to 1 until softmax.

[13]:
```
Z2 = A1 @ W2 + b2
print(f"A1 Shape: {A1.shape}")
print(f"W2 Shape: {W2.shape}")
print(f"b2 Shape: {b2.shape}")
print(f"Z2 shape: {Z2.shape}")           # (5, 10)
print(f"Z2:\n {Z2.round(4)}")        # raw logits - not probabilities
```

```
A1 Shape: (5, 128)
W2 Shape: (128, 10)
b2 Shape: (1, 10)
Z2 shape: (5, 10)
Z2:
 [[-0.2057  0.417   0.0436 -0.0819  0.2787 -0.4466 -0.0917 -0.1095  0.5506
  -0.4546]
 [-0.3342  0.5532  0.1802  0.1357  0.2675 -0.3328 -0.3438 -0.0171  0.5439
  -1.0738]
 [ 0.0372  0.0202  0.1701  0.4295  0.3833 -0.0484  0.0224  0.1858 -0.2235
  -0.2611]
 [-0.4904  0.1216  0.0862 -0.2179  0.1832  0.2243 -0.3095 -0.1947  0.3629
  -0.8076]
 [ 0.0508 -0.1908  0.1958  0.5223  0.1538  0.2427  0.1996 -0.0067 -0.0168
  -0.8186]]
```

2.2.6 A2 = softmax(Z2) — output probabilities

Softmax converts the 10 logits into a probability distribution. The predicted class is the highest probability one.

`Z2: (m, 10)  →  A2: (m, 10)`

At initialization with random weights, all 10 classes get roughly equal probability (~0.1 each). The network has no knowledge yet — it's essentially guessing uniformly. As such accuracy is 10% per image (1 in 10 change of getting it right).

[14]:
```
A2 = softmax(Z2)
print(f"A2 shape: {A2.shape}")
print(f"A2[0]:    {A2[0].round(4)}")              # probabilities - should sum␣
 ↪to 1
print(f"row sums: {A2.sum(axis=1).round(6)}")     # all 1s
print(f"predicted: {np.argmax(A2, axis=1)}")      # random guesses at init
print(f"true:      {y_train[:5]}")
```

```
A2 shape: (5, 10)
A2[0]:    [0.078  0.1454 0.1001 0.0883 0.1266 0.0613 0.0874 0.0859 0.1662
0.0608]
row sums: [1. 1. 1. 1. 1.]
```

```
predicted: [8 1 3 8 3]
true:      [5 0 4 1 9]
```

2.3 5. Loss — Cross-Entropy

Cross-entropy for one example:

`L = -log(A2[k])`

Just the negative log of the probability assigned to the correct class — all the zeros in `Y` wipe out every other term.

Why log? The penalty grows steeply as confidence in the wrong answer increases. - `A2[k] = 1.0` (perfect) → loss = 0 - `A2[k] = 0.5` (uncertain) → loss = 0.69 - `A2[k] = 0.01` (very wrong) → loss = 4.6

Over a batch of m examples, sum over all classes and average:

`L = -(1/m) * Σ_examples Σ_classes  Y * log(A2)`

Numerical stability: `log(0) = -inf`. If the network assigns zero probability to the correct class, loss blows up so we clip A2 to a small minimum before taking log:

`np.log(np.clip(A2, 1e-12, 1.0))`

[15]:
```
def cross_entropy_loss(A2, Y):
    m = Y.shape[0]
    log_probs = np.log(np.clip(A2, 1e-12, 1.0))
    return -np.sum(Y * log_probs) / m

# test in 5-example batches
loss = np.mean([cross_entropy_loss(A2, Y_train[x:x+5]) for x in range(0, 100,␣
 ↪5)])
print(f"loss: {loss:.4f}")
print(f"expected at random init: ~{-np.log(1/10):.4f}  (-log(0.1) - uniform␣
 ↪over 10 classes)")
```

```
loss: 2.3926
expected at random init: ~2.3026  (-log(0.1) - uniform over 10 classes)
```

2.4 6. Backpropagation

Now that we have computed a loss, we use backprop to see how does the loss change if we nudge each weight? (the gradient) — and we use it to update the weights in whichever direction reduces loss.

