

Training Neural Networks From Gradient Descent to ResNets

1 Training a Neural Network

A neural network contains parameters, usually called weights and biases. Training means finding parameter values that make the network perform well on the training data.

We define a loss function

$$L(\theta)$$

where θ represents all of the network's parameters.

The basic idea is:

prediction $\rightarrow$ loss $\rightarrow$ gradient $\rightarrow$ update parameters.

The gradient tells us how the loss changes when we change the parameters.

For a parameter θ ,

$$\frac{\partial L}{\partial \theta}$$

tells us the direction in which the loss increases.

Therefore, to reduce the loss, we move in the opposite direction.

2 Parameter Initialization

Before training begins, the network's weights must be assigned initial values.

This is called *parameter initialization*.

A naive approach would be to initialize every weight to zero.

This creates a problem in a multilayer network. If neurons in the same layer all start with exactly the same weights, they receive the same inputs, produce the same outputs, and receive the same gradients. They therefore remain identical during training.

For this reason, weights are normally initialized with small random values.

However, 'small' does not simply mean 'as close to zero as possible.' The scale of the weights is important.

2.1 Why Weights Should Not Be Too Large

Consider a neuron using the sigmoid activation function:

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

Its derivative is

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

The derivative is largest around $z = 0$:

$$\sigma'(0) = 0.25.$$

But when z becomes very large or very negative, the sigmoid saturates:

$$\sigma(z) \approx 1 \quad \text{or} \quad \sigma(z) \approx 0.$$

In either case,

$$\sigma'(z) \approx 0.$$

For example,

$$\sigma(10) \approx 1$$

and

$$\sigma'(10) \approx 0.$$

If the weights are initialized too large, the weighted sums

$$z = w^T x + b$$

can become very large even before training begins.

The neurons can therefore start in a saturated region where the activation function has an almost-zero derivative.

Since backpropagation uses derivatives, this can make the gradient almost zero:

large initial weights $\rightarrow$ large activations $\rightarrow$ saturated neurons $\rightarrow$ almost-zero gradients

If the gradient is almost zero, gradient descent has very little information with which to change the weights.

2.2 Why Weights Should Not Be Too Small

The opposite extreme is also undesirable.

If all weights are extremely close to zero, the signals flowing through the network can become very small.

For some activation functions and network architectures, repeated multiplication by small values can cause gradients to shrink as they propagate backward.

Thus, we want an appropriate scale rather than simply the smallest possible weights.

The goal is approximately:

weights small enough to avoid saturation, but large enough to maintain useful signals and gradients.
--

2.3 Random Initialization

Weights are therefore commonly initialized using a random distribution whose scale depends on the number of inputs and outputs of a layer.

Two important initialization methods are Xavier initialization and He initialization.

For a layer with n_{in} input units and n_{out} output units, Xavier initialization commonly uses a variance related to

$\text{Var}(w) \approx \frac{2}{n_{\text{in}} + n_{\text{out}}}$
--

or, in a simplified version,

$$\text{Var}(w) \approx \frac{1}{n_{\text{in}}}.$$

For networks using ReLU activations, He initialization is commonly used:

$$\boxed{\text{Var}(w) \approx \frac{2}{n_{\text{in}}}}$$

The exact initialization used in practice depends on the activation function and network architecture.

The important idea is that initialization is not arbitrary.

It is chosen so that signals and gradients remain at a useful scale as they move through the network.

2.4 Initialization Before Training

The overall process therefore begins with

$$\boxed{\text{initialize parameters} \rightarrow \text{forward pass} \rightarrow \text{loss} \rightarrow \text{backpropagation} \rightarrow \text{update parameters}}$$

Good initialization gives gradient descent a reasonable starting point.

It does not solve the optimization problem by itself, but it can make the difference between a network that trains effectively and one in which gradients are extremely small or unstable from the beginning.

3 Gradient Descent

The basic gradient-descent update is

$$\boxed{\theta_t = \theta_{t-1} - \eta \nabla_{\theta} L_t}$$

where

- θ_t is the new parameter value;
- θ_{t-1} is the previous parameter value;
- η is the learning rate;
- $\nabla_{\theta} L_t$ is the gradient.

For example, suppose

$$\theta = 2, \quad \frac{\partial L}{\partial \theta} = 4, \quad \eta = 0.1.$$

Then

$$\theta_{\text{new}} = 2 - (0.1)(4) = 1.6.$$

The parameter moves in the direction that reduces the loss.

