hw10.pdf
PDF 495.98KB

Complete parts (a) and (b) from Problem 1: Kernelized Linear Attention (Part II).

Thought for 239 seconds

> We are asked to complete parts (a) and (b) from Problem 1: Kernelized Linear Attention (Part II).
>
> Part (a) has two subparts: i and ii.
>
> Part (a)i: Rewrite the softmax similarity function using Gaussian Kernel.
>
> We are given that the softmax similarity function is:
> sim_softmax(q, k) = exp(q^T k / sqrt(D)).

I like how DeepSeek's chain of reasoning always states the givens of the problem, as well as what it's destination is. Using all provided information to it's disposal is key in solving these types of math problems

> We are to use the Gaussian Kernel: K_Gauss(q, k) = exp(-||q-k||_2^2 / (2σ^2)).
>
> We are to rewrite the softmax similarity function in terms of the Gaussian Kernel, assuming σ^2 = sqrt(D).
>
> Hint: We can write the softmax as the product of the Gaussian Kernel and two other terms.
>
> Let's expand the Gaussian Kernel:
>
> K_Gauss(q, k) = exp(-||q-k||^2 / (2σ^2)) = exp(-(||q||^2 + ||k||^2 - 2q^T k) / (2σ^2)) = exp(-(||q||^2+||k||^2)/(2σ^2)) * exp(q^T k / σ^2).

Good choice to expand, now it figures out that it can substitute in K_Gauss(q, k)

> So, we have:
> exp(q^T k / σ^2) = K_Gauss(q, k) * exp((||q||^2+||k||^2)/(2σ^2)).
>
> But our softmax similarity is exp(q^T k / sqrt(D)). We are given σ^2 = sqrt(D). So then:
>
> exp(q^T k / sqrt(D)) = exp(q^T k / σ^2) because σ^2 = sqrt(D).
>
> Therefore:
>
> exp(q^T k / sqrt(D)) = K_Gauss(q, k) * exp((||q||^2+||k||^2)/(2σ^2)) = K_Gauss(q, k) * exp((||q||^2+||k||^2)/(2 sqrt(D))).
>
> So the softmax similarity function can be written as:
>
> sim_softmax(q, k) = exp(q^T k / sqrt(D)) = K_Gauss(q, k) * exp((||q||^2+||k||^2)/(2 sqrt(D))).
>
> Alternatively, we can write:

sim_softmax(q, k) = exp(q^T k / sqrt(D)) = exp(-||q-k||^2/(2 sqrt(D))) * exp((||q||^2+||k||^2)/(2 sqrt(D))).

But note that exp((||q||^2+||k||^2)/(2 sqrt(D))) = exp(||q||^2/(2 sqrt(D))) * exp(||k||^2/(2 sqrt(D))).

DeepSeek also finds an alternate path towards the solutiion

So we can write:

sim_softmax(q, k) = [exp(||q||^2/(2 sqrt(D)))] * [exp(-||q-k||^2/(2 sqrt(D)))] * [exp(||k||^2/(2 sqrt(D)))].

But the middle term is the Gaussian kernel with σ^2 = sqrt(D). So:

sim_softmax(q, k) = exp(||q||^2/(2 sqrt(D))) * K_Gauss(q, k) * exp(||k||^2/(2 sqrt(D))).

This is the required rewriting.

Part (a)ii: Use ϕ_random to approximate the above softmax similarity function with Gaussian Kernel and derive the computation cost for computing all the V'.

We are given the random feature map:

ϕ_random(q) = sqrt(1/D_random) [sin(w1 q), ..., sin(w_{D_random} q), cos(w1 q), ..., cos(w_{D_random} q)]^T.

Where w_i are sampled from N(0, σ^2 I_D), and we have:

E[ϕ_random(q) · ϕ_random(k)] = exp(-||q-k||^2/(2σ^2)).

But note that we have the softmax similarity function which is exp(q^T k / sqrt(D)) = exp(-||q-k||^2/(2 sqrt(D))) * exp((||q||^2+||k||^2)/(2 sqrt(D))).

We can approximate the Gaussian kernel part (exp(-||q-k||^2/(2 sqrt(D)))) by the inner product of the random features. However, note that the random feature map is designed to approximate the Gaussian kernel, not the entire softmax similarity. So we can write:

exp(q^T k / sqrt(D)) = exp(||q||^2/(2 sqrt(D))) * exp(-||q-k||^2/(2 sqrt(D))) * exp(||k||^2/(2 sqrt(D))).

Let K_Gauss(q,k) = exp(-||q-k||^2/(2 sqrt(D))). Then we have:

K_Gauss(q,k) ≈ ϕ_random(q) · ϕ_random(k) in expectation.

So then:

exp(q^T k / sqrt(D)) ≈ exp(||q||^2/(2 sqrt(D))) * (ϕ_random(q) · ϕ_random(k)) * exp(||k||^2/(2 sqrt(D))).

But note that the terms exp(||q||^2/(2 sqrt(D))) and exp(||k||^2/(2 sqrt(D))) are scalar functions that depend only on q and k respectively. We can absorb them into the feature map. Alternatively, we can define a new feature map that includes these terms.

However, the problem says: "Use ϕ_random to approximate the above softmax similarity function with Gaussian Kernel". So we are to use the random feature map to approximate the Gaussian kernel part. But then we have two additional terms. Alternatively, we can define a modified random feature map that directly approximates the softmax similarity. But note that the random feature map is for the Gaussian kernel, and the softmax similarity is not a shift-invariant kernel because of the additional terms. So we have to be careful.