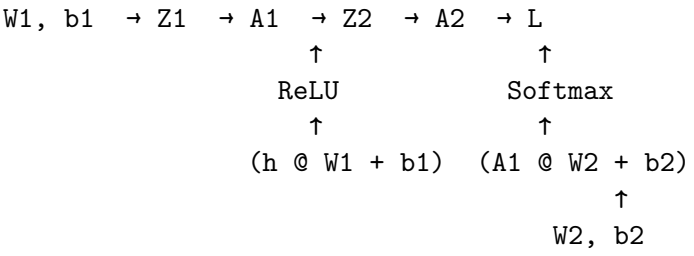
chain rule

since the loss is a composition of functions, its gradient wrt any parameter is a **product** of local gradients along the path from that parameter to the loss. For example, the derivative of the final output wrt to any early input is:

$dL/dW1 = dL/dA2 \times dA2/dZ2 \times dZ2/dA1 \times dA1/dZ1 \times dZ1/dW1$

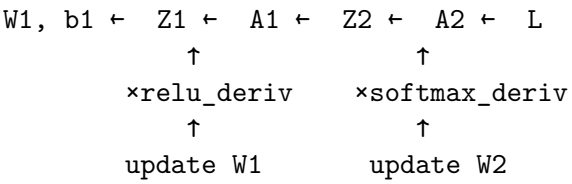
where each term is a local gradient, how does this step chaneg when you nudge its input and we use backprop to compute those gradients **right to left**.

Foward pass



$$
\begin{aligned}
Z_1 &= HW_1 + b_1 \\
A_1 &= \mathrm{ReLU}(Z_1) \\
Z_2 &= A_1 W_2 + b_2 \\
A_2 &= \mathrm{softmax}(Z_2) \\
L &= -\sum_i Y_i \log(A_{2,i}) \quad \text{(cross-entropy)}
\end{aligned}
$$

Backward pass (right → left):



We want to compute gradient of the loss L wrt to each param, dL/dW1, dL/db1, dL/dW2, dL/db2.

1. Gradient at Z2 (dL/dZ2) is just A2-Y (proof below) with shape (n, 10). Simply, prediction minus truth.
2. Gradient of W2 and b2.

a. W2

dL/dW2 = dL/dZ2 * dZ2/dW2

$Z2_{kj} = \sum_p A1_{kp} \cdot W2_{pj} + b2_j$ W2_{ij} only affects j-th column of Z2 (when p=i): this is important for next derivation.

Chain ruke: W2_{ij} affects m different images so according to chain rule we sum over all of them:

$$\frac{\partial L}{\partial W2_{ij}} = \sum_{k=1}^{m} \frac{\partial L}{\partial Z2_{kj}} \cdot \frac{\partial Z2_{kj}}{\partial W2_{ij}}$$

dL/dZ2_{kj} is the (k,j) entry of dZ2, dZ2_{kj}/dW2_{ij} is: the formula above differentiated by W2. Only the term where p=i survives, $A1_{ki} \cdot W2_{ij}$, so $= A1_{ki}$. so:

$$\frac{\partial L}{\partial W2_{ij}} = 1/m * \sum_{k=1}^{m} A1_{ki} \cdot dZ2_{kj}$$

to turn it into a matrix multiplication we note that if we transpose A1_{ki} it, it becomes this form which can be turned into dot prod notation of form $(X \cdot Y)_{ij} = \sum_k X_{ik} \cdot Y_{kj}$. so finally we get:

$$dL/dW2 = dW2 = 1/m * A1^T \cdot dZ2$$

$$dL/dW2 = 1/m * A1.T@dZ2$$

of course we also need to remember that the average cross entropy is $1/m \sum_{k=1}^{m} L_k$ so as we differentiate we need to bring this constant (which is why there is 1/m above).

b. b2

$Z2_{kj} = \sum_p A1_{kp} \cdot W2_{pj} + b2_j$

b2_j only affects the j-th column of Z2 and affects each row Z2_{1j}…Z2_{mj}. i.e it affects, for each image, the j-th output neuron.

