Real-Time Neural Appearance Models — Supplemental material

TIZIAN ZELTNER*, NVIDIA, Switzerland
FABRICE ROUSSELLE*, NVIDIA, Switzerland
ANDREA WEIDLICH*, NVIDIA, Canada
PETRIK CLARBERG*, NVIDIA, Sweden
JAN NOVÁK*, NVIDIA, Czech Republic
BENEDIKT BITTERLI*, NVIDIA, USA
ALEX EVANS, NVIDIA, United Kingdom
TOMÁŠ DAVIDOVIČ, NVIDIA, Czech Republic
SIMON KALLWEIT, NVIDIA, Switzerland
AARON LEFOHN, NVIDIA, USA

ACM Reference Format:
Tizian Zeltner, Fabrice Rousselle, Andrea Weidlich, Petrik Clarberg, Jan Novák, Benedikt Bitterli, Alex Evans, Tomáš Davidovič, Simon Kallweit, and Aaron Lefohn. 2024. Real-Time Neural Appearance Models — Supplemental material. *ACM Trans. Graph.* 43, 3, Article 33 (June 2024), 33 pages. https://doi.org/10.1145/3659577

Contents

*Equal contribution. Order determined by a rock-paper-scissors tournament.

1 TRAINING PROCEDURE DETAILS

The following list summarizes the parameters used for training the neural materials shown in the main paper. All our training takes place in PyTorch [Paszke et al. 2019].

Optimizer. We use the ADAM optimizer [Kingma and Ba 2014] ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1\text{e}{-7}$, and zero weight decay) starting with a learning rate of 1e−3 that is progressively reduced based on a cosine decay until it reaches 1e−4. I.e. it is scaled by

$$\text{learning rate scale} = 0.1 + 0.5 * (1 - 0.1) * \left(1 + \cos\left(\frac{i\,\pi}{N}\right)\right), \tag{1}$$

where i and N are the current and maximum iteration count.

Mollification. Over the first $M = 20000$ iterations of training, the angle of our *mollification* cone is reduced to from 10 to 0 degrees with another cosine decay, i.e.

$$\text{mollification cone angle} = 0.5 * 10^\circ * \left(1 + \cos\left(\frac{i\,\pi}{M}\right)\right). \tag{2}$$

We use 256 sampled $\boldsymbol{\omega}_\text{o}$ directions within the cone in order to estimate the blurred version of the BRDF.

2 RUNTIME IMPLEMENTATION

2.1 Neural material model

In this section, we provide pseudocode for an unoptimized implementation of the neural material model for functional reproducibility of our results. The code follows the syntax of the Slang shading language [He et al. 2018], which is close to HLSL. The MLP network layers are here implemented using regular FP16 math operations in the shader code. Refer to Section 7 in the main paper for details on the specific optimizations we perform.

Listing 1. Pseudocode for fetching the latent code.

```
Texture2D<float4> latentTextures[2];  // 8D latent code stored as 2 x RGBA textures
SamplerState bilinearSampler;

/**
 * Fetch latent code (8D).
 * The gradients (duvdx, duvdy) represent a pixel/path footprint in texture space.
 * The u parameter is a random number in [0,1).
 */
void fetch_latent_code(float2 uv, float2 duvdx, float2 duvdy, float u, out half latents[8])
{
    // Compute LOD by approximating an isotropic footprint.
    float lod = 0.5 * log2(length(duvdx) * length(duvdy));

    // Scale by texture resolution to get fractional mip level.
    uint w, h;
    latentTextures[0].GetDimensions(w, h);
    lod += 0.5 * log2(w * h);

    // Pick between two nearest mip levels stochastically.
    int mip = frac(lod) < u ? floor(lod) : floor(lod) + 1;

    float4 l0 = latentTextures[0].SampleLevel(bilinearSampler, uv, mip);
    float4 l1 = latentTextures[1].SampleLevel(bilinearSampler, uv, mip);
    // Copy elements of l0, l1 into latents ...
}
```

Listing 2. Pseudocode for neural BRDF decoder (unoptimized).

```
/**
 * Evaluate the neural BRDF.
 * The weights W_{i,j} and biases b_i below are loaded from memory.
 * Incident (wi) and outgoing direction (wo) are in the local frame.
 * The function returns f(wi,wo) * dot(n,wo) (the cosine term is baked into the MLP).
 */
float3 eval_brdf(half latents[8], float3 wi, float3 wo)
{
    half input[20];
    // Copy latents into input[12], input[13], ...

    // Evaluate shading frame network to generate predicted shading frames (n=2).
    // Note that this is just a linear layer without biases or activations.
    // Input: latent code, output: array of (wi,wo) transformed into the predicted frames.
    half sf[12] = {};
    for (int i=0; i < 12; ++i)
        for (int j=0; j < 8; ++j)
            sf[i] += W^sf_{i,j} * latents[j]

    for (int i=0; i < 12; i+=6)
    {
```

```
        float3 N = normalize(float3(sf[i+0], sf[i+1], sf[i+2] + 1.0f));
        float3 T = normalize(float3(sf[i+3] + 1.0f, sf[i+4], sf[i+5]));
        float3 B = cross(N, T); // Note (T,B,N) is not orthogonalized

        float3 wi_pred = float3(dot(wi, T), dot(wi, B), dot(wi, N));
        float3 wo_pred = float3(dot(wo, T), dot(wo, B), dot(wo, N));
        // Copy wi_pred, wo_pred into input[i], input[i+1], ...
    }

    // Evaluate BRDF decoder network (2 x 32 neurons hidden layers).
    // Input: latent code and predicted frames, output: cosine-weighted BRDF RGB-value.

    // Hidden layer (l=1)
    half h1[32] = { b0^l1, b1^l1, ... };
    for (int i=0; i < 32; ++i)
        for (int j=0; j < 20; ++j)
            h1[i] += W_{i,j}^l1 * input[j];

    for (int i=0; i < 32; ++i)
        h1[i] = max(h1[i], 0.f); // ReLU activation

    // Hidden layer (l=2)
    half h2[32] = { b0^l2, b1^l2, ... };
    for (int i=0; i < 32; ++i)
        for (int j=0; j < 32; ++j)
            h2[i] += W_{i,j}^l2 * h1[j];

    for (int i=0; i < 32; ++i)
        h2[i] = max(h2[i], 0.f); // ReLU activation

    // Output layer
    half output[3] = { b0^out, b1^out, b2^out };
    for (int i=0; i < 3; ++i)
        for (int j=0; j < 32; ++j)
            output[i] += W_{i,j}^out * h2[j];

    for (int i=0; i < 3; ++i)
        output[i] = exp(output[i] - 3.0f); // Exponential activation (offset=-3)

    return float3(output[0], output[1], output[2]);
}
```

Listing 3. Pseudocode for neural BRDF sampler (unoptimized).

```
/**
 * Sample the neural BRDF.
 * The weights W_{i,j} and biases b_i below are loaded from memory.
 * Incident (wi) and sampled outgoing direction (wo) are in the local frame.
 * The function returns the weight: f(wi,wo) * dot(n,wo) / pdf.
 */
float3 sample_brdf(half latents[8], float3 wi, out float3 wo, out float pdf)
{
    half input[11];
    // Copy wi into first three elements
    // Copy latents into input[3], input[4], ...

    // Evaluate BRDF sampling network (3 x 32 neurons hidden layers).
    // Input: latent code and wi, output: sampling parameters.
```

```
// Hidden layer (l=1)
half h1[32] = { b_0^l1, b_1^l1, ... };
for (int i=0; i < 32; ++i)
    for (int j=0; j < 11; ++j)
        h1[i] += W_{i,j}^l1 * input[j];

for (int i=0; i < 32; ++i)
    h1[i] = max(h1[i], 0.f); // ReLU activation

// Hidden layer (l=2)
half h2[32] = { b_0^l2, b_1^l2, ... };
for (int i=0; i < 32; ++i)
    for (int j=0; j < 32; ++j)
        h2[i] += W_{i,j}^l2 * h1[j];

for (int i=0; i < 32; ++i)
    h2[i] = max(h2[i], 0.f); // ReLU activation

// Hidden layer (l=3)
half h3[32] = { b_0^l3, b_1^l3, ... };
for (int i=0; i < 32; ++i)
    for (int j=0; j < 32; ++j)
        h3[i] += W_{i,j}^l3 * h2[j];

for (int i=0; i < 32; ++i)
    h3[i] = max(h3[i], 0.f); // ReLU activation

// Output layer (linear activation)
half output[9] = { b_0^out, b_1^out, ... };
for (int i=0; i < 9; ++i)
    for (int j=0; j < 32; ++j)
        output[i] += W_{i,j}^out * h3[j];

// Compute sampling paramters based on network outputs.
int k = 0;
float alphaX = 1e-4f + 0.5f * (1.f + tanh_approx(output[k++]));
float alphaY = 1e-4f + 0.5f * (1.f + tanh_approx(output[k++]));
float rho = tanh_approx(output[k++]);

float slopeSpecX = sinh_approx(output[k++]);
float slopeSpecY = sinh_approx(output[k++]);
float slopeDiffX = sinh_approx(output[k++]);
float slopeDiffY = sinh_approx(output[k++]);

float wSpec = exp(output[k++]);
float wDiff = exp(output[k]);
float norm = 1.f / (wSpec + wDiff);
wSpec *= norm;
wDiff *= norm;

float3 alpha = float3(alphaX, alphaY, rho);
float2 slopeSpec = float2(slopeSpecX, slopeSpecY);
float2 slopeDiff = float2(slopeDiffX, slopeDiffY);
float2 weights = float2(wSpec, wDiff);

// Sample analytic model to obtain wo, pdf.
```

```
    wo = sample_analytic(alpha, ..., pdf);

    // Return weight.
    return eval_brdf(latents, wi, wo) / pdf;
}
```

