LayerCal

Deep Learning Model Parameter Calculator

Technical Guide & Project Documentation

Chanjoong Kim

me@chanjoongx.com

Version 1.1.0 | February 2026

Project Information	
Live Demo	layercal.com
Tech Stack	React, Vite, Tailwind CSS
Deployment	Cloudflare Pages

Table of Contents

1. Project Overview

What is LayerCal?

LayerCal is a browser-based tool designed to help ML engineers and researchers quickly estimate neural network complexity before training.

Key capabilities:

- Real-time calculation of parameters, memory usage, and FLOPs
- Auto-generated code for PyTorch, TensorFlow, and JAX
- Runs entirely in browser with no backend dependencies

Problem Statement

Before training deep learning models, practitioners need to understand resource requirements. Manual calculation of parameters for complex architectures is error-prone and time-consuming. Existing tools often require installation or rely on backend servers.

Target Users

- ML/AI researchers designing new architectures
- Students learning deep learning concepts
- Engineers estimating GPU memory requirements
- Teams planning training infrastructure

2. Core Features

Feature	Description
Parameter Counting	Accurate calculation for 14 layer types with configurable hyperparameters
Memory Estimation	Inference/Training modes, FP32/FP16/INT8 precision, optimizer overhead
FLOPs Calculation	Forward pass computational cost estimation
Code Generation	PyTorch, TensorFlow/Keras, JAX/Flax code output
Drag & Drop UI	Intuitive layer palette with instant parameter updates
Dark Mode	System preference detection and manual toggle
Multi-language	8 languages: EN, KO, JA, ZH, ES, FR, DE, PT
PNG Export	One-click screenshot for documentation

Key Differentiators

- **Zero Backend:** All computation happens client-side
- **Smart Code Generation:** Consecutive layer grouping, context-aware normalization
- **Framework Parity:** Idiomatic code for each framework
- **Responsive Design:** Works on desktop and mobile

3. Supported Layer Types

LayerCal supports 14 common neural network layer types organized by category:

Category	Layer	Configurable Parameters
Embedding	Embedding	vocab_size, embedding_dim
Linear	Linear (Dense)	input_dim, output_dim, use_bias
Convolution	Conv2D	in_channels, out_channels, kernel_size, use_bias
Recurrent	LSTM	input_size, hidden_size, num_layers, bidirectional
Recurrent	GRU	input_size, hidden_size, num_layers, bidirectional
Attention	Transformer	d_model, num_heads, d_ff
Attention	Multi-Head Attention	d_model, num_heads
Normalization	BatchNorm	num_features
Normalization	LayerNorm	normalized_shape
Pooling	MaxPool2D	kernel_size
Pooling	AvgPool2D	kernel_size
Regularization	Dropout	rate (0.0-1.0)
Activation	ReLU	(no parameters)
Activation	Softmax	(no parameters)

Note: Dropout, ReLU, and Softmax have no trainable parameters but are included for architecture completeness.

4. Calculation Formulas

Parameter Count Formulas

Layer	Formula	Notes
Embedding	$V \times E$	vocab_size × embedding_dim
Linear	$(I \times O) + O$	with bias
Conv2D	$(C_{in} \times C_{out} \times K \times K) + C_{out}$	with bias
LSTM	$4 \times (I \cdot H + H^2 + 2H) \times L \times dir$	4 gates, 2 biases per gate
GRU	$3 \times (I \cdot H + H^2 + 2H) \times L \times dir$	3 gates, 2 biases per gate
Transformer	$12 \times d^2 + 13d$	per block (d_ff = 4d)
BatchNorm	$2 \times F$	gamma + beta
LayerNorm	$2 \times N$	gamma + beta
Attention	$4 \times (d^2 + d)$	Q, K, V, O projections with bias

Legend: V = vocab size, E = embedding dim, I = input dim, O = output dim, C = channels, K = kernel size, H = hidden size, L = num layers, d = model dim, F = features, N = normalized shape, dir = direction (1 for unidirectional, 2 for bidirectional). Transformer calculates per block; stack multiple blocks by adding layers.

Memory Estimation

Inference Mode

`Memory = Parameters × Bytes per Parameter`

Precision	Bytes/Param	Use Case
FP32	4	Full precision training/inference
FP16	2	Mixed precision training, GPU inference
INT8	1	Quantized inference, edge deployment

Training Mode

Training requires additional memory for gradients and optimizer states. LayerCal uses the Adam multiplier (4×) for training estimation:

`Training Memory = Parameters × Bytes per Parameter × 4`

This accounts for: weights + gradients + first moment + second moment (Adam optimizer). For reference, other optimizers have different multipliers: SGD (2×), SGD + Momentum (3×).

FLOPs Estimation (Forward Pass, per sample)

Layer	FLOPs Formula
Linear	$2 \times I \times O$ (multiply-add)
Conv2D	$2 \times C_{in} \times C_{out} \times K^2 \times H_{out} \times W_{out}$
LSTM (per timestep)	$8 \times H^2 + 8 \times I \times H$ (per layer, per direction)
Transformer	$8 \times S \times d^2 + 2 \times S^2 \times d + 4 \times S \times d \times d_{ff}$
Attention	$8 \times S \times d^2 + 2 \times S^2 \times d$

Note: S = sequence length. Transformer and Attention use S = 512 by default; LSTM and GRU use S = 128. Transformer FLOPs include attention (QKV projections + scores + output projection) plus FFN (two linear layers). Attention FLOPs cover the attention mechanism only without FFN.

5. Code Generation

LayerCal generates idiomatic, runnable code for three major frameworks. The code generator includes smart features that produce cleaner output than simple concatenation.

Smart Features

Feature	Description	Example
Layer Grouping	Groups consecutive identical layers	3× Transformer → TransformerEncoder(num_layers=3)
Context-aware BatchNorm	Selects 1D/2D based on previous layer	After Conv2D → BatchNorm2d
Semantic Naming	Meaningful variable names	self.embed, self.fc1, self.conv
Framework Idioms	Follows each framework's conventions	TF: Sequential vs Functional API

PyTorch Output Example

```python
import torch
import torch.nn as nn

class GeneratedModel(nn.Module):
    """Generated by LayerCal - layercal.com"""

    def __init__(self):
        super().__init__()
        self.embed = nn.Embedding(50000, 512)
        self.transformer_layer = nn.TransformerEncoderLayer(
            d_model=512, nhead=8,
            dim_feedforward=2048, batch_first=True
        )
        self.transformer = nn.TransformerEncoder(
            self.transformer_layer, num_layers=6
        )
        self.fc = nn.Linear(512, 10)

    def forward(self, x):
        x = self.embed(x)
        x = self.transformer(x)
        x = self.fc(x[:, 0])
        return x
```

Framework Support Matrix

Layer	PyTorch	TensorFlow	JAX/Flax
Embedding	✓	✓	✓
Linear	✓	✓	✓

Conv2D	✓	✓	✓
LSTM	✓	✓ (partial)	○
GRU	✓	✓ (partial)	○
Transformer	✓	○	○
Attention	✓	✓	○
BatchNorm	✓	✓	○
LayerNorm	✓	✓	✓
Pooling	✓	✓	○
Dropout	✓	✓	✓
Activations	✓	✓	✓

✓ = Full support, ○ = Placeholder/manual implementation needed

6. Technical Architecture

Tech Stack

Layer	Technology	Purpose
UI Framework	React 18	Component-based UI with hooks
Build Tool	Vite	Fast HMR, optimized production builds
Styling	Tailwind CSS	Utility-first CSS framework
Components	shadcn/ui	Accessible, customizable primitives
Icons	Lucide React	Consistent icon set
State	React useState + localStorage	Client-side persistence
Hosting	Cloudflare Pages	Global CDN, automatic HTTPS

Project Structure

```
layercal/
├── src/
│   ├── components/
│   │   ├── LayerCal.jsx   # Main component
│   │   └── ui/            # shadcn components
│   ├── config/
│   │   ├── layerTypes.js  # Layer definitions, formulas
│   │   └── translations.js # i18n strings
│   ├── lib/
│   │   └── utils.js       # shadcn utility (cn)
│   ├── utils/
│   │   ├── codeGenerator.js
│   │   ├── imageExport.js
│   │   └── localStorage.js
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── public/
│   ├── calculator-icon.svg
│   └── favicon-*.png
├── docs/
│   └── LayerCal-Guide.pdf
├── index.html
├── vite.config.js
├── tailwind.config.js
├── postcss.config.js
└── package.json
```

7. Performance Optimizations

LayerCal implements several React optimization patterns to ensure smooth interaction even with many layers:

Technique	Implementation	Benefit
useMemo	LAYER_TYPES, totalParams, modelSizeMB	Prevents recalculation on every render
useCallback	All event handlers (addLayer, deleteLayer, etc.)	Stable function references
Functional setState	setModelLayers(prev => ...)	Avoids stale closure issues
Conditional Rendering	Language menu, modals	Reduces DOM nodes
Event Delegation	Single handler for layer palette	Fewer event listeners

Bundle Size: Production build with Vite produces optimized chunks. Tree shaking removes unused code from dependencies. Total bundle size is under 200KB gzipped.

8. Internationalization

LayerCal supports 8 languages: English, Korean, Japanese, Chinese (Simplified), Spanish, French, German, and Portuguese.

Translations are stored as nested objects in `translations.js`. Current language is persisted to localStorage. Layer descriptions, UI labels, and tooltips are all translated.

Code	Language
en	English
ko	한국어
ja	日本語
zh	中文
es	Español
fr	Français
de	Deutsch
pt	Português

LayerCal developed by Chanjoong Kim

Email: me@chanjoongx.com | GitHub: @chanjoongx