Chain rule:

$$\frac{\partial L}{\partial b2_j} = \sum_{k=1}^{m} \frac{\partial L}{\partial Z2_{kj}} \cdot \frac{\partial Z2_{kj}}{\partial b2_j}$$

dL/dZ2_{kj} is just the (k,j) entry of dZ2. the dZ2_{kj}/db2_{j} is:

$$Z2_{kj} = \sum_p A1_{kp} \cdot W2_{pj} + b2_j$$

diferentiate wrt to b2_{j}, the sum is 0 only b2_{j} survives and its derivative wrt to itself is 1. which gives us:

$$\frac{\partial L}{\partial b2_j} = \sum_{k=1}^{m} dZ2_{kj} \cdot 1 = \sum_{k=1}^{m} dZ2_{kj}$$

as we can see that the summation is just the sum of column j of dZ2. bringing back the 1/m coefficient gives: $1/m * \sum_k^m = 1 dL/dZ2_{k,:}$ which is mean(dZ2, axis=0)

AI Generated

2.4.1 dW2, db2 — output layer weight gradients

`Z2 = A1 @ W2 + b2`, so by chain rule:

`dL/dW2 = A1.T @ dZ2 / m`

Shapes: `A1.T (128, m) @ dZ2 (m, 10)` → `dW2 (128, 10)` — same shape as W2.

The `/m` comes from the fact that our loss is averaged over the batch. Each example contributes one outer product `a1  dz2` to the weight gradient — we average them.

db2: b2 is broadcast across all m examples, so each example contributes equally to its gradient. We average dZ2 across the batch dimension:

```
db2 = mean(dZ2, axis=0)    →  shape (1, 10), same as b2
```

[ ]:

AI generated

2.4.2 dZ2 = A2 - Y — combined softmax + cross-entropy gradient

This is the only non-obvious result in the whole derivation. The chain rule through softmax produces a Jacobian (matrix of partial derivatives), but when composed with cross-entropy it collapses to something clean.

Derivation (single example):

Cross-entropy loss: `L = -Σ_k Y_k * log(A2_k)`

Chain rule: `dL/dZ2_j = Σ_k (dL/dA2_k) * (dA2_k/dZ2_j)`

The two pieces:

```
dL/dA2_k = -Y_k / A2_k                          ← derivative of log

dA2_k/dZ2_j = A2_k(1 - A2_k)   if k == j       ← softmax Jacobian
              -A2_k * A2_j      if k != j
```

Substituting in:

```
dL/dZ2_j = (-Y_j/A2_j)(A2_j)(1 - A2_j)  +  Σ_{k j} (-Y_k/A2_k)(-A2_k * A2_j)
         = -Y_j(1 - A2_j)  +  A2_j * Σ_{k j} Y_k
         = -Y_j + Y_j*A2_j + A2_j * Σ_{k j} Y_k
         = -Y_j + A2_j * (Y_j + Σ_{k j} Y_k)
         = -Y_j + A2_j * Σ_k Y_k
```

Since Y is one-hot, `Σ_k Y_k = 1`:

```
dL/dZ2_j = A2_j - Y_j
```

So for the full batch: **`dZ2 = A2 - Y`**

The interpretation is direct: for each example, dZ2 is the difference between what the network predicted and what the correct answer was. If the network is confident and correct, dZ2 is near zero — almost no gradient, almost no update needed.

[16]:
```
m = X_batch.shape[0]

dZ2 = A2 - Y_train[:5]
print(f"dZ2 shape: {dZ2.shape}")
```

```
print(f"dZ2[0]: {dZ2[0].round(4)}")        # predicted minus true: one entry␣
 ↪near -0.9 since 0.1 (random guesses in A2[0]) - 1 (the actual from one hot),␣
 ↪rest small positive
print(f"dZ2 row sums: {dZ2.sum(axis=1).round(6)}")  # should be ~0 (A2 sums to␣
 ↪1, Y sums to 1)
```

```
dZ2 shape: (5, 10)
dZ2[0]: [ 0.078   0.1454  0.1001  0.0883  0.1266 -0.9387  0.0874  0.0859  0.1662
  0.0608]
dZ2 row sums: [ 0.  0.  0.  0. -0.]
```

[17]:
```
dW2 = A1.T @ dZ2 / m
db2 = np.mean(dZ2, axis=0, keepdims=True)

print(f"dW2 shape: {dW2.shape}")   # (128, 10) - matches W2
print(f"db2 shape: {db2.shape}")   # (1, 10)   - matches b2
```

```
dW2 shape: (128, 10)
db2 shape: (1, 10)
```

dL/dA1

2.4.3 dA1 — gradient flowing back through W2

We have the gradient at Z2 (`dZ2`). To continue backprop into the hidden layer, we need the gradient at A1 — how does the loss change w.r.t. each hidden activation?

