Neural Networks

Hypotheses, activations, forward propagation, losses, and initialization

2019-02-15

Table of contents

1 Neural network hypothesis **1**

2 Activation functions **2**

3 Forward propagation **2**

4 Loss functions **2**

5 Parameter initialisation **2**

6 Minimal NumPy forward pass **3**

7 Practical habits **4**

8 Where to go next **4**

Neural networks can feel abstract until you see how the pieces fit together. This walkthrough links the core ideas to the supporting math and code so each component slots into place.

1 Neural network hypothesis

A feed-forward neural network applies a sequence of affine transformations followed by element-wise nonlinear functions. Each layer produces intermediate activations that serve as the inputs to the next layer.

Mathematics. In linear or logistic regression, the hypothesis maps inputs x to output $\hat{y}$ via a single affine transform (and possibly a sigmoid). A neural network composes multiple transforms and nonlinearities across layers:

Let the network have L layers (not counting the input as a layer). For layer $\ell = 1, \ldots, L$: - Parameters: $W^{[\ell]} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $b^{[\ell]} \in \mathbb{R}^{n_\ell}$ - Pre-activation: $z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}$ - Activation: $a^{[\ell]} = g^{[\ell]}(z^{[\ell]})$

with $a^{[0]} = x$. The hypothesis is the function $h_\Theta(x) = a^{[L]}$ produced by repeated **forward propagation**.

Batch view. Vectorised over a batch $X \in \mathbb{R}^{n_0 \times m}$: $Z^{[\ell]} = W^{[\ell]} A^{[\ell-1]} + b^{[\ell]} \mathbf{1}^\top, \quad A^{[\ell]} = g^{[\ell]}(Z^{[\ell]})$.

2 Activation functions

Activation functions introduce nonlinearity, allowing the network to represent decision boundaries that are not purely linear. They also control gradient flow during training, so the choice of function affects both expressiveness and optimisation stability.

What to know. - **Sigmoid** $\sigma(z) = \frac{1}{1+e^{-z}}$: outputs $(0, 1)$. Great for probabilities; watch out for saturated gradients when $|z|$ is large. - **Tanh** $\tanh(z)$: zero-centered, mitigating bias shifts; still saturates at the extremes. - **ReLU** $\max(0, z)$: keeps positive values, zeroes the rest. Trains quickly but can "die" if gradients vanish. - **Leaky ReLU / GELU / ELU**: variants that smooth or allow a small negative slope to avoid dead neurons. - **Softmax** for multi-class output: $\text{softmax}(z)_k = \dfrac{e^{z_k}}{\sum_j e^{z_j}}$ to turn logits into probabilities.

Default playbook. ReLU (or GELU) covers most hidden layers, sigmoid suits binary outputs, softmax suits multi-class heads, and tanh remains useful when symmetry around zero is helpful.

---

3 Forward propagation

The forward pass iterates through layers, computing pre-activations and activations in order. Intermediate values are stored for later use in gradient computations and for inspection when debugging.

Algorithm. For each layer $\ell = 1 \dots L$: 1. $Z^{[\ell]} = W^{[\ell]} A^{[\ell-1]} + b^{[\ell]} \mathbf{1}^\top$ 2. $A^{[\ell]} = g^{[\ell]}(Z^{[\ell]})$

Return $A^{[L]}$ as the prediction(s). Vectorisation keeps this efficient for batches.

Numerical stability tips. Use the `logsumexp` trick for softmax and clamp probabilities to $[\varepsilon, 1-\varepsilon]$ when computing log-losses.

---

4 Loss functions

Loss functions convert network predictions and reference targets into a single scalar objective. Minimising that scalar guides the optimiser toward parameters that align predictions with observed data.

Core losses. - **Binary classification:** $J = -\frac{1}{m} \sum_i y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$ with $\hat{y} = \sigma(z^{[L]})$. - **Multi-class (softmax):** $J = -\frac{1}{m} \sum_i \log \hat{p}_{i,y_i}$ where $\hat{p}_i = \text{softmax}(z_i^{[L]})$. - **Regression:** Mean squared error $J = \frac{1}{2m} \sum_i \|\hat{y}_i - y_i\|^2$.

Add **regularisation** (e.g., L2 weight decay) to reduce overfitting: $J_\lambda = J + \frac{\lambda}{2m} \sum_\ell \|W^{[\ell]}\|_F^2$.

---

5 Parameter initialisation

Suitable initial values keep activations and gradients within reasonable ranges at the start of training. Poor choices can create symmetric neurons that behave identically or cause activations to saturate before learning begins.

Guidelines. - **Avoid** all-zero weights; symmetry would keep neurons identical. - **Xavier/Glorot** for tanh or sigmoid: $W^{[\ell]} \sim \mathcal{N}(0, \frac{2}{n_{\ell-1}+n_\ell})$ or the uniform equivalent. - **He/Kaiming** for ReLU: $W^{[\ell]} \sim \mathcal{N}(0, \frac{2}{n_{\ell-1}})$. - Biases $b^{[\ell]} = 0$ are typically fine.

---

6 Minimal NumPy forward pass

This reference implementation keeps the forward equations explicit for inspection or quick experiments.

```
import numpy as np

def sigmoid(z):
    out = np.empty_like(z, dtype=float)
    pos = z >= 0
    neg = ~pos
    out[pos] = 1.0 / (1.0 + np.exp(-z[pos]))
    ez = np.exp(z[neg])
    out[neg] = ez / (1.0 + ez)
    return out

def relu(z):
    return np.maximum(0, z)

def he_init(n_in, n_out, rng):
    return rng.normal(0, np.sqrt(2.0/n_in), size=(n_out, n_in))

def forward(X, params):
    "Run forward propagation over layers defined by params."
    A = X
    for (W, b, act) in params[:-1]:
        Z = W @ A + b[:, None]
        A = relu(Z) if act == "relu" else np.tanh(Z)
    W, b, _ = params[-1]
    ZL = W @ A + b[:, None]
    return sigmoid(ZL)

# Example architecture: 2 -> 8 -> 4 -> 1 (binary)
rng = np.random.default_rng(0)
n_in, m = 2, 512
X = rng.normal(size=(n_in, m))

params = []
W1 = he_init(n_in, 8, rng);  b1 = np.zeros(8)
W2 = he_init(8, 4, rng);     b2 = np.zeros(4)
W3 = he_init(4, 1, rng);     b3 = np.zeros(1)
params = [(W1, b1, "relu"), (W2, b2, "relu"), (W3, b3, "sigmoid")]

AL = forward(X, params)
print(AL.shape, AL.min(), AL.max())  # (1, m), ~[0,1]
```

For multi-class problems, replace the final activation with softmax and compute cross-entropy loss on one-hot targets.

7 Practical habits

- **Scaling.** Standardise inputs and reuse the same transform at inference time.
- **Regularisation.** Weight decay, dropout, and early stopping control variance.
- **Batching.** Mini-batches stabilise gradient estimates and hardware throughput.
- **Monitoring.** Track training versus validation metrics to catch drift early.
- **Reproducibility.** Fix RNG seeds and log configurations so experiments can be replayed.

8 Where to go next

- Implement backpropagation for the architectures above to connect gradients to the forward equations.
- Explore modern activation choices (GELU, Swish) and compare training curves.
- Experiment with initialisation scalings on deeper networks to observe exploding or vanishing activations.