Listing 4. Utility functions used by the BRDF sampler.

```
float tanh_approx(float x) { return x / sqrt(1 + x * x); }
float sinh_approx(float x) { return x * sqrt(1 + x * x); }

void tangents(float3 n, out float3 s, out float3 t)
{
    const float3 u = float3(1.f, 0.f, 0.f);
    s = normalize(u - n * dot(n, u));
    t = cross(n, s);
}

float3 to_world(float3 w, float3 n)
{
    float3 s, t;
    tangents(n, s, t);
    return w.x * s + w.y * t + w.z * n;
}

float3 to_local(float3 w, float3 n)
{
    float3 s, t;
    tangents(n, s, t);
    return float3(dot(w, s), dot(w, t), dot(w, n));
}

float3 sample_diffuse(float3 wi, float2 slope, float2 u)
{
    // Samle from cosine distribution.
    float3 wo = <sample hemisphere from cosine distribution>

    // Apply normal mapping.
    float3 n = normalize(float3(-slope.x, -slope.y, 1.f));
    return to_world(wo, n);
}

float pdf_diffuse(float3 wi, float3 wo, float2 slope)
{
    // Apply normal mapping.
    float3 n = normalize(float3(-slope.x, -slope.y, 1.f));
    wo = to_local(wo, n);

    // Evaluate cosine density.
    return wo.z / M_PI;
}

float3 sample_specular(float3 wi, float3 alpha, float2 slope, float2 u)
{
    float rho = alpha.z;
    float sqrtOneMinusRho = sqrt(1.f - rho * rho);

    // Sample base configuration slope.
```

```
    float s = sqrt(u.x) / sqrt(1 - u.x);
    float phi = 2 * M_PI * u.y;
    float sxStd = s * cos(phi);
    float syStd = s * sin(phi);

    // Stretch slope based on elliptical anisotropy.
    float sx = alpha.x * sxStd;
    float sy = alpha.y * (rho * sxStd + sqrtOneMinusRho * syStd);

    // Apply the slope offset.
    sx += slope.x;
    sy += slope.y;

    // Convert slope to a half-vector.
    float3 wh = normalize(float3(-sx, -sy, 1.f));
    return 2.f * dot(wi, wh) * wh - wi;
}

float pdf_specular(float3 wi, float3 wo, float3 alpha, float2 slope)
{
    float rho = alpha.z;
    float sqrtOneMinusRho = sqrt(1.f - rho * rho);

    // Construct half-vector.
    float3 wh = normalize(wi + wo);
    wh *= sign(wh.z);

    float cosTheta = wh.z;
    if (cosTheta <= 1e-4f)
    {
        return 0.f;
    }

    // Convert to slope space.
    float sx = -wh.x / cosTheta;
    float sy = -wh.y / cosTheta;

    // Revert the slope offset.
    sx -= slope.x;
    sy -= slope.y;

    // Unstretch slope based on elliptical anisotropy.
    float sxStd = sx / alpha.x;
    float normalization = 1.f / (alpha.x * alpha.y * sqrtOneMinusRho);
    float syStd = (alpha.x * sy - rho * alpha.y * sx) * normalization;

    // Evaluate basic slope density.
    float r2 = sxStd * sxStd + syStd * syStd;
    float p22Std = 1.f / (M_PI * (1.f + r2) * (1.f + r2));
    float p22 = p22Std * normalization;

    float pdfH = p22 / (cosTheta * cosTheta * cosTheta);

    // Account for specular reflection Jacobian.
    float Jh = 4 * abs(dot(wi, wh));
    return pdfH / Jh;
}
```

```
// Samples the analytic BRDF model.
// The u parameter is a uniform random number in [0,1)^2.
// The function returns the sampled direction wo and pdf.
float3 sample_analytic(float3 alpha, float2 slopeSpec, float2 slopeDiff, float weightSpec,
    float3 wi, float2 u, out float pdf)
{
    float3 wo;
    if (u.x < weightSpec) // specular lobe
    {
        u.x = u.x / weightSpec;
        wo = sample_specular(wi, alpha, slopeSpec, u);
    }
    else // diffuse lobe
    {
        u.x = (u.x - weightSpec) / (1.f - weightSpec);
        wo = sample_diffuse(wi, slopeDiff, u);
    }

    pdf = 0.f;
    pdf += weightSpec * pdf_specular(wi, wo, alpha, slopeSpec);
    pdf += (1.f - weightSpec) * pdf_diffuse(wi, wo, slopeDiff);

    return wo;
}
```

2.2 Integration in renderer

It should be clear that the neural BRDF evaluation and sample operations above can be integrated in any physically-based renderer based on path tracing or rasterization. For completeness, we include pseudocode for one example of each here.

Listing 5. Pseudocode for integration in path tracer.

```
RWTexture2D<float4> color;  // Frame buffer.

struct PathState
{
    float3 pos;  // Position in world space.
    float3 dir;  // Direction in world space.
    float3 thp;  // Current path throughput.
    float3 L;    // Accumulated path contribution.
    ...
}

[shader("miss")]
void miss(inout PathState path)
{
    // Path missed the scene; add contribution from environment map.
    float3 env_light = eval_envmap(path.dir);
    path.L += path.thp * env_light;
    path.terminate();
}

[shader("closesthit")]
void closest_hit(inout PathState path)
{
    // Compute texture footprint by projecting current path footprint to texture space
```

```
    // using ray differentials [Igehy 1999] or ray cones [Akenine-M\"{o}ller et al. 2021].
    float2 duvdx = ...
    float2 duvdy = ...

    half latents[8];
    fetch_latent_code(duvdx, duvdy, latents);

    // Sample illumination using next-event estimation.
    uint i = <Pick light source>
    float3 L = normalize(light[i].pos - path.pos);
    float3 wi = <Transform path.dir to local frame>
    float3 wo = <Transform L to local frame>

    TraceRay(scene.accel, ...)  // Trace shadow ray
    if (<Light is visible>)
    {
        path.L += path.thp * eval_brdf(latents, wi, wo) * light[i].emission;
    }

    // Generate next path segment.
    if (path.bounces >= max_bounces)
    {
        path.terminate();
    }
    else
    {
        // Sample BRDF to generate path direction.
        float3 wo;
        float pdf;
        path.thp *= sample_brdf(latents, wi, wo, pdf);
        path.dir = <Transform wo to world space>
    }
}

[shader("raygeneration")]
float4 raygen()
{
    PathState path = <Setup camera path>

    while (!path.is_terminated())
    {
        // Trace path segment.
        // The hit/miss shaders update the ray payload (path).
        TraceRay(scene.accel, ..., path);
    }

    // Write color to frame buffer.
    uint2 pixel = DispatchRaysIndex().xy;
    color[pixel] = float4(path.L, 1.0f);
}
```

Listing 6. Pseudocode for integration in forward renderer (pixel shader).

```
    struct VSOut
    {
        linear float3 normal  : NORMAL;    // Shading normal in world space.
        linear float3 tangenW : TANGENT;   // Shading tangent in world space.
        linear float2 texC    : TEXCRD;    // Texture coordinate.
```

```
    linear float3 pos     : POSITION;  // Position in world space.
}

float4 ps_main(VSOut vs) : SV_Target0
{
    // Compute texture footprint using HW finite differences and fetch latent code.
    float2 duvdx = ddx(vs.texC);
    float2 duvdy = ddy(vs.texC);

    half latents[8];
    fetch_latent_code(duvdx, duvdy, latents);

    // Compute view direction.
    float3 V = normalize(vs.pos - camera.pos);
    float3 wi = <Transform V to the local frame>

    // Evaluate the accumulated contribution from all lights.
    float3 color = {};
    for (int i=0; i < lightCount; ++i)
    {
        float3 L = normalize(light[i].pos - vs.pos);
        float3 wo = <Transform L to local frame>

        // The cosine term is baked into the neural BRDF model.
        color += eval_brdf(latents, wi, wo) * light[i].emission;
    }

    return float4(color, 1.0f);
}
```

3 REFERENCE MATERIALS SHADER NODE GRAPHS

The following MaterialX node graphs were used to design the reference materials and illustrate their complexity. The initial setup was created in Houdini whereas some nodes would be later replaced by custom nodes. Please note that since MaterialX is not aware of the concept of layer weights, all BRDFs are mixed with transparent nodes to account for this. In practice we do not evaluate such BRDFs.

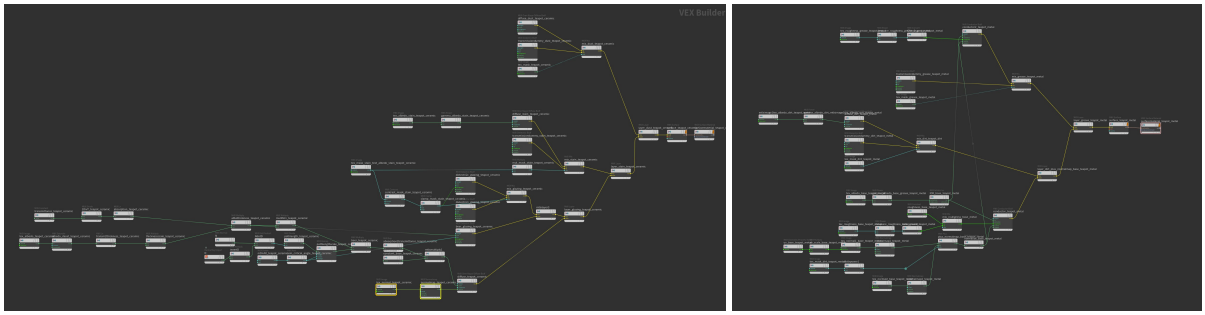

Fig. 1. MaterialX shader graph of the Teapot. Left: Ceramic glazing. Right: Metallic handle.

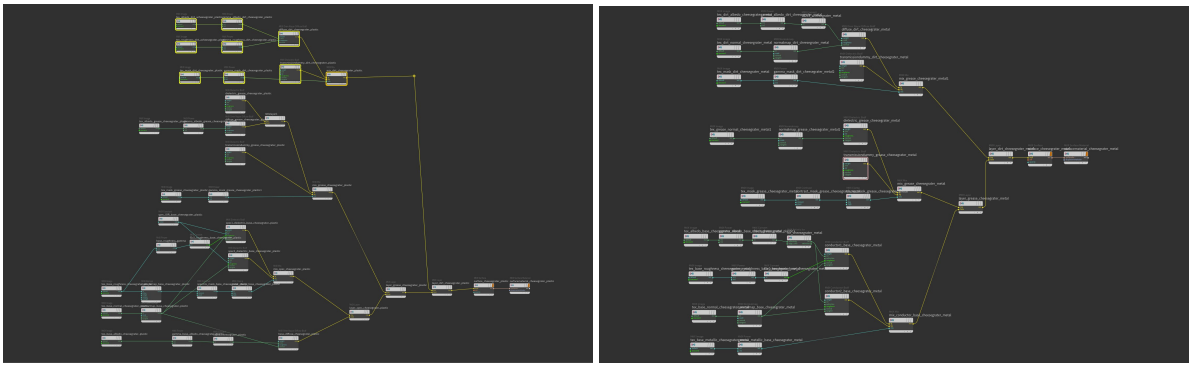

Fig. 2. MaterialX shader graph of the Cheese slicer. Left: Plastic handle. Right: Metallic blade.

Fig. 3. MaterialX shader graph of the Inkwell.