`Z2 = A1 @ W2 + b2`, so:

`dL/dA1 = dZ2 @ W2.T`

Shapes: `dZ2 (m, 10) @ W2.T (10, 128)` → `dA1 (m, 128)` — same shape as A1.

Intuitively: each hidden neuron's activation influenced the loss through all 10 output neurons. `W2.T` routes the output-layer error signals back, weighted by how strongly each hidden neuron connected to each output.

A1_{ki} is the activation of hidden neuron i for image k.

$$Z2_{kj} = \sum_{p} A1_{kp} \dot{W}2_{pj} + b2_{j}$$

A1_{ki} only appears when p=i, it shows up for every output neuron j (different j use different columns of W2 but the same row k of A1). And it only shows up in row k of Z2 so images k's activations dont affect any of the other image's pre activations. So A1_{ki} affects n2 different entries of Z2 (Z2_{k,1}…Z2_{k,n2}) i.e all output neurons but only for image k.

chain rule:

$$\frac{\partial L}{\partial A1_{ki}} = \sum_{j=1}^{n_2} \frac{\partial L}{\partial Z2_{kj}} \cdot \frac{\partial Z2_{kj}}{\partial A1_{ki}}$$

for $\frac{\partial Z2_{kj}}{\partial A1_{ki}}$, differentiate the sum only the p=i survives and d wrt A1_{ki} is $W2_{ij}$:

$$\frac{\partial L}{\partial A1_{ki}} = \sum_{j=1}^{n_2} dZ2_{kj} \cdot W2_{ij}$$

as a matrix multiplication (similar to before):

$dA1 = dZ2 \cdot W2^T$

note, there is no 1/m since its already baked into dZ2.

[18]:
```
dA1 = dZ2 @ W2.T
print(f"dA1 shape: {dA1.shape}")   # (5, 128) - matches A1
```

```
dA1 shape: (5, 128)
```

dL/dZ1

2.4.4 dZ1 — gradient through ReLU

We have `dA1` (gradient at the hidden activations). We need to push it back through ReLU to get the gradient at the pre-activations `Z1`.

`A1 = relu(Z1)`, so by chain rule:

`dL/dZ1 = dL/dA1 * relu'(Z1)`

This is element-wise multiplication. `relu'(Z1)` is the binary mask we defined earlier — 1 where the neuron was active, 0 where it was zeroed out by ReLU.

The key consequence: **neurons that were off (Z1 0) receive zero gradient**. The error signal is completely blocked for those neurons on this example. Their weights get no update from it. This is the "dying ReLU" tradeoff — ReLU is fast and works well, but inactive neurons contribute nothing to learning.

A1 depends only on Z1 so chain rule:

$$\frac{\partial L}{\partial Z1_{ki}} = \frac{\partial L}{\partial A1_{ki}} \cdot \frac{\partial A1_{ki}}{\partial Z1_{ki}}$$

dA1_{ki} is computed already and the second piece is the derivative of ReLU (as done previously):

$$dZ1 = dA1 \odot \text{ReLU}'(Z1)$$

i.e: dZ1 = dA1 * relu_deriv(Z1)

where is the hadamard product, elementwise multiplication. the intuition is simply that the "on" neurons are changed and the "off" ones are bitmasked out.

[19]:
```
dZ1 = dA1 * relu_deriv(Z1)
print(f"dZ1 shape: {dZ1.shape}")                          # (5, 128) - matches␣
 ↪Z1
print(f"fraction with zero gradient: {(dZ1 == 0).mean():.2f}")  # ~0.5 -␣
 ↪neurons that were off
```

```
dZ1 shape: (5, 128)
fraction with zero gradient: 0.51
```

2.4.5 dW1, db1 — hidden layer weight gradients

Same pattern as dW2/db2, but one layer earlier. `Z1 = X @ W1 + b1`, so:

```
dL/dW1 = X.T @ dZ1 / m     →  shape (784, 128) - matches W1
db1    = mean(dZ1, axis=0)  →  shape (1, 128)   - matches b1
```

This completes the backward pass. We now have gradients for all four parameters: `dW1`, `db1`, `dW2`, `db2`.