3.1 The Learning Rate Problem

The learning rate η controls the size of each step.

If η is too small, training can be extremely slow.

If η is too large, the optimizer can jump over the minimum.

For example:

$$\text{minimum} \leftarrow \text{overshoot} \rightarrow \text{overshoot}.$$

A fixed learning rate is therefore not always ideal.

4 Learning-Rate Schedules

A common improvement is to change the learning rate during training.

For example, we might start with

$$\eta = 0.1$$

and gradually reduce it:

$$0.1 \rightarrow 0.05 \rightarrow 0.025 \rightarrow \dots$$

Early in training, larger steps can help the optimizer move quickly toward a good region.

Later, smaller steps allow the optimizer to make more precise adjustments.

A simple decay rule is

$$\eta_t = \frac{\eta_0}{1 + kt}$$

where k controls how quickly the learning rate decreases.

Another common strategy is to reduce the learning rate at selected points during training.

4.1 Warmup

Sometimes a network is unstable if training begins immediately with a large learning rate.

A solution is learning-rate warmup.

The learning rate starts small and increases during the first few training steps:

$$\eta_0 < \eta_1 < \eta_2 < \dots$$

After warmup, a normal learning-rate schedule can be used.

Thus, learning-rate schedules address the problem that one fixed learning rate is not always appropriate.

5 Momentum

Ordinary gradient descent uses only the current gradient.

This creates a problem when the loss surface has a long, narrow valley. The gradient may repeatedly point mostly sideways relative to the direction toward the minimum.

The result can be a zig-zag path:

$$\text{left} \rightarrow \text{right} \rightarrow \text{left} \rightarrow \text{right}.$$

The optimizer eventually reaches the minimum, but it can take many steps.

This motivated methods that remember information from previous gradients.

Momentum treats optimization somewhat like a physical object that has velocity. We initialize the velocity to zero:

$$v_0 = 0$$

At step t , let

$$g_t = \nabla_{\theta} L_t$$

be the current gradient.

The velocity is updated using

$$v_t = \beta v_{t-1} + g_t$$

and the parameters are updated using

$$\theta_t = \theta_{t-1} - \eta v_t.$$

Here:

- v_t is the accumulated velocity;
- g_t is the current gradient;
- β controls how much previous gradients influence the current velocity;
- η is the learning rate.

A common value is

$$\beta = 0.9.$$

For example,

$$v_t = 0.9v_{t-1} + g_t.$$

If the gradients keep pointing in approximately the same direction, the velocity builds up in that direction.

If the gradient alternates,

$$+g, \quad -g, \quad +g, \quad -g,$$

the alternating components tend to cancel.

Therefore momentum:

1. accelerates movement in directions where gradients are consistently pointing;
2. reduces zig-zagging caused by rapidly changing gradient directions.

The important idea is:

Momentum gives the optimizer a memory of the direction in which it has recently been moving.

5.1 Momentum Conventions

There are several momentum conventions in the literature.

Another common form is

$$v_t = \beta v_{t-1} + (1 - \beta)g_t,$$

which treats v_t explicitly as an exponential moving average of the gradients.

That convention is also valid, but the subsequent update is normally written with a correspondingly different scaling.

For these notes, we use

$$v_t = \beta v_{t-1} + g_t, \quad \theta_t = \theta_{t-1} - \eta v_t$$

because it makes the role of the learning rate particularly clear.

6 AdaGrad

Momentum remembers the direction of previous gradients, but another problem remains.

Different parameters can have very different gradient magnitudes.

For example, one parameter might repeatedly receive gradients around

$$10$$

while another receives gradients around

$$0.01.$$

Using the same learning rate for both may not be appropriate.

AdaGrad addresses this by keeping track of the accumulated squared gradients.

Define

$$G_t = G_{t-1} + g_t^2$$

where the square is applied element-by-element.

The parameter update is

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t} + \epsilon} g_t$$

where ϵ is a small number that prevents division by zero.

Parameters that have repeatedly received large gradients therefore get smaller effective updates.

7 The Problem with AdaGrad

AdaGrad continually accumulates squared gradients:

$$G_t = g_1^2 + g_2^2 + \cdots + g_t^2.$$

Therefore G_t can become very large.

As training continues, the effective learning rate can become extremely small.

Training may eventually slow down too much.

This motivated RMSProp.

8 RMSProp

RMSProp does not remember every gradient forever.

Instead, it keeps an exponentially weighted average of recent squared gradients:

$$s_t = \beta s_{t-1} + (1 - \beta)g_t^2.$$

The update is

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t} + \epsilon} g_t$$

The important difference is that old gradients gradually lose their influence.

Thus RMSProp adapts the step size for each parameter without allowing the accumulated value to grow forever.

9 Adam

We now have two useful ideas:

1. momentum remembers the direction of previous gradients;
2. RMSProp adapts the step size based on recent gradient magnitudes.

Adam combines these ideas.

Adam maintains two quantities.

The first moment tracks the average gradient:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t.$$

The second moment tracks the average squared gradient:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2.$$

The initial values are

$$m_0 = 0, \quad v_0 = 0.$$

Because these quantities start at zero, Adam uses bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

and

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

The parameter update is

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typical values are

$$\beta_1 = 0.9, \quad \beta_2 = 0.999.$$

9.1 Why Adam Adapts Each Parameter

The denominator

$$\sqrt{\hat{v}_t} + \epsilon$$

is calculated separately for each parameter.

Suppose one parameter has recently had large gradients while another has had small gradients.

The first parameter gets divided by a larger number, reducing its effective step.

The second parameter gets divided by a smaller number, so its effective step remains larger.

Thus Adam automatically scales the update separately for each parameter.

This is different from ordinary gradient descent, where every parameter uses the same scalar learning rate η .

10 Overfitting

An optimizer can make the training loss extremely small while the network performs poorly on new data.

This is called overfitting.

The network has effectively learned details of the training examples rather than the general pattern.

We therefore need methods that constrain the model.

11 L2 Regularization

One simple method is to penalize large weights.

Instead of minimizing only

$$L,$$

we minimize

$$L_{\text{total}} = L + \lambda \sum_i w_i^2$$

where λ controls the strength of the penalty.

Large weights increase the loss, encouraging the network to use smaller weights.

12 L1 Regularization

Another possibility is

$$L_{\text{total}} = L + \lambda \sum_i |w_i|$$

L1 regularization can encourage some weights to become exactly zero.

This can produce a sparse model.

13 Dropout

Dropout addresses overfitting by randomly removing some neurons during training.

For example, if a layer contains 8 neurons, a training step might randomly disable several of them.

The network therefore cannot depend too heavily on any one neuron.

At evaluation time, dropout is disabled and the full network is used.

The intuition is that dropout encourages the network to learn redundant and robust representations.

14 Hyperparameters

A parameter is learned by the network.

A hyperparameter is selected by the person training the network.

Examples include:

- learning rate η ;
- momentum coefficient β ;
- Adam coefficients β_1, β_2 ;
- regularization strength λ ;
- dropout probability;
- number of layers;
- number of neurons per layer;
- batch size.

These values are not normally learned directly through gradient descent.

They are chosen experimentally or through an automated search.

15 A New Problem: Images

Fully connected neural networks treat each input feature somewhat independently.

Images contain important spatial structure.

For example, nearby pixels are usually more closely related than pixels on opposite sides of an image.

This motivates convolutional neural networks.

16 Convolutional Neural Networks

A convolutional layer applies small filters to local regions of an image.

For example, a filter might be

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

The filter moves across the image and produces a feature map.

Different filters can learn to detect different patterns:

- edges;
- corners;
- textures;
- shapes;
- eventually, more complex objects.

The same filter is reused at many spatial locations.

This dramatically reduces the number of parameters compared with a fully connected layer.

17 A New Problem: Sequences

Images have spatial structure.

Other data has temporal or sequential structure:

$$x_1, x_2, x_3, \dots, x_t.$$

Examples include:

- sentences;
- speech;
- stock prices;
- sensor readings;
- time-series data.

The meaning of the current input can depend on previous inputs.

This motivates recurrent neural networks.

18 Recurrent Neural Networks

An RNN maintains a hidden state.

At time t ,

$$h_t = f(W_x x_t + W_h h_{t-1} + b).$$

The current hidden state therefore depends on both

$$x_t$$

and

$$h_{t-1}.$$

Information can therefore flow through time.

However, this creates another problem.

19 The Vanishing-Gradient Problem

During backpropagation through many time steps, gradients are repeatedly multiplied by derivatives.

Suppose the relevant derivative is approximately

$$0.5.$$

After one step:

$$0.5.$$

After two:

$$0.5^2 = 0.25.$$

After ten:

$$0.5^{10} \approx 0.001.$$

After many steps, the gradient can become extremely small.

This is the vanishing-gradient problem.