Alternatively, we can note that the softmax similarity can be written as:

sim_softmax(q, k) = exp(q^T k / sqrt(D)) = exp(-||q||^2/(2 sqrt(D))) * exp(-||k||^2/(2 sqrt(D))) * exp(||q+k||^2/(2 sqrt(D))).

Wait, that might not help.

Alternatively, we can use the fact that the Gaussian kernel with a particular σ is being approximated. We are given that we set σ^2 = sqrt(D). Then the Gaussian kernel is exp(-||q-k||^2/(2 sqrt(D))). And we have:

sim_softmax(q, k) = exp(||q||^2/(2 sqrt(D))) * exp(-||q-k||^2/(2 sqrt(D))) * exp(||k||^2/(2 sqrt(D))).

So if we approximate the Gaussian kernel by ϕ_random(q)·ϕ_random(k), then we get:

sim_softmax(q, k) ≈ exp(||q||^2/(2 sqrt(D))) * (ϕ_random(q)·ϕ_random(k)) * exp(||k||^2/(2 sqrt(D))).

This can be written as:

sim_softmax(q, k) ≈ [exp(||q||^2/(2 sqrt(D))) ϕ_random(q)] · [exp(||k||^2/(2 sqrt(D))) ϕ_random(k)].

Because the dot product is bilinear. So if we define a new feature map:

ψ(q) = exp(||q||^2/(2 sqrt(D))) ϕ_random(q),

then we have:

sim_softmax(q, k) ≈ ψ(q) · ψ(k).

So we can use ψ as the feature map for the softmax similarity. But note that ψ(q) is a vector of dimension 2∗D_random. And we have:

ψ(q) = exp(||q||^2/(2 sqrt(D))) * sqrt(1/D_random) [sin(w1 q), ..., sin(w_{D_random} q), cos(w1 q), ..., cos(w_{D_random} q)]^T.

Then the attention computation becomes:

V'*i = (ψ(Q_i) · Σ*{j=1}^N ψ(K_j) V_j^T) / (ψ(Q_i) · Σ_{j=1}^N ψ(K_j)).

But note: In equation (2), we had:

V'*i = (ϕ(Q_i) Σ*{j=1}^N ϕ(K_j)^T V_j) / (ϕ(Q_i) Σ_{j=1}^N ϕ(K_j)^T).

But in our case, the feature map ψ is for the similarity function. So if we let ψ be the feature map, then we can write:

V'*i = (ψ(Q_i) · (Σ*{j=1}^N ψ(K_j) V_j^T)) / (ψ(Q_i) · (Σ_{j=1}^N ψ(K_j))).

However, note that V_j is a row vector of dimension M. So we are doing:

Numerator: ψ(Q_i) · (Σ_{j=1}^N ψ(K_j) V_j) [but note: ψ(K_j) is a column vector of dimension 2D_random, and V_j is a row vector of dimension M, so ψ(K_j) V_j is a matrix of size (2D_random x M). Then Σ_{j=1}^N ψ(K_j) V_j is a matrix of size (2D_random x M). Then ψ(Q_i) (a row vector of dimension 2D_random) times that matrix gives a row vector of dimension M.

Denominator: ψ(Q_i) · (Σ_{j=1}^N ψ(K_j)) is a scalar.

So we can compute:

S = Σ_{j=1}^N ψ(K_j) V_j (a matrix of size (2D_random x M))
Z = Σ_{j=1}^N ψ(K_j) (a vector of size (2D_random))

Then for each i, we compute:

V'_i = (ψ(Q_i) S) / (ψ(Q_i) Z).

Now, the computational cost:

We have to compute ψ for each query and key. For each vector (q or k), we compute:

- ||q||^2: O(D)
- exp(||q||^2/(2 sqrt(D))): O(1)
- Then compute the random features: we have D_random random vectors w_i, each of dimension D. Computing w_i q for one w_i: O(D). So for D_random of them: O(D_random * D). Then we take sin and cos, so we get 2D_random features. Then we multiply by the scalar and by sqrt(1/D_random). So per vector, the cost is O(D_random * D).

We have N queries and N keys, so total cost for computing all ψ: O(2N ∗ D_random * D).

Then we compute S and Z: S = Σ ψ(K_j) V_j. For each j, ψ(K_j) is a vector of size 2D_random, and V_j is a vector of size M. So the outer product (or more precisely, ψ(K_j) V_j) is a matrix of size (2D_random x M). Computing that for one j: O(2D_random * M). Then summing over N: O(N ∗ D_random * M). Similarly, Z = Σ ψ(K_j): O(N ∗ D_random).

Then for each query i, we compute ψ(Q_i) S: cost O(2D_random * M). And ψ(Q_i) Z: cost O(2D_random). So for N queries: O(N ∗ D_random * M).

So total cost:

- Compute ψ for Q and K: O(N D_random D)
- Compute S and Z: O(N D_random M)
- Compute V'_i for each i: O(N D_random M)

Impressively, DeepSeek is able to determine the correct computational complexity for computing V'

So overall: O(N D_random (D + M)).

We also have the cost of generating the random vectors w_i: O(D_random * D), but that is done once and can be considered preprocessing.

So the computational cost is O(N D_random (D + M)).

Now, note that in the problem, we are asked: "derive the computation cost for computing all the V′ here." So we can state that.

But wait, we have an approximation. The random feature map approximates the Gaussian kernel, and then we have the additional terms to get the softmax similarity. But as we showed, we can absorb those terms into the feature map, so we are effectively using a feature map ψ that approximates the softmax similarity.