[20]:
```
dW1 = X_batch.T @ dZ1 / m
db1 = np.mean(dZ1, axis=0, keepdims=True)

print(f"dW1 shape: {dW1.shape}")   # (784, 128) - matches W1
print(f"db1 shape: {db1.shape}")   # (1, 128)   - matches b1

# sanity check: all gradient shapes match their parameter shapes
assert dW1.shape == W1.shape
assert db1.shape == b1.shape
assert dW2.shape == W2.shape
assert db2.shape == b2.shape
print("all gradient shapes match - backprop is consistent")
```

```
dW1 shape: (784, 128)
db1 shape: (1, 128)
all gradient shapes match - backprop is consistent
```

2.5 7. Training Loop

Mini-batch SGD: random batches of 64. Each batch gives a noisy but cheap estimate of the true gradient.

One **epoch** = one full pass through the training set. With 60,000 examples and batch size 64, that's ~937 gradient updates per epoch.

Weight update rule (SGD):

```
W -= lr * dW
b -= lr * db
```

`lr` (learning rate) controls step size. Too large and updates overshoot else if too small then training is slow. we are using 0.1.

[21]:
```
### Weight initialization
W1 = np.random.randn(784, 128) * np.sqrt(2 / 784)    # He init
b1 = np.zeros((1, 128))

W2 = np.random.randn(128, 10) * np.sqrt(2 / 128)     # He init
b2 = np.zeros((1, 10))

print(f"W1: {W1.shape},  std={W1.std():.4f}  (expected  {np.sqrt(2/784):.4f})")
print(f"W2: {W2.shape},  std={W2.std():.4f}  (expected  {np.sqrt(2/128):.4f})")

def forward(X):
    Z1 = X @ W1 + b1
    A1 = relu(Z1)
    Z2 = A1 @ W2 + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

def backward(X, Y, Z1, A1, Z2, A2):
    m = X.shape[0]
    dZ2 = A2 - Y
    dW2 = A1.T @ dZ2 / m
    db2 = np.mean(dZ2, axis=0, keepdims=True)
    dA1 = dZ2 @ W2.T
    dZ1 = dA1 * relu_deriv(Z1)
    dW1 = X.T @ dZ1 / m
    db1 = np.mean(dZ1, axis=0, keepdims=True)
    return dW1, db1, dW2, db2

# hyperparams
EPOCHS     = 20
BATCH_SIZE = 64
LR         = 0.1

n = X_train.shape[0]
history = {'loss': [], 'acc': []}
test_history= {'loss': [], 'acc': []}

for epoch in range(EPOCHS):
    # shuffle training data each epoch
    idx = np.random.permutation(n)
    X_shuf, Y_shuf = X_train[idx], Y_train[idx]

    for i in range(0, n, BATCH_SIZE):
        X_batch = X_shuf[i:i+BATCH_SIZE]
```

```
        Y_batch = Y_shuf[i:i+BATCH_SIZE]

        Z1, A1, Z2, A2 = forward(X_batch)
        dW1, db1, dW2, db2 = backward(X_batch, Y_batch, Z1, A1, Z2, A2)

        W1 -= LR * dW1
        b1 -= LR * db1
        W2 -= LR * dW2
        b2 -= LR * db2

