

COMPSCI 1810 Spring 2026 Section 5

Neural Network Training and Architectures

Authored by Zidi Xiong

1 Neural Network Part II: Training Neural Networks

Training loop (one mental model). Training a neural network is repeating the same 4-step loop:

1. **Forward pass:** compute predictions $\hat{y} = f_{\theta}(x)$.
2. **Loss:** compute a scalar objective $L(\hat{y}, y)$.
3. **Backward pass (backprop):** compute gradients $\nabla_{\theta} L$ efficiently.
4. **Update (SGD):** move parameters in the negative-gradient direction.

1.1 Backpropagation

The core idea is straightforward: a neural network is just a composition of simple functions, and the chain rule tells us how to differentiate a composition. What backpropagation adds is *efficiency*. A naïve application of the chain rule would recompute many shared sub-expressions. Backprop avoids this by computing the forward pass first (storing intermediate results), and then sweeping backward through the network, reusing those stored values.

Concretely, backprop works in two phases:

- **Forward pass:** compute and store all intermediate values (pre-activations $z^{(\ell)}$, activations $h^{(\ell)}$) and the final loss L .
- **Backward pass:** starting from $\frac{\partial L}{\partial L} = 1$, propagate gradients layer by layer from the output back to the input, using the stored intermediate values.

To make the Backward pass more concrete: suppose we have a tiny one-layer network $\hat{y} = w^{\top} x$ with loss $L = \frac{1}{2}(\hat{y} - y)^2$. The gradient with respect to $\hat{y}$ is $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$. Multiplying by the local Jacobian $\frac{\partial \hat{y}}{\partial w} = x$ gives us $\frac{\partial L}{\partial w} = (\hat{y} - y) x$.

Common derivatives of activation functions. When the backward pass hits an activation function, you need its derivative. You will use these repeatedly:

$$\text{ReLU}'(u) = 1[u > 0], \quad \sigma'(u) = \sigma(u)(1 - \sigma(u)), \quad \tanh'(u) = 1 - \tanh^2(u).$$

Here $\sigma(u)$ denotes the sigmoid function, defined as

$$\sigma(u) = \frac{1}{1 + e^{-u}}.$$

Notice that $\sigma'(u)$ is maximized at $u = 0$ where it equals $\frac{1}{4}$, and it decays toward 0 for large $|u|$. This will matter a great deal when we discuss vanishing gradients below.

1.2 Exercise: Backpropagation Gradients

Consider the following scalar (1D) two-layer network with a sigmoid hidden layer and a linear output:

$$\begin{aligned}z &= w_1x + b_1 \\h &= \sigma(z) \\\hat{y} &= w_2h + b_2 \\L &= \frac{1}{2}(\hat{y} - y)^2\end{aligned}$$

where $\sigma(t) = \frac{1}{1+e^{-t}}$.

Let $x = 2$, $y = 1$, and parameters $w_1 = 0.5$, $b_1 = -1$, $w_2 = 3$, $b_2 = 0$.

1. Compute z , h , and $\hat{y}$.
2. Compute the exact gradients $\frac{\partial L}{\partial w_2}$ and $\frac{\partial L}{\partial b_2}$.
3. Compute the exact gradients $\frac{\partial L}{\partial w_1}$ and $\frac{\partial L}{\partial b_1}$.
4. What is the sign of $\frac{\partial L}{\partial w_2}$, and what does that imply about how SGD will update w_2 for this example?

Solution: We first do the forward pass:

$$z = w_1x + b_1 = 0.5 \cdot 2 - 1 = 0, \quad h = \sigma(0) = \frac{1}{2}, \quad \hat{y} = w_2h + b_2 = 3 \cdot \frac{1}{2} + 0 = 1.5.$$

Then $L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(0.5)^2 = 0.125$ and $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = 0.5$.