4 STAGE SCENE OVERVIEW

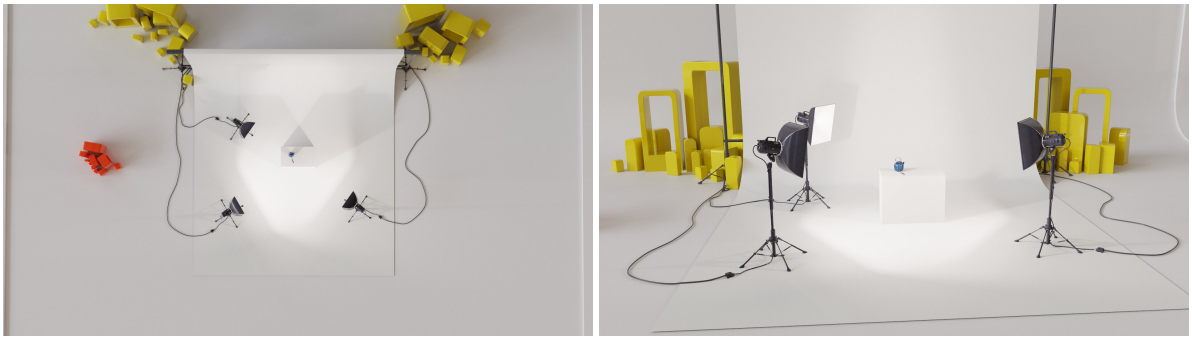

Fig. 4. Our neural BRDF exposes the same *eval*, *sample*, and *evalPdf* interfaces as traditional models, and can therefore be integrated with any physically-based renderer. To illustrate this generality and to produce visually pleasing results, all images were rendered using *unbiased* path tracing with realistic lighting setups. For example, the Stage scene is lit by a distant high-dynamic range environment map *and* three local stage lights. Each of the latter is modelled with emissive triangles placed in a softbox with a translucent diffuser. Due to the complexity of light transport with this lighting setup, we used 4096–8192 SPP for the final images. It should be noted that real-time applications would typically use denoising and/or simpler lighting setups for noise-free rendering at 1 SPP, which is an orthogonal problem not addressed by this work.

5 APPROXIMATION WITH AN 8-CHANNEL BRDF

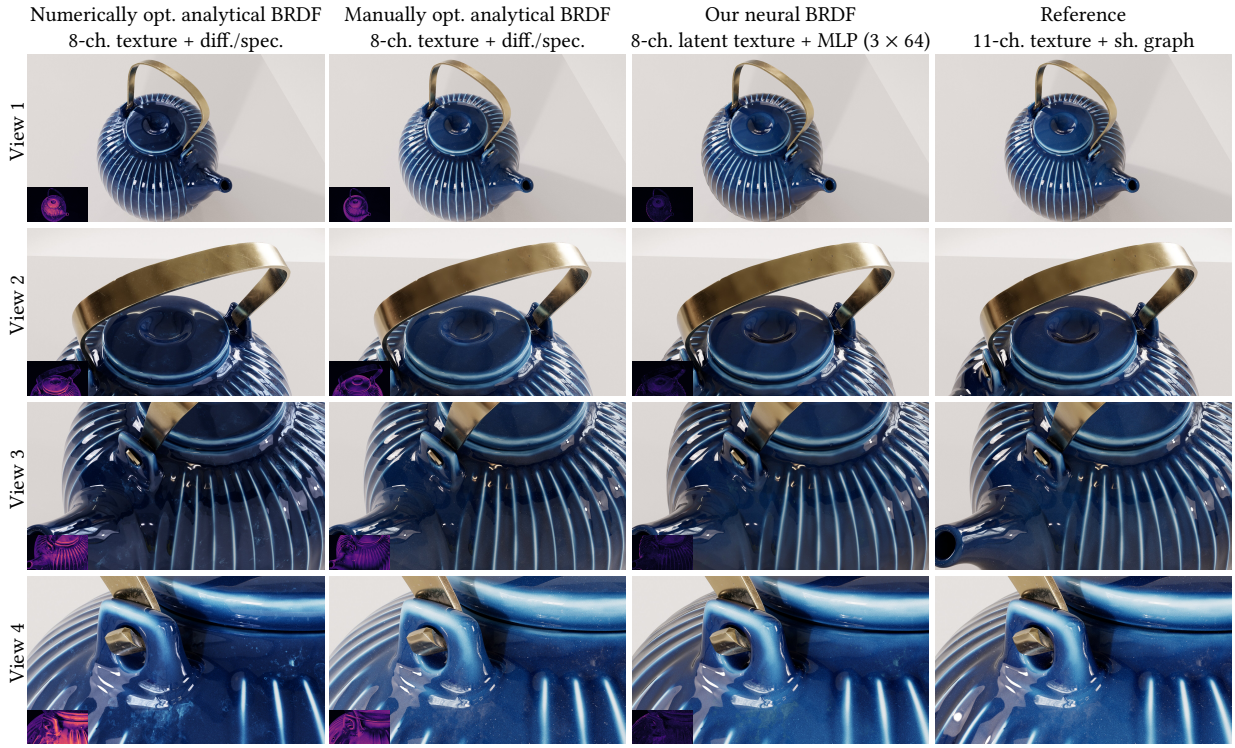

Fig. 5. Extended comparison of Figure 3 from the main paper. Our neural BRDF model successfully approximates the multi-layer Teapot materials with an 8 channel latent texture whereas a simple analytic BRDF with an equivalent number of input parameters has much higher approximation error. Small insets show the ꟻLIP error measuring the perceptual differences visible when toggling between each image and its corresponding reference. The individual renderings are also part of the supplemental image viewer to enable simpler inspection and comparison.

6 IMAGE ERROR METRICS

6.1 Ablation

Table 1. Image error metrics for the ablation study on the Inkwell metal body material (first row of Figure 8 in the main paper).

	Vanilla MLP	with encoder	with frame transform
Mean ꟻLIP	0.2589	0.1921	0.0703
Mean abs. error	0.0403	0.0284	0.0068
Mean sqr. error	0.0055	0.0029	0.0002
Mean rel. abs. error	0.2269	0.1443	0.0397
Mean rel. sqr. error	0.0676	0.0334	0.0043
SMAPE	0.2316	0.1700	0.0412

Table 2. Image error metrics for the ablation study on the Teapot ceramic body material (second row of Figure 8 in the main paper).

	Vanilla MLP	with encoder	with frame transform
Mean ꟻLIP	0.3343	0.2552	0.1362
Mean abs. error	0.1311	0.0979	0.0331
Mean sqr. error	0.1112	0.5560	0.0182
Mean rel. abs. error	0.2956	0.2767	0.1089
Mean rel. sqr. error	0.1369	0.1140	0.0176
SMAPE	0.3989	0.3999	0.1230

Table 3. Image error metrics for the ablation study on the Cheese slicer blade material (third row of Figure 8 in the main paper).

	Vanilla MLP	with encoder	with frame transform
Mean ꟻLIP	0.1381	0.1268	0.0774
Mean abs. error	0.0769	0.0522	0.0251
Mean sqr. error	0.0954	0.0402	0.0038
Mean rel. abs. error	0.1203	0.1027	0.0618
Mean rel. sqr. error	0.0444	0.0286	0.0107
SMAPE	0.1446	0.1183	0.0677

Table 4. Image error metrics for the ablation study on the Cheese slicer handle material (fourth row of Figure 8 in the main paper).

	Vanilla MLP	with encoder	with frame transform
Mean ꟻLIP	0.2245	0.2083	0.0423
Mean abs. error	0.0591	0.0822	0.0080
Mean sqr. error	0.0607	40.1738	0.0004
Mean rel. abs. error	0.2278	0.8520	0.0519
Mean rel. sqr. error	0.0704	1061.4310	0.0033
SMAPE	0.2931	0.2707	0.0532

6.2 Model quality

Table 5. Image error metrics for view 1 in the Inkwell scene (Figure 15 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.0303	0.0245	0.0244
Mean abs. error	0.0033	0.0025	0.0025
Mean sqr. error	0.0001	0.0001	0.0021
Mean rel. abs. error	0.0200	0.0150	0.0140
Mean rel. sqr. error	0.0012	0.0019	0.0008
SMAPE	0.0206	0.0154	0.0145

Table 6. Image error metrics for view 2 in the Inkwell scene (Figure 15 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.0617	0.0453	0.0430
Mean abs. error	0.0081	0.0054	0.0049
Mean sqr. error	0.0003	0.0002	0.0001
Mean rel. abs. error	0.0394	0.0274	0.0244
Mean rel. sqr. error	0.0030	0.0017	0.0014
SMAPE	0.0404	0.0282	0.0253

Table 7. Image error metrics for view 1 in the Stage scene (Figure 16 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.0967	0.0430	0.0321
Mean abs. error	0.0402	0.0139	0.0116
Mean sqr. error	1.0302	0.0110	0.0057
Mean rel. abs. error	0.0909	0.0317	0.0247
Mean rel. sqr. error	0.0367	0.0044	0.0029
SMAPE	0.1338	0.0347	0.0259

Table 8. Image error metrics for view 2 in the Stage scene (Figure 16 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.1658	0.0581	0.0454
Mean abs. error	0.1144	0.0165	0.0144
Mean sqr. error	8.1037	0.0436	0.0456
Mean rel. abs. error	0.1801	0.0466	0.0375
Mean rel. sqr. error	0.0651	0.0054	0.0037
SMAPE	0.2676	0.0514	0.0394

Table 9. Image error metrics for view 3 in the Stage scene (Figure 16 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.2771	0.1363	0.0985
Mean abs. error	0.0971	0.0331	0.0242
Mean sqr. error	0.5584	0.0176	0.0152
Mean rel. abs. error	0.2929	0.1089	0.0805
Mean rel. sqr. error	0.1210	0.0175	0.0097
SMAPE	0.4325	0.1231	0.0855

Table 10. Image error metrics for view 4 in the Stage scene (Figure 16 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.0895	0.0480	0.0392
Mean abs. error	0.0330	0.0200	0.0174
Mean sqr. error	0.0048	0.0019	0.0016
Mean rel. abs. error	0.0710	0.0401	0.0319
Mean rel. sqr. error	0.0147	0.0054	0.0032
SMAPE	0.0811	0.0429	0.0332

Table 11. Image error metrics for view 5 in the Stage scene (Figure 16 in the main paper).

	2×16	2×32	3×64
Mean FLIP	0.0400	0.0302	0.0280
Mean abs. error	0.0114	0.0099	0.0095
Mean sqr. error	0.0011	0.0007	0.0006
Mean rel. abs. error	0.0349	0.0307	0.0296
Mean rel. sqr. error	0.0054	0.0032	0.0030
SMAPE	0.0383	0.0317	0.0302

7 MODEL EVALUATION PLOTS

7.1 MERL database materials

We additionally trained a neural material on the complete MERL database [Matusik et al. 2003] including 100 different BRDFs. As these are obtains from real-world measurements we do not have access to suitable surface parameters (albedo, roughness, etc.) that could be fed into our encoder. Instead, the encoder receives the material index (1–100) via one-hot encoding. Finally, all neural variants of the MERL materials can be evaluated by the combination of i) a unique eight-dimensional latent code and ii) the decoder MLP that is shared between all 100 BRDFs. We show (log-scale) polar plots for our evaluation and importance sampling network evaluations in Figure 6 and Figure 7. PDF plots are scaled to compare against the learned BRDF up to a constant normalization factor.

7.2 High-fidelity materials

The remainder of this document contains a set of similar (log-scale) polar plots for the neural materials discussed in the main paper. Each BRDF subplot corresponds to a fixed spatial location (obtained via regularly sampling UV space). These additionally visualize any unqique per-layer surface normal that deviates from the canonical ($[0, 0, 1]$) z-axis. PDF plots are scaled to compare against the learned BRDF up to a constant normalization factor.