Alternatively, we can note that the softmax similarity is not a kernel in the usual sense because it is not symmetric? Actually, exp(q^T k) is a kernel (the exponential kernel). But the normalization by sqrt(D) is just a scaling. So we are approximating the exponential kernel. But the random feature map we are given is for the Gaussian kernel. However, we can use the fact that the exponential kernel can be written in terms of the Gaussian kernel with an additional factor. So the above approach is valid.

But note: the problem says: "Use ϕ_random to approximate the above softmax similarity function with Gaussian Kernel". So we are to use the given ϕ_random (which approximates the Gaussian kernel) and then incorporate the additional terms. So the above derivation is acceptable.

We can also note that if we set σ^2 appropriately, we might get a direct approximation. Alternatively, we can use random Fourier features for the exponential kernel directly. But the problem specifies using the Gaussian kernel and then the random feature map for the Gaussian kernel.

So we proceed.

Now, part (b): Causal masking.

We are given that in causal masking, for position i, we only sum over j from 1 to i.

So equation (1) becomes:

This is the key realization needed in order to solve this problem. After this, you can just linearize and plug in

V'*i = (Σ*{j=1}^i sin(Q_i, K_j) V_j) / (Σ_{j=1}^i sin(Q_i, K_j)).

And equation (2) becomes:

V'*i = (φ(Q_i) Σ*{j=1}^i φ(K_j)^T V_j) / (φ(Q_i) Σ_{j=1}^i φ(K_j)^T).

Now, we are to write these in terms of S_i and Z_i, defined as:

S_i = Σ_{j=1}^i φ(K_j)^T V_j, [note: S_i is a matrix of size (feature_dim x M)? Actually, φ(K_j) is a column vector of dimension d (the feature map dimension). V_j is a row vector of dimension M. So φ(K_j)^T V_j is a matrix of size (d x M)? Wait, careful:

In equation (2), we have: V'*i = (φ(Q_i) Σ*{j=1}^N φ(K_j)^T V_j) / (φ(Q_i) Σ_{j=1}^N φ(K_j)^T).

Here, φ(Q_i) is a row vector (1 x d), φ(K_j)^T is a column vector (d x 1)? Actually, if φ(K_j) is a column vector, then φ(K_j)^T is a row vector. But then φ(K_j)^T V_j: if V_j is a row vector (1 x M), then φ(K_j)^T V_j is a row vector (d x M)? Actually, no: φ(K_j)^T is 1 x d, V_j is 1 x M, so the product is not defined in the usual matrix multiplication. Alternatively, we might consider φ(K_j) as a column vector and then φ(K_j) V_j is an outer product. Let's check equation (2) from the problem:

In equation (2): V'*i = (φ(Q_i) Σ*{j=1}^N φ(K_j)^T V_j) / (φ(Q_i) Σ_{j=1}^N φ(K_j)^T).

The notation is ambiguous. In the numerator, we have φ(Q_i) times a sum. The sum is over φ(K_j)^T V_j. If φ(K_j) is a column vector (d x 1), then φ(K_j)^T is a row vector (1 x d). Then φ(K_j)^T V_j: if V_j is a row vector (1 x M), then we cannot multiply a (1xd) and (1xM) unless we take the transpose? Alternatively, we might consider V_j as a column vector. But in the context, V_j is the value vector, which is usually a row vector for the j-th token. Alternatively, we can think of V_j as a column vector of dimension M. Then φ(K_j)^T is a row vector (1 x d) and V_j is a column vector (M x 1)? Then the product φ(K_j)^T V_j is a scalar? That doesn't match.

Looking back at equation (2) in the problem statement:

V'*i = (φ(Q_i) Σ*{j=1}^N φ(K_j)^T V_j) / (φ(Q_i) Σ_{j=1}^N φ(K_j)^T).

I believe the intended meaning is that φ(K_j) is a row vector? Or perhaps we are using column vectors for the feature maps and then the multiplication is adjusted.

In the context of kernelized attention, we have:

V'_i = (Σ_j K(Q_i, K_j) V_j) / (Σ_j K(Q_i, K_j)).

And if K(q,k) = φ(q) φ(k)^T, then:

V'_i = (φ(Q_i) (Σ_j φ(K_j)^T V_j)) / (φ(Q_i) (Σ_j φ(K_j)^T)).

So here, φ(Q_i) is a row vector (1 x d), φ(K_j) is a row vector (1 x d), then φ(K_j)^T is a column vector (d x 1). Then φ(K_j)^T V_j: if V_j is a row vector (1 x M), then we cannot multiply a (d x 1) and (1 x M) without considering the transpose. Alternatively, we can consider V_j as a column vector (M x 1). Then φ(K_j)^T V_j is a (d x M) matrix? Actually, if φ(K_j)^T is d x 1 and V_j is M x 1, then the product is d x M? No, the product of a column vector and a row vector is an outer product. So if we take φ(K_j)^T (d x 1) and V_j^T (1 x M), then we get a d x M matrix. But the equation writes φ(K_j)^T V_j. So perhaps we are to interpret V_j as a column vector and then φ(K_j)^T V_j is a scalar? That doesn't make sense for the numerator.

Looking at the dimensions: We want V'_i to be a vector of dimension M. So the numerator must be a vector of dimension M. So if φ(Q_i) is 1 x d, and we multiply by a d x M matrix, we get 1 x M. So the sum Σ_j φ(K_j)^T V_j should be a d x M matrix.

So we can interpret: φ(K_j) is a column vector (d x 1). Then φ(K_j)^T is 1 x d. But then to get a d x M matrix, we do: φ(K_j) V_j^T, which is d x M if V_j^T is M x 1. But the equation writes φ(K_j)^T V_j. So perhaps the notation is that φ(K_j) is a row vector and then φ(K_j)^T is a column vector, and then we do φ(K_j)^T V_j meaning the outer product? Alternatively, we can define that V_j is a row vector and then we do φ(K_j)^T ⊗ V_j, but that is not standard.

To avoid confusion, let's define:

Let φ(x) be a column vector of dimension d. Then the kernel is K(q,k) = φ(q)^T φ(k). But then equation (2) becomes:

V'_i = (Σ_j φ(Q_i)^T φ(K_j) V_j) / (Σ_j φ(Q_i)^T φ(K_j)) = (φ(Q_i)^T Σ_j φ(K_j) V_j^T) / (φ(Q_i)^T Σ_j φ(K_j)) if we consider V_j as a row vector? Actually, no.

Wait, if V_j is a row vector (1 x M), then we can write:

V'_i = (Σ_j (φ(Q_i)^T φ(K_j)) V_j) / (Σ_j φ(Q_i)^T φ(K_j)).

But then we can factor out φ(Q_i)^T:

V'_i = (φ(Q_i)^T Σ_j φ(K_j) V_j) / (φ(Q_i)^T Σ_j φ(K_j)).

DeepSeek struggles a little bit with clearing up ambiguities in the mathematical notation used. It gives a valiant effort in resolving these ambiguities and creates a consistent alternative

But note that Σ_j φ(K_j) V_j: here φ(K_j) is a column vector (d x 1) and V_j is a row vector (1 x M), so φ(K_j) V_j is a d x M matrix. Then φ(Q_i)^T times that d x M matrix gives a 1 x M vector. And the denominator: φ(Q_i)^T Σ_j φ(K_j) is a scalar.

So in this notation, equation (2) should be written as:

V'*i = (φ(Q_i)^T Σ*{j=1}^N φ(K_j) V_j) / (φ(Q_i)^T Σ_{j=1}^N φ(K_j)).

But the problem writes φ(Q_i) without transpose and then φ(K_j)^T. So perhaps they are using row vectors for the feature maps. Then φ(Q_i) is 1 x d, φ(K_j) is 1 x d, then φ(K_j)^T is d x 1, and then φ(K_j)^T V_j is d x M? Only if we consider V_j as a row vector and then we do (d x 1) (1 x M) = d x M. Then φ(Q_i) (1 x d) times that d x M gives 1 x M. So that works.

So we assume:

- φ(x) is a row vector (1 x d).

- Then K(q,k) = φ(q) φ(k)^T.
- Then equation (2) is: V'_i = (φ(Q_i) [Σ_j φ(K_j)^T V_j]) / (φ(Q_i) [Σ_j φ(K_j)^T]).

And note that Σ_j φ(K_j)^T is a d x 1 vector? Actually, φ(K_j)^T is d x 1, so the sum is d x 1. Then φ(Q_i) (1 x d) times that is a scalar. And the numerator: Σ_j φ(K_j)^T V_j: each term is d x M, so the sum is d x M. Then φ(Q_i) times that is 1 x M.

So that is consistent.

Now, for causal masking, we define:

S_i = Σ_{j=1}^i φ(K_j)^T V_j (a d x M matrix)
Z_i = Σ_{j=1}^i φ(K_j)^T (a d x 1 vector)

Then equation (2) with causal masking becomes:

V'_i = (φ(Q_i) S_i) / (φ(Q_i) Z_i).

DeepSeek is able to land at the correct equation for V'i, but starts to overthink the intended answer and goes back to rederiving. It later confirms this result.

And equation (1) with causal masking becomes:

V'*i = (Σ*{j=1}^i sin(Q_i, K_j) V_j) / (Σ_{j=1}^i sin(Q_i, K_j)).

But we are asked to write equation (1) and (2) in terms of S_i and Z_i. Equation (2) is already in terms of S_i and Z_i. For equation (1), we cannot write it directly in terms of S_i and Z_i because equation (1) uses the sin function, not the feature map. But note that in the kernelized case, we have sin = φ(q) φ(k)^T. So if we are using the same kernel, then equation (1) becomes the same as equation (2). But the problem says "Write equation 1 and 2 in terms of S_i and Z_i". So for equation (1), we can write:

V'*i = (Σ*{j=1}^i φ(Q_i) φ(K_j)^T V_j) / (Σ_{j=1}^i φ(Q_i) φ(K_j)^T) = (φ(Q_i) Σ_{j=1}^i φ(K_j)^T V_j) / (φ(Q_i) Σ_{j=1}^i φ(K_j)^T) = (φ(Q_i) S_i) / (φ(Q_i) Z_i).

So both equations become the same in the kernelized case. But if we are not using the kernelized form, then equation (1) does not decompose. However, the problem is about kernelized linear attention, so we assume we are in the kernelized case. So we can write:

Equation (1) with causal masking: V'_i = (φ(Q_i) S_i) / (φ(Q_i) Z_i).

Equation (2) with causal masking: V'_i = (φ(Q_i) S_i) / (φ(Q_i) Z_i).

So they are the same.

Now, we are also to derive the computational complexity of this new causal masking formulation.

In the causal case, we cannot compute the global sum S and Z for all j. Instead, we compute for each i from 1 to N:

We can compute S_i and Z_i recursively:

S_i = S_{i-1} + ϕ(K_i)^T V_i, with S_0 = 0.
Z_i = Z_{i-1} + ϕ(K_i)^T, with Z_0 = 0.

Then for each i, we compute:

V'_i = (ϕ(Q_i) S_i) / (ϕ(Q_i) Z_i).

So the steps:

1. Compute ϕ for all queries and keys. We have N queries and N keys. Cost: O(N * cost_of_ϕ). But note that in the kernelized case, we have a feature map ϕ. The cost of computing ϕ for one vector is, say, O(d D) if the feature map is of dimension d and the original dimension is D. But in the random feature map, d = 2 D_random and the cost is O(D_random D).
2. Then we compute S_i and Z_i recursively. For each i from 1 to N:
    - Compute ϕ(K_i)^T V_i: cost O(d M) because ϕ(K_i)^T is d x 1 and V_i is 1 x M, so the outer product is d x M and we do N of them? Actually, we compute for each i, we add one term. So the total cost for computing all S_i is O(N d M). Similarly, Z_i: O(N d).
3. Then for each i, we compute ϕ(Q_i) S_i: cost O(d M). And ϕ(Q_i) Z_i: cost O(d). So for all i: O(N d M).

So total cost:

- Compute ϕ for Q and K: O(N * cost_of_ϕ). Let cost_of_ϕ = C. In the random feature case, C = O(D_random D).
- Then compute S_i and Z_i: O(N d M) and O(N d).
- Then compute V'_i: O(N d M).

So overall: O(N (C + d M)).

In the random feature case, d = 2 D_random, so:

O(N (D_random D + D_random M)) = O(N D_random (D + M)).

Which is the same as the non-causal case? But note that in the non-causal case, we computed S and Z for all j in one go and then for each i we used the same S and Z. In the causal case, we have to compute S_i and Z_i for each i, but we do it recursively so the total cost is still O(N d M) for the S_i and Z_i part, which is the same as the non-causal case. However, in the non-causal case, we computed S = Σ_{j=1}^N ϕ(K_j)^T V_j once and used it for every i. In the causal case, we compute S_i for each i, but note that S_i is different for each i. So we cannot reuse the same S for all i. But we compute them in a cumulative way, so the total cost is still O(N d M). So the overall complexity is the same.

But wait, in the non-causal case, we computed S once and then for each i we compute ϕ(Q_i) S, which is O(N d M). In the causal case, we compute for each i: ϕ(Q_i) S_i, and S_i is of the same size, so it is also O(N d M). So the complexity is the same.

However, note that in the causal case, we cannot parallelize the computation of S_i across i because each S_i depends on the previous. But the complexity in terms of operations is the same.

So we can state: The computational complexity for the causal masking formulation is O(N d (D + M)) if the feature map computation cost is O(d D). In the random feature case, d = 2 D_random, so O(N D_random (D + M)).

But note: in the causal case, we have to compute the feature map for each token, and then do the recursive sums. So the complexity is linear in N.

Now, we are to write the answers.

We eventually arrive at the correct compultational complexity for the causal case

So for part (a)i:

We rewrite the softmax similarity function as:

sim_softmax(q, k) = exp(q^T k / sqrt(D)) = exp(||q||^2/(2 sqrt(D))) * exp(-||q-k||^2/(2 sqrt(D))) * exp(||k||^2/(2 sqrt(D))) = exp(||q||^2/(2 sqrt(D))) * K_Gauss(q, k) * exp(||k||^2/(2 sqrt(D))),

where K_Gauss(q, k) = exp(-||q-k||^2/(2 sqrt(D))) and we set σ^2 = sqrt(D).

For part (a)ii:

We use the random feature map ϕ_random to approximate the Gaussian kernel. Then we define:

ψ(q) = exp(||q||^2/(2 sqrt(D))) ϕ_random(q).

Then we have:

sim_softmax(q, k) ≈ ψ(q) · ψ(k).

Then the attention computation becomes:

V'*i = (ψ(Q_i) Σ*{j=1}^N ψ(K_j)^T V_j) / (ψ(Q_i) Σ_{j=1}^N ψ(K_j)^T).

We can compute:

S = Σ_{j=1}^N ψ(K_j)^T V_j (a matrix of size (2 D_random x M))
Z = Σ_{j=1}^N ψ(K_j)^T (a vector of size (2 D_random))

Then V'_i = (ψ(Q_i) S) / (ψ(Q_i) Z).

The computational cost:

> - Compute ψ for all Q and K: O(N * (D_random D)) because for each vector, we compute the random features in O(D_random D) and then multiply by the scalar exp(||·||^2/(2 sqrt(D))), which is O(1). So total O(N D_random D).
> - Compute S and Z: O(N D_random M) and O(N D_random).
> - Compute V'_i for each i: O(N D_random M).
>
> So total: O(N D_random (D + M)).
>
> For part (b):
>
> With causal masking, equation (1) and (2) become:
>
> V'_i = (ϕ(Q_i) S_i) / (ϕ(Q_i) Z_i),
>
> where S_i = Σ_{j=1}^i ϕ(K_j)^T V_j and Z_i = Σ_{j=1}^i ϕ(K_j)^T.
>
> The computational complexity:
>
> We compute ϕ for all Q and K: O(N * cost_of_ϕ). Then we compute S_i and Z_i recursively for i=1 to N. The cost for computing S_i and Z_i is O(N d M) and O(N d), where d is the feature map dimension. Then for each i, we compute ϕ(Q_i) S_i and ϕ(Q_i) Z_i: O(N d M).
>
> So total complexity: O(N (cost_of_ϕ + d M)). In the random feature case, cost_of_ϕ = O(D_random D) and d = 2 D_random, so O(N D_random (D + M)).
>
> Now, we write the answers.

Problem 1: Kernelized Linear Attention (Part II)

(a) Approximating softmax attention with linearized kernel attention

i. Rewrite the softmax similarity function using Gaussian Kernel.

The softmax similarity function is:

$$\mathrm{sim}_{\mathrm{softmax}}(q,k) = \exp\left(\frac{q^\top k}{\sqrt{D}}\right).$$

Using the Gaussian kernel $\mathcal{K}_{\mathrm{Gauss}}(q,k) = \exp\left(-\frac{\|q-k\|_2^2}{2\sigma^2}\right)$ with $\sigma^2 = \sqrt{D}$, we have:

$$\exp\left(-\frac{\|q-k\|_2^2}{2\sqrt{D}}\right) = \exp\left(-\frac{\|q\|^2 + \|k\|^2 - 2q^\top k}{2\sqrt{D}}\right) = \exp\left(-\frac{\|q\|^2 + \|k\|^2}{2\sqrt{D}}\right)$$

Rearranging:

$$\exp\left(\frac{q^\top k}{\sqrt{D}}\right) = \exp\left(\frac{\|q\|^2}{2\sqrt{D}}\right)\exp\left(-\frac{\|q-k\|_2^2}{2\sqrt{D}}\right)\exp\left(\frac{\|k\|^2}{2\sqrt{D}}\right).$$

Thus,

$$\text{sim}_{\text{softmax}}(q,k) = \exp\left(\frac{\|q\|^2}{2\sqrt{D}}\right)\cdot \mathcal{K}_{\text{Gauss}}(q,k)\cdot \exp\left(\frac{\|k\|^2}{2\sqrt{D}}\right),$$

where $\mathcal{K}_{\text{Gauss}}(q,k) = \exp\left(-\frac{\|q-k\|_2^2}{2\sqrt{D}}\right)$.

ii. Use ϕ_{random} to approximate the softmax similarity function and derive the computation cost.

The random feature map ϕ_{random} is defined as:

$$\phi_{\text{random}}(x) = \sqrt{\frac{1}{D_{\text{random}}}}\left[\sin(\mathbf{w}_1 x),\ldots,\sin(\mathbf{w}_{D_{\text{random}}} x),\cos(\mathbf{w}_1 x),\ldots,\cos(\mathbf{w}_{D_{\text{ran}}}\right.$$

where $\mathbf{w}_i \sim \mathcal{N}(0,\sqrt{D}\cdot \mathbf{I}_D)$. This approximates the Gaussian kernel:

$$\mathbb{E}_{\mathbf{w}_i}[\phi_{\text{random}}(q)\cdot\phi_{\text{random}}(k)] = \exp\left(-\frac{\|q-k\|^2}{2\sqrt{D}}\right).$$

To approximate the softmax similarity, define a modified feature map:

$$\psi(x) = \exp\left(\frac{\|x\|^2}{2\sqrt{D}}\right)\phi_{\text{random}}(x).$$

Then:

$$\text{sim}_{\text{softmax}}(q,k) \approx \psi(q)\cdot\psi(k).$$

Using ψ, the attention output for position i becomes:

$$V_i' = \frac{\psi(Q_i)\sum_{j=1}^N \psi(K_j)^\top V_j}{\psi(Q_i)\sum_{j=1}^N \psi(K_j)^\top}.$$

Let:

- $S = \sum_{j=1}^{N} \psi(K_j)^\top V_j$ (a matrix of size $2D_{\text{random}} \times M$)
- $Z = \sum_{j=1}^{N} \psi(K_j)^\top$ (a vector of size $2D_{\text{random}}$)

Then:

$$V_i' = \frac{\psi(Q_i)S}{\psi(Q_i)Z}.$$

Computation cost:

- Compute ψ for all queries and keys:
  Each $\psi(x)$ requires computing $\|x\|^2$ ($O(D)$), the random features ($O(D_{\text{random}}D)$), and scaling ($O(1)$). For N queries and N keys: $O(2ND_{\text{random}}D)$.
- Compute S and Z:
  For each j, $\psi(K_j)^\top V_j$ costs $O(2D_{\text{random}}M)$. Summing over N: $O(ND_{\text{random}}M)$. Similarly, Z costs $O(ND_{\text{random}})$.
- Compute V_i' for all i:
  Each i requires $\psi(Q_i)S$ ($O(2D_{\text{random}}M)$) and $\psi(Q_i)Z$ ($O(2D_{\text{random}})$). For N queries: $O(ND_{\text{random}}M)$.

Total cost: $O(ND_{\text{random}}(D+M))$.

(b) Causal masking

With causal masking, the attention for position i only depends on positions $j \leq i$. Thus, equations (1) and (2) become:

Equation (1):

$$V_i' = \frac{\sum_{j=1}^{i} \text{sim}(Q_i, K_j)V_j}{\sum_{j=1}^{i} \text{sim}(Q_i, K_j)}.$$

Equation (2) (kernelized form):

$$V_i' = \frac{\phi(Q_i)\sum_{j=1}^{i} \phi(K_j)^\top V_j}{\phi(Q_i)\sum_{j=1}^{i} \phi(K_j)^\top}.$$

Define cumulative sums:

$$S_i = \sum_{j=1}^{i} \phi(K_j)^\top V_j, \quad Z_i = \sum_{j=1}^{i} \phi(K_j)^\top.$$

Then both equations can be written as:

$$V_i' = \frac{\phi(Q_i) S_i}{\phi(Q_i) Z_i}.$$

Computational complexity:

- Compute ϕ for all Q_i and K_i: same as before, $O(N \cdot \text{cost}(\phi))$. For random features, $O(ND_{\text{random}}D)$.
- Compute S_i and Z_i recursively: $S_i = S_{i-1} + \phi(K_i)^\top V_i$, $Z_i = Z_{i-1} + \phi(K_i)^\top$. Each update costs $O(dM)$ and $O(d)$, where $d = 2D_{\text{random}}$. Over N steps: $O(ND_{\text{random}}M)$ and $O(ND_{\text{random}})$.
- Compute V_i' for each i: $\phi(Q_i)S_i$ ($O(dM)$) and $\phi(Q_i)Z_i$ ($O(d)$). Total: $O(ND_{\text{random}}M)$.

Total cost: $O(ND_{\text{random}}(D + M))$, same as the non-causal case, but with sequential dependencies in the cumulative sums.

1503.03832v3.pdf
PDF 4.49MB

Read the attached paper on FaceNet, and answer all parts of Question 5: Read a Research Paper: FaceNet.

This problem is great at testing DeepSeek's capabilities of parsing a research paper with many mathematical equations, tables, and visuals

Thought for 56 seconds

We are to answer all parts of question 5 from the homework, which is about the FaceNet paper.
The paper is provided in the attachment. We need to read it and answer the questions (a) through (j).

Let's go through each question and answer based on the paper.

(a) What are the two neural network architectures considered by the authors?

From the paper, in Section 3.3 (Deep Convolutional Networks) and also in the Abstract and Introduction, the two architectures are:

Correct; these can be found in tables within the paper

1. Zeiler&Fergus [22] style networks with 1x1 convolutions (NN1 in Table 1).
2. Inception [16] type networks (NN2, NN3, NN4, NNS1, NNS2 in Table 3 and Figure 4).

(b) Briefly describe the triplet loss and how it differs from a typical supervised learning objective.

The triplet loss is defined in Section 3.1 (Triplet Loss). It takes an anchor image, a positive image (same identity as anchor), and a negative image (different identity). The loss function is:

L = ∑_i [||f(x_i^a) - f(x_i^p)||_2^2 - ||f(x_i^a) - f(x_i^n)||*2^2 + α]+*

where α is a margin, and [·]_+ is the hinge function (max(0, ·)).

The goal is to make the distance between the anchor and positive smaller than the distance between the anchor and negative by at least α.

Most importantly, the distinction is made between triplet loss and typical SL objectives

It differs from a typical supervised learning objective (like classification) because it does not use class labels directly. Instead, it uses relative comparisons (triplets) to learn a metric (embedding) such that similar examples are close and dissimilar examples are far apart.

(c) What is the challenge with generating all possible triplets? Briefly describe how the authors address this challenge.

Correctly identifies the main challenge and the ways in which the authors combat this challenge

The challenge is that the number of all possible triplets is huge (O(N^3) for N images). Moreover, many triplets are easy and do not contribute to the training (they already satisfy the triplet constraint). Training on all triplets would be inefficient and slow.

The authors address this by selecting hard triplets (triplets that violate the triplet constraint) or semi-hard triplets (where the negative is farther than the positive but still within the margin). They use an online triplet selection method within each mini-batch (see Section 3.2: Triplet Selection). They ensure that each mini-batch has a sufficient number of examples from each identity (around 40 faces per identity per mini-batch) and then select the hard positives and negatives from within the mini-batch.

(d) How many parameters and floating point operations (FLOPs) do the authors use for their neural network? How does this compare to a ResNet-50?

From the paper:

For the Zeiler&Fergus based model (NN1) in Table 1: 140 million parameters and 1.6 billion FLOPS.
For the Inception based model (NN2) in Table 2: 7.5 million parameters and 1.6 billion FLOPS.

The paper does not specify a ResNet-50, but we are to compare with ResNet-50 from the original ResNet paper.

According to the ResNet paper (He et al., 2016), ResNet-50 has about 25.6 million parameters and approximately 4.1 billion FLOPS (for a 224x224 input image).

So, the Inception based model (NN2) has about 1/3 the parameters and about 40% of the FLOPS of ResNet-50. The Zeiler&Fergus based model (NN1) has about 5.5 times the parameters of ResNet-50 but about 40% of the FLOPS.

DeepSeek successfully pulls the correct information from a completely separate research paper

(e) What do the authors mean by semi-hard negatives?

In Section 3.2 (Triplet Selection), the authors define semi-hard negatives as those negatives that are further away from the anchor than the positive, but still within the margin α. Specifically, they satisfy:

DeepSeek contradicts itself here, first claiming that semi-hard negatives are further away from the anchor than the positive, then switching up and saying that they are closer to the anchor than the positive

$$||f(x_i^a) - f(x_i^p)||_2^2 < ||f(x_i^a) - f(x_i^n)||_2^2 < ||f(x_i^a) - f(x_i^p)||_2^2 + \alpha$$

These negatives are hard because they are closer to the anchor than the positive plus the margin, but they are not the hardest (which would be the closest negative). The authors use these to avoid bad local minima (like a collapsed model) early in training.

(f) What are harmonic embeddings?

Harmonic embeddings are introduced in Section 7 (Appendix: Harmonic Embedding). They are embeddings generated by different models (say v1 and v2) that are compatible with each other, meaning that the distance between an embedding from v1 and an embedding from v2 is meaningful and can be compared. This is achieved by training the new model (v2) with a harmonic triplet loss that mixes v1 and v2 embeddings in the triplets.

Good

(g) How does the performance vary with embedding dimensionality?

Section 5.4 (Embedding Dimensionality) and Table 5 show that the performance (validation rate at 10E-3 false accept rate) for embedding dimensionalities 64, 128, 256, and 512 are:

64: 86.8% ± 1.7
128: 87.9% ± 1.9
256: 87.7% ± 1.9
512: 85.6% ± 2.0

The performance peaks at 128 dimensions, and then slightly drops for 256 and 512. The differences are not statistically significant except for 512 which drops more.

(h) How does the performance vary with increasing amounts of training data?

Table 6 in Section 5.5 (Amount of Training Data) shows:

> | #training images | VAL |
> |---|---|
> | 2,600,000 | 76.3% |
> | 26,000,000 | 85.1% |
> | 52,000,000 | 85.1% |
> | 260,000,000 | 86.2% |

DeepSeek accurately parses the table shown in the paper and gives relevant statistical analysis. I would have liked more here, though, such as quantifying the changes in improvement

> There is a big jump from 2.6M to 26M images, but then the improvement tapers off.
>
> (i) Briefly share your favorite emergent property/result of the learned behavior with a triplet loss from the paper.
>
> One interesting emergent property is the robustness to pose, illumination, and occlusion. For example, in Figure 1, the model can handle extreme pose and lighting variations. Also, in the face clustering results (Figure 7), the model groups together faces of the same person across large variations in age, pose, and occlusion.
>
> (j) Which approach taken by the authors interested you the most? Why? (≈ 100 words)
>
> The online triplet mining approach is particularly interesting. Instead of precomputing triplets, the authors dynamically select hard and semi-hard triplets within each mini-batch during training. This ensures that the model is always learning from meaningful examples and avoids the inefficiency of processing easy triplets. This approach is crucial for training with large-scale datasets and leads to faster convergence and better performance. The idea of curriculum learning (increasing difficulty) is also elegantly incorporated by the online selection.
>
> Now, let's write the answers in a structured way.

It is interesting to read the responses that DeepSeek gives to these open ended questions

Question 5: Read a Research Paper: FaceNet

(a) What are the two neural network architectures considered by the authors?

The authors explore two deep convolutional network architectures:

1. **Zeiler & Fergus–style networks** (22 layers deep) with added 1×1 convolutions (NN1 in Table 1).
2. **Inception-style networks** (GoogLeNet-based) that are more parameter- and FLOP-efficient (e.g., NN2, NN3, NN4, NNS1, NNS2 in Table 2 and Figure 4).

(b) Briefly describe the triplet loss and how it differs from a typical supervised learning objective.

The triplet loss learns an embedding $f(x)$ such that, for a triplet of images (anchor x^a, positive x^p of the same identity, and negative x^n of a different identity), the squared L2 distance between the anchor and positive is smaller than that between the anchor and negative by a margin α:

$$\|f(x^a) - f(x^p)\|_2^2 + \alpha < \|f(x^a) - f(x^n)\|_2^2.$$

Unlike typical classification losses (e.g., cross-entropy) that assign discrete labels, triplet loss directly optimizes relative distances in the embedding space, encouraging better metric learning for verification/recognition.

(c) What is the challenge with generating all possible triplets? Briefly describe how the authors address this challenge.
Generating all possible triplets (which grows cubically with dataset size) results in many easy triplets that satisfy the constraint trivially, slowing training and convergence.
The authors use **online triplet mining**: within each large mini-batch (≈1,800 images), they select hard positives (farthest same-identity sample) and semi-hard negatives (samples that violate the margin but are not the hardest) to ensure active triplets that contribute to learning.

(d) How many parameters and floating point operations (FLOPs) do the authors use for their neural network? How does this compare to a ResNet-50?

- **NN1 (Zeiler & Fergus–style)**: 140M parameters, 1.6B FLOPs.
- **NN2 (Inception-style)**: 7.5M parameters, 1.6B FLOPs.
- **ResNet-50** (from He et al., 2016): ~25.6M parameters, ~4.1B FLOPs (for 224×224 input).

FaceNet's Inception models are more parameter-efficient than ResNet-50 and require fewer FLOPs.

(e) What do the authors mean by semi-hard negatives?
Semi-hard negatives are negative examples that are farther from the anchor than the positive example, but still within the margin α:

$$\|f(x^a) - f(x^p)\|_2^2 < \|f(x^a) - f(x^n)\|_2^2 < \|f(x^a) - f(x^p)\|_2^2 + \alpha.$$

They are "hard" because they are close enough to challenge the model, but not the hardest (which could cause training instability).

(f) What are harmonic embeddings?

Harmonic embeddings are compatible embeddings produced by different model versions (e.g., v1 and v2) that can be directly compared. They are learned by mixing embeddings from both versions in the triplet loss, ensuring smooth model upgrades without recomputing all embeddings.

(g) How does the performance vary with embedding dimensionality?

Performance (validation rate at FAR=10^{-3}) peaks at 128 dimensions (87.9%), with slightly lower performance at 64 (86.8%), 256 (87.7%), and 512 (85.6%) dimensions (Table 5). The differences are small, but 128-D offers a compact yet effective representation.

(h) How does the performance vary with increasing amounts of training data?

Increasing training data from 2.6M to 26M images improves validation rate from 76.3% to 85.1% (Table 6). Gains taper beyond 26M images, with 260M images yielding 86.2%. This shows diminishing returns after tens of millions of examples.

(i) Briefly share your favorite *emergent* property/result of the learned behavior with a triplet loss from the paper.

The model exhibits remarkable invariance to **pose, illumination, occlusion, and even aging** without explicit 3D alignment (Figure 1 and Figure 7). This emerges purely from metric learning on large-scale data, showcasing the power of triplet loss to capture intrinsic identity manifolds.

(j) Which approach taken by the authors interested you the most? Why? (≈ 100 words)