    # eval on full training set at end of epoch
    _, _, _, A2_train = forward(X_train)
    loss = cross_entropy_loss(A2_train, Y_train)
    acc  = (np.argmax(A2_train, axis=1) == y_train).mean()
    history['loss'].append(loss)
    history['acc'].append(acc)
    print(f"epoch {epoch+1:2d}/{EPOCHS} Training: loss={loss:.4f}  acc={acc:.
↪4f} | ", end="")


    # eval on full test set at end of epoch
    _, _, _, A2_test = forward(X_test)
    test_loss=cross_entropy_loss(A2_test, Y_test)
    test_acc=(np.argmax(A2_test, axis=1) == y_test).mean()
    test_history['loss'].append(test_loss)
    test_history['acc'].append(test_acc)
    print(f"Testing: loss={test_loss:.4f}  acc={test_acc:.4f}")
```

```
W1: (784, 128),  std=0.0504  (expected  0.0505)
W2: (128, 10),  std=0.1266  (expected  0.1250)
epoch  1/20 Training: loss=0.2276  acc=0.9343 | Testing: loss=0.2196  acc=0.9360
epoch  2/20 Training: loss=0.1629  acc=0.9538 | Testing: loss=0.1657  acc=0.9511
epoch  3/20 Training: loss=0.1359  acc=0.9600 | Testing: loss=0.1430  acc=0.9581
epoch  4/20 Training: loss=0.1036  acc=0.9709 | Testing: loss=0.1158  acc=0.9648
epoch  5/20 Training: loss=0.0911  acc=0.9741 | Testing: loss=0.1058  acc=0.9671
epoch  6/20 Training: loss=0.0781  acc=0.9782 | Testing: loss=0.0963  acc=0.9702
epoch  7/20 Training: loss=0.0679  acc=0.9808 | Testing: loss=0.0894  acc=0.9727
epoch  8/20 Training: loss=0.0626  acc=0.9826 | Testing: loss=0.0856  acc=0.9750
epoch  9/20 Training: loss=0.0541  acc=0.9856 | Testing: loss=0.0793  acc=0.9760
epoch 10/20 Training: loss=0.0478  acc=0.9874 | Testing: loss=0.0752  acc=0.9771
epoch 11/20 Training: loss=0.0453  acc=0.9881 | Testing: loss=0.0744  acc=0.9761
```

```
epoch 12/20 Training: loss=0.0395  acc=0.9907 | Testing: loss=0.0721  acc=0.9777
epoch 13/20 Training: loss=0.0380  acc=0.9909 | Testing: loss=0.0735  acc=0.9776
epoch 14/20 Training: loss=0.0343  acc=0.9916 | Testing: loss=0.0698  acc=0.9784
epoch 15/20 Training: loss=0.0308  acc=0.9924 | Testing: loss=0.0679  acc=0.9796
epoch 16/20 Training: loss=0.0301  acc=0.9930 | Testing: loss=0.0695  acc=0.9802
epoch 17/20 Training: loss=0.0284  acc=0.9933 | Testing: loss=0.0691  acc=0.9790
epoch 18/20 Training: loss=0.0252  acc=0.9945 | Testing: loss=0.0671  acc=0.9805
epoch 19/20 Training: loss=0.0220  acc=0.9960 | Testing: loss=0.0639  acc=0.9803
epoch 20/20 Training: loss=0.0213  acc=0.9959 | Testing: loss=0.0660  acc=0.9808
```