The network has difficulty learning relationships between events that are far apart in time.

20 The Exploding-Gradient Problem

The opposite can also happen.

If repeated multiplication produces values larger than 1, the gradient can grow rapidly.

For example,

$$1.5^{10} \approx 57.7.$$

After many steps, the gradient can become enormous.

This is the exploding-gradient problem.

21 Gradient Clipping

A simple solution to exploding gradients is gradient clipping.

For example, if the gradient norm exceeds some threshold c , we can rescale it:

$$g \leftarrow g \frac{c}{\|g\|} \quad \text{if } \|g\| > c.$$

This prevents a single update from becoming excessively large.

Gradient clipping addresses exploding gradients, but it does not solve the fundamental problem of vanishing gradients.

22 LSTM

Long Short-Term Memory networks were designed to help neural networks remember information for long periods.

An LSTM introduces a memory cell and gates that control the flow of information.

The main gates are:

- forget gate;

- input gate;
- output gate.

The forget gate decides what information should be removed.

The input gate decides what new information should be stored.

The output gate decides what information should be exposed as the current hidden state.

The key idea is that information can travel through the cell state without being repeatedly transformed in the same way as a standard RNN hidden state.

This provides a much better path for gradients over long sequences.

23 A New Problem: Very Deep Networks

As neural networks became deeper, another problem appeared.

In principle, adding more layers should allow a network to represent more complicated functions.

In practice, simply adding layers can make optimization harder.

A deeper network can sometimes have higher training error than a shallower network.

This is not merely overfitting.

Even training the network itself becomes difficult.

This motivated normalization techniques and residual networks.

24 Normalization

Neural network activations can change substantially as information passes through many layers.

If the distribution of activations changes dramatically from layer to layer, optimization can become difficult.

Normalization layers rescale and shift activations into a more controlled range.

A simplified normalization operation is

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

followed by a learned scale and shift:

$$y = \gamma \hat{x} + \beta.$$

Here:

- μ is a mean;
- σ^2 is a variance;
- γ is a learned scale;
- β is a learned shift.

Different normalization methods use different definitions of the statistics.

For example, batch normalization computes statistics over a batch, while layer normalization computes statistics within an individual example's features.

24.1 Why Normalization Helps

Normalization helps keep activations in a more predictable range.

This can make the optimization landscape easier to navigate.

It can also make gradients more stable and allow the use of larger learning rates in many situations.

Normalization therefore became an important tool for training deeper networks.

25 Residual Networks

Even with normalization, very deep networks can be difficult to optimize.

Residual networks, or ResNets, introduced a simple but powerful idea.

Instead of asking a stack of layers to learn a complete function

$$H(x),$$

we let the layers learn a residual:

$$F(x) = H(x) - x.$$

The output becomes

$$\boxed{H(x) = F(x) + x}$$

The original input x is therefore added directly to the output.

This is called a skip connection.

26 Why Skip Connections Help

Consider a residual block:

$$x \rightarrow \text{layers} \rightarrow F(x)$$

with

$$y = F(x) + x.$$

During backpropagation,

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1.$$

The important part is the $+1$.

There is now a direct path through which the gradient can flow.

Without the skip connection, the gradient would have to pass entirely through the learned layers.

With the skip connection, part of the gradient can travel directly backward.

27 ResNets as Short Paths

A very deep residual network can be viewed as containing many paths of different lengths.

Some paths pass through many residual blocks.

Other paths effectively skip blocks through the identity connections.

Thus, although the network may contain hundreds of layers, there are many much shorter computational paths between the input and output.

A useful mental model is:

A ResNet behaves like an ensemble of paths of different depths.

Some paths are very long.

Some paths are much shorter.

The shorter paths make optimization easier because gradients do not always have to travel through the entire depth of the network.

28 ResNets and Initial Layers

The skip connections are particularly useful for gradients flowing back toward the initial layers.

Suppose a gradient reaches a residual block:

$$y = F(x) + x.$$

Then

$$\frac{\partial y}{\partial x} = \frac{\partial F}{\partial x} + 1.$$

The identity term provides a direct contribution to the gradient.

Across many residual blocks, the gradient therefore has paths that avoid repeated multiplication by complicated layer derivatives.

This helps prevent gradients from becoming extremely small.

In this sense, ResNets provide more stable routes through which gradient information can travel toward the early layers.

29 Gradient Shattering

There is another difficulty in very deep networks.

Even when gradients do not simply vanish, repeated nonlinear transformations can cause gradients to become highly irregular.

Small changes in the input or parameters can produce large changes in the gradient.

This phenomenon is sometimes called gradient shattering.

The problem is that the gradient becomes noisy and poorly structured, making optimization difficult.

Normalization and residual connections both help provide more stable optimization paths.

30 Normalization and ResNets Together

Normalization and residual connections solve related but different problems.

Normalization helps control the scale and distribution of activations and gradients.

Residual connections provide short paths for information and gradients.

Together they make it practical to train very deep networks.

A simplified view is:

Normalization $\rightarrow$ stable representations

and

Residual connections $\rightarrow$ short gradient paths

The combination is much easier to optimize than a very deep plain feed-forward network.

31 The Historical Progression

The development of modern neural-network training can be viewed as a sequence of problems and solutions.

1. Parameter initialization.

“ The network needs a useful starting point.

Solution: initialize weights with an appropriate small random scale so that activations do not immediately saturate and gradients remain useful.

2. Basic gradient descent.

Update parameters using the negative gradient.

$$\theta \leftarrow \theta - \eta \nabla_{\theta} L$$

3. Learning-rate problem.

A fixed learning rate is not always appropriate.

Solution: learning-rate schedules and warmup.

4. Zig-zagging.

Gradients can repeatedly change direction.

Solution: momentum.

5. Different gradient scales.

Different parameters may need different effective step sizes.

Solution: AdaGrad.

6. AdaGrad slows down too much.

Accumulated squared gradients continually grow.

Solution: RMSProp.

7. Adaptive scaling plus momentum.

We want both ideas.

Solution: Adam.

8. Overfitting.

The network may memorize the training data.

Solutions: L1, L2, and dropout.

9. Image structure.

Fully connected layers ignore useful spatial structure.

Solution: CNNs.

10. Sequential structure.

The current input may depend on previous inputs.

Solution: RNNs.

11. Vanishing and exploding gradients.

Long sequences create repeated gradient multiplication.

Solutions: gradient clipping and LSTM.

12. Very deep networks.

Deep plain networks can be difficult to optimize.

Solution: normalization.

13. Gradient instability and very long paths.

Gradients can become irregular or difficult to propagate through many layers.

Solution: residual connections and ResNets. ““

32 Key Formulas

Parameter Initialization

For Xavier initialization:

$$\text{Var}(w) \approx \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

For He initialization:

$$\text{Var}(w) \approx \frac{2}{n_{\text{in}}}$$

The purpose is to keep activations and gradients at a useful scale.

Gradient Descent

$$\theta_t = \theta_{t-1} - \eta \nabla_{\theta} L_t$$

Momentum

Initialize

$$v_0 = 0$$

then

$$v_t = \beta v_{t-1} + g_t$$

and

$$\theta_t = \theta_{t-1} - \eta v_t$$

where

$$g_t = \nabla_{\theta} L_t.$$

AdaGrad

$$G_t = G_{t-1} + g_t^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t} + \epsilon} g_t$$

RMSProp

$$s_t = \beta s_{t-1} + (1 - \beta) g_t^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t} + \epsilon} g_t$$

Adam

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

with

$$m_0 = 0, \quad v_0 = 0.$$

Bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

and update:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

L2 Regularization

$$L_{\text{total}} = L + \lambda \sum_i w_i^2$$

L1 Regularization

$$L_{\text{total}} = L + \lambda \sum_i |w_i|$$

Normalization

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

followed by

$$y = \gamma \hat{x} + \beta$$

Residual Connection

$$y = F(x) + x$$

with gradient

$$\frac{\partial y}{\partial x} = \frac{\partial F}{\partial x} + 1.$$

33 The Big Picture

The central idea behind all of these techniques is that training a neural network is an optimization problem.

We repeatedly perform

initialize → forward pass → loss → backpropagation → optimizer → new parameters

The history of neural-network optimization can largely be understood as finding better ways to make this process stable and efficient.

Parameter initialization provides a useful starting point.

Gradient descent provides the basic optimization mechanism.

Momentum gives the optimizer memory.

AdaGrad and RMSProp adapt the scale of updates.

Adam combines momentum and adaptive scaling.

Regularization reduces overfitting.

CNNs exploit spatial structure.

RNNs exploit sequential structure.

LSTMs provide better long-term information flow.

Normalization stabilizes deep networks.

Residual connections provide short paths for information and gradients.

Together, these ideas made it possible to train increasingly large and deep neural networks.