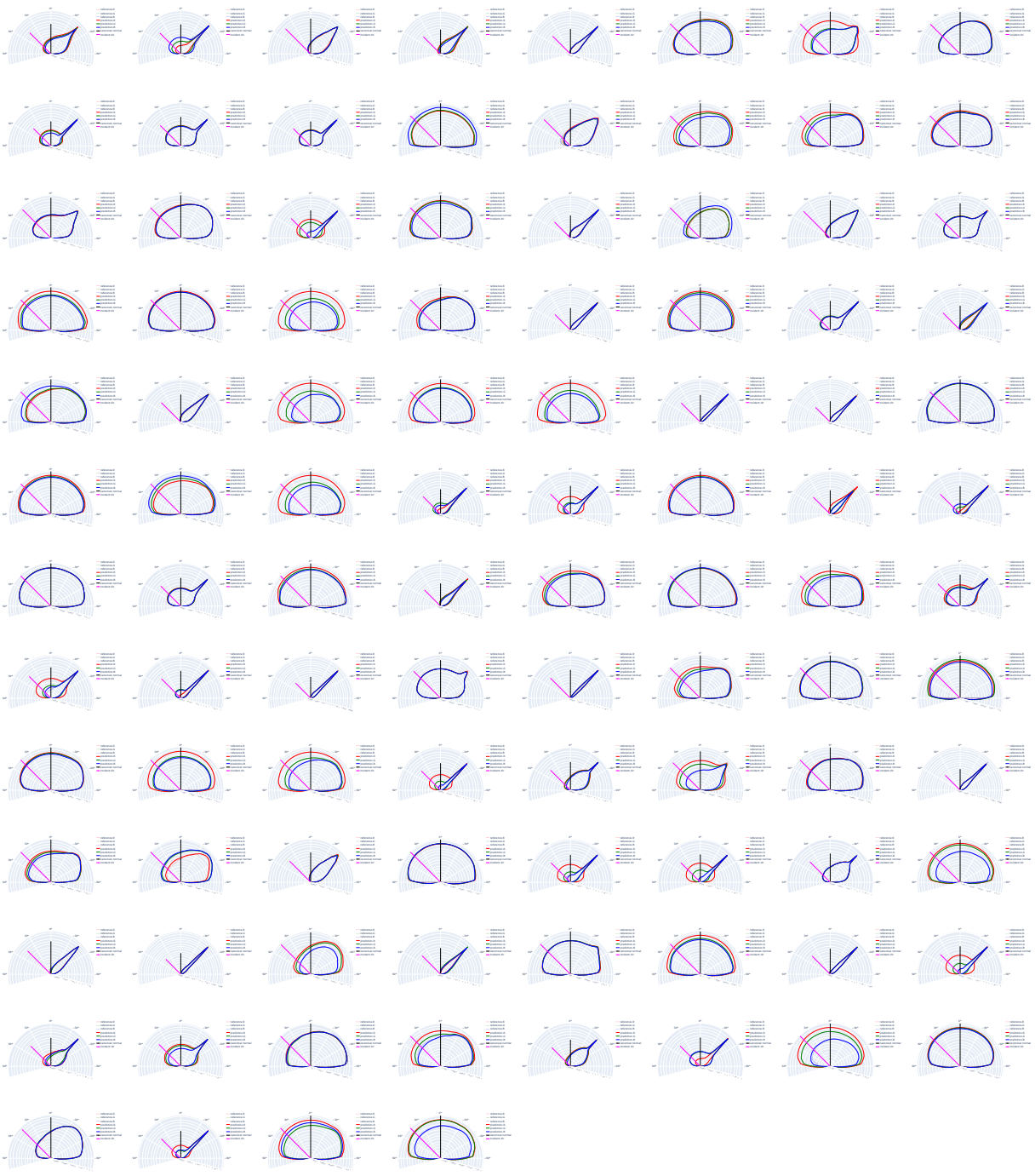

Fig. 6. MERL database: evaluation network (3 layers with 64 neurons) plots.

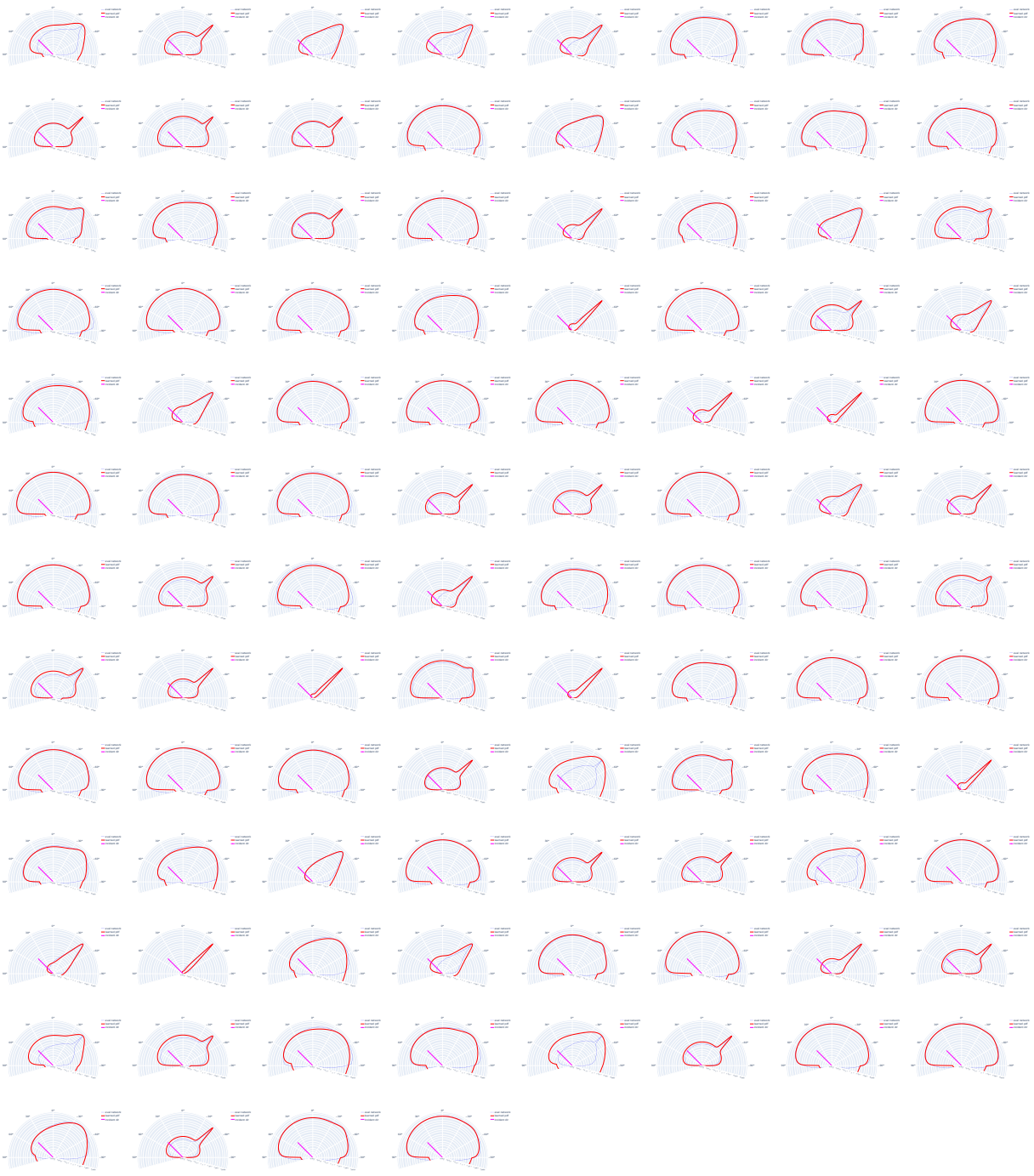

Fig. 7. MERL database: importance sampling network plots.

Teapot ceramic body (3 layers with 64 neurons)

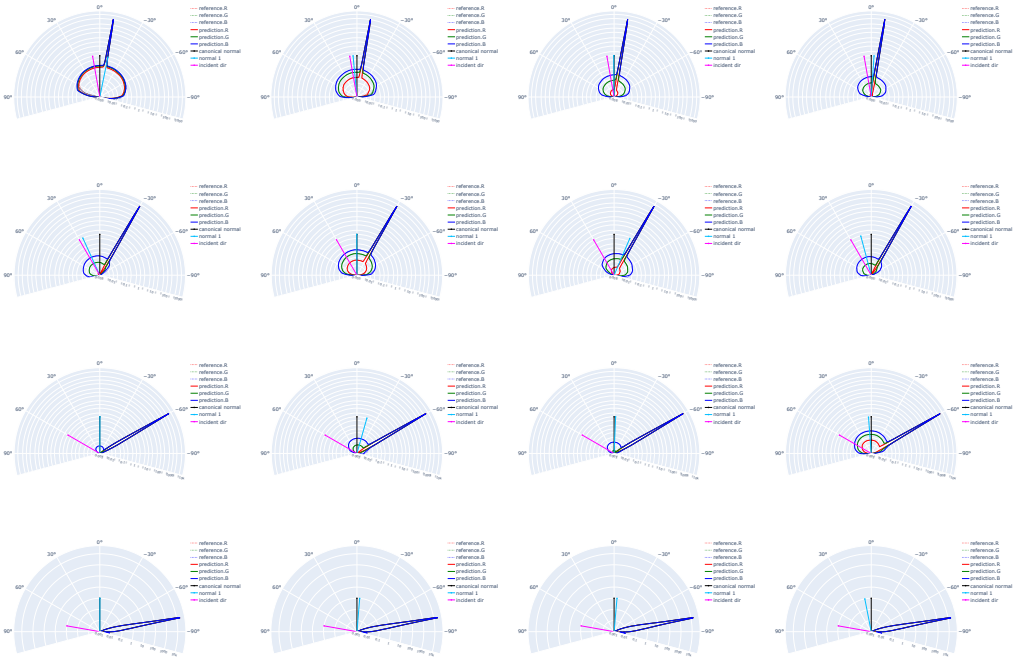

Fig. 8. Teapot ceramic body: evaluation network (3 layers with 64 neurons) plots.

