Episode Book

EPISODE #6694

Weights, Biases, and What Fine-Tuning Actually Moves

PUBLISHED	19 November 2026 at 22:33 UTC
DURATION	30:34
LISTEN	myweirdprompts.com/episode/weights-biases-neural-network-finetuning-71138/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	z-ai/glm-5.3
VOICES	omnivoice
PIPELINE	V5.36
SOURCES	16 cited by the research (listed at the end)
BOOK GENERATED	20 November 2026 at 03:35 UTC

SYNOPSIS

Every neuron in NVIDIA's 600-million-parameter Parakeet speech recogniser runs on simple school-algebra math: weights act as slopes on inputs, while biases act as resting offsets. Though biases make up a tiny fraction of a layer's parameters, they receive special treatment in PyTorch recipes and let a network respond to silence. From Frank Rosenblatt's motorized 1957 Mark I Perceptron to modern gradient descent, this breakdown traces how individual numbers scale up into a 2.47-gigabyte model and what fine-tuning actually updates.

TRANSCRIPT



Corn

Six hundred million numbers, and one of them is doing a job nobody ever mentions. That's where Daniel wants to go this time. He's been fine-tuning Parakeet, NVIDIA's speech recogniser, and he's realised he can talk about weights all day without knowing what a bias is, or why the whole thing borrows the word "neural" from brains it doesn't resemble. So he's asking us to go back to the floor: what a weight is, what a bias is, whether fine-tuning moves both, and how you get from one lonely number to a 2.47 gigabyte file on disk. One clear picture, bottom to top.



Herman

And the picture is a ladder, that's the good news. It's not six separate mysteries. It's one number, then a row of numbers, then a grid, then a named tensor, then a file, then a model. Every rung is the same stuff with one more dimension.



Corn

I want to test that claim before we climb it, because it sounds like a comfort. "It's just dimensions" — is that actually true all the way up, or does something qualitatively new happen when we get to the file?



Herman

Fair challenge, and the honest answer is that the numbers never change character. What changes is bookkeeping. A scalar, a vector, a matrix, a tensor — the arithmetic at the bottom is multiplication and addition, and at the top it's still multiplication and addition, just organised so a machine can do billions of them a second. The new thing — the only new thing — arrives at the naming step, when numbers get labels. And that's near the top of the ladder, not the bottom.



Corn

Then let's start on the bottom rung.



Herman

The thing let me flag before we climb anywhere is the asymmetry, because it's hiding in plain sight. Everyone says "weights". Nobody says "biases". But the moment you look at an actual training recipe, biases get special treatment — they're initialised to zero while weights get random values, and the standard optimiser setup explicitly turns off weight decay for them. PyTorch's own documentation shows that recipe. So the parameter nobody talks about is the parameter everybody handles with gloves.



Corn

Which either means they're precious or they're an afterthought.



Herman

My read is precious. Here's the one-line distinction, and the rest of the episode unpacks it: a weight says how much a particular input matters. A bias says how eager the neuron is to fire at all, regardless of input. A weight is a slope. A bias is an offset.



Corn

Give me that in something I can hold. Slope and offset — where do I meet those outside of neural networks?



Herman

Everywhere, honestly. The classic one is the straight line from school: y equals m-x plus b. The m is the slope — how much the output responds to the input. The b is the offset — where the line sits when the input is zero. Every neuron in every network, including all six hundred million parameters in Parakeet, is doing school-algebra slope-and-offset at its core. The fancy part is only that there are many inputs and then a squashing function at the end.



Corn

And every number in Parakeet — all six hundred million — is one of those two things, sitting in a FastConformer-TDT that eats sixteen kilohertz audio and spits out text. Six hundred million slopes and offsets, 2.47 gigabytes on disk.



Herman

So we climb. And we should climb honestly — starting with the smallest thing there is, one number, one connection, one neuron.



Corn

The arithmetic, then. Plainly.



Herman

Plainly. One neuron takes its inputs, multiplies each by its own weight — one weight per connection, that's the crucial part — sums the products, then adds one extra number that belongs to no input at all. That's the bias. Then it pushes all of it through a non-linearity, an activation function. Weighted sum, plus bias, through a squashing function. That's a neuron. That's all a neuron is.



Corn

Say the neuron has two inputs. What does the calculation actually look like on paper?



Herman

Two inputs, call them x-one and x-two. Two weights, w-one and w-two, one per input. One bias, b. The neuron computes w-one times x-one, plus w-two times x-two, plus b — and then squashes that total through the activation. If it helps, imagine a neuron deciding whether a sound is speech: one input might be "energy at this frequency", another "energy at that frequency", and the weights say which frequency the neuron trusts more. The bias is its general level of suspicion before it's heard anything.



Corn

I want to slow down on the bias, because that's Daniel's actual blind spot. Why does it exist? What breaks without it?



Herman

The threshold. A biological neuron doesn't fire until its membrane voltage crosses a certain level — the threshold potential. The bias is the artificial version of that, loosely. It moves the point at which the artificial neuron switches on, and it does that independently of what the inputs are doing. Without a bias, a neuron receiving all-zero inputs is stuck producing zero output no matter how you've set the weights. The bias is what lets it have an opinion at rest.



Corn

That's the most sloth thing a parameter can be.



Herman

It is! It's the parameter that works when nothing's happening. Weights live on the edges of the graph, on the connections. The bias lives on the node. That's the cleanest way to hold it — weights are per-connection, one bias per neuron, attached to nothing.



Corn

And when is that opinion at rest actually useful? Give me a case where the zero-input situation matters rather than being a corner case.



Herman

Silence is the obvious one for a speech model, funnily enough. A long stretch of quiet audio is close to a zero-ish input vector, and the network still has to decide things about it — that this is silence, that no token should be emitted. Neurons with no bias have no vocabulary for "nothing is happening, and here's what I think about that". The bias is that vocabulary.



Corn

Now the word itself. "Neural." Is a neuron a thing in there?



Herman

No, and that's worth being blunt about. An artificial neuron is a mathematical function. It's a small arithmetic recipe — the one I just described. The name goes back to 1943, McCulloch and Pitts, a paper called "A Logical Calculus of the Ideas Immanent in Nervous Activity". They were modelling biological neurons, inputs analogous to excitatory and inhibitory potentials at the dendrites, weights analogous to synaptic strengths, the output analogous to an action potential down the axon. Wikipedia actually calls the simple versions "caricature models" — intended to reflect neurophysiological observations, without regard to realism.



Corn

A caricature that got naming rights to the entire field.



Herman

And in a transformer, the caricature thins out completely. A "neuron" is really just a row of a matrix. The vocabulary is a keepsake from nineteen forty-three.



Corn

You own a DJ mixing desk, Herman. You know what a row of a matrix is when it's hardware.



Herman

Faders, honestly, that's closer than I'd like to admit. But if you want the real physical version — the machine where you could watch a weight move — it's Rosenblatt's Mark I Perceptron. Frank Rosenblatt simulated the perceptron on an IBM 704 in 1957, then built the hardware at Cornell Aeronautical Laboratory in Buffalo. The Mark I had four hundred photocells in a twenty-by-twenty grid, an input retina, feeding five hundred and twelve association units, feeding eight response units. And here's the part I love: the adjustable weights were encoded in potentiometers, and weight updates during learning were performed by electric motors. Physically turning the knobs.



Corn

A weight was literally a knob. Six hundred million knobs and the motors would still be turning.



Herman

First public demonstration was June 23rd, 1960. And the structure of that machine is the structure we still use — layers, a transformation at each layer, signals travelling from input through hidden layers to output. The one thing the Mark I generation got wrong-ish was the non-linearity's importance, so let me pin that down, because it answers why the networks are deep at all.



Corn

Go.



Herman

If you stack linear layers with no activation function, the whole stack collapses. A linear map composed with a linear map is a linear map. Ten layers, a thousand layers, doesn't matter — you could replace the entire tower with one matrix multiplication. The non-linearity is what makes depth mean anything. That's also why the multilayer perceptron era mattered — once you had non-linear activations stacked, you could separate data that wasn't linearly separable, which is the whole reason anyone cared.



Corn

And for anyone whose eyes glaze at "linearly separable" — the classroom version?



Herman

The classic is two clusters of points on a sheet of paper, and asking whether you can draw one straight line that separates them. Dots and crosses interleaved like the stripes on a chessboard's diagonals — no single line does it. But fold the paper once, and a flat cut through the fold does it. Stacking non-linear layers is the folding; each crease is a layer, and depth is how many folds you get to make.



Corn

Okay. Now the count, because I know you've been saving it.



Herman

I have, and it reframes everything after it. Take a plain linear layer, twenty inputs, forty outputs. The weight is a grid of forty by twenty — eight hundred numbers. The bias is forty numbers, one per output neuron. So the biases are four point eight percent of that layer's parameters. And in a transformer, where the weight matrices are enormous, the fraction shrinks further still. Biases are a rounding error in the count.



Corn

Eight hundred slopes, forty opinions.



Herman

That ratio is the whole answer to Daniel's fine-tuning question, so let's take it head-on.



Corn

He asked it directly: when I fine-tune, does it change both? My guess before you answer is that the bias is so outnumbered it just gets dragged along.



Herman

Your guess is right for full fine-tuning, and I'd phrase it "updated alongside" rather than dragged. The bias is a parameter like any other. Gradient descent doesn't care what a number is called — it moves every parameter in the direction that reduces the loss. The LoRA paper states the baseline in one line: the model is initialised to the pre-trained weights and updated by repeatedly following the gradient. The parameter set includes biases. Nothing excludes them.



Corn

Let me push on the mechanics of that for a second, because "gradient descent moves everything" is easy to say. How does the bias know which way to move? It's not attached to any input.



Herman

Right, and that's the neat part. Every parameter gets the same treatment mathematically: you ask, if I nudge this number up a hair, does the loss get better or worse, and by how much. That question is answerable for a bias exactly as it is for a weight. The bias feeds into the neuron's output, the output feeds into the loss, so the chain of cause and effect exists — it just doesn't run through any input. The gradient for a weight involves the input value on that connection. The gradient for a bias is the same calculation with the input factor replaced by one. It's the weight of an invisible, always-on input.



Corn

An input that's always one. So the bias is a weight on a connection from nothing.



Herman

That's a legitimate way to see it, and some textbooks draw it exactly that way — a phantom input fixed at one. But I'd keep the two concepts separate in your head, because the special treatment we opened with only makes sense if you remember the bias is structurally different: it's not scaling anything.



Corn

And Parakeet itself is a fine-tune.



Herman

A full one, in stages, per the model card. Started from a FastConformer checkpoint pretrained with a wav2vec method on LibriLight, then trained for a hundred and fifty thousand steps on sixty-four A100s, then a stage-two fine-tune of twenty-five hundred steps on four A100s over roughly five hundred hours of human-transcribed audio. That's the "both moved" case. Every weight and every bias in there shifted from where pretraining left it.



Corn

So Daniel's fine-tune, the full one, moved all six hundred million. But then the asymmetry you opened with — zero-initialised, no weight decay — that treatment exists even though biases are, by the count, nothing.



Herman

And that's the interesting tension. Weight decay pulls parameters toward zero so the model doesn't overfit to quirks of the training data. The standard recipe spares biases from that. The reasoning you usually hear is that a bias doesn't scale an input — it shifts a threshold — so shrinking it doesn't regularise anything, it just handicaps the neuron's operating point.



Corn

Handicaps it how? If the bias drifts toward zero over training, what concretely goes wrong?



Herman

The neuron's threshold drifts with it. Every unit in the network has calibrated where its firing point sits relative to what the layers above expect — decay drags that calibration around for no reason connected to fitting the data. It's like tuning a whole orchestra by tightening every string a little each rehearsal regardless of whether it's flat. The strings that were in tune get worse, and the ones that weren't don't get better in any targeted way.



Corn

Four point eight percent of the numbers getting one hundred percent of the special handling.



Herman

Now the turn, because the plain answer to Daniel has a second half. LoRA — and Daniel has touched this, anyone fine-tuning on modest hardware touches it — flips the picture. Low-Rank Adaptation freezes the pre-trained weights entirely and injects small trainable matrices into each layer. The frozen weights don't move at all. Only the injected patch trains.



Corn

How small?



Herman

Against full fine-tuning of GPT-3 at 175 billion parameters with Adam, LoRA needs about ten thousand times fewer trainable parameters and a third of the GPU memory. And the striking result: a rank of one or two can suffice for the adaptation, even where the full rank of the weight matrix is twelve thousand, two hundred and eighty-eight. The bet, stated in the paper, is that the change in weights during adaptation has low intrinsic rank — the delta is simpler than the matrix it patches.



Corn

Hold on. The whole matrix has rank twelve thousand, but the change you need has rank two?



Herman

That's the hypothesis, and it keeps working in practice, with caveats worth airing. Later work — GraLoRA at NeurIPS, LoFT more recently — shows LoRA can underperform full fine-tuning and converge more slowly, and that the gap isn't just about rank. It's about optimizer dynamics. LoFT's finding is that Adam's momentum and variance need to live in the same low-rank subspace as the update, or you're not really mimicking full fine-tuning. GraLoRA found LoRA works best at ranks thirty-two to sixty-four and degrades above that — a structural bottleneck, gradients getting tangled across unrelated input channels.



Corn

So the simple story — "just train a small patch" — is simple in the way that gets you ninety percent of the way.



Herman

It gets you most of the way most of the time, and the reason people tolerate the gap is the deployment property, which is where this connects back to files. Because LoRA is linear, you can merge the trained matrices back into the frozen weights when you deploy. Fold the patch into the big matrix. No inference latency, by construction. The adapter is a small file that dissolves into the big one.



Corn

Which is the bridge, isn't it. At the file level a weight and a bias are just… separate named tensors, and whether they moved is a question about which names you allowed to move.



Herman

That's the ladder, so let's climb it properly, because this is the "one clear picture" Daniel asked for. Level zero: a scalar. One number, no dimensions. A single weight. The safetensors spec explicitly allows shape-empty tensors — zero-rank, quote, "merely a scalar".



Corn

That's charming. The format has to reassure you that one number counts.



Herman

Level one: a vector. A row of numbers. A single bias is a scalar; the biases of a whole layer form a vector with one entry per neuron. A frame of audio coming into Parakeet is a vector. Level two: a matrix — a grid, rows and columns, the weight matrix of a layer, forty rows by twenty columns in our earlier example. Level three and up: tensor, which is just the general name for the n-dimensional case. A three-dimensional tensor like a batch of one, sixteen heads, two hundred fifty-six dimensions. Nothing new happens at level three — it's the same numbers with more indices.



Corn

And the reason the higher dimensions exist at all — is that just convenience? Batching, heads, and so on?



Herman

Largely, yes. The extra dimensions are a way to keep related grids together under one name so the hardware can chew through them in one go. A batch of thirty-two audio frames isn't conceptually different from thirty-two separate vectors — but one three-dimensional tensor lets the GPU process all of them without the framework juggling thirty-two separate lookups. Dimensions past two are an efficiency of naming, not a new kind of number.



Corn

Level four. Where the names come from.



Herman

The state dict. A checkpoint is a dictionary of named tensors, and PyTorch's own pruning example shows it plainly: the keys come back as bias, weight_orig, weight_mask. In a real model it's a few thousand entries, every layer contributing its own dot-weight and dot-bias. That naming is exactly why you can freeze one and train the other — they're separate tensors with separate names, and the optimiser can be handed one list and not the other.



Corn

I did enjoy that the pruning example keeps the mask in the same dictionary. Even the surgery is just more named tensors.



Herman

Level five: the file. And this is the punchline of the whole climb, because a safetensors file is almost insultingly simple. Eight bytes at the front — an unsigned little-endian sixty-four-bit integer giving the size of the header. Then that many bytes of JSON. Then the rest of the file is one flat byte buffer. The header is a dictionary mapping each tensor name to its dtype, its shape, and its byte range — offsets relative to the start of the buffer. Row-major, little-endian.



Corn

So you're telling me the "model" I downloaded is a JSON index and a blob.



Herman

That's what I'm telling you, and here's the clarifying part: everything people call the model — the architecture, the layers, the attention — is not in the file. The structure lives entirely in the shape fields in that header, and in the code that knows how to wire the named tensors into layers. The bytes themselves are just a sequence. That's why you can stare at a 2.47 gigabyte download and not find the model in it. The model is the file plus the code.



Corn

And the proof of that, if anyone doubts it — hand the file to someone with no framework installed. What can they do with it?



Herman

They can read the JSON header with any text tools and tell you every tensor's name, shape, and dtype. They can list the model's skeleton, essentially. But they cannot run it, cannot transcribe a single word, because the wiring — which output of which matrix feeds which input of which next layer — exists only in the code. The file is a parts catalogue, fully labelled, with no assembly instructions.



Corn

Daniel's Parakeet file, though — that's a .nemo, not a bare safetensors.



Herman

Correct, and worth being honest about what we can and can't say. The .nemo container is a NeMo archive, and the card doesn't state the dtype split. But we can do the arithmetic live: six hundred million parameters at two bytes each — fp16 or bf16 — is about one point two gigabytes. The shipped file is 2.47 gig. That's roughly four bytes per parameter, which is consistent with a thirty-two-bit checkpoint, or with optimizer and extra state bundled into the archive. The card doesn't say which, and I'm not going to pretend it does.



Corn

Two bytes for a number that took sixty-four A100s to set. Memory is cheap; knowing is not.



Herman

One more note on the format itself, because it explains why it exists at all. The stated rationale is safety — removing the need for pickle, which on PyTorch meant that a randomly downloaded file could run arbitrary code when you loaded it. safetensors can't. The library is around four hundred lines of code versus two hundred ten thousand for HDF5. And the README is candid that "zero-copy" is aspirational — no format is really zero-copy in machine learning, and on GPU a copy is always required.



Corn

A file format that admits its own marketing is soft. Rare in this field.



Herman

In fairness to it, eight bytes, a JSON header, a blob — there's not much marketing to hide behind.



Hilbert

Hilbert's Anteater Trivia. Xenarthrans first appeared in South America more than fifty million years ago. I have been at this show a while, but not that while.



Corn

Fifty million years and they still haven't shipped a successor format.



Herman

Where were we — the buffer. Row-major, so the flattening rule is: last index varies fastest. The header tells you the shape; the layout tells you the order; the code tells you the meaning. That's the entire division of labour in a checkpoint.



Corn

Say the flattening rule in terms of counting, because "last index varies fastest" is compact to the point of opaque.



Herman

Think of a car odometer. The rightmost digit spins fastest, and only when it rolls over does the digit to its left move. Row-major flattening is an odometer where the indices are the digits: you walk the last index through its whole range, then tick the second-to-last by one, and so on. To find any single number in the blob, you compute its odometer reading — one flat offset — and jump straight there. That's how the header's byte ranges let you slice a tensor out of raw bytes.



Corn

It's a strange feeling, honestly. You spend years hearing "the model learned", and the artefact is a JSON file and a very long list of numbers.



Herman

I find it clarifying rather than deflating. Every mystical word — attention, encoder, TDT decoder — bottoms out in tensors with names like that. The decoder isn't in the file as a decoder. It's in the file as several dozen named grids of floats, and the word "decoder" lives in the config and the code.



Corn

Daniel's fine-tuning questions, then — the ones that led here. He was changing numbers inside that blob and watching his transcriptions change, without ever seeing the numbers.



Herman

And now he has the picture. One number is a weight or a bias. A row is a layer's biases. A grid is a layer's weights. Named grids in a dictionary are the checkpoint. The dictionary plus a header and a blob is the file. The file plus the wiring code is the model. Fine-tuning is moving some or all of those numbers, and which ones you're allowed to move is just... which names you hand the optimiser.



Hilbert

The bias isn't a number. It's a setting. You've had that backwards for twenty minutes.



Herman

The arithmetic holds, Hilbert. A bias is a stored value that shifts the threshold.



Hilbert

A stored value is a setting, Herman. I've got a potentiometer on the desk that settles this. Salvaged it from a piece of equipment I'd rather not name. Bakelite, hand-scratched scale up the side. Been turning it back and forth through most of this episode, if you were



Hilbert (continued)

wondering about the clicking.



Corn

I assumed it was you. The clicking, I mean.



Hilbert

The scratched scale is the part. Whoever marked it put the zero partway along, not at the end. Offset from the stop. So the knob doesn't start at zero, it starts wherever the zero mark sits, and everything after that is measured from there. That's the bias you keep describing as a number. It's where you start turning from. The weight is how far you turn.



Herman

The zero mark sitting off the stop — that's the same structure as a threshold shift. An offset baked into the scale itself.



Hilbert

Of course it is. Corn would have got there faster if he'd ever held one. The detent at the zero mark on this one is good, by the way. Most of them aren't. You can feel the zero without looking, which is the entire point of a detent, and most manufacturers have forgotten it.



Corn

What's it connected to?



Hilbert

Nothing. It's connected to nothing. I don't keep it wired, I keep it for the detent.



Herman

So the physical version of the bias is a knob attached to no circuit, kept for how it feels.



Hilbert

The feel is the specification. And before you ask, no, I'm not lending it to the show. I did a stint calibrating audio gear once and I learned what happens to pots you lend out.



Corn

One honest use for your offset, though — that zero-mark placement. You could set the knob's resting position to sit exactly on the zero mark, and then the detent would hold your baseline for you. Start from the bias every time, no look.



Hilbert

That's what the detent is for. You'd get more out of equipment if you read the scratches.



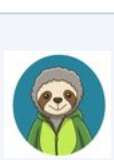
Corn

That's where I'll leave the ladder, I think. One number, one row, one grid, one named tensor, one JSON header, one flat buffer, one model that transcribes speech.



Herman

And if you take one thing from this episode, take the ratio. Eight hundred weights, forty biases. The bias is the rarest parameter in the network and the most protected — zero-initialised, spared from decay. Whatever it's doing, the people who build these things decided it was worth treating differently.



Corn

My addition: the model is not in the file. The file is numbers and shapes. The model is those numbers plus the code that knows what they mean — which means "downloading the model" was always a slight of hand.



Herman

A couple of open questions, honestly. If biases are four point eight percent of a layer's parameters, why does every recipe shield them — nobody in the literature says it directly, it's all convention inherited from convention.



Corn

And the Parakeet arithmetic. Four bytes per parameter when two would do — thirty-two-bit checkpoint, or optimizer state riding along in the.nemo. The card doesn't say, and I'd like to know which. My money's on bundled state, since Herman's on record saying a NeMo checkpoint can triple in size from dequantization alone and file size tells you nothing.



Herman

That's one bet I still hold. Big thanks as always to our producer, Hilbert Flumingtop, at the desk, as ever, clicking quietly. If this was your kind of episode, go back for episode eleven, How Does Fine Tuning Work Anyway; episode three, Safetensors or something else; and episode twelve, The AI Breakthrough. This has been My Weird Prompts — no question too obscure, no rabbit hole too deep. If you've enjoyed this one, the whole archive, all six thousand five hundred and sixty-eight episodes of it, lives at my weird prompts dot com.



Corn

We'll be back soon. Until then, remember that everything you've ever downloaded was mostly a blob.



Herman

See you tomorrow.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **safetensors README, huggingface/safetensors, retrieved 2026-11-19. [primary]**
https://github.com/huggingface/safetensors

2. **Hu et al., LoRA: Low-Rank Adaptation of Large Language Models, submitted 2021-06-21, retrieved 2026-11-19. [primary]**
https://arxiv.org/abs/2106.09685

3. **NVIDIA, Parakeet TDT 0.6B v2 model card, release date 05/01/2025, last modified 2026-06-29, retrieved 2026-11-19. [primary]**
https://huggingface.co/nvidia/parakeet-tdt-0.6b-v2

4. **Hugging Face file listing, 2.47 GB, retrieved 2026-11-19. [primary]**
https://huggingface.co/nvidia/parakeet-tdt-0.6b-v2/tree/main

5. **PyTorch docs, torch.optim, v2.14, retrieved 2026-11-19. [primary]**
https://docs.pytorch.org/docs/2.14/optim.html

6. **PyTorch docs, v2.14, retrieved 2026-11-19. [primary]**
https://docs.pytorch.org/docs/2.14/generated/torch.nn.utils.prune.l1_unstructured.html

7. **PyTorch docs, v2.14, retrieved 2026-11-19. [primary]**
https://docs.pytorch.org/docs/2.14/generated/torch.nn.utils.weight_norm.html

8. **PyTorch docs, module initialisation notes, v2.14, retrieved 2026-11-19. [primary]**
https://docs.pytorch.org/docs/2.14/notes/modules.html

9. **Wikipedia, Artificial neuron, last edited 2026-11-15, retrieved 2026-11-19.**
https://en.wikipedia.org/wiki/Artificial_neuron

10. **(machine_learning)**
Wikipedia, Neural network (machine learning), last edited 2026-11-18, retrieved 2026-11-19.
https://en.wikipedia.org/wiki/Neural_network

11. **Wikipedia, Multilayer perceptron, last edited 2026-11-16, retrieved 2026-11-19.**
https://en.wikipedia.org/wiki/Multilayer_perceptron

12. **Wikipedia, Perceptron, last edited 2026-11-18, retrieved 2026-11-19.**
https://en.wikipedia.org/wiki/Perceptron

13. **Tastan et al., LoFT: Low-Rank Adaptation That Behaves Like Full Fine-Tuning, submitted 2025-05-27, revised 2026-03-09, ICLR 2026, retrieved 2026-11-19.**
https://arxiv.org/abs/2505.21289v2

14. **Jung et al., GraLoRA: Granular Low-Rank Adaptation, NeurIPS 2025, revised 2026-01-15, retrieved 2026-11-19.**
https://arxiv.org/abs/2505.20355v2

15. **Liang & Bharadwaj, QR-LoRA, submitted 2025-08-29, retrieved 2026-11-19.**
https://arxiv.org/abs/2508.21810v1

16. **Hacker News comment by mike_hearn on Llama from scratch, 2023-08-09, retrieved 2026-11-19.**
https://news.ycombinator.com/item?id=37062331