2.6 8. Evaluation

[22]:
```
_, _, _, A2_test = forward(X_test)

test_loss = cross_entropy_loss(A2_test, Y_test)
test_acc  = (np.argmax(A2_test, axis=1) == y_test).mean()

print(f"test  loss={test_loss:.4f}  acc={test_acc:.4f}")
print(f"train loss={history['loss'][-1]:.4f}  acc={history['acc'][-1]:.4f}")
print(f"overfit gap: {history['acc'][-1] - test_acc:.4f}")
```

```
test  loss=0.0660  acc=0.9808
train loss=0.0213  acc=0.9959
overfit gap: 0.0151
```

2.7 9. Interactive Inference

Draw a digit on the 28×28 grid and the model runs inference in real time, showing the probability distribution across all 10 classes.

Requires `ipympl` for interactive matplotlib in JupyterLab (`%matplotlib widget`).

[23]:
```
!pip install ipympl -q
```

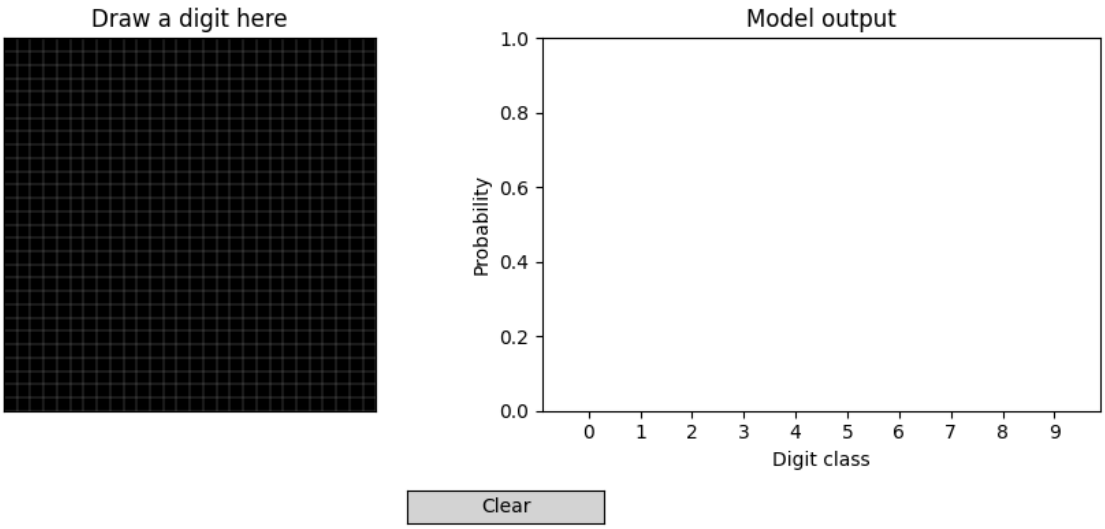
```
zsh:1: command not found: pip
```

[24]:
```
# AI Generated

from matplotlib.widgets import Button

canvas = np.zeros((28, 28), dtype=np.float64)

fig, (ax_draw, ax_prob) = plt.subplots(1, 2, figsize=(9, 4))
plt.subplots_adjust(bottom=0.15)
```

```
# --- drawing panel ---
img = ax_draw.imshow(canvas, cmap='gray', vmin=0, vmax=1,␣
 ↪interpolation='nearest')
ax_draw.set_title('Draw a digit here')
ax_draw.set_xticks(np.arange(-0.5, 28, 1), minor=True)
ax_draw.set_yticks(np.arange(-0.5, 28, 1), minor=True)
ax_draw.grid(which='minor', color='gray', linewidth=0.3)
ax_draw.tick_params(which='both', bottom=False, left=False, labelbottom=False,␣
 ↪labelleft=False)

# --- probability panel ---
bars = ax_prob.bar(range(10), np.zeros(10), color='steelblue')
ax_prob.set_xticks(range(10))
ax_prob.set_ylim(0, 1)
ax_prob.set_xlabel('Digit class')
ax_prob.set_ylabel('Probability')
ax_prob.set_title('Model output')
pred_text = ax_prob.text(0.5, 0.93, '', transform=ax_prob.transAxes,
                         ha='center', fontsize=13, fontweight='bold')

def run_inference():
    x_input = canvas.reshape(1, 784)
    _, _, _, A2 = forward(x_input)
    probs = A2[0]
    pred  = np.argmax(probs)
    for bar, p in zip(bars, probs):
        bar.set_height(p)
    for i, bar in enumerate(bars):
        bar.set_color('tomato' if i == pred else 'steelblue')
    pred_text.set_text(f'Guess: {pred}  ({probs[pred]*100:.1f}%)')
    fig.canvas.draw_idle()

def paint(event):
    if event.inaxes != ax_draw:
        return
    x, y = int(round(event.xdata)), int(round(event.ydata))
    for dy in range(-1, 2):       # 3x3 brush
        for dx in range(-1, 2):
            nx, ny = x + dx, y + dy
            if 0 <= nx < 28 and 0 <= ny < 28:
                canvas[ny, nx] = 1.0
    img.set_data(canvas)
    run_inference()

is_drawing = [False]

def on_press(event):
```

```
    is_drawing[0] = True
    paint(event)

def on_release(event):
    is_drawing[0] = False

def on_move(event):
    if is_drawing[0]:
        paint(event)

fig.canvas.mpl_connect('button_press_event', on_press)
fig.canvas.mpl_connect('button_release_event', on_release)
fig.canvas.mpl_connect('motion_notify_event', on_move)

# --- clear button ---
ax_btn = fig.add_axes([0.42, 0.02, 0.16, 0.06])
btn = Button(ax_btn, 'Clear', color='lightgray')

def clear(_):
    canvas[:] = 0
    img.set_data(canvas)
    for bar in bars:
        bar.set_height(0)
        bar.set_color('steelblue')
    pred_text.set_text('')
    fig.canvas.draw_idle()

btn.on_clicked(clear)

plt.tight_layout(rect=[0, 0.08, 1, 1])
plt.show()
```

```
/var/folders/31/yr0kwy9s1b57dcqcfv96tqb00000gn/T/ipykernel_75412/1235942900.py:8
4: UserWarning: This figure includes Axes that are not compatible with
tight_layout, so results might be incorrect.
  plt.tight_layout(rect=[0, 0.08, 1, 1])
```


Draw a digit here
Model output
1.0
0.8
0.6
0.4
0.2
0.0
Probability
0
1
2
3
4
5
6
7
8
9
Digit class
Clear

[ ]: