Neural networks from scratch, then in PyTorch

Build logistic regression and a one-hidden-layer network with NumPy only (forward pass, backprop, gradient descent), check the gradients, then build the same network in PyTorch, and finish with a deeper network on handwritten digits.

Open in Google Colab · part of the inPace course library. Slides © DeepLearning.AI (CC BY-SA 2.0); this notebook: inPace (CC BY-SA 4.0).

Run it top to bottom (Runtime → Run all in Colab). Each part says what to look for.

```
# Works in Google Colab as it is (PyTorch is already installed there).
# On your own computer:  pip install torch torchvision scikit-learn scipy matplotlib pandas
# One framework per notebook: PyTorch and TensorFlow can clash when loaded into the same Python process.
import numpy as np, matplotlib.pyplot as plt, torch
from torch import nn
torch.manual_seed(0); np.random.seed(0)
device = "cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu"
print("PyTorch", torch.__version__, "| device:", device)
```

1. The data: two moons

Two classes that a straight line can't separate. 1,000 points: 800 to train, 200 to check (dev).

```
from sklearn.datasets import make_moons
X, y = make_moons(1000, noise=0.2, random_state=0)
X_tr, y_tr, X_dv, y_dv = X[:800], y[:800], X[800:], y[800:]
# The course's convention: one example per COLUMN, so X has shape (n_x, m).
Xt, Yt, Xd, Yd = X_tr.T, y_tr.reshape(1, -1), X_dv.T, y_dv.reshape(1, -1)
print("X:", Xt.shape, " Y:", Yt.shape)
plt.scatter(X_tr[:, 0], X_tr[:, 1], c=y_tr, cmap="coolwarm", s=8);
plt.title("training set"); plt.show()
```

2. Logistic regression, vectorized (no loops over examples)

`Z = wᵀX + b`, `A = σ(Z)`, cost = mean cross-entropy, then `dZ = A − Y`, `dw = X dZᵀ / m`, `db = mean(dZ)`.

```
def sigmoid(z): return 1 / (1 + np.exp(-z))

def logistic_regression(X, Y, lr=0.5, steps=2000):
    w, b = np.zeros((X.shape[0], 1)), 0.0
    m = X.shape[1]
    for i in range(steps):
```

```
        A = sigmoid(w.T @ X + b)                                    # forward, all
m examples at once
        cost = -np.mean(Y * np.log(A + 1e-12) + (1 - Y) * np.log(1 - A + 1e-12))
        dZ = A - Y                                                  # backward
        w -= lr * (X @ dZ.T) / m
        b -= lr * dZ.mean()
    return w, b, cost

w, b, cost = logistic_regression(Xt, Yt)
acc = lambda P, Y: (P == Y).mean()
print(f"final cost {cost:.3f} | train acc {acc((sigmoid(w.T @ Xt + b) > .5),
Yt):.3f} | dev acc {acc((sigmoid(w.T @ Xd + b) > .5), Yd):.3f}")
```

```
def plot_boundary(predict, title):
    xx, yy = np.meshgrid(np.linspace(-1.5, 2.5, 300), np.linspace(-1, 1.5, 300))
    zz = predict(np.c_[xx.ravel(), yy.ravel()].T).reshape(xx.shape)
    plt.contourf(xx, yy, zz, levels=20, cmap="coolwarm", alpha=.5)
    plt.scatter(X_tr[:, 0], X_tr[:, 1], c=y_tr, cmap="coolwarm", s=6);
plt.title(title); plt.show()

plot_boundary(lambda P: sigmoid(w.T @ P + b), "logistic regression: one straight
line")
```

3. One hidden layer, from scratch

Shapes to keep in your head: `W1 (n_h, n_x)`, `b1 (n_h, 1)`, `W2 (1, n_h)`, `b2 (1, 1)`. tanh in the hidden layer, sigmoid at the output. Small random weights (never zeros: every hidden unit would stay identical).

```
def init(n_x, n_h, seed=1):
    rng = np.random.default_rng(seed)
    return {"W1": rng.normal(0, 0.5, (n_h, n_x)), "b1": np.zeros((n_h, 1)),
            "W2": rng.normal(0, 0.5, (1, n_h)), "b2": np.zeros((1, 1))}

def forward(p, X):
    Z1 = p["W1"] @ X + p["b1"]; A1 = np.tanh(Z1)
    Z2 = p["W2"] @ A1 + p["b2"]; A2 = sigmoid(Z2)
    return A2, (A1, A2)

def cost_fn(A2, Y):
    return -np.mean(Y * np.log(A2 + 1e-12) + (1 - Y) * np.log(1 - A2 + 1e-12))

def backward(p, X, Y, cache):
    A1, A2 = cache; m = X.shape[1]
    dZ2 = A2 - Y
    dW2 = dZ2 @ A1.T / m;  db2 = dZ2.mean(axis=1, keepdims=True)
    dZ1 = (p["W2"].T @ dZ2) * (1 - A1 ** 2)          # tanh'(z) = 1 - tanh(z)^2
    dW1 = dZ1 @ X.T / m;   db1 = dZ1.mean(axis=1, keepdims=True)
    return {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2}

p, costs = init(2, 8), []
for i in range(5000):
    A2, cache = forward(p, Xt)
    g = backward(p, Xt, Yt, cache)
    for k in p: p[k] -= 1.0 * g[k]
    costs.append(cost_fn(A2, Yt))
print(f"train acc {acc(forward(p, Xt)[0] > .5, Yt):.3f} | dev acc {acc(forward(p,
```

```
Xd)[0] > .5, Yd):.3f}")
plt.plot(costs); plt.yscale("log"); plt.title("cost"); plt.xlabel("step");
plt.show()
plot_boundary(lambda P: forward(p, P)[0], "8 tanh units: a curved boundary")
```