- For the output layer: $\frac{\partial \hat{y}}{\partial w_2} = h$, $\frac{\partial \hat{y}}{\partial b_2} = 1$, so

$$\frac{\partial L}{\partial w_2} = (\hat{y} - y)h = 0.5 \cdot 0.5 = 0.25, \quad \frac{\partial L}{\partial b_2} = (\hat{y} - y) = 0.5.$$

- Backprop to h : $\frac{\partial L}{\partial h} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} = (\hat{y} - y)w_2 = 0.5 \cdot 3 = 1.5$.
- Through the sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, so here $\sigma'(0) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, hence $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial h} \sigma'(z) = 1.5 \cdot \frac{1}{4} = 0.375$.
- Into (w_1, b_1) : $\frac{\partial z}{\partial w_1} = x$ and $\frac{\partial z}{\partial b_1} = 1$, so

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z} x = 0.375 \cdot 2 = 0.75, \quad \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z} = 0.375.$$

Since $\frac{\partial L}{\partial w_2} > 0$, gradient descent updates $w_2 \leftarrow w_2 - \eta \frac{\partial L}{\partial w_2}$ will decrease w_2 , reducing the output $\hat{y}$ (which is currently too large).

1.3 Vanishing and Exploding Gradients

Now that you have seen backprop in action, we can discuss the central engineering challenge of training deep networks. The backward pass multiplies Jacobian matrices together, one per layer. What happens when you multiply many matrices together?

The core issue. Consider an L -layer network. The gradient of the loss with respect to a parameter in layer ℓ involves a product of roughly $L - \ell$ Jacobian factors. If the typical singular values of these Jacobians are:

- **Less than 1:** the product shrinks exponentially with depth, so gradients **vanish** in early layers. The early layers receive almost no learning signal.
- **Greater than 1:** the product grows exponentially, so gradients **explode**. Weight updates become enormous and training diverges.

A concrete example: why sigmoid networks are hard to train deeply. Recall from the exercise above that $\sigma'(z) \leq \frac{1}{4}$ everywhere (with equality only at $z = 0$). If all layers use sigmoid activations, the backward pass multiplies by $\sigma'(z^{(\ell)})$ at every layer. Even in the best case, this means the gradient is scaled by at most $(\frac{1}{4})^L$. For a 10-layer network, that is $(0.25)^{10} \approx 10^{-6}$, meaning the gradient arriving at layer 1 is a *million times smaller* than the gradient at layer 10. In practice, the activations are rarely all at $z = 0$, so the situation is typically even worse. This is the primary reason deep sigmoid/tanh networks were considered untrainable for decades, and a major motivation for the shift to ReLU (where $\text{ReLU}'(z) = 1$ for $z > 0$, so gradients do not shrink through active neurons).

The weight matrix side. Even with ReLU, the weight matrices themselves can cause problems. If $\|W^{(\ell)}\|$ (the spectral norm) is systematically greater than 1 across layers, the gradient product will grow exponentially. If it is systematically less than 1, the product shrinks. This is especially visible in RNNs, where the *same* recurrence matrix W_{hh} is applied at every time step — so you are literally computing powers of a single matrix, and the compounding effect can be extreme.

What people do about it. These are not just “tricks” — each one directly addresses the mathematical cause of the problem:

- **ReLU-like activations:** ReLU has derivative 1 for positive inputs, avoiding the $\leq \frac{1}{4}$ bottleneck of sigmoid. This is the single biggest reason modern deep networks are trainable.
- **Careful initialization:** by choosing the initial weight scale so that each layer approximately preserves the variance of both activations (forward) and gradients (backward), we start training in a regime where neither vanishing nor exploding has yet occurred.
- **Residual connections:** adding a skip path $x \mapsto x + F(x)$ means the gradient of the skip path is the identity, giving gradients a “highway” that bypasses the Jacobian products entirely. We will see this in Part III.