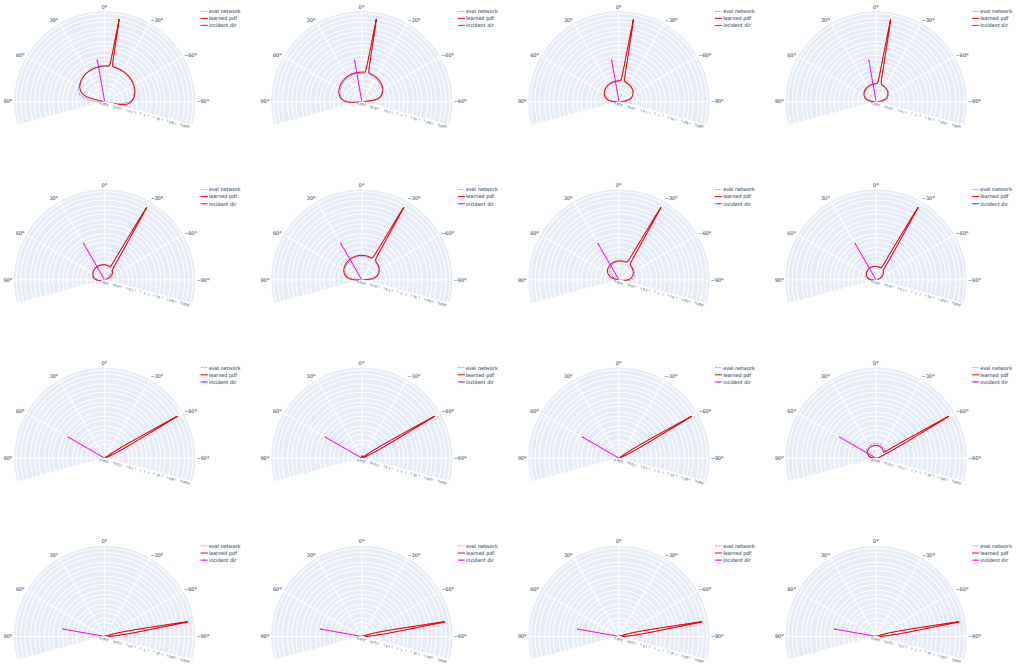

Fig. 9. Teapot ceramic body: importance sampling network plots.

Teapot ceramic body (2 layers with 32 neurons)

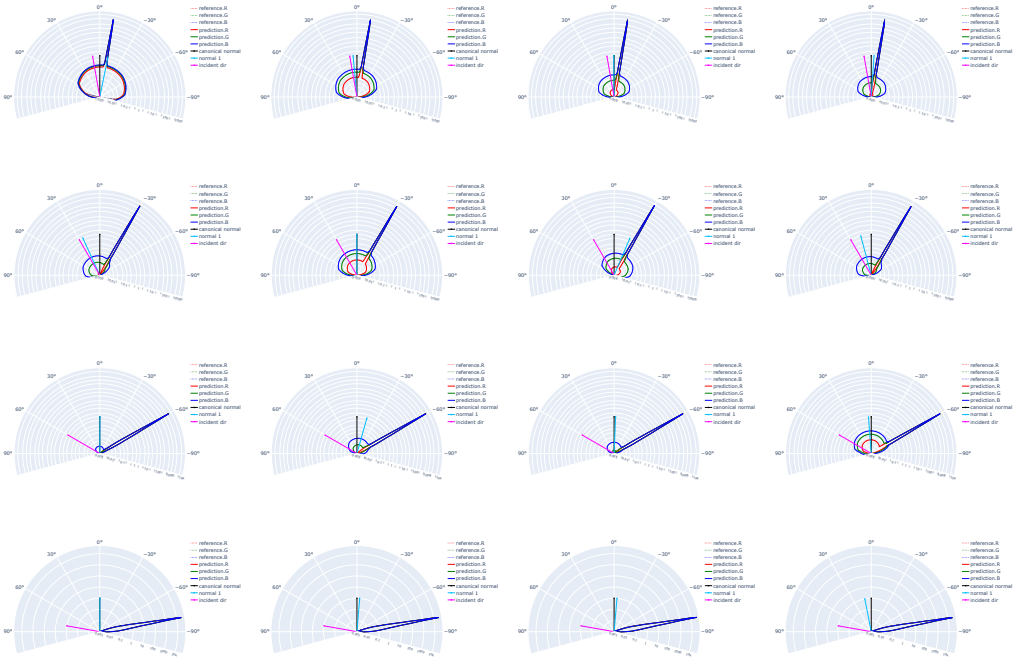

Fig. 10. Teapot ceramic body: evaluation network (2 layers with 32 neurons) plots.

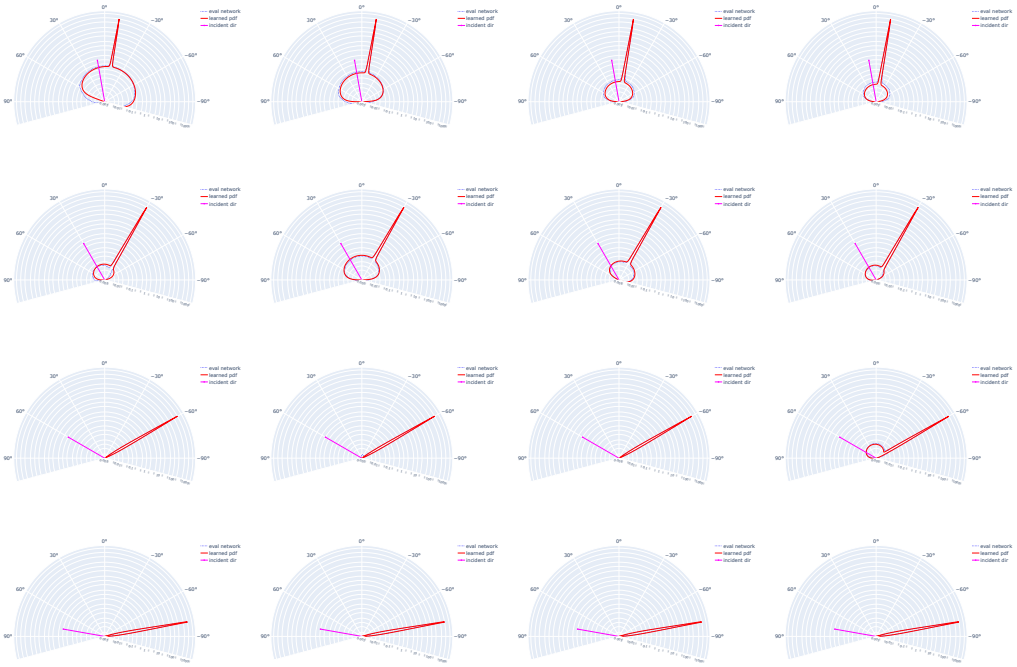

Fig. 11. Teapot ceramic body: importance sampling network plots.

Teapot ceramic body (2 layers with 16 neurons)

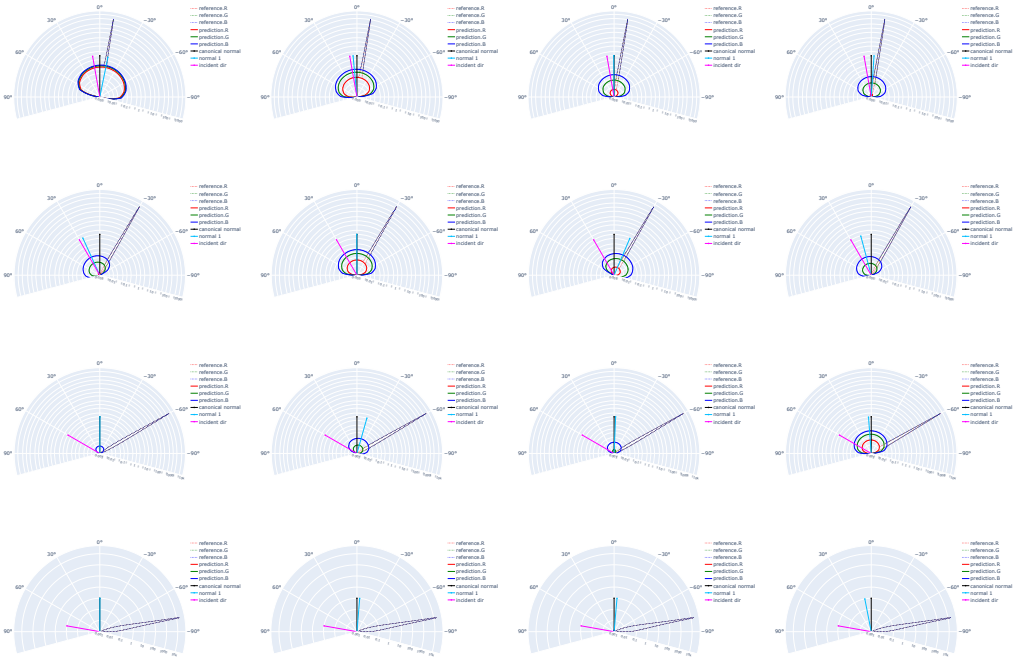

Fig. 12. Teapot ceramic body: evaluation network (2 layers with 16 neurons) plots.

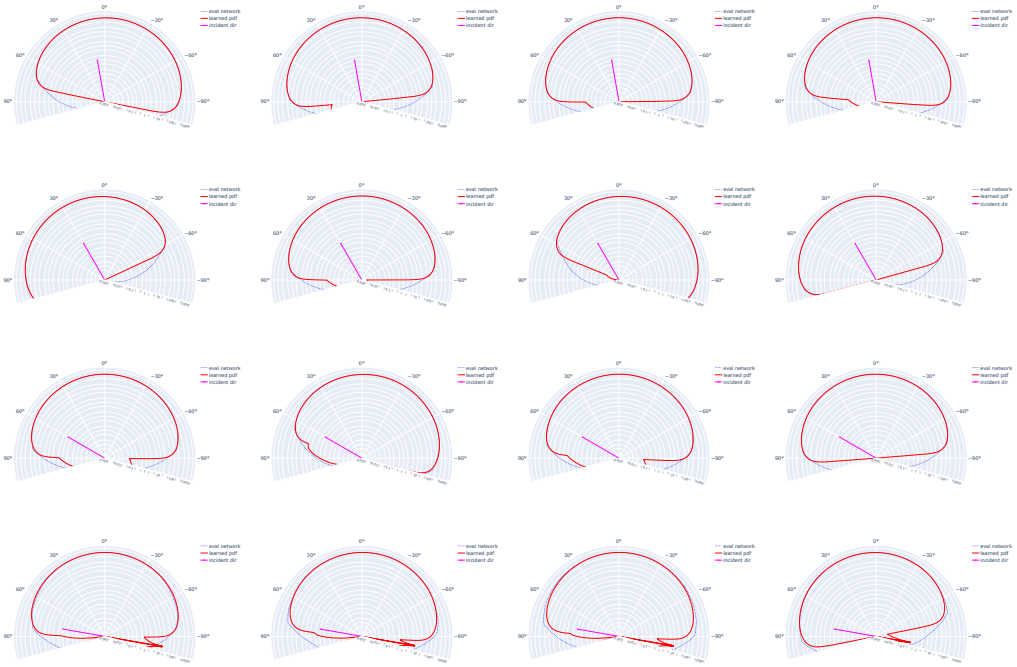

Fig. 13. Teapot ceramic body: importance sampling network plots.

Teapot metal handle (3 layers with 64 neurons)

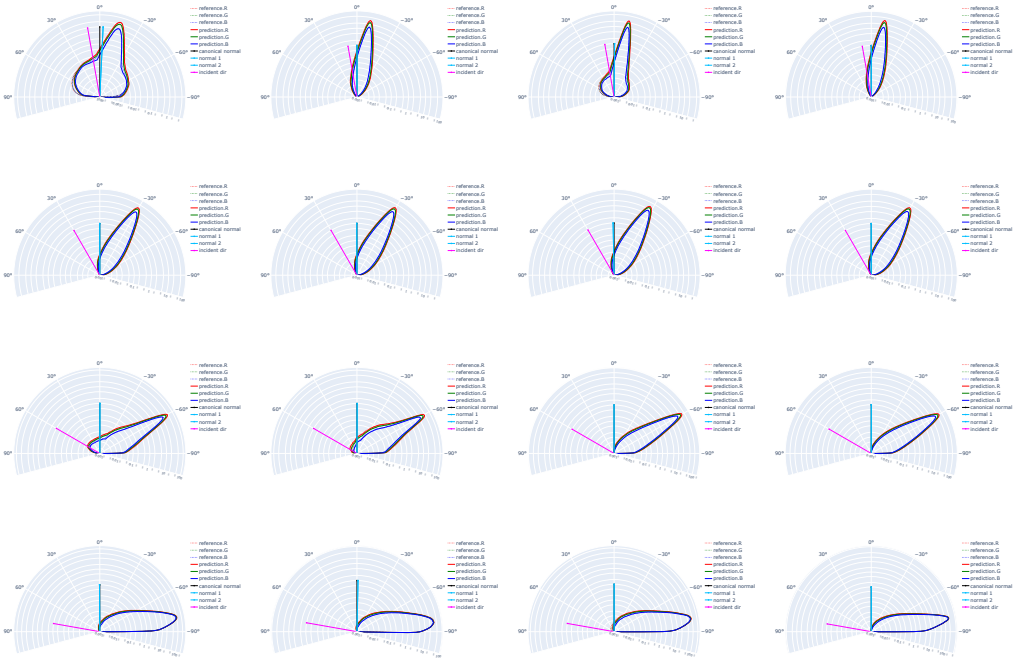

Fig. 14. Teapot metal handle: evaluation network (3 layers with 64 neurons) plots.

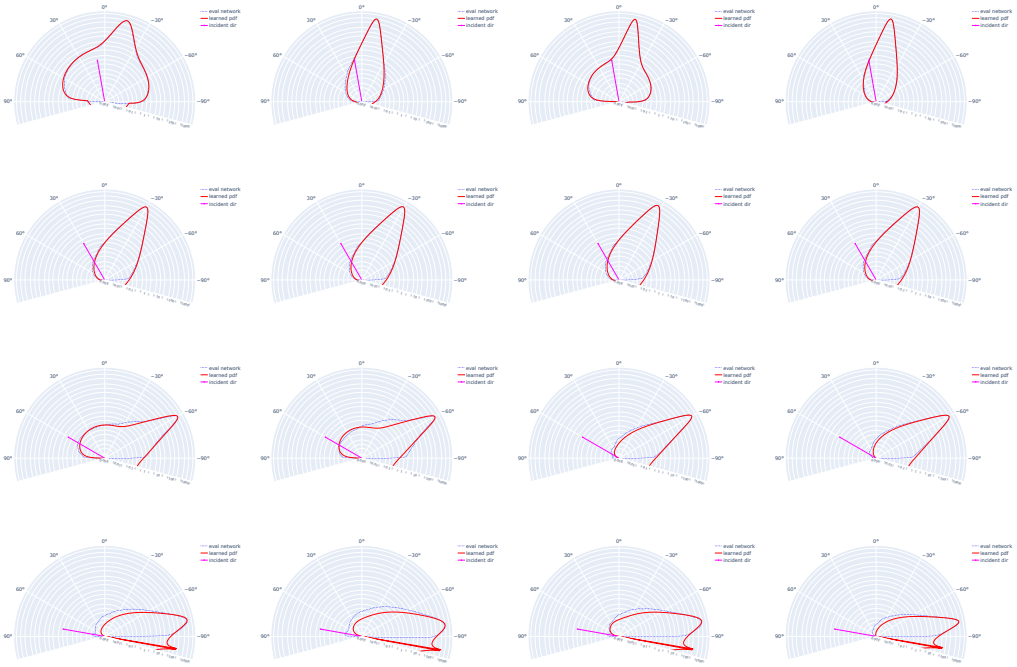

Fig. 15. Teapot metal handle: importance sampling network plots.

Teapot metal handle (2 layers with 32 neurons)

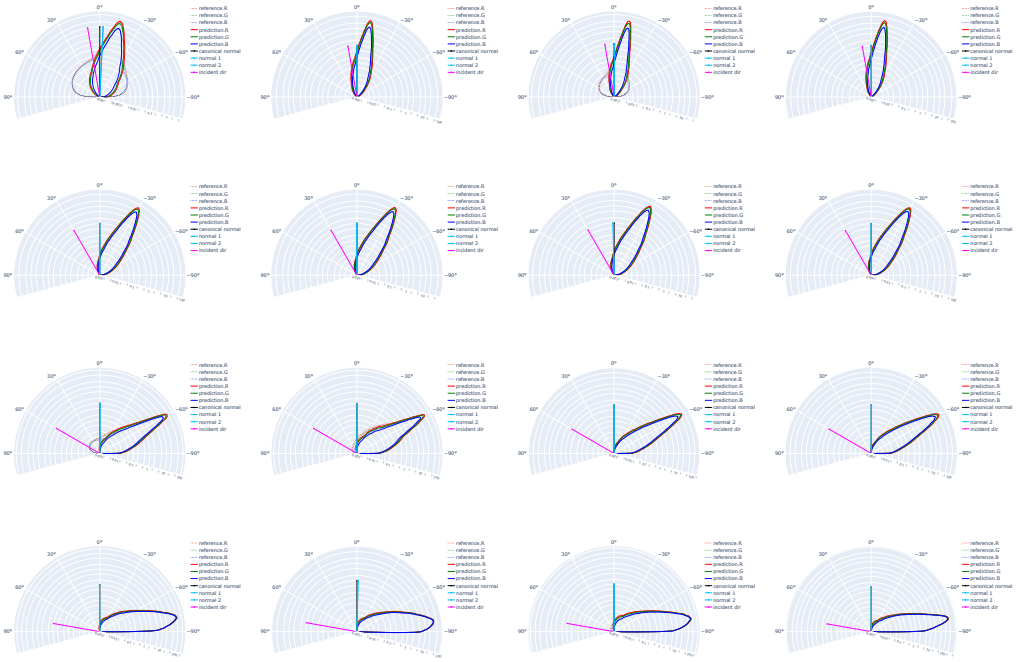

Fig. 16. Teapot metal handle: evaluation network (2 layers with 32 neurons) plots.

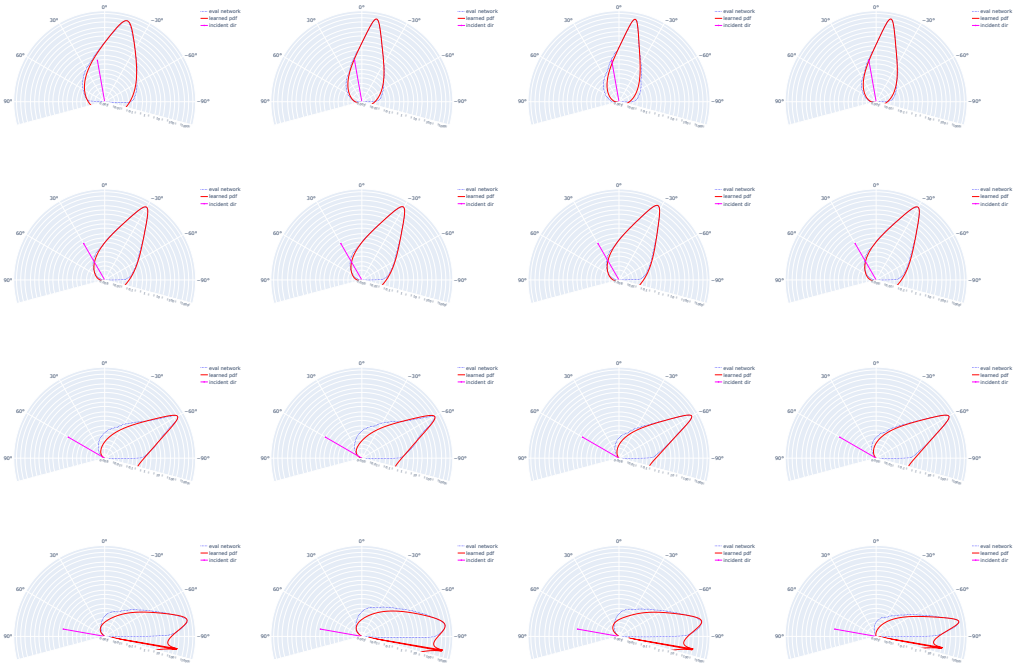

Fig. 17. Teapot metal handle: importance sampling network plots.

Teapot metal handle (2 layers with 16 neurons)

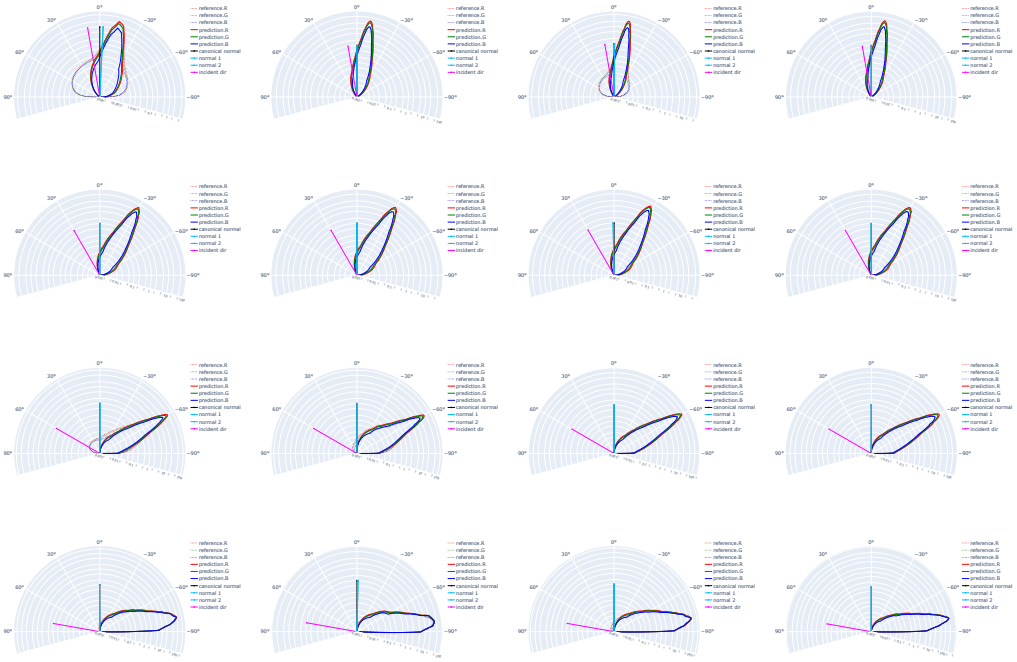

Fig. 18. Teapot metal handle: evaluation network (2 layers with 16 neurons) plots.

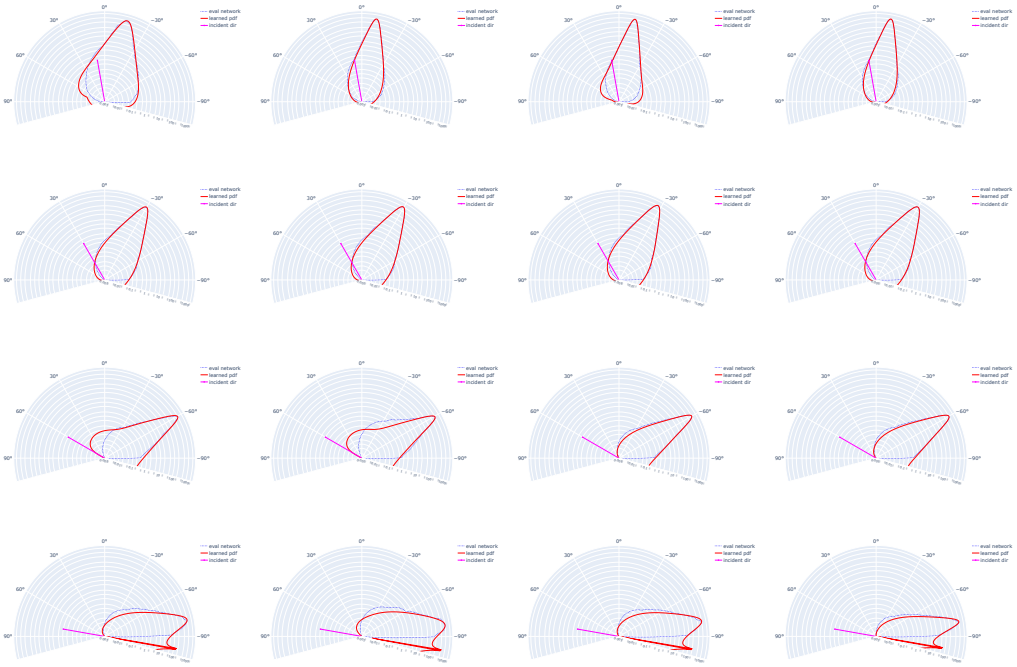

Fig. 19. Teapot metal handle: importance sampling network plots.

Cheese slicer blade(3 layers with 64 neurons)

Fig. 20. Cheese slicer blade: evaluation network (3 layers with 64 neurons) plots.

Fig. 21. Cheese slicer blade: importance sampling network plots.

Cheese slicer blade(2 layers with 32 neurons)

Fig. 22. Cheese slicer blade: evaluation network (2 layers with 32 neurons) plots.

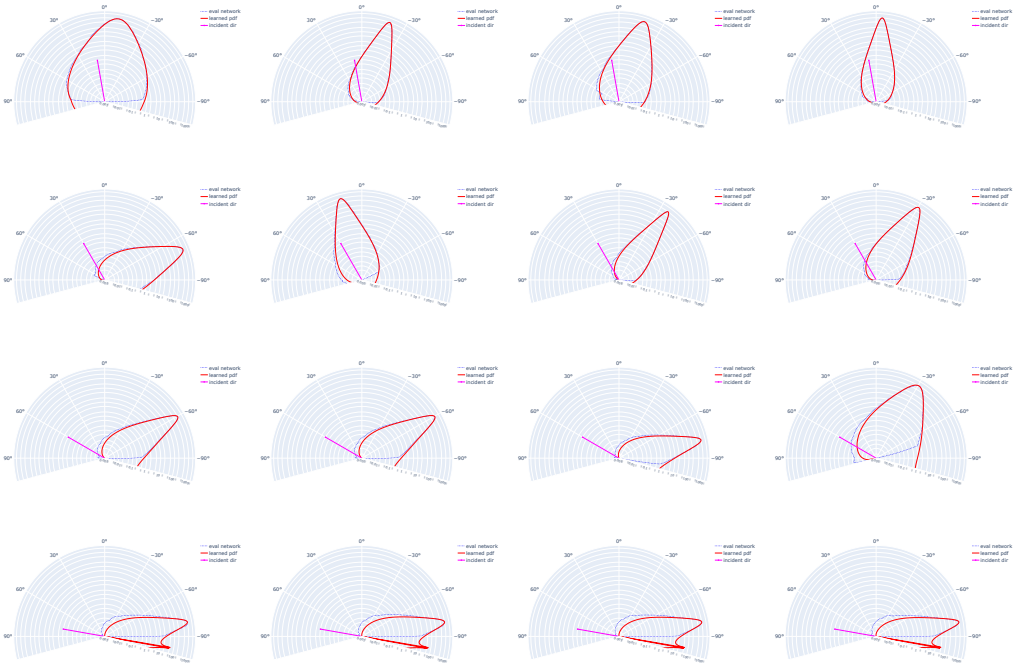

Fig. 23. Cheese slicer blade: importance sampling network plots.

Cheese slicer blade(2 layers with 16 neurons)

Fig. 24. Cheese slicer blade: evaluation network (2 layers with 16 neurons) plots.

Fig. 25. Cheese slicer blade: importance sampling network plots.

Cheese slicer handle(3 layers with 64 neurons)

Fig. 26. Cheese slicer handle: evaluation network (3 layers with 64 neurons) plots.

Fig. 27. Cheese slicer handle: importance sampling network plots.

Cheese slicer handle(2 layers with 32 neurons)

Fig. 28. Cheese slicer handle: evaluation network (2 layers with 32 neurons) plots.

Fig. 29. Cheese slicer handle: importance sampling network plots.

Cheese slicer handle(2 layers with 16 neurons)

Fig. 30. Cheese slicer handle: evaluation network (2 layers with 16 neurons) plots.

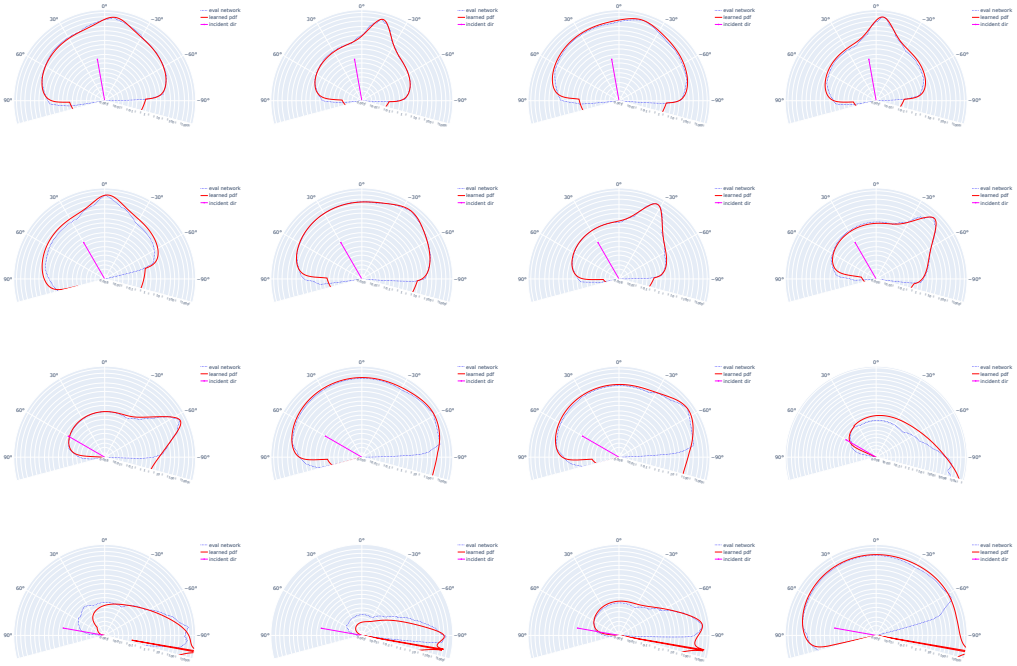

Fig. 31. Cheese slicer handle: importance sampling network plots.

Inkwell metal body (3 layers with 64 neurons)

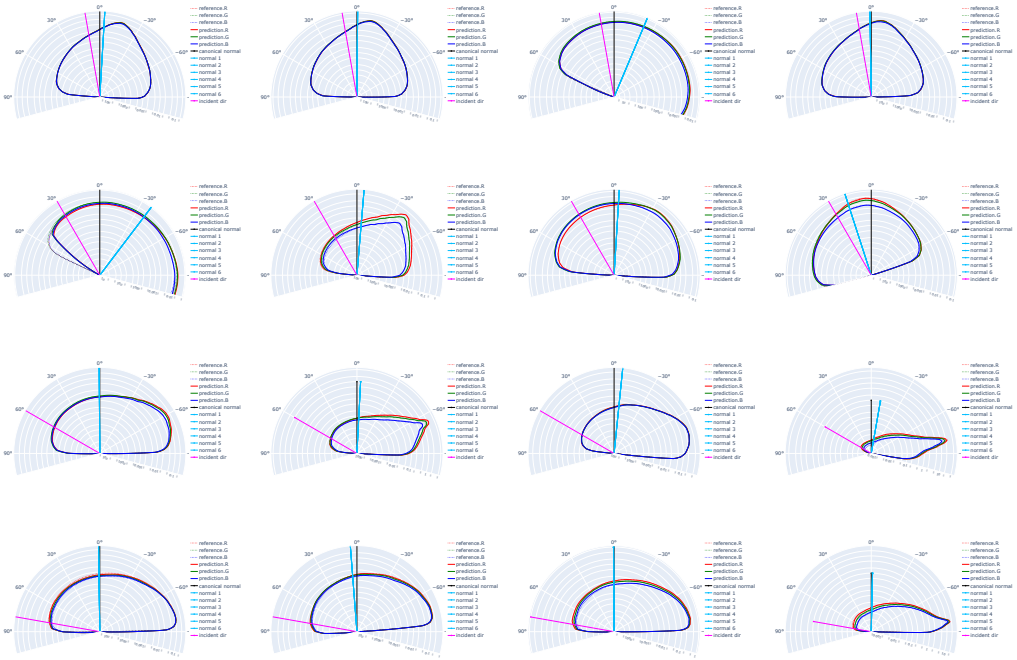

Fig. 32. Inkwell metal body: evaluation network (3 layers with 64 neurons) plots.

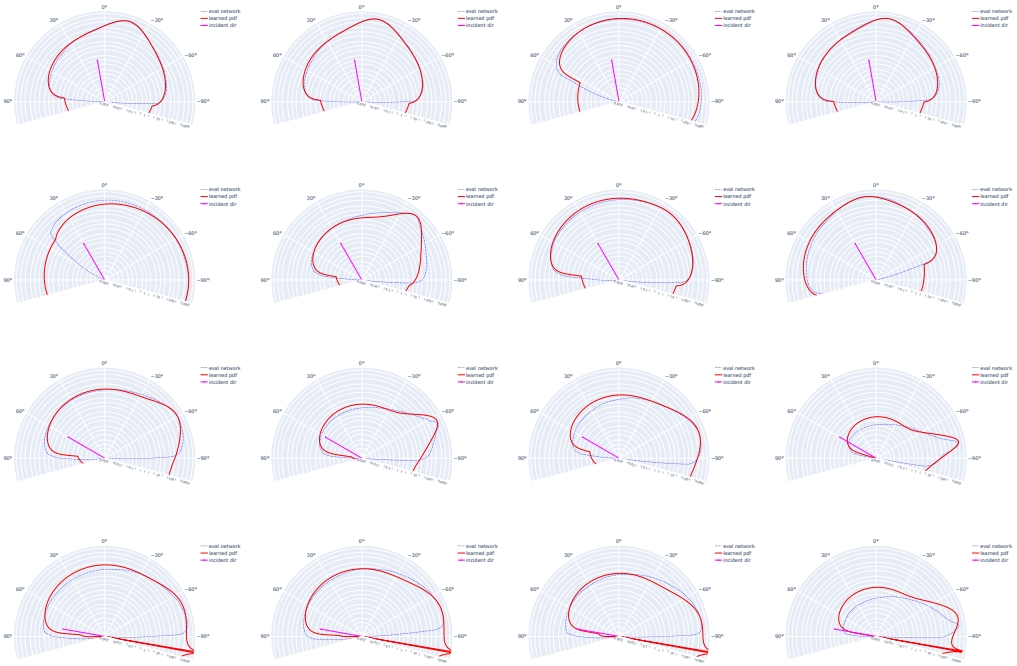

Fig. 33. Inkwell metal body: importance sampling network plots.

Inkwell metal body (2 layers with 32 neurons)

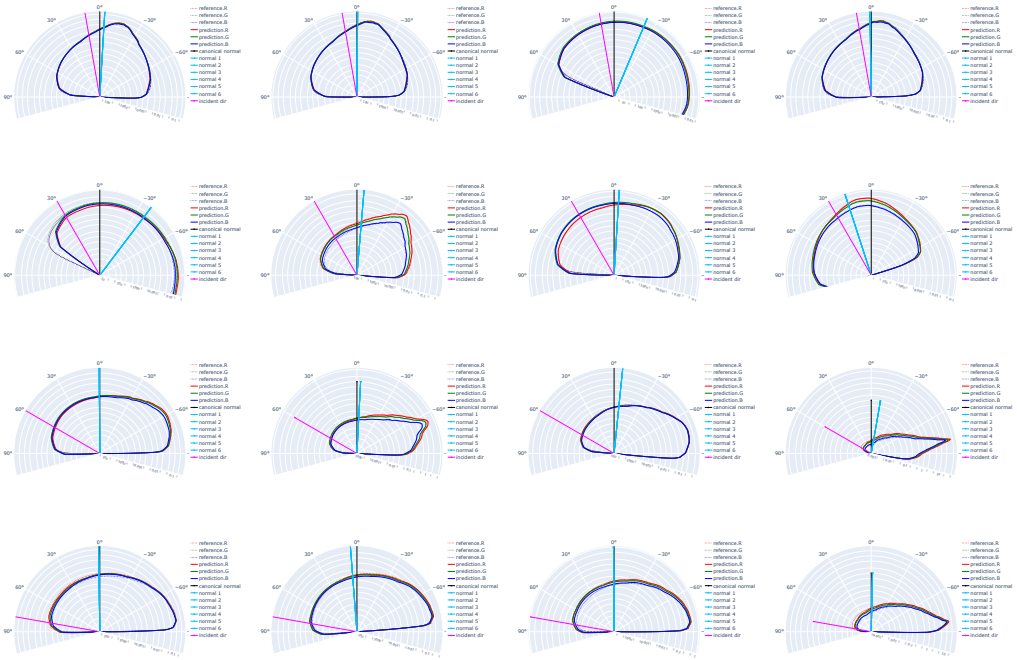

Fig. 34. Inkwell metal body: evaluation network (2 layers with 32 neurons) plots.

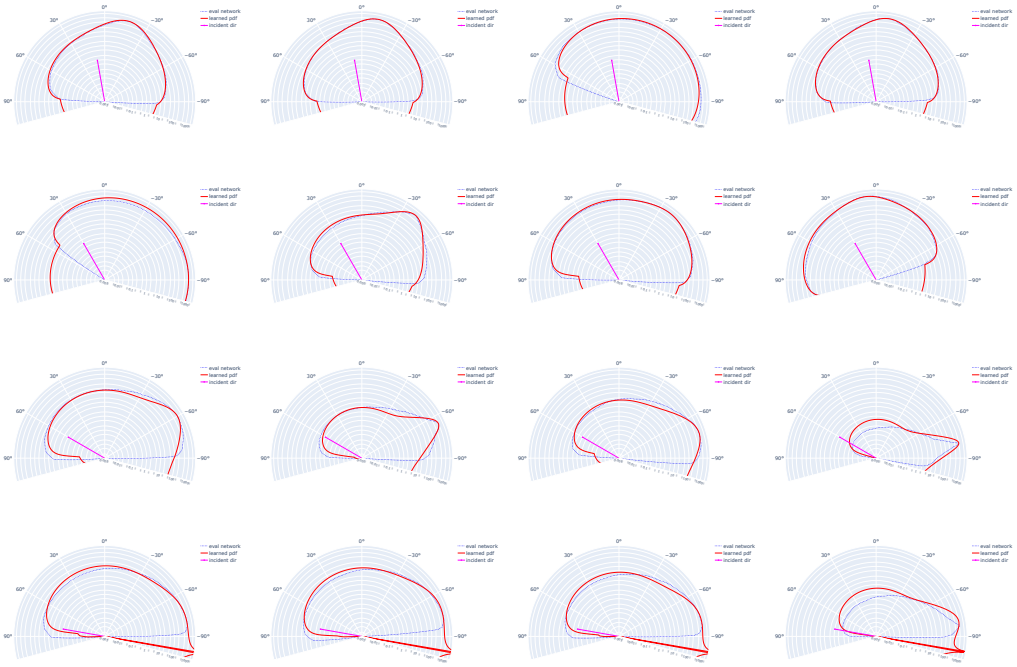

Fig. 35. Inkwell metal body: importance sampling network plots.

Inkwell metal body (2 layers with 16 neurons)

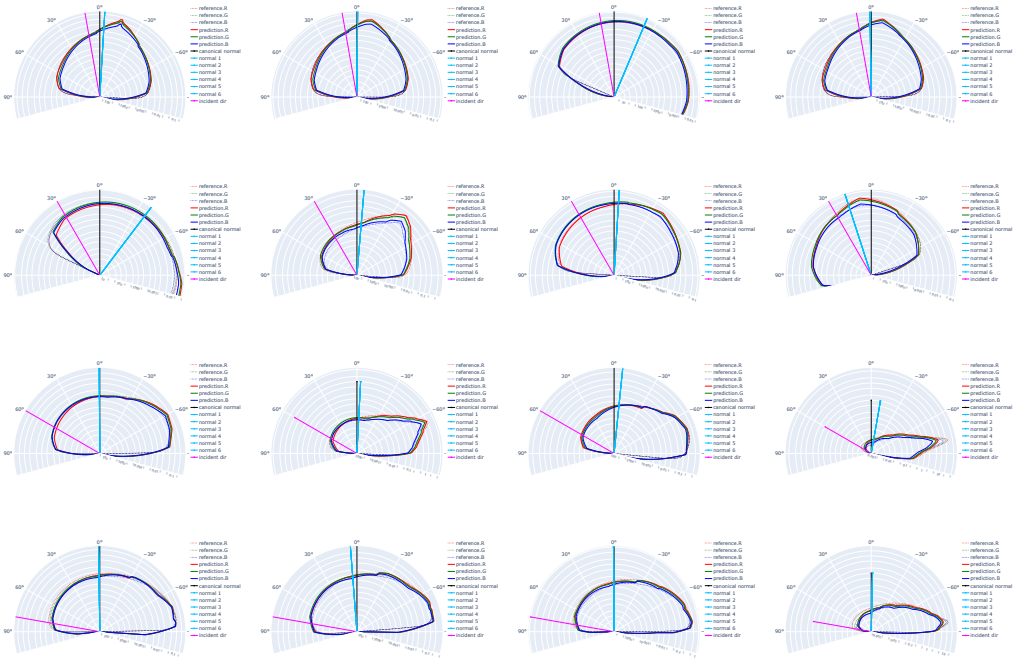

Fig. 36. Inkwell metal body: evaluation network (2 layers with 16 neurons) plots.

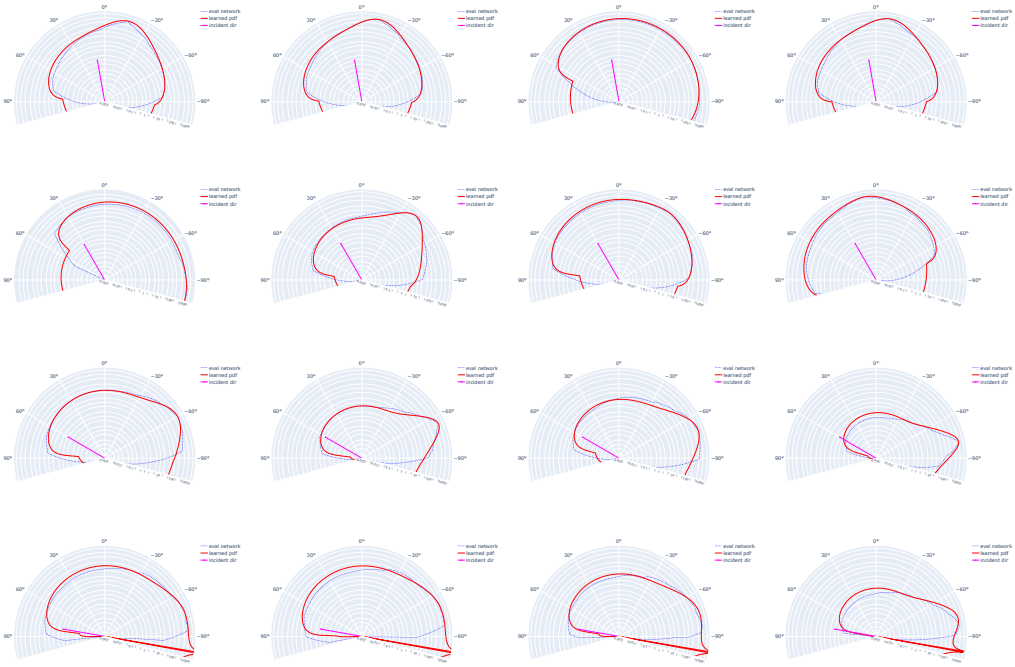

Fig. 37. Inkwell metal body: importance sampling network plots.

REFERENCES

Yong He, Kayvon Fatahalian, and Theresa Foley. 2018. Slang: Language Mechanisms for Extensible Real-time Shading Systems. *ACM Transactions on Graphics* 37, 4, Article 141 (Jul 2018), 13 pages. https://doi.org/10.1145/3197517.3201380

Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (12 2014).

Wojciech Matusik, Hanspeter Pfister, Matt Brand, and Leonard McMillan. 2003. A data-driven reflectance model. *ACM Transactions on Graphics* 22, 3 (Jul 2003), 759–769. https://doi.org/10.1145/882262.882343

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf