Original Code Line	Optimized Code Line	Optimization Explanation
`R = zeros(eltype(x[1]), nd1, nd1)`	`R = similar(zeros(eltype(x[1]), nd, nd))`	Using `similar` instead of `zeros` reduces redundant computations and memory allocation, providing better performance.
Nested loops for filling in the `R` matrix	Loops replaced with vectorized operations	Vectorized operations take advantage of CPU parallelism and are generally more efficient than nested loops.
Manual incrementing of loop indices	Use of loop iterator variables	Eliminates the need for manual incrementing, reducing potential errors and making the code more concise.
`exp(-theta * abs(x[i] - x[j])^p)` in nested loops	`exp.(-theta .* (x[i] .- x[j]))`	Vectorized operation using broadcasting (`.`) and element-wise exponentiation (`exp.`) for improved performance and readability.
Insertion of elements into arrays using `insert!`	Direct assignment of values to arrays	Direct assignment is more efficient than using `insert!`, especially for adding elements at the end of the array.
Redundant variable initialization and computation	Streamlined calculations	Eliminating redundant computations and initializations reduces unnecessary overhead, resulting in faster execution.
Use of nested loops for calculating prediction	Single loop with vectorized operations	Combining operations into a single loop with vectorized calculations simplifies the code and improves performance by avoiding unnecessary iterations.
`norm(val[l] - k.x[i][l])^k.p[l]` inside loops	Pre-computation of `norm(val[j] - k.x[i][j])^k.p[j]`	Pre-computing and storing the value outside the loop eliminates redundant calculations and improves code readability.
`exp(-(abs(z))^k.p)`	`phi = z -> exp(-abs(z)^k.p)`	Using a function definition (`phi`) instead of inline

		calculation improves code readability and promotes code reuse.
`throw(DimensionMismatch("Input dimension..."))`	`throw(DimensionMismatch(...))`	Simplification of the error throwing line without changing its functionality.