4. Gradient check: is our backprop right?

Compare each analytic gradient with `(J(θ+ε) − J(θ−ε)) / 2ε`. The relative difference should be around `1e-7` or smaller; `1e-3` means a bug.

```
def grad_check(p, X, Y, eps=1e-6):
    _, cache = forward(p, X); g = backward(p, X, Y, cache)
    num, ana = [], []
    for k in p:
        for idx in np.ndindex(p[k].shape):
            old = p[k][idx]
            p[k][idx] = old + eps; jp = cost_fn(forward(p, X)[0], Y)
            p[k][idx] = old - eps; jm = cost_fn(forward(p, X)[0], Y)
            p[k][idx] = old
            num.append((jp - jm) / (2 * eps)); ana.append(g[k][idx])
    num, ana = np.array(num), np.array(ana)
    return np.linalg.norm(num - ana) / (np.linalg.norm(num) + np.linalg.norm(ana))

print("relative difference:", grad_check(init(2, 4), Xt[:, :50], Yt[:, :50]))
```

5. The same network in PyTorch

PyTorch stores one example per **row**, so `X` is `(m, n_x)`. `loss.backward()` does section 3's backward pass for us (autograd). `BCEWithLogitsLoss` takes raw scores (logits) and is more stable than sigmoid + log.

```
Xtr_t, ytr_t = torch.tensor(X_tr, dtype=torch.float32), torch.tensor(y_tr,
dtype=torch.float32)
Xdv_t, ydv_t = torch.tensor(X_dv, dtype=torch.float32), torch.tensor(y_dv,
dtype=torch.float32)

class ShallowNet(nn.Module):
    def __init__(self, n_h=8):
        super().__init__()
        self.hidden = nn.Linear(2, n_h)
        self.out = nn.Linear(n_h, 1)
    def forward(self, x):
        return self.out(torch.tanh(self.hidden(x))).squeeze(1)   # logits

model = ShallowNet().to(device)
opt = torch.optim.SGD(model.parameters(), lr=1.0)
loss_fn = nn.BCEWithLogitsLoss()
for step in range(3000):
    opt.zero_grad()
    loss = loss_fn(model(Xtr_t.to(device)), ytr_t.to(device))
    loss.backward()        # autograd: every gradient in one call
    opt.step()
with torch.no_grad():
    dev_acc = ((model(Xdv_t.to(device)) > 0).float().cpu() ==
```

```
ydv_t).float().mean().item()
print(f"final loss {loss.item():.3f} | dev acc {dev_acc:.3f}")
```

7. A deeper network on handwritten digits

1,797 images of 8×8 pixels, 10 classes. Two hidden ReLU layers, softmax output (inside `CrossEntropyLoss`), Adam.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
d = load_digits()
Xa, Xb, ya, yb = train_test_split(d.data / 16.0, d.target, test_size=0.25,
random_state=0)
Xa, Xb = torch.tensor(Xa, dtype=torch.float32), torch.tensor(Xb,
dtype=torch.float32)
ya, yb = torch.tensor(ya), torch.tensor(yb)
deep = nn.Sequential(nn.Linear(64, 128), nn.ReLU(), nn.Linear(128, 64), nn.ReLU(),
nn.Linear(64, 10))
opt = torch.optim.Adam(deep.parameters(), lr=1e-3)
for epoch in range(300):
    opt.zero_grad(); loss = nn.functional.cross_entropy(deep(Xa), ya);
loss.backward(); opt.step()
with torch.no_grad():
    print(f"parameters: {sum(p.numel() for p in deep.parameters()):,} | test
accuracy {(deep(Xb).argmax(1) == yb).float().mean():.3f}")
fig, ax = plt.subplots(1, 8, figsize=(10, 1.6))
for i, a in enumerate(ax):
    a.imshow(Xb[i].reshape(8, 8), cmap="gray_r"); a.set_title(f"pred
{deep(Xb[i:i+1]).argmax().item()}"); a.axis("off")
plt.show()
```

Try this

1. Change `n_h` in section 3 to 1, 2, 4 and 50. When does the boundary curve? When does it overfit?
2. Replace `tanh` with ReLU in section 3 (`np.maximum(0, Z1)`, slope `Z1 > 0`). Rerun the gradient check.
3. Initialize `W1` with zeros. What happens to the hidden units, and why?
4. In section 7, remove one hidden layer. How much does test accuracy change?