- **Gradient clipping:** a brute-force safeguard for exploding gradients. If $\|\nabla_{\theta} L\| > \tau$ for some threshold τ , rescale it to $\tau \cdot \frac{\nabla_{\theta} L}{\|\nabla_{\theta} L\|}$. This does not solve the root cause but prevents catastrophic divergence.
- **Normalization layers (BatchNorm, LayerNorm, etc.):** normalization explicitly controls the distribution of pre-activations by re-centering and re-scaling them at each layer. This prevents inputs before each layer has well-controlled distributions.

1.4 Regularization: Controlling Overfitting

A sufficiently large neural network can memorize any finite training set, including its noise. This means that even if training loss reaches zero, the network may perform terribly on unseen data. The following techniques help control this overfitting.

Weight regularization. Just as ridge regression adds an L_2 penalty to discourage large weights, we can add a regularization term to the neural network loss. Given a base loss $L_0(\theta)$, the regularized objective becomes:

$$L(\theta) = L_0(\theta) + \lambda \Omega(\theta),$$

where $\lambda > 0$ controls the strength of regularization and $\Omega(\theta)$ is a penalty on the weights. The two most common choices are:

- **L_2 regularization:** $\Omega(\theta) = \frac{1}{2} \|\theta\|_2^2 = \frac{1}{2} \sum_i \theta_i^2$. This penalizes large weights and encourages the network to spread information across many small weights rather than relying on a few large ones.
- **L_1 regularization:** $\Omega(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$. This encourages *sparsity*: many weights are shrink to zero, effectively pruning connections. L_1 is less commonly used in deep learning than L_2 , but it can be useful when you want a compact model.

Dropout. Dropout is a remarkably simple and effective regularization technique. During each training step, every hidden neuron is independently “dropped” (set to zero) with probability p . This means the network cannot rely on any single neuron or any specific combination of neurons — it must learn redundant representations that are robust to missing units.

2 Neural Network Part III: Architectures

2.1 Convolutional Neural Networks (CNNs)

Consider a 256×256 RGB image. As a vector, this input has $256 \times 256 \times 3 \approx 200,000$ dimensions. A single fully connected hidden layer of width 1024 would require $200,000 \times 1024 \approx 200$ million parameters for one layer. This is computationally expensive. CNNs exploit two key ideas to address this: **local connectivity** (each neuron only looks at a small patch) and **weight sharing** (the same filter is applied everywhere).

Local receptive fields and weight sharing. Instead of connecting every neuron to every input pixel, each unit in a convolutional layer connects only to a small **local neighborhood** (its receptive field). Moreover, the same filter weights are **shared** across all spatial locations. This builds in the inductive bias that local patterns (edges, textures, corners) can appear anywhere in the image, so the network should look for them everywhere using the same detector.

Convolution in 1D (a concrete example). Given an input sequence $x \in \mathbb{R}^n$ and a filter $w \in \mathbb{R}^k$, a (valid) 1D convolution produces an output $y \in \mathbb{R}^{n-k+1}$ with

$$y[t] = \sum_{i=0}^{k-1} w[i] x[t+i].$$

For a concrete example: let $x = [1, 3, 2, 5, 4]$ and $w = [1, -1]$ (a simple “difference” filter with $k = 2$). Then:

$$\begin{aligned} y[0] &= 1 \cdot 1 + (-1) \cdot 3 = -2, & y[1] &= 1 \cdot 3 + (-1) \cdot 2 = 1 \\ y[2] &= 1 \cdot 2 + (-1) \cdot 5 = -3, & y[3] &= 1 \cdot 5 + (-1) \cdot 4 = 1. \end{aligned}$$

So $y = [-2, 1, -3, 1]$. Notice that this filter detects *changes* in the input — it outputs a large magnitude wherever consecutive values differ.

Convolution in 2D, channels, stride, padding. For an image-like tensor $X \in \mathbb{R}^{H \times W \times C_{\text{in}}}$, a convolutional layer applies a bank of filters, each of shape $K_h \times K_w \times C_{\text{in}}$, to produce C_{out} output channels (one per filter). Each filter slides across the spatial dimensions with stride S and optional zero-padding P . The spatial output size is:

$$H_{\text{out}} = \left\lfloor \frac{H + 2P - K_h}{S} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W + 2P - K_w}{S} \right\rfloor + 1.$$

Pooling. After convolution and a nonlinearity, it is common to downsample feature maps using **pooling** (max-pooling or average-pooling). Pooling reduces the spatial resolution, which serves two purposes: it makes the representation more compact (fewer values to process downstream), and it introduces a degree of translation invariance (small shifts in the input lead to the same pooled output). A basic CNN block is:

Conv $\rightarrow$ Nonlinearity (e.g., ReLU) $\rightarrow$ (optional) Pooling.

Equivariance and why it matters. Convolutions are **translation equivariant**: if you shift the input image by 3 pixels to the right, the output feature map shifts by 3 pixels to the right as well. This means a cat detector does not need to learn “cat in the top-left corner” and “cat in the bottom-right corner” separately — it learns “cat” once, and equivariance ensures this detector works everywhere. Pooling and global averaging can then introduce (approximate) **translation invariance**, where the final representation is insensitive to the exact position. Equivariance in the early layers (to detect features wherever they are) combined with invariance in the later layers (to classify regardless of position) is a key reason CNNs are so effective on visual tasks.

2.2 Exercise: CNN Output Shapes and Parameters

An input image has shape $32 \times 32 \times 3$ (height $\times$ width $\times$ channels).

1. You apply a conv layer with 16 filters of size 5×5 , stride $S = 1$, and padding $P = 0$ (“valid”). What is the output shape?
2. Same conv layer, but now with padding $P = 2$. What is the output shape?
3. You then apply 2×2 max-pooling with stride 2 to the output of part (2). What is the new shape?
4. How many learnable parameters does the conv layer in part (1) have (including biases)?
5. (Conceptual) If you replaced this conv layer with a fully connected layer mapping the $32 \times 32 \times 3$ input to the same number of output neurons ($28 \times 28 \times 16$), how many parameters would that require? What does this comparison tell you about the efficiency of weight sharing?

Solution:

1. Using the formula: $H_{\text{out}} = \lfloor (32 + 0 - 5)/1 \rfloor + 1 = 28$ and similarly $W_{\text{out}} = 28$. Each filter produces one output channel, and we have 16 filters, so the output shape is $28 \times 28 \times 16$.
2. With $P = 2$: $H_{\text{out}} = \lfloor (32 + 4 - 5)/1 \rfloor + 1 = 32$ and $W_{\text{out}} = 32$, so $32 \times 32 \times 16$.
3. Max-pooling with a 2×2 window and stride 2 halves both spatial dimensions: $16 \times 16 \times 16$. Note that pooling does not change the number of channels — it only affects spatial resolution.
4. Each filter has shape $5 \times 5 \times 3 = 75$ weights, plus 1 bias, giving 76 parameters per filter. With 16 filters: $16 \times 76 = 1,216$ parameters total.
5. A fully connected layer would need $(32 \times 32 \times 3) \times (28 \times 28 \times 16) = 3,072 \times 12,544 \approx 38.5$ million parameters. Compared to the conv layer’s 1,216 parameters, this is roughly 30,000 $\times$ more.

2.3 Recurrent Neural Networks (RNNs)

Feedforward networks (including CNNs) assume that inputs are fixed-size and independent. But many real-world tasks involve **sequences**: language (a sentence is a sequence of words), time series (stock prices, sensor readings), music, and even video (a sequence of frames). The key challenge is that sequence elements depend on each other, often over long distances. We need an architecture that can process variable-length inputs and maintain some notion of memory.

Hidden state as memory. An RNN processes a sequence $x_1, x_2, \dots, x_T$ one element at a time, maintaining a **hidden state** h_t that summarizes everything the network has seen so far:

$$h_t = \phi(W_{hh}h_{t-1} + W_{xh}x_t + b_h),$$

and optionally produces an output $o_t = W_{hy}h_t + b_y$ at each step. You can think of h_t as a compressed summary of the sequence up to time t . The critical point is that the *same* weight matrices W_{hh}, W_{xh} are applied at every time step — this is weight sharing across time, analogous to how CNNs share weights across space.

Training: backpropagation through time (BPTT). To train an RNN, we “unroll” the recurrence across time steps, creating what looks like a very deep feedforward network (one “layer” per time step), and apply standard backprop. This is called **backpropagation through time**. The problem is immediate: because the *same* matrix W_{hh} is multiplied at every step, the gradient involves products like $(W_{hh}^\top)^T$ for a sequence of length T . If the spectral radius of W_{hh} is greater than 1, gradients explode; if less than 1, they vanish. This is the same vanishing/exploding gradient problem from Part II, but it hits RNNs especially hard because the “depth” equals the sequence length, which can be hundreds or thousands of steps.

LSTM: gated memory cells. Long Short-Term Memory (LSTM) networks address the vanishing gradient problem by introducing a separate **memory cell** c_t that is updated via addition rather than repeated multiplication. An LSTM uses three gates — each a sigmoid layer that outputs values in $[0, 1]$:

- **Forget gate:** $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$ — what fraction of the old memory to keep.
- **Input gate:** $i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$ — how much of the new candidate to write.
- **Output gate:** $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$ — how much of the cell to expose as h_t .

The cell and hidden state are then:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \quad h_t = o_t \odot \tanh(c_t),$$

where $\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$ is the candidate update and $\odot$ is element-wise multiplication. Because c_t is a *sum* of old and new information (not a product of matrices), gradients can flow through the cell state across many time steps without vanishing.

GRU: a simpler alternative. The Gated Recurrent Unit (GRU) simplifies the LSTM to just two gates:

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z), \quad r_t = \sigma(W_r[h_{t-1}, x_t] + b_r),$$

$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h), \quad h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t.$$

The **update gate** z_t blends old and new: when $z_t \approx 0$ the hidden state is copied forward unchanged; when $z_t \approx 1$ it is replaced. The **reset gate** r_t controls how much past information feeds into the candidate $\tilde{h}_t$. GRUs have fewer parameters than LSTMs and often perform comparably in practice.

2.4 Exercise: RNN gradient

Consider a 1D (scalar) vanilla RNN with tanh nonlinearity:

$$h_t = \tanh(w_h h_{t-1} + w_x x_t), \quad h_0 = 0$$

$$\hat{y} = h_t, \quad L = \frac{1}{2}(\hat{y} - y)^2$$

1. Let $x_1 = 1, x_2 = 0, y = 0, \hat{y} = h_2$, and parameters $w_h = 1, w_x = 1$. Compute h_1 and h_2 .
2. Using the setting in (1), compute the exact gradient $\frac{\partial L}{\partial w_h}$ (you will need to backprop through time for two steps).
3. Suppose the sequence has length K , so $\hat{y} = h_K$. Write closed-form expressions for $\frac{\partial L}{\partial w_h}$ and $\frac{\partial L}{\partial w_x}$.
4. Using your answer to (3), explain when and why the gradient of an RNN vanishes as K grows.

Solution:

1. Forward:

$$h_1 = \tanh(w_h h_0 + w_x x_1) = \tanh(1), \quad h_2 = \tanh(w_h h_1 + w_x x_2) = \tanh(h_1).$$

2. Since $y = 0$, we have $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = h_2$. Let $a_1 = w_h h_0 + w_x x_1$ and $a_2 = w_h h_1 + w_x x_2$. Then $h_t = \tanh(a_t)$ and $\tanh'(u) = 1 - \tanh^2(u)$.

$$\frac{\partial L}{\partial w_h} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial a_2} \frac{\partial a_2}{\partial w_h} + \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial a_2} \frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1} \frac{\partial a_1}{\partial w_h}.$$

Now $\frac{\partial L}{\partial h_2} = h_2$, $\frac{\partial h_2}{\partial a_2} = 1 - h_2^2$, $\frac{\partial a_2}{\partial w_h} = h_1$, $\frac{\partial a_2}{\partial h_1} = w_h = 1$, $\frac{\partial h_1}{\partial a_1} = 1 - h_1^2$, and $\frac{\partial a_1}{\partial w_h} = h_0 = 0$. So the second term is zero and

$$\frac{\partial L}{\partial w_h} = h_2(1 - h_2^2)h_1.$$

3. Let $a_t = w_h h_{t-1} + w_x x_t$ so that $h_t = \tanh(a_t)$. By BPTT, w_h influences L through every time step, giving

$$\frac{\partial L}{\partial w_h} = \frac{\partial L}{\partial h_K} \sum_{t=1}^K \left(\prod_{k=t+1}^K (1 - h_k^2) w_h \right) (1 - h_t^2) h_{t-1},$$

where the product equals 1 when $t = K$. Each factor $(1 - h_k^2) w_h$ arises from $\frac{\partial h_k}{\partial h_{k-1}} = \tanh'(a_k) w_h = (1 - h_k^2) w_h$, and the explicit dependence on w_h at step t contributes $\frac{\partial a_t}{\partial w_h} \Big|_{\text{explicit}} = h_{t-1}$.

Similarly, since $\frac{\partial a_t}{\partial w_x} \Big|_{\text{explicit}} = x_t$,

$$\frac{\partial L}{\partial w_x} = \frac{\partial L}{\partial h_K} \sum_{t=1}^K \left(\prod_{k=t+1}^K (1 - h_k^2) w_h \right) (1 - h_t^2) x_t.$$

4. The key quantity controlling gradient magnitude is the product

$$\prod_{k=t+1}^K (1 - h_k^2) w_h.$$

This product has $K - t$ factors; for the earliest time step $t = 1$ there are $K - 1$ factors. As each factor has magnitude $|w_h|(1 - h_k^2) < |w_h| < 1$, so the product decays *exponentially* in $K - t$. Gradients from early time steps become negligibly small.

2.5 Residual Connections

By now you have seen a recurring theme: depth is powerful (it lets networks compose features hierarchically), but depth is also dangerous (gradients vanish or explode). Residual connections, introduced in the ResNet paper, are an elegantly simple idea that largely resolves this tension.

The idea. Instead of asking a block to learn a full transformation $\mathcal{F}(x)$, a residual block learns only the *residual* — the difference between the desired output and the input:

$$x \mapsto x + F(x).$$

This seemingly minor change has profound consequences.

Why this helps optimization. If the best thing for a layer to do is approximately preserve the input (which is common in deep networks — not every layer needs to dramatically transform the representation), then the network only needs to learn $F(x) \approx 0$, i.e. the weights just need to be small. Without the skip connection, the network must learn $\mathcal{F}(x) \approx x$, which requires precise coordination among the weights.

Why this helps gradient flow. Consider the gradient through a residual block. Let $y = x + F(x)$. Then:

$$\frac{\partial y}{\partial x} = I + \frac{\partial F}{\partial x}.$$

Even if $\frac{\partial F}{\partial x}$ is small (which would cause vanishing gradients in a plain network), the identity term I ensures that the gradient is at least 1 in every direction. In a deep network with L residual blocks, the gradient from the loss to the input passes through L such factors, each of the form $I + \frac{\partial F_\ell}{\partial x}$. The product always contains a direct path (the product of all the I terms) that carries gradient without attenuation. This is the same “gradient highway” principle we saw in LSTMs — and it is the key reason why ResNets were the first architectures to successfully train networks with 100+ layers.

3 Training Neural Networks with Pytorch

Please refer to our [Google Colab tutorial](#) on training a ResNet on the MNIST dataset.